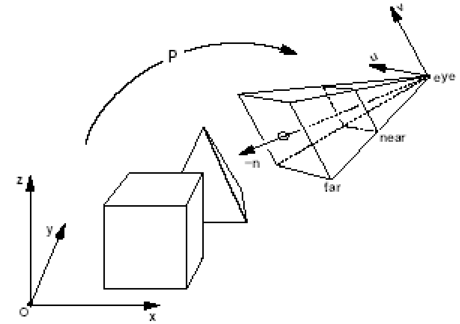
Graphics Pipeline & Rasterization II

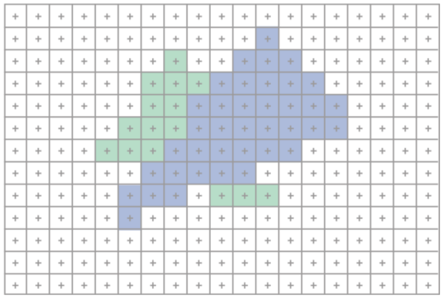
Image removed due to copyright restrictions.

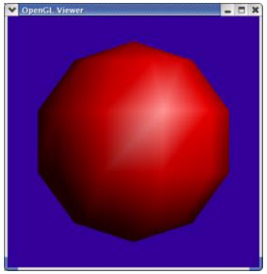
MIT EECS 6.837
Computer Graphics
Wojciech Matusik

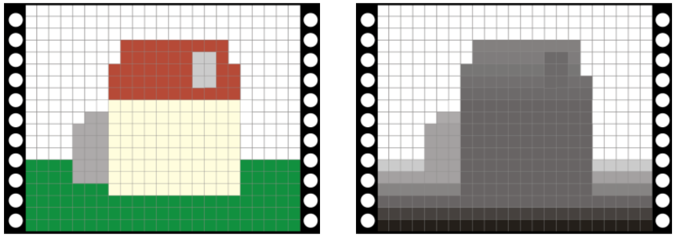
Modern Graphics Pipeline

- 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

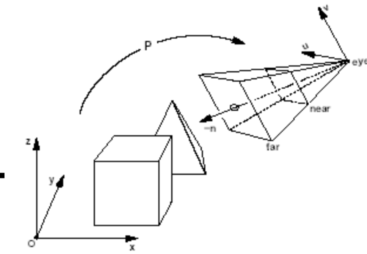




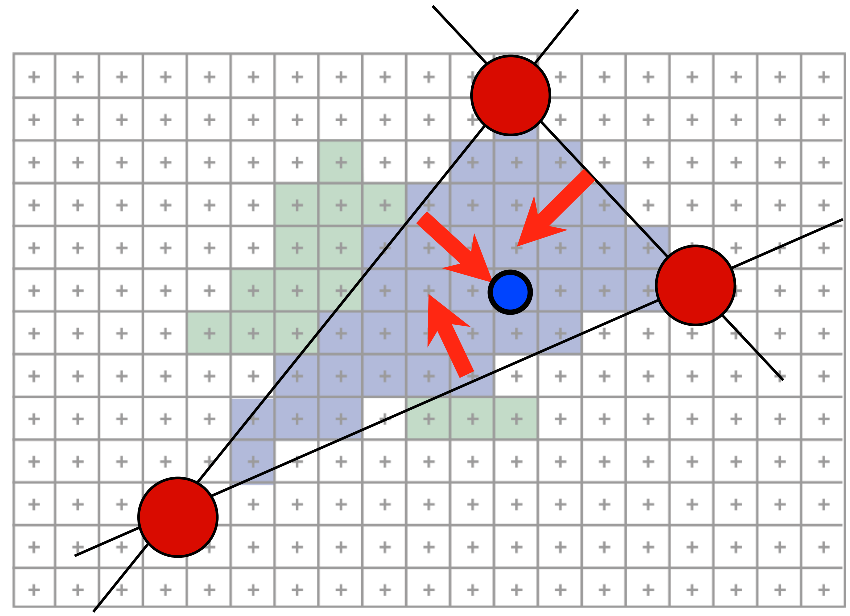


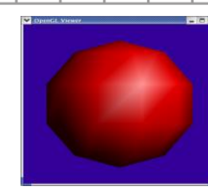


Modern Graphics Pipeline

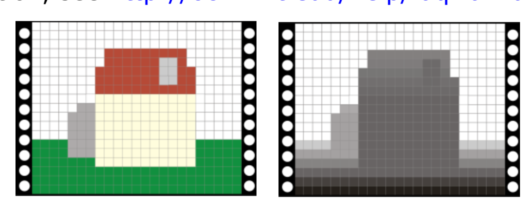


- Project vertices to 2D (image)
- Rasterize triangle: find which pixels should be lit
  - For each pixel, test 3 edge equations
    - if all pass, draw pixel

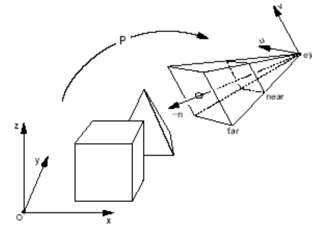




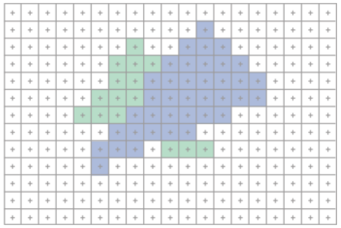
- Compute per-pixel color
- Test visibility (Z-buffer), update frame buffer color



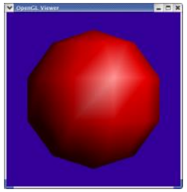
Modern Graphics Pipeline



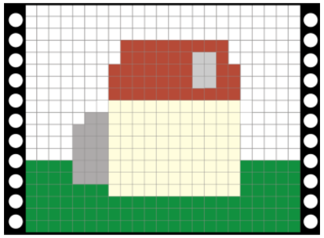
- Perform projection of vertices
- Rasterize triangle: find which pixels should be lit



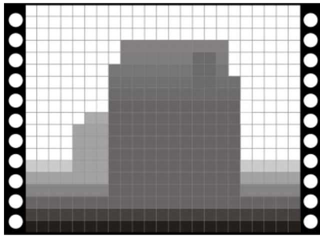
- Compute per-pixel color



- 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
    ```



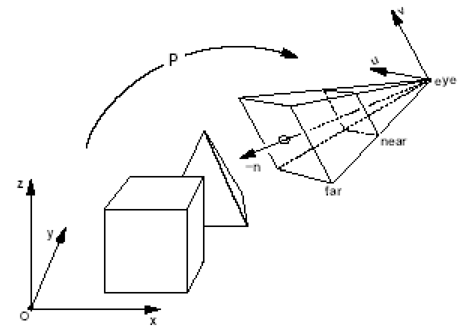
frame buffer

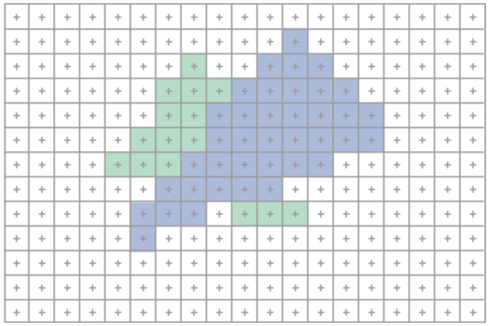


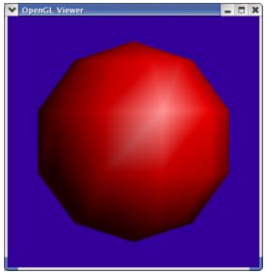
Z buffer
    ```

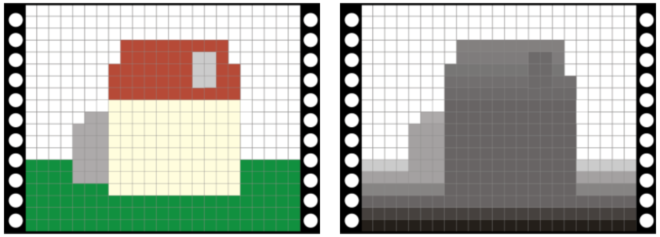
Modern Graphics Pipeline

```
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<zbuffer[x,y]
            zbuffer[x,y]=z
            framebuffer[x,y]=shade()
```

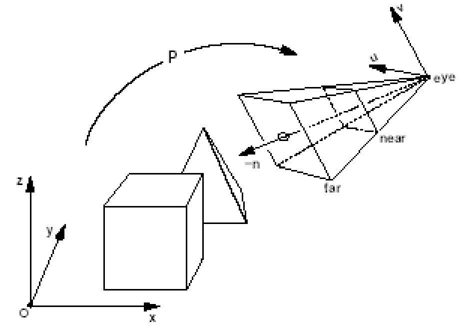




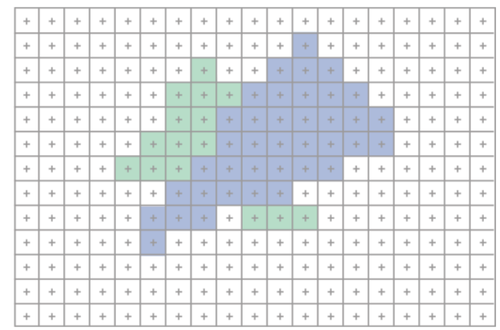


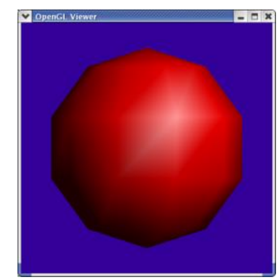


Modern Graphics Pipeline

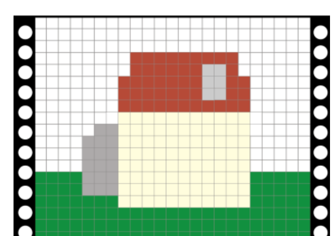


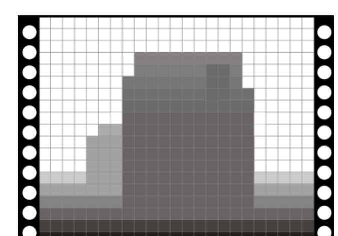

```
For each triangle
   transform into eye space
   (perform projection)
   setup 3 edge equations
   for each pixel x,y
      if passes all edge equations
         compute z
         if z<zbuffer[x,y]
            zbuffer[x,y]=z
            framebuffer[x,y]=shade()
```





Questions?

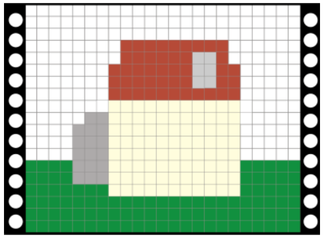


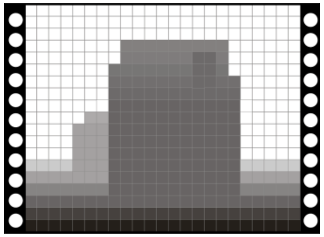


Interpolation in Screen Space

- 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

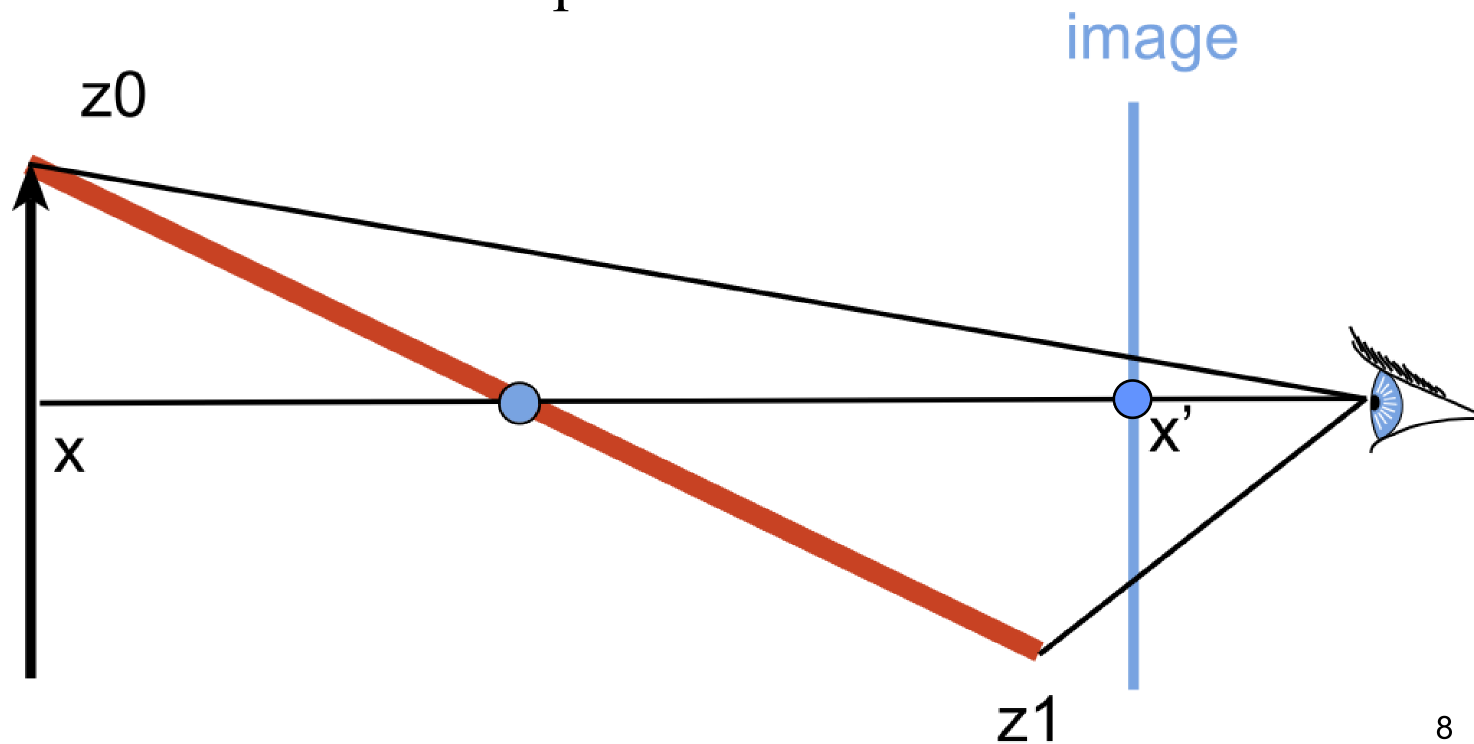
```
For each triangle
   for each pixel (x,y)
      if passes all edge equations
         compute z
         if z<zbuffer[x,y]
            zbuffer[x,y]=z
            framebuffer[x,y]=shade()
```



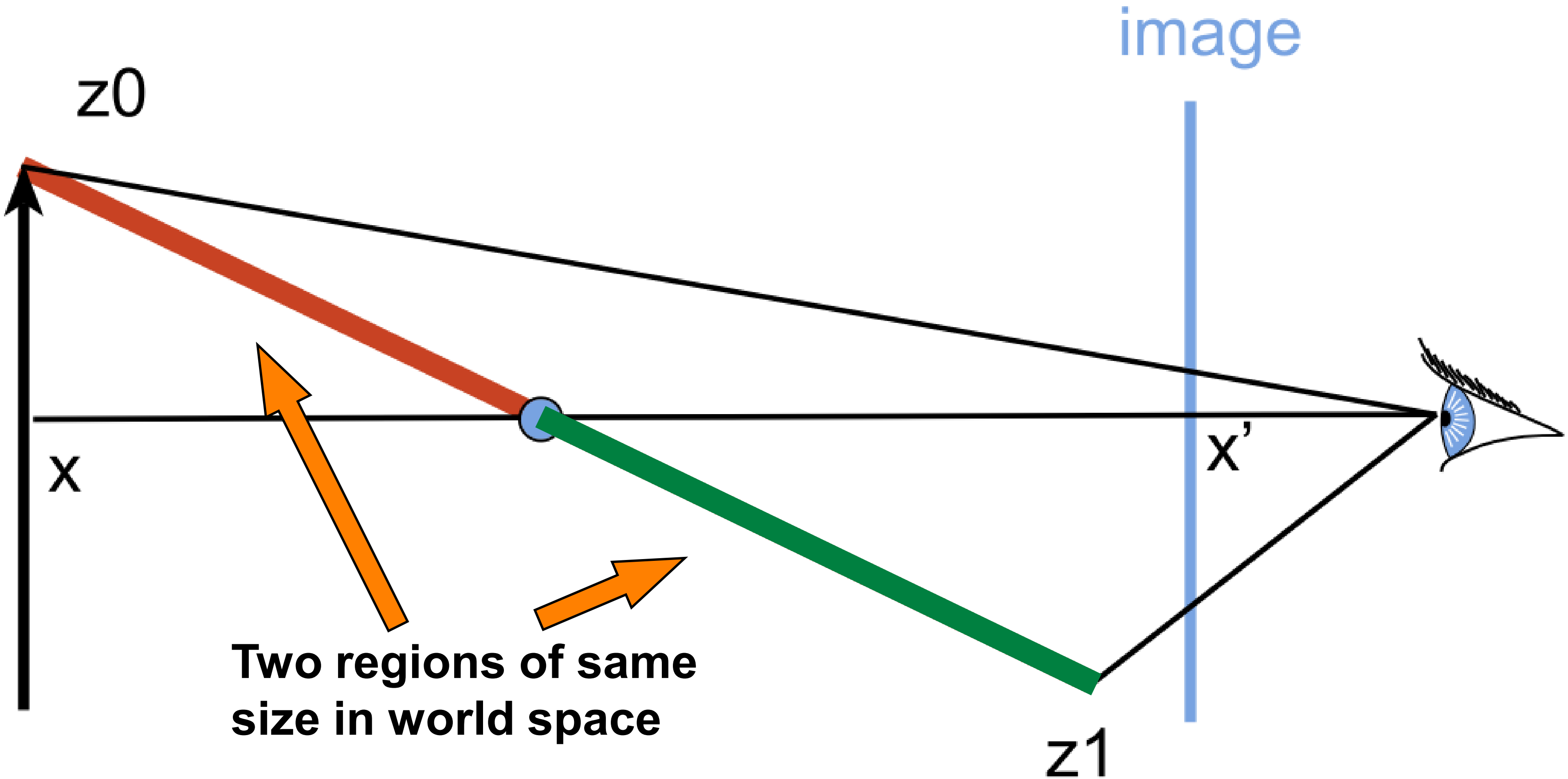


Interpolation in Screen Space

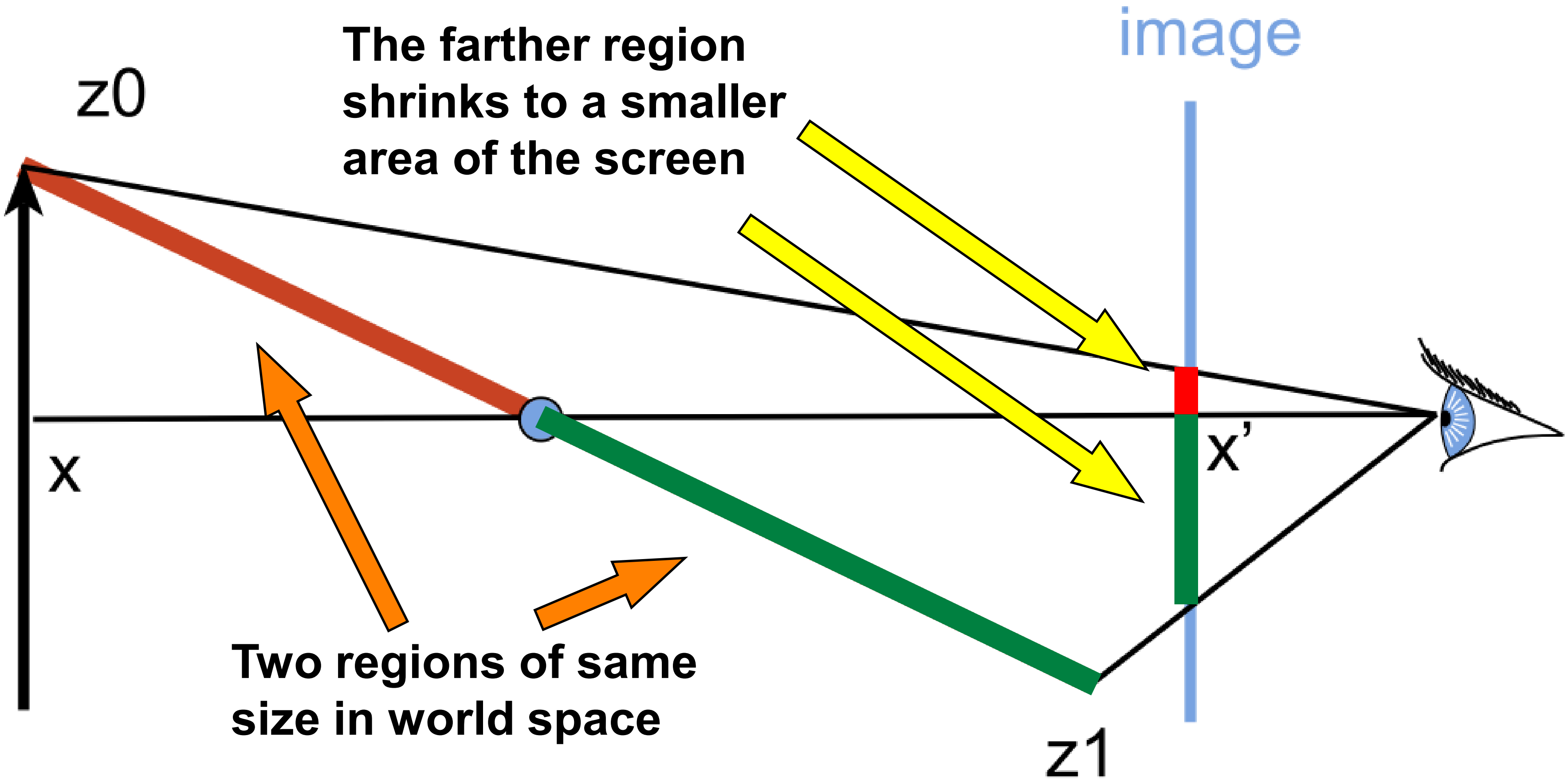
- Also need to interpolate color, normals, texture coordinates, etc. between vertices
  - We did this with barycentrics in ray casting
    - Linear interpolation in object space
  - Is this the same as linear interpolation on the screen?



