Lecture 5:

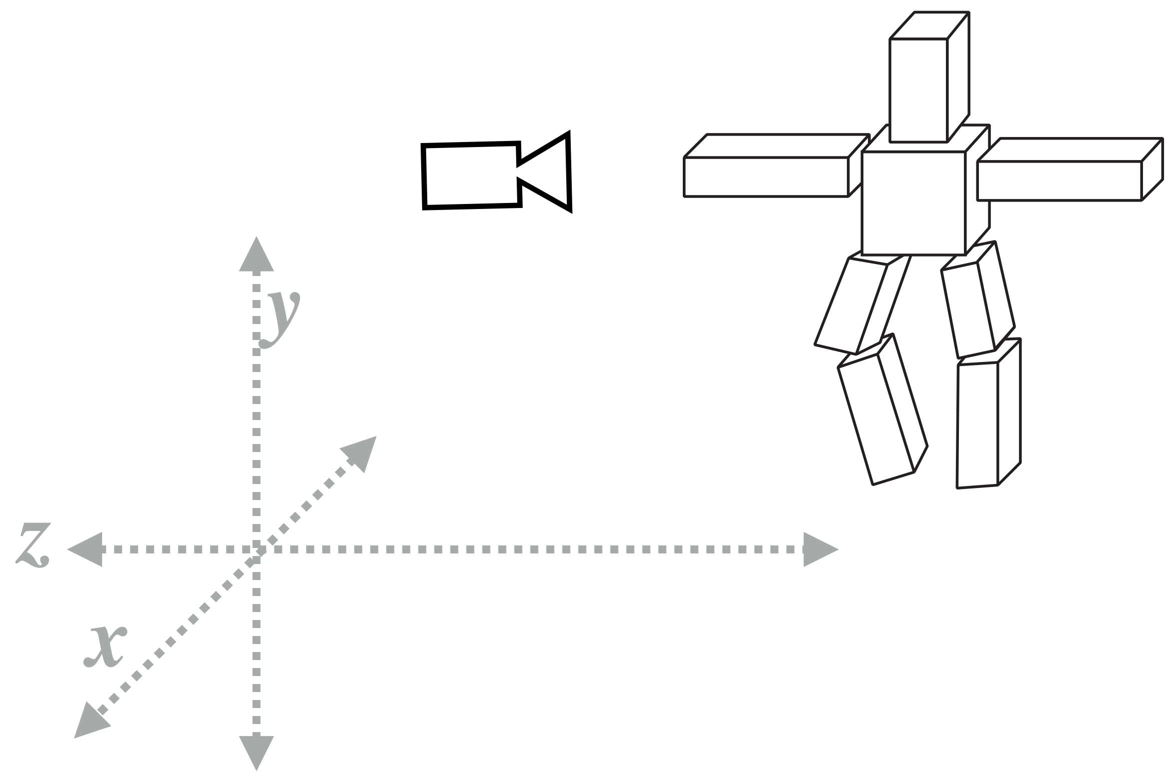
The Rasterization Pipeline

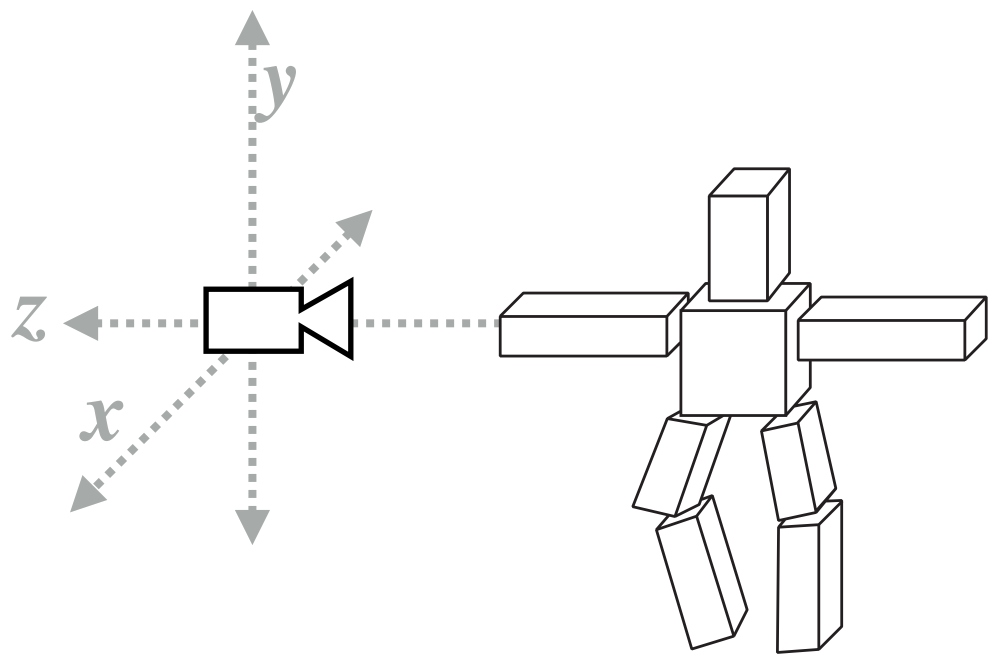
(and its implementation on GPUs)

Computer Graphics: Rendering, Geometry, and Image Manipulation
Stanford CS248A, Winter 2024

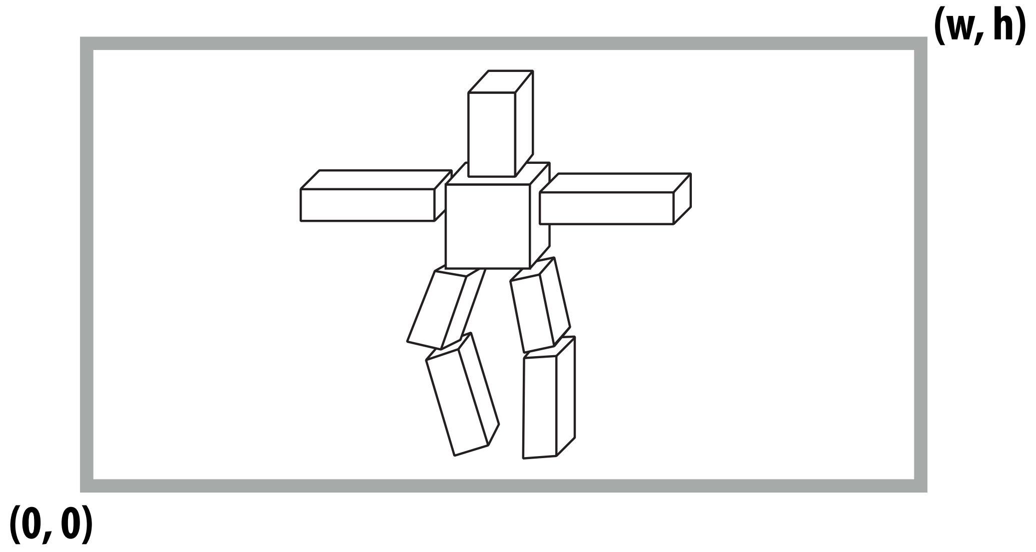
What you know how to do (at this point in the course)

Position objects and the camera in the world

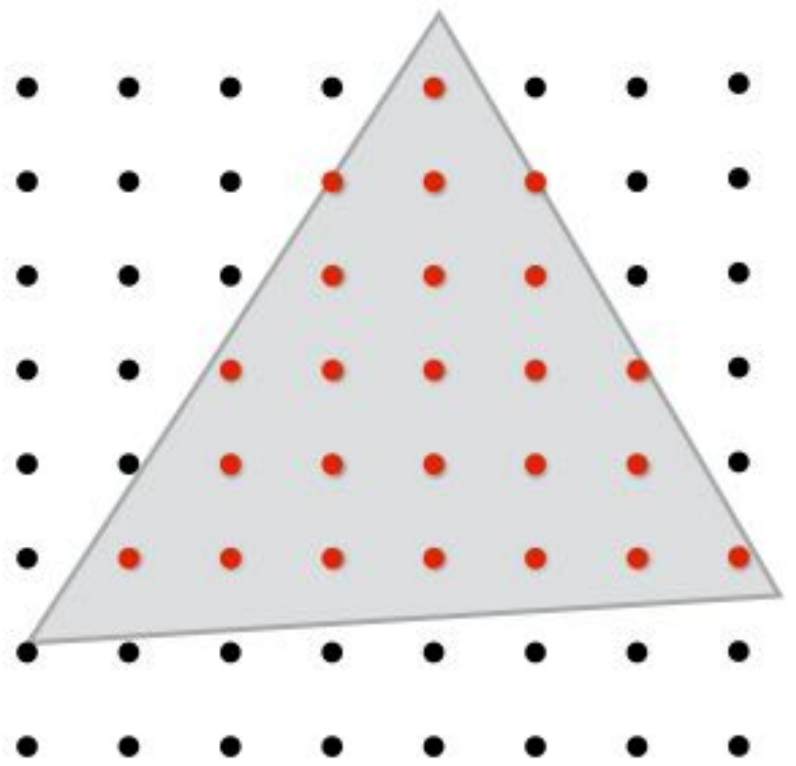




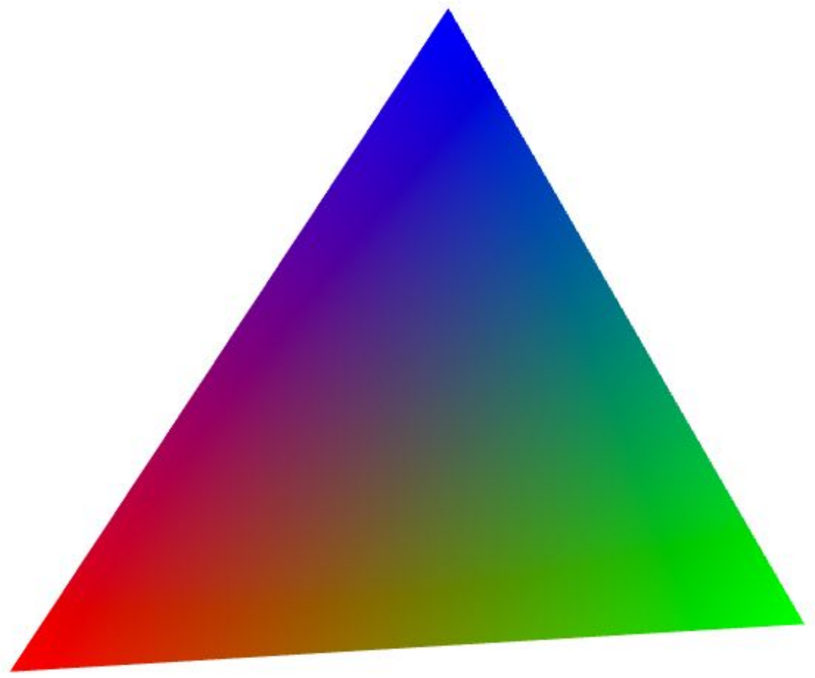
Determine the position of objects relative to the camera



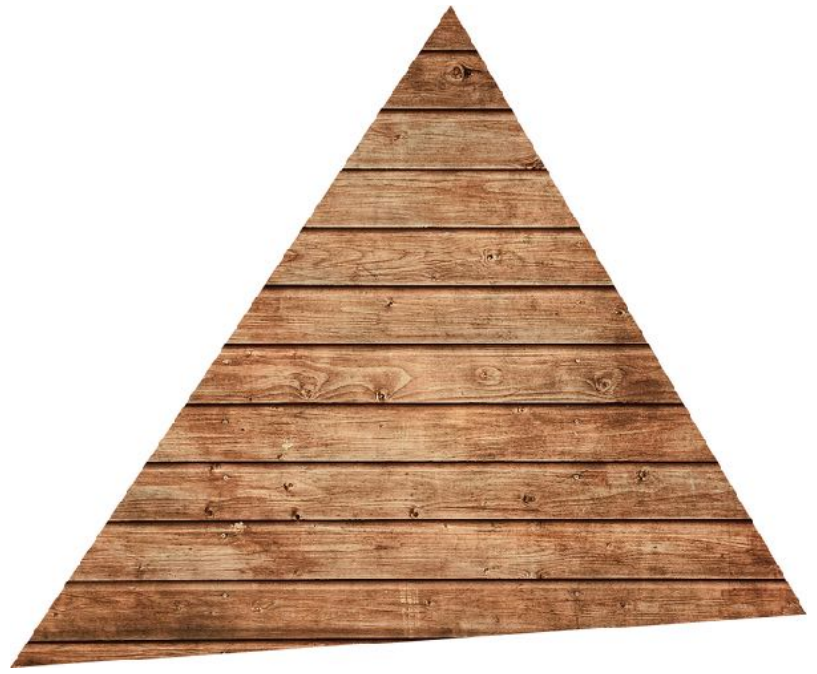
Project objects onto the screen



Sample triangle coverage



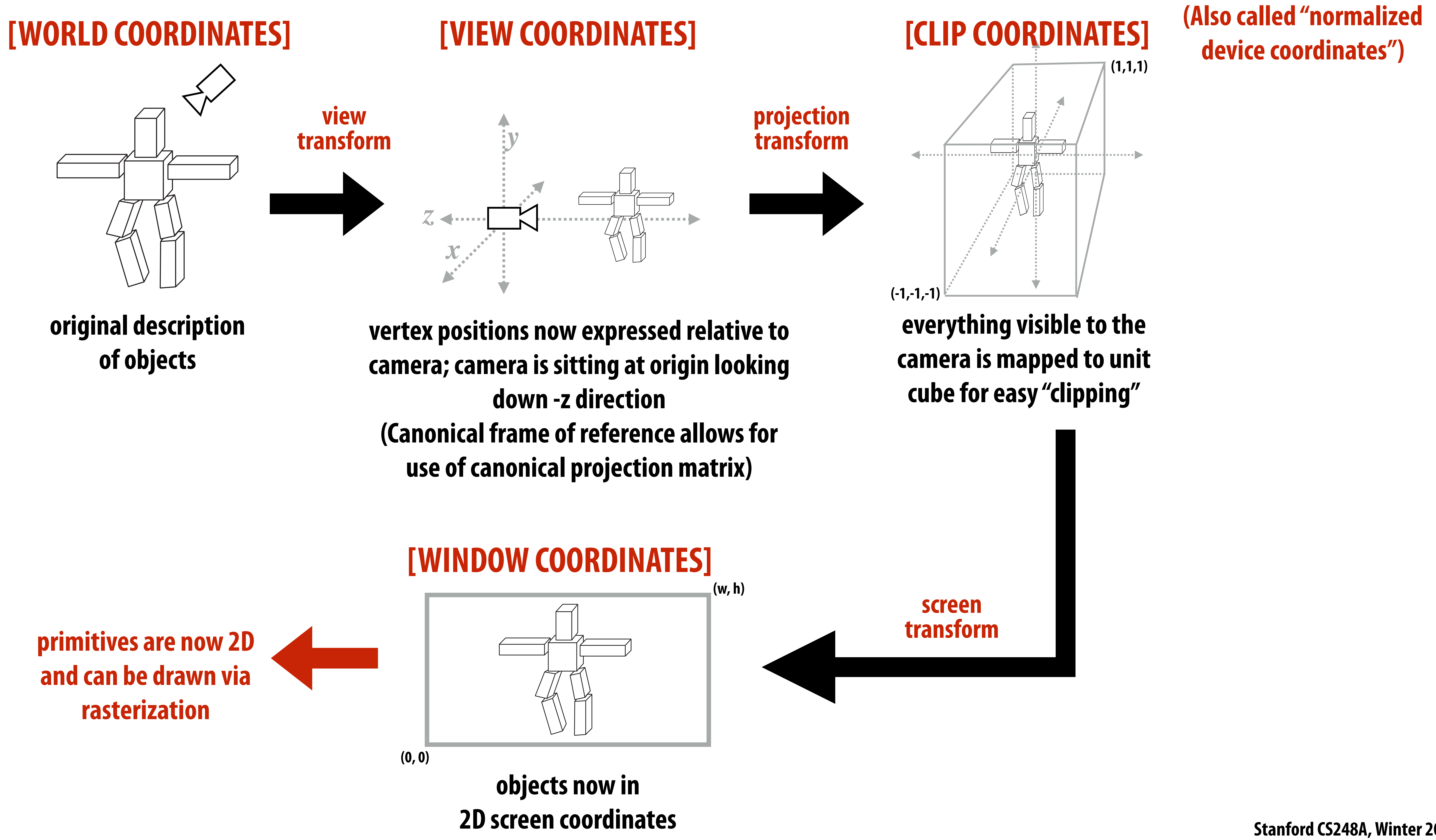
Compute triangle attribute values at covered sample points (Color, texture coords, depth)



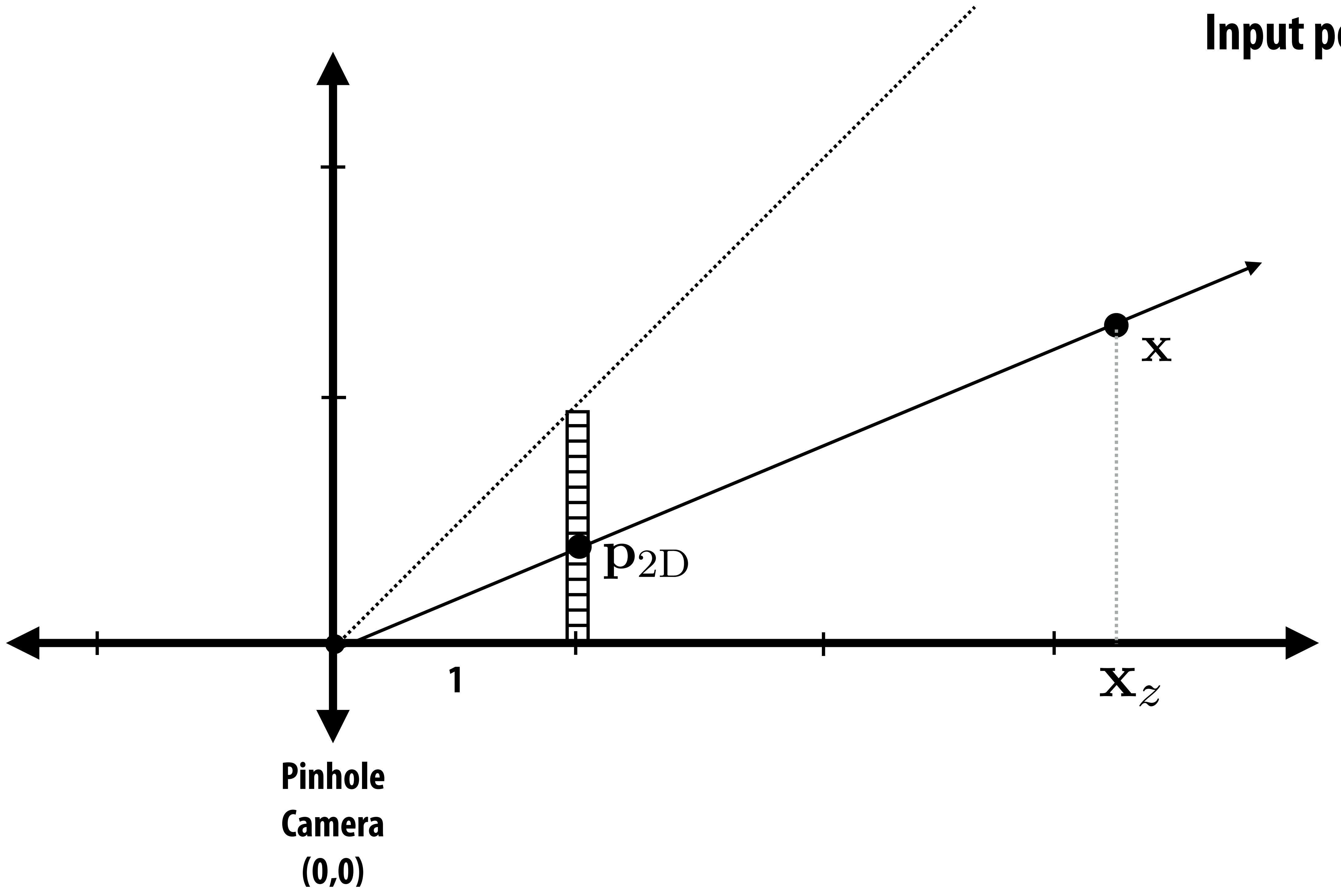
Sample texture maps

One more detail on perspective projection

Transformations: from objects to the screen



Basic perspective projection



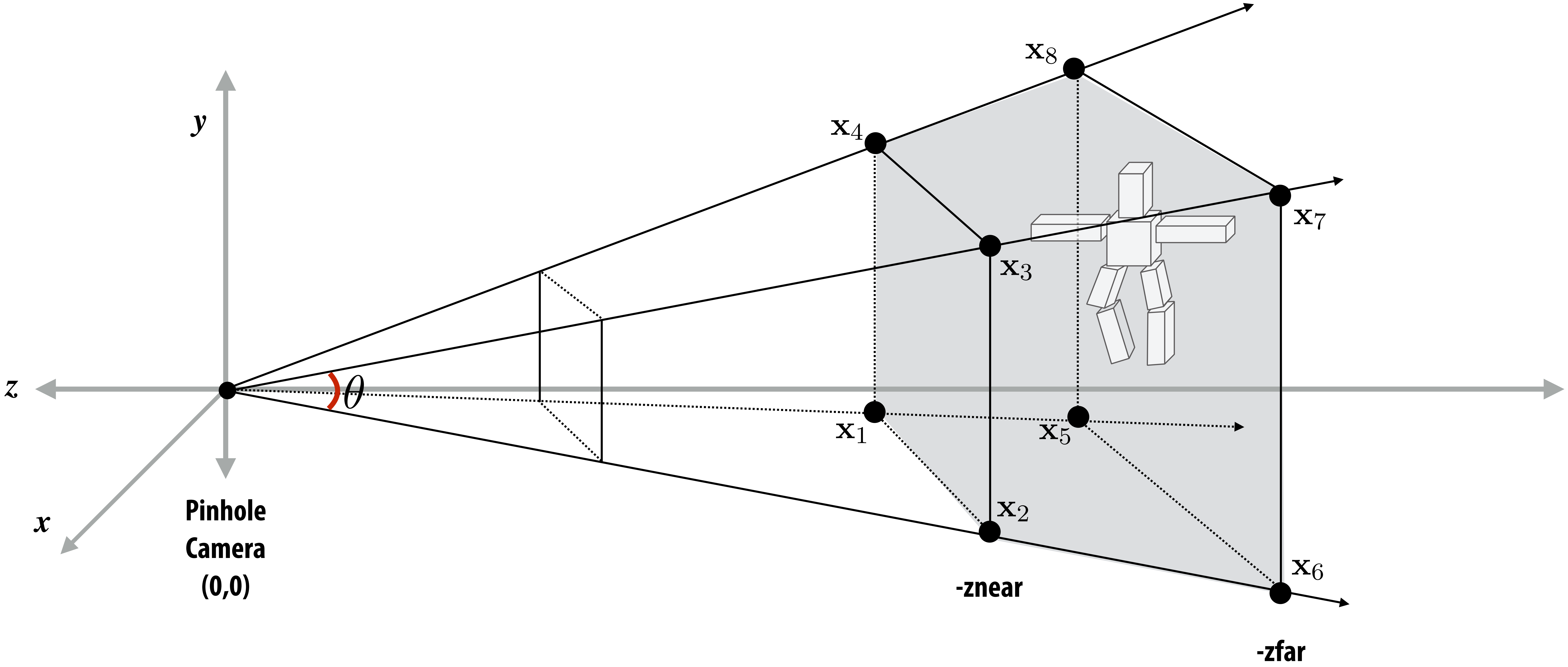
Input point in 3D-H: $\mathbf{x} = \begin{bmatrix} \mathbf{x}_x & \mathbf{x}_y & \mathbf{x}_z & 1 \end{bmatrix}^T$

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

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

View frustum

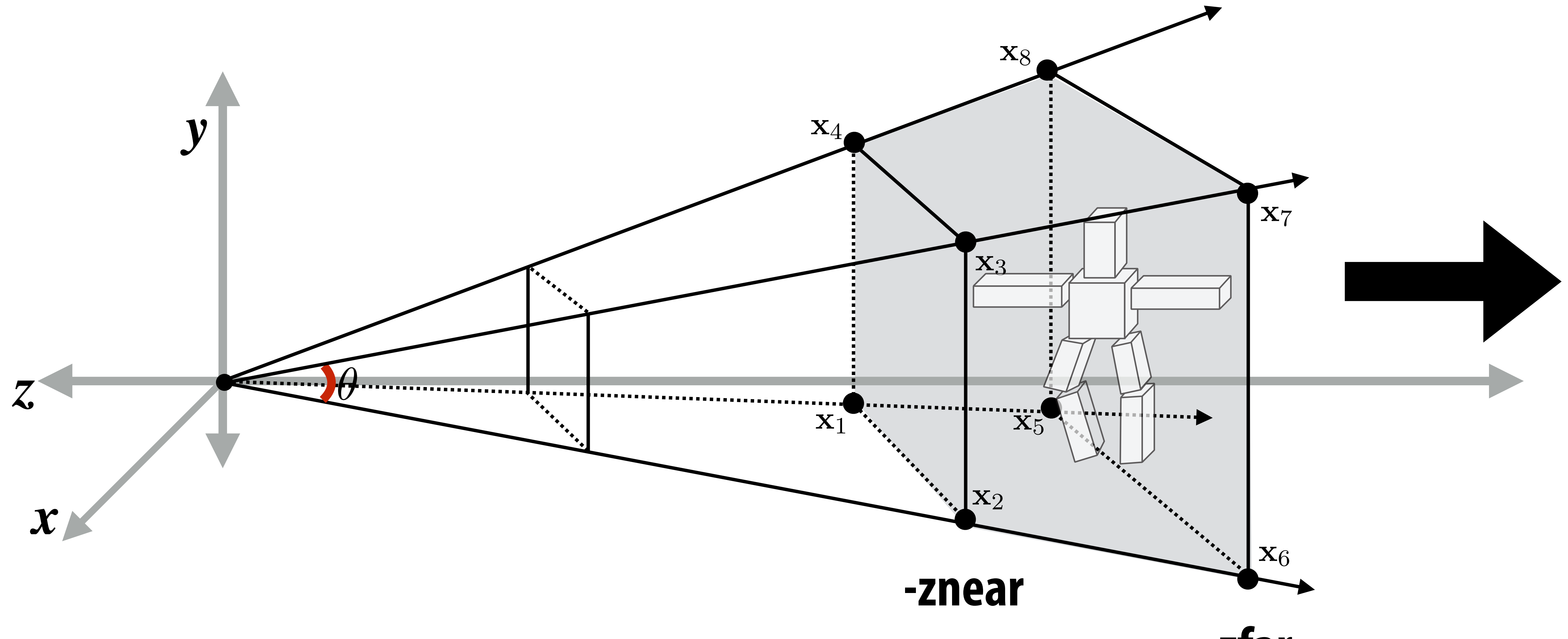
View frustum is the region of space the camera can see:

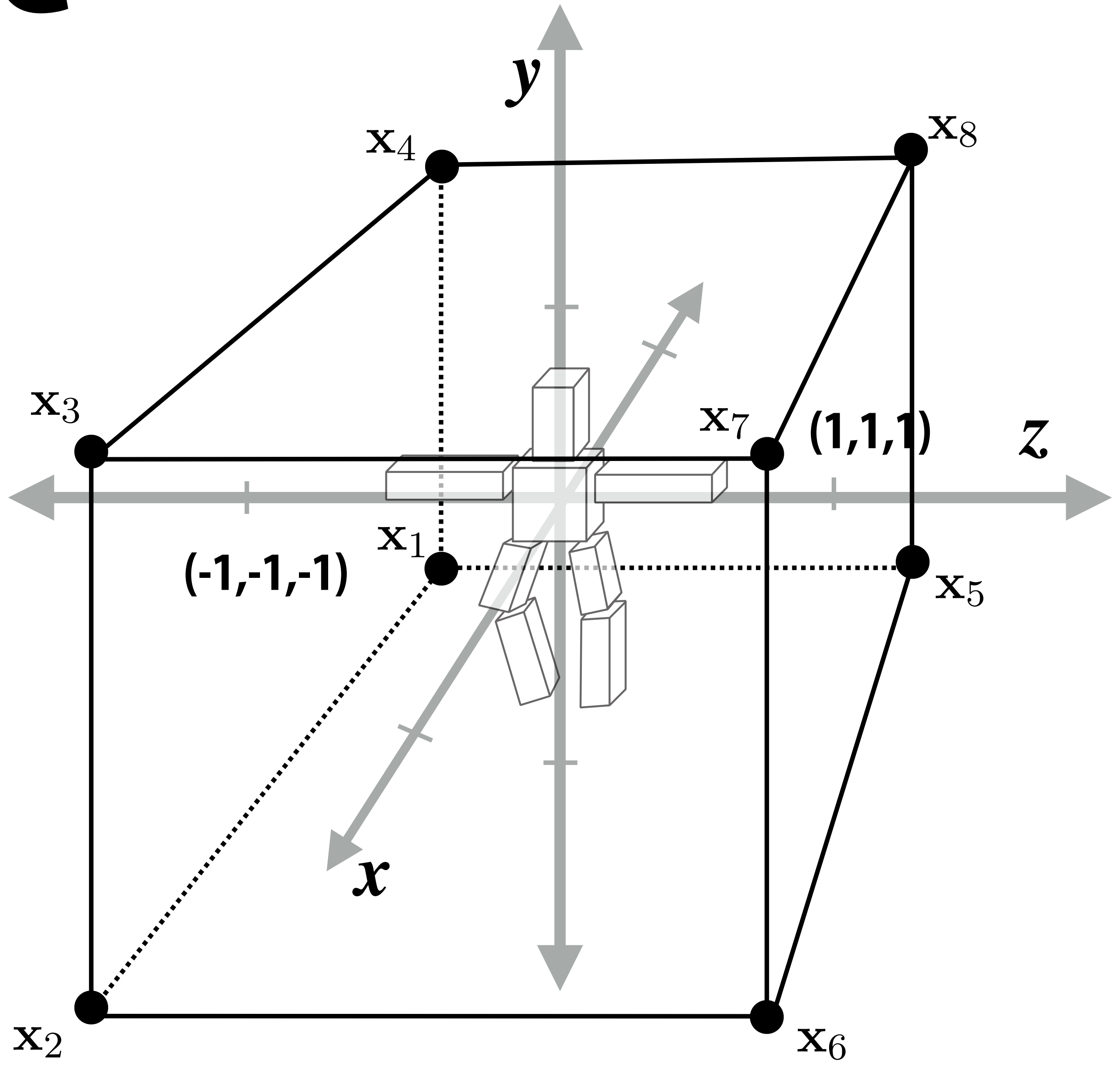


- **Top/bottom/left/right planes correspond to sides of screen**
- **Near/far planes correspond to closest/furthest thing we want to draw**

Mapping frustum to normalized cube

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





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

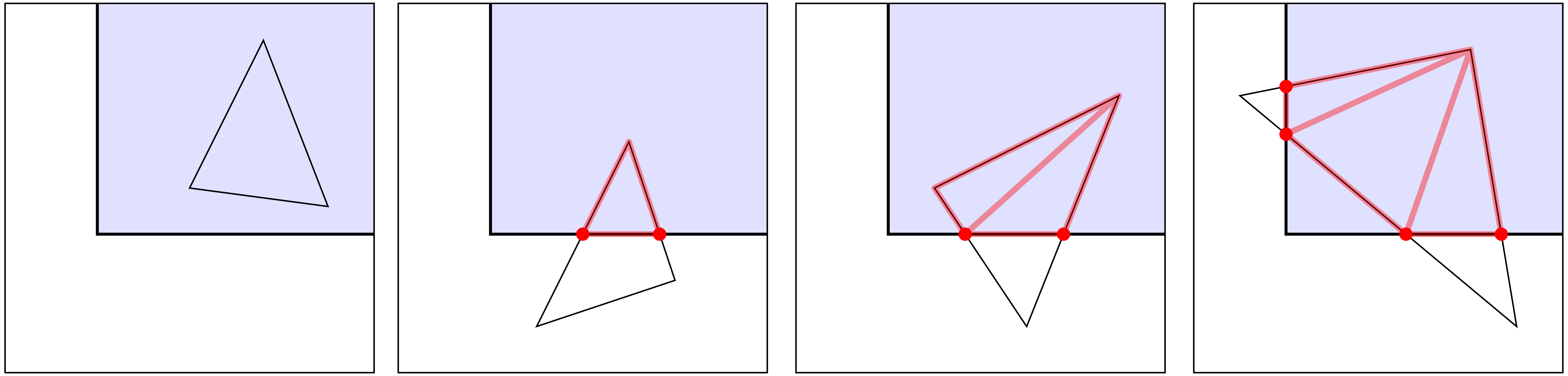
Why do we map frustum to unit cube?

