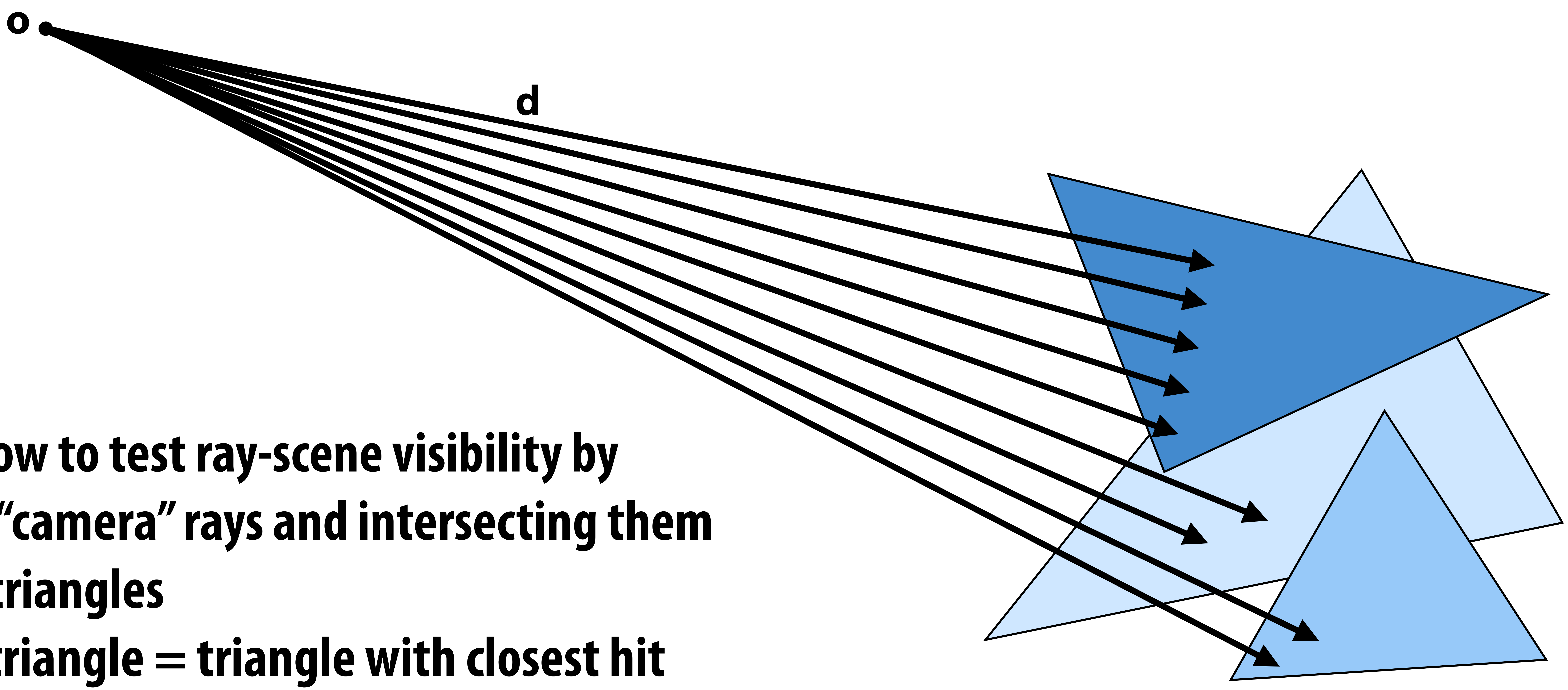
Lecture 7:

The Rasterization Pipeline

(and its implementation on GPUs)

Computer Graphics: Rendering, Geometry, and Image Manipulation
Stanford CS248A, Winter 2026

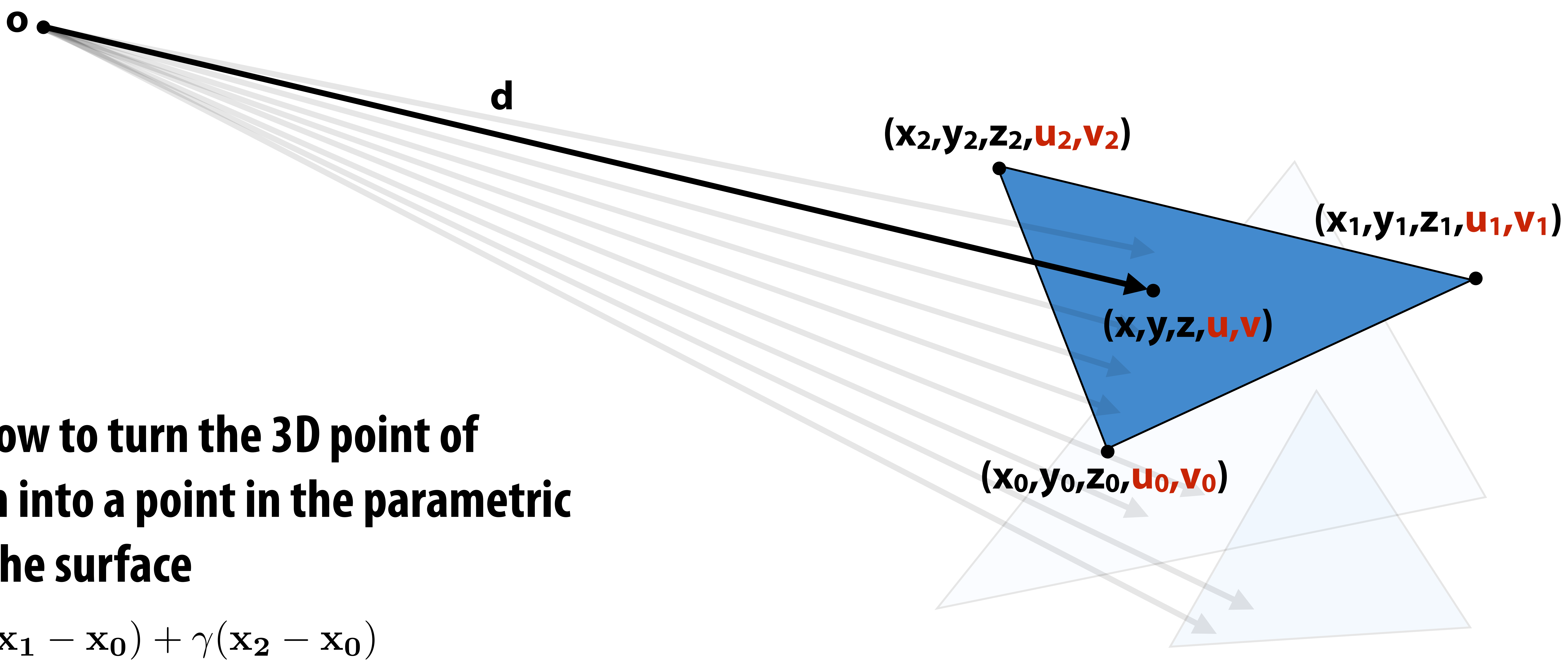
Review: raycasting to compute visibility



You know how to test ray-scene visibility by generating "camera" rays and intersecting them with scene triangles

- **visible triangle = triangle with closest hit**

Review: determining the location of the hit point (in the domain of the surface)

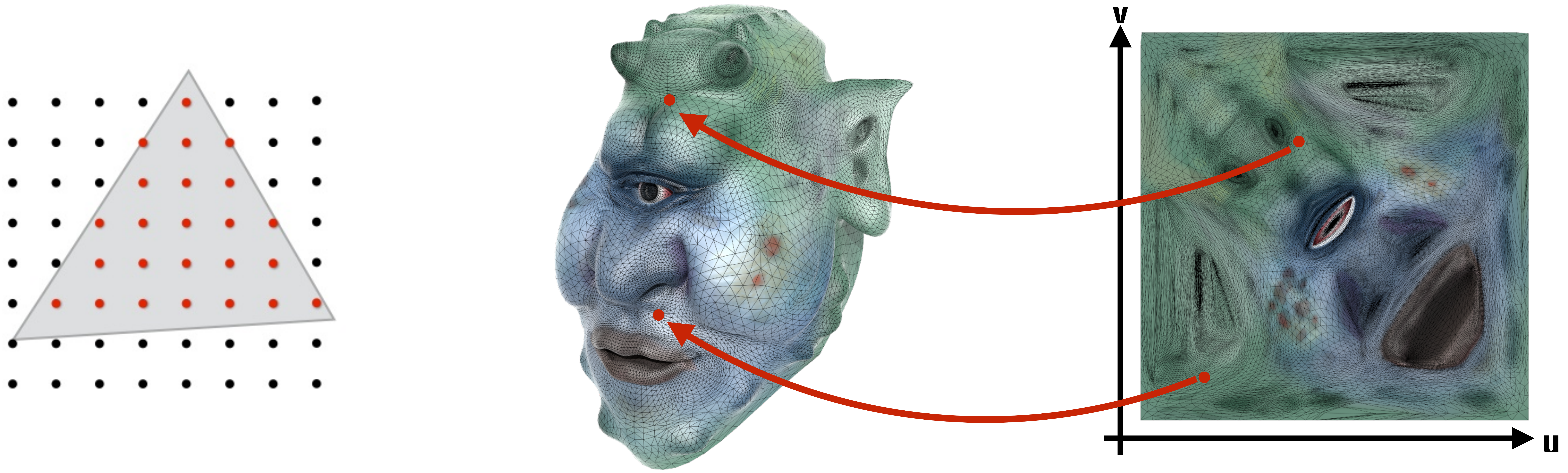


You know how to turn the 3D point of intersection into a point in the parametric domain of the surface

$$\begin{aligned}\mathbf{x} &= \mathbf{x_0} + \beta(\mathbf{x_1} - \mathbf{x_0}) + \gamma(\mathbf{x_2} - \mathbf{x_0}) \\ &= (1 - \alpha - \beta)\mathbf{x_0} + \beta(\mathbf{x_1} - \mathbf{x_0}) + \gamma(\mathbf{x_2} - \mathbf{x_0}) \\ &= \alpha\mathbf{x_0} + \beta\mathbf{x_1} + \gamma\mathbf{x_2}\end{aligned}$$

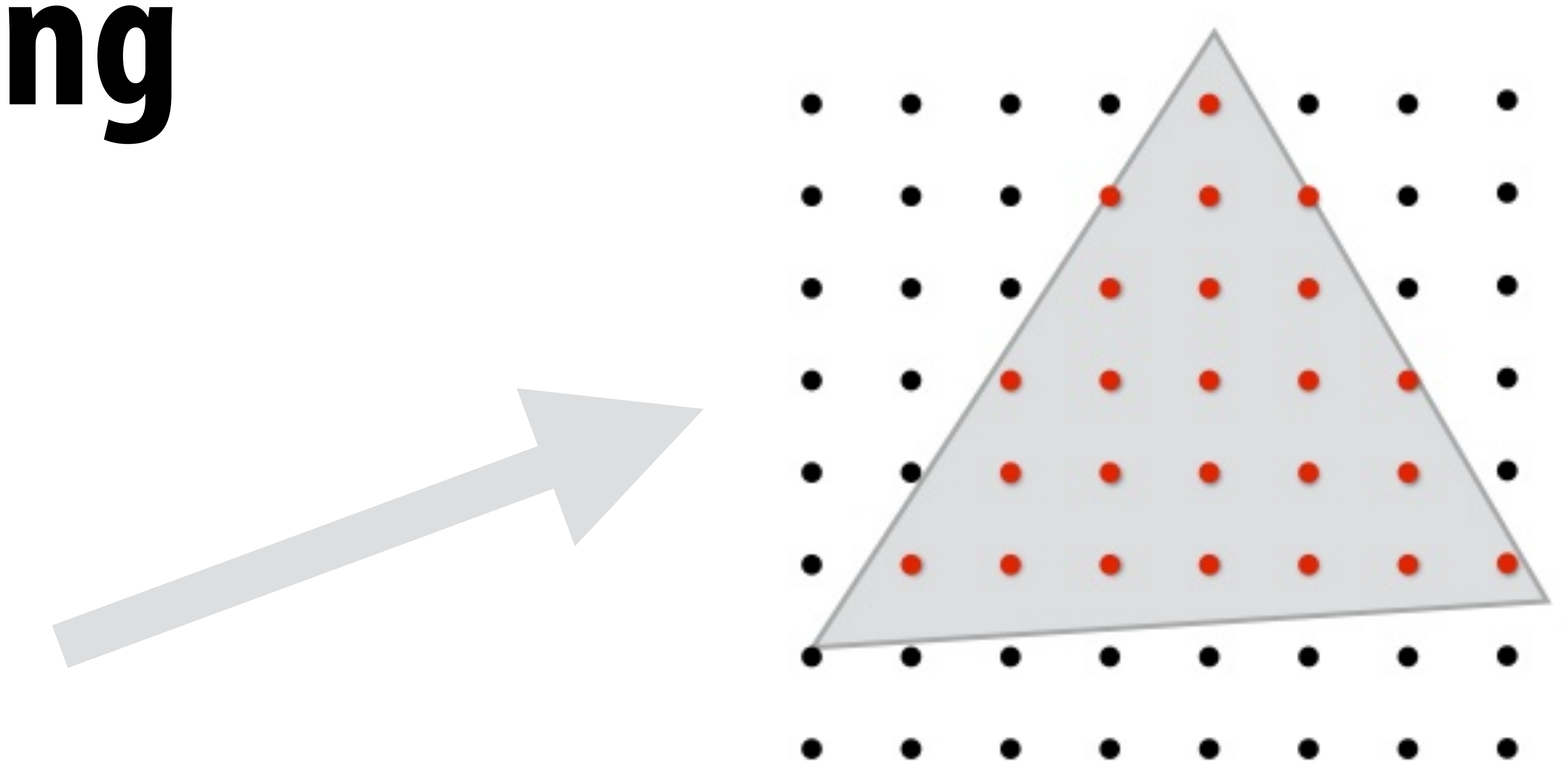
Review

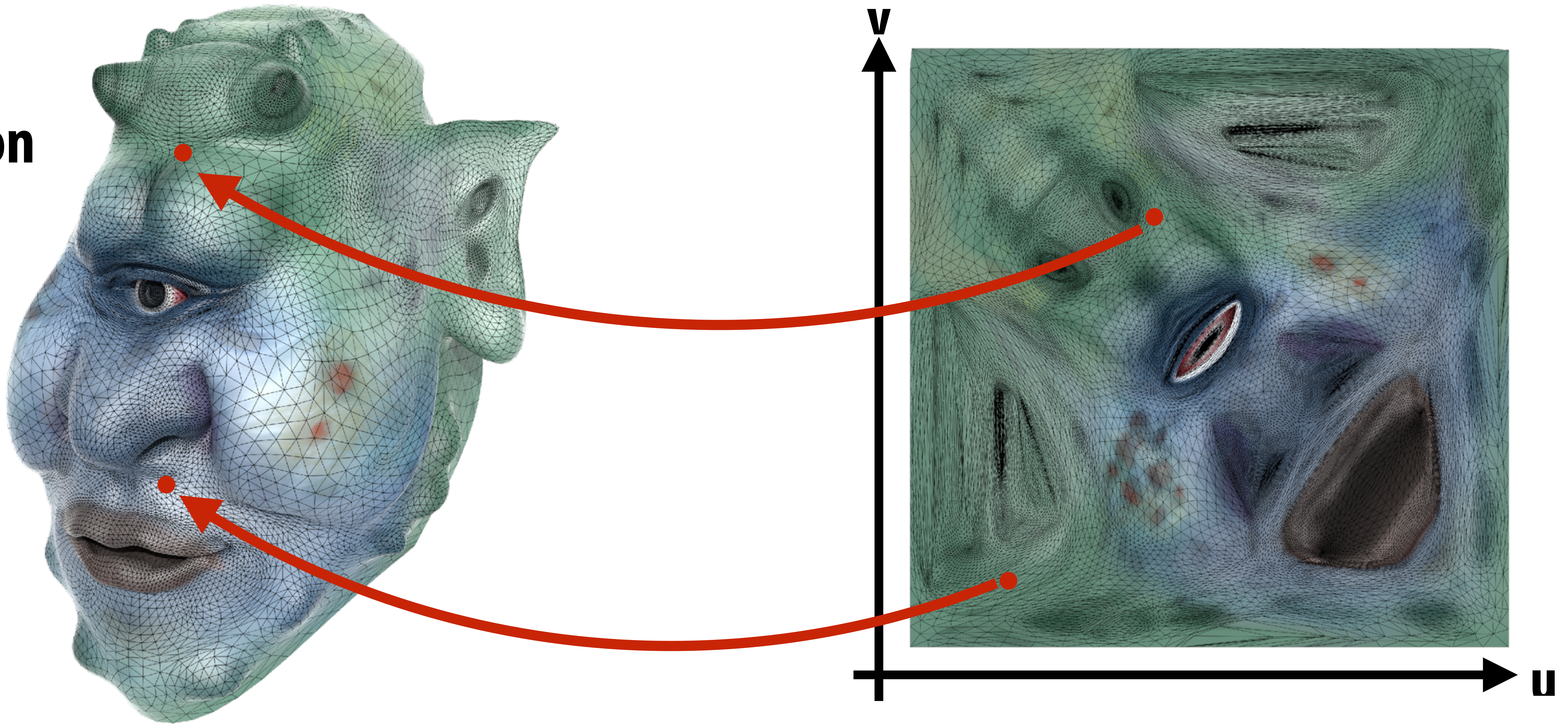
And so we have a mapping between 2D screen sample points … to 3D points on the surface … to the 2D parametric domain of the surface.



Review: texture mapping

- A texture is a function that defines a value (e.g., a surface color) at every point on a surface:
  - texture(u,v)
- When drawing a surface, was sample the screen's 2D domain (x,y)
- Each sample (x,y) corresponds to a hit point on a triangle, which corresponds to a point (u,v) on the hit surface. We color the sample according to texture(u,v)
- So we are sampling the texture function uniformly in screen (x,y)… but potentially non-uniformly in texture space (u,v)





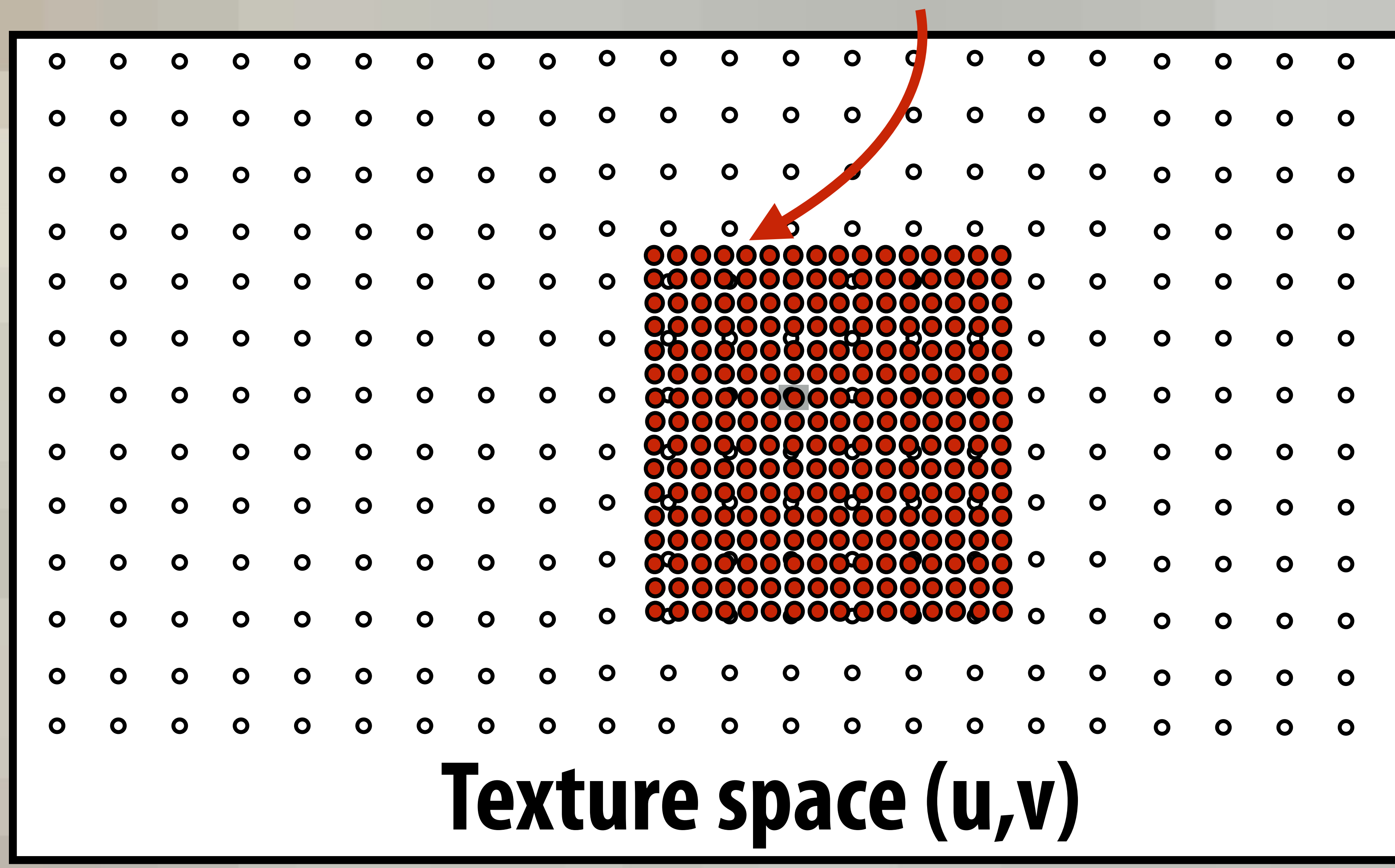
Review: potential for aliasing

- Depending on the position/orientation of the camera and surface, the texture function might be under sampled or oversampled

Consider this texture map, applied to a plane

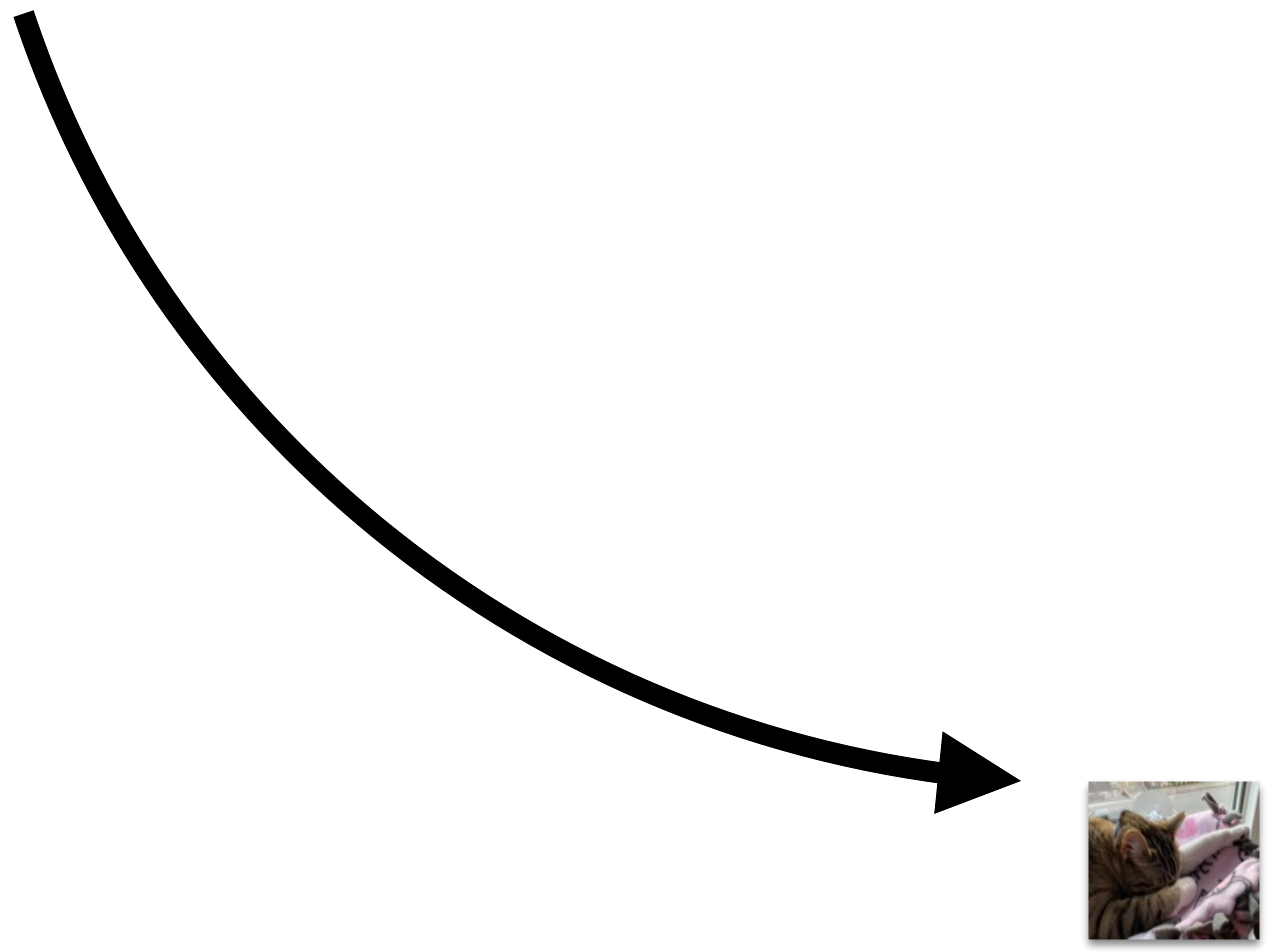
Review: texture magnification (using nearest neighbor filtering)

In this slide, we're heavily zoomed into the plane.



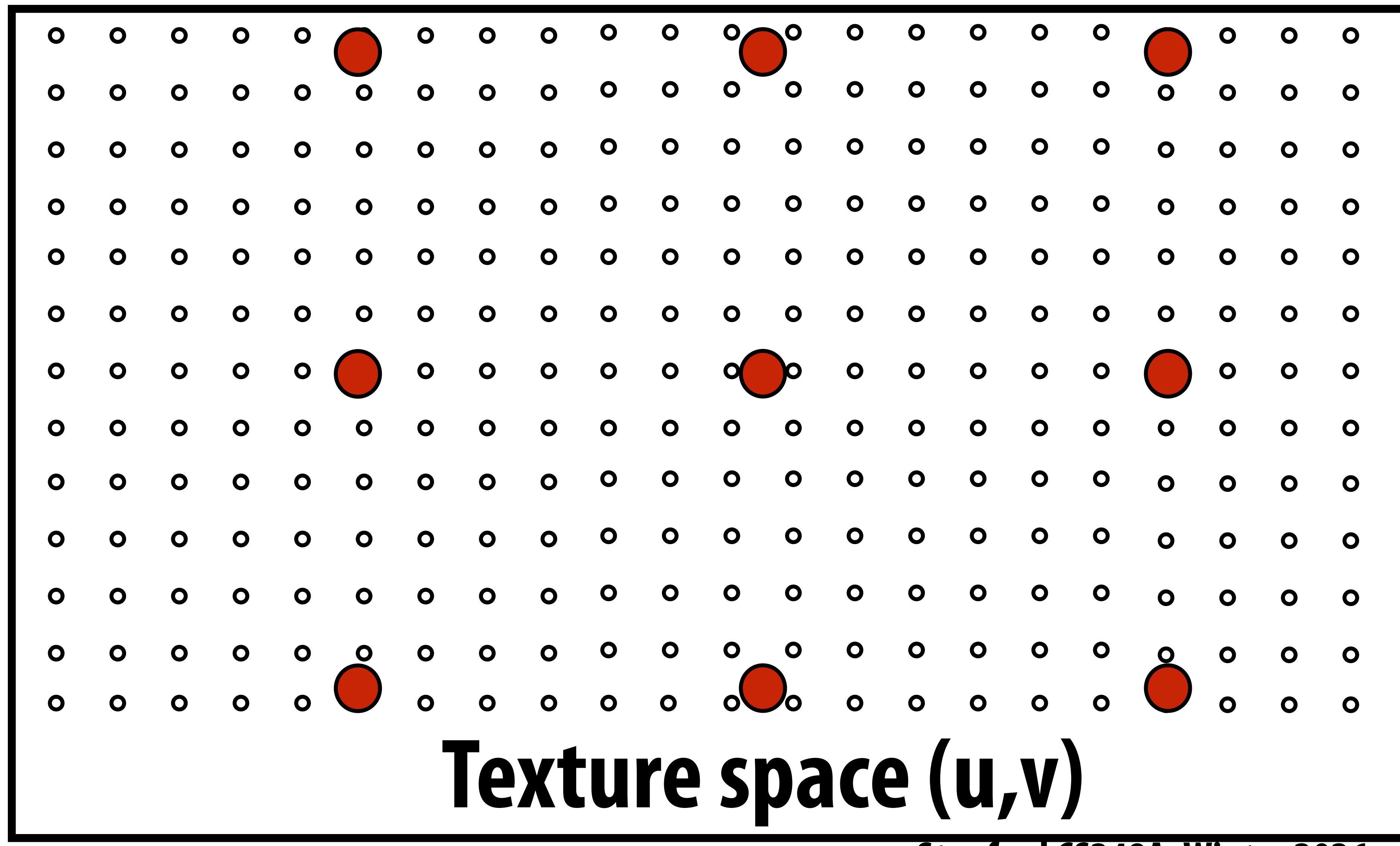
Review: texture minification

Now image the surface with the Josephine texture map is very far away from the camera



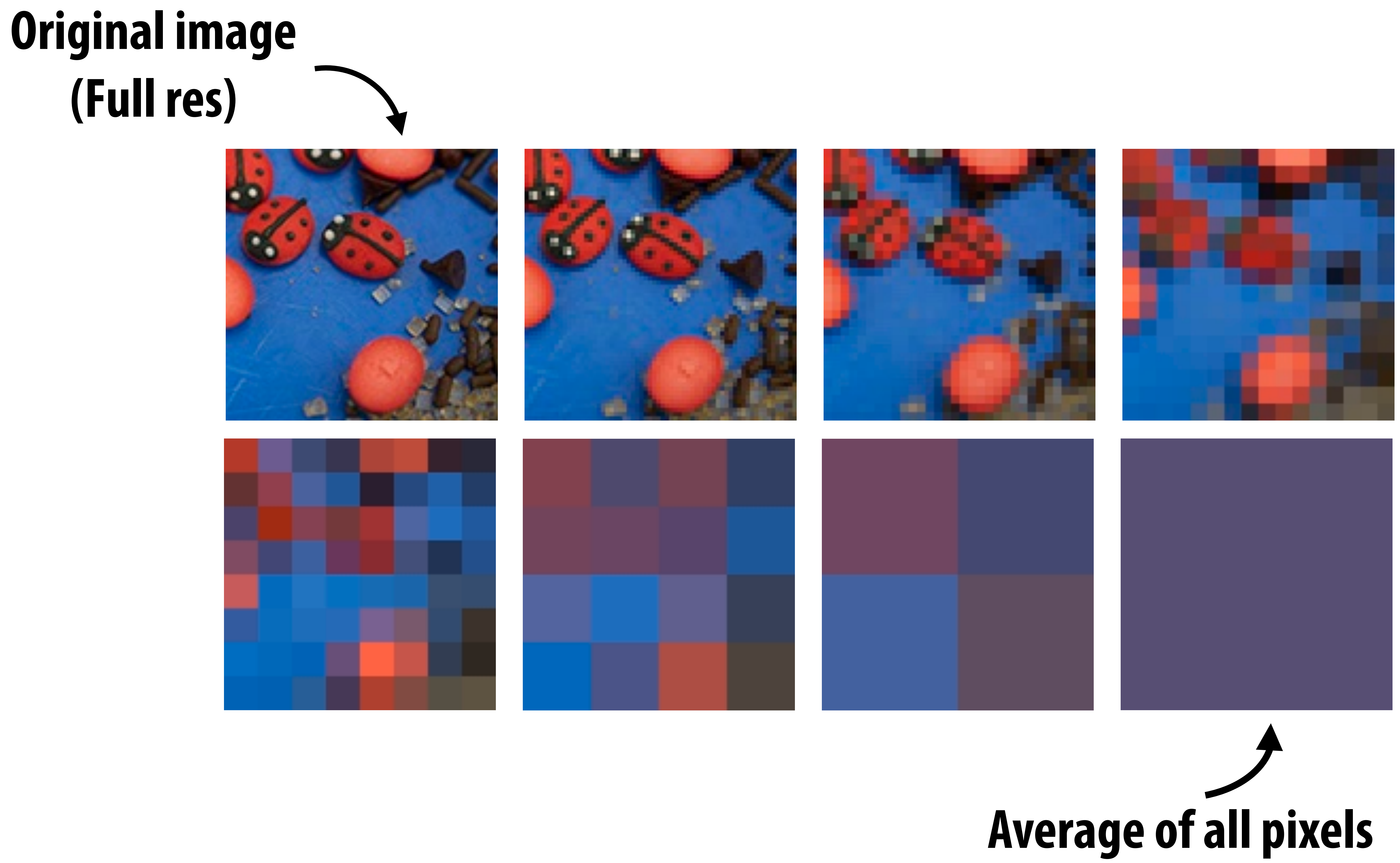
If texture(u,v) has high frequency content, then aliasing could occur. (And in this case it does have very high frequency content — see detail in cat fur)

Red dots = location of screen samples in texture space



Review: key idea - mip mapping

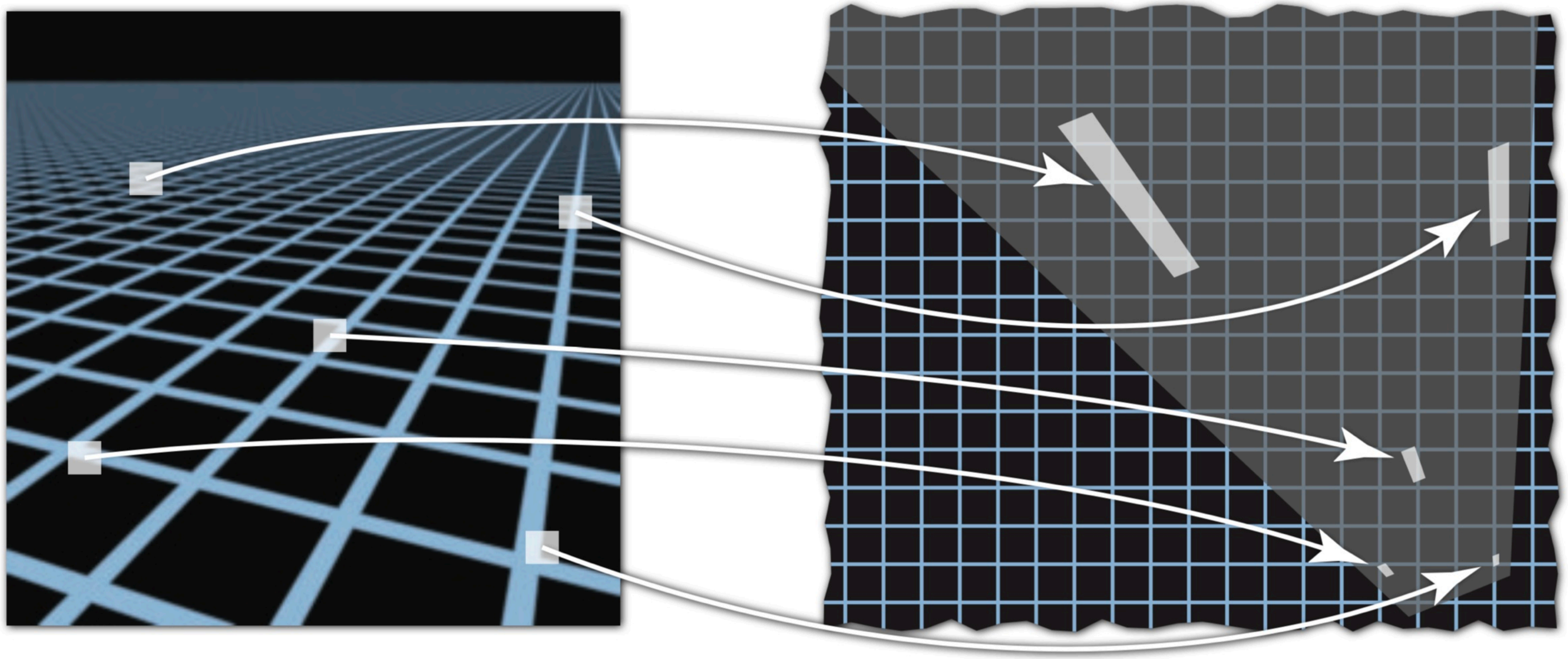
- Remove high frequencies from the texture map prior to sampling
- But since plane can be viewed from many different distances in an interactive graphics application, we precompute many different amounts of filtering



Dynamically choose which image to sample from based on change in u and v across adjacent screen samples

In a raytracer, du/dx, du/dy, dv/dx, dv/dy can be computed using "ray differentials" which you'll learn about in assignment 2.

Review: a sample in a mip-map level is a precomputed average over a square region of texture(u,v)… but in practice we seek to filter over non-isotropic regions of texture space

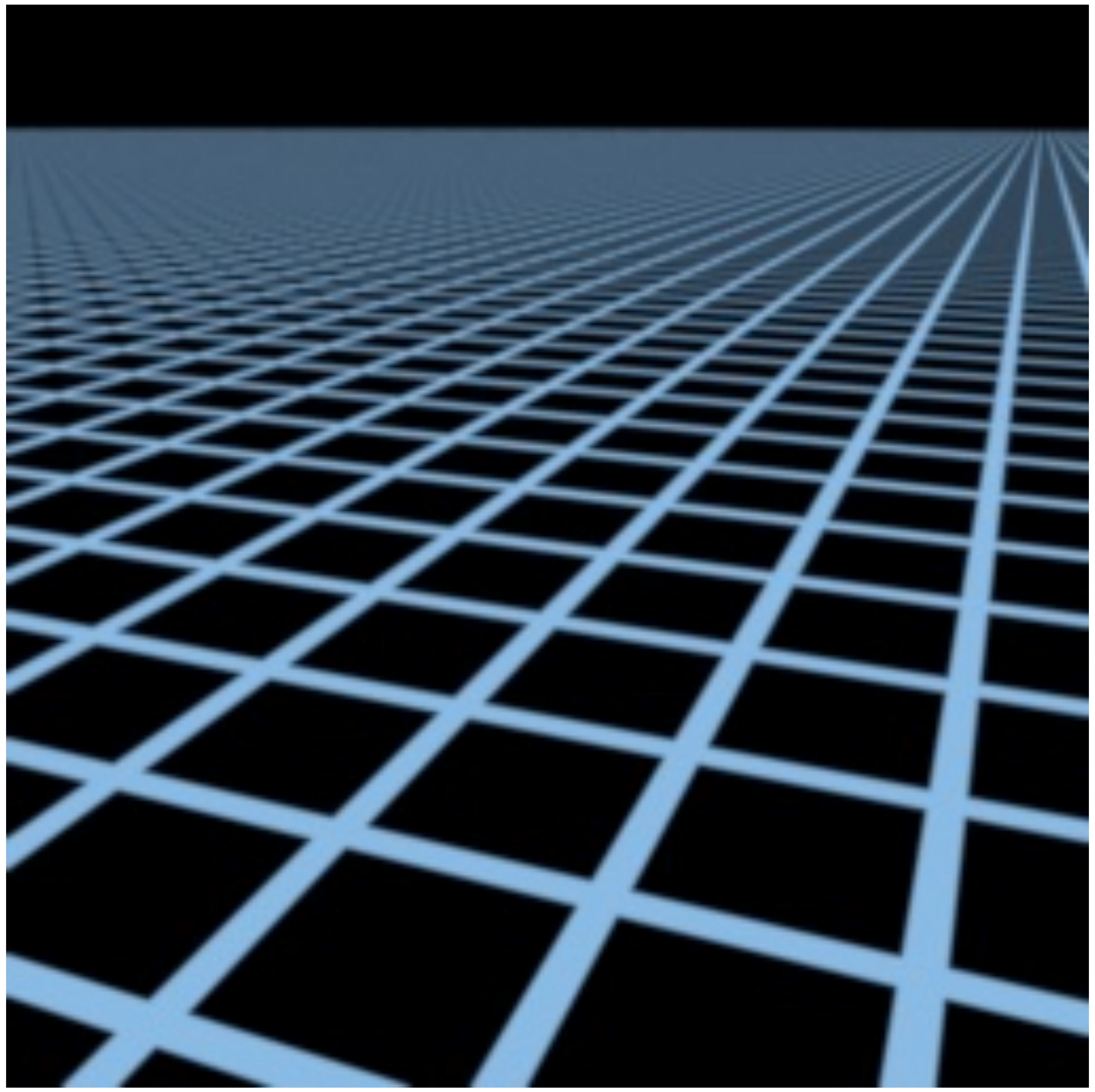


Screen space

Texture space

Texture sampling pattern not rectilinear or isotropic

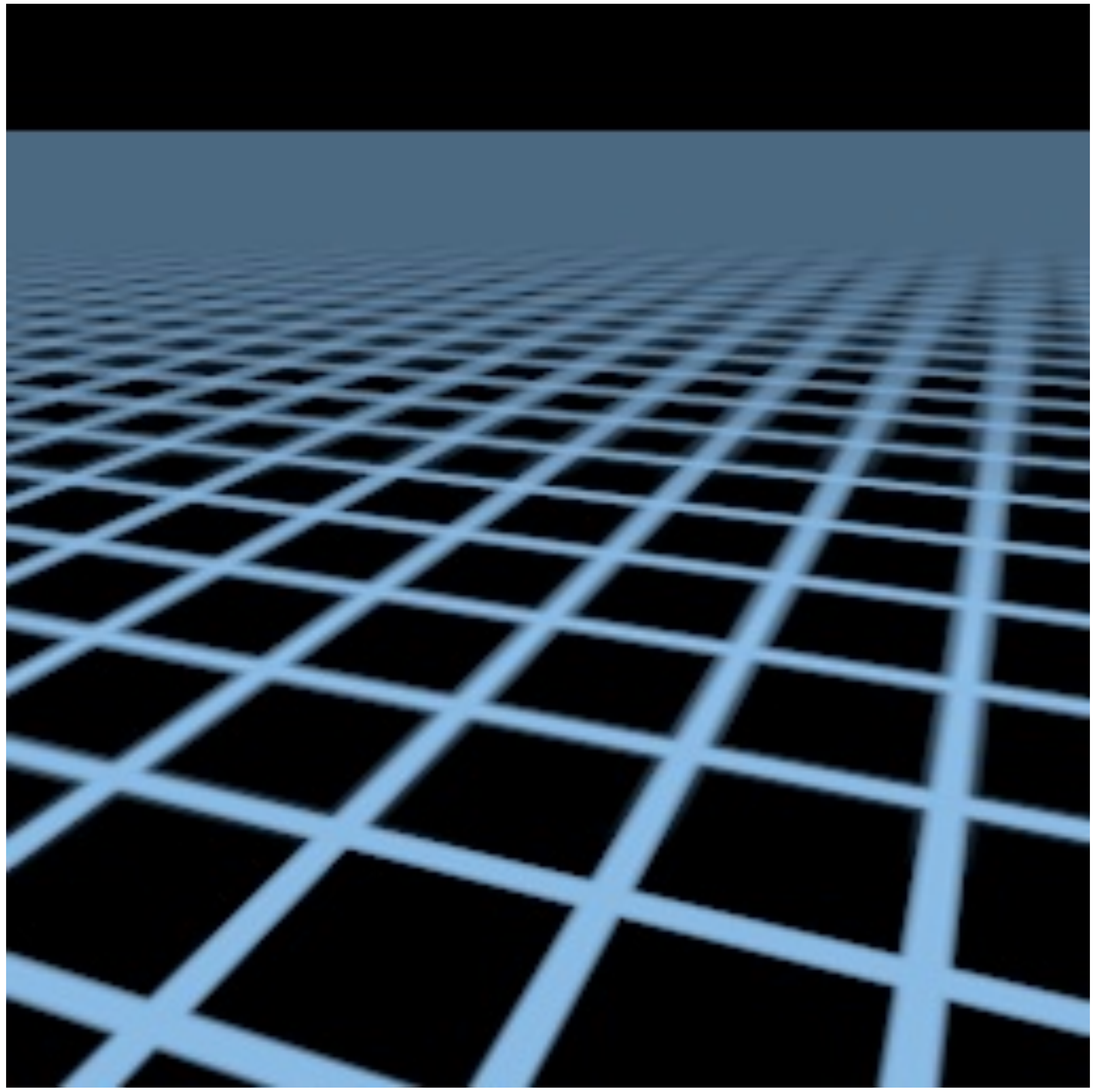
Example: mipmap limitations



Supersampling: 512 texture samples per pixel
(desired answer)

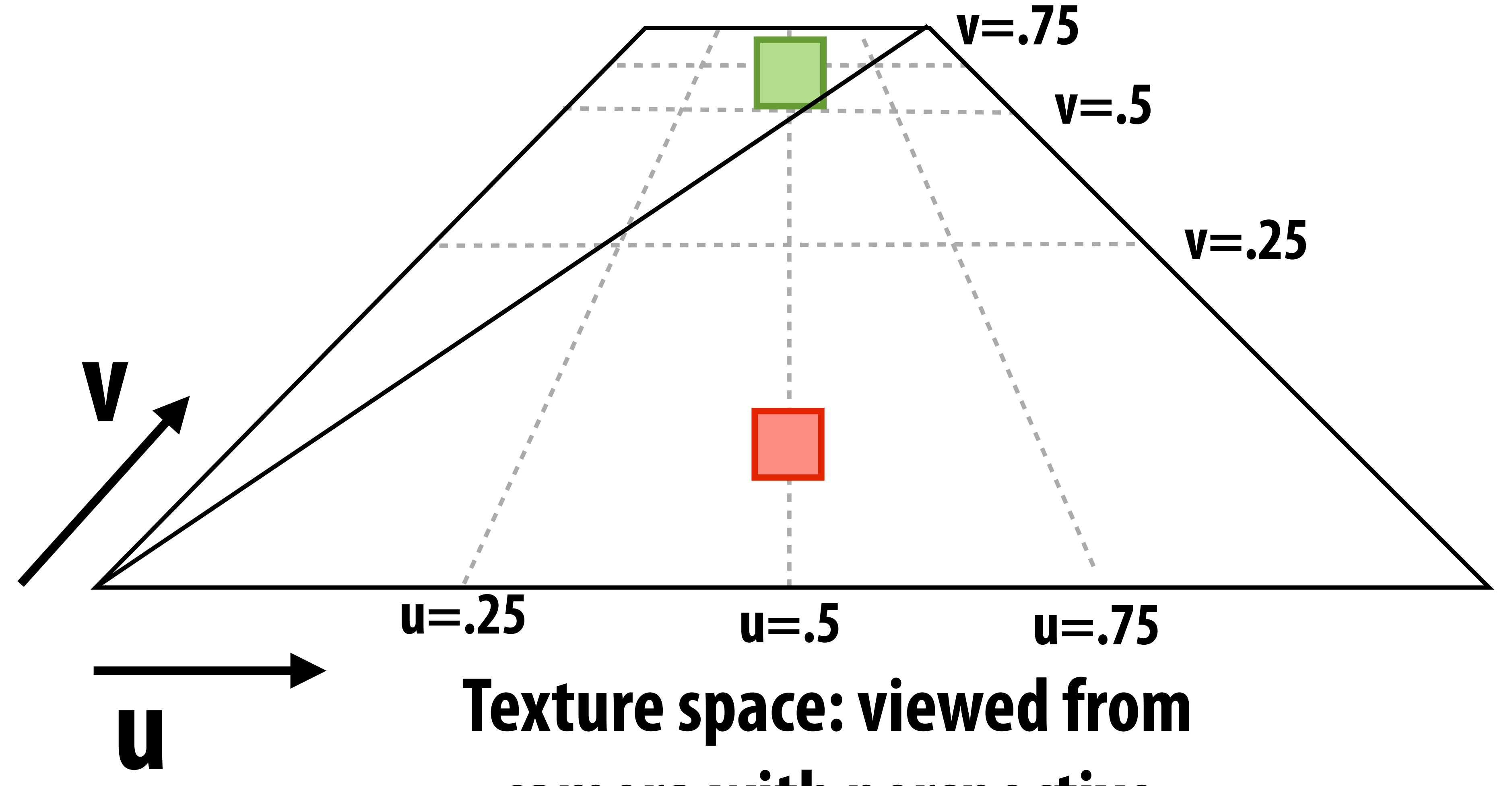
Example: mipmap limitations

Overblurs
Why?

