How to rasterize a triangle!

Edge functions and interpolation

Tomas Akenine-Möller

Department of Computer Science

Lund University

Misc stuff

- Exciting stuff:
  - At E3: NOKIA announced that they will release phone with graphics hardware
- Assignment 1 due Tuesday next week
- Assignment 2 should be released today or possibly on Monday...

- Do you check out the Online Discussion forum? You should...

- The following lectures will be about graphics hardware. I've put together some notes (~100 pages) that will be available for free on the course website

Overview of today's lecture

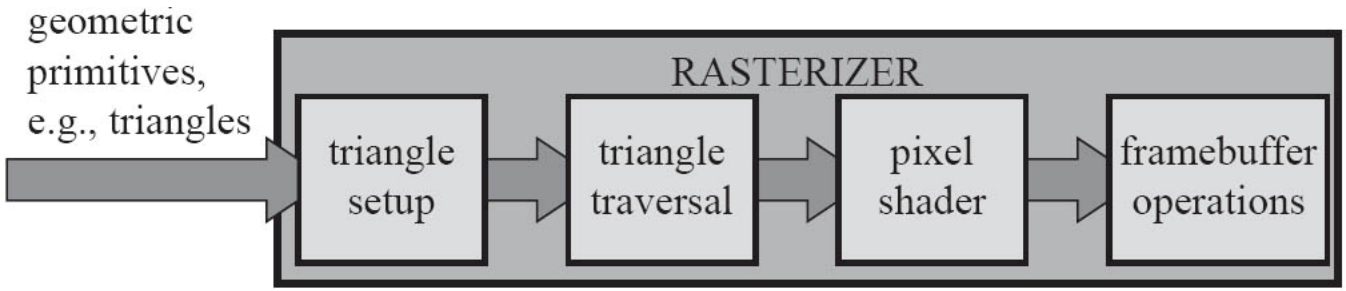
- How graphics hardware can draw a triangle:
  - Edge functions
  - Triangle traversal
  - Interpolation

- Why do I need to know about this?
  - Before you can learn how to walk, you need to know...
  - If you want to design your own hardware
  - If you design new hardware algorithms
  - If you work at company XXX and need to buy graphics hardware for that company's mobile phones, for example

The pipeline is long, why rasterization?

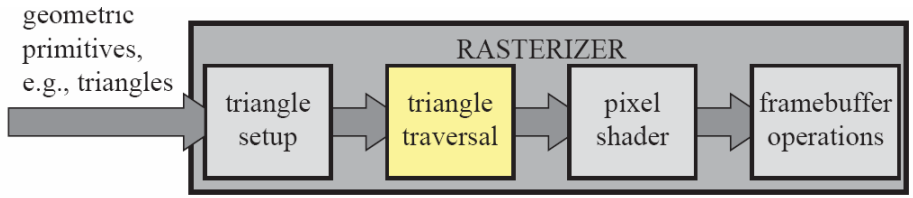
- For any computer system, memory accesses are the operation that uses most energy!
- For mobile devices: consume as little energy as possible!
- Fact: pipeline is roughly two parts:
  - GEOMETRY (per-vertex)
  - RASTERIZER (per-pixel)
- Fact: RASTERIZER accesses memory much more than GEOMETRY
- **Conclusion**: focus on making RASTERIZER as efficient as possible

Overview of rasterizer

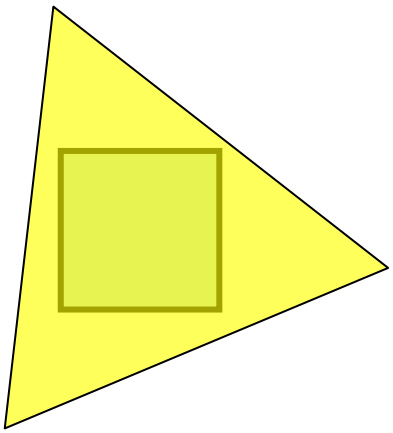


- Triangle setup
  - Put everything we can factor out from the rest of the pipeline, i.e., all ”per-triangle” computations
- Triangle traversal: find pixels (or samples) that are inside the triangle
- Pixel shader: apply per-pixel computations, texture accesses, depth buffer testing, stencil, etc
- Frame buffer operations: write to buffers, e.g.

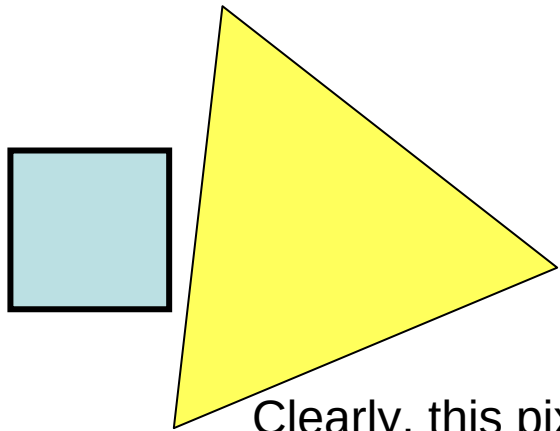
Let's study triangle traversal



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

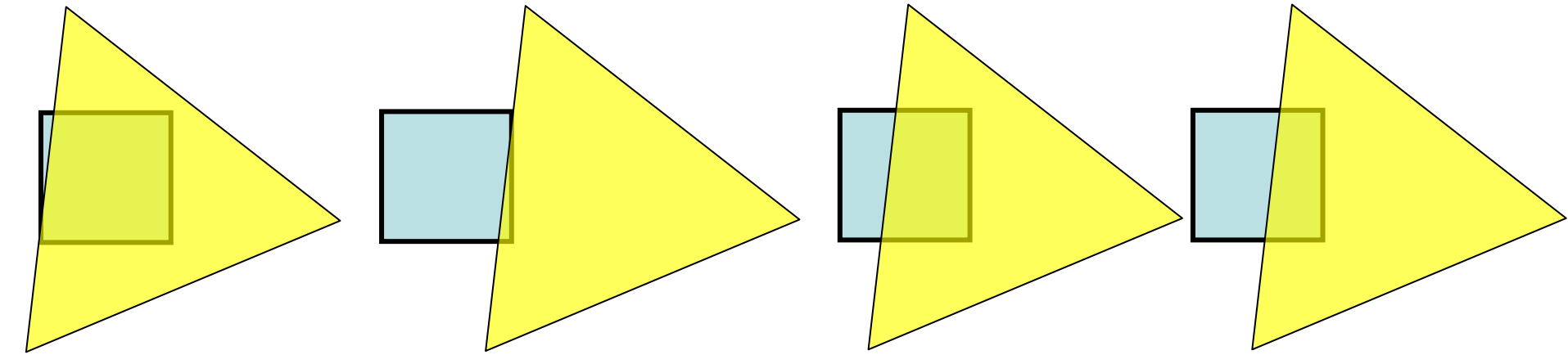


Clearly, this pixel belongs to the triangle

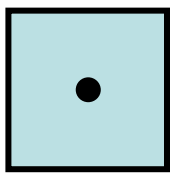


Clearly, this pixel does NOT belong to the triangle

When does a pixel belong to a triangle?



- It all depends on where you *sample*!
- For (low-quality) normal rasterizaion, you sample in the center of the triangle:



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

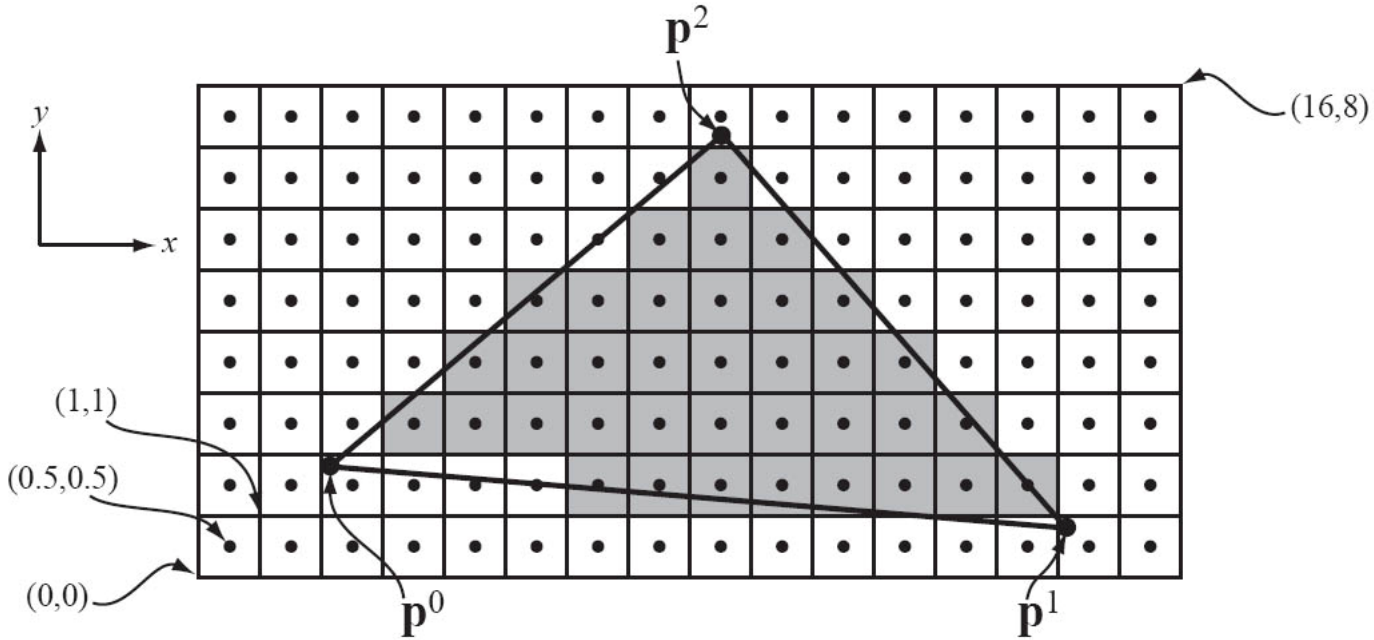
Which coordinate system? Where are we?

