Depth and Transparency

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

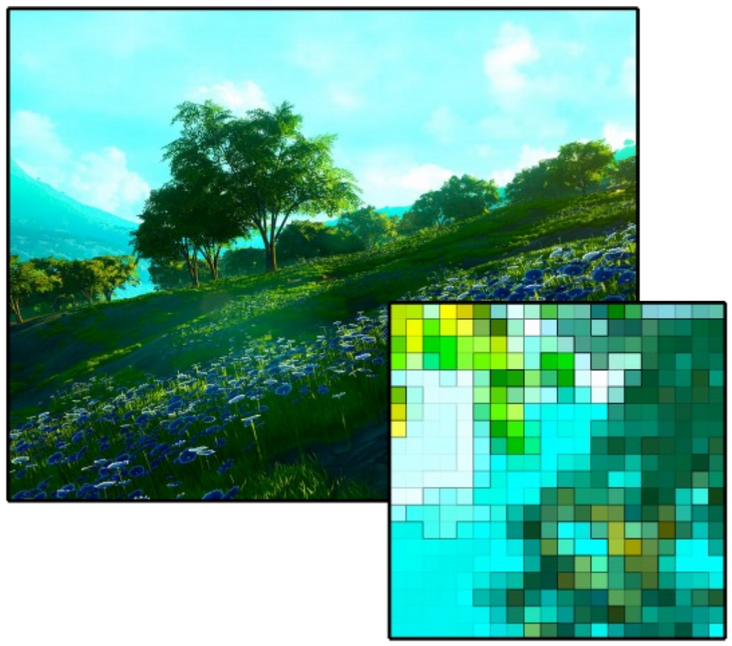
```
A: ( 1, 1, 1 )   E: ( 1, 1,-1 )
B: (-1, 1, 1 )   F: (-1, 1,-1 )
C: ( 1,-1, 1 )   G: ( 1,-1,-1 )
D: (-1,-1, 1 )   H: (-1,-1,-1 )
```

TRIANGLES

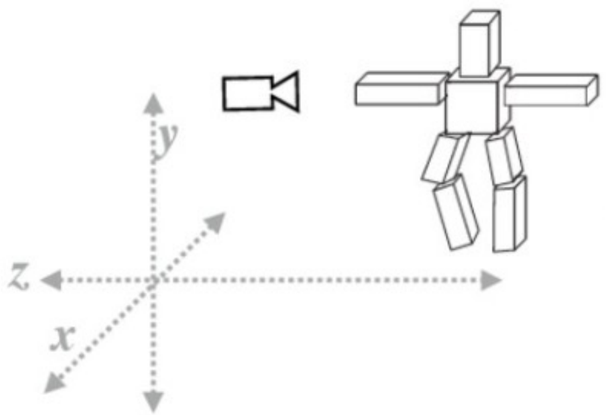
```
EHF, GFH, FGB, CBG,
GHC, DCH, ABD, CDB,
HED, ADE, EFA, BAF
```

RASTERIZATION PIPELINE

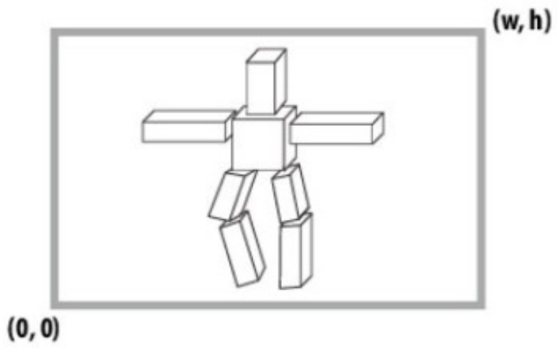
OUTPUT (BITMAP IMAGE)



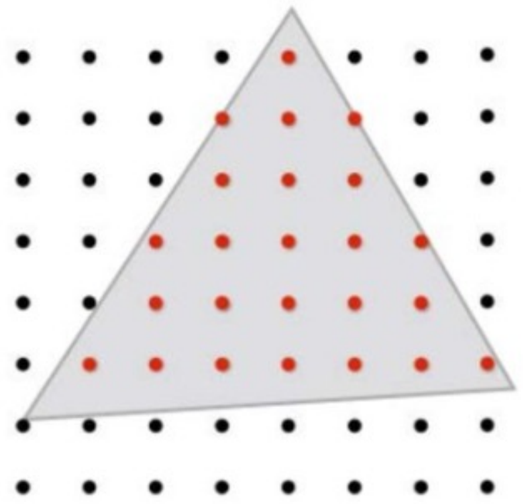
What we know how to do so far…



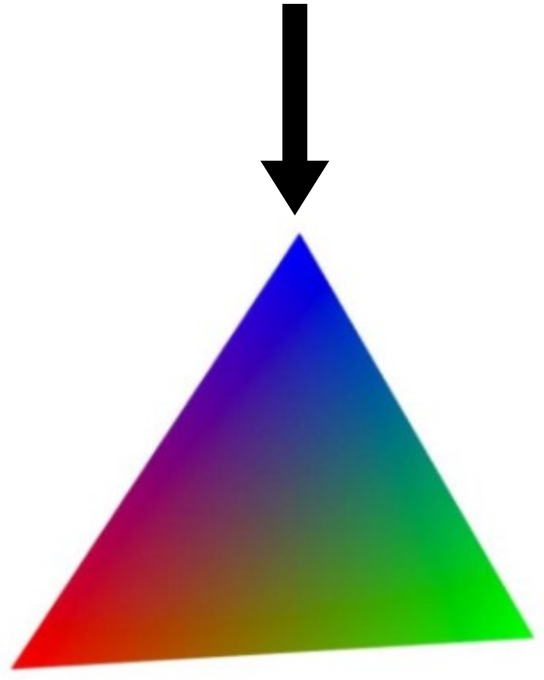
position objects in the world
(3D transformations)



project objects onto the screen
(perspective projection)



sample triangle coverage
(rasterization)



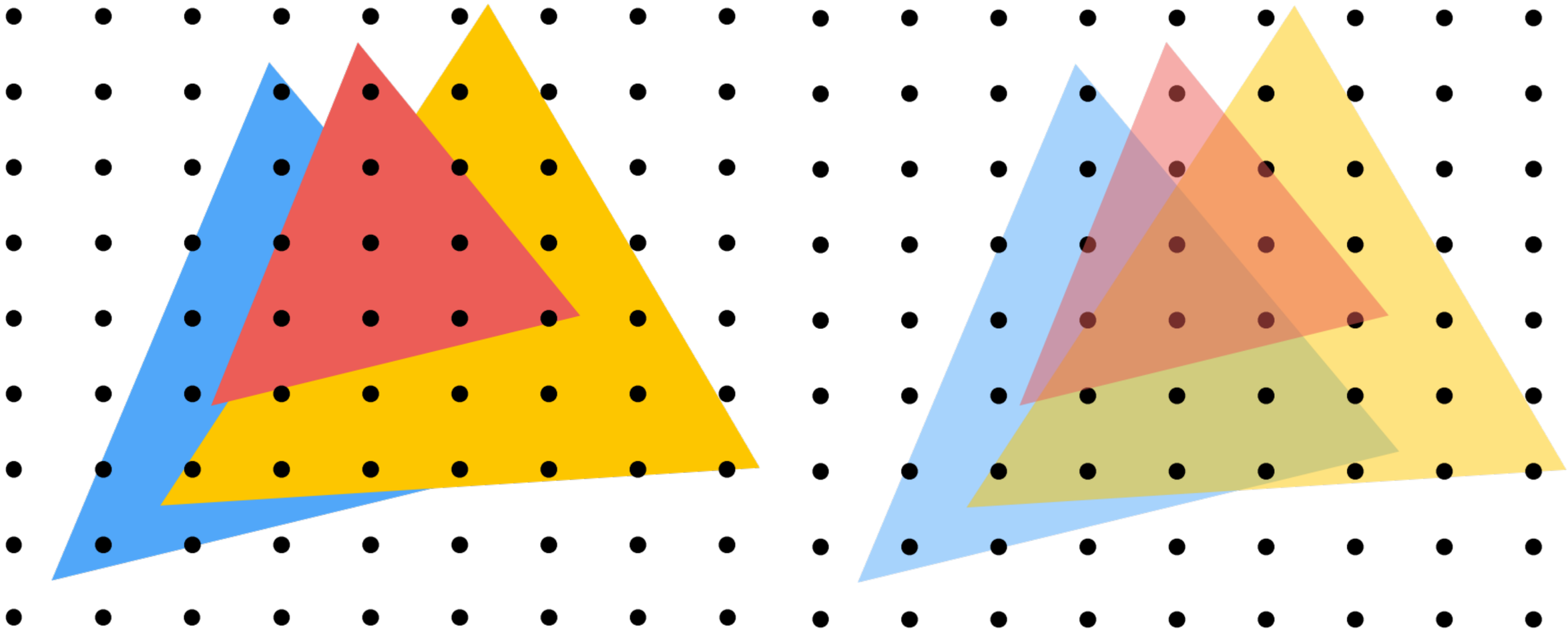
interpolate vertex attributes
(barycentric coodinates)

sample texture maps
(filtering, mipmapping)

put samples into frame buffer
(depth & alpha)

Occlusion

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



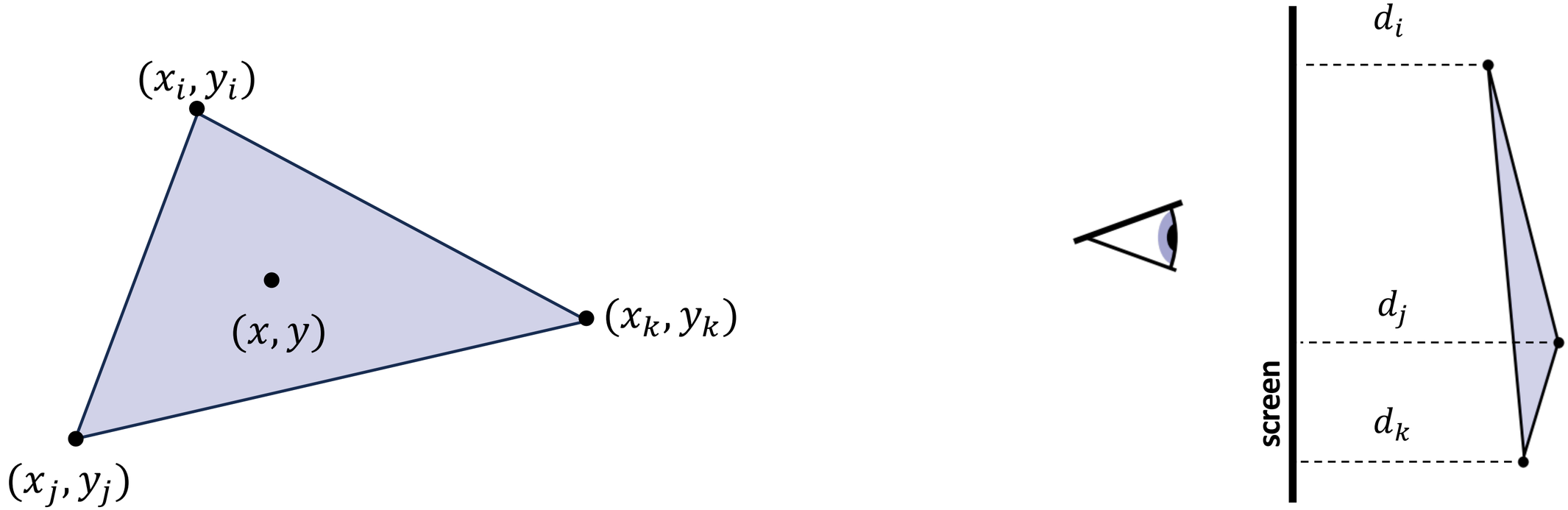
Opaque Triangles

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)**

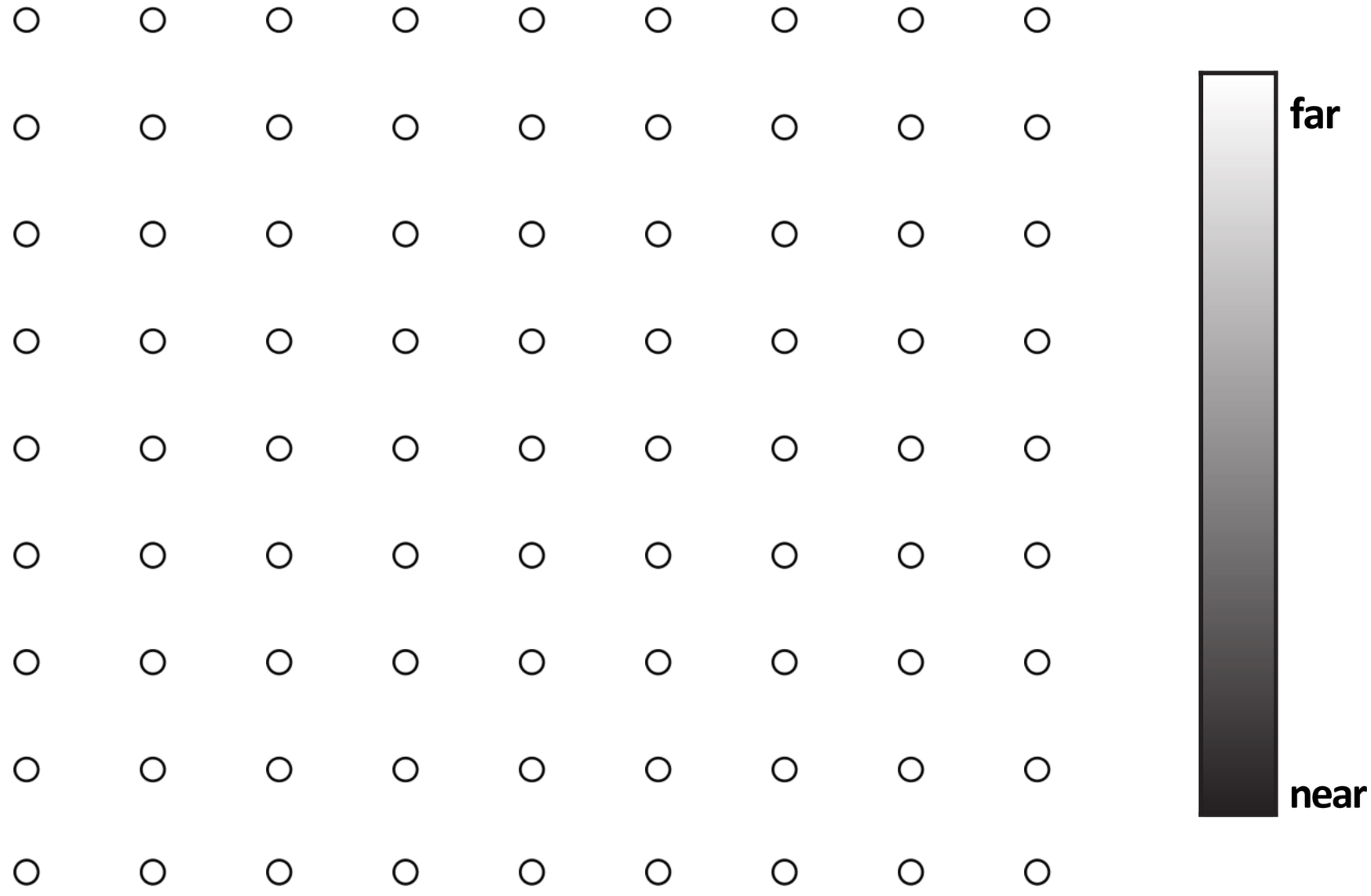


Q: How do we compute the depth d at a given sample point (x, y)?

