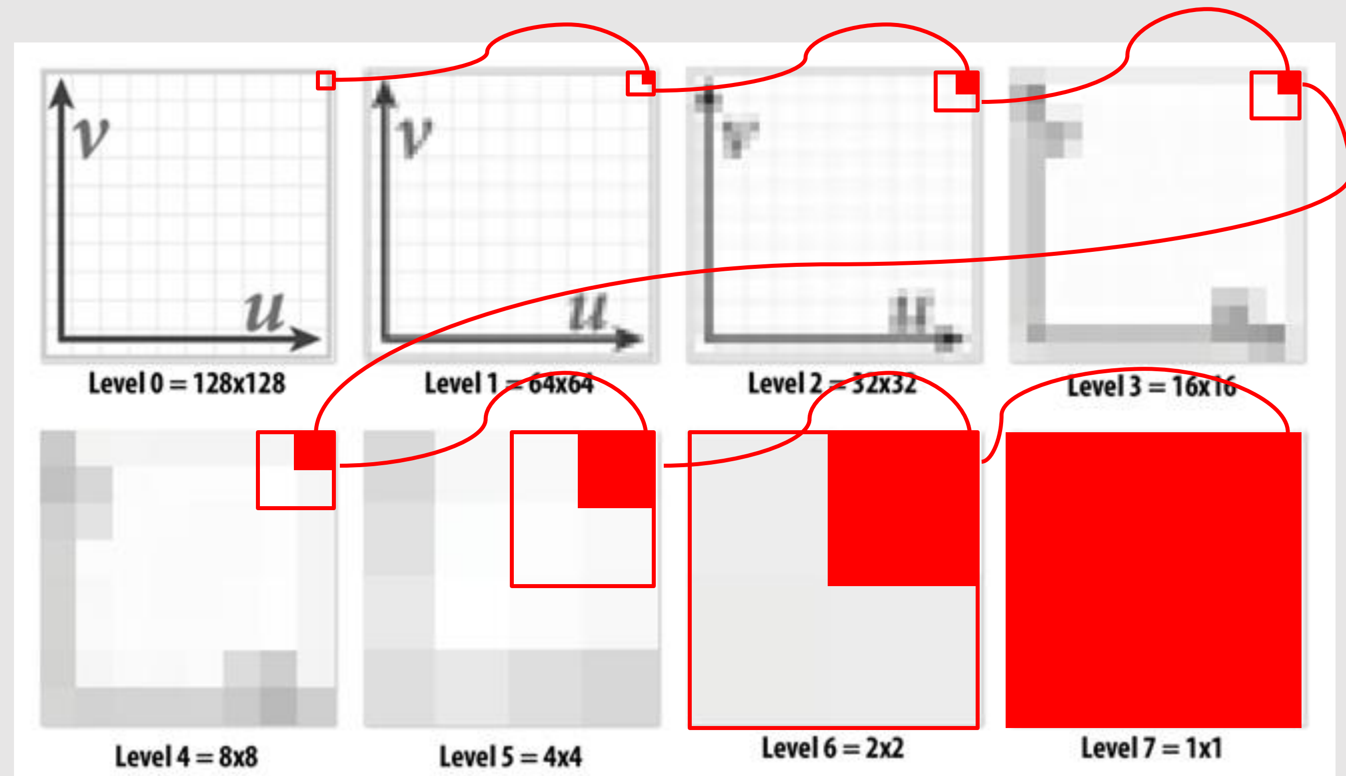
Wrapping up Part 1 of Computer Graphics

- Mip maps
- Depth Testing
- Alpha Blending
- Revisiting the Graphics pipeline
- 3D Rotations

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
  - $1 + \frac{1}{4} + \frac{1}{16} + \cdots = \sum \frac{1}{4}^j = \frac{1}{1-\frac{1}{4}} = \frac{4}{3}$



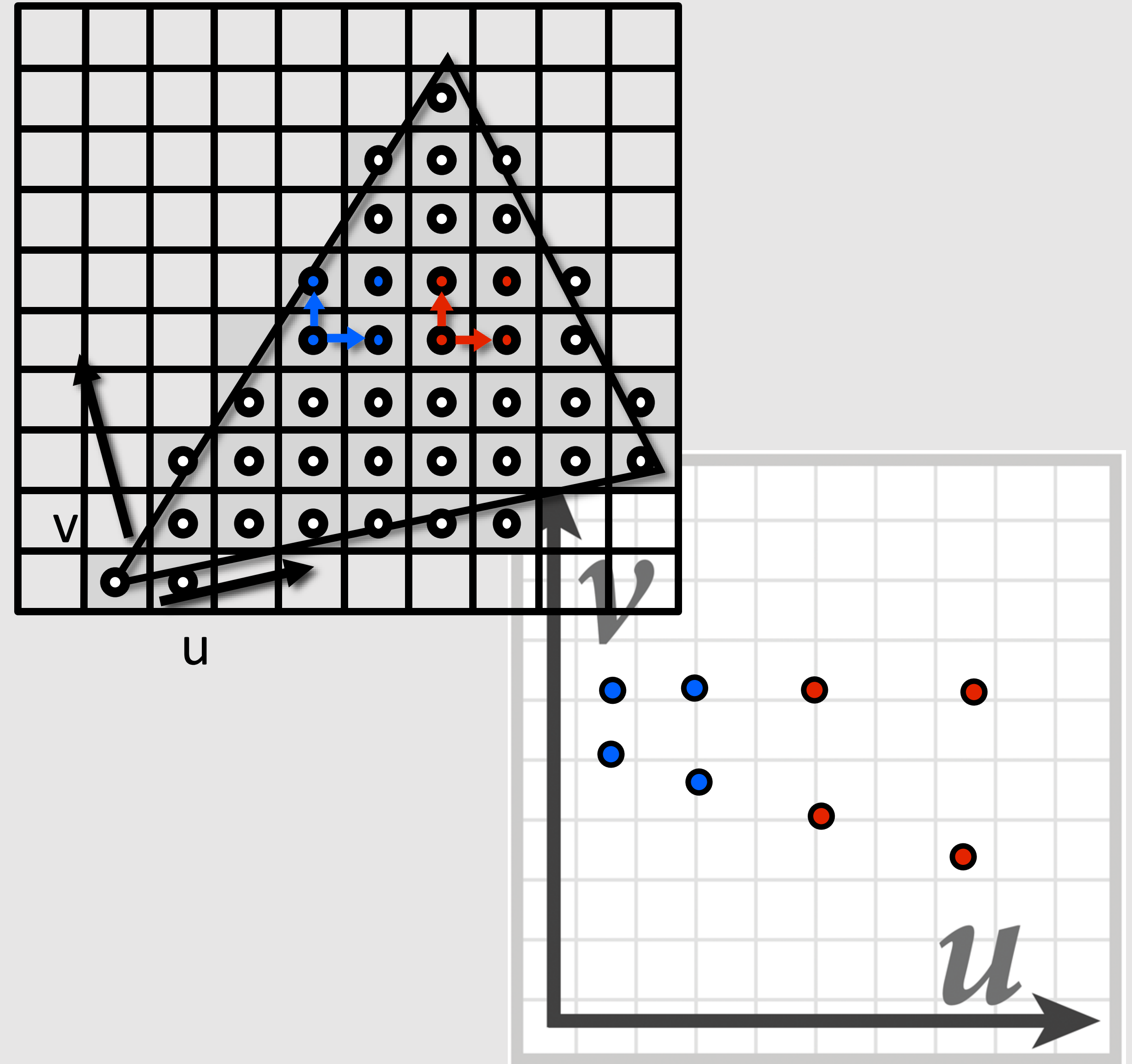
Which mip-map level do we use?

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:

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

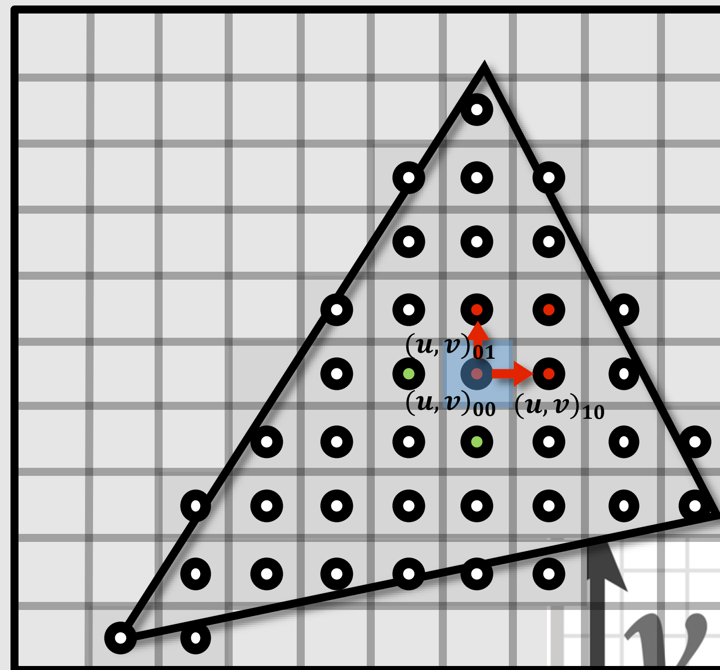
$$\frac{dv}{dx} = v_{10} - v_{00} \qquad \frac{dv}{dy} = v_{01} - v_{00}$$

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

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

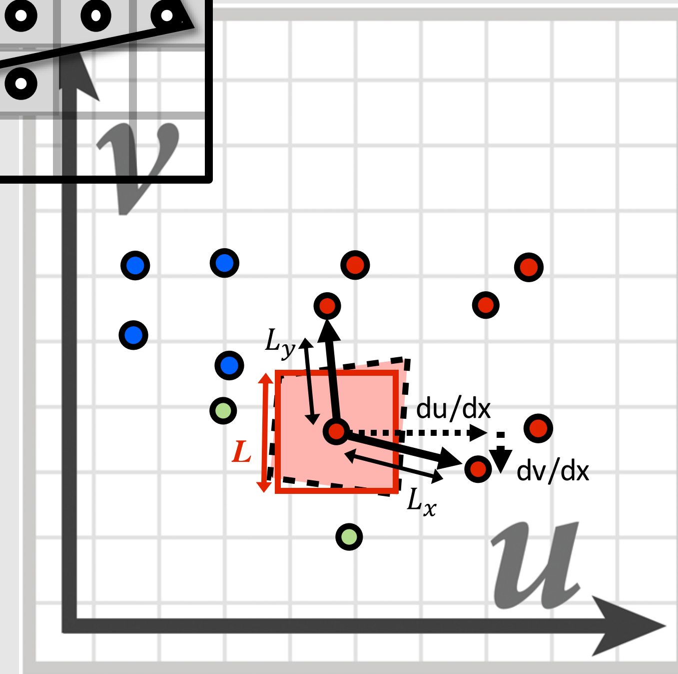
$$L = \sqrt{max(L_x^2, L_y^2)}$$

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

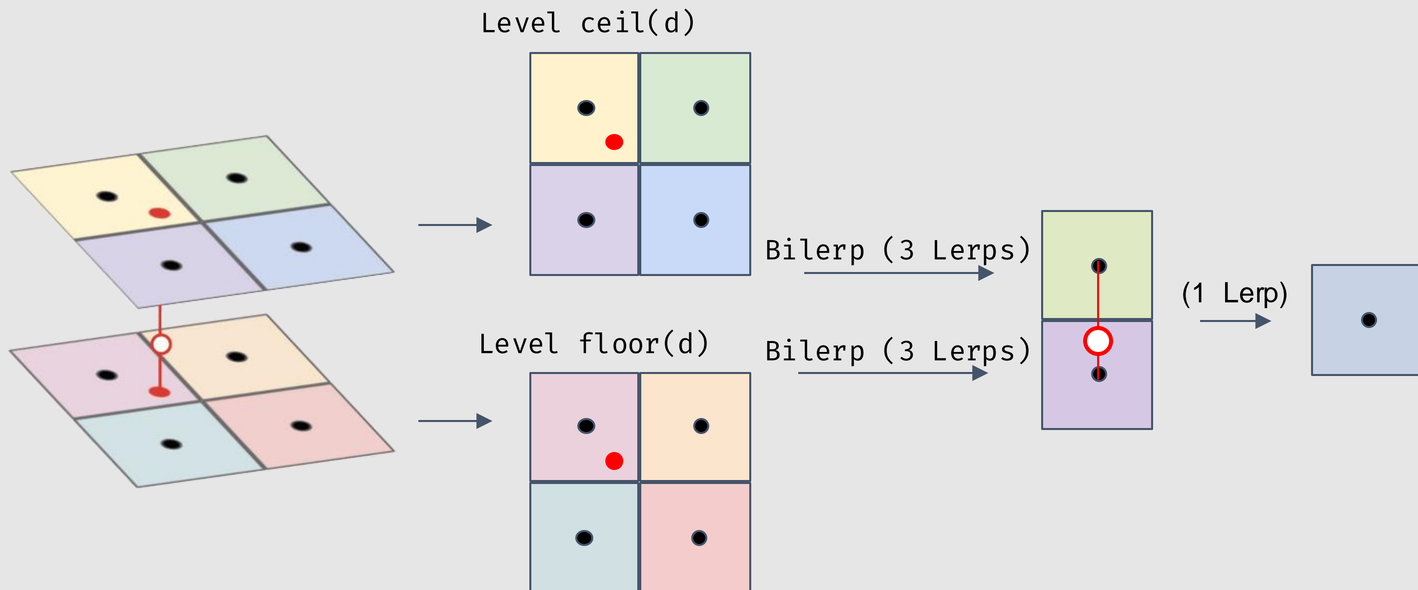
- **Requires:**
  - 8 memory lookup
  - 7 linear interpolations

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

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

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

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



Trilinear Interpolation Sampling

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

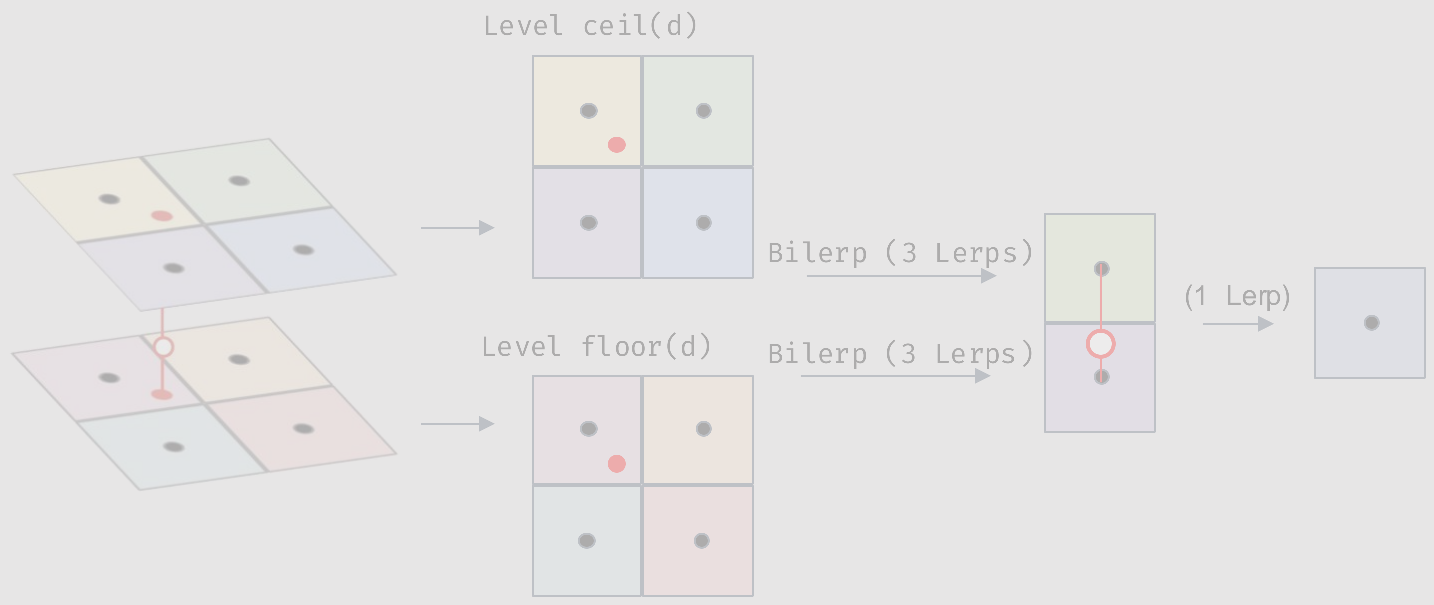
- **Requires:**
  - 8 memory lookup
  - 7 linear interpolations

why are we taking the max?

