Lecture 4:

Visibility

(coverage and occlusion using rasterization and the Z-buffer)

Visual Computing Systems
CMU 15-869, Fall 2013

Visibility

- **Very imprecise definition: computing what scene geometry is visible within each screen pixel**
  - **What scene geometry projects into a screen pixel? (screen coverage)**
  - **Which geometry is actually visible from the camera at that pixel? (occlusion)**

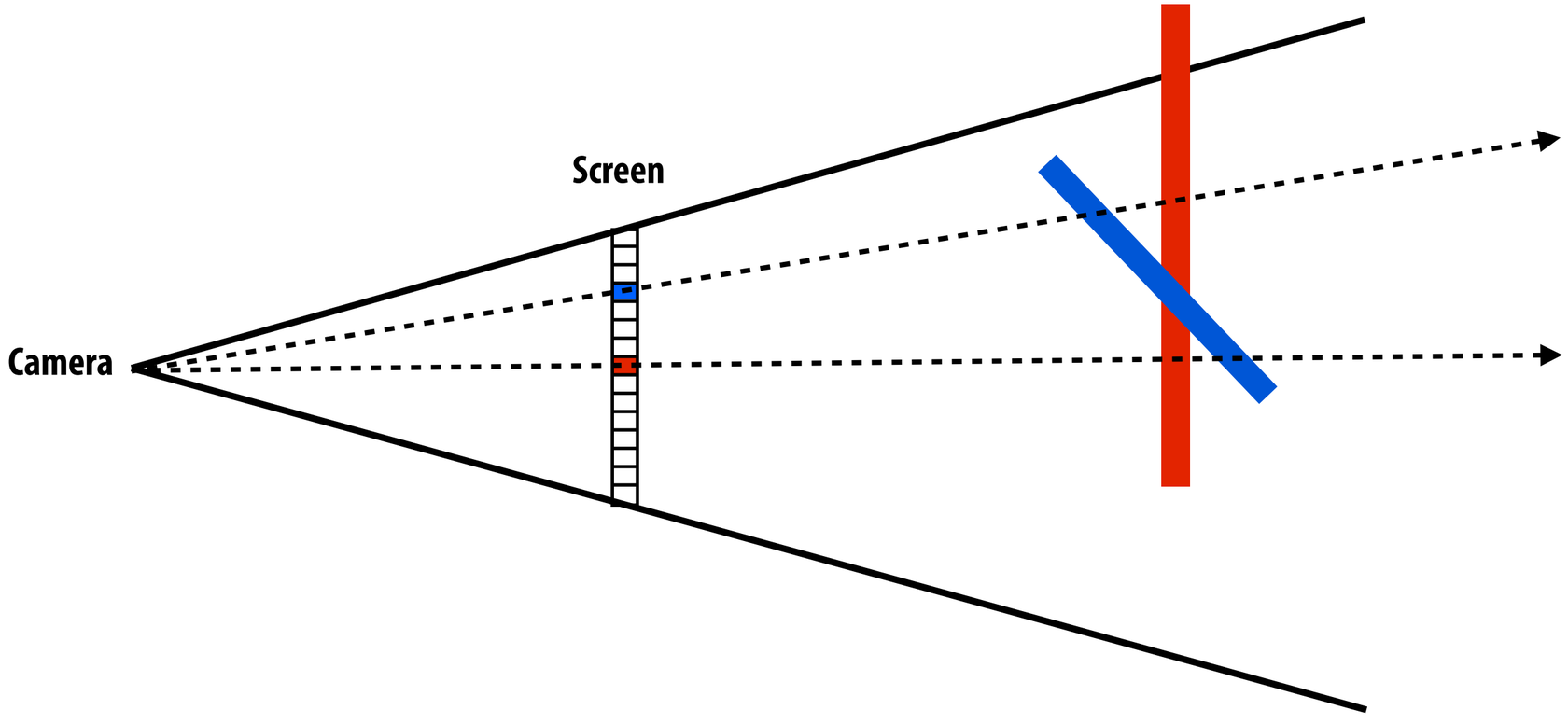
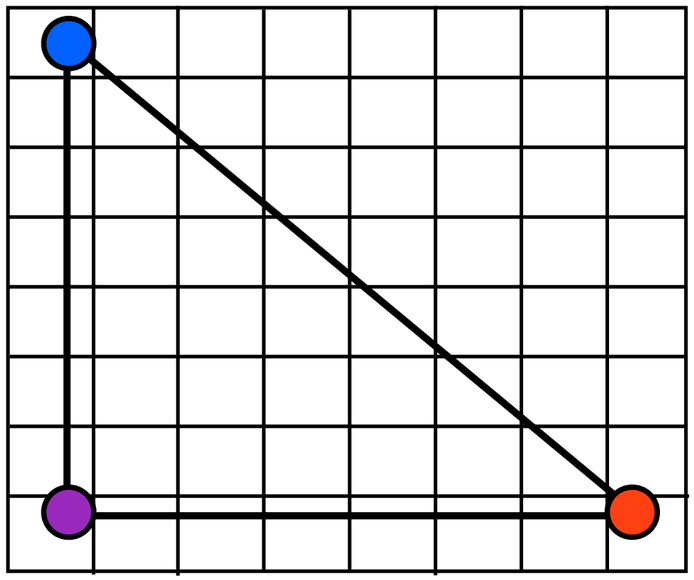


Image synthesis using the graphics pipeline

(As taught in graphics 101)

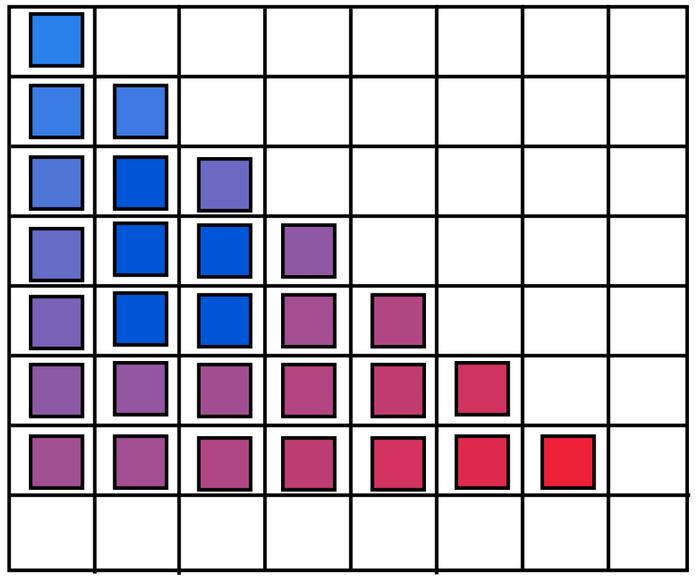
For each vertex, compute its projected position (and other surface attributes, like normal, color)...



Then given a projected triangle...

Image synthesis using the graphics pipeline

(As taught in graphics 101)



```
struct my_fragment
{
   // application-defined attributes (opaque to pipeline)

   float3 normal;          // surface normal
   float2 texcoord1;       // texture coordinate for texture 1
   float2 texcoord2;       // texture coordinate for texture 2
   float3 worldPosition;   // world-space position of surface

   // pipeline-interpreted fields:

   int x, y;           // screen pixel position of fragment
   float depth;        // triangle depth for fragment
};
```

Rasterization converts the projected triangle into fragments.

Issue 1: determining coverage (what fragments should be generated?)

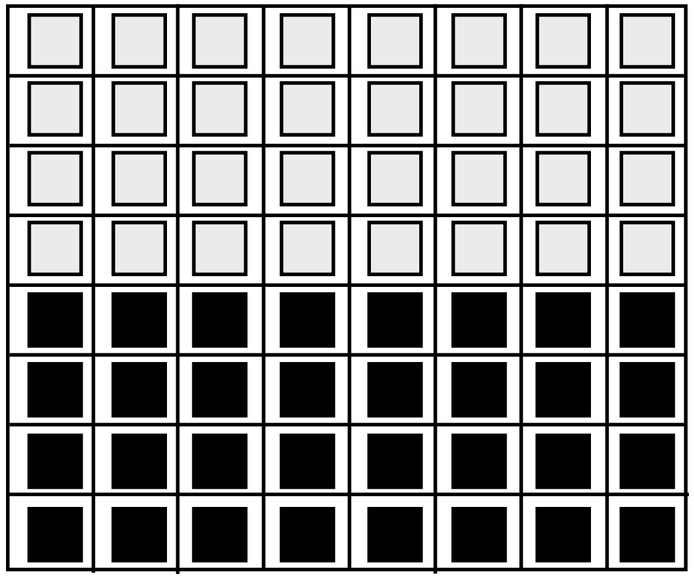
Issue 2: attribute interpolation (how to compute the value of surface attributes for each fragment?)

Image synthesis using the graphics pipeline

(As taught in graphics 101)

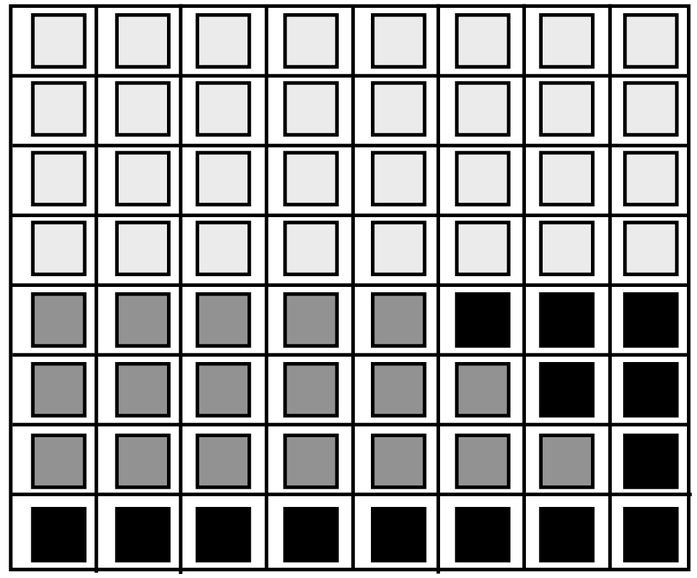
Z-buffer algorithm is used to determine the "closest" fragment at each pixel

Depth buffer before processing fragments from new triangle



Near values = ☐ Far values = ■

Depth buffer after processing fragments from new triangle

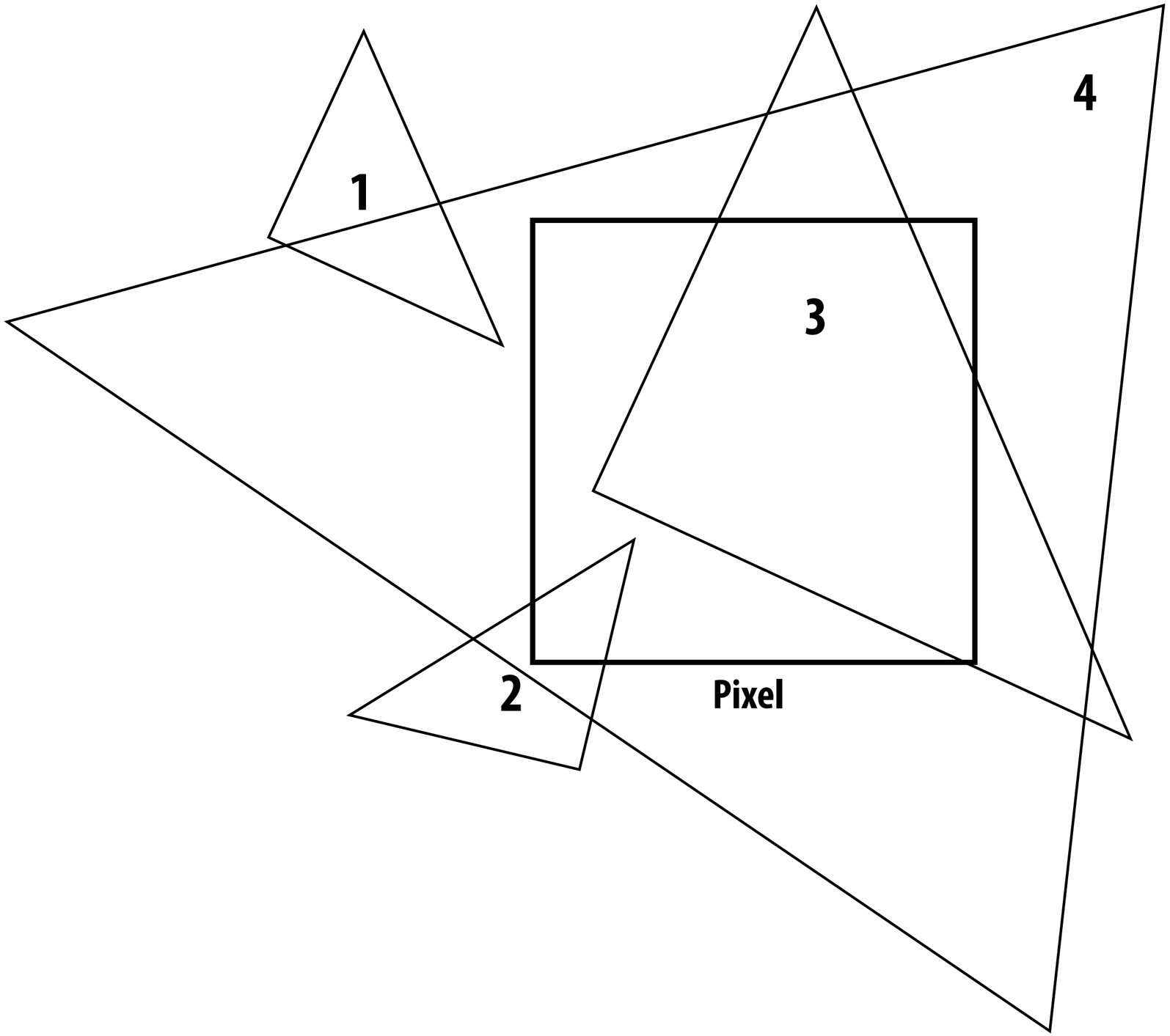


Let depth of new triangle = ▒

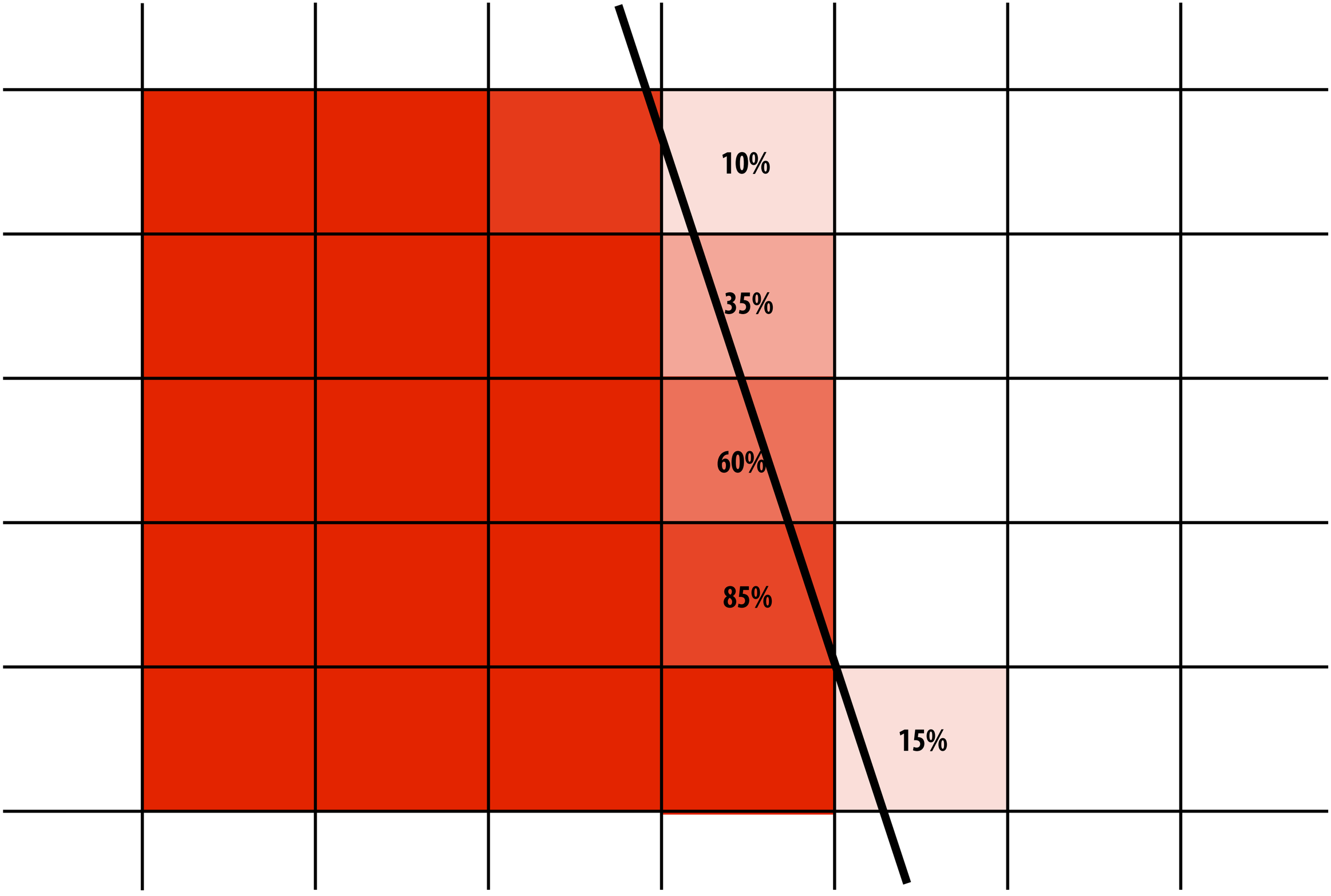
```
bool pass_depth_test(float a, float b) {
   return a < b;    // less-than predicate (the predicate is configurable in a modern pipeline)
}


// "depth test"
if (pass_depth_test(input_frag.depth, zbuffer[input_frag.x][input_frag.y])
{
   zbuffer[input_frag.x][input_frag.y] = input_frag.depth;
   *** UPDATE COLOR BUFFER HERE AS WELL ***
}
```

What does it mean for a pixel to be covered by a triangle?

