

Lecture 8:

Depth and Transparency

Computer Graphics
CMU 15-462/15-662

Today: Wrap up the rasterization pipeline!

Remember our goal:

- Start with **INPUTS** (triangles)
 - possibly w/ other data (e.g., colors or texture coordinates)
- Apply a series of transformations: **STAGES** of pipeline
- Produce **OUTPUT** (final image)

INPUT (TRIANGLES)

VERTICES

A: (1, 1, 1)	E: (1, 1, -1)
B: (-1, 1, 1)	F: (-1, 1, -1)
C: (1, -1, 1)	G: (1, -1, -1)
D: (-1, -1, 1)	H: (-1, -1, -1)

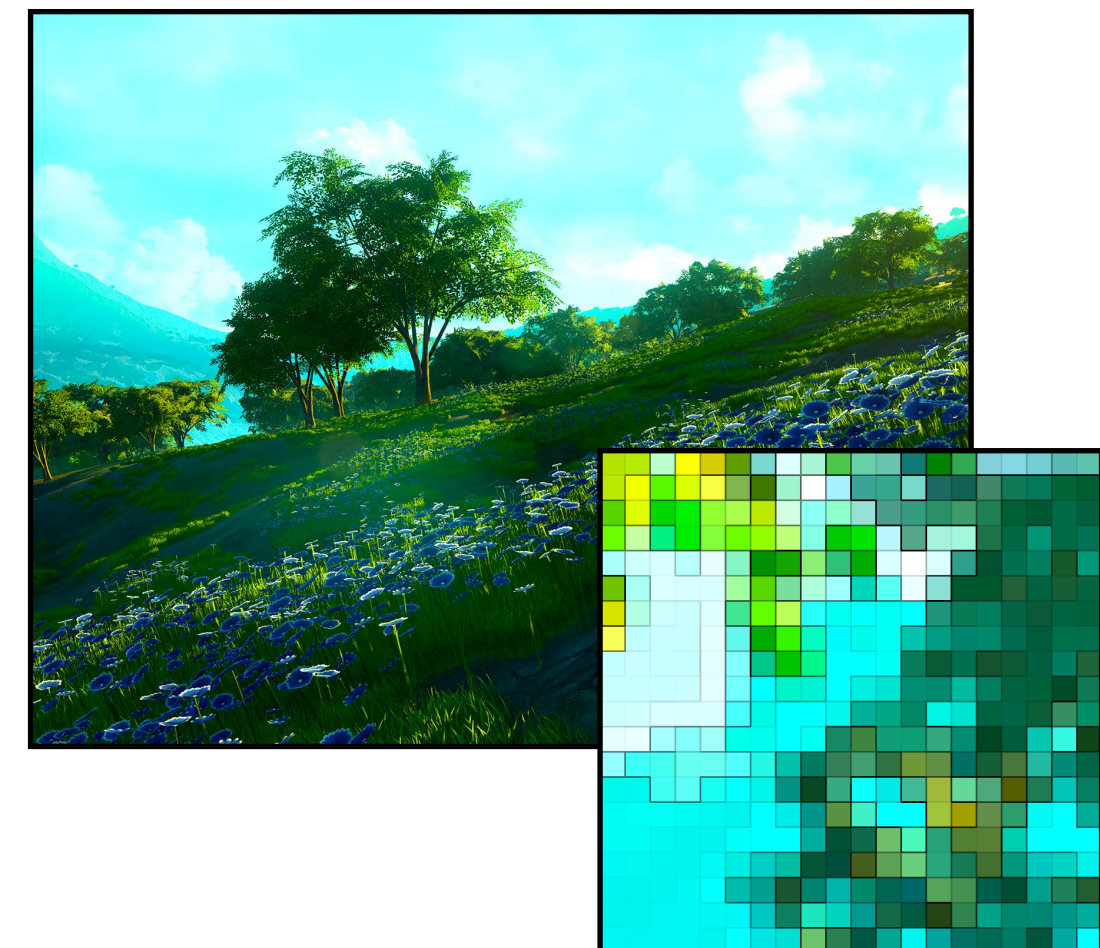
TRIANGLES

EHF, GFH, FGB, CBG,
GHC, DCH, ABD, CDB,
HED, ADE, EFA, BAF

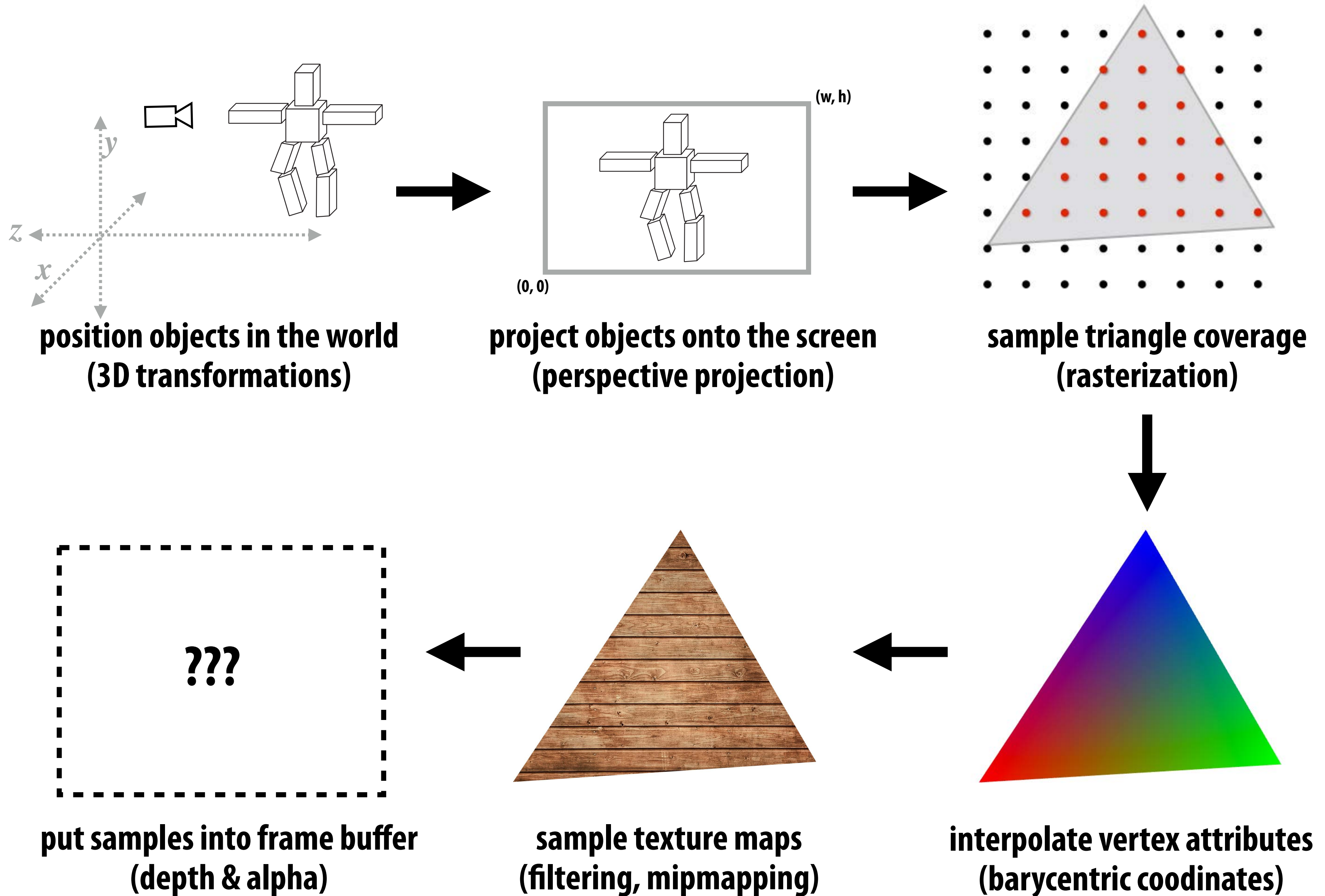
RASTERIZATION PIPELINE



OUTPUT (BITMAP IMAGE)

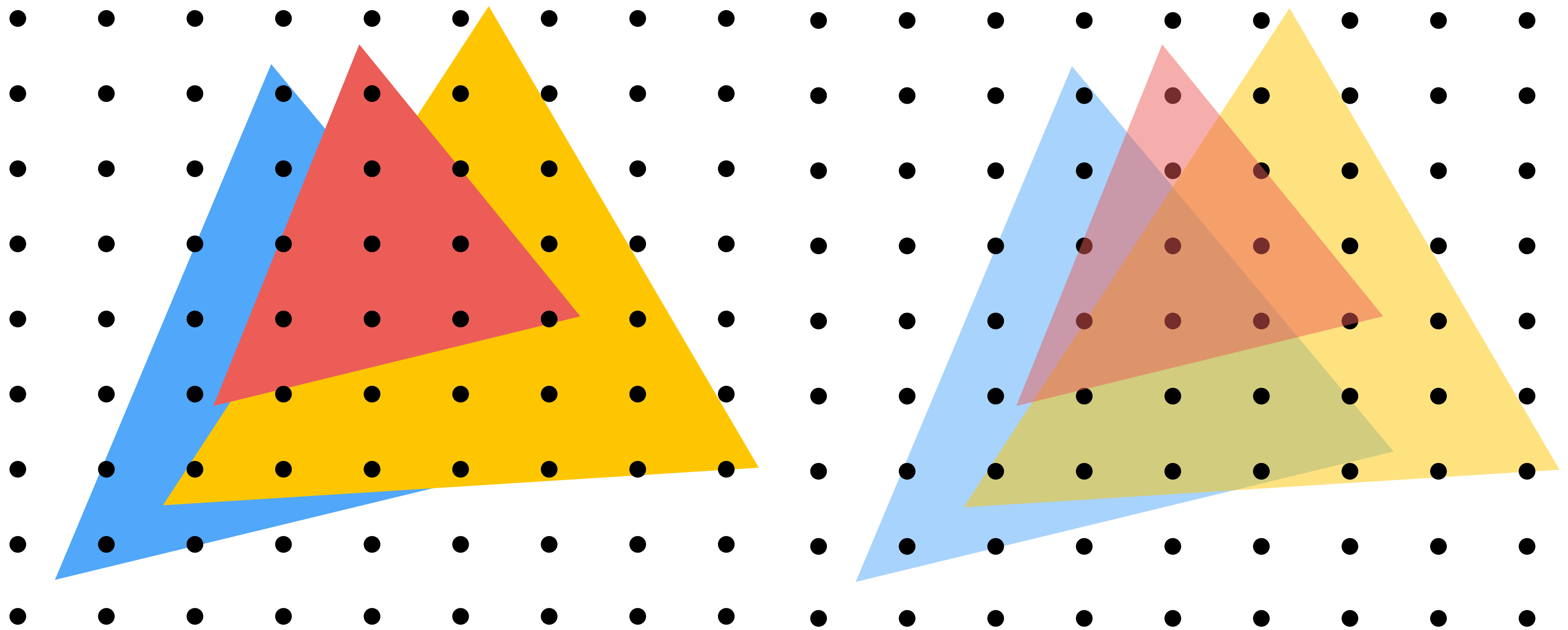


What we know how to do so far...



Occlusion

Occlusion: which triangle is visible at each covered sample point?



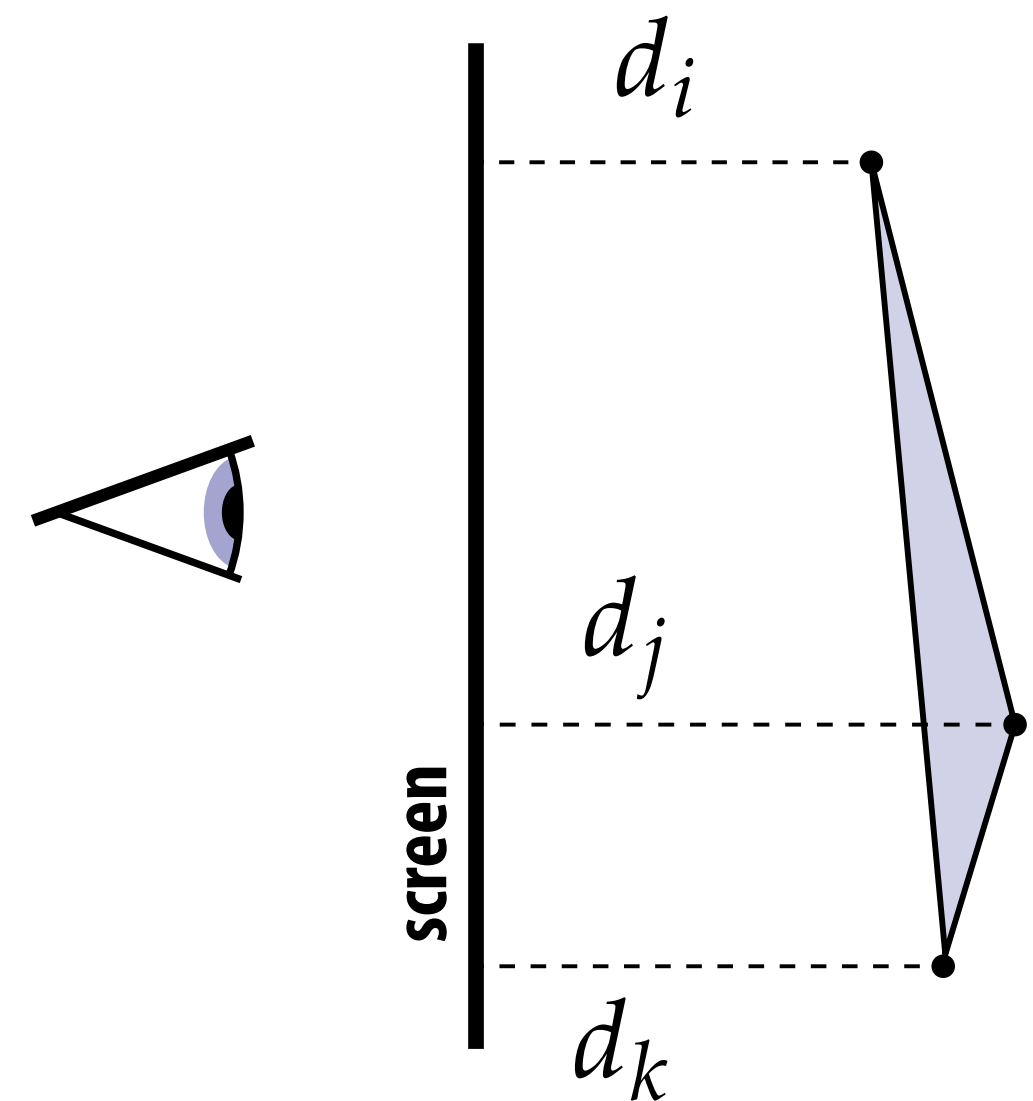
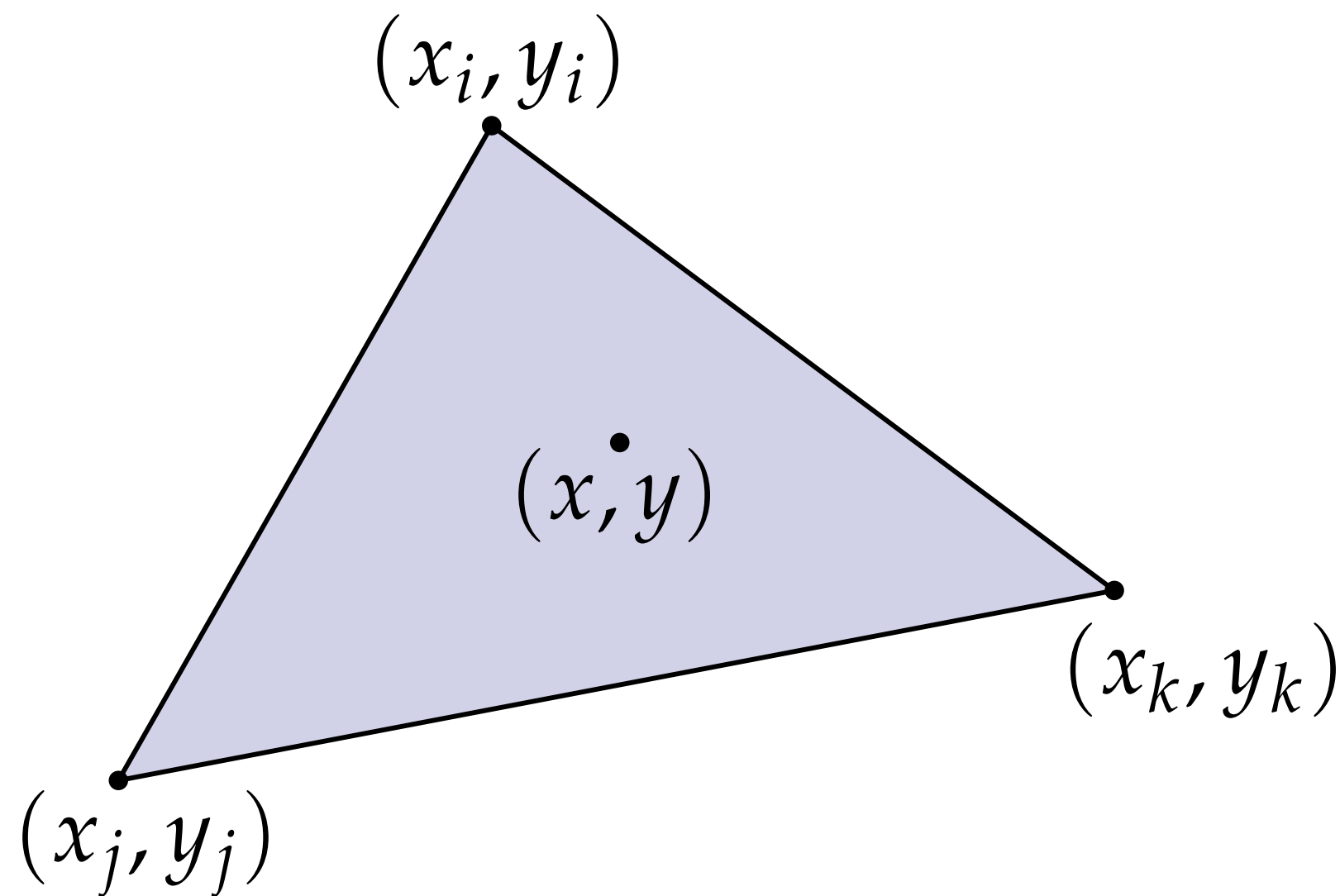
Opaque Triangles

50% transparent triangles

Sampling Depth

Assume we have a triangle given by:

- the projected 2D coordinates (x_i, y_i) of each vertex
- the “depth” d_i of each vertex (i.e., distance from the viewer)



Q: How do we compute the depth d at a given sample point (x, y) ?

A: Interpolate it using barycentric coordinates—just like any other attribute that varies linearly over the triangle

The depth-buffer (Z-buffer)

For each sample, *depth-buffer* stores the depth of the **closest** triangle seen so far



Initialize all depth buffer values to “infinity” (max value)

