

**Lecture 5:**

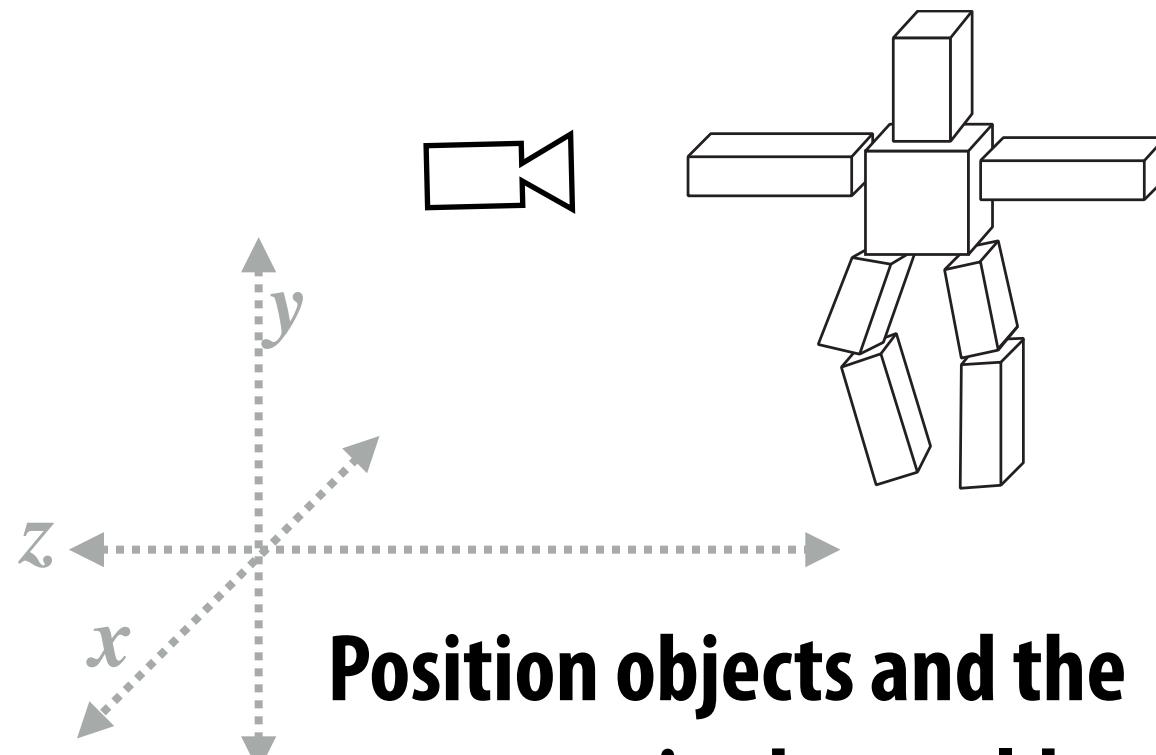
# **The Rasterization Pipeline**

## **(and its implementation on GPUs)**

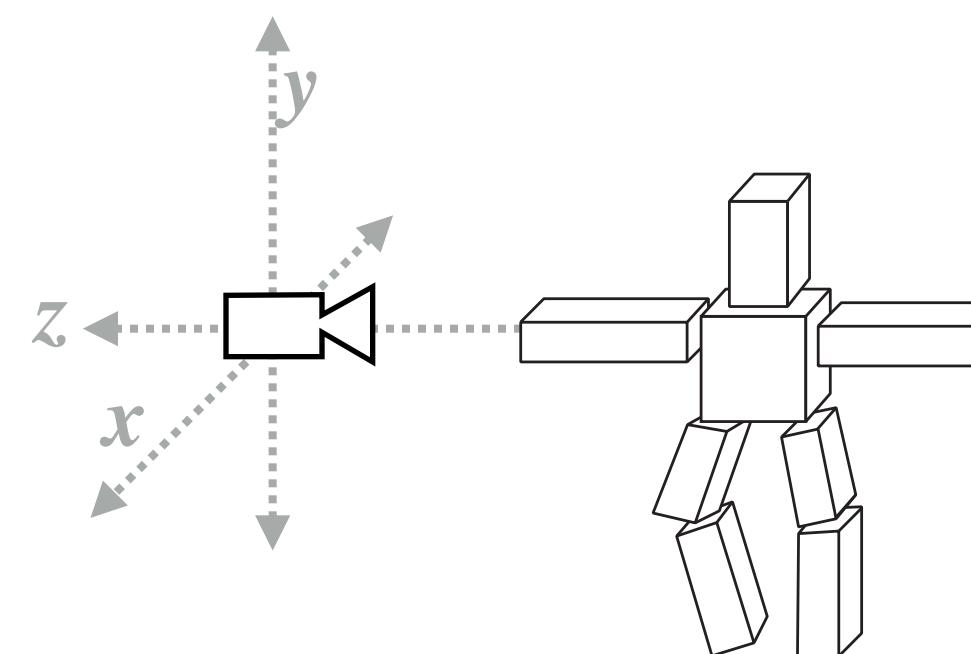
---

**Computer Graphics**  
**CMU 15-462/15-662, Fall 2015**

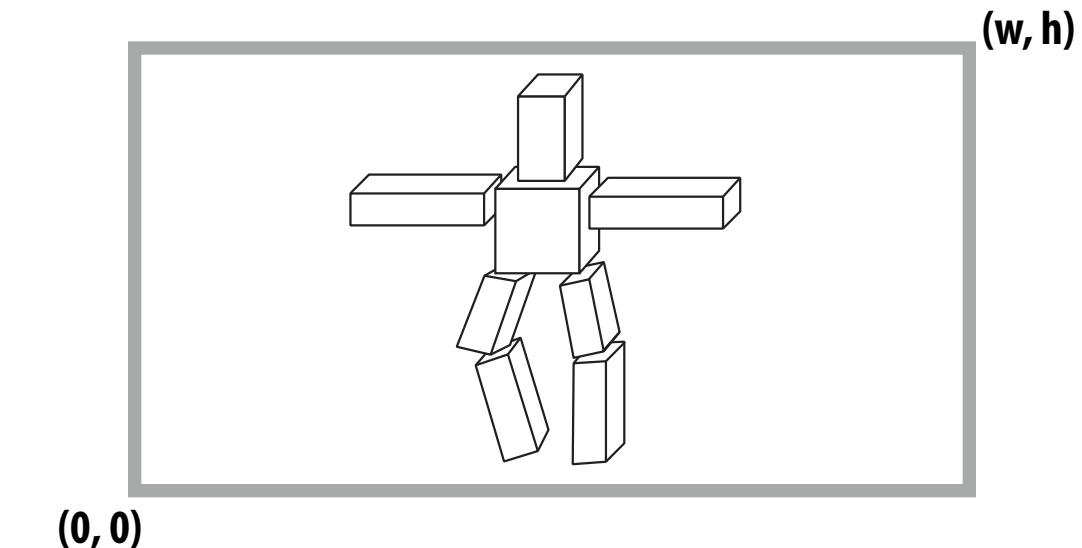
# What you know how to do (at this point in the course)



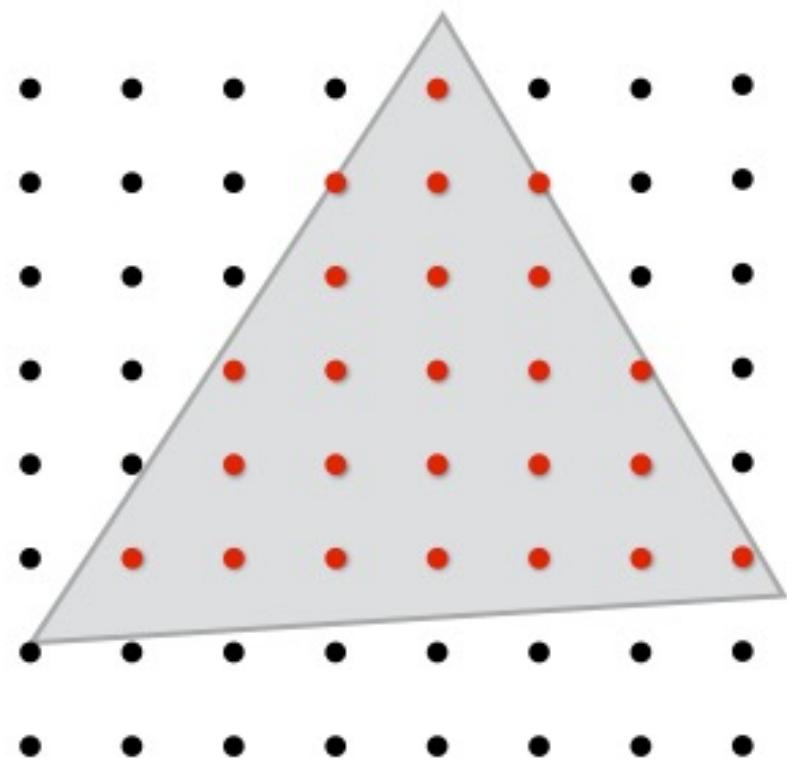
**Position objects and the camera in the world**



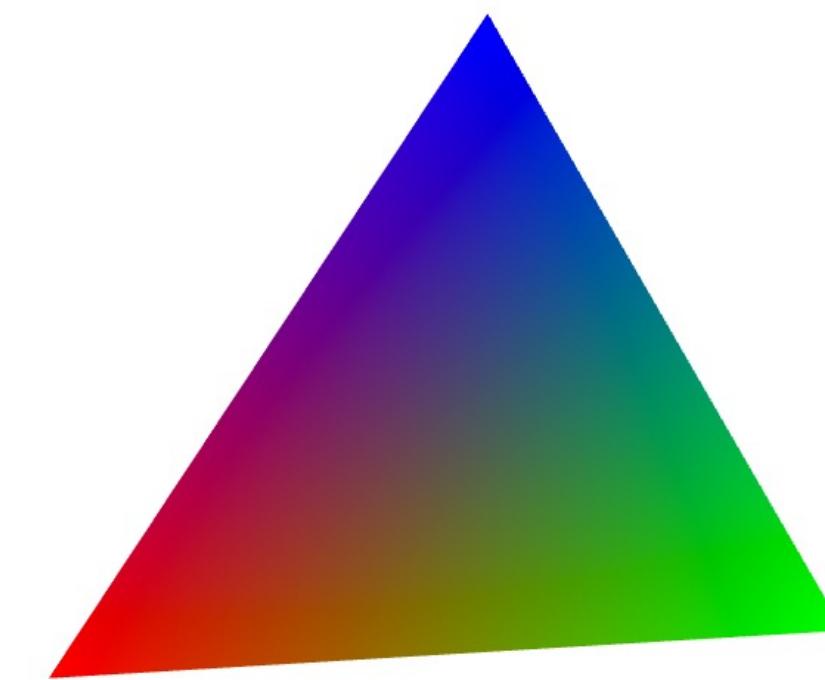
**Determine the position of objects relative to the camera**



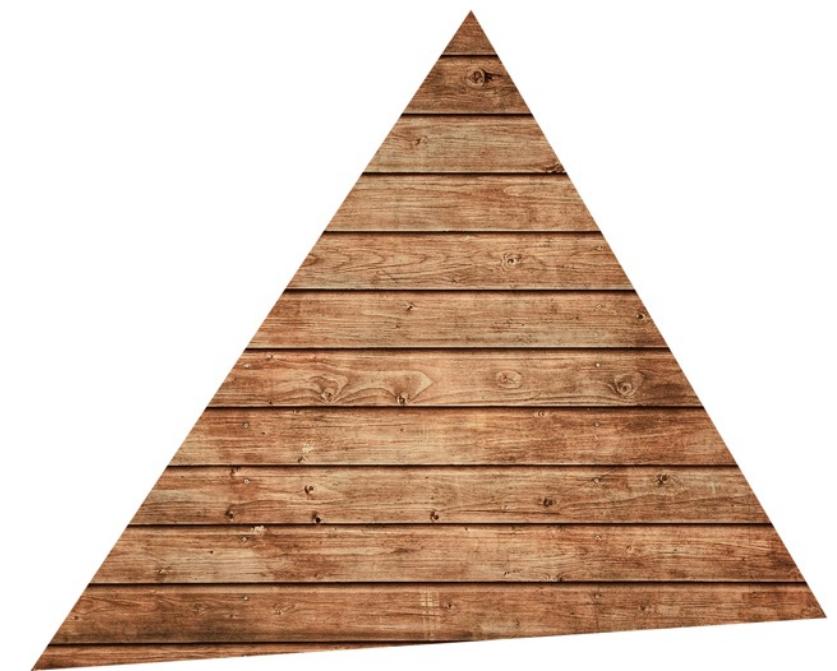
**Project objects onto the screen**



**Sample triangle coverage**



**Compute triangle attribute values at covered sample points**



**Sample texture maps**

# 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.



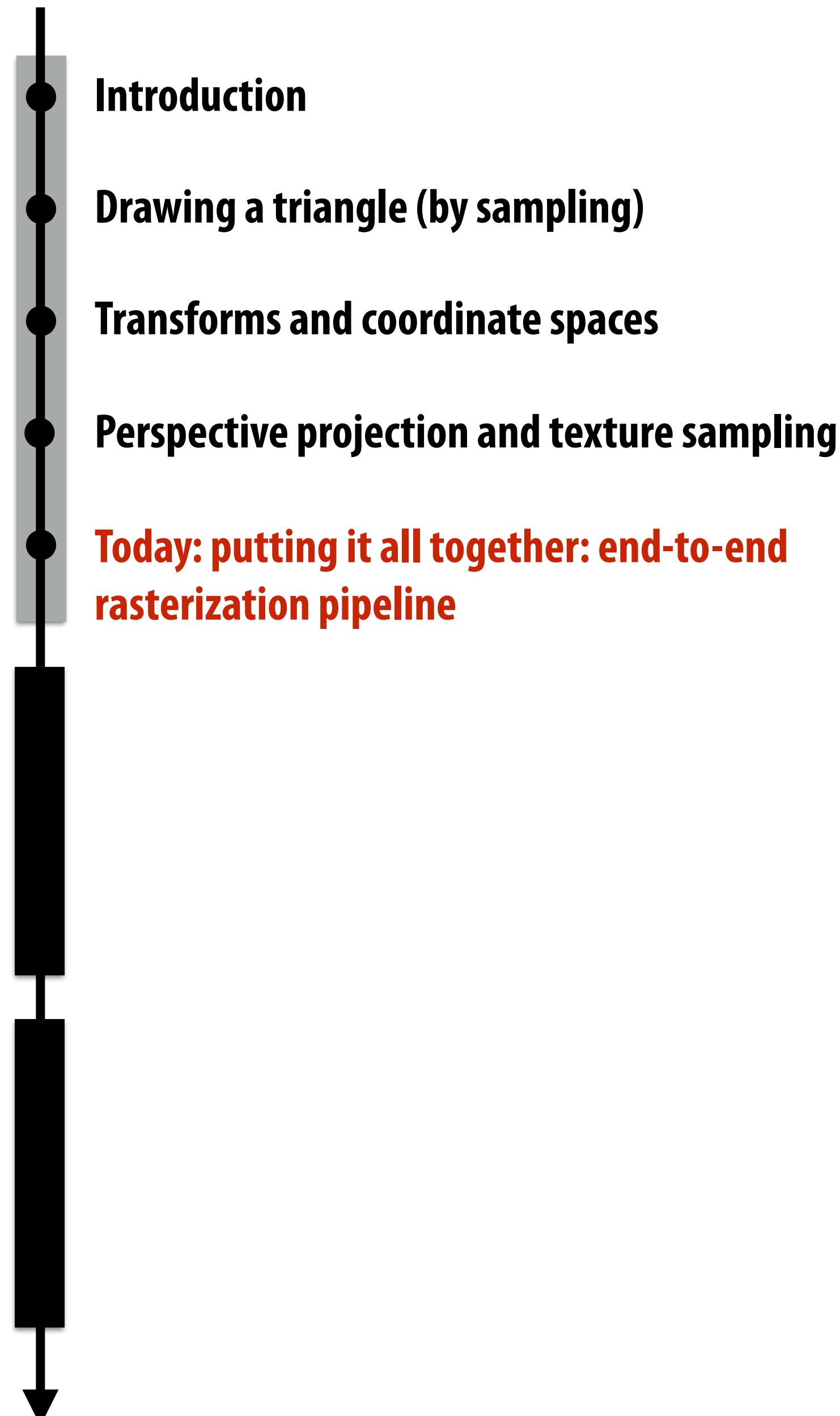
# Course roadmap

## Drawing Things

Key concepts:

Sampling (and anti-aliasing)

Coordinate Spaces and Transforms

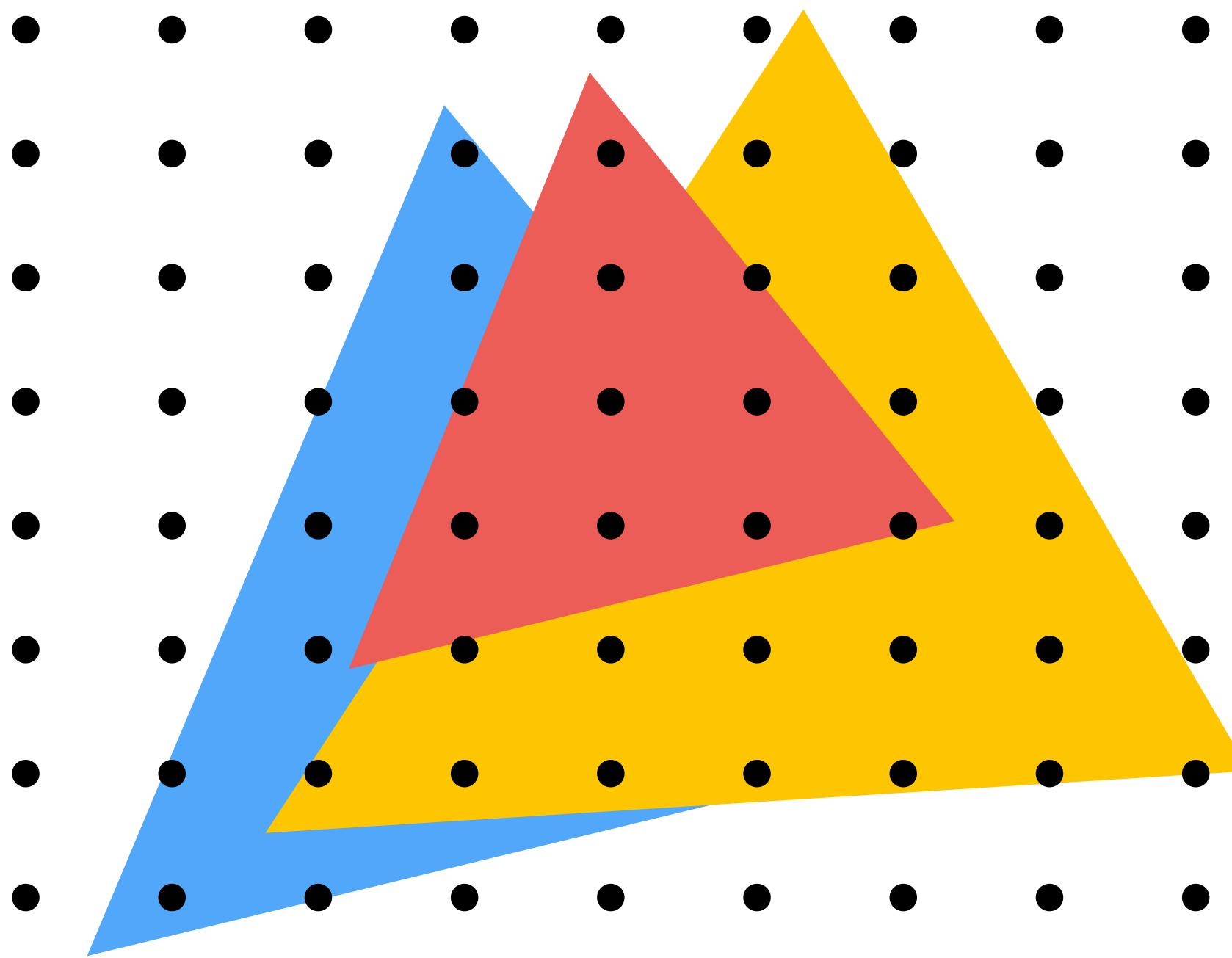


## Geometry

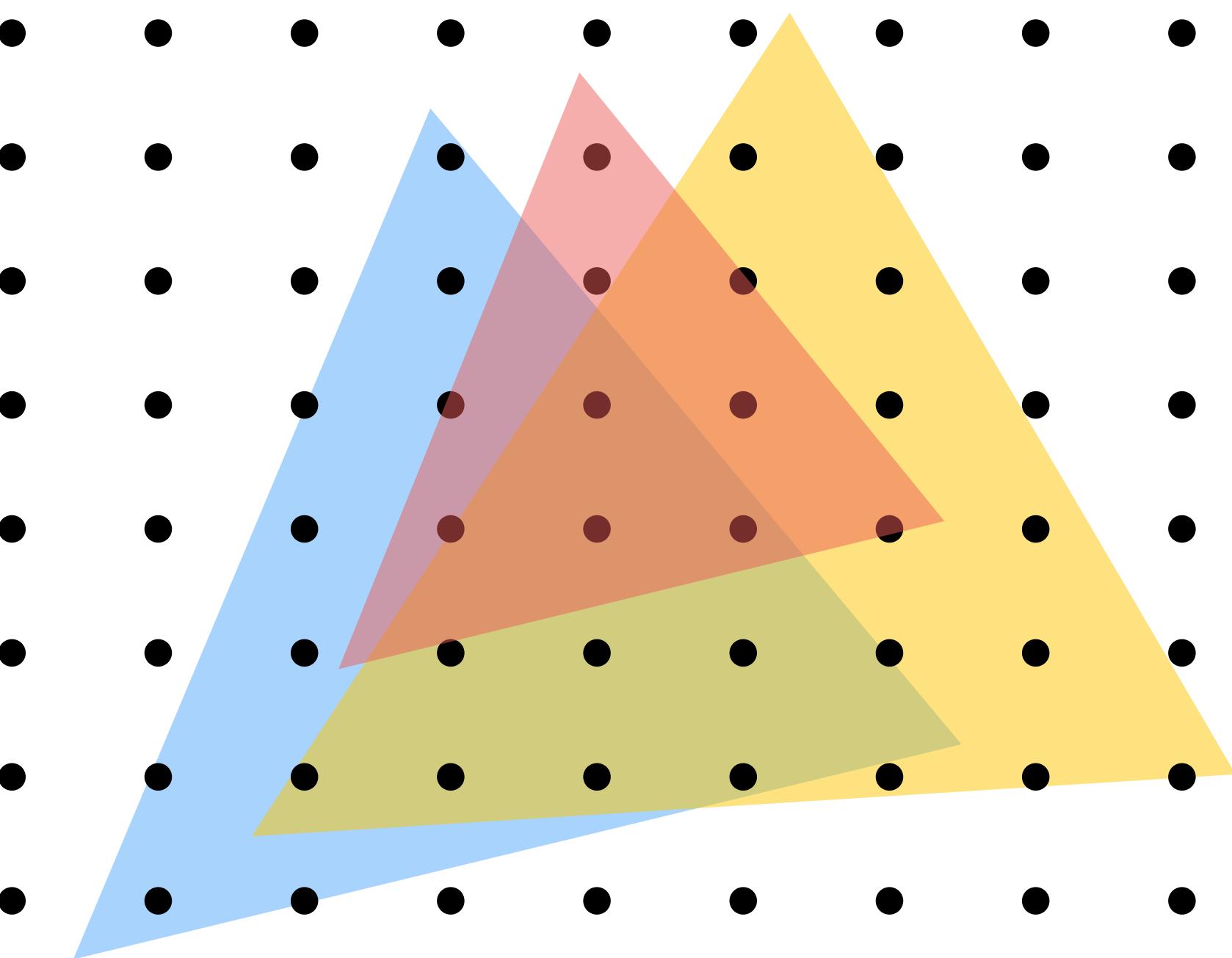
## Materials and Lighting

# Occlusion

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



Opaque Triangles



50% transparent triangles

# 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.**

$$[\mathbf{p}_{0x} \quad \mathbf{p}_{0y}]^T, \quad d_0$$

$$[\mathbf{p}_{1x} \quad \mathbf{p}_{1y}]^T, \quad d_1$$

$$[\mathbf{p}_{2x} \quad \mathbf{p}_{2y}]^T, \quad d_2$$

**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.**

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

For each coverage sample point, 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