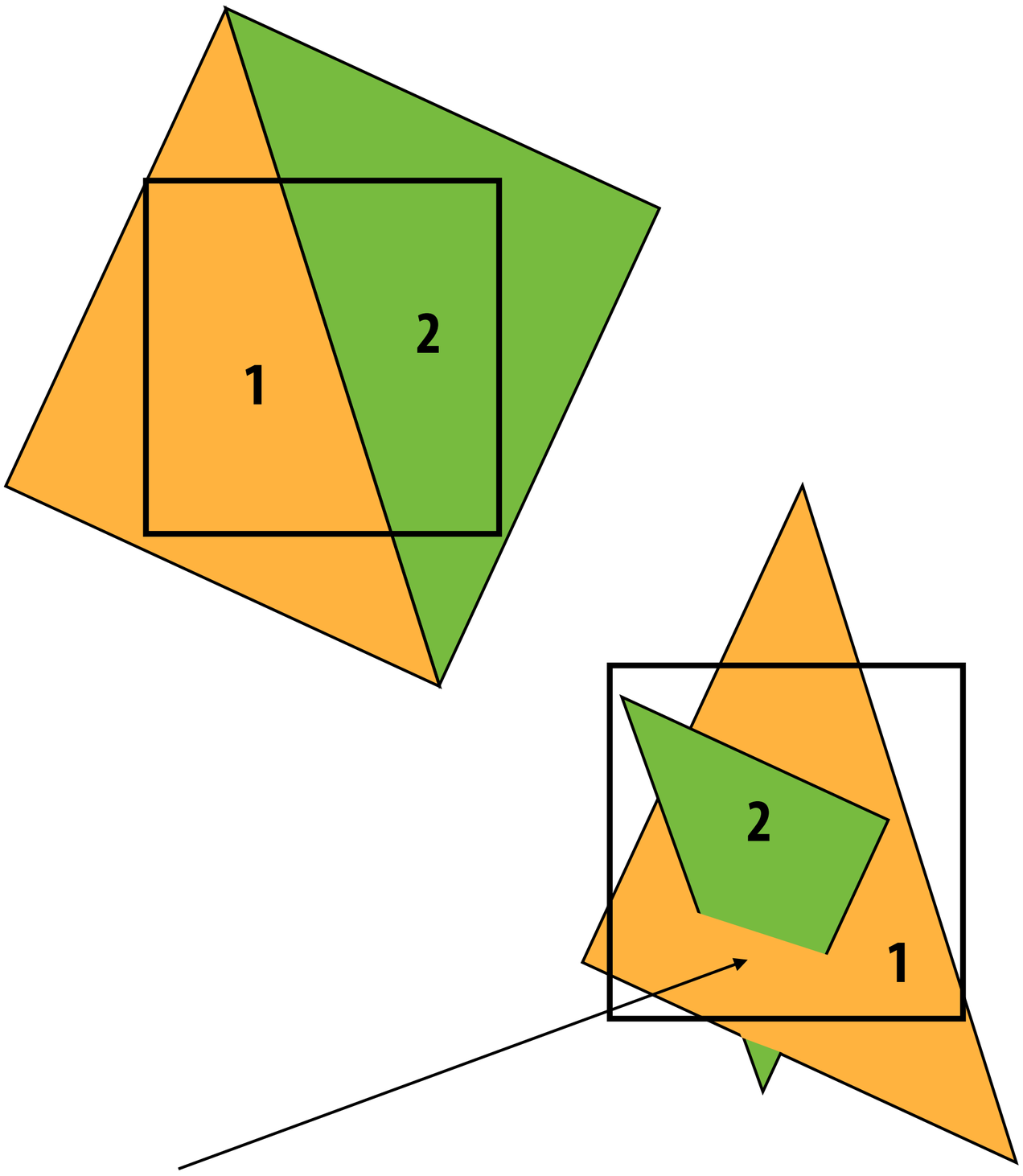
Question: which triangles "cover" this pixel?



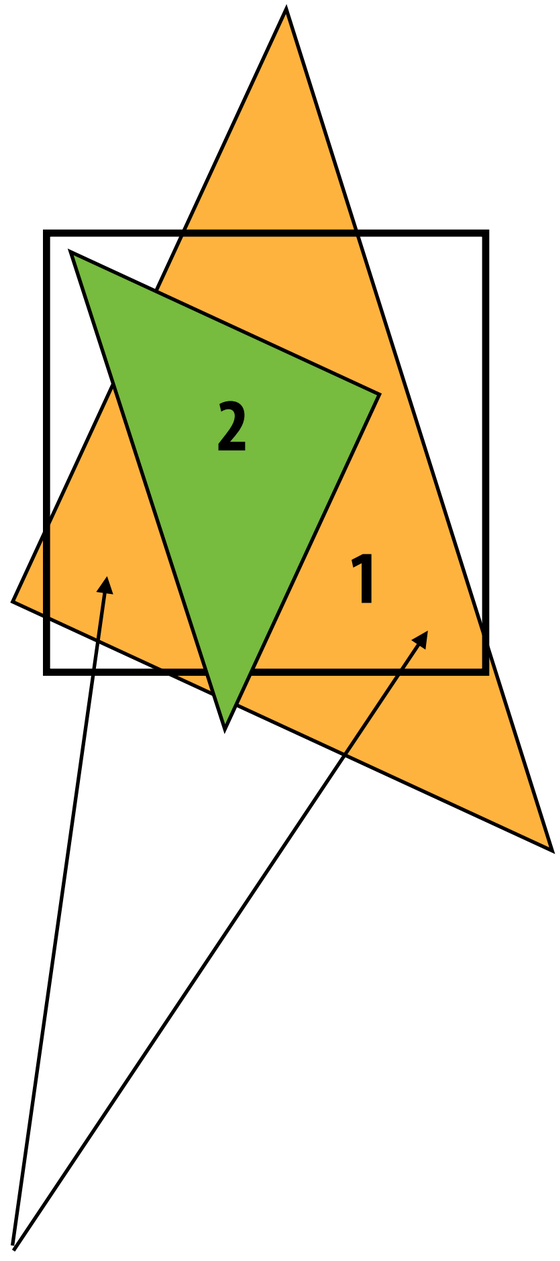
One option: analytically compute fraction of pixel covered by triangle



Analytical schemes get tricky when considering occlusion

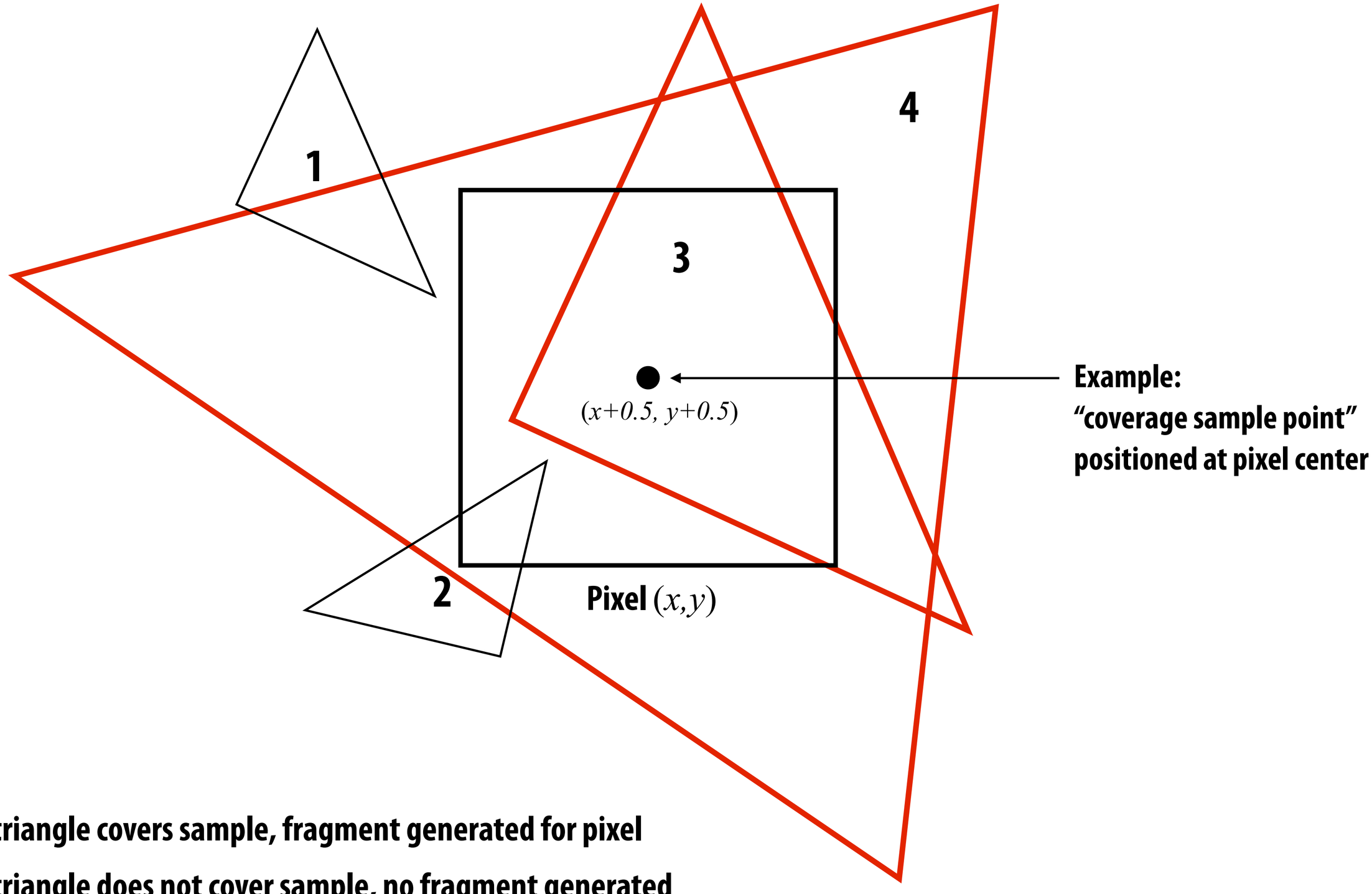


Interpenetration: even worse



Two regions of [1] contribute to pixel. One of these regions is not even convex.

The rasterizer estimates triangle-screen coverage using point sampling (this is the definition of when to generate a fragment in the spec)

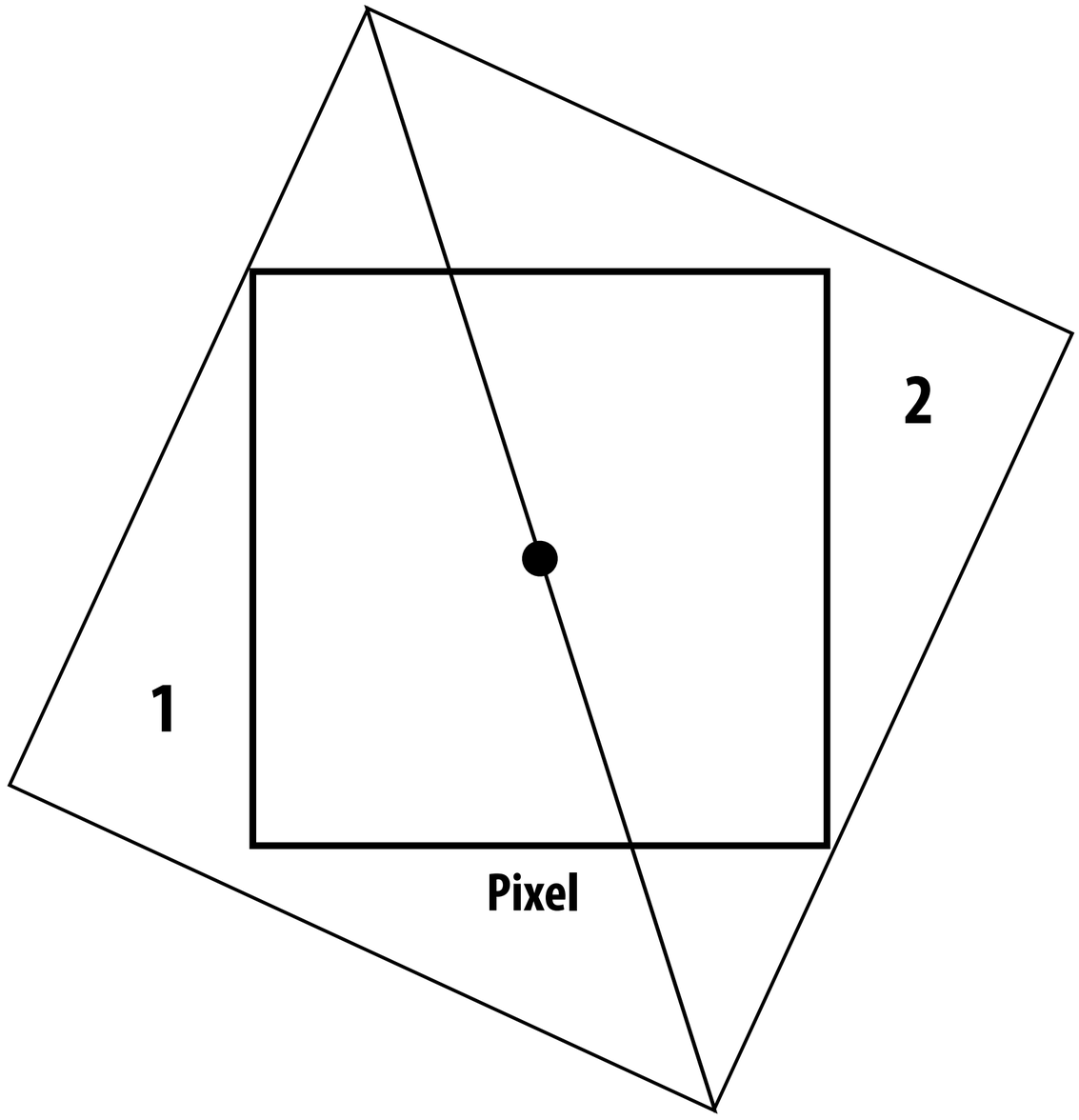


= triangle covers sample, fragment generated for pixel

= triangle does not cover sample, no fragment generated

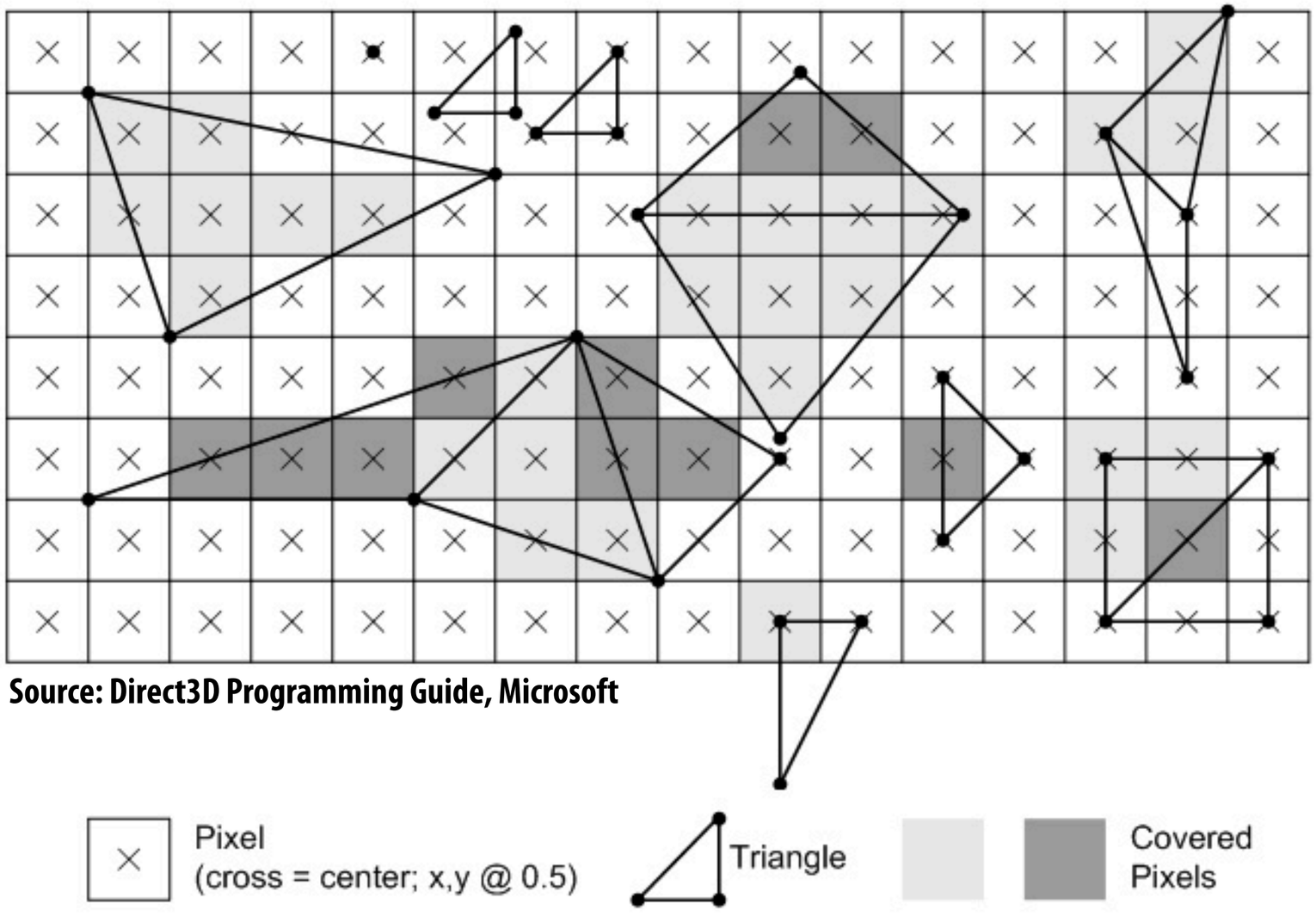
Edge cases (literally)

Is fragment generated for triangle 1? for triangle 2?



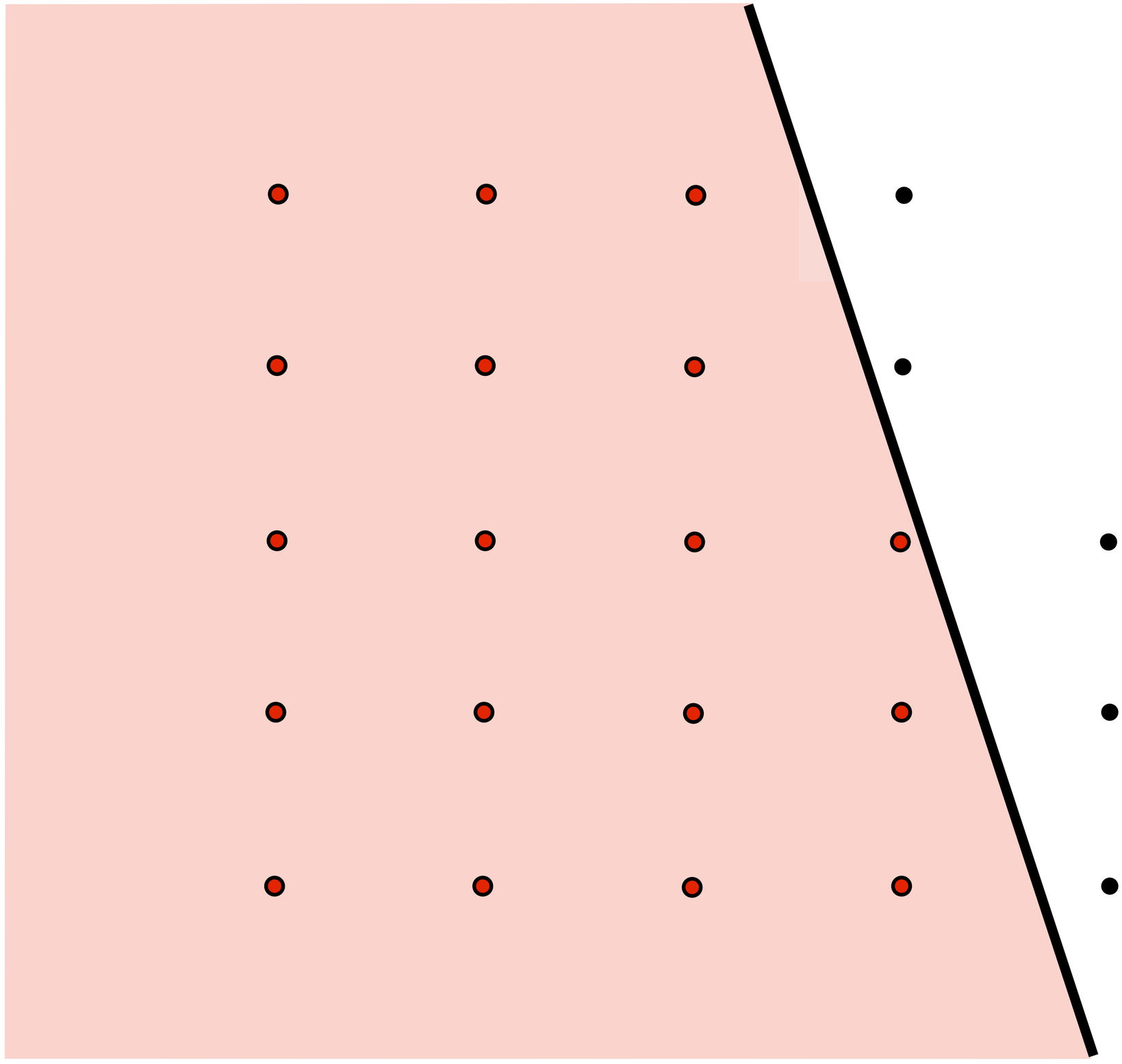
Edge rules

- Direct3D rules: when edge falls directly on sample, sample classified as within triangle if the edge is a "top edge" or "left edge"
  - Top edge: horizontal edge that is above all other edges
  - Left edge: an edge that is not exactly horizontal and is on the left side of the triangle. (triangle can have one or two left edges)