Depth buffer example

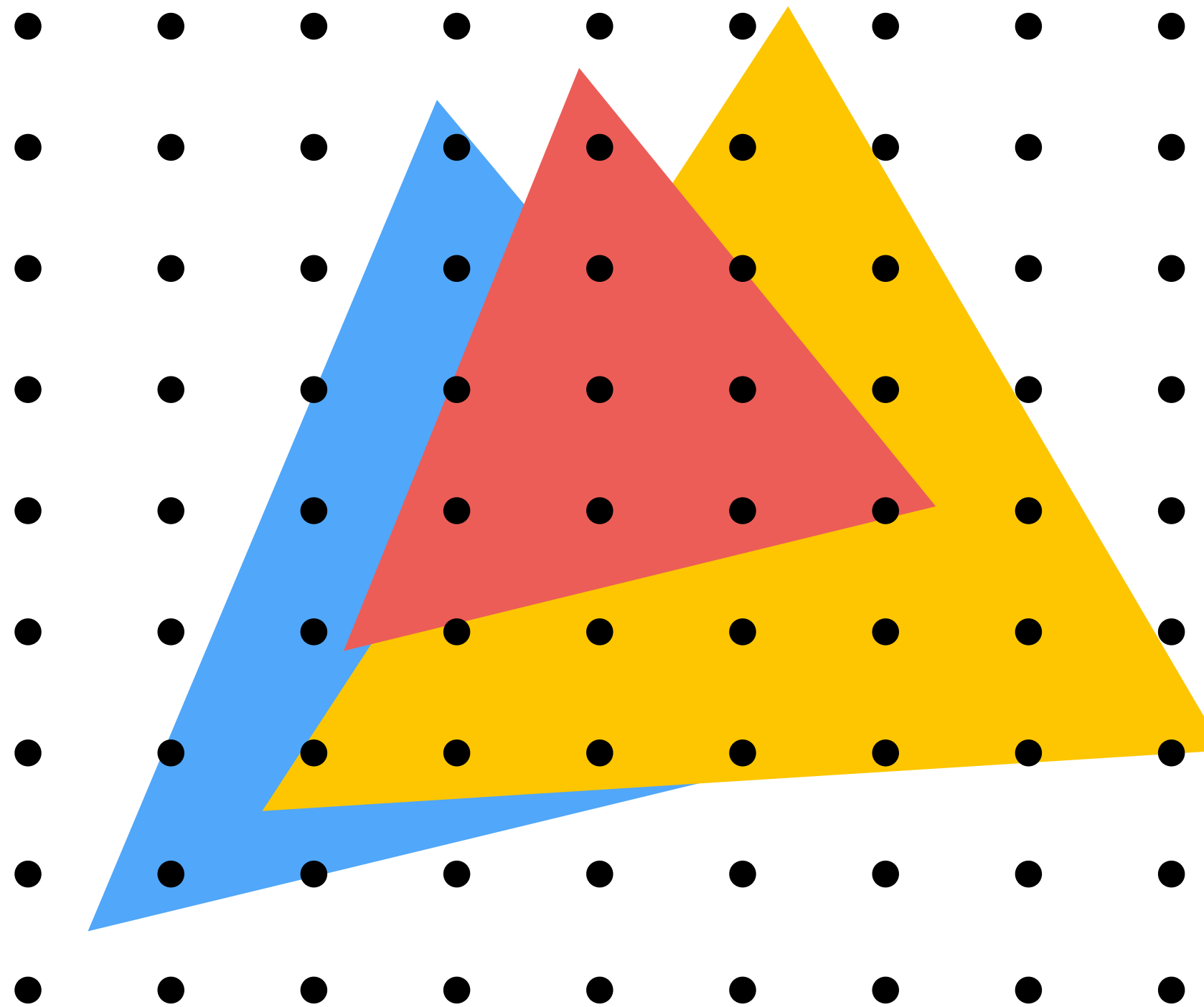


near



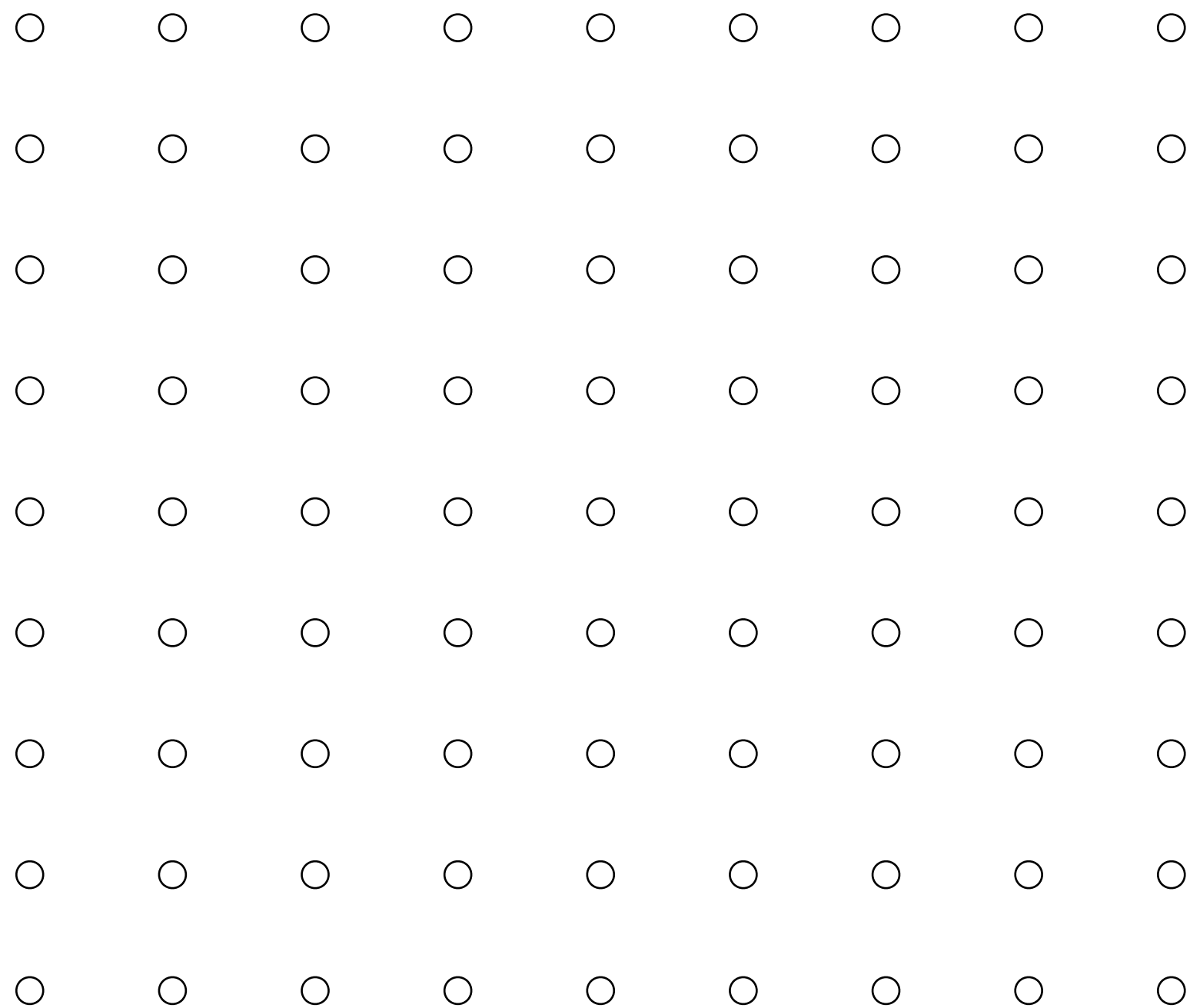
far

Example: rendering three opaque triangles



Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:
depth = 0.5



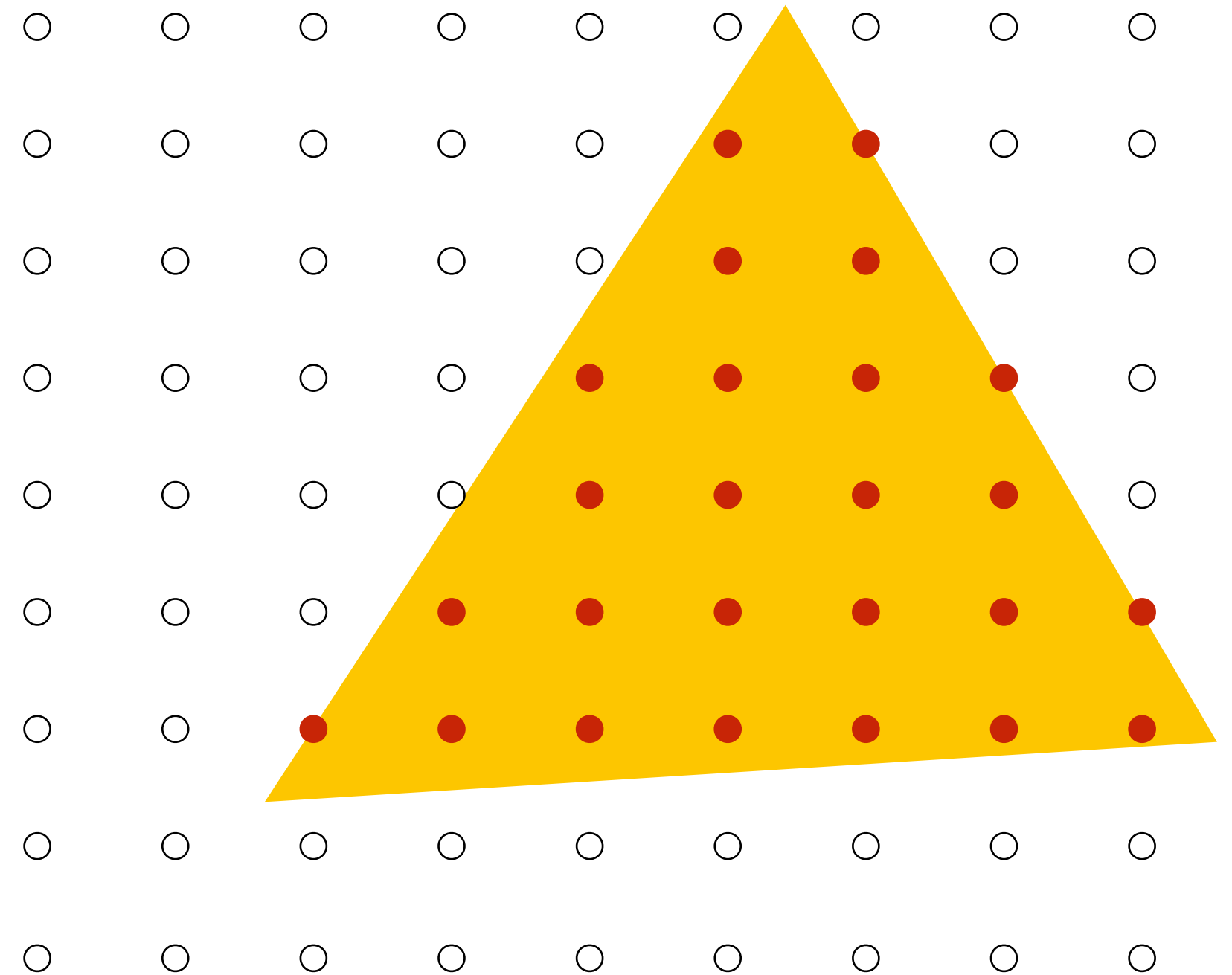
Color buffer contents

near



far

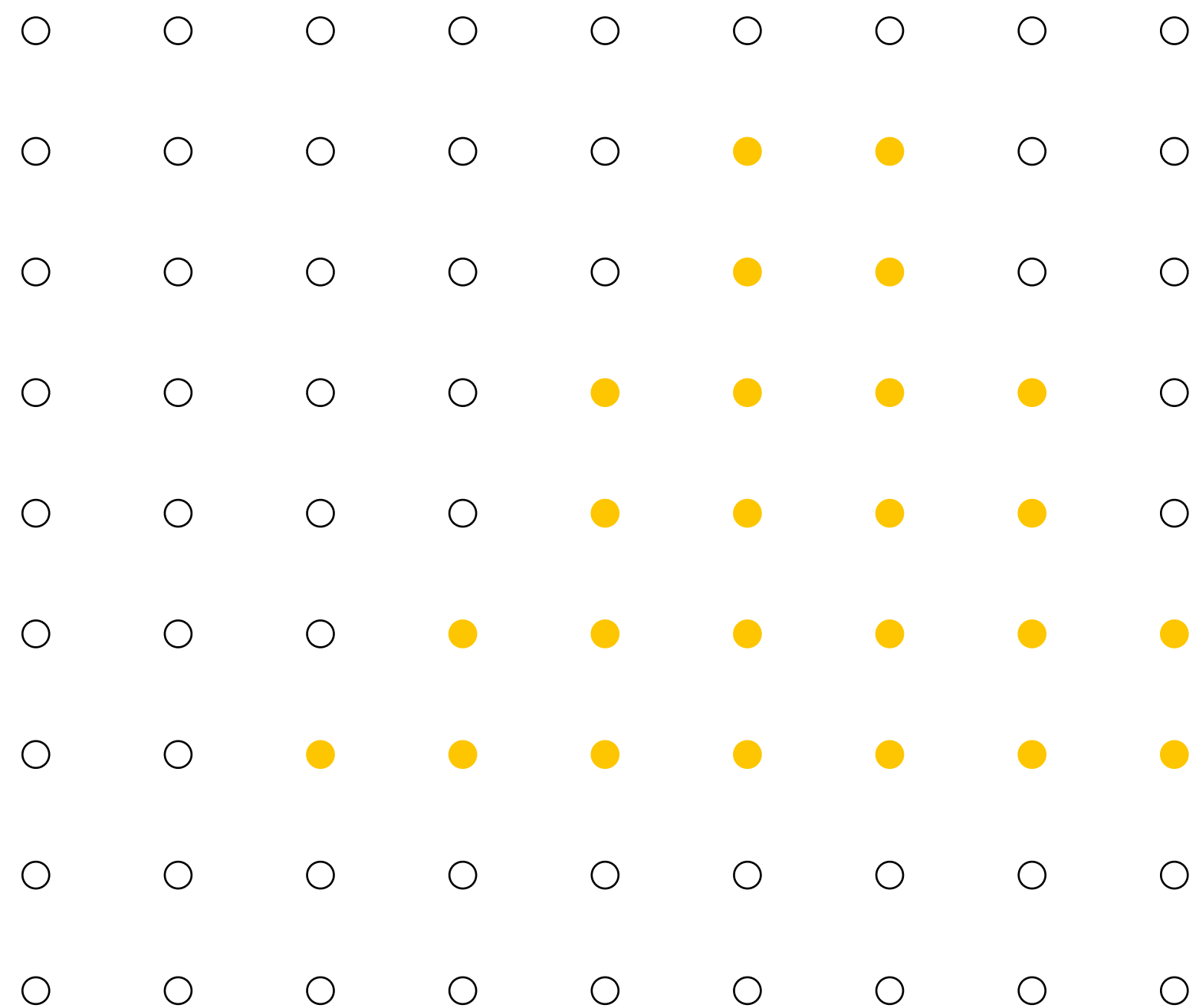
● — sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing yellow triangle:



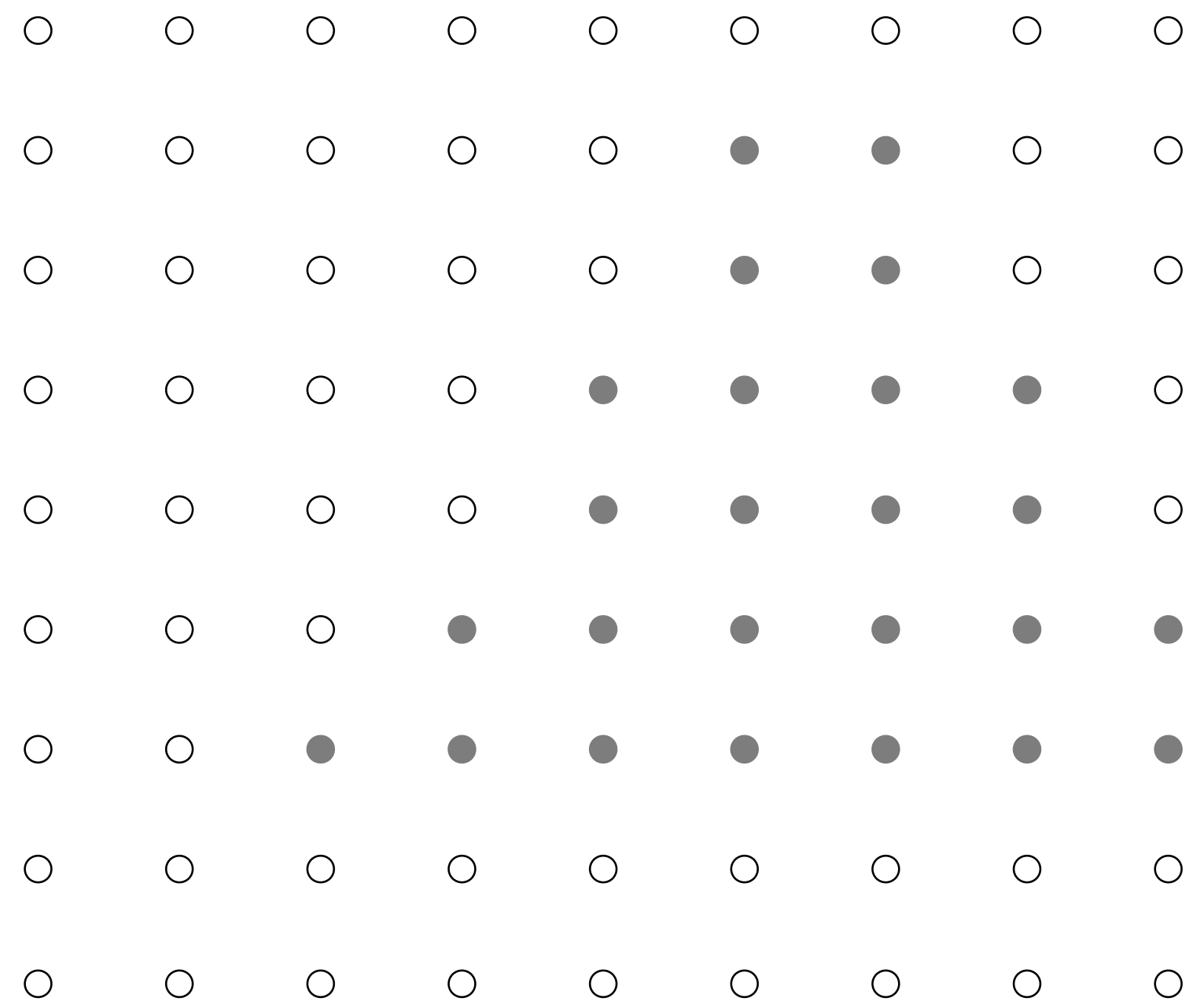
Color buffer contents

near



far

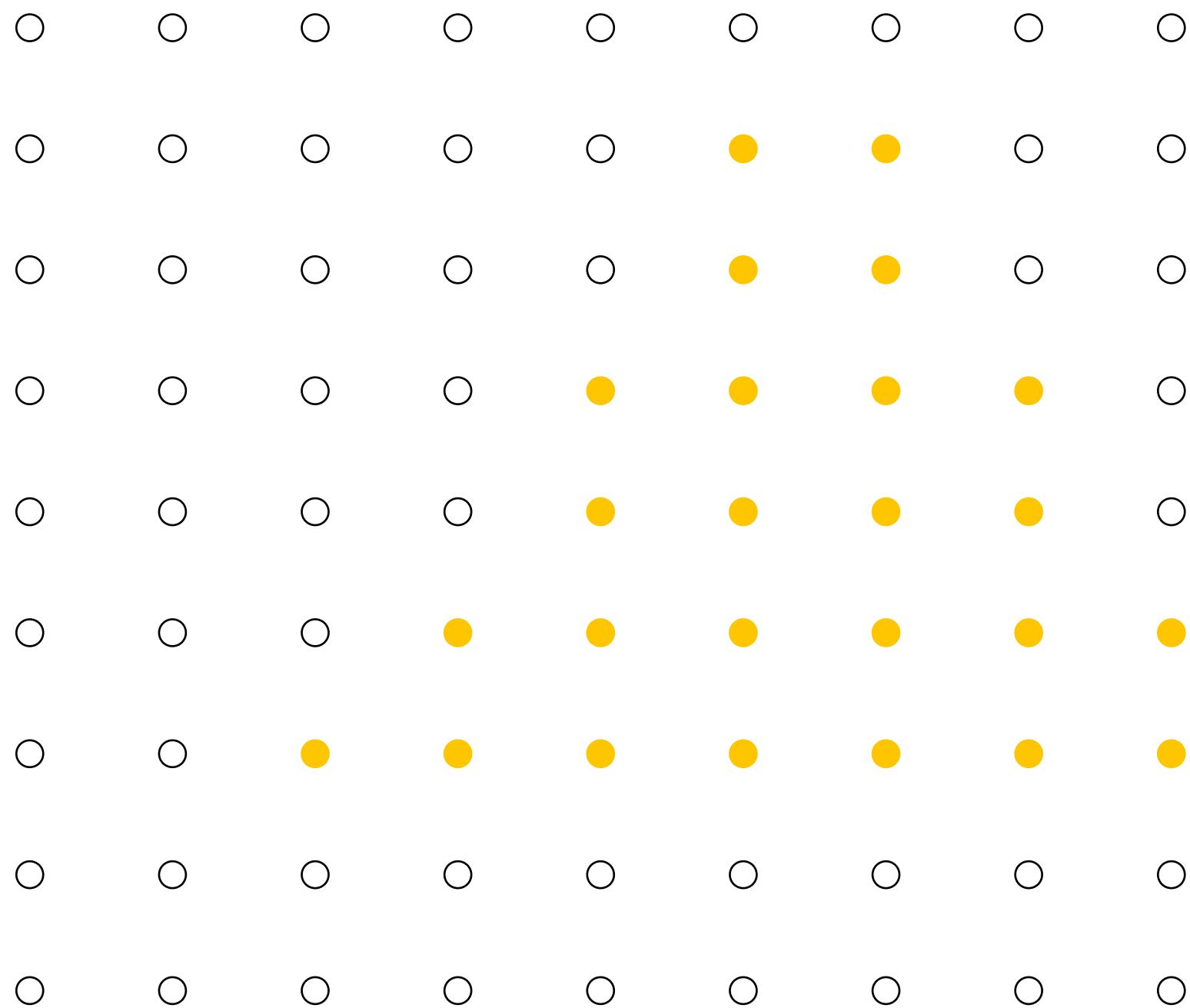
● — sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:
depth = 0.75



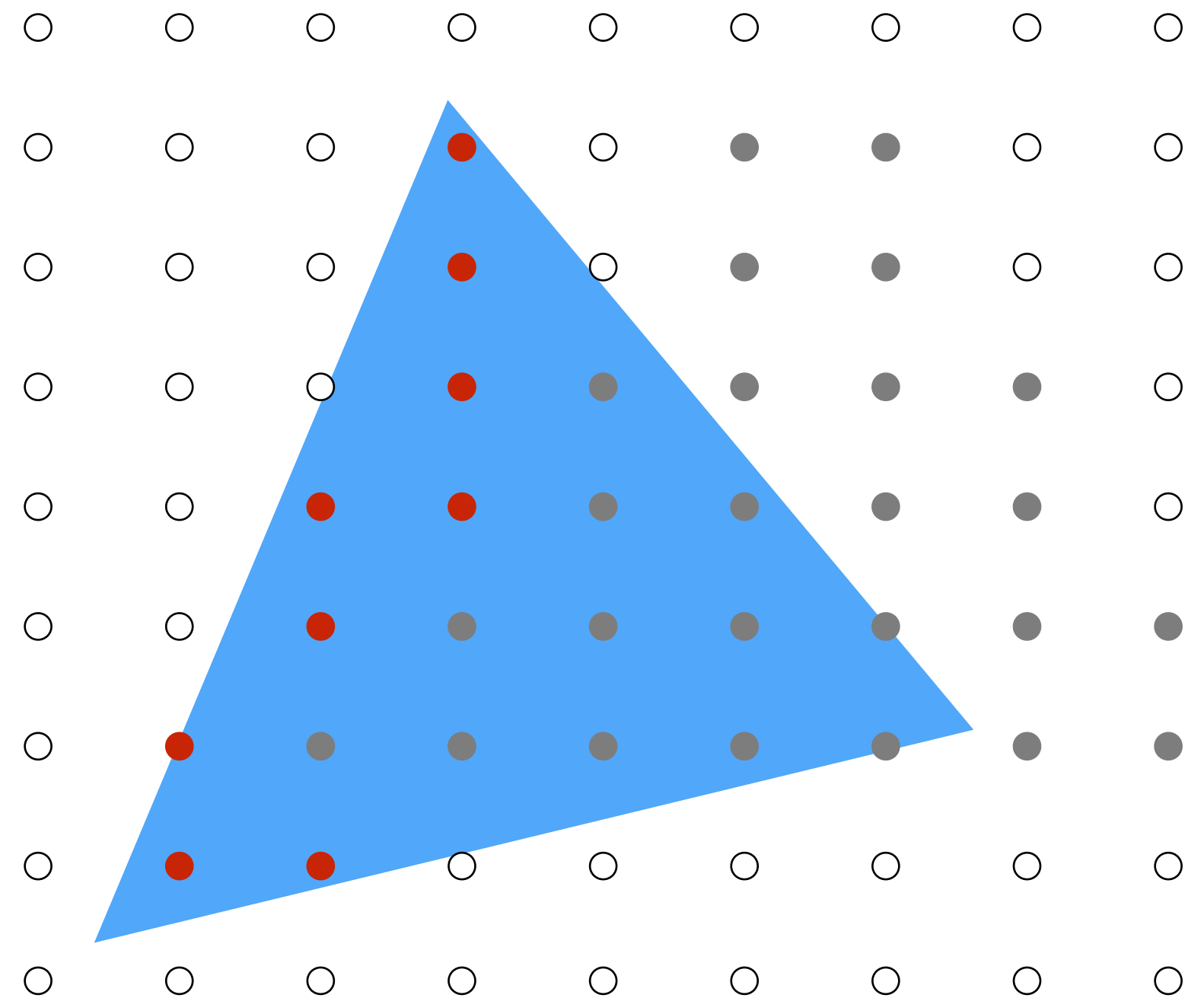
Color buffer contents

near



far

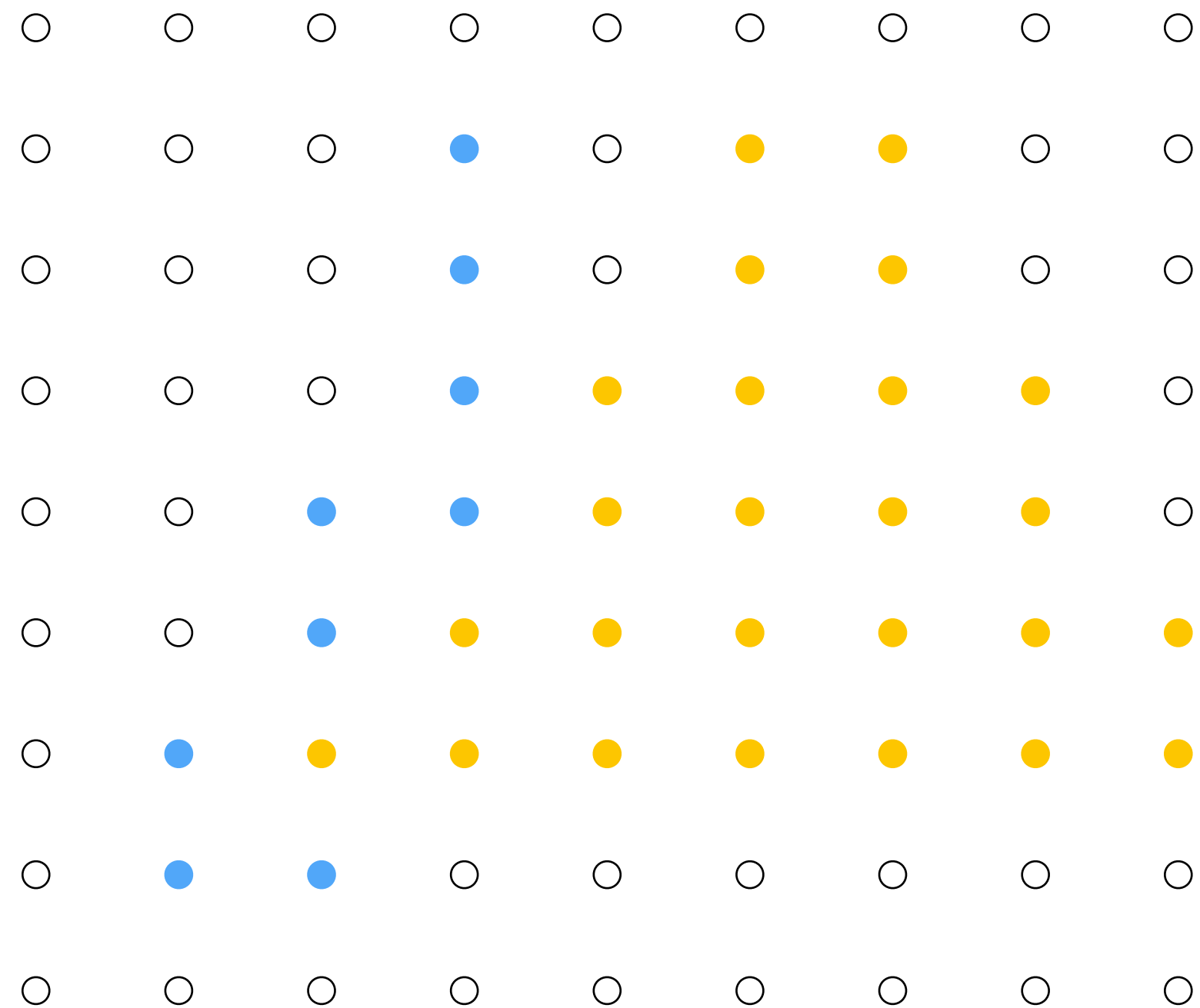
● — sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

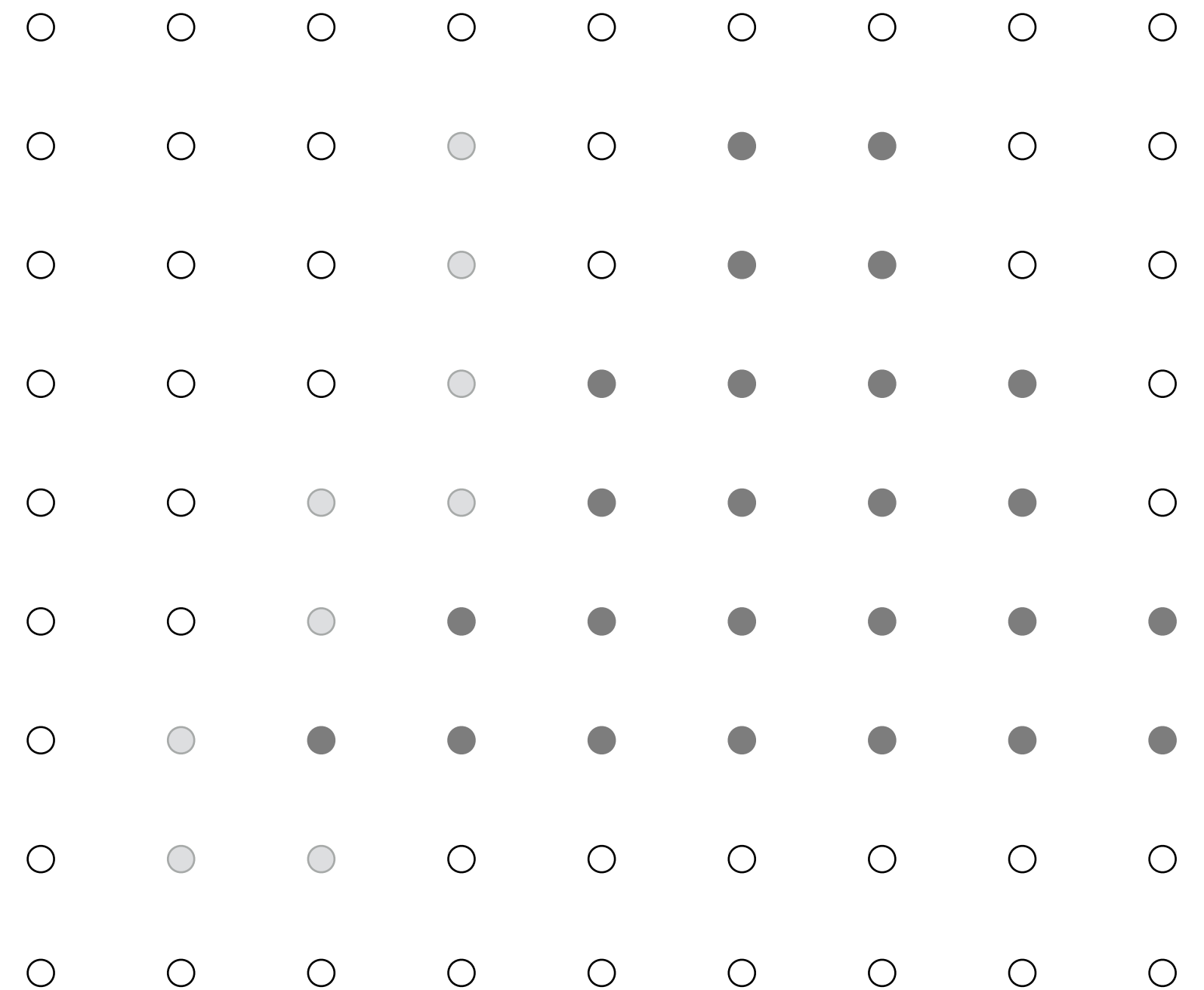
After processing blue triangle:



Color buffer contents

near  far

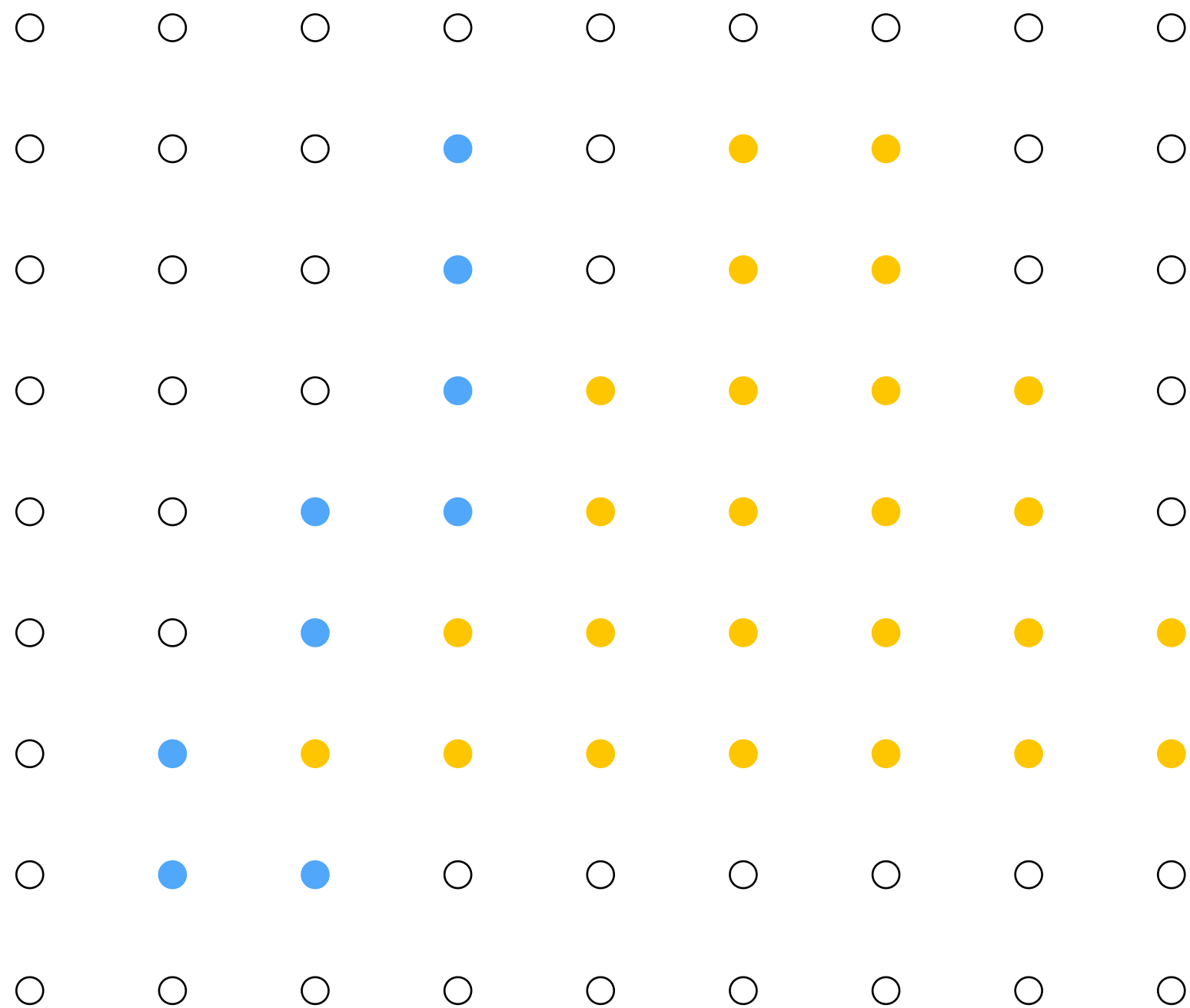
● — sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

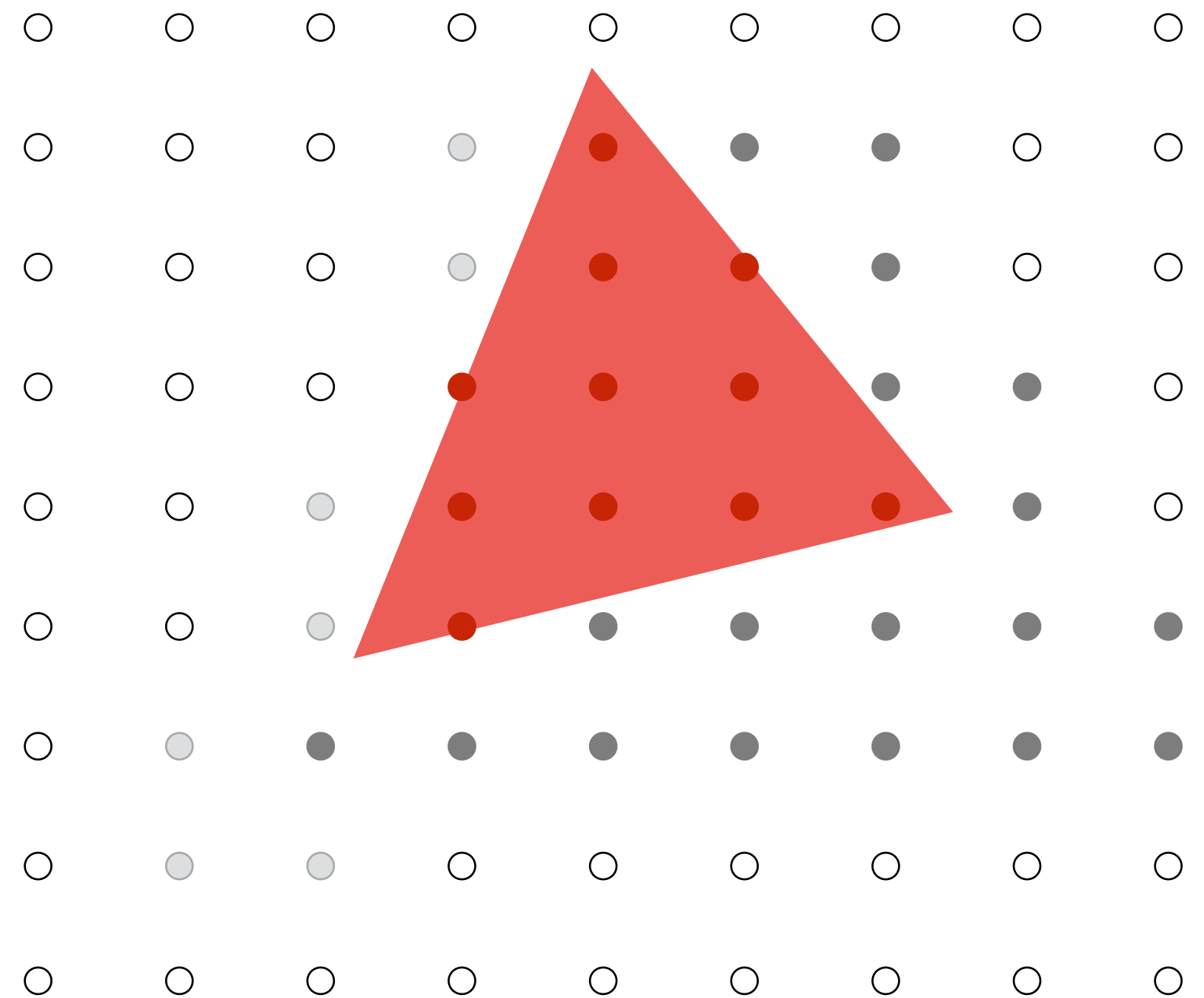
Processing red triangle:
depth = 0.25



Color buffer contents

near  far

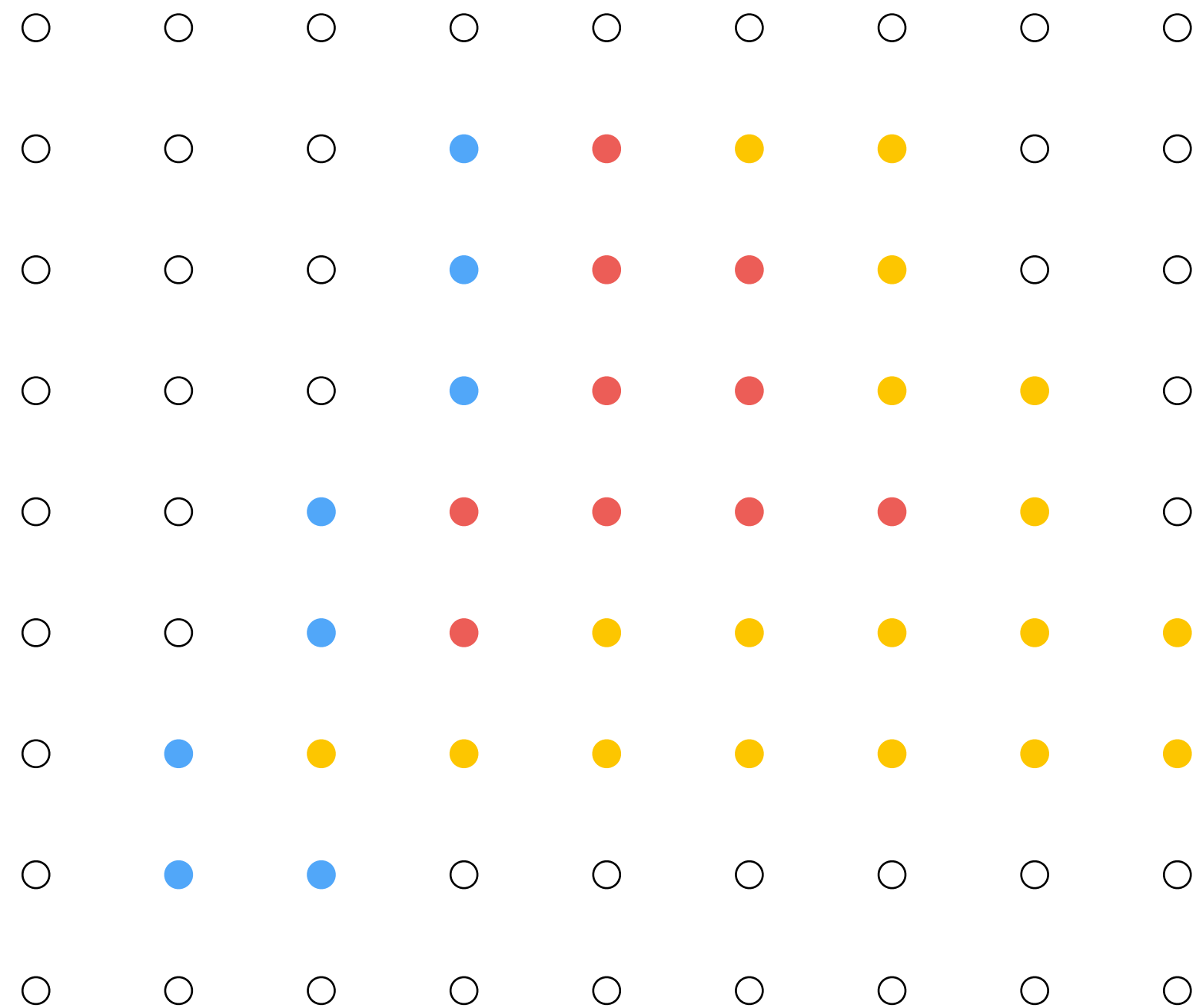
● — sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

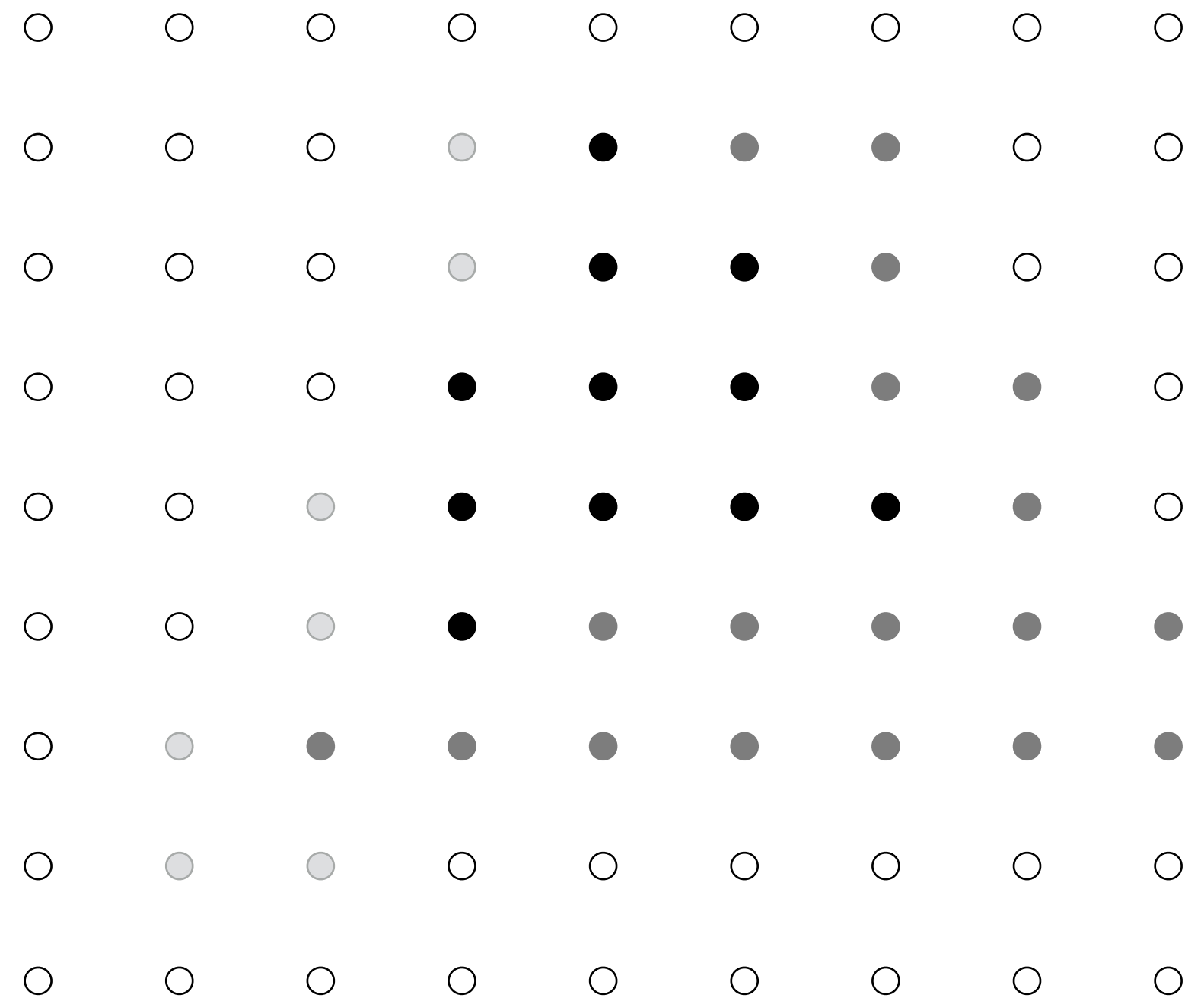
After processing red triangle:



Color buffer contents

near  far

● — sample passed depth test



Depth buffer contents

Occlusion using the depth buffer

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

```
draw_sample(x, y, d, c) //new depth d & color c at (x,y)
{
    if( pass_depth_test( d, zbuffer[x][y] ))
    {
        // triangle is closest object seen so far at this
        // sample point. Update depth and color buffers.
        zbuffer[x][y] = d; // update zbuffer
        color[x][y] = c;   // update color buffer
    }
    // otherwise, we've seen something closer already;
    // don't update color or depth
}
```

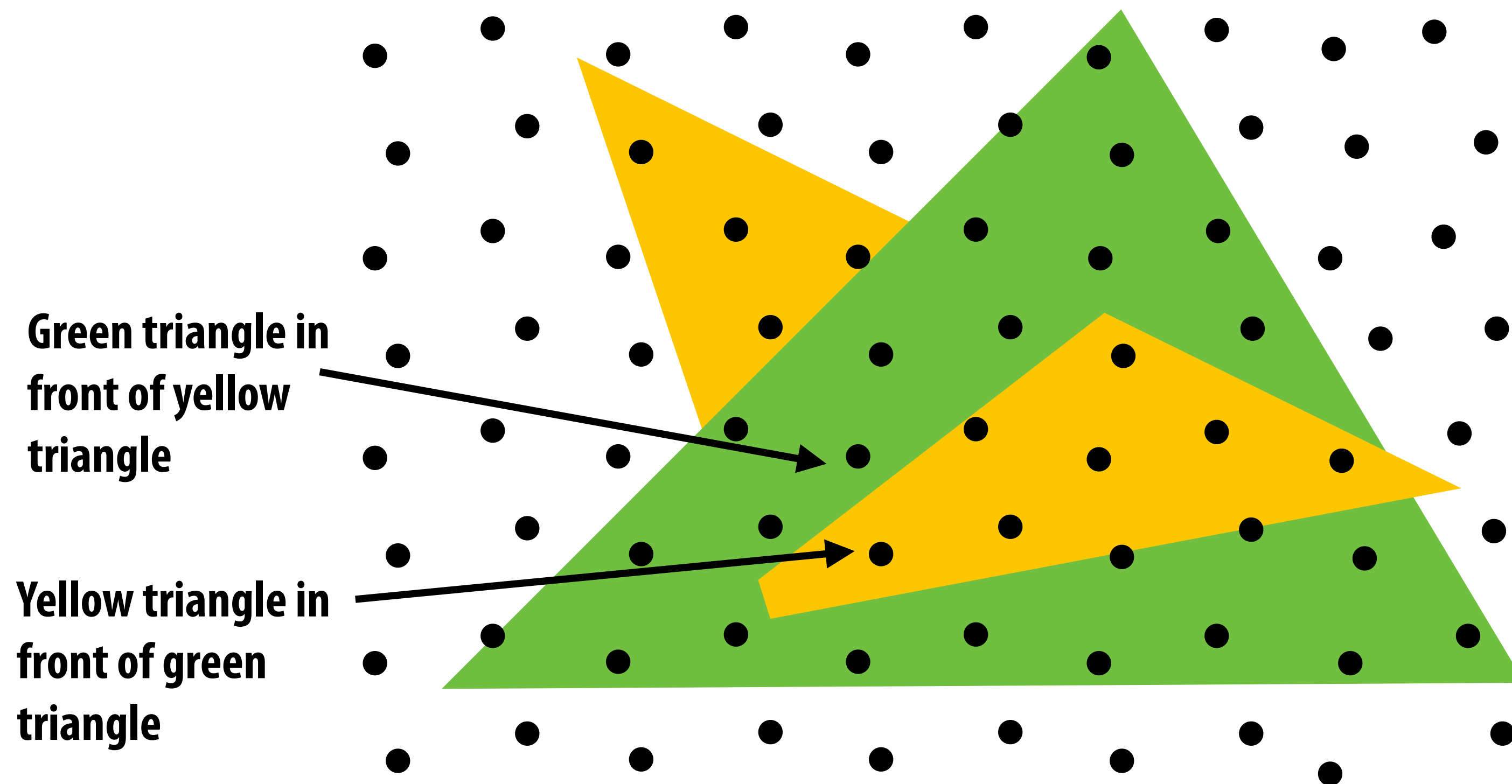
Depth + Intersection

Q: Does depth-buffer algorithm handle interpenetrating surfaces?

A: Of course!

Occlusion test is based on depth of triangles at a given sample point.

Relative depth of triangles may be different at different sample points.



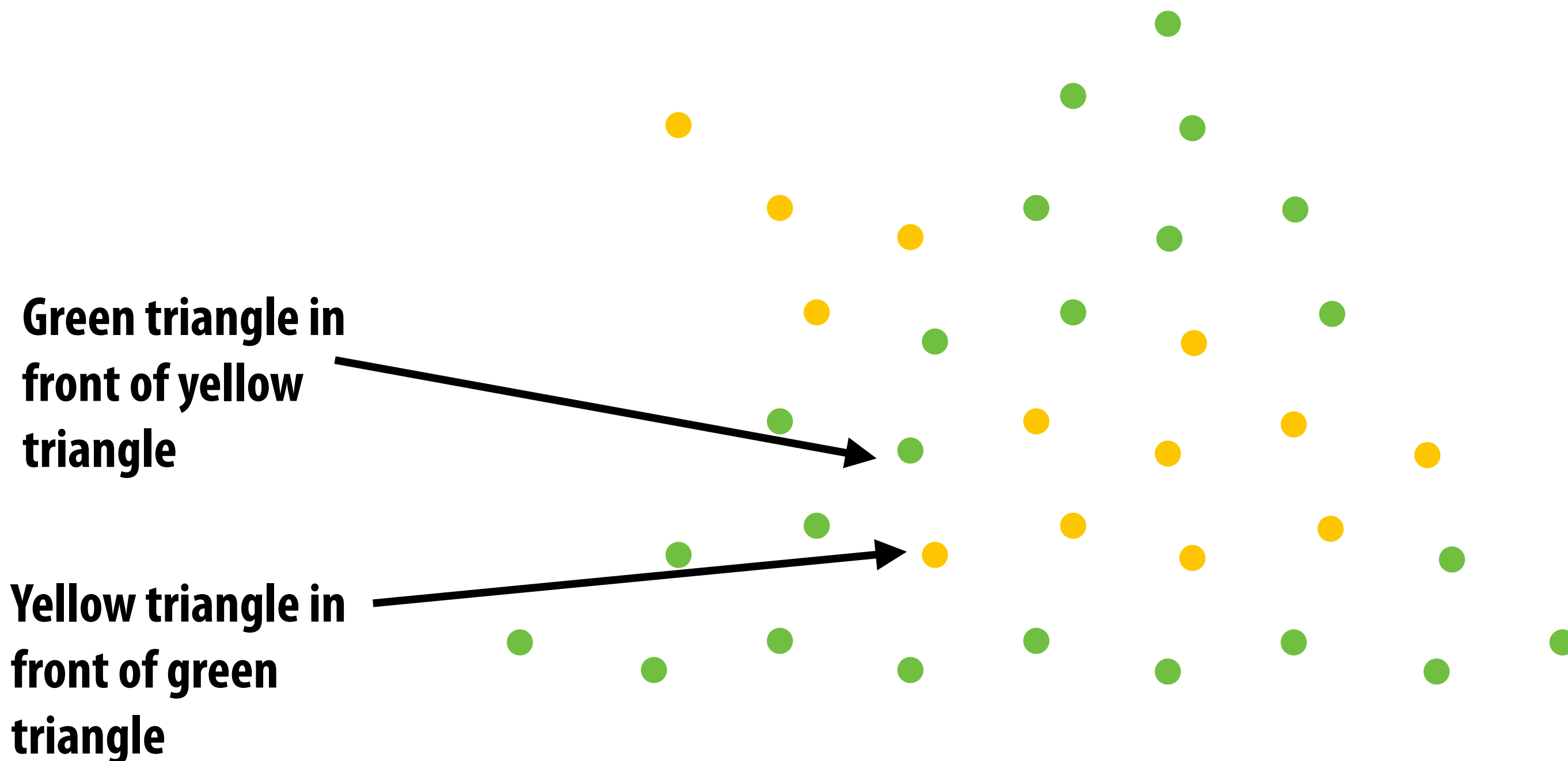
Intersection

Q: Does depth-buffer algorithm handle interpenetrating surfaces?

A: Of course!

Occlusion test is based on depth of triangles at a given sample point.

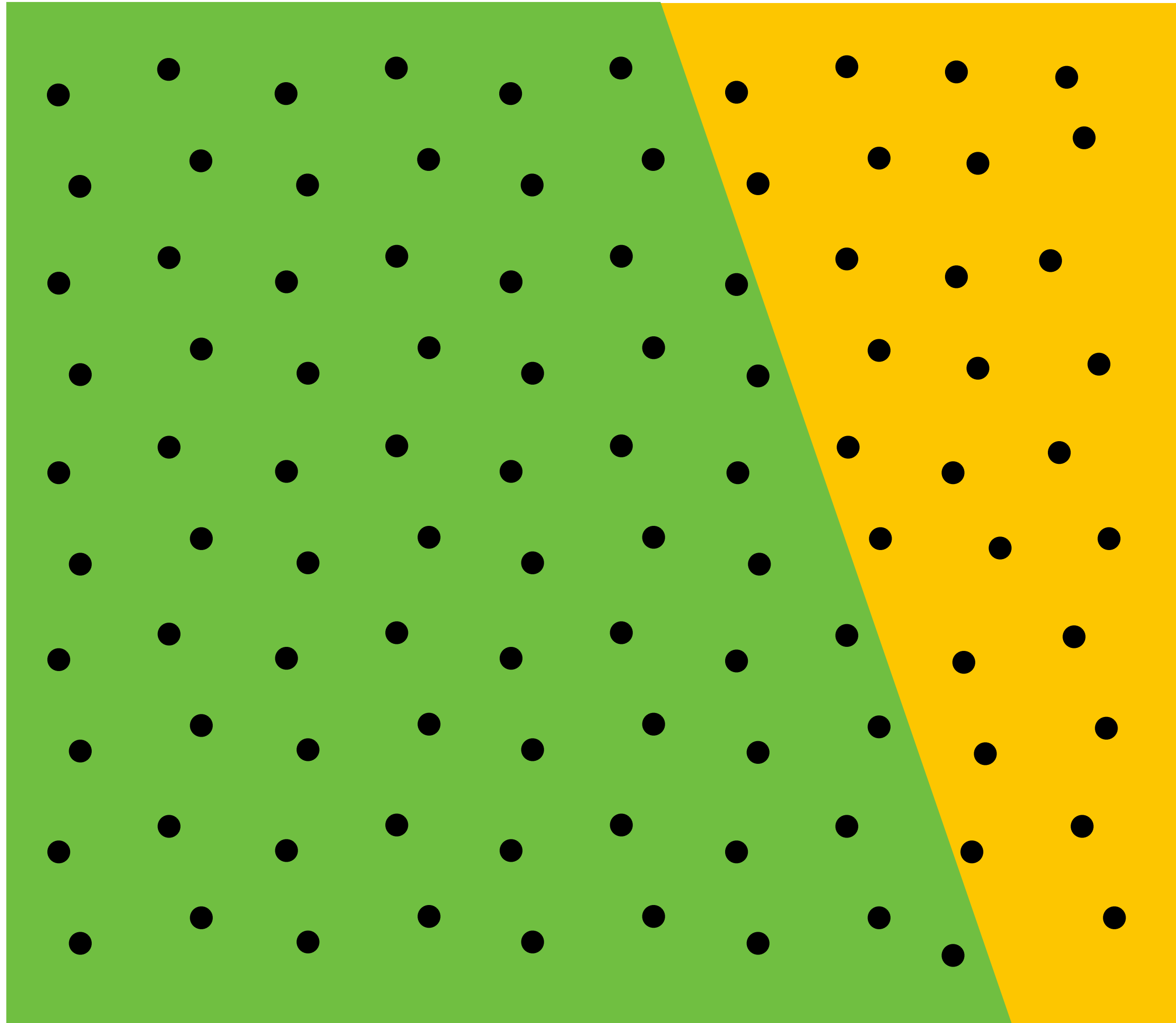
Relative depth of triangles may be different at different sample points.



