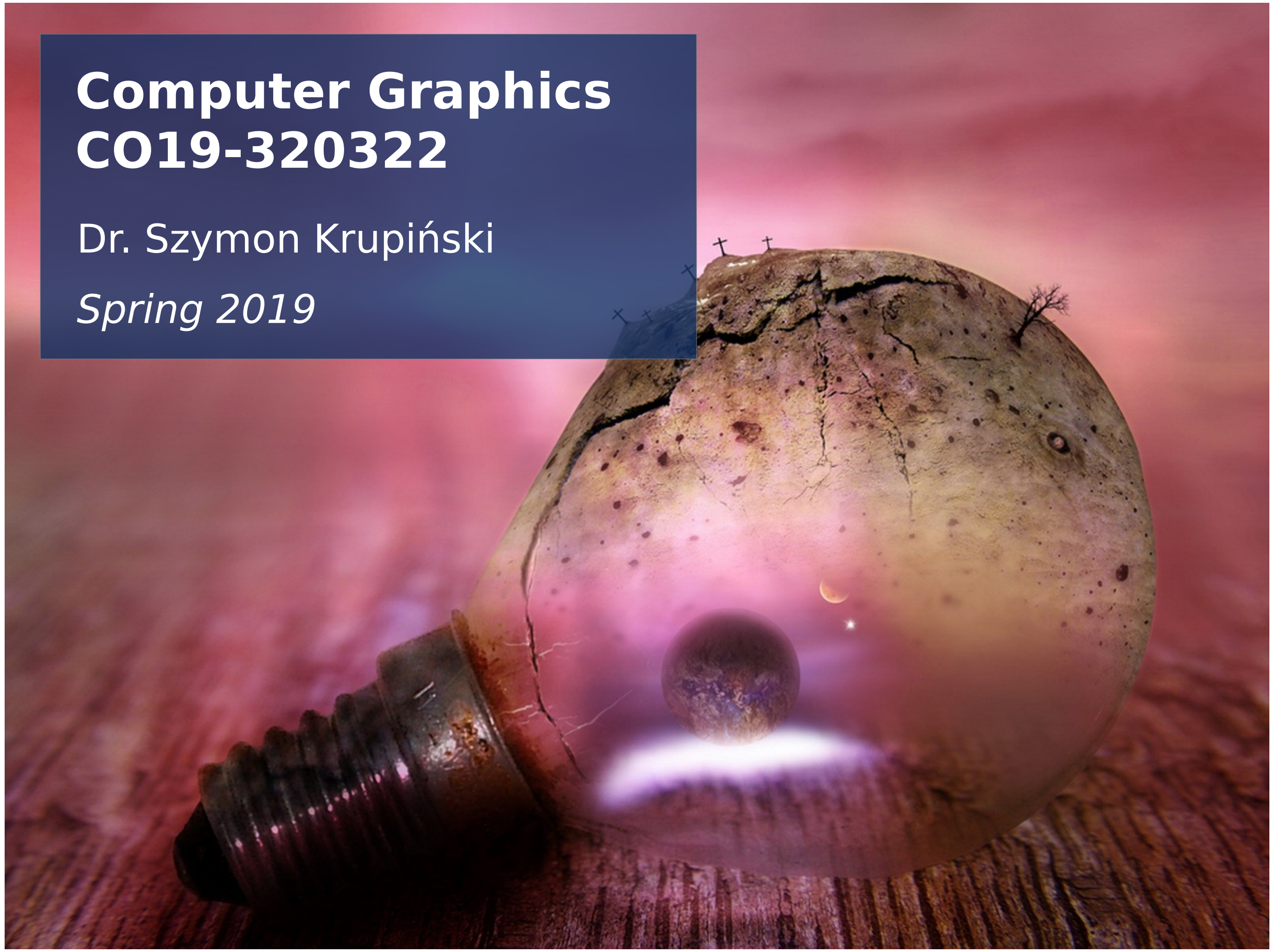
Lecture 9: Rasterisation 3

Computer Graphics
CO19-320322

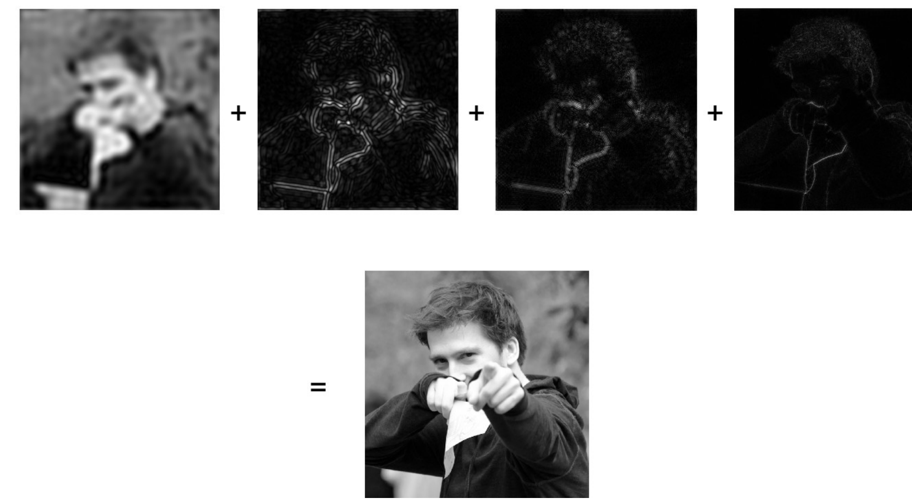
Dr. Szymon Krupiński

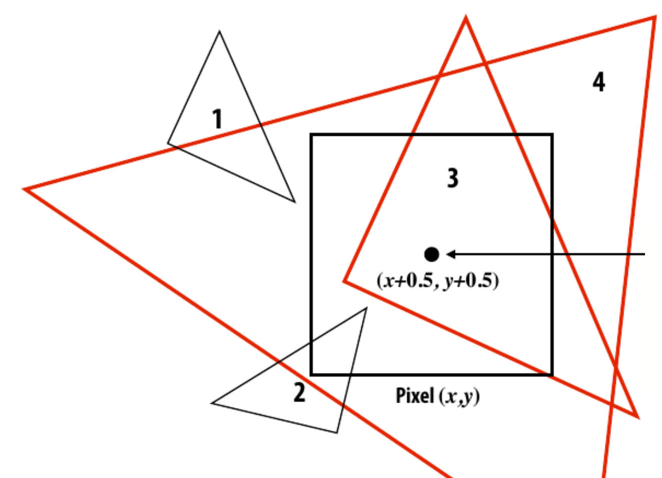
Spring 2019



In the last episode...

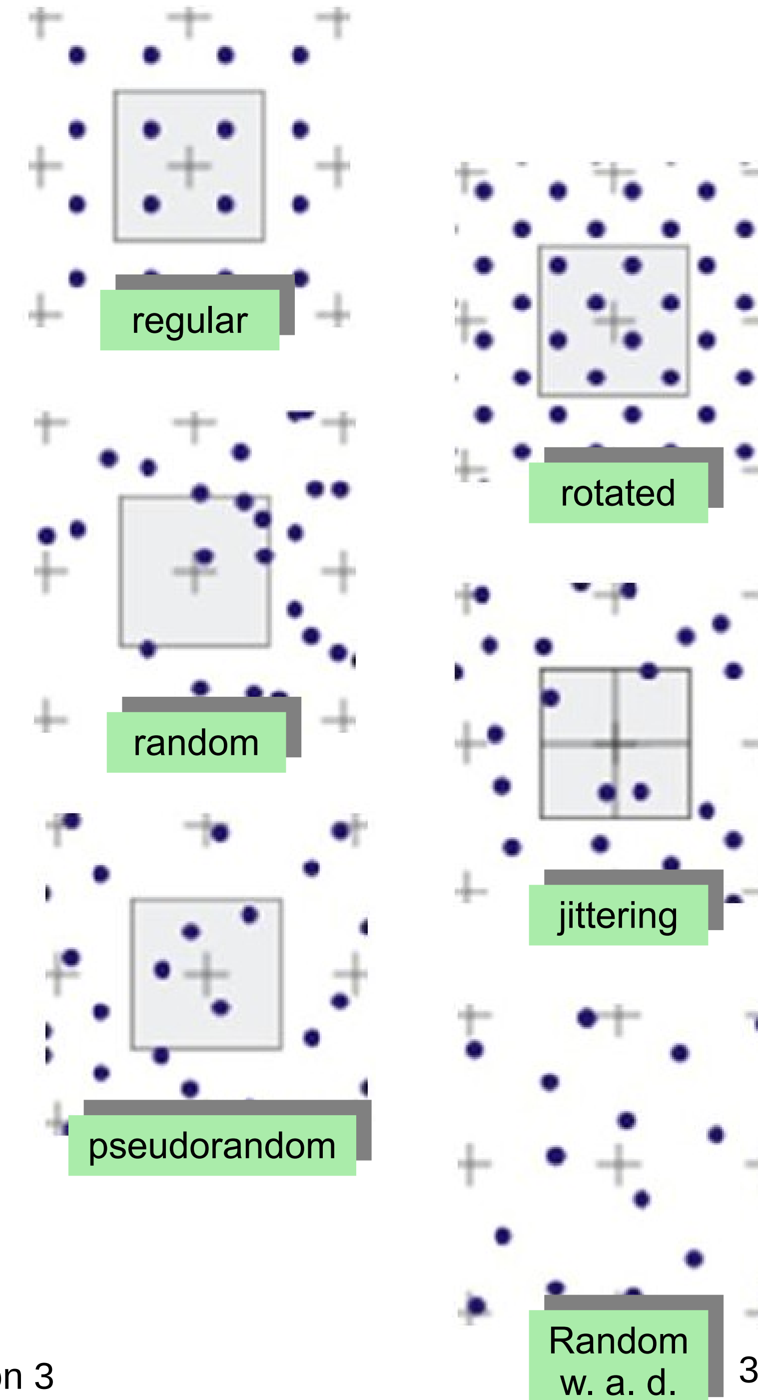
- Frequency components in images
- Triangles: how to deal with coverage?
- ...and how to do it better?
  - First answer: supersampling