$$L_x{}^2 \leftarrow \frac{du}{dx}^2 + \frac{dv}{dx}^2$$
$$L_y{}^2 \leftarrow \frac{du}{dy}^2 + \frac{dv}{dy}^2$$
$$L \leftarrow \sqrt{\max(L_x{}^2, L_y{}^2)}$$
$$d \leftarrow \log_2 L$$

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

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

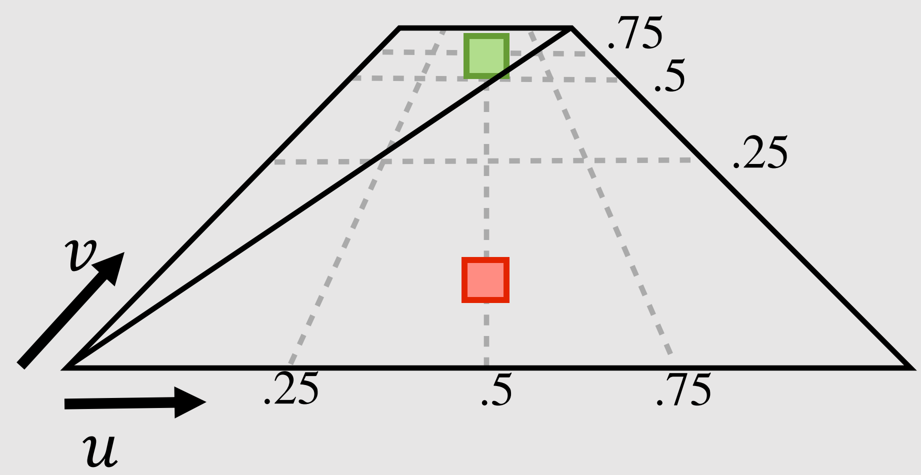


Trilinear Assumption

- 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

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

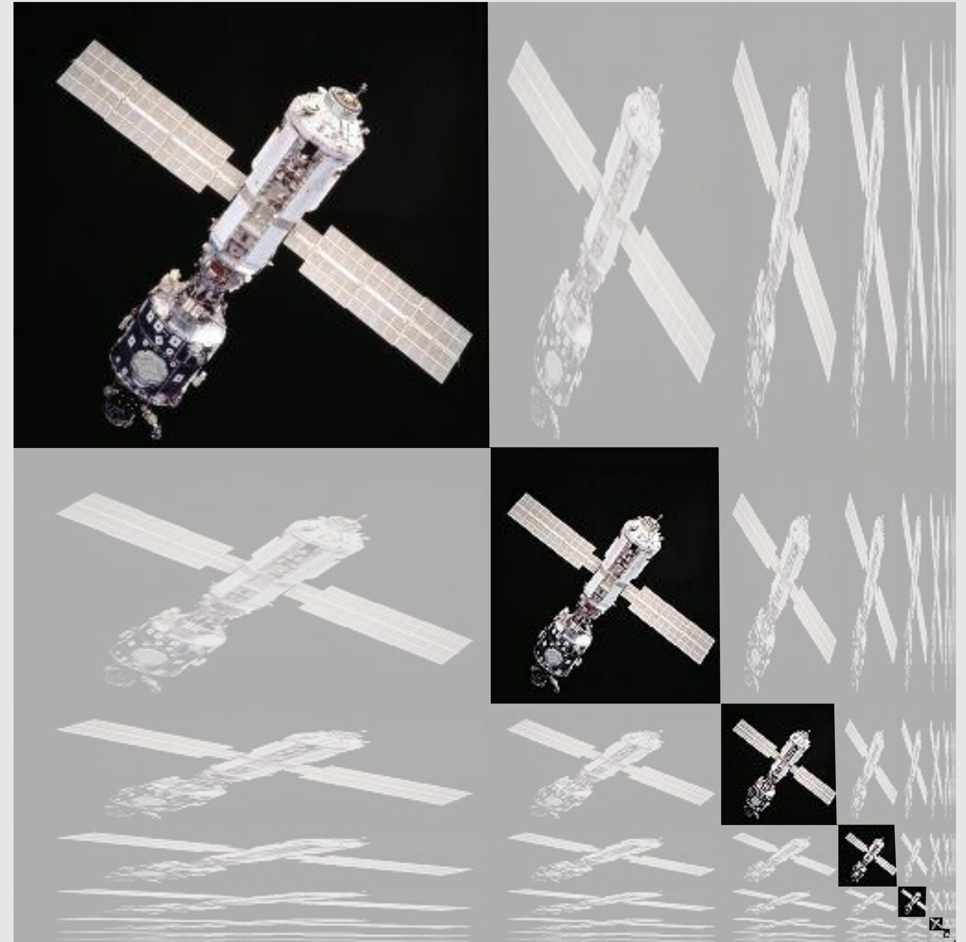
$$(d_x, d_y) = (\ log_2\sqrt{L_x^2}\ , log_2\sqrt{L_y^2}\ )$$

- 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…)

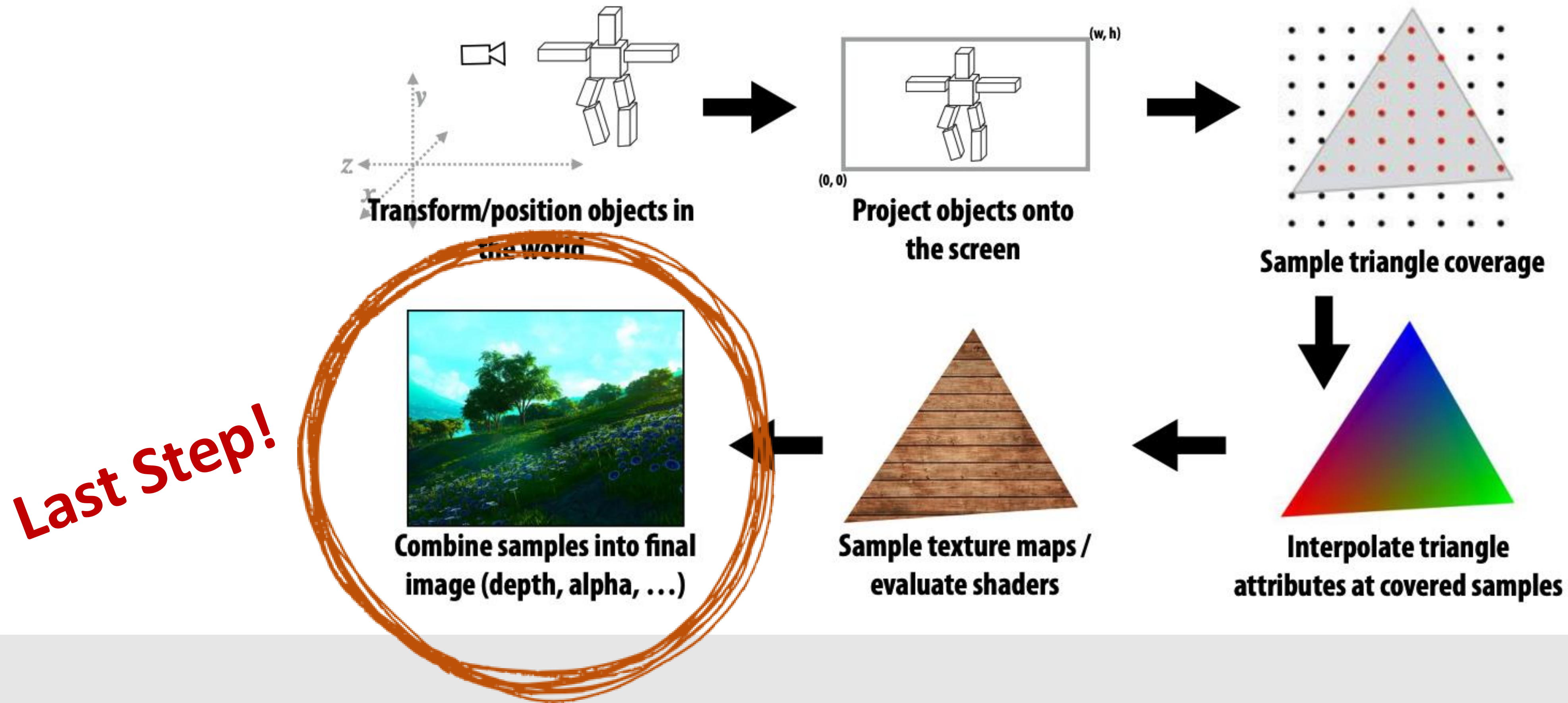
Lot of repetitive work every time we want to shade a pixel!

GPUs instead implement these instructions on fixed-function hardware.

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

- ~~Mip maps~~
- Depth Testing
- Alpha Blending
- Revisiting the Graphics pipeline
- 3D Rotations

The "Simpler" Graphics Pipeline

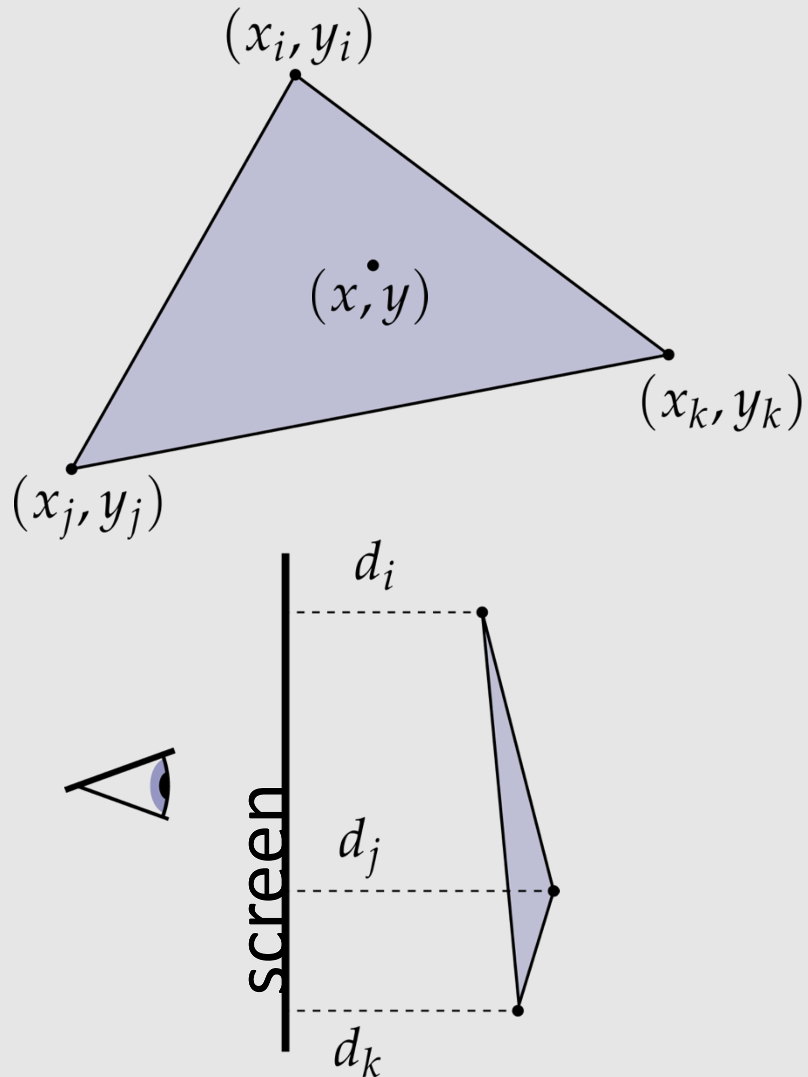


Depth Buffer (Z-buffer)