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

*** Question: what does the frustum of an orthographic camera look like?**

Clipping

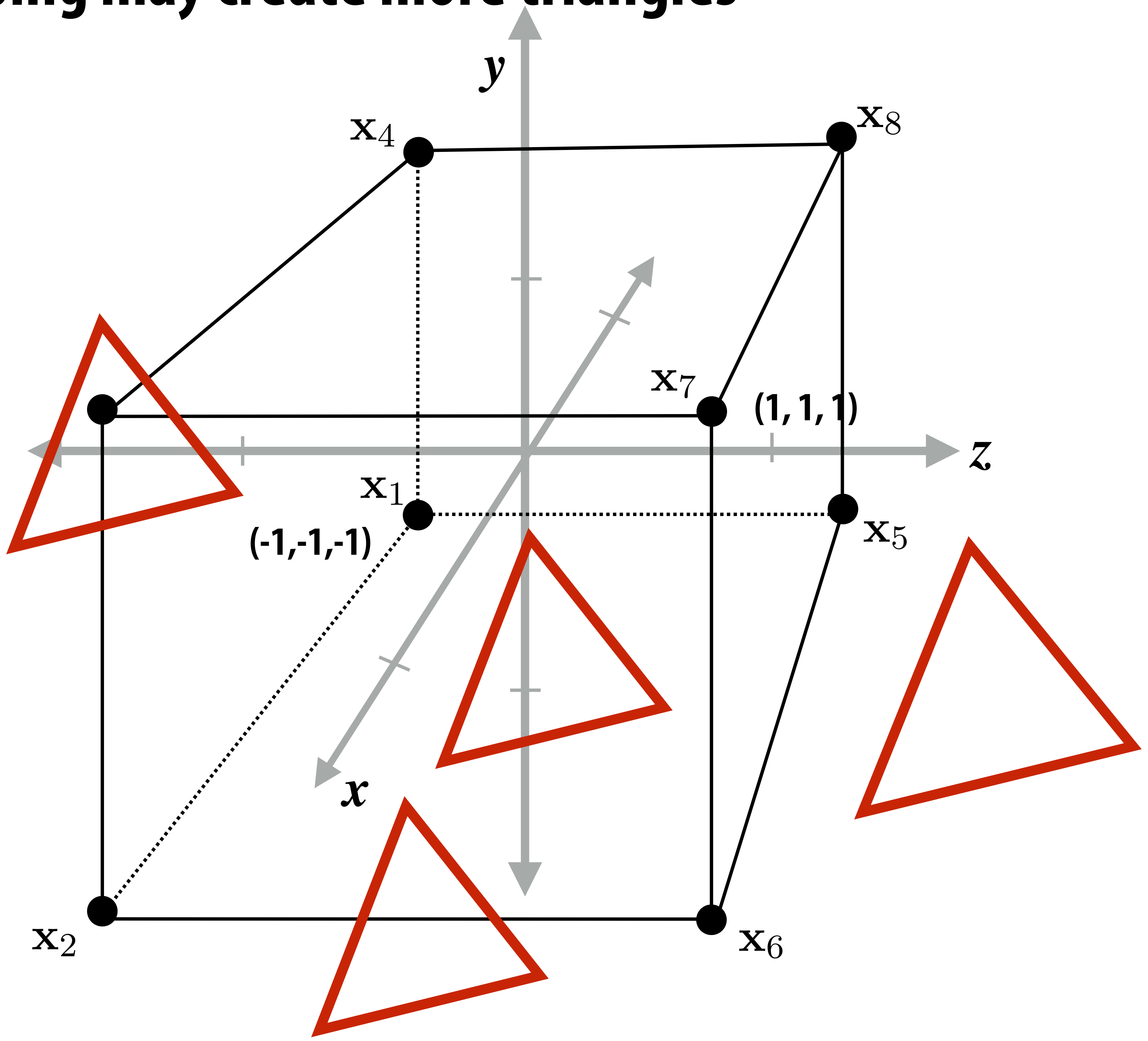
- "Clipping" is the process of eliminating triangles that aren't visible from the camera (they outside the view frustum)
  - Don't waste time computing the appearance of primitives the camera can't see!
  - Sample-in-triangle tests are expensive ("fine granularity" visibility)
  - Makes more sense to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



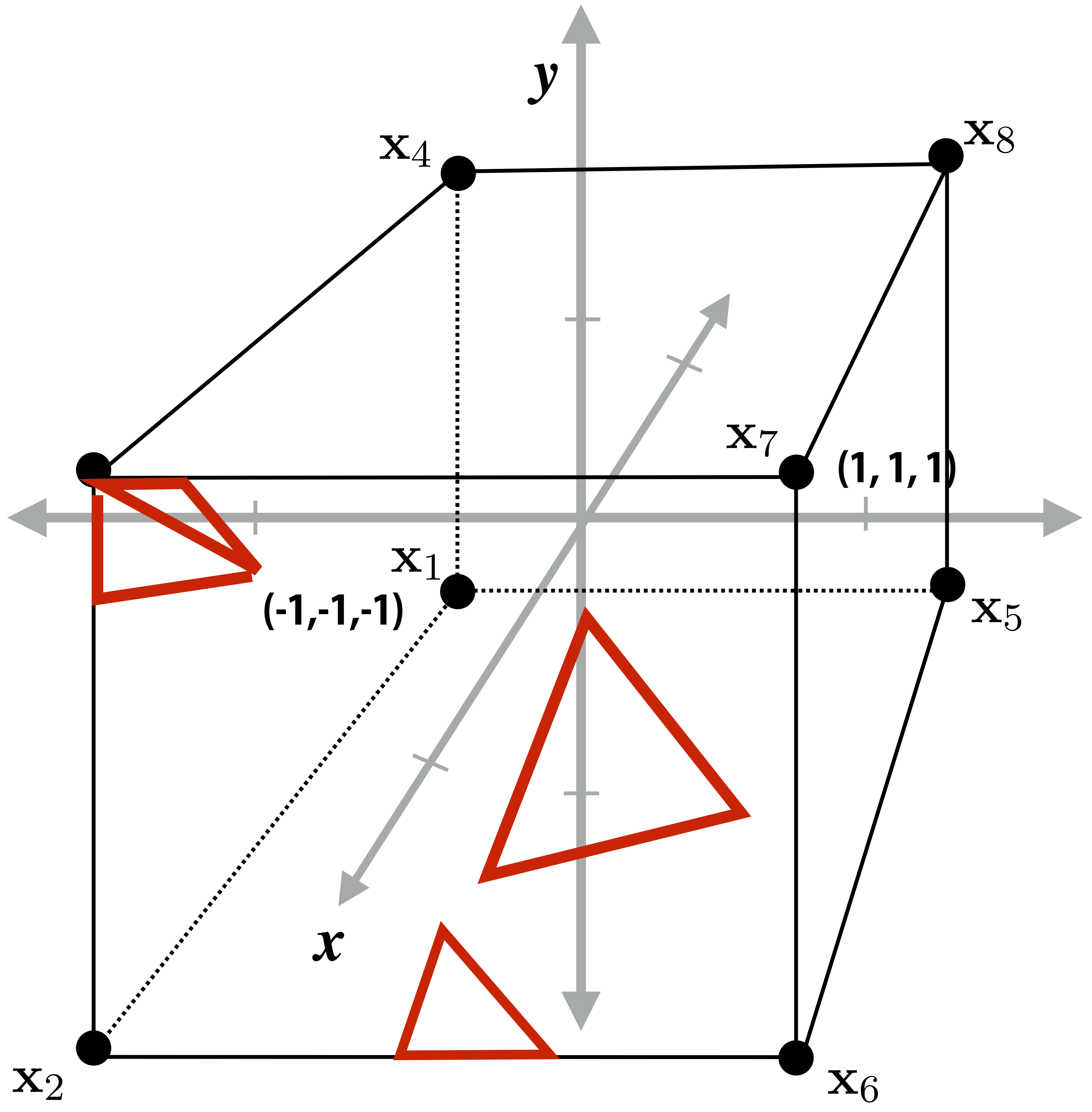
from: https://paroj.github.io/gltut/

Clipping in normalized device coordinates (NDC)

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



Triangles before clipping

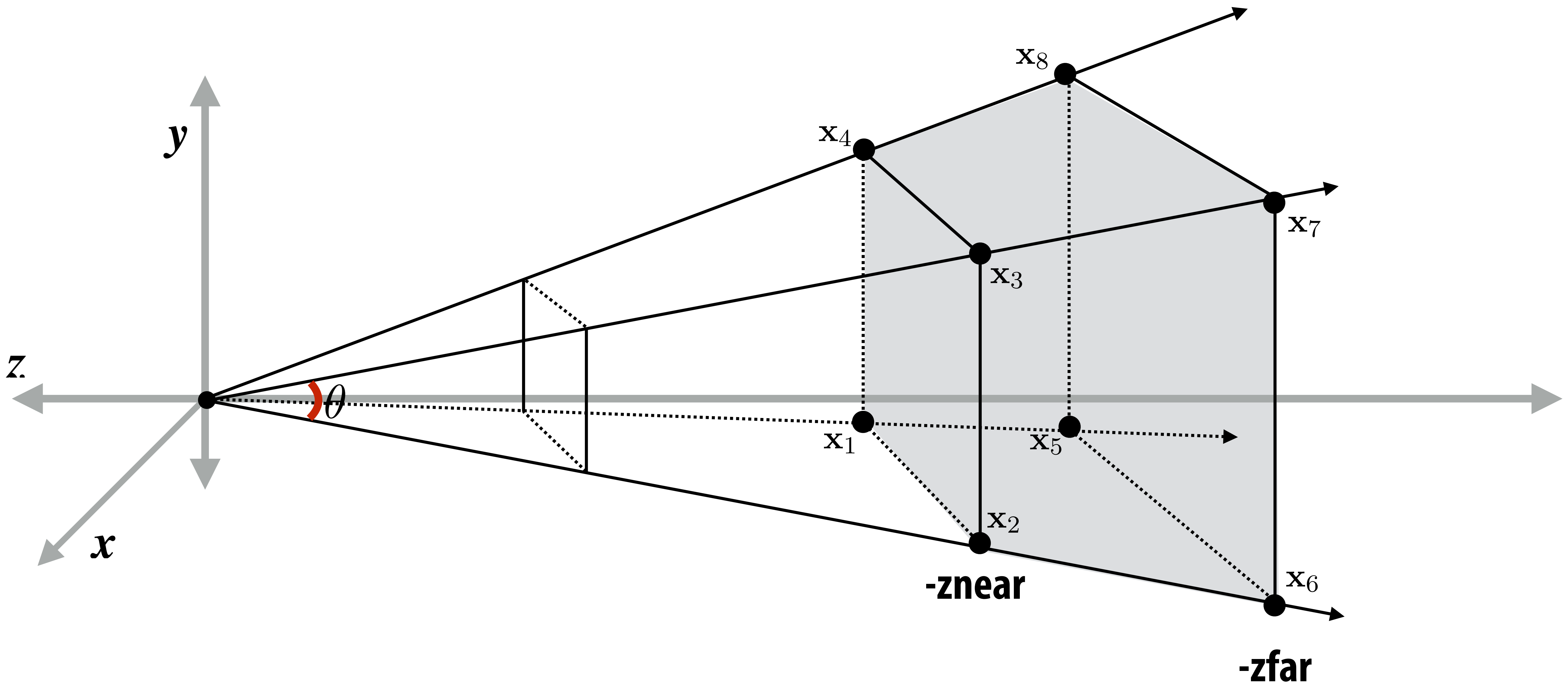


Triangles after clipping

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

Matrix for perspective transform

Takes into account geometry of view frustum:



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

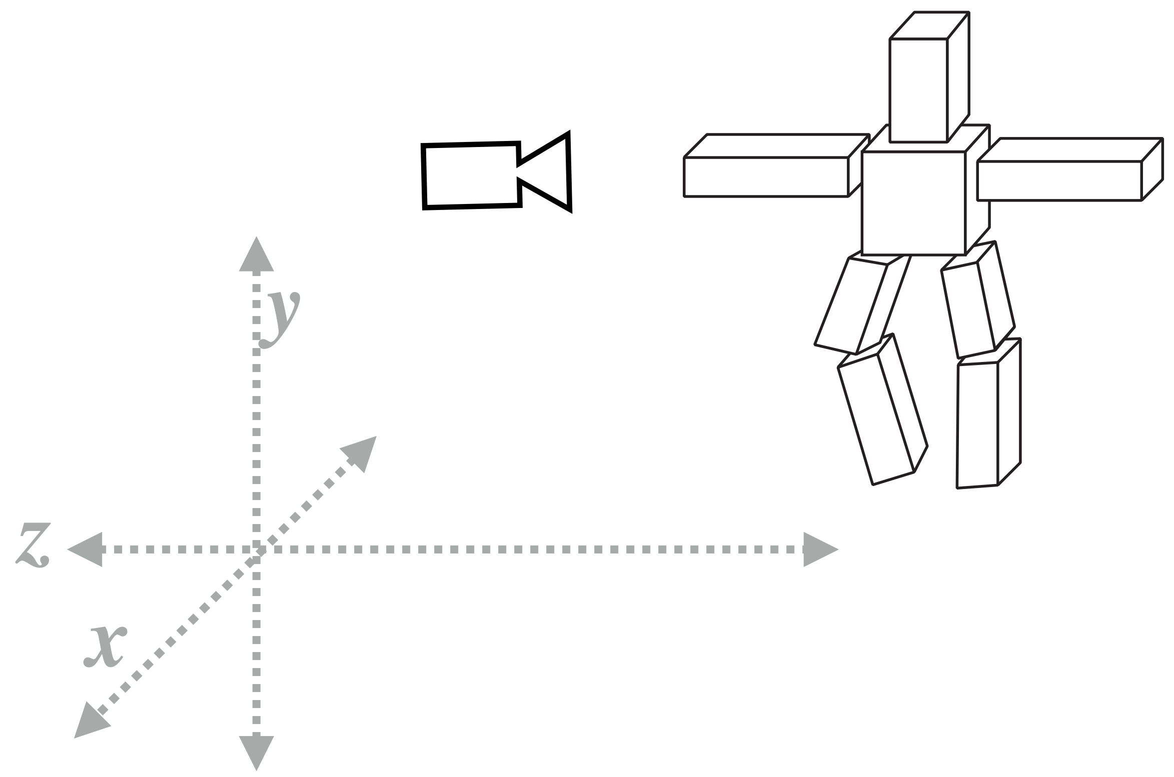
left (l), right (r), top (t), bottom (b), near (n), far (f)

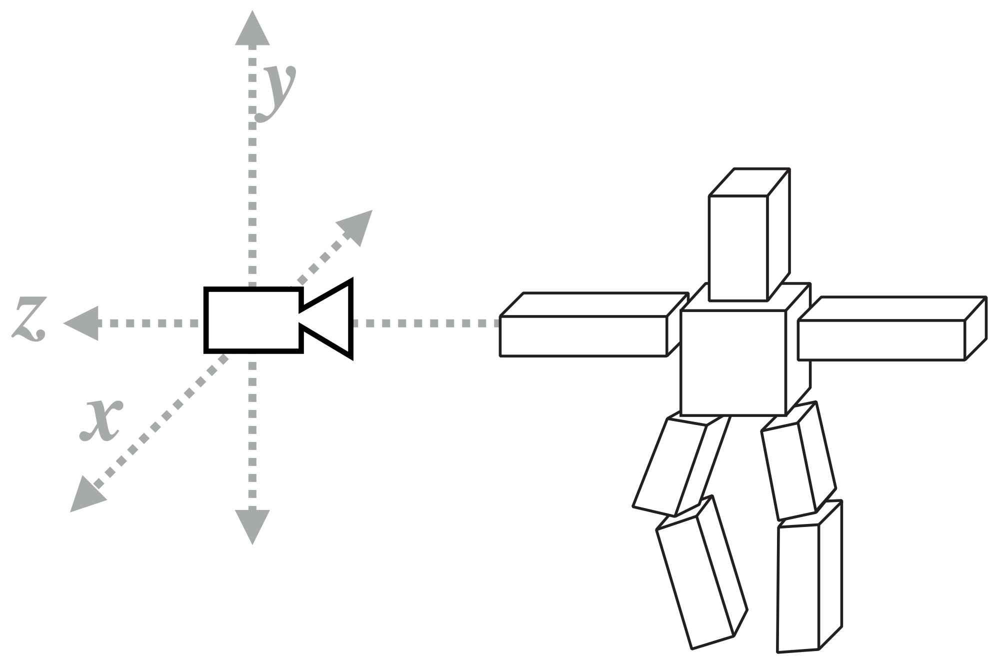
(matrix at left is perspective projection for frustum that is symmetric about x,y axes: l=-r, t=-b)

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

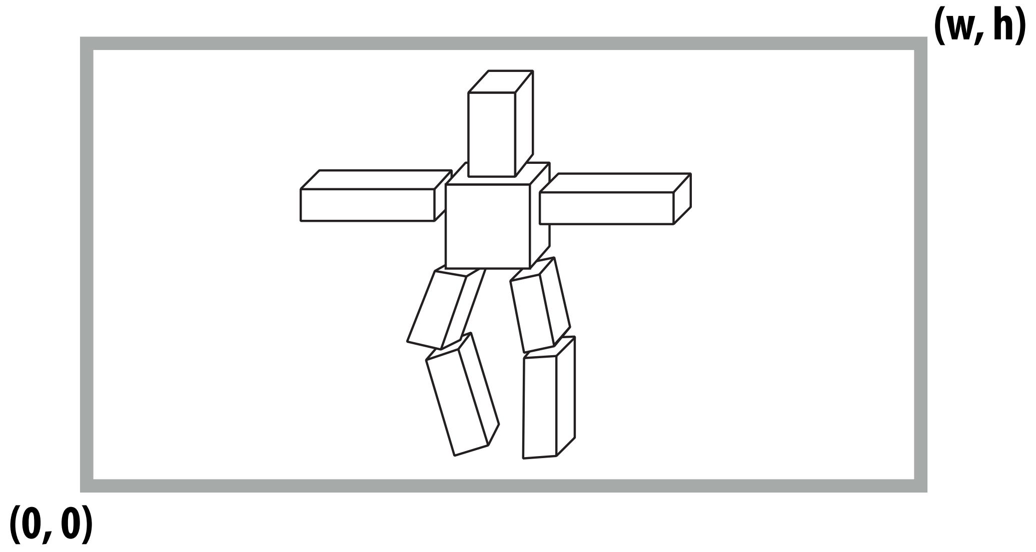
What you know how to do (at this point in the course)

Position objects and the camera in the world

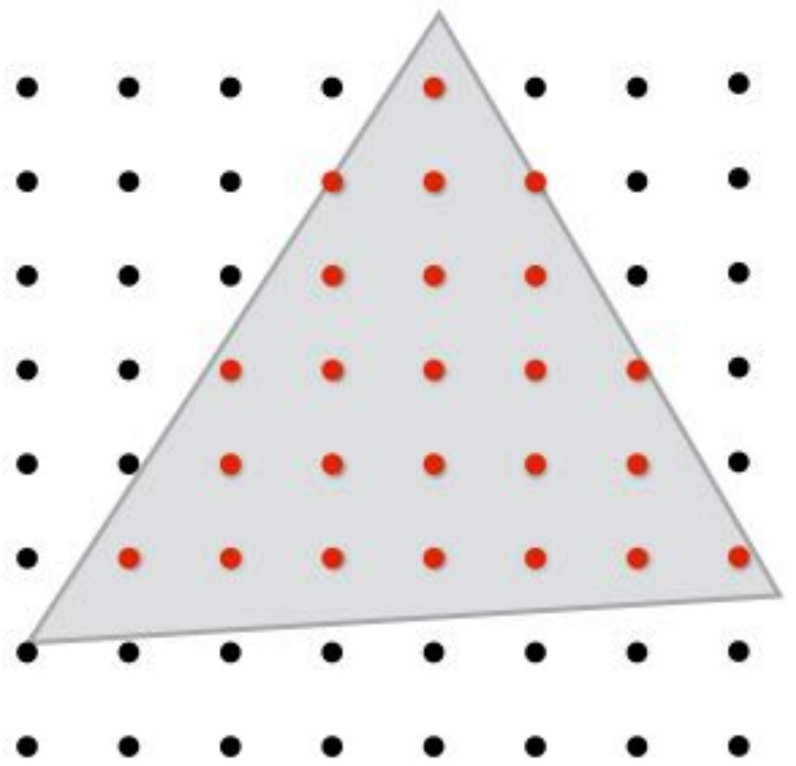




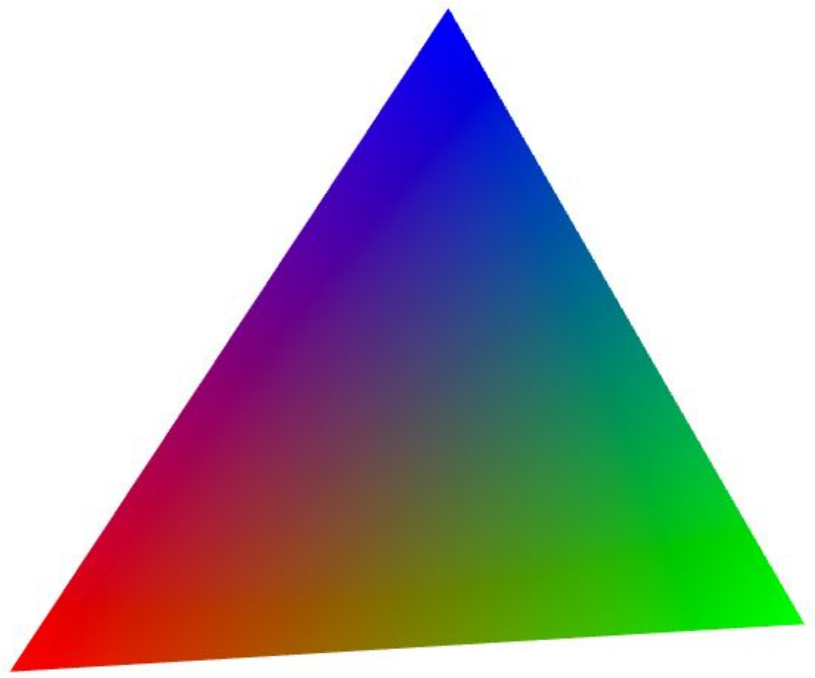
Determine the position of objects relative to the camera



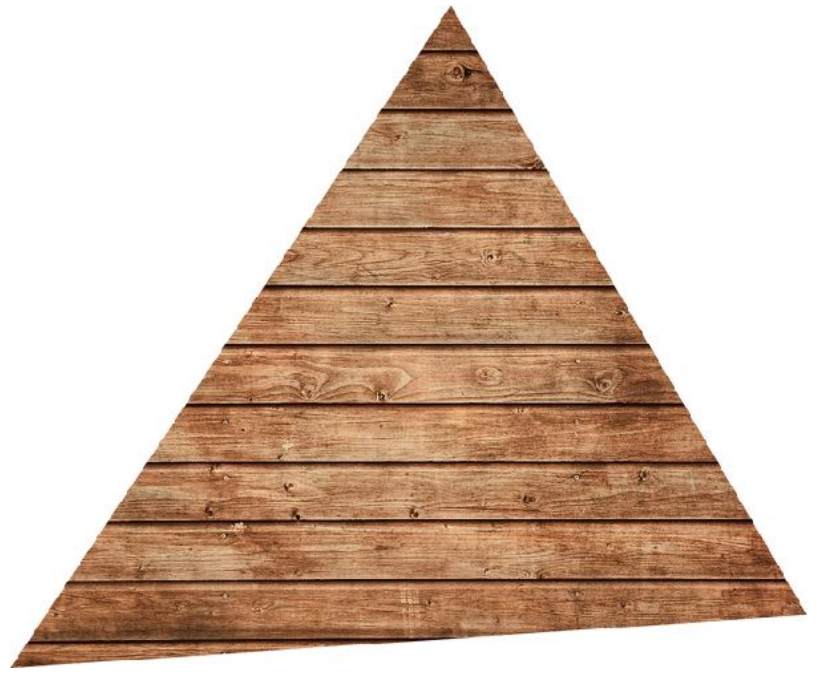
Project objects onto the screen



Sample triangle coverage



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



Sample texture maps

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

Key concepts:

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

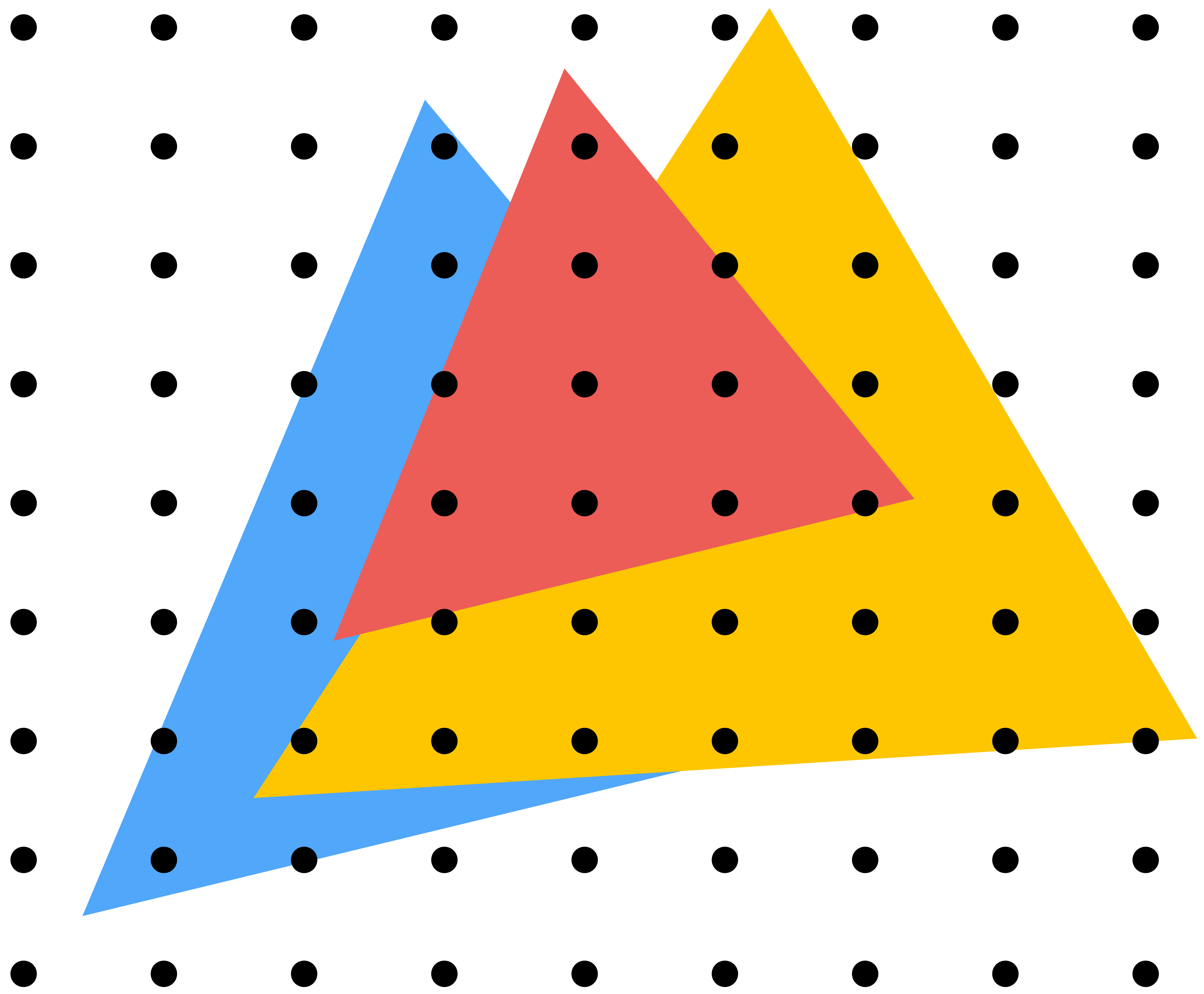
- Introduction
- Drawing a triangle (by sampling)
- Transforms and coordinate spaces
- Perspective projection and texture sampling
- Today: putting it all together: end-to-end triangle rasterization pipeline

Geometry Processing

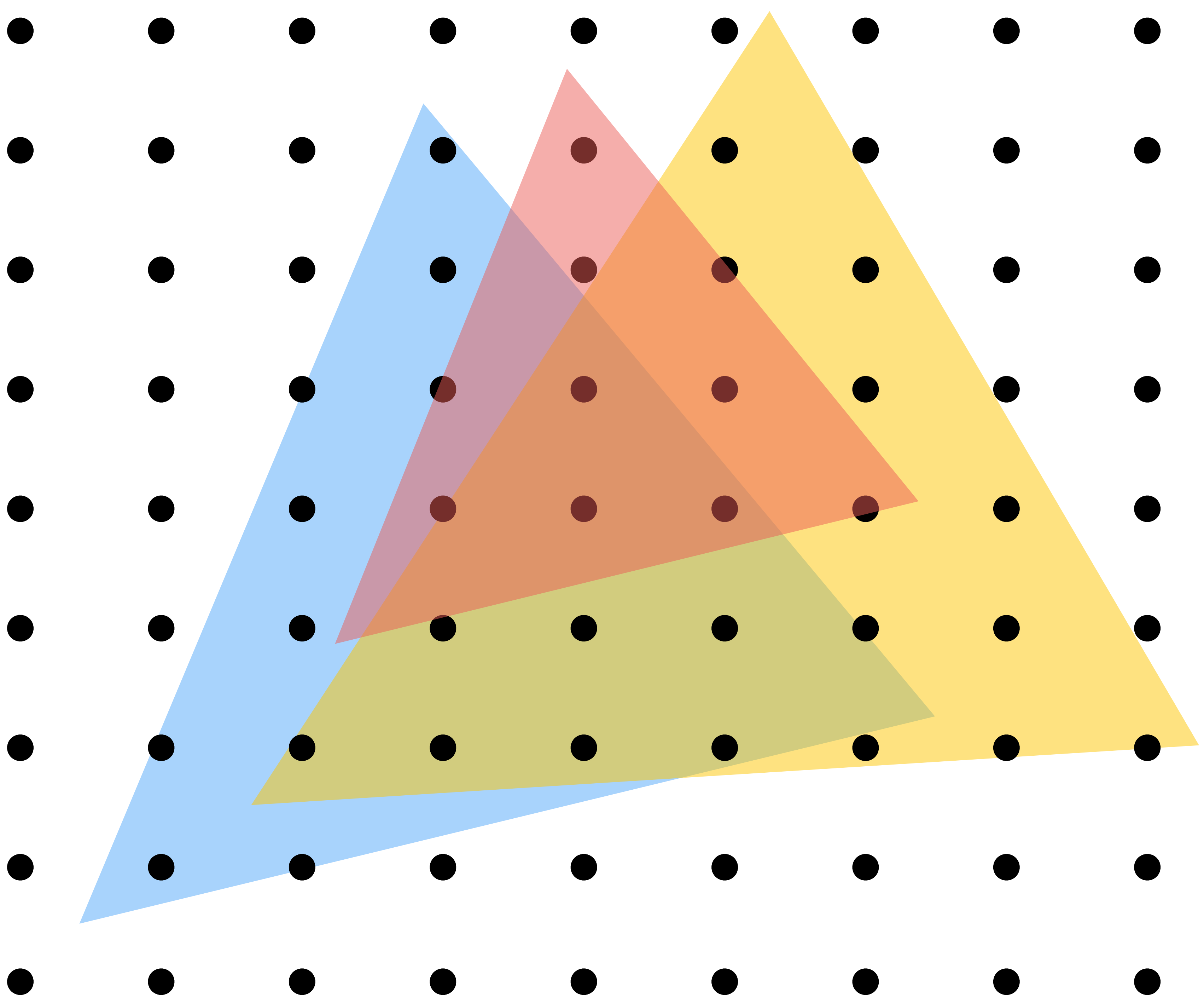
Materials and Lighting

Occlusion using the Depth Buffer

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



Opaque Triangles



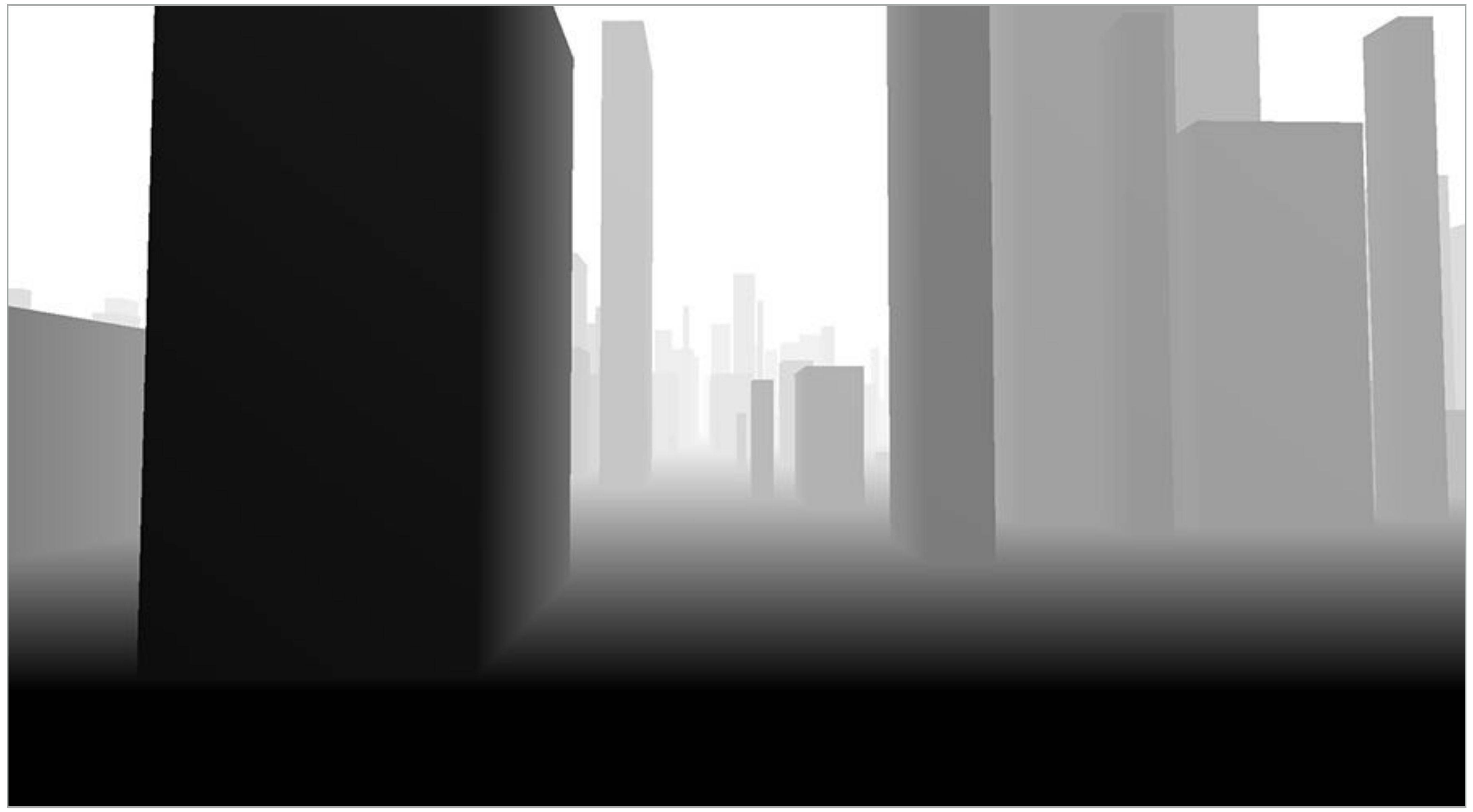
50% transparent triangles

Depth buffer (aka "Z buffer")

