### Lecture 5:

# The Rasterization Pipeline

## (and its implementation on GPUs)

---

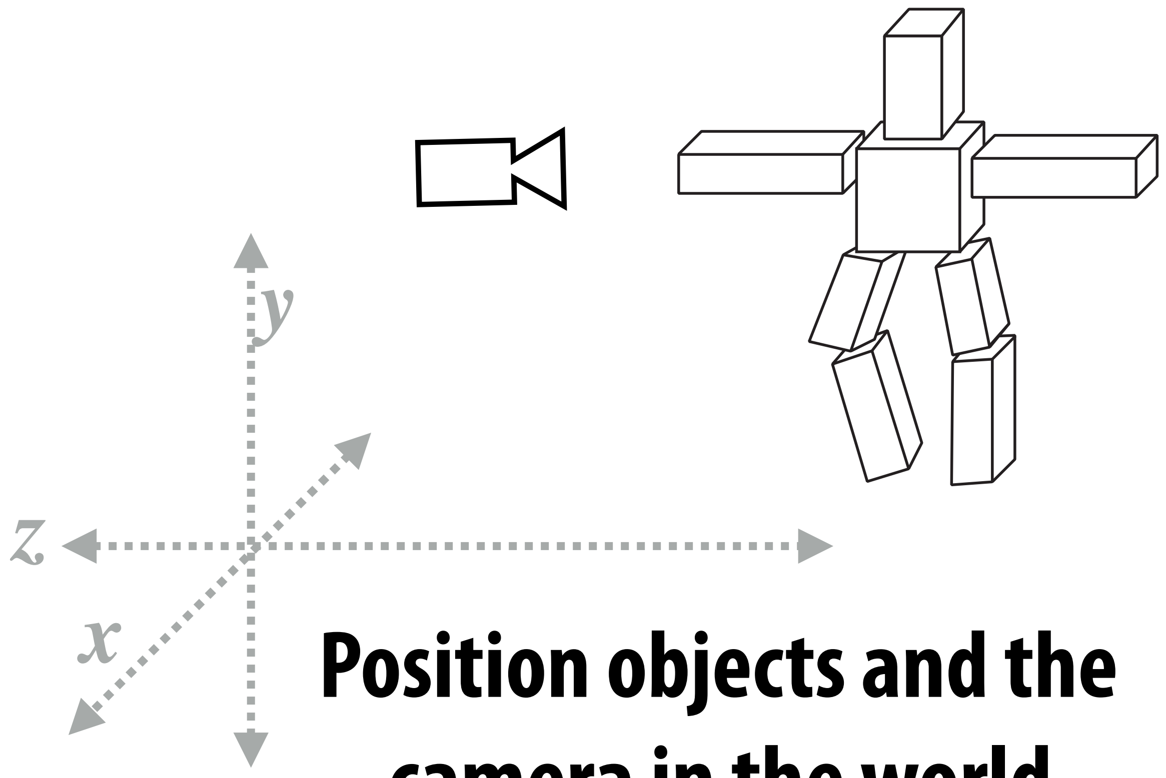
**Interactive Computer Graphics**
**Stanford CS248, Winter 2020**

# Tunes

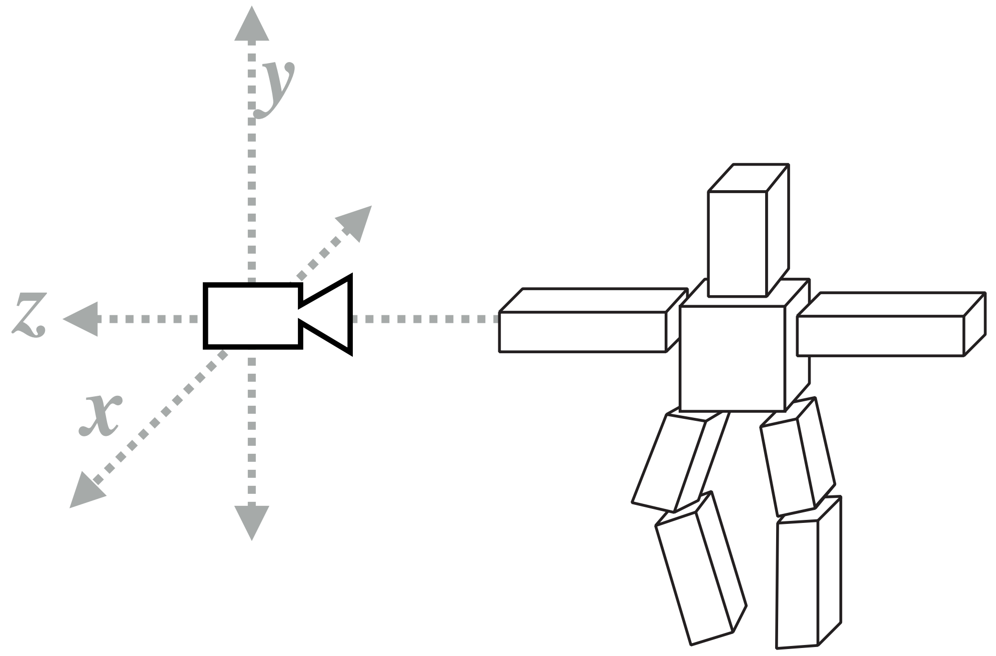
## Amy Winehouse
## “Back to Black”
### (Back to Black)

*“It’s what happens to your silhouettes when you forget to use premultiplied alpha.”*
*- Amy Winehouse*

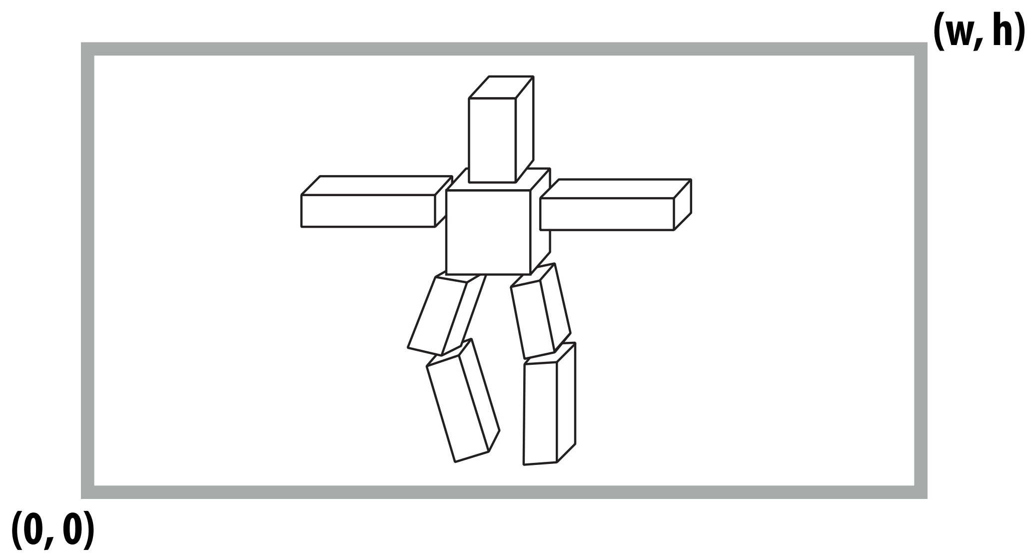
# What you know how to do (at this point in the course)



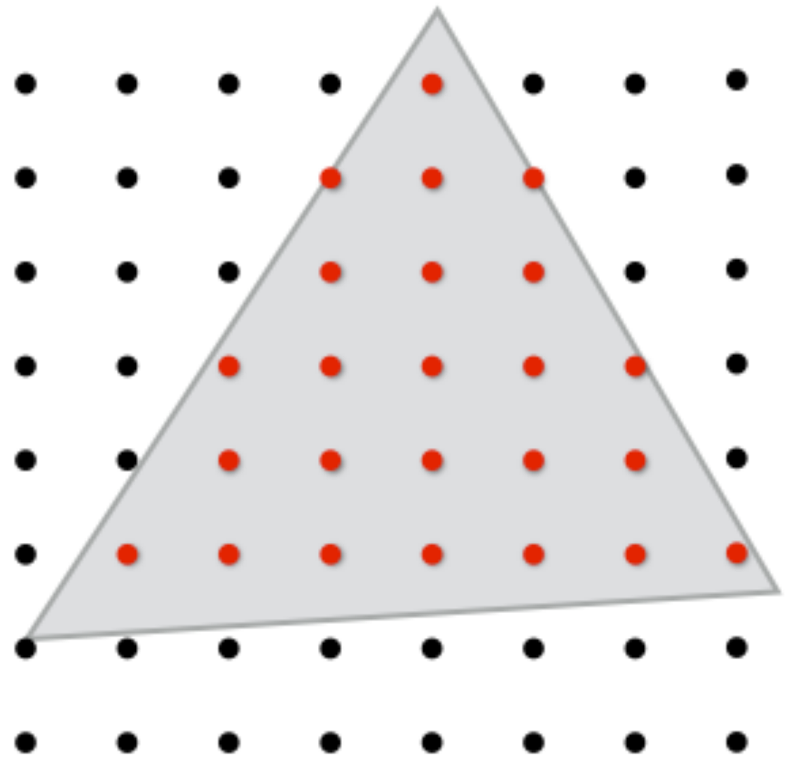
Position objects and the camera in the world



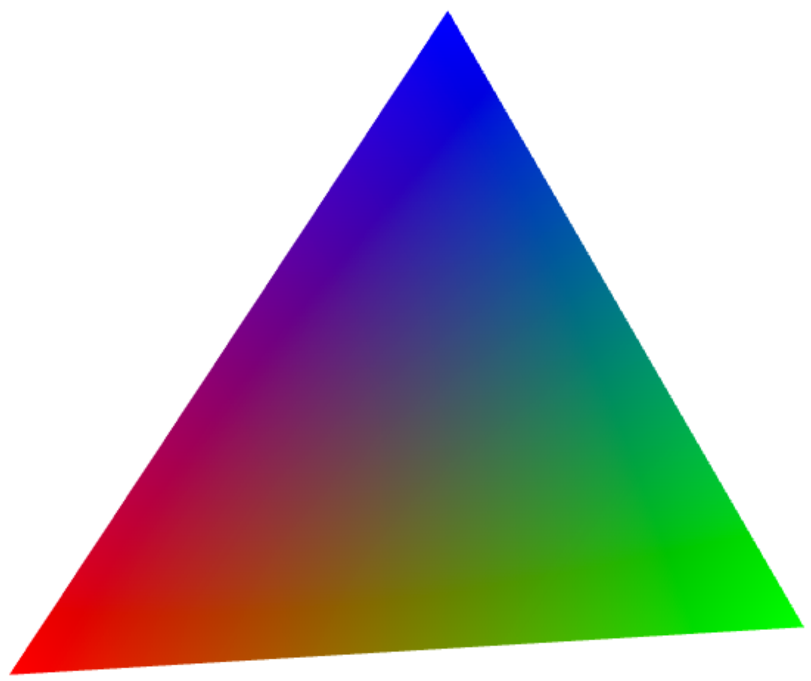
Determine the position of objects relative to the camera



Project objects onto the screen



Sample triangle coverage



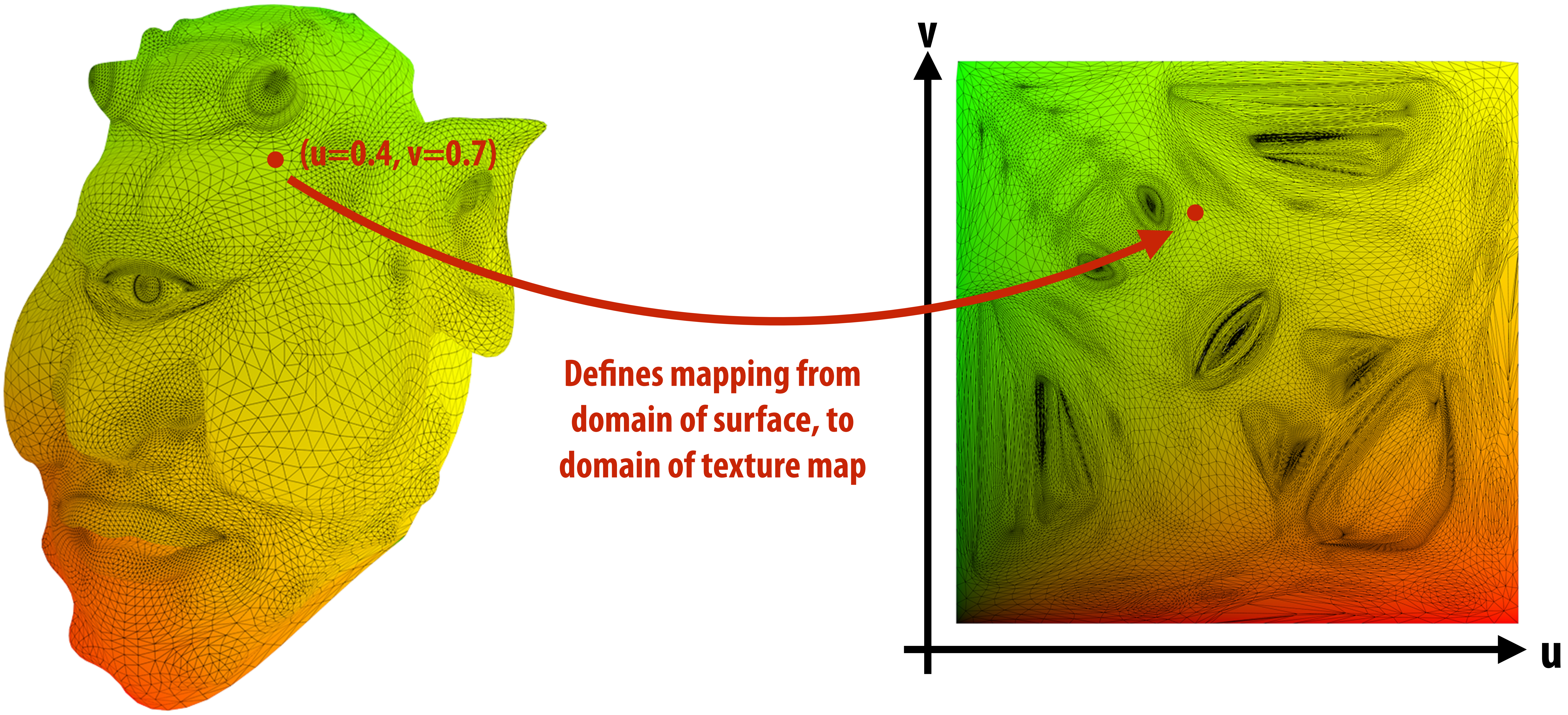
Compute triangle attribute values at covered sample points (Color, texture coords, depth)

Sample texture maps

# Texture mapping review

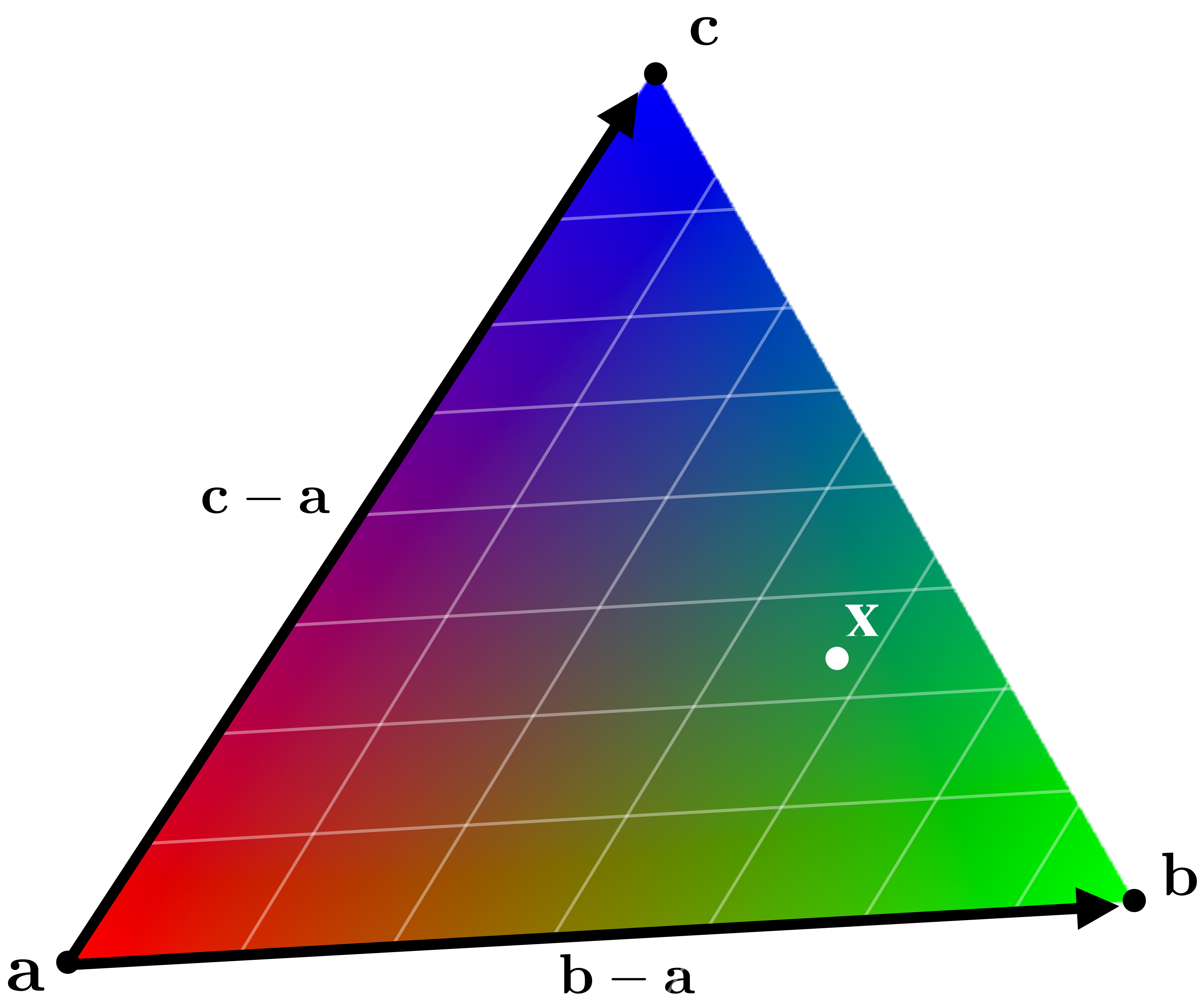
# Per-vertex information

- **Inputs:**
  - Per-vertex position [x,y,z]
  - Per-vertex texture coordinates [u,v]



Defines mapping from domain of surface, to domain of texture map

# Linearly interpolate texture coordinate samples



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

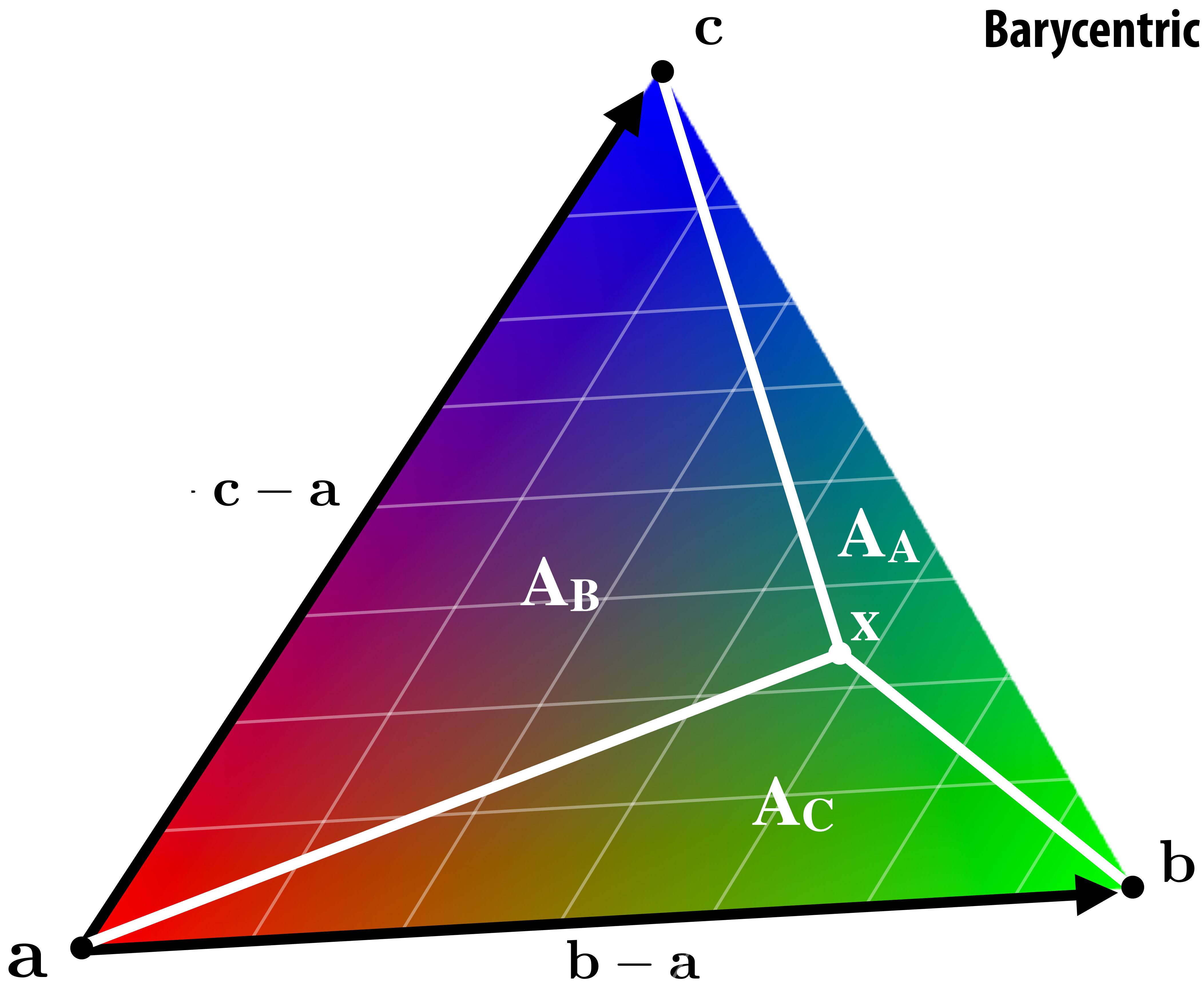
$$\begin{aligned}\mathbf{x} &= \mathbf{a} + \beta(\mathbf{b} - \mathbf{a}) + \gamma(\mathbf{c} - \mathbf{a}) \\ &= (1 - \beta - \gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} \\ &= \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}\end{aligned}$$

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

**UV at $\mathbf{x}$ is linear combination of UV at three triangle vertices.**

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

# Barycentric coordinates as ratio of areas



**Barycentric coordinates as ratio of *signed* areas:**

$$\alpha = A_A / A$$

$$\beta = A_B / A$$

$$\gamma = A_C / A$$

**Given XYZ positions of triangle vertices, compute barycentric coordinates…**

# Interpolating texture coordinates in 2D

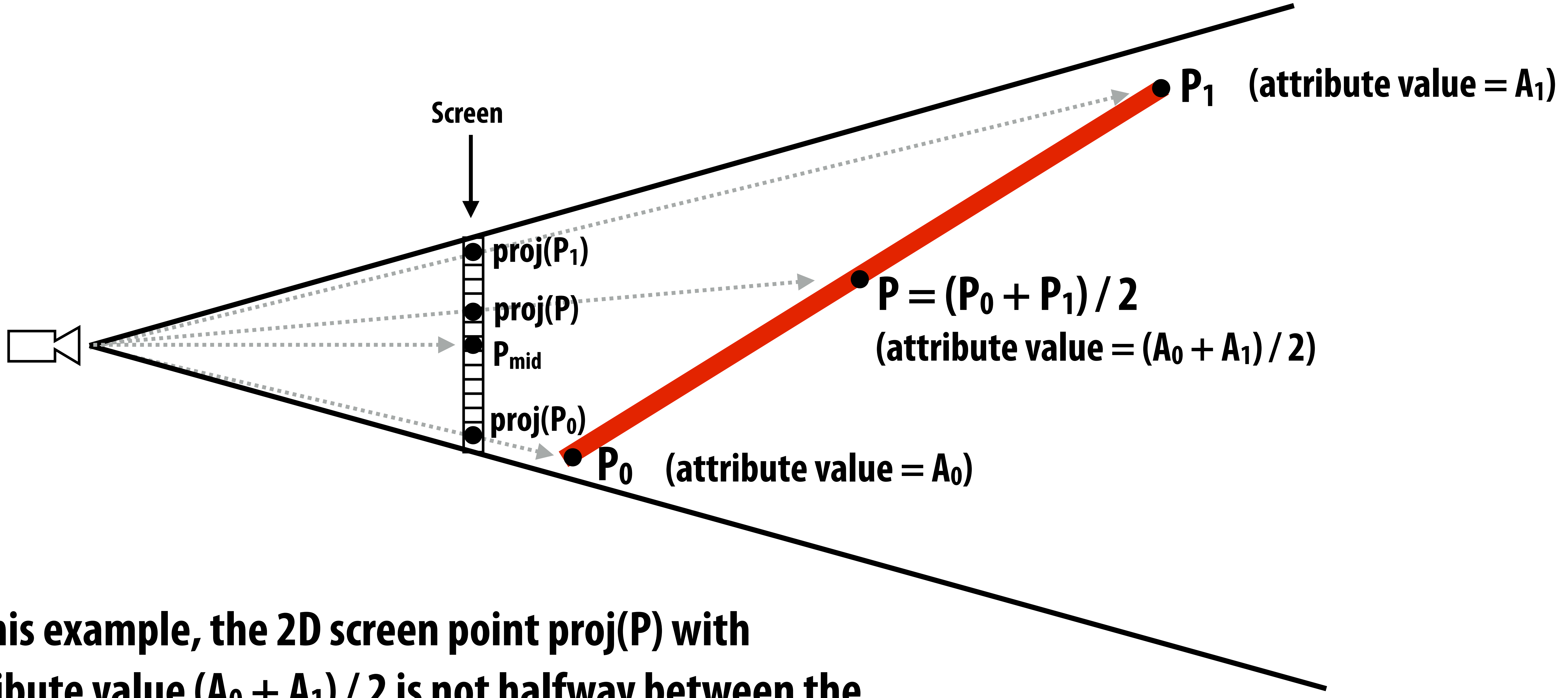
- **But consider assignment 1…**

- **You are given 2D position of triangle coordinates, and you have to sample coverage (and now UV) at a given 2D screen point (X,Y)**

# 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 is not affine in 2D screen XY coordinates**



**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 $(A_0 + A_1) / 2$.**

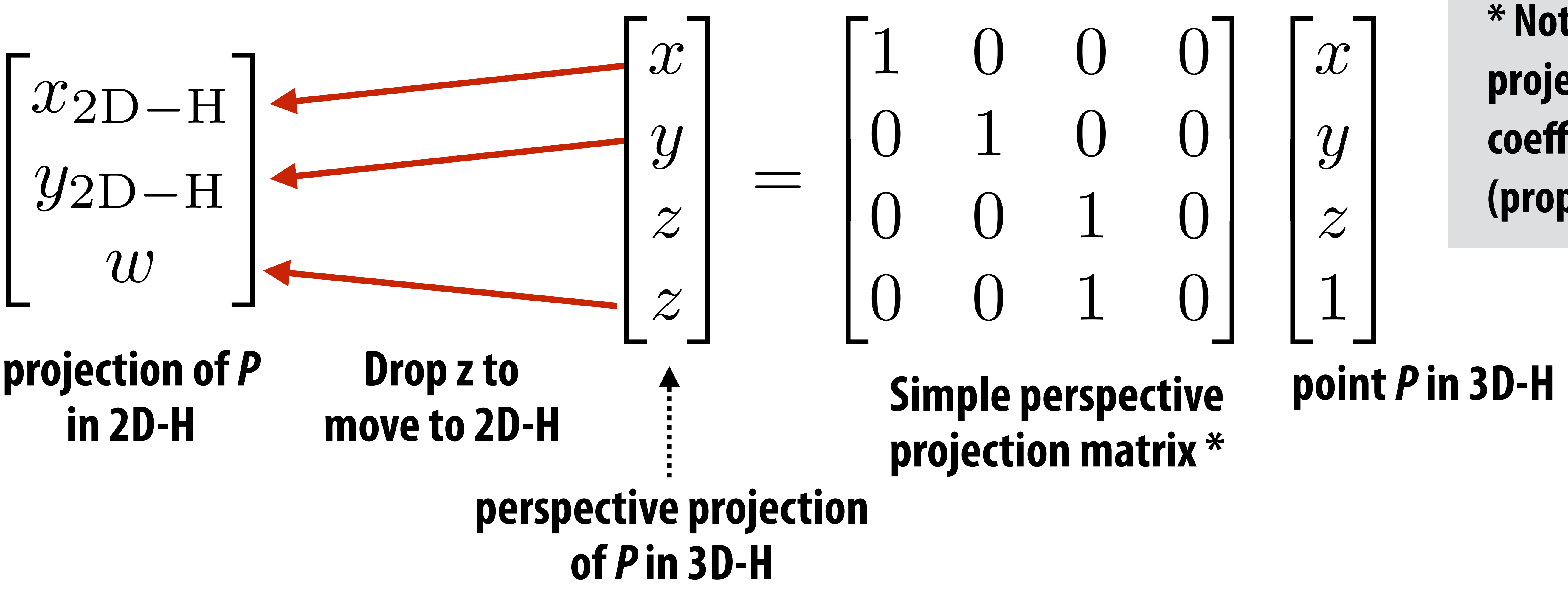
# Perspective-correct interpolation

**Assume triangle attribute varies linearly across the triangle**

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

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

**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…**

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

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

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

**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

**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***

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

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

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

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

**This is done as a per triangle "setup" computation prior to sampling, just like you computed edge equations for evaluating coverage.**

# Efficient perspective-correct interpolation

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

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

To evaluate surface attribute $f$ at every covered sample:

- Evaluate ${}^{1}/_{w}\,(x,y)$ (from precomputed equation for 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

# What else do you need to know to render a picture like this?

**Surface representation**

How to represent complex surfaces?

