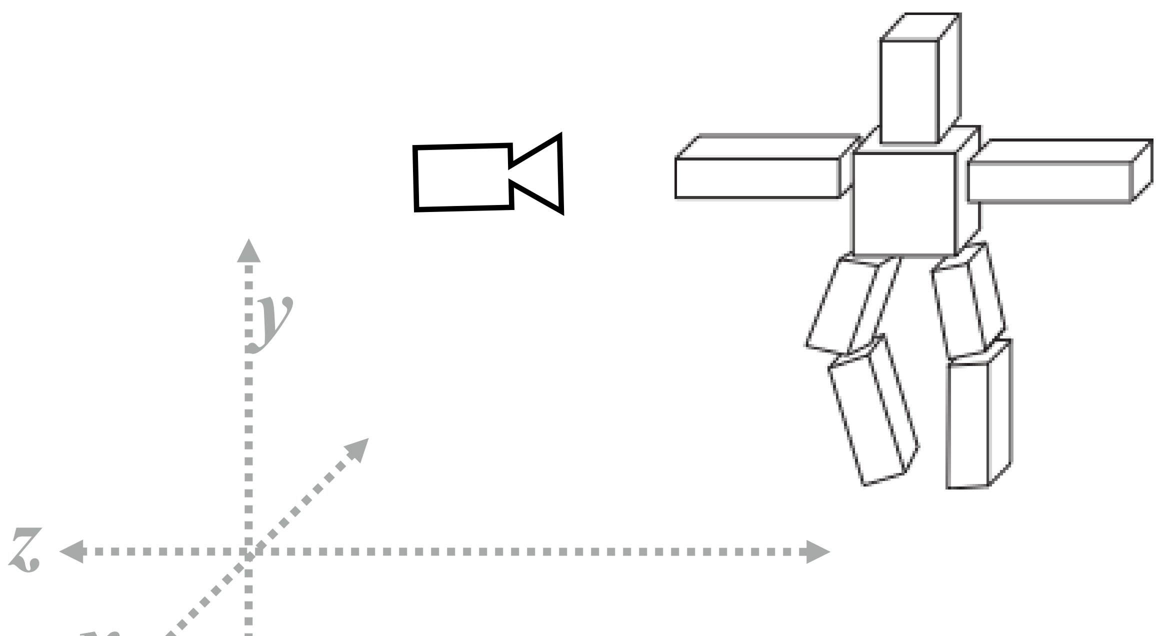
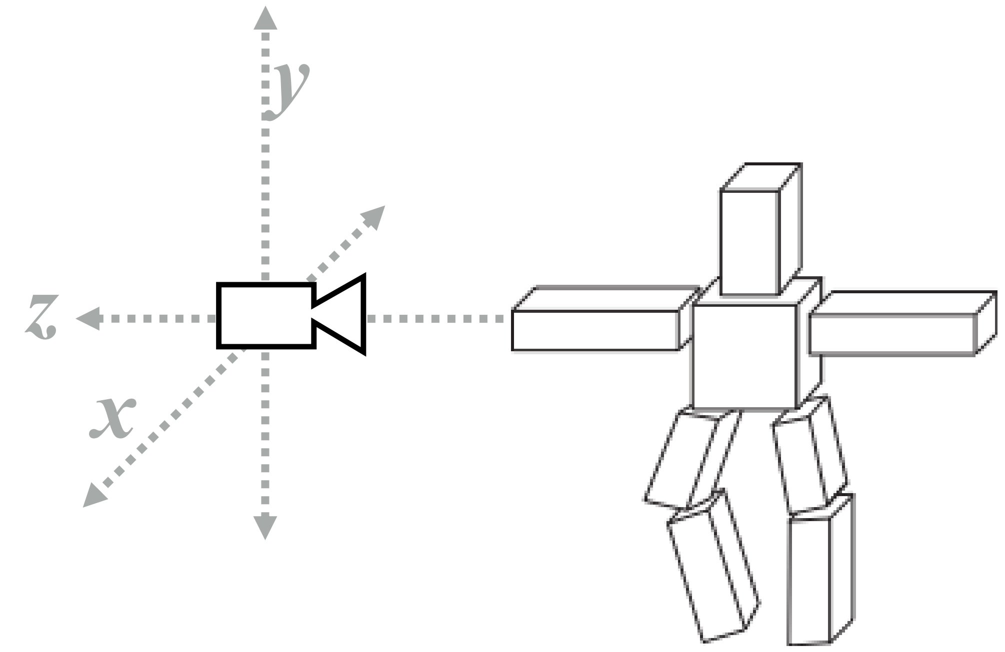
The Rasterization Pipeline

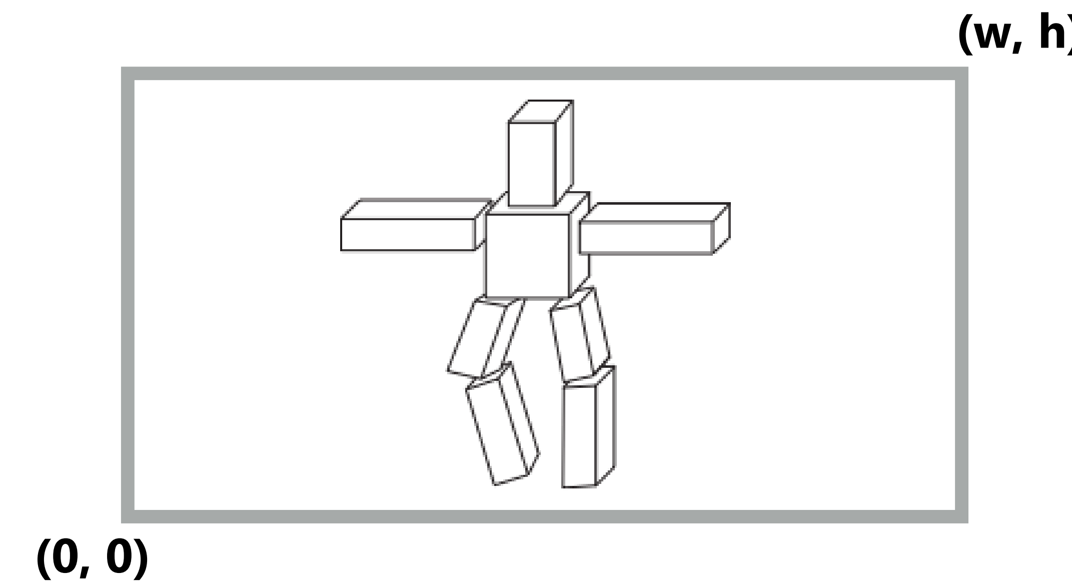
What you know how to do at this point in the course



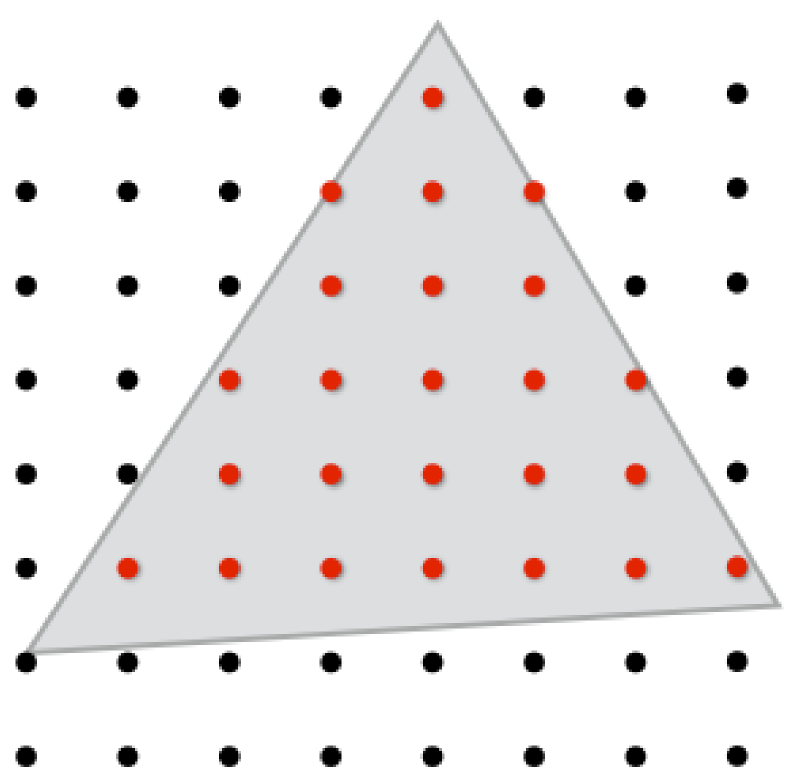
Position objects and the camera in the world



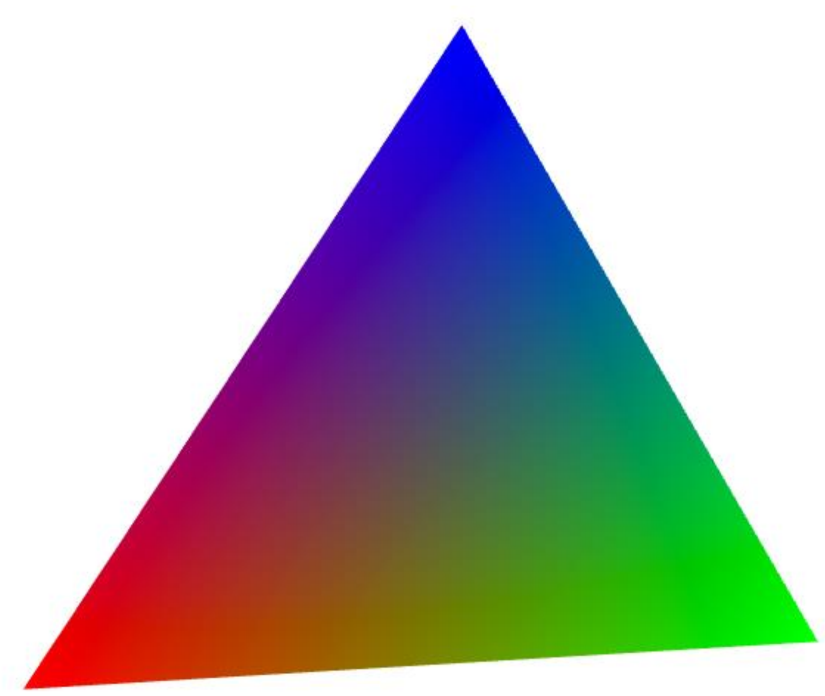
Determine the position of objects relative to the camera



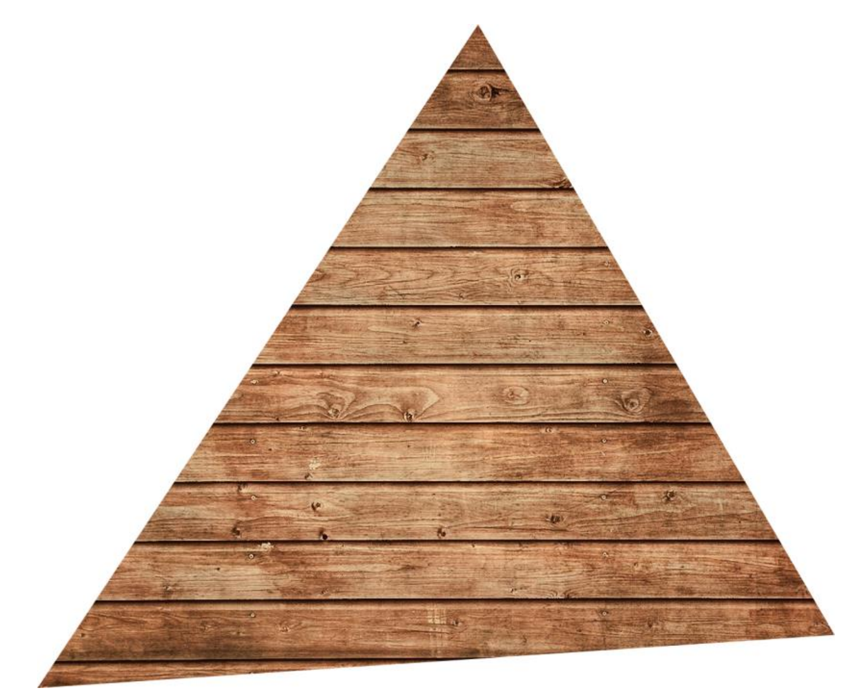
Project objects onto the screen



Sample triangle coverage



Compute triangle attribute values at covered sample points



Sample texture maps

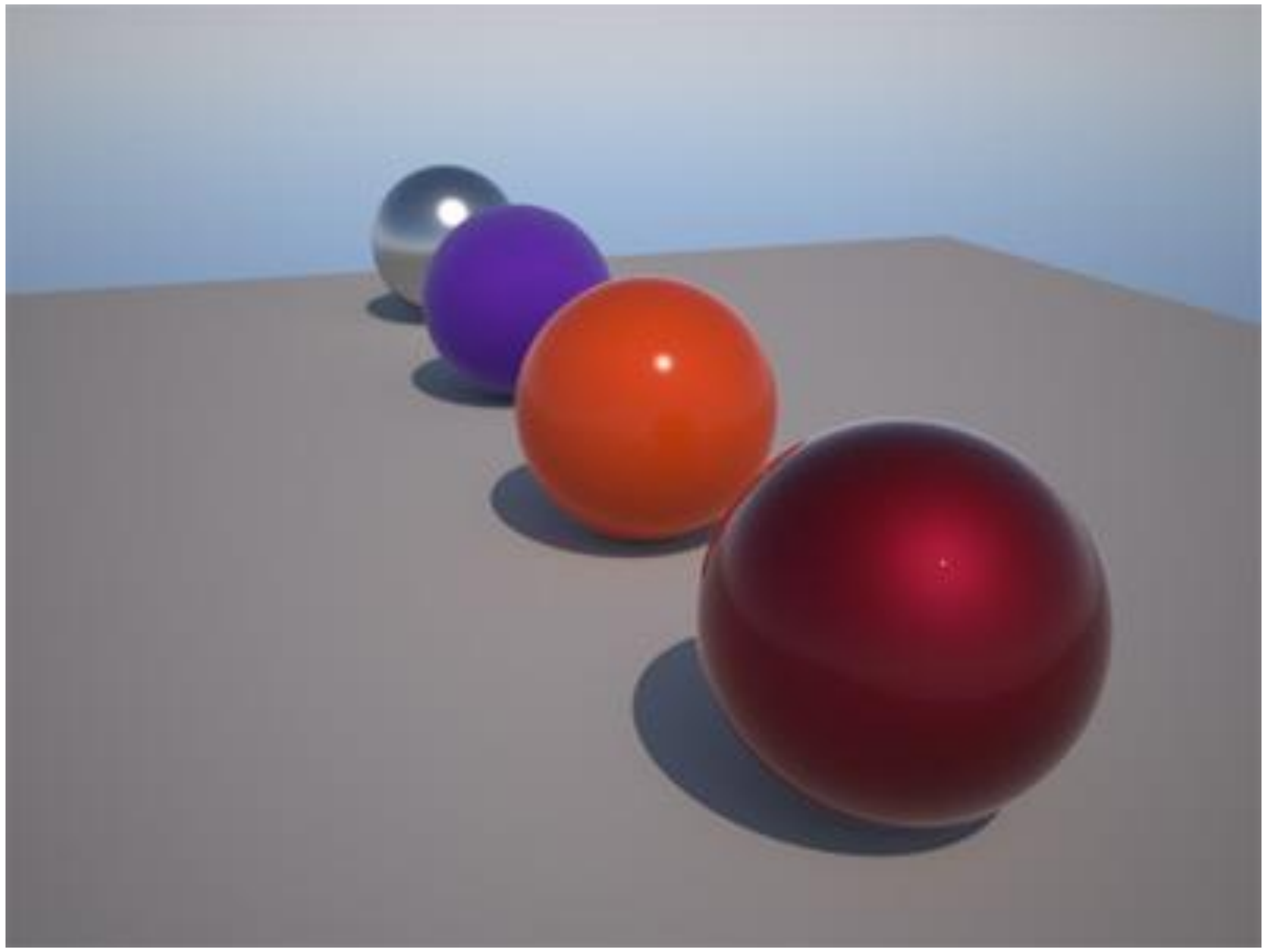
Course roadmap

Drawing Things

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

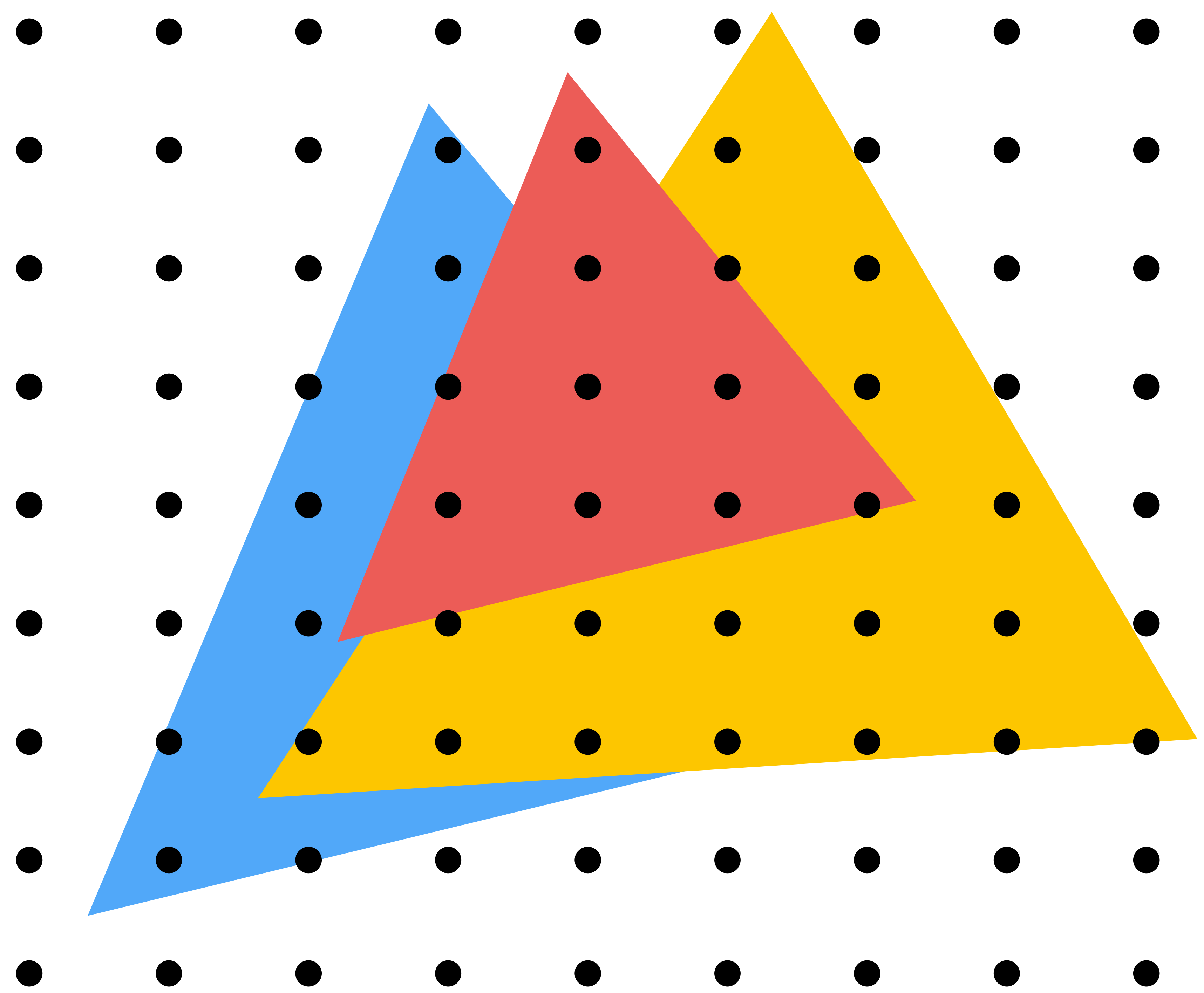
Materials and Lighting

Animation



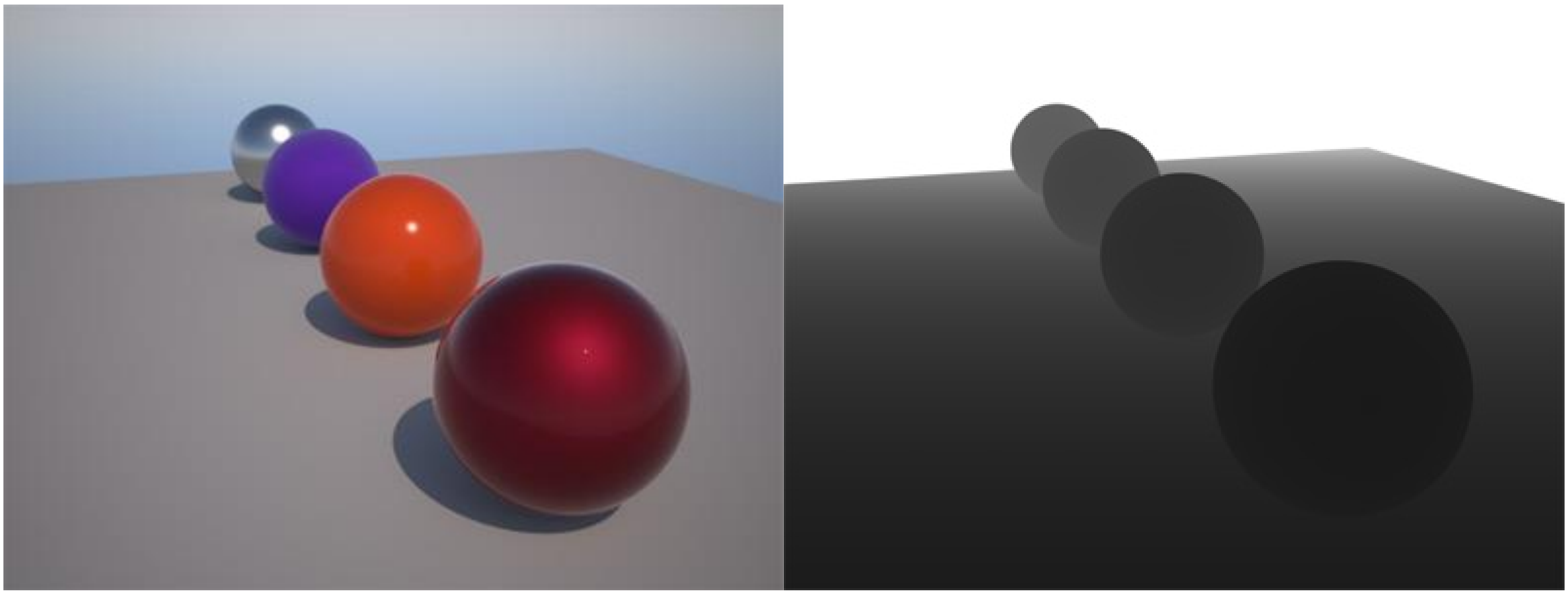
Occlusion

Which triangle is visible at each pixel?



Opaque Triangles

The depth buffer (Z-buffer)



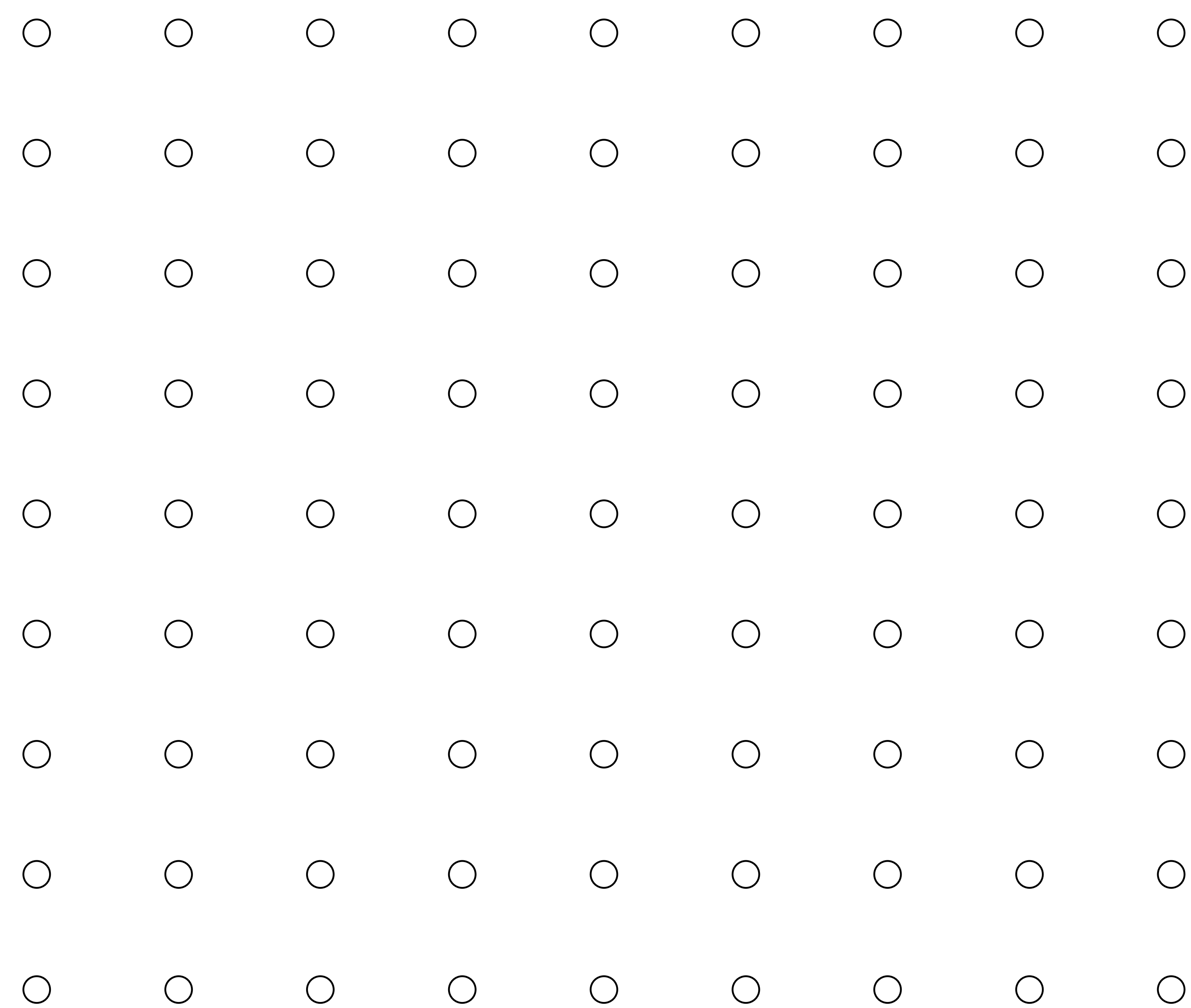
Q: How do we compute the depth of sampled points on a triangle?

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.

