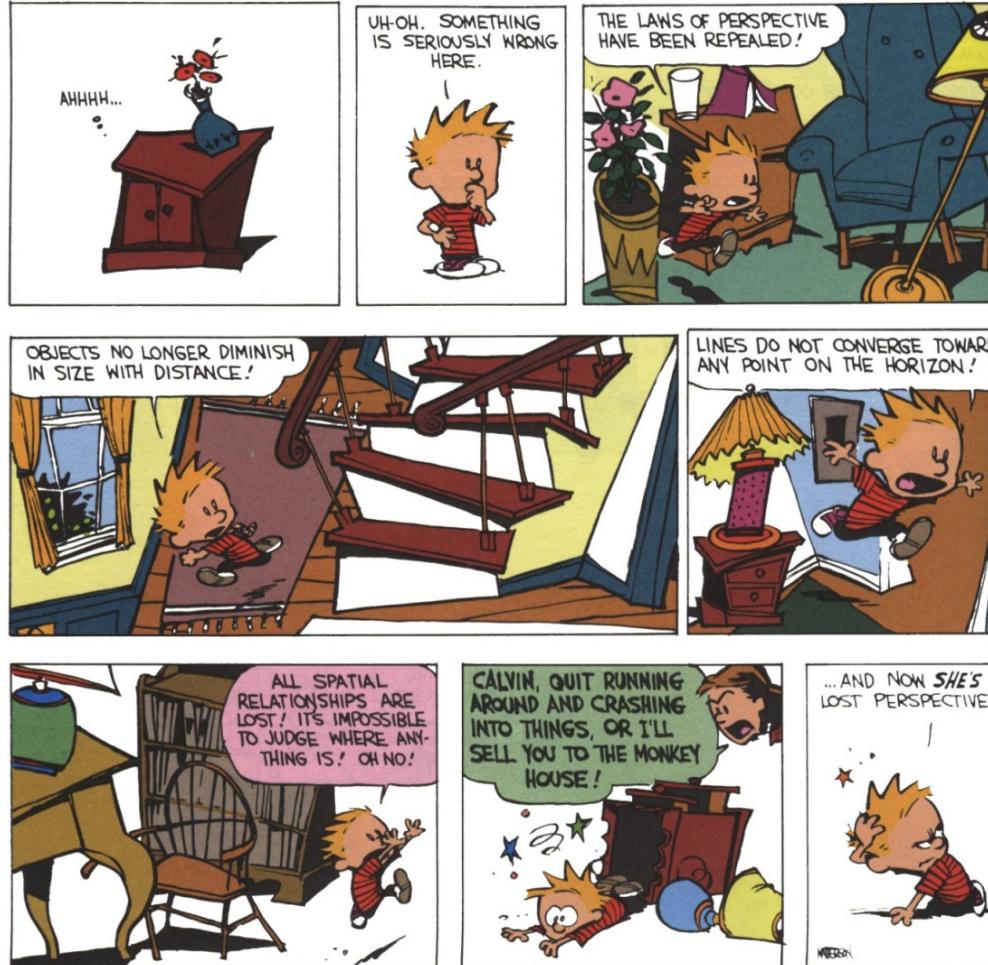
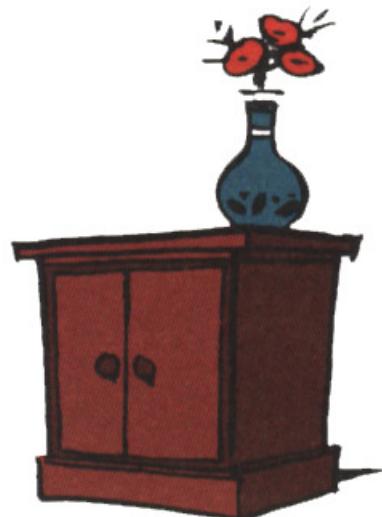


Graphics Pipeline & Rasterization II

calvin
and
hobbes

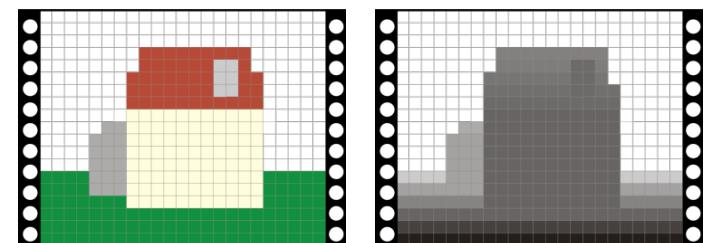
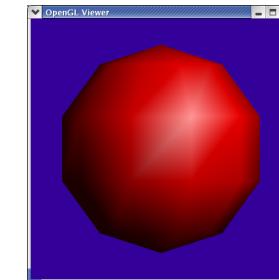
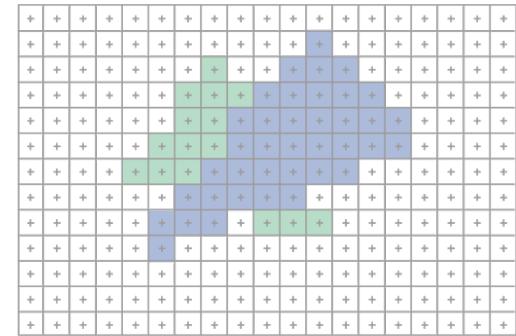
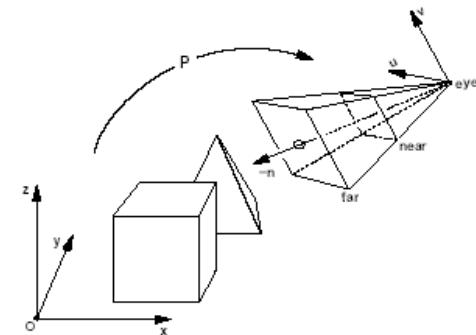
BY WATTERSON



ISTD 50.017
Graphics & Visualization
Sai-Kit Yeung

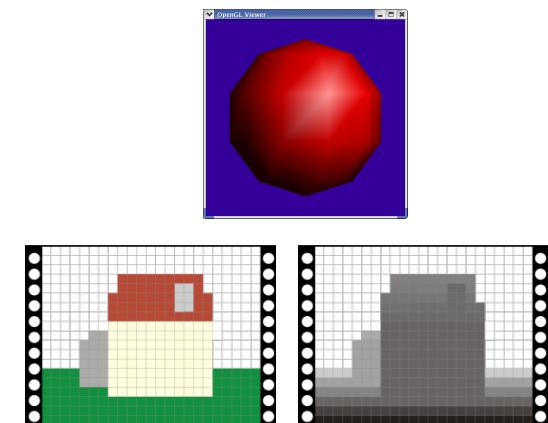
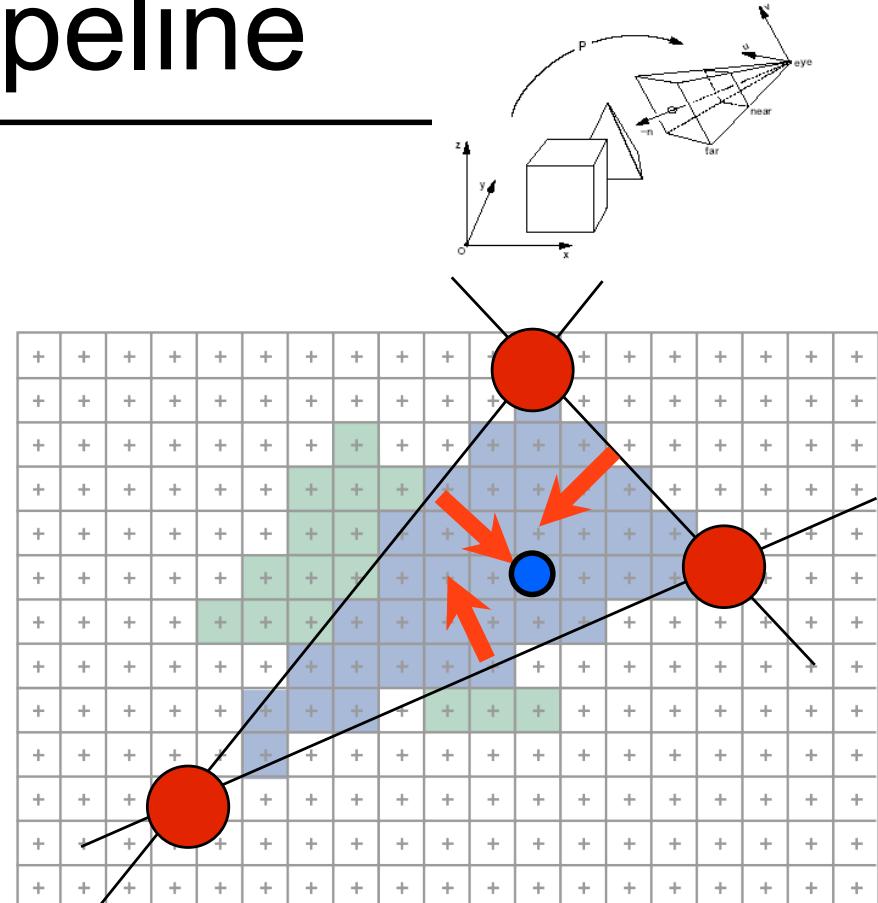
Modern Graphics Pipeline

- Project vertices to 2D (image)
- Rasterize triangle: find which pixels should be lit
- Compute per-pixel color
- Test visibility (Z-buffer), update frame buffer color



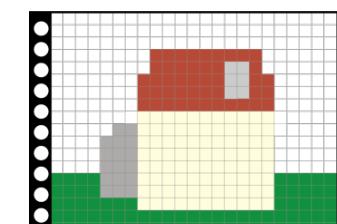
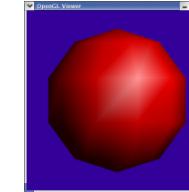
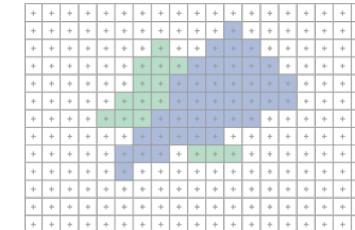
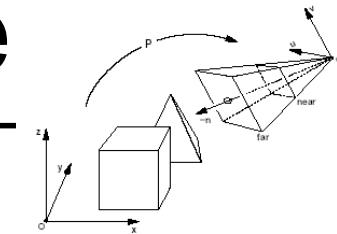
Modern Graphics Pipeline

- Project vertices to 2D (image)
- Rasterize triangle: find which pixels should be lit
 - For each pixel, test 3 edge equations
 - if all pass, draw pixel
- Compute per-pixel color
- Test visibility (Z-buffer), update frame buffer color

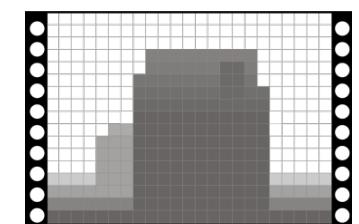


Modern Graphics Pipeline

- Perform projection of vertices
- Rasterize triangle: find which pixels should be lit
- Compute per-pixel color
- Test visibility,
update frame buffer color
 - Store minimum distance to camera
for each pixel in “Z-buffer”
 - ~same as t_{min} in ray casting!
 - ```
if new_z < zbuffer[x, y]
 zbuffer[x, y] = new_z
 framebuffer[x, y] = new_color
```



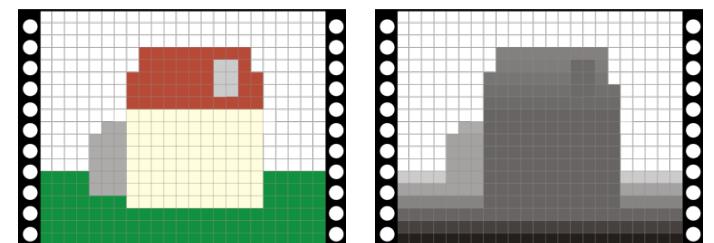
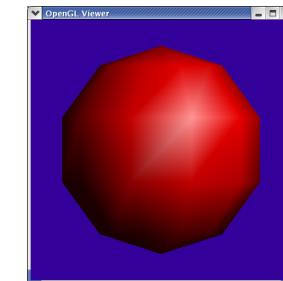
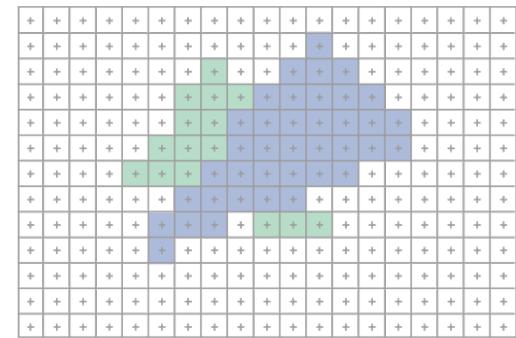
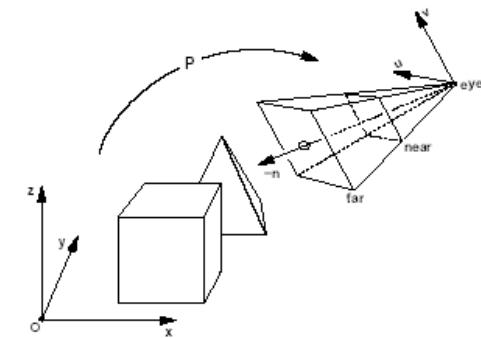
frame buffer



Z buffer

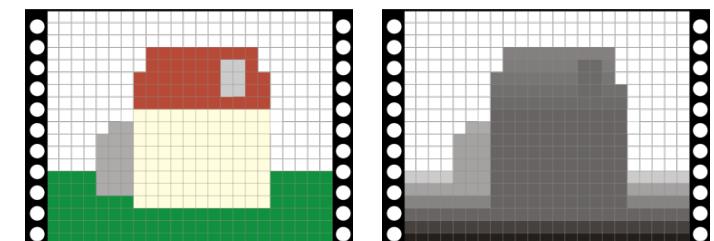
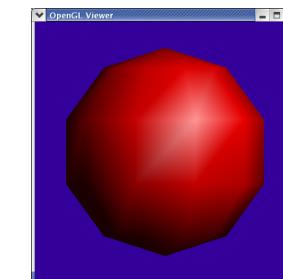
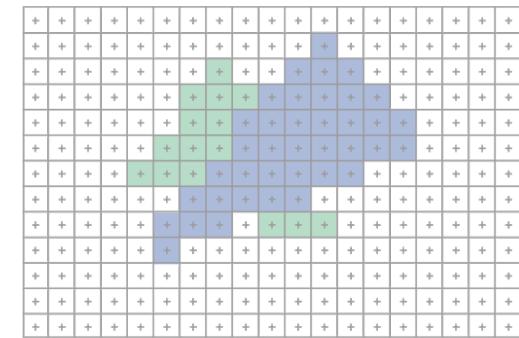
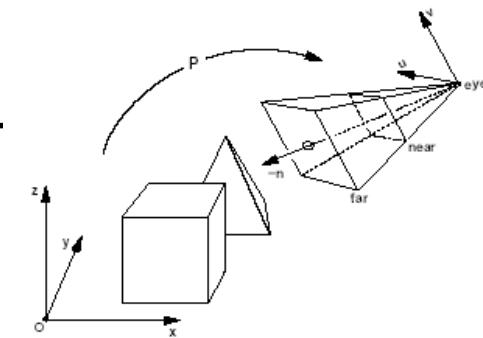
# Modern Graphics Pipeline

For each triangle  
transform into eye space  
(perform projection)  
setup 3 edge equations  
for each pixel  $x, y$   
if passes all edge equations  
compute  $z$   
if  $z < z\text{buffer}[x, y]$   
 $z\text{buffer}[x, y] = z$   
 $\text{framebuffer}[x, y] = \text{shade}()$



# Modern Graphics Pipeline

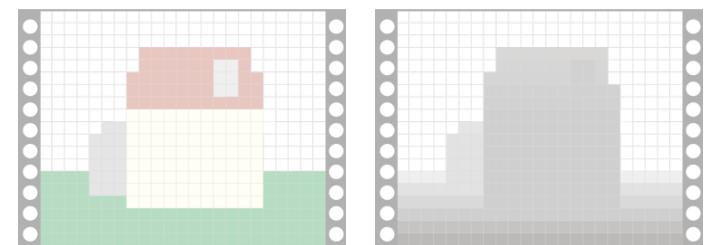
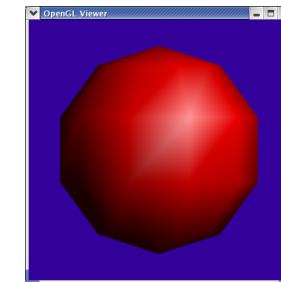
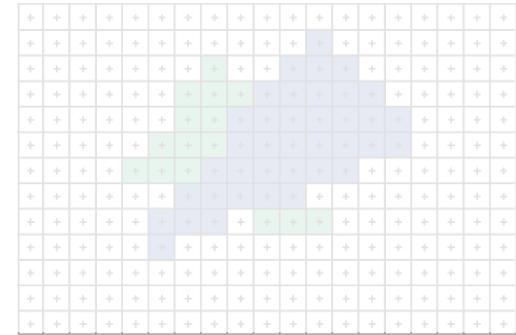
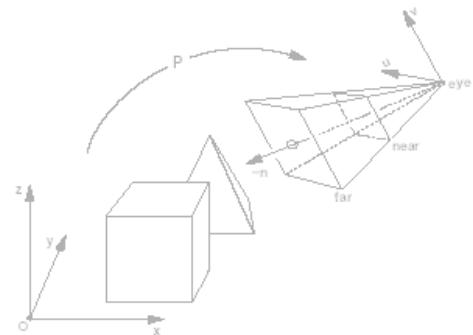
For each triangle  
transform into eye space  
(perform projection)  
setup 3 edge equations  
for each pixel  $x, y$   
if passes all edge equations  
compute  $z$   
if  $z < z\text{buffer}[x, y]$   
 $z\text{buffer}[x, y] = z$   
 $\text{framebuffer}[x, y] = \text{shade}()$



## Questions?

# Modern Graphics Pipeline

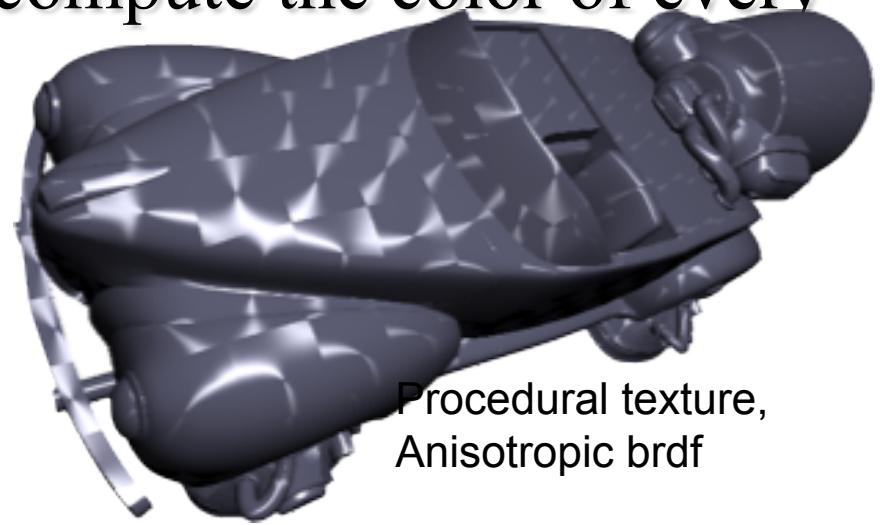
- Perform projection of vertices
- Rasterize triangle: find which pixels should be lit
- Compute per-pixel color
- Test visibility, update frame buffer



# Pixel Shaders

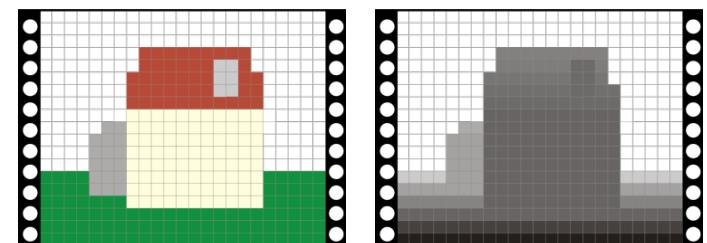
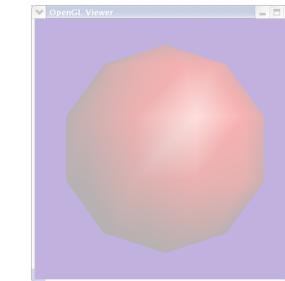
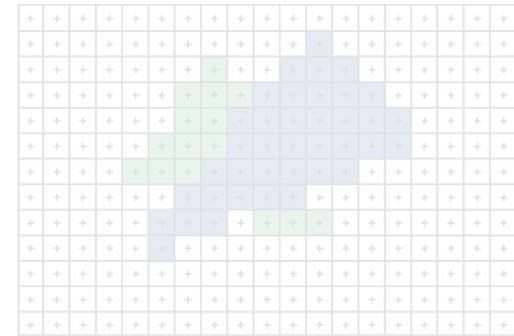
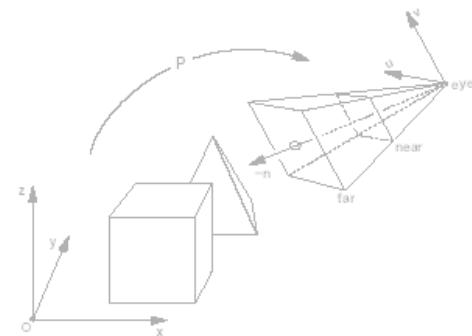
---

- Modern graphics hardware enables the execution of rather complex programs to compute the color of every single pixel
  - More next lesson



# Modern Graphics Pipeline

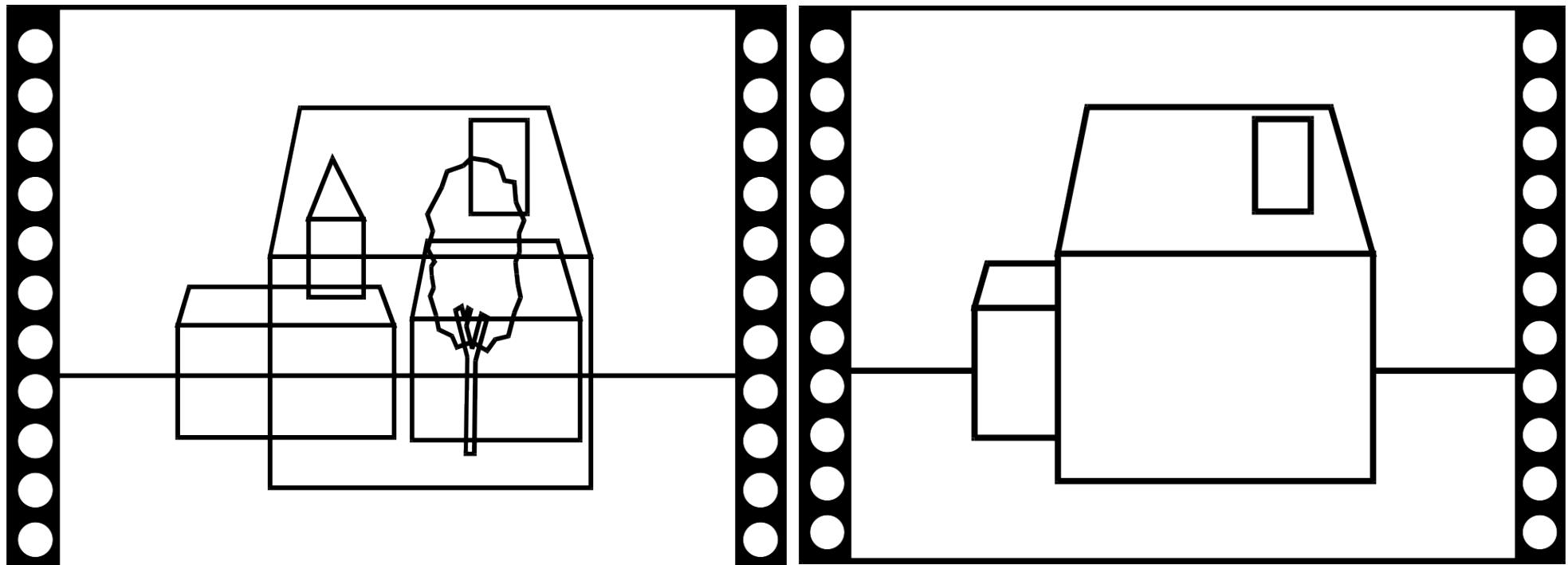
- Perform projection of vertices
- Rasterize triangle: find which pixels should be lit
- Compute per-pixel color
- Test visibility, update frame buffer



# Visibility

---

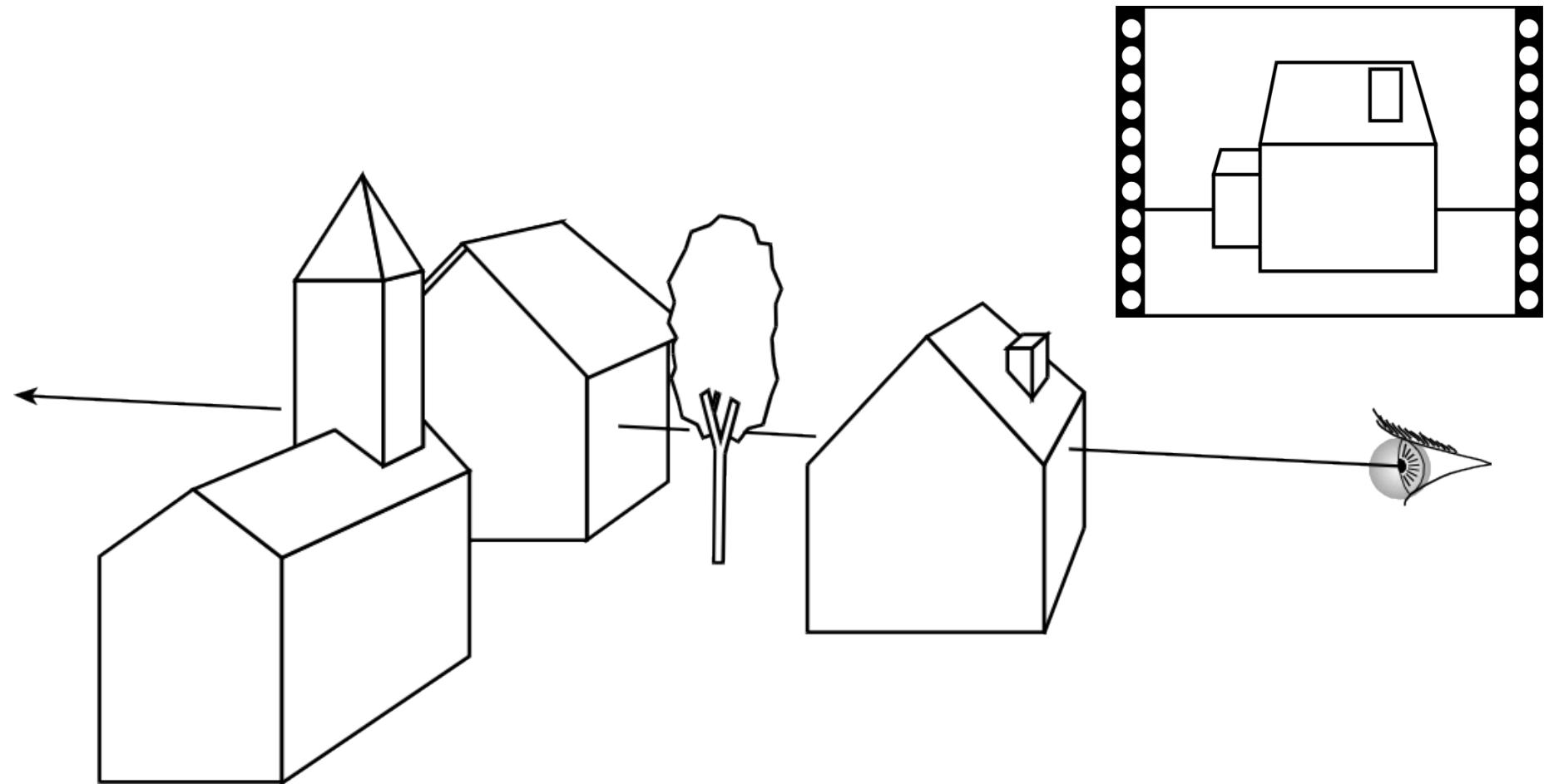
- How do we know which parts are visible/in front?



