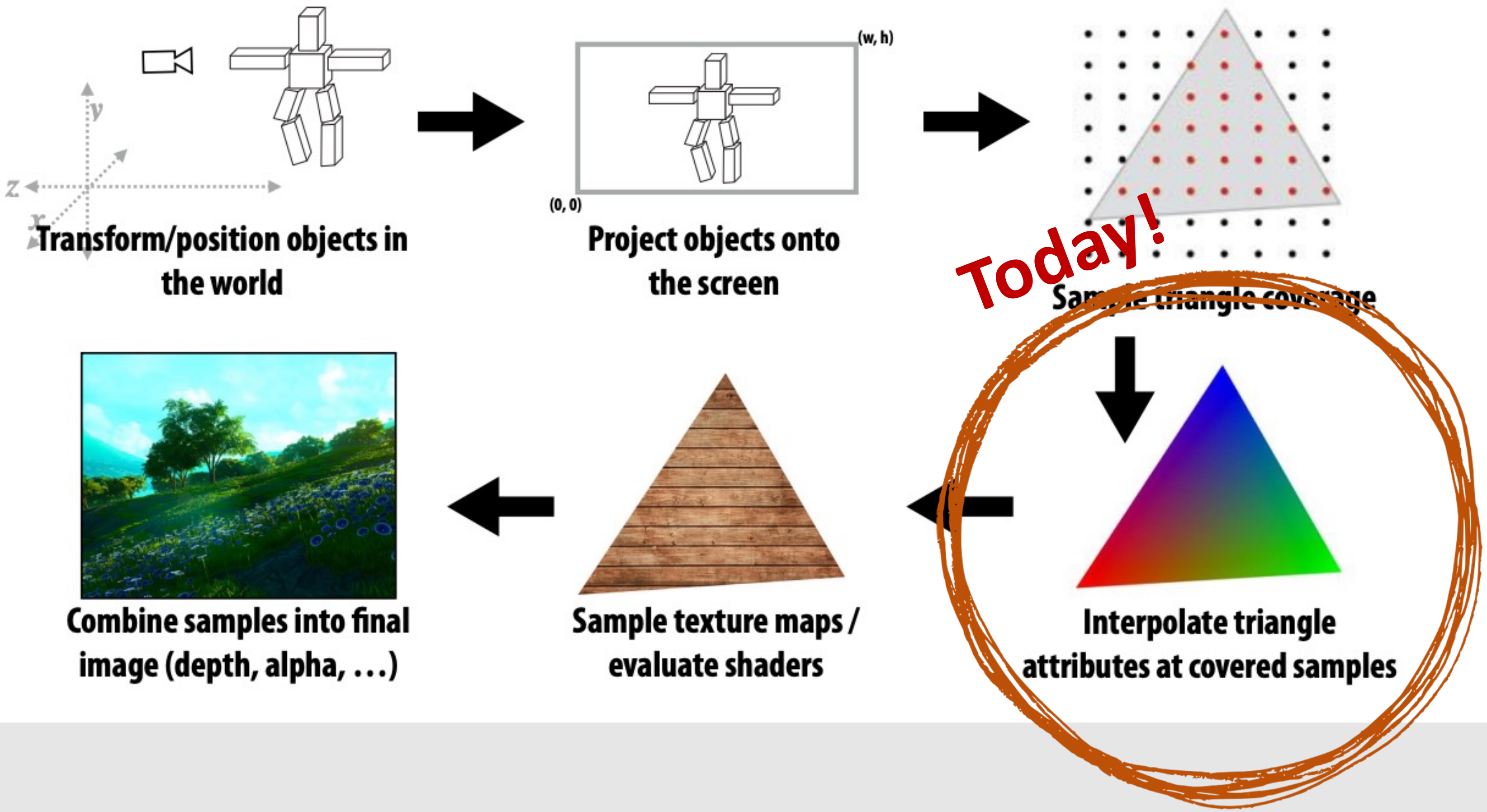
Transparency & Texturing

- Barycentric Coordinates
- Texturing Surfaces
- Depth Testing
- Alpha Blending
- The Graphics Pipeline Revisited

The “Simpler” Graphics Pipeline

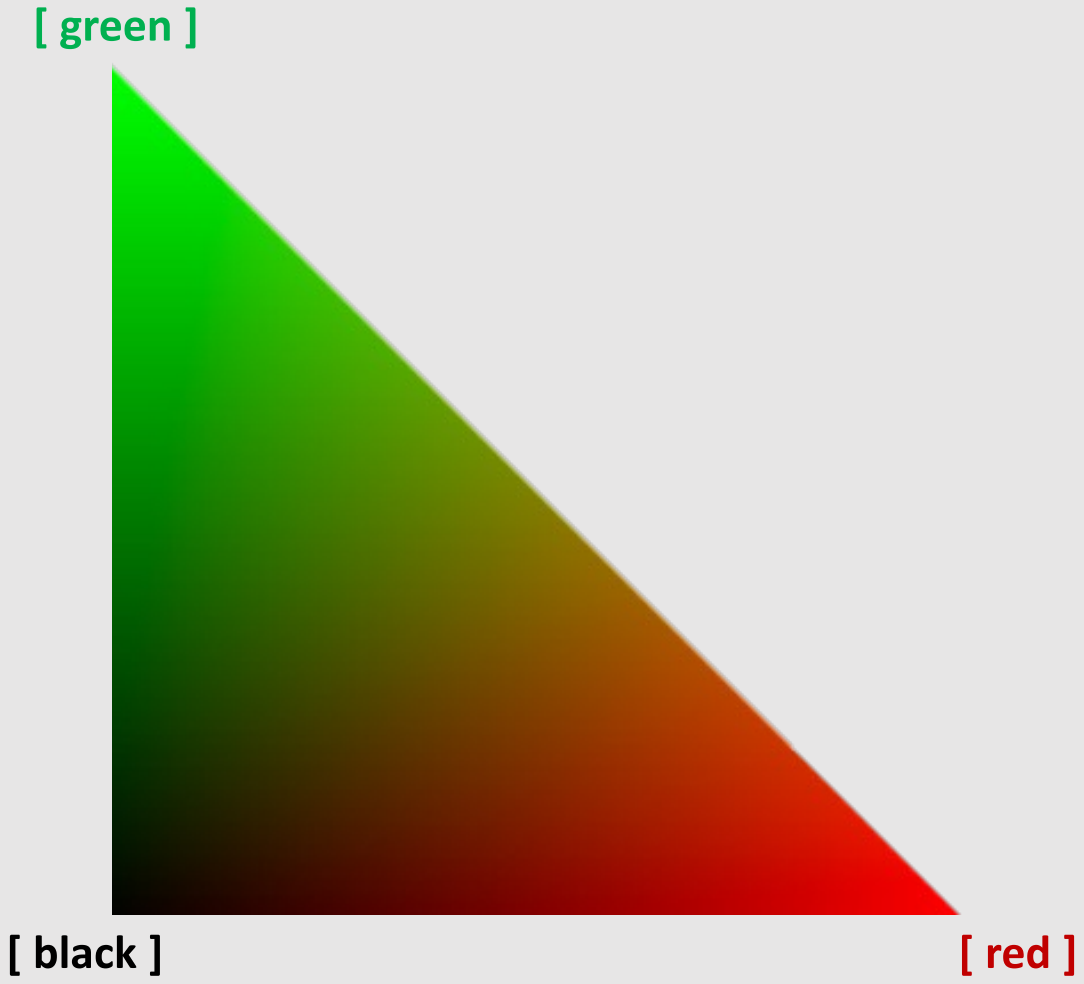


Interpolating Values for Triangles

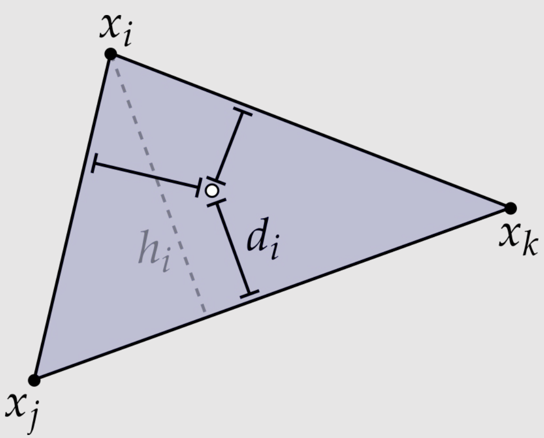
- **Goal:** interpolate triangle vertices for any point within triangle

- Coordinates (ϕ_i, ϕ_j, ϕ_k) should represent weighted average
  - $\phi_i + \phi_j + \phi_k = 1$
  - Similarly, $1 - \phi_i - \phi_j = \phi_k$
  - Gives a 2D parameterization of triangle point (ϕ_i, ϕ_j)
    - Known as **barycentric coordinates**

- If each point has some attribute $(\alpha_i, \alpha_j, \alpha_k)$, can linearly interpolate $\alpha_i\phi_i + \alpha_j\phi_j + \alpha_k\phi_k$
  - **Example:** [black]ϕ_i + [green]ϕ_j + [red] ϕ_k



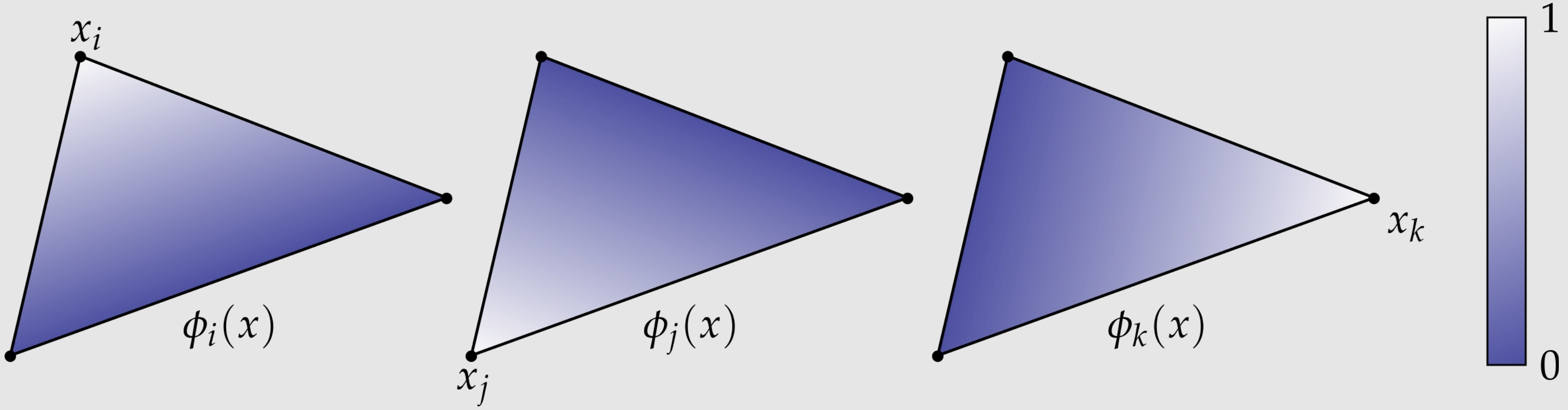
Barycentric Coordinates



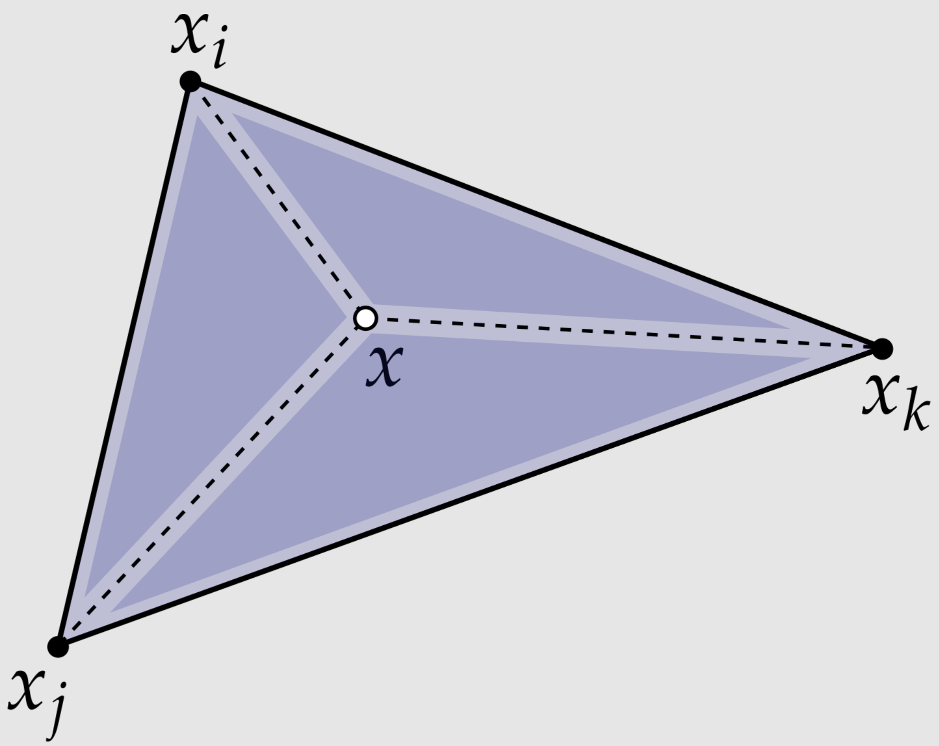
- Inversely proportional to the distance between the target point and a point within the triangle
- Can be computed as:

$$\phi_i(x) = d_i(x)/h_i$$

- How would you compute h_i? $d_i(x)$?



Barycentric Coordinates [Another Way]



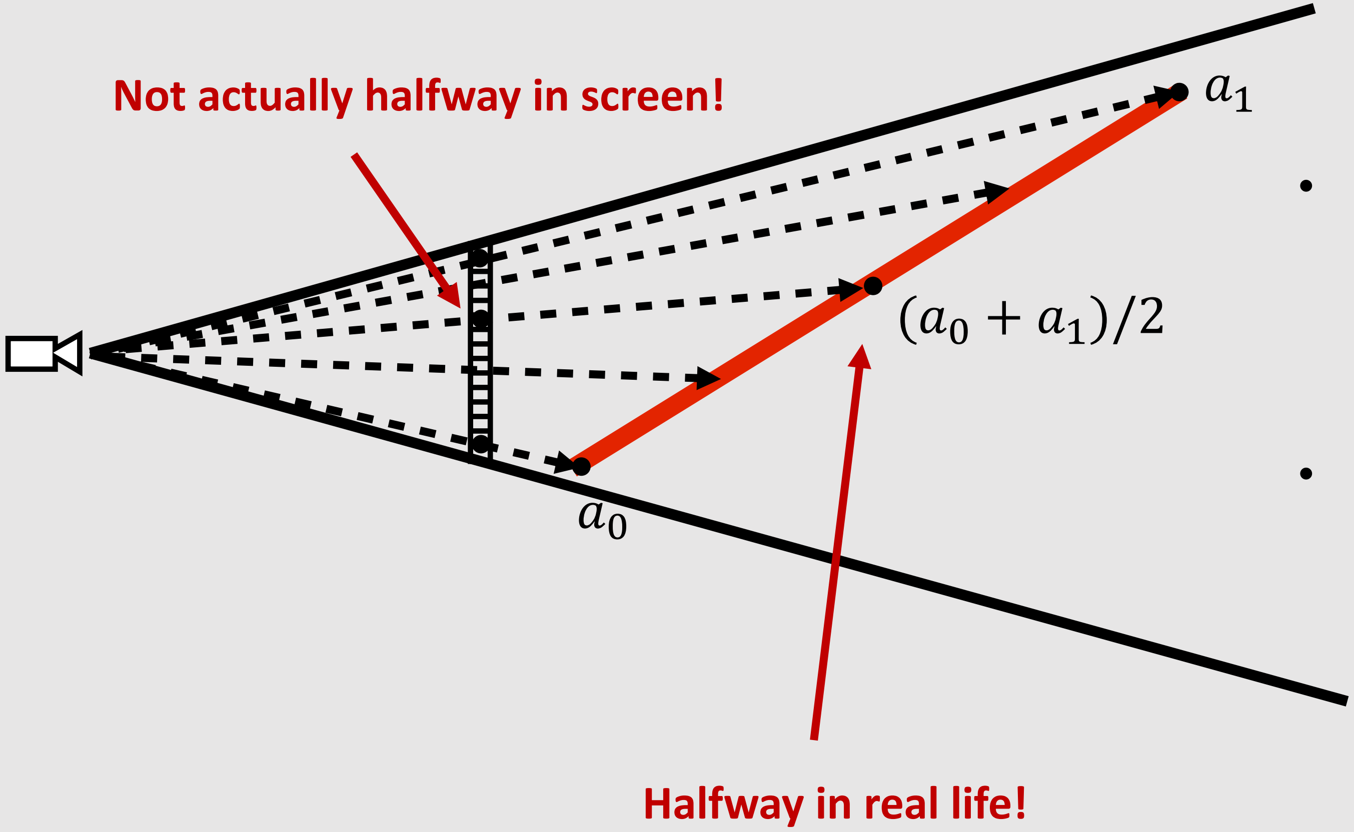
- Directly proportional to the area created by the triangle composed of the other two target points and a point within the triangle

- Can be computed as:

$$\phi_i(x) = \frac{\text{area}(x, x_j, x_k)}{\text{area}(x_i, x_j, x_k)}$$

** Interesting read of barycentric coordinates for n-gons: https://www.inf.usi.ch/hormann/barycentric/

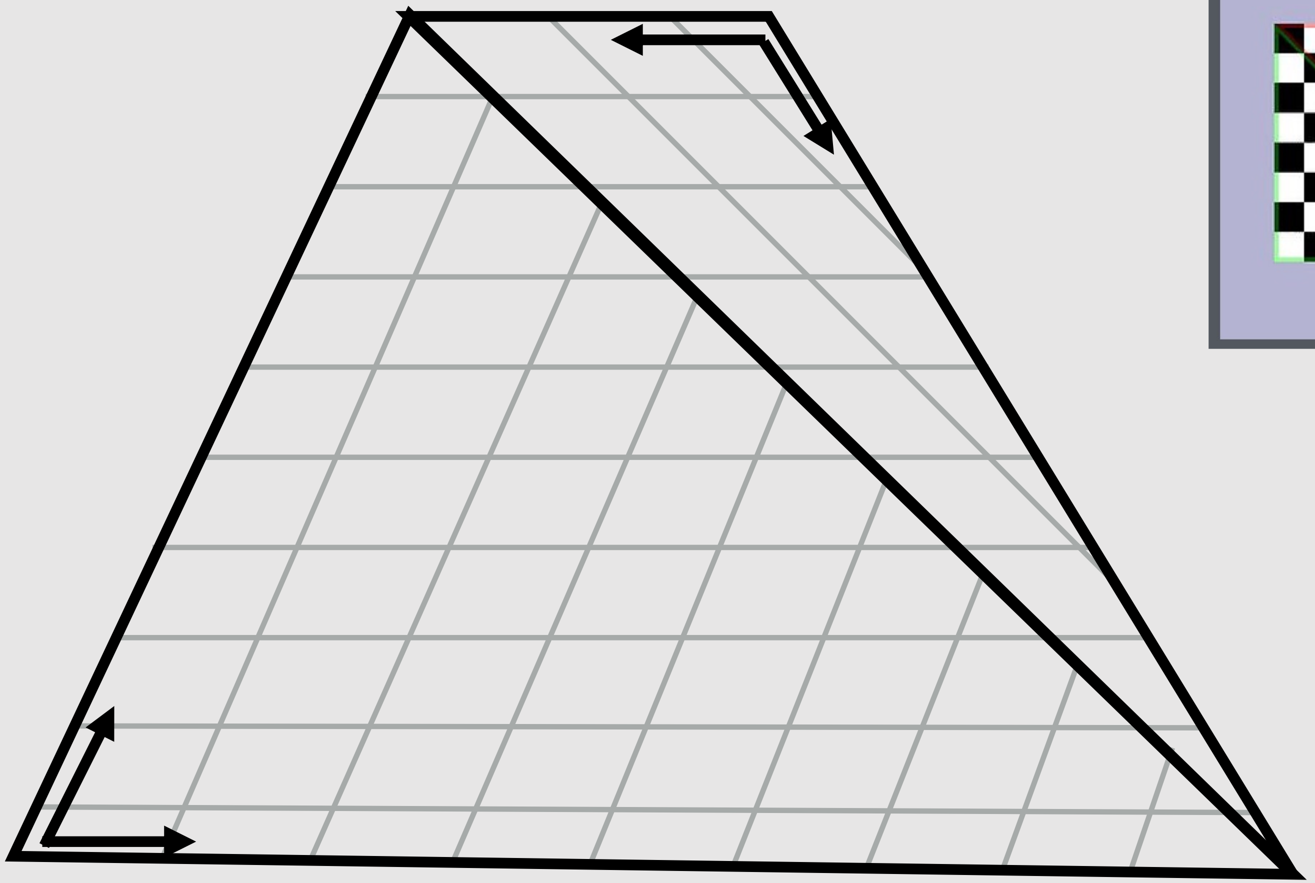
Perspective-Incorrect Interpolation

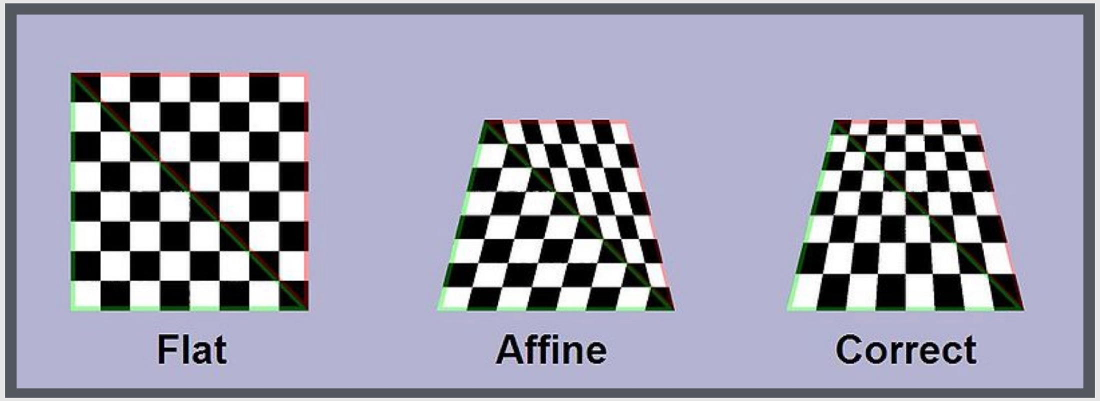


- Due to perspective projection (homogeneous divide), barycentric interpolation of values on a triangle with different depths is not an affine function of screen XY coordinates

- Want to interpolate attribute values linearly in **3D object space**, not image space.