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

$\mathbf{M}$ is a projection matrix

- After projection matrix, and homogenization, we have $\mathbf{p}=(p_x,p_y,p_z,1)^T$
- (p_x,p_y) are screen-space coordinates.
- Those are used for triangle traversal
- We assume that triangle have been clipped to screen-space, that is, (p_x,p_y) are in [0,w] x [0,h] (and w x h is screen resolution)

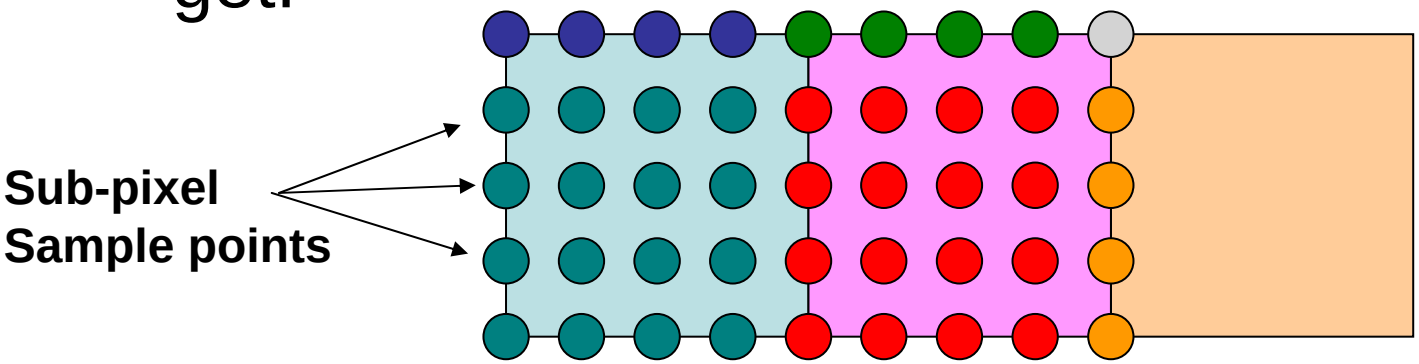
Screen-space coordinates



- This is according to the specification of OpenGL

After projection: sub-pixel coords

- Projected points are floating-point
- Due to limited range of the coordinates, we can use fixed point math (integer)
- However, we cannot round off to nearest pixel center
- Instead use *sub-pixel* coordinates
- With 2 subpixel fractional bits per x, and y, we get:



Floating point coordinates snaps to the closest sub-pixels sample

Remember: integer coords at pixel corners!

What do you get if you do not have sub-pixel coordinates...

- ...and instead only round off to the center of each pixel?
- Bad quality!

- Imagine a slowly translating triangle
- If no sub-pixel coords: triangle vertices will jump abruptly from pixel center to the next
- With sub-pixel coords: much smoother!

Edge functions

- Hardware uses these to find pixels inside a triangle
- For each edge of the triangle, create a line equation (implicit form): $ax + by + c = 0$
- Edge function for two points $\mathbf{p}^0$ and $\mathbf{p}^1$:

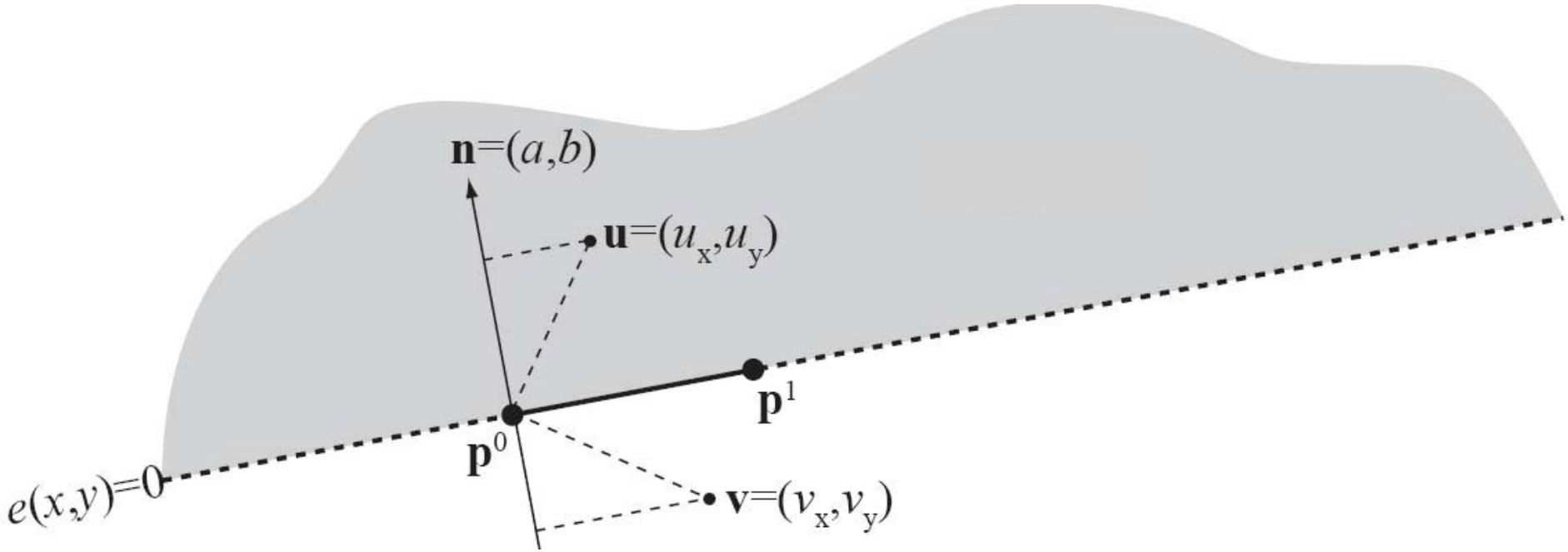
$$e(x,y) = -(p_y^1 - p_y^0)(x - p_x^0) + (p_x^1 - p_x^0)(y - p_y^0)$$

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

Can be interpreted as the ”normal” of the line

- Clearly $e(x,y)=0$, if (x,y) is exactly **on** the (infinite) line through $\mathbf{p}^1$ and $\mathbf{p}^0$
- How about other points?

Intuiton about edge functions

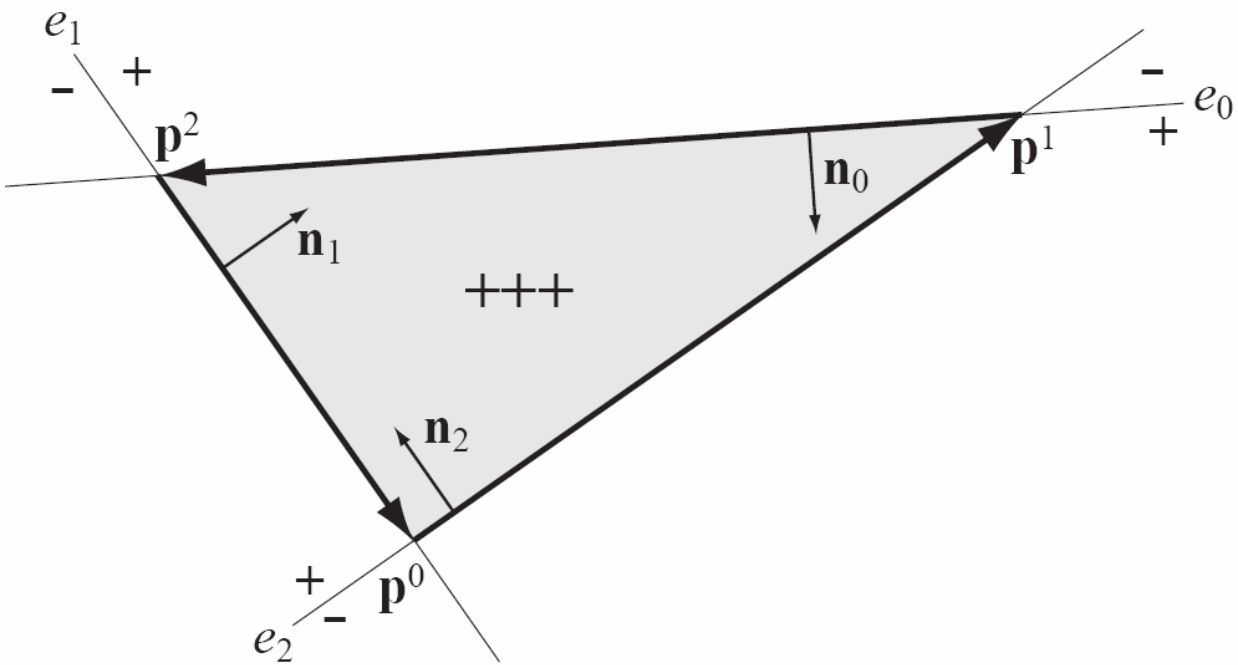


- $e(\mathbf{u})$ must be >0 or <0? Why?
- $e(\mathbf{v})$ must be >0 or <0? Why?