# Ray Casting

---

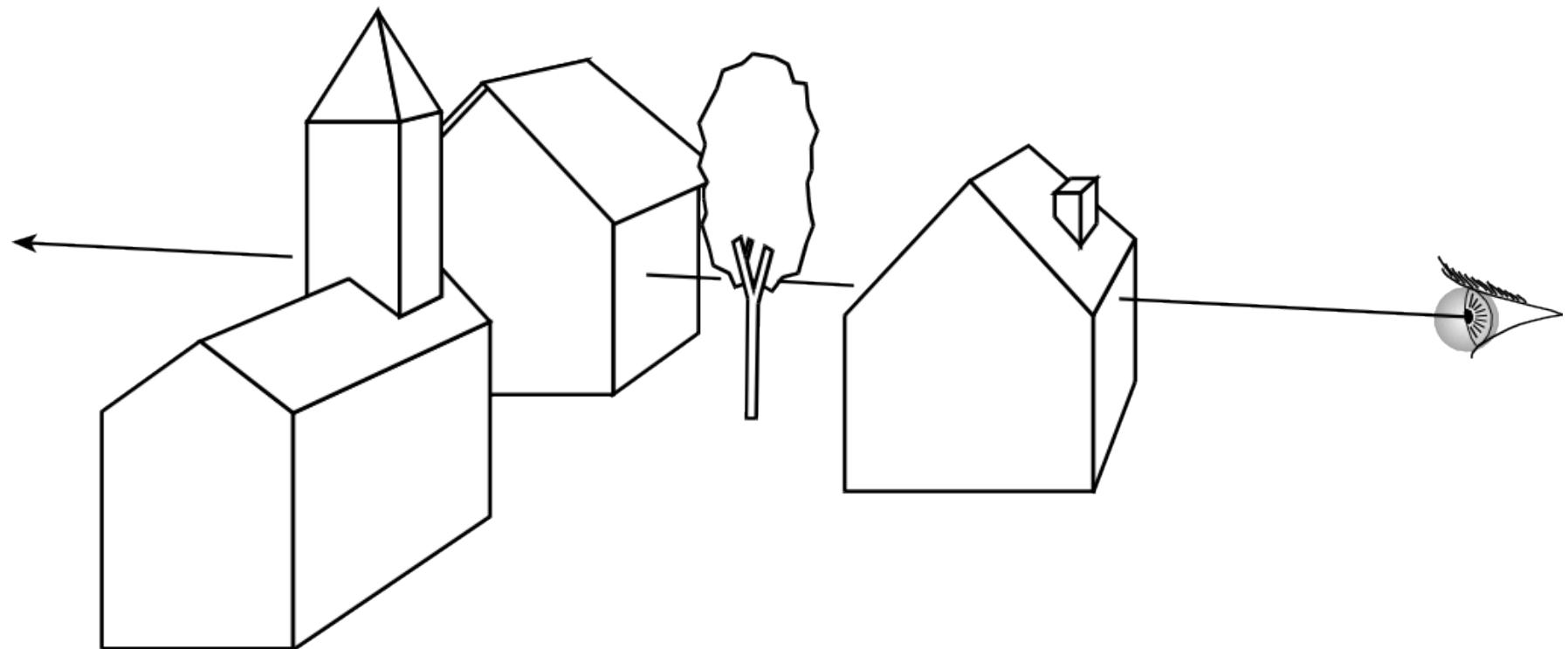
- Maintain intersection with closest object



# Visibility

---

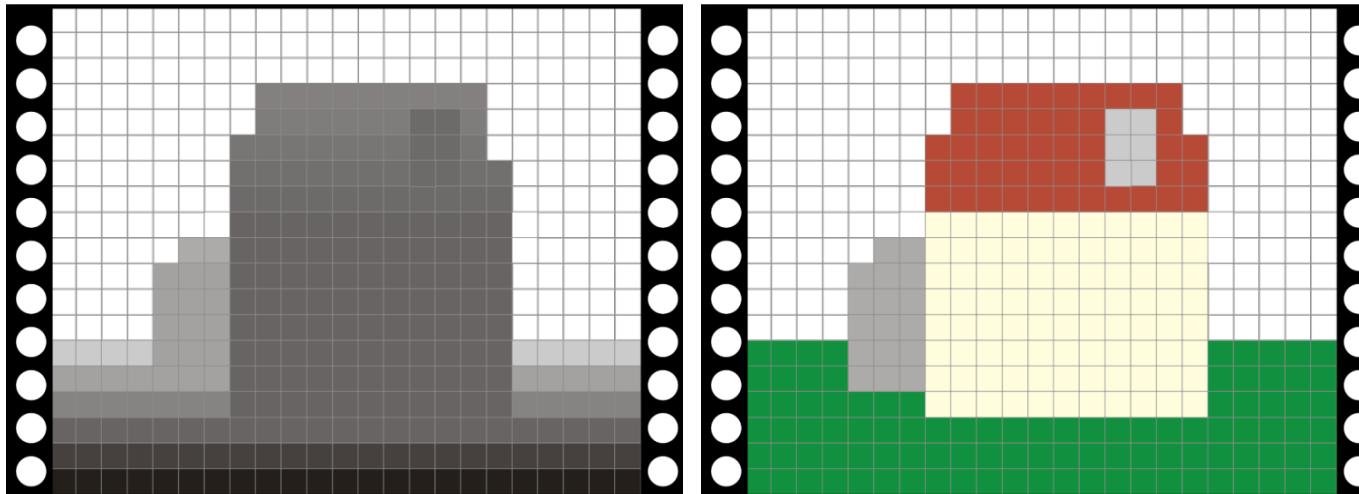
- In ray casting, use intersection with closest  $t$
- Now we have swapped the loops (pixel, object)
- What do we do?



# Z buffer

---

- In addition to frame buffer (R, G, B)
- Store distance to camera (*z*-buffer)
- Pixel is updated only if *newz* is closer than *z*-buffer value



# Z-buffer pseudo code

---

For every triangle

    Compute Projection, color at vertices

    Setup line equations

    Compute bbox, clip bbox to screen limits

    For all pixels in bbox

        Increment line equations

        Compute currentZ

        Compute currentColor

        If all line equations>0 //pixel [x,y] in triangle

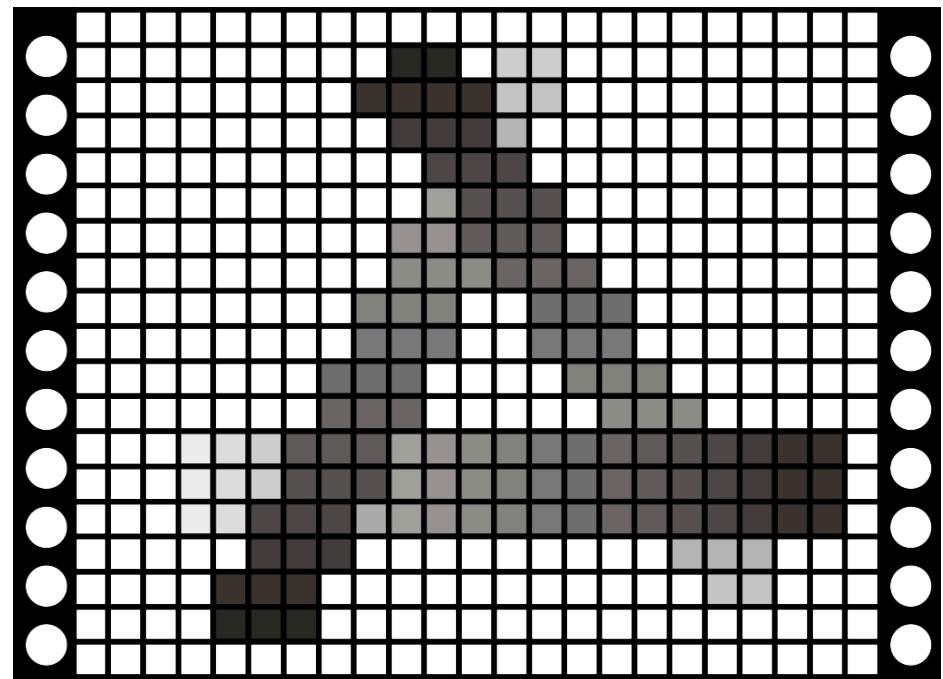
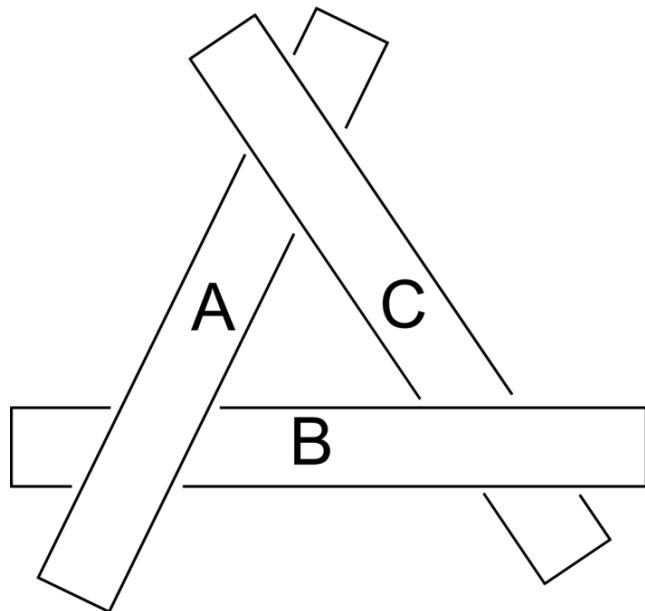
            If currentZ<zBuffer[x, y] //pixel is visible

                Framebuffer[x, y]=currentColor

                zBuffer[x, y]=currentZ

# Works for hard cases!

---



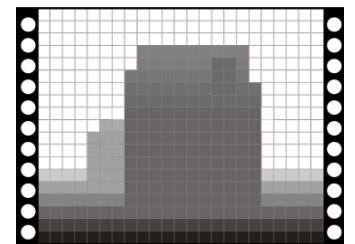
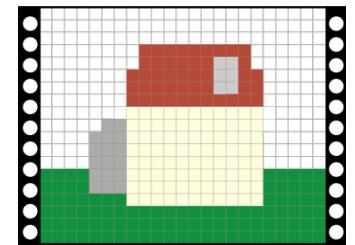
# Interpolation in Screen Space

---

- How do we get that Z value for each pixel?
  - We only know z at the vertices...
  - (Remember, screen-space z is actually  $z'/w'$ )
  - Must interpolate from vertices into triangle interior

For each triangle

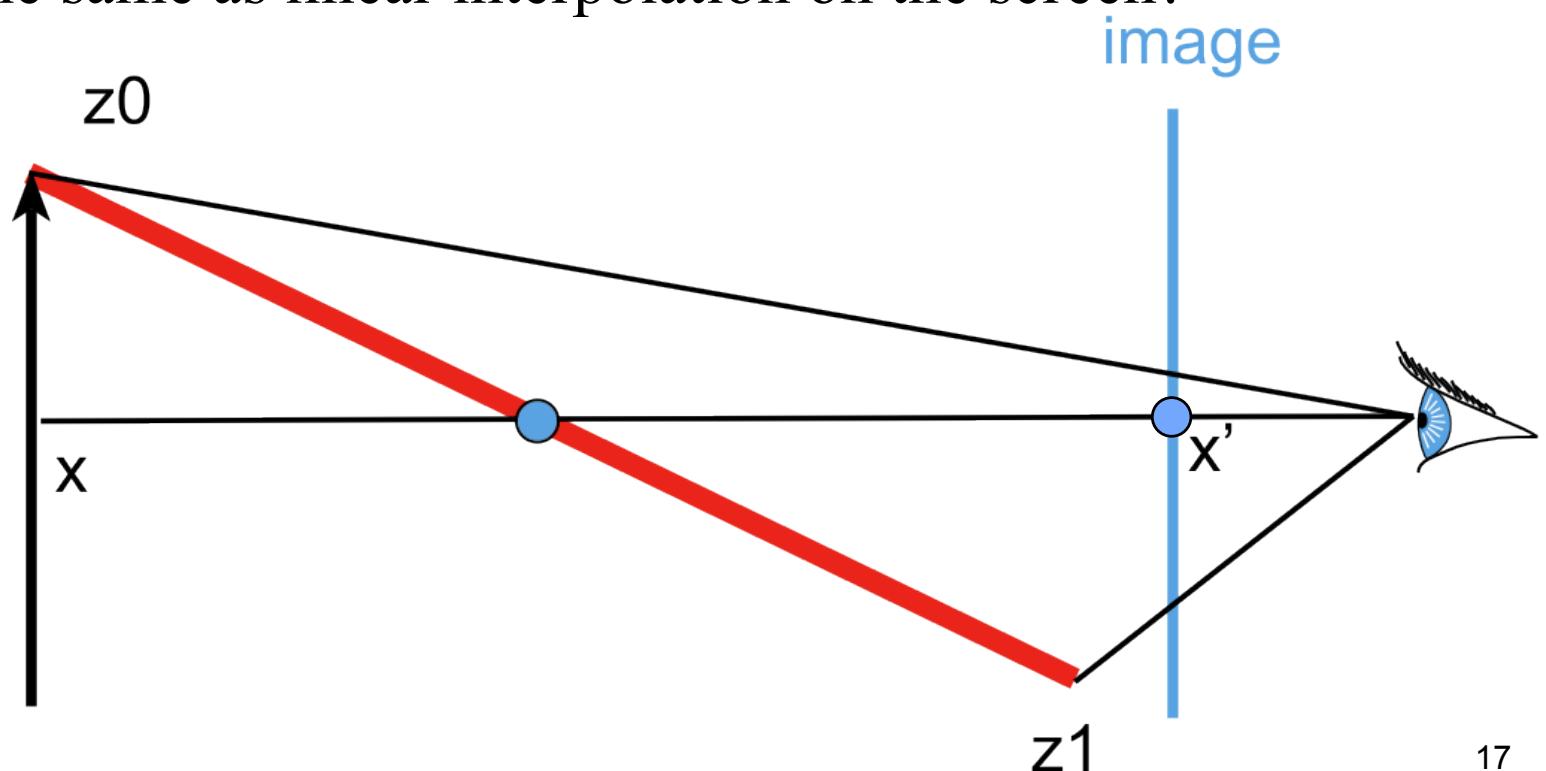
```
for each pixel (x, y)
 if passes all edge equations
 compute z
 if z < zbuffer[x, y]
 zbuffer[x, y] = z
 framebuffer[x, y] = shade()
```



# Interpolation in Screen Space

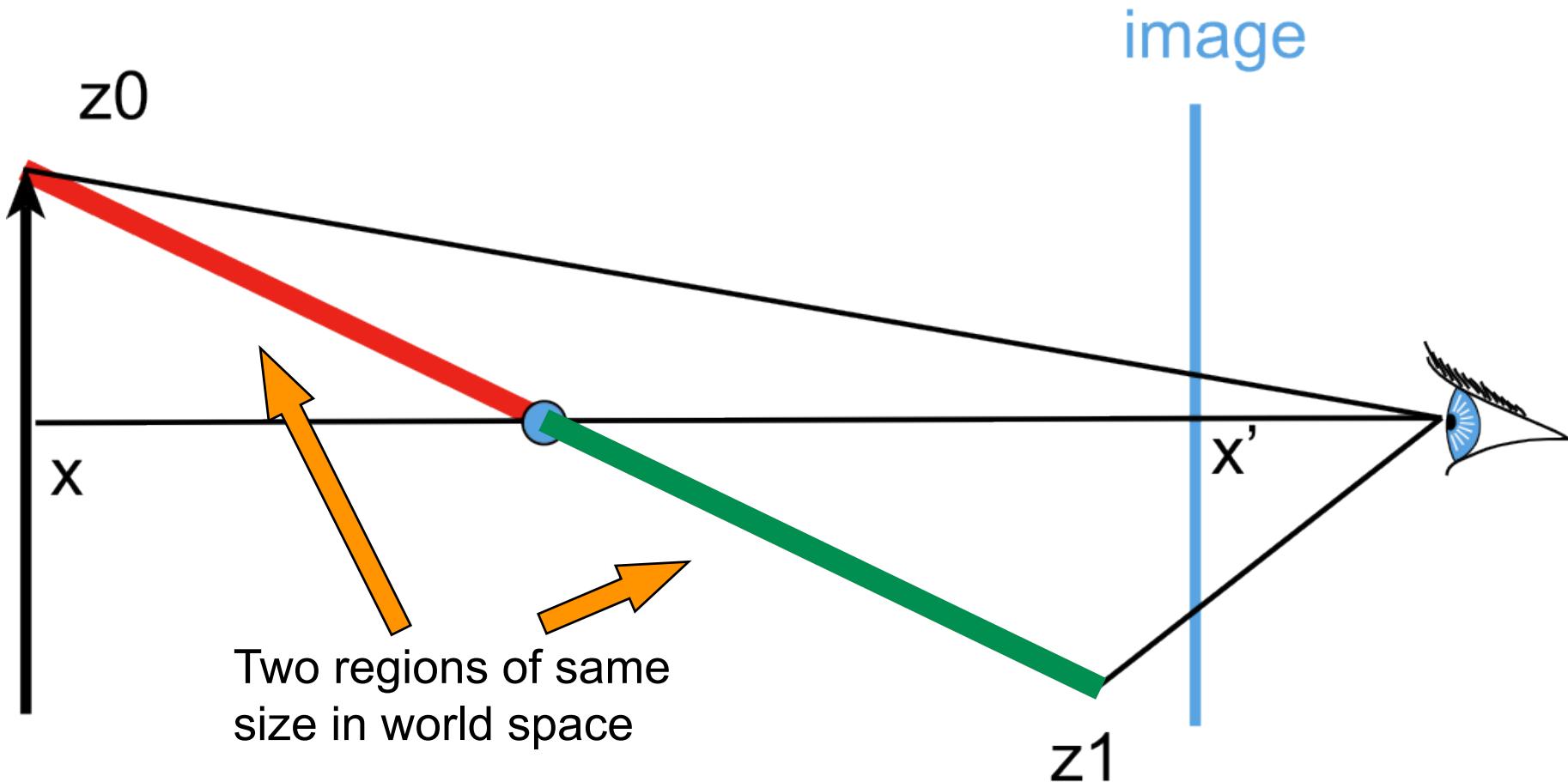
---

- Also need to interpolate color, normals, texture coordinates, etc. between vertices
  - We did this with barycentrics in ray casting
    - Linear interpolation in object space
  - Is this the same as linear interpolation on the screen?



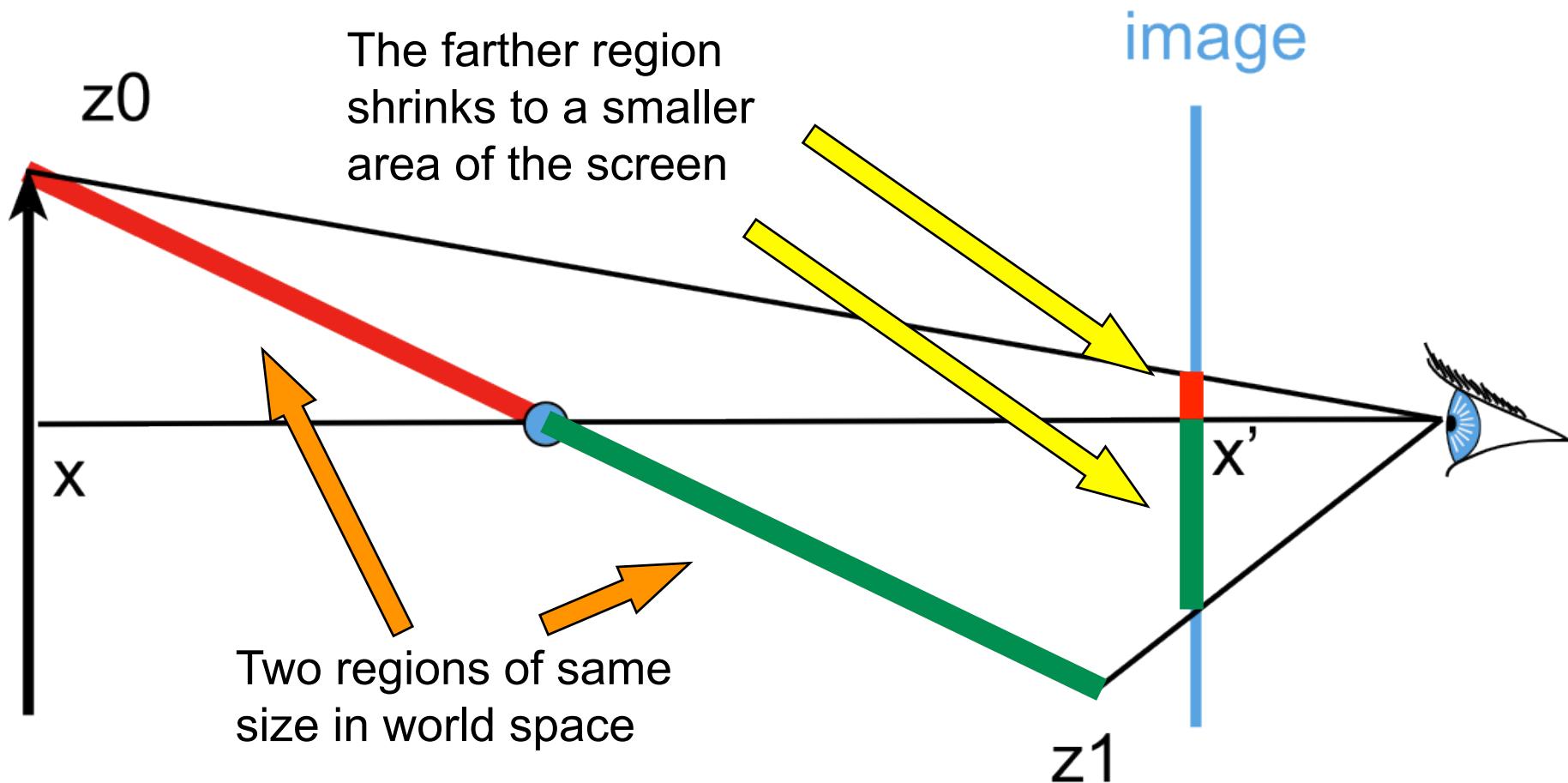
# Interpolation in Screen Space

---



# Interpolation in Screen Space

---

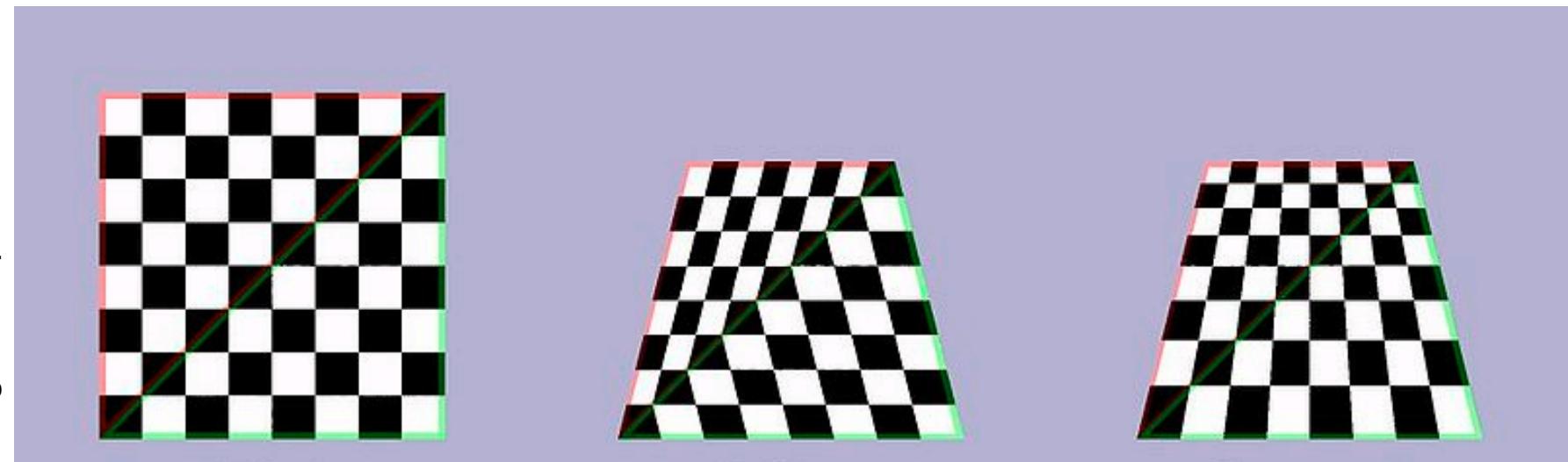


# Nope, Not the Same

---

- Linear variation in world space does not yield linear variation in screen space due to projection
  - Think of looking at a checkerboard at a steep angle; all squares are the same size on the plane, but not on screen