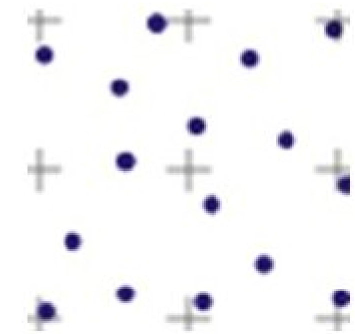


Stochastic supersampling

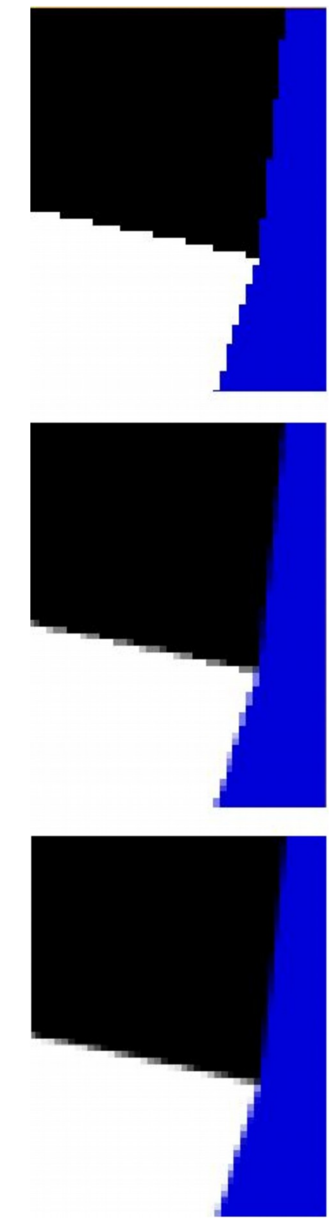
- How do we arrange our new sample points?
  - Regular
  - Rotated
  - Random
  - Jittering
    Instead of picking n points randomly from $[0,1]^2$, one partitions the unit square into n regions of equal measure and then chooses a point randomly from each partition.
  - Pseudorandom
    E.g. number generating sequence such as Holton seq.
  - Random with assured min. distance
    Using so called Poisson disk



Stochastic supersampling



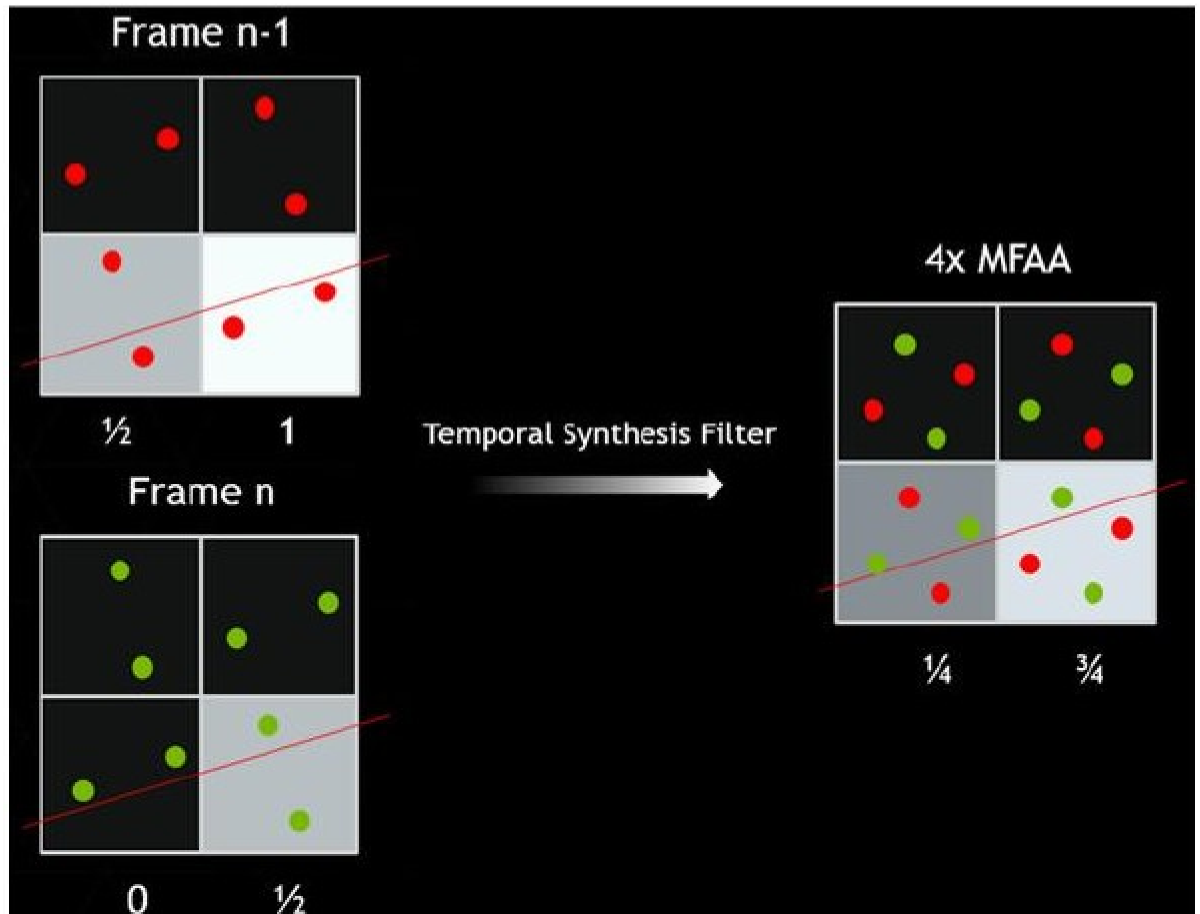
- Stochastic sampling helps to further reduce regular patterns leading to aliasing artifacts
- It’s a long road of improvements...
  - GeForce 2: introduction (oops, need memory!)
  - GeForce 3: use 5 or 9 sample masks but only 2 or 4 of these samples are used to compute the color
  - GeForce 8: masks suitable for coverage sampling, decoupled from color/shading
  - GeForce GTX 980: use temporal dimension, reuse samples from previous steps: MFSAA



Anti-aliasing

- Multi frame sampled anti-aliasing:

Sampling pattern is complementary in consecutive frames



Synthesis is a better estimation of the real signal

Anti-aliasing

- How does it look in OpenGL?
  - Some solutions are already programmed
  - `glEnable(mode)`

    `GL_POINT_SMOOTH`

    `GL_LINE_SMOOTH`

    `GL_POLYGON_SMOOTH`
  - Coverage computed by supersampling and stored in the alpha channel (= transparency, we will learn more about it soon)
  - Blending needs to be enabled
    `glEnable(GL_BLEND);`

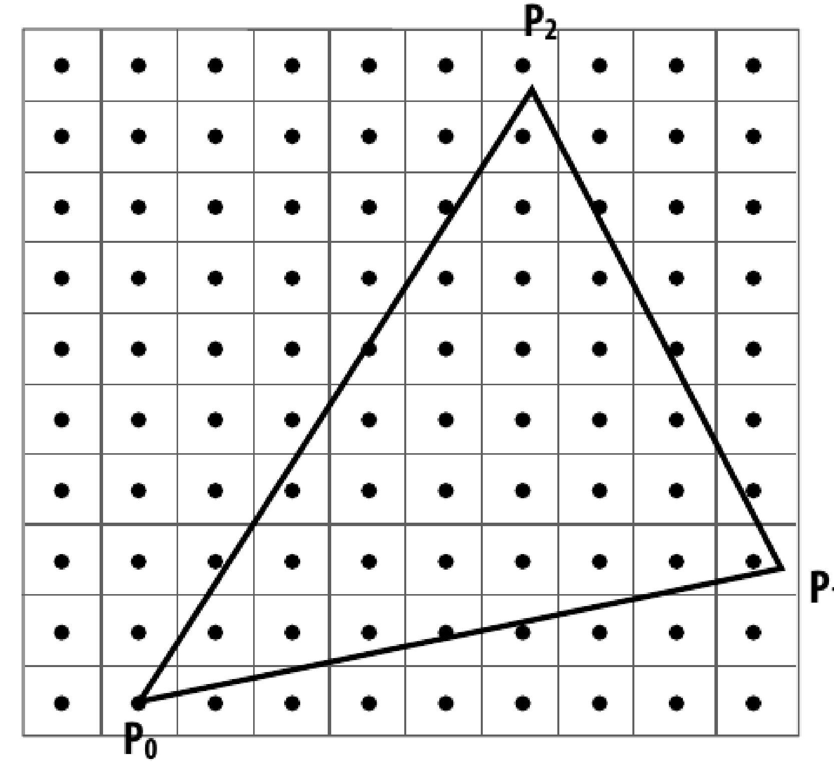
Other ideas

- Post filtering
  - Simple signal processing idea: if I want to remove spurious high frequency information, I can just apply a low-pass filter
  - Equivalent to smoothing/blurring/local averaging
  - Applied equally to the entire scene
- In OpenGL: use accumulation buffers for the color information
  - Can lead to loss of acuracy of the stored colors
  - Take data from read buffer, when done, just transfer to the framebuffer for displaying
  - `glAcum( operation, value ): GL_ADD, GL_MULT`, ...

In or out?