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

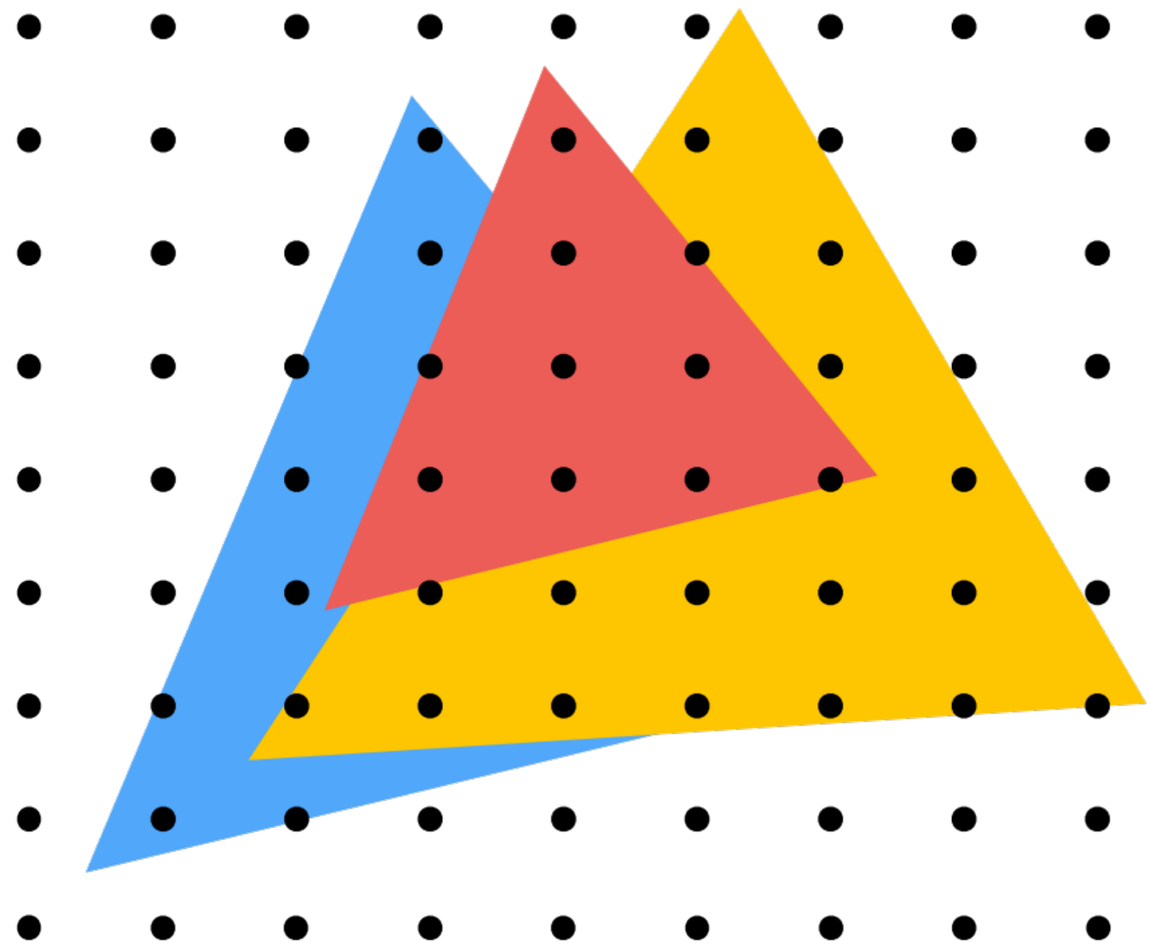


Initialize all depth buffer values to “infinity” (max value)

Depth buffer example

near far

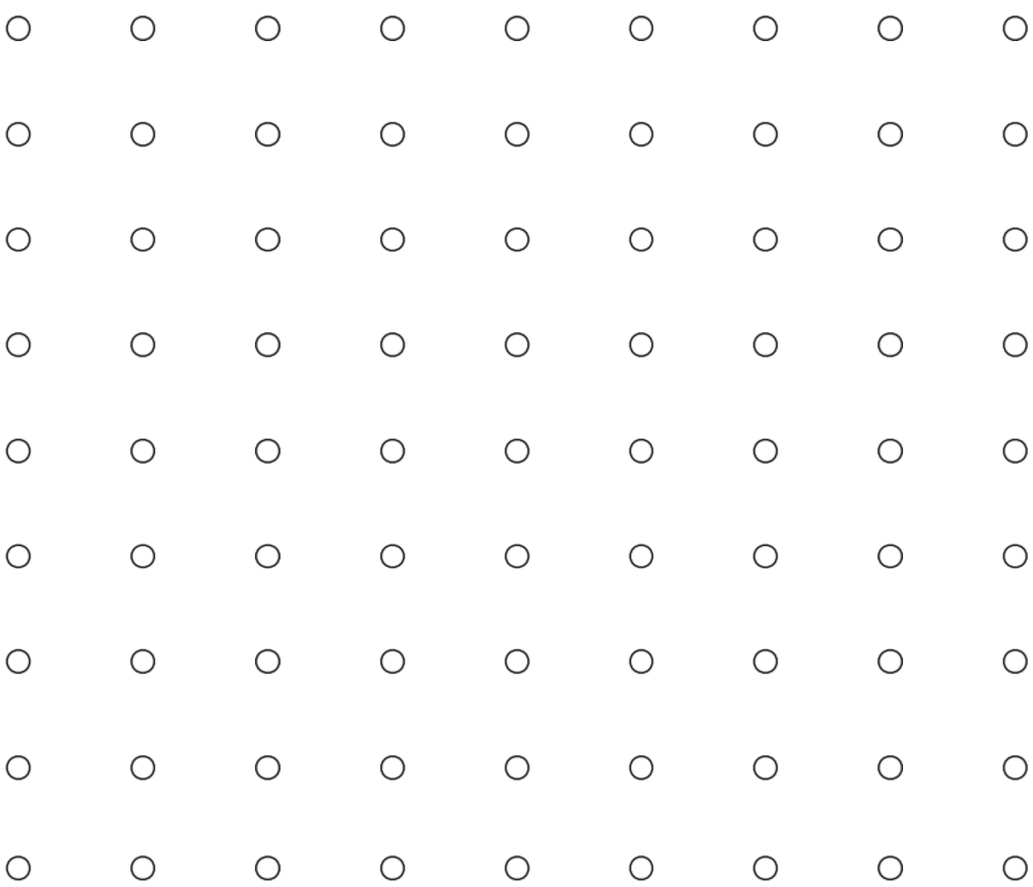
Example: rendering three opaque triangles



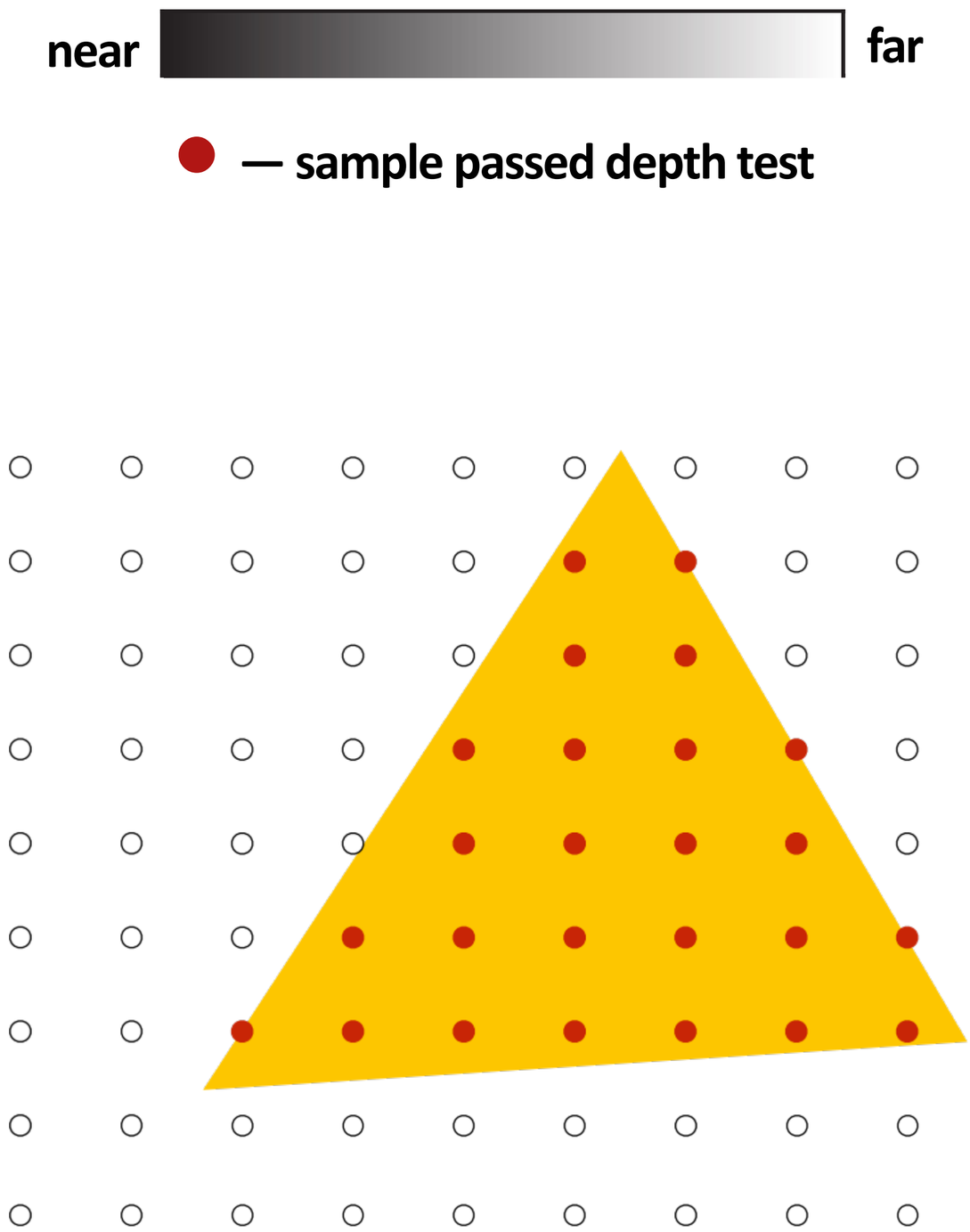
Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:

depth = 0.5



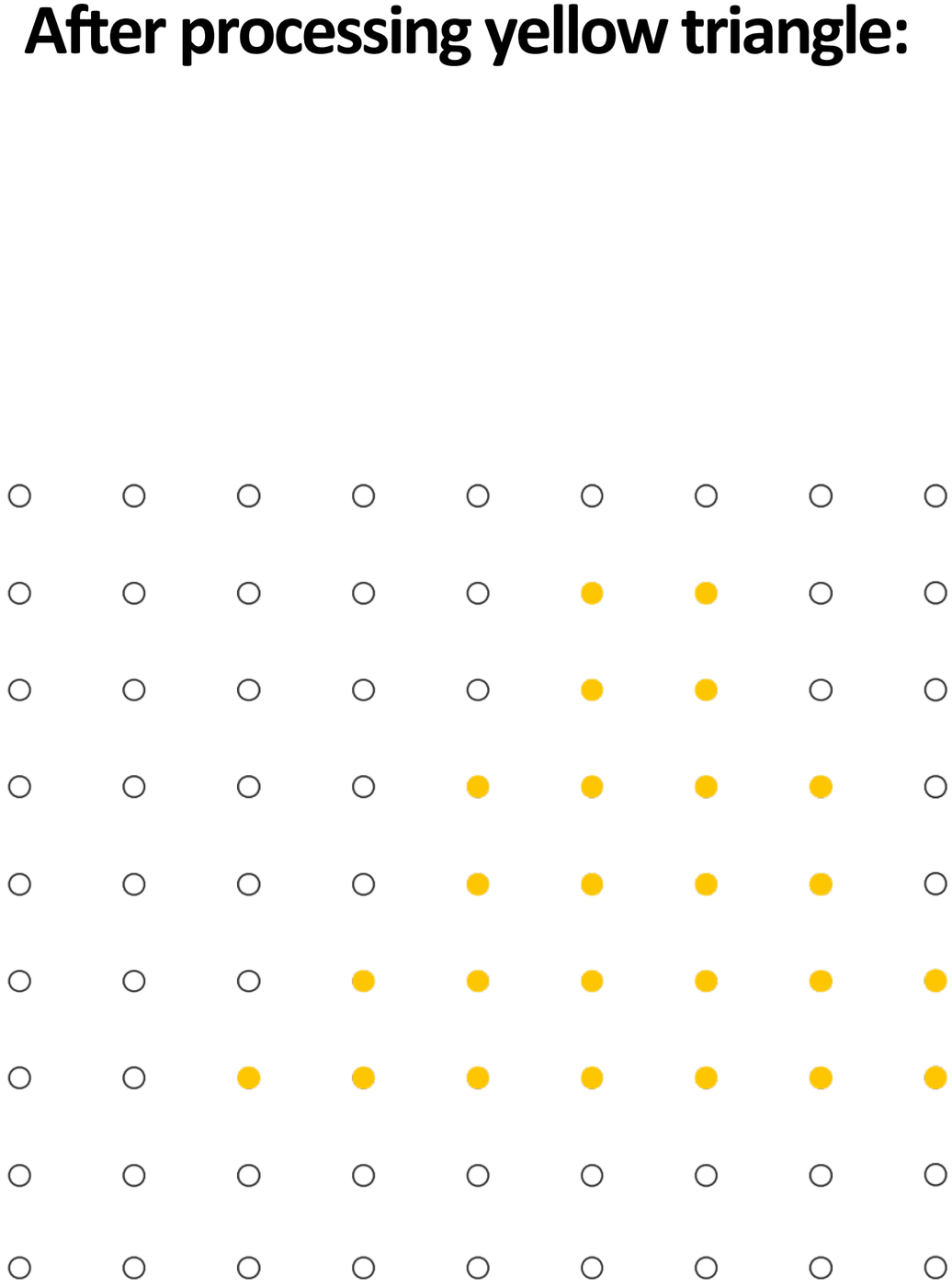
Color buffer contents



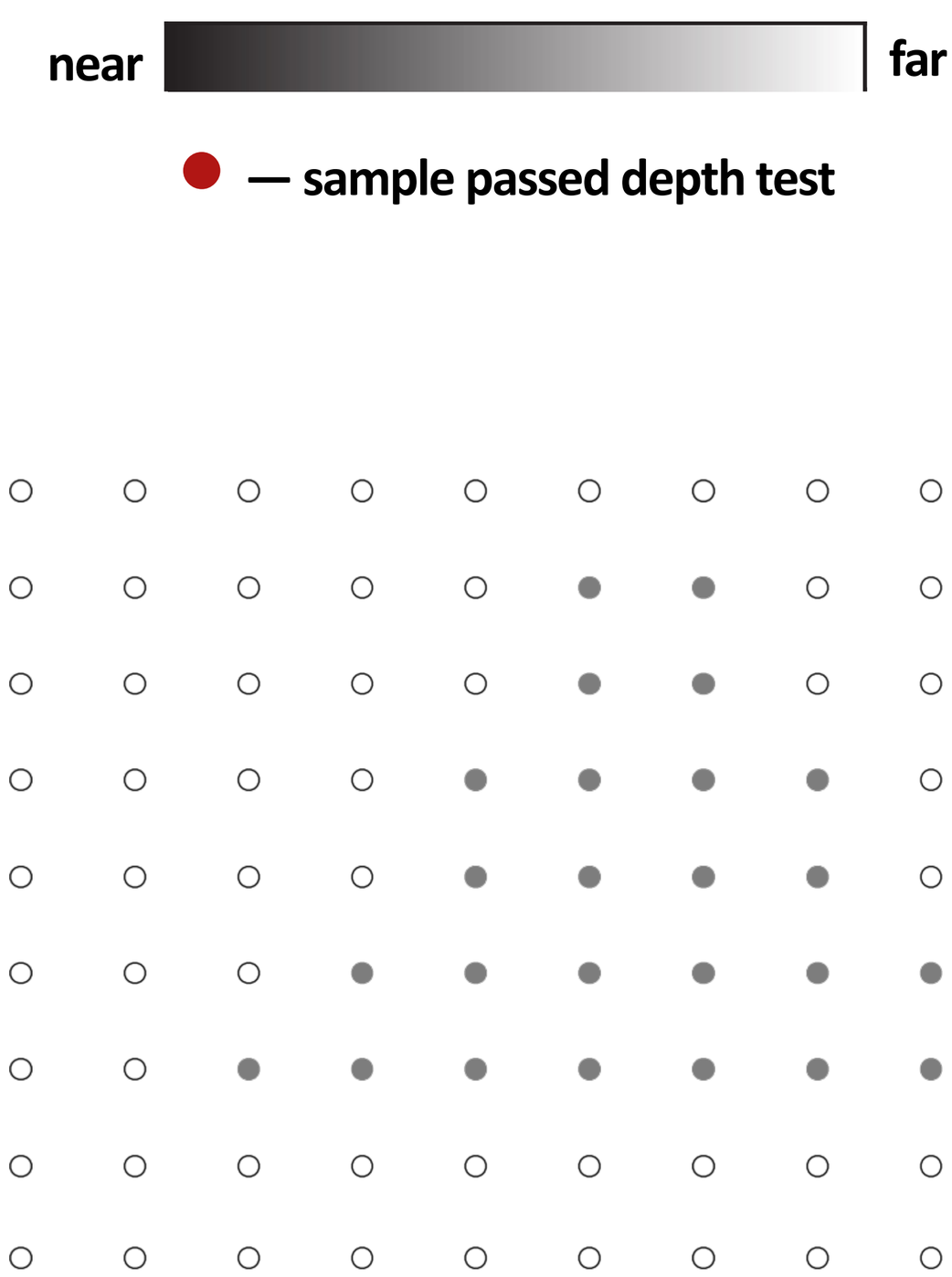
Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing yellow triangle:



Color buffer contents

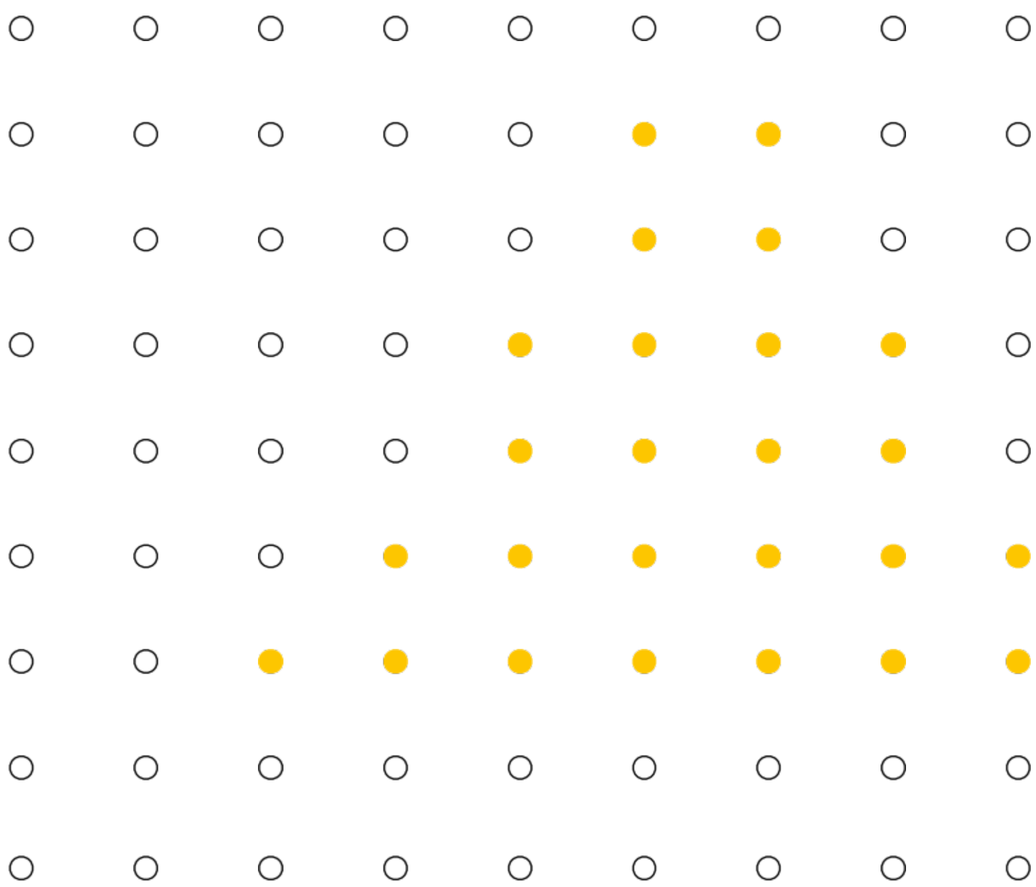


Depth buffer contents

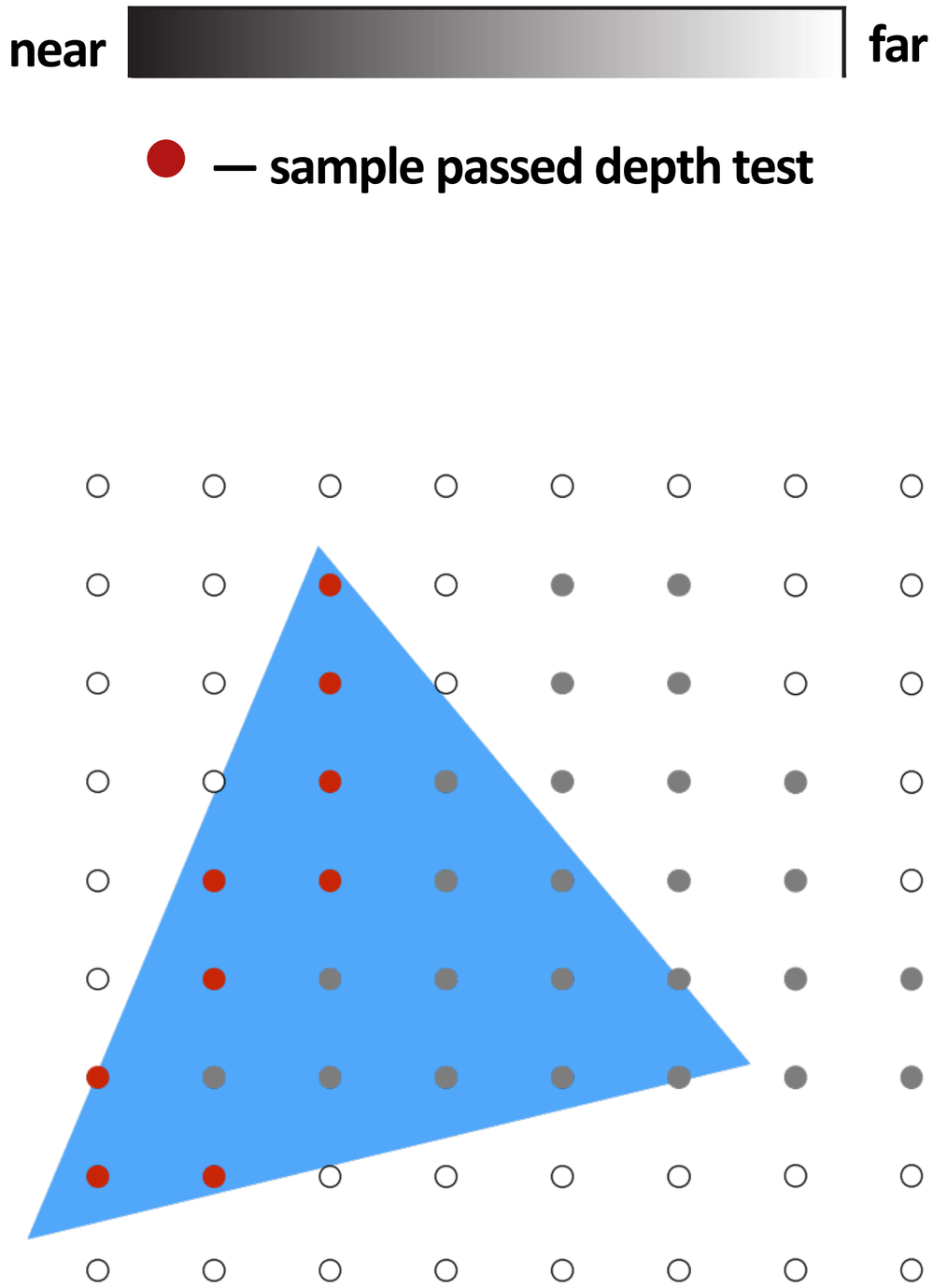
Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:

depth = 0.75



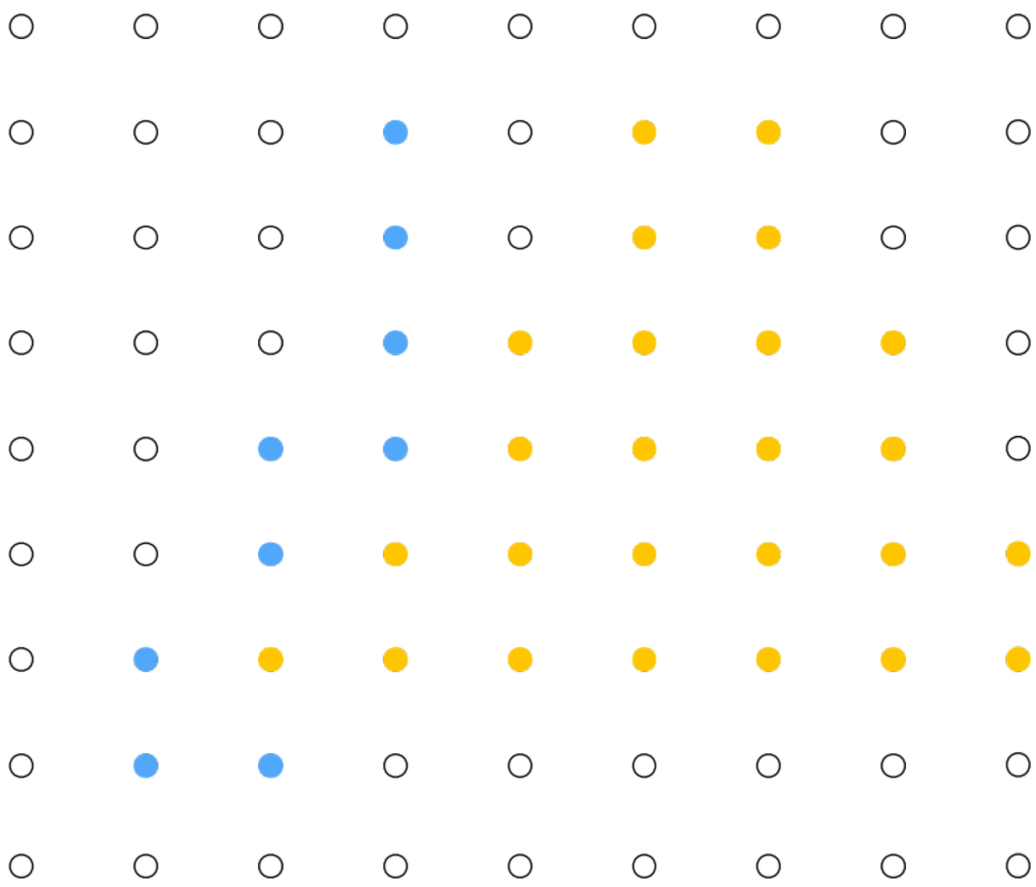
Color buffer contents



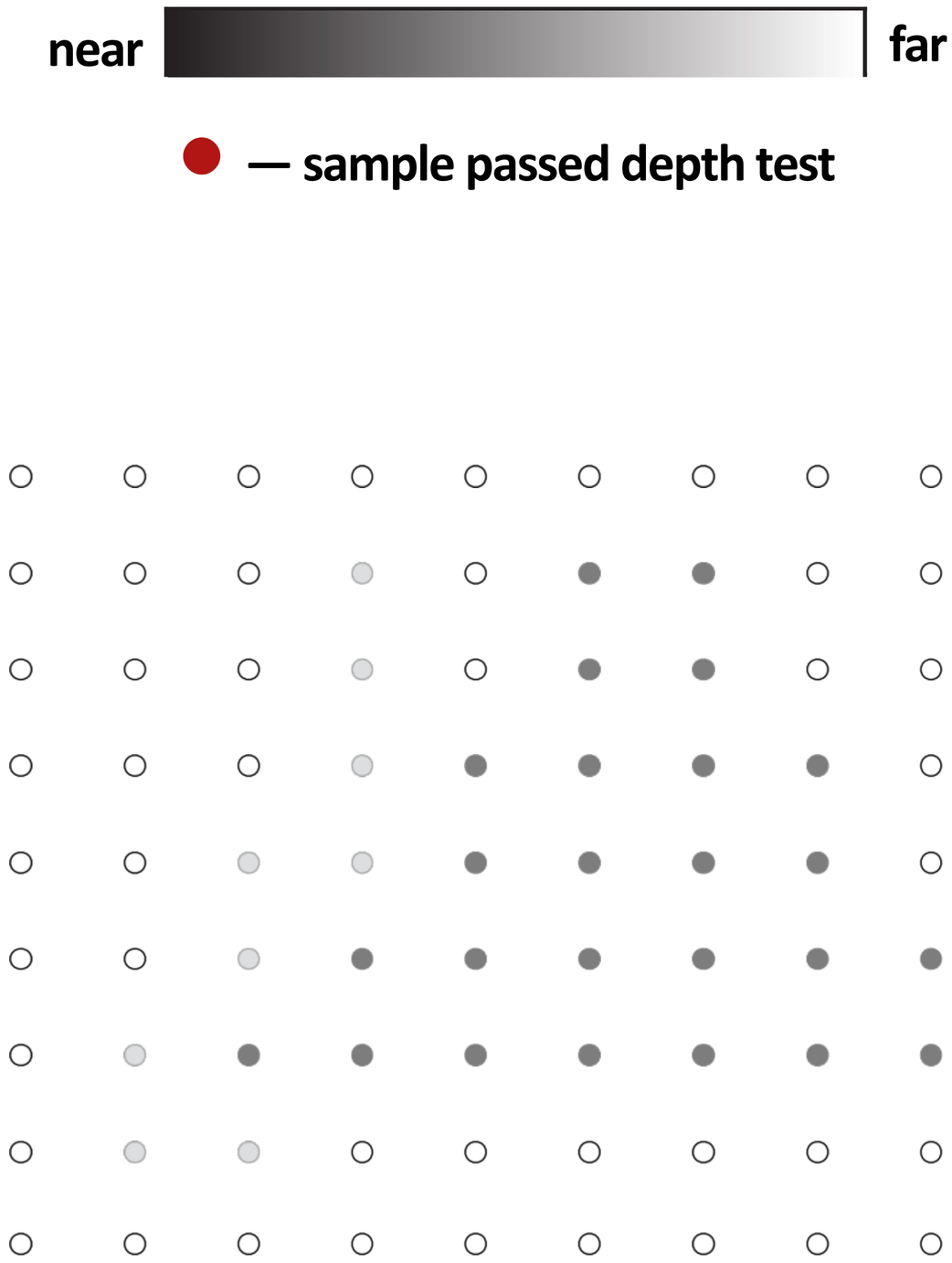
Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing blue triangle:



Color buffer contents

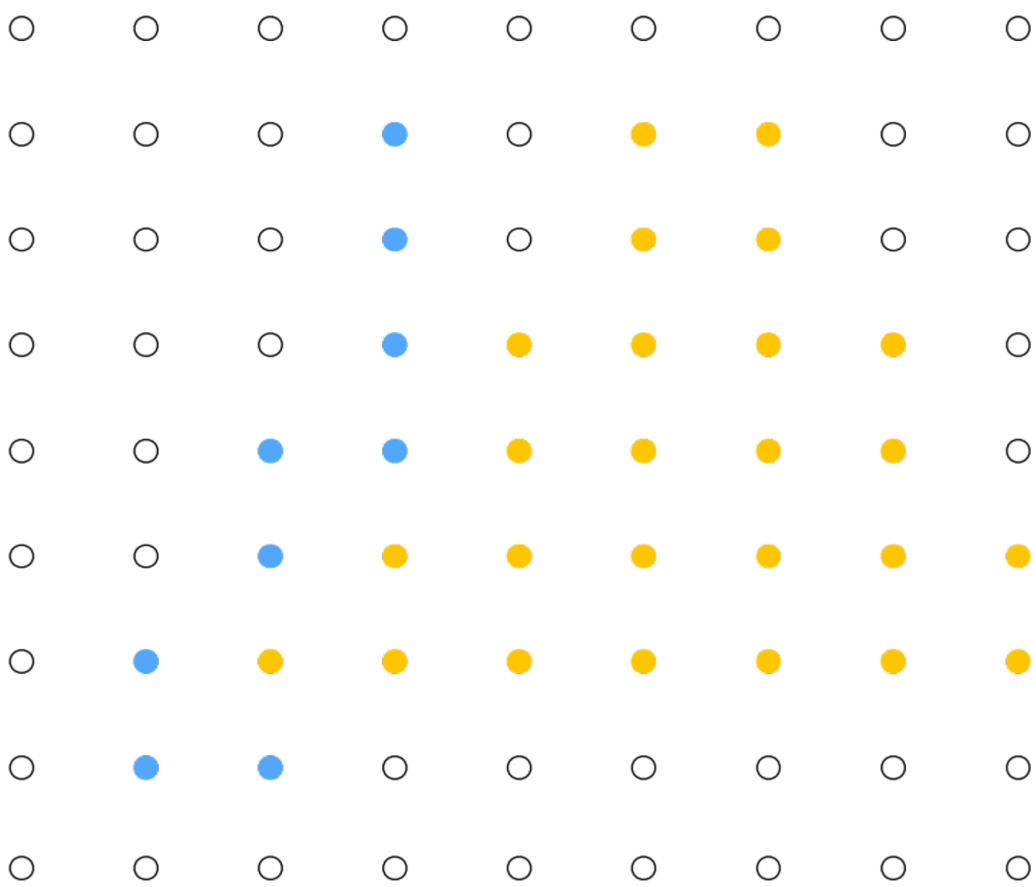


Depth buffer contents

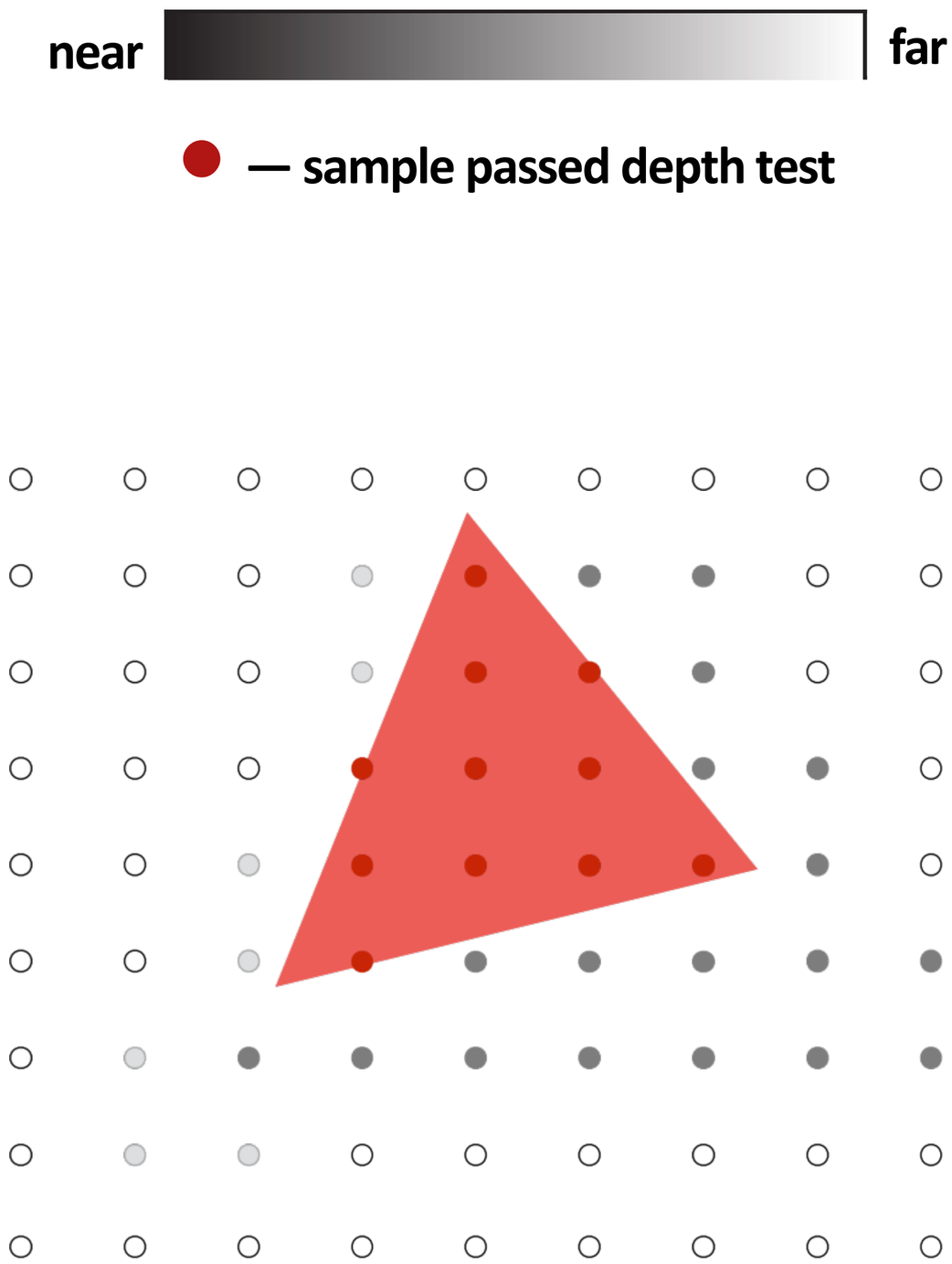
Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:

depth = 0.25



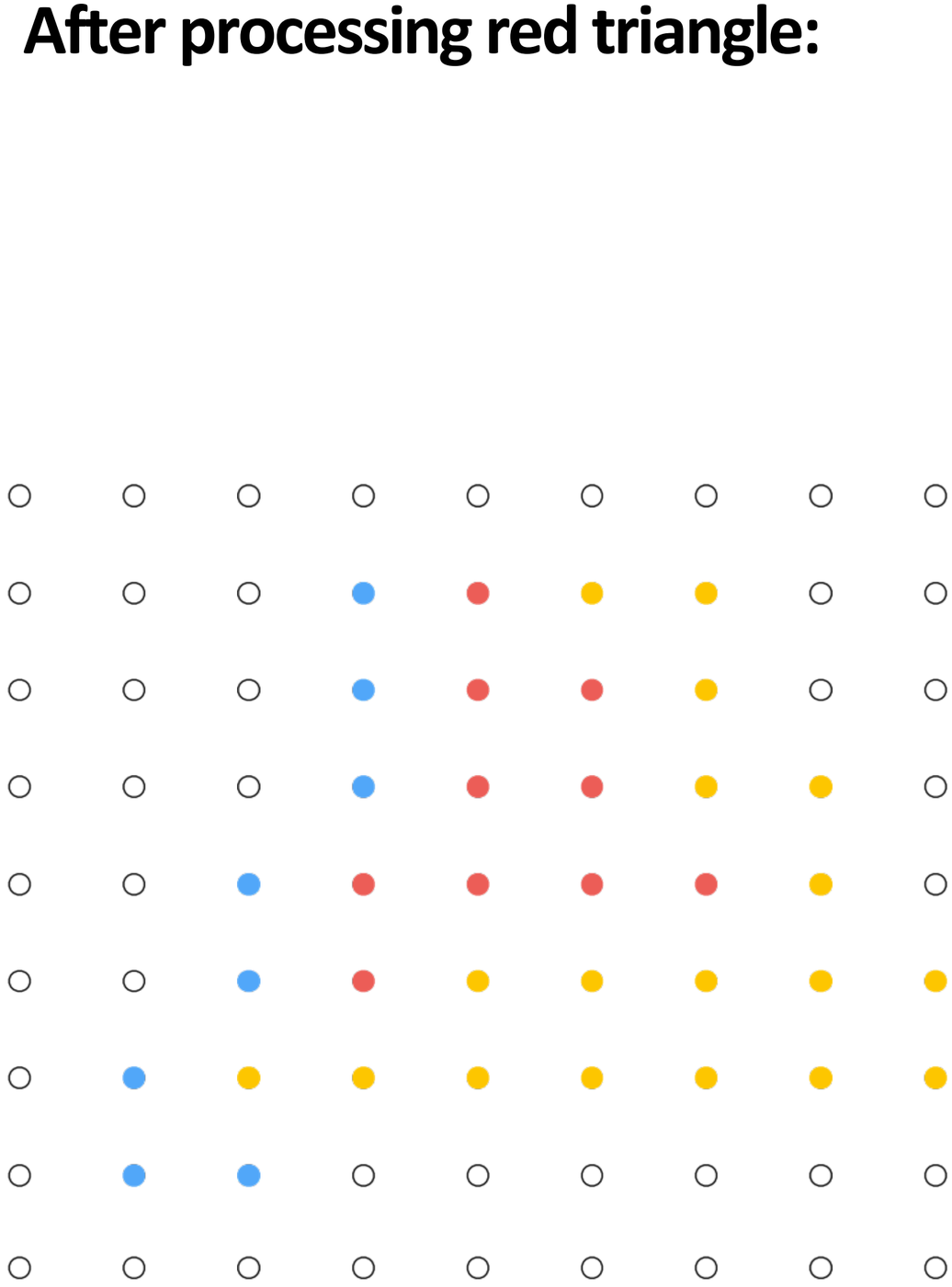
Color buffer contents



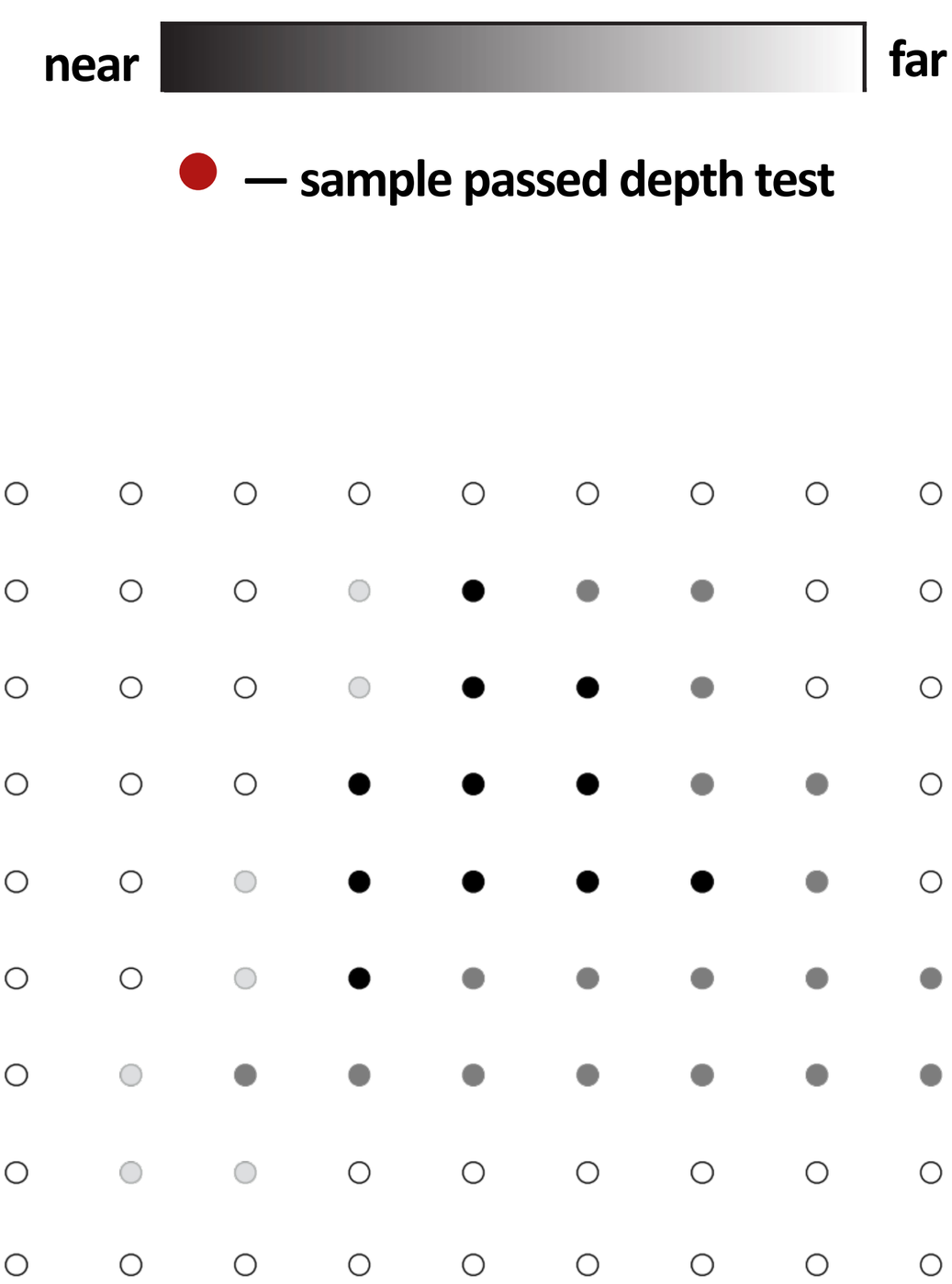
Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:



Color buffer contents



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
}
```

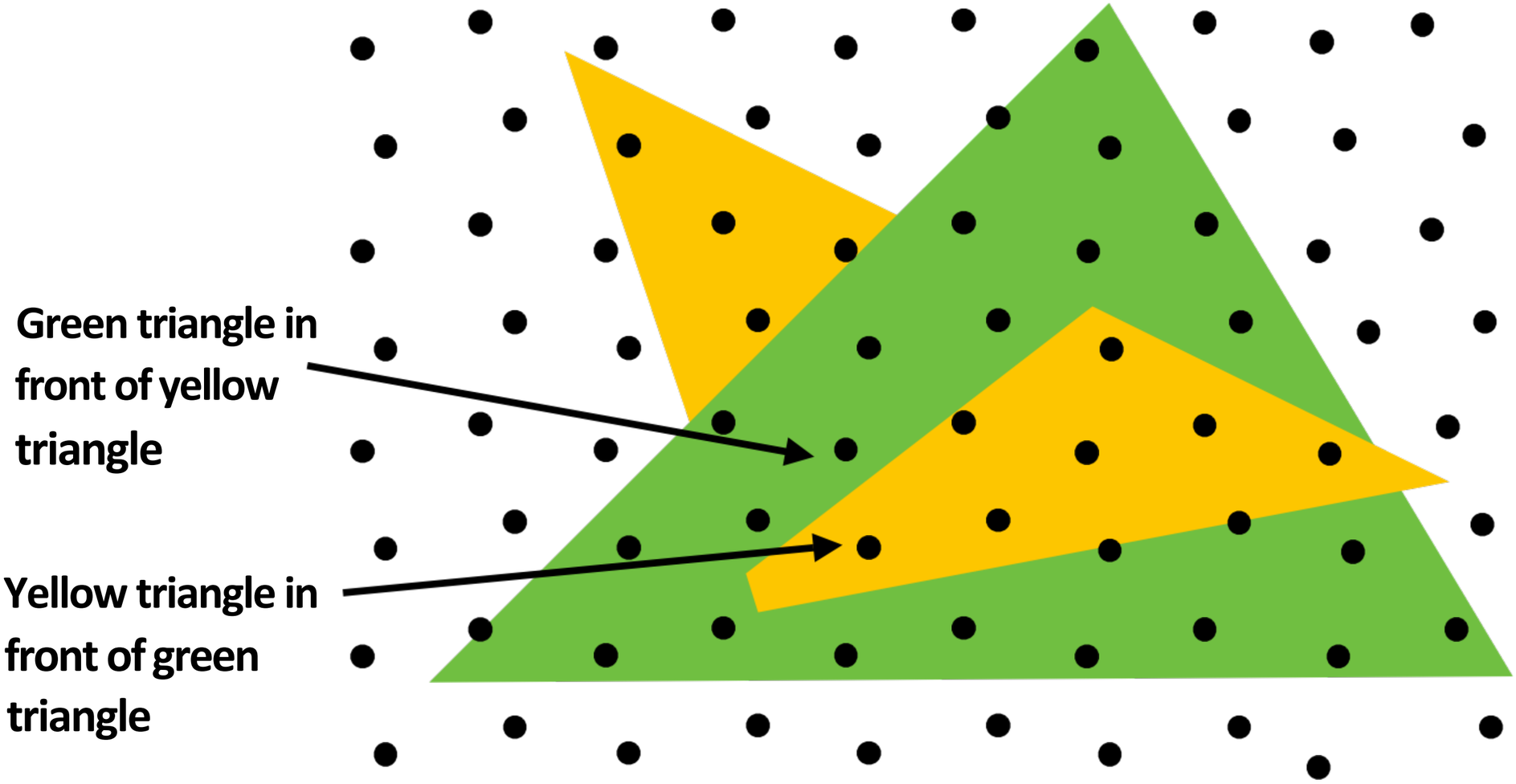
Depth + Intersection

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 maybe different at different sample points.

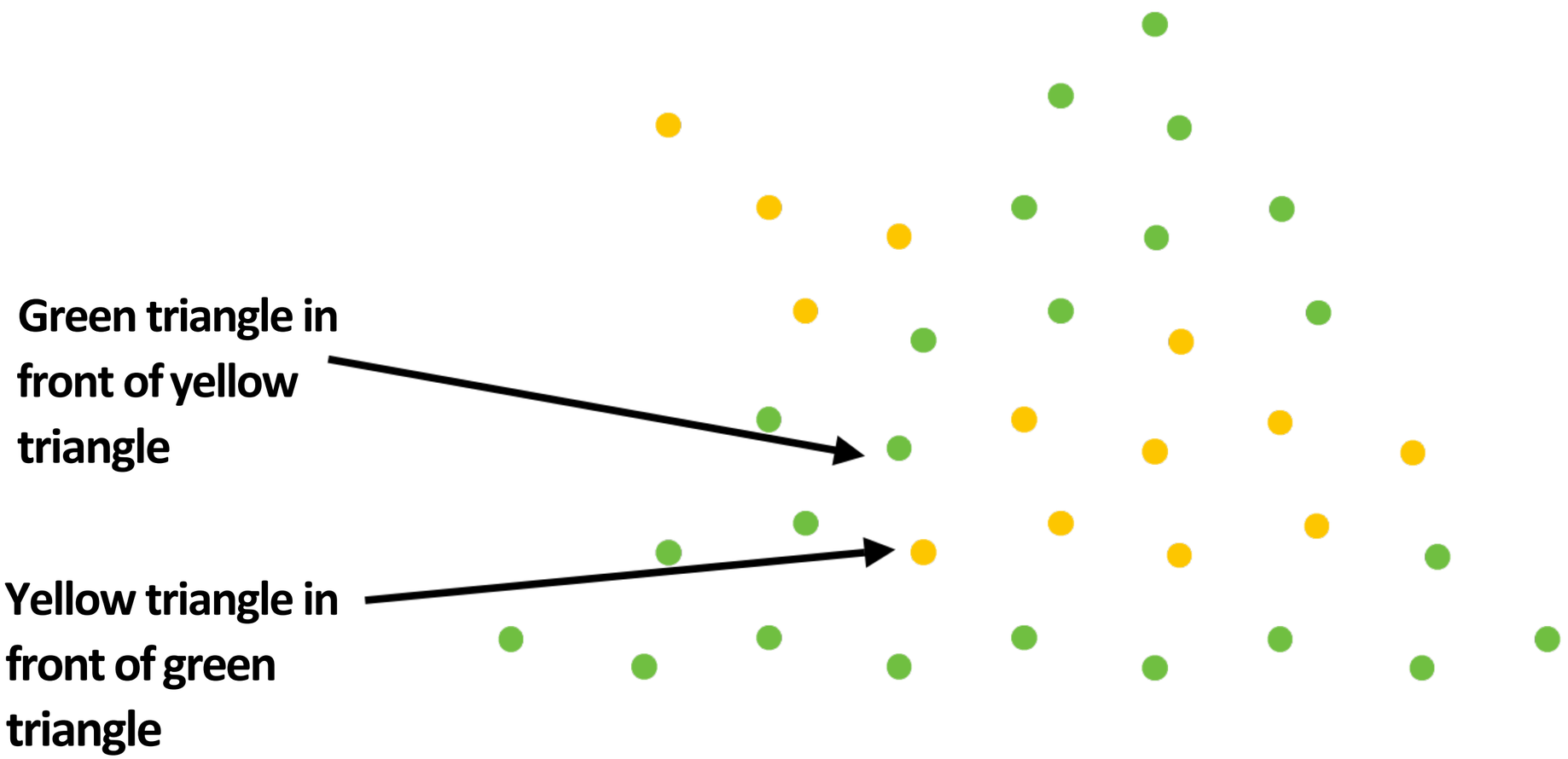


Intersection

Q: Does depth-buffer algorithm handle interpenetrating surfaces?

A: Of course!

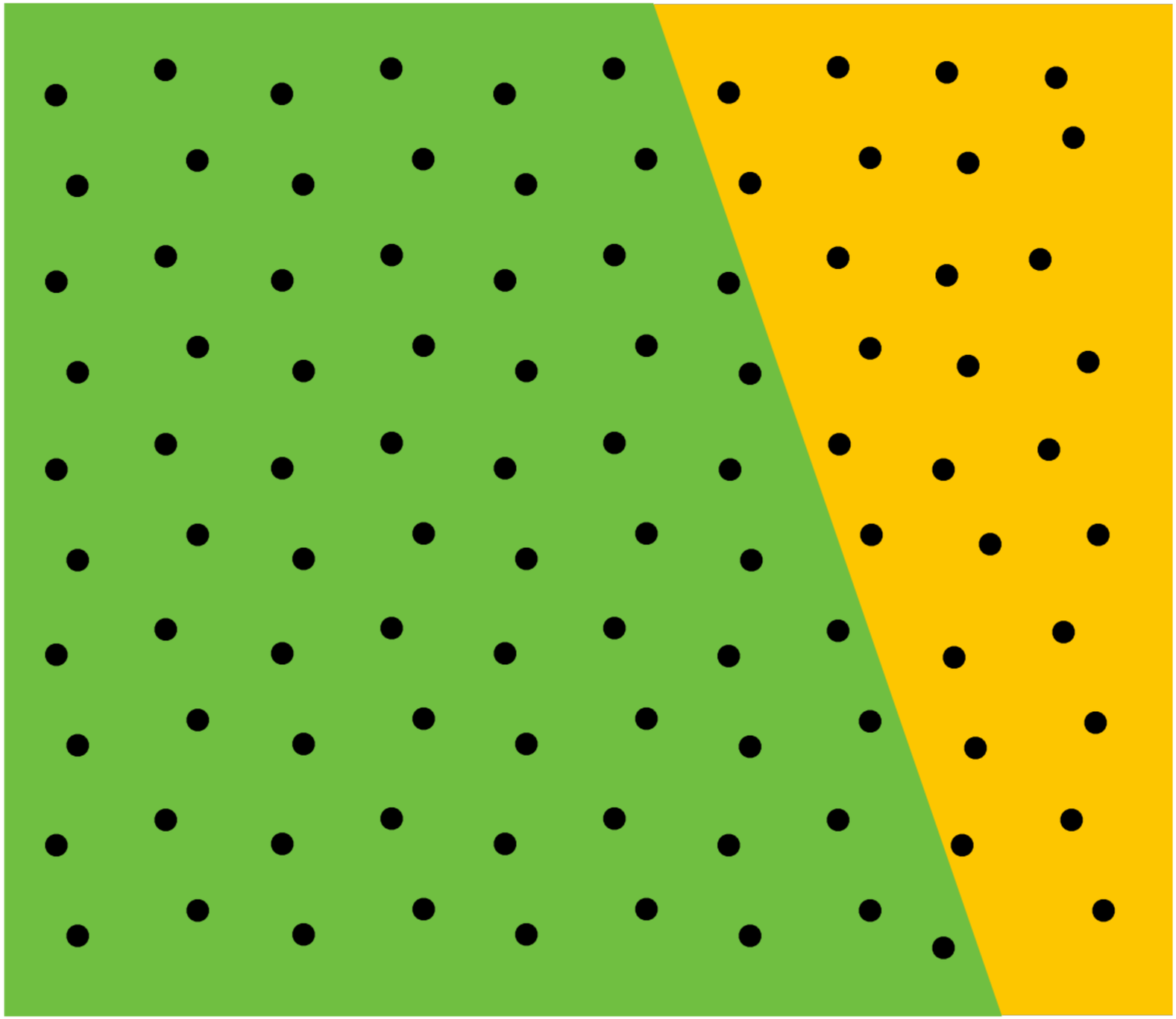
Occlusion testis based on depth of triangles at a given sample point.
Relative depth of triangles maybe different at different sample points.



Depth + Supersampling

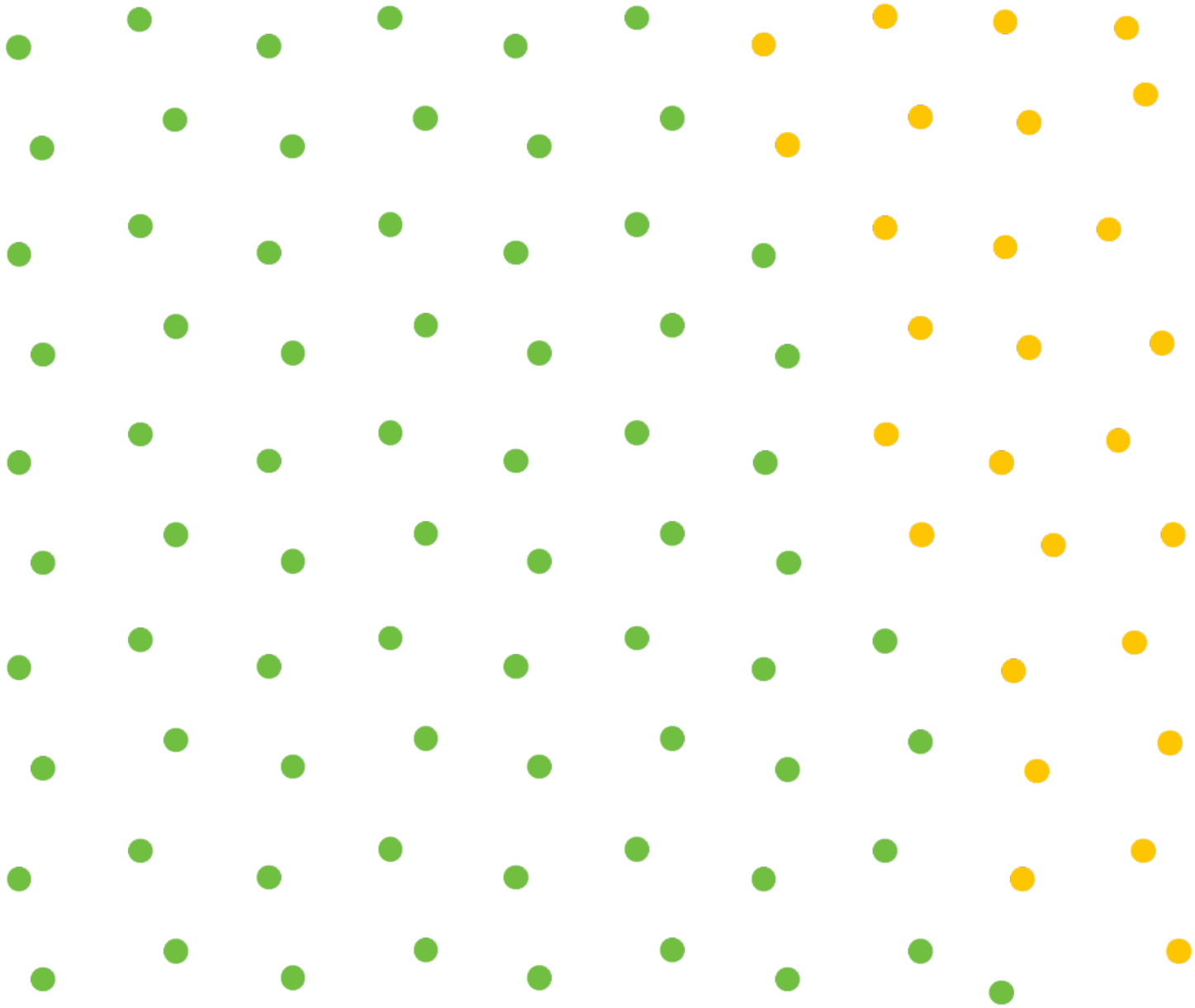
Q: Does depth buffer work with super sampling?

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



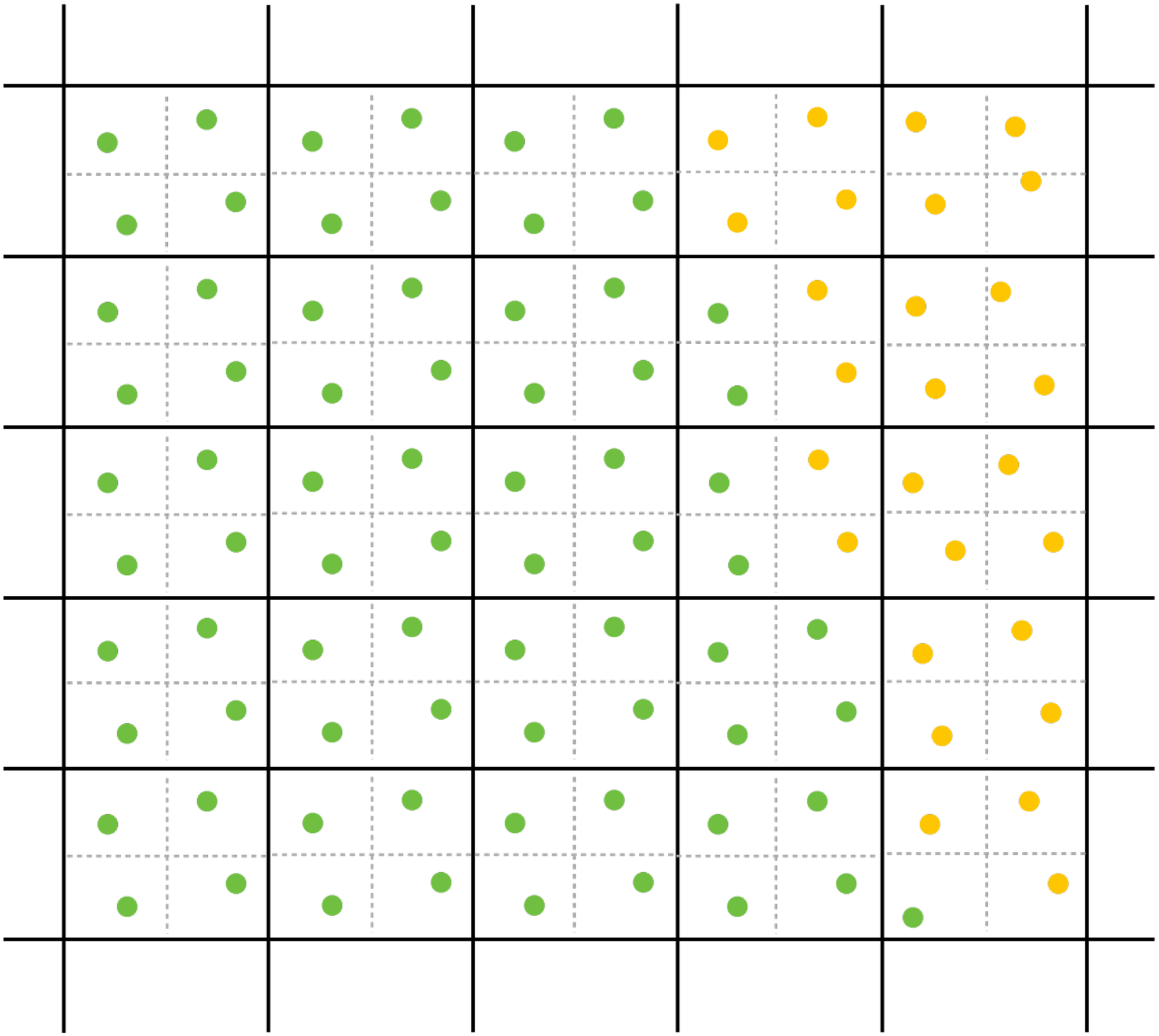
(Here: green triangle occludes yellow triangle)

Depth + Supersampling

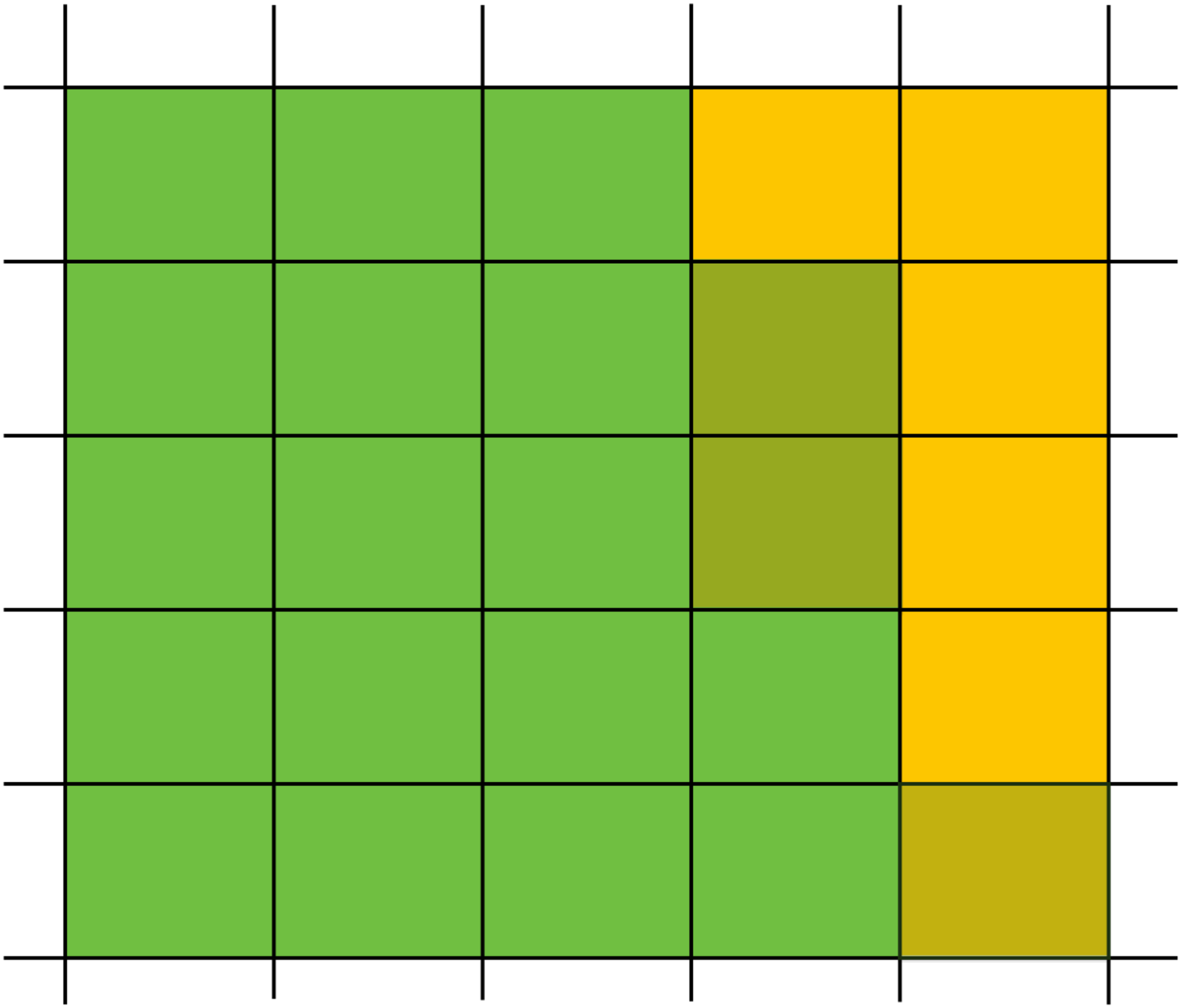


Color of super samples after rasterizing w/ depth buffer

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

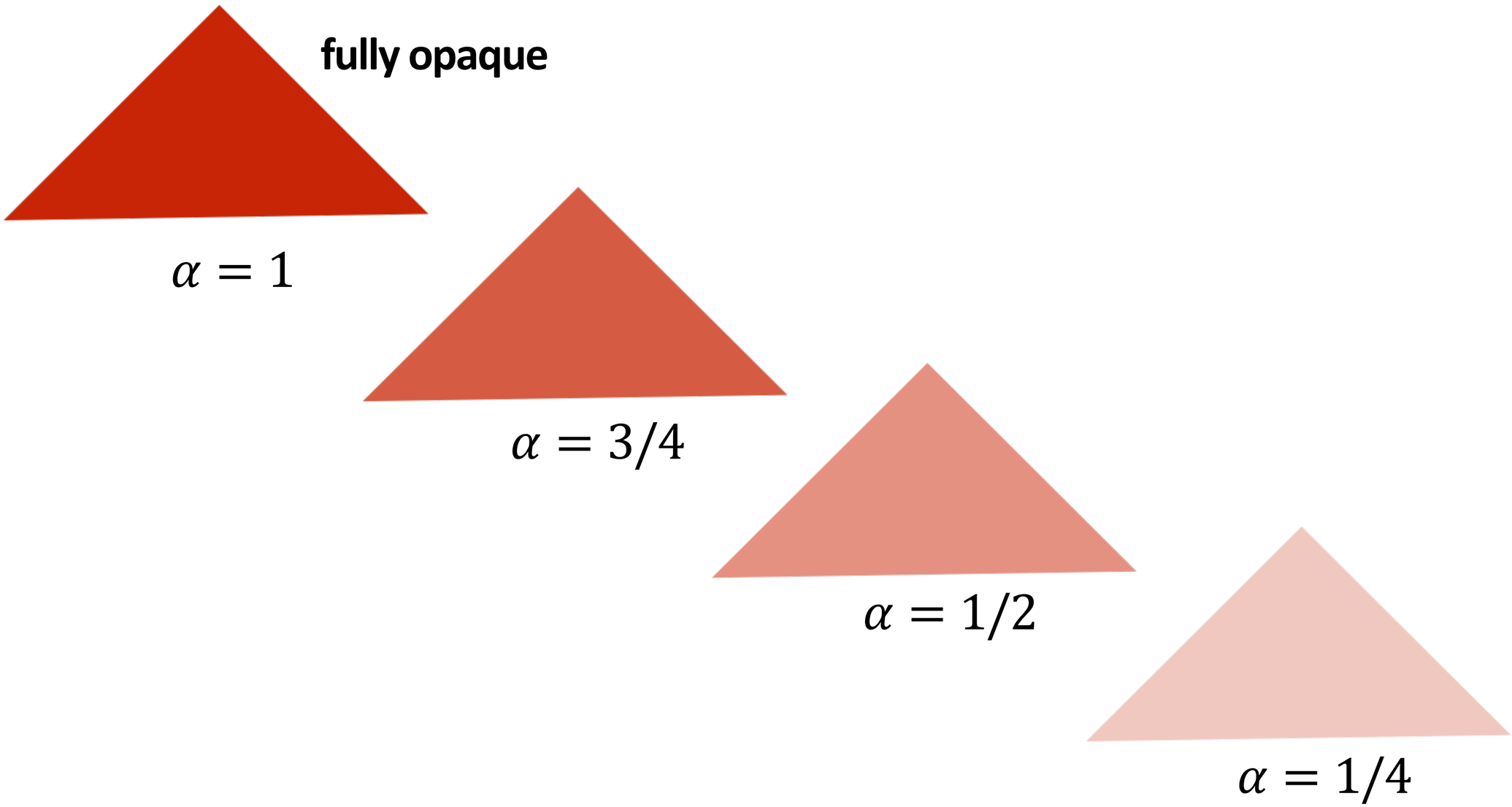
- Store one depth value per (super) sample—not one per pixel!
- Constant additional space per sample
  - Hence, constant space for depth buffer
  - Doesn't depend on number of overlapping primitives!
- Constant time occlusion test per covered sample
  - Read-modify write of depth buffer if "pass" depth 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

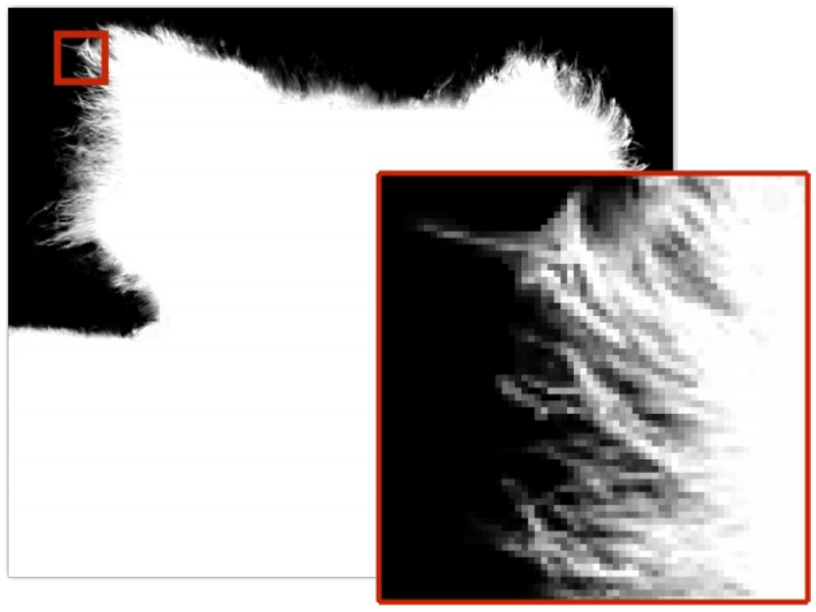
An "alpha" value $0 \leq \alpha \leq 1$ describes the <u>opacity</u> of an object



Alpha channel of an image

color channels

α channe

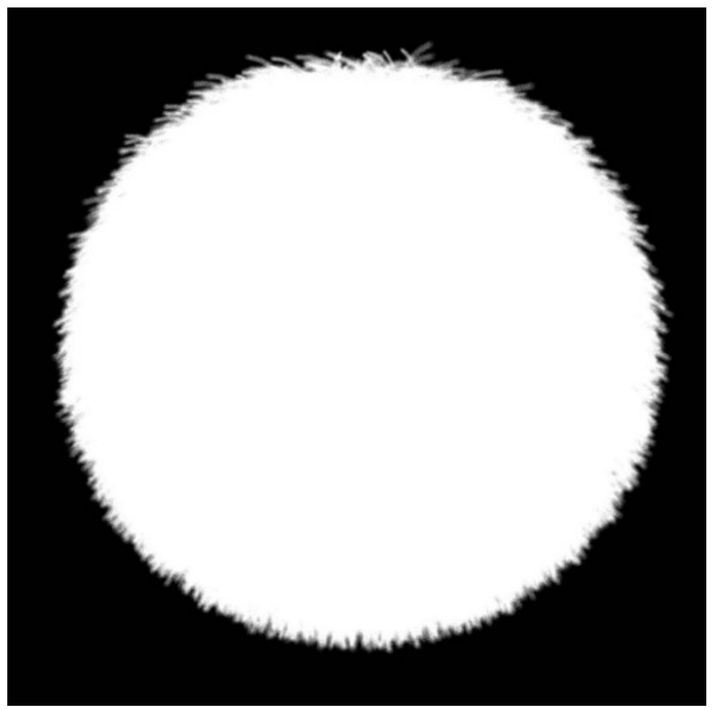


