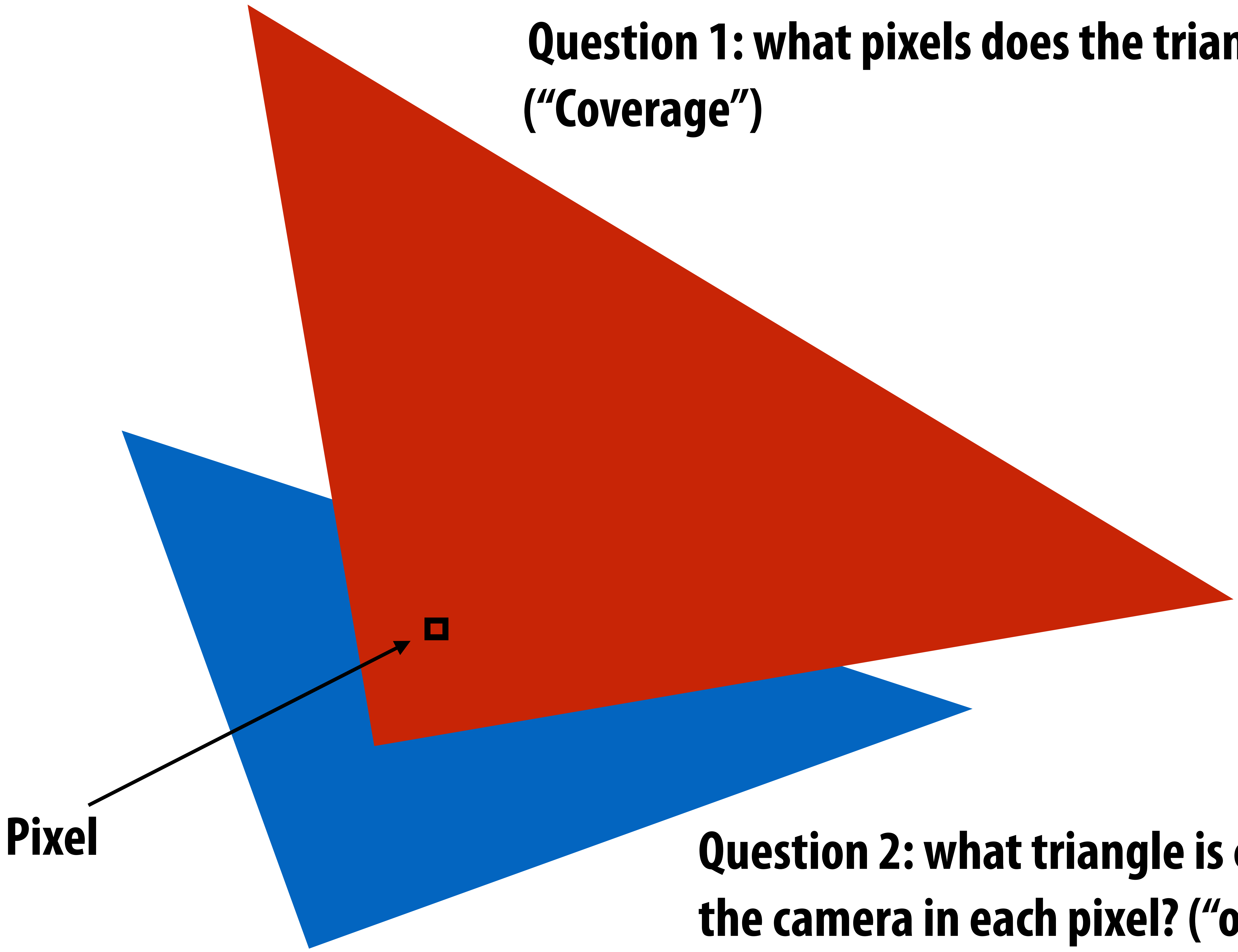
Lecture 27:

Parallel Triangle Rendering on a Modern GPU

Parallel Computer Architecture and Programming
CMU 15-418/15-618, Spring 2015

Let's draw a triangle on the screen

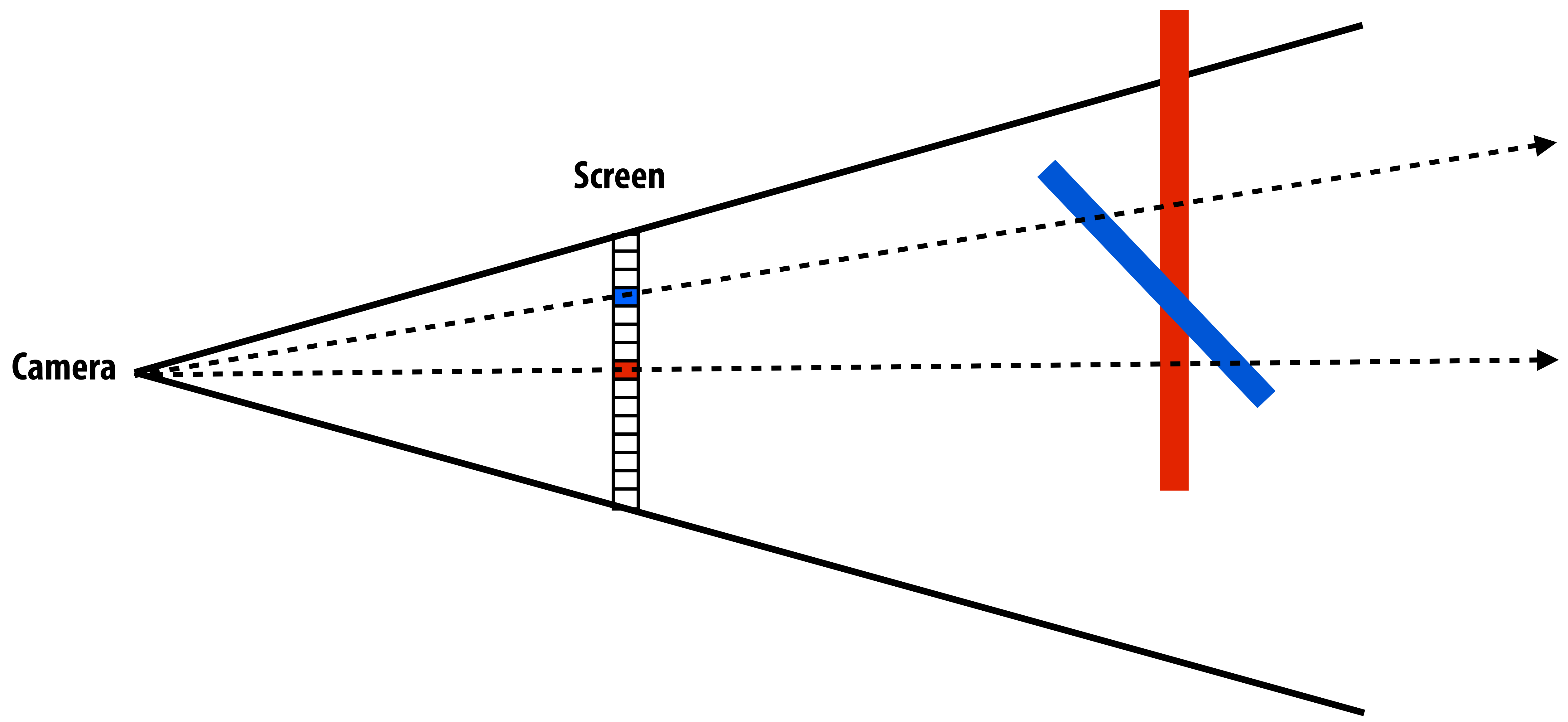


Question 1: what pixels does the triangle overlap? ("Coverage")

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

The visibility problem

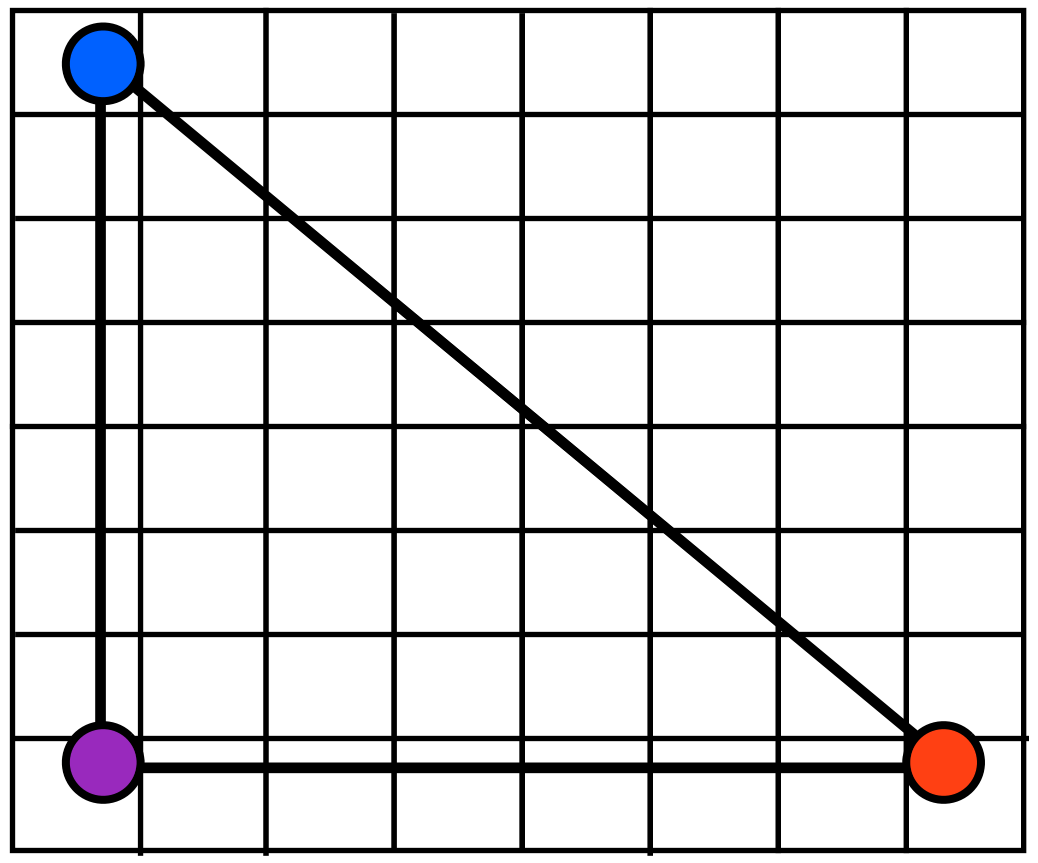
- Very imprecise definition: computing what scene geometry is visible within each screen pixel
  - What scene geometry projects into a screen pixel? (screen coverage)
  - Which geometry is actually visible from the camera at that pixel? (occlusion)



Coverage: input = vertices on screen

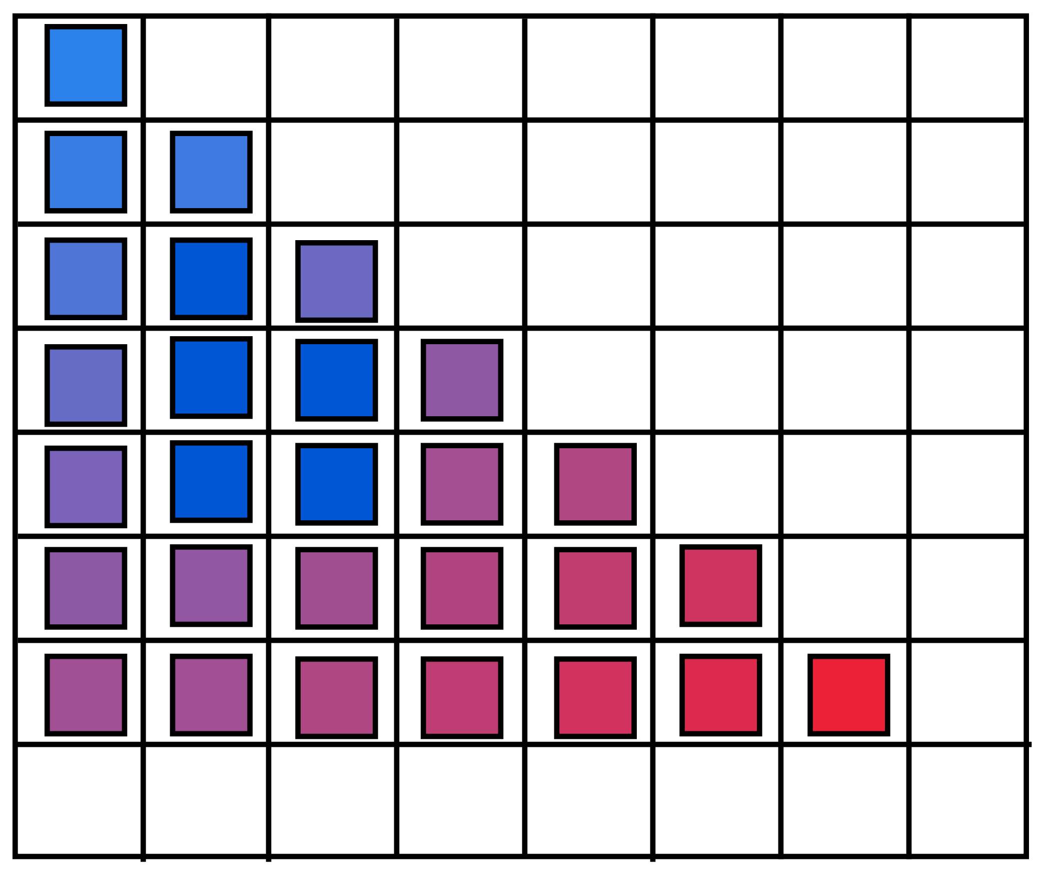
```
struct input_vertex {
   float3 color;
   float3 xyz;
};
```

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



Then given a projected triangle...

Coverage: output = one fragment for each overlapped pixel

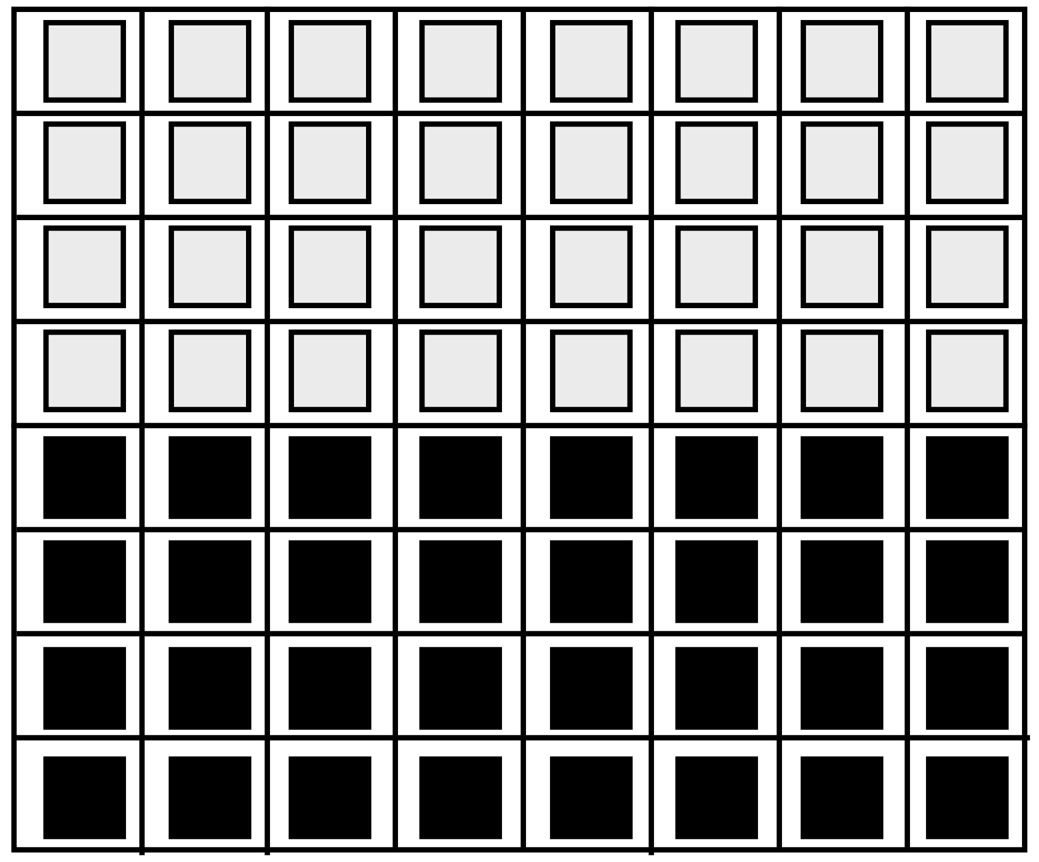


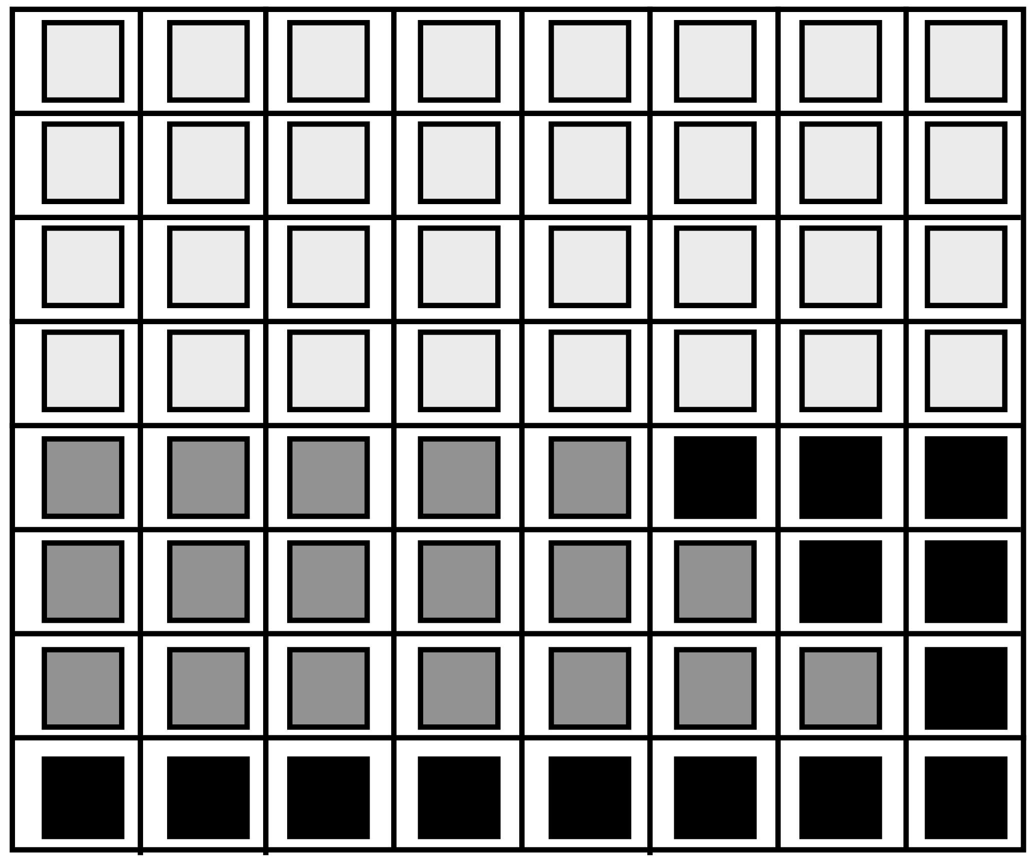
```
struct output_fragment
{

   float3 color;      // color of surface at pixel
   int x, y;          // pixel position of fragment
   float depth;       // triangle depth for fragment
};
```

Rasterization is the act of computing coverage:
Rasterization converts the projected triangle into fragments.

Occlusion using a Z-buffer

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





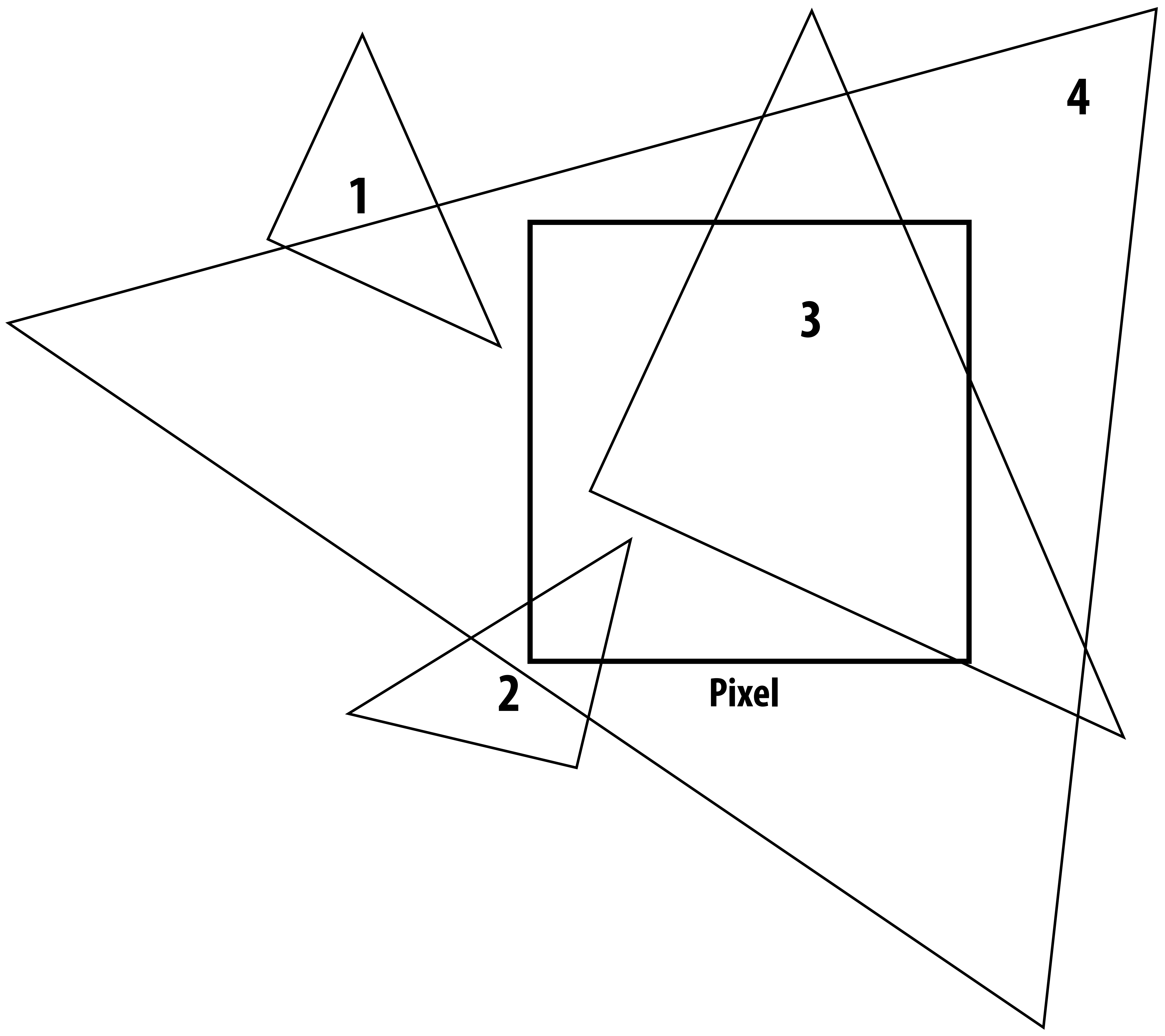
Near values = ⬜ Far values = ⬛

Let depth of new triangle = ▨

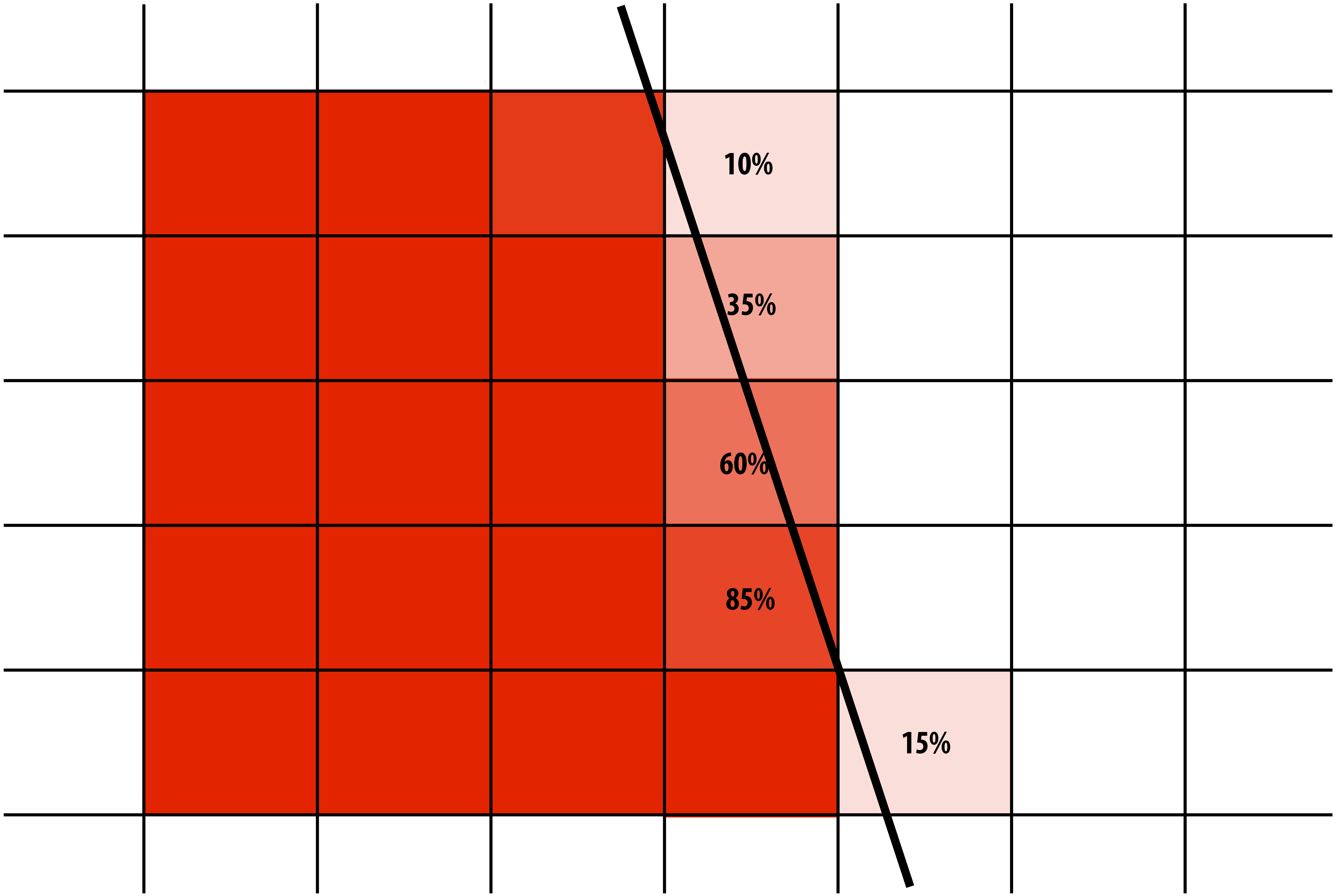
```
bool pass_depth_test(float a, float b) {
   return a < b;    // less-than predicate (configurable in a modern pipeline)
}

// "depth test" executed per fragment
if (pass_depth_test(input_frag.depth, zbuffer[input_frag.x][input_frag.y])
{
   int x = input_frag.x; int y = input_frag.y;
   zbuffer[x][y] = input_frag.depth;
   color[x][y] = blend(color[x][y], input_frag.color);
}
```

