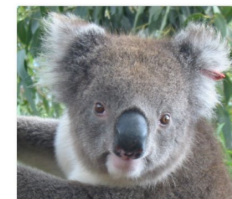
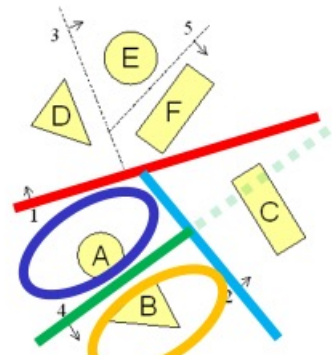
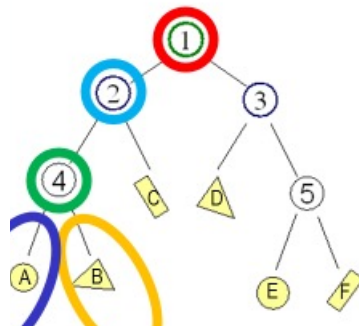
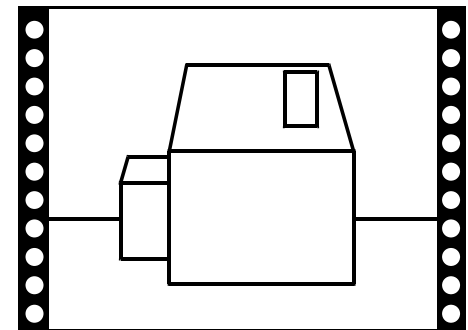
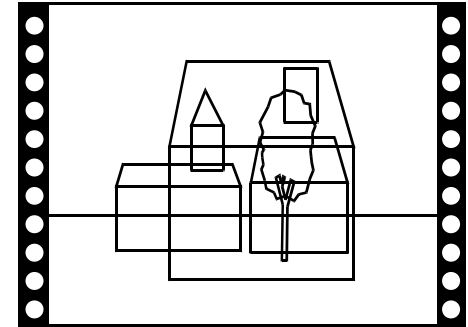


# Visibility - CSEI 60

- Visibility Problem Statement
- Depth Buffer
- Compositing
- Back To Front: Painters Algorithm and BSP Trees
- From Vertices to Frame Buffer
- Administrative
- Q&A





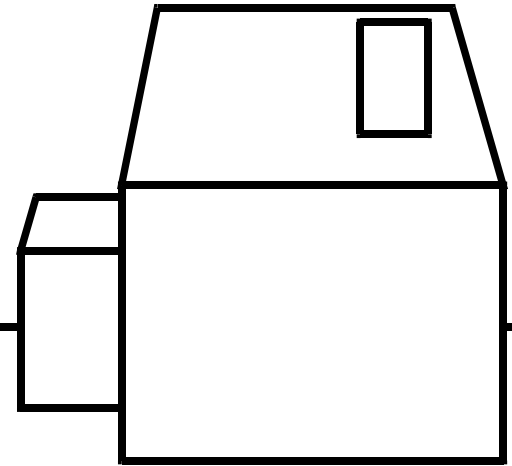
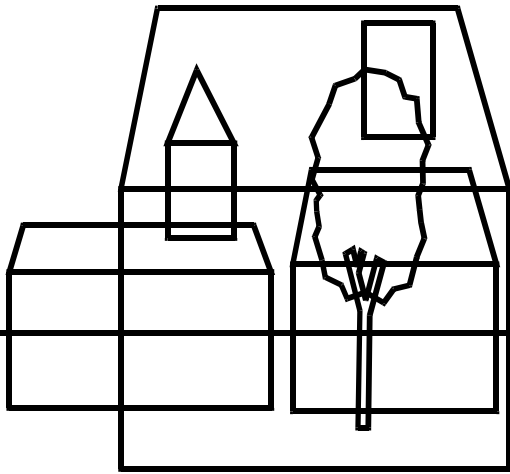
# Visibility Problem Statement



# Visibility

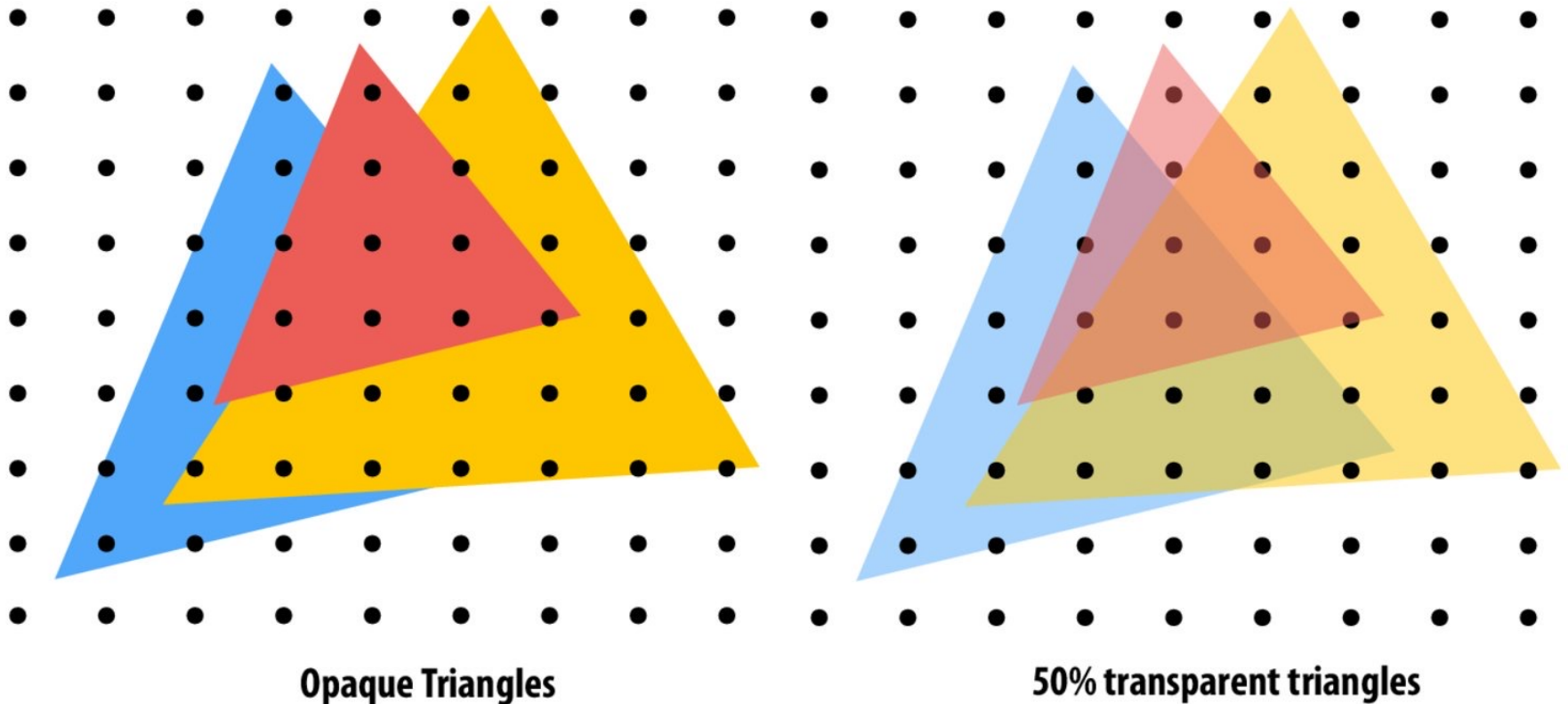
---

- How do we know which parts are visible/in front?





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





# Depth Buffer



# Depth buffer (aka “Z buffer”)

**Color buffer:**

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



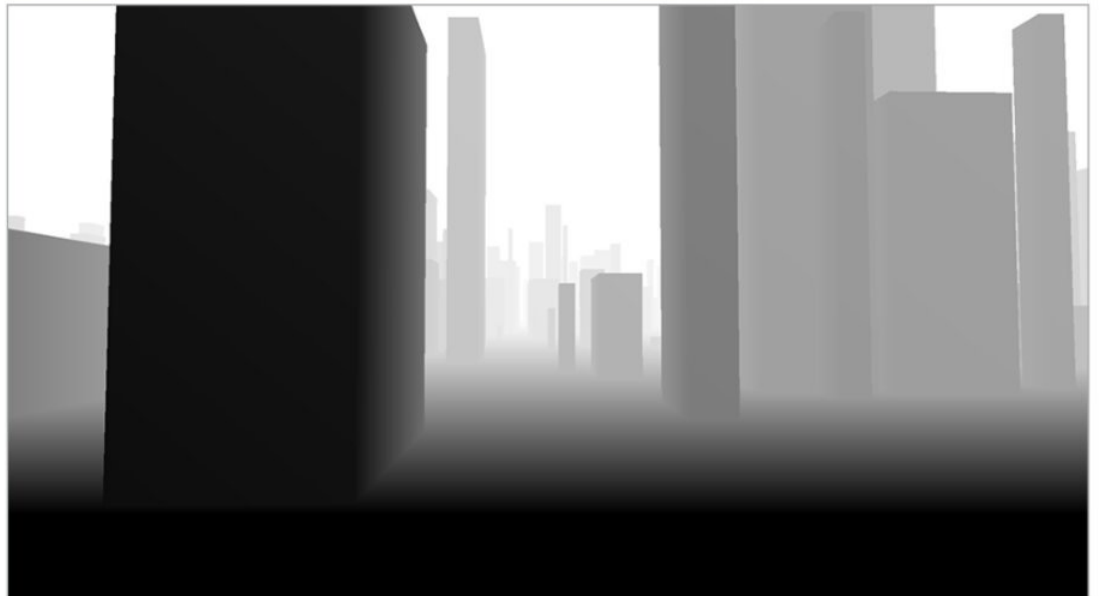
**Depth buffer:**

(stores depth per sample)

Stores depth of closest surface  
drawn so far

black = close depth

white = far depth





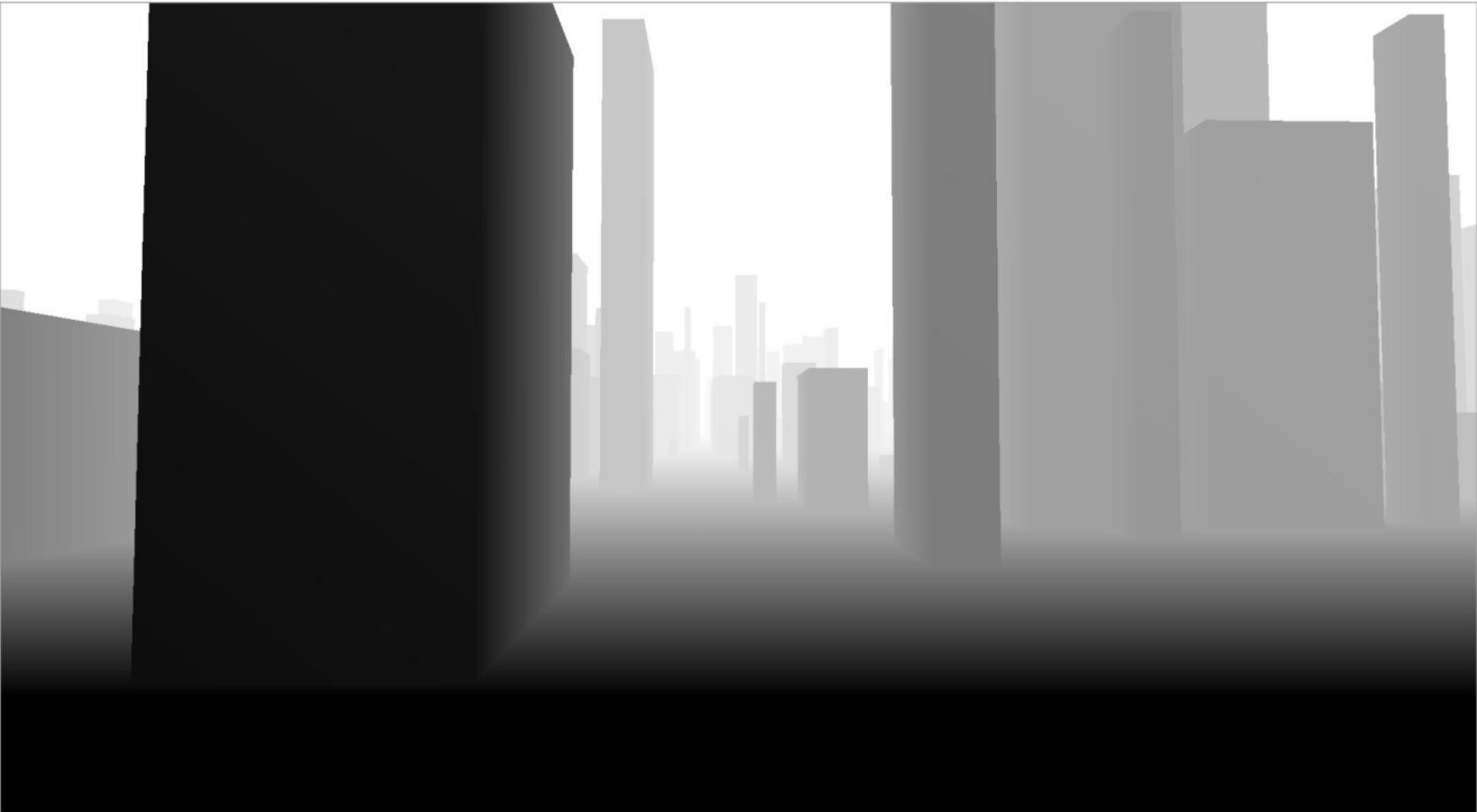
# Depth buffer (a better look)



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



# Depth buffer (a better look)



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

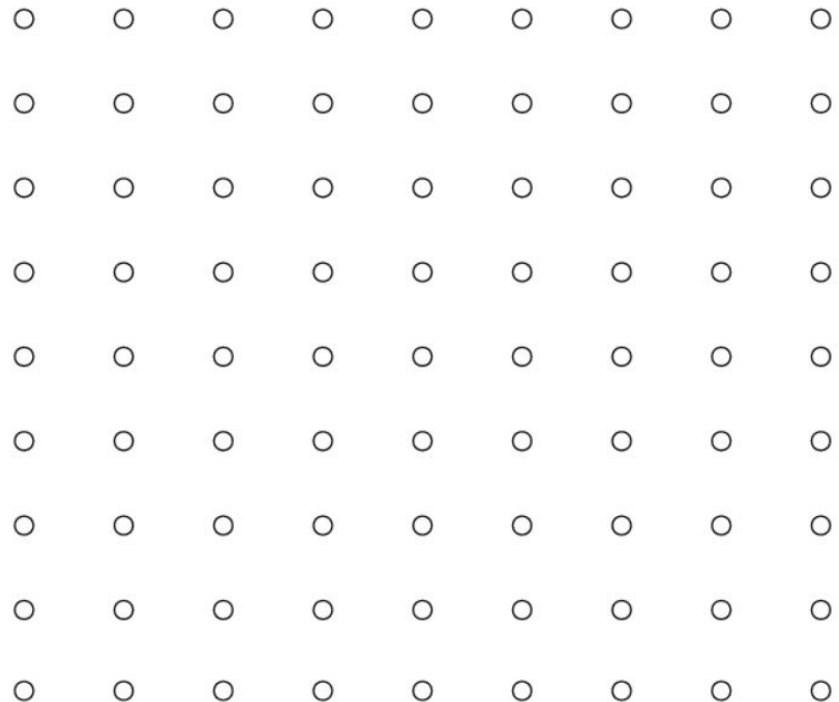


# Occlusion using the depth-buffer (“Z-buffer”)

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

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

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



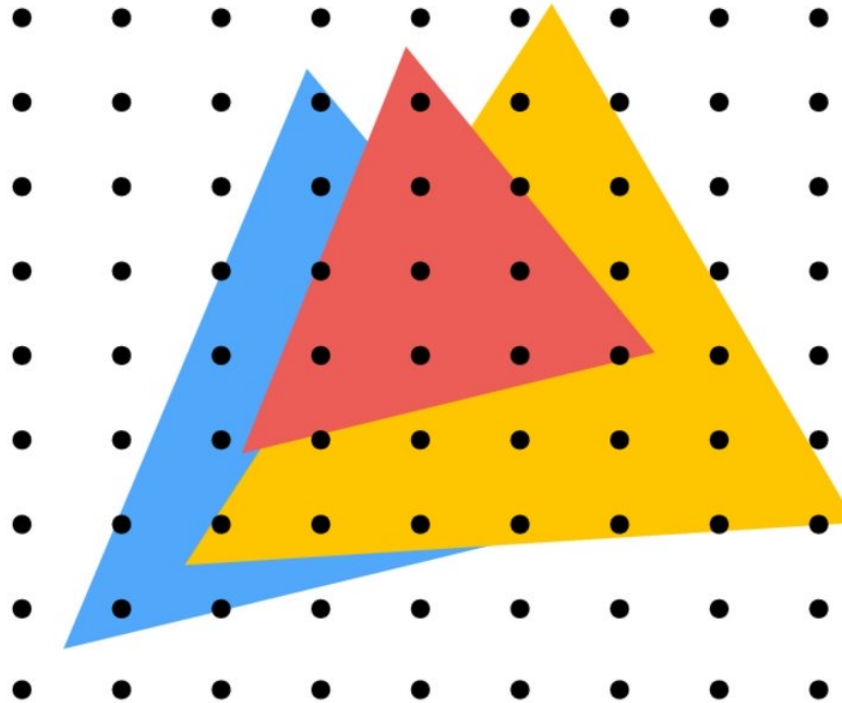
Grayscale value of sample point  
used to indicate distance

Black = small distance

White = large distance



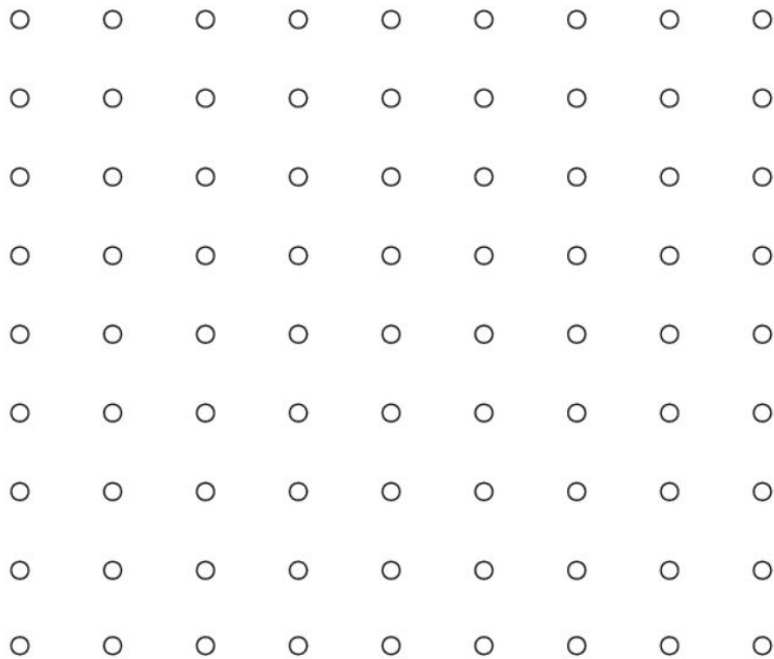
# Example: rendering three opaque triangles





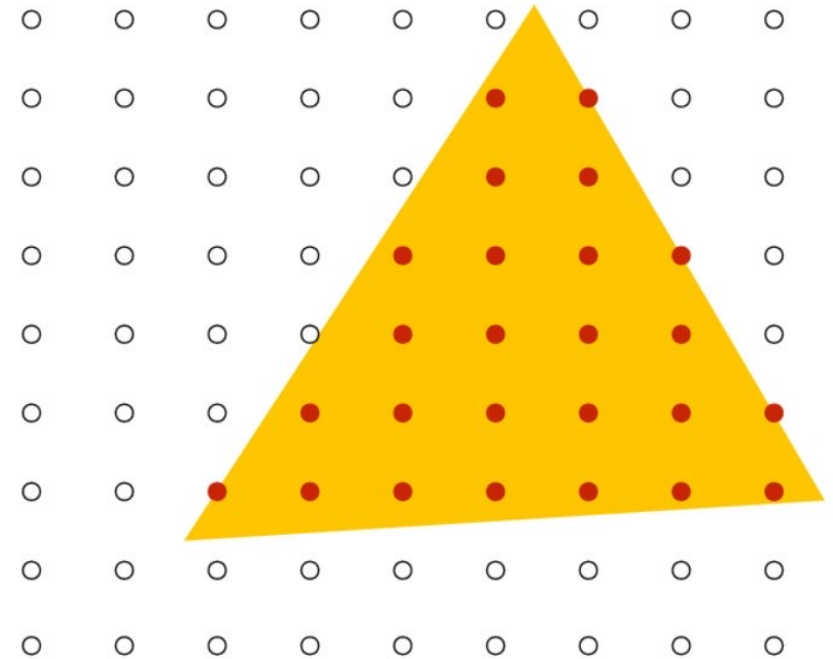
# Occlusion using the depth-buffer (Z-buffer)