Perspective-Incorrect Interpolation





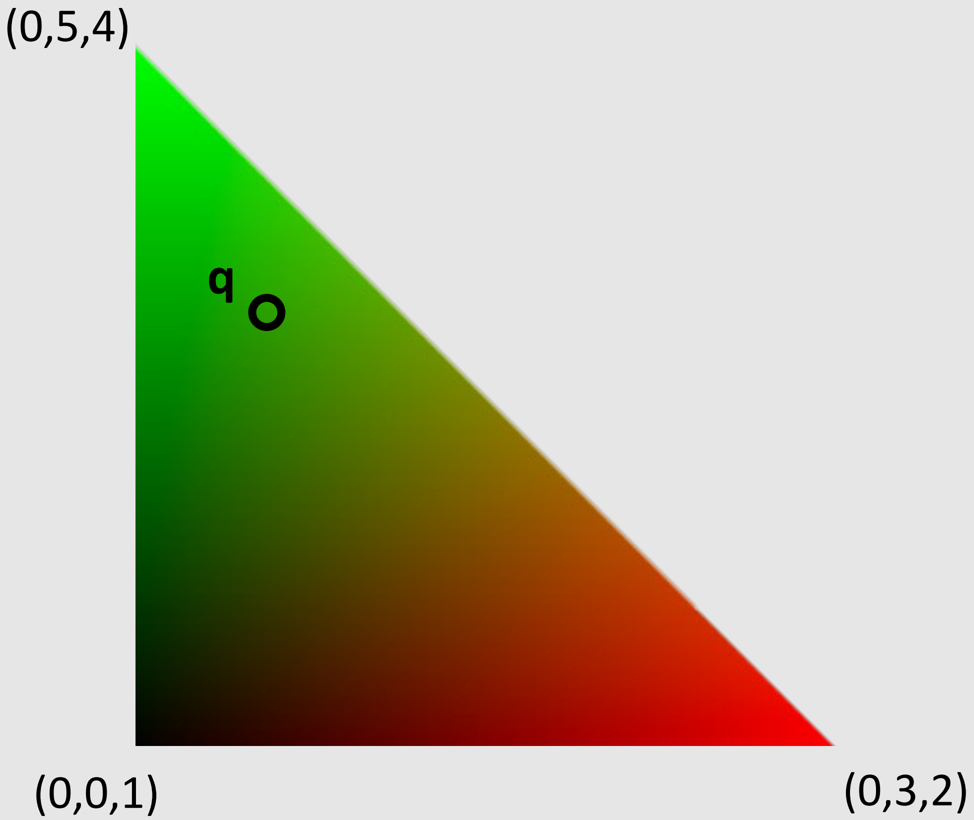
If we compute barycentric coordinates using 2D (projected) coordinates, leads to (derivative) discontinuity in interpolation where quad was split

Perspective-Correct Interpolation

- **Goal:** interpolate some attribute v at vertices
  - Compute depth z at each vertex
  - Evaluate $Z := 1/z$ and $P := v/z$ at each vertex
  - Interpolate Z and P using standard (2D) barycentric coordinates
  - At each fragment, divide interpolated P by interpolated Z to get final value

Perspective-Correct Interpolation

$$\phi_{(0,0,1)} = 0.2 \qquad P_{(0,0,1)} = (0,0,0)/1 \qquad Z_{(0,0,1)} = 1$$
$$\phi_{(0,3,2)} = 0.1 \qquad P_{(0,3,2)} = (1,0,0)/2 \qquad Z_{(0,3,2)} = 1/2$$
$$\phi_{(0,5,4)} = 0.7 \qquad P_{(0,5,4)} = (0,1,0)/4 \qquad Z_{(0,5,4)} = 1/4$$



$$P_{interp} = 0.2 * [(0,0,0)/1] + 0.1 * [(1,0,0)/2] * 0.7 * [(0,1,0)/4]$$
$$P_{interp} = (0.05, 0.175, 0)$$

$$Z_{interp} = 0.2 * [1/1] + 0.1 * [1/2] * 0.7 * [1/4]$$
$$Z_{interp} = 0.425$$

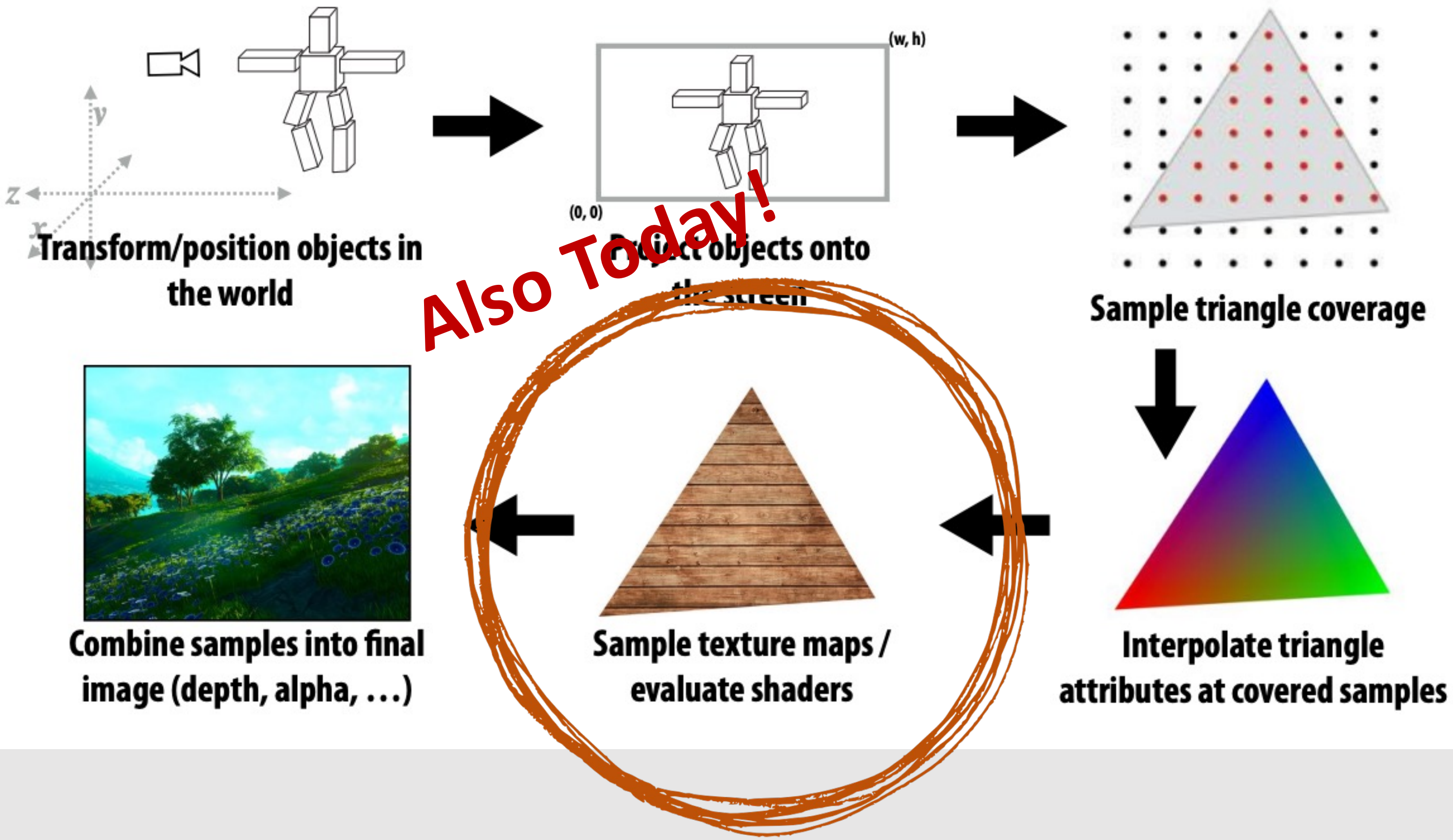
$$q = (0.05, 0.175, 0)/0.425$$
$$q = (0.12, 0.412, 0)$$

What if z is equal to 0?

Remember the near clipping plane!

- ~~Barycentric Coordinates~~
- Texturing Surfaces
- Depth Testing
- Alpha Blending
- The Graphics Pipeline Revisited

The “Simpler” Graphics Pipeline



Textures in Graphics

- Textures are buffers of data (images) that are read into the graphics pipeline and are used for:
  - Coloring mapping
  - Normal mapping
  - Displacement mapping
  - Roughness mapping
  - Occlusion mapping
  - Reflection mapping
    - Textures can also be written into
      - Think a scratch pad for data

- Useful for maximizing quality while minimizing the number of polygons
  - Rough surfaces can be approximated by smooth surfaces with rough textures

- A single pixel of a texture is known as a **texel**

The Last of Us Part II (2020) Naughty Dog

Textures in Graphics

changes the visual appearance (color of fur)

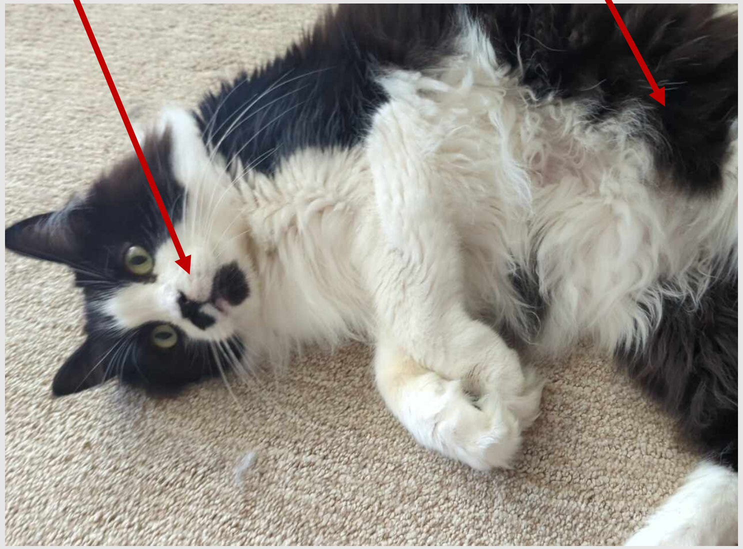
preserves geometric fluff

[fluffy geometry]

+

[monochrome texture]

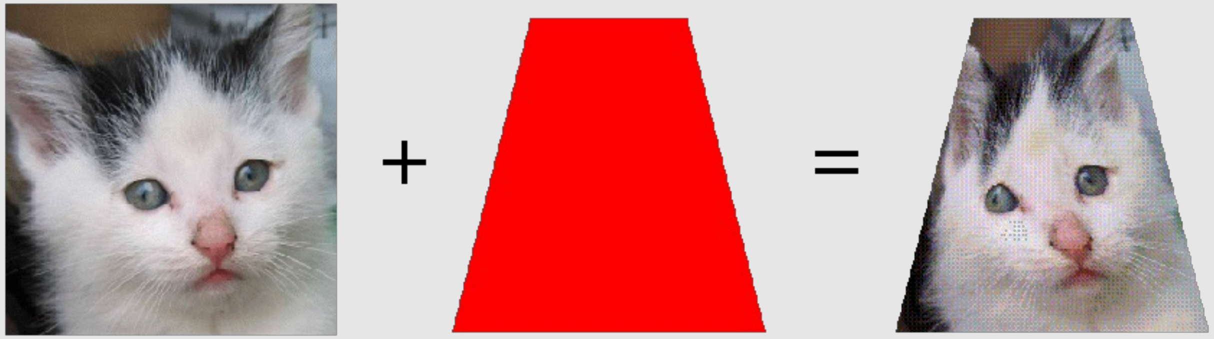
=



[textured geometry]

Texture Coordinates

- **Goal:** map surface geometry coordinates to image coordinates
- Barycentric coordinates let us represent 3D geometry in 2D by their surface coordinates
    - Known as **surface parameterization**
- Not always a 1-to-1 map!
    - A surface only half the number of pixels of a texture may only use up half the texels**

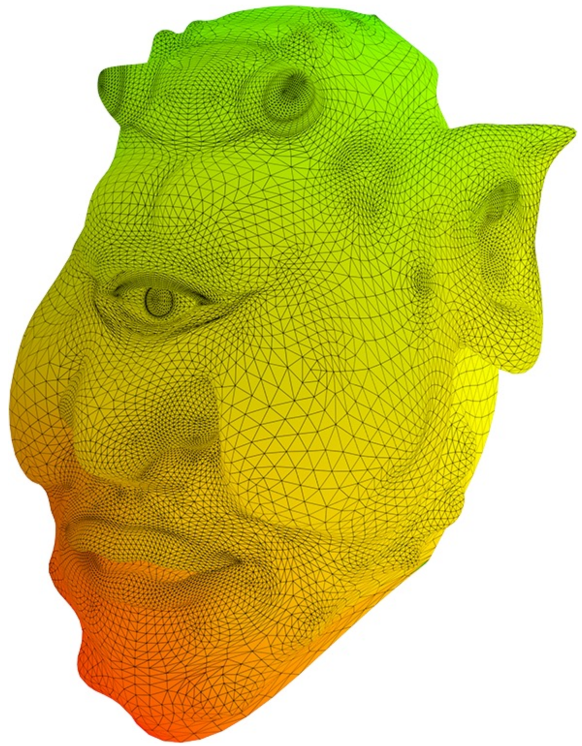


[texture] **[geometry]** **[render]**

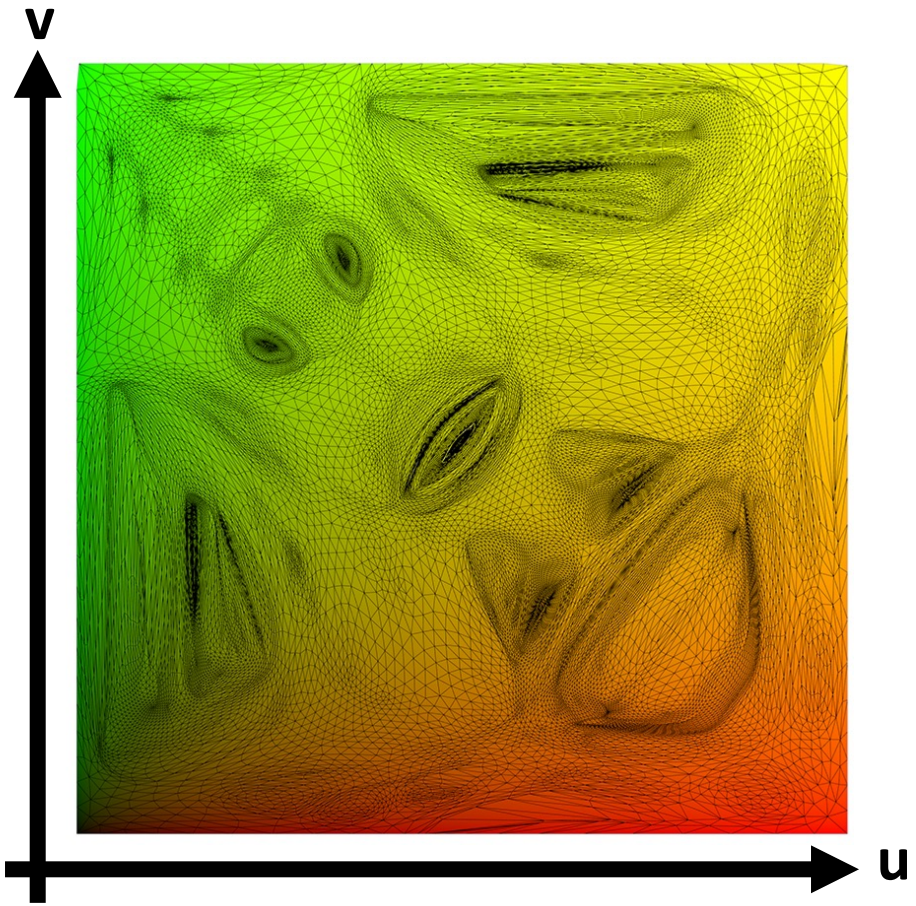
**We will learn ways that surfaces may use more texels than there are pixels on the surface

Texture Example

[texture coordinates on surface]



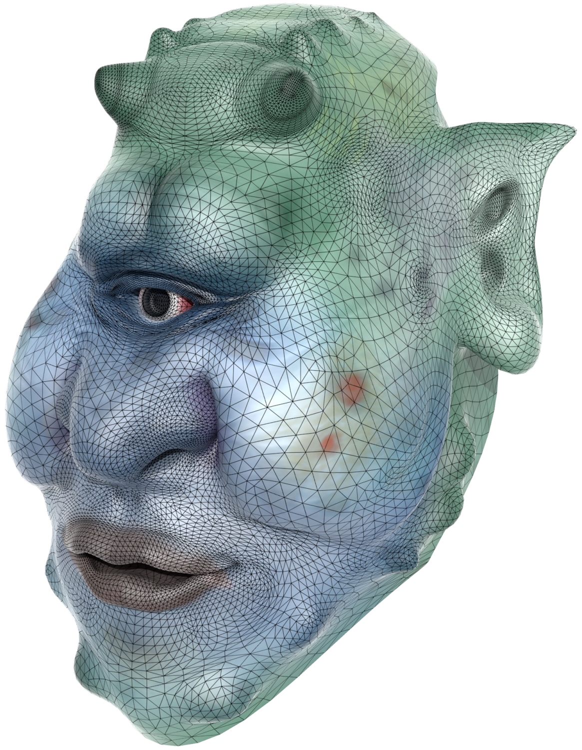
[texture coordinates on texture]



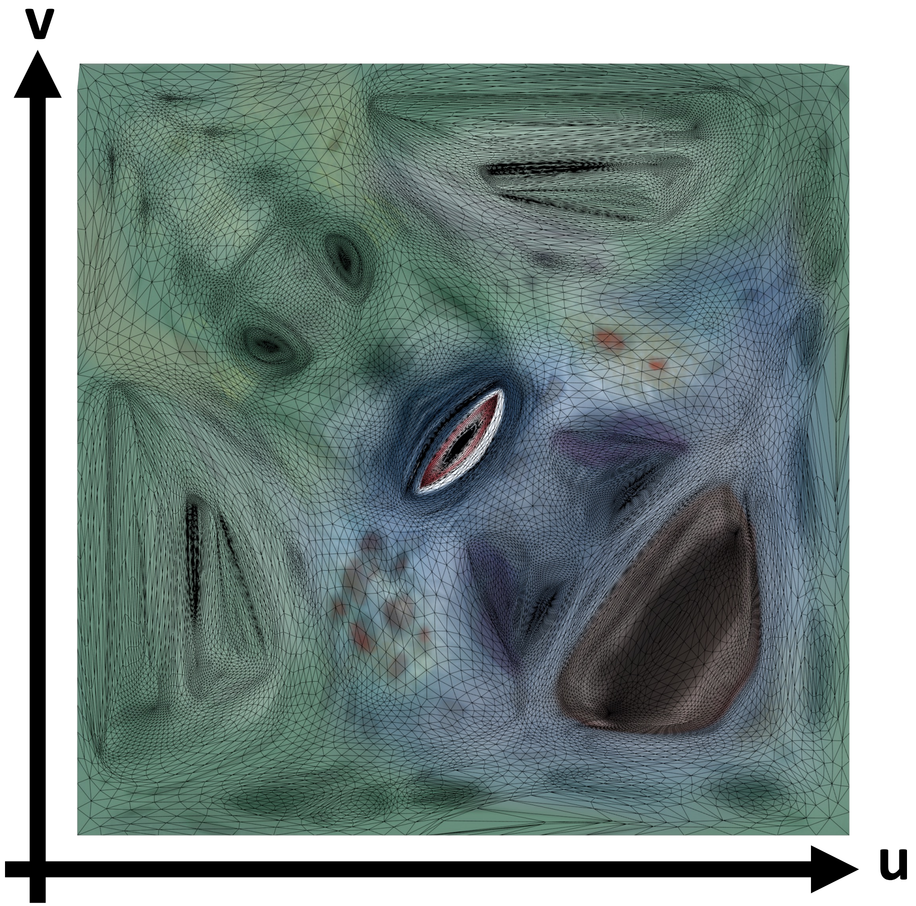
Each vertex has a coordinate (u,v) in texture space

Texture Example

[rendered results]

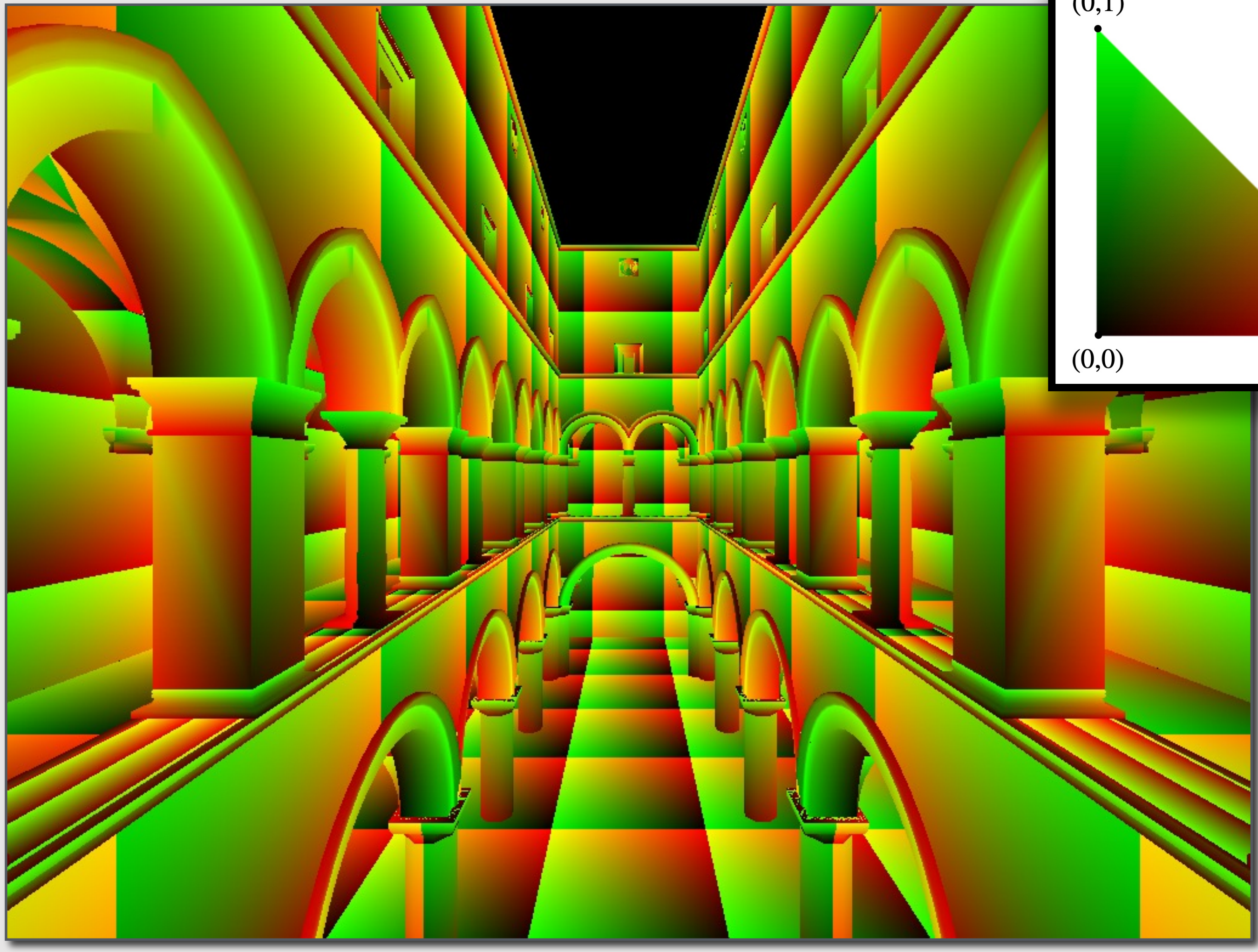


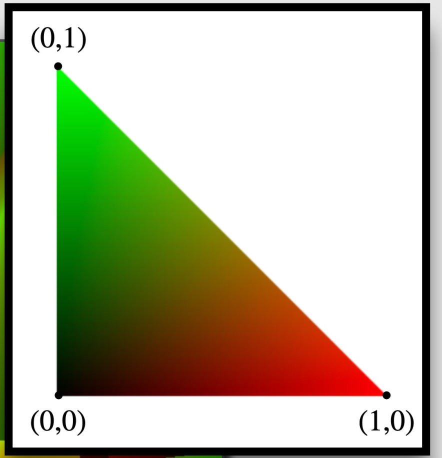
[texture data]



Each triangle “copies” a piece of the image back to the surface

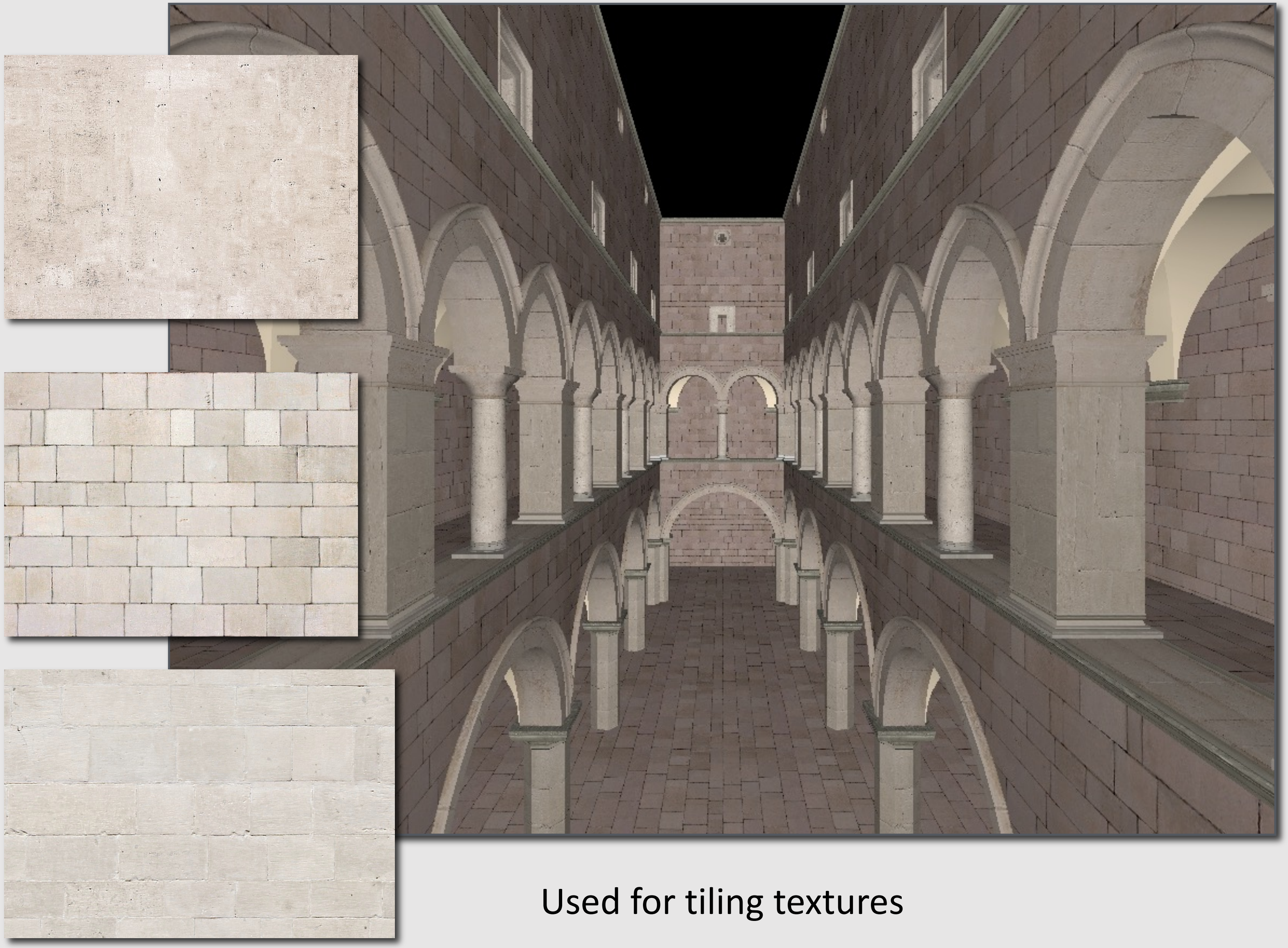
Periodic Texturing





Why do you think texture coordinates might repeat over the surface?