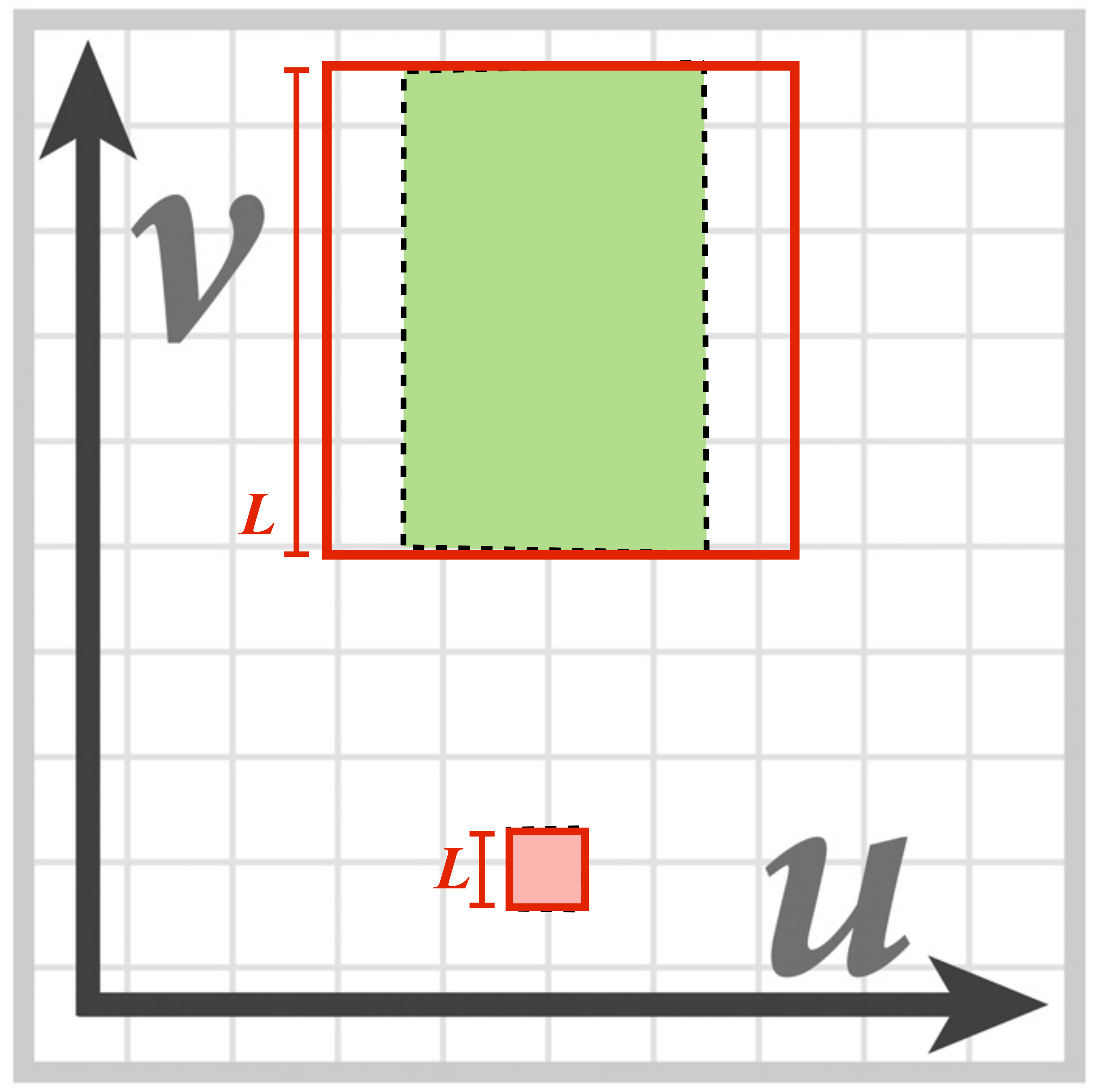


Mipmap trilinear sampling

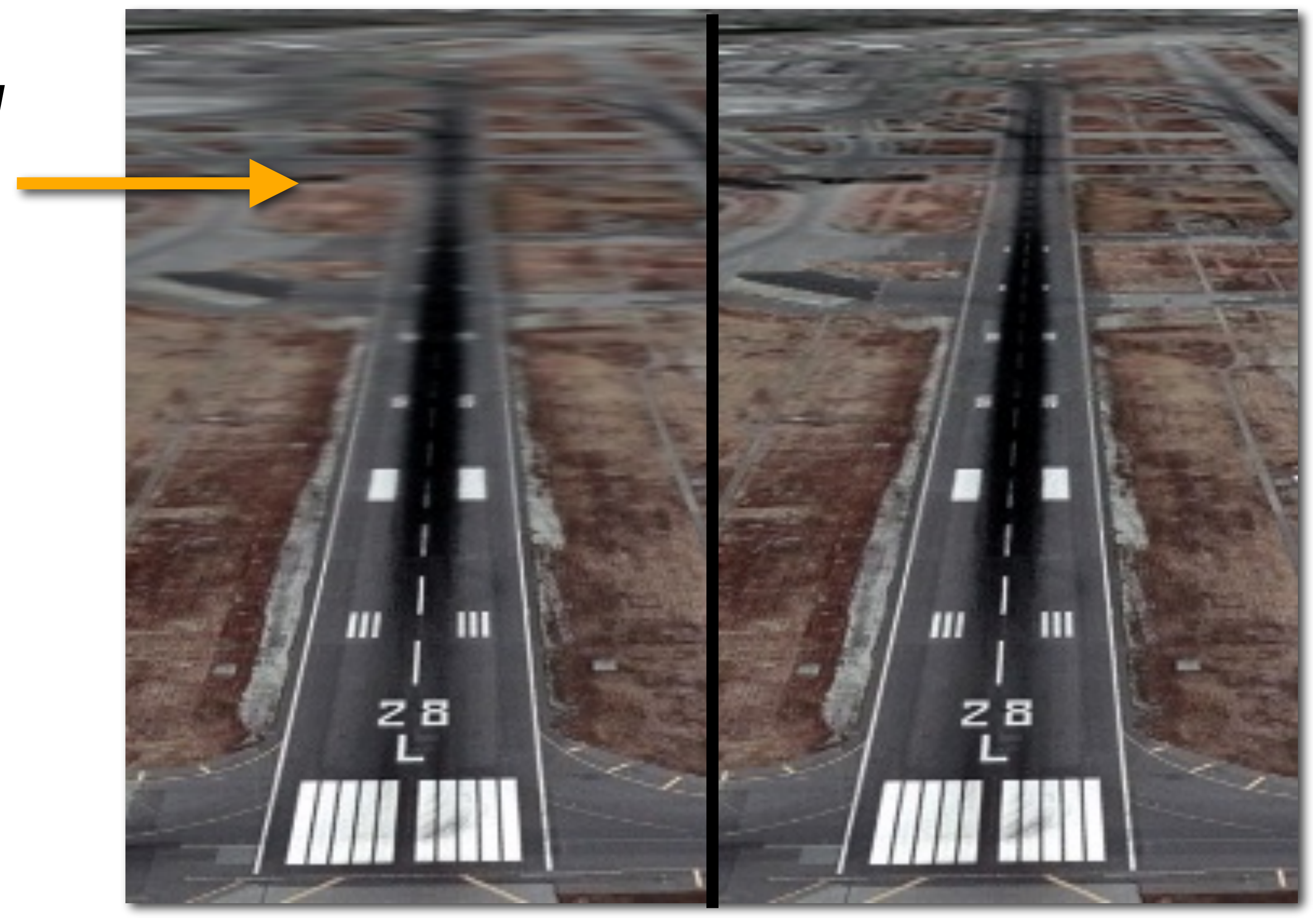
Proper texture filtering requires anisotropic filter footprint



Texture space: viewed from camera with perspective



Overblurring in *u* direction



Trilinear (Isotropic) Filtering

Anisotropic Filtering

(Modern anisotropic texture filtering solutions combine multiple mip map samples to approximate integral of texture value over arbitrary texture space regions)

Today's topic: the rasterization pipeline

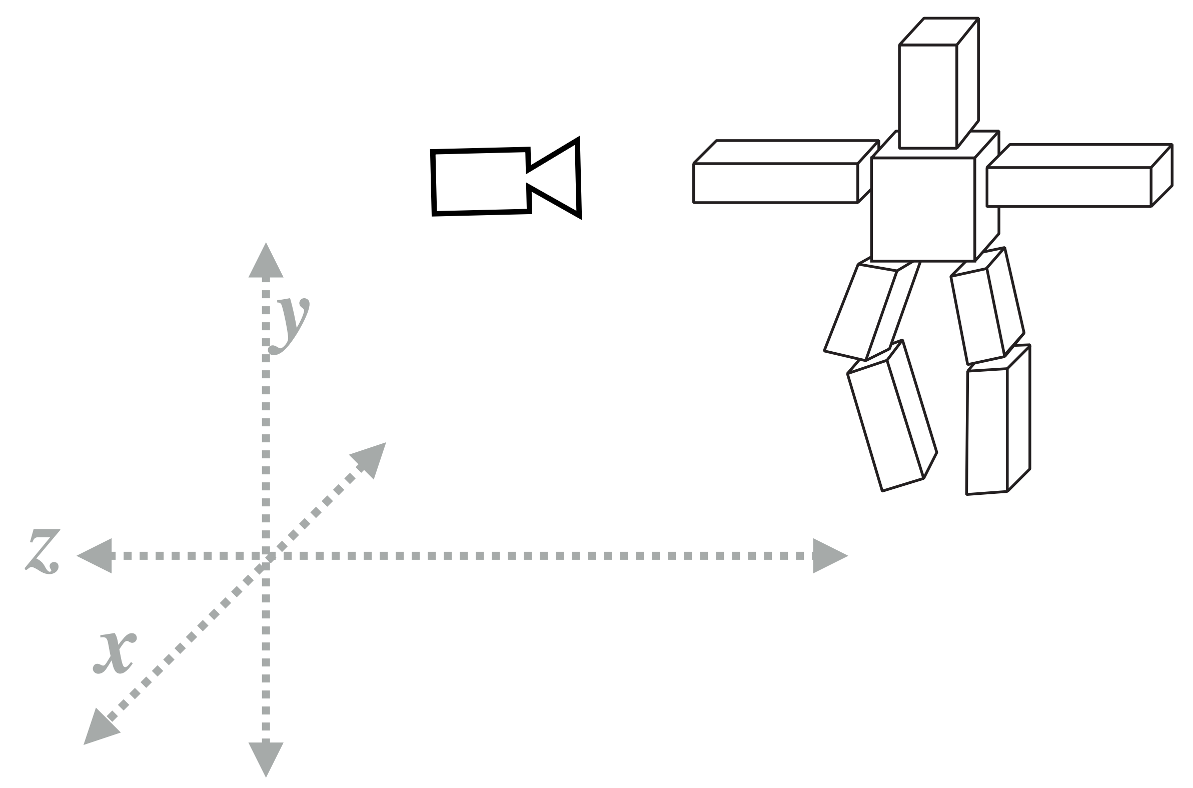
- Your homework assignments in this class, you implement raytracing based rendering that simulates image formation by a pinhole camera

- But we've seen how rasterization is an alternative way to perform the same calculation

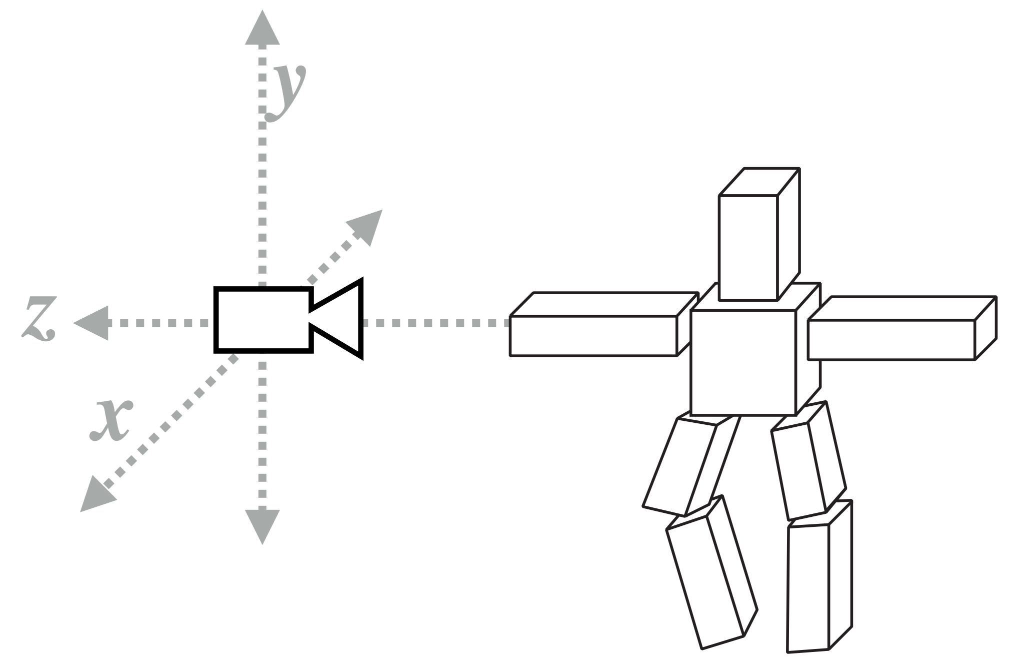
- Rasterization has long been used by most interactive graphics systems due to its high efficiency, and amenability to acceleration via customized hardware that's been present in GPUs for decades
  - And has long be supported by graphics APIs like OpenGL, Direct3D, and Vulkan

- So today we're going to focus on the details of the rasterization pipeline and how its implemented by modern GPUs

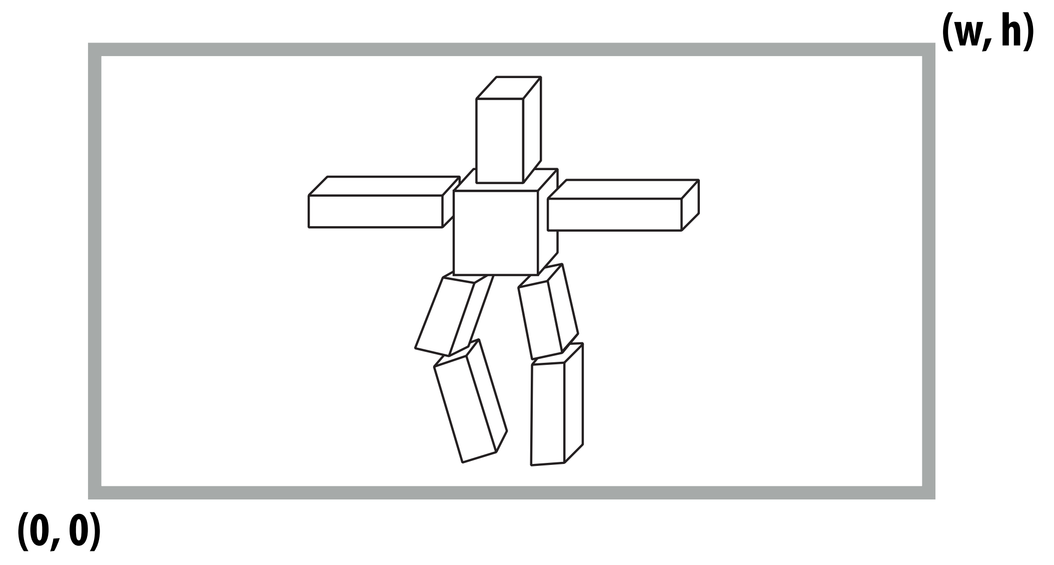
What you know how to do (at this point in the course)



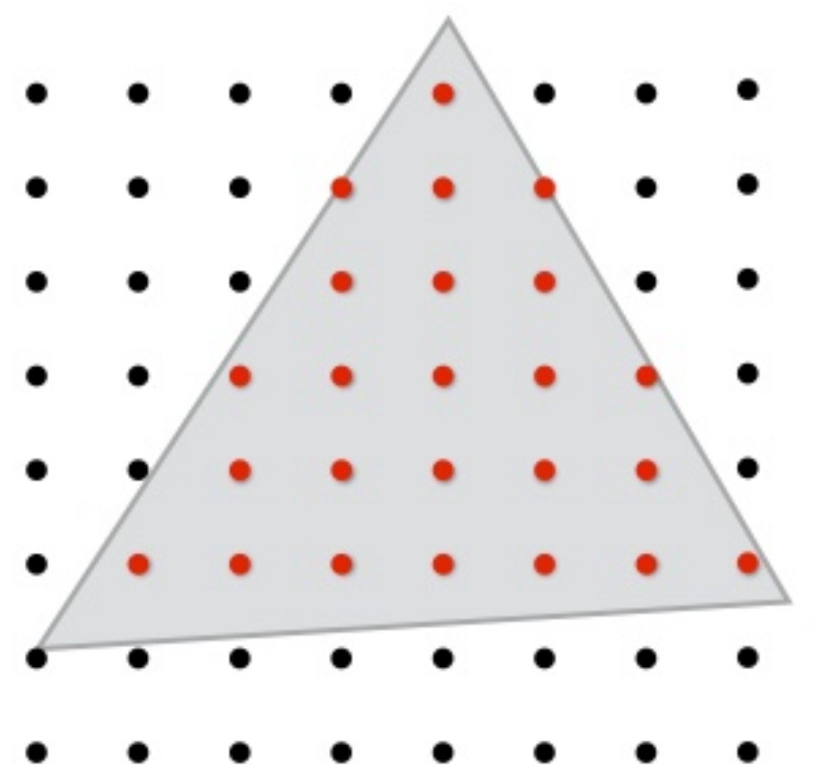
Position objects and the camera in the world



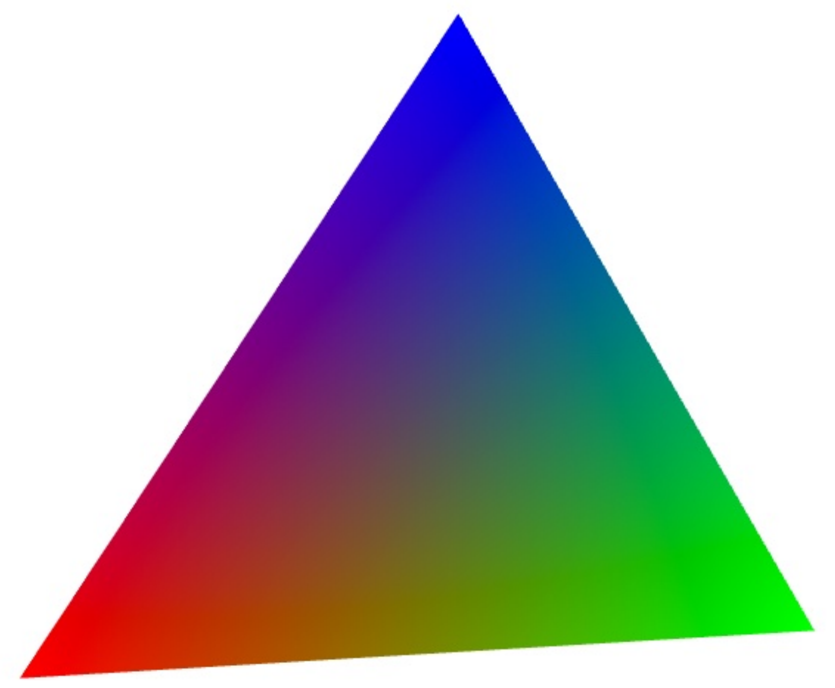
Determine the position of objects relative to the camera



Project objects onto the screen



Sample triangle coverage

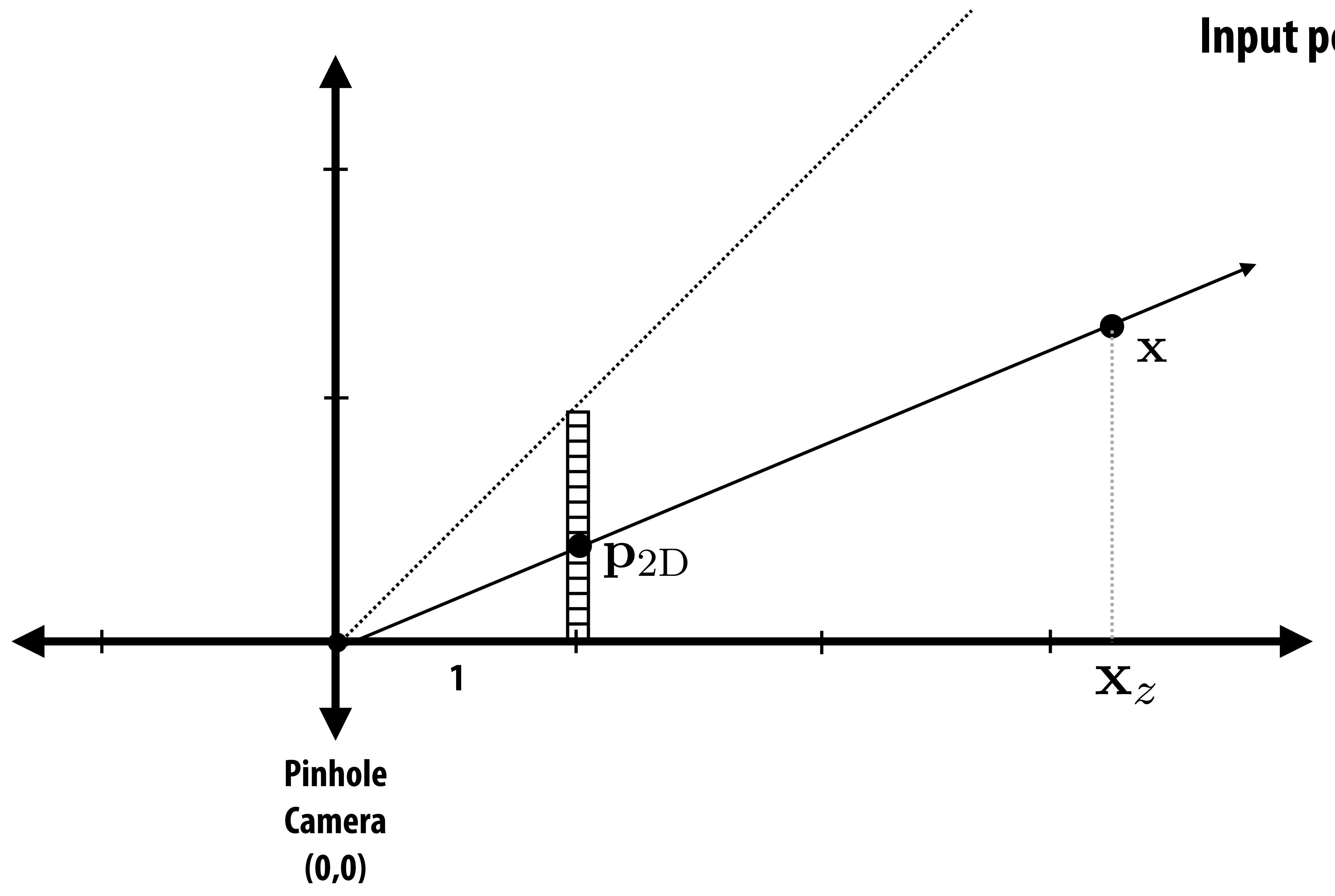


Compute triangle attribute values at covered sample points (Color, texture coords, depth)

Sample texture maps

One more detail on perspective projection

Basic perspective projection



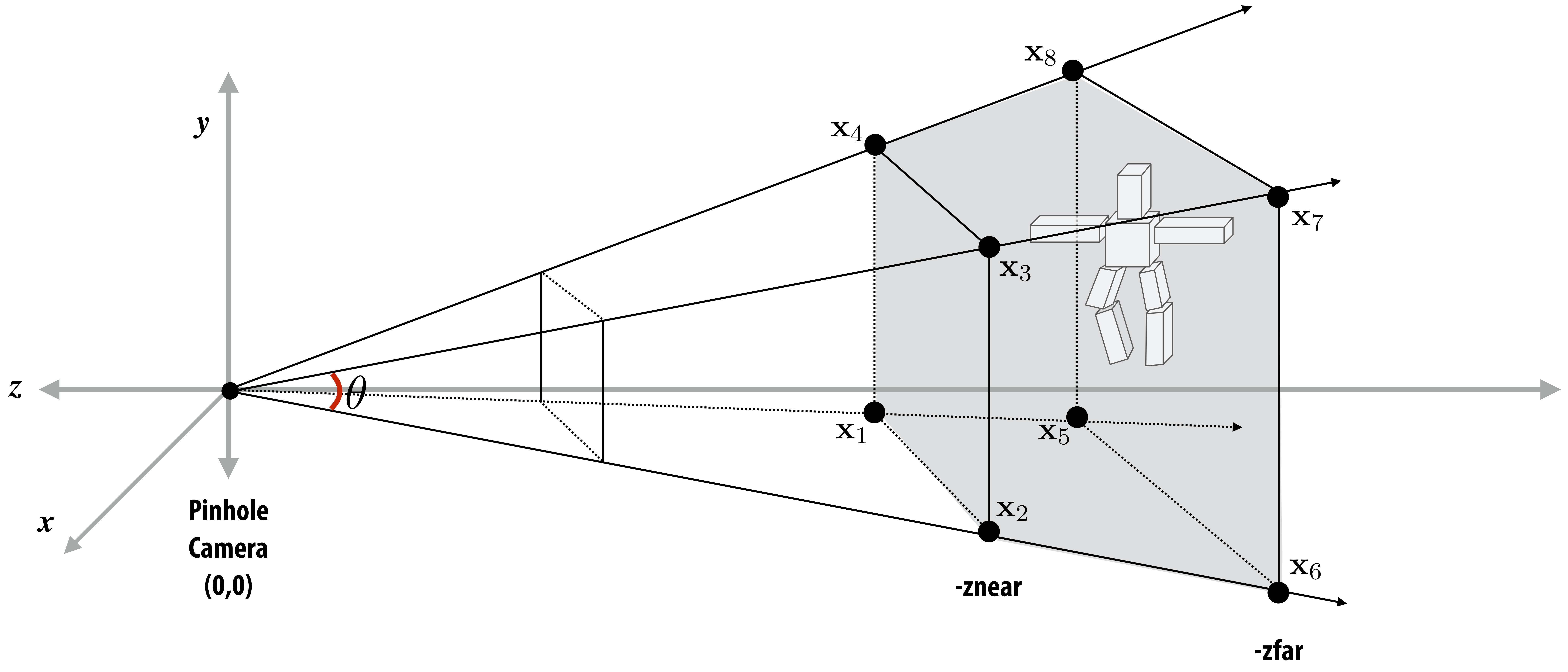
Input point in 3D-H: $\mathbf{x} = \begin{bmatrix} \mathbf{x}_x & \mathbf{x}_y & \mathbf{x}_z & 1 \end{bmatrix}^T$

$$\mathbf{P} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix}$$

Assumption:
Pinhole camera at (0,0) looking down z

View frustum

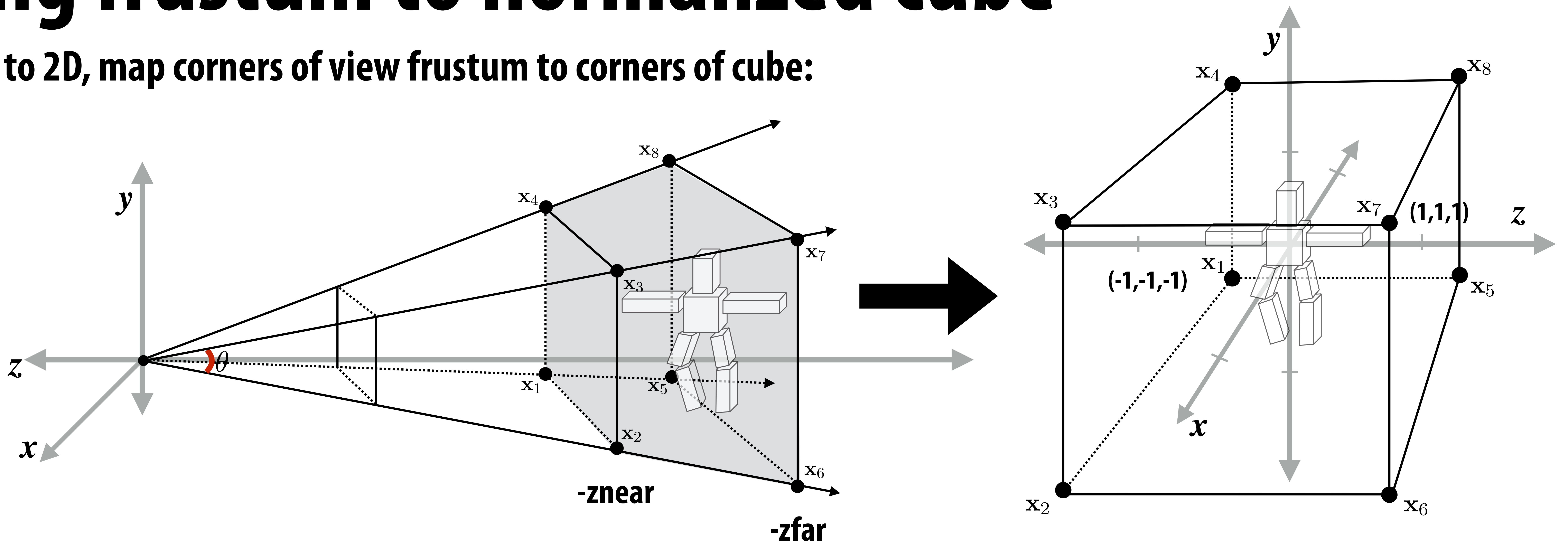
View frustum is the region of space the camera can see:



- **Top/bottom/left/right planes correspond to sides of screen**
- **Near/far planes correspond to closest/furthest thing we want to draw**

Mapping frustum to normalized cube

Before moving to 2D, map corners of view frustum to corners of cube:



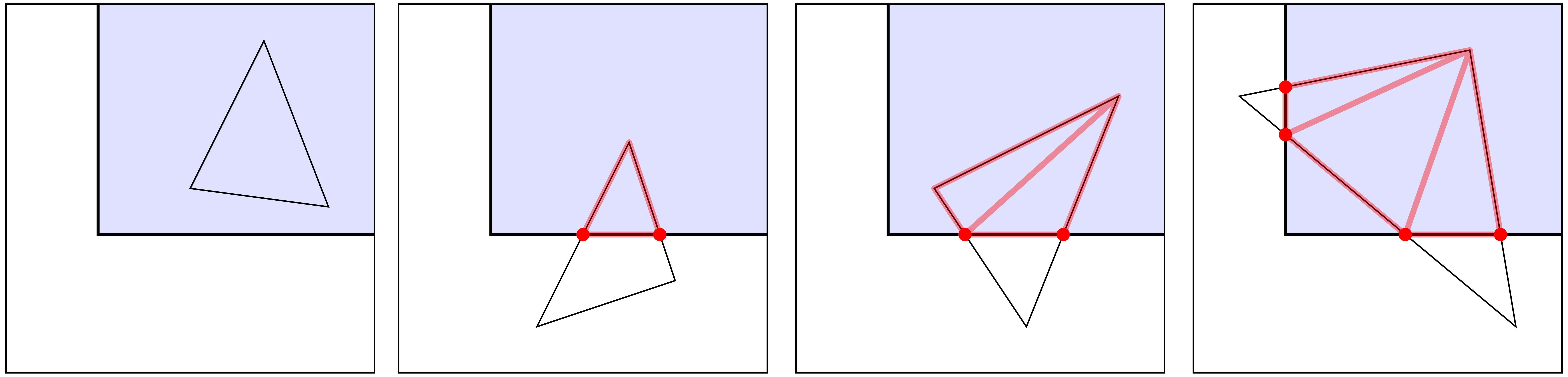
View frustum corresponding to pinhole camera
(perspective projection transform transforms this volume to normalized cube)

Why do we map frustum to unit cube?

1. **Makes *clipping* much easier! (see next slide)**
   - **Can quickly discard geometry outside range [-1,1]**
2. **Represent all vertices in normalized cube in fixed point math**

Clipping

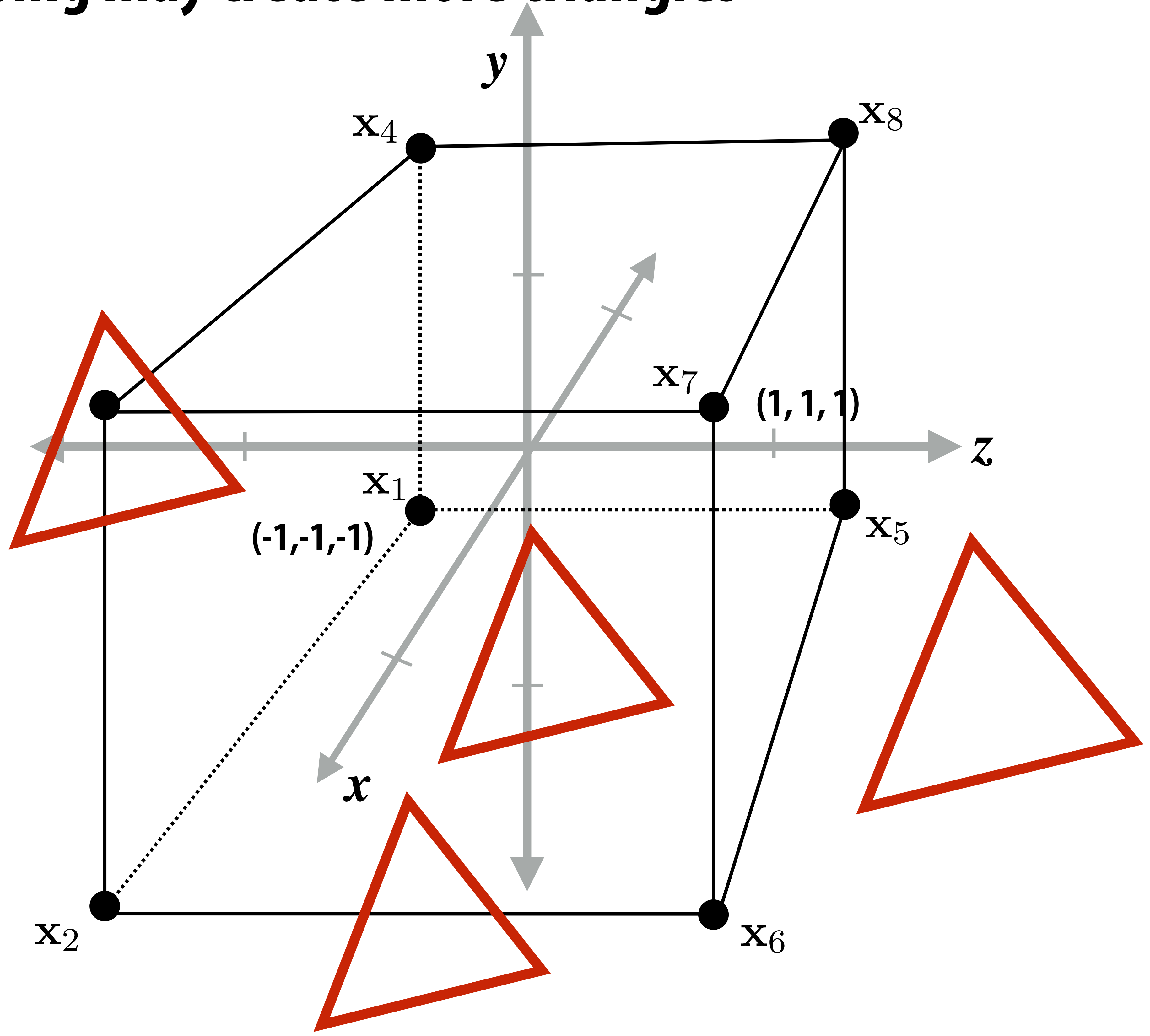
- "Clipping" is the process of eliminating triangles that aren't visible from the camera (they outside the view frustum)
  - Don't waste time computing the appearance of primitives the camera can't see!
  - Sample-in-triangle tests are expensive ("fine granularity" visibility)
  - Makes more sense to toss out entire primitives ("coarse granularity")
  - Must deal with primitives that are partially clipped…



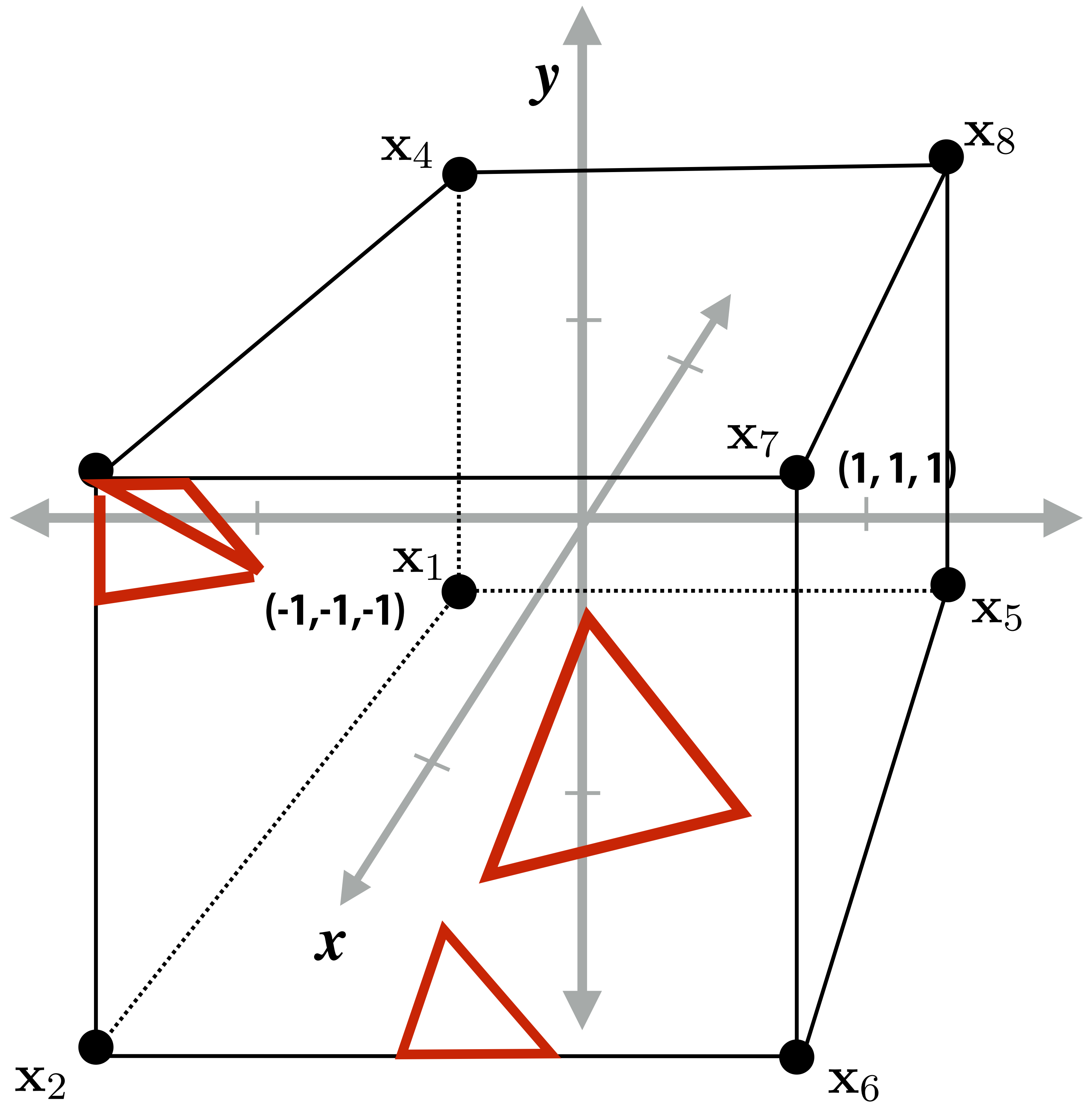
from: https://paroj.github.io/gltut/

Clipping in normalized device coordinates (NDC)

- Discard triangles that lie complete outside the normalized cube (culling)
  - They are off screen, don't bother processing them further
- Clip triangles that extend beyond the cube… to the sides of the cube
  - Note: clipping may create more triangles



Triangles before clipping

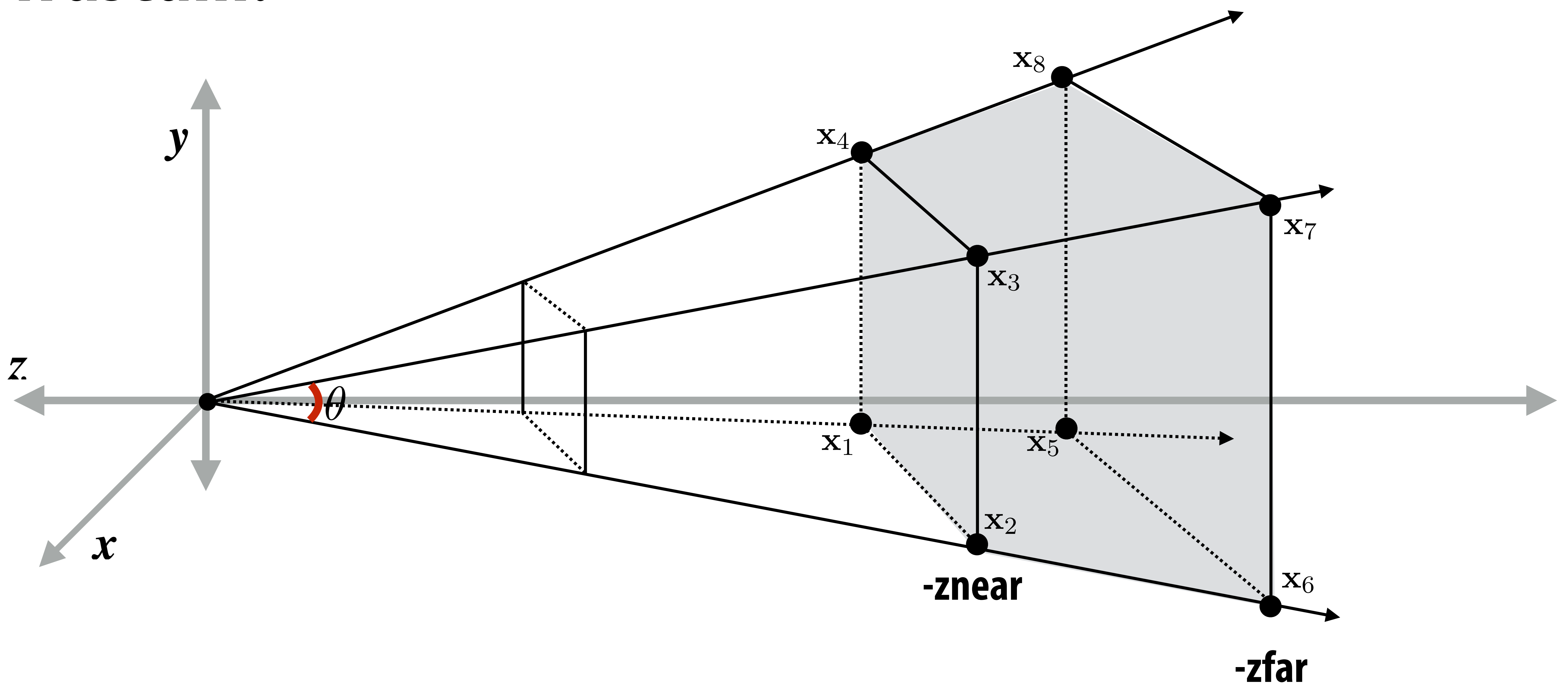


Triangles after clipping

* These figures are correct: normalized device coordinates for OpenGL (a popular graphics API) is left-handed coordinate space

Matrix for perspective transform

Takes into account geometry of view frustum:



$$\begin{pmatrix} \frac{n}{r} & 0 & 0 & 0 \\ 0 & \frac{n}{t} & 0 & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{pmatrix}$$

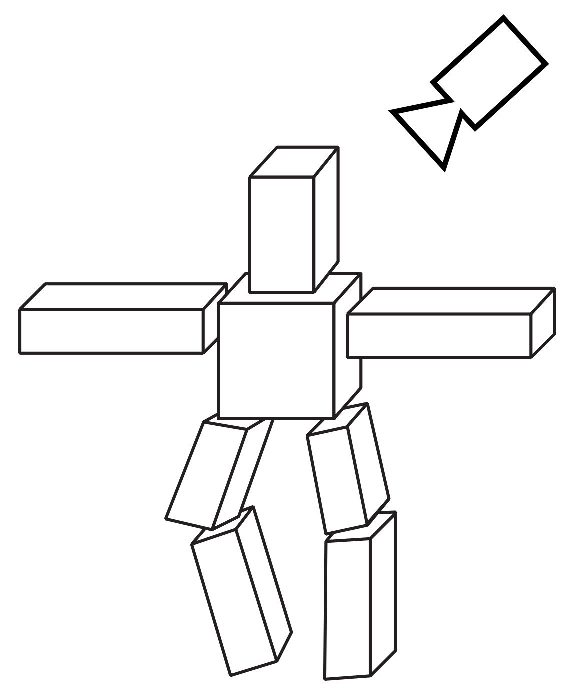
left (l), right (r), top (t), bottom (b), near (n), far (f)

(matrix at left is perspective projection for frustum that is symmetric about x,y axes: l=-r, t=-b)

For a derivation: http://www.songho.ca/opengl/gl_projectionmatrix.html

Transformations: from objects to the screen

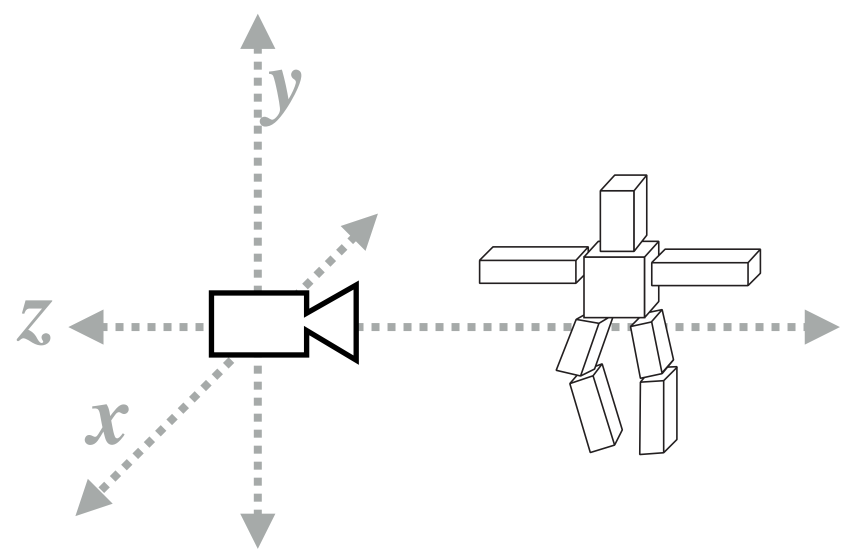
[WORLD COORDINATES]



original description of objects

view transform

[VIEW COORDINATES]

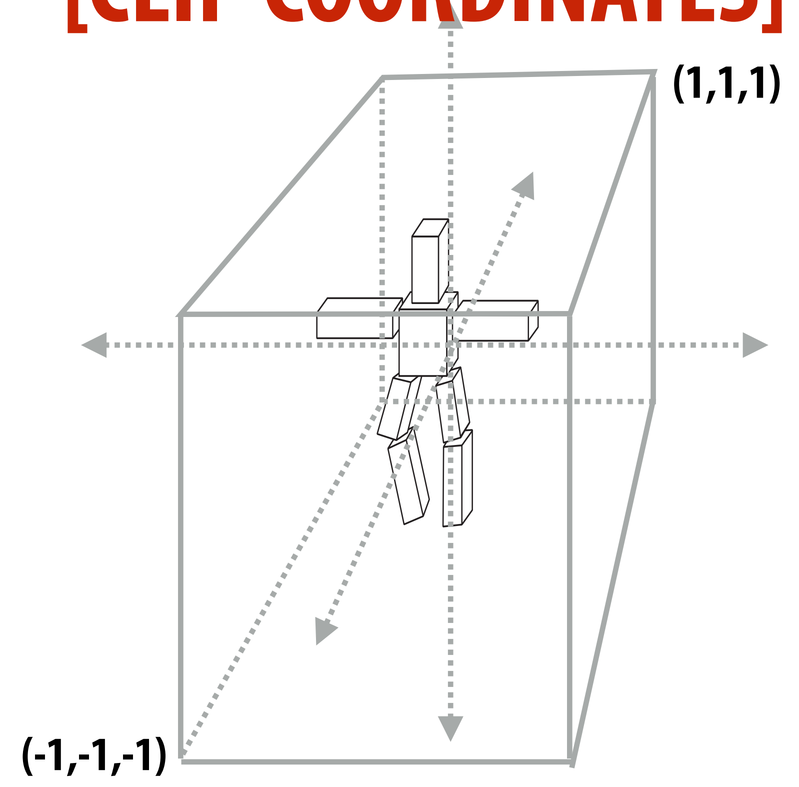


vertex positions now expressed relative to camera; camera is sitting at origin looking down -z direction
(Canonical frame of reference allows for use of canonical projection matrix)

projection transform

[CLIP COORDINATES]