Image: Wikipedia



Head-on view

linear screen-space  
("Gouraud") interpolation

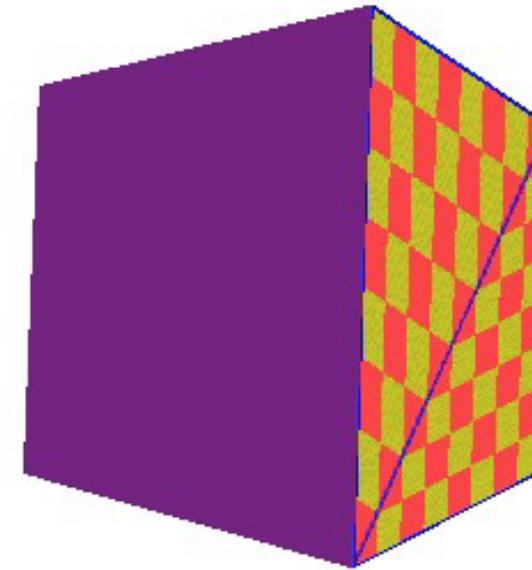
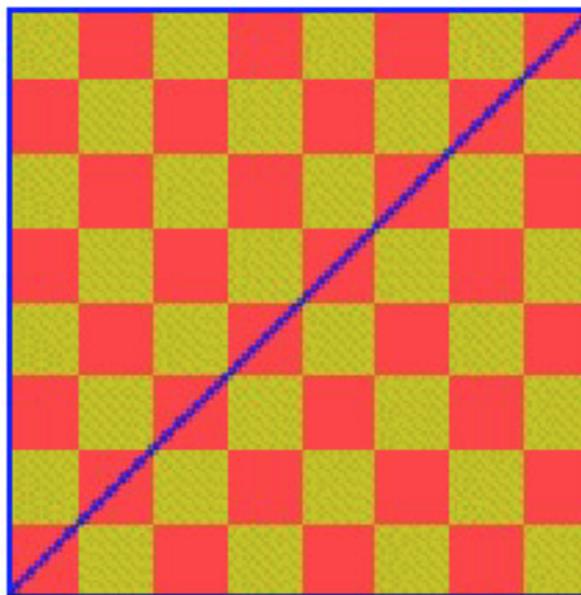
**BAD**

Perspective-correct  
Interpolation

# Nope, Not the Same

---

- Linear variation in world space does not yield linear variation in screen space due to projection
  - Distortion when the triangle vertices do not have the same depth



# Back to the basics: Barycentrics

---

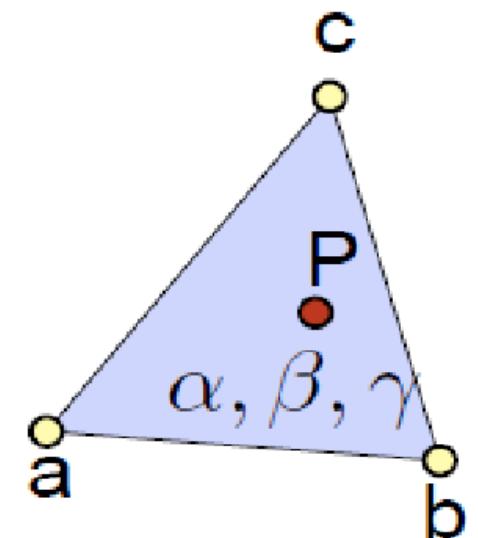
- Barycentric coordinates for a triangle ( $a, b, c$ )

$$P(\alpha, \beta, \gamma) = \alpha \mathbf{a} + \beta \mathbf{b} + \gamma \mathbf{c}$$

- Remember,  $\alpha + \beta + \gamma = 1$ ,  $\alpha, \beta, \gamma \geq 0$

- Barycentrics are very general:

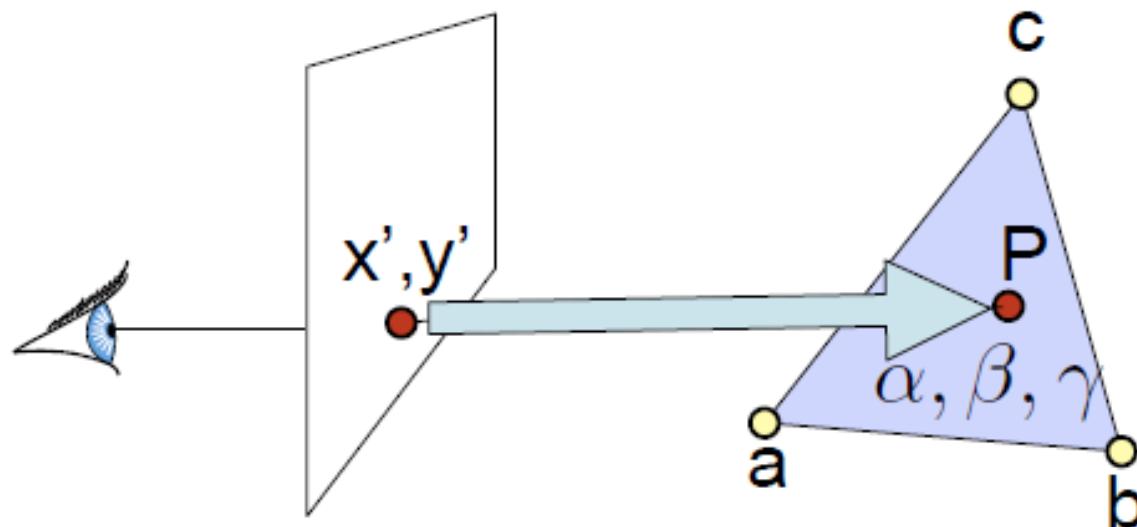
- Work for  $x, y, z, u, v, r, g, b$
  - Anything that varies linearly in object space
  - including  $z$



# Basic strategy

---

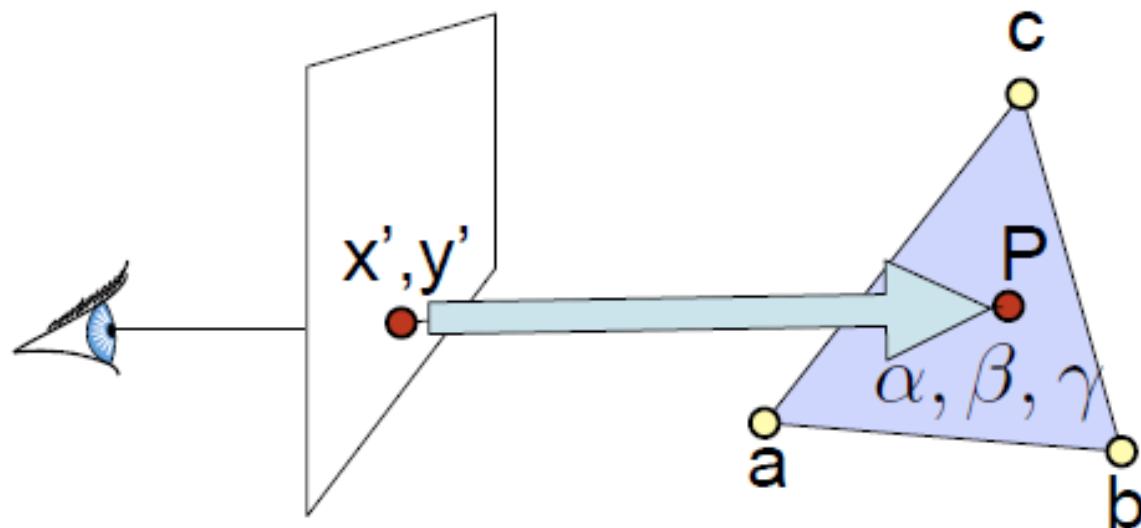
- Given screen-space  $x'$ ,  $y'$  (determined from edge function)
- Compute barycentric coordinates
- Interpolate anything specified at the three vertices



# Basic strategy

---

- How to compute barycentric weights given  $x'$ ,  $y'$ 
  - start by computing  $x'$ ,  $y'$  given barycentrics
  - invert
- Later: shortcut barycentrics, directly build interpolants



# From barycentric to screen-space

---

- Barycentric coordinates for a triangle (a, b, c)

$$P(\alpha, \beta, \gamma) = \alpha \mathbf{a} + \beta \mathbf{b} + \gamma \mathbf{c}$$

– Remember,  $\alpha + \beta + \gamma = 1$ ,  $\alpha, \beta, \gamma \geq 0$

- Assume we know the barycentric coordinate
- Let's project point P by projection matrix C

$$\begin{aligned} CP &= C(\alpha \mathbf{a} + \beta \mathbf{b} + \gamma \mathbf{c}) \\ &= \alpha C\mathbf{a} + \beta C\mathbf{b} + \gamma C\mathbf{c} \\ &= \alpha \mathbf{a}' + \beta \mathbf{b}' + \gamma \mathbf{c}' \end{aligned}$$

a', b', c' are the  
projected  
homogeneous  
vertices before  
division by w

# Projection

---

- Let's use simple formulation of projection going from 3D homogeneous coordinates (4 numbers) to 2D homogeneous coordinates (3 numbers)

$$C = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

- No crazy near-far or storage of  $1/z$ , we can skip the last row
- We use ' for screen space coordinates

# From barycentric to screen-space

---

- From previous slides:

$$P' = CP = \alpha \mathbf{a}' + \beta \mathbf{b}' + \gamma \mathbf{c}'$$

a', b', c' are the  
projected  
homogeneous  
vertices

- Seems to suggest it's linear in screen space.  
But it's homogenous coordinates

# From barycentric to screen-space

---

- From previous slides:

$$P' = CP = \alpha \mathbf{a}' + \beta \mathbf{b}' + \gamma \mathbf{c}'$$

a', b', c' are the projected homogeneous vertices

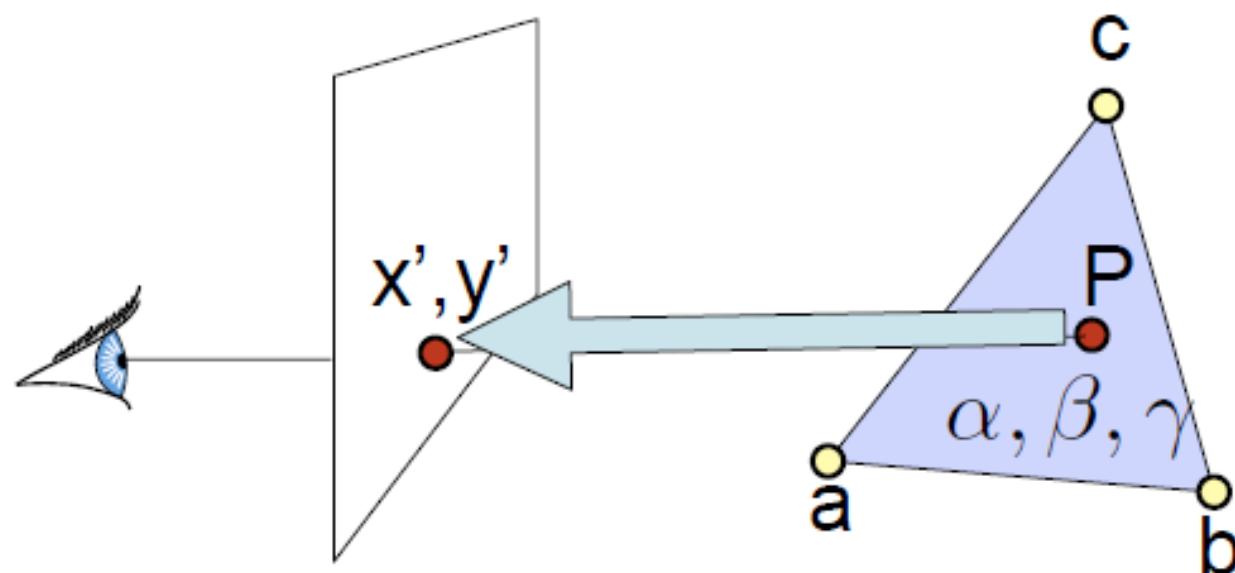
- Seems to suggest it's linear in screen space.  
But it's homogenous coordinates
- After division by  $w$  ( $z$  in reality), the  $(x,y)$  screen coordinates are

$$(P'_x/P'_w, P'y/P'w) =$$

$$\left( \frac{\alpha a'_x + \beta b'_x + \gamma c'_x}{\alpha a'_w + \beta b'_w + \gamma c'_w}, \frac{\alpha a'_y + \beta b'_y + \gamma c'_y}{\alpha a'_w + \beta b'_w + \gamma c'_w} \right)$$

# Recap: barycentric to screen-space

| projective equivalence<br>(up to scale)       | 2D homogenous coordinates                                   | projected vertices                                                                             | barycentric coordinates                                   |
|-----------------------------------------------|-------------------------------------------------------------|------------------------------------------------------------------------------------------------|-----------------------------------------------------------|
| $\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix}$ | $\sim \begin{pmatrix} P'_x \\ P'_y \\ P'_w \end{pmatrix} =$ | $\begin{pmatrix} a'_x & b'_x & c'_x \\ a'_y & b'_y & c'_y \\ a'_z & b'_z & c'_z \end{pmatrix}$ | $\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}$ |



# From screen-space to barycentric

---

| projective<br>equivalence<br>(up to scale)                                                                                                                                                                                                                   | 2D<br>homogenous<br>coordinates | projected<br>vertices | barycentric<br>coordinates |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------|-----------------------|----------------------------|
| $\begin{pmatrix} x' \\ y' \\ 1 \end{pmatrix} \sim \begin{pmatrix} P'_x \\ P'_y \\ P'_w \end{pmatrix} = \begin{pmatrix} a'_x & b'_x & c'_x \\ a'_y & b'_y & c'_y \\ a'_z & b'_z & c'_z \end{pmatrix} \begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix}$ |                                 |                       |                            |

- It's a projective mapping from the barycentrics onto screen coordinates!
  - Represented by a 3x3 matrix
- We'll take the inverse mapping to get from  $(x, y, 1)$  to the barycentrics!

# From Screen to Barycentrics

---

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} \underset{\text{projective equivalence}}{\sim} \begin{pmatrix} a'_x & b'_x & c'_x \\ a'_y & b'_y & c'_y \\ a'_w & b'_w & c'_w \end{pmatrix}^{-1} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- Recipe
  - Compute projected homogeneous coordinates  $a'$ ,  $b'$ ,  $c'$
  - Put them in the columns of a matrix, invert it
  - Multiply screen coordinates  $(x, y, 1)$  by inverse matrix
  - Then divide by the sum of the resulting coordinates
    - This ensures the result sums to one like barycentrics should
  - Then interpolate value (e.g.  $Z$ ) from vertices using them!

# From Screen to Barycentrics

---

$$\begin{pmatrix} \alpha \\ \beta \\ \gamma \end{pmatrix} \sim \begin{pmatrix} a'_x & b'_x & c'_x \\ a'_y & b'_y & c'_y \\ a'_w & b'_w & c'_w \end{pmatrix}^{-1} \begin{pmatrix} x \\ y \\ 1 \end{pmatrix}$$

- Notes:
  - matrix is inverted once per triangle
  - Do not need to do this per pixel, at least..
  - Do not need to explicitly compute barycentrics
  - can be used to interpolate z, color, texture coordinates, etc.

# Pseudocode – Rasterization

---

```
For every triangle
 ComputeProjection
 Compute interpolation matrix (a', b', c' inverse)
 Compute bbox, clip bbox to screen limits
 For all pixels x,y in bbox
 Test edge functions
 If all Ei>0
 compute barycentrics
 interpolate z from vertices
 if z < zbuffer[x,y]
 interpolate UV coordinates from vertices
 look up texture color kd
 Framebuffer[x,y] = kd //or more complex shader
```



# Pseudocode – Rasterization

---

```
For every triangle
 ComputeProjection
 Compute interpolation matrix
 Compute bbox, clip bbox to screen limits
 For all pixels x,y in bbox
 Test edge functions
 If all $E_i > 0$
 compute barycentrics
 interpolate z from vertices
 if $z < zbuffer[x,y]$
 interpolate UV coordinates from vertices
 look up texture color k_d
 Framebuffer[x,y] = k_d //or more complex shader
```

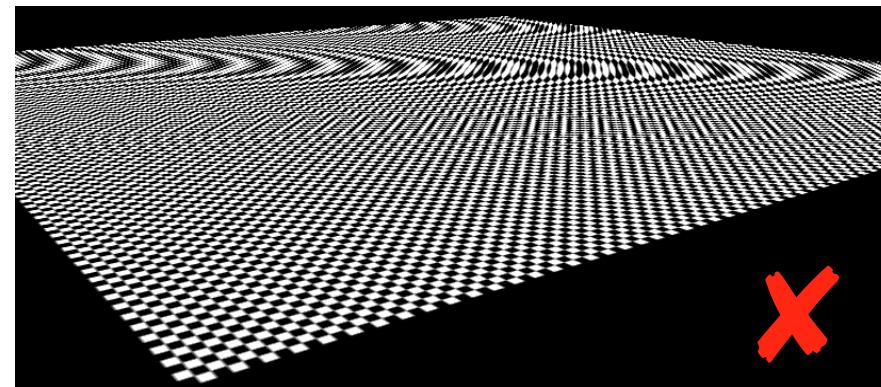


## Questions?

# Supersampling

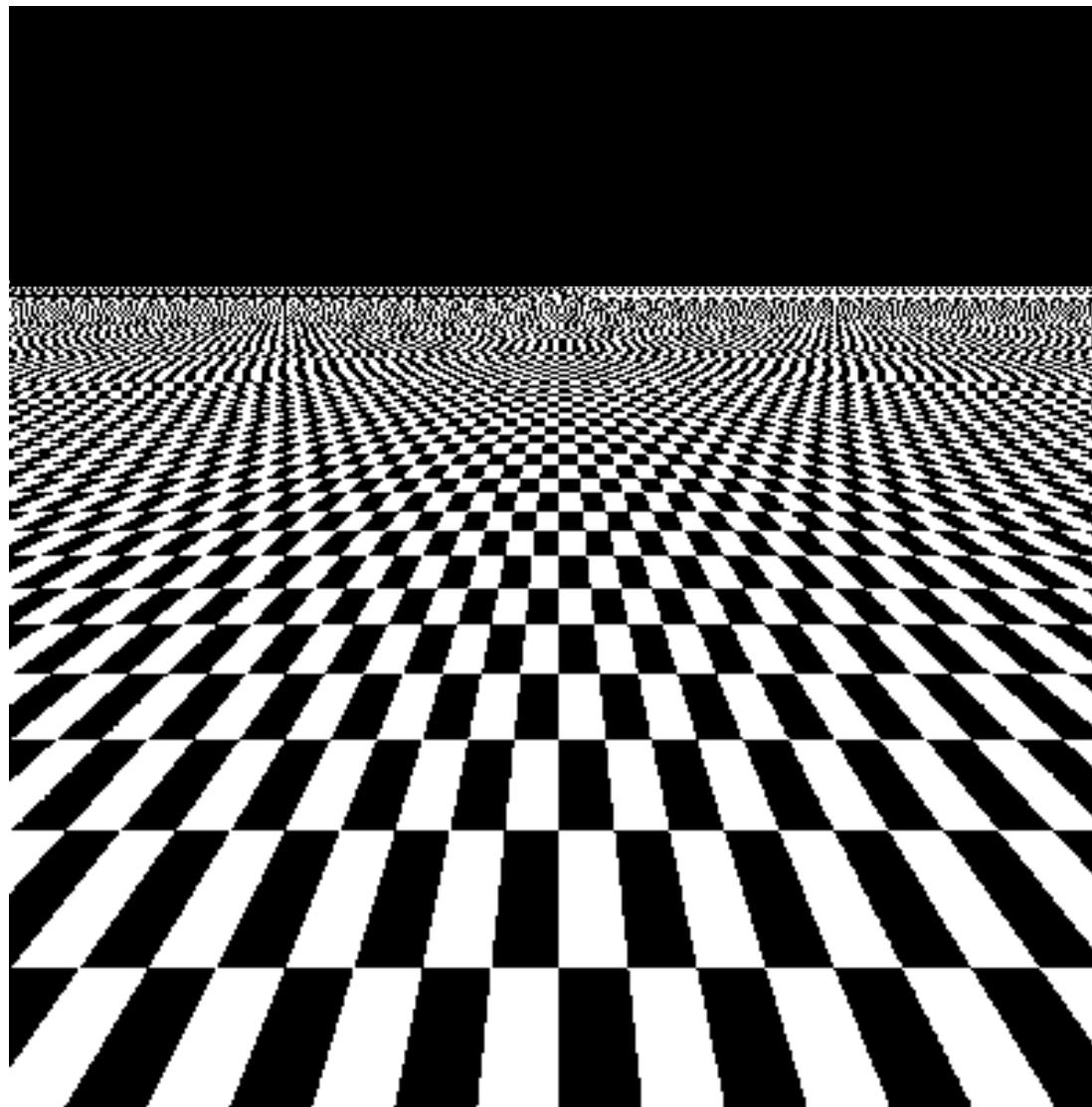
---

- Trivial to do with rasterization as well
- Often rates of 2x to 8x
- Requires to compute per-pixel average at the end
- Most effective against edge jaggies
- Usually with jittered sampling
  - pre-computed jittering pattern for a big block of pixels