Processing yellow triangle:  
depth = 0.5



Color buffer contents

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

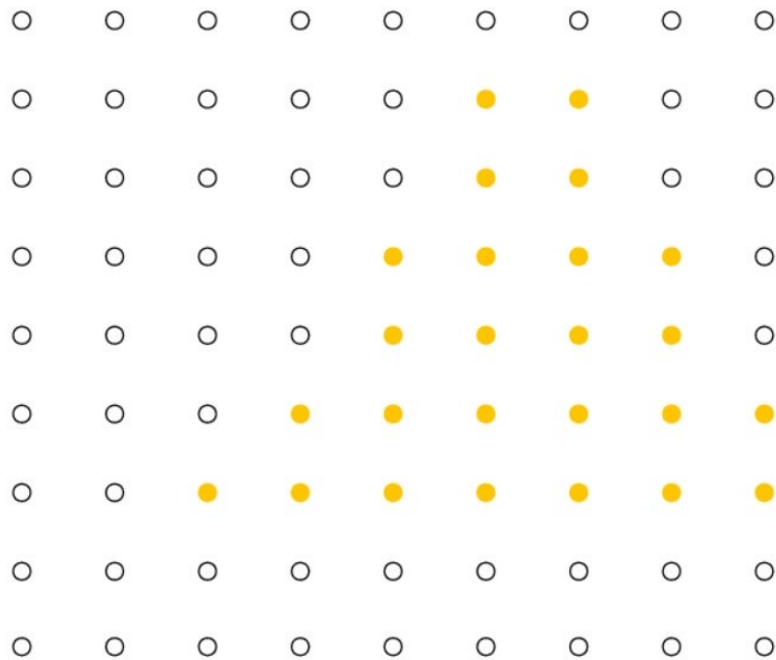


Depth buffer contents



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

After processing yellow triangle:



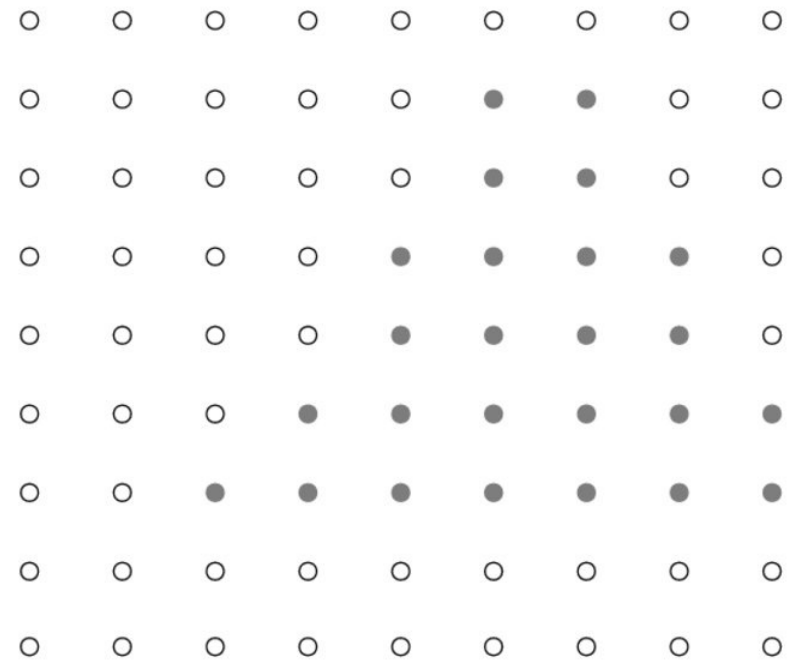
Color buffer contents

Grayscale value of sample point  
used to indicate distance

White = large distance

Black = small distance

Red = samples that pass depth test

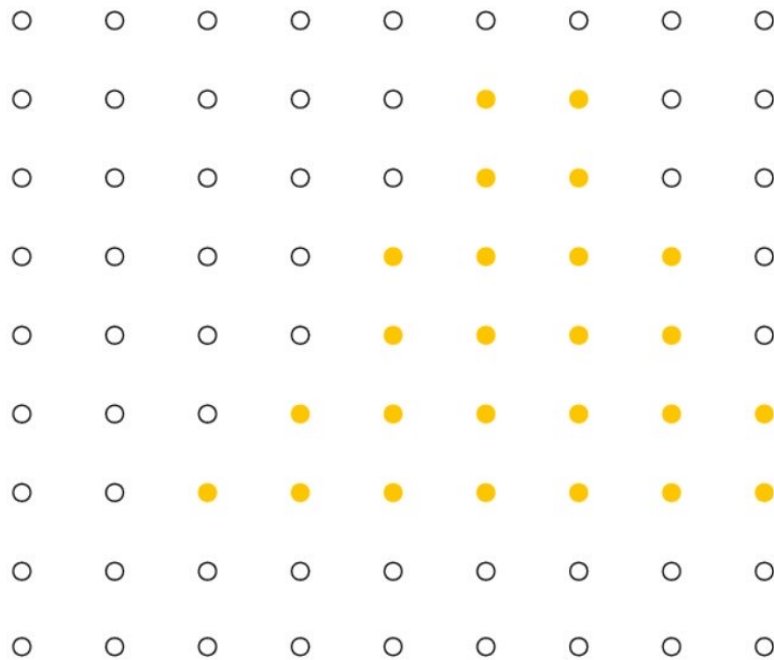


Depth buffer contents



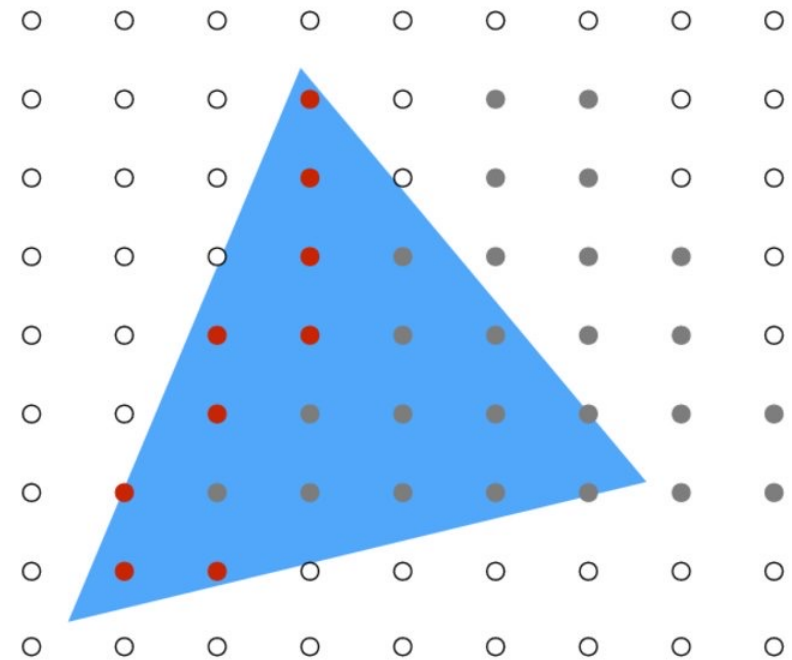
# Occlusion using the depth-buffer (Z-buffer)

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

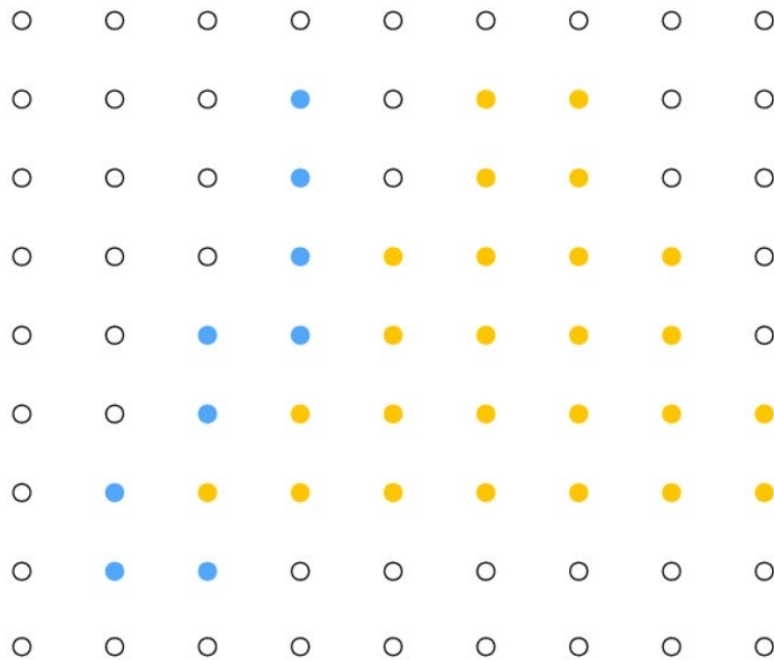


Depth buffer contents



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

After processing blue triangle:



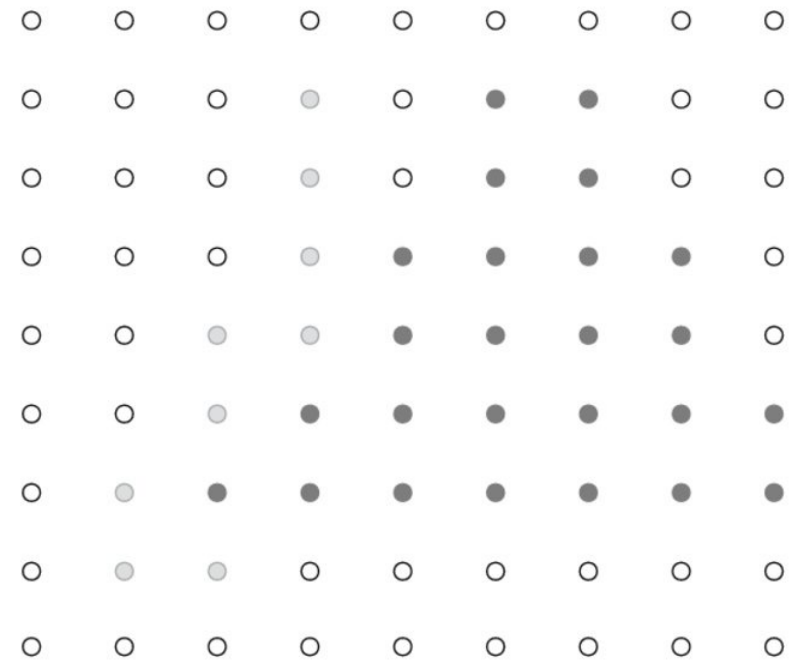
Color buffer contents

Grayscale value of sample point  
used to indicate distance

White = large distance

Black = small distance

Red = samples that pass depth test

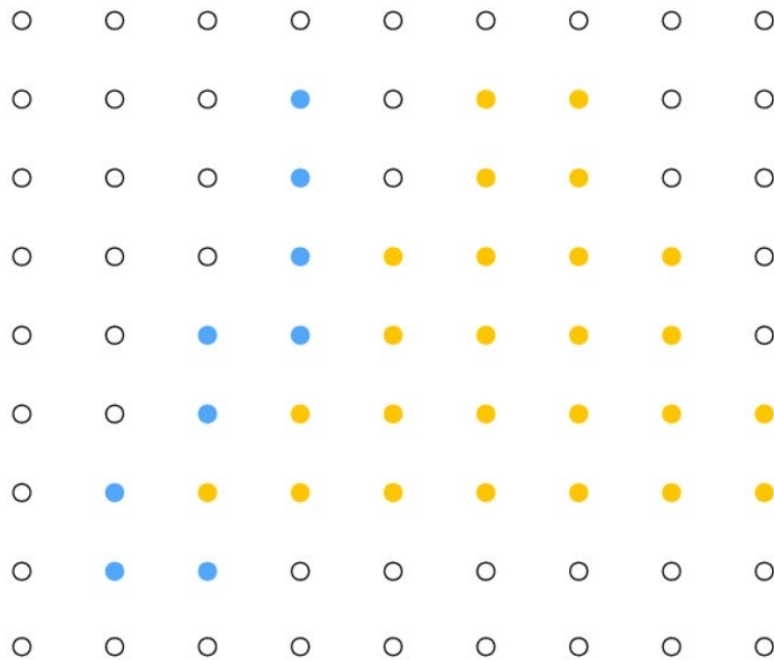


Depth buffer contents



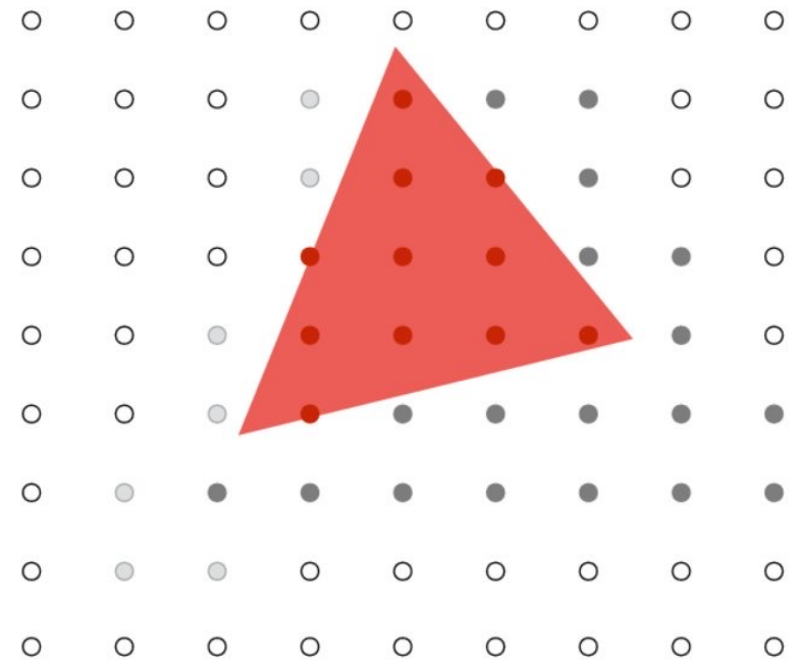
# Occlusion using the depth-buffer (Z-buffer)

Processing red triangle:  
depth = 0.25



Color buffer contents

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

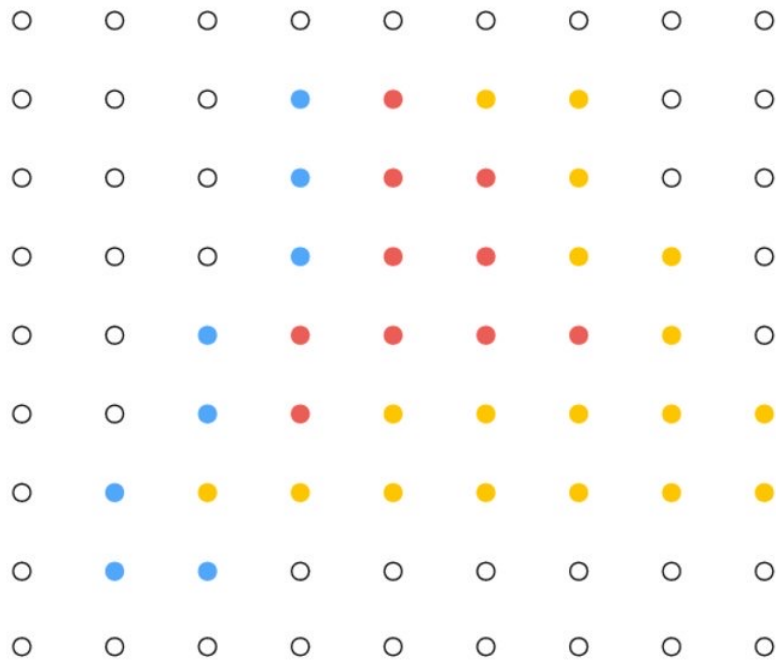


Depth buffer contents



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

After processing red triangle:



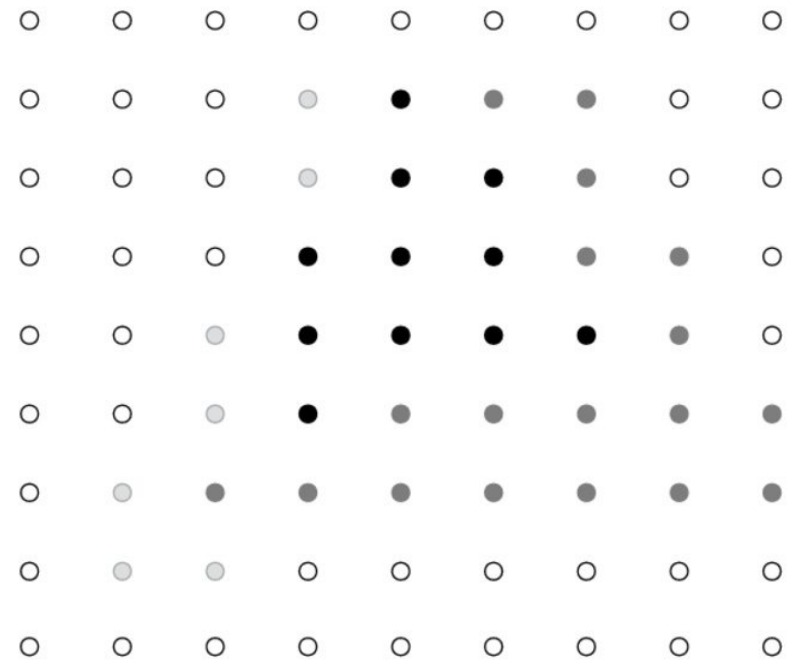
Color buffer contents

Grayscale value of sample point  
used to indicate distance

White = large distance

Black = small distance

Red = samples that pass depth test



Depth buffer contents



# Occlusion using the depth buffer (opaque surfaces)

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

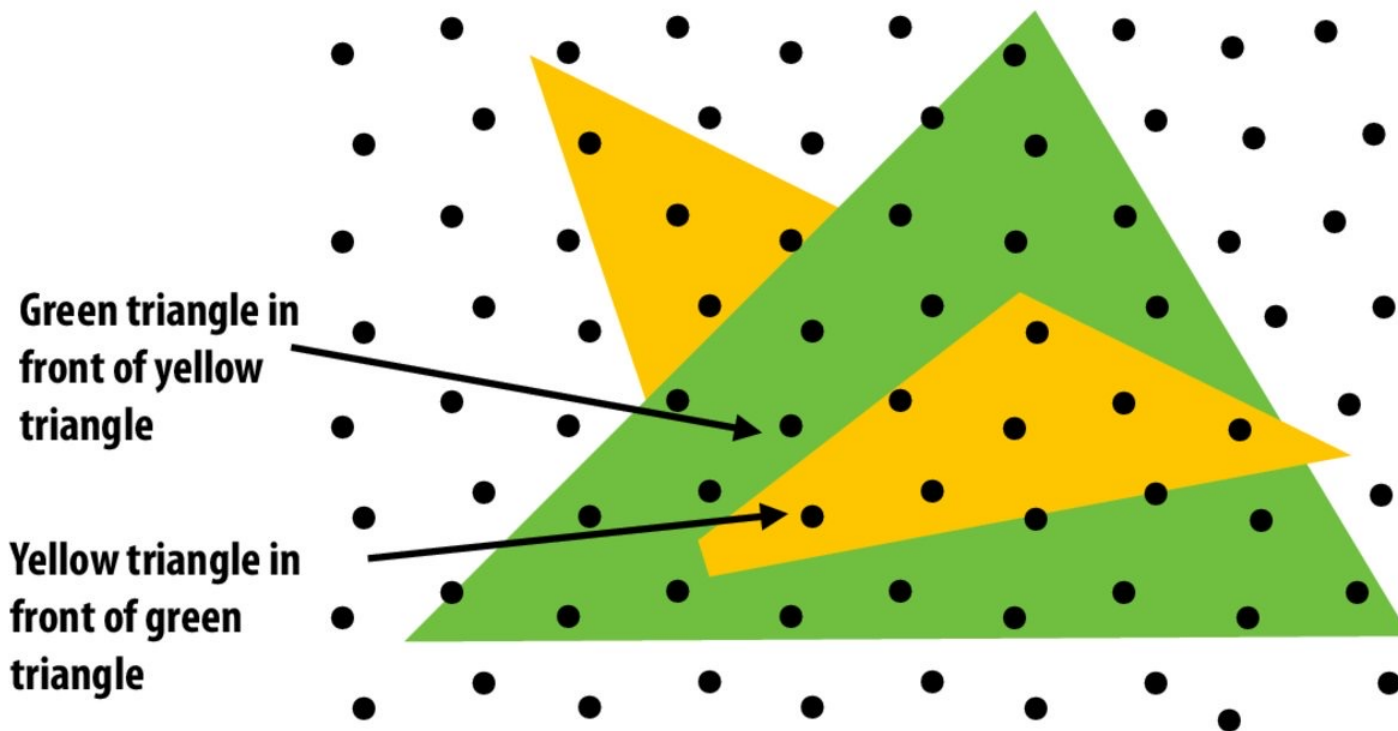
```
depth_test(tri_d, tri_color, x, y) {  
    if (pass_depth_test(tri_d, depth_buffer[x][y])) {  
        // triangle is closest object seen so far at this  
        // sample point. Update depth and color buffers.  
  
        depth_buffer[x][y] = tri_d;    // update depth_buffer  
        color[x][y]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



# Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

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

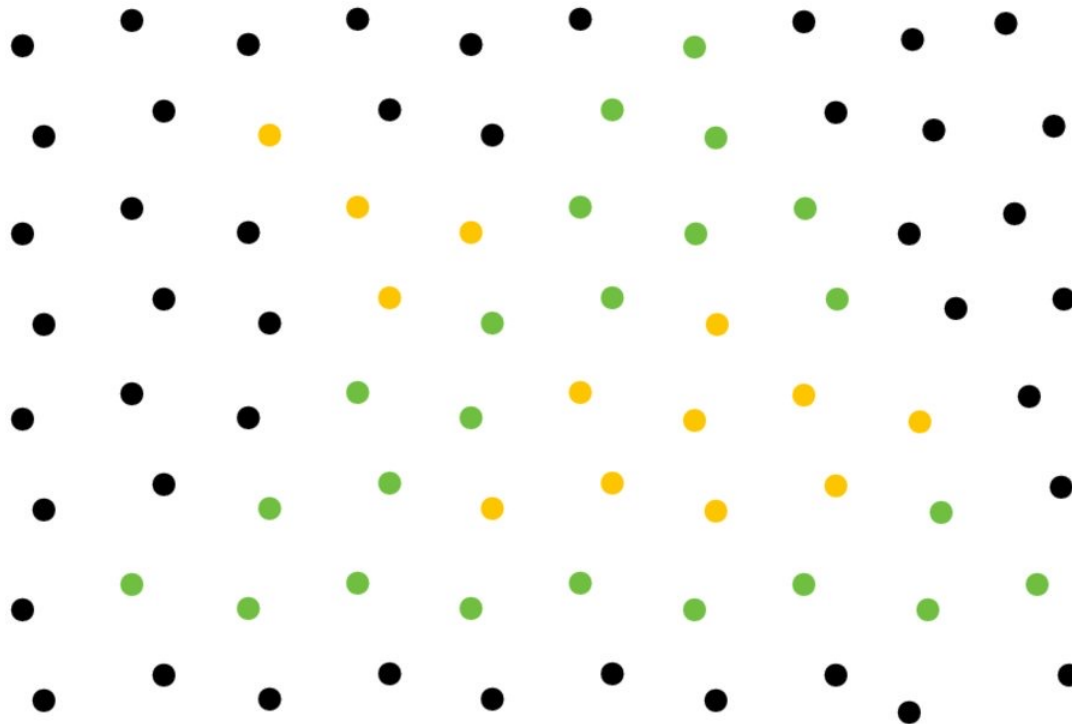




# Does depth-buffer algorithm handle interpenetrating surfaces?

Of course!

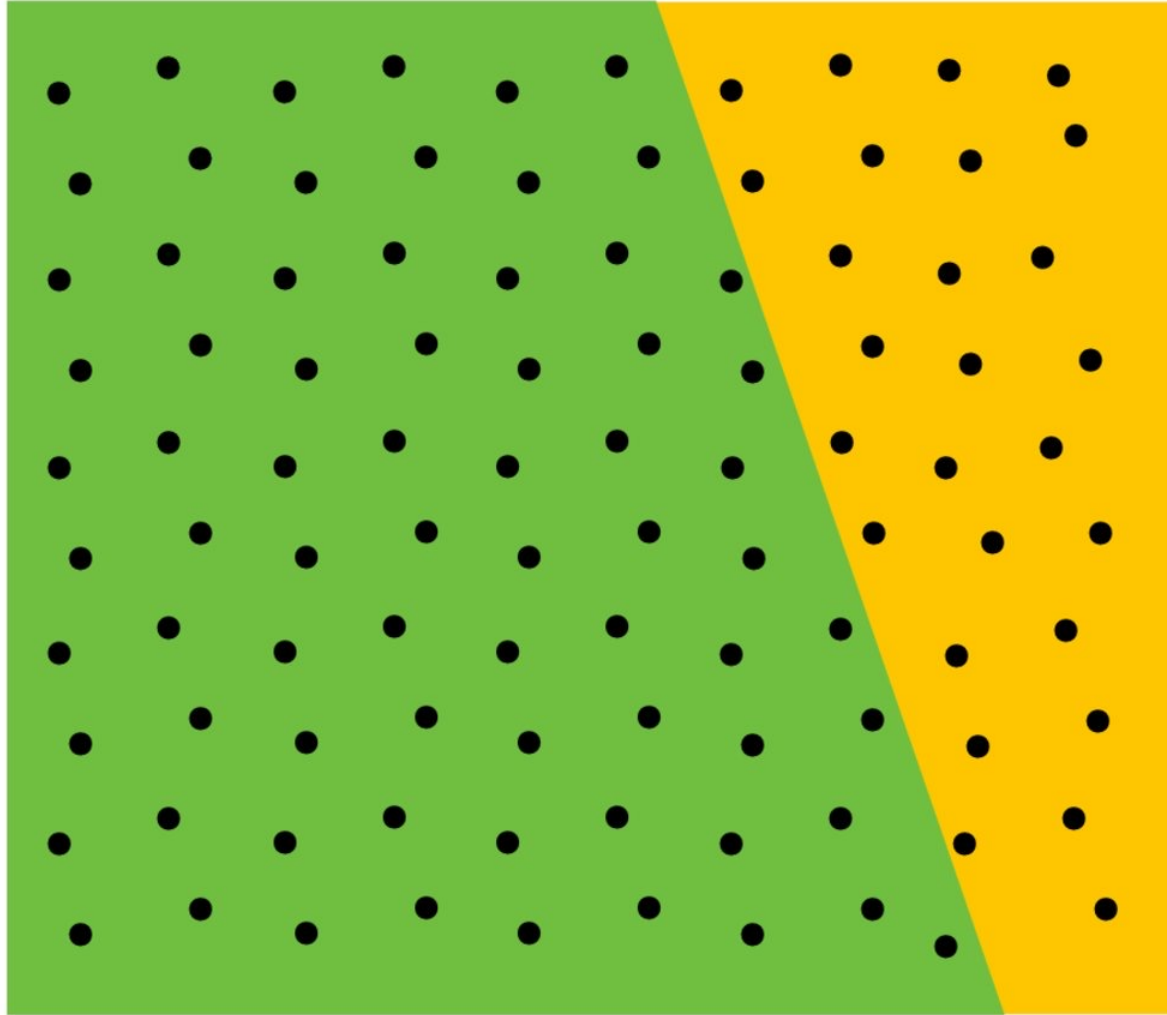
Occlusion test is based on depth of triangles *at a given sample point*. The relative depth of triangles may be different at different sample points.





# Does depth buffer work with super sampling?

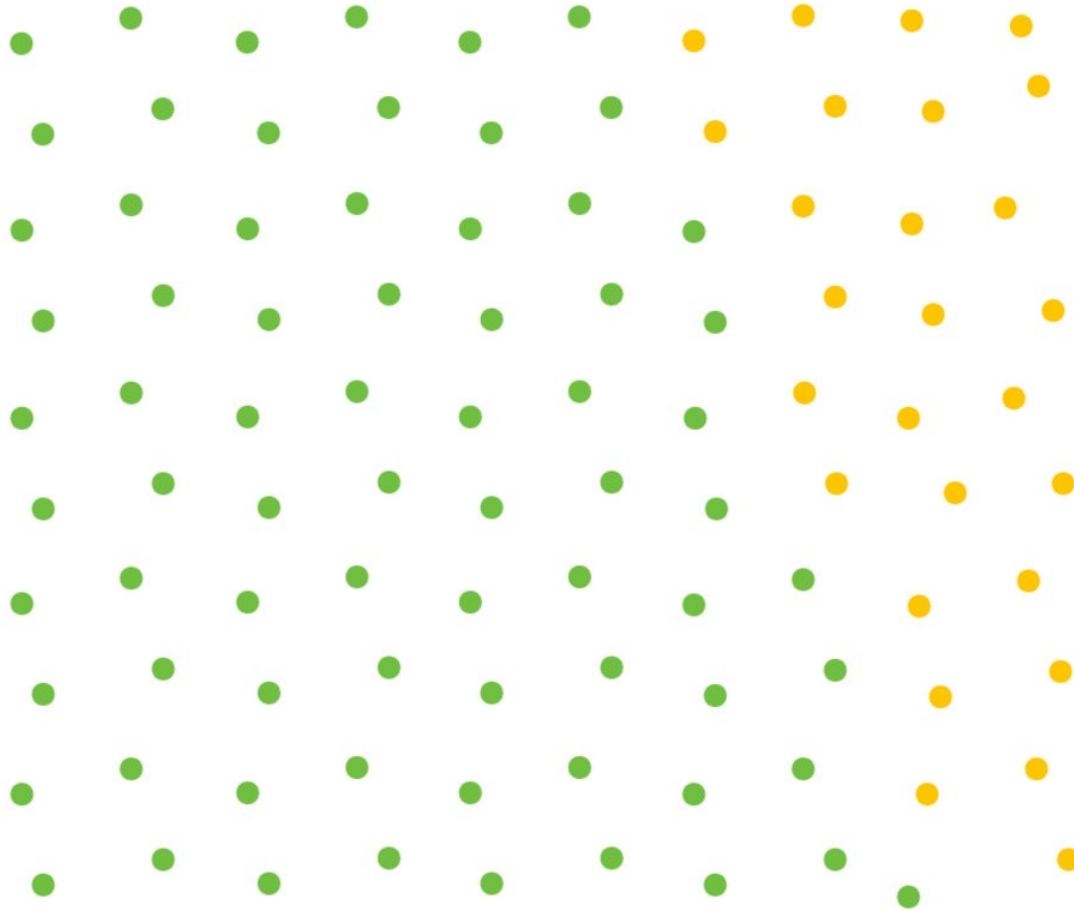
Of course! Occlusion test is per sample, not per pixel!



This example: green triangle occludes yellow triangle

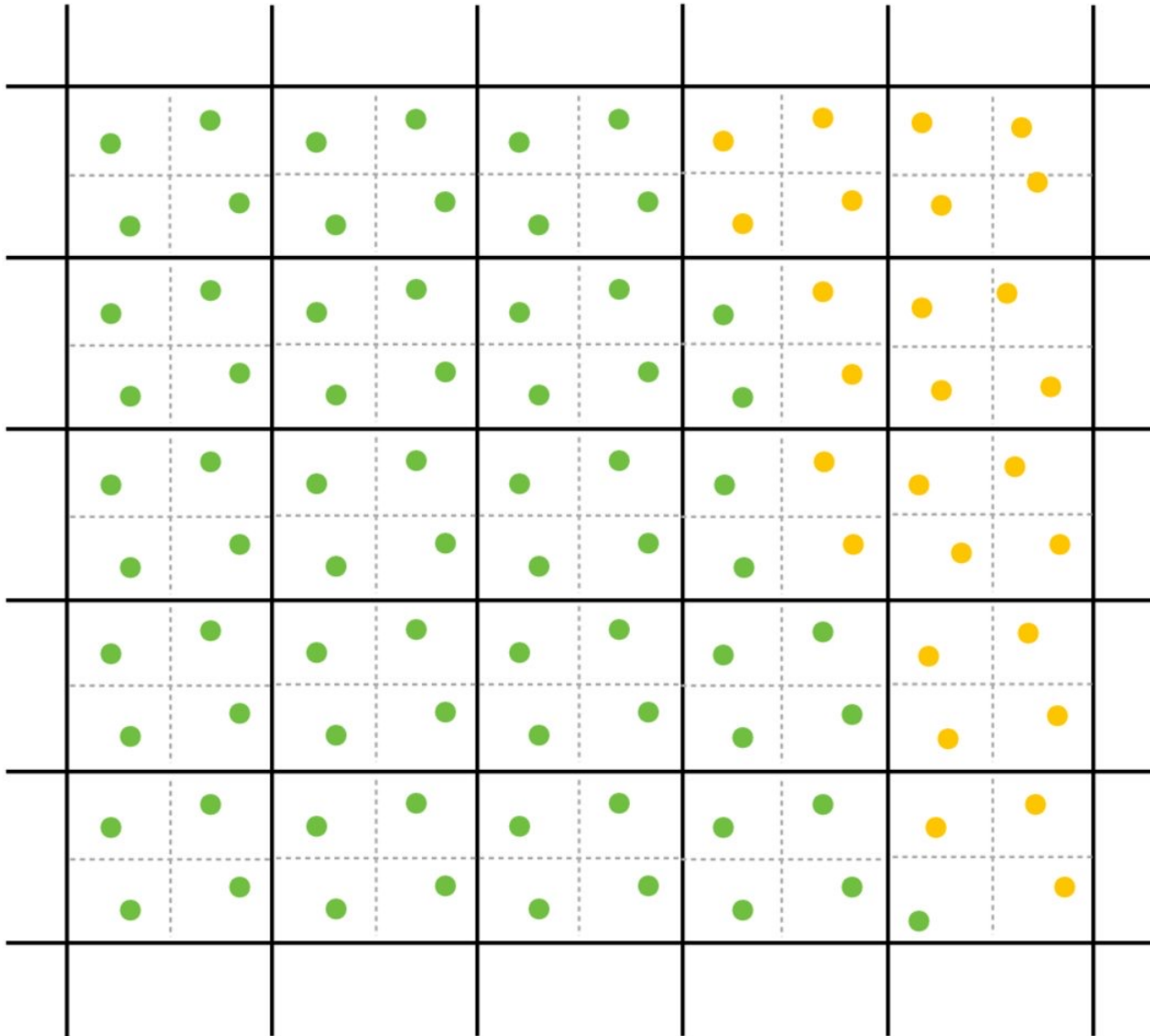


# Color buffer contents



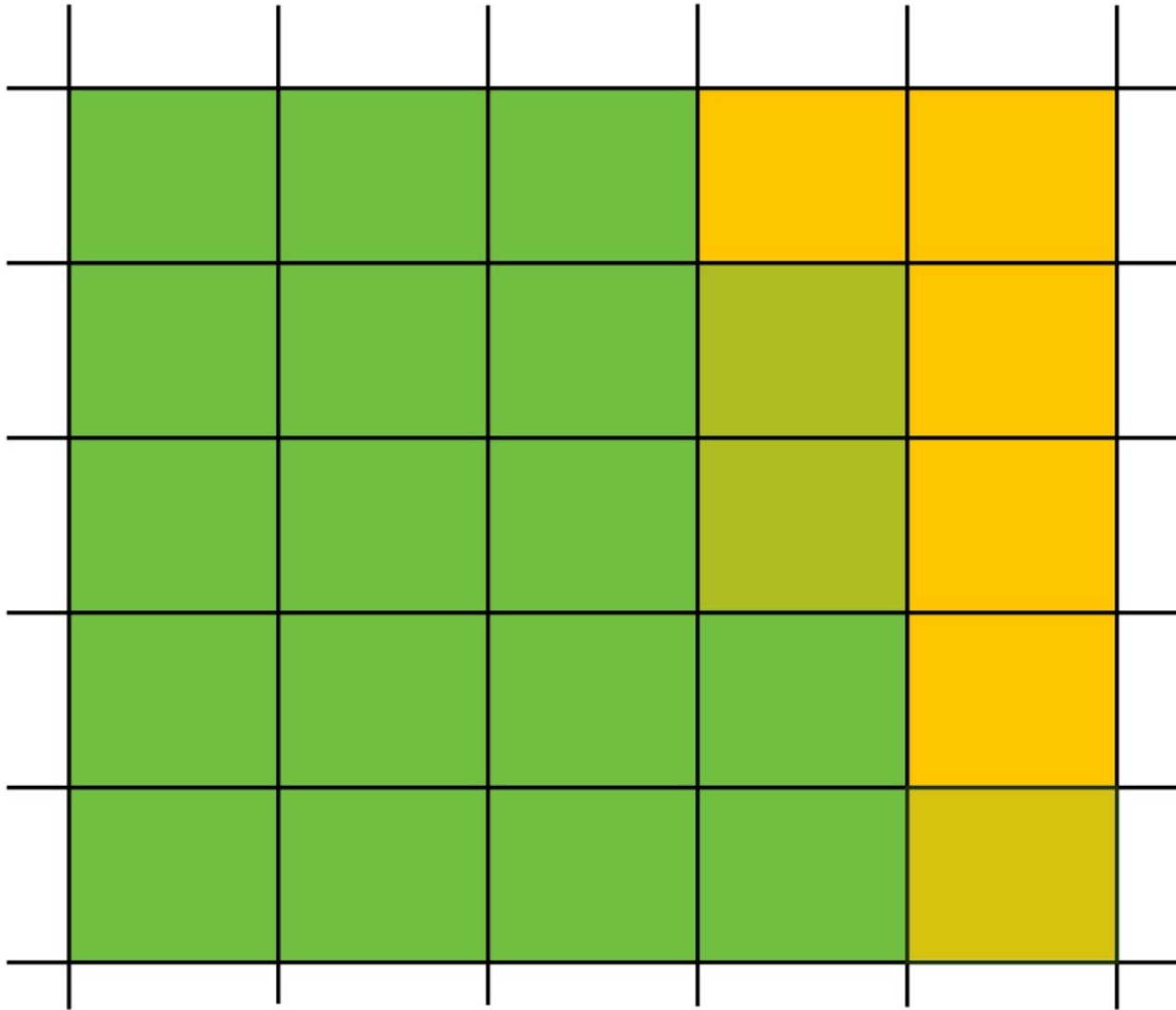


# Color buffer contents (4 samples per pixel)





# Final resampled result



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



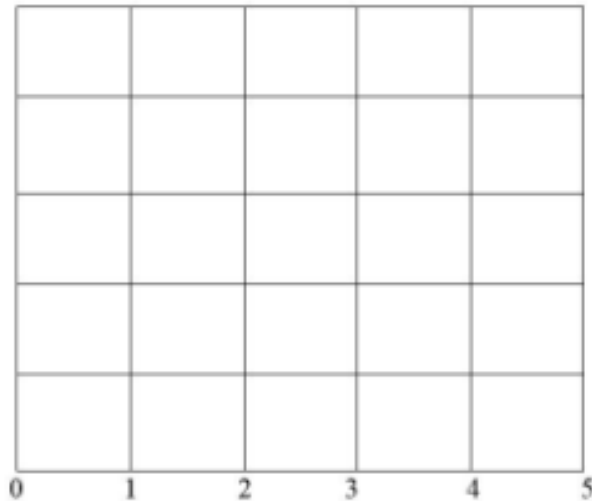
# Summary: occlusion using a depth buffer

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

**But what about semi-transparent surfaces?**



What is the state of the Z-buffer after rendering this scene?  
 $Z=0$  is near.  $Z=9$  is far.



There is a red square with corners at

$(0, 0, 1)$ ,  
 $(1, 0, 1)$ ,  
 $(1, 1, 1)$ ,  
 $(0, 1, 1)$ .

There is a green square with corners at

$(0,0,2)$   
 $(2,0,2)$   
 $(2,2,2)$   
 $(0,2,2)$

9	9	9	9	9
9	9	9	9	9
9	9	9	9	9
2	2	9	9	9
1	2	9	9	9
0	1	2	3	4

A

9	9	9	9	9
9	9	9	9	9
9	9	9	9	9
2	2	9	9	9
2	2	9	9	9
0	1	2	3	4

B

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
2	2	0	0	0
1	2	0	0	0
0	1	2	3	4

C

0	0	0	0	0
0	0	0	0	0
0	0	0	0	0
2	2	0	0	0
2	2	0	0	0
0	1	2	3	4

D



# Compositing

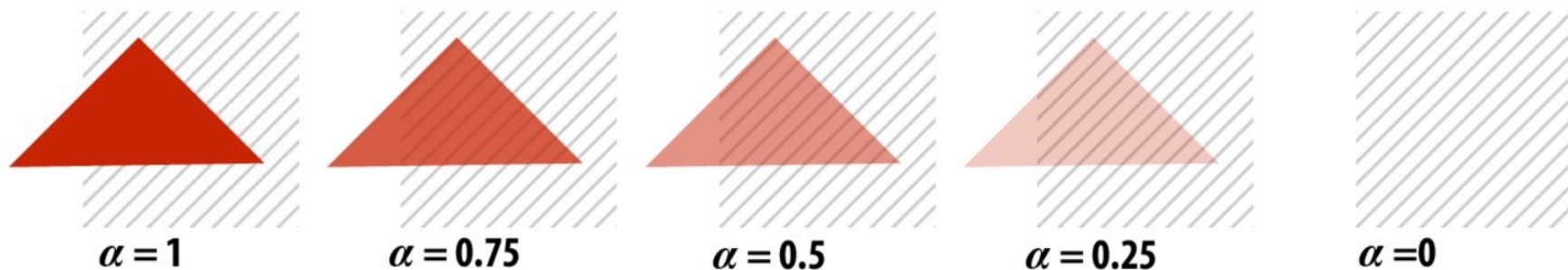


# Representing opacity as alpha

Alpha describes the opacity of an object

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

Red triangle with decreasing opacity





# Alpha: coverage analogy

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

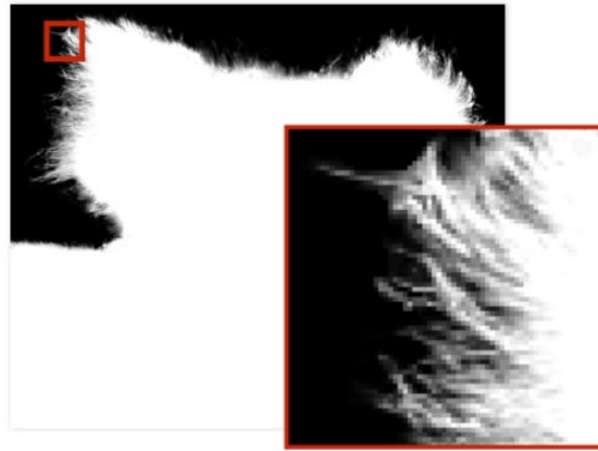
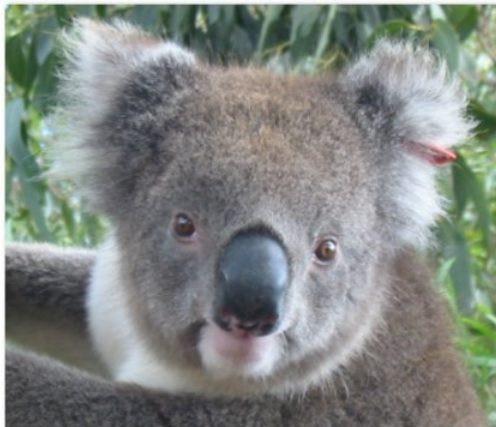
$$\alpha = 0.5$$



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



# Alpha: additional channel of image (rgba)

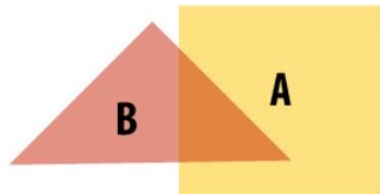


$\alpha$  of foreground object

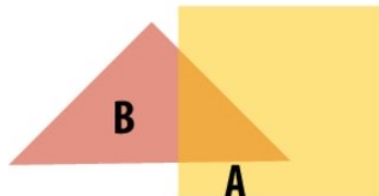


# Over operator:

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



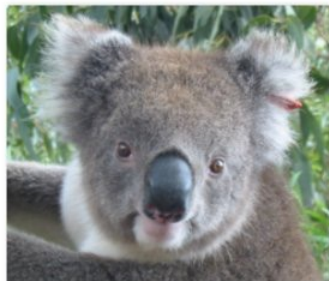
B over A



A over B

$A \text{ over } B \neq B \text{ over } A$

"Over" is not commutative



Koala over NYC



# Over operator: non-premultiplied alpha

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

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

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

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

Composited color:

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

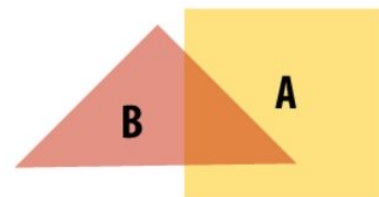
↑  
Appearance of  
semi-transparent B

↑  
What B lets through

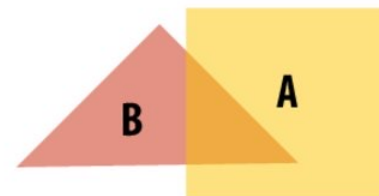
Appearance of semi-transparent A  