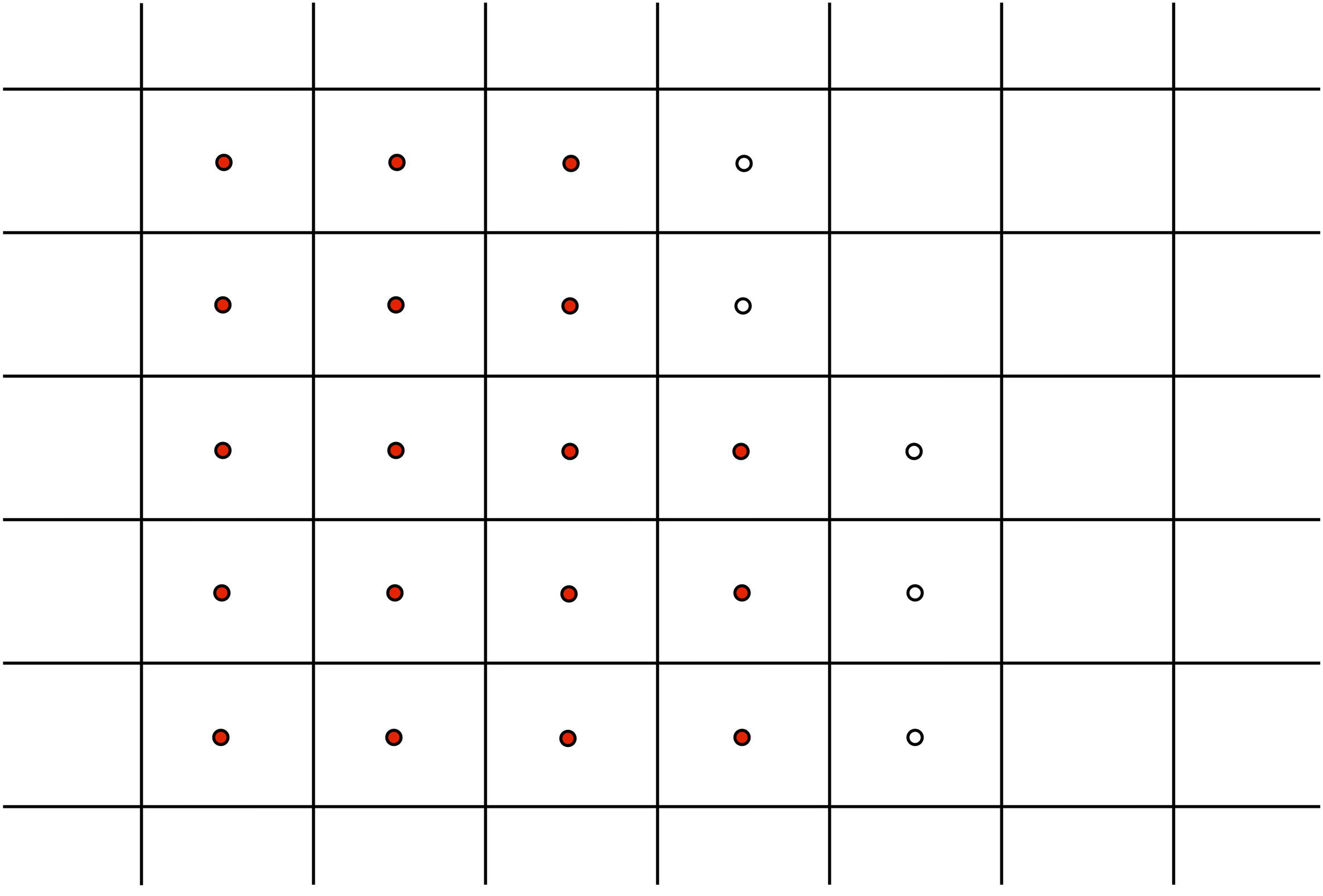


Source: Direct3D Programming Guide, Microsoft

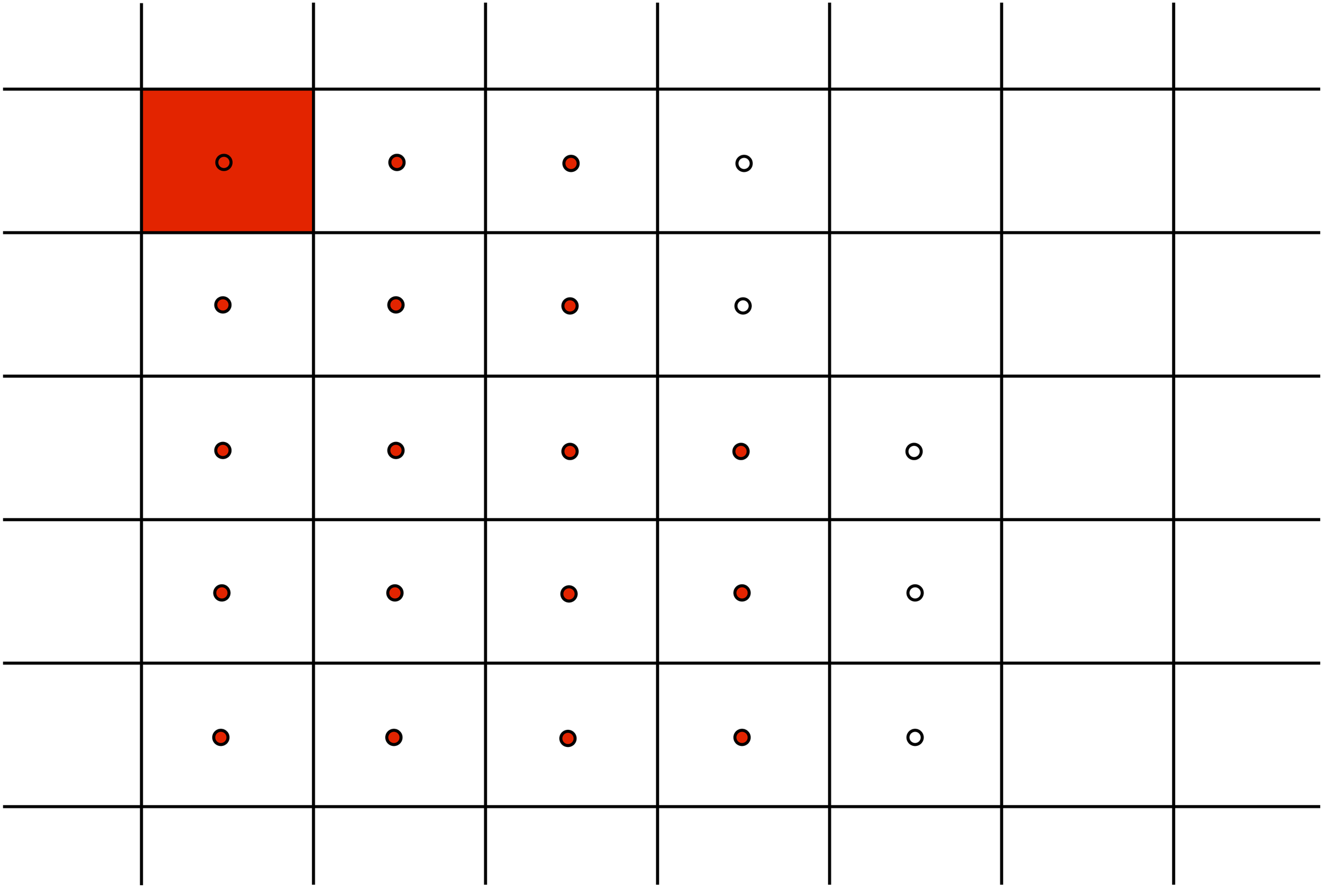
Results of sampling



Results of sampling (red dots = covered)

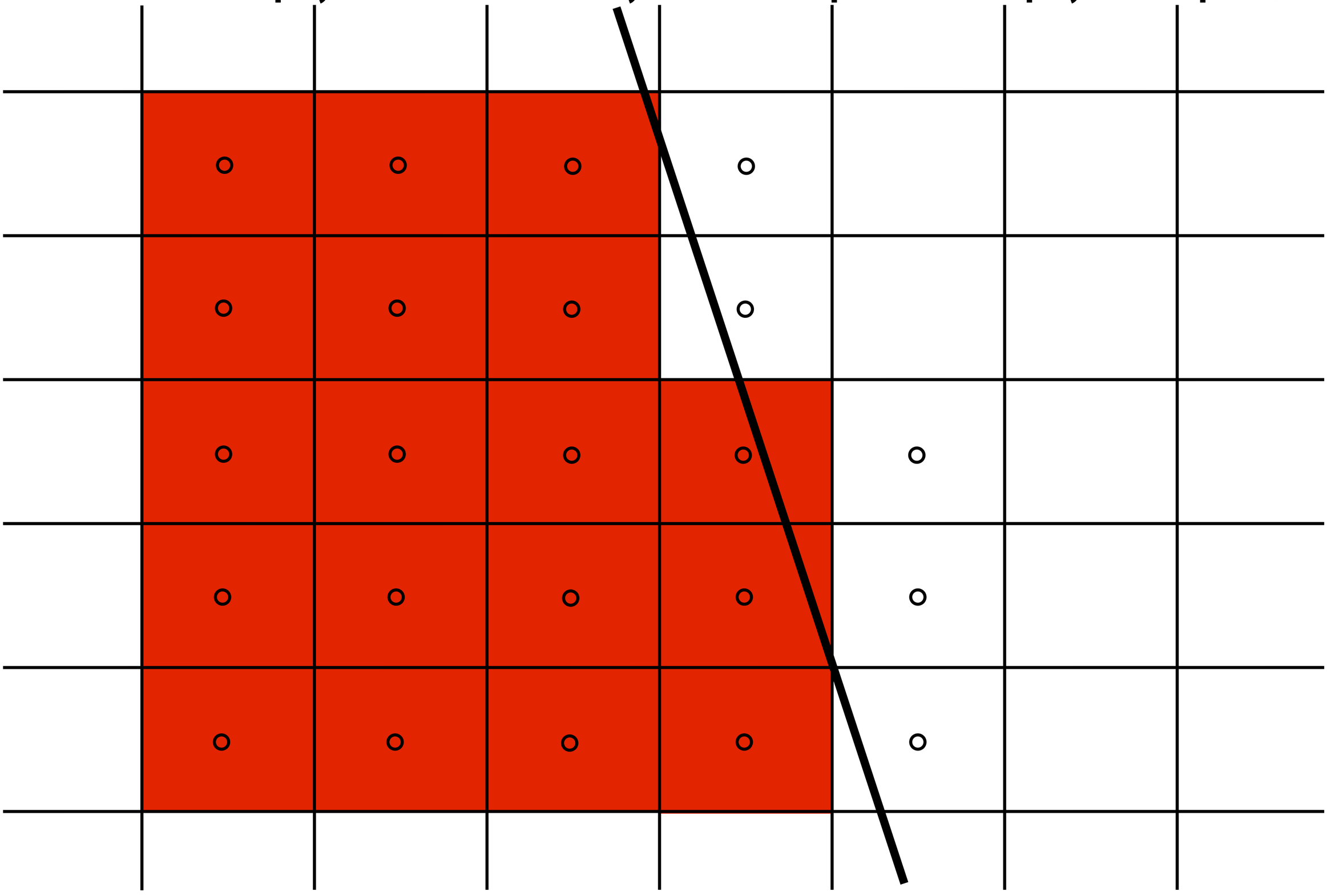


Reconstructing signal with a box filter



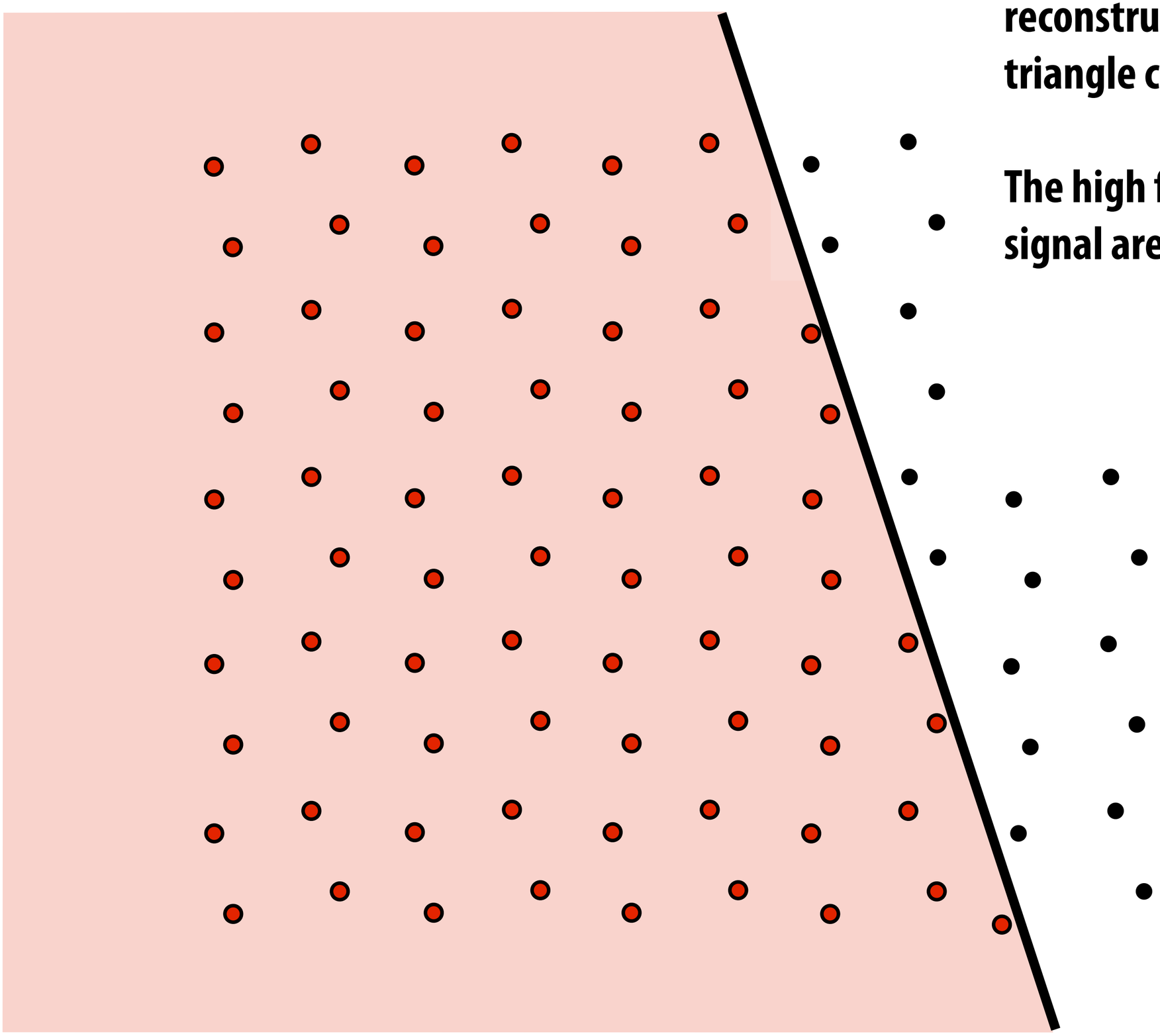
Reconstruction with box filter (aliased edges)

(consider this the displayed result: not actually true since a "pixel" on a display is not square)



Problem: aliasing

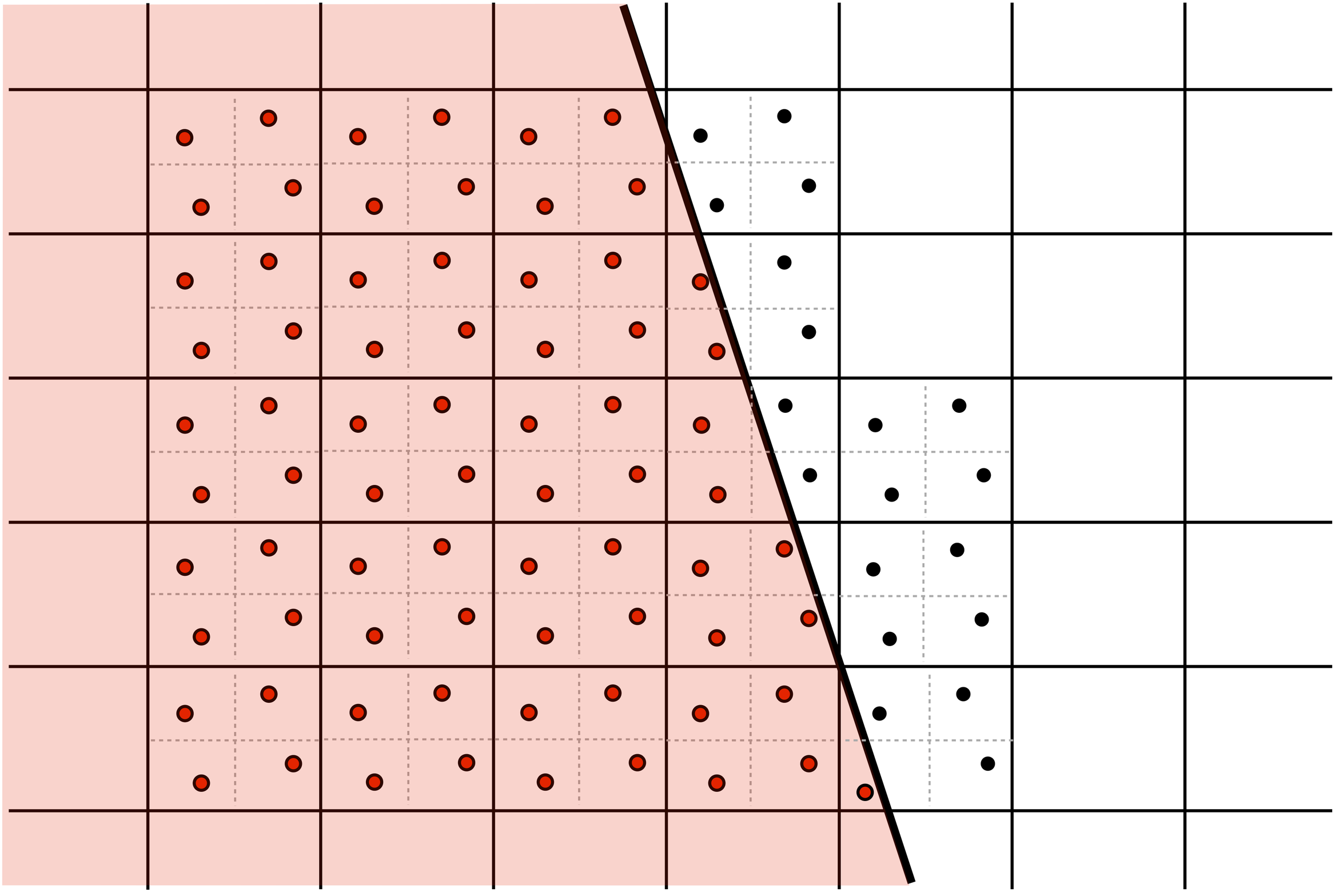
- Undersampling high frequency signal results in aliasing
  - "Jaggies" in a single image
  - "Roping" or "shimmering" in an animation



