Lecture 18:

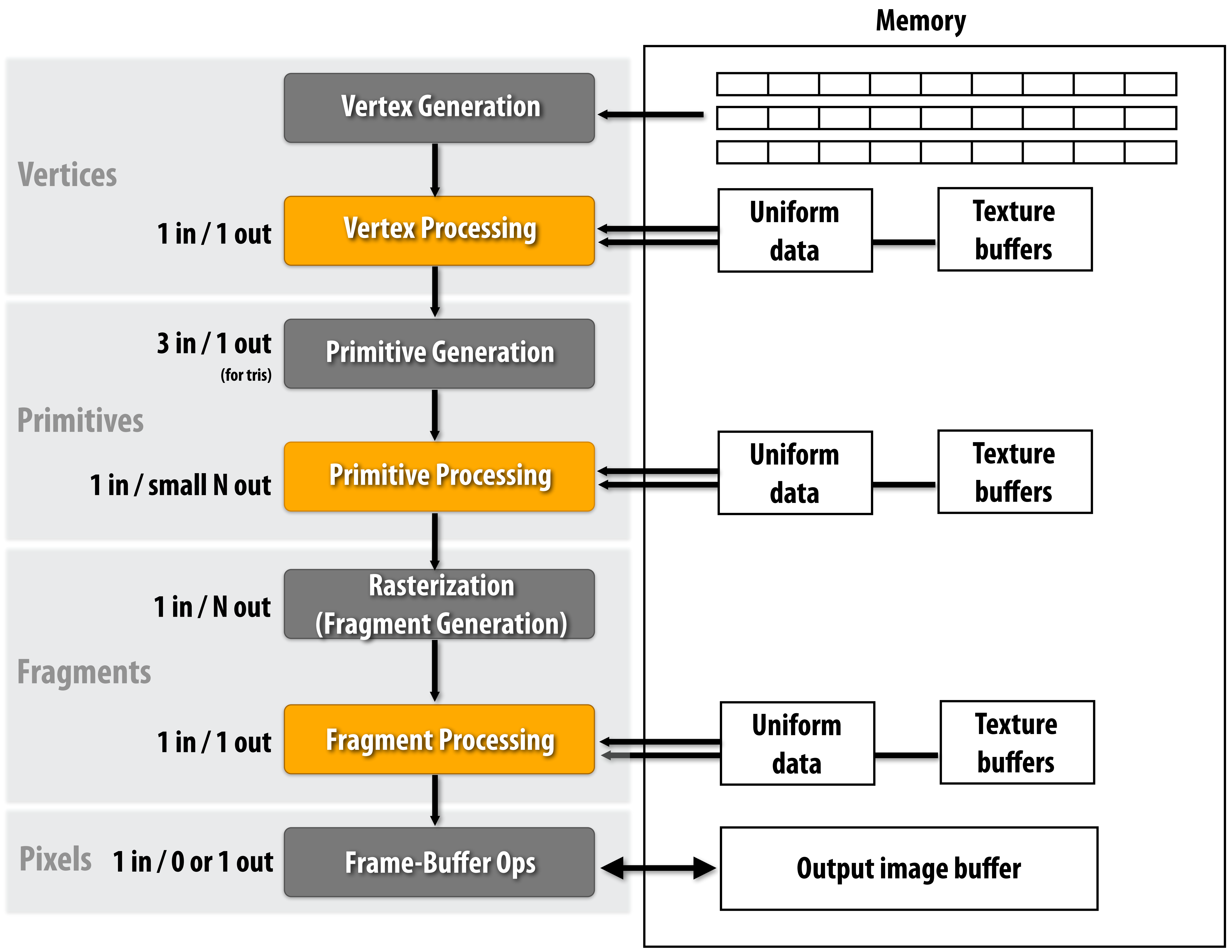
Visibility

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

Visual Computing Systems
CMU 15-769, Fall 2016

Last time: graphics pipeline architecture

Interface to GPU used for real-time 3D graphics applications



Surprising to some: what the pipeline architecture <u>does not</u> have primitives for

- **Modern graphics pipeline has no concept of lights, materials, geometric modeling transforms**
  - Only streams of records processed by application defined kernels: vertices, primitives, fragments, pixels
  - And pipeline state (input/output buffers, "shaders", and fixed-function configuration parameters)
  - Applications implement lights, materials, etc. using these basic abstractions

- **The graphics pipeline has no concept of a scene**

- **Just a machine that executes pipeline state change and draw primitives commands**

Last time: programming the graphics pipeline

Issue draw commands $\longrightarrow$ output image contents change

Command Type	Command
State change	Bind shaders, textures, uniforms
Draw	Draw using vertex buffer for object 1
State change	Bind new uniforms
Draw	Draw using vertex buffer for object 2
State change	Bind new shader
Draw	Draw using vertex buffer for object 3
State change	Change depth test function
State change	Bind new shader
Draw	Draw using vertex buffer for object 4

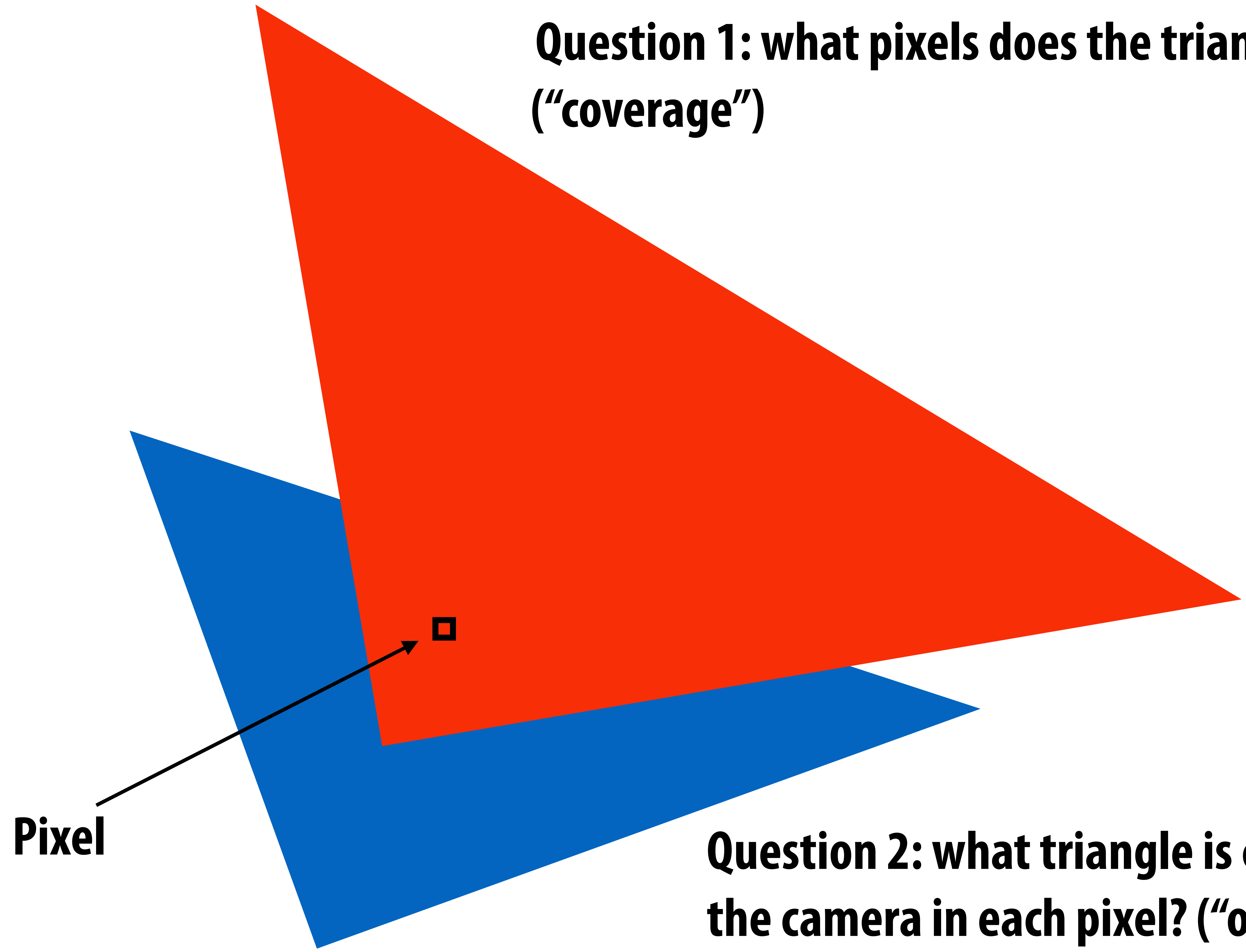
Major graphics-specific services of pipeline

- Efficiently implementing visibility (generating fragments from primitives in the "fragment generation" stage) and occlusion ("pixel ops stage")
  - Today's topic
  - Implemented in fixed-function hardware on modern GPU

- Providing efficient implementation of texture mapping and tessellation
  - Future topics

- Scheduling these operations and application-provided kernels efficiently on heterogeneous, parallel GPU hardware

Let's draw some triangles on the screen

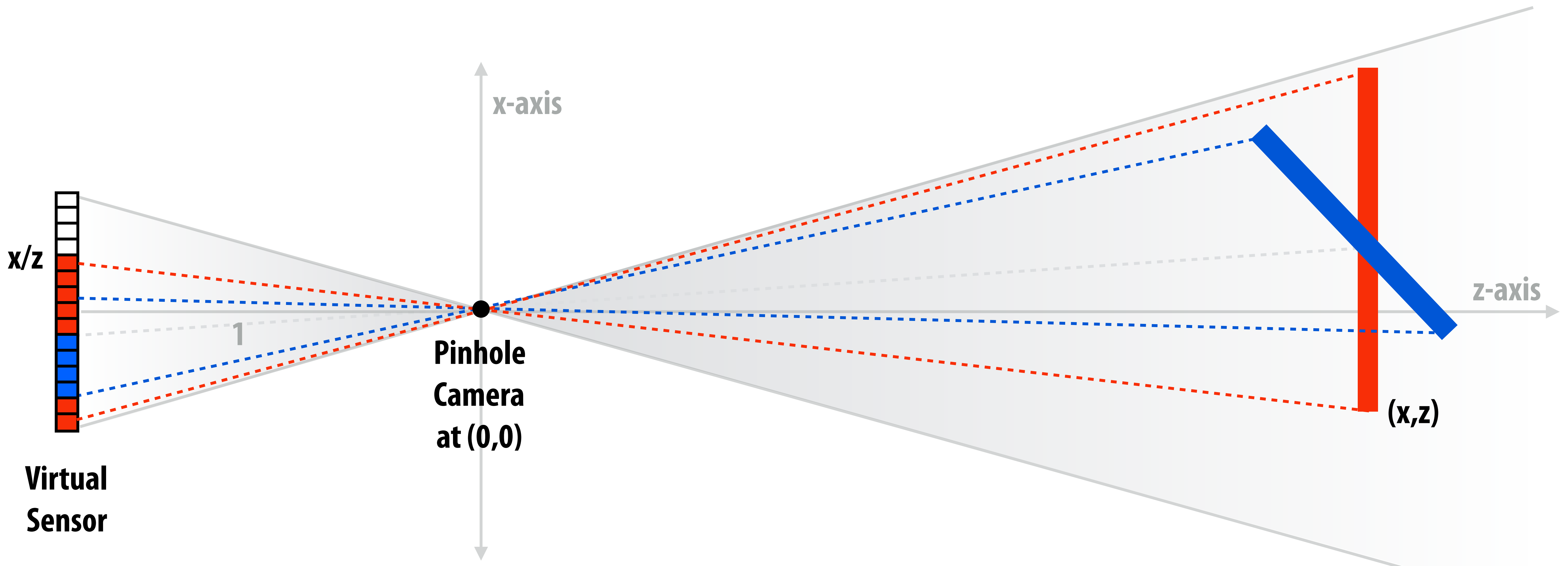


Question 1: what pixels does the triangle overlap? ("coverage")

Question 2: what triangle is closest to the camera in each pixel? ("occlusion")

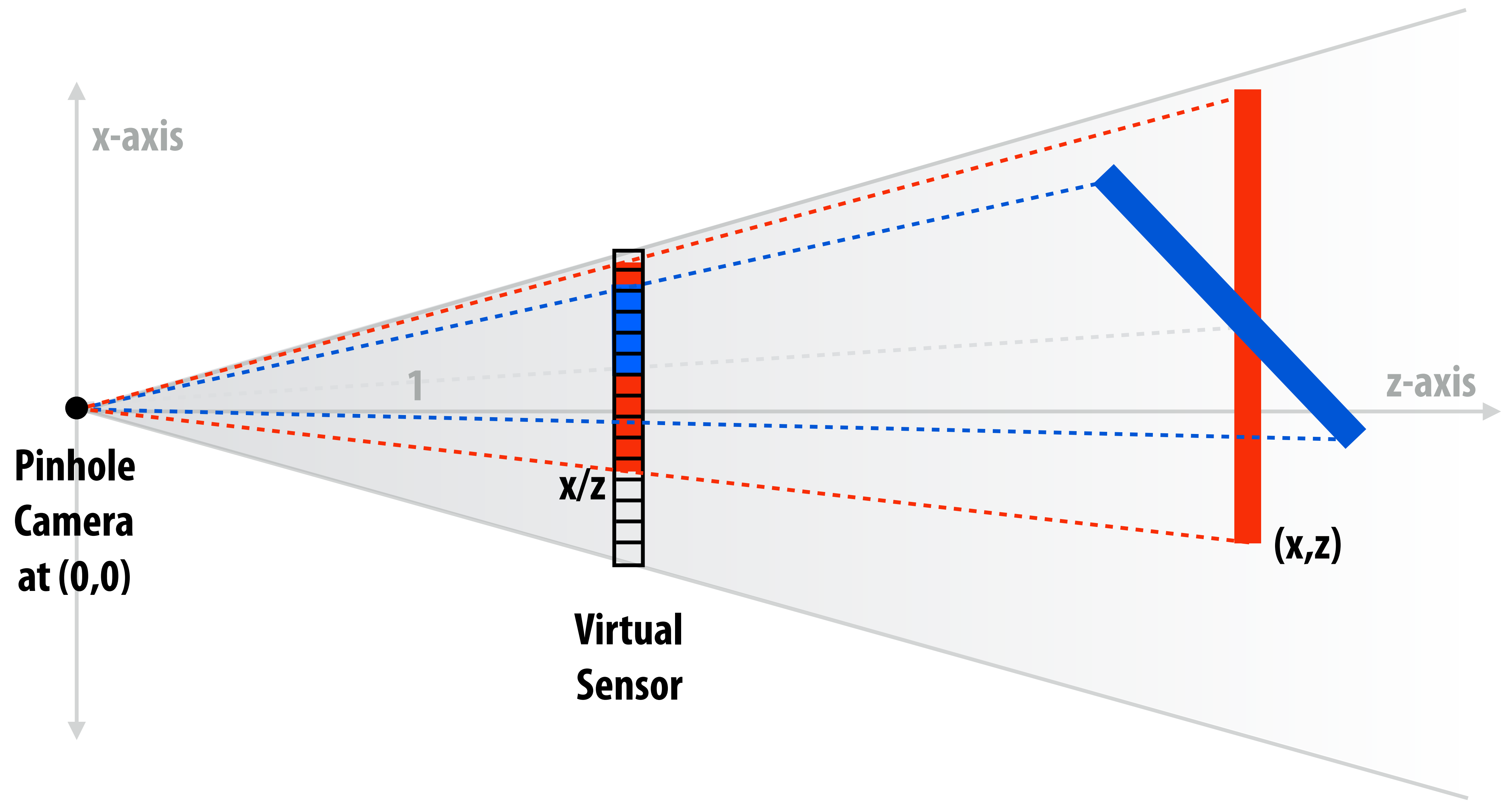
The visibility problem

- An informal definition: what scene geometry is visible within each screen pixel?
    - What scene geometry projects into a screen pixel? (coverage)
    - Which geometry is visible from the camera at that pixel? (occlusion)



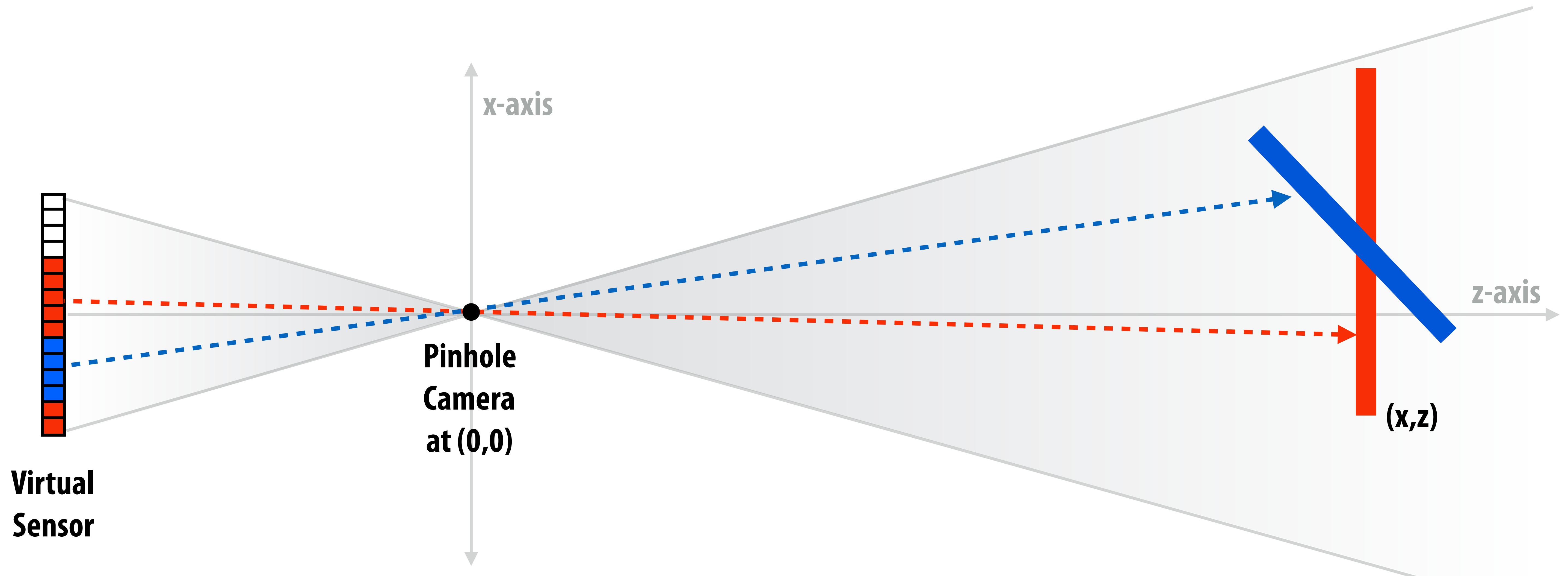
The visibility problem

- An informal definition: what scene geometry is visible within each screen pixel?
  - What scene geometry projects into a screen pixel? (coverage)
  - Which geometry is visible from the camera at that pixel? (occlusion)



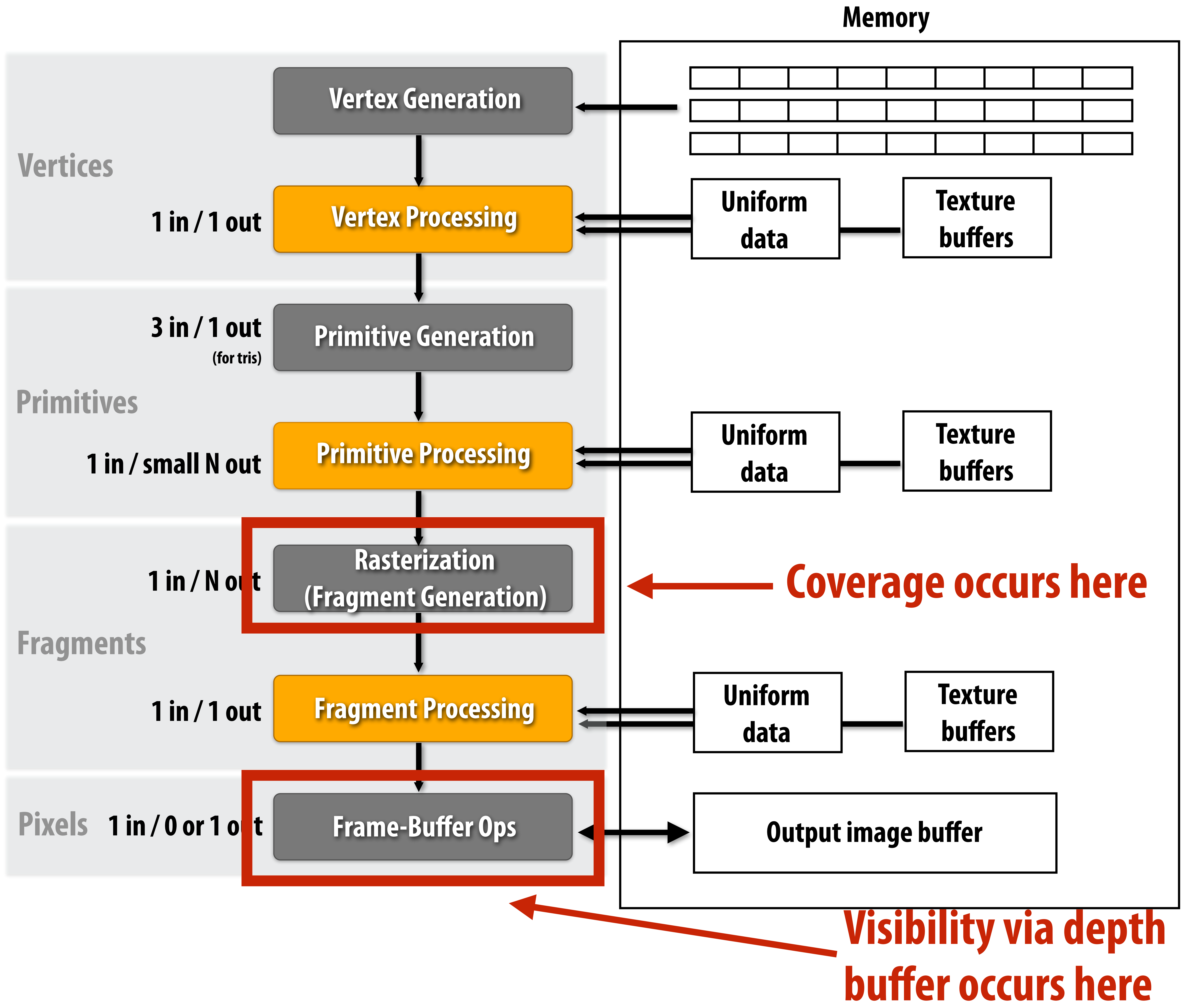
The visibility problem (said differently)

- In terms of light rays:
  - What scene geometry is hit by a ray from a pixel through the pinhole? (coverage)
  - What object is the first hit along that ray? (occlusion)



Hold onto this thought for later in the semester.

Visibility: coverage + occlusion



Today's main ideas

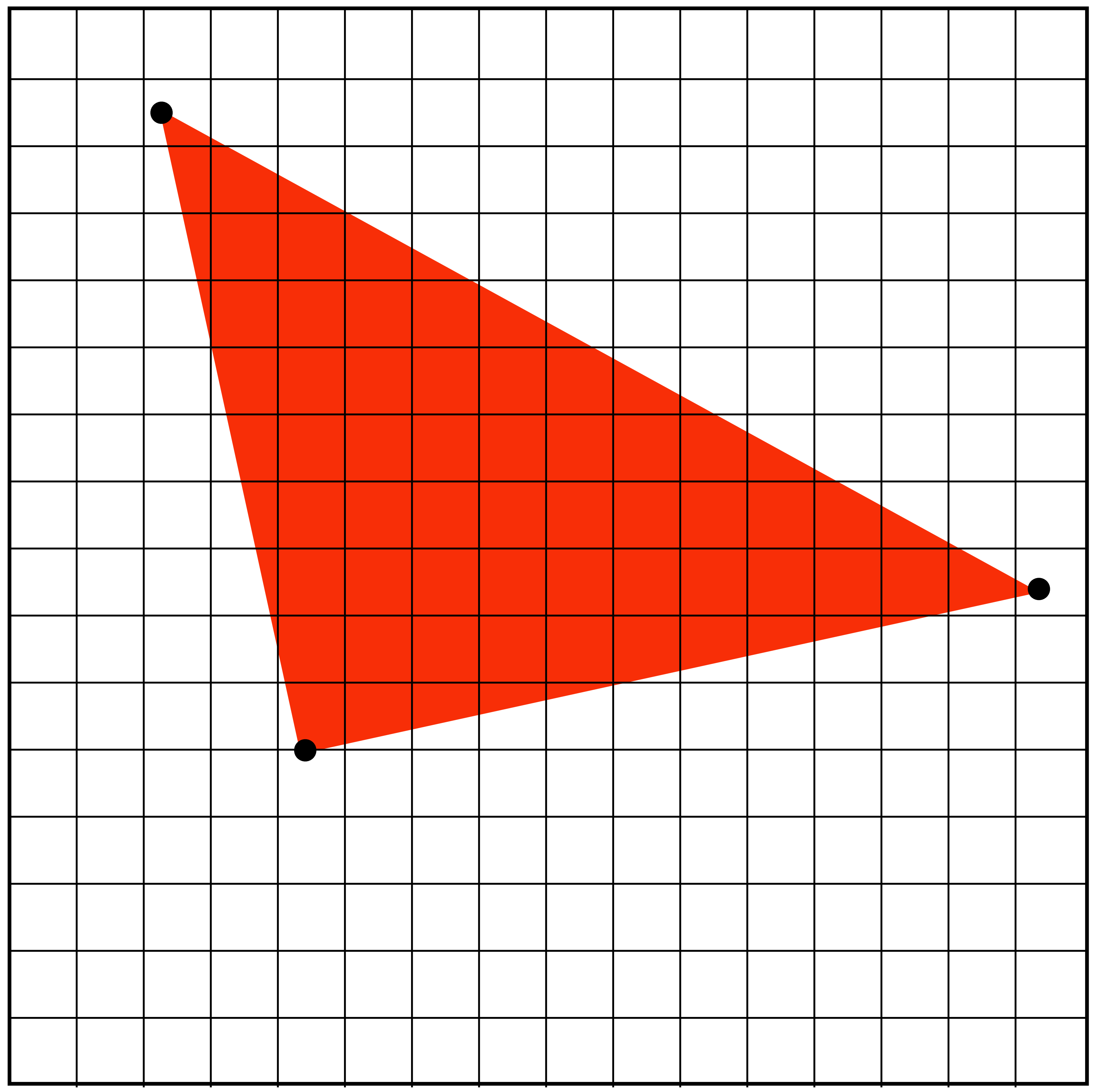
- **Hardware-friendly visibility algorithms**
  - **Tension between work-efficient (but serial) and "brute force" wide data parallel methods**
  - **Hierarchical techniques (per tile, per pixel) pay off in multiple context (save work, basis for compression, …)**

- **Hardware optimized data compression**
  - **Domain-specific data compression techniques to alleviate bandwidth bottleneck**

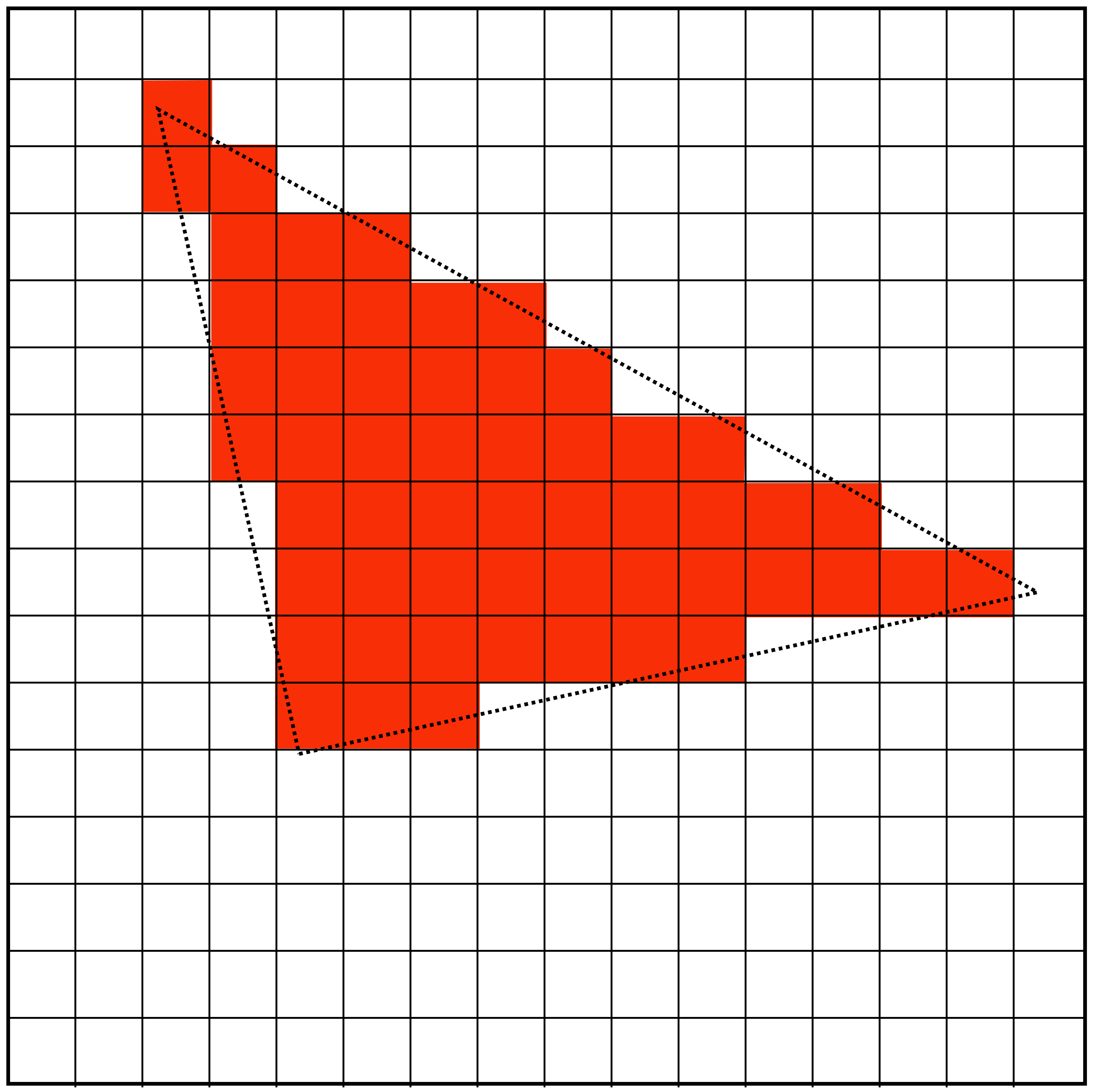
Computing triangle coverage

What pixels does the triangle overlap?