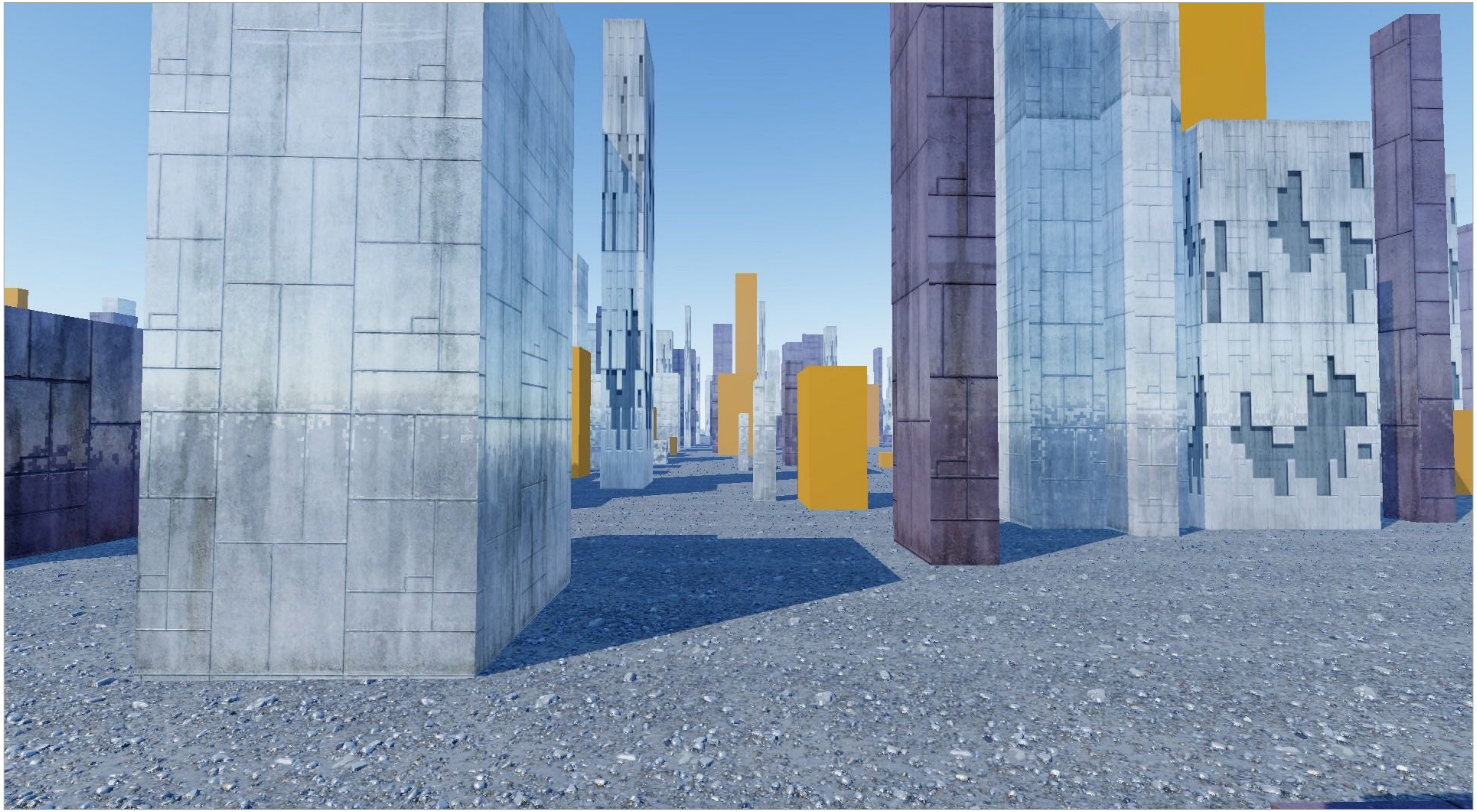
Color buffer:
(stores color per sample… e.g., RGB)

Depth buffer:
(stores depth per sample)

Stores depth of closest surface drawn so far
black = close depth
white = far depth



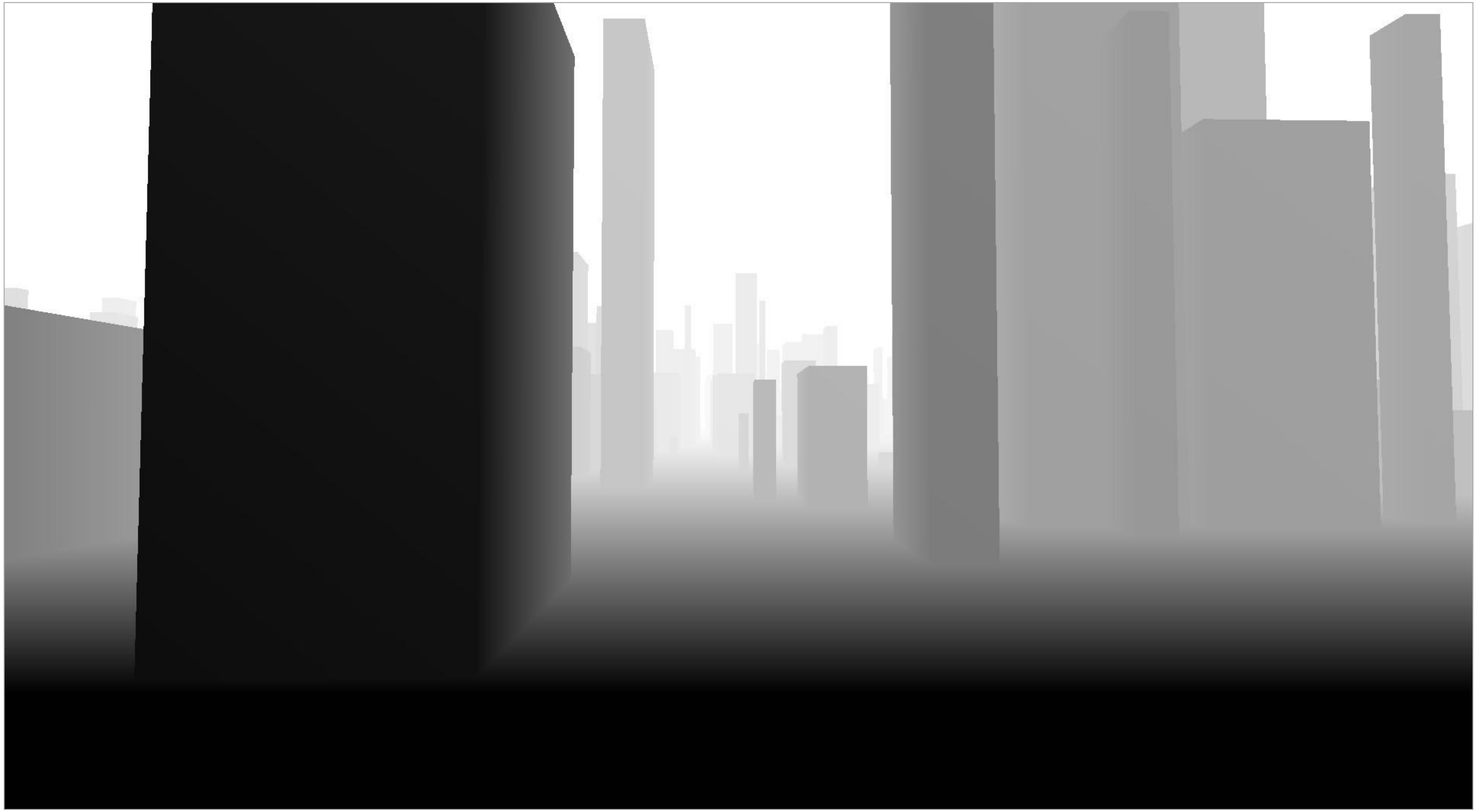
Depth buffer (a better look)



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

Depth buffer (a better look)

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



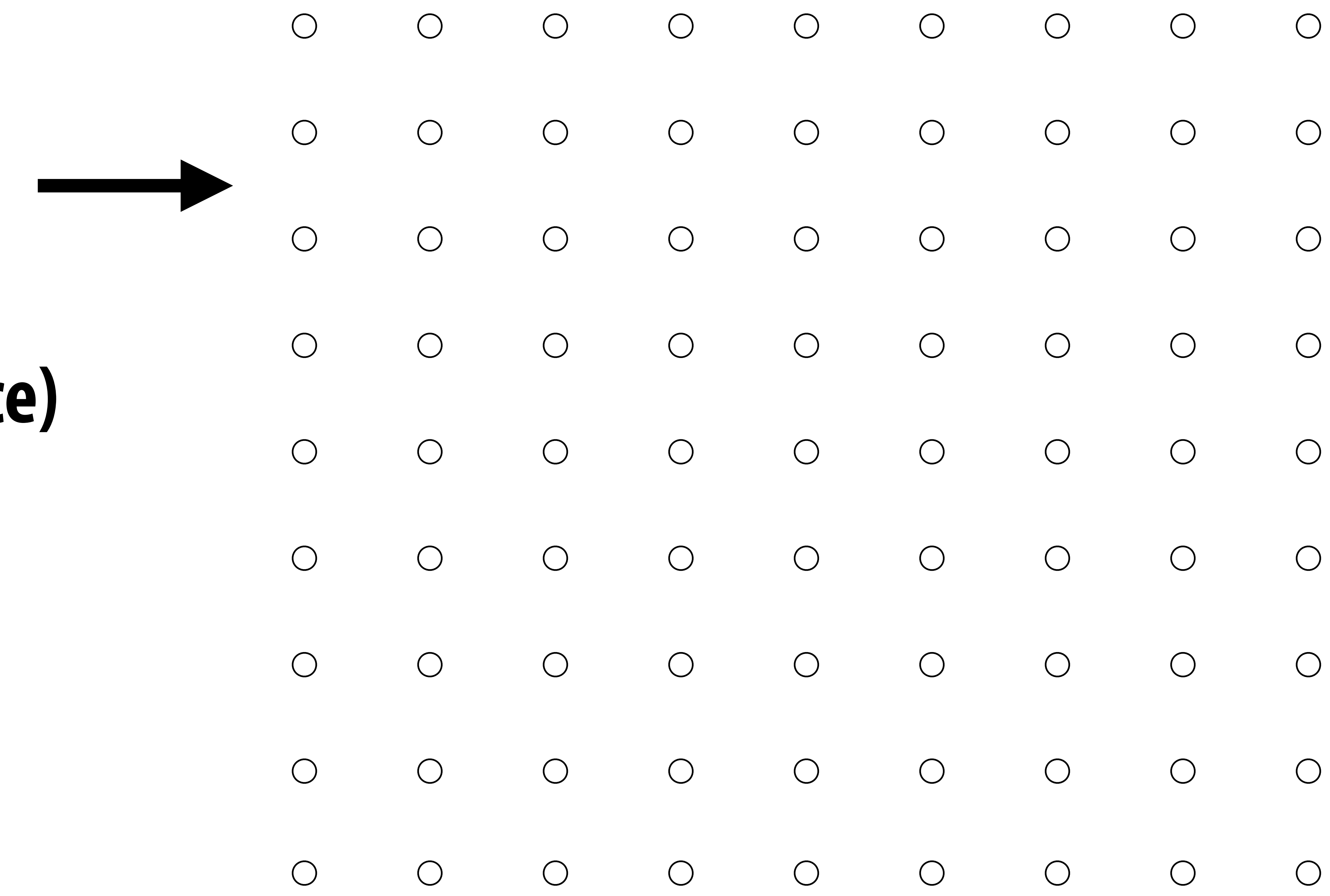
Corresponding depth buffer after rendering all triangles (stores closest scene depth per sample)

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

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

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

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



Grayscale value of sample point used to indicate distance

Black = small distance

White = large distance

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.

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

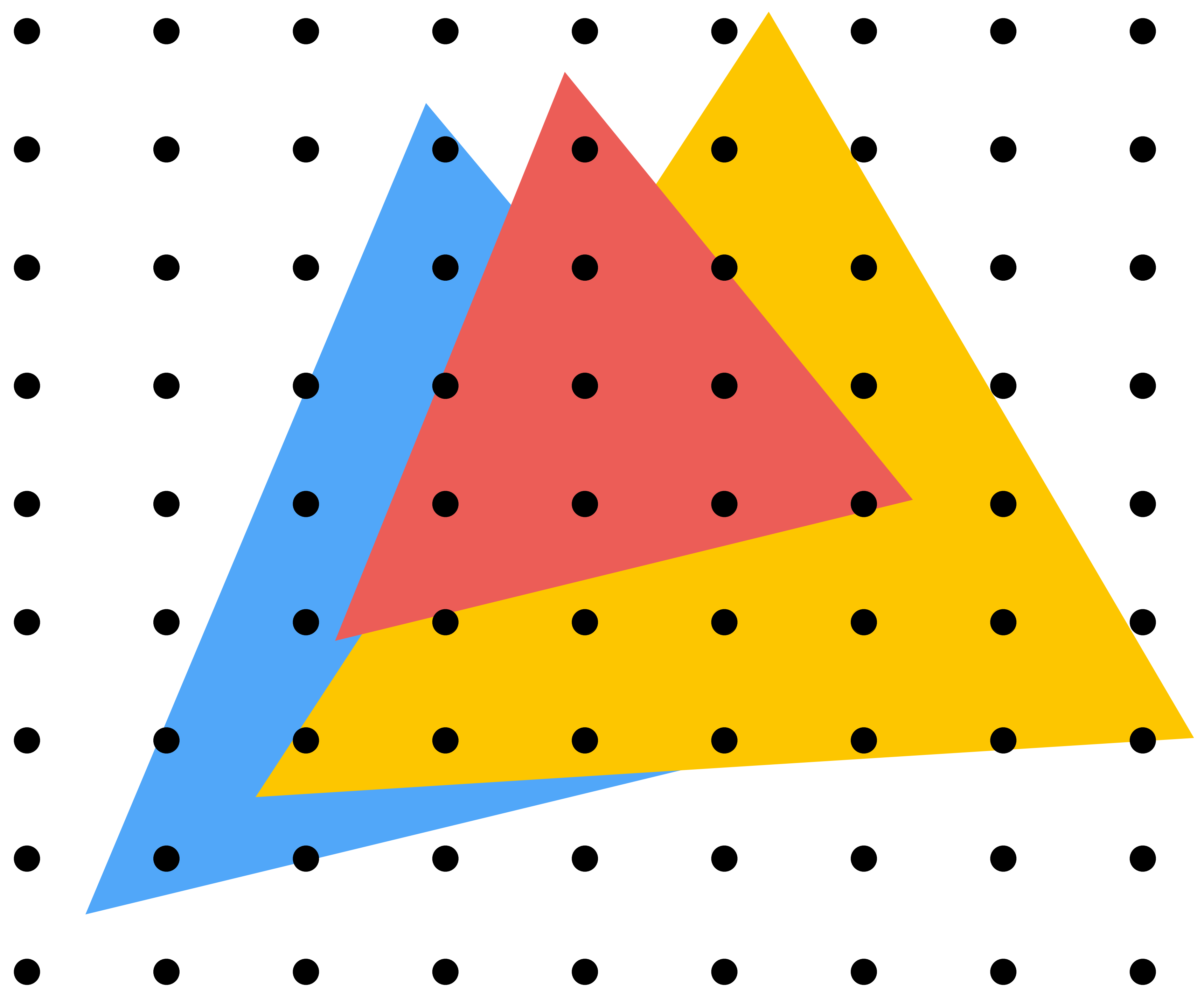
$$\begin{bmatrix}\mathbf{p}_{1x} & \mathbf{p}_{1y}\end{bmatrix}^T, \quad d_1$$

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

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:

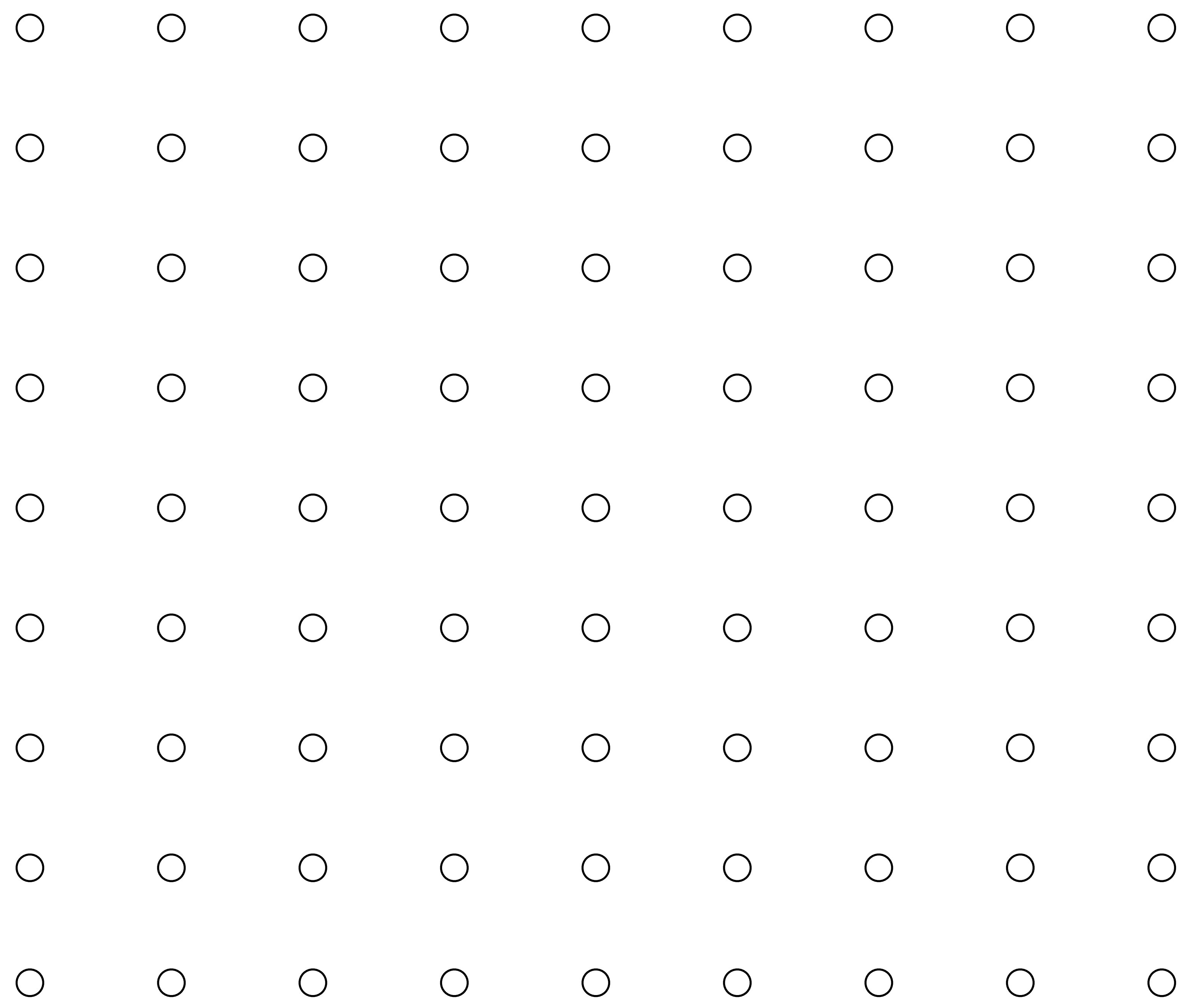
depth = 0.5

Grayscale value of sample point used to indicate distance

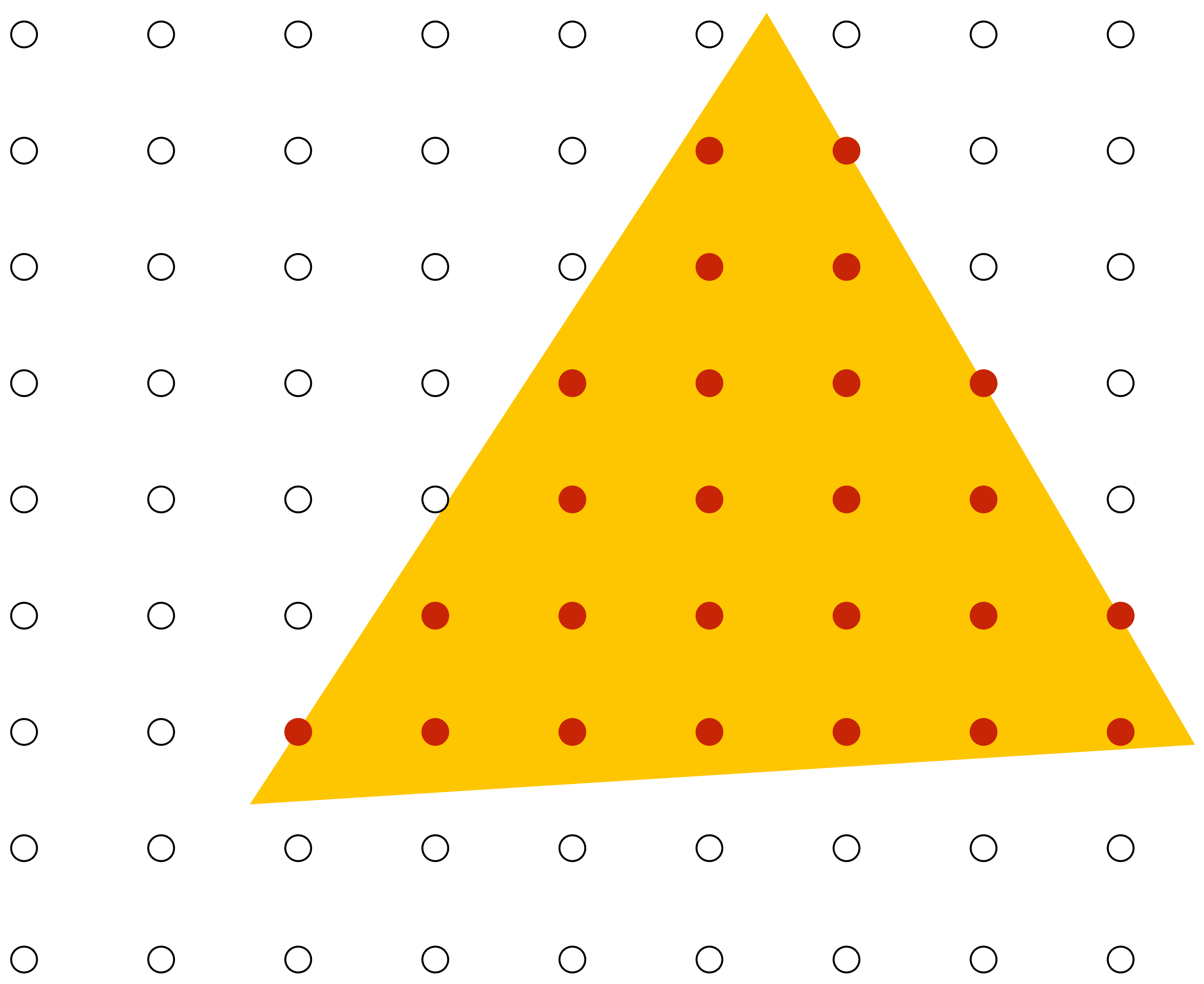
White = large distance

Black = small distance

Red = samples that pass depth test



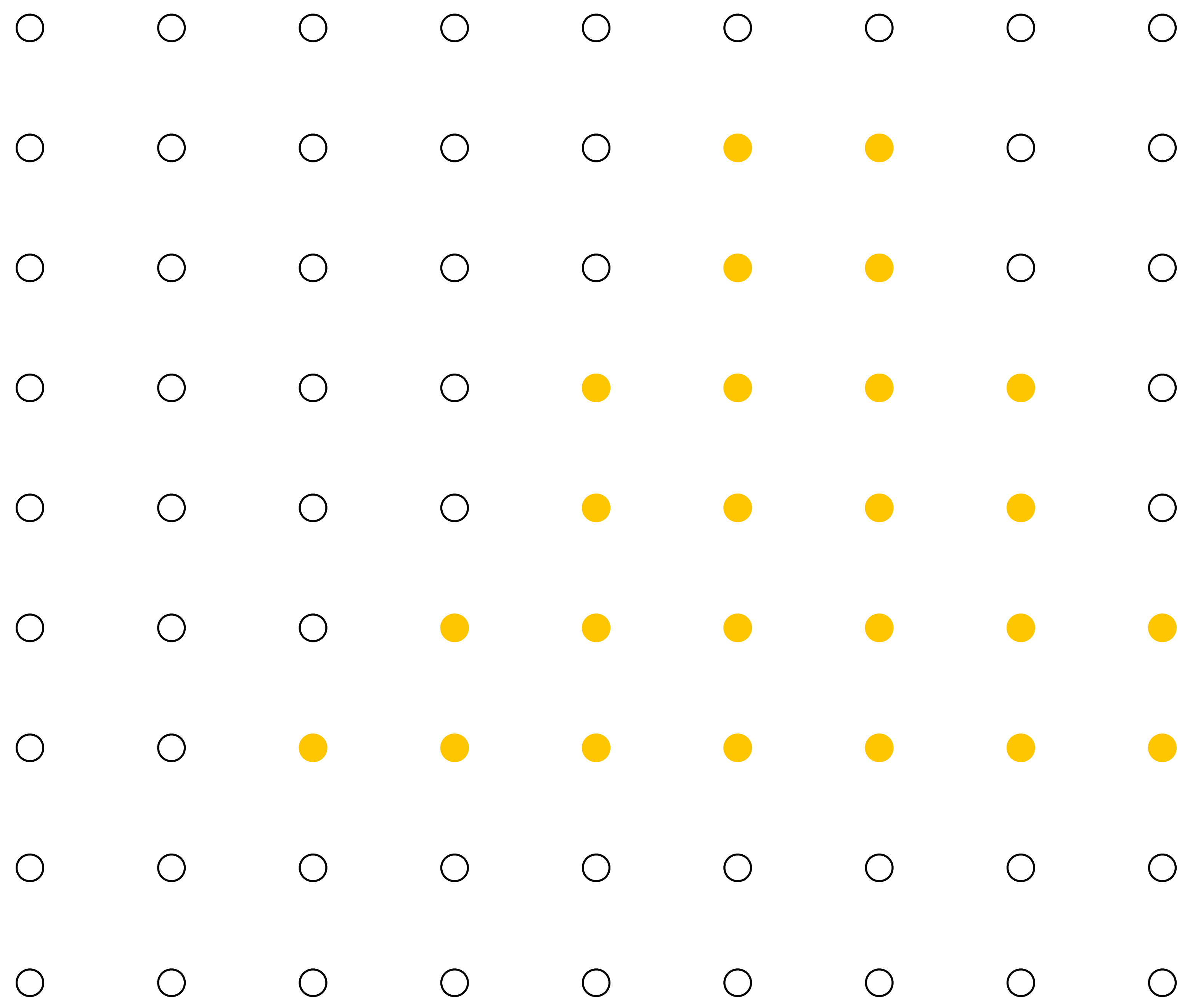
Color buffer contents



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing yellow triangle:



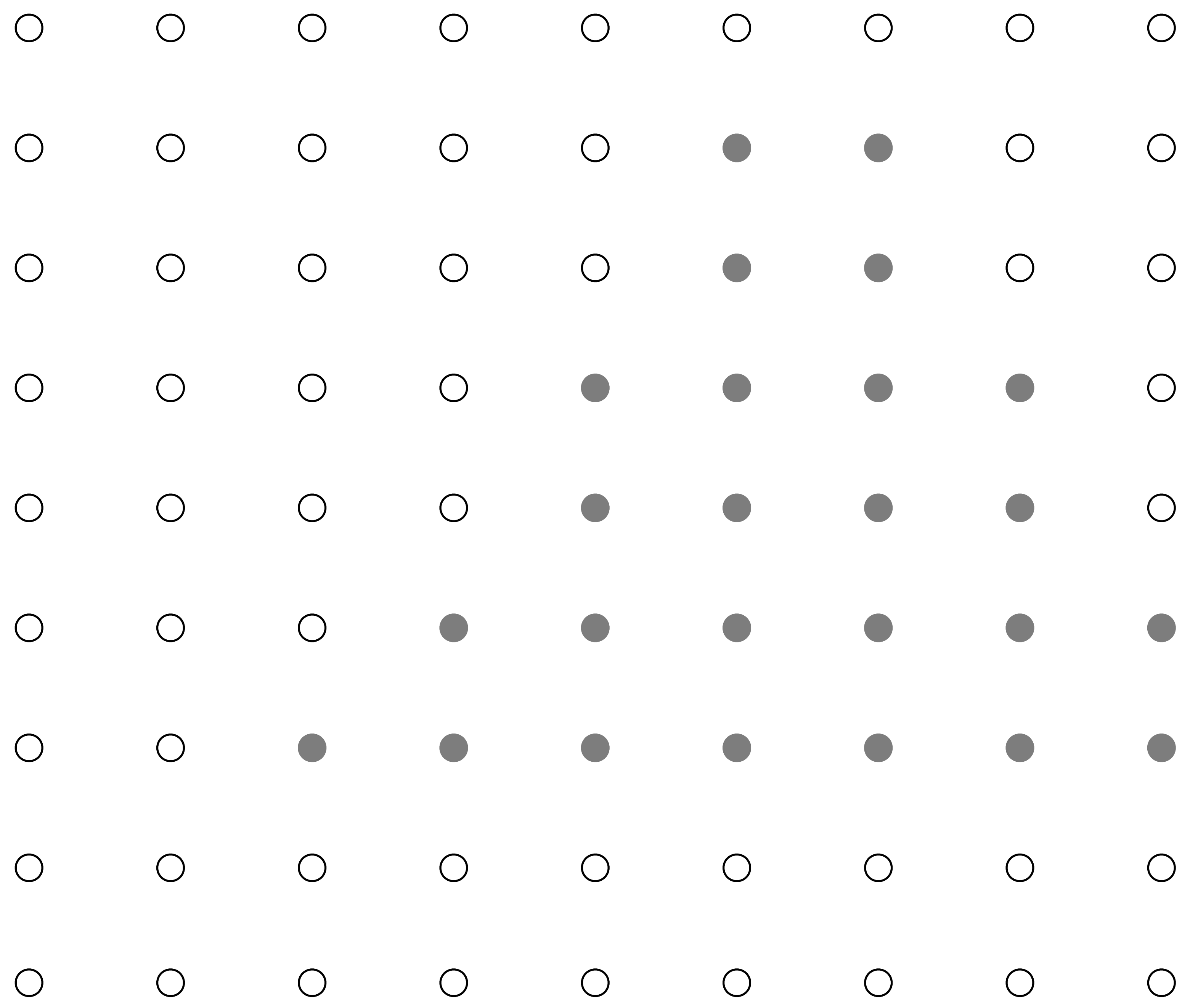
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

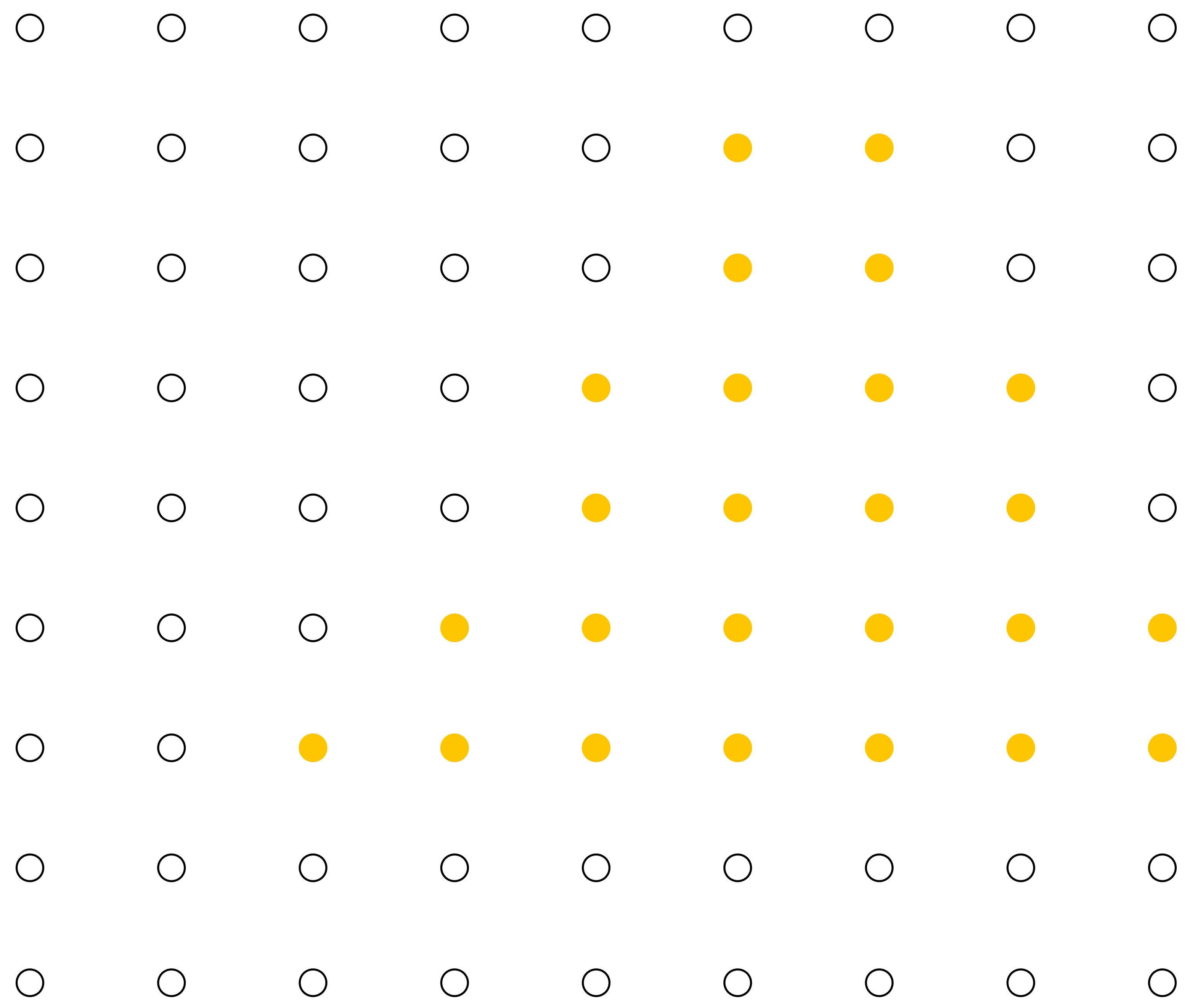
Red = samples that pass depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing blue triangle:
depth = 0.75