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

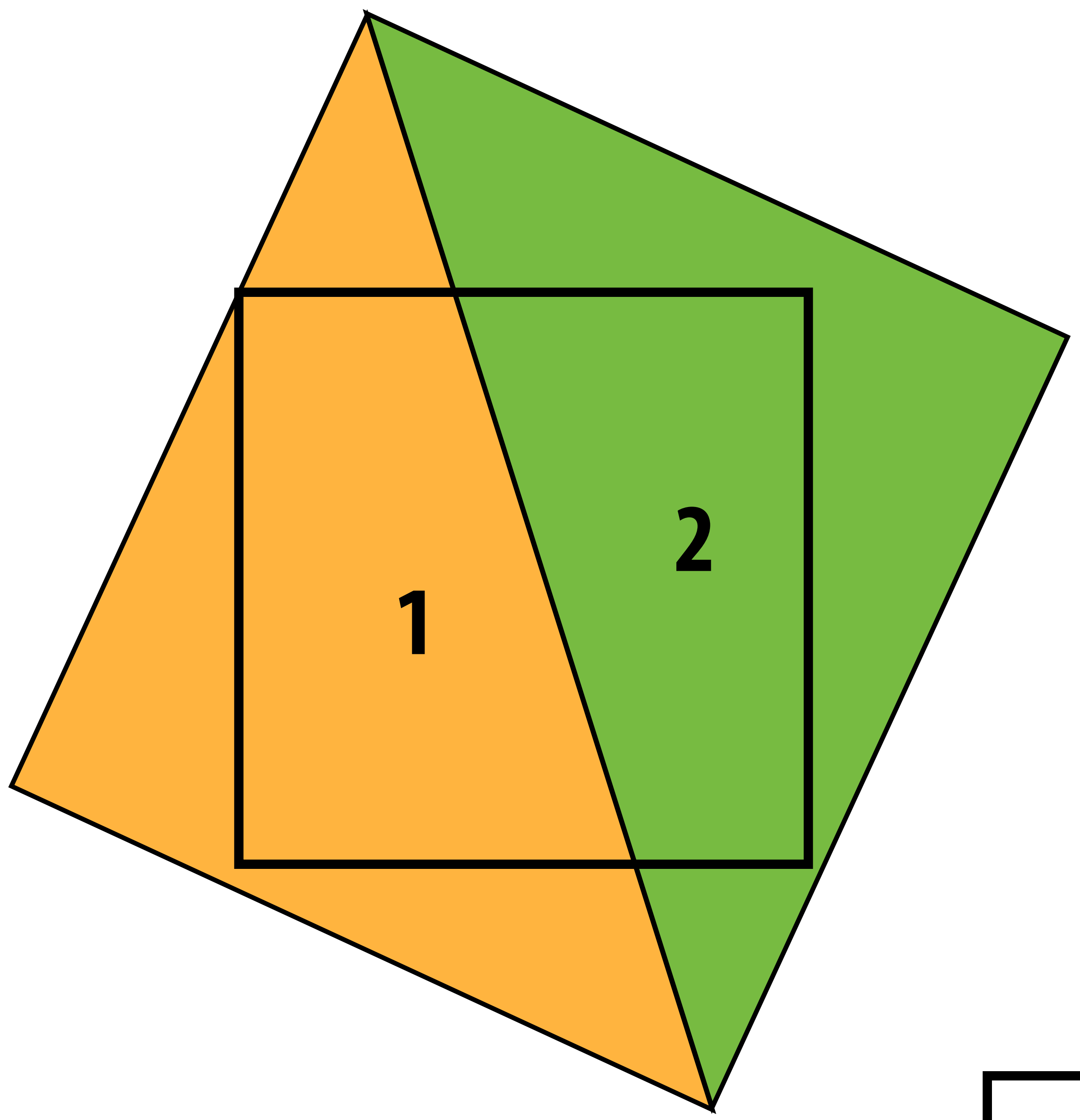
Question: which triangles "cover" this pixel?

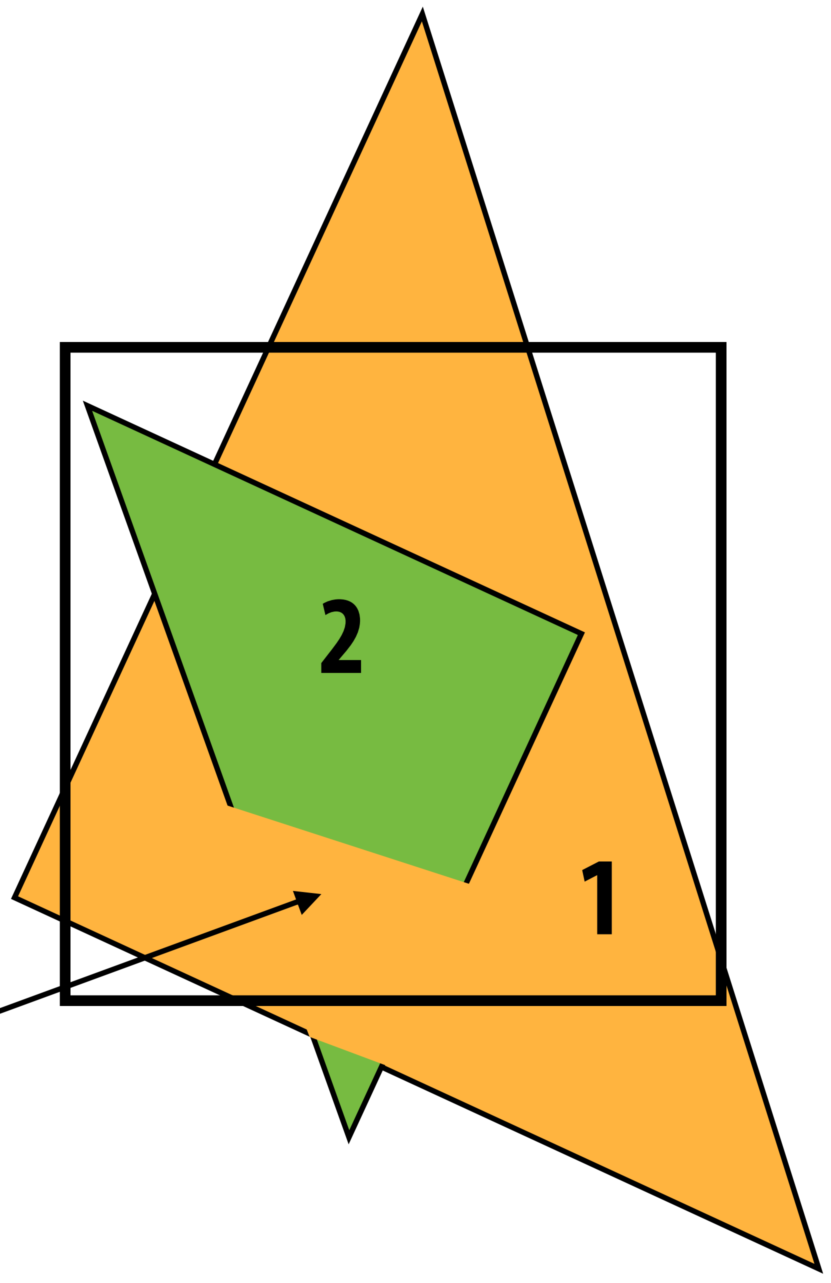


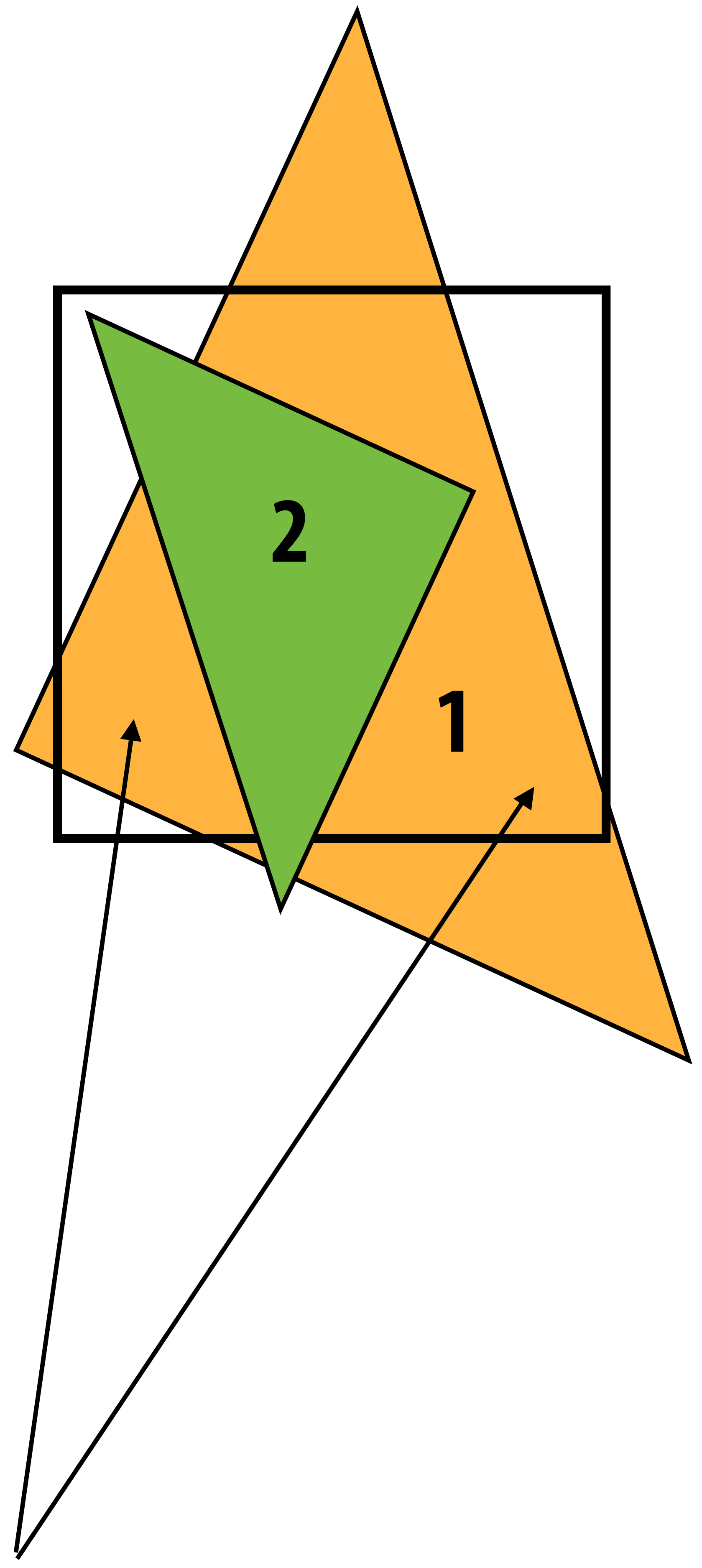
One option: analytically compute fraction of pixel covered by triangle



Analytical schemes get tricky when considering occlusion





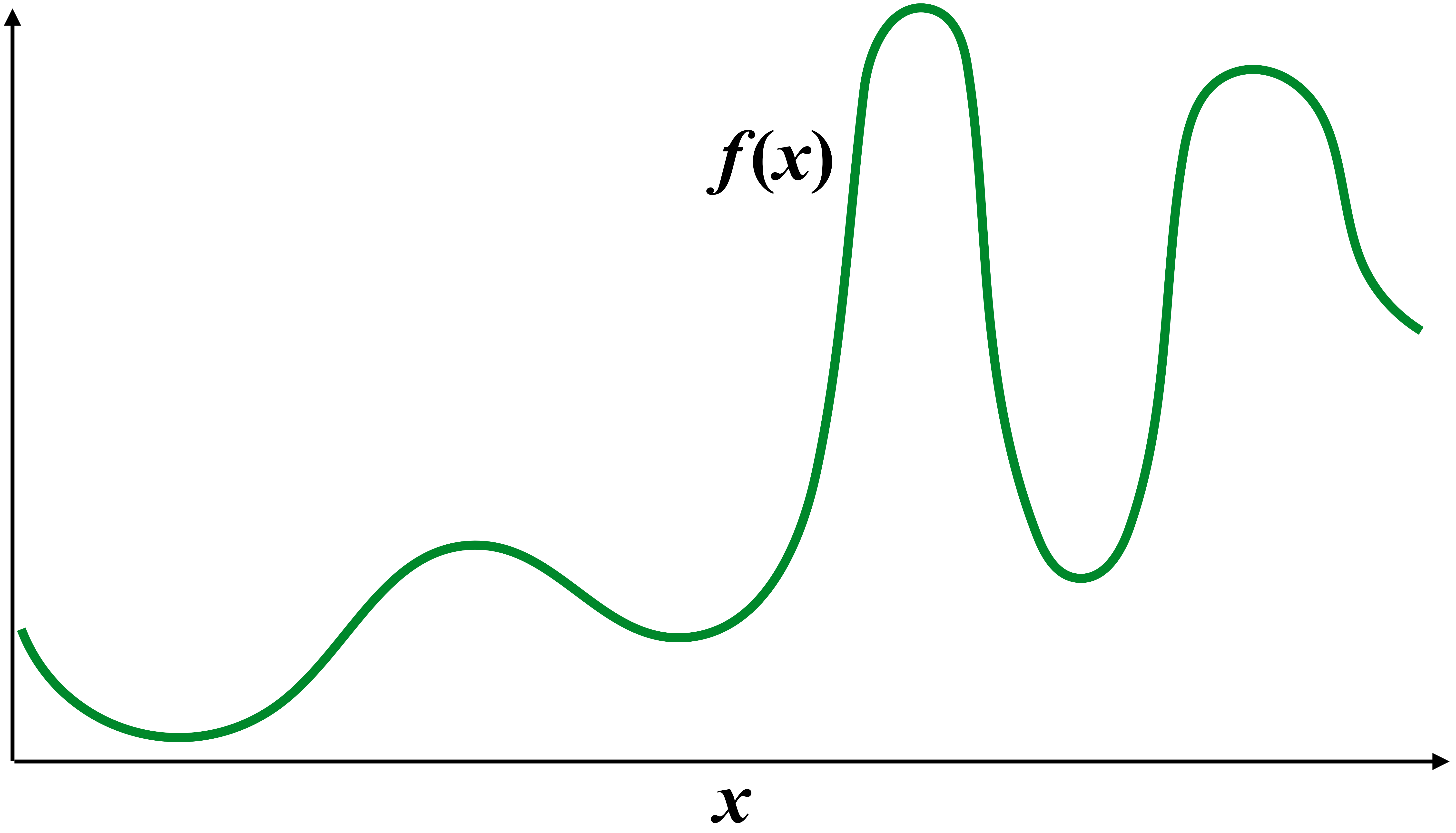


Interpenetration: even worse

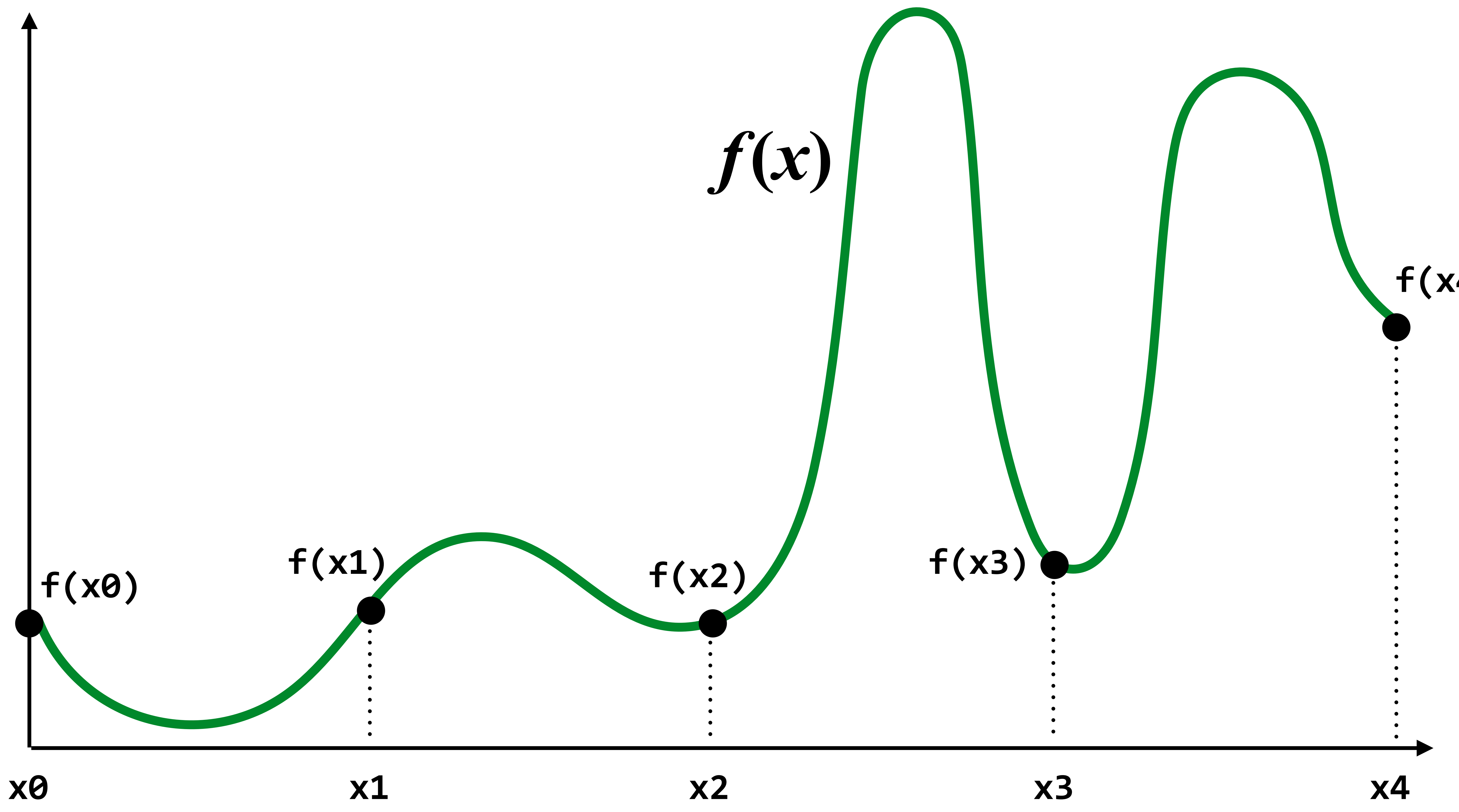
Two regions of [1] contribute to pixel. One of these regions is not even convex.