Input:
projected position of triangle vertices: P_0, P_1, P_2

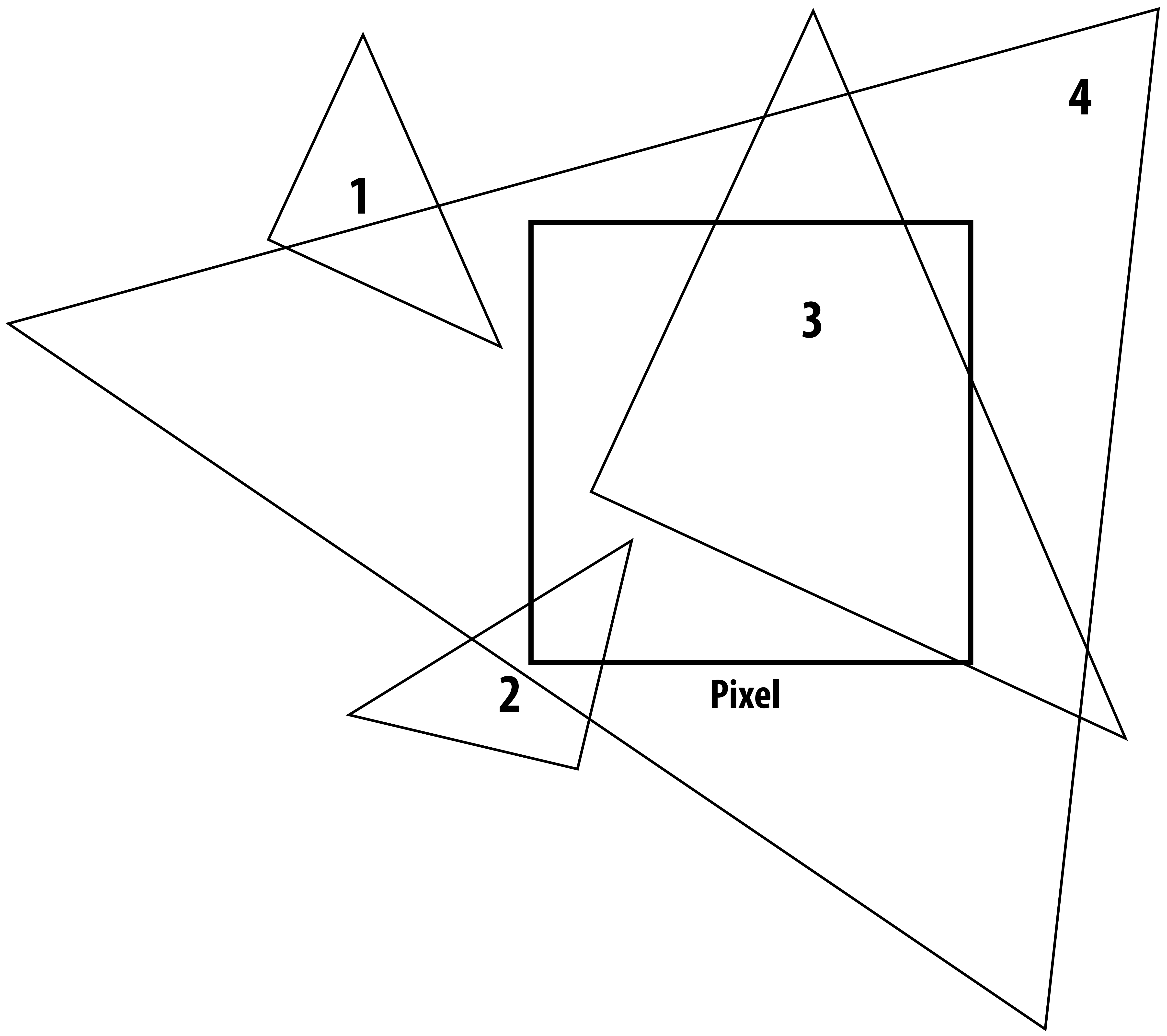


Output:
set of pixels "covered" by the triangle

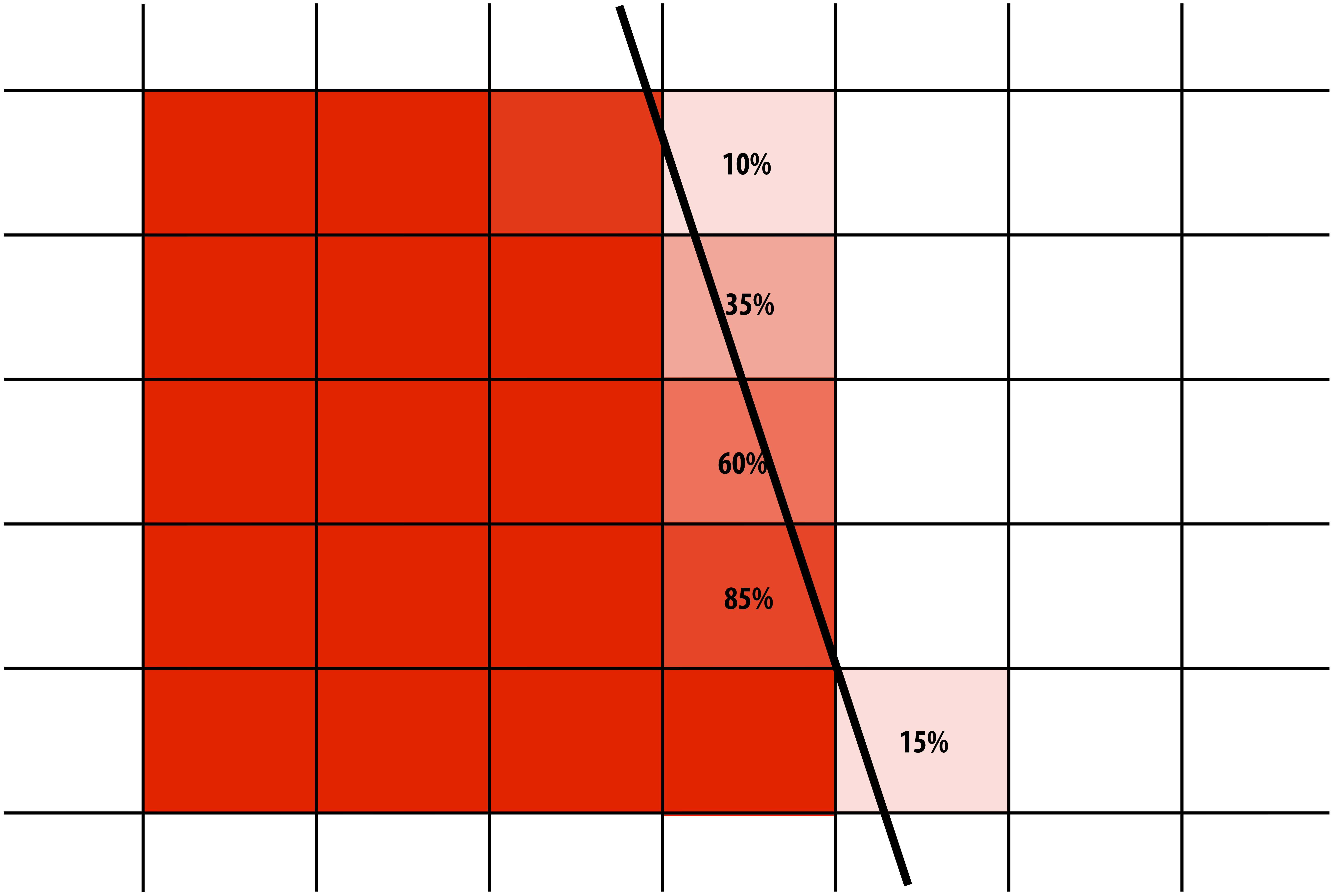


What does it mean for a pixel to be covered by a triangle?

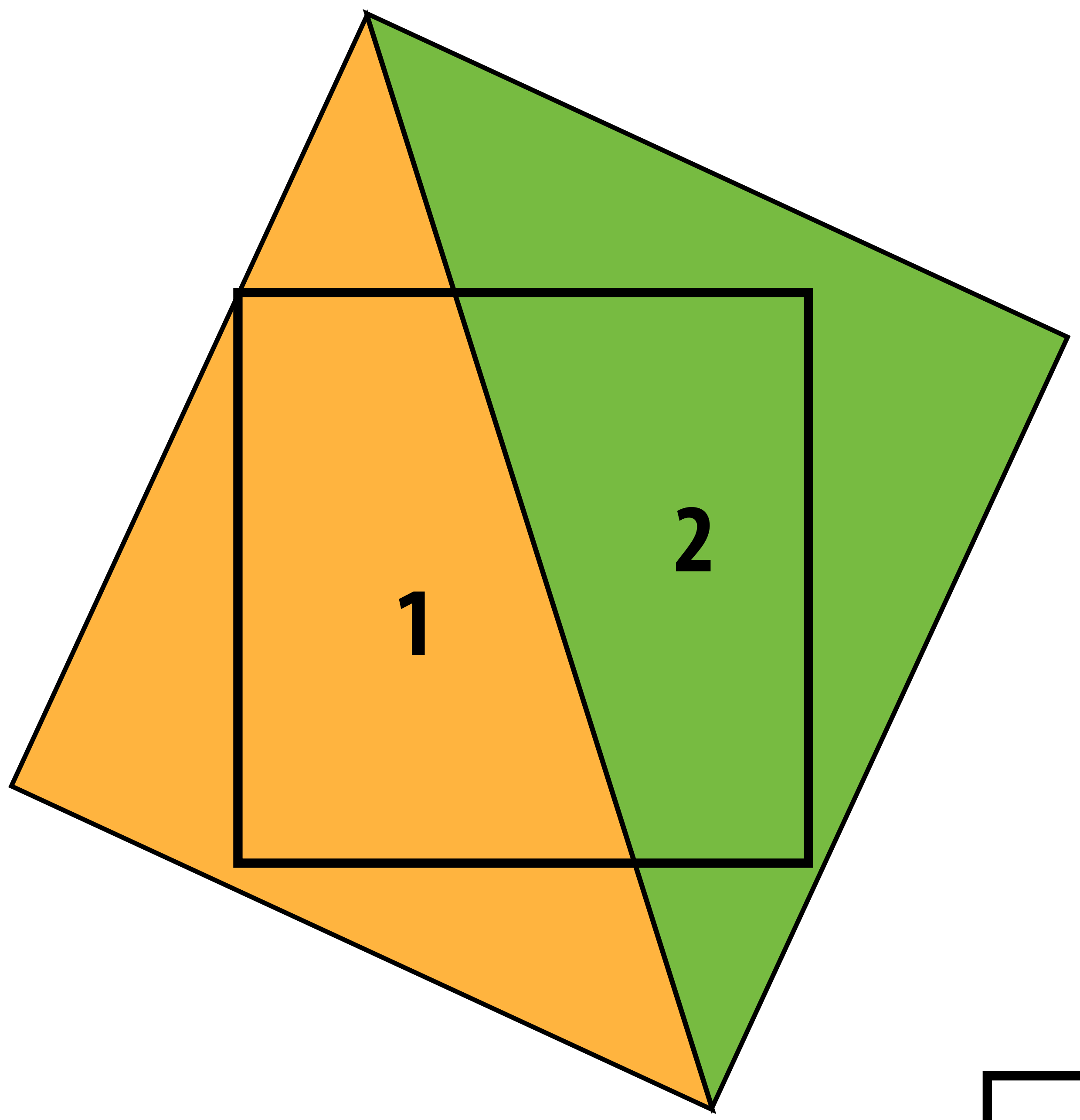
Question: which triangles "cover" this pixel?

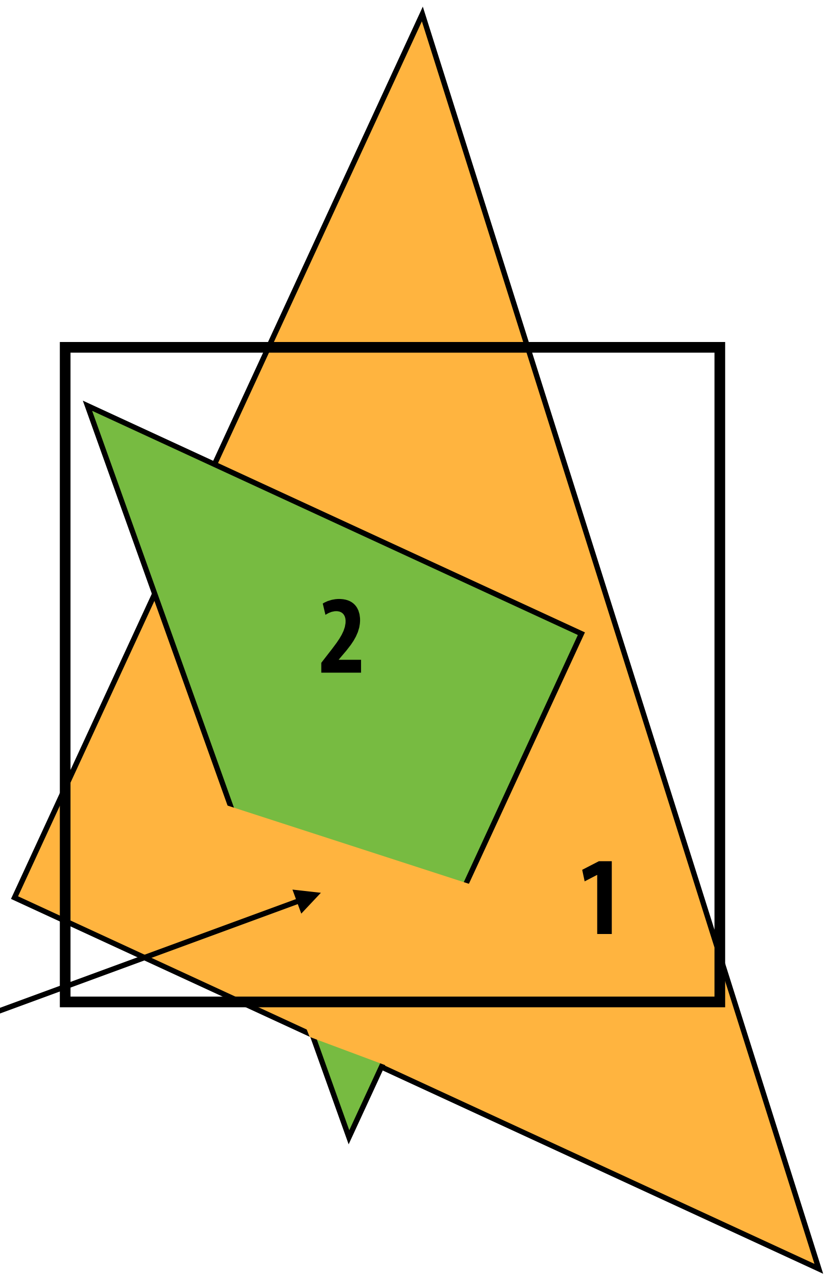


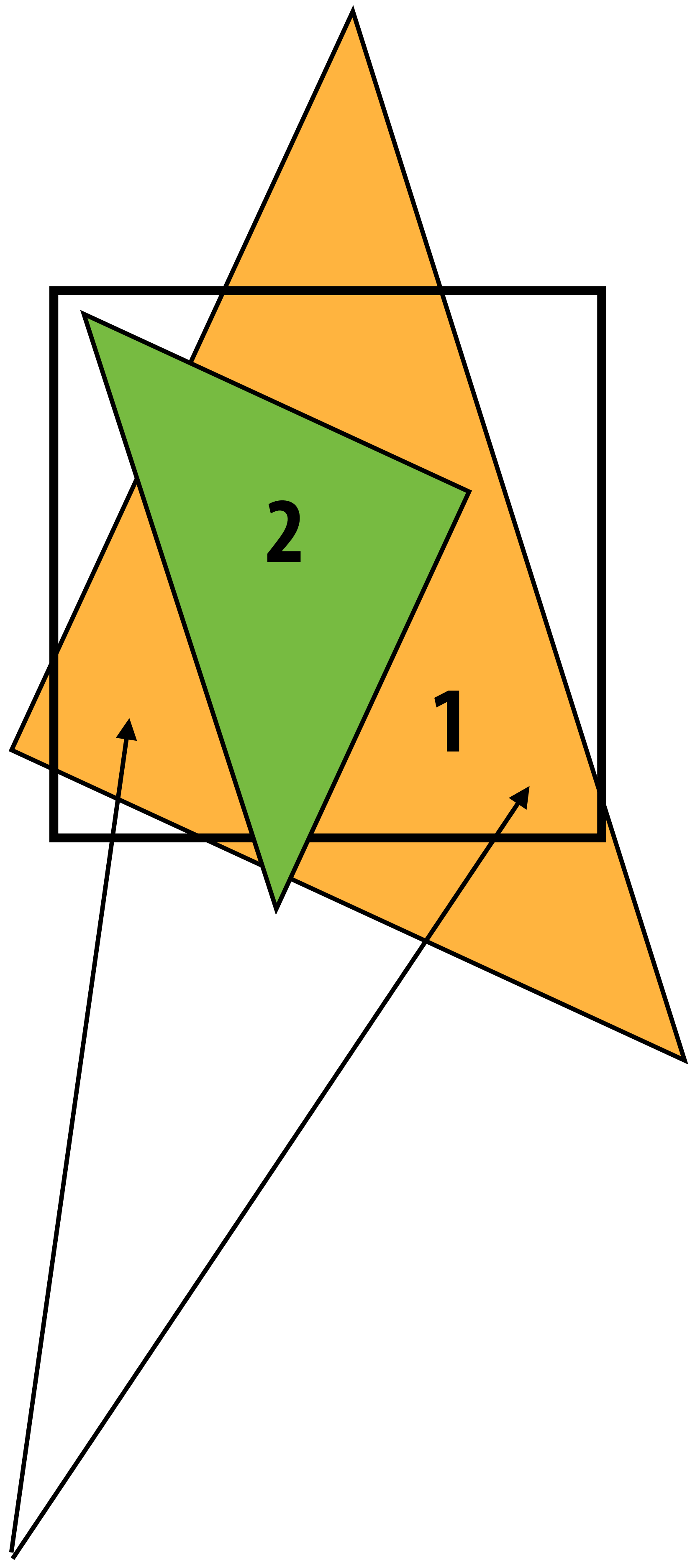
One option: analytically compute fraction of pixel covered by triangle



Analytical schemes get tricky when considering occlusion







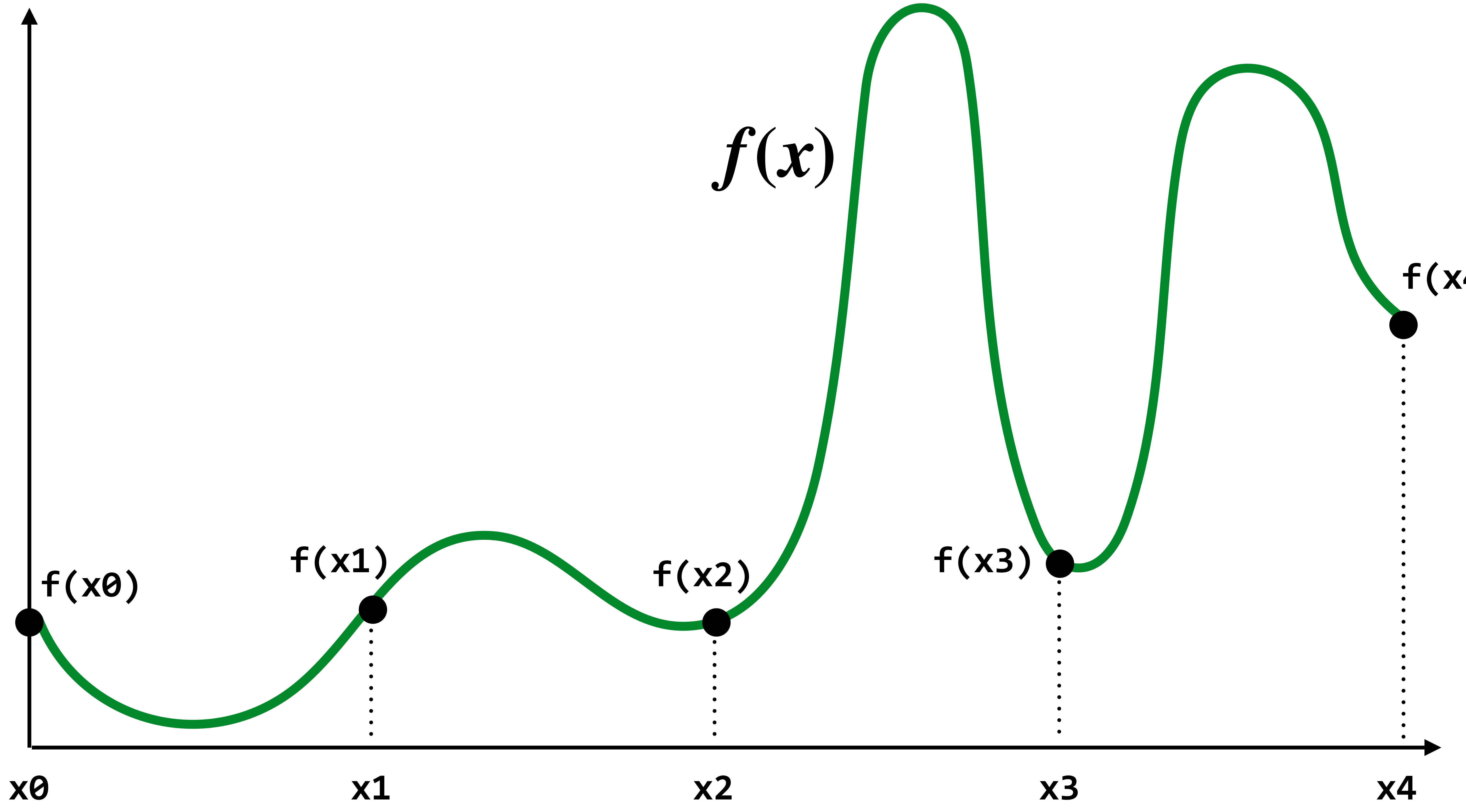
Interpenetration: even worse

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

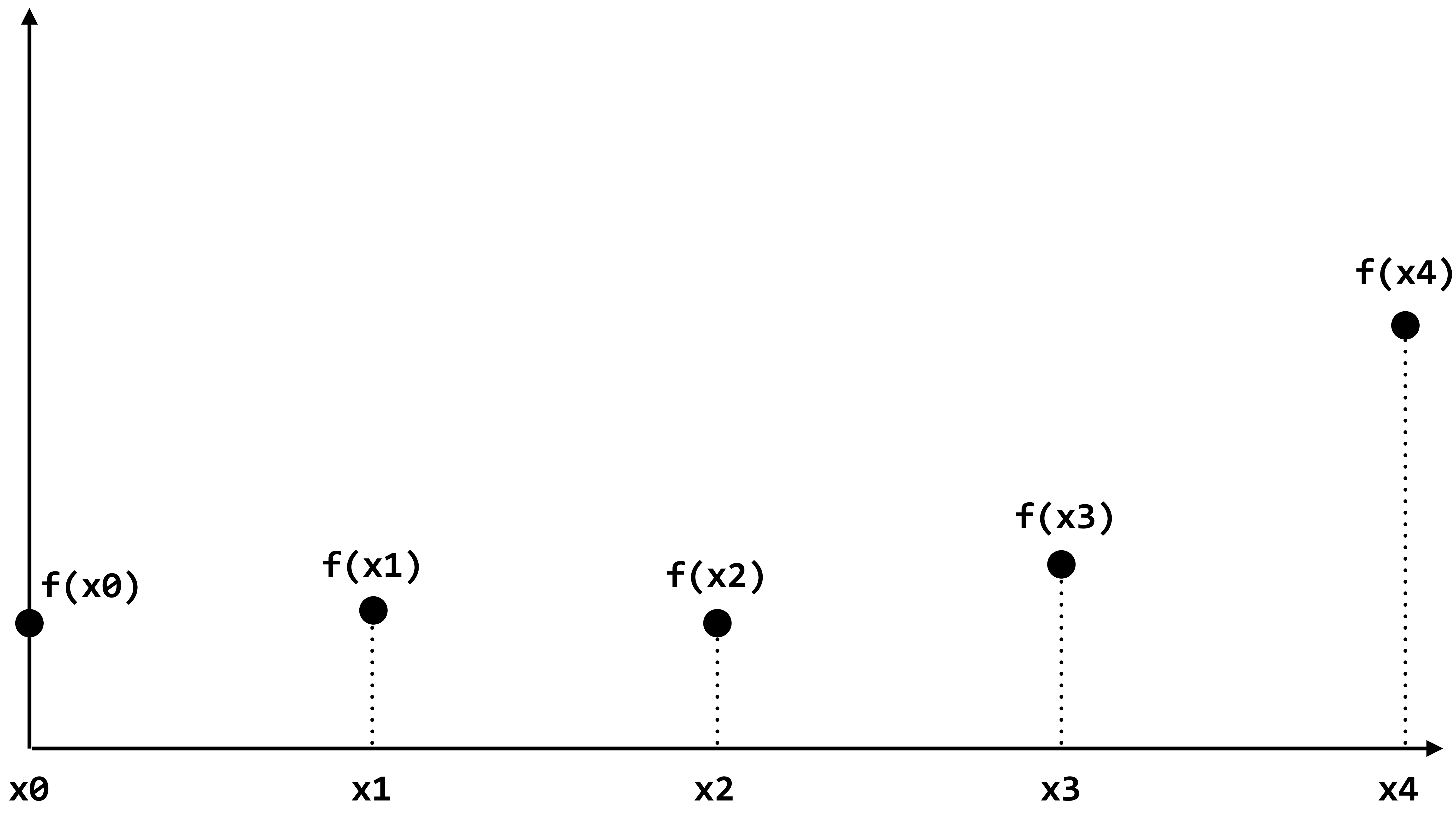
Sampling 101

Sampling: taking measurements a signal

Below: five measurements ("samples") of 1D signal $f(x)$



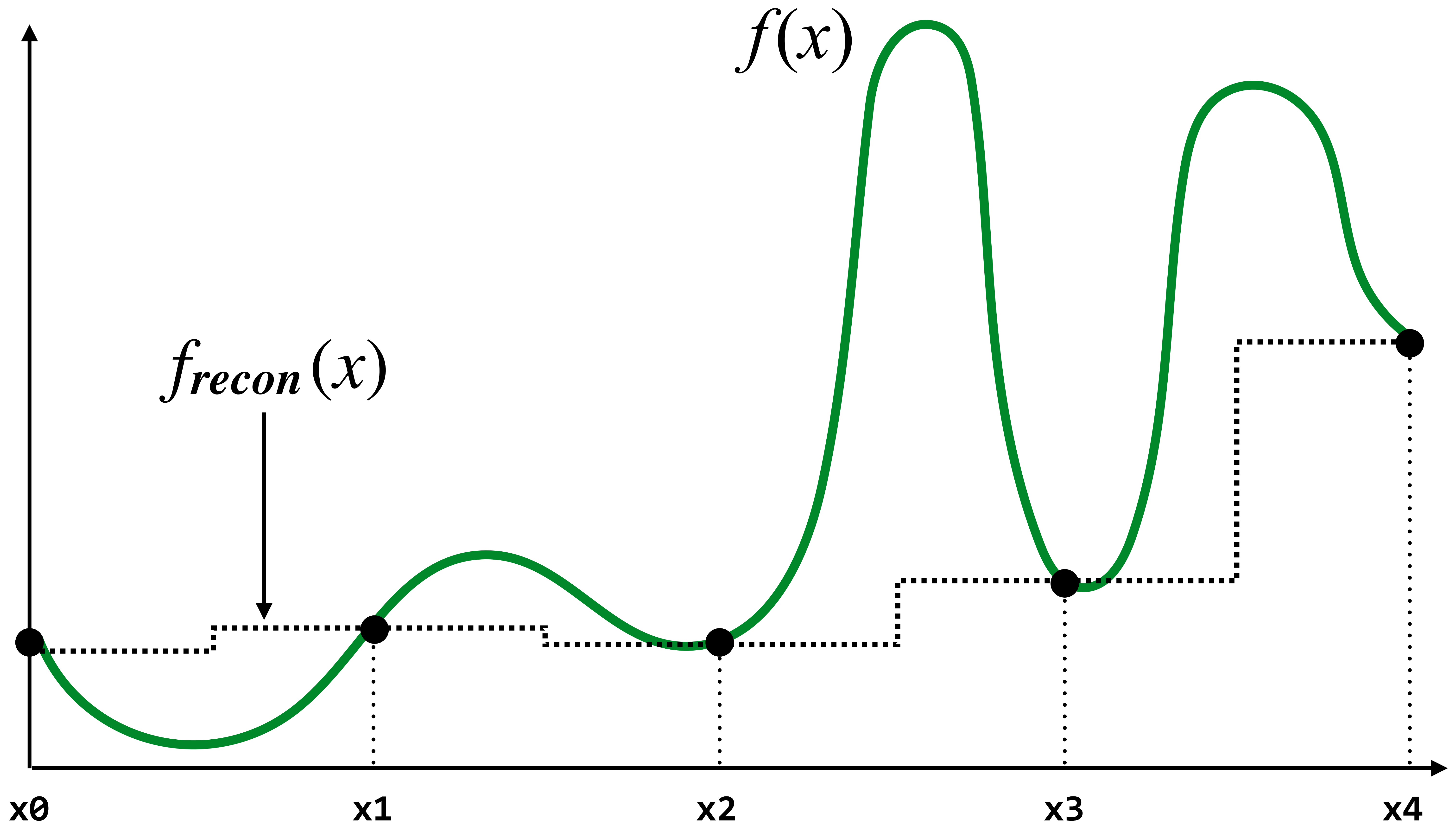
Reconstruction: given a set of samples, how might we attempt to reconstruct the original signal $f(x)$?