Recall this one:

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

Three edge funcs per triangle

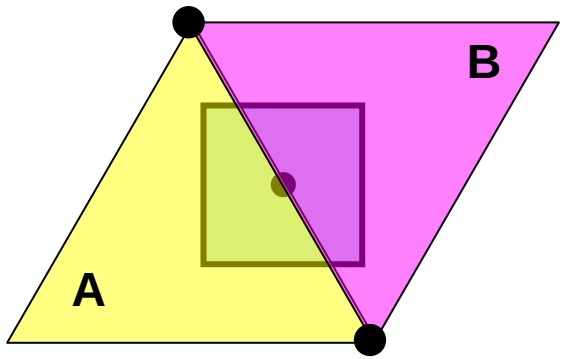


$$
\begin{aligned}
e_0(x,y) &= -(p_y^2 - p_y^1)(x - p_x^1) + (p_x^2 - p_x^1)(y - p_y^1) = a_0 x + b_0 y + c_0 \\
e_1(x,y) &= -(p_y^0 - p_y^2)(x - p_x^2) + (p_x^0 - p_x^2)(y - p_y^2) = a_1 x + b_1 y + c_1 \\
e_2(x,y) &= -(p_y^1 - p_y^0)(x - p_x^0) + (p_x^1 - p_x^0)(y - p_y^0) = a_2 x + b_2 y + c_2
\end{aligned}
$$

- A sample point (x,y) is inside the triangle if $e_i(x,y) >= 0$ for $y = 0,1,2$

Not really that simple, though :-(

- What happens to pixels exactly on an edge?



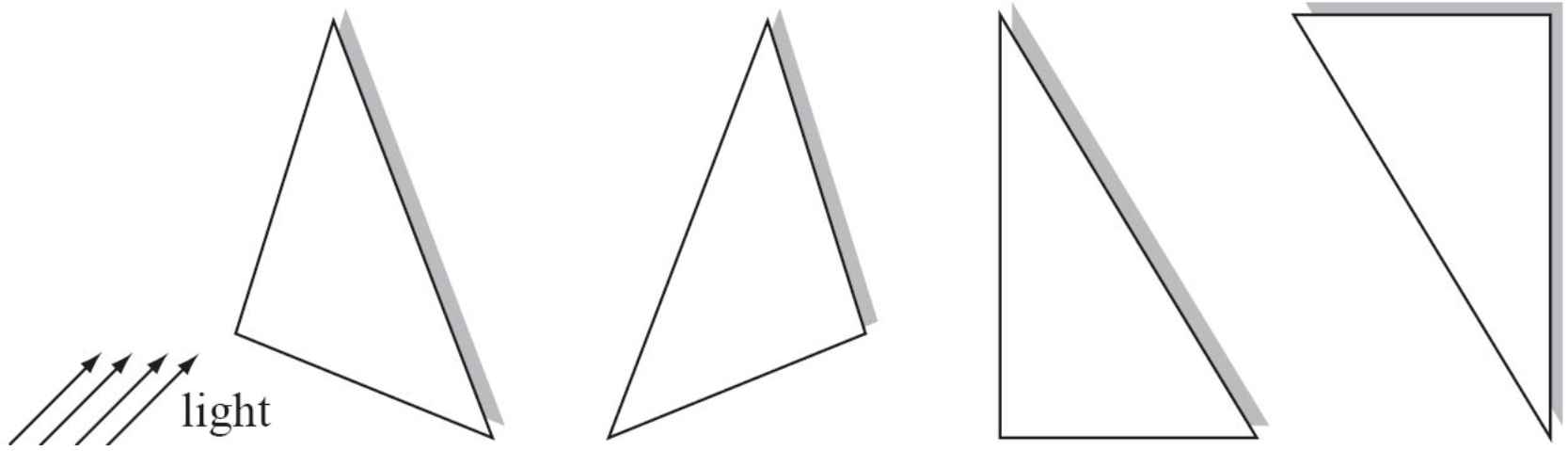
Does the pixel belong to A or B, or both or neither of them?

- Correct answer: one and only one of A or B
- Why?
  - Neither → cracks between triangles
  - Both → unnecessary work is done, transparency and shadow volumes (e.g.) give incorrect results

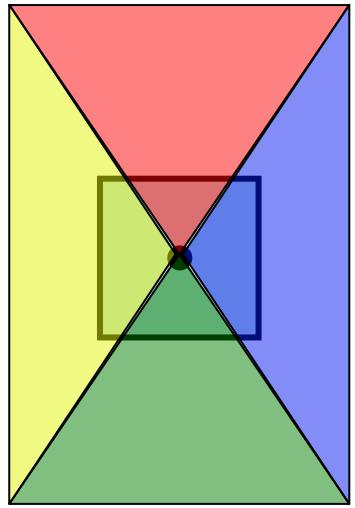
A solution (by McCool et al)

```
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;
```

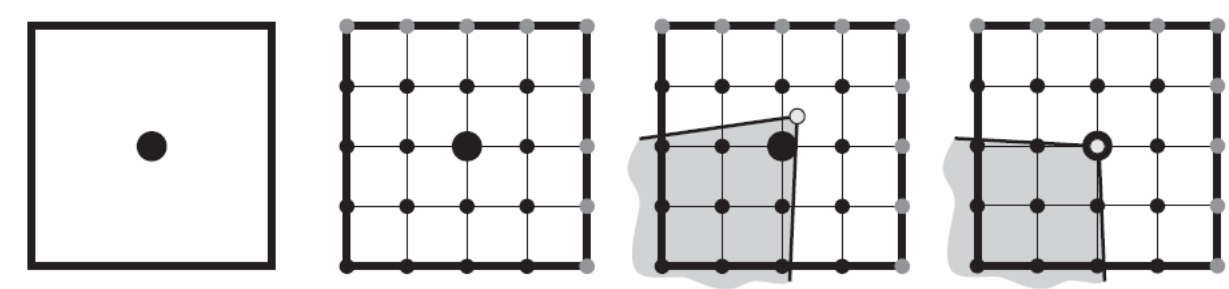
- Another way to think about it:
  - We exclude shadowed edges



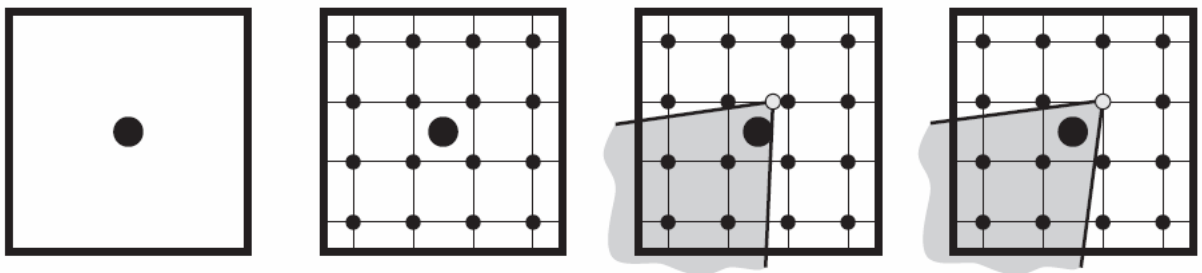
How about when a vertex coincides with the sampling point?



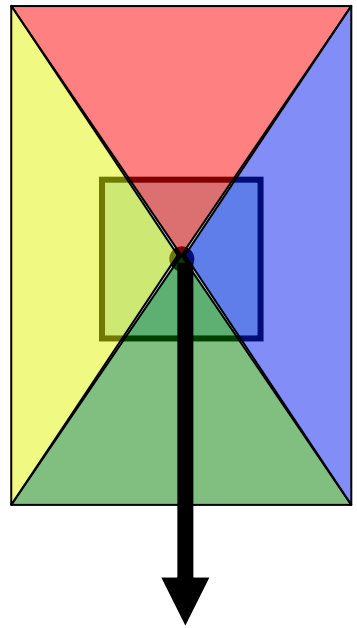
- You get the same kind of problems!



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



Another solution



- Don’t change the sub-pixel grid, allow vertices to coincide with sample point
- So, now, two edge functions will be zero for this pixel: $e_1(x,y)=0$ and $e_2(x,y)=0$
- How would you do it?
- 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...

Incremental updates

- Reduce operations when moving from one pixel to the next

$$e(x+1,y) = a(x+1) + by + c = e(x,y) + a$$

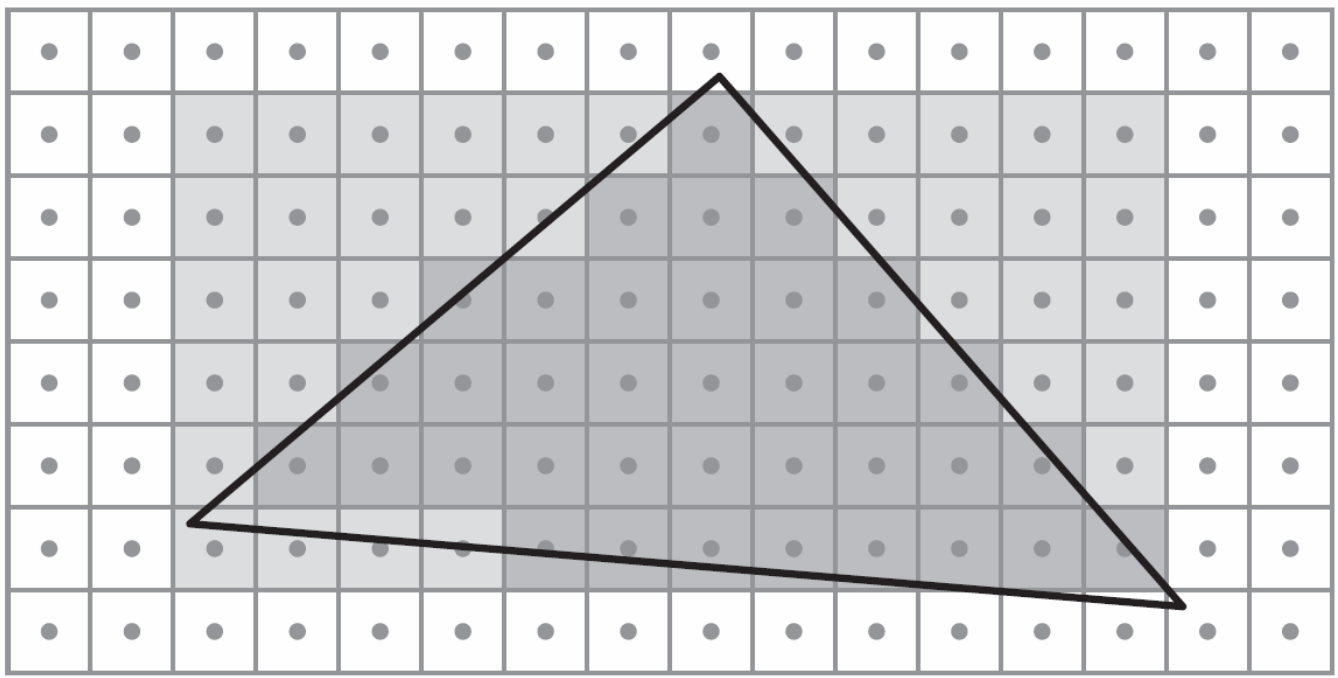
- Similar for the other neighbors, and in general:

$$e(\mathbf{s}+\mathbf{t}) = e(\mathbf{s}) + \mathbf{n}\cdot\mathbf{t}$$

- Finally, we have the tool (edge functions) needed for triangle traversal!