Sampling 101

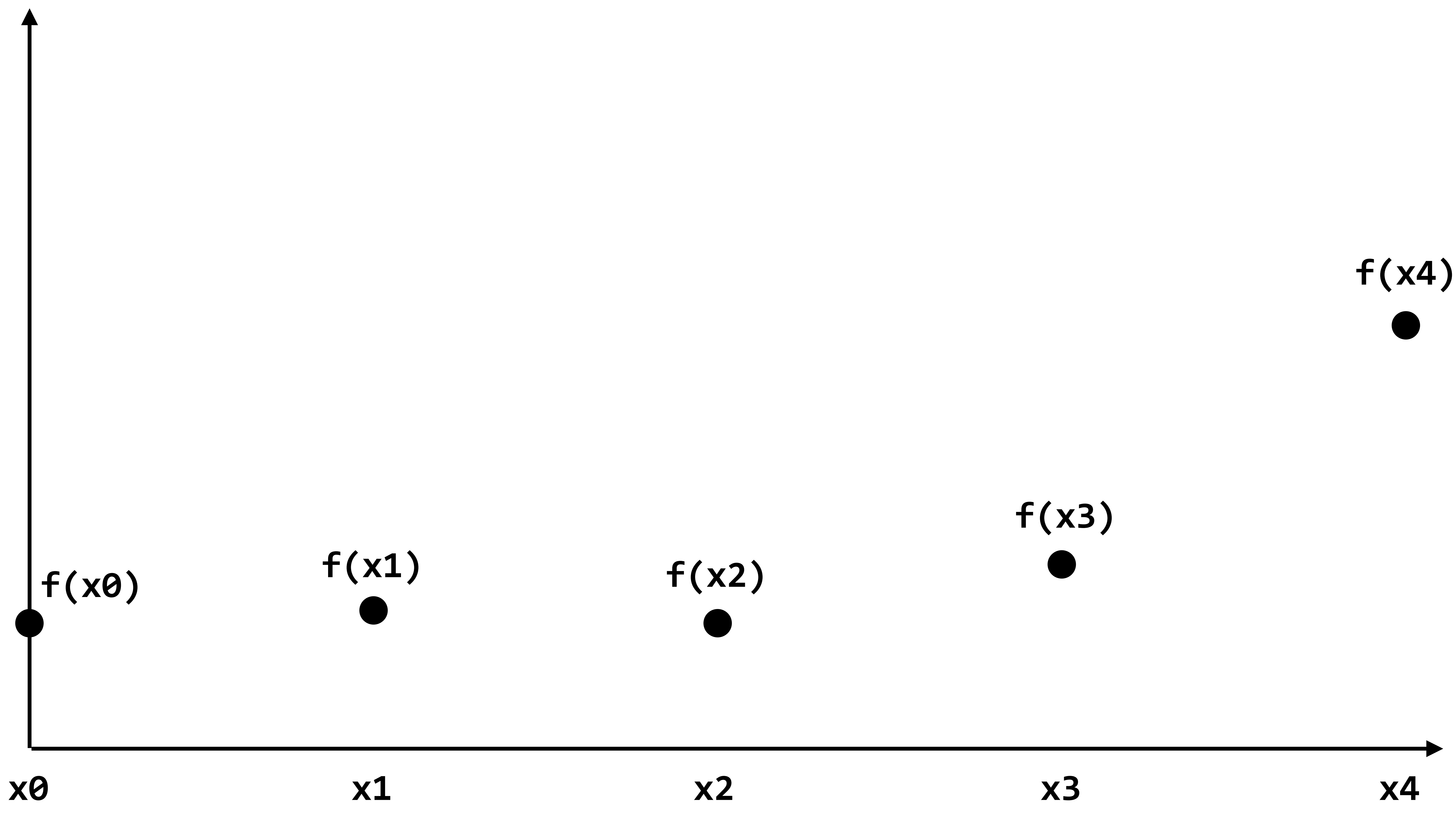
Continuous 1D signal



Measurements of 1D signal (5 measurements)

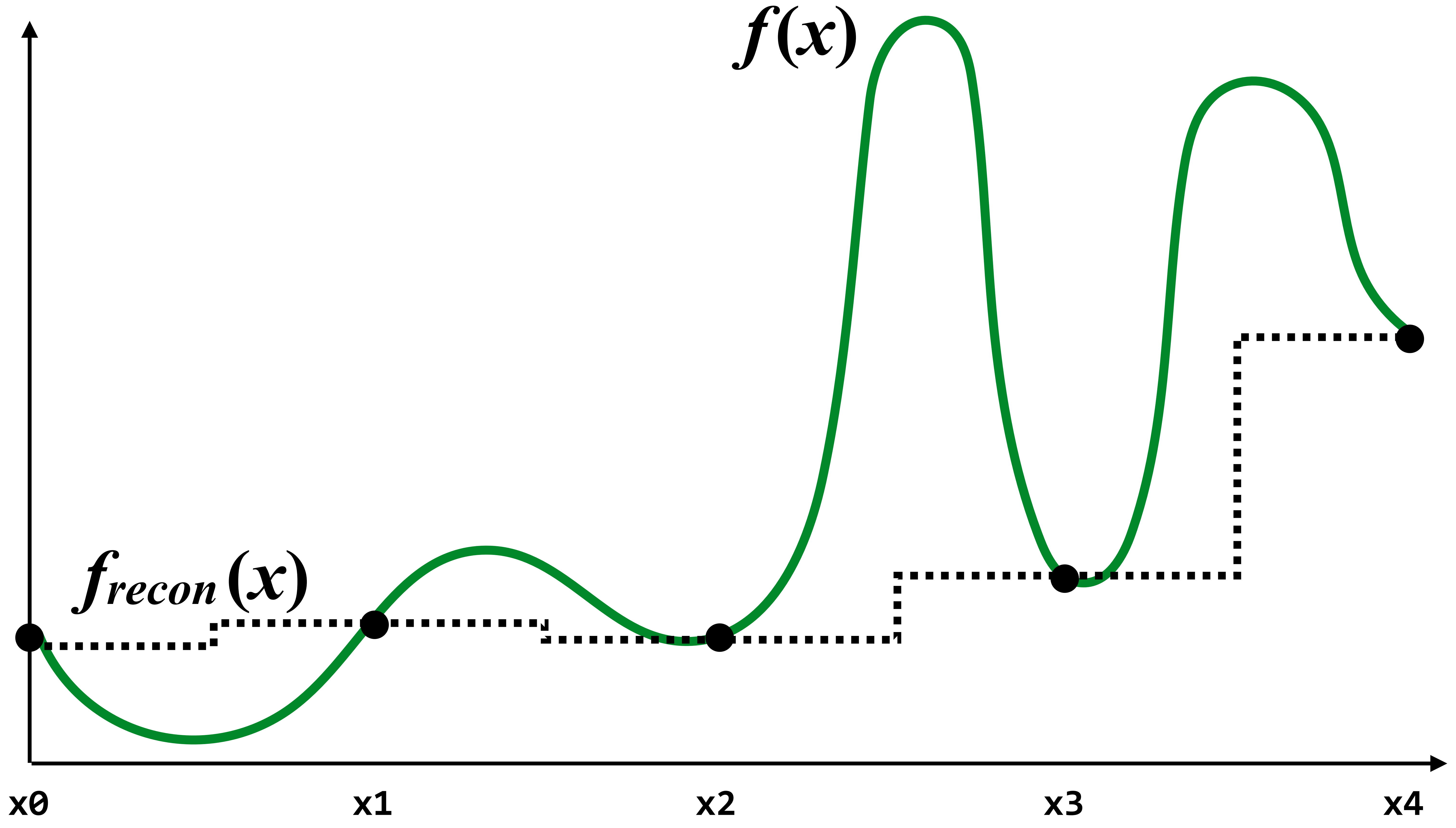


Reconstructing the signal



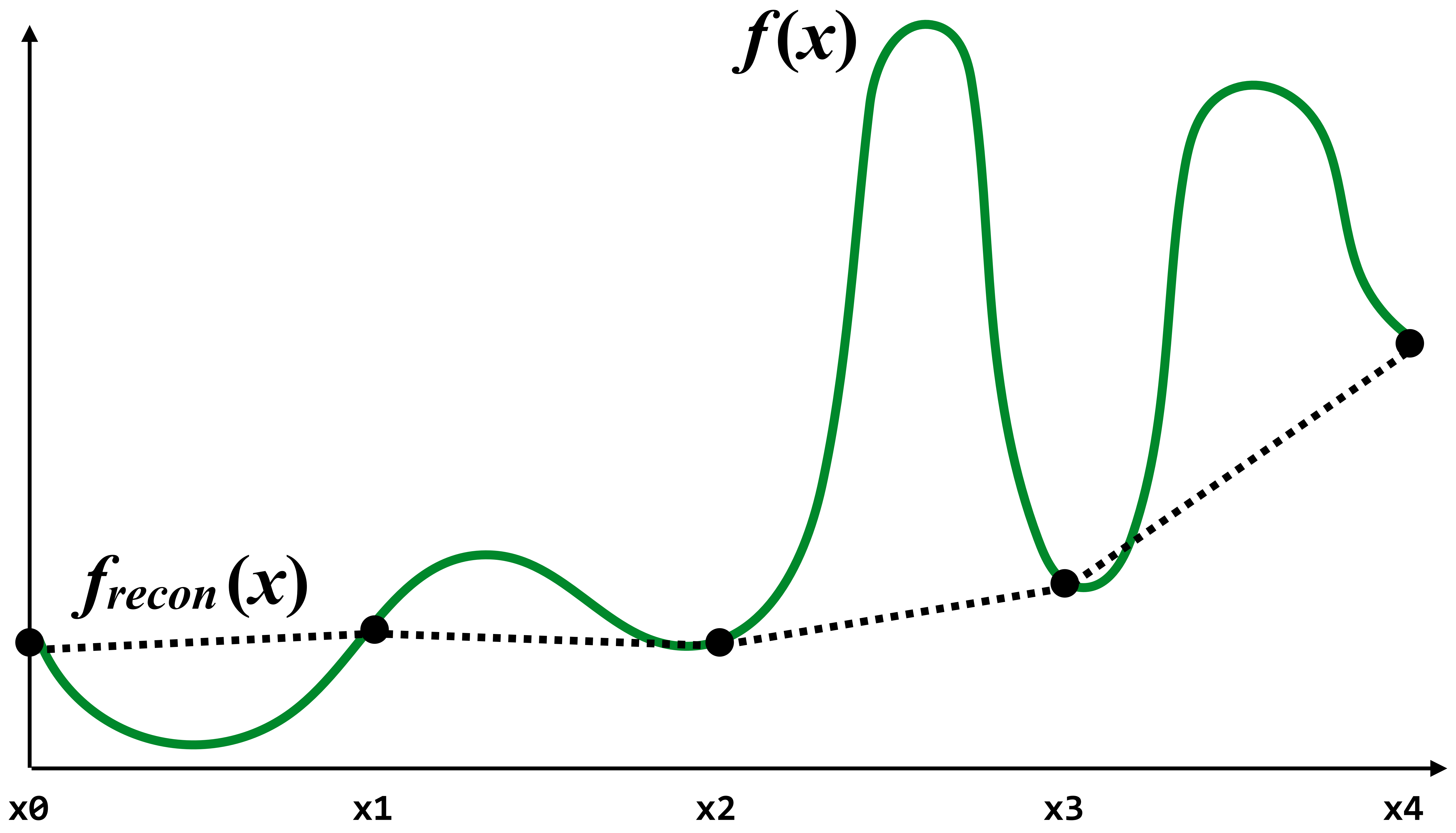
Piecewise constant approximation

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

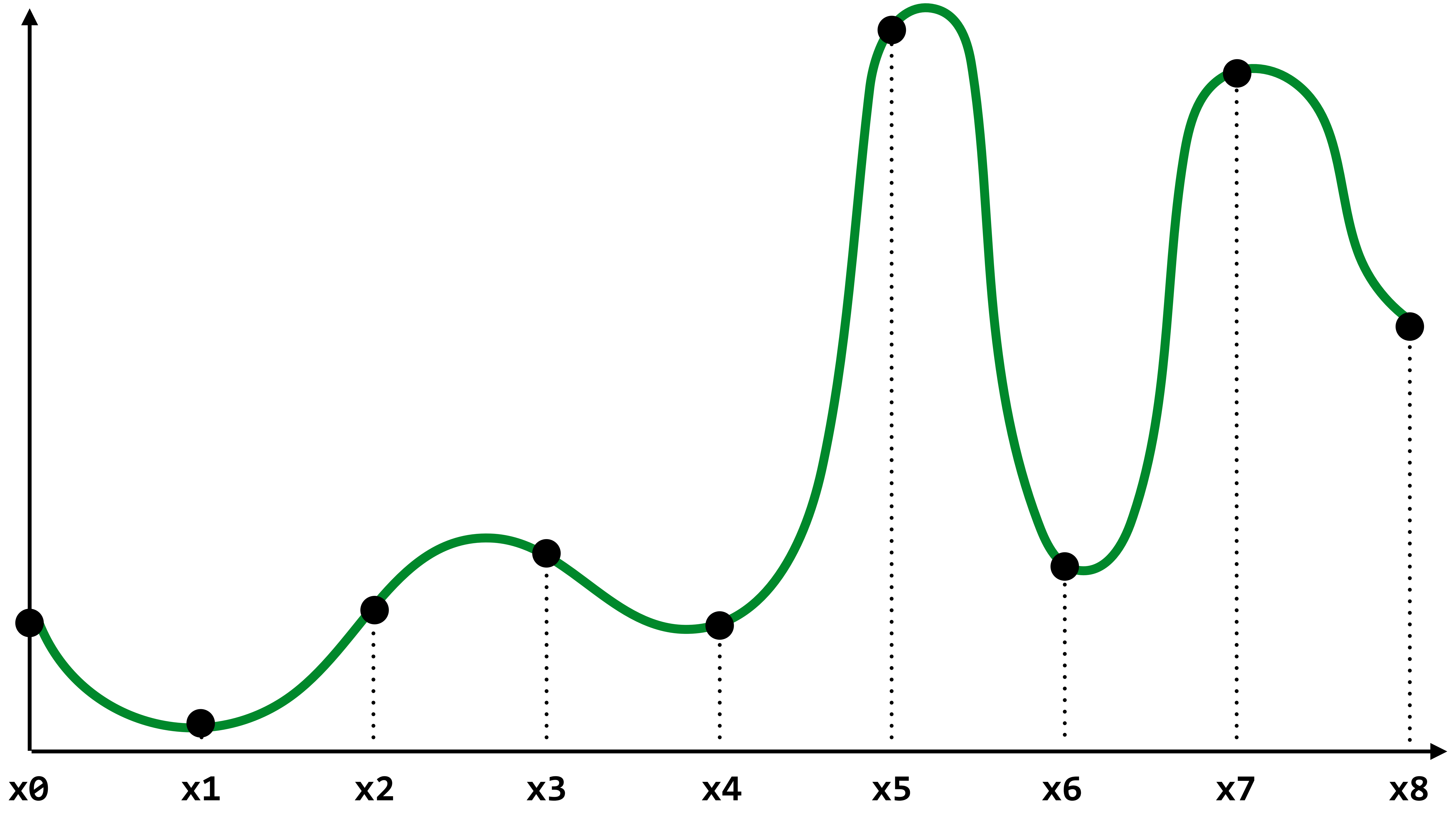


Piecewise linear approximation

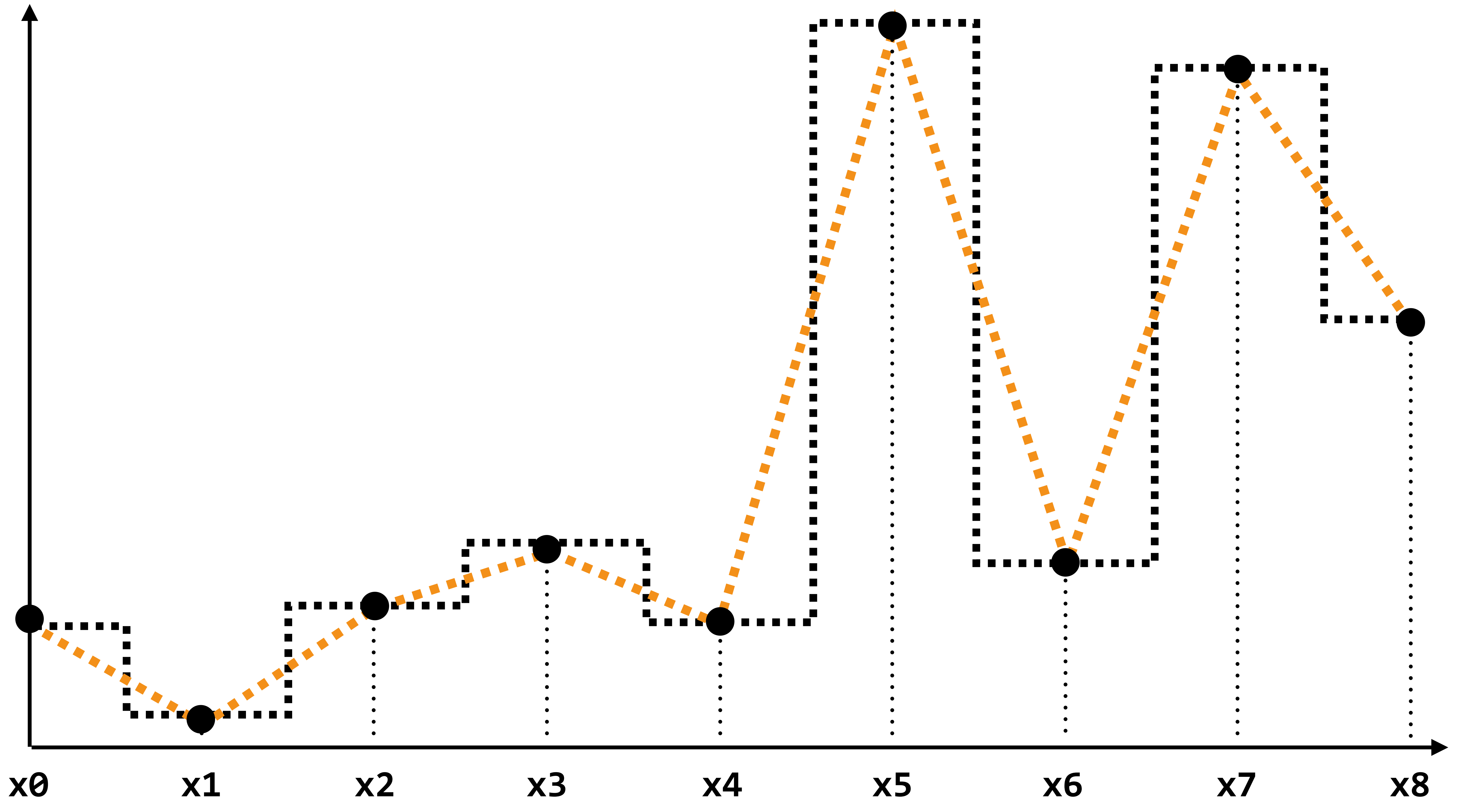
$f_{recon}(x)$ **approximates** $f(x)$



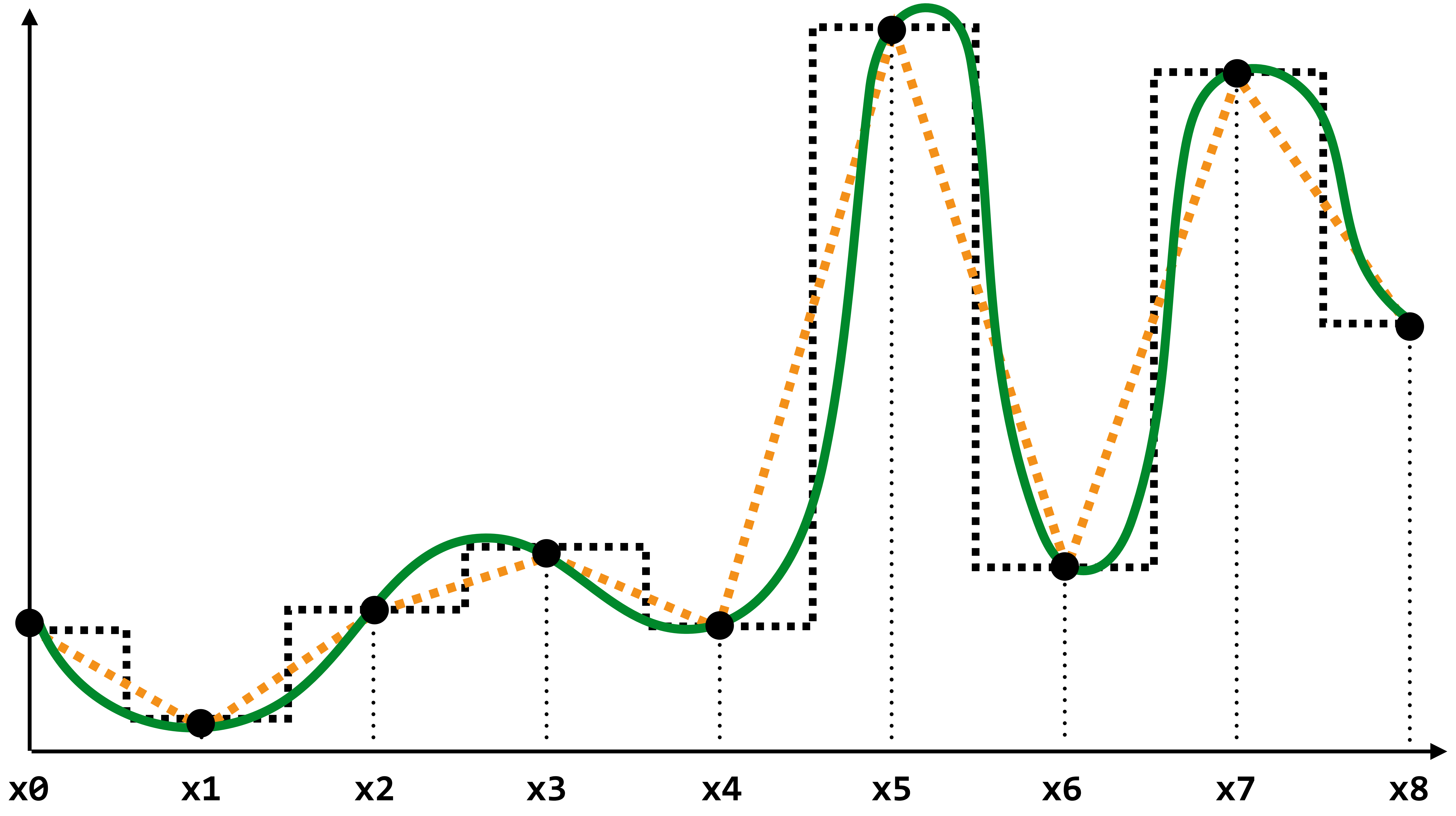
How can we represent the signal more accurately?



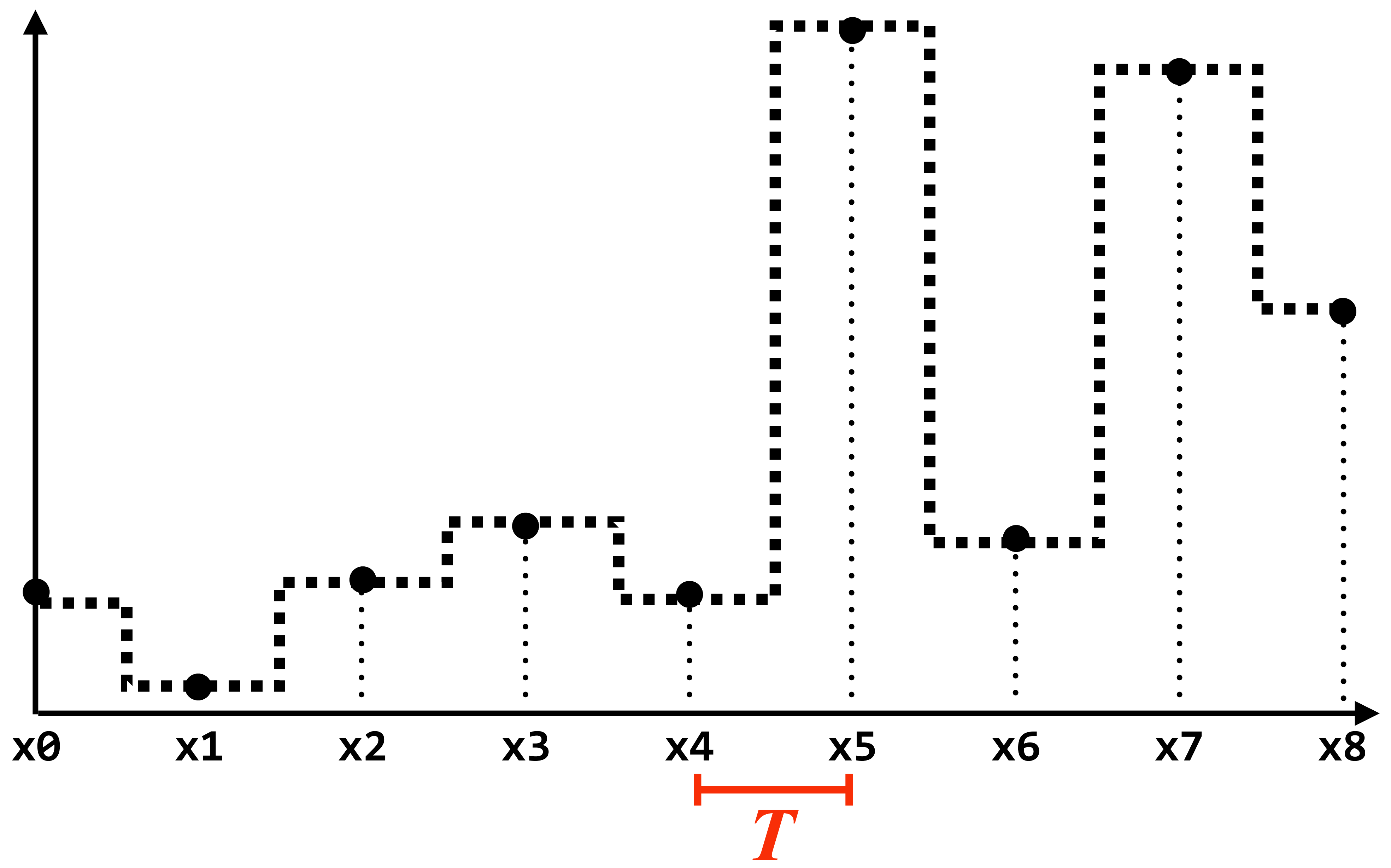
Reconstruction from denser sampling



Reconstruction from denser sampling



Reconstruction as convolution (with box filter)