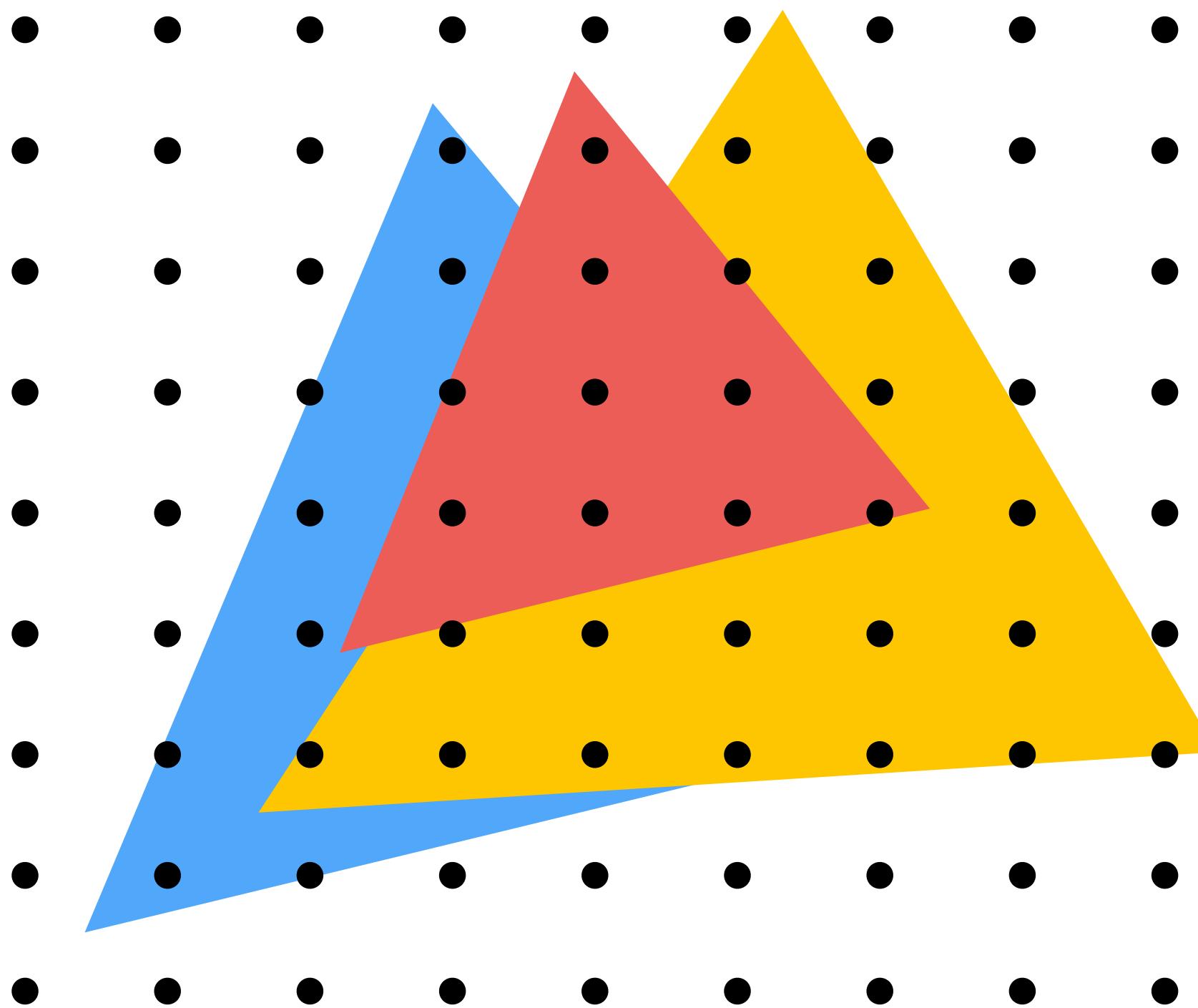
Black = small distance

White = large distance

# Depth buffer example

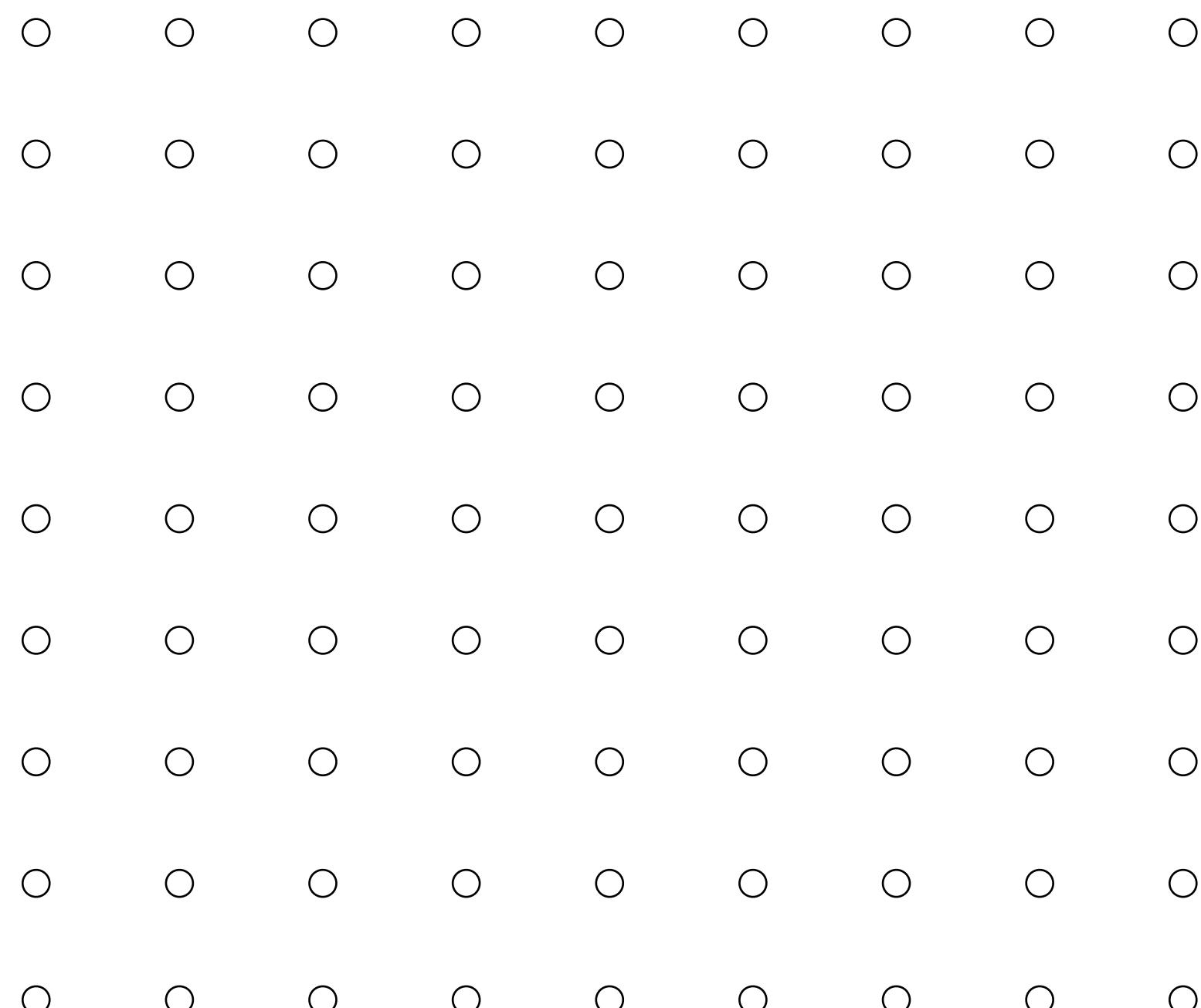


# Example: rendering three opaque triangles



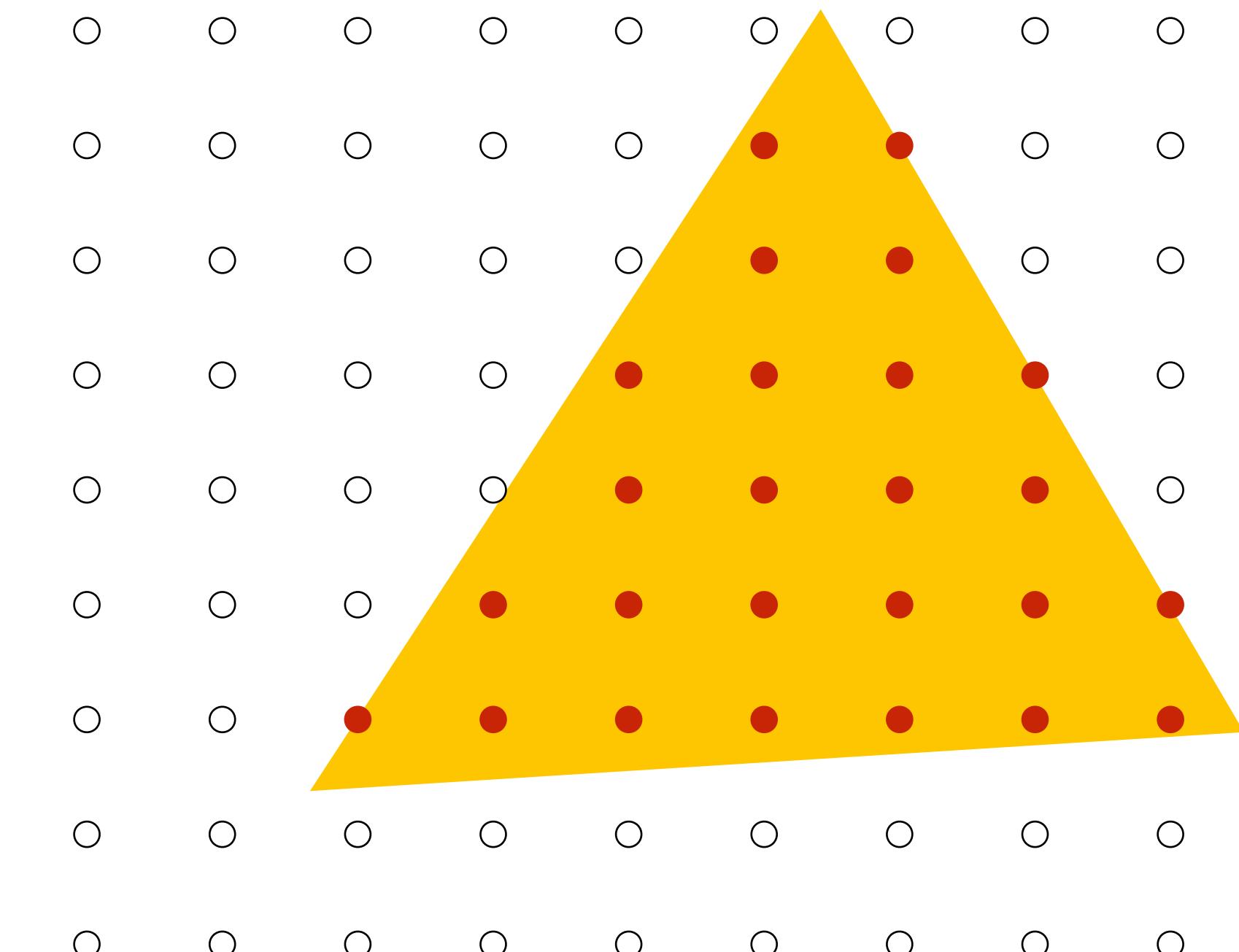
# Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:  
**depth = 0.5**



**Color buffer contents**

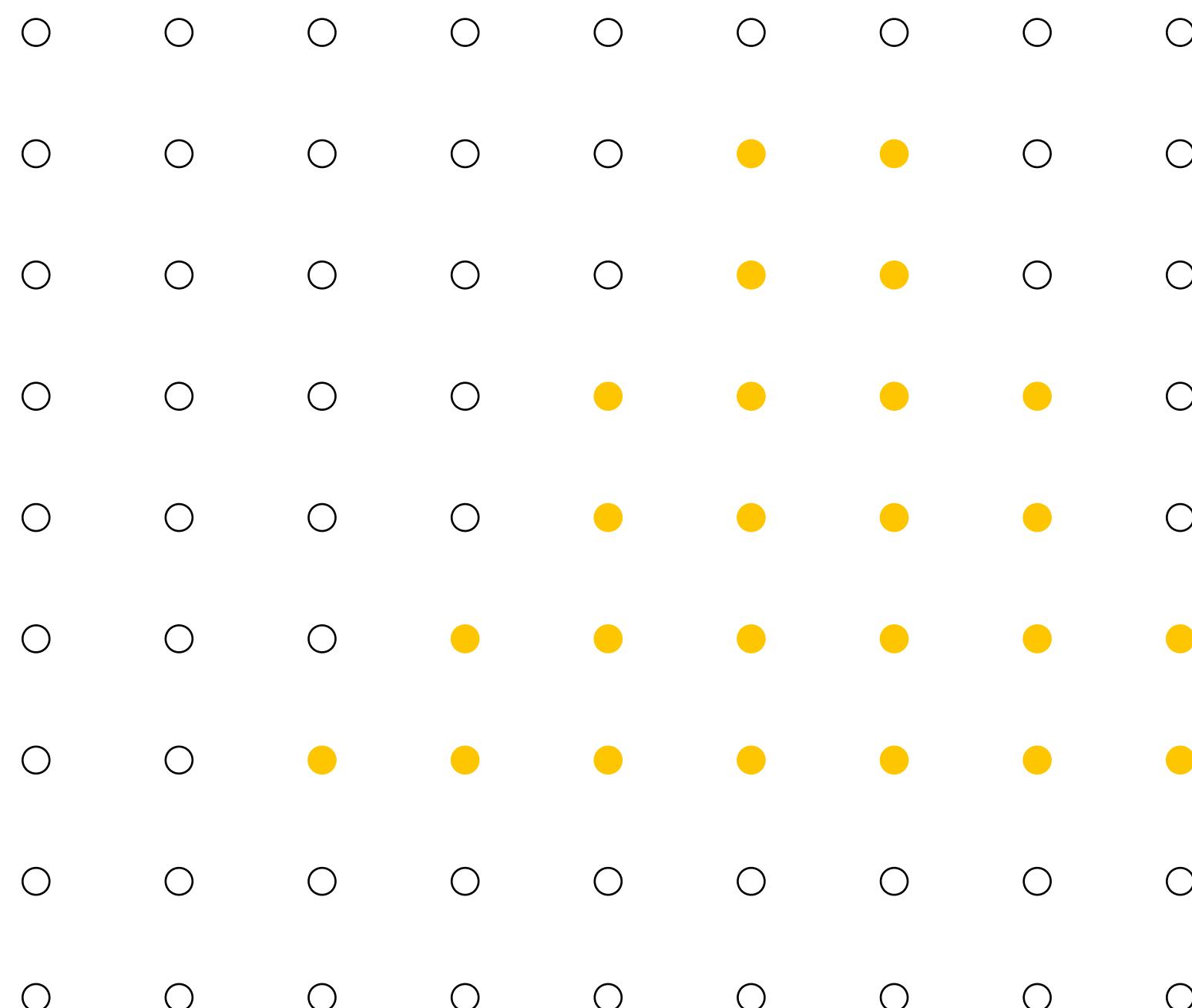
Grayscale value of sample point  
used to indicate distance  
**White = large distance**  
**Black = small distance**  
**Red = sample passed depth test**



**Depth buffer contents**

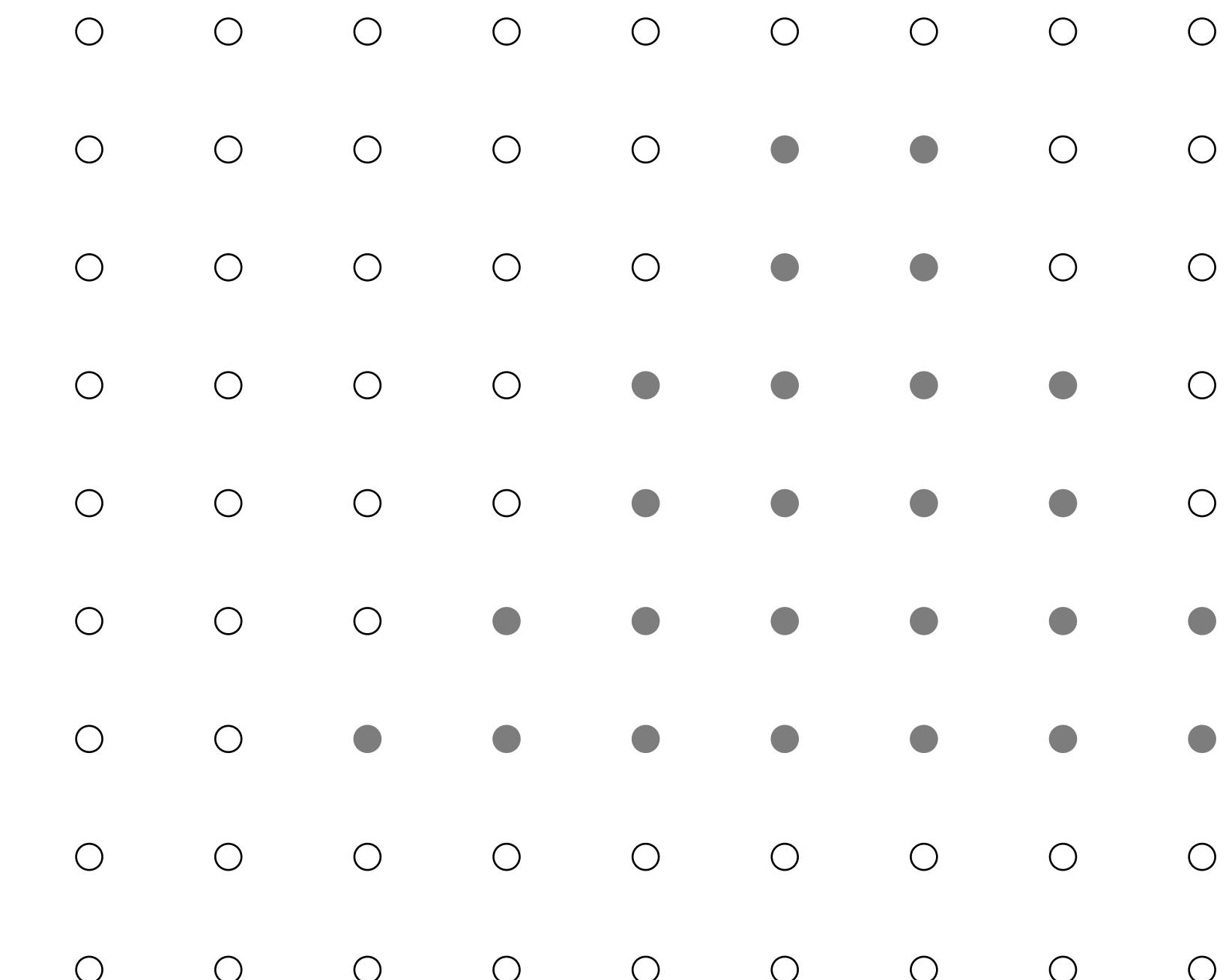
# Occlusion using the depth-buffer (Z-buffer)

After processing yellow triangle:



Color buffer contents

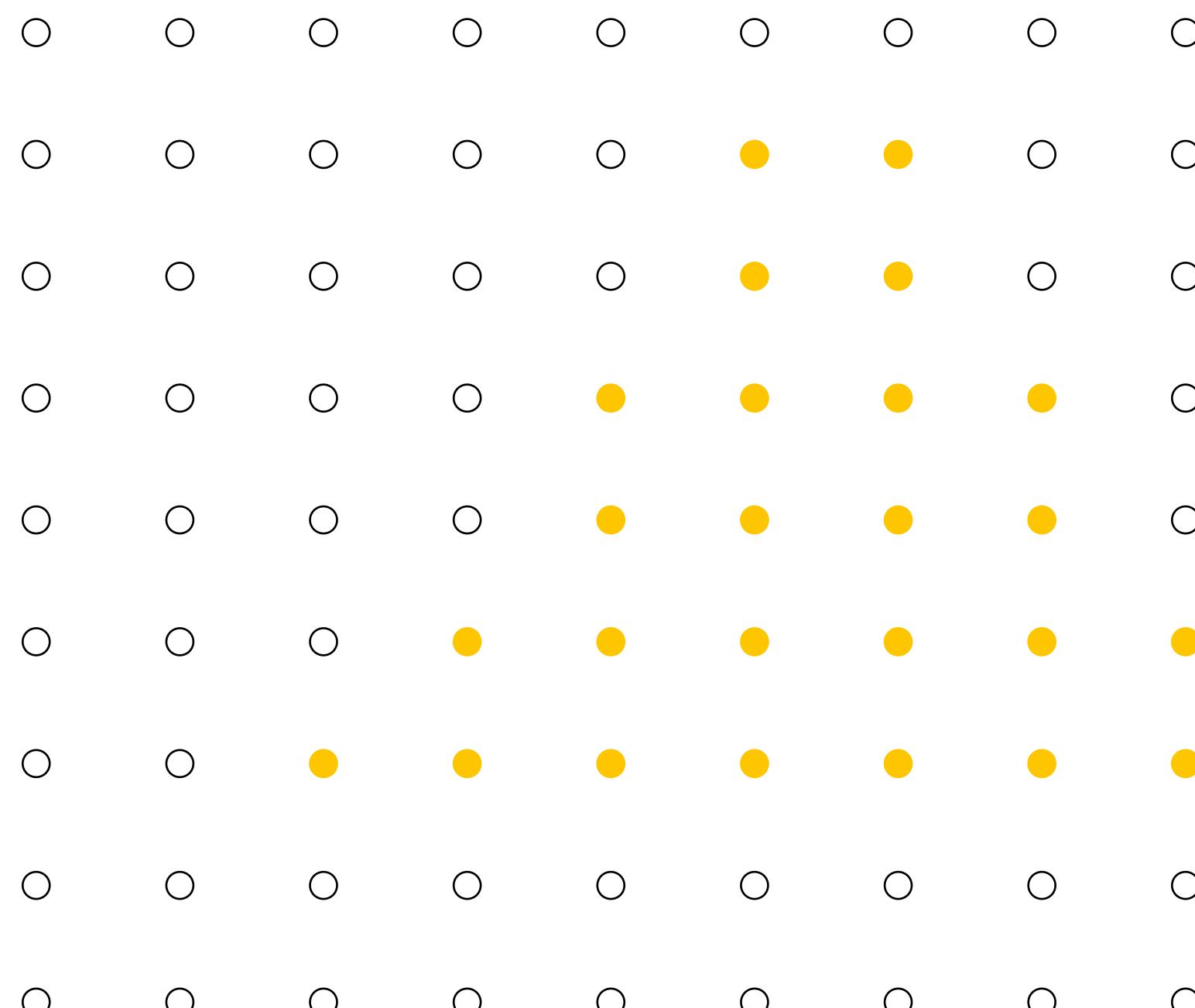
Grayscale value of sample point  
used to indicate distance  
White = large distance  
Black = small distance  
Red = sample passed depth test



Depth buffer contents

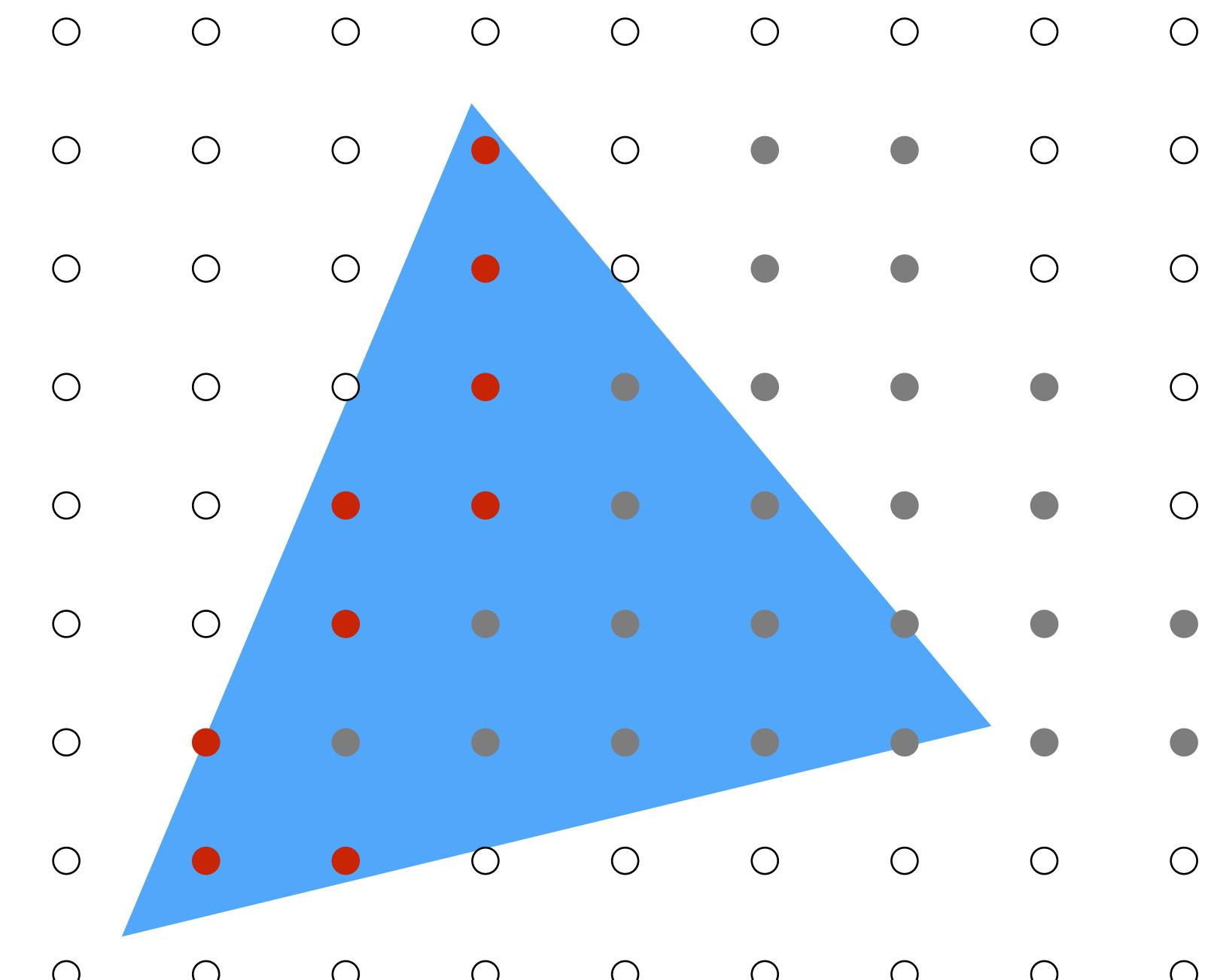
# Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:  
depth = 0.75



Color buffer contents

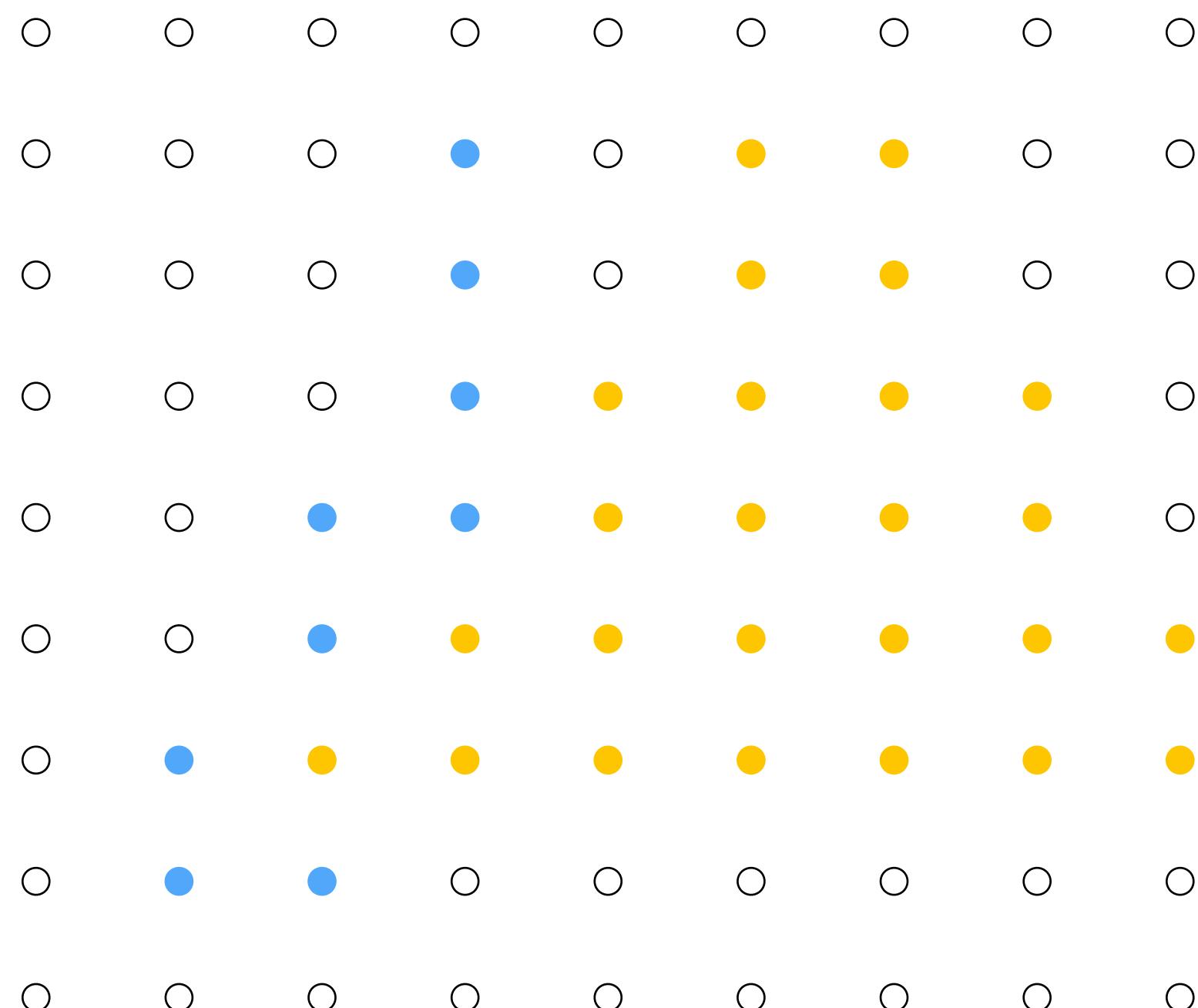
Grayscale value of sample point  
used to indicate distance  
White = large distance  
Black = small distance  
Red = sample passed depth test



Depth buffer contents

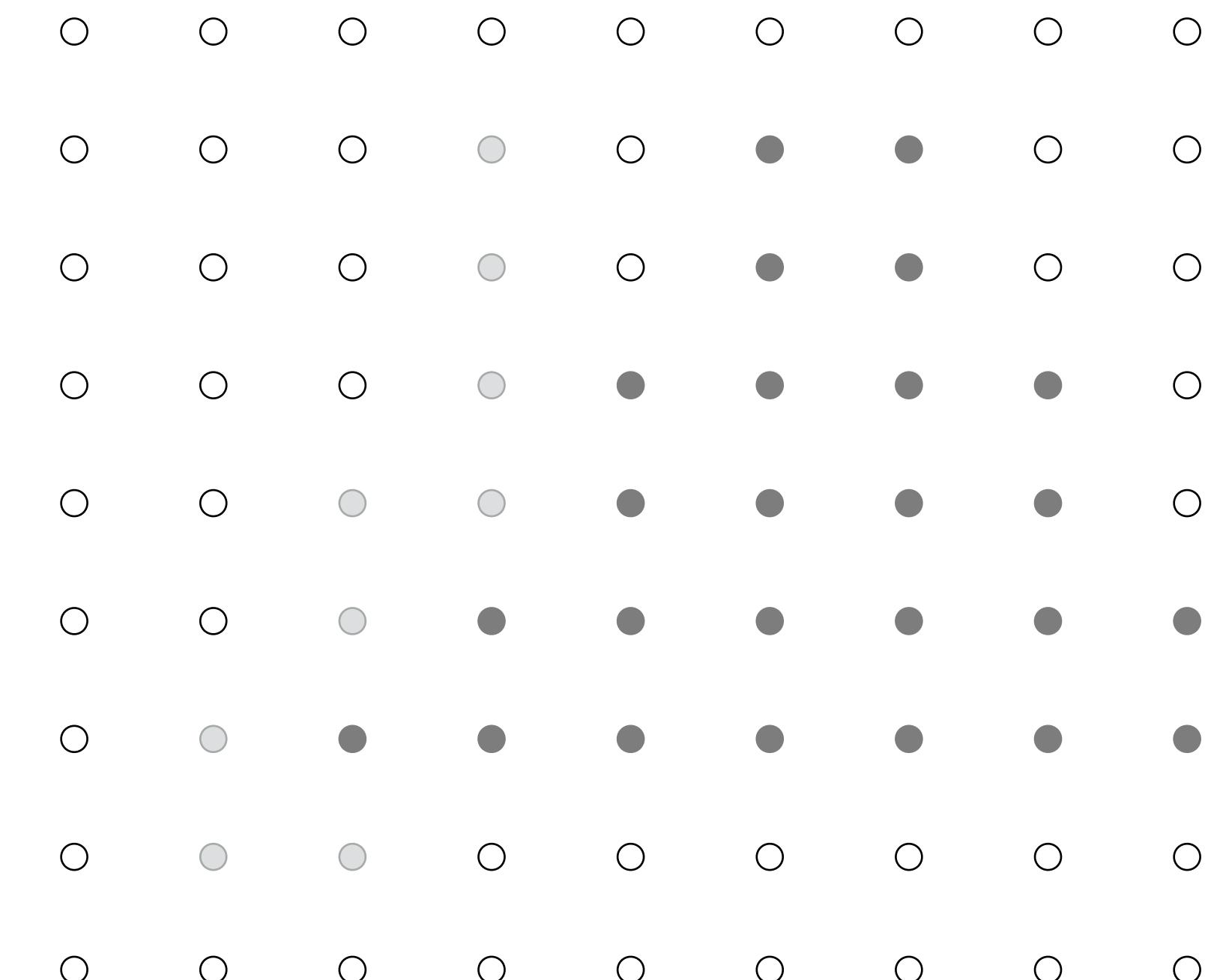
# Occlusion using the depth-buffer (Z-buffer)

After processing blue triangle:



Color buffer contents

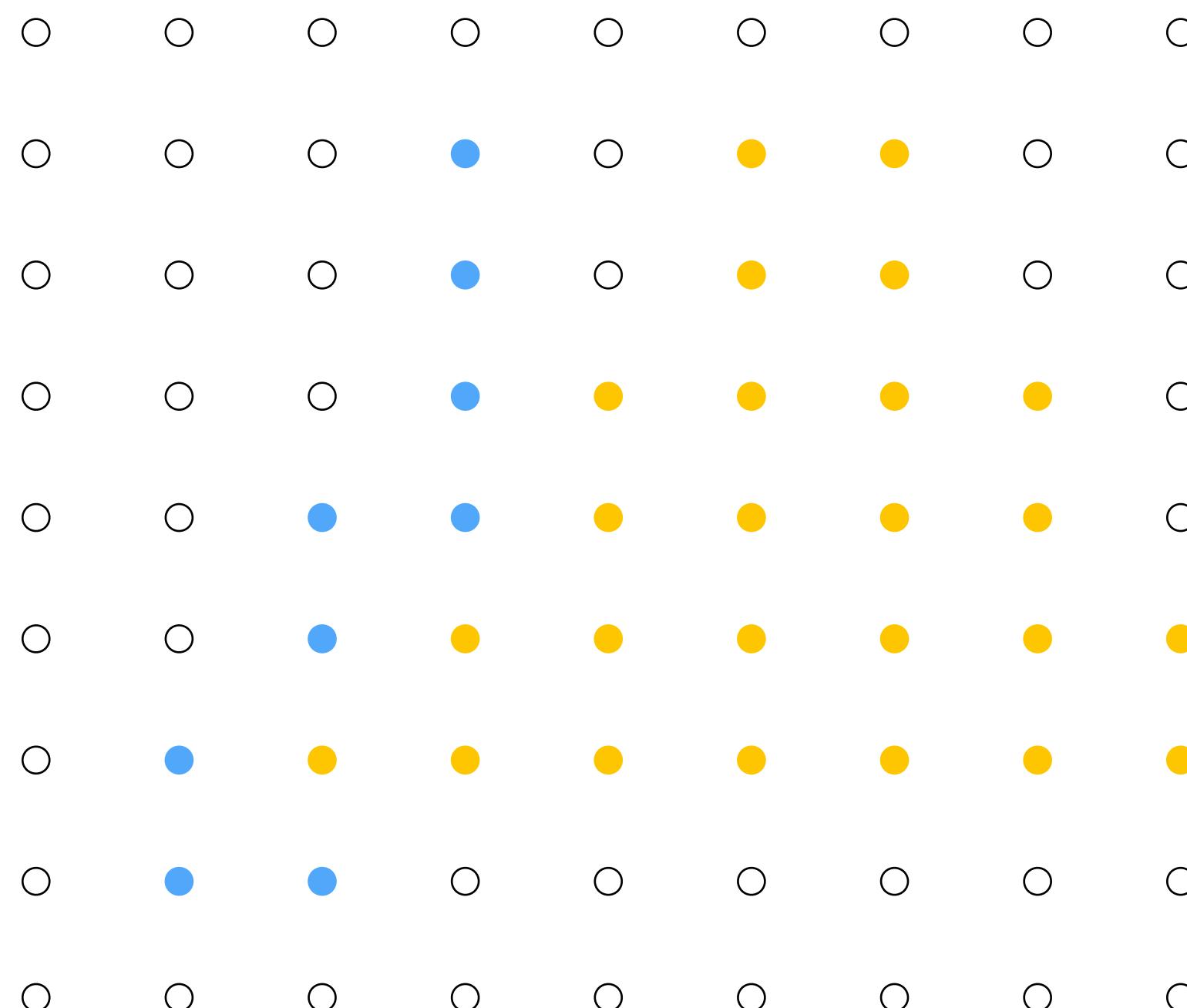
Grayscale value of sample point  
used to indicate distance  
White = large distance  
Black = small distance  
Red = sample passed depth test



Depth buffer contents

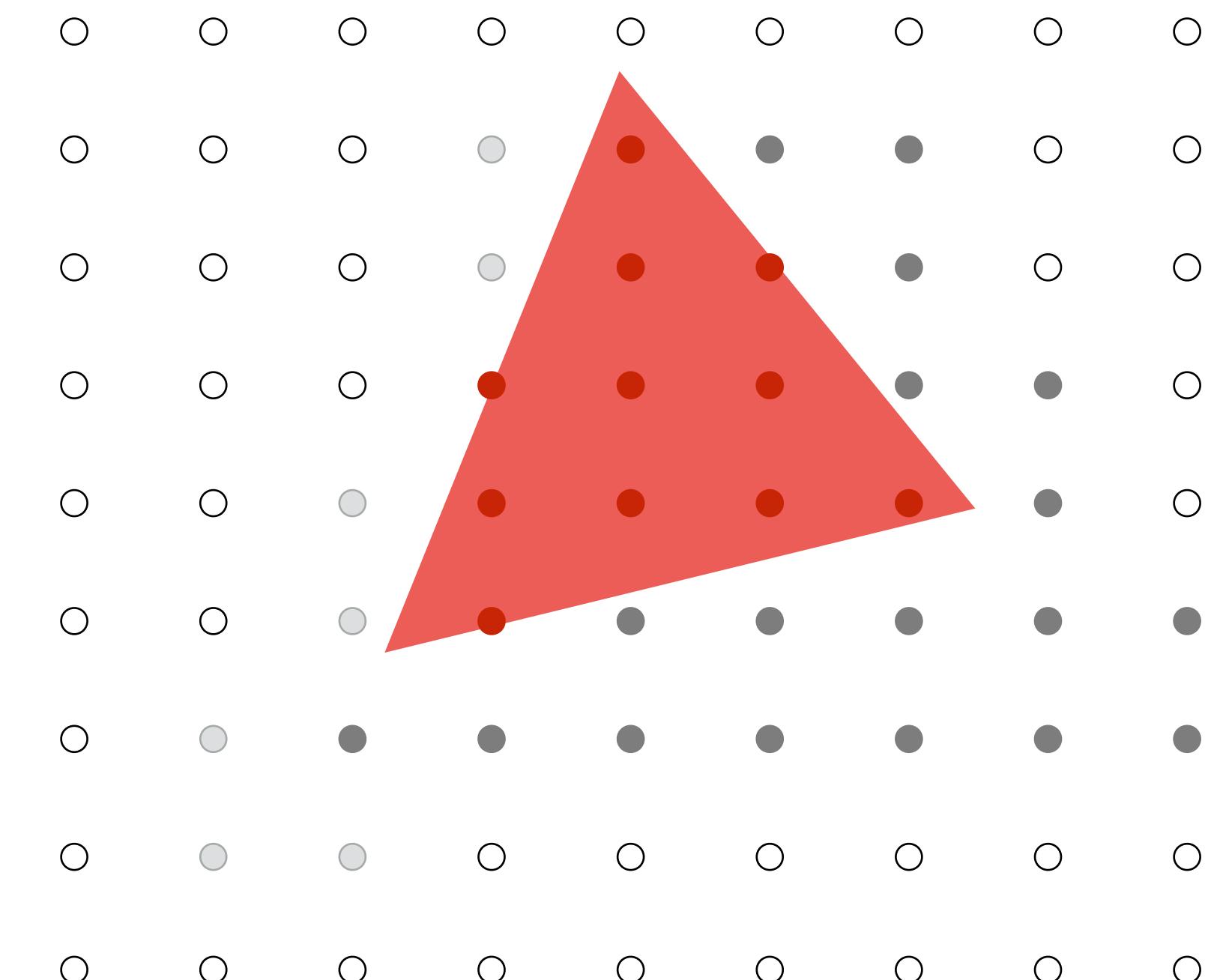
# Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:  
depth = 0.25