Piecewise constant approximation

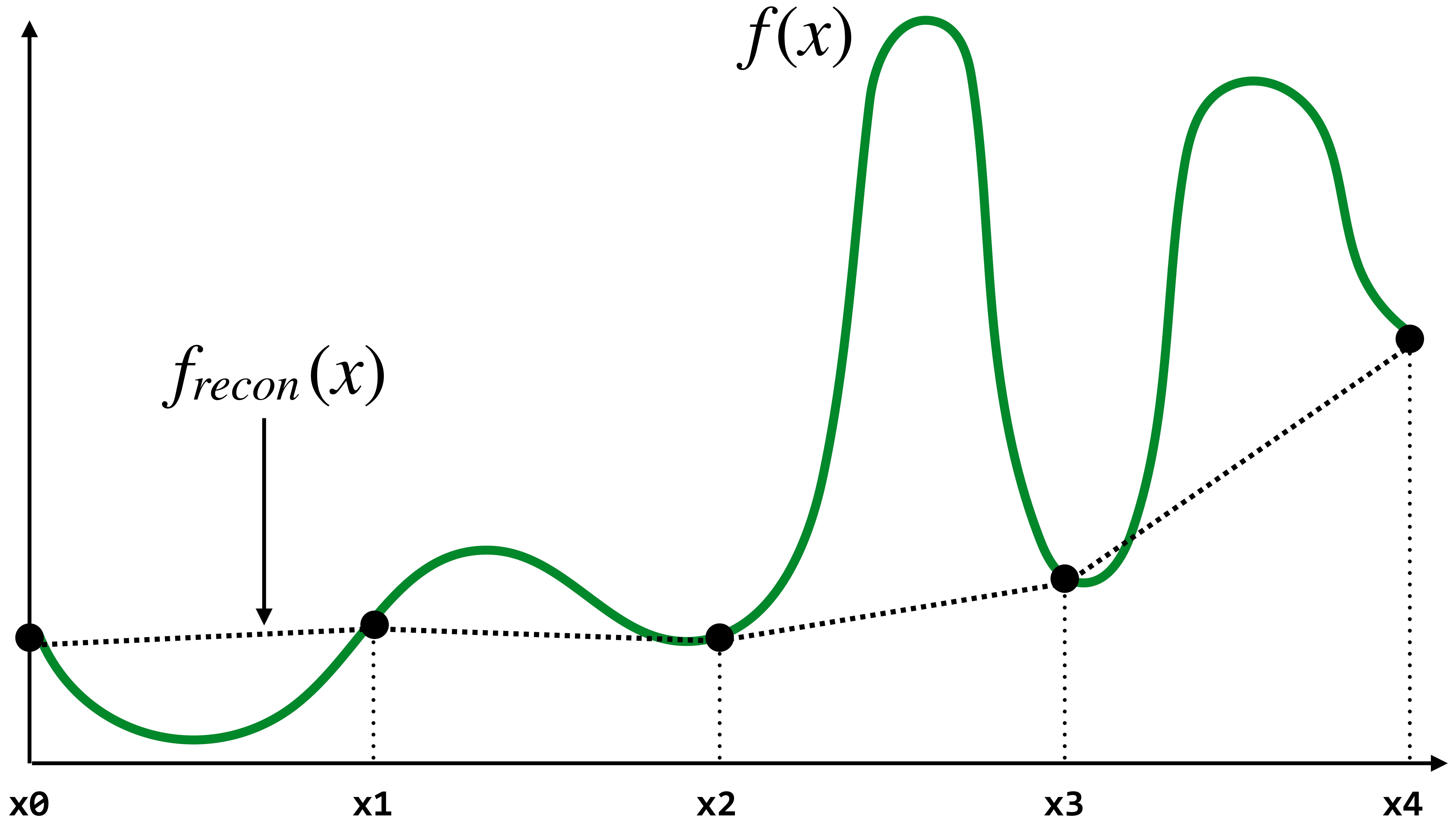
$f_{recon}(x)$ = value of sample closest to x

$f_{recon}(x)$ approximates $f(x)$

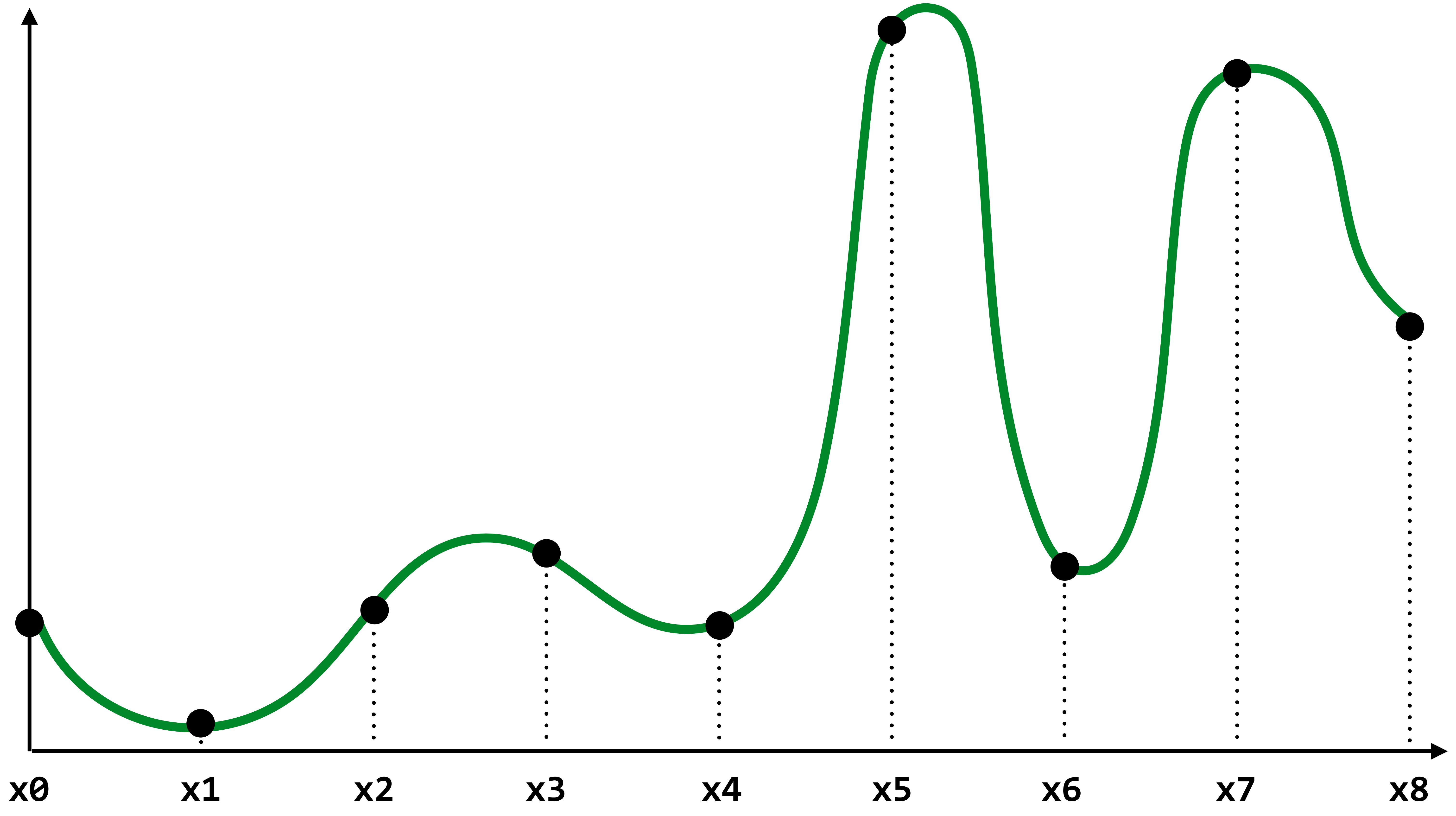


Piecewise linear approximation

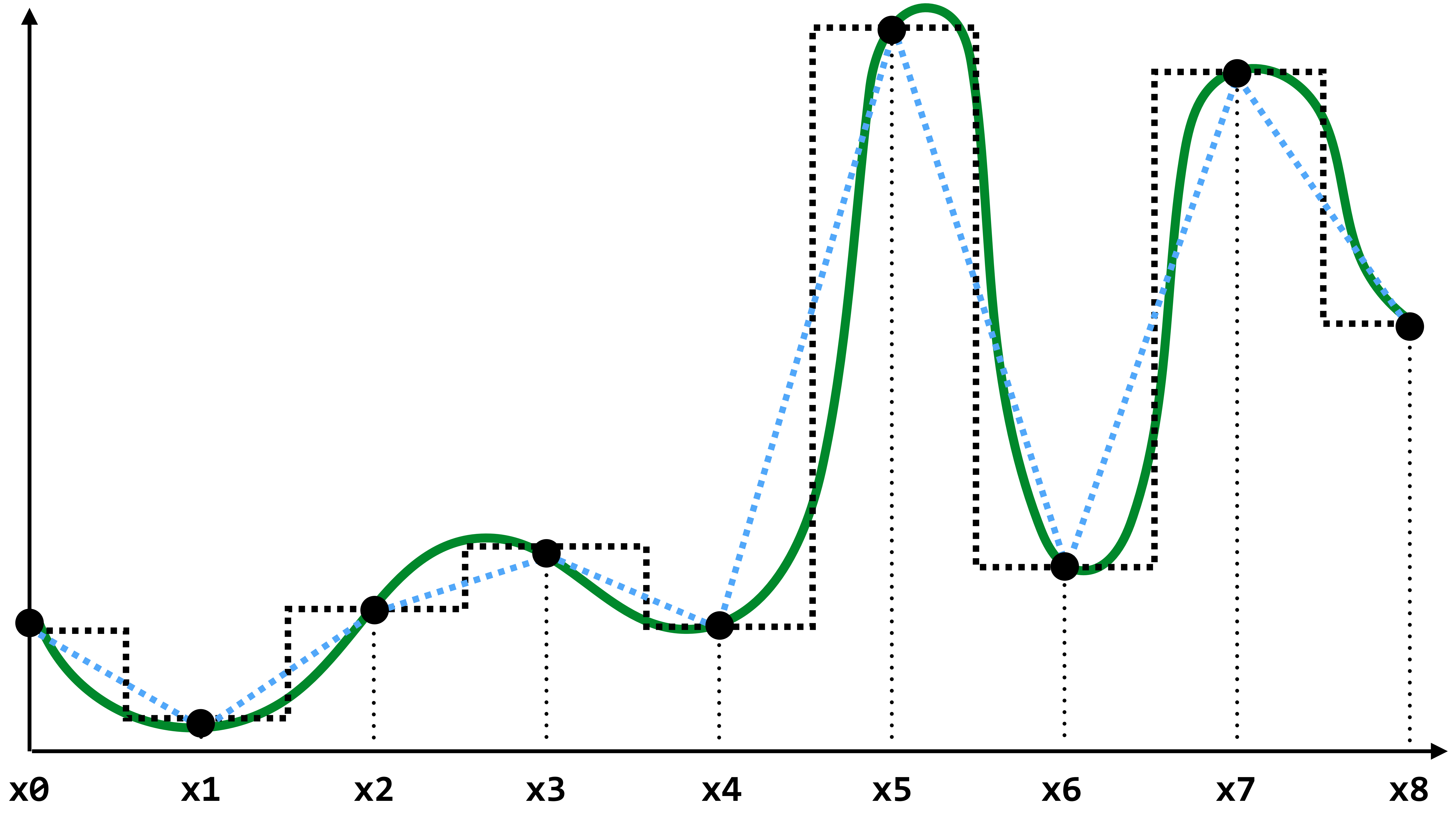
$f_{recon}(x)$ = linear interpolation between values of two closest samples to x



Sampling signal more densely (increasing sampling rate) enables more accurate reconstruction



Reconstruction from denser sampling



▪▪▪▪▪ = reconstruction via nearest

▪▪▪▪▪ = reconstruction via linear interpolation

Reconstruction as convolution (box filter)

Sampled signal:

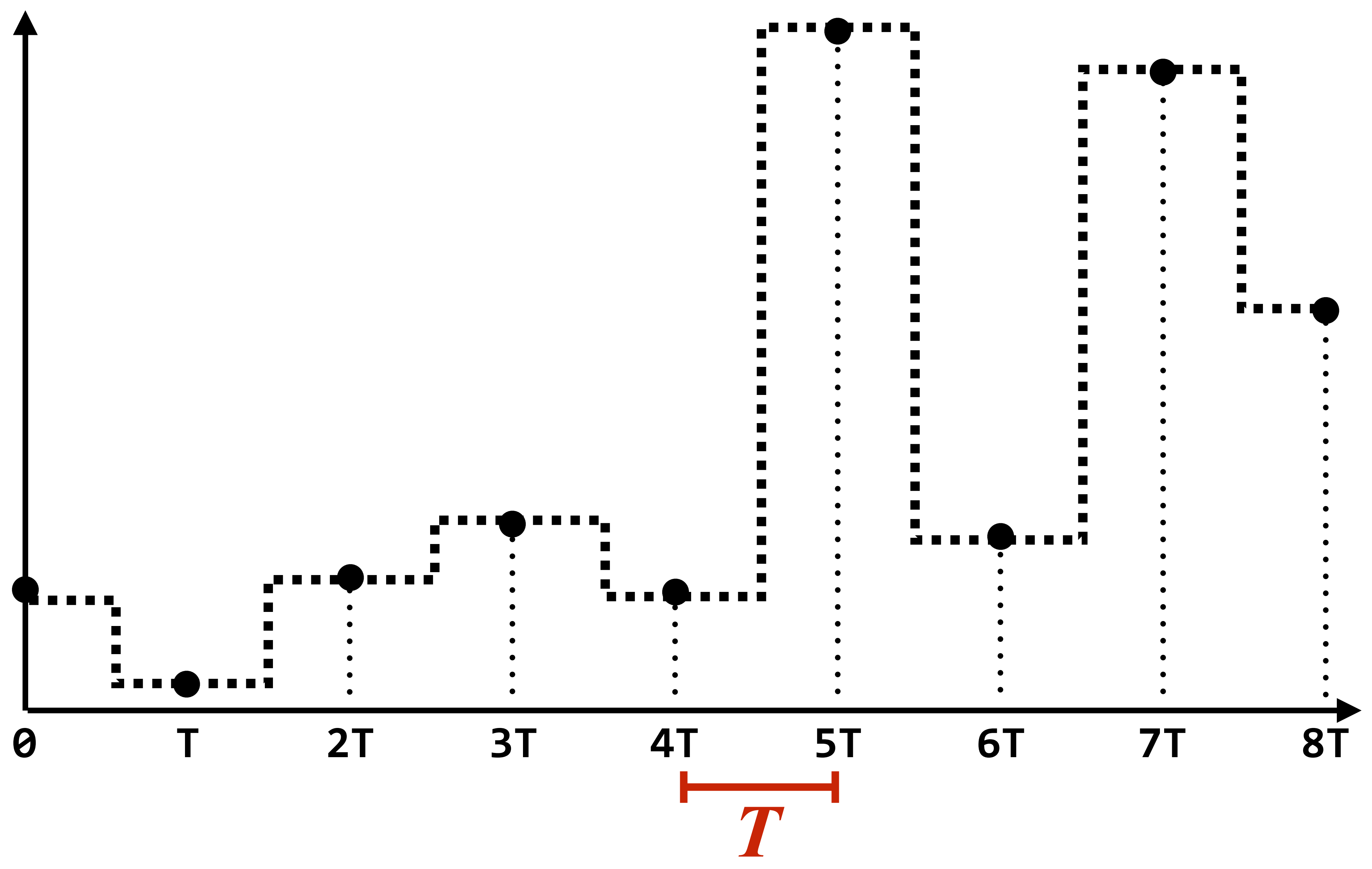
(with period T)

$$g(x) = \text{Ш}_T(x)f(x) = T\sum_{i=-\infty}^{\infty} f(iT)\delta(x - iT)$$

Reconstruction filter:

(unit area box of width T)

$$h(x) = \begin{cases} 1/T & |x| \le T/2 \\ 0 & otherwise \end{cases}$$



Reconstructed signal:

(chooses nearest sample)

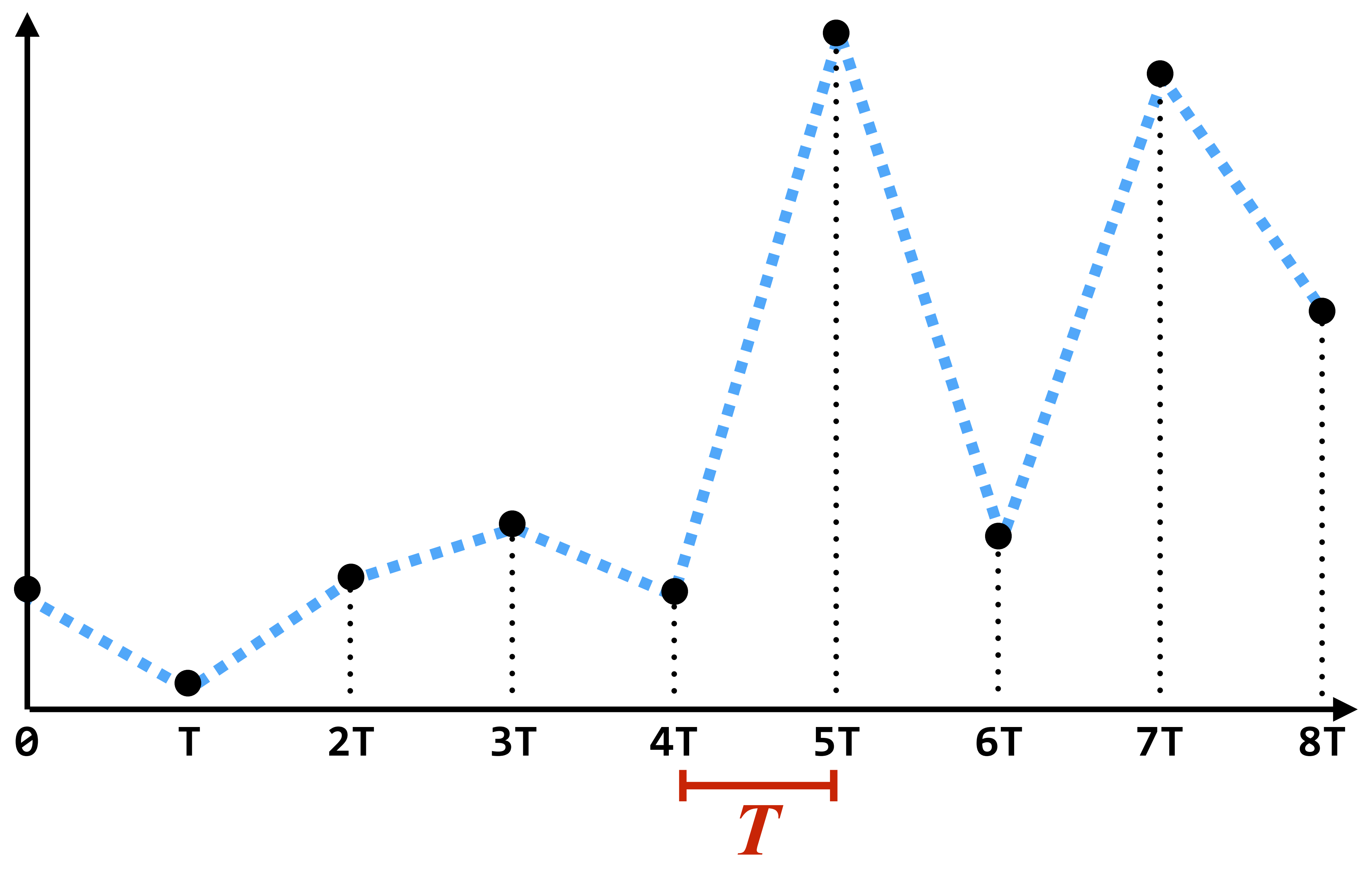
$$f_{recon}(x) = (h * g)(x) = T\int_{-\infty}^{\infty} h(y)\sum_{i=-\infty}^{\infty} f(iT)\delta(x - y - iT)dy = \int_{-T/2}^{T/2}\sum_{i=-\infty}^{\infty} \underbrace{f(iT)\delta(x - y - iT)}_{\text{non-zero only for } iT \text{ closest to } x}$$

Reconstruction as convolution (triangle filter)

Sampled signal:

(with period T)

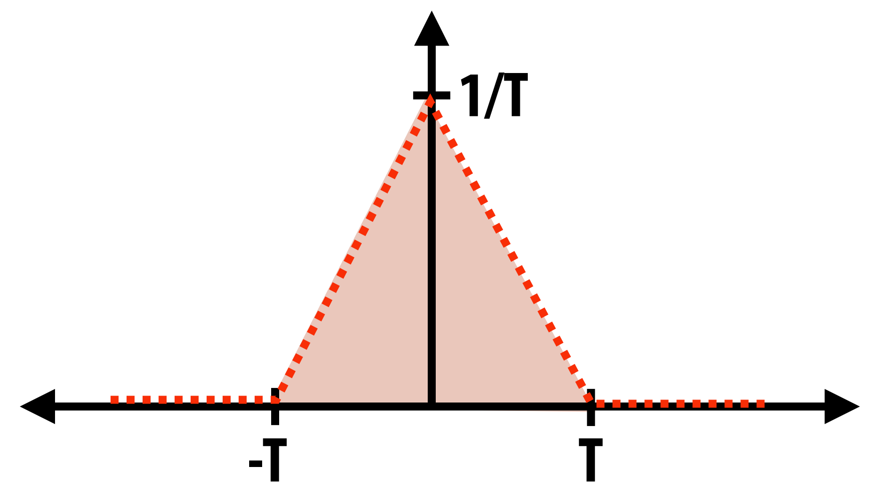
$$g(x) = \text{Ш}_T(x)f(x) = T\sum_{i=-\infty}^{\infty} f(iT)\delta(x - iT)$$



Reconstruction filter:

(unit area triangle of width T)

$$h(x) = \begin{cases} (1 - \frac{|x|}{T})/T & |x| \le T \\ 0 & otherwise \end{cases}$$



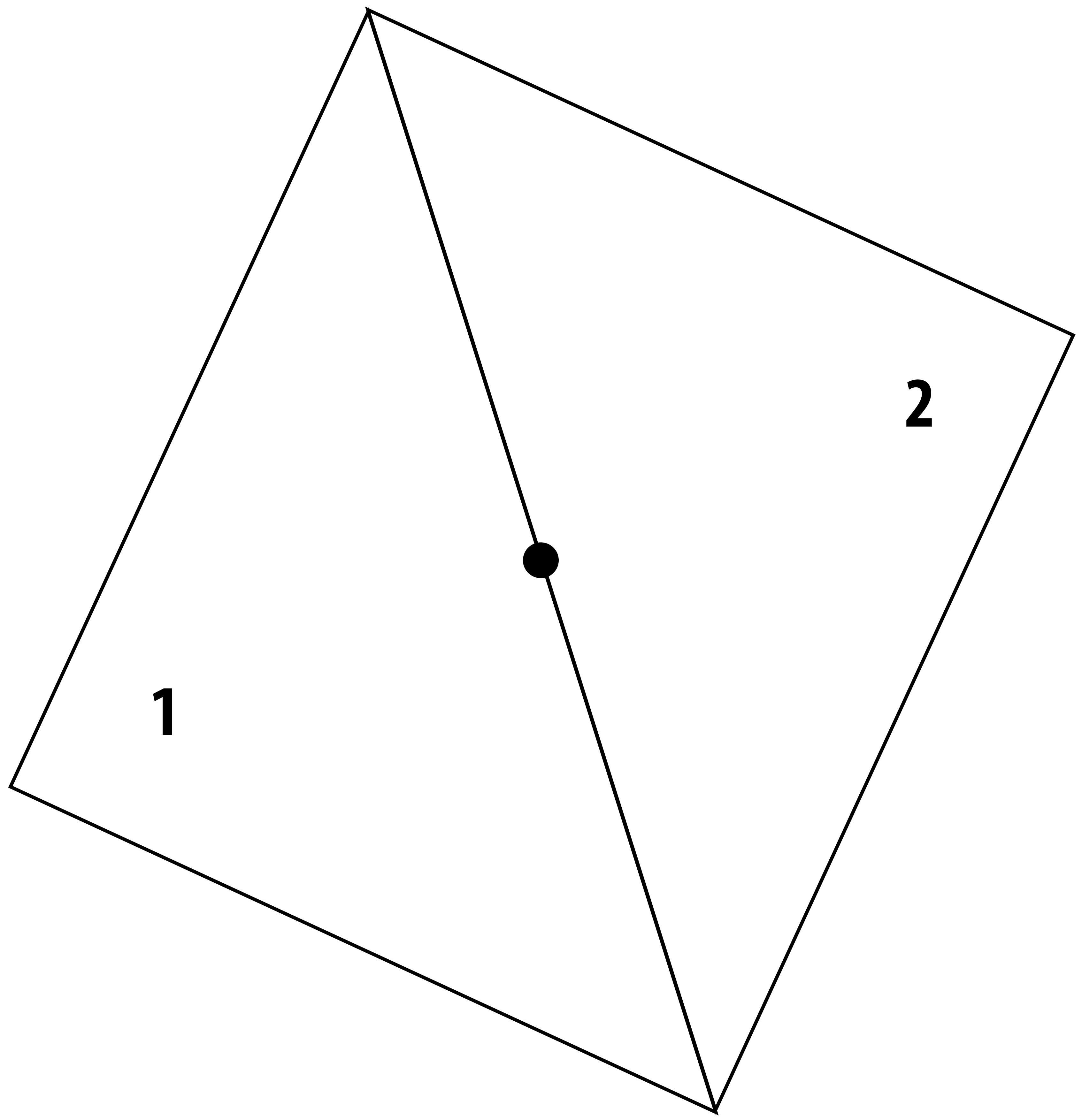
Back to computing coverage

Sampling 2D triangle coverage signal

$$\texttt{coverage(x,y)} = \begin{cases} \texttt{1} & \texttt{if the triangle contains point (x,y)} \\ \texttt{0} & \texttt{otherwise} \end{cases}$$

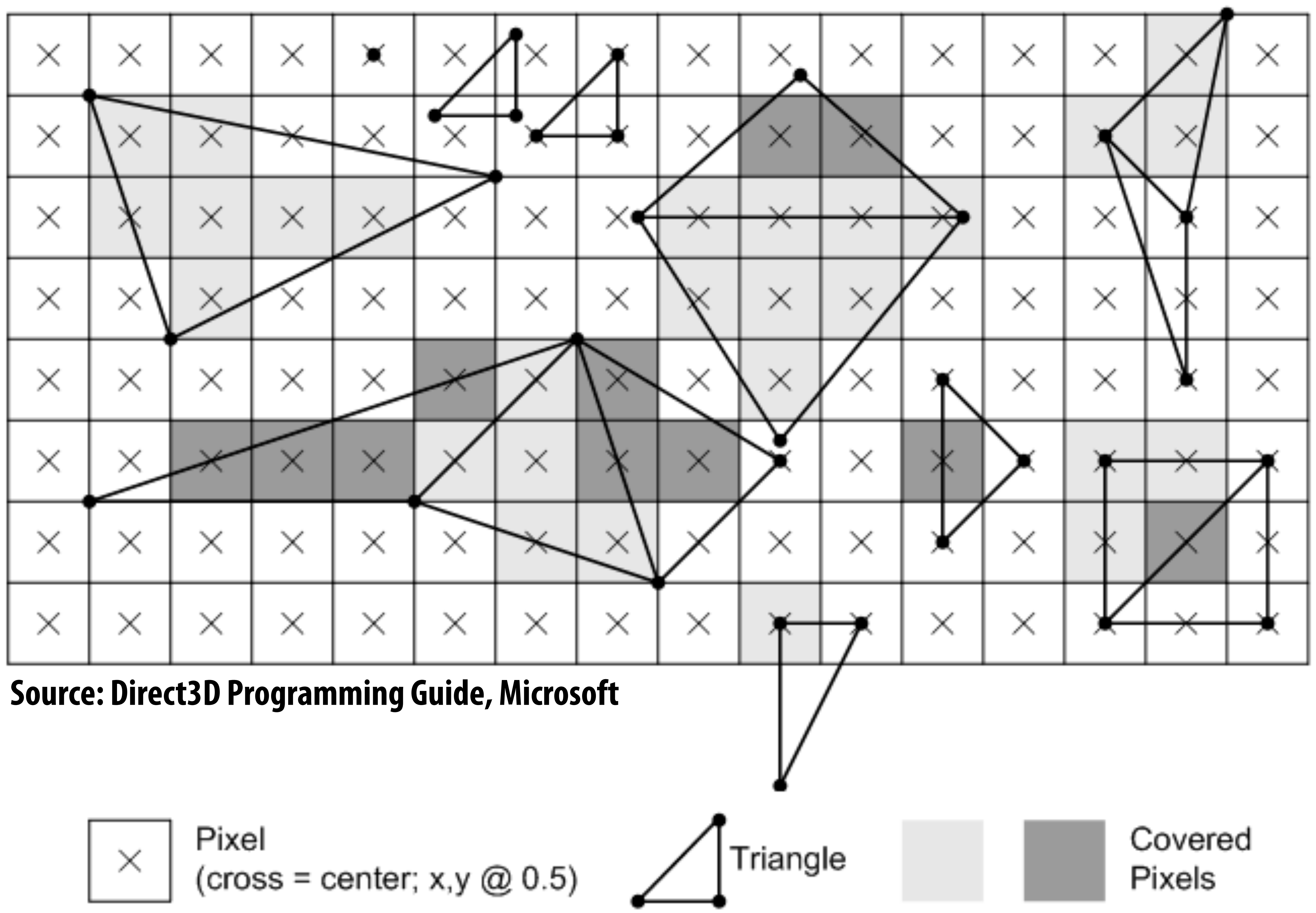
Edge cases (literally)

Is this sample point covered by triangle 1? or triangle 2? or both?