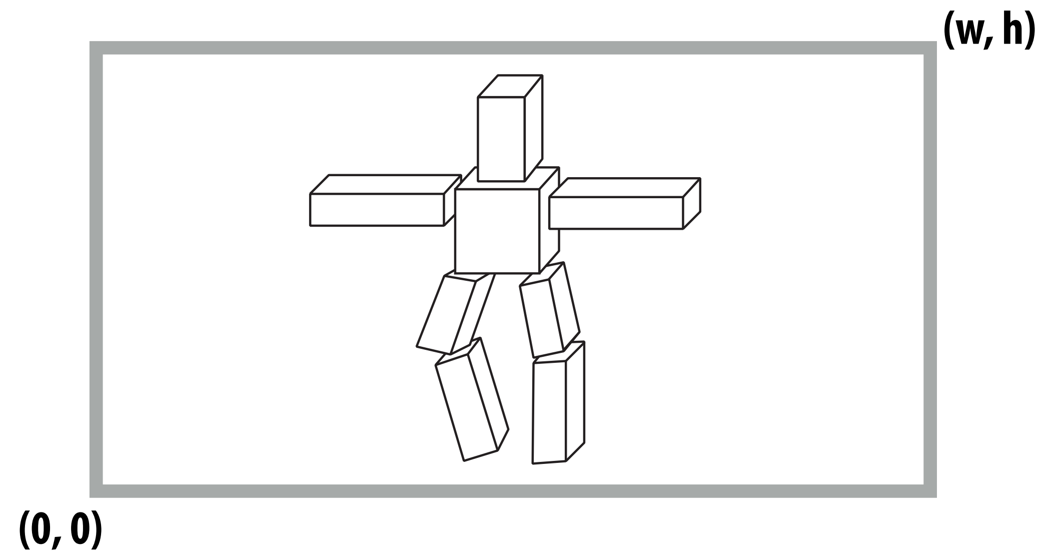
(Also called "normalized device coordinates")



everything visible to the camera is mapped to unit cube for easy "clipping"

screen transform

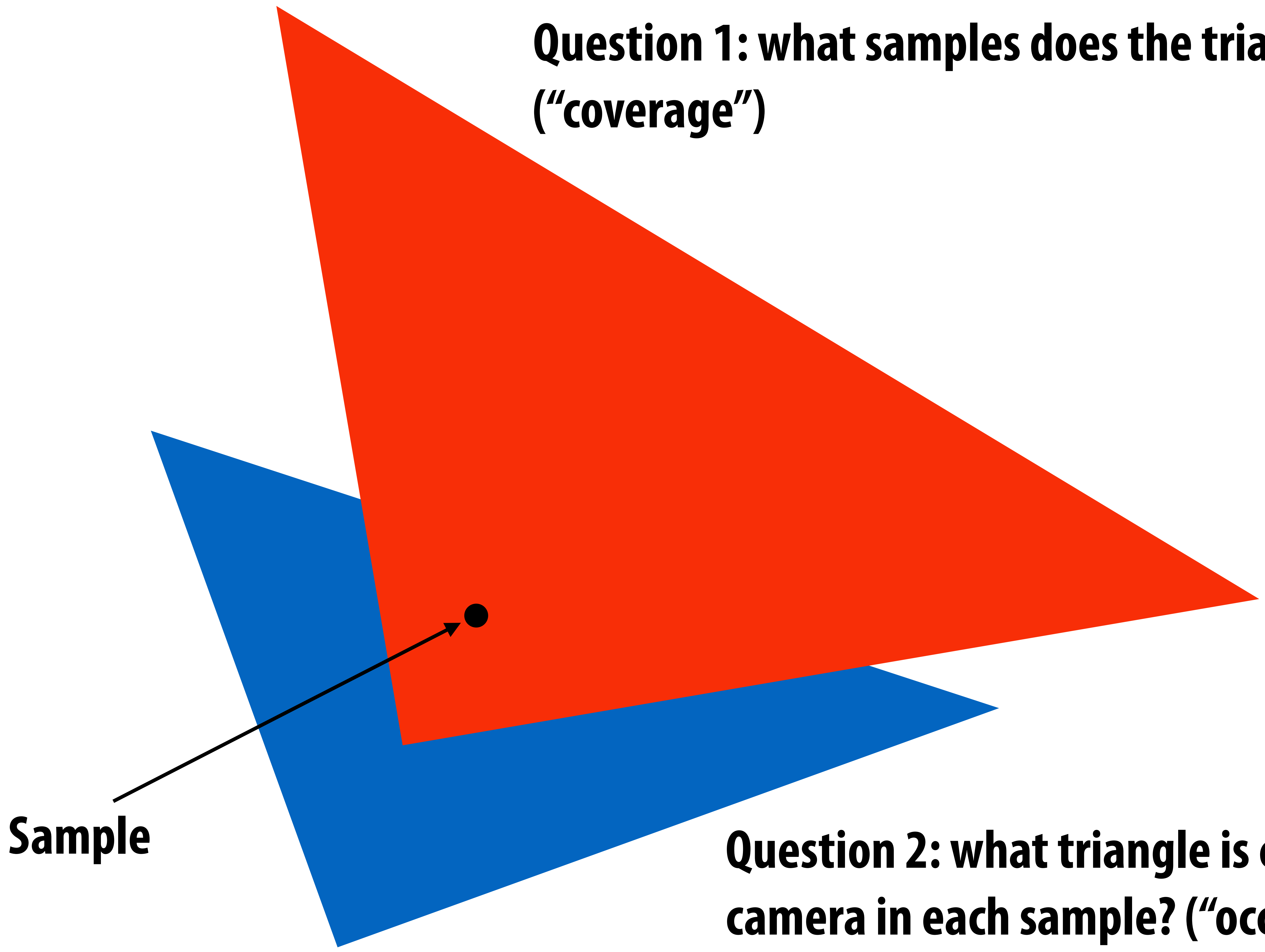
[WINDOW COORDINATES]



objects now in 2D screen coordinates

primitives are now 2D and can be drawn via rasterization

Triangle visibility problem… in a rasterizer

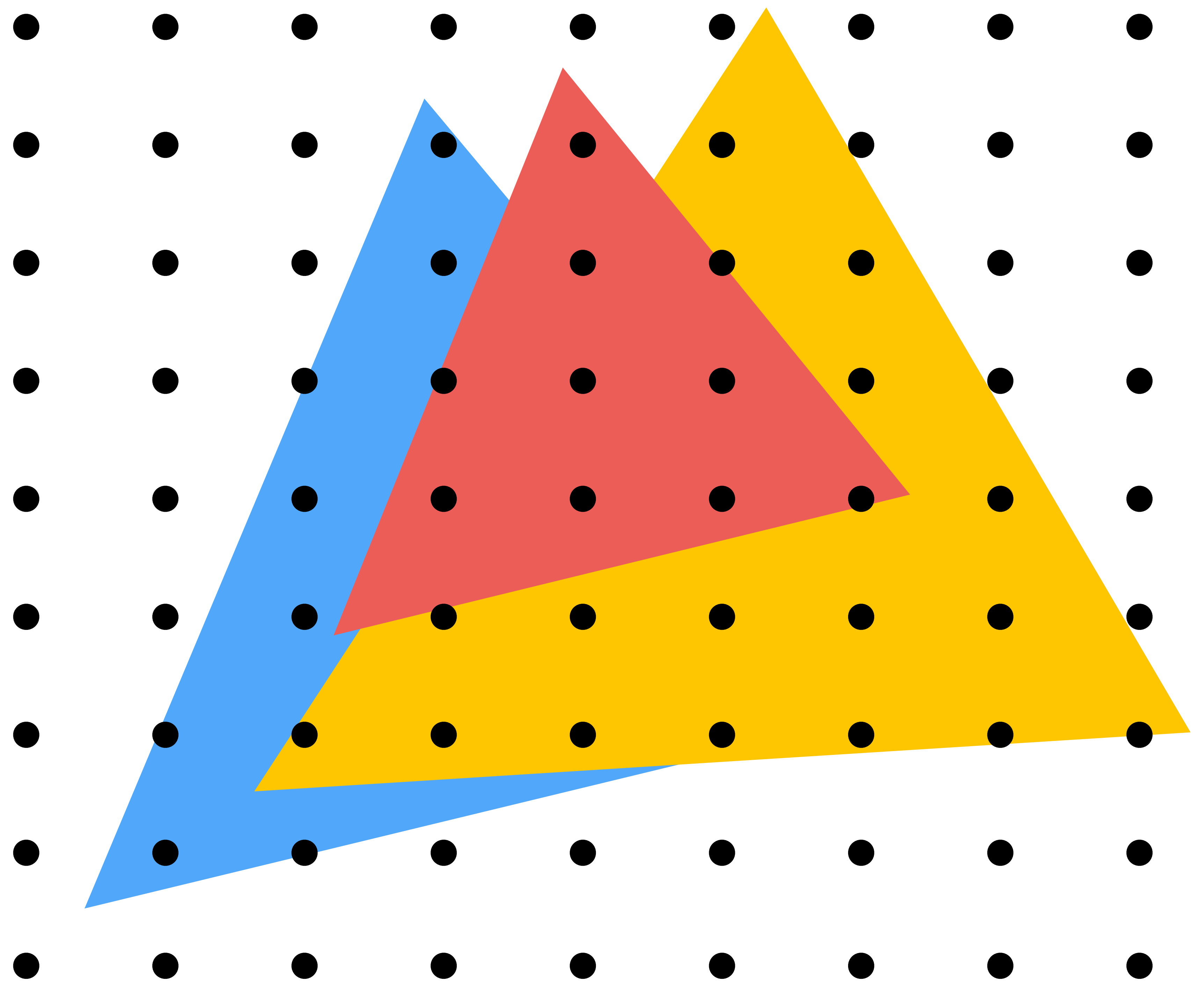


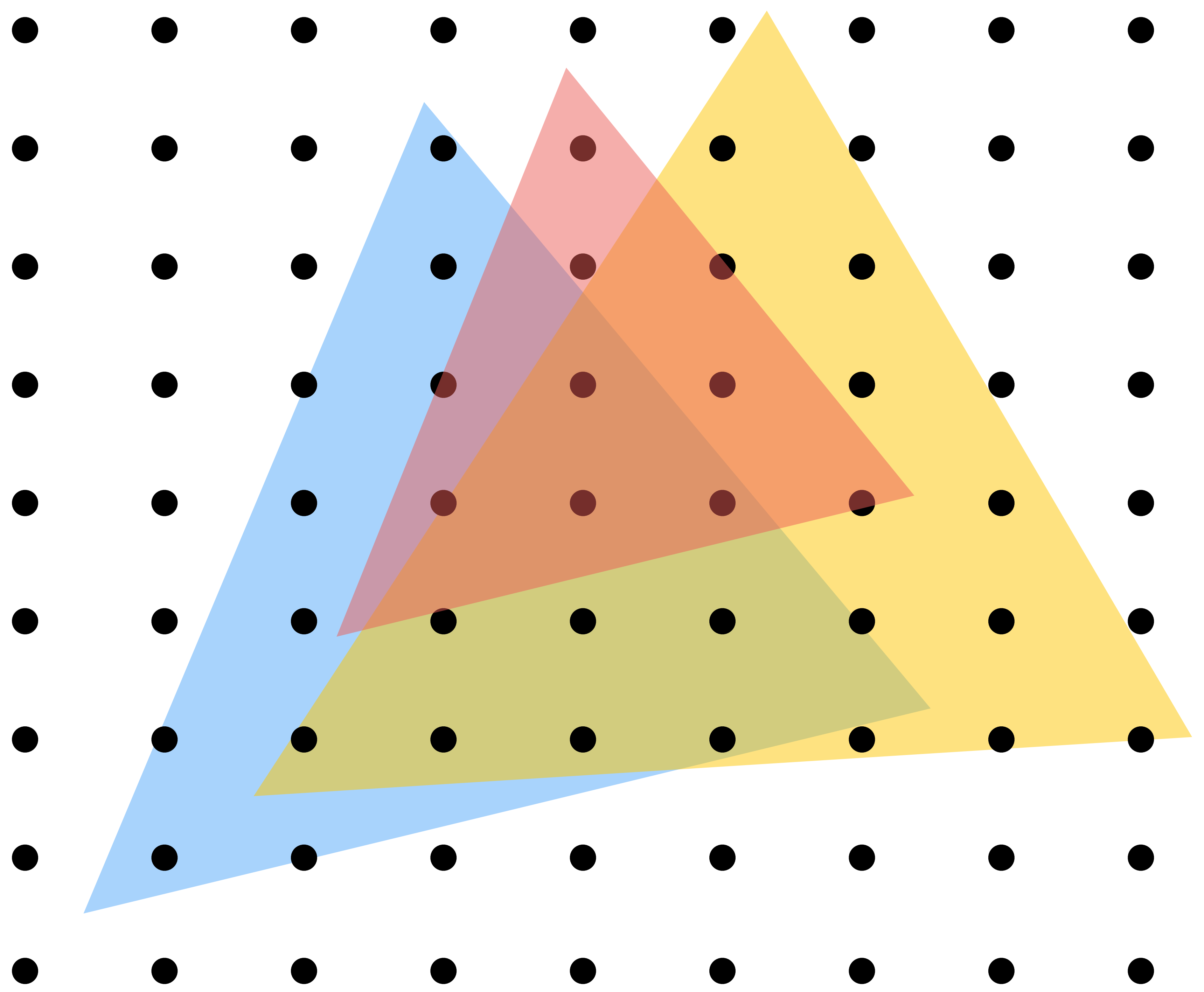
Question 1: what samples does the triangle overlap? ("coverage")

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

Occlusion using the Depth Buffer

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

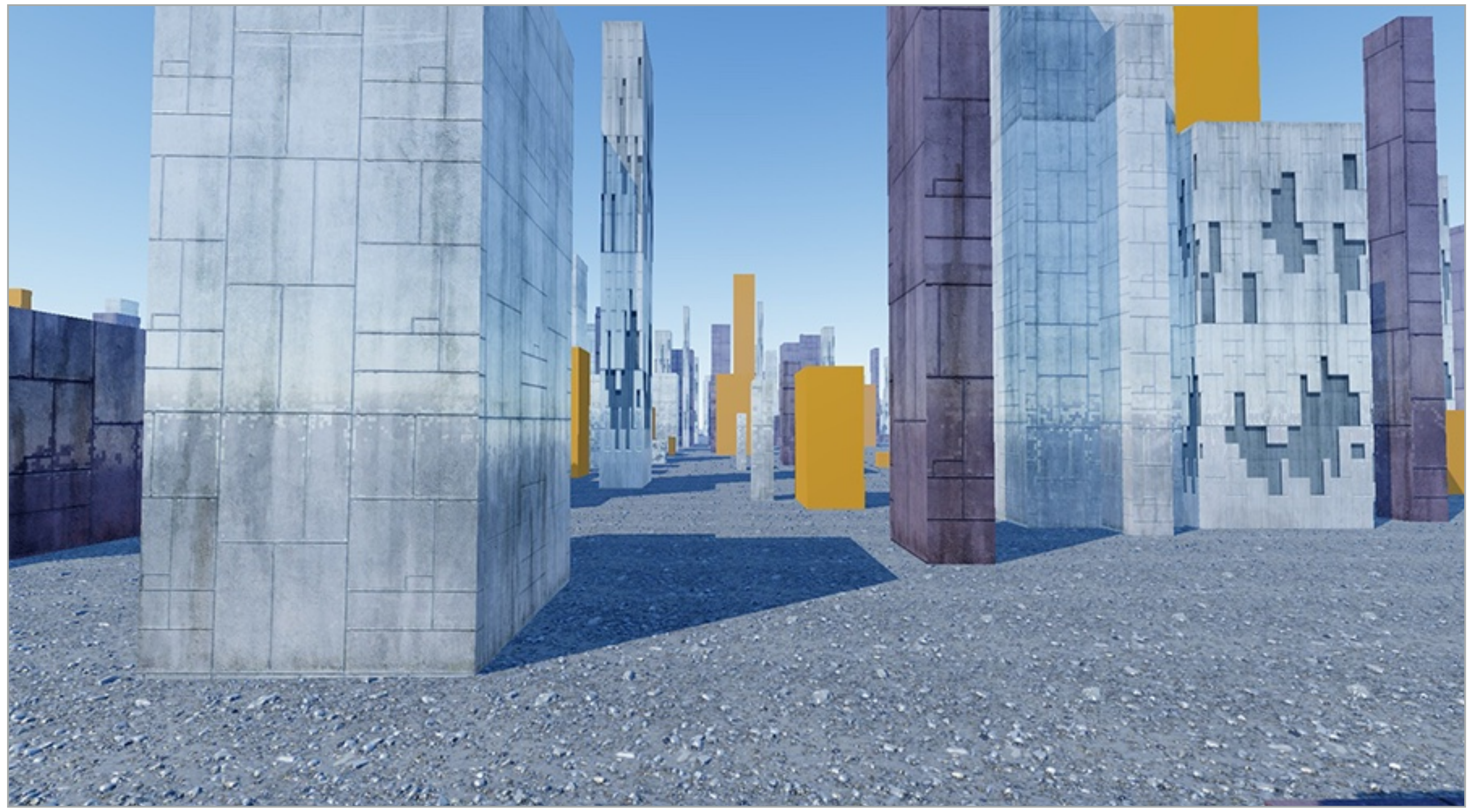

Opaque Triangles


50% transparent triangles

Depth buffer (aka "Z buffer")

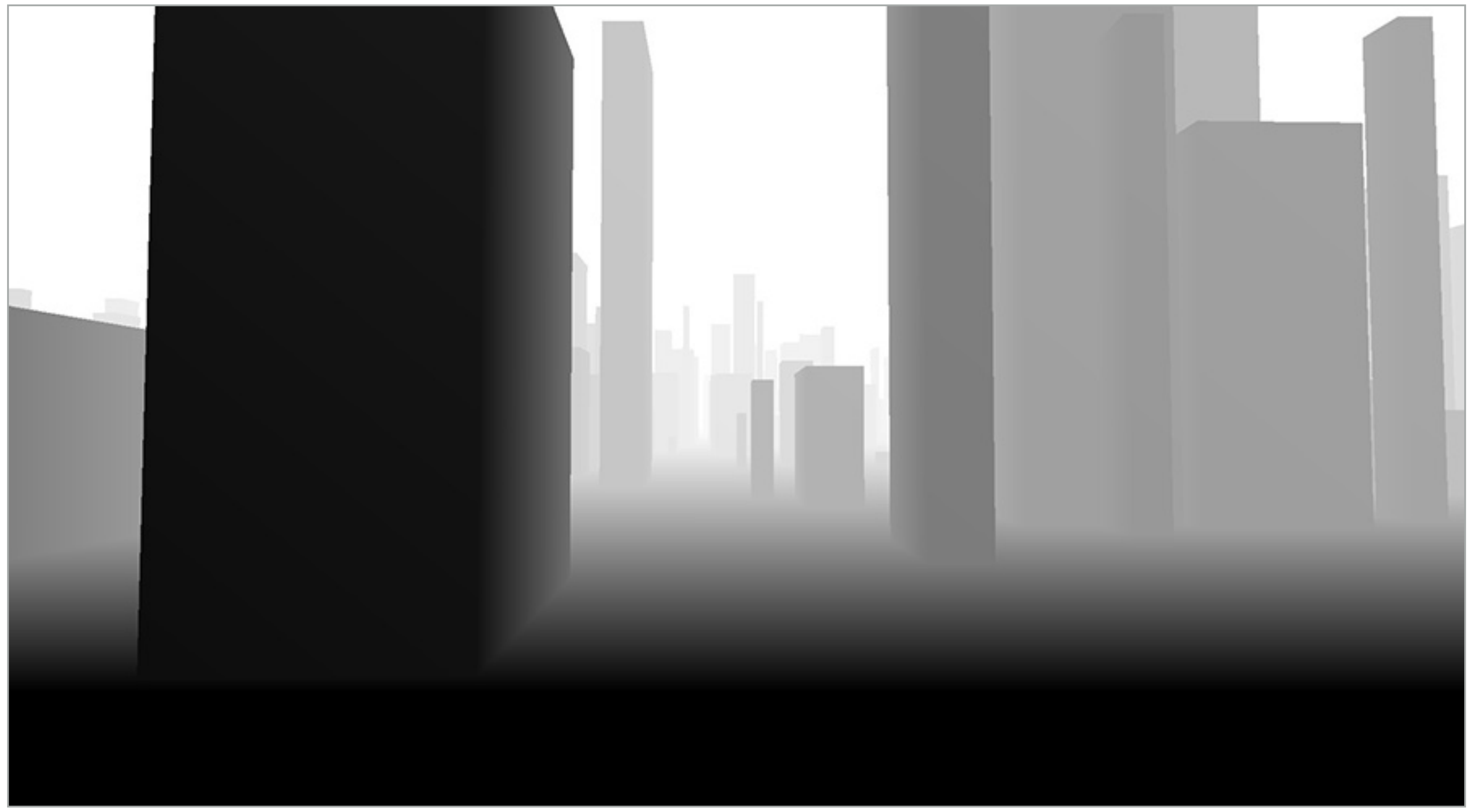
Color buffer:

(stores color per sample… e.g., RGB)



Depth buffer:

(stores depth per sample)

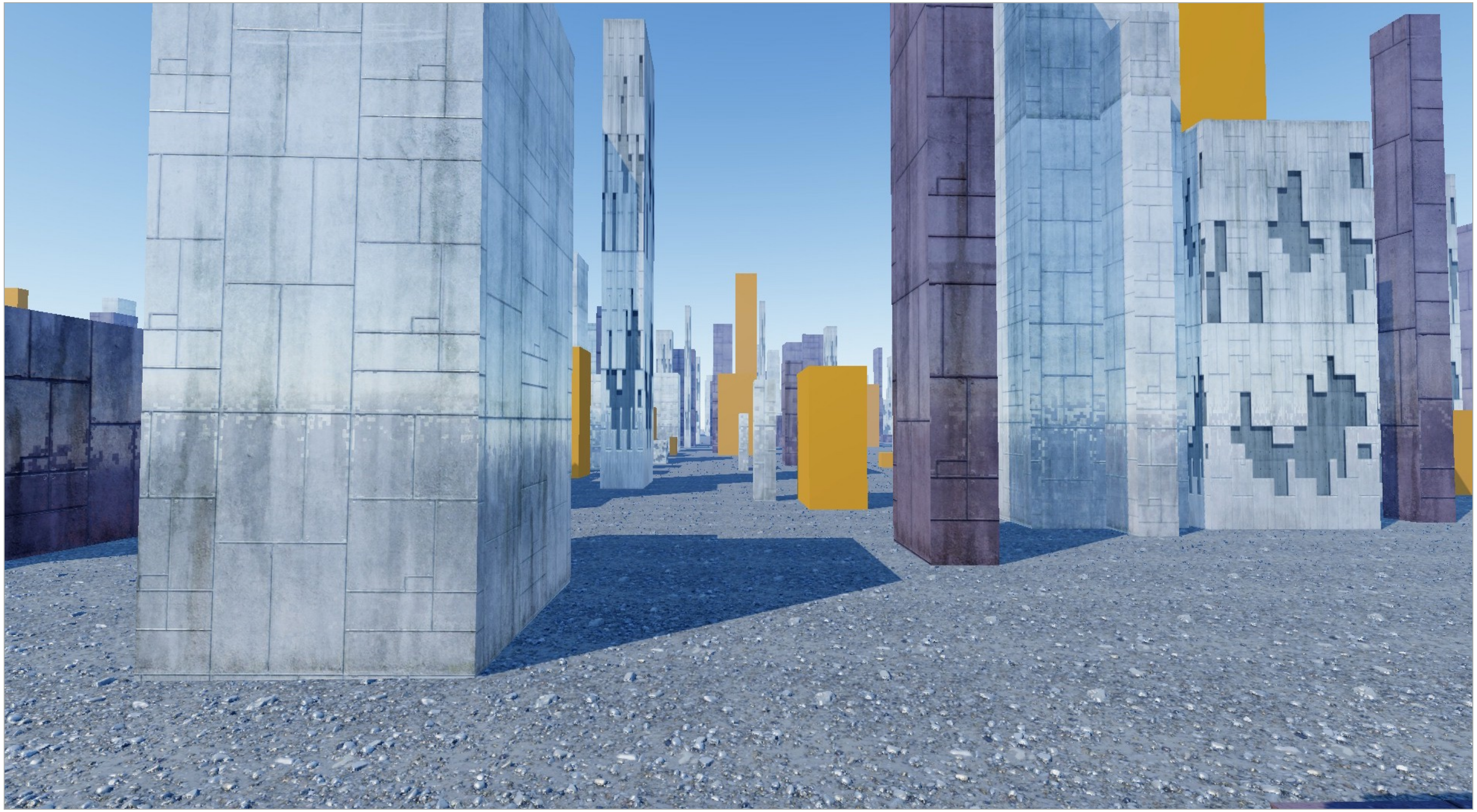


Stores depth of closest surface drawn so far

black = close depth

white = far depth

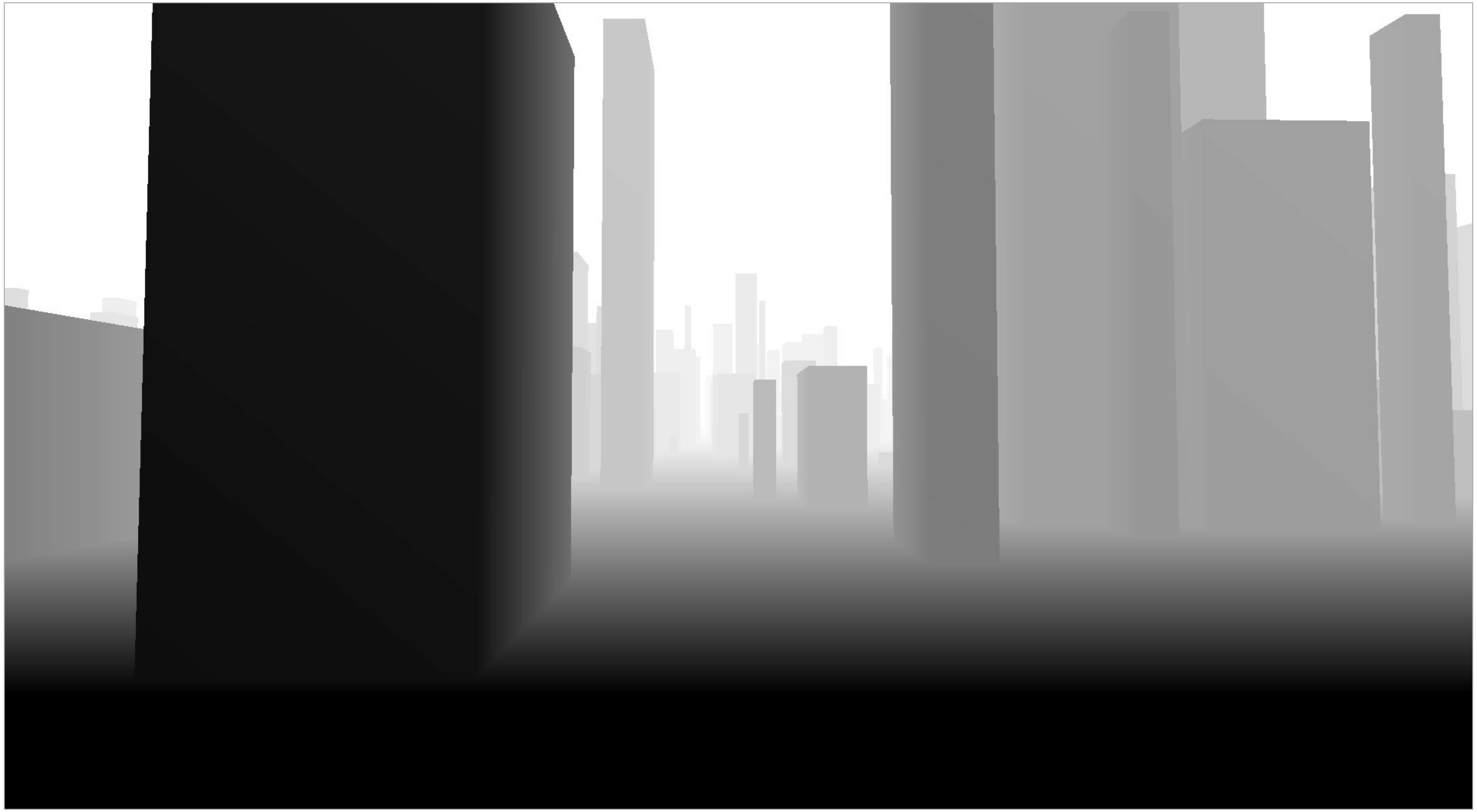
Depth buffer (a better look)



Color buffer (stores color measurement per sample, eg., RGB value per sample)

Depth buffer (a better look)

Visualization: the darker the pixel, the shorter the distance to the closest object



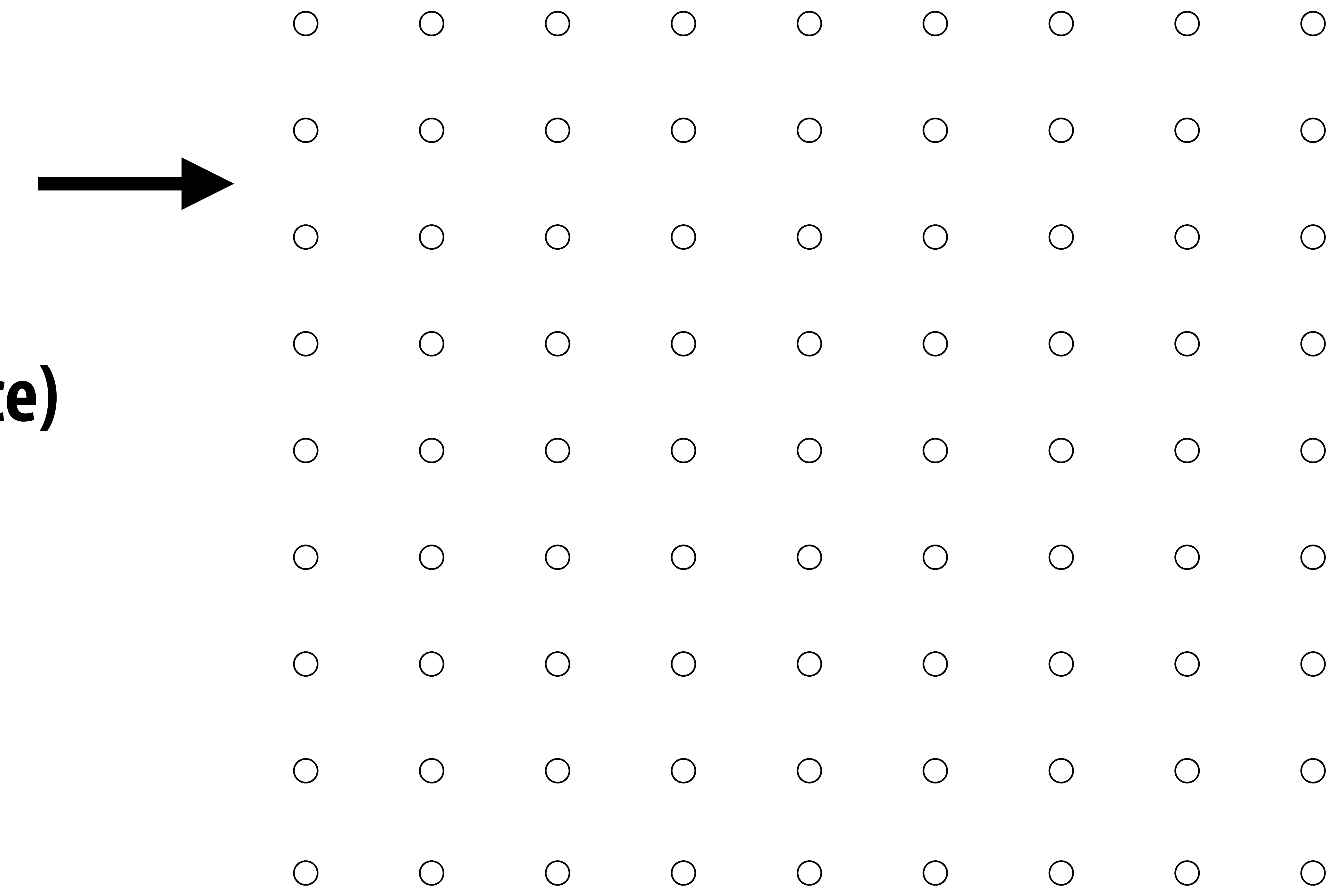
Corresponding depth buffer after rendering all triangles (stores closest scene depth per sample)

Occlusion using the depth-buffer ("Z-buffer")

For each coverage sample point, the depth-buffer stores depth of closest triangle at this sample point that has been processed by the renderer so far.

Closest triangle at sample point (x,y) is triangle with minimum depth at (x,y)

Initial state of depth buffer before rendering any triangles (all samples store "farthest" distance)



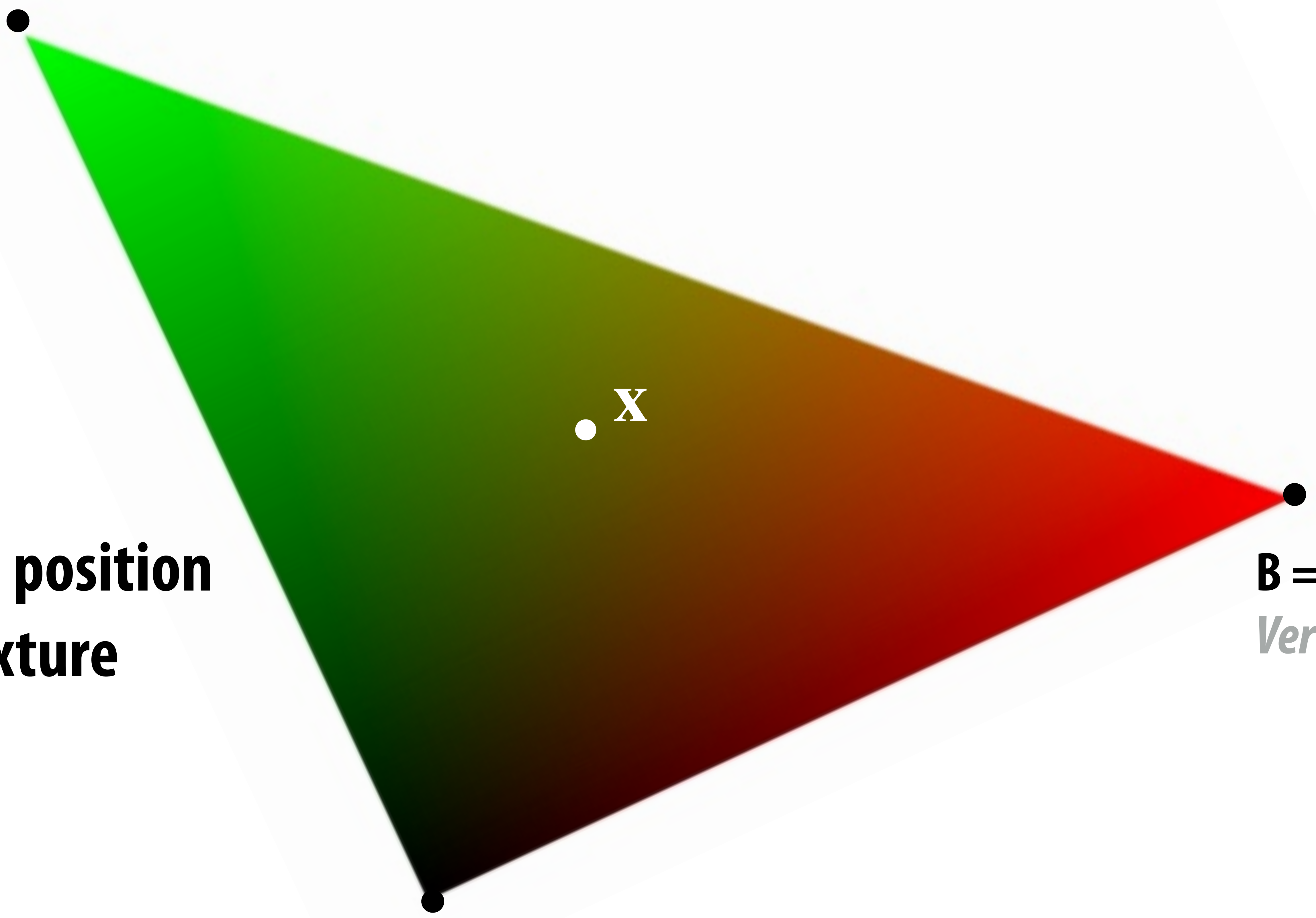
Grayscale value of sample point used to indicate distance

Black = small distance

White = large distance

How do we compute the triangle's depth at a screen sample point (x,y)?

Recall linear interpolation of values (defined at vertices)



C = (x2, y2, z2, r2, g2, b2, u2, v2)

Vertex is green, so (r2,g2,b2) = (0,1,0)

B = (x1, y1, z1, r1, g1, b1, u1, v1)

Vertex is red, so (r1,g1,b1) = (1,0,0)

A = (x0, y0, z0, r0, g0, b0, u0, v0)

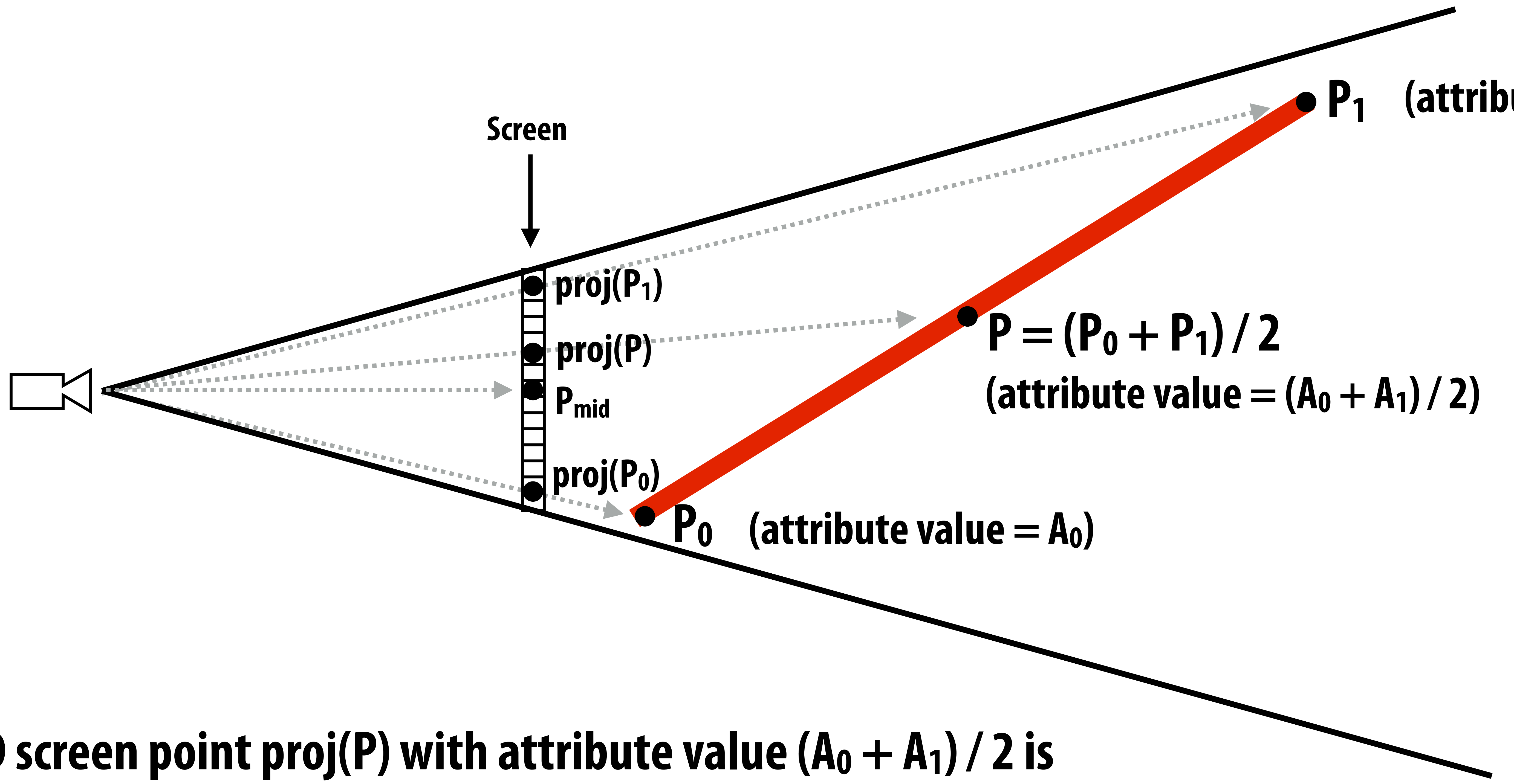
Vertex is black, so (r0,g0,b0) = (0,0,0)

Here, I'm interpolating the position (x,y,z), color (r,g,b), and texture coordinate values (u,v)

Not so fast… perspective incorrect interpolation

The value of an attribute at the 3D point P on a triangle is a linear combination of attribute values at vertices.

But due to perspective projection, barycentric interpolation of values on a triangle with vertices of different depths in 3D <u>is not</u> linear in 2D screen XY coordinates (vertex coordinates *after* projection)


Screen
proj(P_1)
proj(P)
P_{mid}
proj(P_0)
P_1 (attribute value = A_1)
$P = (P_0 + P_1) / 2$
(attribute value = $(A_0 + A_1) / 2$)
P_0 (attribute value = A_0)

In this example, the 2D screen point proj(P) with attribute value $(A_0 + A_1) / 2$ is not halfway between the 2D screen points proj(P_0) and proj(P_1).

Similarly, the attribute's value at $P_{mid} = (\text{proj}(P_0) + \text{proj}(P_1)) / 2$ is not $(A_0 + A_1) / 2$.

Perspective correct interpolation on a projected triangle (in 2D)

- **Given:**
  - Some value f_i at each of a 3D triangle's vertices, that is linearly interpolated across the triangle in 3D
  - The 2D screen coordinates $P_i=(x_i,y_i)$ of each of a triangle's vertices after projection
  - The homogenous coordinate w_i for each vertex

- **Compute:**
  - The value of $f(x,y)$ for the projected triangle at a given 2D screen space location (x,y)

Perspective-correct interpolation

Assume a triangle attribute varies linearly across the triangle (in 3D)

Attribute's value at 3D point on triangle $P = \begin{bmatrix} x & y & z \end{bmatrix}^T$ is given by:

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

Perspective project P, get 2D homogeneous representation:

$$\begin{bmatrix} x_{\mathrm{2D-H}} \\ y_{\mathrm{2D-H}} \\ w \end{bmatrix} \leftarrow \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

projection of *P* in 2D-H

Drop z to move to 2D-H

perspective projection of *P* in 3D-H

Simple perspective projection matrix *

point *P* in 3D-H

*** Note: using a more general perspective projection matrix only changes the coefficient in front of x_{2d} and y_{2d} . (property that f/w is affine still holds)**

Then plug back in to equation for *f* at top of slide…

$$f(x_{\mathrm{2D-H}}, y_{\mathrm{2D-H}}) = a x_{\mathrm{2D-H}} + b y_{\mathrm{2D-H}} + cw$$

$$\frac{f(x_{\mathrm{2D-H}}, y_{\mathrm{2D-H}})}{w} = \frac{a}{w} x_{\mathrm{2D-H}} + \frac{b}{w} y_{\mathrm{2D-H}} + c$$

$$\frac{f(x_{\mathrm{2D}}, y_{\mathrm{2D}})}{w} = \frac{a}{w} x_{\mathrm{2D}} + \frac{b}{w} y_{\mathrm{2D}} + c$$

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

Direct evaluation of surface attributes from 2D-H vertices

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

w coordinate of vertex *a* after perspective projection transform

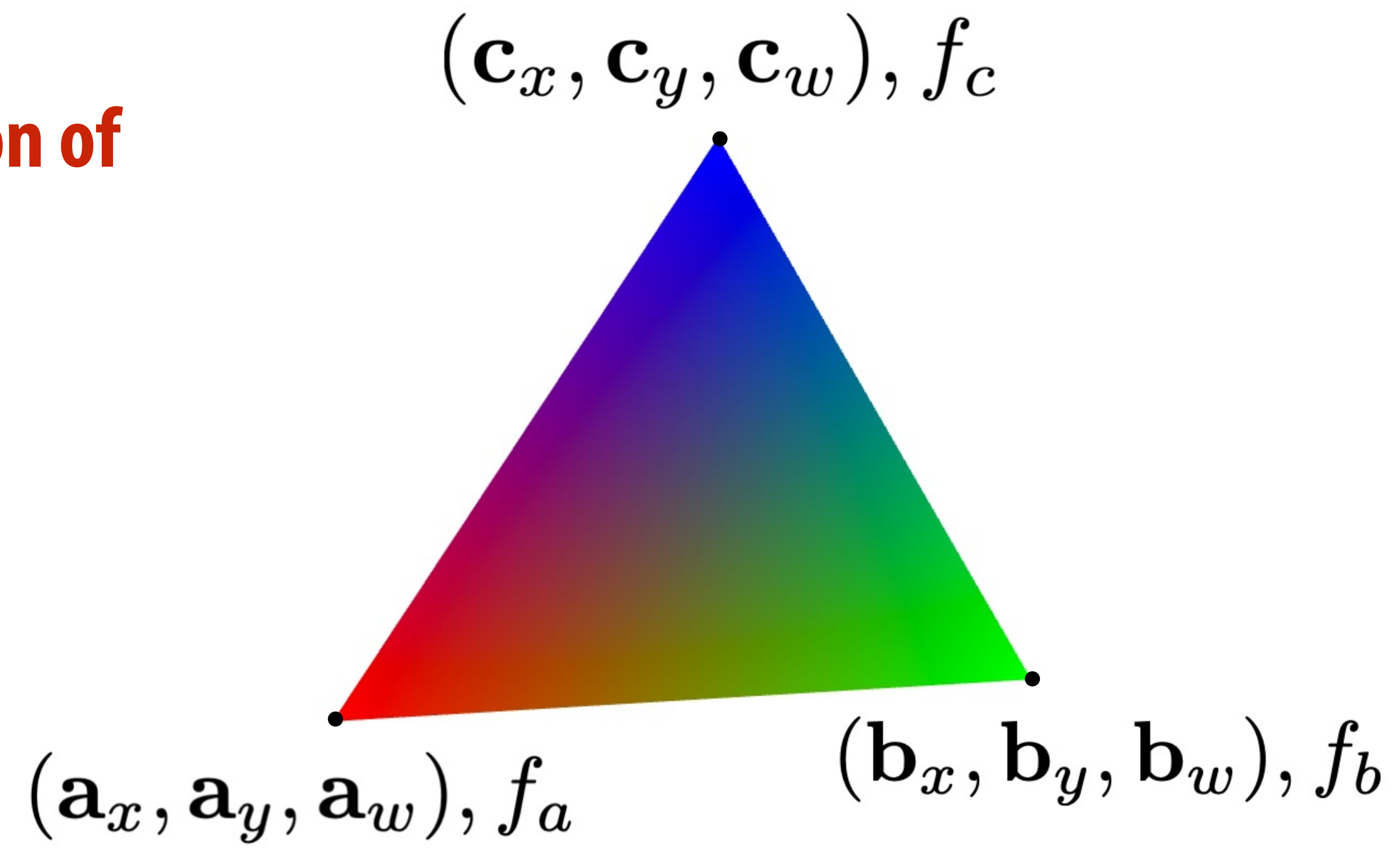
value of attribute at vertex *a*

projected 2D position of vertex *a*

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

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

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


$(\mathbf{c}_x, \mathbf{c}_y, \mathbf{c}_w), f_c$
$(\mathbf{a}_x, \mathbf{a}_y, \mathbf{a}_w), f_a$
$(\mathbf{b}_x, \mathbf{b}_y, \mathbf{b}_w), f_b$

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

This is done as a per-triangle "setup" computation prior to sampling.

Efficient perspective-correct interpolation

Setup:

Given f_a, f_b, f_c and w_a, w_b, w_c . . . compute A, B, C for *f/w(x,y)* = Ax + By + C

Also compute equation for *1/w(x,y)*

To evaluate surface attribute f(x,y) at every covered sample (x,y):

Evaluate $^1/_w$*(x,y)* **(from precomputed equation for value $^1/_w$)**

Reciprocate $^1/_w$*(x,y)* to get *w(x,y)*

For each triangle attribute:

- **Evaluate $^f/_w$*(x,y)*** **(from precomputed equation for value $^f/_w$)**
- **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

See Low: "Perspective-Correct Interpolation"

How do we compute a triangle's depth at a screen (x,y)?