Supersampling: increase rate of sampling to more accurately reconstruct high frequencies in triangle coverage signal

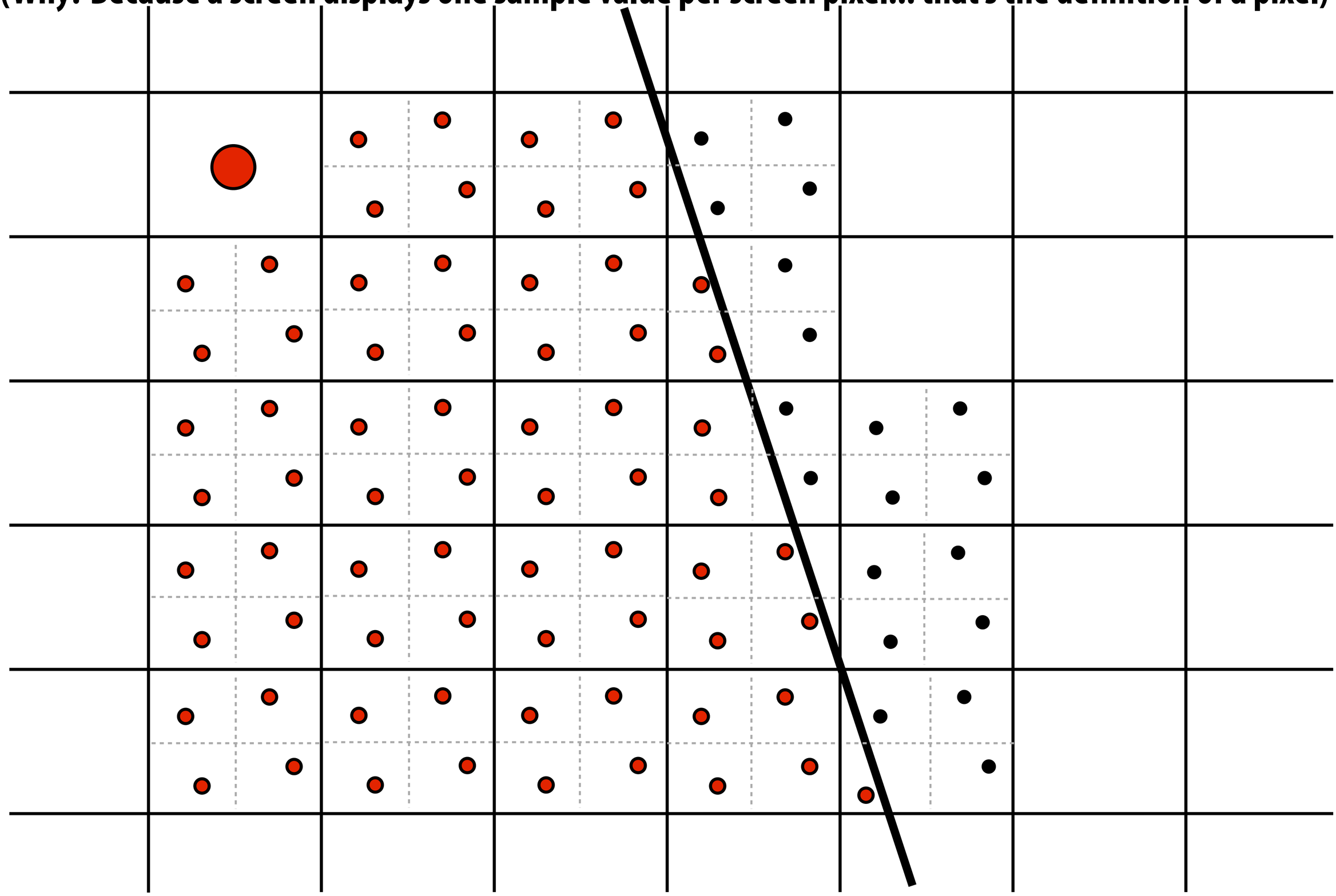
The high frequencies in coverage signal are created by triangle edges

Stratified sampling using four samples per pixel

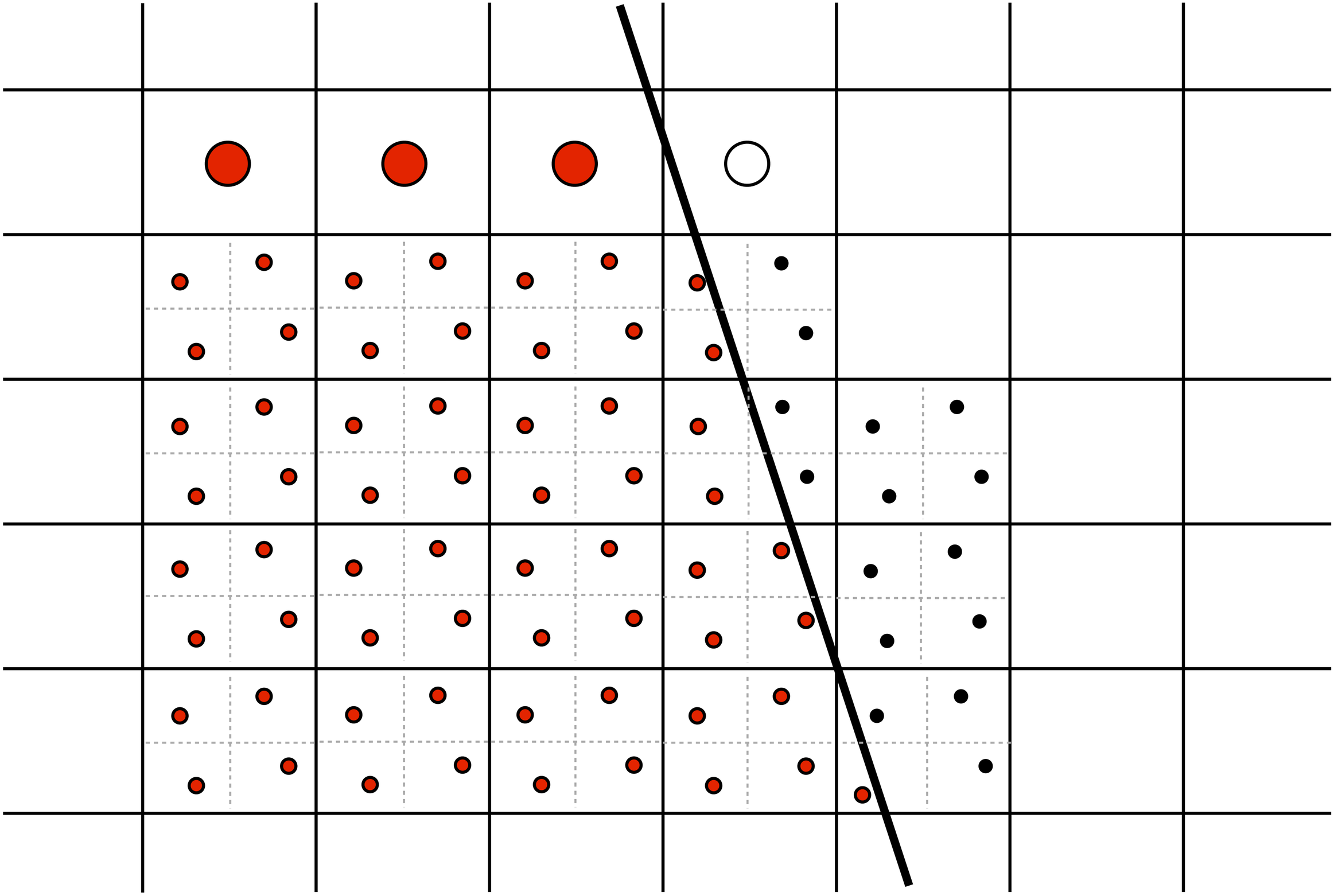


Resample to display's pixel rate (using box filter)

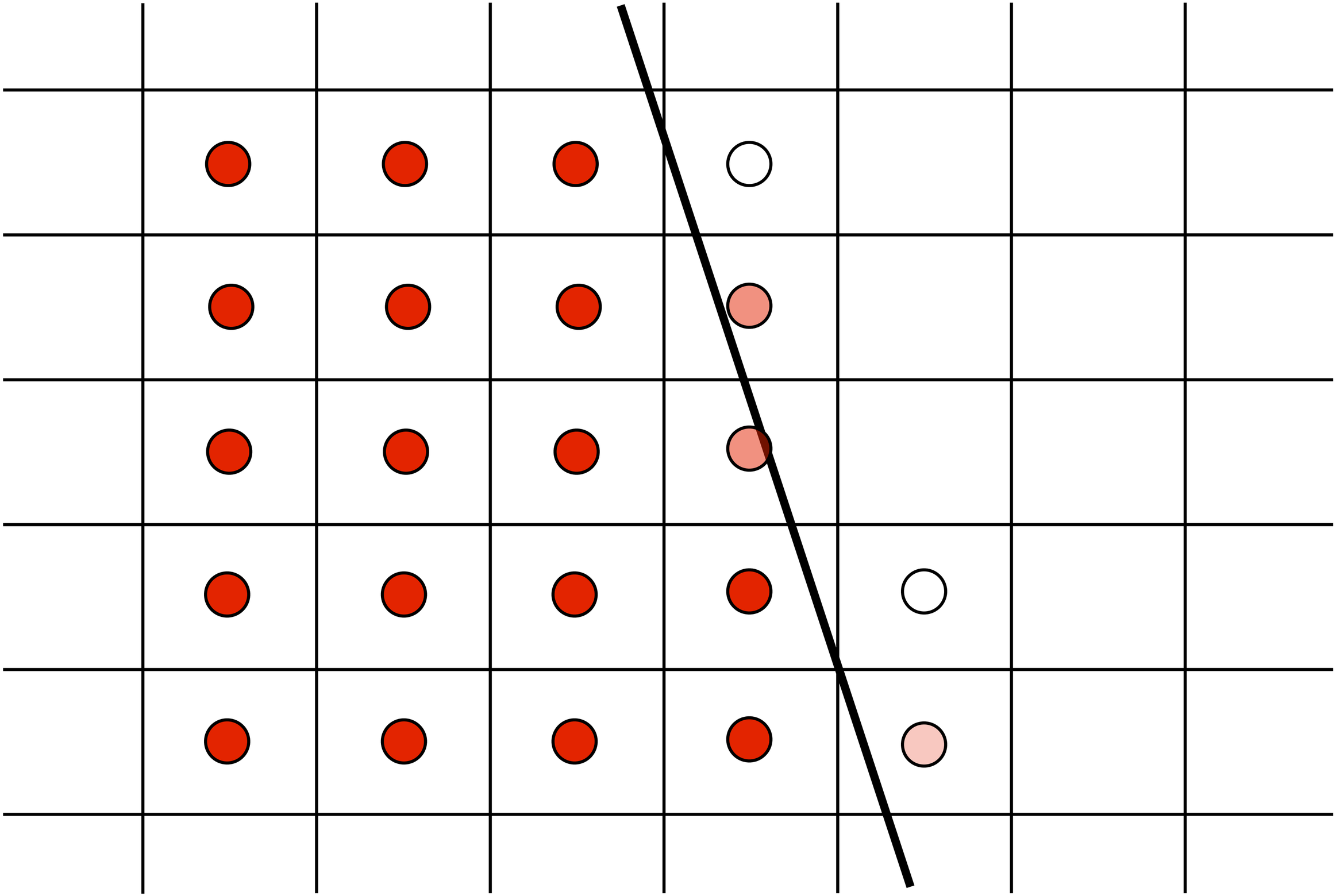
(Why? Because a screen displays one sample value per screen pixel... that's the definition of a pixel)



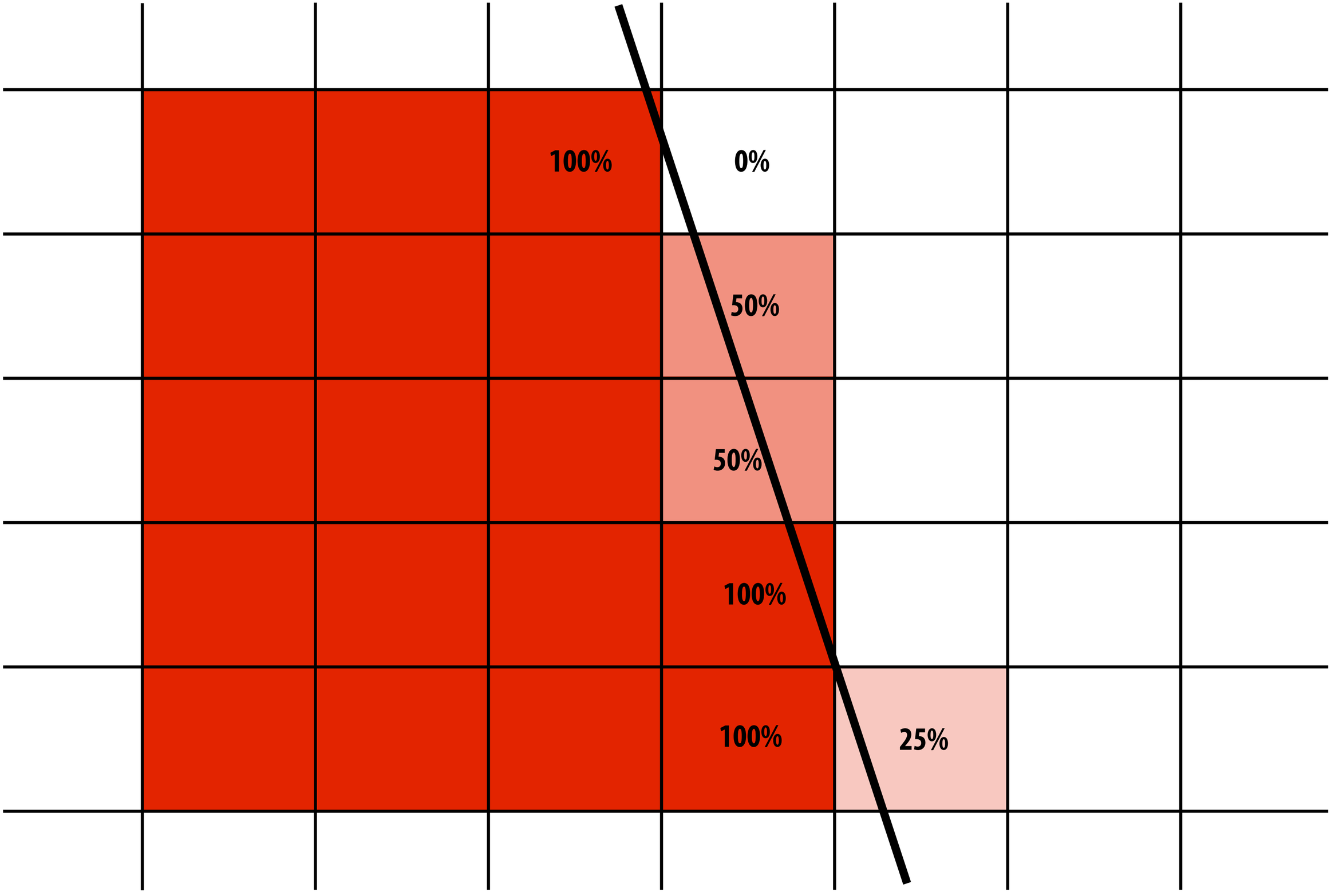
Resample to display's pixel rate (using box filter)



Resample to display's pixel rate (using box filter)



Displayed result (note anti-aliased edges)



Sampling coverage

- What we really want is the actual displayed intensity of a region of the physical screen to closely approximate the exact intensity of that region as measured by the scene's virtual camera.

- So we want to produce values to send to display that approximate the integral of scene radiance for the region illuminated by a display pixel (supersampling is used to estimate this integral)

Fragment generation

- Supersampling: generate one fragment for each covered sample

- Multi-sampling: general one fragment per pixel if any sample point within the pixel is covered

- Today, let's assume that the number of samples per pixel is one. (thus, both of the above schemes are equivalent)