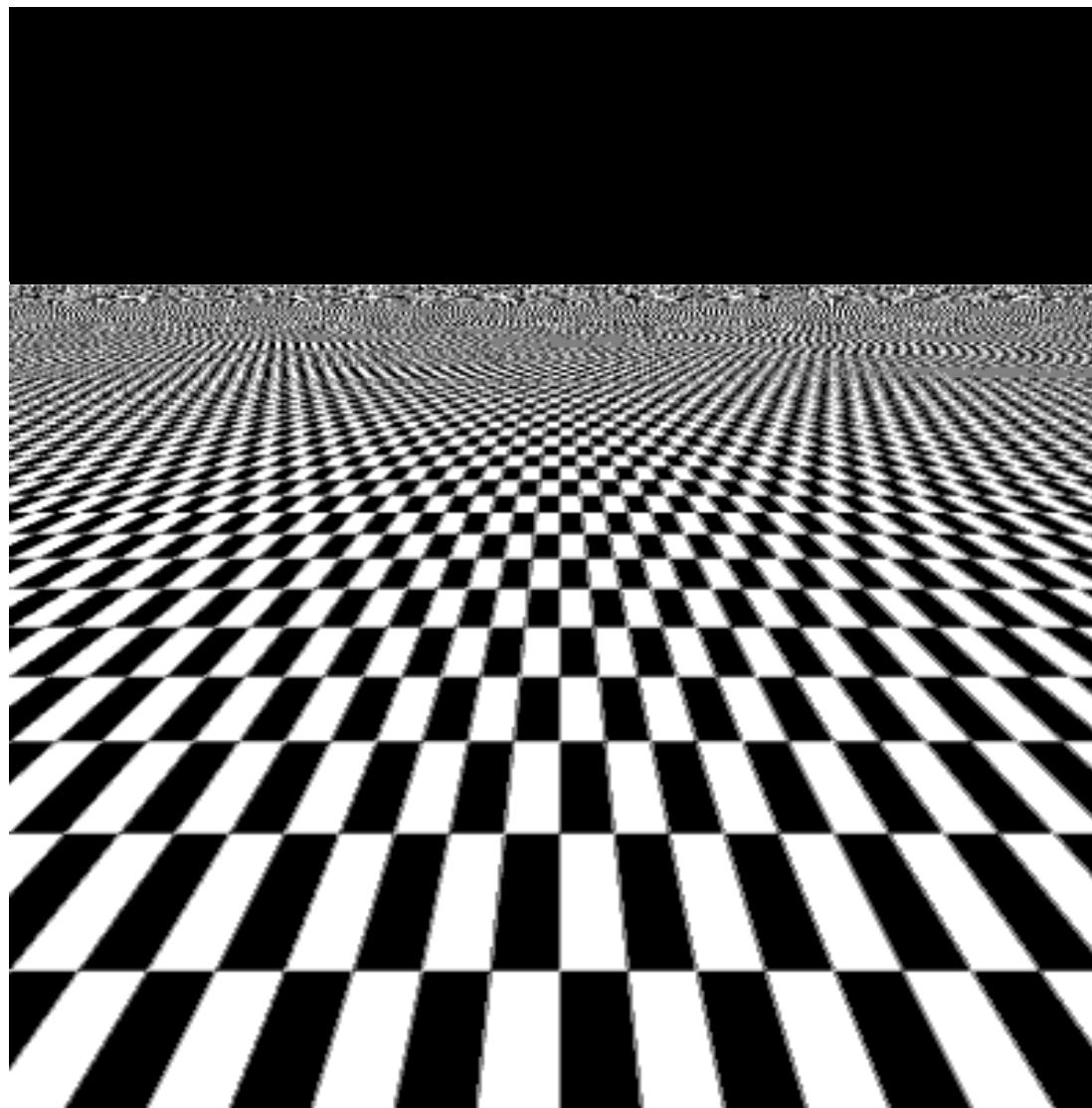
# 1 Sample / Pixel

---



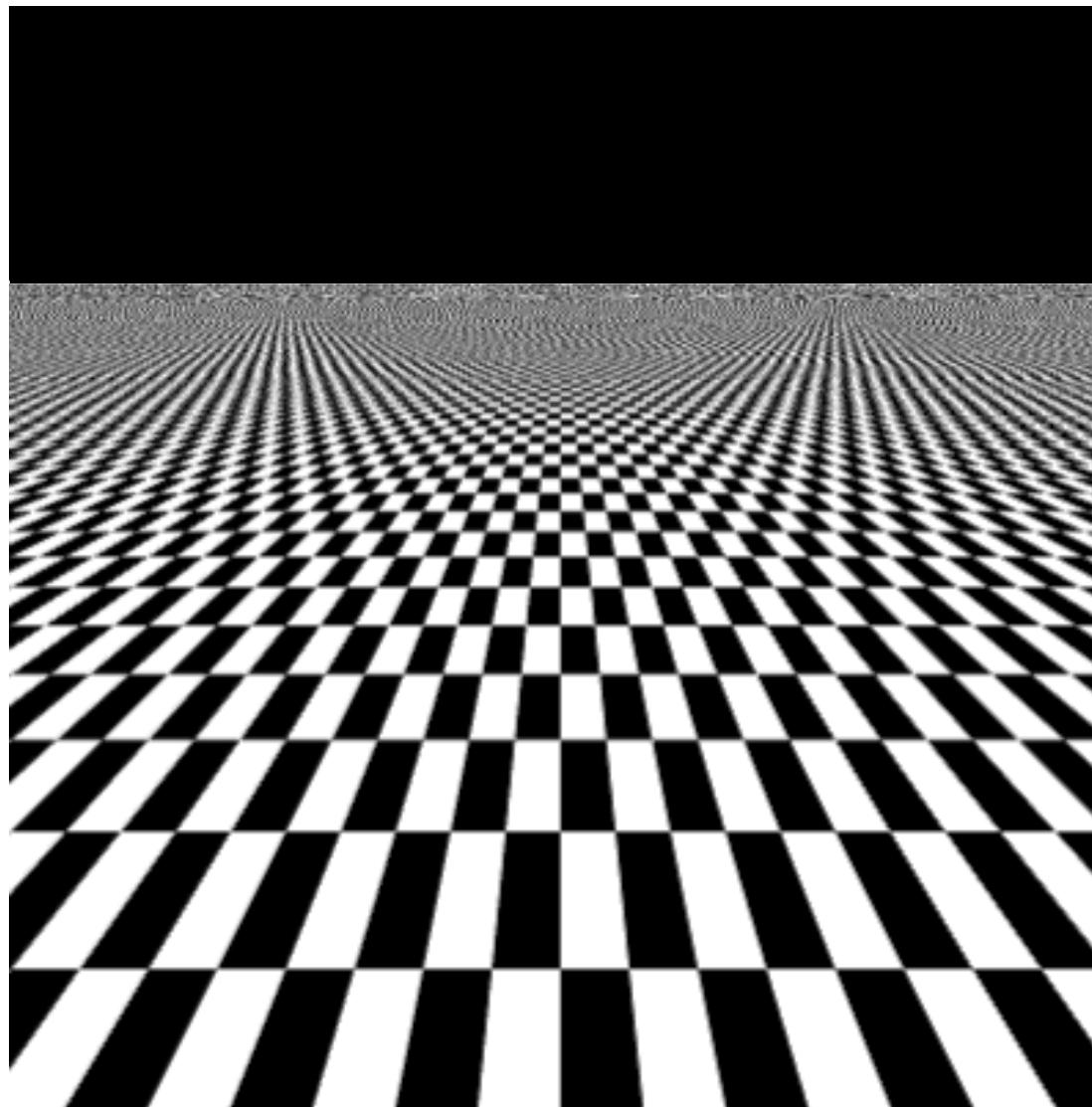
# 4 Samples / Pixel

---



# 16 Samples / Pixel

---

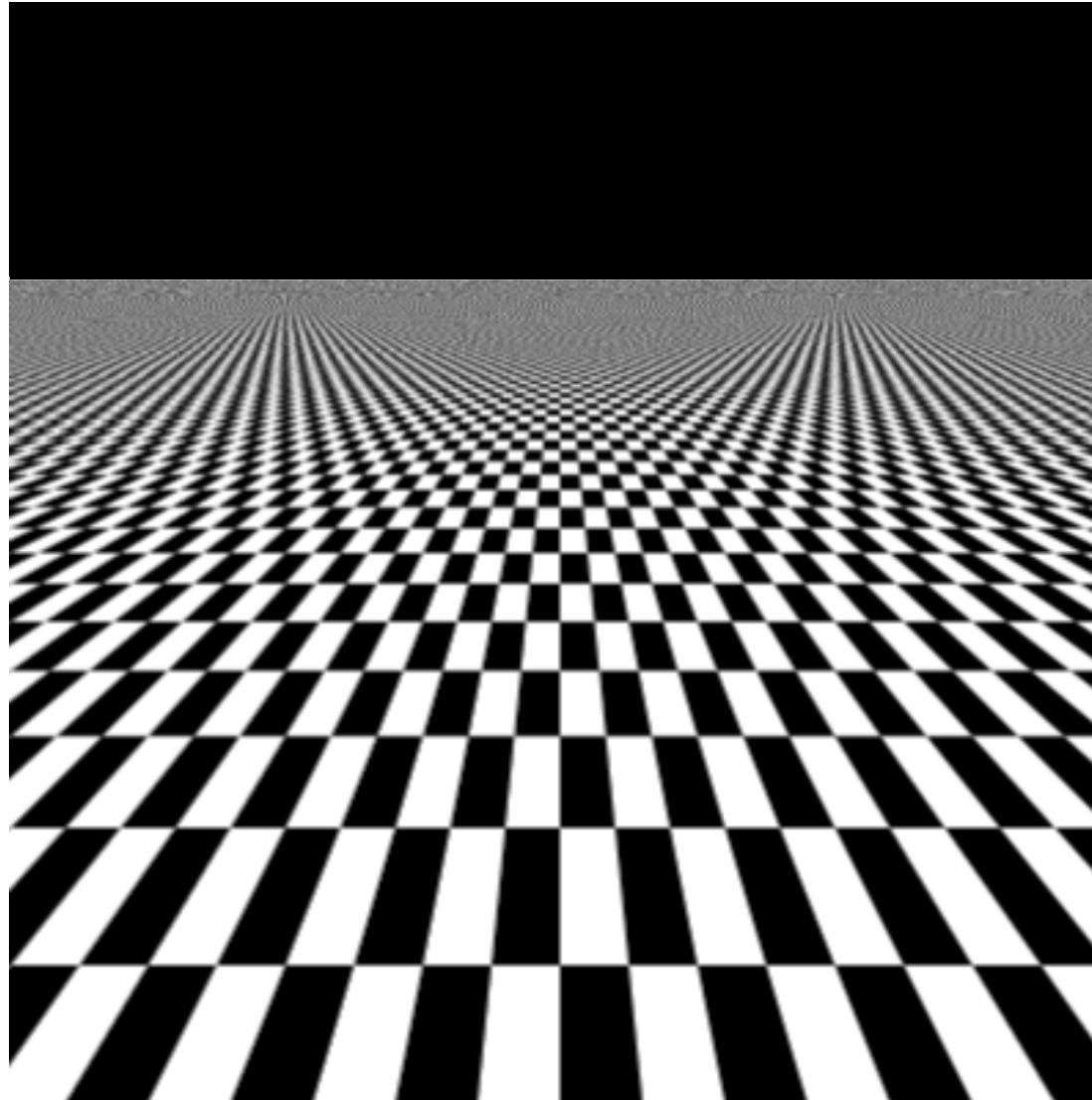


# 100 Samples / Pixel

---

Even this sampling rate cannot get rid of all aliasing artifacts!

We are really only pushing the problem farther.



# Related Idea: Multisampling

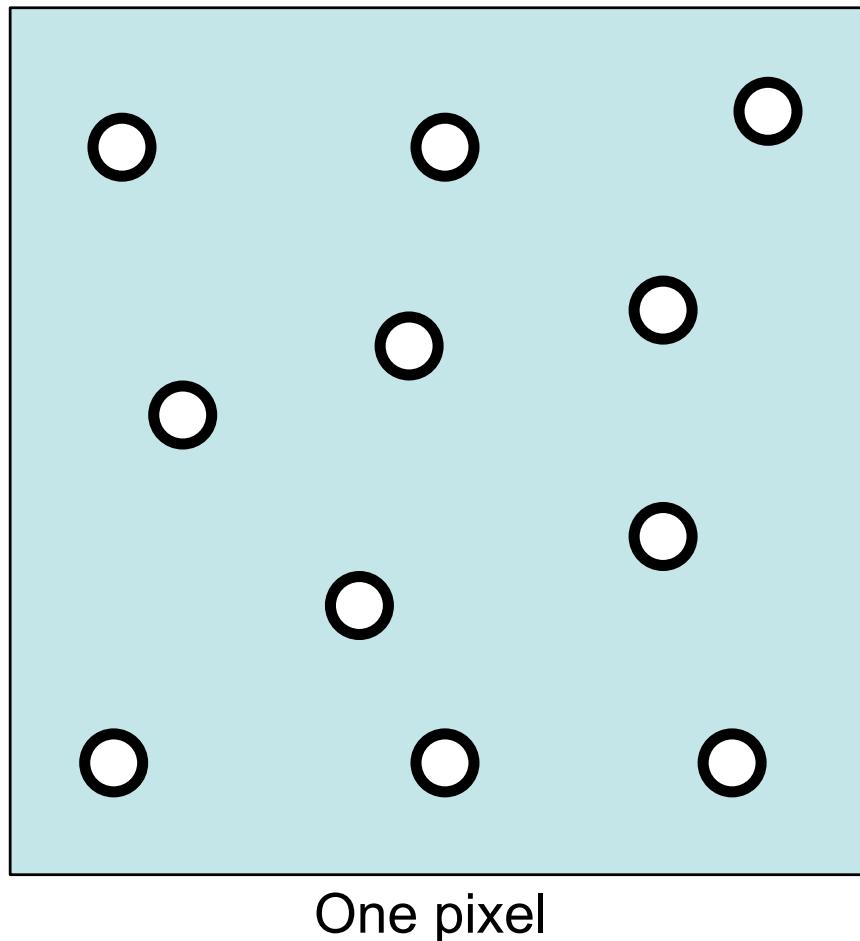
---

- Problem
  - Shading is very expensive today (complicated shaders)
  - Full supersampling has linear cost in #samples ( $k^2$ )
- Goal: High-quality edge antialiasing at lower cost
- Solution
  - Compute shading only once per pixel for each primitive, but resolve visibility at “sub-pixel” level
    - Store ( $k^2$  width,  $k^2$  height) frame and z buffers, but share shading results between sub-pixels within a real pixel
  - When visibility samples within a pixel hit different primitives, we get an average of their colors
    - Edges get antialiased without large shading cost

# Multisampling, Visually

---

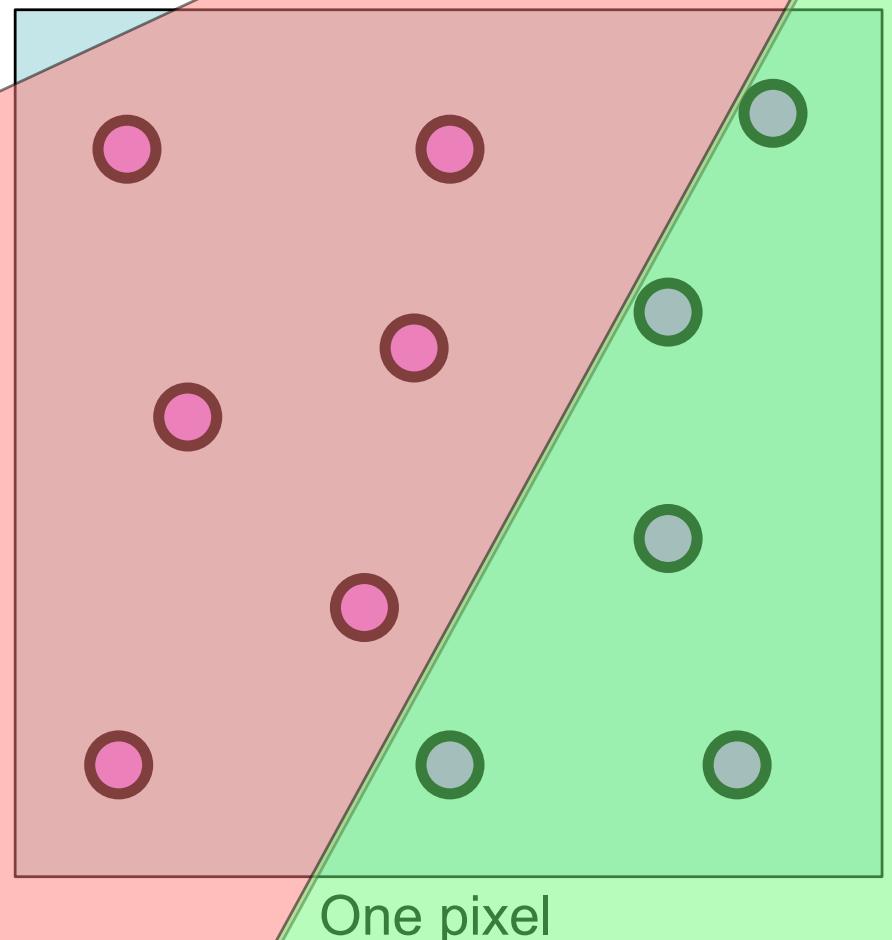
○ = sub-pixel visibility sample



# Multisampling, Visually

---

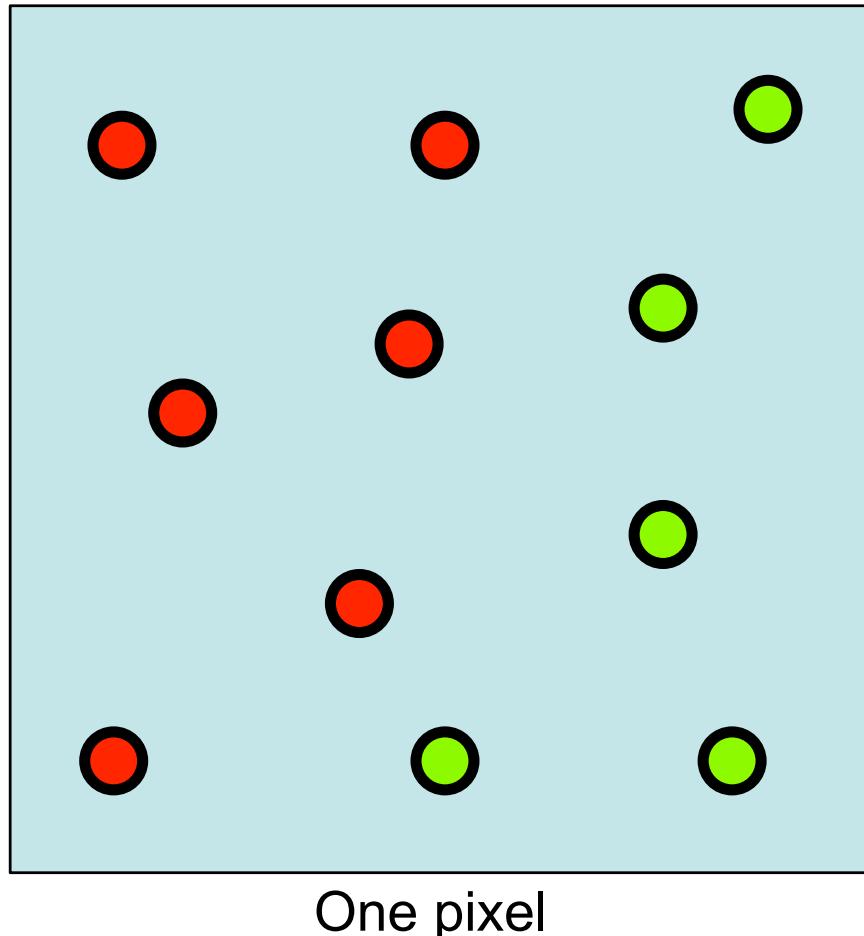
○ = sub-pixel visibility sample



# Multisampling, Visually

---

○ = sub-pixel visibility sample

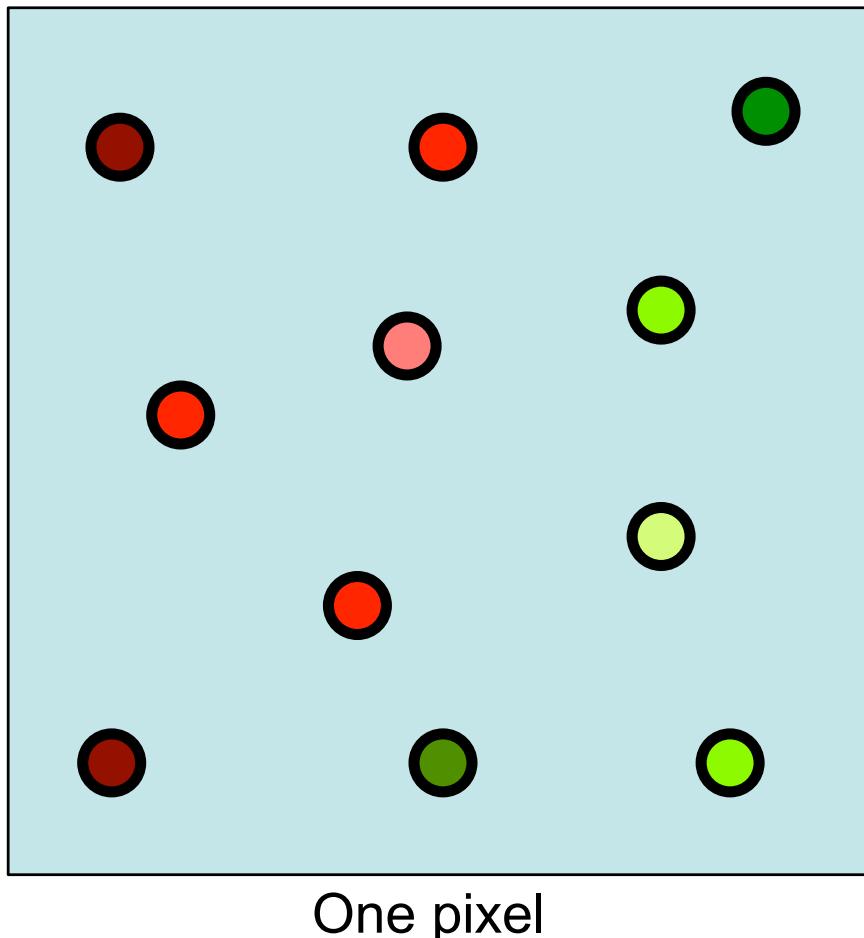


The color is only computed **once per pixel per triangle** and reused for all the visibility samples that are covered by the triangle.

# Supersampling, Visually

---

○ = sub-pixel visibility sample



When supersampling, we compute colors independently for all the visibility samples.

# Multisampling Pseudocode

---

For each triangle

    For each pixel

        if pixel overlaps triangle

            color=shade() // only once per pixel!

            for each sub-pixel sample

                compute edge equations & z

                if subsample passes edge equations

                    && z < zbuffer[subsample]

                    zbuffer[subsample]=z

                    framebuffer[subsample]=color

# Multisampling Pseudocode

---

For each triangle

    For each pixel

        if pixel overlaps triangle

            color=shade() // only once per pixel!

            for each sub-pixel sample

                compute edge equations & z

                if subsample passes edge equations

                    && z < zbuffer[subsample]

                    zbuffer[subsample]=z

                    framebuffer[subsample]=color

At display time: //this is called “resolving”

    For each pixel

        color = average of subsamples

# Multisampling vs. Supersampling

---

- Supersampling
  - Compute an entire image at a higher resolution, then downsample (blur + resample at lower res)
- Multisampling
  - Supersample visibility, compute expensive shading only once per pixel, reuse shading across visibility samples
- This is how GPUs perform antialiasing these days

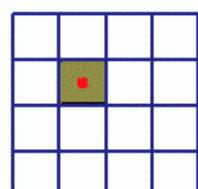
# Examples of Texture Aliasing

---

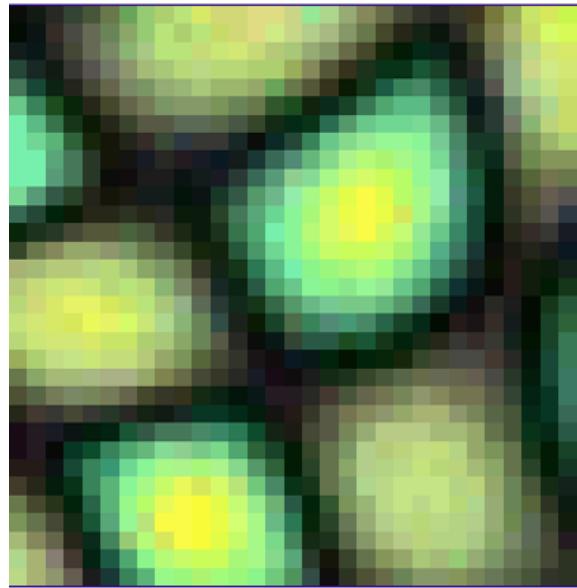
Magnification

|    |    |    |    |    |
|----|----|----|----|----|
| 21 | 22 | 23 | 24 | 25 |
| 16 | 17 | 18 | 19 | 20 |
| 11 | 12 | 13 | 14 | 15 |
| 6  | 7  | 8  | 9  | 10 |
| 1  | 2  | 3  | 4  | 5  |

Texture



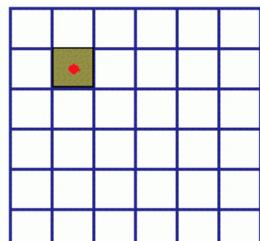
Polygon



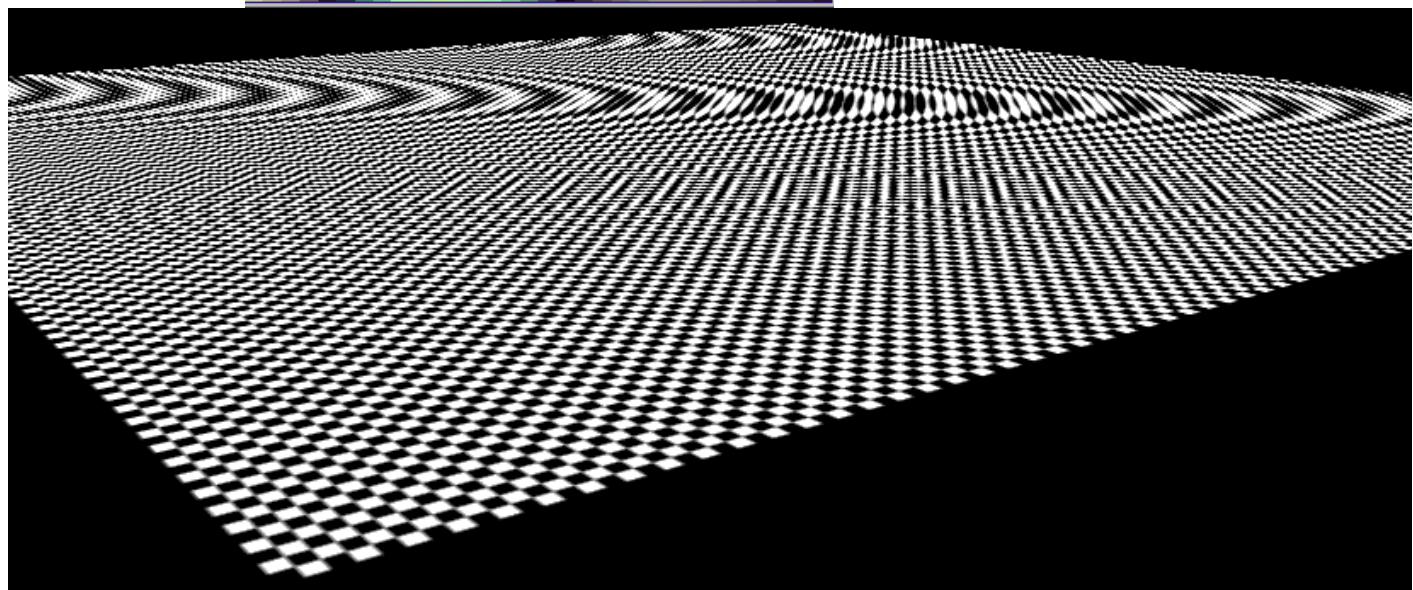
Minification

|    |    |    |    |
|----|----|----|----|
| 13 | 14 | 15 | 16 |
| 9  | 10 | 11 | 12 |
| 5  | 6  | 7  | 8  |
| 1  | 2  | 3  | 4  |