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



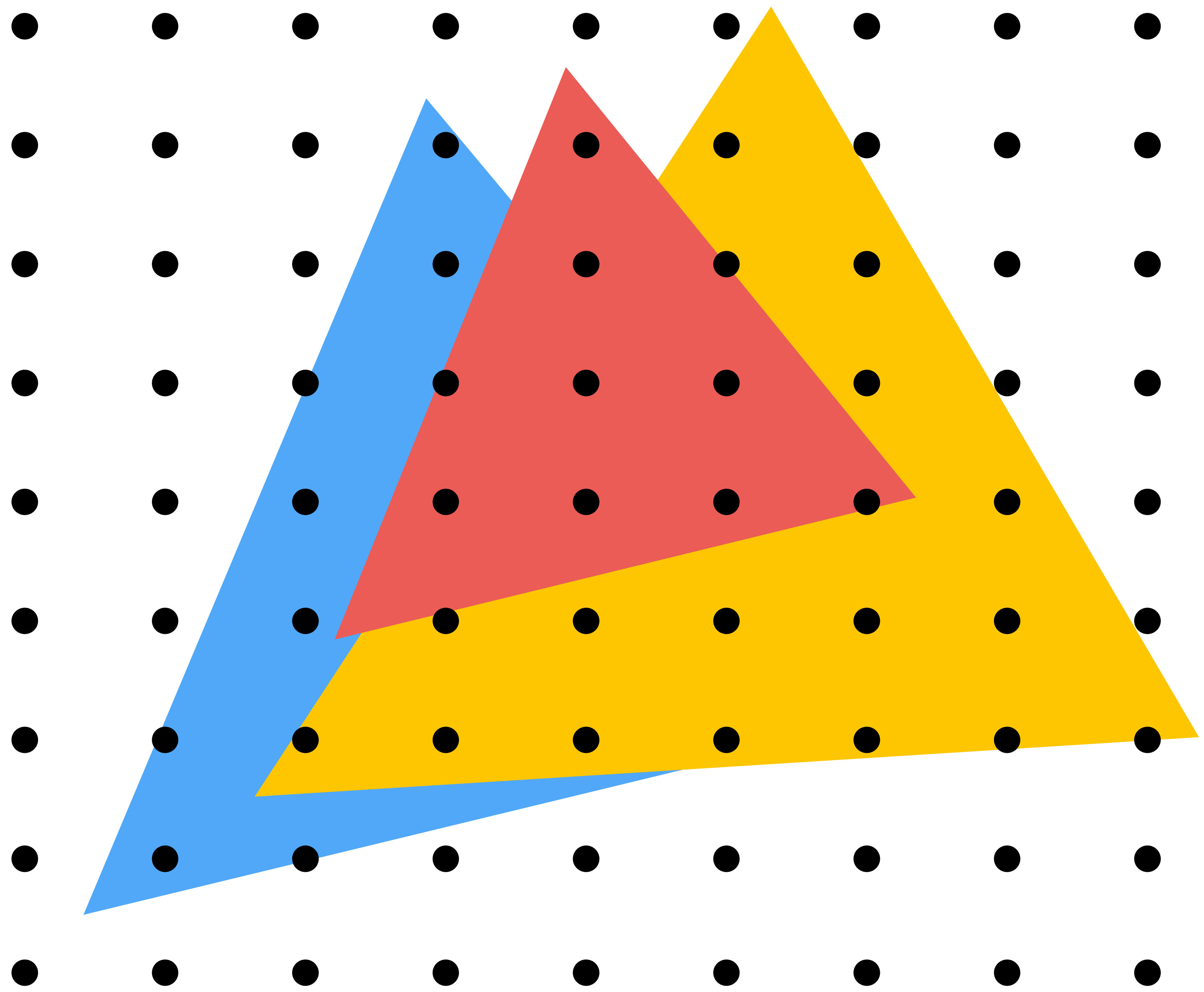
Grayscale value of sample point used to indicate distance

Black = small distance

White = large distance

Depth buffer example

Example: rendering three opaque triangles



Occlusion using the depth-buffer (Z-buffer)

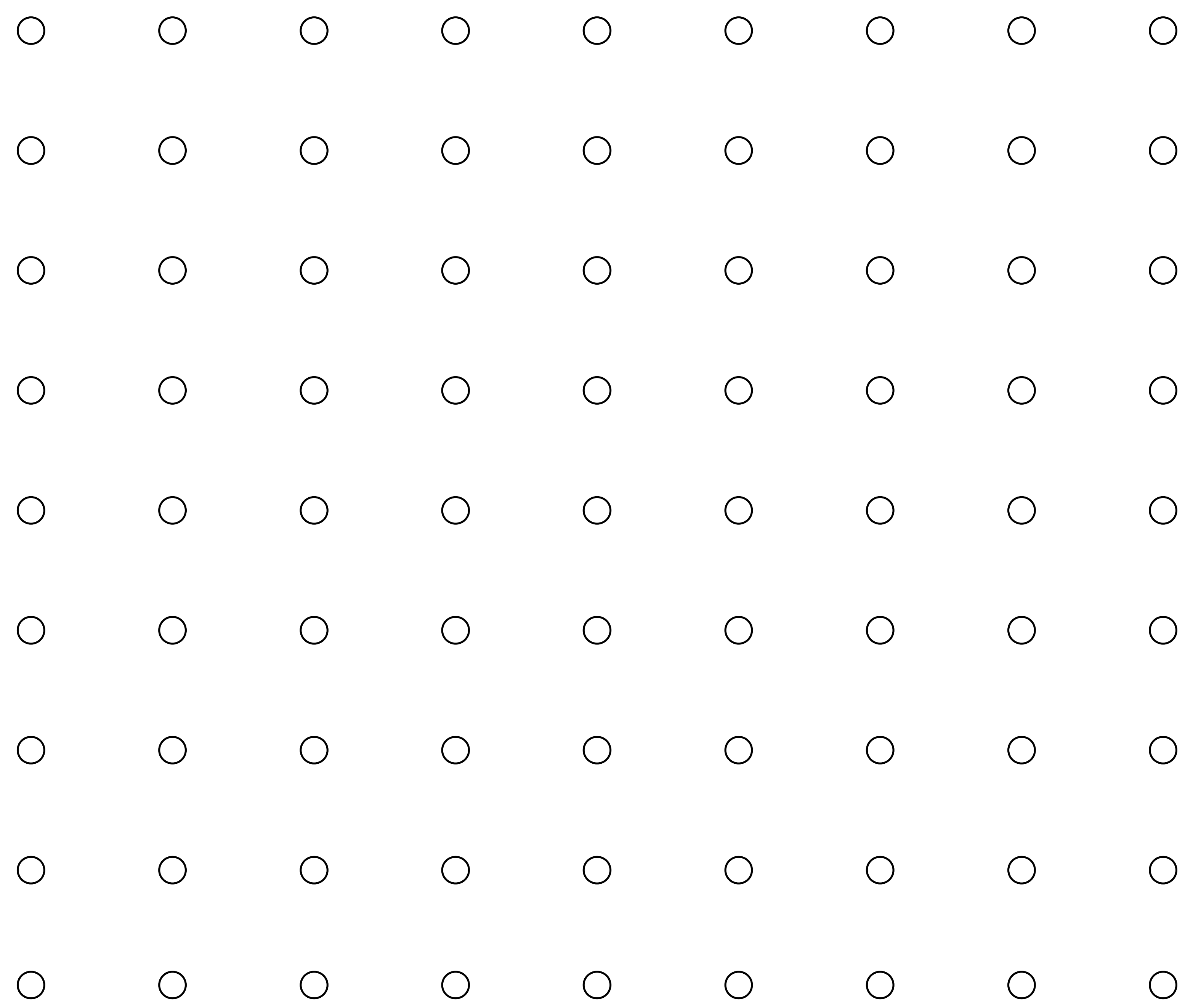
Processing yellow triangle:
depth = 0.5

Grayscale value of sample point used to indicate distance

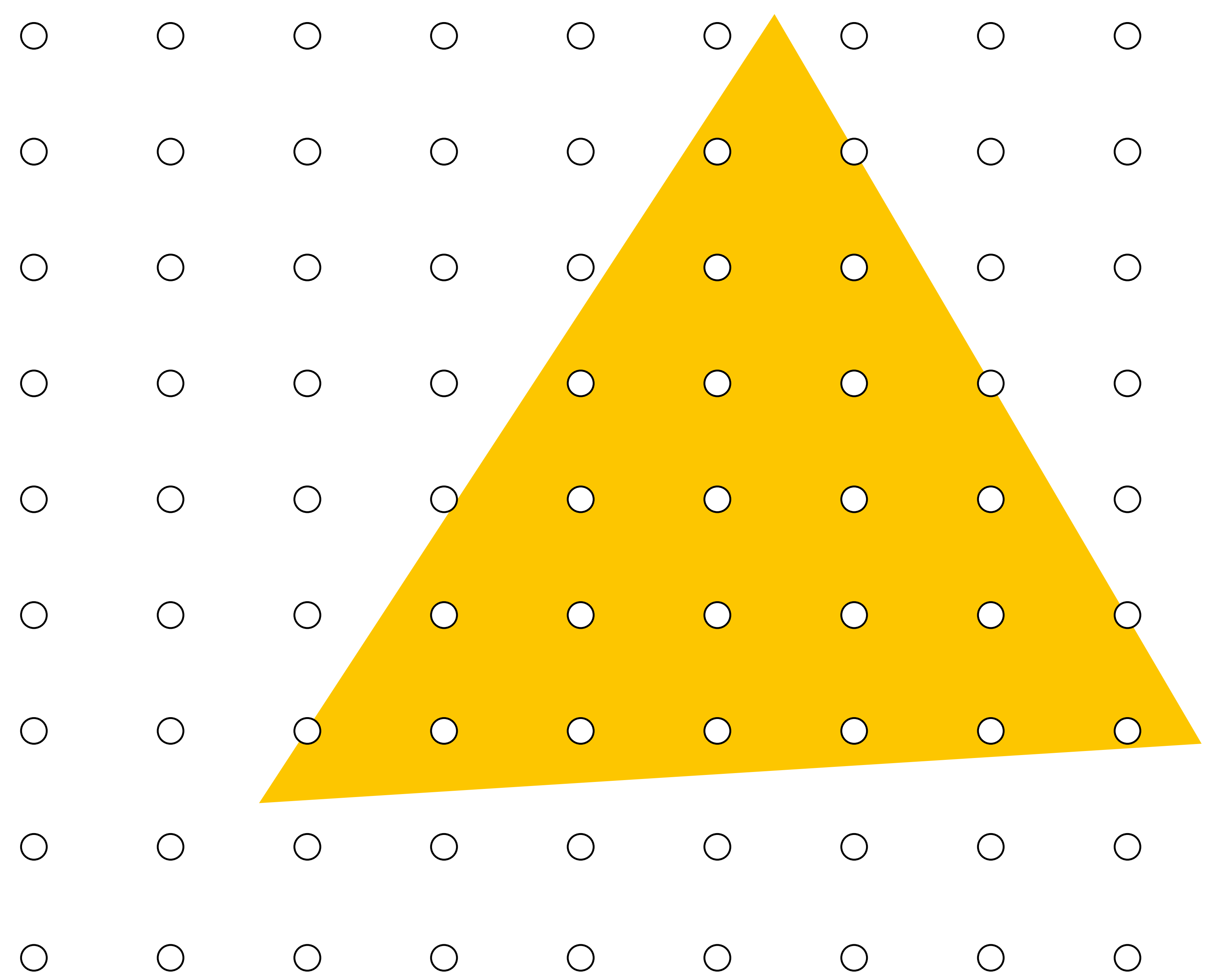
White = large distance

Black = small distance

Red = sample passed depth test



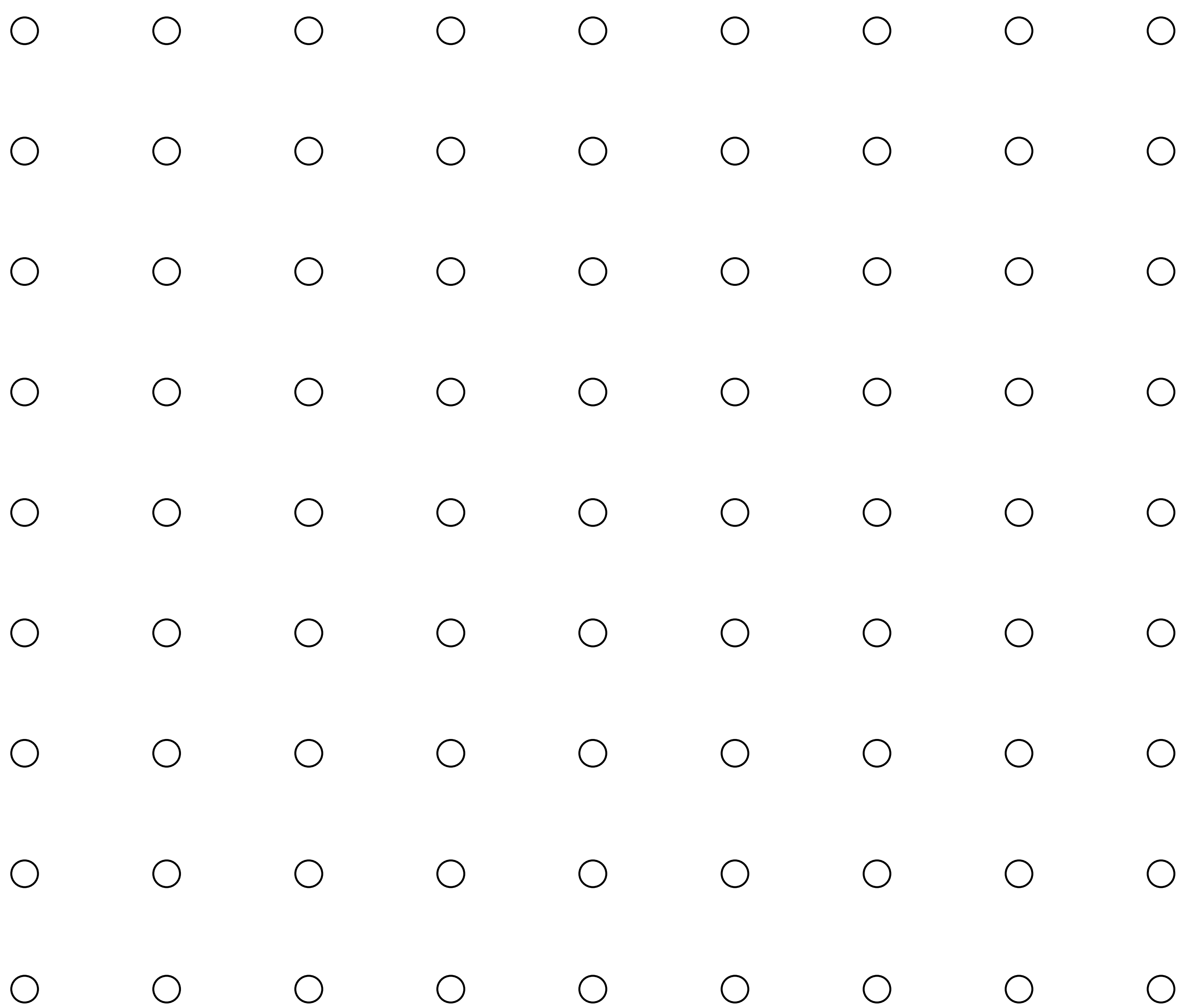
Color buffer contents



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

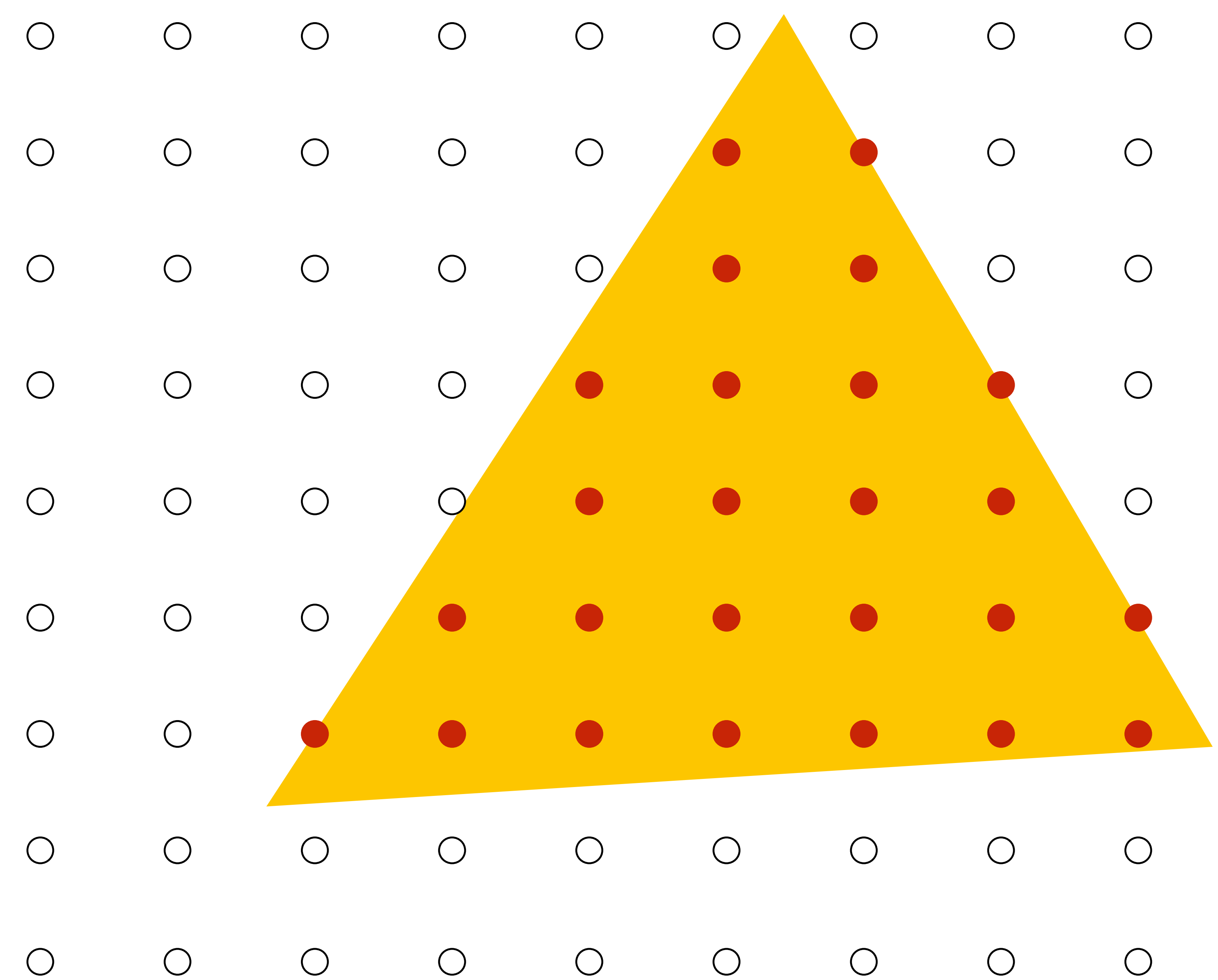
Processing yellow triangle:
depth = 0.5



Color buffer contents

Grayscale value of sample point used to indicate distance

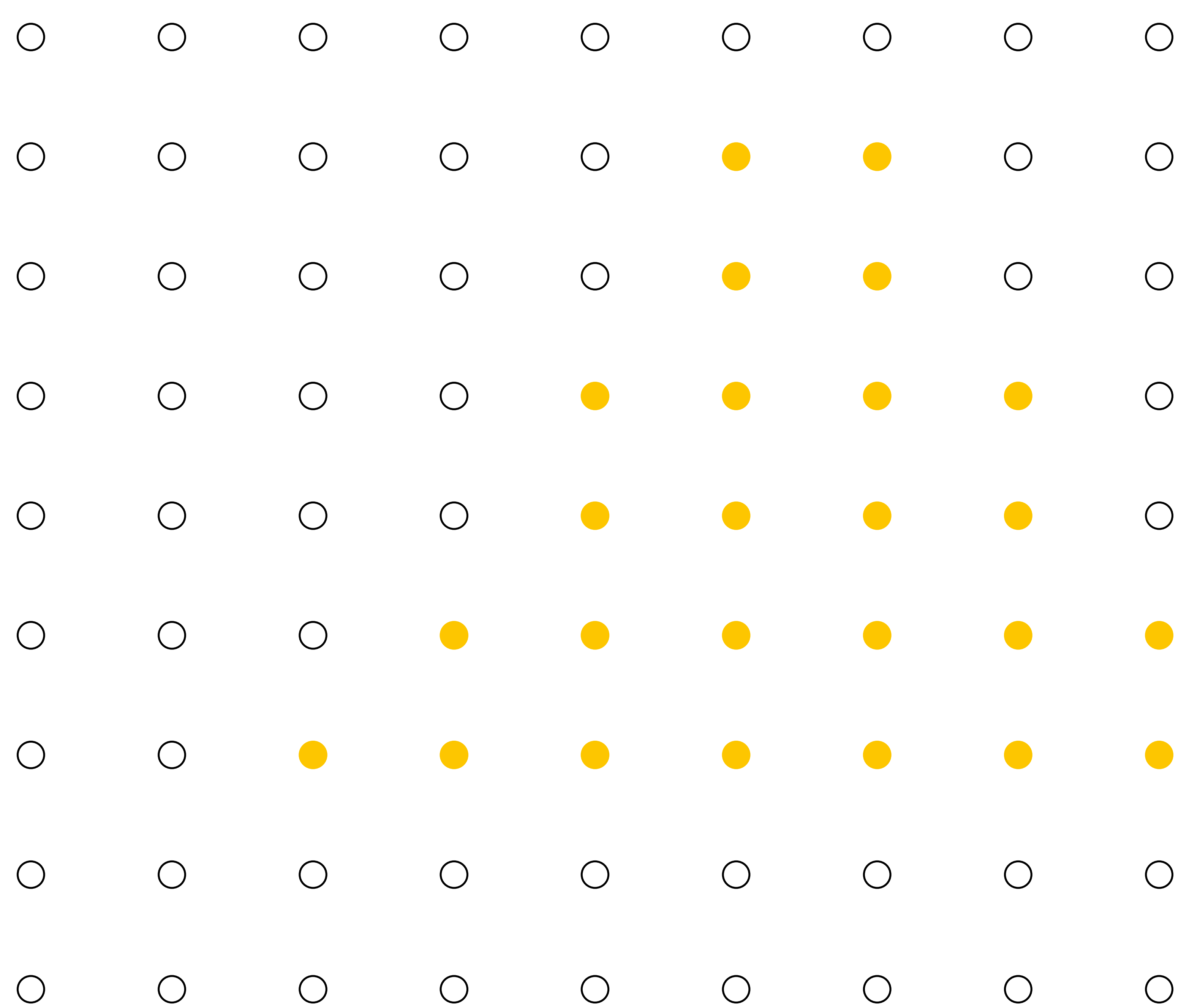
White = large distance
Black = small distance
Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing yellow triangle:



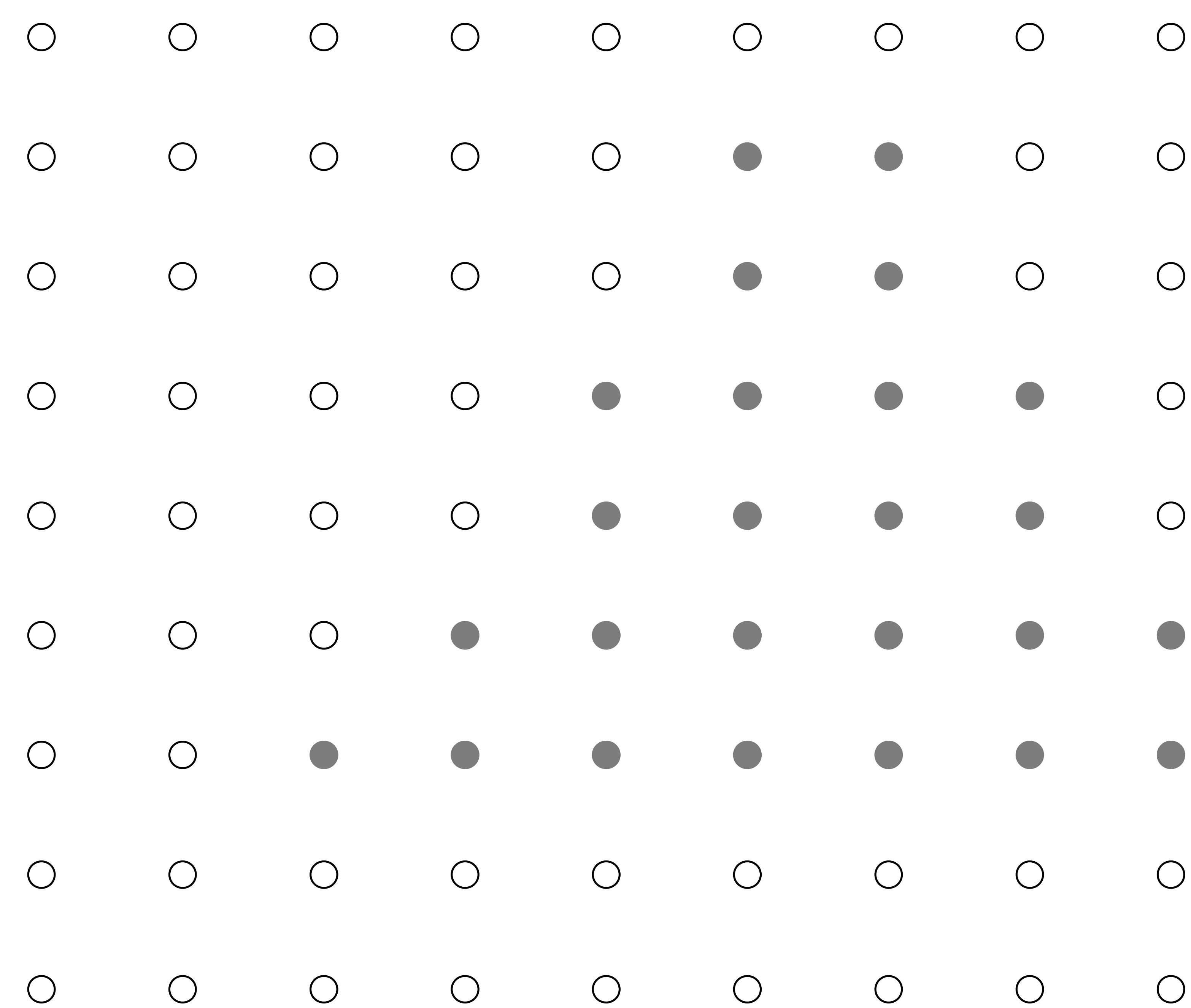
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

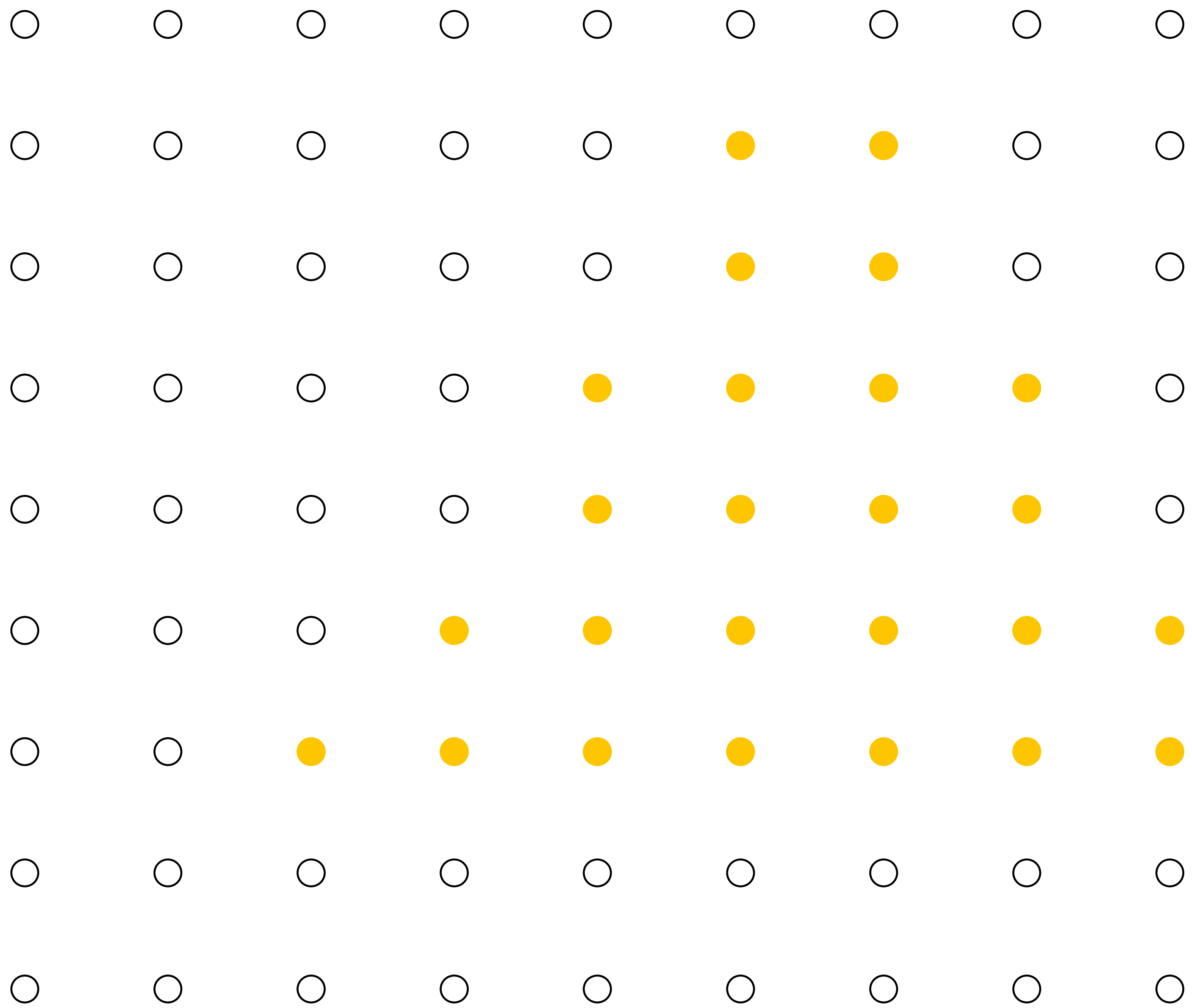
Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