Key idea: can use α channel to composite one image on top of another.

Fringing

Poor treatment of color/alpha can yield dark "fringing":

foreground color



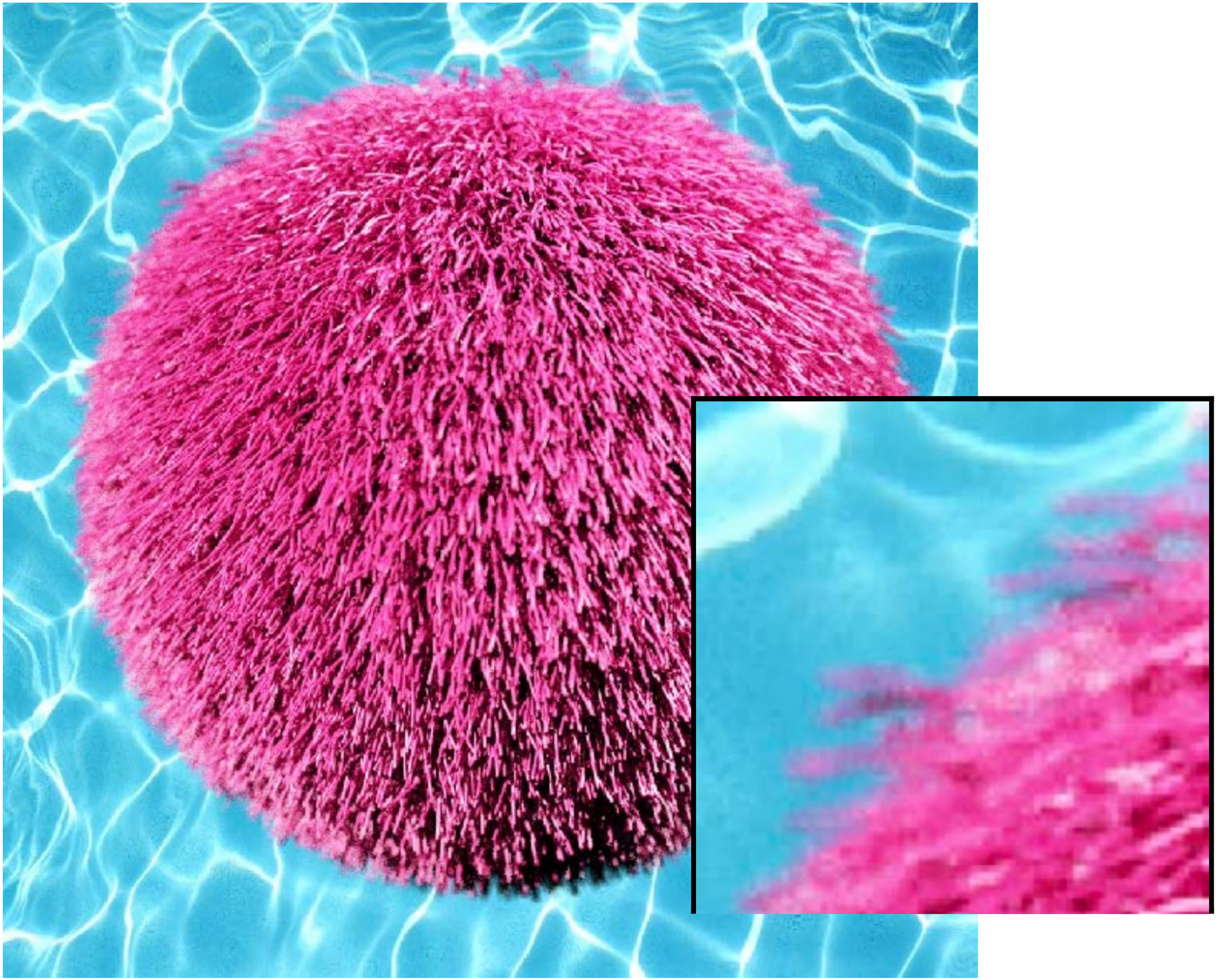
foreground alpha

background color

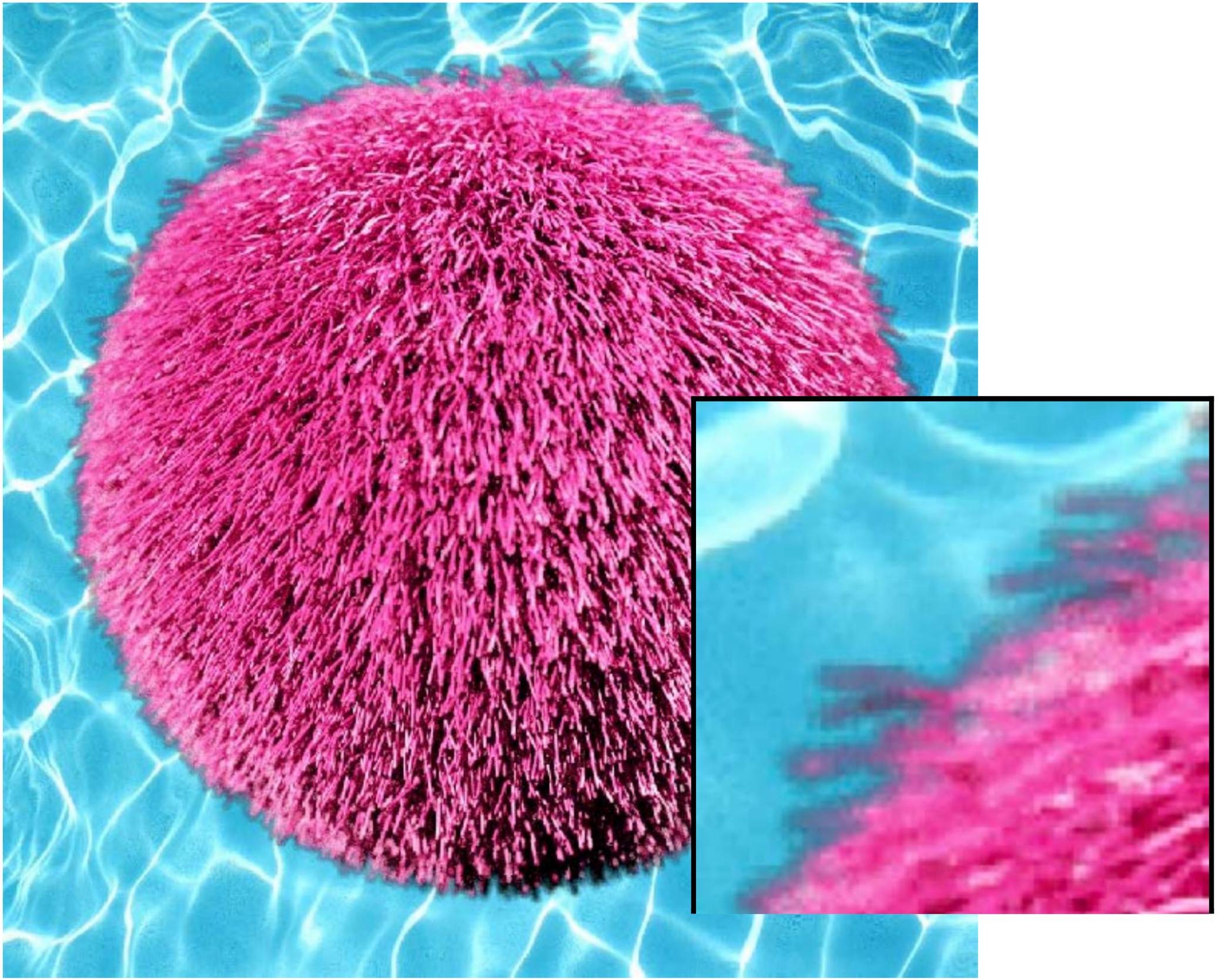
fringing

no fringing

No fringing



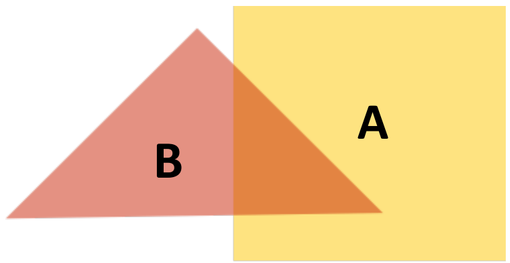
Fringing (…why does this happen?)



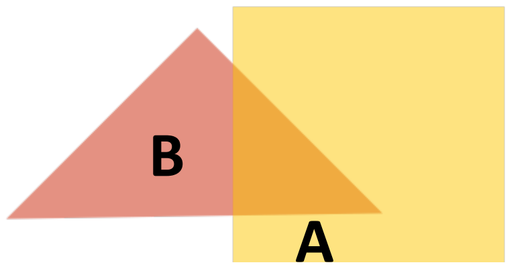
Over operator:

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

Informally, captures behavior of "tinted glass"



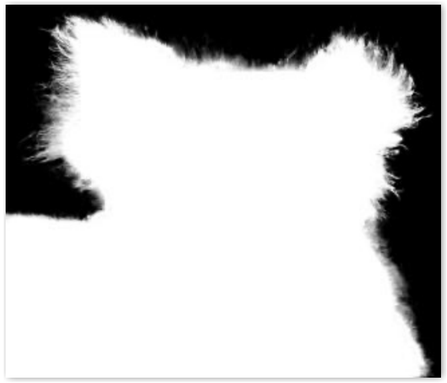
B **over** A



A **over** B

Notice: "over" is not commutative

$$A \; over \; B \neq B \; over \; A$$



Koala over NYC

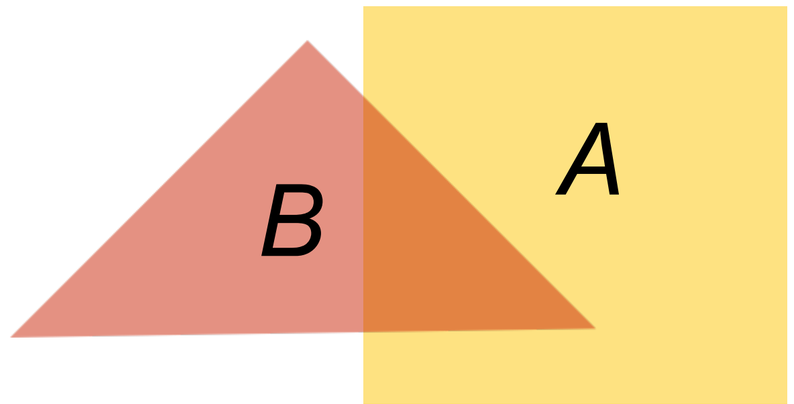
Porter & Duf "Compositing Digital Images" (1984)

Over operator: non-premultiplied alpha

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

A first attempt:

$$A = (A_r, A_g, A_b)$$
$$B = (B_r, B_g, B_b)$$



B **over** A

<u>Composite color:</u>

what B lets through

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

appearance of semi-transparent B

appearance of semi-transparent A

<u>Composite alpha:</u>

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

Over operator: premultiplied alpha

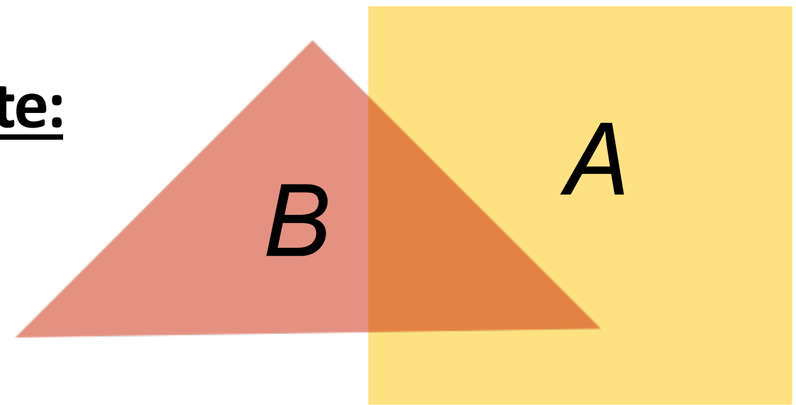
Composites image B with opacity α_B over image A with opacity α_A

Permultiplied alpha-----multiply color by α, then <u>composite:</u>

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

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

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



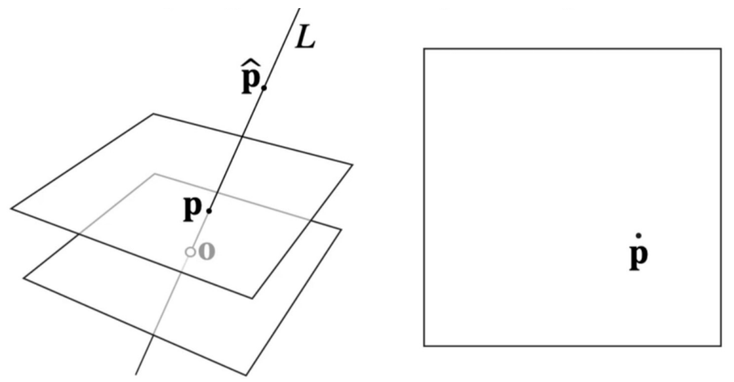