Triangle traversal strategies

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

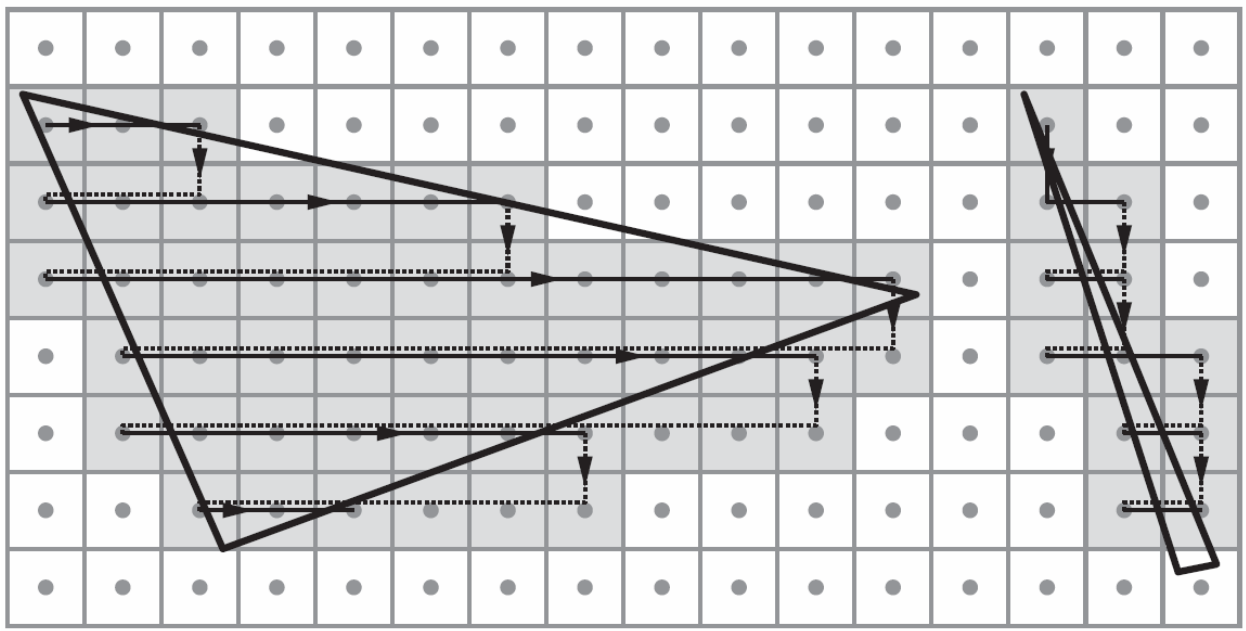


Visits all light and dark gray pixels

But only dark gray pixels are inside, and sent down the pipeline for further processing

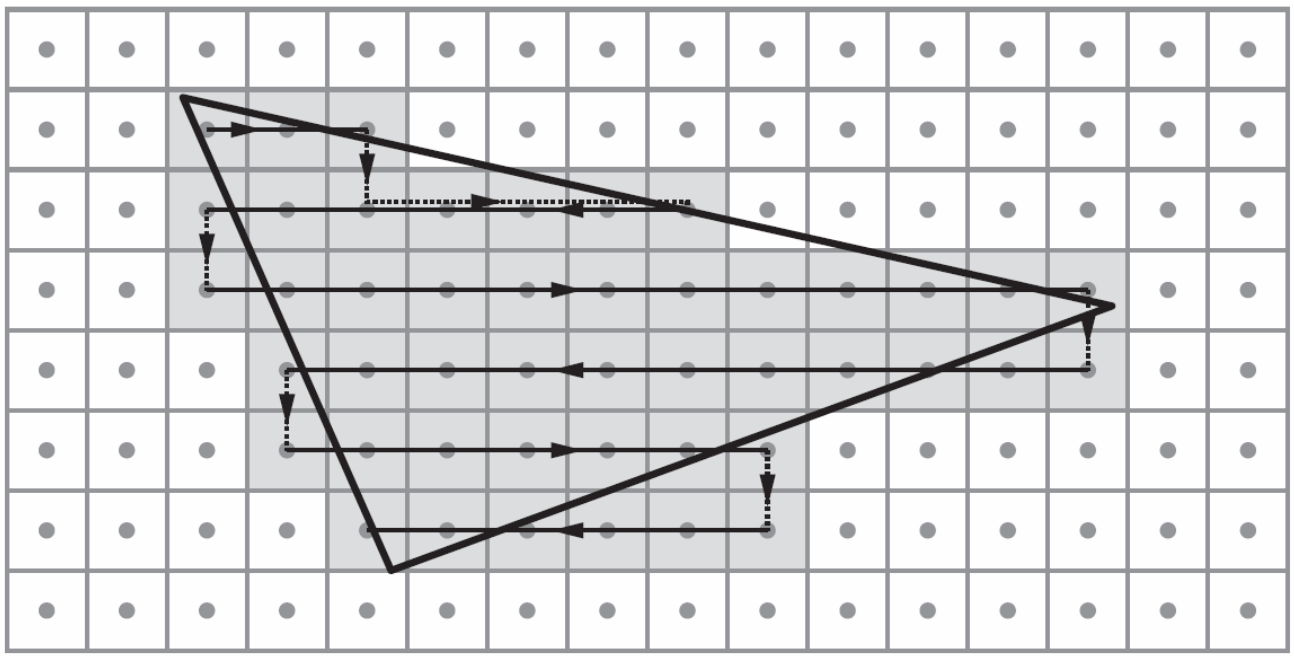
Backtrack traversal

- 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)

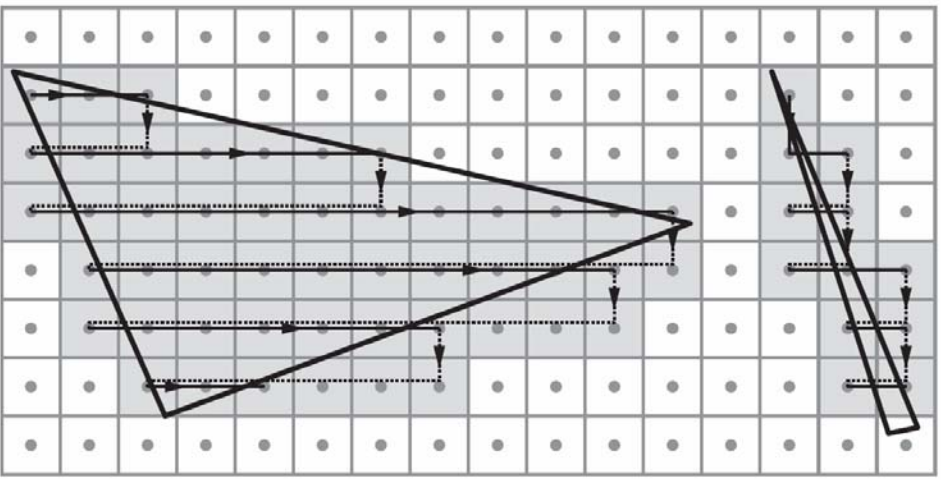


Zigzag traversal

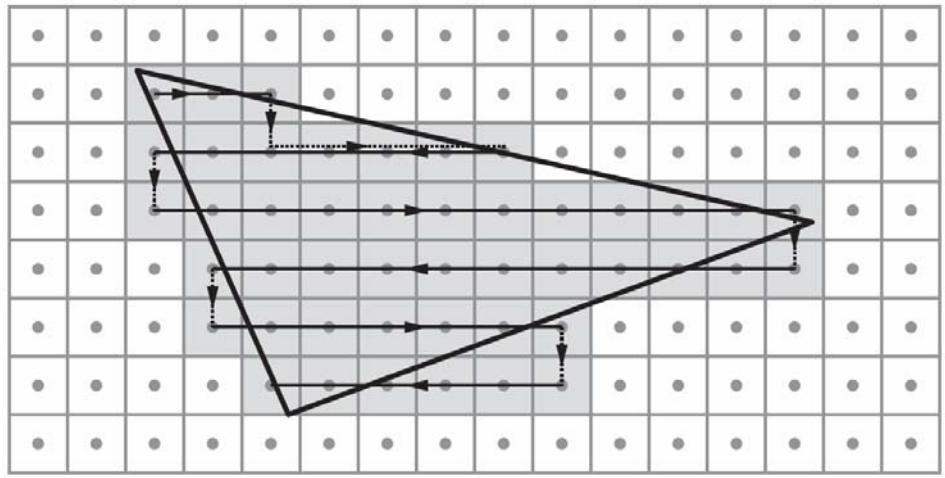
- Simple technique that avoids backtracking
  - Otherwise, very similar
  - Can still visit a bunch of pixel outside (see next to most bottom scanline)
  - Can be solved, but requires more **`Inside()`**-testing