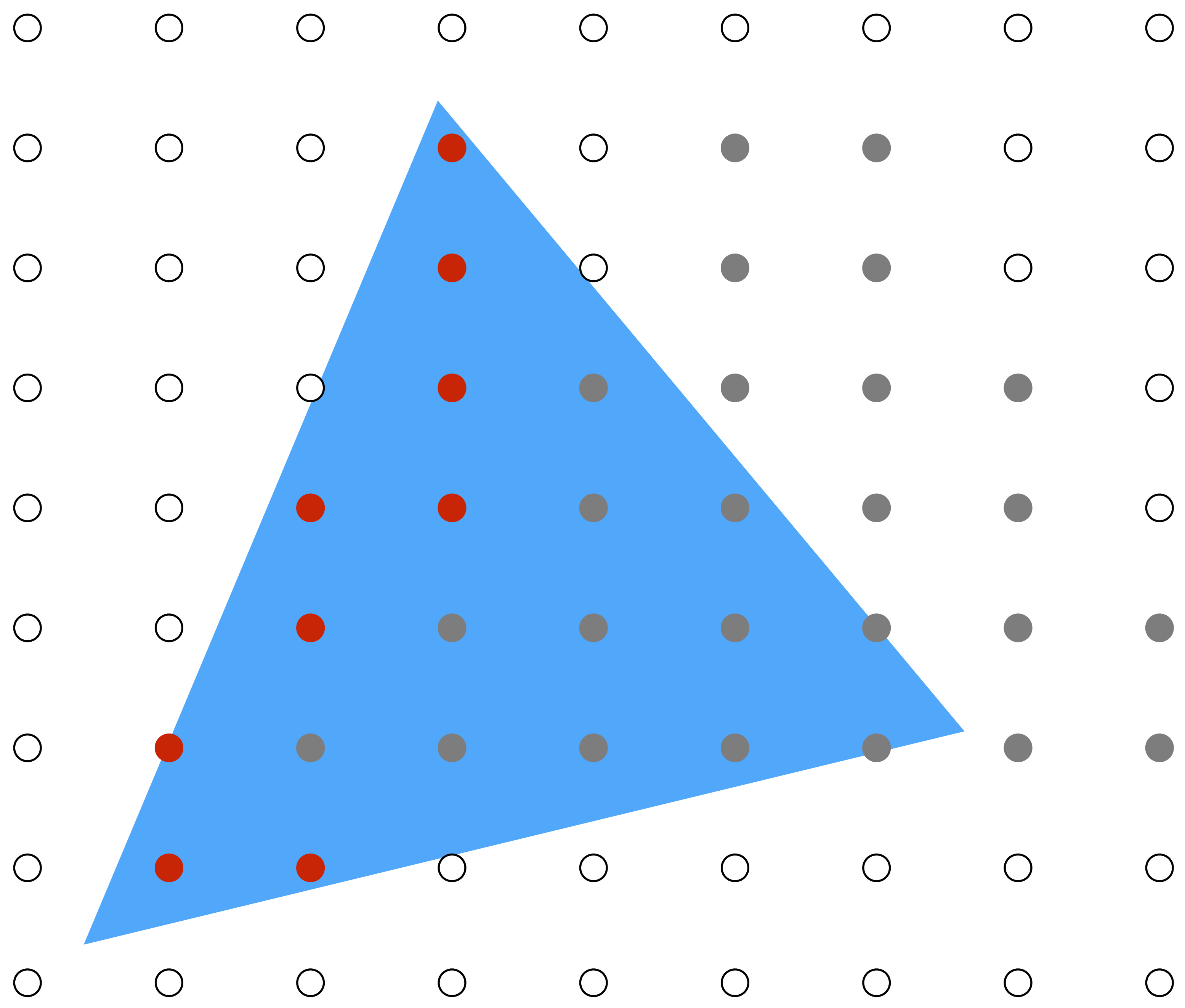
Color buffer contents

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)

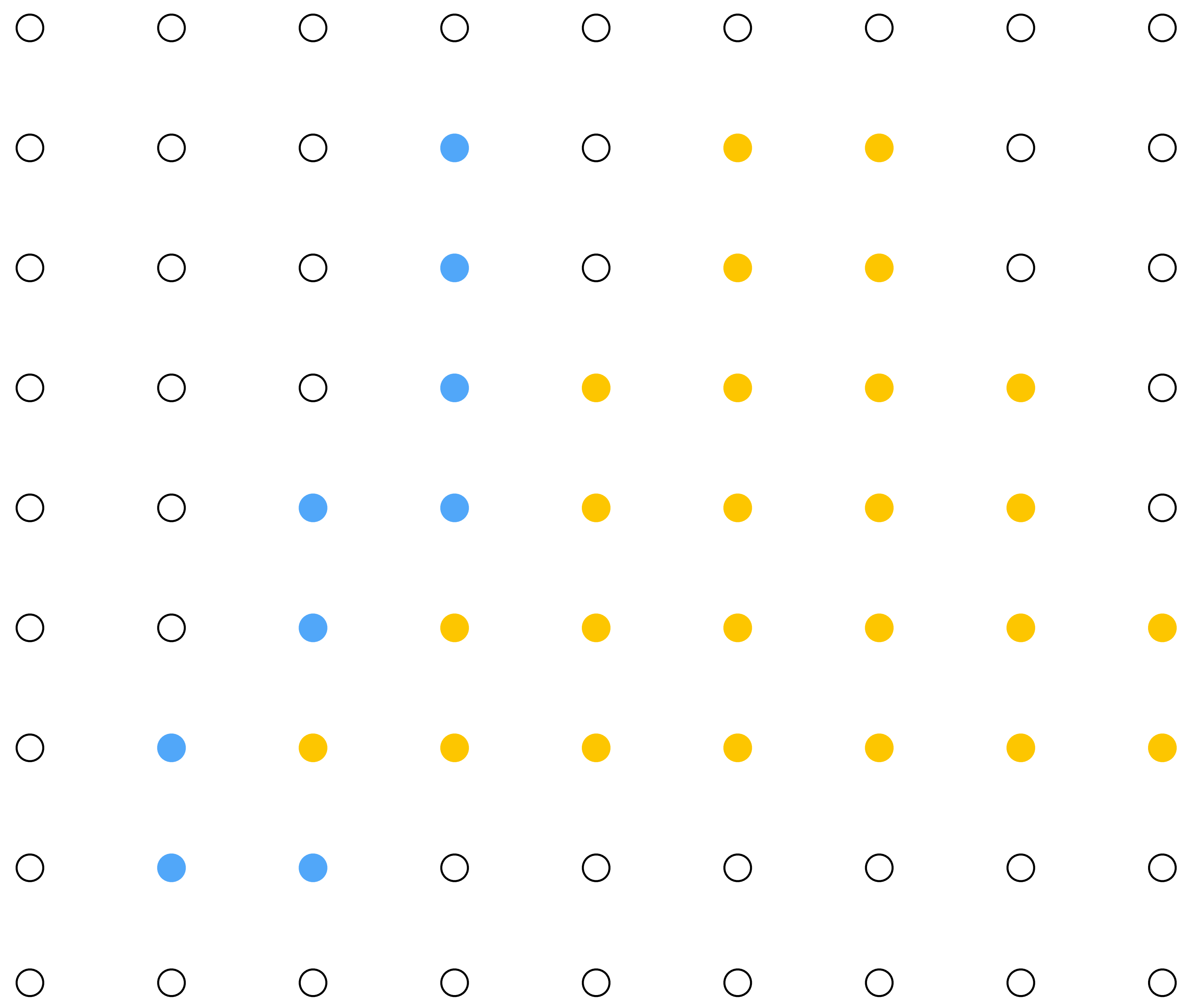
After processing blue triangle:

Grayscale value of sample point used to indicate distance

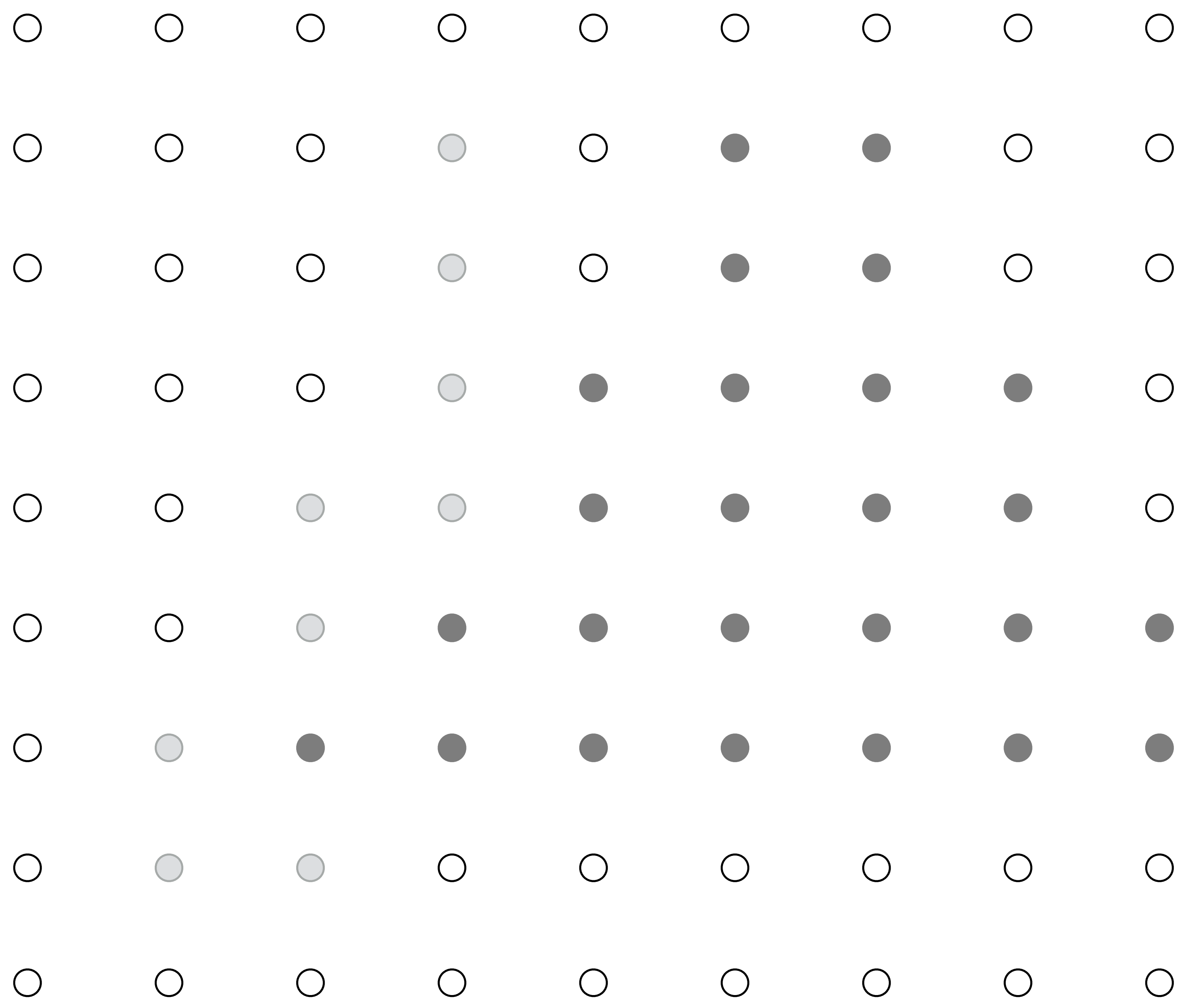
White = large distance

Black = small distance

Red = samples that pass depth test



Color buffer contents

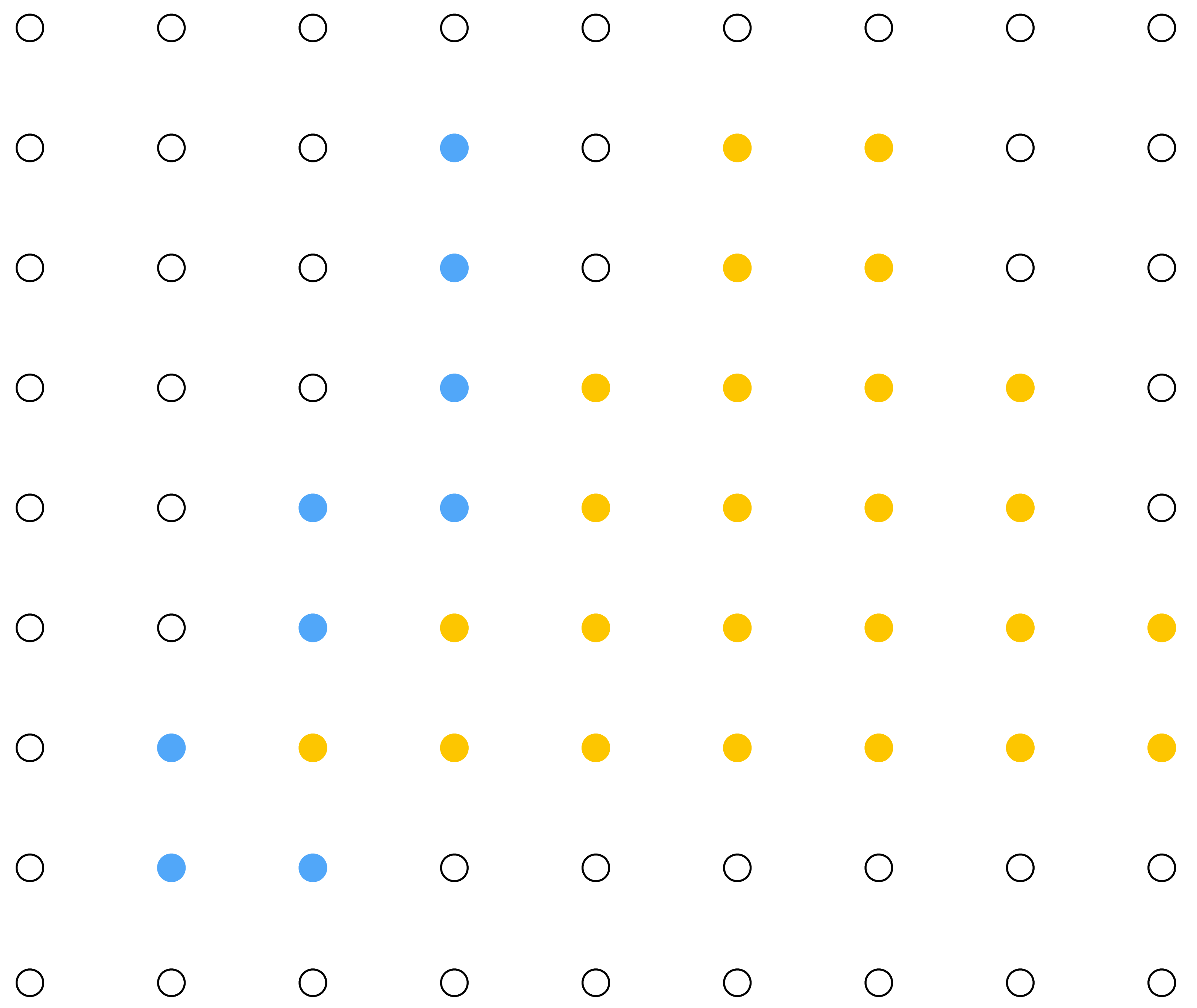


Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:

depth = 0.25



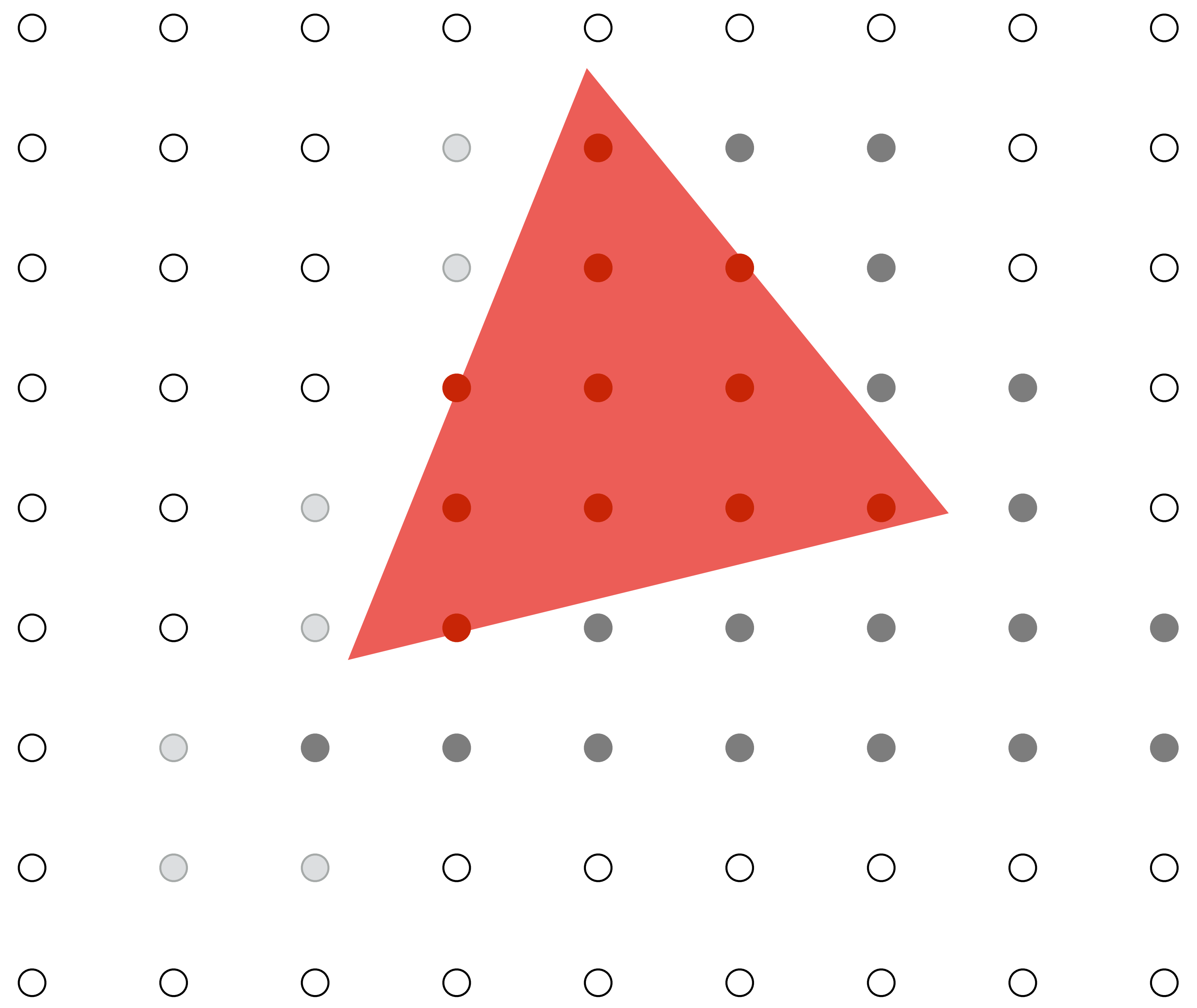
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

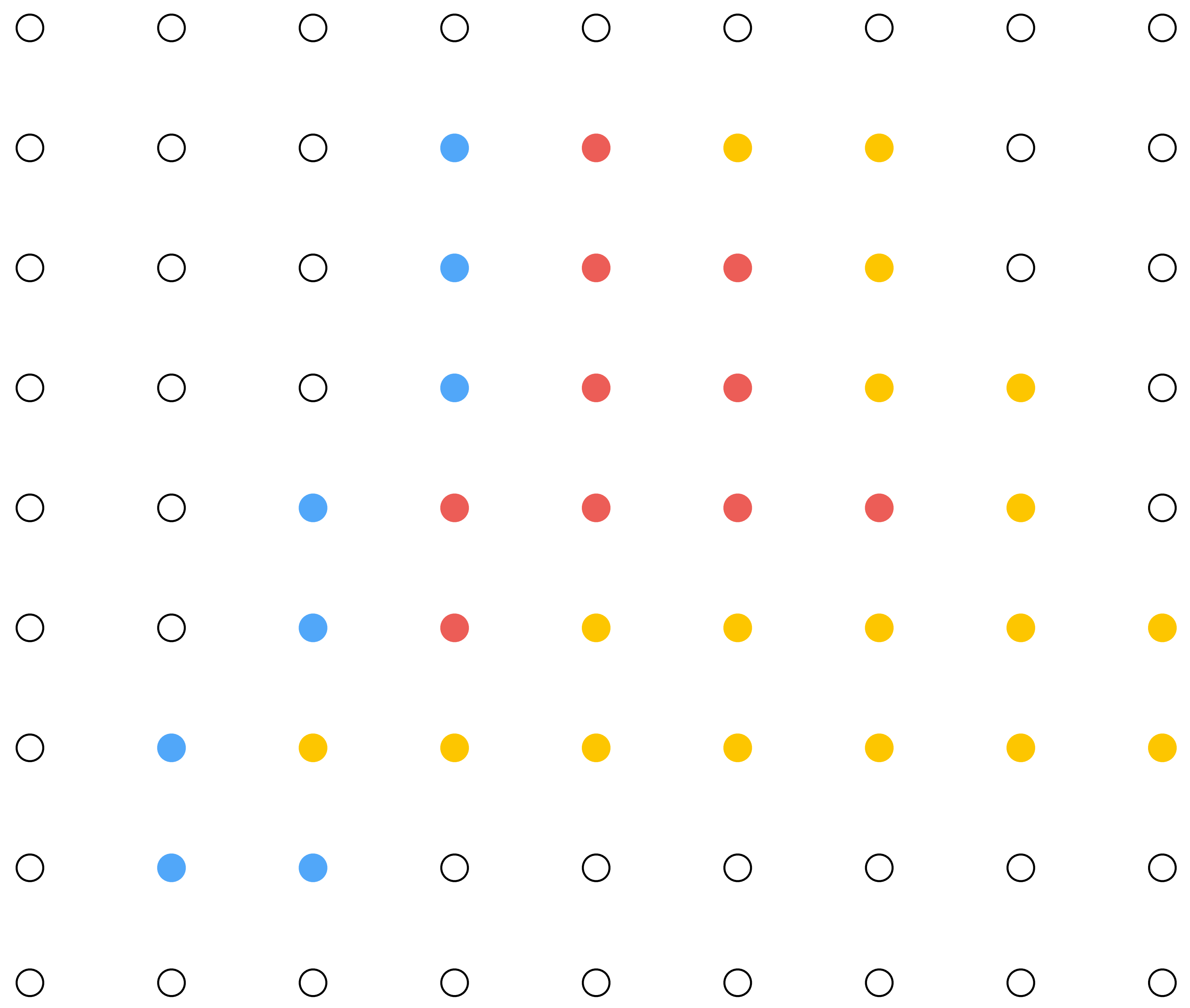
Red = samples that pass depth test



Depth buffer contents

Occlusion using the depth-buffer (Z-buffer)

After processing red triangle:



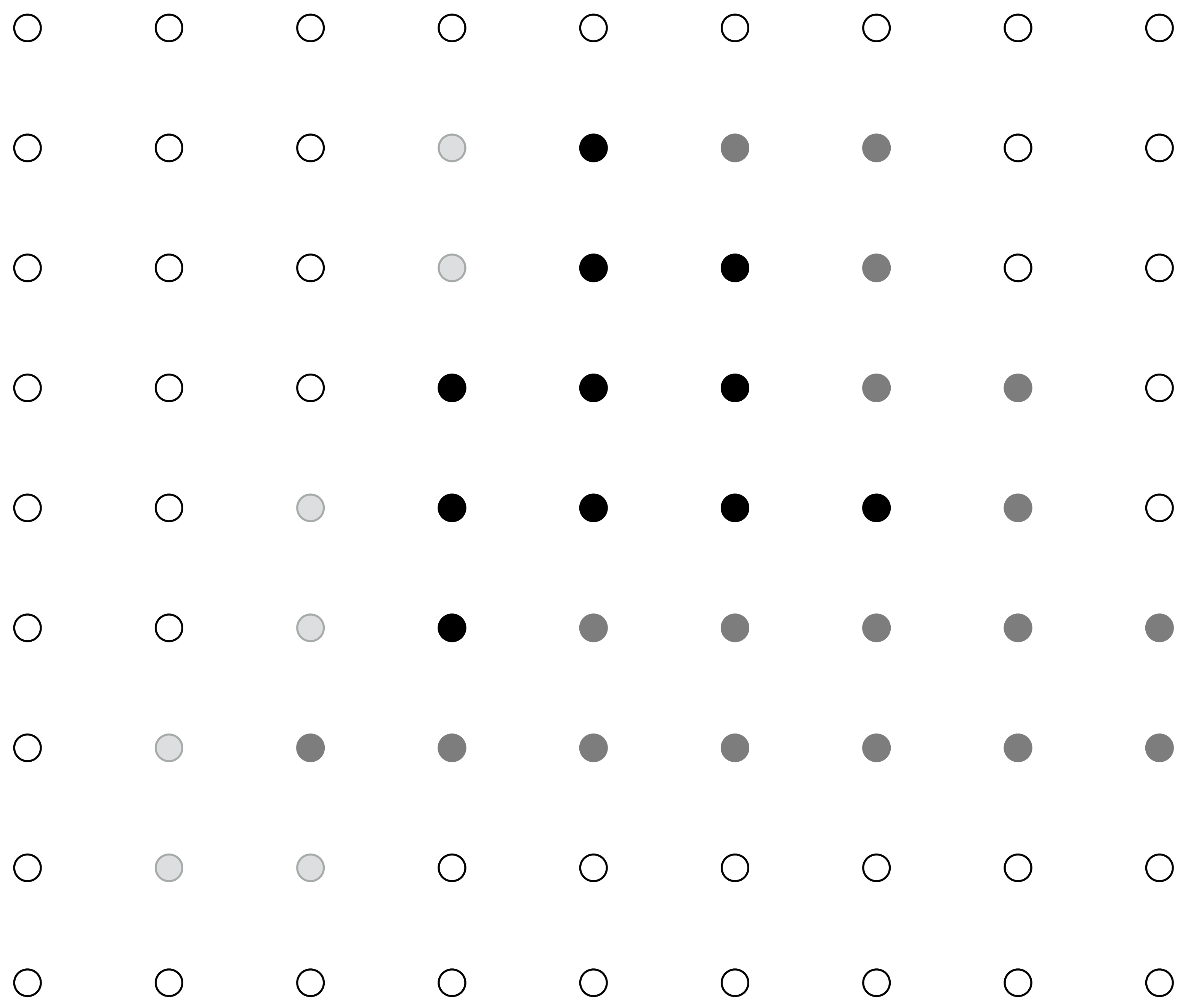
Color buffer contents

Grayscale value of sample point used to indicate distance

White = large distance

Black = small distance

Red = samples that pass depth test



Depth buffer contents

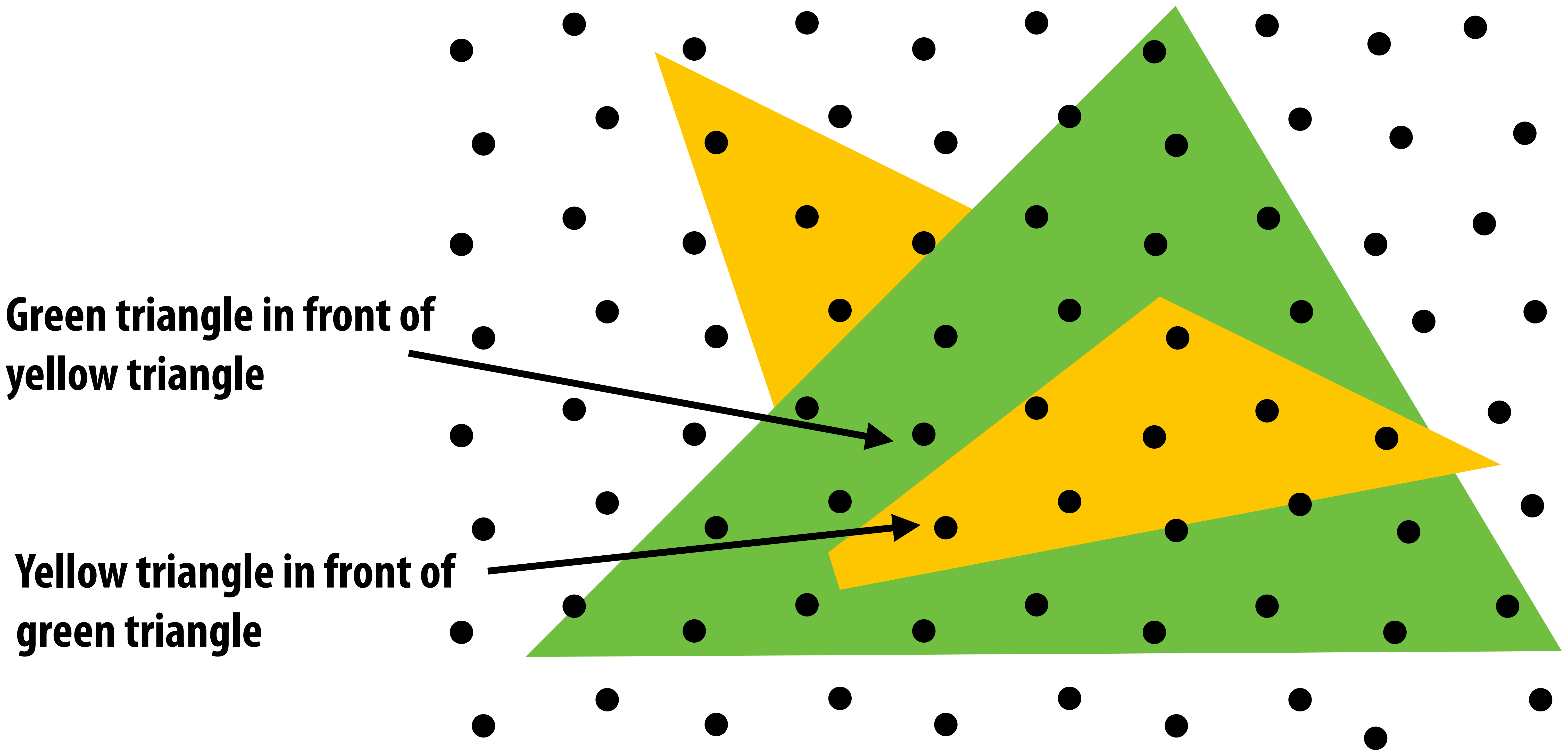
Occlusion using the depth buffer (opaque surfaces)