Sampled signal

$$g(x) = \sum_{i=0}^{8} f(iT)\delta(x - iT)$$

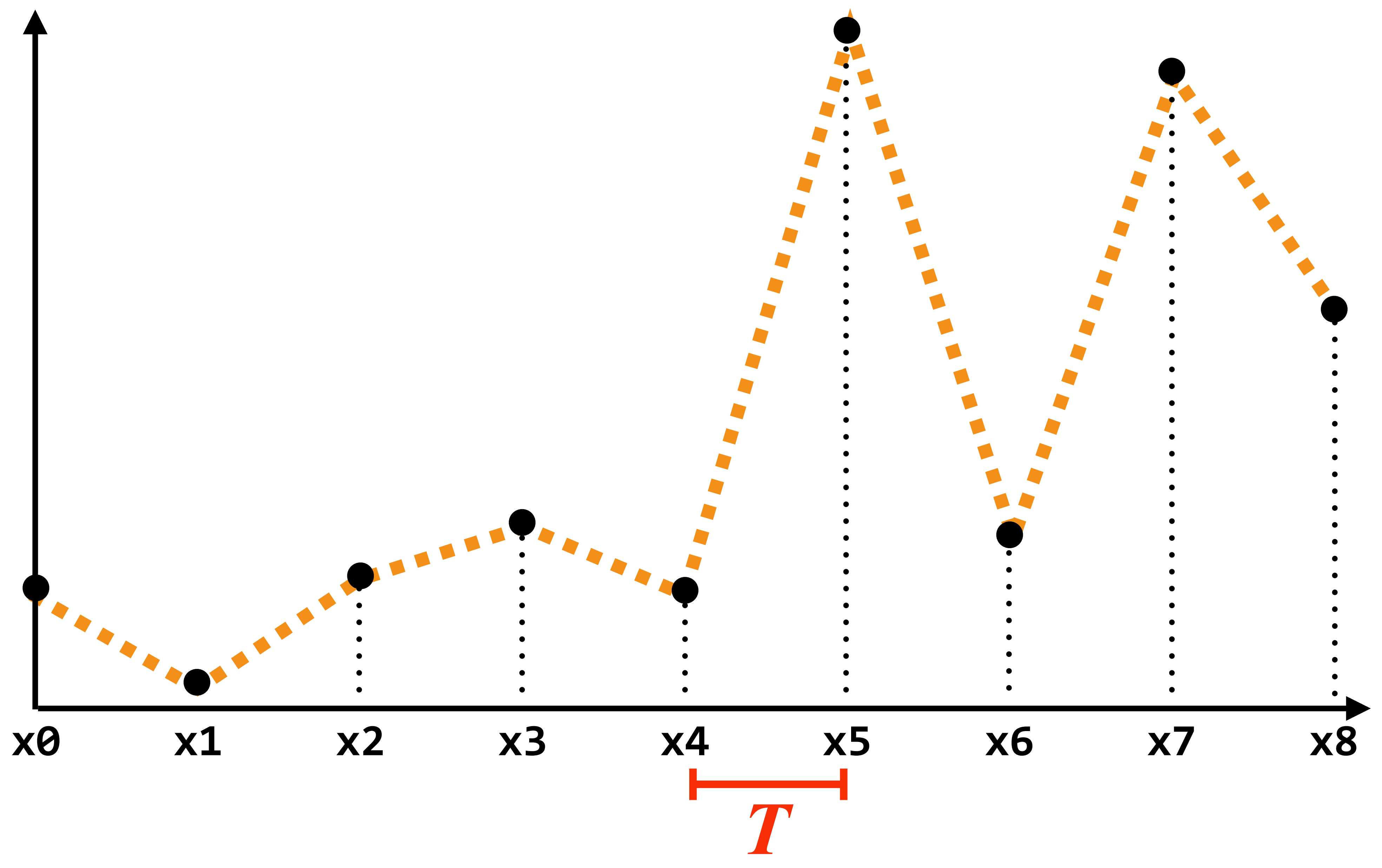
Reconstruction filter (box)

$$h(x) = \begin{cases} 1, & \text{if } |x| \leq T/2 \\ 0, & \text{if } |x| > T/2 \end{cases}$$

Constructed signal
(Chooses nearest sample)

$$f_{recon}(x) = \int_{-\infty}^{\infty} h(x')g(x - x')dx' = \int_{-T/2}^{T/2} \sum_{i=0}^{8} f(iT)\delta(x - x' - iT)dx'$$

Reconstruction as convolution (triangle filter)



Sampled signal

$$g(x) = \sum_{i=0}^{8} f(iT)\delta(x - iT)$$

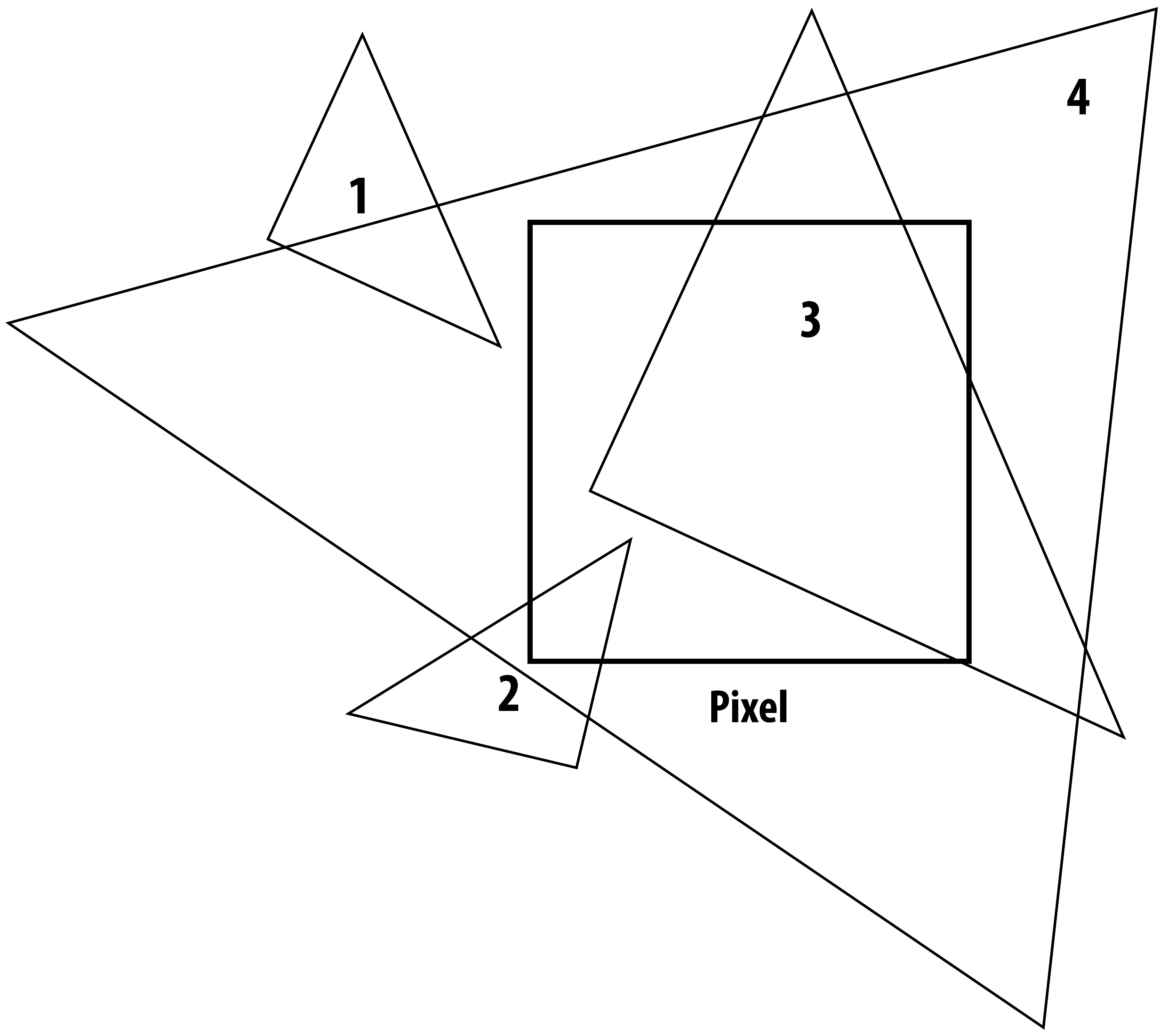
Triangle reconstruction filter: (yields linear interpolation of samples)

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

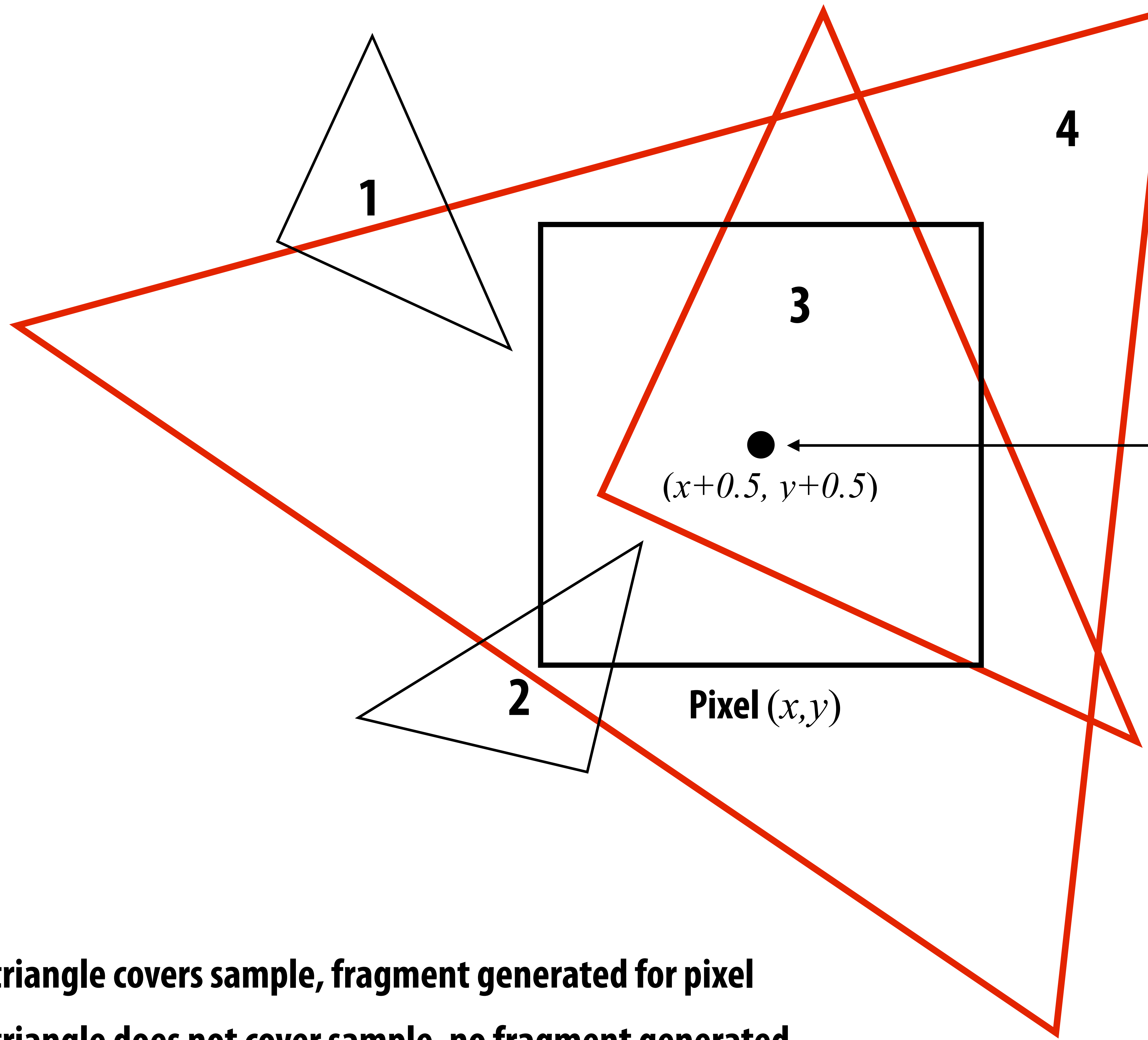
Coverage

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

Question: which triangles "cover" this pixel?



The rasterizer estimates triangle-screen coverage using point sampling
It samples the continuous binary function: `coverage(x,y)`



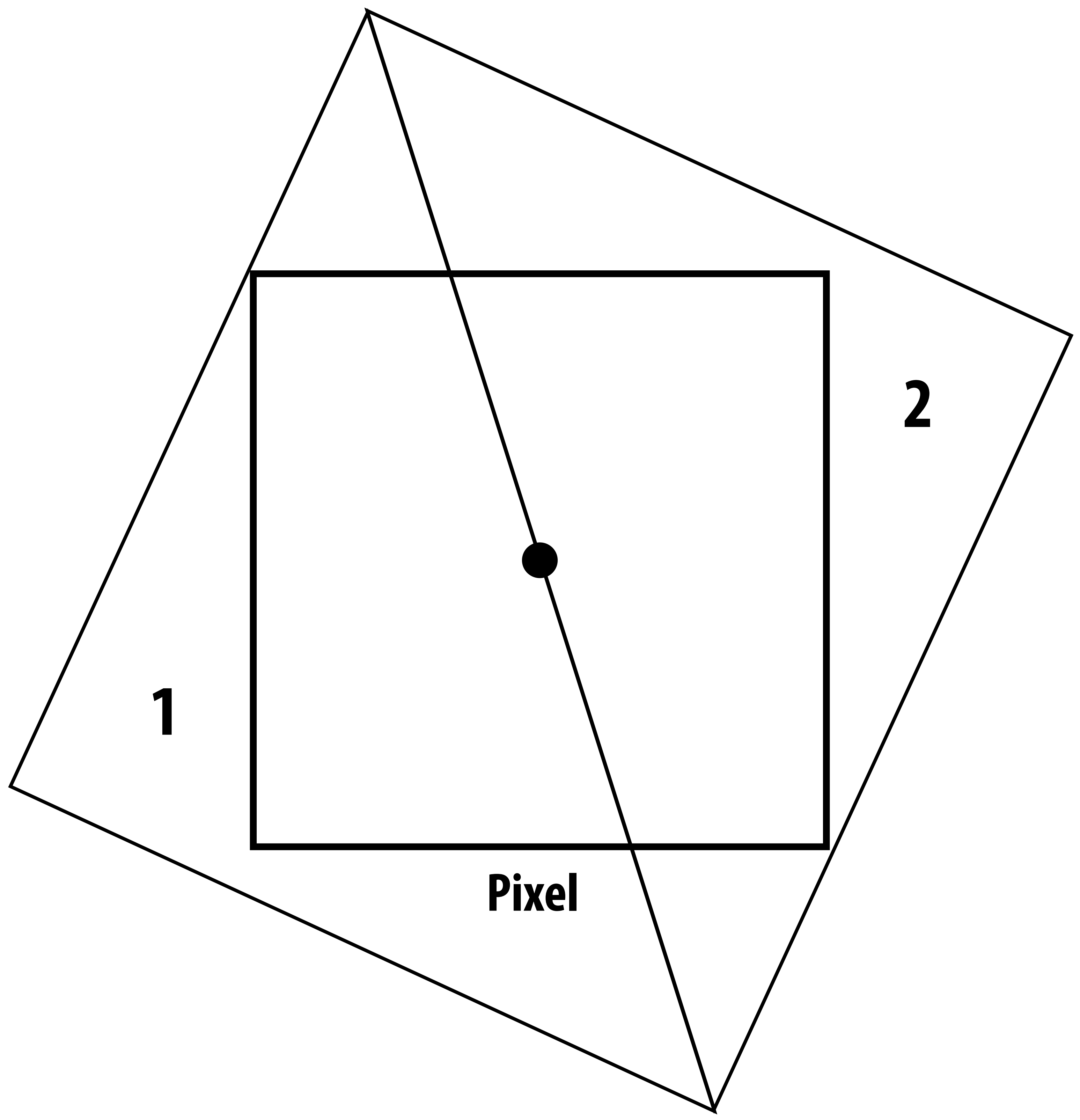
Example:
"coverage sample point" positioned at pixel center

◺ = triangle covers sample, fragment generated for pixel

◺ = triangle does not cover sample, no fragment generated

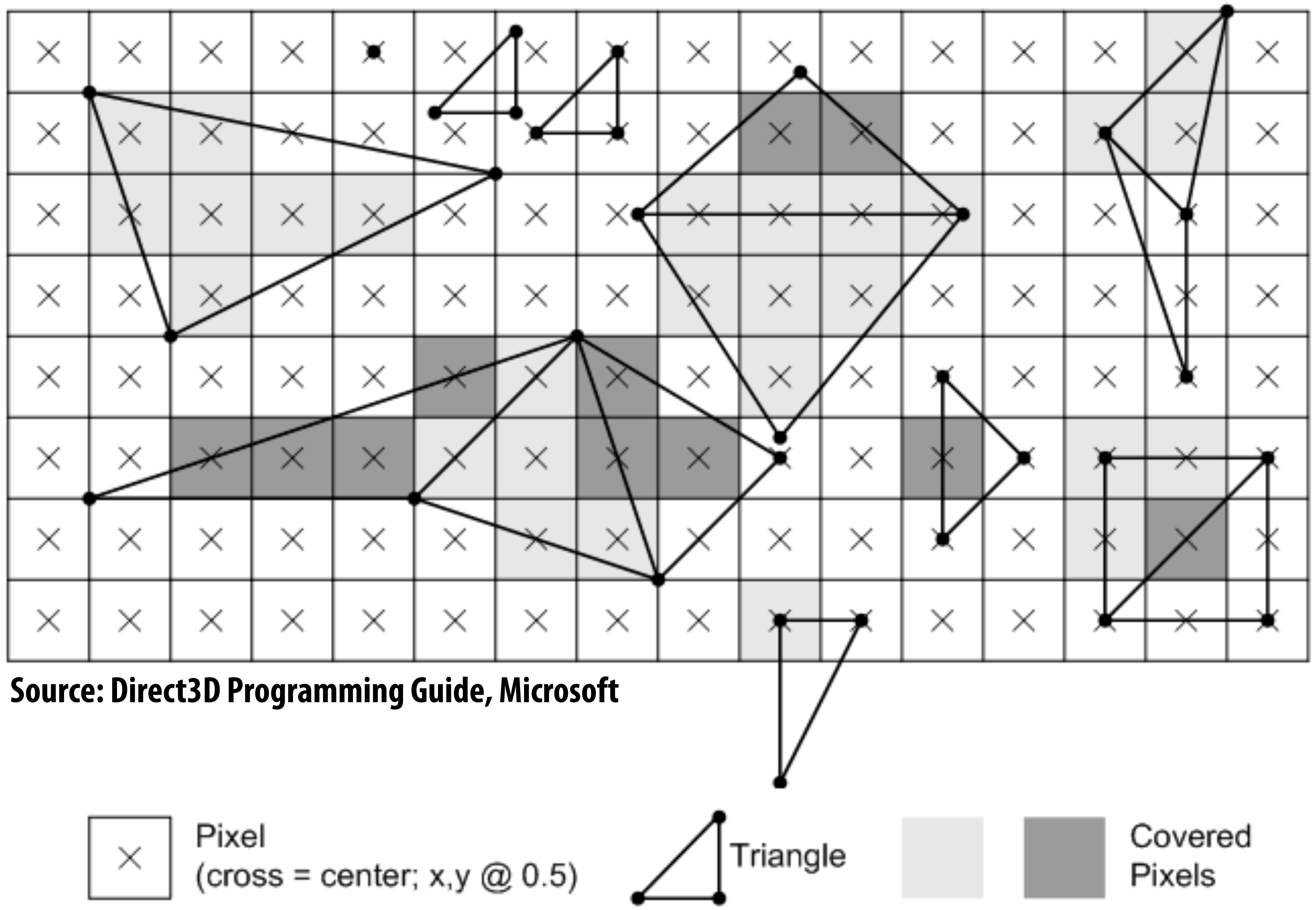
Edge cases (literally)

Is fragment generated for triangle 1? for triangle 2?



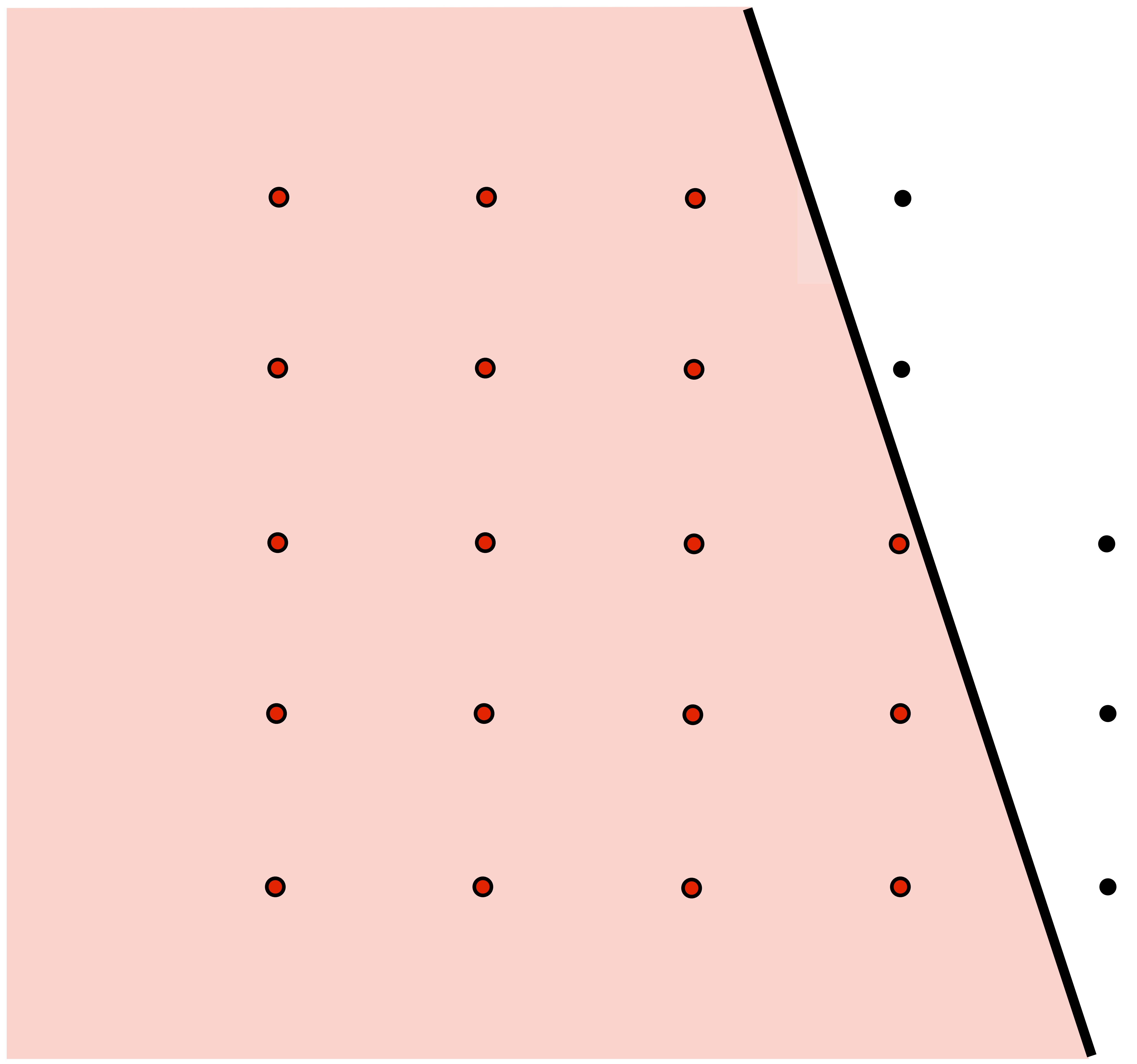
Edge rules

- OpenGL/Direct3D specification: 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)

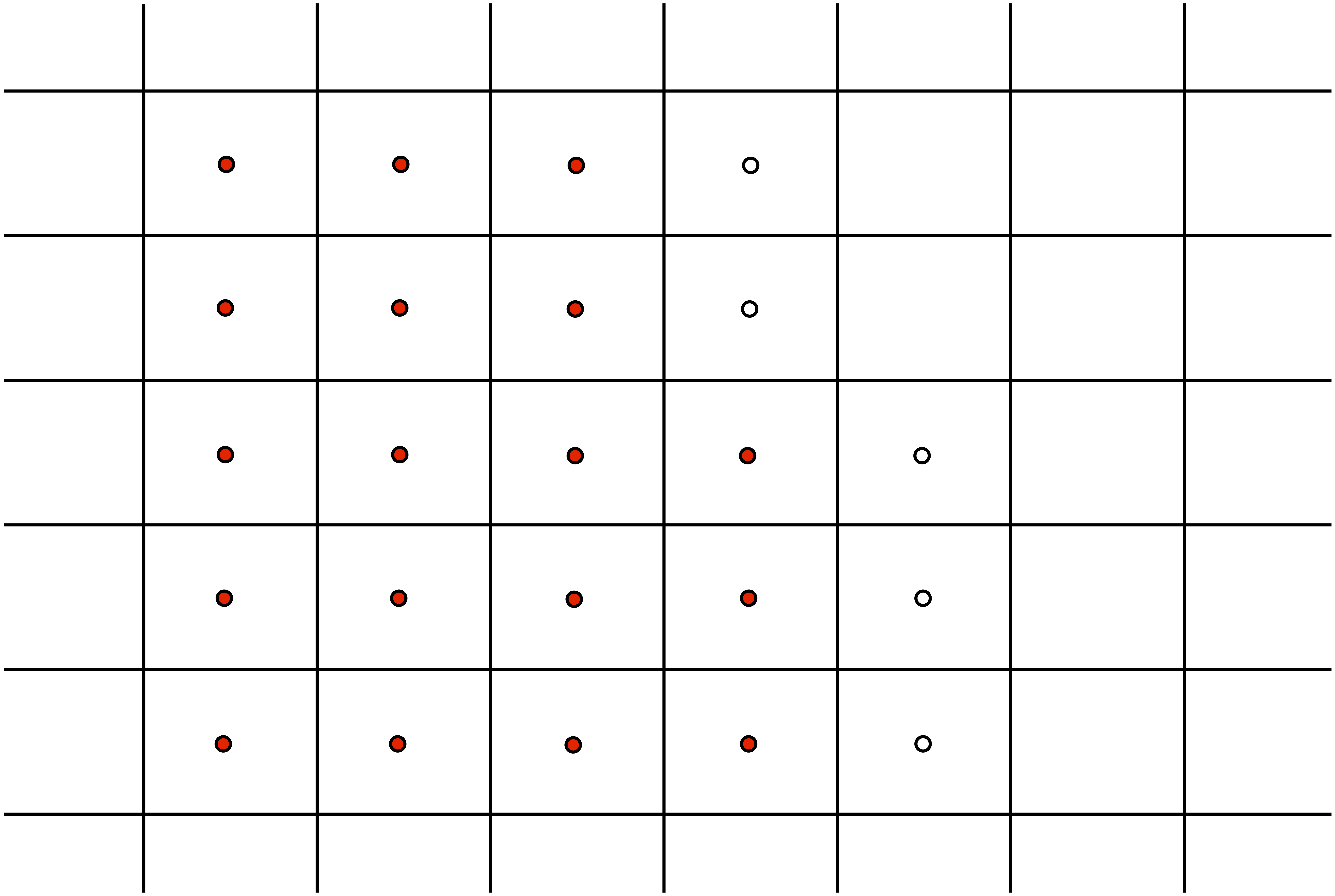


Source: Direct3D Programming Guide, Microsoft

Results of sampling triangle coverage



Results of sampling (red dots = covered)