Interpolation in Screen Space

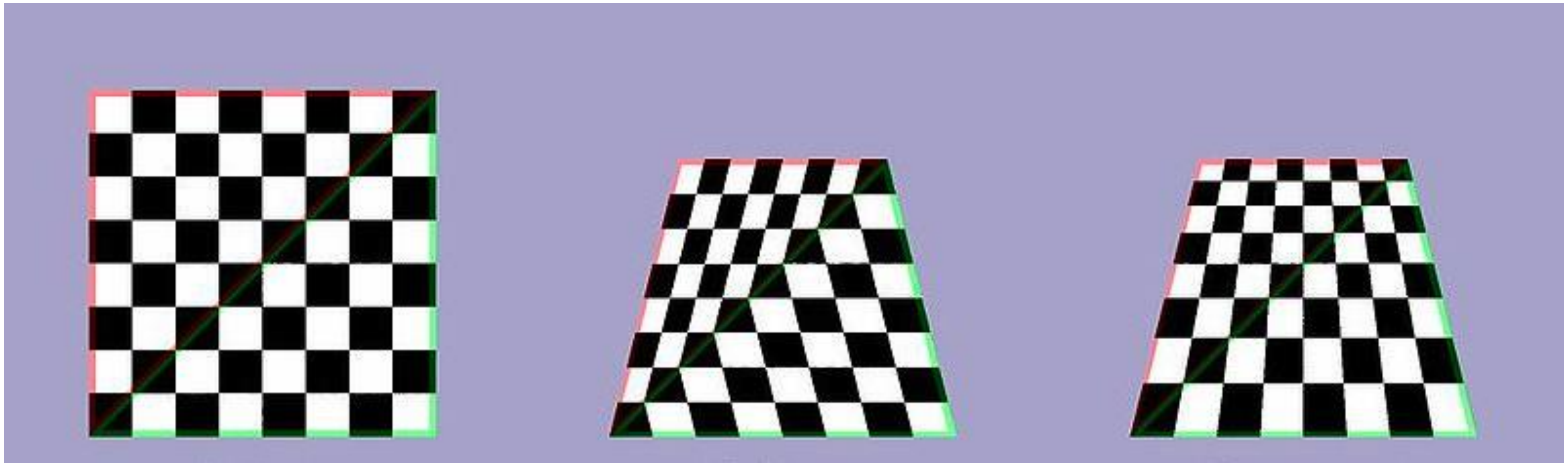


Interpolation in Screen Space



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



Head-on view

linear screen-space ("Gouraud") interpolation
BAD

Perspective-correct Interpolation

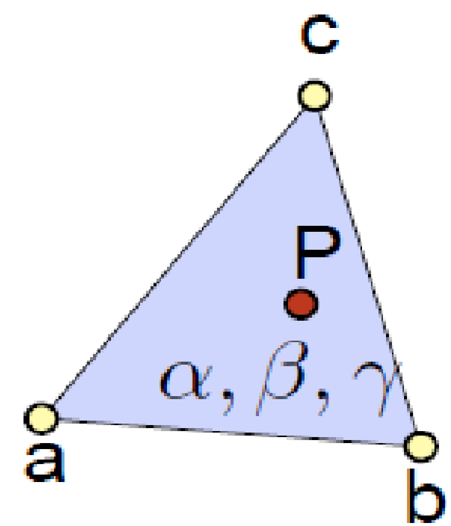
Back to the basics: Barycentrics

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

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

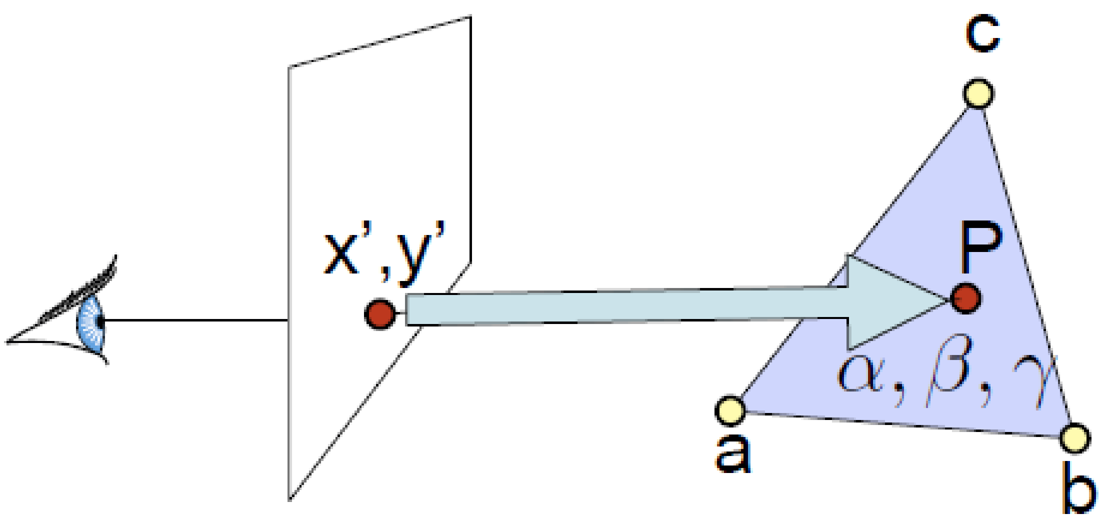
  – Remember, $\alpha + \beta + \gamma = 1, \quad \alpha, \beta, \gamma \geq 0$

- Barycentrics are very general:
  – Work for x, y, z, u, v, r, g, b
  – Anything that varies linearly in object space
  – including z



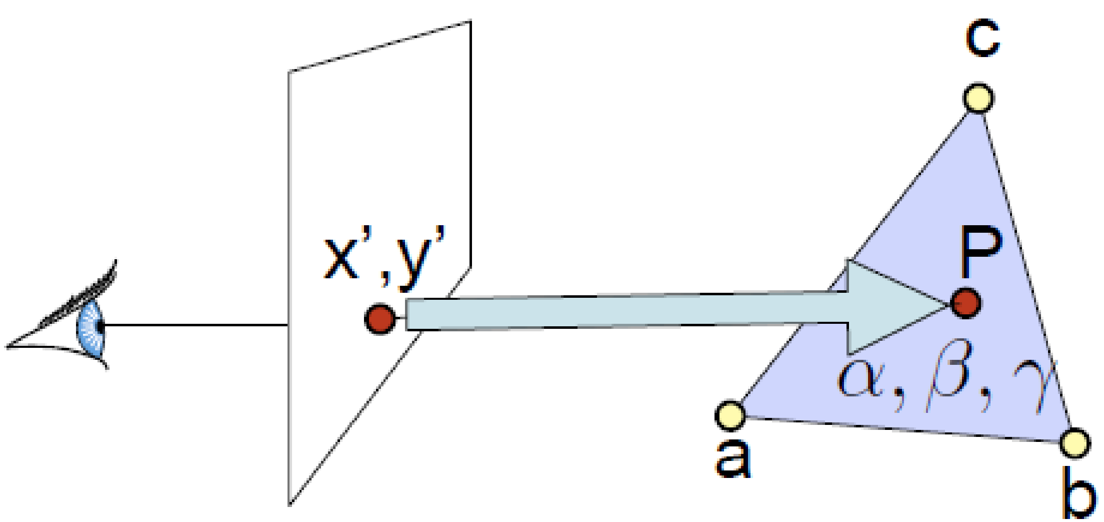
Basic strategy

- Given screen-space x’, y’
- Compute barycentric coordinates
- Interpolate anything specified at the three vertices



Basic strategy

- How to make it work
  - 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**)

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

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

- Let's project point P by projection matrix **C**

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

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 to 2D homogeneous coordinates

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

- No crazy near-far or storage of 1/z
- We 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*, the *(x,y)* screen coordinates are

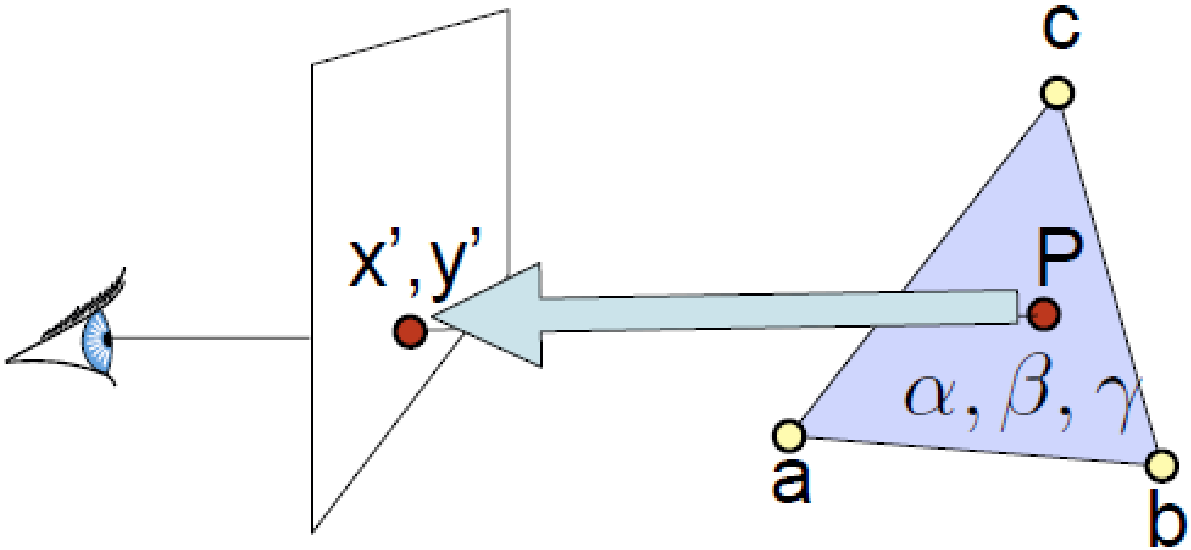
$$(P'_x/P'_w, P'y/P'w) =$$

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

Recap: barycentric to screen-space

projective equivalence (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

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

- 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

projective equivalence

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

- 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 is 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
  - can be used to interpolate z, color, texture coordinates, etc.

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 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
```

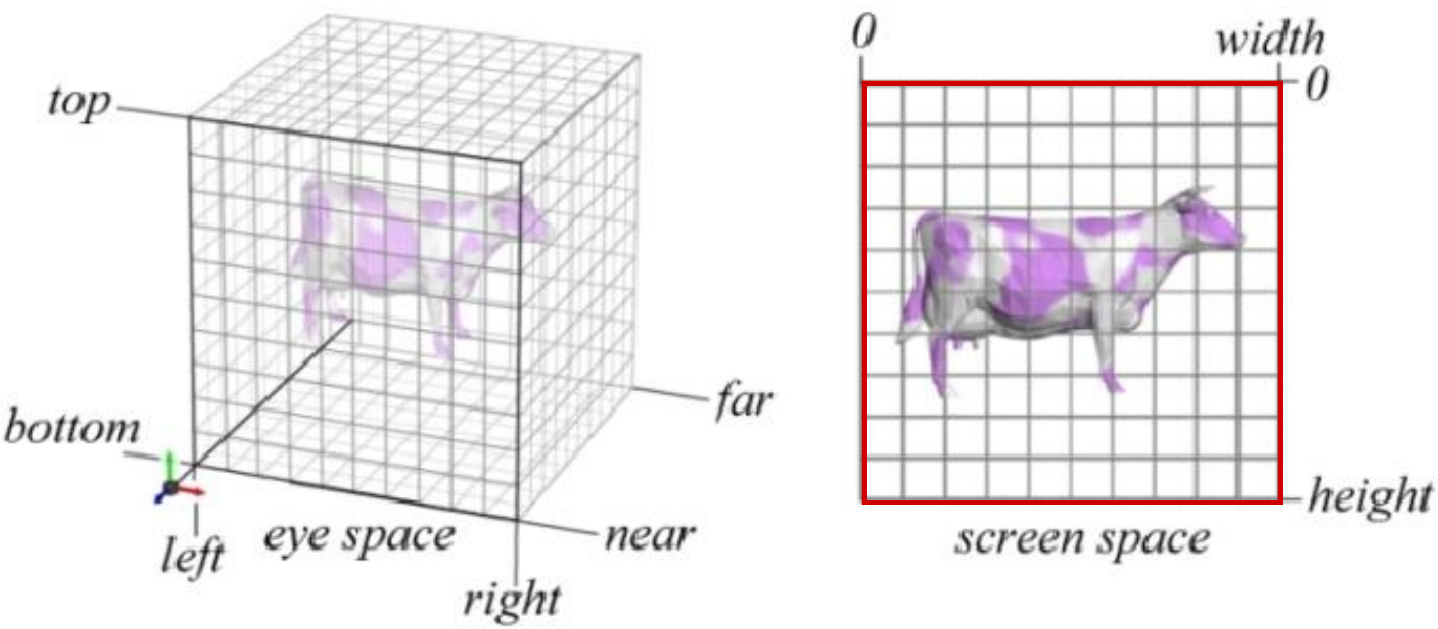
Pseudocode – Rasterization