- How do we actually decide if a point is in triangle or not..?
  - Again, the simplicity of triangle and mathematics comes in handy
  - For any sampling point (x,y) and a triangle of vertices P_i, P_{i+1} and P_{i+2}

$$
\begin{aligned}
P_i &= (X_i, Y_i) \\
\\
dX_i &= X_{i+1} - X_i \\
dY_i &= Y_{i+1} - Y_i \\
\\
E_i(x, y) &= (x - X_i)\, dY_i - (y - Y_i)\, dX_i \\
&= A_i\, x + B_i\, y + C_i \\
\\
E_i(x, y) &= 0 \ : \text{point on edge} \\
&> 0 \ : \text{outside edge} \\
&< 0 \ : \text{inside edge}
\end{aligned}
$$



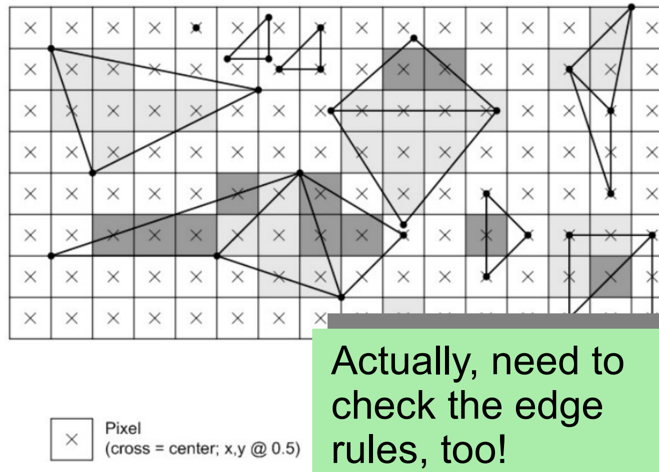
- Why A, B, C?
  - We can optimise incremental calculations while switching to the adjacent sampling point!

$$
\begin{aligned}
dE_i(x+1, y) &= E_i(x,y) + dY_i = E_i(x,y) + A_i \\
dE_i(x, y+1) &= E_i(x,y) + dX_i = E_i(x,y) + B_i
\end{aligned}
$$

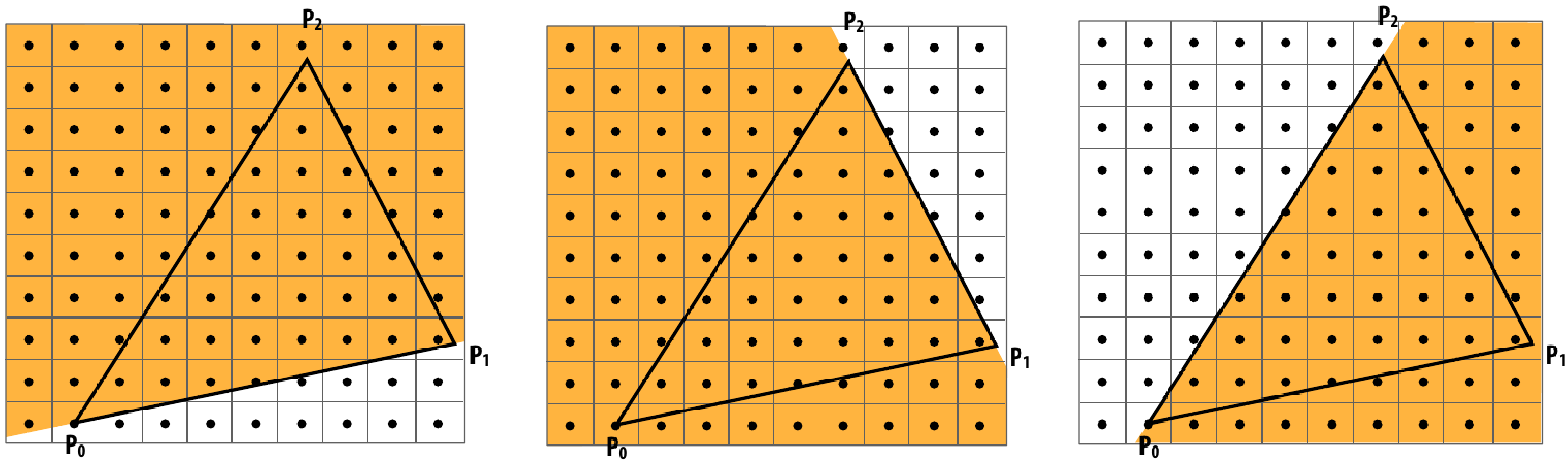
  - We saved a couple of operations per point...

In or out?

- Check if the point is on the "good" side of every edge
- Decide:
  `inside(x,y) = Ei(x,y) < 0 && Ei+1(x,y) < 0 && Ei+2(x,y) < 0;`

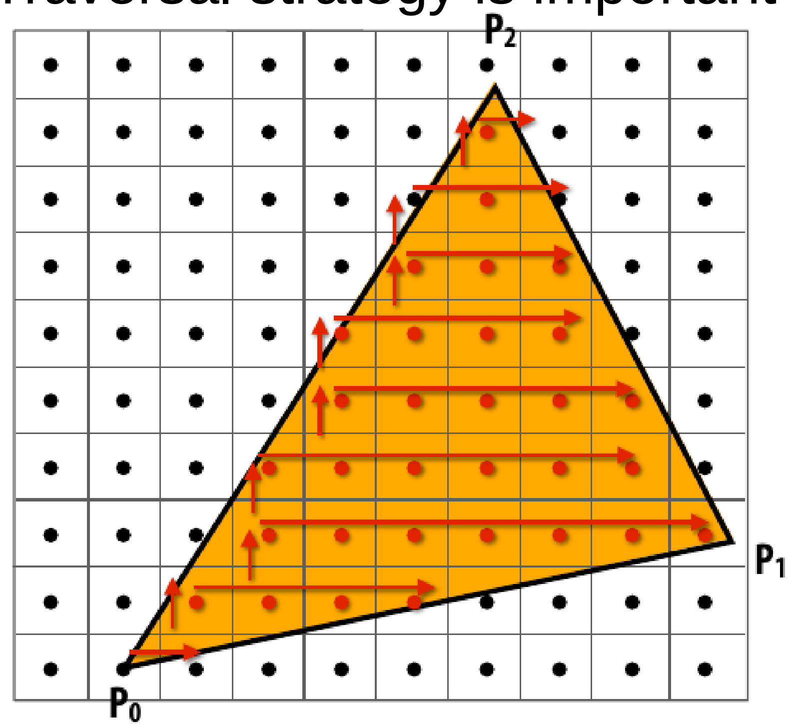


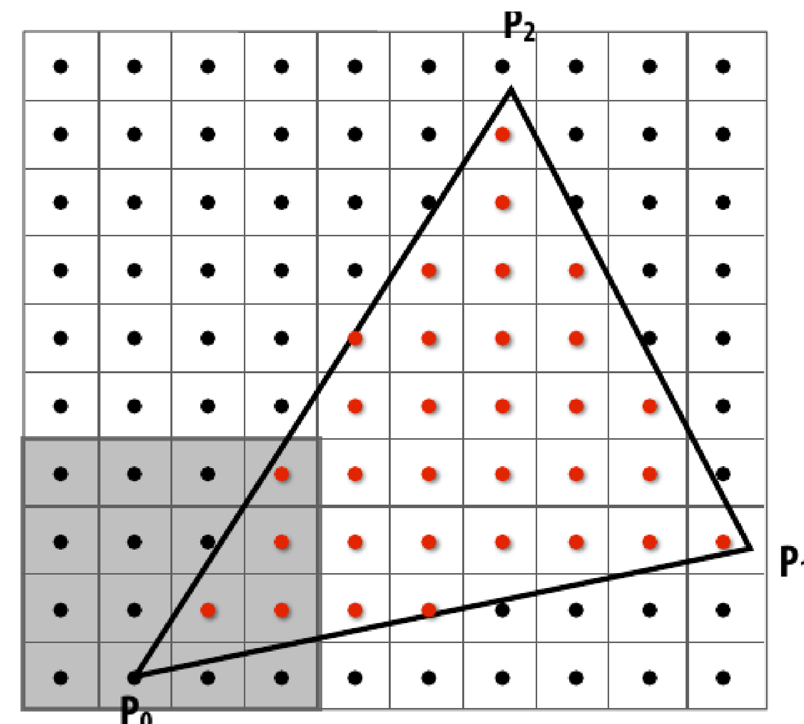
Actually, need to check the edge rules, too!



In or out?

- Traversal strategy is important for optimisation!



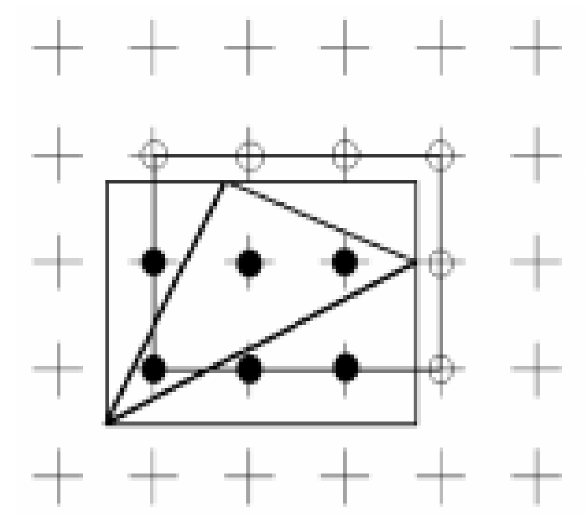


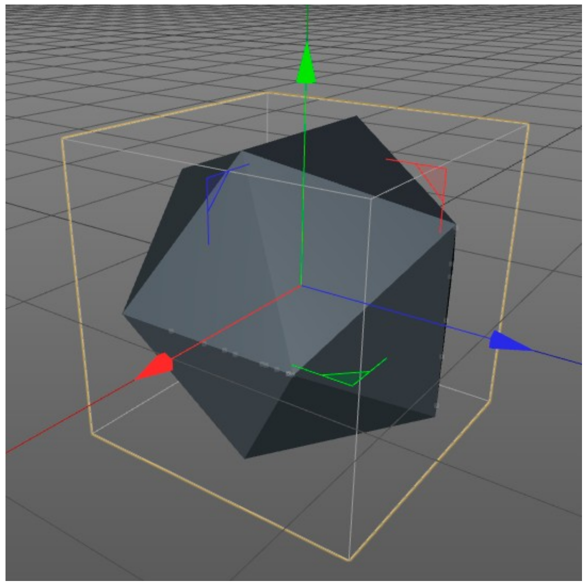
- All in good order...