Side by side comparison Backtrack vs zigzag



Backtrack never visits unnecessary pixels to the left

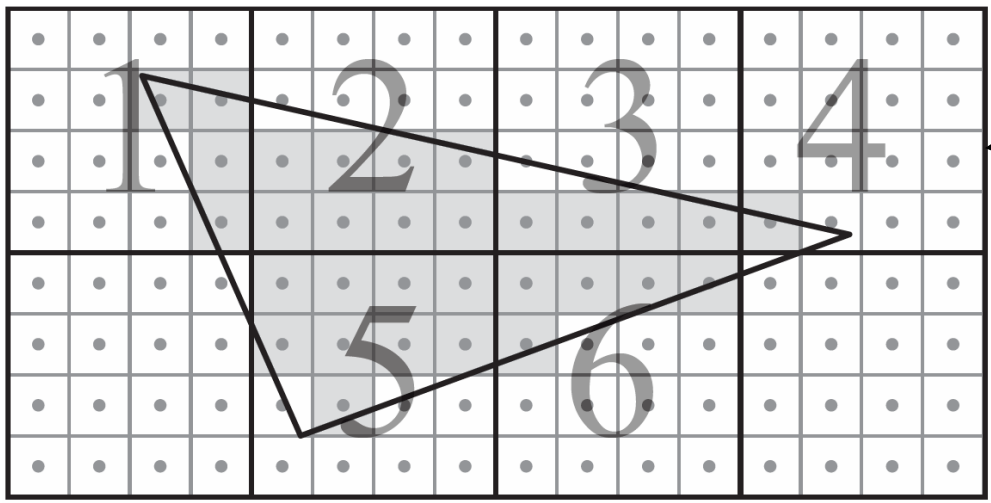


Zigzag never visits unnecessary pixels to the left on even scanlines and to the right on odd scanlines
(and avoids backtracking)

We only cover simple schemes here. There are more sophisticated schemes that visits fewer pixels, but those are more expensive

Tiled traversal

- General idea: divide screen space into non-overlapping tiles (a tile is wxh pixels)
  - Traverse one tile at a time, and finish visiting pixels in tile before moving to next tile



8x8 tile size is common in desktop graphics cards

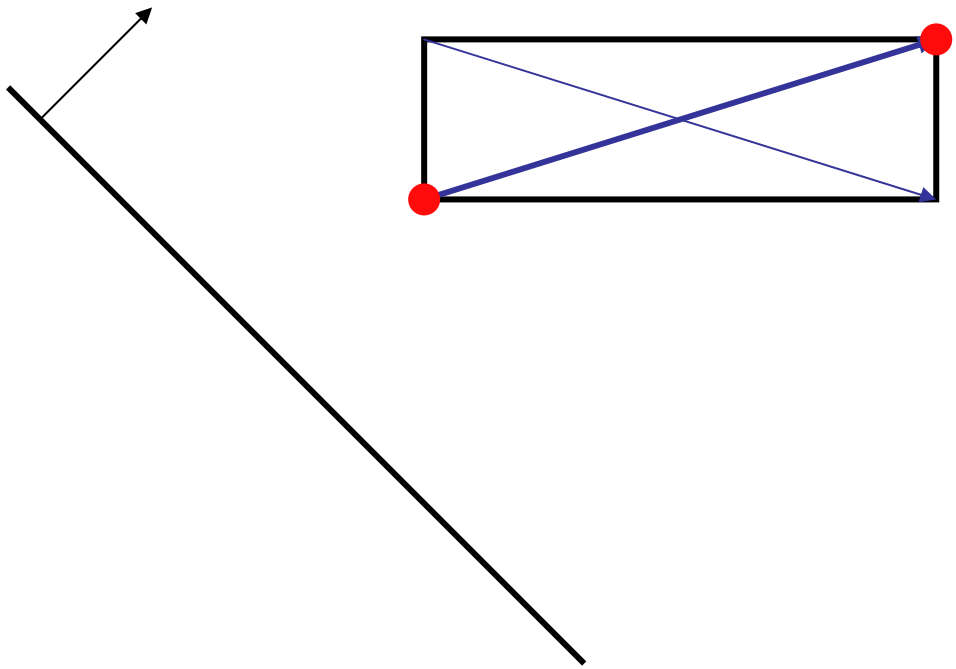
- Better because (all topics will be treated in later lextures):
  - Gives better texture cache performance
  - Enables simple culling (Zmin & Zmax)
  - Real-time buffer compression (color and depth)

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!
  - See next slides...

Tile/triangle overlap test (1)

- Reuse knowledge from 3D intersection testing—Haines and Wallace's trick:

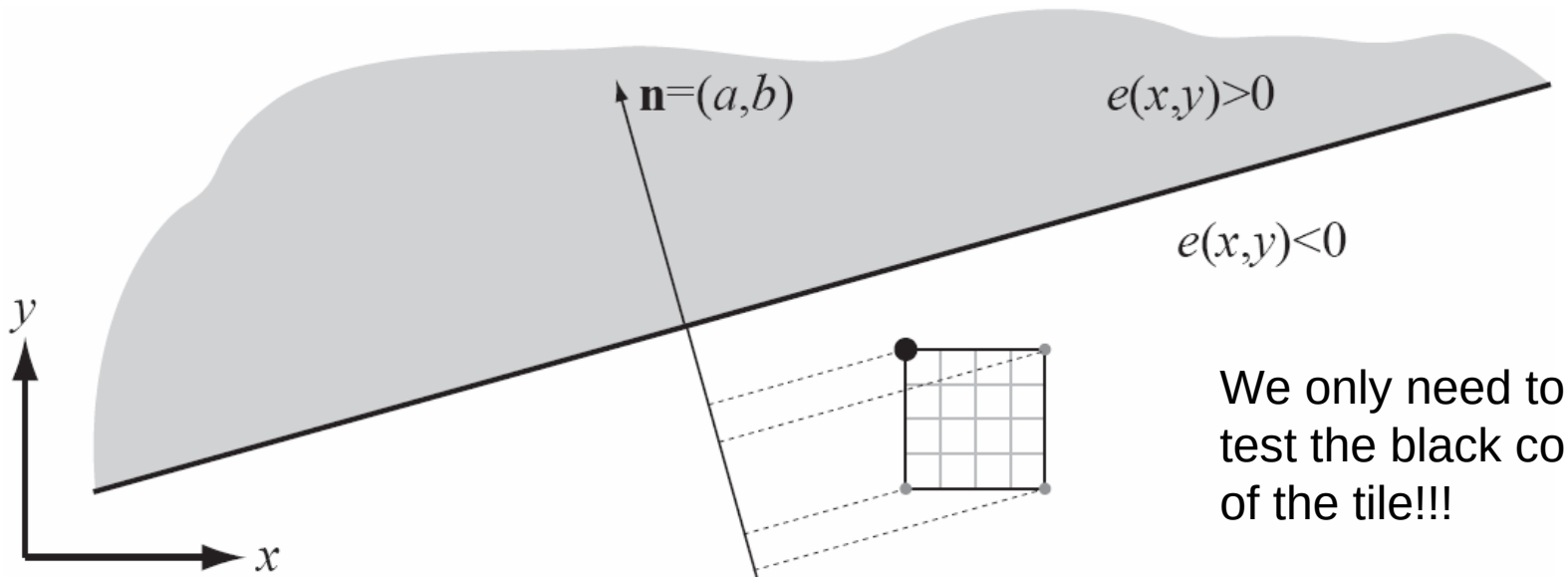


Only need to test the corners whose diagonal aligns best with the normal of the plane

Hoff (1996) has another trick: if you want to find out whether the box is completely inside, then you only need to test one corner!
This is a dramatic reduction from testing all 8 corners of the box

Tile/triangle overlap test (2)

- A tile is fully **outside** a triangle if:
  - Either the tile is outside the bounding box of the triangle (trivial)
  - Or if tile is fully outside at least one edge func (concentrate on this case)



We only need to test the black corner of the tile!!!

How can we know?