**Occlusion**

Determining which surface is visible to the camera at each sample point

**Lighting/materials**

Describing lights in scene and how materials reflect light.

# Course roadmap: what's coming...

## Drawing Things

- Introduction
- Drawing a triangle (by sampling)
- Transforms and coordinate spaces
- Perspective projection and texture sampling
- Today: putting it all together: end-to-end rasterization pipeline

Key concepts:

Sampling (and anti-aliasing)
Coordinate Spaces and Transforms
Rasterization and texturing via sampling

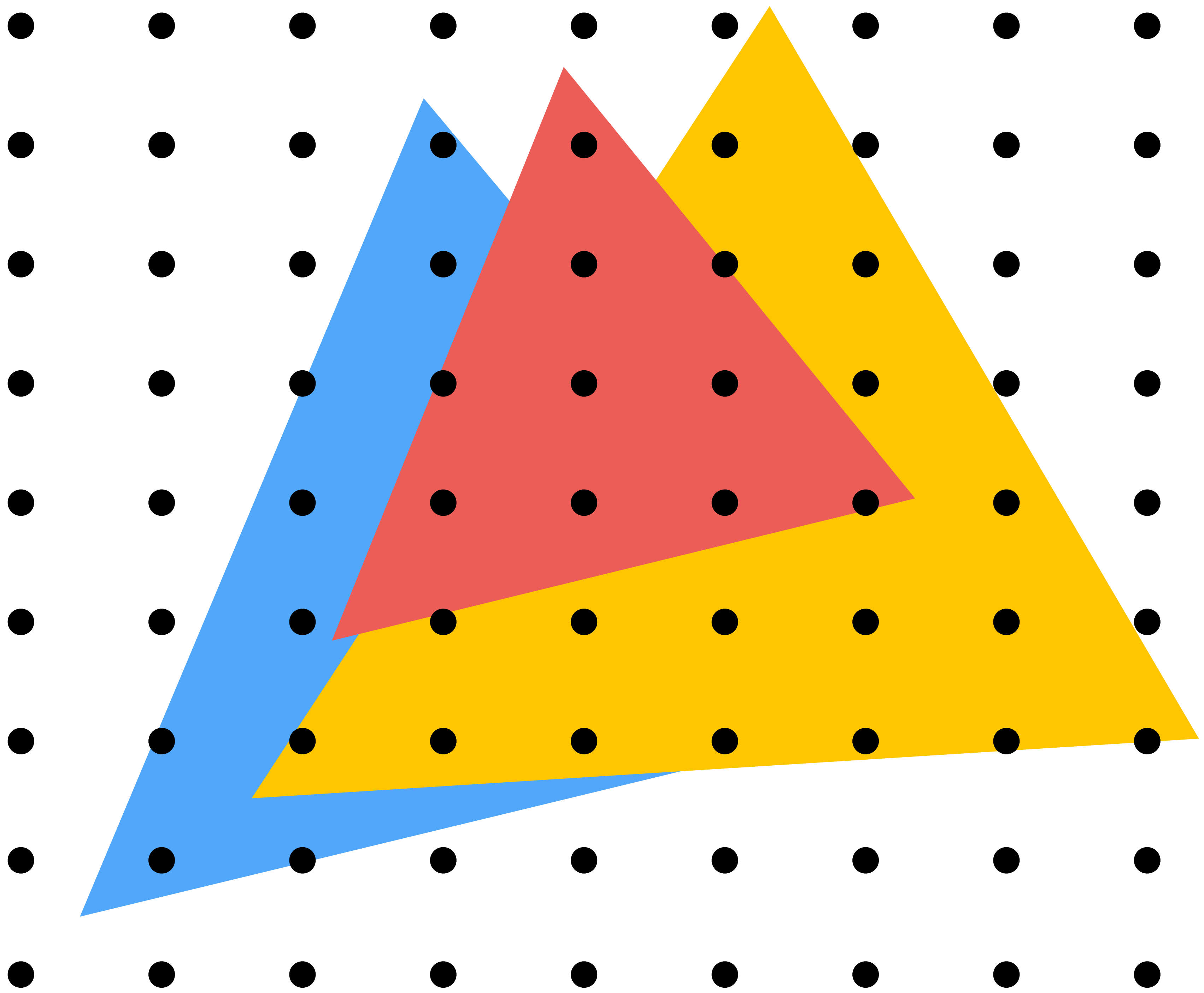
## Geometry Processing

## Materials and Lighting

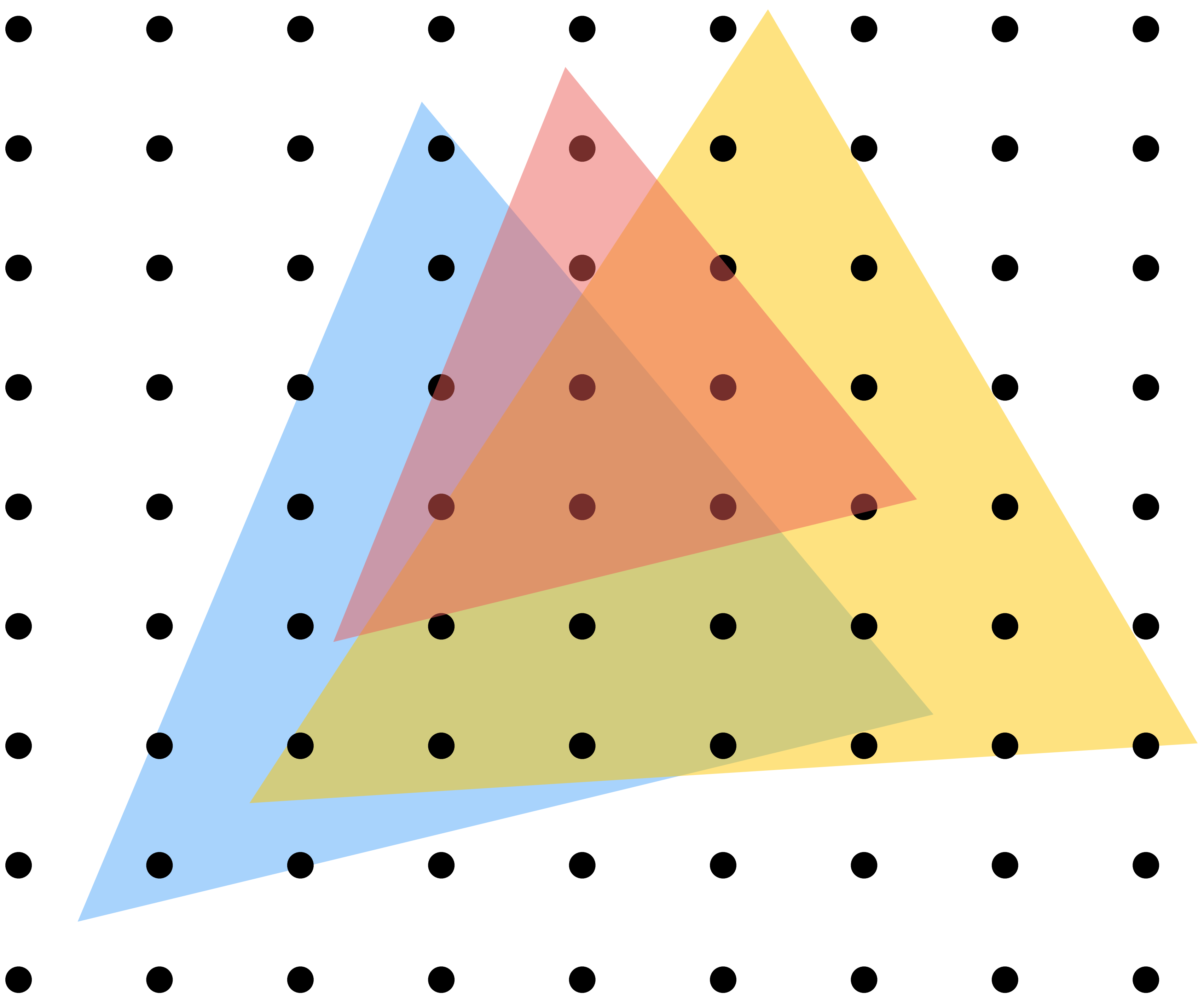
## Midterm

# Occlusion using the Depth Buffer

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



Opaque Triangles



50% transparent triangles

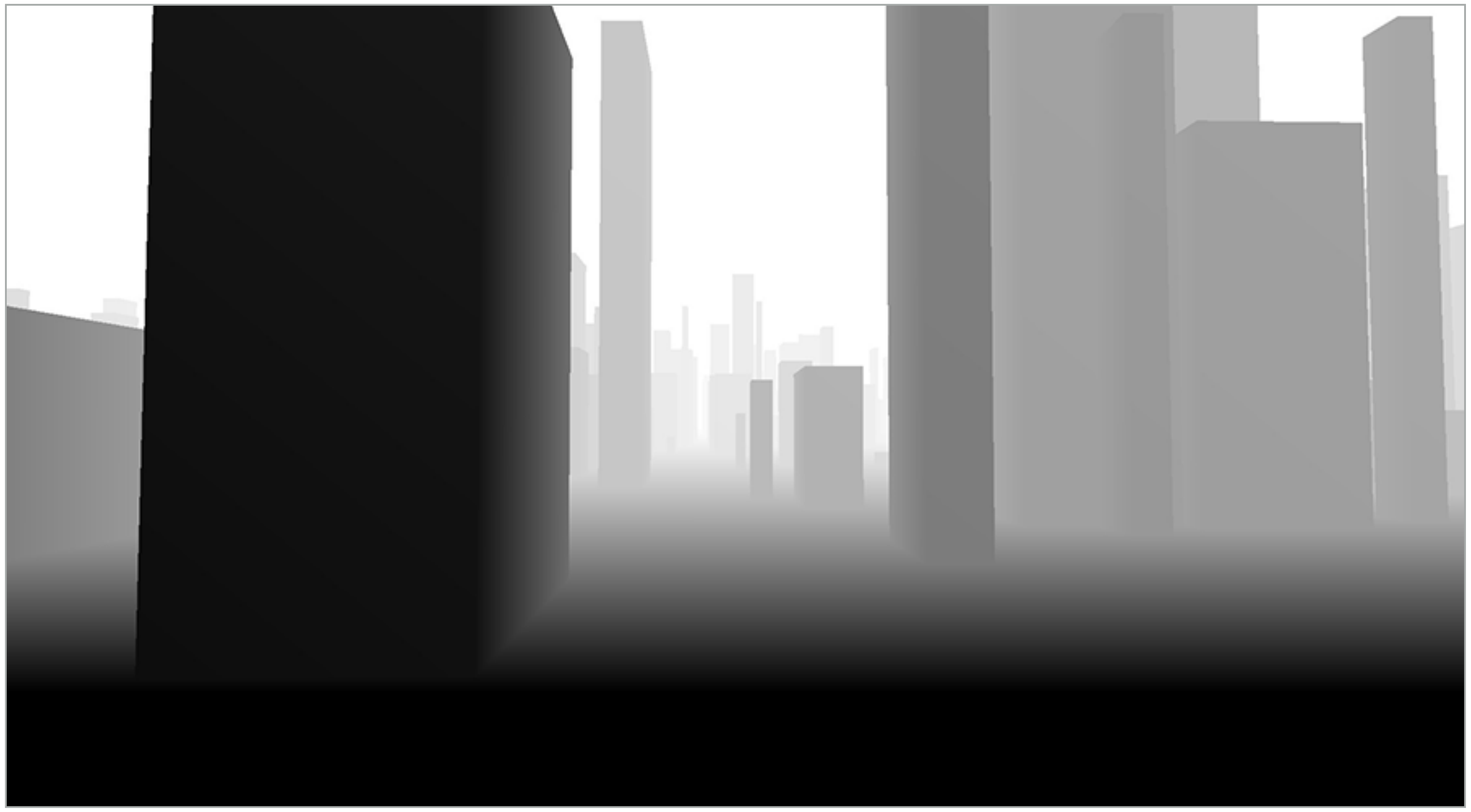
# Depth buffer (aka "Z buffer")

**Color buffer:**

**(stores color per sample… e.g., RGB)**

**Depth buffer:**

**(stores depth per sample)**



Stores depth of closest surface drawn so far

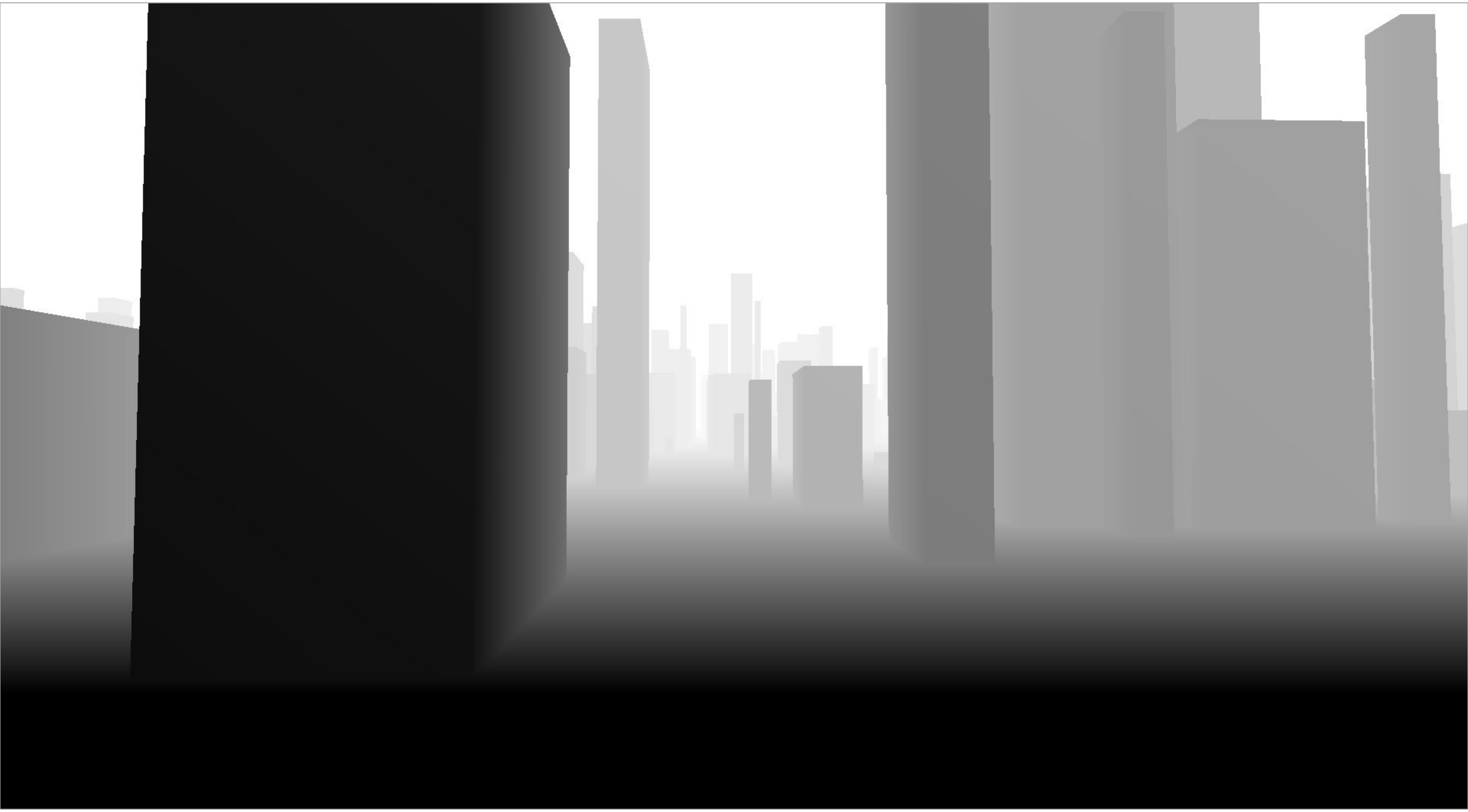
black = close depth

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)



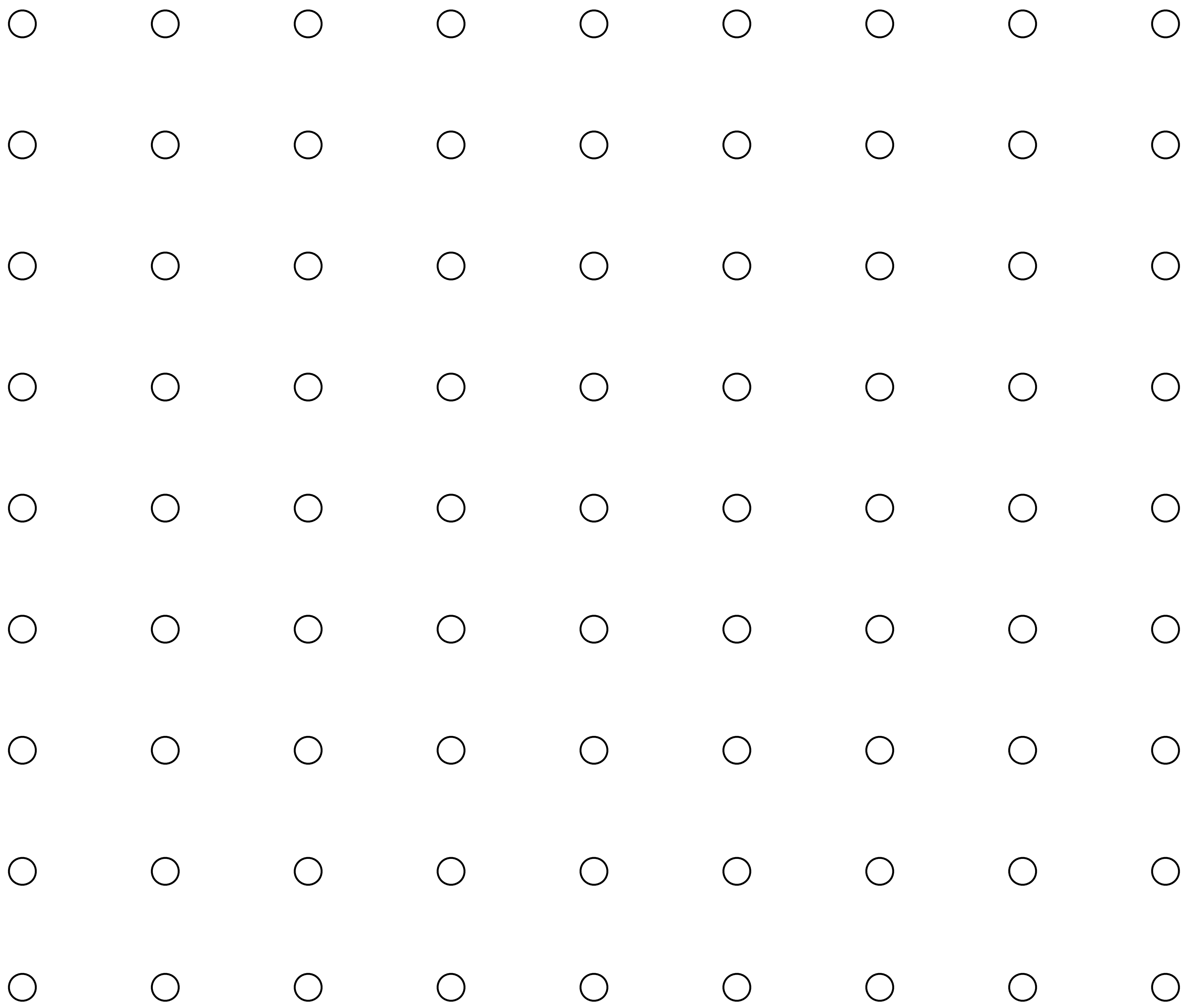
**Corresponding depth buffer after rendering all triangles**
**(stores closest scene depth per sample)**

# 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**

# Review from last class

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

$$\begin{bmatrix}\mathbf{p}_{0x} & \mathbf{p}_{0y}\end{bmatrix}^T, \quad d_0$$

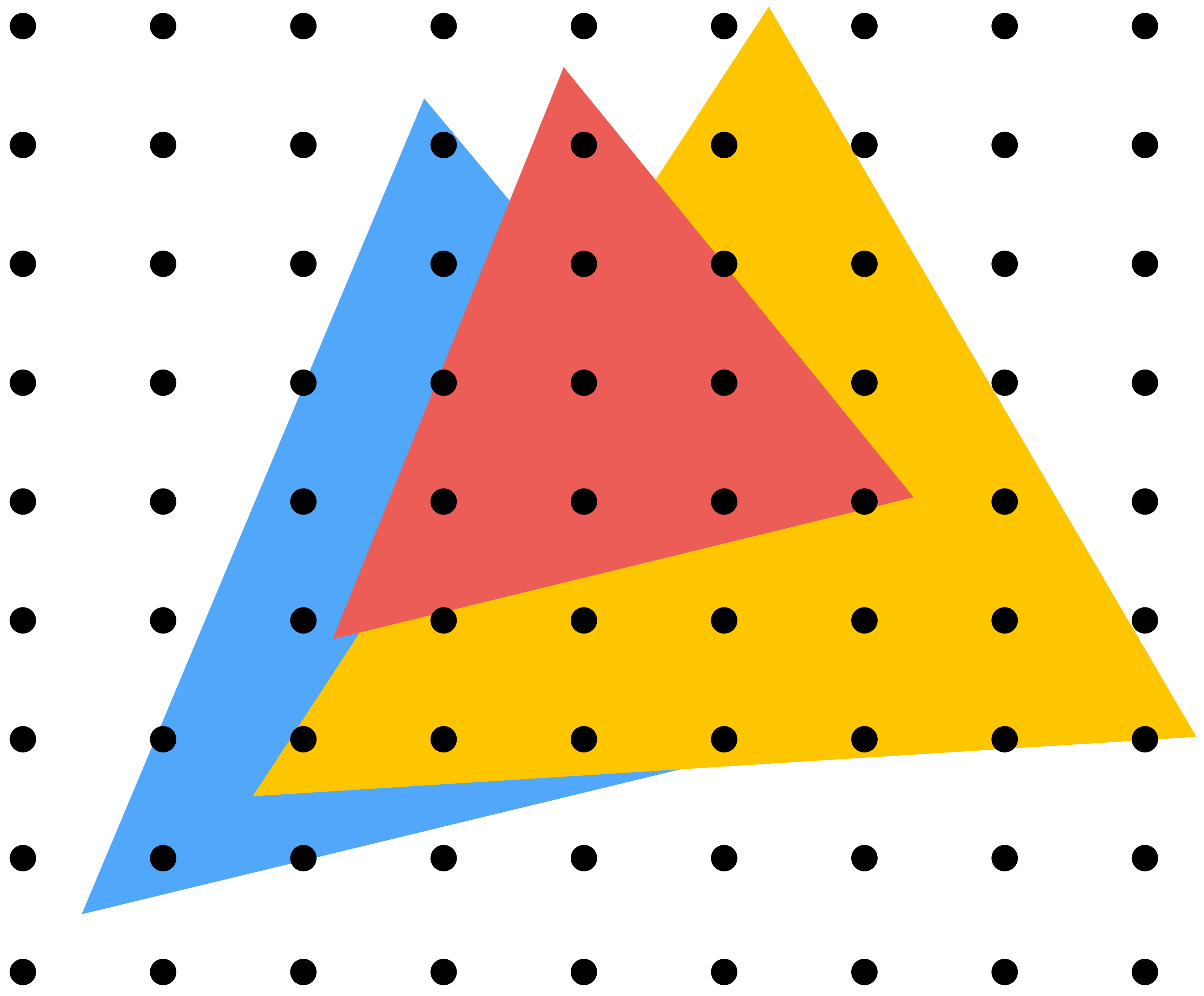
$$\begin{bmatrix}\mathbf{p}_{1x} & \mathbf{p}_{1y}\end{bmatrix}^T, \quad d_1$$

$$\begin{bmatrix}\mathbf{p}_{2x} & \mathbf{p}_{2y}\end{bmatrix}^T, \quad d_2$$

**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.**

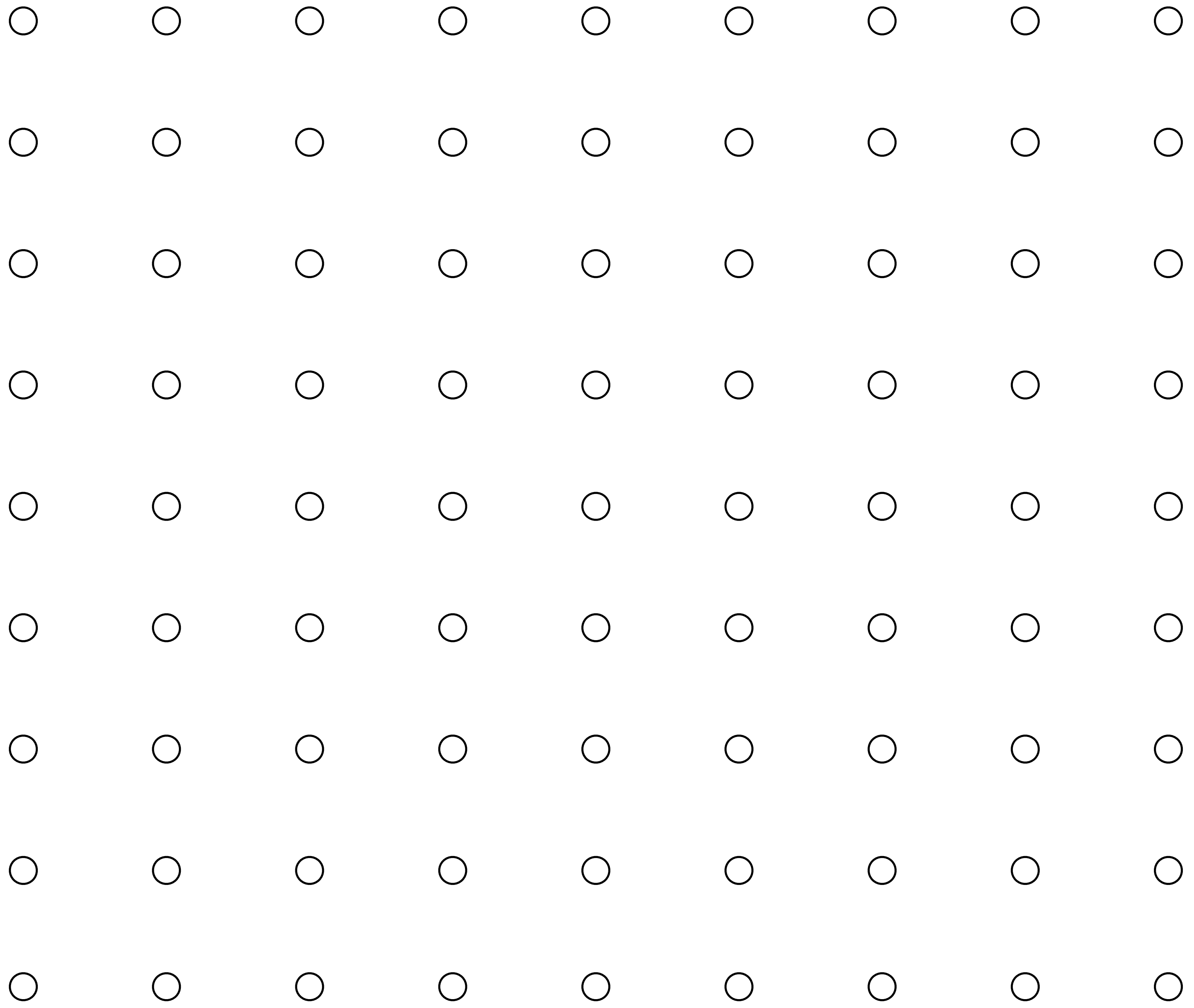
# Example: rendering three opaque triangles



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

**Processing yellow triangle:**

**depth = 0.5**



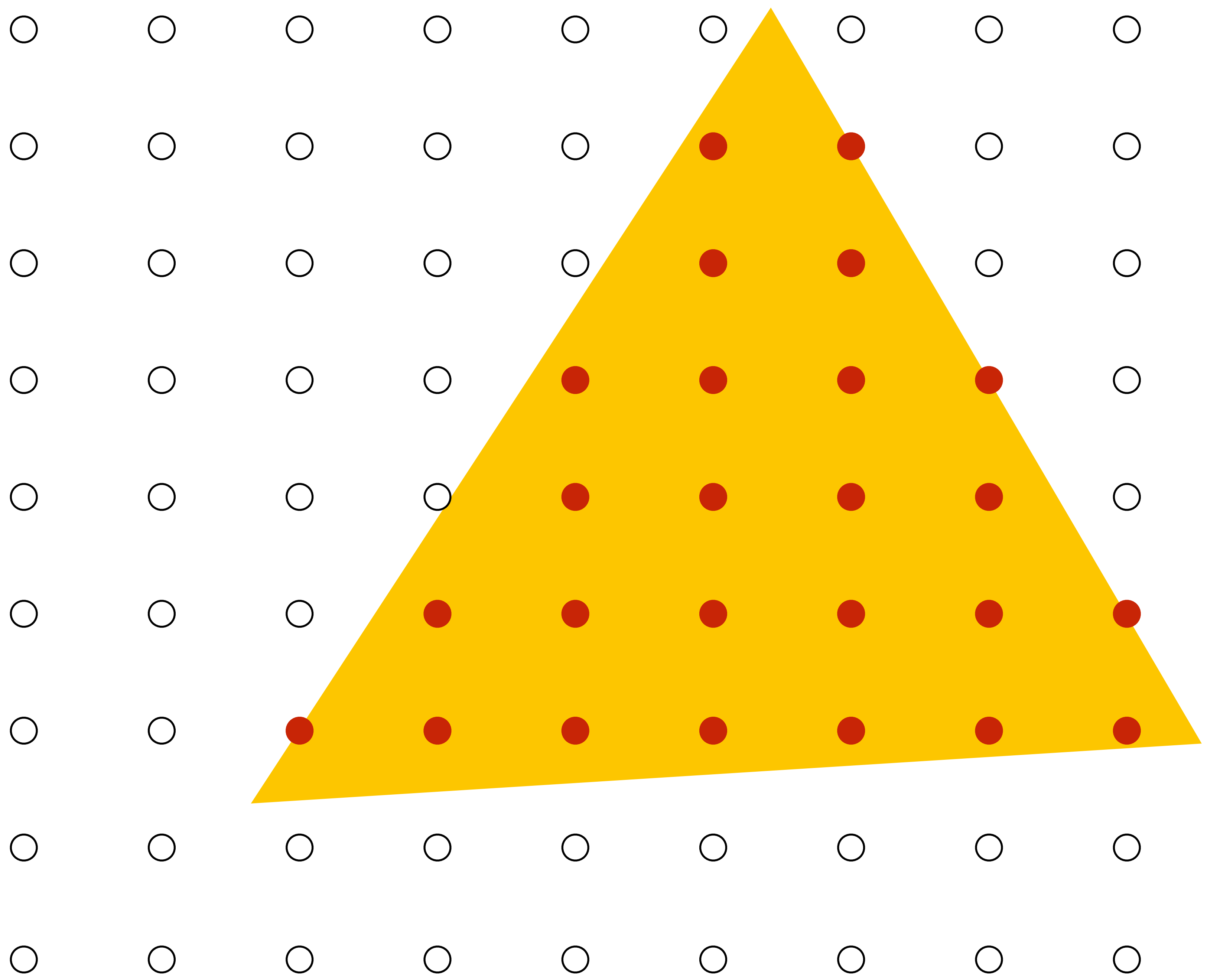
Color buffer contents

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)