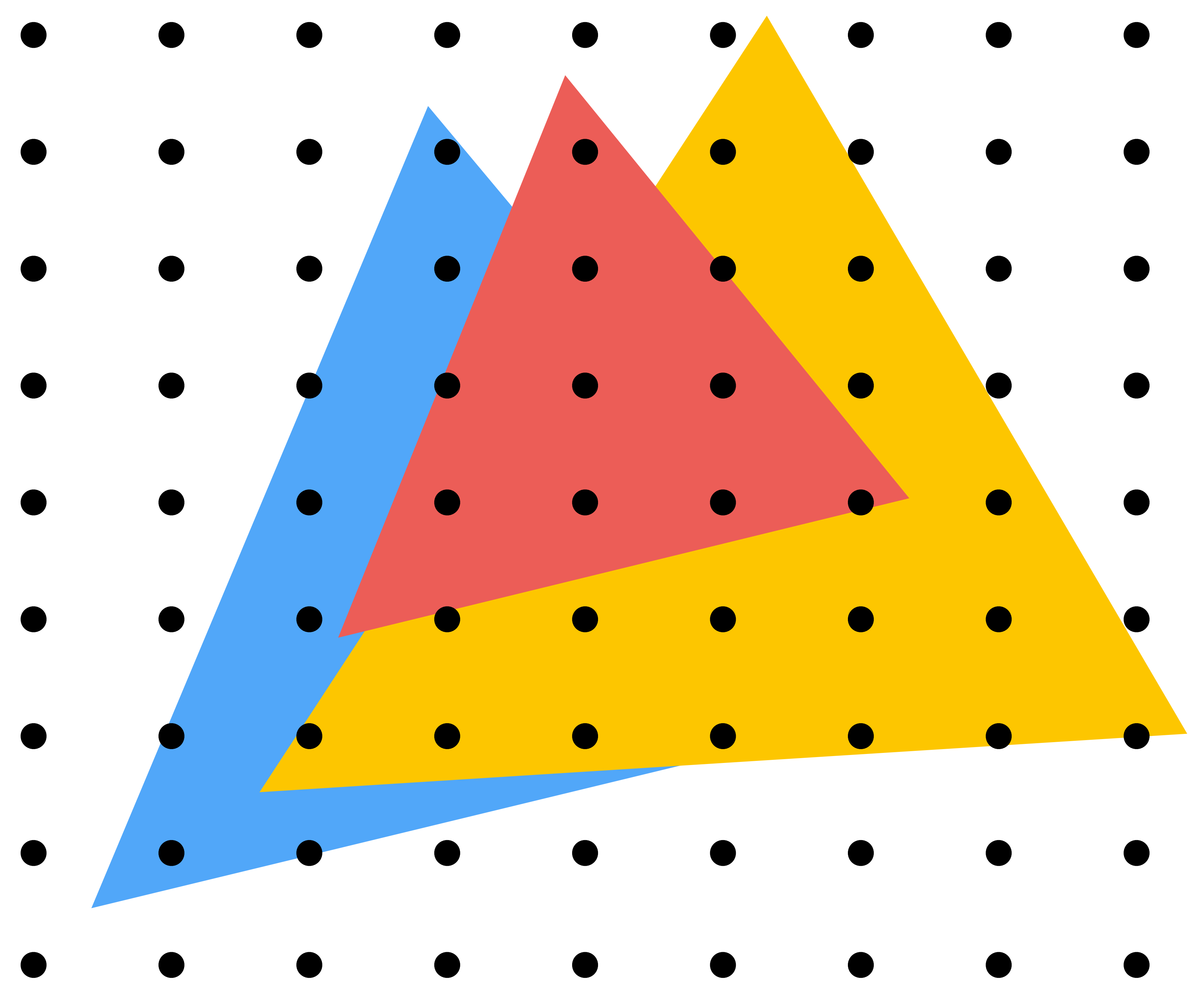
Assume we have a triangle defined by the screen-space 2D position and distance ("depth") from the camera of each vertex.

$$\begin{aligned}
&\begin{bmatrix}\mathbf{p}_{0x} & \mathbf{p}_{0y}\end{bmatrix}^T, \quad w_0, \quad d_0 \\
&\begin{bmatrix}\mathbf{p}_{1x} & \mathbf{p}_{1y}\end{bmatrix}^T, \quad w_1, \quad d_1 \\
&\begin{bmatrix}\mathbf{p}_{2x} & \mathbf{p}_{2y}\end{bmatrix}^T, \quad w_2, \quad d_2
\end{aligned}$$

How do we compute the depth of the triangle at covered sample point (x, y)?

Interpolate it just like any other attribute that varies linearly over the surface of the triangle.

Example: rendering three opaque triangles



Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:

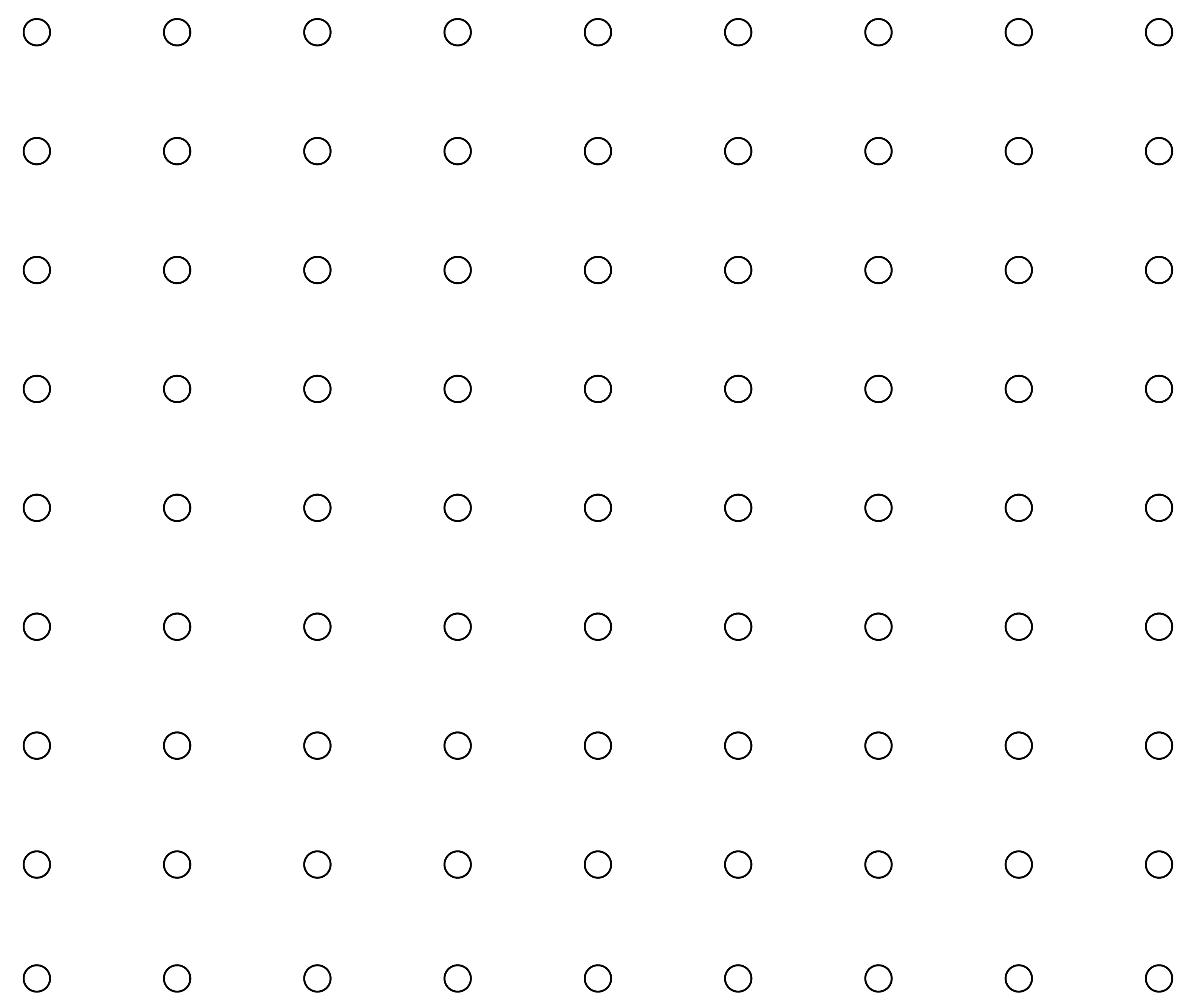
depth = 0.5

Grayscale value of sample point used to indicate distance

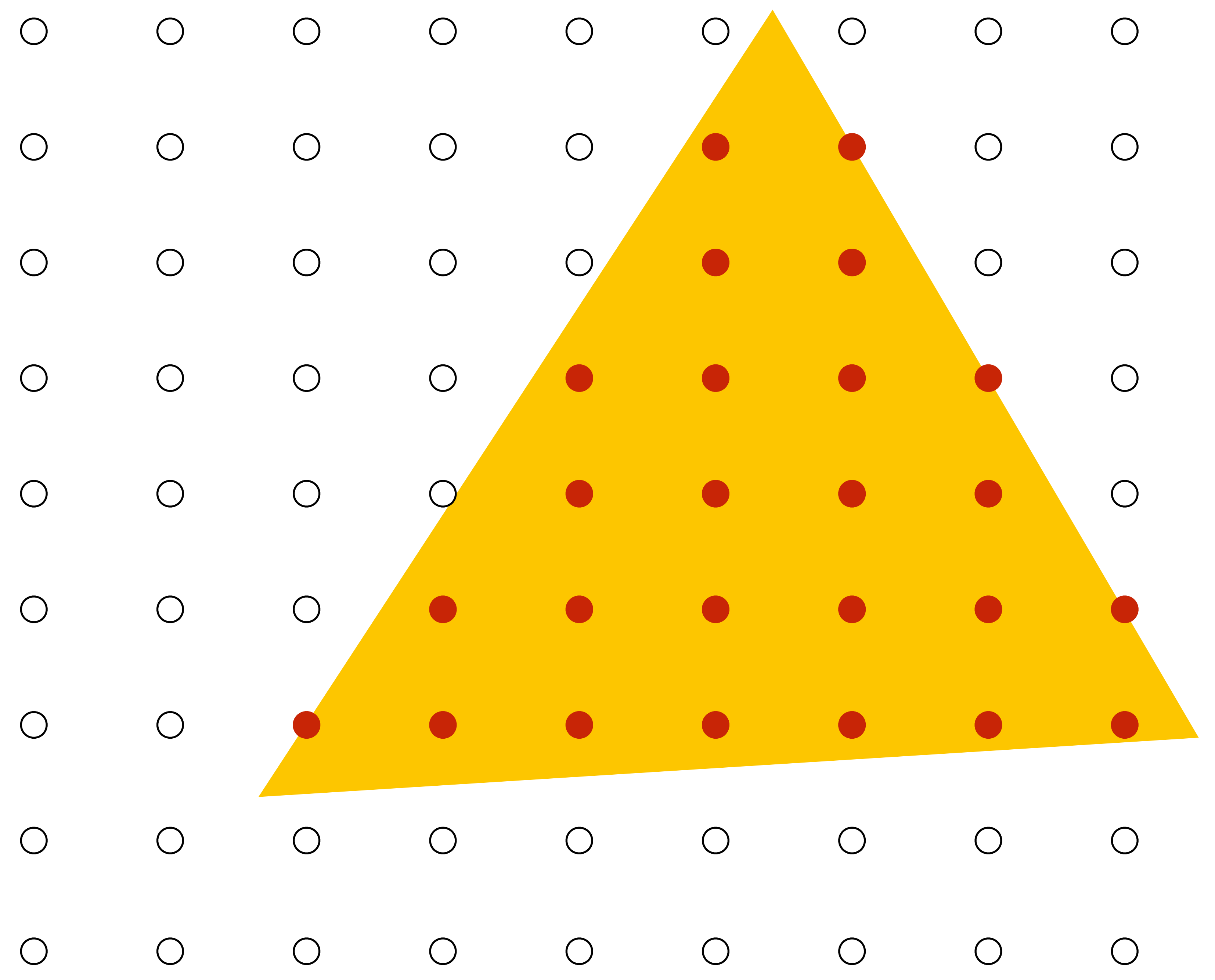
White = large distance

Black = small distance

Red = samples that pass depth test



Color buffer contents



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

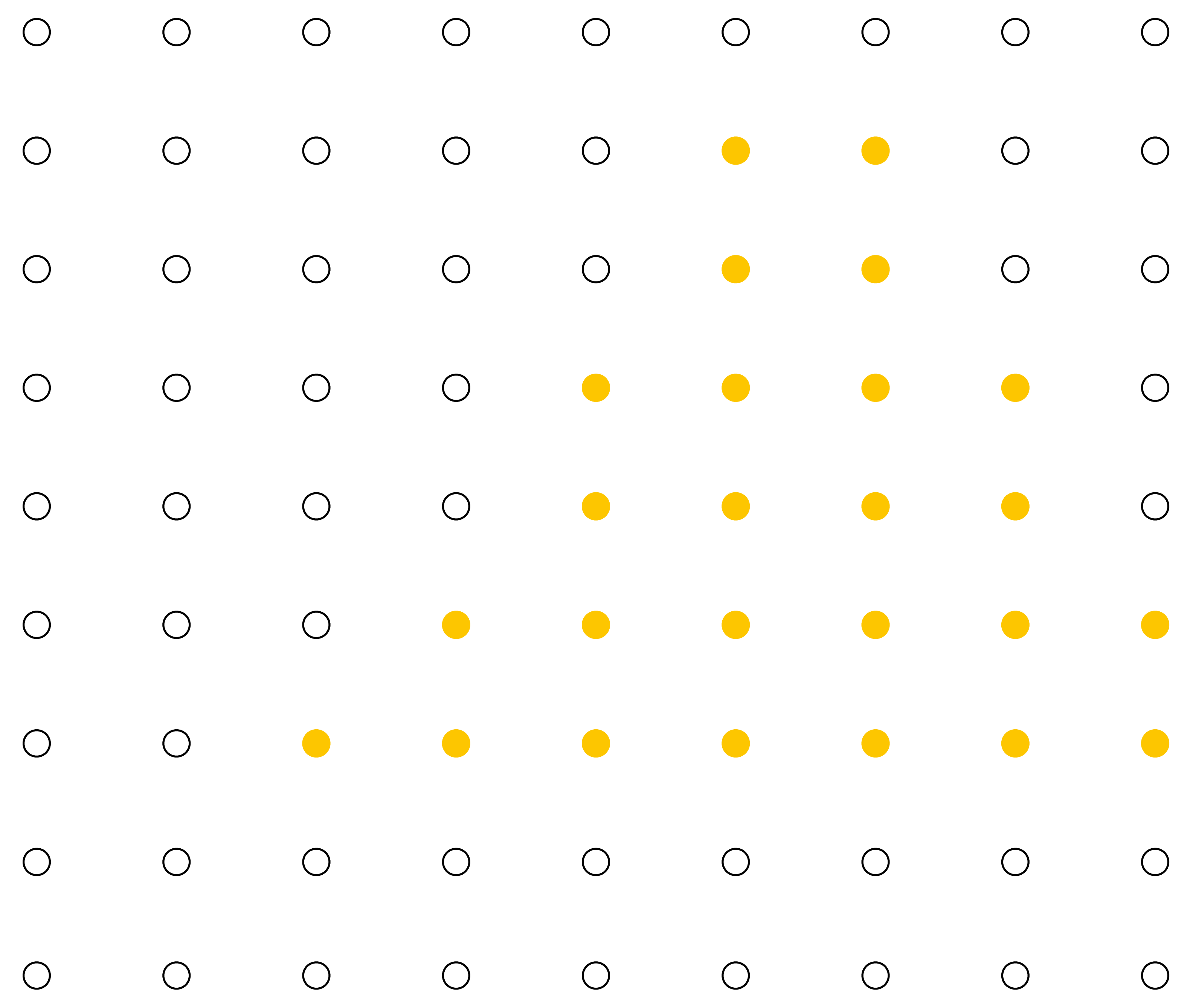
After processing yellow triangle:

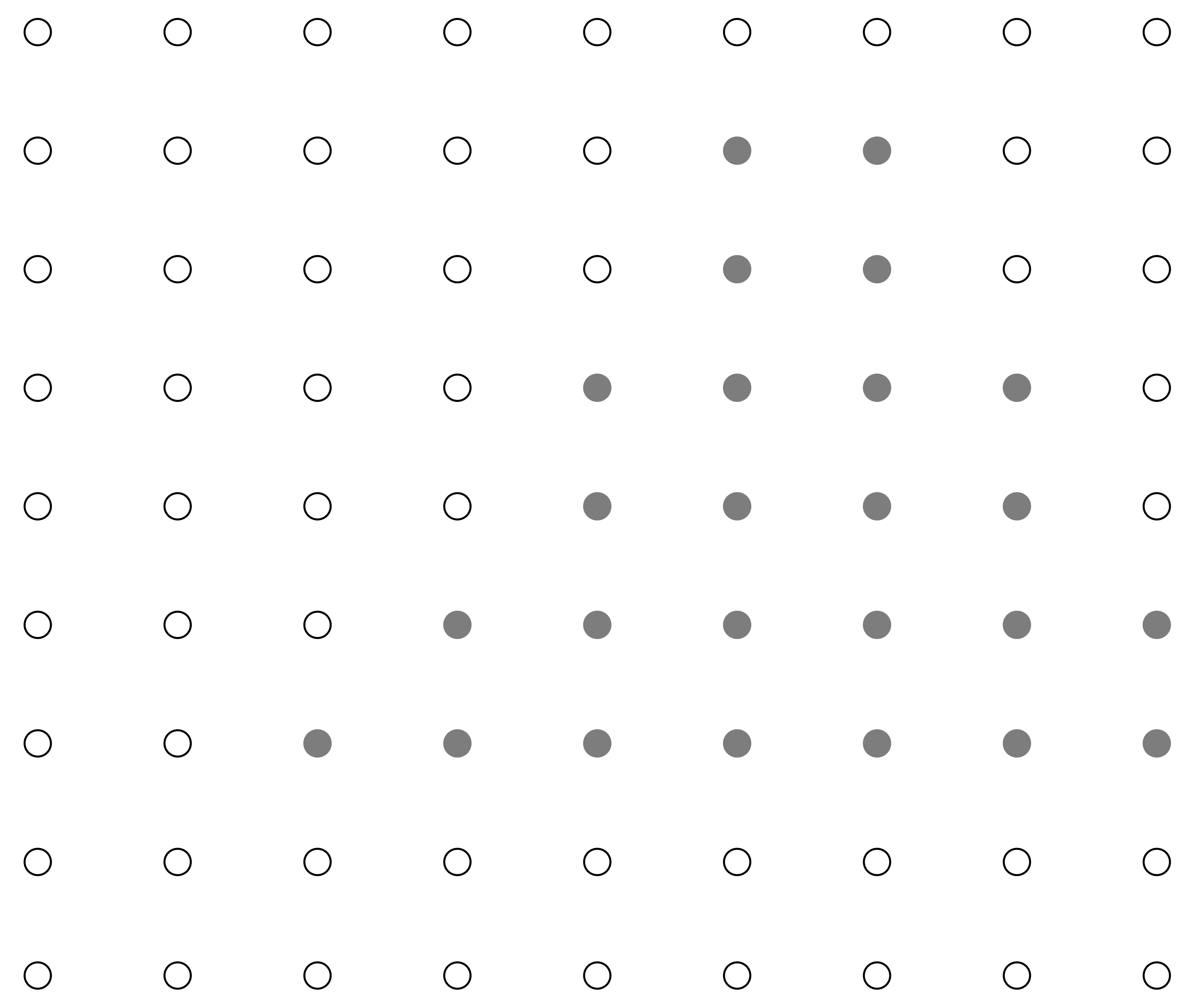
Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = samples that pass depth test


Color buffer contents


Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:

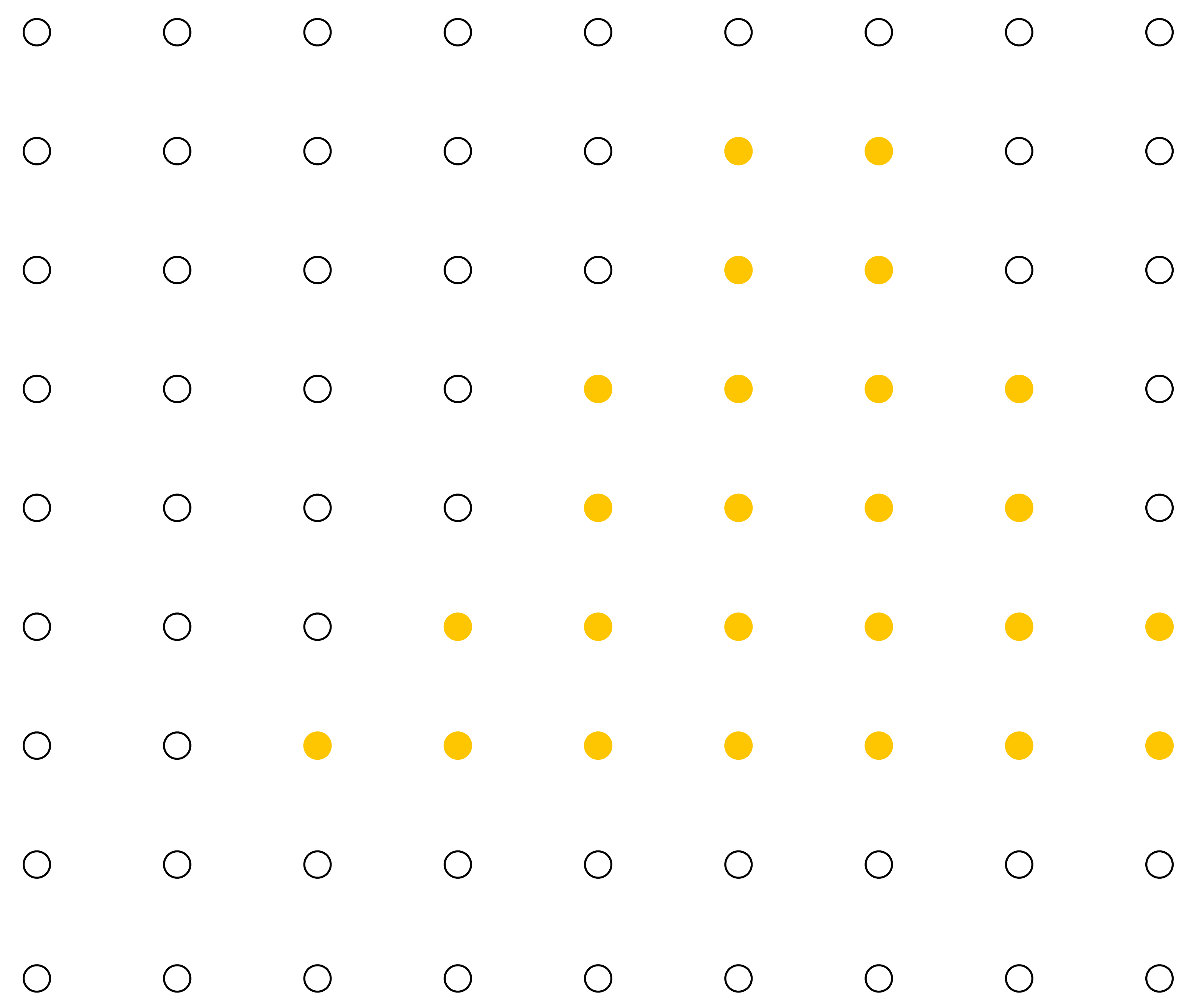
depth = 0.75

Grayscale value of sample point used to indicate distance

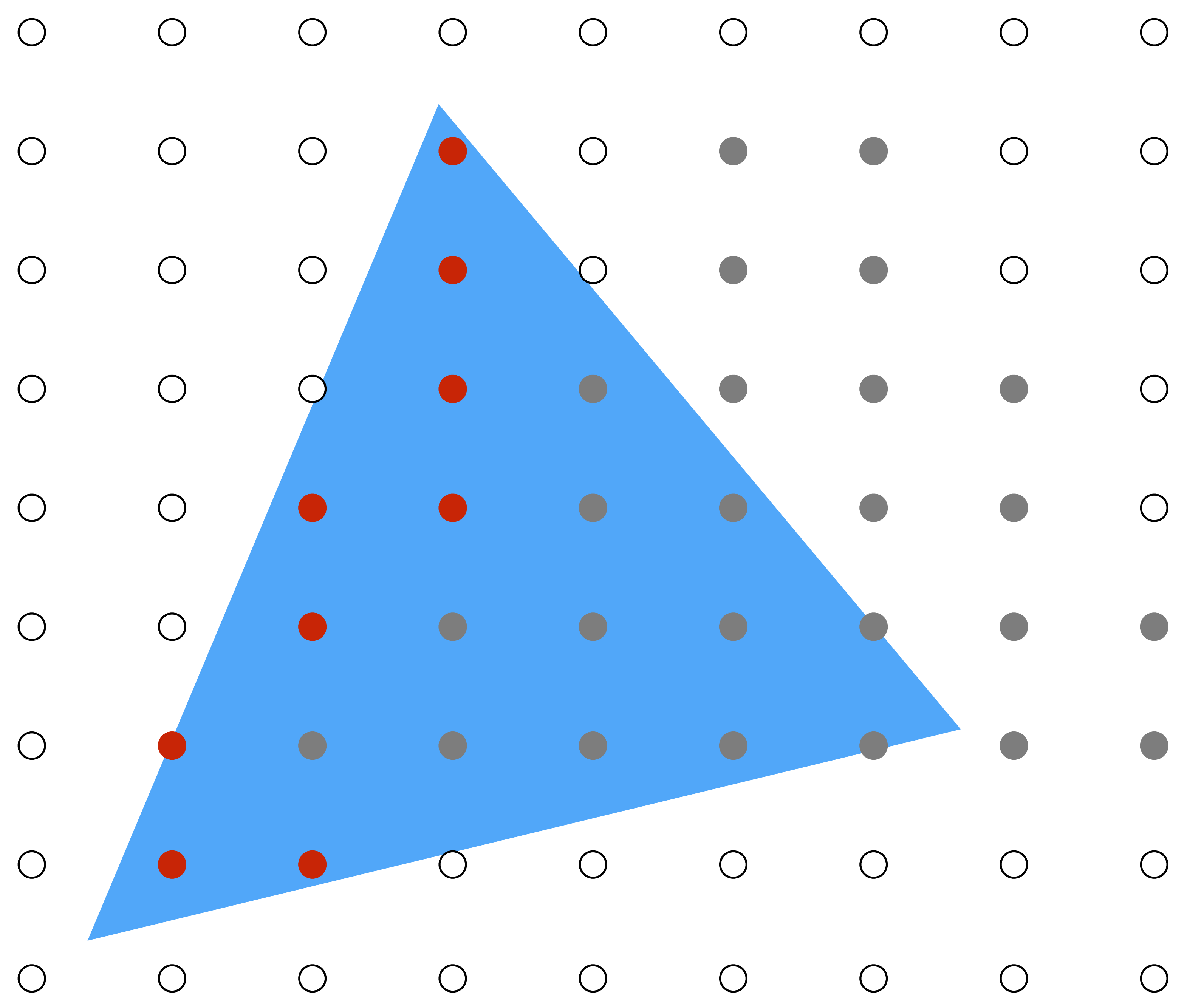
White = large distance

Black = small distance

Red = samples that pass depth test



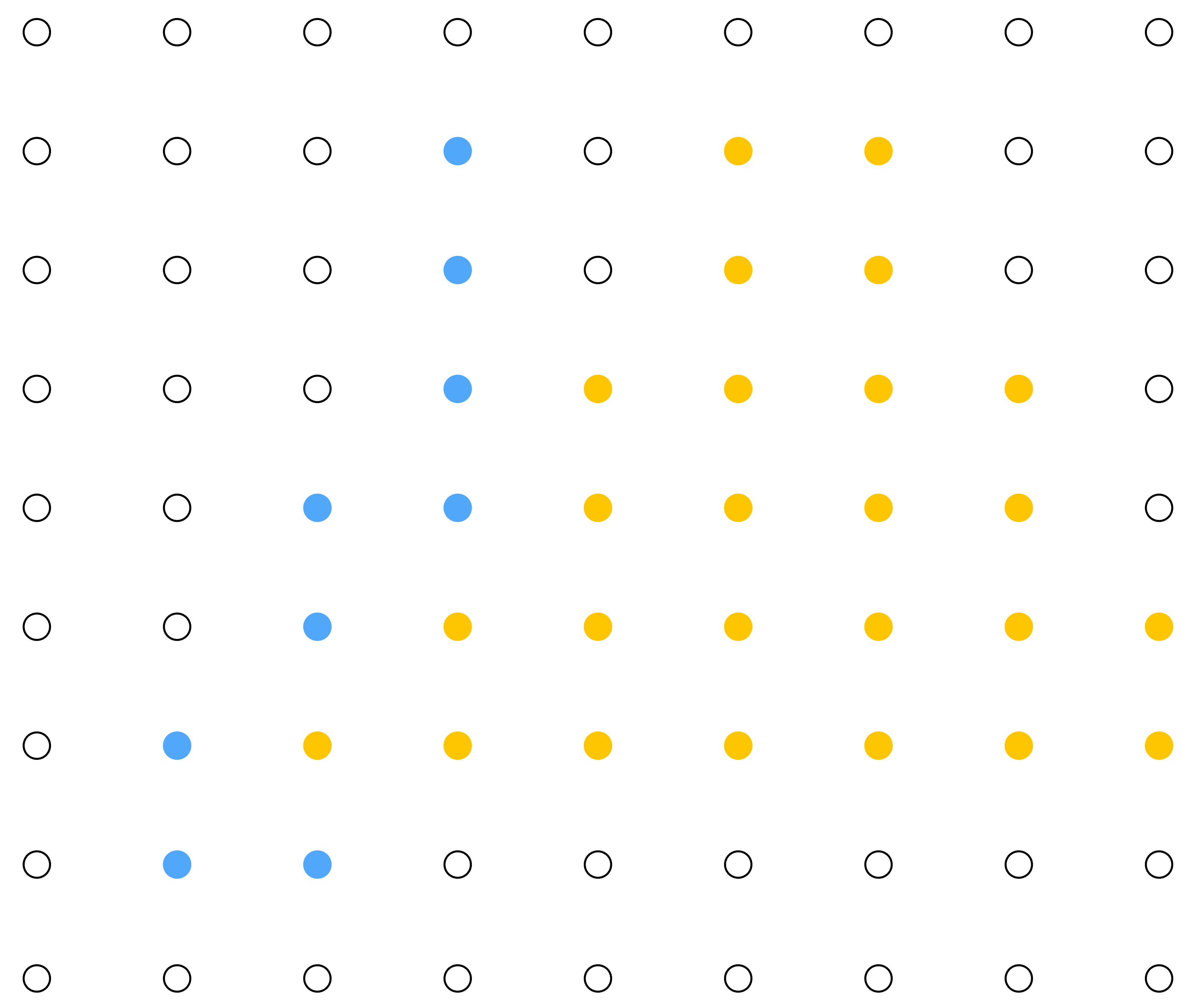
Color buffer contents



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing blue triangle:



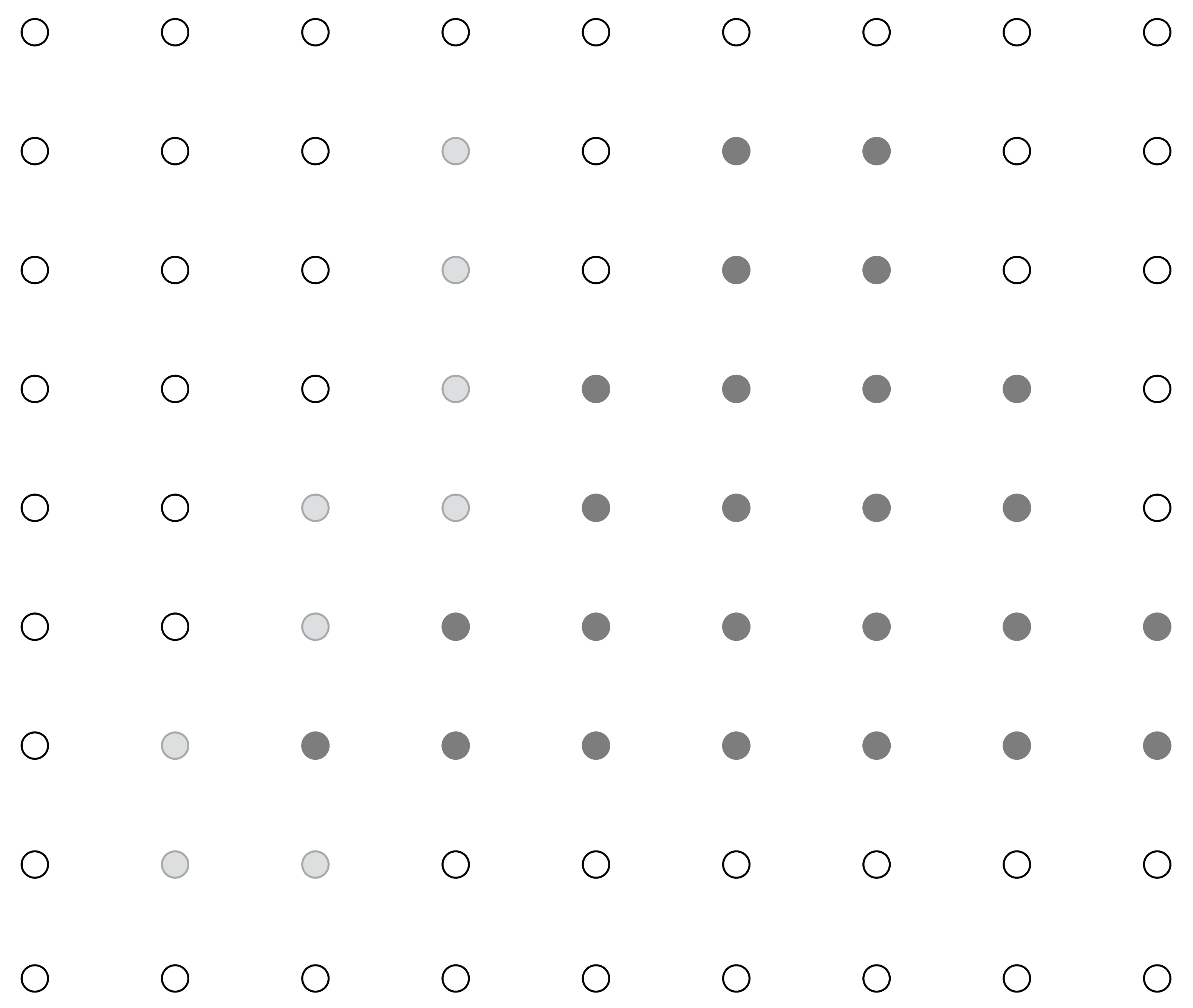
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = samples that pass depth test

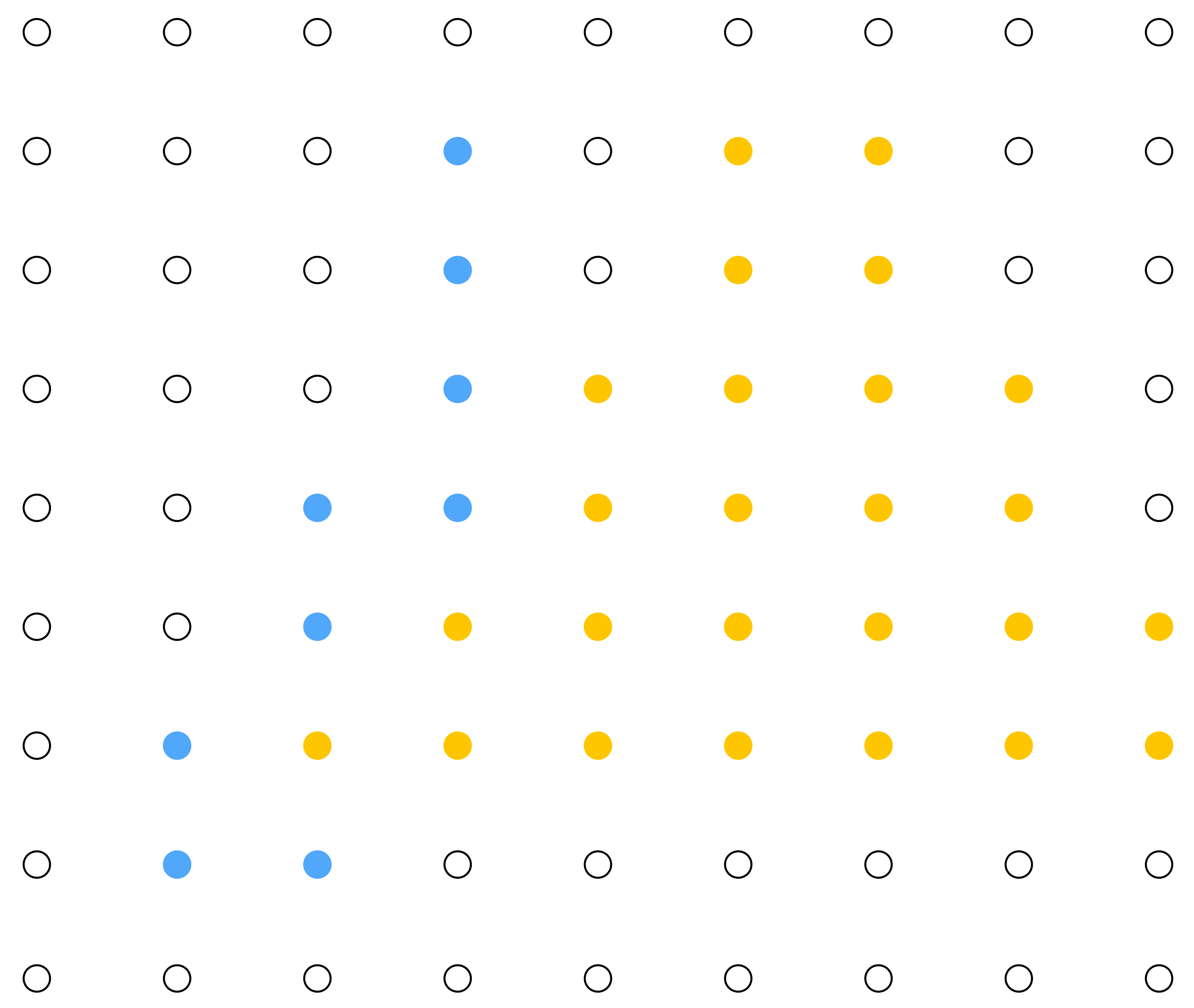


Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:

depth = 0.25



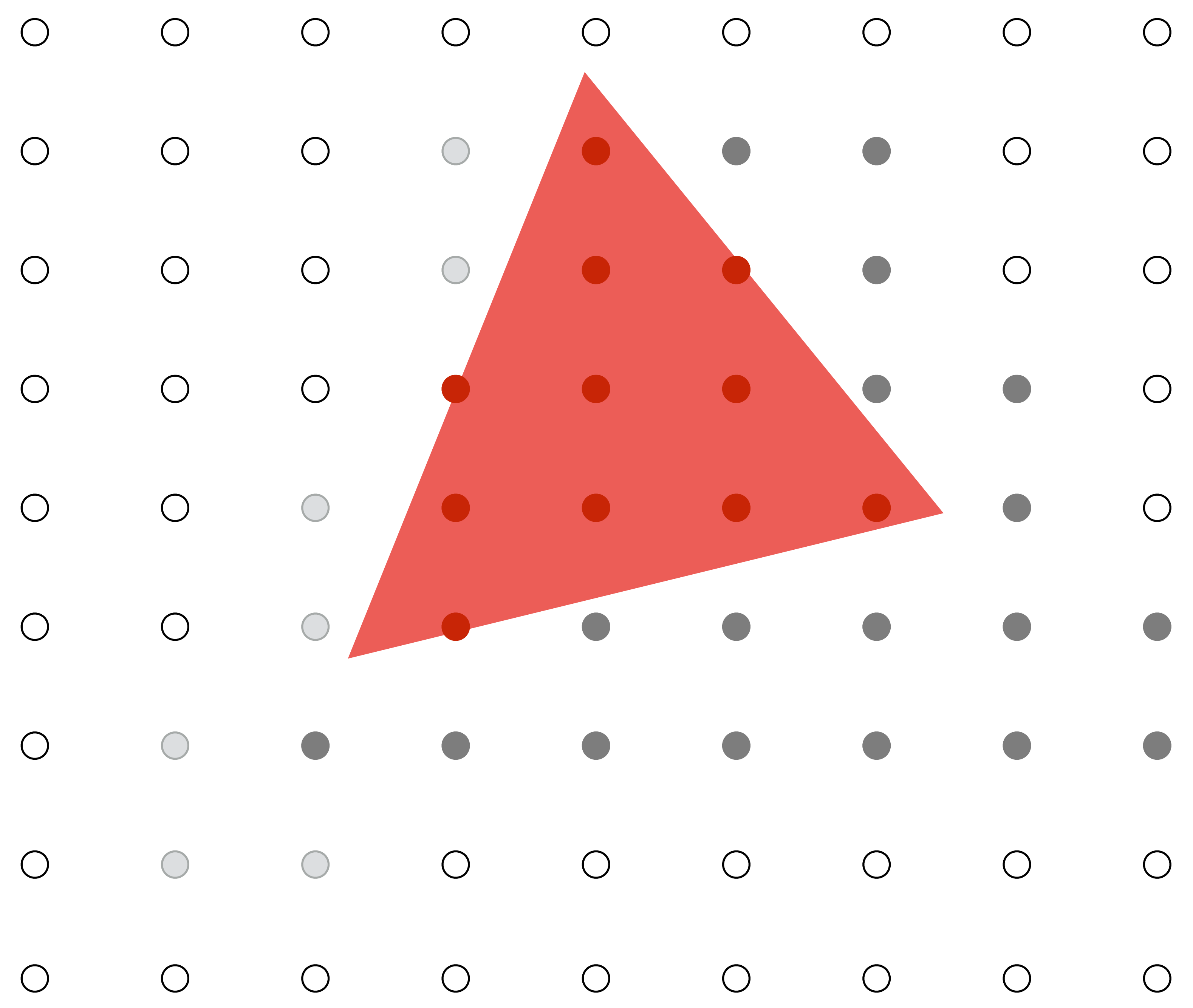
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

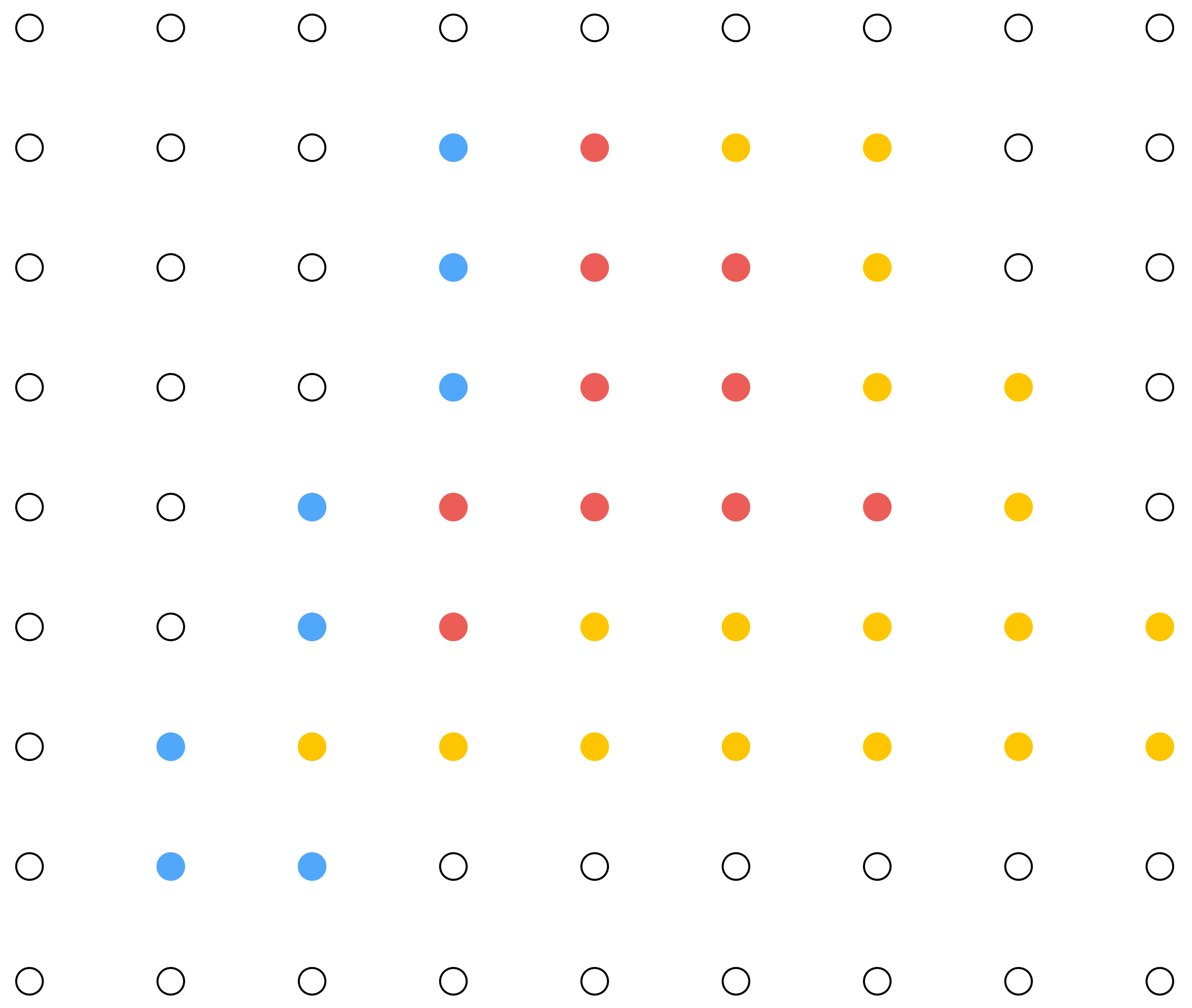
Red = samples that pass depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:



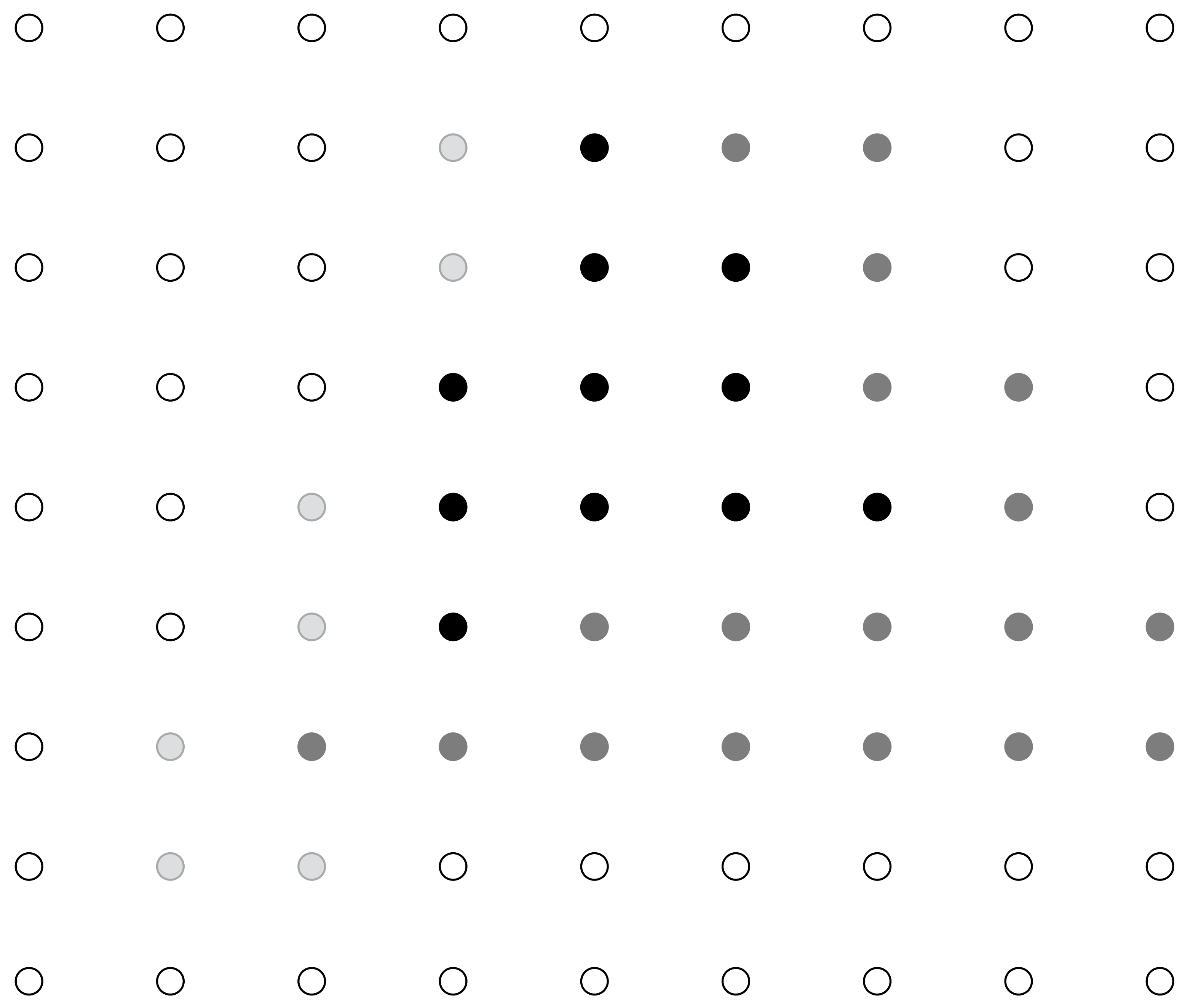
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = samples that pass depth test



Depth buffer contents

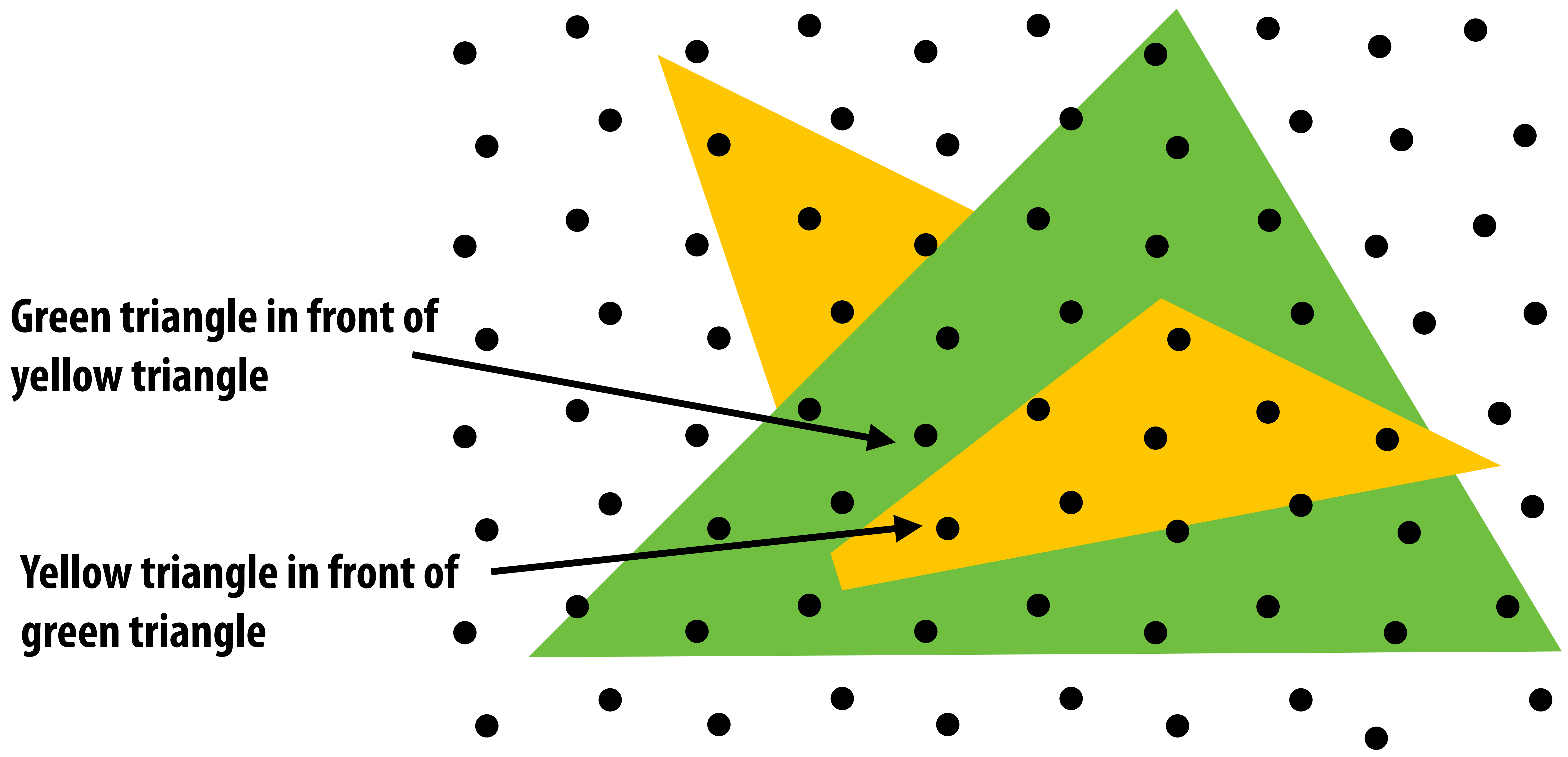
Occlusion using the depth buffer (opaque surfaces)

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


depth_test(tri_d, tri_color, x, y) {

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

    // if triangle is closest object seen so far at this
    // sample point. Update depth and color buffers.