↓

Composite alpha:

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



B over A



A over B

A over B  $\neq$  B over A

“Over” is not commutative



# Premultiplied alpha

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

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

**Example: 50% opaque red**

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



**Example: 75% opaque magenta**

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





# Over operator: using premultiplied alpha

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

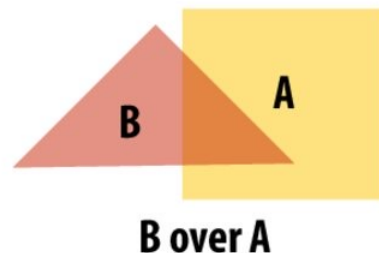
**Non-premultiplied alpha representation:**

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

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

$$C = \alpha_B B + (1 - \alpha_B)\alpha_A A \quad \leftarrow \text{two multiplies, one add}$$

(referring to vector ops on colors)



**Composite alpha:**

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

**Premultiplied alpha representation:**

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

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

$$C' = B' + (1 - \alpha_B)A' \quad \leftarrow \text{one multiply, one add}$$

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

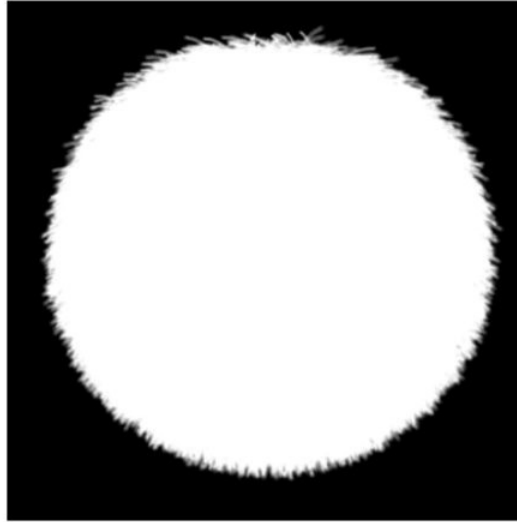


# Fringing

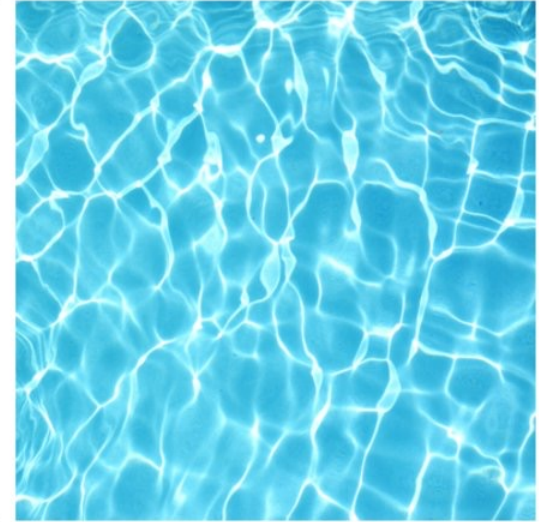
Poor treatment of color/alpha can yield dark “fringing”:



foreground color



foreground alpha



background color



fringing



no fringing



# No fringing





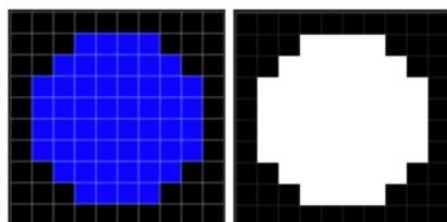
# Fringing (...why does this happen?)





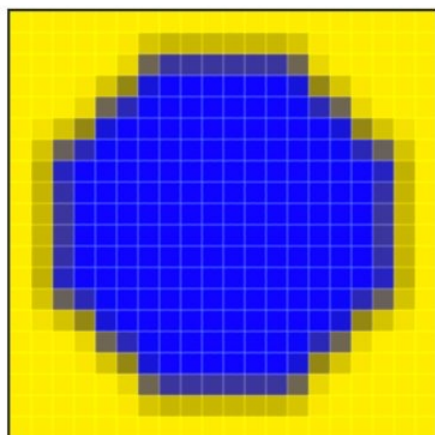
# A problem with non-premultiplied alpha

- 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:

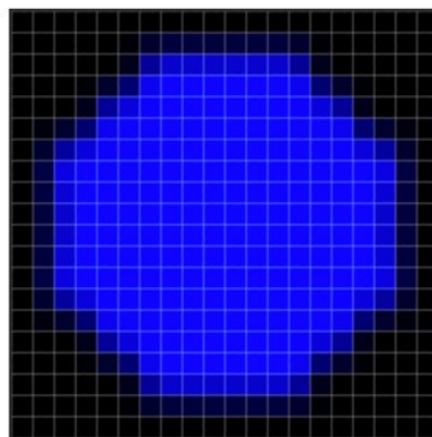


original  
color

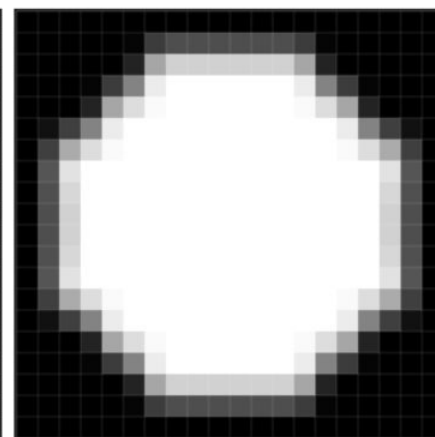
original  
alpha



composited onto  
yellow background



upsampled  
color



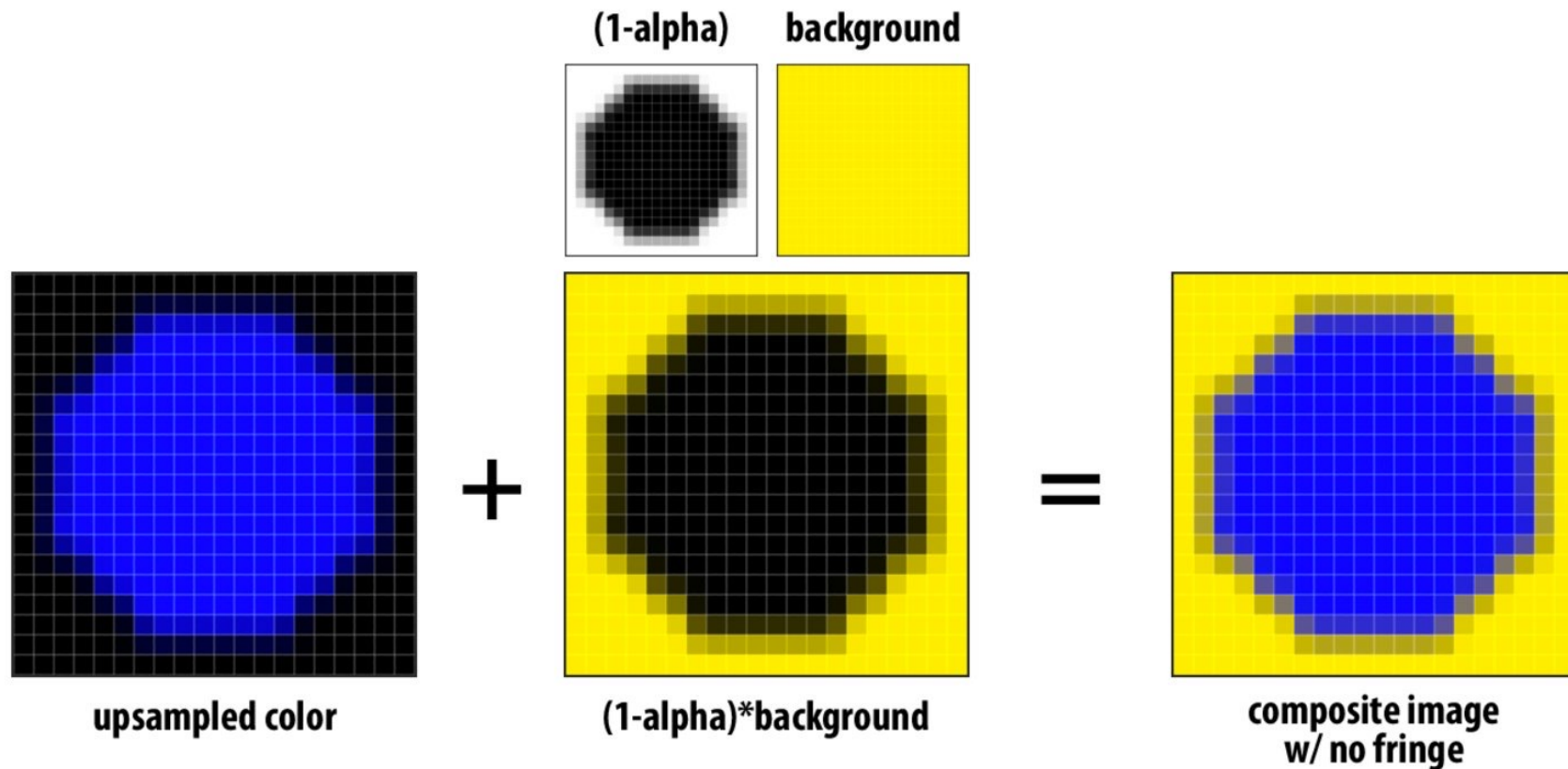
upsampled  
alpha

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:



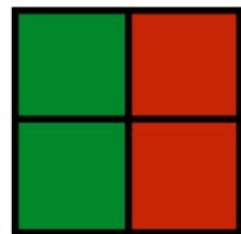


# Another problem with non-premultiplied alpha

Consider pre-filtering a texture with an alpha matte

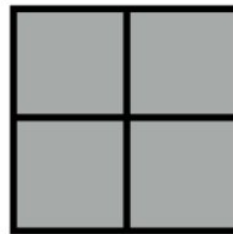
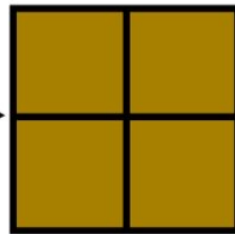


Desired filtered result



input color

input  $\alpha$



filtered color

filtered  $\alpha$

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

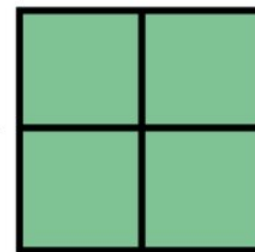


filtered result  
composited over white

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

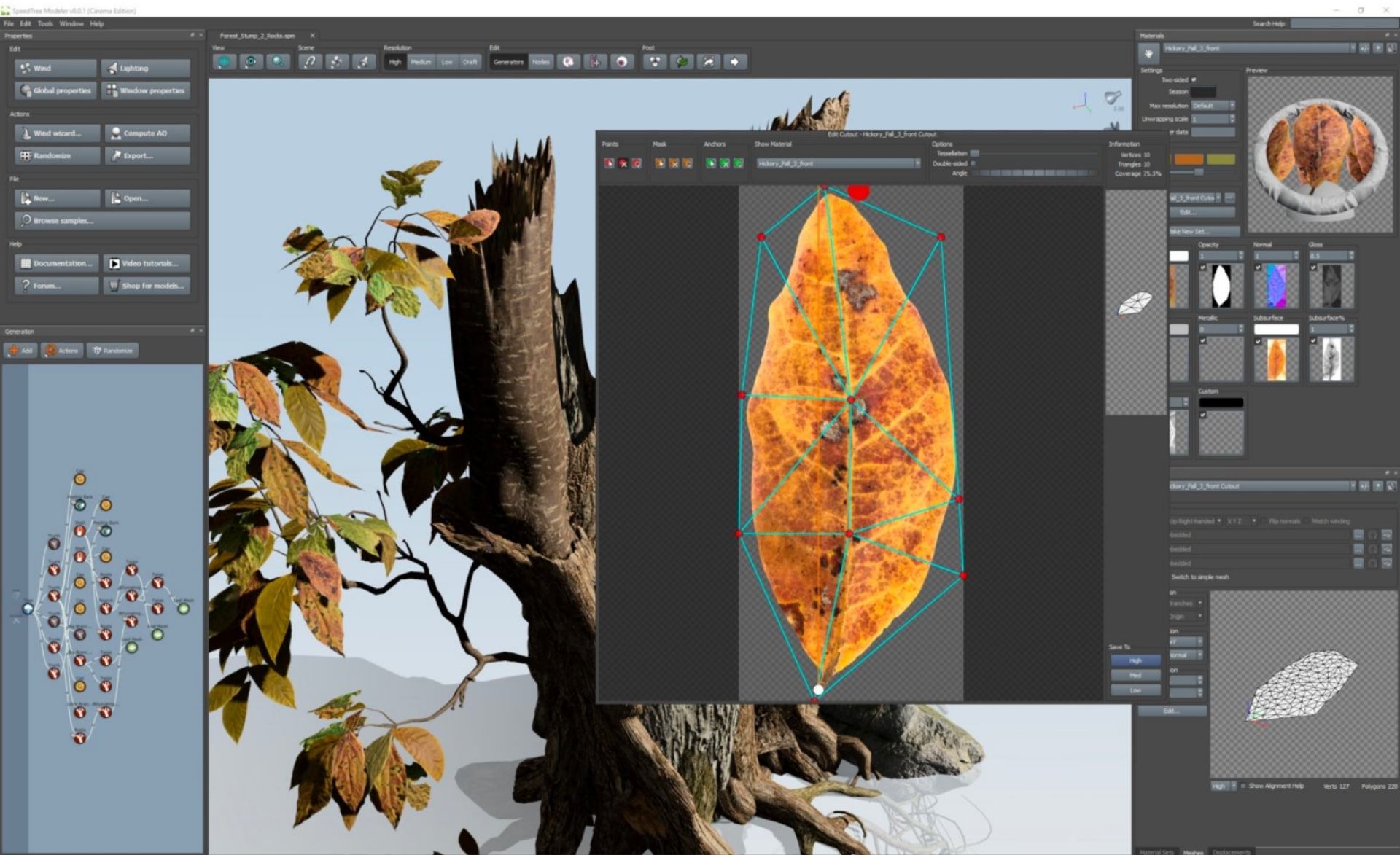
$$(0, 0, 0, 0) + (0, 0, 0, 0)) = (0, 0.5, 0, 0.5)$$

Result of filtering  
premultiplied image





# Common use of textures with alpha: foliage





# Foliage example





# Another problem: applying “over” repeatedly

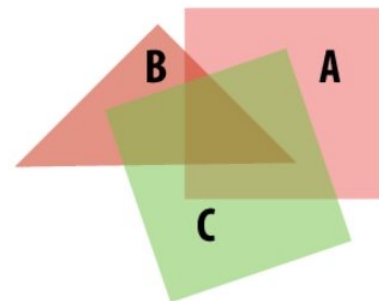
Consider composite image C with opacity  $\alpha_C$  over B with opacity  $\alpha_B$  over image A with opacity  $\alpha_A$

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

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

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

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



C over B over A

Consider first step of 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!

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:  
 $\text{over}(C, \text{over}(B, A))$

There is a closed form for non-premultiplied alpha:

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



# Summary: advantages of premultiplied alpha

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



# Color buffer update: semi-transparent surfaces

Assume: color buffer values and tri\_color are represented with premultiplied alpha