Color buffer contents

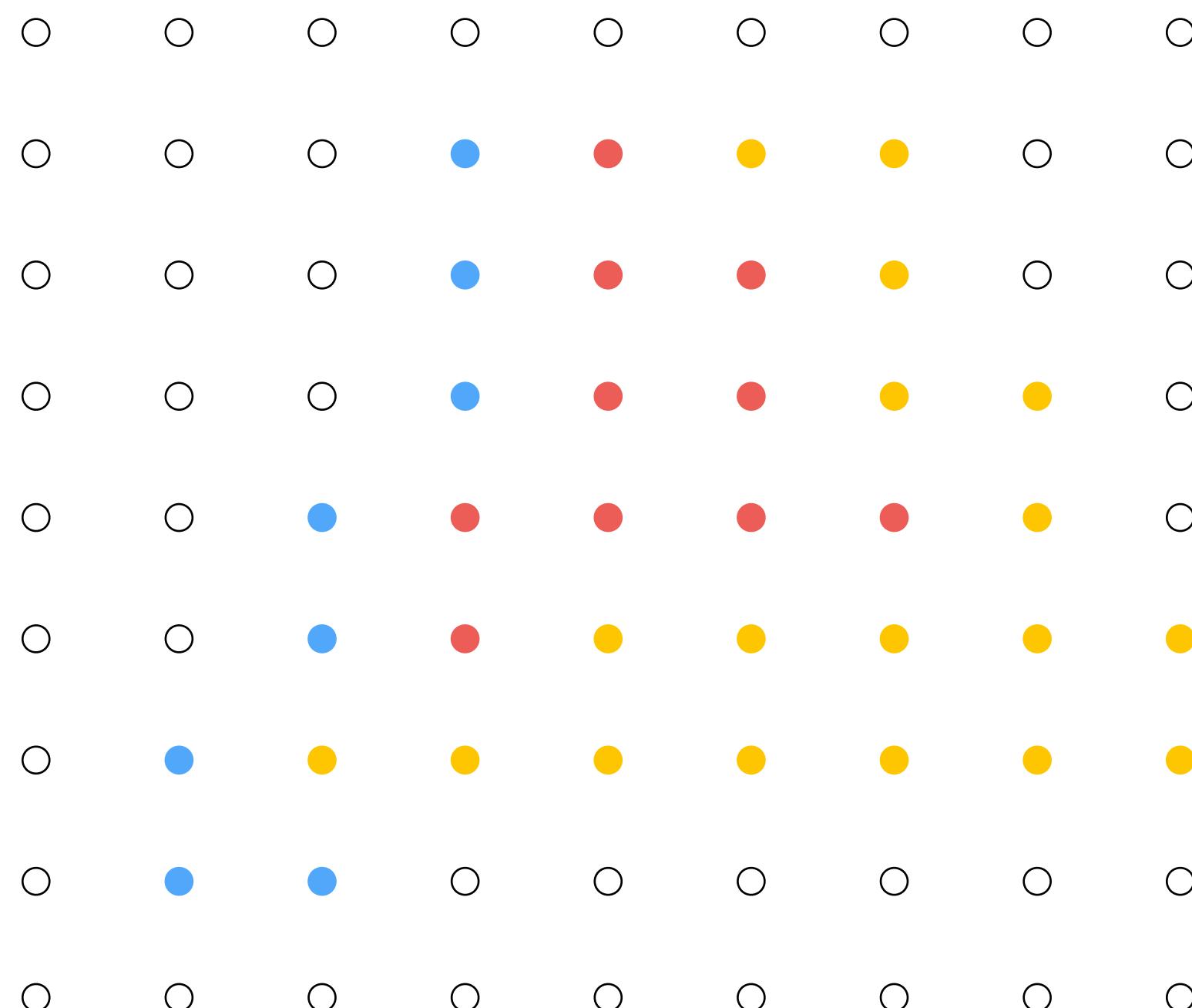
Grayscale value of sample point  
used to indicate distance  
White = large distance  
Black = small distance  
Red = sample passed depth test



Depth buffer contents

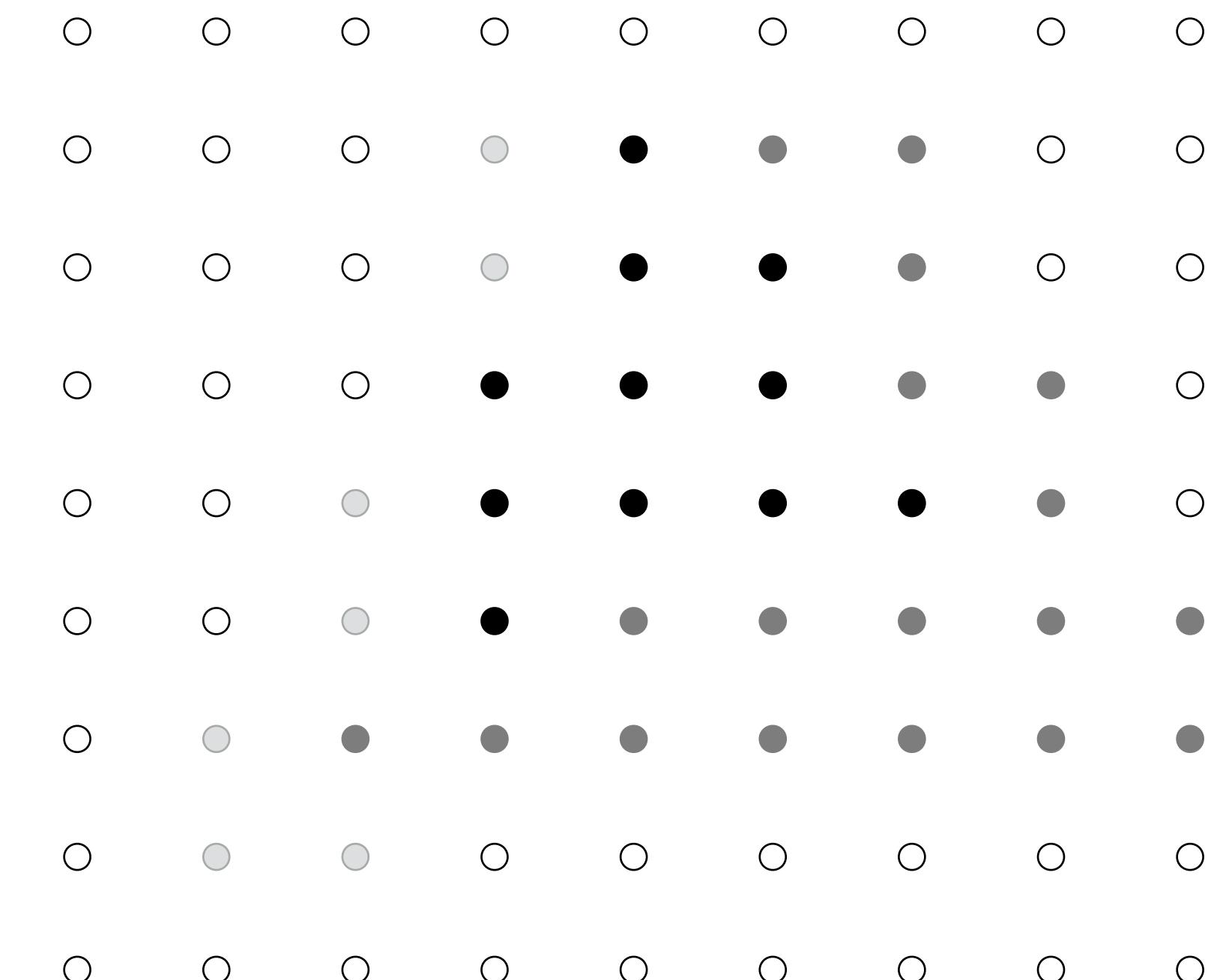
# Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:



Color buffer contents

Grayscale value of sample point  
used to indicate distance  
White = large distance  
Black = small distance  
Red = sample passed depth test



Depth buffer contents

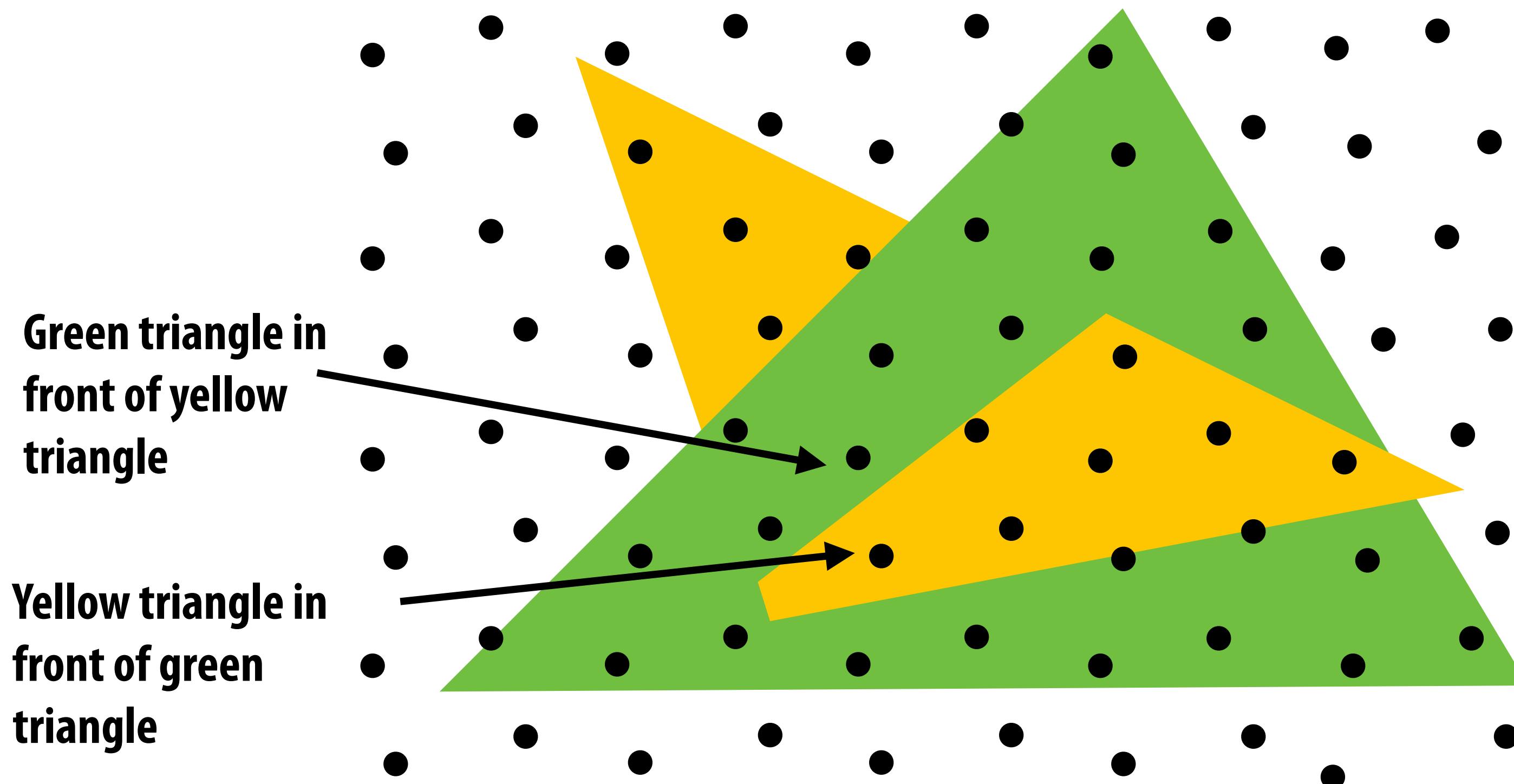
# Occlusion using the depth buffer

```
bool pass_depth_test(d1, d2) {  
    return d1 < d2;  
}  
  
depth_test(tri_d, tri_color, x, y) {  
    if (pass_depth_test(tri_d, zbuffer[x][y])) {  
        // triangle is closest object seen so far at this  
        // sample point. Update depth and color buffers.  
        zbuffer[x][y] = tri_d;      // update zbuffer  
        color[x][y] = tri_color;   // update color buffer  
    }  
}
```

# Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

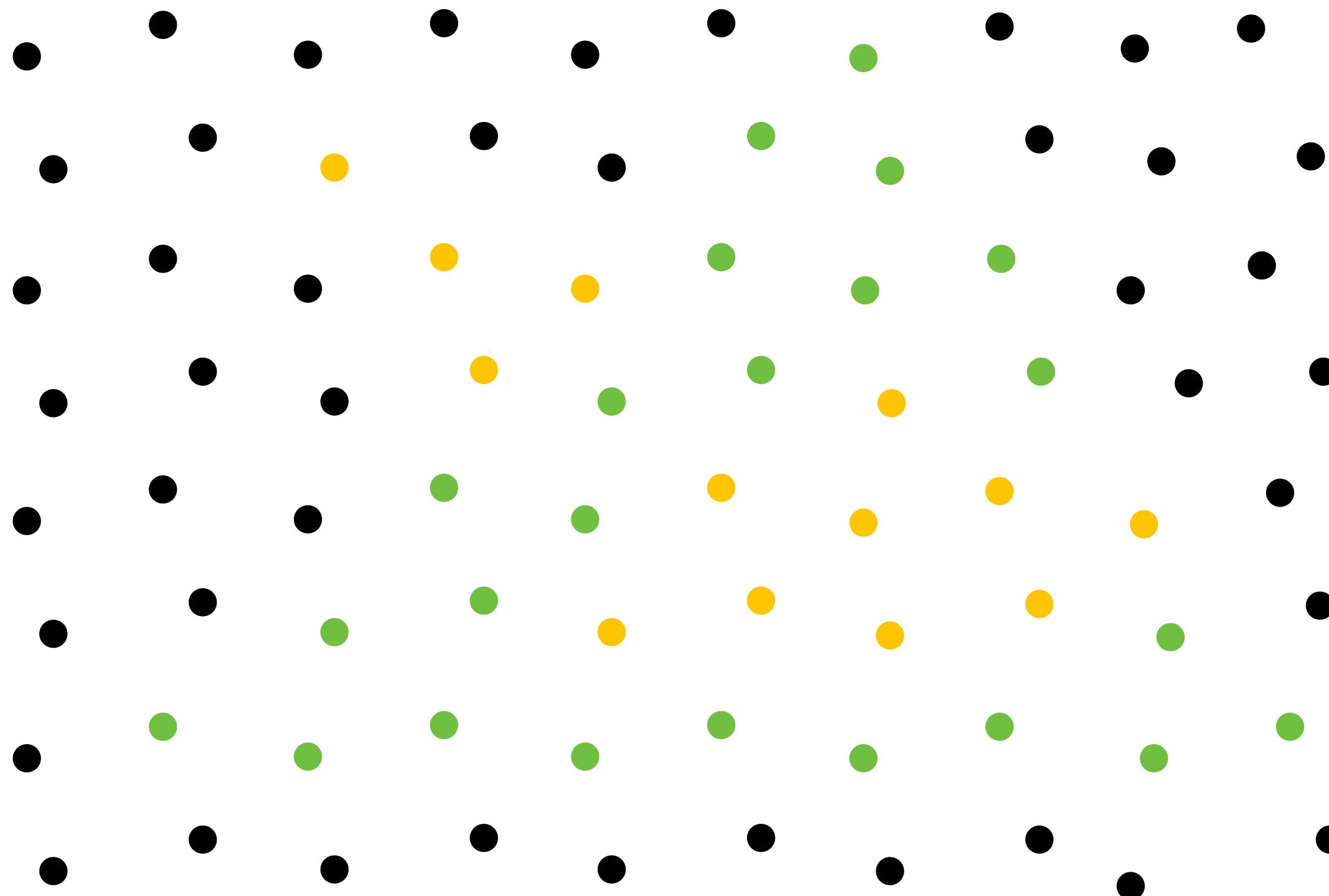
Occlusion test is based on depth of triangles at a given sample point. The relative depth of triangles may be different at different sample points.



# Does depth-buffer algorithm handle interpenetrating surfaces?

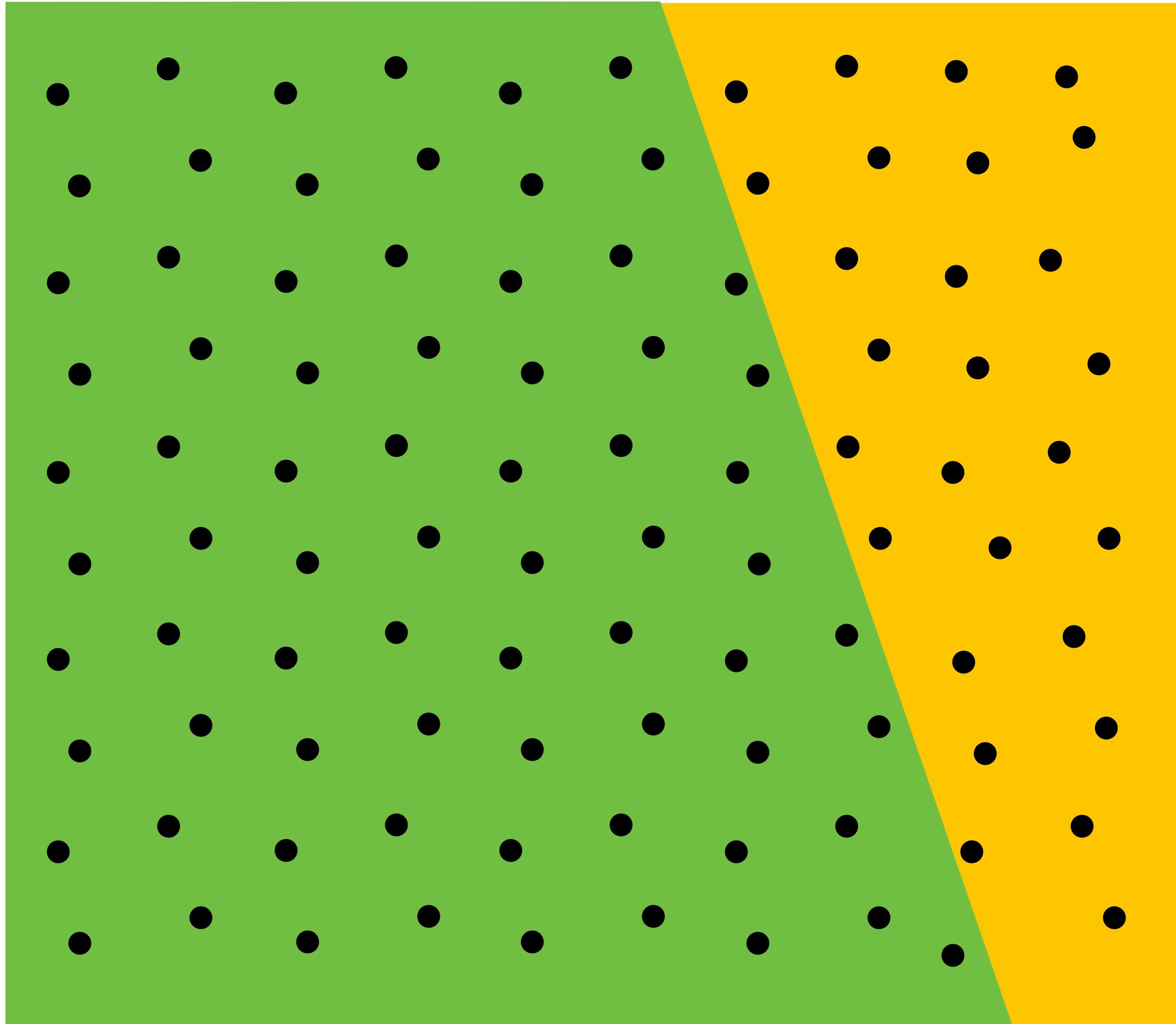
Of course!

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.



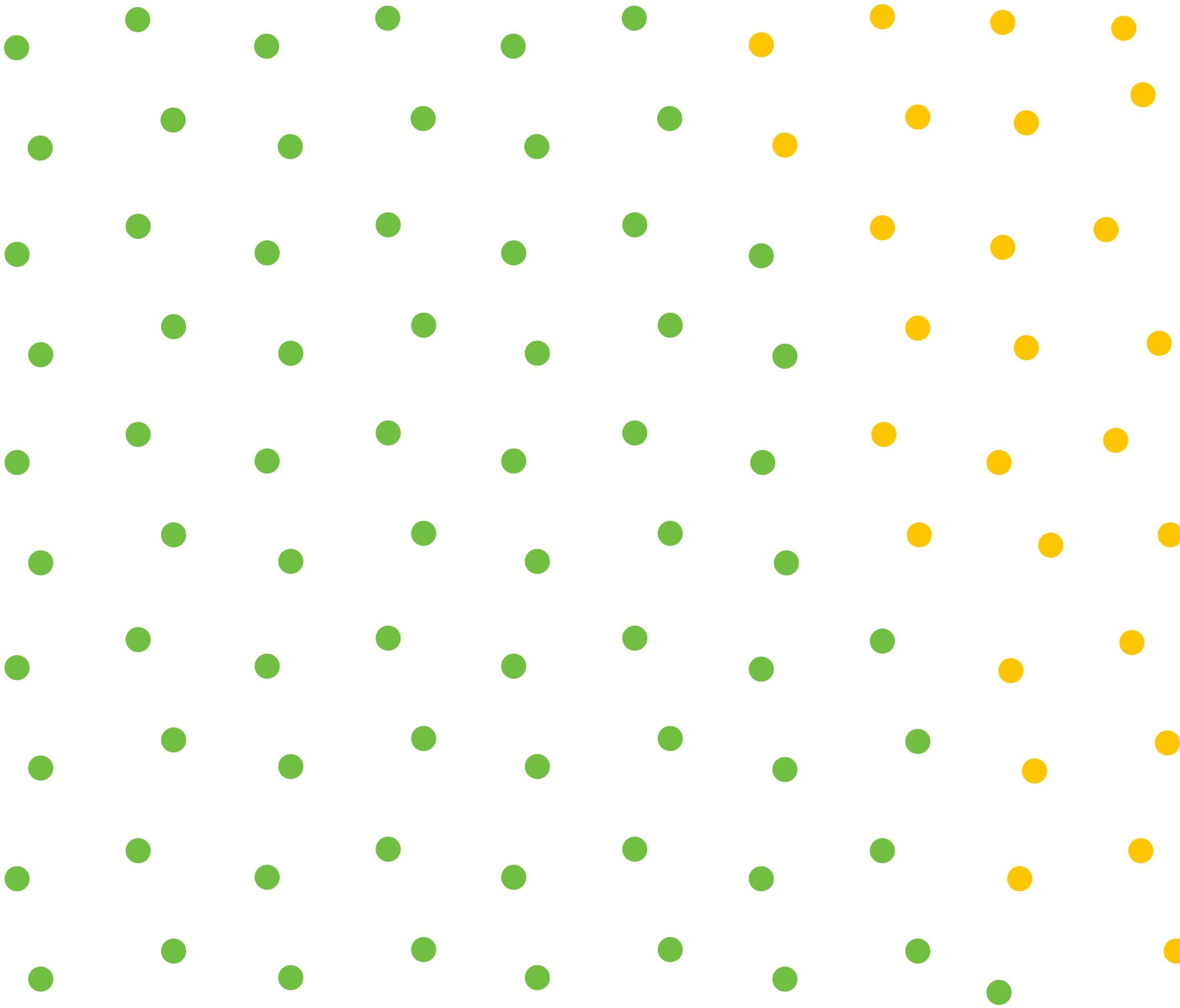
# Does depth buffer work with super sampling?