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

Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

Occlusion test is based on depth of triangles *at a given sample point*. The relative depth of triangles may be different at different sample points.

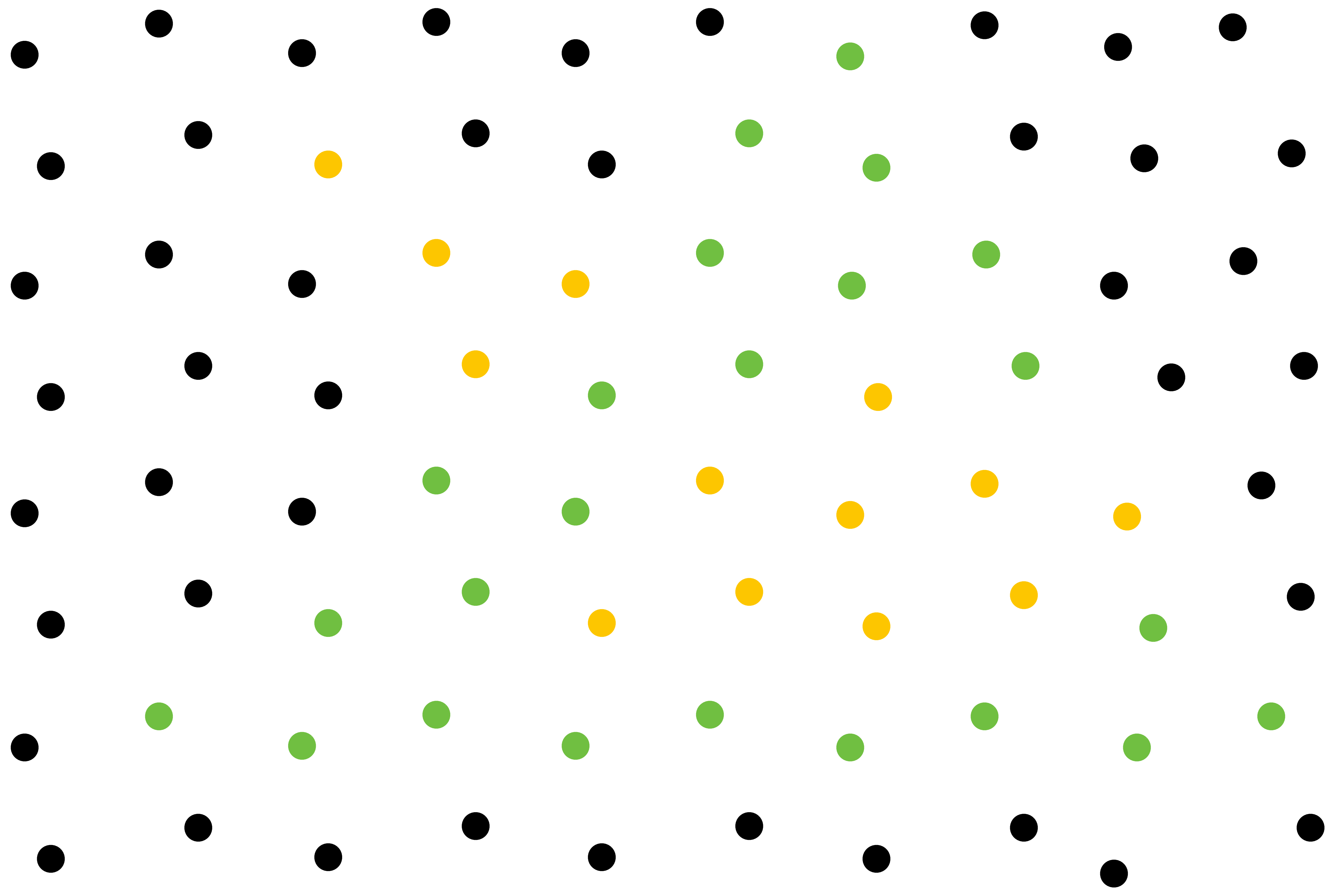

Green triangle in front of yellow triangle
Yellow triangle in front of green triangle

Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

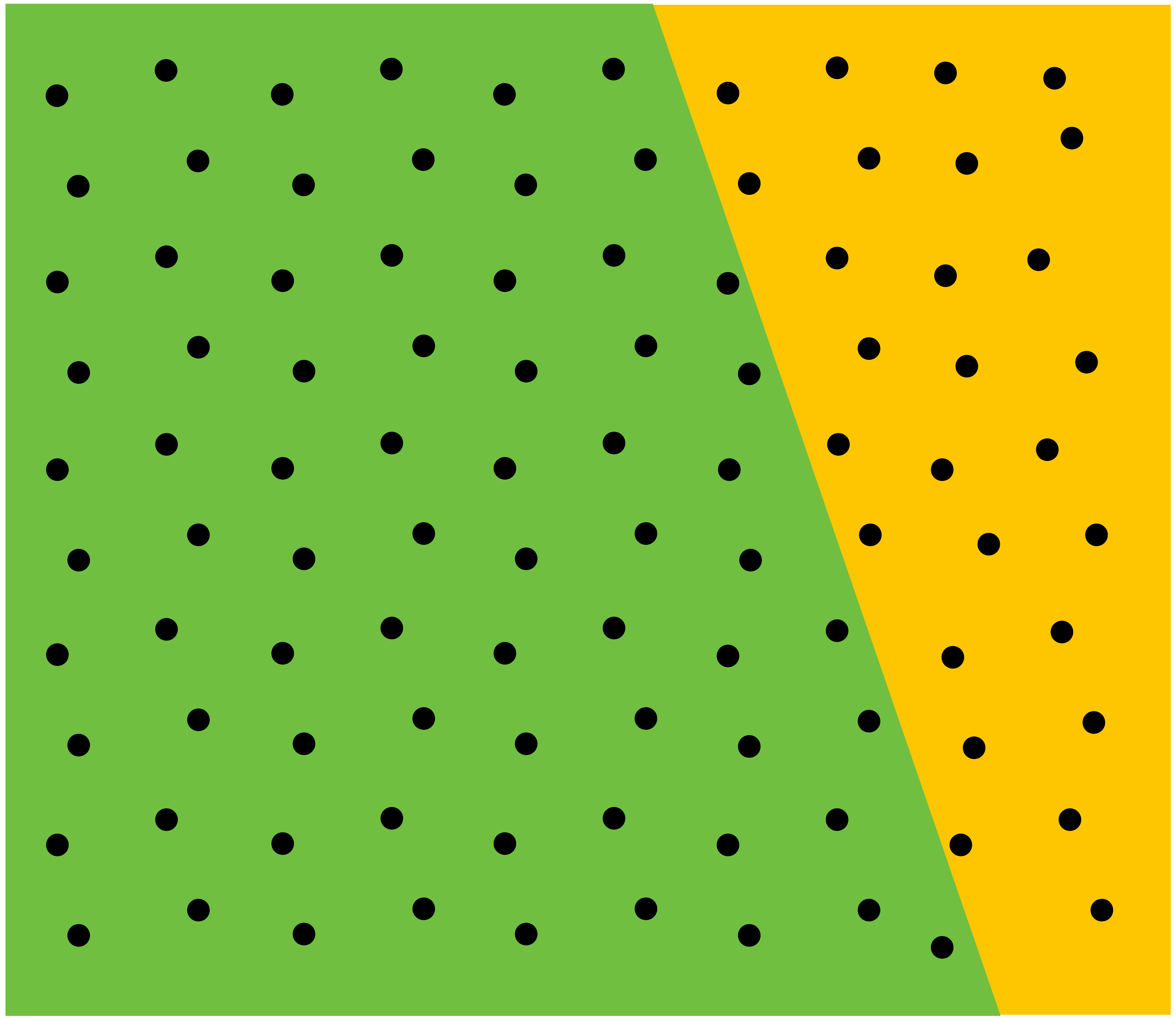
Occlusion test is based on depth of triangles *at a given sample point.* The relative depth of triangles may be different at different sample points.

Now only showing colored samples:



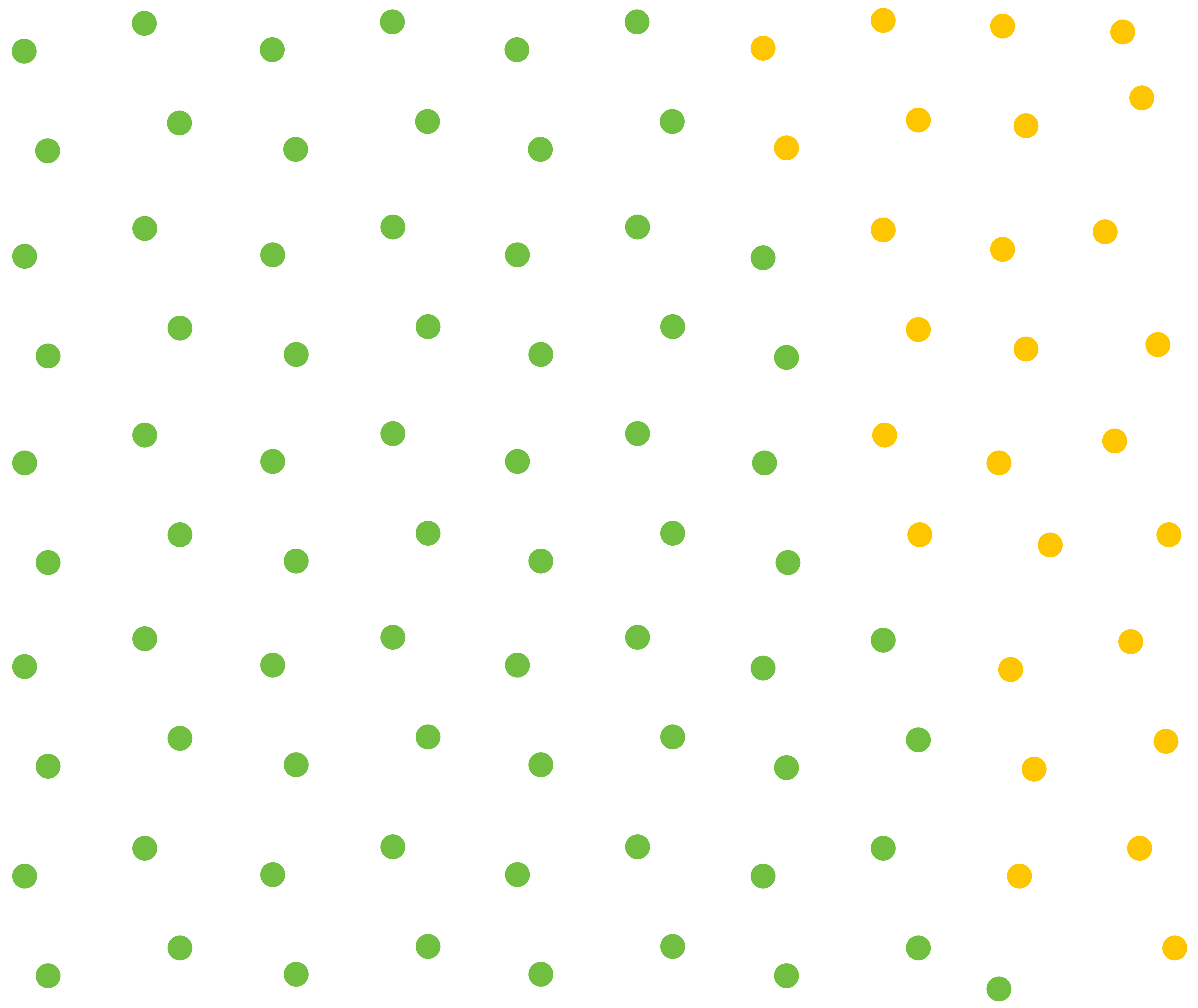
Does depth buffer work with super sampling?

Of course! Occlusion test is per sample, not per pixel!

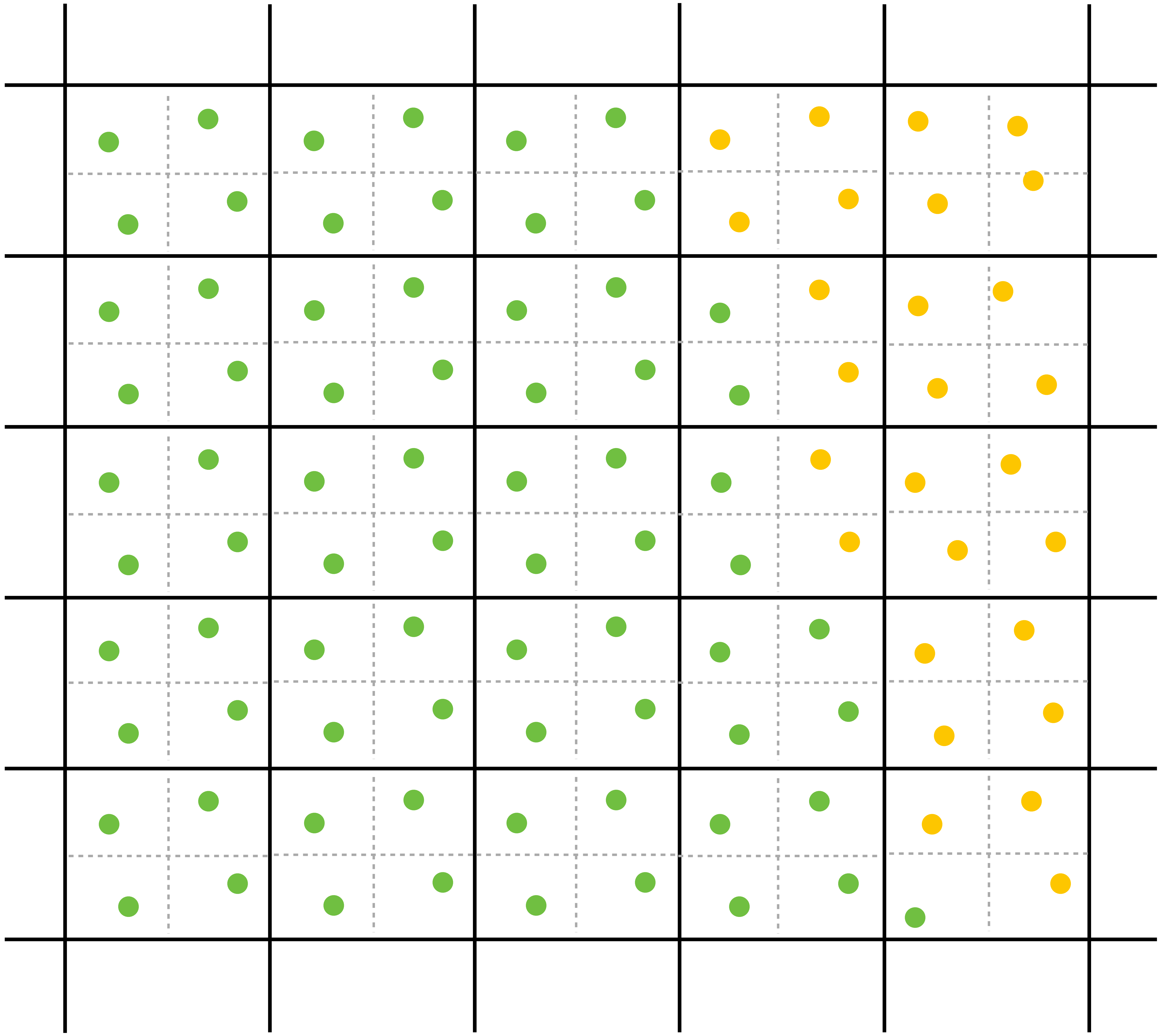


This example: green triangle occludes yellow triangle

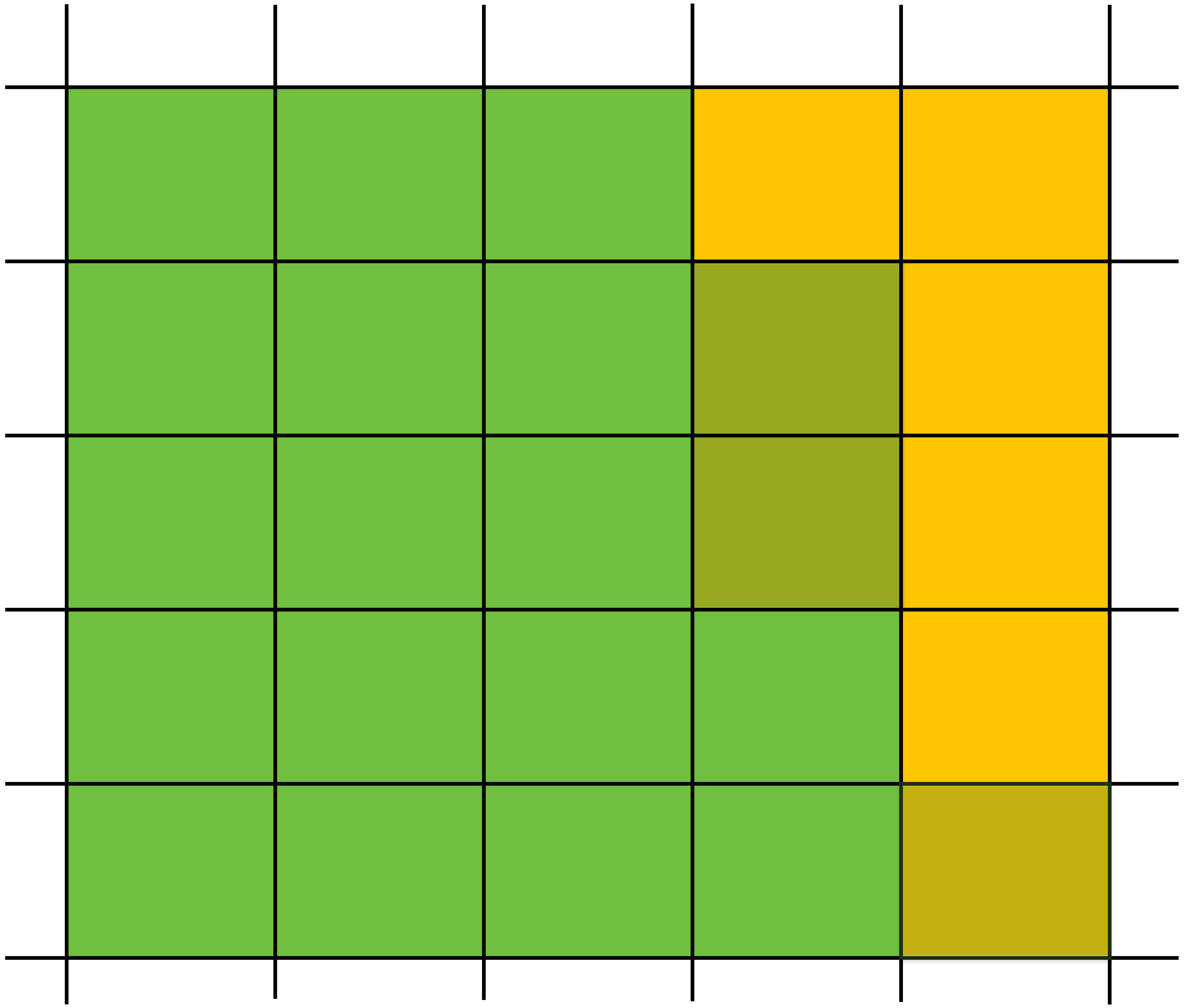
Color buffer contents



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 coverage sample (not per pixel!)
- Constant space per sample
  - Implication: constant space for depth buffer
- Constant time occlusion test per covered sample
  - Read-modify write of depth buffer if "pass" depth test
  - Just a depth buffer 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?

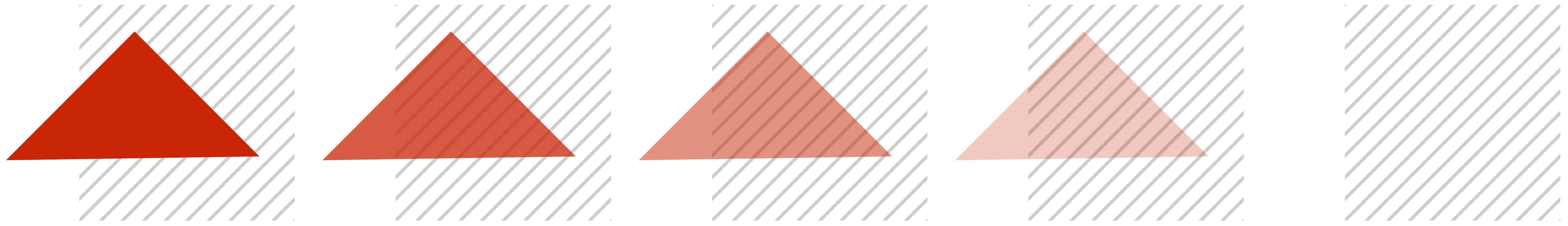
Compositing

Representing opacity as alpha

Alpha describes the opacity of an object

- Fully opaque surface: $\alpha = 1$
- 50% transparent surface: $\alpha = 0.5$
- Fully transparent surface: $\alpha = 0$

Red triangle with decreasing opacity

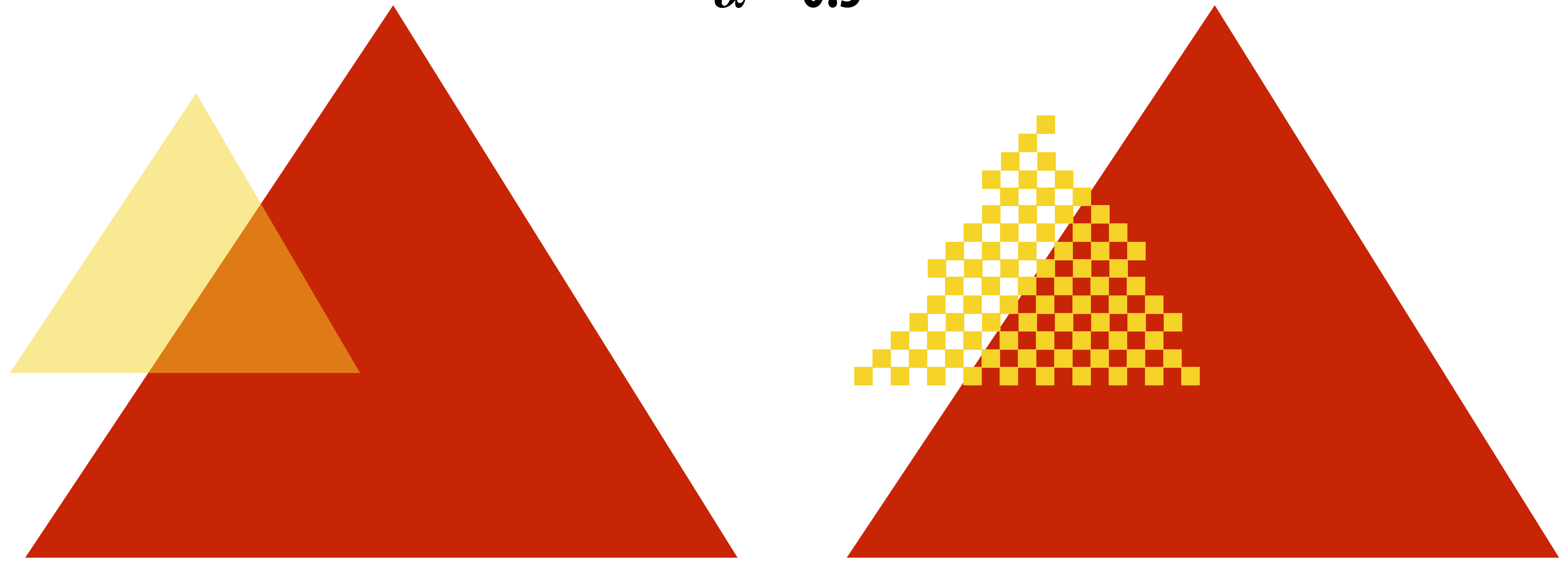


$\alpha = 1$ $\alpha = 0.75$ $\alpha = 0.5$ $\alpha = 0.25$ $\alpha = 0$

Alpha: coverage analogy

- Can think of alpha as describing the opacity of a semi-transparent surface
- Or… as partial coverage by fully opaque object
  - consider a screen door

$\alpha = 0.5$

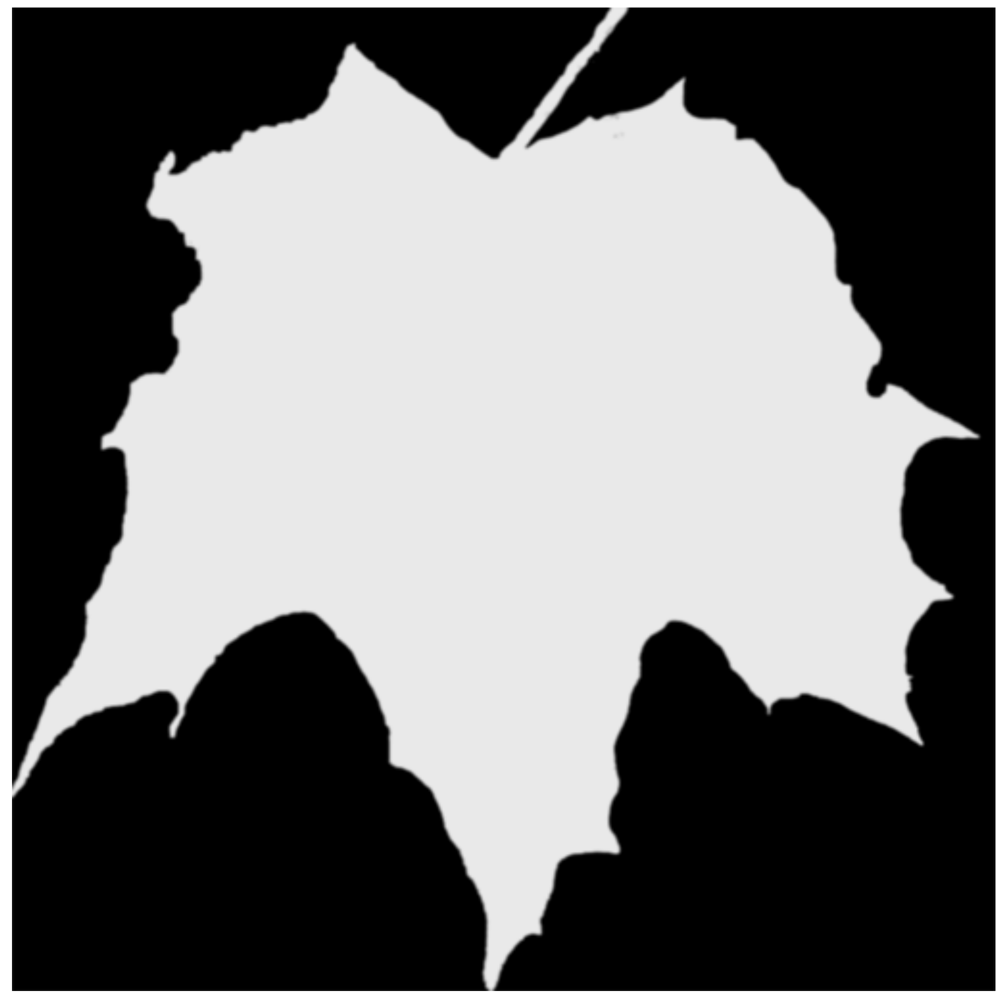


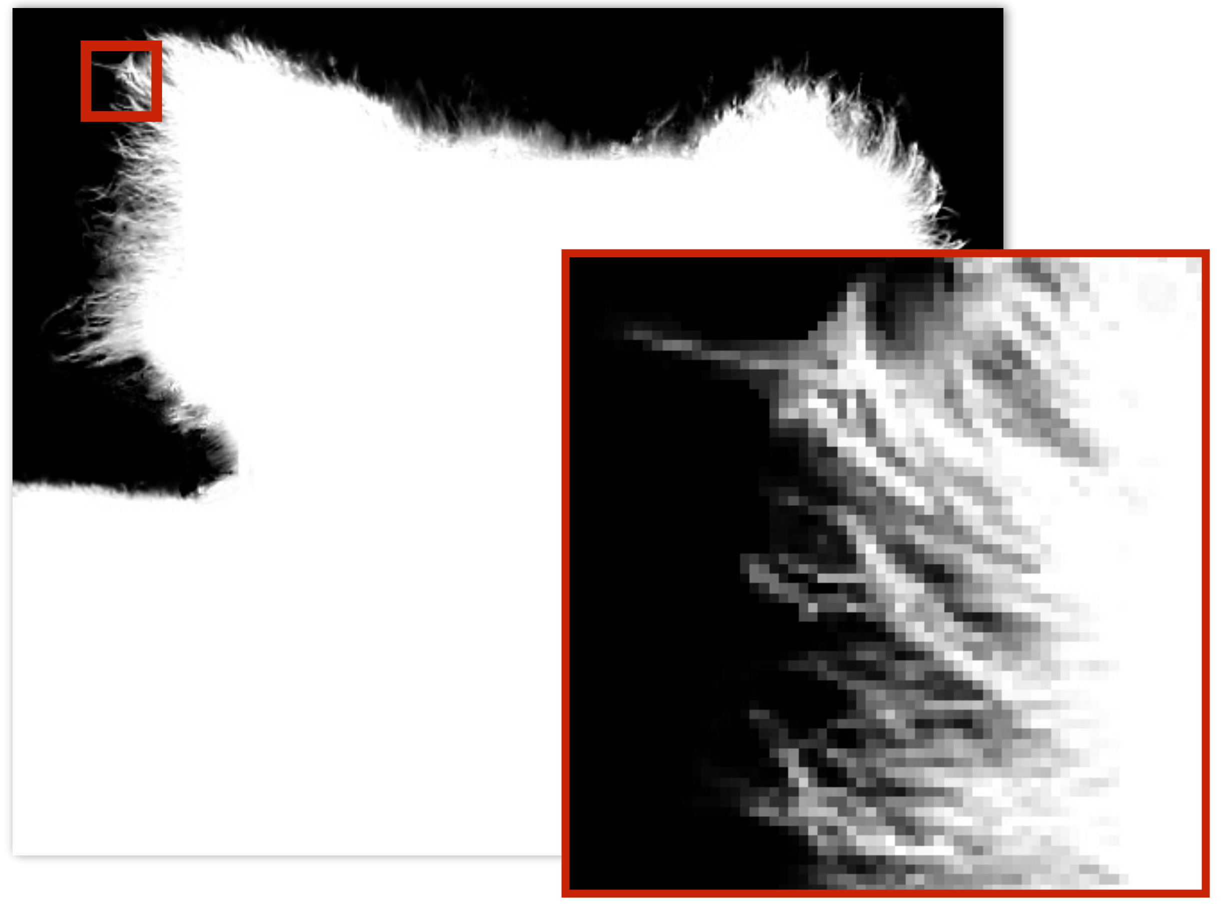
(Squint at this slide and the scene on the left and the right will appear similar)

Alpha: additional channel of image (rgba)

Alpha describes the opacity of an object

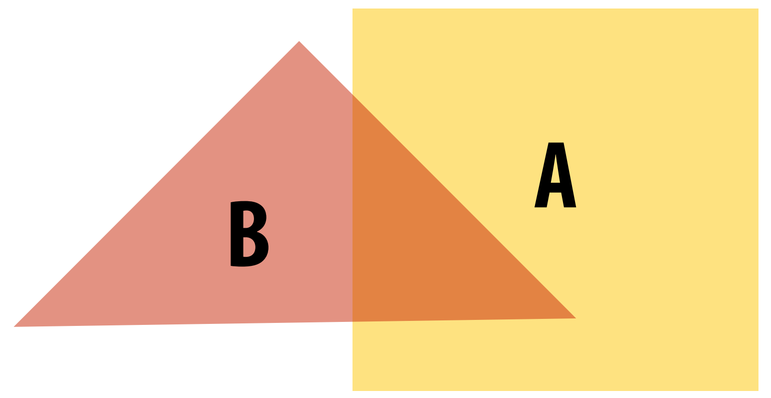
- Fully opaque surface: $\alpha = 1$
- 50% transparent surface: $\alpha = 0.5$
- Fully transparent surface: $\alpha = 0$



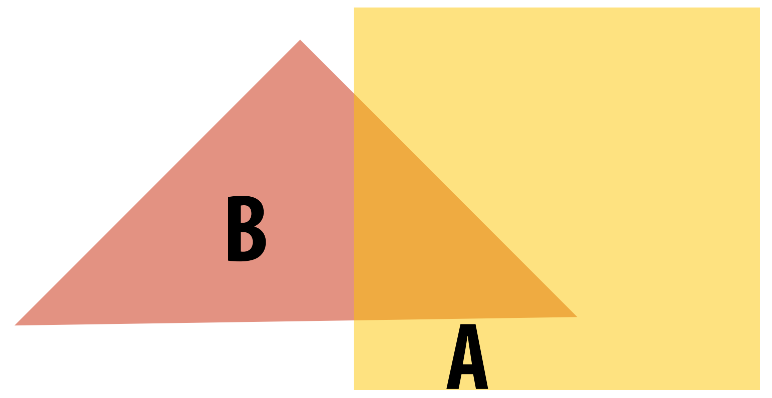

α of foreground object

Over operator:

Composite image B with opacity α_B <u>over</u> image A with opacity α_A



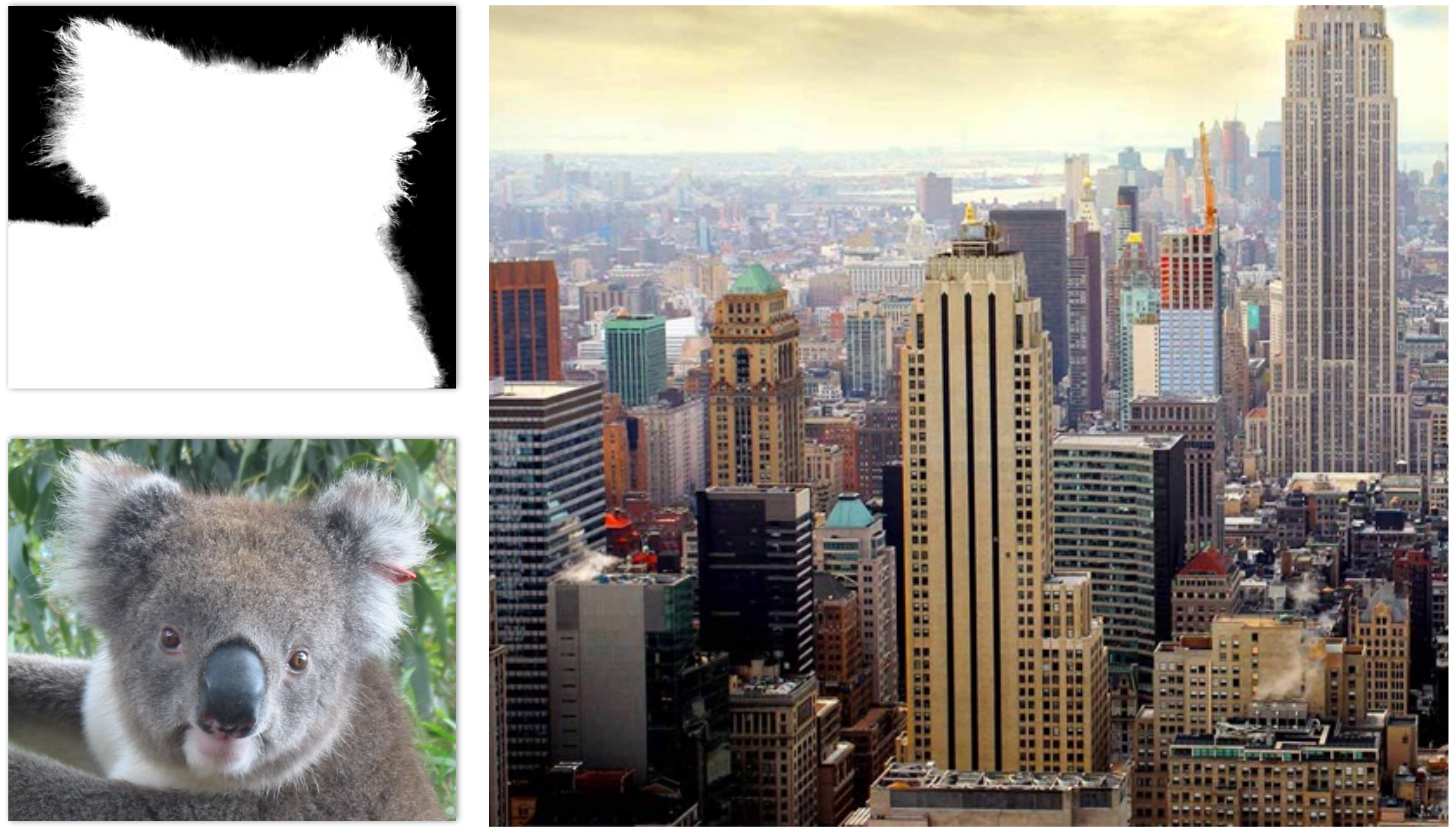
B over A

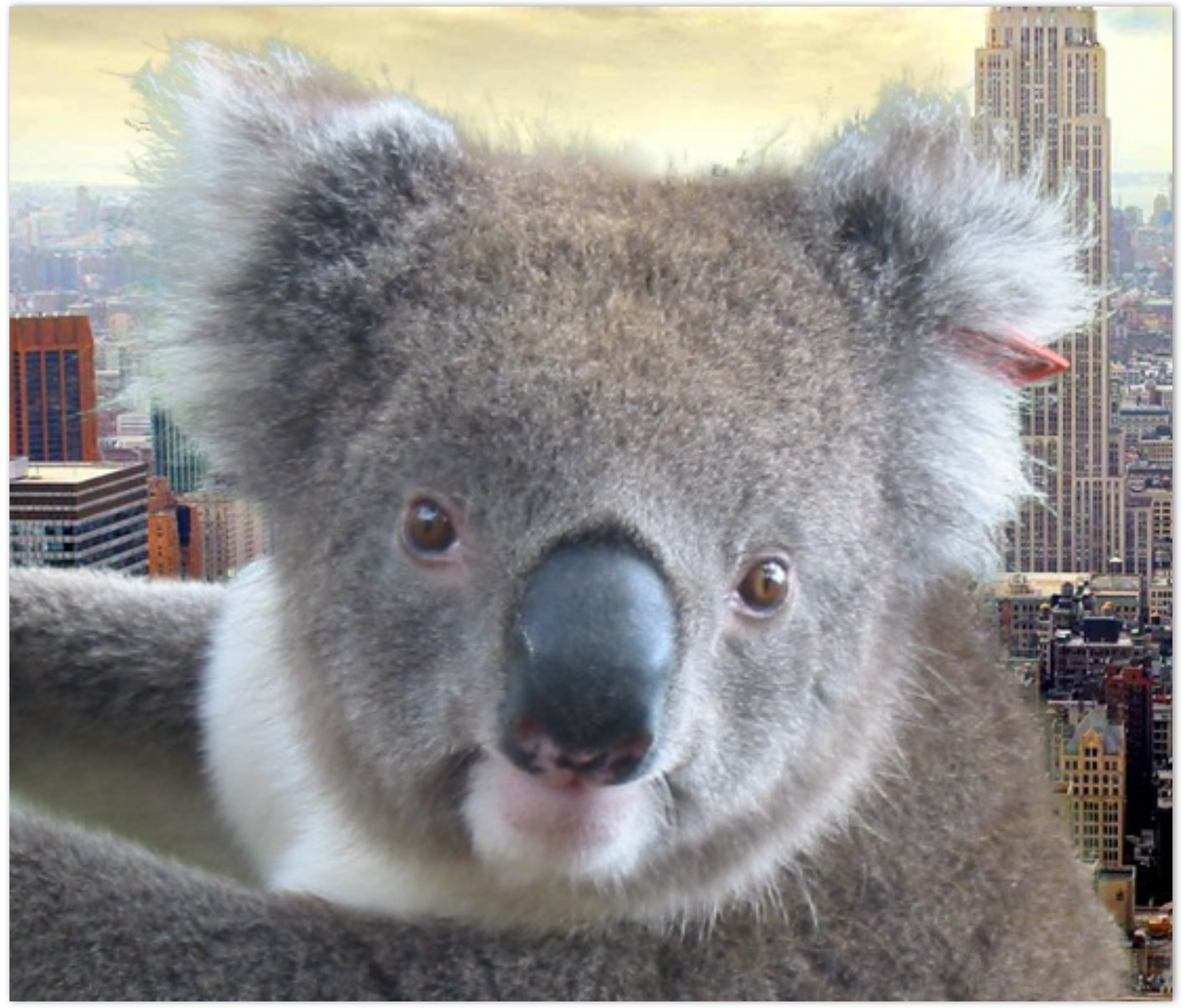


A over B

A over B != B over A

"Over" is not commutative





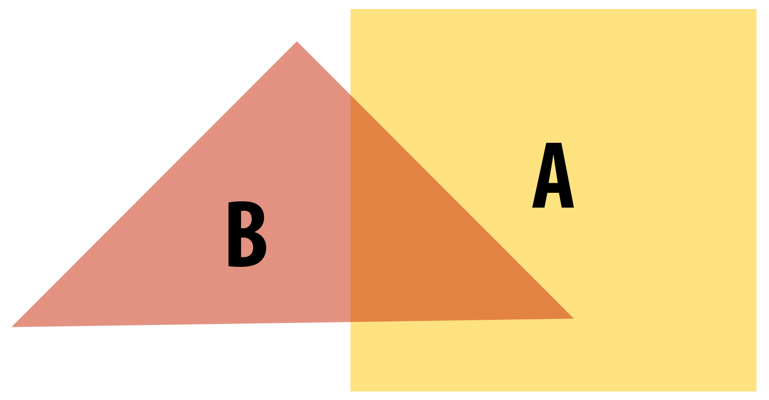
Koala over NYC

Over operator: non-premultiplied alpha

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

First attempt: (represent colors as 3-vectors, alpha separately)

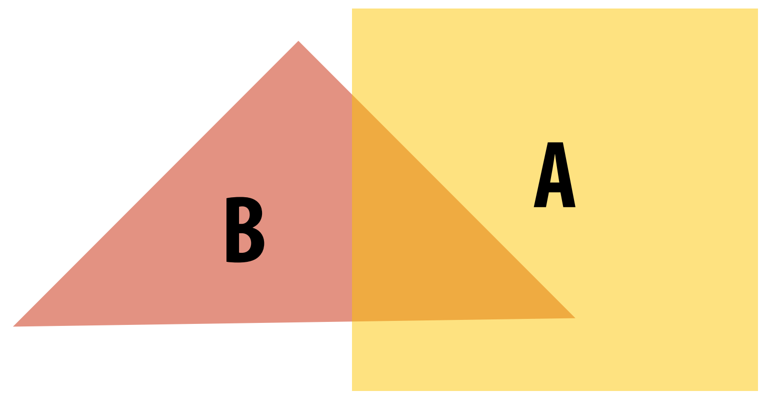
$$A = \begin{bmatrix} A_r & A_g & A_b \end{bmatrix}^T$$
$$B = \begin{bmatrix} B_r & B_g & B_b \end{bmatrix}^T$$



B over A

Composited color:

$$C = \underbrace{\alpha_B B}_{\text{Appearance of semi-transparent B}} + \underbrace{(1 - \alpha_B)}_{\text{What B lets through}} \overbrace{\alpha_A A}^{\text{Appearance of semi-transparent A}}$$



A over B

Composite alpha:

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

A over B != B over A

"Over" is not commutative

Premultiplied alpha representation

- **Represent (potentially transparent) color as a 4-vector where RGB values have been premultiplied by alpha**

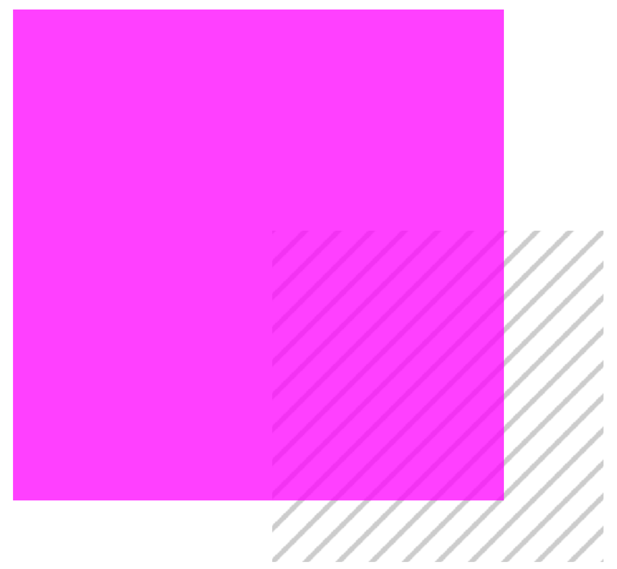
$$A' = \begin{bmatrix} \alpha_A A_r & \alpha_A A_g & \alpha_A A_b & \alpha_A \end{bmatrix}^T$$

Example: 50% opaque red

[0.5, 0.0, 0.0, 0.5]

Example: 75% opaque magenta

[0.75, 0.0, 0.75, 0.75]



Over operator: using premultiplied alpha

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

Non-premultiplied alpha representation:

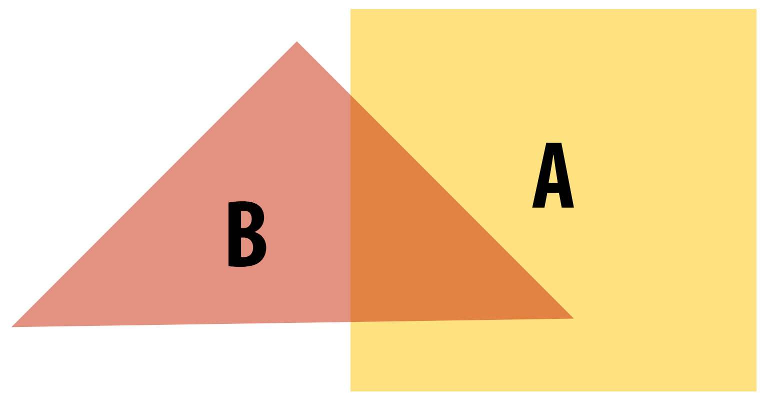
$$A = \begin{bmatrix} A_r & A_g & A_b \end{bmatrix}^T$$

$$B = \begin{bmatrix} B_r & B_g & B_b \end{bmatrix}^T$$

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

two multiplies, one add

(referring to vector ops on colors)



B over A

Composite alpha:

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

Premultiplied alpha representation:

$$A' = \begin{bmatrix} \alpha_A A_r & \alpha_A A_g & \alpha_A A_b & \alpha_A \end{bmatrix}^T$$

$$B' = \begin{bmatrix} \alpha_B B_r & \alpha_B B_g & \alpha_B B_b & \alpha_B \end{bmatrix}^T$$

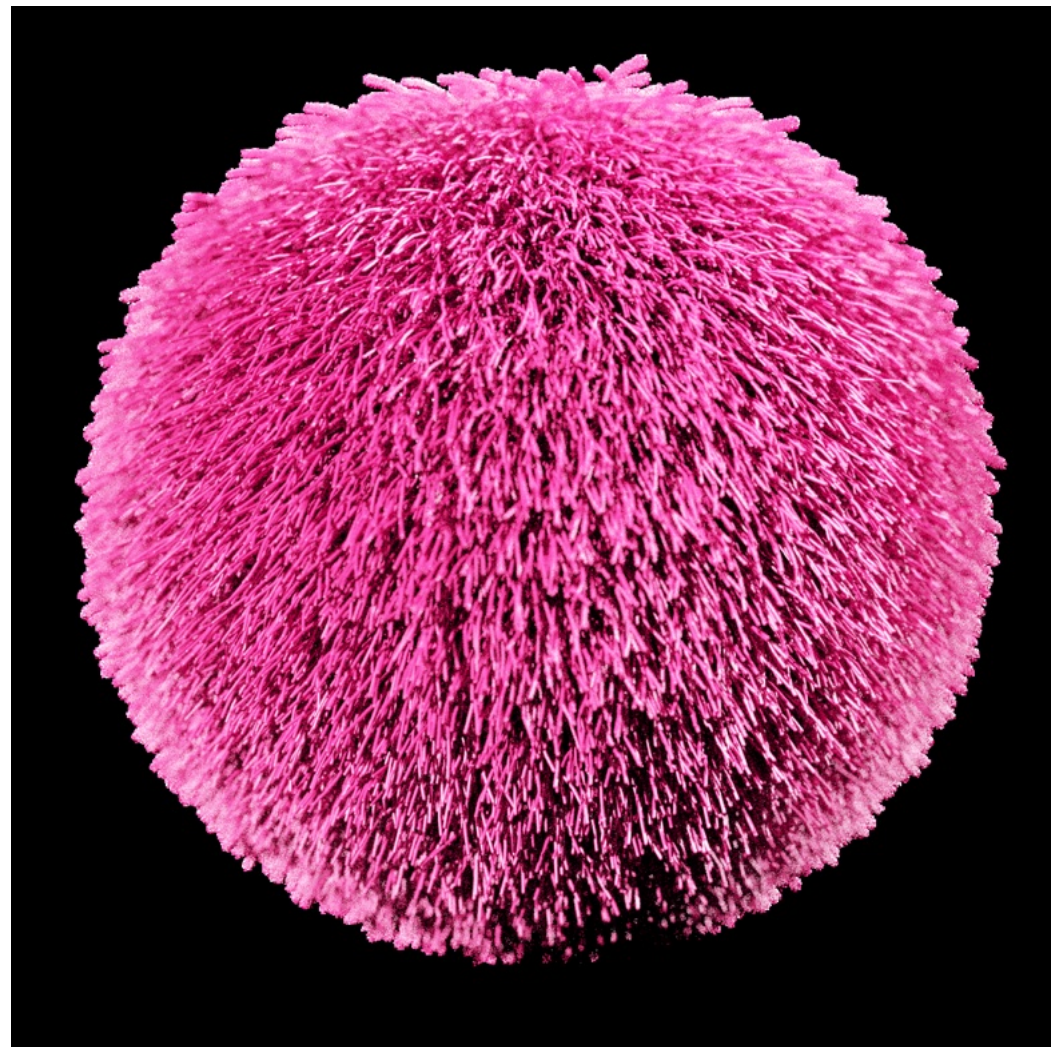
$$C' = B' + (1 - \alpha_B)A' \longleftarrow$$

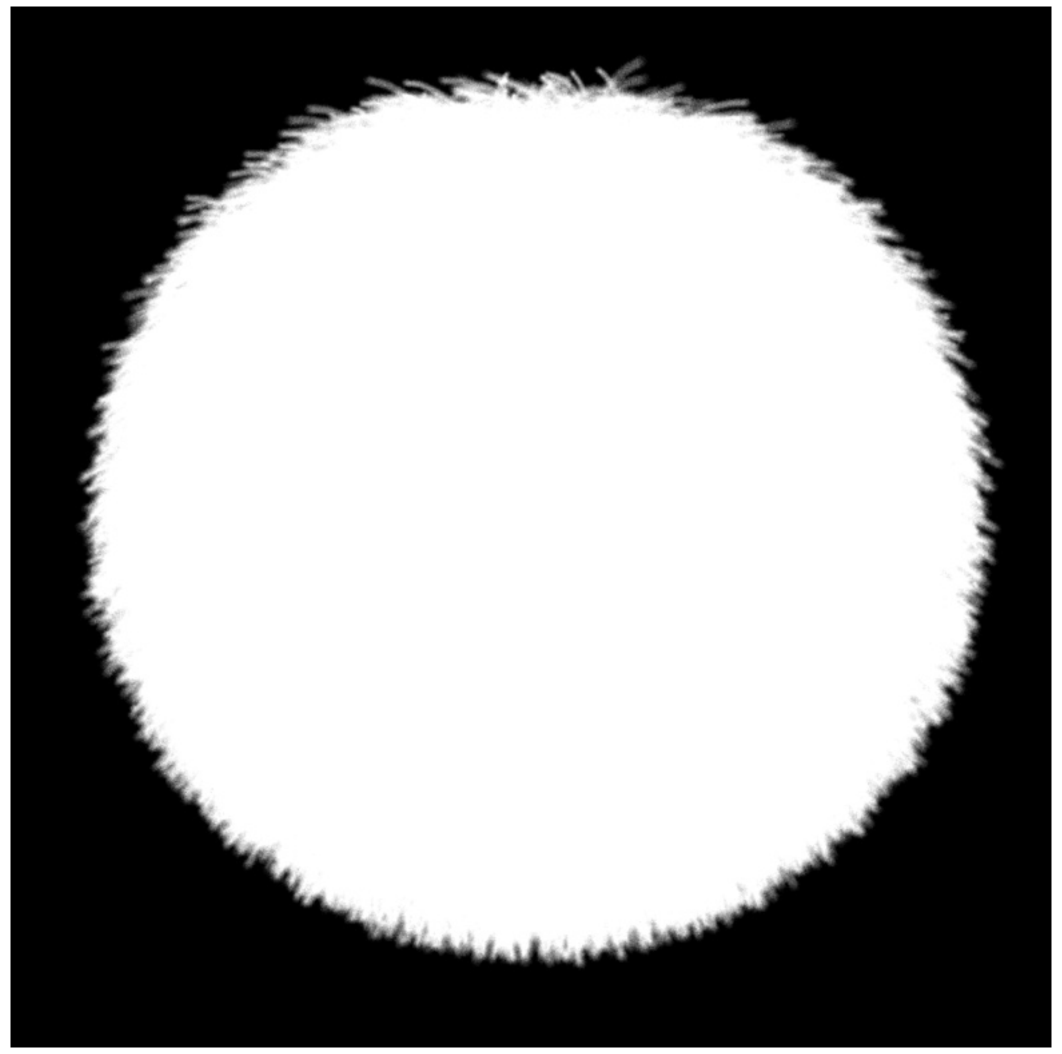
one multiply, one add

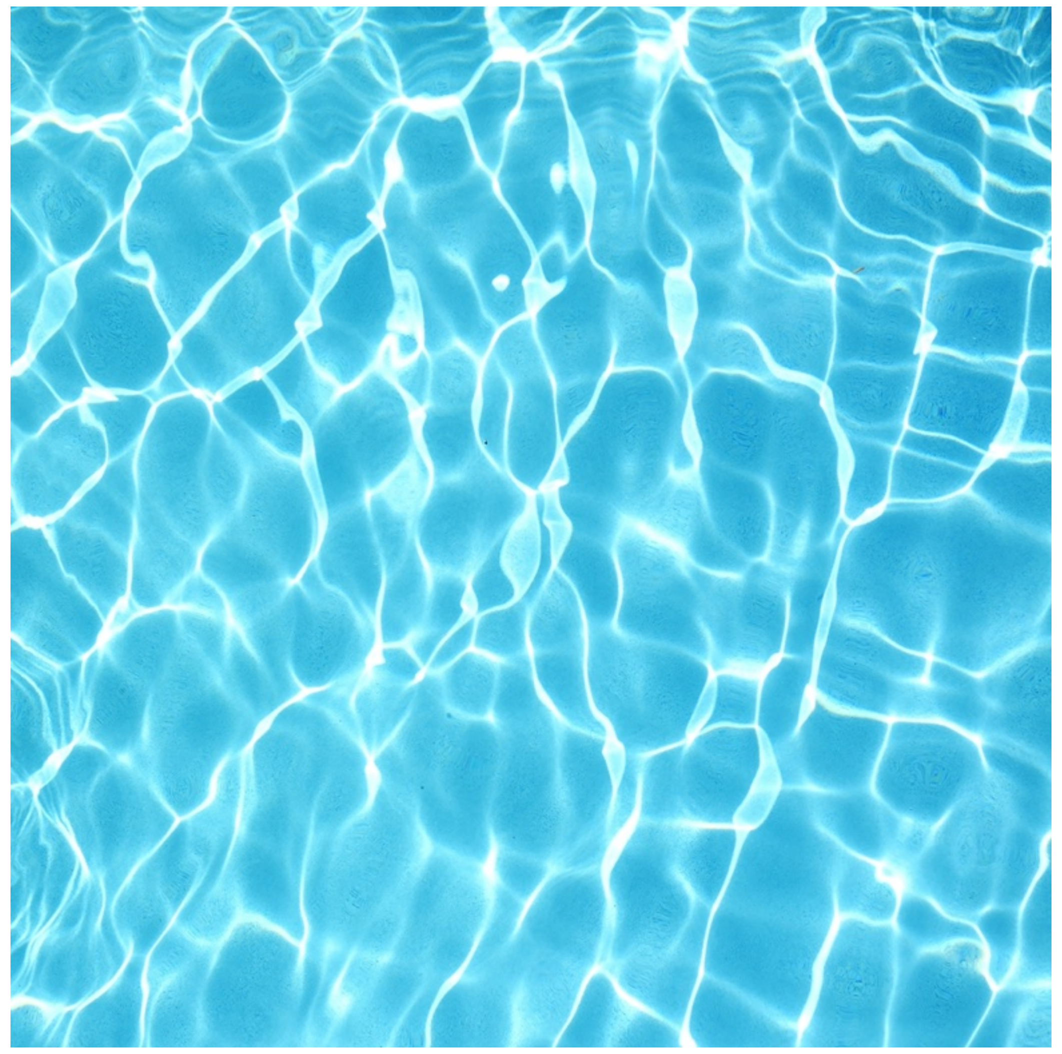
Notice premultiplied alpha composites alpha just like how it composites rgb.

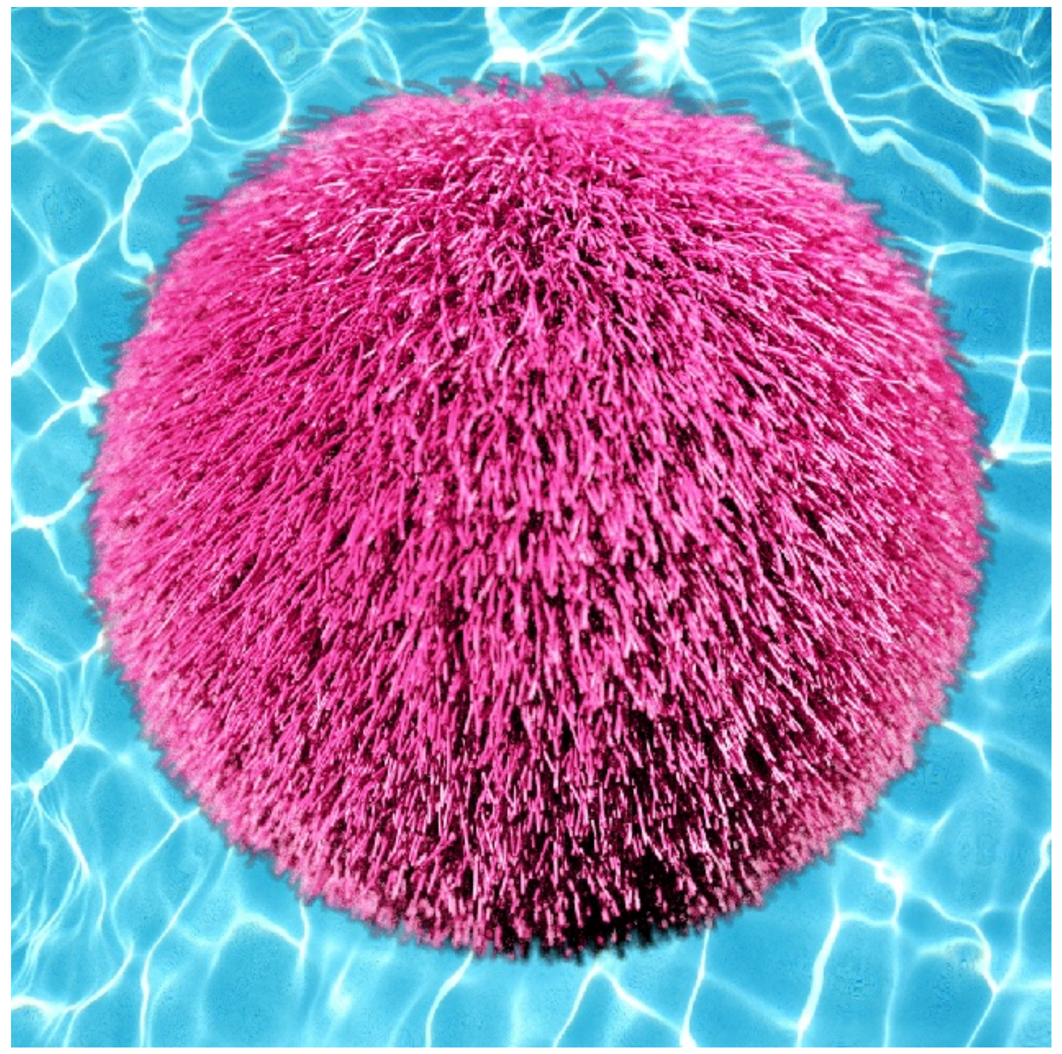
Fringing

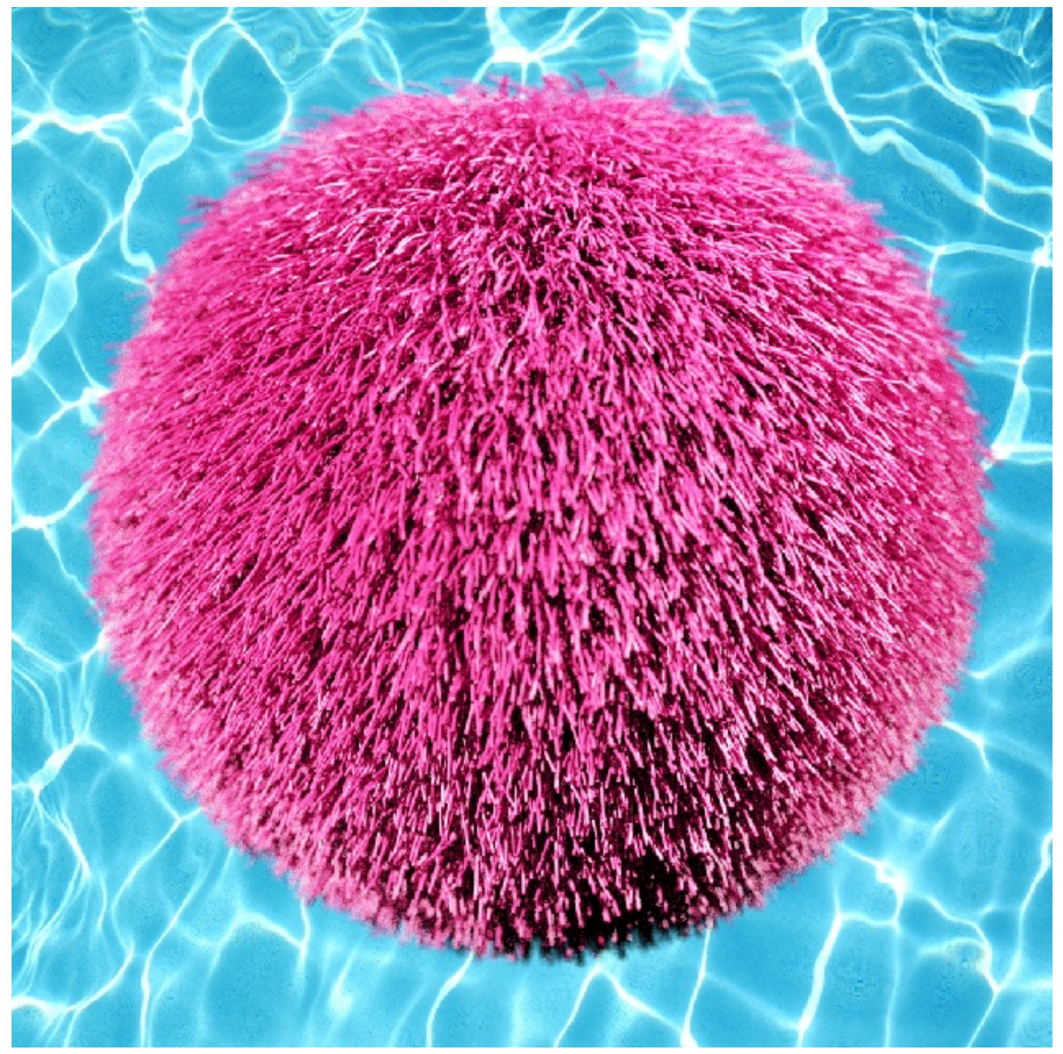
Poor treatment of color/alpha can yield dark "fringing":


foreground color

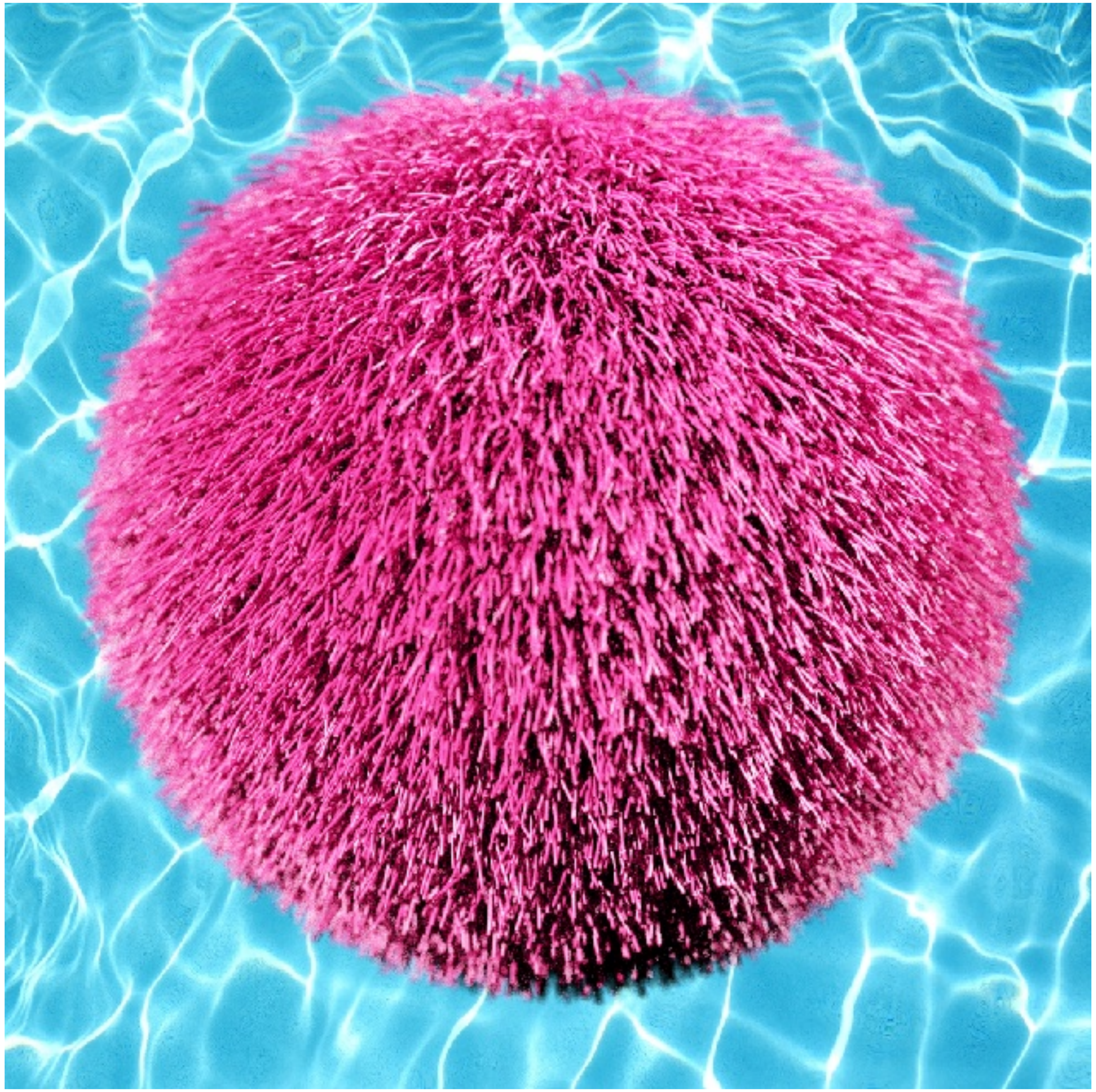

foreground alpha


background color

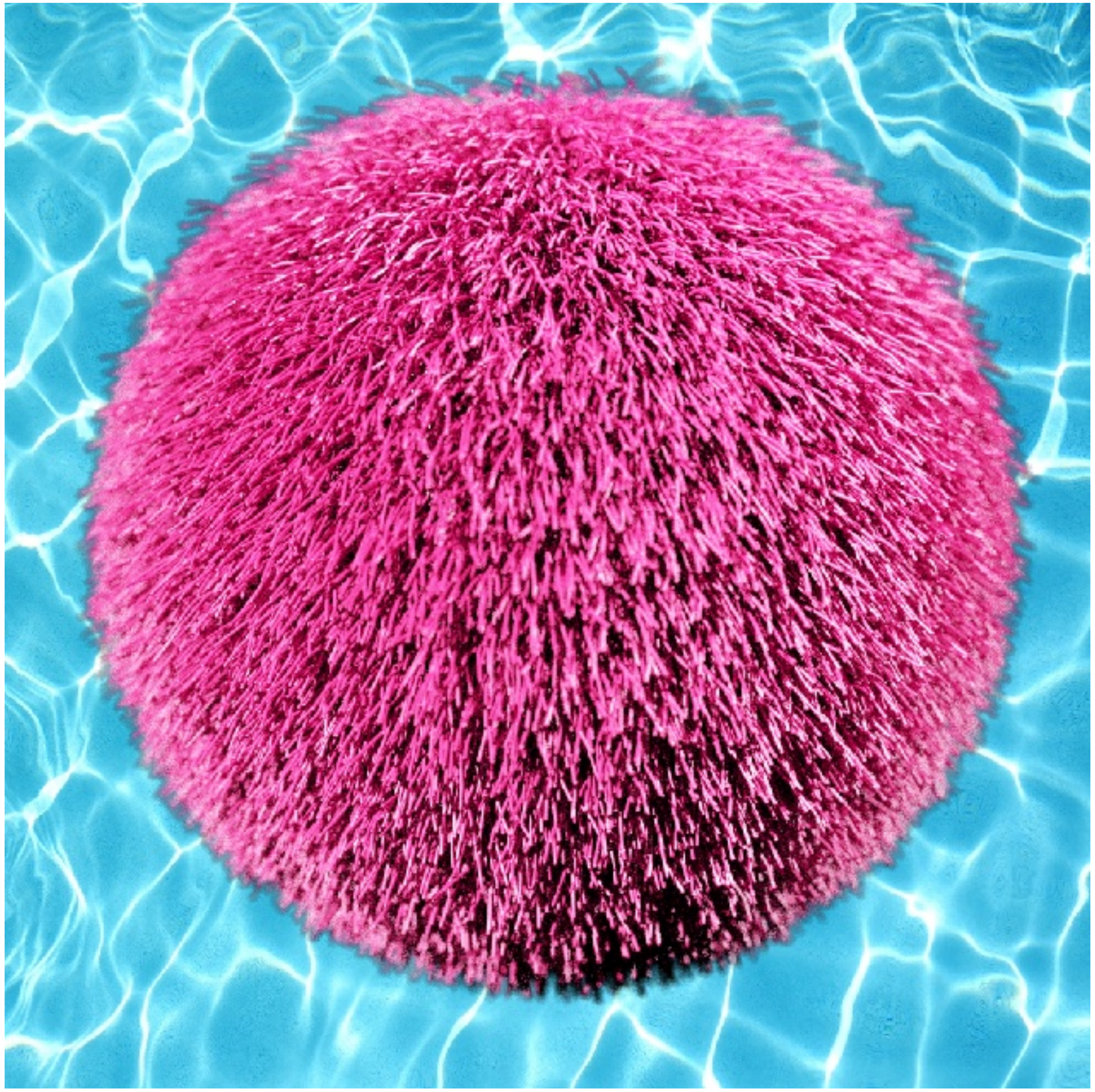

fringing


no fringing

No fringing

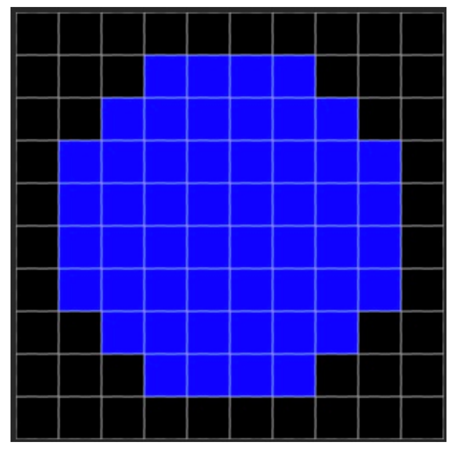


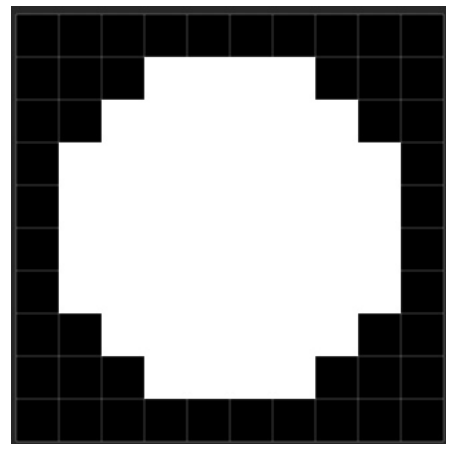
Fringing (…why does this happen?)

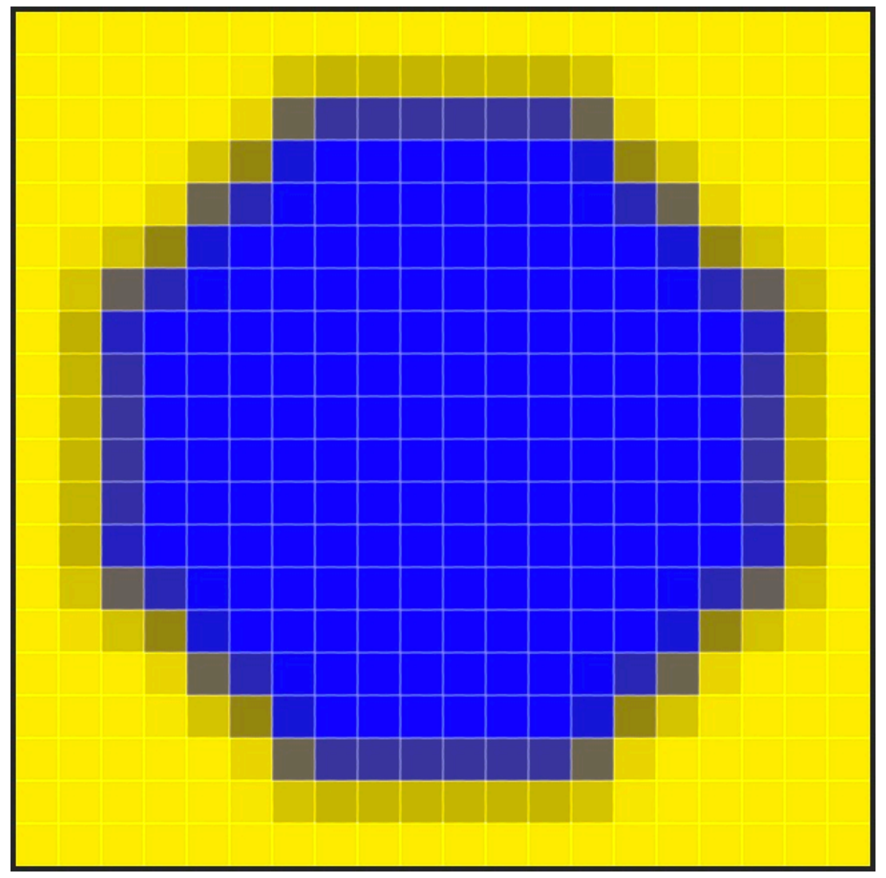


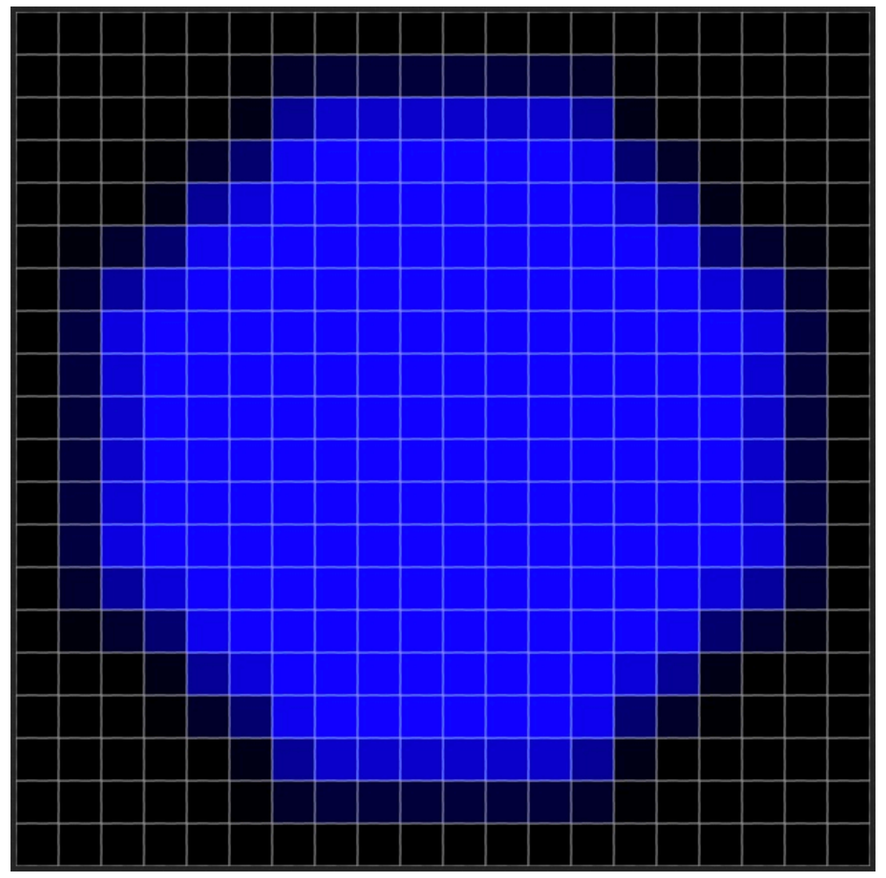
A problem with non-premultiplied alpha

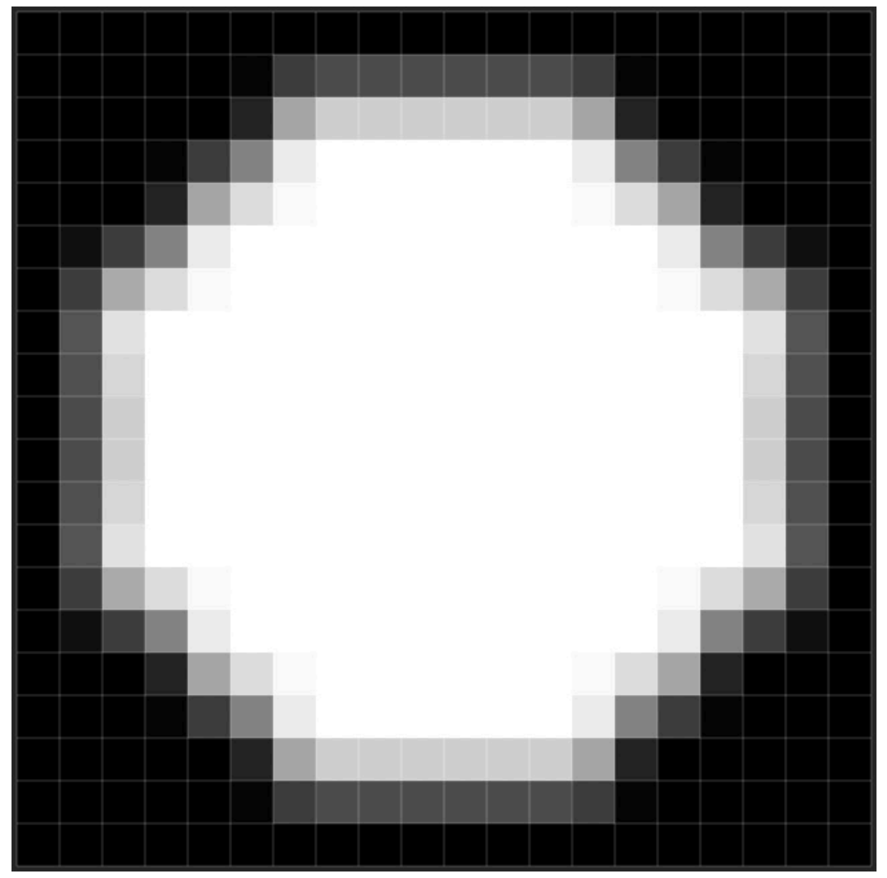
- Suppose we upsample an image w/ an alpha mask, then composite it onto a background
- How should we compute the interpolated color/alpha values?
- If we interpolate color and alpha separately, then blend using the non-premultiplied "over" operator, here's what happens:


original color


original alpha


composited onto yellow background

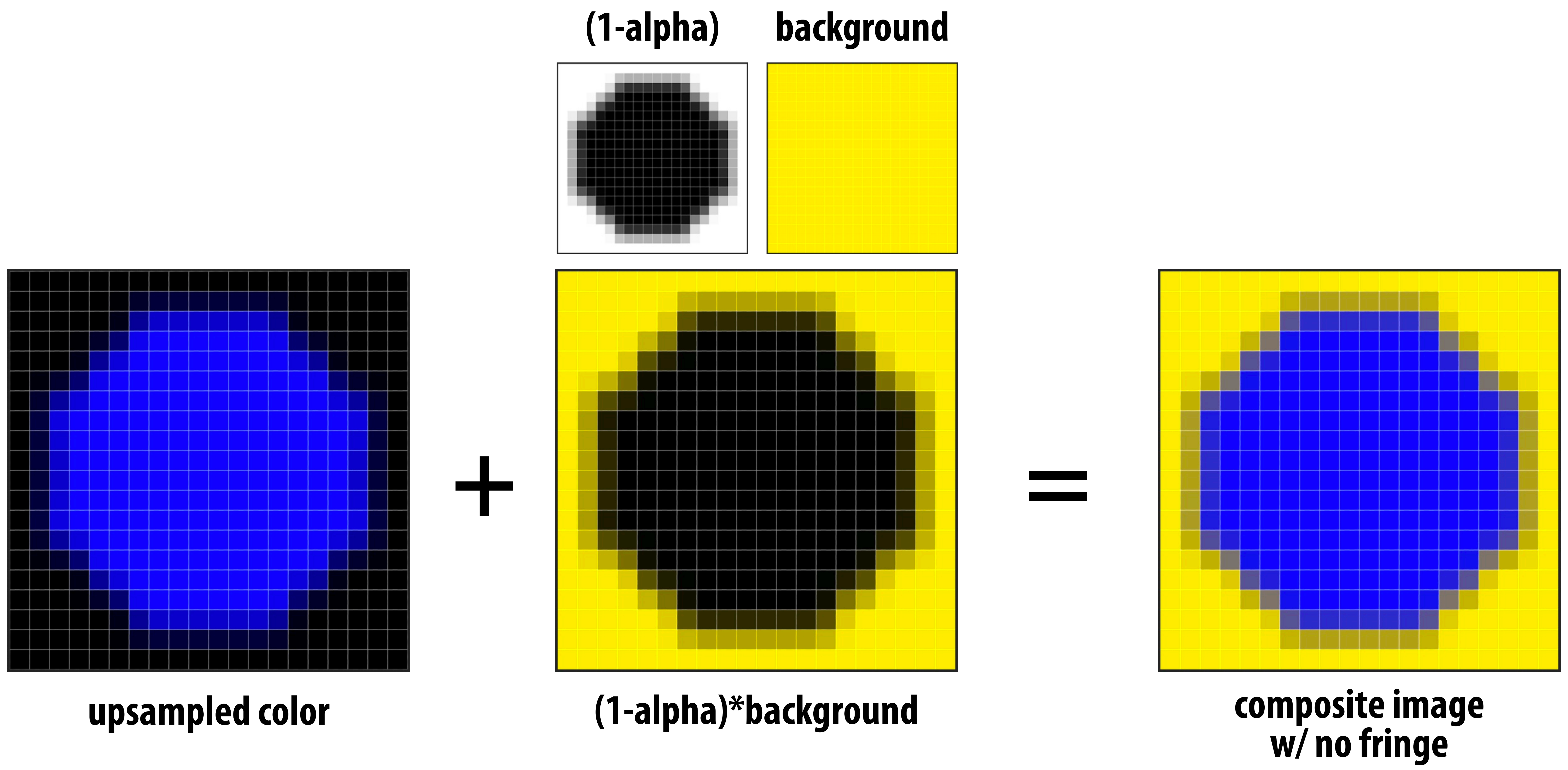

upsampled color


upsampled alpha

Notice black "fringe" that occurs because we're blending, e.g., 50% blue pixels using 50% alpha, rather than, 100% blue pixels with 50% alpha.

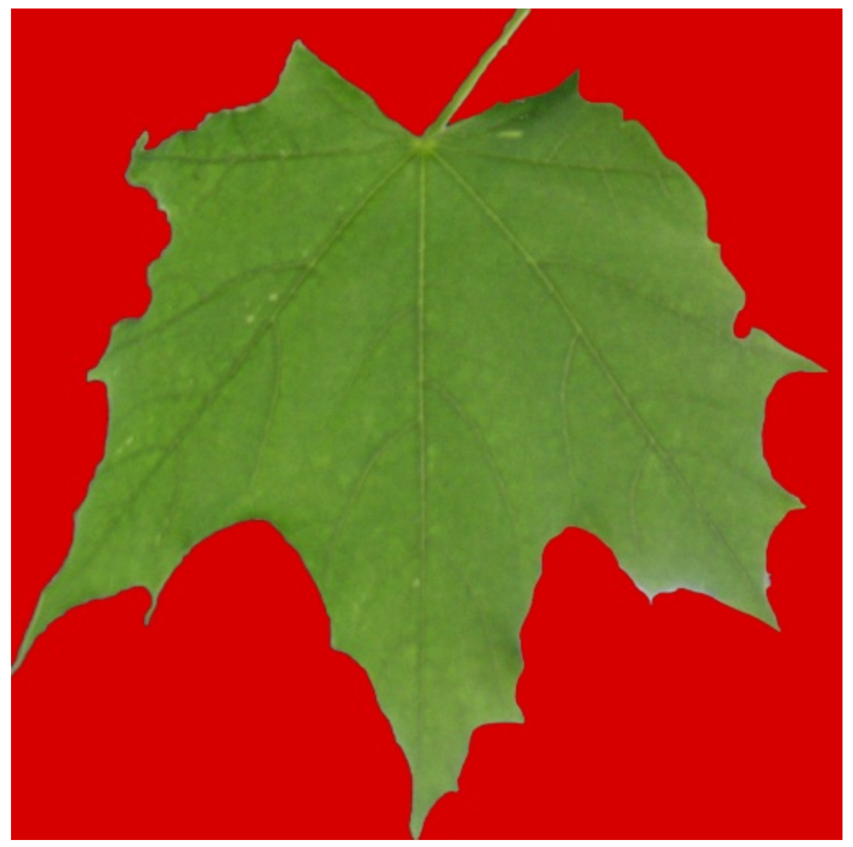
Eliminating fringe w/ premultiplied “over”

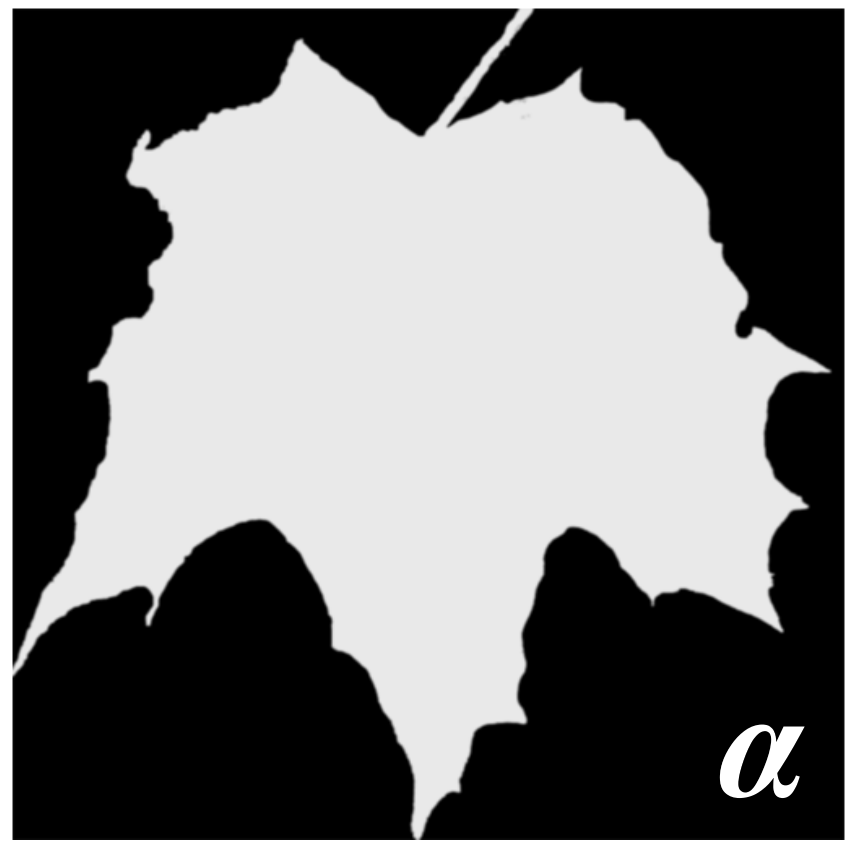
If we instead use the premultiplied “over” operation, we get the correct alpha:



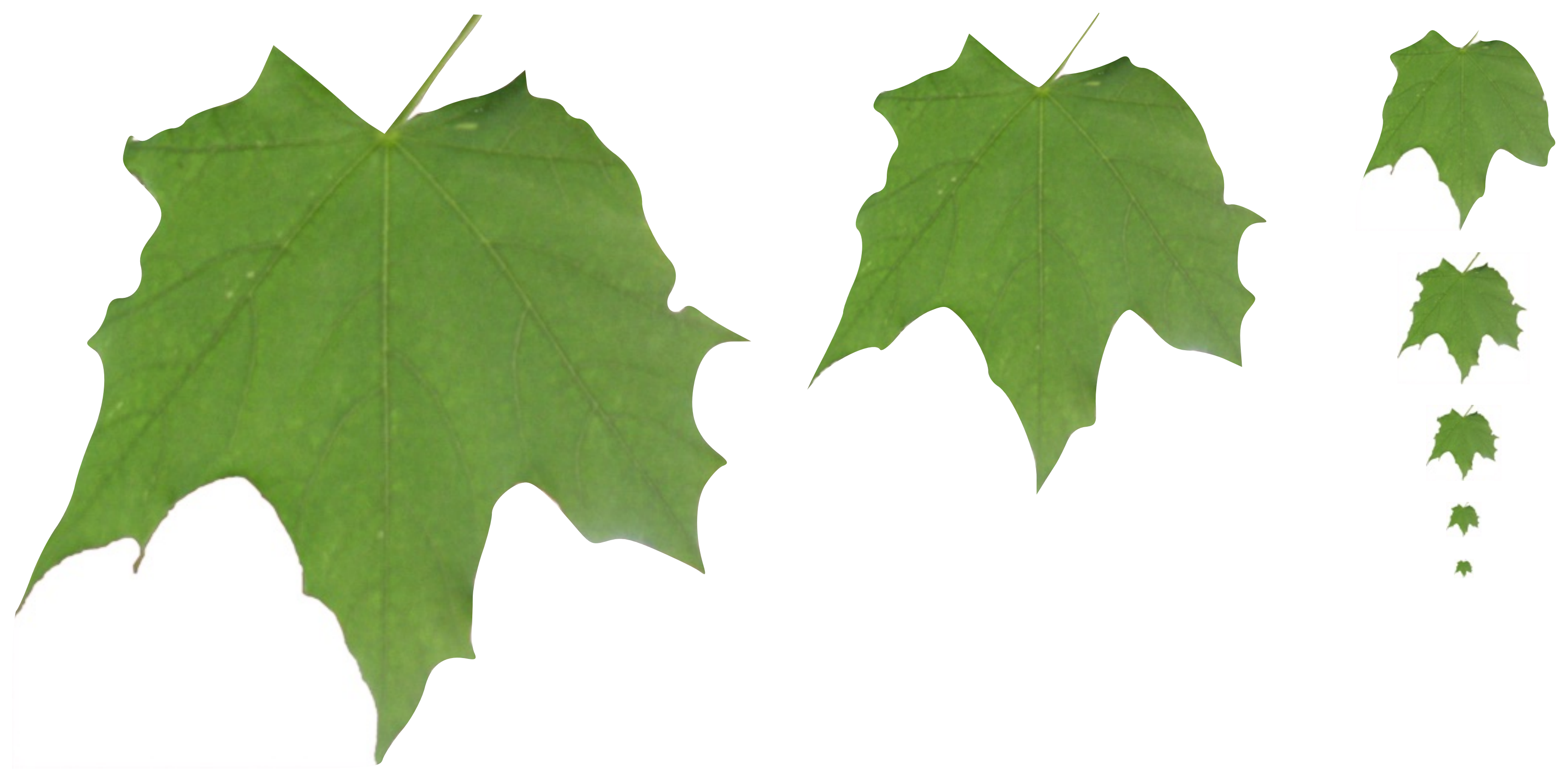
Another problem with non-premultiplied alpha

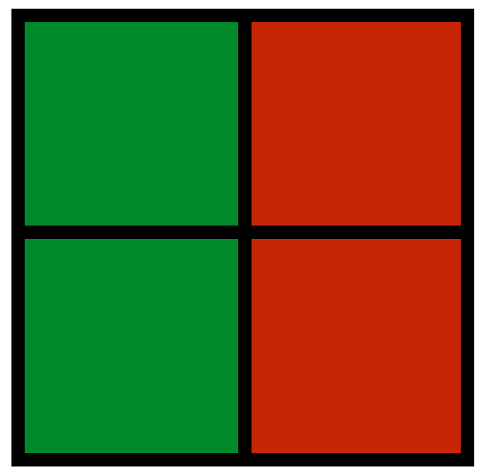
Consider pre-filtering a texture with an alpha matte