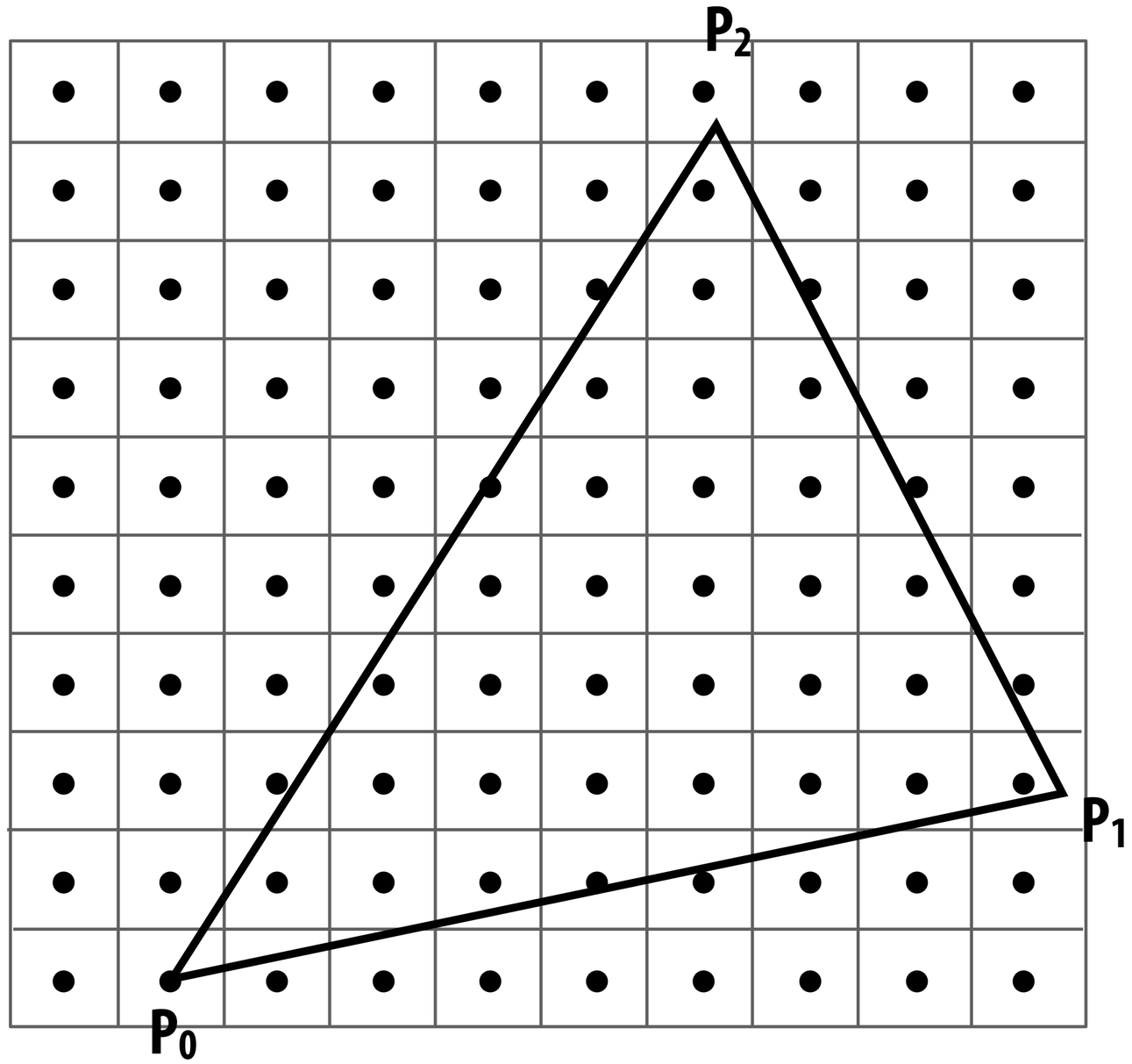
Point-in-triangle test

$$P_i = (x_i/w_i,\ y_i/w_i,\ z_i/w_i) = (X_i,\ Y_i,\ Z_i)$$

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

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

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



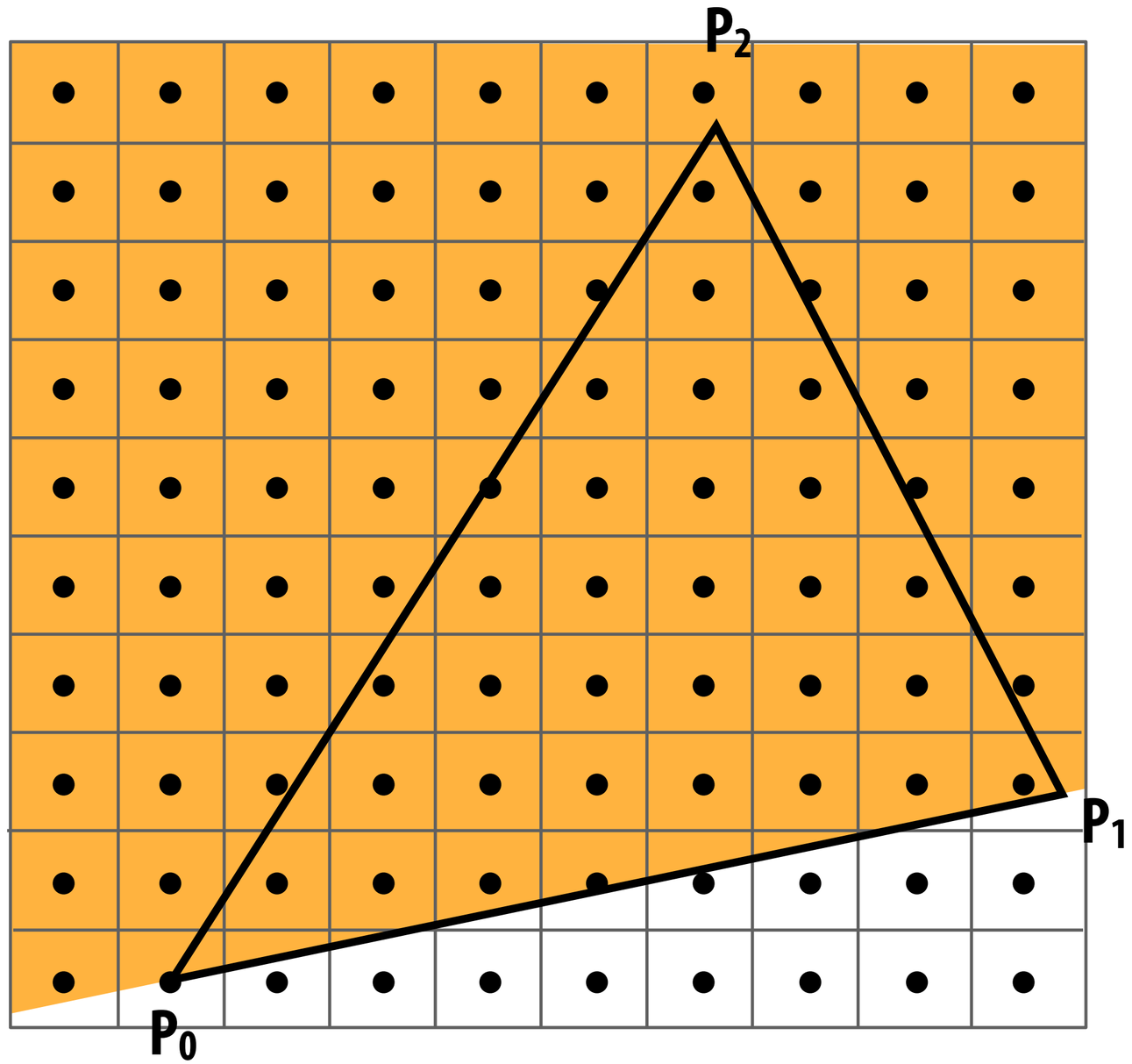
Point-in-triangle test

$$P_i = (x_i/w_i,\ y_i/w_i,\ z_i/w_i) = (X_i,\ Y_i,\ Z_i)$$

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

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

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



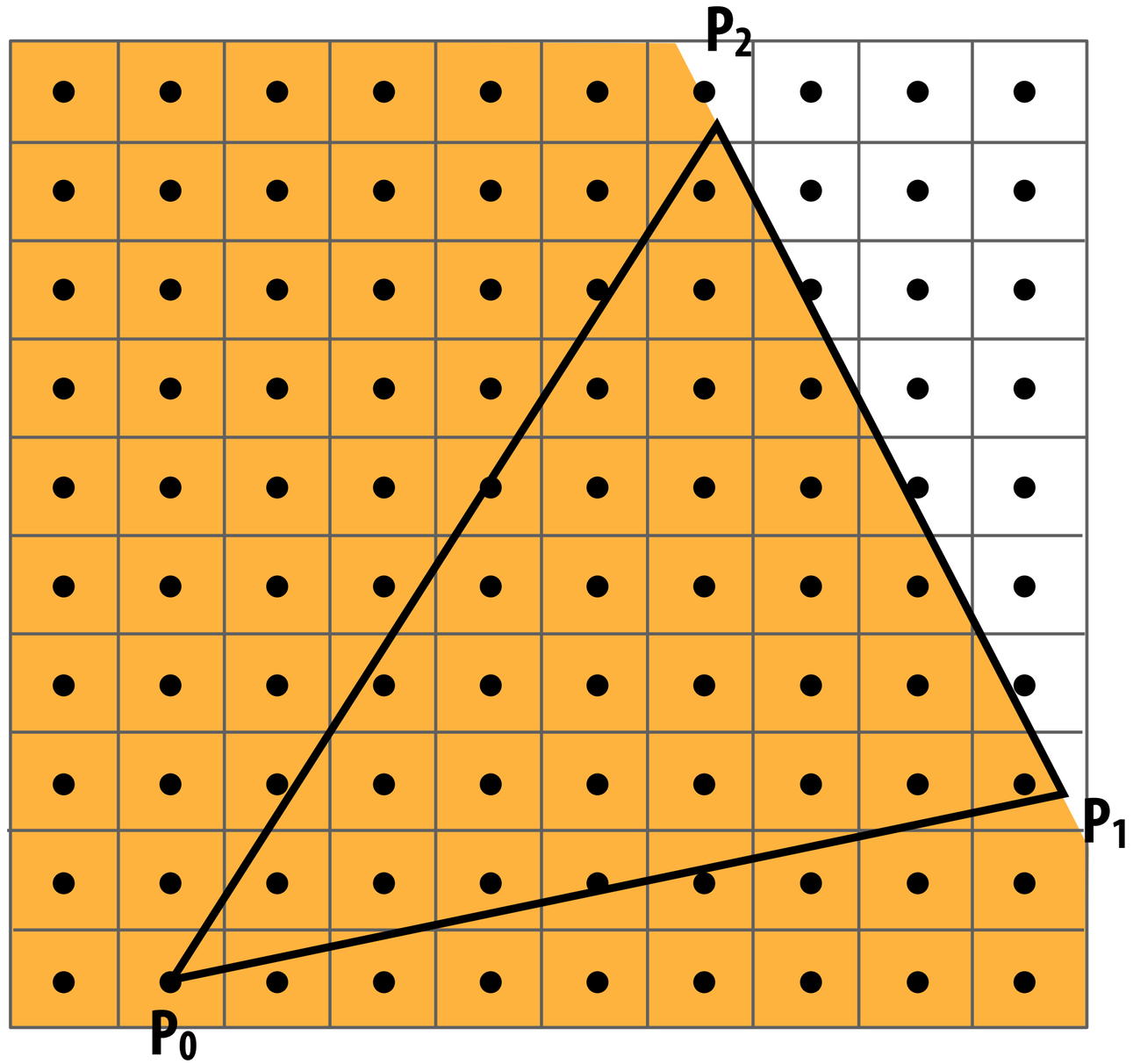
Point-in-triangle test

$$P_i = (x_i/w_i,\ y_i/w_i,\ z_i/w_i) = (X_i,\ Y_i,\ Z_i)$$

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

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

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



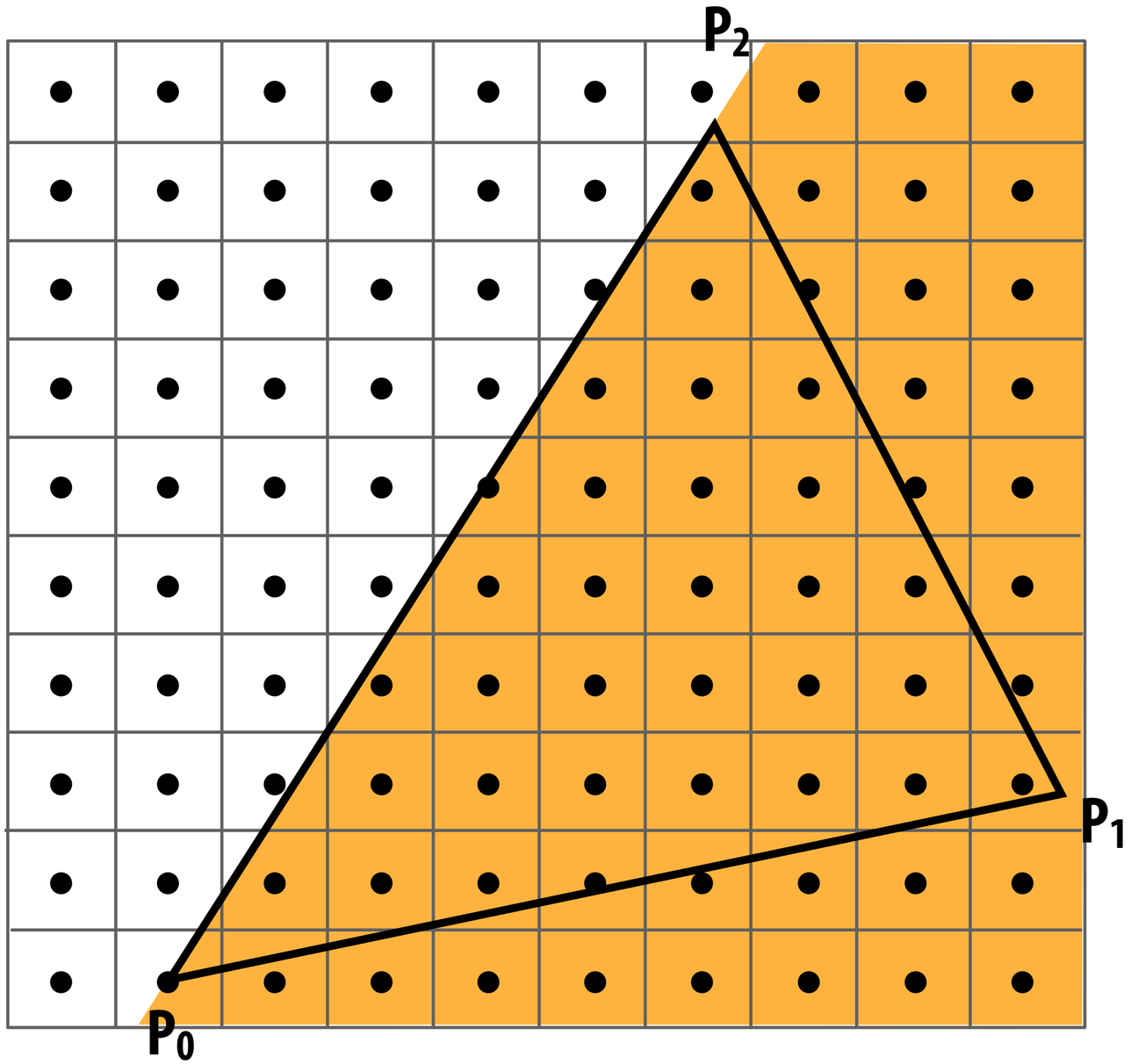
Point-in-triangle test

$$P_i = (x_i/w_i,\ y_i/w_i,\ z_i/w_i) = (X_i,\ Y_i,\ Z_i)$$

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

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

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

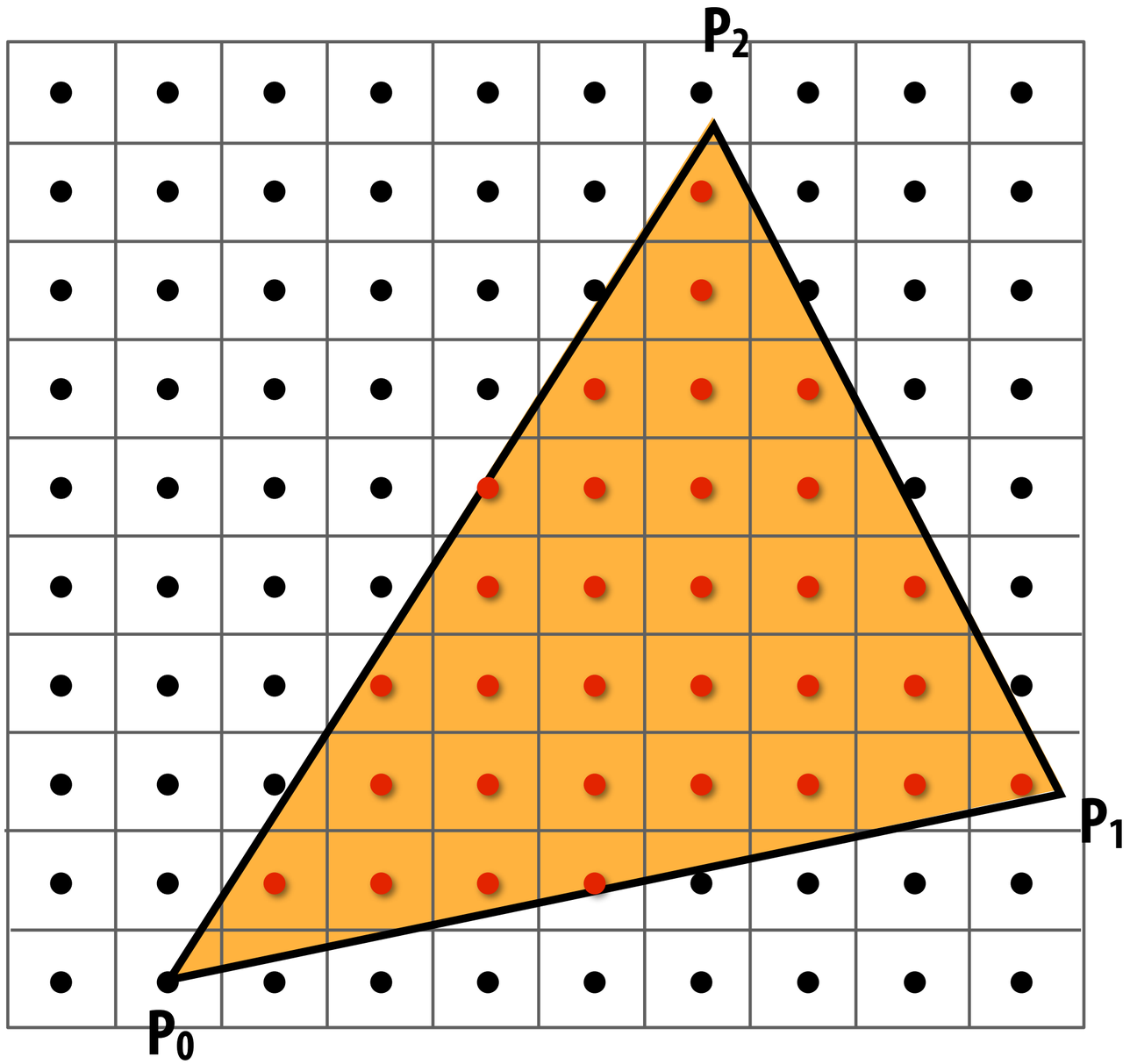


Point-in-triangle test

Sample point $s = (sx,\ sy)$ **is inside the triangle if it is inside all three edges.**

$$inside(sx,sy) =$$
$$E_0(sx,sy) < 0 \;\&\&$$
$$E_1(sx,sy) < 0 \;\&\&$$
$$E_2(sx,sy) < 0;$$

Note: actual implementation of $inside(sx,sy)$ **involves ≤ checks based on pipeline rasterizer's edge rules.**



Sample points inside triangle are highlighted red.

Incremental triangle traversal

$$P_i = (x_i/w_i,\ y_i/w_i,\ z_i/w_i) = (X_i,\ Y_i,\ Z_i)$$

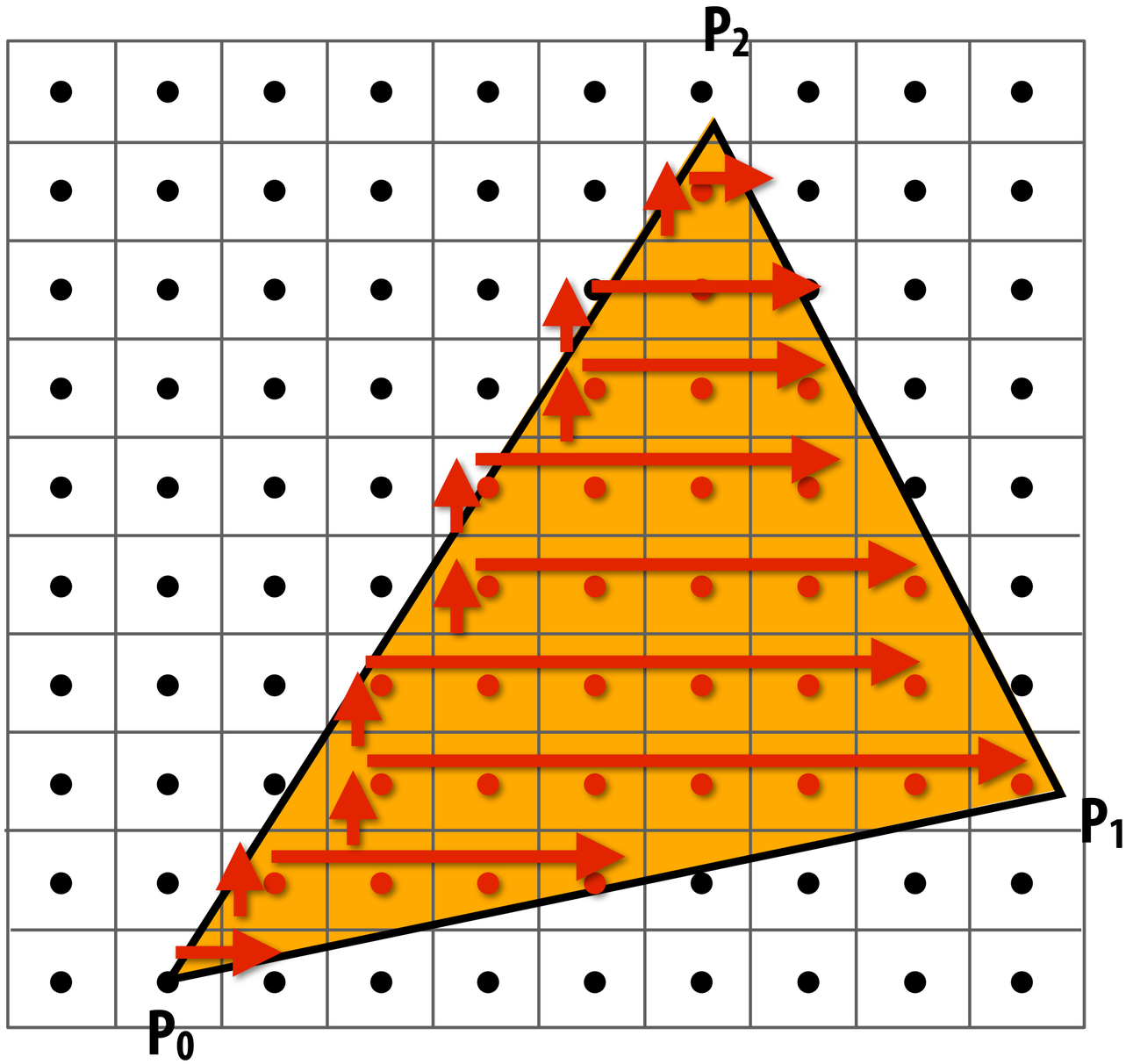
$$dX_i = X_{i+1} - X_i$$
$$dY_i = Y_{i+1} - Y_i$$