```
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 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
```

Questions?

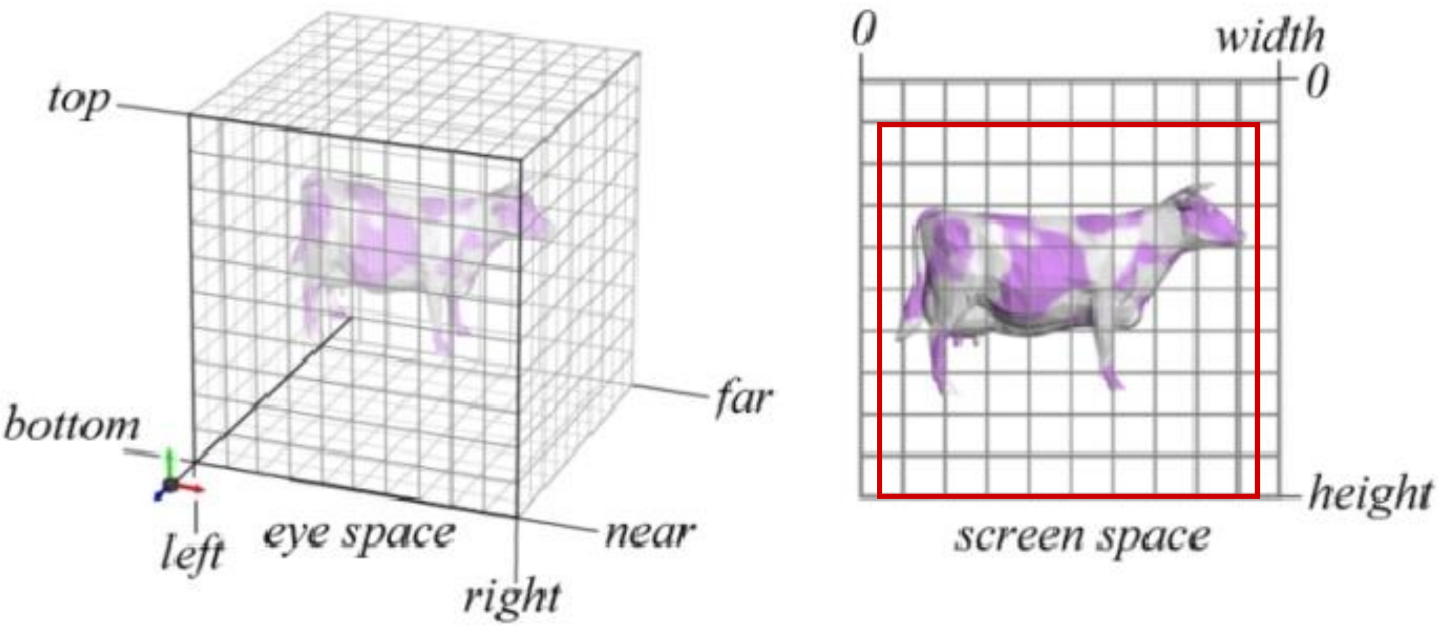
The infamous half pixel

- I refuse to teach it, but it's an annoying issue you should know about
- Do a line drawing of a rectangle from [top, right] to [bottom,left]
- Do we actually draw the columns/rows of pixels?



The infamous half pixel

- Displace by half a pixel so that top, right, bottom, left are in the middle of pixels
- Just change the viewport transform

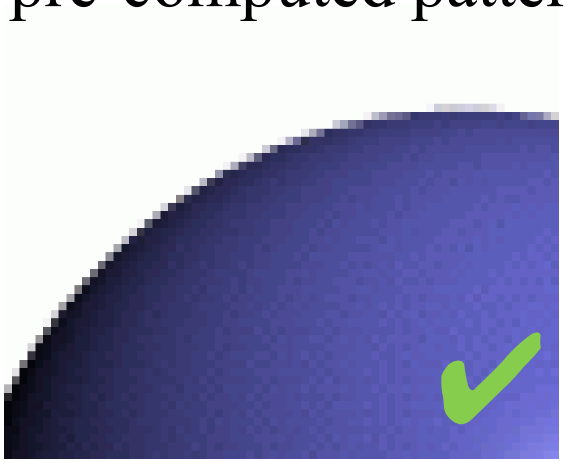


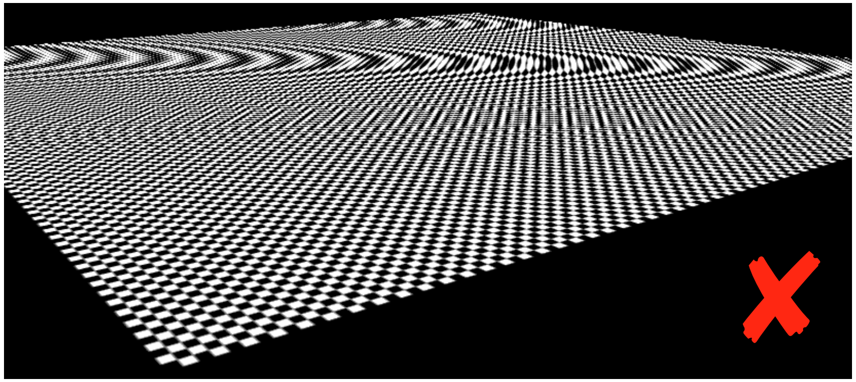
Leonard McMillan, Computer Science at the University of North Carolina in Chapel Hill.

Questions?

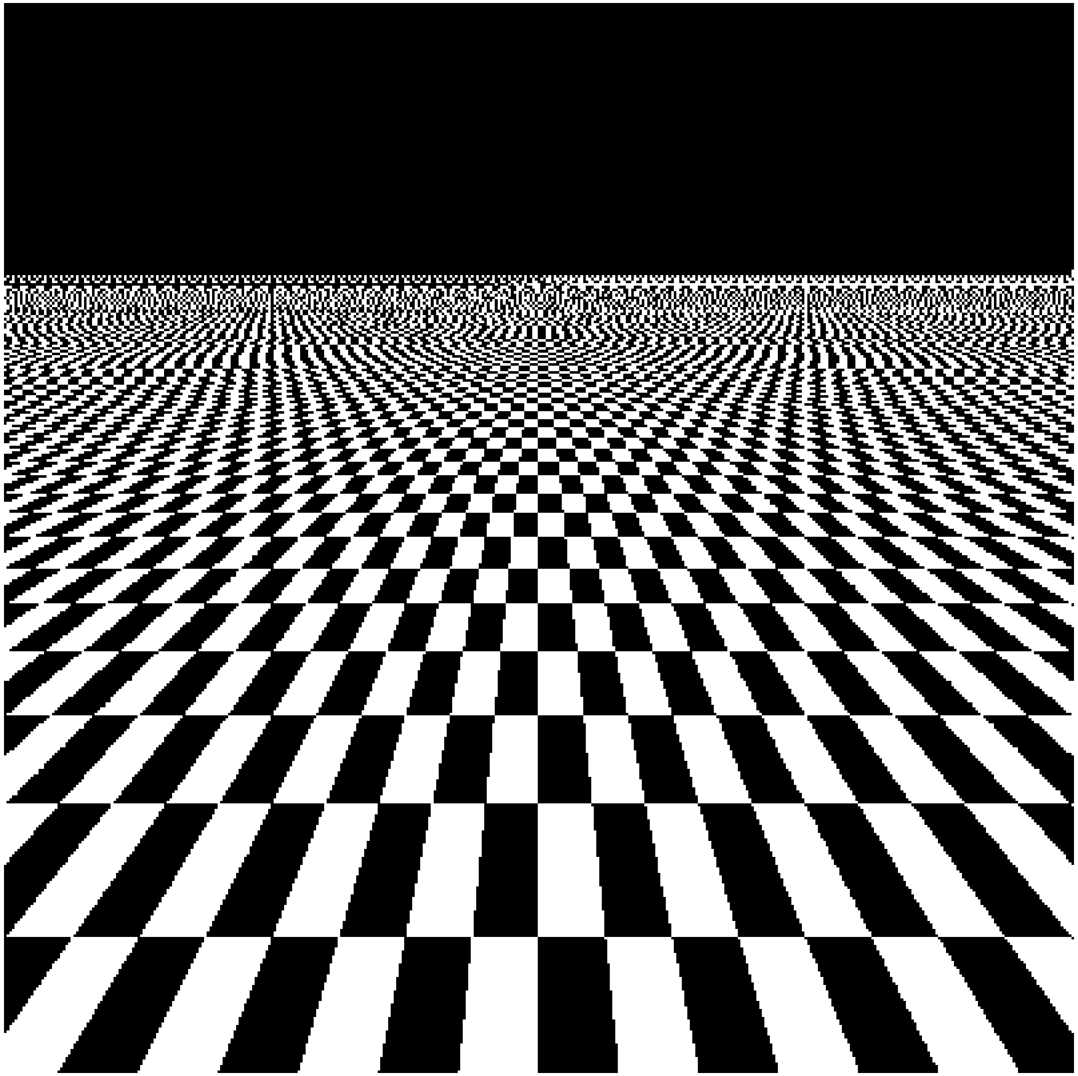
Supersampling

- 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 pattern for a big block of pixels

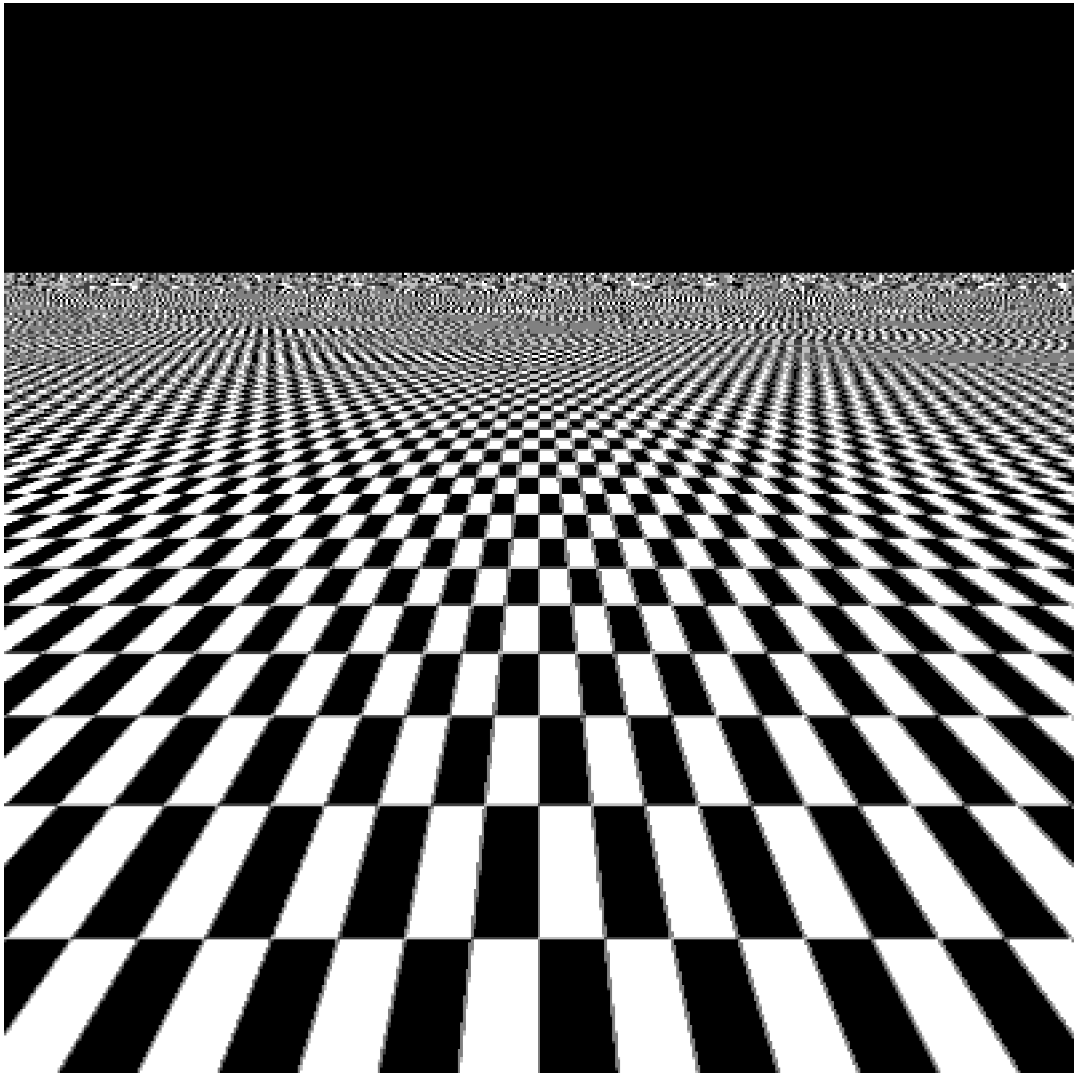




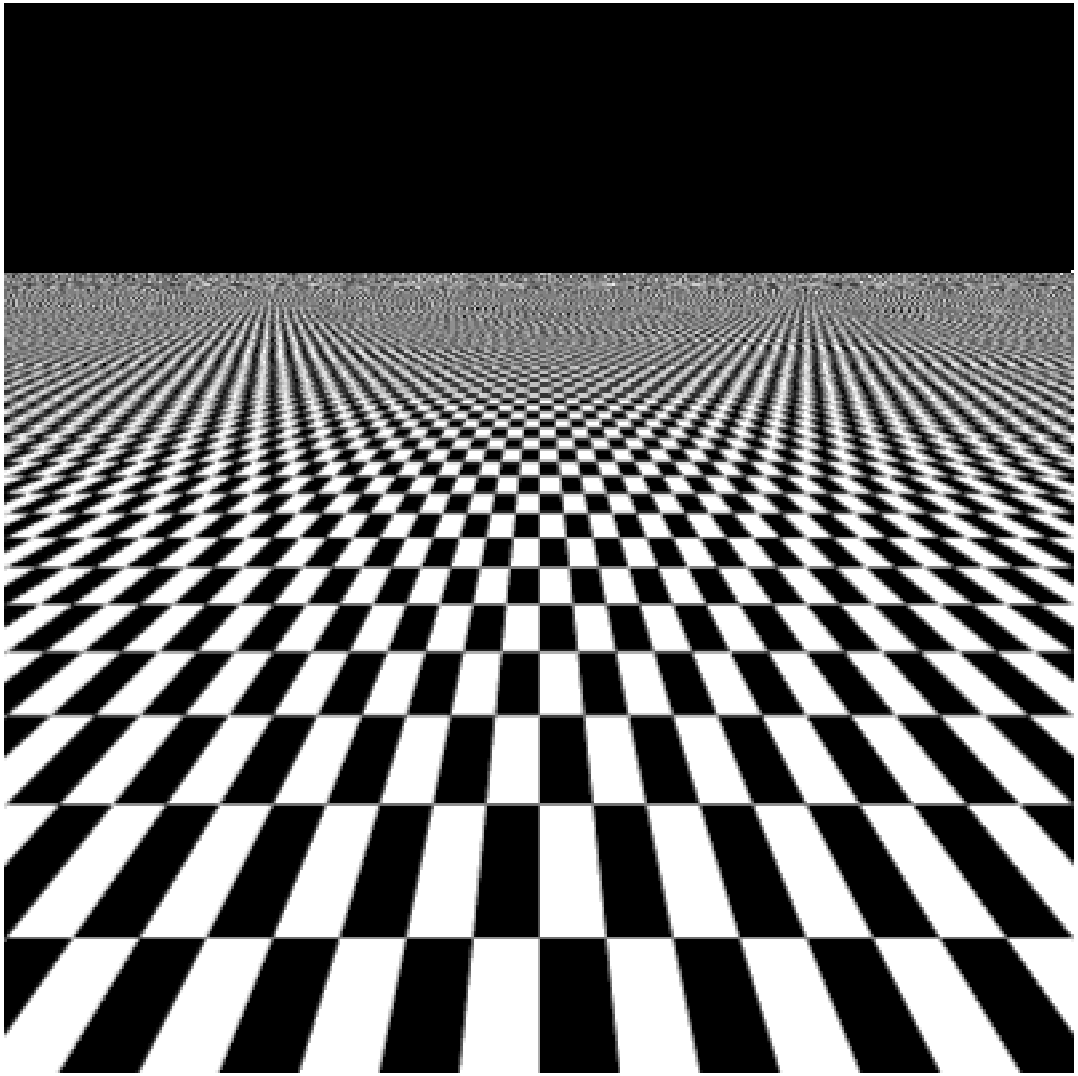
1 Sample / Pixel



4 Samples / Pixel



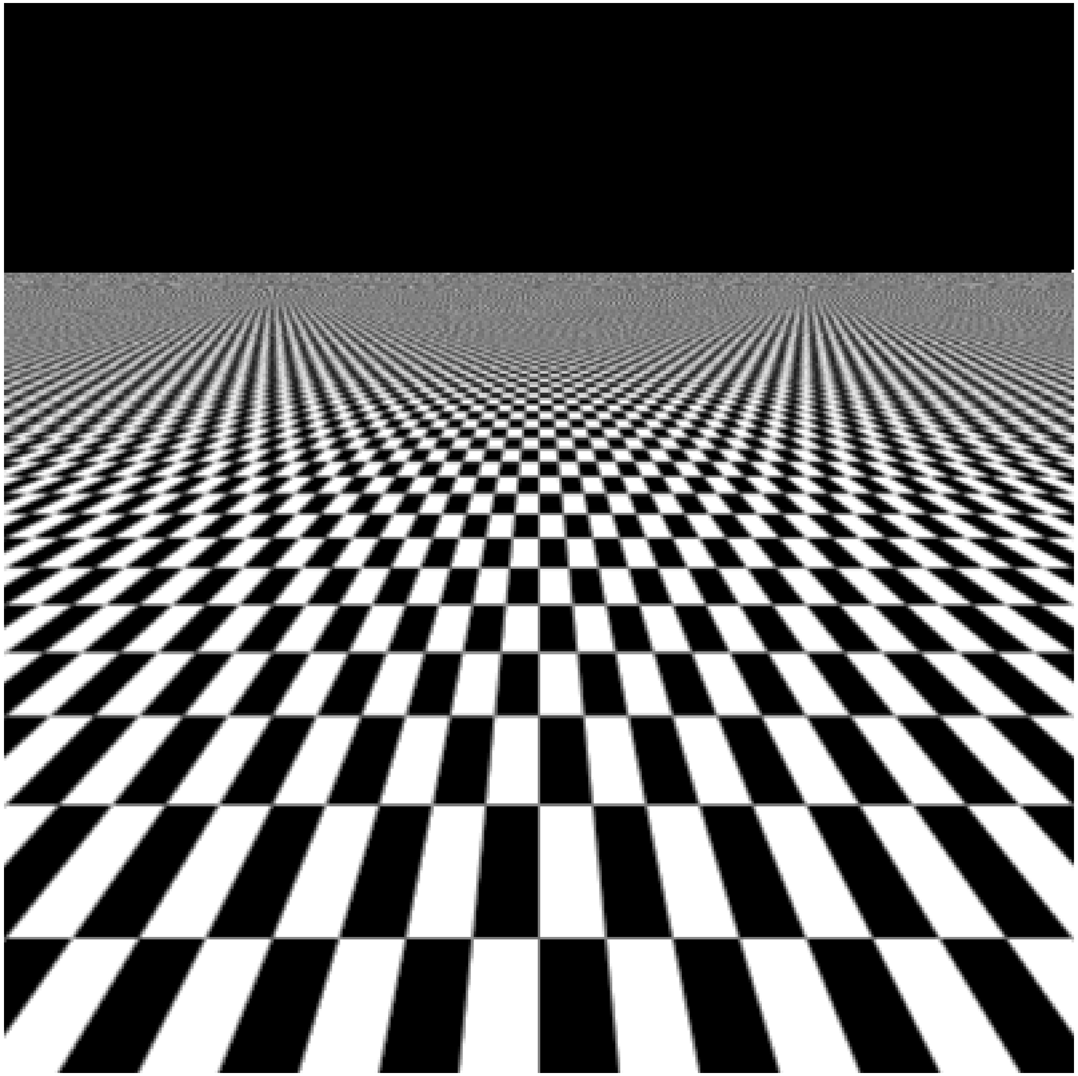
16 Samples / Pixel



100 Samples / Pixel

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

We are really only pushing the problem farther.

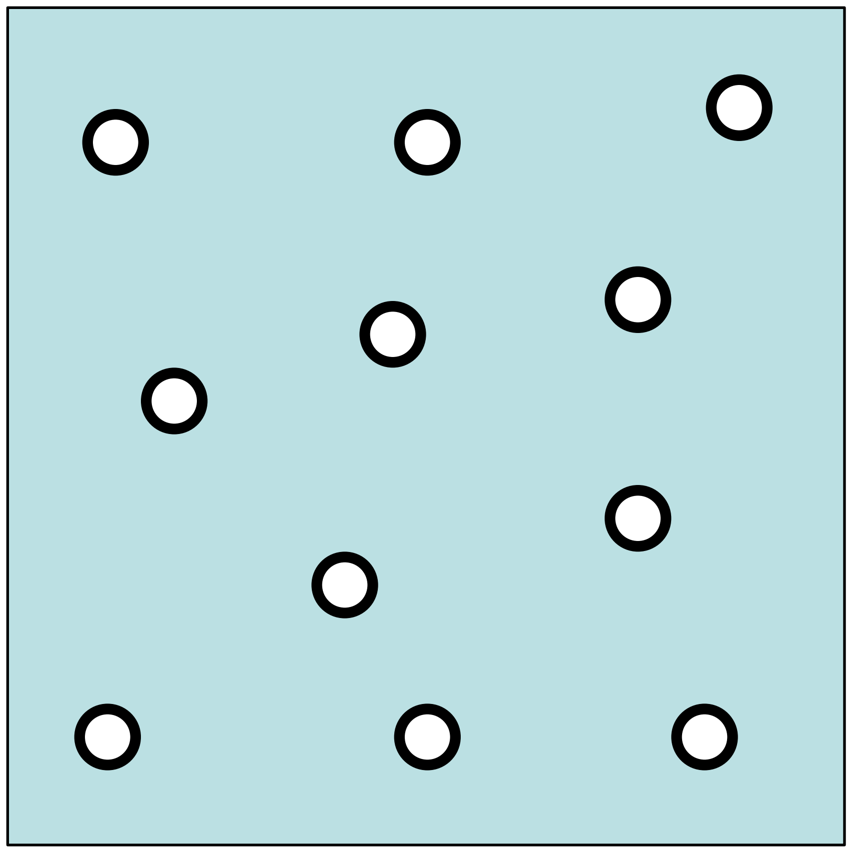


Related Idea: Multisampling

- Problem
  - Shading is very expensive today (complicated shaders)
  - Full supersampling has linear cost in #samples (k*k)
- 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*width, k*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

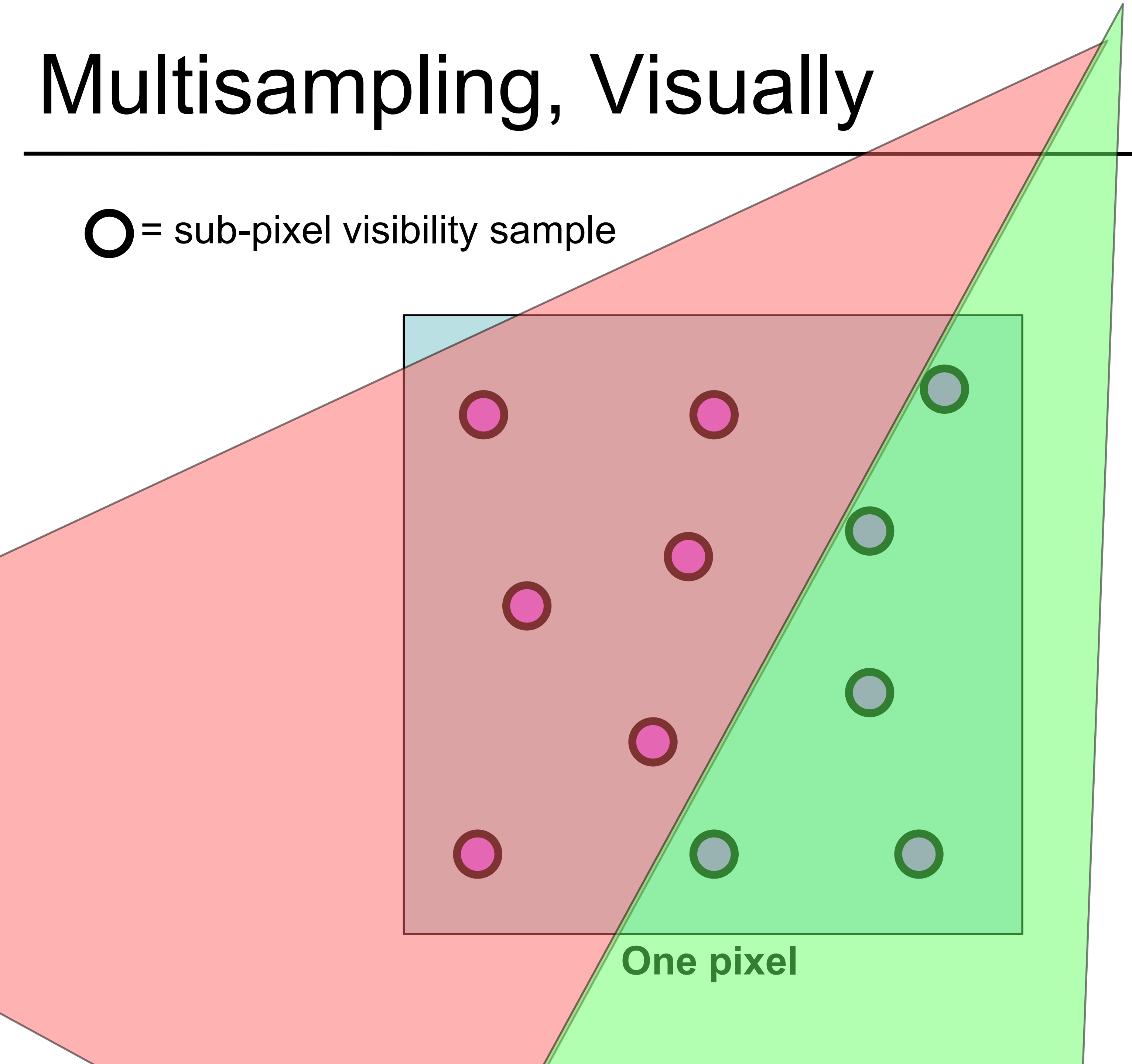
Multisampling, Visually

= sub-pixel visibility sample

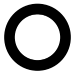


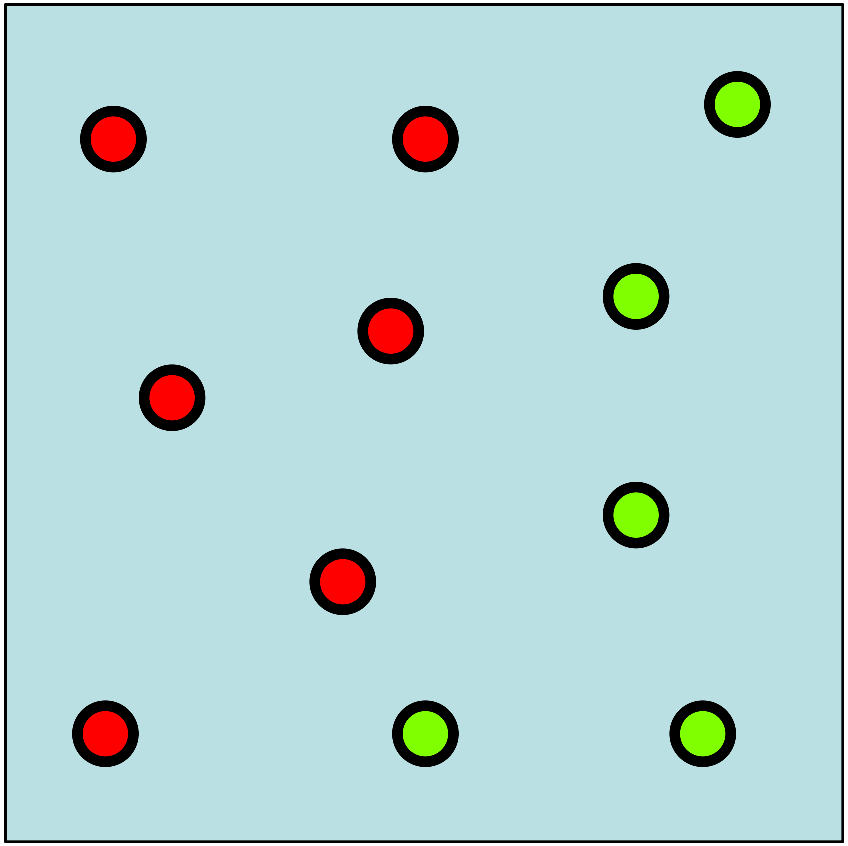
One pixel

Multisampling, Visually



Multisampling, Visually

 = sub-pixel visibility sample

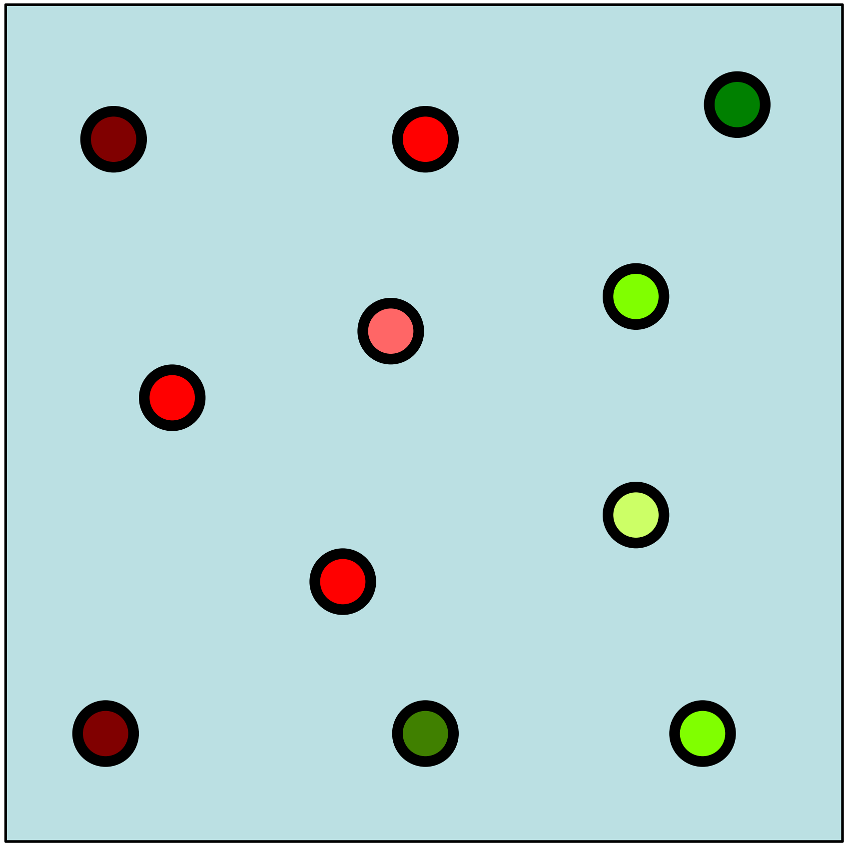


One pixel

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



One pixel

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
```