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

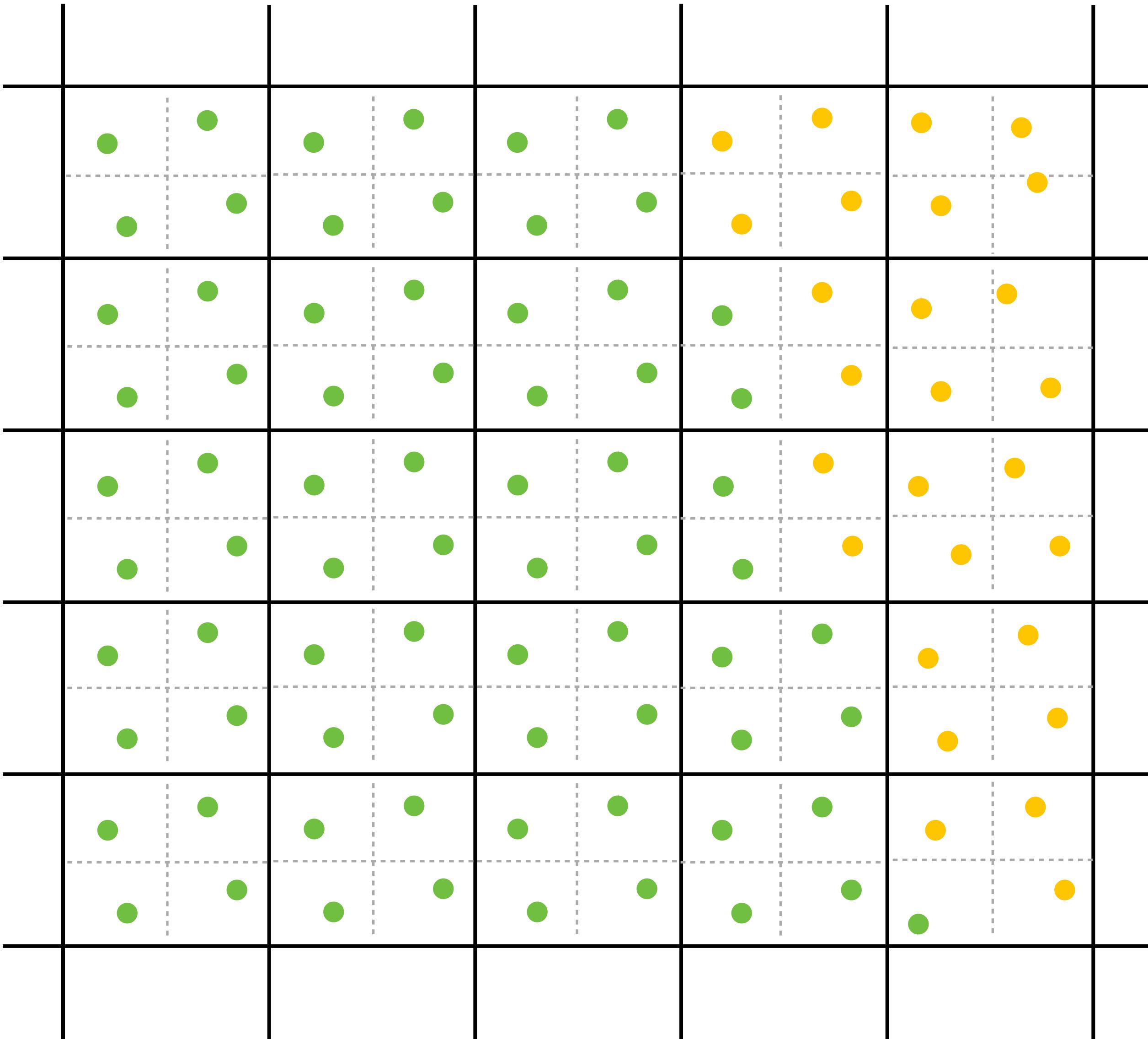


This example: green triangle occludes yellow triangle

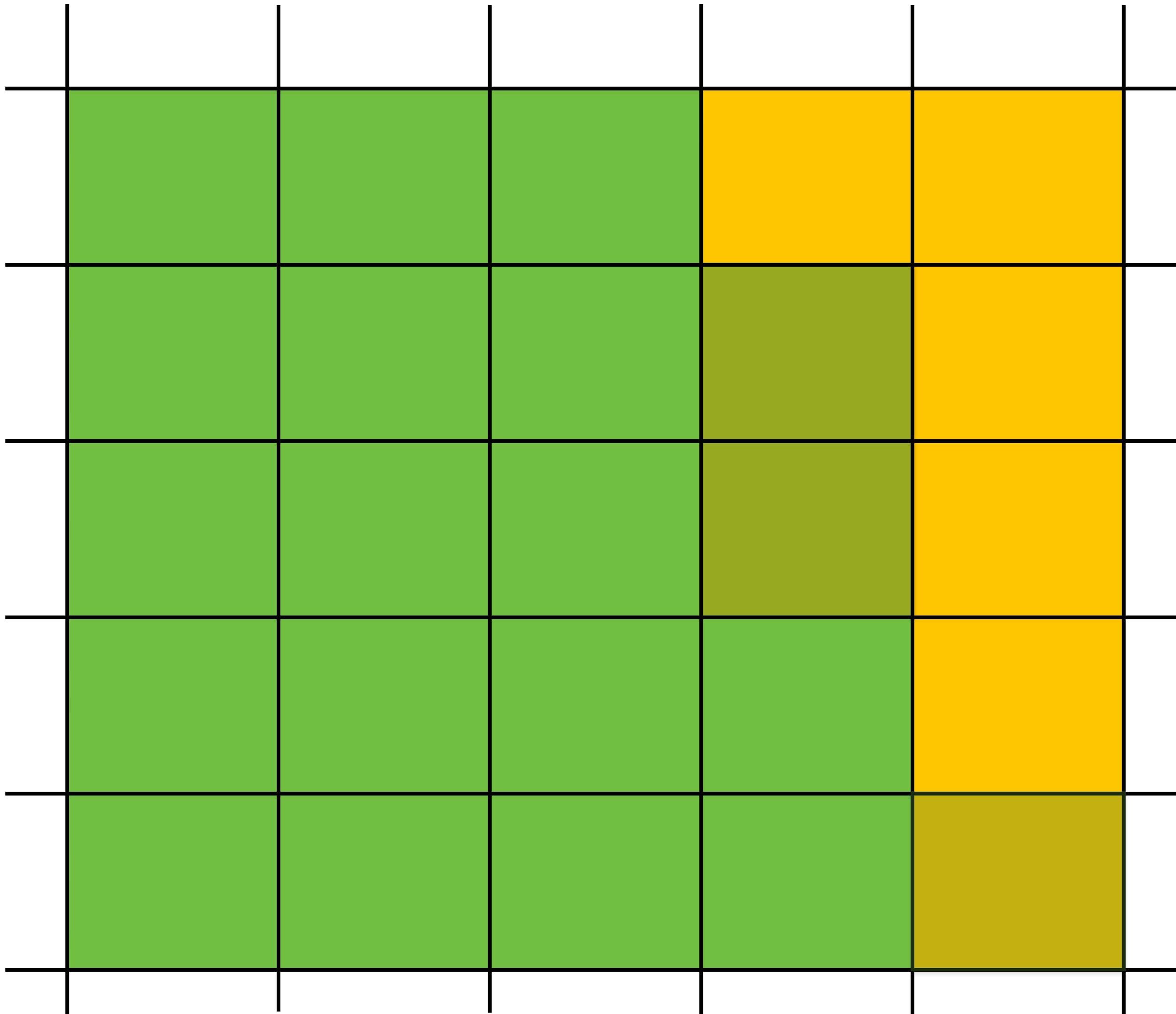
# Color buffer contents



# 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 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?**

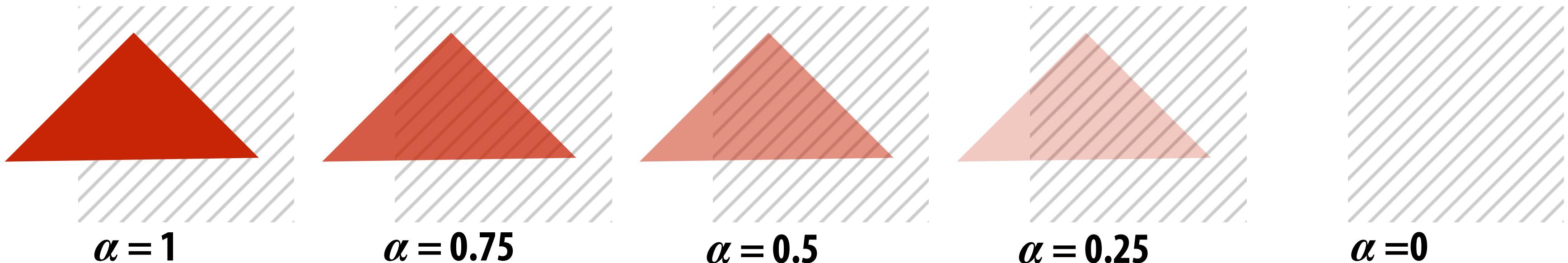
# **Compositing**

# Representing opacity as alpha

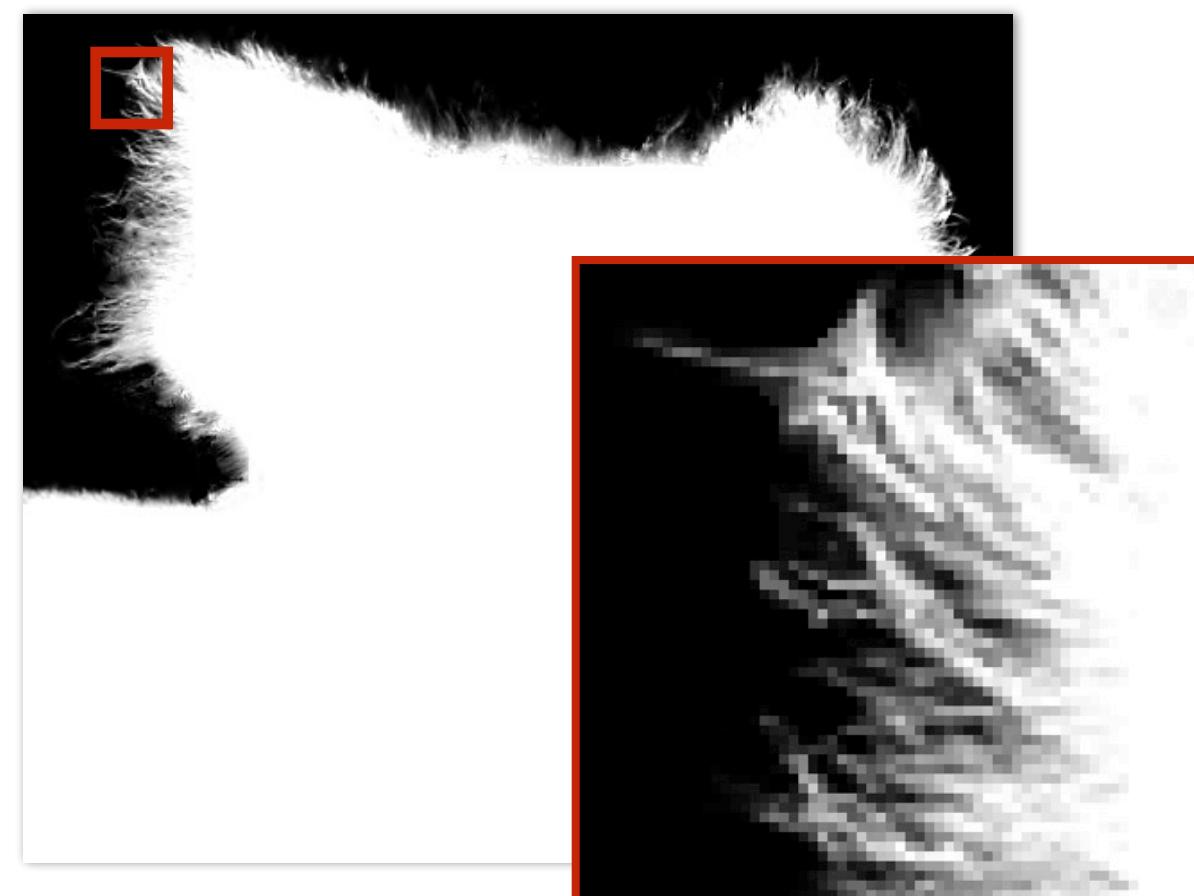
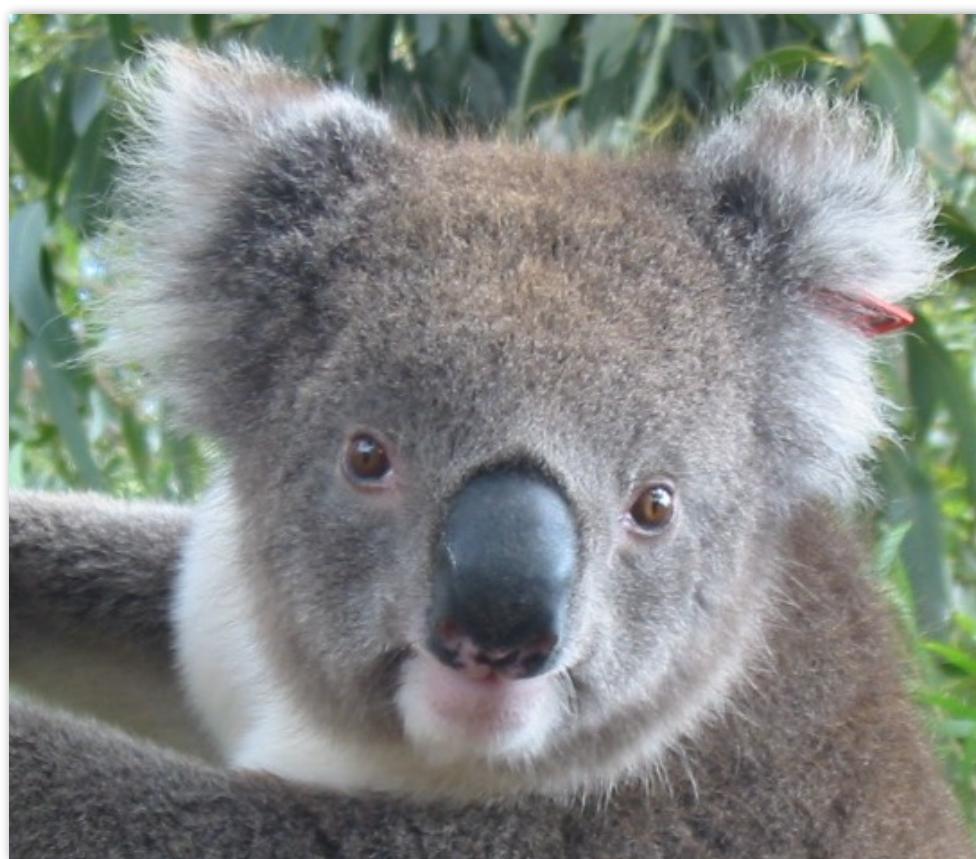
Alpha describes the opacity of an object

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

Red triangle with decreasing opacity



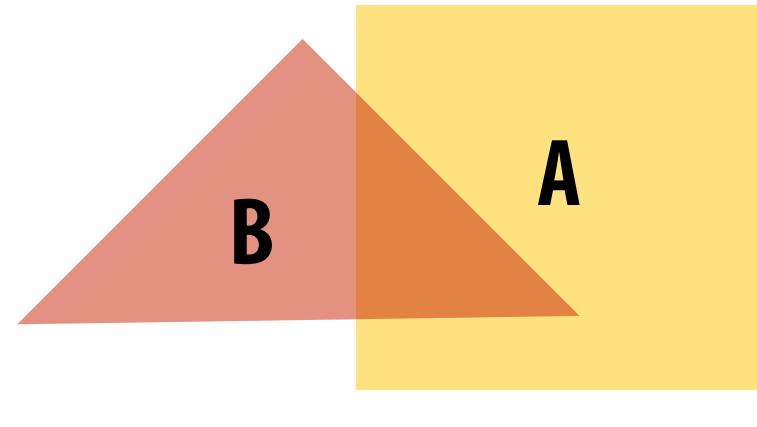
# Alpha: additional channel of image (rgba)



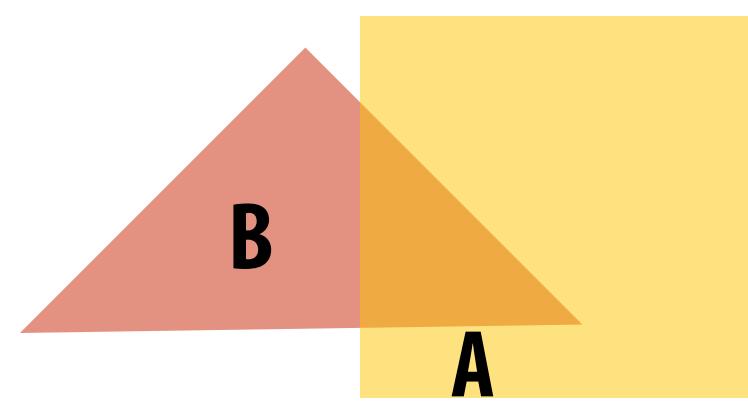
$\alpha$  of foreground object

# Over operator:

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

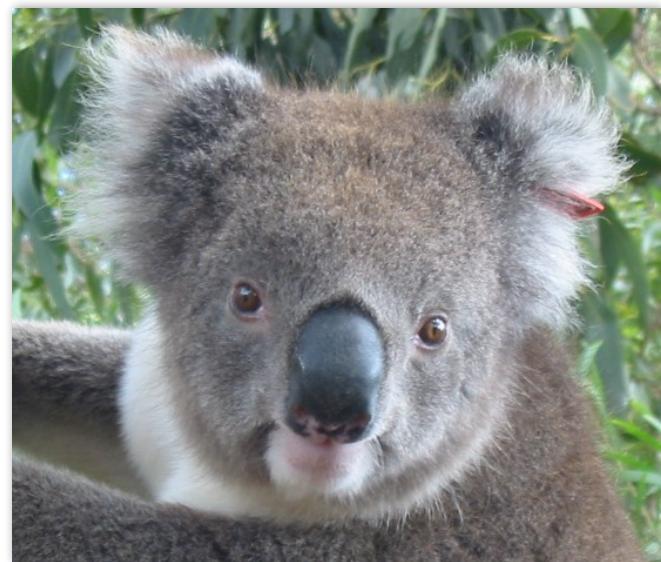
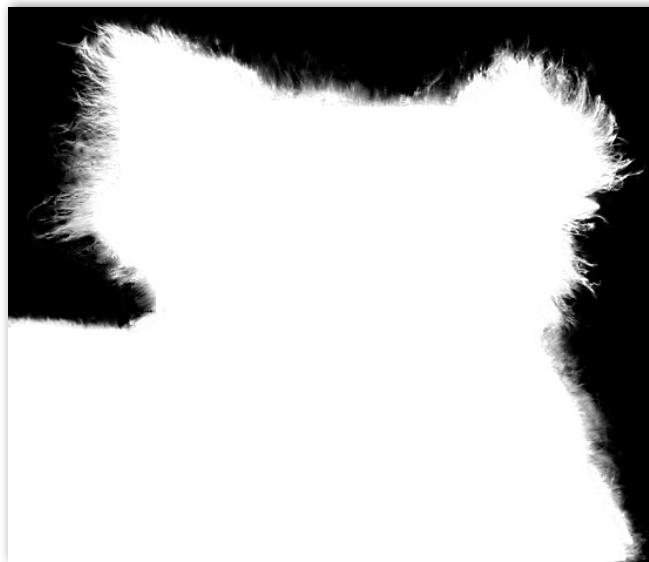


B over A



A over B

$A \text{ over } B \neq B \text{ over } A$   
“Over” is not commutative



Koala over NYC

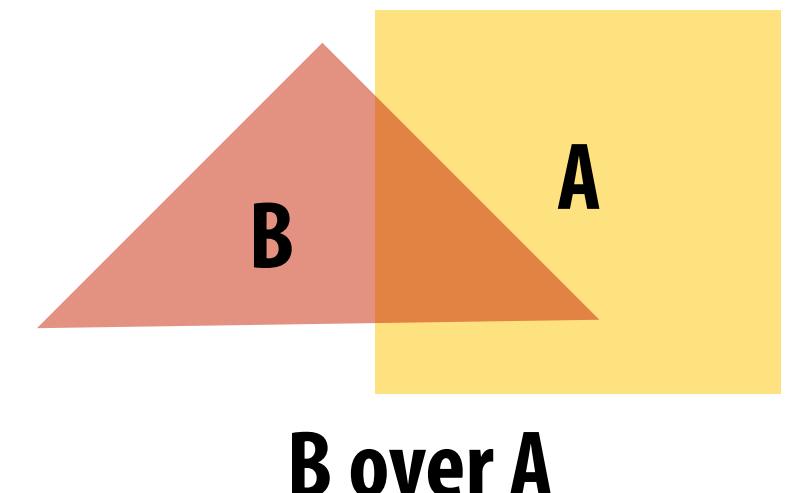
# Over operator: non-premultiplied alpha

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

A first attempt:

$$A = [A_r \quad A_g \quad A_b]^T$$

$$B = [B_r \quad B_g \quad B_b]^T$$



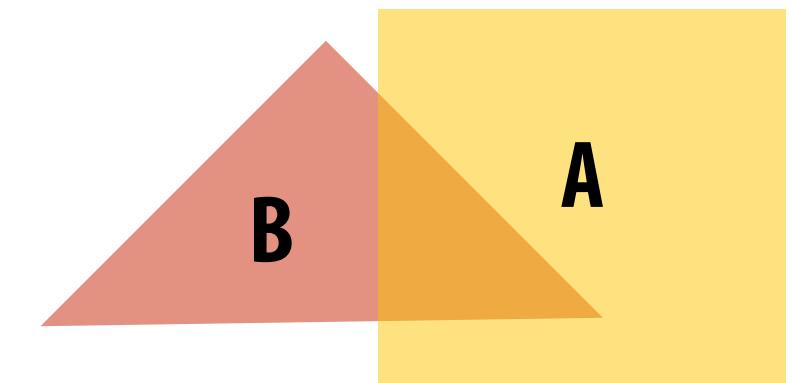
Composed color:

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

Appearance of  
semi-transparent B

What B lets through

Appearance of semi-transparent A



A over B

A over B  $\neq$  B over A

“Over” is not commutative

# Over operator: premultiplied alpha

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

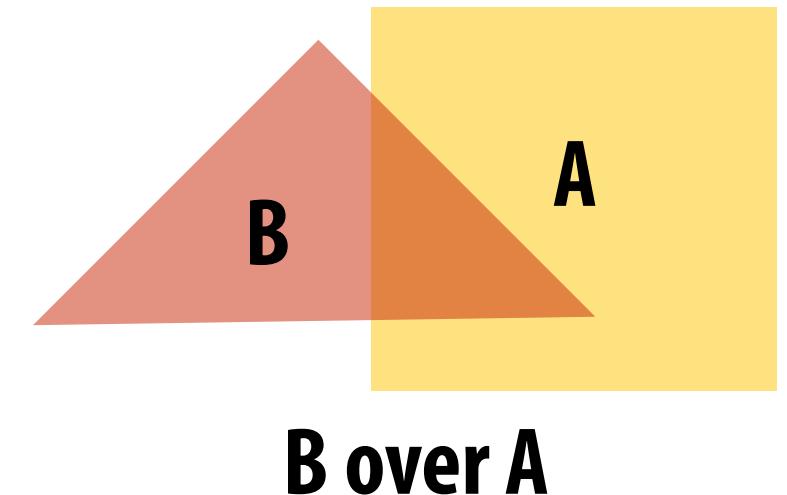
Non-premultiplied alpha:

$$A = [A_r \ A_g \ A_b]^T$$

$$B = [B_r \ B_g \ B_b]^T$$

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

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



Premultiplied alpha:

$$A' = [\alpha_A A_r \ \alpha_A A_g \ \alpha_A A_b \ \alpha_A]^T$$

$$B' = [\alpha_B B_r \ \alpha_B B_g \ \alpha_B B_b \ \alpha_B]^T$$

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

one multiply, one add

Composite alpha:

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

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

# Applying “over” repeatedly

Composite image C with opacity  $\alpha_C$  over B with opacity  $\alpha_B$  over image A with opacity  $\alpha_A$

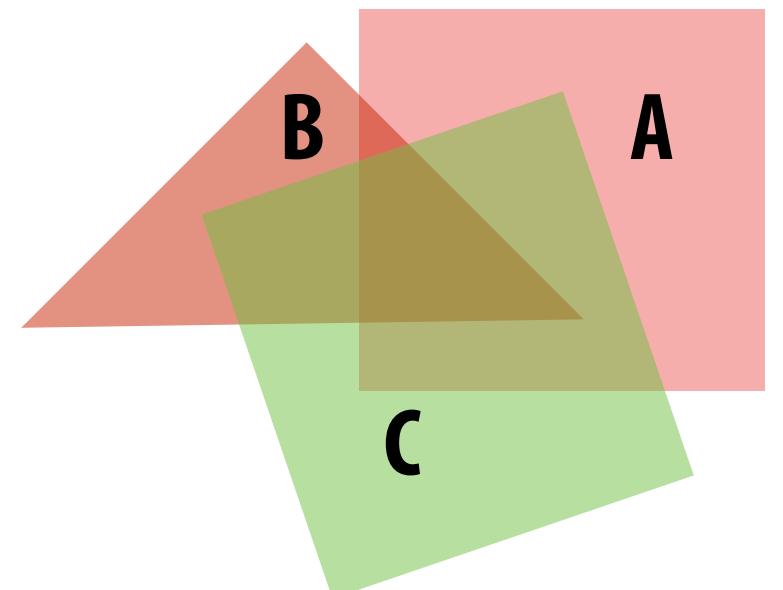
Non-premultiplied alpha is not closed under composition:

$$A = [A_r \quad A_g \quad A_b]^T$$

$$B = [B_r \quad B_g \quad B_b]^T$$

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

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



C over B over A

Consider result of compositing 50% red over 50% red:

$$C = [0.75 \quad 0 \quad 0]^T$$

$$\alpha_C = 0.75$$

Wait... this result is the premultiplied color!