- Today, parallel execution is often more important than math optimisation → **tiled** triangle traversal
- Allows to optimise elsewhere: initiate by checking if the entire block is not “in” or “out” and cut computation if known

Bounding box

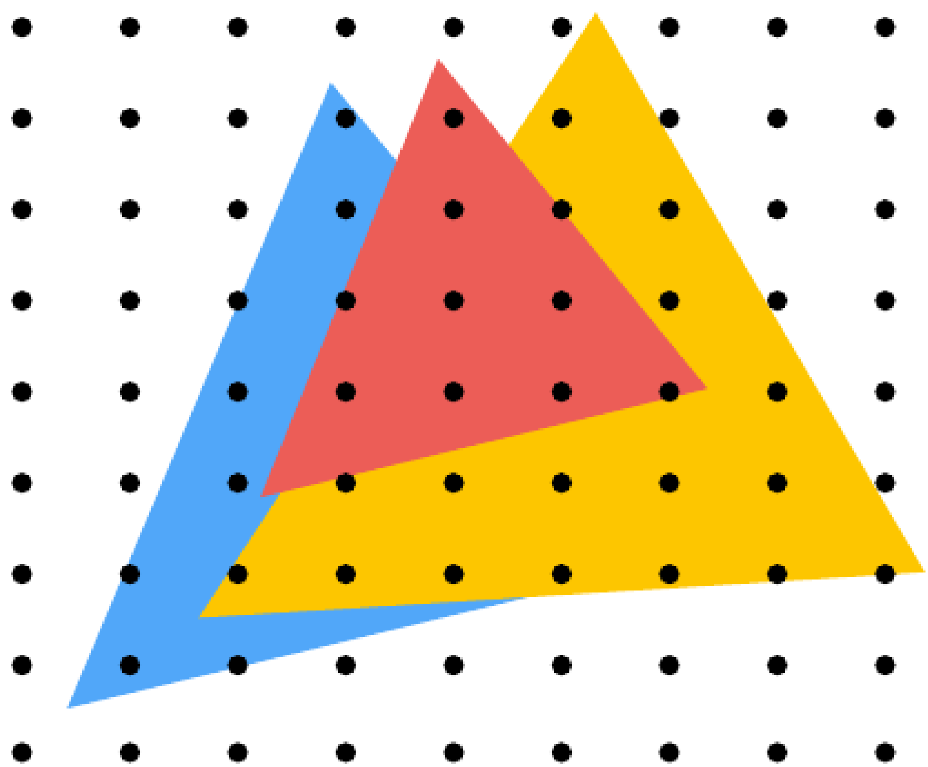
- Another related trick to avoid unnecessary calculations
- It's inefficient to check all pixels on the screen to verify the coverage of one triangle
- Instead, calculate a bounding box – a minimal upright rectangle which will enclose the triangle (shape, object, …)
- [round the coordinates to the nearest integer – efficient left/top edge filter!]
- Check only the pixels in this selected region
- Concept used also at other stages, like collision detection in game, etc.



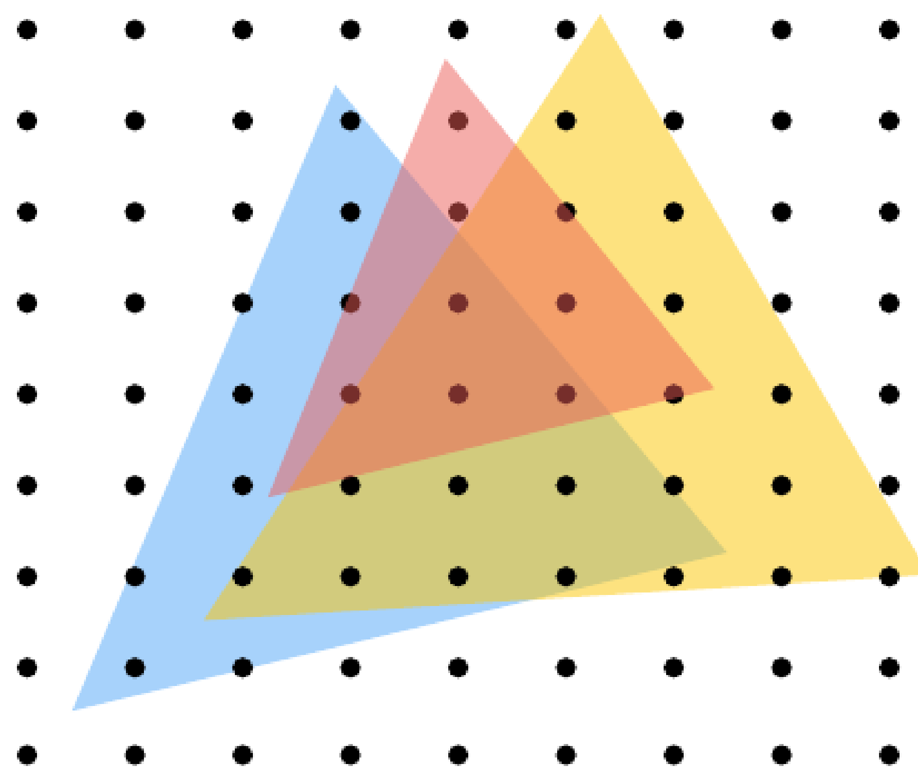


Occlusion

- We know which triangle covers which pixel. But if there are many, which one should I draw?



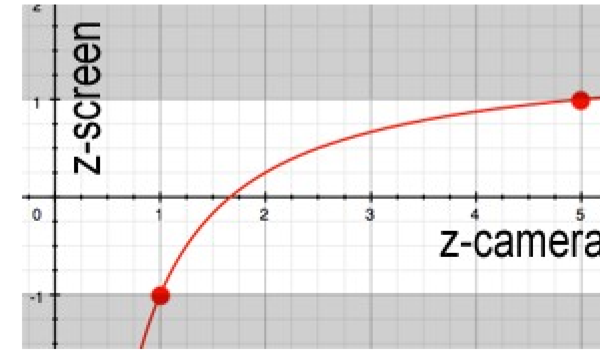
Opaque Triangles



50% transparent triangles

Depth information

$$\begin{bmatrix} x'w' \\ y'w' \\ z'w' \\ w' \end{bmatrix} = \begin{bmatrix} h & 0 & 0 & 0 \\ 0 & h & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$



- Time to use the numbers from the depth buffer created during projection calculations!
- Interpolate from the transformed vertices to get the depth at p=(x,y)

That’s how a z-buffer could look like in a complex scene: **black** represent the nearest objects and **white** the clipping distance

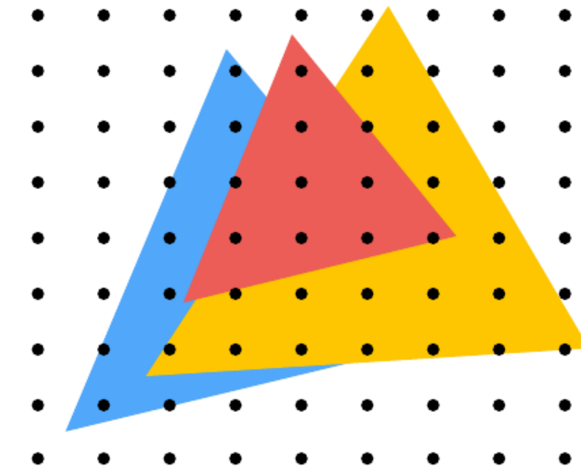
Depth information

- Let's test a simple algorithm

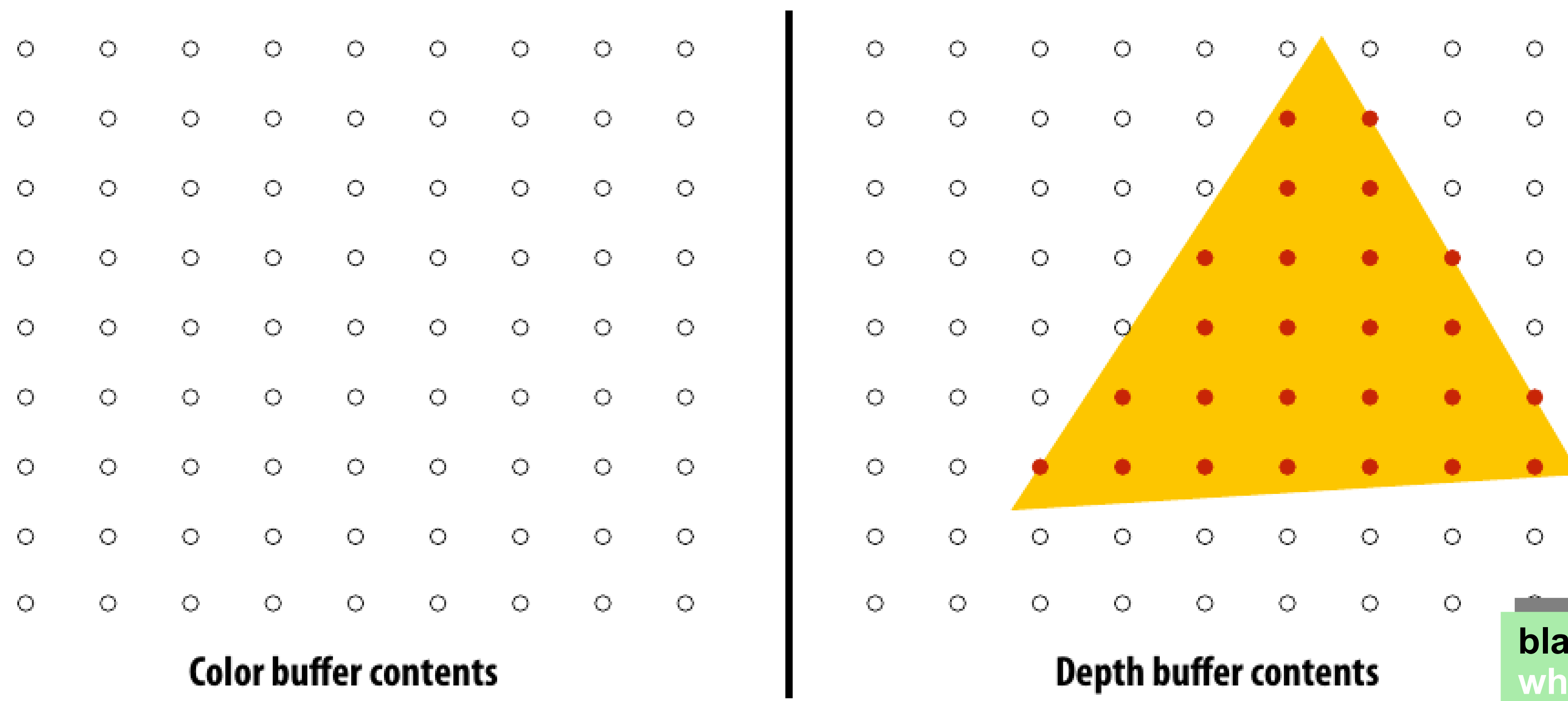
```
bool pass_depth_test(d1, d2) {
   return d1 < d2;
}

depth_test(tri_depth, 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_depth;     // update zbuffer
    color[x][y] = tri_color;       // update color buffer
  }
}
```