Reconstruction with box filter (aliased edges)

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

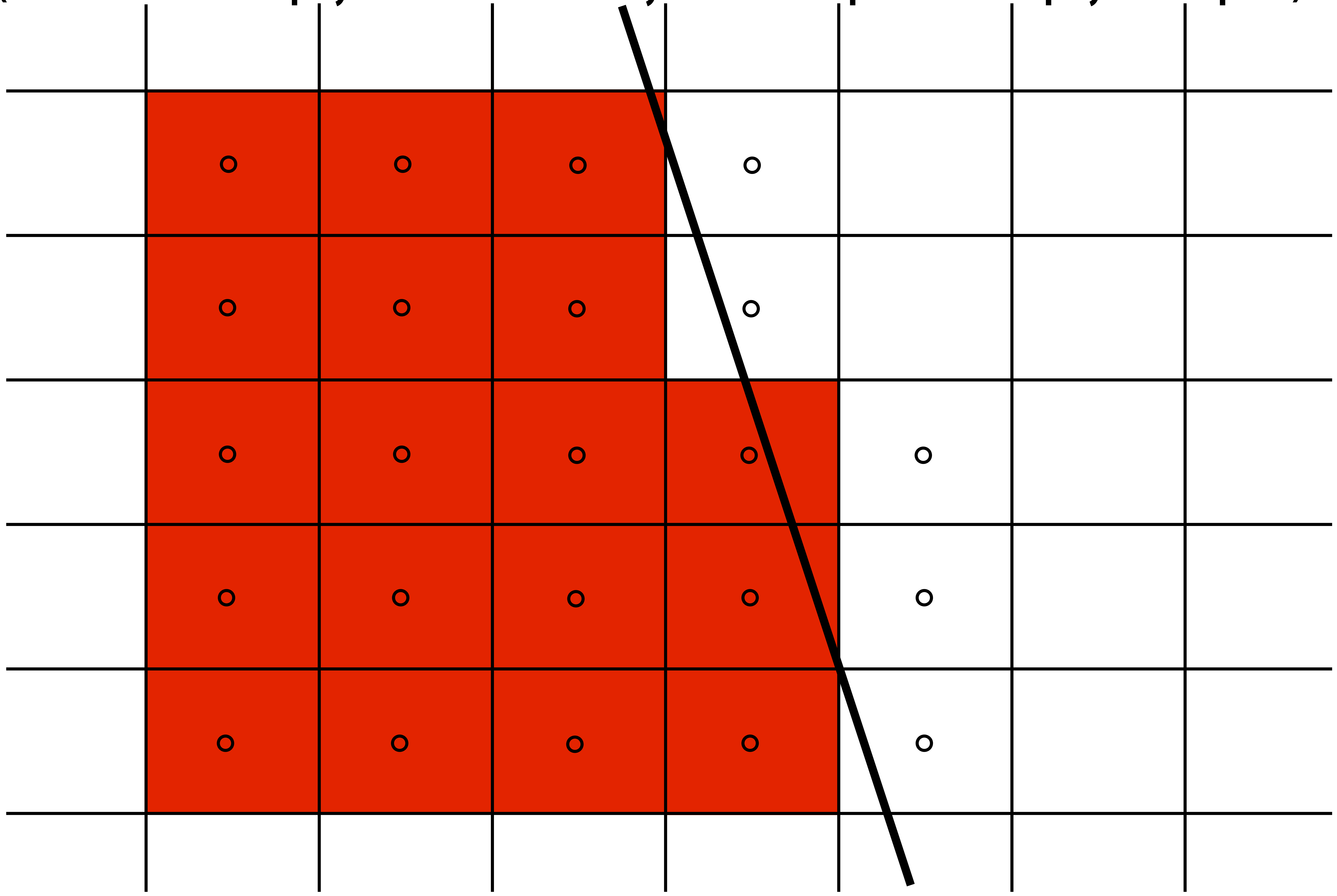
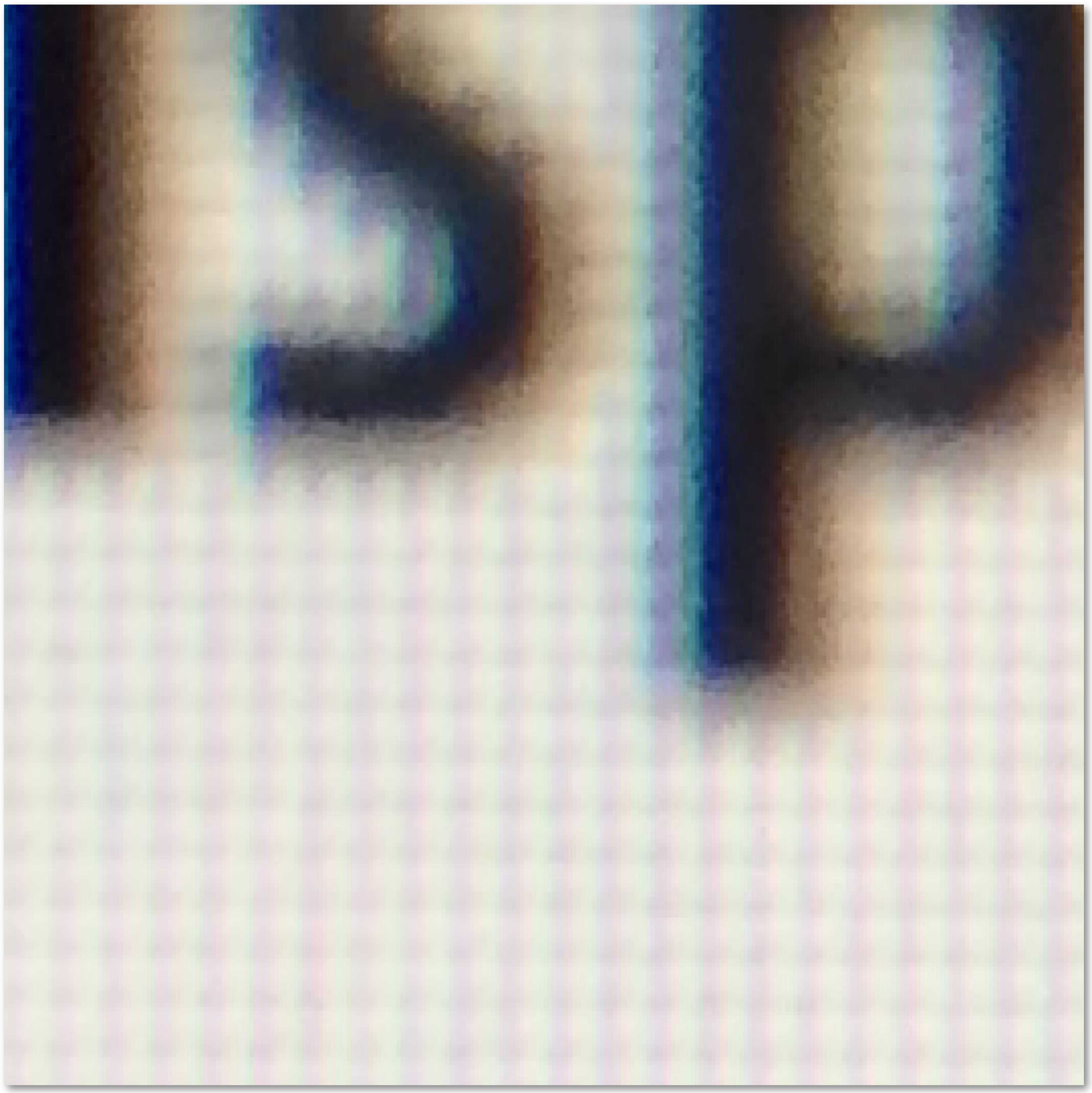


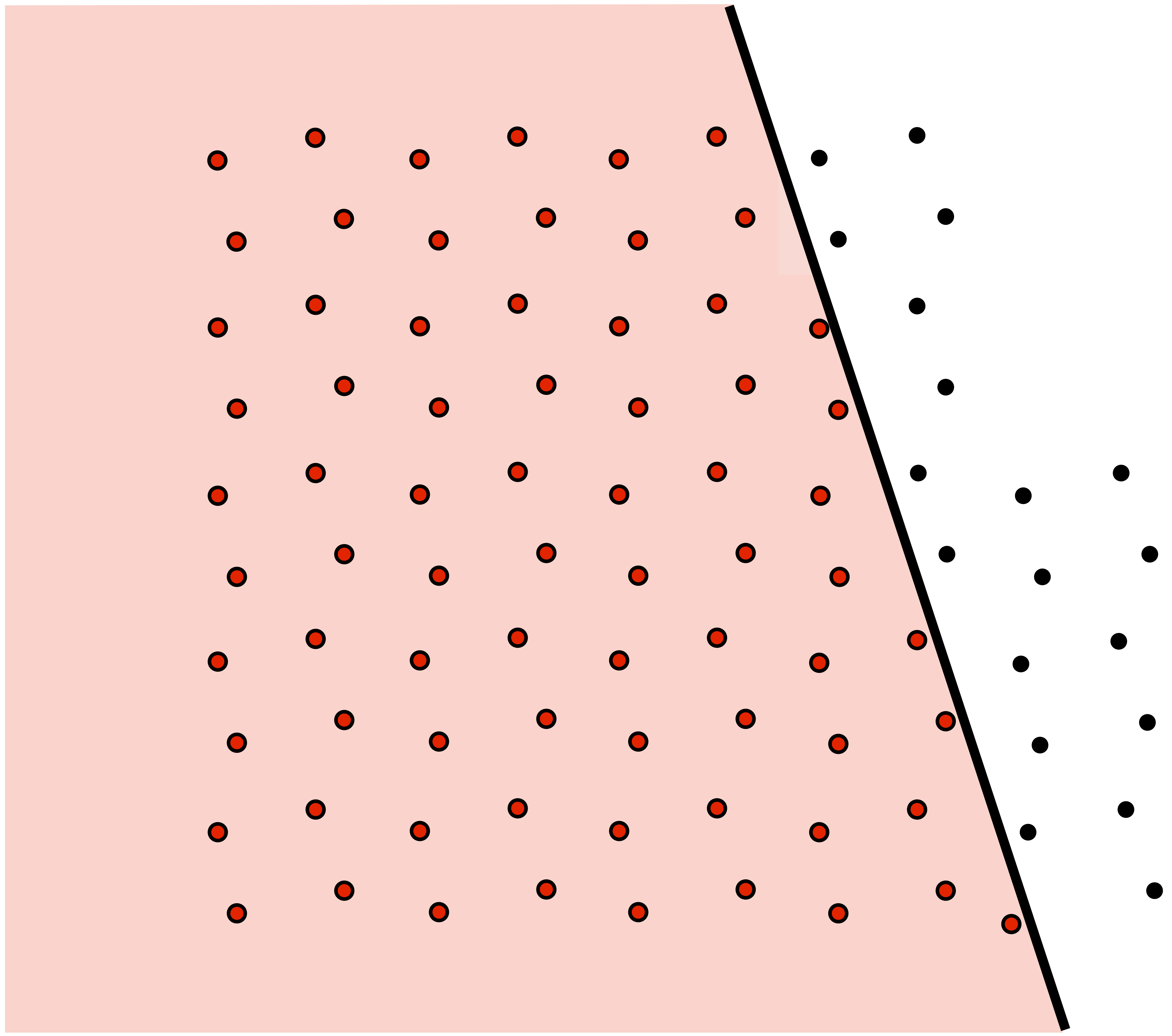
Photo of previous slide displayed on monitor



Problem: aliasing

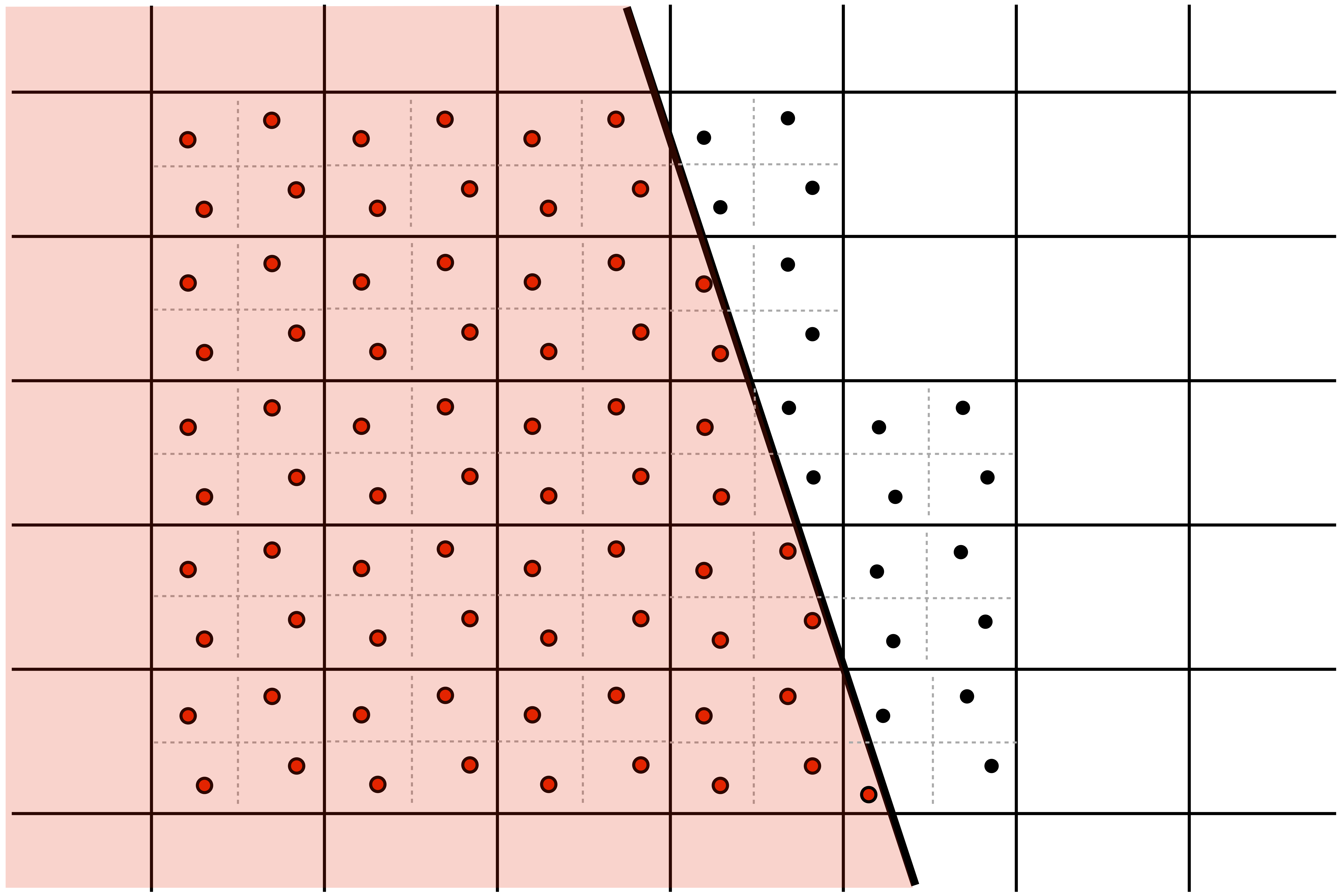
- Undersampling high frequency signal results in aliasing
  - "Jaggies" in a single image
  - "Roping" or "shimmering" in an animation

Supersampling: increase rate of sampling to more accurately reconstruct high frequencies in triangle coverage signal (high frequencies exist because of triangle edges)



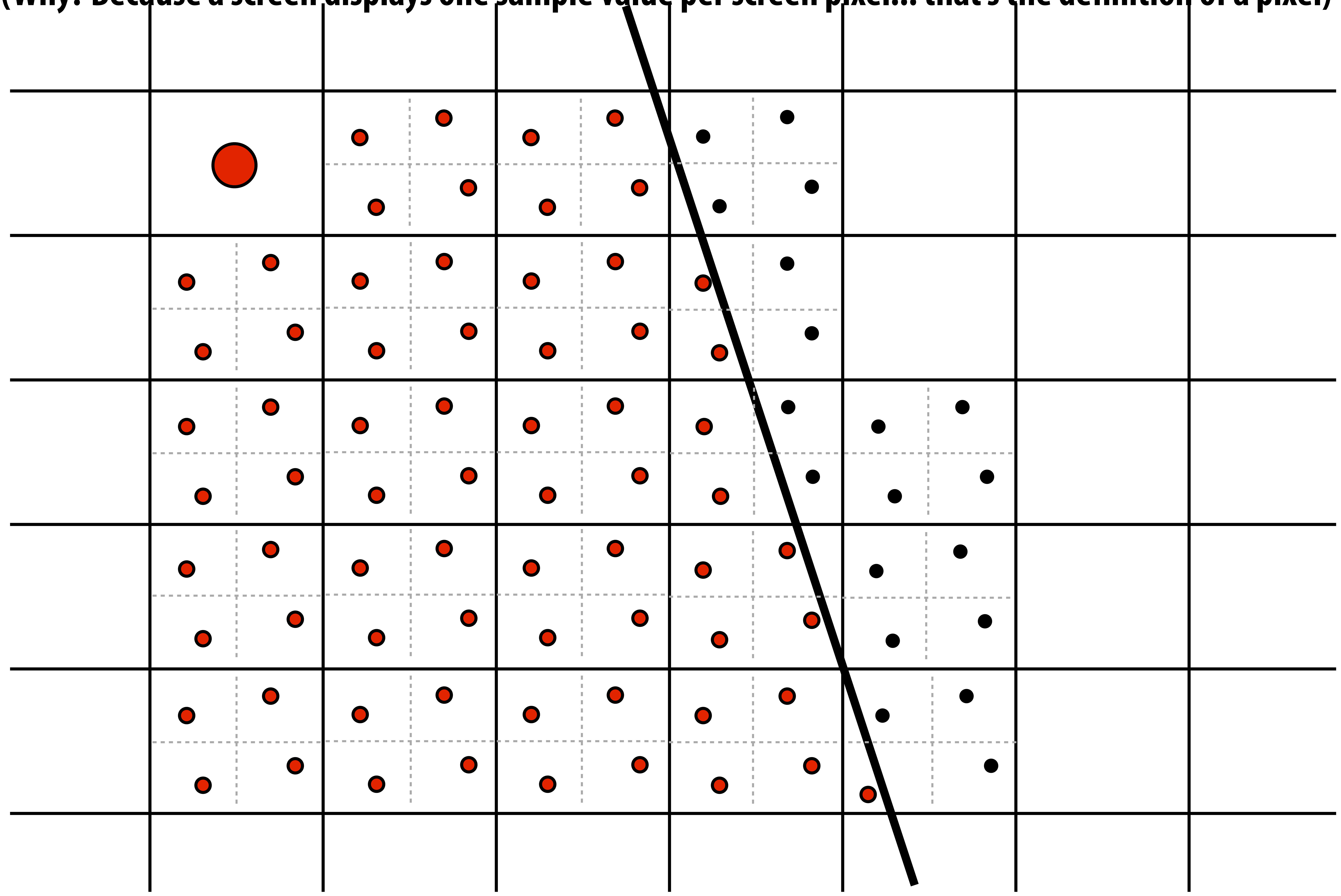
Supersampling

Example: stratified sampling using four samples per pixel

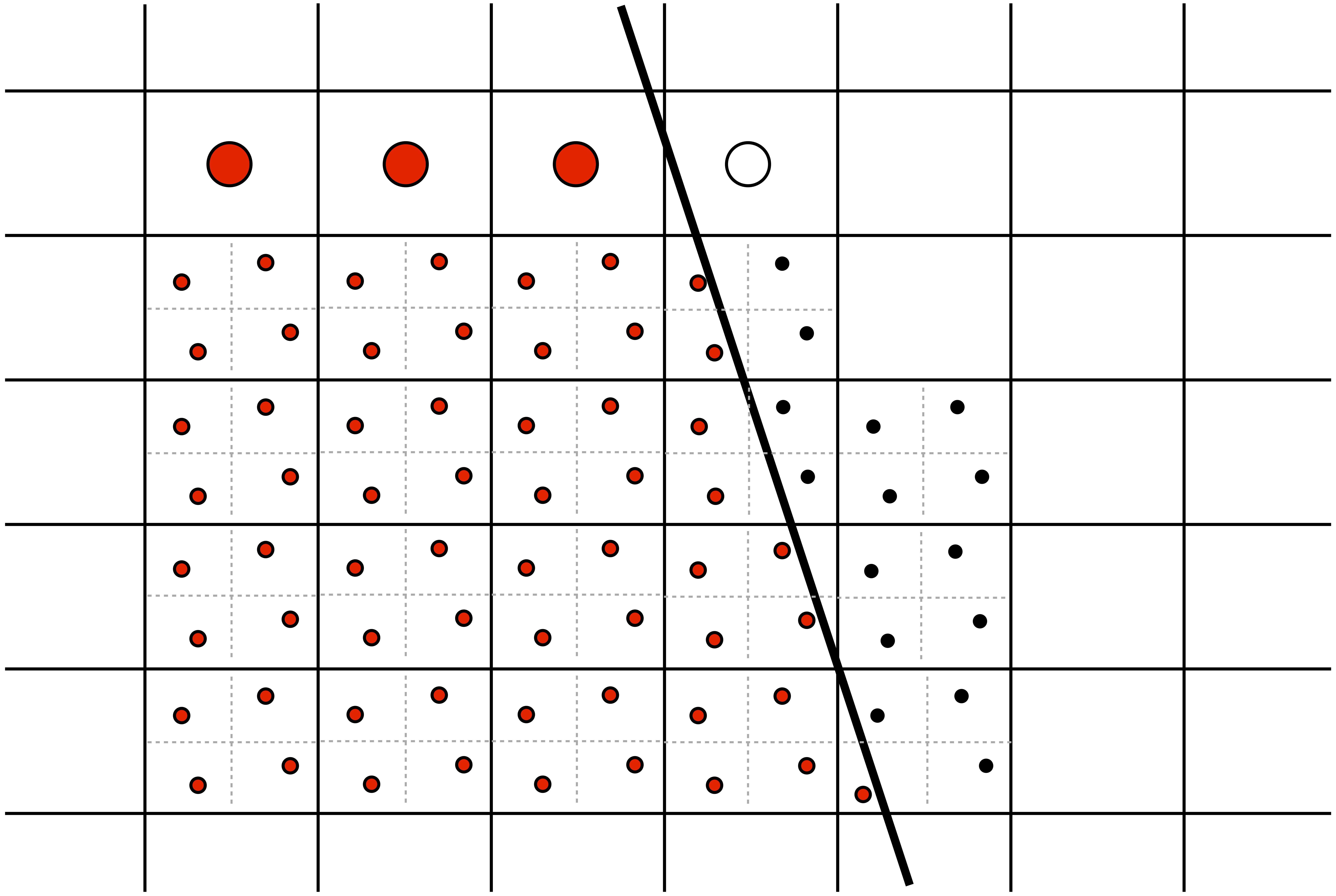


Resample to display's pixel rate (using box filter)