Depth + Supersampling

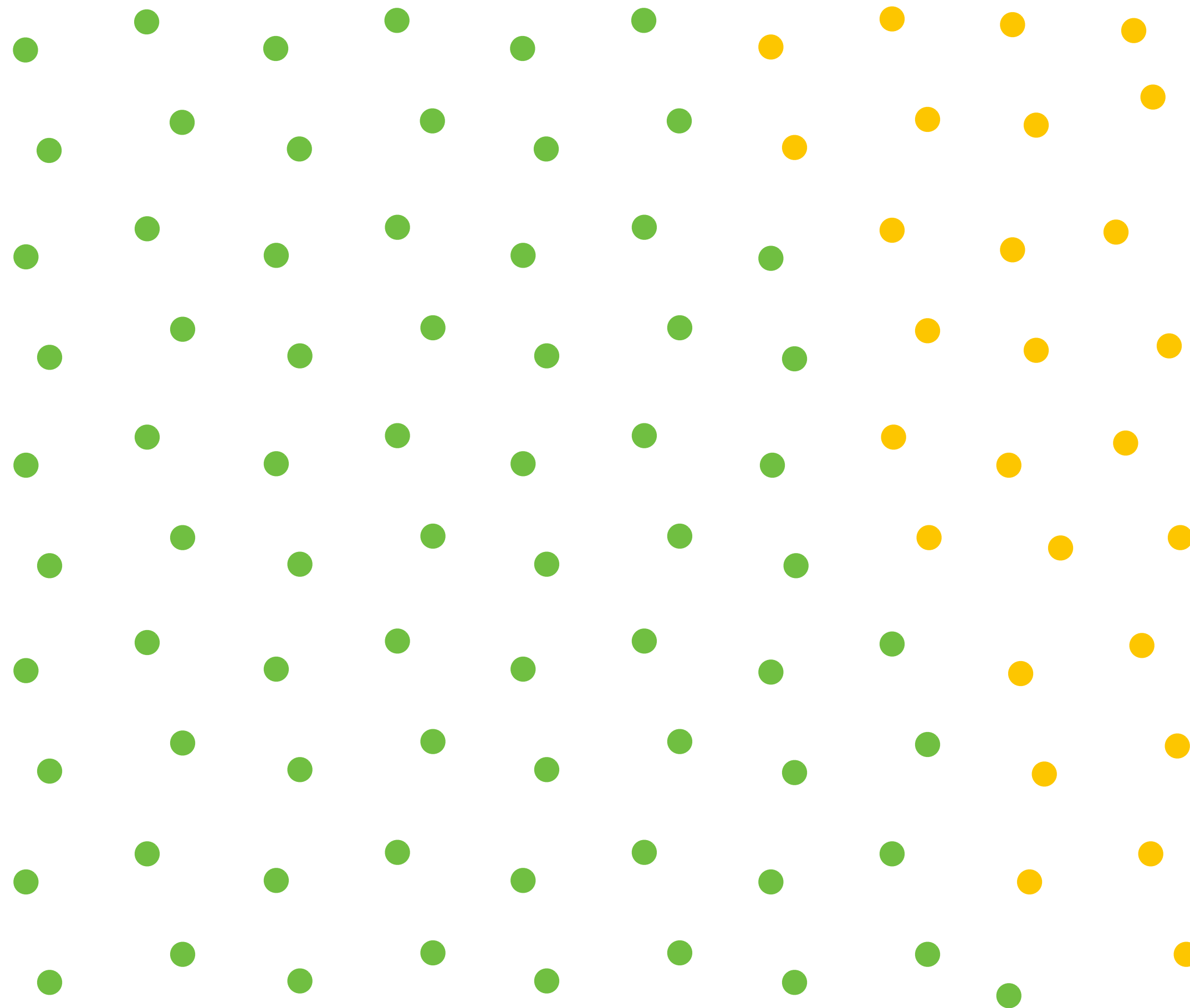
Q: Does depth buffer work with super sampling?

A: Yes! If done per (super) sample.



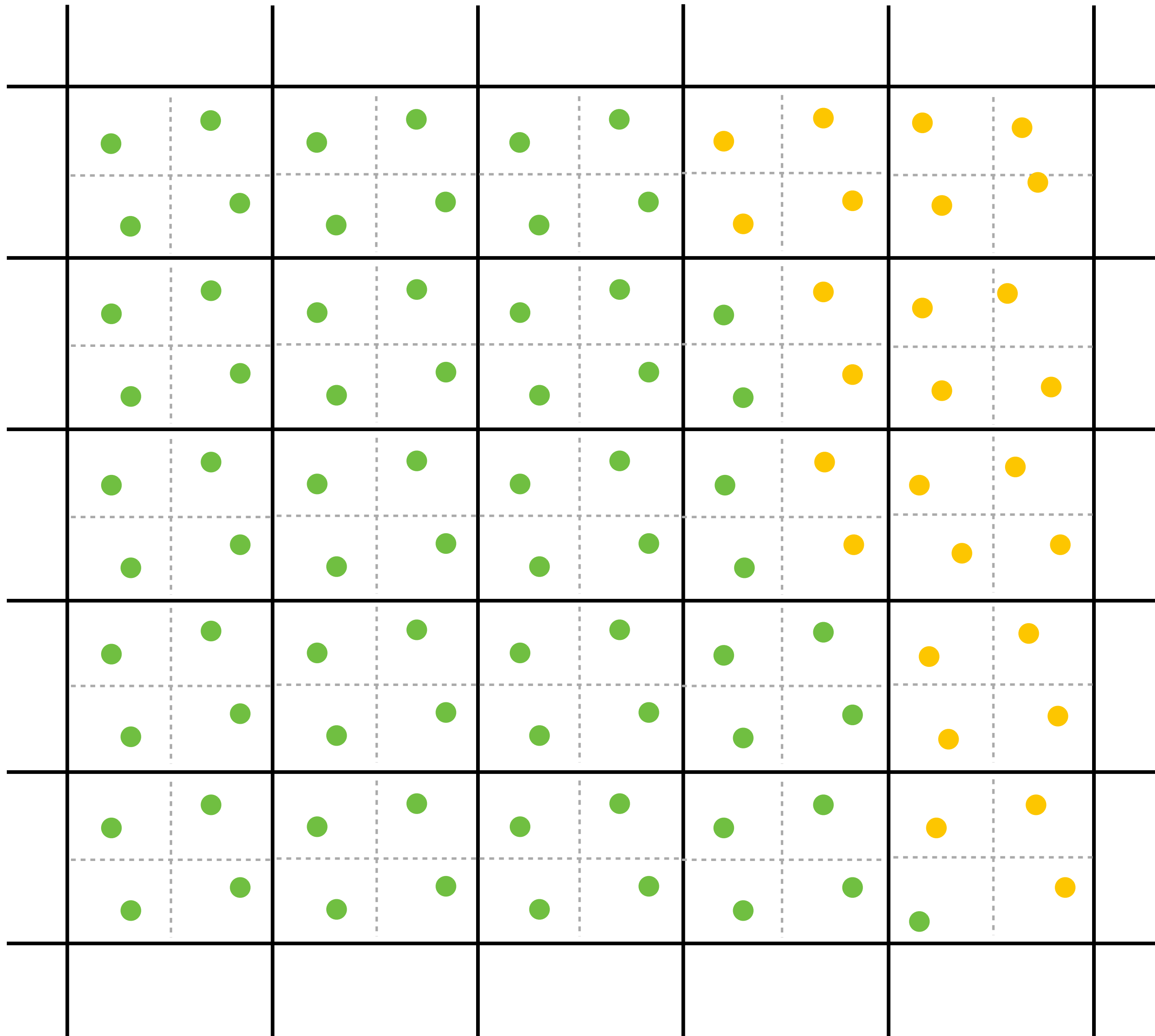
(Here: green triangle occludes yellow triangle)

Depth + Supersampling

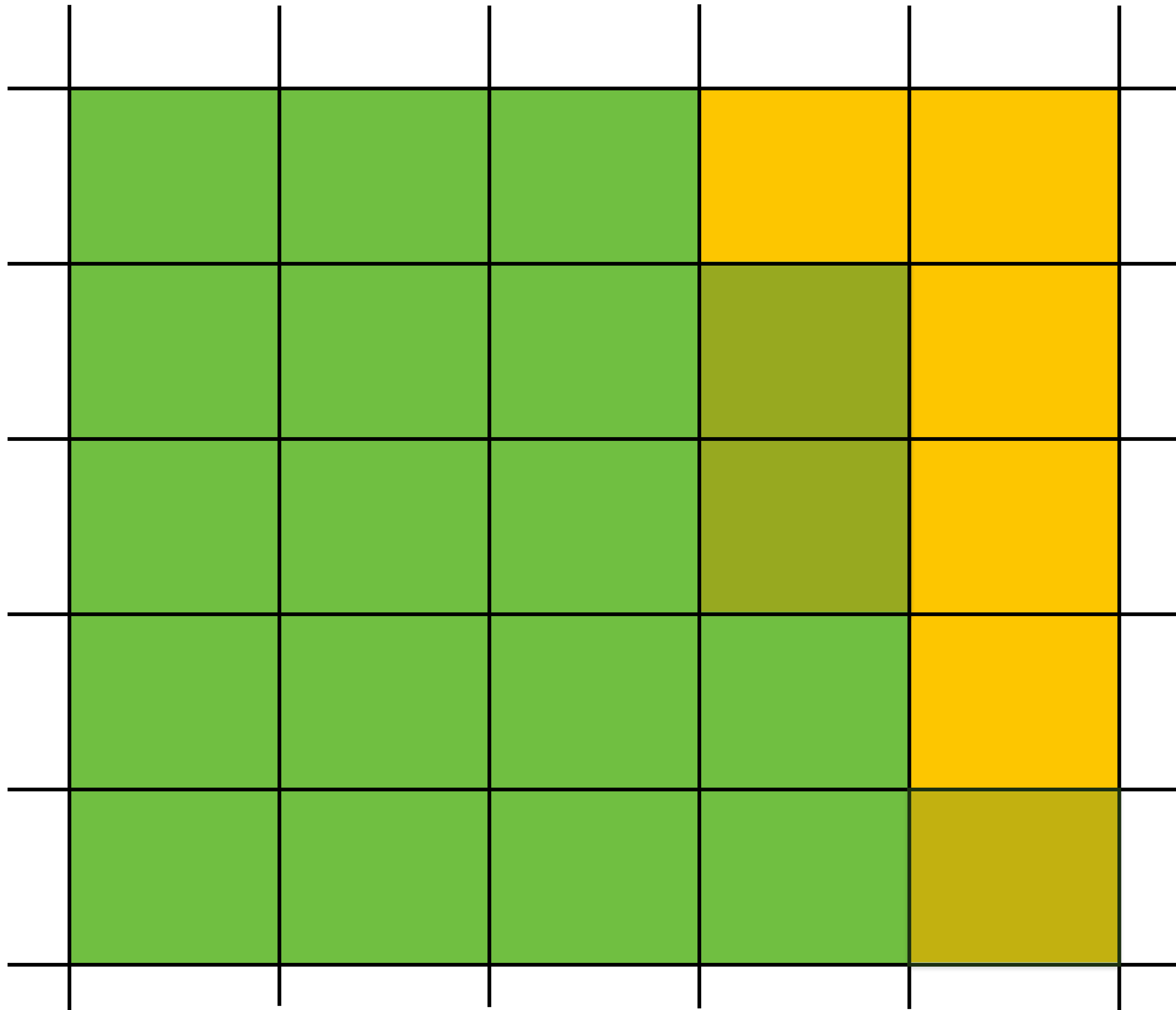


Color of super samples after rasterizing w/ depth buffer

Color buffer contents (4 samples per pixel)



Final resampled result



Note anti-aliasing of edge due to filtering of green and yellow samples

Summary: occlusion using a depth buffer

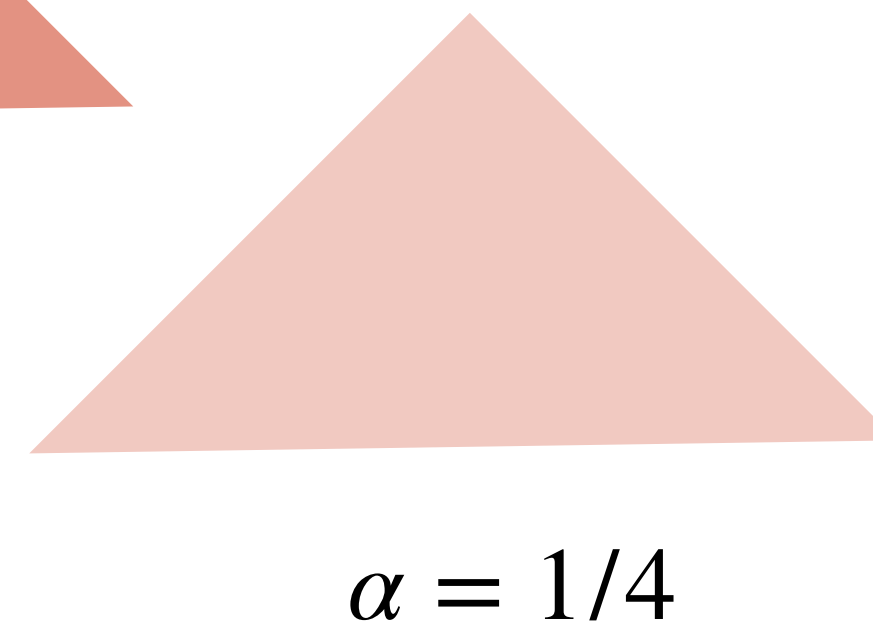
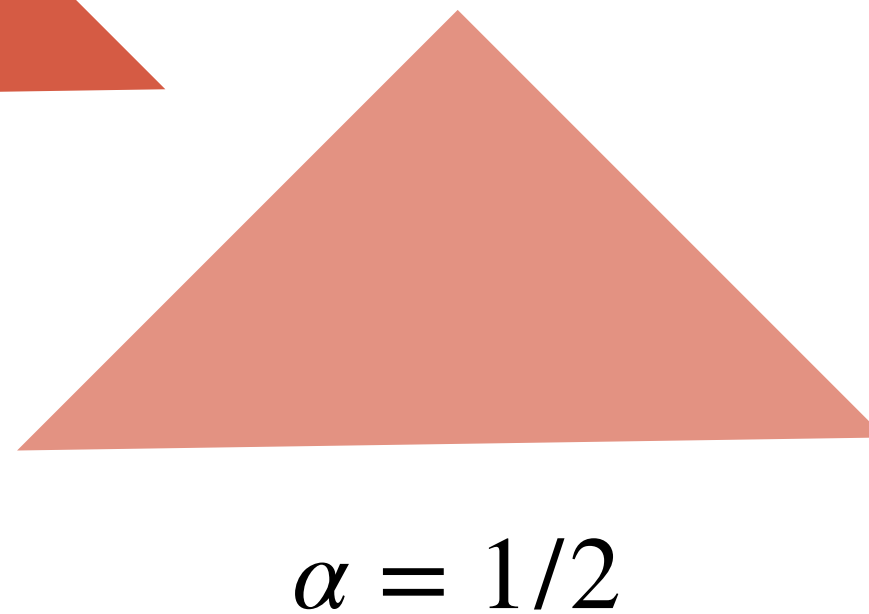
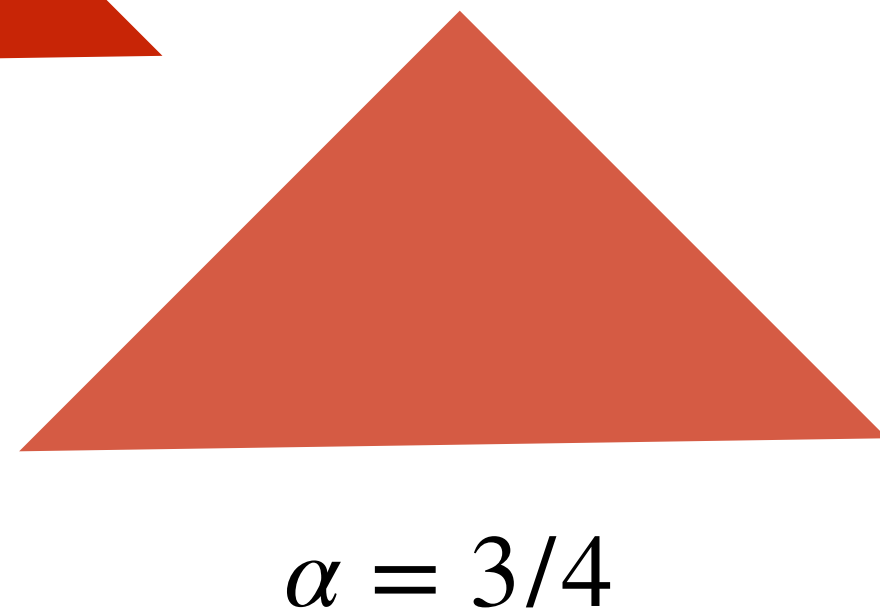
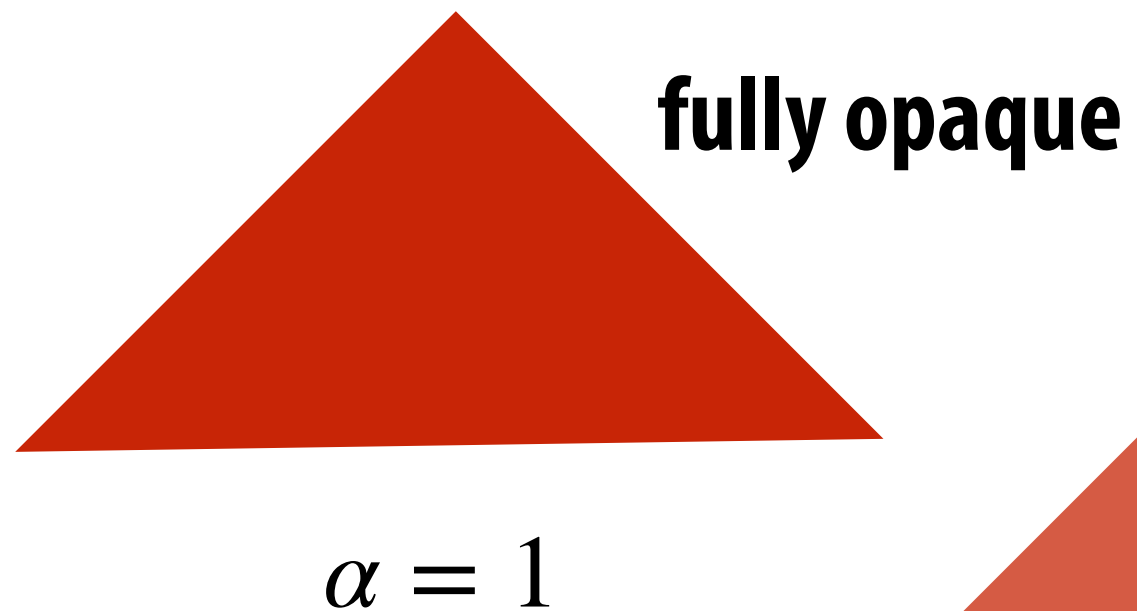
- Store one depth value per (super) sample—not one per pixel!
- Constant additional space per sample
 - Hence, **constant space for depth buffer**
 - **Doesn't** depend on number of overlapping primitives!
- Constant time occlusion test per covered sample
 - Read-modify write of depth buffer if “pass” depth test
 - Just a read if “fail”
- Not specific to triangles: only requires that surface depth can be evaluated at a screen sample point

But what about semi-transparent surfaces?

Compositing

Representing opacity as alpha

An “alpha” value $0 \leq \alpha \leq 1$ describes the opacity of an object

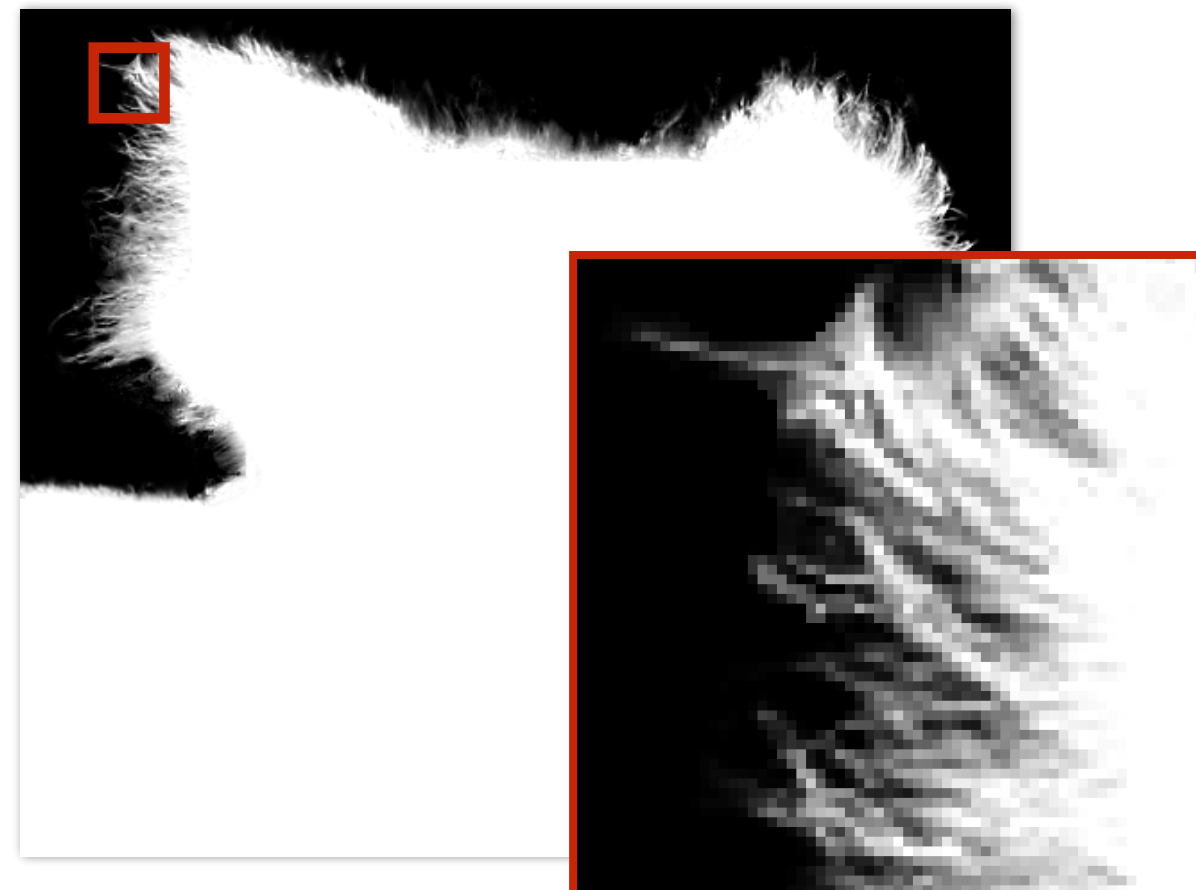


Alpha channel of an image

color channels



α channel



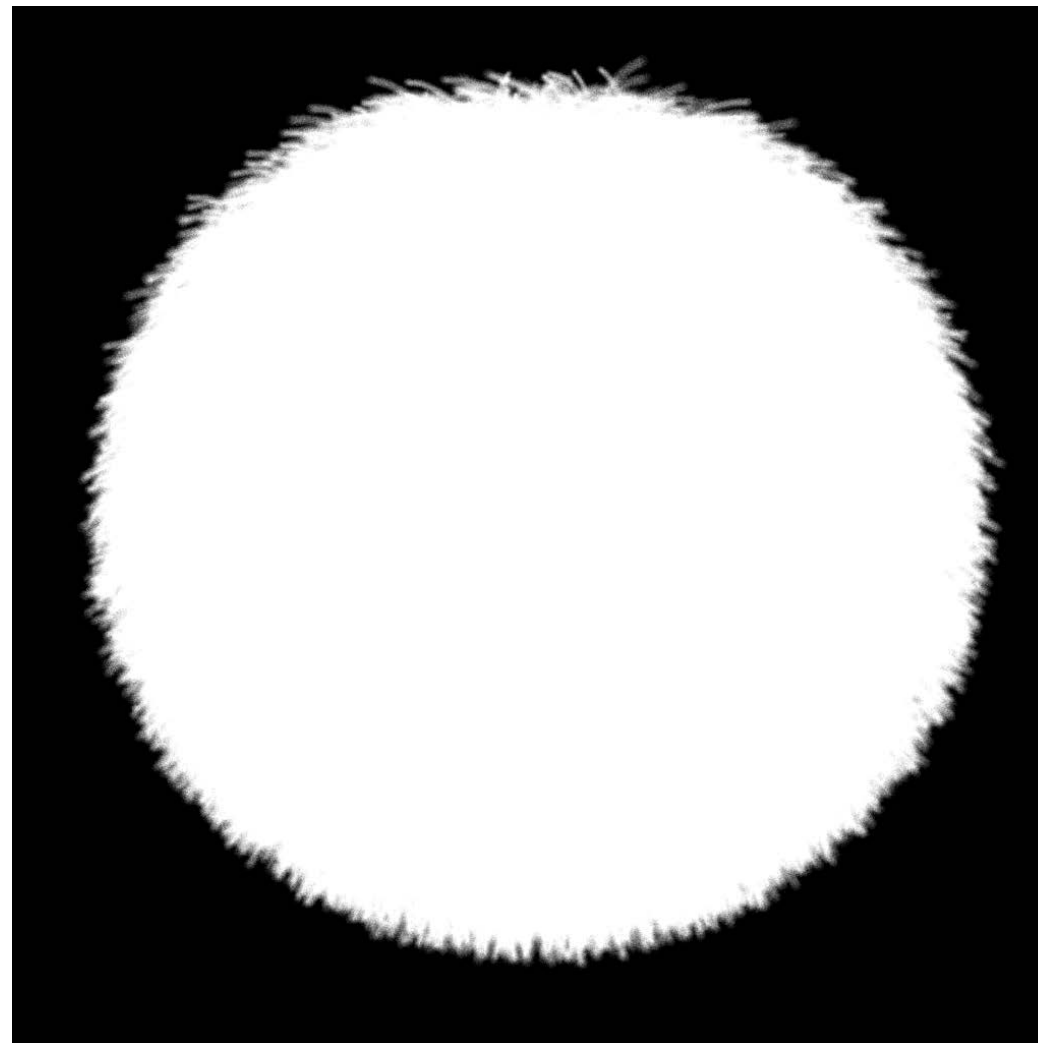
Key idea: can use α channel to composite one image on top of another.