Texture



Polygon



# Texture Filtering

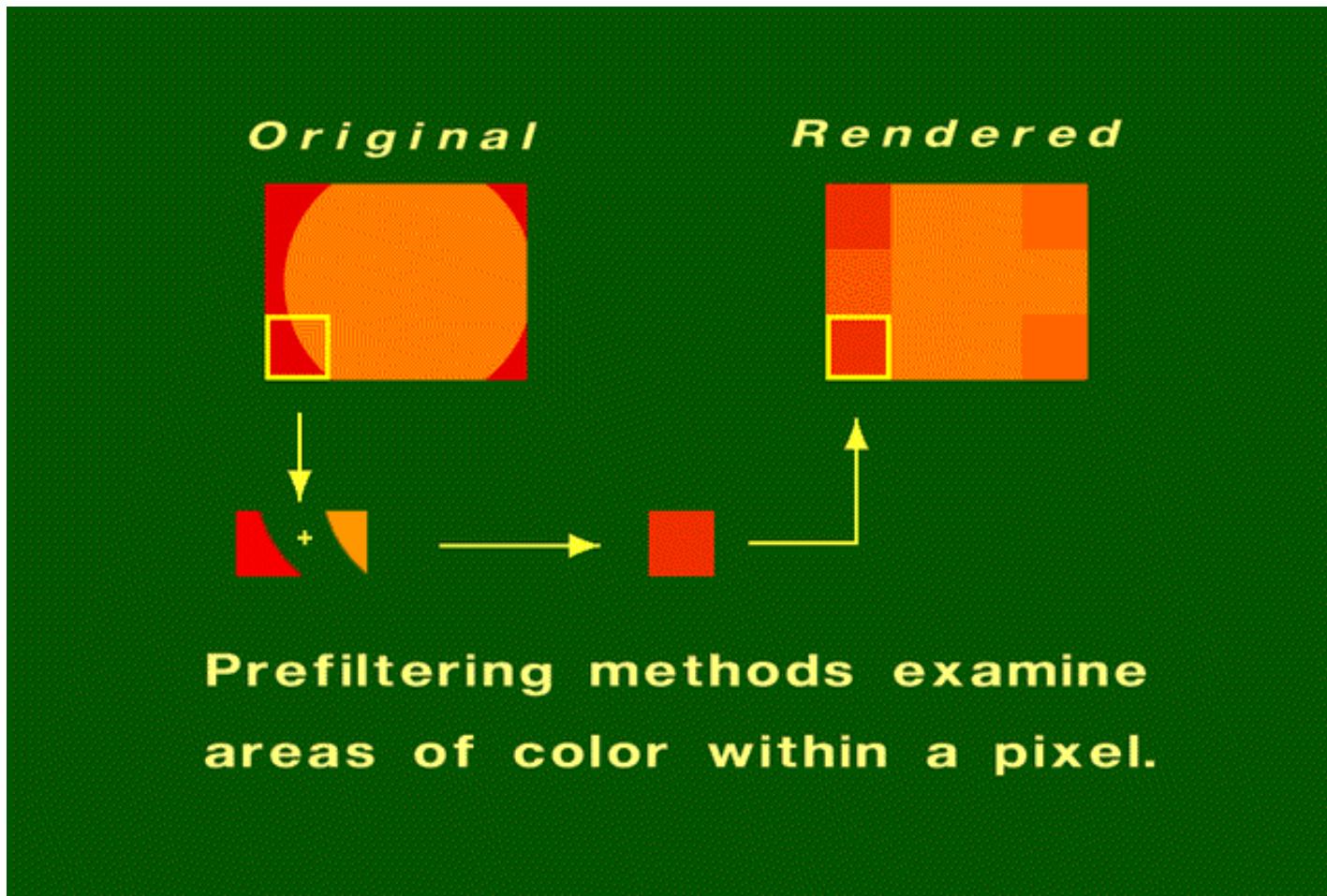
---

- General scene primitives
  - Post-filtering: supersampling, multisampling
  - Filter after sampling
- Texture
  - Pre-filtering
  - Filter before sampling

# Pre-filtering

---

- Treats a pixel as an area



# Pre-filtering

---

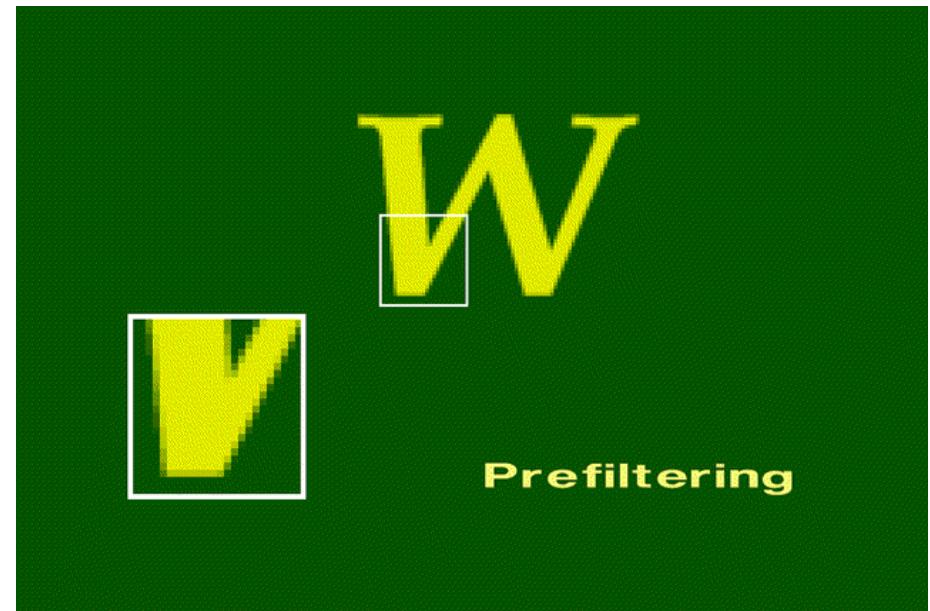
- Treats a pixel as an area



# Pre-filtering

---

- Treats a pixel as an area



# Post-filtering

---

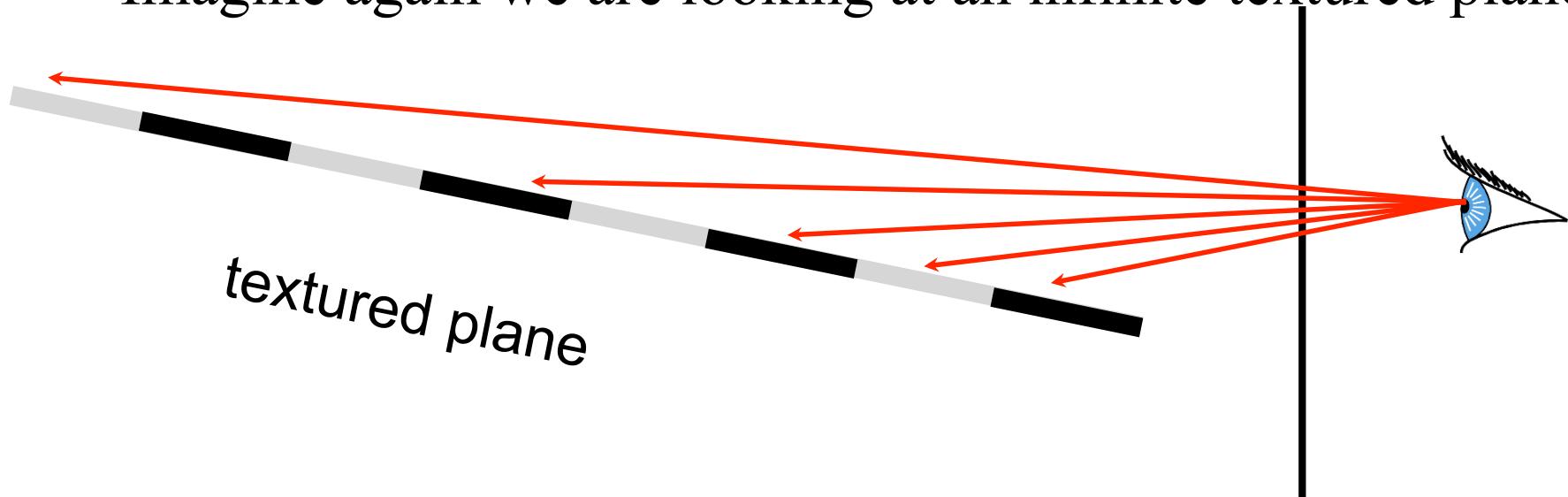
- Supersampling, multisampling
  - Sample the scene at n times the display resolution
  - The color of each pixel in the rendered image will be an average of several samples



# Texture Filtering

---

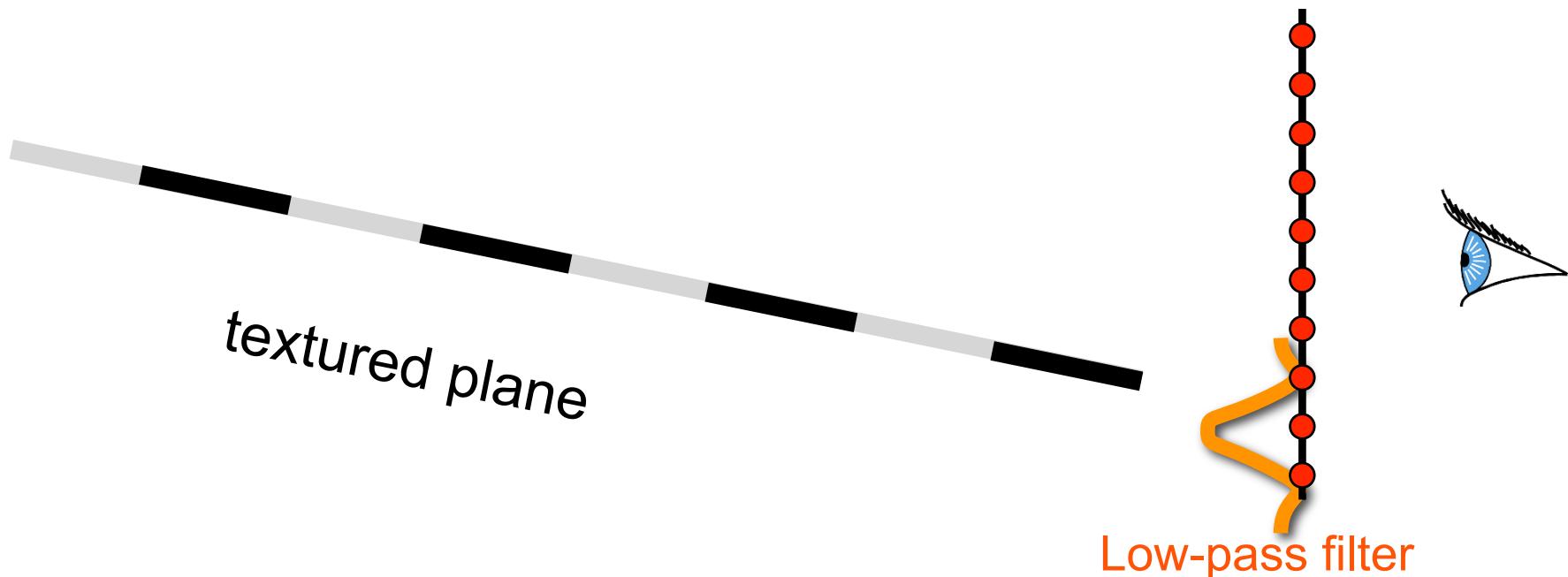
- Problem: Prefiltering is impossible when you can only take point samples
- Texture mapping is simpler
  - Imagine again we are looking at an infinite textured plane



# Texture Filtering

---

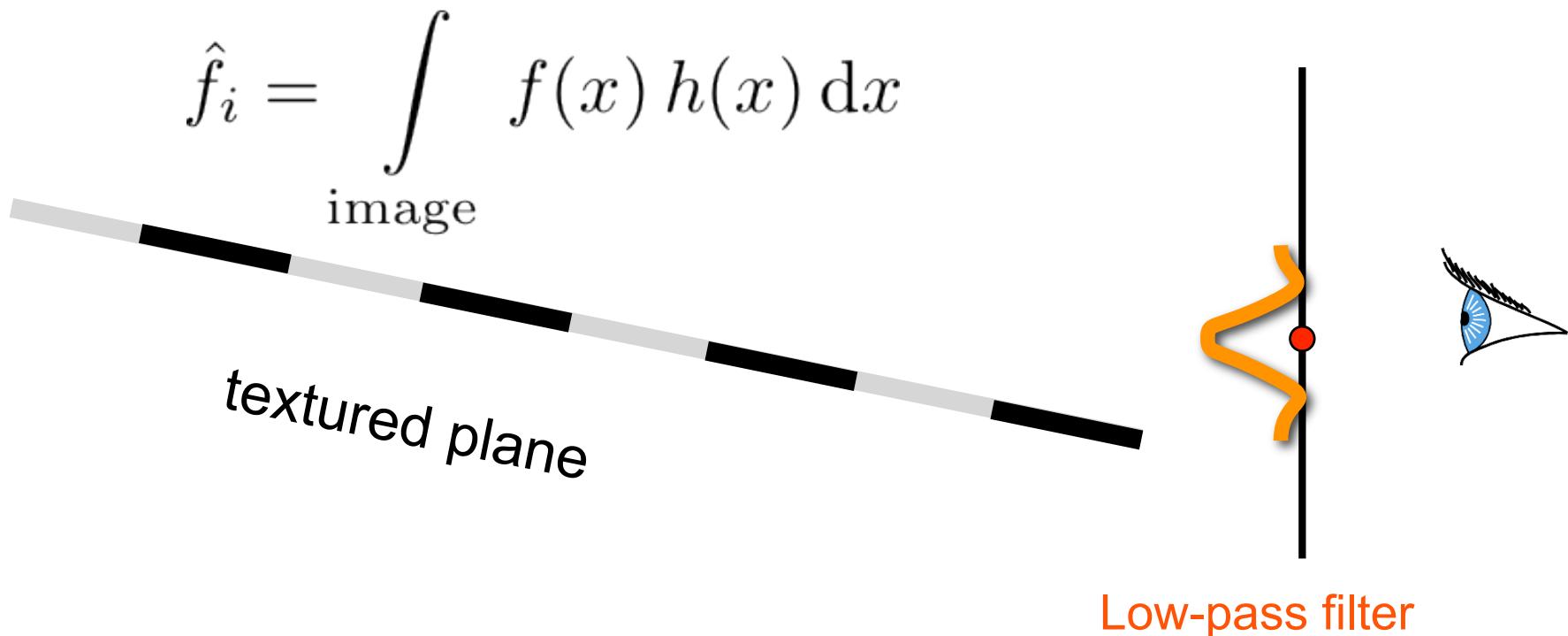
- We should pre-filter image function before sampling
  - That means blurring the image function with a low-pass filter (convolution of image function and filter)



# Texture Filtering

---

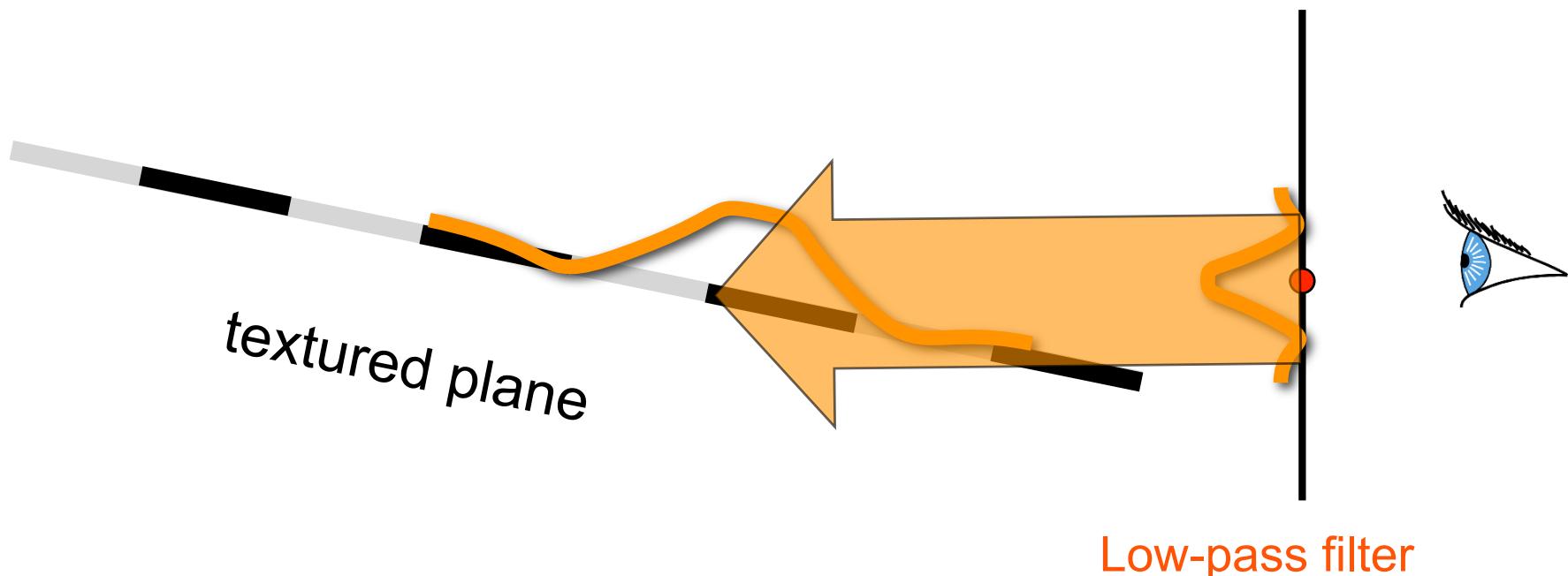
- We can combine low-pass and sampling
  - The value of a sample is the integral of the product of the image  $f$  and the filter  $h$  centered at the sample location
    - “A local average of the image  $f$  weighted by the filter  $h$ ”



# Texture Filtering

---

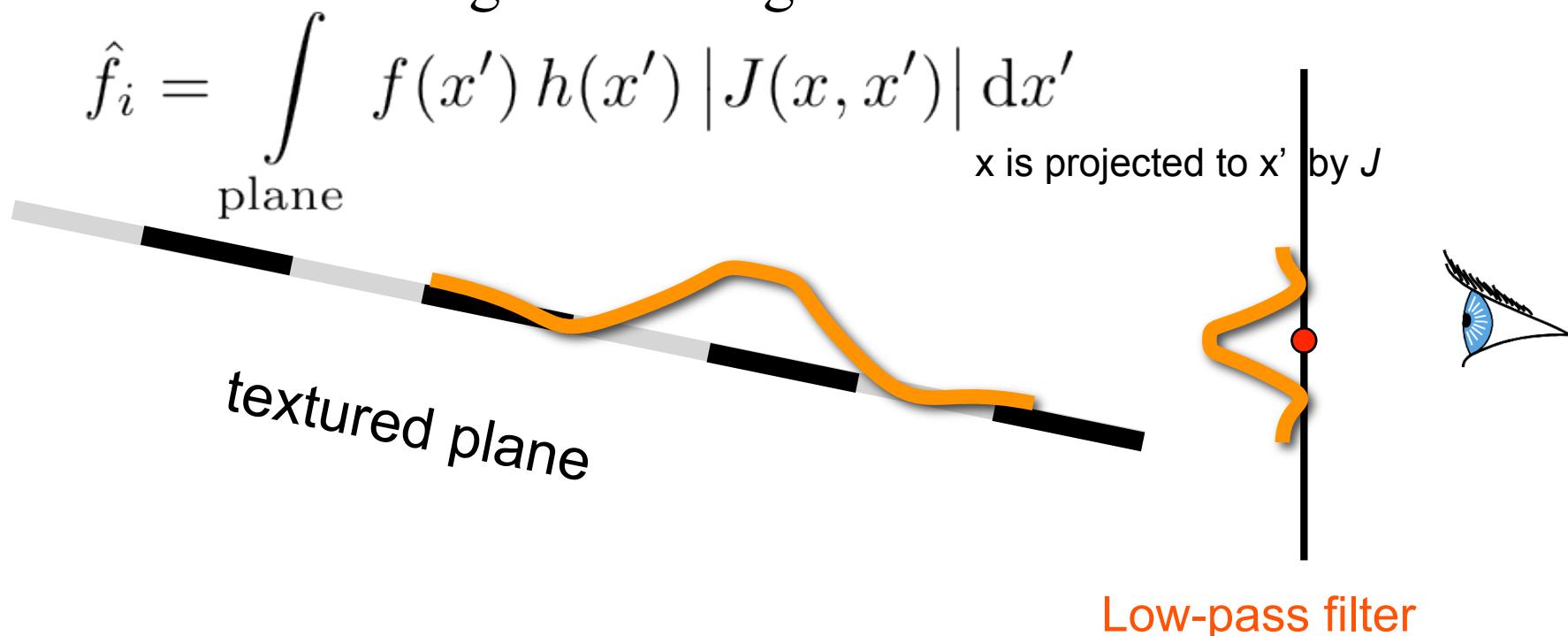
- Well, we can just as well change variables and compute this integral on the textured plane instead
  - In effect, we are projecting the pre-filter onto the plane



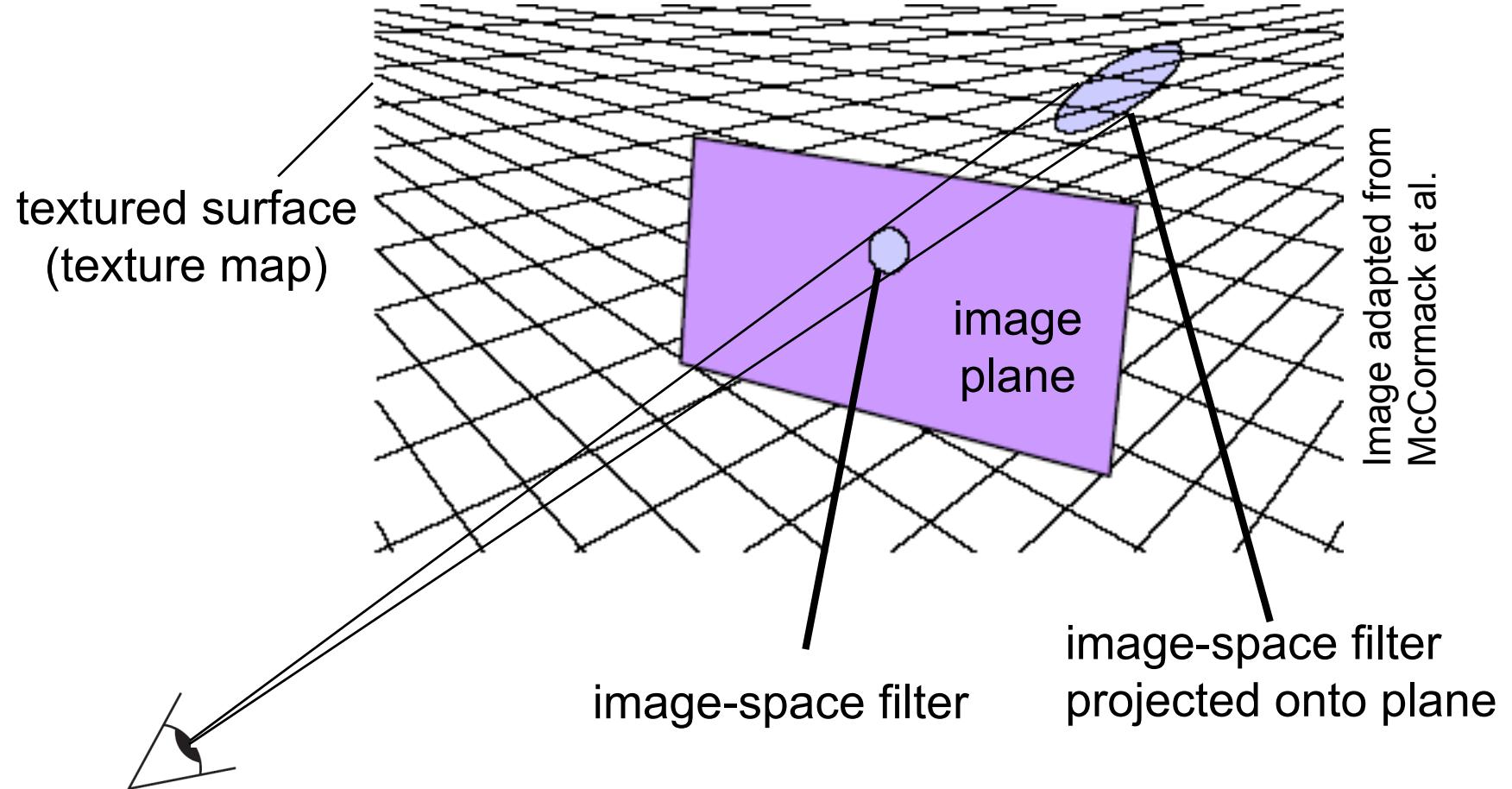
# Texture Filtering

---

- Well, we can just as well change variables and compute this integral on the textured plane instead
  - In effect, we are projecting the pre-filter onto the plane
  - It's still a weighted average of the texture under filter



# Texture Pre-Filtering, Visually

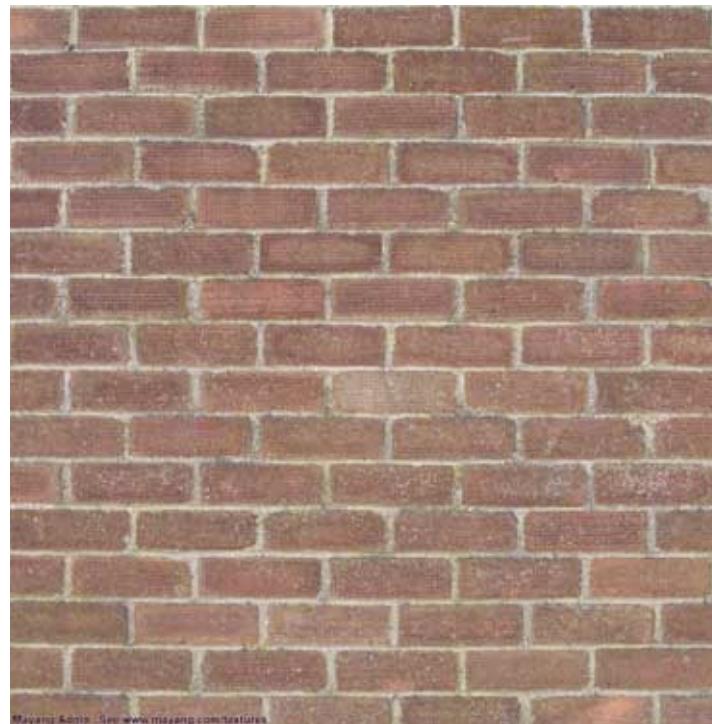


- Must still integrate product of projected filter and texture – That doesn't sound any easier...