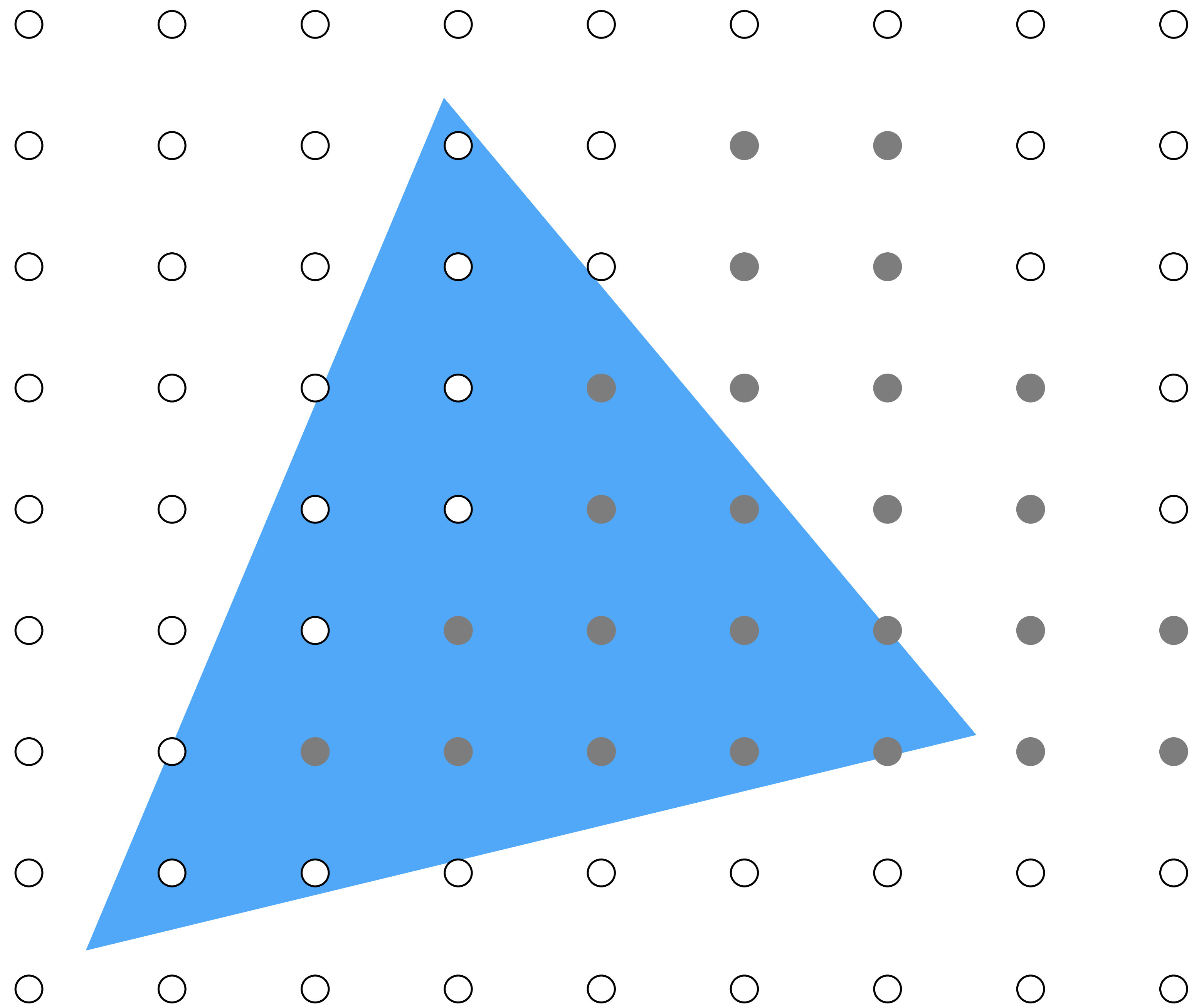
Processing blue triangle:
depth = 0.75



Color buffer contents

Grayscale value of sample point used to indicate distance

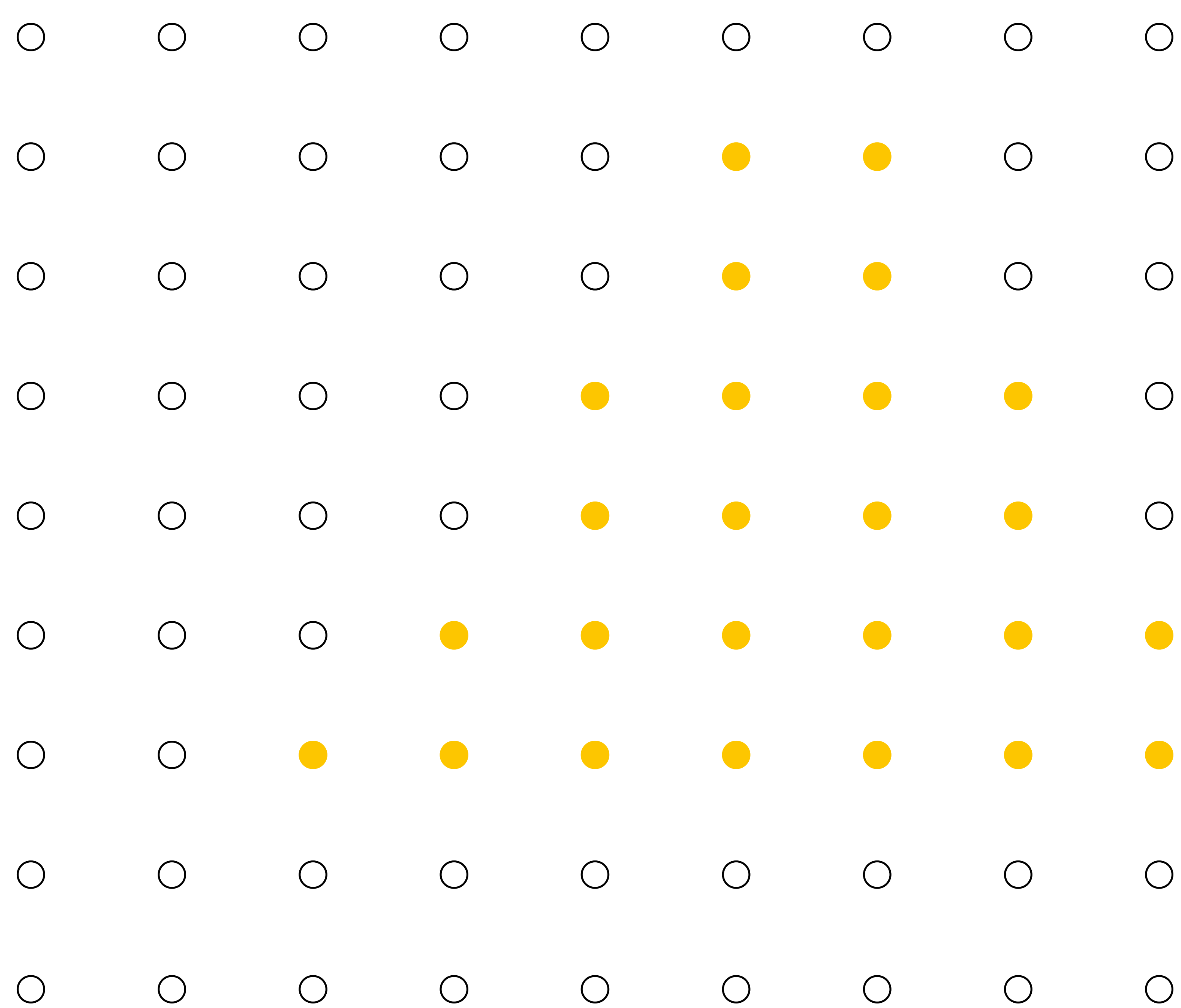
White = large distance
Black = small distance
Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

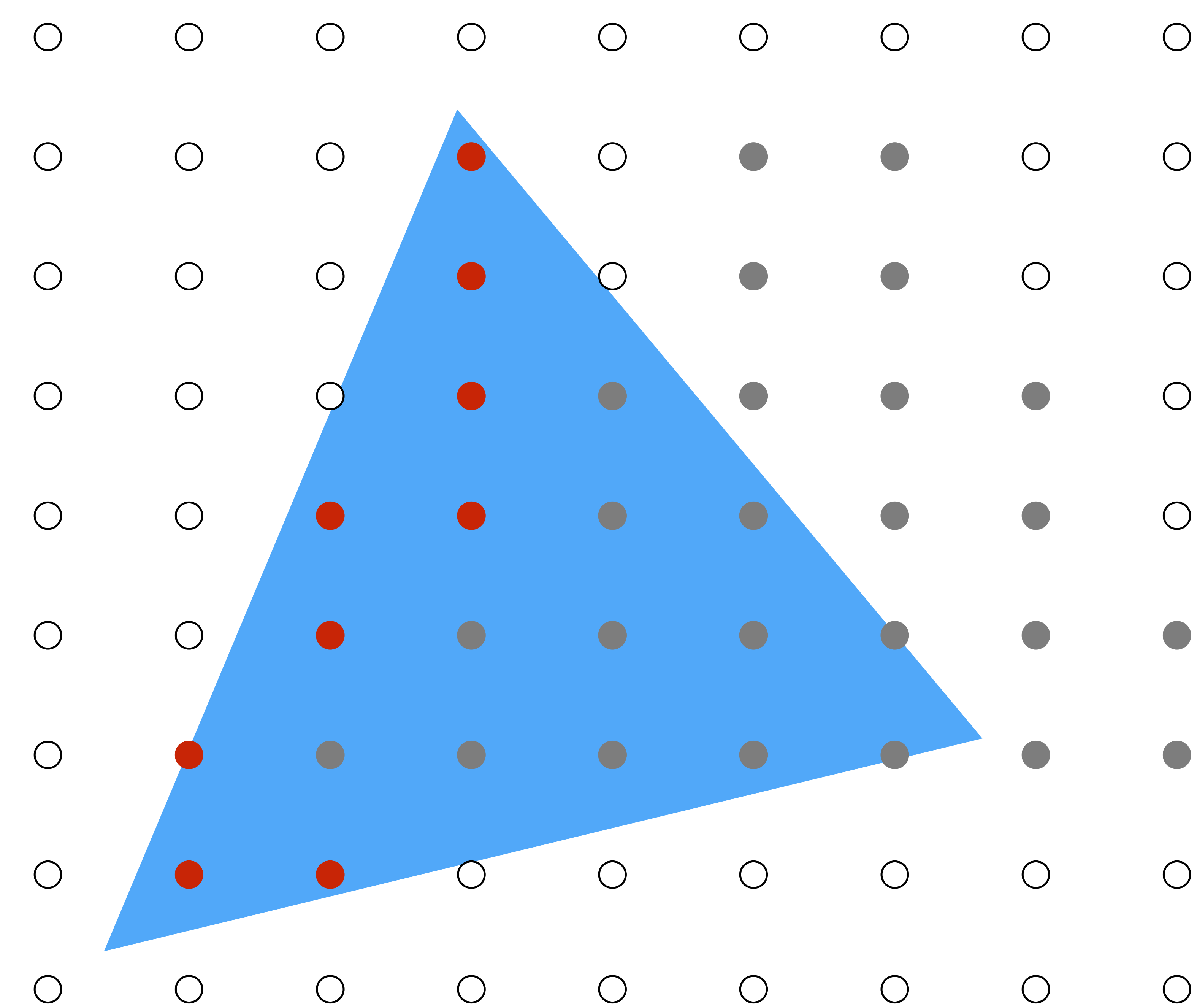
Processing blue triangle:
depth = 0.75



Color buffer contents

Grayscale value of sample point used to indicate distance

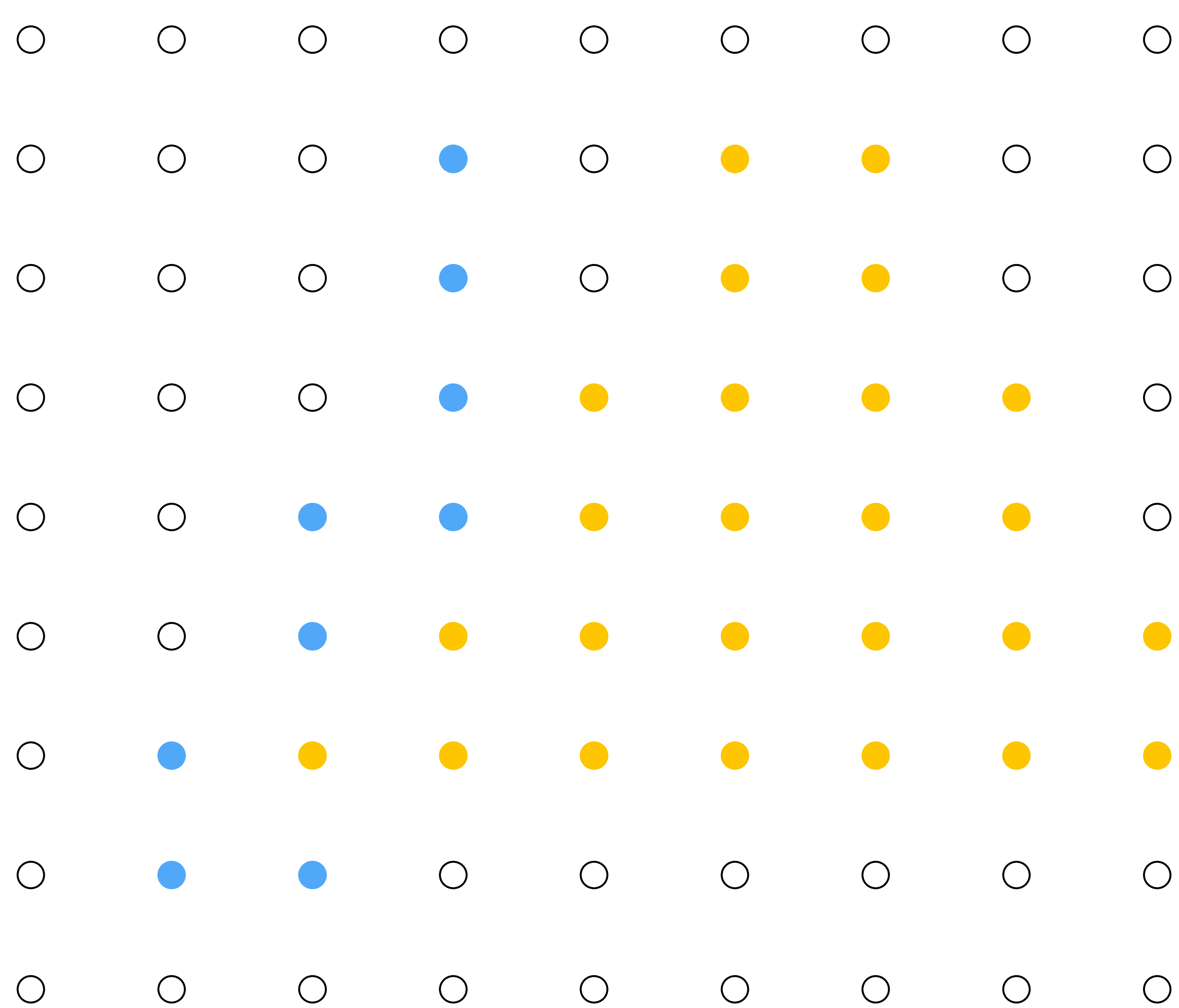
White = large distance
Black = small distance
Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing blue triangle:



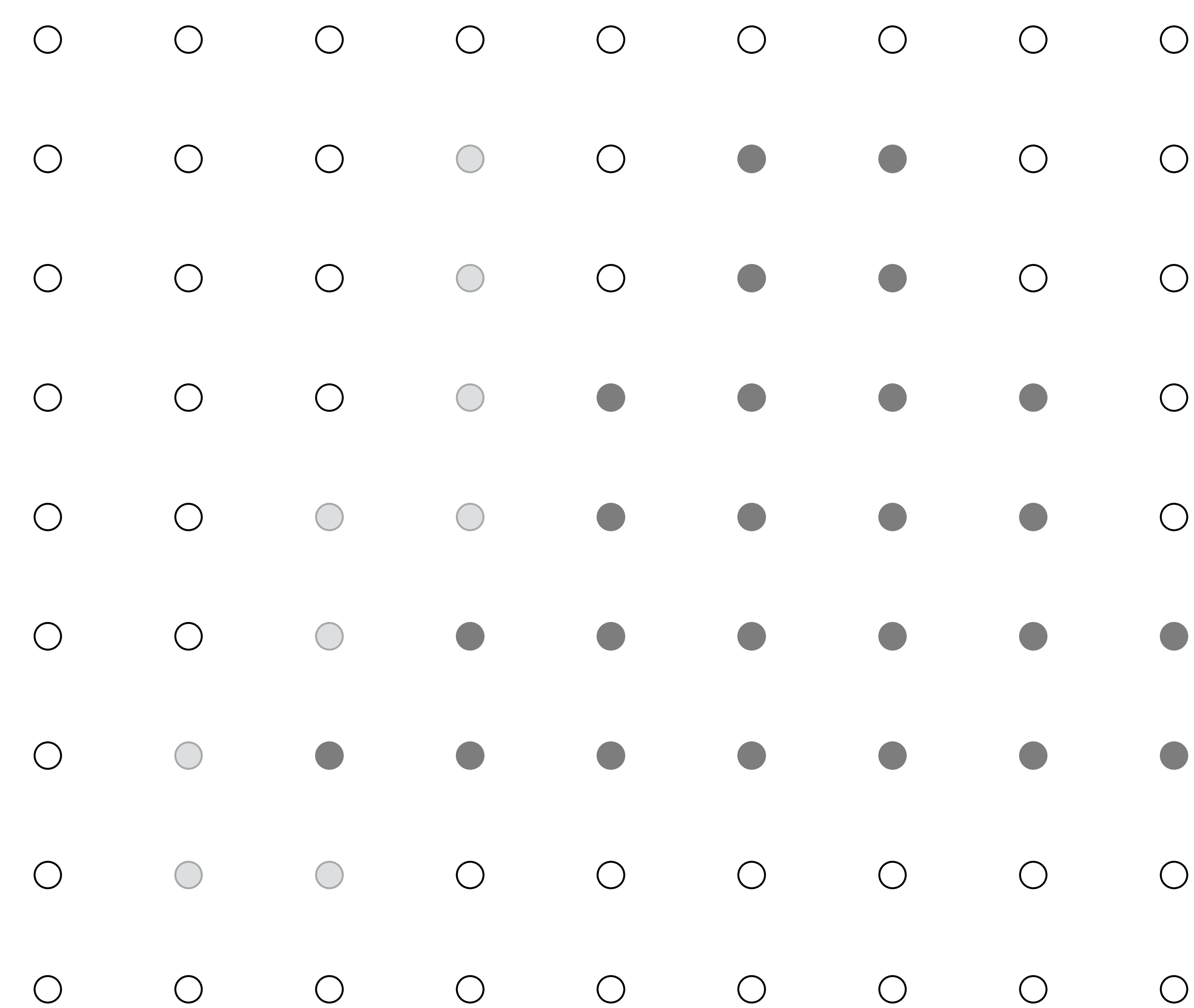
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

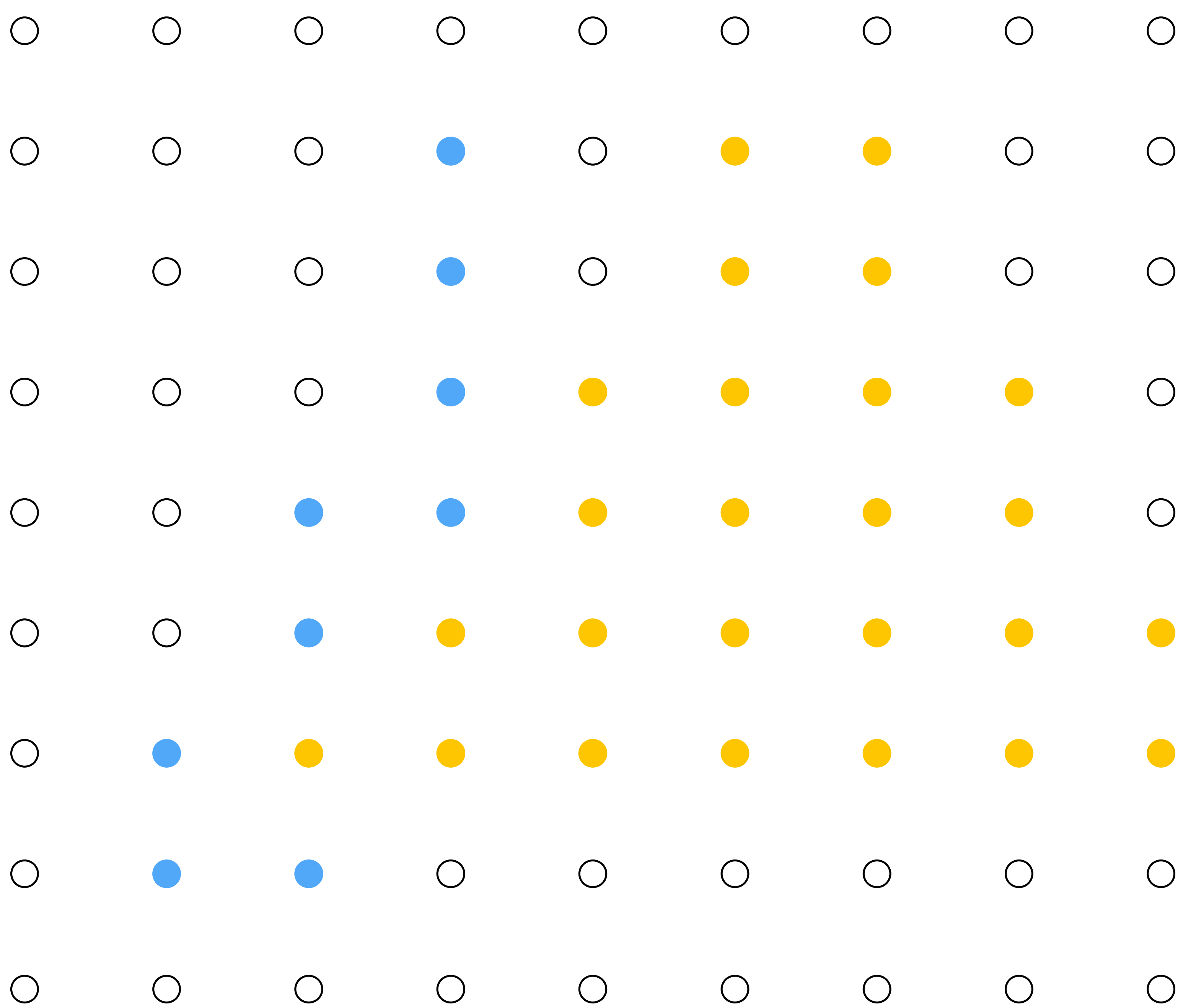
Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

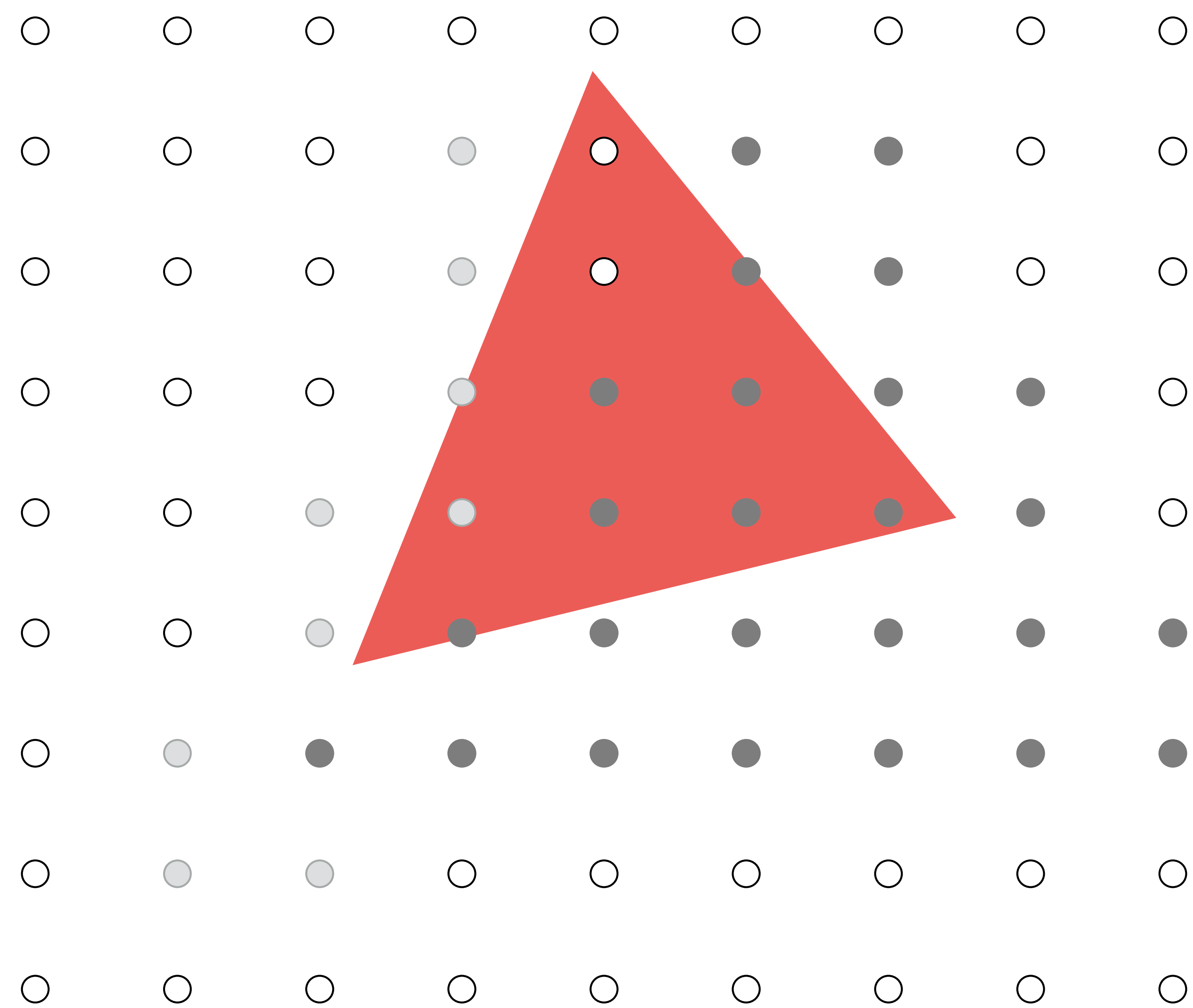
Processing red triangle:
depth = 0.25



Color buffer contents

Grayscale value of sample point used to indicate distance

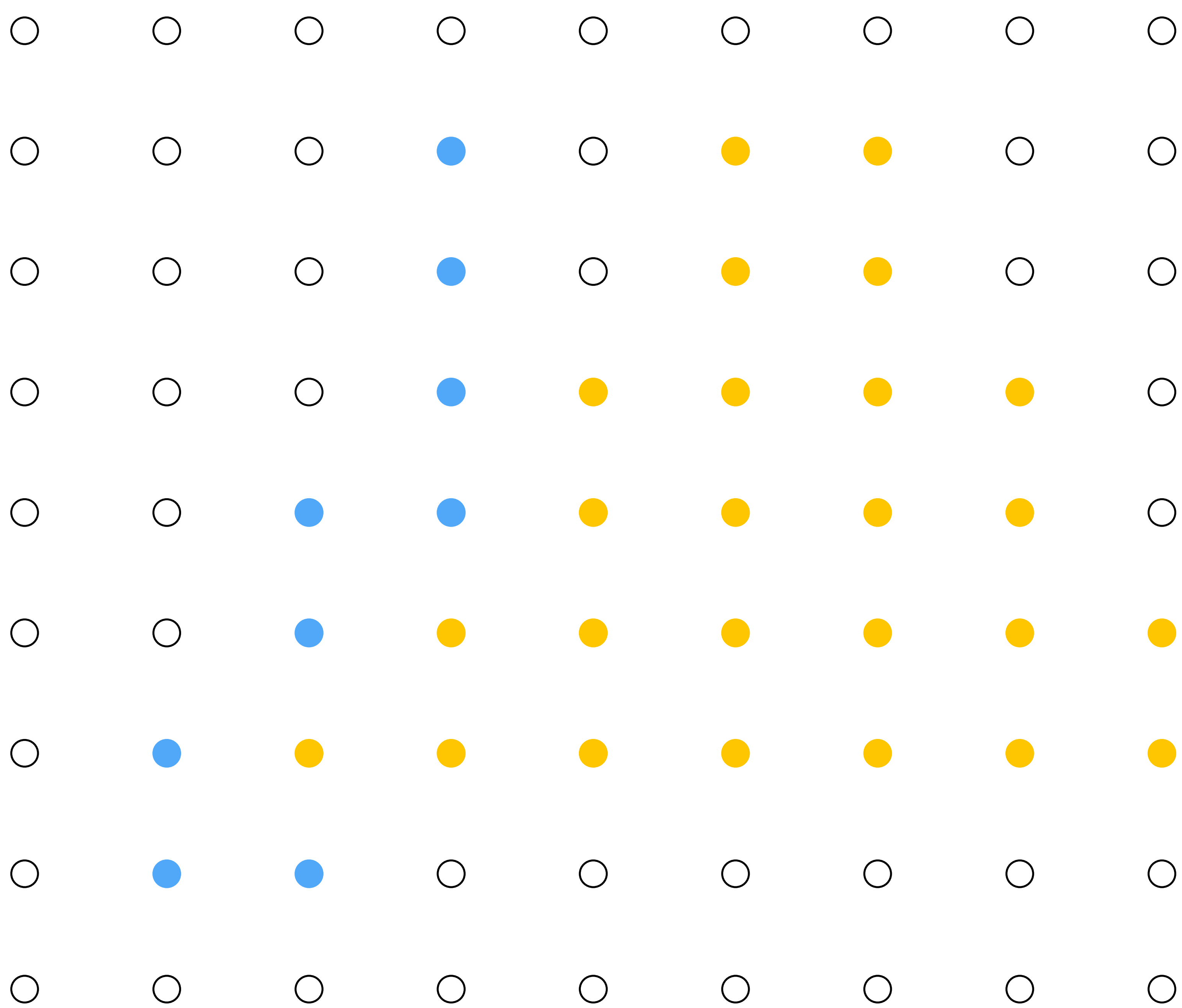
White = large distance
Black = small distance
Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