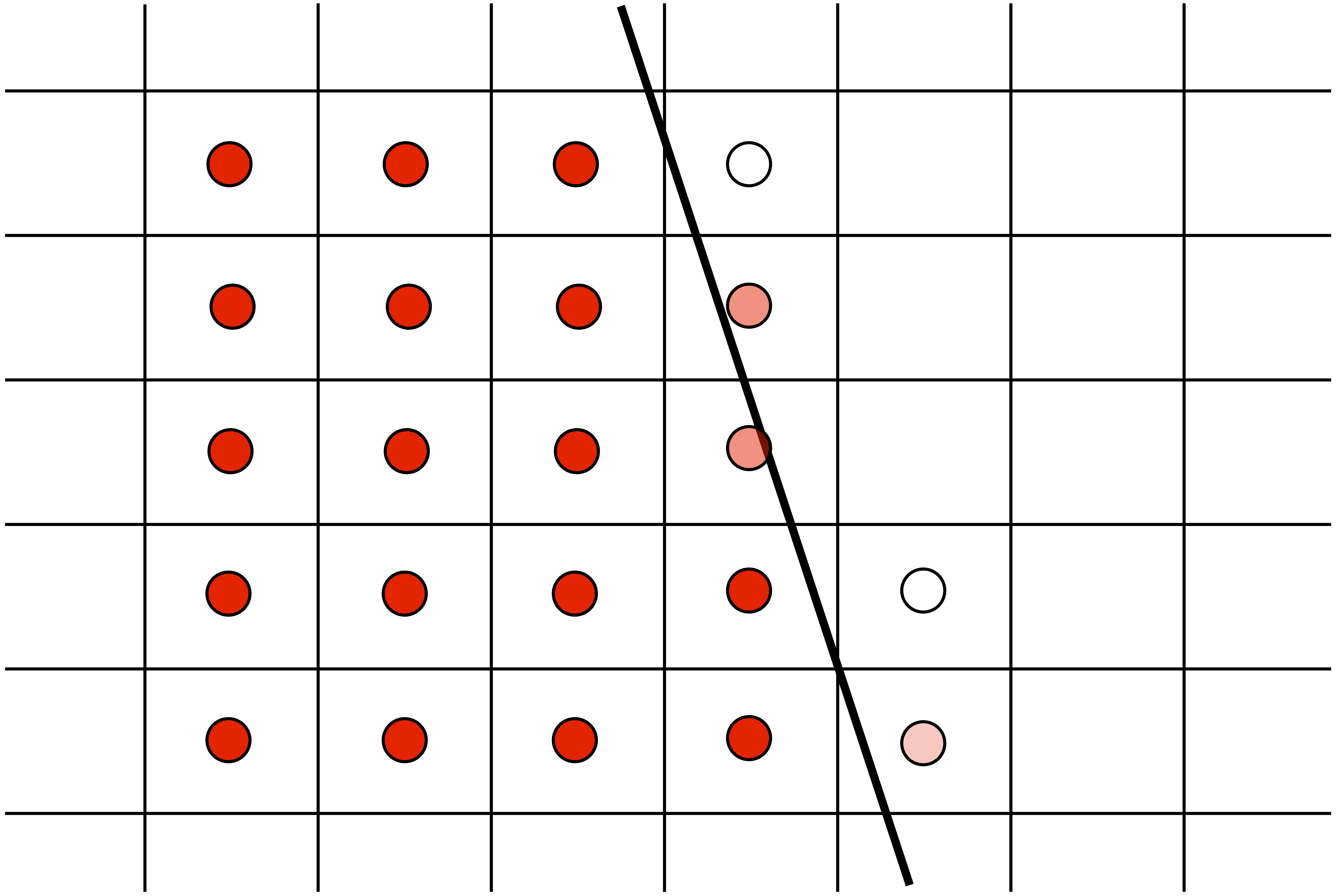
(Why? Because a screen displays one sample value per screen pixel... that's the definition of a pixel)



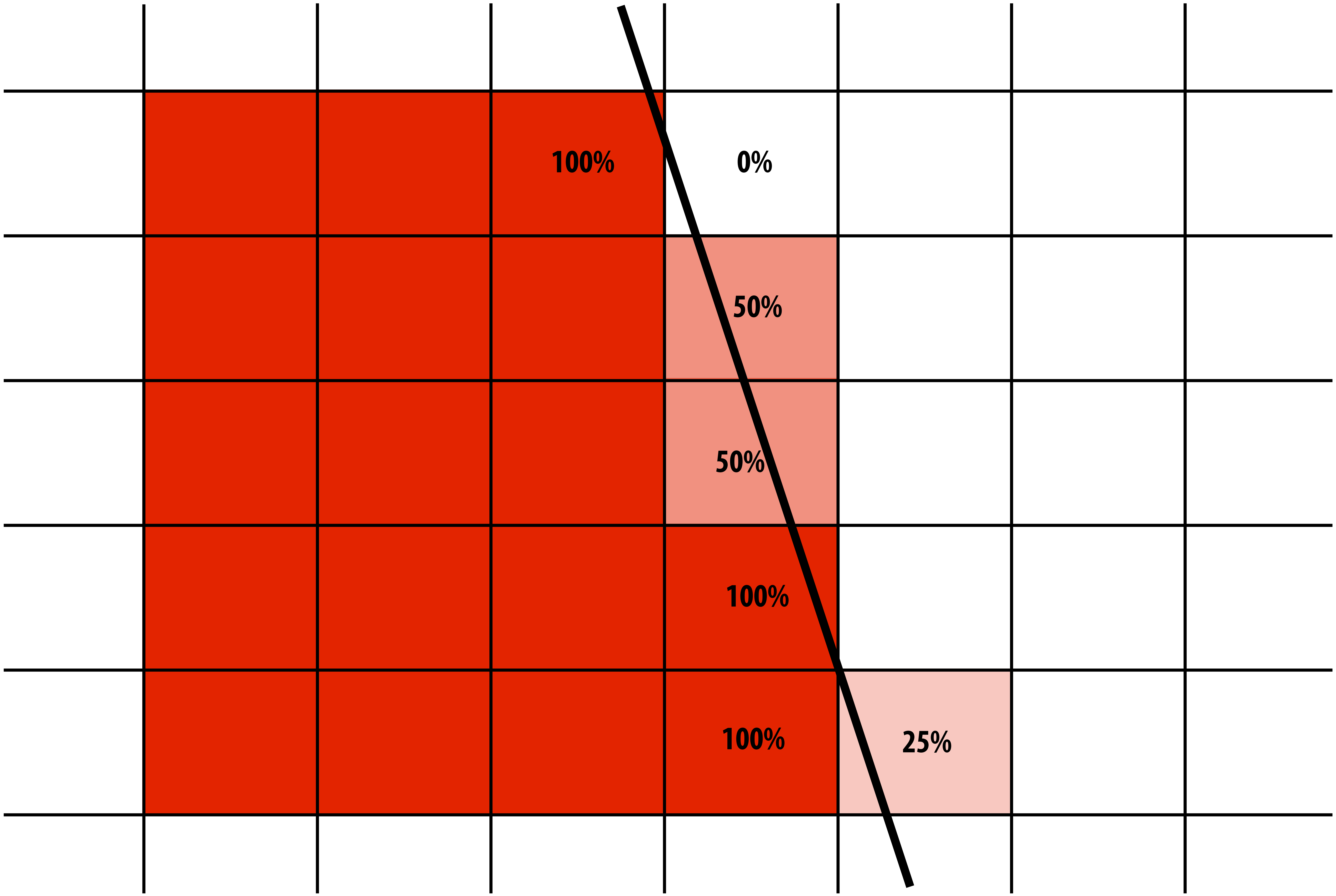
Resample to display's pixel rate (using box filter)



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



Displayed result (note anti-aliased edges)



Sampling coverage

- What we really want is for 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. (We want to reconstruct `coverage(x,y)`)

- 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)

Modes of fragment generation

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

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

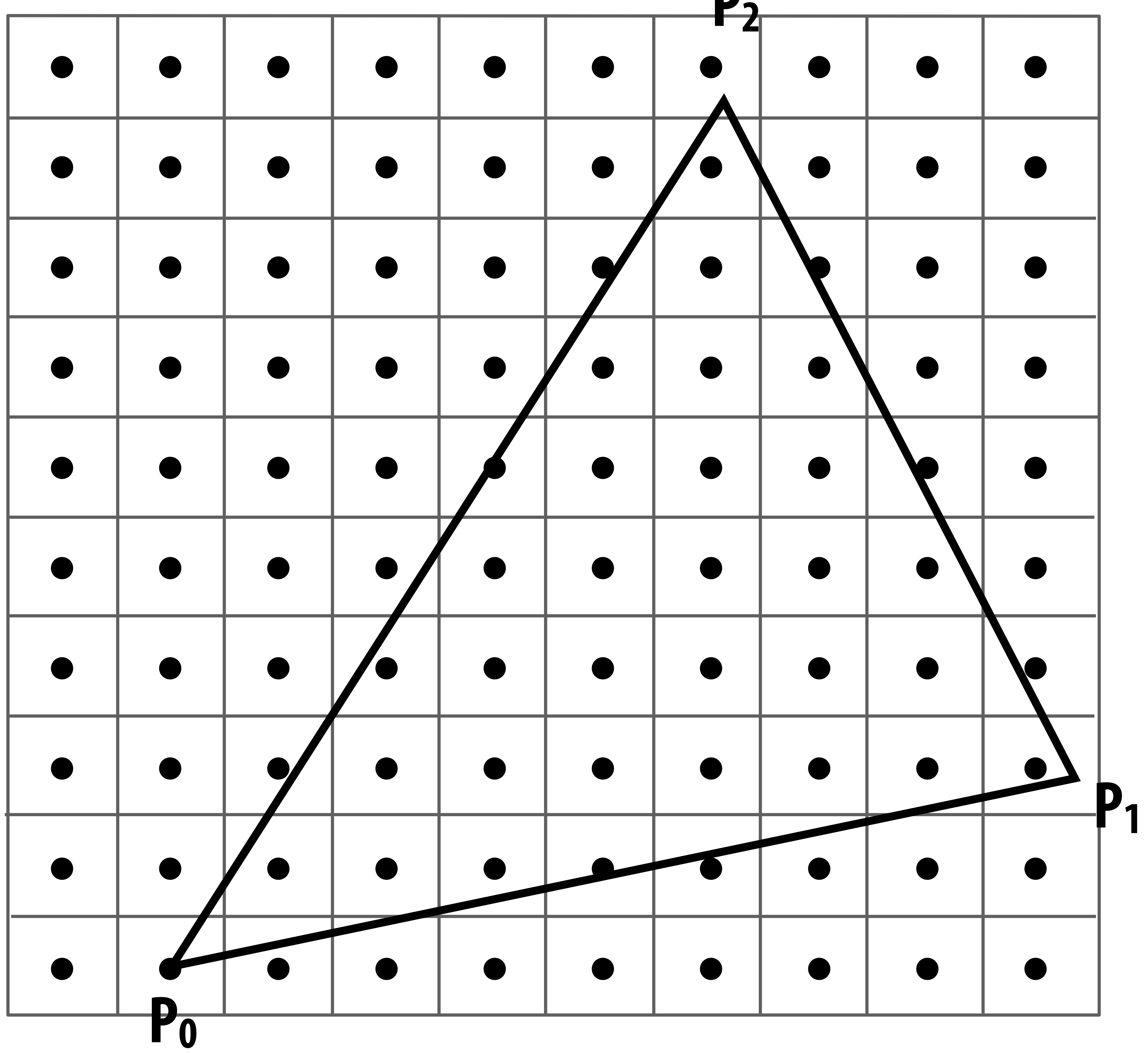
Point-in-triangle test

$P_i = (X_i, Y_i, Z_i)$

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

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

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



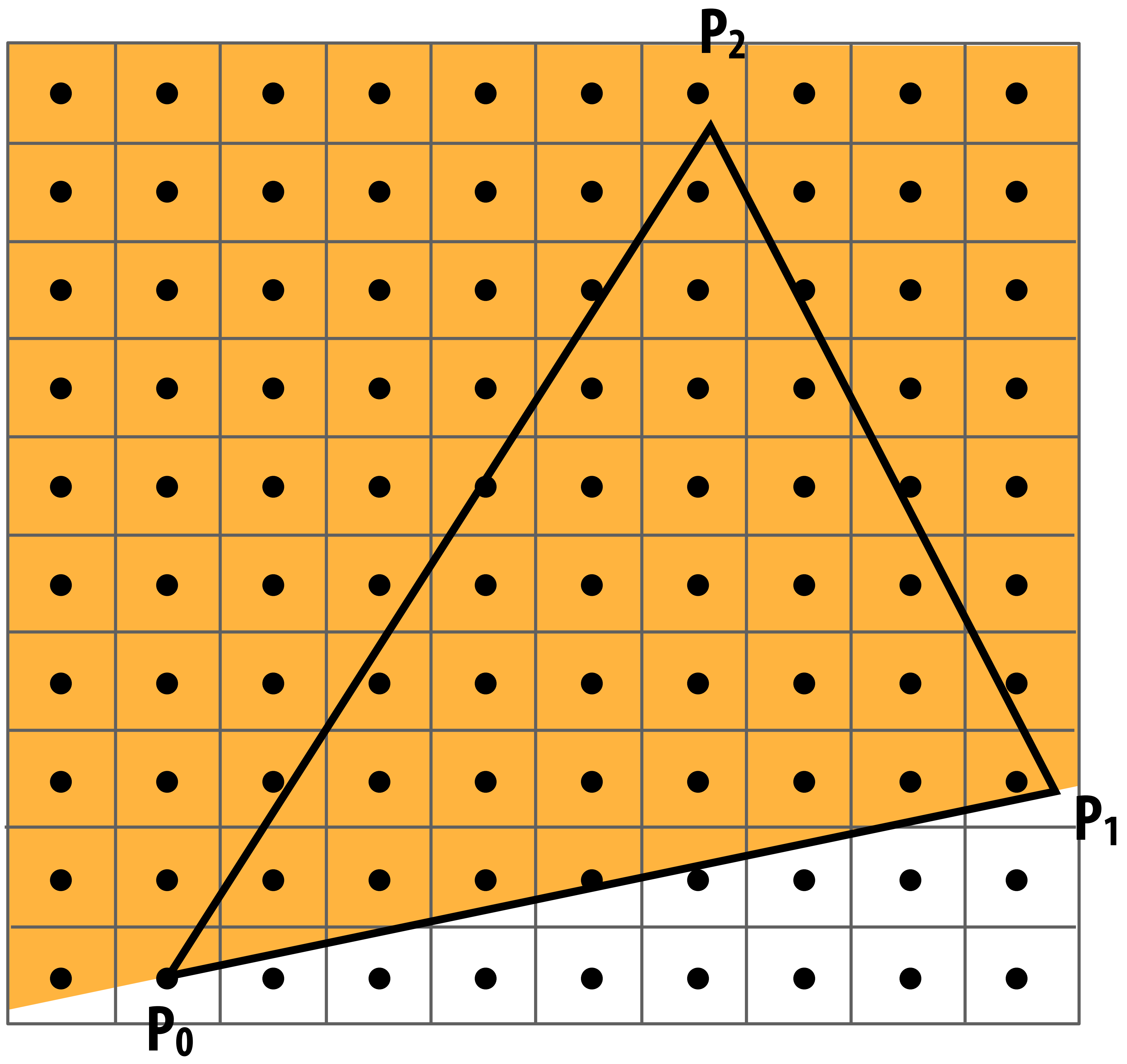
Point-in-triangle test

$P_i = = (X_i, Y_i, Z_i)$

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

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

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



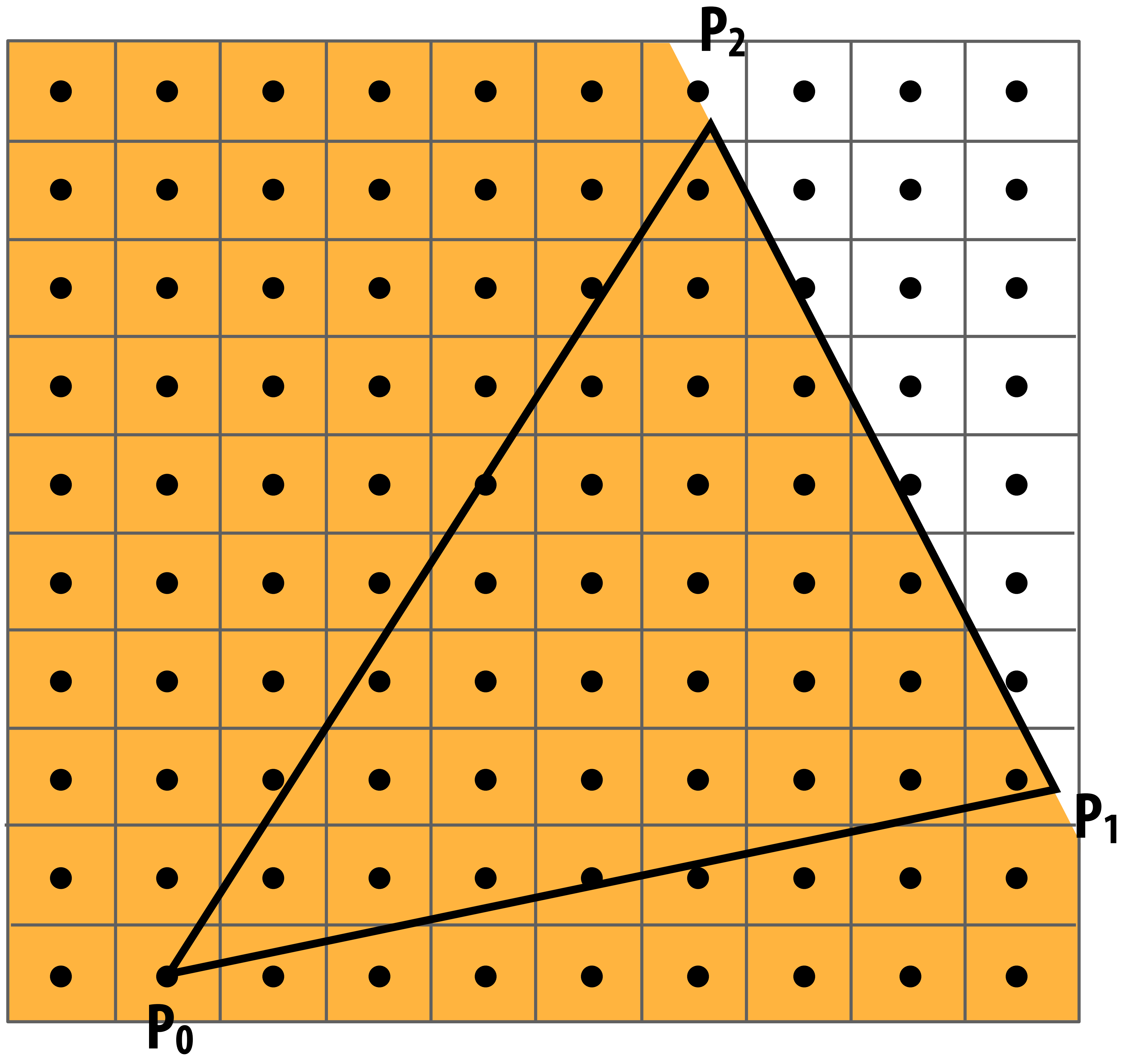
Point-in-triangle test

$P_i = (X_i, Y_i, Z_i)$

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

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

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



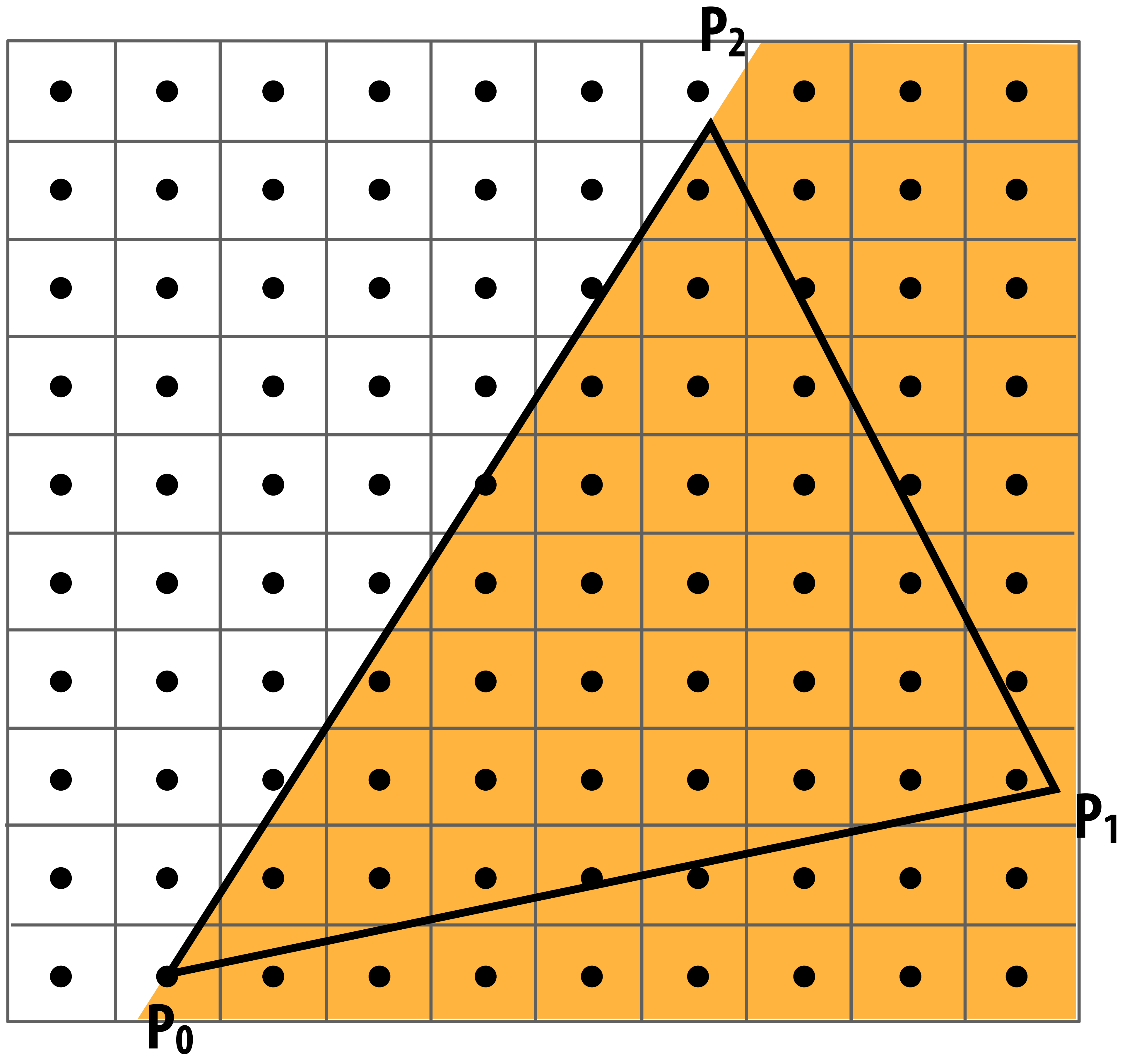
Point-in-triangle test

$P_i = (X_i, Y_i, Z_i)$

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

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

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

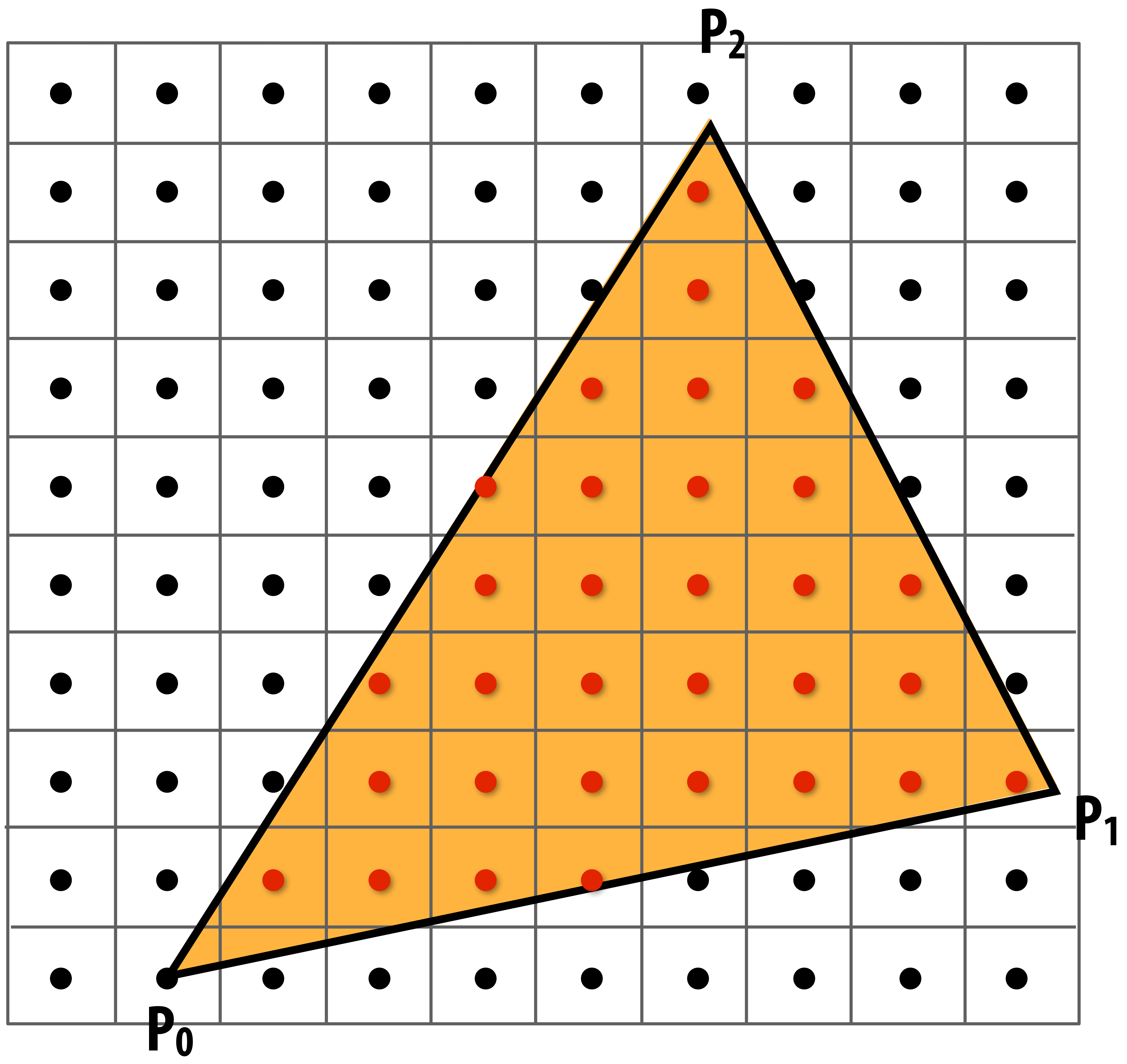


Point-in-triangle test

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 pipeline rasterizer's edge rules.**



Sample points inside triangle are highlighted red.