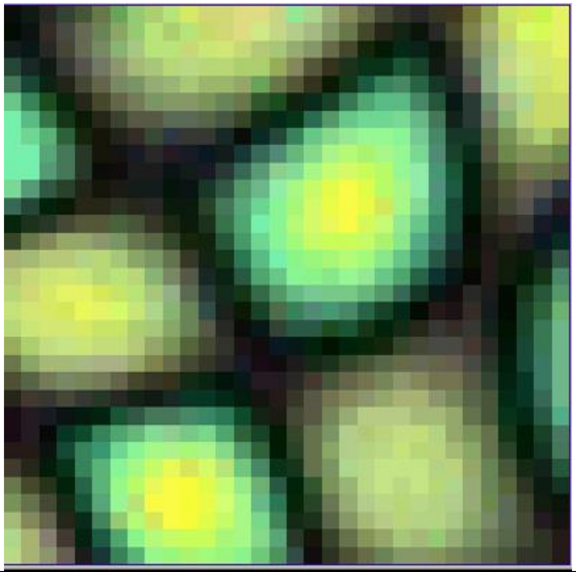
Multisampling vs. Supersampling

- 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
- But Why?
  - Visibility edges are where supersampling really works
  - Shading can be prefiltered more easily than visibility
- This is how GPUs perform antialiasing these days

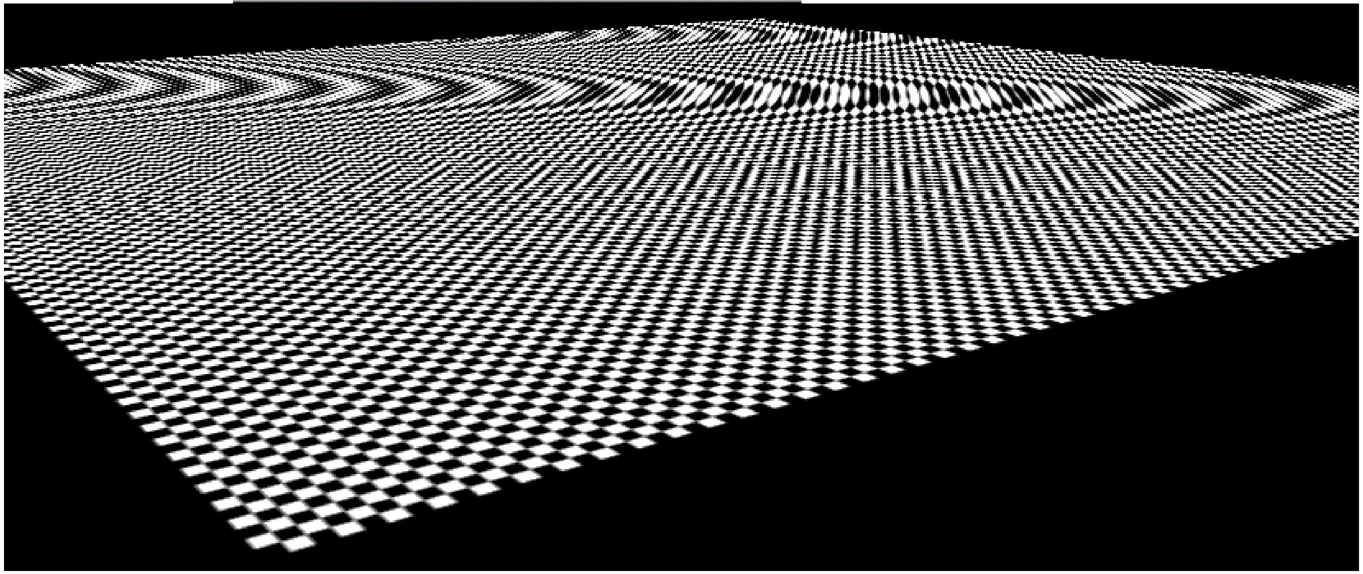
Questions?

Examples of Texture Aliasing

Magnification

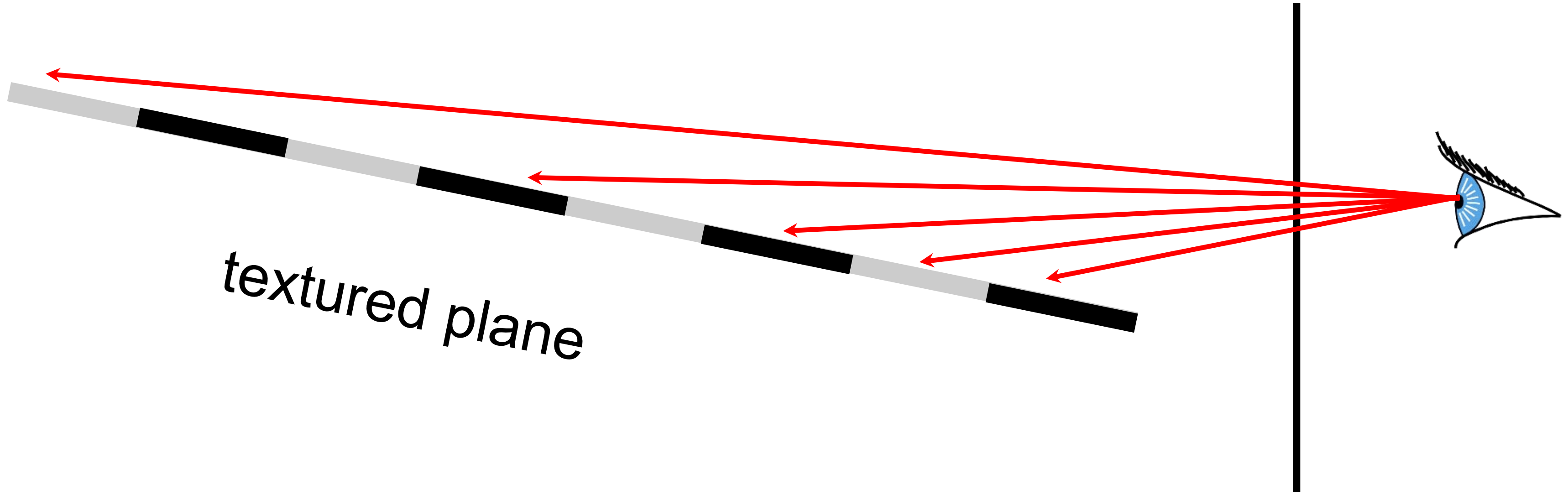


Minification



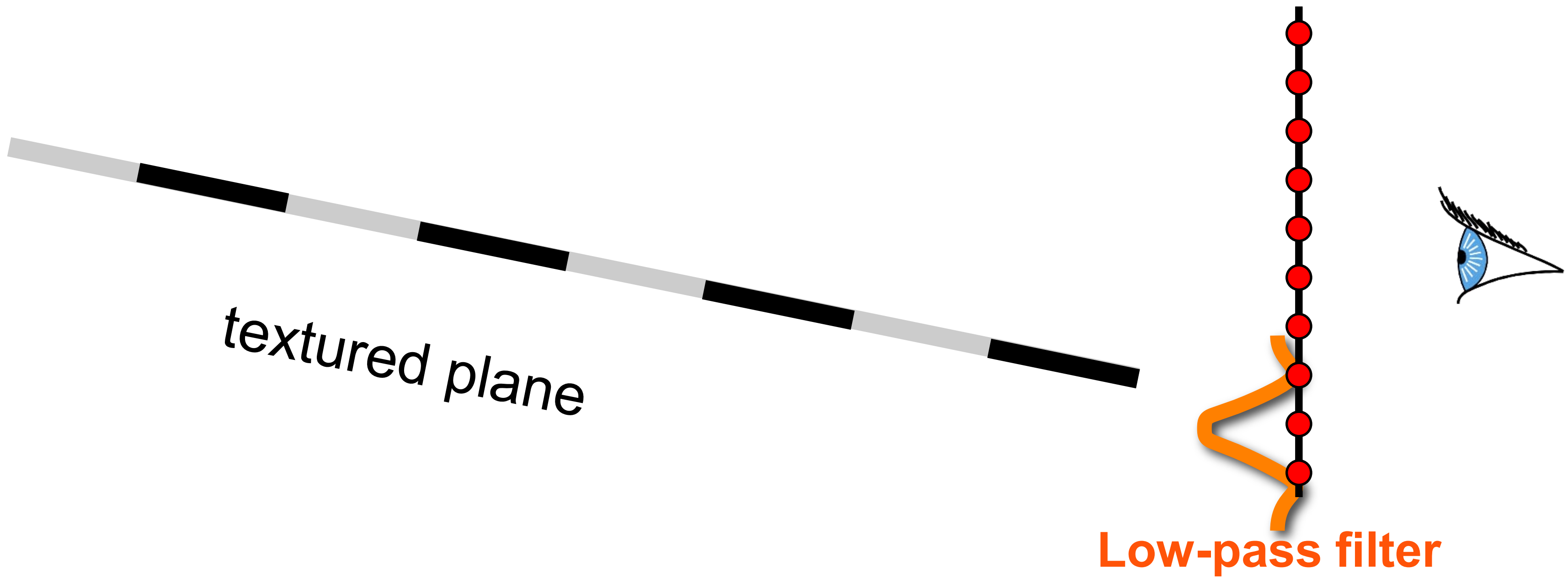
Texture Filtering

- Problem: Prefiltering is impossible when you can only take point samples
  - This is why visibility (edges) need supersampling
- Texture mapping is simpler
  - Imagine again we are looking at an infinite textured plane



Texture Filtering

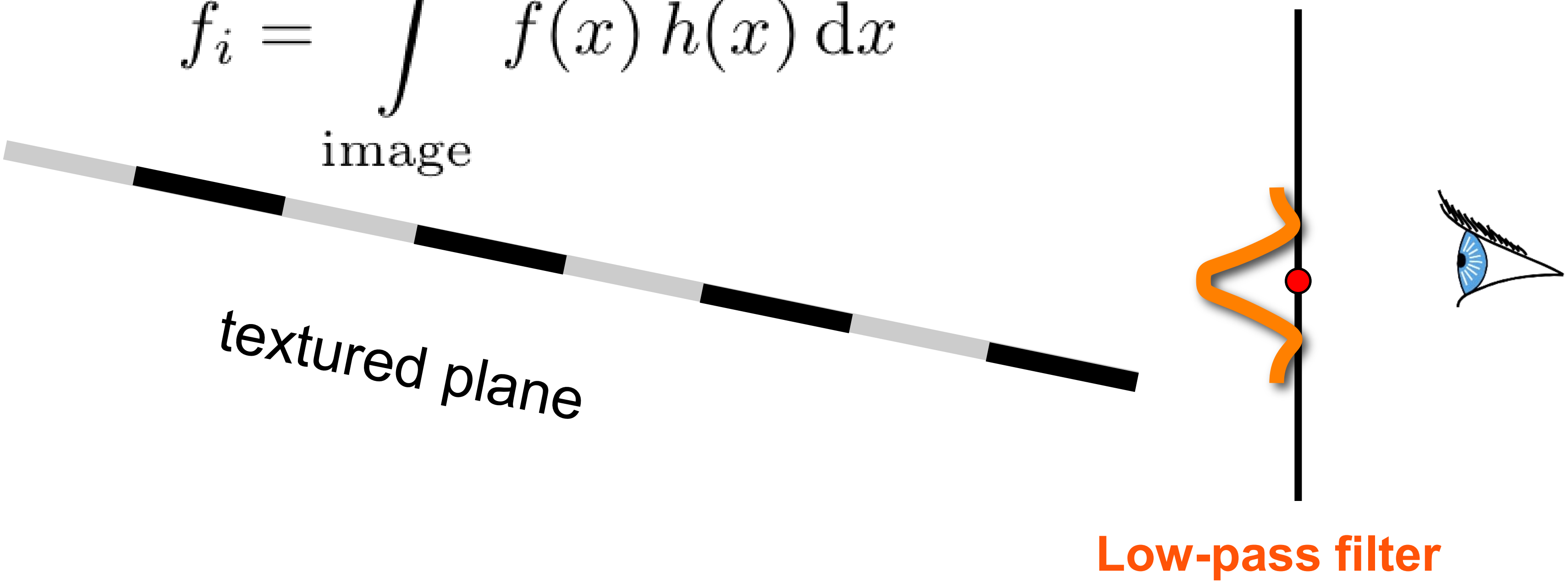
- We should pre-filter image function *before sampling*
  - That means blurring the image function with a low-pass filter (convolution of image function and filter)