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


depth_test(tri_d, tri_color, x, y) {

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

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

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

Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

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

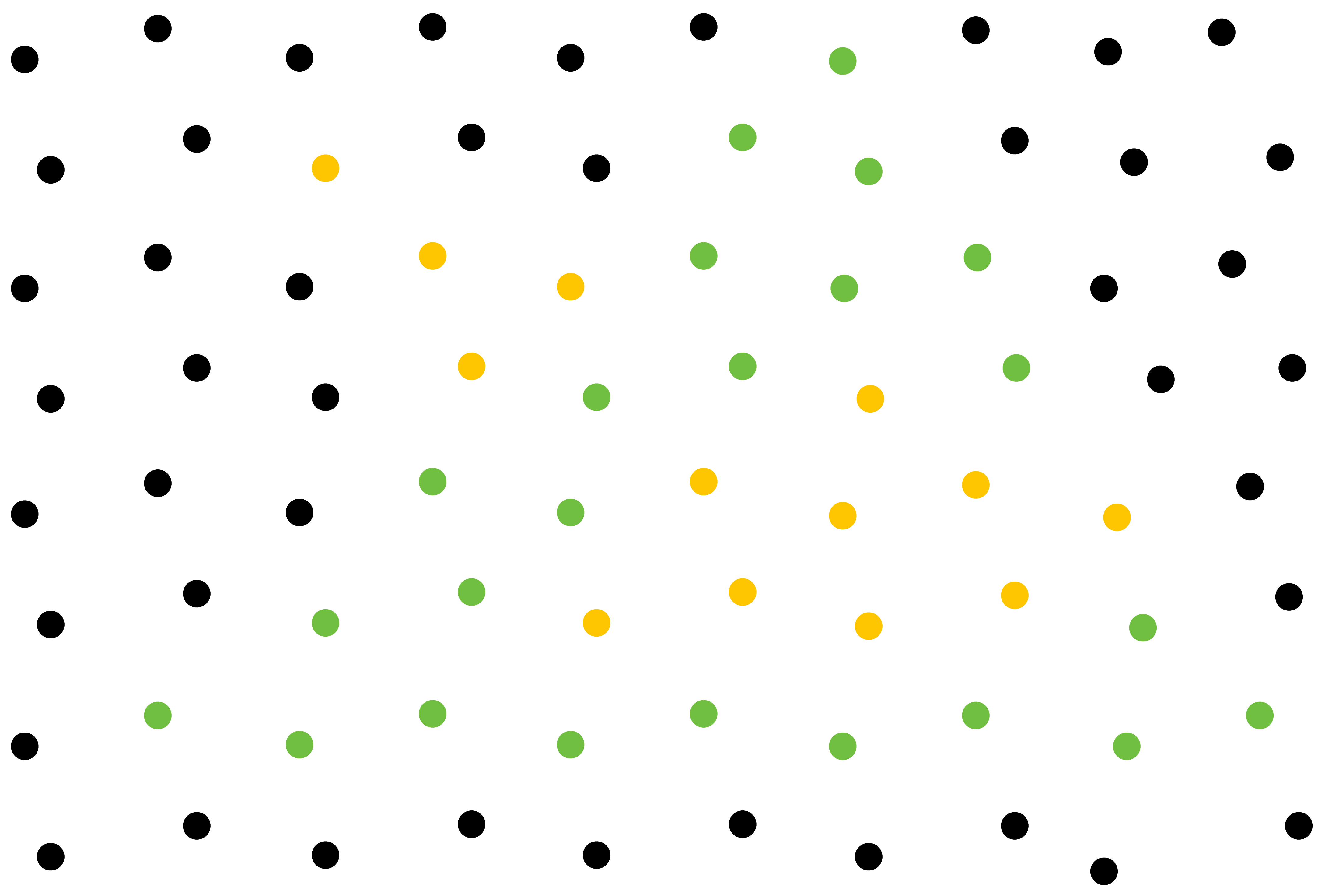


Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

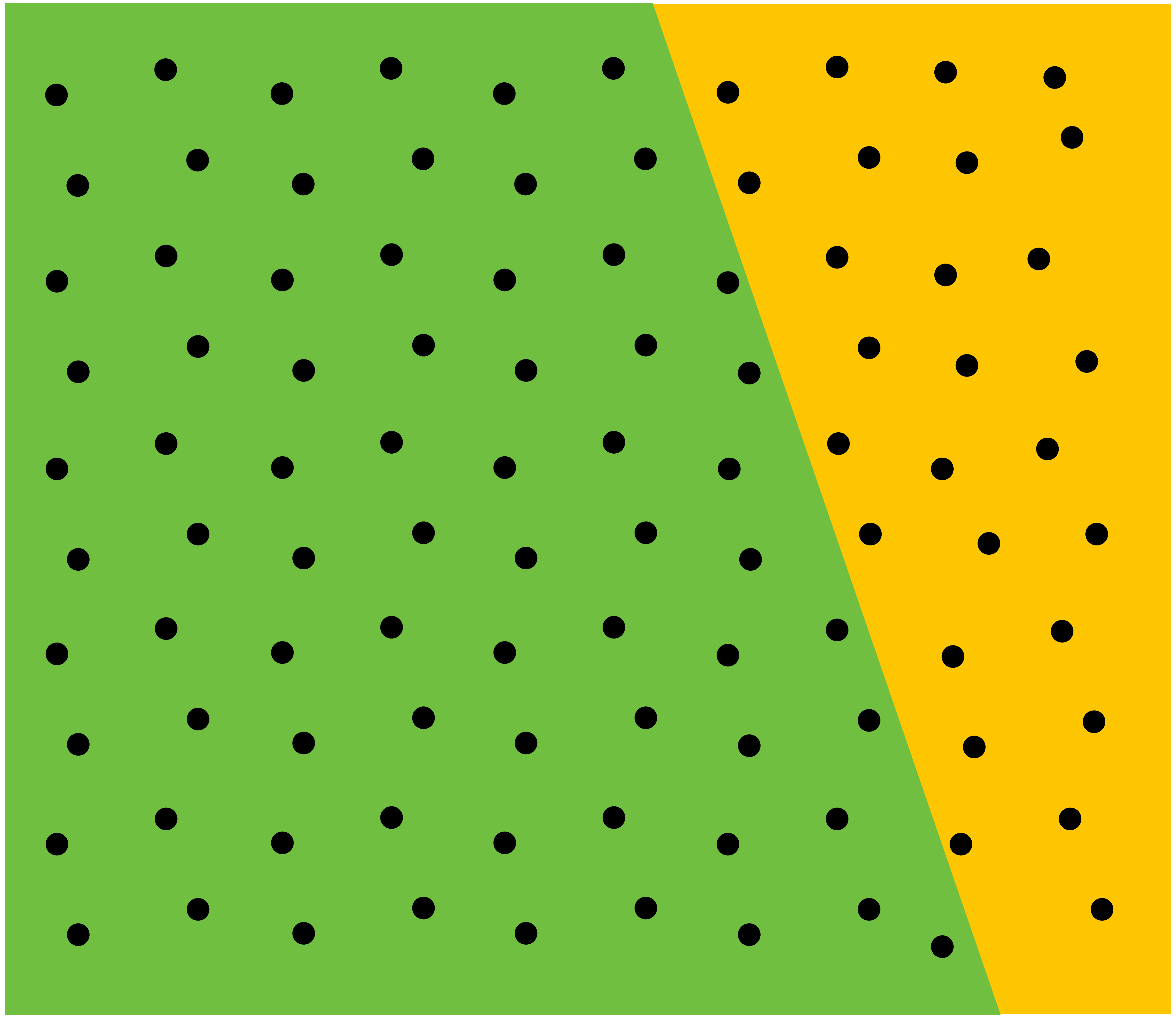
Occlusion test is based on depth of triangles *at a given sample point.* The relative depth of triangles may be different at different sample points.

Now only showing colored samples:



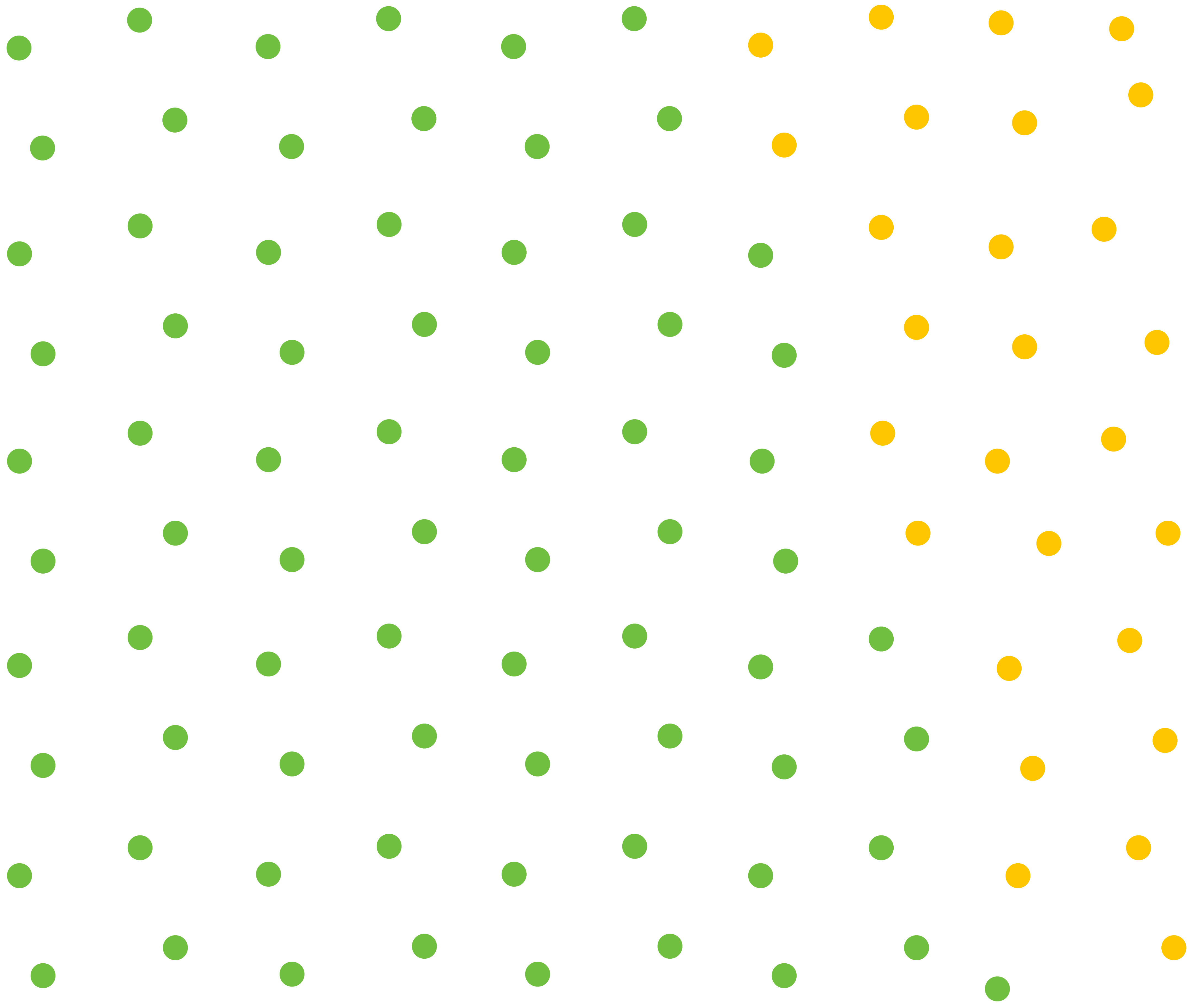
Does depth buffer work with super sampling?

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

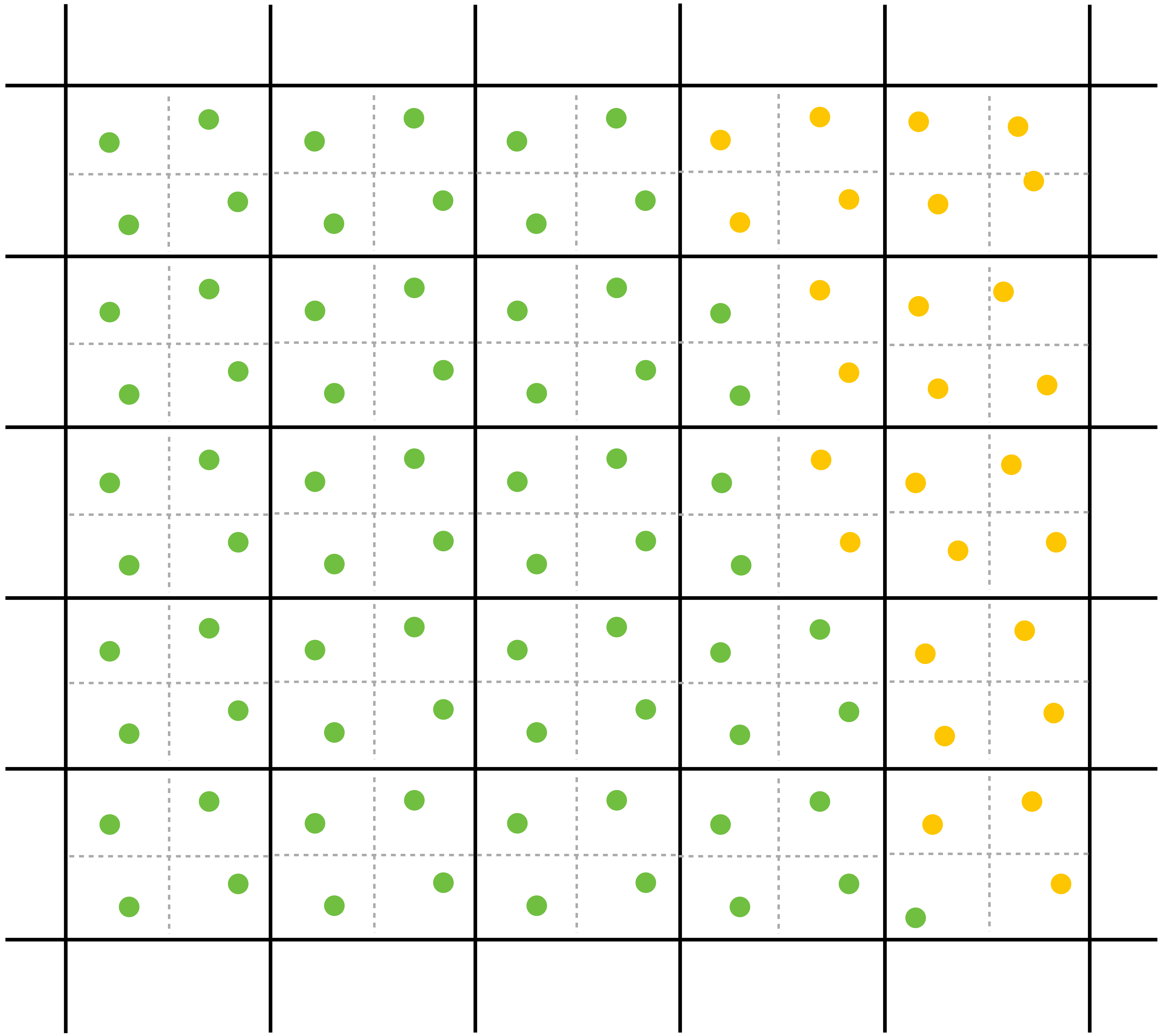


This example: green triangle occludes yellow triangle

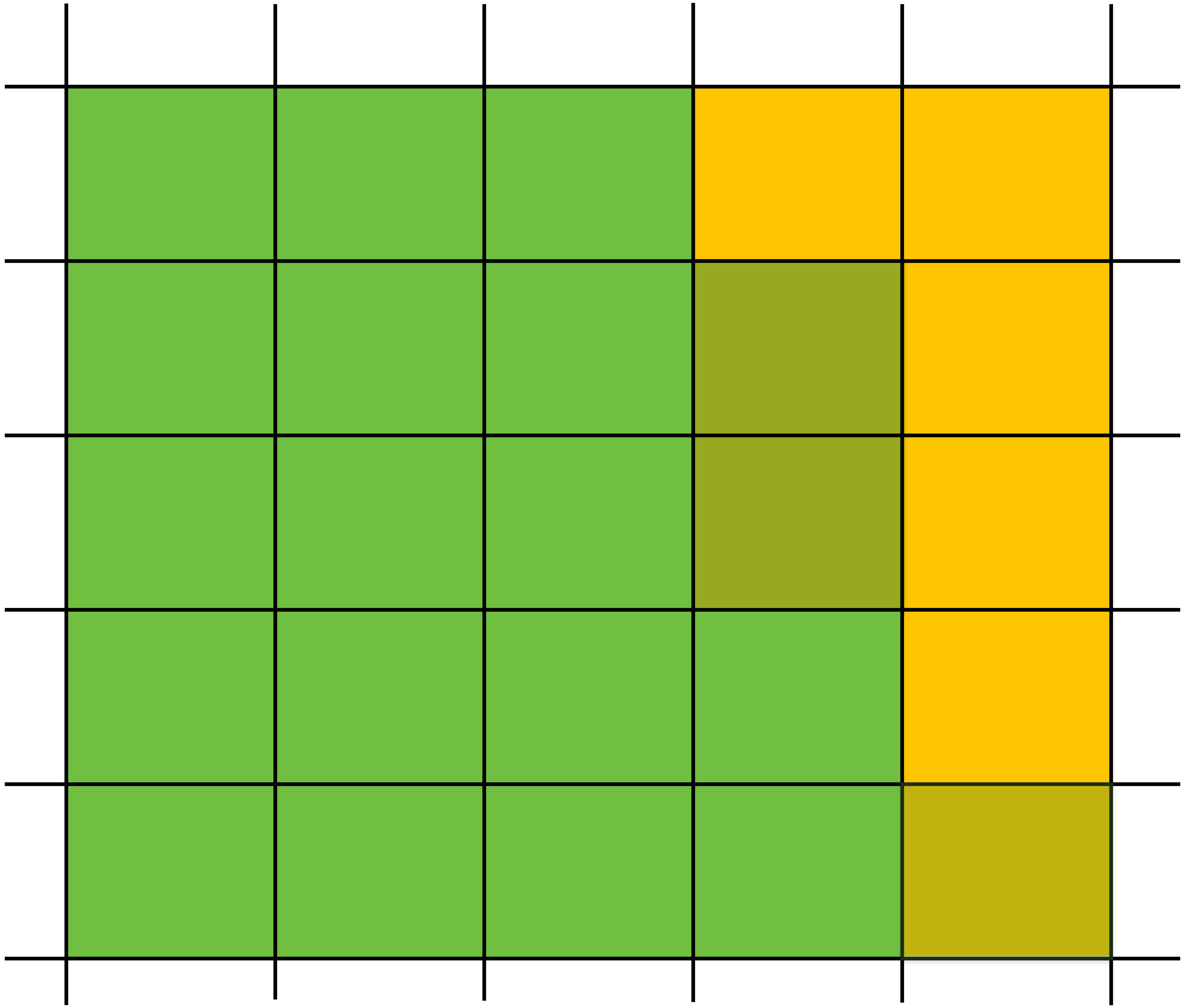
Color buffer contents



Color buffer contents (4 samples per pixel)



Final resampled result



Note anti-aliasing of edge due to filtering of green and yellow samples.

Summary: occlusion using a depth buffer

- Store one depth value per coverage sample (not per pixel!)
- Constant space per sample
  - Implication: constant space for depth buffer
- Constant time occlusion test per covered sample
  - Read-modify write of depth buffer if "pass" depth test
  - Just a depth buffer read if "fail"
- Not specific to triangles: only requires that surface depth can be evaluated at a screen sample point

But what about semi-transparent surfaces?

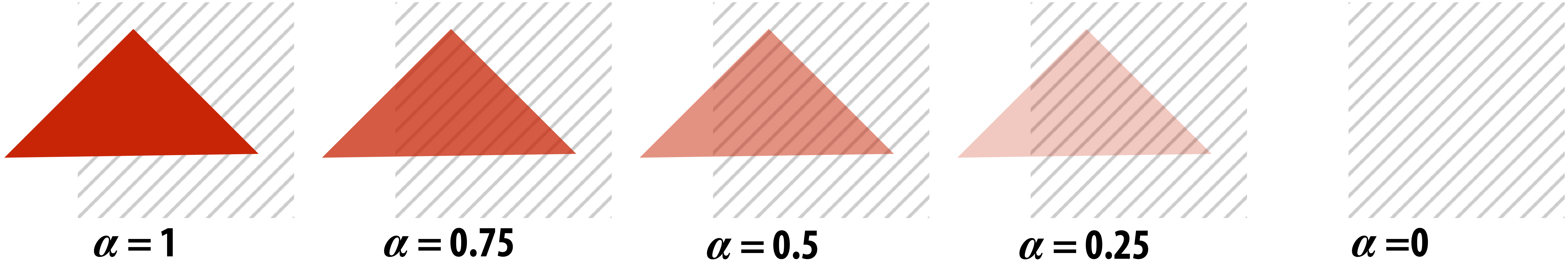
Compositing

Representing opacity as alpha

Alpha describes the opacity of an object

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

Red triangle with decreasing opacity



Alpha: coverage analogy

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

$\alpha = 0.5$

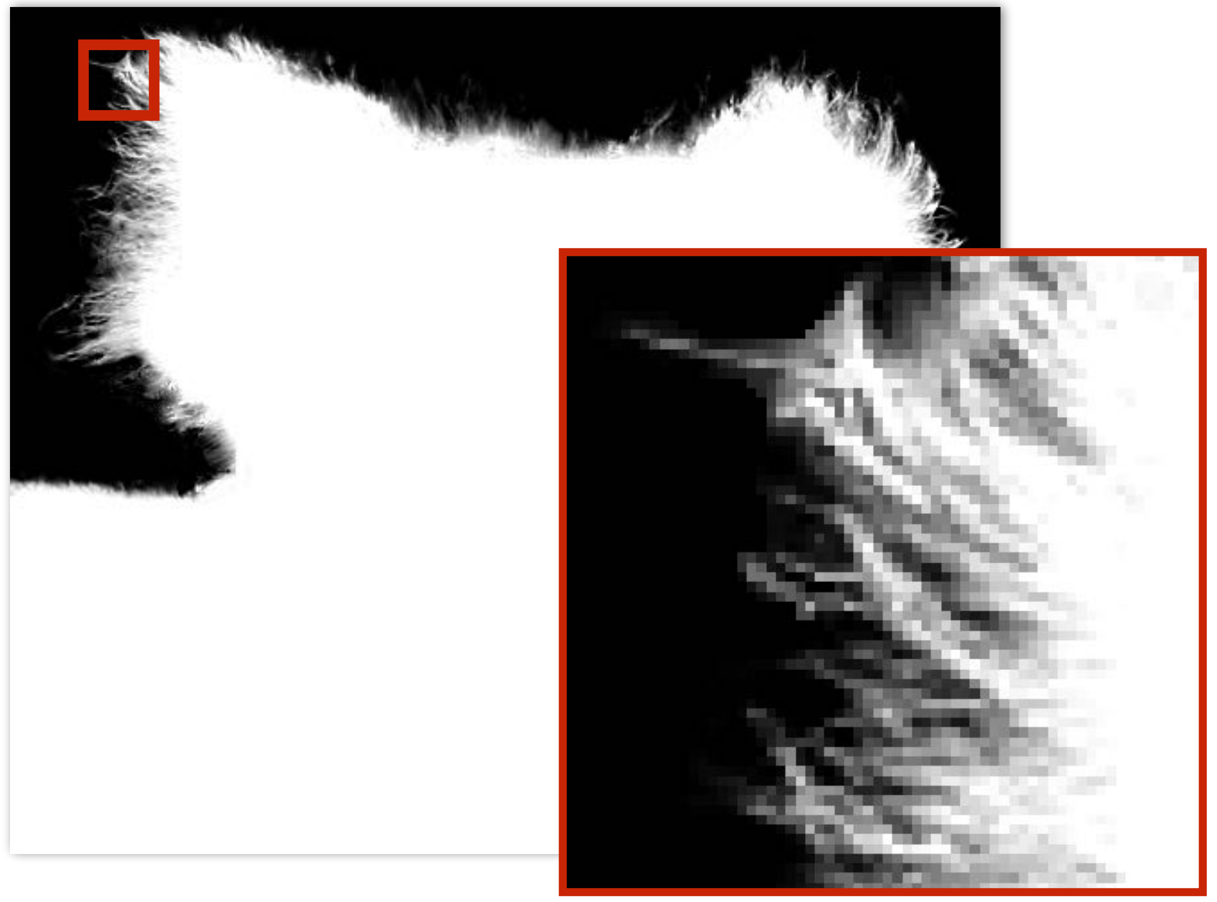
(Squint at this slide and the scene on the left and the right will appear similar)

Alpha: additional channel of image (rgba)

Alpha describes the opacity of an object

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

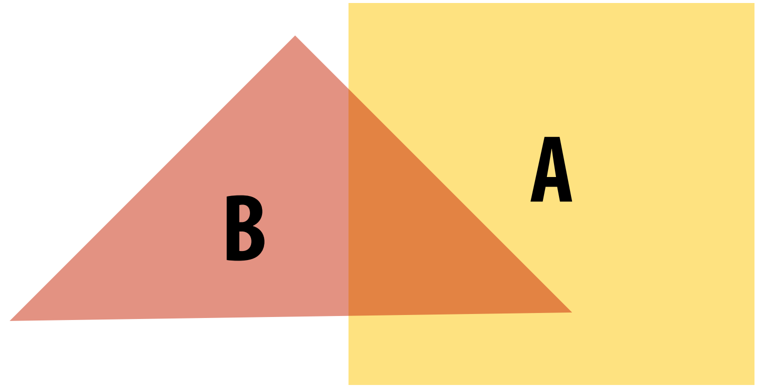


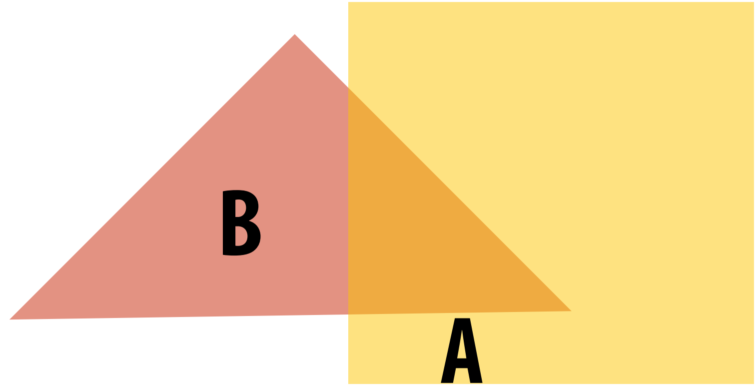
α of foreground object

Over operator:

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



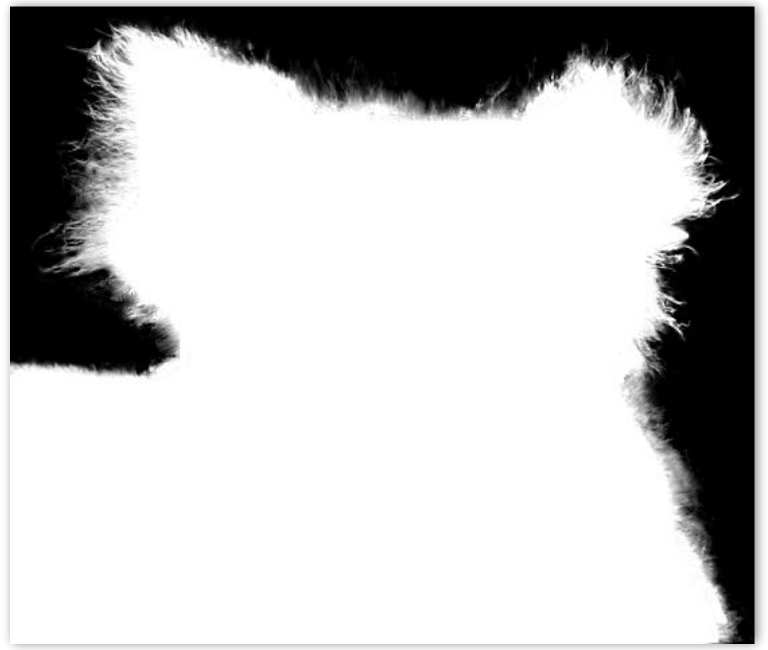
B over A



A over B

A over B != B over A

"Over" is not commutative



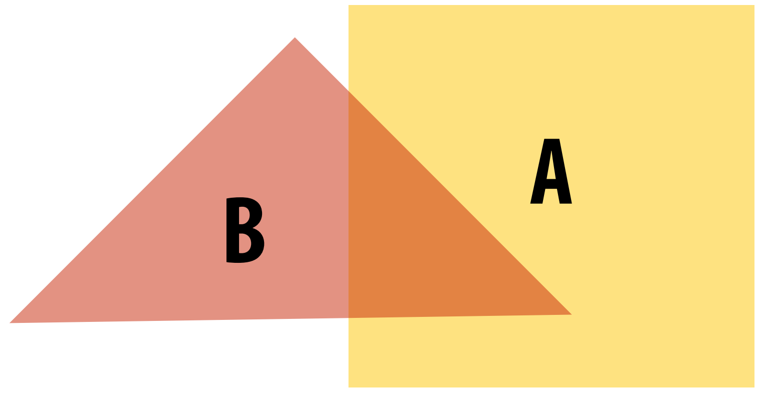
Koala over NYC

Over operator: non-premultiplied alpha

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

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

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



B over A

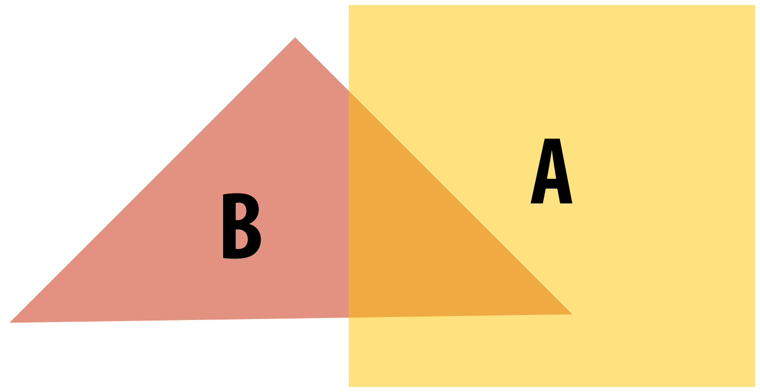
Composited color:

Appearance of semi-transparent A

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

Appearance of semi-transparent B

What B lets through



A over B

A over B != B over A

"Over" is not commutative

Composite alpha:

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

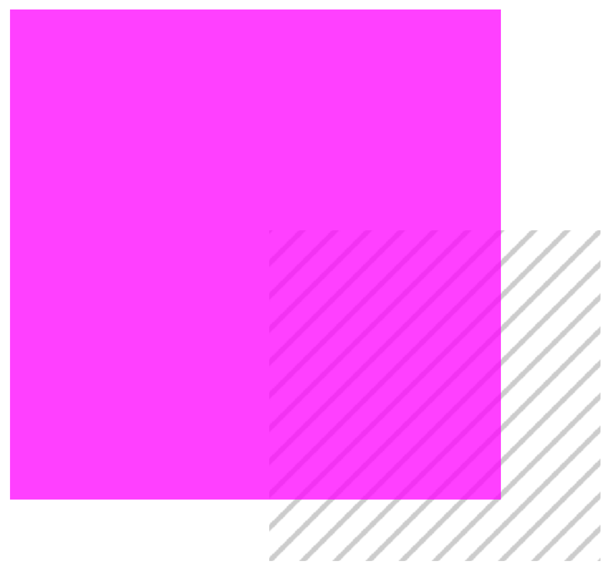
Premultiplied alpha representation

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

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

Example: 50% opaque red
[0.5, 0.0, 0.0, 0.5]

Example: 75% opaque magenta
[0.75, 0.0, 0.75, 0.75]



Over operator: using premultiplied alpha

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

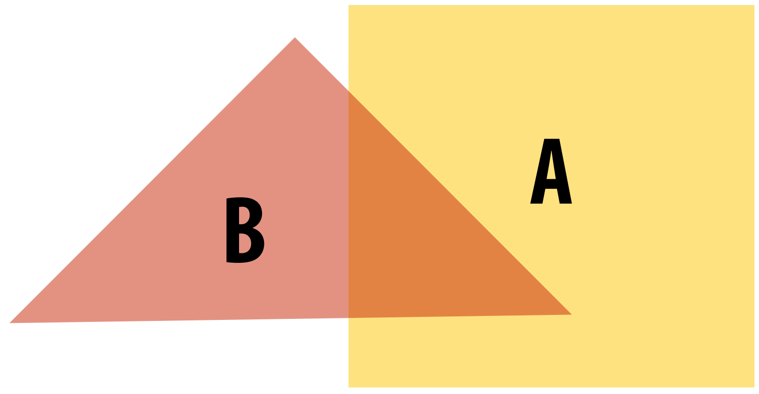
Non-premultiplied alpha representation:

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

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

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

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



B over A

Composite alpha:

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

Premultiplied alpha representation:

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

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

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

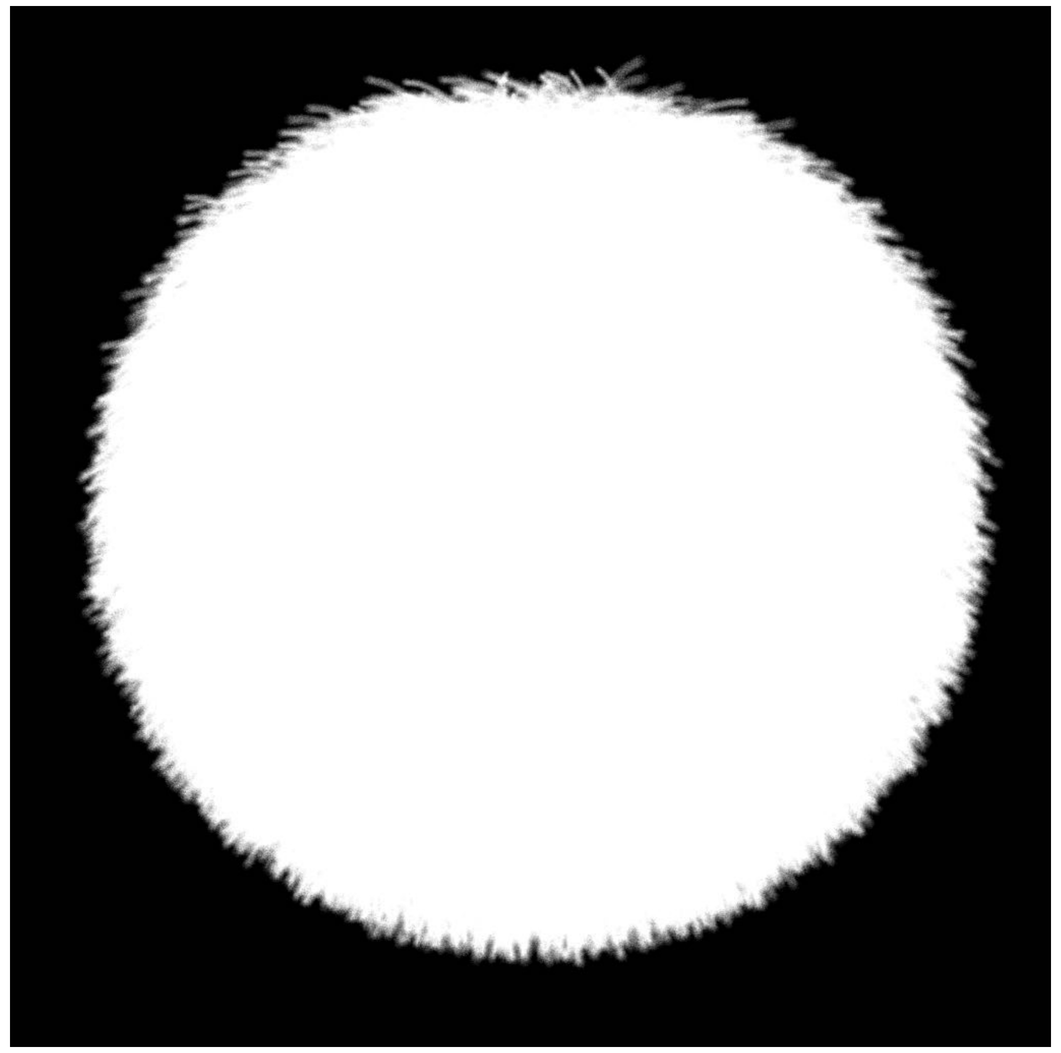
← **one multiply, one add**

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

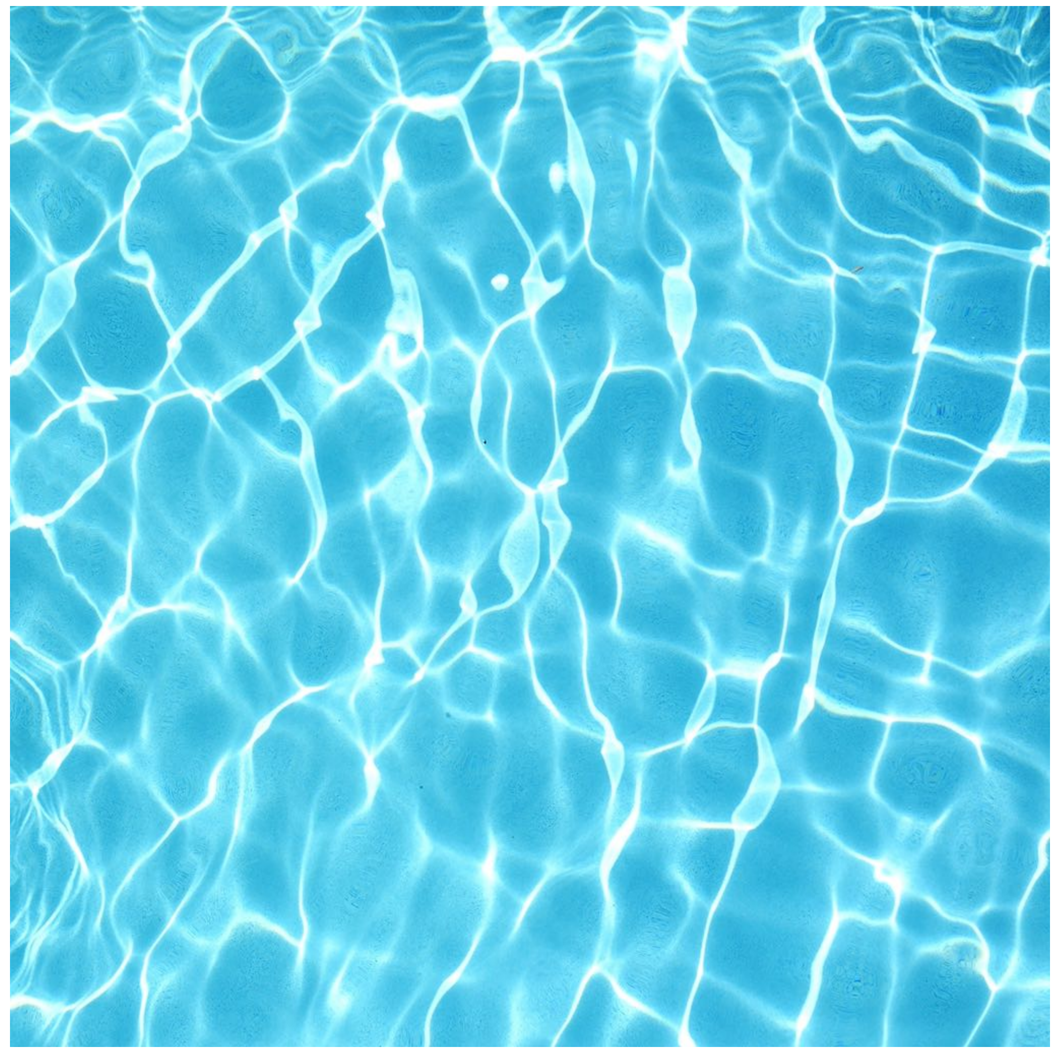
Fringing

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

foreground color



foreground alpha



background color

fringing

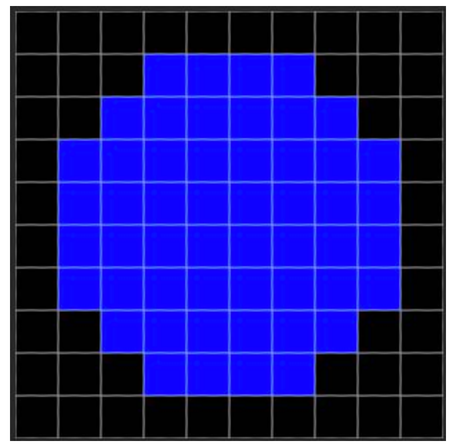
no fringing

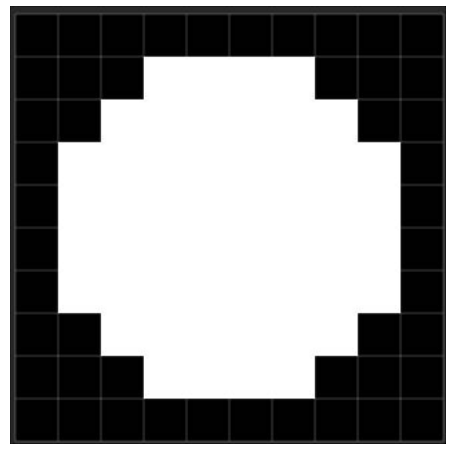
No fringing

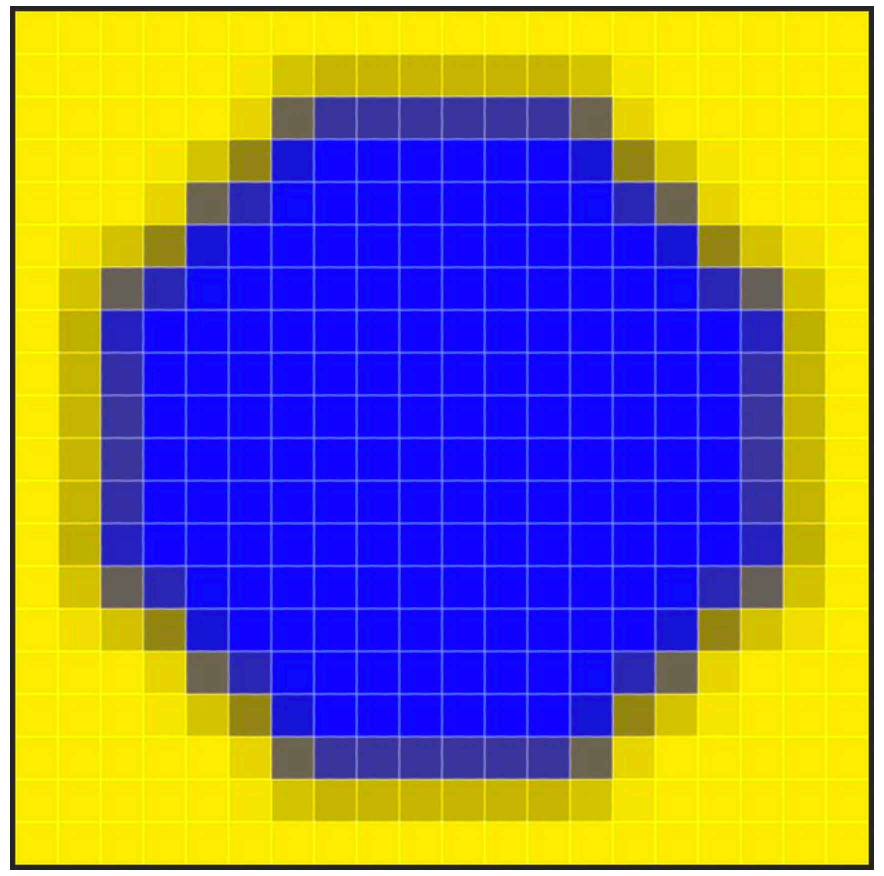
Fringing (…why does this happen?)

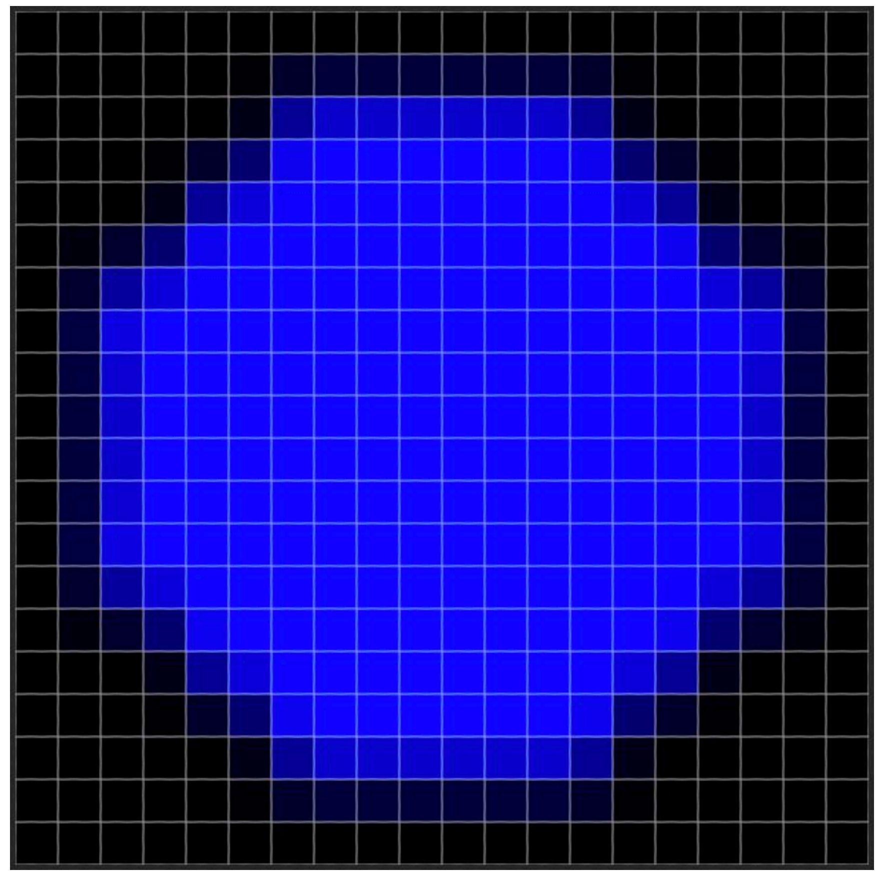
A problem with non-premultiplied alpha

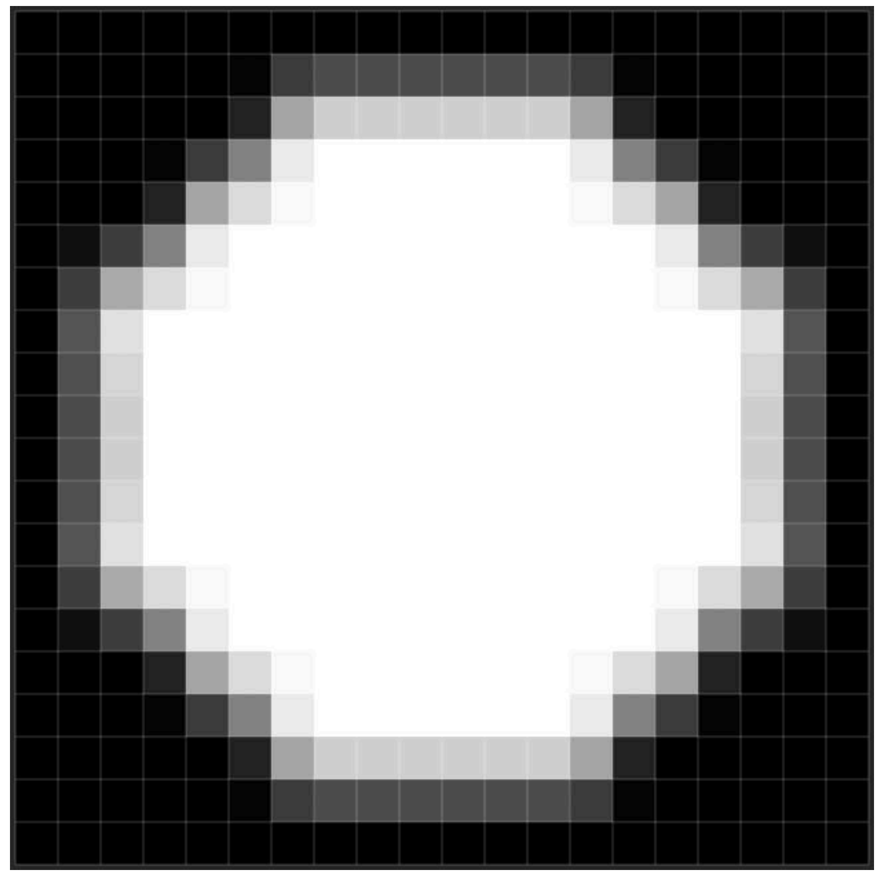
- Suppose we upsample an image w/ an alpha mask, then composite it onto a background
- How should we compute the interpolated color/alpha values?
- If we interpolate color and alpha separately, then blend using the non-premultiplied "over" operator, here's what happens:


original color


original alpha


composited onto yellow background

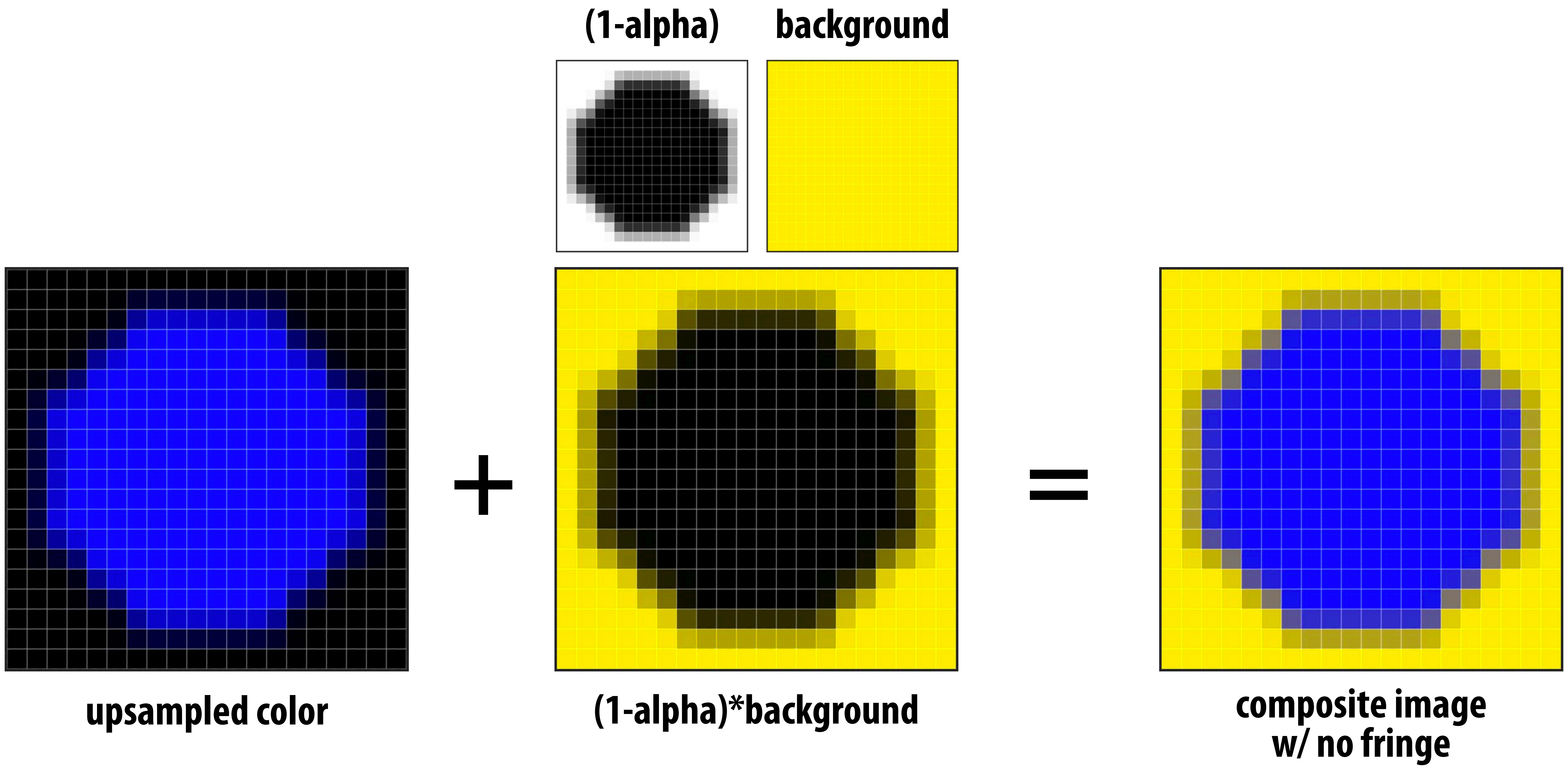

upsampled color


upsampled alpha

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

Eliminating fringe w/ premultiplied “over”

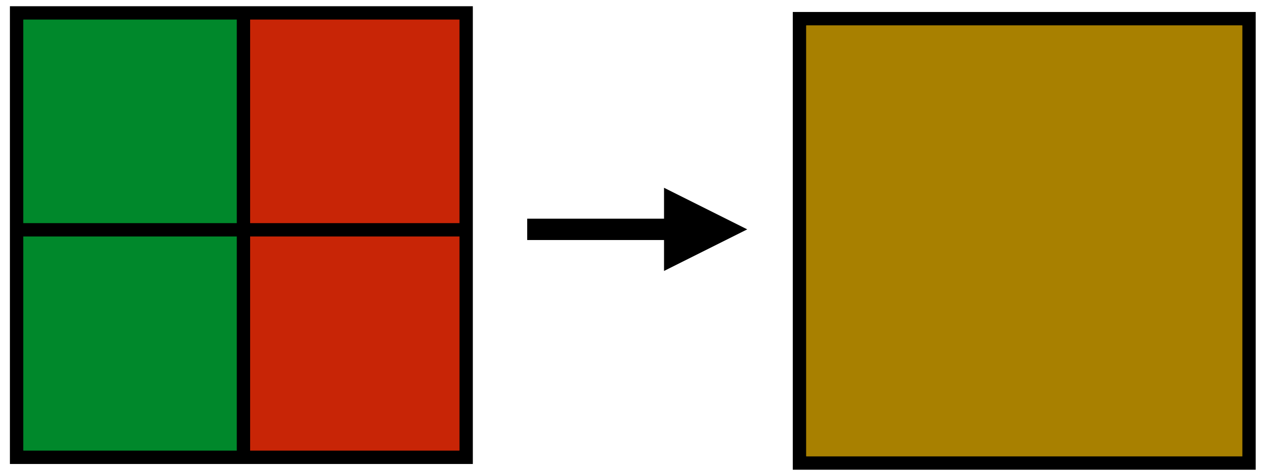
If we instead use the premultiplied “over” operation, we get the correct alpha:



Another problem with non-premultiplied alpha

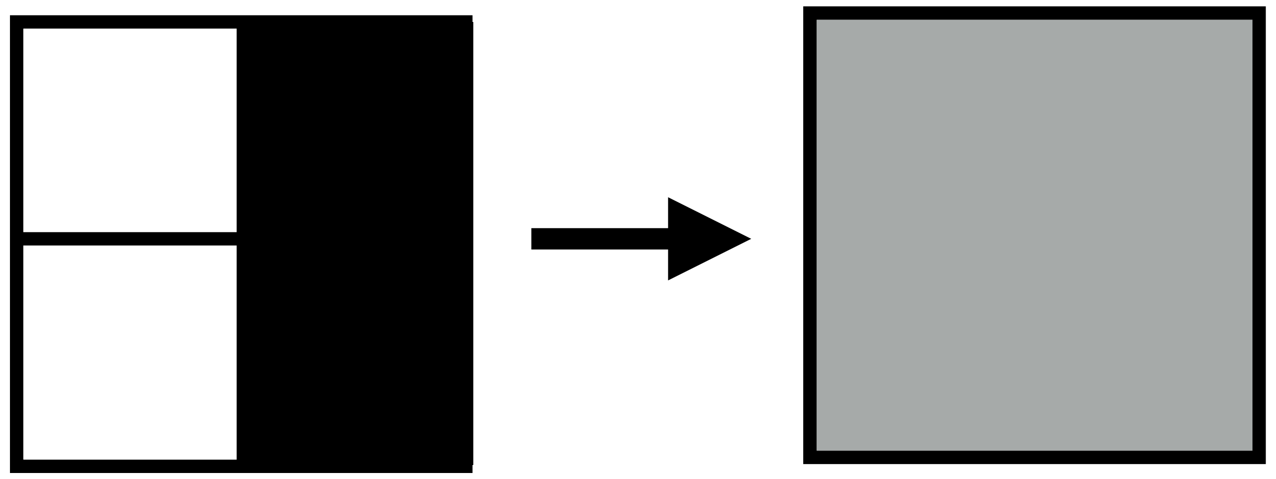
Consider pre-filtering a texture with an alpha matte

Desired filtered result



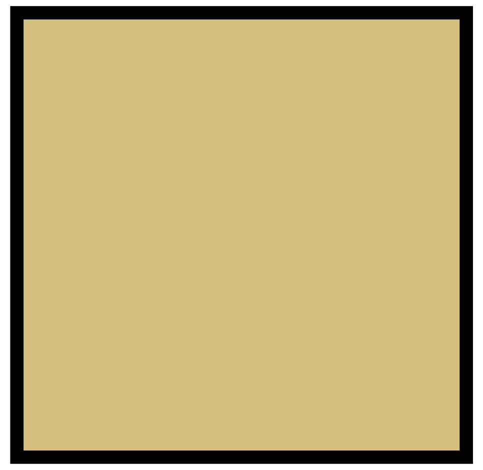
input color

filtered color



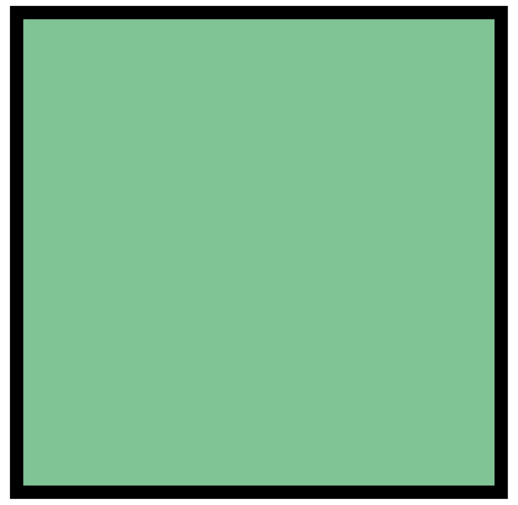
input α

filtered α



filtered result
(composited over white)

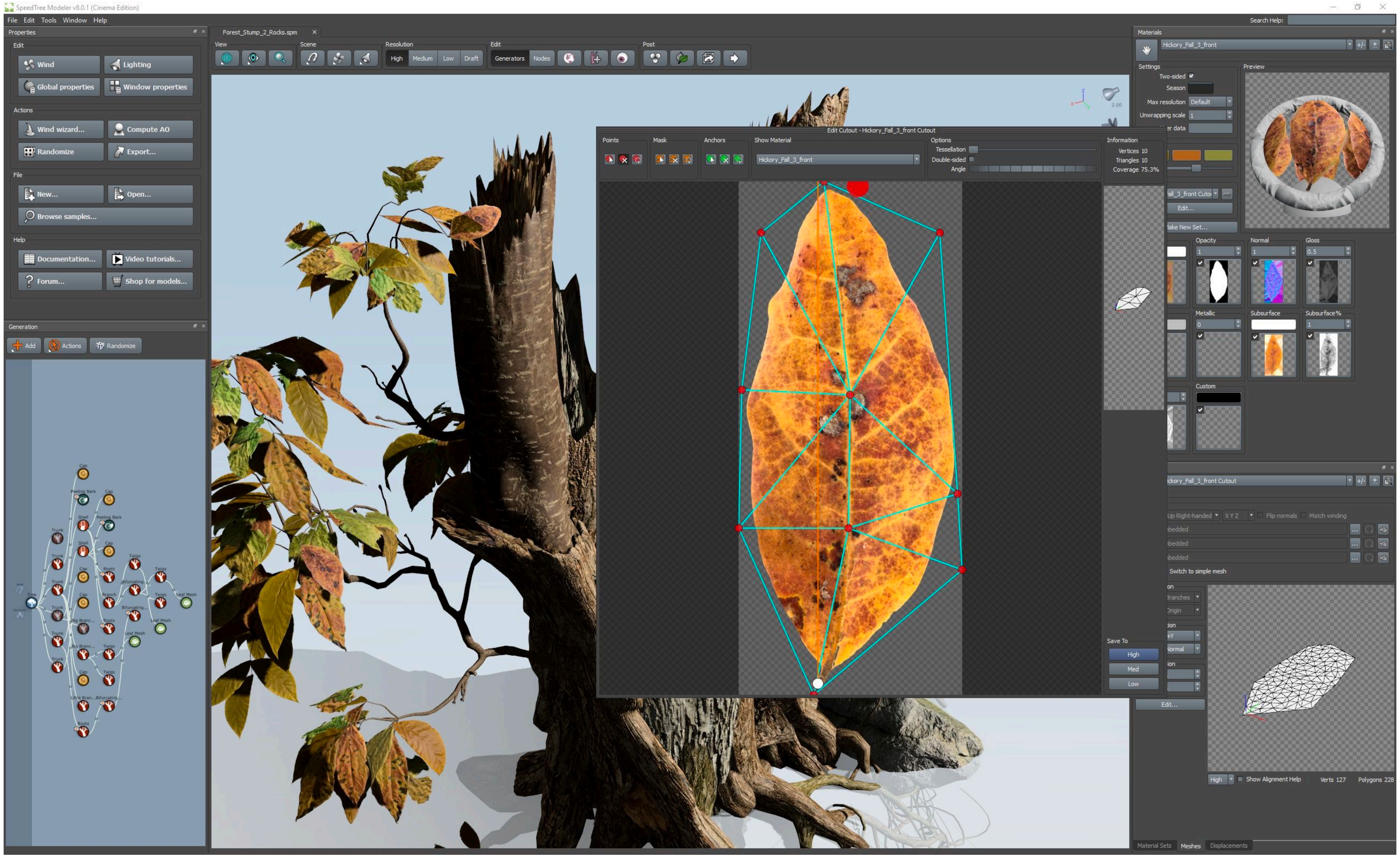
Downsampling non-premultiplied alpha image results in 50% opaque brown (incorrect!)



Result of filtering premultiplied alpha image (correct!)

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

Common use of textures with alpha: foliage



[Image credit: SpeedTree Cinema 8]

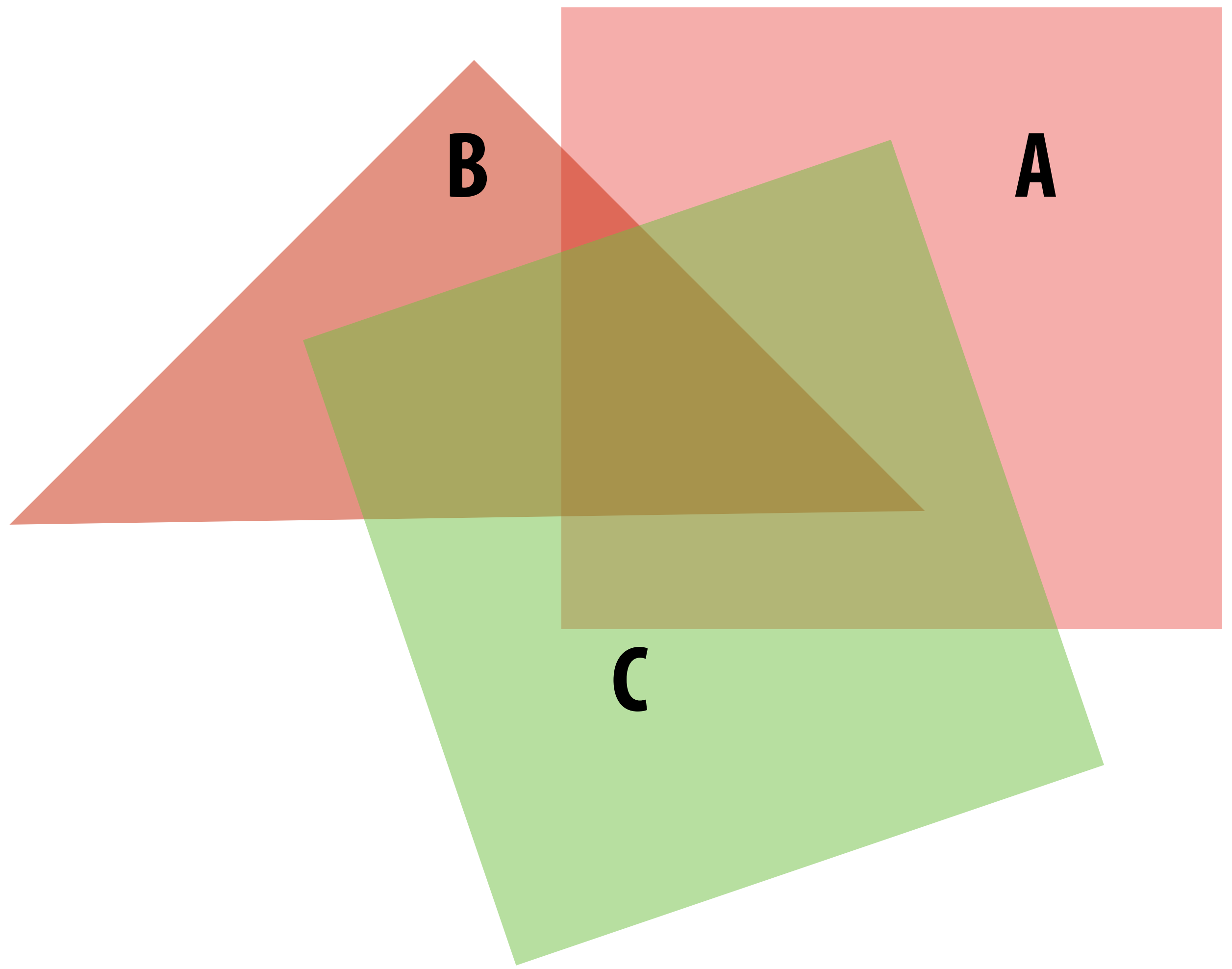
Foliage example

[Image credit: SpeedTree Cinema 8]

Another problem: applying "over" repeatedly

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

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



C over B over A

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

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

Wait… this result is the premultiplied color!

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

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

```
over(C, over(B, A))
```

There is a closed form for non-premultiplied alpha:

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

Summary: advantages of premultiplied alpha

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

Color buffer update: semi-transparent surfaces

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

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

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

What is the assumption made by this implementation?

Triangles must be rendered in back to front order!

What if triangles are rendered in front to back order?