Periodic Texturing



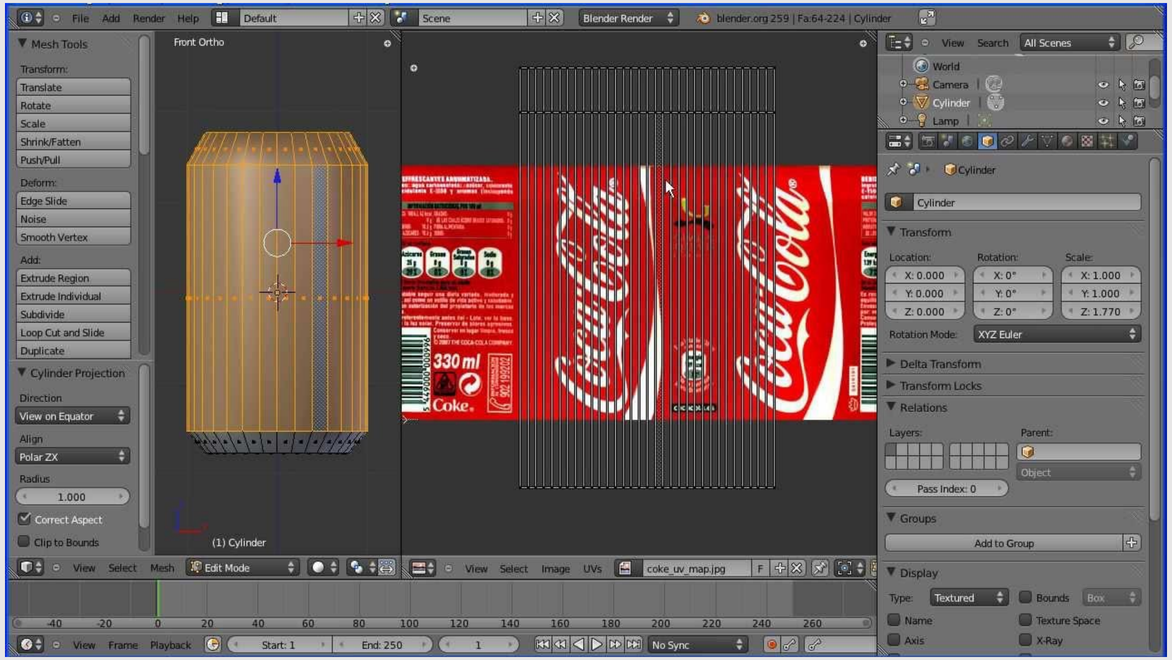
Used for tiling textures

How Texturing Is Done

- An artist goes into a program and drags/paints/stretches/warps textures onto surfaces
  - The resulting distortion of the texture on the surface is saved as the **surface parameterization**

- **Computing the texture mapping function is never done by hand!**
  - Always use an interactive program to do it

- Also known as **uv mapping**
  - u and v are the two barycentric coordinates that we want to map onto texture space



Texturing (2017) Blender

Texture mapping maps a non-integer coordinate to another non-integer coordinate.
But textures can only be accessed via integer…

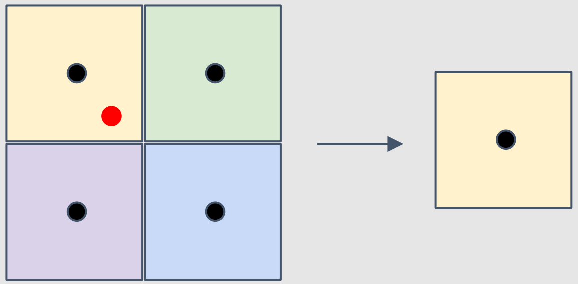
How do we know what texel(s) to sample?

Nearest Neighbor Sampling

- **Idea:** Grab texel nearest to requested location in texture
- **Requires:**
  - 1 memory lookup
  - 0 linear interpolations

$$x' \leftarrow round(x - 0.5), \quad y' \leftarrow round(y - 0.5)$$

$$t \leftarrow tex.lookup(x', y')$$

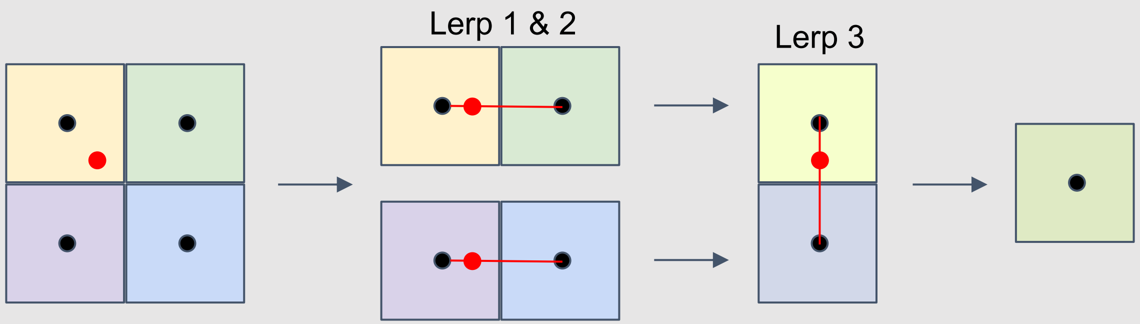


x' and y' are half-integer coordinates
Helps account for 0.5 offset from texture coordinate centers

Bilinear Interpolation Sampling

- **Idea:** Grab nearest 4 texels and blend them together based on their inverse distance from the requested location
  - Blend two sets of pixels along one axis, then blend the remaining pixels

- **Requires:**
  - 4 memory lookup
  - 3 linear interpolations



$$x' \leftarrow floor(x - 0.5), \qquad y' \leftarrow floor(y - 0.5)$$

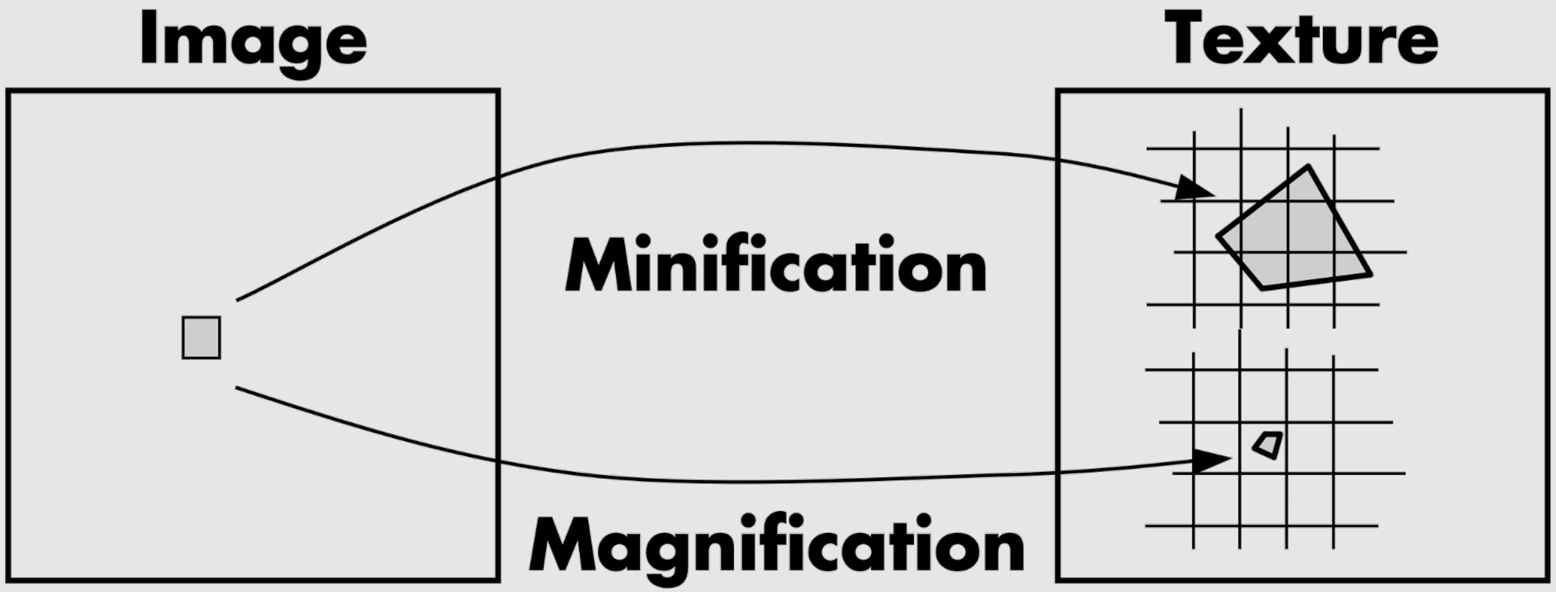
$$\Delta x \leftarrow x - x'$$
$$\Delta y \leftarrow y - y'$$

$$t_{(x,y)} \leftarrow tex.lookup(x', y')$$
$$t_{(x+1,y)} \leftarrow tex.lookup(x' + 1, y')$$
$$t_{(x,y+1)} \leftarrow tex.lookup(x', y' + 1)$$
$$t_{(x+1,y+1)} \leftarrow tex.lookup(x', +1\ y' + 1)$$

$$t_x \leftarrow (1 - \Delta x) * t_{(x,y)} + \Delta x * t_{(x+1,y)}$$
$$t_y \leftarrow (1 - \Delta x) * t_{(x,y+1)} + \Delta x * t_{(x+1,y+1)}$$

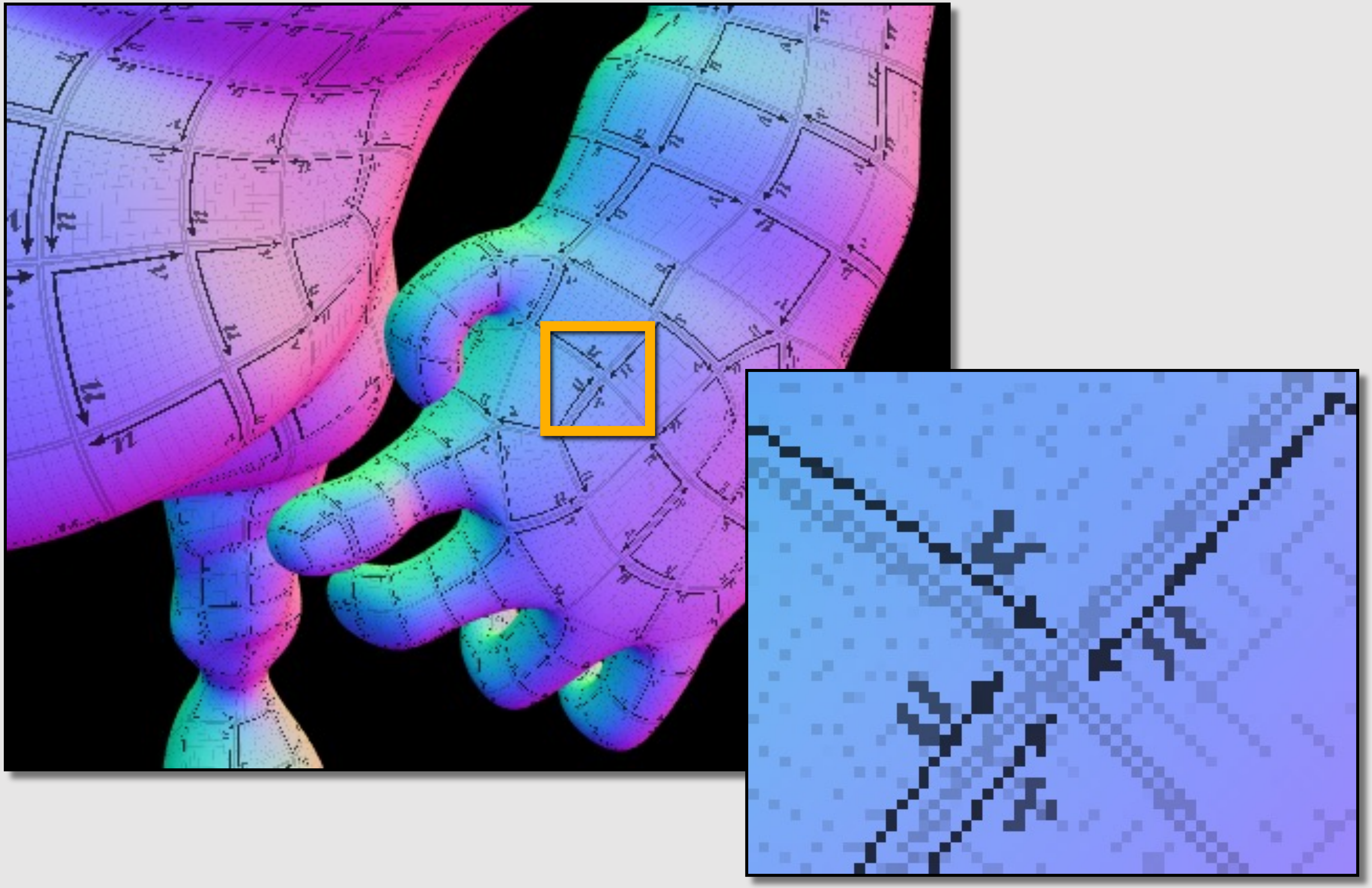
$$t \leftarrow (1 - \Delta y) * t_x + \Delta y * t_y$$

Minification vs. Magnification

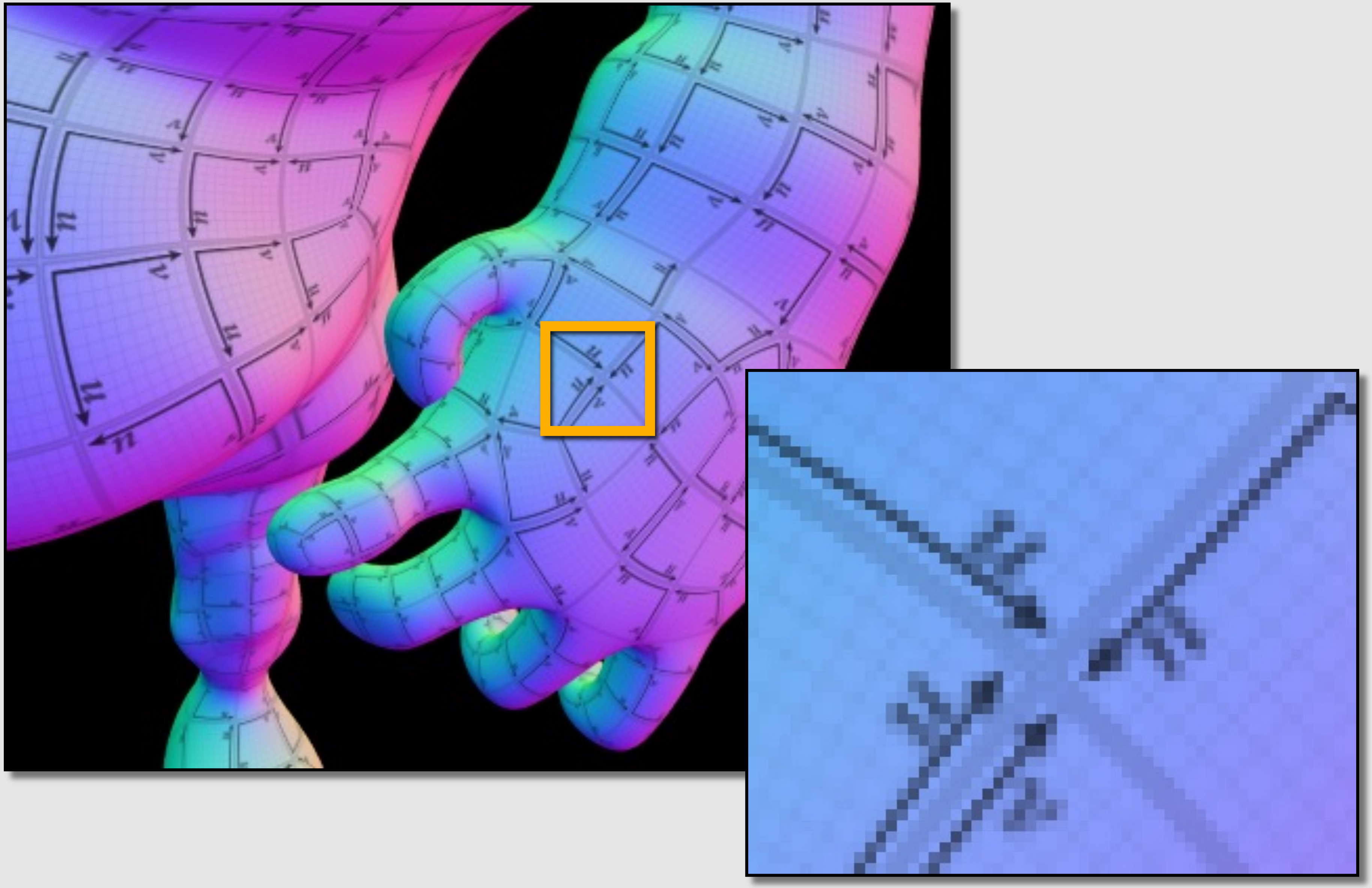


- **Magnification [Nearest Neighbor, Bilinear]:**
  - *Example:* camera is very close to scene object
  - Single screen pixel maps to tiny region of texture
  - Can just interpolate value at screen pixel center
- **Minification [???]**
  - *Example:* scene object is very far away
  - Single screen pixel maps to large region of texture
  - Need to compute average texture value over pixel to avoid aliasing

Aliasing Due To Minification



Pre-Filtering Texture

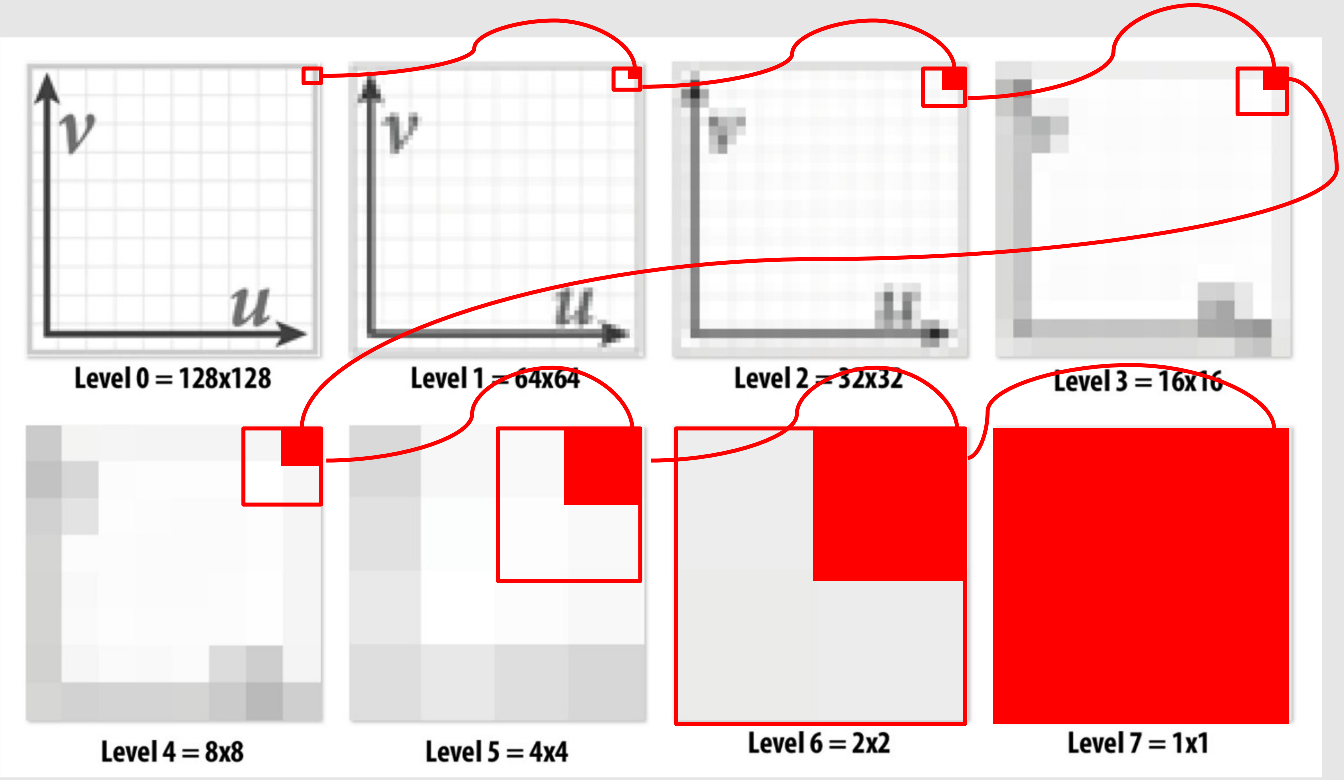


Texture Pre-Filtering

- **Texture aliasing** occurs because a single pixel on the screen covers many pixels of the texture
- Ideally, want to average a bunch of texels in a very large region (expensive!)
  - Instead, we can pre-compute the averages (once) and just look up these averages (many times) at run-time
- Q: Which averages to pre-compute
  - A: a lot of them!

Mip-Map [L. Williams '83]

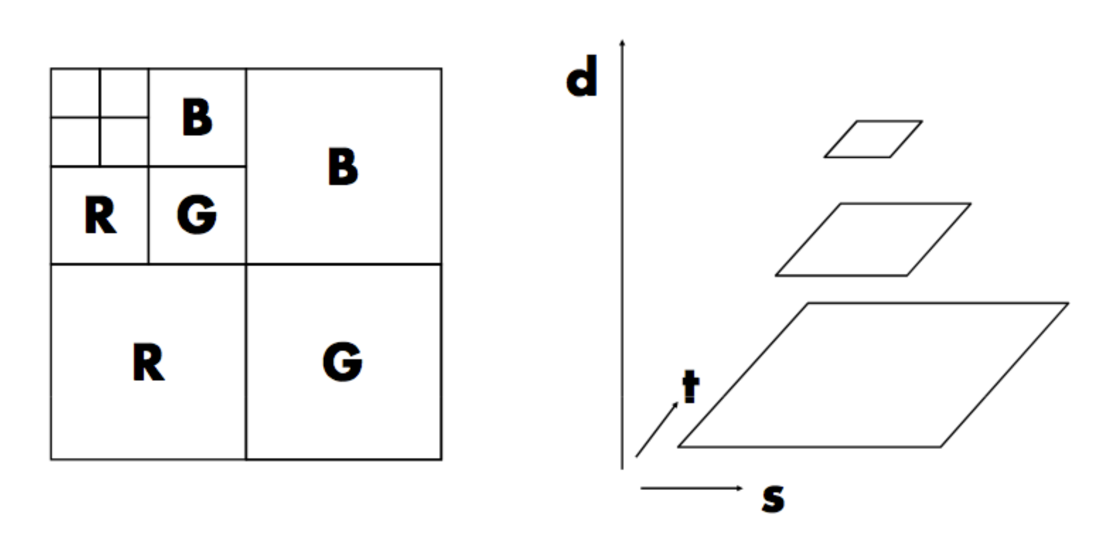
- **Rough idea:** precompute a prefiltered image at every possible scale
  - The image at depth d is the result of applying a 2x2 avg filter on the image at depth d-1
    - The image at depth 0 is the base image
- Mip-Map generates $log_2[\min(wth, hgt)] + 1$ levels
  - Each level the width and height gets halved
- Memory overhead: (1+1/3)x original texture
  - $1 + \frac{1}{4} + \frac{1}{16} + \cdots = \sum \frac{1}{4}^j = \frac{1}{1-\frac{1}{4}} = \frac{4}{3}$

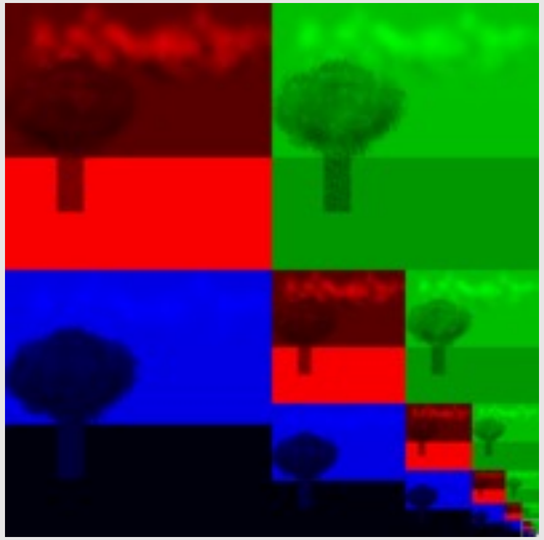


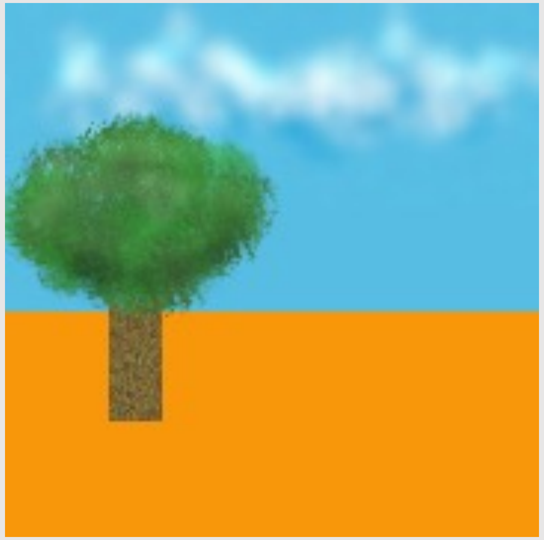
Mip-Map [L. Williams '83]

- Storing an RGB Mip-Map can be fit into an image twice the width and twice the height of the original image
  - See diagram for proof :)
  - Does not work as nicely for RGBA!