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

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

Note 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:
One addition per edge, per sample test

Note: many traversal orders are possible: backtrack, zig-zag, Hilbert/Morton curves (locality maximizing)

Modern hierarchical traversal

Traverse triangle similar to incremental update approach, but in blocks

Test all samples in block against triangle in parallel (e.g., data-parallel hardware implementation)

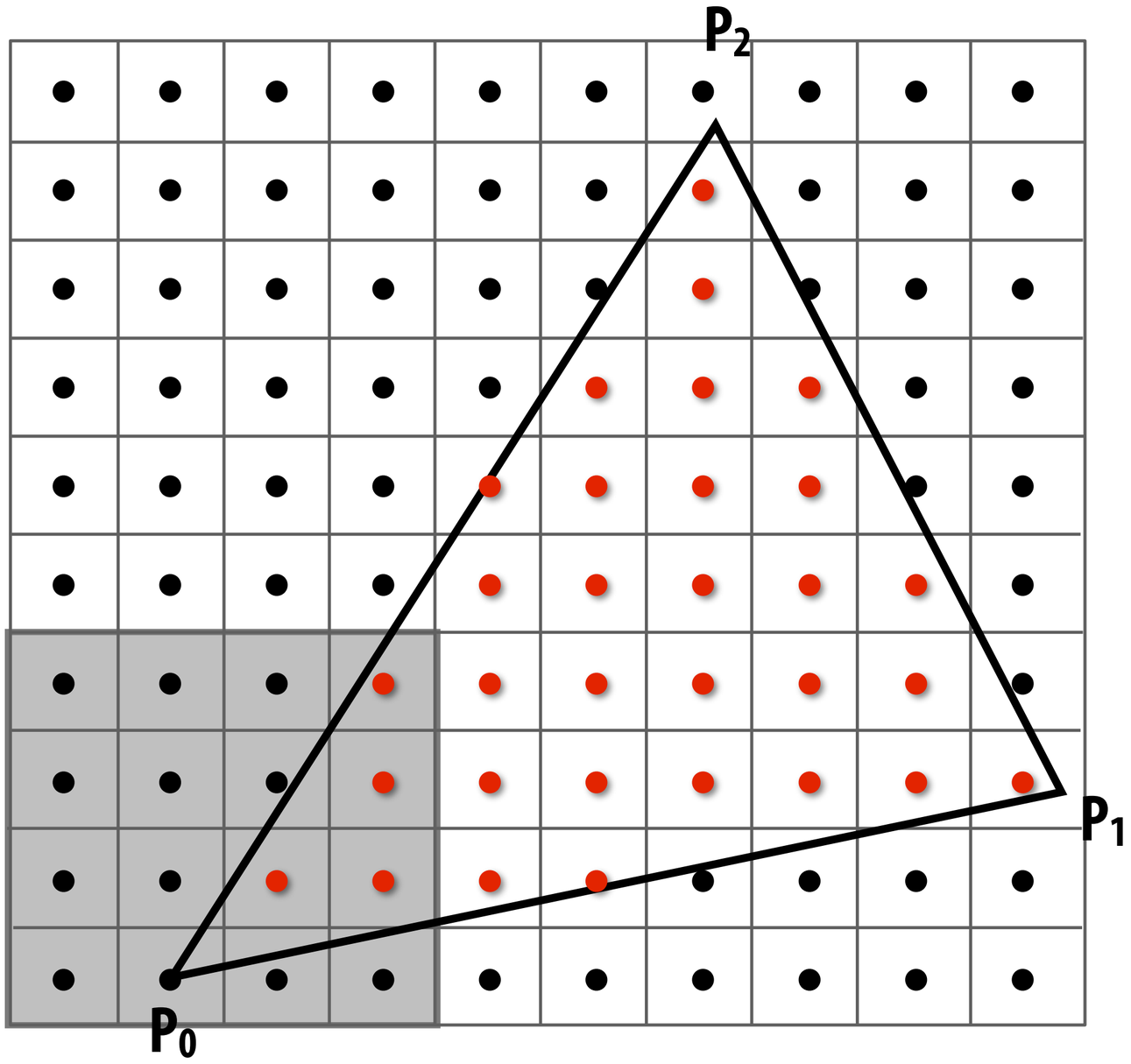
Can be implemented as multi-level hierarchy.

Advantages:

- Simplicity of wide parallel execution overcomes cost of extra point-in-triangle tests (recall: most triangles cover many samples, especially when super-sampling coverage)

- Can skip sample testing work: entire block not in triangle (early out), entire block entirely within triangle (early ins)

- Important for early out based on occlusion cull (later in this lecture)



Attribute assignment

- How are fragment attributes (color, normal, texcoords) computed?
  - Point sample attributes as well. (e.g., evaluated attributes at sample point)
  - Must compute $A(x,y)$ for all attributes

Computing a plane equation for an attribute:

Let A_0, A_1, A_2 be attribute values at the three triangle vertices
Let projected screen-space positions of vertices be $(X_0, Y_0), (X_1, Y_1), (X_2, Y_2)$
Linear interpolation of vertex attributes, so $A(x,y) = ax + by + c$ (attribute plane equation)

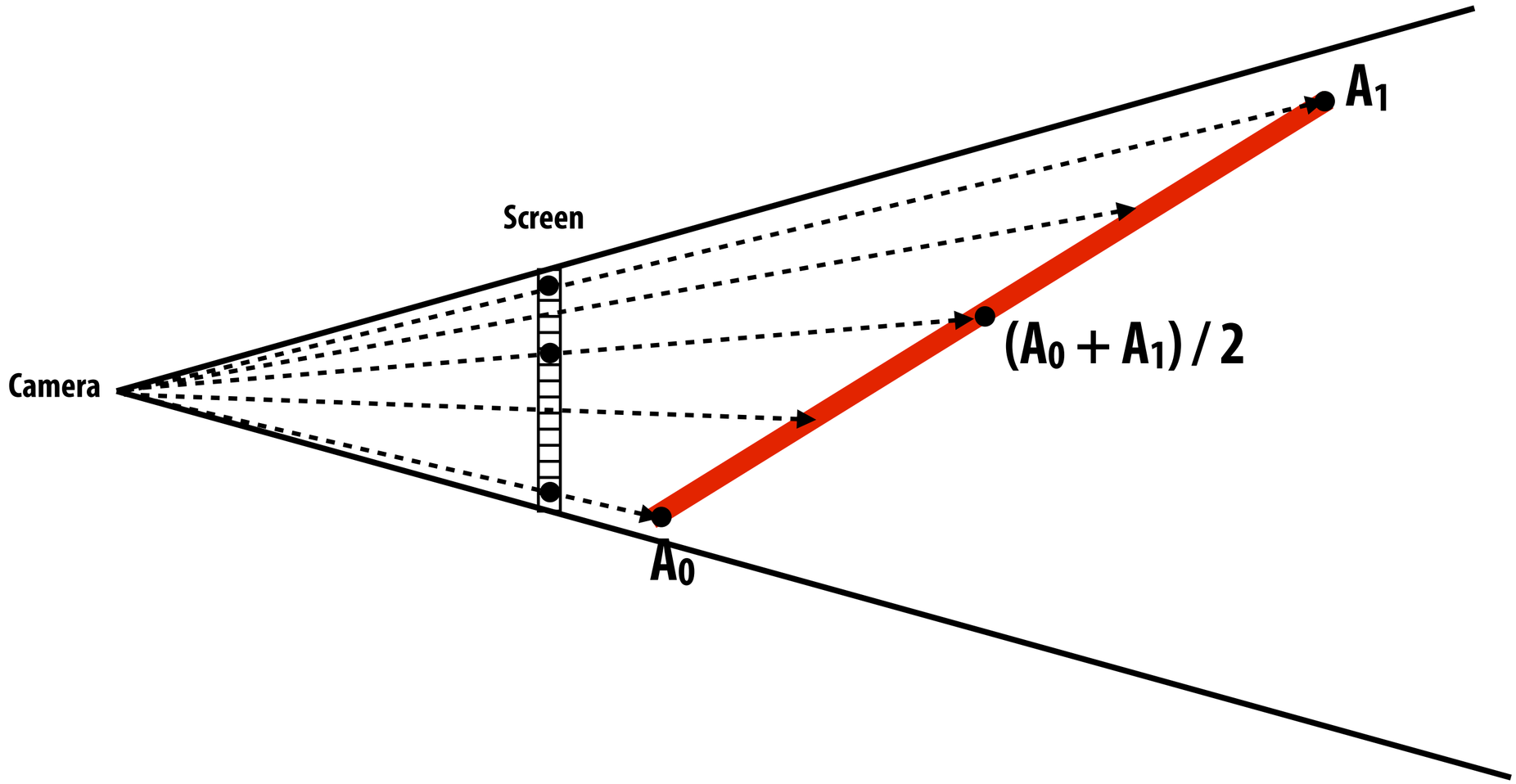
$$A_0 = aX_0 + bY_0 + c$$
$$A_1 = aX_1 + bY_1 + c$$
$$A_2 = aX_2 + bY_2 + c$$

3 equations, 3 unknowns. Solve for a,b,c **

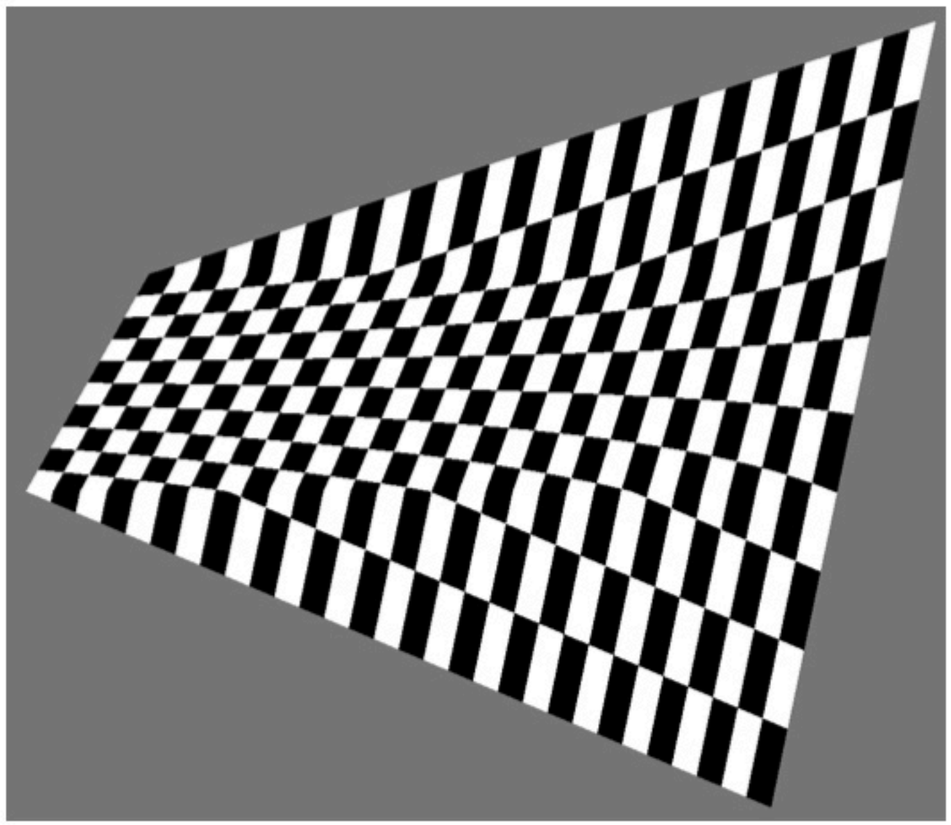
**** Discard zero-area triangles before getting here (recall we computed area in back-face culling)**

Perspective-correct interpolation

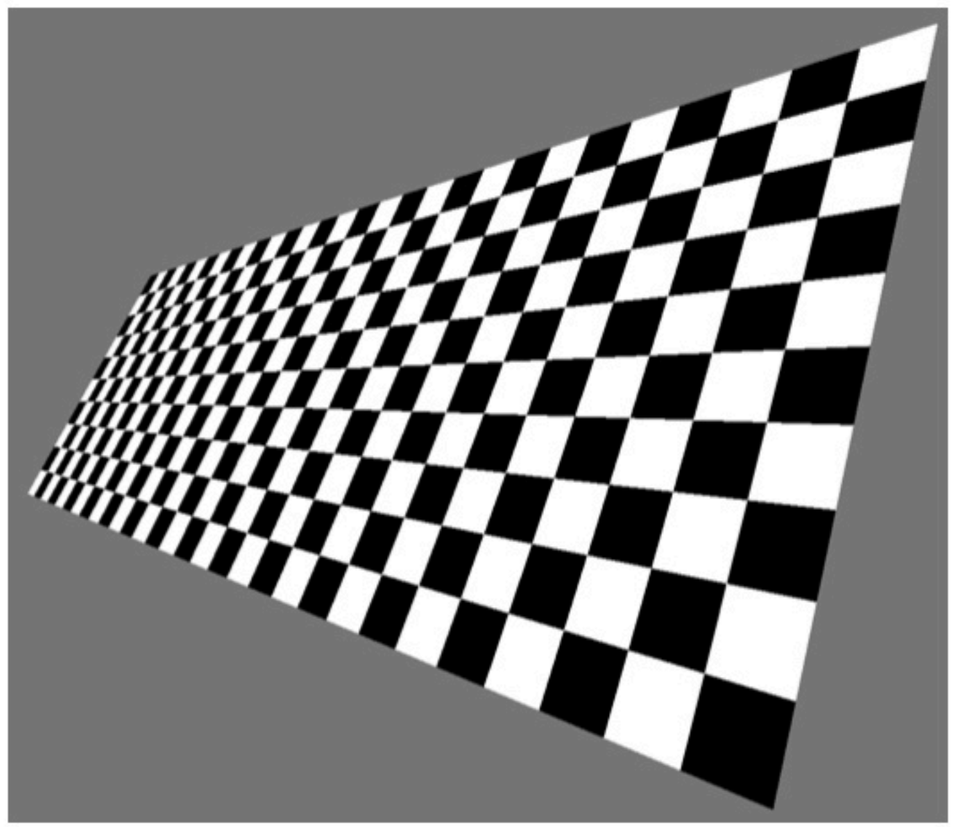
Attribute values should be interpolated linearly on triangle in 3D object space. Due to projection interpolation is not linear in screen XY



Perspective-correct interpolation



Linear screen interpolation of (u,v)



Perspective-correct interpolation of (u,v)

[images from Heckbert and Moreton 1991]

Perspective correct interpolation

Attribute values are linear on triangle in 3D, but not linear in projected screen XY

But... projected attribute values (A/w) **are linear in screen XY!**

For each generated fragment:

Evaluate $1/w\,(x,y)$ **(from precomputed plane equation for** $1/w$**)**

Reciprocate result to get $w(x,y)$

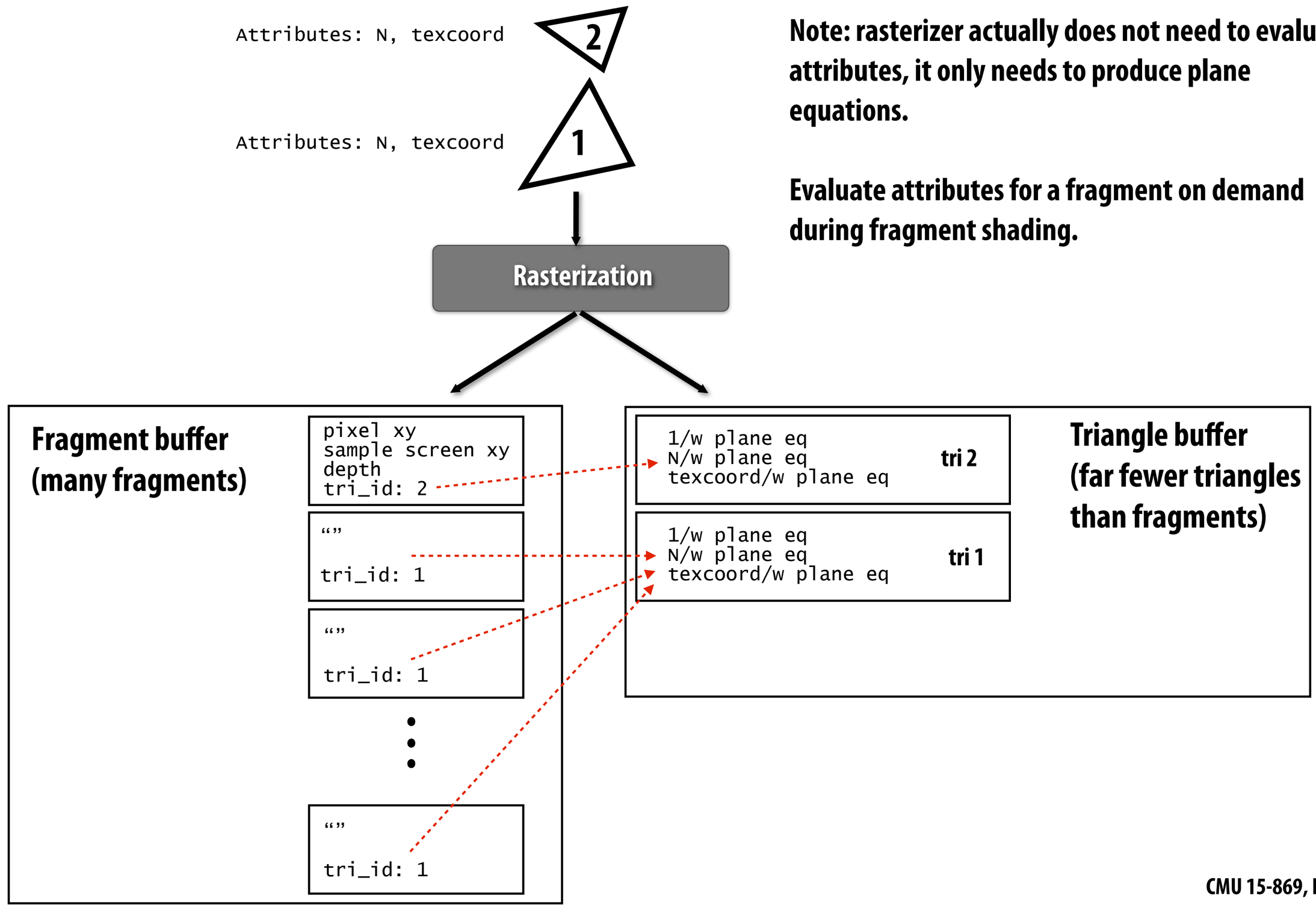
For each triangle attribute:

Evaluate $A/w\,(x,y)$ **(from precomputed plane equation for** A/w**)**

Multiply result by $w(x,y)$ **to get** $A(x,y)$

Key optimization:
store plane equations separate from fragments

(very effective for large triangles)



Note: rasterizer actually does not need to evaluate attributes, it only needs to produce plane equations.

Evaluate attributes for a fragment on demand during fragment shading.