We are working with two buffers:
- z-buffer
- color buffer

Occlusion and z-buffer



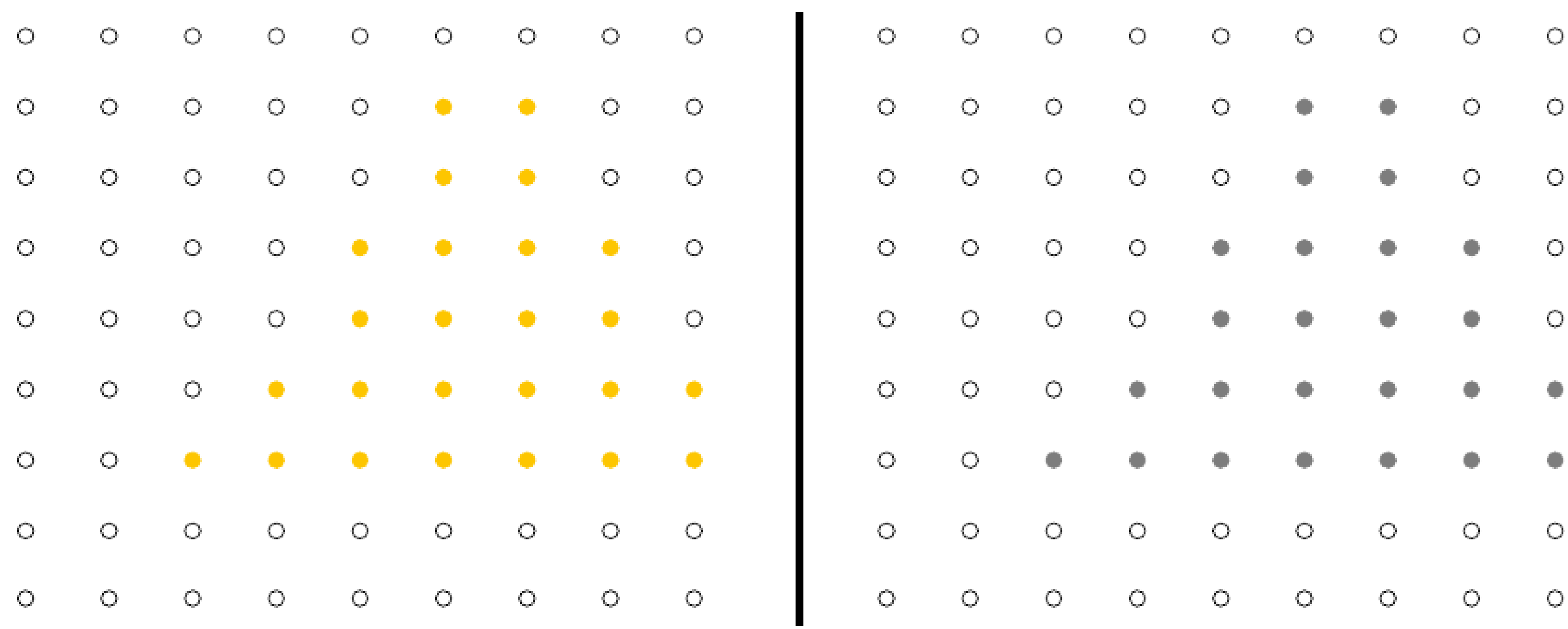
- Let’s deal with opaque triangles first
  - yellow triangle’s depth := 0.5



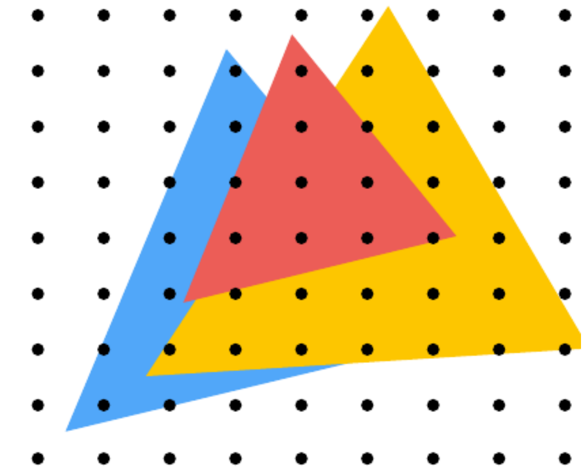
black : nearest
white : farthest
red: passed depth test