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

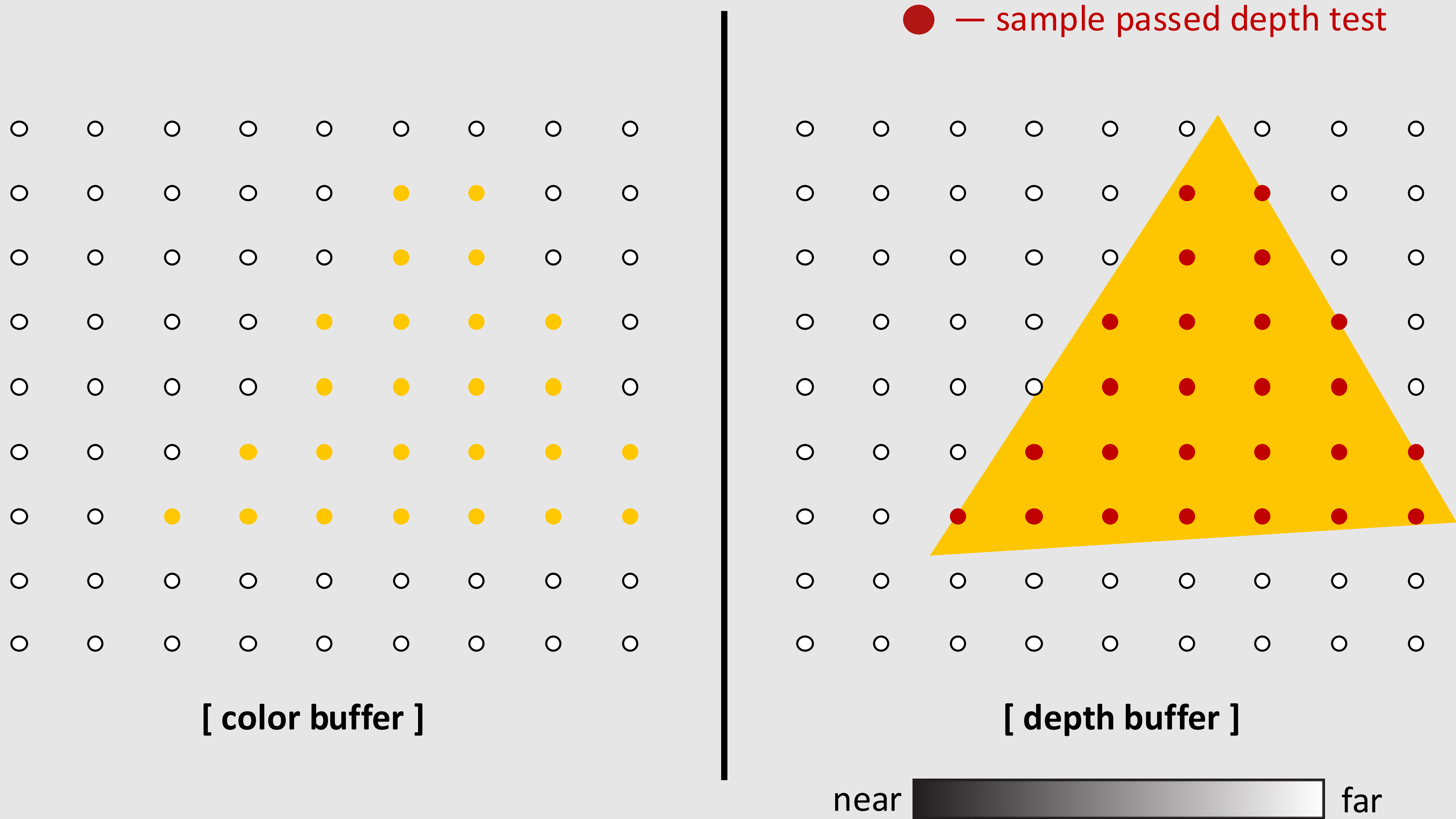
near far

Depth of a Triangle

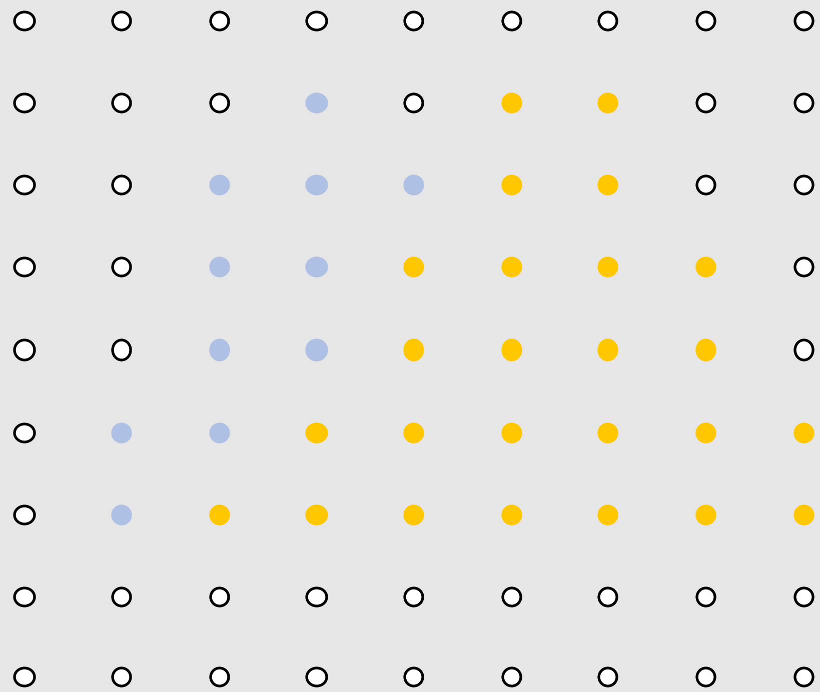


- A triangle is composed of 3 different 3D points, each with a depth value z
- To get the depth at any point (x, y) inside the triangle, interpolate depth at vertices with barycentric coordinates

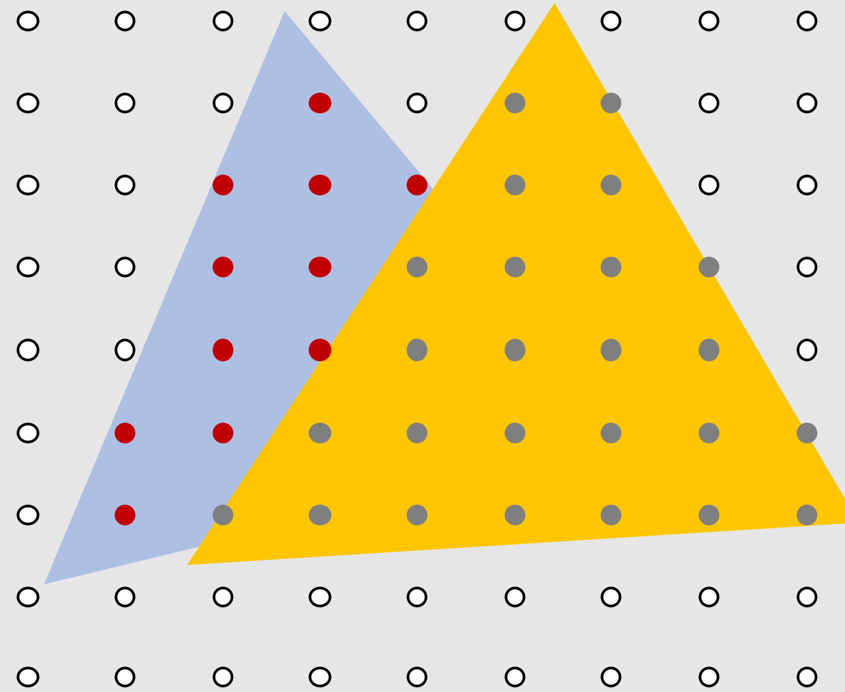
Depth Buffer (Z-buffer)



Depth Buffer (Z-buffer)

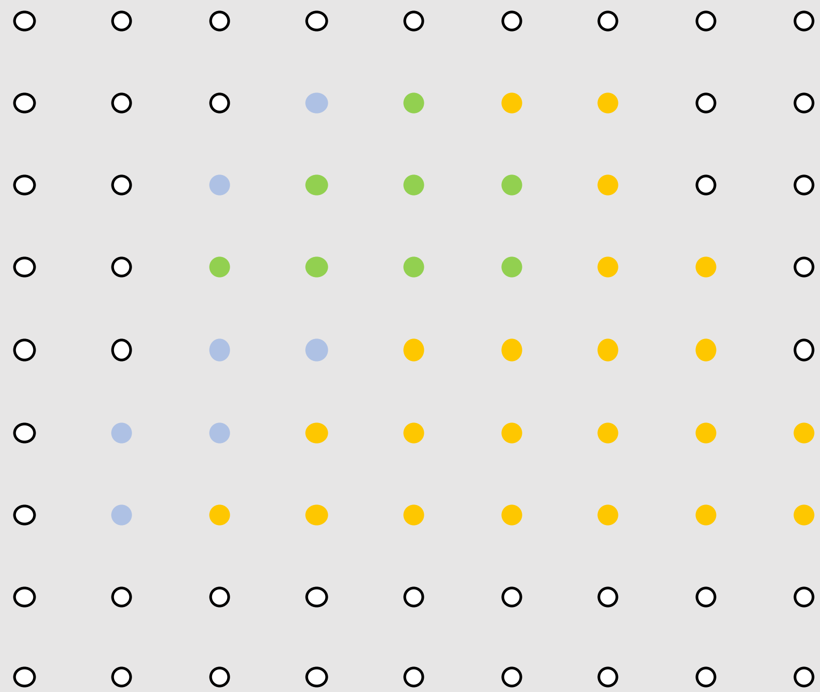


[color buffer]

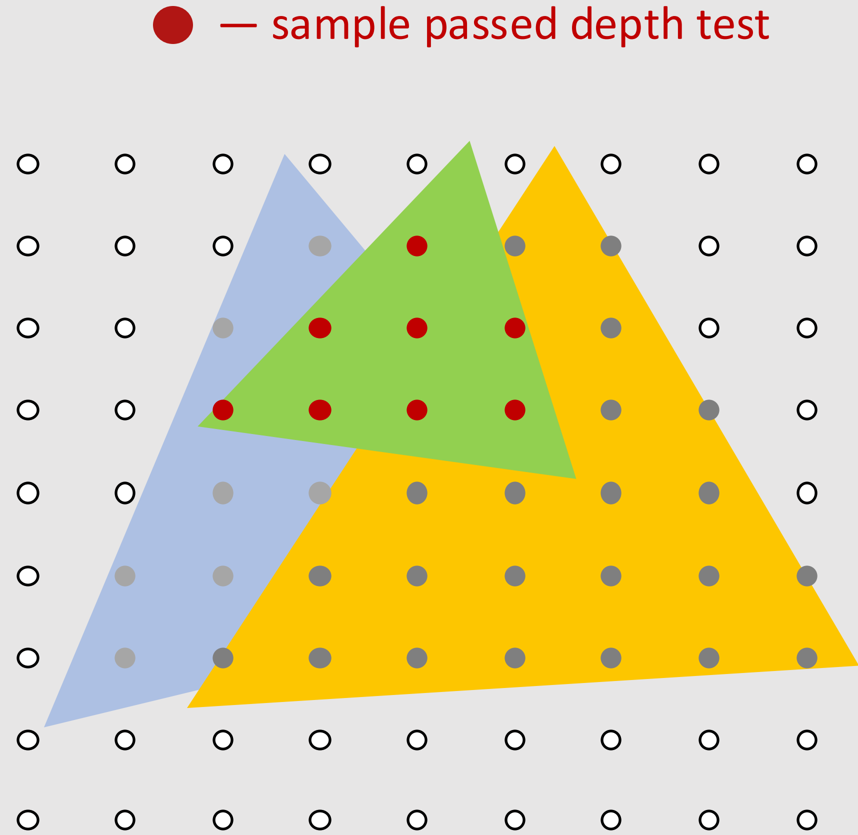


[depth buffer]

Depth Buffer (Z-buffer)



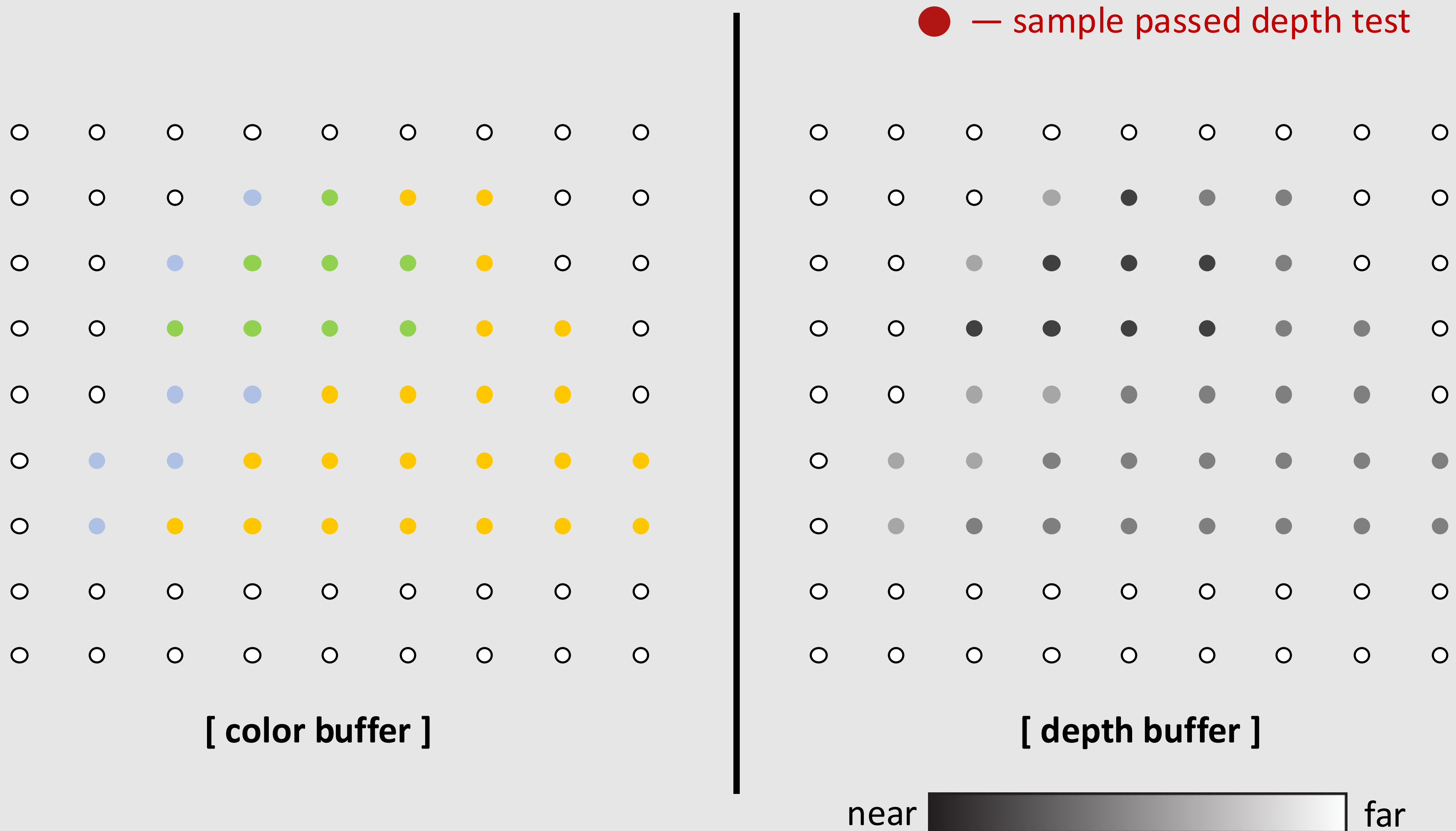
[color buffer]



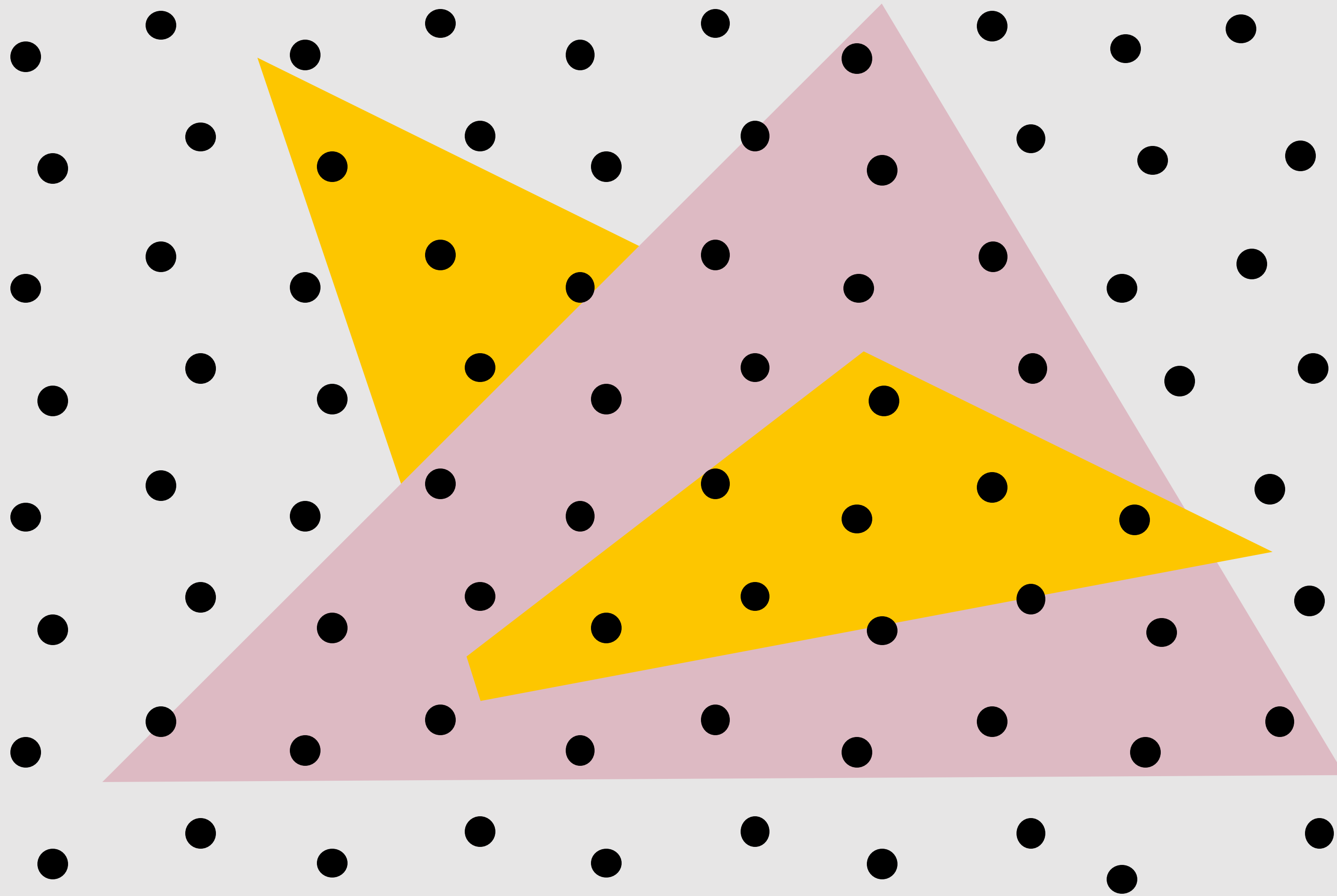
[depth buffer]