- **Issue:** bad spatial locality
  - Requesting a texel requires lookup in 3 very different regions of an image

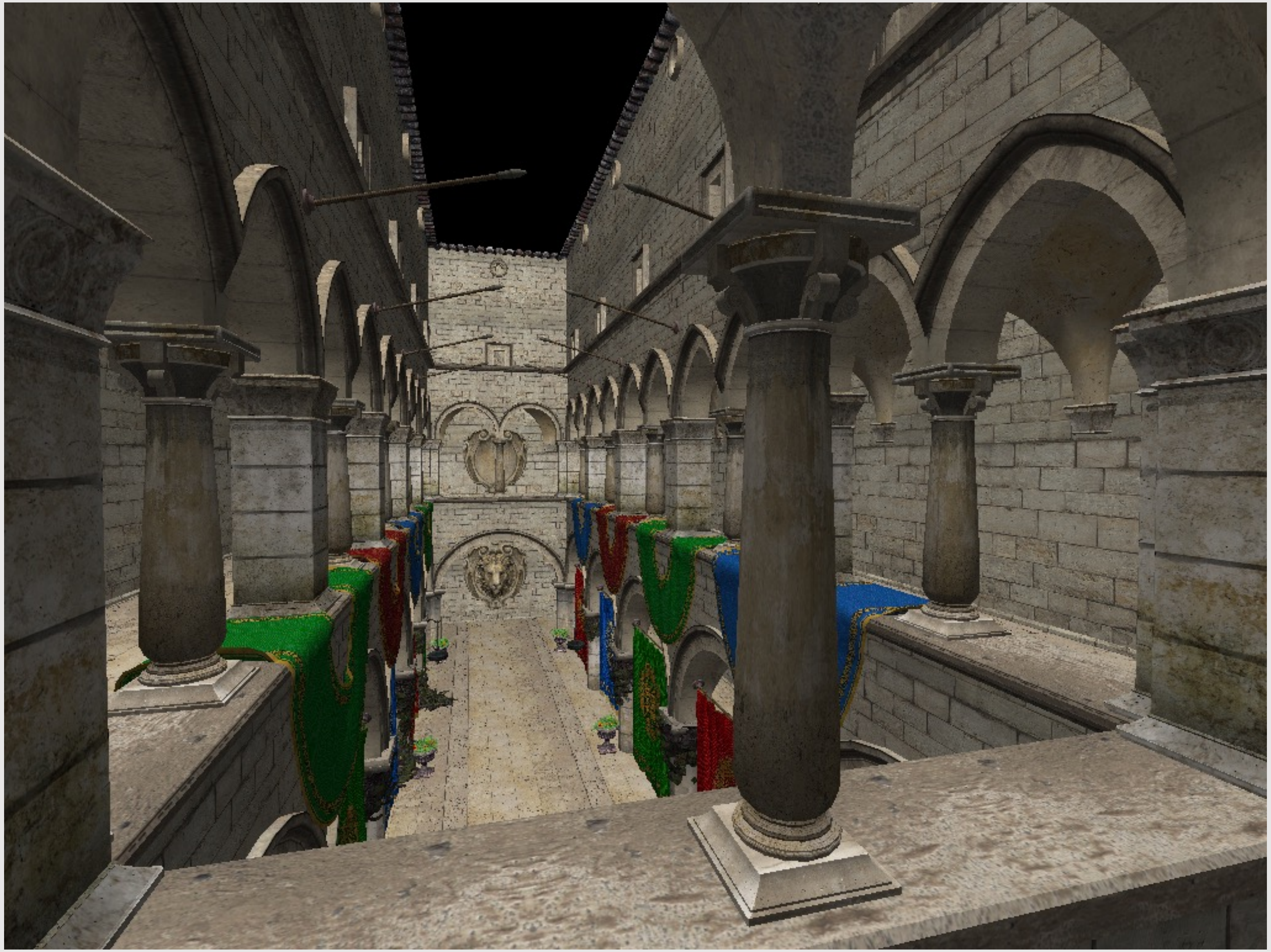




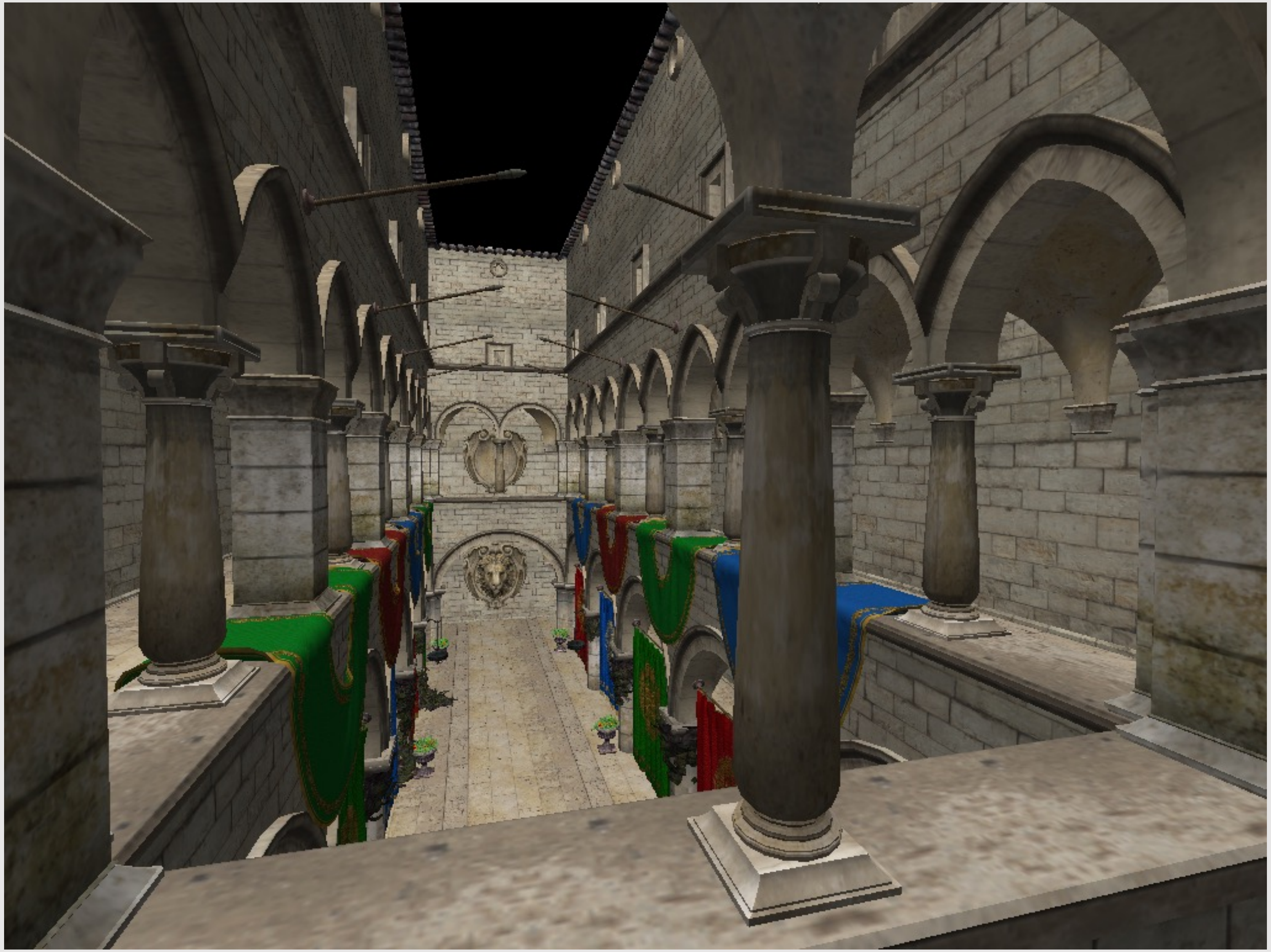


Which mip-map level do we use?

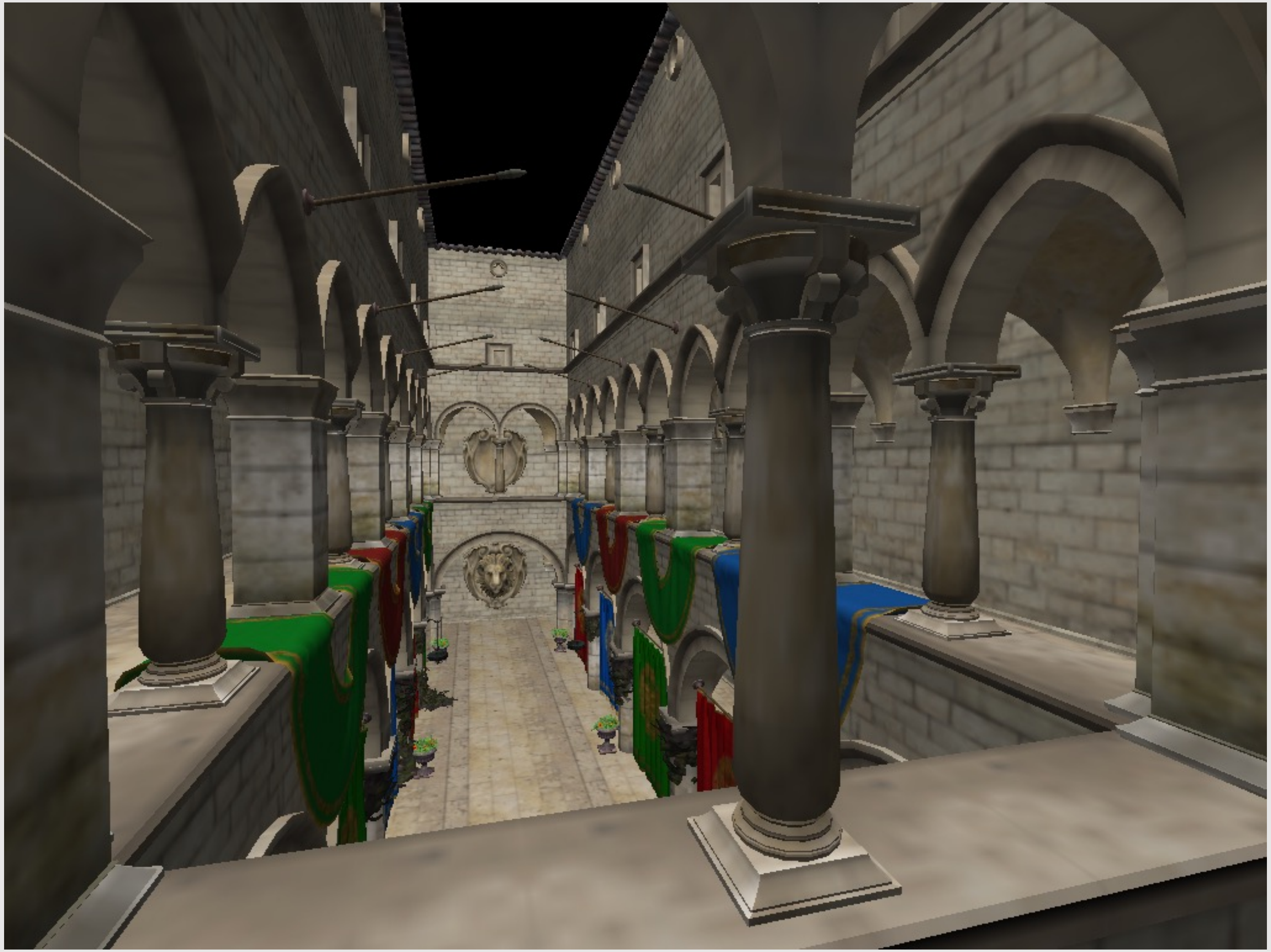
Sponza Bilinear Interpolation [Level 0]



Sponza Bilinear Interpolation [Level 2]

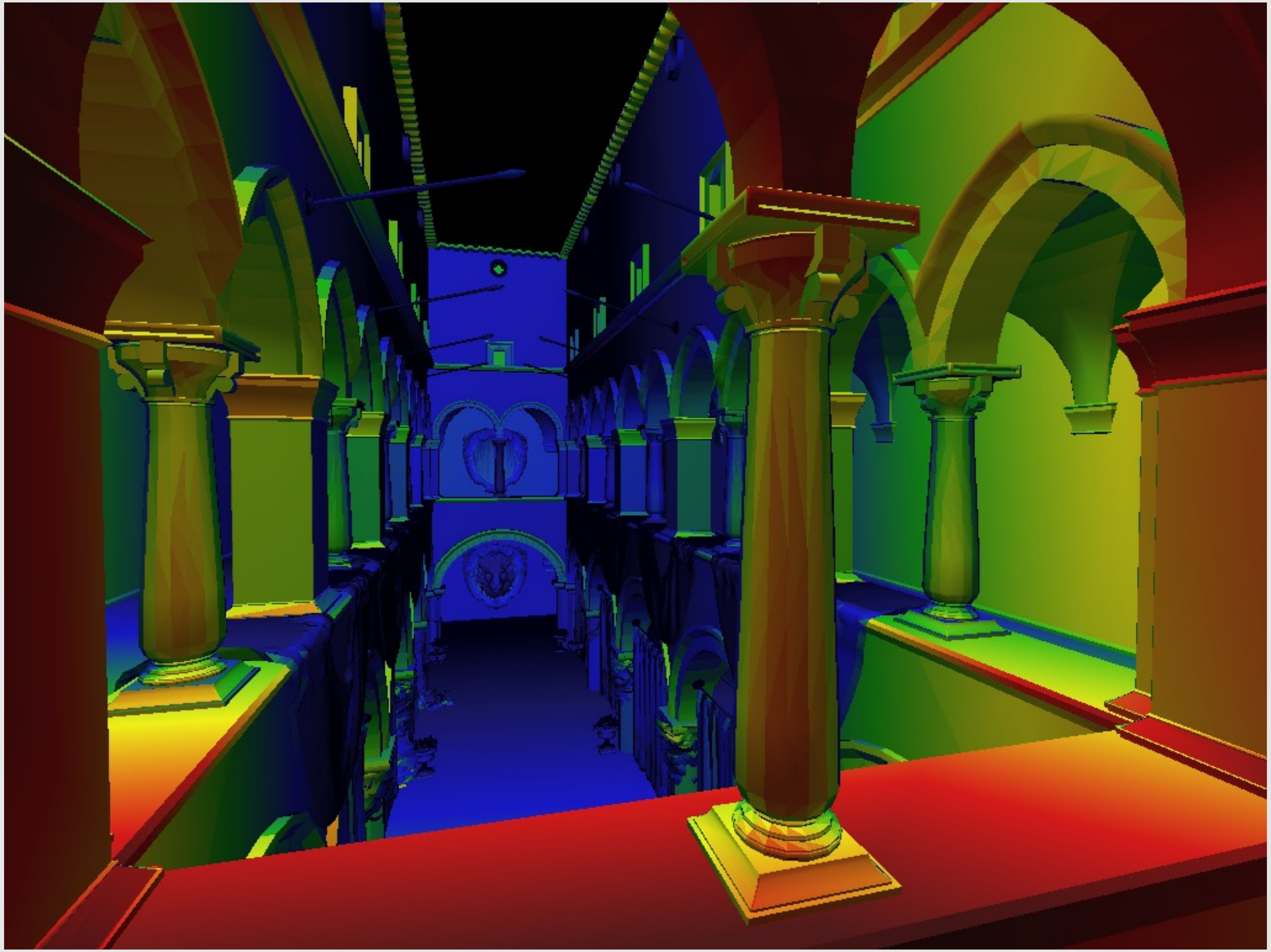


Sponza Bilinear Interpolation [Level 4]



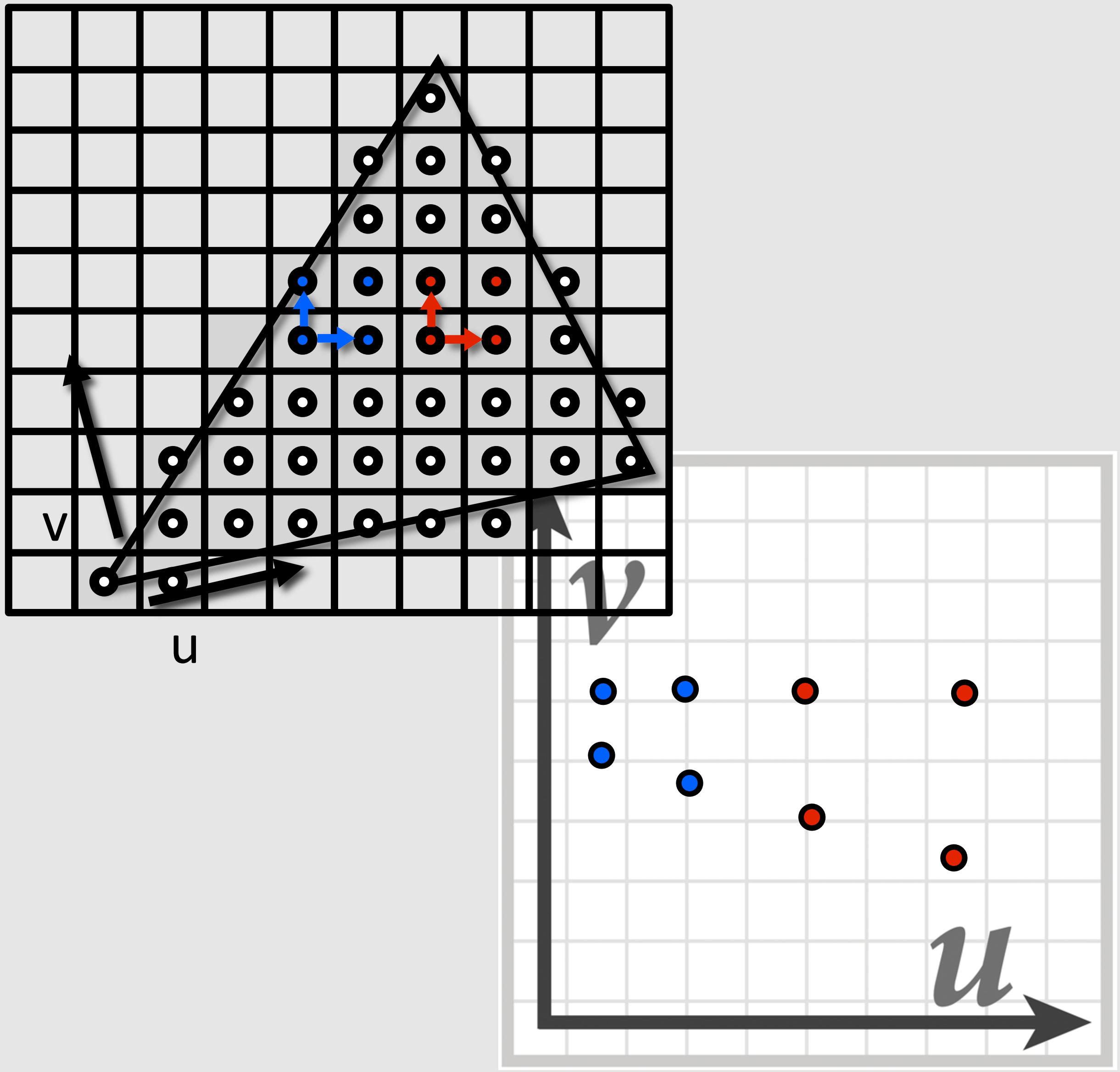
Sponza Bilinear Interpolation [Varying Level]

Sponza Visualization of Level



Computing MipMap Depth

- Correlation between distance of surface to camera and level of mip-map accessed
  - More specifically, **correlation between screen-space movement across the surface compared to texture movement** and level of mip-map access
- If moving over a pixel in screen space is a big jump in texture space, then we call it **minification**
  - Sample from a lower level of mip-map
- If moving over a pixel in screen space is a small jump in texture space, then we call it **magnification**
  - Sample from a higher level of mip-map



Computing MipMap Depth

More formally:

$$\frac{du}{dx} = u_{10} - u_{00} \qquad \frac{du}{dy} = u_{01} - u_{00}$$

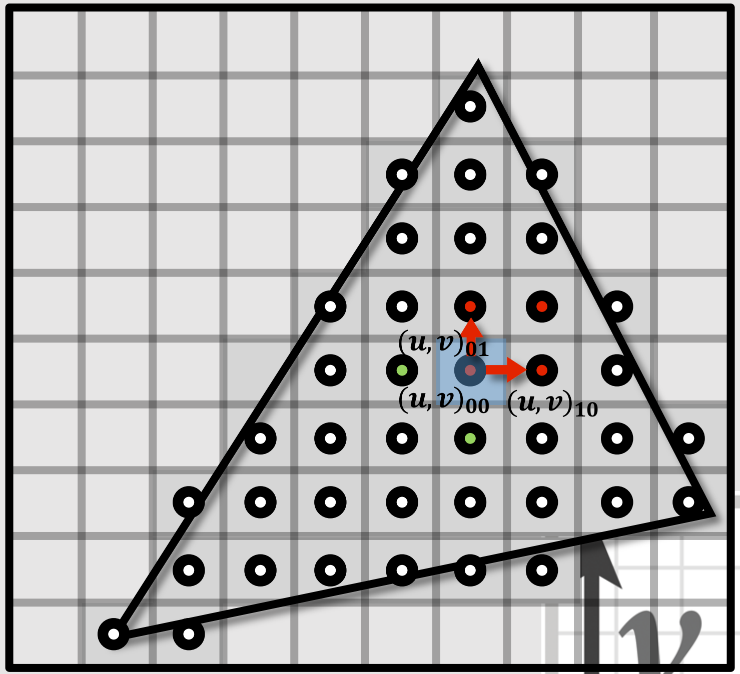
$$\frac{dv}{dx} = v_{10} - v_{00} \qquad \frac{dv}{dy} = v_{01} - v_{00}$$

Where dx and dy measure the change in screen space and du and dv measure the change in texture space

$$L_x^2 = \left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2 \qquad L_y^2 = \left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2$$

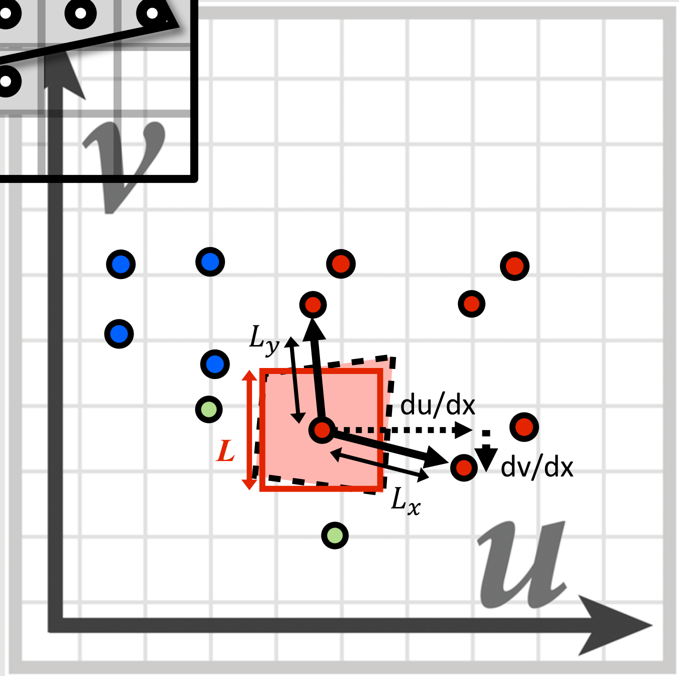
$$L = \sqrt{max(L_x^2, L_y^2)}$$

L measures the Euclidean distance of the change.
We take the max to get a single number.



$$d = \log_2 L$$

[final level d]



The mipmap level is not an integer…
Which level do we use?

Trilinear Interpolation Sampling

- **Idea:** Perform bilinear interpolation on two layers of the mip-map that represents proper minification/magnification, blending the results together

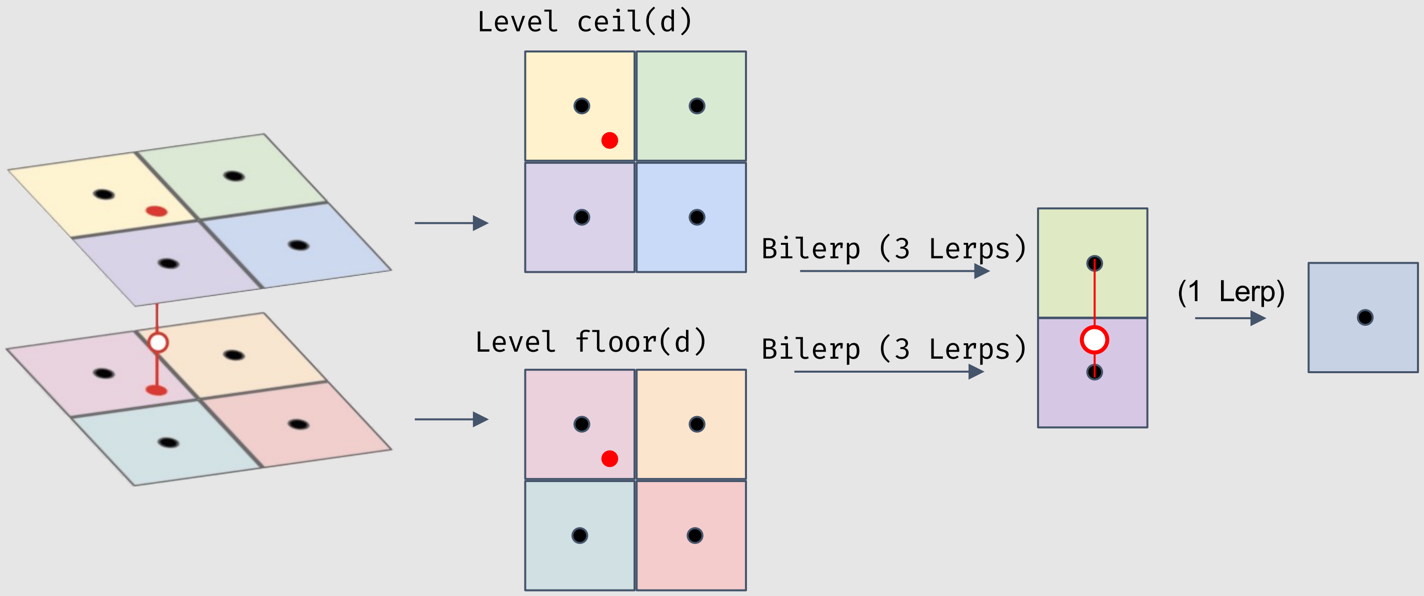
- **Requires:**
  - 8 memory lookup
  - 7 linear interpolations

$$L_x{}^2 \leftarrow \frac{du}{dx}^2 + \frac{dv}{dx}^2$$
$$L_y{}^2 \leftarrow \frac{du}{dy}^2 + \frac{dv}{dy}^2$$

$$L \leftarrow \sqrt{\max(L_x{}^2, L_y{}^2)}$$
$$d \leftarrow log_2 L$$

$$d' \leftarrow floor(d)$$
$$\Delta d \leftarrow d - d'$$

$$t_d \leftarrow tex[d'].bilinear(x, y)$$
$$t_{d+1} \leftarrow tex[d' + 1].bilinear(x, y)$$
$$t \leftarrow (1 - \Delta d) * t_d + \Delta d * t_{d+1}$$



Trilinear Interpolation Sampling

- **Idea:** Perform bilinear interpolation on two layers of the mip-map that represents proper minification/magnification, blending the results together
- **Requires:**
  - 8 memory lookup
  - 7 linear interpolations