Occlusion and z-buffer

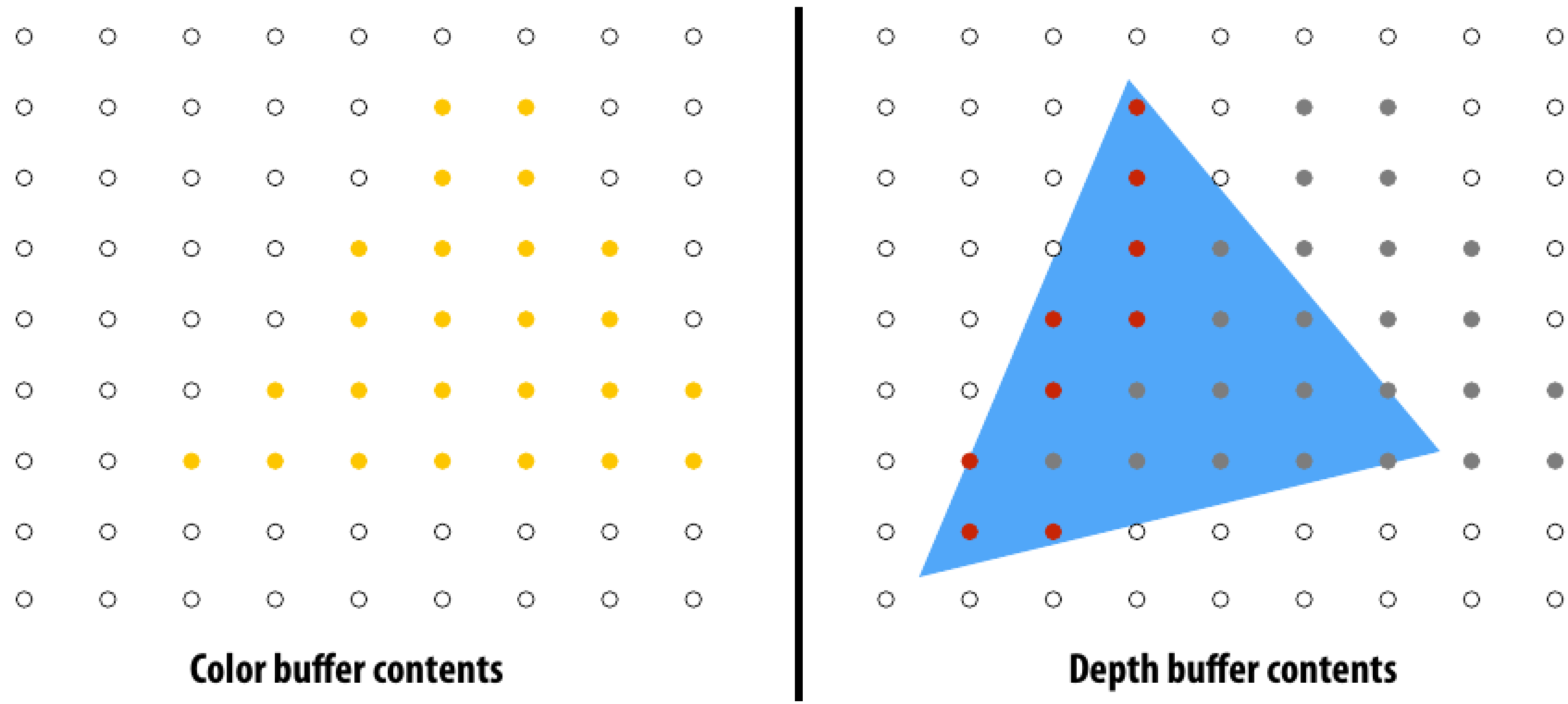
- Result 1



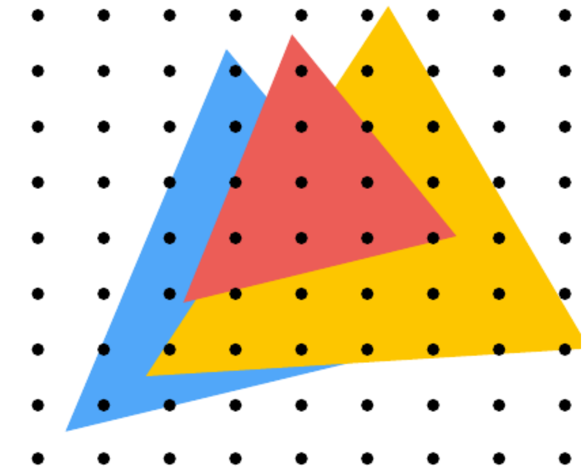
Occlusion and z-buffer



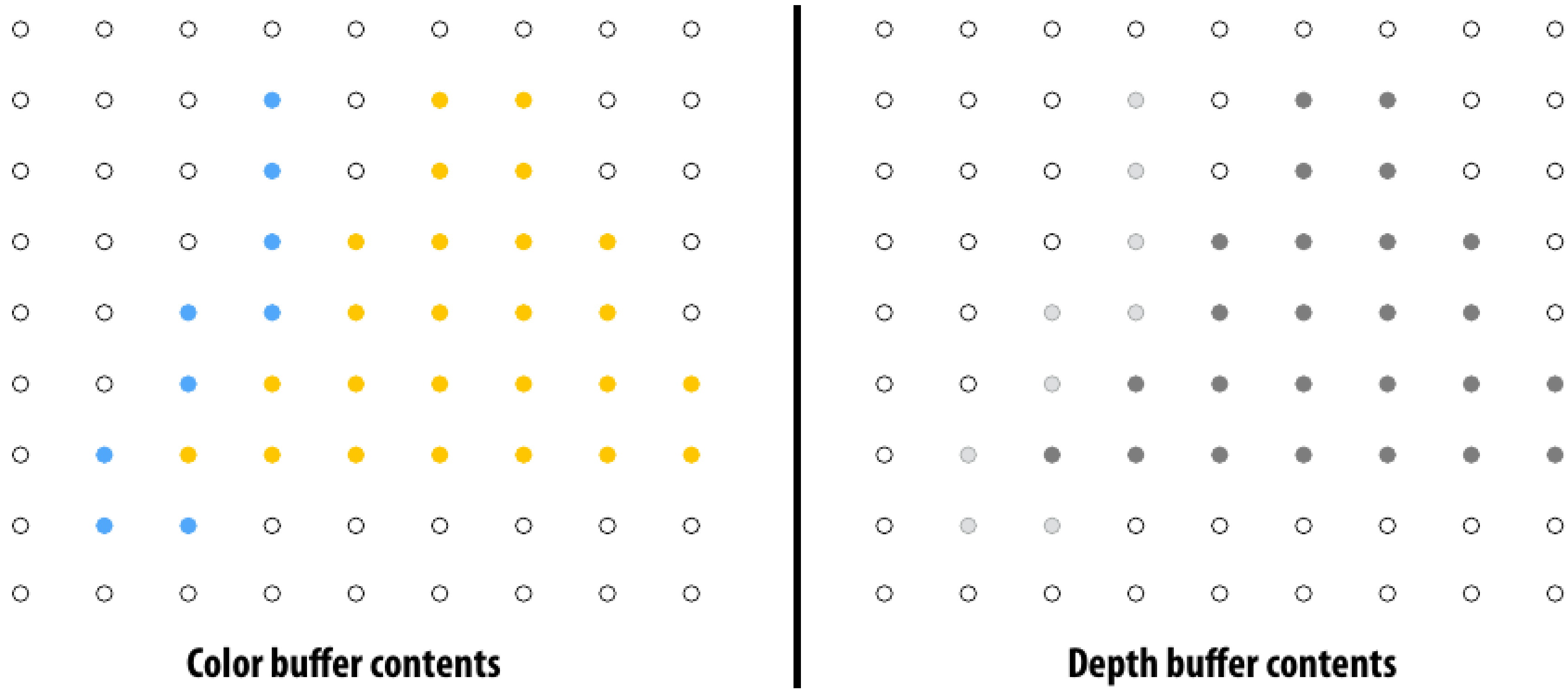
- Enters the blue triangle (z:=0.75)



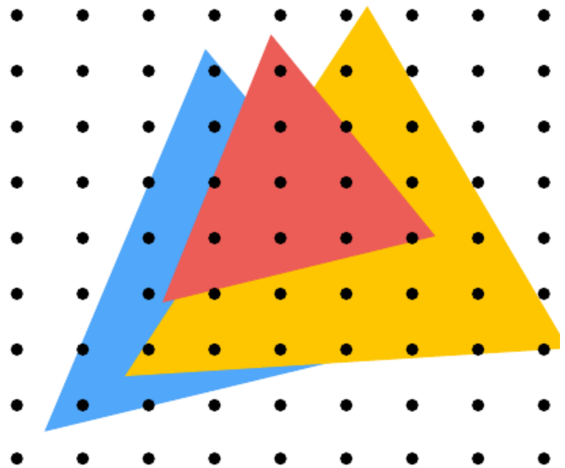
Occlusion and z-buffer



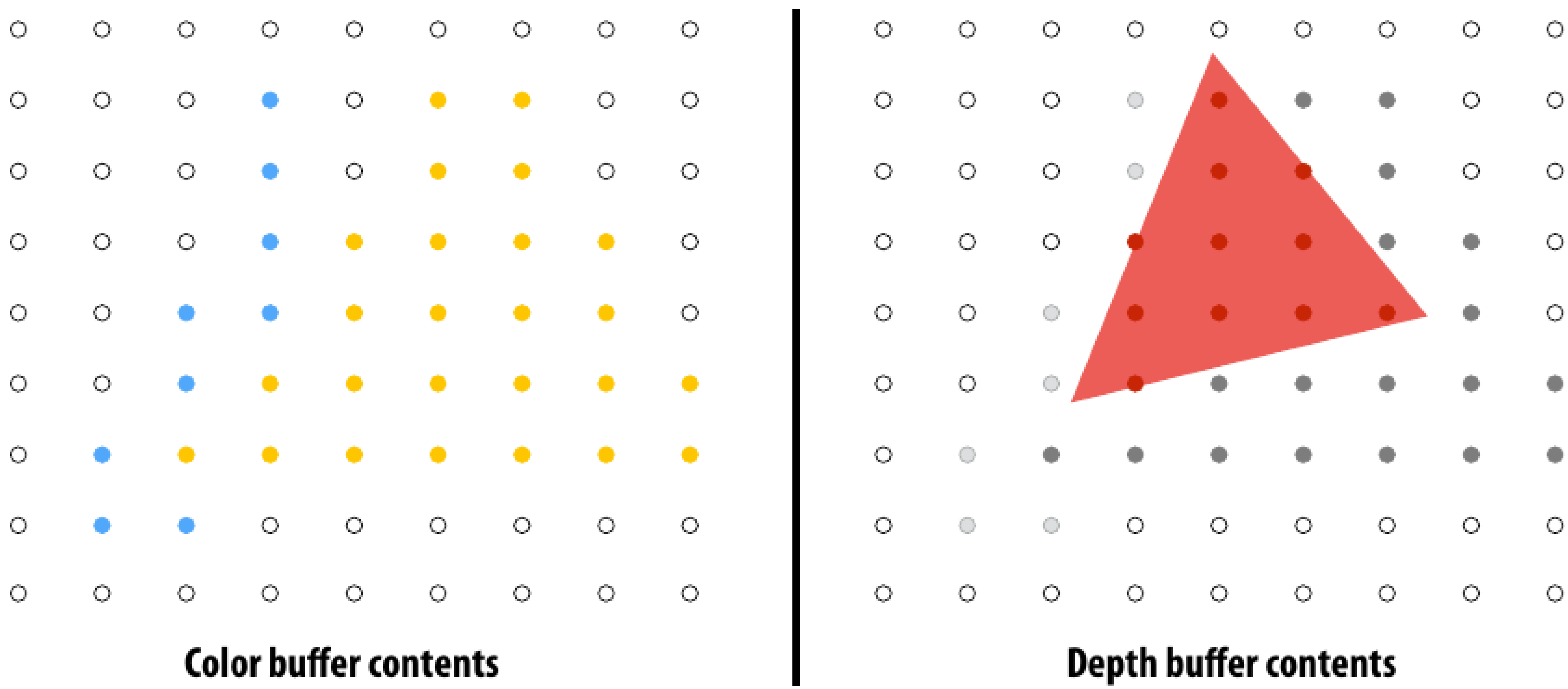
- Result 2



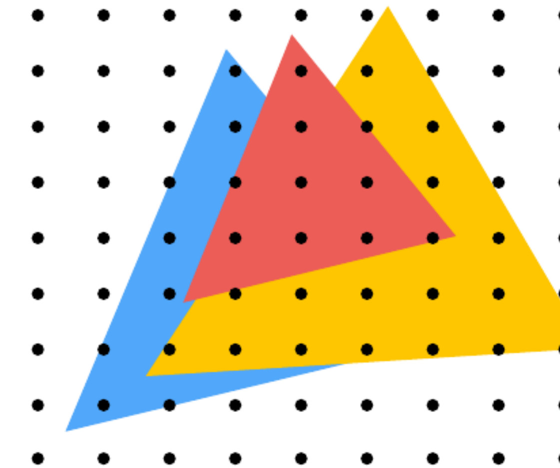
Occlusion and z-buffer



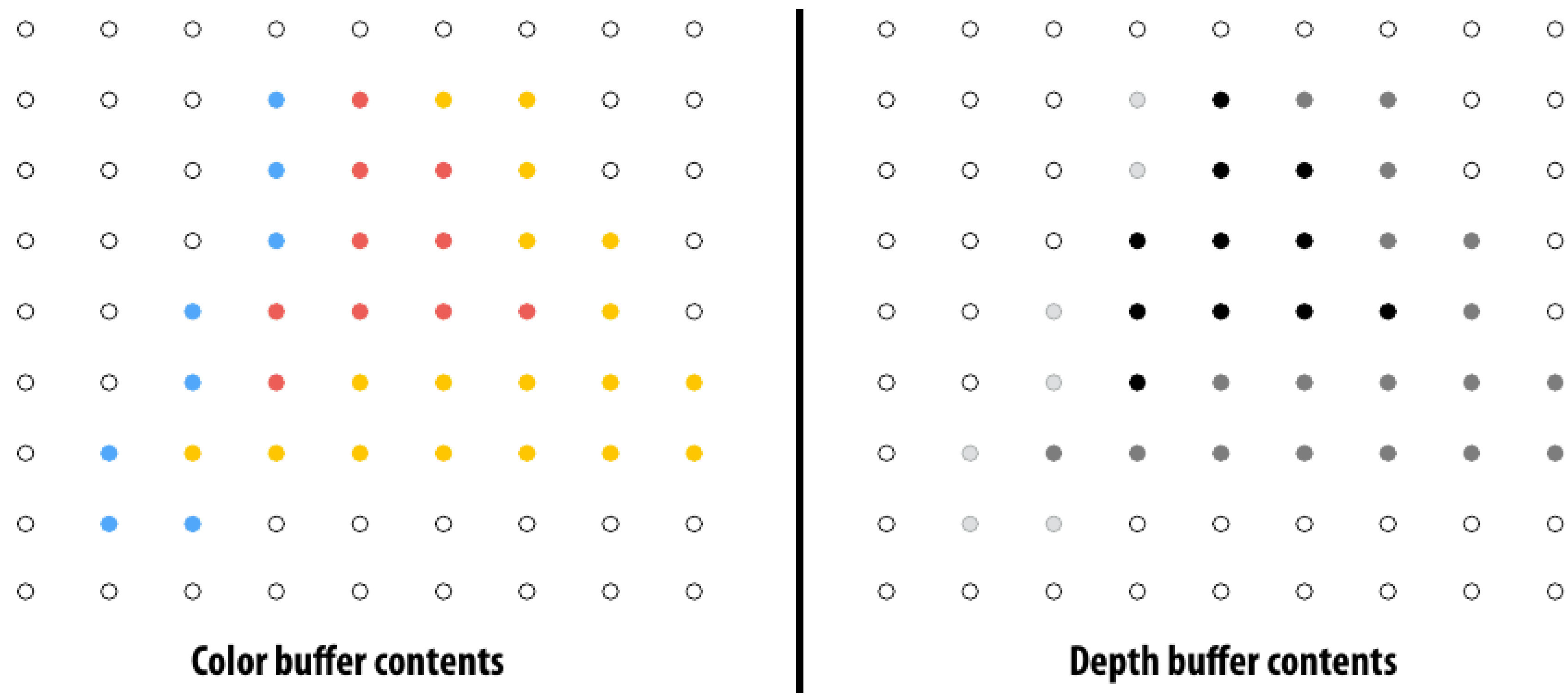
- In comes the red triangle (z:=0.25)