near far

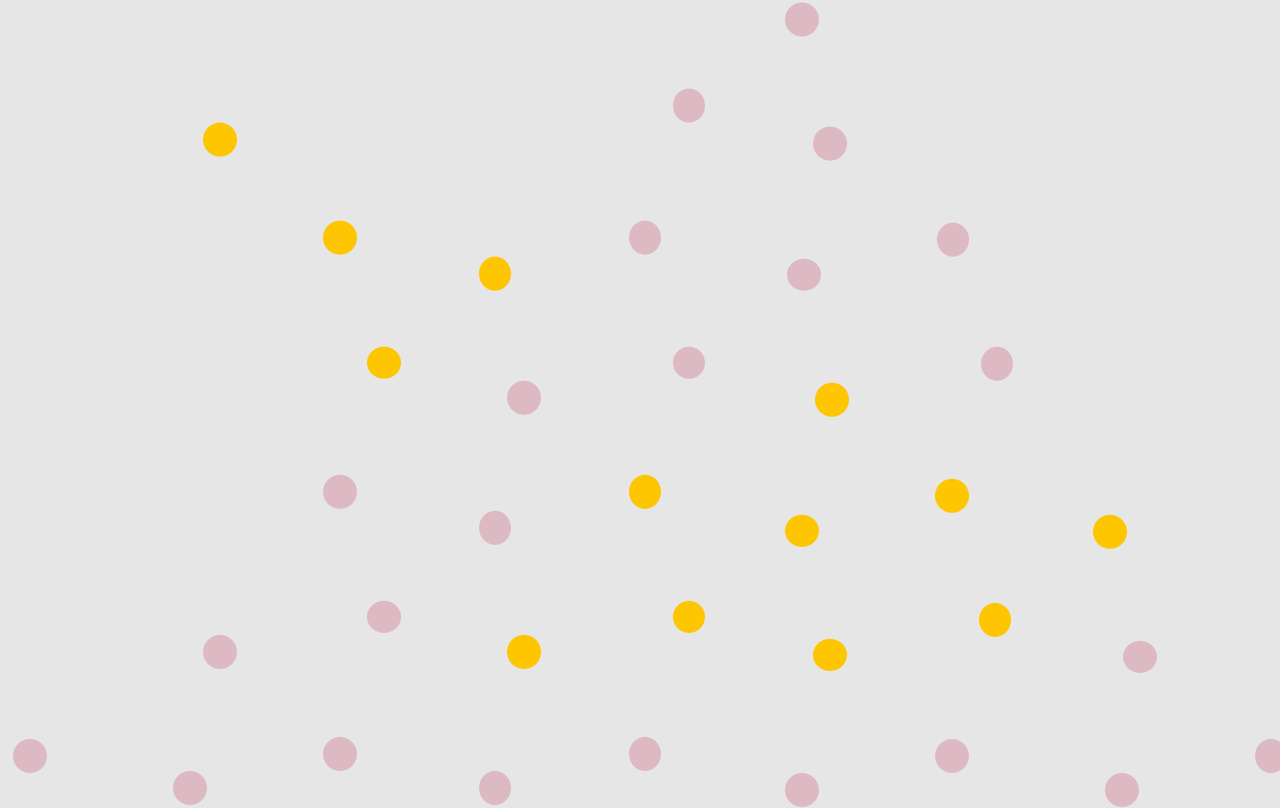
Depth Buffer (Z-buffer)



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?

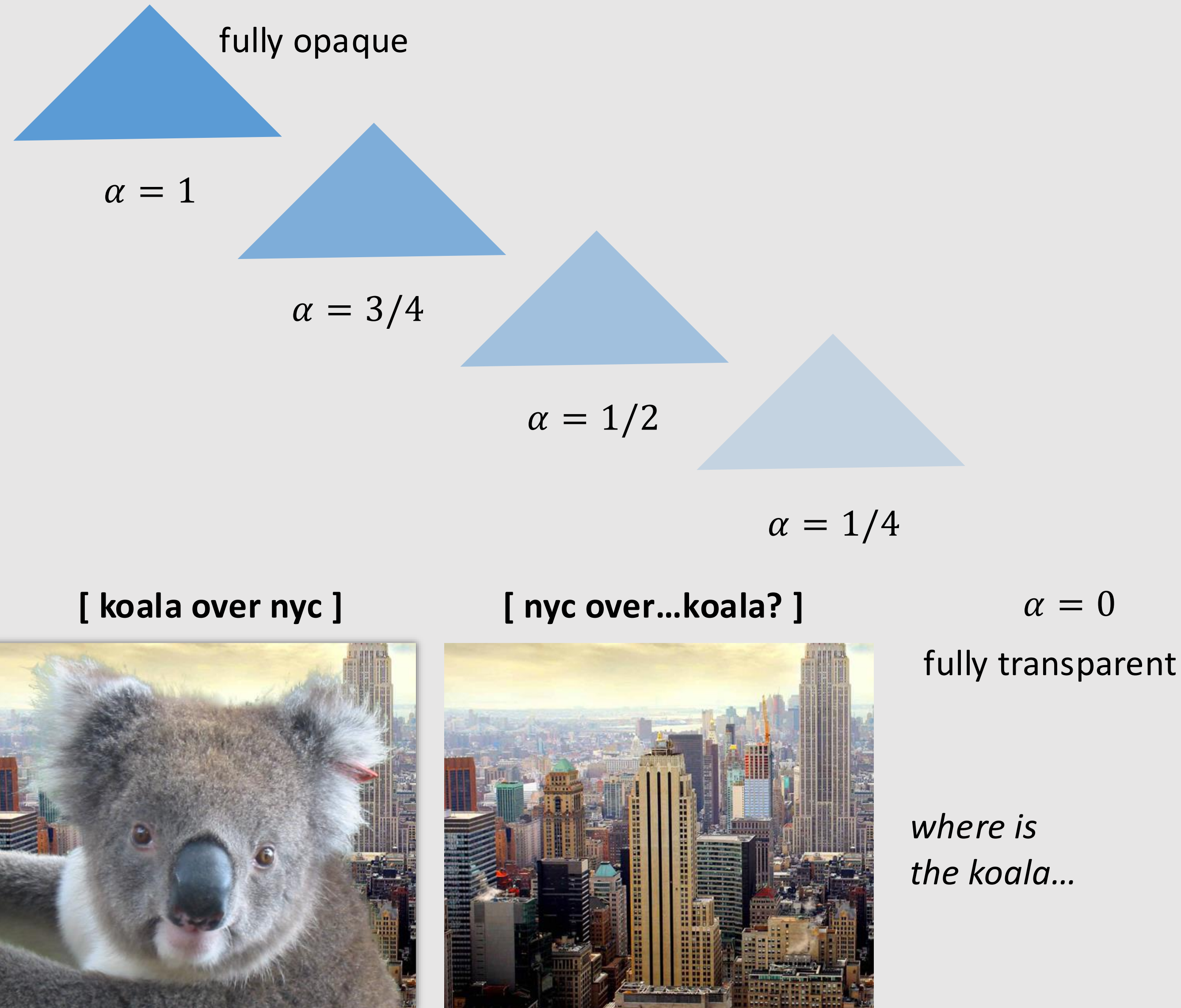
Deferred shading (advanced algorithm) fixes this issue.

- ~~Mip maps~~
- ~~Depth Testing~~
- Alpha Blending
- Revisiting the Graphics pipeline
- 3D Rotations

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)

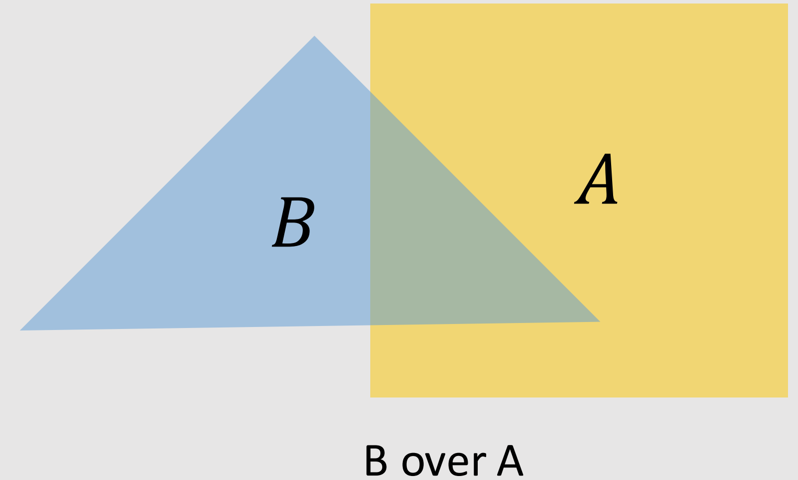
- Compositing A over B != B over A
  - Consider the extreme case of two opaque objects…



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

- <u>Composite RGB:</u>

what B lets through

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

appearance of semi-transparent B

appearance of semi-transparent A

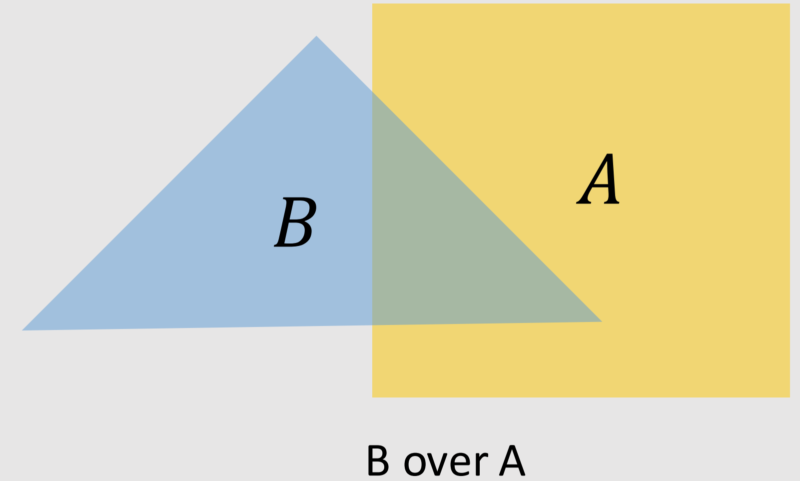
- <u>Composite Alpha:</u>

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

Two different equations is inefficient!!

Premultiplied Alpha

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



B over A

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

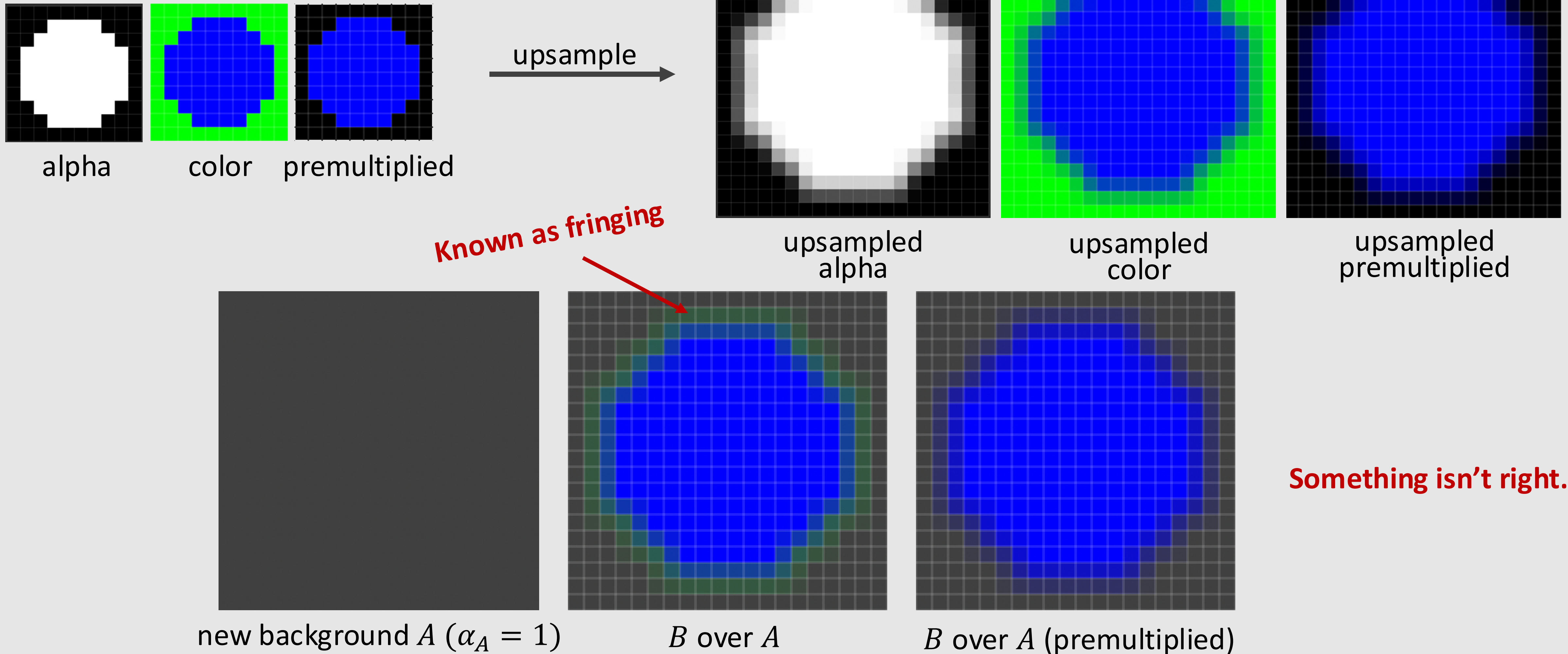
- <u>Composite RGBA:</u>

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

- <u>Un-Premultiply for Final Color:</u>

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

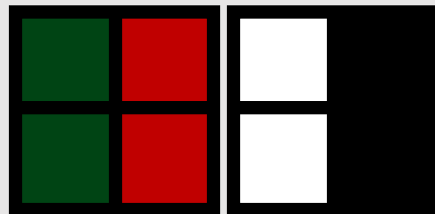
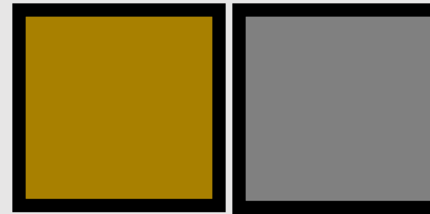
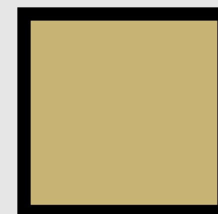
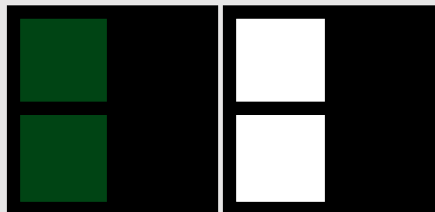
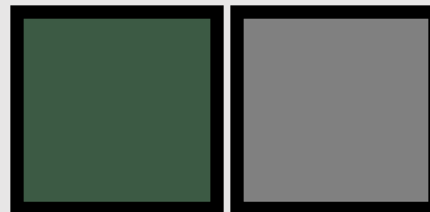
Why Premultiplied Matters [Upsample]