Incremental triangle traversal

$P_i = (X_i, Y_i, Z_i)$

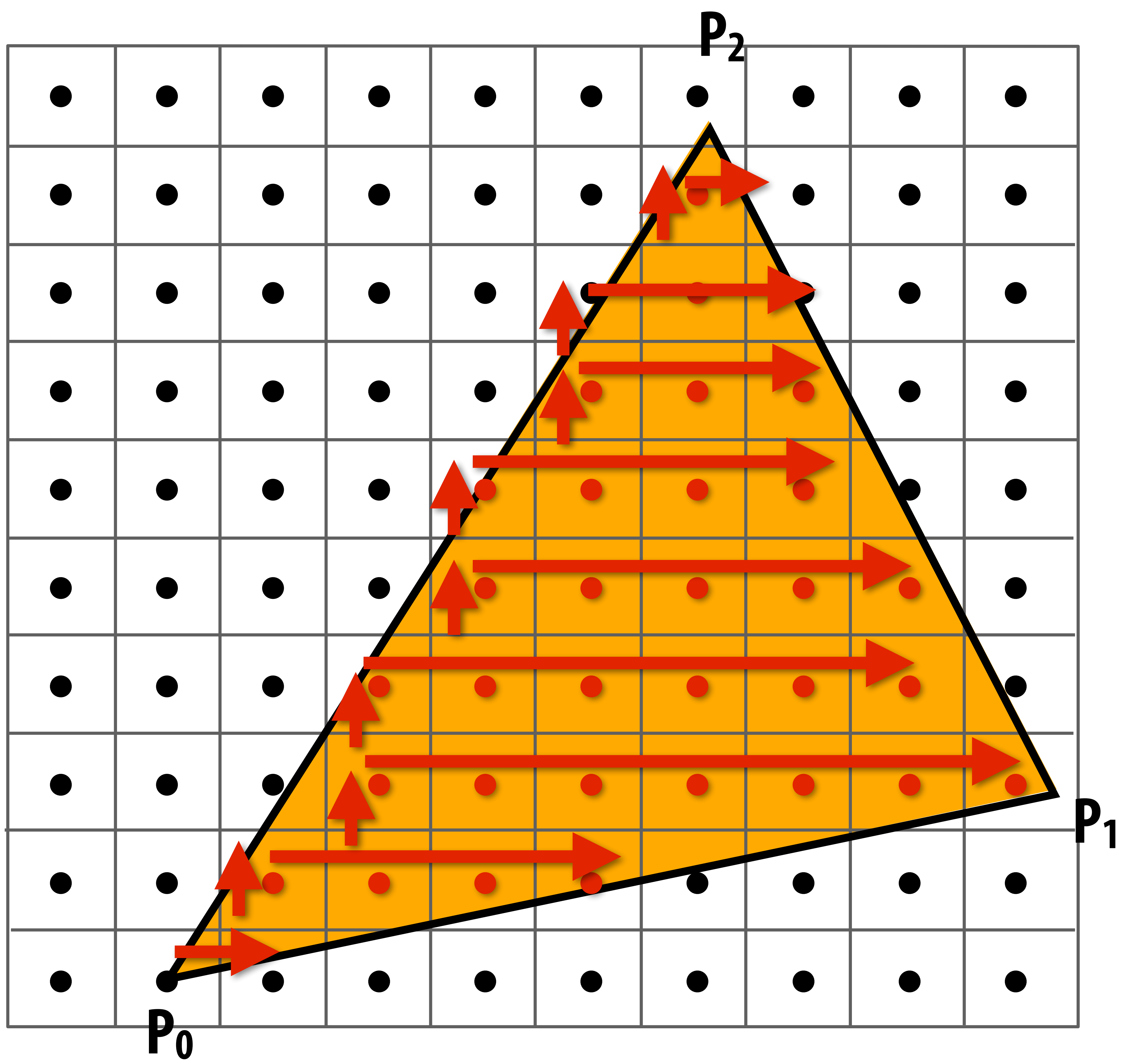
$dX_i = X_{i+1} - X_i$
$dY_i = Y_{i+1} - Y_i$

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

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

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

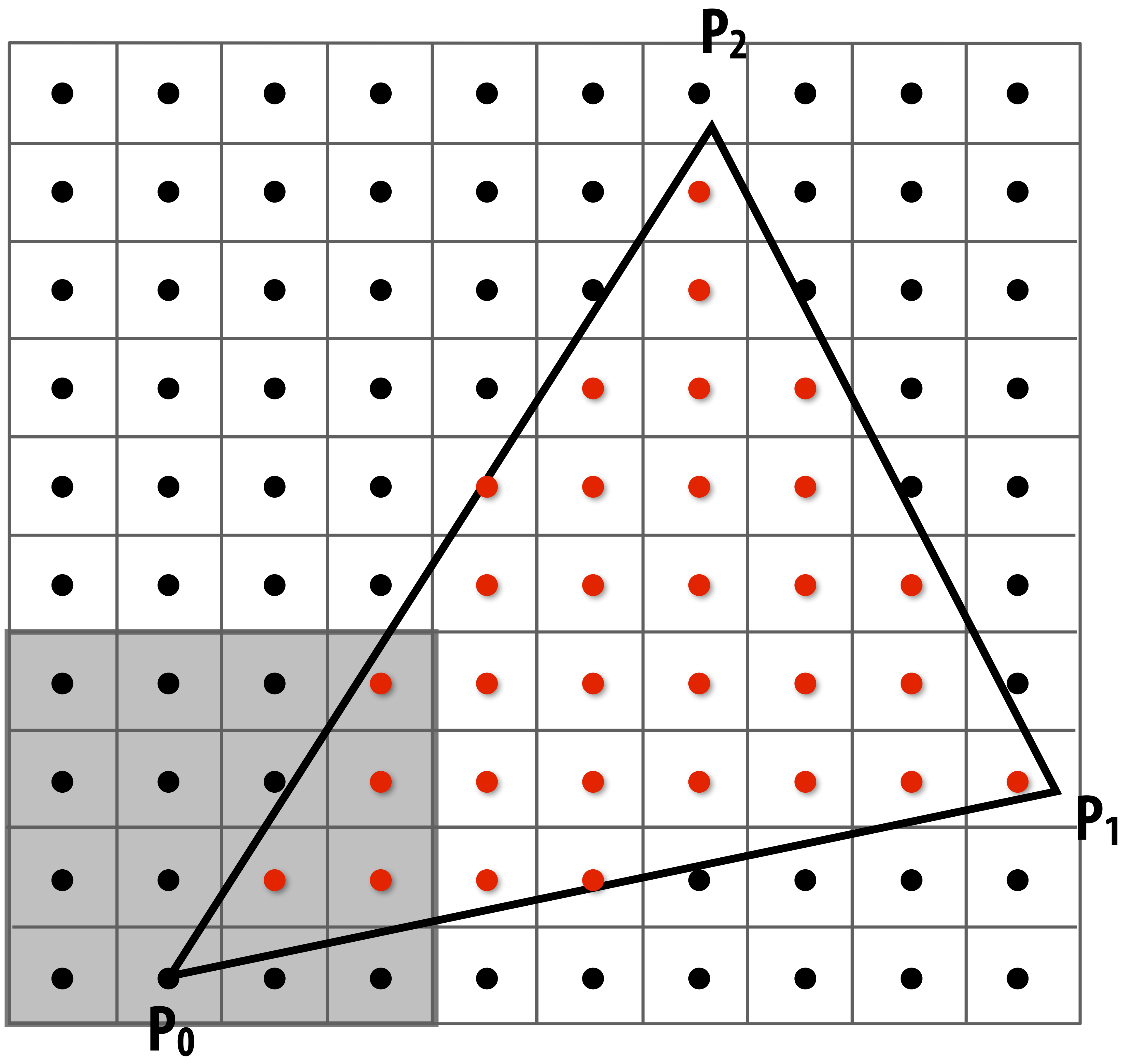
Traverse triangle in blocks

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 in")
- Entire tile may be discarded due to occlusion (see occlusion cull later in this lecture)



Occlusion

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 x 2 MPixel image x avg. depth complexity 4 (assume: replace 50% of time) x 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 (later this lecture)**

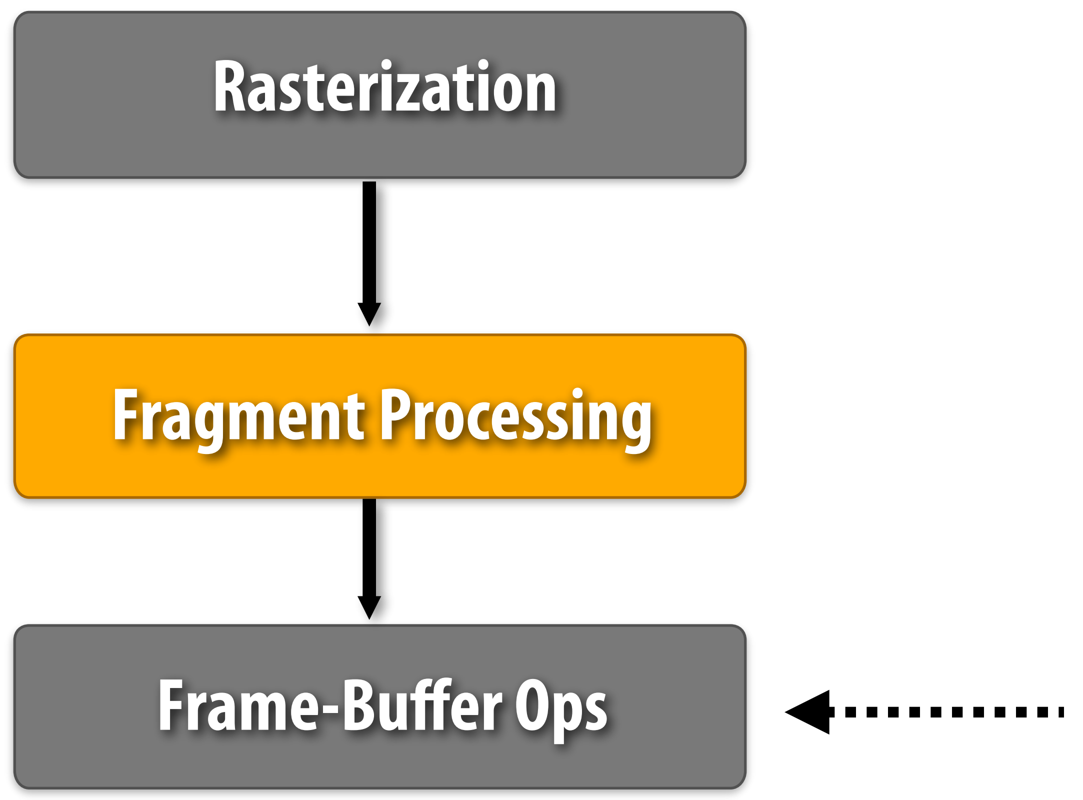
Early occlusion culling

Basic triangle drawing program

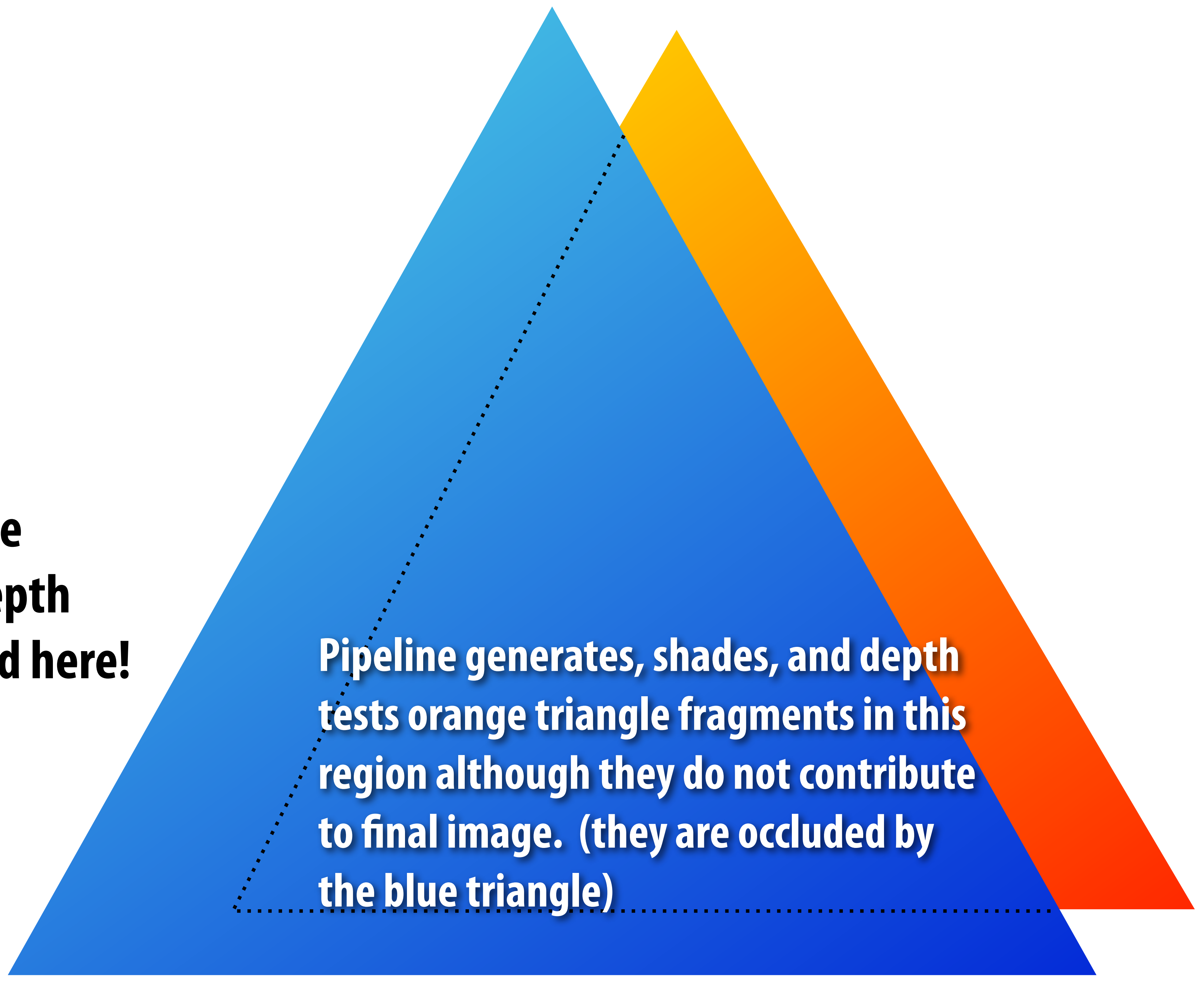
```
For each triangle  // in parallel?
    For each rasterized fragment   // in parallel
        Compute color of fragment (not discussed today)
        Perform depth test and (potentially) update frame buffer
```

Early occlusion-culling ("early Z")

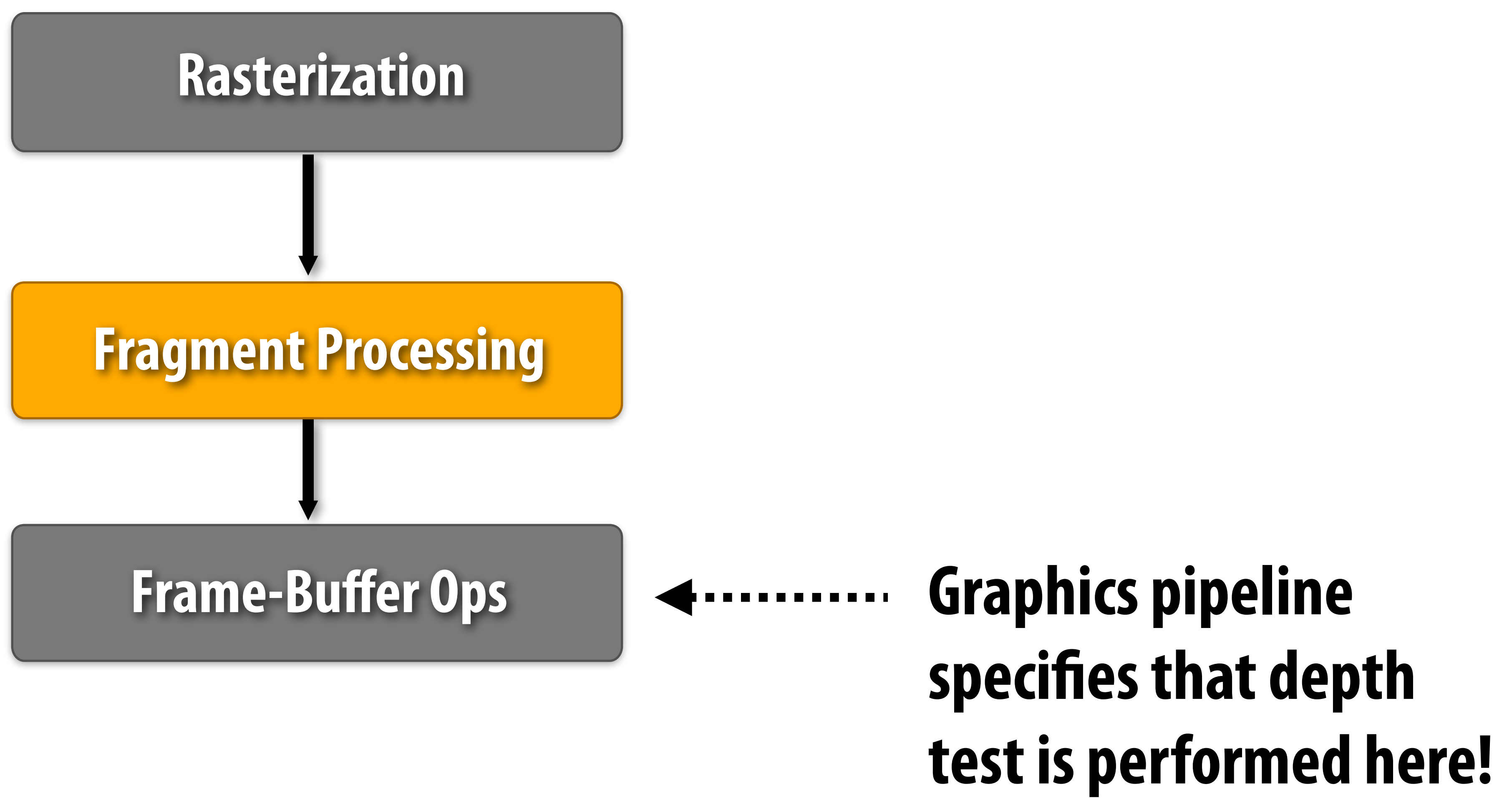
Idea: discard fragments that will not contribute to image as quickly as possible in the pipeline

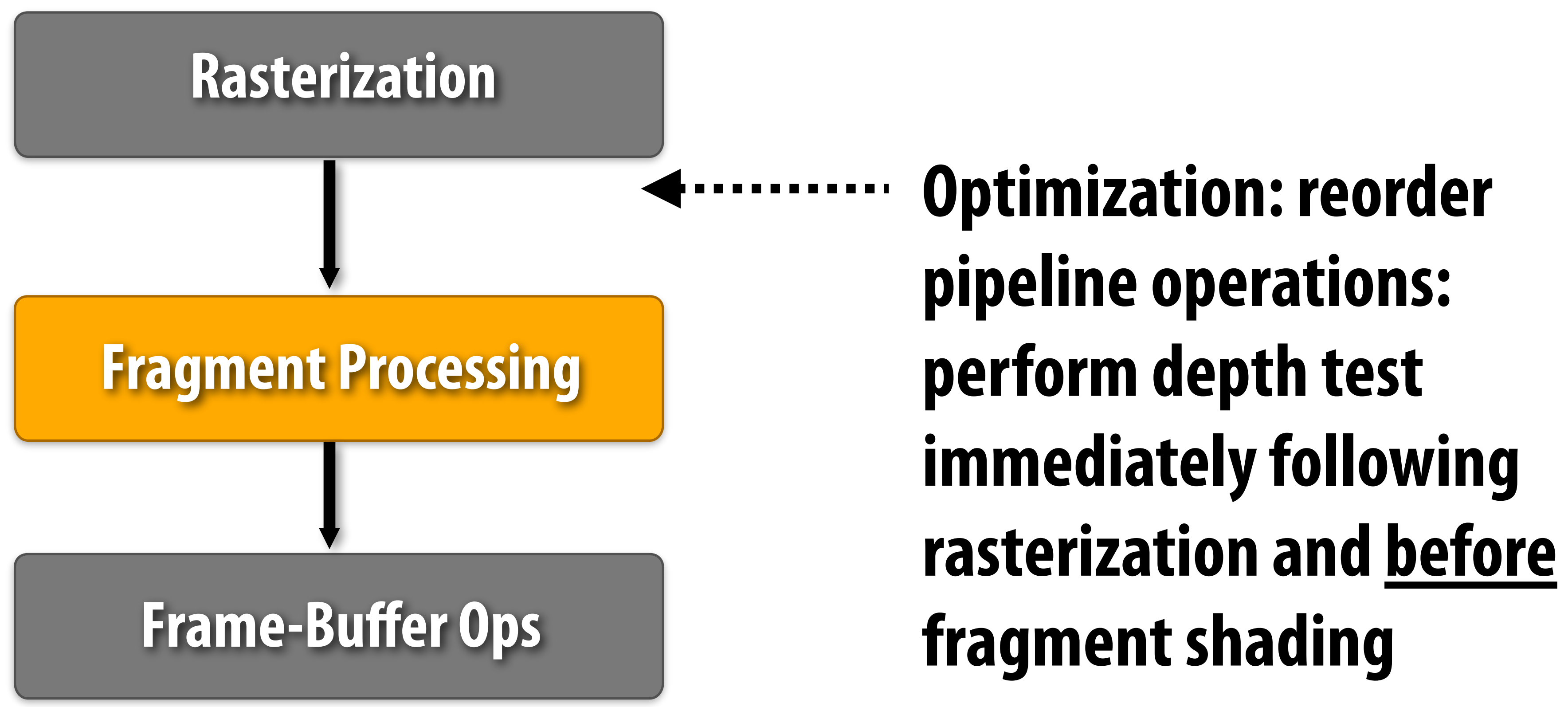


Graphics pipeline specifies that depth test is performed here!