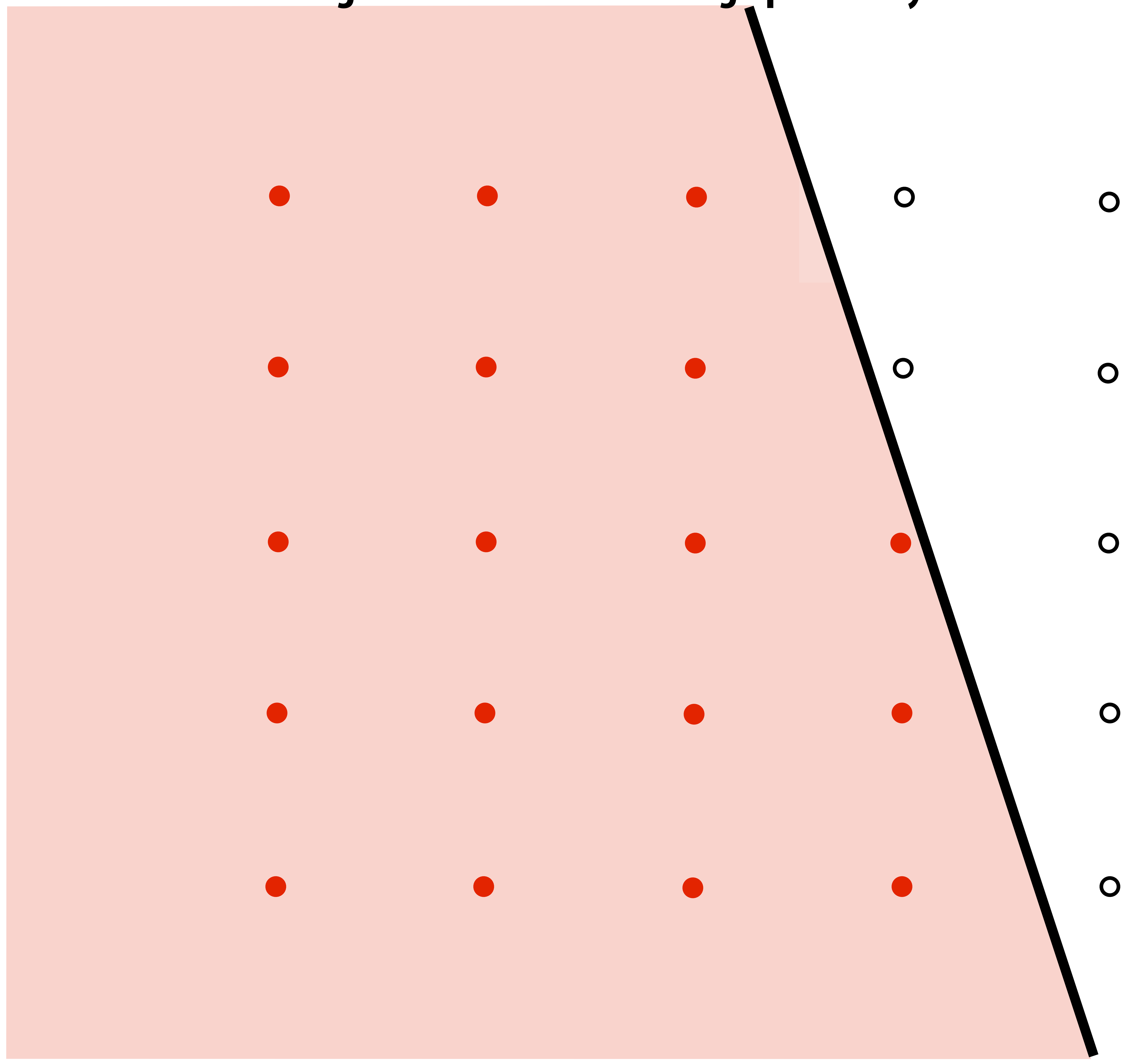
Edge rules

- **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)**

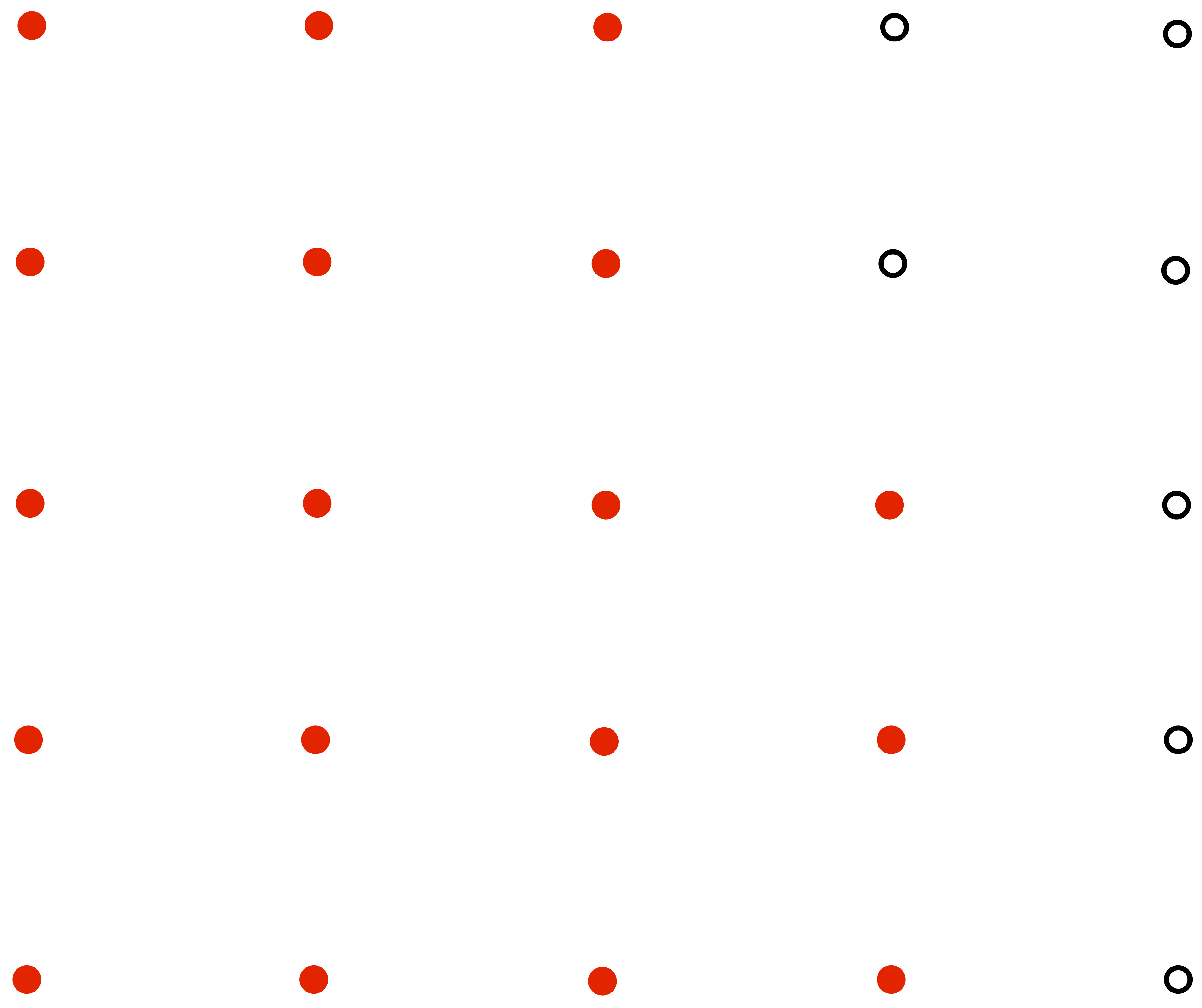

Source: Direct3D Programming Guide, Microsoft
Pixel
(cross = center; x,y @ 0.5)
Triangle
Covered
Pixels

Results of sampling triangle coverage

Note: I'm not drawing the boundaries of image pixels anymore.



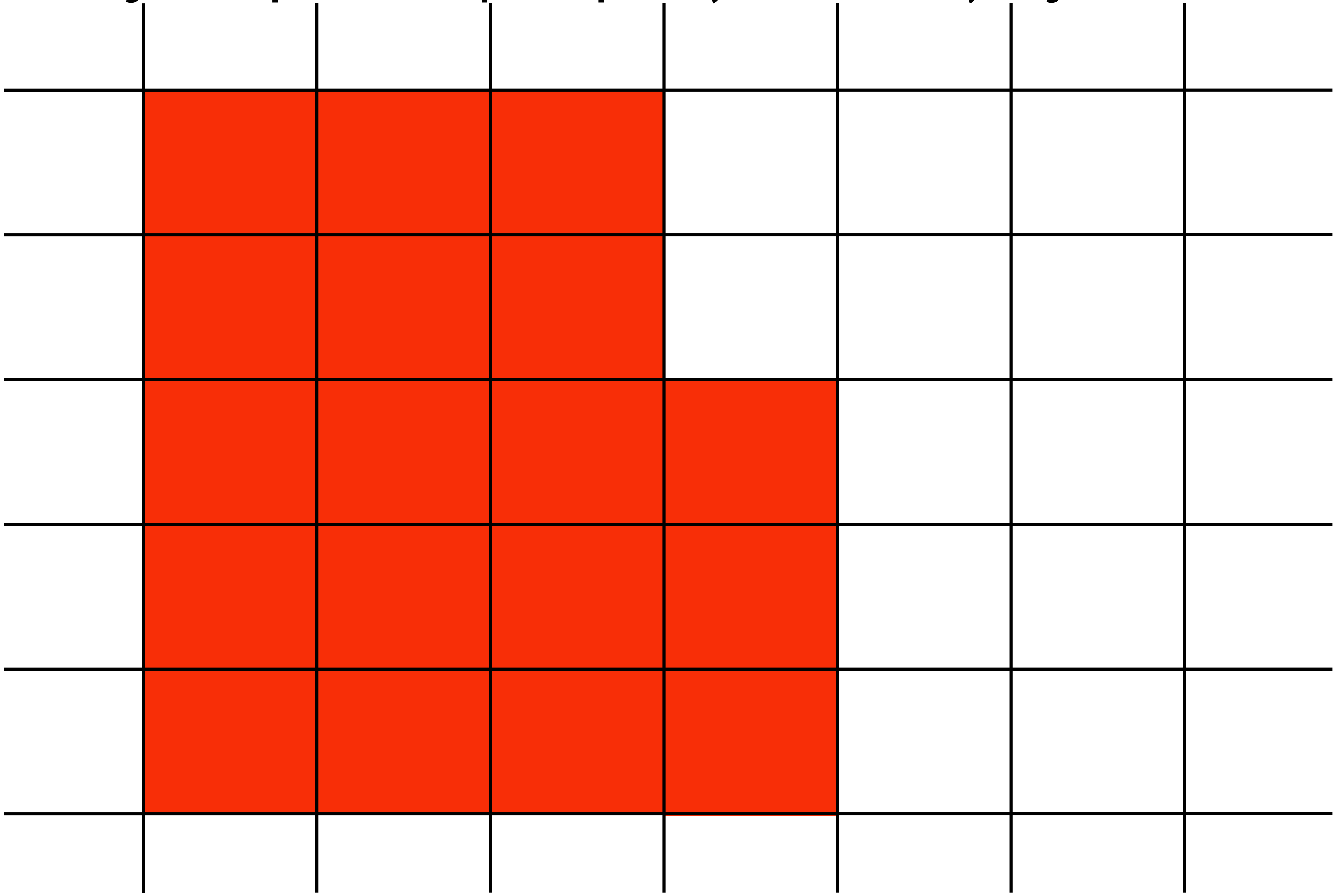
I have a sampled signal, now I want to display it on a screen



So if we send the display this…

We might see this when we look at the screen

(assuming a screen pixel emits a square of perfectly uniform intensity of light)



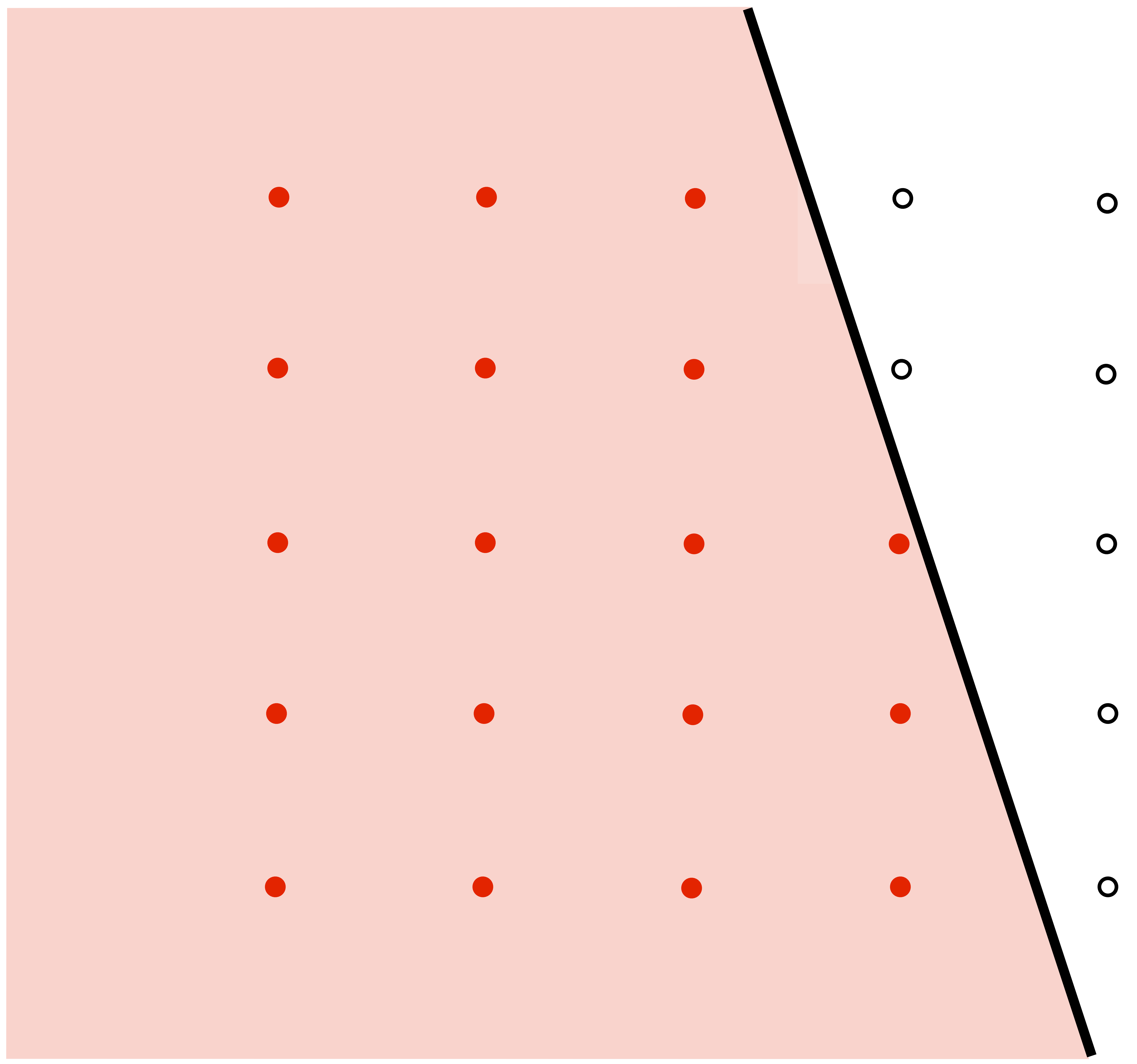
Recall: the real coverage signal was this

Problem: aliasing

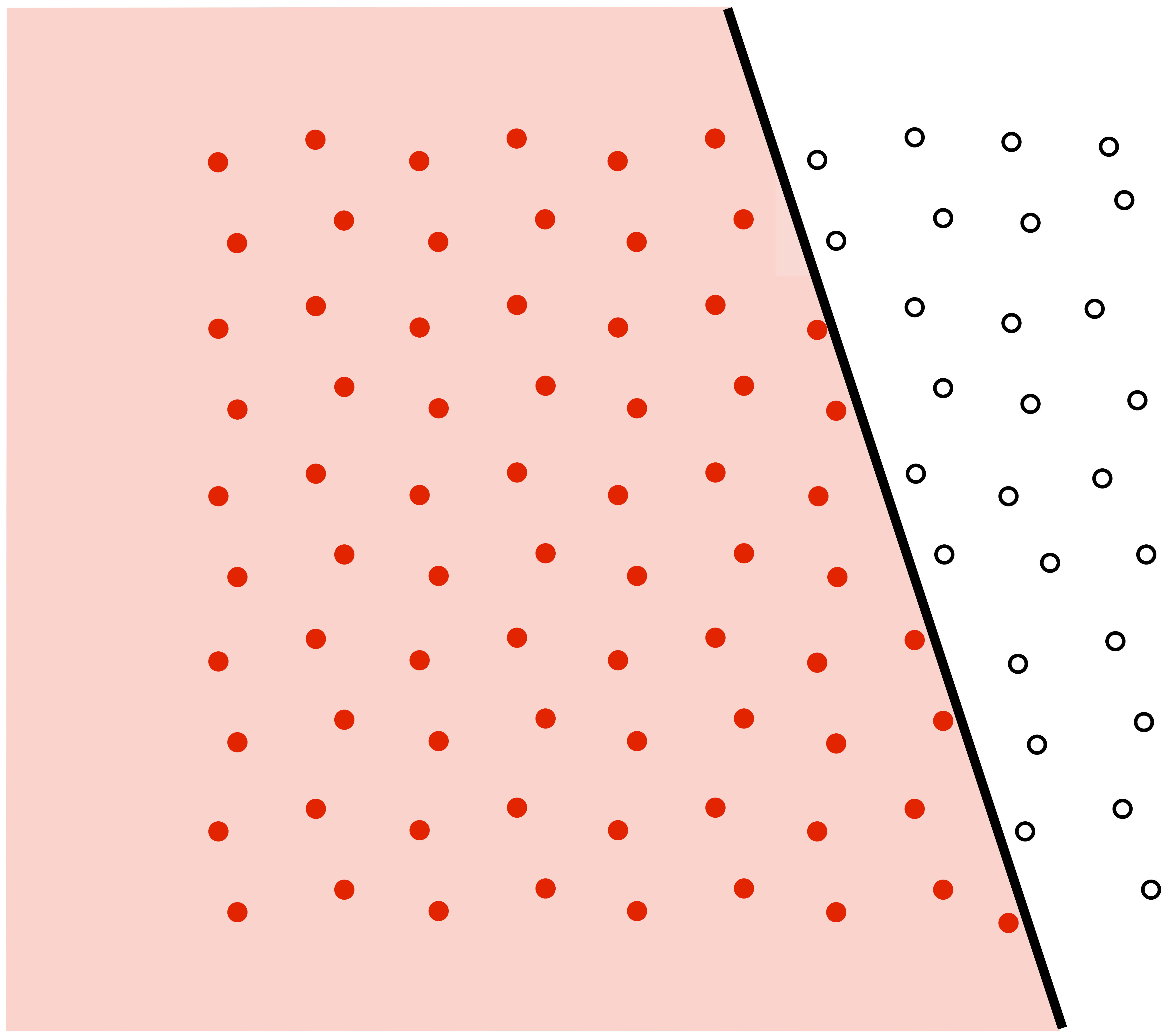
- Undersampling high frequency signal results in aliasing
  - "Jaggies" in a single image
  - "Roping" or "shimmering" in an animation

- High frequencies exist in coverage signal because of triangle edges

Initial coverage sampling rate (1 sample per pixel)