# Solution: Precomputation

---

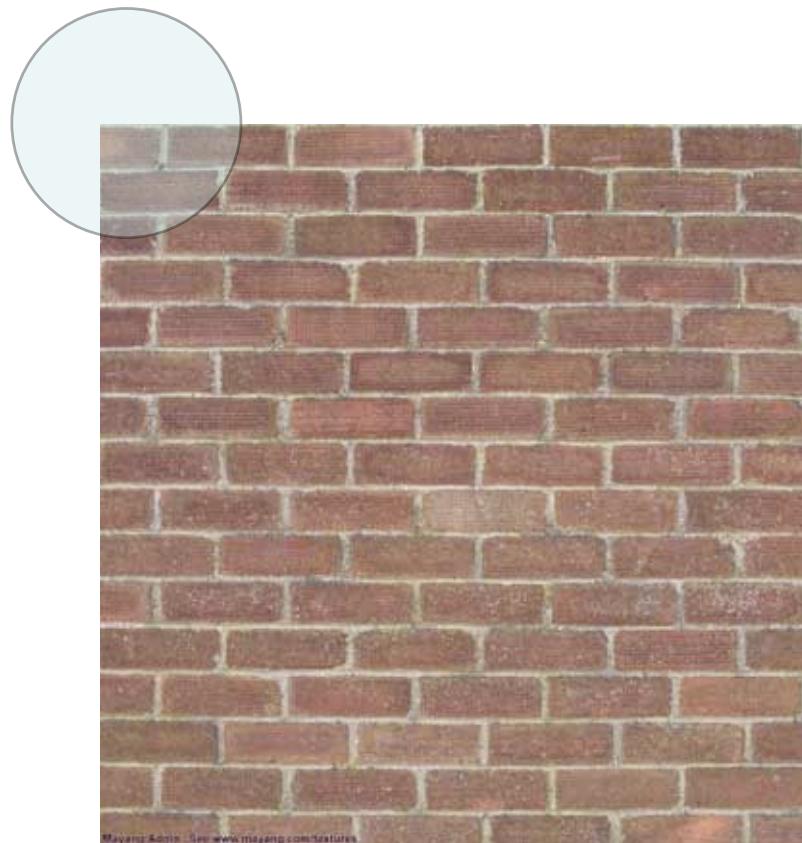
- We'll precompute and store a set of prefiltered results from each texture with different sizes of prefilters



# Solution: Precomputation

---

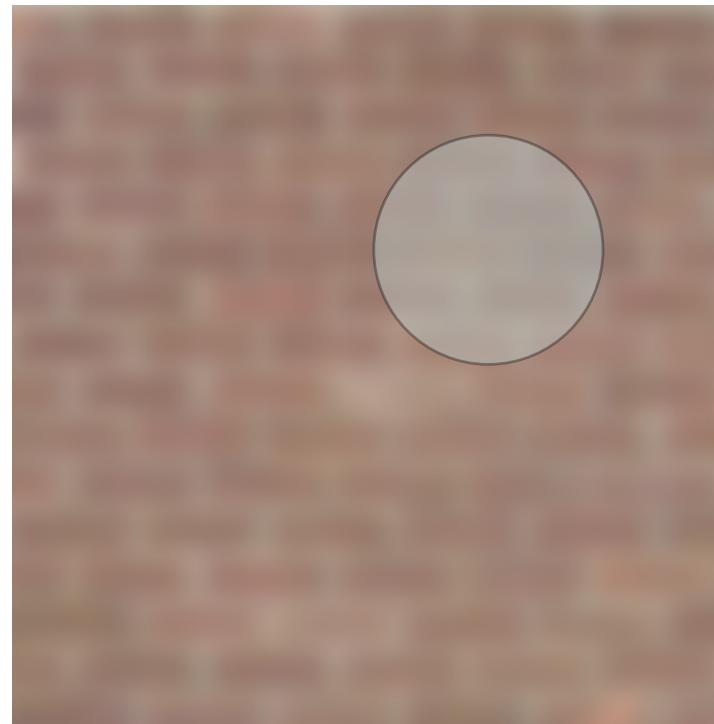
- We'll precompute and store a set of prefiltered results from each texture with different sizes of prefilters



# Solution: Precomputation

---

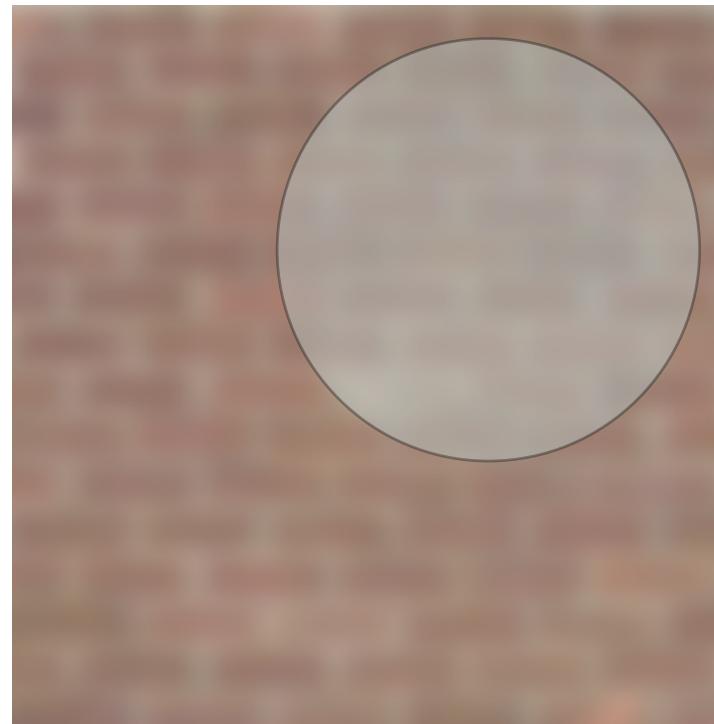
- We'll precompute and store a set of prefiltered results from each texture with different sizes of prefilters
  - Because it's low-passed, we can also subsample (why?)



# Solution: Precomputation

---

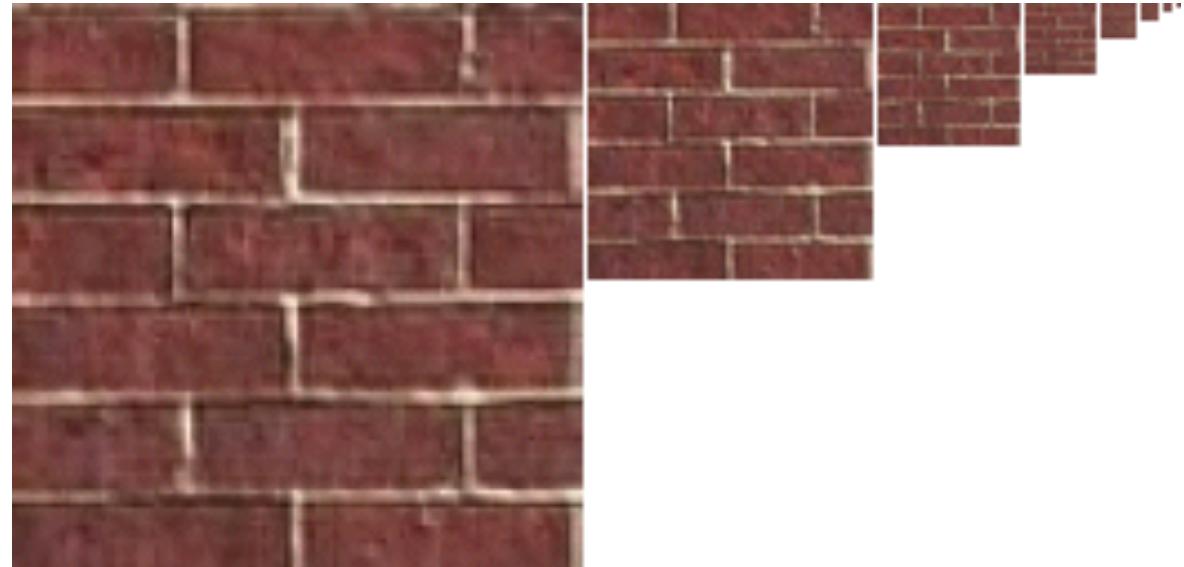
- We'll precompute and store a set of prefiltered results from each texture with different sizes of prefilters
  - Because it's low-passed, we can also subsample



# This is Called “MIP-Mapping”

---

- Construct a pyramid of images that are pre-filtered and re-sampled at  $1/2, 1/4, 1/8$ , etc., of the original image's sampling
- During rasterization we compute the index of the decimated image that is sampled at a rate closest to the density of our desired sampling rate
- MIP stands for multum in parvo which means many in a small place
- MIP-Mapping is an example of pre-filtering

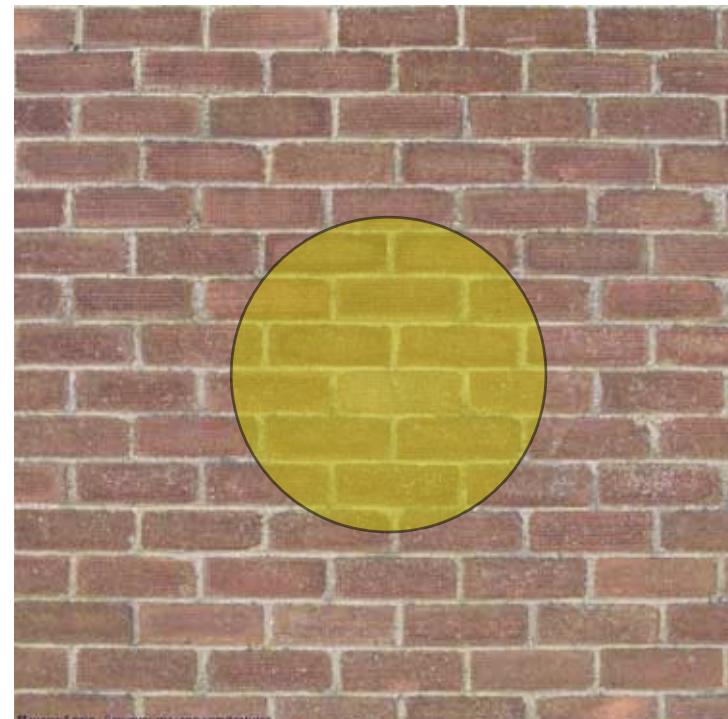


# MIP-Mapping

---

- When a pixel wants an integral of the pre-filtered texture, we must find the “closest” results from the precomputed MIP-map pyramid
  - Must compute the “size” of the projected pre-filter in the texture UV domain

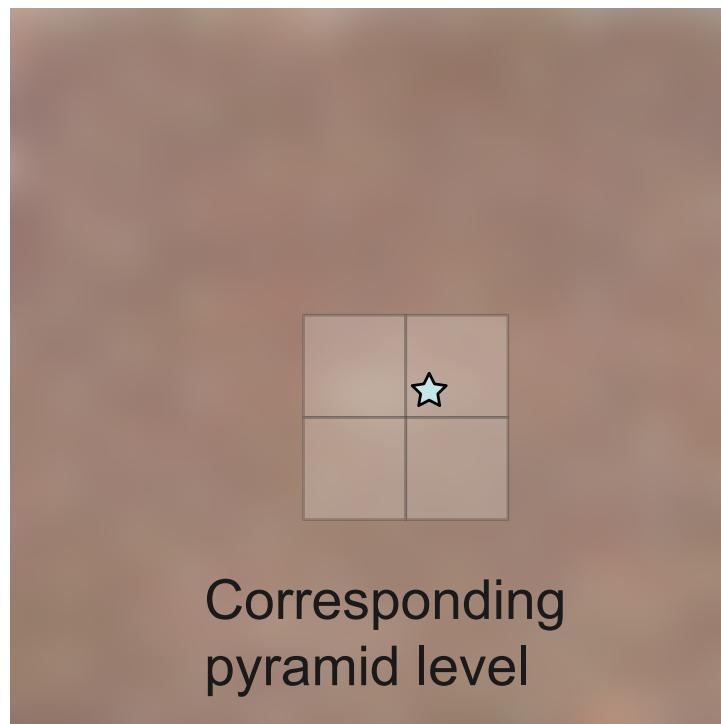
Projected pre-filter



# MIP-Mapping

---

- Simplest method: Pick the scale closest, then do usual reconstruction on that level (e.g. bilinear between 4 closest texture pixels)

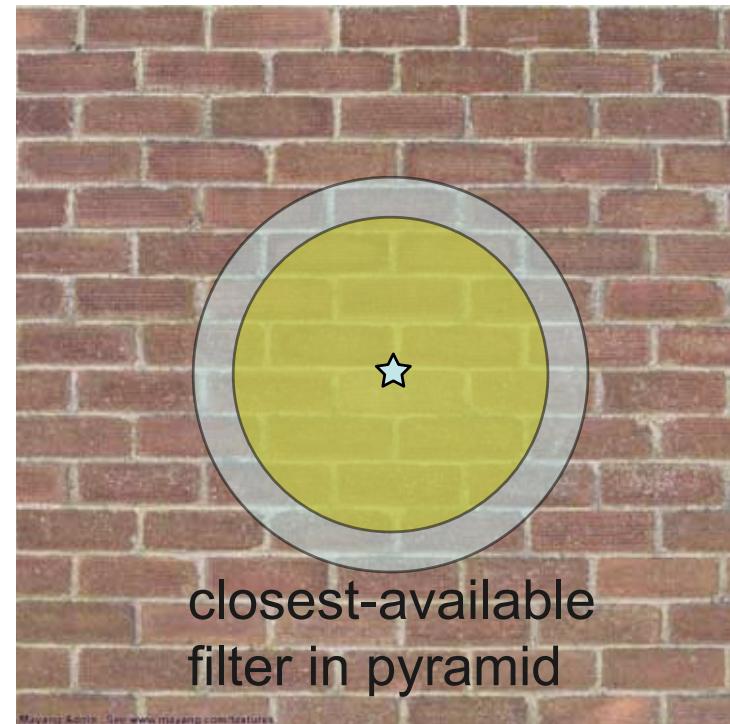
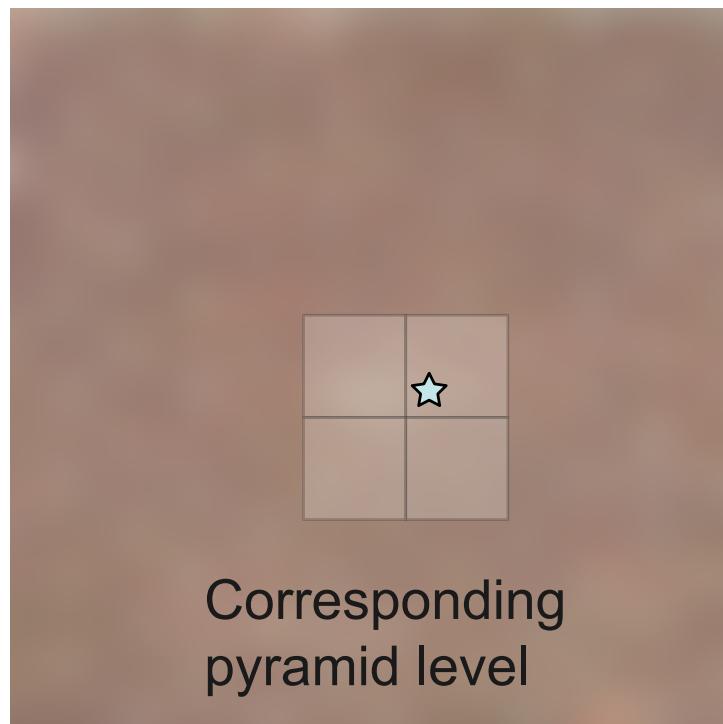


# MIP-Mapping

---

- Simplest method: Pick the scale closest, then do usual reconstruction on that level (e.g. bilinear between 4 closest texture pixels)
- Problem: discontinuity when switching scale

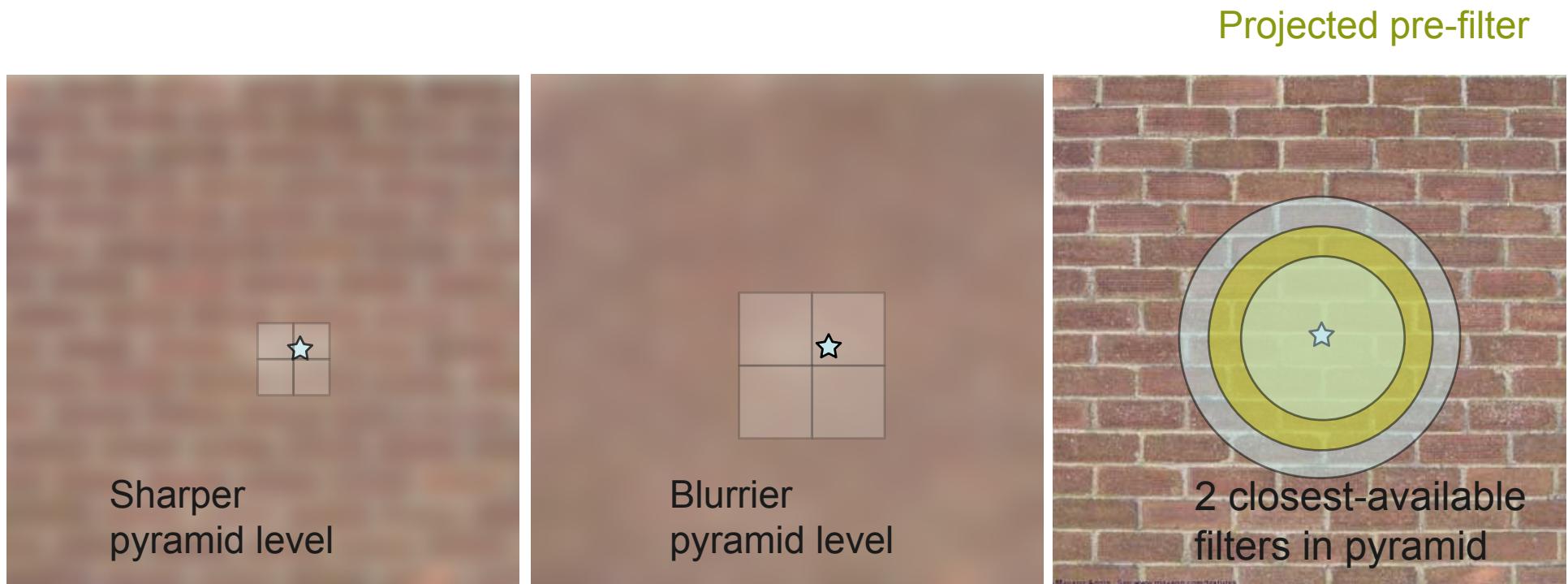
Projected pre-filter



# Tri-Linear MIP-Mapping

---

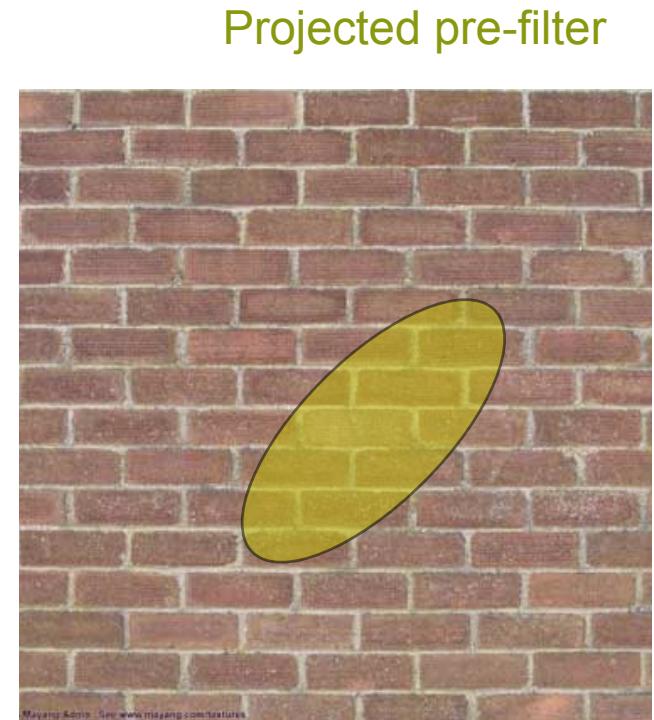
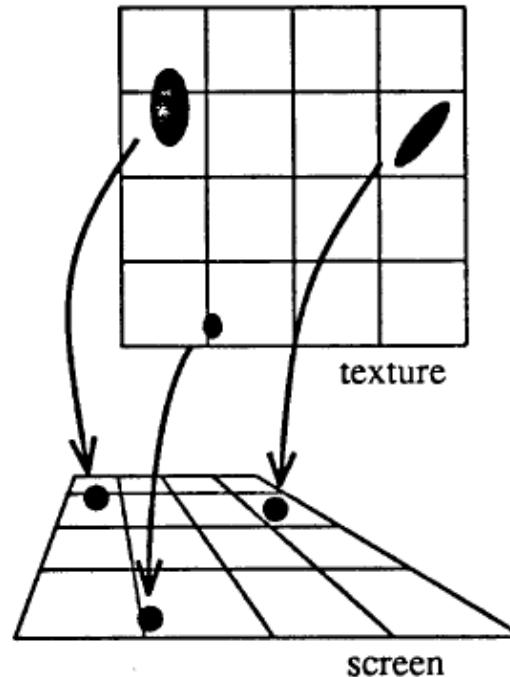
- Use two closest scales, compute reconstruction results from both, and linearly interpolate between them (8 values)



# Tri-Linear MIP-Mapping

---

- Use two closest scales, compute reconstruction results from both, and linearly interpolate between them
- Problem: our filter might not be circular, because of foreshortening

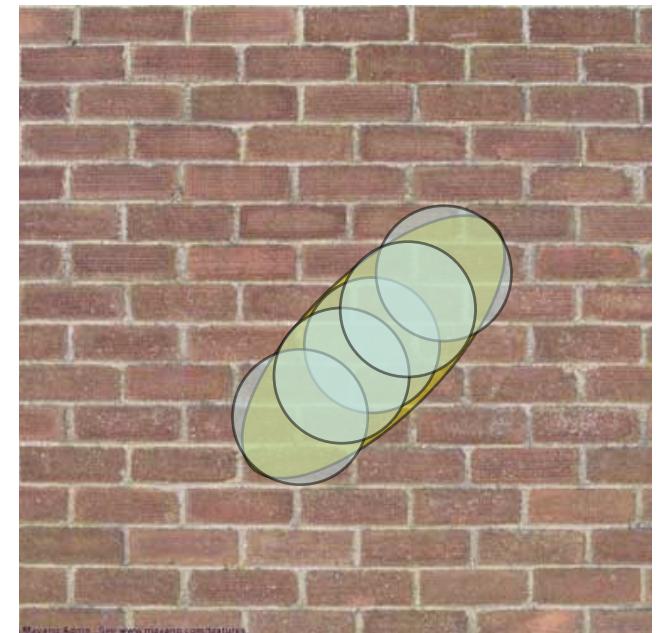


# Anisotropic filtering

---

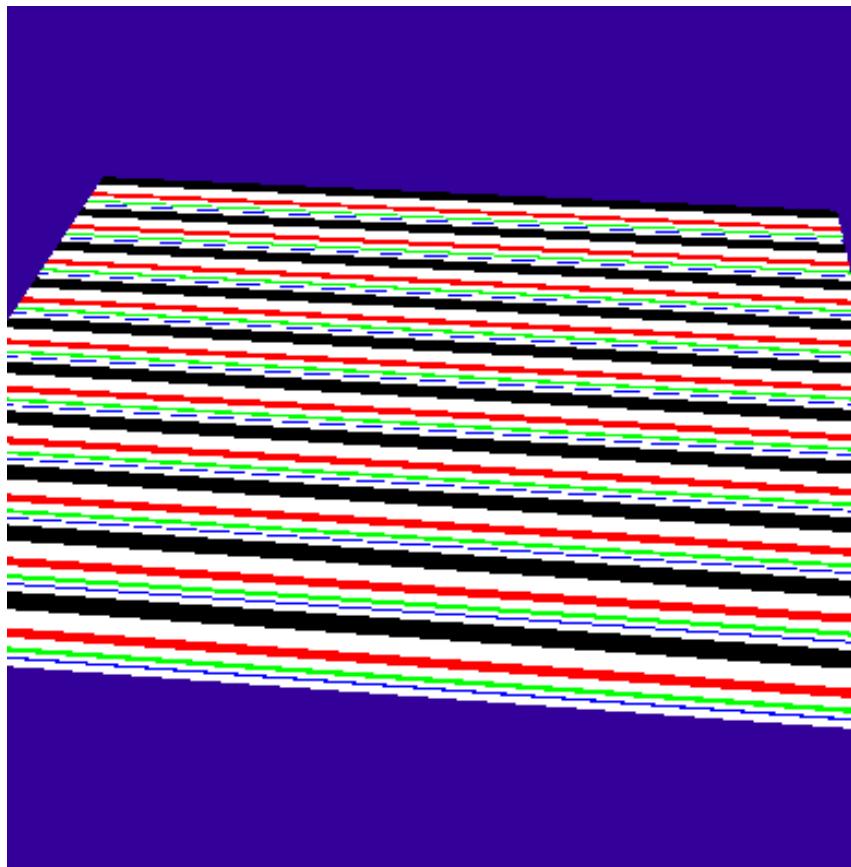
- Approximate Elliptical filter with multiple circular ones (usually 5)
- Perform trilinear lookup at each one
- i.e. consider five times eight values (why?)
  - fair amount of computation
  - this is why graphics hardware has dedicated units to compute trilinear mipmap reconstruction