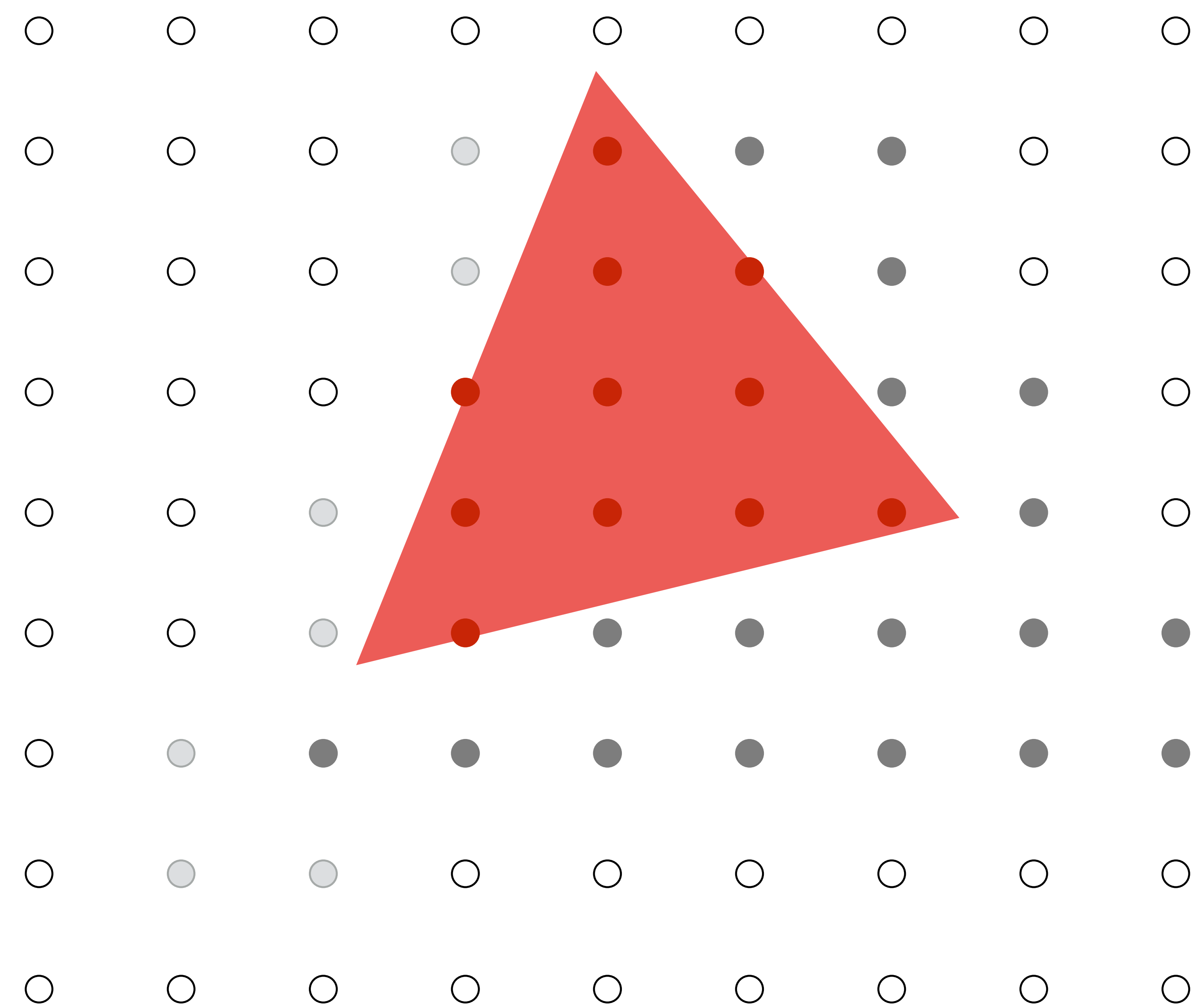
Processing red triangle:
depth = 0.25



Color buffer contents

Grayscale value of sample point used to indicate distance

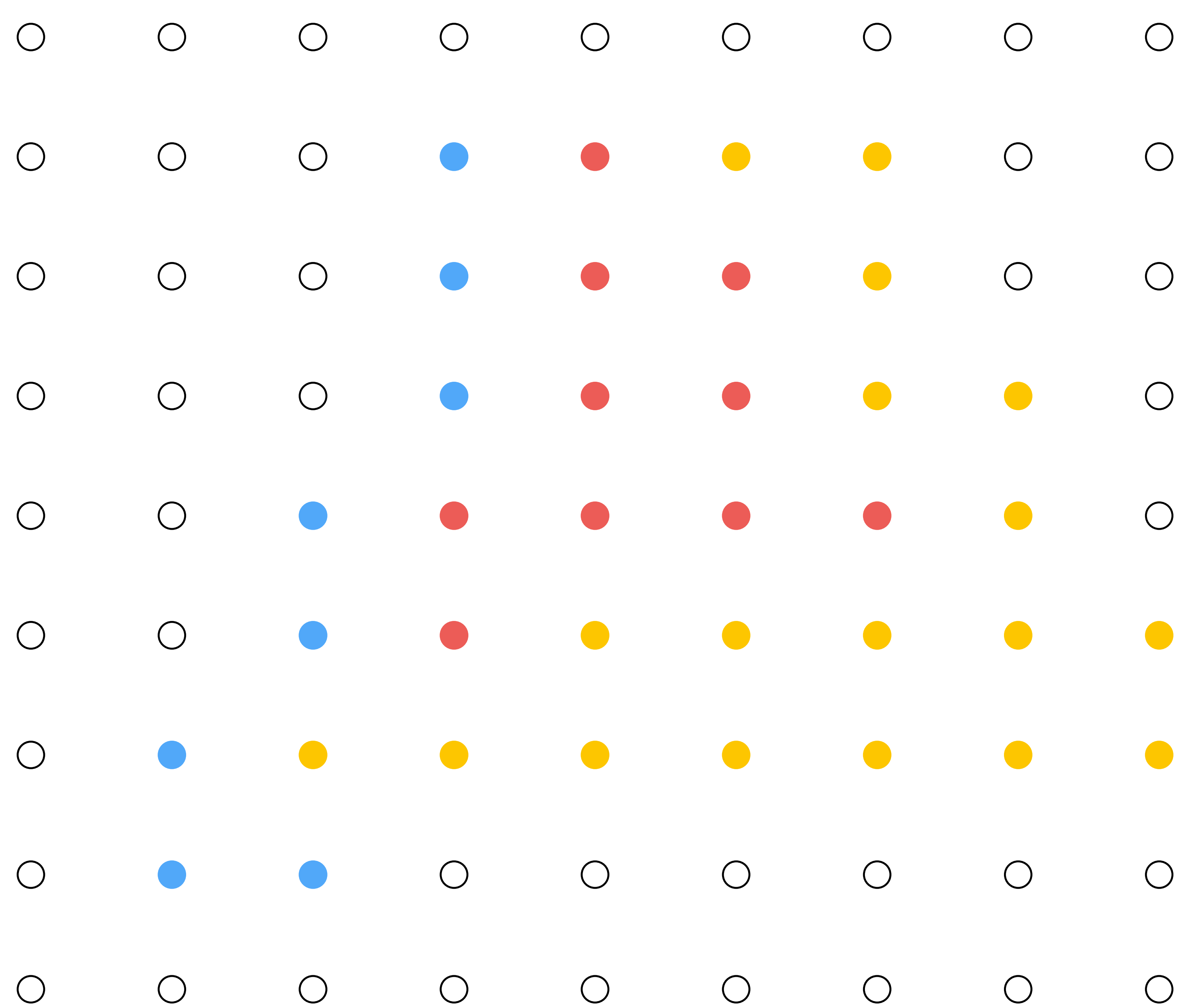
White = large distance
Black = small distance
Red = sample passed depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:



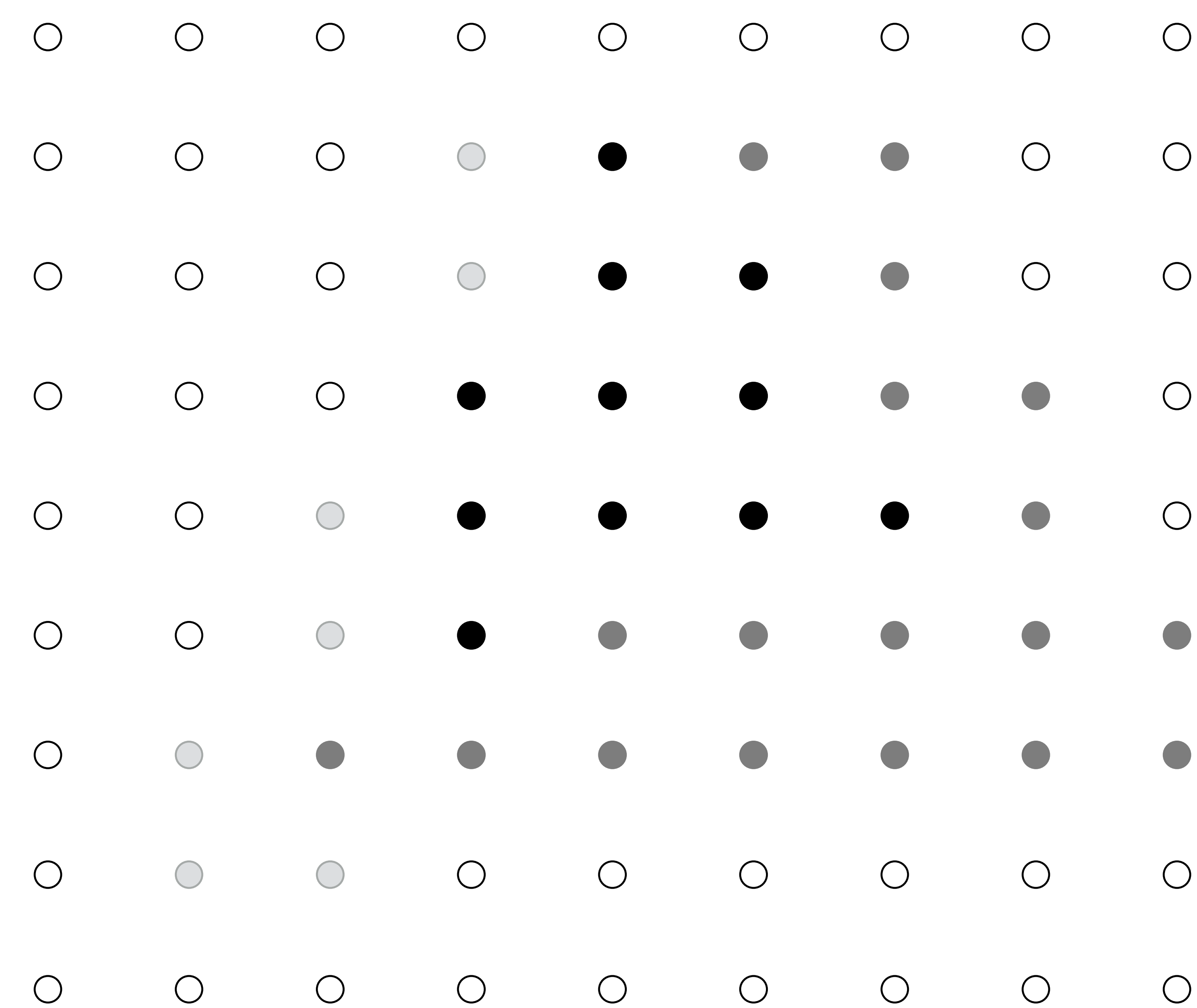
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

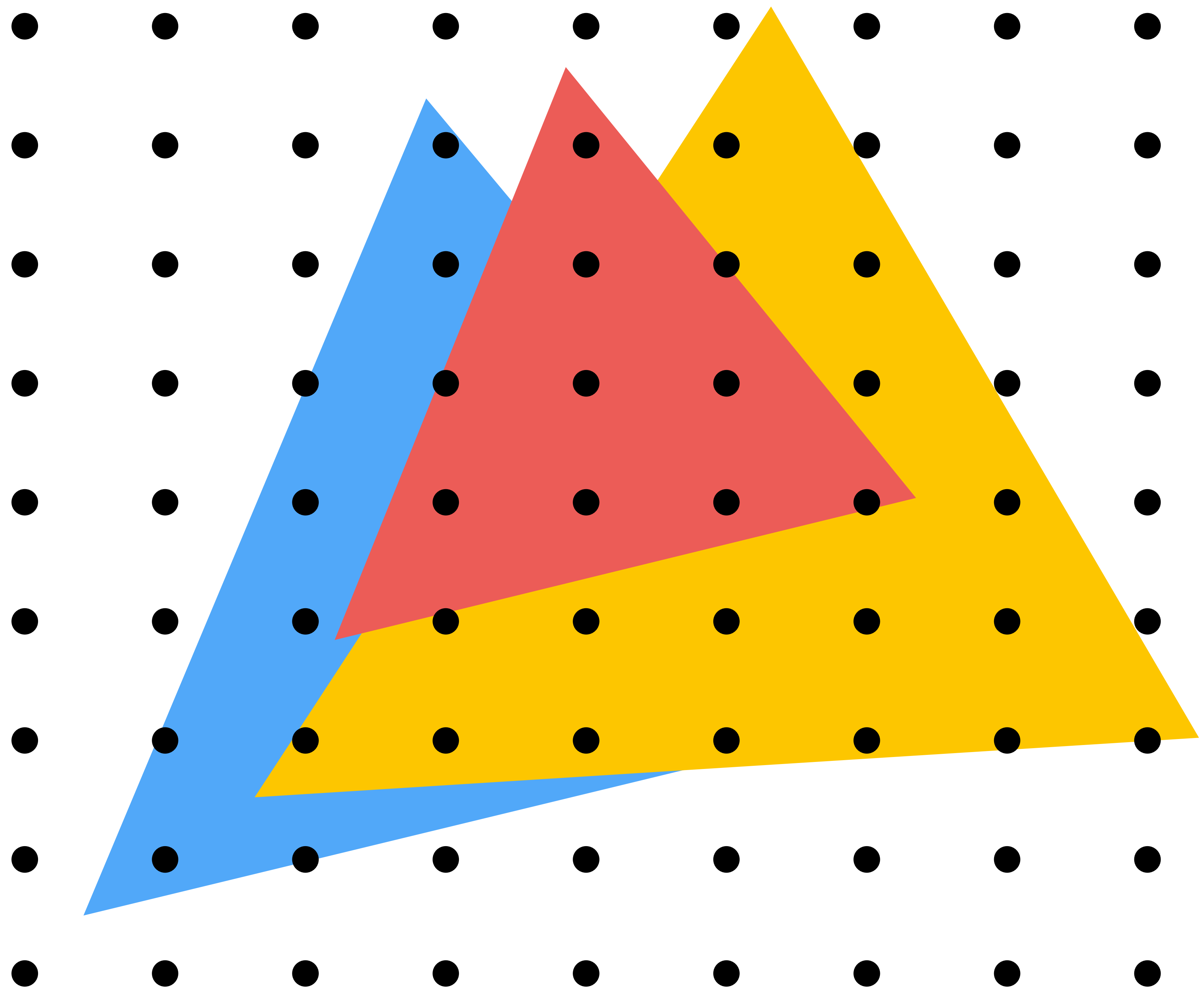
Black = small distance

Red = sample passed depth test



Depth buffer contents

Example: rendering three opaque triangles



Q: Is the result dependent on the order in which triangles are processed?