Fringing

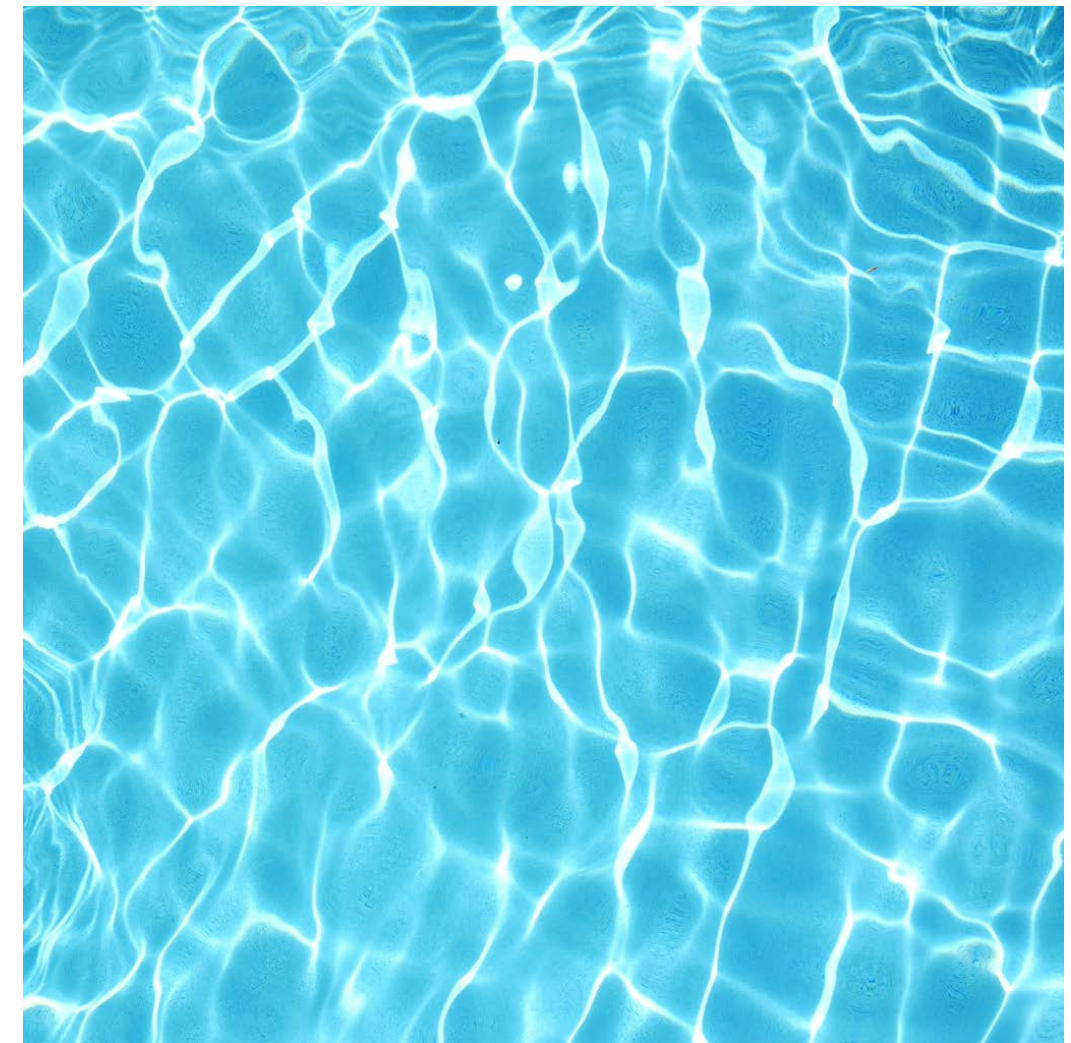
Poor treatment of color/alpha can yield dark “fringing”:



foreground color



foreground alpha



background color

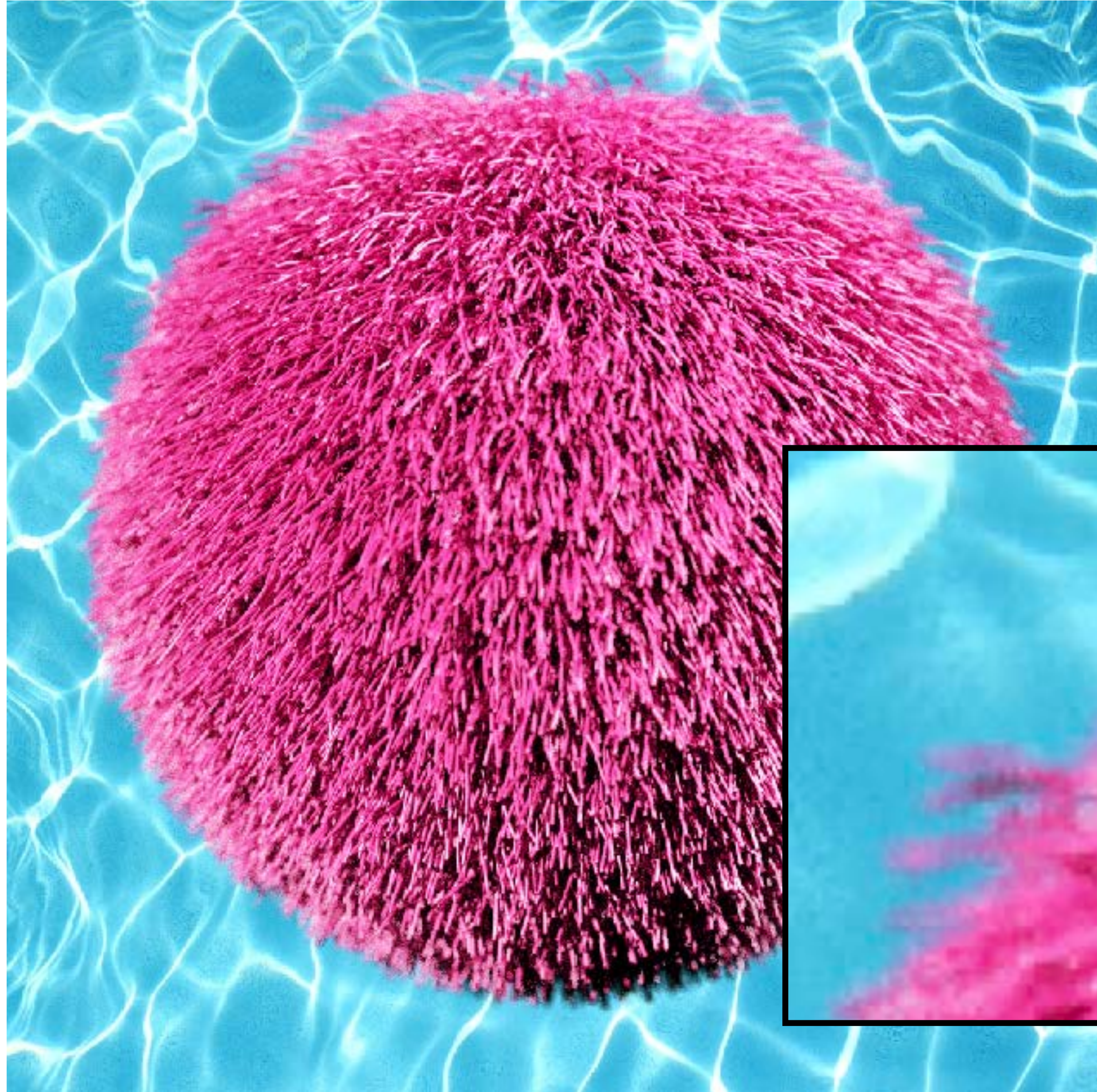


fringing

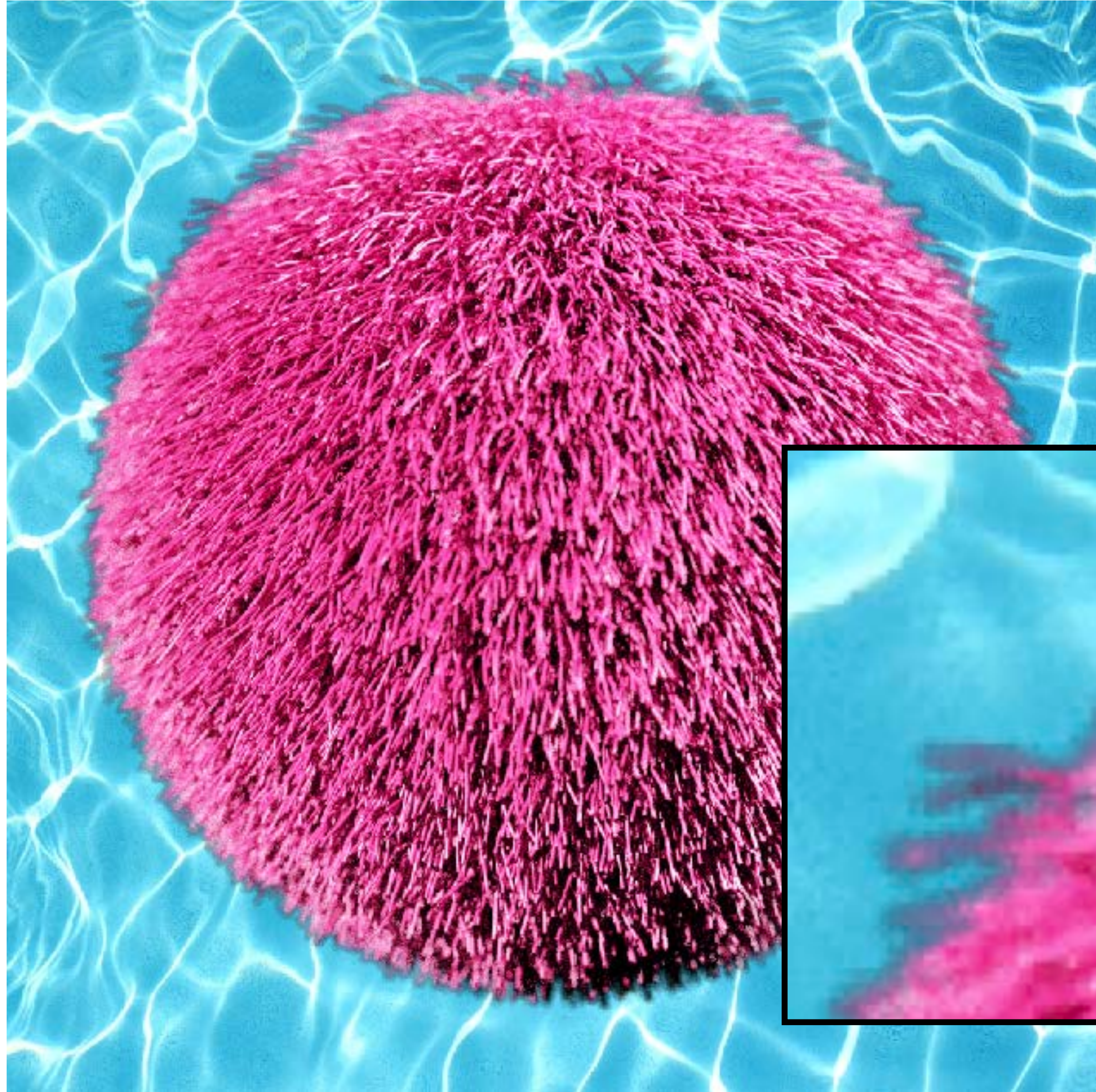


no fringing

No fringing



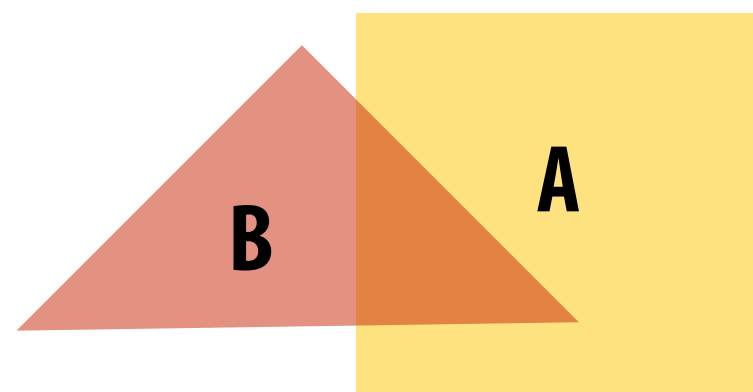
Fringing (...why does this happen?)



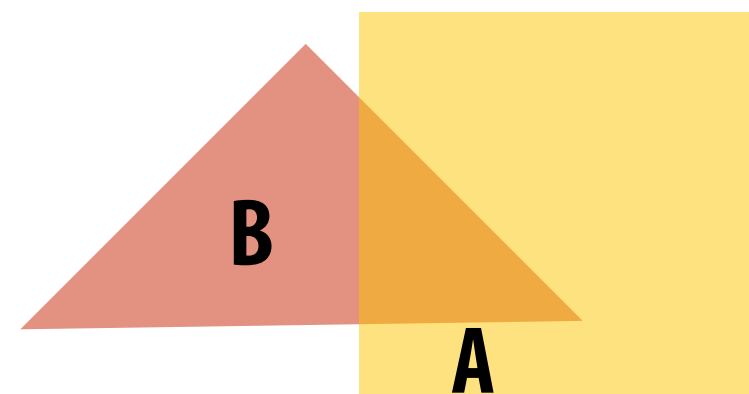
Over operator:

Composites image B with opacity α_B over image A with opacity α_A

Informally, captures behavior of “tinted glass”



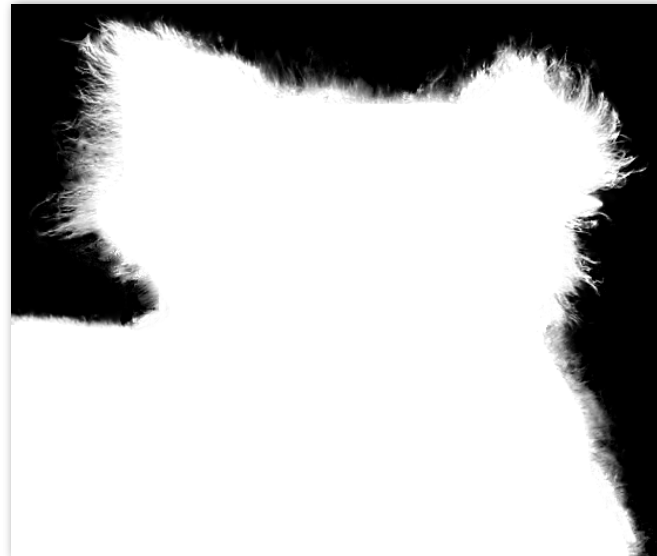
B over A



A over B

Notice: “over” is not commutative

$$A \text{ over } B \neq B \text{ over } A$$



Koala



NYC



Koala over NYC

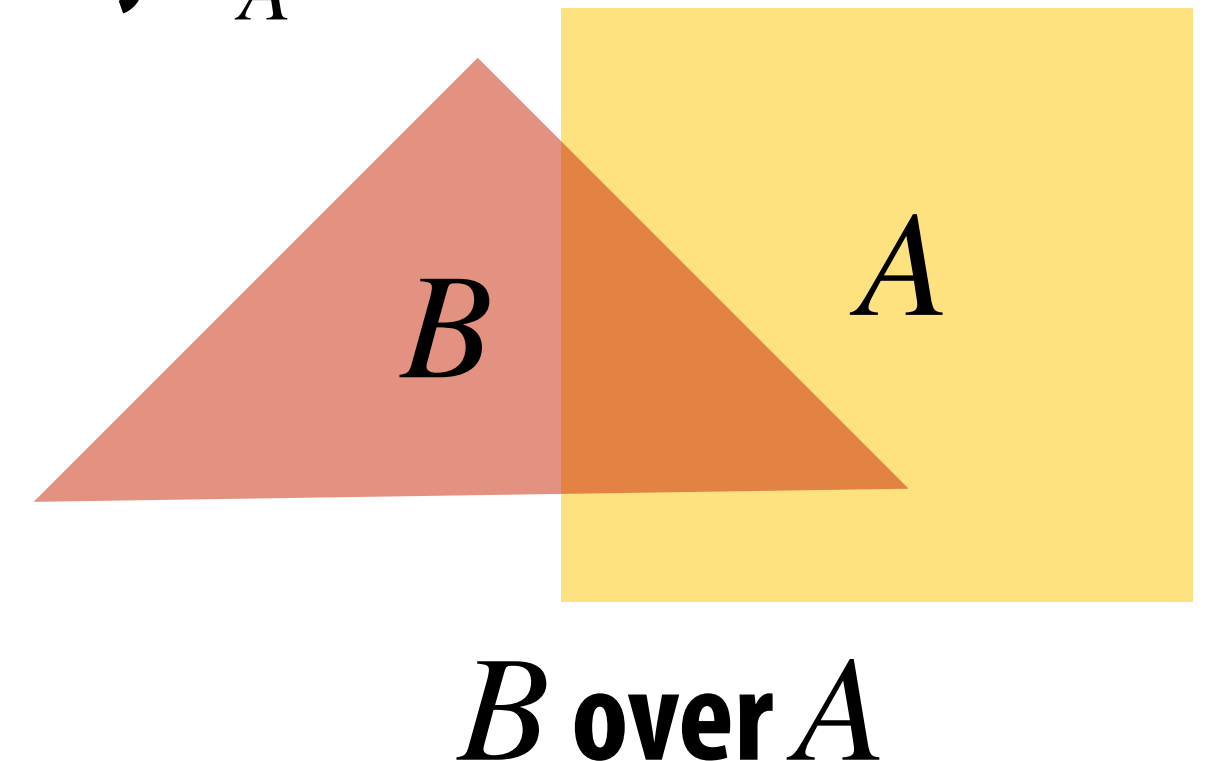
Over operator: non-premultiplied alpha

Composite image B with opacity α_B over image A with opacity α_A

A first attempt:

$$A = (A_r, A_g, A_b)$$

$$B = (B_r, B_g, B_b)$$



Composite color:

$$C = \alpha_B B + (1 - \alpha_B) \alpha_A A$$

↑
appearance of
semi-transparent B

↑
appearance of semi-
transparent A

what B lets through
↓

Composite alpha:

$$\alpha_C = \alpha_B + (1 - \alpha_B) \alpha_A$$

Over operator: premultiplied alpha

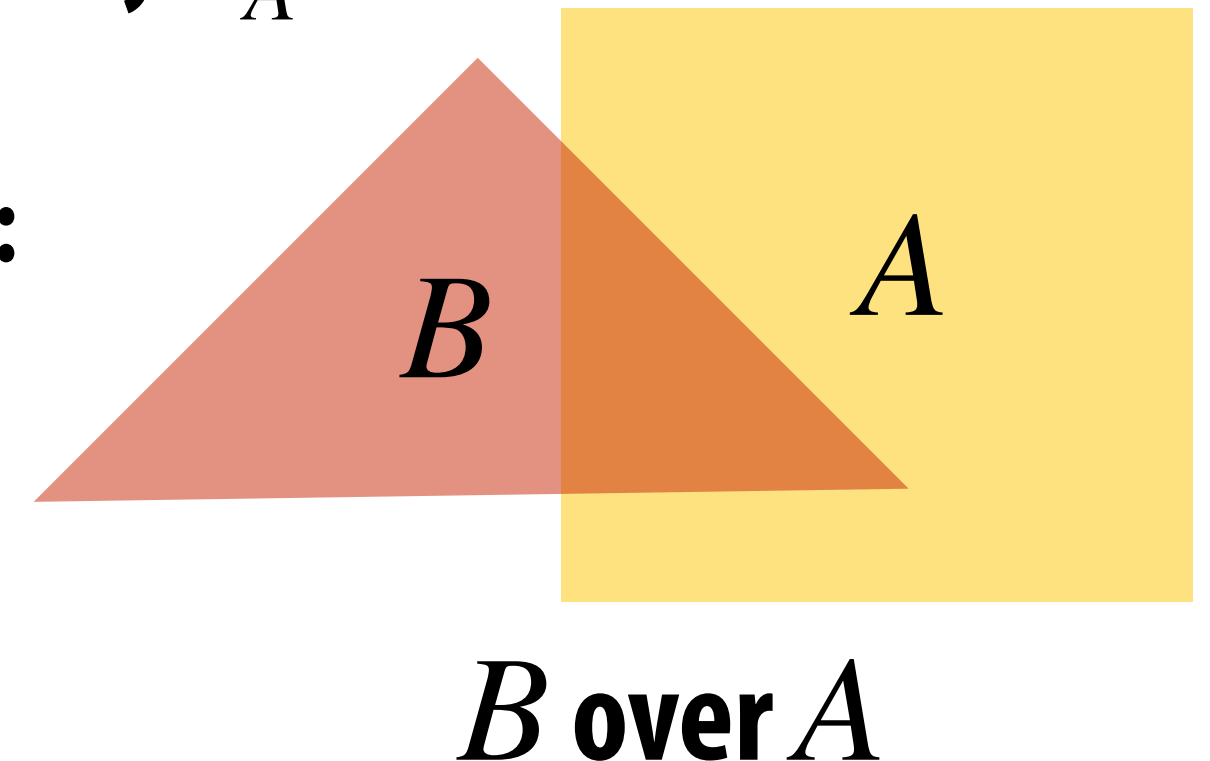
Composite image B with opacity α_B over image A with opacity α_A

Premultiplied alpha—multiply color by α , then composite:

$$A' = (\alpha_A A_r, \alpha_A A_g, \alpha_A A_b, \alpha_A)$$

$$B' = (\alpha_B B_r, \alpha_B B_g, \alpha_B B_b, \alpha_B)$$

$$C' = B' + (1 - \alpha_B)A'$$



Notice premultiplied alpha composites alpha just like how it composites rgb.
(Non-premultiplied alpha composites alpha differently than rgb.)

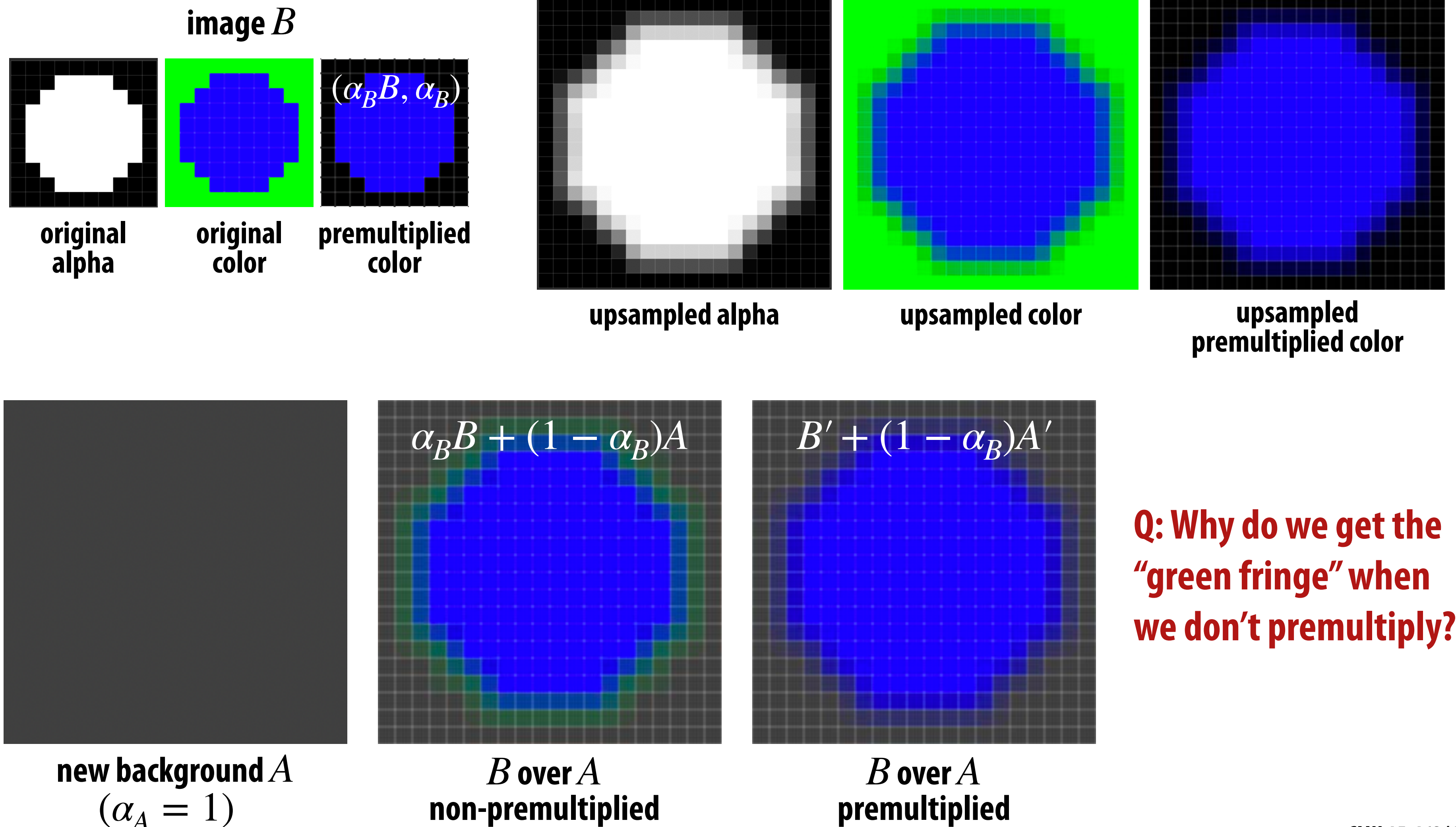
“Un-premultiply” to get final color:

$$(C_r, C_g, C_b, \alpha_C) \implies (C_r/\alpha_C, C_g/\alpha_C, C_b/\alpha_C)$$

Q: Does this division remind you of anything?