Projected pre-filter

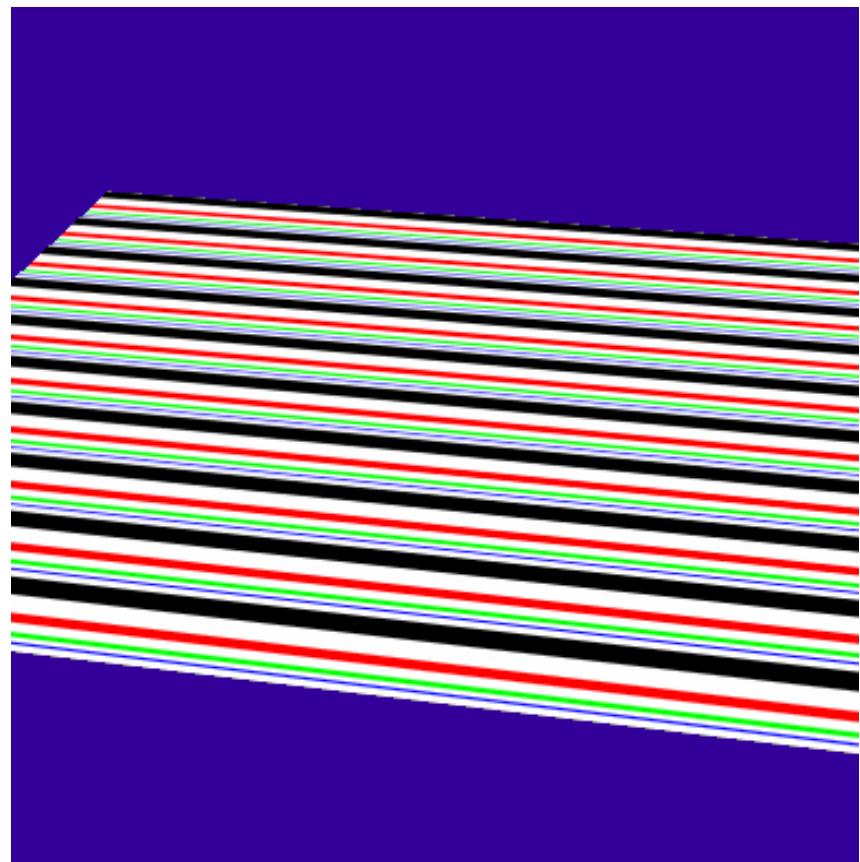


# MIP Mapping Example

---



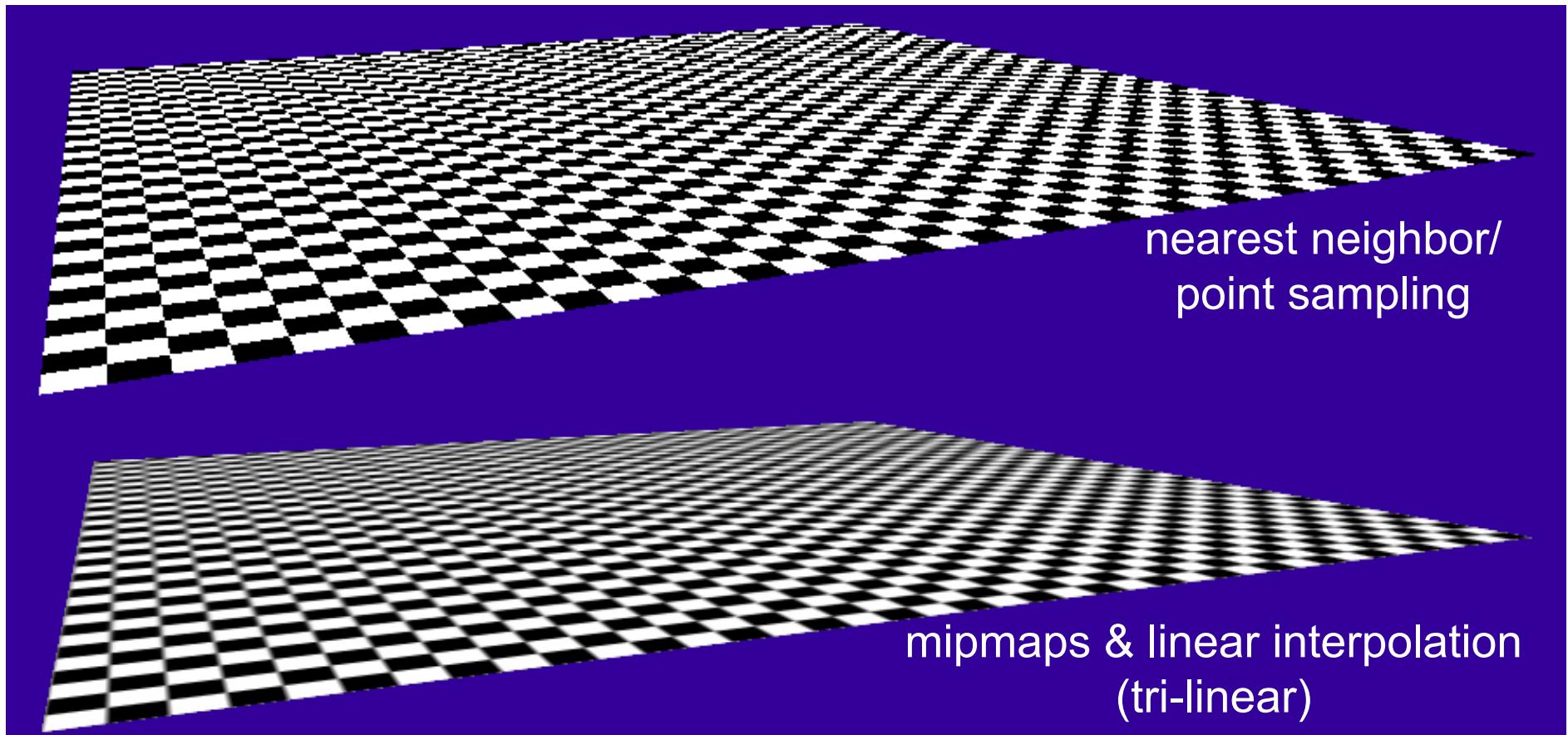
Nearest Neighbor



MIP Mapped (Tri-Linear)

# MIP Mapping Example

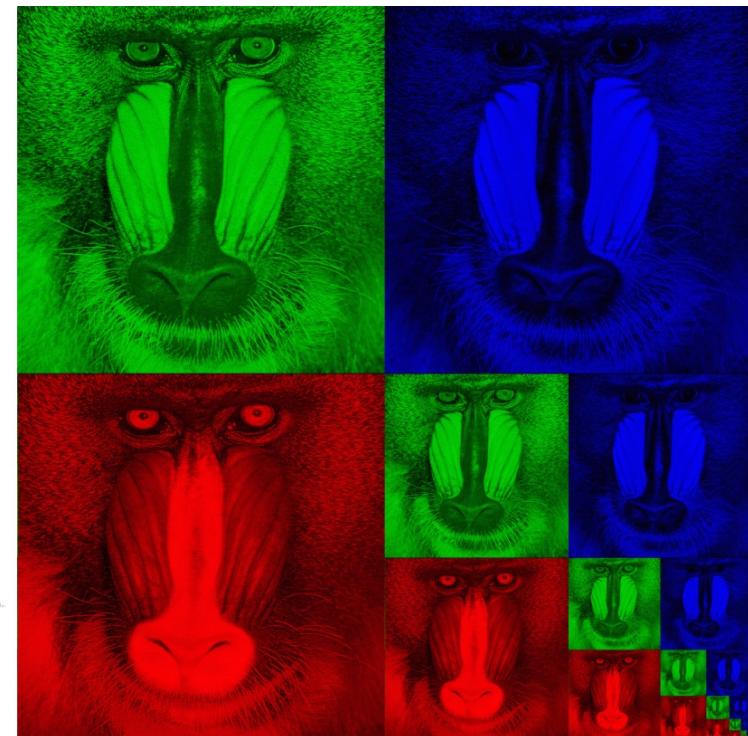
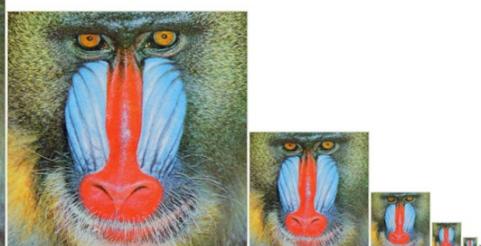
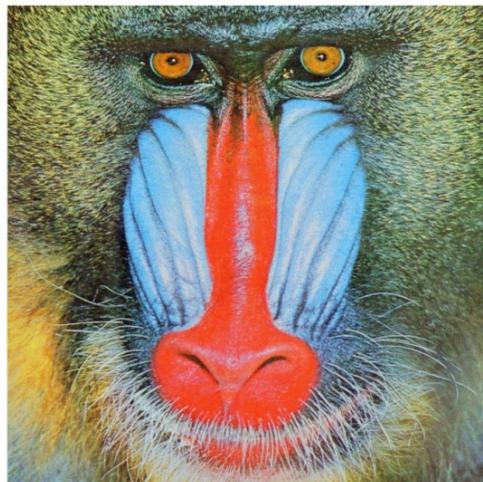
---



# Storing MIP Maps

---

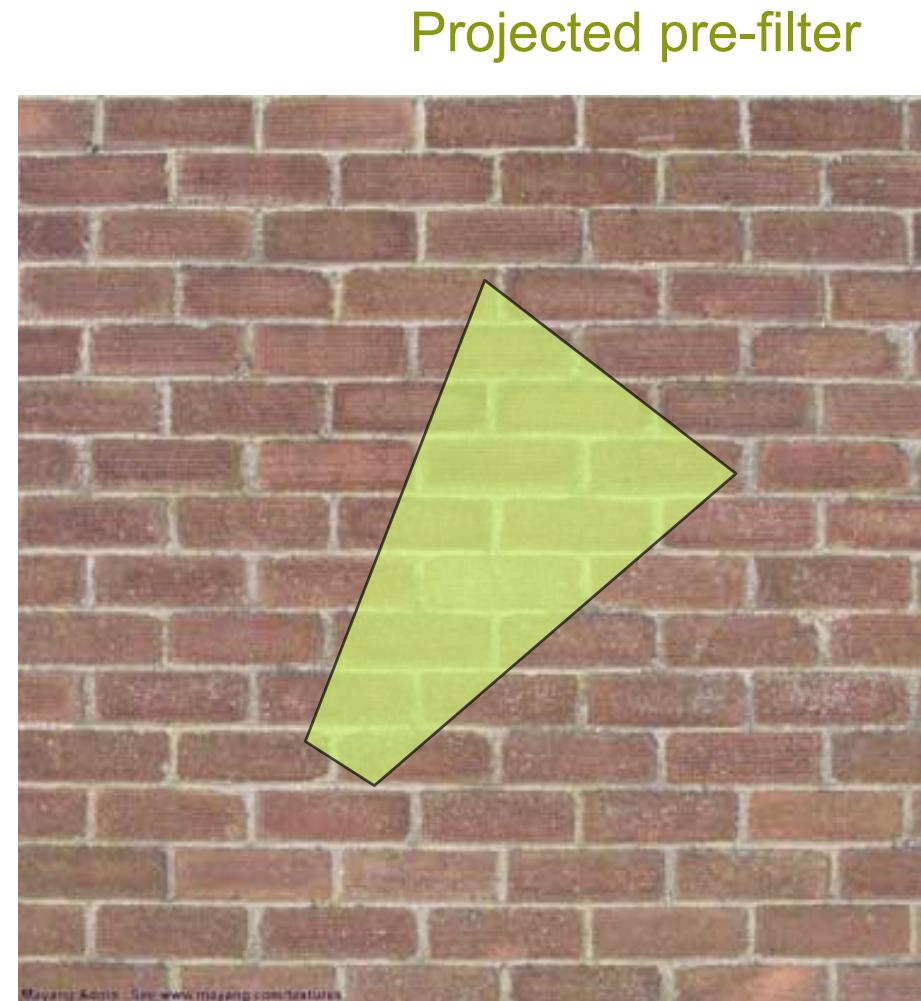
- Can be stored compactly: Only 1/3 more space!  
(verify)



# Finding the MIP Level

---

- Often we think of the pre-filter as a box
  - What is the projection of the square pixel “window” in texture space?



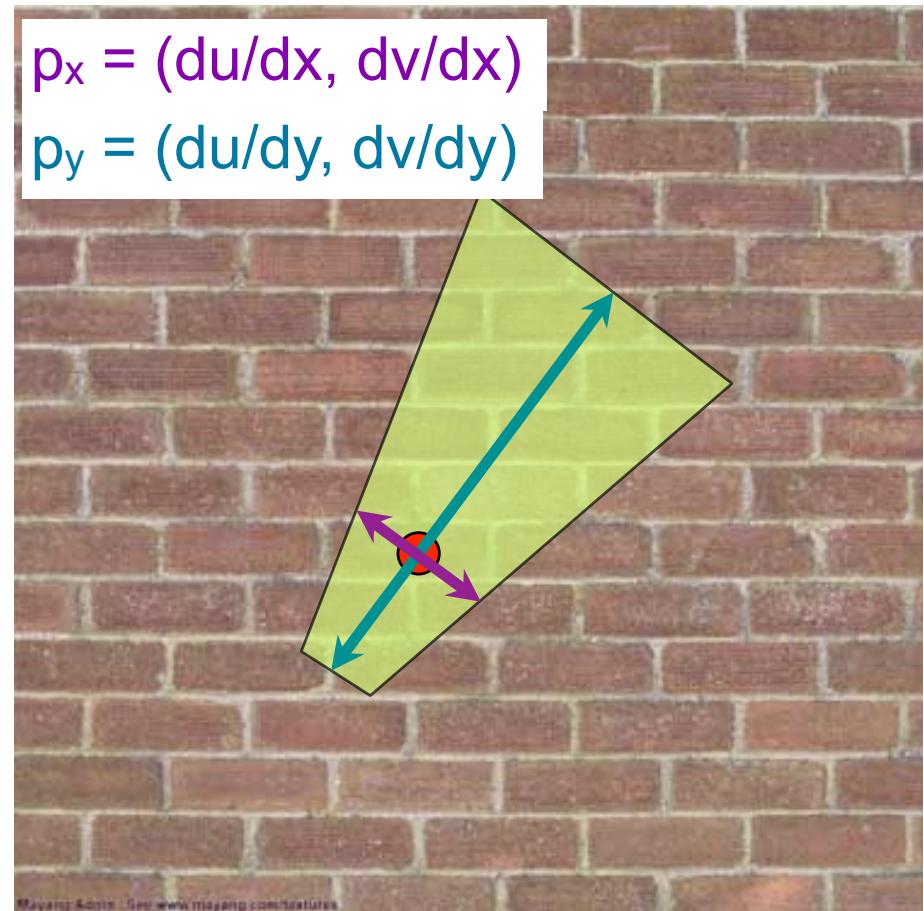
# Finding the MIP Level

---

- Often we think of the pre-filter as a box
  - What is the projection of the square pixel “window” in texture space?
  - Answer is in the partial derivatives  $p_x$  and  $p_y$  of  $(u,v)$  w.r.t. screen  $(x,y)$

● Projection of pixel center

Projected pre-filter

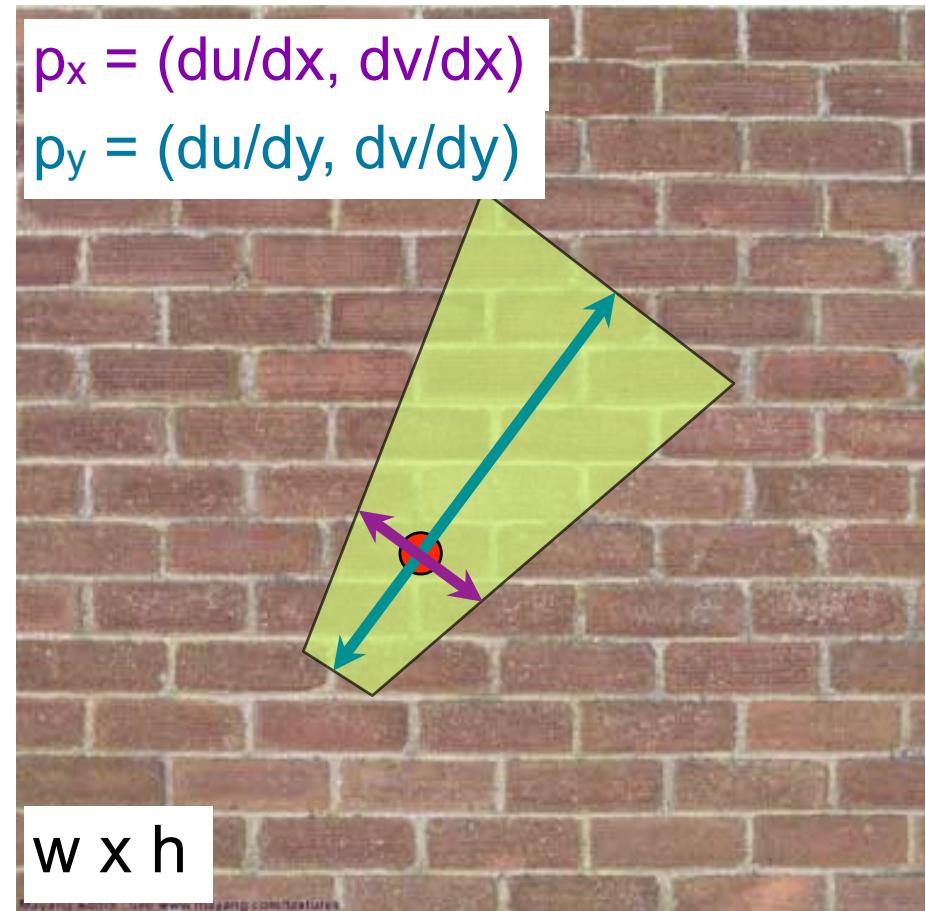


# For isotropic trilinear mipmapping

---

- No right answer, circular approximation
- Two most common approaches are
  - Pick level according to the length (in texels) of the longer partial  
 $\log_2 \max \{w|p_x|, h|p_y|\}$
  - Pick level according to the length of their sum  
 $\log_2 \sqrt{(w|p_x|)^2 + (h|p_y|)^2}$

● Projection of pixel center  
Projected pre-filter



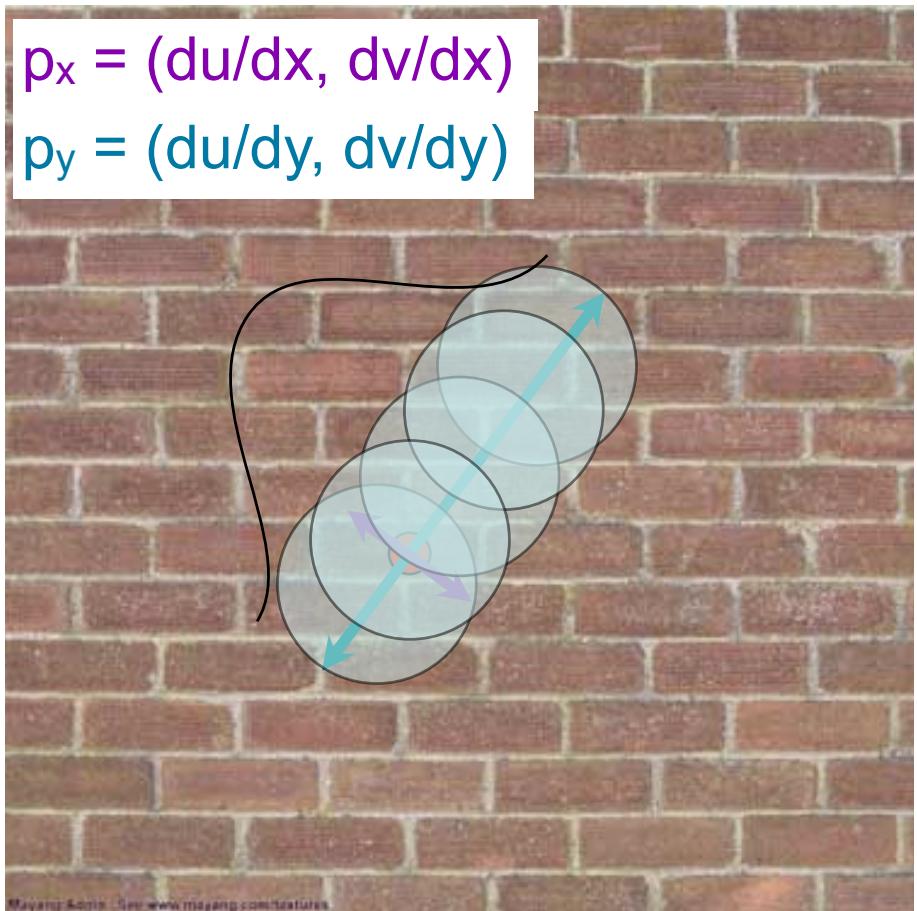
# Anisotropic filtering

---

- Pick levels according to smallest partial
  - well, actually max of (smallest and the largest/5)
- Distribute circular “probes” along longest one
- Weight them by a Gaussian

● Projection of pixel center

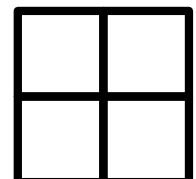
Projected pre-filter



# How Are Partials Computed?

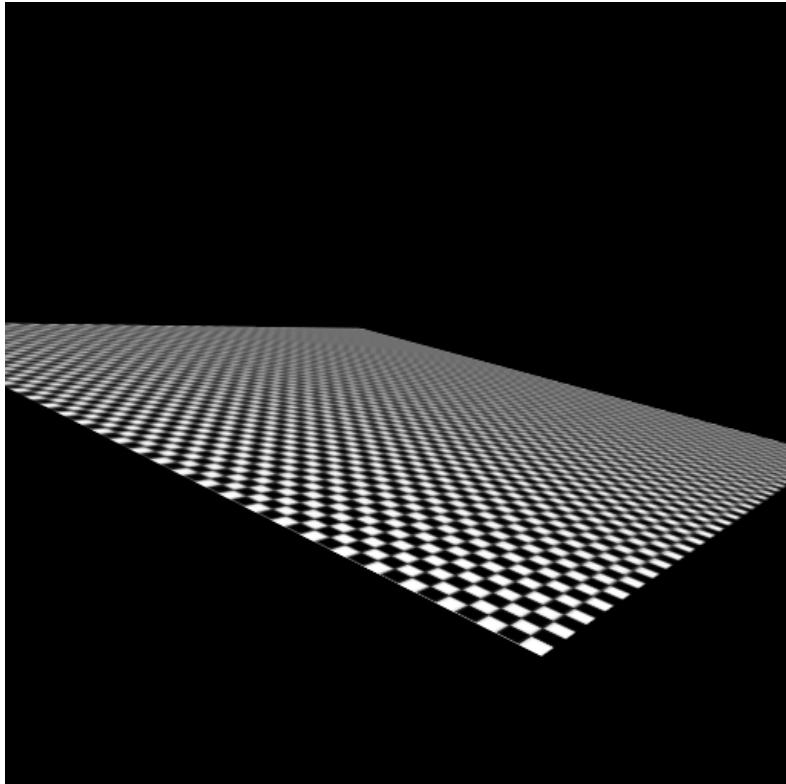
---

- You can derive closed form formulas based on the uv and xyw coordinates of the vertices...
  - This is what used to be done
- ..but shaders may compute texture coordinates programmatically, not necessarily interpolated
  - No way of getting analytic derivatives!
- In practice, use finite differences
  - GPUs process pixels in blocks of (at least) 4 anyway
    - These 2x2 blocks are called quads

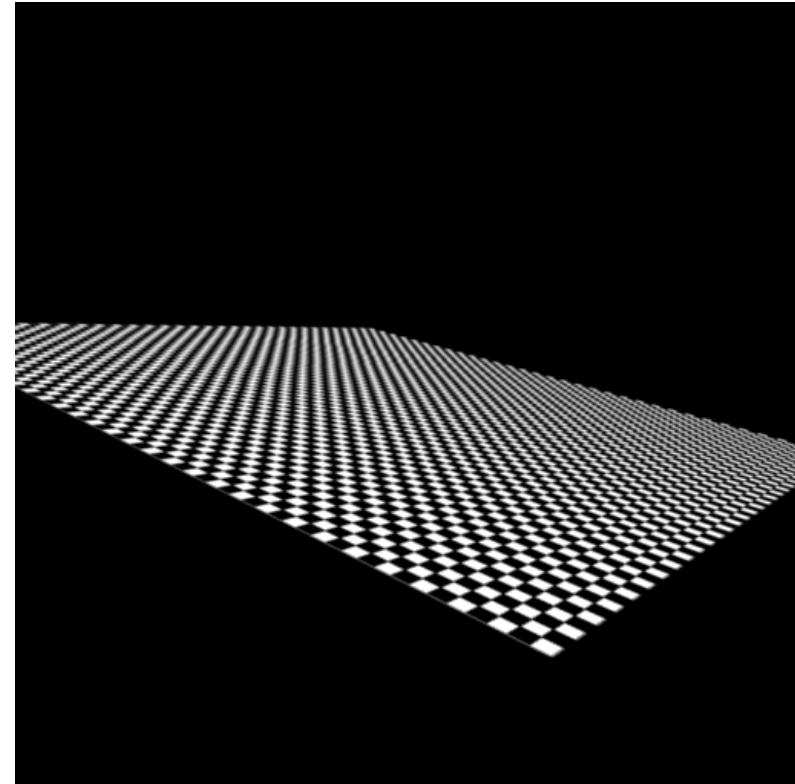


# Image Quality Comparison

---



trilinear mipmapping  
(excessive blurring)



anisotropic filtering

# Further Reading

---

- Paul Heckbert published seminal work on texture mapping and filtering in his master's thesis (!)
  - Including EWA
  - Highly recommended reading!
  - See <http://www.cs.cmu.edu/~ph/texfund/texfund.pdf>
- More reading
  - [Feline: Fast Elliptical Lines for Anisotropic Texture Mapping, McCormack, Perry, Farkas, Jouppi SIGGRAPH 1999](#)
  - [Texram: A Smart Memory for Texturing Schilling, Knittel, Strasser,. IEEE CG&A, 16\(3\): 32-41](#)



# Questions?

---

# Ray Casting vs. Rendering Pipeline

---

## Ray Casting

For each pixel

    For each object

- Ray-centric
- Needs to store scene in memory
- (Mostly) Random access to scene

## Rendering Pipeline

For each triangle

    For each pixel

- Triangle centric
- Needs to store image (and depth) into memory
- (Mostly) random access to frame buffer

Which is smaller? Scene or Frame?