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
  }
}
```

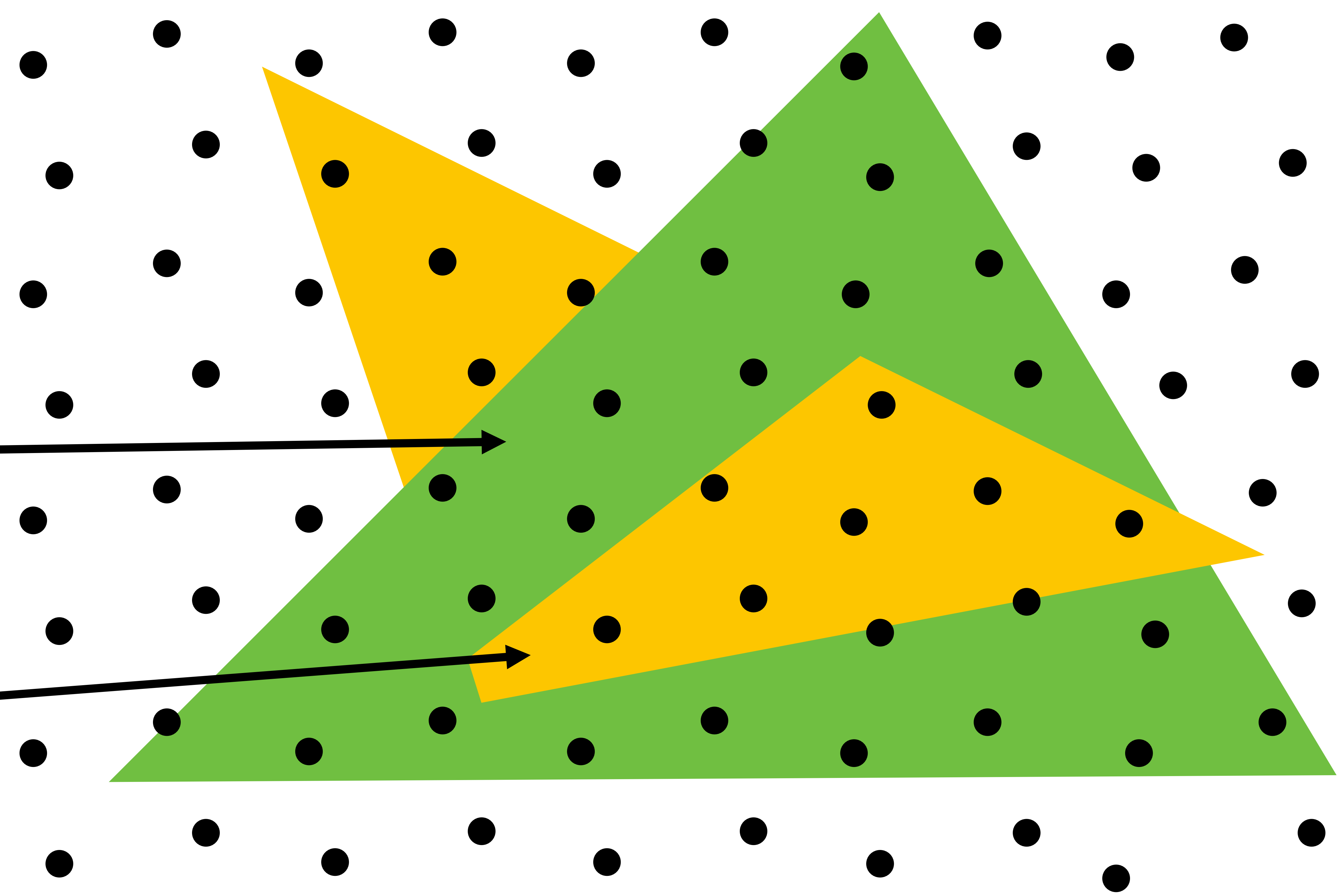
Does the depth-buffer algorithm handle interpenetrating surfaces?

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

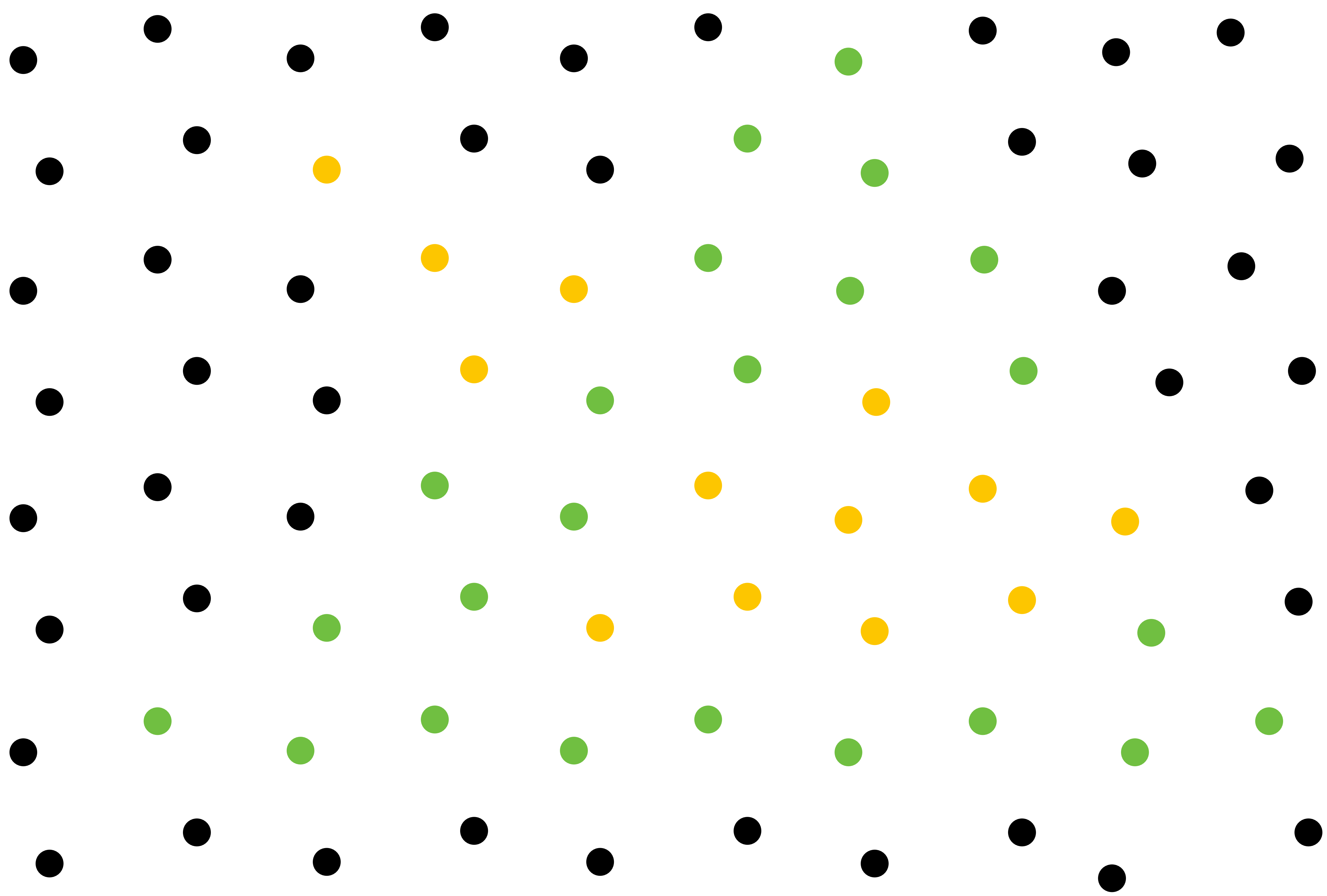
Yellow triangle in front of green triangle



Does the depth-buffer algorithm handle interpenetrating surfaces?

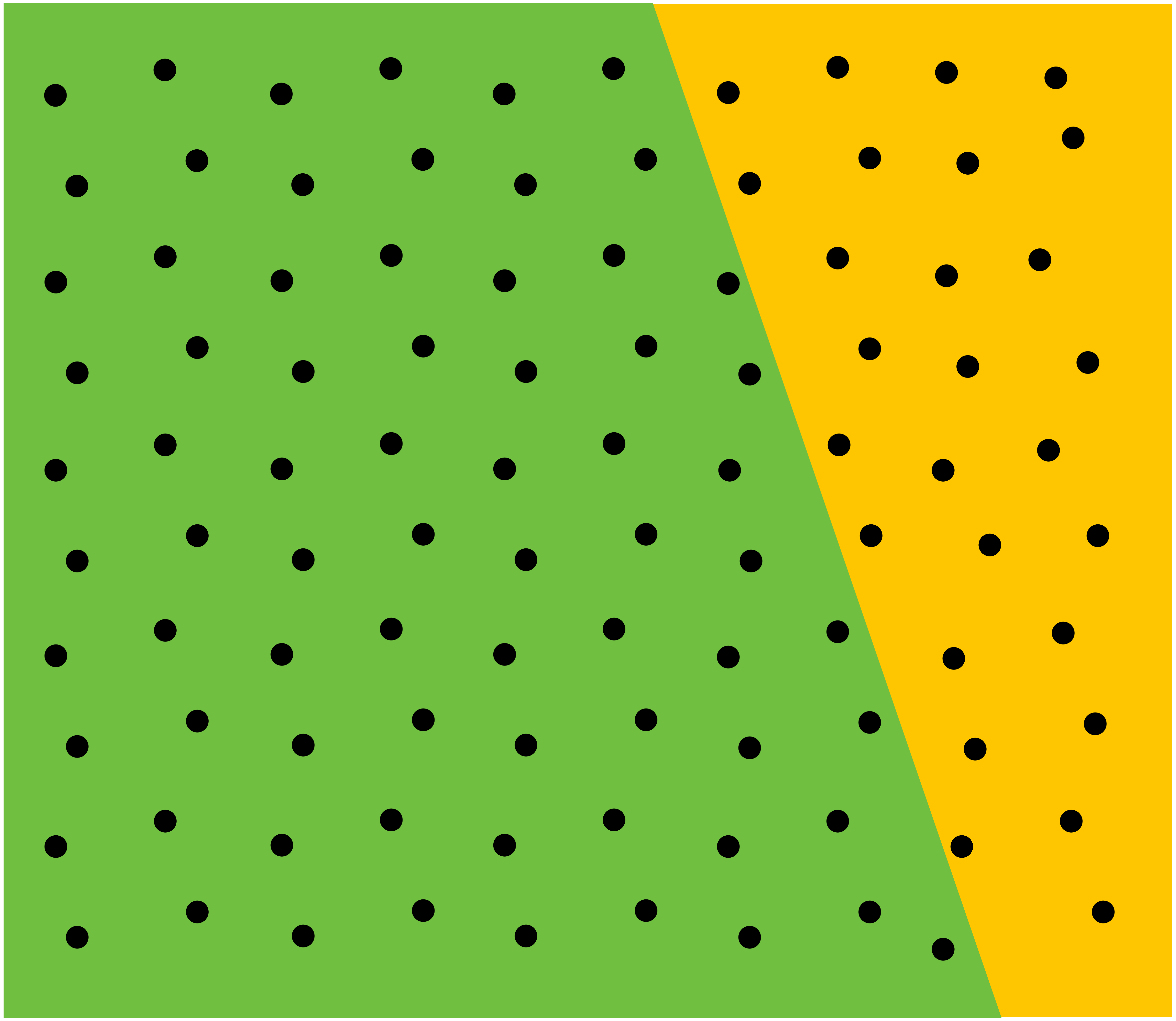
Of course!

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.



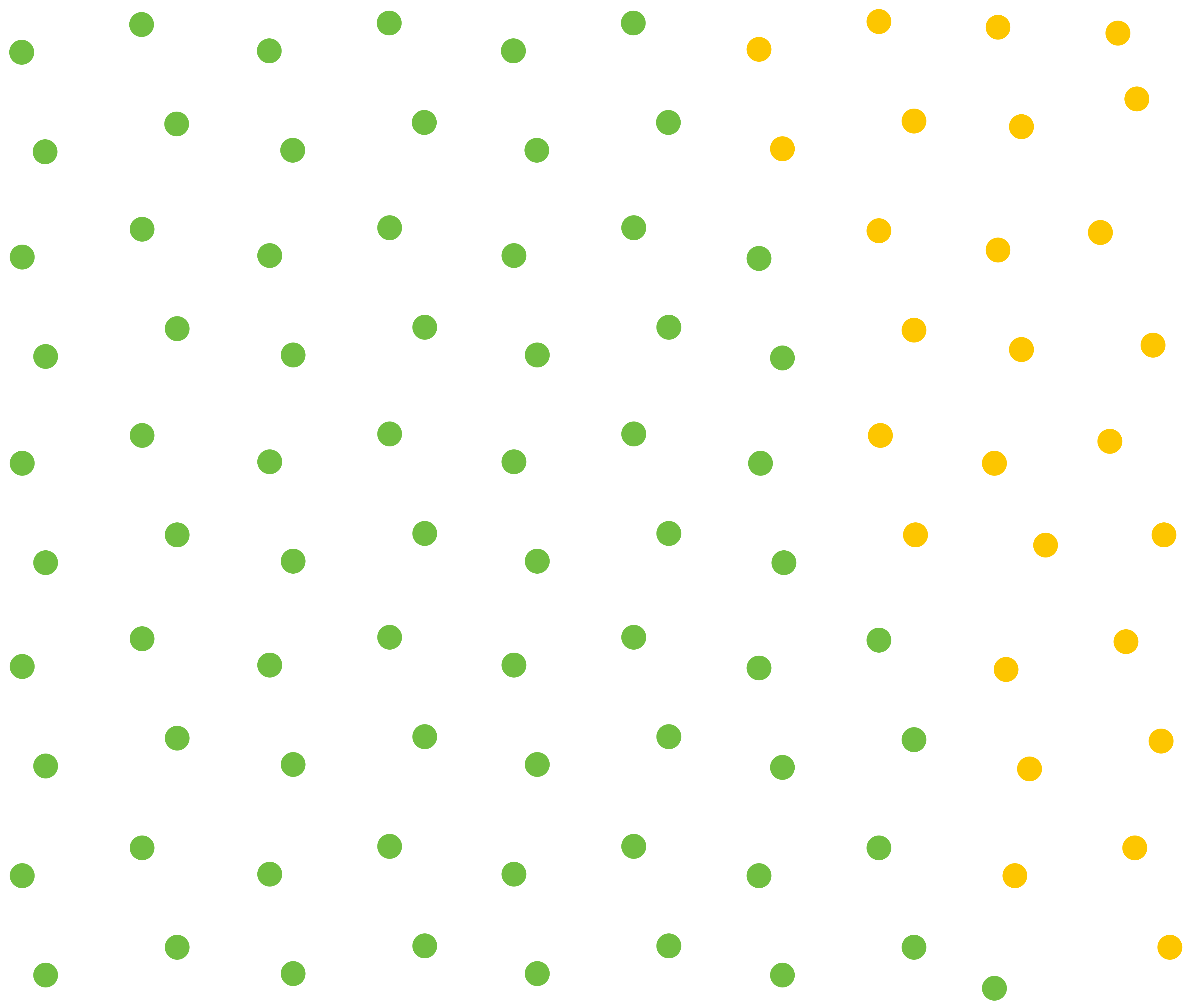
Does it work with super sampling?

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

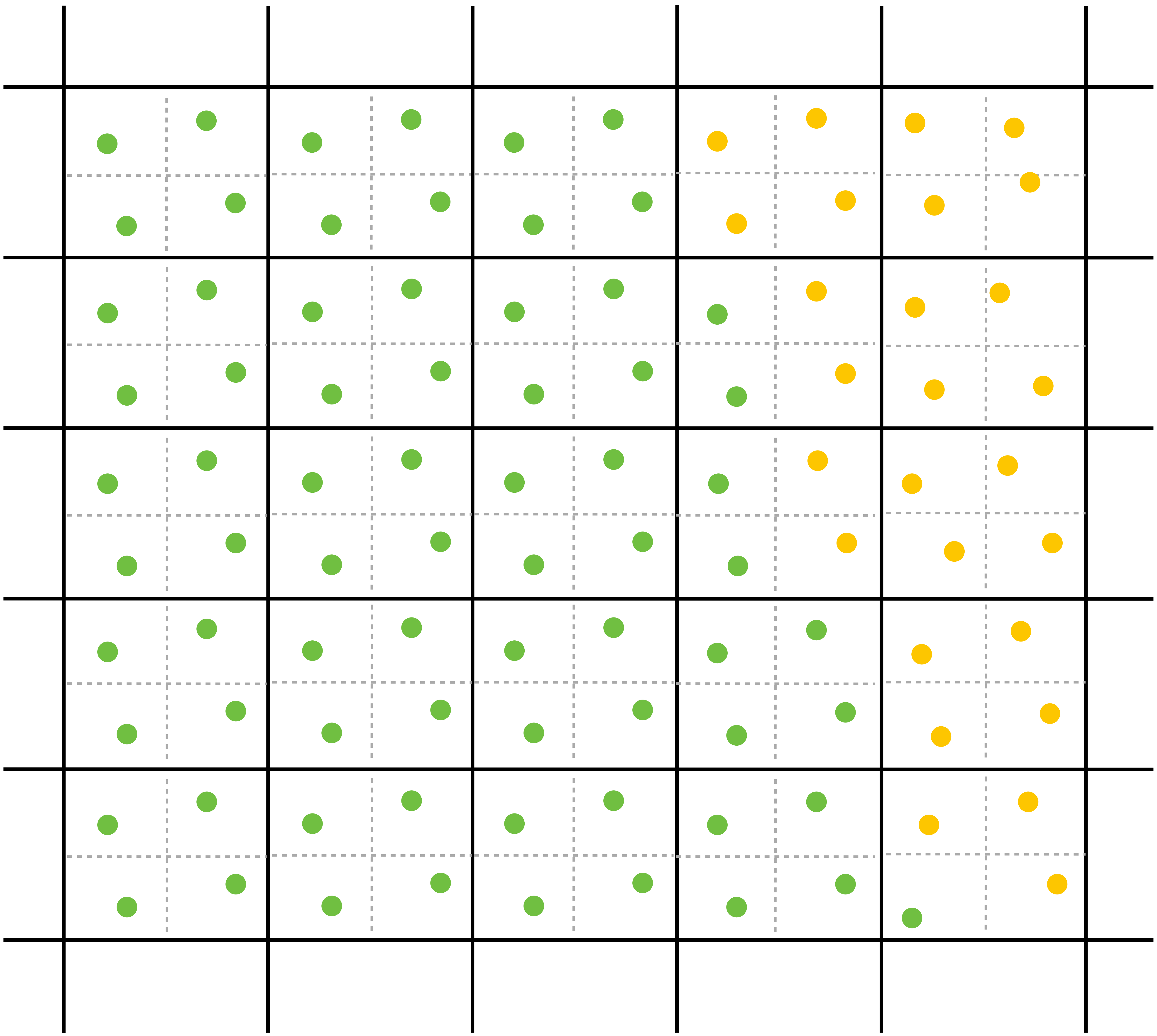


This example: green triangle occludes yellow triangle

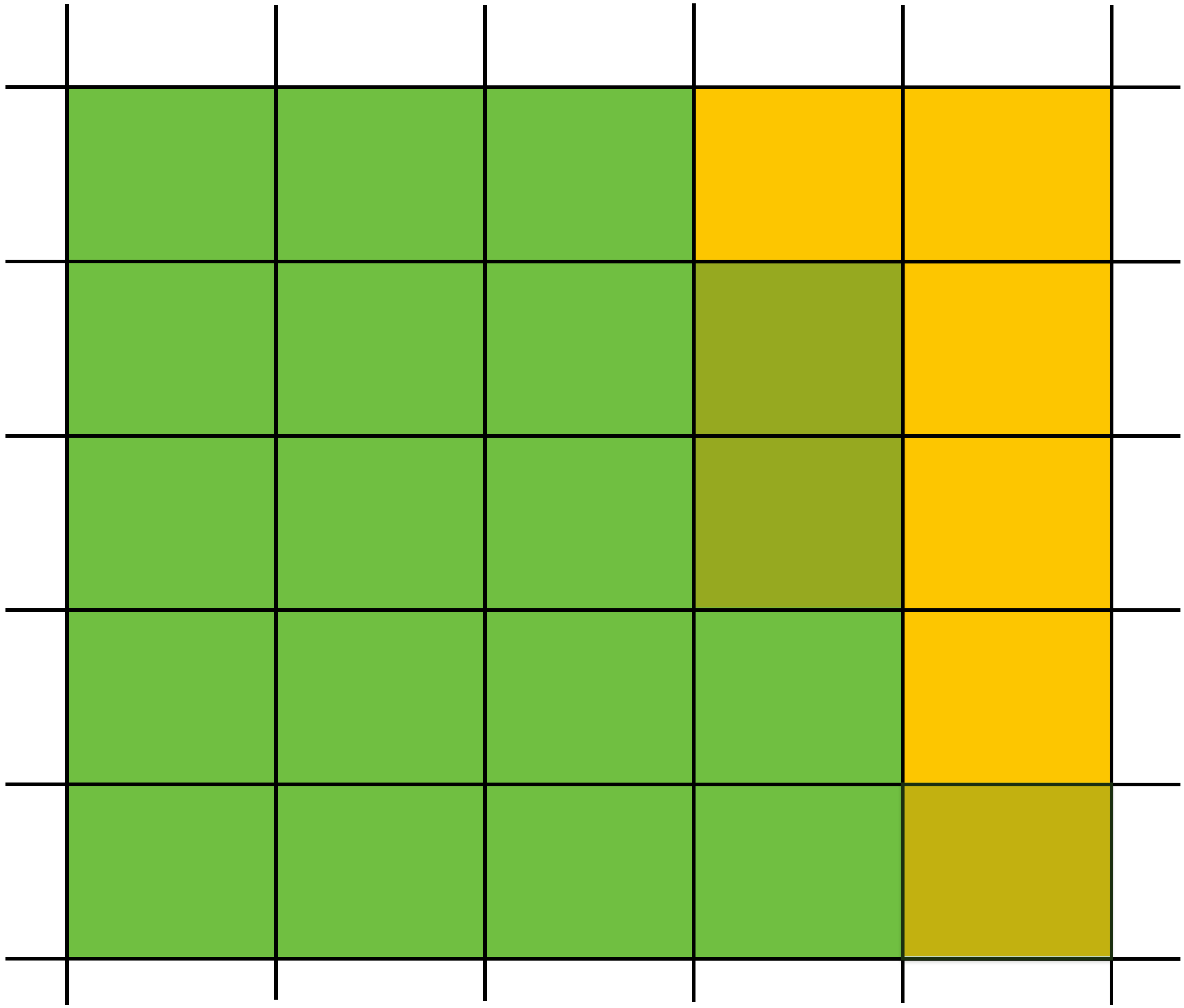
Color buffer contents



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+write of depth buffer if "pass" depth test**
  - **Just a read if "fail"**

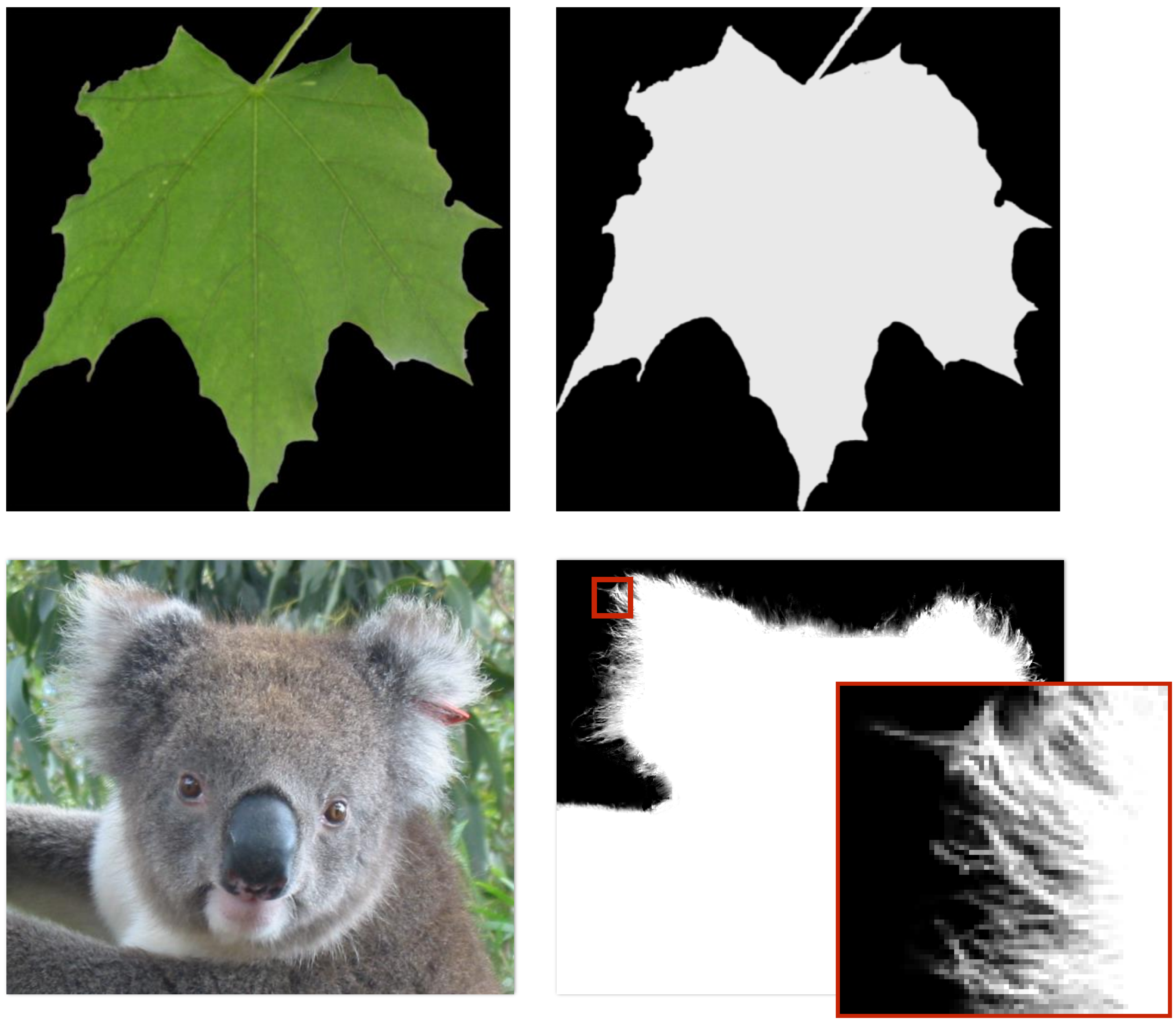
- **Not specific to triangles: only requires that surface depth can be evaluated at a screen sample point**

- **Range of depth values is limited. That's why the near and far planes are used in defining the view frustum!**

But what about semi-transparent objects?

Compositing

Alpha: additional channel of image (rgba)



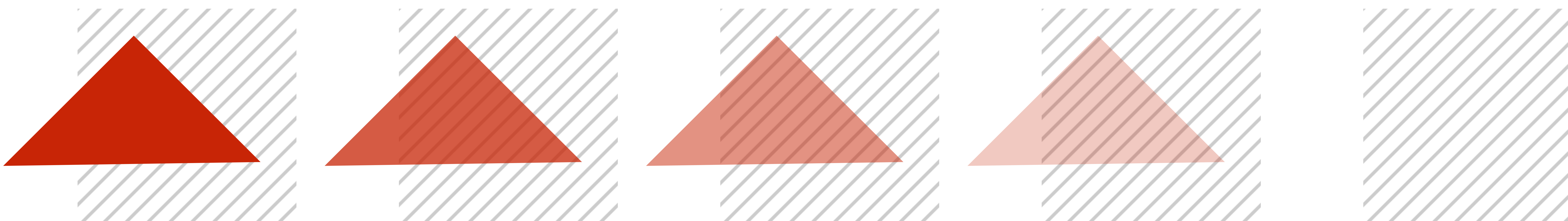
α **of foreground object**

Representing opacity as alpha

Alpha describes the opacity of an object

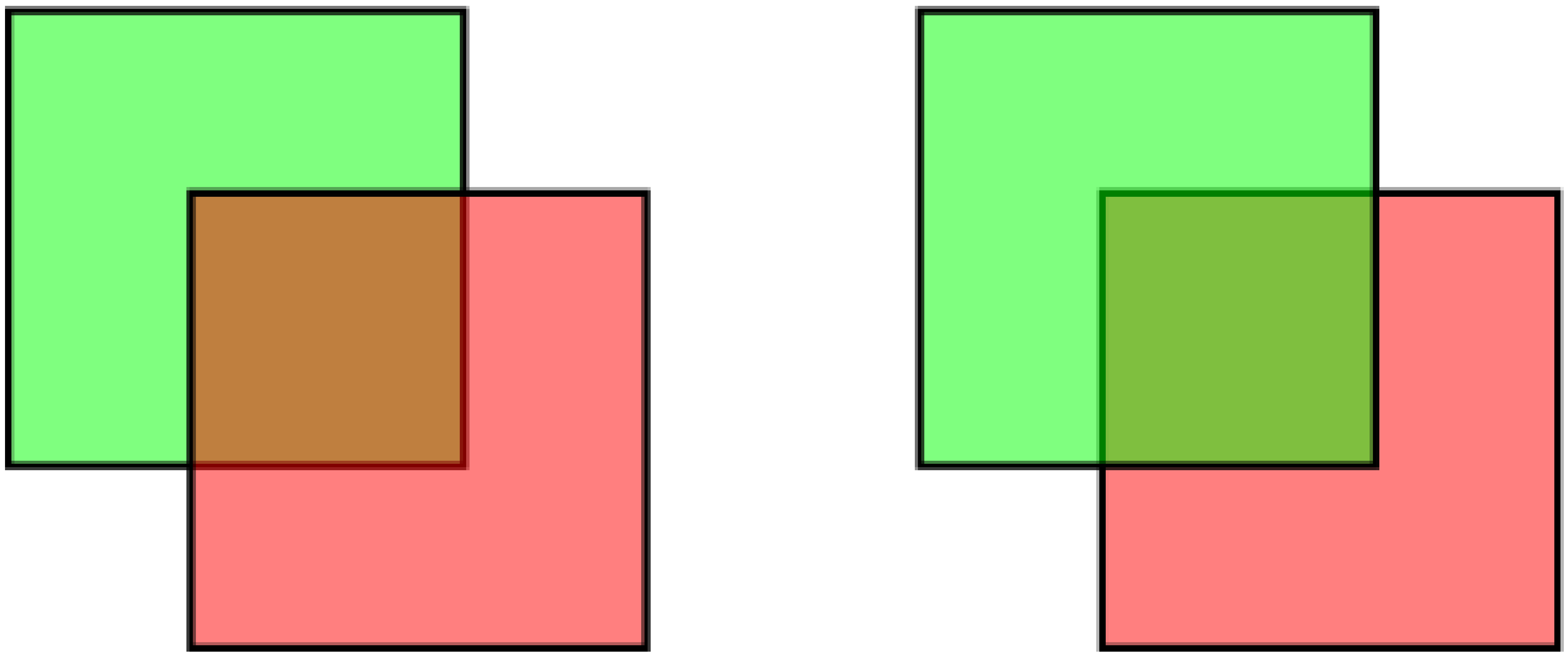
- **Fully opaque surface: α = 1**
- **50% transparent surface: α = 0.5**
- **Fully transparent surface: α = 0**

Red triangle with decreasing opacity



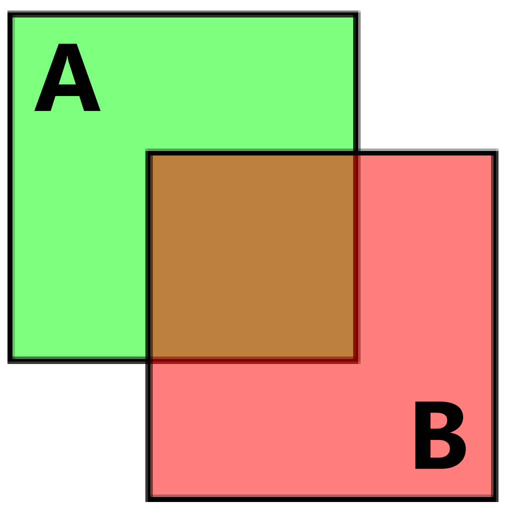
α = 1 | **α = 0.75** | **α = 0.5** | **α = 0.25** | **α =0**

“Over" operator

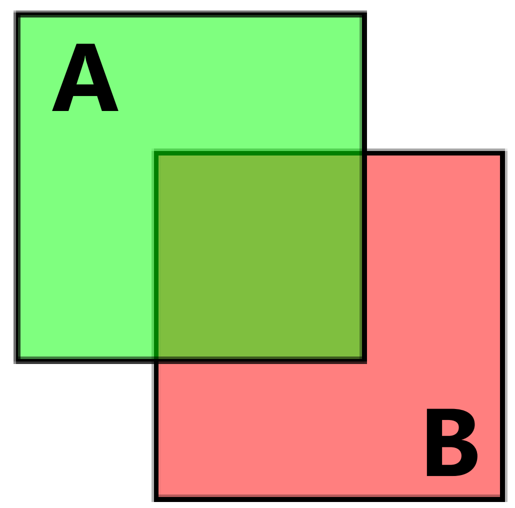


“Over" operator

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

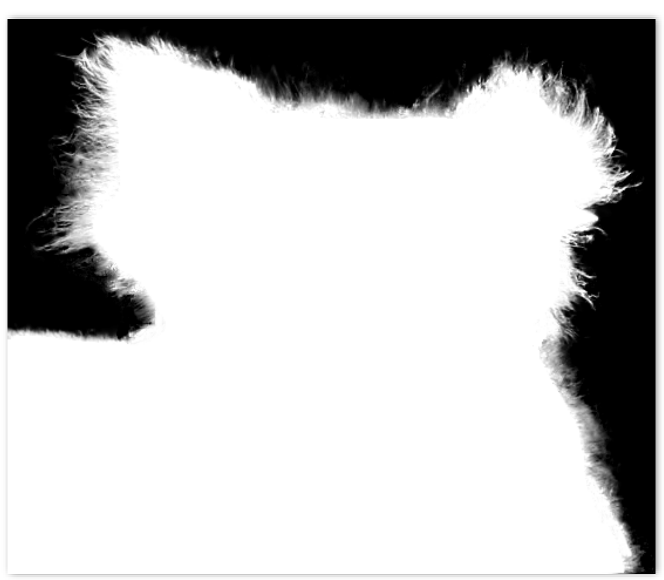


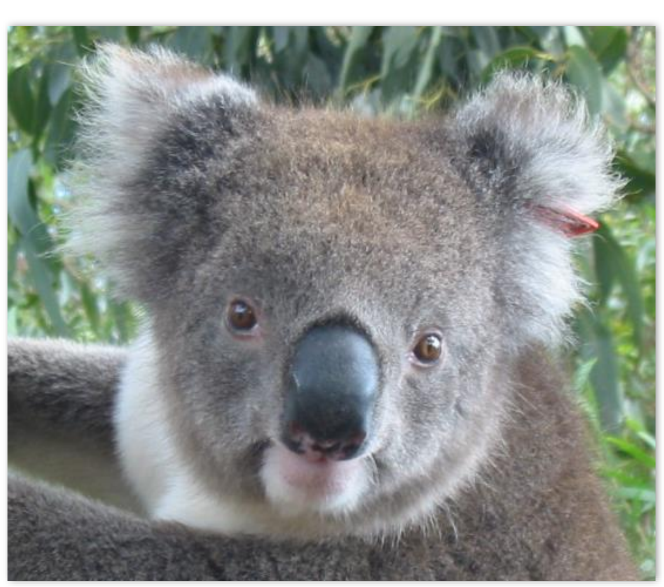
B over A



A over B

A over B != B over A
“Over” is not commutative





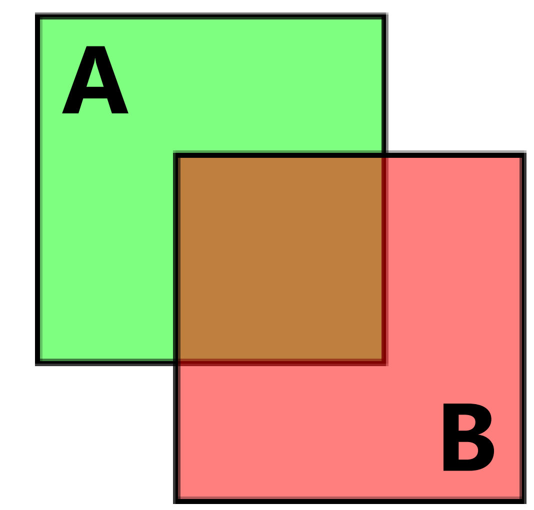
Koala over NYC

“Over" operator

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

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

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



B over A

Appearance of semi-transparent A

Composited color:

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

Appearance of semi-transparent B

What B lets through

A over B != B over A

“Over” is not commutative

What is α_C?

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

“Over" operator

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

First attempt:

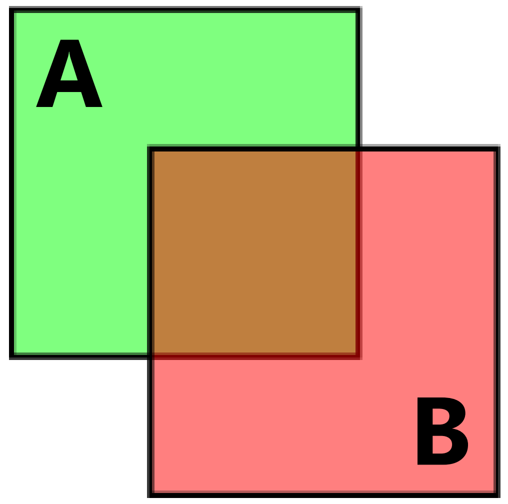
$$A = \begin{bmatrix} A_r & A_g & A_b \end{bmatrix}^T$$

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

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

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

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



B over A

Premultiplied alpha (equivalent):

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

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

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

one multiply, one add

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]);
  }
}
```

← Hmmm, why?

Q: What is the assumption made by this implementation?

Triangles must be rendered in back to front order!

Is this always possible?

Rendering a mixture of opaque and transparent triangles

Step 1: render opaque surfaces using depth-buffered occlusion (If pass depth test passed, triangle overwrites value in color buffer at sample)

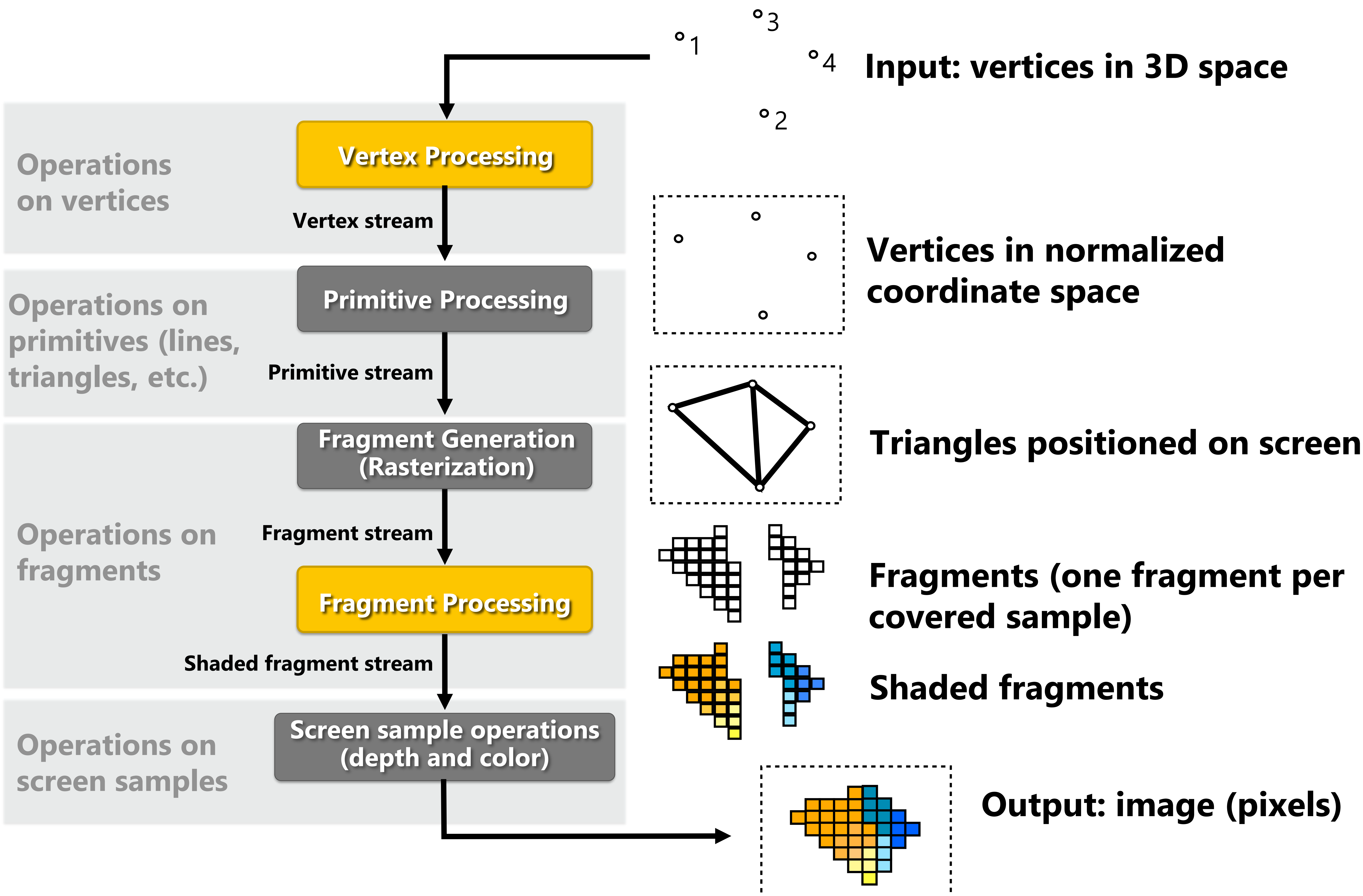
Step 2: disable depth buffer update, render semi-transparent surfaces in back-to-front order. If depth test passed, triangle is composited OVER contents of color buffer at sample

Putting it all together

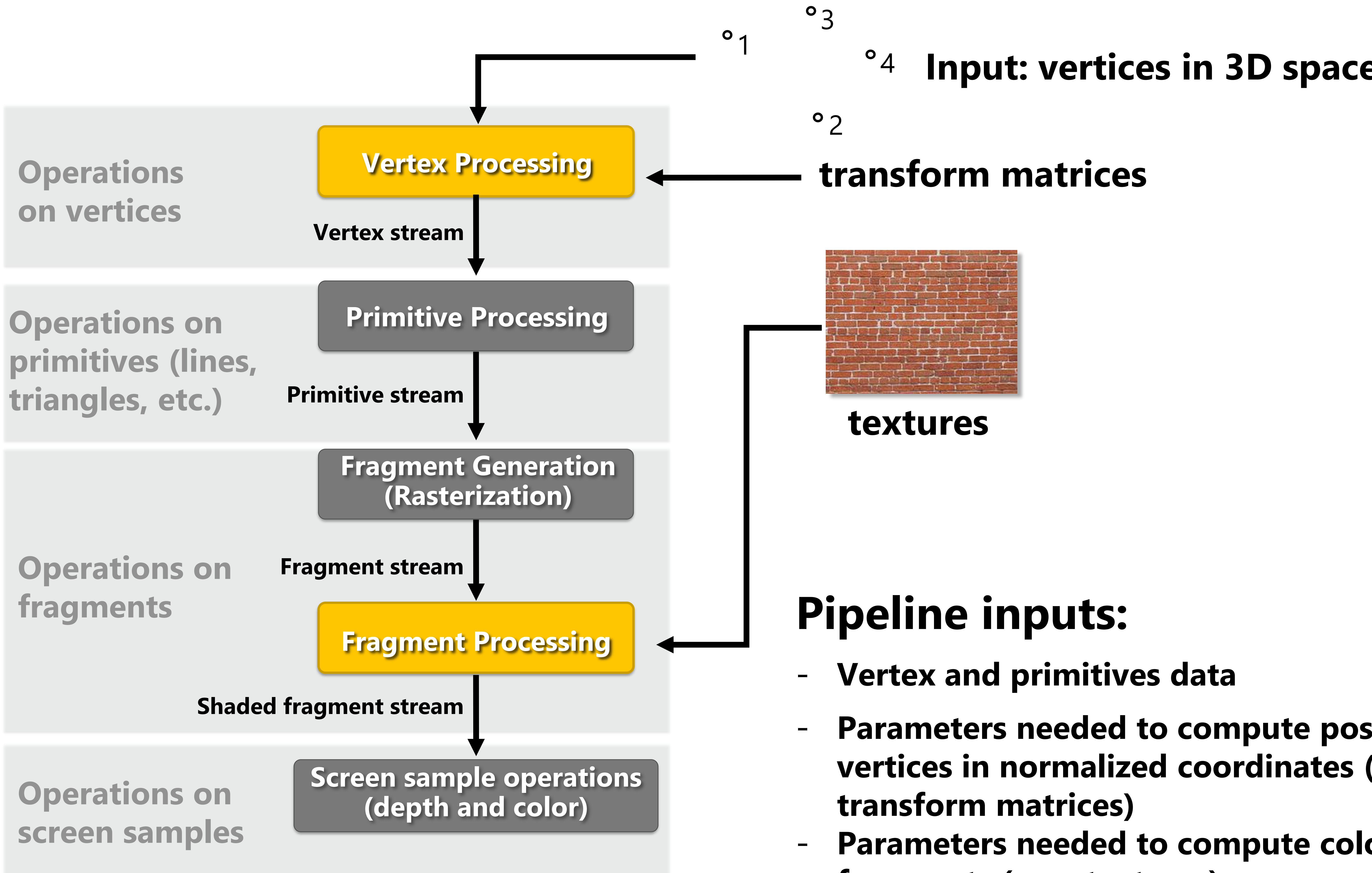
Unreal Engine Kite Demo (Epic Games 2015)

End-to-end rasterization pipeline ("real-time graphics pipeline")

The real-time graphics pipeline



The real-time graphics pipeline



Pipeline inputs:

- Vertex and primitives data
- Parameters needed to compute position of vertices in normalized coordinates (e.g., transform matrices)
- Parameters needed to compute color of fragments (e.g., textures)

Command: draw these triangles!