Texture Filtering

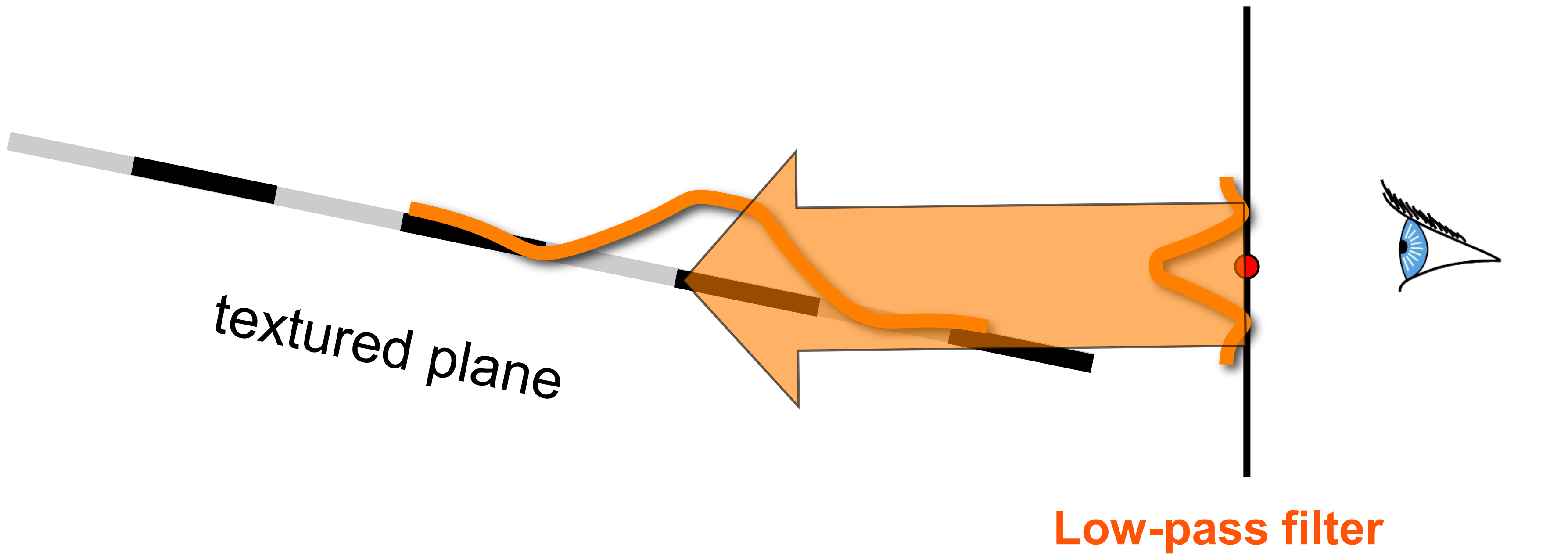
- We can combine low-pass and sampling
  - The value of a sample is the integral of the product of the image f and the filter h centered at the sample location
    - "A local average of the image f weighted by the filter h"

$$\hat{f}_i = \int_{\text{image}} f(x)\, h(x)\, \mathrm{d}x$$



Texture Filtering

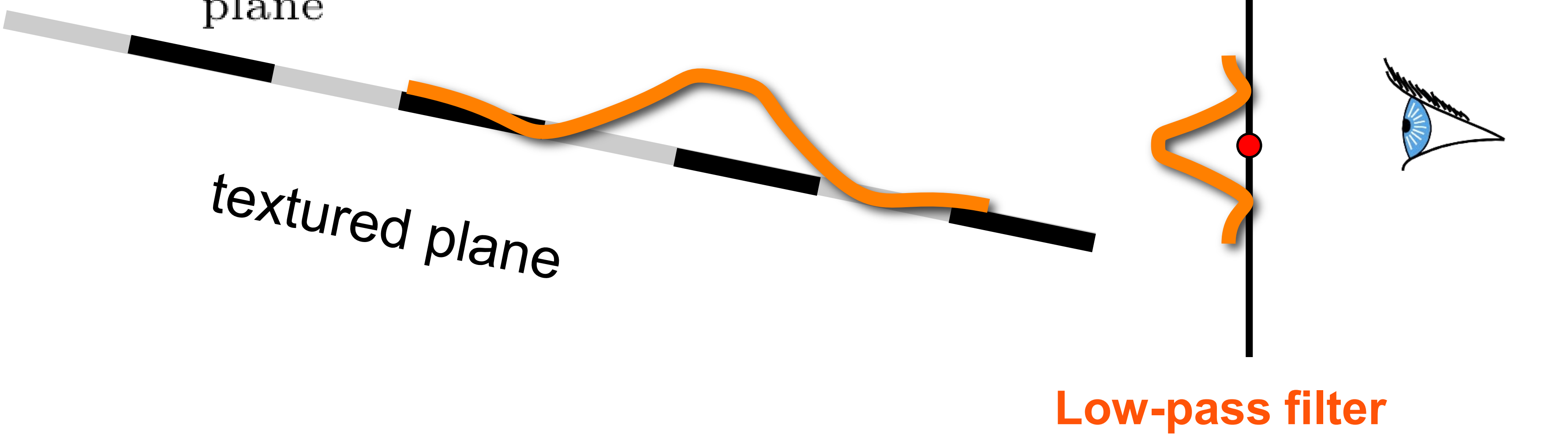
- Well, we can just as well change variables and compute this integral *on the textured plane instead*
  - In effect, we are projecting the pre-filter onto the plane



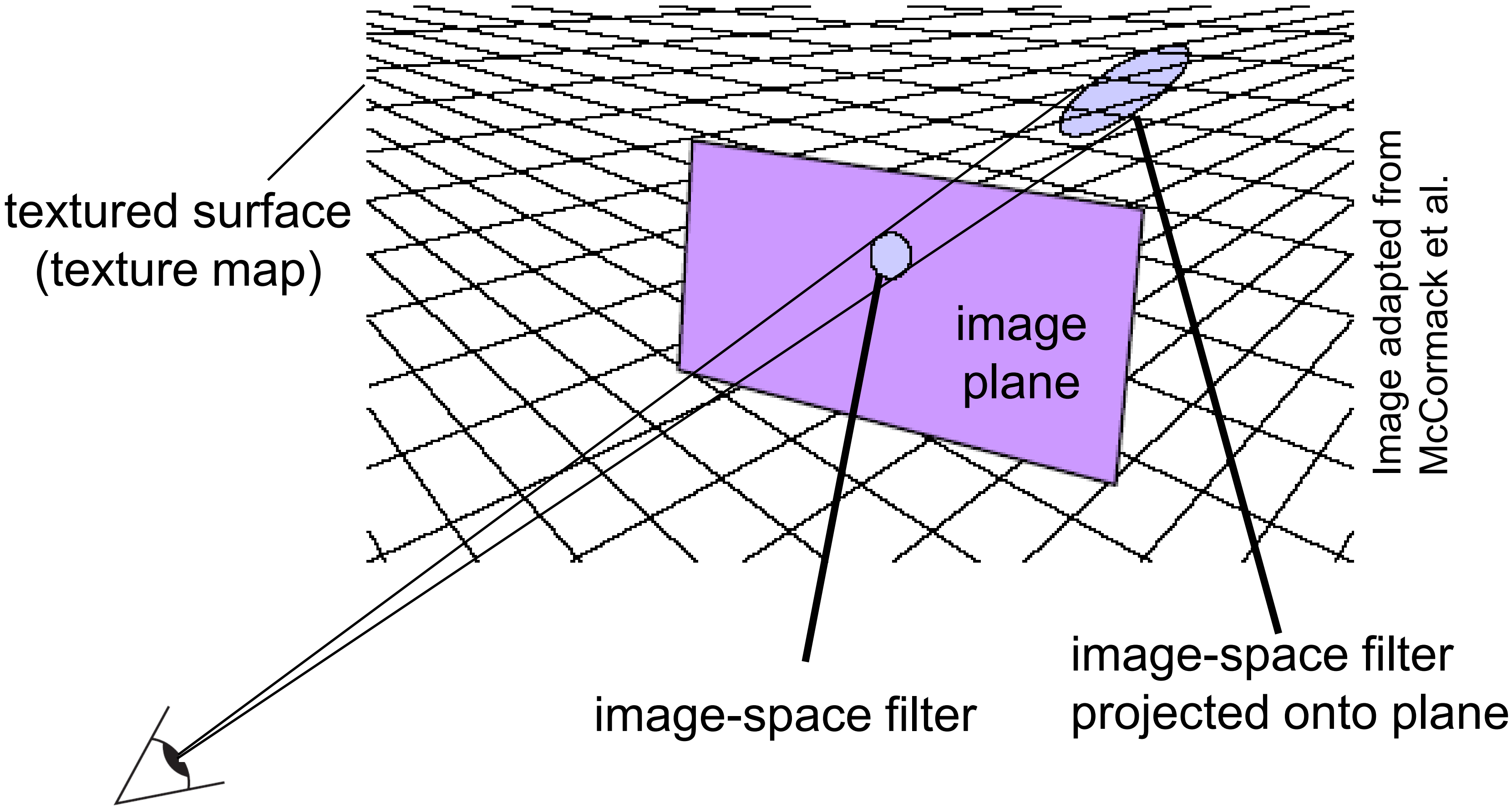
Texture Filtering

- 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

$$\hat{f}_i = \int_{\text{plane}} f(x')\, h(x')\, \left|J(x, x')\right| \mathrm{d}x'$$



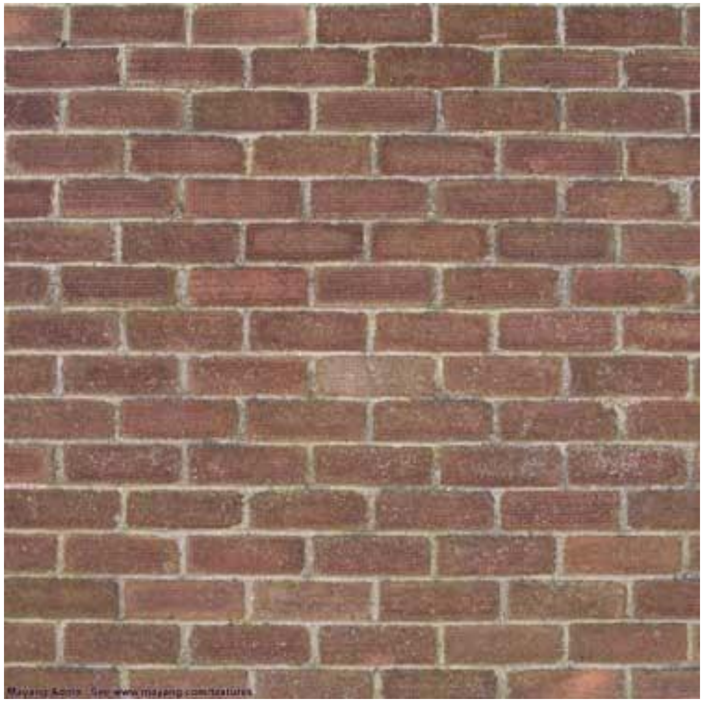
Texture Pre-Filtering, Visually



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

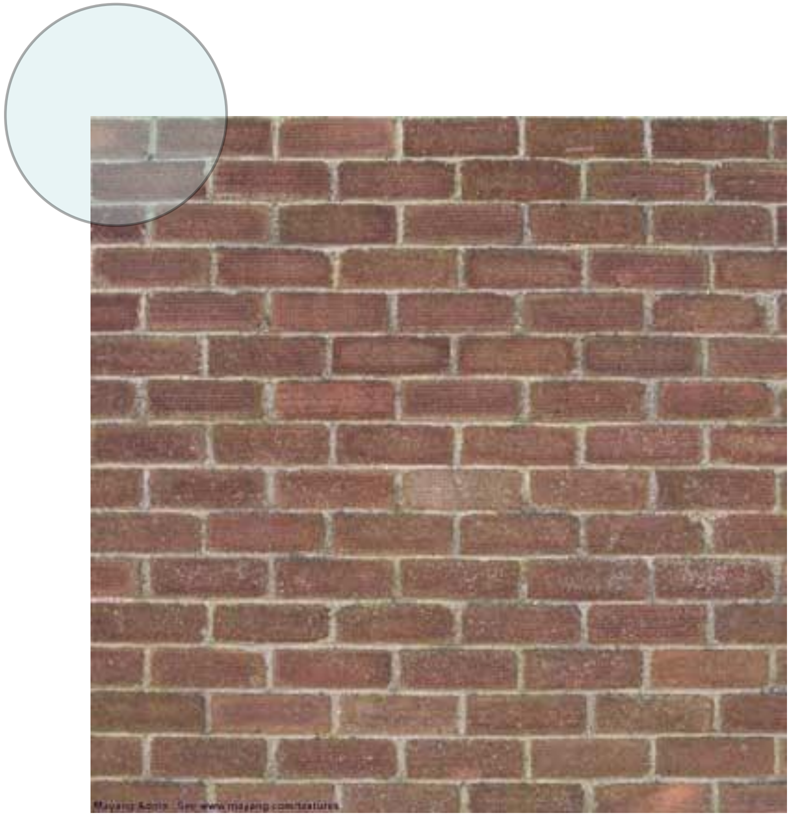
Solution: Precomputation

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



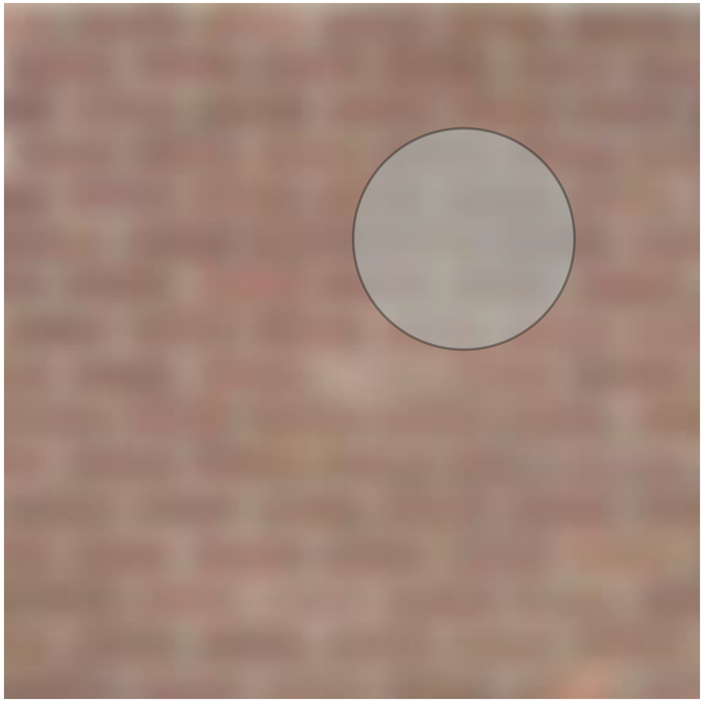
Solution: Precomputation

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



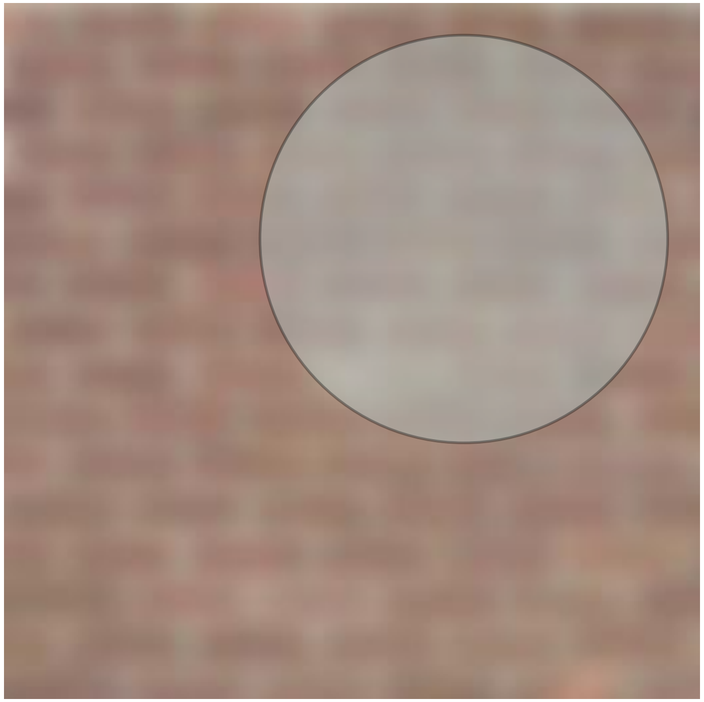
Solution: Precomputation

- 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



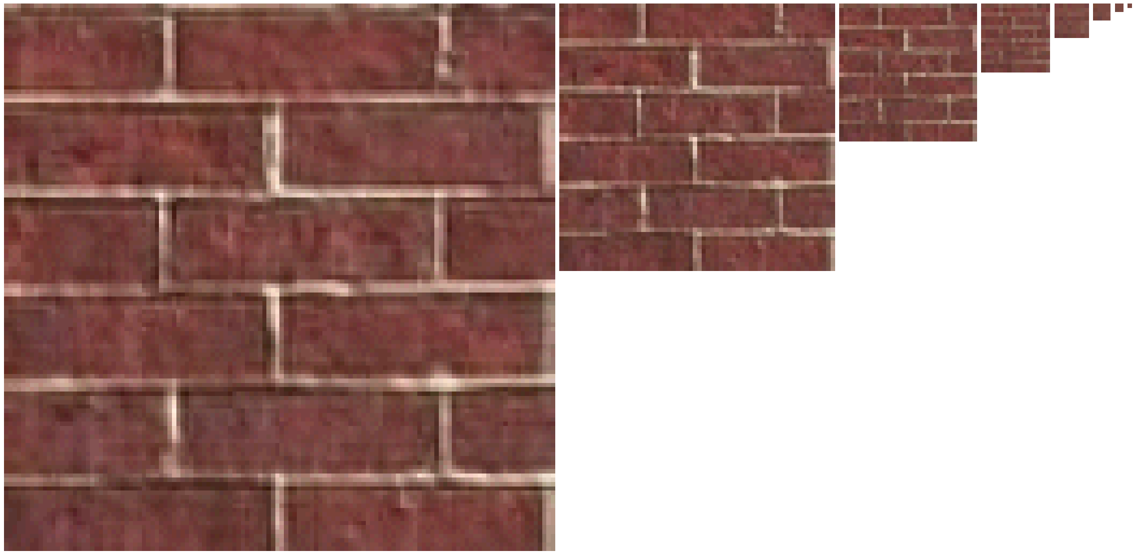
Solution: Precomputation

- 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

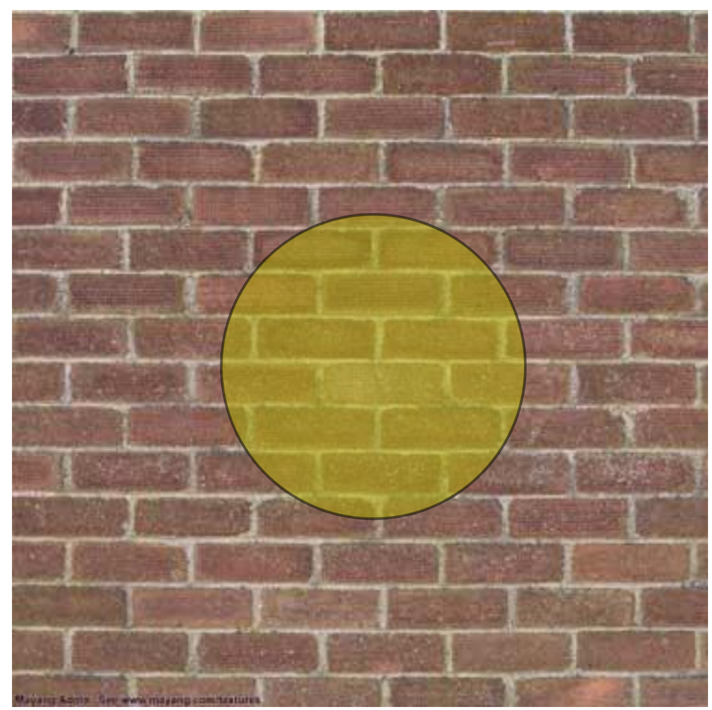


- During rasterization we compute the index of the decimated image that is sampled at a rate closest to the density of our desired sampling rate
- MIP stands for *multum in parvo* which means *many in a small place*