Modern GPU rasterization

- **Triangle setup:**
  - Transform clip space vertex positions to screen space
  - Convert vertex positions to fixed point (Direct3D requires 8 bits of subpixel precision**)
  - Compute triangle edge equations
  - Compute plane equations for all vertex attributes, 1/w, and Z

- **Traverse triangle in blocks:**
  - Attempt to trivially accept/reject block using edge tests on block corners
  - Identify covered samples using edge tests (wide data-parallelism in implementation)
  - Generate and emit fragments (also emit per-triangle data as necessary)

** Note 1: limited precision can be a good thing: really acute triangles snap to 0 area and get discarded

** Note 2: limited precision can be a bad thing: precision limits in (x,y) can limit precision in Z (see Akeley and Su, 2006)

Depth-buffer for fragment occlusion

- **Depth-buffer stores depth of scene at each coverage sample point**
  - Per sample, not per pixel!
  - In practice, depth buffer usually stores z/w
- **Triangles are planar**
  - Each triangle has exactly one depth* at each sample point (so there is a well-defined ordering of fragments at each sample point) ✓
- **Occlusion check using Z-buffer algorithm**
  - Constant-time occlusion test per fragment ✓
  - Constant space per coverage sample ✓
  - Constant space per depth-buffer ✓

* Assumes edge-on triangles have been discarded

Depth-buffer for occlusion

- **High bandwidth requirements (particularly when super-sampling)**
  - Number of Z-buffer reads/writes depends on:
    - Depth complexity of the scene
    - The order triangles are provided to the graphics pipeline
      (if depth test fails, don't write to depth buffer or rgba)

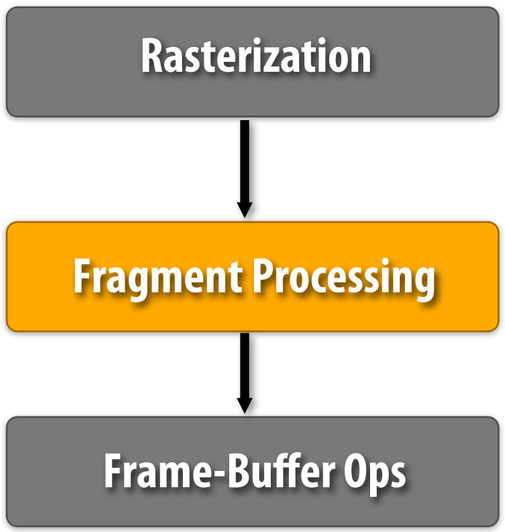
- **Bandwidth estimate:**
  - 60 Hz x 2 MP image x avg. depth complexity 4 (assume replace 50% of time, 32-bit Z) = 2.8 GB/s
  - If super-sampling at 4 times per pixel, multiply by 4
  - Consider five shadow maps per frame (1 MPixel, not super-sampled): additional 8.6 GB/s
  - Note: this is just depth accesses. It does not include color-buffer bandwidth

- **Modern GPUs implement caching and lossless compression of both color and depth buffers**

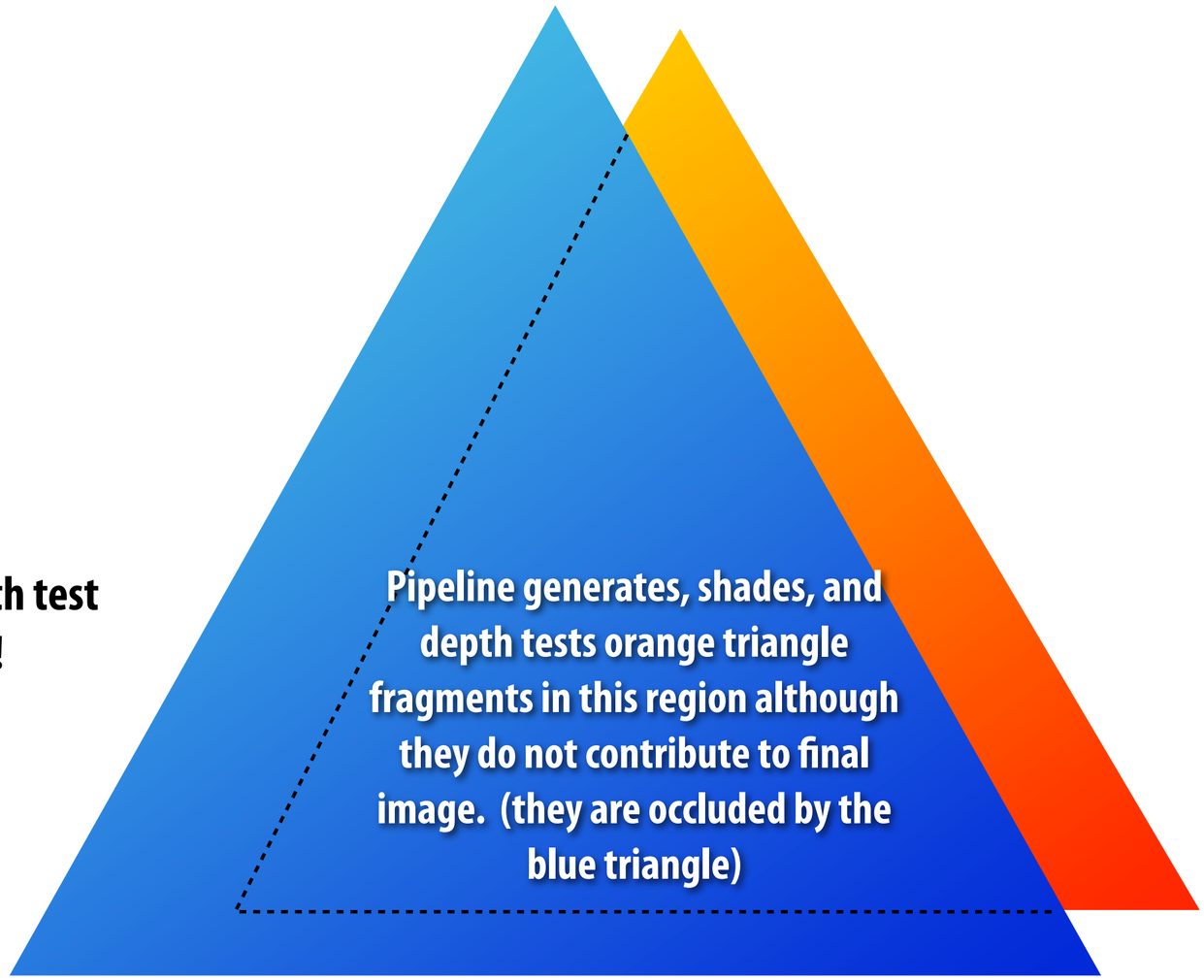
Early occlusion culling

Early occlusion-culling ("early Z")

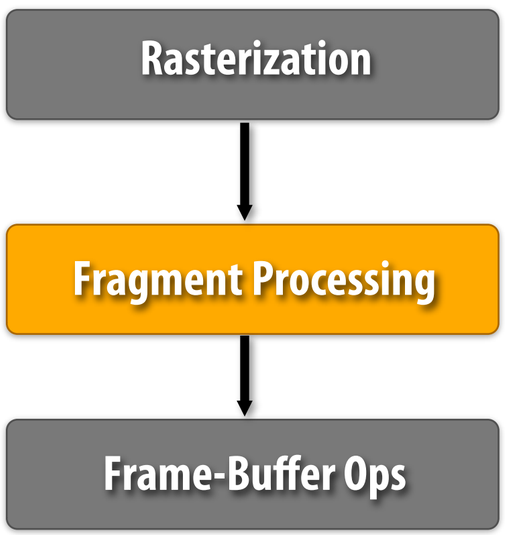
Idea: discard fragments that will not contribute to image as quickly as possible in the pipeline



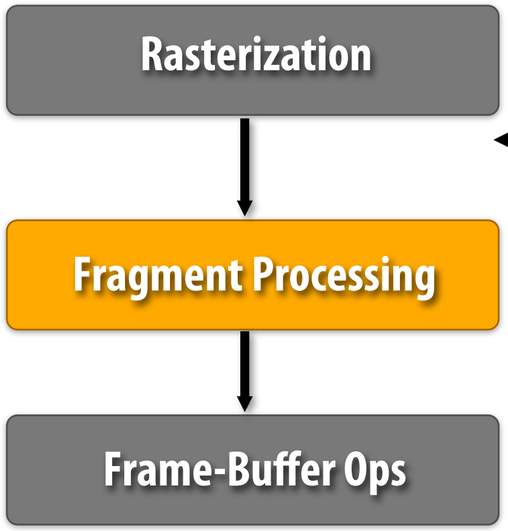
Graphics pipeline specifies that depth test is performed here!



Early occlusion-culling ("early Z")



Graphics pipeline specifies that depth test is performed here!



Optimization: reorder pipeline operations: perform depth test immediately following rasterization and before fragment shading

A GPU implementation detail: not reflected in the graphics pipeline abstraction

Key assumption: occlusion results do not depend on fragment shading

- Example operations that prevent use of early Z optimization: alpha test enabled, fragment shader modifies fragment's Z value

Note: early Z only provides benefit if closer triangle is rendered by application first (application developers are encouraged to submit geometry in front-to-back order if possible)

Summary: early occlusion culling

- Idea: can reorder pipeline operations without impacting correctness: perform depth test prior to fragment shading
- Benefit: reduces fragment processing work
  - Effectiveness of optimization dependent on triangle ordering
  - Ideal geometry submission order: front-to-back order
- Does not reduce depth-buffer bandwidth
  - The same depth-buffer reads and writes still occur (they just occur earlier)
- Implementation-specific optimization, but programmers know it is there
  - Commonly used two-pass technique in graphics applications: "Z-prepass"
    - Pass 1: render all scene geometry, with fragment shading and color buffer writes disabled (initialize depth buffer is now in its end-of-rendering state)
    - Pass 2: re-render scene with shading enabled and with depth test predicate less than-or-equal
  - Overhead: must process scene geometry twice
  - Benefit: minimizes expensive fragment shading work

Hierarchical early occlusion culling: "hi-Z"

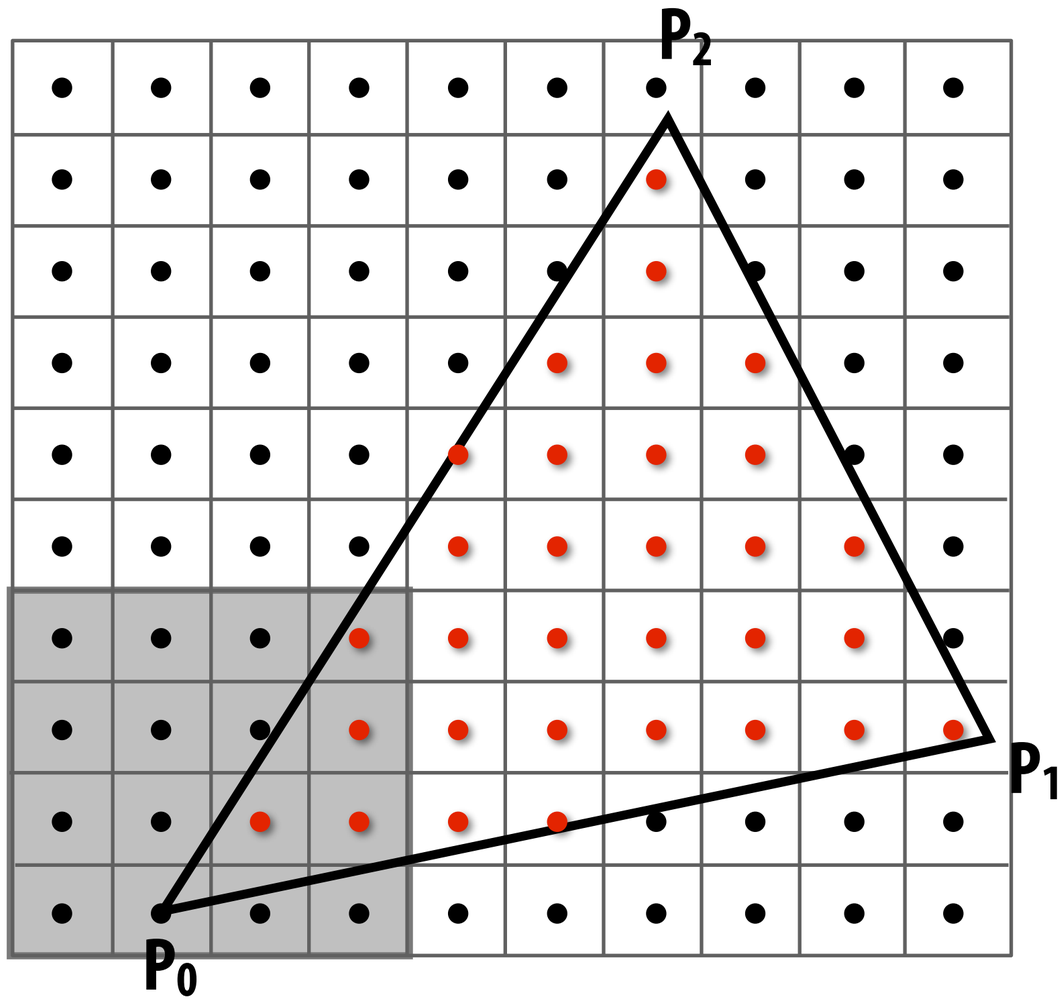
Recall hierarchical traversal during rasterization

Z-Max culling:

For each screen tile, compute farthest value in the depth-buffer: `z_max`

During traversal, for each tile:

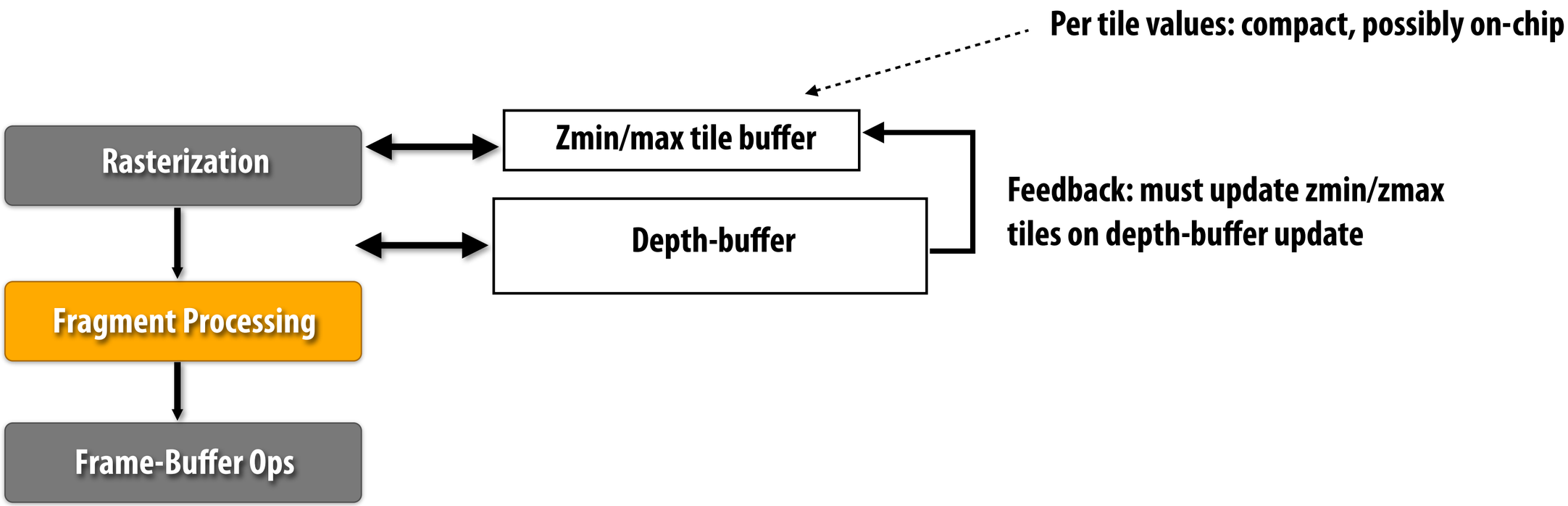
1. **Compute closest point on triangle in tile:** `tri_min` **(using Z plane equation)**
2. **If** `tri_min` > `z_max`**, then triangle is completely occluded in this tile. (The depth test will fail for all samples in the tile.) Proceed to next tile without performing triangle coverage tests for individual samples.**



Z-Min optimization:

Depth-buffer also stores `z_min` **for each tile.**
If `tri_max` < `z_min`**, then all depth tests for fragments in tile will pass. (No need to perform depth test on individual fragments.)**

Hierarchical Z + early Z-culling



Remember: these are GPU implementation optimizations. They are invisible to the programmer and not reflected in the graphics pipeline abstraction

Summary: hierarchical Z

- Idea: perform depth test at coarse tile granularity prior to sampling coverage

- ZMax culling benefits:
  - Reduces rasterization work
  - Reduces depth-testing work (don't process individual depth samples)
  - Reduces depth-buffer bandwidth (don't need to read individual depth samples)
  - Does not reduce fragment processing work more than early Z (hierarchical Z is a conservative optimization: will discard a subset of the fragments early Z does)

- ZMin benefits:
  - Reduces depth-testing work (don't need to test individual depth samples)
  - Reduces depth-buffer bandwidth (don't need to read individual samples, but still must write)

- Costs:
  - Overhead of hierarchical tests
  - Must maintain per-tile Zmin/Zmax values
  - Complex: must update per-tile values frequently to be effective (early Z system feeds results back to hierarchical Z system)

Fast Z clear

- Formerly an important optimization: less important in modern GPU architectures
  - Add "cleared" bit to tile descriptor
  - glClear(GL_DEPTH_BUFFER) sets these bits
  - First write to depth sample in tile clears bit

- Benefits
  - Reduces depth-buffer write bandwidth: avoid frame-buffer write on frame-buffer clear
  - Reduces depth-buffer read bandwidth by skipping first read: if "cleared" bit for tile set, GPU can initialize tile's contents in cache without reading data (a form of lossless compression)

Frame-buffer compression

Depth-buffer compression

- Motivation: reduce bandwidth required for depth-buffer accesses
  - Worst-case (uncompressed) buffer allocated in DRAM
  - Conserving memory footprint is a non-goal

    (Need for real-time guarantees in graphics applications requires application to plan for worse case anyway)

- Lossless compression
  - Q. Why not lossy?