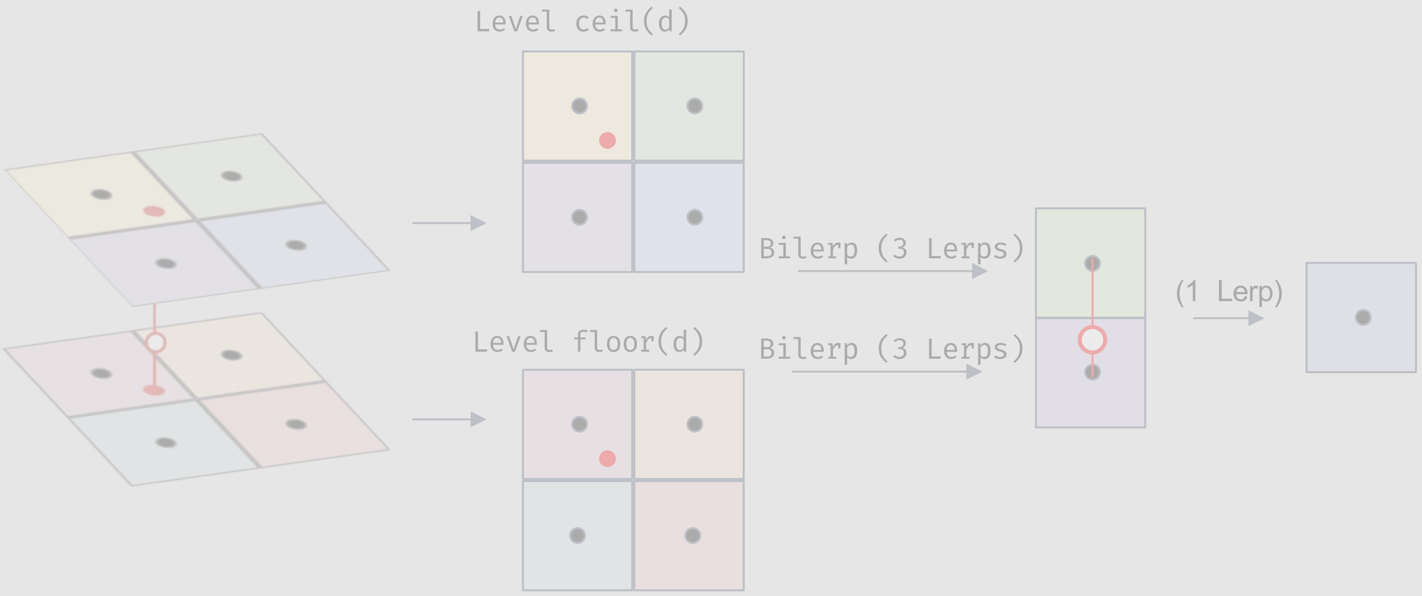
why are we taking the max?

$$L_x{}^2 \leftarrow \frac{du}{dx}^2 + \frac{dv}{dx}^2$$
$$L_y{}^2 \leftarrow \frac{du}{dy}^2 + \frac{dv}{dy}^2$$

$$L \leftarrow \sqrt{\max(L_x{}^2, L_y{}^2)}$$
$$d \leftarrow \log_2 L$$

$$d' \leftarrow floor(d)$$
$$\Delta d \leftarrow d - d'$$

$$t_d \leftarrow tex[d'].bilinear(x,y)$$
$$t_{d+1} \leftarrow tex[d'+1].bilinear(x,y)$$
$$t \leftarrow (1-\Delta d) * t_d + \Delta d * t_{d+1}$$

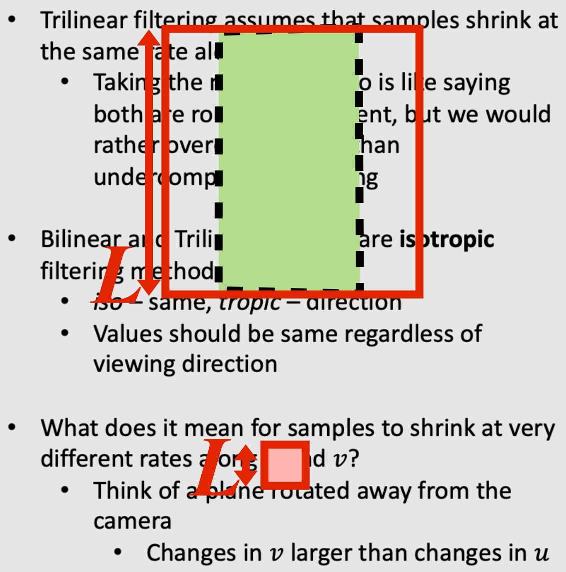


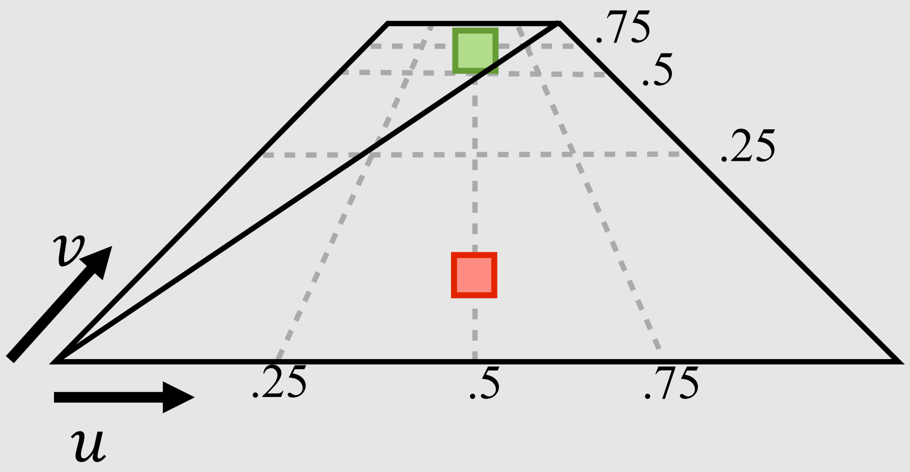
Trilinear Assumption

- Trilinear filtering assumes that samples shrink at the same rate along u and v
  - Taking the max says we would rather overcompensate than undercompensate filtering

- Bilinear and Trilinear filtering are **isotropic** filtering methods
  - ***iso* – same, *tropic* – direction**
  - Values should be same regardless of viewing direction

- What does it mean for samples to shrink at very different rates along u and v?
  - Think of a plane rotated away from the camera
    - Changes in v larger than changes in u





Anisotropic Filtering

- **Anisotropic** filtering is dependent on direction
  - ***an* – not, iso – same, *tropic* – direction**

- **Idea:** create a new texture map that downsamples the x and y axis by 2 separately
  - Instead of taking the max, use each coordinate to index into correct location in map

$$L = \sqrt{\max(L_x^2, L_y^2)}$$

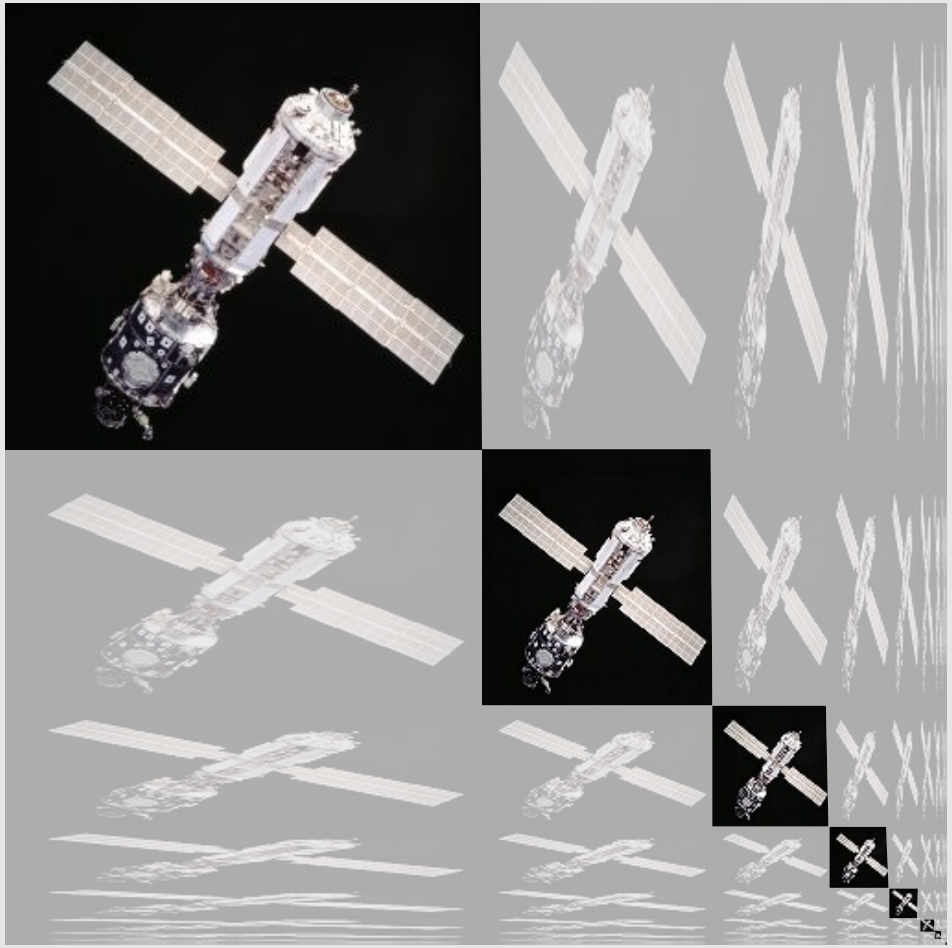
$$(d_x, d_y) = (\,log_2\sqrt{L_x^2}\,, log_2\sqrt{L_y^2}\,)$$

- Texture map is now a grid of downsampled textures
  - Known as a RipMap

Rip Map

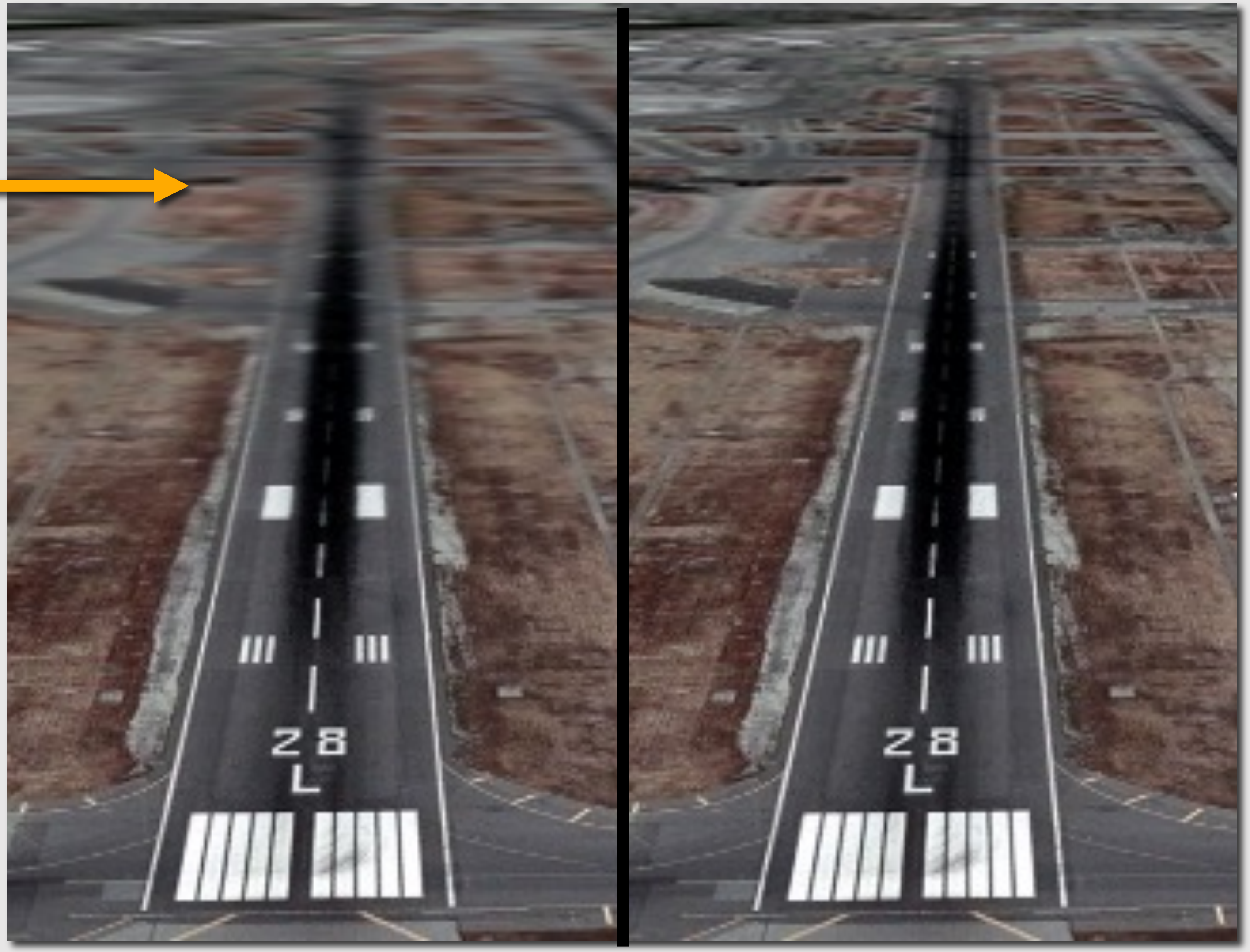
- Same idea as MipMap, but for anisotropic filtering
  - 4x memory footprint
  - New width: $w' = w + \frac{w}{2} + \frac{w}{4} + \cdots = 2w$
  - New height: $h' = h + \frac{h}{2} + \frac{h}{4} + \cdots = 2h$
    - New area: $w'h' = 4wh$

- **Fun fact:** a MipMap is just the diagonal of a RipMap
  - If $d_x = d_y$, then we have trilinear interpolation



Isotropic vs Anisotropic Filtering

overbluring in u direction



[isotropic (trilinear)]

[anisotropic]

Sampling Comparisons

	[Nearest]	[Bilinear]	[Trilinear]	[Anisotropic]
No. samples	1	4	8	16
No. interps	0	3	7	15
No. operations	~3	~19	>54	>54
Texture locality	good	good	bad	very bad
Memory overhead	1x	1x	4/3x	4x
Anti-aliasing	bad	normal	good	great

Texture Sampling Pipeline

1. Compute u and v from screen sample (x,y) via barycentric interpolation
2. Approximate du/dx, du/dy, dv/dx, dv/dy by taking differences of screen-adjacent samples
3. Compute mip map level d
4. Convert normalized [0,1] texture coordinate (u,v) to pixel locations $(U,V) \in [W,H]$ in texture image
5. Determine addresses of texels needed for filter (e.g., eight neighbors for trilinear)
6. Load texels into local registers
7. Perform tri-linear interpolation according to (U,V,d)
8. (…even more work for anisotropic filtering…)

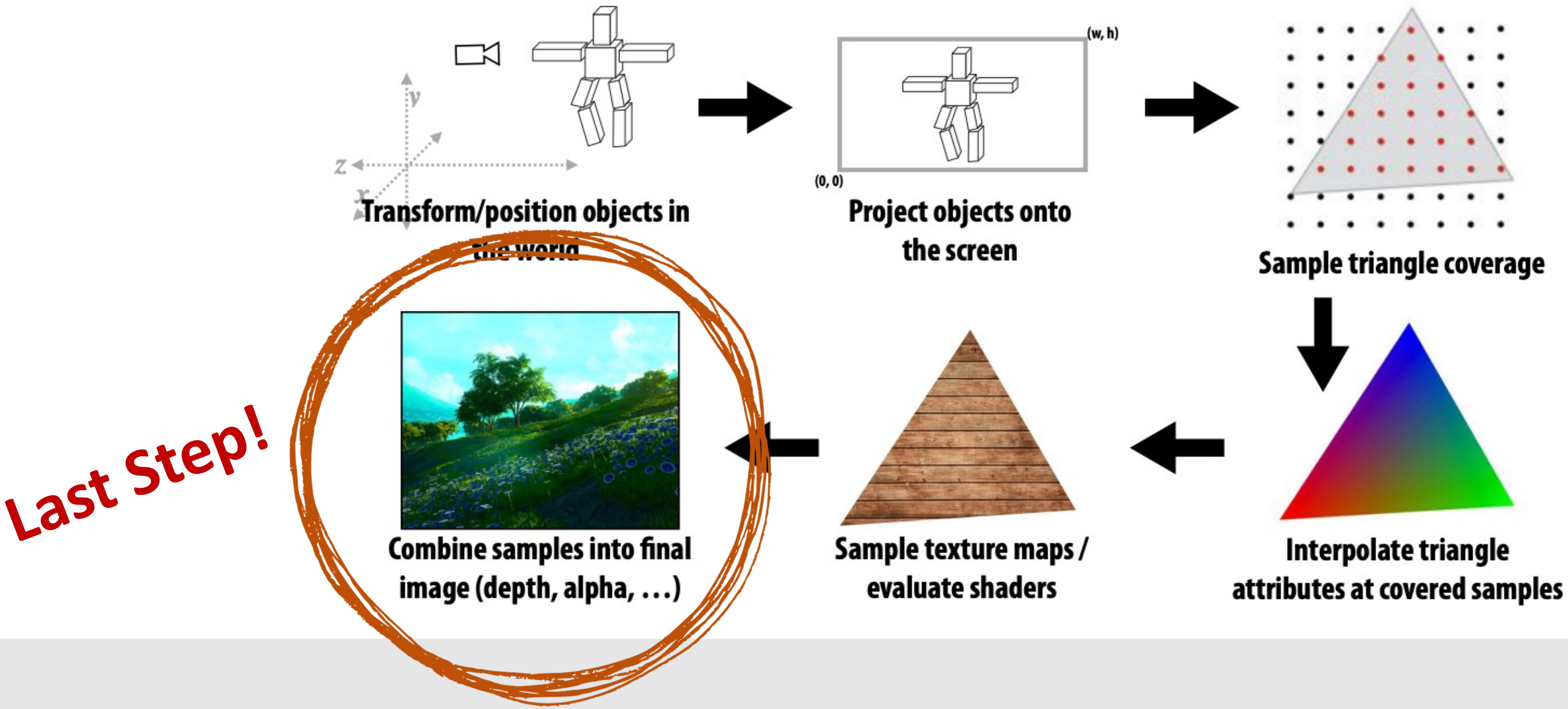
Lot of repetitive work every time we want to shade a pixel!

GPUs instead implement these instructions on fixed-function hardware.

This is why we have texture caches and texture filtering units.

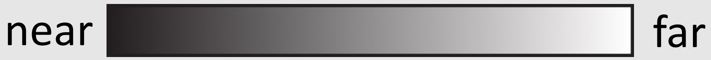
- ~~Barycentric Coordinates~~
- ~~Texturing Surfaces~~
- Depth Testing
- Alpha Blending
- The Graphics Pipeline Revisited

The "Simpler" Graphics Pipeline

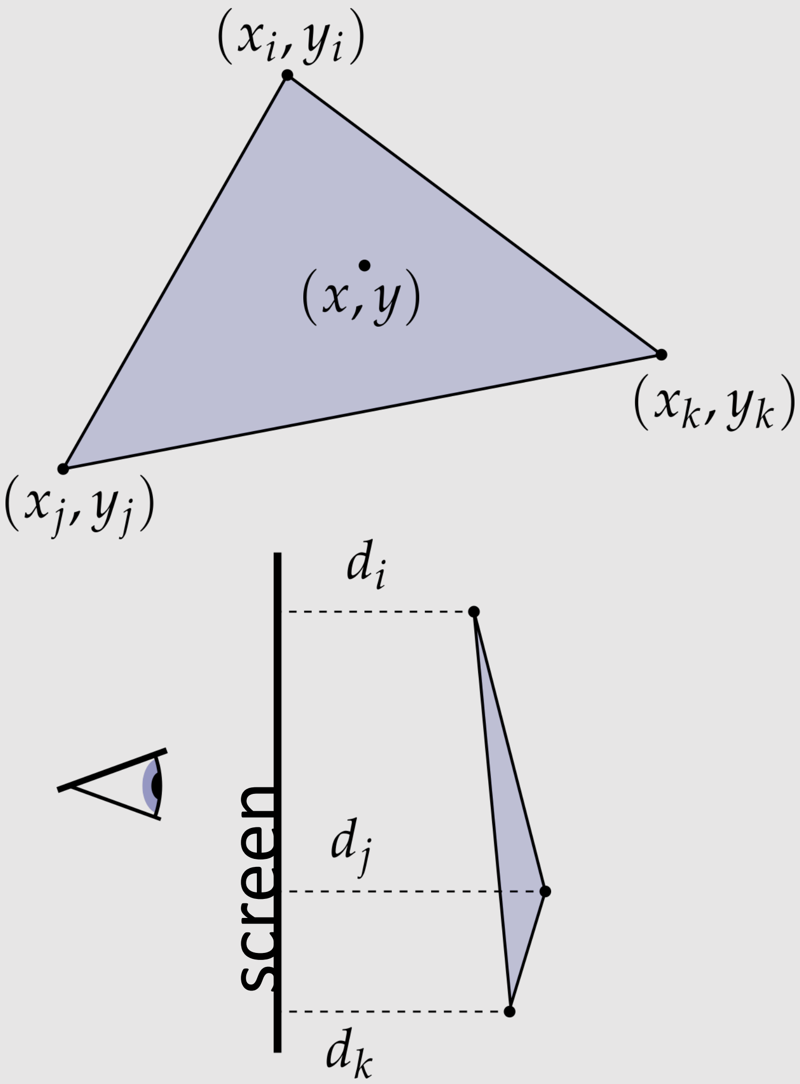


Depth Buffer (Z-buffer)

- For each **sample**, the depth buffer stores the depth of the closest triangle seen so far
  - Done at the sample granularity, not pixel granularity

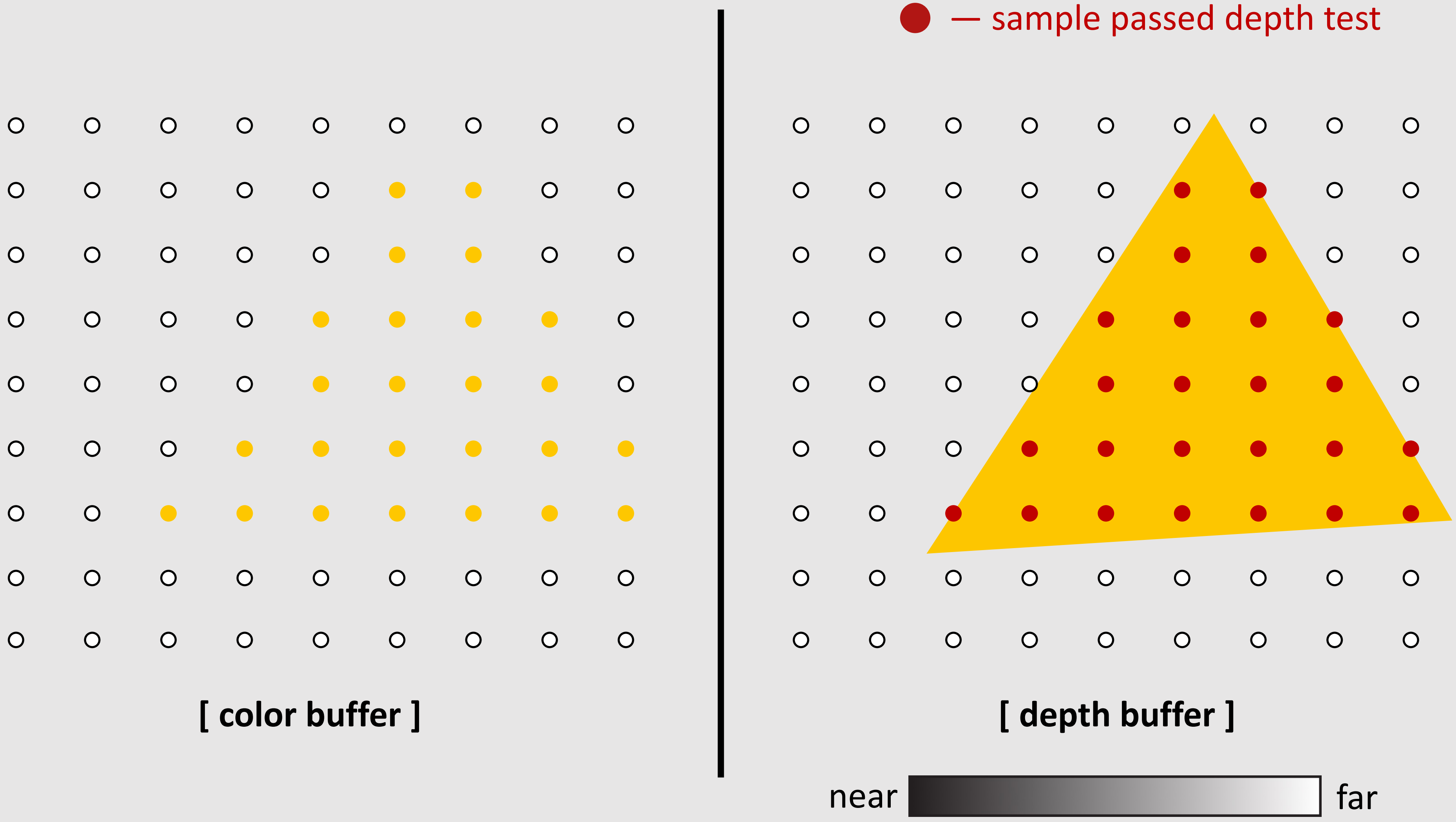


Depth of a Triangle

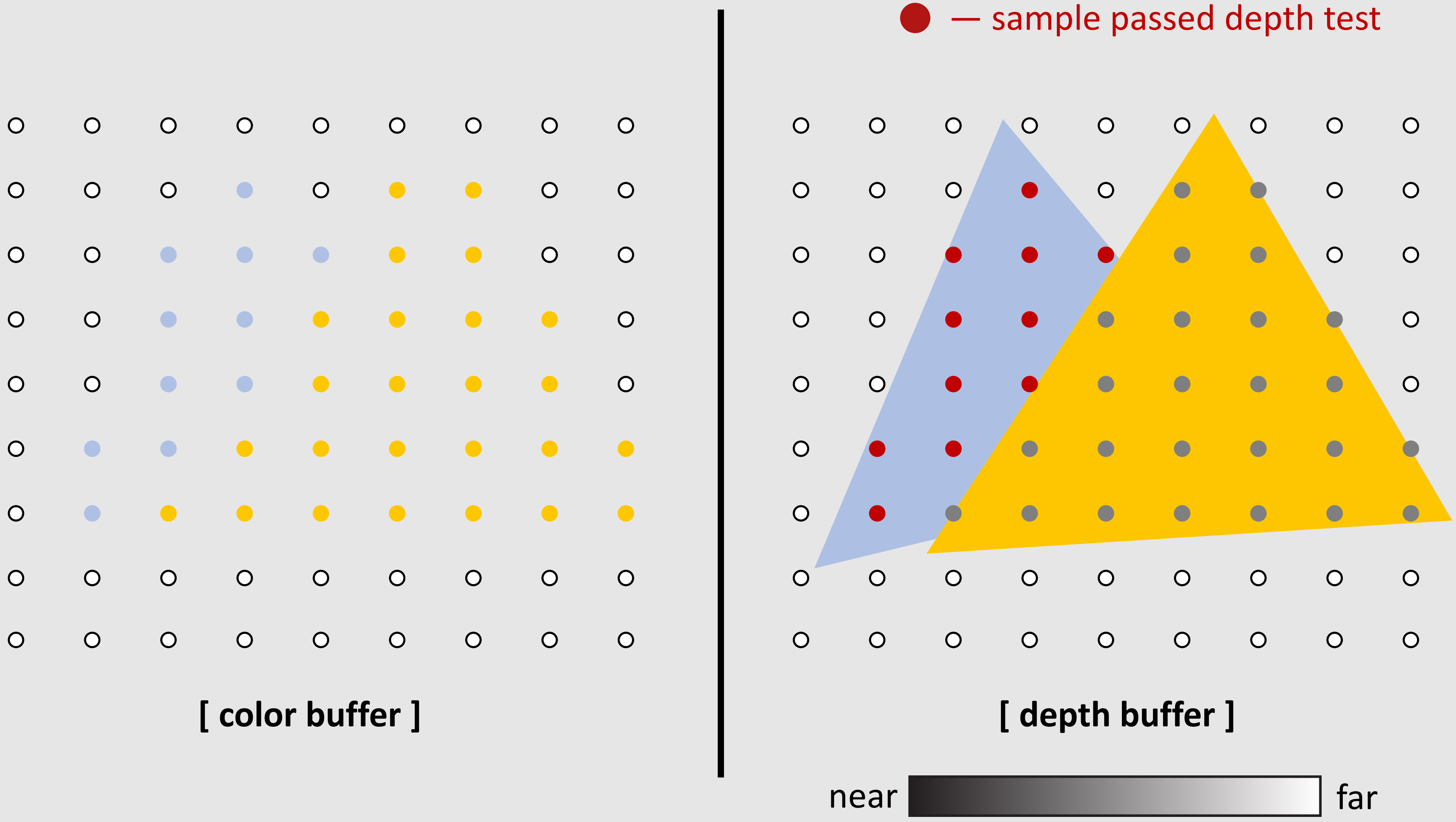


- A triangle is composed of 3 different 3D points, each with a depth value z
- To get the depth at any point (x, y) inside the triangle, interpolate depth at vertices with barycentric coordinates