MIP-Mapping

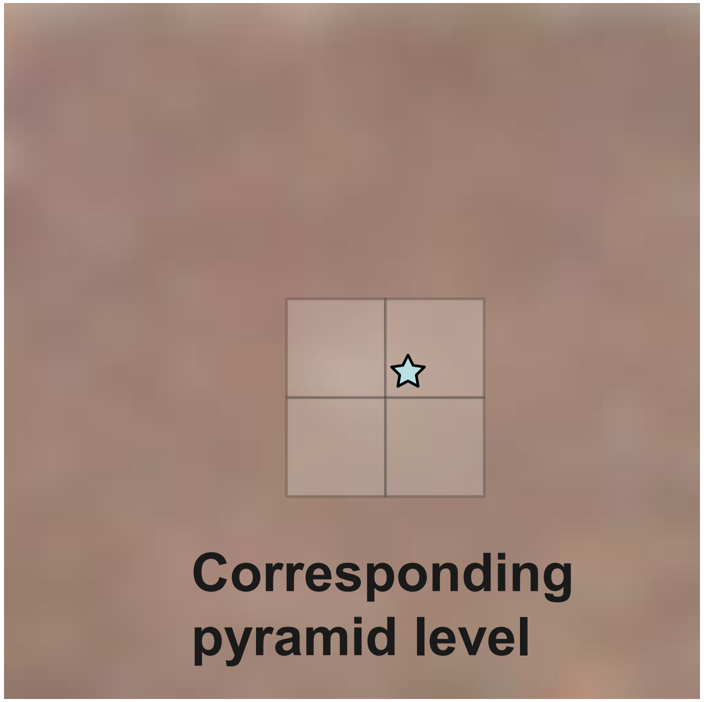
- When a pixel wants an integral of the pre-filtered texture, we must find the “closest” results from the precomputed MIP-map pyramid
  - Must compute the “size” of the projected pre-filter in the texture UV domain

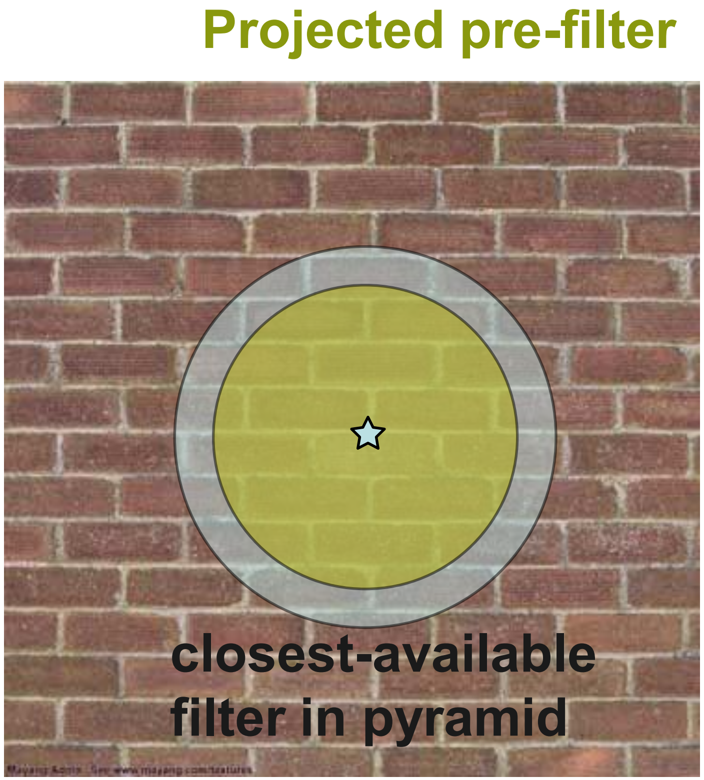
Projected pre-filter



MIP-Mapping

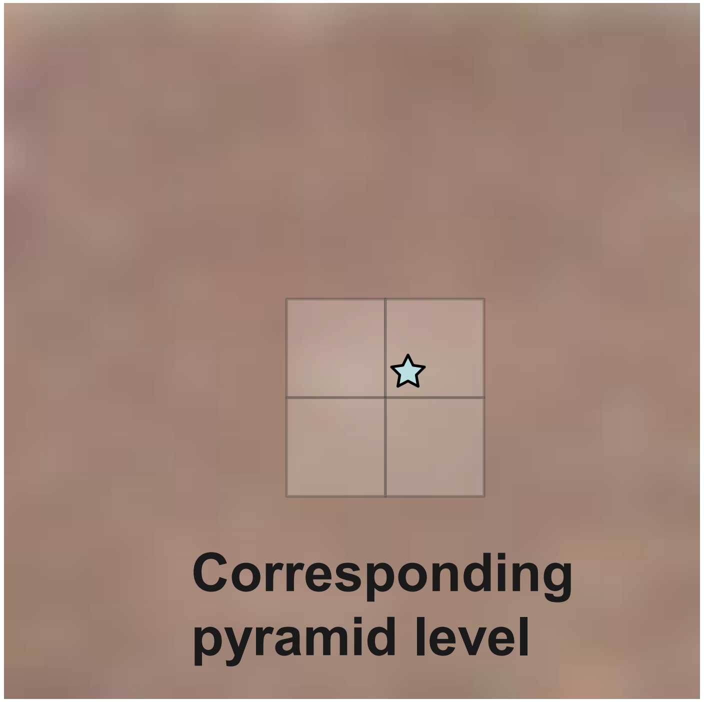
- Simplest method: Pick the scale closest, then do usual reconstruction on that level (e.g. bilinear between 4 closest texture pixels)

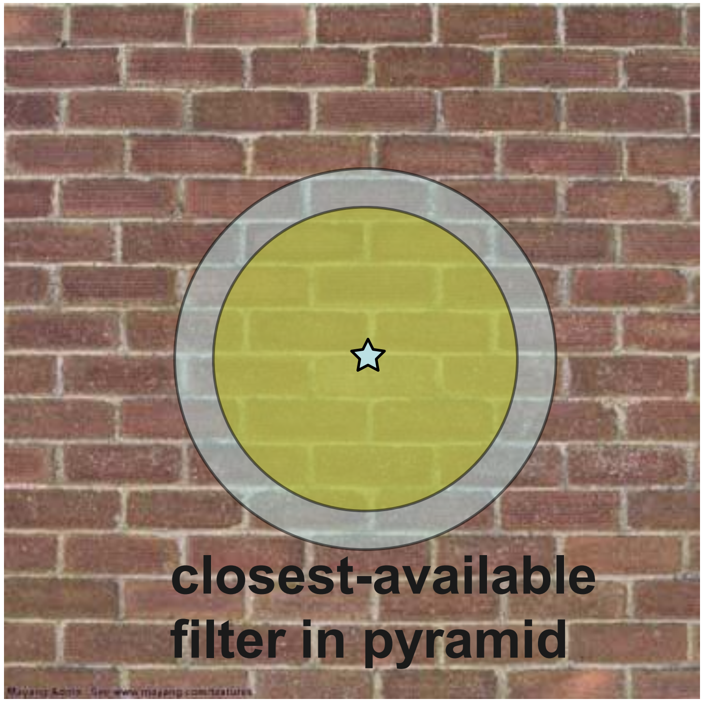




MIP-Mapping

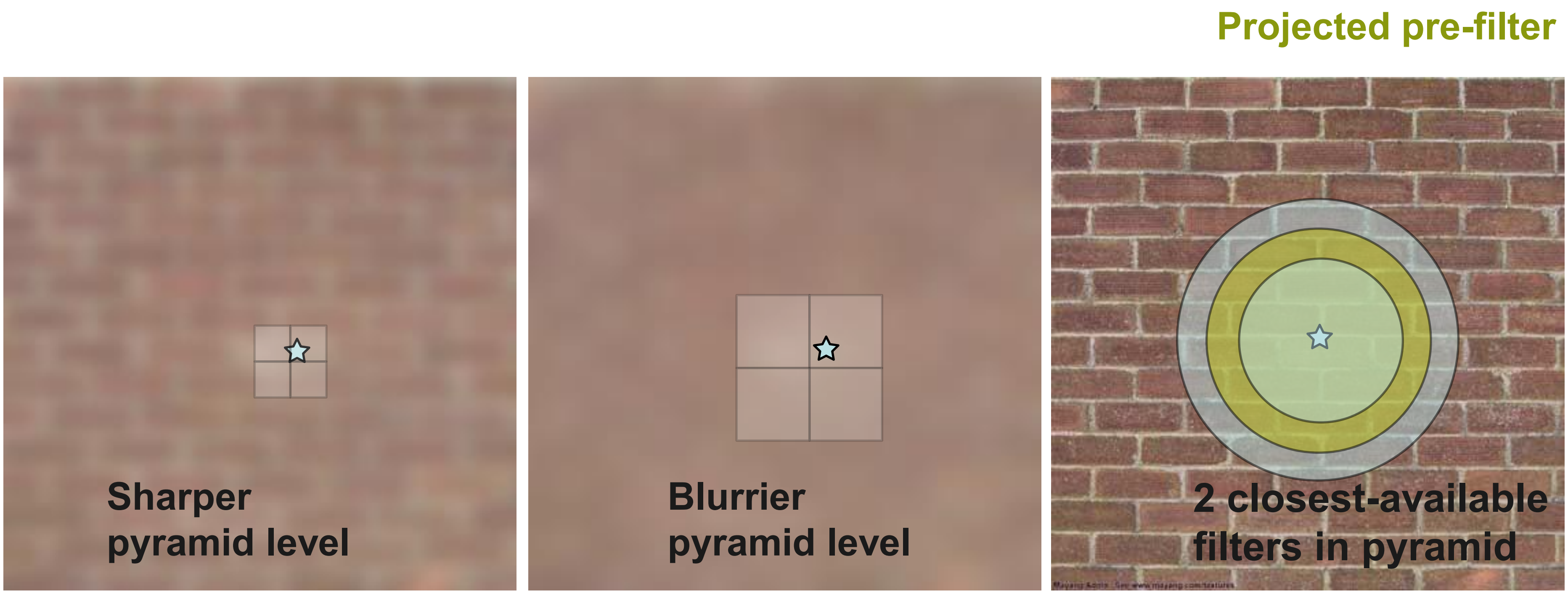
- Simplest method: Pick the scale closest, then do usual reconstruction on that level (e.g. bilinear between 4 closest texture pixels)
- Problem: discontinuity when switching scale





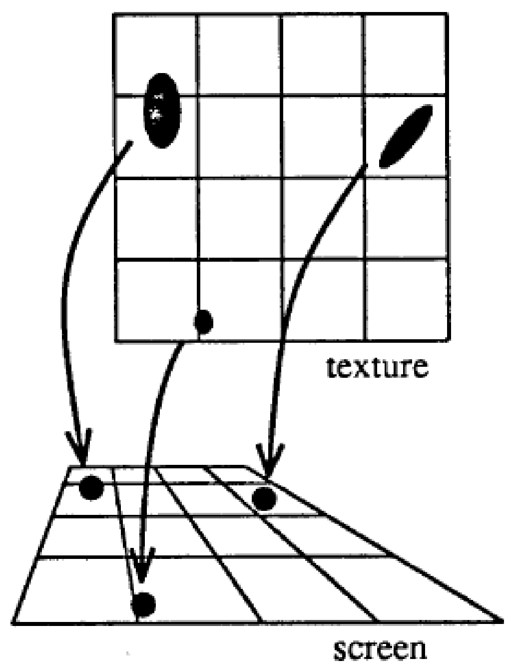
Tri-Linear MIP-Mapping

- Use **two** closest scales,
  compute reconstruction results from both,
  and linearly interpolate between them

