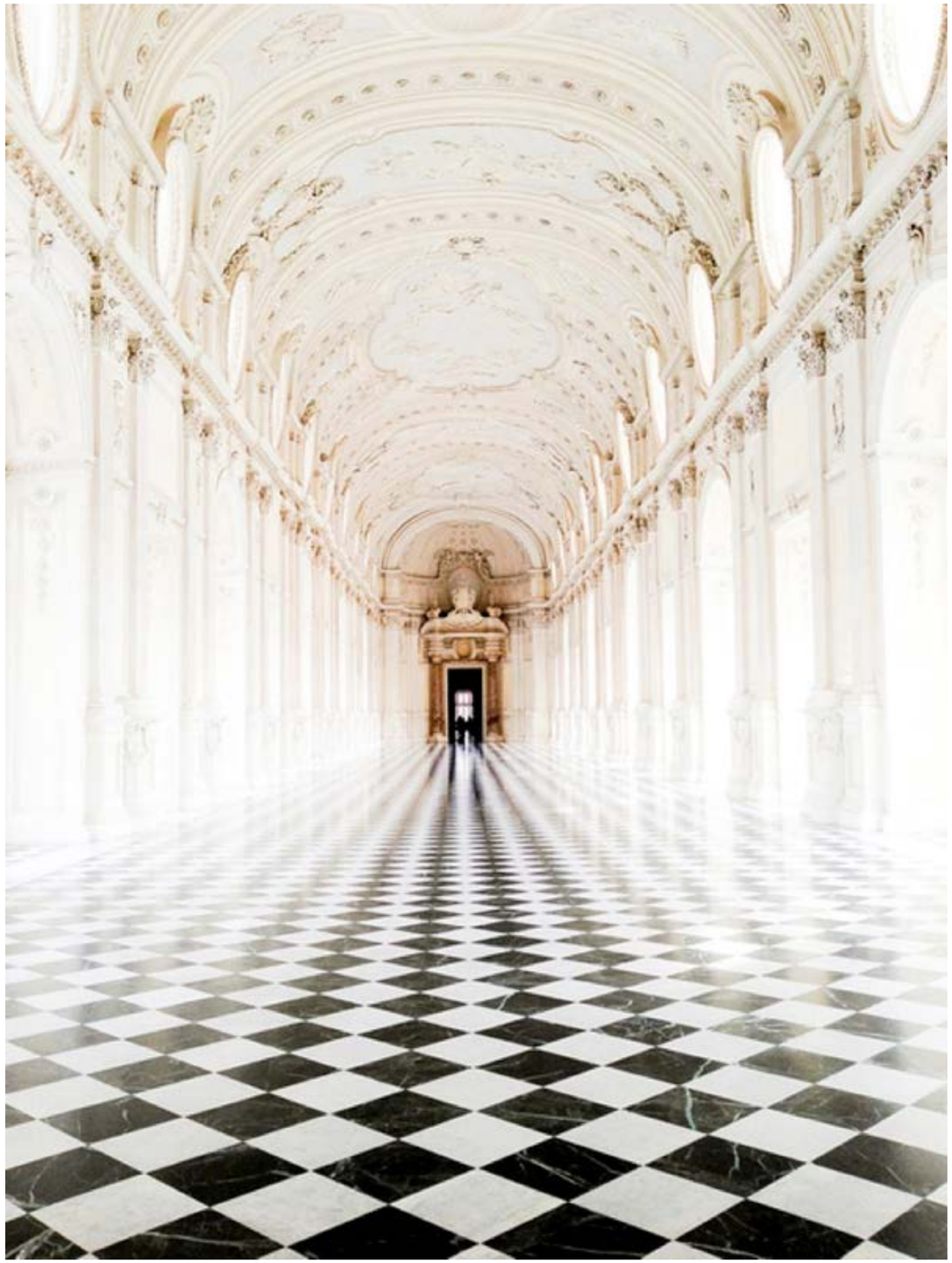
Perspective Projection and Texture Mapping