Which is easiest to access randomly?

# Ray Casting vs. Rendering Pipeline

---

## Ray Casting

For each pixel

    For each object

- Ray-centric
- Needs to store scene in memory
- (Mostly) Random access to scene

## Rendering Pipeline

For each triangle

    For each pixel

- Triangle centric
- Needs to store image (and depth) into memory
- (Mostly) random access to frame buffer

Which is smaller? Scene or Frame?

Frame

Which is easiest to access randomly?

Frame because regular sampling

# Ray Casting vs. Rendering Pipeline

---

## Ray Casting

For each pixel

    For each object

- Whole scene must be in memory
- Needs spatial acceleration to be efficient
- + Depth complexity: no computation for hidden parts
- + Atomic computation
- + More general, more flexible
  - Primitives, lighting effects, adaptive antialiasing

## Rendering Pipeline

For each triangle

    For each pixel

- Harder to get global illumination
- Needs smarter techniques to address depth complexity (overdraw)
- + Primitives processed one at a time
- + Coherence: geometric transforms for vertices only
- + Good bandwidth/computation ratio
- + Minimal state required, good memory behavior

# Conclusions

---

- Rasterization and ray casting do the same thing
- Just swap the two loops
- And cache what is independent of pixel location

# Good References

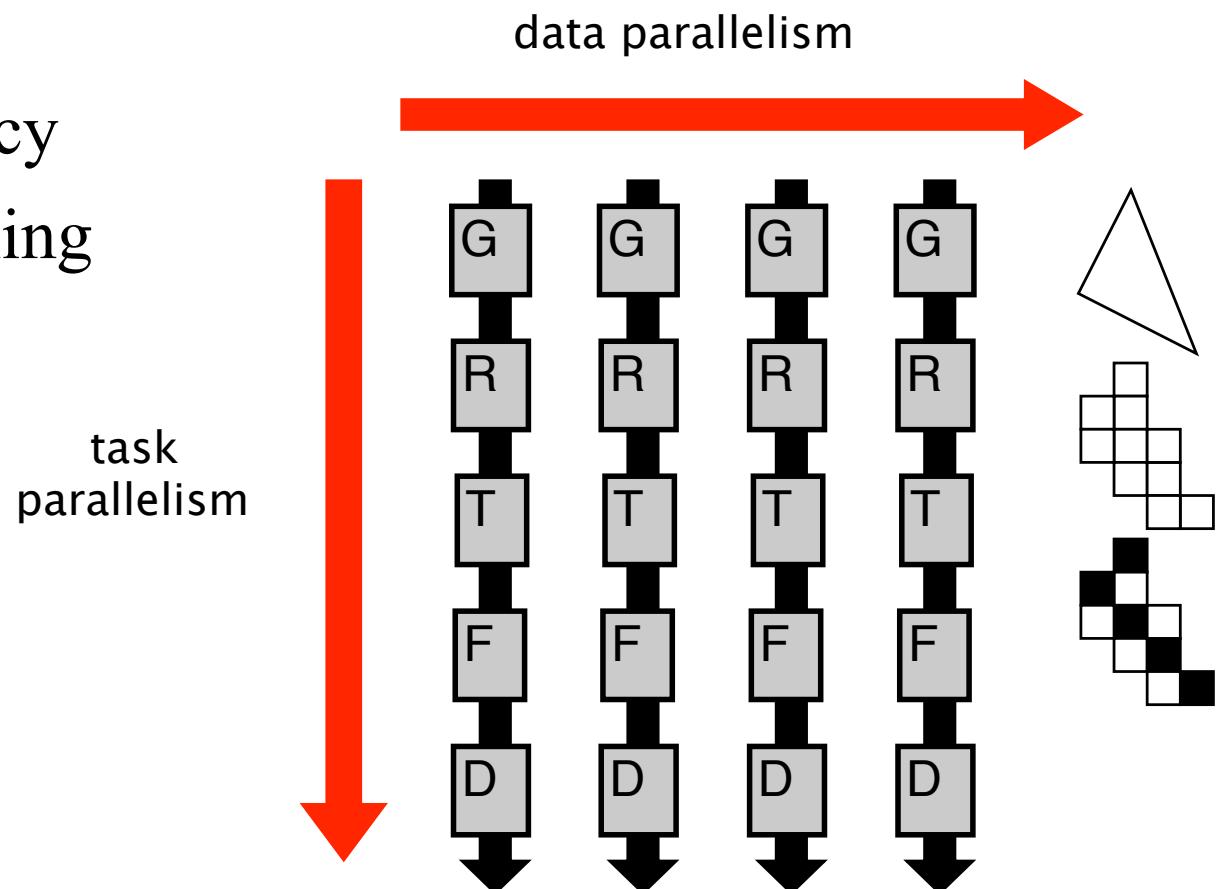
---

- <http://www.tomshardware.com/reviews/ray-tracing-rasterization,2351.html>
- <http://c0de517e.blogspot.com/2011/09/raytracing-myths.html>
- <http://people.csail.mit.edu/fredo/tmp/rendering.pdf>

# Graphics Hardware

---

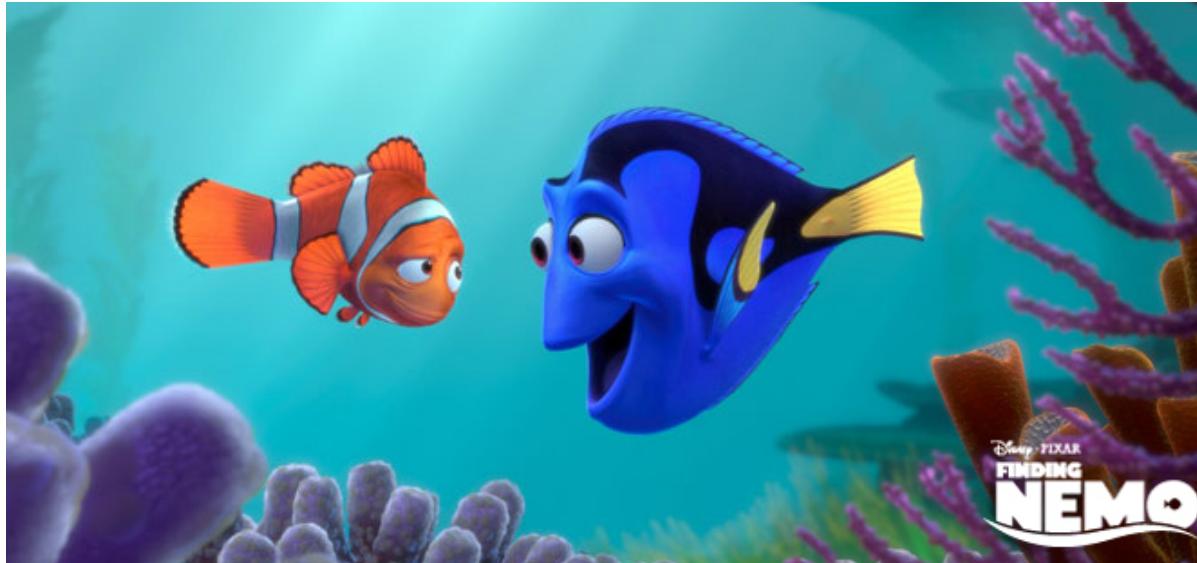
- High performance through
  - Parallelism
  - Specialization
  - No data dependency
  - Efficient pre-fetching
- More later



# Movies

---

both rasterization and ray tracing



# Games

rasterization



GAMESPOT

rasterization

# Simulation



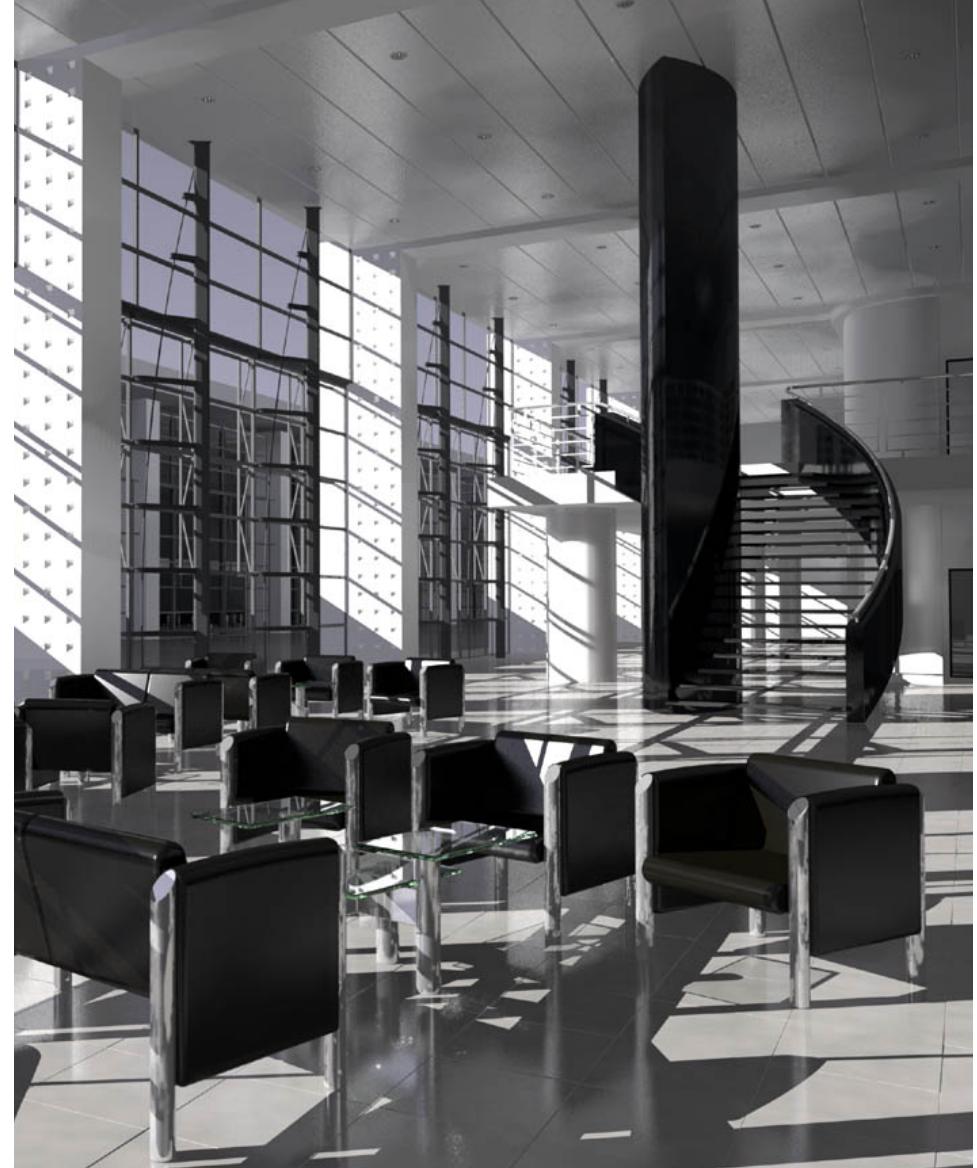
# CAD-CAM & Design

rasterization for GUI,  
anything for final image



# Architecture

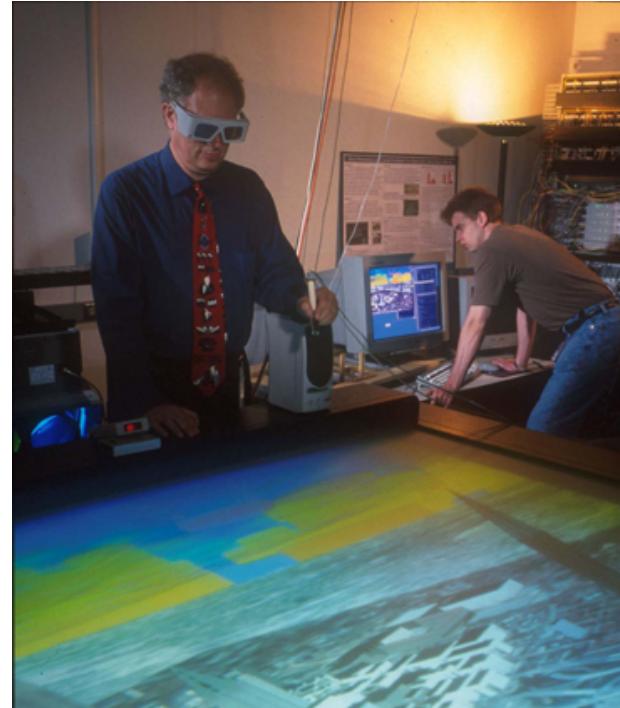
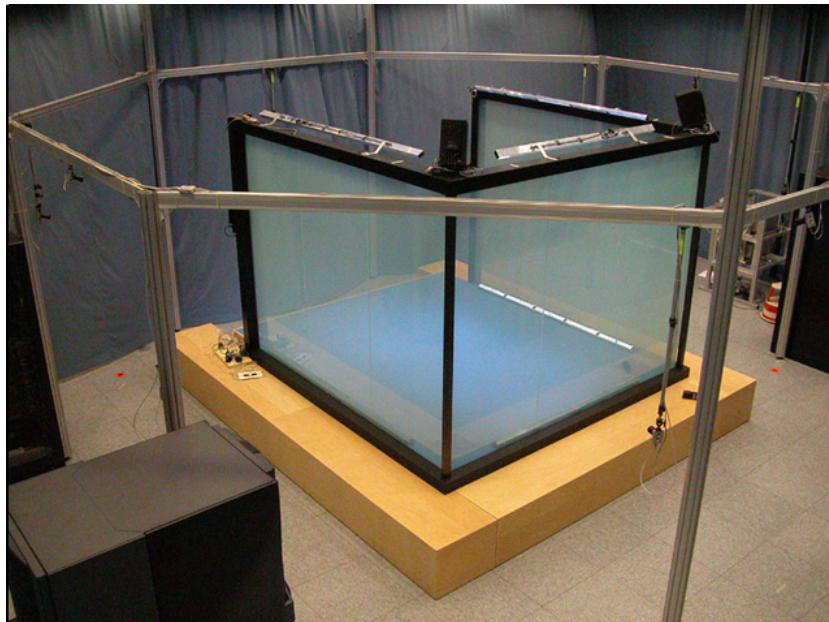
ray-tracing, rasterization with preprocessing for complex lighting



# Virtual Reality

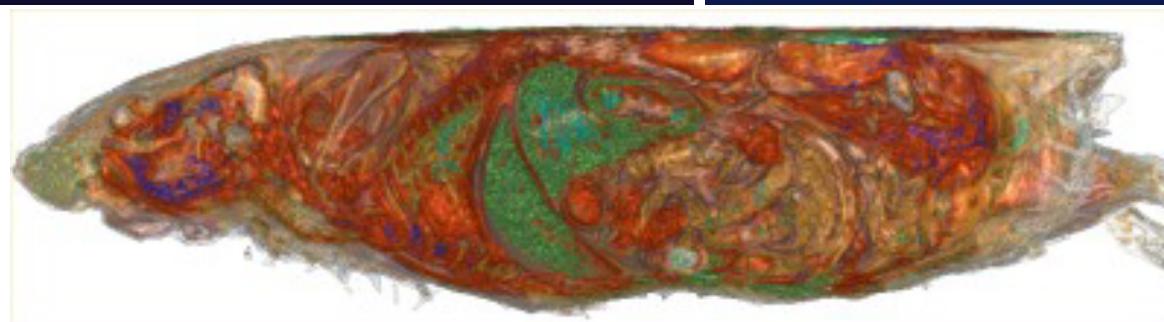
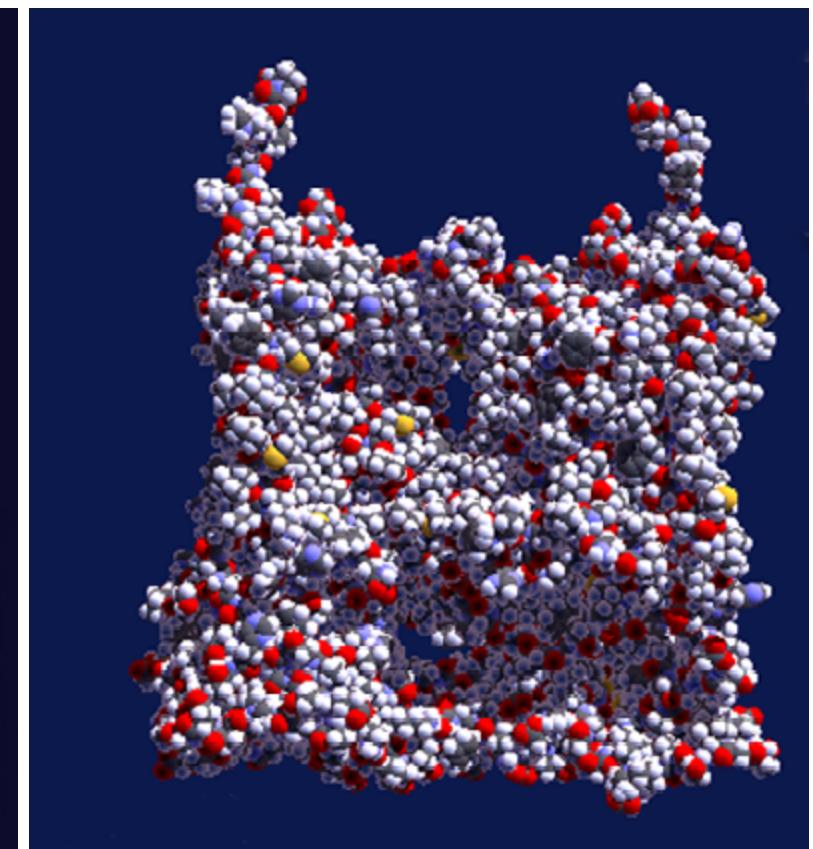
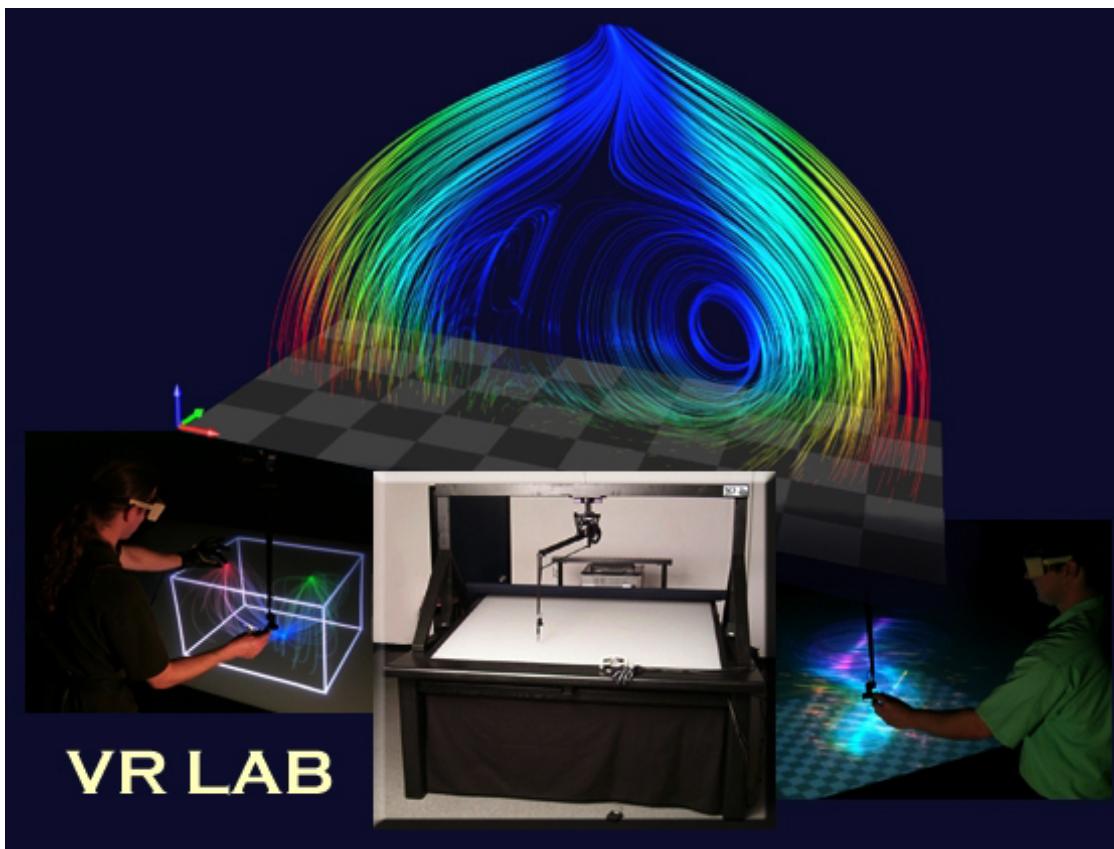
---

rasterization



# Visualization

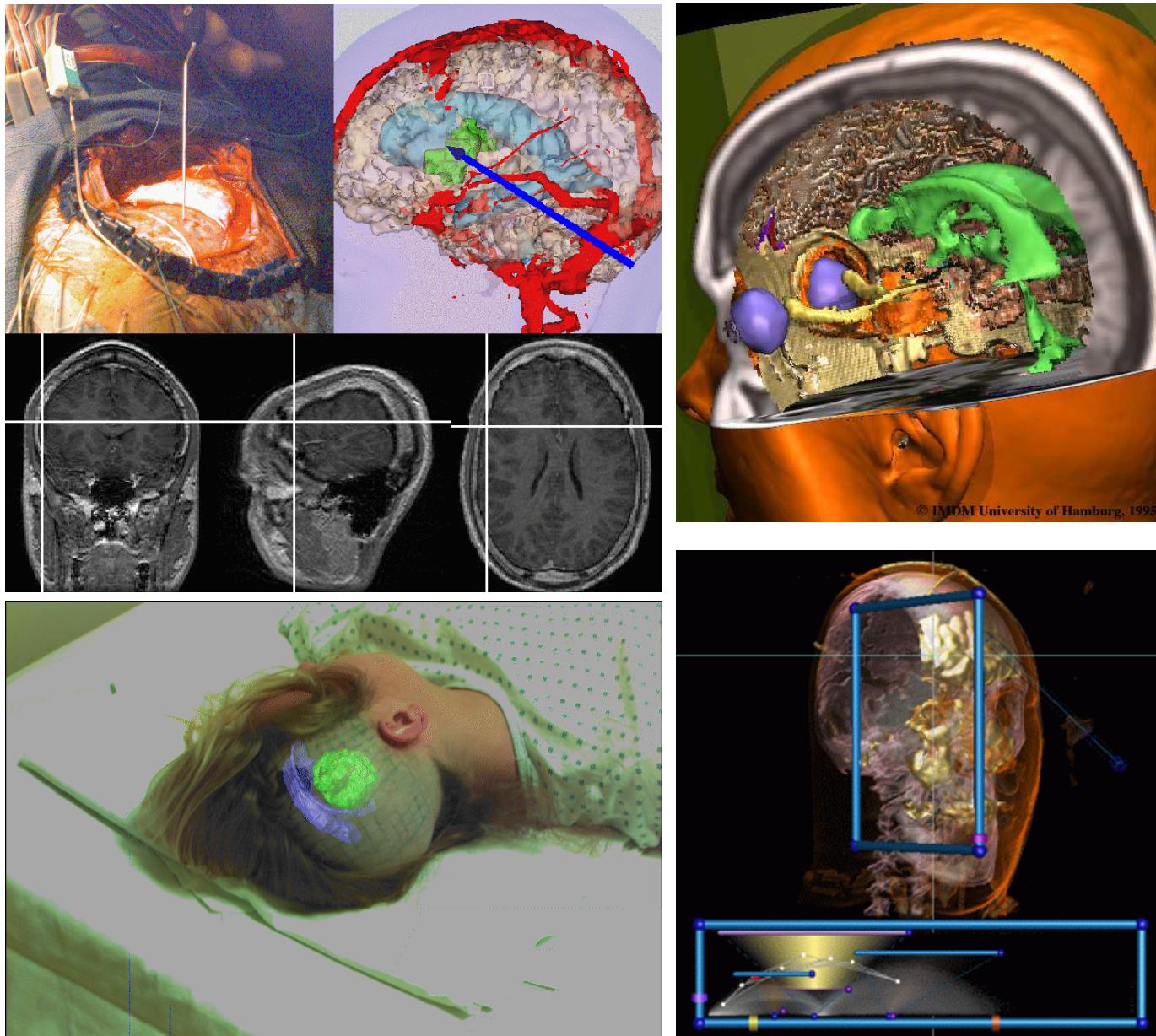
mostly rasterization,  
interactive ray-tracing is starting



# Medical Imaging

---

same as  
visualization



# More issues

---

- Transparency
  - Difficult, pretty much unsolved!
- Alternative
  - Reyes (Pixar's Renderman)
  - deferred shading
  - pre-Z pass
  - tile-based rendering
- Shadows
- Reflections, global illumination

# That's all for today!

---