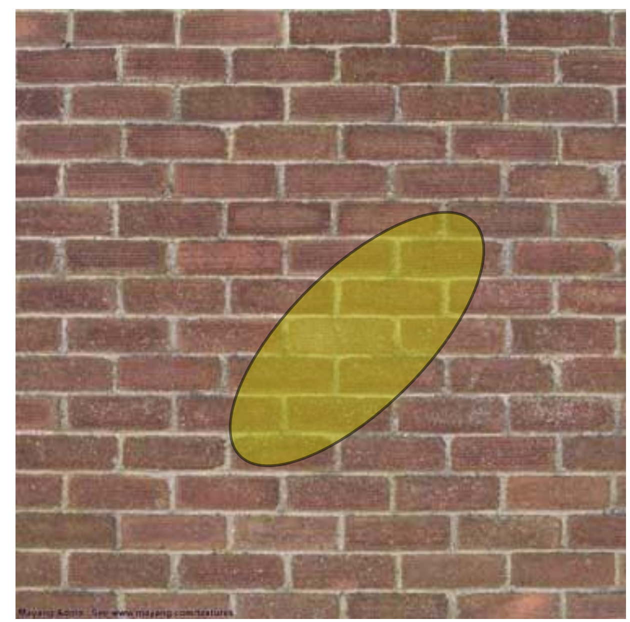


Tri-Linear MIP-Mapping

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



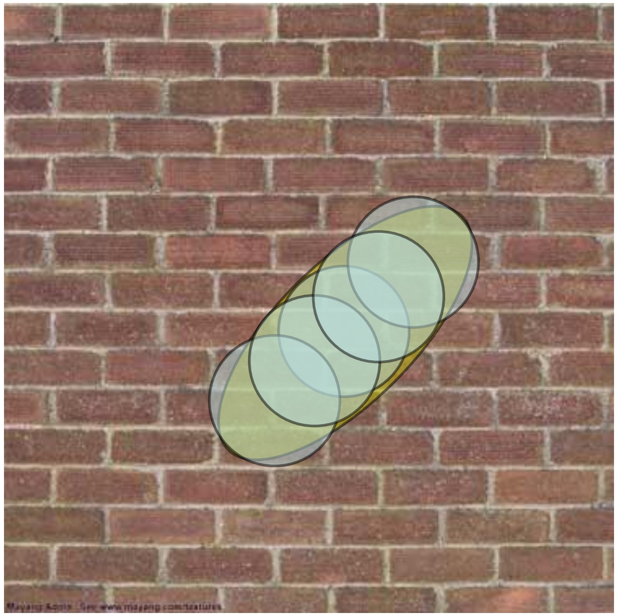
Projected pre-filter



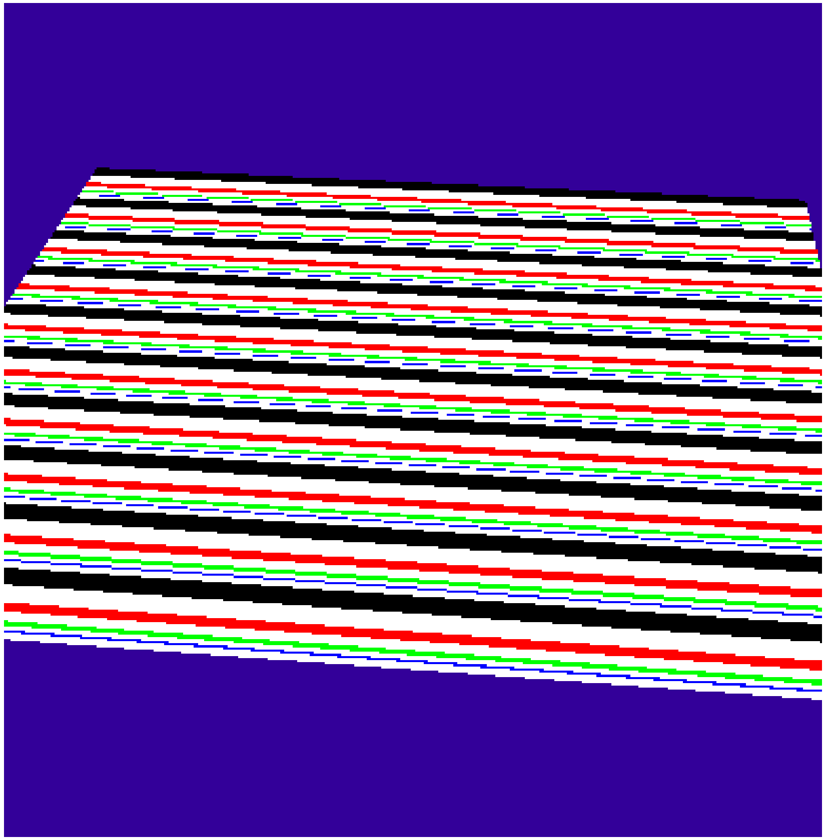
Anisotropic filtering

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

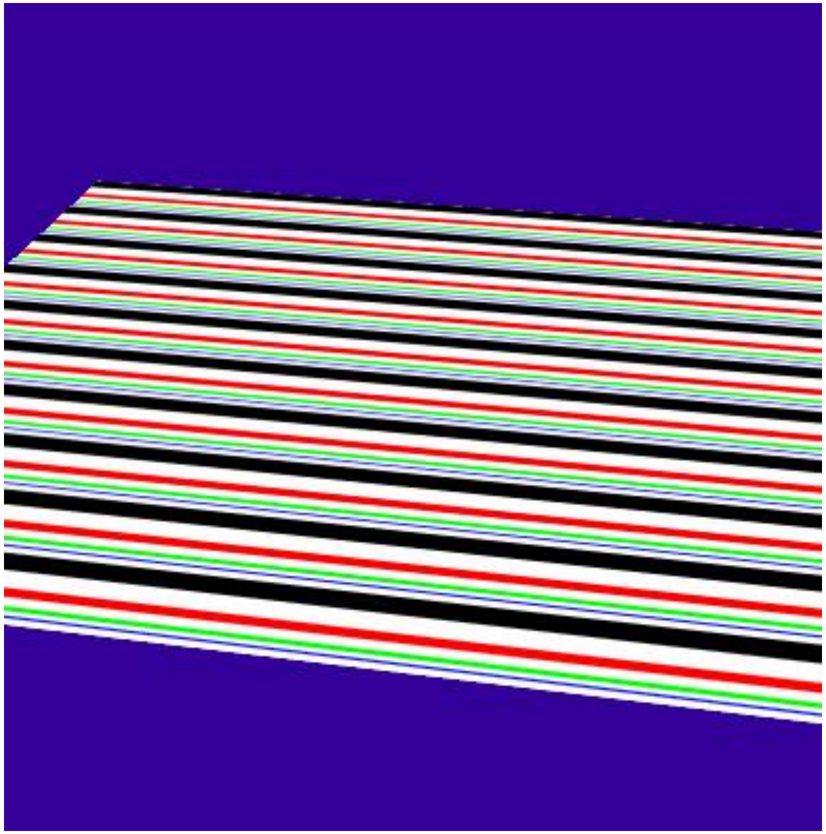
Projected pre-filter



MIP Mapping Example

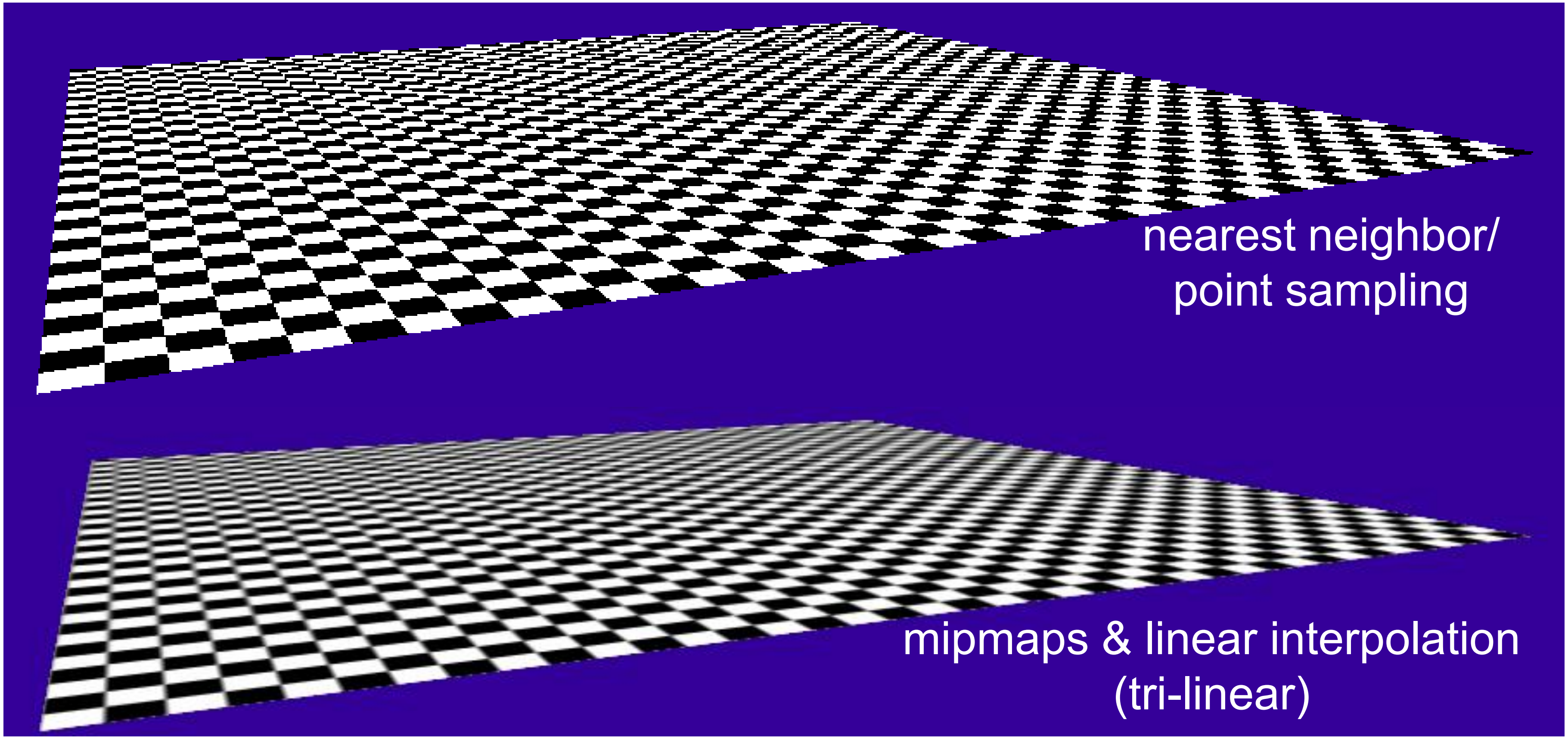


Nearest Neighbor



MIP Mapped (Tri-Linear)

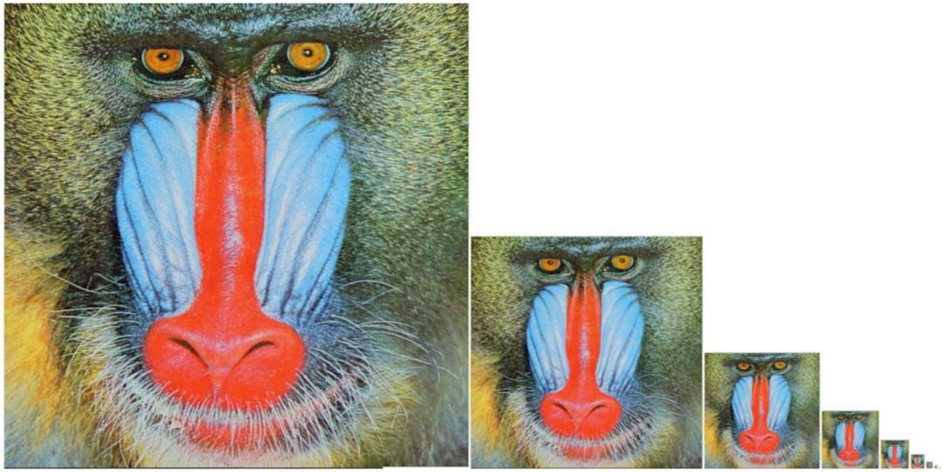
MIP Mapping Example

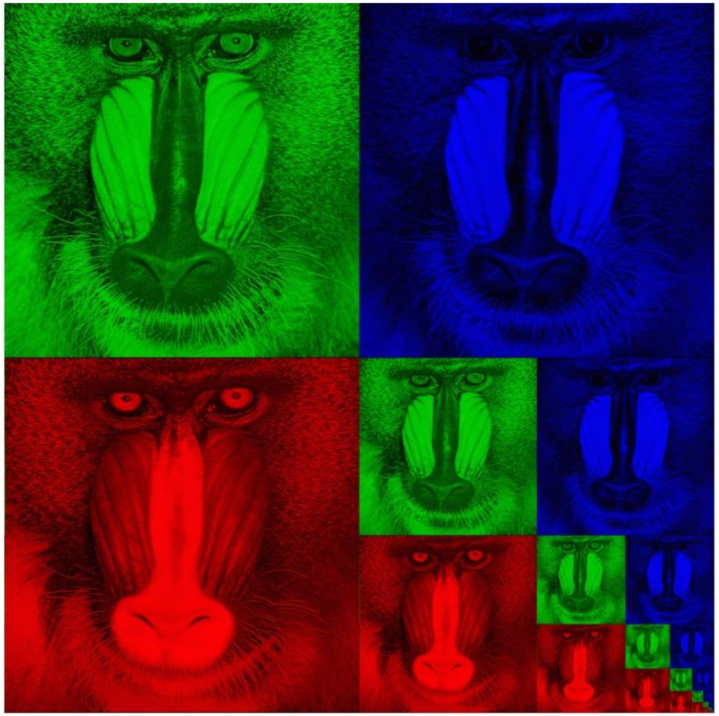


Questions

Storing MIP Maps

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

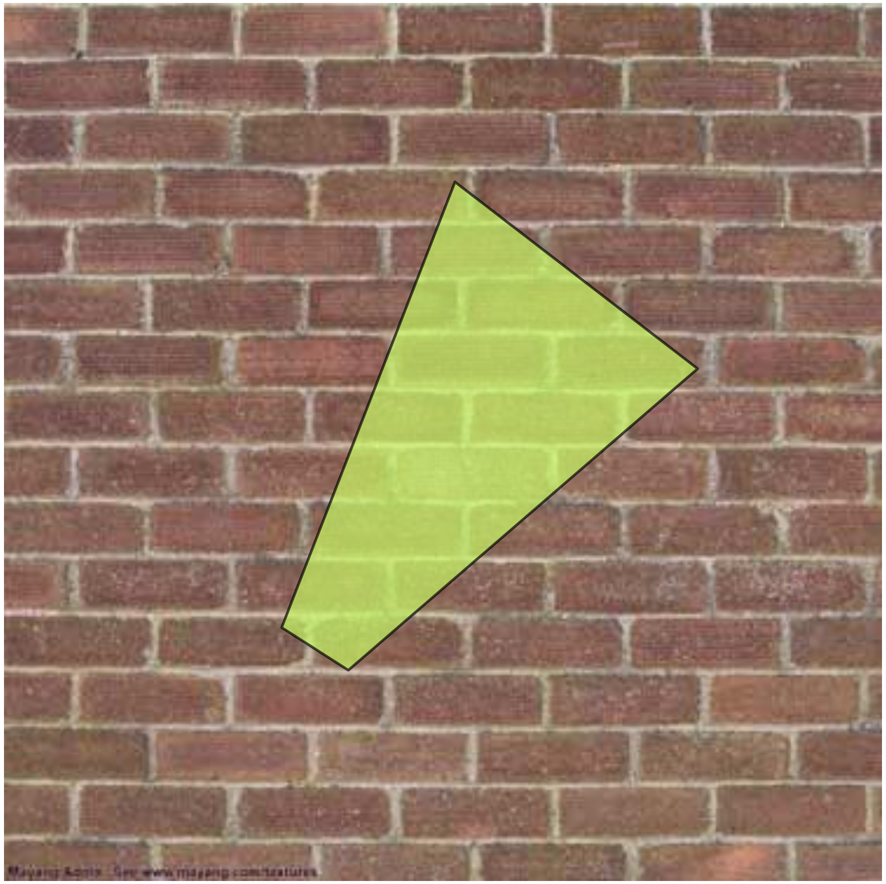




Finding the MIP Level

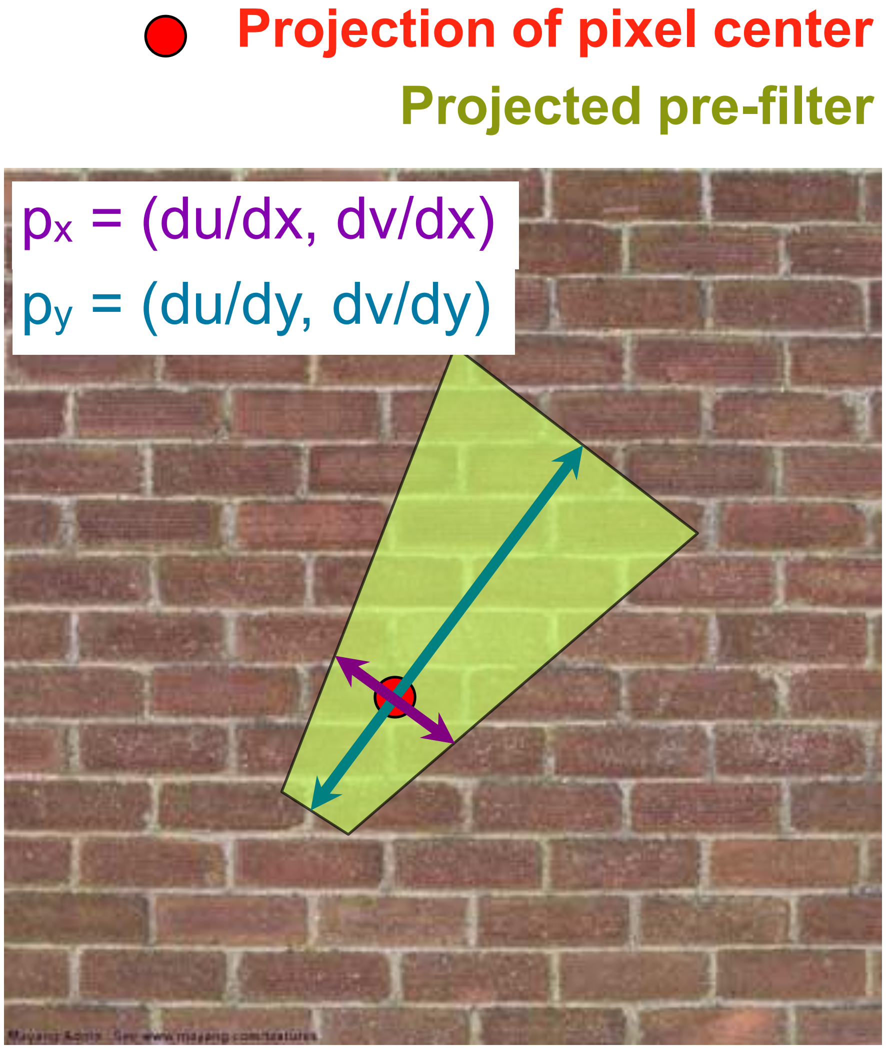
- Often we think of the pre-filter as a box
  - What is the projection of the square pixel “window” in texture space?

Projected pre-filter



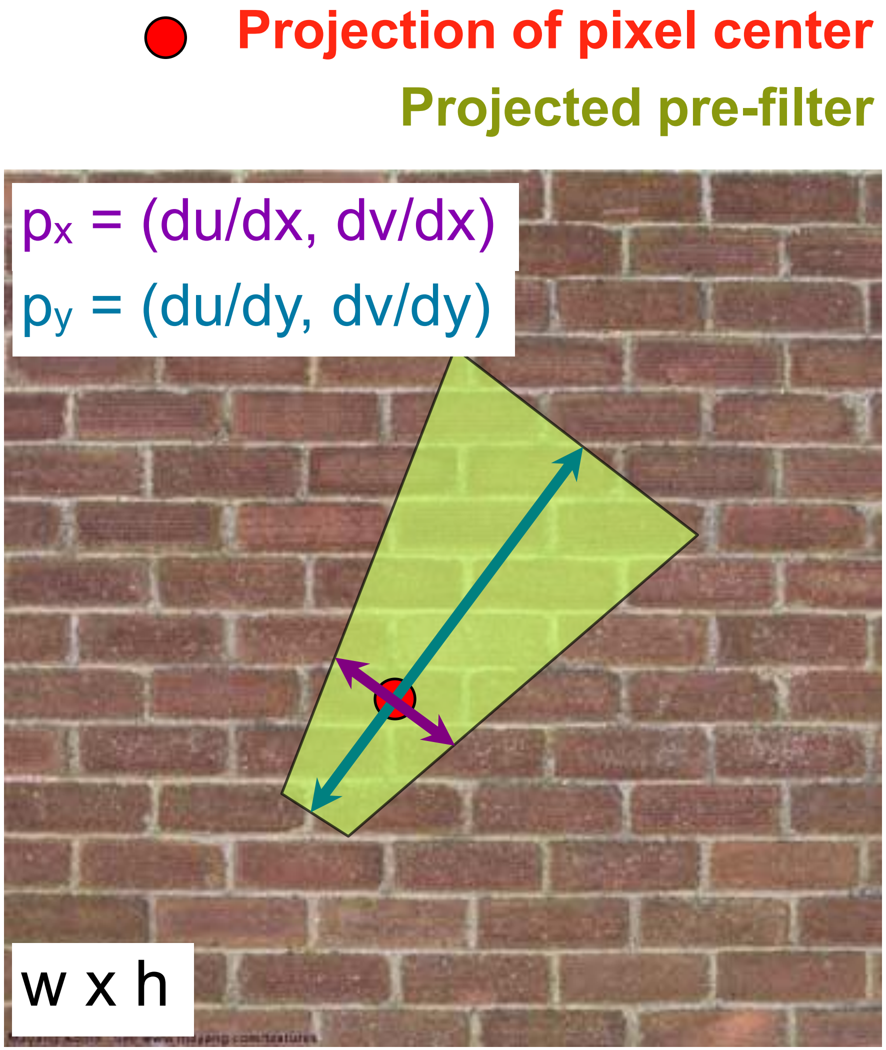
Finding the MIP Level

- 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)



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
    $$\log_2 \sqrt{(w|p_x|)^2 + (h|p_y|)^2}$$

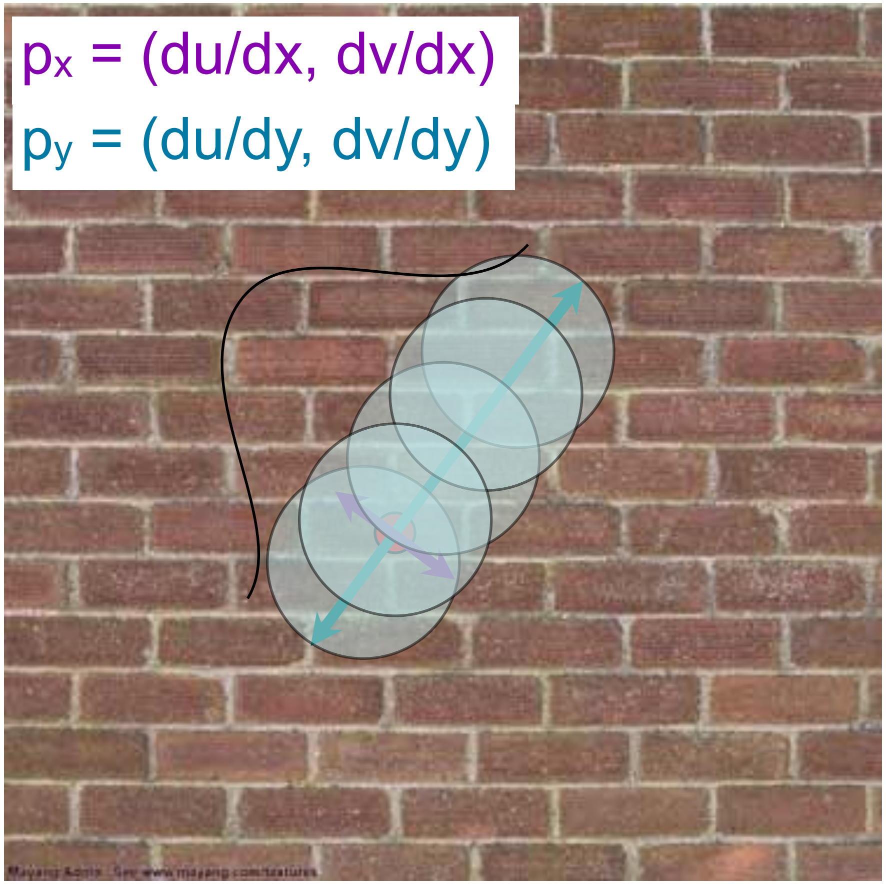


Anisotropic filtering

- Pick levels according to smallest partial
  - well, actually max of the 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*

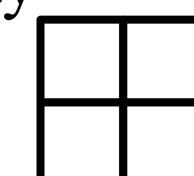
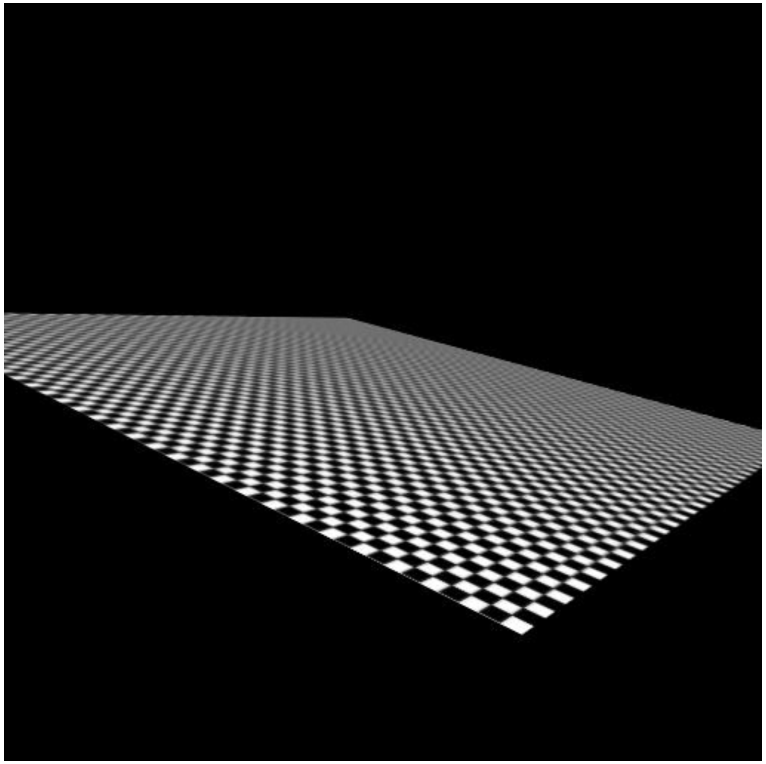
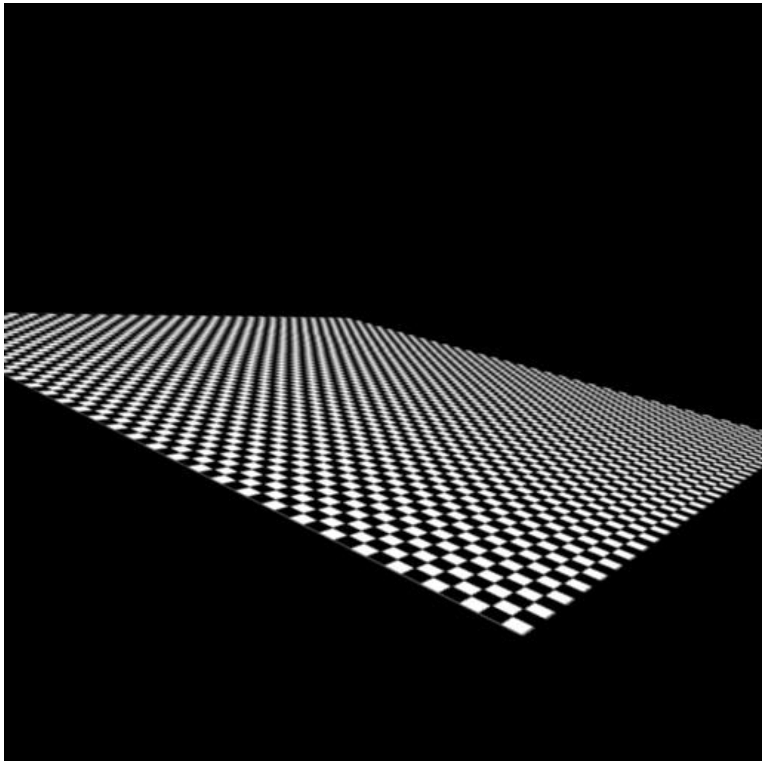


Image Quality Comparison



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

Arf!