Tile/triangle overlap test (3)

- Assume we have evaluated the edge function for the lower left corner, $\mathbf{s}=(s_x,s_y)$, of the tile, i.e., $e(\mathbf{s})$
- We can evaluate the tile corners with:

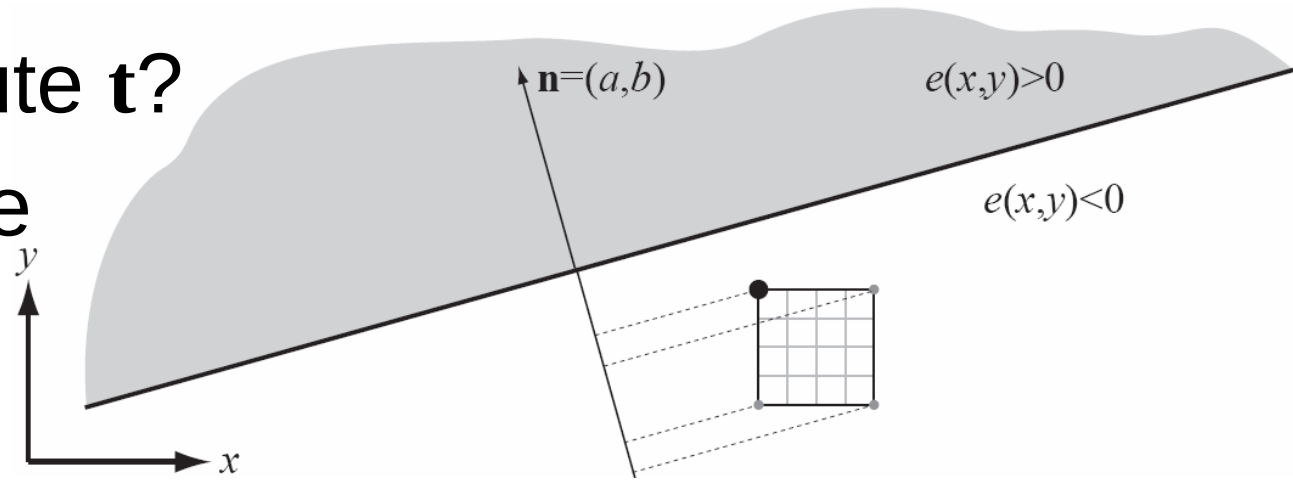
$$e(\mathbf{s}+\mathbf{t}) = e(\mathbf{s}) + \mathbf{n}\cdot\mathbf{t}$$

- Where $\mathbf{t}=(0,0)$, or $\mathbf{t}=(w,0)$, or $\mathbf{t}=(0,h)$, or $\mathbf{t}=(w,h)$
  - Since $\mathbf{n}=(a,b)$ is known from the edge function, and w & h in all realistic scenarios are powers of two (e.g., 2^g)...
  - ...this reduces to adding shifted versions of a and b

- How compute $\mathbf{t}$?

Tile/triangle overlap test (4)

- How compute **t**?
- Try example to the right



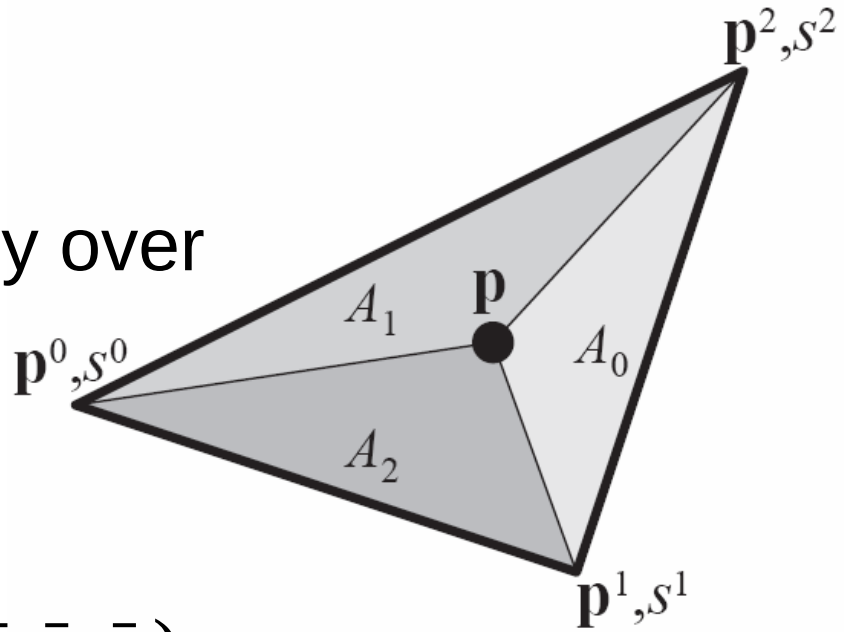
- Look at normal direction $\mathbf{n}=(a,b)=(n_x,n_y)$
- In general:

$$t_x = \begin{cases} w, & n_x \geq 0 \\ 0, & n_x < 0 \end{cases} \qquad t_y = \begin{cases} h, & n_y \geq 0 \\ 0, & n_y < 0 \end{cases}$$

- **t** is computed as part of the ”triangle setup”, since the info is constant all over the triangle [could be used incrementally too]

Interpolation

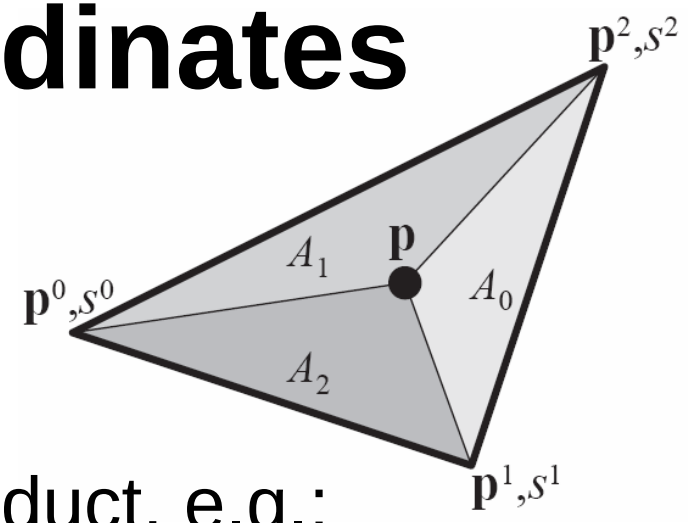
- Now, we can find pixels inside triangles
- Next, we need to interpolate parameters across triangles
- What is s at $\mathbf{p}$?
- $s(x,y)$ should vary smoothly over the triangle!



- Can be done using barycentric coordinates, $(\bar{u},\bar{v},\bar{w})$

Barycentric coordinates

- Are proportional to the signed areas of the subtriangles formed by $\mathbf{p}$ and the vertices



- Area computed using cross product, e.g.:

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

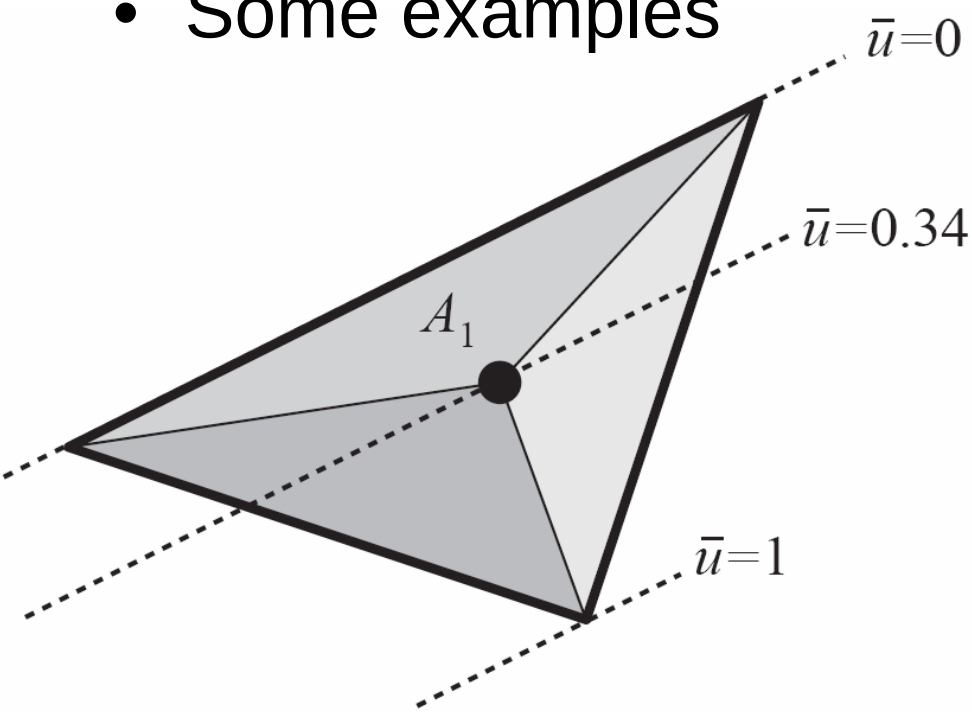
- In graphics, we always use barycentric coordinates normalized with respect to triangle area:

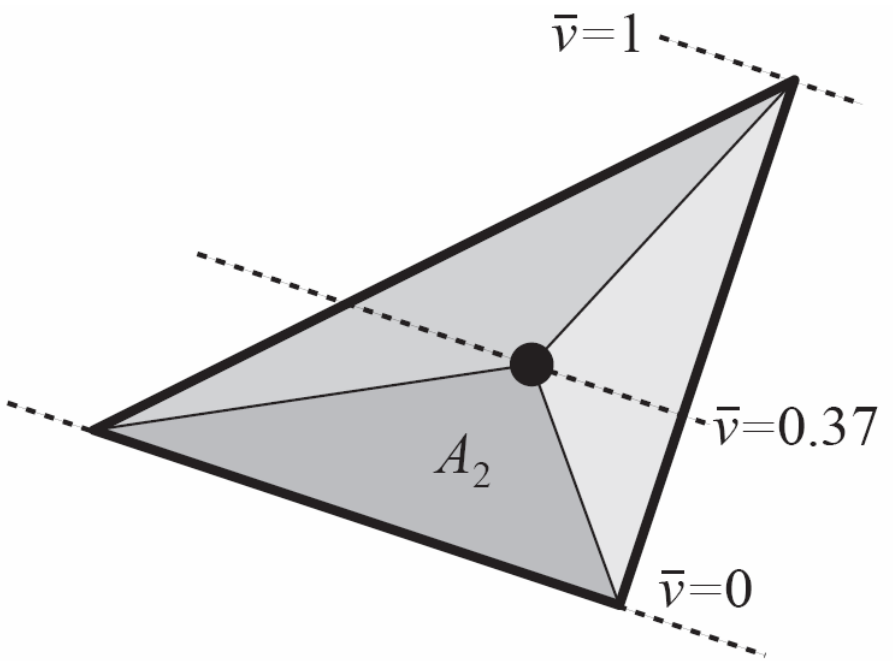
$$(\bar{u}, \bar{v}, \bar{w}) = \frac{(A_1, A_2, A_0)}{A_\Delta} \qquad A_\Delta = A_0 + A_1 + A_2$$

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

What are those barycentric coordinates?