```
over(c1, c2) {  
    return c1 + (1-c1.a) * c2;  
}  
  
update_color_buffer(tri_z, tri_color, x, y) {  
    // Note: no depth check, no depth buffer update  
    color[x][y] = over(tri_color, color[x][y]);  
}
```

What is the assumption made by this implementation?

**Triangles must be rendered in back to front order!**

What if triangles are rendered in front to back order?

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



# Putting it all together \*

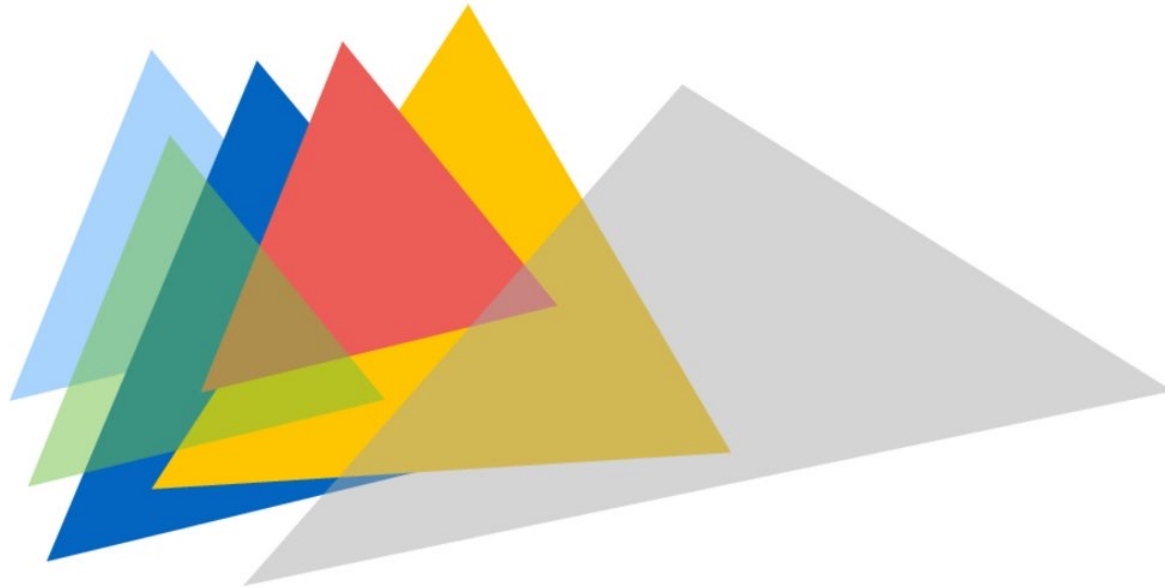
**Consider rendering a mixture of opaque and transparent triangles**

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

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

**Step 2: disable depth buffer update, render semi-transparent surfaces in back-to-front order.**

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



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



# Combining opaque and semi-transparent triangles

Assume: color buffer values and tri\_color are represented with premultiplied alpha

*// phase 1: render opaque surfaces*