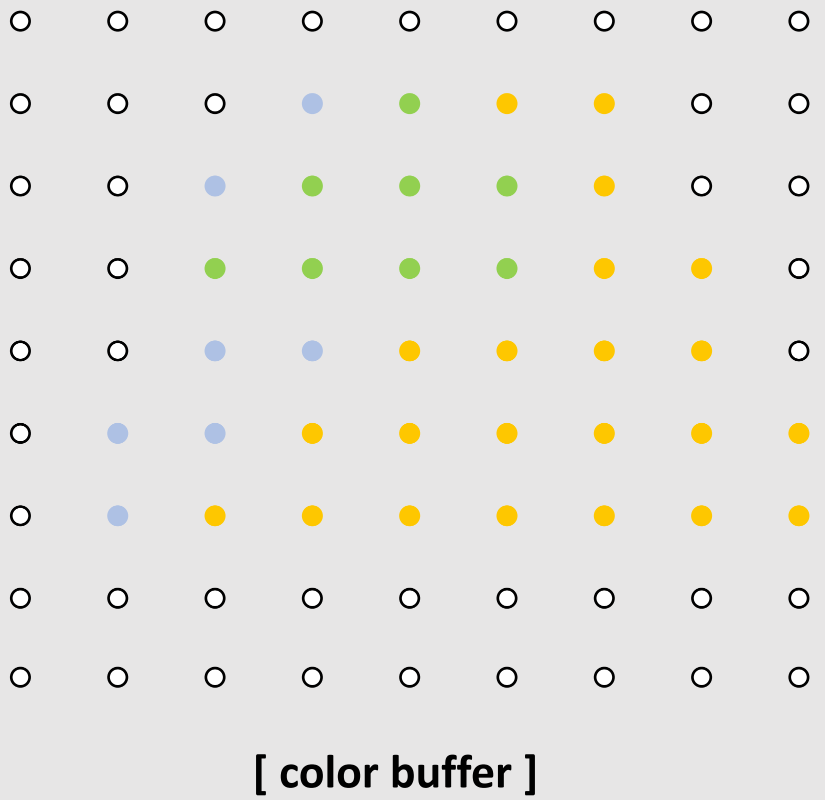
Depth Buffer (Z-buffer)

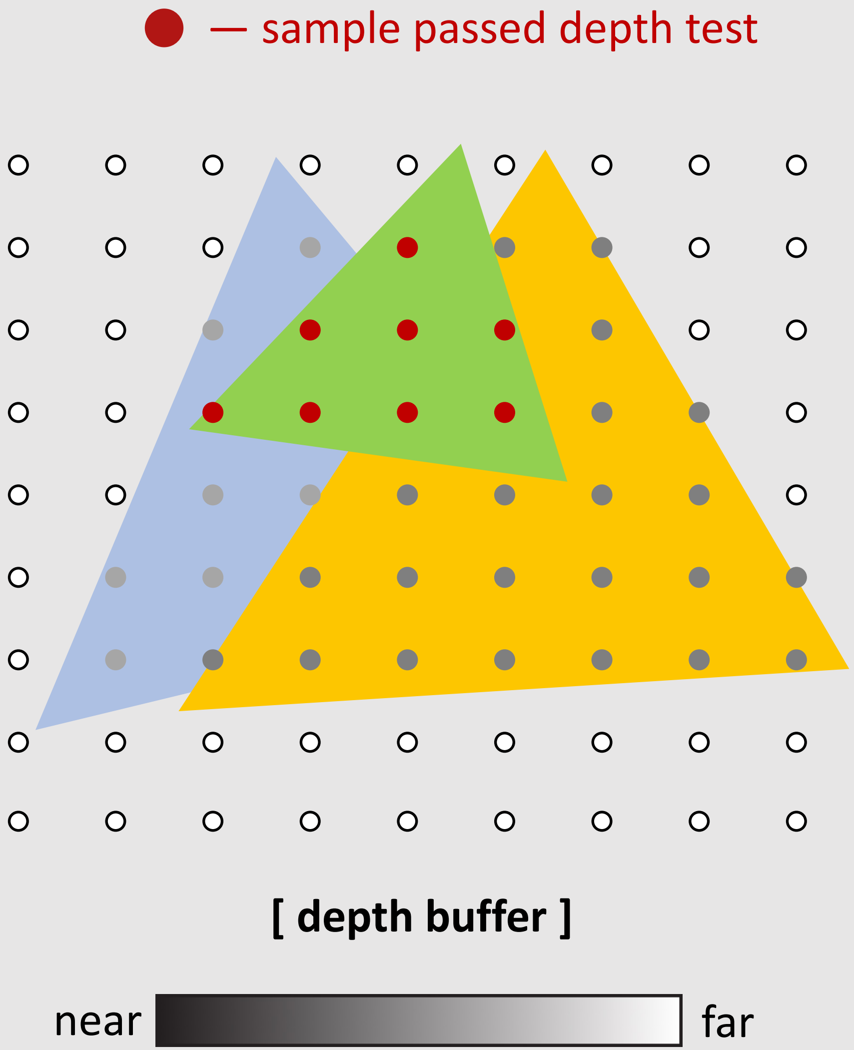


Depth Buffer (Z-buffer)

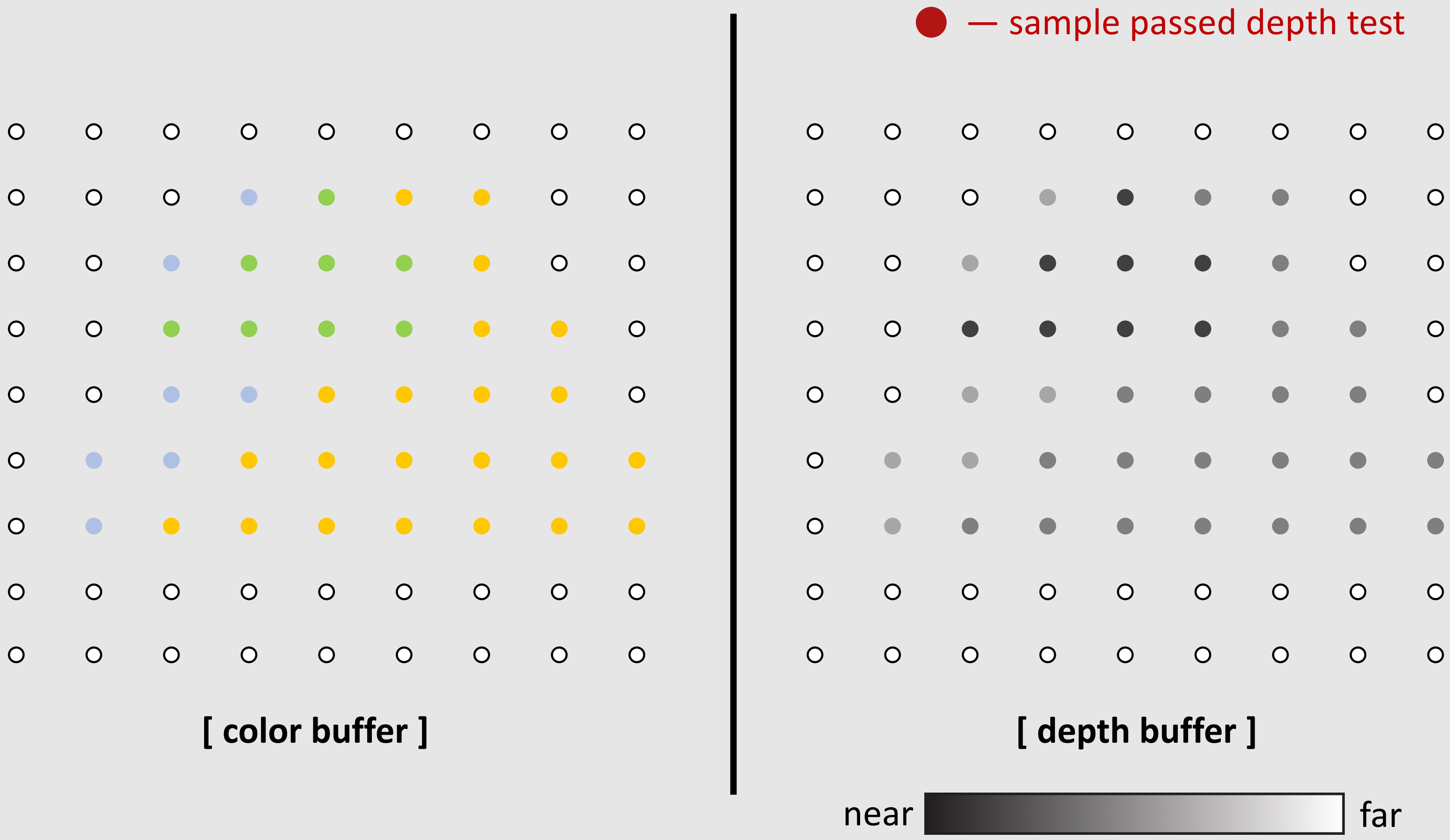


Depth Buffer (Z-buffer)

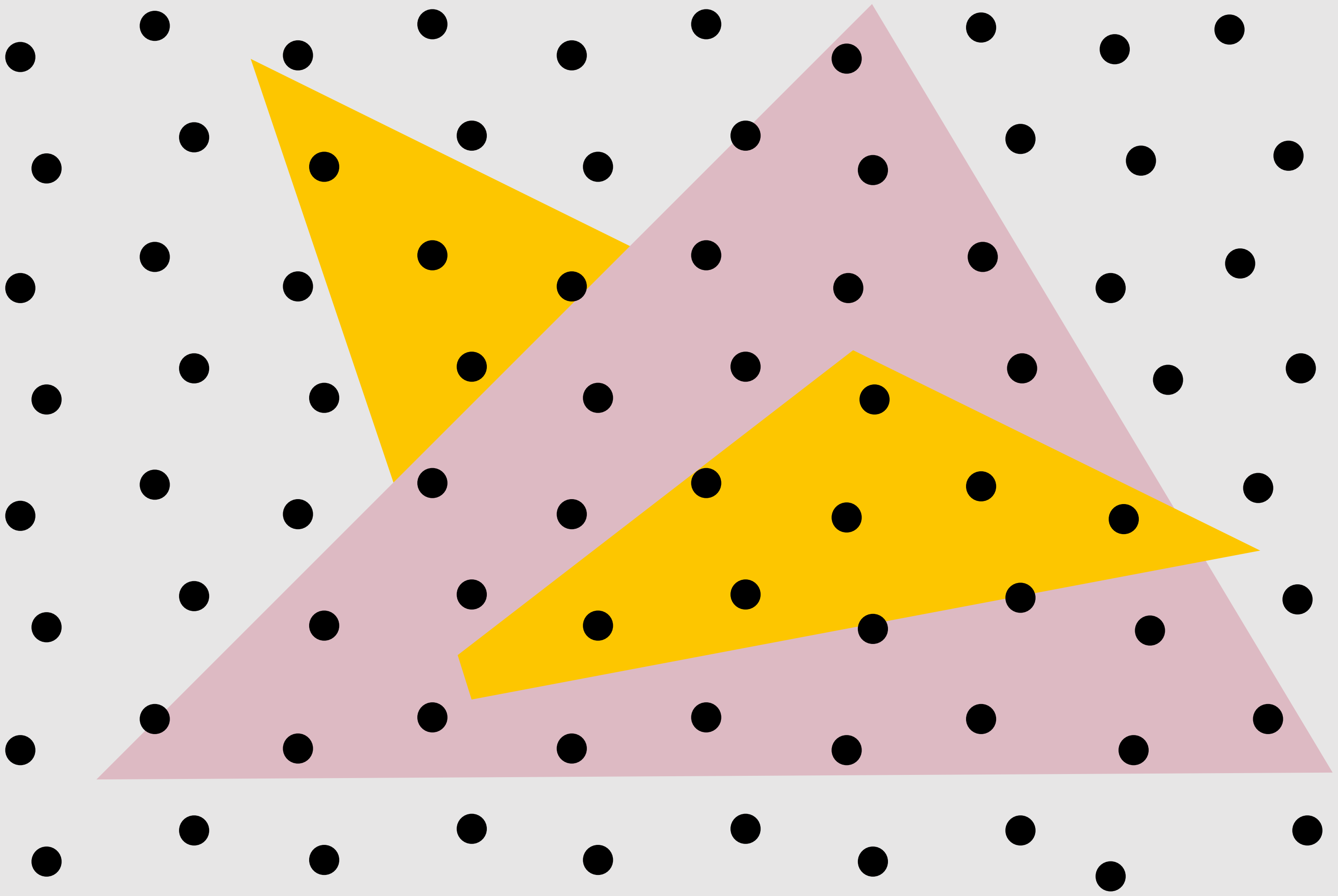




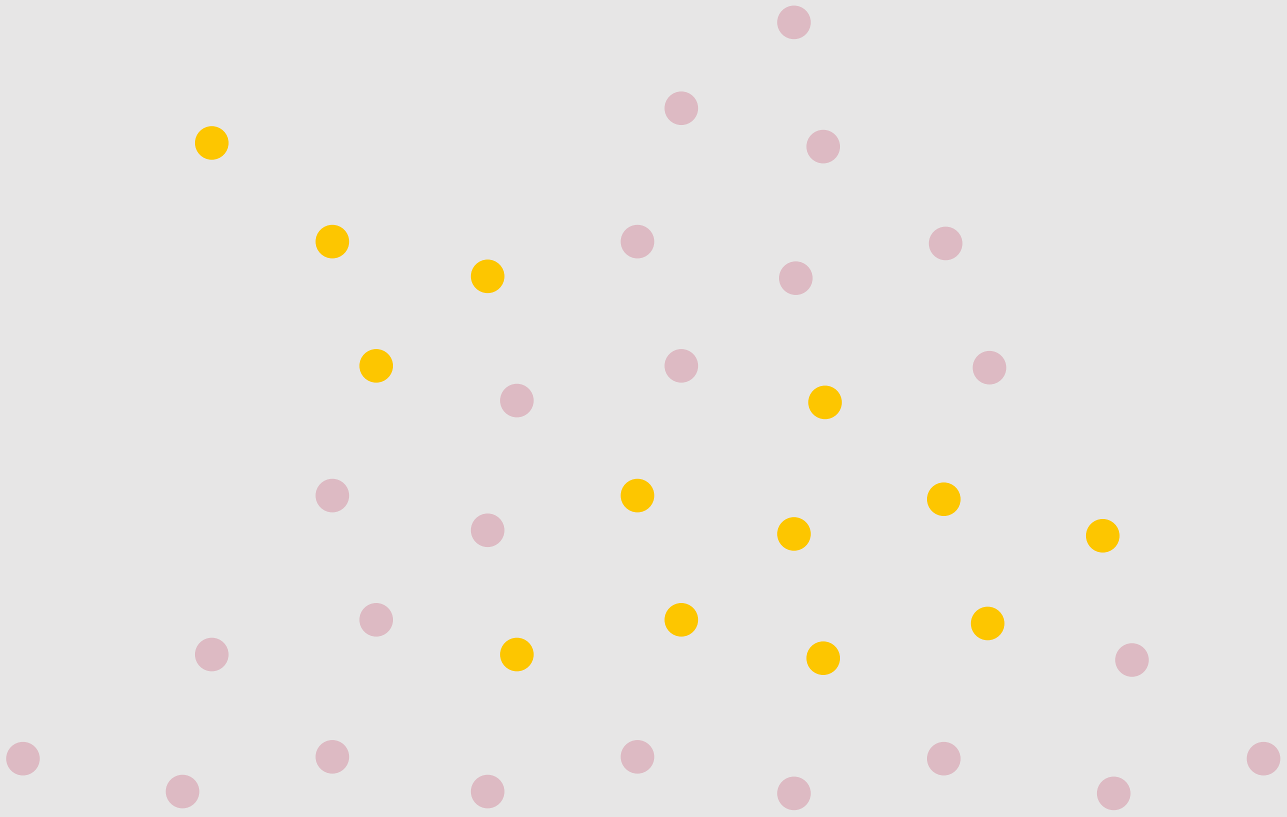
Depth Buffer (Z-buffer)



Depth Buffer (Z-buffer) Per Sample



Depth Buffer (Z-buffer) Per Sample



Able to capture triangle intersections by performing tests per sample

Depth Buffer (Z-buffer) Sample Code

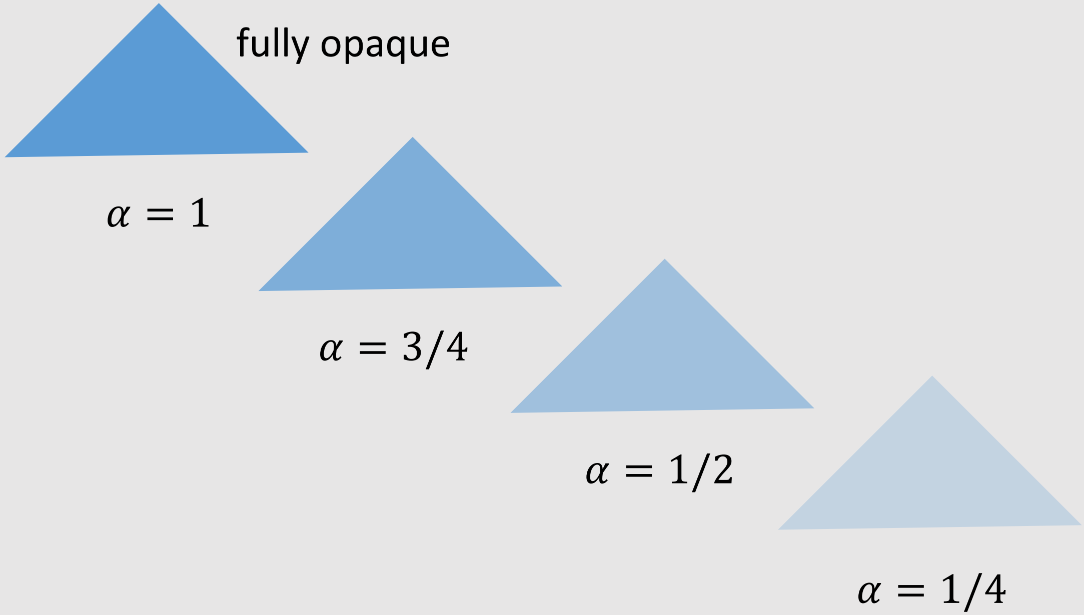
```
draw_sample(x, y, d, c) //new depth d & color c at (x,y)
{
  if(d < zbuffer[x][y])
  {
    // triangle is closest object seen so far at this
    // sample point. Update depth and color buffers.
    zbuffer[x][y] = d;  // update zbuffer
    color[x][y] = c;    // update color buffer
  }
  // otherwise, we've seen something closer already;
  // don't update color or depth
}
```

Why is it that we first shade the pixel and then assign the resulting color after depth check?
Deferred shading (advanced algorithm) fixes this issue.

- ~~Barycentric Coordinates~~
- ~~Texturing Surfaces~~
- ~~Depth Testing~~
- Alpha Blending
- The Graphics Pipeline Revisited

Alpha Values

- Another common image format: RGBA
  - Alpha channel specifies ‘opacity’ of object
  - Basically how transparent it is
  - Most common encoding is 8-bits per channel (0-255)
- Compositing A over B != B over A
  - Consider the extreme case of two opaque objects…



$\alpha = 0$

fully transparent

[koala over nyc]

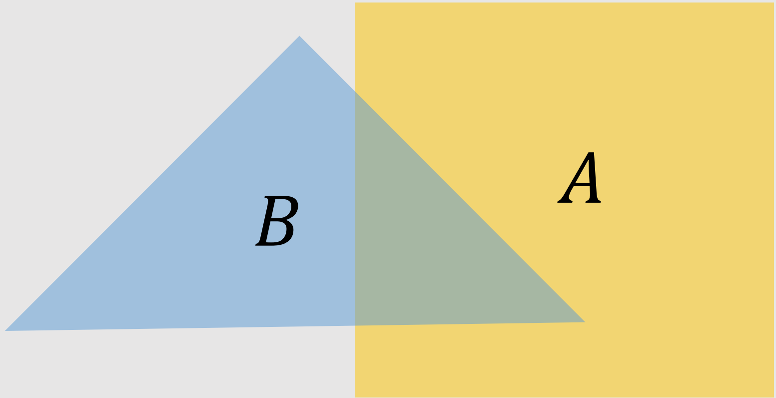
[nyc over…koala?]

where is the koala…

Non-Premultiplied Alpha

- **Goal:** Composite image B with alpha α_B over image A with alpha α_A

$$A = (A_r, A_g, A_b)$$
$$B = (B_r, B_g, B_b)$$



B over A

- Composite RGB:

what B lets through

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

appearance of semi-transparent B

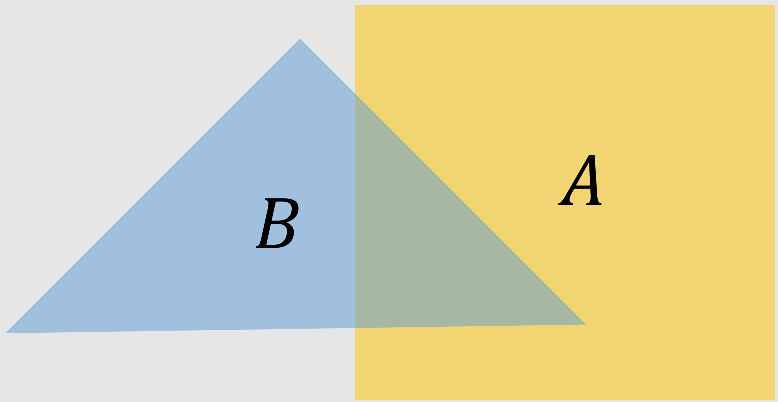
appearance of semi-transparent A

- Composite Alpha:

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

Two different equations is inefficient!!