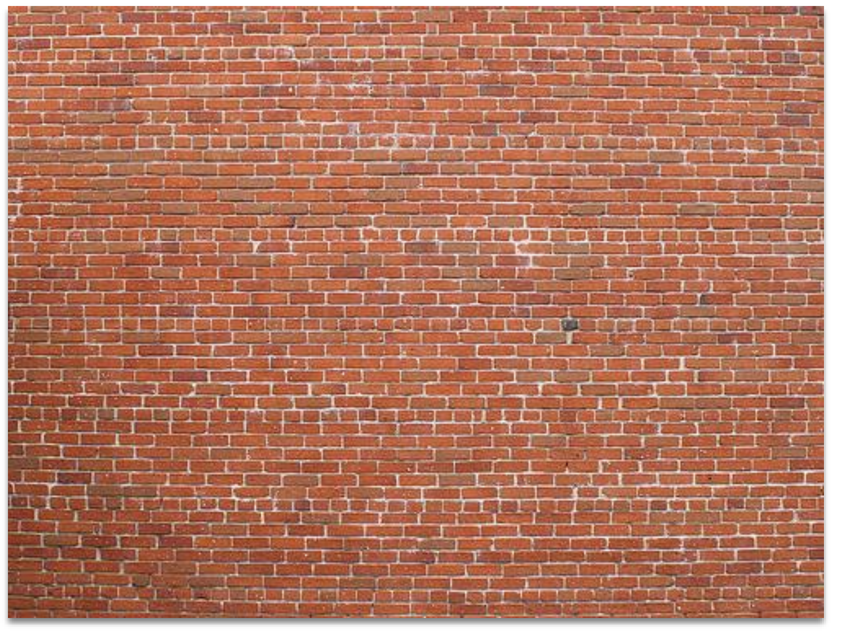
Inputs:

```
list_of_positions = {

    v0x, v0y, v0z,
    v1x, v1y, v1z,
    v2x, v2y, v2z,
    v3x, v3y, v3z,
    v4x, v4y, v4z,
    v5x, v5y, v5z   };
```

```
list_of_texcoords = {

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



Texture map

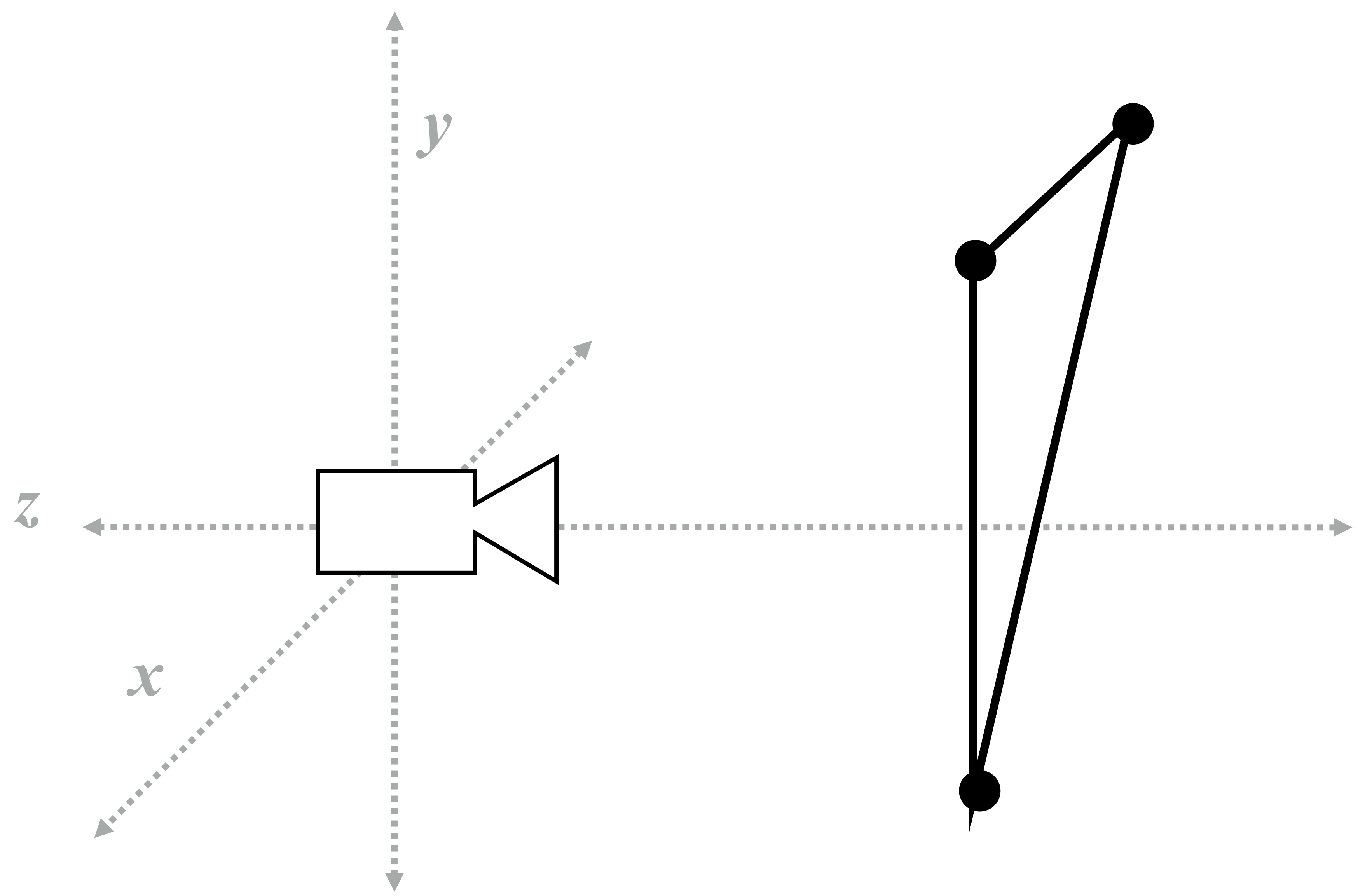
Object-to-camera-space transform *T*

Perspective projection transform *P*

Size of output image (W, H)

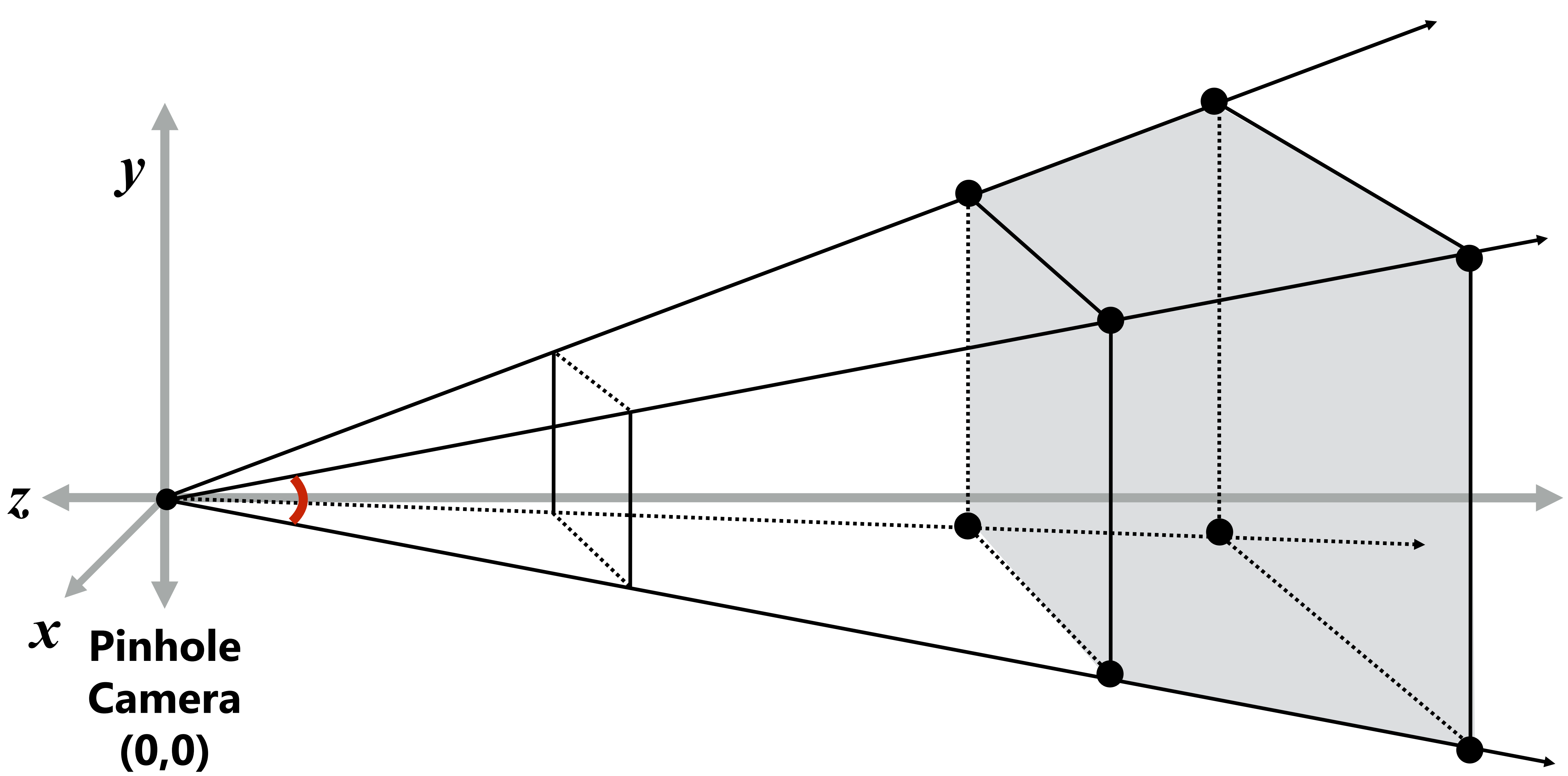
Step 1:

Transform triangle vertices into camera space

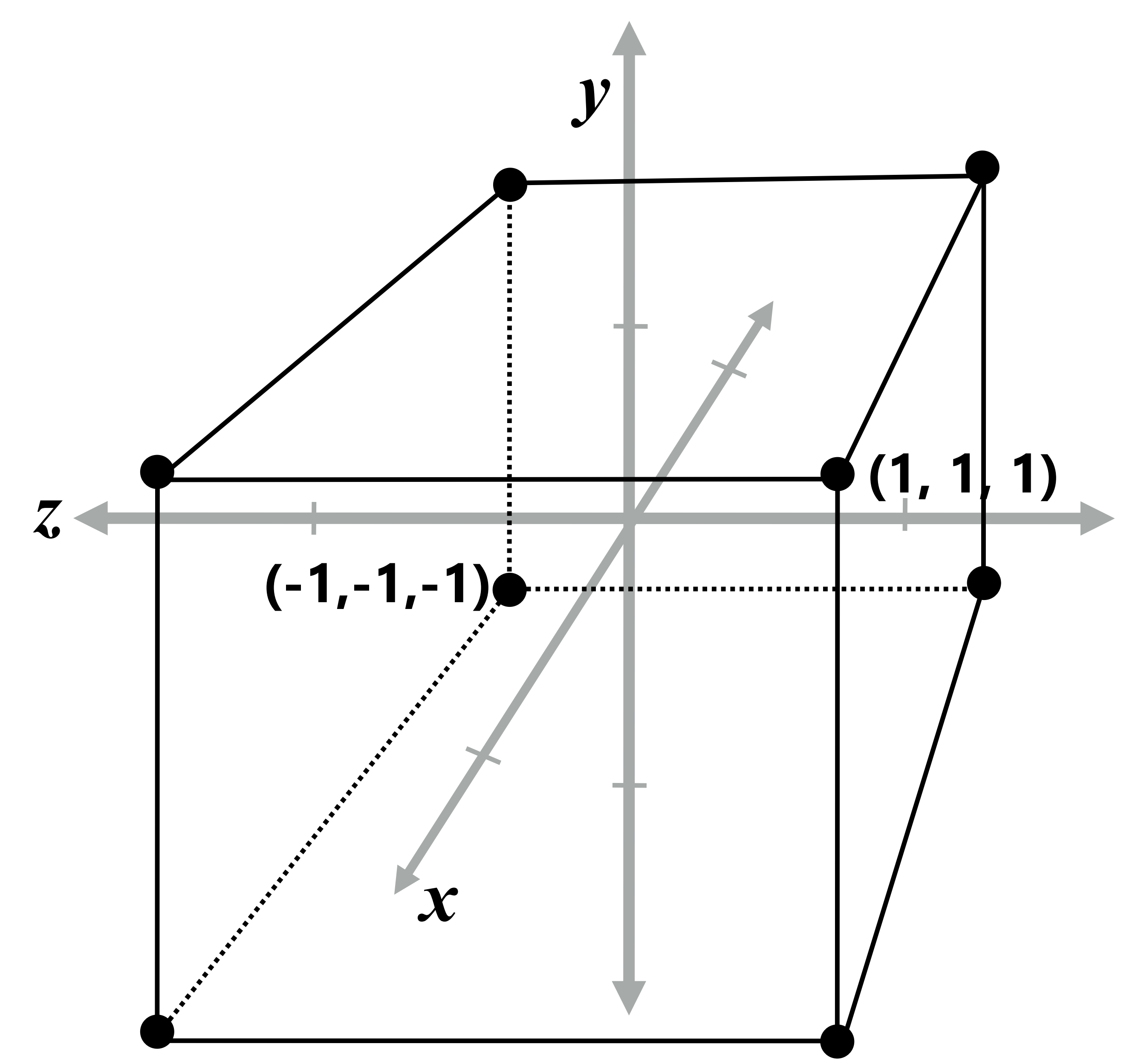


Step 2:

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



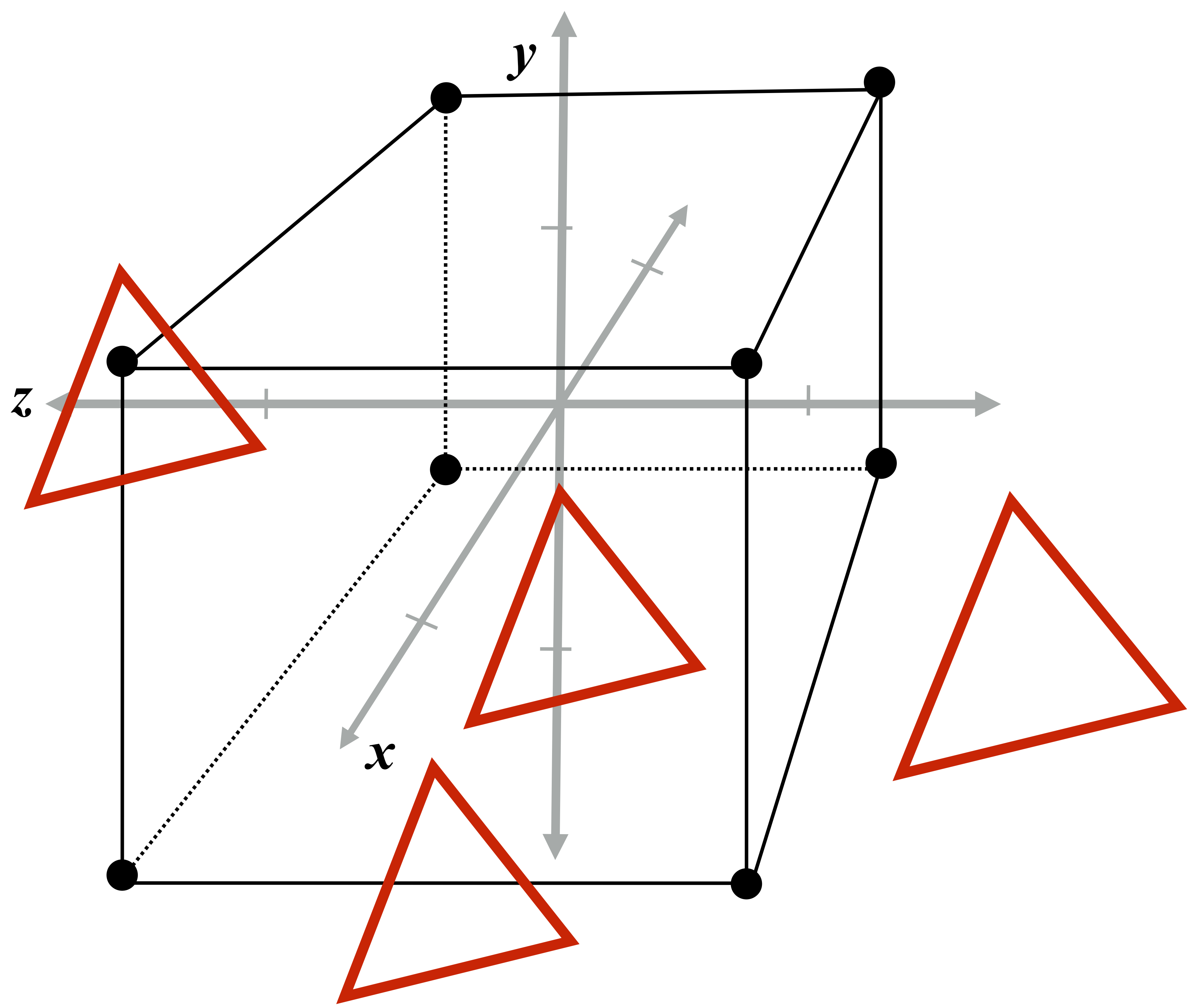
Camera-space positions: 3D



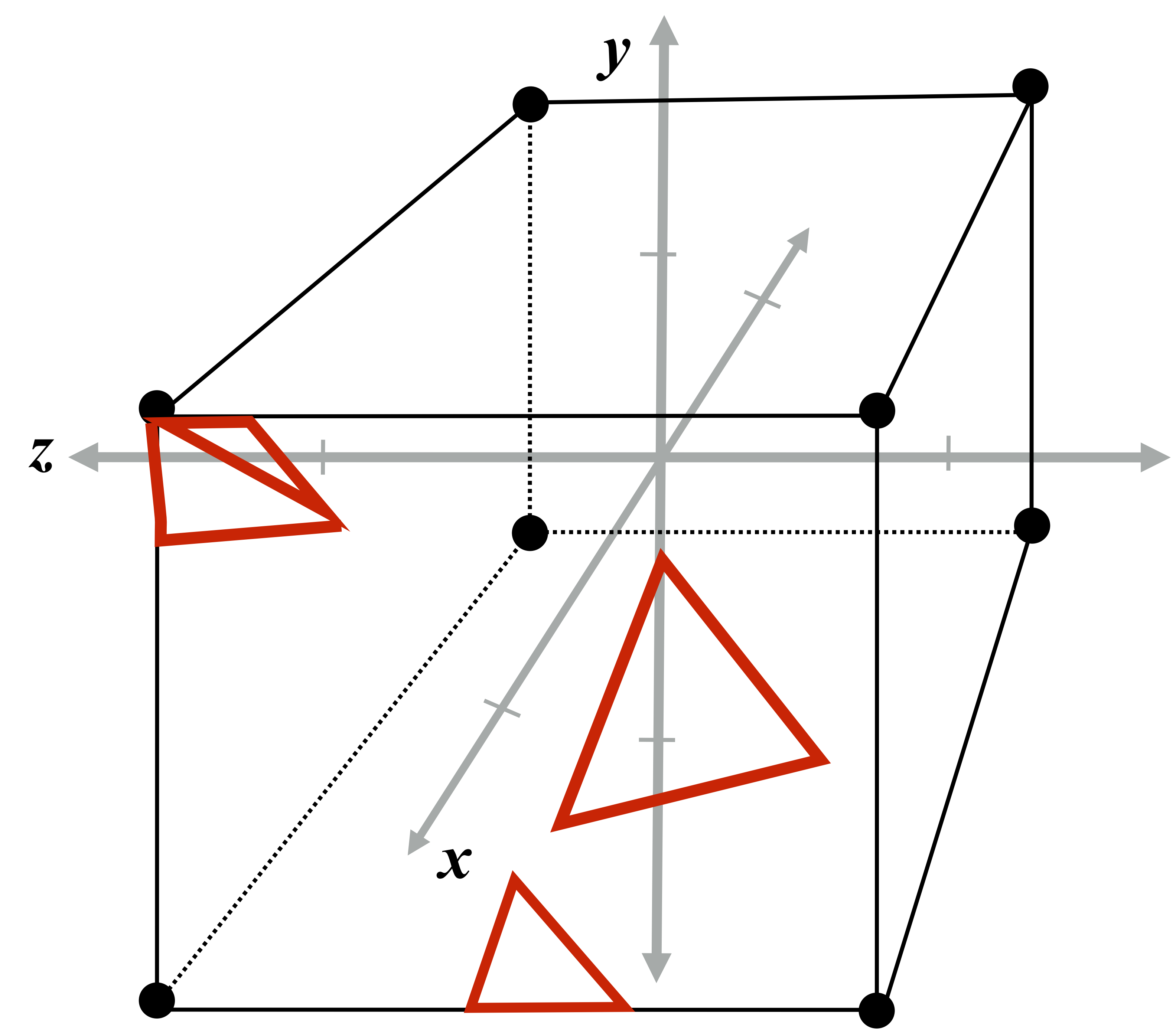
Normalized space positions

Step 3:

- 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



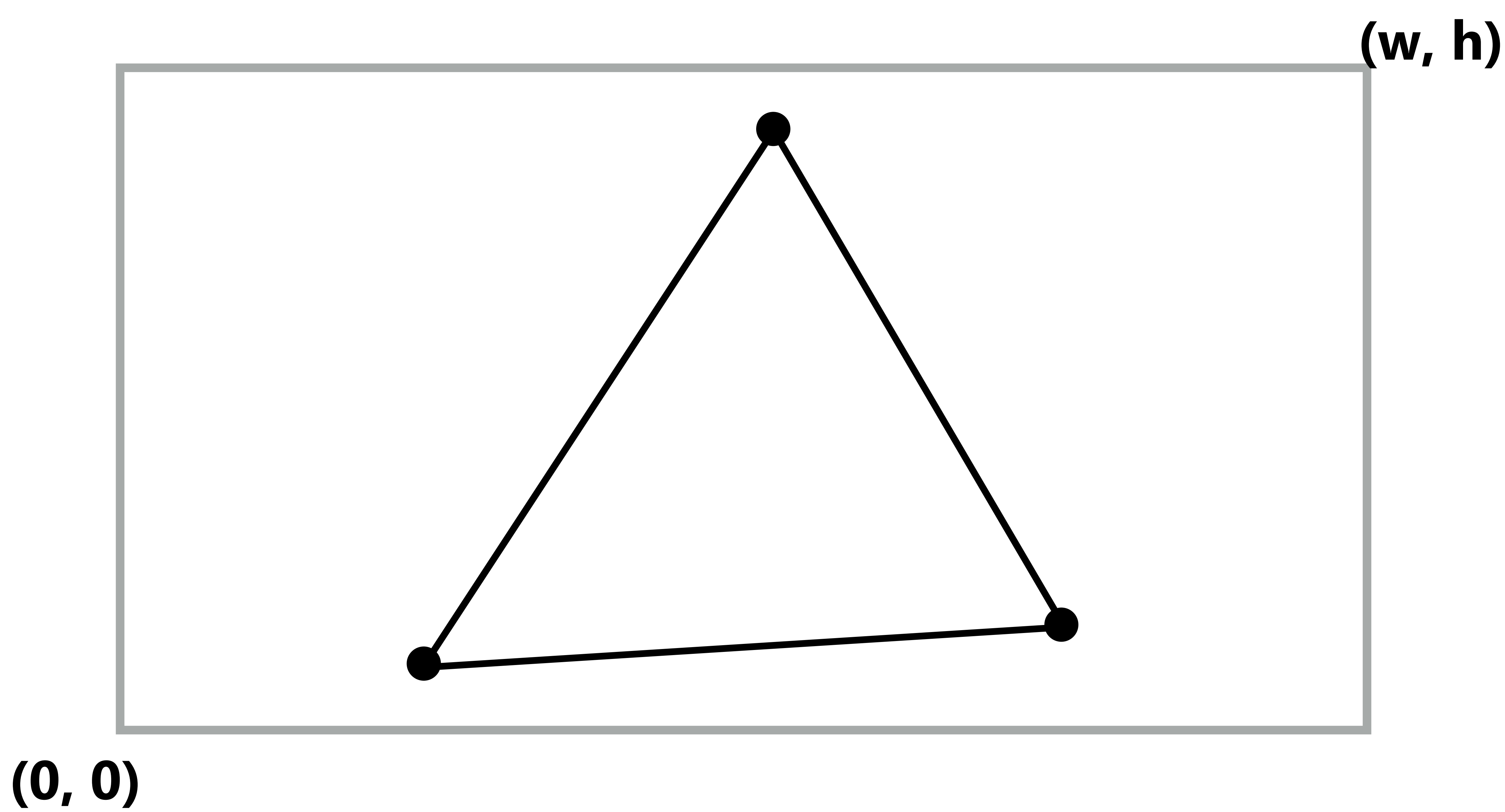
Triangles before clipping



Triangles after clipping

Step 4:

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



Step 5:

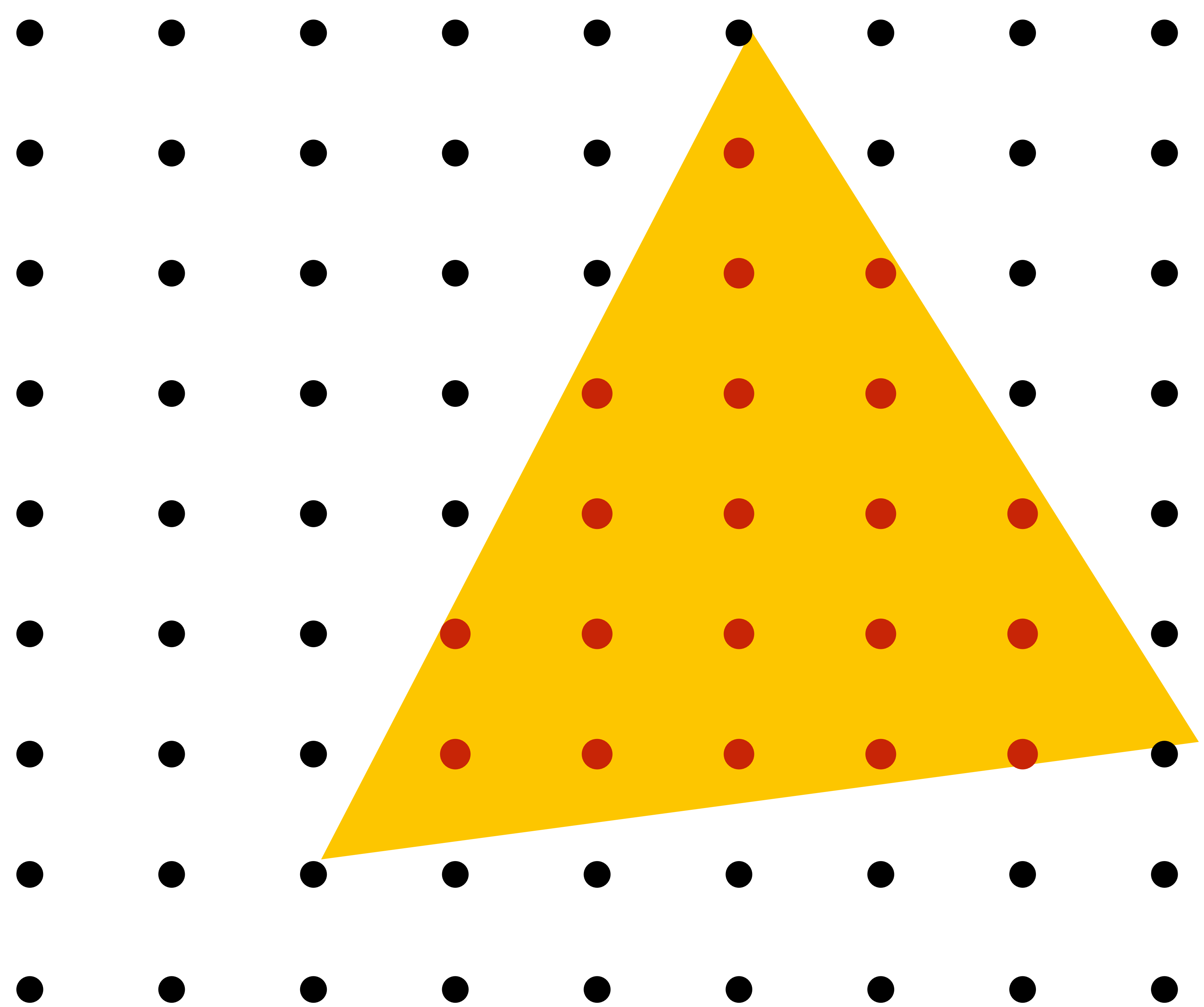
Triangle preprocessing

- **Compute triangle edge equations**
- **Compute triangle attribute equations**

$$\mathbf{E}_{01}(x,y) \qquad \mathbf{U}(x,y)$$
$$\mathbf{E}_{12}(x,y) \qquad \mathbf{V}(x,y)$$
$$\mathbf{E}_{20}(x,y)$$
$$\frac{1}{\mathbf{w}}(x,y)$$
$$\mathbf{Z}(x,y)$$

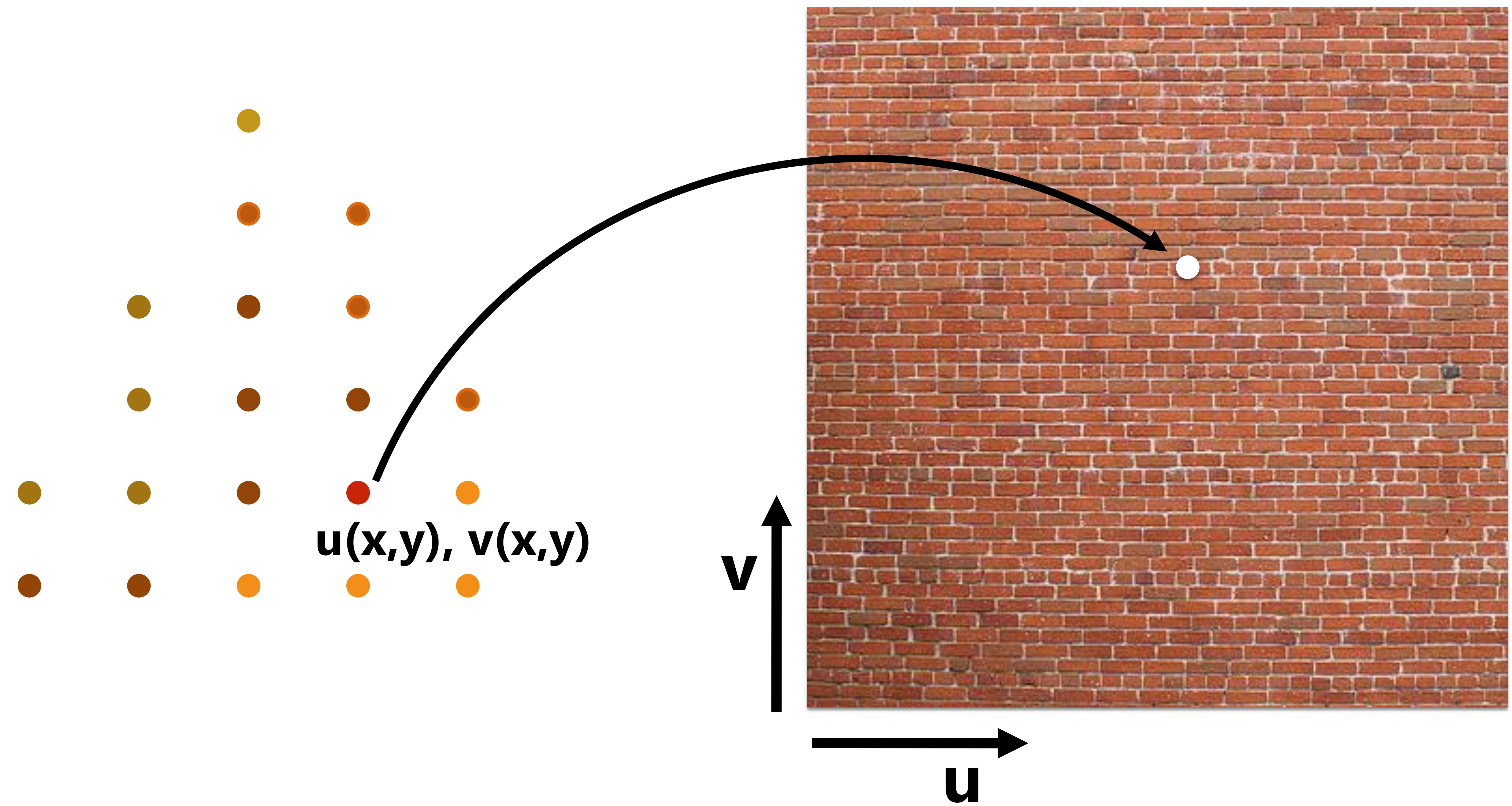
Step 6:

Sample coverage, evaluate attributes Z, u, v at all covered samples



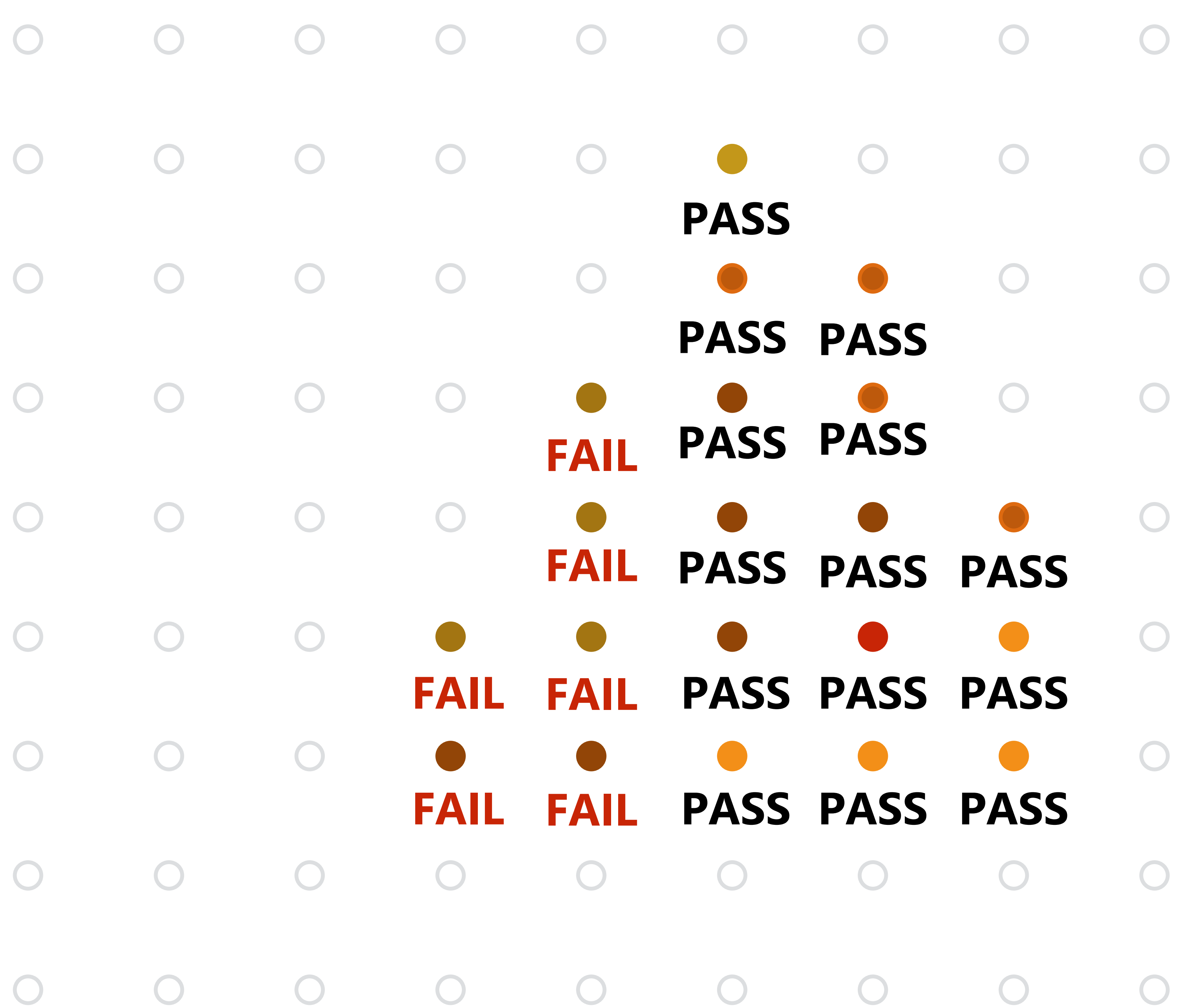
Step 7:

Compute triangle color at sample point (color interpolation, sample texture map, or more advanced shading algorithms)



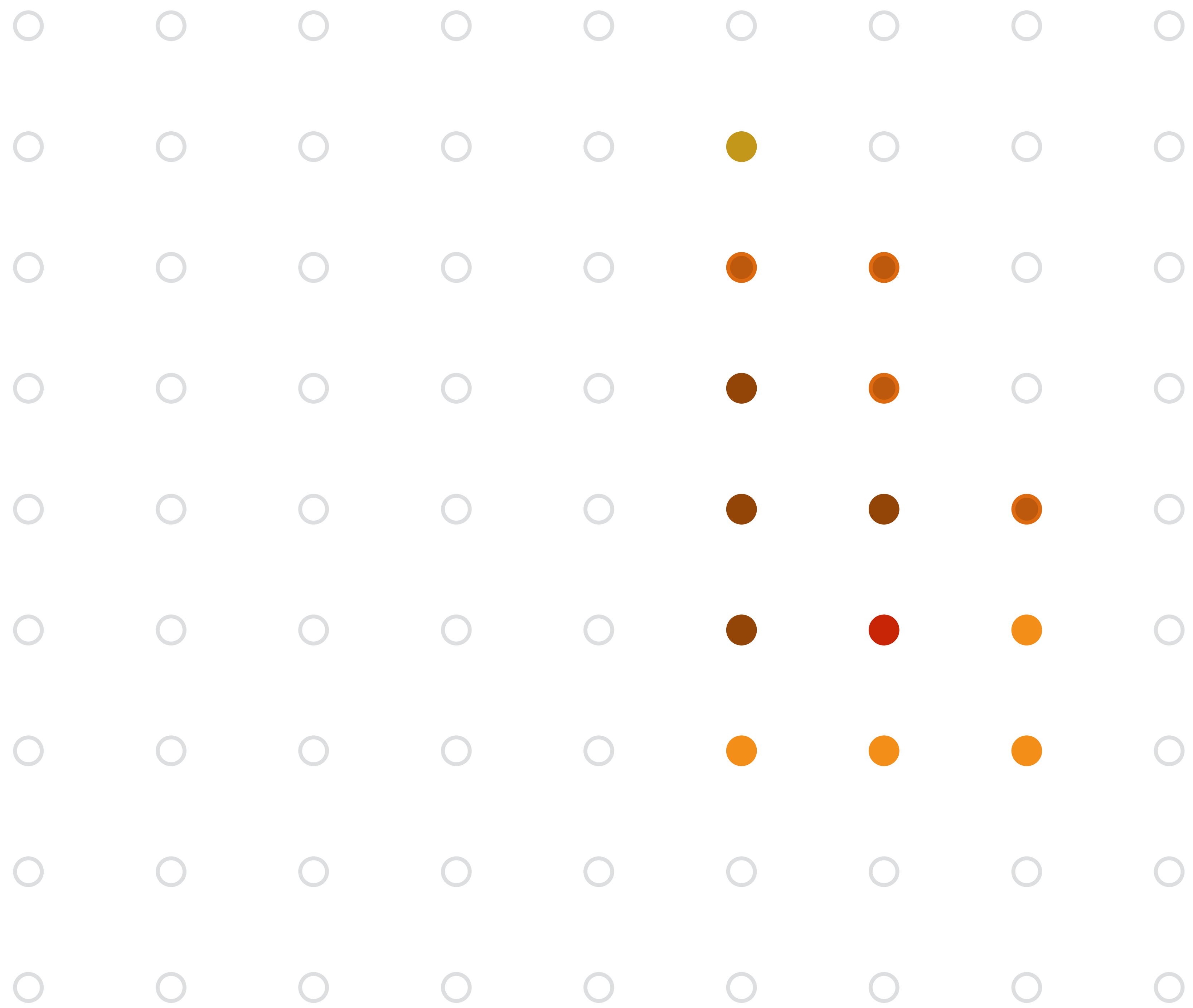
Step 8:

Perform depth test (if enabled) and update depth value at covered samples (if necessary)

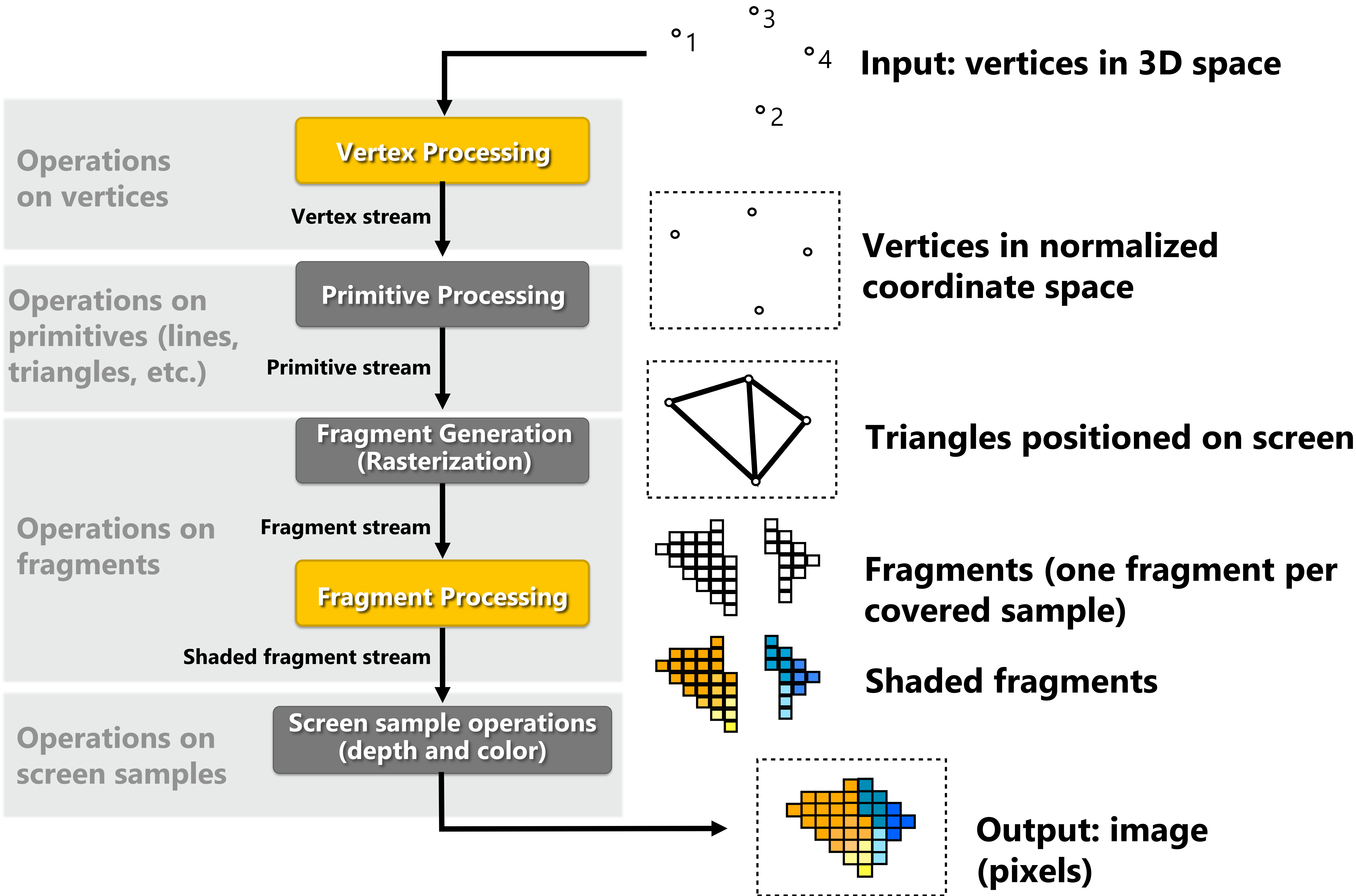


Step 9:

update color buffer (if depth test passed)



OpenGL/Direct3D graphics pipeline*



Note: "Shader" programs 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 that corresponds to fragment.