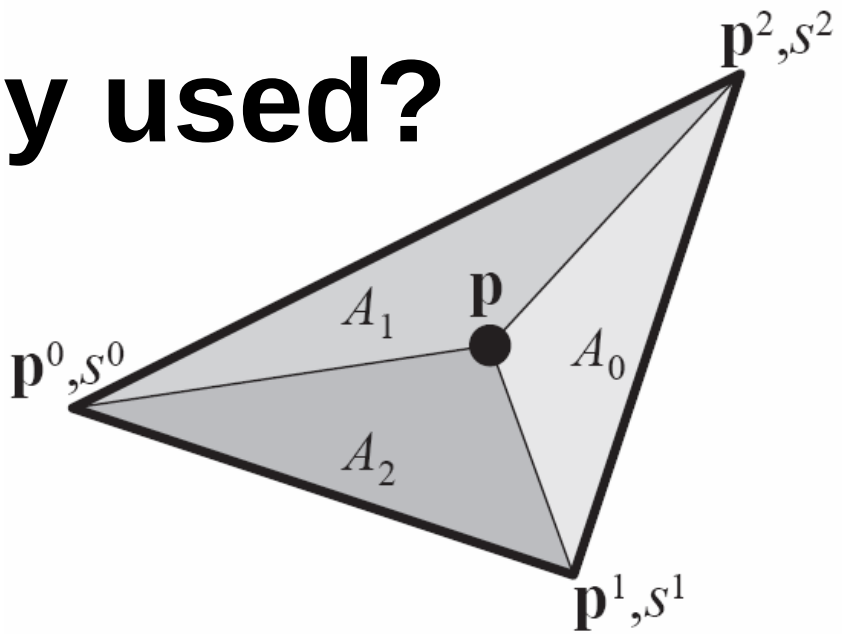
- Some examples





- Why are they constant on lines parallel to an edge?
- Because height of subtriangle is constant!

How are they used?



- 1: Compute barycentric coordinates for a pixel
- 2: Interpolate vertex parameters, s^0, s^1, s^2:

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

- Depth $d=p_z= h_z/h_w$ should be interpolated like this

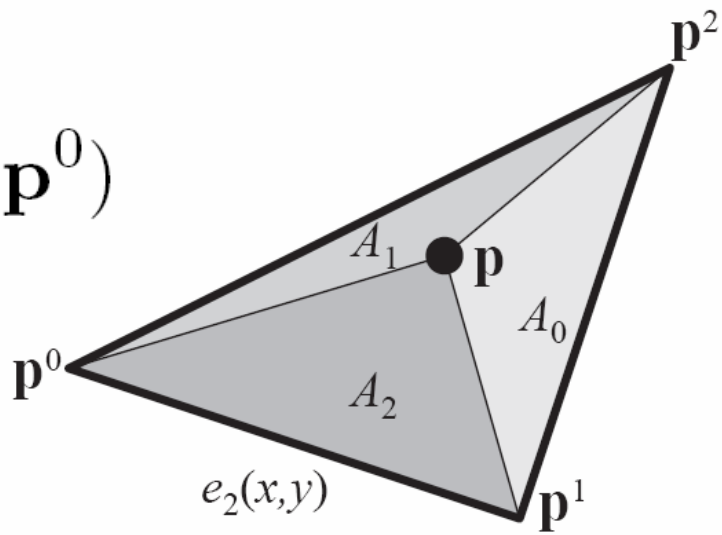
- Note also:
  - And they are negative outside the triangle,
  - Or >1

$$\begin{aligned} \bar{u} &\geq 0, \\ \bar{v} &\geq 0, \text{ and} \\ \bar{w} &\geq 0 \Leftrightarrow \bar{u} + \bar{v} \leq 1 \end{aligned}$$

Barycentric coordinates from edge functions (1)

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

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

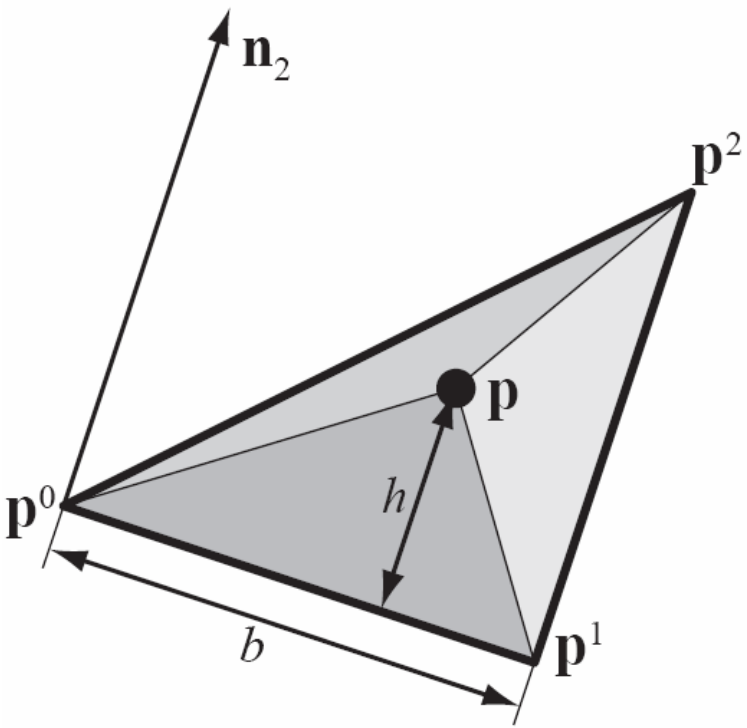


Barycentric coordinates from edge functions (2)

- 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:

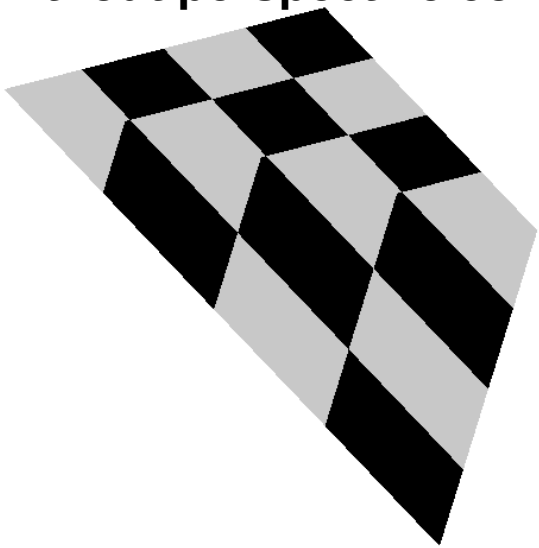
$$\bar{u} = \frac{e_1(x,y)}{2A_\Delta}$$

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

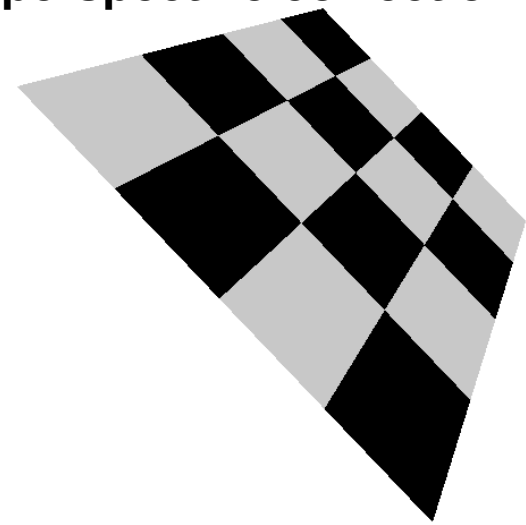
- And $1/(2A_\Delta)$ can be computed in the triangle setup (once per triangle)

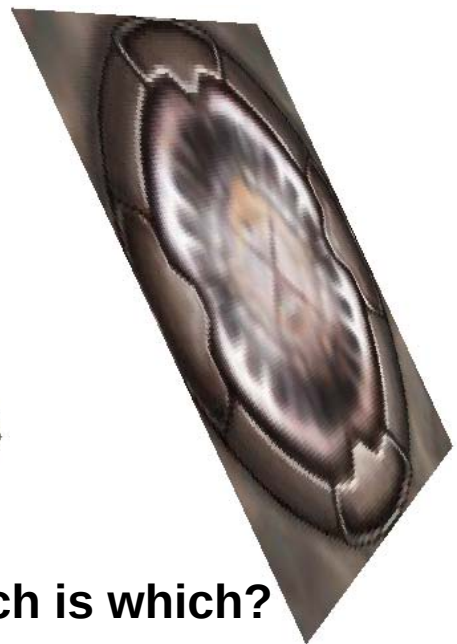
Resulting interpolation

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



With perspective correction



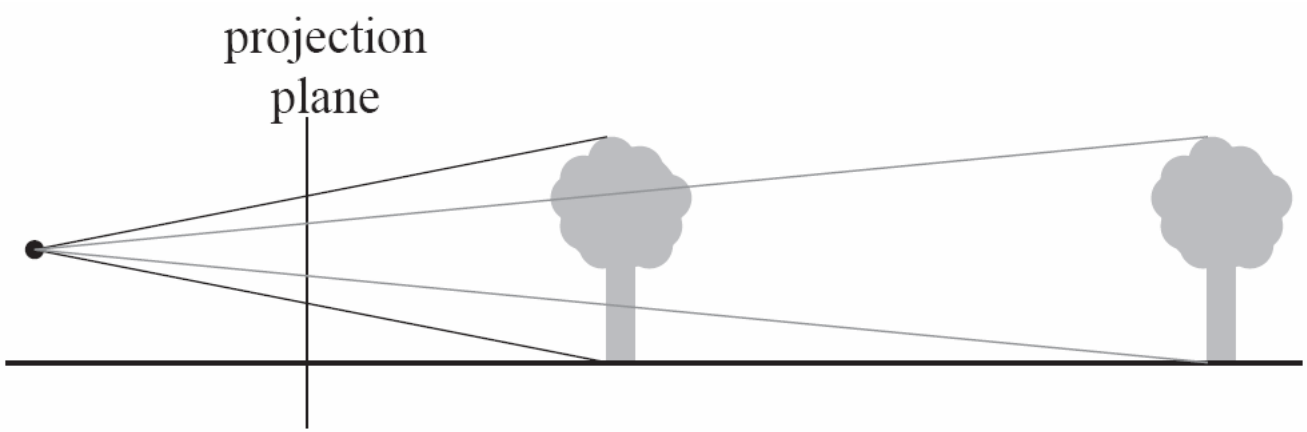


Which is which?