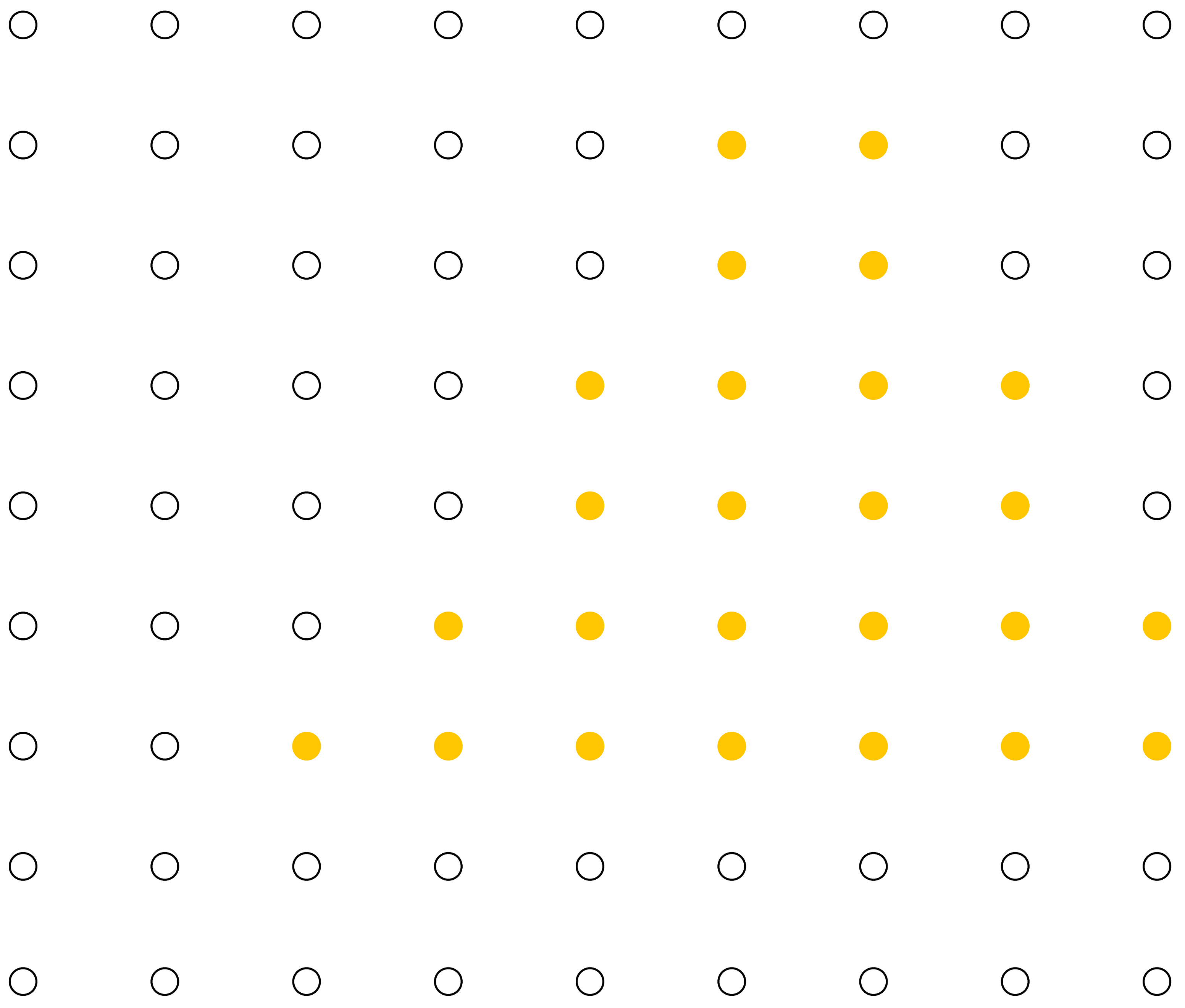
## After processing yellow triangle:

Grayscale value of sample point used to indicate distance

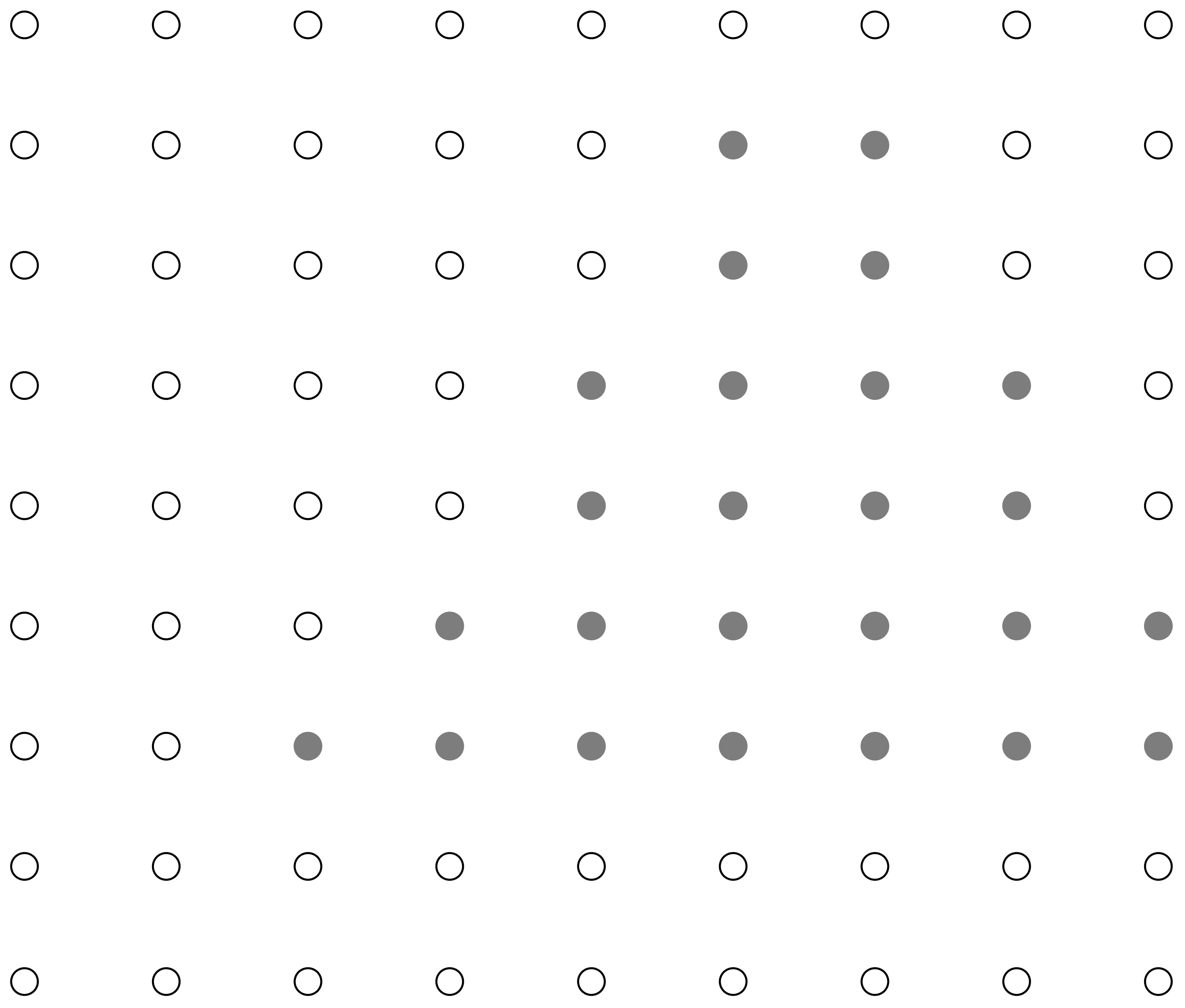
White = large distance

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:

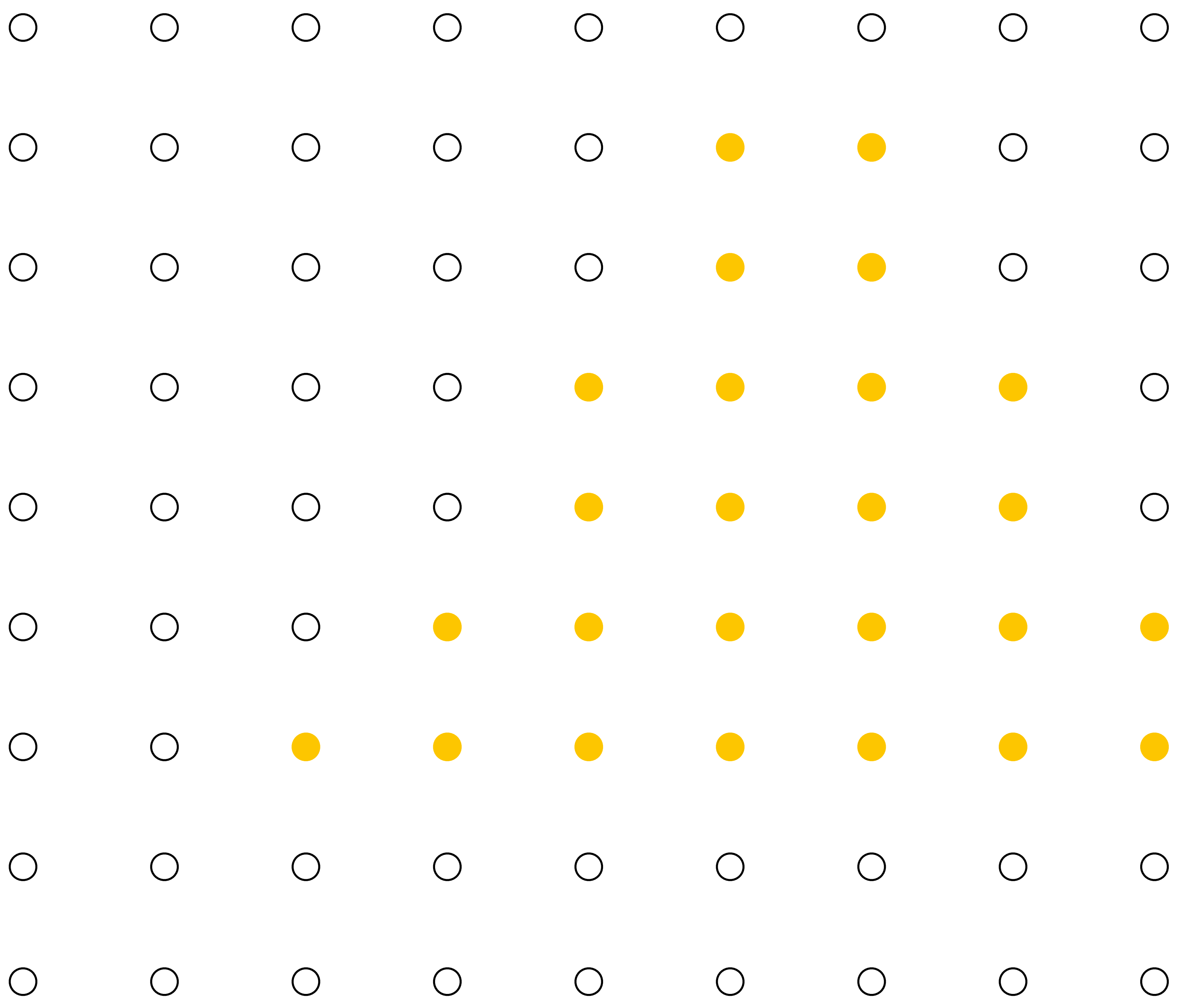
depth = 0.75

Grayscale value of sample point used to indicate distance

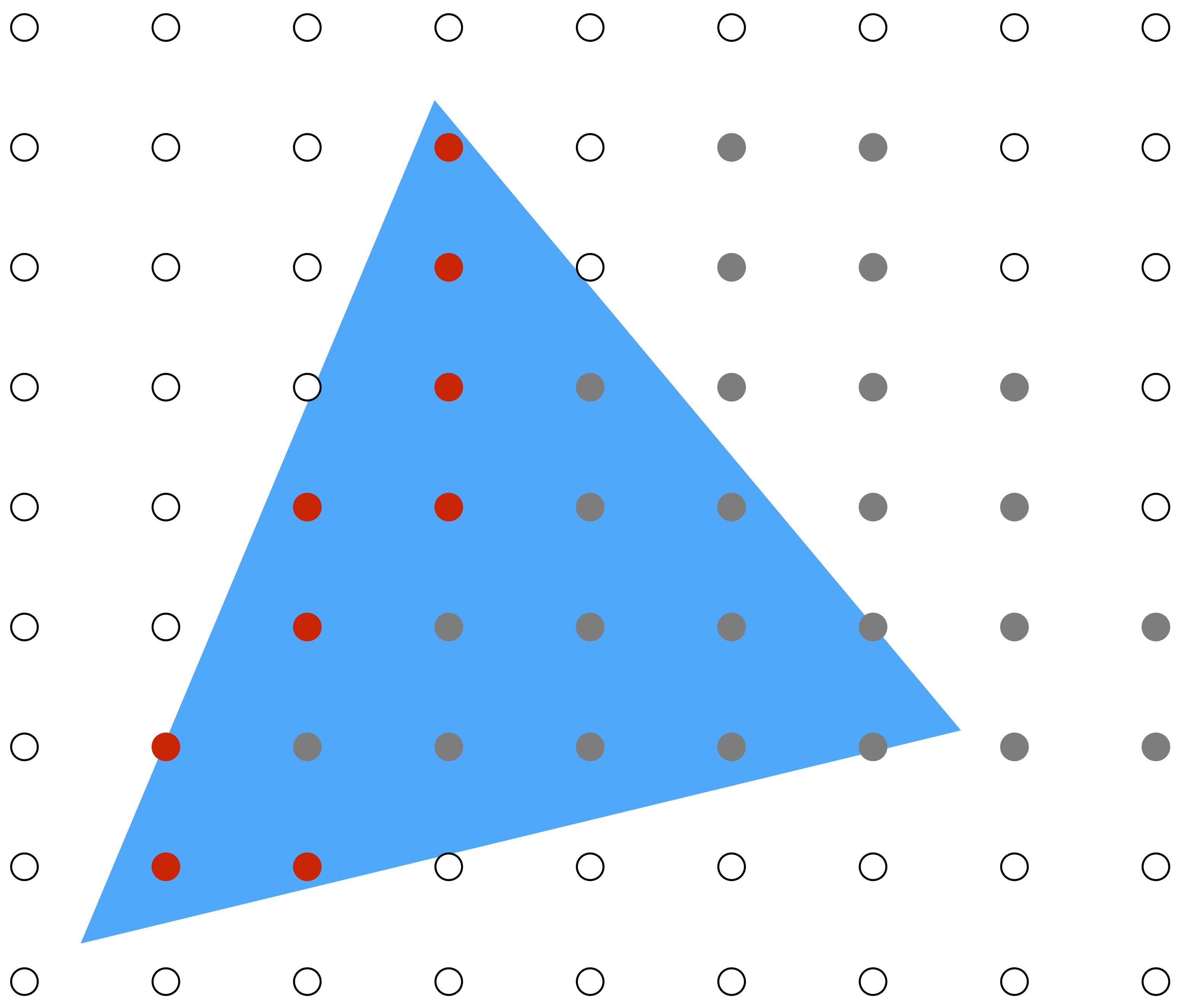
White = large distance

Black = small distance

Red = samples that pass depth test



Color buffer contents



Depth buffer contents

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

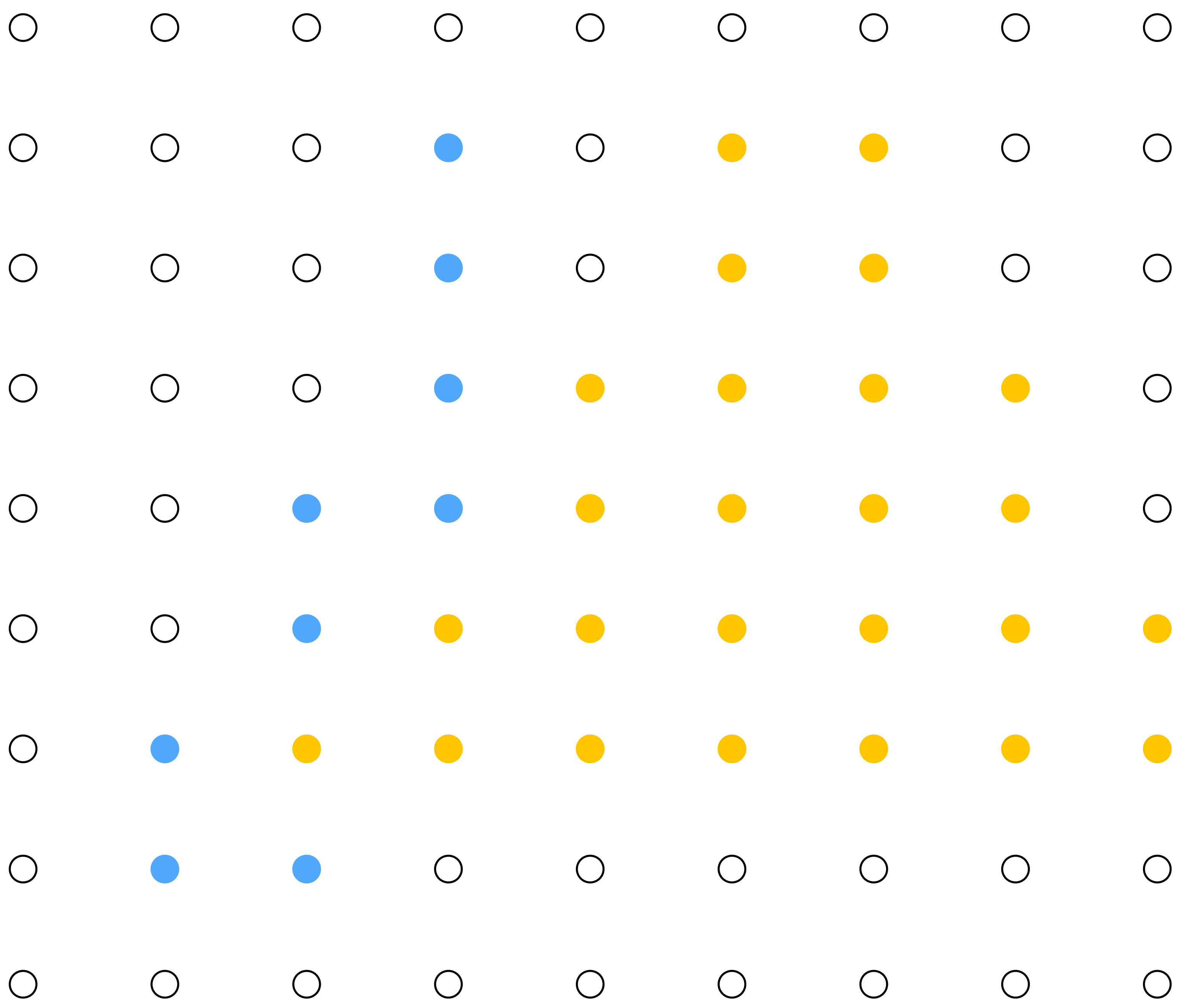
## After processing blue triangle:

Grayscale value of sample point used to indicate distance

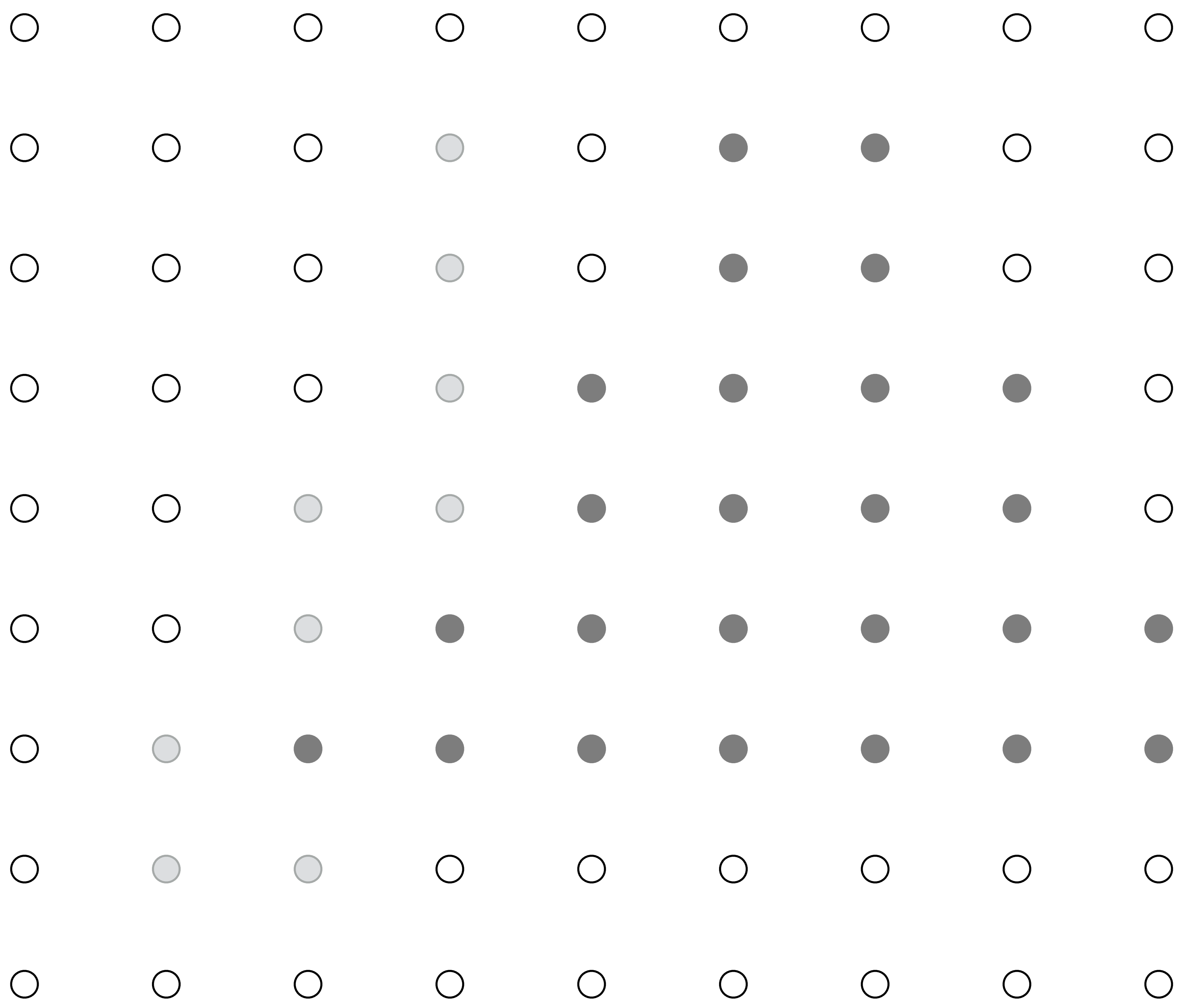
White = large distance

Black = small distance

Red = samples that pass depth test



Color buffer contents



Depth buffer contents

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

**Processing red triangle:**

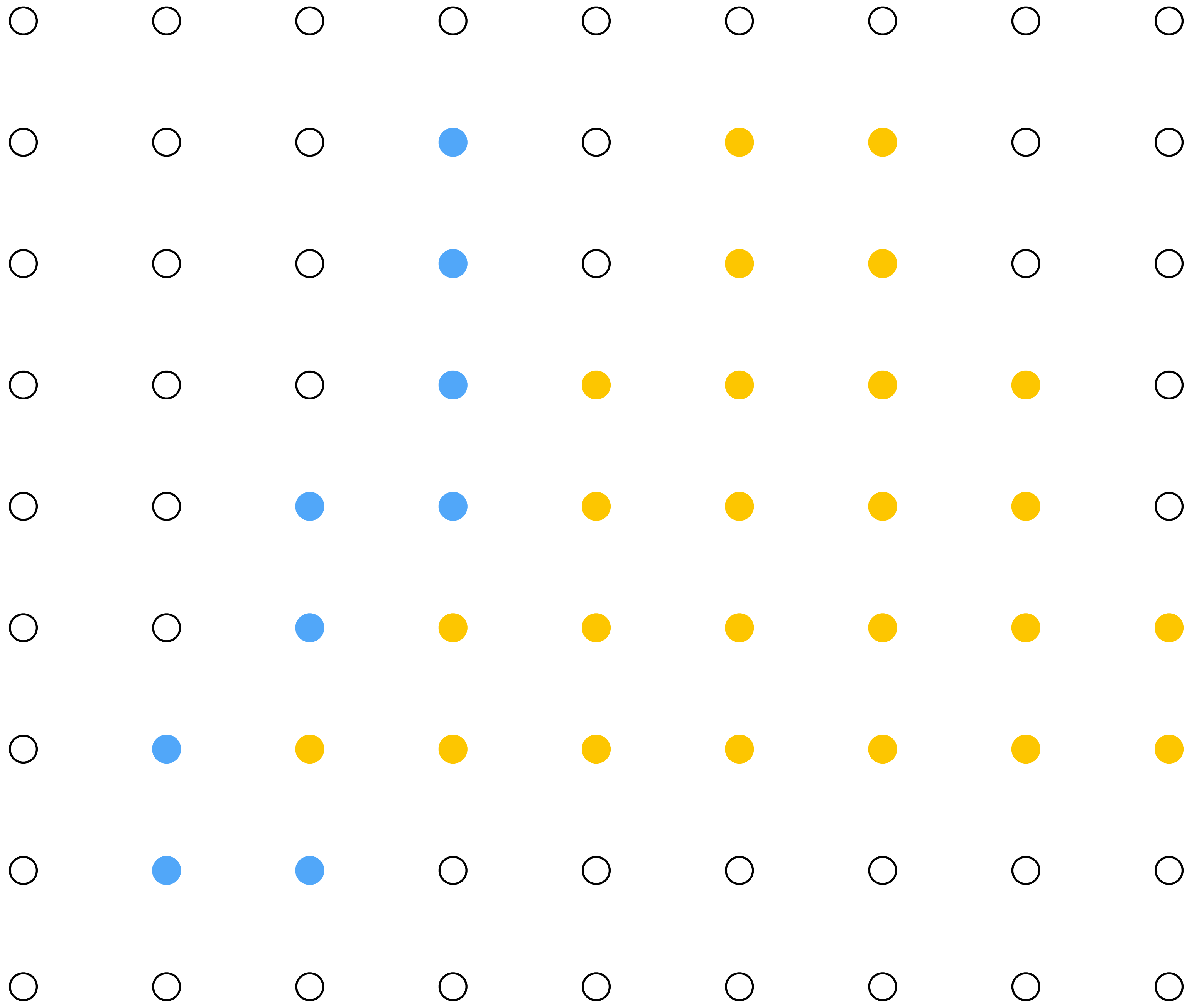
**depth = 0.25**

Grayscale value of sample point used to indicate distance

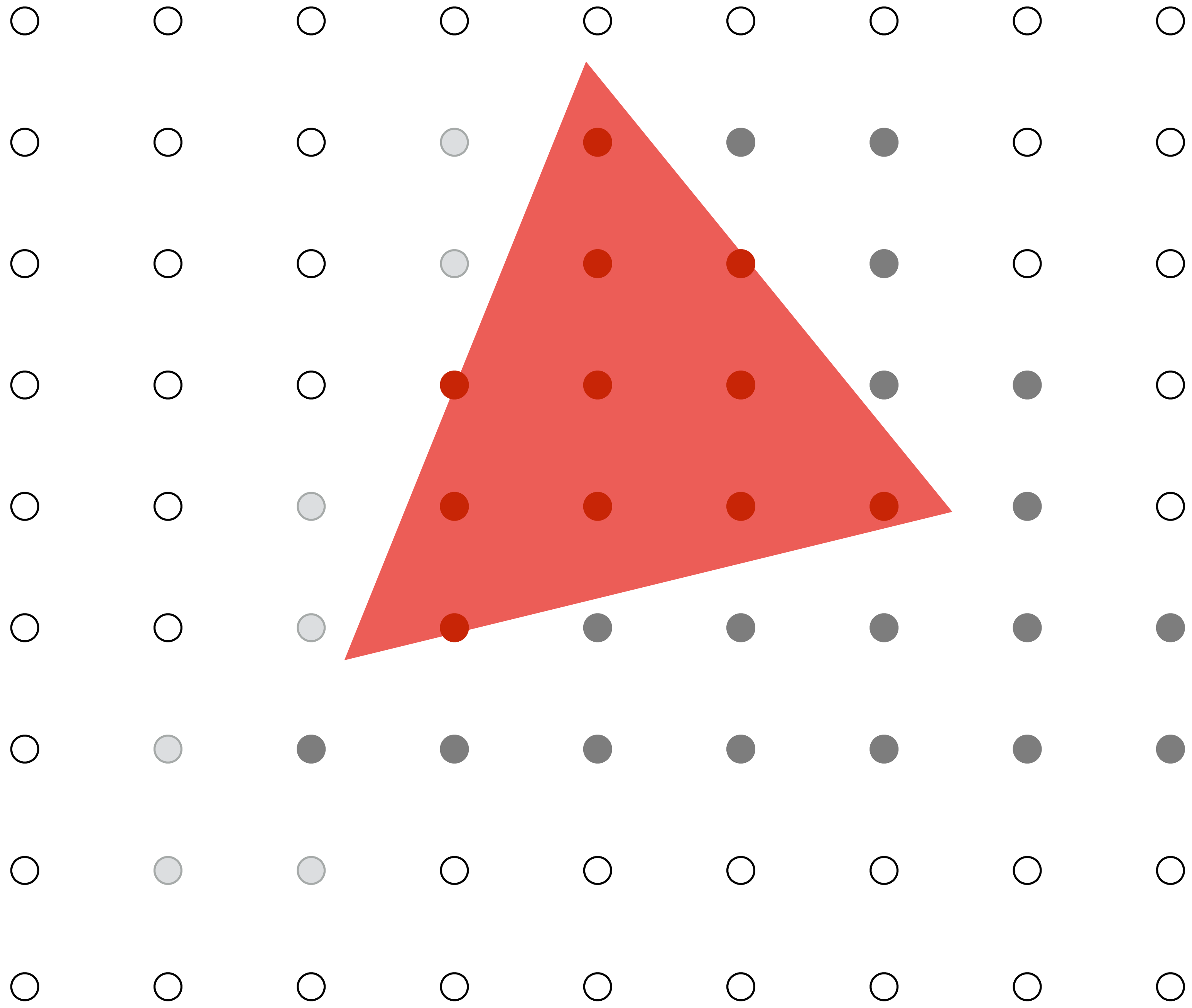
White = large distance

Black = small distance

Red = samples that pass depth test



Color buffer contents



Depth buffer contents

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

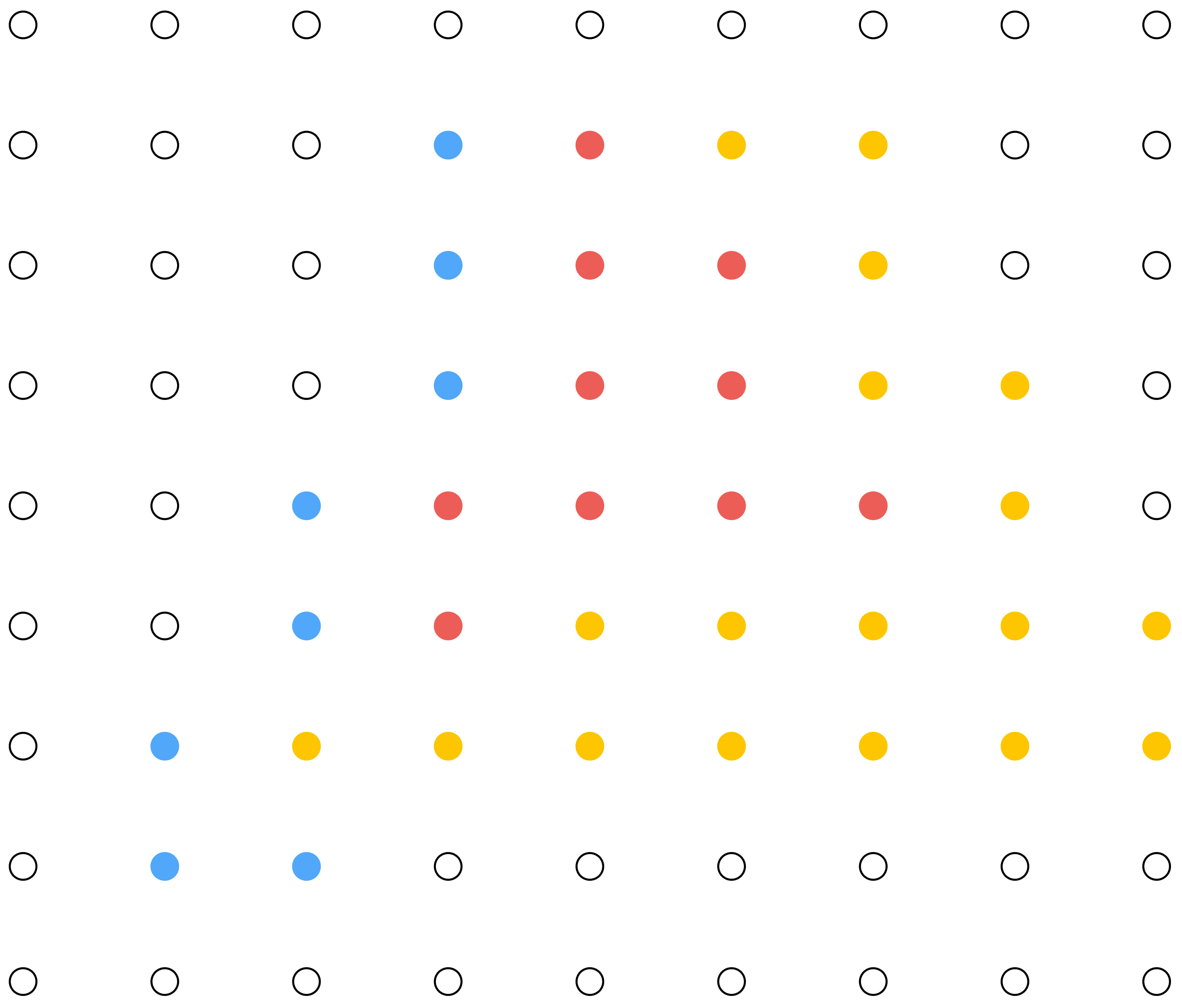
## After processing red triangle:

Grayscale value of sample point used to indicate distance

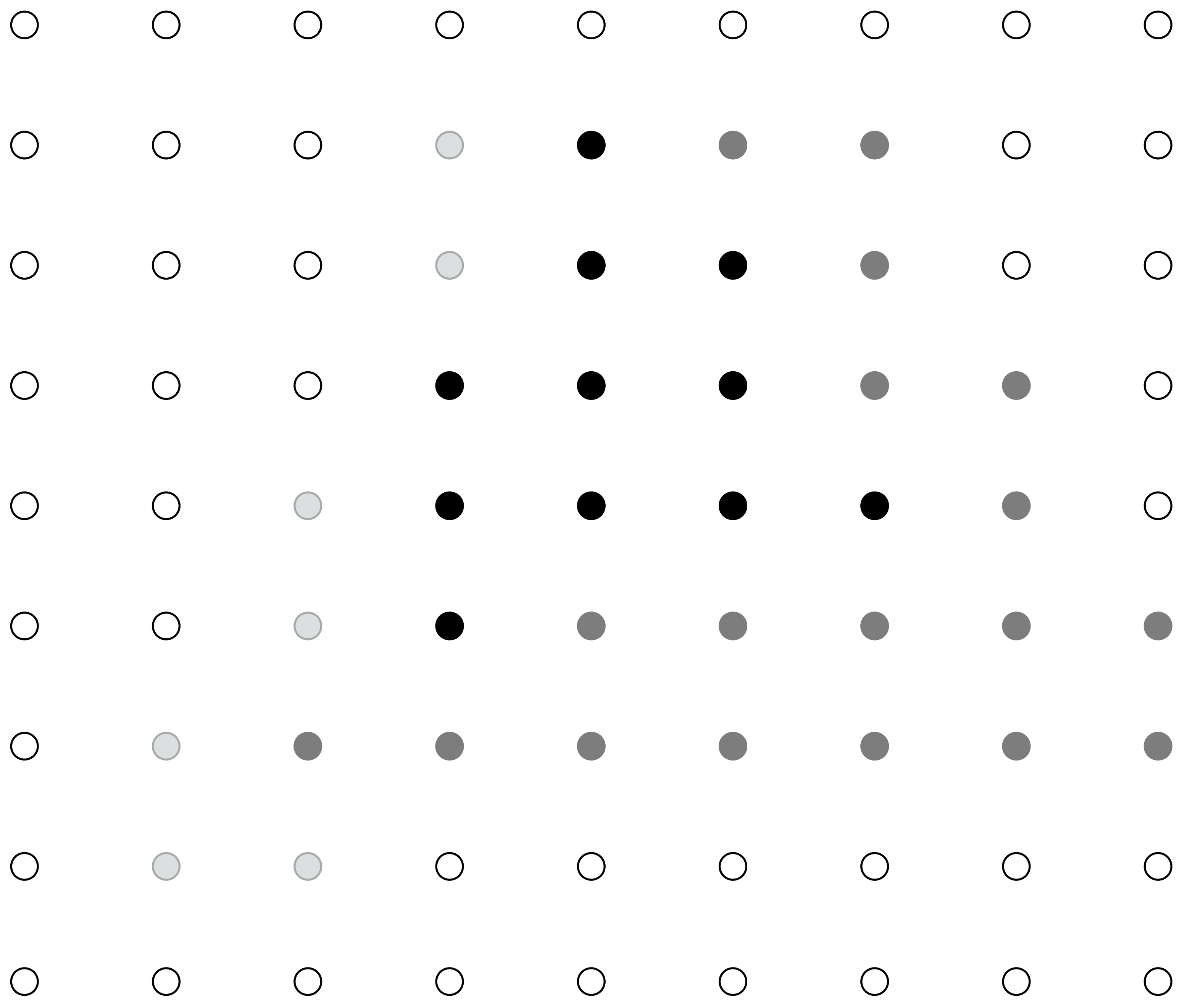
White = large distance

Black = small distance

Red = samples that pass depth test



Color buffer contents



Depth buffer contents

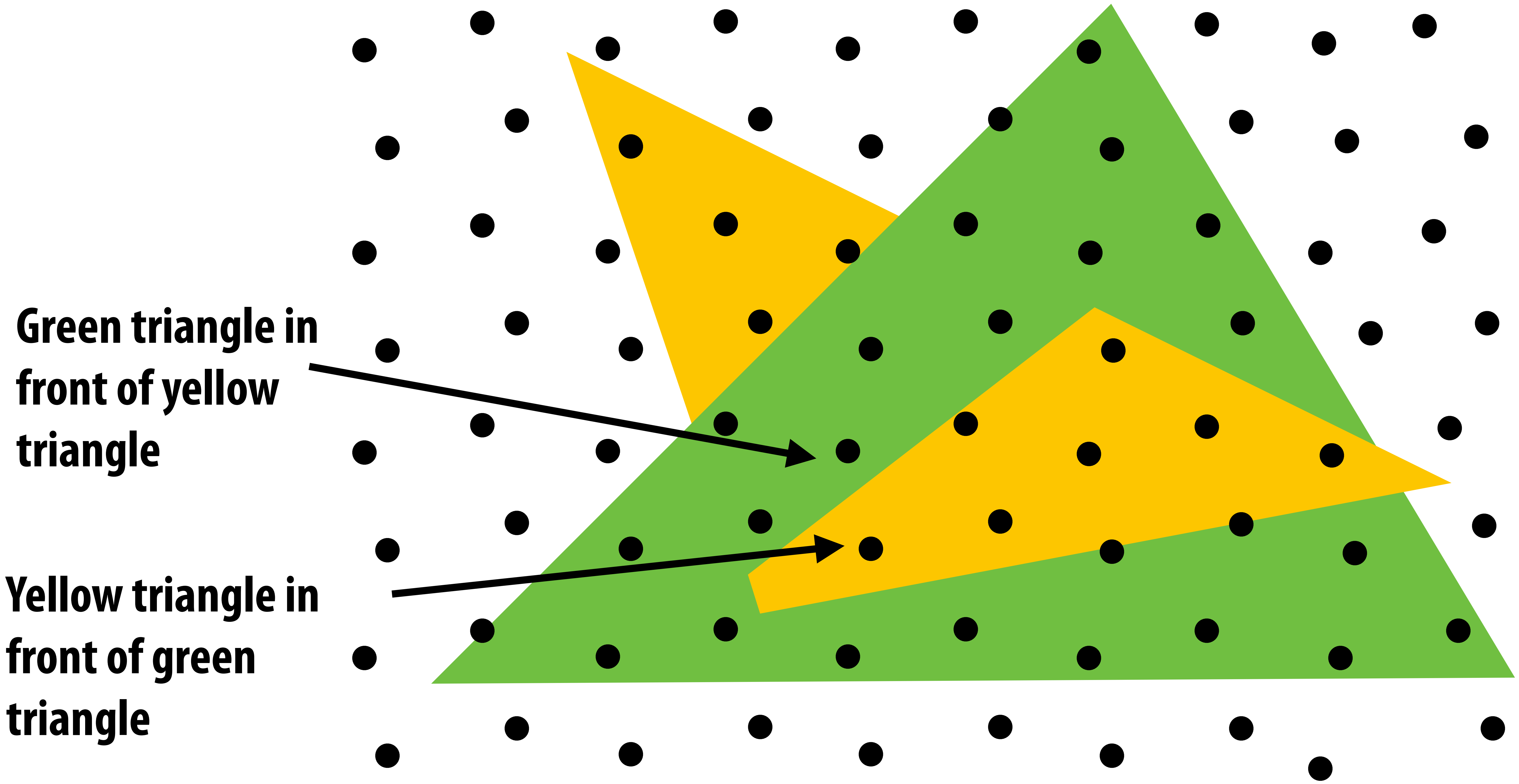
# Occlusion using the depth buffer (opaque surfaces)

```
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]) {

    // 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!**

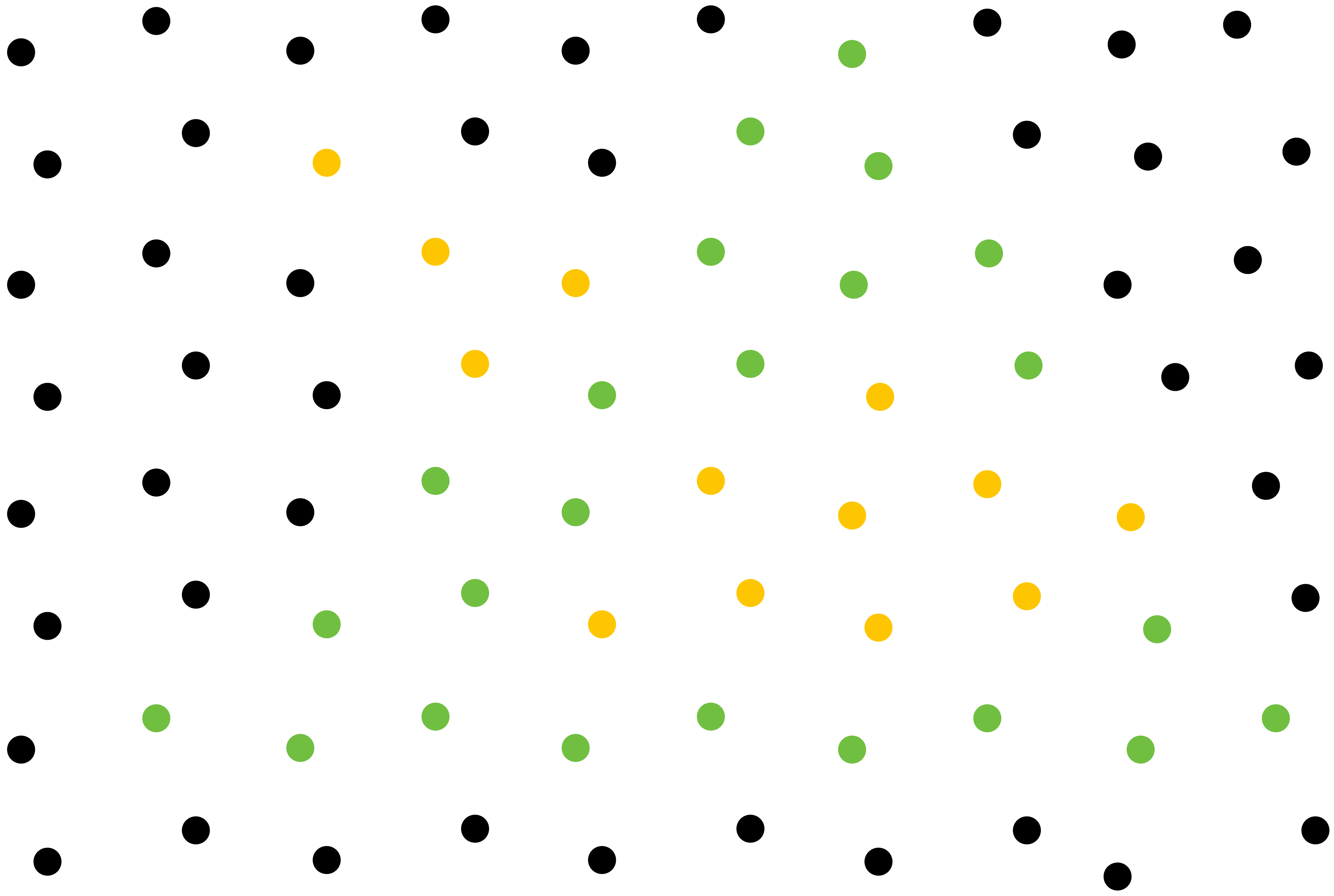
**Occlusion test is based on depth of triangles *at a given sample point*. The relative depth of triangles may be different at different sample points.**



# Does depth-buffer algorithm handle interpenetrating surfaces?

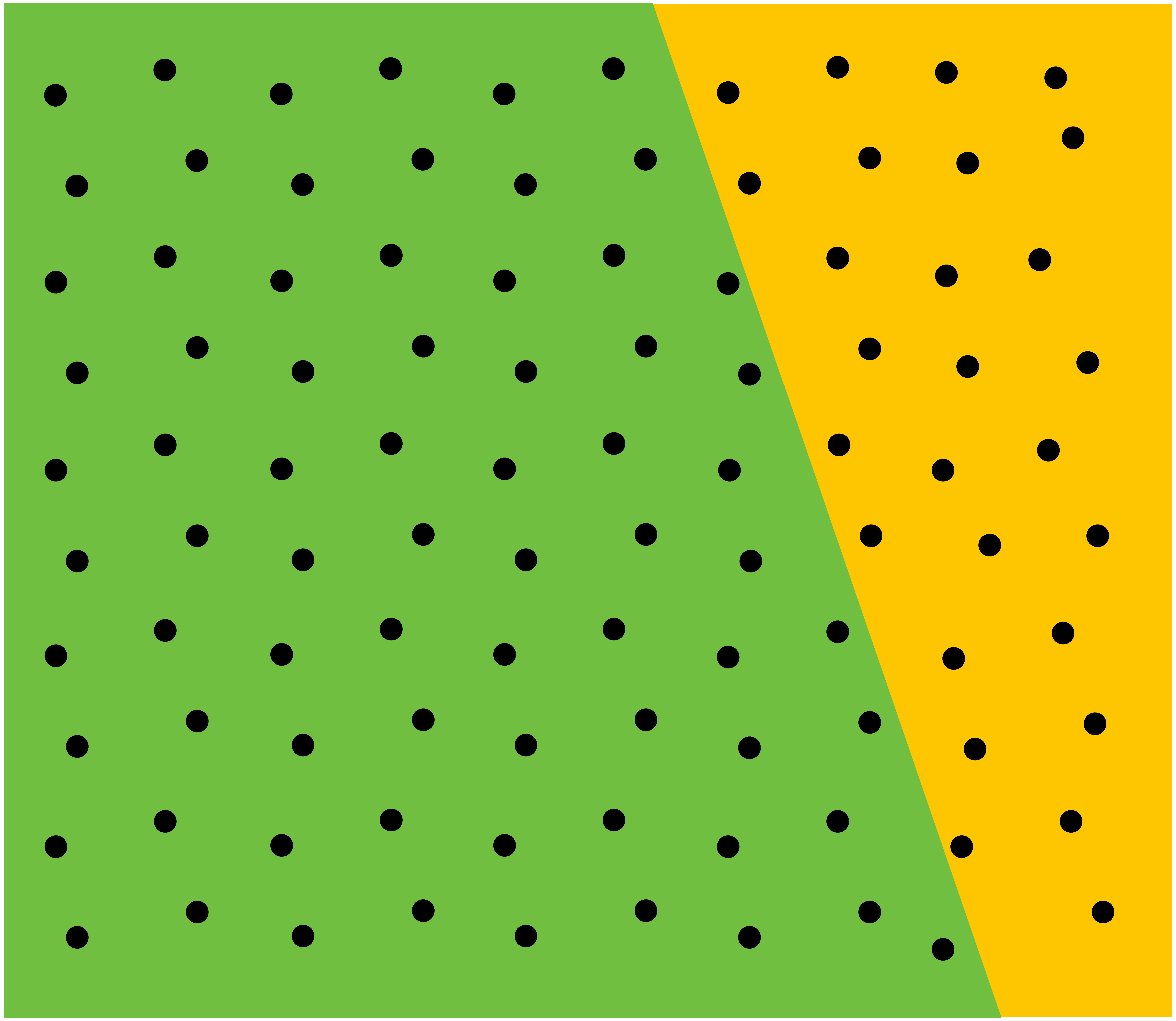
Of course!

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.



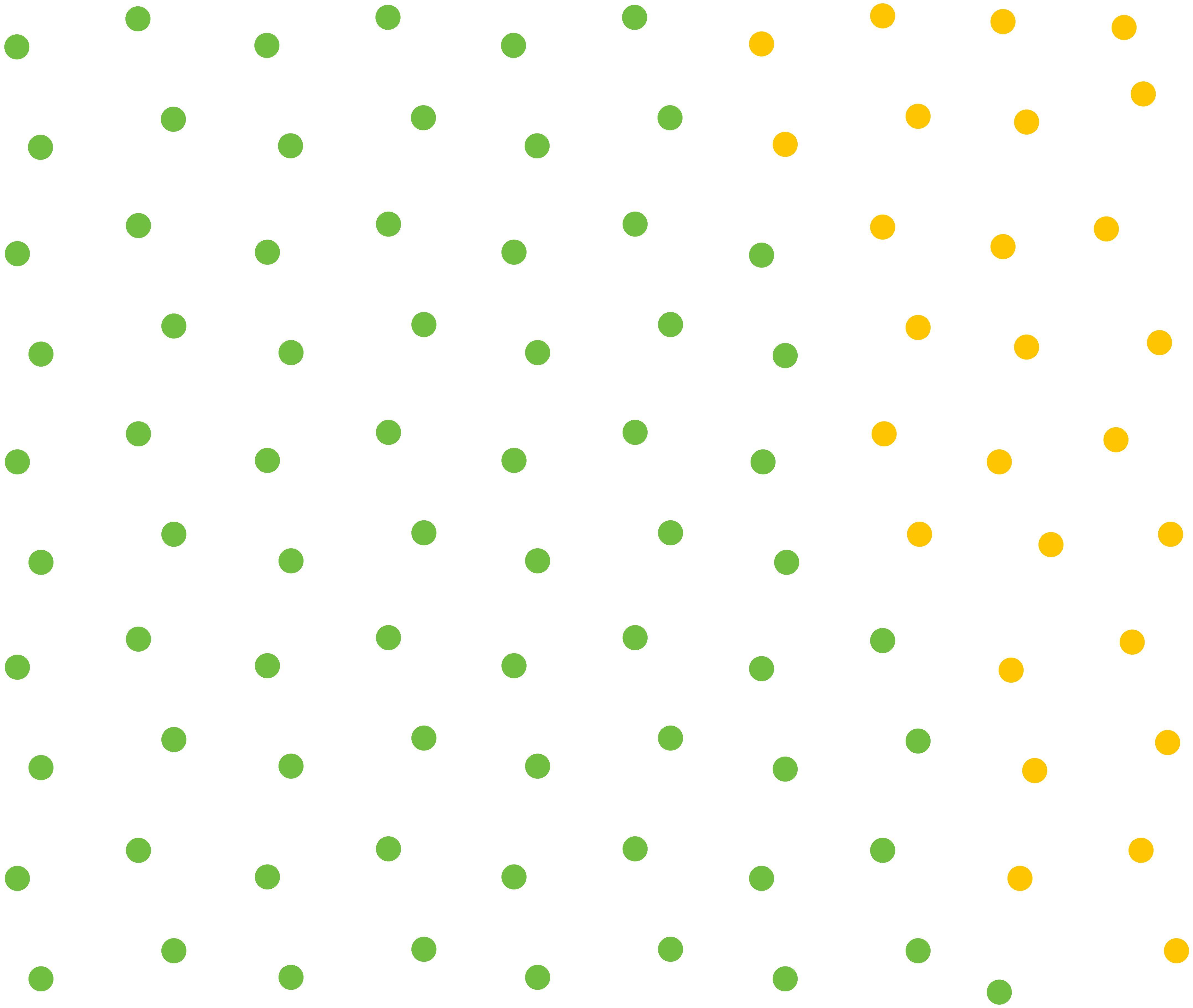
# Does depth buffer work with super sampling?

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

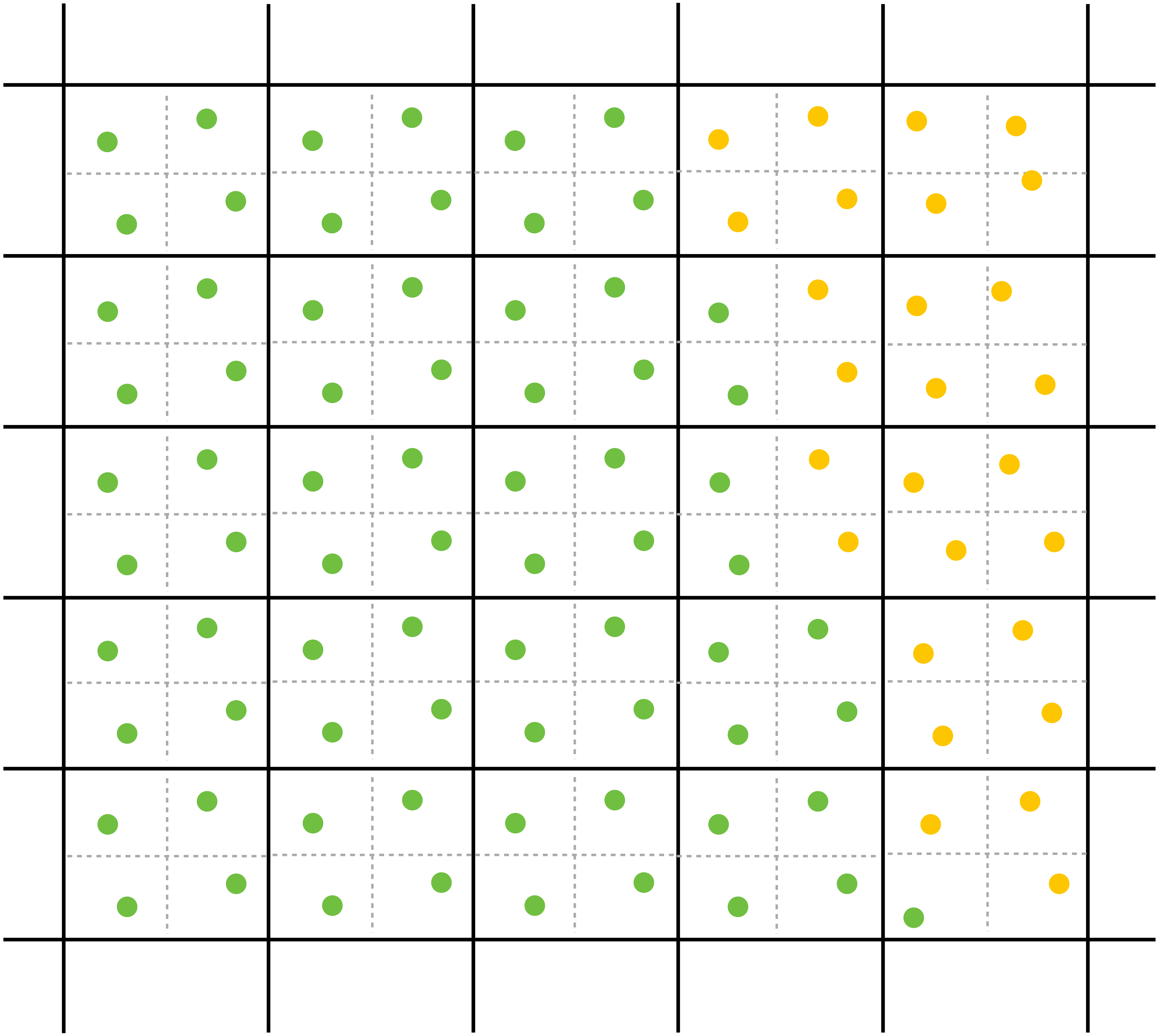


This example: green triangle occludes yellow triangle

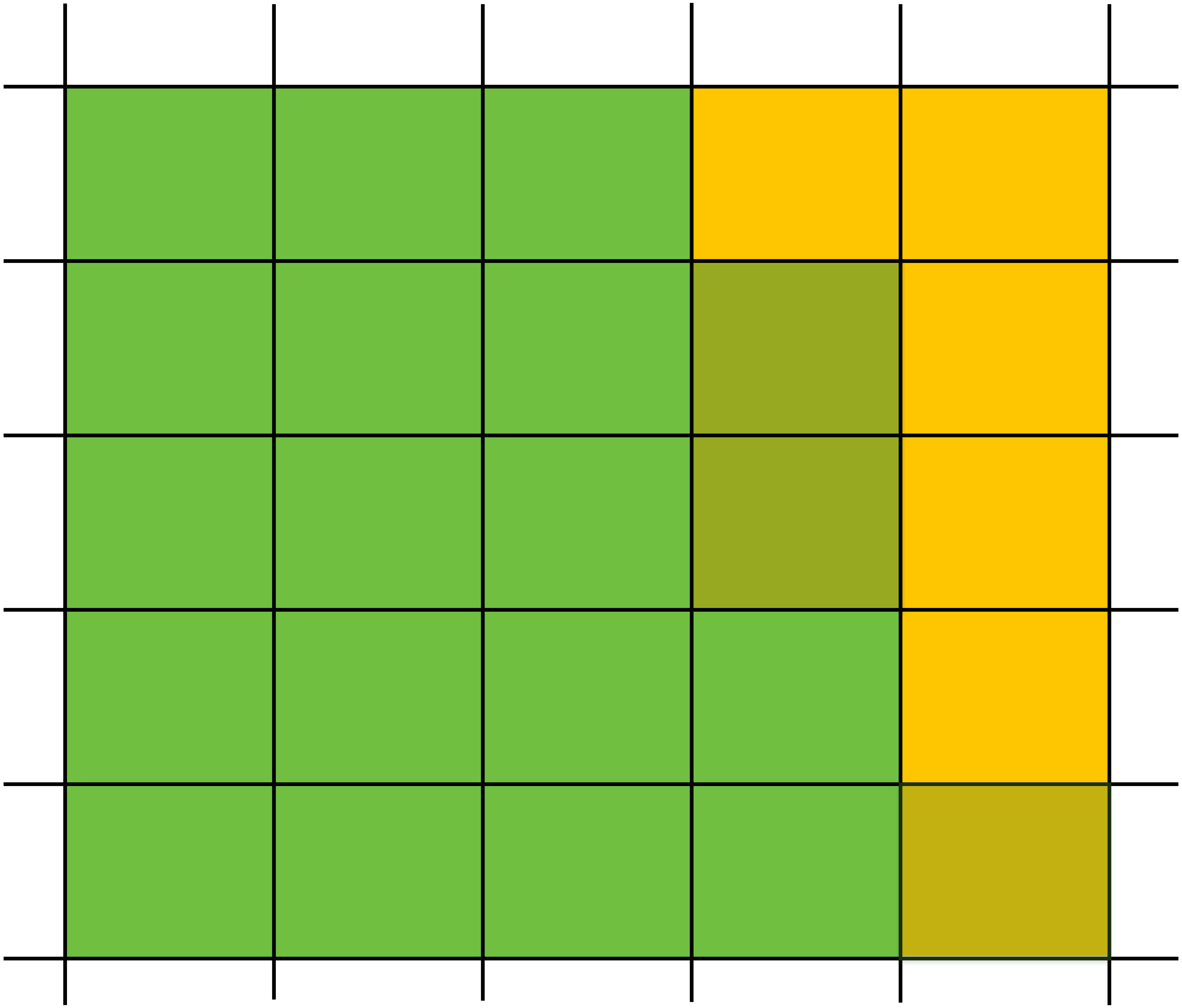
# Color buffer contents



# 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?**

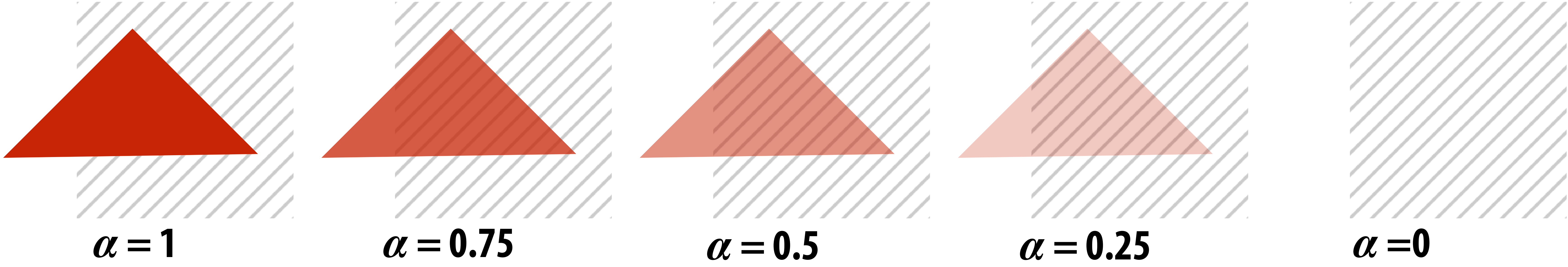
# Compositing

# 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**



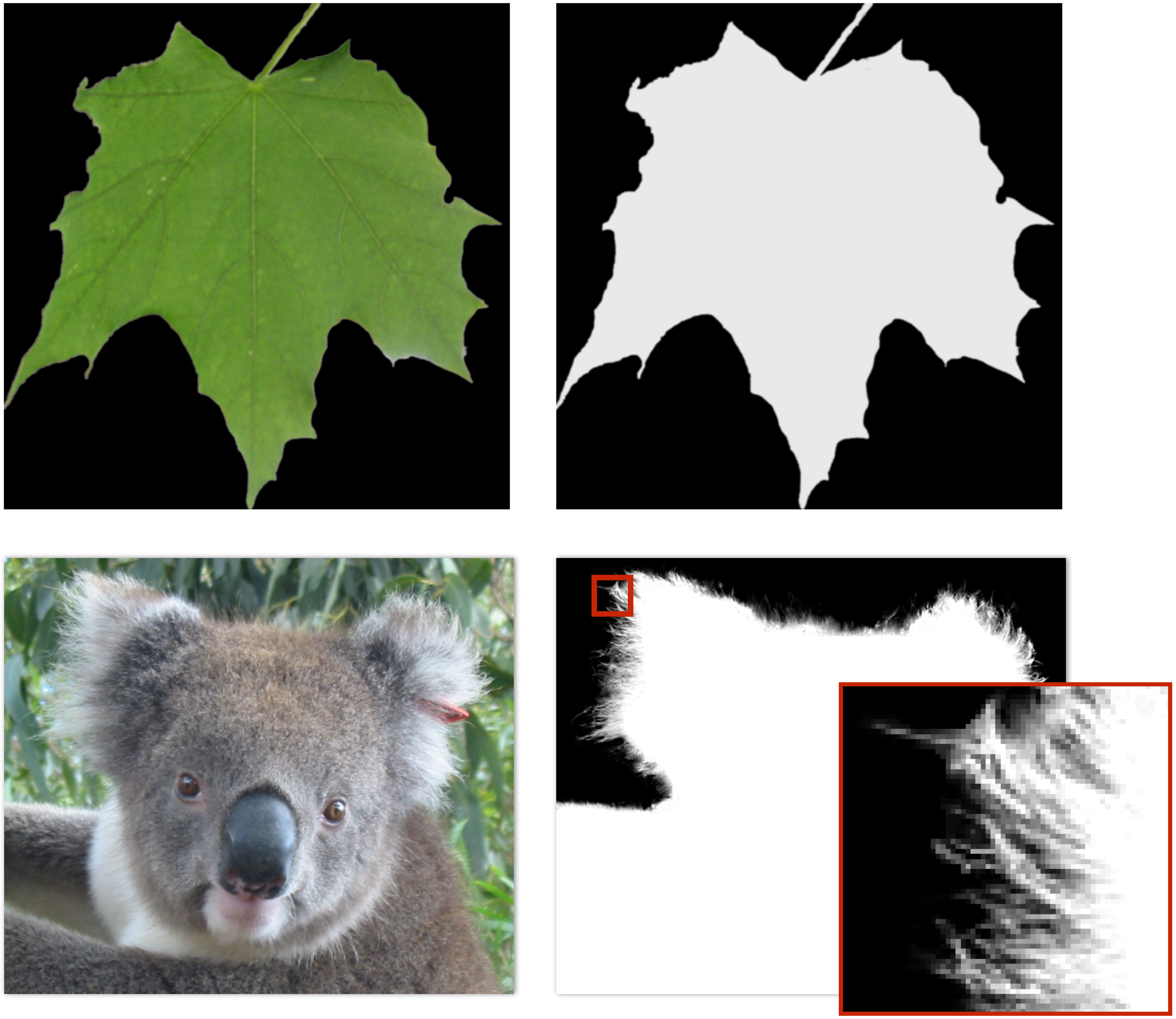
# Alpha: coverage analogy

- 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)

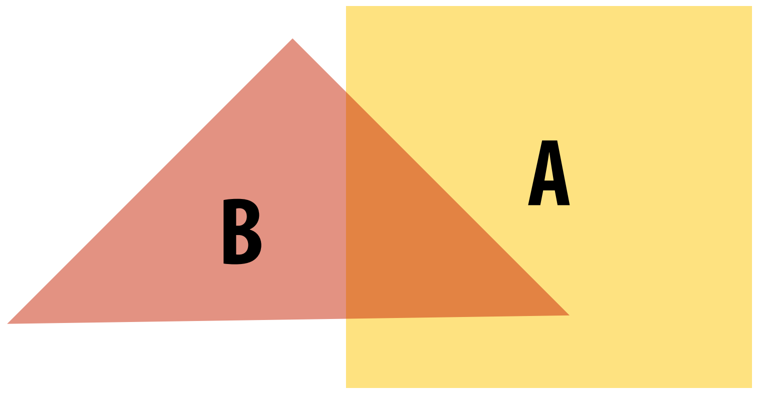
# Alpha: additional channel of image (rgba)



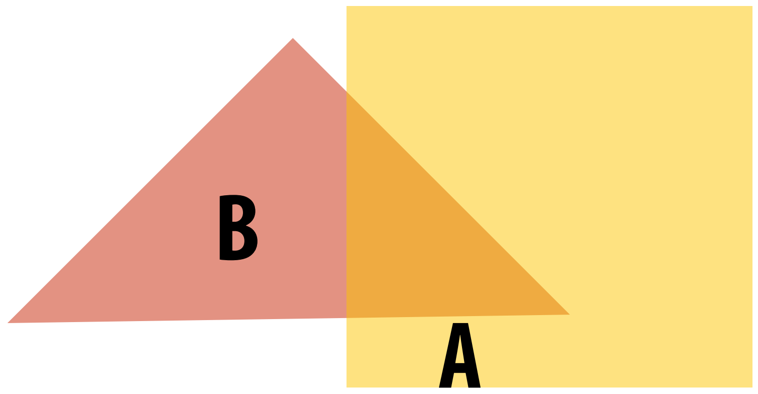
$\alpha$ of foreground object

# Over operator:

**Composite image B with opacity $\alpha_B$ over image A with opacity $\alpha_A$**



B over A



A over B

A over B != B over A

“Over” is not commutative

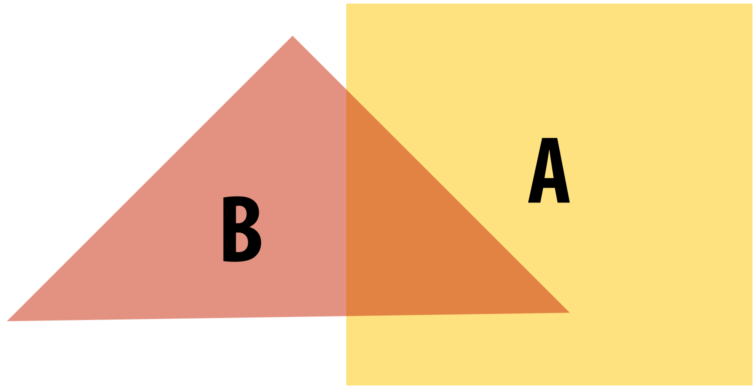
Koala over NYC

# Over operator: non-premultiplied alpha

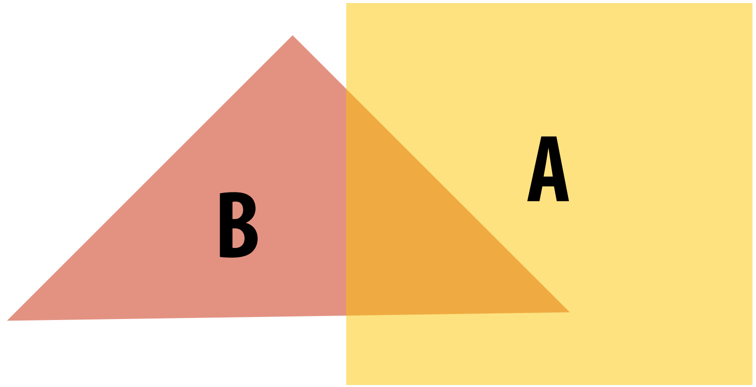
**Composite image B with opacity $\alpha_B$ over image A with opacity $\alpha_A$**

**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



A over B

A over B != B over A

"Over" is not commutative

**Composited color:**

Appearance of semi-transparent A

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

Appearance of semi-transparent B

What B lets through

**Composite alpha:**

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

# Premultiplied alpha

- **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**

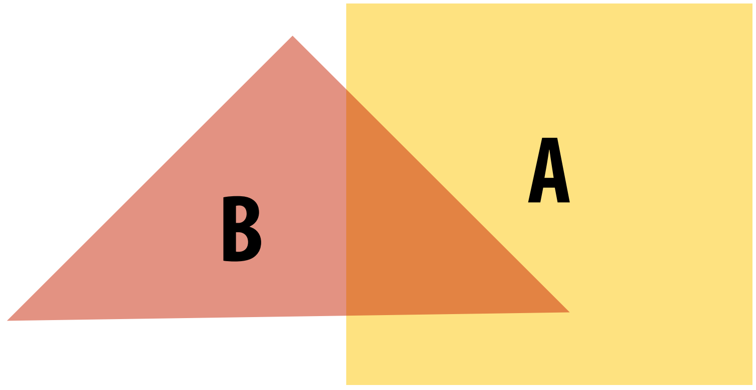
**[0.5, 0.0, 0.0, 0.5]**

**Example: 75% opaque magenta**

**[0.75, 0.0, 0.75, 0.75]**

# Over operator: using premultiplied alpha

**Composite image B with opacity $\alpha_B$ over image A with opacity $\alpha_A$**



B over A

## Non-premultiplied alpha representation:

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

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

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

← two multiplies, one add (referring to vector ops on colors)

## Composite alpha:

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

## Premultiplied alpha representation:

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

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

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

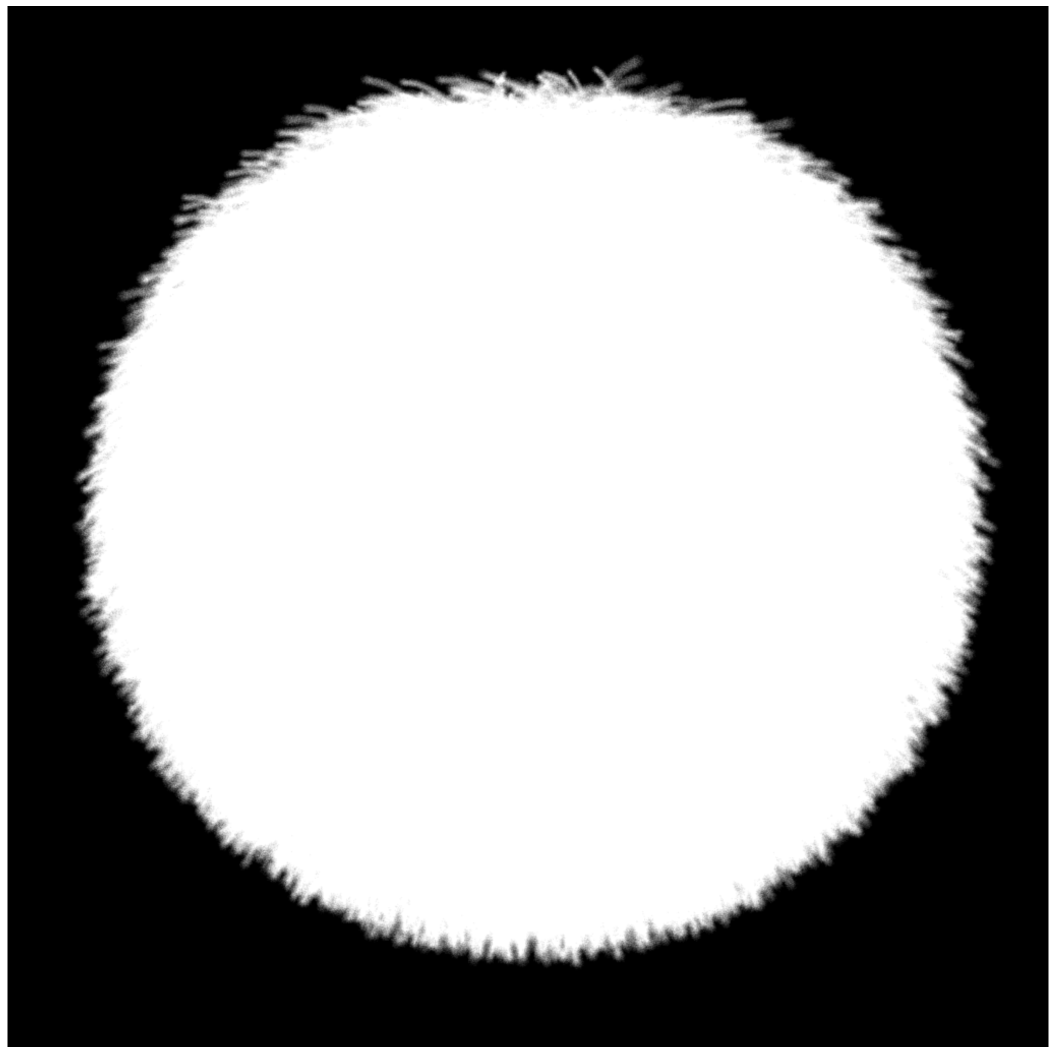
← one multiply, one add

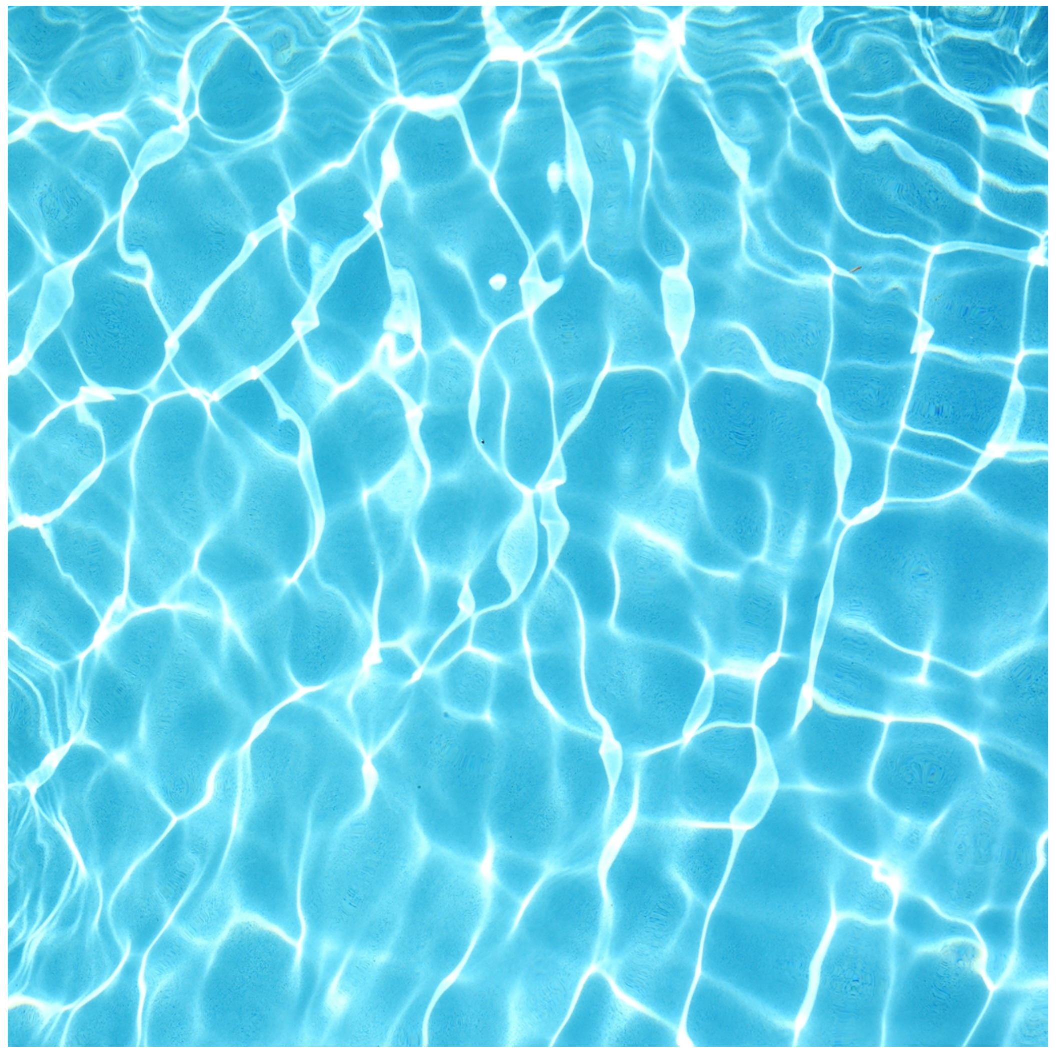
Notice premultiplied alpha composites alpha just like how it composites rgb.

# Fringing

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


foreground color


foreground alpha


background color


fringing

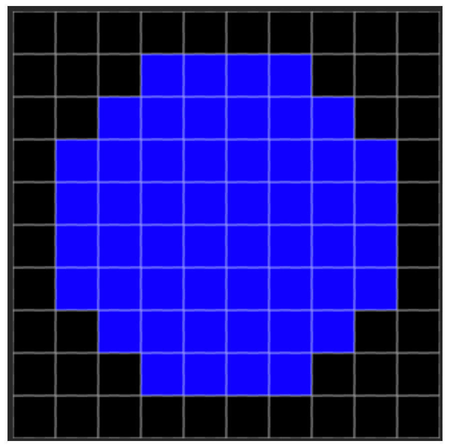

no fringing

# No fringing

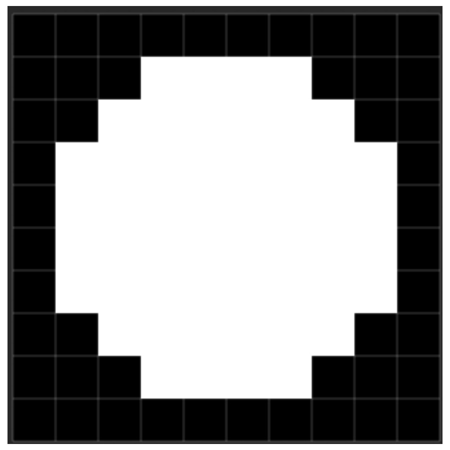
# Fringing (…why does this happen?)

# A problem with non-premultiplied alpha

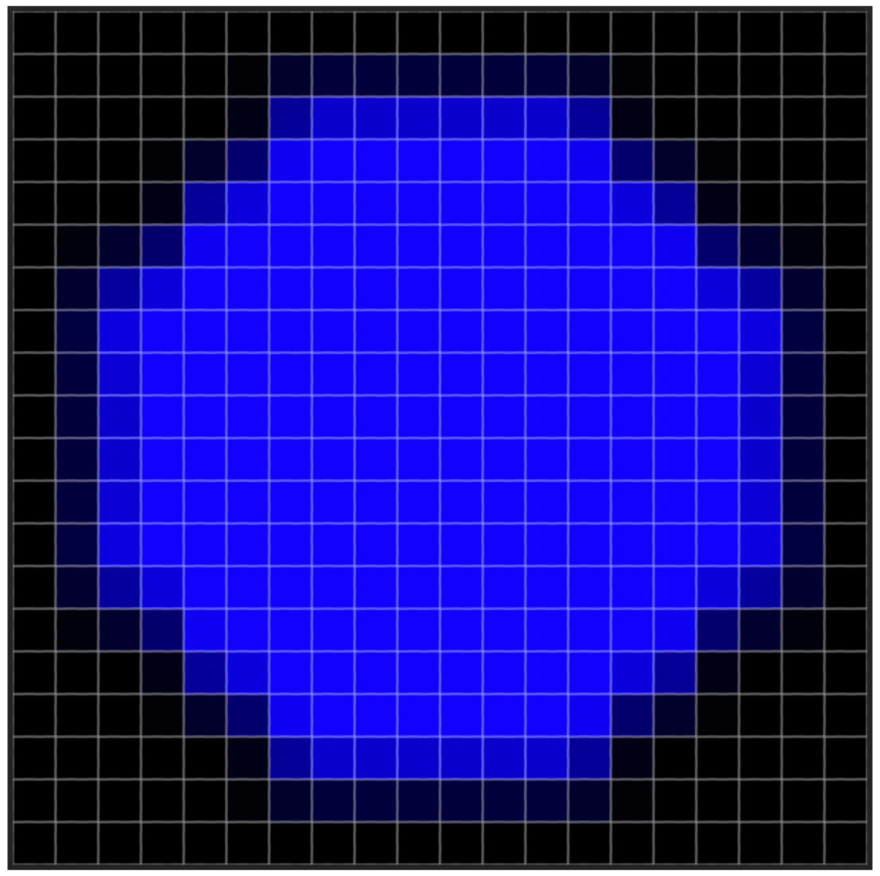
- Suppose we upsample an image w/ an alpha mask, then composite it onto a background
- How should we compute the interpolated color/alpha values?
- If we interpolate color and alpha separately, then blend using the non-premultiplied "over" operator, here's what happens:



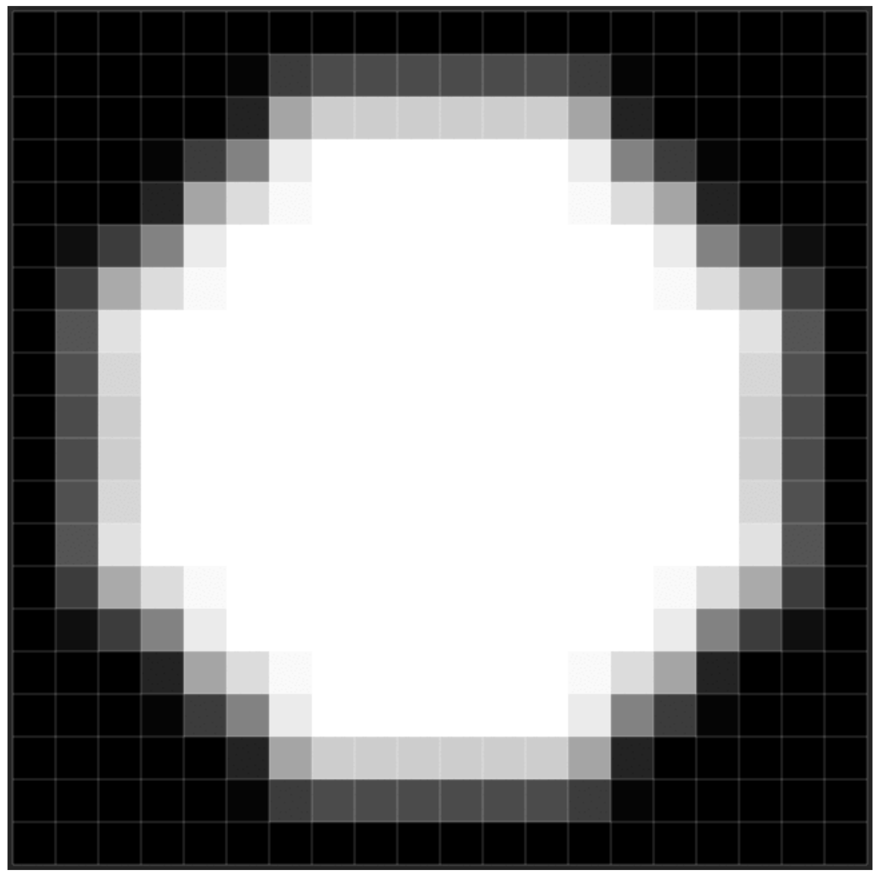
original color



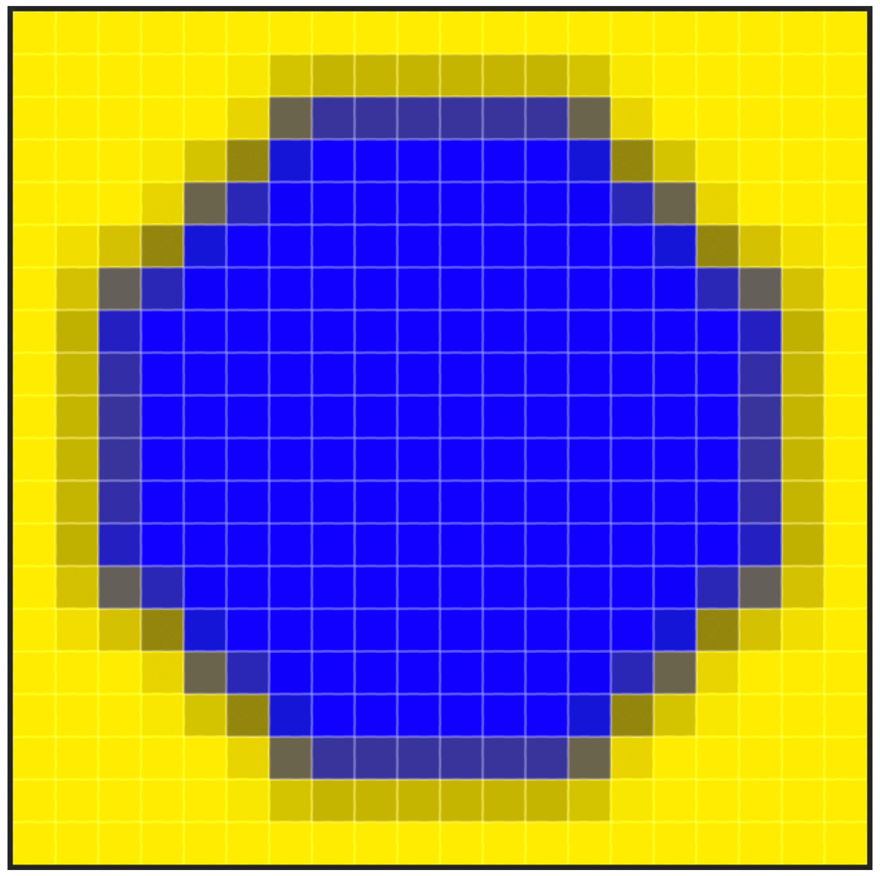
original alpha



upsampled color



upsampled alpha



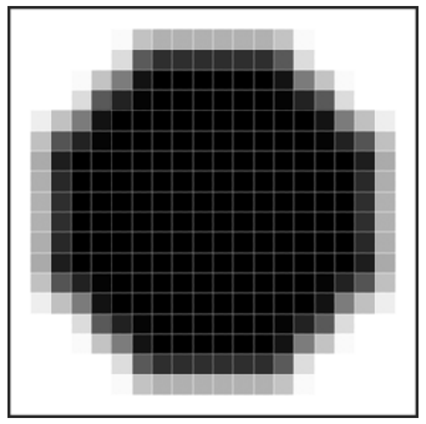
composited onto yellow background

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.

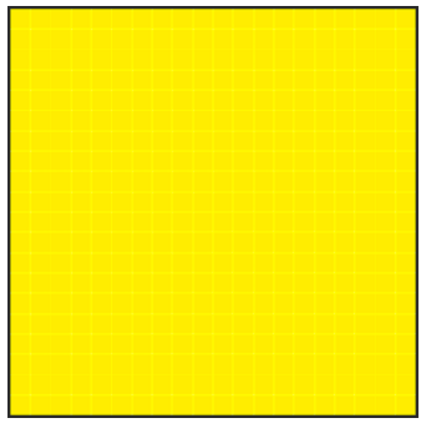
# Eliminating fringe w/ premultiplied “over”

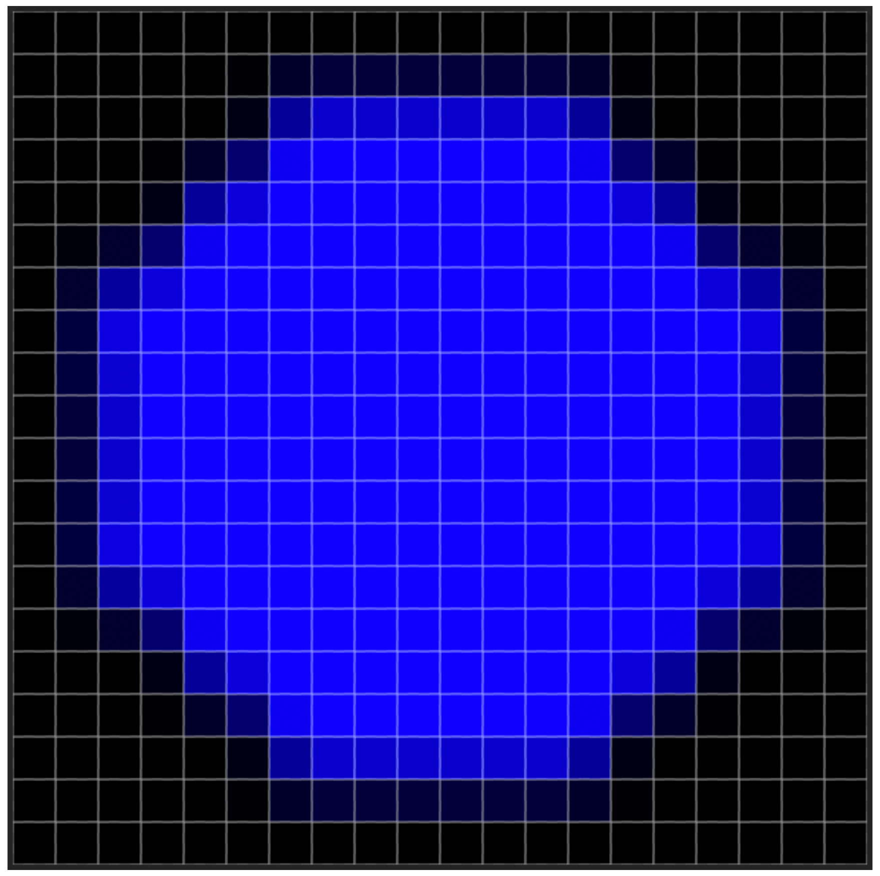
**If we instead use the premultiplied “over” operation, we get the correct alpha:**

(1-alpha)



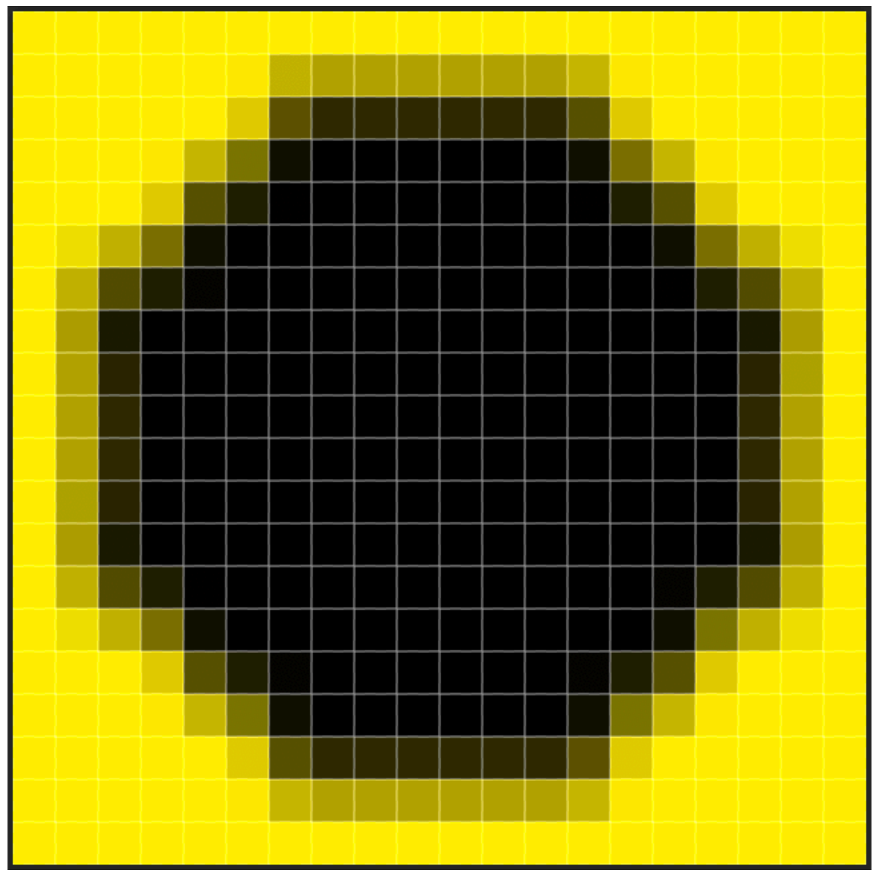
background





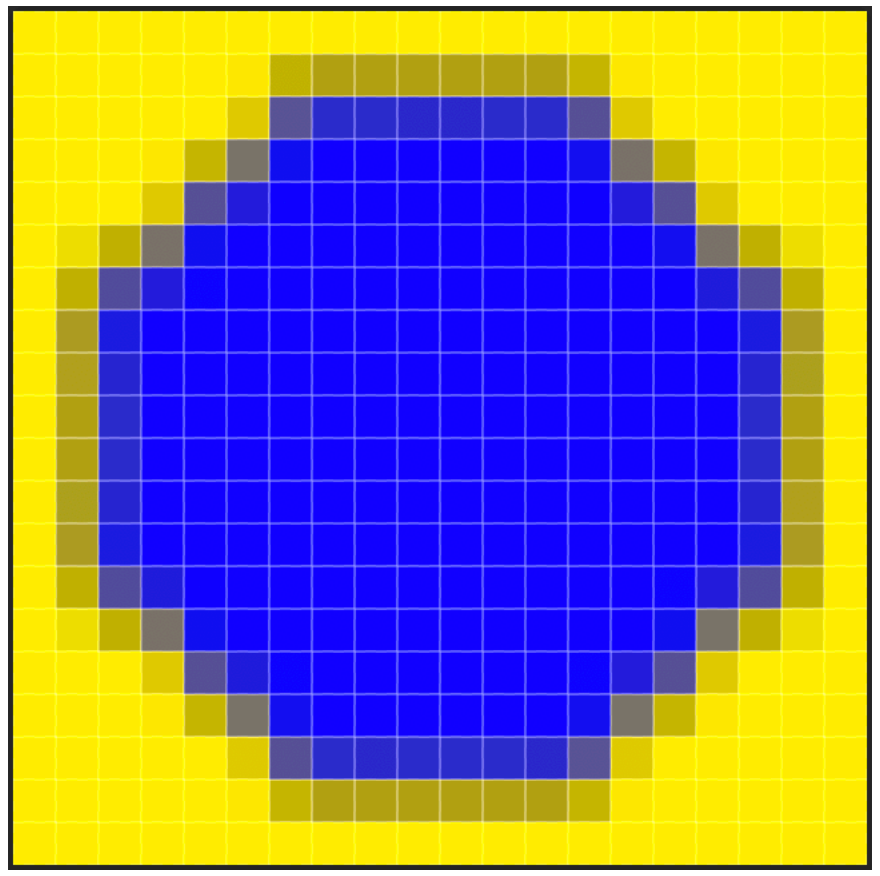
upsampled color

+



(1-alpha)*background

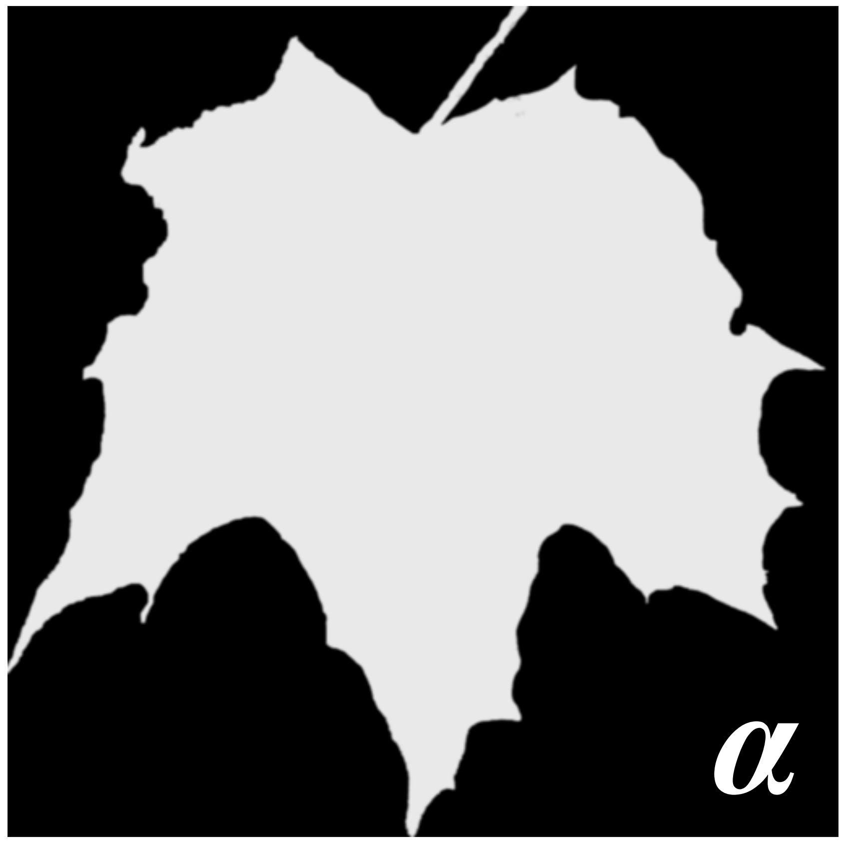
=



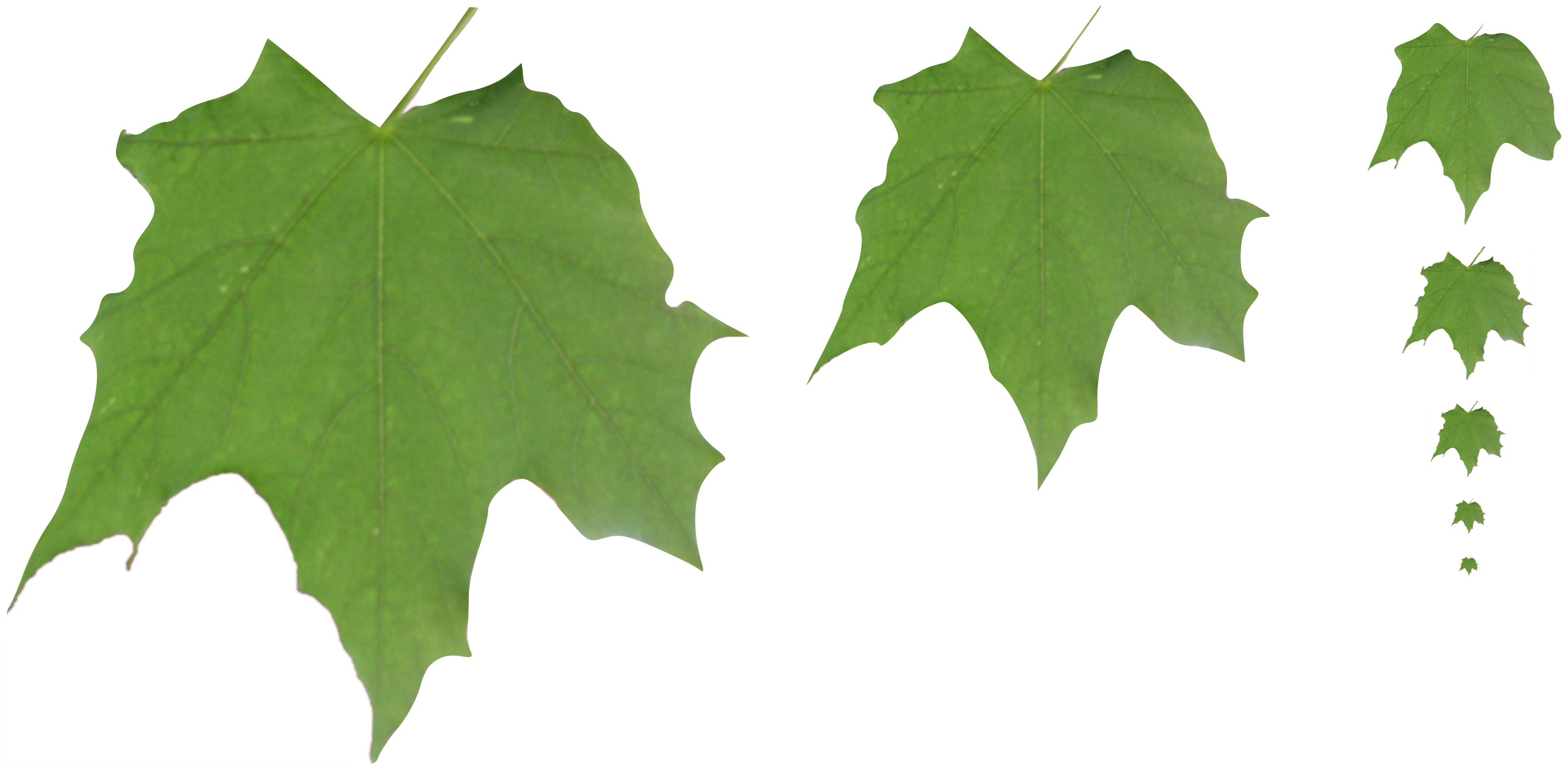
composite image
w/ no fringe

# Another problem with non-premultiplied alpha

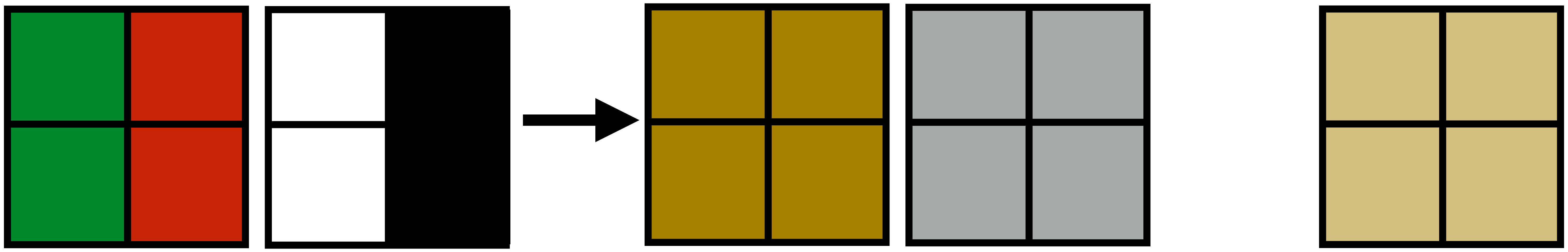
## Consider pre-filtering a texture with an alpha matte



**Desired filtered result**



---



input color

input $\alpha$

filtered color

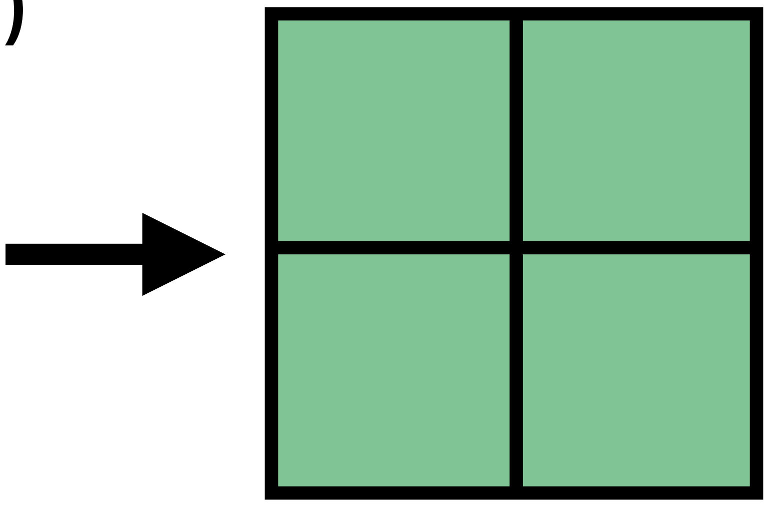
filtered $\alpha$

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

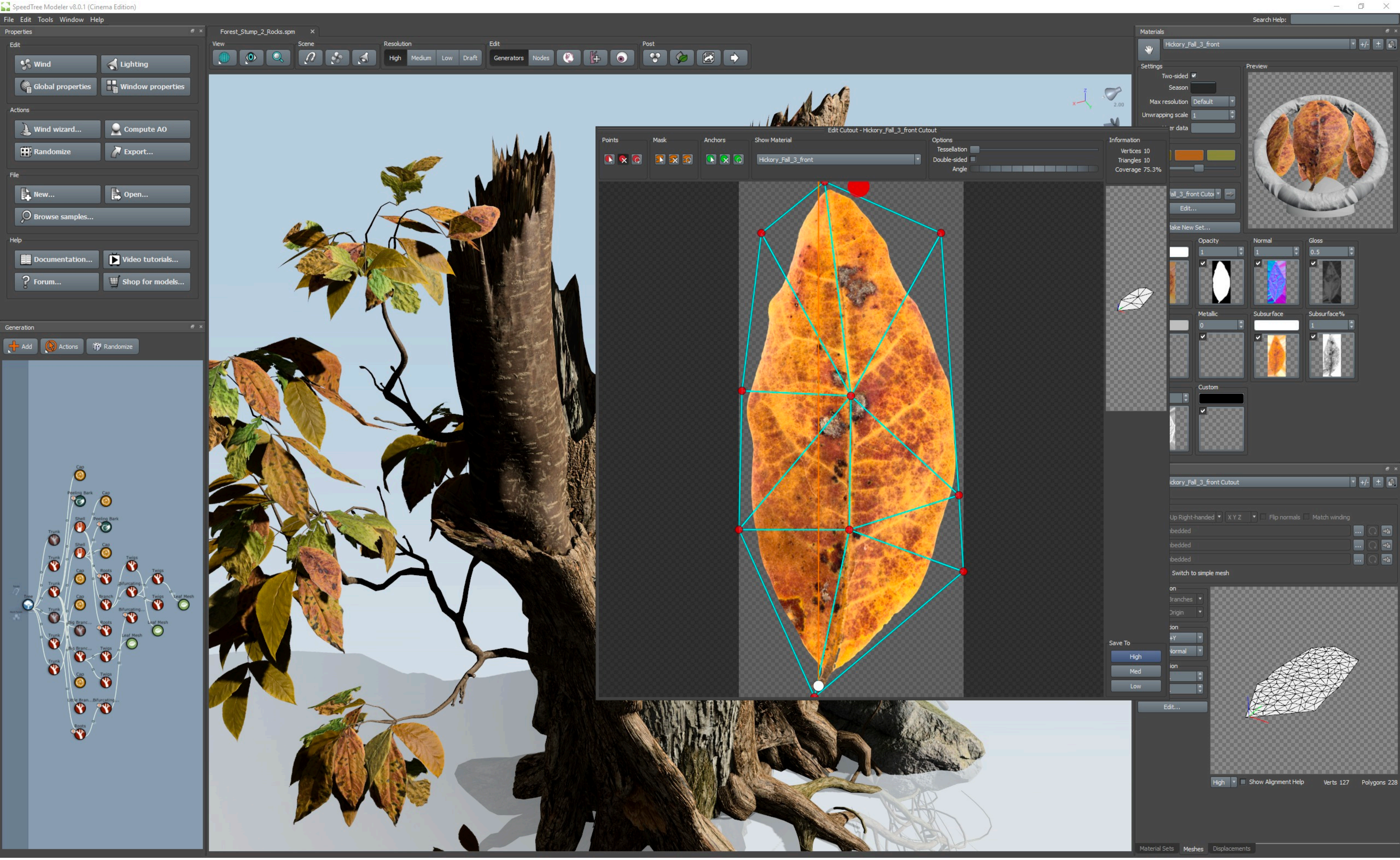
filtered result composited over white

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

Result of filtering premultiplied image



# Common use of textures with alpha: foliage



Show Material: Hickory_Fall_3_front

Tessellation

Double-sided

Angle

Vertices 10

Triangles 10

Coverage 75.3%

Two-sided ✓

Max resolution Default

Unwrapping scale 1

Verts 127

Polygons 228

[Image credit: SpeedTree Cinema 8]

# Foliage example

[Image credit: SpeedTree Cinema 8]

# Another problem: applying "over" repeatedly

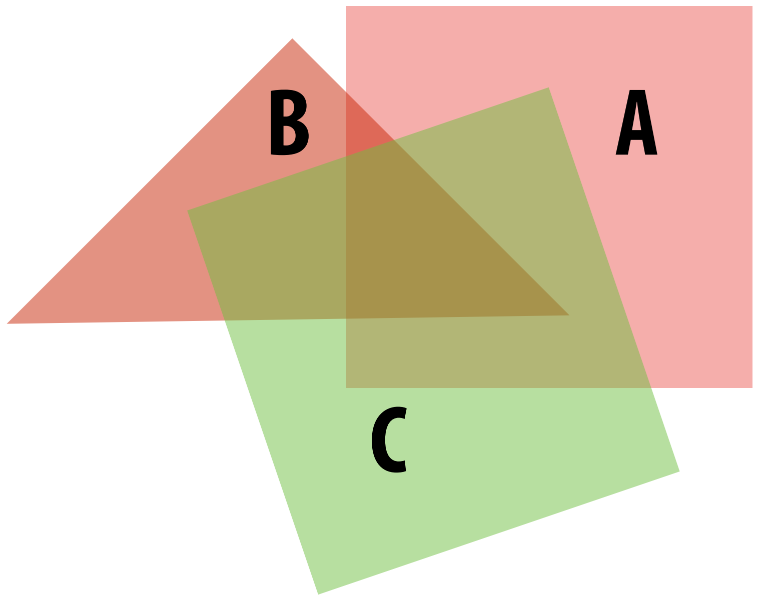
**Consider composite image C with opacity $\alpha_C$ over B with opacity $\alpha_B$ over image A with opacity $\alpha_A$**

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

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

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

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



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 *

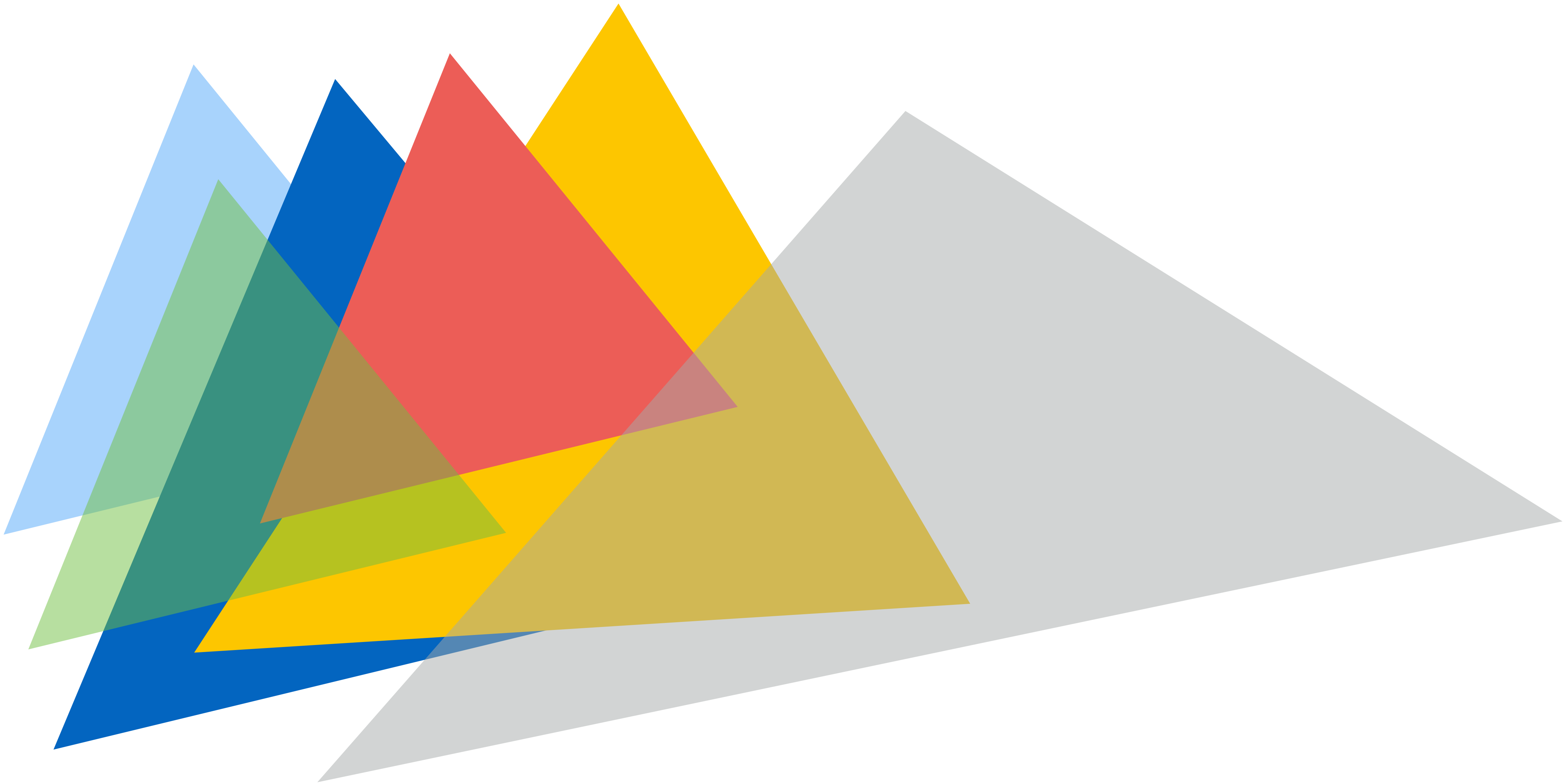
## Consider rendering a mixture of opaque and transparent triangles

**Step 1: render opaque surfaces using depth-buffered occlusion**

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

**Step 2: disable 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 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]);
  }
}
```