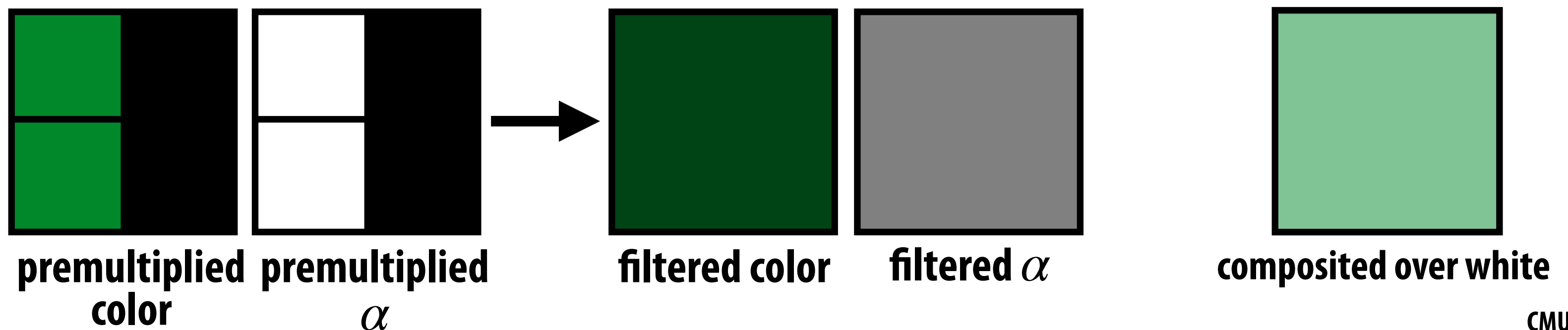
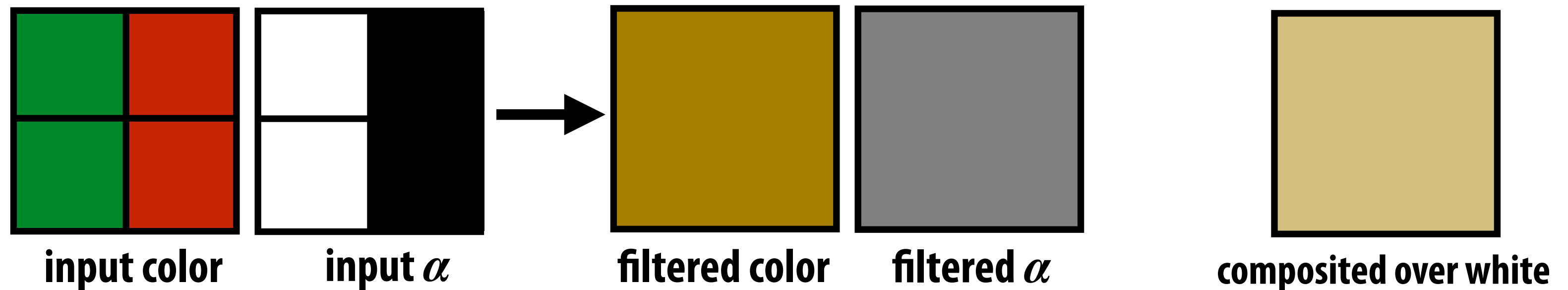
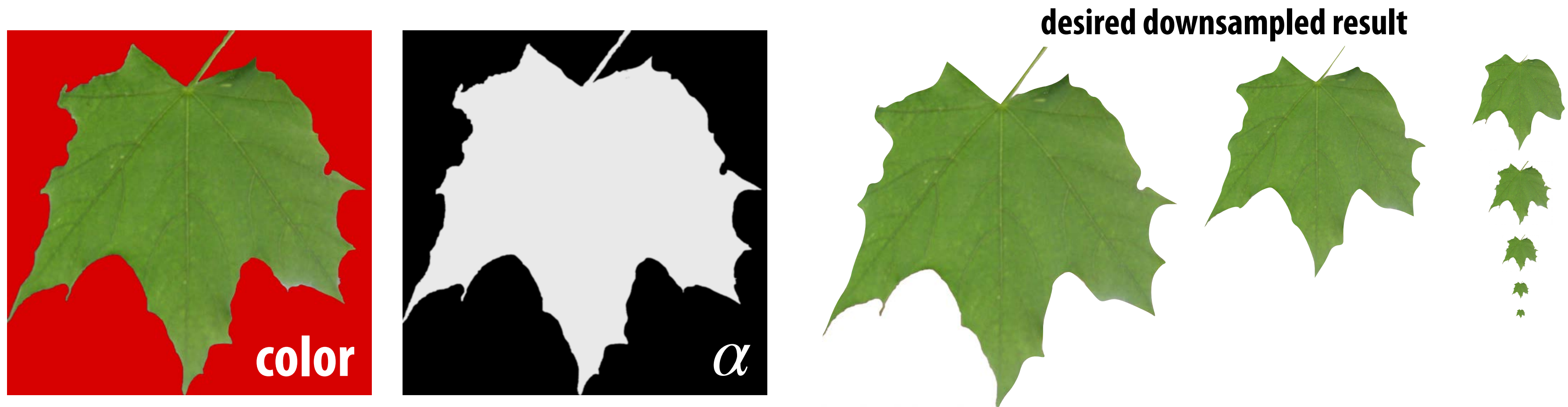
Compositing with & without premultiplied α

Suppose we upsample an image w/ an α channel, then composite it onto a background:



Similar problem with non-premultiplied α

Consider pre-filtering (downsampling) a texture with an alpha matte

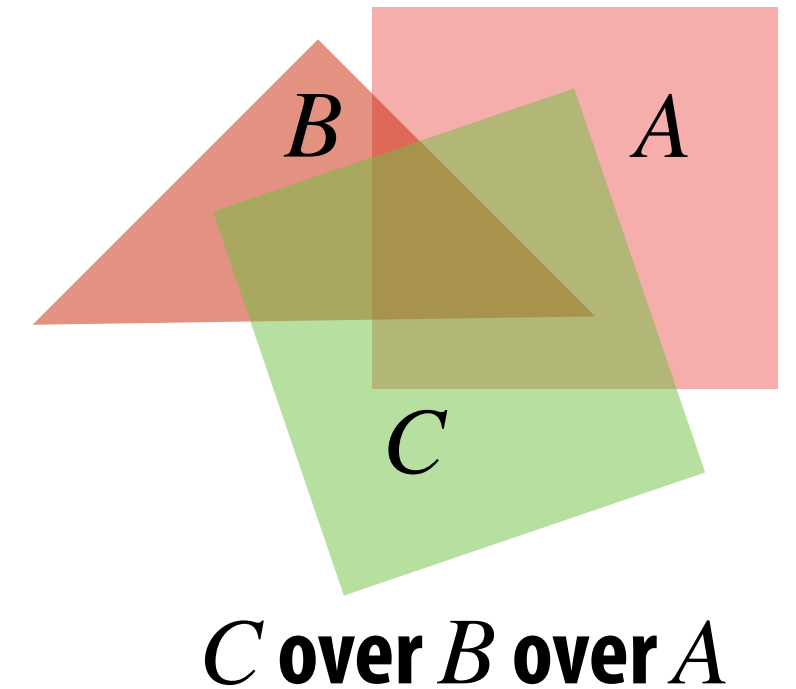


More problems: applying “over” repeatedly

Composite image C with opacity α_C over B with opacity α_B over image A with opacity α_A

**Premultiplied alpha is closed under composition;
non-premultiplied alpha is not!**

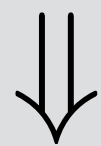
Example: composite 50% bright red over 50% bright red
(where “bright red” = $(1,0,0)$, and $\alpha = 0.5$)



non-premultiplied

color

$$.5(1,0,0) + (1-.5).5(1,0,0)$$



$$(0.75,0,0) \text{ too dark!}$$

alpha

$$.5 + (1-.5).5 = .75$$

premultiplied

color

$$(.5,0,0,.5) + (1-.5)(.5,0,0,.5)$$



$$(.75,0,0.75)$$



divide by α

$$\text{bright red } (1,0,0)$$

alpha

$$\alpha = 0.75$$

Summary: advantages of premultiplied alpha

- Compositing operation treats all channels the same (color and α)
- Fewer arithmetic operations for “over” operation than with non-premultiplied representation
- Closed under composition (repeated “over” operations)
- Better representation for filtering (upsampling/downsampling) images with alpha channel
- Fits naturally into rasterization pipeline (homogeneous coordinates)

Strategy for drawing semi-transparent primitives

Assuming all primitives are semi-transparent, and color values are encoded with premultiplied alpha, here's a strategy for rasterizing an image:

```
over(c1, c2)
{
    return c1.rgba + (1-c1.a) * c2.rgba;
}
```

```
update_color_buffer( x, y, sample_color, sample_depth )
{
    if (pass_depth_test(sample_depth, zbuffer[x][y])
    {
        // (how) should we update depth buffer here??
        color[x][y] = over(sample_color, color[x][y]);
    }
}
```

Q: What is the assumption made by this implementation?

Triangles must be rendered in back to front order!

Putting it all together

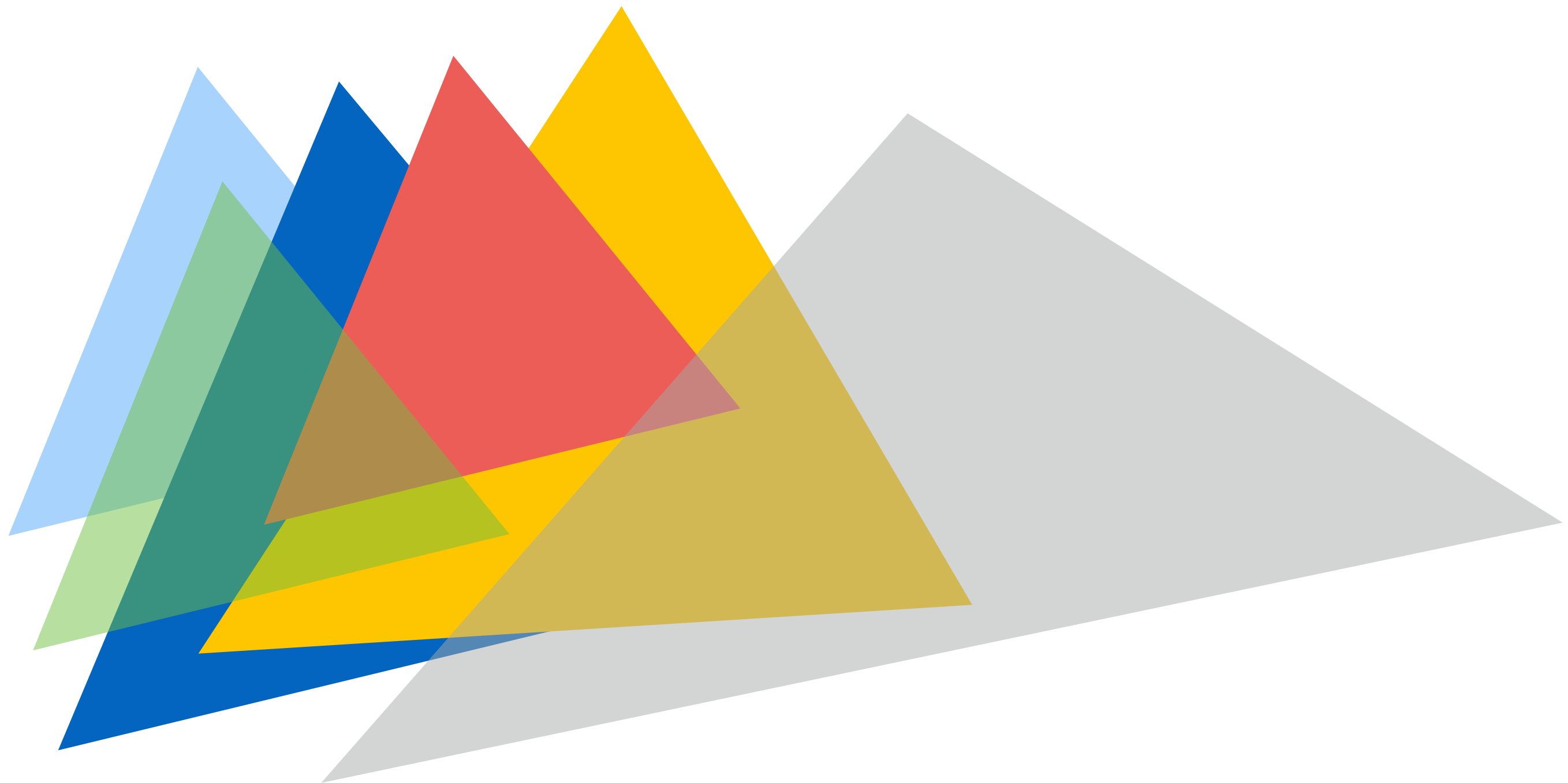
What if we have a mixture of opaque and transparent triangles?

Step 1: render opaque primitives (in any order) using depth-buffered occlusion

If pass depth test, triangle overwrites value in color buffer at sample

Step 2: disable depth buffer update, render semi-transparent surfaces in back-to-front order.

If pass depth test, triangle is composited OVER contents of color buffer at sample



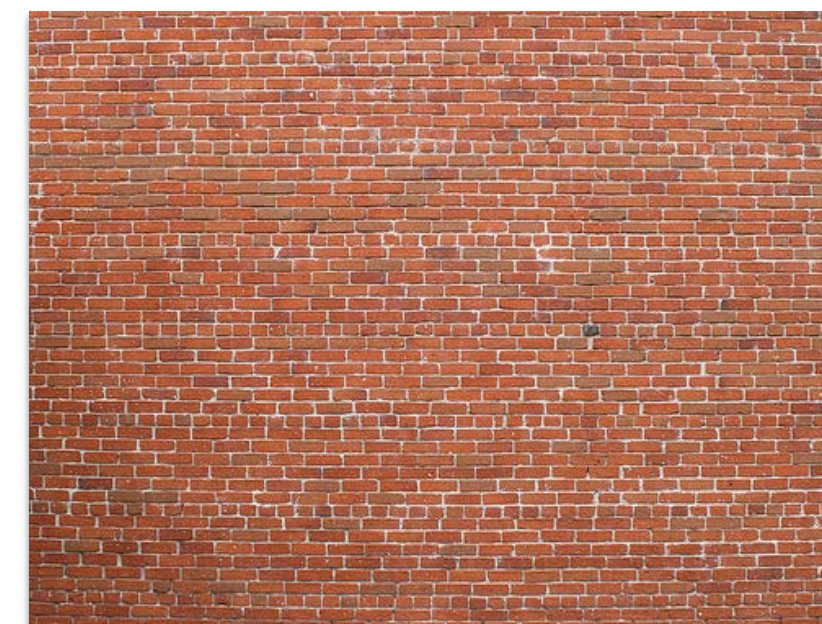
End-to-end rasterization pipeline

Goal: turn inputs into an image!

Inputs:

```
positions = {  
    v0x, v0y, v0z,  
    v1x, v1y, v1x,  
    v2x, v2y, v2z,  
    v3x, v3y, v3x,  
    v4x, v4y, v4z,  
    v5x, v5y, v5x  
};
```

```
texcoords = {  
    v0u, v0v,  
    v1u, v1v,  
    v2u, v2v,  
    v3u, v3v,  
    v4u, v4v,  
    v5u, v5v  
};
```



texture map

Object-to-camera-space transform $T \in \mathbb{R}^{4 \times 4}$

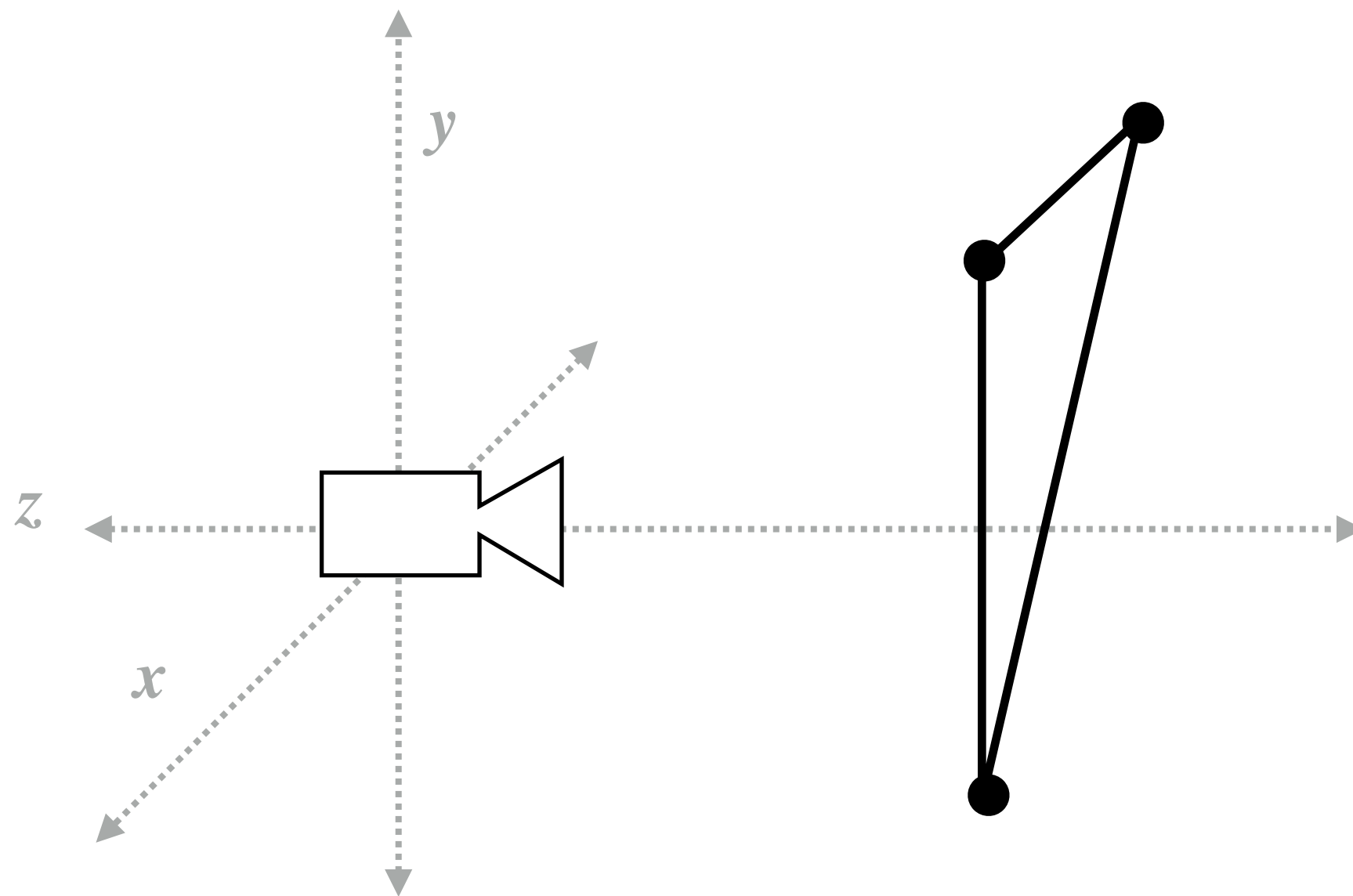
Perspective projection transform $P \in \mathbb{R}^{4 \times 4}$

Size of output image (W, H)

At this point we have all the tools we need to make an image...
Let's review!

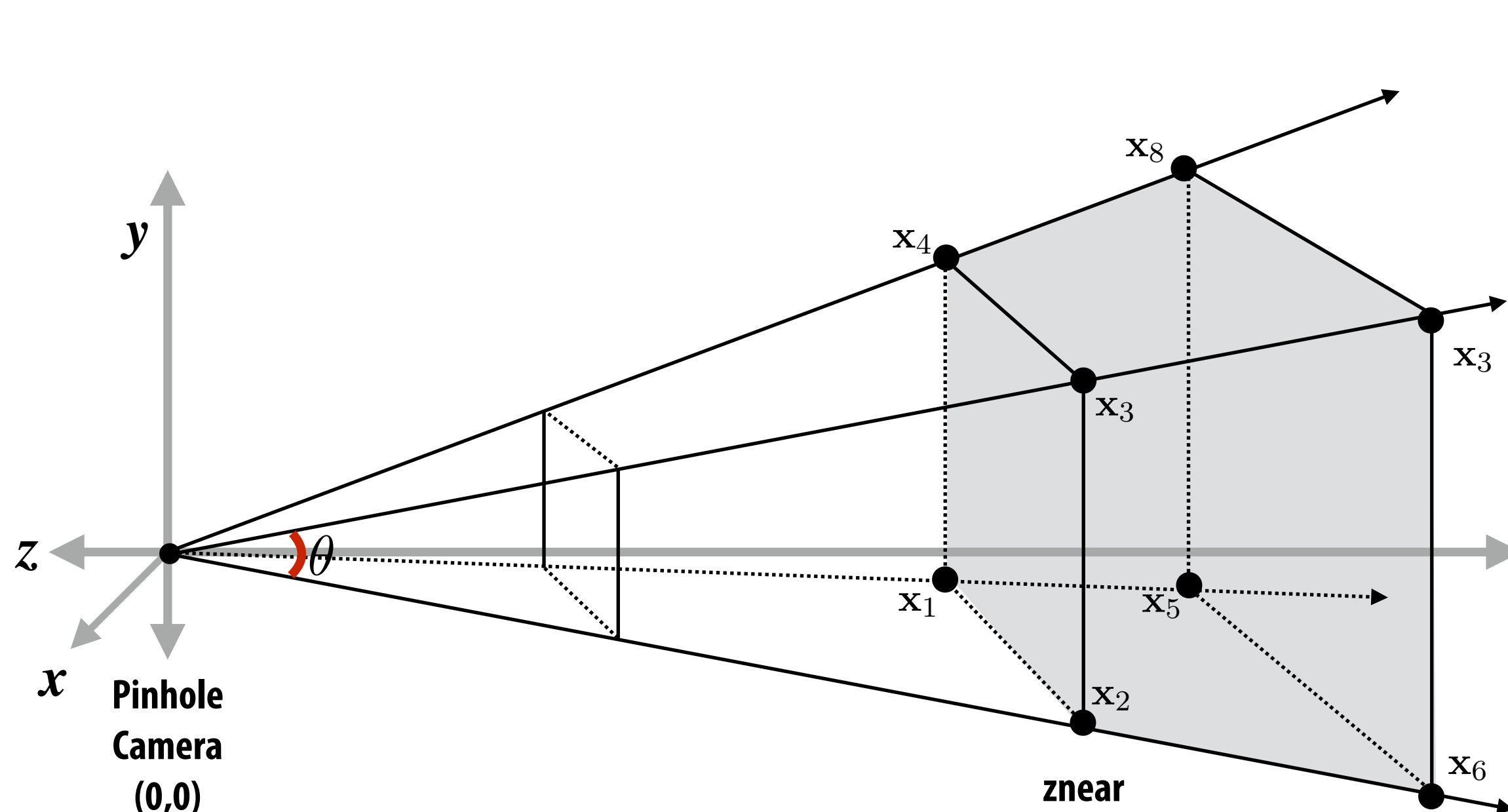
Step 1:

Transform triangle vertices into camera space

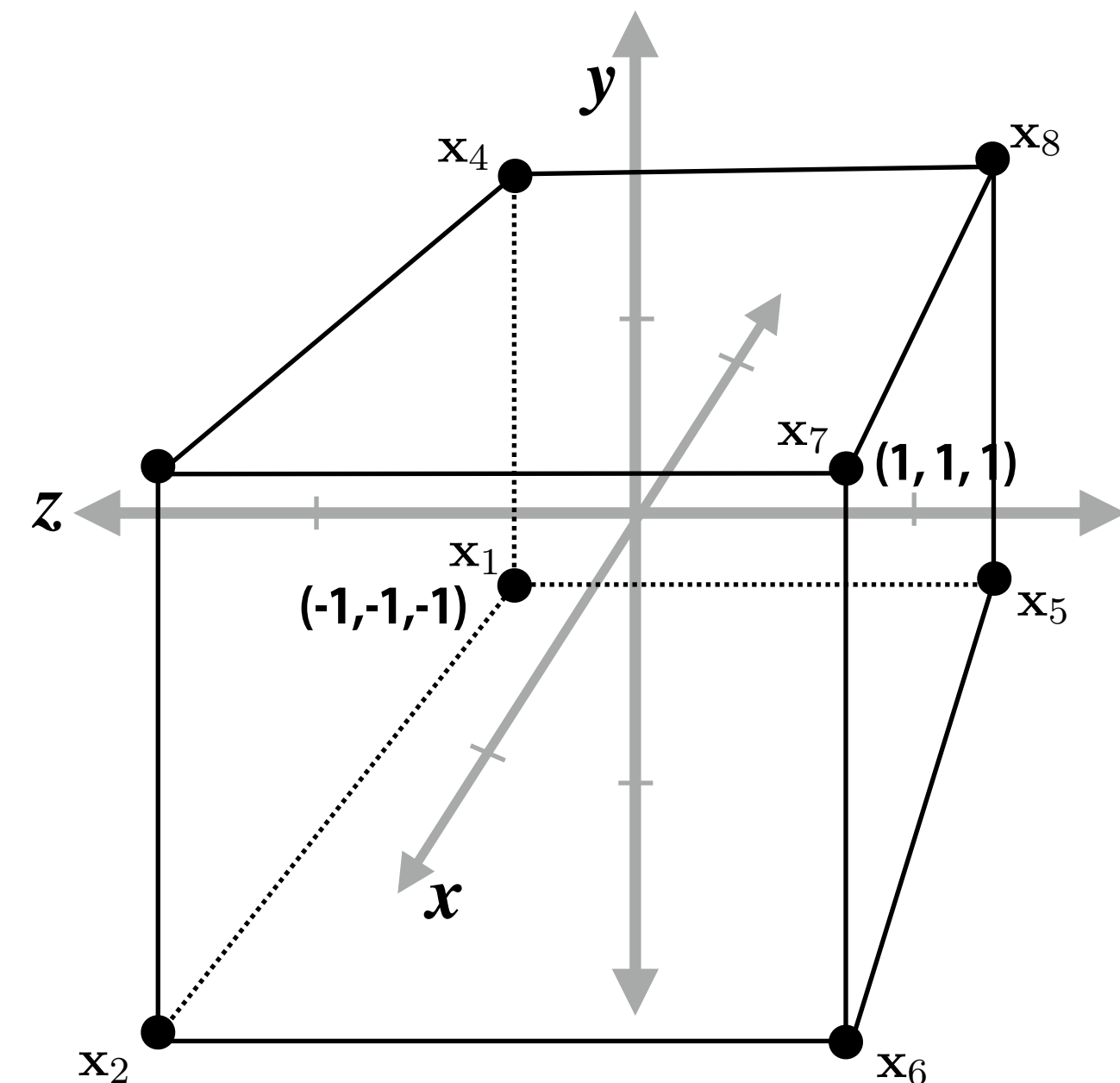


Step 2:

Apply perspective projection transform to transform triangle vertices into normalized coordinate space