Something isn't right…

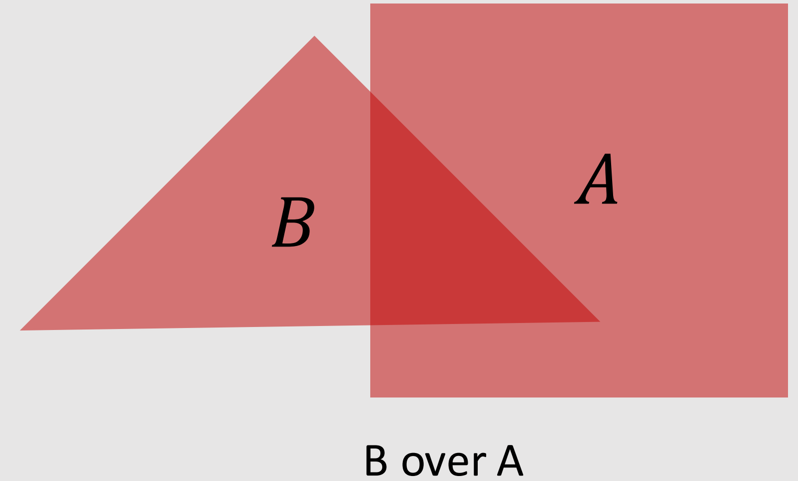
Why Premultiplied Matters [Downsample]

[RGB] **[A]**

	original			downsampled			
	color	alpha		color	alpha		composite
regular							
premultiplied							