```
update_color_buffer(tri_z, tri_color, x, y) {  
    if (pass_depth_test(tri_z, zbuffer[x][y])) {  
        color[x][y] = tri_color;  
        zbuffer[x][y] = tri_z;  
    }  
}
```

*// phase 2: render semi-transparent surfaces*

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



# Participation Survey

## Participation May 7

Form description

This form is automatically collecting email addresses for UC Santa Cruz users. [Change settings](#)

I was in class May 7

- ☐ Yes
- ☐ No

No really. Do you usually go to class?

- ☐ Yes. I listen to the whole lecture as my primary activity.
- ☐ Yes. But I check my email, play a game, or something else on screen during class
- ☐ Yes, But I frequently walk away to do something else
- ☐ Yes, But I watch the video afterwards, not live Zoom
- ☐ Sort of, I mean to watch the video afterwards, but I only do it sometimes
- ☐ Sort of, I come long enough to find the participation survey, then I'm out of here
- ☐ Nah, I get the survey link somehow and just make sure I did it
- ☐ Nah, I'm wasn't even here
- ☐ Other...

Do you ever look at the recorded Zoom video files?

Suggestions: [Add all](#) | [Yes](#) [No](#) [Maybe](#)

- ☐ Yes. Every time.
- ☐ Yes, but only when I need to go back and check on something, like during HW
- ☐ Nope
- ☐ Other...



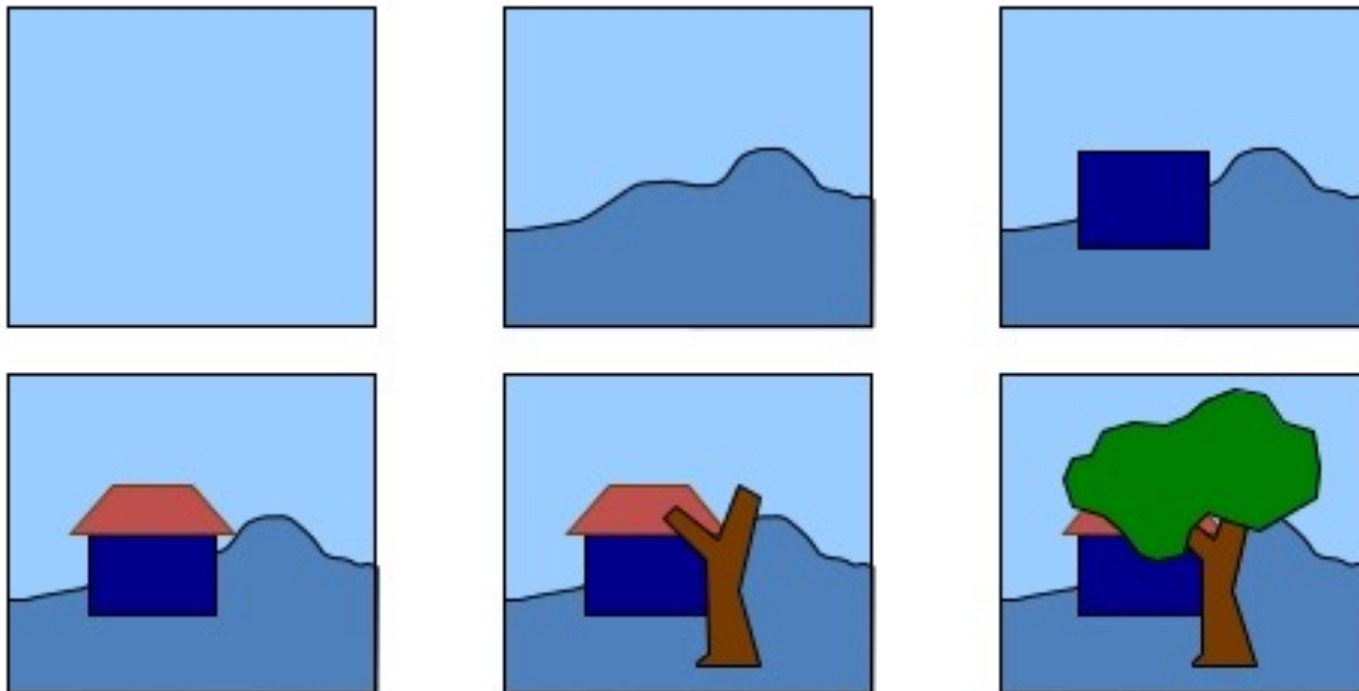
# Back To Front: Painters Algorithm and BSP Trees



# Painter's Algorithm

Draw surfaces from back (farthest away) to front (closest):

- Sort surfaces/polygons by their depth (z value)
- Draw objects in order (farthest to closest)
- Closer objects paint over the top of farther away objects

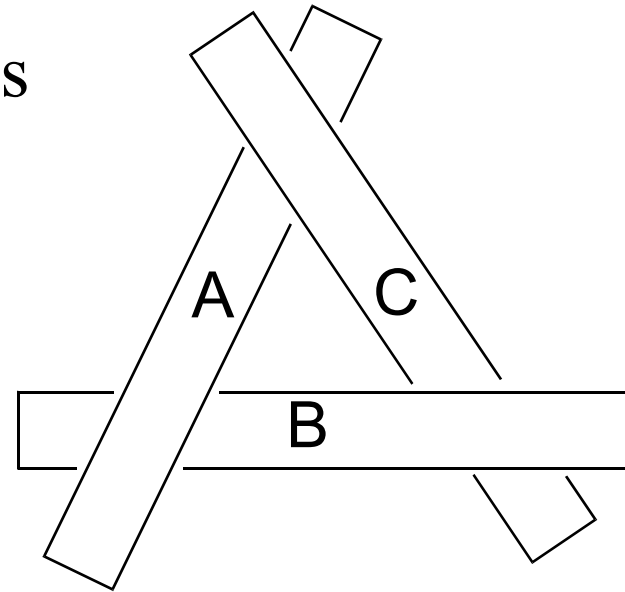
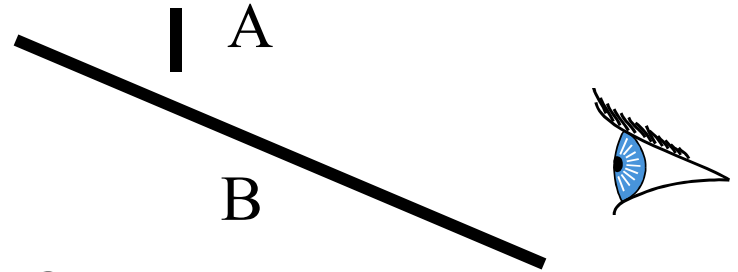




# Painter's algorithm

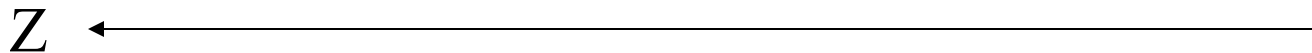
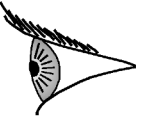
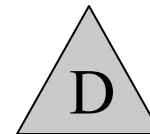
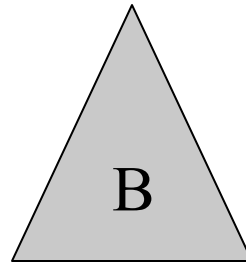
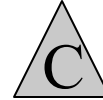
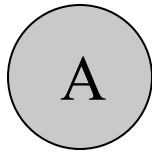
---

- Draw back-to-front
- How do we sort objects?
- Can we always sort objects?
  - No, there can be cycles
  - Requires to split polygons





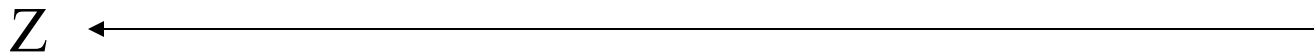
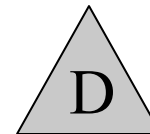
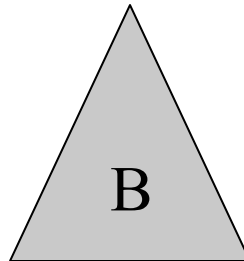
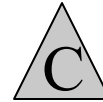
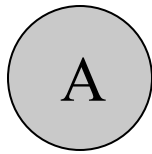
Worse, there is no single sort: ABCD?







Camera moved : ABCD?  
Resort every frame?

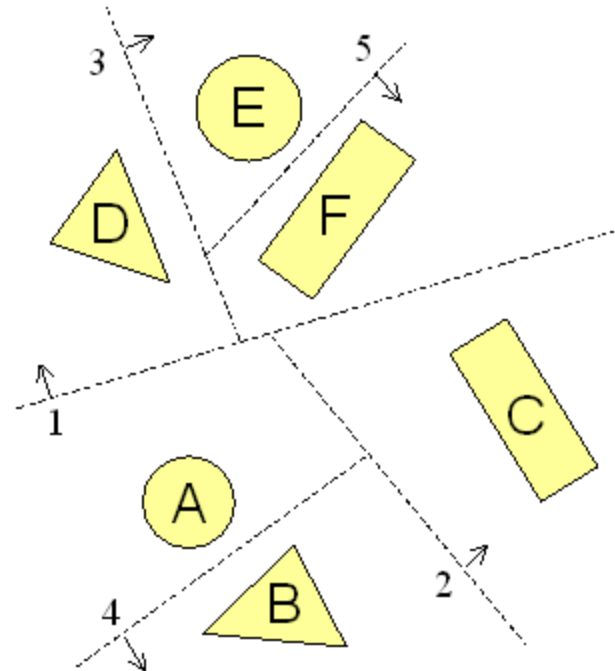