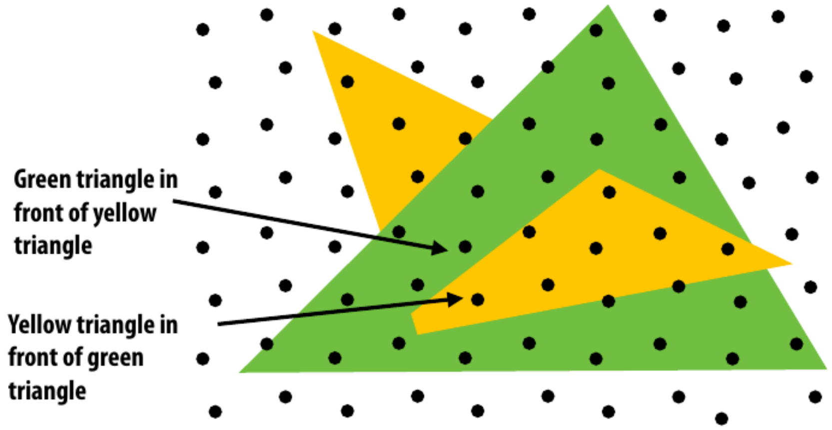
Occlusion and z-buffer



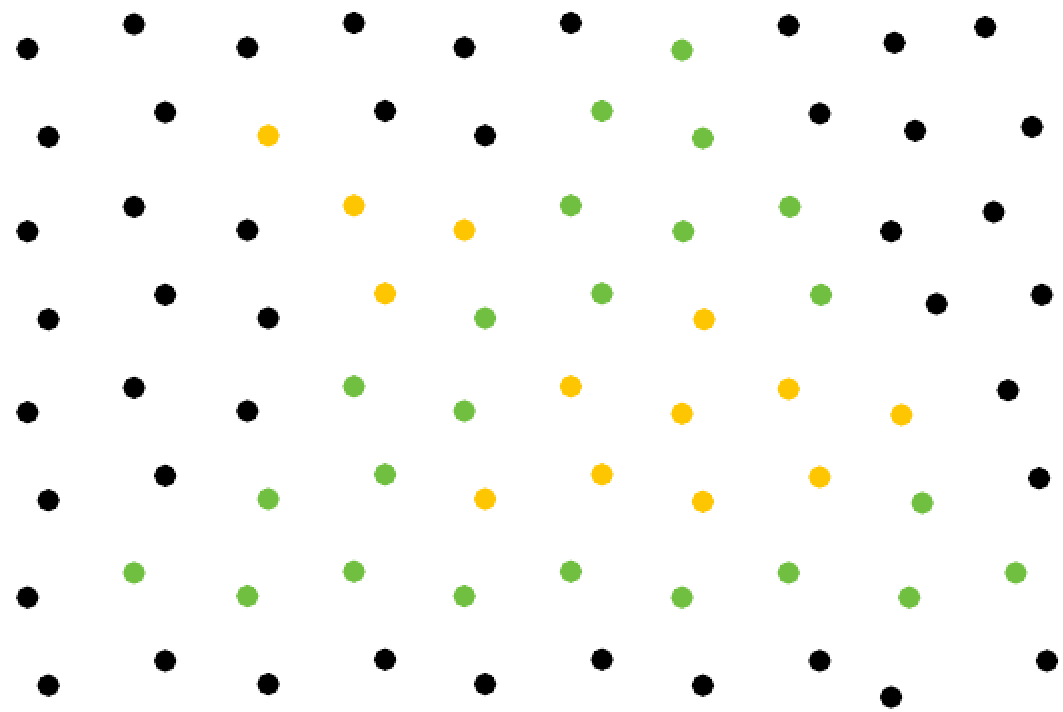
- Result 3



Intersecting geometries

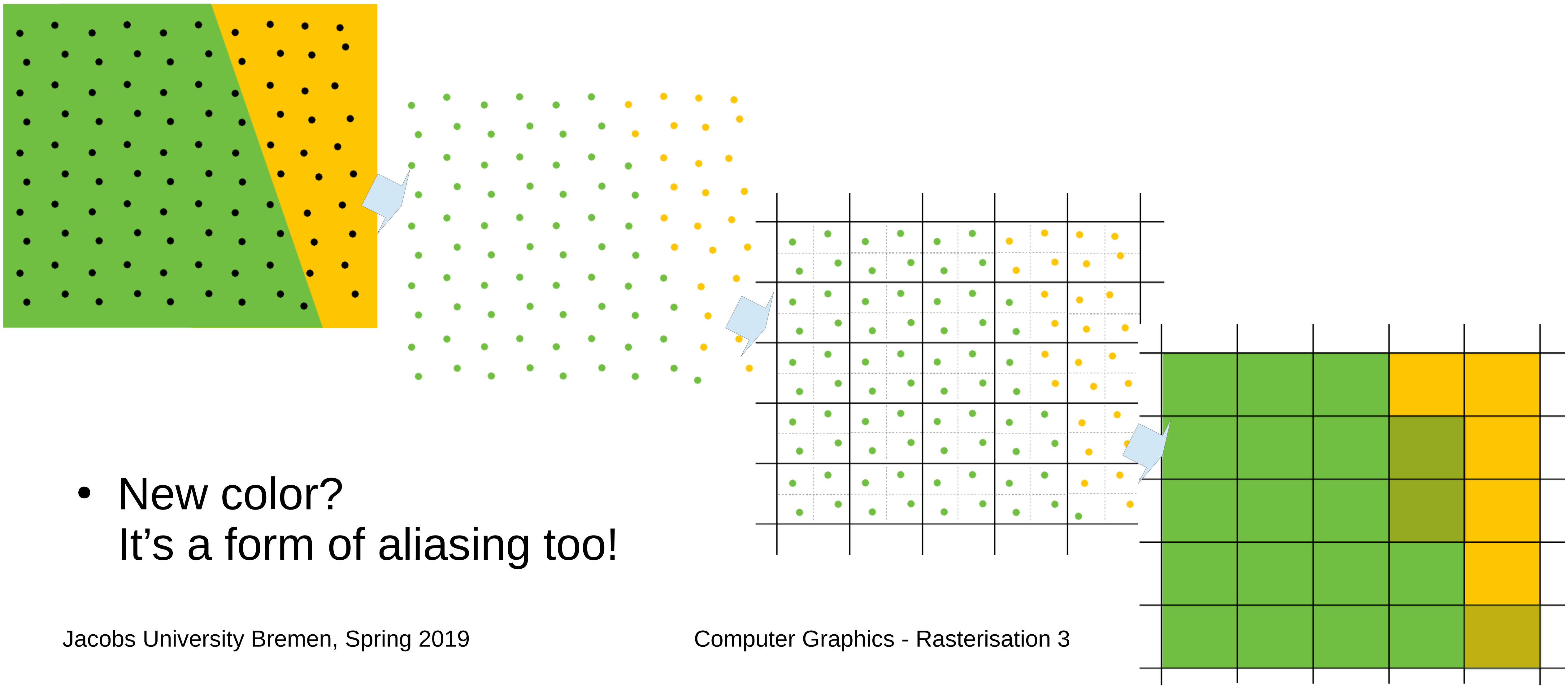


- Would it still work correctly if we have angled and “clashing” geometry?
  - Yes, for now we’re checking depth at each point, so it would be distinguish which triangle is visible at given point
  - Result



Supersampling

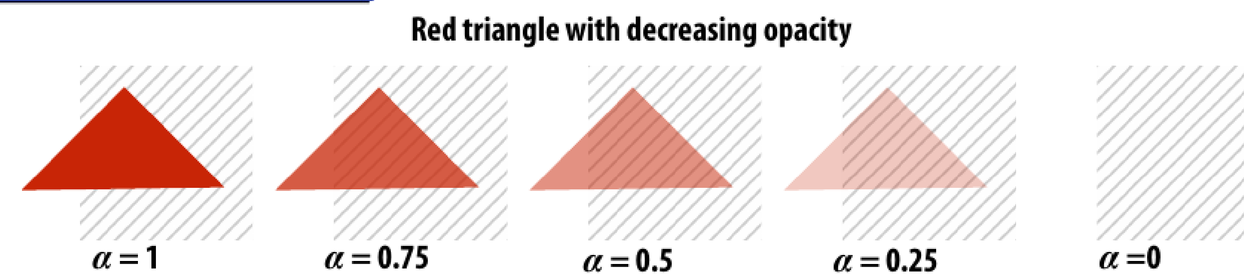
- Would supersampling make any difference?