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

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

- <u>Non-Premultiplied:</u>

color

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

alpha

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

- <u>Premultiplied:</u>

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

↓ 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
}
```

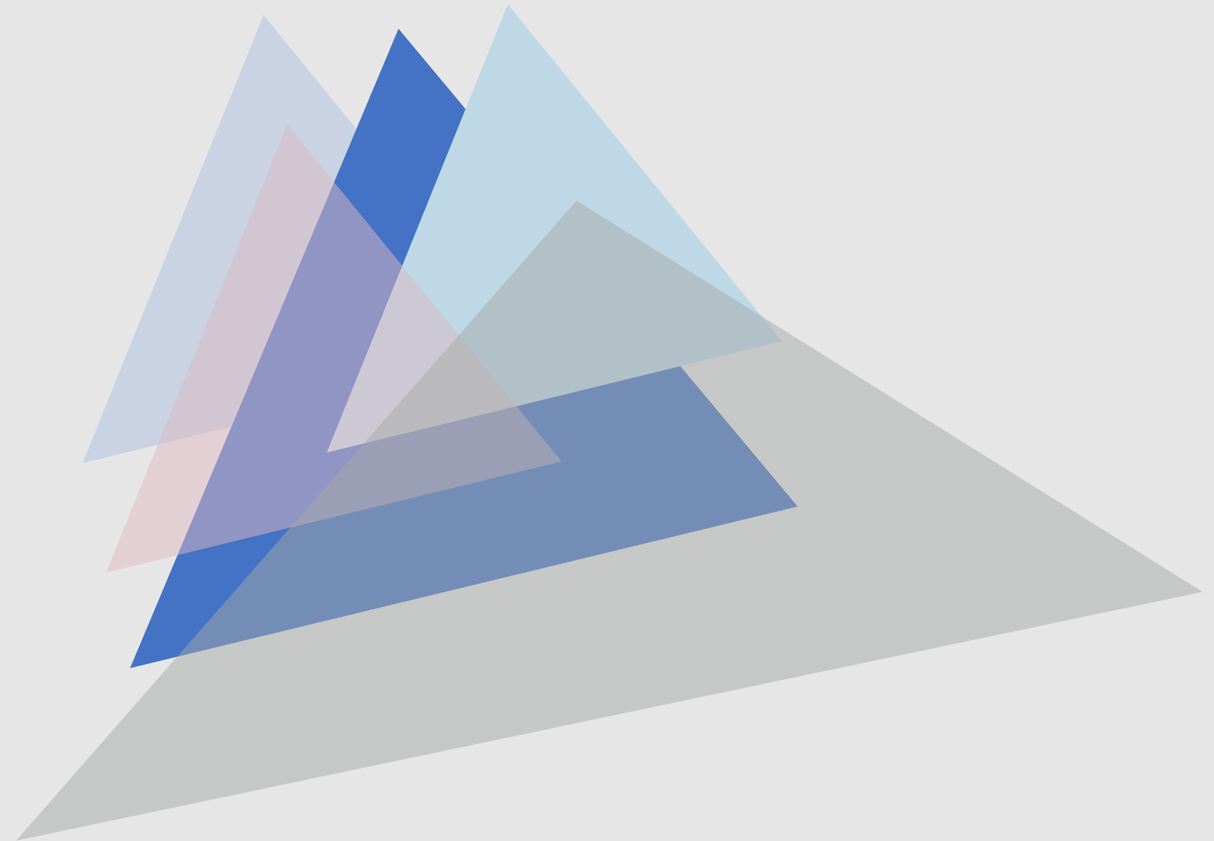
Should we still be doing depth writes for alpha primitives?

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

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

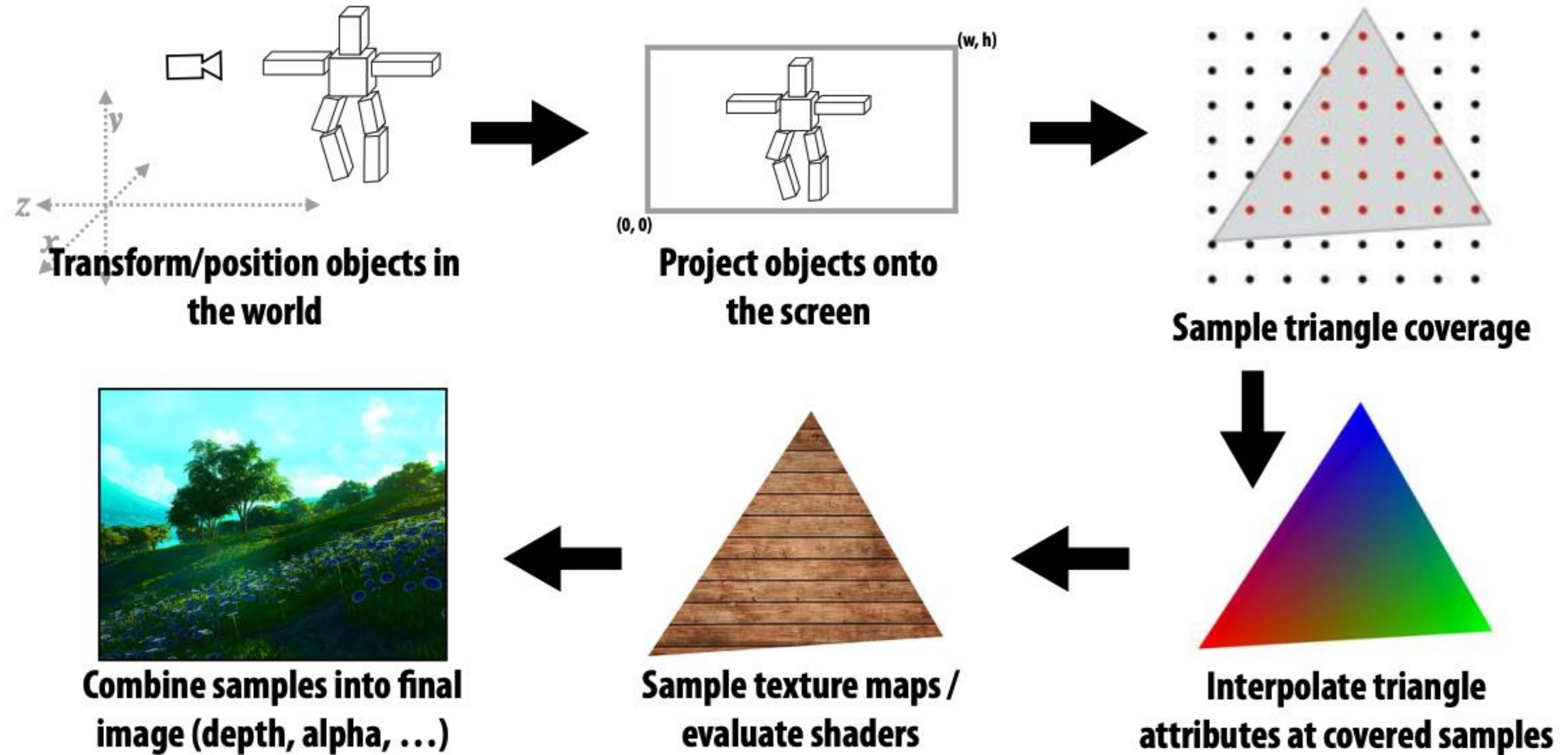
Blend Render Order

- 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



- ~~Mip maps~~
- ~~Depth Testing~~
- ~~Alpha Blending~~
- Revisiting the Graphics pipeline
- 3D Rotations

The “Simpler” Graphics Pipeline



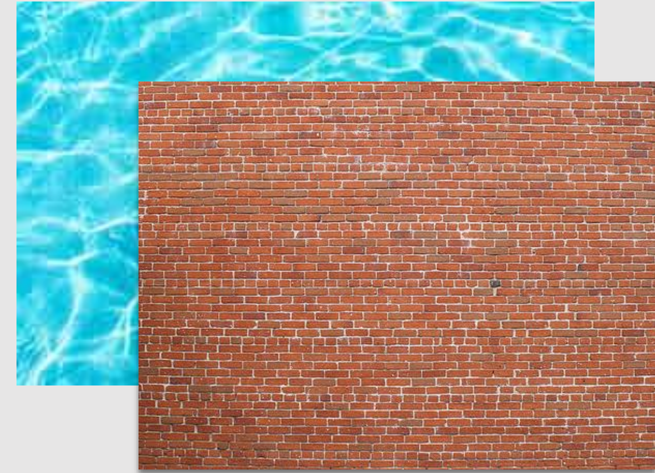
Now Let’s Put It All Together!

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 $P \in \mathbb{R}^{4\times4}$

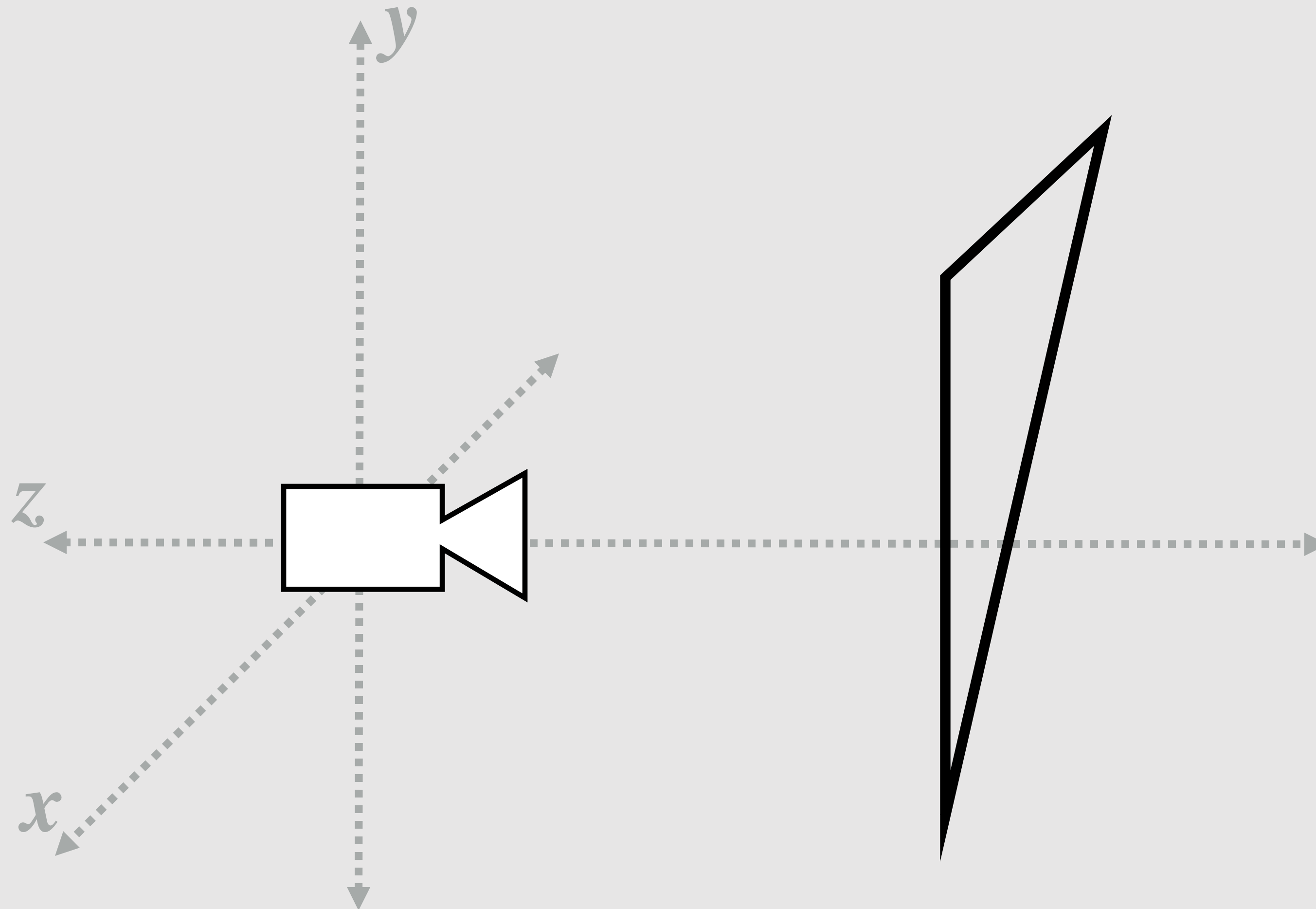
Output image (W, H)

[camera properties]

[machine]

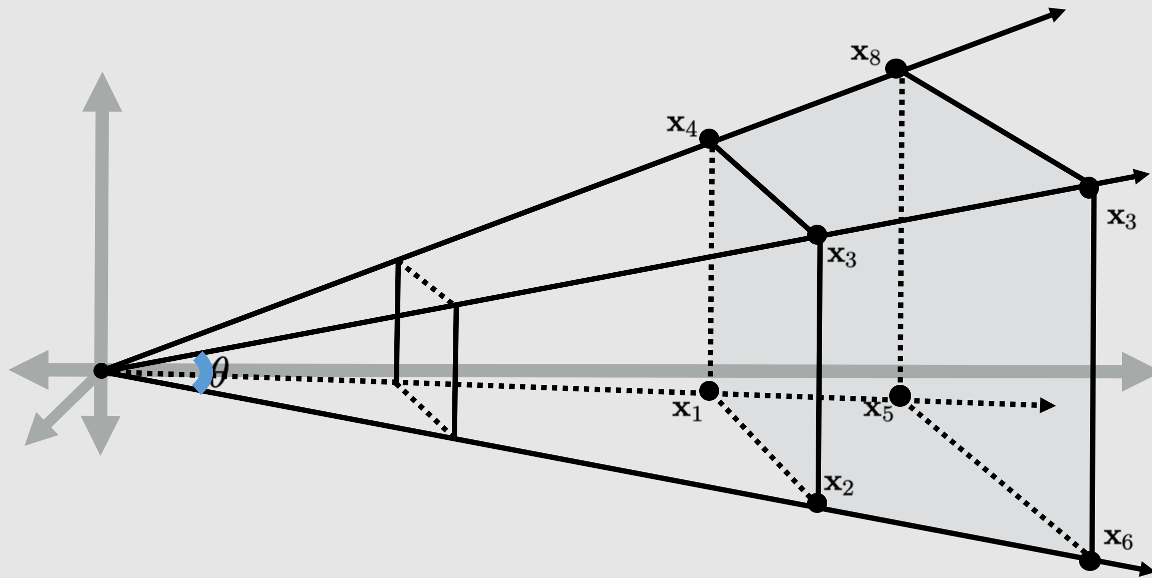
Step 1: Transform

Transform triangle vertices into camera space

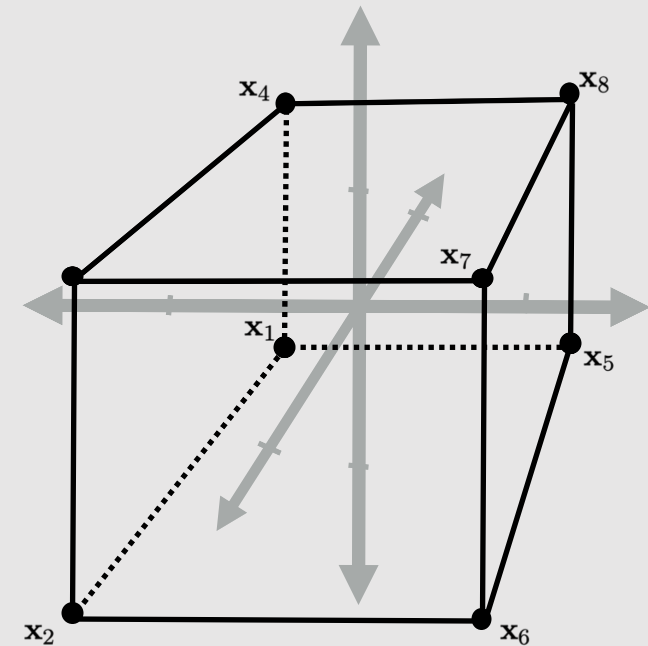


Step 2: Perspective Projection

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



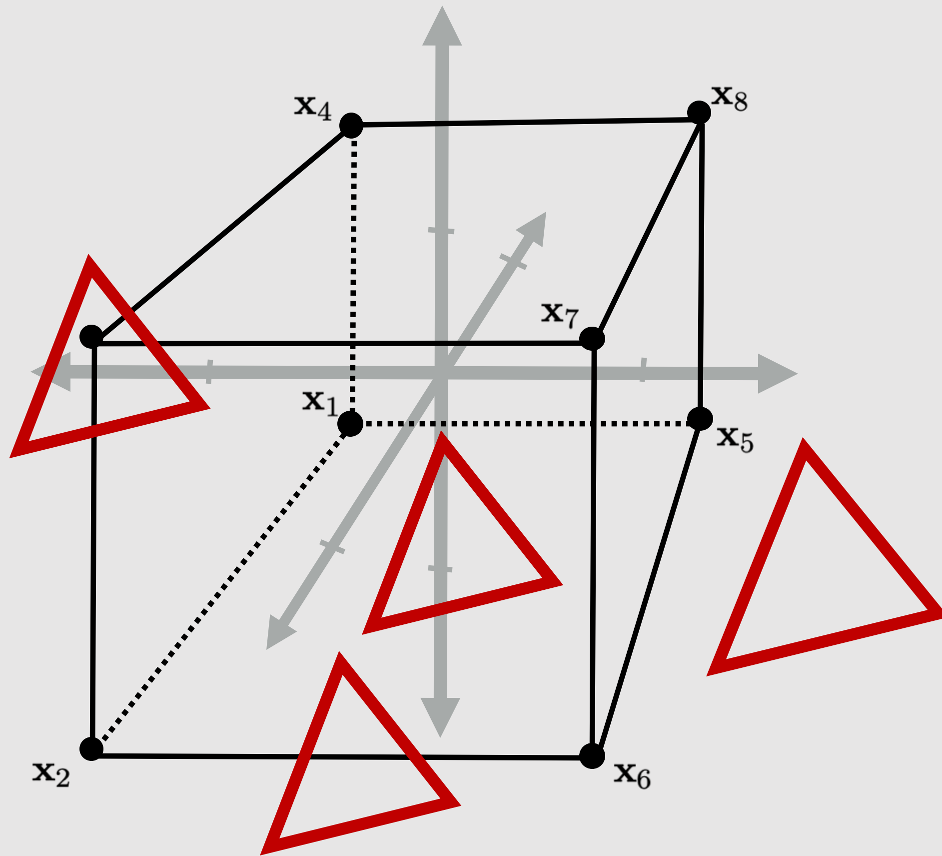
[3D camera space position]



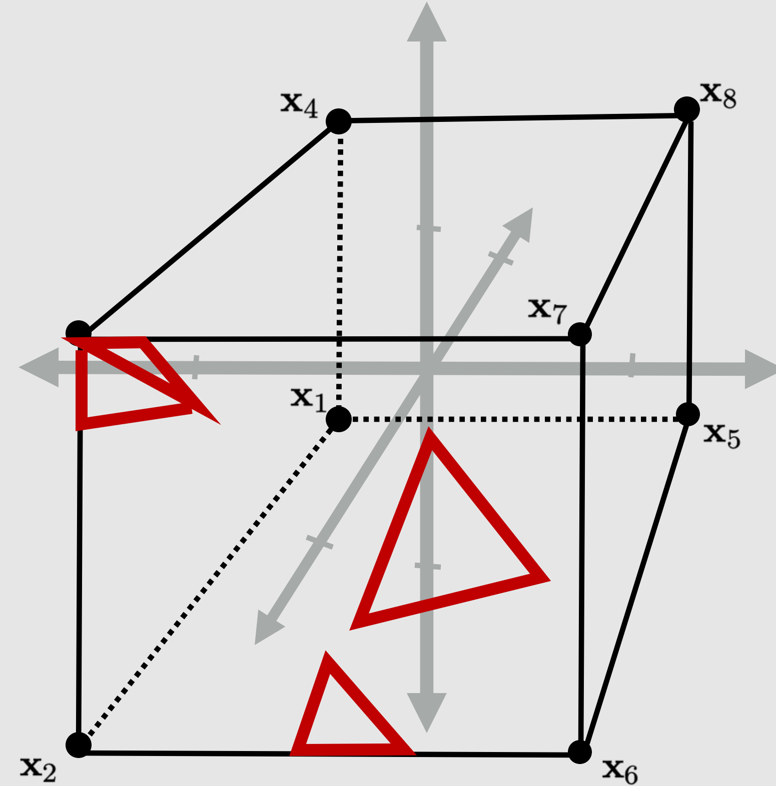
[normalized space position]

Step 3: Clipping

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



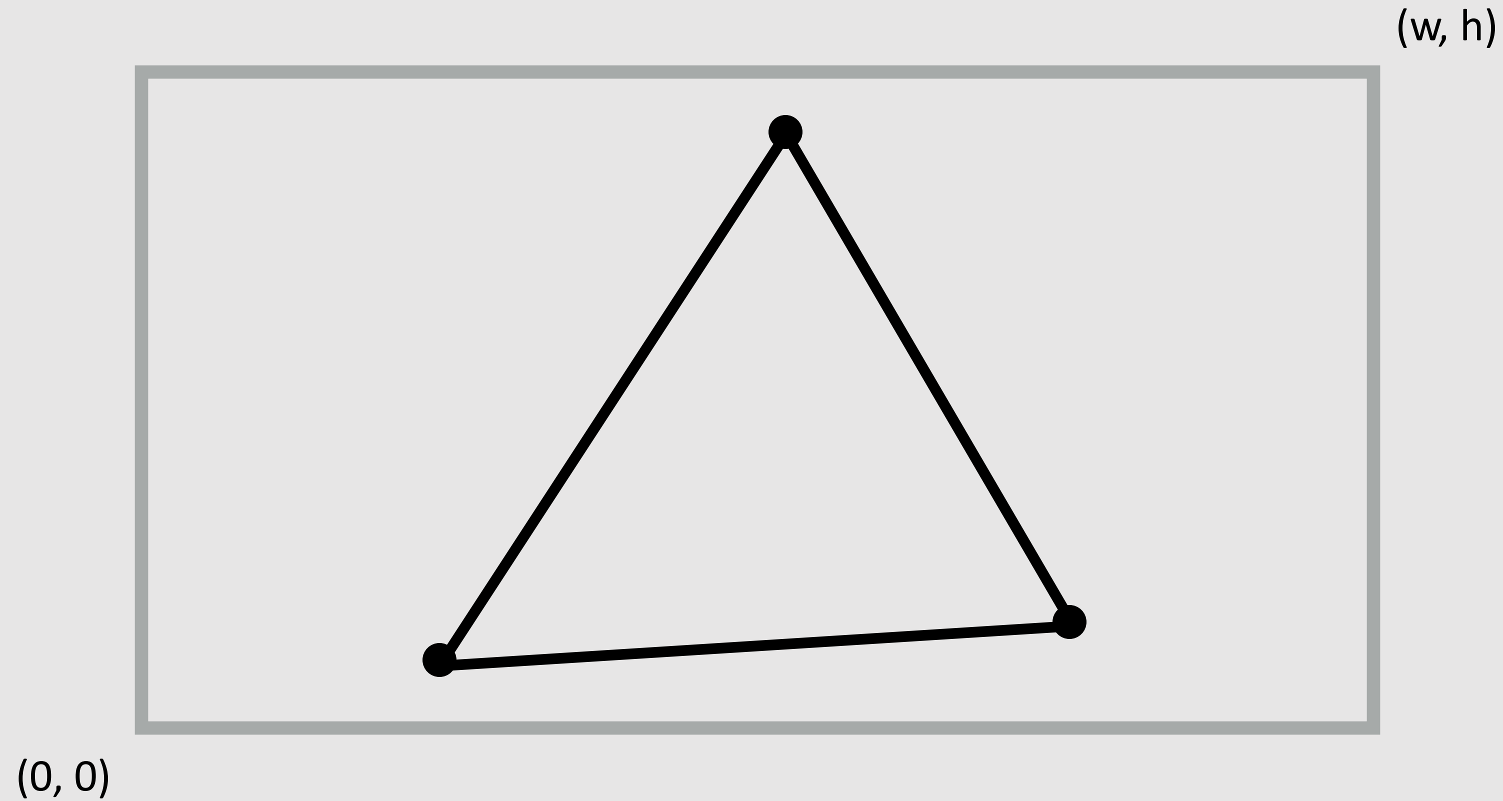
[pre-clipping]



[post-clipping]

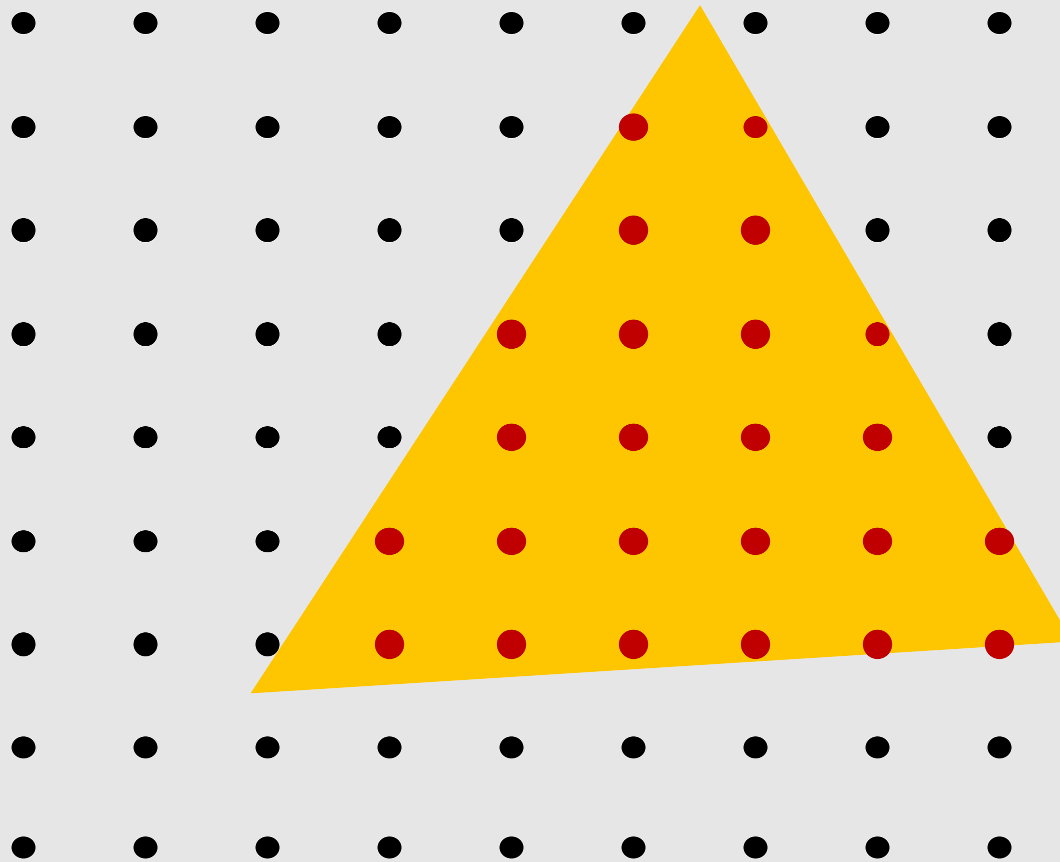
Step 4: Transform To Screen Coordinates

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



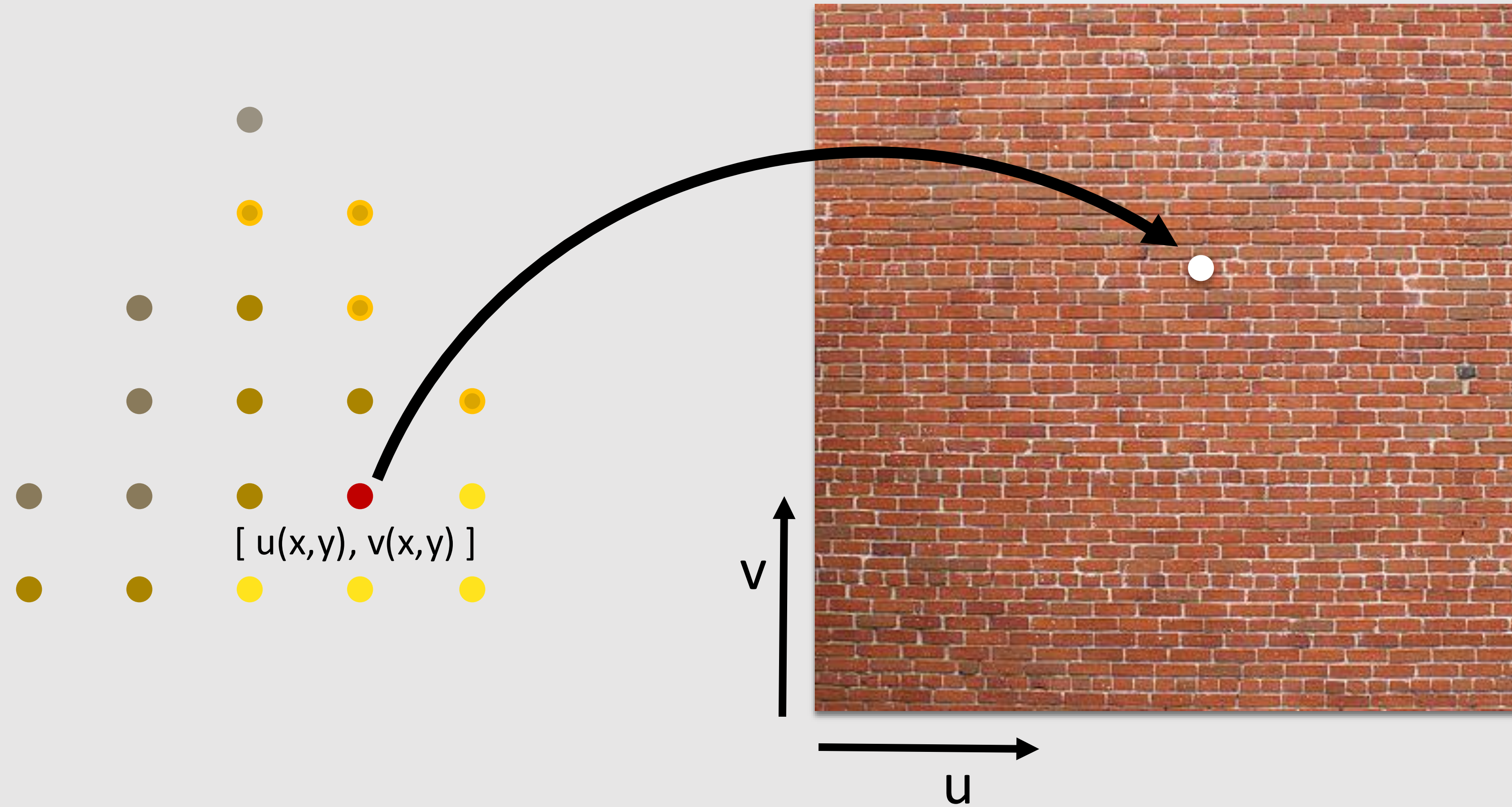
Step 5: Sample Coverage

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



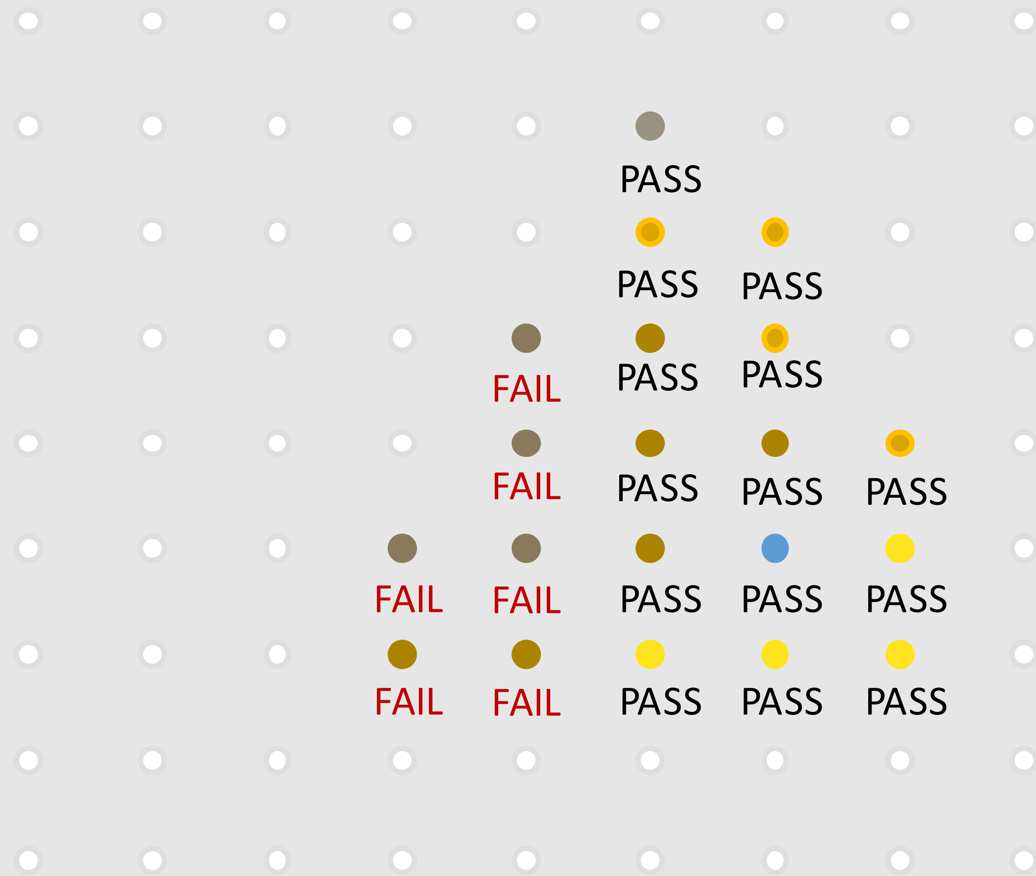
Step 6: Compute Color

Texture lookups, color interpolation, etc.



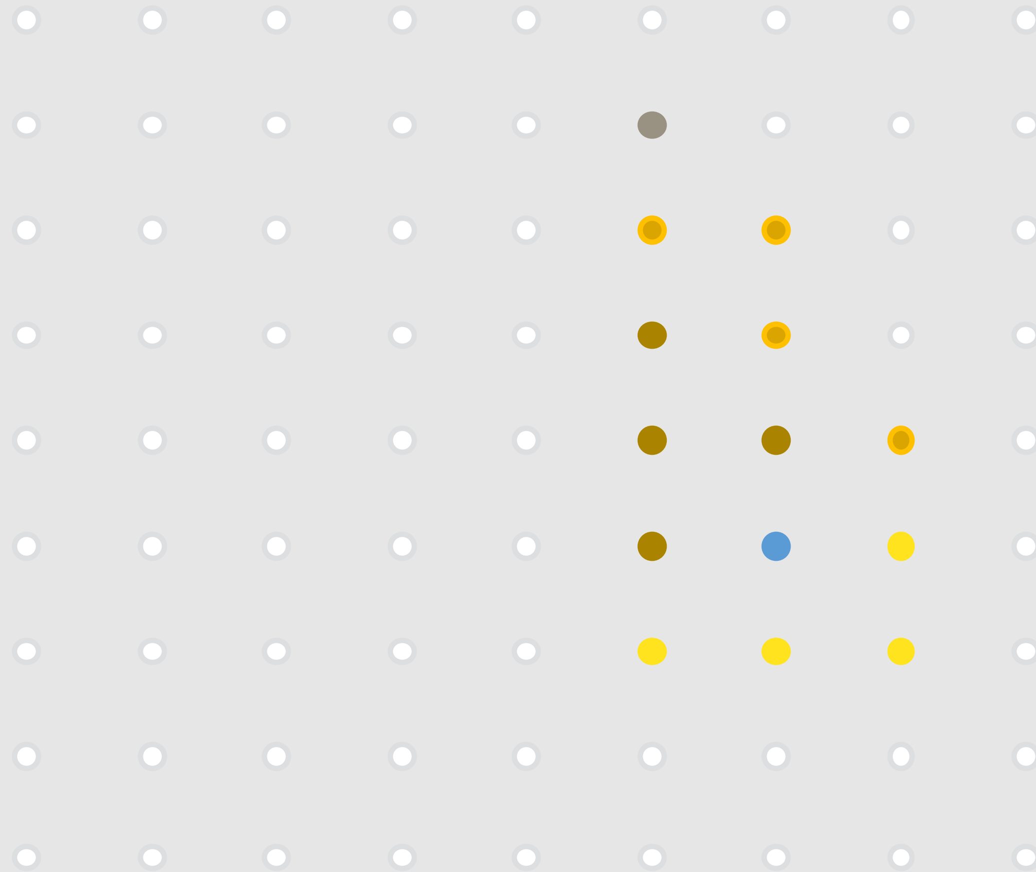
Step 7: Depth Test

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



Step 8: Color Blending

Update color buffer with correct blending operation.

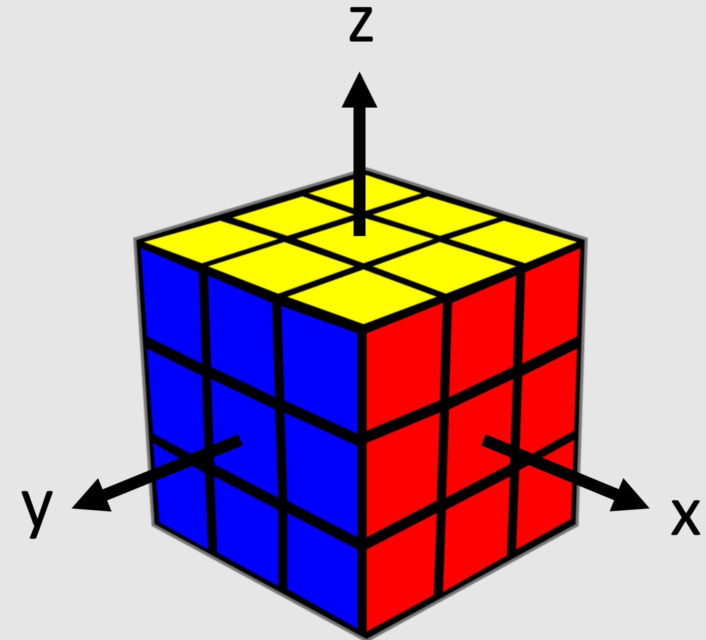


- ~~Mip maps~~
- ~~Depth Testing~~
- ~~Alpha Blending~~
- ~~Revisiting the Graphics pipeline~~
- 3D Rotations

3D Rotations

- Rotating in 2D is the same as rotating around the z-axis
- **Idea:** independently rotate around each (x,y,z)-axis for 3D rotations

- **Problem:** order of rotation matters!
  - Rotate a Rubik's cube 90deg around the y-axis and 90deg around the z-axis
  - Rotate a Rubik's cube 90deg around the z-axis and 90deg around the y-axis
    - They will not be the same!
  - Order of rotation must be specified



3D Rotations in Matrix Form

Idea: independently rotate around each (x,y,z)-axis for 3D rotations:

$$R_x = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta_x & -\sin\theta_x \\ 0 & \sin\theta_x & \cos\theta_x \end{bmatrix} \quad R_y = \begin{bmatrix} \cos\theta_y & 0 & \sin\theta_y \\ 0 & 1 & 0 \\ -\sin\theta_y & 0 & \cos\theta_y \end{bmatrix} \quad R_z = \begin{bmatrix} \cos\theta_z & -\sin\theta_z & 0 \\ \sin\theta_z & \cos\theta_z & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