Premultiplied Alpha



B over A

- **Goal:** Composite image B with alpha α_B over image A with alpha α_A

$$A' = (\alpha_A A_r, \alpha_A A_g, \alpha_A A_b, \alpha_A)$$
$$B' = (\alpha_B B_r, \alpha_B B_g, \alpha_B B_b, \alpha_B)$$

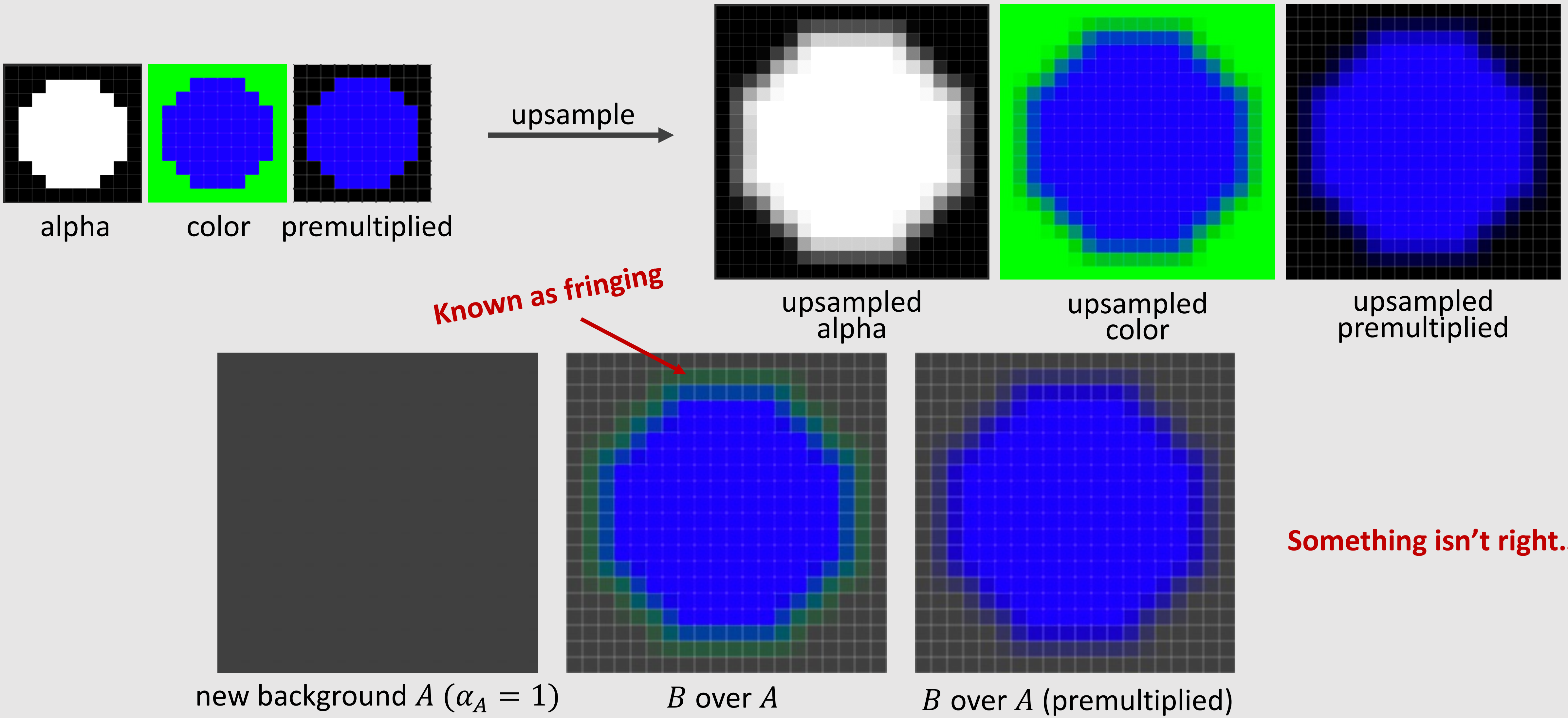
- Composite RGBA:

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

- Un-Premultiply for Final Color:

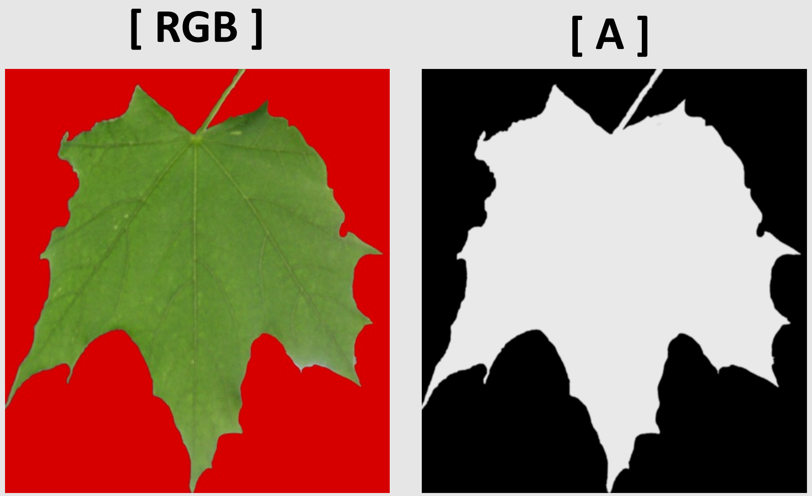
$$(C_r, C_g, C_b, \alpha_C) \Longrightarrow (C_r/\alpha_C, C_g/\alpha_C, C_b/\alpha_C)$$

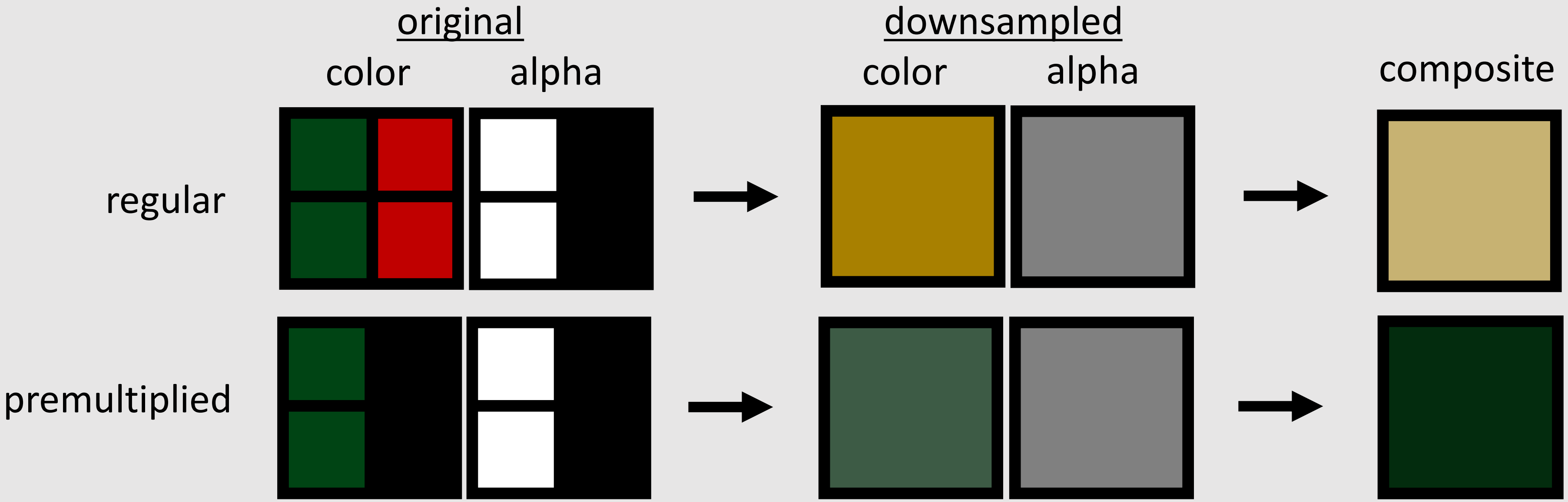
Why Premultiplied Matters [Upsample]



Something isn't right…

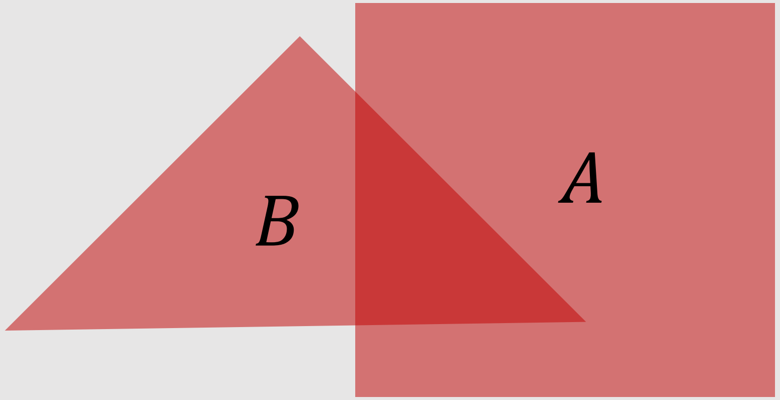
Why Premultiplied Matters [Downsample]





Closed Under Composition

- **Goal:** Composite bright red image B with alpha 0.5 over bright red image A with alpha 0.5



B over A

$$A = (1, 0, 0, 0.5)$$
$$B = (1, 0, 0, 0.5)$$

- Non-Premultiplied:

color

$$0.5 * (1,0,0) + (1 - 0.5) * 0.5 * (1,0,0)$$
$$\downarrow$$
$$(0.75, 0, 0)$$

alpha

$$0.5 + (1 - 0.5) * 0.5 = 0.75$$

- Premultiplied:

$$0.5 * (0.5,0,0,0.5) + (1 - 0.5) * (0.5,0,0,0.5)$$
$$\downarrow$$
$$(0.75, 0, 0, 0.75)$$

↓ divide out alpha

$$(1, 0, 0)$$

Blend Methods

When writing to color buffer, can use any blend method

$$D_{RGBA} = S_{RGBA} + D_{RGBA}$$
$$D_{RGBA} = S_{RGBA} - D_{RGBA}$$
$$D_{RGBA} = -S_{RGBA} + D_{RGBA}$$
$$D_{RGBA} = \min(S_{RGBA}, D_{RGBA})$$
$$D_{RGBA} = \max(S_{RGBA}, D_{RGBA})$$
$$D_{RGBA} = S_{RGBA} + D_{RGBA} * (1 - S_A)$$

Blend Add
Blend Subtract
Blend Reverse Subtract
Blend Min
Blend Max
Blend Over

S_{RGBA} and D_{RGBA} are pre-multiplied

Updated Depth Buffer (Z-buffer) Sample Code

```
draw_sample(x, y, d, c) //new depth d & color c at (x,y)
{
  if (d < zbuffer[x][y])
  {
    // triangle is closest object seen so far at this
    // sample point. Update depth and color buffers.
    zbuffer[x][y] = d;
    color[x][y] = c.rgba + (1-c.a) * color[x][y];
  }
  // otherwise, we've seen something closer already;
  // don't update color or depth
}
```

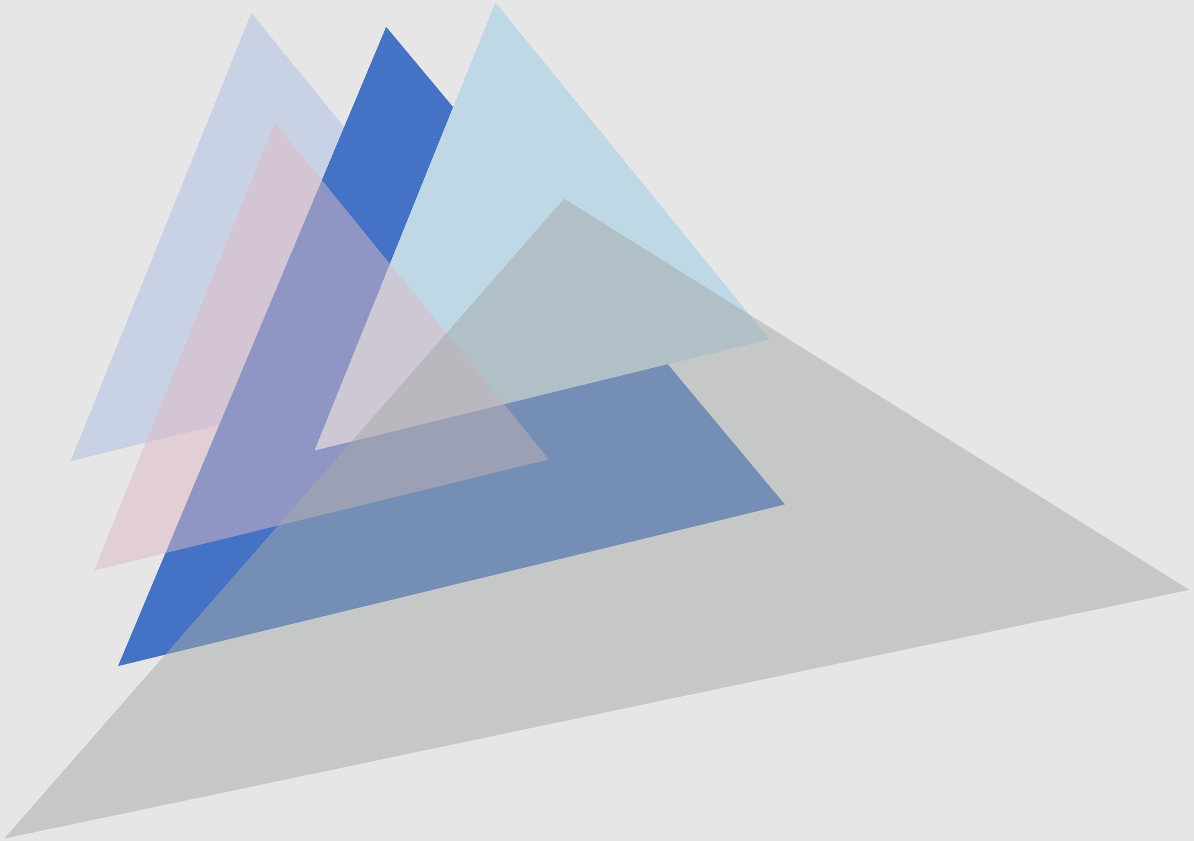
Should we still be doing depth writes for alpha primitives?

Assumes color[x][y] and c are both premultiplied.

Triangles must be rendered back to front!
A over B != B over A

Blend Render Order

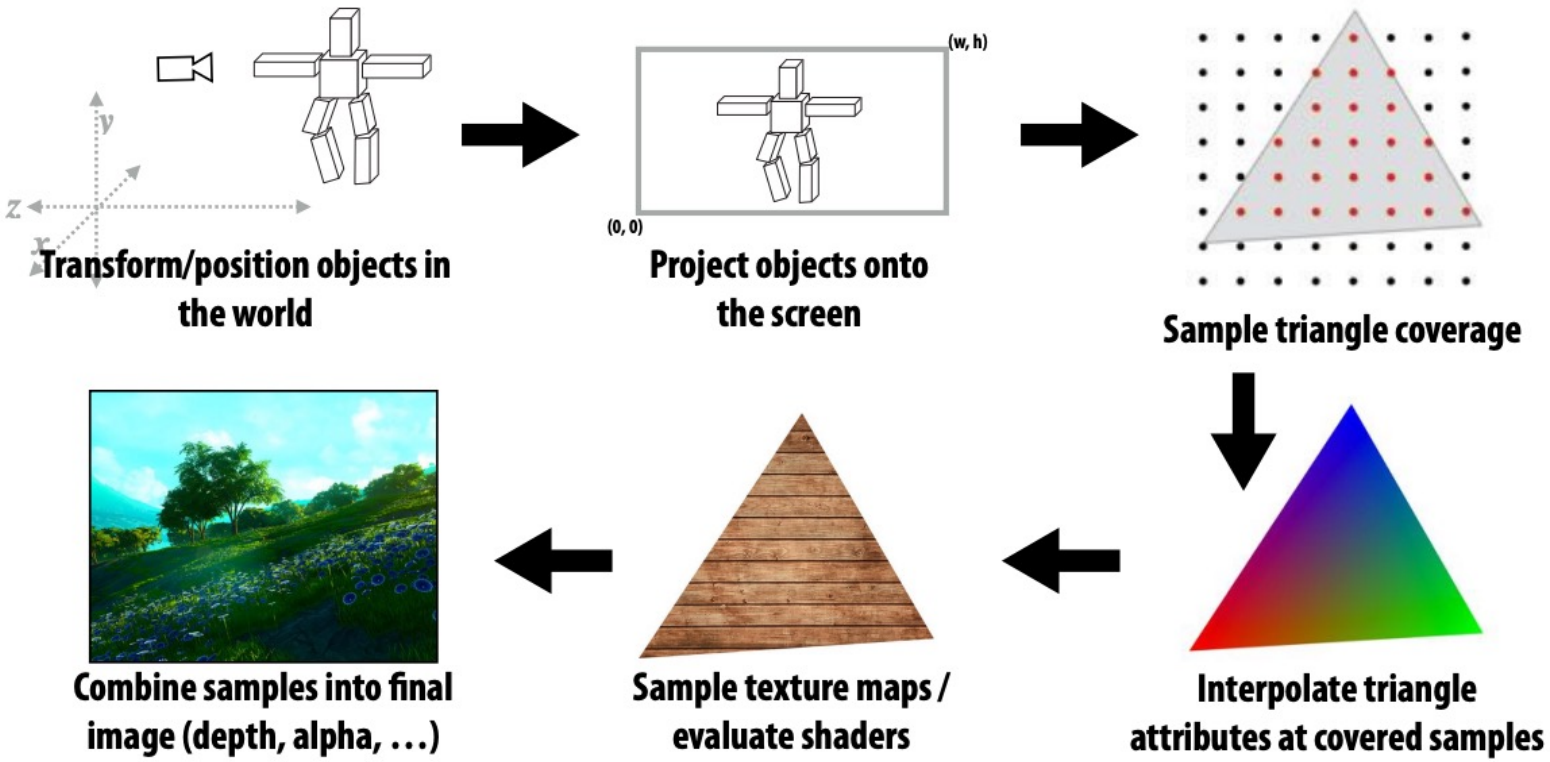
- For mixtures of opaque and transparent triangles:
  - **Step 1:** render opaque primitives (in any order) using depth-buffered occlusion
    - If pass depth test, triangle overwrites value in color buffer at sample
    - Depth **READ** and **WRITE**
  - **Step 2:** disable depth buffer update, render semi-transparent surfaces in back-to-front order.
    - If pass depth test, triangle is composited OVER contents of color buffer at sample
    - Depth **READ** only



- ~~Barycentric Coordinates~~
- ~~Texturing Surfaces~~
- ~~Depth Testing~~
- ~~Alpha Blending~~
- The Graphics Pipeline Revisited

The “Simpler” Graphics Pipeline

Now Let’s Put It All Together!

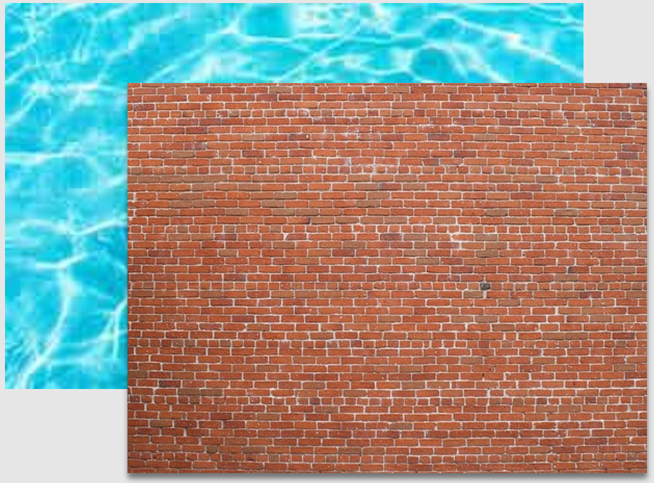


The Inputs

```
positions = {
    v0x, v0y, v0z,
    v1x, v1y, v1x,
    v2x, v2y, v2z,
    v3x, v3y, v3x,
    v4x, v4y, v4z,
    v5x, v5y, v5x
};
```

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

[vertices]



[textures]

Object-to-camera-space transform $T \in \mathbb{R}^{4\times4}$

Perspective projection transform $P \in \mathbb{R}^{4\times4}$

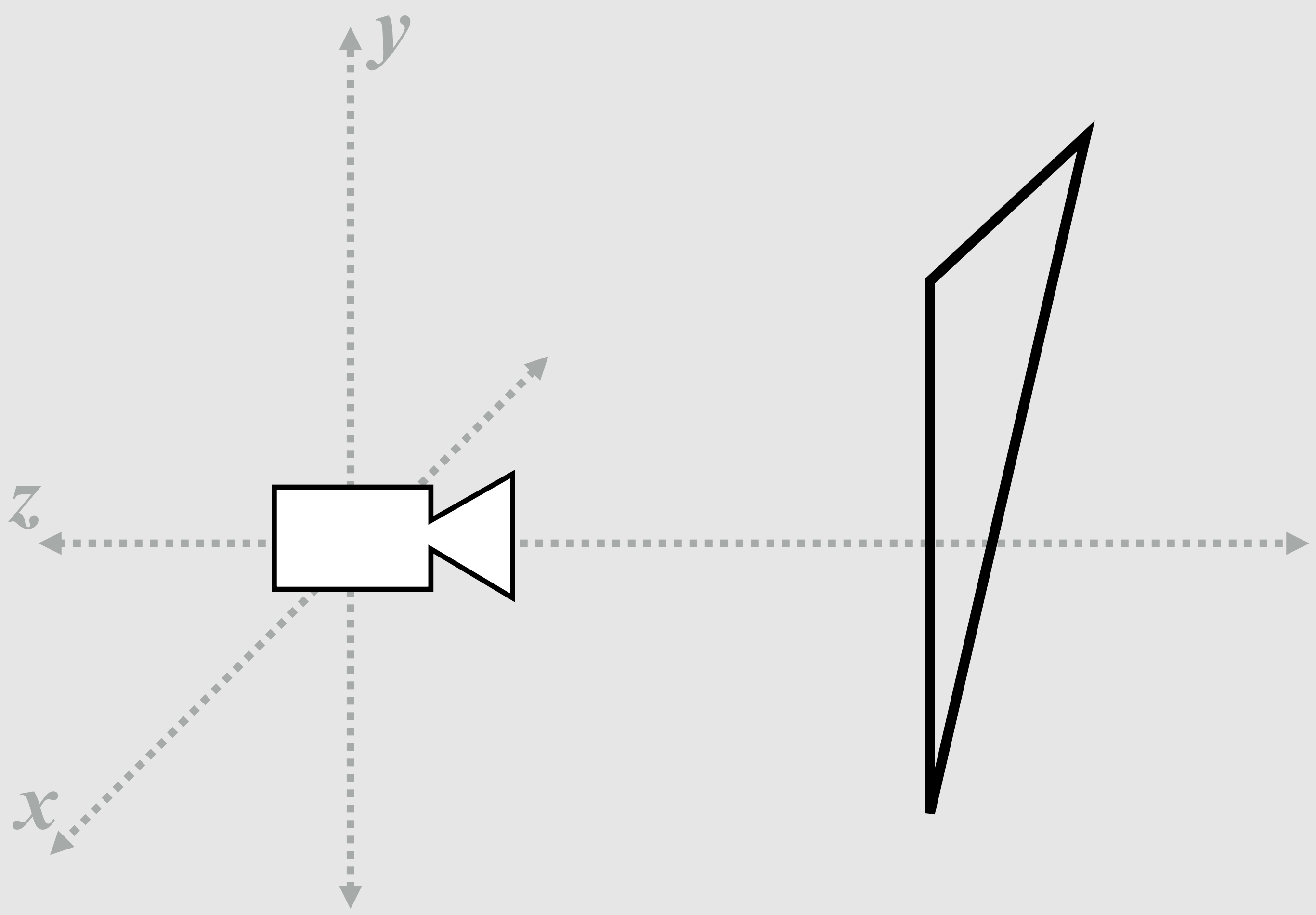
Output image (W, H)

[camera properties]

[machine]