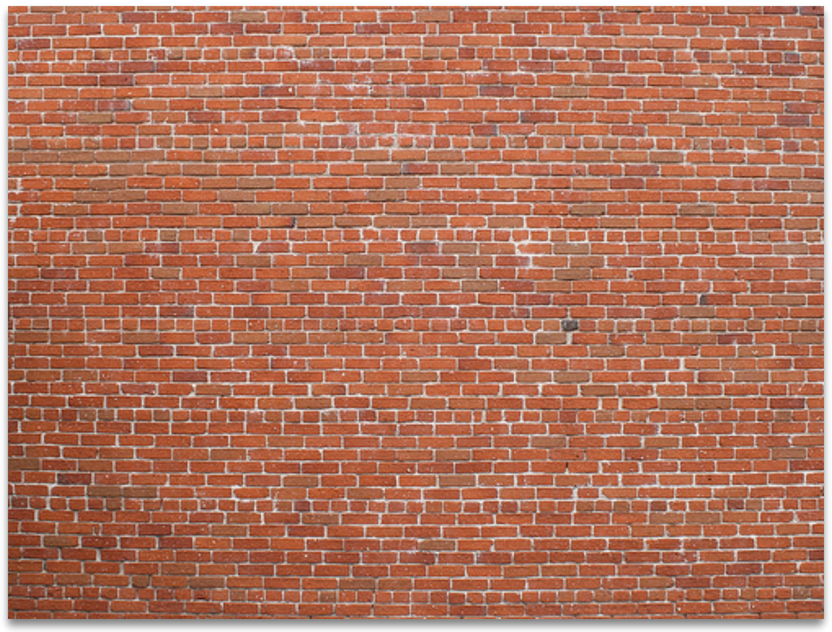
# End-to-end rasterization pipeline (“real-time graphics pipeline”)