“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))`

Closed form of non-premultiplied alpha:

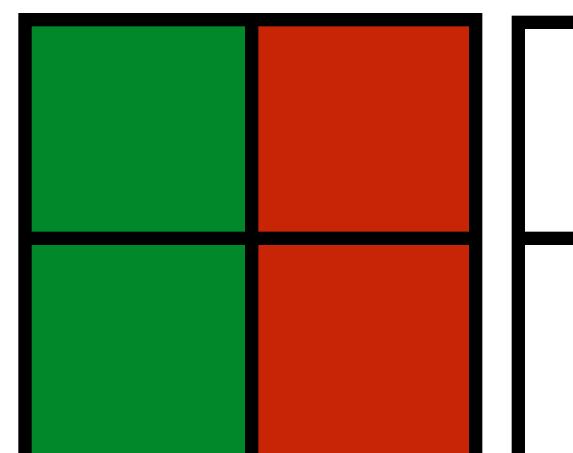
$$C = \frac{1}{\alpha_C} (\alpha_B B + (1 - \alpha_B) \alpha_A A)$$

# Another problem with non-premultiplied alpha

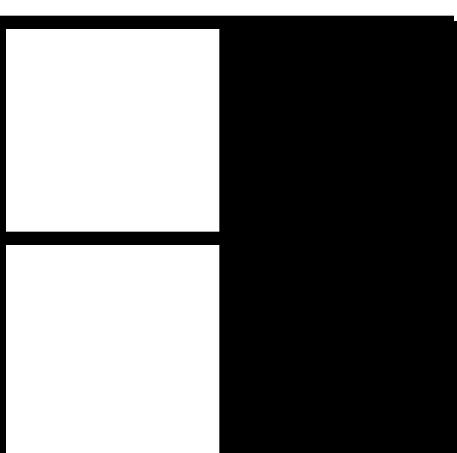
Consider pre-filtering a texture with an alpha matte