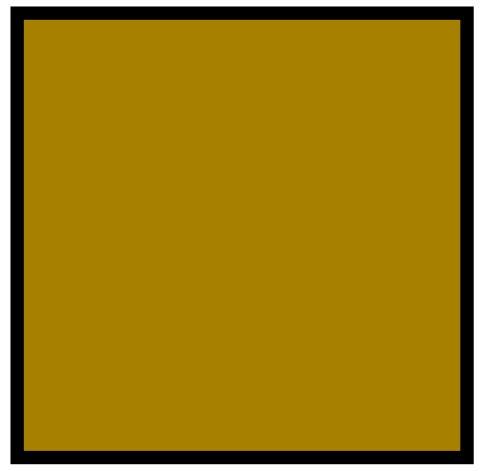


Desired filtered result

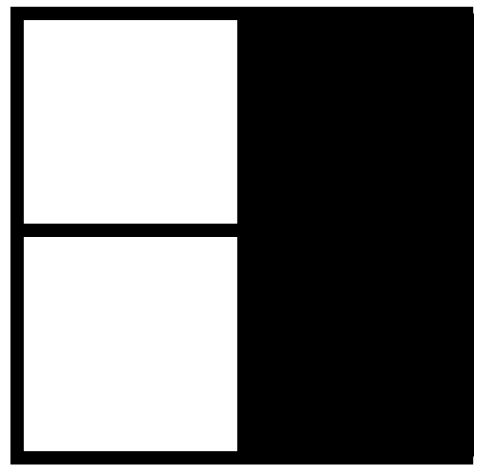




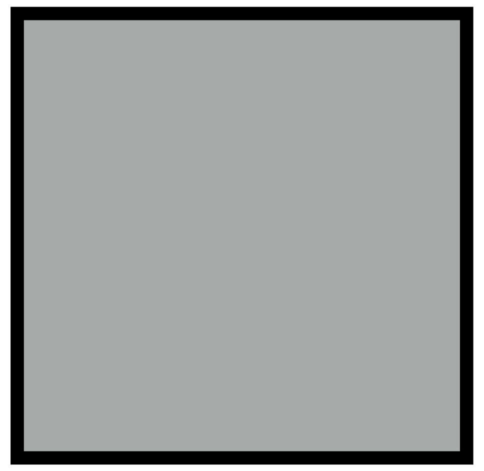
input color



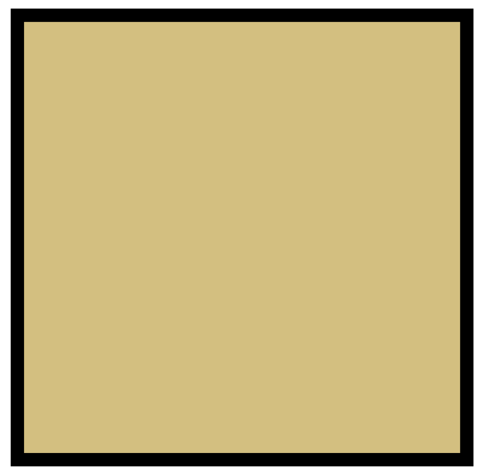
filtered color



input α

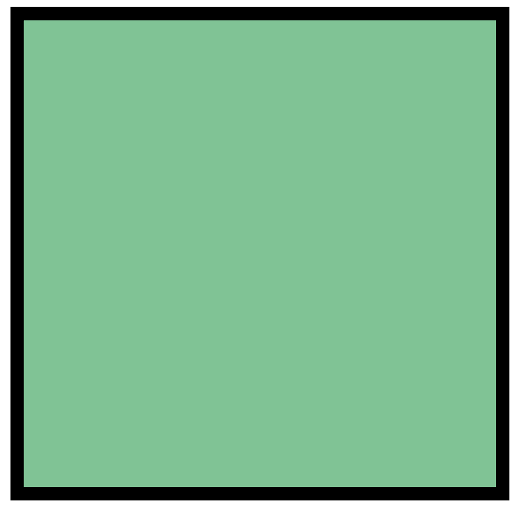


filtered α



filtered result (composited over white)

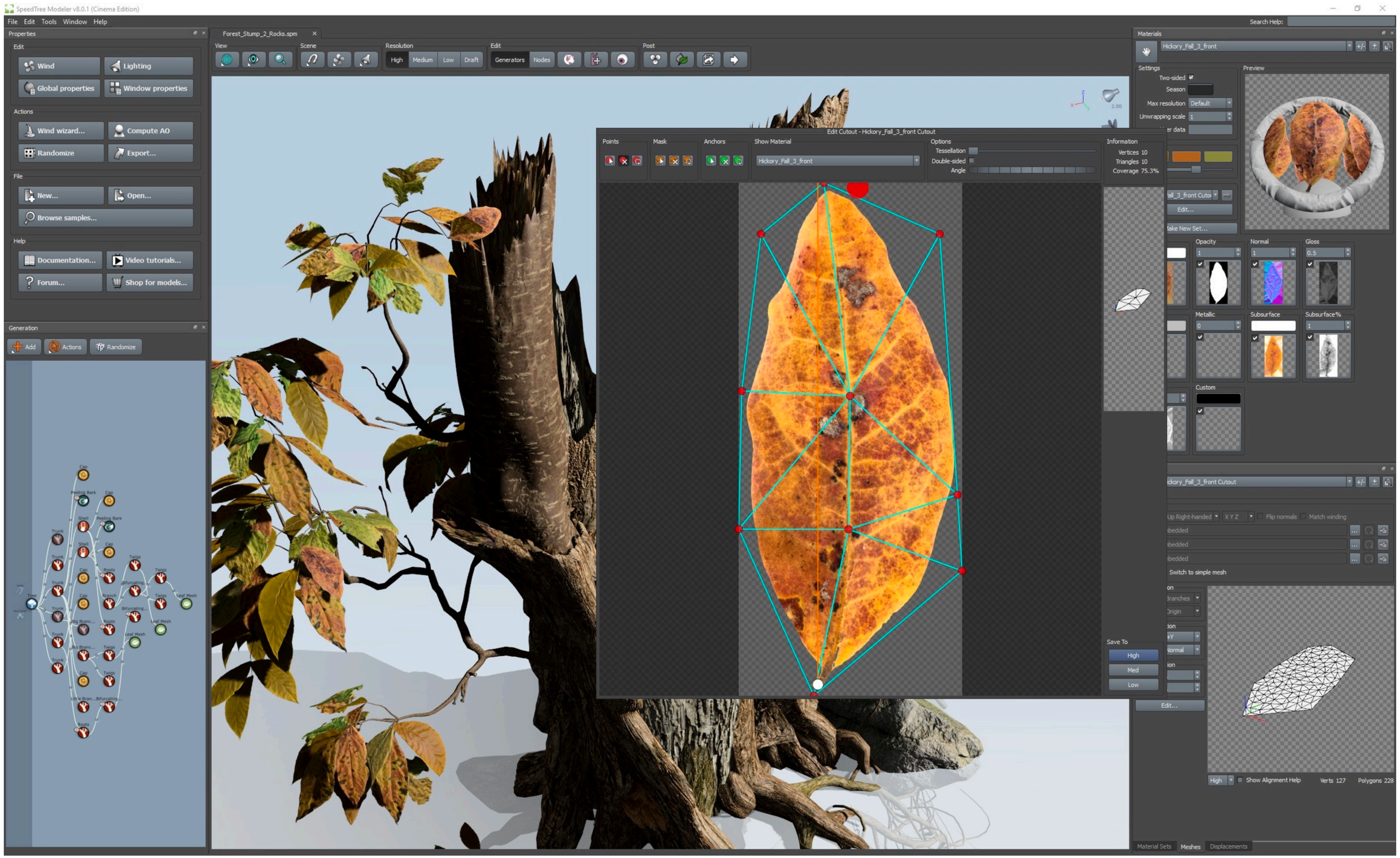
Downsampling non-premultiplied alpha image results in 50% opaque brown (incorrect!)



Result of filtering premultiplied alpha image (correct!)

$$0.25 * ((0, 1, 0, 1) + (0, 1, 0, 1) + (0, 0, 0, 0) + (0, 0, 0, 0)) = (0, 0.5, 0, 0.5)$$

Common use of textures with alpha: foliage



[Image credit: SpeedTree Cinema 8]

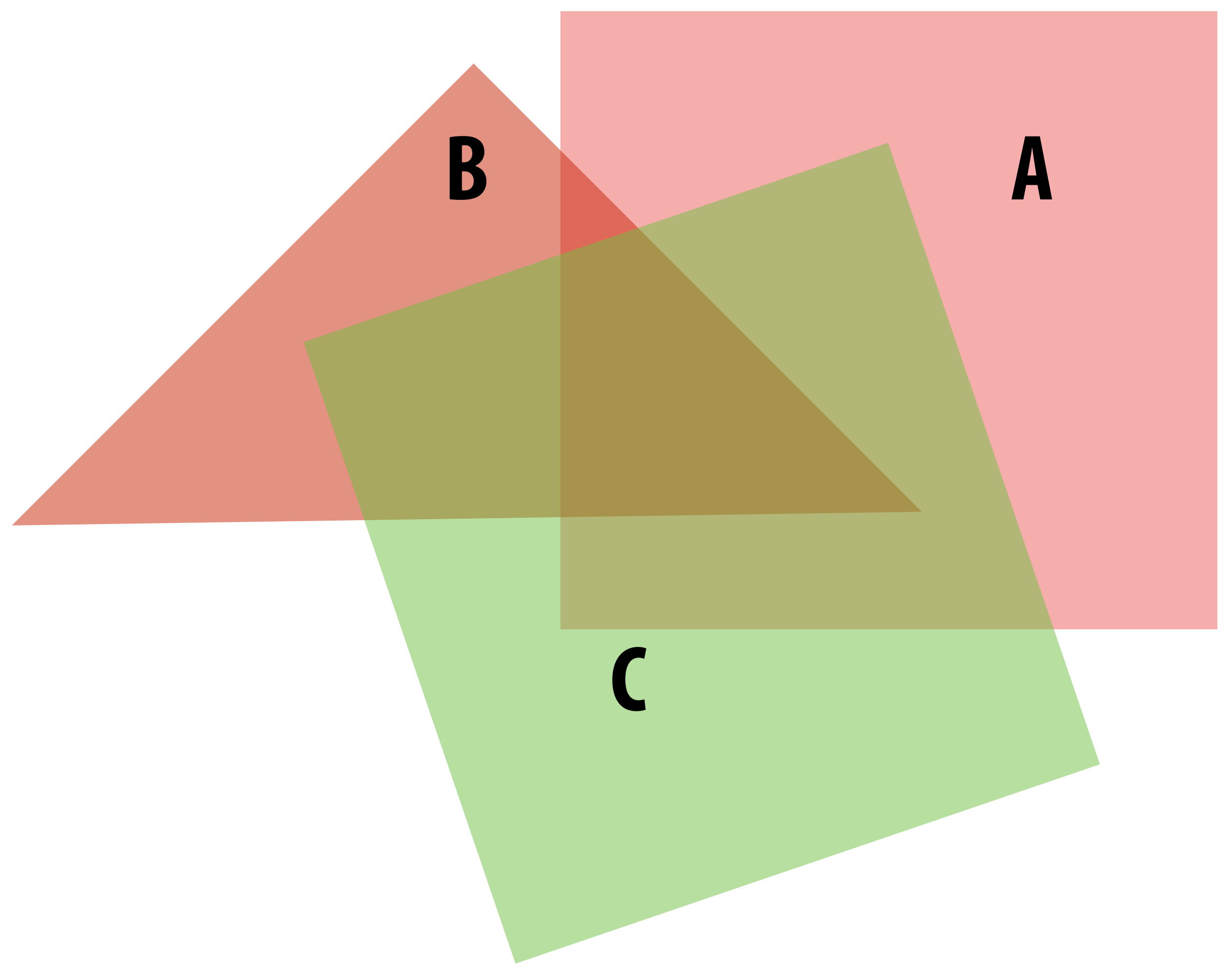
Foliage example

[Image credit: SpeedTree Cinema 8]

Another problem: applying "over" repeatedly

Consider composite image C with opacity α_C over B with opacity α_B over image A with opacity α_A

$$A = \begin{bmatrix} A_r & A_g & A_b \end{bmatrix}^T$$
$$B = \begin{bmatrix} B_r & B_g & B_b \end{bmatrix}^T$$
$$C = \alpha_B B + (1 - \alpha_B)\alpha_A A$$
$$\alpha_C = \alpha_B + (1 - \alpha_B)\alpha_A$$



C over B over A

Consider first step of of compositing 50% red over 50% red:

$$C = \begin{bmatrix} 0.75 & 0 & 0 \end{bmatrix}^T$$
$$\alpha_C = 0.75$$

Wait… this result is the premultiplied color!

So "over" for non-premultiplied alpha takes non-premultiplied colors to premultiplied colors ("over" operation is not closed)

Cannot compose "over" operations on non-premultiplied values:

over(C, over(B, A))

There is a closed form for non-premultiplied alpha:

$$C = \frac{1}{\alpha_C}(\alpha_B B + (1 - \alpha_B)\alpha_A A)$$

Summary: advantages of premultiplied alpha

- **Simple: compositing operation treats all channels (rgb and a) the same**
- **Closed under composition**
- **Better representation for filtering textures with alpha channel**
- **More efficient than non-premultiplied representation: "over" requires fewer math ops**

Color buffer update: semi-transparent surfaces

Assume: color buffer values and tri_color are represented with premultiplied alpha

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

update_color_buffer(tri_z, tri_color, x, y) {
   // Note: no depth check, no depth buffer update
   color[x][y] = over(tri_color, color[x][y]);
}
```

What is the assumption made by this implementation?

Triangles must be rendered in back to front order!

What if triangles are rendered in front to back order?

Modify code: over(color[x][y], tri_color)

Putting it all together *

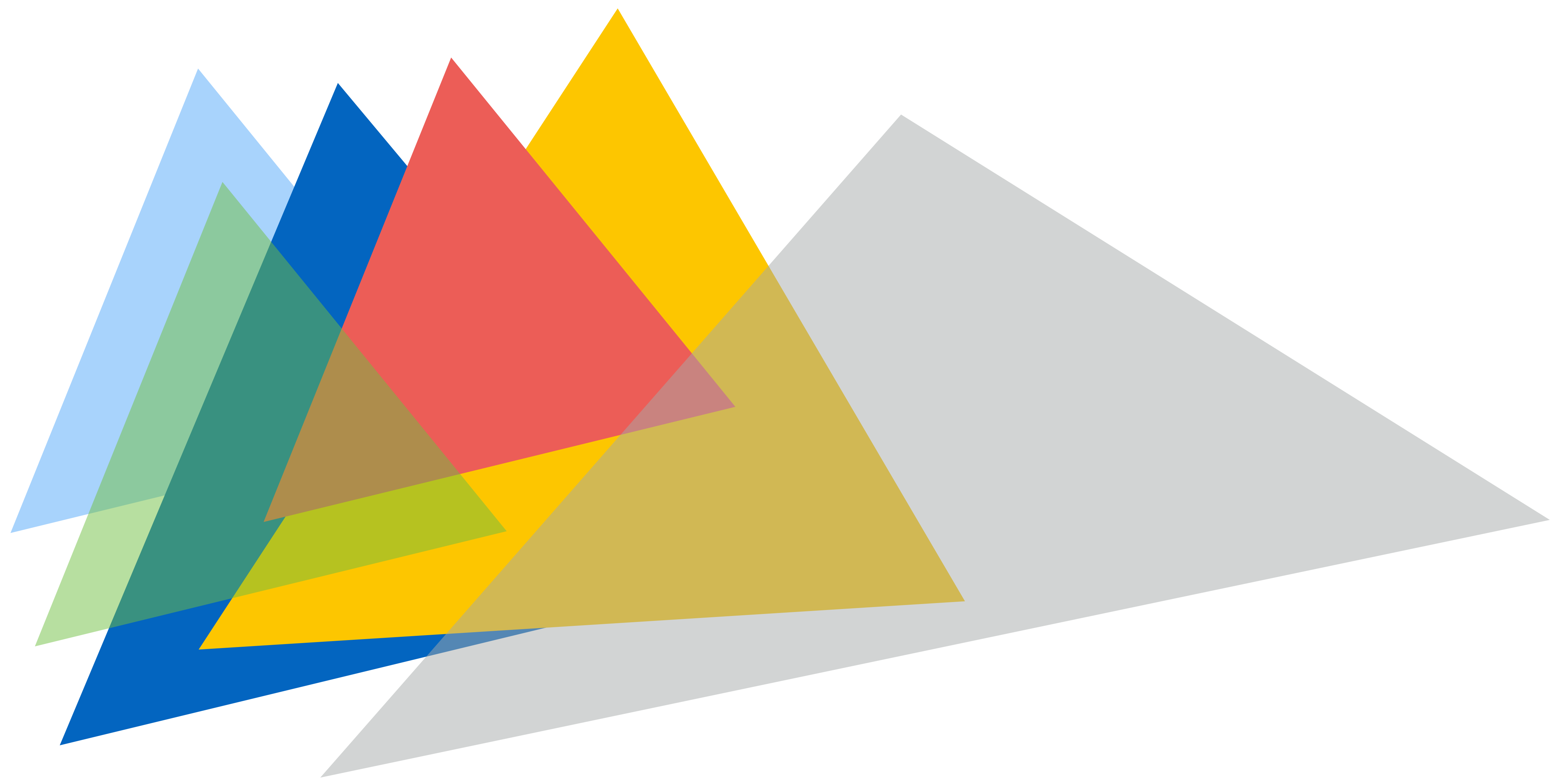
Consider rendering a mixture of opaque and transparent triangles

Step 1: render opaque surfaces using depth-buffered occlusion

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

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

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



* If this seems a little complicated, you will enjoy the simplicity of using ray tracing algorithm for rendering.
More on this later in the course, and in CS348B

Combining opaque and semi-transparent triangles

Assume: color buffer values and tri_color are represented with premultiplied alpha

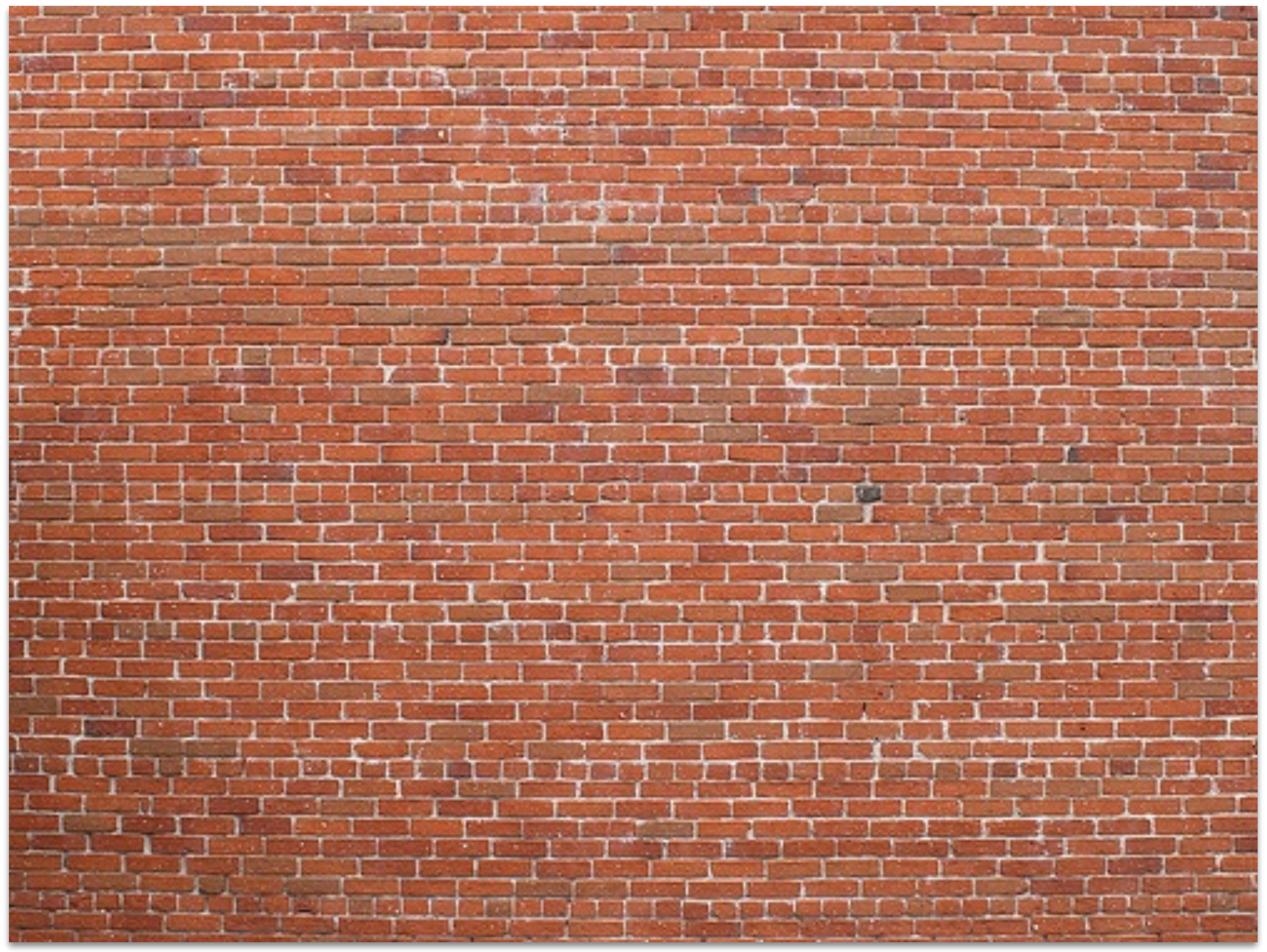
```
// phase 1: render opaque surfaces
update_color_buffer(tri_z, tri_color, x, y) {
   if (pass_depth_test(tri_z, zbuffer[x][y]) {
      color[x][y] = tri_color;
      zbuffer[x][y] = tri_z;
   }
}


// phase 2: render semi-transparent surfaces
update_color_buffer(tri_z, tri_color, x, y) {

   if (pass_depth_test(tri_z, zbuffer[x][y]) {
       // Note: no depth buffer update
       color[x][y] = over(tri_color, color[x][y]);
  }
}
```

End-to-end rasterization pipeline
(“real-time graphics pipeline”)

Command: draw these triangles!

Inputs:

```
list_of_positions = {    list_of_texcoords = {

    v0x, v0y, v0z,           v0u, v0v,
    v1x, v1y, v1z,           v1u, v1v,
    v2x, v2y, v2z,           v2u, v2v,
    v3x, v3y, v3z,           v3u, v3v,
    v4x, v4y, v4z,           v4u, v4v,
    v5x, v5y, v5z   };       v5u, v5v   };
```


Texture map

Object-to-camera-space transform: $\mathbf{T}$

Perspective projection transform $\mathbf{P}$

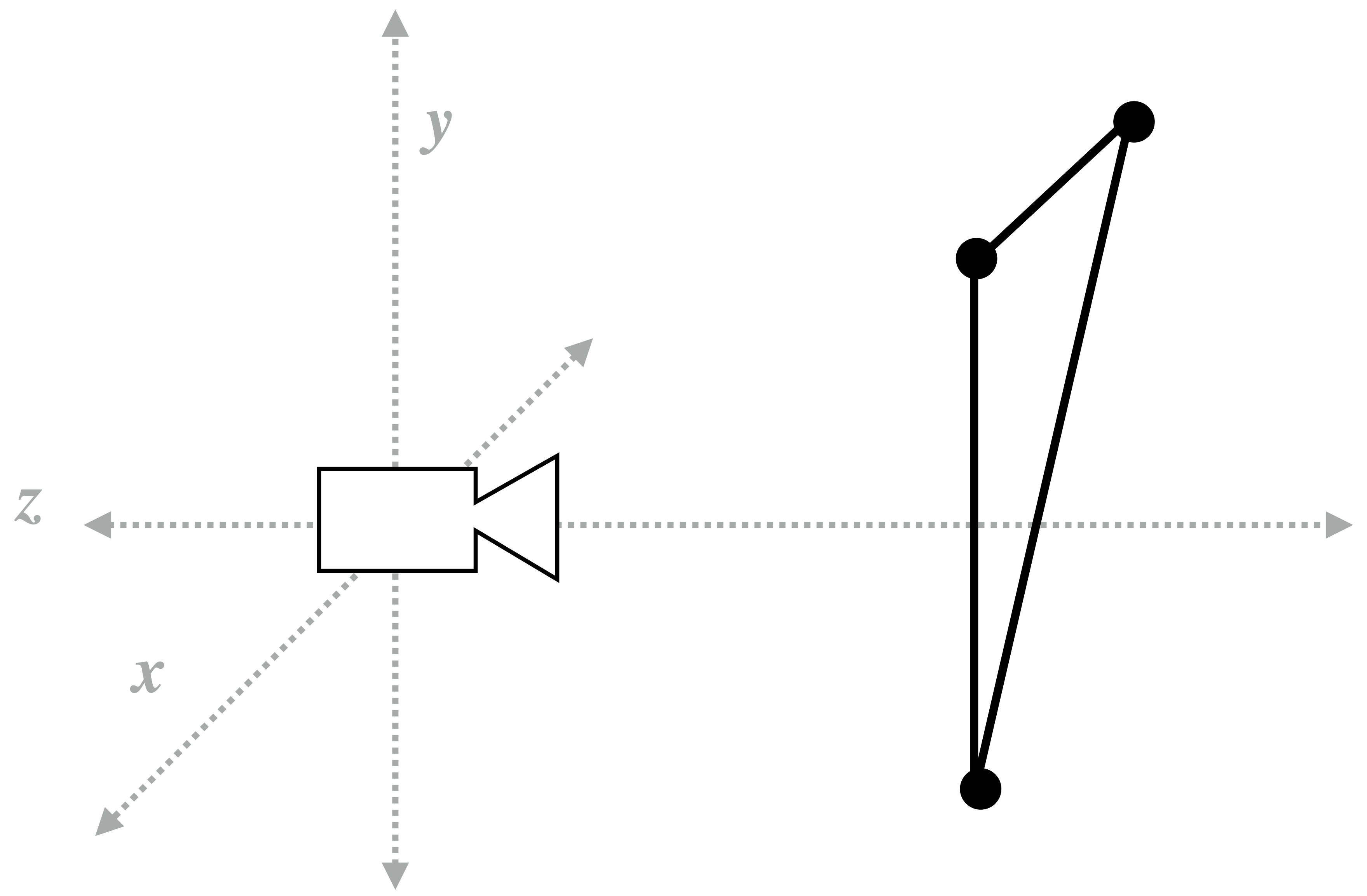
Size of output image (W, H)

Use depth test /update depth buffer: YES!

Step 1:

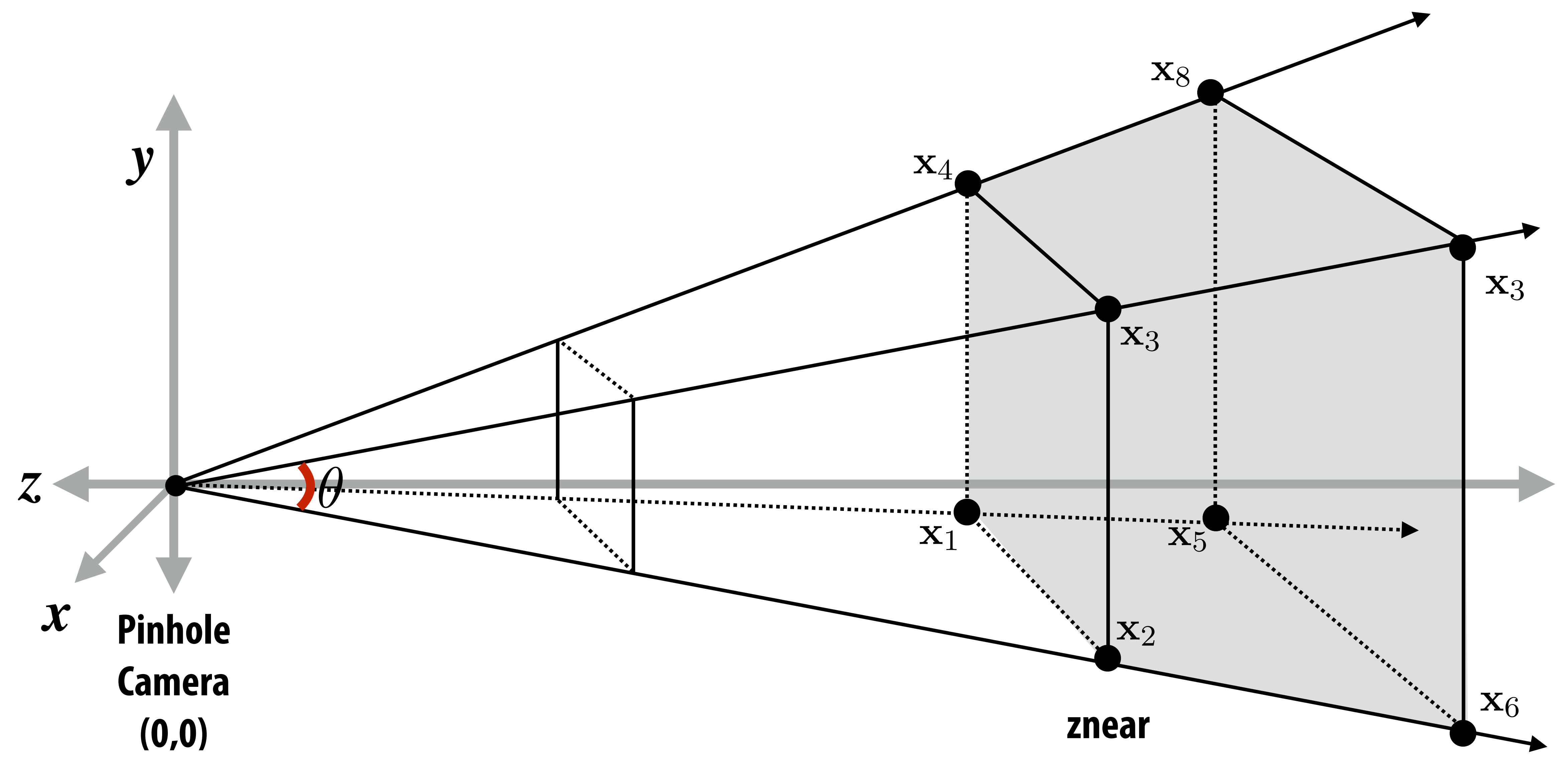
Transform triangle vertices into camera space

(apply modeling and camera transform)

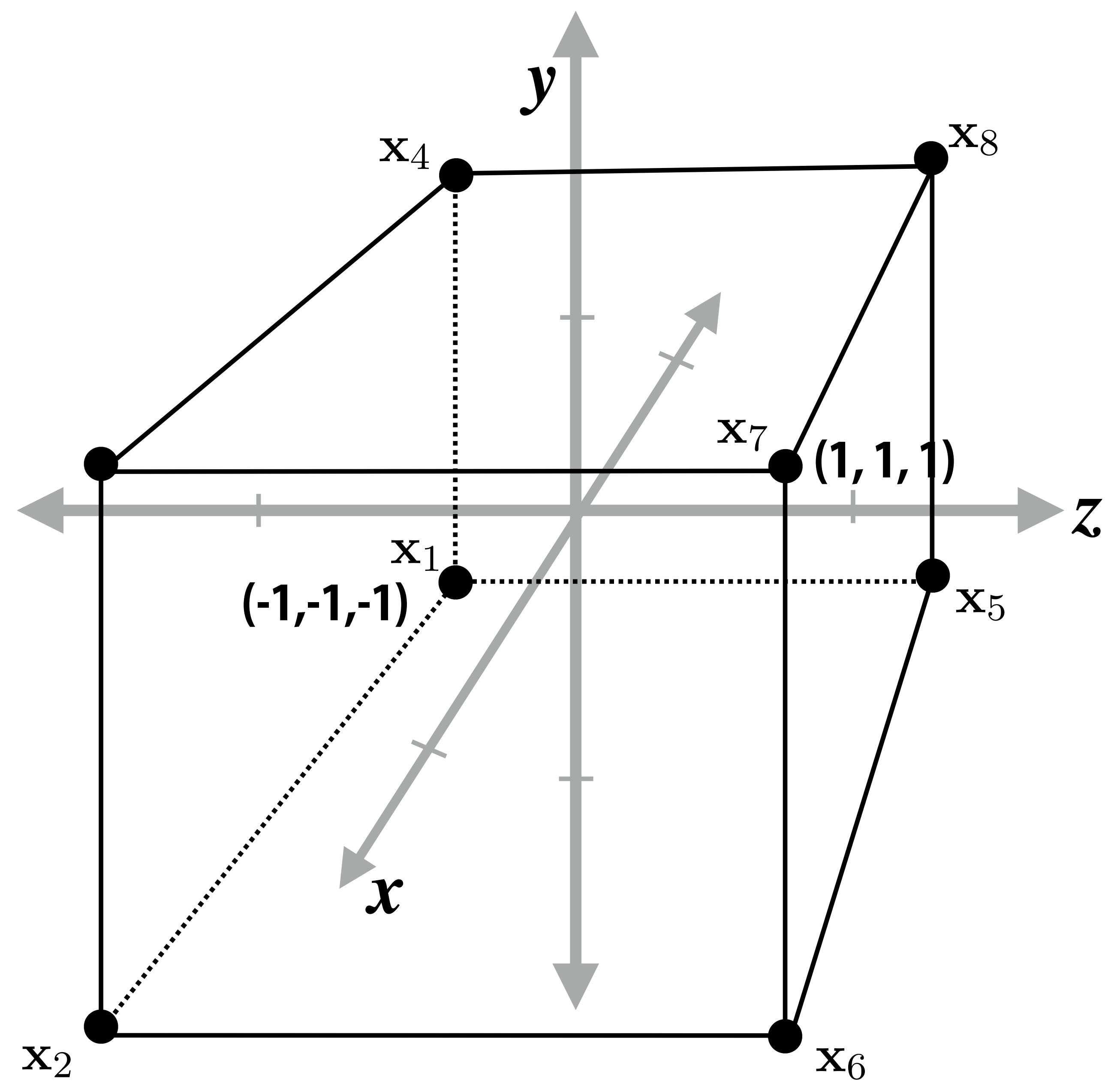


Step 2:

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



Camera-space positions: 3D



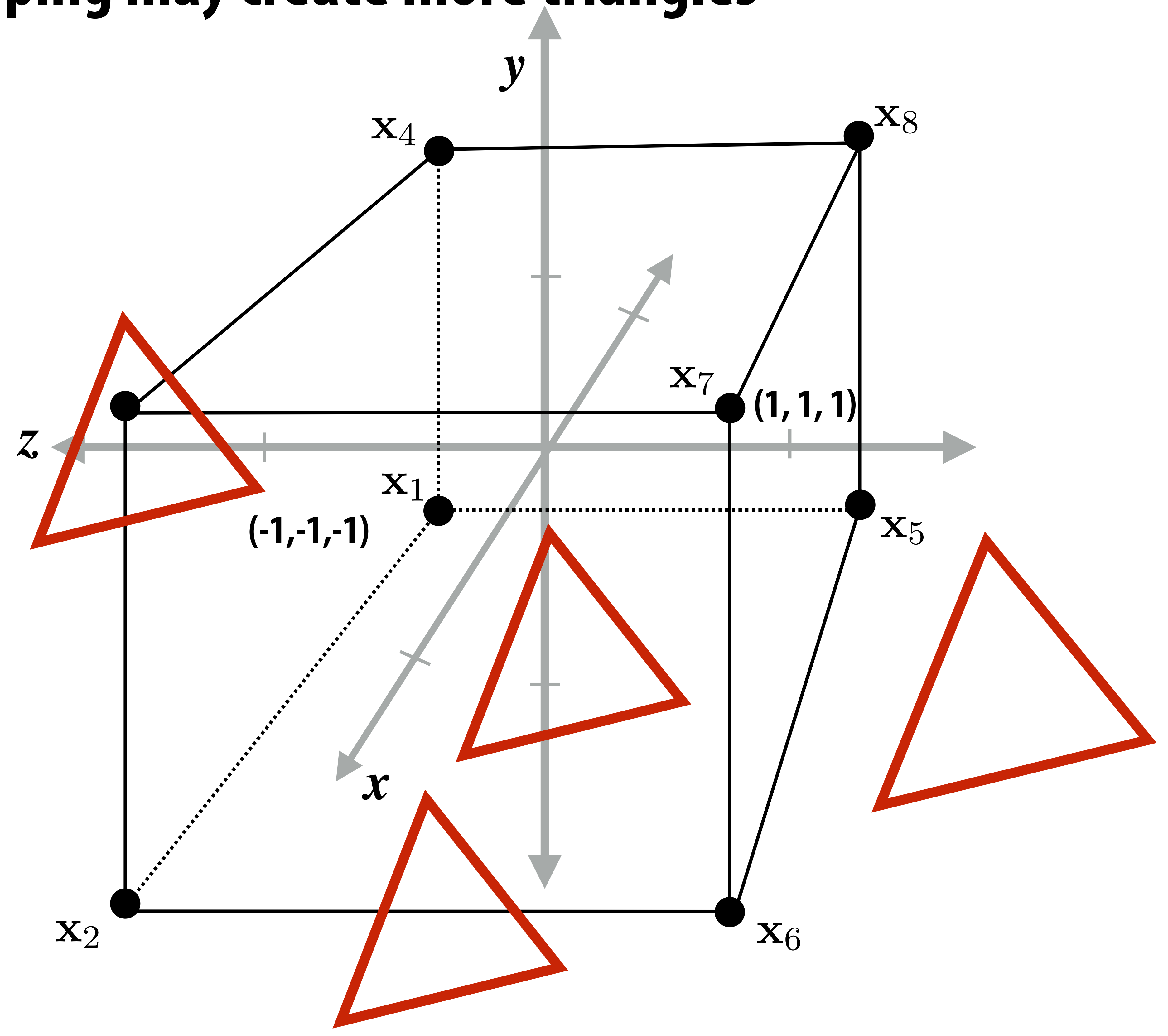
Normalized space positions

Note: I'm illustrating normalized 3D space after the homogeneous divide, it is more accurate to think of this volume in 3D-H space as defined by:

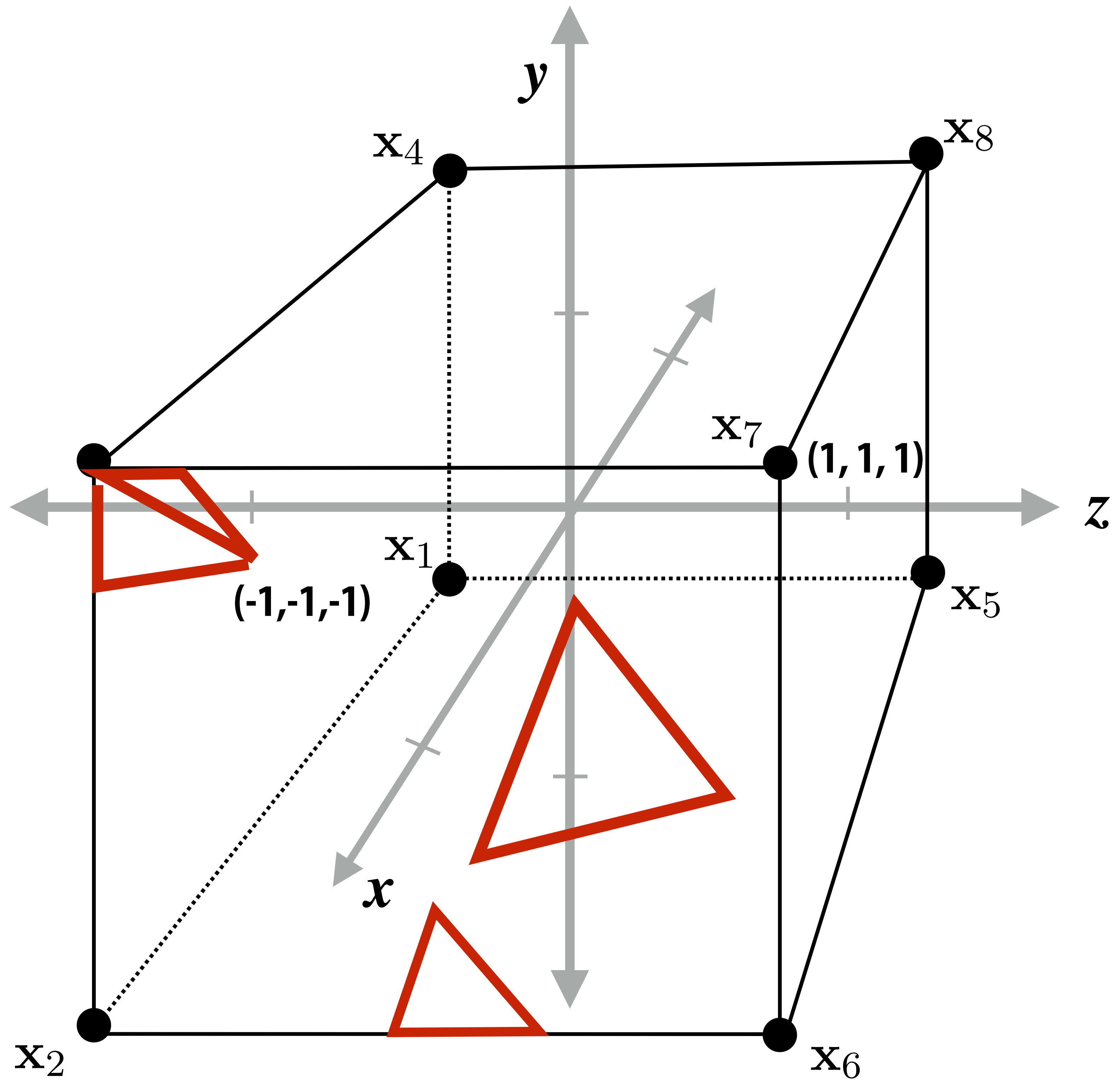
(-w, -w, -w, w) and (w, w, w, w)

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
  - Note: clipping may create more triangles



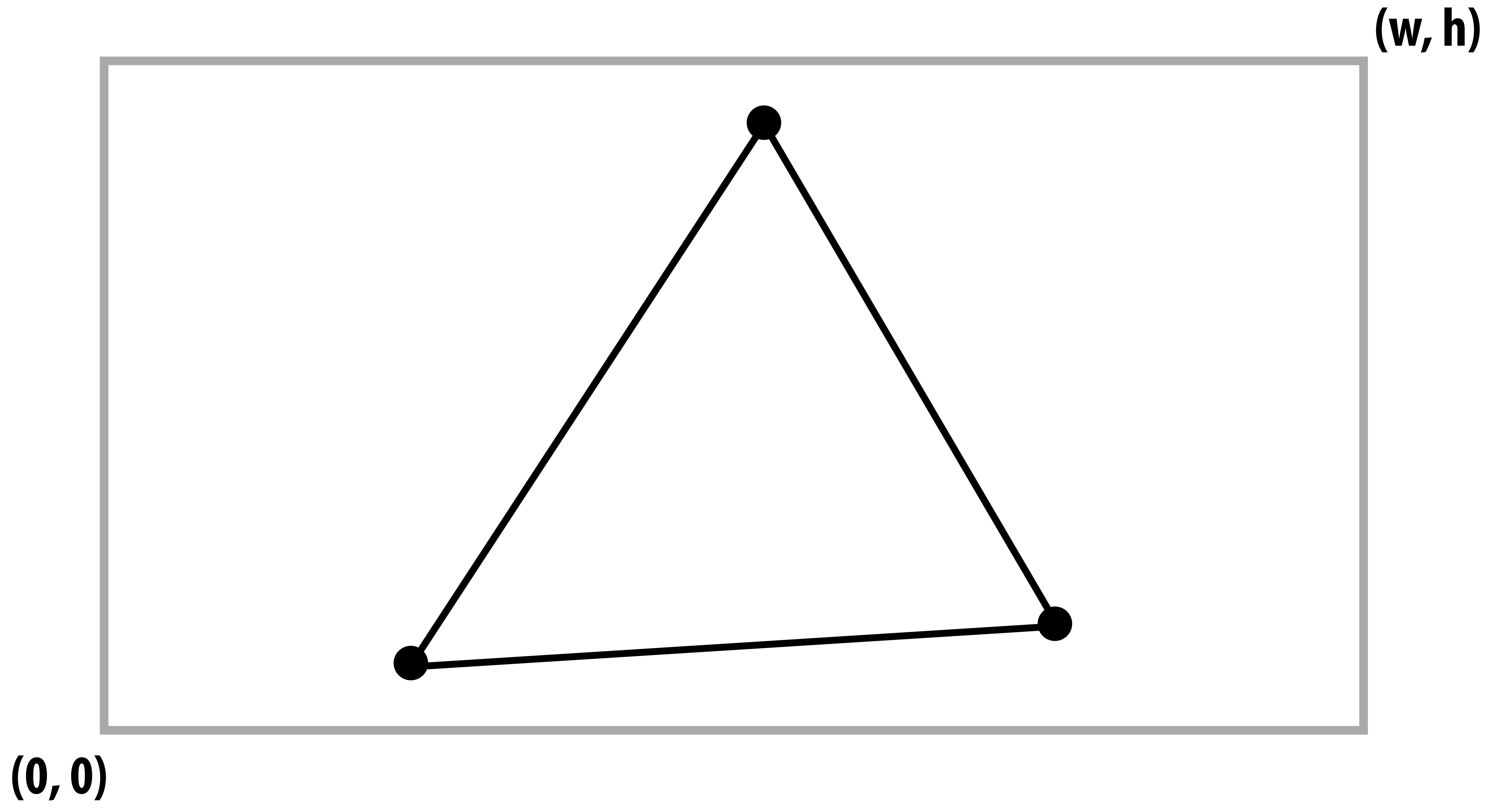
Triangles before clipping



Triangles after clipping

Step 4: transform to screen coordinates

Transform vertex xy positions from normalized coordinates into screen coordinates (based on screen w,h)



Step 5: setup triangle (triangle preprocessing)

Compute triangle edge equations (implicit equations for inside/outside tests)

Compute triangle attribute interpolation equations

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

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

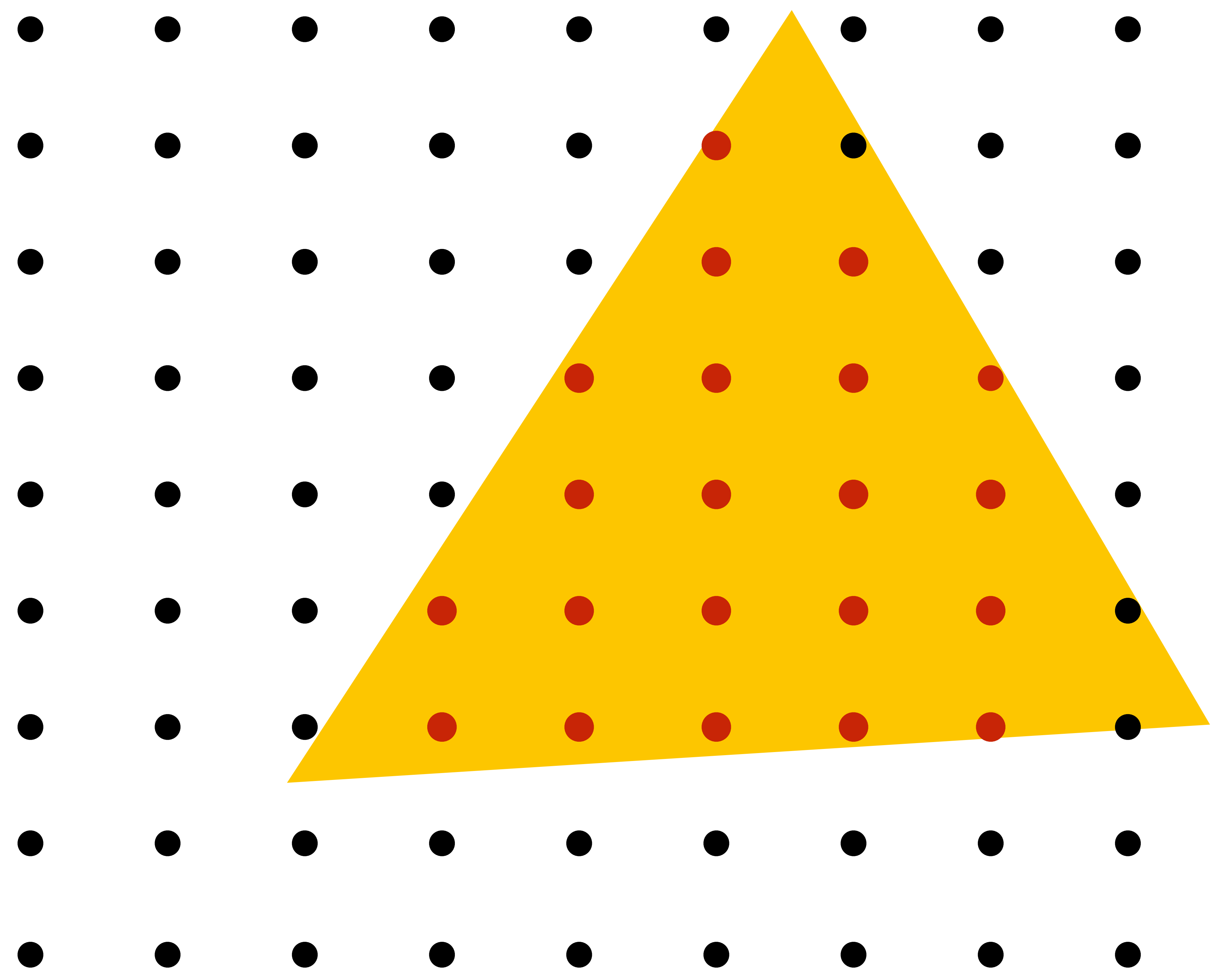
$$\mathbf{E}_{20}(x,y)$$

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

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

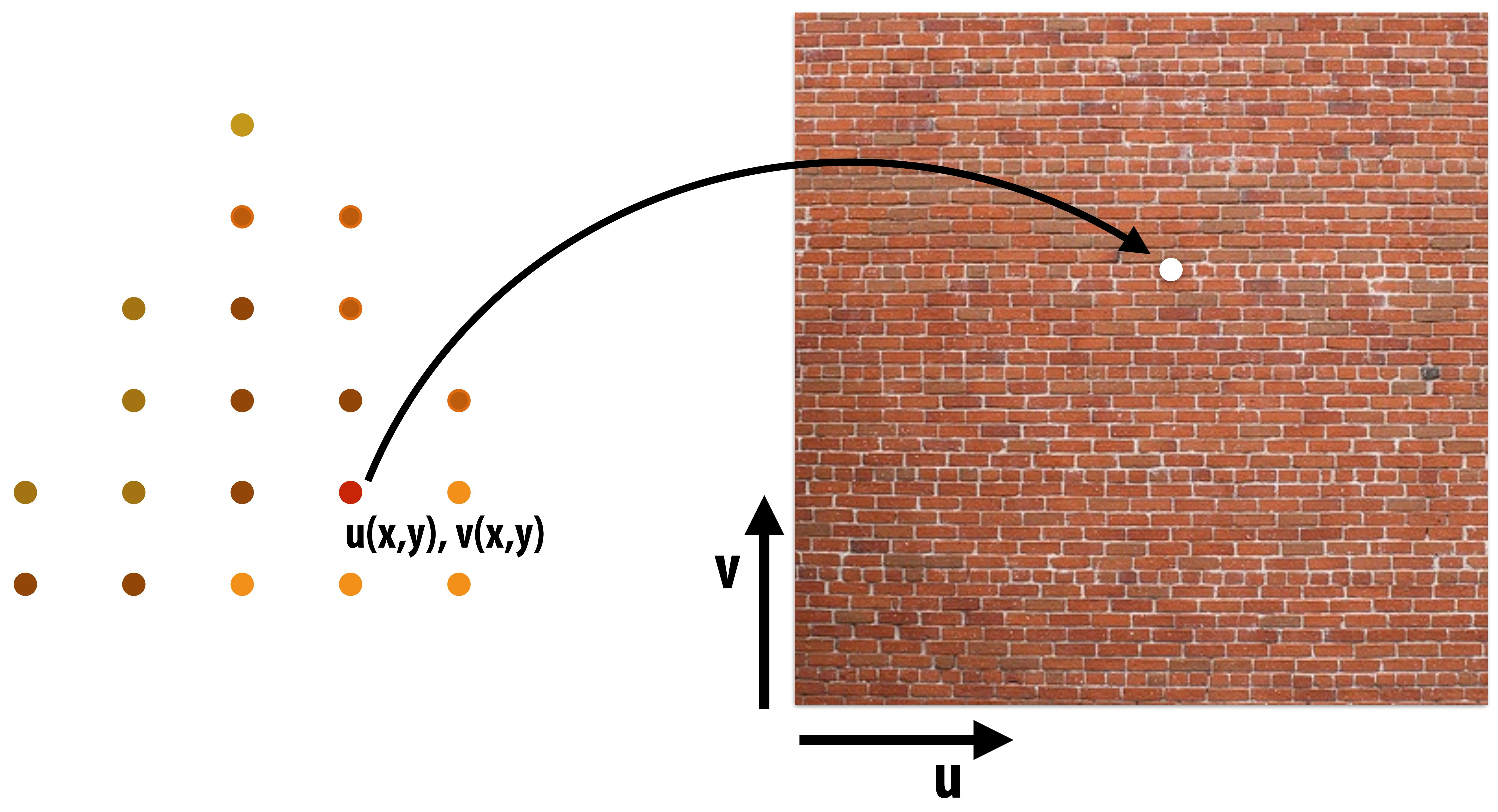
Step 6: sample coverage

Evaluate attributes z, u, v at all covered samples



Step 6: compute triangle color at sample point

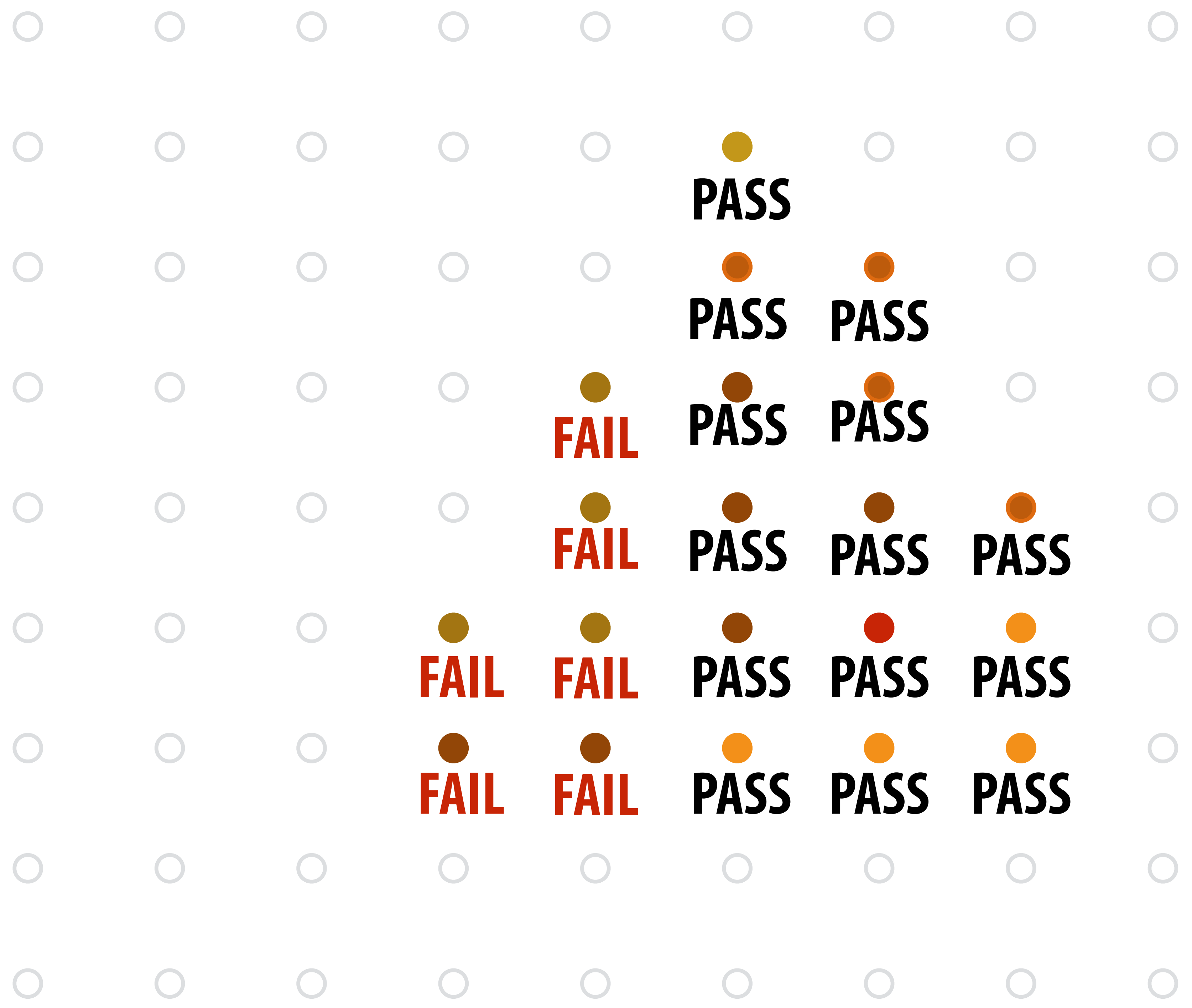
e.g., sample texture map *



* So far, we've only described computing triangle's color at a point by interpolating per-vertex colors, or by sampling a texture map. Later in the course, we'll discuss more advanced algorithms for computing its color based on material properties and scene lighting conditions.

Step 7: perform depth test (if enabled)

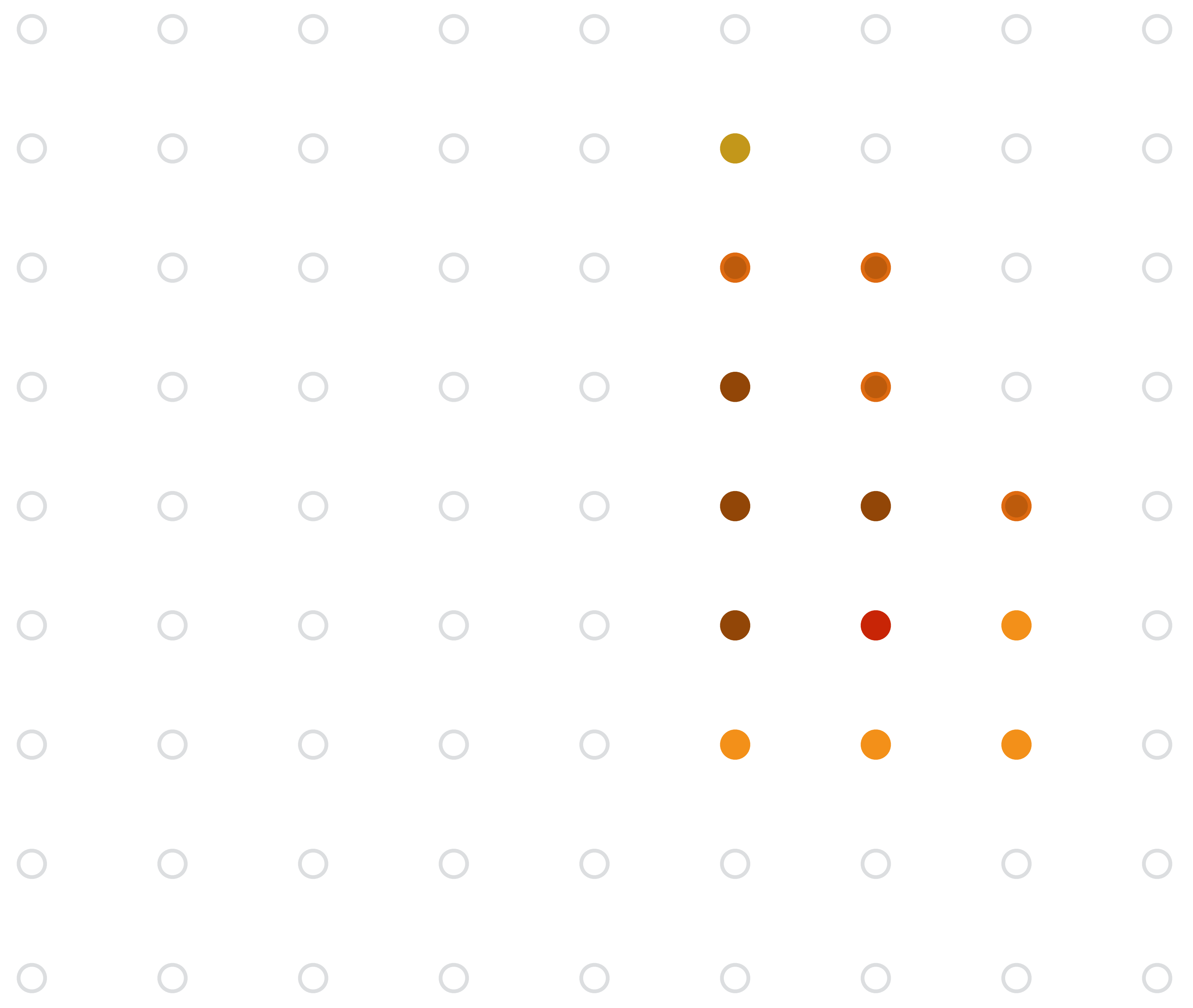
Also update depth value at covered samples (if necessary)



Review: occlusion using the depth buffer (opaque surfaces)

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


depth_test(tri_d, tri_color, x, y) {

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

    // if triangle is closest object seen so far at this
    // sample point. Update depth and color buffers.

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

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



Step 9:

- Repeat steps 1-8 for all triangles in the scene!

One reminder about transparent surfaces

Color buffer update: semi-transparent surfaces

Assume: color buffer values and tri_color are represented with premultiplied alpha

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

update_color_buffer(tri_z, tri_color, x, y) {
   // Note: no depth check, no depth buffer update
   color[x][y] = over(tri_color, color[x][y]);
}
```