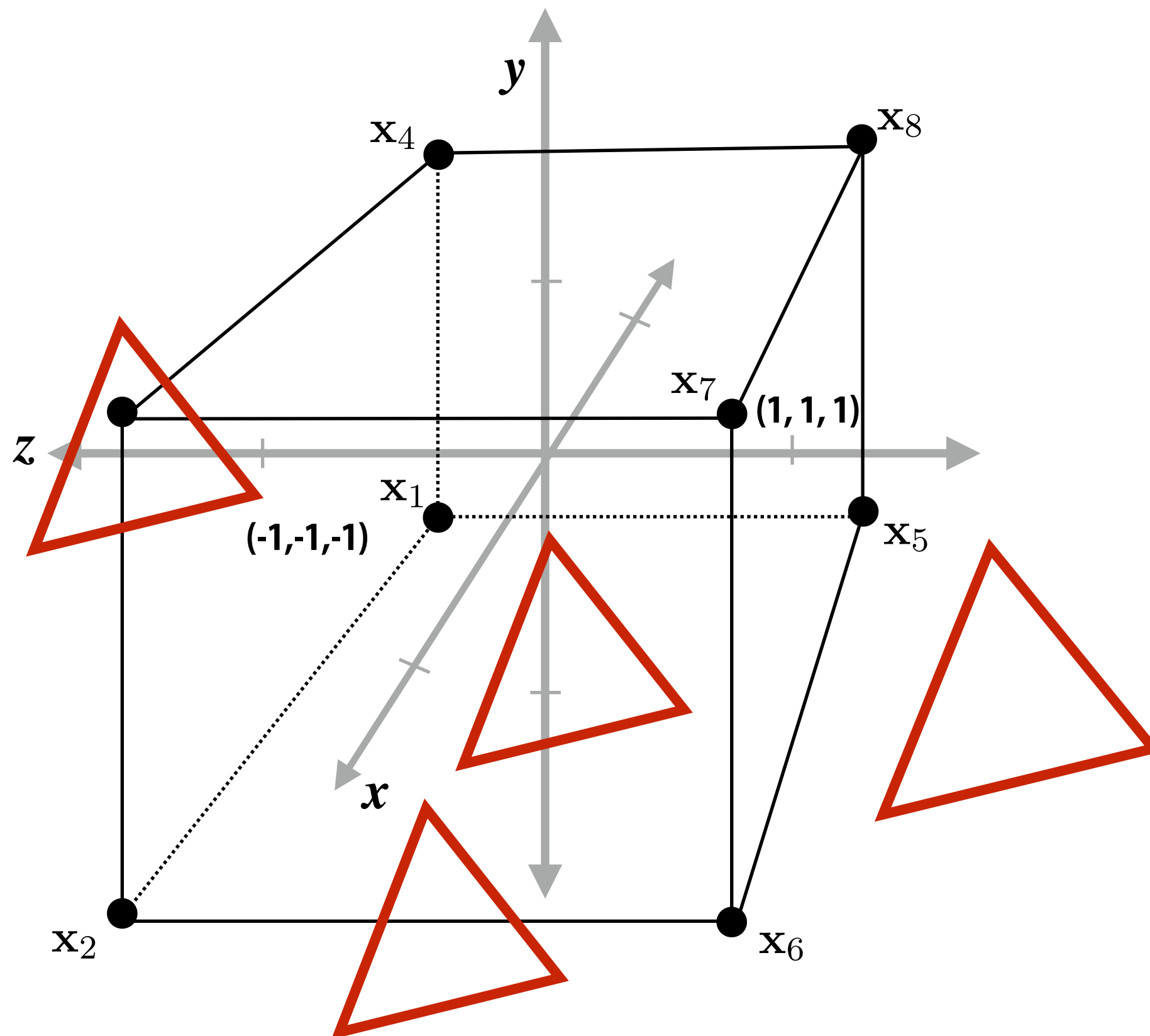
Camera-space positions: 3D



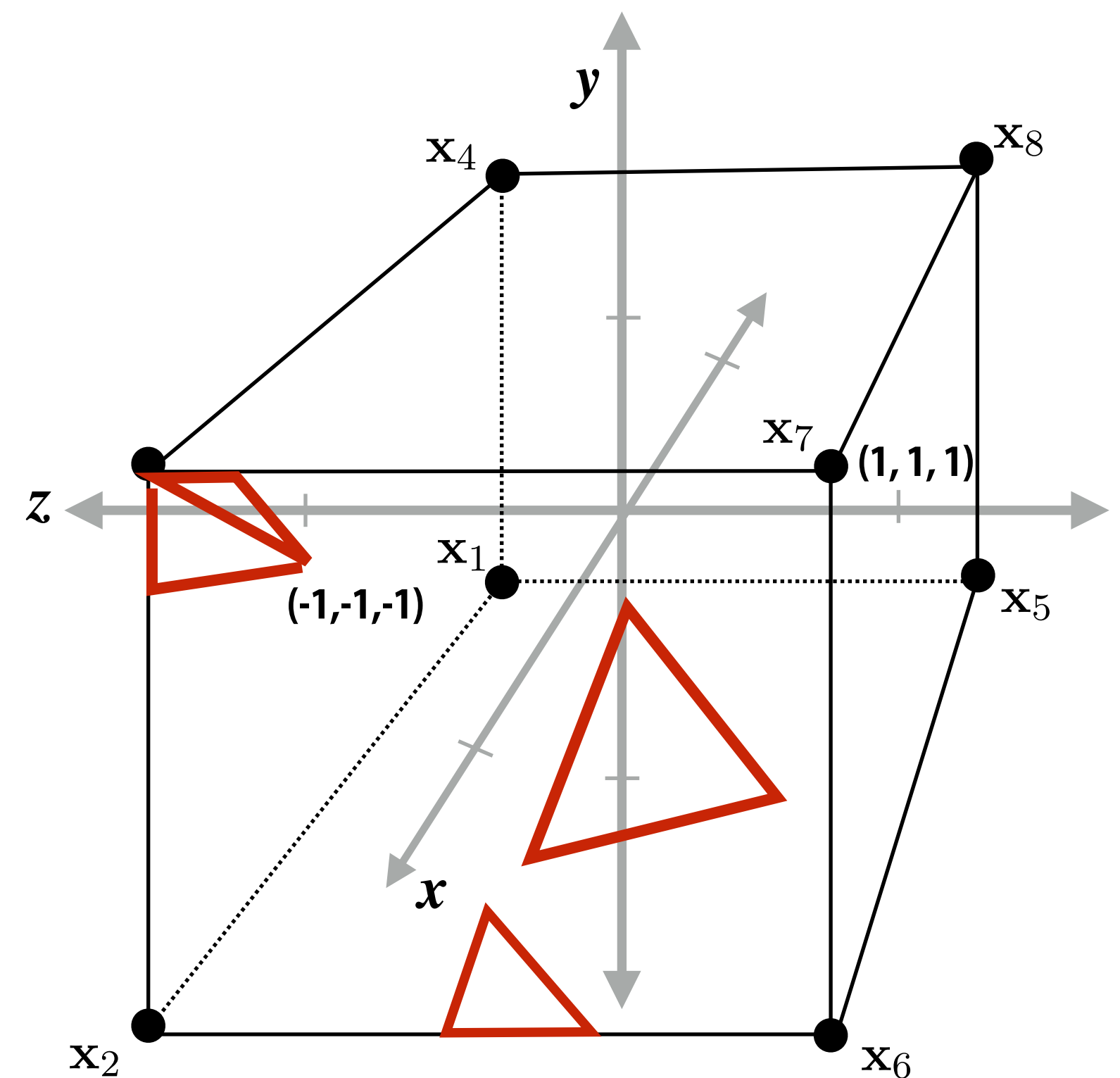
Normalized space positions

Step 3: clipping

- Discard triangles that lie complete outside the unit cube (culling)
 - They are off screen, don't bother processing them further
- Clip triangles that extend beyond the unit cube to the cube
 - (possibly generating new triangles)



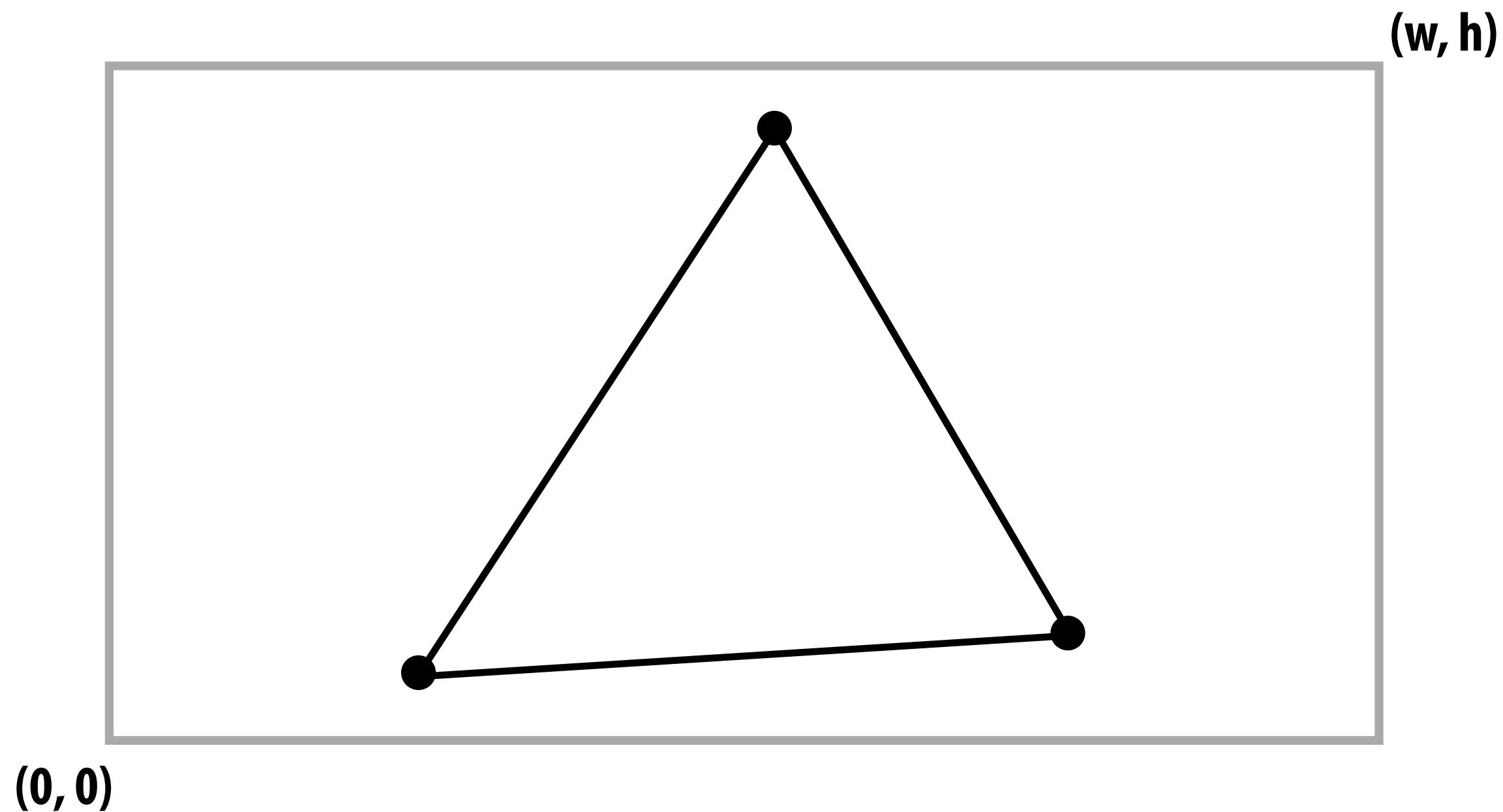
Triangles before clipping



Triangles after clipping

Step 4: transform to screen coordinates

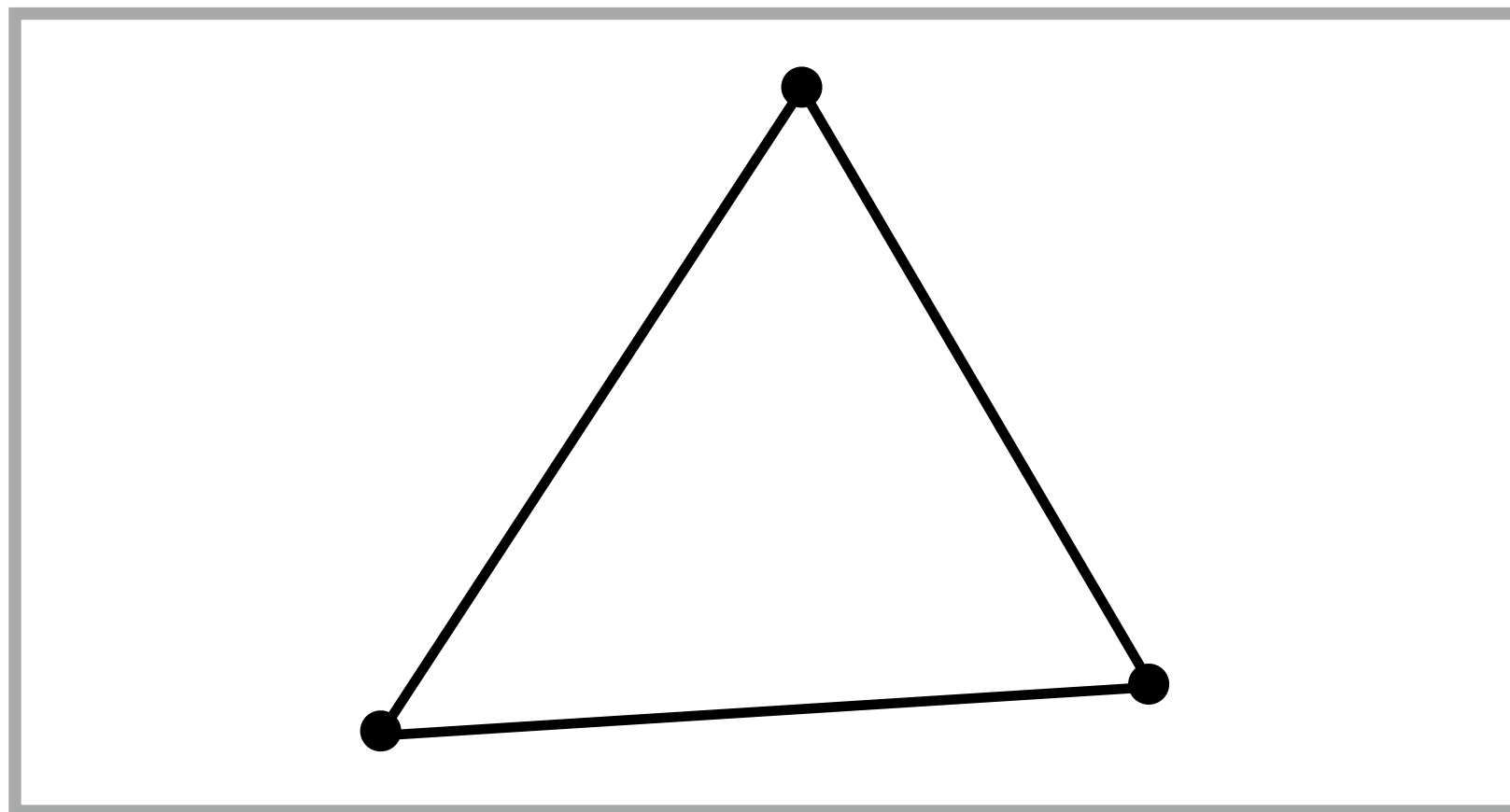
Perform homogeneous divide, transform vertex xy positions from normalized coordinates into screen coordinates (based on screen w,h)



Step 5: setup triangle (triangle preprocessing)

Before rasterizing triangle, can compute a bunch of data that will be used by all fragments, e.g.,

- triangle edge equations
- triangle attribute equations
- etc.