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

Putting it all together *

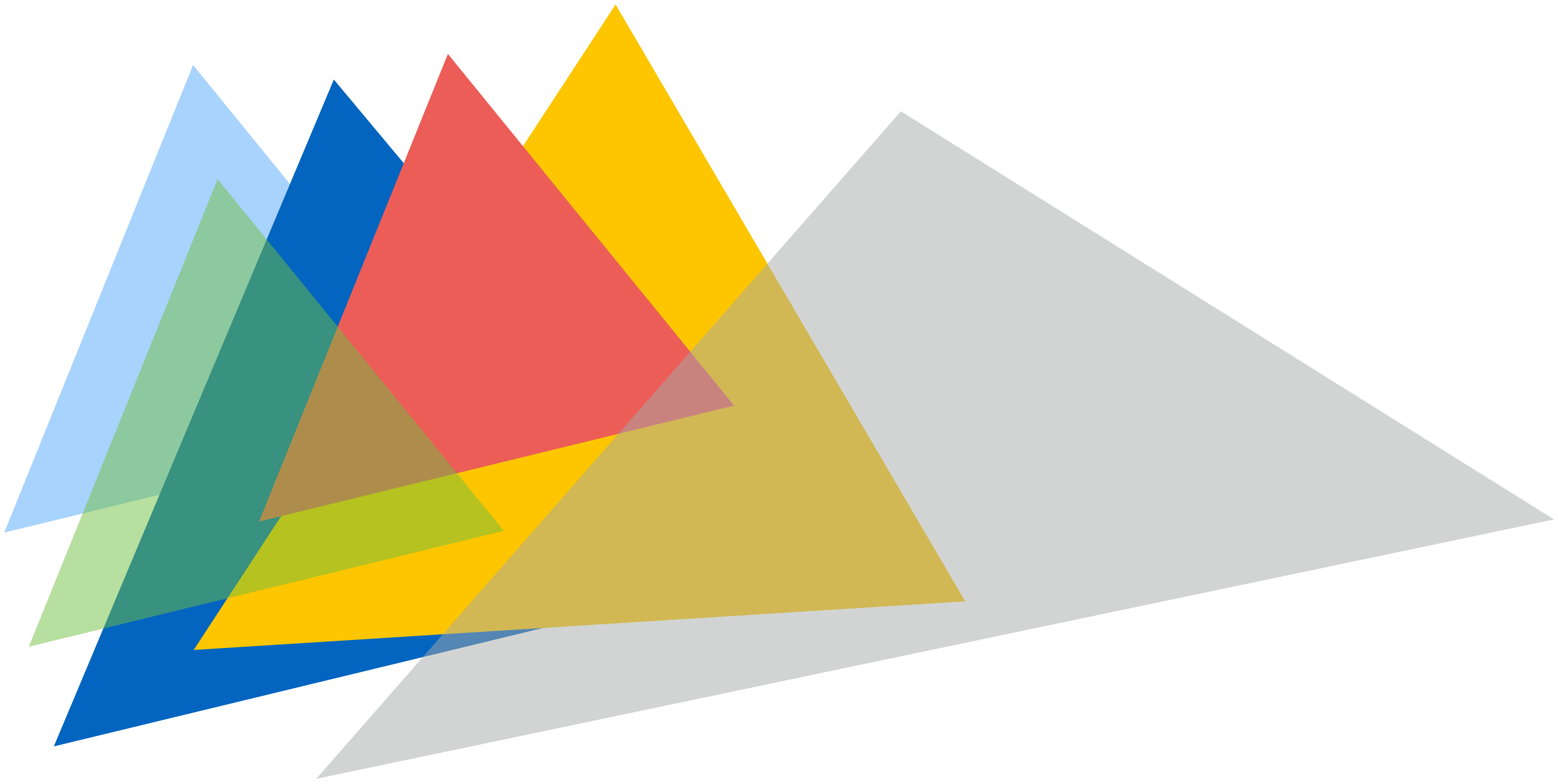
Consider rendering a mixture of opaque and transparent triangles

Step 1: render opaque surfaces using depth-buffered occlusion

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

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

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



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

Combining opaque and semi-transparent triangles

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

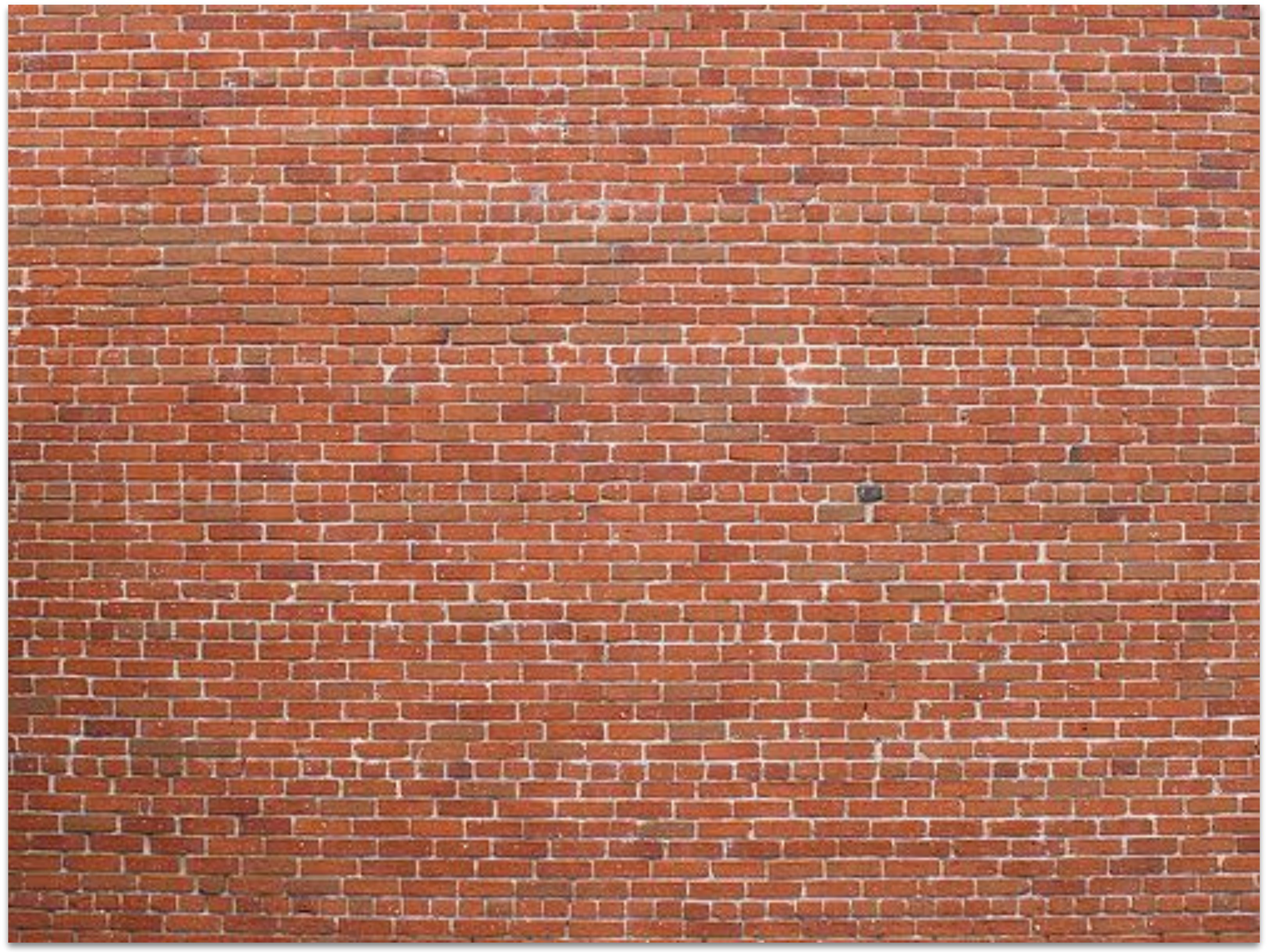
```
// phase 1: render opaque surfaces
update_color_buffer(tri_z, tri_color, x, y) {
   if (pass_depth_test(tri_z, zbuffer[x][y]) {
      color[x][y] = tri_color;
      zbuffer[x][y] = tri_z;
   }
}


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

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

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

Command: draw these triangles!

Inputs:

```
list_of_positions = {          list_of_texcoords = {

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



Texture map

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

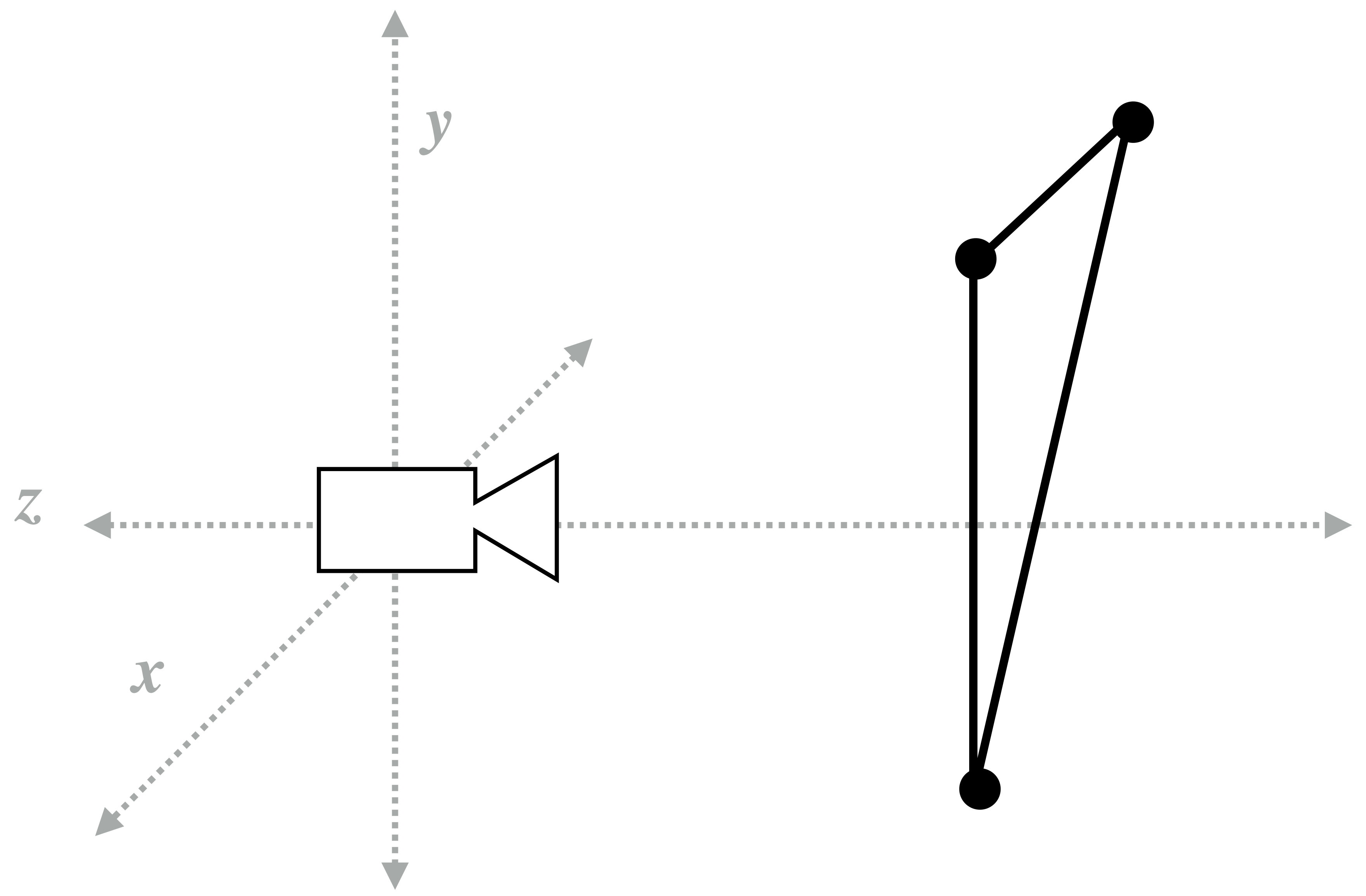
Perspective projection transform $\mathbf{P}$

Size of output image (W, H)

Use depth test /update depth buffer: YES!

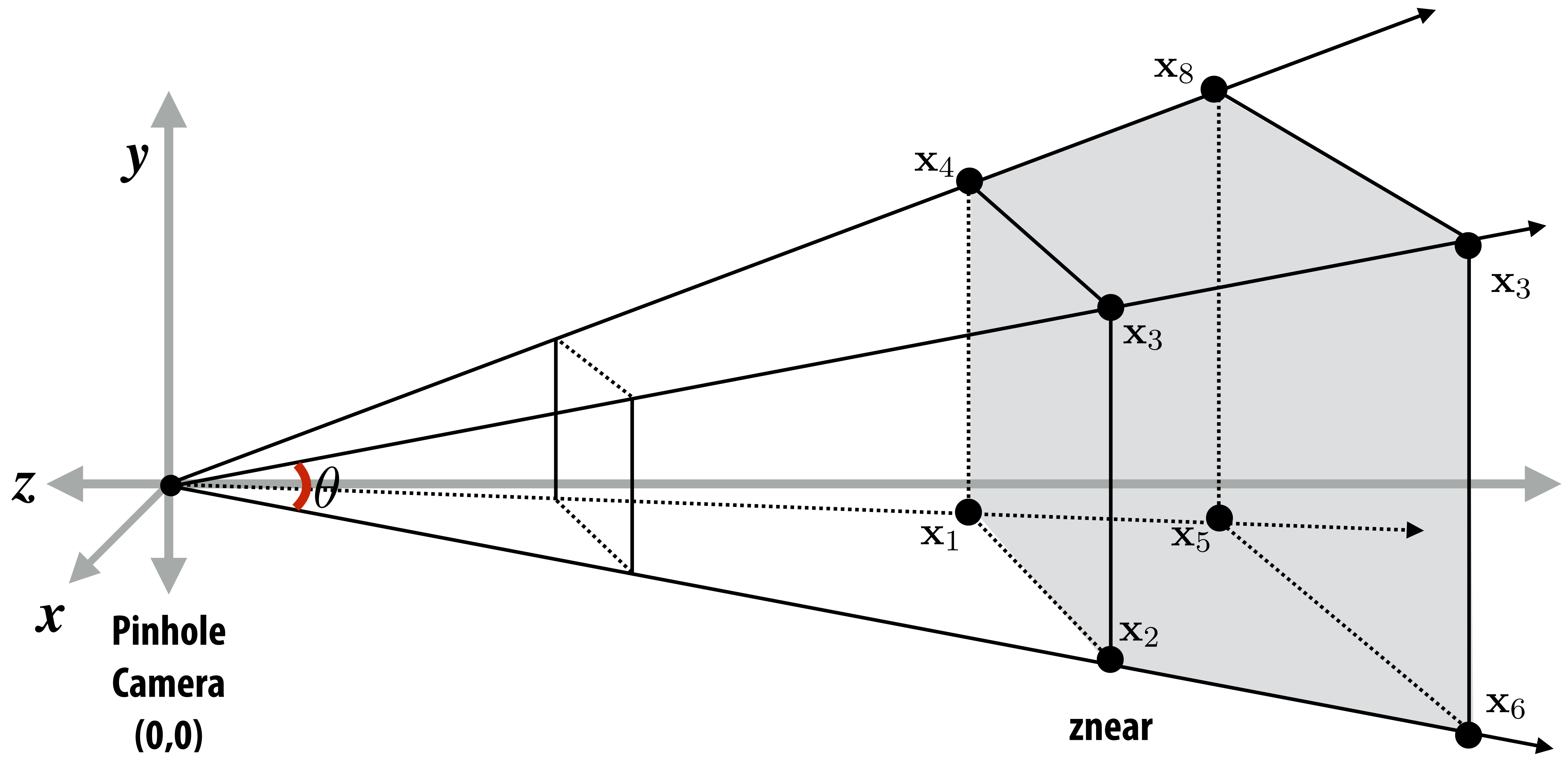
Step 1:

Transform triangle vertices into camera space
(apply modeling and camera transform)

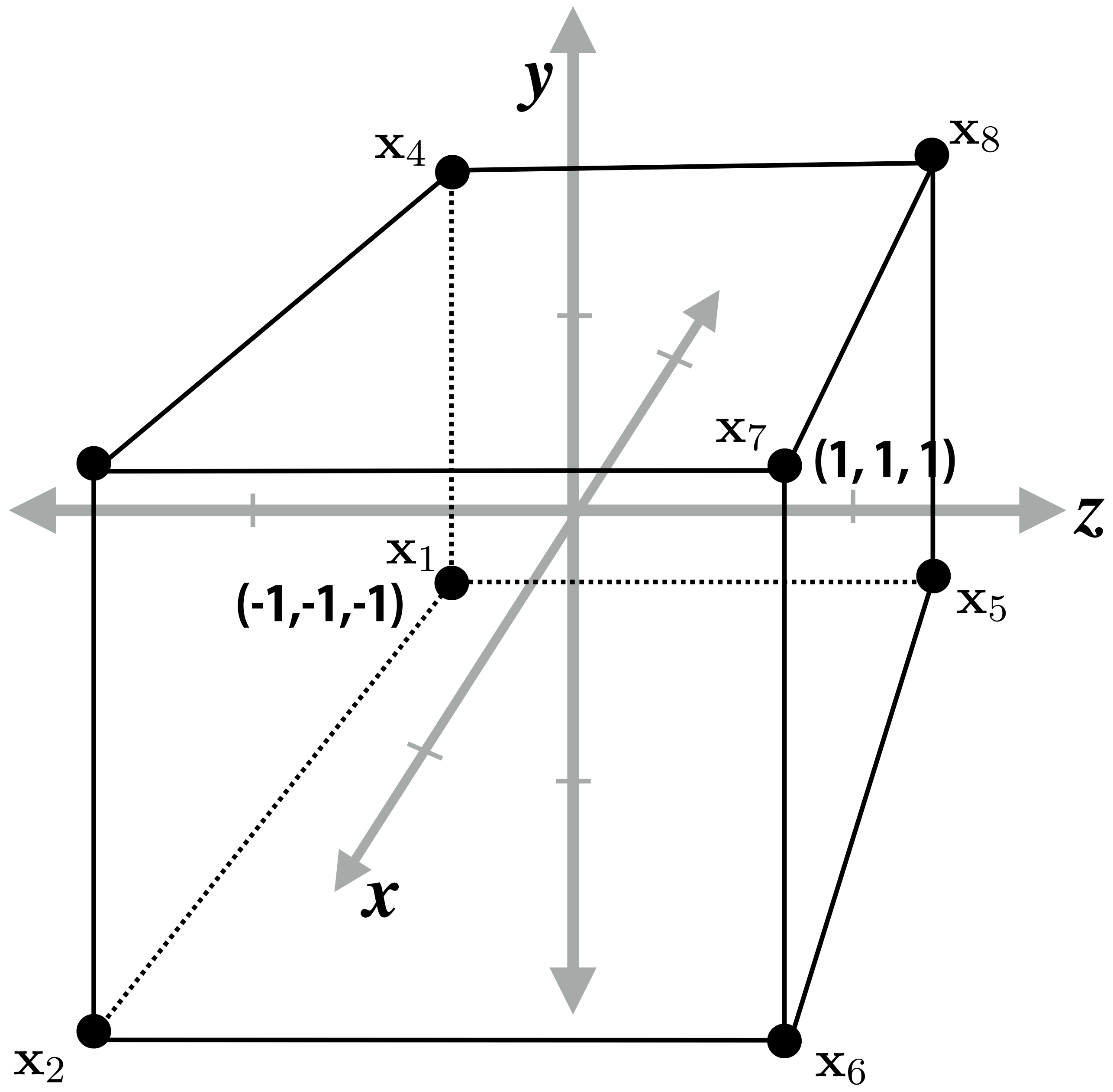


Step 2:

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



Camera-space positions: 3D



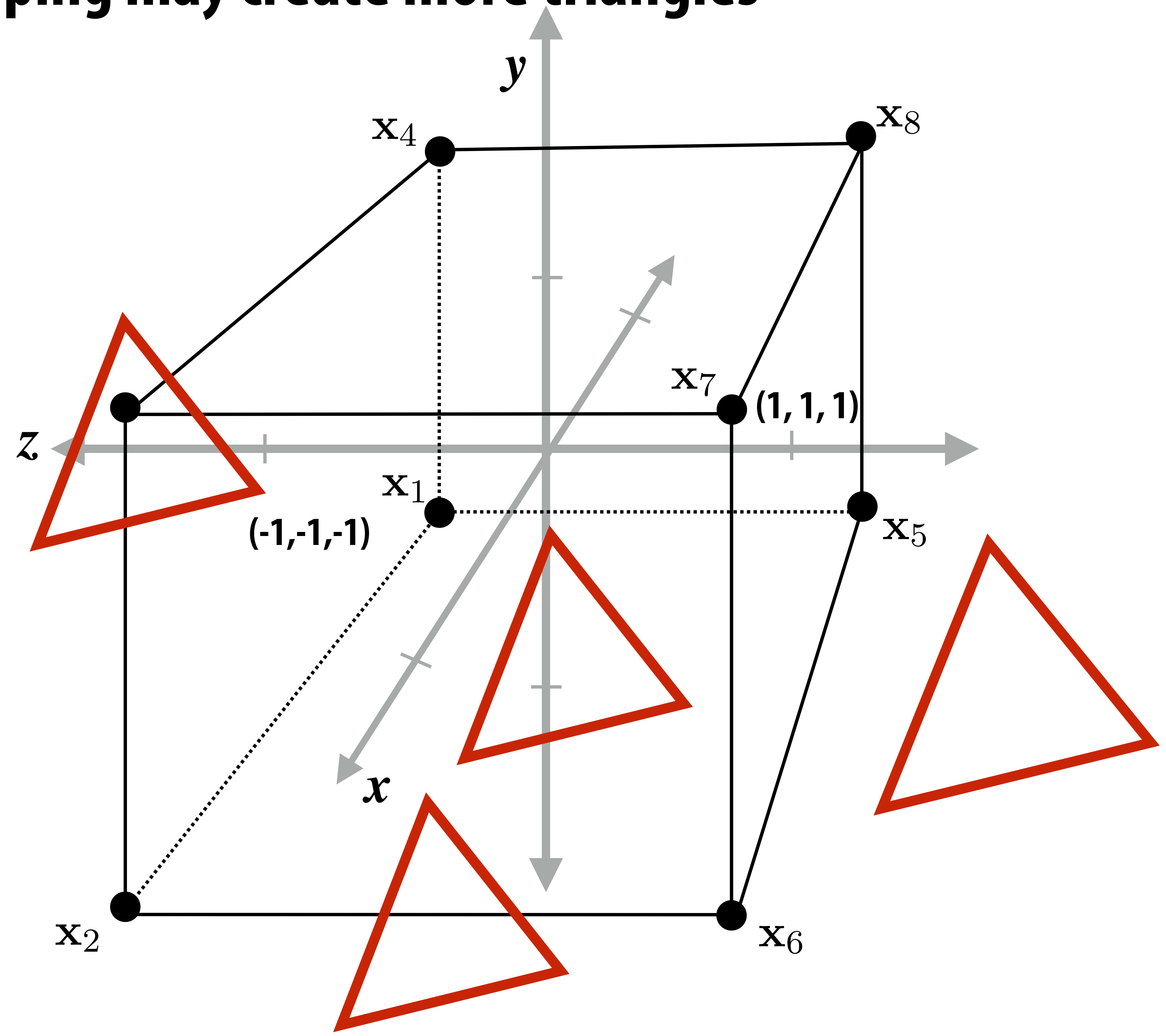
Normalized space positions

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

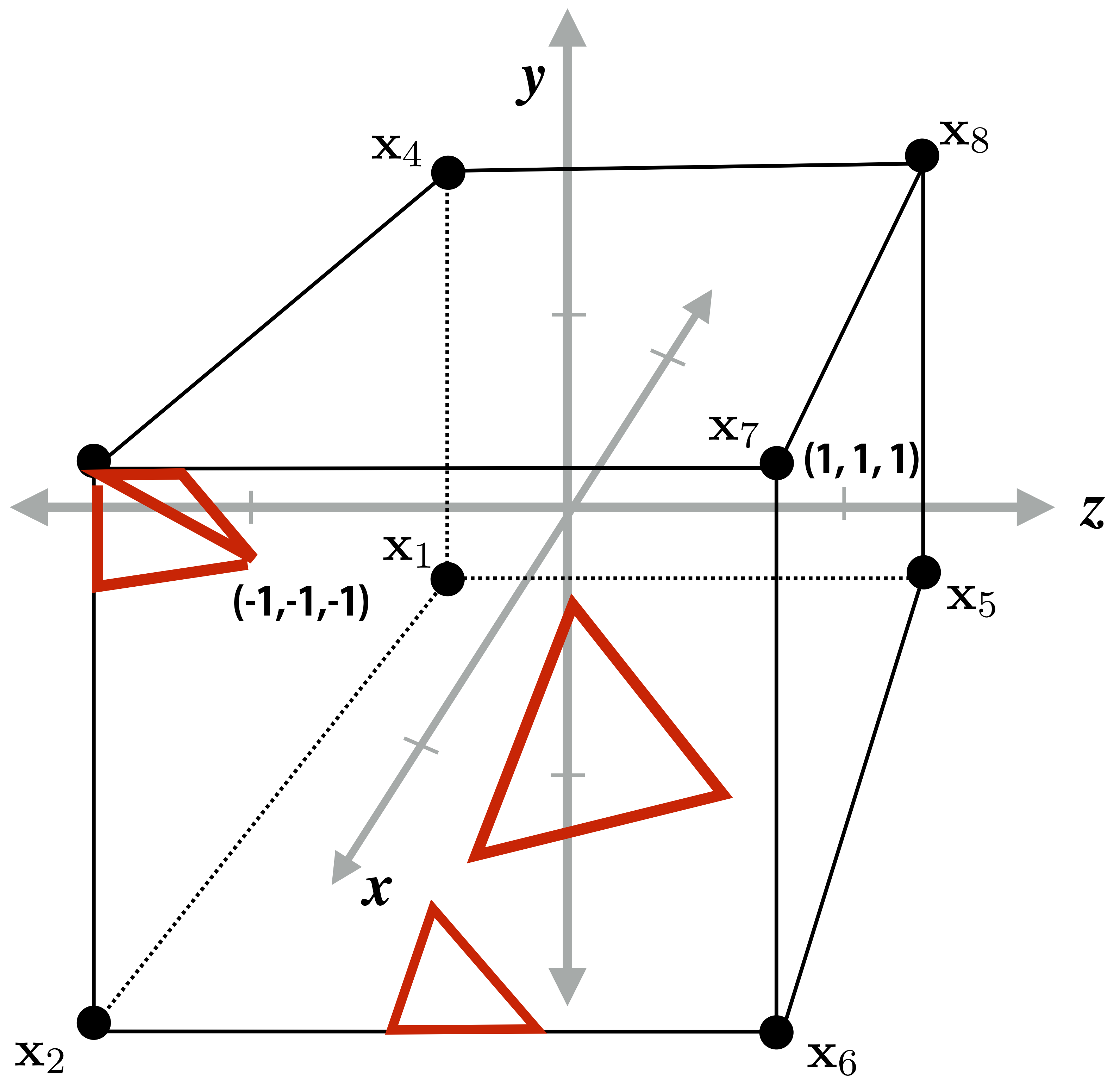
(-w, -w, -w, w) and (w, w, w, w)

Step 3: clipping

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



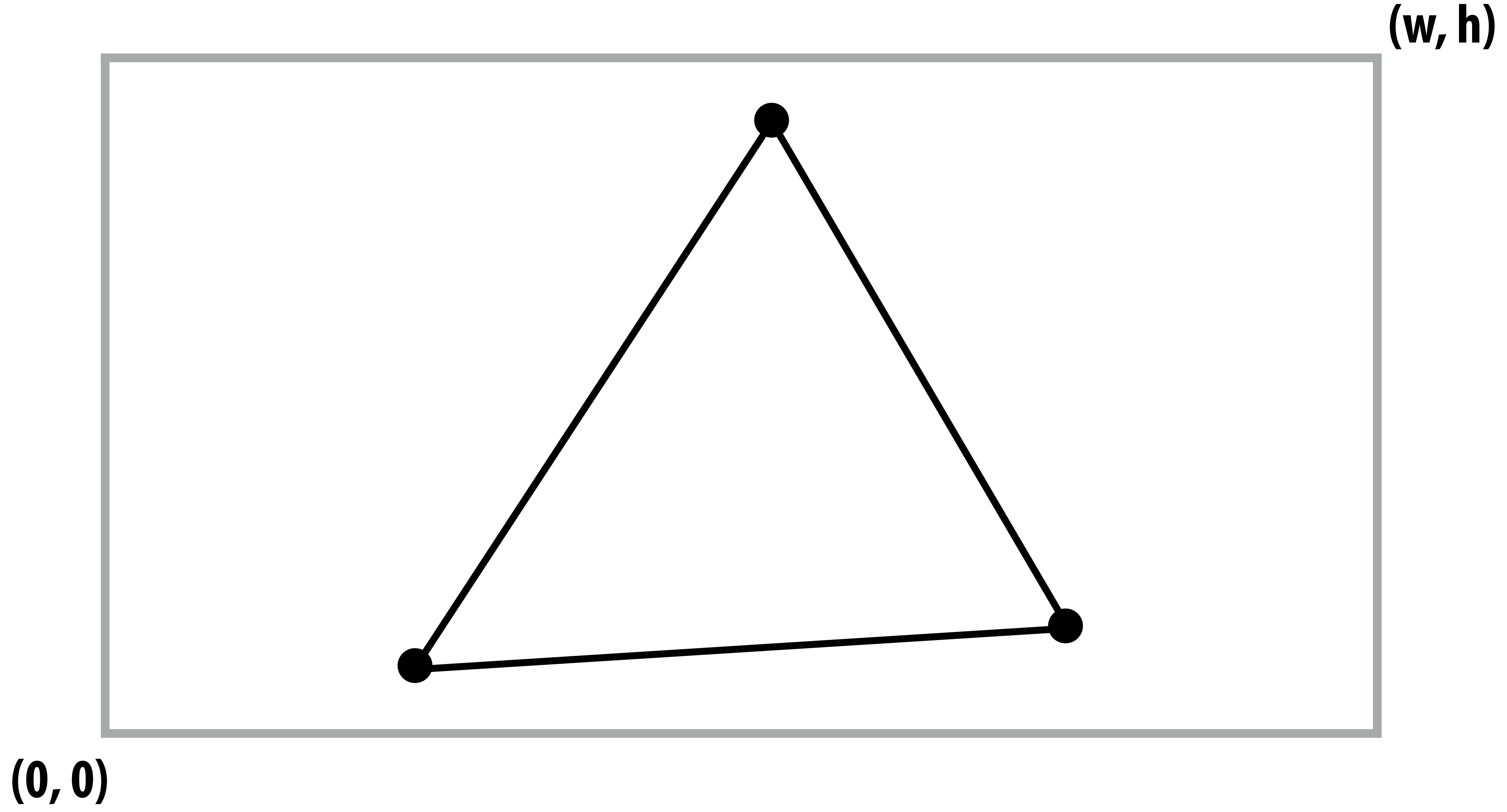
Triangles before clipping



Triangles after clipping

Step 4: transform to screen coordinates

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



Step 5: setup triangle (triangle preprocessing)