- Designed for fixed-point numbers (fixed-point math in rasterizer)

Depth-buffer compression is tile-based

- Main idea: exploit locality of values within a screen tile

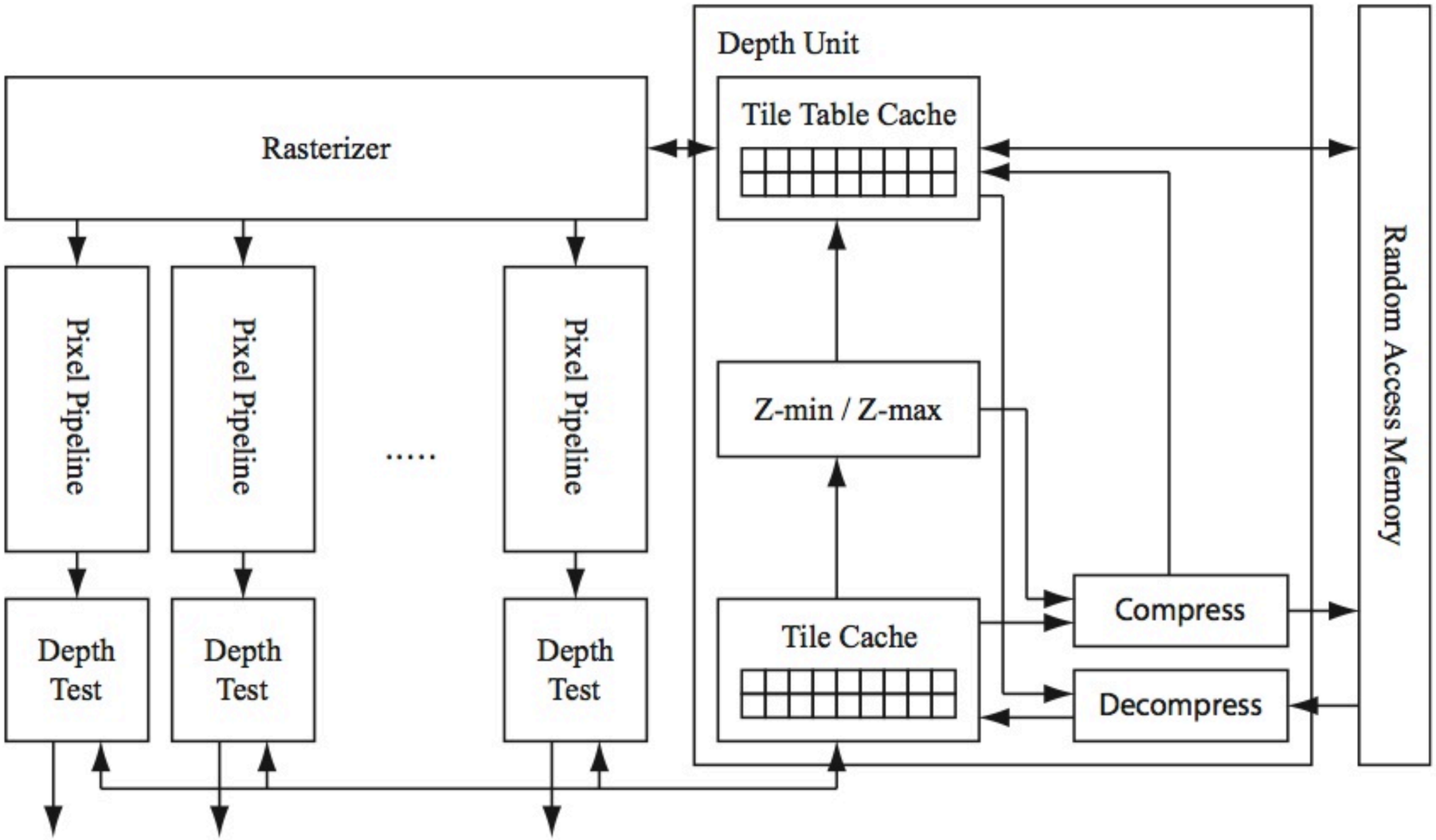


Figure credit: [Hasselgren et al. 2006]

On tile evict:

1. Compute zmin/zmax (needed for hierarchical culling and/or compression)
2. Attempt to compress
3. Update tile table
4. Store tile to memory

On tile load:

1. Check tile table for compression scheme
2. Load required bits from memory
3. Decompress into tile cache

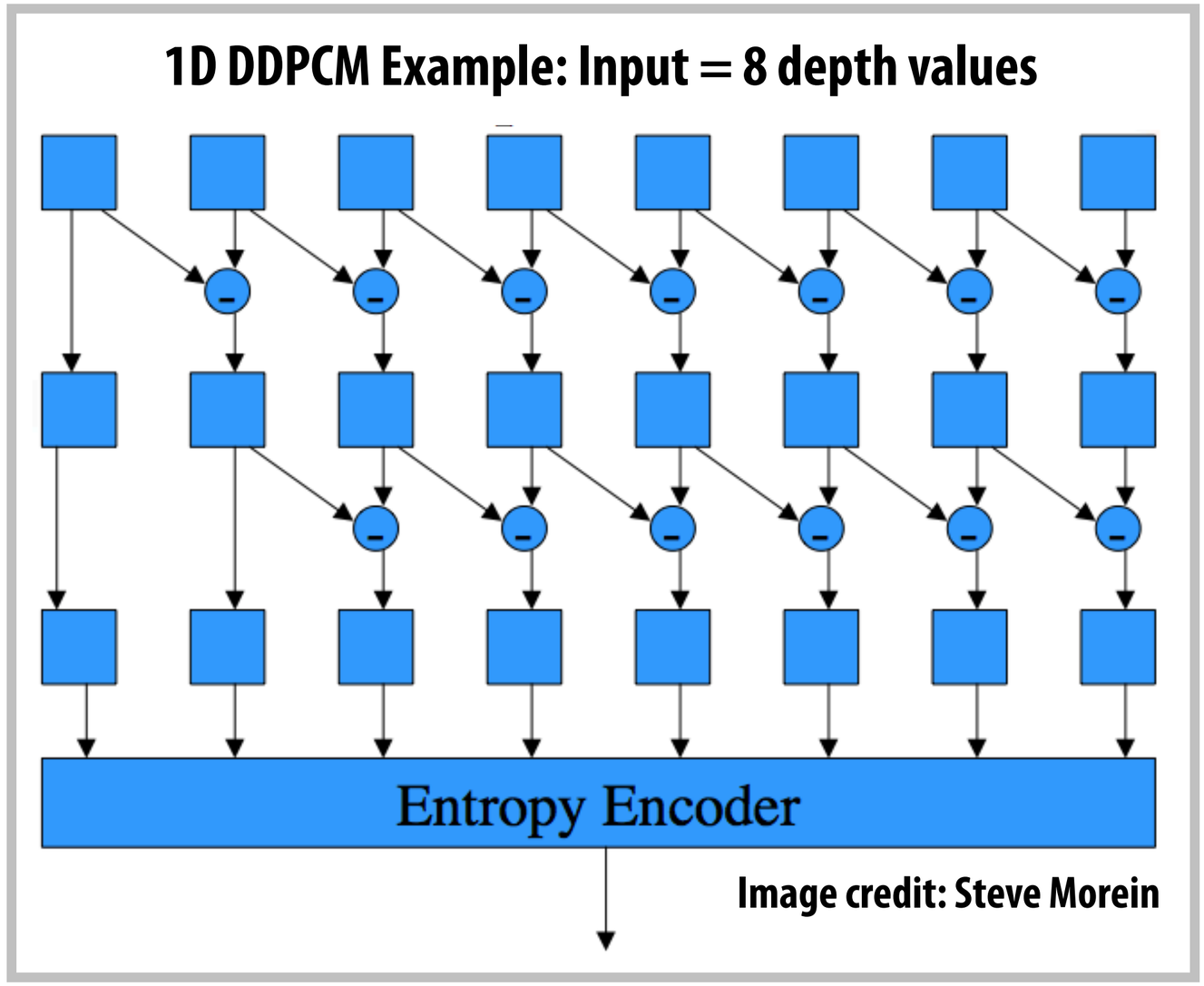
DDPCM: differential pulse code modulation

- Recall: z/w is interpolated linearly in screen space
- Compute first and second-order differentials for tile data

- Example scheme:

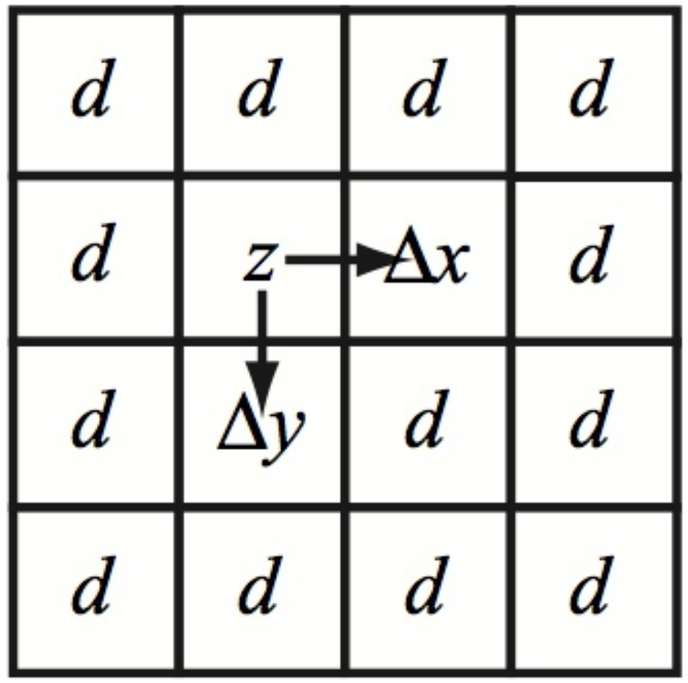
  For an 8x8 sample tile, store:
  - 32 bit reference value
  - 2 x 33 bit DX and DY values
  - 61 x 2 bit {-1,0,1} values for 2nd order differentials
  - 220 bits per 64-sample tile

    (round up to 256 yields 8-to-1 ratio vs uncompressed 8x8x32 bits)



Anchor encoding

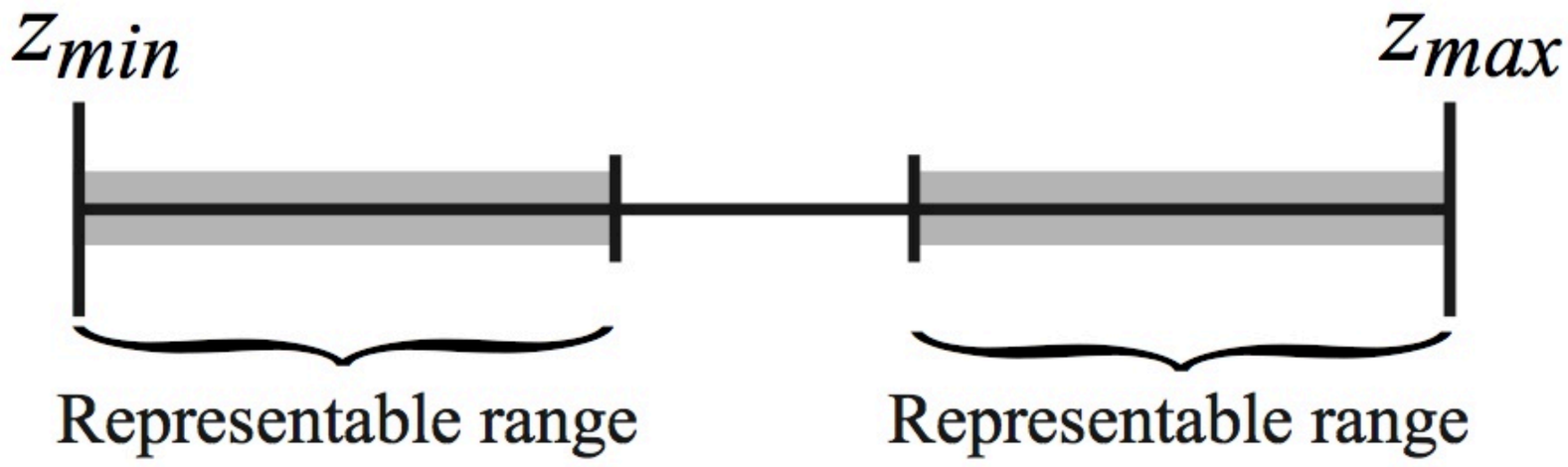
- Choose anchor value and compute DX, DY from adjacent pixels (fits a plane to the data, similar to DDCPM)
- Use plane to predict depths at other pixels, store offset d from prediction at each pixel
- Scheme (for 24-bit depth buffer)
  - Anchor: 24 bits (full resolution)
  - DX, DY: 15 bits
  - Per-sample offsets: 5 bits



[Van Dyke and Margeson]

Depth-offset compression

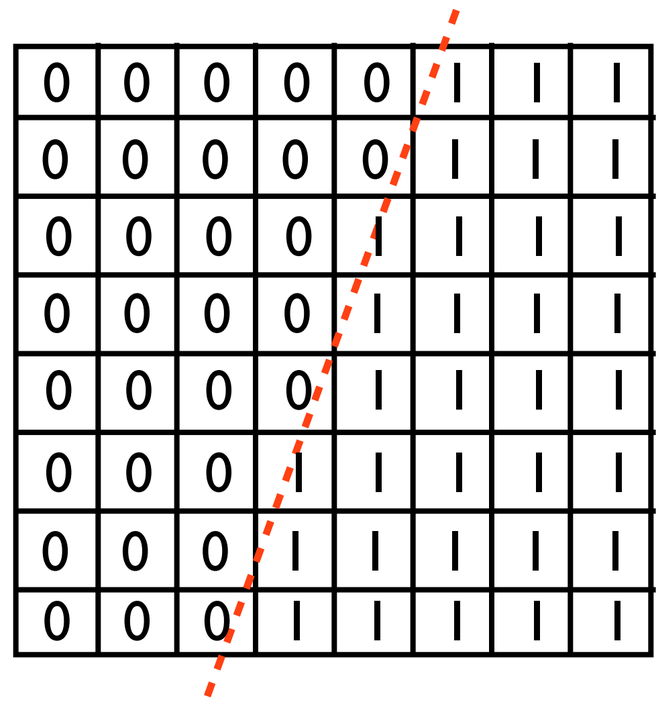
- Assume depth values have low dynamic range relative to tile's zmin and zmaz (assume two surfaces)
- Store zmin/zmax (needed to anyway for hierarchical Z)
- Store low-precision (8-12 bits) offset value for each sample
  - MSB encodes if offset is from zmin or zmax



[Morein and Natali]

Plane-encoding

- Do not attempt to infer prediction plane, just get the plane equation directly from the rasterizer
  - Store plane equation in tile (values must be stored with high precision: to match exact math performed by rasterizer)
  - Store bit per sample indicating coverage
- Simple extension to multiple triangles per tile:
  - Store up to N plane equations in tile
  - Store $\log_2(N)$ bit id per depth sample indicating which triangle it belongs to
- When new triangle contributes coverage to tile:
  - Add new plane equation if storage is available, else decompress
- To decompress:
  - For each sample, evaluate Z(x,y) for appropriate plane



Summary: depth-buffer bandwidth reduction

- Tile caching: access memory less
- Hierarchical Z techniques (zmin/zmax culling): access individual samples less
- Data compression: reduce number of bits that must be read from memory

---

- Color buffer is also compressed using similar techniques
  - Depth-buffer typically achieves higher compression ratios than color buffer. Why?

Visibility/culling relationships

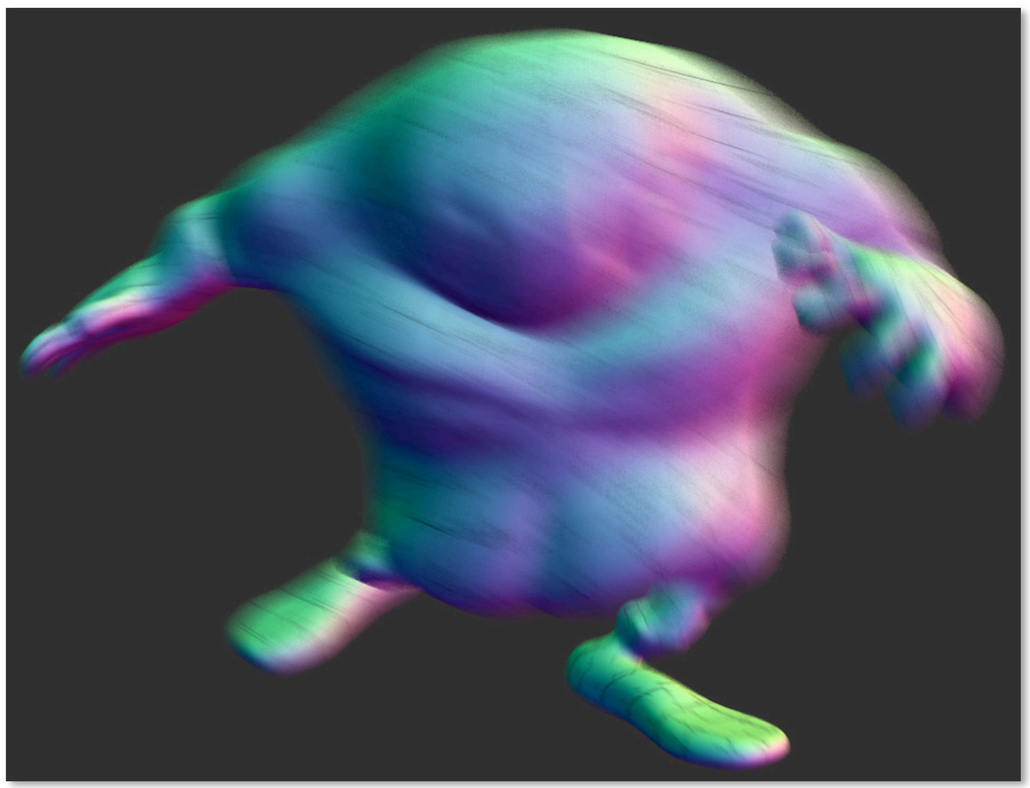
- Hierarchical traversal during rasterization
  - Leveraged to reduce coverage testing and occlusion work
  - Tile size likely coupled to hierarchical Z granularity
  - May also be coupled to compression tile granularity

- Hierarchical culling and plane-based buffer compression are most effective when triangles are reasonably large (recall triangle size discussion in lecture 2)

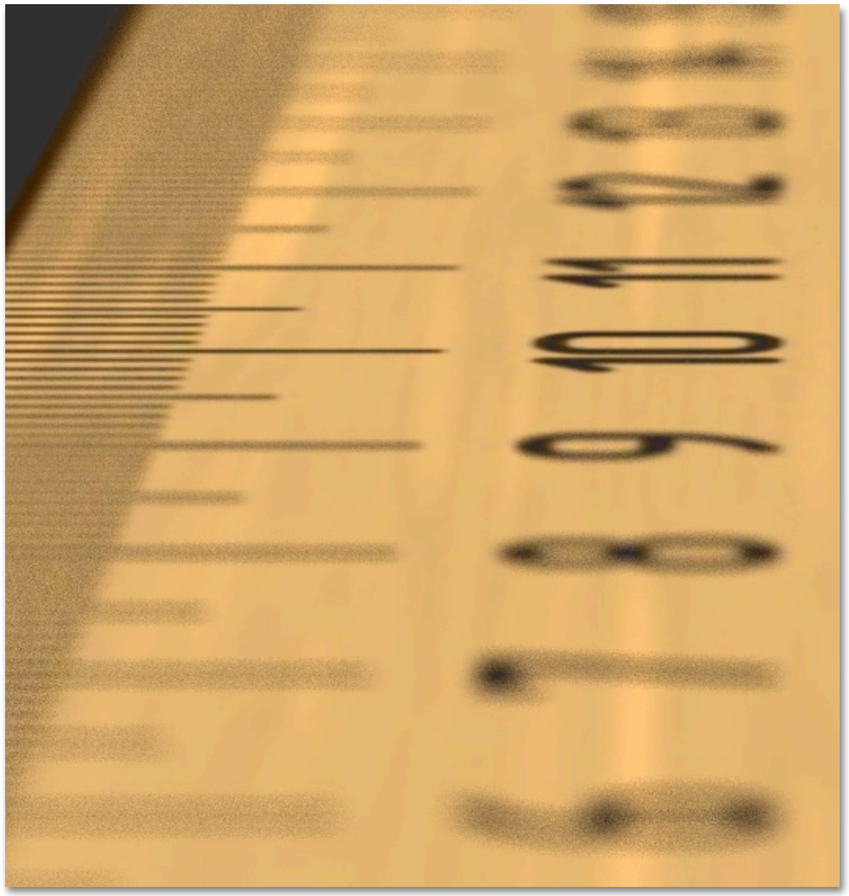
Stochastic rasterization

- **Accurate camera simulation in real-time rendering**
  - Visibility algorithms discussed today simulate image formation by virtual pinhole camera, with infinite shutter
  - Real cameras have finite apertures, finite exposure duration
  - Visibility computation requires integration over time and lens aperture (high computational cost + diminished spatial coherence)

Time integration: motion blur



Lens aperture integration: defocus blur



Readings

- M. Abrash, Rasterization on Larrabee, Dr. Dobbs Portal. May 1, 2009

- A. R. Smith, *A Pixel is Not a Little Square, a Pixel is Not a Little Square, a Pixel is Not a Little Square!* (and a Voxel is Not a Little Cube) Microsoft Technical Memo, 1995