Combining the matrices:

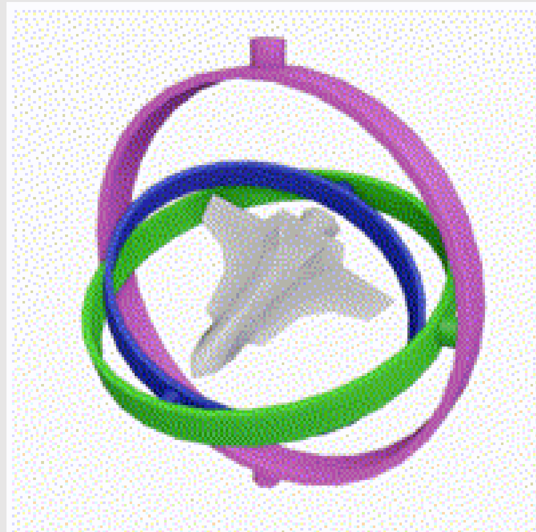
$$R_x R_y R_z = \begin{bmatrix} \cos\theta_y\cos\theta_z & -\cos\theta_y\sin\theta_z & \sin\theta_y \\ \cos\theta_z\sin\theta_x\sin\theta_y + \cos\theta_x\sin\theta_z & \cos\theta_x\cos\theta_z - \sin\theta_x\sin\theta_y\sin\theta_z & -\cos\theta_y\sin\theta_x \\ -\cos\theta_x\cos\theta_z\sin\theta_y + \sin\theta_x\sin\theta_z & \cos\theta_z\sin\theta_x + \cos\theta_x\sin\theta_y\sin\theta_z & \cos\theta_x\cos\theta_y \end{bmatrix}$$

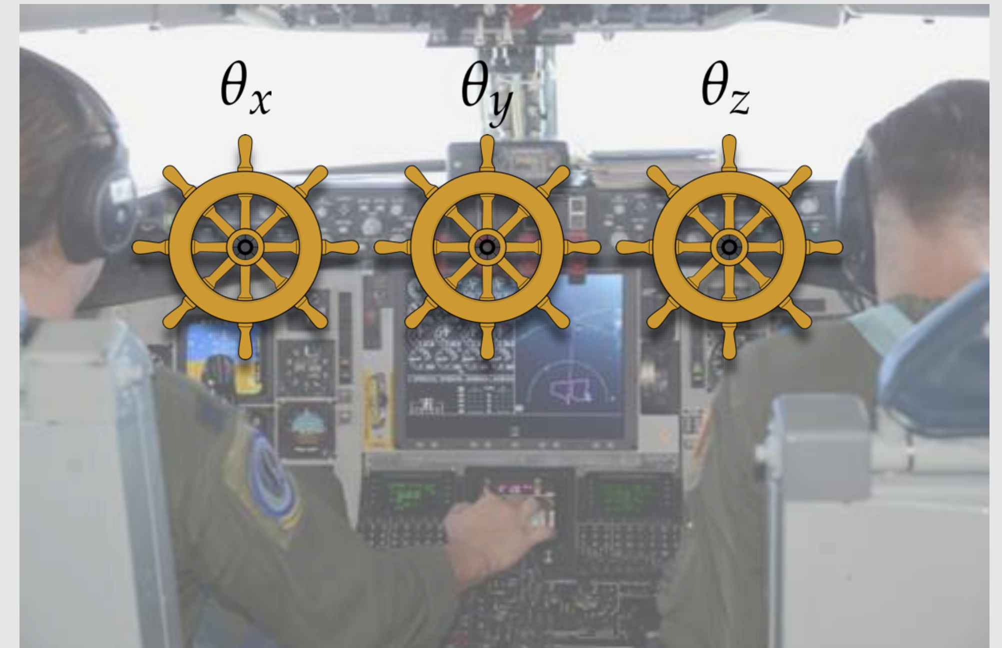
Consider the special case $\theta_y = \pi/2$ (so, $\cos\theta_y = 0$, $\sin\theta_y = 1$):

$$\implies \begin{bmatrix} 0 & 0 & 1 \\ \cos\theta_z\sin\theta_x + \cos\theta_x\sin\theta_z & \cos\theta_x\cos\theta_z - \sin\theta_x\sin\theta_z & 0 \\ -\cos\theta_x\cos\theta_z + \sin\theta_x\sin\theta_z & \cos\theta_z\sin\theta_x + \cos\theta_x\sin\theta_z & 0 \end{bmatrix}$$

Gimbal Lock

- **No matter how we adjust θx, θz, can only rotate in one plane!**
- We are now “locked” into a single axis of rotation
  - Not a great design for airplane controls!





$$\Longrightarrow \begin{bmatrix} 0 & 0 & 1 \\ \cos\theta_z \sin\theta_x + \cos\theta_x \sin\theta_z & \cos\theta_x \cos\theta_z - \sin\theta_x \sin\theta_z & 0 \\ -\cos\theta_x \cos\theta_z + \sin\theta_x \sin\theta_z & \cos\theta_z \sin\theta_x + \cos\theta_x \sin\theta_z & 0 \end{bmatrix}$$

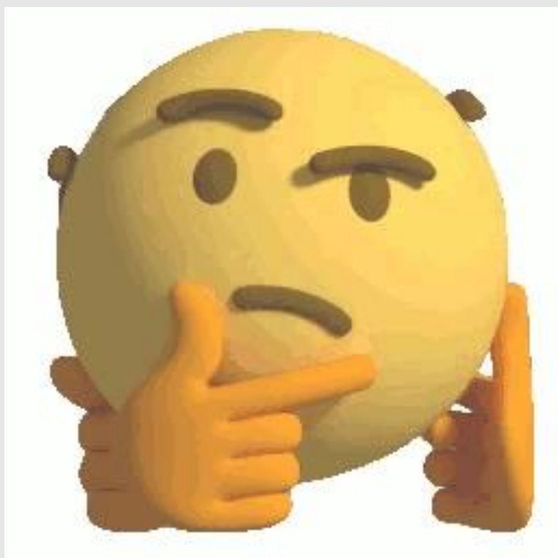
Rotation From Axis/Angle

Alternatively, there is a general expression for a matrix that performs a rotation around a given axis u by a given angle θ:

$$\begin{bmatrix} \cos\theta + u_x^2(1-\cos\theta) & u_x u_y(1-\cos\theta) - u_z\sin\theta & u_x u_z(1-\cos\theta) + u_y\sin\theta \\ u_y u_x(1-\cos\theta) + u_z\sin\theta & \cos\theta + u_y^2(1-\cos\theta) & u_y u_z(1-\cos\theta) - u_x\sin\theta \\ u_z u_x(1-\cos\theta) - u_y\sin\theta & u_z u_y(1-\cos\theta) + u_x\sin\theta & \cos\theta + u_z^2(1-\cos\theta) \end{bmatrix}$$

Just memorize this matrix! :)

Is there a better way to perform 3D rotations?



Bridging The Rotation Gap

- Hamilton wanted to make a 3D equivalent for complex numbers
  - One day, when crossing a bridge, he realized he needed 4 (not 3) coordinates to describe 3D complex number space
    - 1 real and 3 complex components
  - He carved his findings onto a bridge (still there in Dublin)
  - Later known as quaternions

Here as he walked by
on the 16th of October 1843
Sir William Rowan Hamilton
in a flash of genius discovered
the fundamental formula for
quaternion multiplication
$i^2 = j^2 = k^2 = ijk = -1$
& cut it on a stone of this bridge

William Rowan Hamilton
[1805 – 1865]

Quaternions For Math People

- 4 coordinates (1 real, 3 complex) comprise coordinates.
  - H is known as the 'Hamilton Space'

$$\mathbb{H} := \text{span}(\{1, \imath, \jmath, k\})$$

$$q = a + b\imath + c\jmath + dk \in \mathbb{H}$$

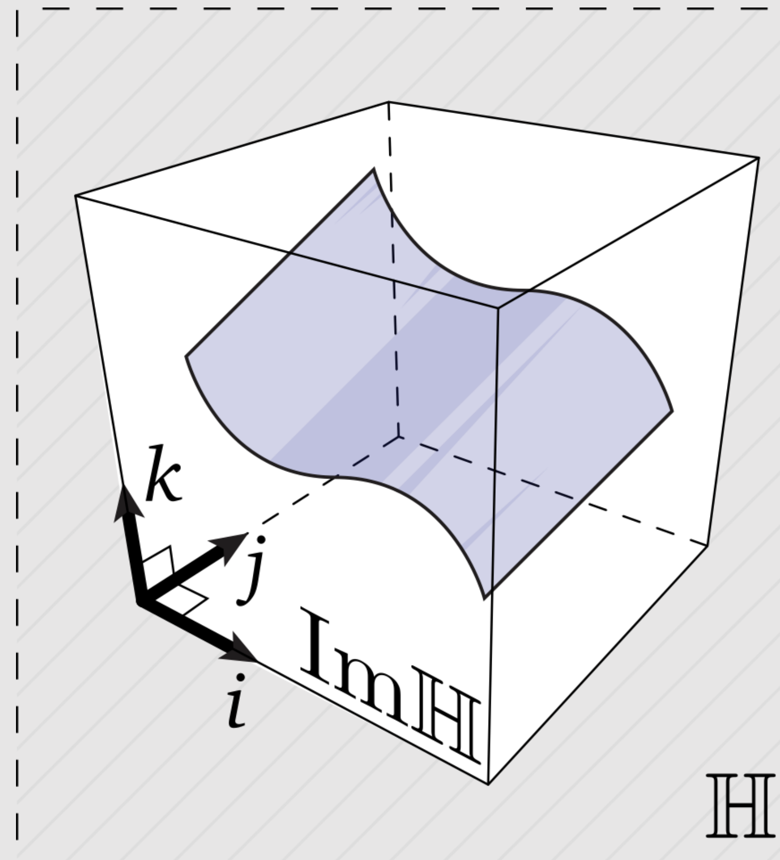
- Quaternion product determined by:

$$\imath^2 = \jmath^2 = k^2 = \imath\jmath k = -1$$

- **Warning:** product no longer commutes!