$$\mathbf{E}_{01}(x, y)$$

$$\mathbf{U}(x, y)$$

$$\mathbf{E}_{12}(x, y)$$

$$\mathbf{V}(x, y)$$

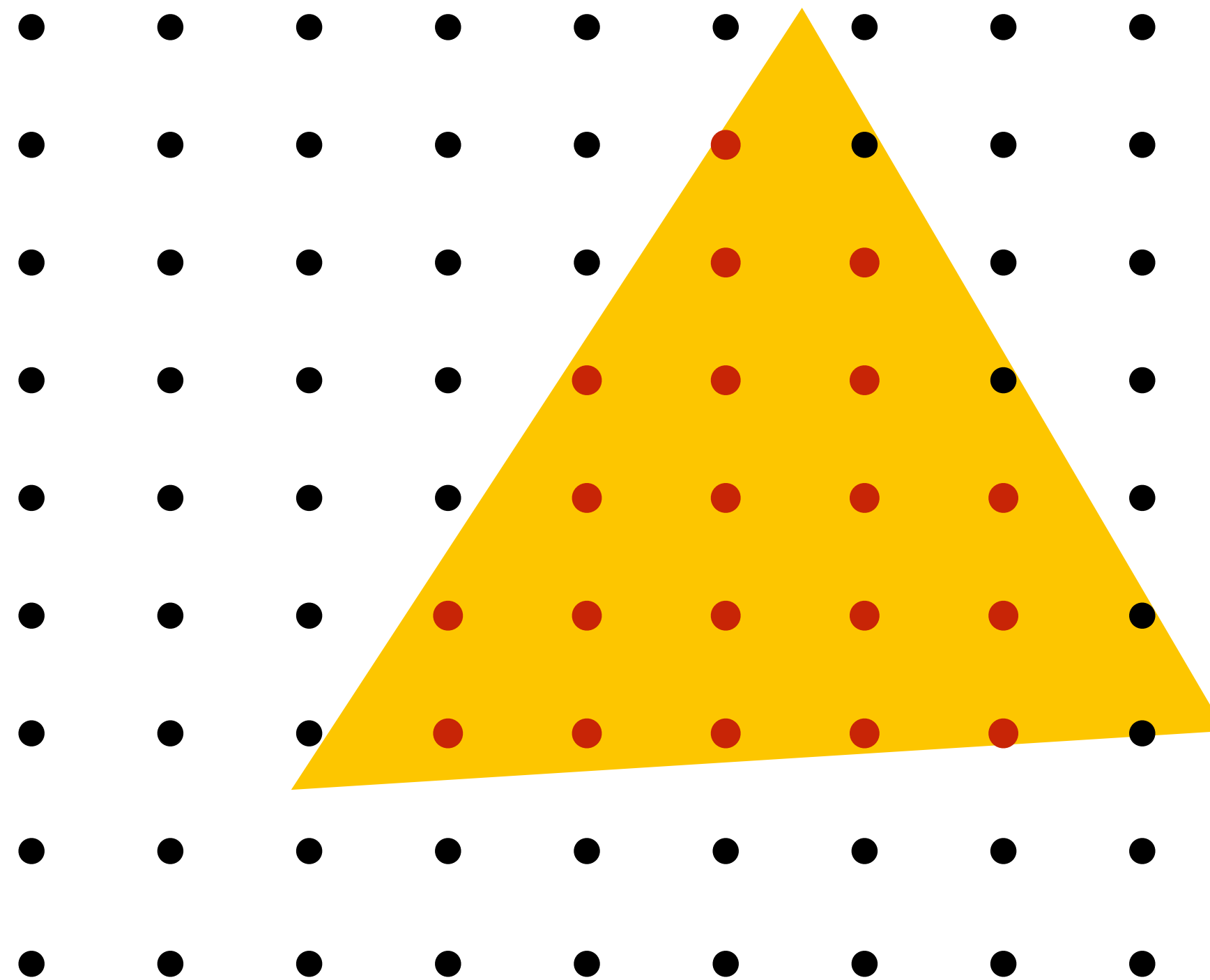
$$\mathbf{E}_{20}(x, y)$$

$$\frac{1}{\mathbf{w}}(x, y)$$

$$\mathbf{Z}(x, y)$$

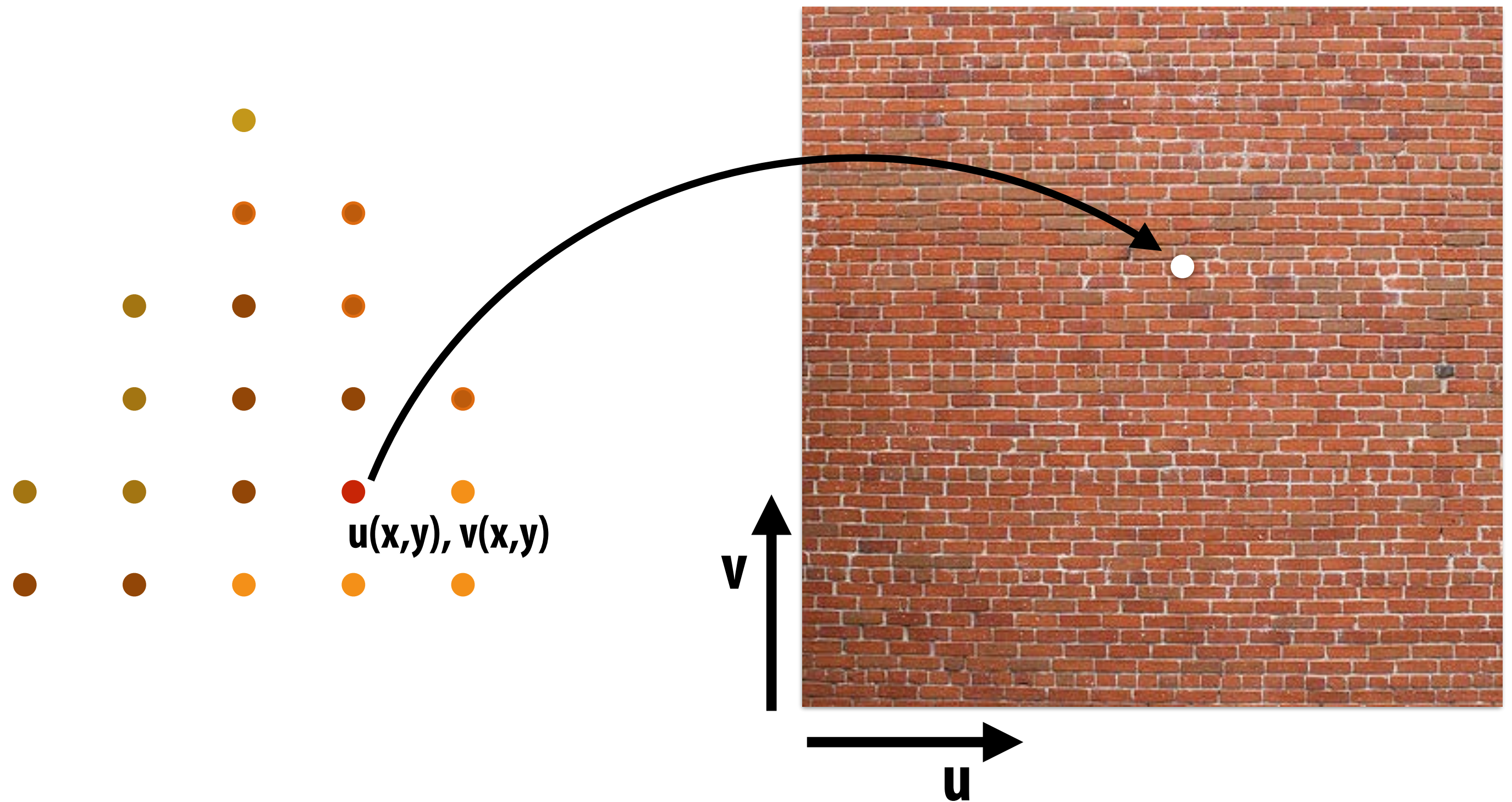
Step 6: sample coverage

Evaluate attributes z , u , v at all covered samples



Step 6: compute triangle color at sample point *

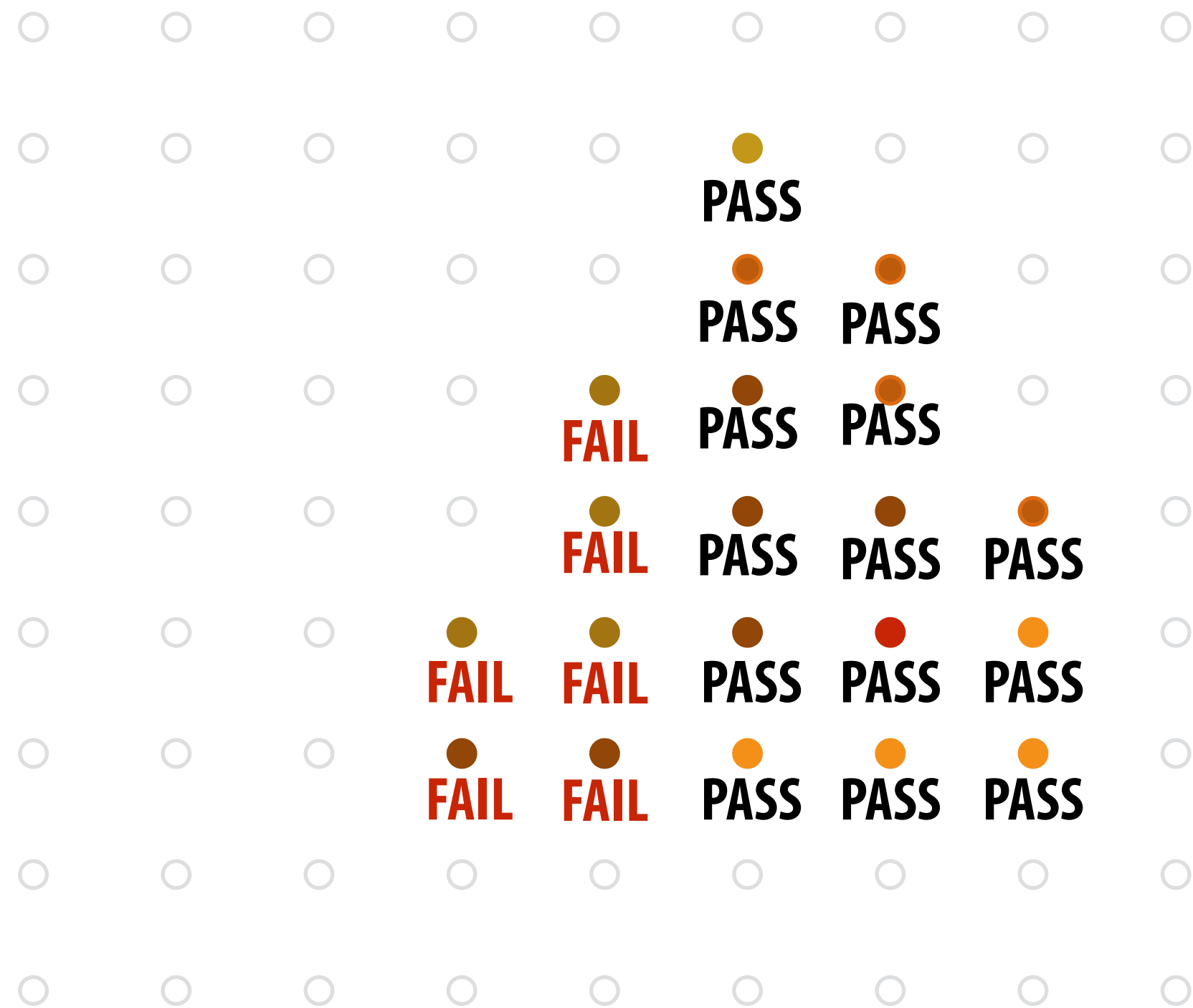
e.g., sample texture map *



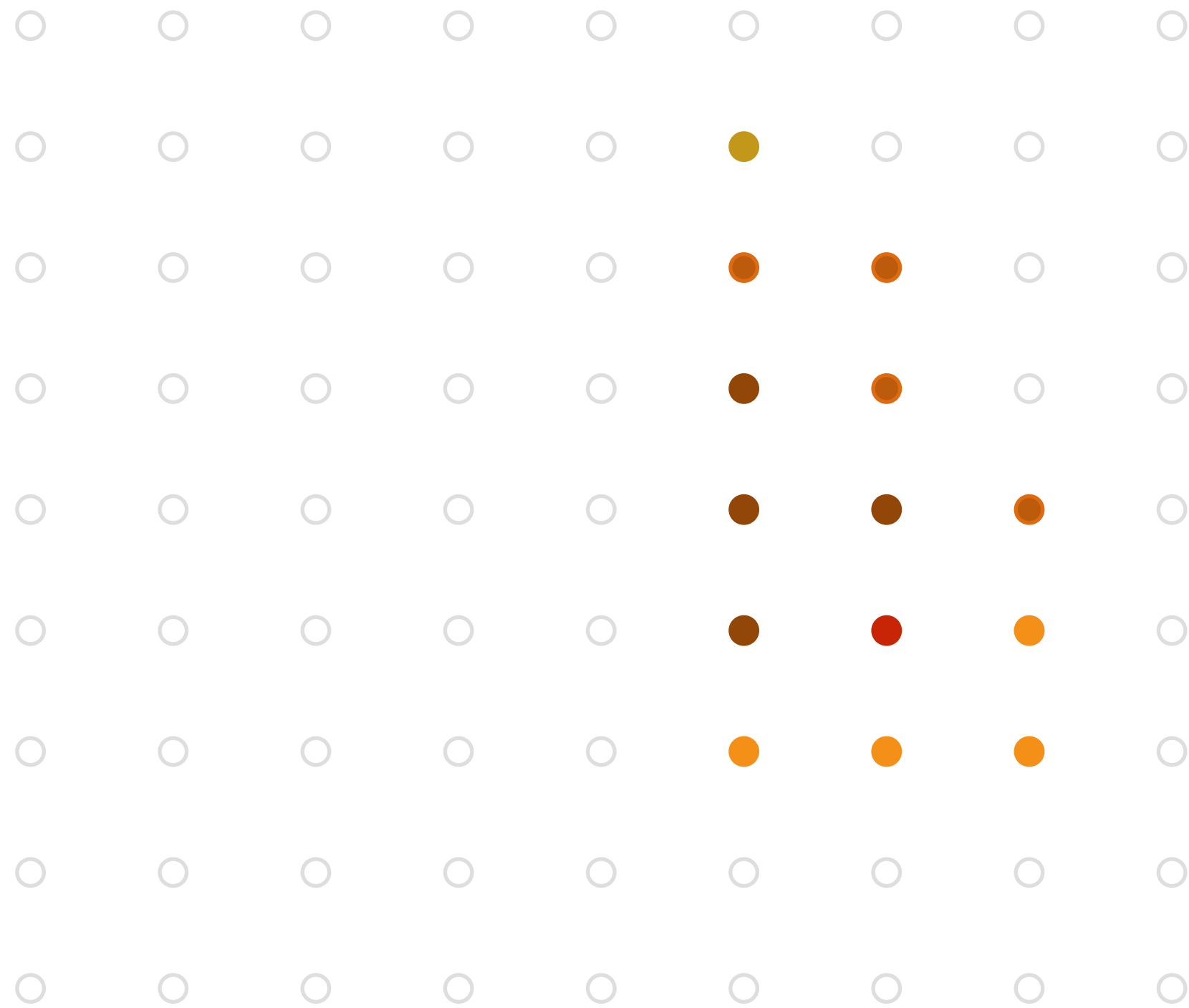
*Not the only way to get a color! Later we'll talk about more general models of materials...

Step 7: perform depth test (if enabled)

Also update depth value at covered samples (if necessary)



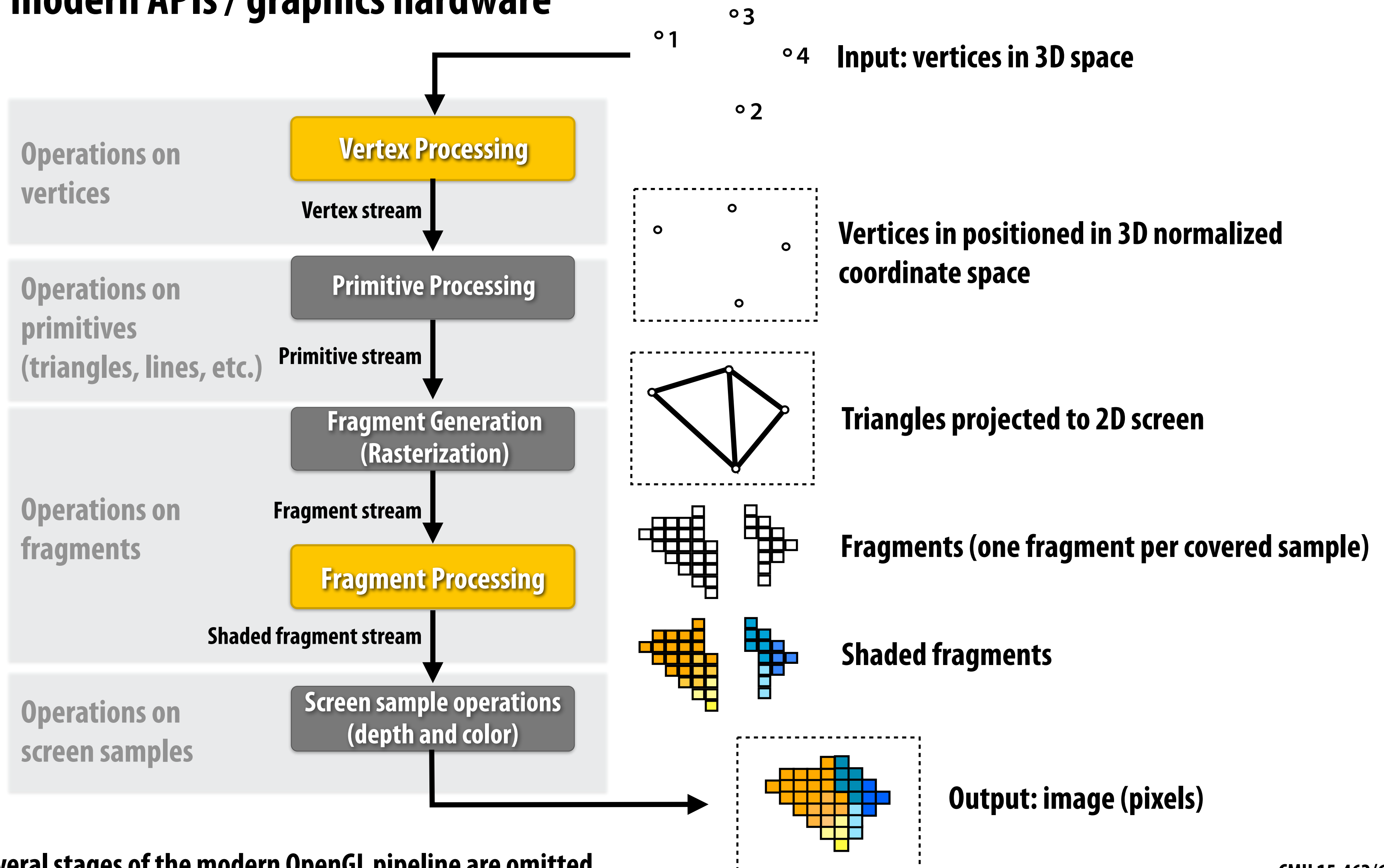
Step 8: update color buffer* (if depth test passed)



* Possibly using OVER operation for transparency

OpenGL/Direct3D graphics pipeline

Our rasterization pipeline doesn't look much different from "real" pipelines used in modern APIs / graphics hardware



Goal: render very high complexity 3D scenes

- 100's of thousands to millions of triangles in a scene
- Complex vertex and fragment shader computations
- High resolution screen outputs ($\sim 10\text{Mpixel}$ + *supersampling*)
- 30-120 fps

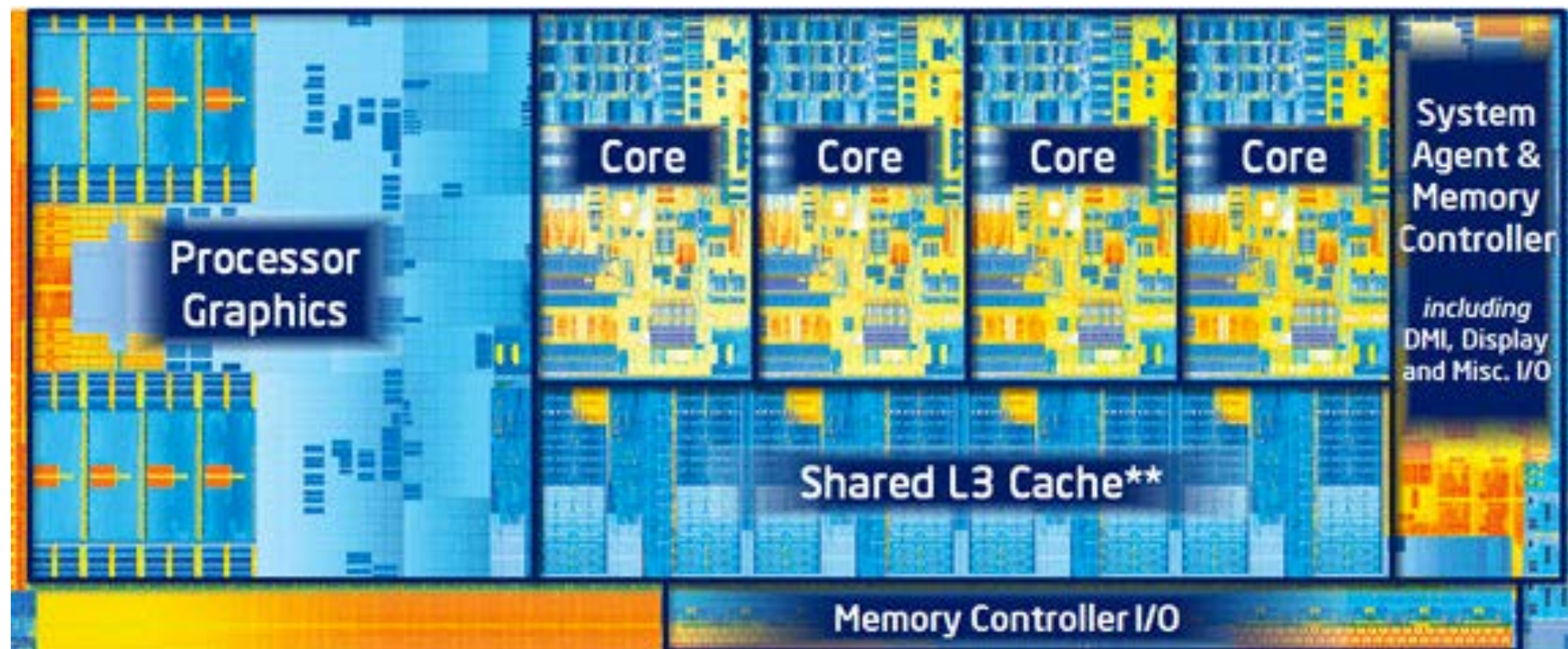


Graphics pipeline implementation: GPUs

Specialized processors for executing graphics pipeline computations

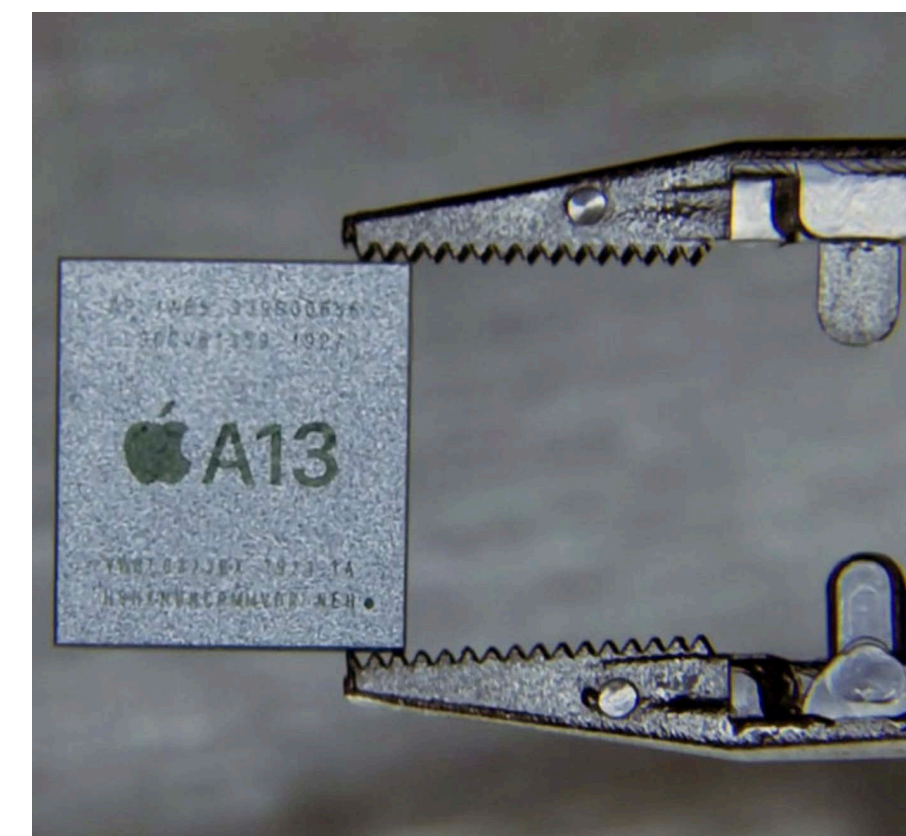


discrete GPU card



integrated GPU: part of modern CPU die

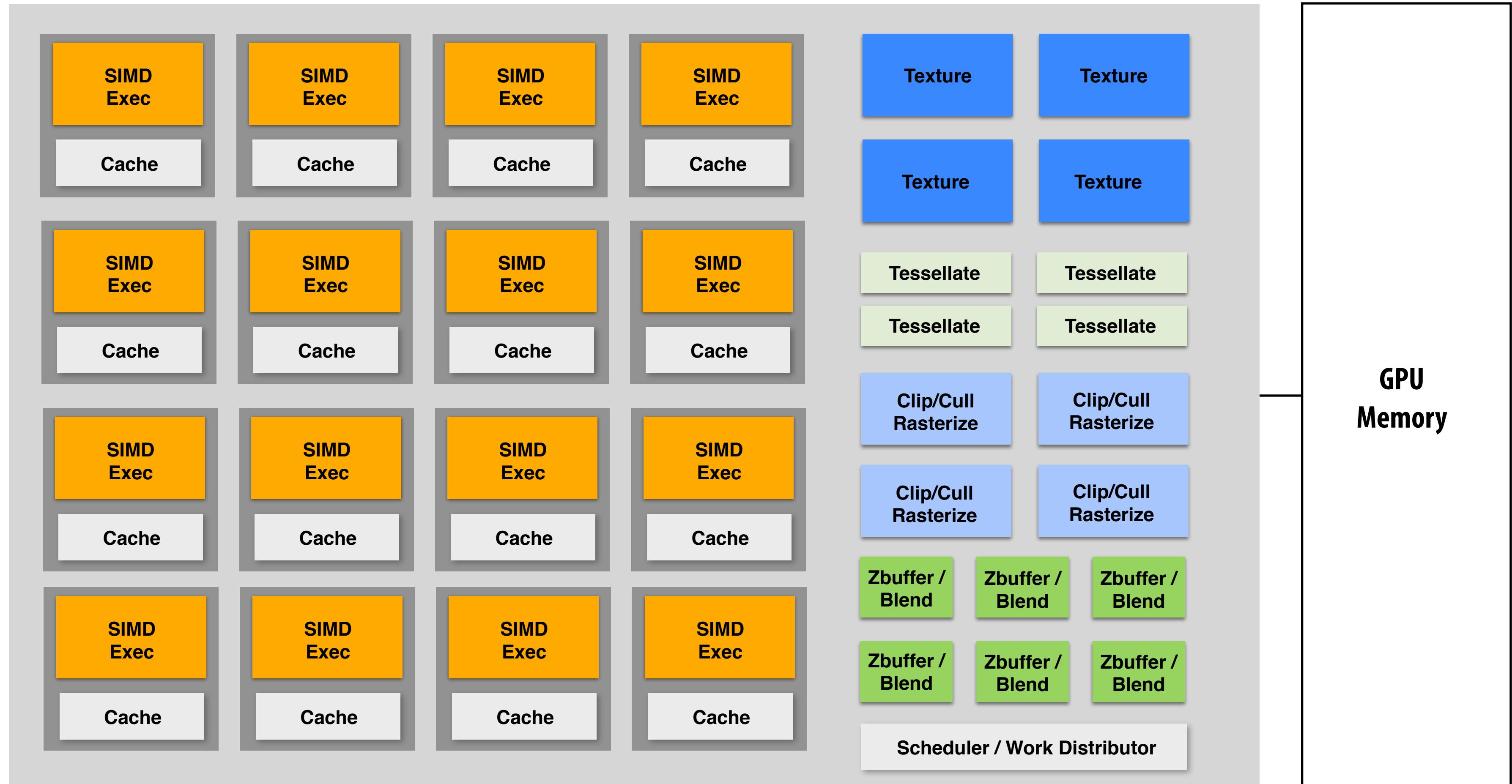
smartphone GPU (integrated)



GPU: heterogeneous, multi-core processor

Modern GPUs offer ~35 TFLOPs of performance for generic vertex/fragment programs (“compute”)

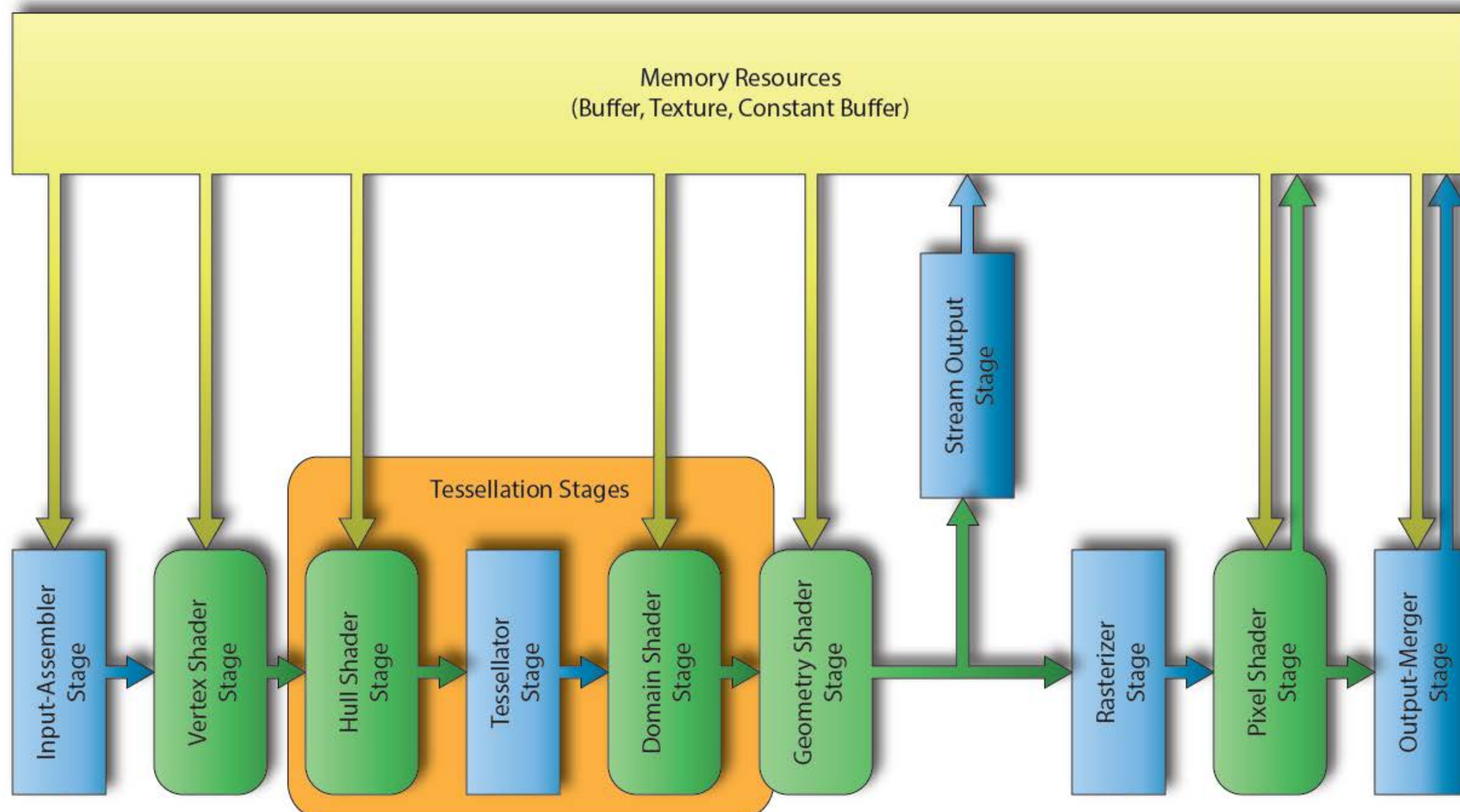
still enormous amount of *fixed-function* compute over here



This part (mostly) not used by CUDA/OpenCL; raw graphics horsepower still greater than compute!

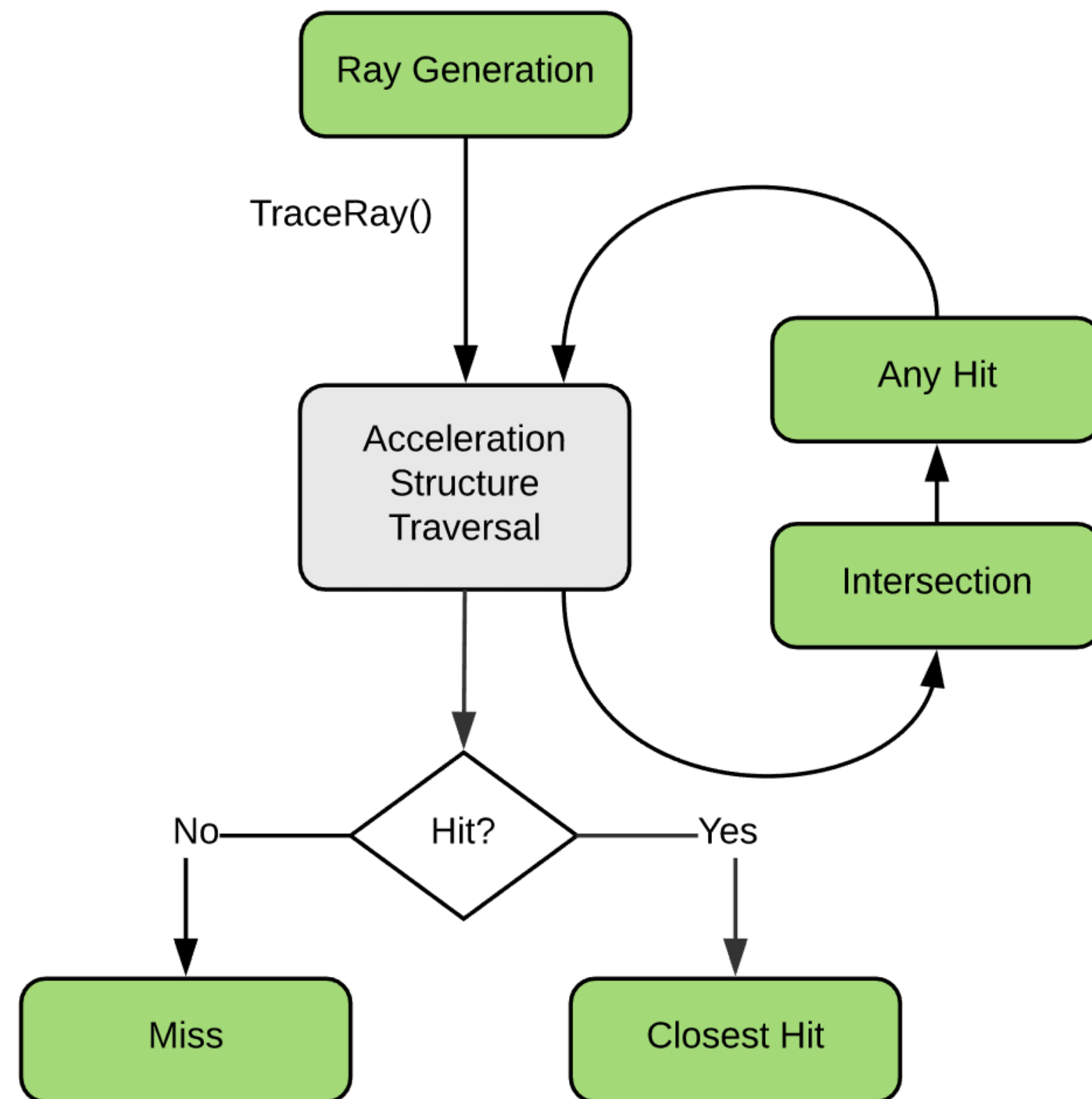
Modern Rasterization Pipeline

- Trend toward more generic (but still highly parallel!) computation:
 - make stages programmable
 - replace fixed function vertex, fragment processing
 - add geometry, tessellation shaders
 - generic “compute” shaders (whole other story...)
 - more flexible scheduling of stages



Ray Tracing in Graphics Pipeline

- More recently: specialized pipeline for ray tracing (NVIDIA RTX)



GPU Ray Tracing Demo (“Marbles at Night”)



What else do we need to know to generate images like these?

GEOMETRY

How do we describe complex shapes (so far just triangles...)

RENDERING

How does light interact w/ materials to produce color?

ANIMATION

How do we describe the way things move?



("Moana", Disney 2016)

Course roadmap