(this shader performs a texture lookup to obtain the surface's material color at this point, then performs a simple lighting computation)

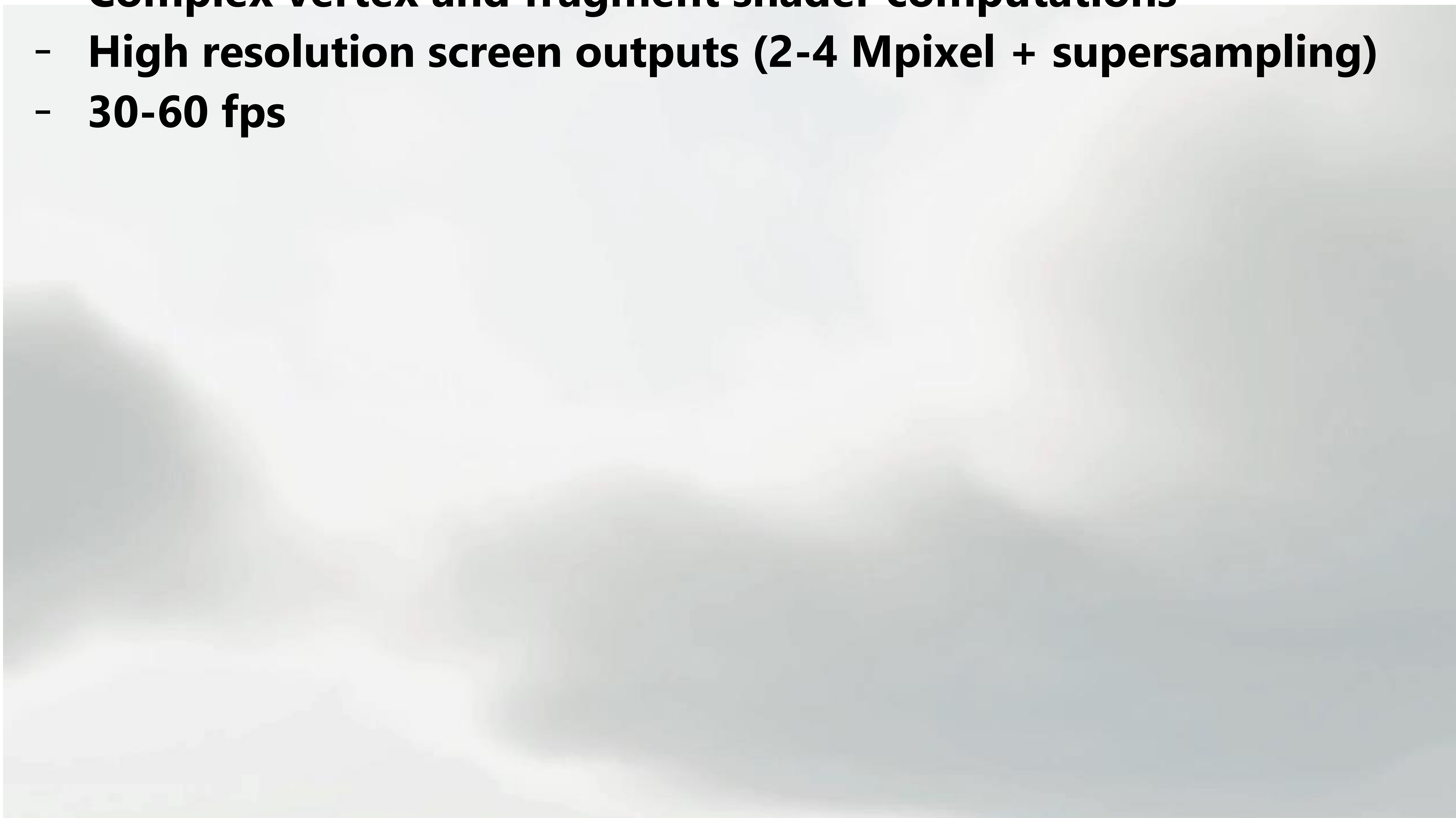
Graphics pipeline hardware implementation: GPUs

Specialized processors for executing graphics pipeline computations

NVIDIA GeForce Titan X

GPUs 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)

Summary

- Occlusion resolved independently at each screen sample using the depth buffer

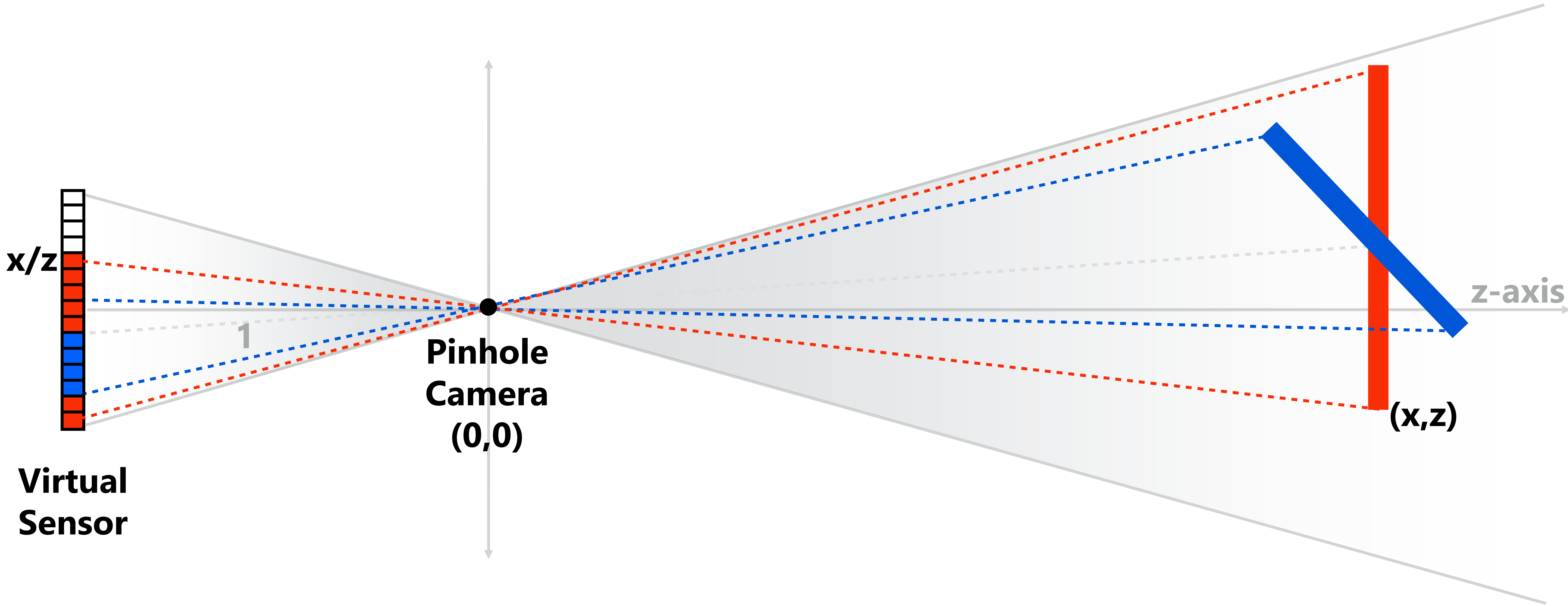
- Alpha compositing for semi-transparent surfaces
  - Premultiplied alpha forms simply repeated composition
  - "Over" compositing operations is not commutative: requires triangles to be processed in back-to-front (or front-to-back) order

- 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)

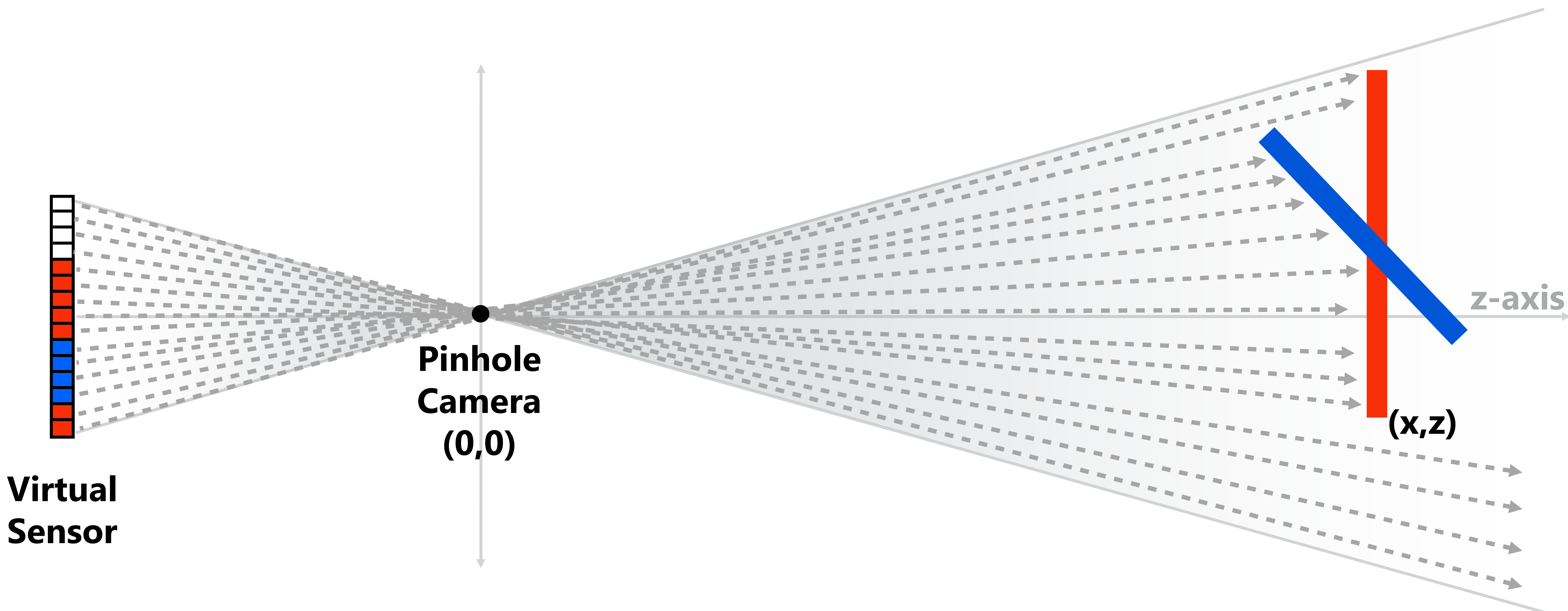
You know “everything” you need to create this image.

But it looks so “flat” ☹

Rasterization



Rendering via ray-casting



We need to compute intersections between rays and the scene

Q: How should occlusions be handled?

Basic ray casting algorithm

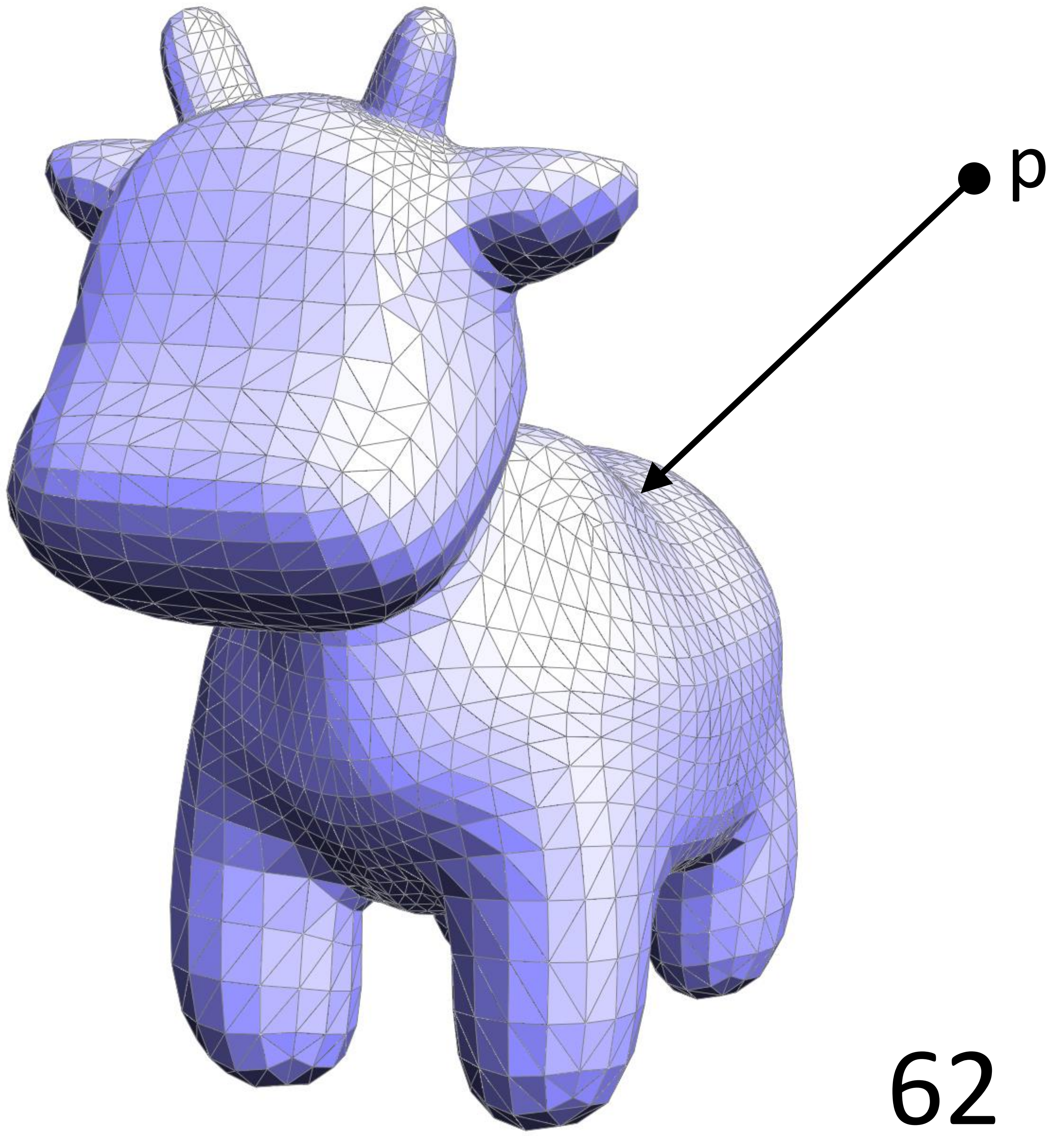
Sample = a ray in 3D

Coverage: does ray "hit" triangle (ray-triangle intersection tests)

Occlusion: closest intersection along ray

Q: What should happen once the point hit by a ray is found?

Q: What are the main differences between rasterization and ray casting?



Rasterization vs. ray casting

- **Rasterization:**
  - Proceeds in triangle order
  - Most processing is based on 2D primitives (3d geometry projected into screen space)
  - Store depth buffer (random access to regular structure of fixed size)

- **Ray casting:**
  - Proceeds in screen sample order
    - Never have to store depth buffer (just current ray)
    - Natural order for rendering transparent surfaces (process surfaces in the order the are encountered along the ray: front-to-back or back-to-front)
  - Must store entire scene (random access to irregular structure of variable size: depends on complexity and distribution of scene)