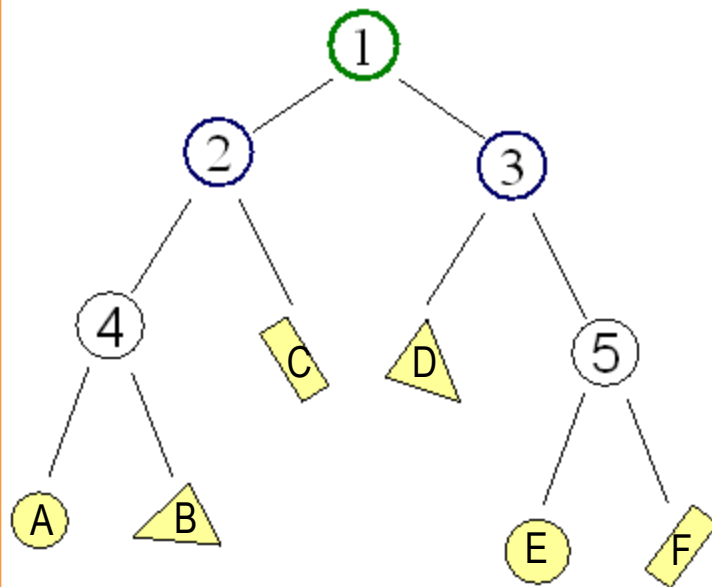




# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron



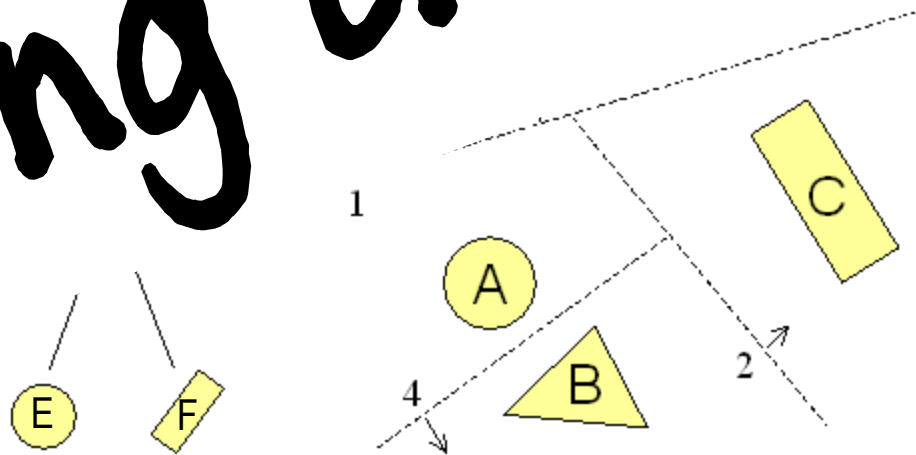


# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron

Building the tree

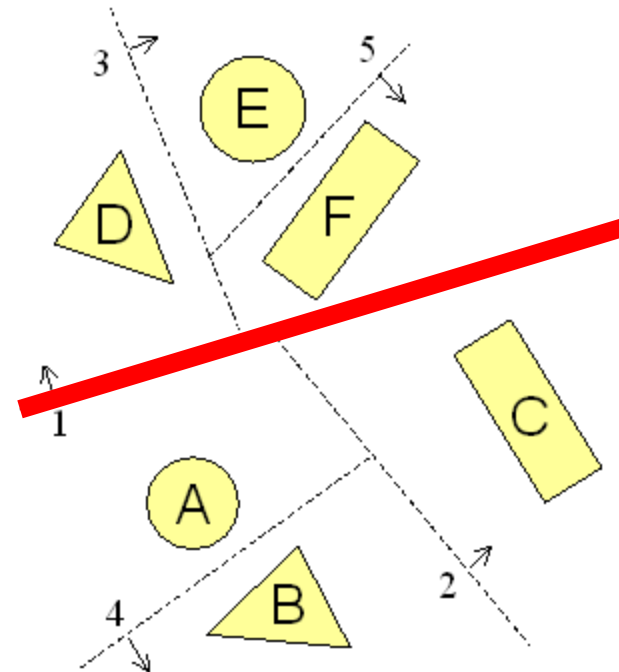
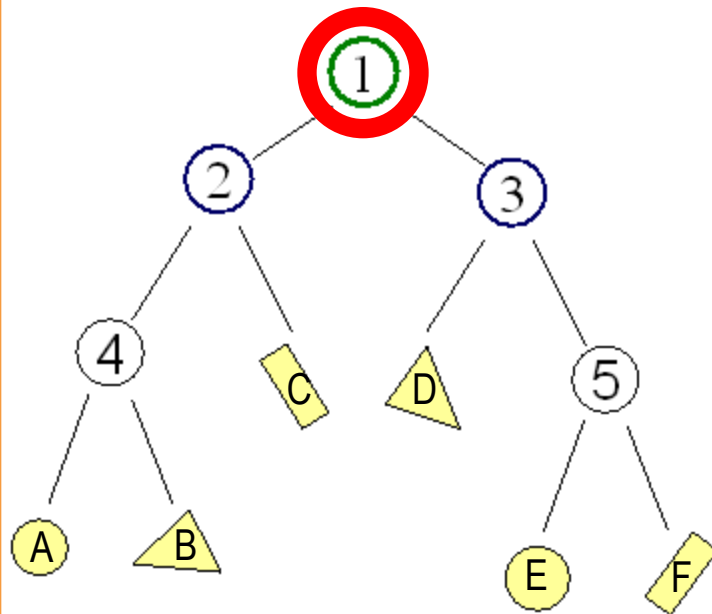




# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron

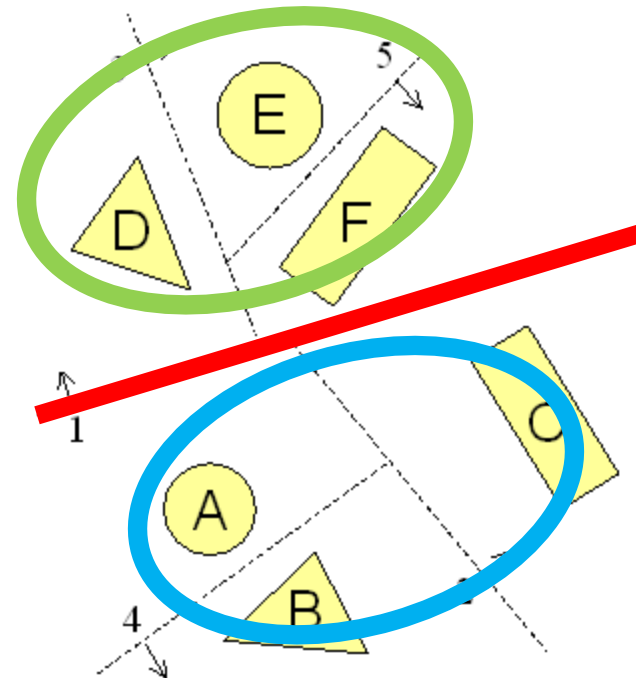
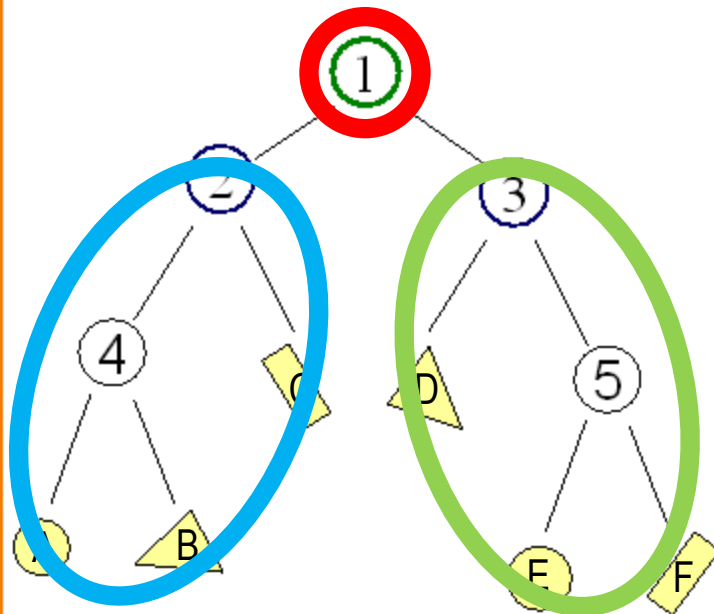




# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron

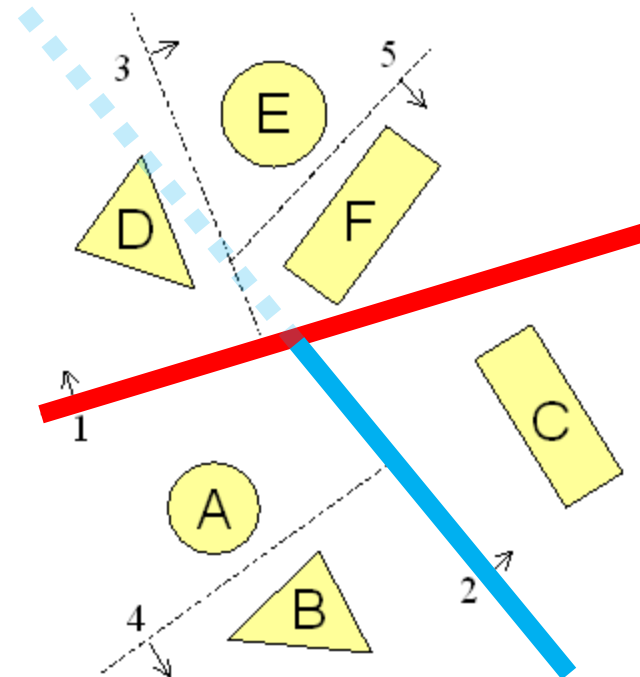
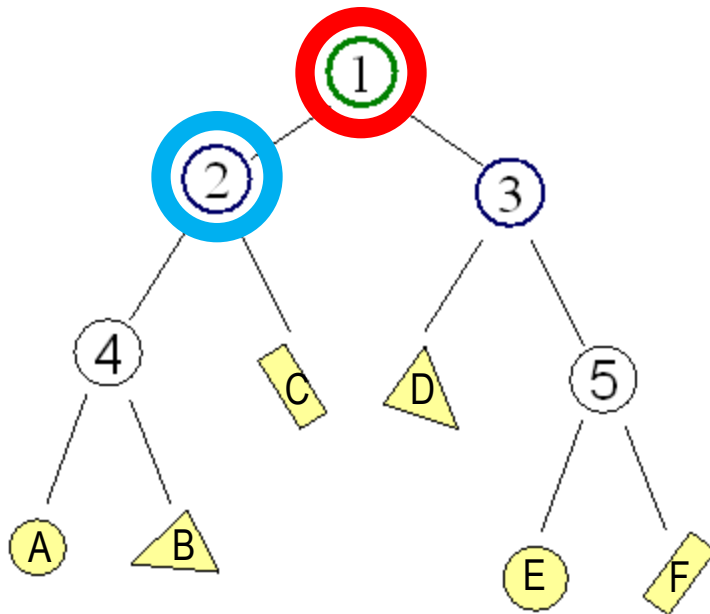




# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron

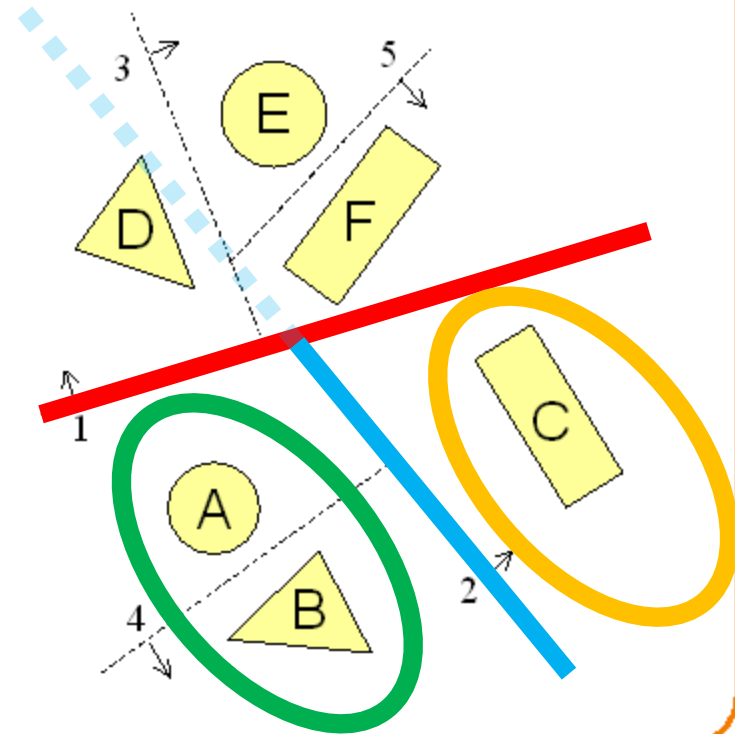
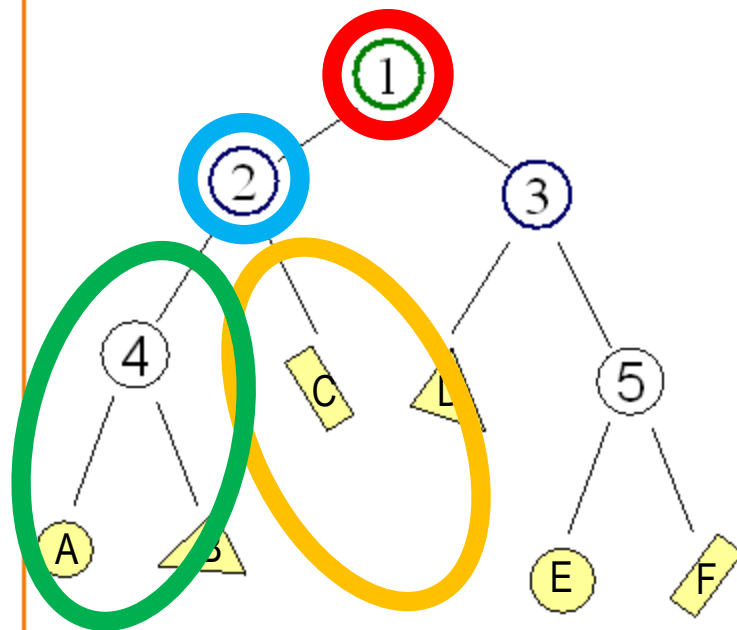




# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron

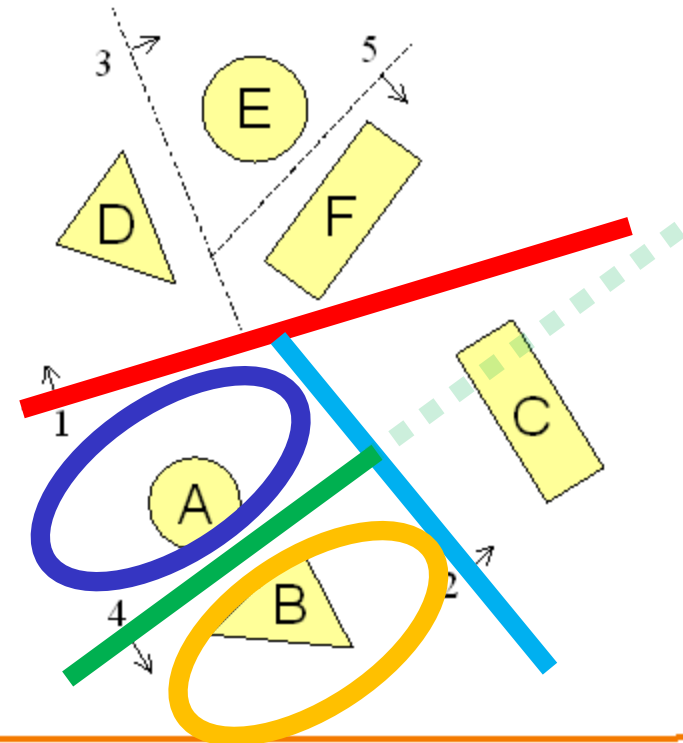
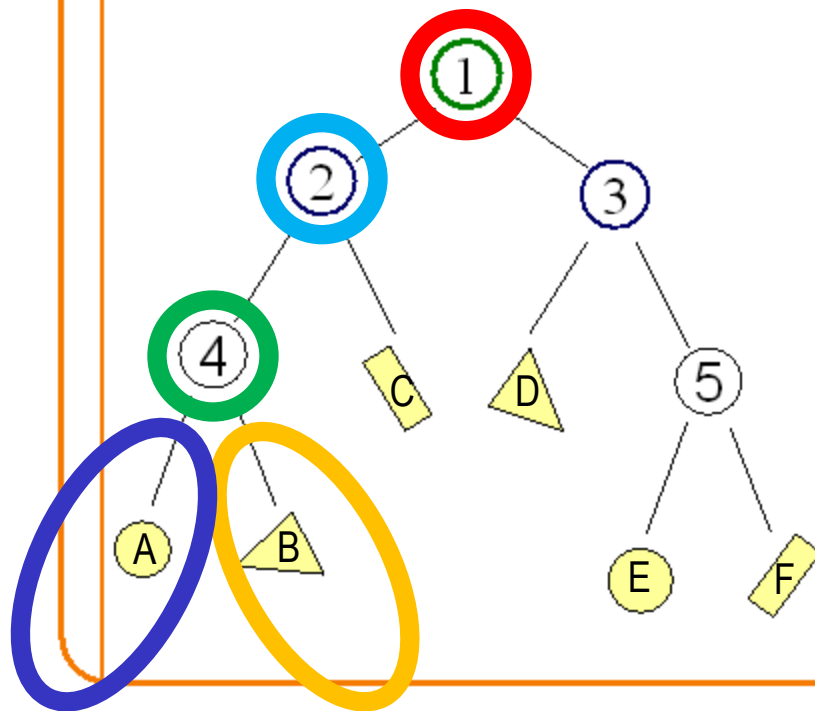




# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron

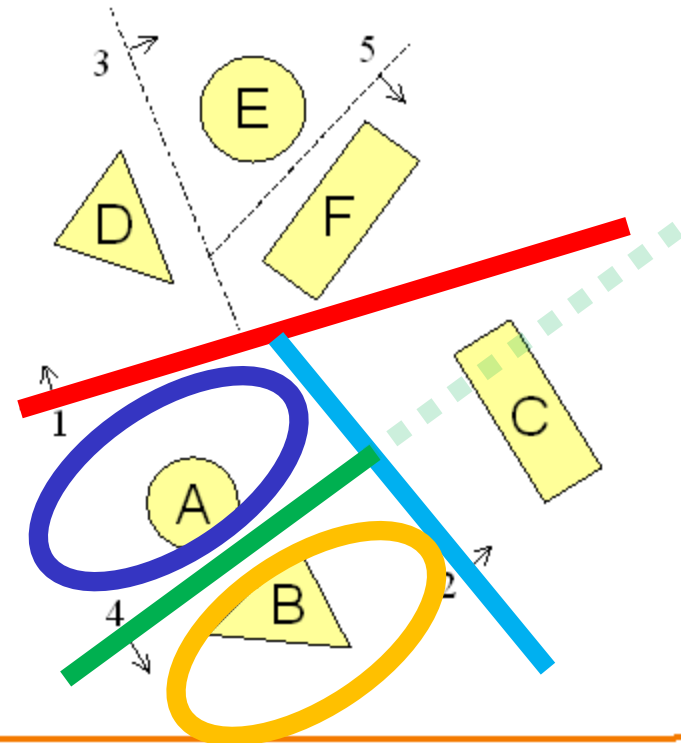
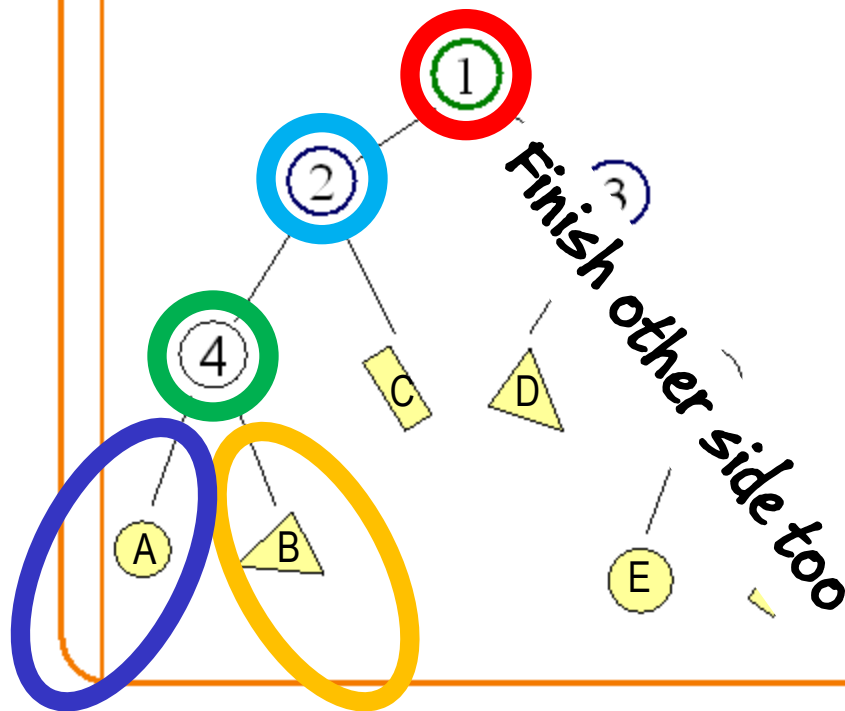




# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron



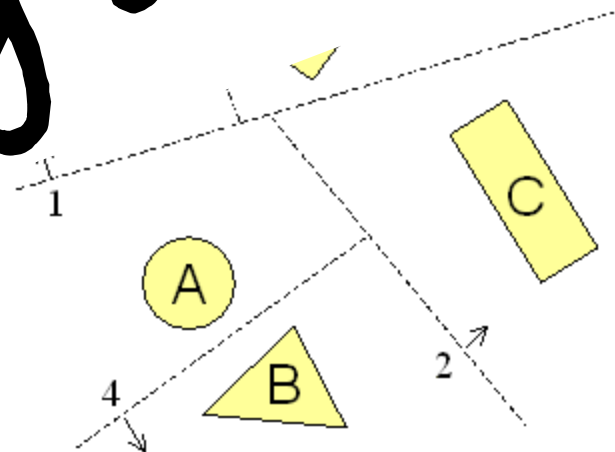
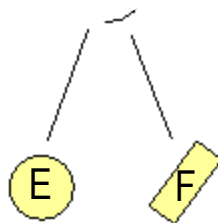
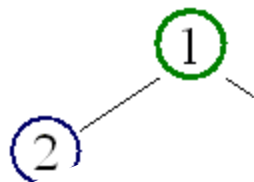


# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron

Traversing the tree

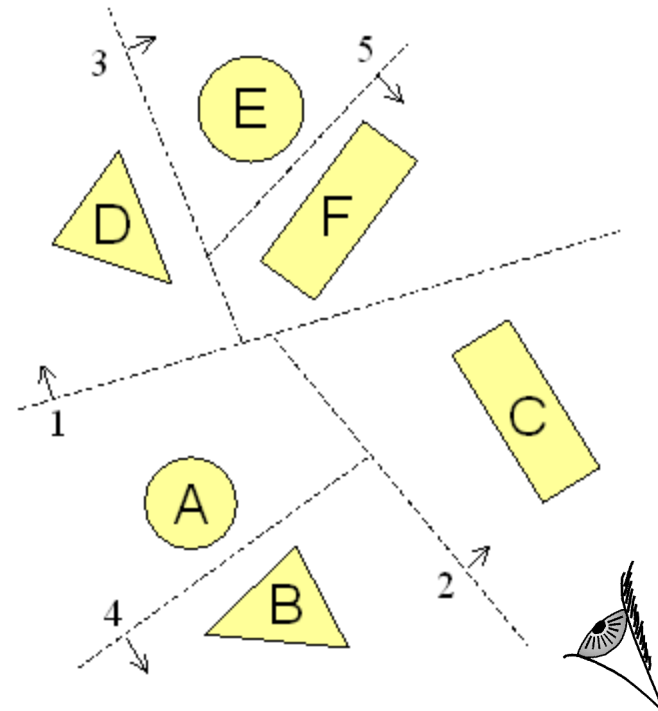
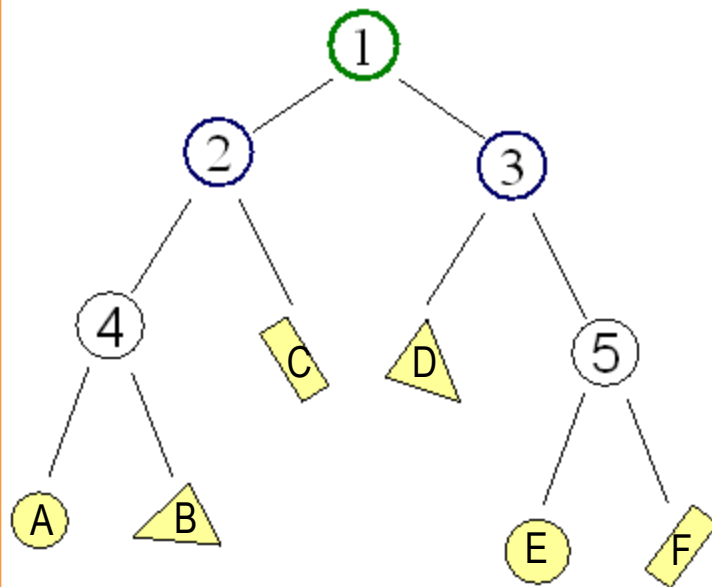




# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron

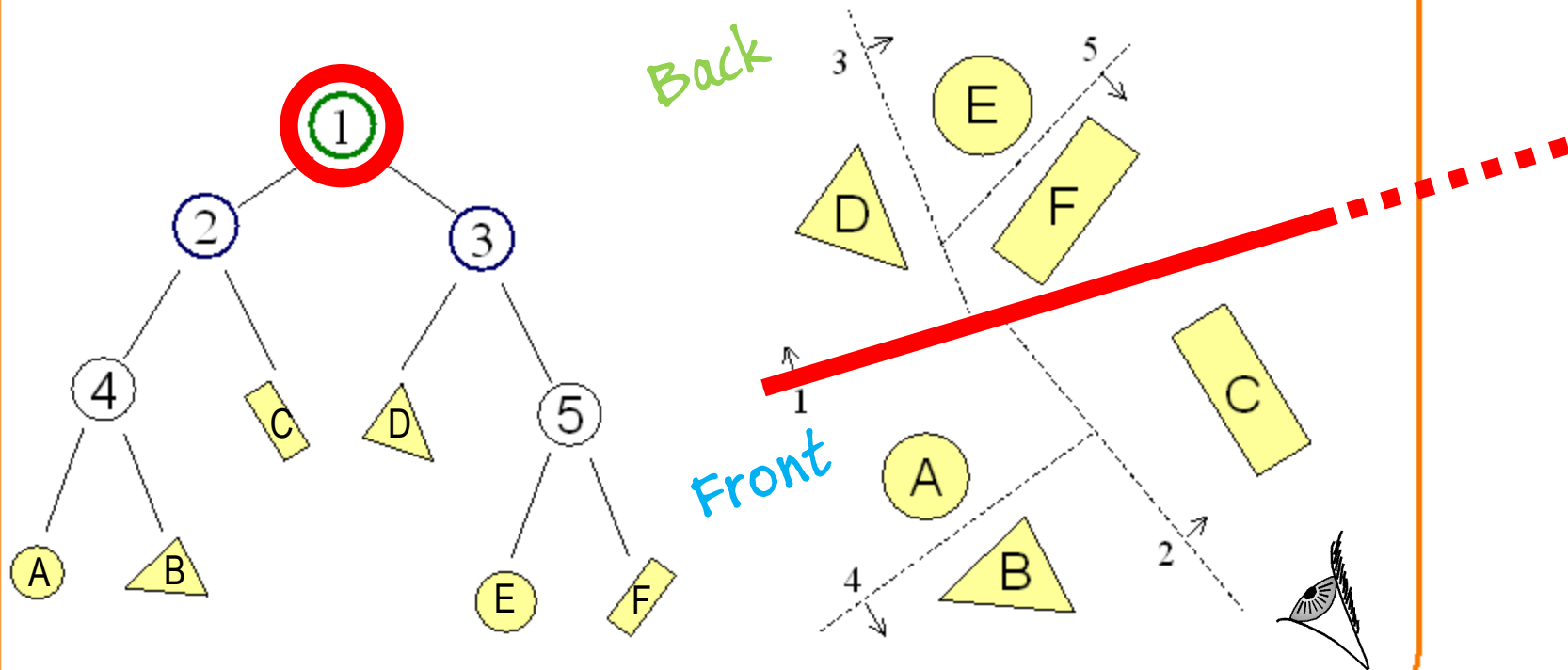




# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron

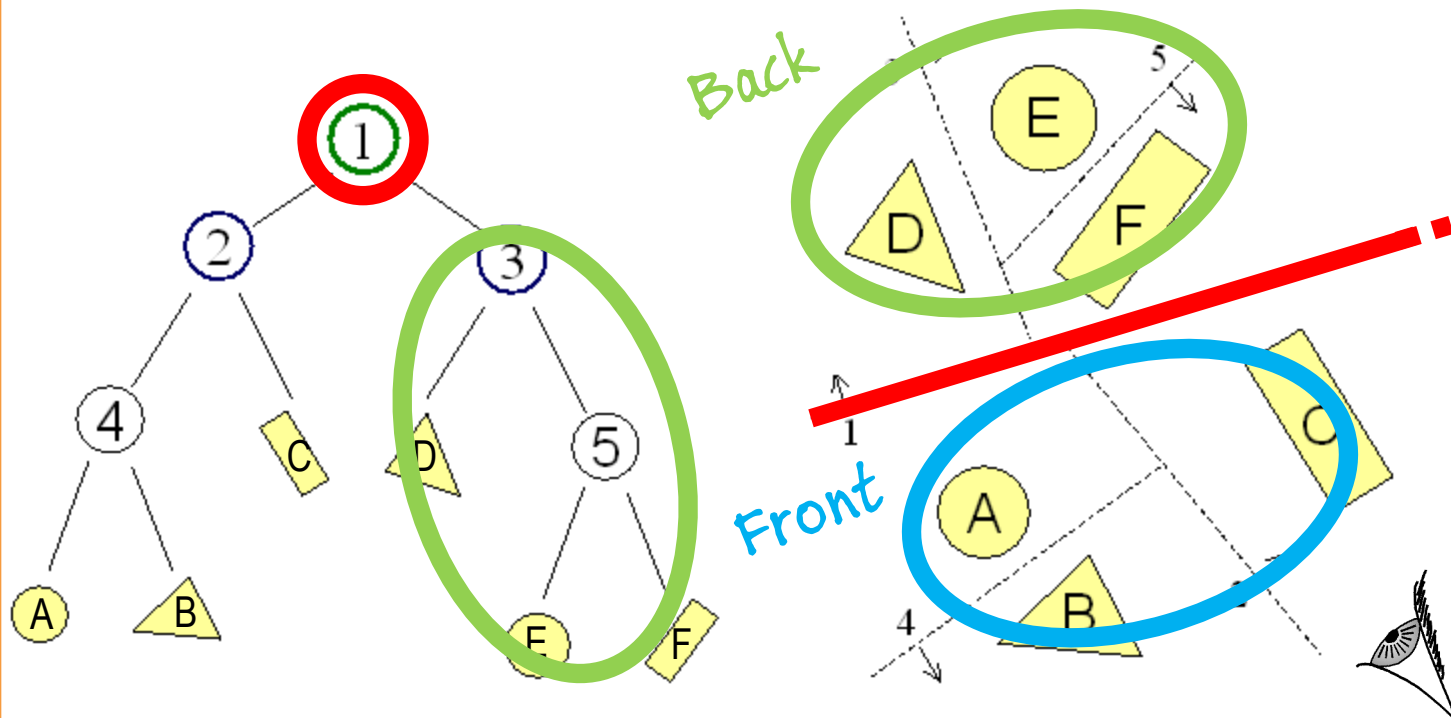




# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron





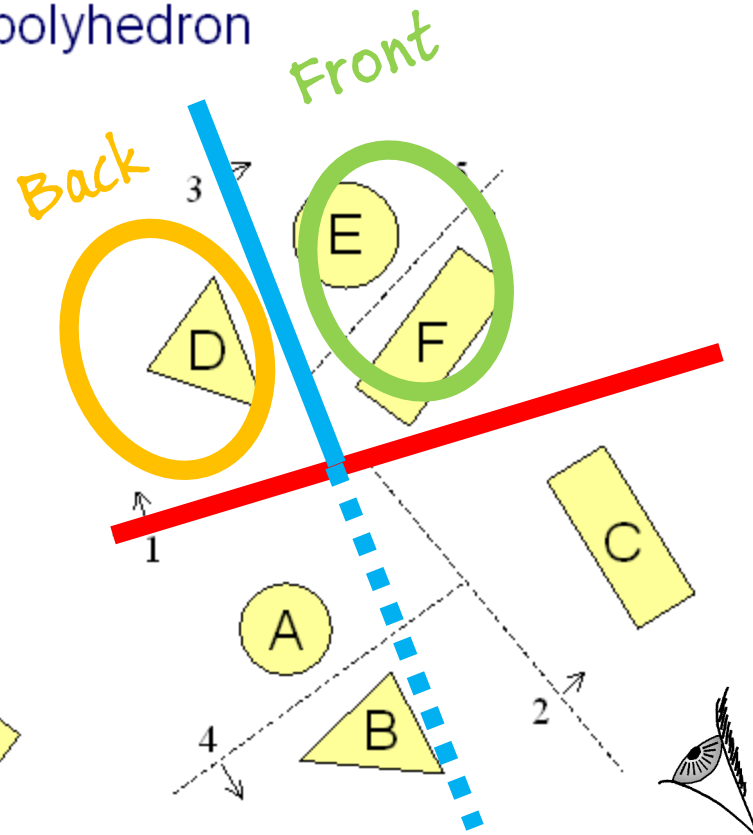
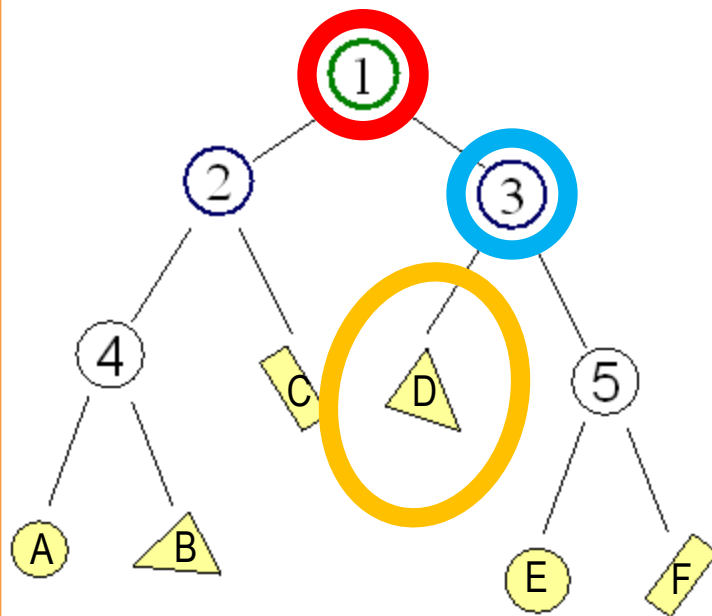




# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron



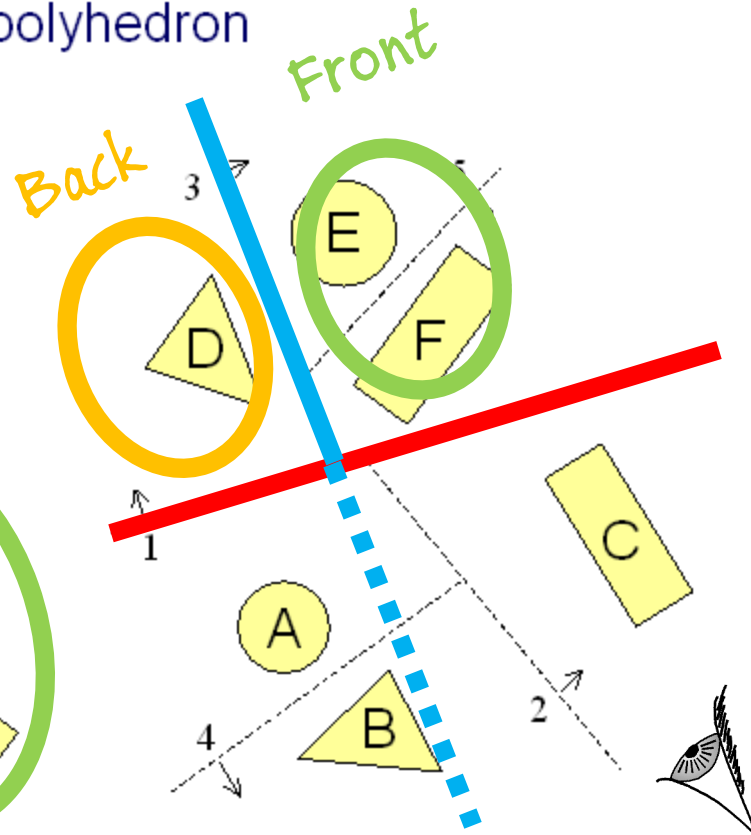
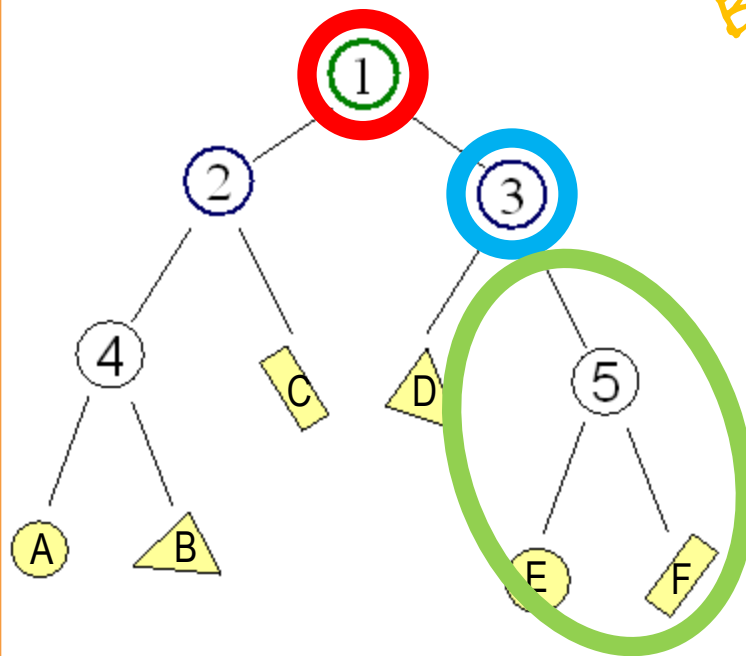
Drawing order: D,



# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron



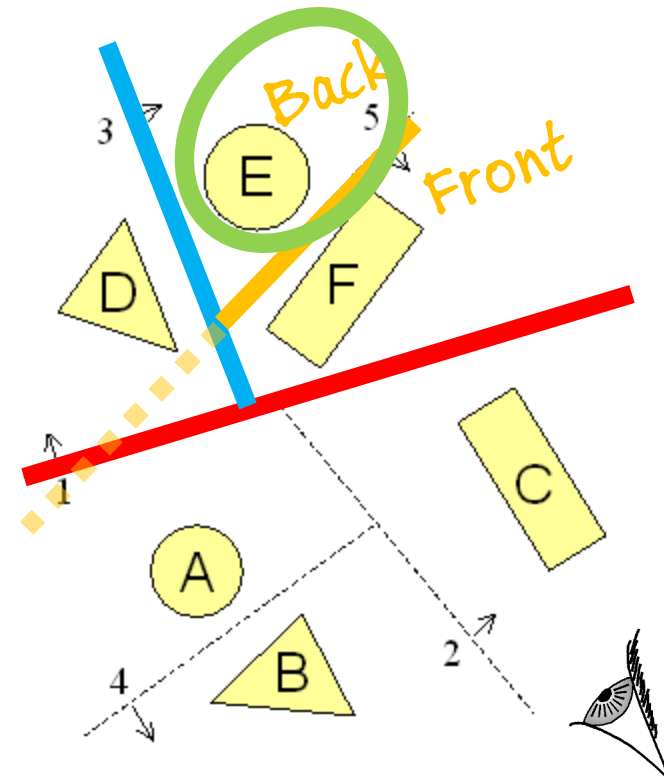
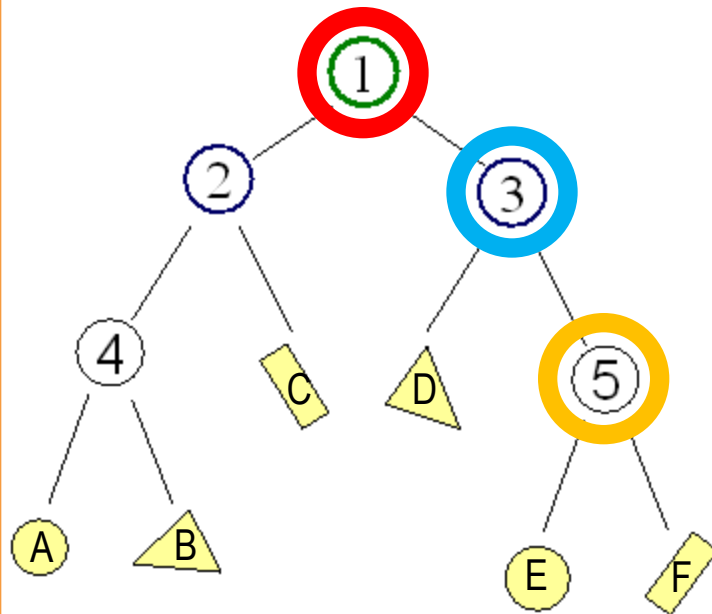
Drawing order: D, (EF)



# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron



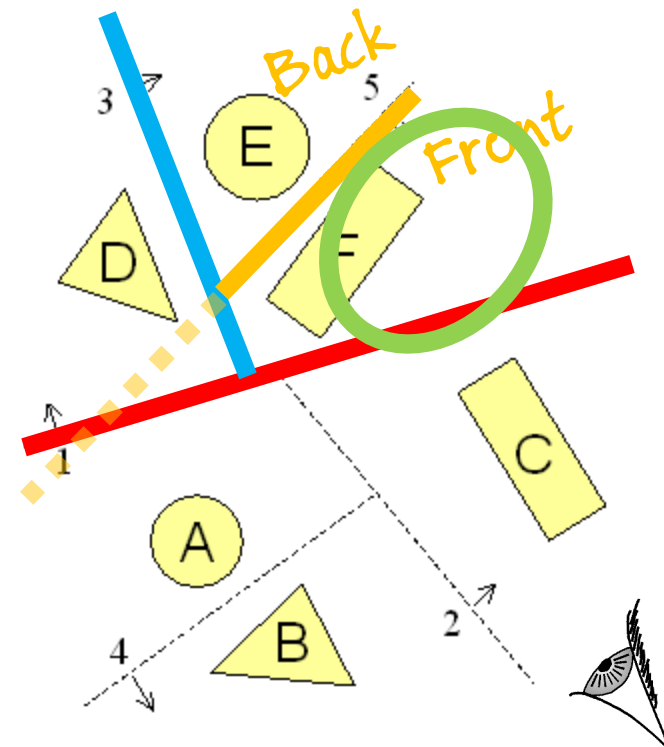
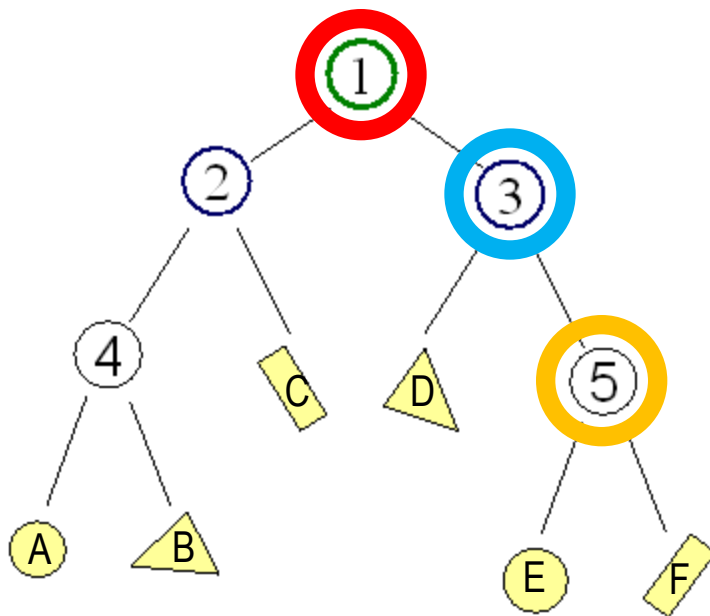
Drawing order: D, E,



# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron



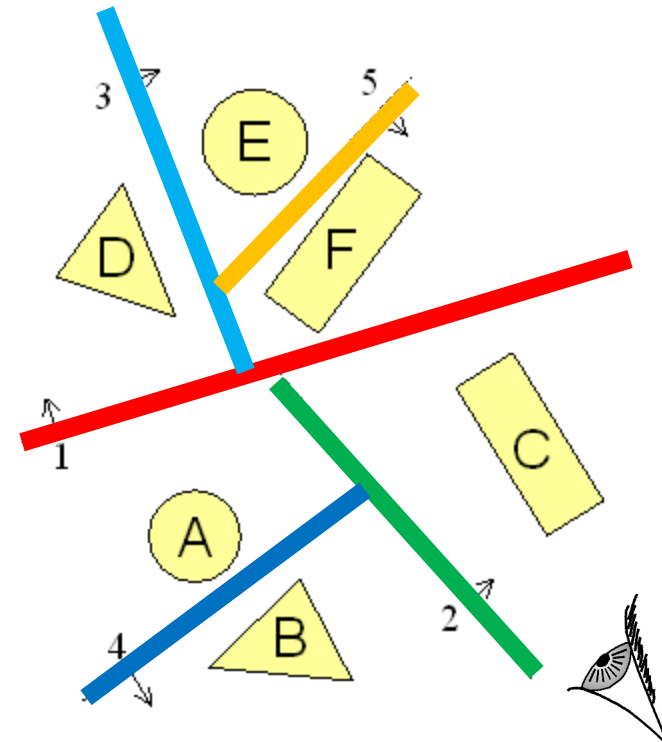
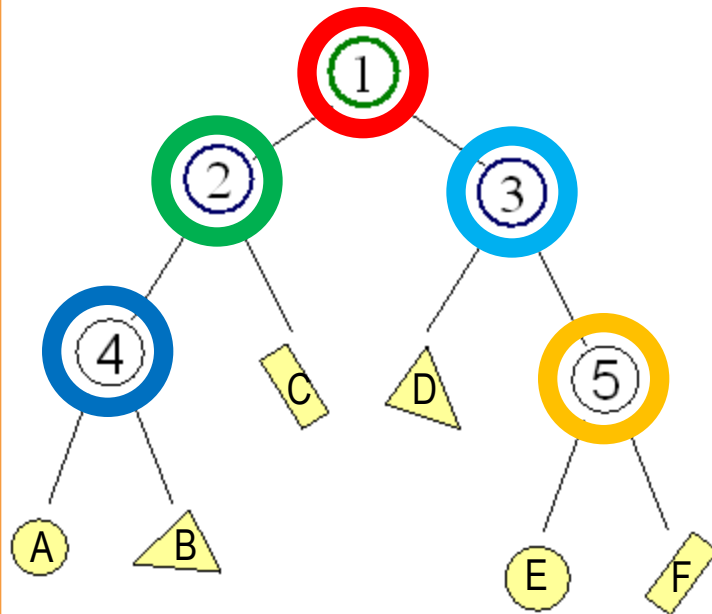
Drawing order: D, E, F



# Binary Space Partition (BSP) Tree



- Recursively partition space by planes