What is the assumption made by this implementation?

Triangles must be rendered in back to front order!

What if triangles are rendered in front to back order?

Modify code: over(color[x][y], tri_color)

Putting it all together *

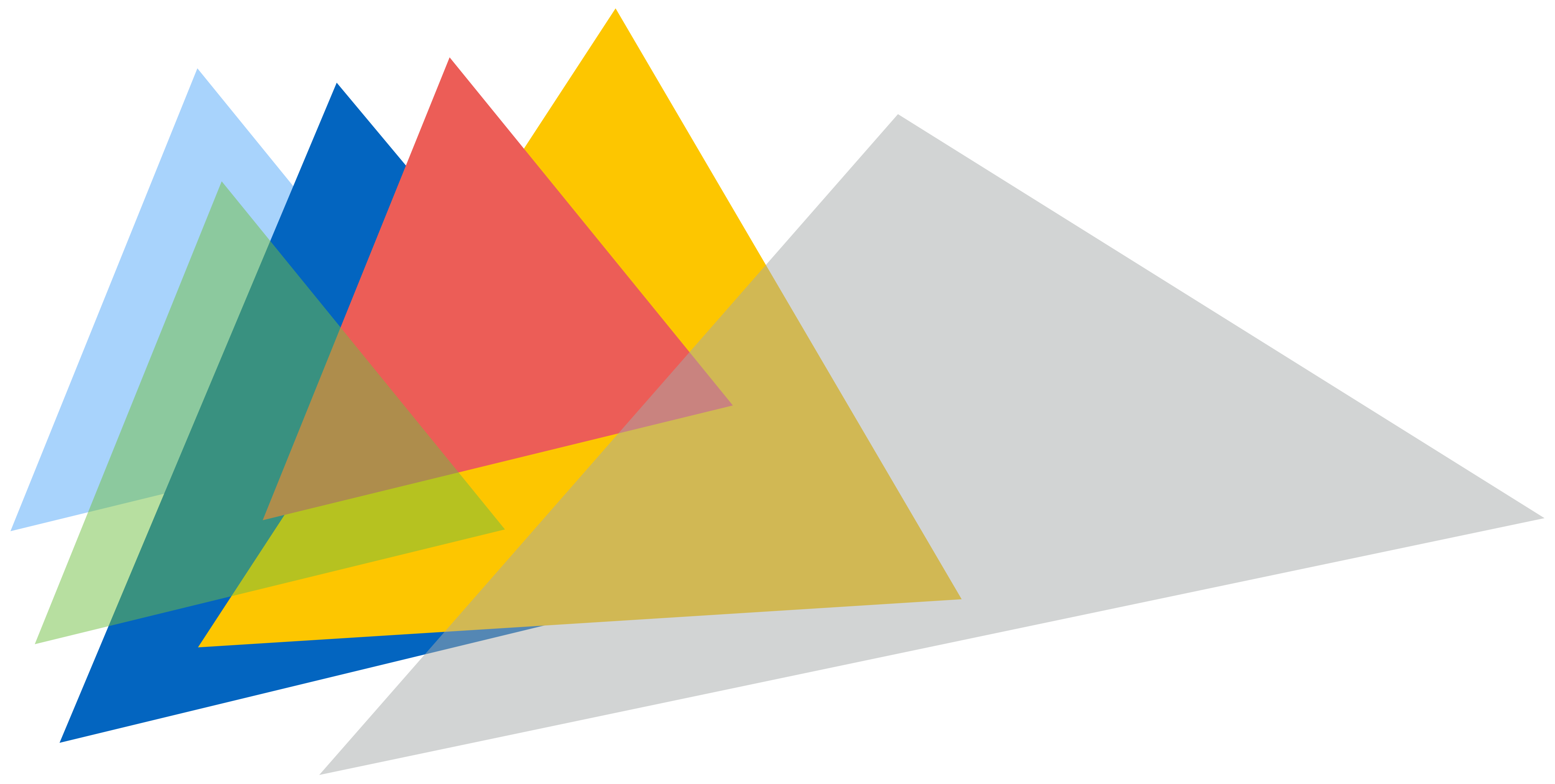
Consider rendering a mixture of opaque and transparent triangles

Step 1: render opaque surfaces 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



* If this seems complicated, you will enjoy the simplicity of using ray tracing algorithm for rendering. More on this in a few weeks.

Combining opaque and semi-transparent triangles

Assume: color buffer values and tri_color are represented with premultiplied alpha

```
// phase 1: render opaque surfaces
update_color_buffer(tri_z, tri_color, x, y) {
   if (pass_depth_test(tri_z, zbuffer[x][y]) {
      color[x][y] = tri_color;
      zbuffer[x][y] = tri_z;
   }
}



// phase 2: render semi-transparent surfaces
update_color_buffer(tri_z, tri_color, x, y) {

   if (pass_depth_test(tri_z, zbuffer[x][y]) {
       // Note: no depth buffer update
       color[x][y] = over(tri_color, color[x][y]);
  }
}
```

Real time graphics APIs

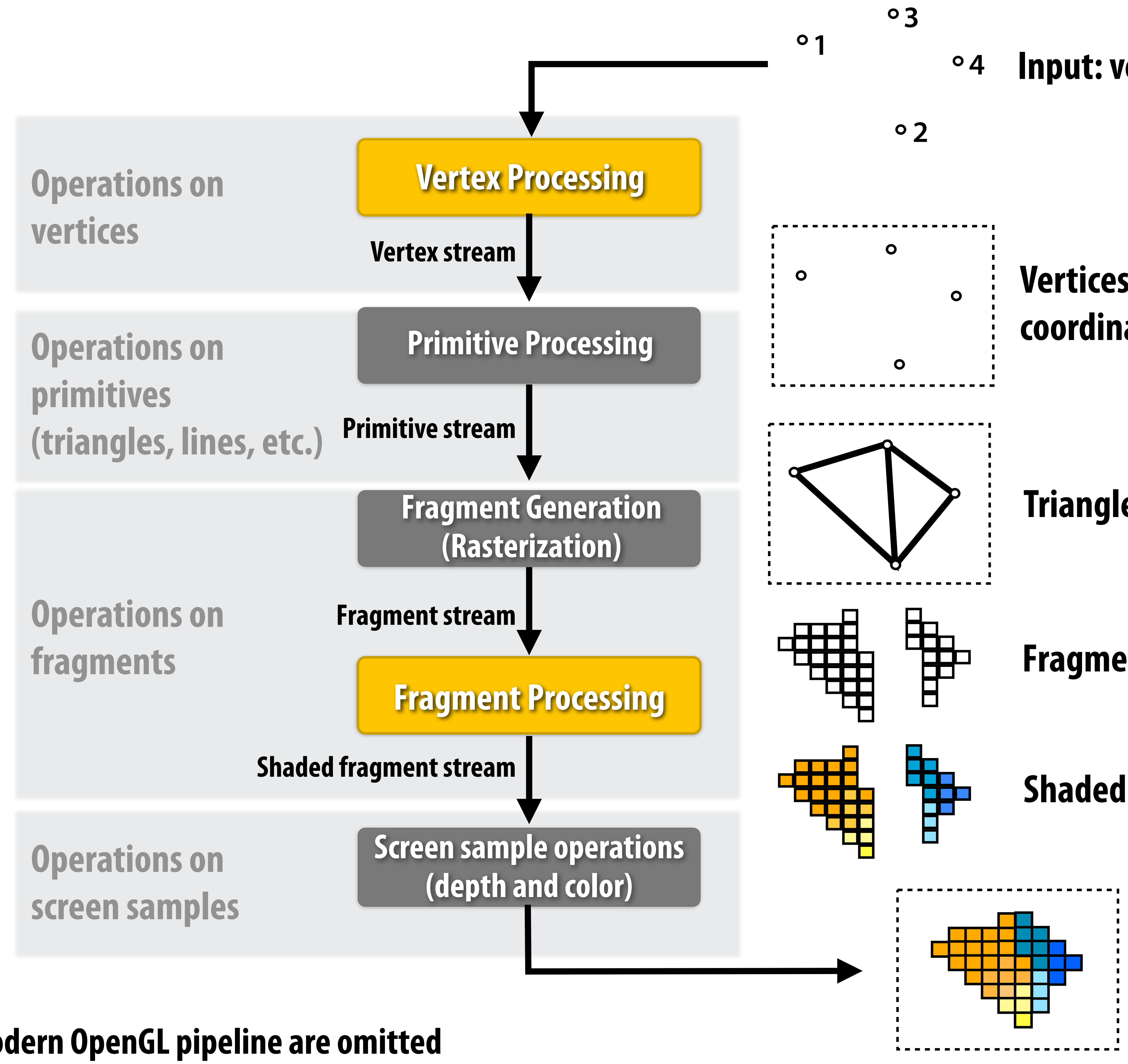
- OpenGL
- Microsoft Direct3D
- Apple Metal

- You now know a lot about the algorithms implemented underneath these APIs: drawing 3D triangles (key transformations and rasterization), texture mapping, anti-aliasing via supersampling, etc.

- Internet is full of useful tutorials on how to program using these APIs

OpenGL/Direct3D graphics pipeline *

Structures rendering computation as a series of operations on vertices, primitives, fragments, and screen samples



Input: vertices in 3D space

Vertices in positioned in normalized coordinate space

Triangles positioned on screen

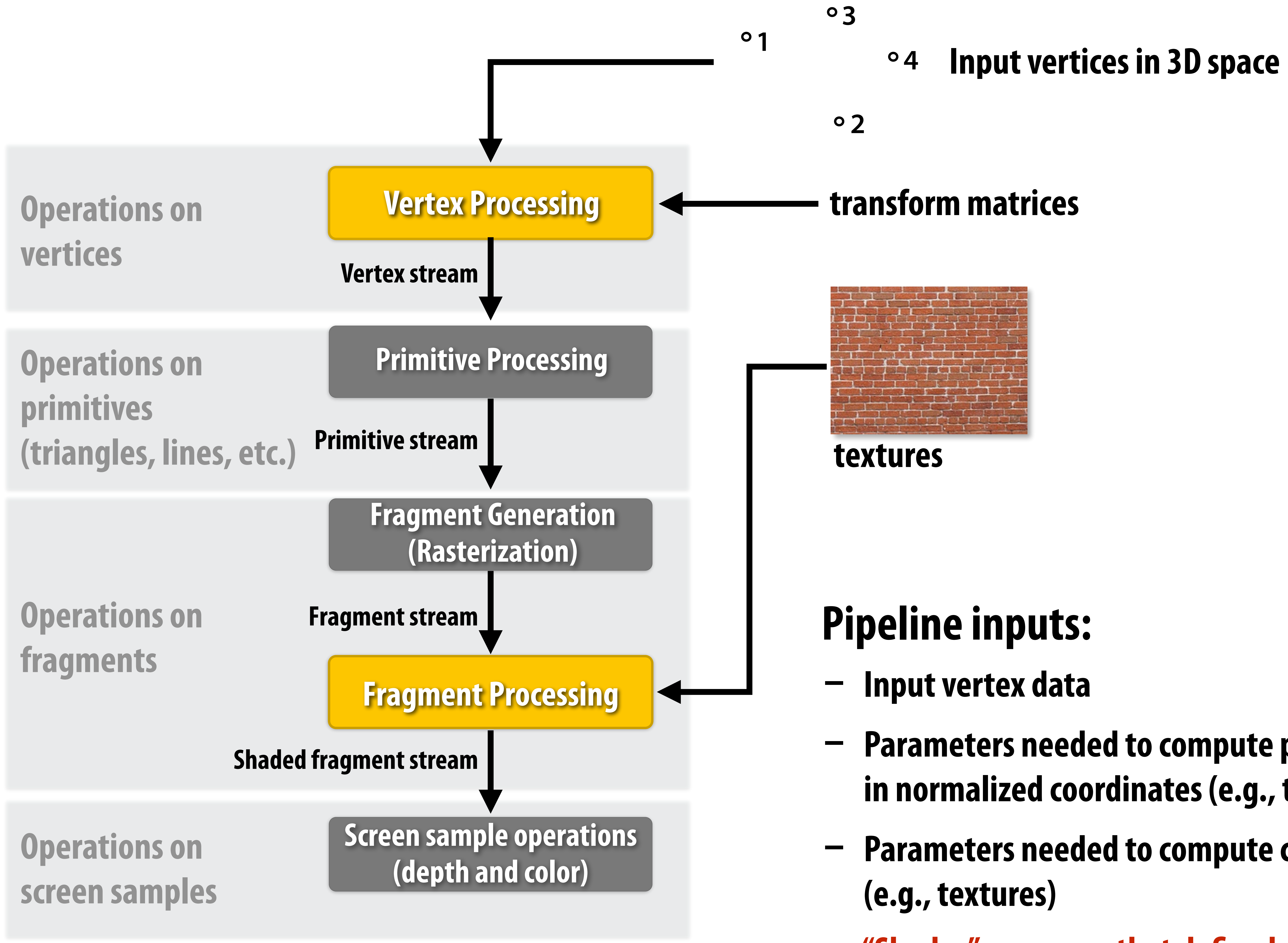
Fragments (one fragment per covered sample)

Shaded fragments

Output: image (pixels)

* Several stages of the modern OpenGL pipeline are omitted

OpenGL/Direct3D graphics pipeline *

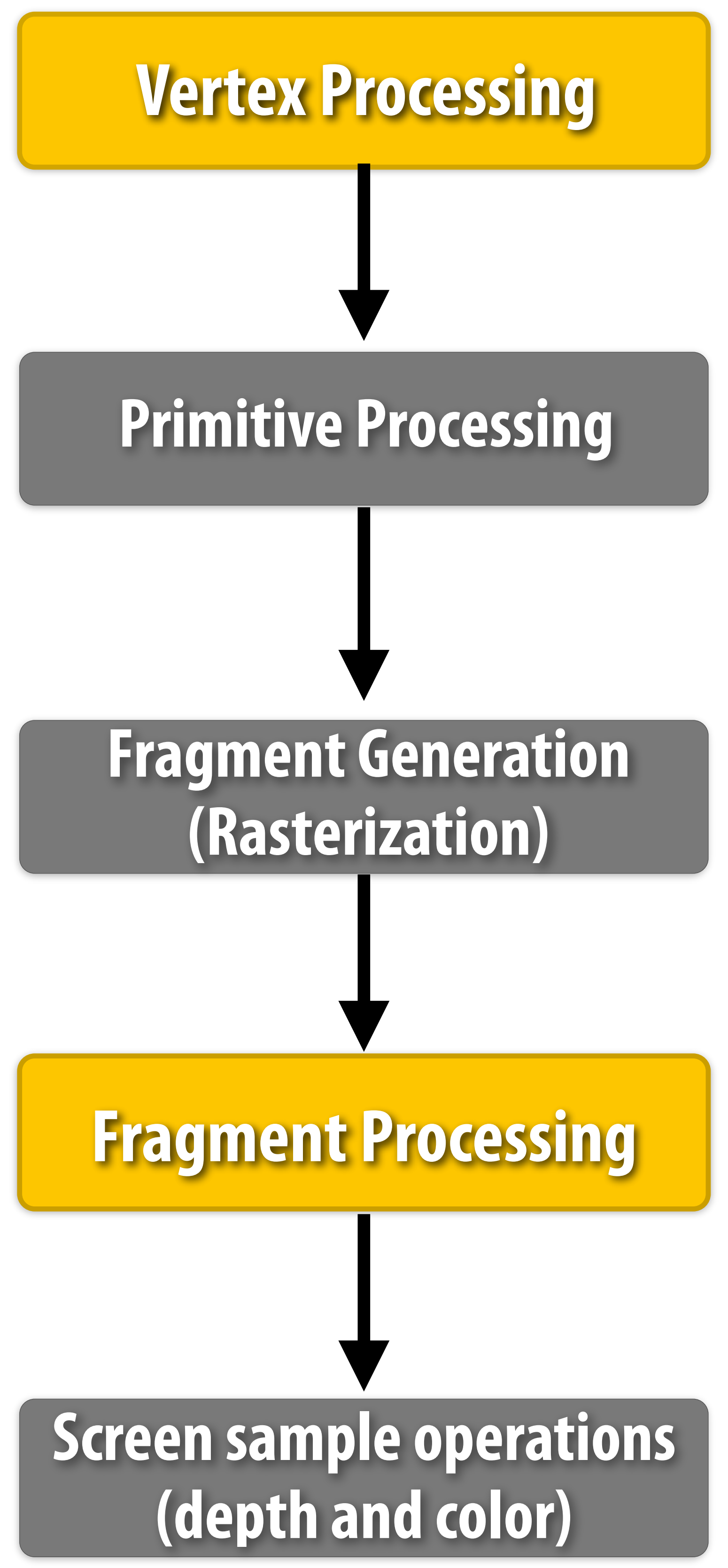


Pipeline inputs:

- Input vertex data
- Parameters needed to compute position on vertices in normalized coordinates (e.g., transform matrices)
- Parameters needed to compute color of fragments (e.g., textures)
- "Shader" programs that define behavior of vertex and fragment stages

* Several stages of the modern OpenGL pipeline are omitted

OpenGL/Direct3D graphics pipeline *


Vertex Processing
Primitive Processing
Fragment Generation (Rasterization)
Fragment Processing
Screen sample operations (depth and color)

* Several stages of the modern OpenGL pipeline are omitted

Shader programs

Define behavior of vertex processing and fragment processing stages

Describe operation on a single vertex (or single fragment)

Example GLSL fragment shader program

```
uniform sampler2D myTexture;
uniform vec3 lightDir;
varying vec2 uv;
varying vec3 norm;

void diffuseShader()
{
  vec3 kd;
  kd = texture2d(myTexture, uv);
  kd *= clamp(dot(-lightDir, norm), 0.0, 1.0);
  gl_FragColor = vec4(kd, 1.0);
}
```

Program parameters

Per-fragment attributes (interpolated by rasterizer)

Sample surface albedo (reflectance color) from texture

Shader outputs surface color

Modulate surface albedo by incident irradiance (incoming light)

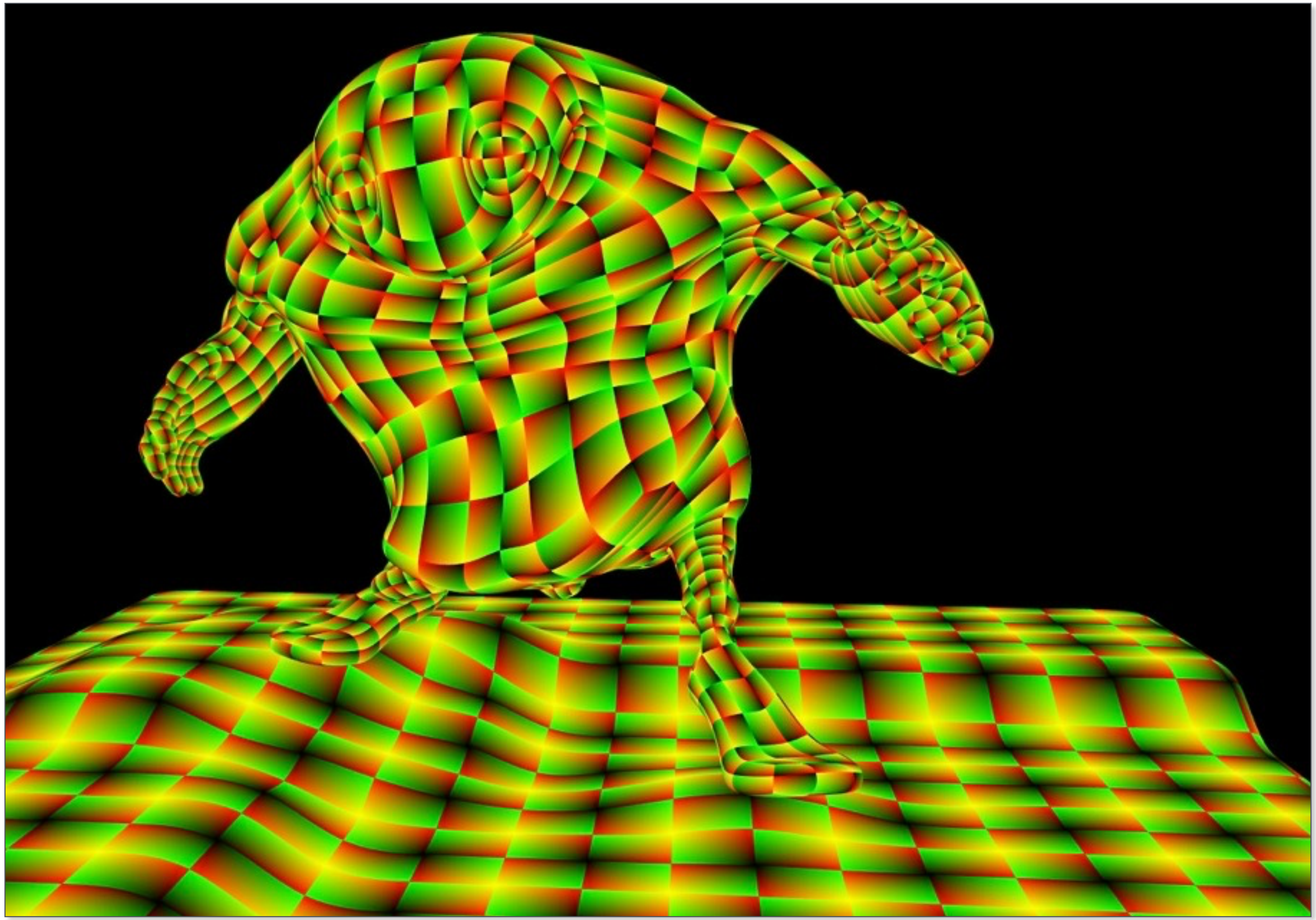
Shader function executes once per fragment.

Outputs color of surface at sample point corresponding to fragment.

(this shader performs a texture lookup to obtain the surface's material color at this point, then performs a simple lighting computation)

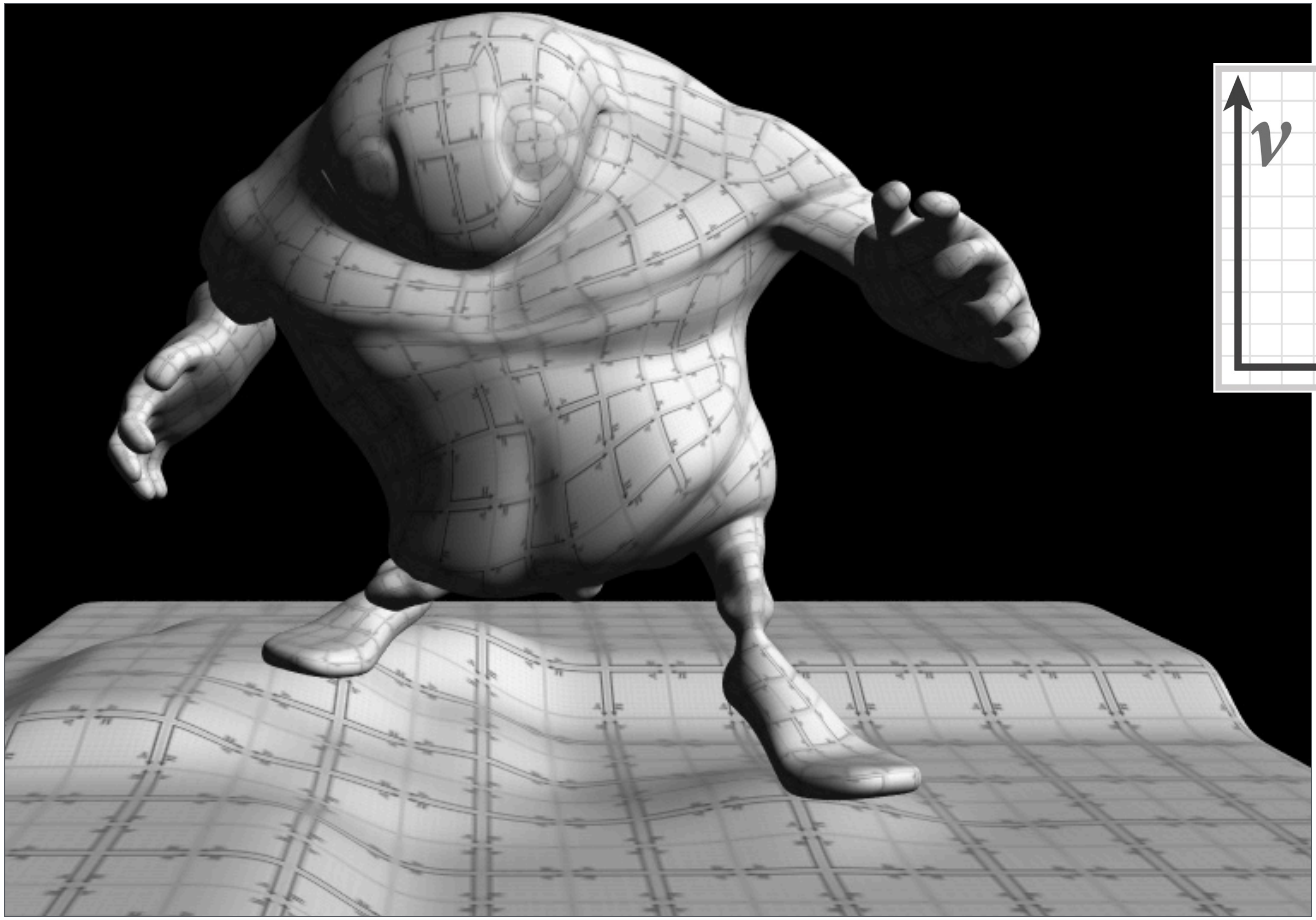
Texture coordinate visualization

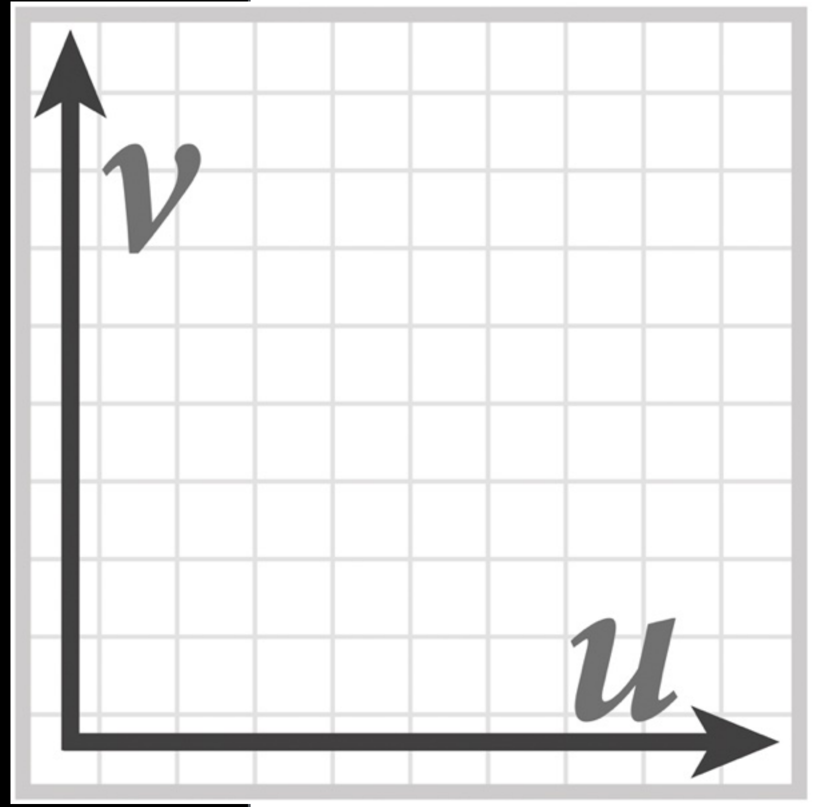
Defines mapping from point on surface to point (uv) in texture domain



Red channel = u, Green channel = v
So uv=(0,0) is black, uv=(1,1) is yellow

Rendered result (after evaluating fragment shader for each pixel)





Goal: render very high complexity 3D scenes

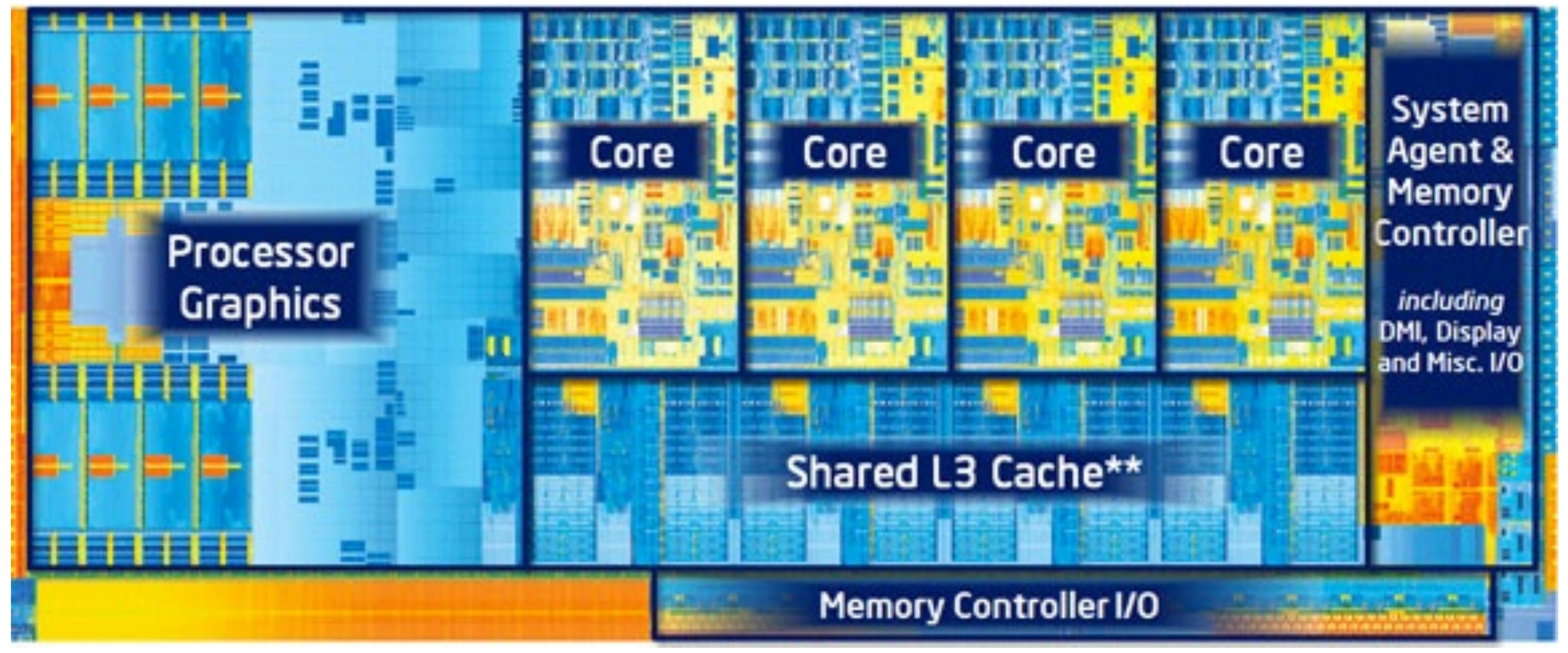
- 100's of thousands to millions of triangles in a scene
- Complex vertex and fragment shader computations
- High resolution screen outputs (2-4 Mpixel + supersampling)
- 30-60 fps

Red Dead Redemption 2

Graphics pipeline implementation: GPUs

Specialized processors for executing graphics pipeline computations

Discrete GPU card
(NVIDIA RTX 4090 GPU)

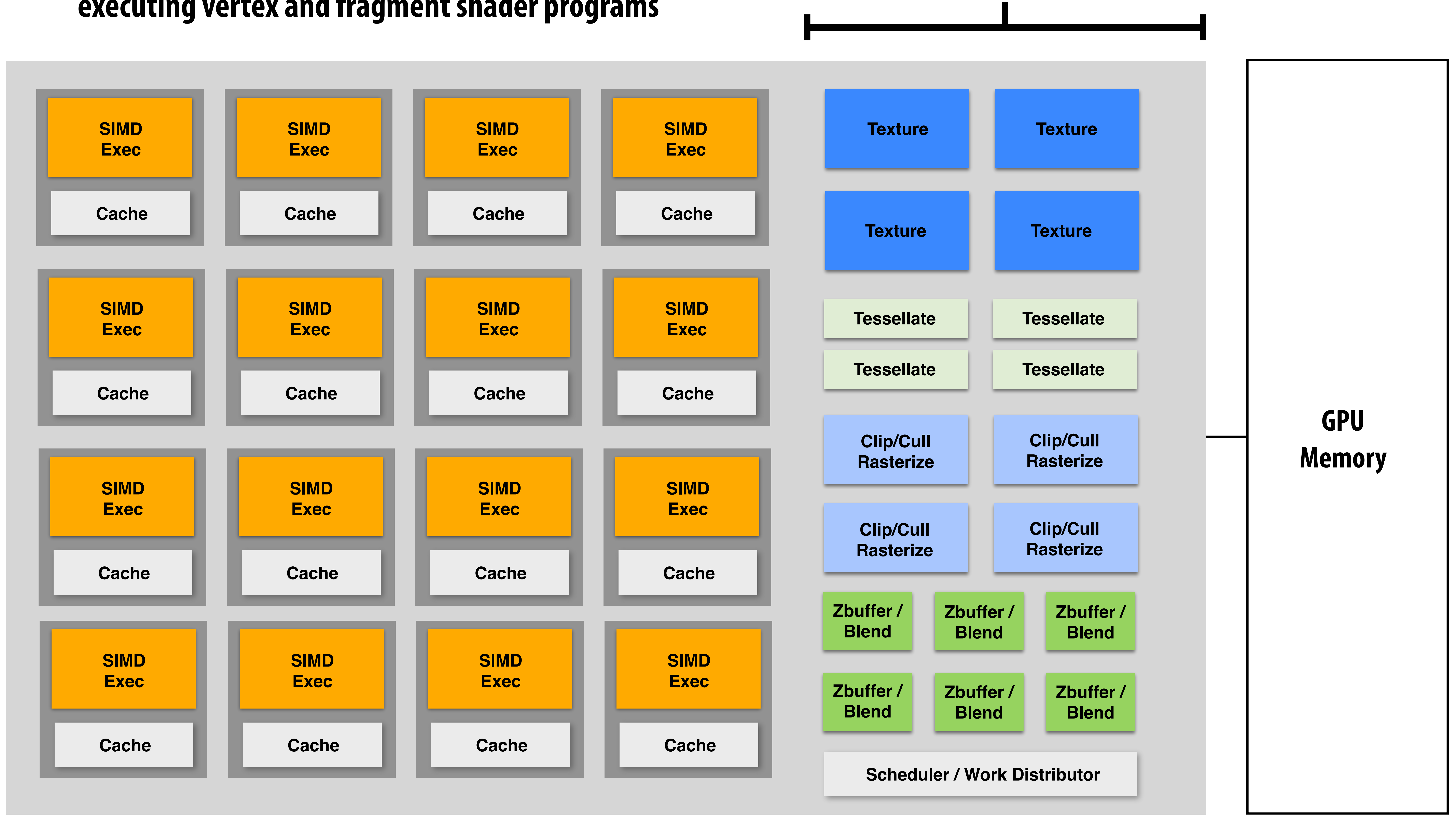


Integrated GPU: part of modern Intel CPU chip

GPU: heterogeneous, multi-core processor

Modern GPUs offer ~2-4 TFLOPs of performance for executing vertex and fragment shader programs

T-OP's of fixed-function compute capability over here



Take Kayvon's Visual Computing Systems course (CS348V) for more details!

Summary

- Occlusion resolved independently at each screen sample using the depth buffer
- Alpha compositing for semi-transparent surfaces
  - Premultiplied alpha forms simply repeated composition
  - “Over” compositing operations is not commutative: requires triangles to be processed in back-to-front (or front-to-back) order

- Rasterization-based GPU-accelerated graphics pipeline:
  - Abstracts rendering computation as a sequence of operations performed on vertices, primitives (e.g., triangles), fragments, and screen samples
  - Behavior of parts of the pipeline is application-defined… using shader programs
  - Pipeline operations implemented by highly, optimized parallel processors and fixed-function hardware (GPUs)