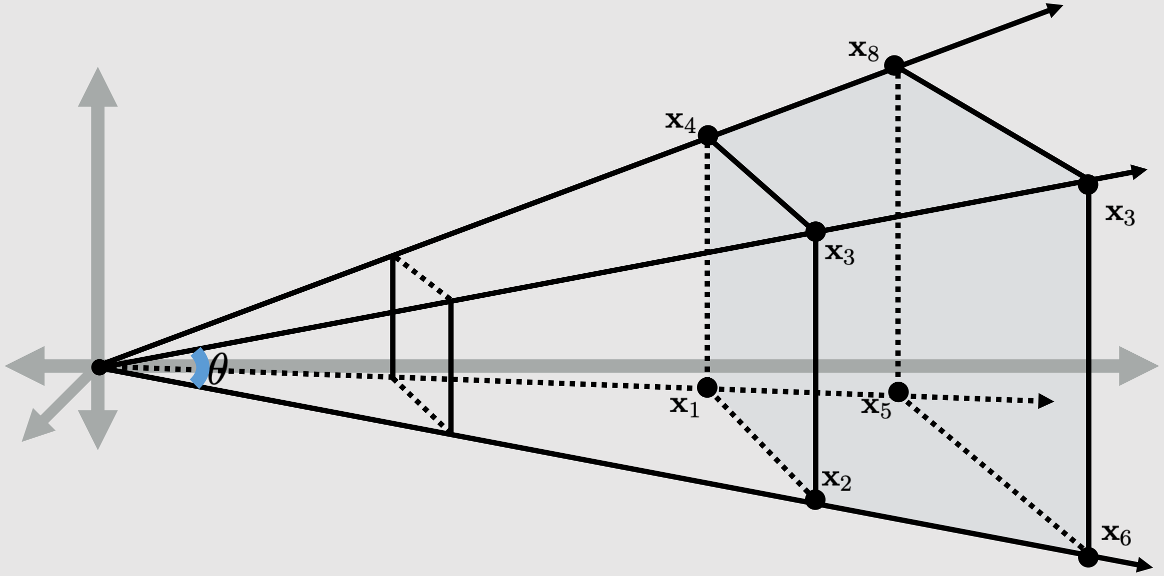
Step 1: Transform

Transform triangle vertices into camera space

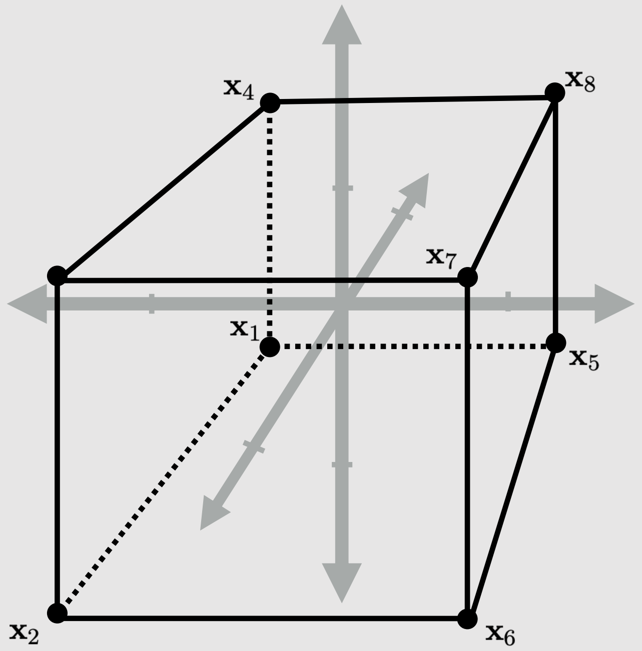


Step 2: Perspective Projection

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



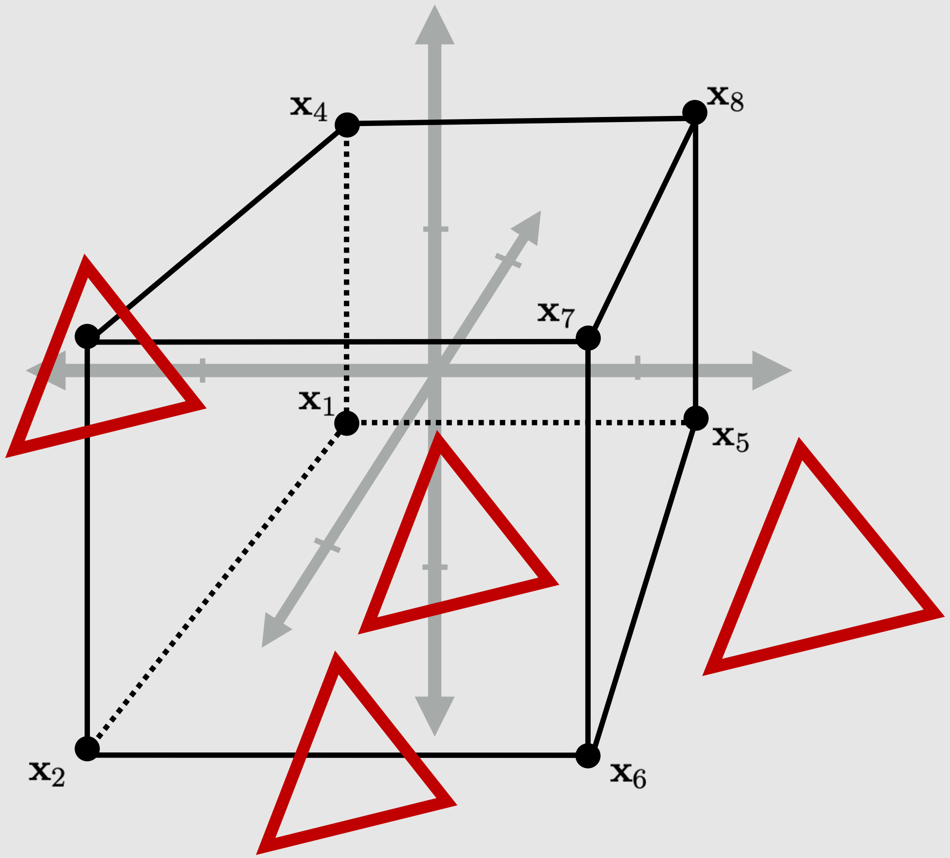
[3D camera space position]



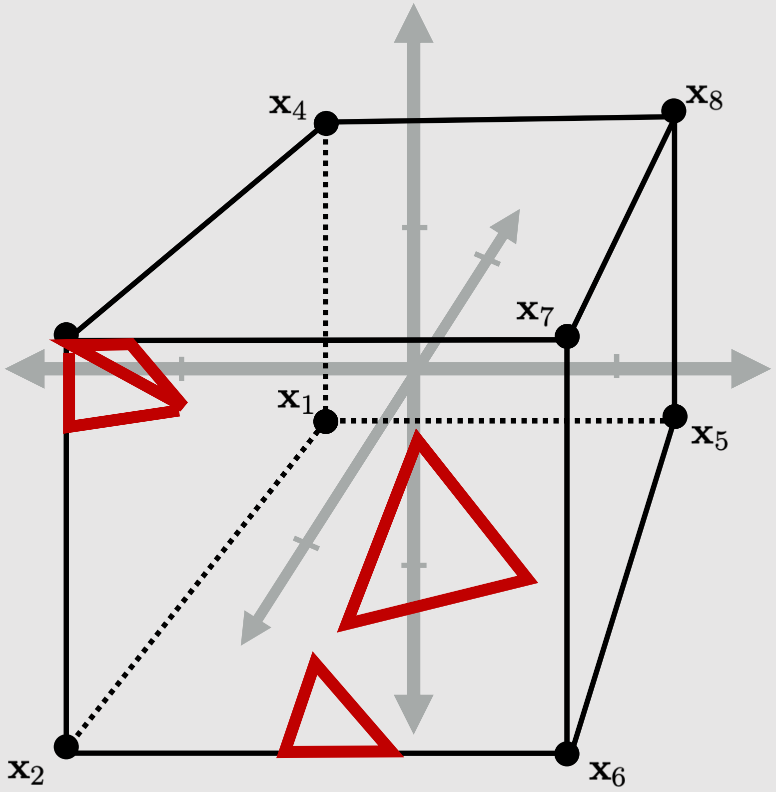
[normalized space position]

Step 3: Clipping

Discard triangles completely outside cube.
Clip triangles partially in cube.



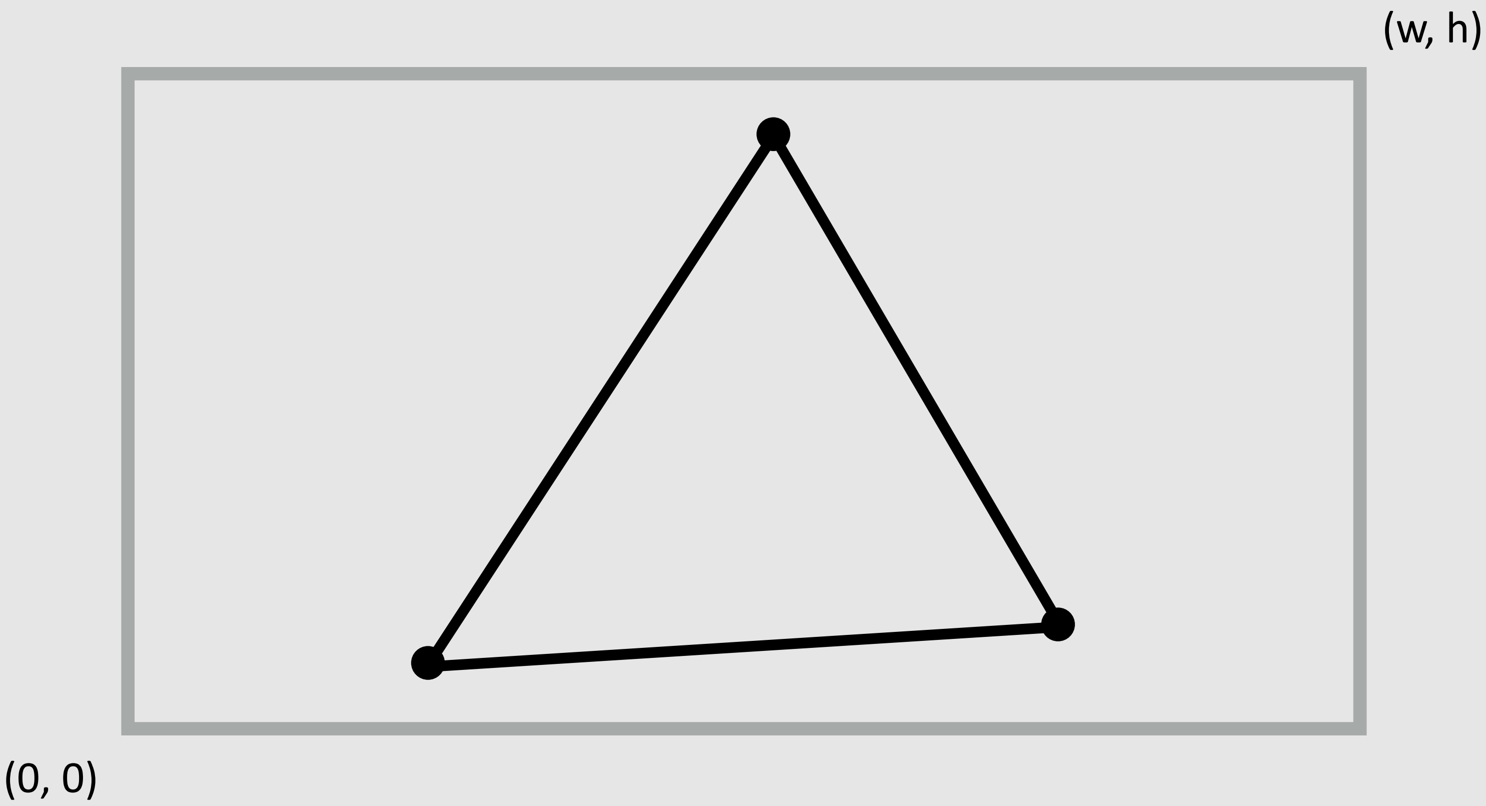
[pre-clipping]



[post-clipping]

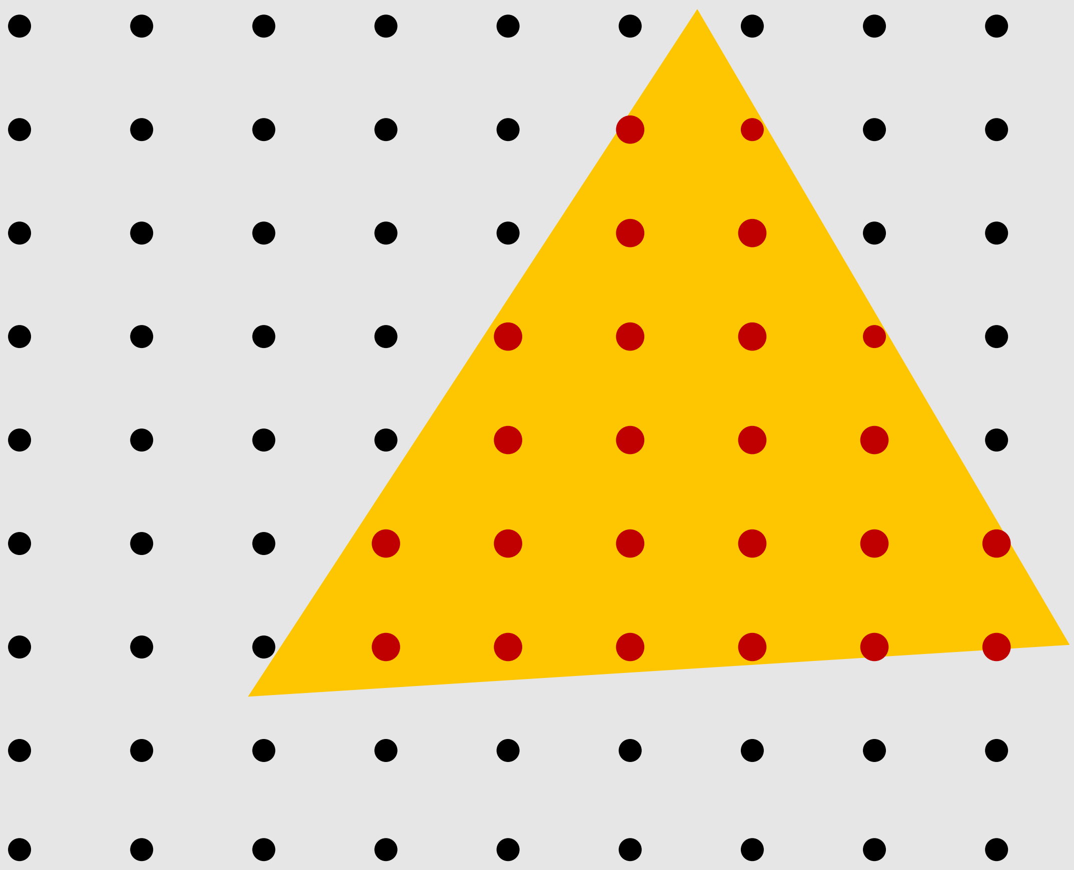
Step 4: Transform To Screen Coordinates

Perform homogeneous divide.
Transform vertex xy positions from normalized coordinates into screen coordinates (based on screen [w, h]).



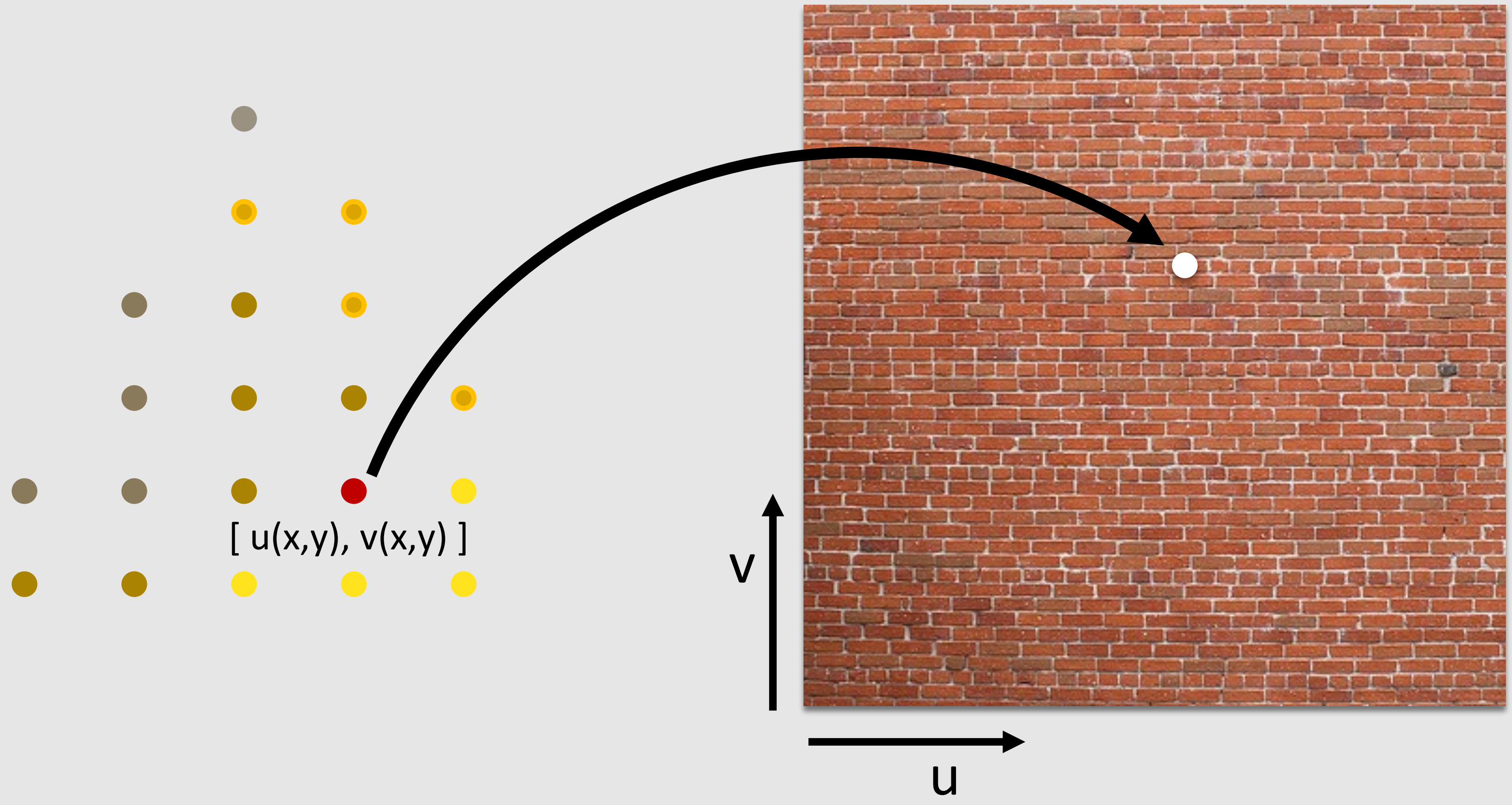
Step 5: Sample Coverage

Check if samples lie inside triangle.
Evaluate depth and barycentric coordinates at all passing samples.



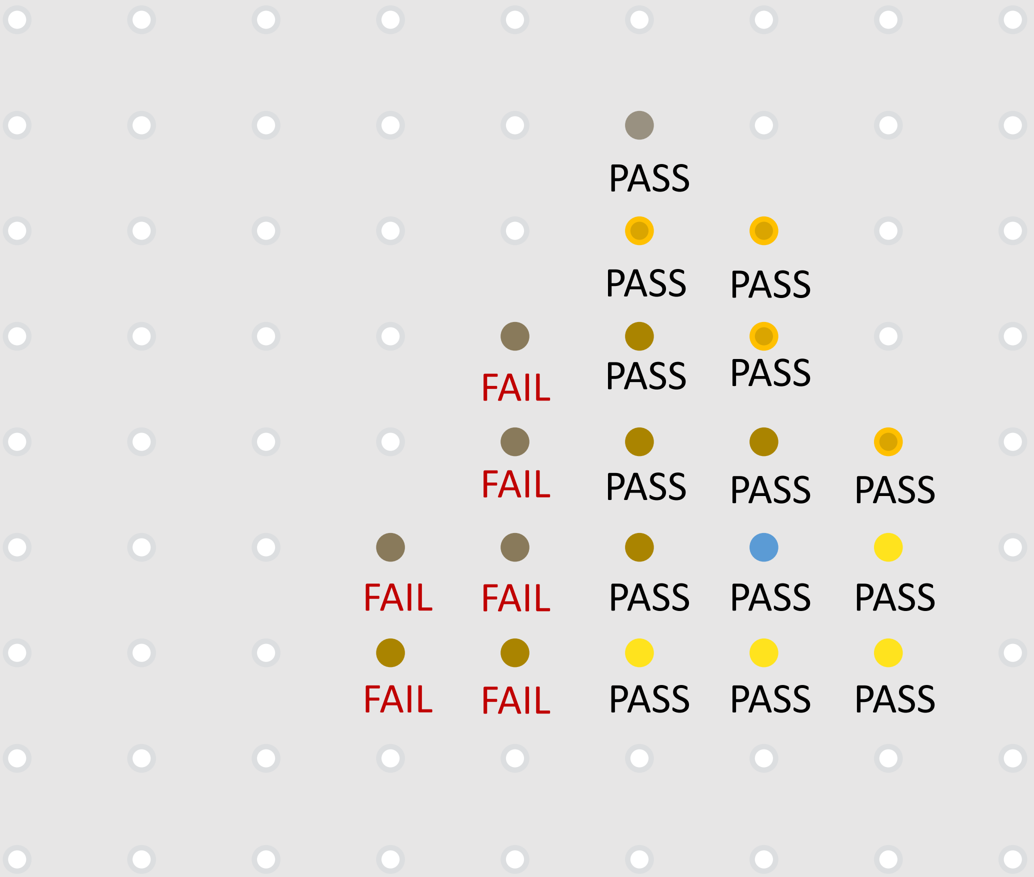
Step 6: Compute Color

Texture lookups, color interpolation, etc.



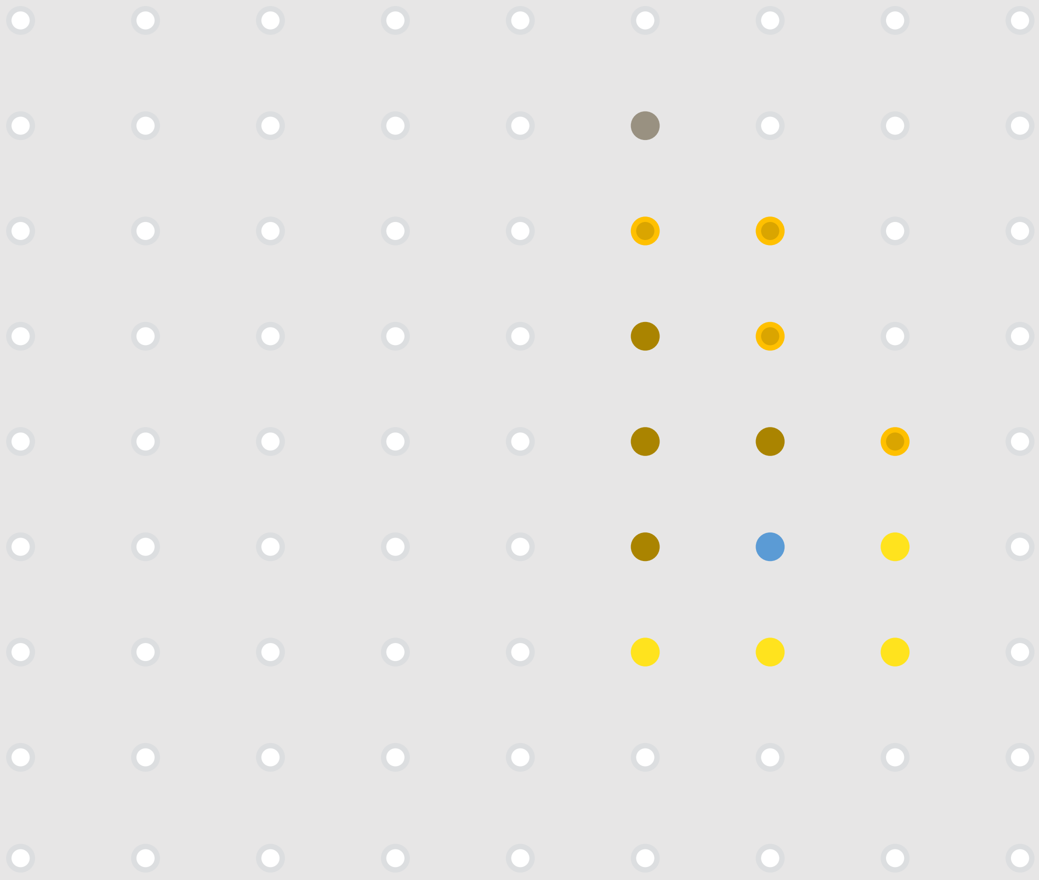
Step 7: Depth Test

Check depth and update depth if closer primitive found.
(can be disabled)

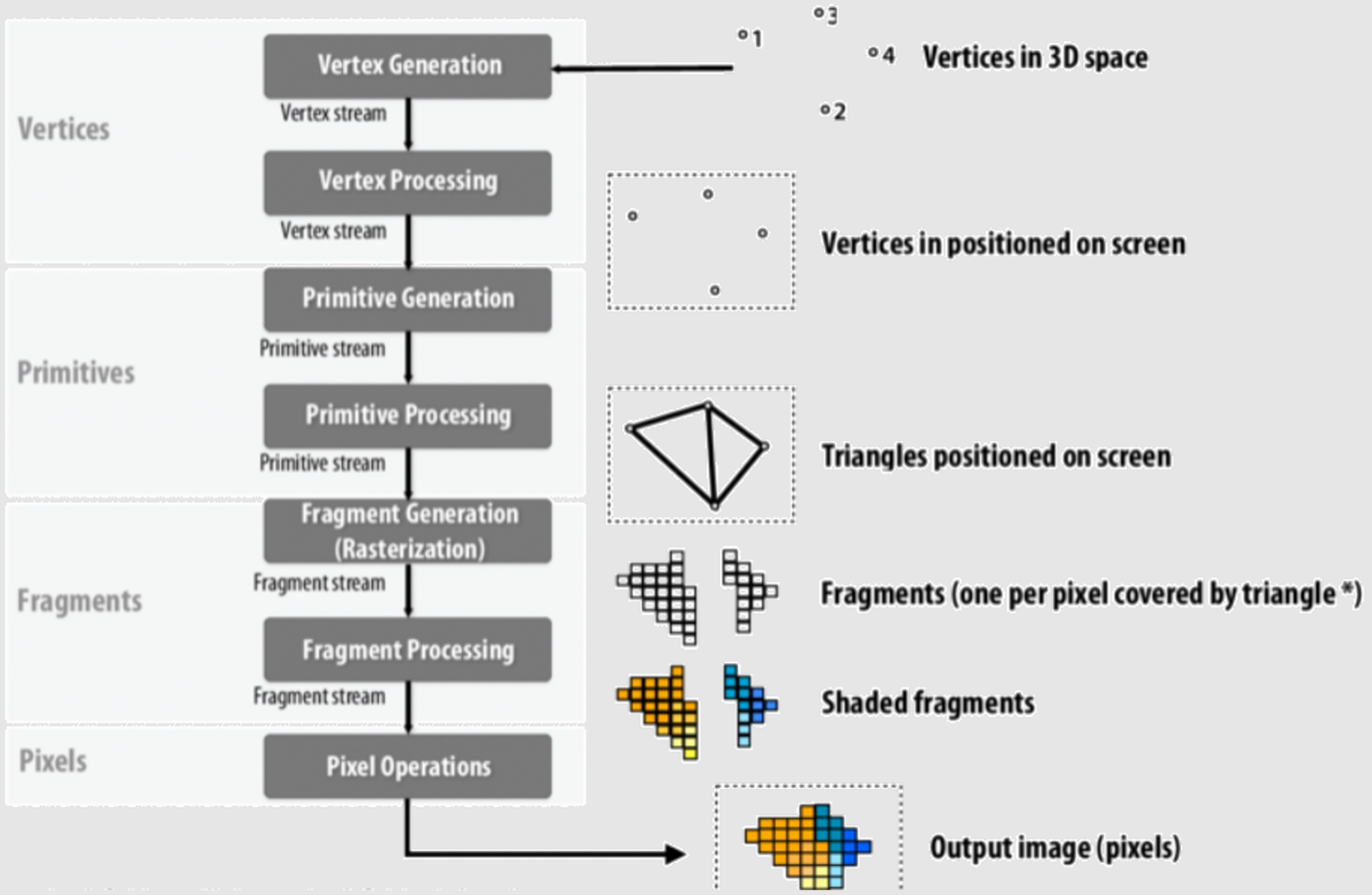


Step 8: Color Blending

Update color buffer with correct blending operation.



The "Real" Graphics Pipeline



Doesn't look much different than what we discussed…