- **Conceptually, compared to rasterization approach, ray casting is just a reordering of loops + math in 3D**

Rasterization and ray casting are two approaches for solving the same problem: determining "visibility"

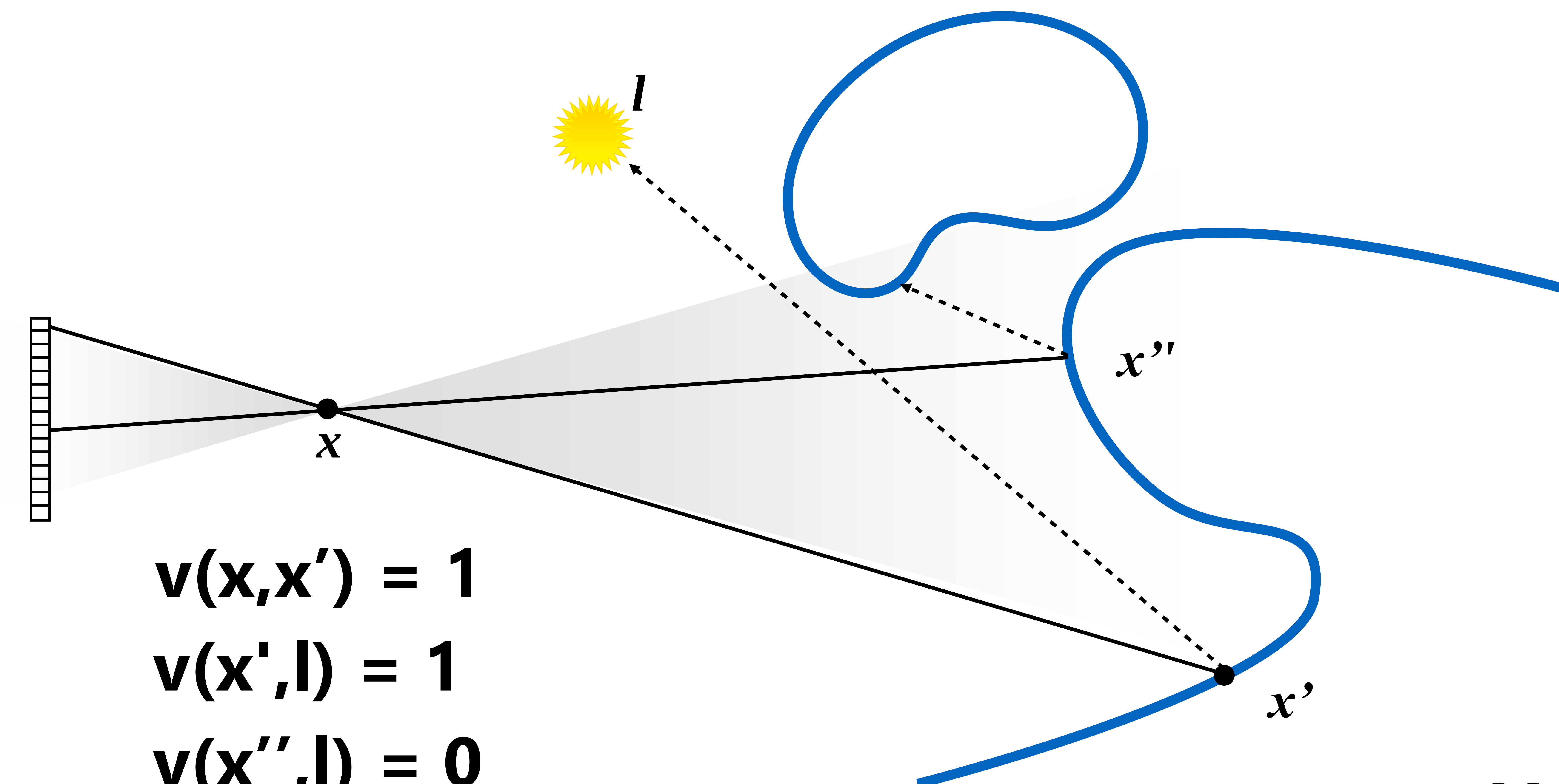
Rasterization vs Ray tracing

“loop over primitives”

“loop over screen pixels”

Ray tracing: a more general mechanism for answering "visibility" queries

$v(x_1,x_2) = 1$ if x_1 is visible from x_2, 0 otherwise



$v(x,x') = 1$

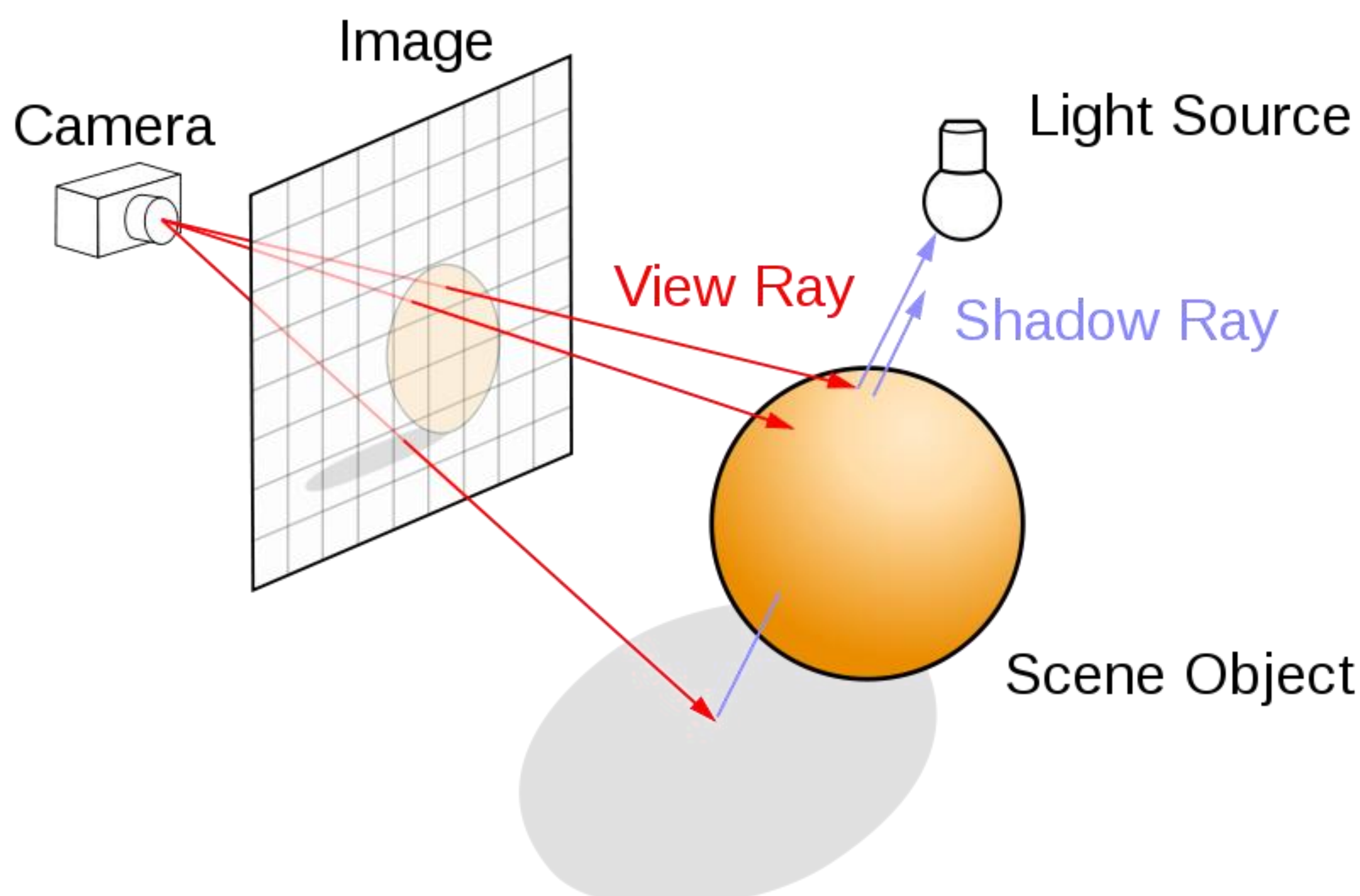
$v(x',l) = 1$

$v(x'',l) = 0$

Shadows: ray tracing

Recursive ray tracing

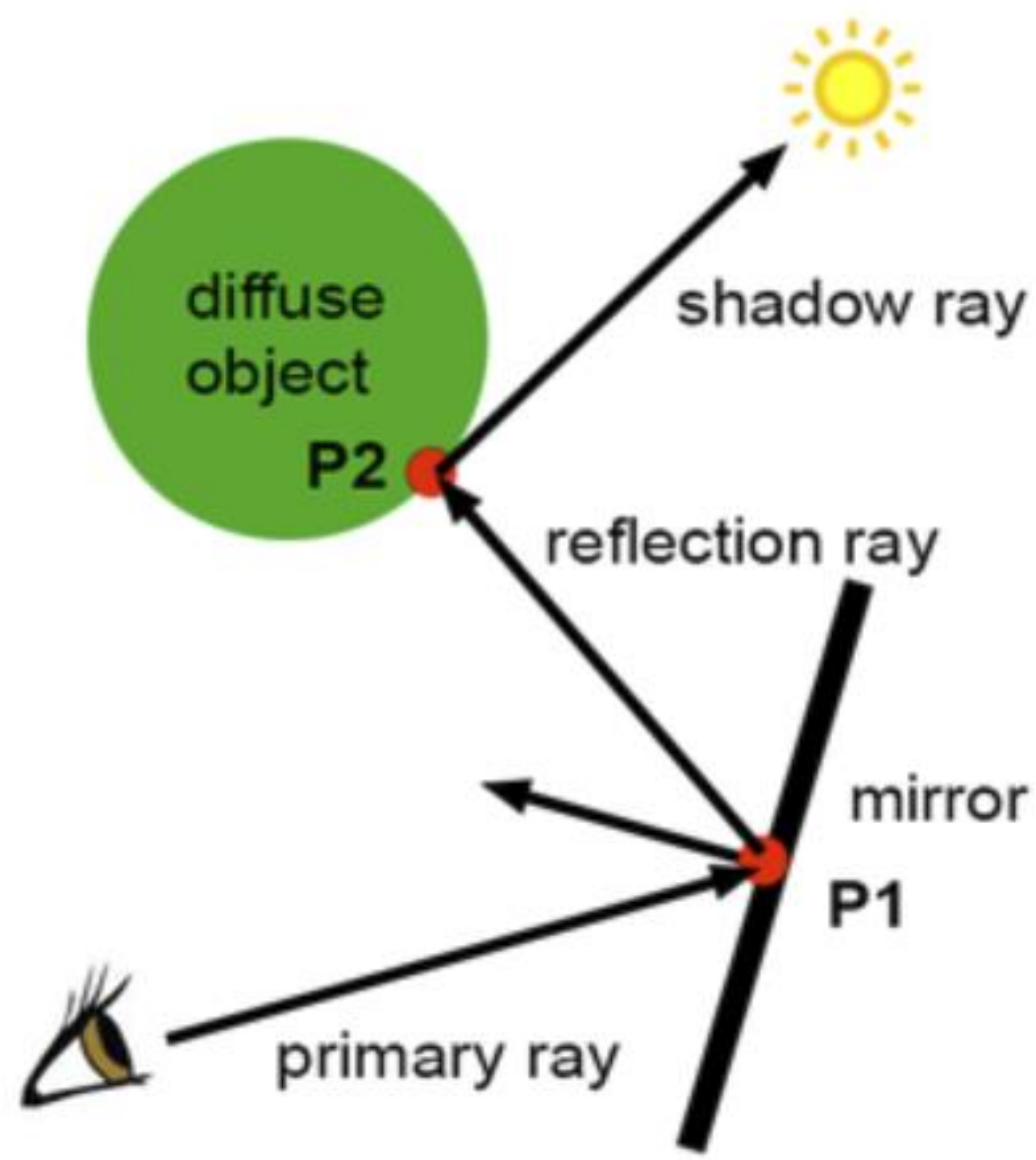
- shoot “shadow” rays towards light source from points where camera rays intersect scene
  - If unconcluded, point is directly lit by light source



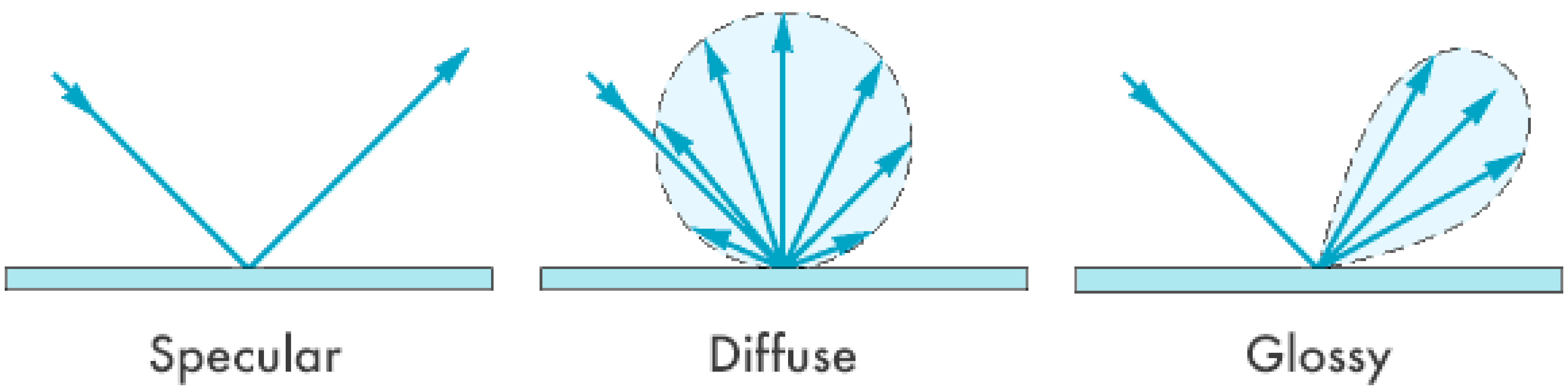
Reflections

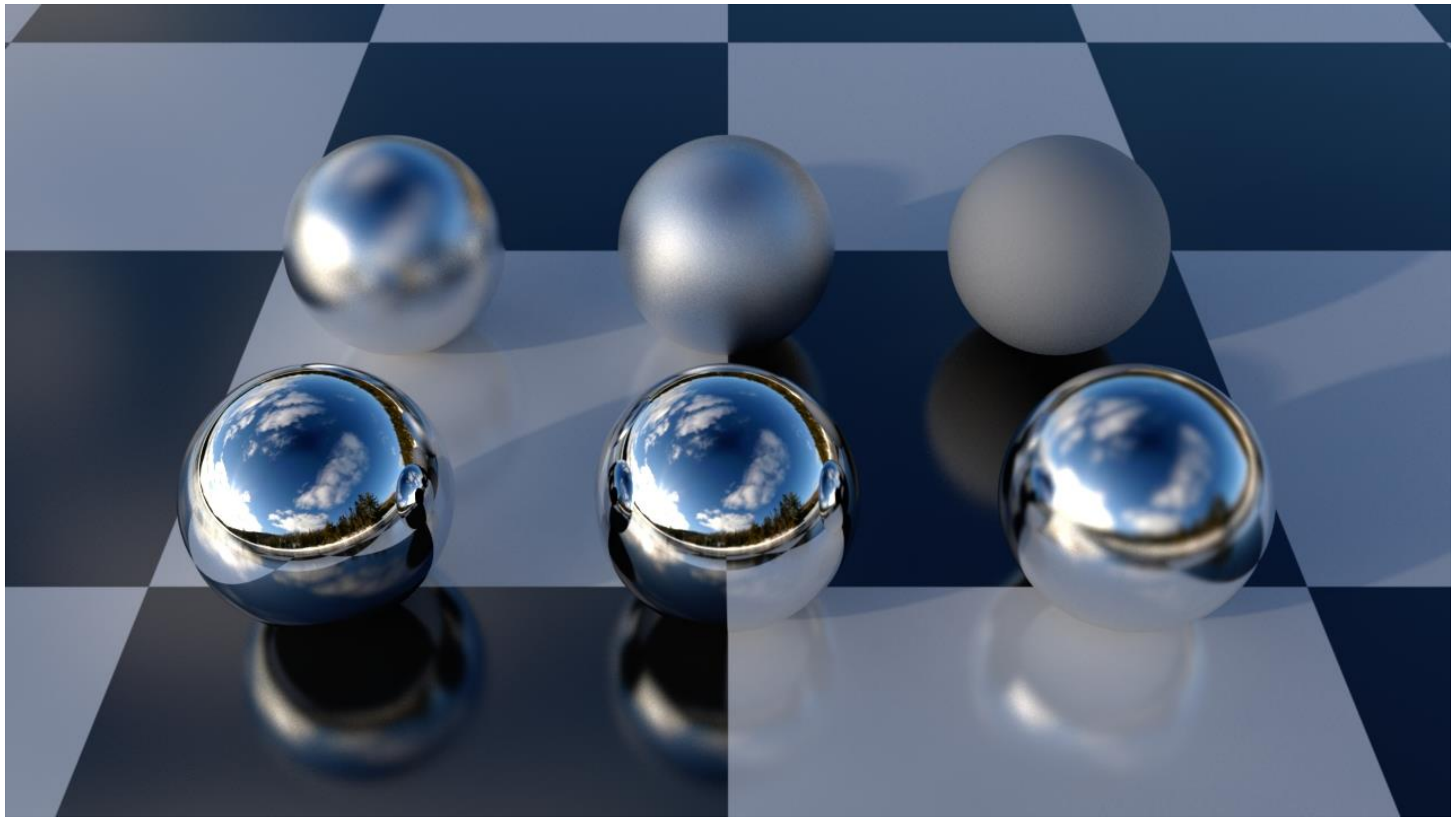
Reflections: ray tracing

Recursive ray tracing – more secondary rays



Reflections: ray tracing

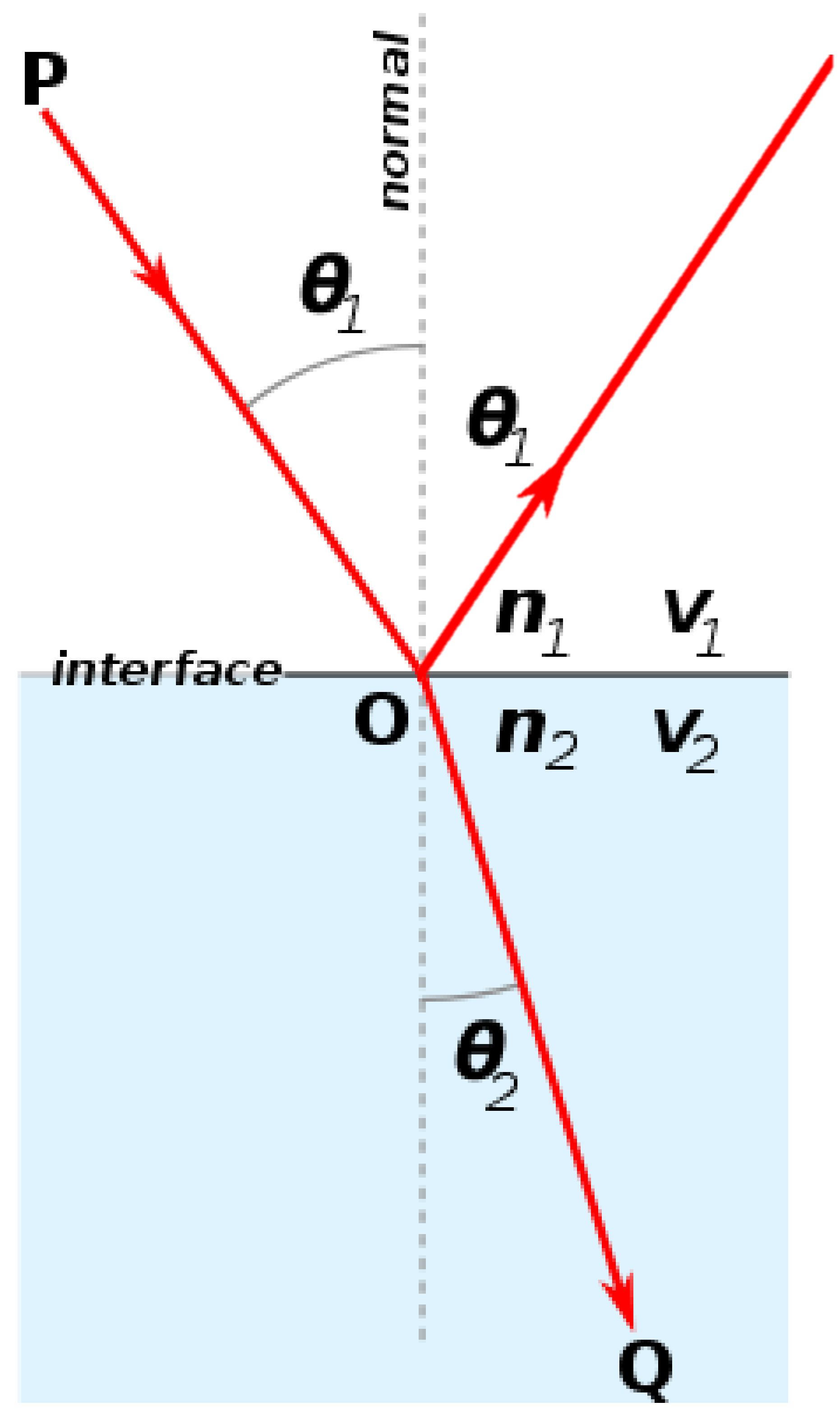




Q: How would you model appearance of a rougher glossy object?

Refractions

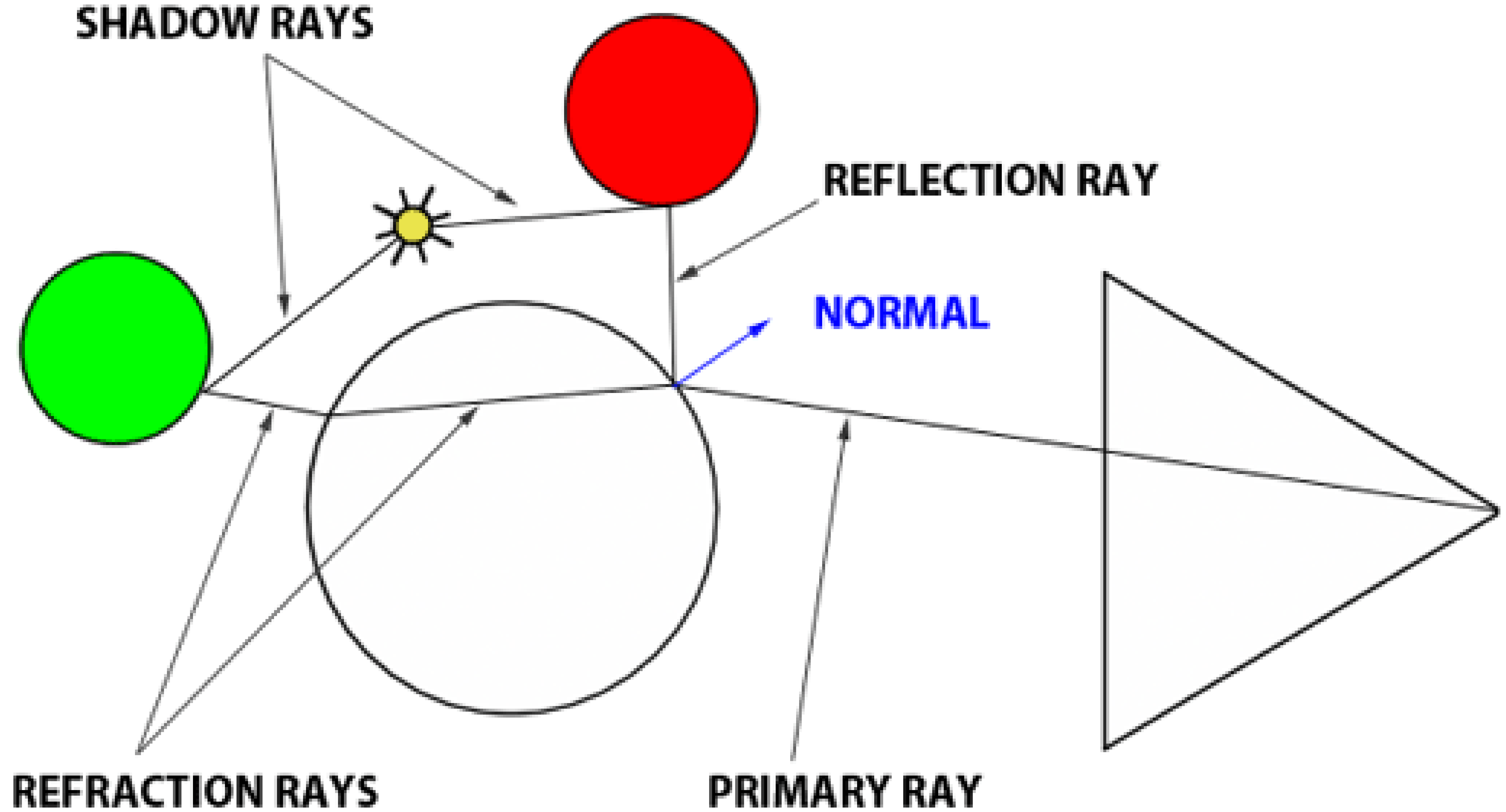
How do we refract rays?



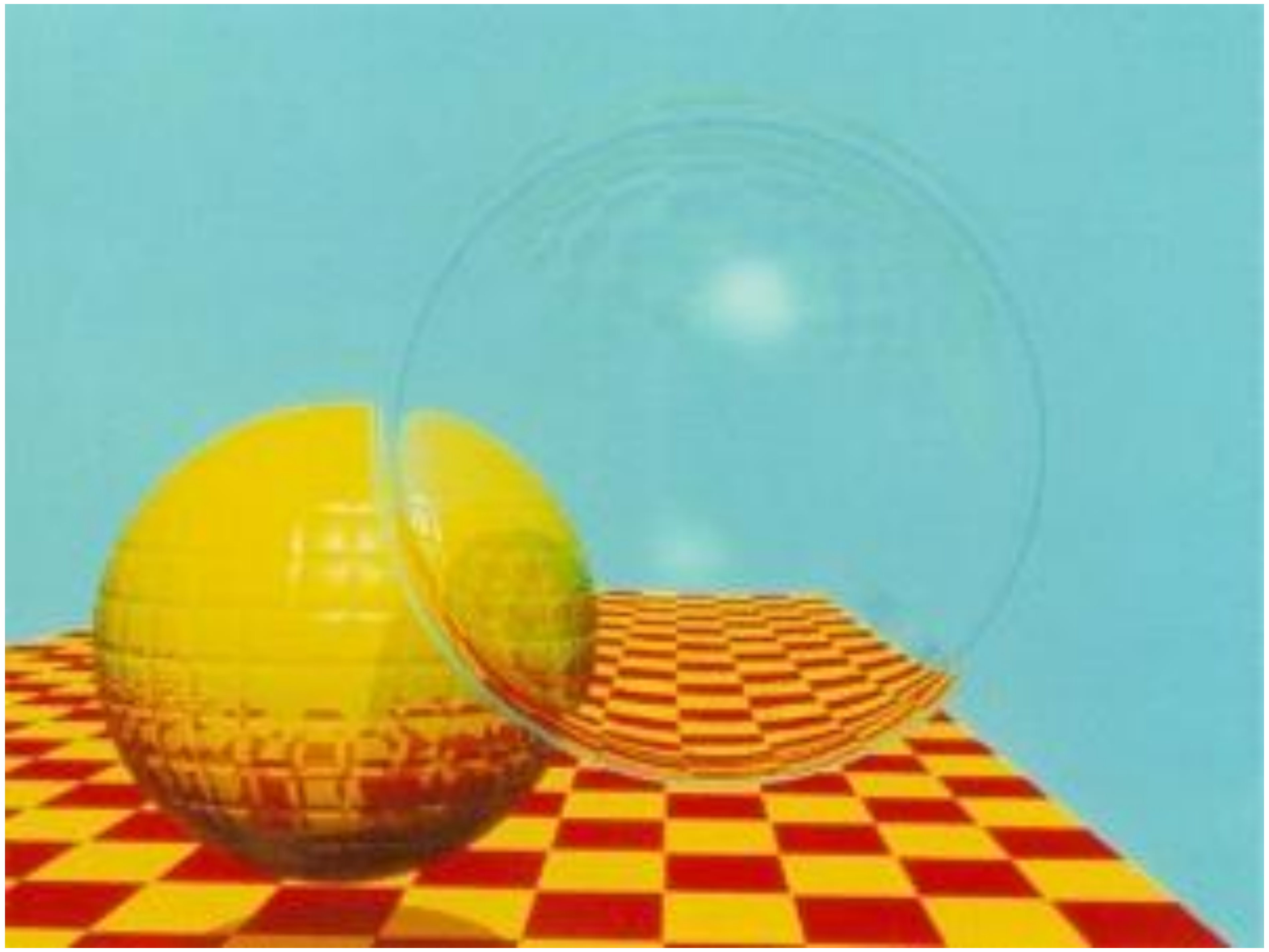
Refraction governed by Snell's law:

$$\frac{\sin\theta_2}{\sin\theta_1} = \frac{n_1}{n_2}$$

Shadows, Reflections, Refractions: recursive ray tracing

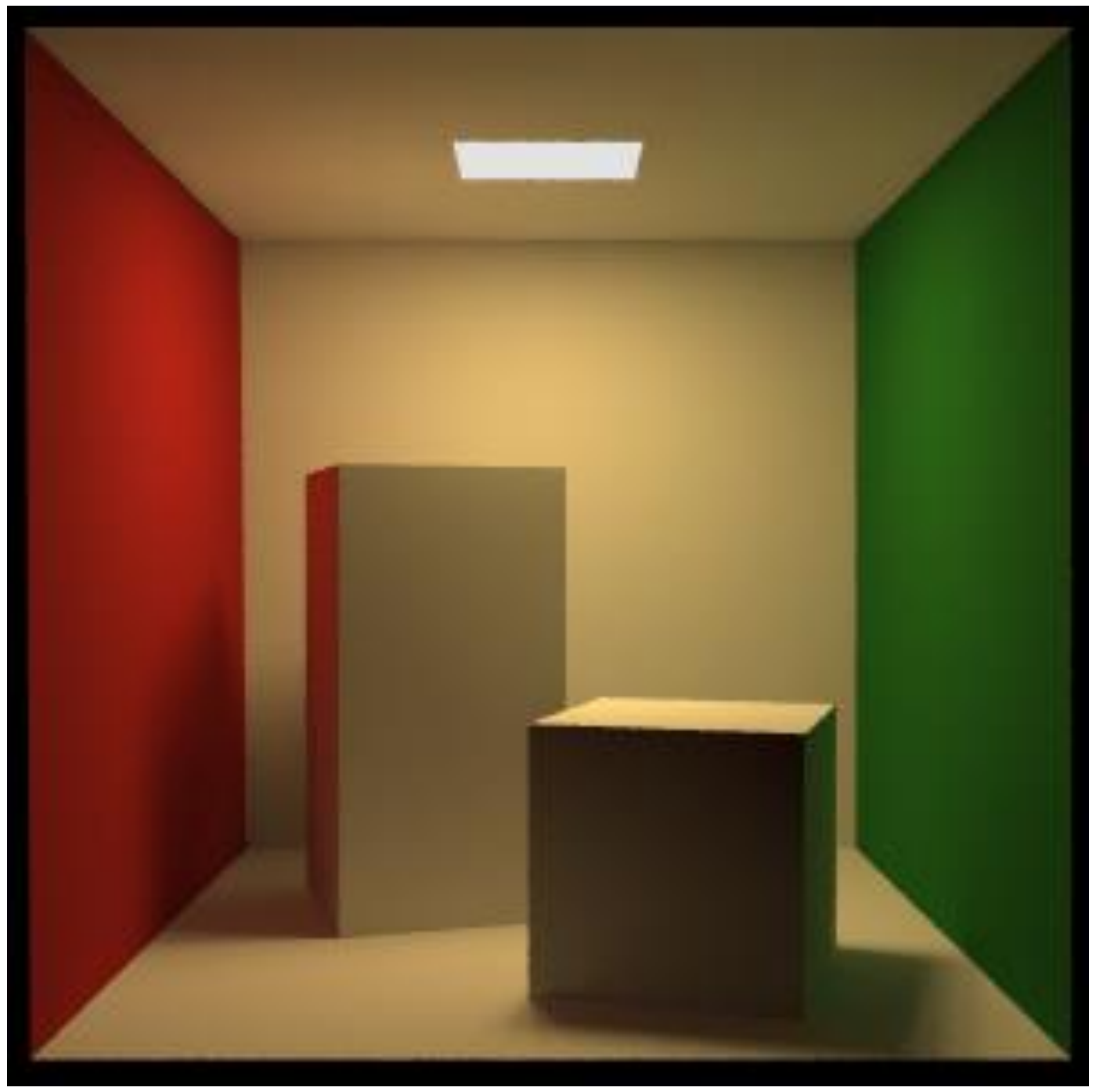


Ray tracing history



“An improved illumination model for shaded display” by T. Whitted, CACM 1980

The Cornell Box



And now: https://www.youtube.com/watch?v=tjf-1BxpR9c

It’s all about ray-scene intersections

Geometric Queries

- **Q: Given a point, in space (e.g., a new sample point), how do we find the closest point on a given surface?**
- **Q: Does implicit/explicit representation make this easier?**
- **Q: How do we find the distance to a single triangle? Or the point on the triangle that is hit by the ray? How about an entire 3D object? Or an entire virtual world?**
- **So many questions!**