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")


Rasterization
Fragment Processing
Frame-Buffer Ops
Graphics pipeline specifies that depth test is performed here!


Rasterization
Fragment Processing
Frame-Buffer Ops
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 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"

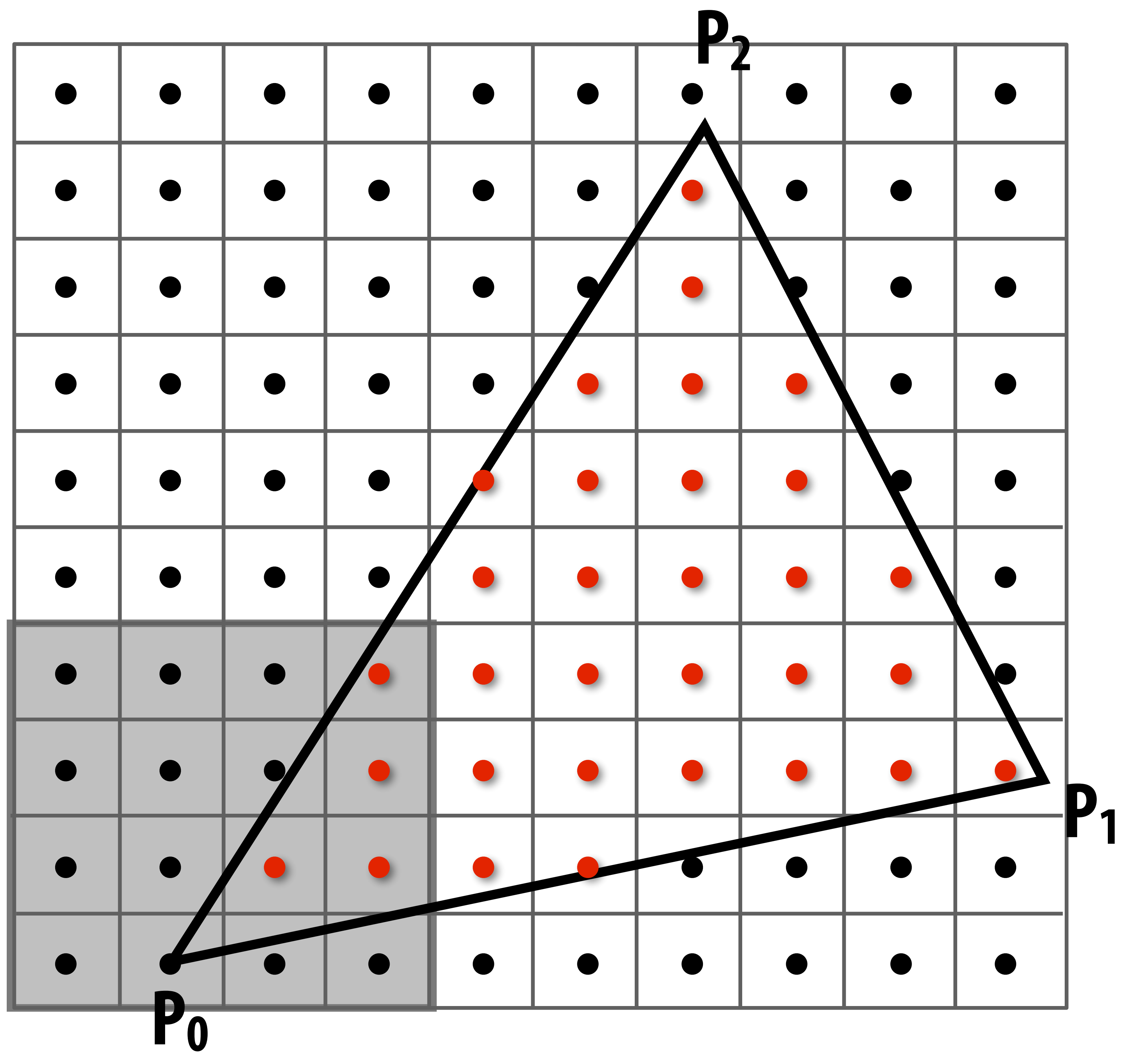
Recall hierarchical traversal during rasterization

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

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

During traversal, for each tile:

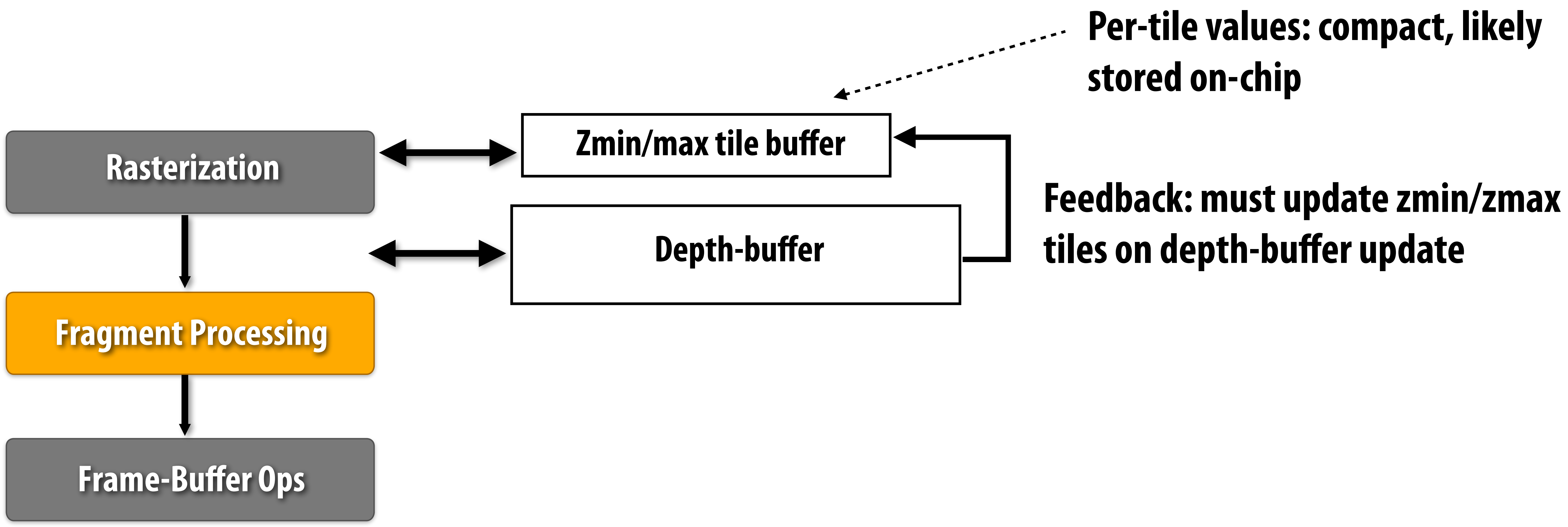
1. Compute closest point on triangle in tile: `tri_min` (using Z plane equation)
2. If `tri_min > z_max`, then triangle is completely occluded in this tile. (The depth test will fail for all samples in the tile.) Proceed to next tile without performing coverage tests for individual samples in tile.



<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
Zmin/max tile buffer
Rasterization
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)

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 unsets the "cleared" 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 <u>footprint</u> is a non-goal

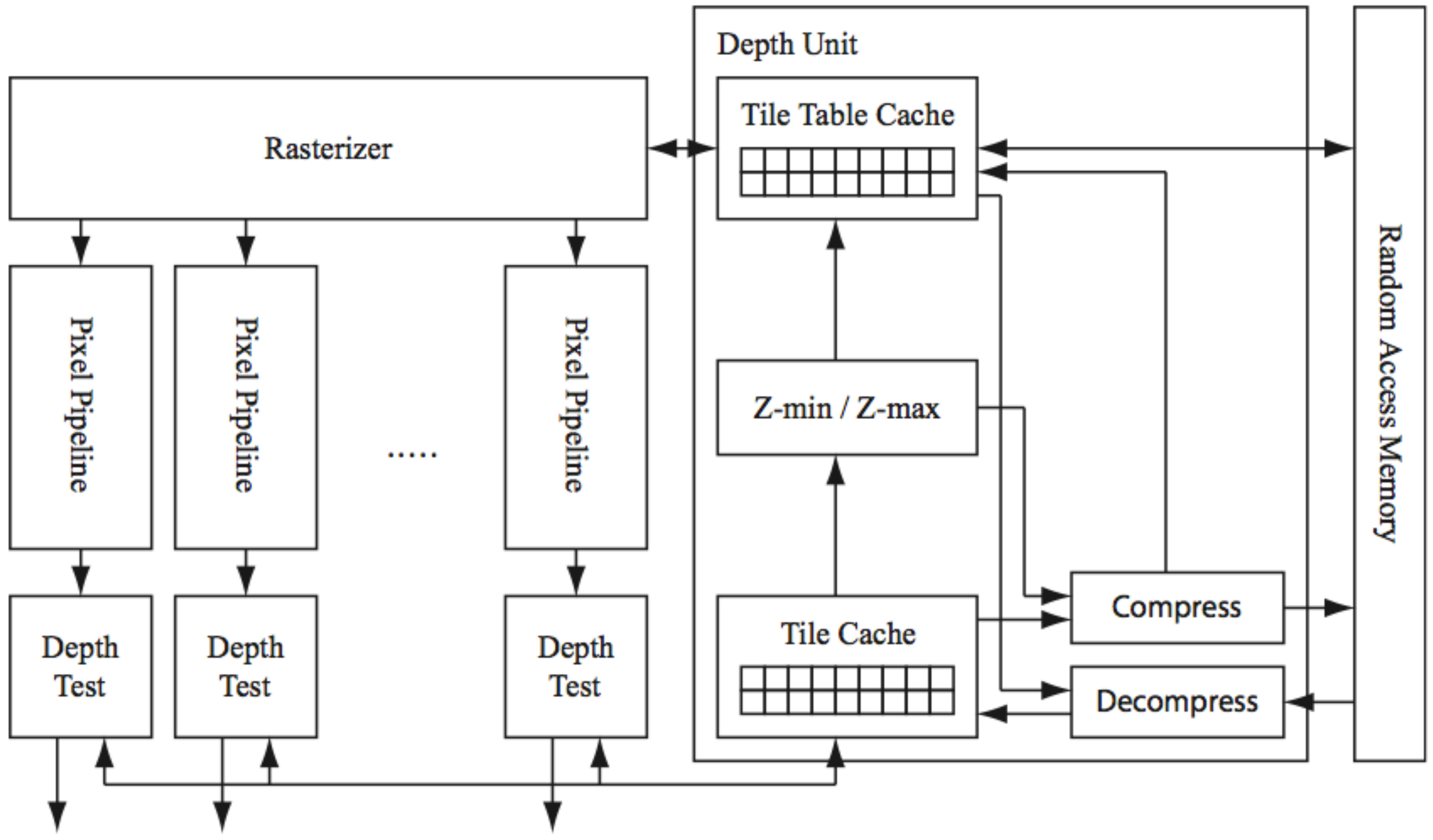
    (Need for real-time guarantees in graphics applications requires application to plan for worst case anyway)

- **Lossless compression**
  - Q. 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

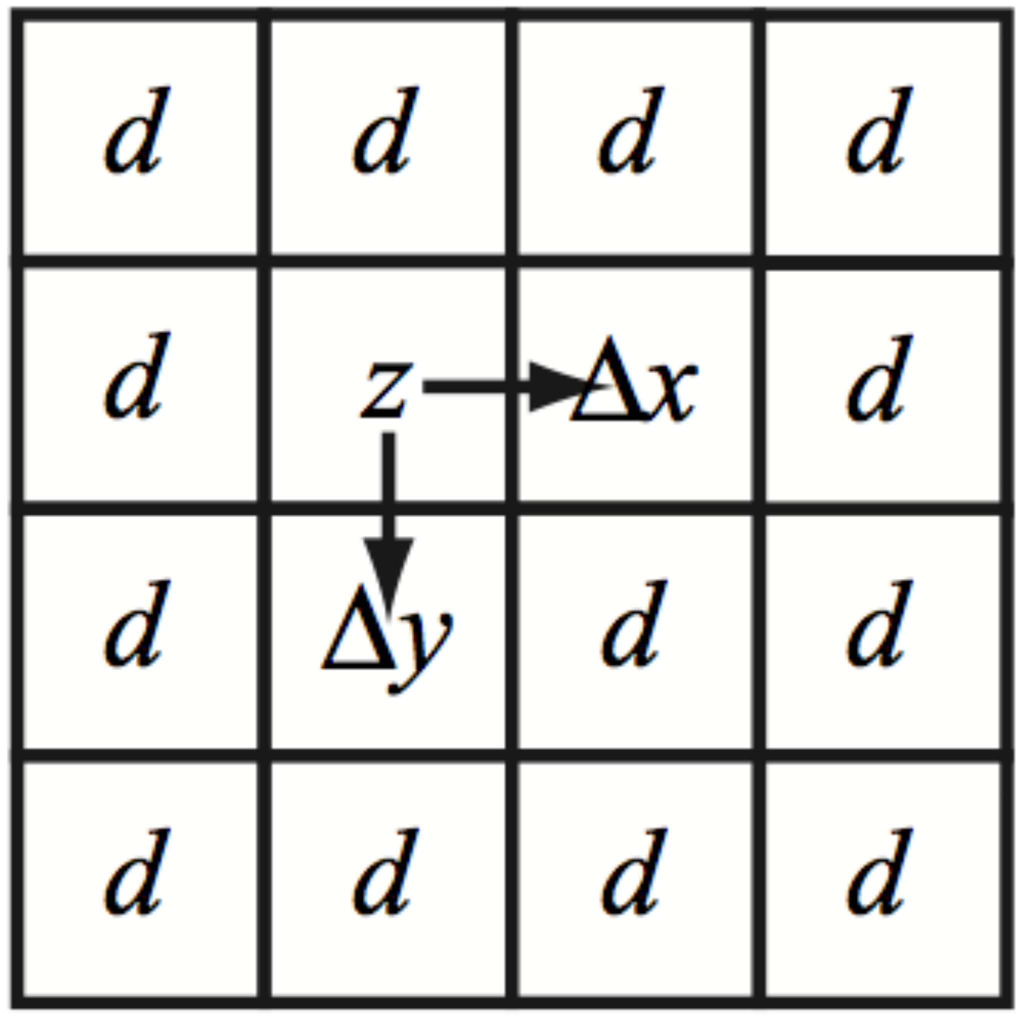
On tile load:

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

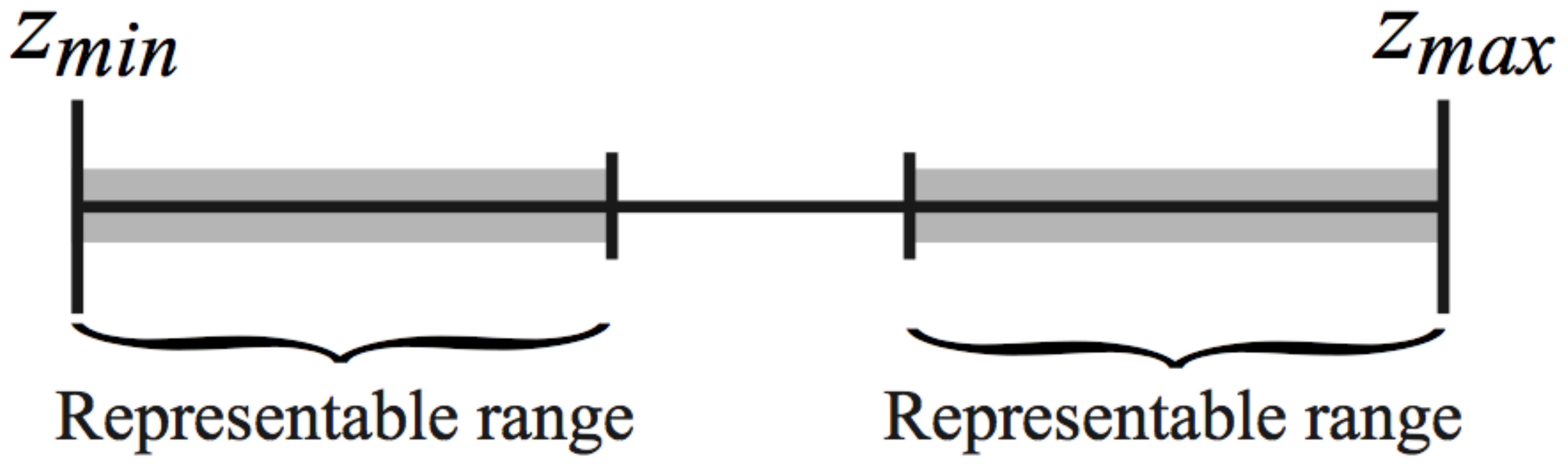
- Choose anchor value and compute DX, DY from adjacent pixels (fits a plane to the data)
- Use plane to predict depths at other pixels, store offset d from prediction at each pixel
- Scheme (for 24-bit depth buffer)
  - Anchor: 24 bits (full resolution)
  - DX, DY: 15 bits
  - Per-sample offsets: 5 bits



[Van Dyke and Margeson]

Depth-offset compression

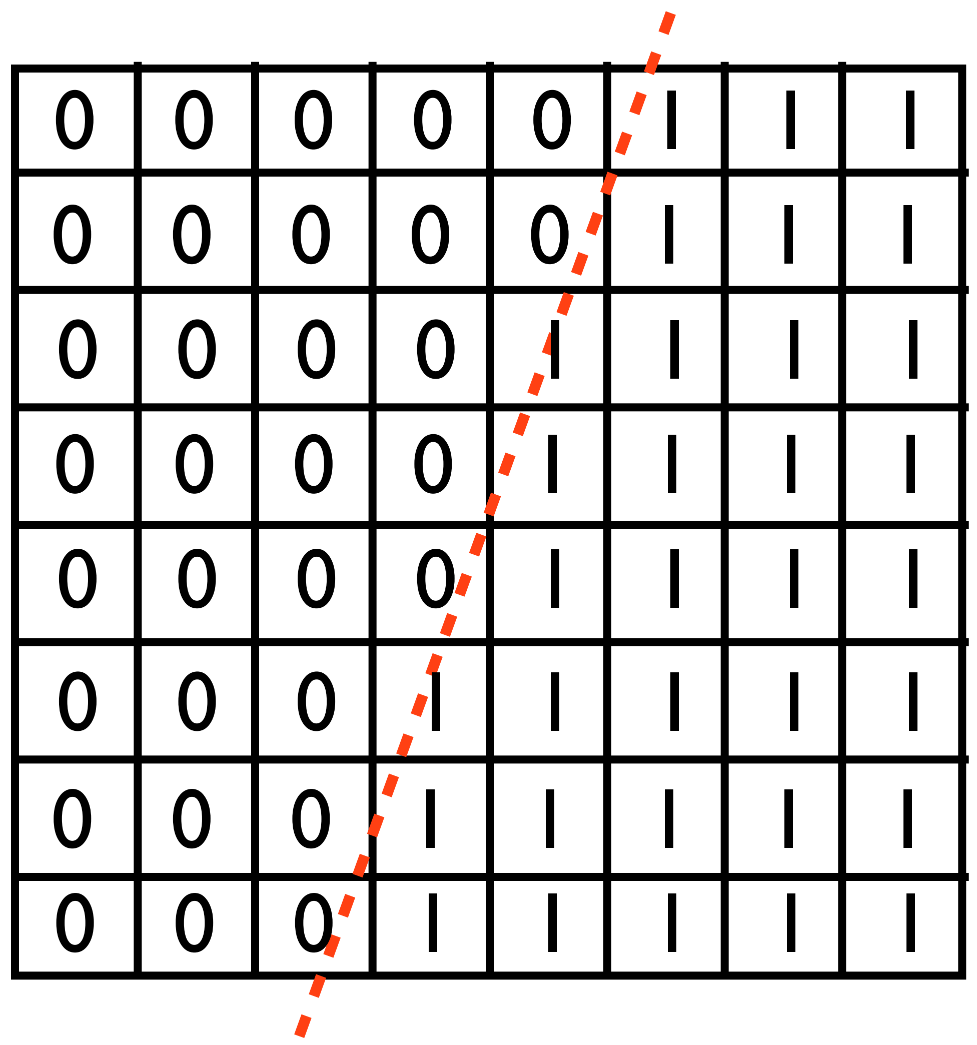
- Assume depth values have low dynamic range relative to tile's zmin and zmaz (assume two surfaces)
- Store zmin/zmax (need to anyway for hierarchical Z)
- Store low-precision (8-12 bits) offset value for each sample
    - MSB encodes if offset is from zmin or zmax



[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



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 accesses individual sample data less often**

- **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

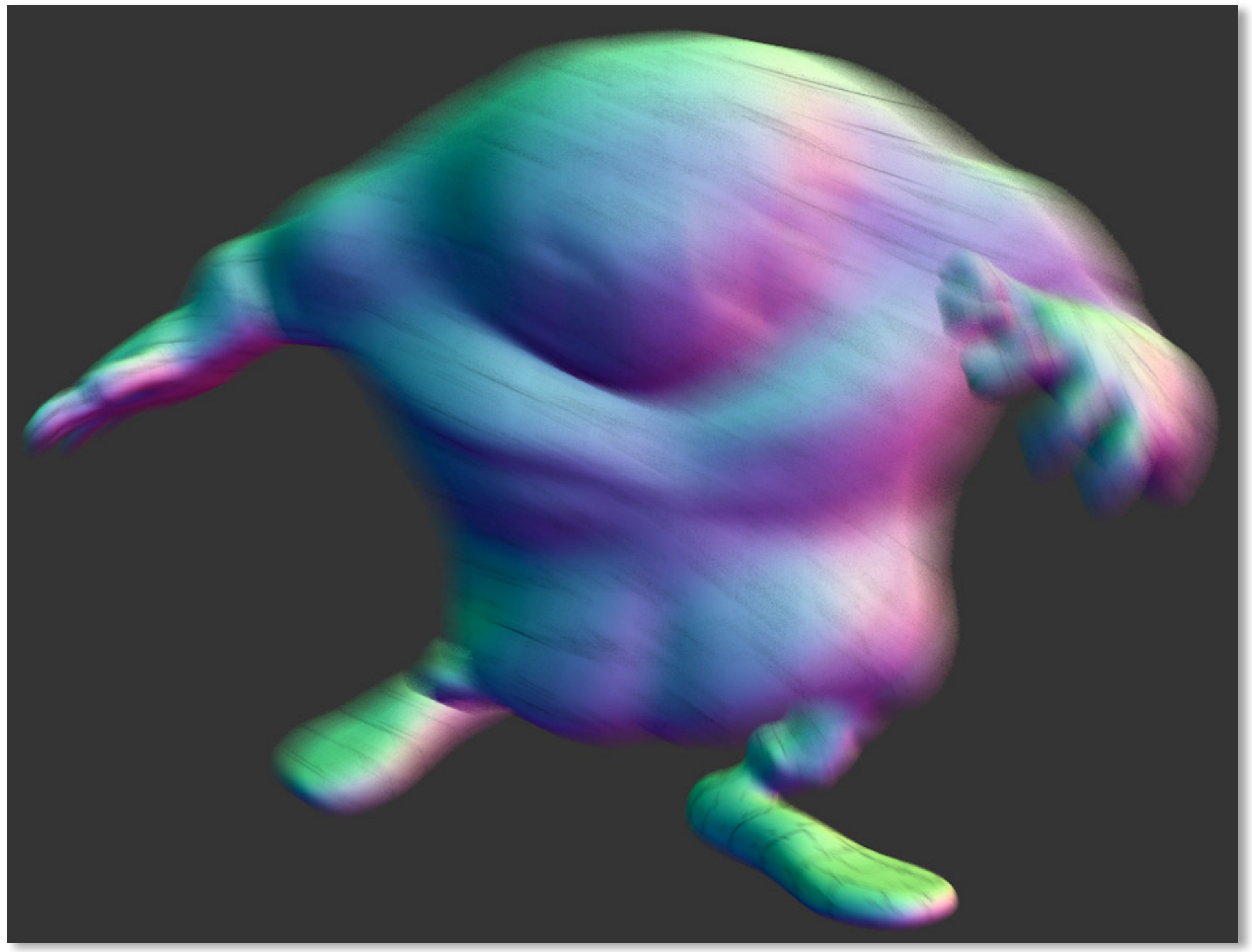
- **Hierarchical traversal during rasterization**
  - Leveraged to reduce coverage testing and occlusion work
  - Tile size likely coupled to hierarchical Z granularity
  - May also be coupled to compression tile granularity

- **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 a virtual pinhole camera with and infinitely fast shutter
  - Real cameras have finite apertures and finite exposure duration
  - Accurate camera simulation requires visibility computation to be integrated over time and lens aperture (very high computational cost, see readings)

Time integration: motion blur



Lens aperture integration: defocus blur