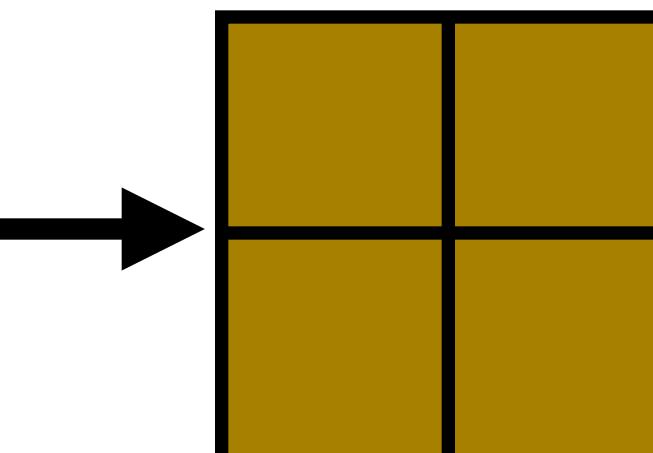
Desired filtered result



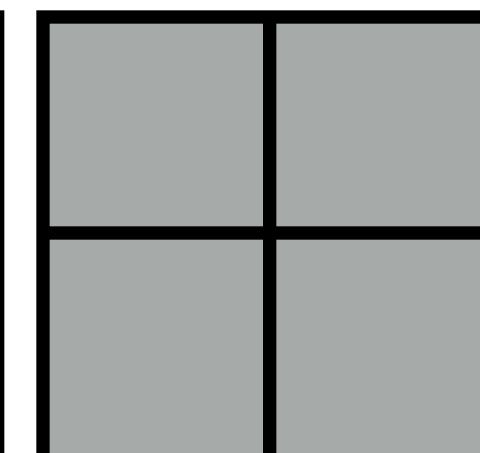
input color



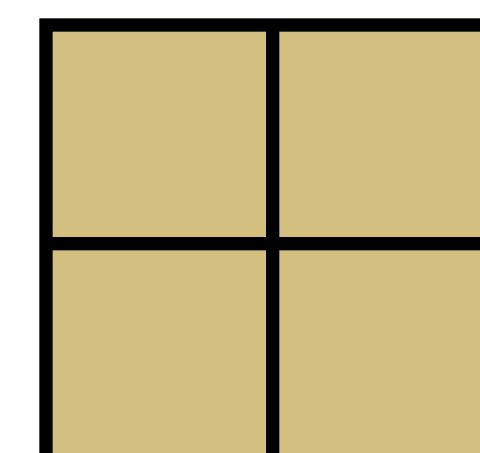
input  $\alpha$



filtered color



filtered  $\alpha$



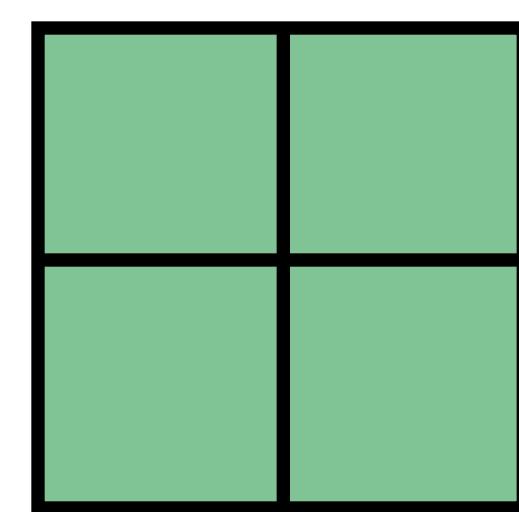
filtered result

composited over white

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

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

Result of filtering  
premultiplied image



# **Summary: advantages of premultiplied alpha**

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

# Color buffer update: semi-transparent surfaces

**Color buffer values and tri\_color are represented with premultiplied alpha**

```
over(c1, c2) {  
    return c1 + (1-c1.a) * c2;  
}  
  
update_color_buffer(tri_d, tri_color, x, y) {  
  
    if (pass_depth_test(tri_d, zbuffer[x][y]) {  
        // update color buffer  
        // Note: 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

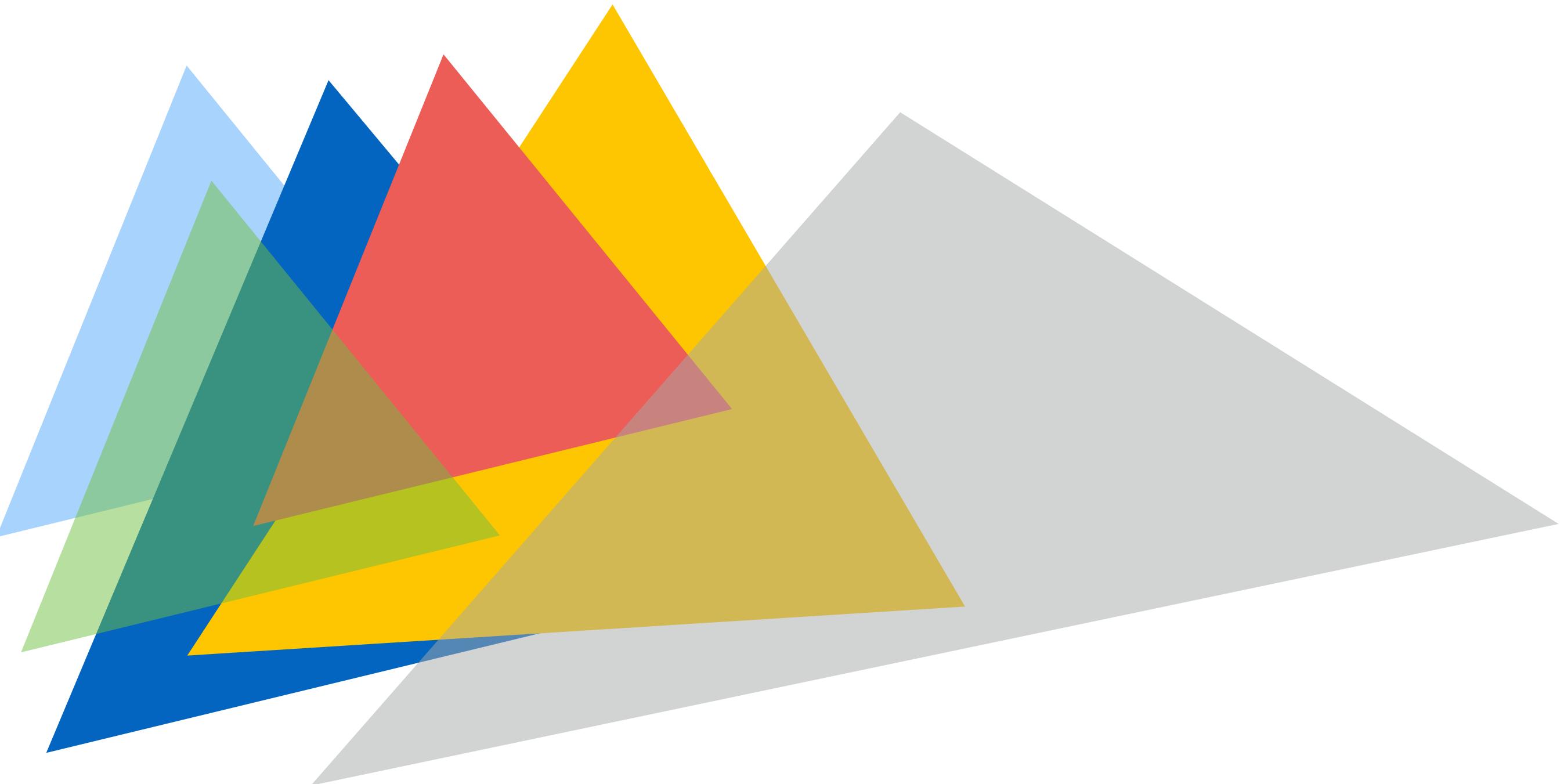
Consider rendering a mixture of opaque and transparent triangles

**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



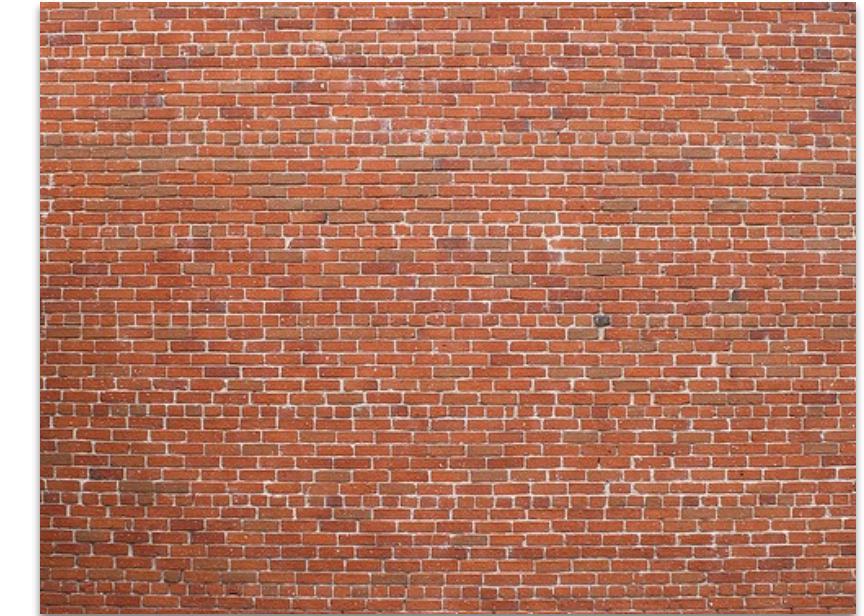
# **End-to-end rasterization pipeline**

## **(“real-time graphics pipeline”)**

# Command: draw these triangles!

Inputs:

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



Texture map

Object-to-camera-space transform:  $T$

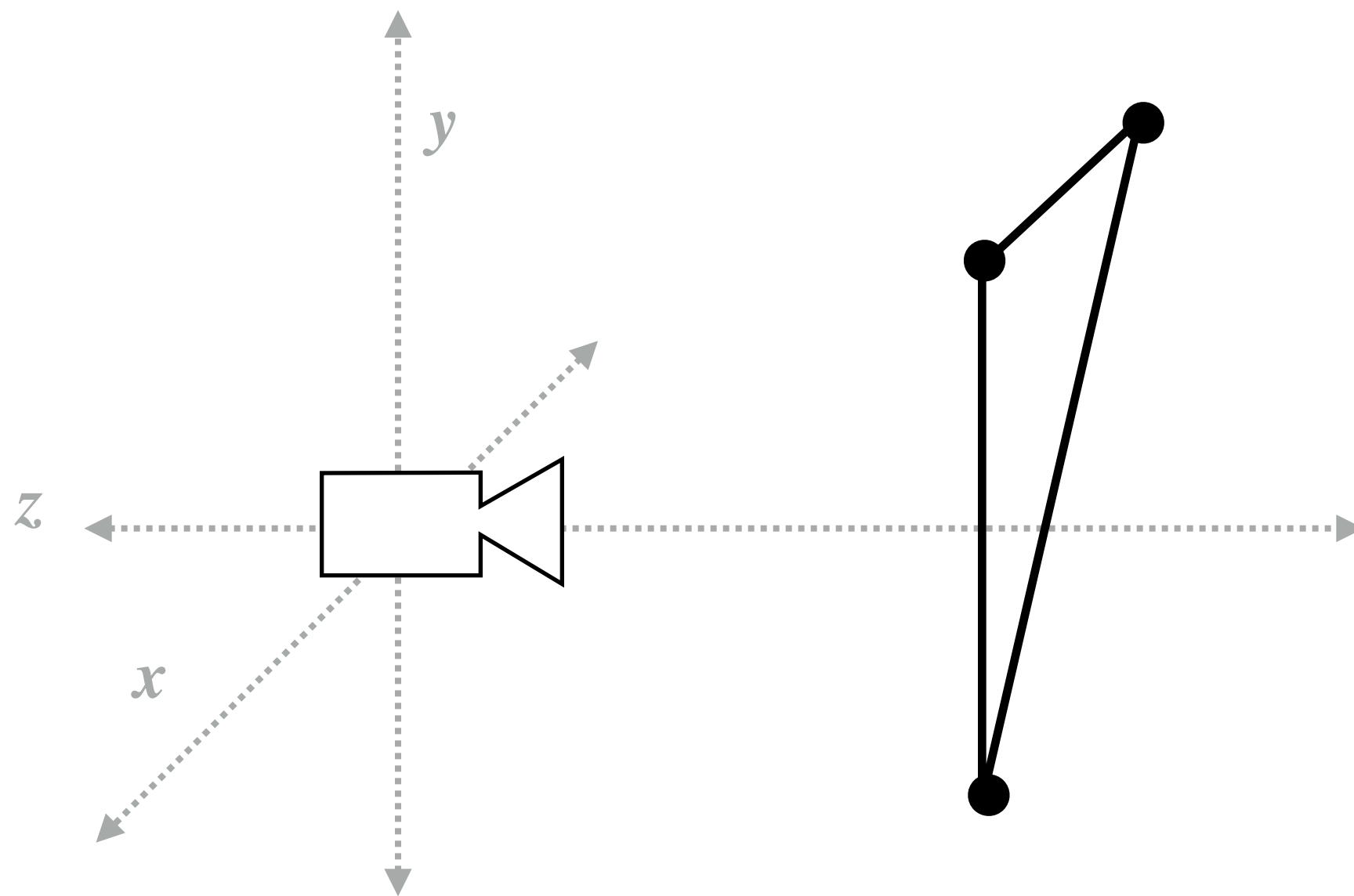
Perspective projection transform  $P$

Size of output image ( $W, H$ )

Use depth test /update depth buffer: YES!

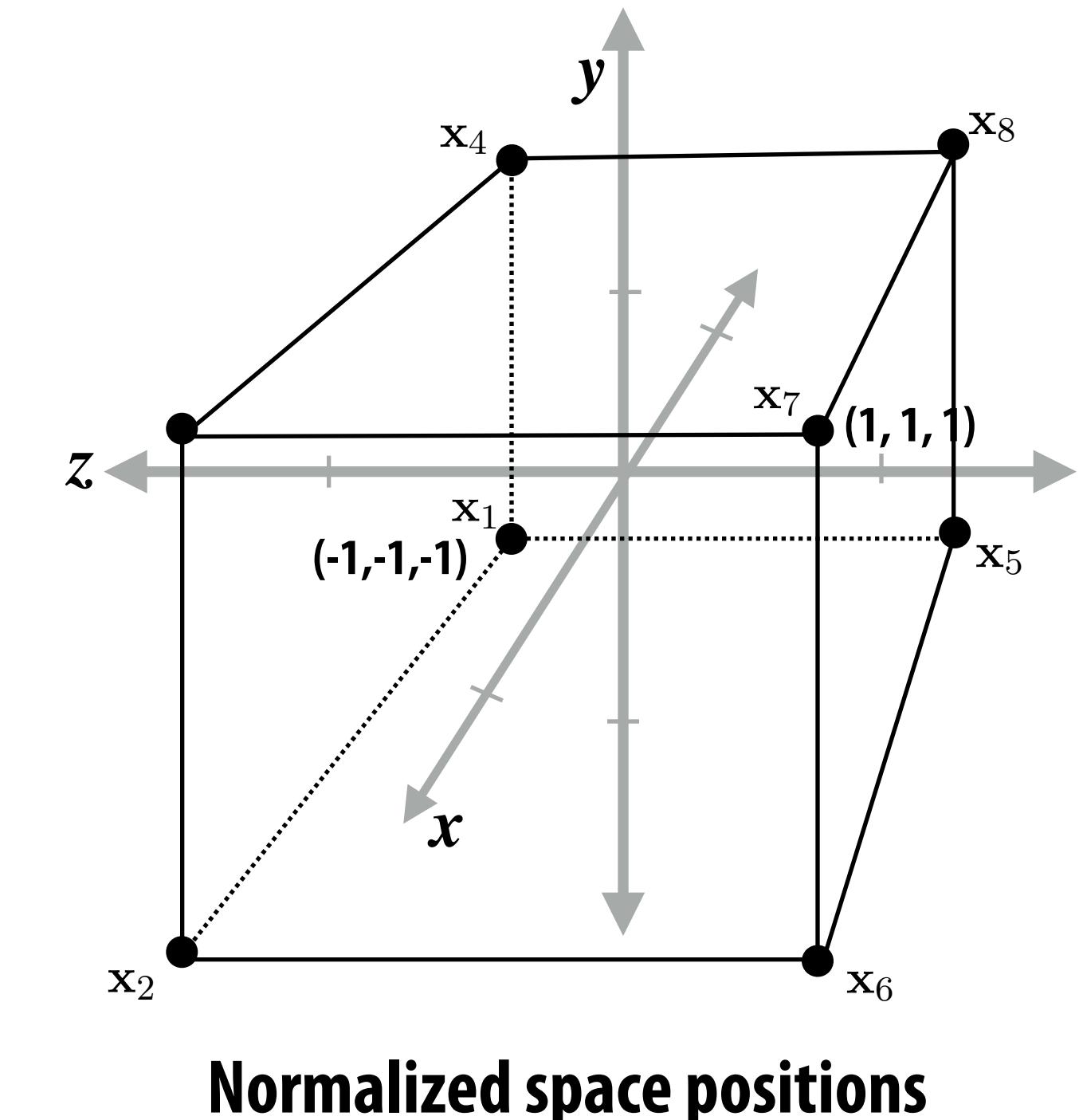
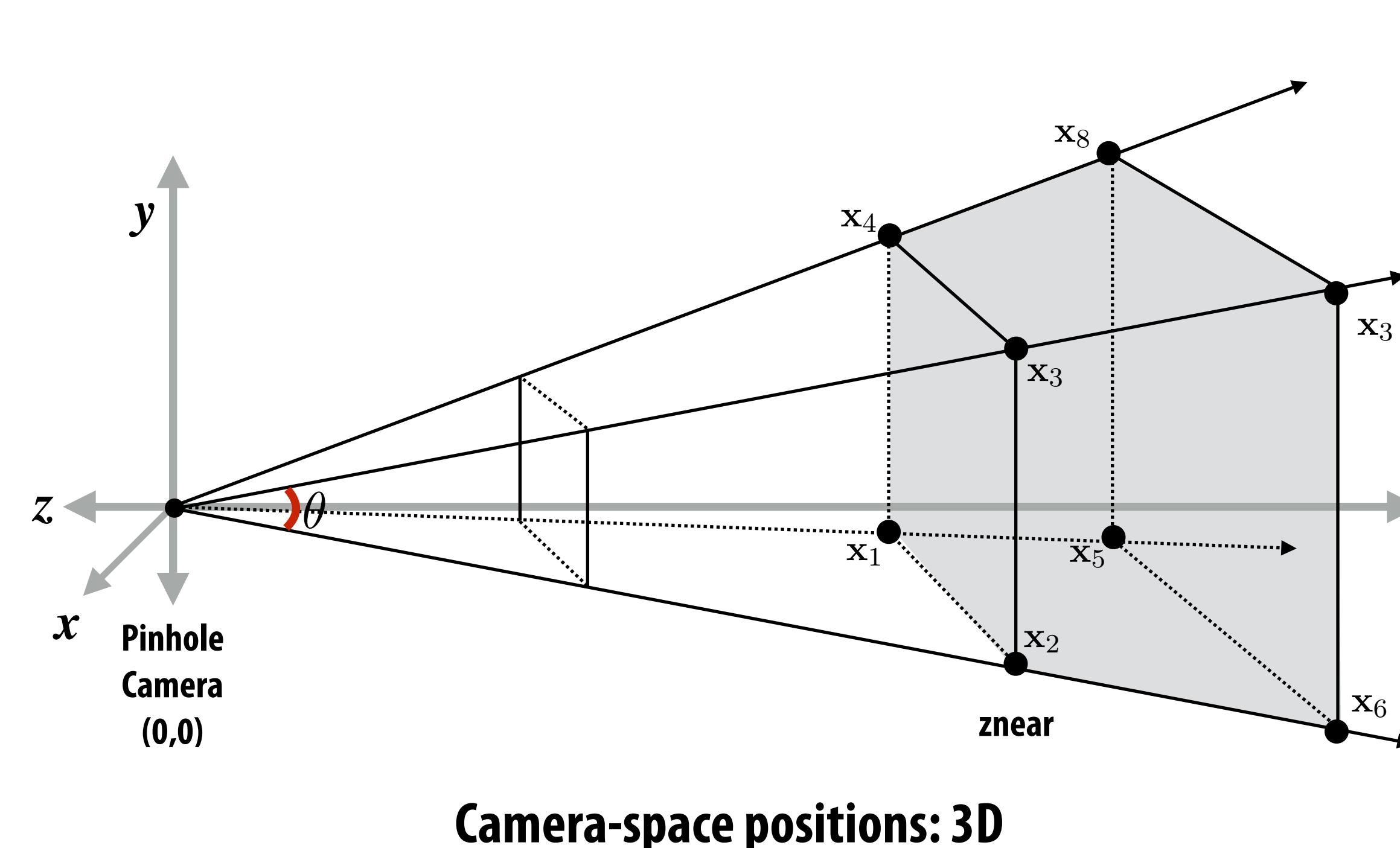
# Step 1:

## Transform triangle vertices into camera space



# Step 2:

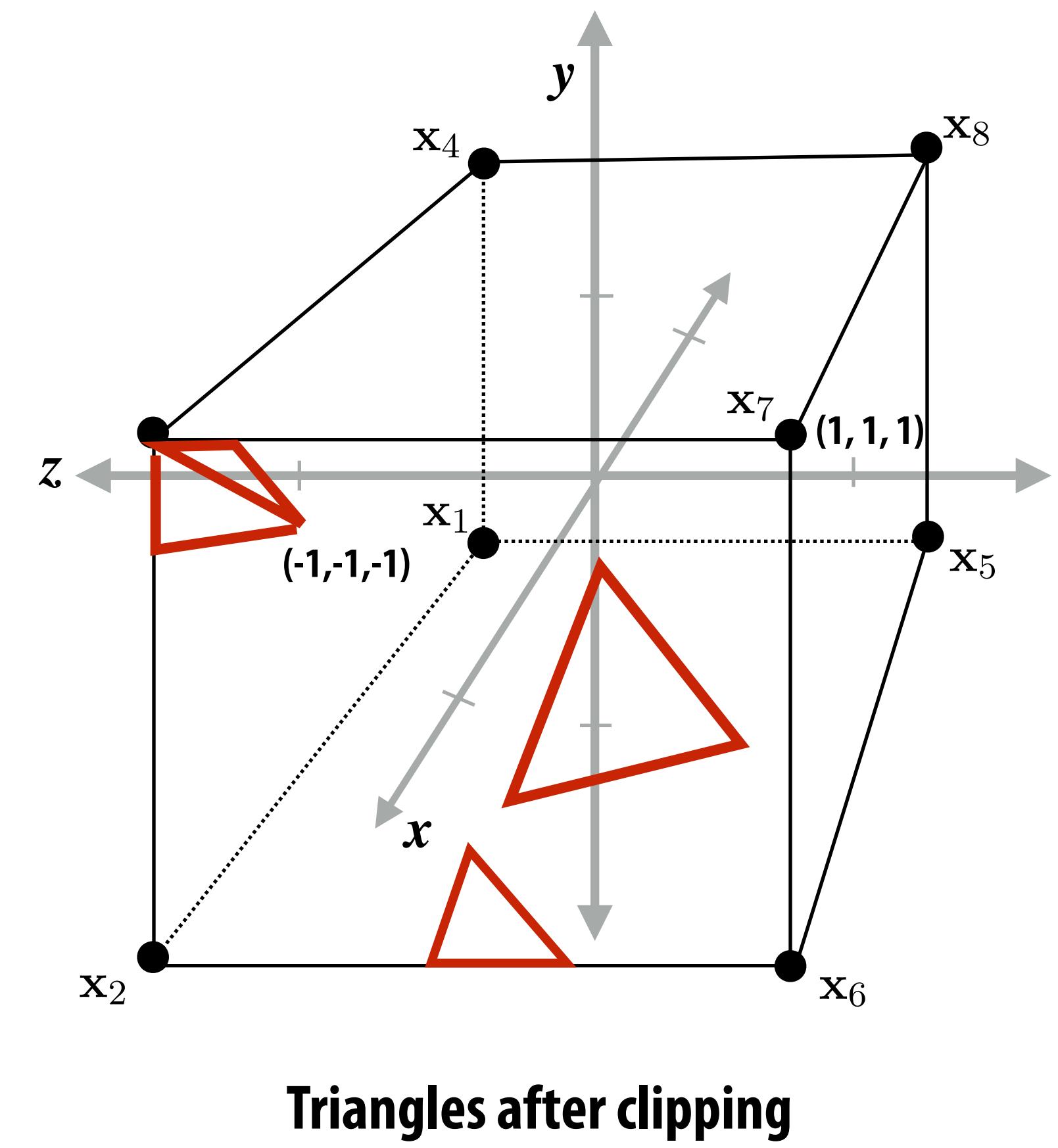
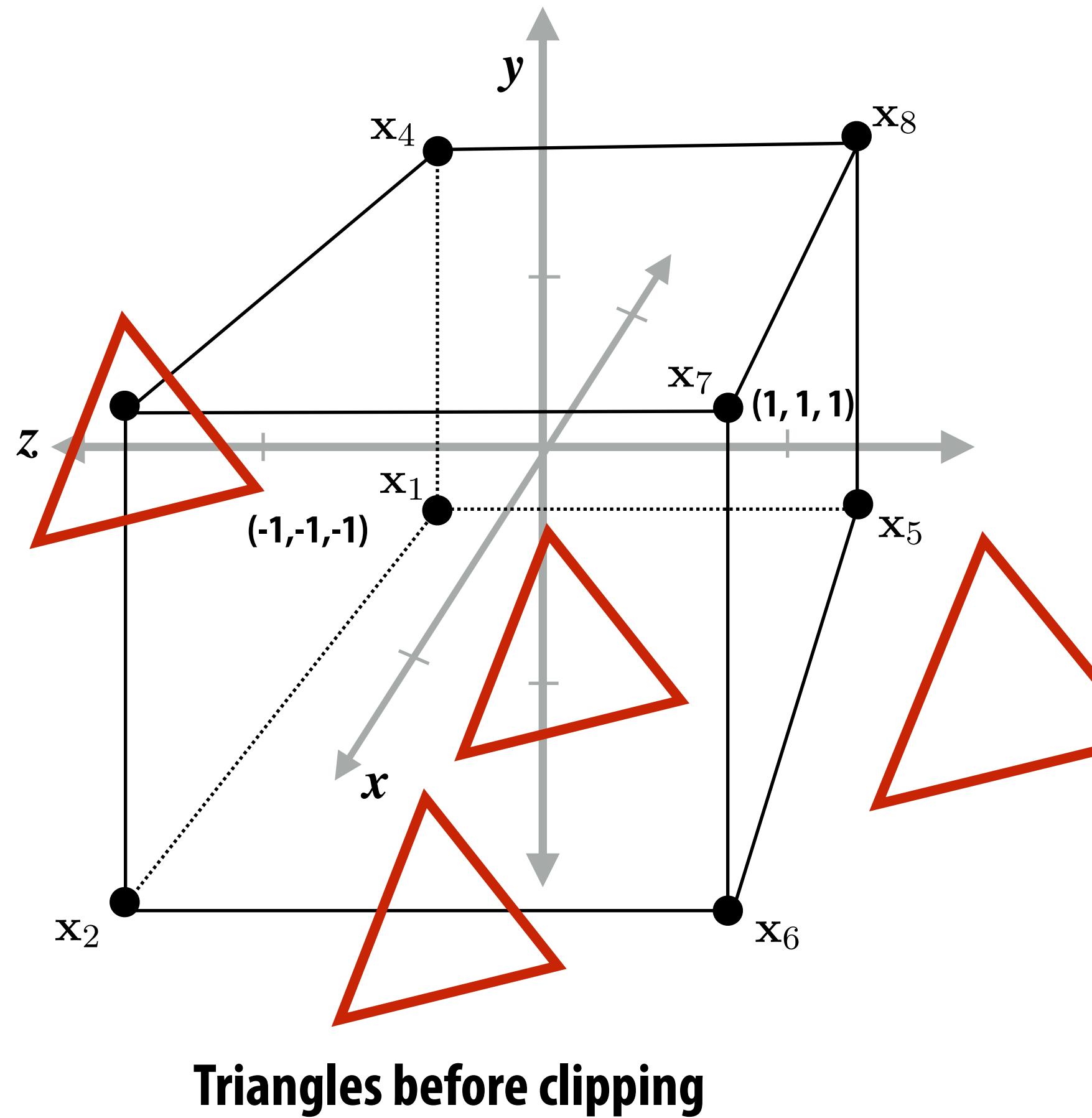
Apply perspective projection transform to transform triangle vertices into normalized coordinate space



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  $(w, w, w, w)$

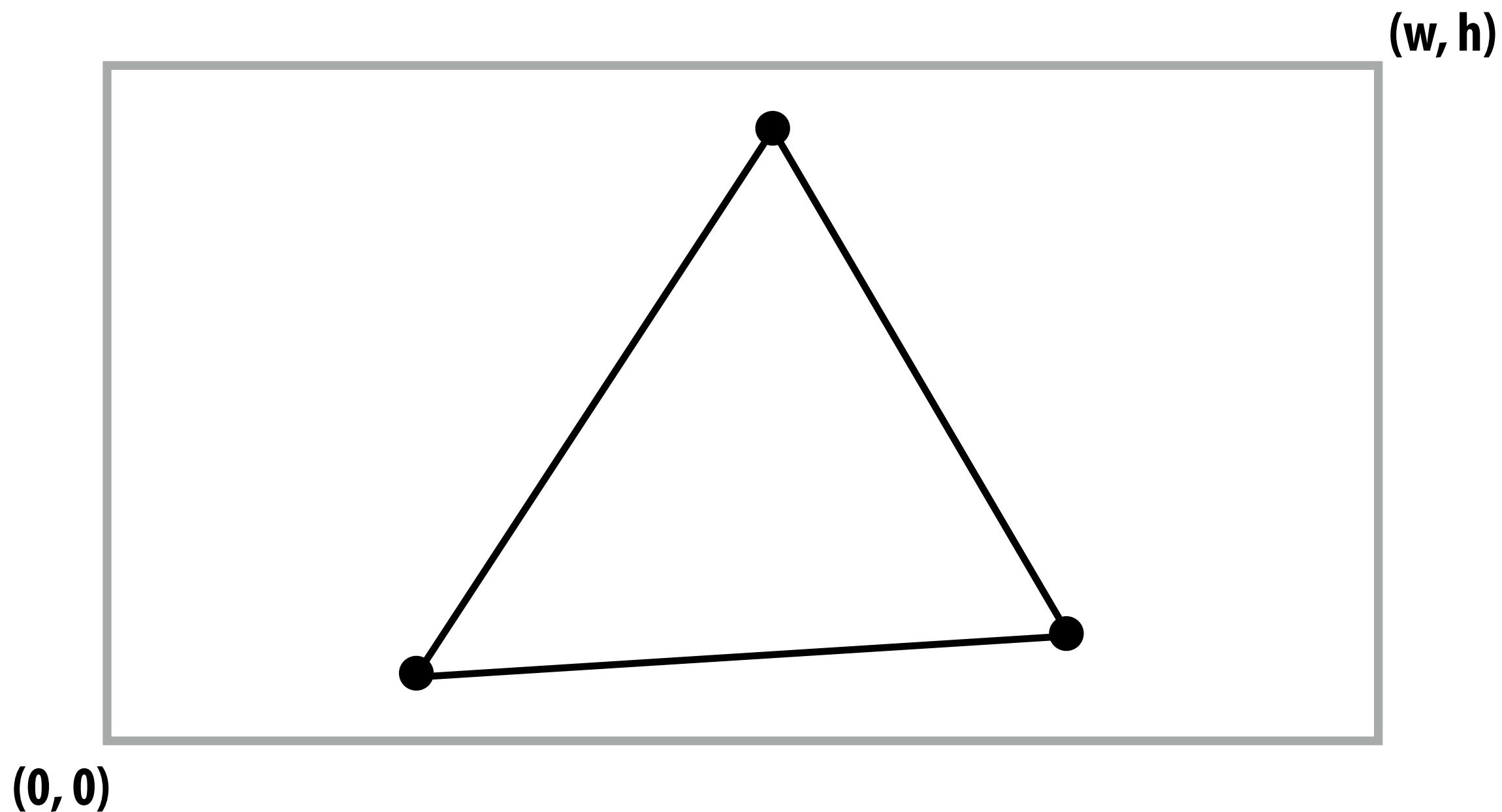
# Step 3: clipping

- Discard triangles that lie completely 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



# 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

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

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

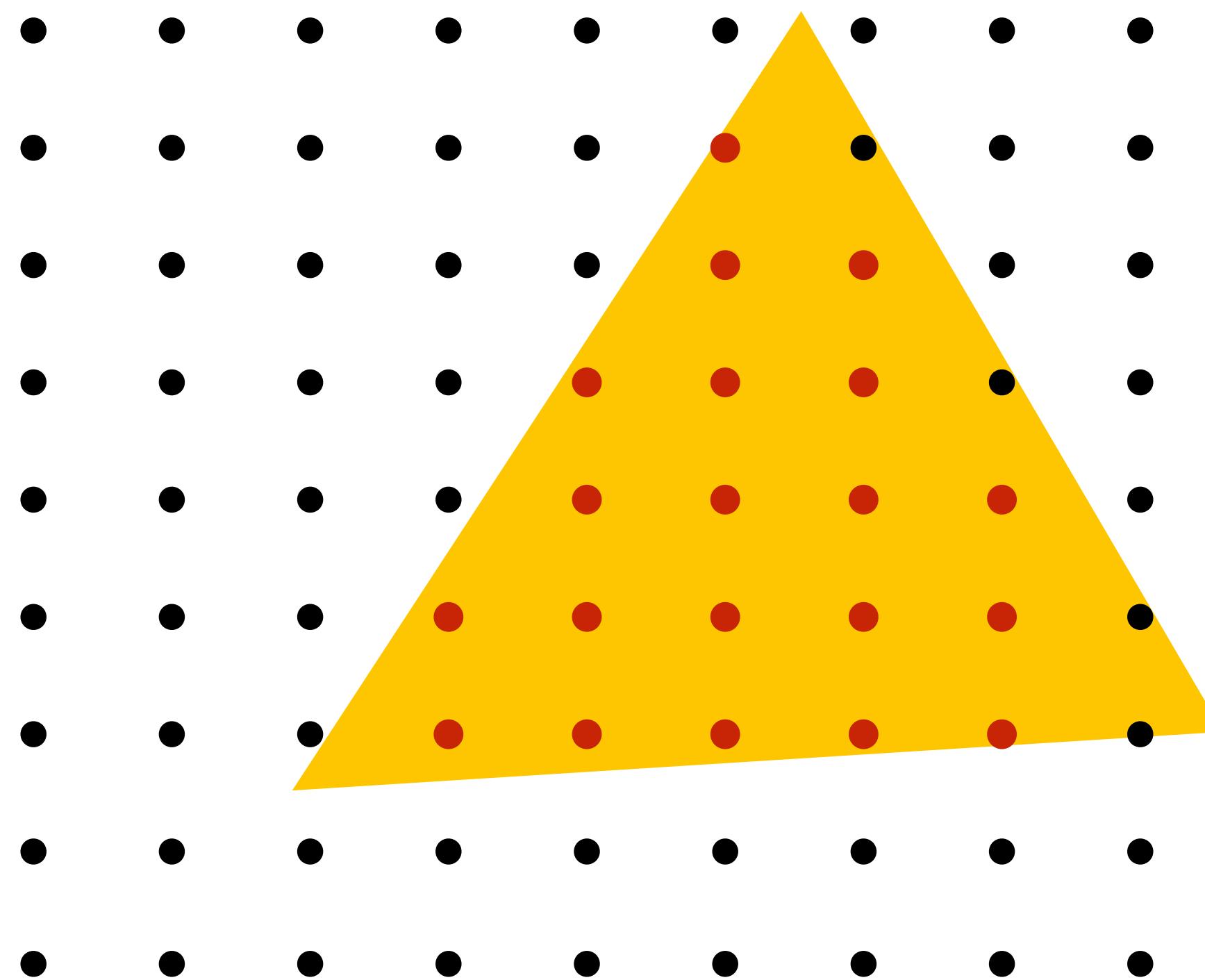
$$\mathbf{E}_{20}(x, y)$$

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

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

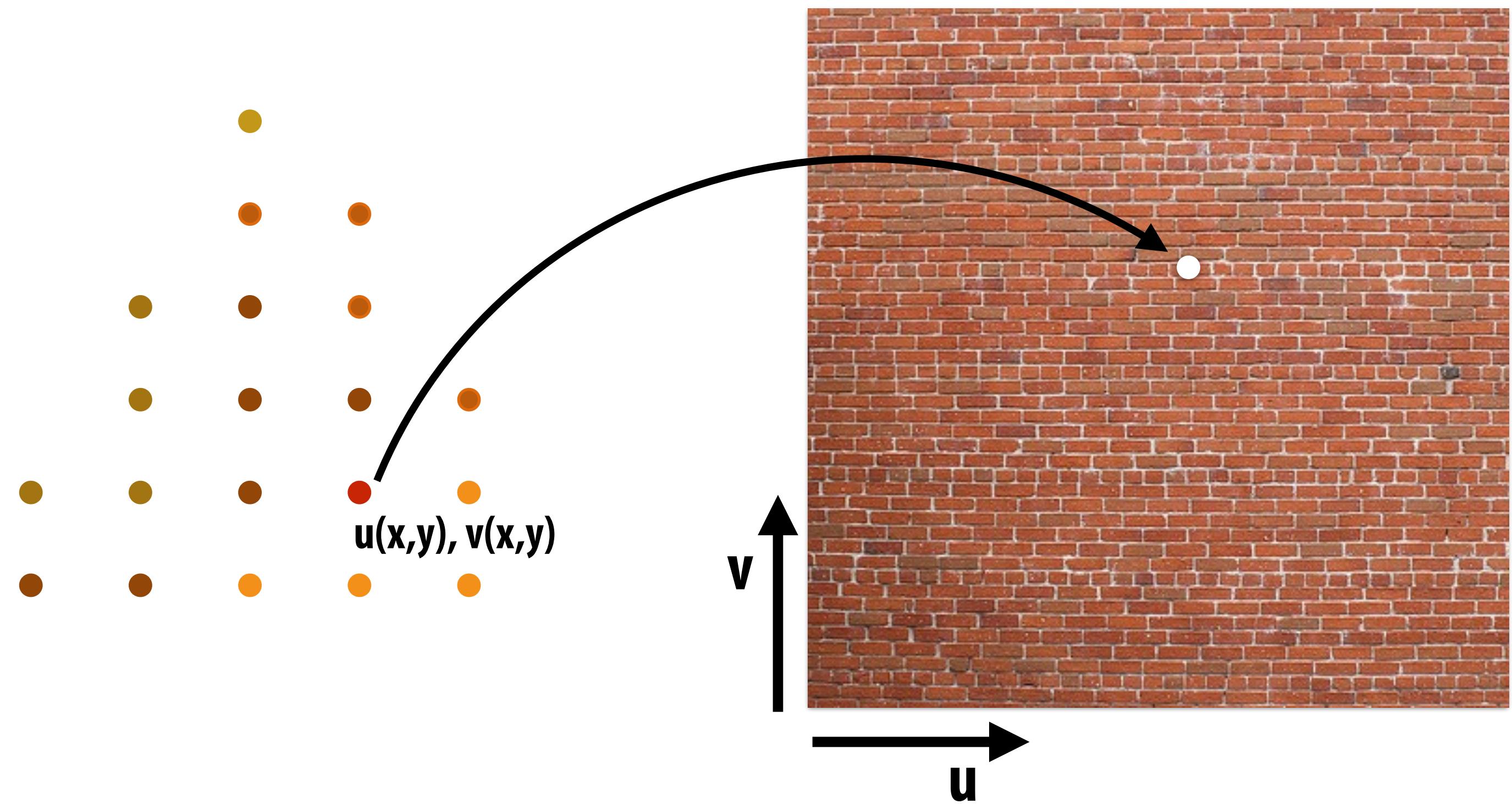
# Step 6: sample coverage

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



# Step 6: compute triangle color at sample point

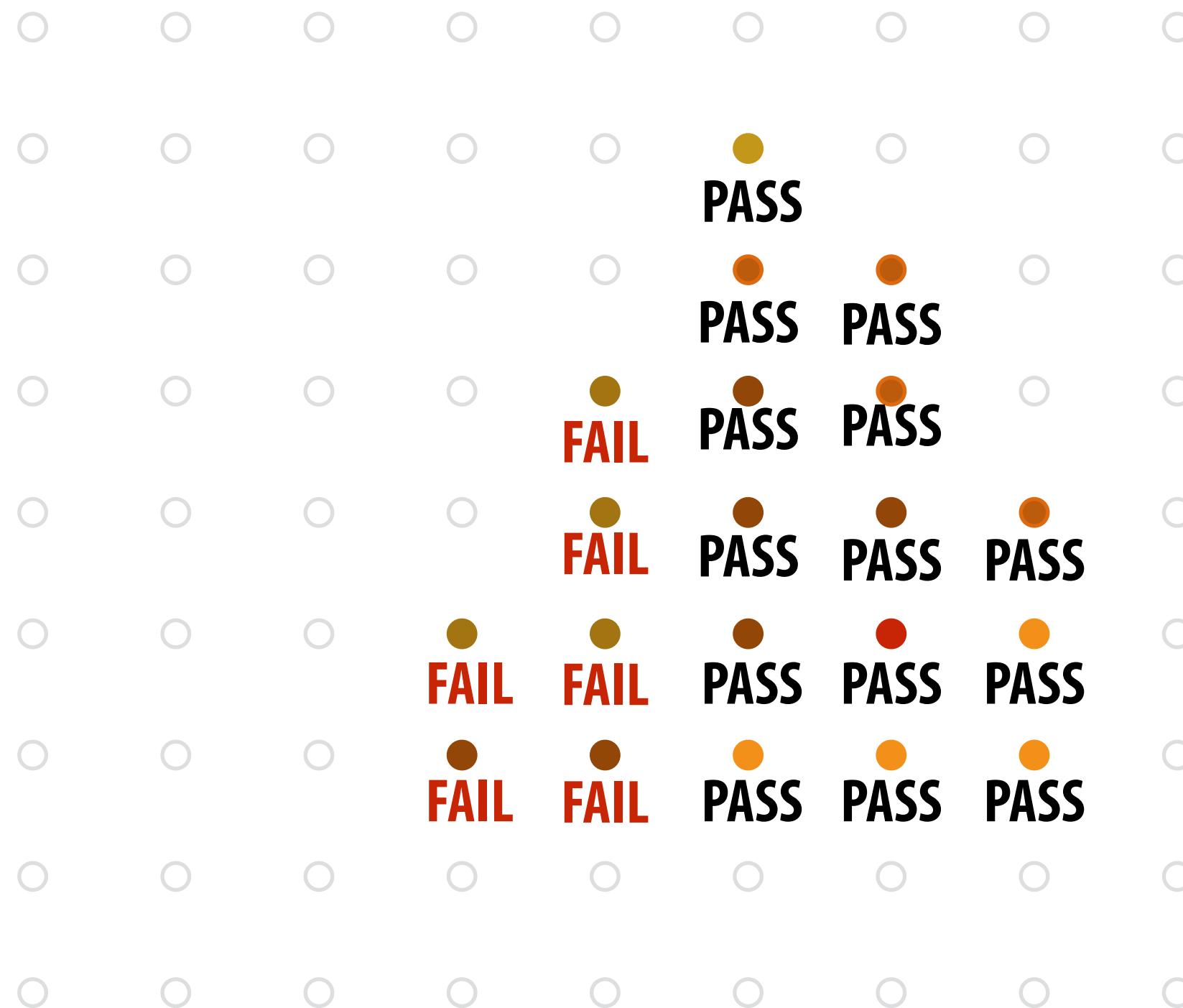
e.g., sample texture map \*



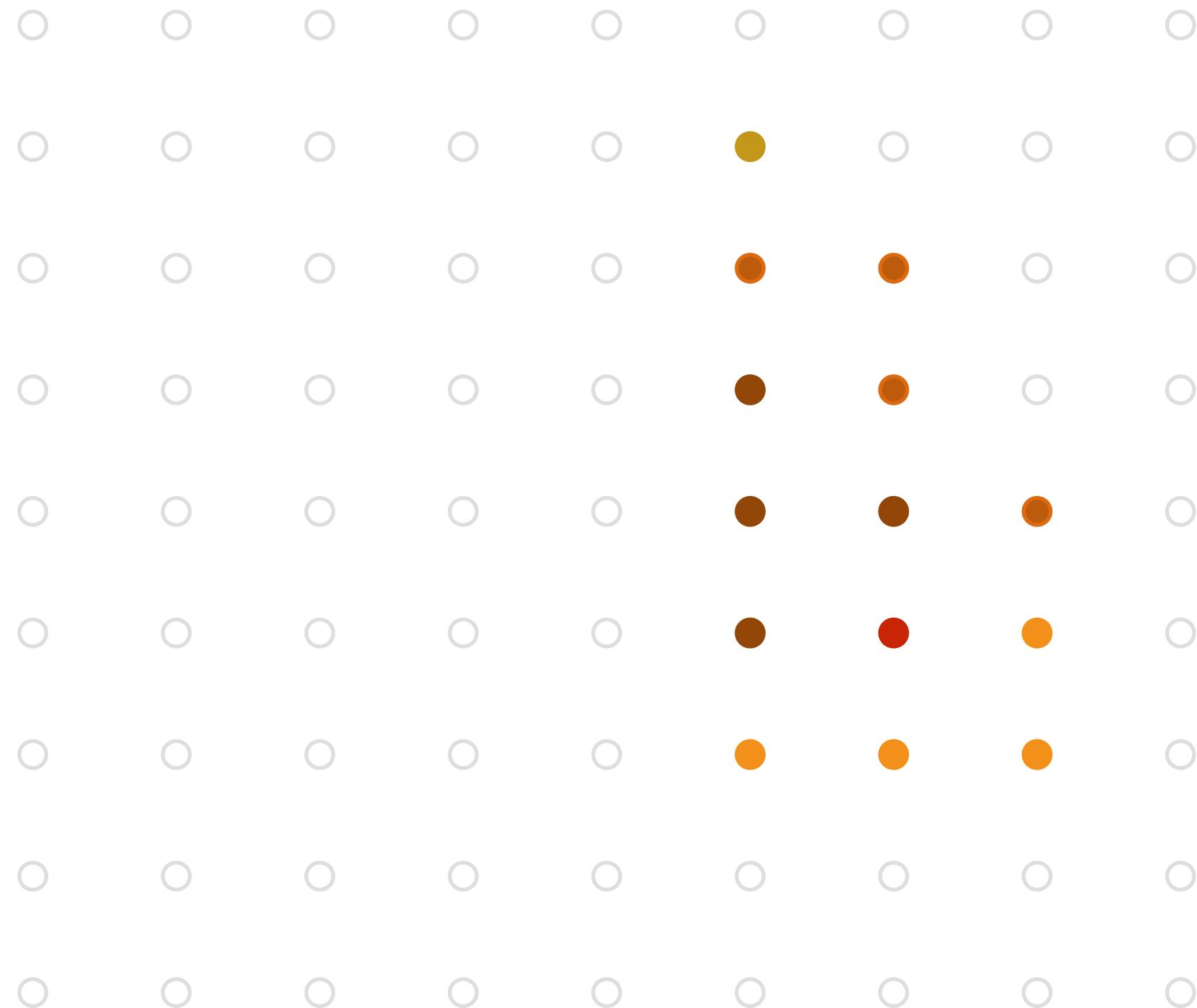
\* So far, we've only described computing triangle's color at a point by interpolating per-vertex colors, or by sampling a texture map. Later in the course, we'll discuss more advanced algorithms for computing its color based on material properties and scene lighting conditions.

# Step 7: perform depth test (if enabled)

Also update depth value at covered samples (if necessary)

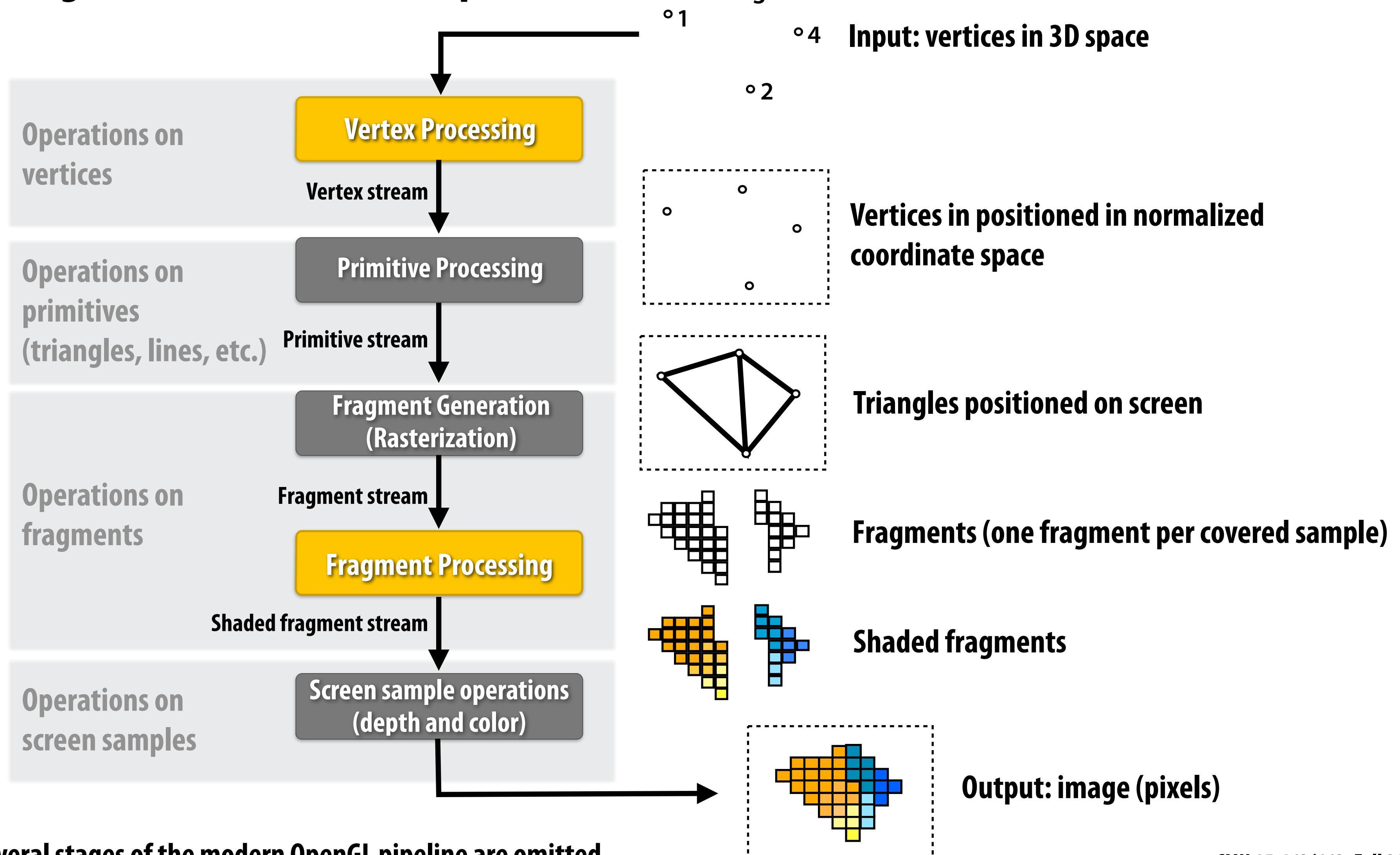


# Step 8: update color buffer (if depth test passed)



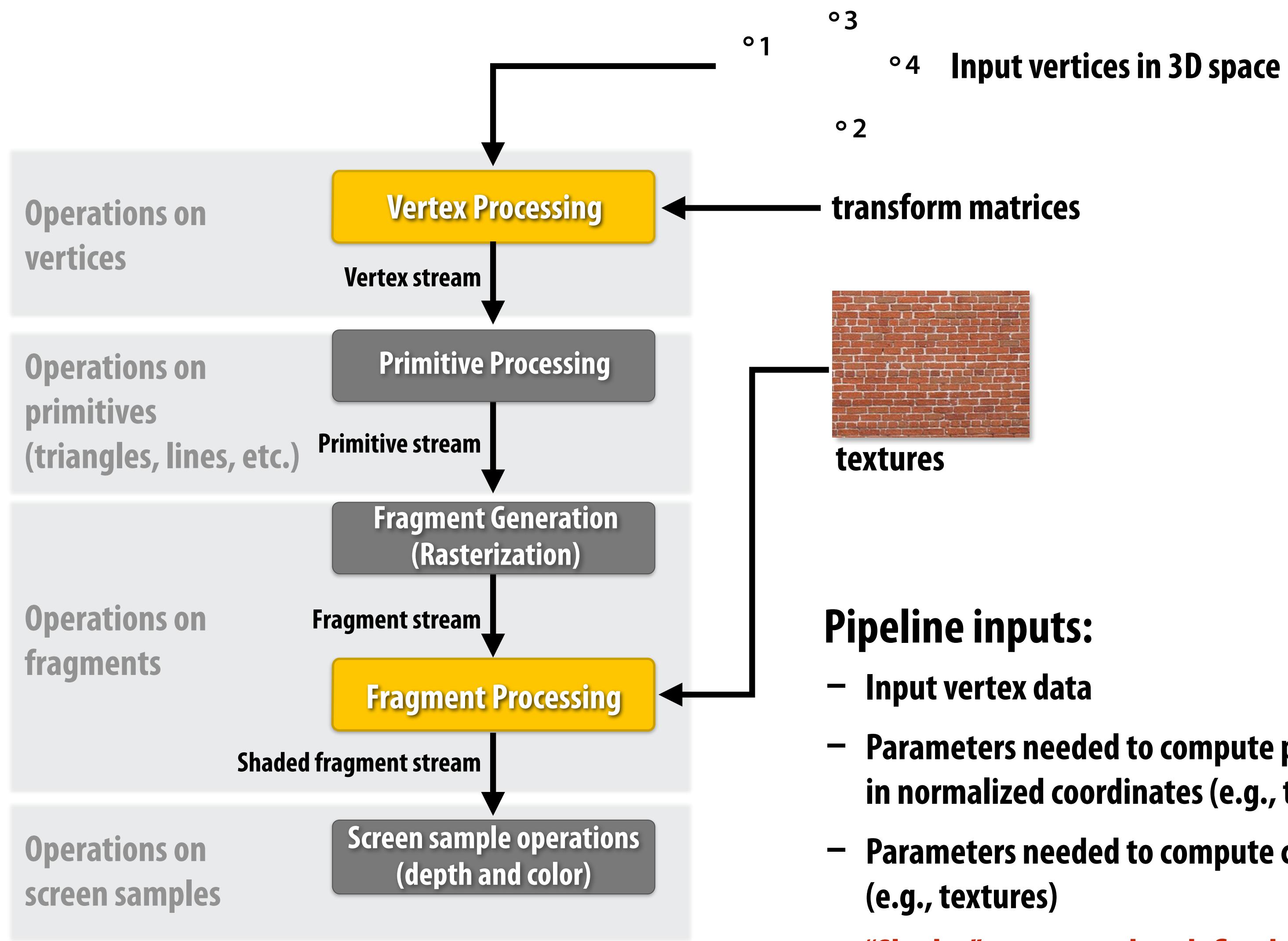
# OpenGL/Direct3D graphics pipeline \*

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

# 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;           Program parameters
uniform vec3 lightDir;
varying vec2 uv;                      Per-fragment attributes
varying vec3 norm;                    (interpolated by rasterizer)

void diffuseShader()
{
    vec3 kd;
    kd = texture2d(myTexture, uv);      Sample surface albedo
                                         (reflectance color) from texture
    kd *= clamp(dot(-lightDir, norm), 0.0, 1.0);
    gl_FragColor = vec4(kd, 1.0);
}
```