# Command: draw these triangles!

## Inputs:

```
list_of_positions = {

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

```
list_of_texcoords = {

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



**Texture map**

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

**Perspective projection transform** $\mathbf{P}$

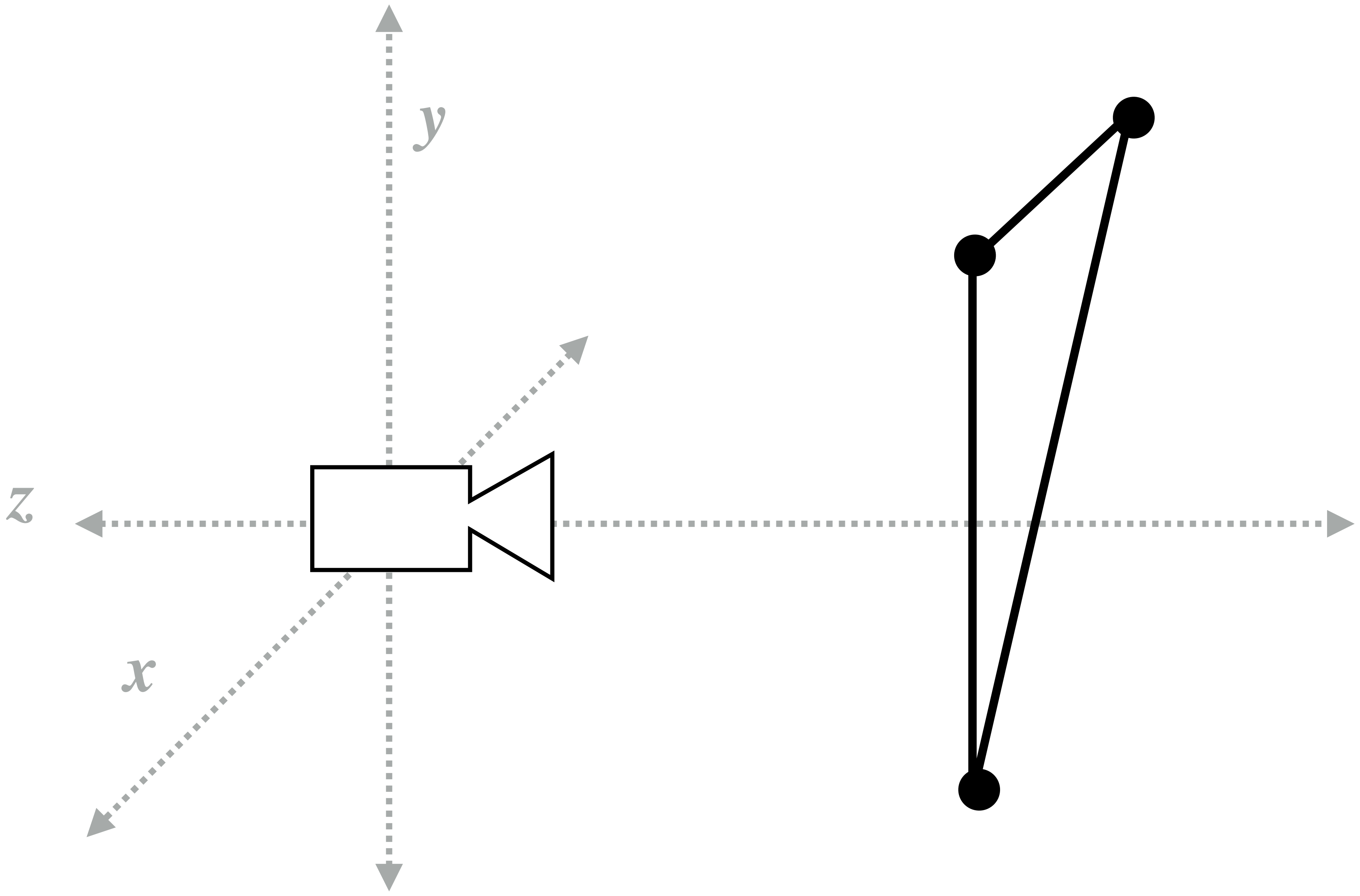
**Size of output image (W, H)**

**Use depth test /update depth buffer: YES!**

# Step 1:

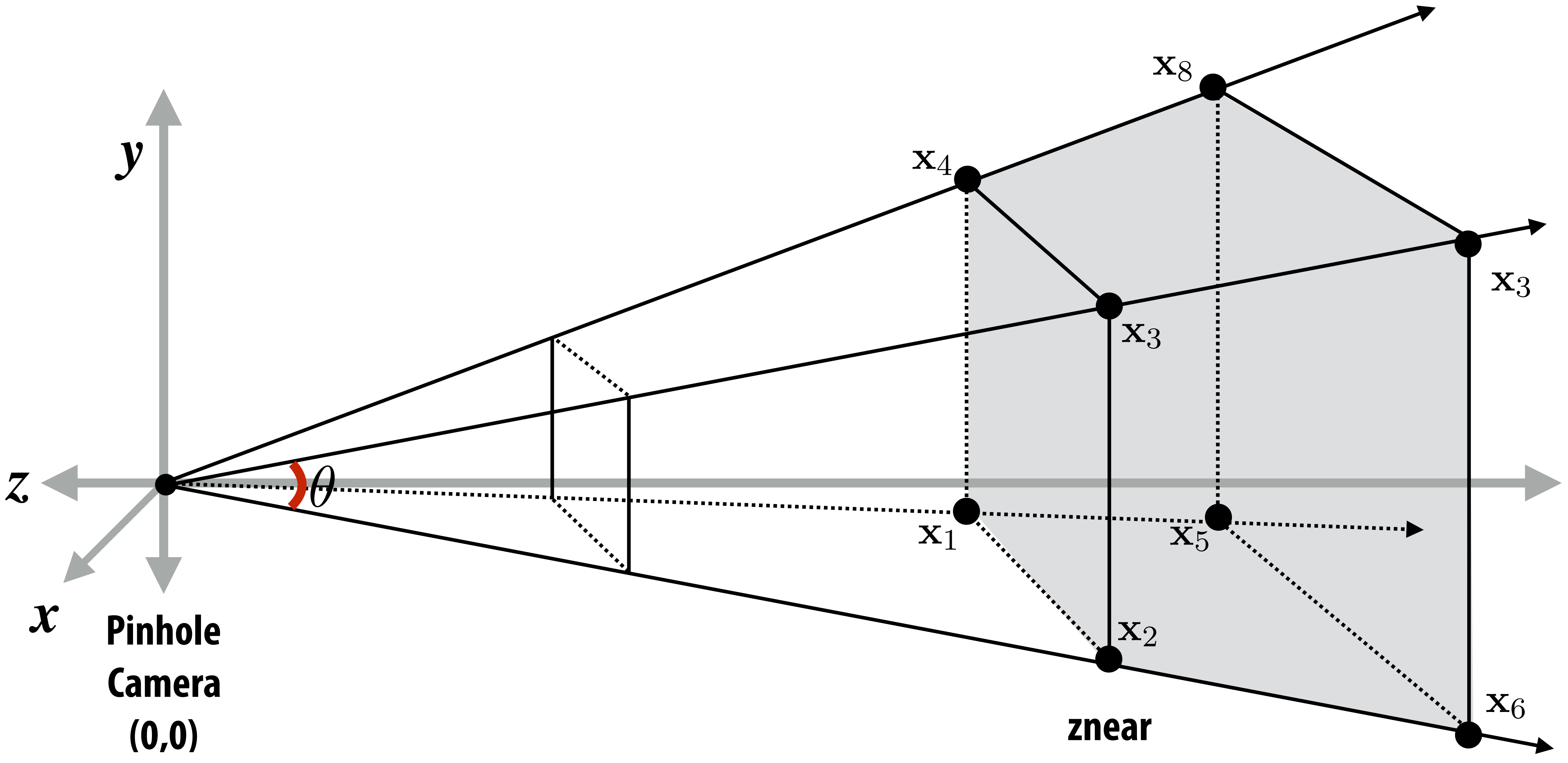
Transform triangle vertices into camera space

(apply modeling and camera transform)