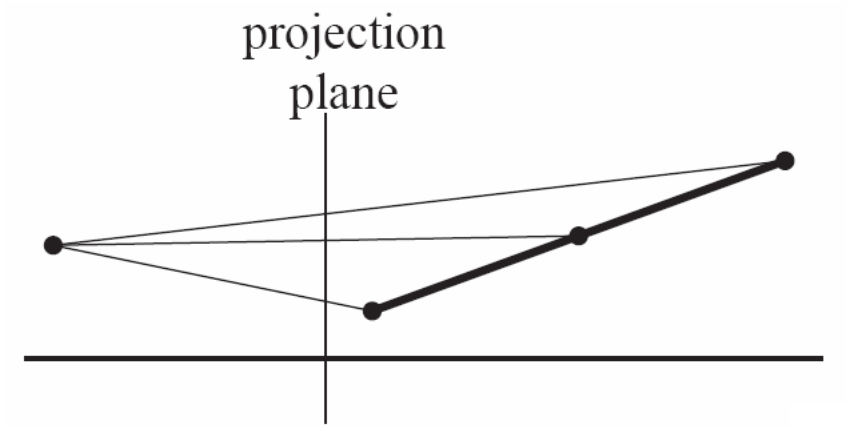
- Looks even worse when animated...
- Clearly, perspective correction is needed!

Perspective-correct interpolation

- Why?
  - Things farther away appear smaller!



- And even inside objects, of course:



Per-pixel division is required for perspective correct interpolation

- Surprisingly difficult to explain...
- Recall perspective projection: $\mathbf{Me=h}$
  - Where $\mathbf{M}$ is projection matrix, $\mathbf{e}$ is coord in eye space, $\mathbf{h}$ is the result (but no homogenization done)
- h_x and h_y are simply scaled and translated eye coordinates in x and y (look at $\mathbf{M}$ to understand this)
- h_w is simply the z-coordinate in eye-space
- Thus (h_x, h_y, h_w) is $\mathbf{e}$ but in a slightly distorted space!

Perspective correct interpolation (2)

- Now, we have parameters, s^i, per vertex that we want to interpolate.
- In distorted eye space, this is straightforward, because we can write the following for each vertex:

$$s^i = kh_x^i + lh_x^i + mh_w^i \qquad i=0,1,2$$

- Three equation, three unknowns (k,l,m) → solvable:

$$s(h_x, h_y, h_w) = kh_x + lh_y + mh_w$$

- We need to express this in screen space: (p_x,p_y)
  - But $(p_x,p_y)=(h_x/h_w, h_y/h_w)$

Perspective correct interpolation (3)

$$s(h_x, h_y, h_w) = kh_x + lh_y + mh_w$$

- Divide equation by h_w ! →

$$s(h_x, h_y, h_w) = kh_x + lh_y + mh_w \Leftrightarrow$$

$$k\frac{h_x}{h_w} + l\frac{h_y}{h_w} + m\frac{h_w}{h_w} = kp_x + lp_y + m = \hat{s}(p_x, p_y)$$

- That is, ***linear*** interpolation in **screen space**!
- But unfortunately, we interpolate s/h_w ☹
- Simple to correct for though:
  - Linearly interpolation of: $1/h_w$ Call func: $\hat{o}(p_x, p_y)$
  - And then…

Perspective correct interpolation (4)

- Correction is done as: $s(p_x, p_y) = \frac{\hat{s}(p_x, p_y)}{\hat{o}(p_x, p_y)}$
- Using sloppy notation (simpler to understand, perhaps): $\frac{s/w}{1/w} = \frac{sw}{w} = s$
- So, to sum up:
  - Linearly interpolate s/w in screen space
  - Linearly interpolate 1/w in screen space
  - Once s/w and 1/w have been computed for a pixel (or sample point), recover s as: (s/w)/(1/w)
  - Linear interpolation can be done with barycentric coordinates: $(\bar{u},\bar{v})$

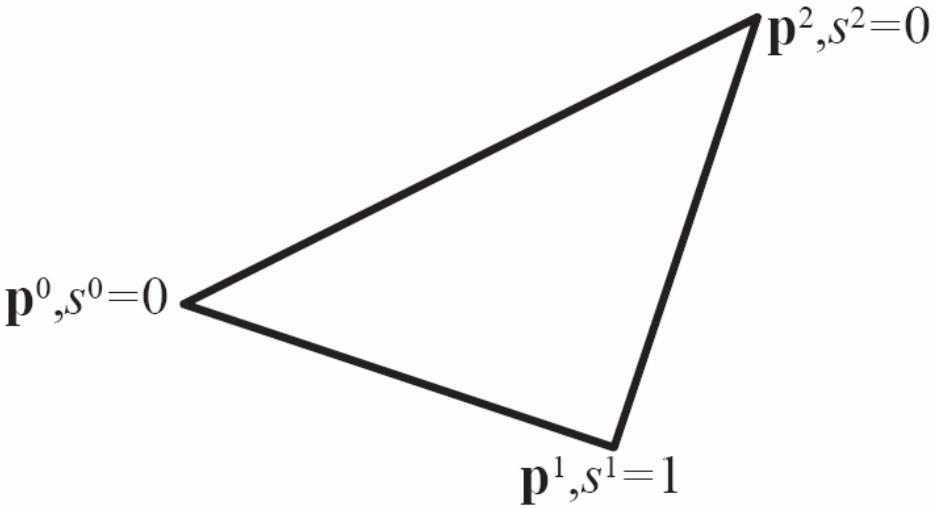
“Perspectively-correct interpolation coordinates” (1)

- Background: many parameters to interpolate (texture coordinates (1-8), fog, color, etc) and expensive to compute parameter/h_w
- Better: compute (u,v) that are similar to barycentric coordinates, $(\bar{u},\bar{v})$ but (u,v) have taken perspective into account
- Advantage: once (u,v) have been computed, we can interpolate all parameters with perspective as:

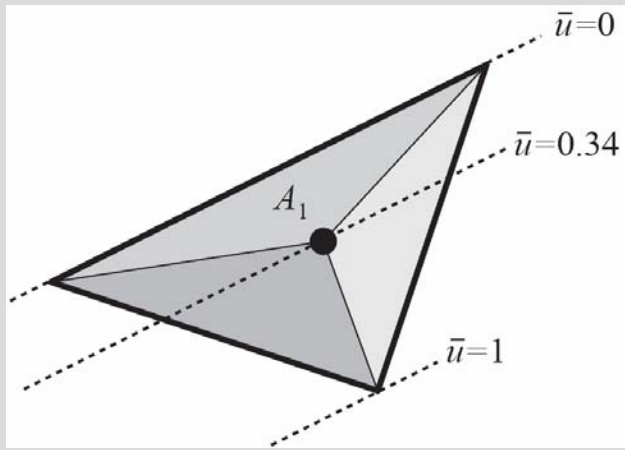
$$s(p_x, p_y) = (1-u-v)s^0 + us^1 + vs^2 = s^0 + u(s^1 - s^0) + v(s^2 - s^0)$$

“Perspectively-correct barycentric coordinates” (2)

- So, how compute (u,v)?
- Imagine, creating these per-vertex parameters, s^i :



You might want to recall this figure



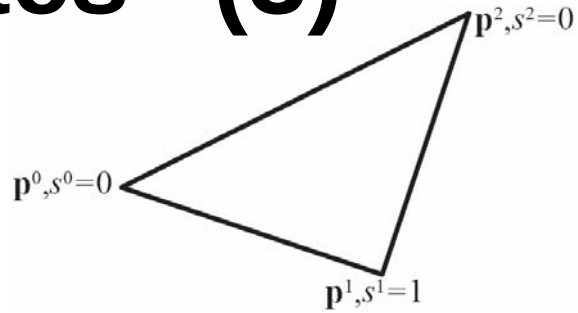
- This means, that if we interpolate s with perspective in mind, we will get u, which is perspectively correct!
- Similar trick can be done for v → (u,v)
- Can all be done with edge functions too!

“Perspectively-correct barycentric coordinates” (3)

- Recall: $u(p_x, p_y) = \dfrac{\hat{s}(p_x, p_y)}{\hat{o}(p_x, p_y)}$

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

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



Recall:

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

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

- Simplify:

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

Very important result

$$u = \frac{f_1}{f_0 + f_1 + f_2}$$

$$v = \frac{f_2}{f_0 + f_1 + f_2}$$

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

Triangle setup vs per-pixel computations

- 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 (only most basic stuff)

	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)

Summary

- Edge functions are very powerful
  - Crackfree rasterization (and no unecessary overdraw)
  - Can evaluate pixels in parallel if needed
  - Can interpolate using them as well
    - Perspective-correct and...
    - ...without perspective
  - Tiled traversal possible too

- Next week, the fun graphics hardware stuff starts!
  - Time to save memory accesses!!
  - And power!

- Before next lecture: read about caches
  - Chapter 5, section 5 in online notes