  - Every cell is a convex polyhedron



Drawing order: D, E, F, A, B, C



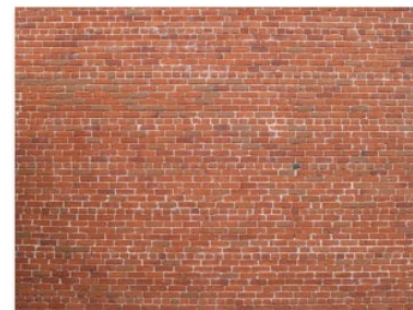
# From Vertices to Frame Buffer



# Command: draw these triangles!

## Inputs:

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



Texture map

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

Perspective projection transform **P**

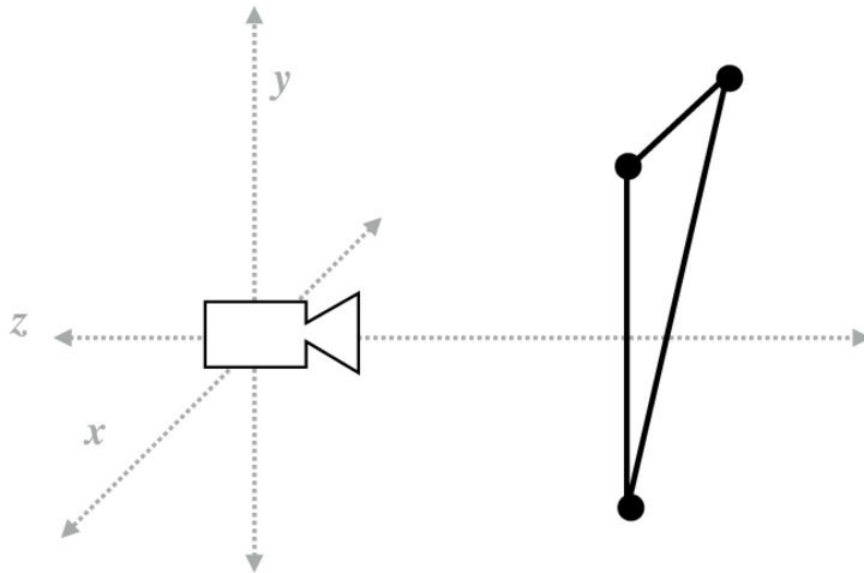
Size of output image (W, H)

Use depth test /update depth buffer: YES!



# Step 1:

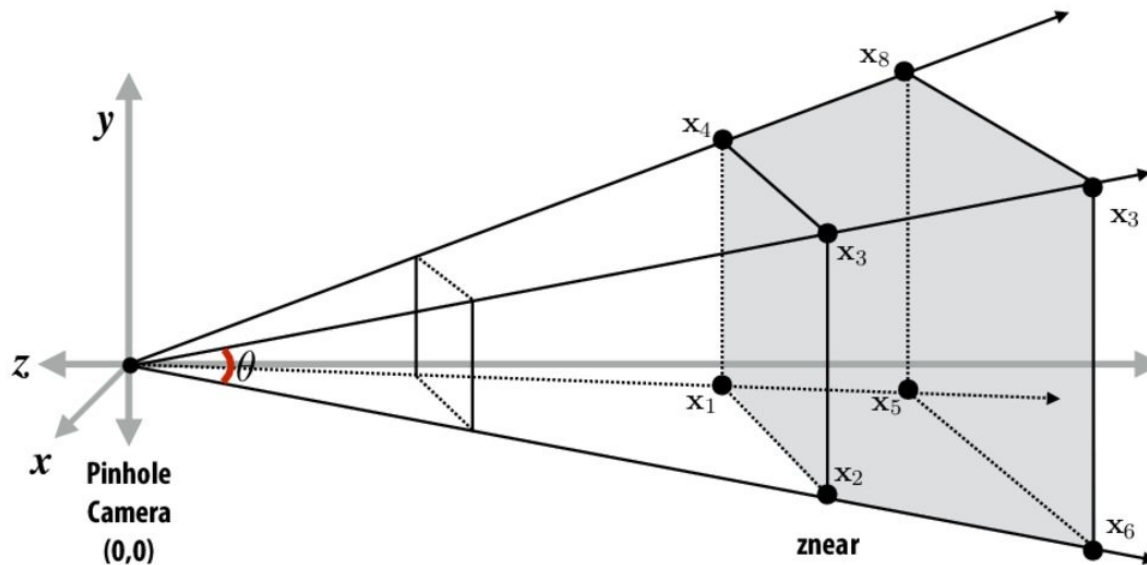
**Transform triangle vertices into camera space  
(apply modeling and camera transform)**



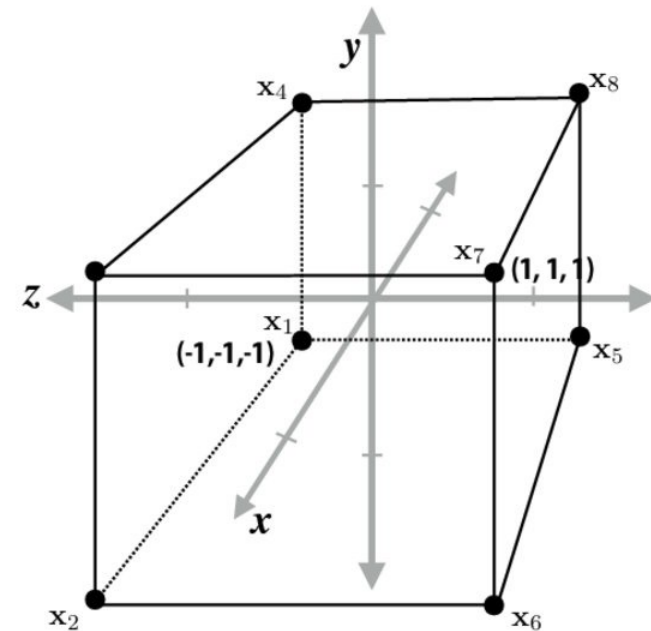


# Step 2:

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



Camera-space positions: 3D



Normalized space positions

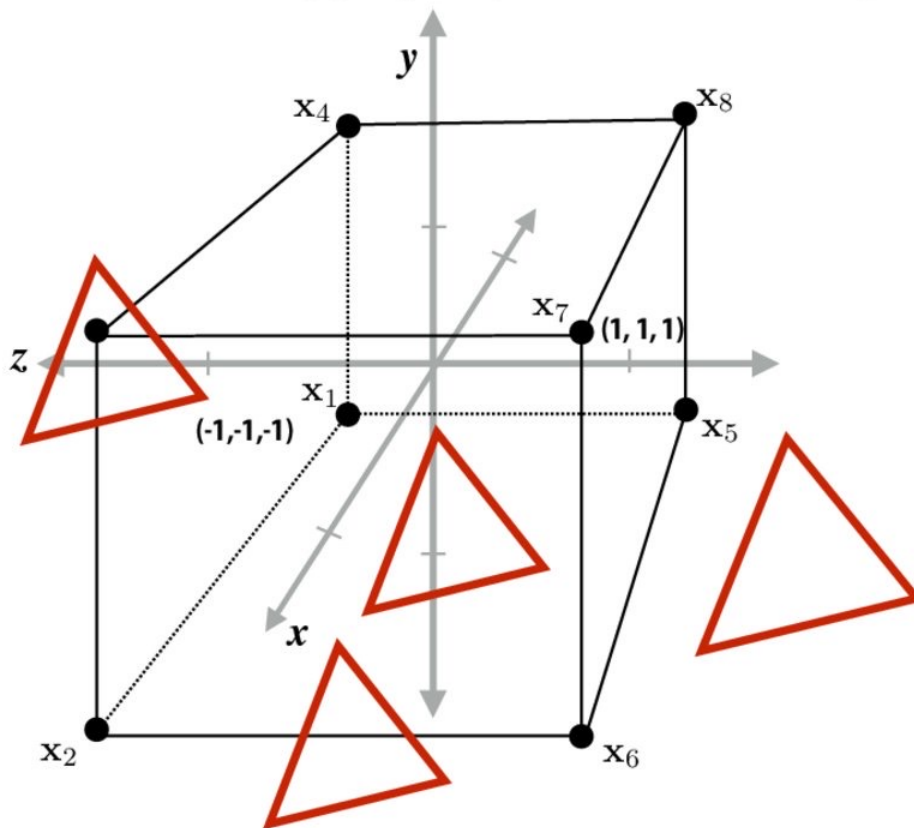
Note: I'm illustrating normalized 3D space after the homogeneous divide, it is more accurate to think of this volume in 3D-H space as defined by:

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

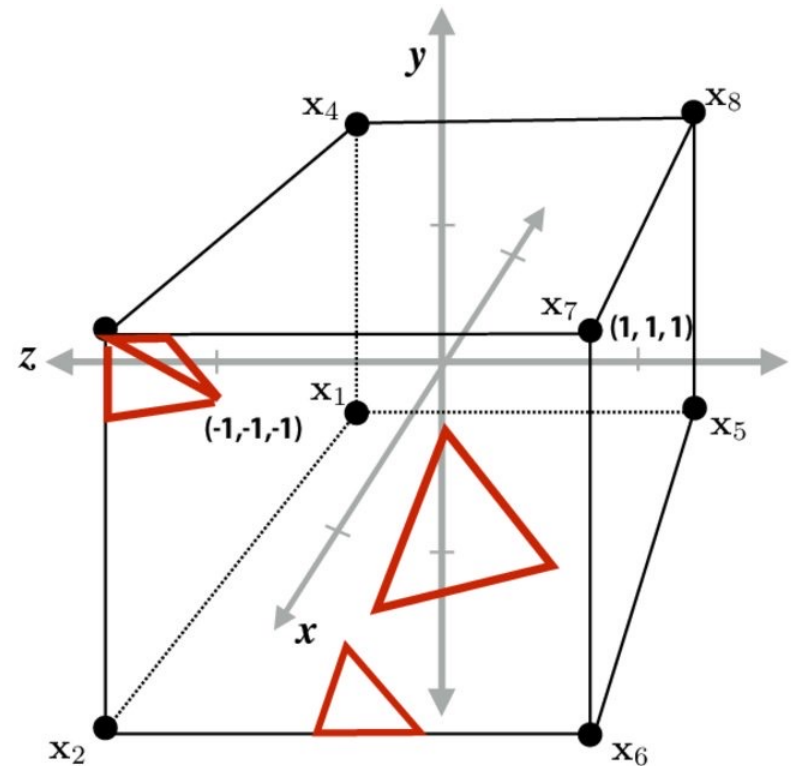


# Step 3: clipping

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



Triangles before clipping

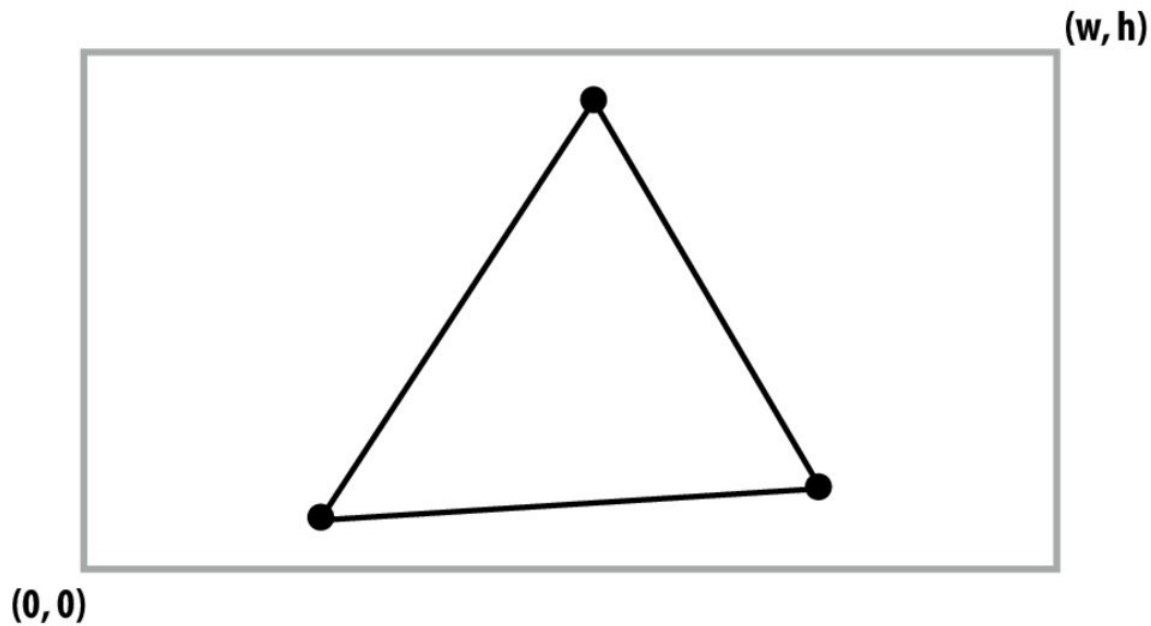


Triangles after clipping



# Step 4: transform to screen coordinates

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





# Step 5: setup triangle (triangle preprocessing)

Compute triangle edge equations

Compute triangle attribute equations

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

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

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

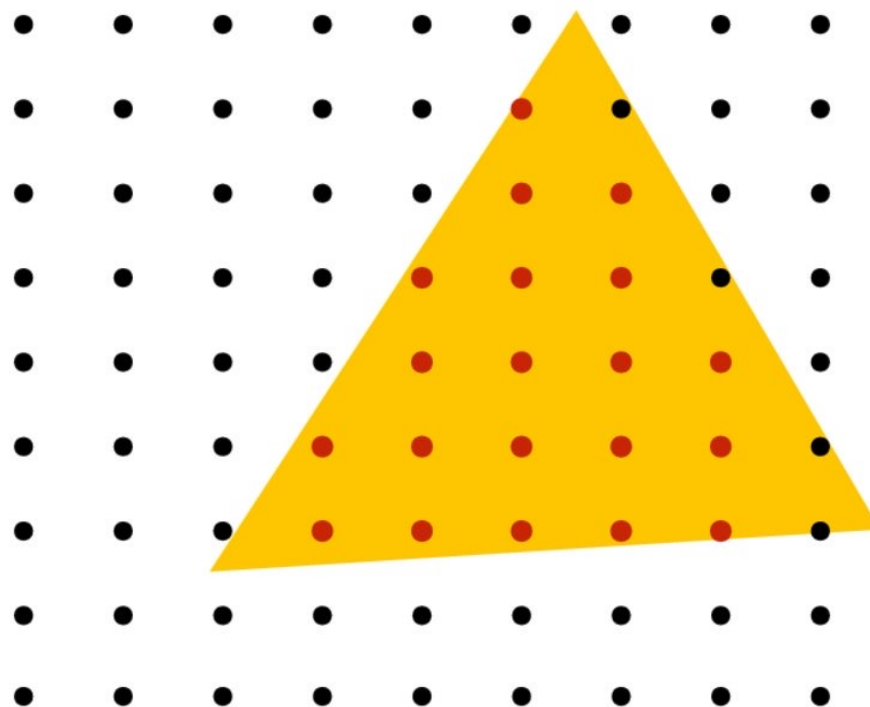
$$\frac{1}{\mathbf{w}}(x, y)$$

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



# Step 6: sample coverage

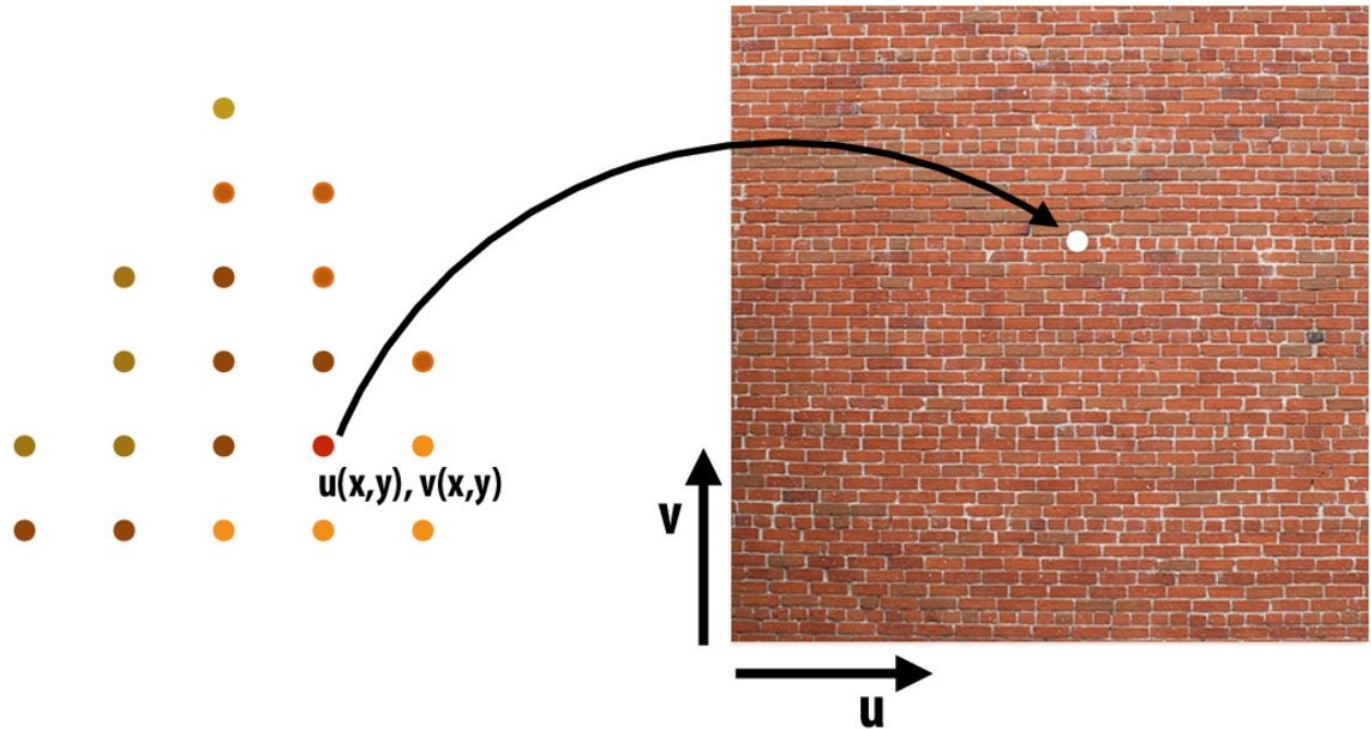
Evaluate attributes  $z, u, v$  at all covered samples





# Step 6: compute triangle color at sample point

e.g., sample texture map \*

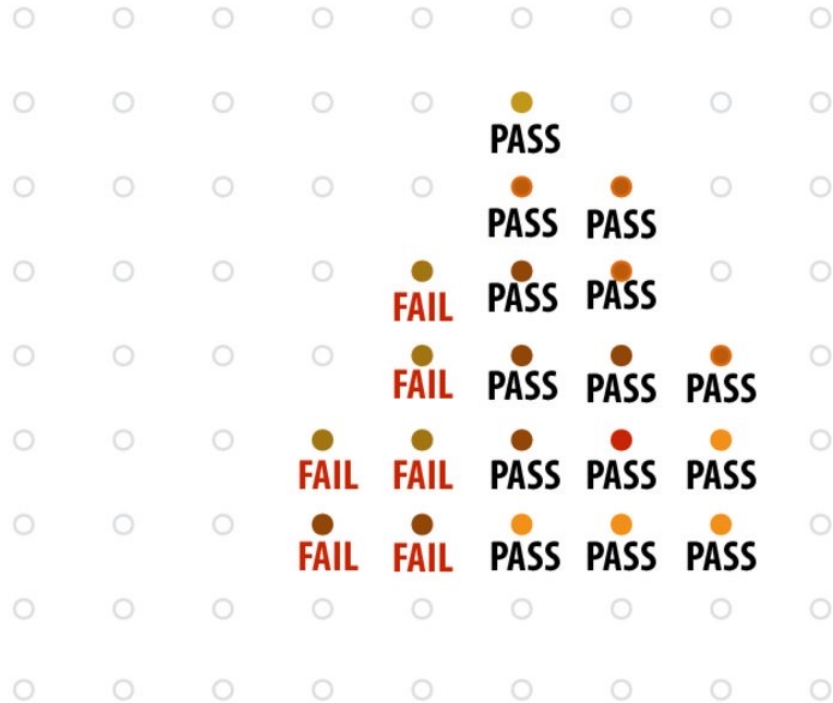


\* So far, we've only described computing triangle's color at a point by interpolating per-vertex colors, or by sampling a texture map. Later in the course, we'll discuss more advanced algorithms for computing its color based on material properties and scene lighting conditions.



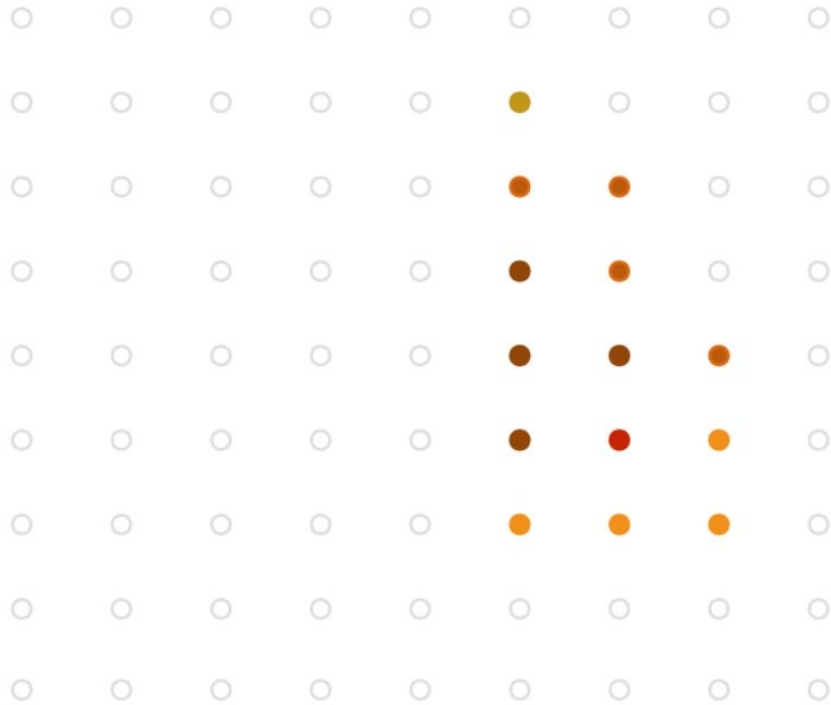
# Step 7: perform depth test (if enabled)

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!**



**Administrative**



Q&A



End