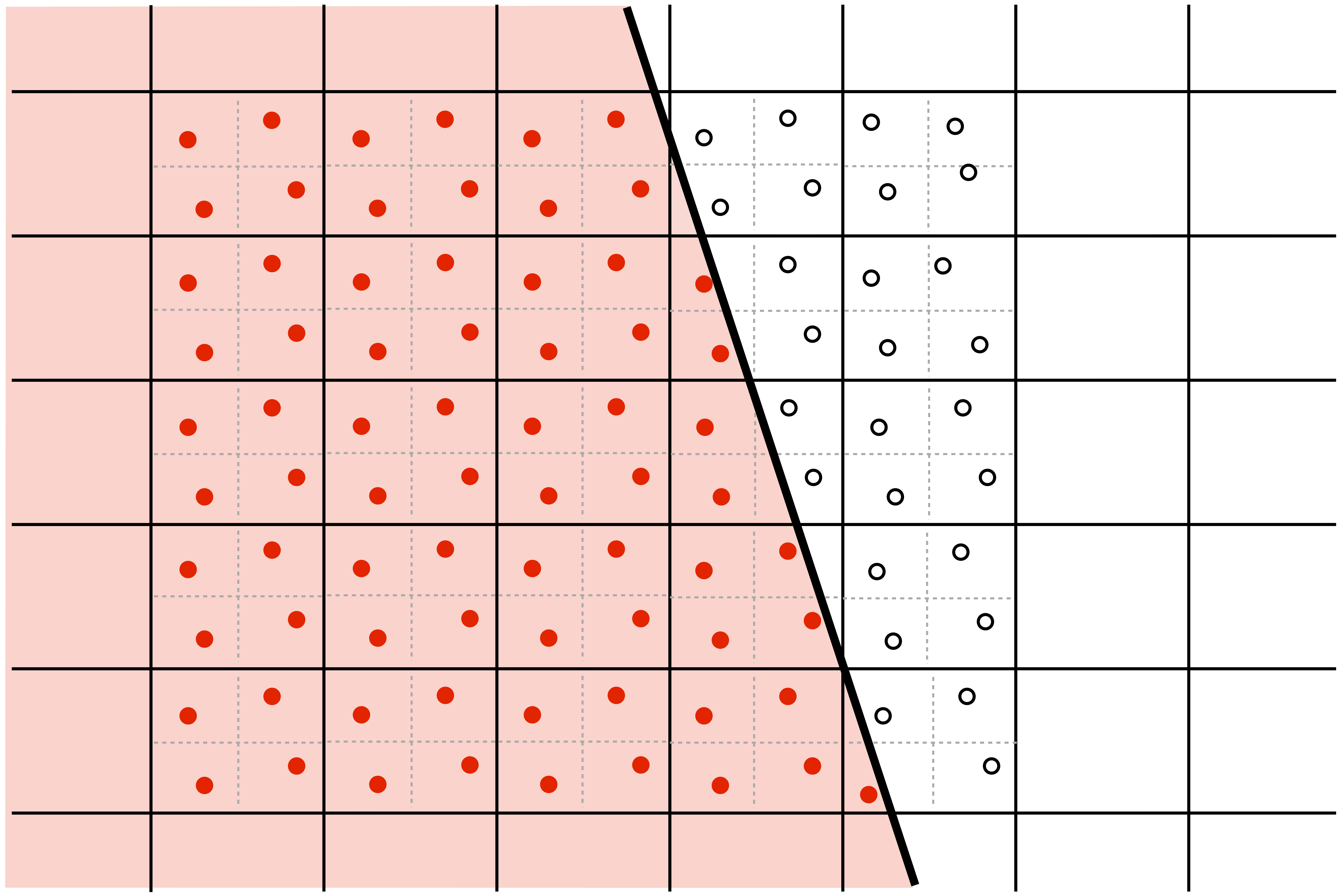


Increase density of sampling coverage signal



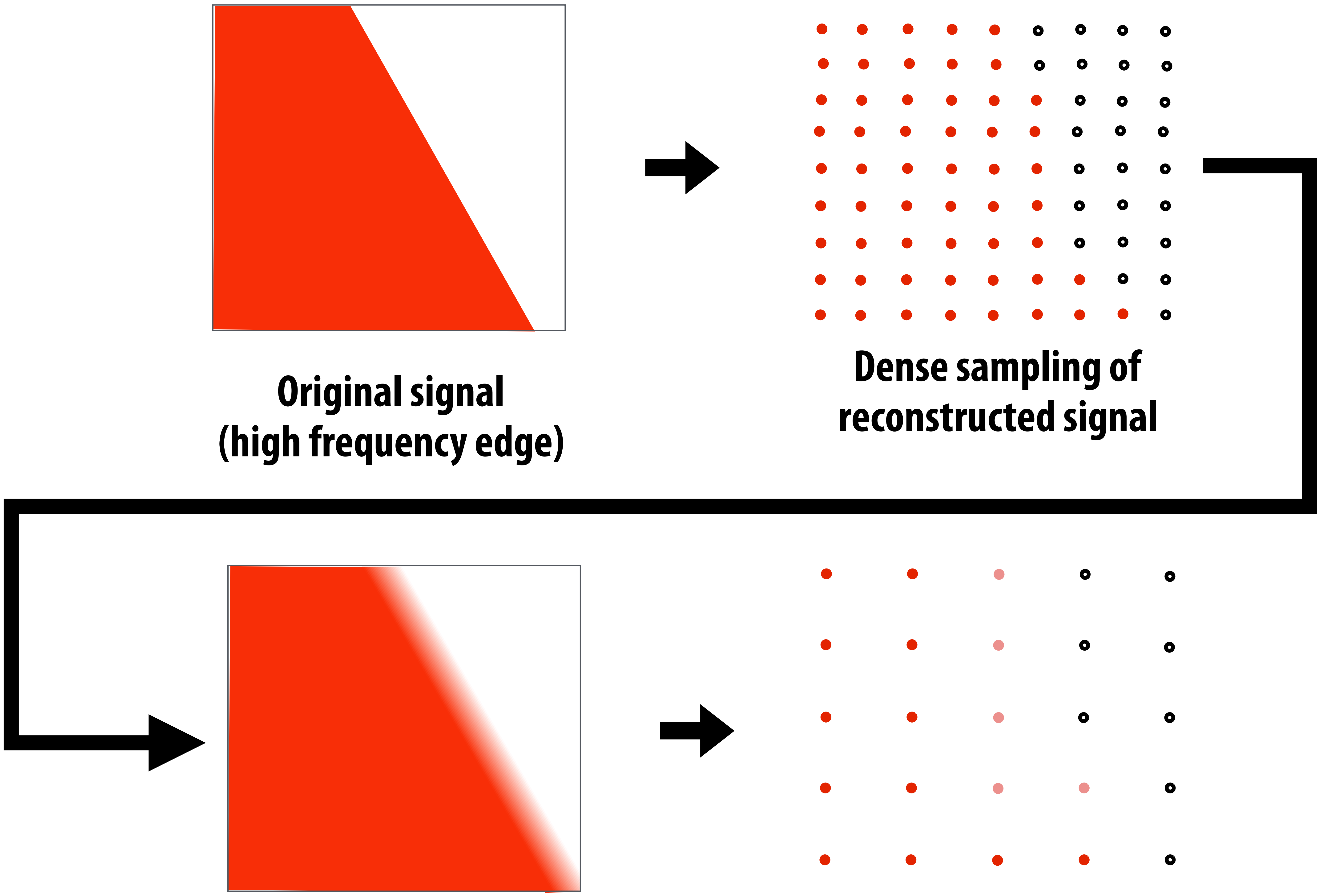
Supersampling

Example: stratified sampling using four samples per pixel



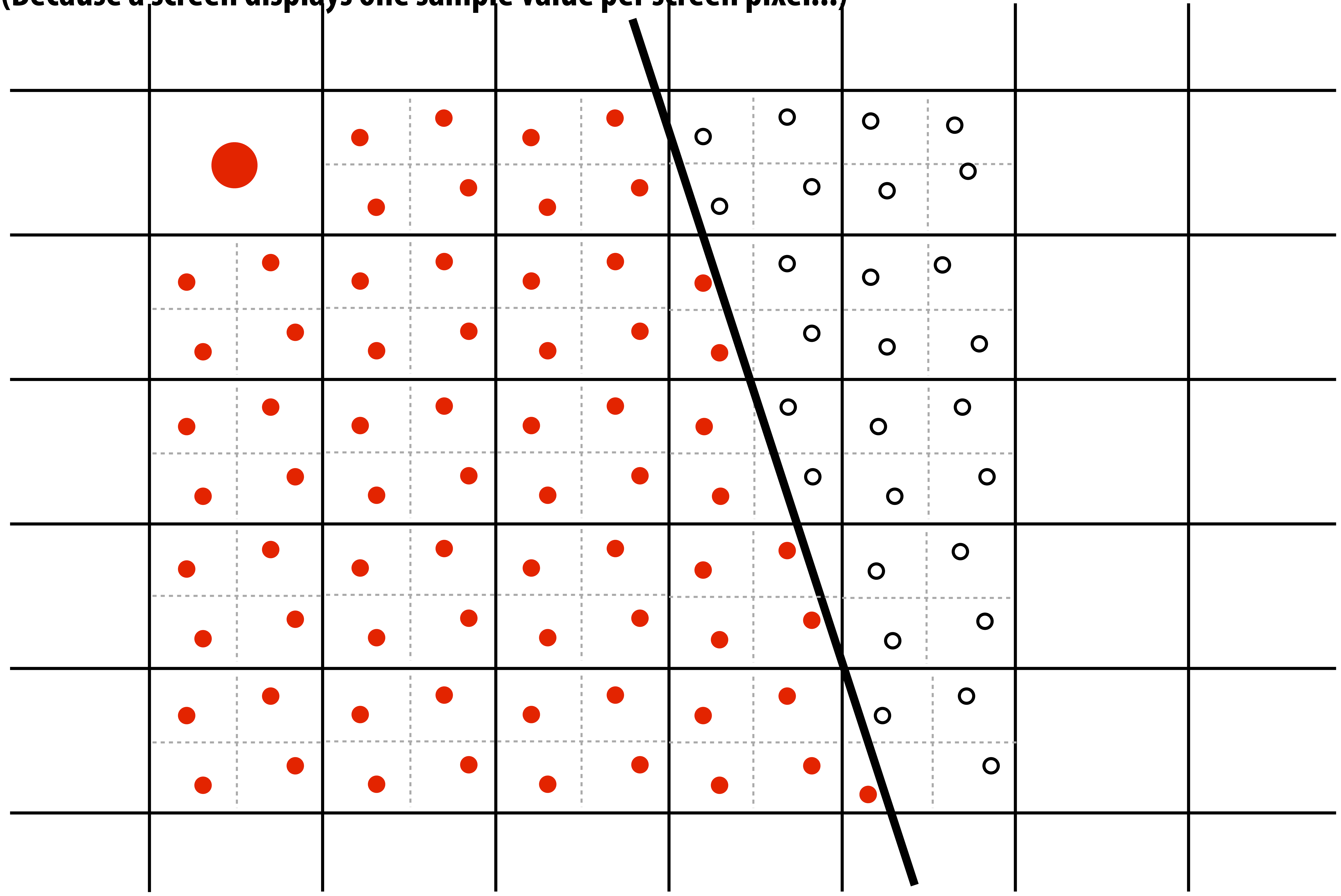
Resampling

Converting from one discrete sampled representation to another

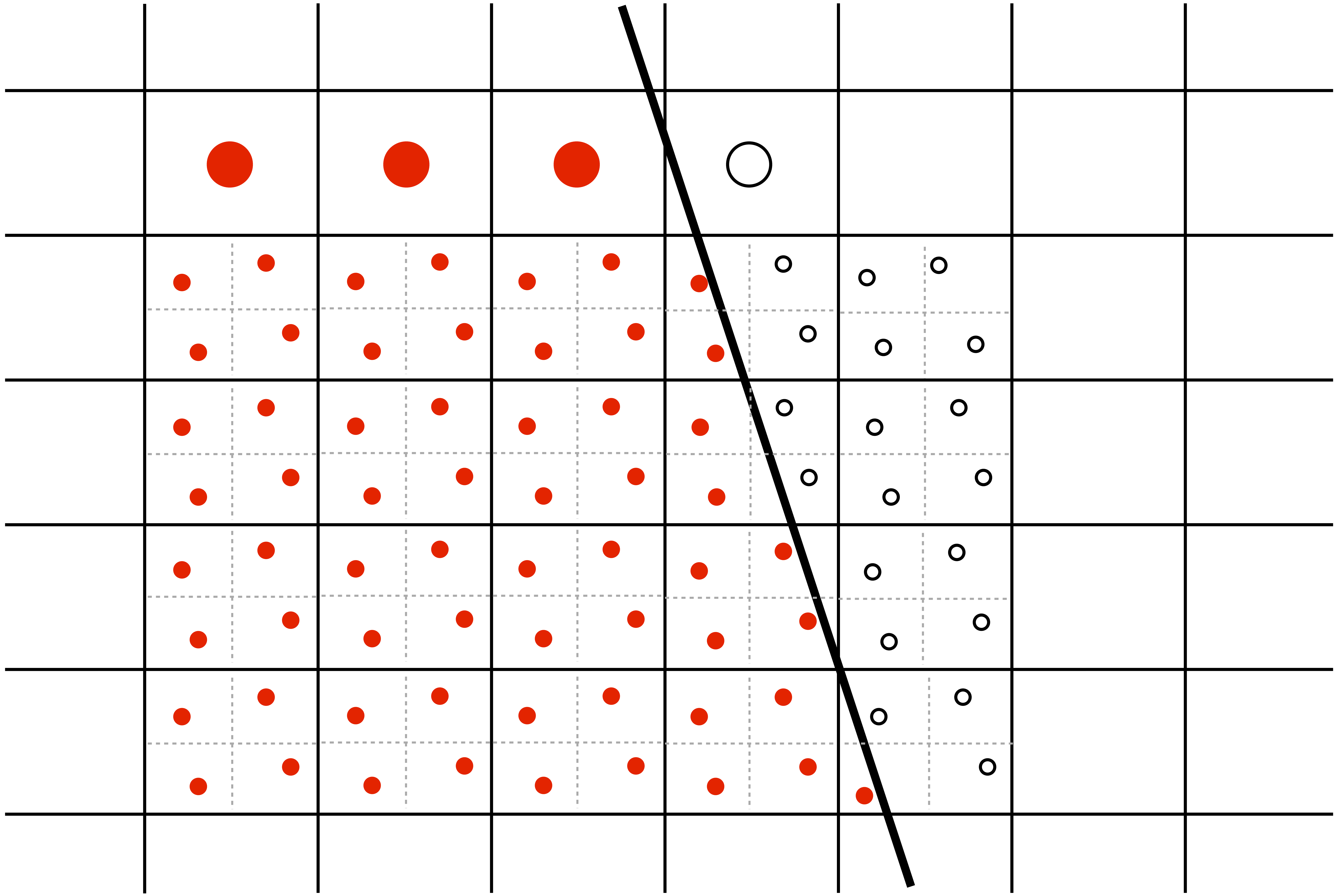


Resample to display's pixel resolution

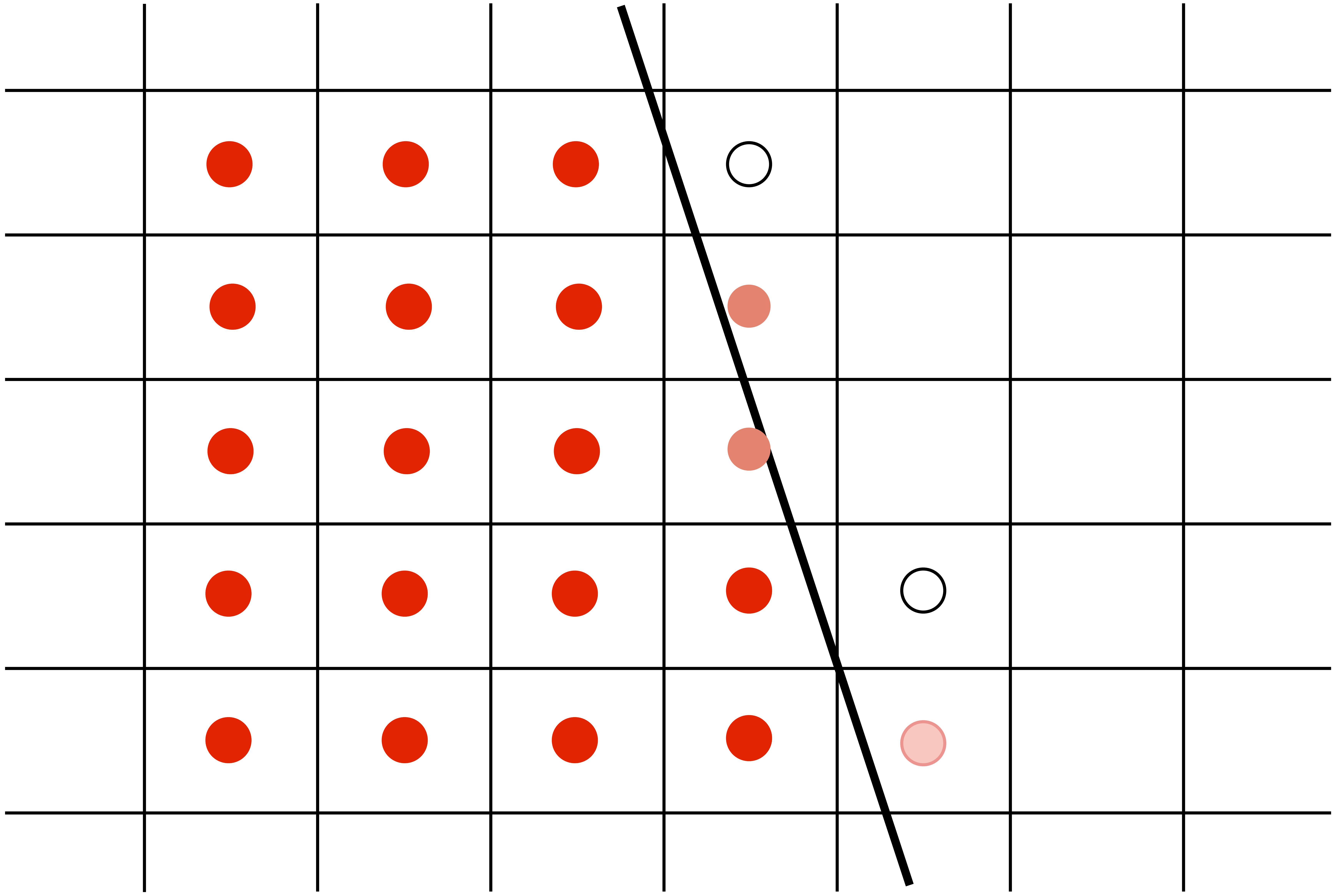
(Because a screen displays one sample value per screen pixel...)



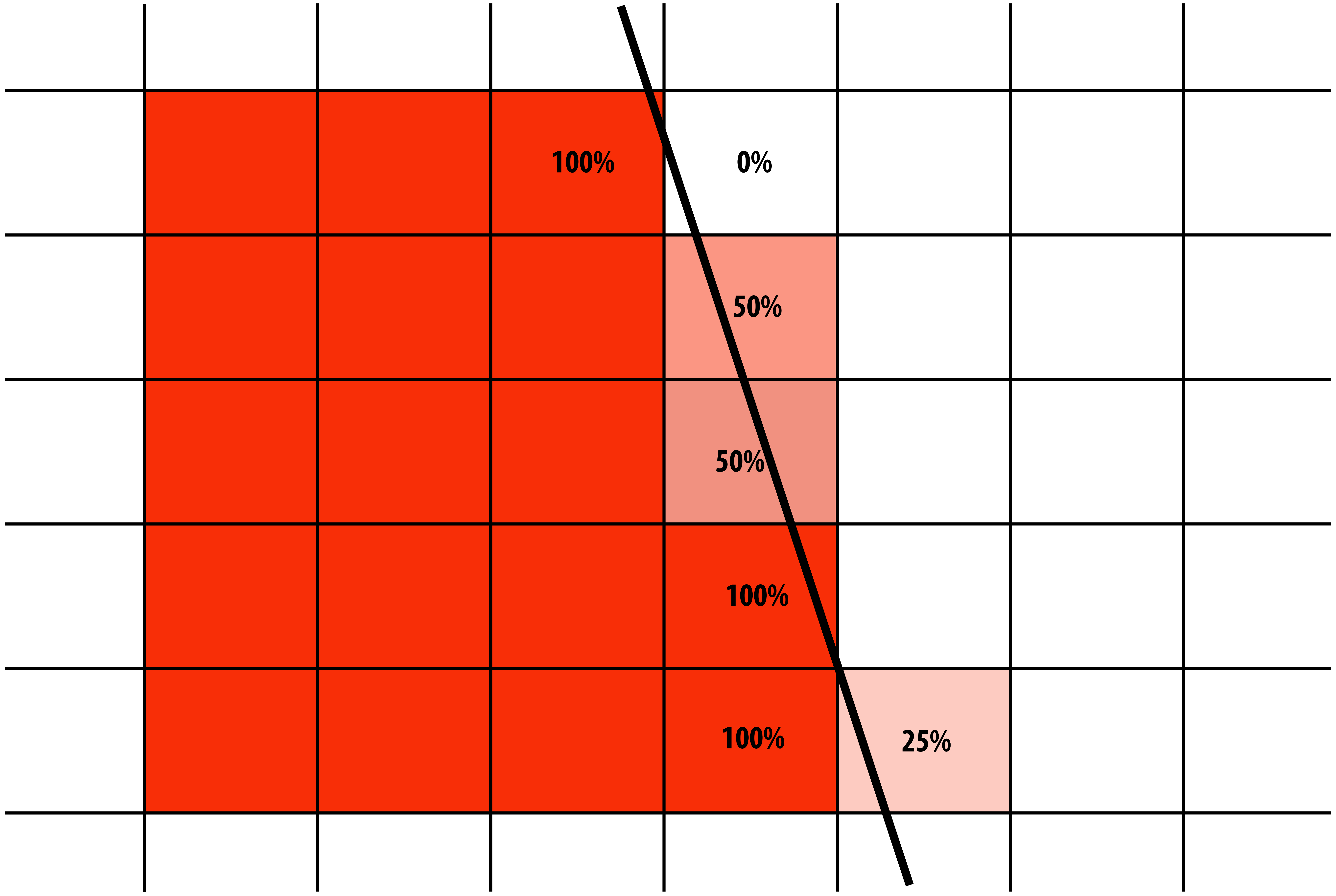
Resample to display's pixel rate (box filter)



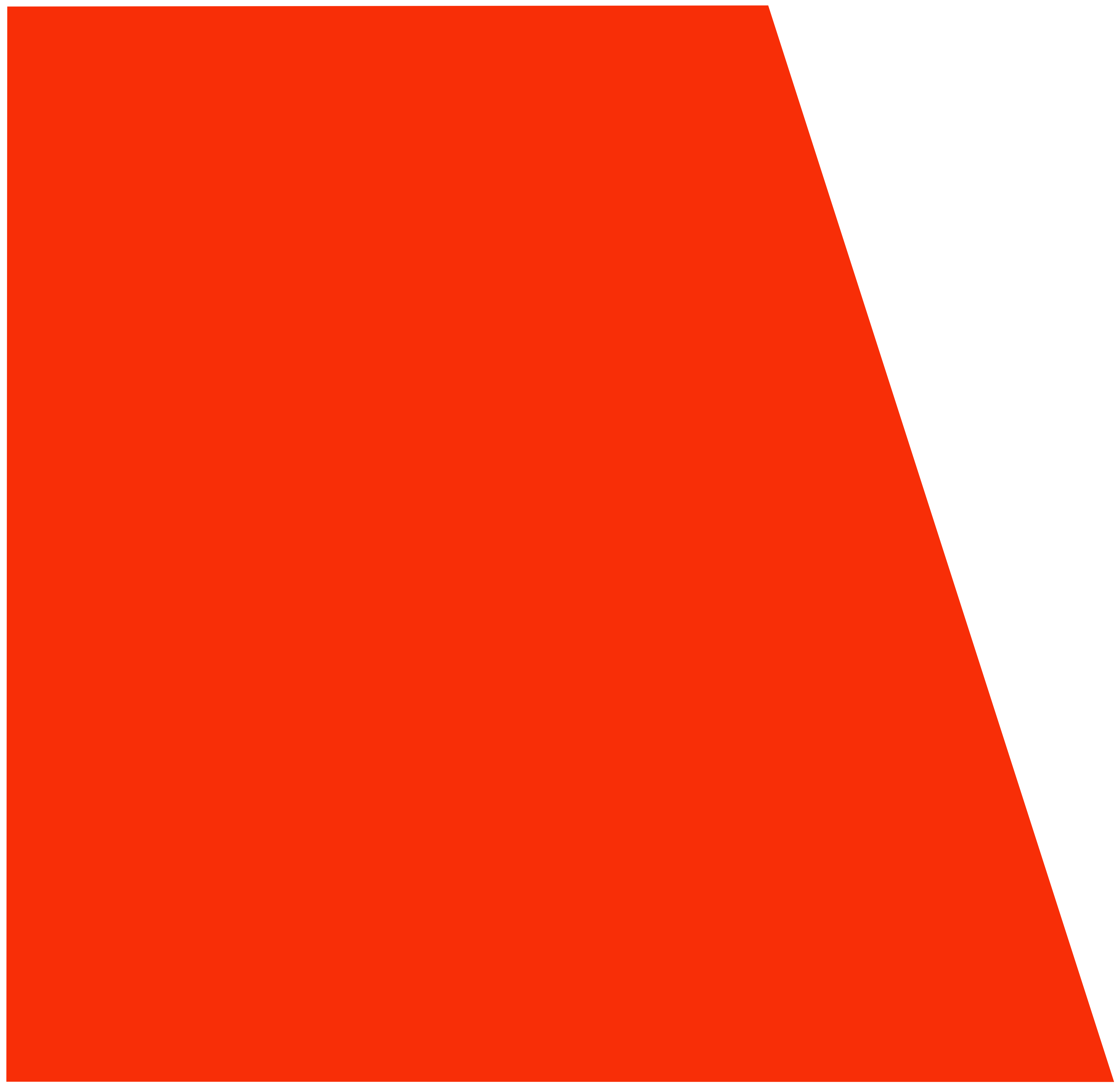
Resample to display's pixel rate (box filter)



Displayed result (note anti-aliased edges)



Recall: the real coverage signal was this



Sampling coverage

- We want the light emitted from a display to be an accurate to match the ground truth signal: `coverage(x,y)`)

- Resampling a densely sampled signal (supersampled) integrates coverage values over the entire pixel region. The integrated result is sent to the display (and emitted by the pixel) so that the light emitted by the pixel is similar to what would be emitted in that screen region by an "infinite resolution display"

Modes of fragment generation

- **Supersampling: generate one fragment for <u>each</u> covered sample**

- **Multi-sampling: general one fragment per pixel if <u>any</u> 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

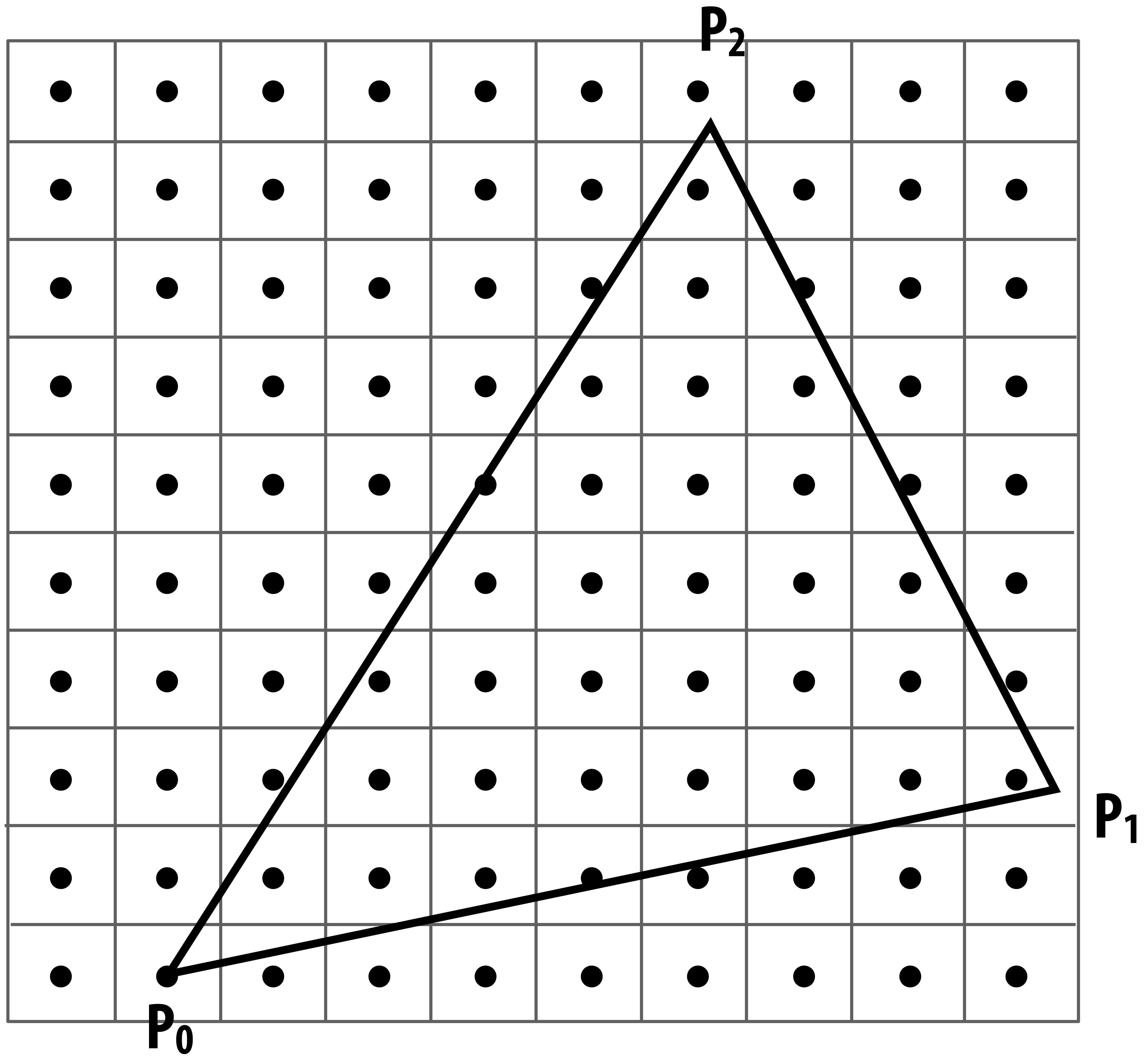
Compute triangle edge equations from projected positions of vertices

$P_i = (X_i, Y_i)$

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

$$\begin{aligned} E_i(x, y) &= (x - X_i)\, dY_i - (y - Y_i)\, dX_i \\ &= A_i x + B_i y + C_i \end{aligned}$$

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



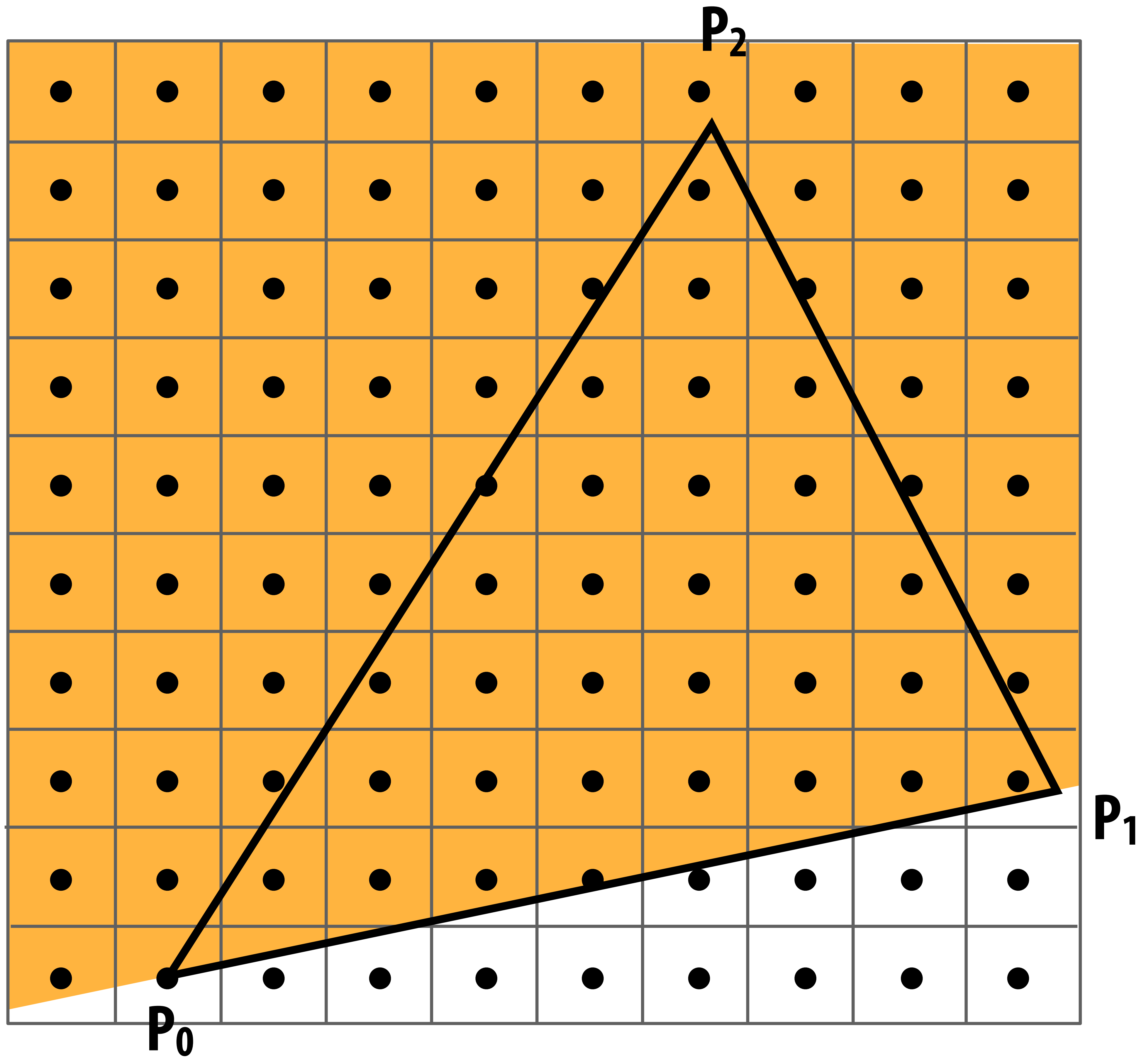
Point-in-triangle test

$P_i = (X_i, Y_i)$

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

$$\begin{aligned} E_i(x, y) &= (x - X_i)\, dY_i - (y - Y_i)\, dX_i \\ &= A_i\, x + B_i\, y + C_i \end{aligned}$$