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)

# 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



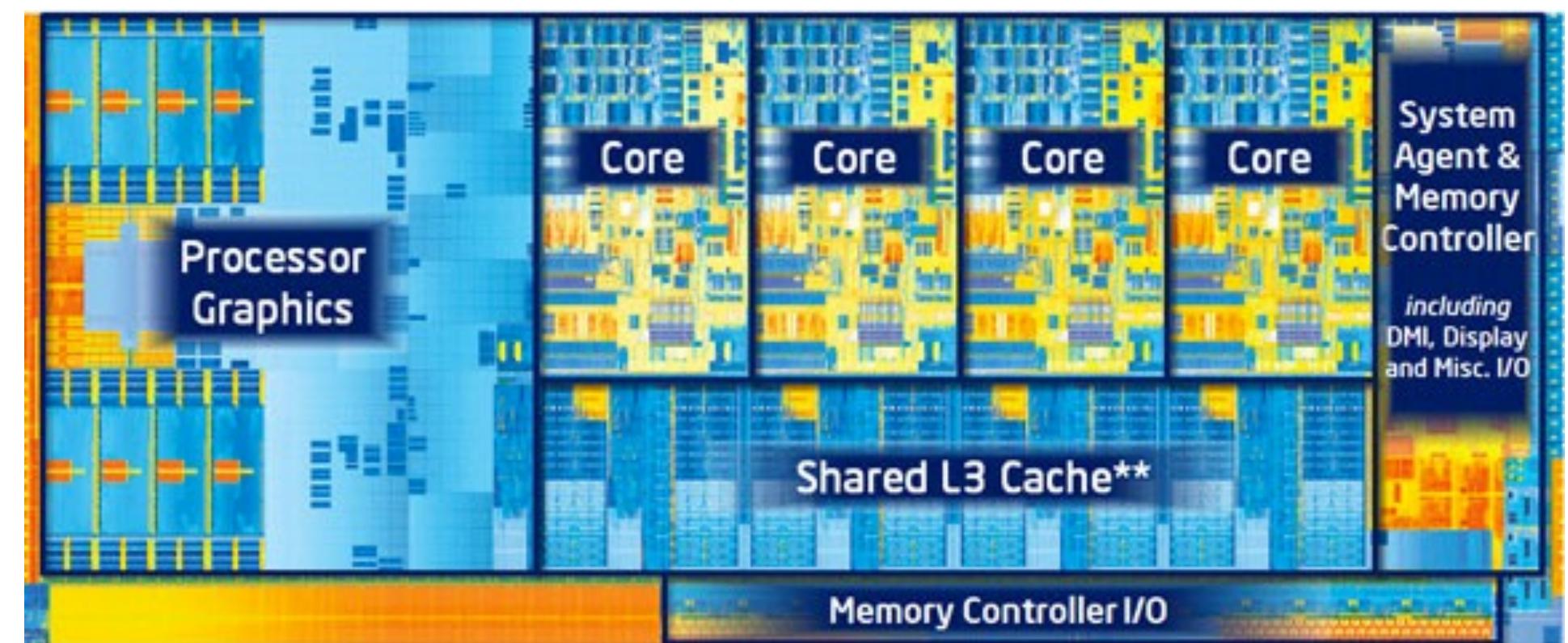
# Graphics pipeline implementation: GPUs

Specialized processors for executing graphics pipeline computations



bit-tech.net

Discrete GPU card  
(NVIDIA GeForce Titan X)

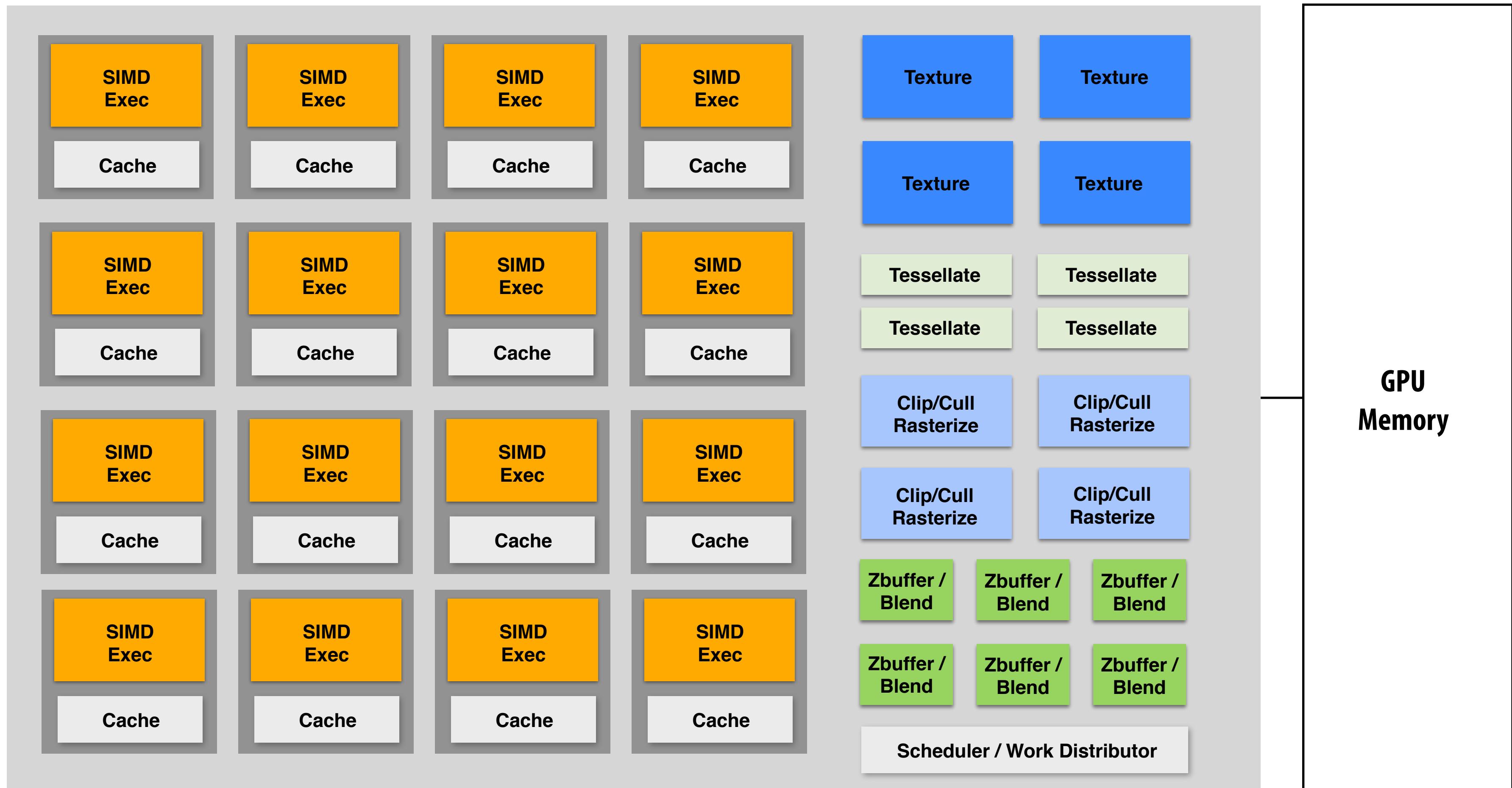


Integrated GPU: part of modern Intel CPU die

# GPU: heterogeneous, multi-core processor

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

T-0P's of fixed-function  
compute capability over here



Take Kayvon's parallel computing course (15-418) 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)