$$\text{For } q, p \in \mathbb{H}, \quad qp \neq pq$$

  - With 3D rotations, order matters.



Quaternions For Non-Math People

- Recall axis-angle rotations
  - Represent an axis with 3 coordinates (i, j, k)
  - Represent an angle by some scalar a

$$q = a + b\imath + c\jmath + dk \in \mathbb{H}$$

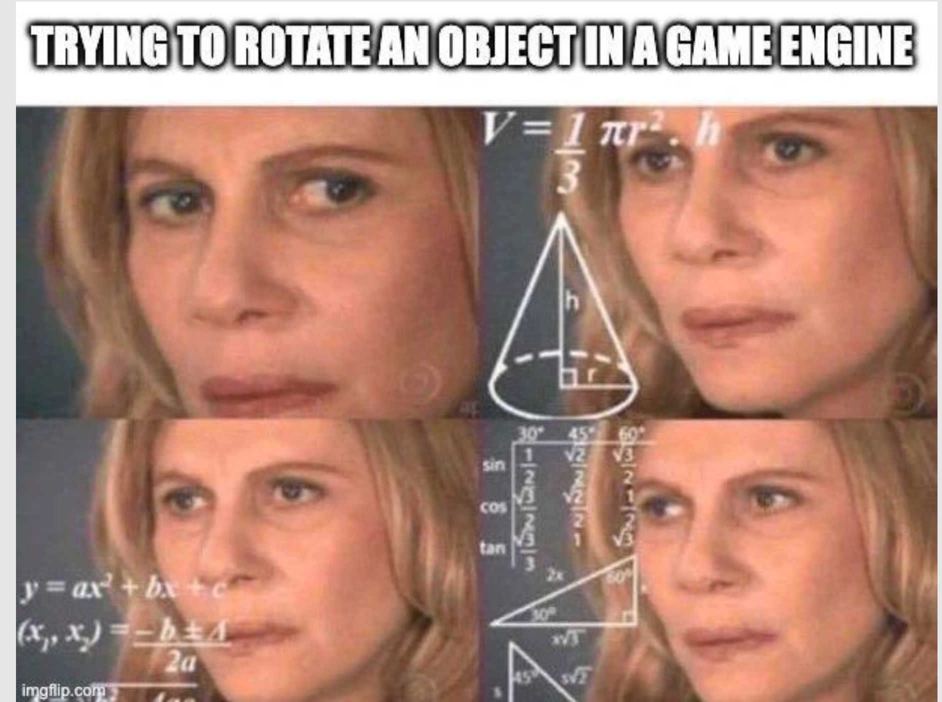
- Just like how we multiply rotation matrices together, we can also multiply complex components. If we represent:
  - i as a 90deg rotation about x-axis
  - j as a 90deg rotation about y-axis
  - k as a 90deg rotation about z-axis

$$\imath^2 = \jmath^2 = k^2 = \imath\jmath k = -1$$

  - Then two 90deg rotations about the same axis will produce the inverted image, the same as scaling by -1
  - This can also be rewritten as:

$$ij = k$$

    - A 90deg x-axis rotation and a 90deg y-axis rotation is the same as a 90deg z-axis rotation
    - Can be rewritten in any other way



Multiplying Quaternions

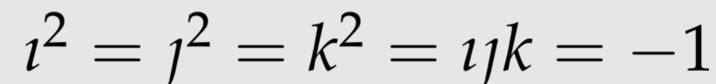
Given two quaternions:

$$q = a_1 + b_1 \imath + c_1 \jmath + d_1 k$$
$$p = a_2 + b_2 \imath + c_2 \jmath + d_2 k$$

Can express their product as:

$$\begin{aligned} qp = & \; a_1a_2 - b_1b_2 - c_1c_2 - d_1d_2 \\ & + (a_1b_2 + b_1a_2 + c_1d_2 - d_1c_2)\imath \\ & + (a_1c_2 - b_1d_2 + c_1a_2 + d_1b_2)\jmath \\ & + (a_1d_2 + b_1c_2 - c_1b_2 + d_1a_2)k \end{aligned}$$

recall



The result still looks like a quaternion
But there's a better way to multiply…

Multiplying Quaternions

Recall quaternions can be thought of as an axis and angle:

$$(x, y, z) \mapsto 0 + x\imath + y\jmath + zk$$

$$\underset{\mathbb{R}}{\underset{\cap}{(\text{scalar},}}\ \underset{\mathbb{R}^3}{\underset{\cap}{\text{vector}}}) \in \mathbb{H}$$

Can express their product as:

$$(a, \mathbf{u})(b, \mathbf{v}) = (ab - \mathbf{u} \cdot \mathbf{v}, a\mathbf{v} + b\mathbf{u} + \mathbf{u} \times \mathbf{v})$$

If the scalar components are 0, we get:

$$\mathbf{uv} = \mathbf{u} \times \mathbf{v} - \mathbf{u} \cdot \mathbf{v}$$

Rotating With Quaternions

- **Goal:** rotate x by angle θ around axis $u = (x, y, z)$:
  - Make x imaginary, and build q based on u and θ
    - **Note:** components of q must be normalized!

$$x \in \mathrm{Im}(\mathbb{H})$$

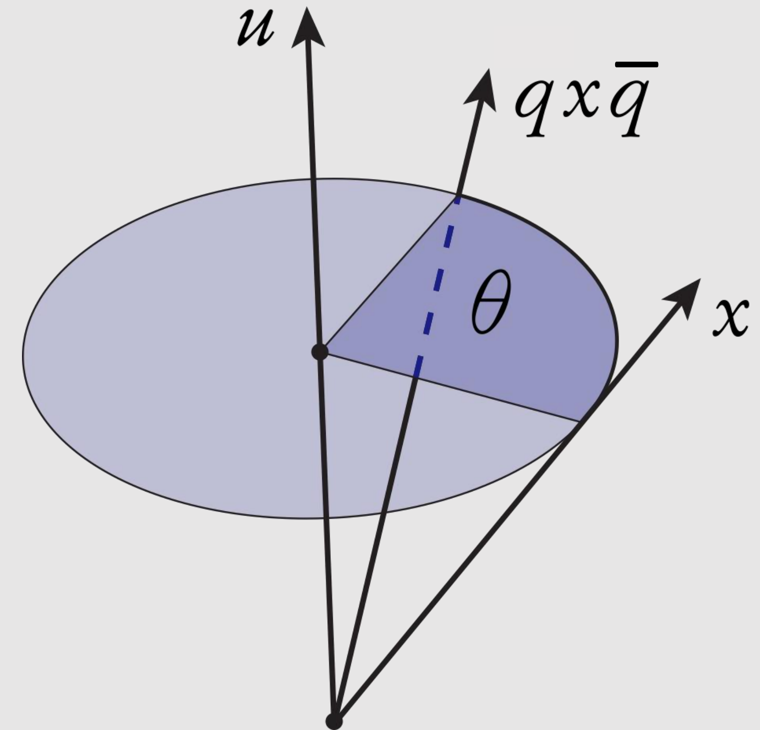
$$q \in \mathbb{H}, \quad |q|^2 = 1$$

$$q = \cos(\theta/2) + \sin(\theta/2)u$$

- q now looks like:

$$q = a + b\imath + c\jmath + dk \in \mathbb{H}$$

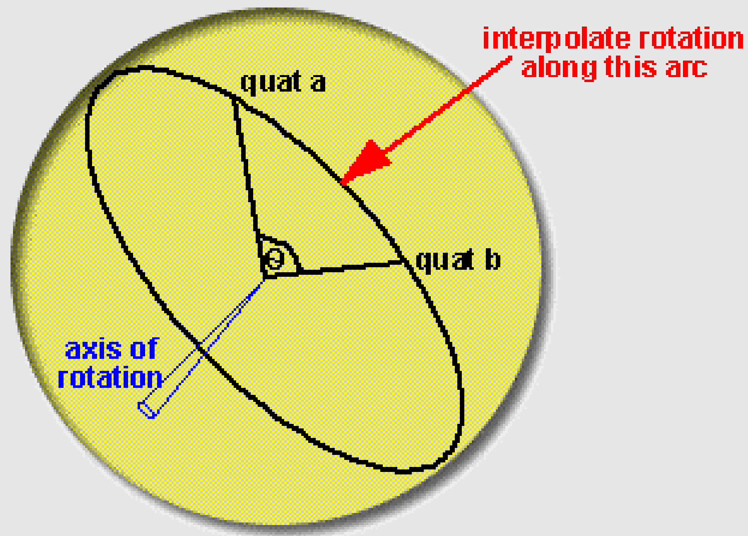
- $\bar{q}$ is q with every complex component negative
- Now just compute $qx\bar{q}$to get final rotation



Interpolating With Quaternions

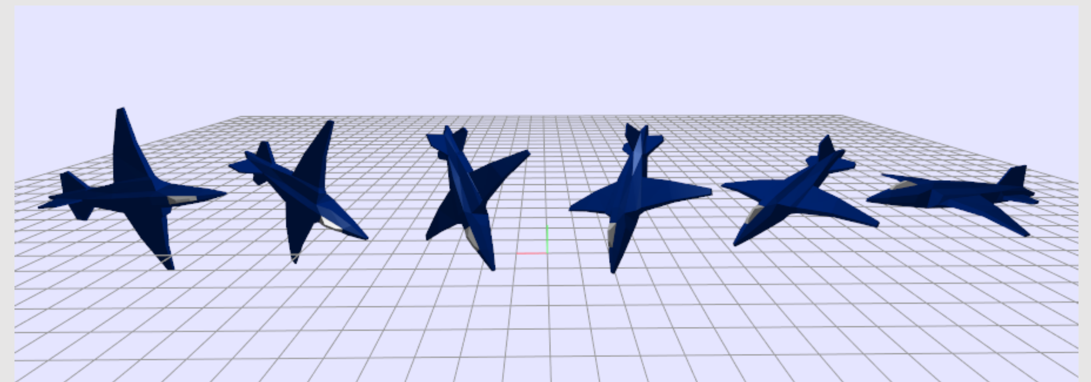
- Interpolating Euler angles can yield strange-looking paths, non-uniform rotation speed, etc.
  - Simple solution w/ quaternions: "SLERP" (spherical linear interpolation):

$$\text{Slerp}(q_0, q_1, t) = q_0(q_0^{-1}q_1)^t, \quad t \in [0,1]$$



Animating Rotation with Quaternion Curves (1985) Shoemake

Fifa '15 (2014) Electronic Arts



Texture Mapping With Quaternions

- Quaternions can be used to generate texture maps coordinates
  - Complex numbers are natural language for angle-preserving ("conformal") maps