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



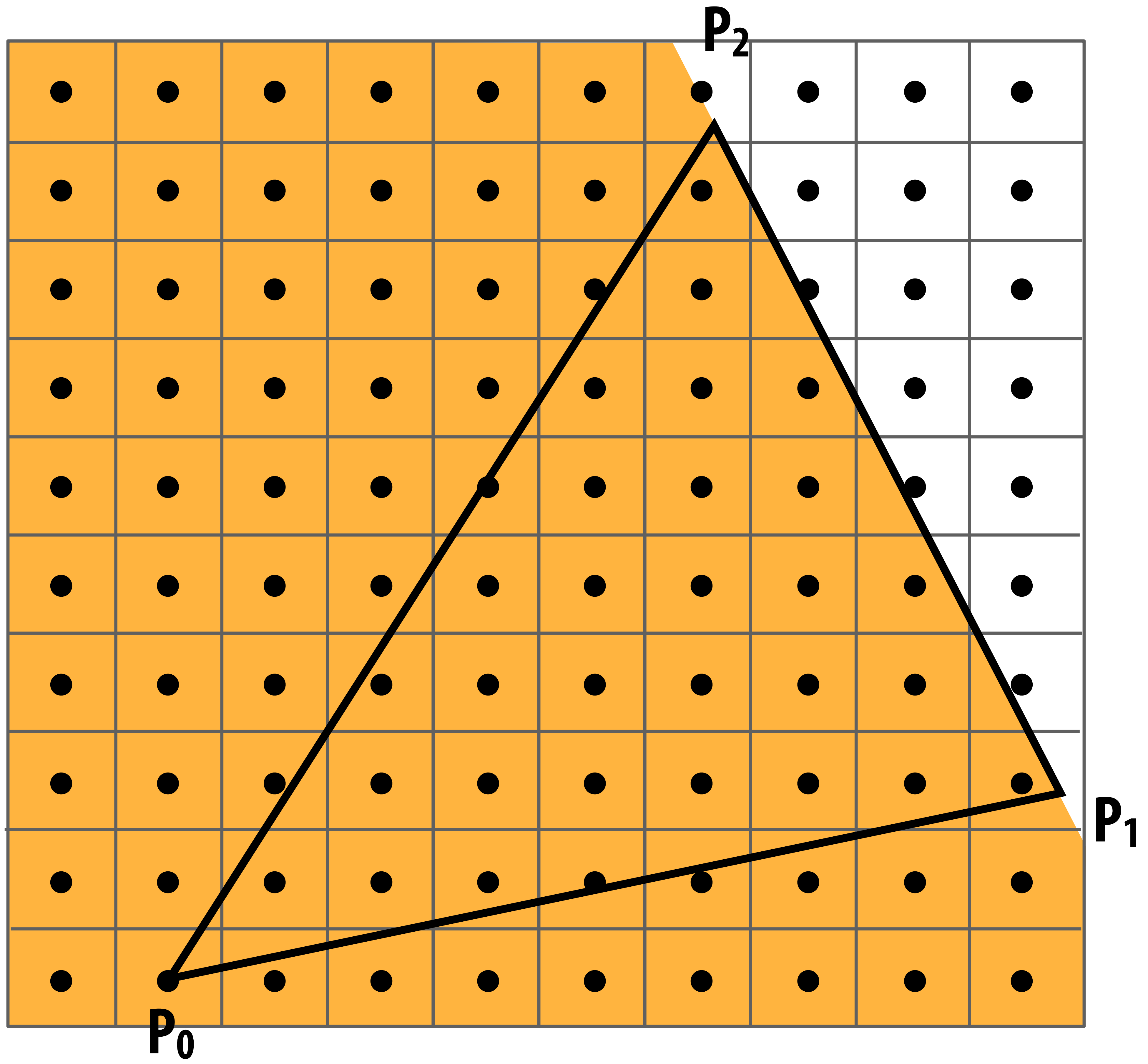
Point-in-triangle test

$P_i = (X_i, Y_i)$

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

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

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



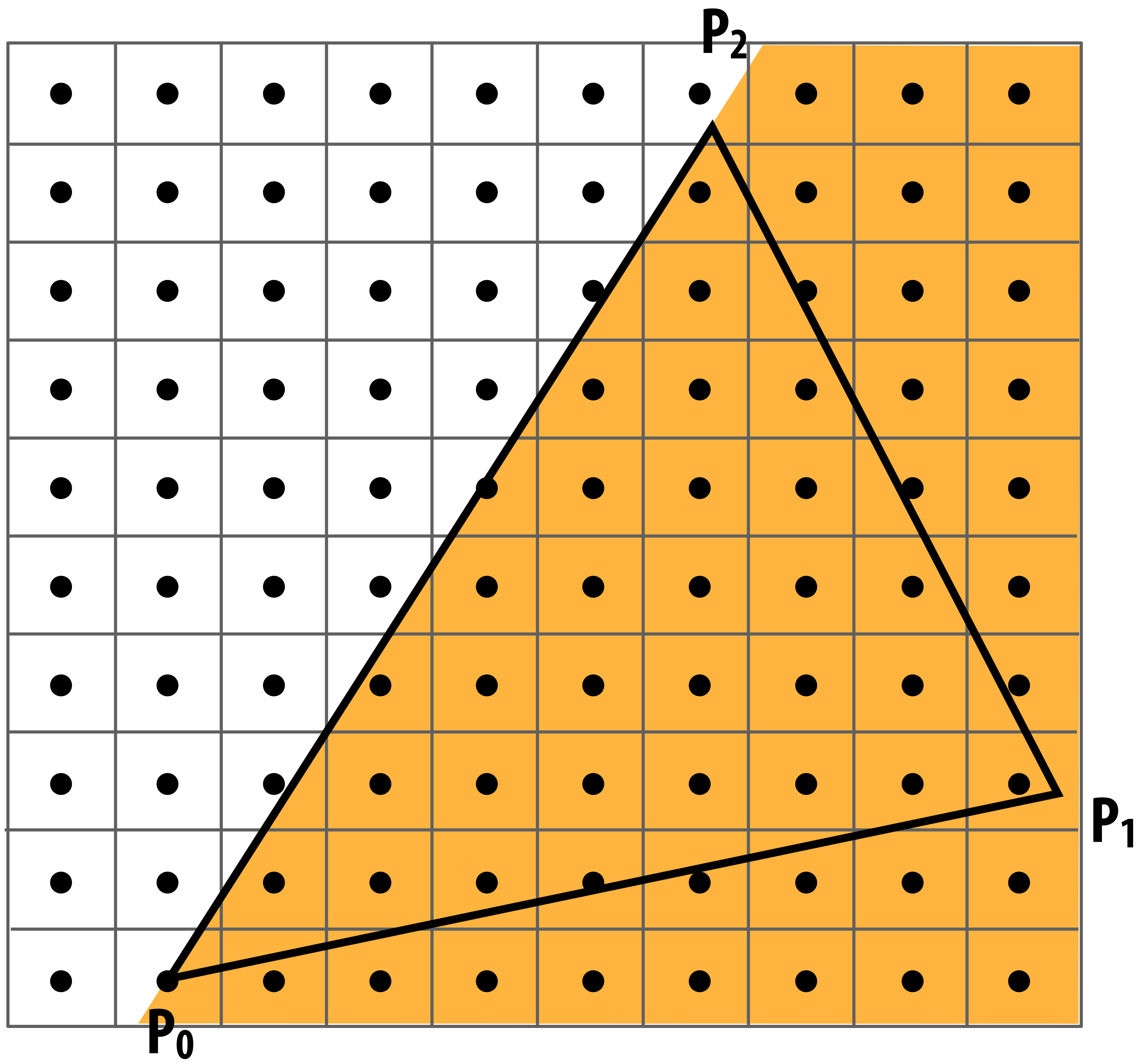
Point-in-triangle test

$P_i = (X_i, Y_i)$

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

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

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

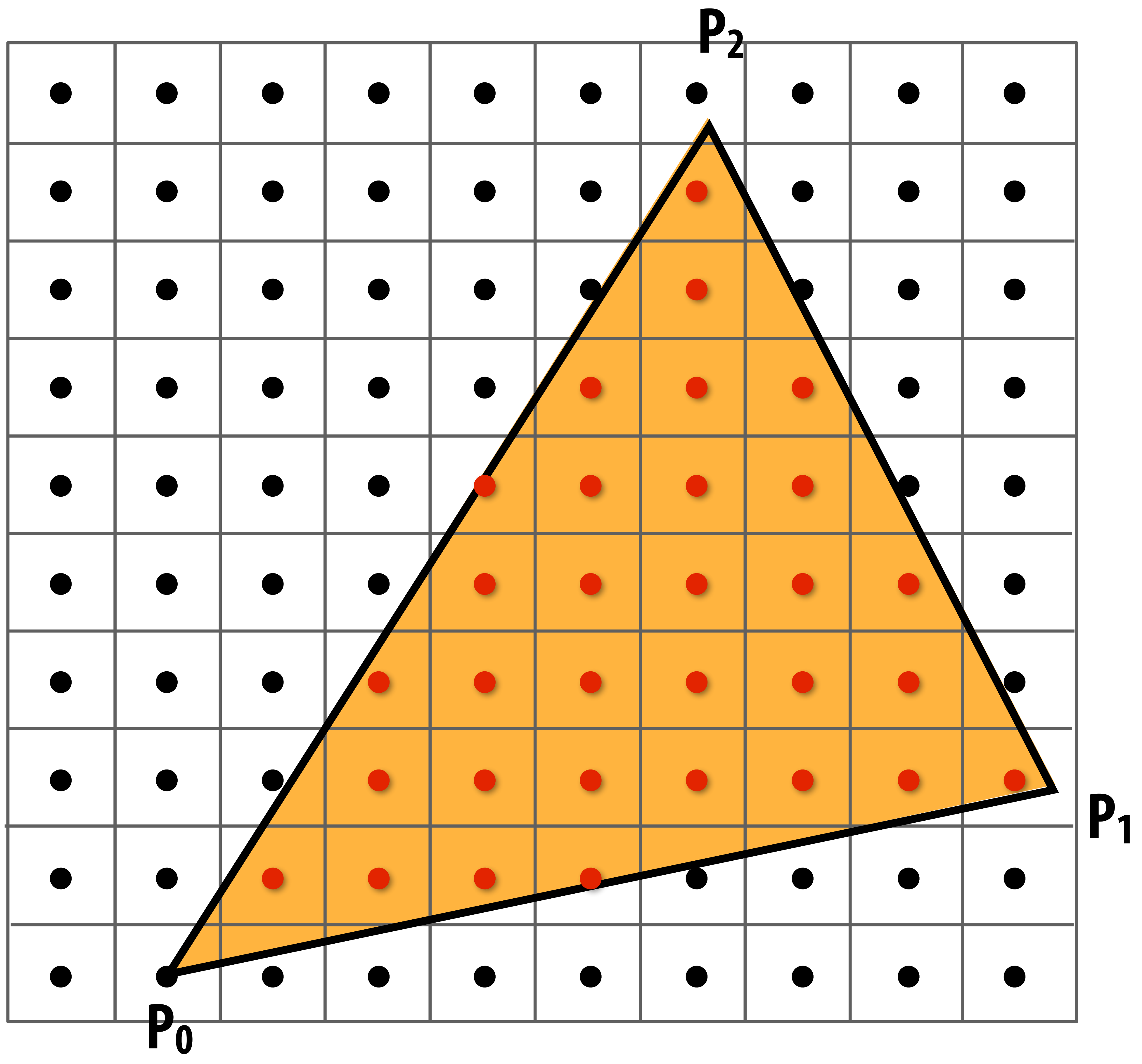


Point-in-triangle test

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

$$
\begin{aligned}
& inside(sx, sy) = \\
& \quad E_0(sx, sy) < 0 \ \&\& \\
& \quad E_1(sx, sy) < 0 \ \&\& \\
& \quad E_2(sx, sy) < 0;
\end{aligned}
$$

Note: actual implementation of $inside(sx,sy)$ **involves ≤ checks based on the triangle coverage edge rules (see beginning of lecture)**



Sample points inside triangle are highlighted red.

Incremental triangle traversal

$P_i = (X_i, Y_i, Z_i)$

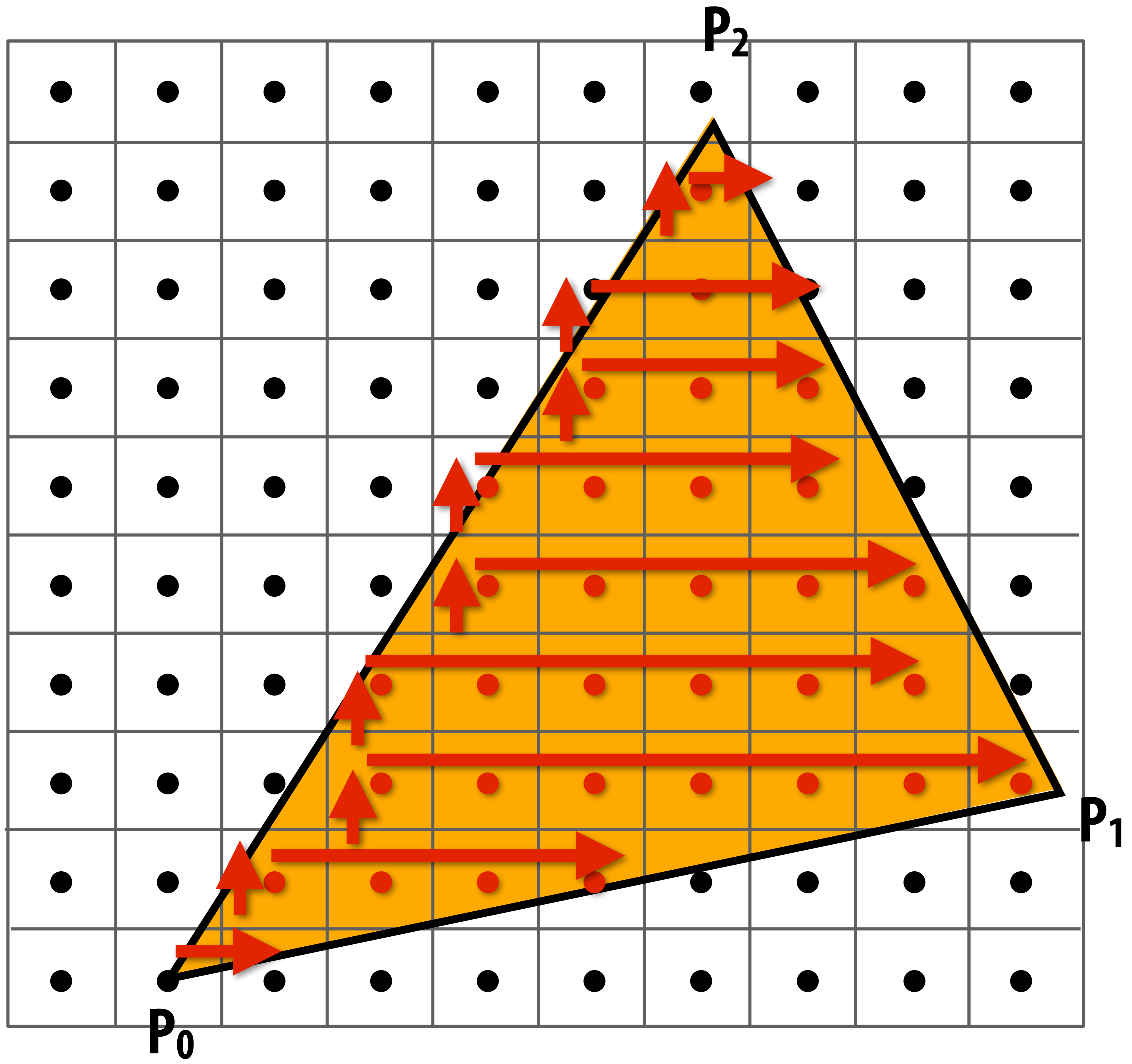
$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

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

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

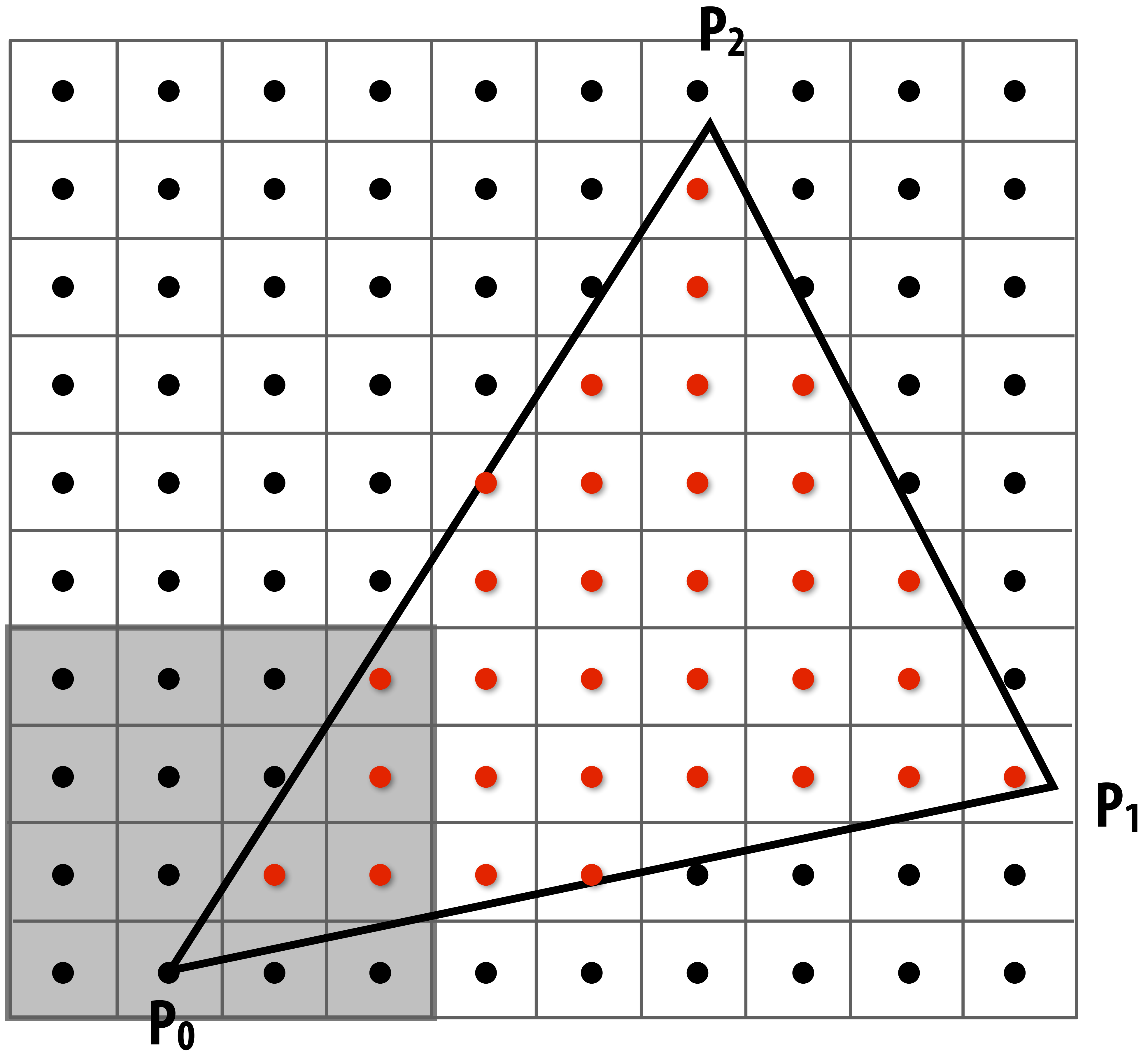
Modern approach: tiled triangle traversal

Traverse triangle in blocks

Test all samples in block against triangle in parallel

Advantages:

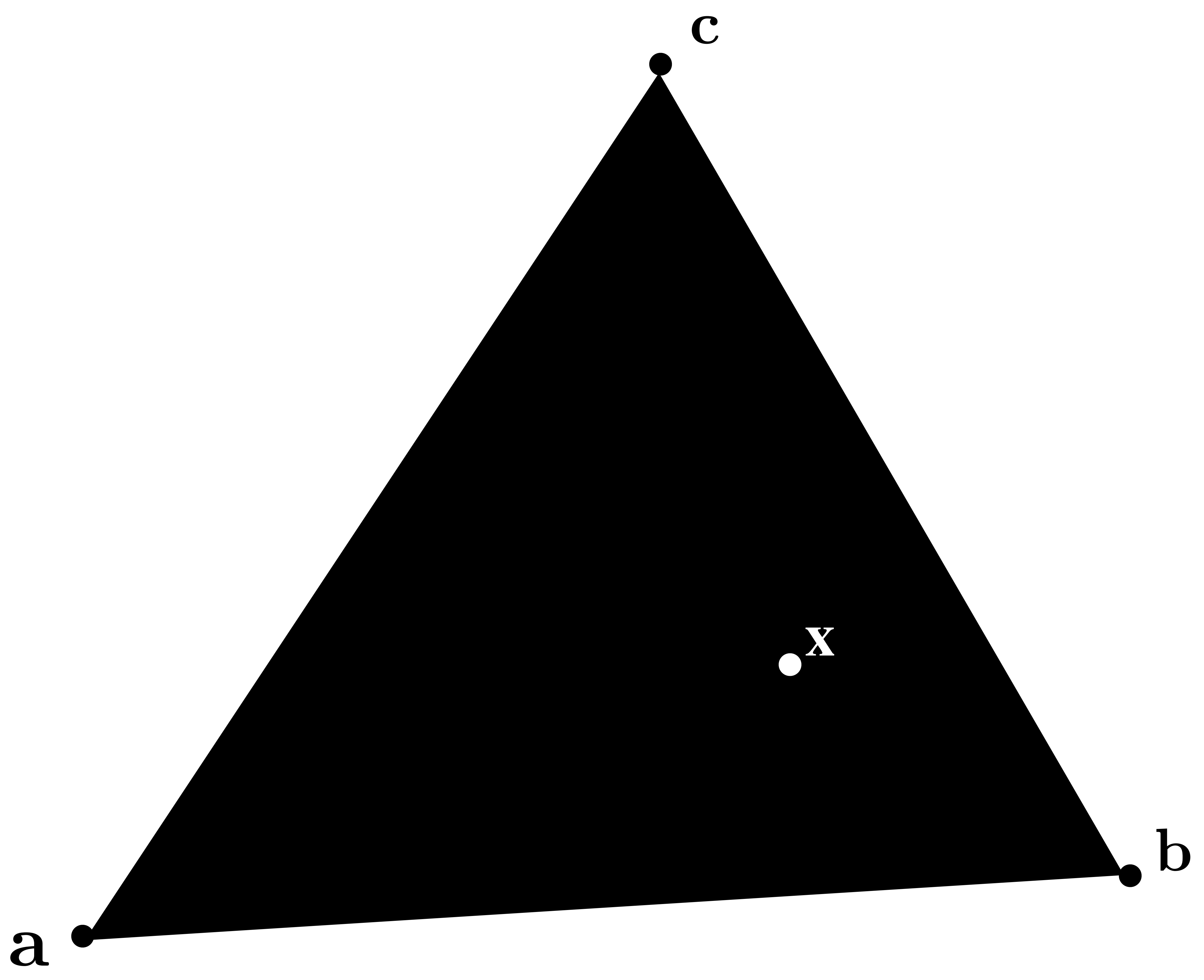
- Simplicity of wide parallel execution overcomes cost of extra point-in-triangle tests (most triangles cover many samples, especially when super-sampling coverage)
- Can skip sample testing work: entire block not in triangle ("early out"), entire block entirely within triangle ("early in")
- Additional advantaged related to accelerating occlusion computations (not discussed today)



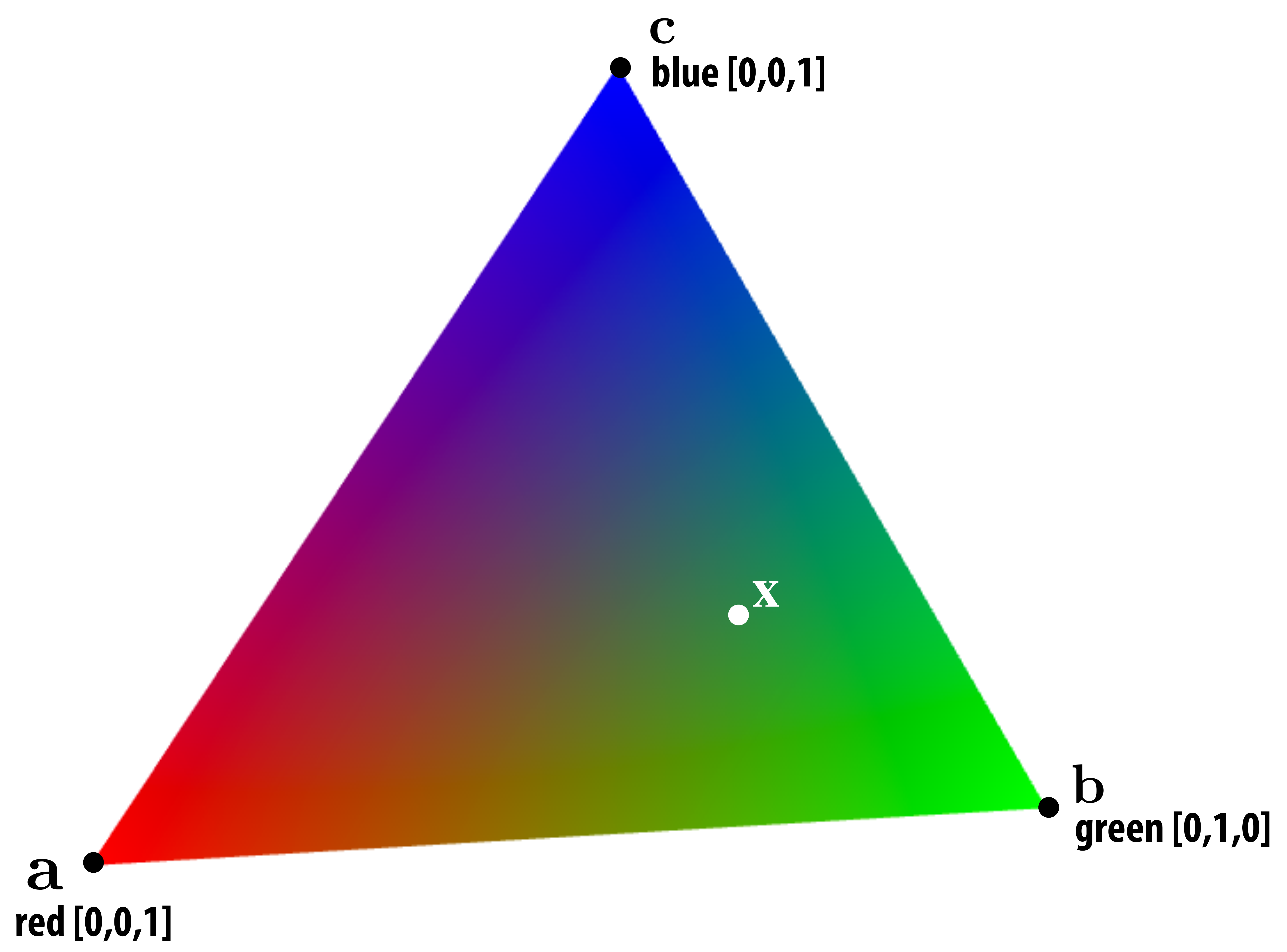
All modern GPUs have special-purpose hardware for efficiently performing point-in-triangle tests

Sampling triangle attributes

Coverage(x,y)



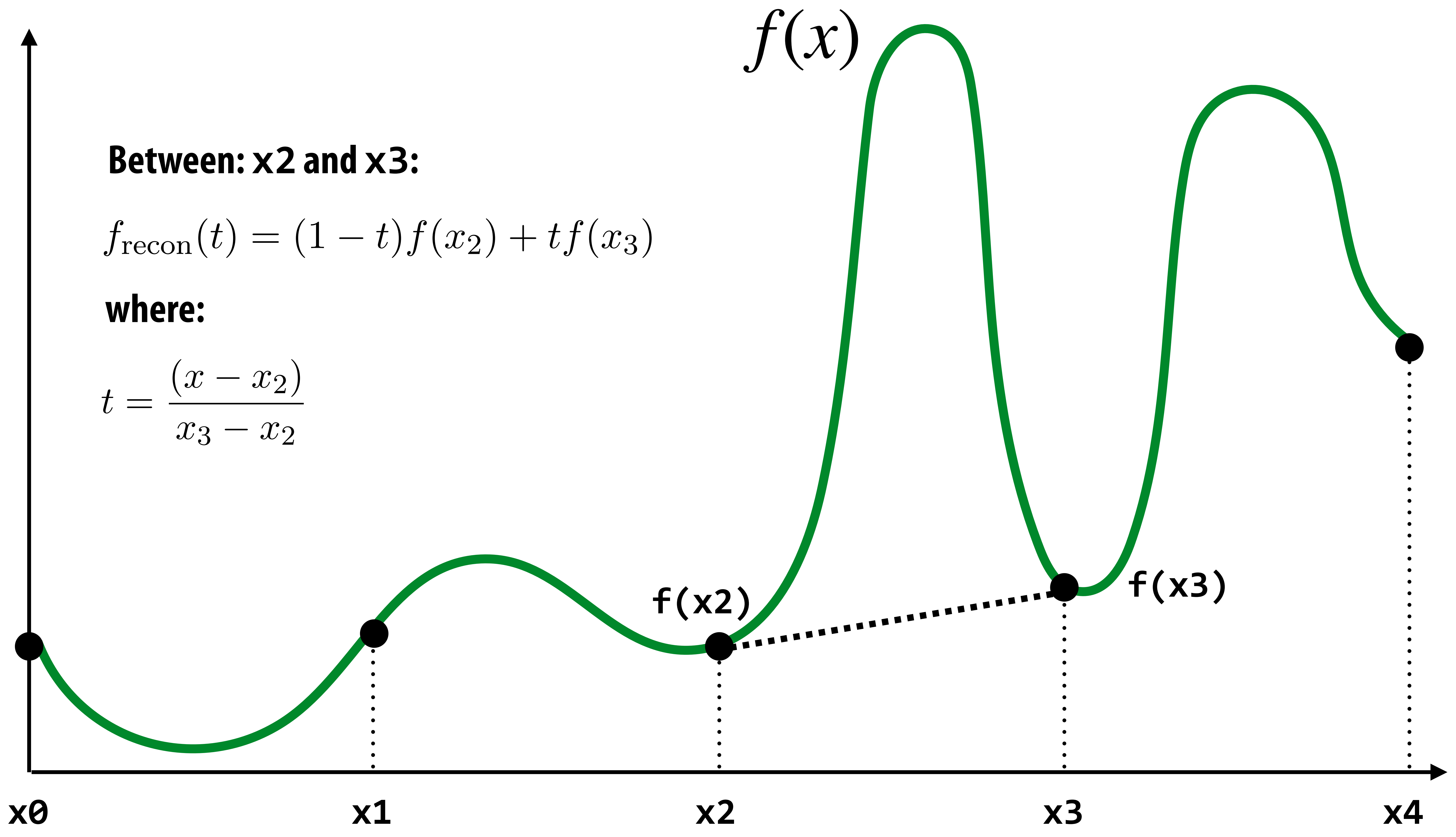
Consider sampling surface_color(x,y)



What is the triangle's color at the point $\mathbf{x}$?