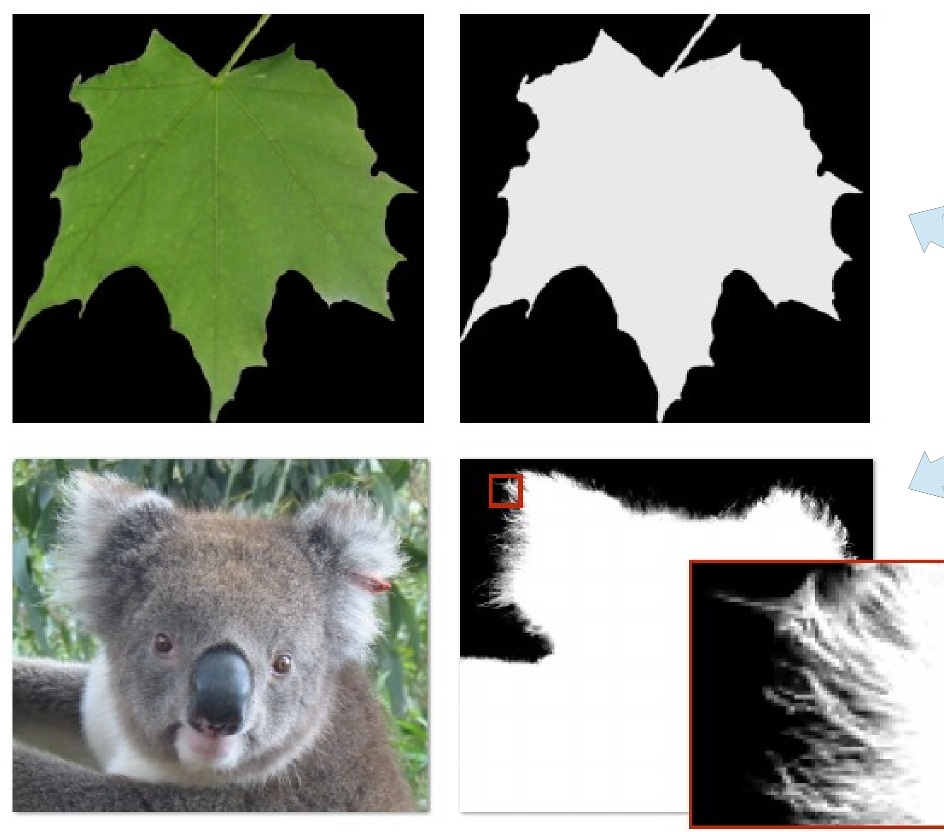


- New color?
  It’s a form of aliasing too!

Transparency

- Concept of α
  - $\alpha = 1$ – object fully opaque
  - $\alpha = 0$ – object fully transparent
- Stored as an additional color layer in RGBA mode

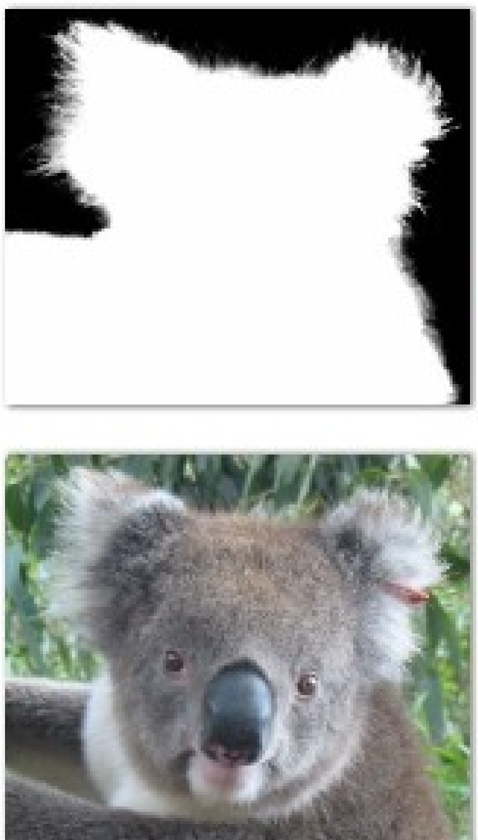




Alpha layer.
(can also vary, as other color intensity, here represented by gray level)

Transparency

- What can we do with the “alpha mask”?
  - blending / compositing!



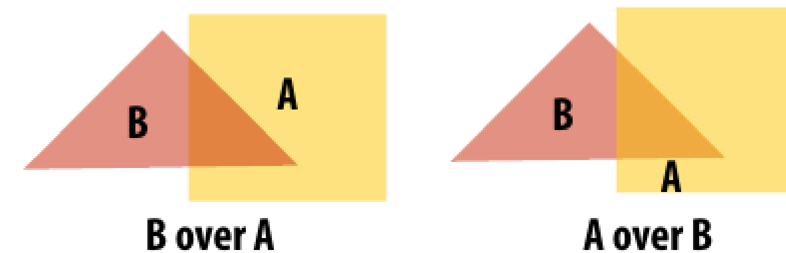
+

=

→ for coverage we had a function `coverage(x,y)`,
for occlusion we can talk of a function `over(A,B)`...

Transparency

- Transparency is actually not "commutative"
  - Two partially opaque objects will give two different intersection colors when overlaid in different order!



Important visual cue about depth for humans!

- In a simple world, shape B with transparency α_B over shape A with α_A

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

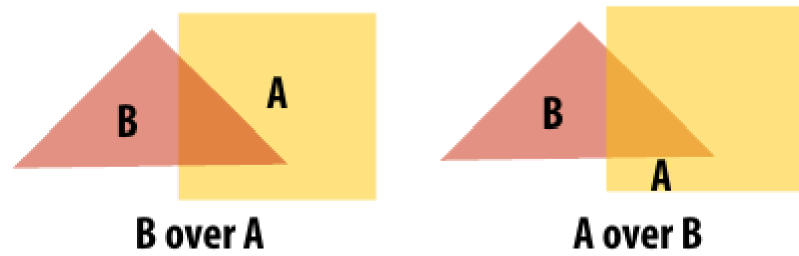
- Or, using pre-multiplied alpha:

$$A' = \begin{bmatrix} \alpha_A A_r & \alpha_A A_g & \alpha_A A_b & \alpha_A \end{bmatrix}^T$$
$$B' = \begin{bmatrix} \alpha_B B_r & \alpha_B B_g & \alpha_B B_b & \alpha_B \end{bmatrix}^T$$
$$C' = B + (1 - \alpha_B)A$$

- The resulting transparency of the two objects:

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

Transparency



Important visual cue about depth for humans!

- Transparency is actually not "commutative"
  - Two partially opaque objects will give two different intersection colors when overlaid in different order!
- In a simple world, shape B with transparency α_B over shape A with α_A

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

- Or, using pre-multiplied alpha:

(one less operation!)

$$A' = \begin{bmatrix} \alpha_A A_r & \alpha_A A_g & \alpha_A A_b & \alpha_A \end{bmatrix}^T$$
$$B' = \begin{bmatrix} \alpha_B B_r & \alpha_B B_g & \alpha_B B_b & \alpha_B \end{bmatrix}^T$$
$$C' = B + (1 - \alpha_B)A$$

- The resulting transparency of the two objects:

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

You can decide how OpenGL does it:

```
glEnable(GL_BLEND);
glBlendFunc(GL_SRC_ALPHA,
    GL_ONE_MINUS_SRC_ALPHA);
```

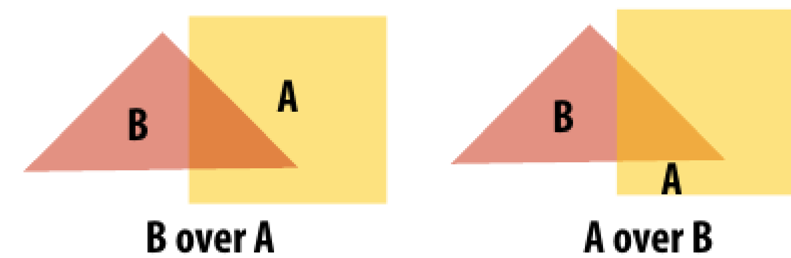
Transparency

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

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

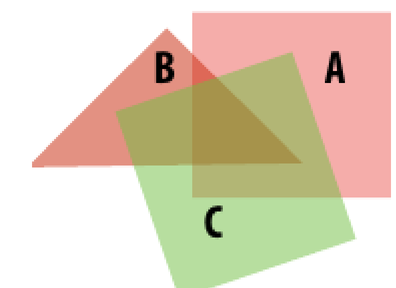
$$C = \alpha_B B + (1 - \alpha_B)\alpha_A A \qquad C' = B + (1 - \alpha_B)A$$

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

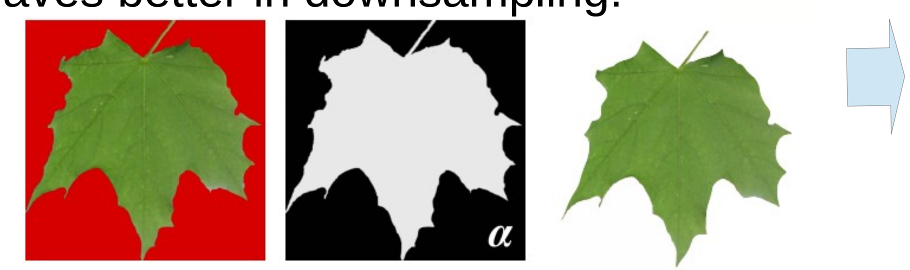


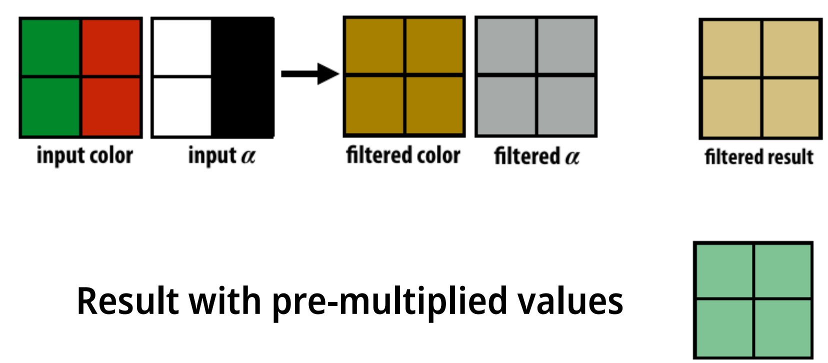
- Not so fast, does our mechanism really work for non pre-multiplied case? What if we need to continue repeating the operation?
- Ooops, it produces pre-multiplied color values… Need to correct it:

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



- The pre-multiplied model has many advantages!
- Also behaves better in downsampling:





Transparency

- Seems like a simple issue but watch out: we have just seen an example of 2-D shapes overlapping
  - No light model, no 3-D information
  - In practice: challenging computation even for modern hardware

Thank you!

- Questions?