Compute triangle edge equations

Compute triangle attribute interpolation equations

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

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

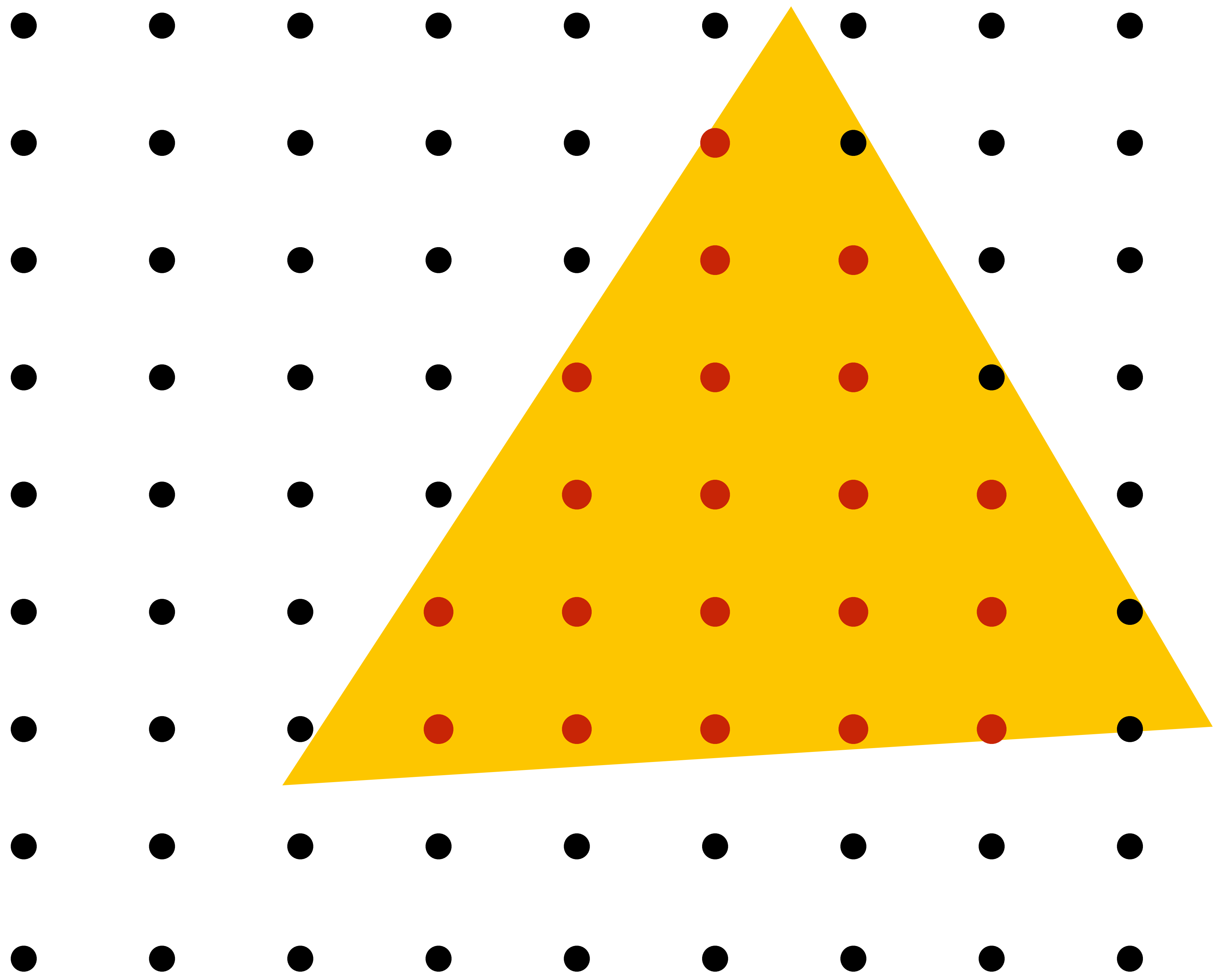
$$\mathbf{E}_{20}(x,y)$$

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

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

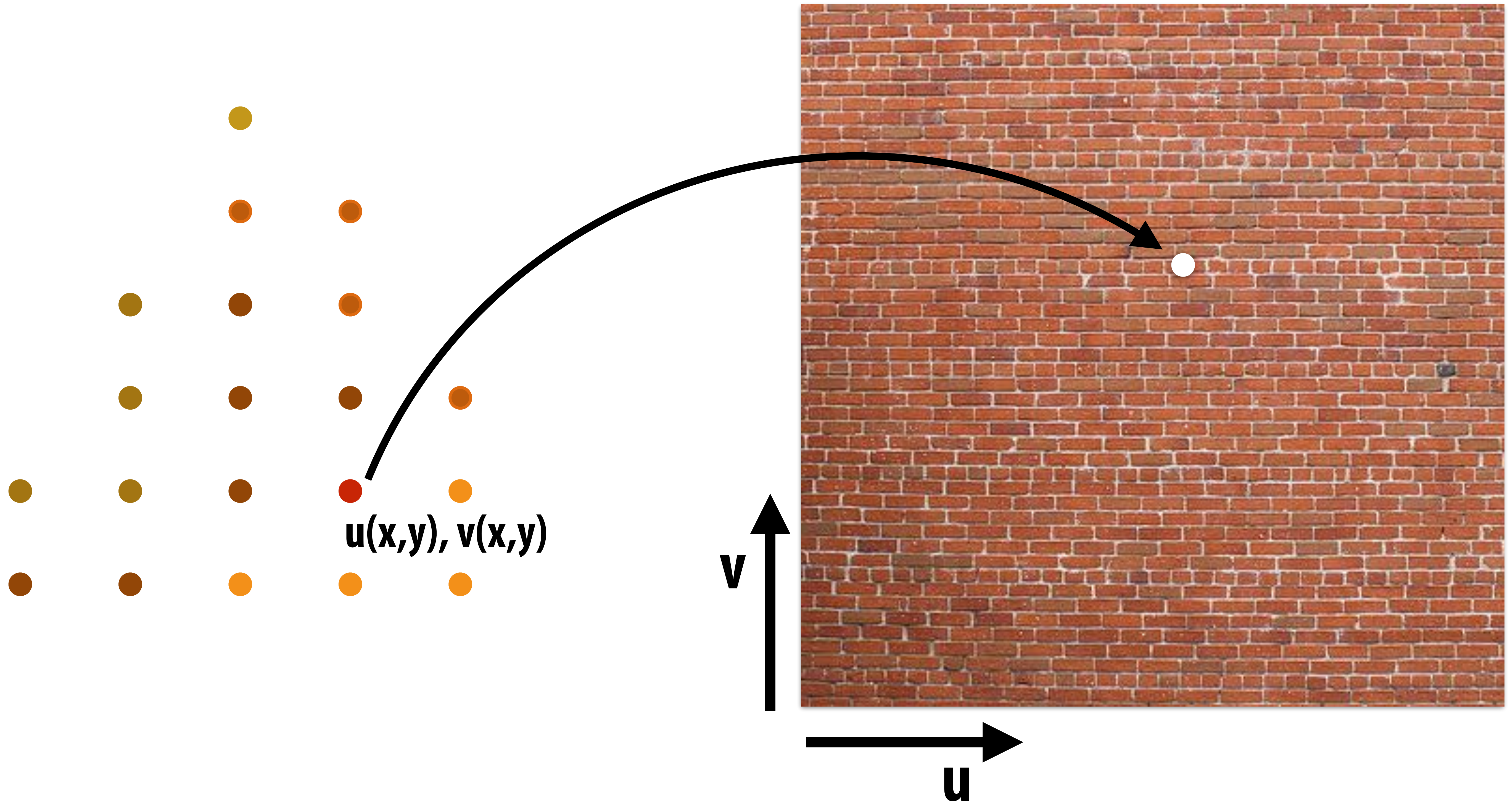
Step 6: sample coverage

Evaluate attributes z, u, v at all covered samples



Step 6: compute triangle color at sample point

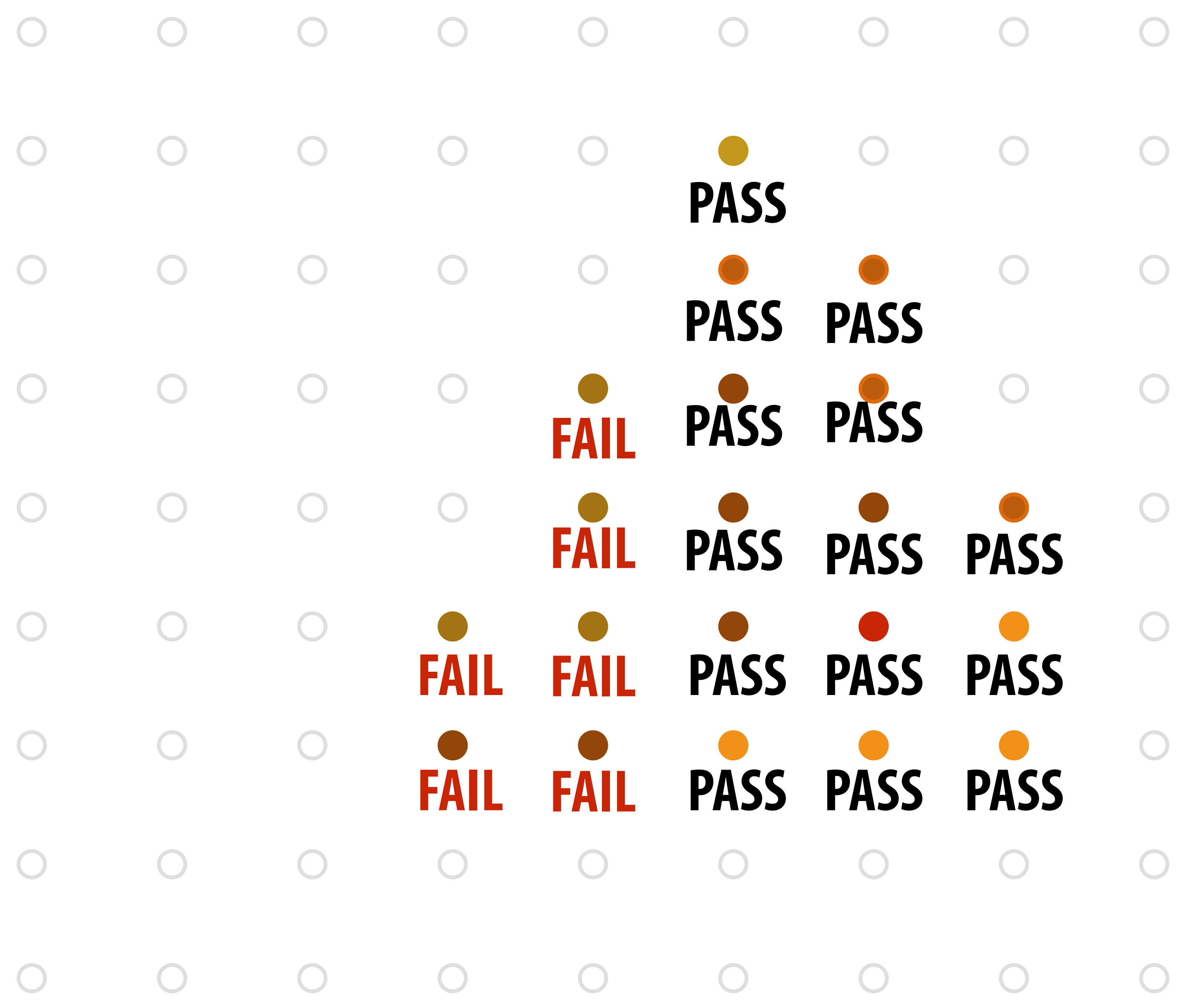
e.g., sample texture map *



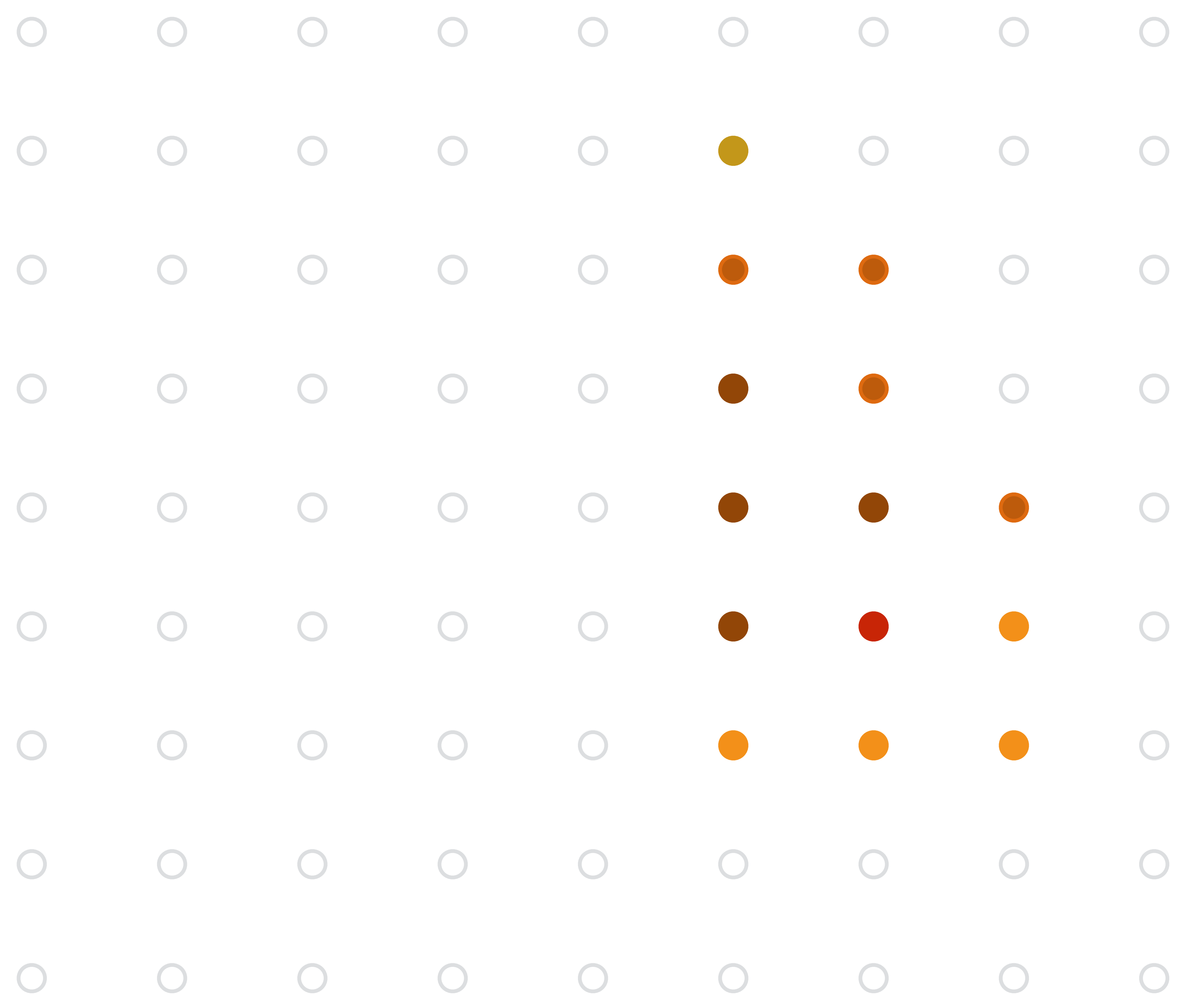
* So far, we've only described computing triangle's color at a point by interpolating per-vertex colors, or by sampling a texture map. Later in the course, we'll discuss more advanced algorithms for computing its color based on material properties and scene lighting conditions.

Step 7: perform depth test (if enabled)

Also update depth value at covered samples (if necessary)



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

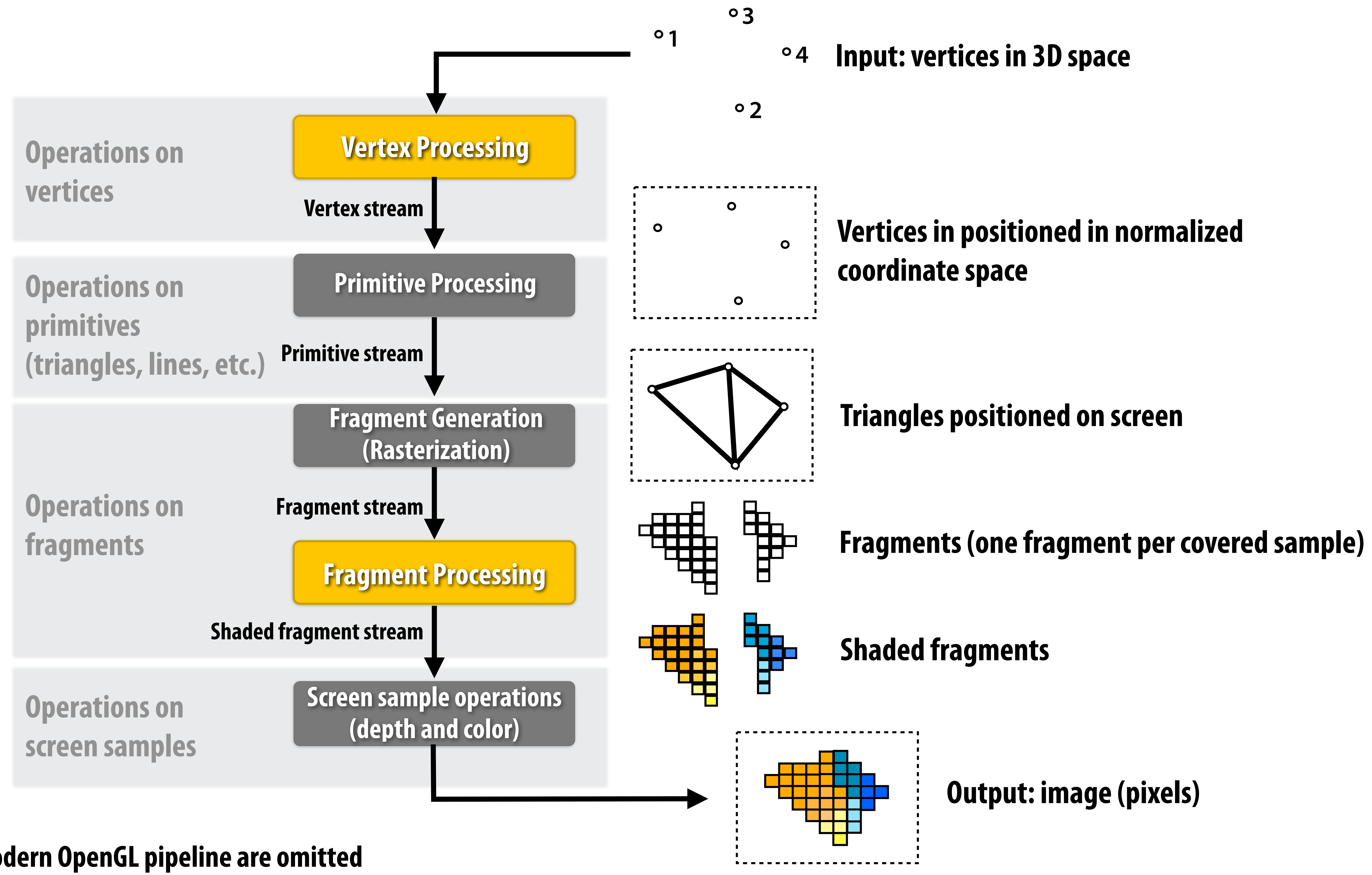
- OpenGL
- Microsoft Direct3D
- Apple Metal

- 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

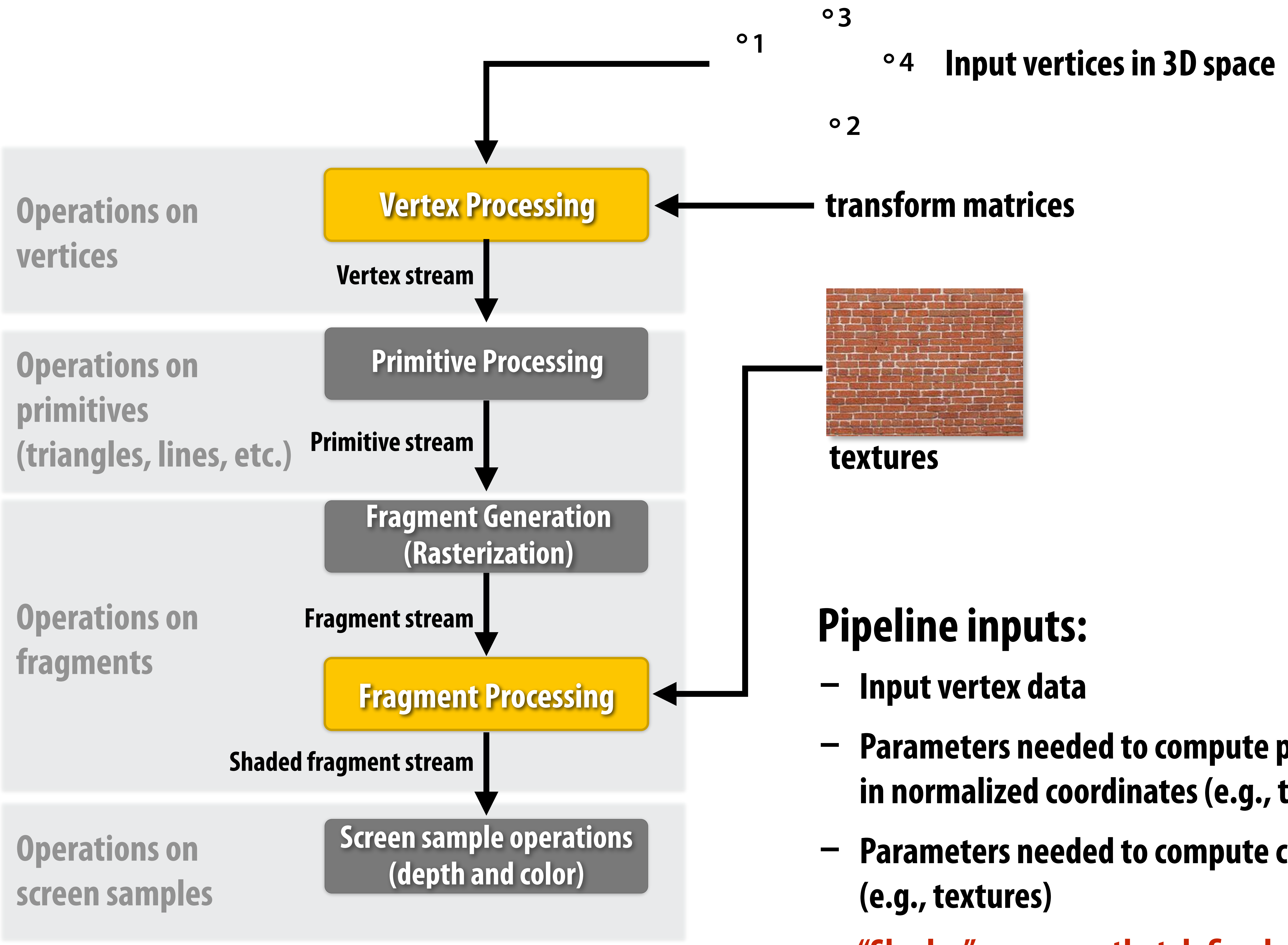
OpenGL/Direct3D graphics pipeline *

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 (uniform sampler2D myTexture; uniform vec3 lightDir;)

Per-fragment attributes (interpolated by rasterizer) (varying vec2 uv; varying vec3 norm;)

Sample surface albedo (reflectance color) from texture

Modulate surface albedo by incident irradiance (incoming light)

Shader outputs surface color

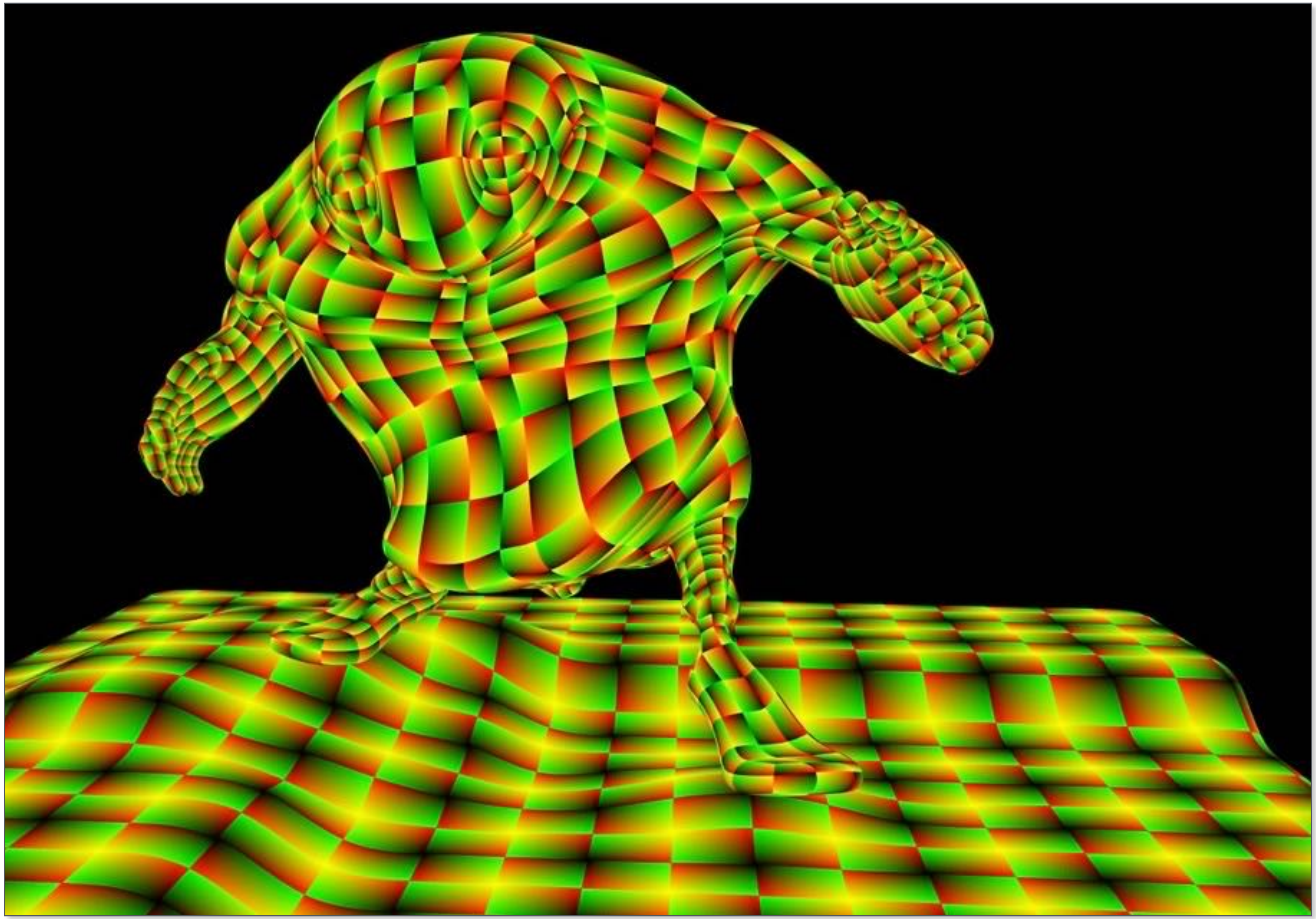
Shader function executes once per fragment.

Outputs color of surface at sample point corresponding to fragment.

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

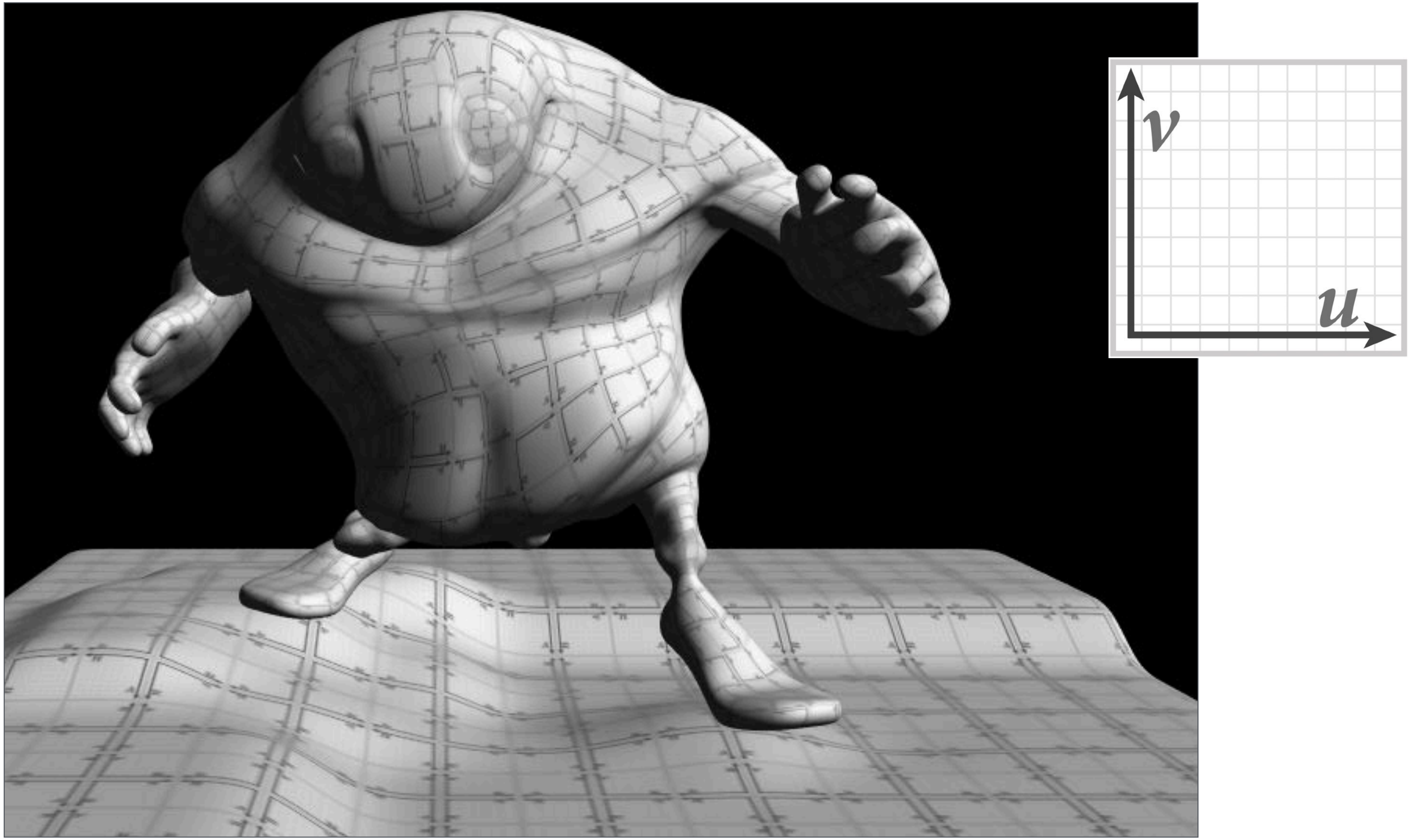
Texture coordinate visualization

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



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

Rendered result (after evaluating fragment shader for each pixel)



Goal: render very high complexity 3D scenes

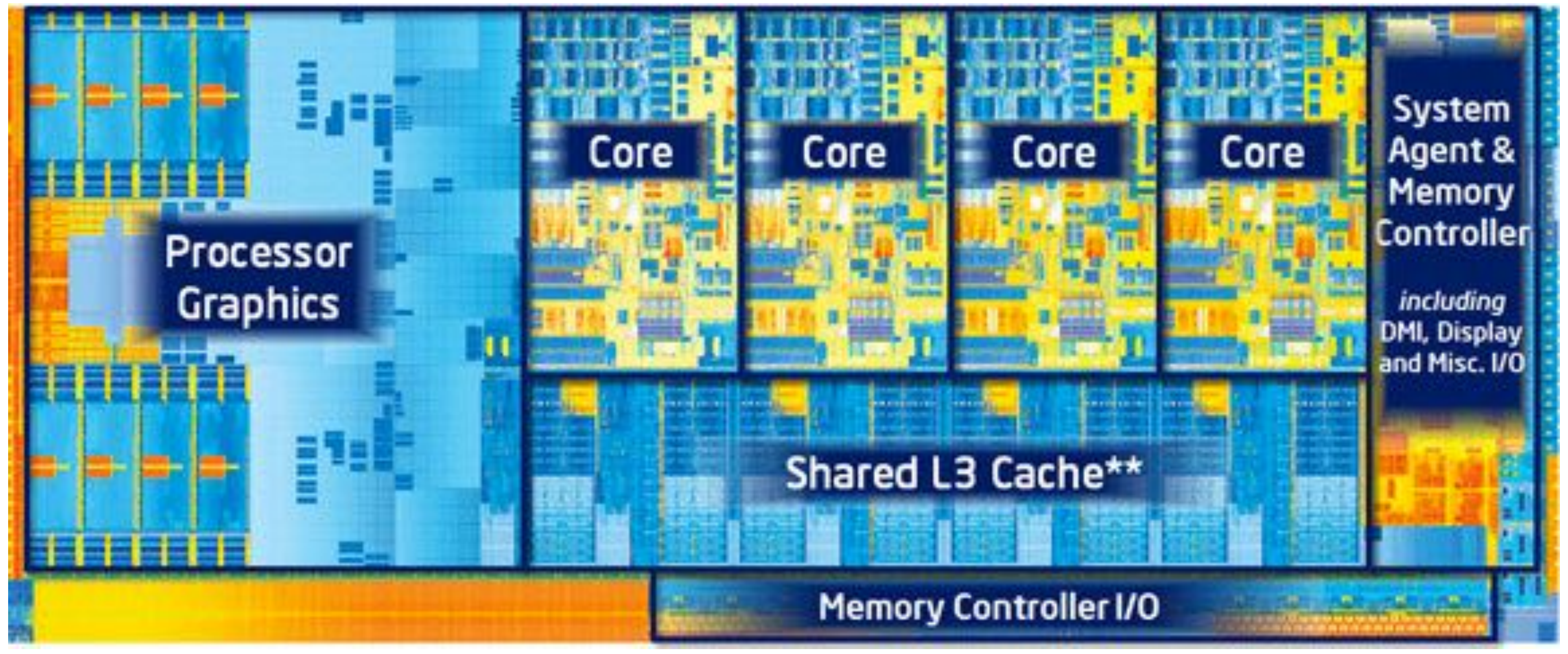
- 100's of thousands to millions of triangles in a scene
- Complex vertex and fragment shader computations
- High resolution screen outputs (2-4 Mpixel + supersampling)
- 30-60 fps

Unreal Engine Kite Demo (Epic Games 2015)

Graphics pipeline implementation: GPUs

Specialized processors for executing graphics pipeline computations

Discrete GPU card
(NVIDIA RTX 3090 GPU)

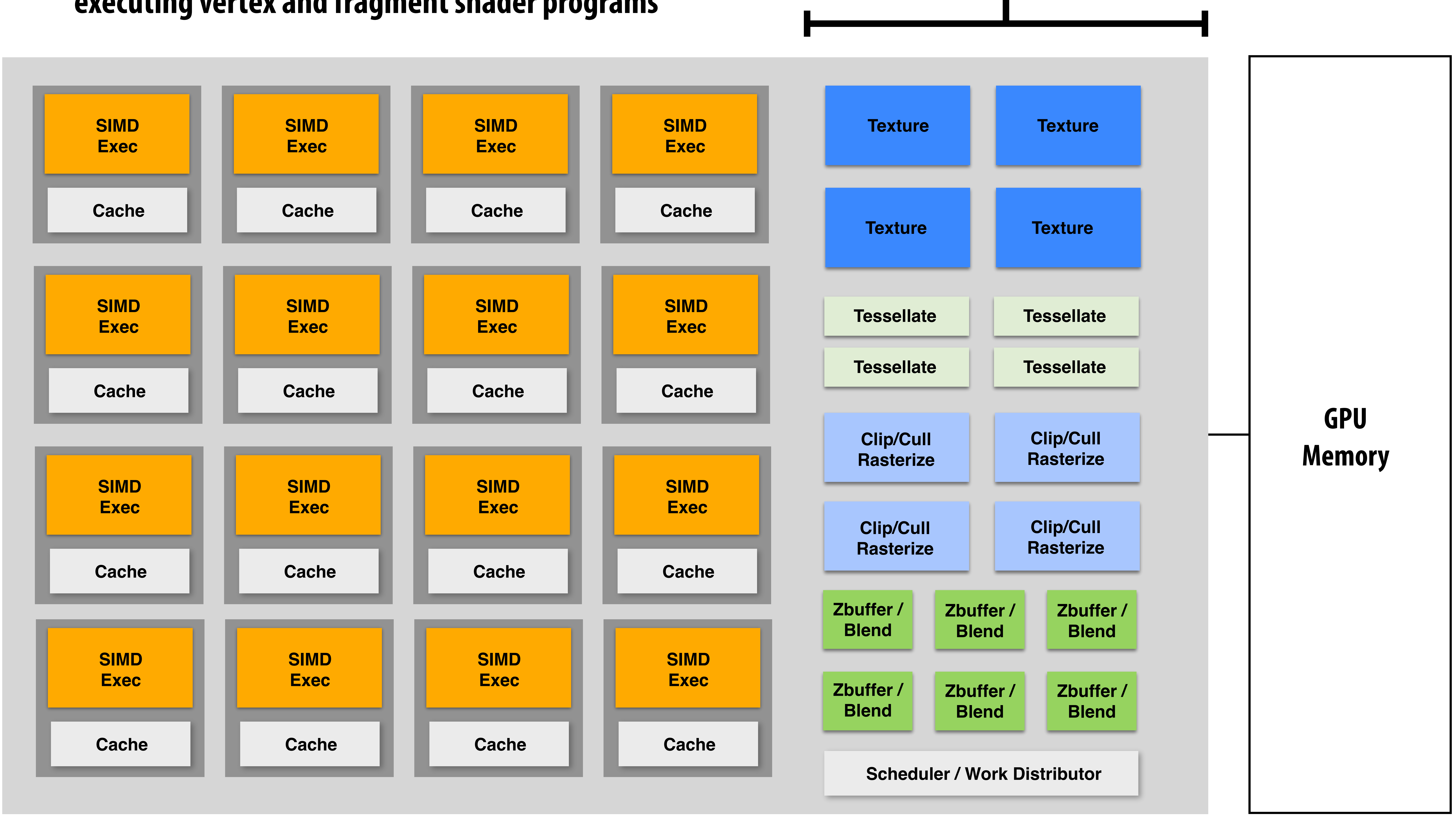


Integrated GPU: part of modern Intel CPU chip

GPU: heterogeneous, multi-core processor

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

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



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

Summary

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

- 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)