Review: interpolation in 1D

$f_{recon}(x)$ = linear interpolation between values of two closest samples to x



Between: x2 and x3:

$$f_{\text{recon}}(t) = (1-t)f(x_2) + tf(x_3)$$

where:

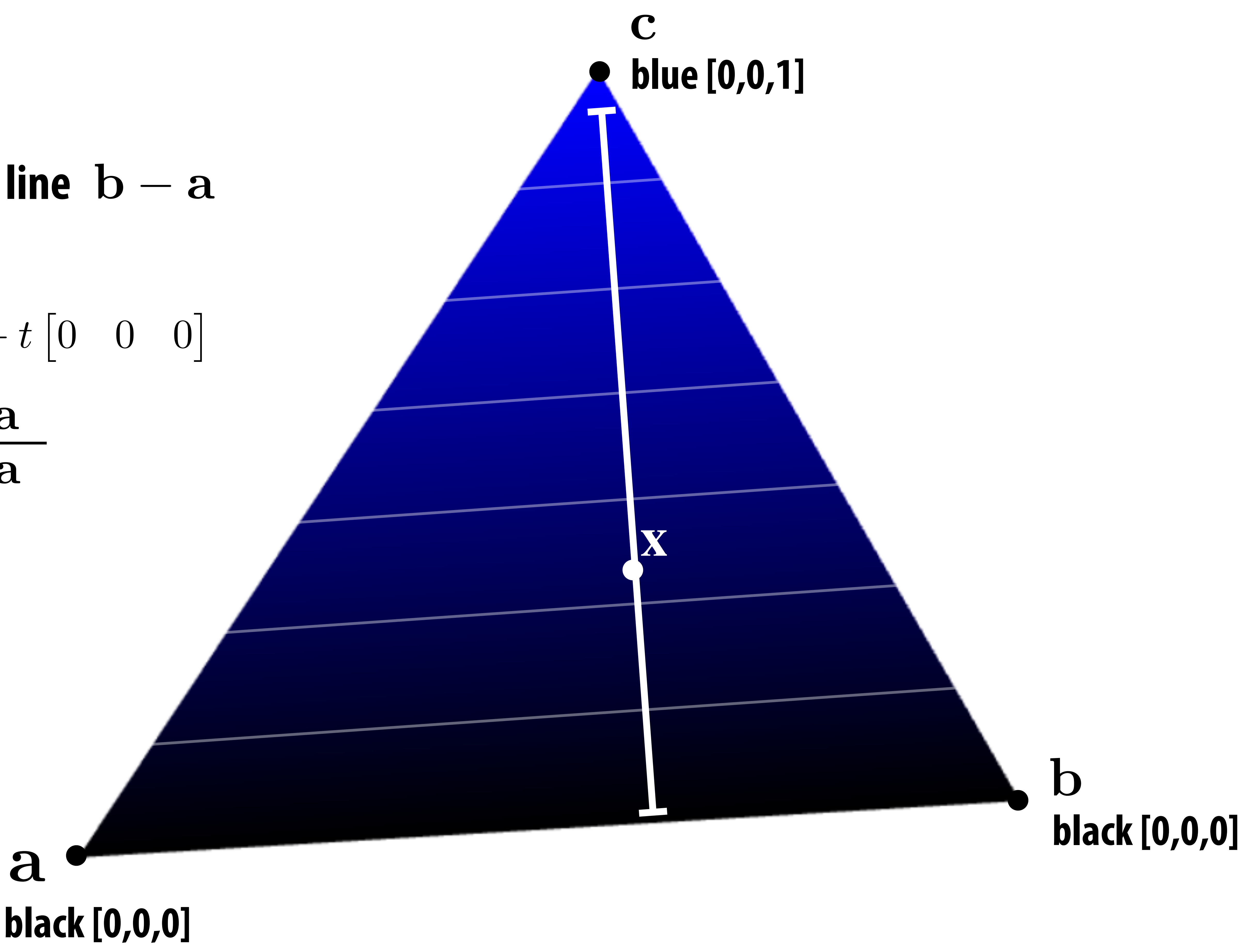
$$t = \frac{(x - x_2)}{x_3 - x_2}$$

Consider similar behavior on triangle

Color depends on distance from line $\mathbf{b} - \mathbf{a}$

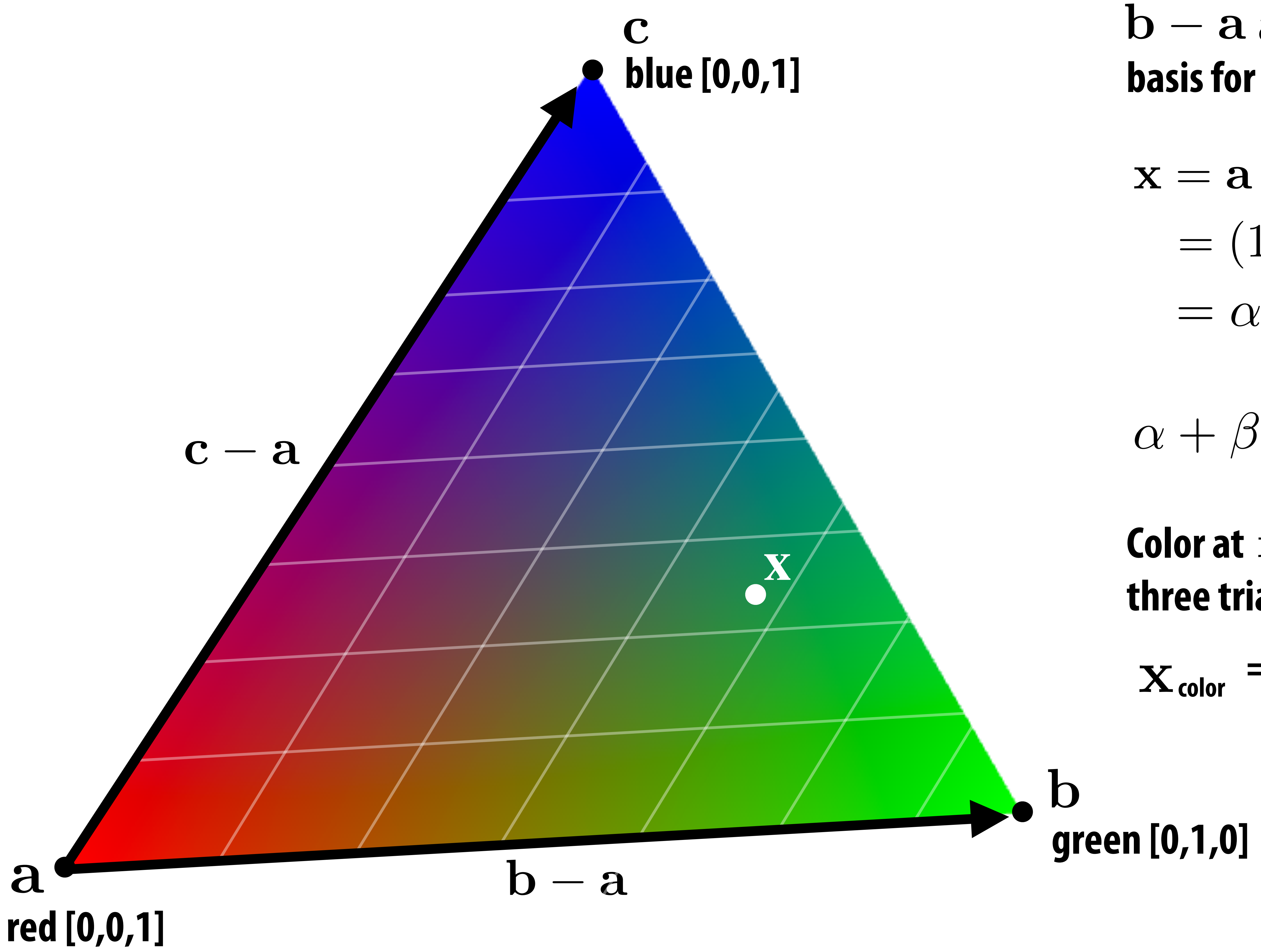
color at $\mathbf{x} = (1-t)\begin{bmatrix}0 & 0 & 1\end{bmatrix} + t\begin{bmatrix}0 & 0 & 0\end{bmatrix}$

$$t = \frac{\text{distance from } \mathbf{x} \text{ to } \mathbf{b}-\mathbf{a}}{\text{distance from } \mathbf{c} \text{ to } \mathbf{b}-\mathbf{a}}$$



How can we interpolate in 2D between three values?

Interpolation via barycentric coordinates



$\mathbf{b} - \mathbf{a}$ and $\mathbf{c} - \mathbf{a}$ form a non-orthogonal basis for points in triangle (origin at $\mathbf{a}$)

$$\begin{aligned} \mathbf{x} &= \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}) \\ &= (1 - \beta - \gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} \\ &= \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} \end{aligned}$$

$$\alpha + \beta + \gamma = 1$$

Color at $\mathbf{x}$ is linear combination of color at three triangle vertices.

$$\mathbf{x}_{\text{color}} = \alpha\mathbf{a}_{\text{color}} + \beta\mathbf{b}_{\text{color}} + \gamma\mathbf{c}_{\text{color}}$$

Direct evaluation of surface attributes (on a 2D triangle)

Let projected screen-space positions of vertices be: $\boldsymbol{a}=(X_0,\ Y_0)$, $\boldsymbol{b}=(X_1,\ Y_1)$, $\boldsymbol{c}=(X_2,\ Y_2)$

Attribute value at point (x,y): $f\,(x,y)$ **is affine combination of value at vertices**

$f\,(x,y) = Ax + By + C$ **(attribute plane equation)**

For any surface attribute (with value defined at triangle vertices as: f_a, f_b, f_c**)**

$$f_a = A\mathbf{a}_x + B\mathbf{a}_y + C$$

$$f_b = A\mathbf{b}_x + B\mathbf{b}_y + C$$

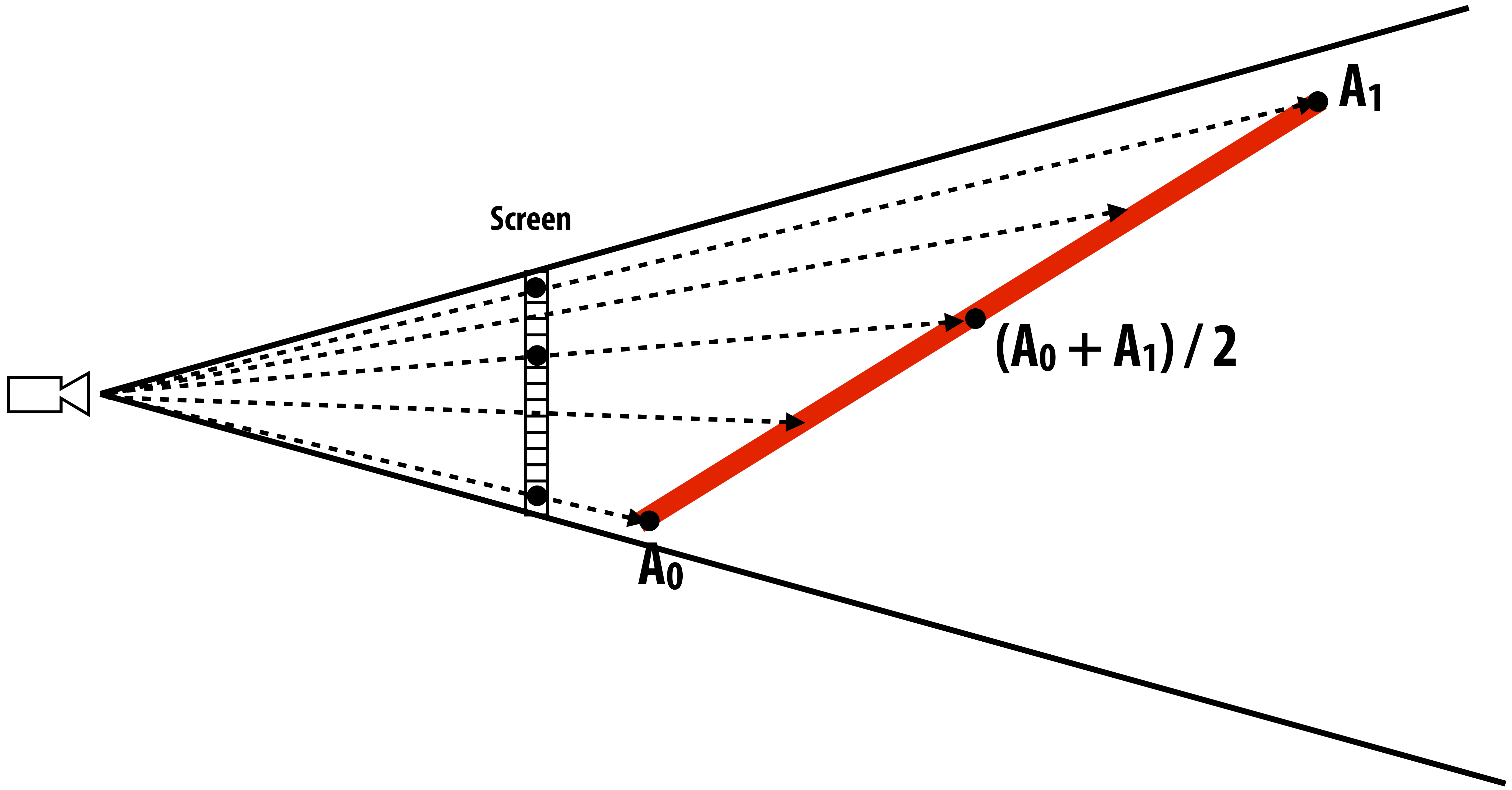
$$f_c = A\mathbf{c}_x + B\mathbf{c}_y + C$$

3 equations, solve for 3 unknowns (A, B, C)

Perspective-incorrect interpolation

Due to projection, interpolation of values on a triangle with vertices at different depths <u>is not</u> an affine function of screen XY coordinates

Attribute values must be interpolated linearly in 3D object space.



Perspective-correct interpolation

Assume triangle attribute varies linearly across the triangle

Attribute's value at 3D (non-homogeneous) point $P = \begin{bmatrix} x & y & z \end{bmatrix}^T$ **is:**

$$f(x, y, z) = ax + by + cz$$

Project $\boldsymbol{P}$**, get 2D homogeneous representation:** $\begin{bmatrix} x_{\text{2D-H}} & y_{\text{2D-H}} & w \end{bmatrix}^T = \begin{bmatrix} x & y & z \end{bmatrix}^T$

Rewrite attribute equation for f **in terms of 2D homogeneous coordinates:**

$$f = ax_{\text{2D-H}} + by_{\text{2D-H}} + cw$$

$$\frac{f}{w} = a\frac{x_{\text{2D-H}}}{w} + b\frac{y_{\text{2D-H}}}{w} + c$$

$$\frac{f}{w} = ax_{\text{2D}} + by_{\text{2D}} + c$$

Where $\begin{bmatrix} x_{\text{2D}} & y_{\text{2D}} \end{bmatrix}^T$ **are projected screen 2D coordinates (after homogeneous divide)**

So … $\dfrac{f}{w}$ **is affine function of 2D screen coordinates:** $\begin{bmatrix} x_{\text{2D}} & y_{\text{2D}} \end{bmatrix}^T$

Efficient perspective-correct interpolation

Attribute values vary linearly across triangle in 3D, but not in projected screen XY

Projected attribute values (f/w) are affine functions of screen XY!

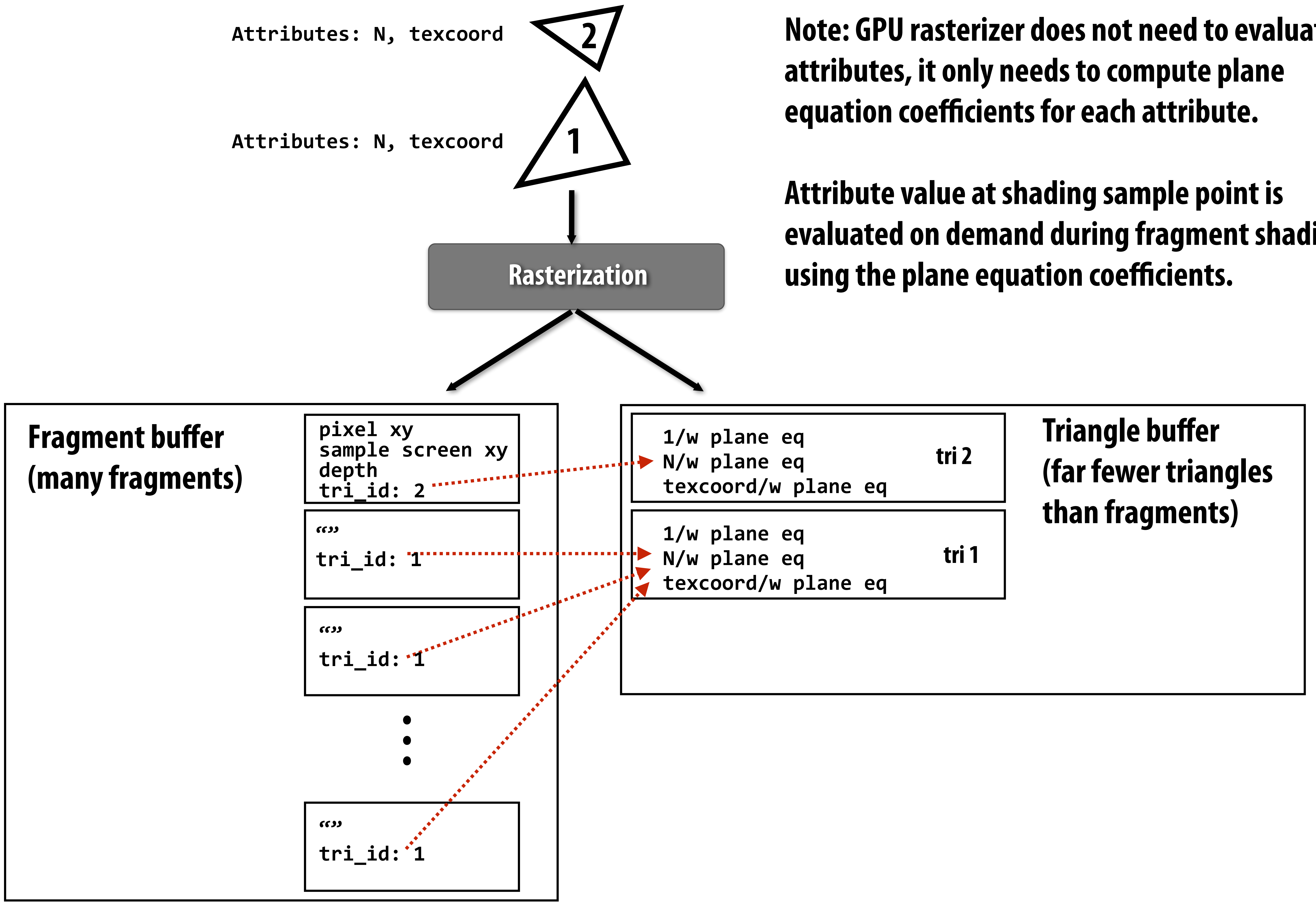
To evaluate surface attribute f at every covered sample:

- **Evaluate** $^1/_w\,(x,y)$ **(from precomputed equation for $^1/_w$)**
- **Reciprocate** $^1/_w\,(x,y)$ **to get** $w(x,y)$
- **For each triangle attribute:**
    - **Evaluate** $^f/_w\,(x,y)$ **(from precomputed equation for $^f/_w$)**
    - **Multiply** $^f/_w\,(x,y)$ **by** $w(x,y)$ **to get** $f\,(x,y)$

Works for any surface attribute f that varies linearly across triangle:

e.g., color, depth, texture coordinates

GPUs store attribute plane equations separately from individual fragments (effectively a form of fragment compression)



Note: GPU rasterizer does not need to evaluate attributes, it only needs to compute plane equation coefficients for each attribute.

Attribute value at shading sample point is evaluated on demand during fragment shading, using the plane equation coefficients.

Modern GPU rasterization

All modern GPUs have fixed-function hardware to perform triangle "setup" and rasterization.

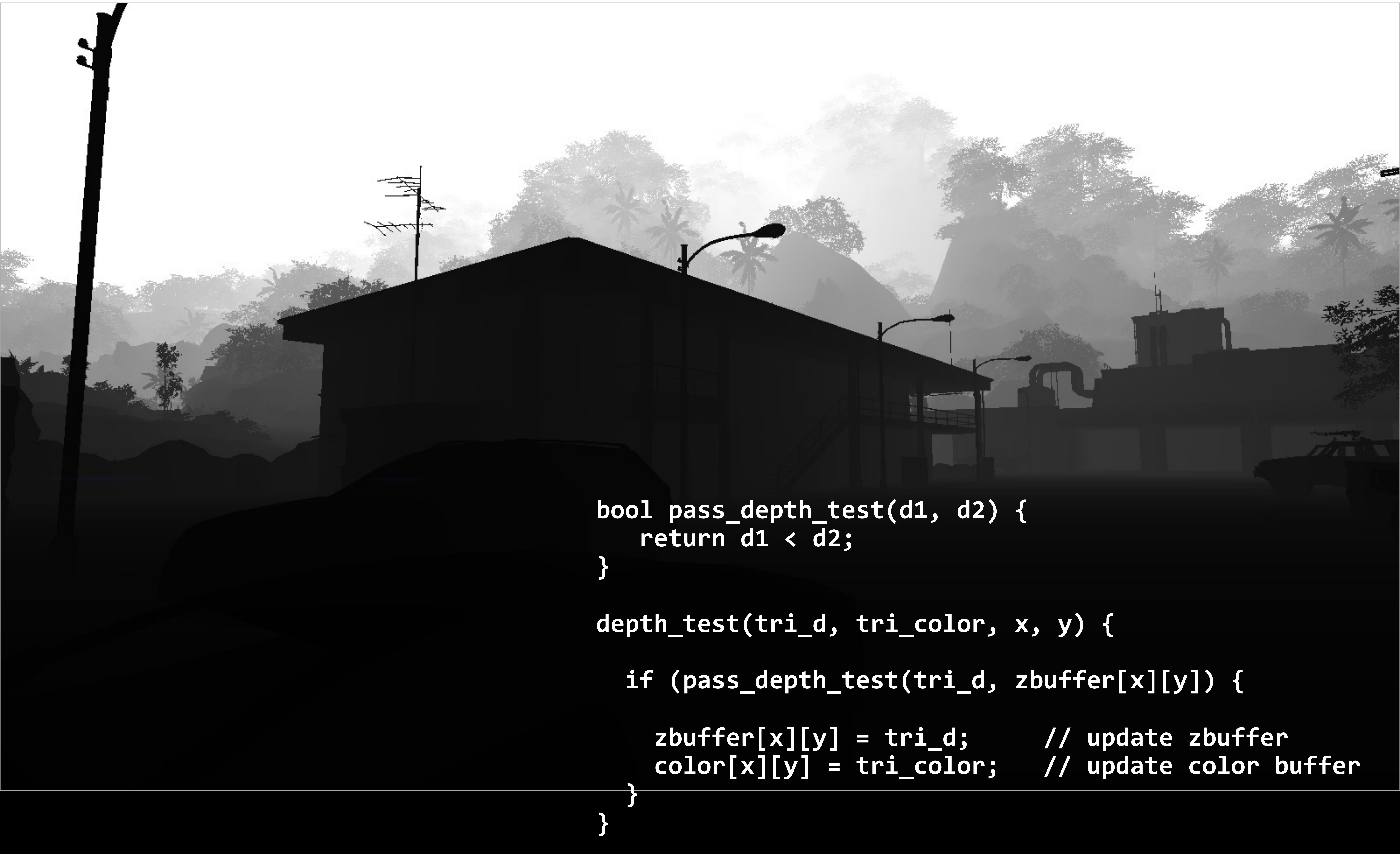
- **Triangle setup:**
  - Transform clip-space vertex positions to screen space
  - Convert vertex positions to fixed point (Direct3D requires 8 bits of subpixel precision**)
  - Compute triangle edge equations (for coverage testing)
  - Compute plane equations for all vertex attributes (including 1/W and depth)
- **Traverse triangle in blocks:**
  - Identify covered samples using edge tests (wide data parallelism in implementation)
  - May attempt to trivially accept/reject block using edge tests on block corners
  - Generate and emit fragments based on coverage (also emit per-triangle data as necessary)
  - Block granularity order is also important for… (topics of future lectures)
    - Shading derivatives, maximizing data locality (cache locality/compression), maximizing control locality (avoiding SIMD divergence during fragment shading)

** 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)

Occlusion

From last time: occlusion via the depth buffer



```
bool pass_depth_test(d1, d2) {
   return d1 < d2;
}

depth_test(tri_d, tri_color, x, y) {

  if (pass_depth_test(tri_d, zbuffer[x][y]) {

    zbuffer[x][y] = tri_d;      // update zbuffer
    color[x][y] = tri_color;    // update color buffer
  }
}
```

Depth buffer for occlusion

- Depth buffer stores depth of scene at each coverage sample point
  - Stored per sample, not per pixel!
- Triangles are planar
  - Each triangle has exactly one depth at each sample point * ✓
    (so triangle order 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 (overall) ✓

* Assumes edge-on triangles have been discarded due to zero area

Depth buffer for occlusion

- **Z-buffer algorithm has high bandwidth requirements (particularly when super-sampling triangle coverage)**
  - Number of Z-buffer reads/writes for a frame 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)

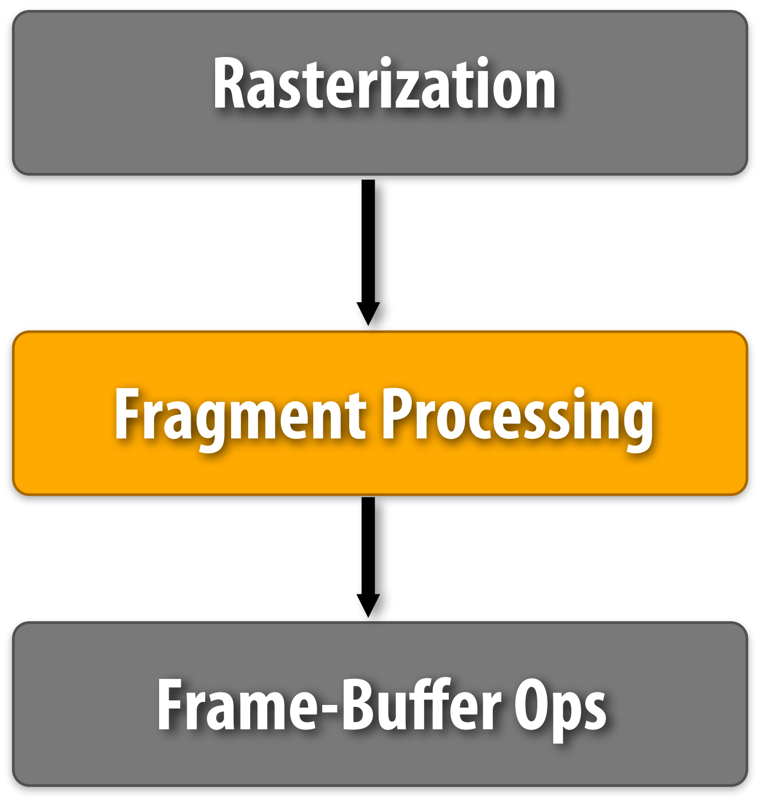
- **Bandwidth estimate:**
  - 60 Hz $\times$ 2 MPixel image $\times$ avg. depth complexity 4 (assume: replace 50% of time) $\times$ 32-bit Z = 2.8 GB/s
  - If super-sampling at 4 times per pixel, multiply by 4
  - Consider five shadow maps per frame (1 MPixel, not super-sampled): additional 8.6 GB/s
  - 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 to reduce bandwidth (coming slides)**

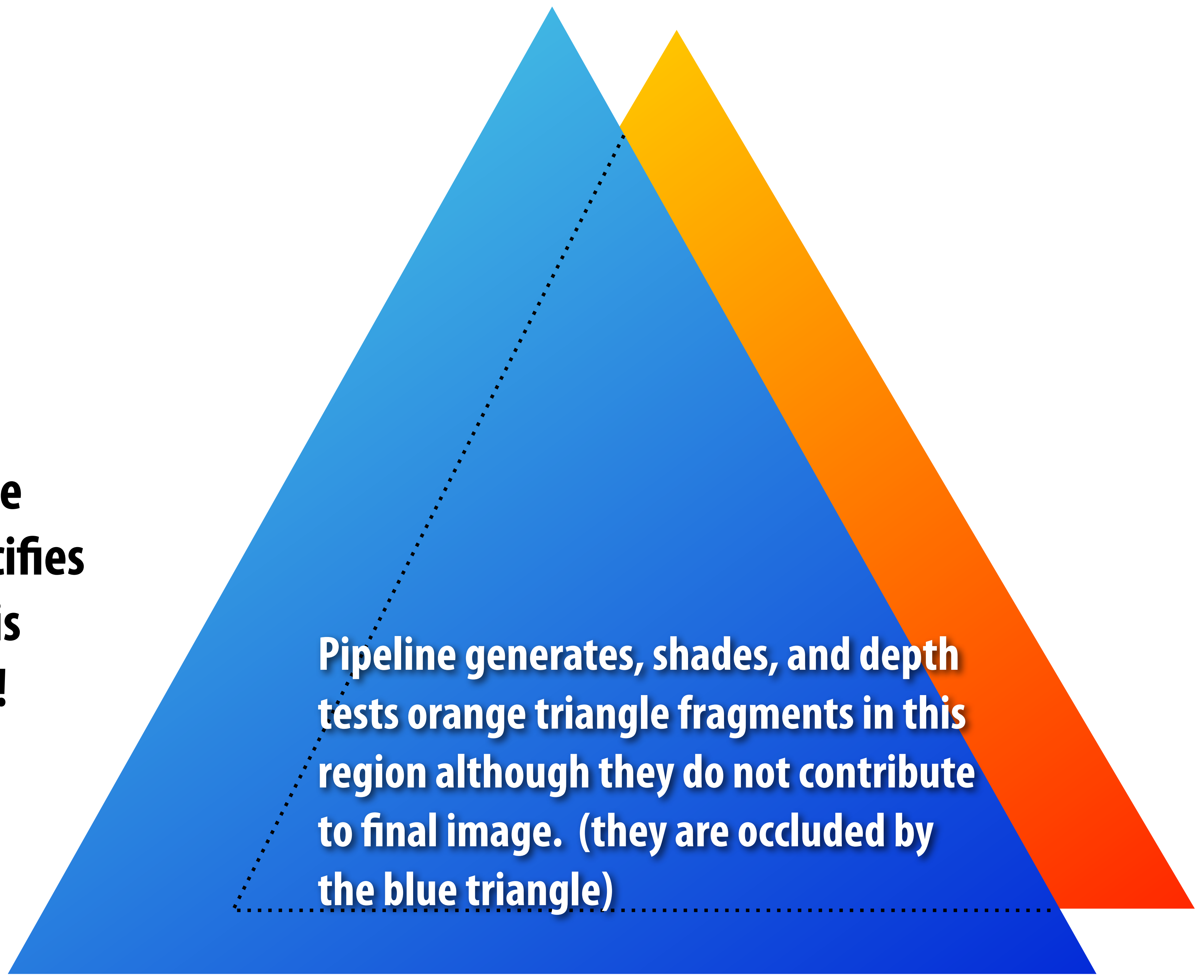
Early occlusion culling

Early occlusion-culling ("early Z")