Questions?

Image removed due to copyright restrictions.

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

http://xkcd.com/386/

Image removed due to copyright restrictions – please see the link above for further details.

Bad example

Image removed due to copyright restrictions -- please see https://blogs.intel.com/intellabs/2007/10/10/real_time_raytracing_the_end_o/ for further details.

Ray-triangle intersection

- Triangle ABC
- Ray O+t*D
- Barycentric coordinates *α*, *β*, *γ*
- Ray-triangle intersection

$$P(t) = O + t * D = A + \beta AB + \gamma AC$$

- or in matrix form

$$(-AB \;\; -AC \;\; D) \begin{pmatrix} \beta \\ \gamma \\ t \end{pmatrix} = (OA)$$

Ray-triangle

$$(-AB \;\; -AC \;\; D)\begin{pmatrix} \beta \\ \gamma \\ t \end{pmatrix} = (OA)$$

- Cramer's rule (where | | is the determinant)

$$\beta = \frac{|OA \;\; -AC \;\; D|}{|M|}$$

$$\gamma = \frac{|-AB \;\; OA \;\; D|}{|M|}$$

$$t = \frac{|-AB \;\; -AC \;\; OA|}{|M|}$$

Determinant

- Cross product and dot product
- i.e., for a matrix with 3 columns vectors: M=UVW

$$|M| = U \times V \cdot W$$

Back to ray-triangle

$$\underset{\text{M}}{\left(-AB \quad -AC \quad D\right)} \begin{pmatrix} \beta \\ \gamma \\ t \end{pmatrix} = (OA)$$

$$\beta = \frac{|OA \quad -AC \quad D|}{|M|}$$

$$\gamma = \frac{|-AB \quad OA \quad D|}{|M|}$$

$$t = \frac{|-AB \quad -AC \quad OA|}{|M|}$$

$$\begin{aligned} \Delta_M &= BA \times CA \cdot D \\ \Delta_\beta &= OA \times CA \cdot D \\ \Delta_\gamma &= BA \times OA \cdot D \\ \Delta_t &= BA \times CA \cdot OA \end{aligned}$$

Ray-triangle recap

$$\begin{aligned}
\Delta_M &= BA \times CA \cdot D \\
\Delta_\beta &= OA \times CA \cdot D \\
\Delta_\gamma &= BA \times OA \cdot D \\
\Delta_t &= BA \times CA \cdot OA
\end{aligned}$$

- And $\beta = \Delta_\beta / \Delta_M$

$$\gamma = \Delta_\gamma / \Delta_M$$

$$t = \Delta_t / \Delta_M$$

- Intersection if $0 \leq \beta \leq 1 \quad 0 \leq \gamma \leq 1$

Rasterization

- Viewpoint is known and fixed
- Let's extract what varies per pixel

$$
\begin{aligned}
\Delta_M &= BA \times CA \cdot D \\
\Delta_\beta &= OA \times CA \cdot D \\
\Delta_\gamma &= BA \times OA \cdot D \\
\Delta_t &= BA \times CA \cdot OA
\end{aligned}
$$

- Only D!

Rasterization

$$\begin{aligned} \Delta_M &= Eq_M \cdot D \\ t &= \Delta_t / \Delta_M \\ \beta &= Eq_\beta \cdot D / \Delta_M \\ \gamma &= Eq_\gamma \cdot D / \Delta_M \end{aligned}$$

- Cache redundant computation independent of D:

$$\begin{aligned} Eq_M &= BA \times CA \\ Eq_\beta &= OA \times CA \\ Eq_\gamma &= BA \times OA \\ \Delta_t &= BA \times CA \cdot OA \end{aligned}$$

Equivalent to the setup of edge equations and interpolants in rasterization

- And for each pixel

$$\begin{aligned} \Delta_M &= BA \times CA \cdot D \\ \Delta_\beta &= OA \times CA \cdot D \\ \Delta_\gamma &= BA \times OA \cdot D \\ \Delta_t &= BA \times CA \cdot OA \end{aligned}$$

Per-pixel calculation of edge equations and z (=t)

Conclusions

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

Ray casting (Python)

```
def intersectWithBarycentric (self, triangle, orig, D):
          detM=triangle.BA.cross(triangle.CA)*D
          if fabs(detM)<epsilon: return False, 0
          OA=triangle.A-orig
          detBeta=OA.cross(triangle.CA)*D
          beta=detBeta/detM
          detGamma=triangle.BA.cross(OA)*D
          gamma=detGamma/detM
          detT=triangle.BA.cross(triangle.CA)*OA
          t=detT/detM
          if beta<-epsilon or gamma<-epsilon or beta+gamma>1+epsilon:
               return False, 0
        else: return True, t
```

$$\Delta_M = BA \times CA \cdot D$$
$$\Delta_\beta = OA \times CA \cdot D$$
$$\Delta_\gamma = BA \times OA \cdot D$$
$$\Delta_t = BA \times CA \cdot OA$$

$$\beta = \Delta_\beta / \Delta_M$$
$$\gamma = \Delta_\gamma / \Delta_M$$
$$t = \Delta_t / \Delta_M$$

Ray Casting

```
def intersectWithBarycentric (self, triangle, orig, D):
        detM=triangle.BA.cross(triangle.CA)*D
        if fabs(detM)<epsilon: return False, 0
        OA=triangle.A-orig
        detBeta=OA.cross(triangle.CA)*D
        beta=detBeta/detM
        detGamma=triangle.BA.cross(OA)*D
        gamma=detGamma/detM
        detT=triangle.BA.cross(triangle.CA)*OA
        t=detT/detM
        if beta<-epsilon or gamma<-epsilon or beta+gamma>1+epsilon:
            return False, 0
      else: return True, t
```

```
def intersectWithBarycentric (self, triangle, orig, D):
          detM=triangle.BA.cross(triangle.CA)*D
          if fabs(detM)<epsilon: return False, 0
          OA=triangle.A-orig
          detBeta=OA.cross(triangle.CA)*D
          beta=detBeta/detM
          detGamma=triangle.BA.cross(OA)*D
          gamma=detGamma/detM
          detT=triangle.BA.cross(triangle.CA)*OA
          t=detT/detM
          if beta<-epsilon or gamma<-epsilon or beta+gamma>1+epsilon:
               return False, 0
        else: return True, t
```

Ray Casting

```
def setUpTriangle (self, triangle, orig):
        self.detMEq=triangle.BA.cross(triangle.CA)
        OA=triangle.A-orig
        self.detBetaEq=OA.cross(triangle.CA)
        self.detGammaEq=triangle.BA.cross(OA)
        self.detT=triangle.BA.cross(triangle.CA)*OA
```

Raterization setup

```
def testPixel(self, D):
        detM=self.detMEq*D
        if fabs(detM)<epsilon: return False, 0
        detBeta= self.detBetaEq*D
        beta=detBeta/detM
        detGamma=self.detGammaEq*D
        gamma=detGamma/detM
        t=self.detT/detM
        if beta<-epsilon or gamma<-epsilon or beta+gamma>1+epsilon:
              return False, 0
        else: return True, t
```

Raterization per pixel

Main loops

```
def raycast(scene, width, height):
    im=Image.new('RGB',(width,height))
    for y in range(height):
        for x in range(width):
            dir=vec3(2.0*x/width-1.0, 1.0-2.0*y/height, 1.0)
            tmin=infinity
            for T in scene.triangles:
                test, t=inter.intersectWithBarycentric (triangle, orig, dir)
                if test and t>0 and t<tmin:
                    im.putpixel((x,y), T.shade())
                    tmin=t
    return im
```

Ray generation

Ray intersection

t test

```
def rasterize(scene, width, height):
    im=Image.new('RGB',(width,height))
    tmin = [[infinity for col in range(width)] for row in range(height)]
    for T in scene.triangles:
        inter.setUpTriangle(T, orig)
        for y in range(height):
            for x in range(width):
                dir=vec3(2.0*x/width-1.0, 1.0-2.0*y/height, 1.0)
                test, t=inter.testPixel(dir)
                if test and t>0 and t<tmin[x][y]:
                    im.putpixel((x,y), T.shade())
                    tmin[x][y]=t
    return im
```

z buffer initialization

edge equation setup

convert pixel to direction D

per-pixel edge equation

z buffer update

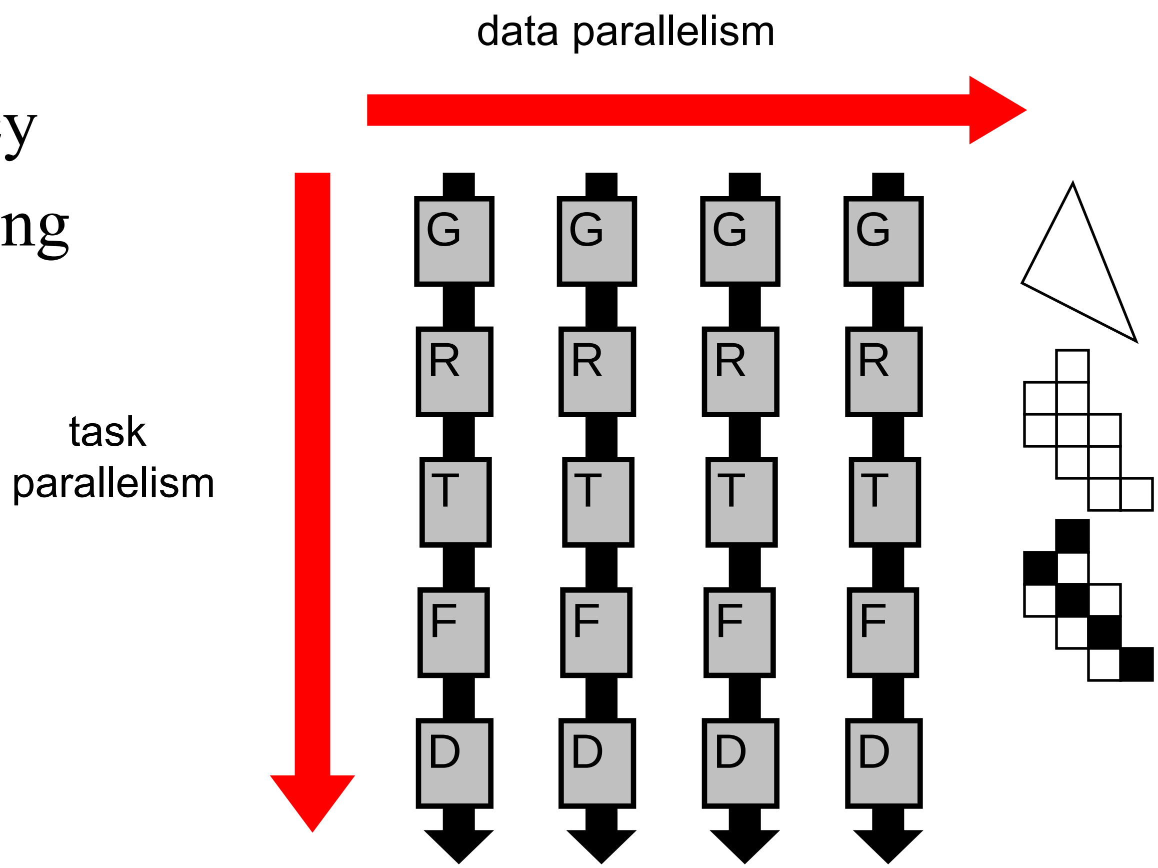
Good References

- 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 next week



Questions?

Movies

both rasterization and ray tracing

Images removed due to copyright restrictions.

Games

rasterization

Images removed due to copyright restrictions.

Simulation

rasterization (painter for a long time)

Images removed due to copyright restrictions.

CAD-CAM & Design

rasterization for GUI,
anything for final image

Images removed due to copyright restrictions.

Architecture

ray-tracing, rasterization with preprocessing for complex lighting

Images removed due to copyright restrictions.

Virtual Reality

rasterization

Images removed due to copyright restrictions.

Visualization

mostly rasterization,
interactive ray-tracing is starting

Images removed due to copyright restrictions.

Medical Imaging

same as visualization

Images removed due to copyright restrictions.

Questions?

More issues

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

- Shadows
  - Next time
- Reflections, global illumination

Transparency

- Triangles and pixels can have transparency (alpha)
- But the result depends on the order in which triangles are sent

- Big problem: visibility
  - There is only one depth stored per pixel/sample
  - transparent objects involve multiple depth
  - full solutions store a (variable-length) list of visible objects and depth at each pixel
    - see e.g. the A-buffer by Carpenter http://portal.acm.org/citation.cfm?id=808585

Deferred shading

- Avoid shading fragments that are eventually hidden
  - shading becomes more and more costly
- First pass: rasterize triangles, store information such as normals, BRDF per pixel
- Second pass: use stored information to compute shading

- Advantage: no useless shading
- Disadvantage: storage, antialiasing is difficult

Pre z pass

- Again, avoid shading hidden fragment
- First pass: rasterize triangles, update only z buffer, not color buffer
- Second pass: rasterize triangles again, but this time, do full shading

- Advantage over deferred shading: less storage, less code modification, more general shading is possible, multisampling possible
- Disadvantage: needs to rasterize twice

Tile-based rendering

- Problem: framebuffer is a lot of memory, especially with antialiasing
- Solution: render subsets of the screen at once
- For each tile of pixels
  - For each triangle
    - for each pixel

- Might need to handle a triangle in multiple tiles
  - redundant computation for projection and setup
- Used in mobile graphics cards

Reyes - Pixar’s Renderman

- Cook et al. http://graphics.pixar.com/library/Reyes/
- Based on micropolygons
  - each primitive gets diced into polygons as small as a pixel
- Enables antialiasing motion blur, depth of field
- Shading is computed at the micropolygon level, not pixel
  - related to multisampling: shaded value will be used for multiple visibility sample

Dicing and rasterization

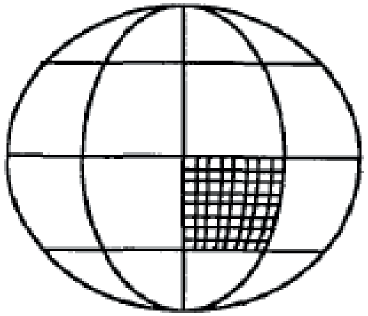


Figure 4a. A sphere is split into patches, and one of the patches is diced into a 8×8 grid of micropolygons.

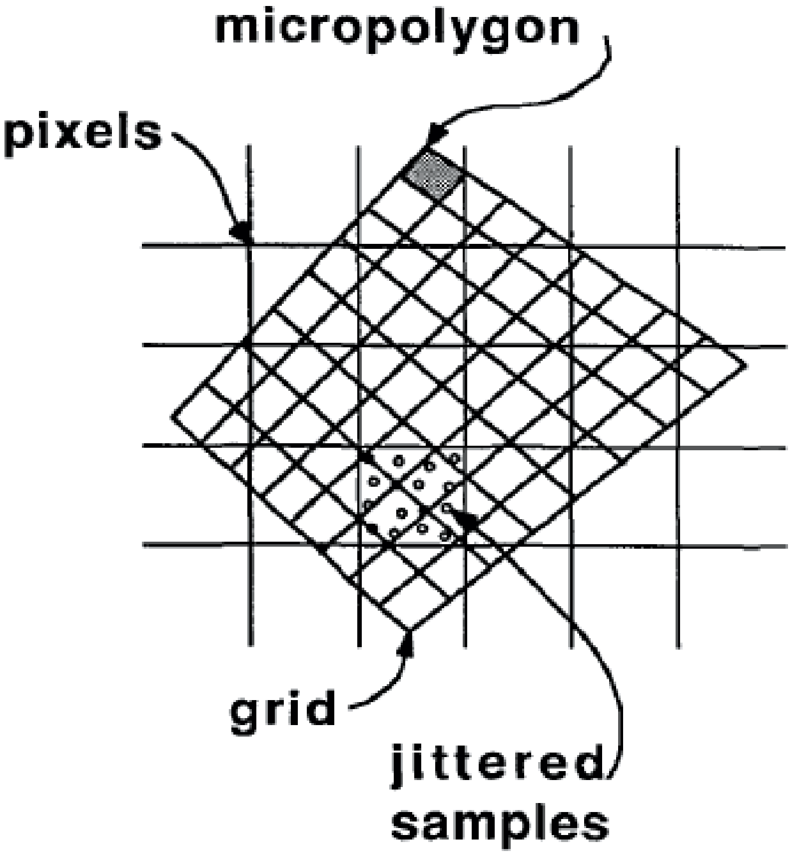


Figure 4b. The micropolygons in the grid are transformed to screen space, where they are stochastically sampled.

Reyes - Pixar's Renderman

- Tile-based to save memory and maximize texture coherence
- Order-independent transparency
  - stores list of fragments and depth per pixel
- Micropolygons get rasterized in space, lens and time
  - frame buffer has multiple samples per pixel
  - each sample has lens coordinates and time value

Reyes - ignoring transparency

- For each tile of pixels
  - For each geometry
    - Dice into micropolygons adaptively
    - For each micropolygon
      - compute shaded value
      - For each sample in tile at coordinates x, y, u, v, t
        - » reproject micropolygon to its position at time t, and lens position uv
        - » determine if micropolygon overlaps samples
        - » if yes, test visibility (z-buffer)
        - » if z buffer passes, update framebuffer

REYES results

Figure 6. 1986 Pixar Christmas Card by John Lasseter and Eben Ostby.

Questions?