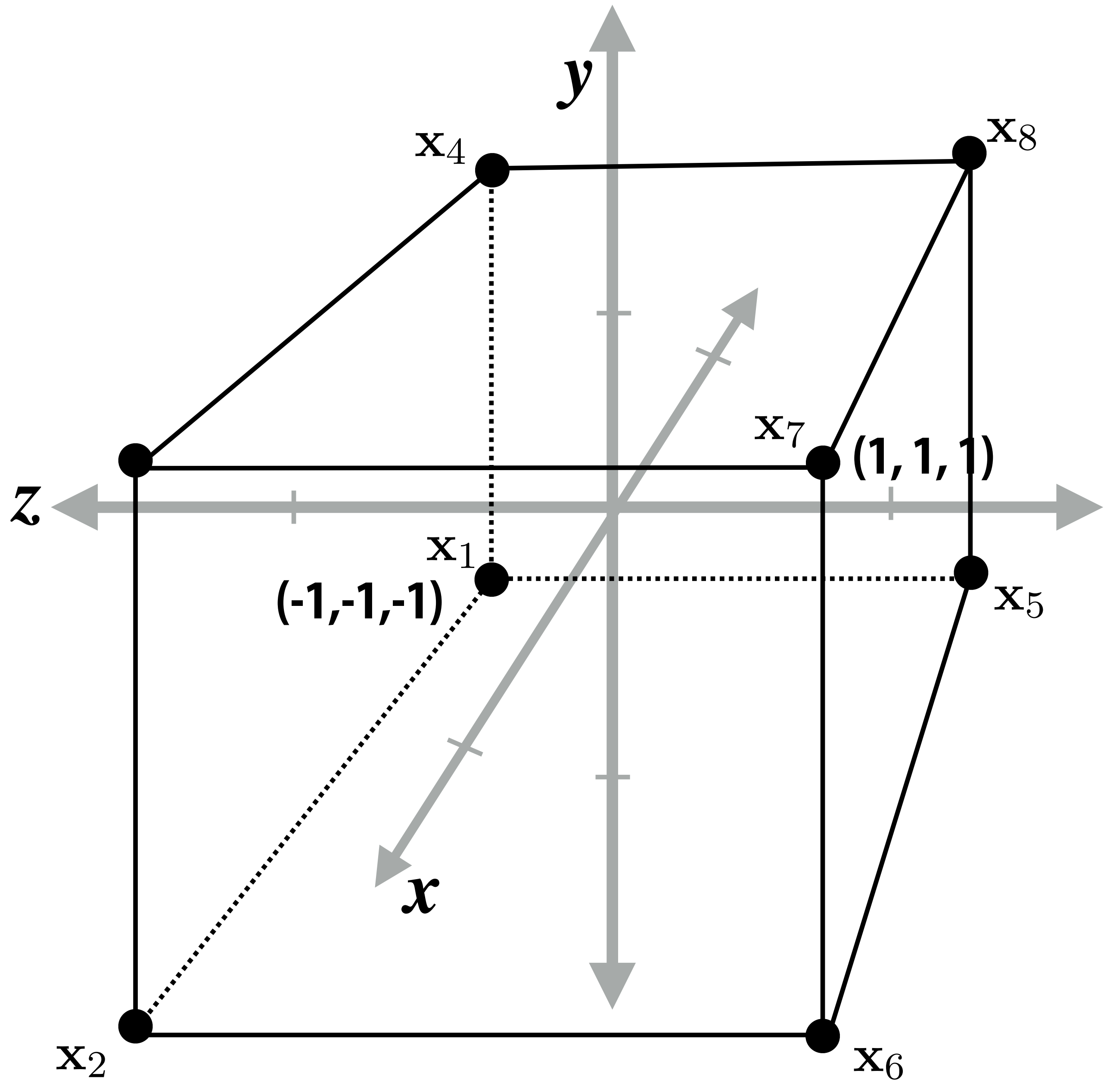


# Step 2:

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



Camera-space positions: 3D



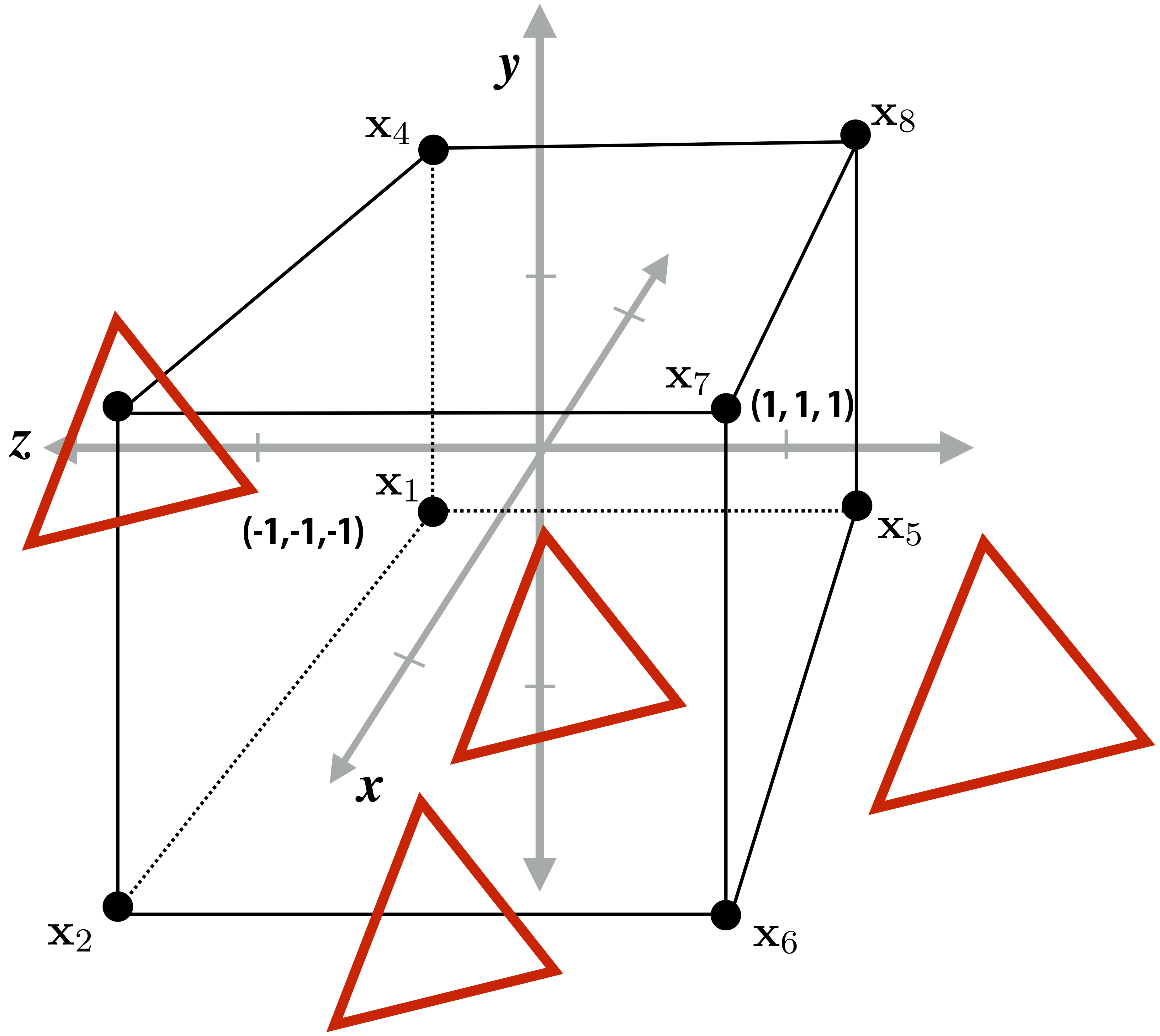
Normalized space positions

**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:**

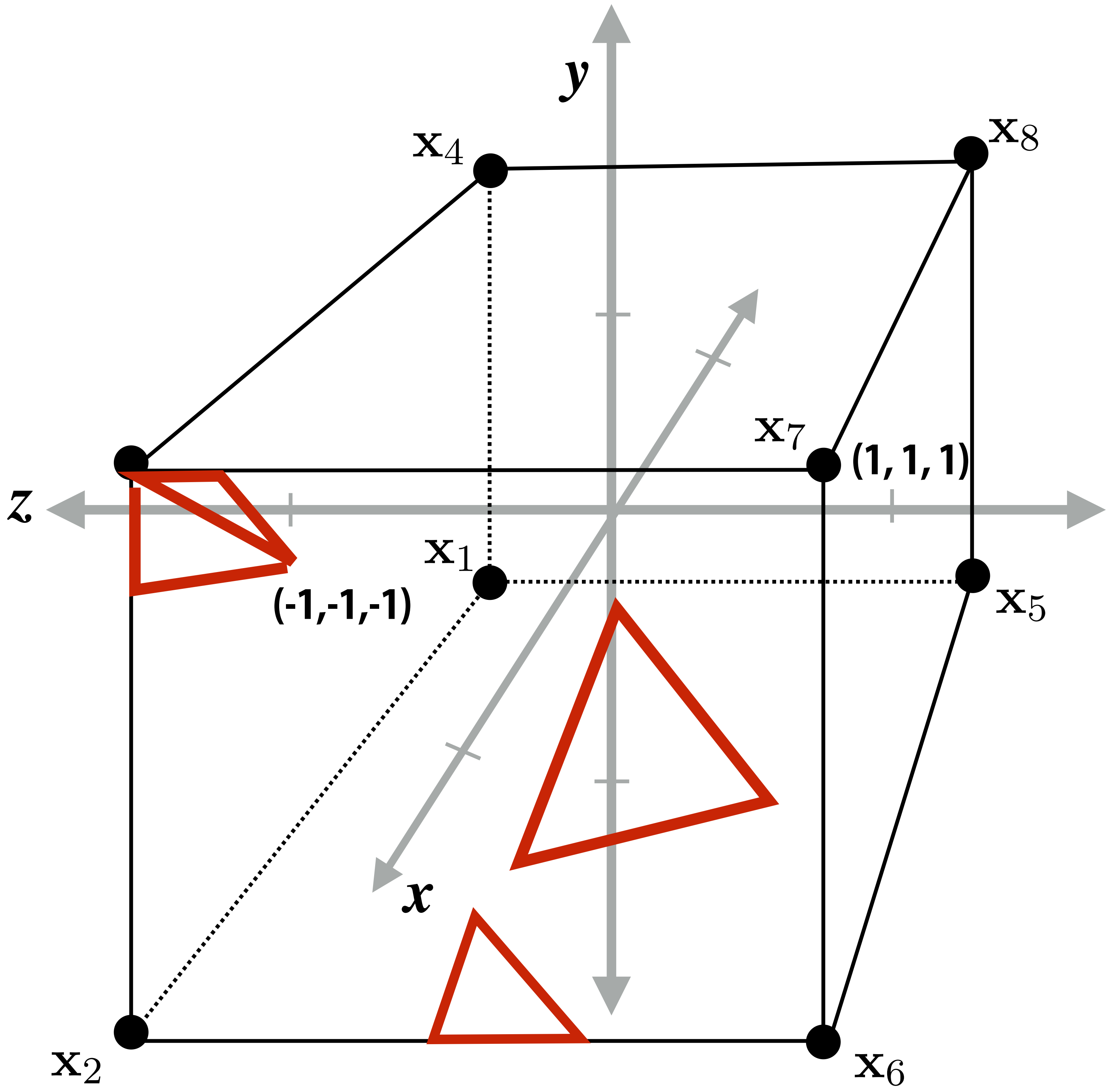
**(-w, -w, -w, w) and (w, w, w, w)**

# 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**



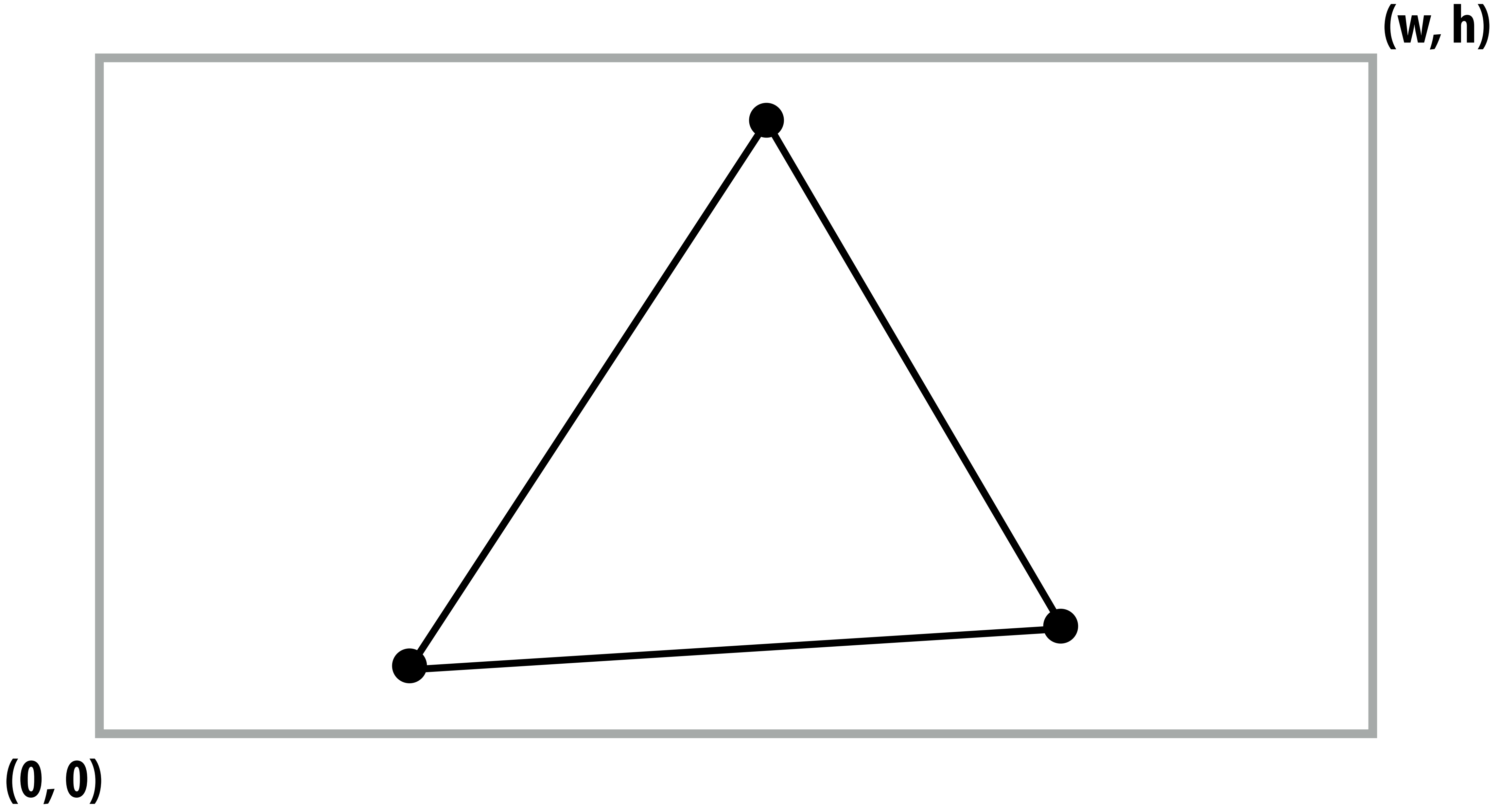
**Triangles before clipping**



**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**

**Compute triangle attribute equations**

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

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

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

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

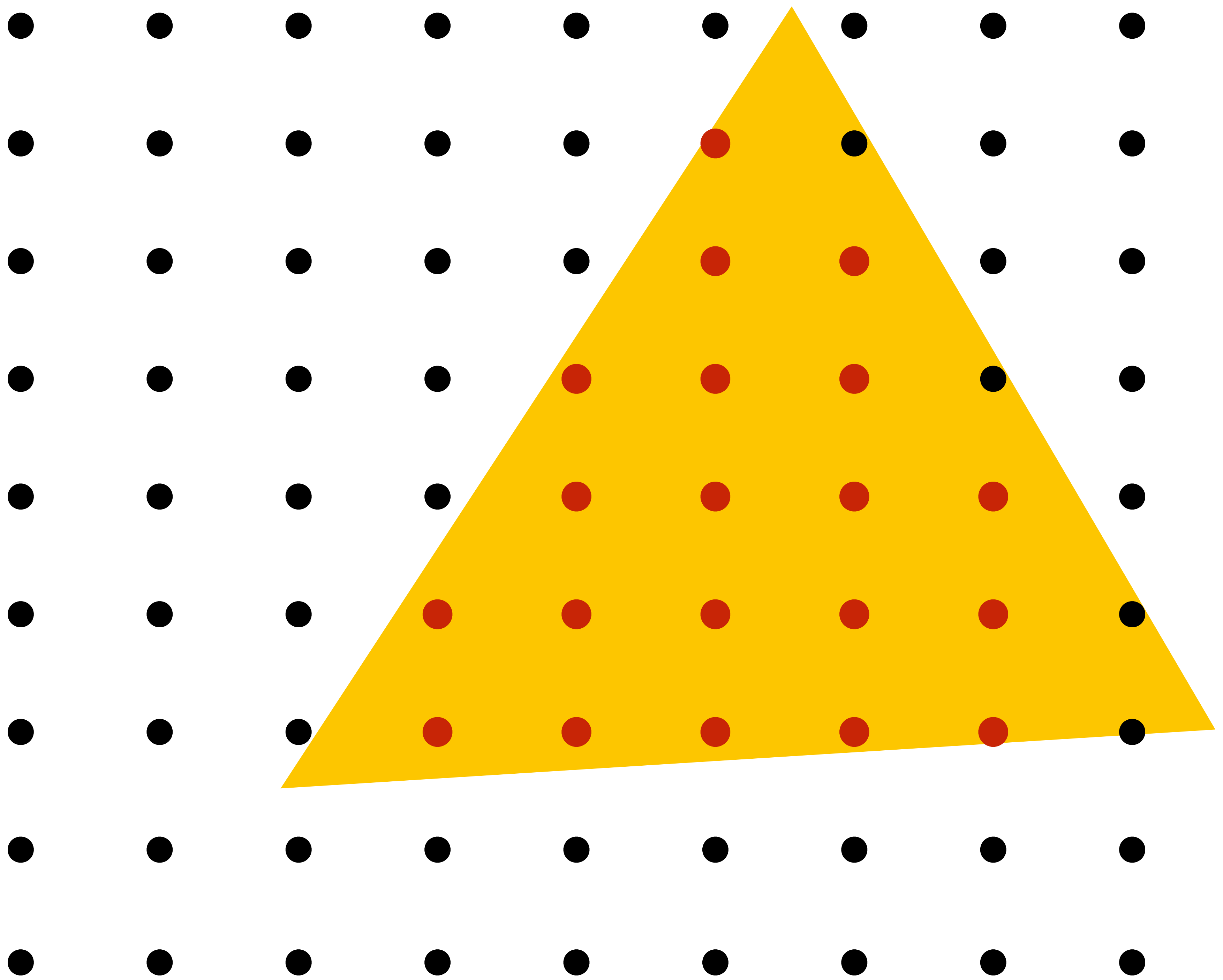
$$\mathbf{Z}(x, y)$$

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

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

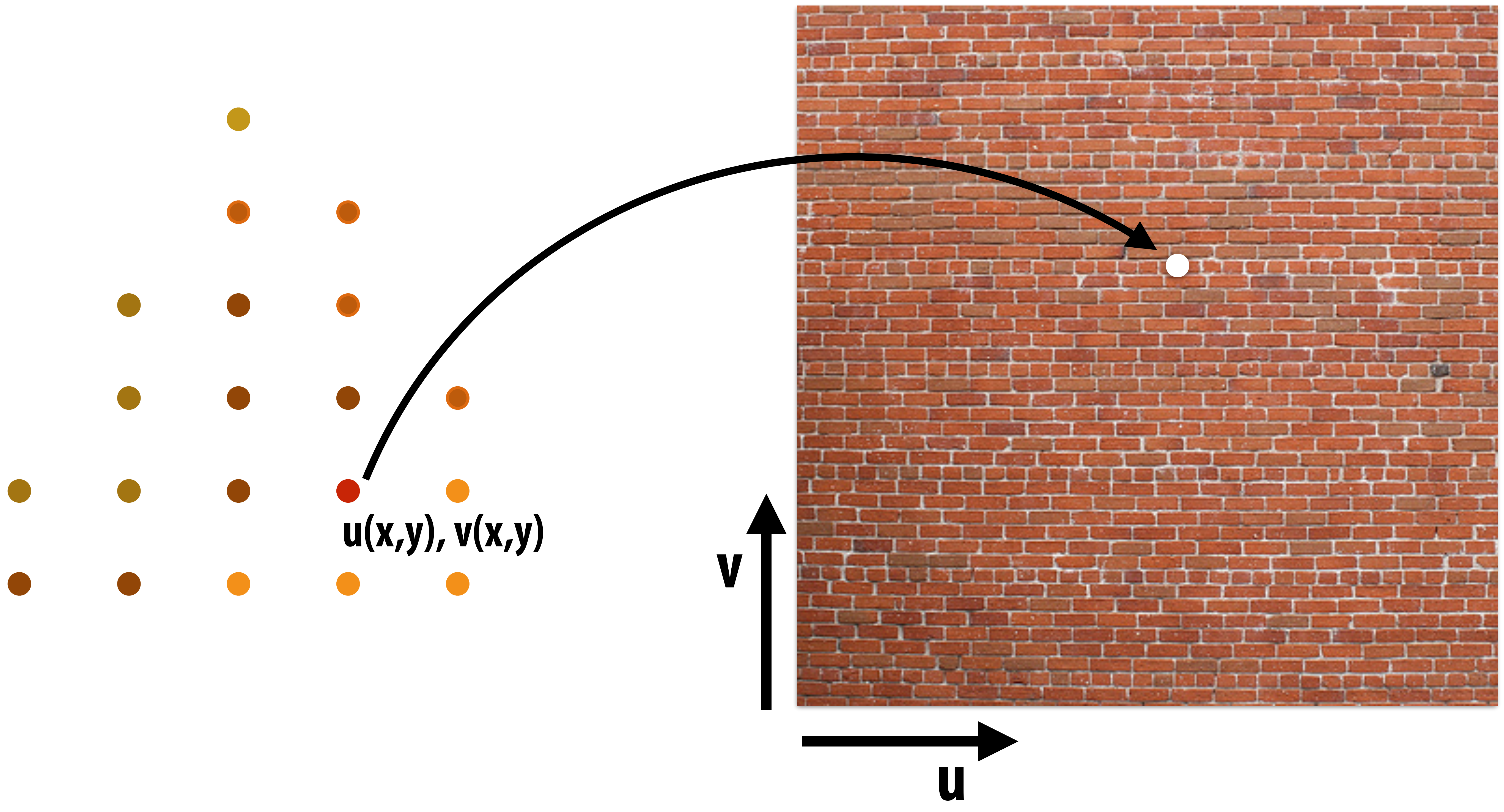
# Step 6: sample coverage

## Evaluate attributes z, u, v at all covered samples



# Step 6: compute triangle color at sample point

## e.g., sample texture map *

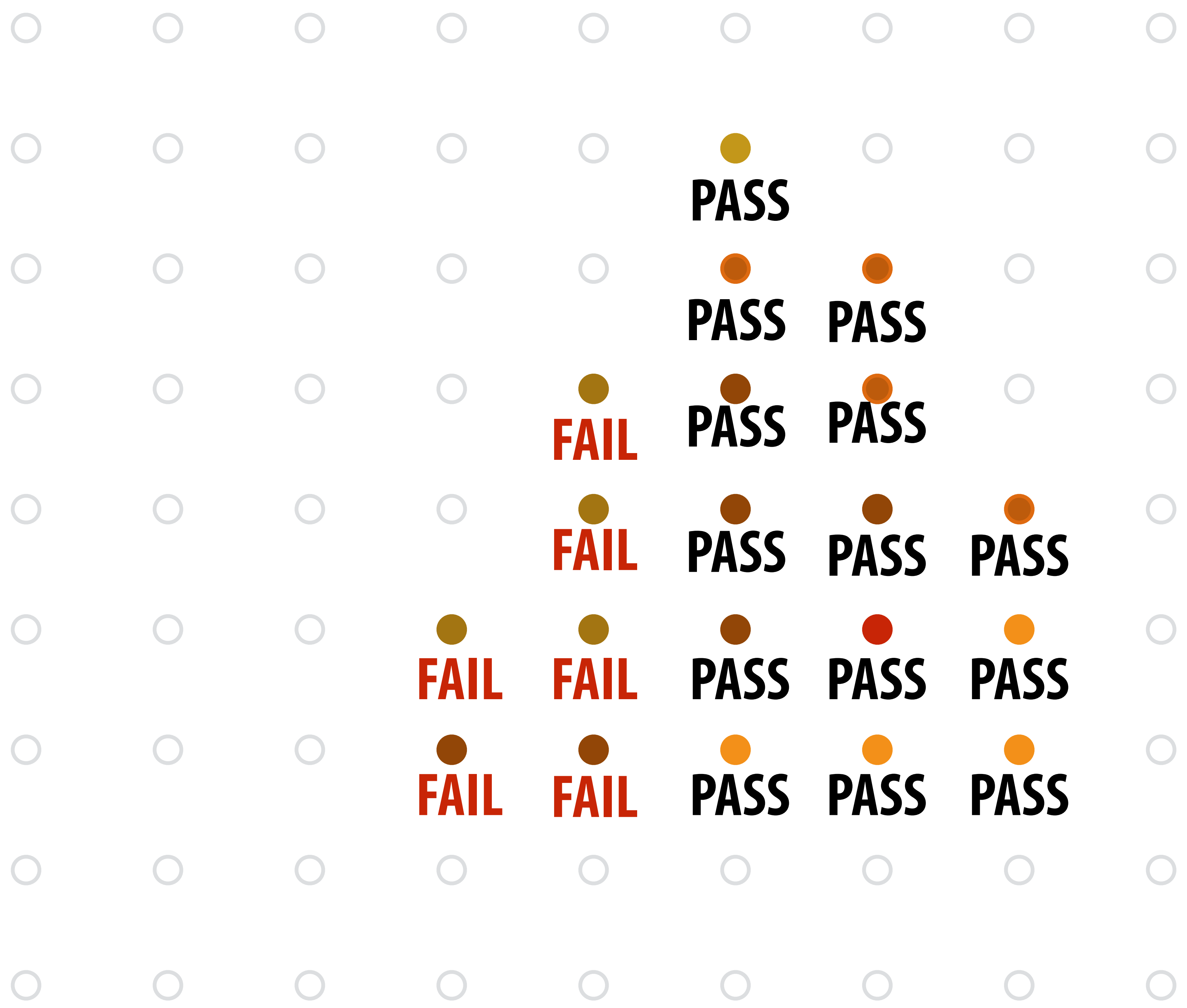


---

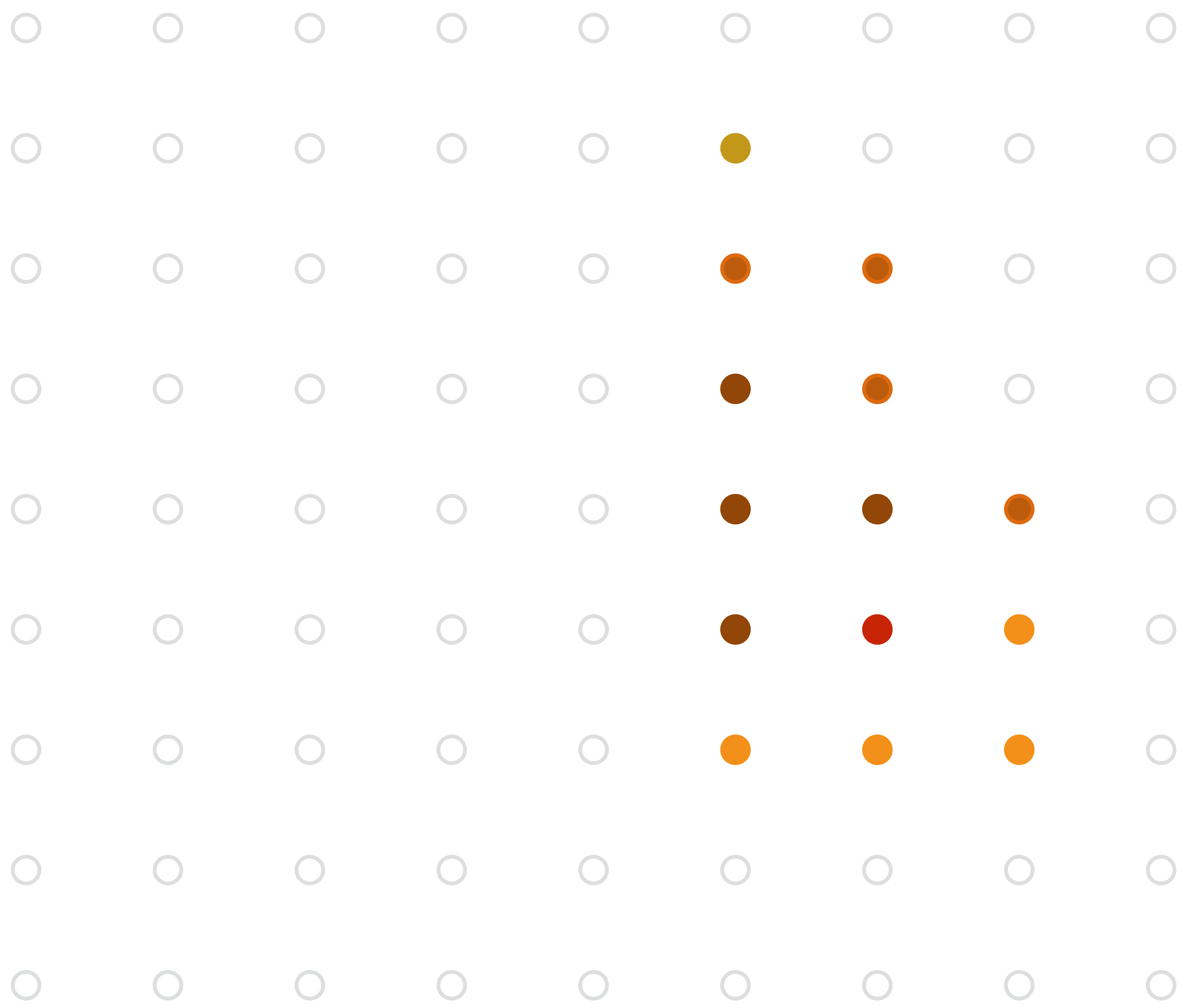
* So far, we've only described computing triangle's color at a point by interpolating per-vertex colors, or by sampling a texture map. Later in the course, we'll discuss more advanced algorithms for computing its color based on material properties and scene lighting conditions.

# Step 7: perform depth test (if enabled)

## Also update depth value at covered samples (if necessary)



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



# Step 9:

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

# Real time graphics APIs

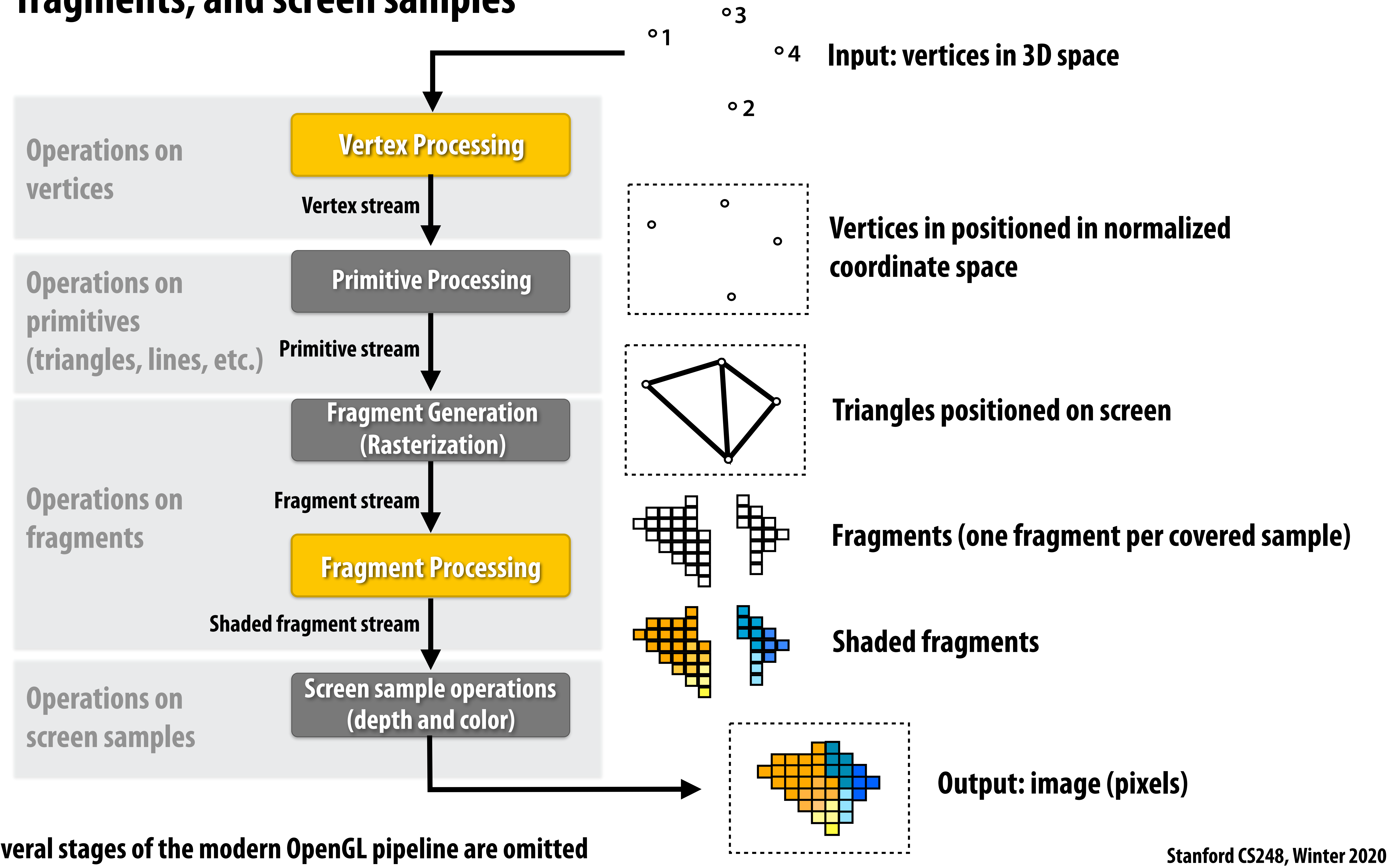
- OpenGL
- Microsoft Direct3D
- Apple Metal

- 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

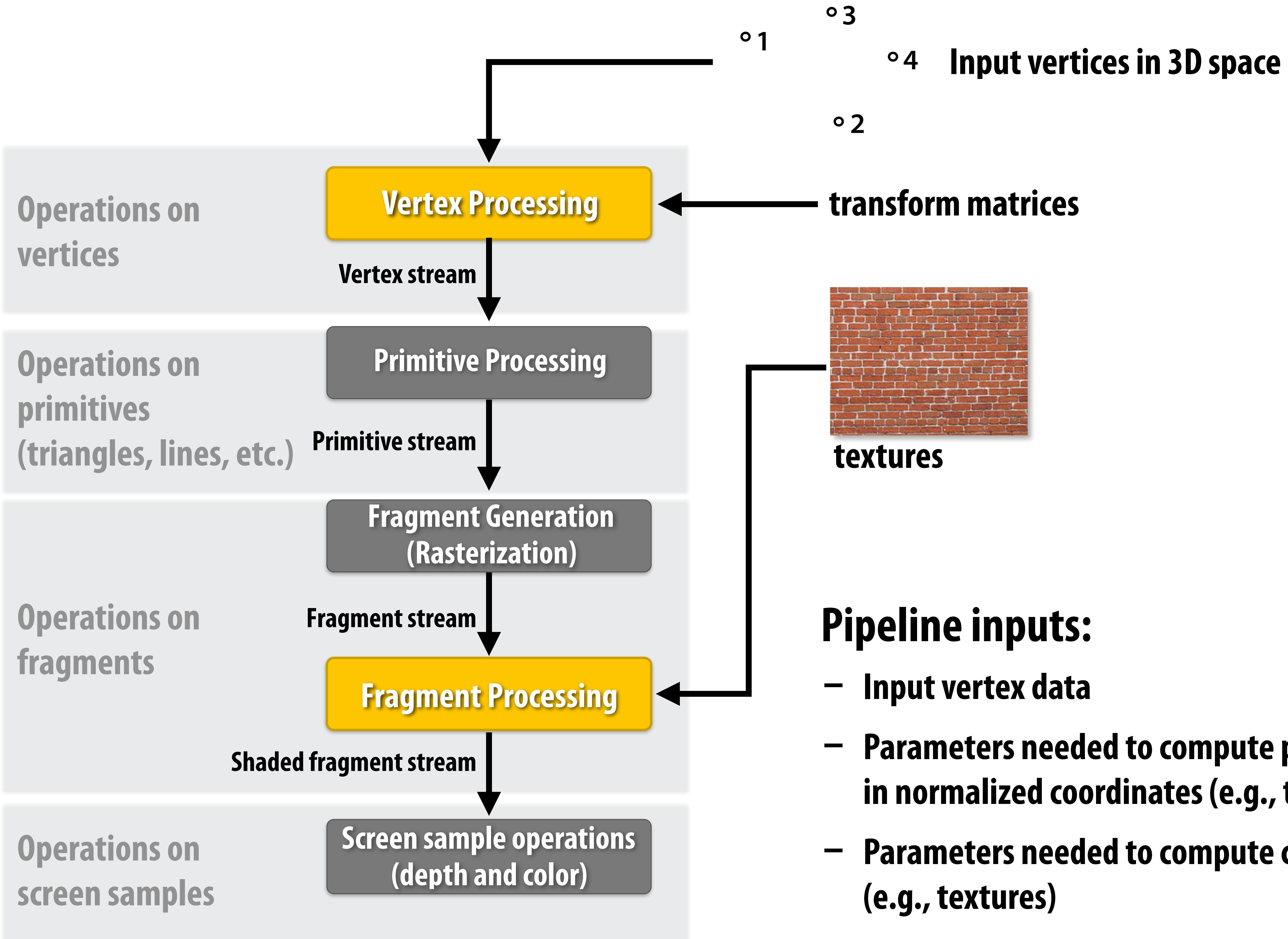
# OpenGL/Direct3D graphics pipeline *

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



* 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

# 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)

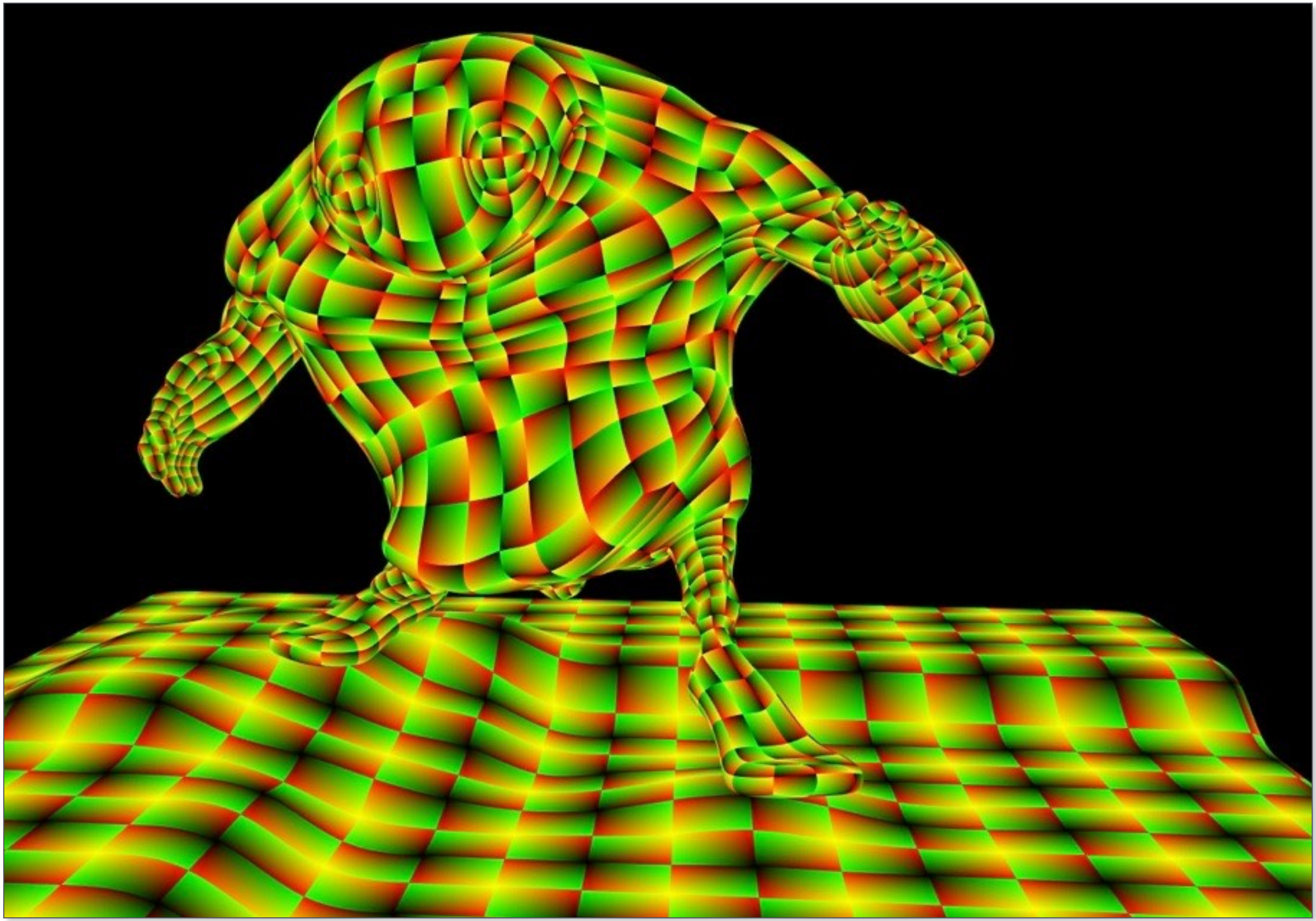
**Shader function executes once per fragment.**

**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

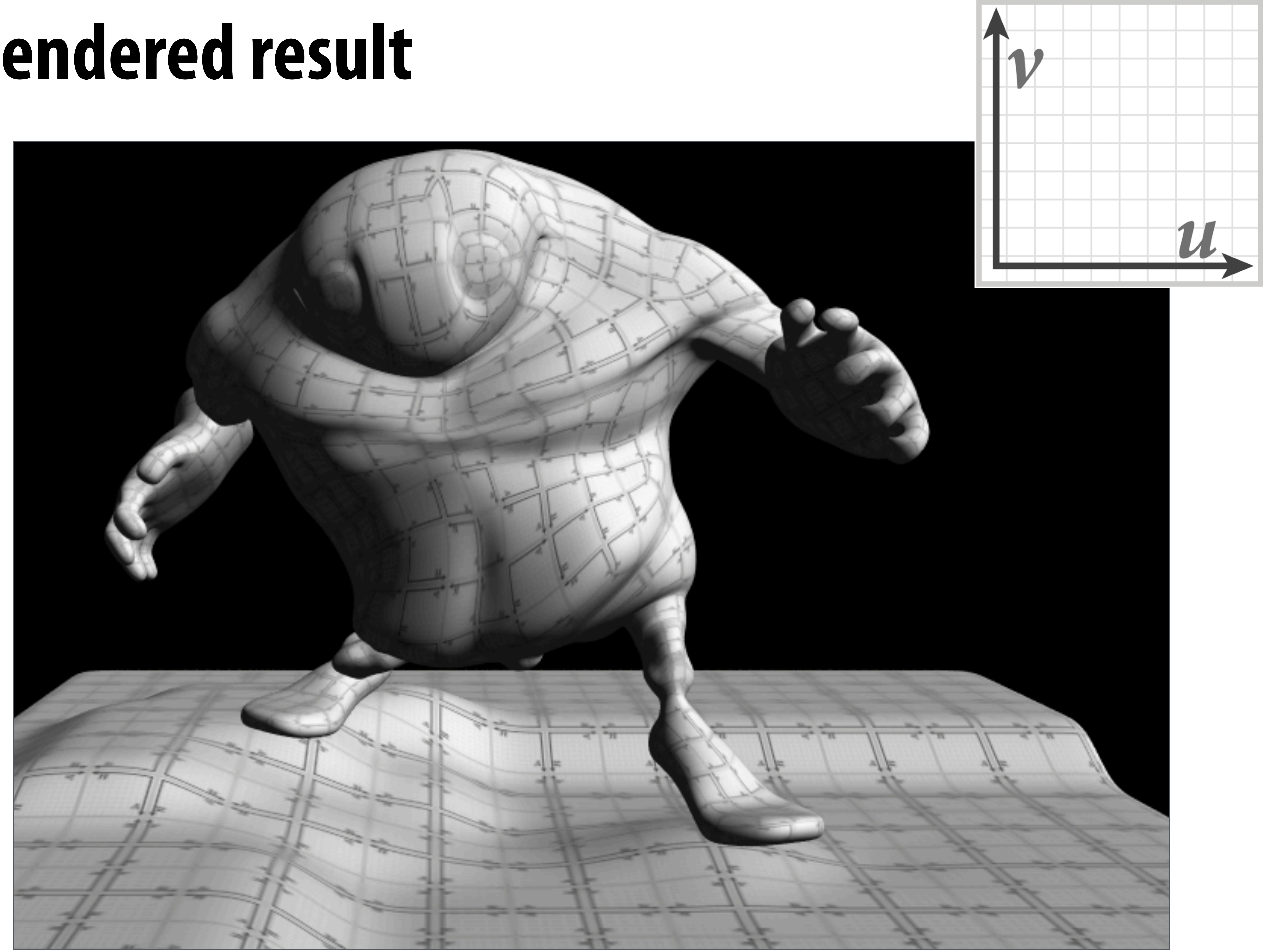
## Defines mapping from point on surface to point (uv) in texture domain



**Red channel = u, Green channel = v**

**So uv=(0,0) is black, uv=(1,1) is yellow**

# Rendered result



# Goal: render very high complexity 3D scenes

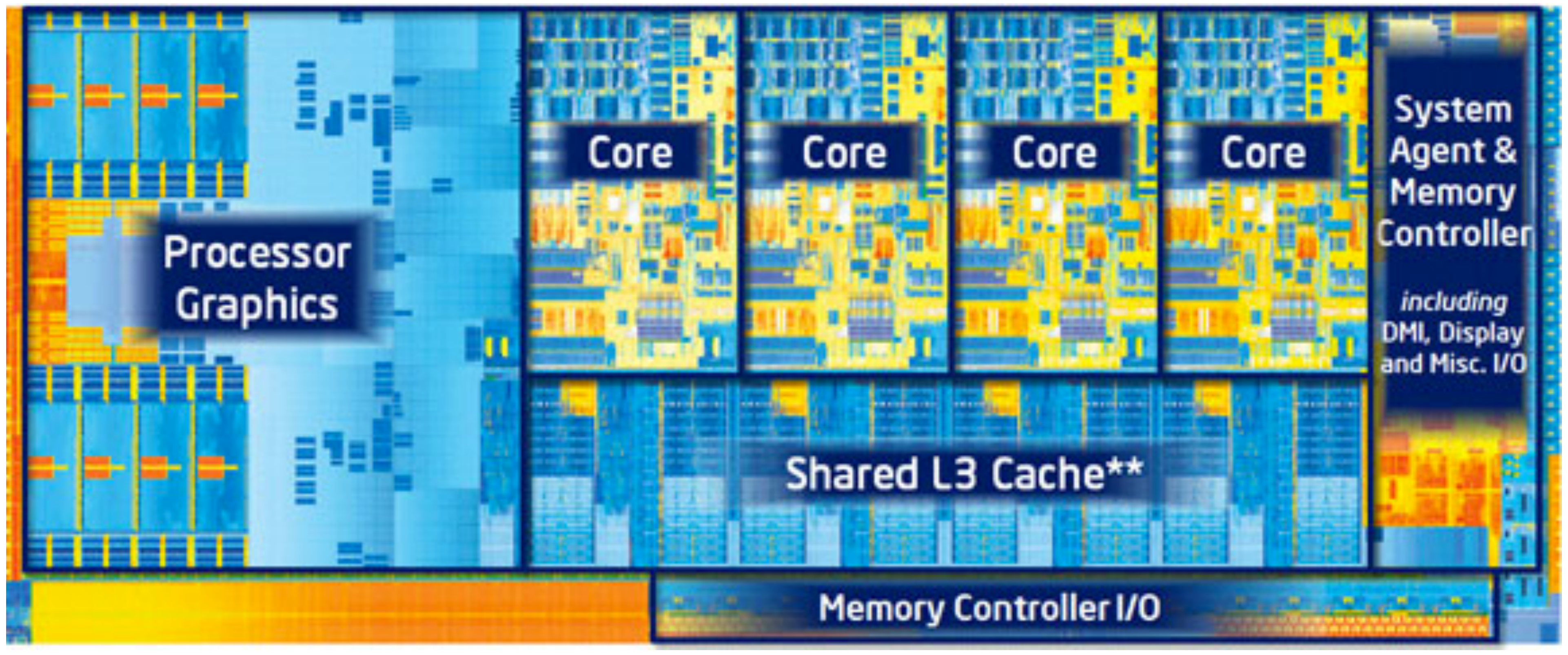
- 100's of thousands to millions of triangles in a scene
- Complex vertex and fragment shader computations
- High resolution screen outputs (2-4 Mpixel + supersampling)
- 30-60 fps

Unreal Engine Kite Demo (Epic Games 2015)

# Graphics pipeline implementation: GPUs

## Specialized processors for executing graphics pipeline computations

**Discrete GPU card**
**(NVIDIA GeForce Titan X)**

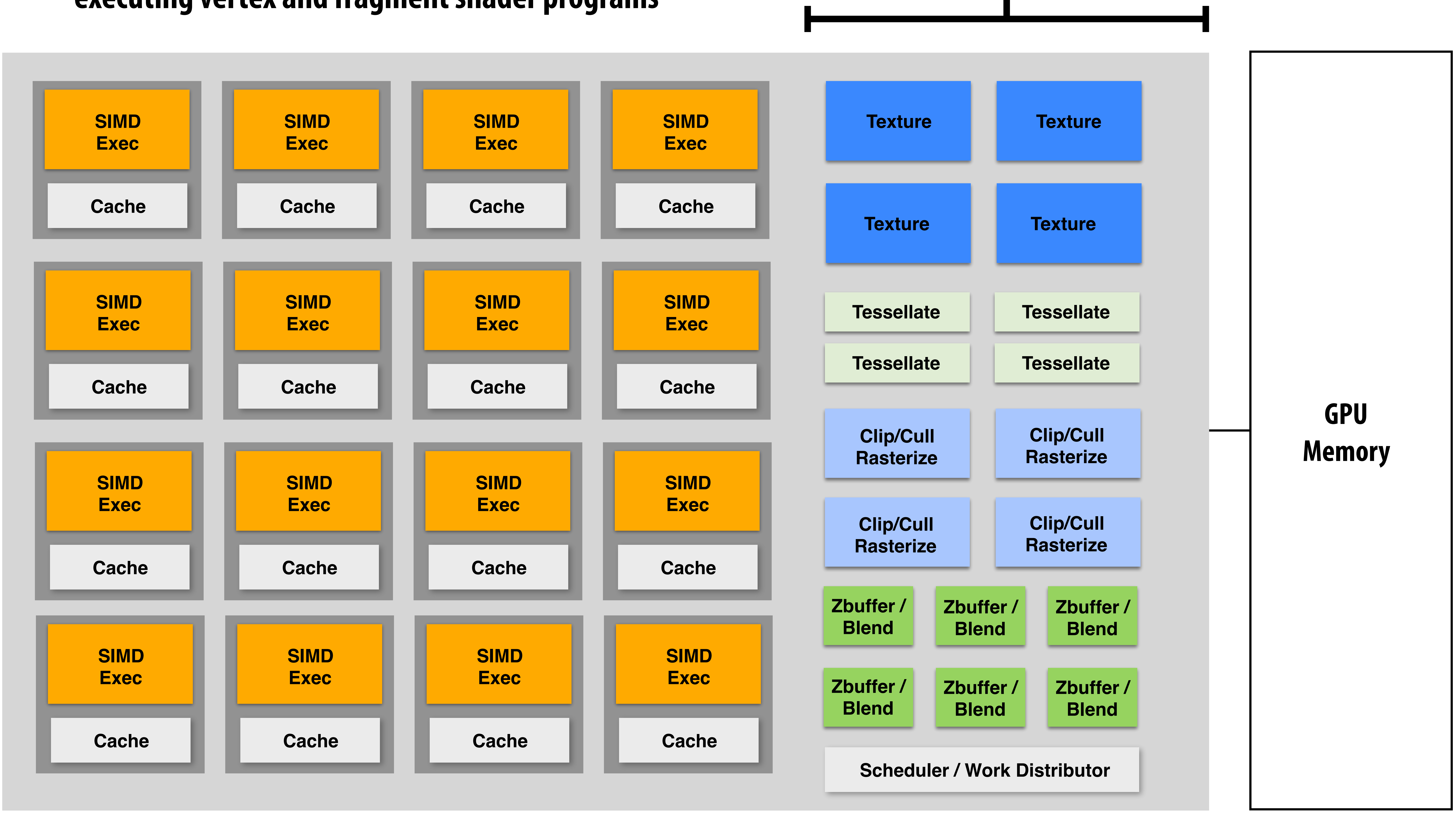


**Integrated GPU: part of modern Intel CPU chip**

# 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

- Graphics pipeline:
  - Structures 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)