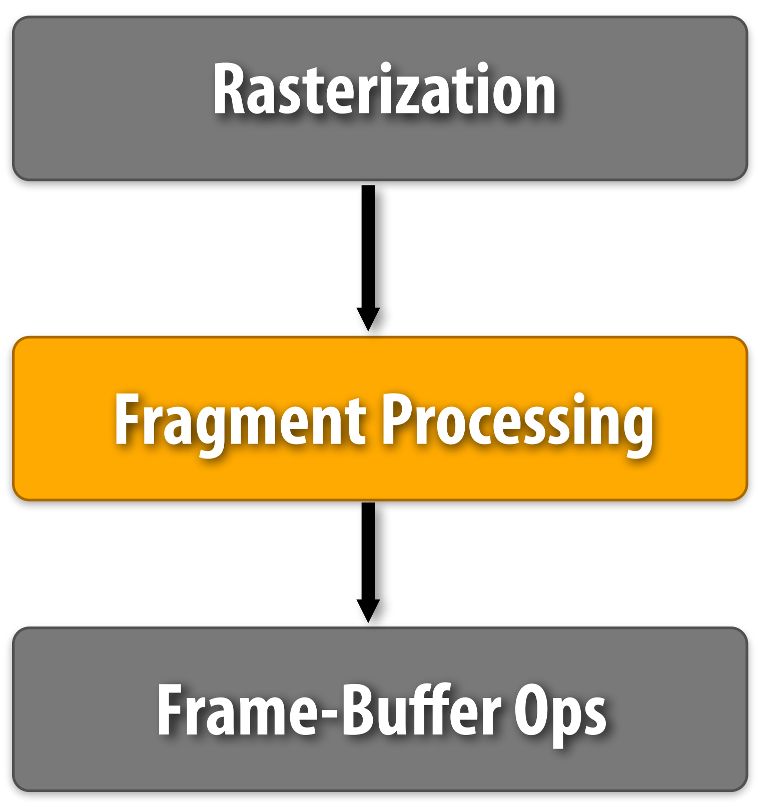
Idea: discard fragments that will not contribute to image as quickly as possible in the pipeline



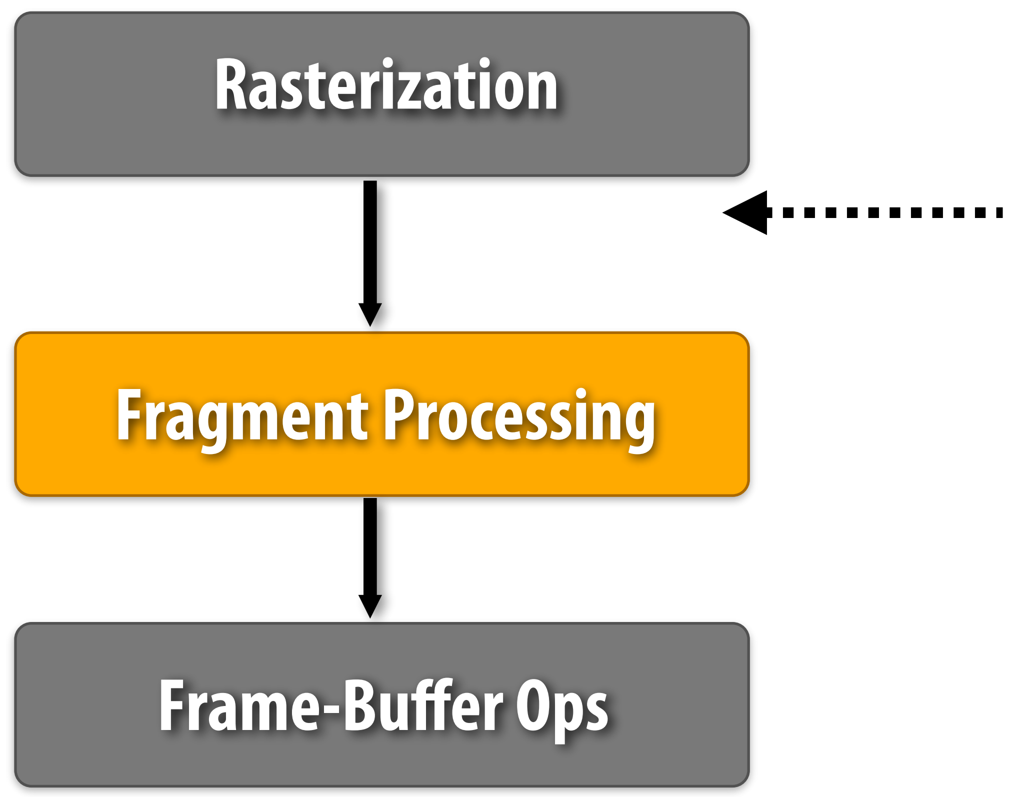
Graphics pipeline abstraction specifies that depth test is performed here!


Pipeline generates, shades, and depth tests orange triangle fragments in this region although they do not contribute to final image. (they are occluded by the blue triangle)

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 <u>before</u> 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 this early Z optimization: enabling alpha test, 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 as close to front-to-back order as possible)

Summary: early occlusion culling

- Key observation: can reorder pipeline operations without impacting correctness: perform depth test <u>prior</u> to fragment shading
- Benefit: reduces fragment processing work
  - Effectiveness of optimization is dependent on triangle ordering
  - Ideal geometry submission order: front-to-back order
- <u>Does not</u> reduce amount of bandwidth used to perform depth tests
  - The same depth-buffer reads and writes still occur (they just occur before fragment shading)
- Implementation-specific optimization, but programmers know it is there
  - Commonly used two-pass technique: rendering with a "Z-prepass"
    - Pass 1: render all scene geometry, with fragment shading and color buffer writes disabled (Put the depth buffer in its end-of-frame state)
    - Pass 2: re-render scene with shading enabled and with depth-test predicate: less than-or-equal
  - Overhead: must process and rasterizer scene geometry twice
  - Benefit: minimizes expensive fragment shading work by only shading visible fragments

Hierarchical early occlusion culling: "hi-Z"

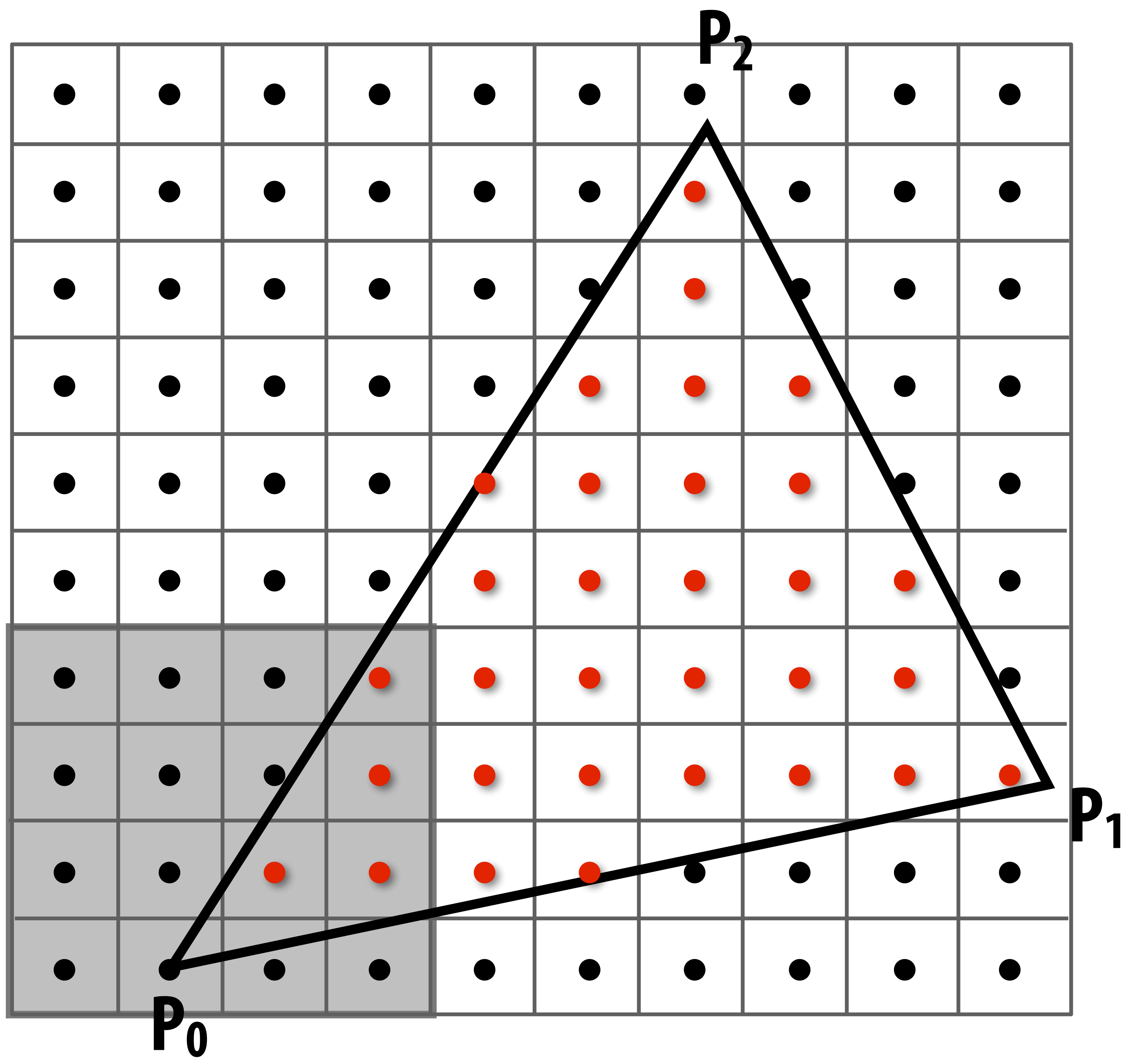
Recall hierarchical traversal during rasterization

<u>Z-Max culling</u>:

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

During traversal, for each tile:

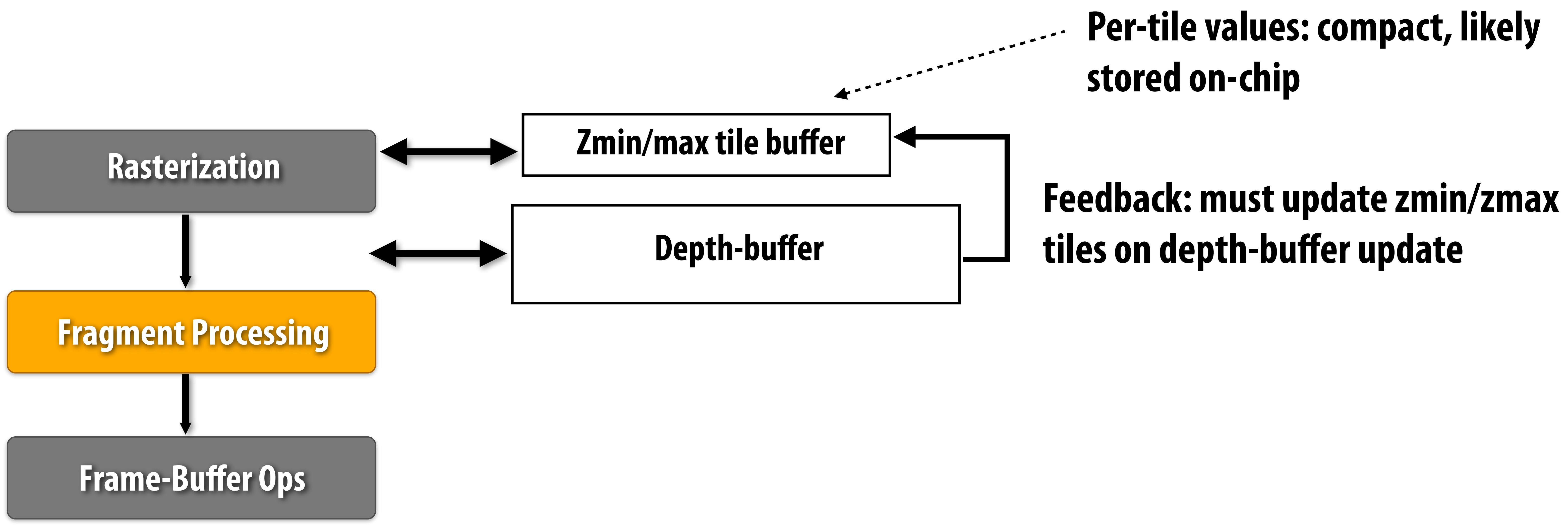
1. Compute closest point on triangle in tile: `tri_min` (using Z plane equation)
2. If `tri_min > z_max`, then triangle is completely occluded in this tile. (The depth test will fail for all samples in the tile.) Proceed to next tile without performing coverage tests for individual samples in tile.



<u>Z-min optimization</u>:

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


Per-tile values: compact, likely stored on-chip
Rasterization
Zmin/max tile buffer
Depth-buffer
Feedback: must update zmin/zmax tiles on depth-buffer update
Fragment Processing
Frame-Buffer Ops

Remember: these are GPU implementation details (common optimizations performed by most GPUs). 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 memory bandwidth requirements (don't need to read individual depth samples)
  - Eliminates <u>less</u> fragment processing work than early Z (Since hierarchical Z is a conservative approximation to early X results, it will only discard a subset of the fragments early Z does)

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

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

Frame-buffer compression

Depth-buffer compression

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

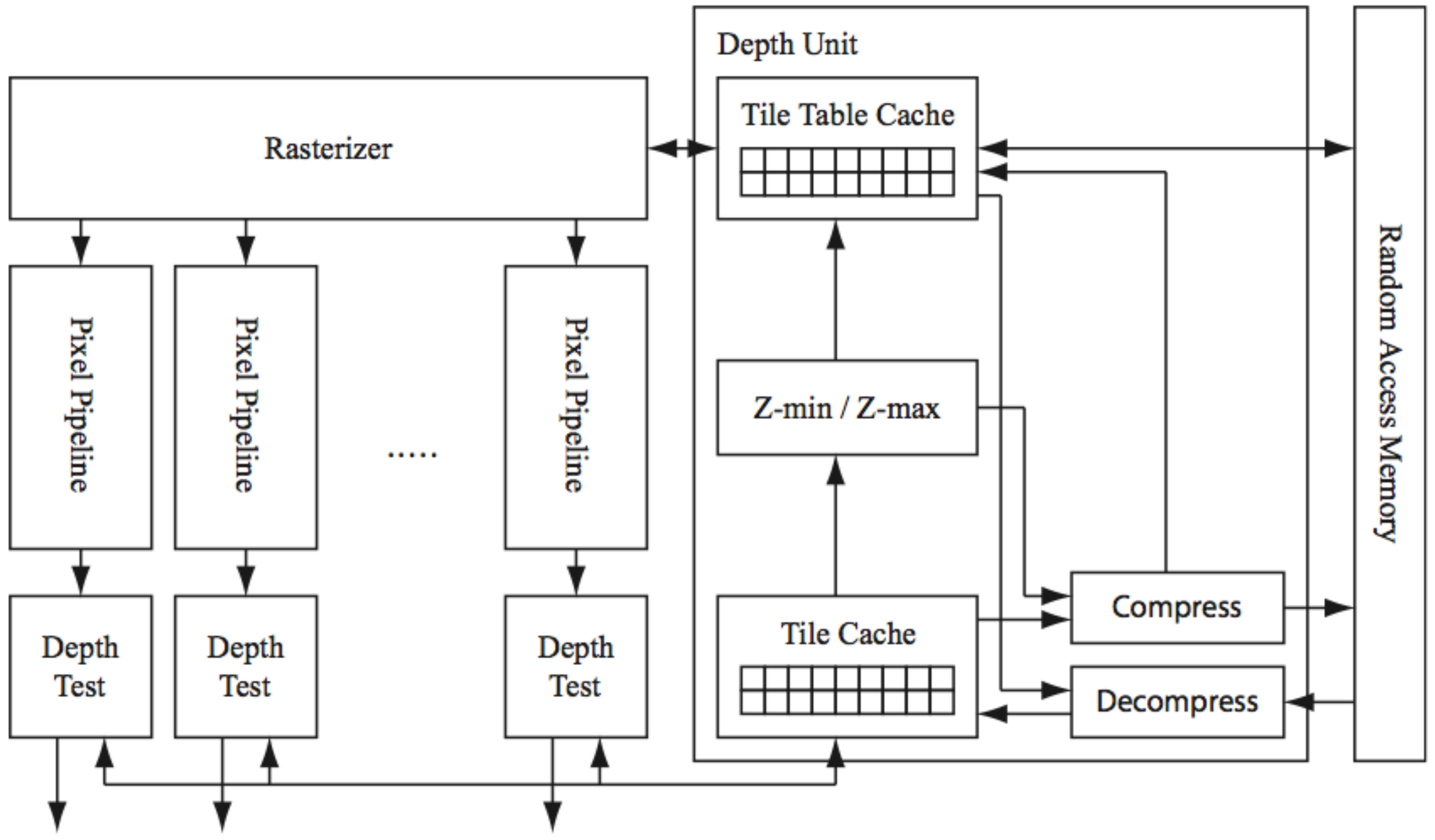
    (Need for real-time guarantees in graphics applications requires application to plan for worst case anyway)

- **Lossless compression**
  - **Question: why not lossy?**

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

Depth-buffer compression is tile based

Main idea: exploit similarity of values within a screen tile



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

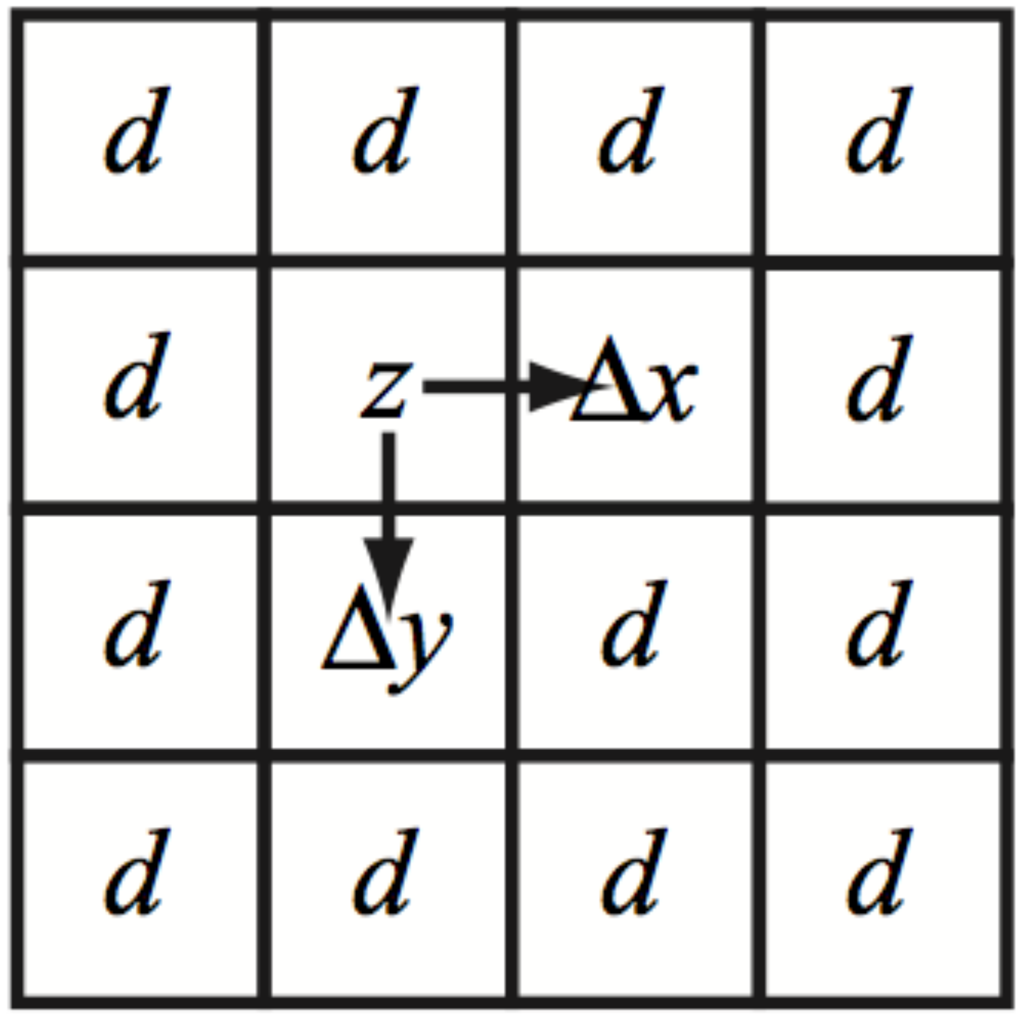
On tile load:

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

Figure credit: [Hasselgren et al. 2006]

Anchor encoding

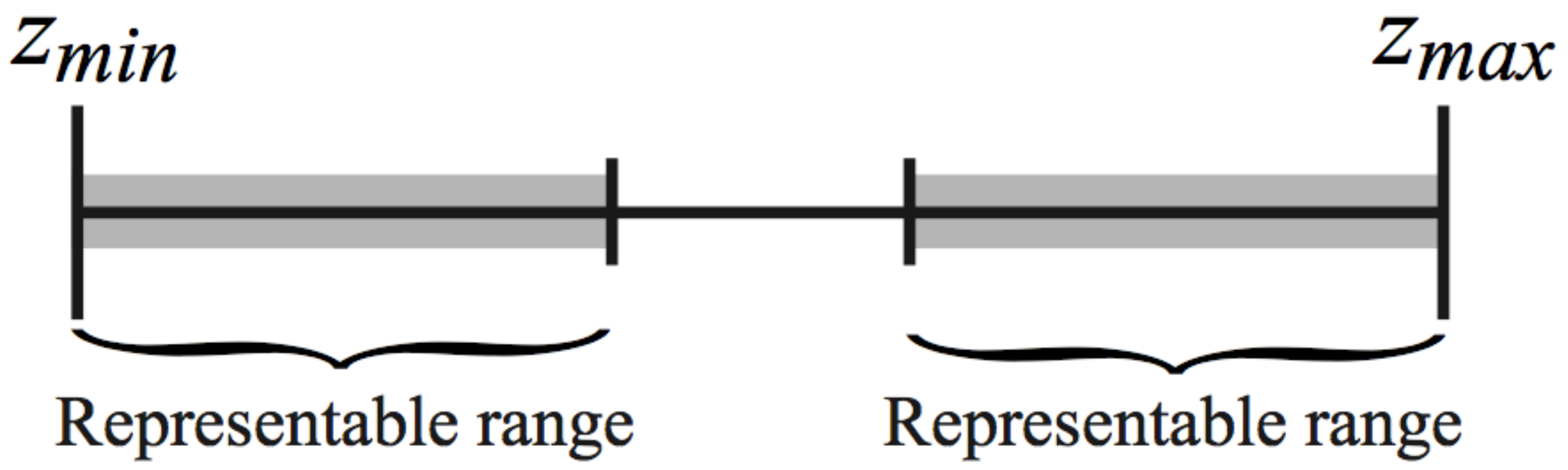
- Choose anchor value and compute DX, DY from adjacent pixels (fits a plane to the data)
- Use plane to predict depths at other pixels, store offset d from prediction at each pixel
- Scheme (for 24-bit depth buffer)
  - Anchor: 24 bits (full resolution)
  - DX, DY: 15 bits
  - Per-sample offsets: 5 bits



[Van Dyke and Margeson]

Depth-offset compression

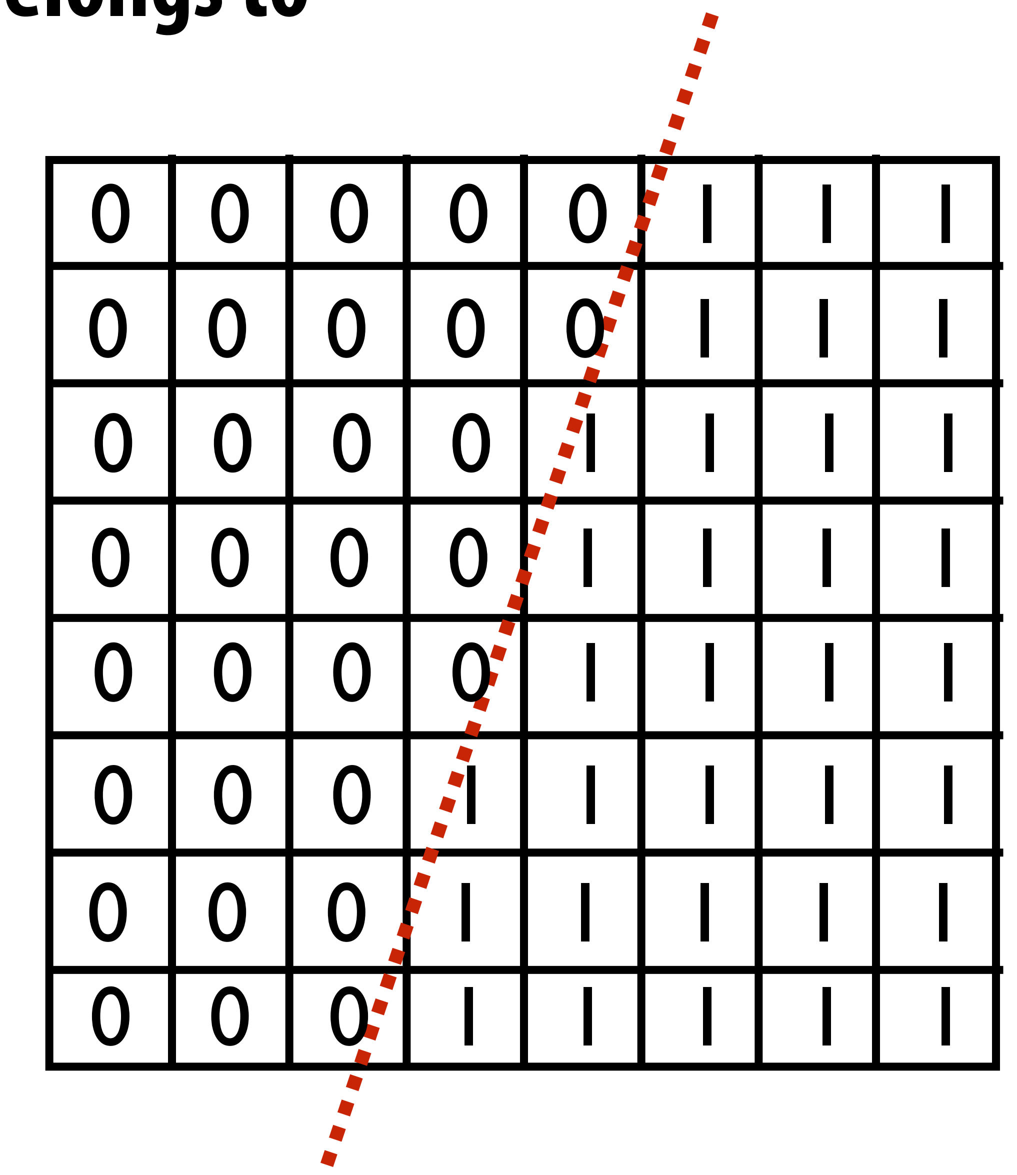
- Assume depth values have low dynamic range relative to tile's zmin and zmaz (assume two surfaces)
- Store zmin/zmax (need to anyway for hierarchical Z)
- Store low-precision (8-12 bits) offset value for each sample
  - MSB encodes if offset is from zmin or zmax



[Morein and Natali]

Explicit 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



0	0	0	0	0	1	1	1
0	0	0	0	0	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	0	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1
0	0	0	1	1	1	1	1

Summary: reducing the bandwidth requirements of depth testing

- Caching: access DRAM less often (by caching depth buffer data)
- Hierarchical Z techniques (zmin/zmax culling): “early outs” result in accessing individual sample data less often
- Data compression: reduce number of bits that must be transferred from memory to read/write a depth sample

---

- The pipeline’s output color buffer (output image) is also compressed using similar techniques
  - Depth buffer typically achieves higher compression ratios than color buffer. Why?

Cross-cutting issues

- **Hierarchical traversal during rasterization**
  - Leveraged to reduce coverage testing and also occlusion work
  - Tile size often 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**
  - Modern GPU implementations are still optimized for triangles of area ~ tens of pixels