B **over** A

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

"Un-premultiply" to get final color

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

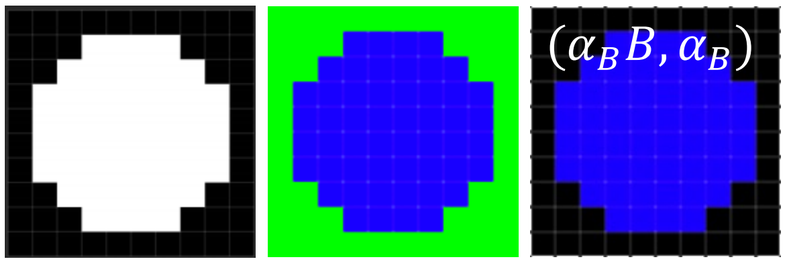
Q: Does this division remind you of anything



Compositing with & without premultiplied α

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

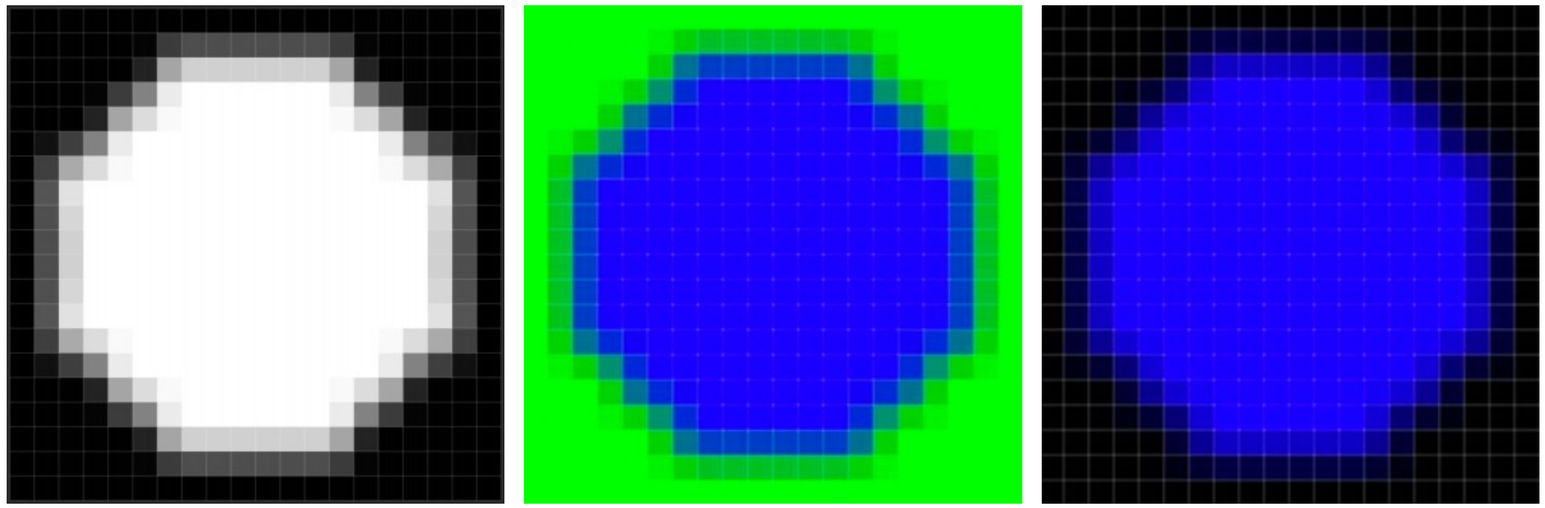
image B



original alpha

original color

premultiplied color

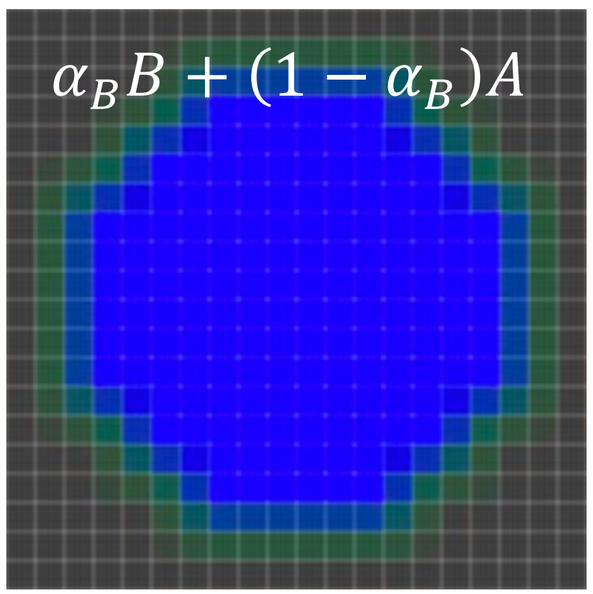


upsampled alpha

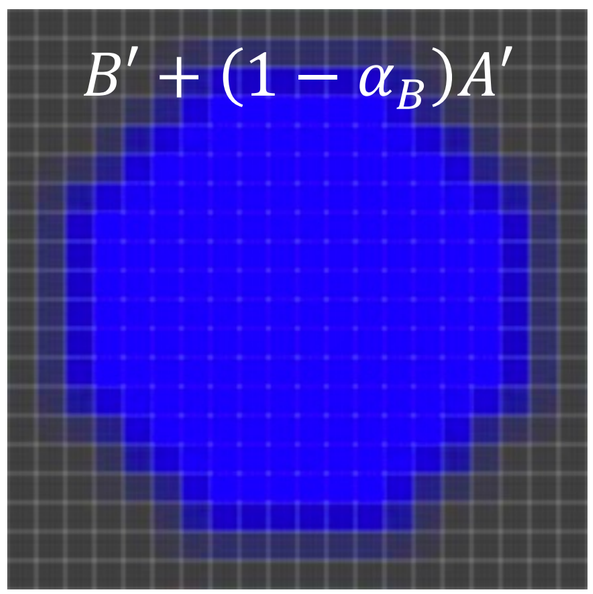
upsampled color

upsampled premultiplied color

new background A $(\alpha_A = 1)$



B over A non-premultiplied



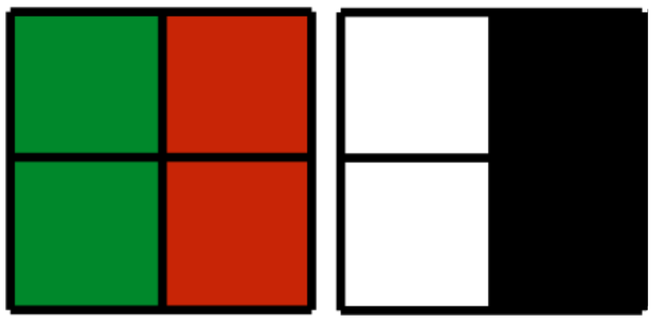
B over A premultiplied

Q: Why do we get the "green fringe" when we don't premultiply?

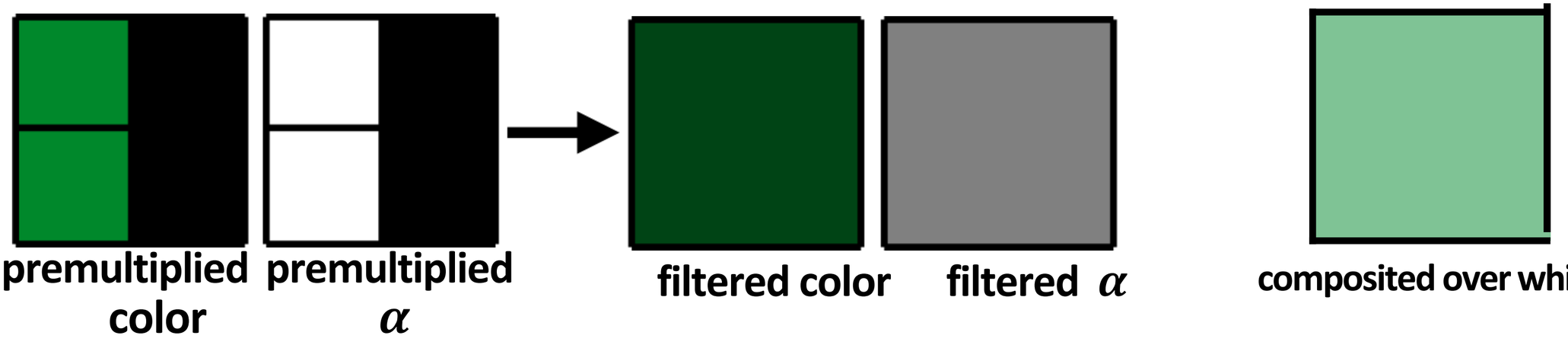
Similar problem with non-premultiplied α

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

desired downsampled result



input color | input α | filtered color | filtered α | composited over white

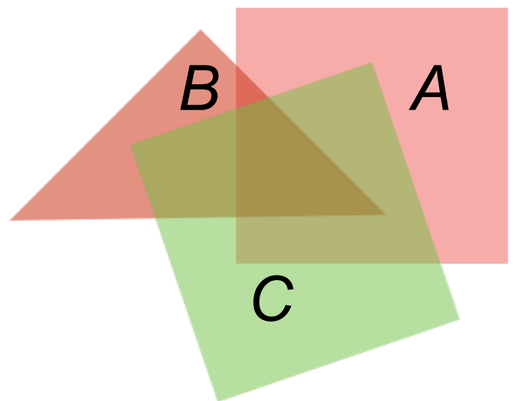


premultiplied color | premultiplied α | filtered color | filtered α | composited over white

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!



C **over** B **over** A

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

non-premultiplied

color

$$0.5(1,0,0) + (1 - 0.5)0.5(1,0,0)$$

$$\downarrow$$

$$(0.75,0,0)$$

too dark!

alpha

$$0.5 + (1 - 0.5)0.5 = 0.75$$

premultiplied

color

$$(0.5,0,0,0.5) + (1 - 0.5)(0.5,0,0.5)$$

$$\downarrow$$

$$(0.75,0,0,0.75)$$

$$\downarrow$$ divide by α

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