Computer Graphics
CMU 15-462/15-662

Perspective & Texture

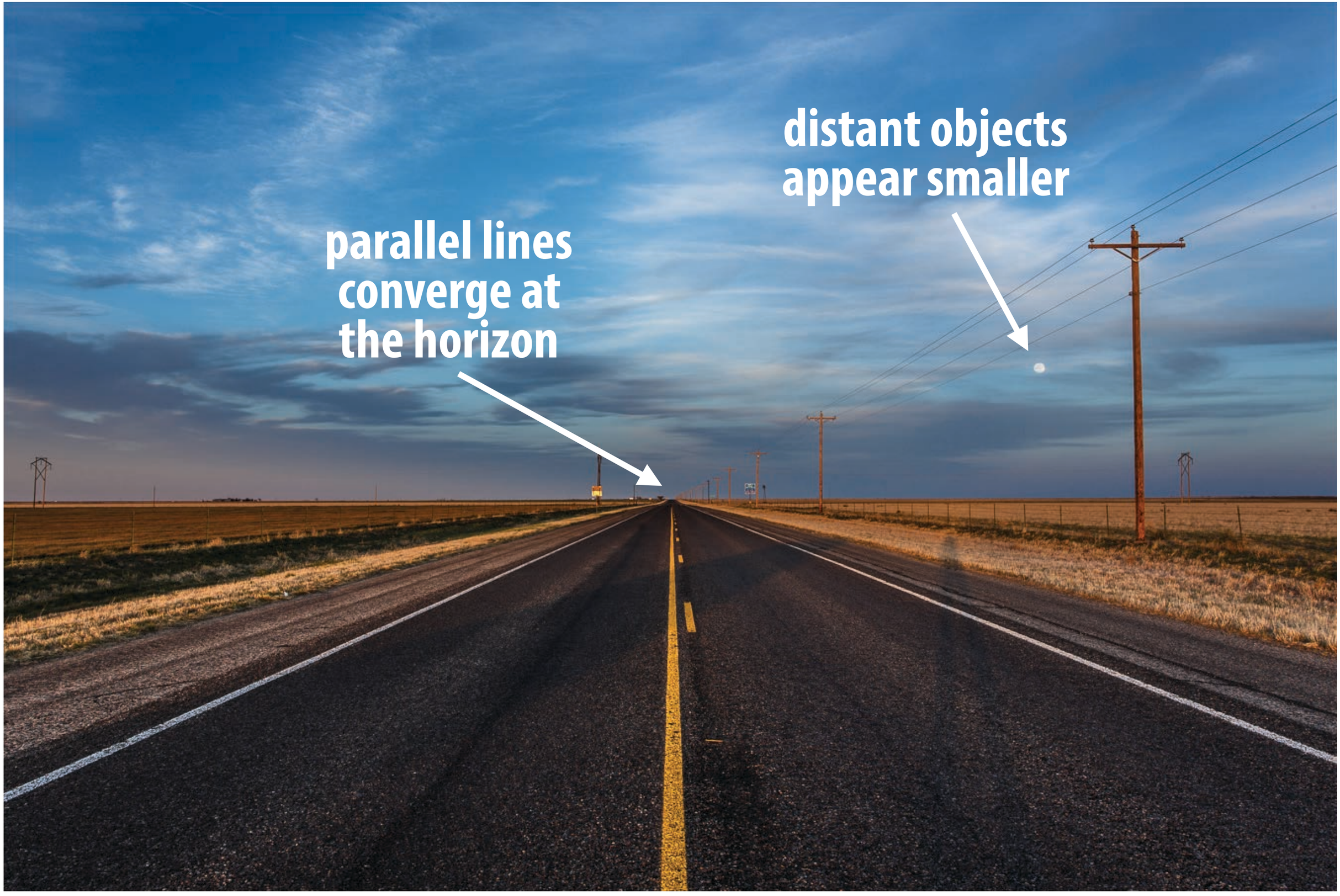
- **PREVIOUSLY:**
  - ***rasterization*** **(how to turn primitives into pixels)**
  - ***transformations*** **(how to manipulate primitives in space)**
- **TODAY:**
  - **see where these two ideas come crashing together!**
  - **revisit *perspective* transformations**
  - **talk about how to map *texture* onto a primitive to get more detail**
  - **…and how perspective creates challenges for texture mapping!**



Why is it hard to render an image like this?

Perspective Projection

Perspective projection



Early painting: incorrect perspective

Carolingian painting from the 8-9th century

Evolution toward correct perspective

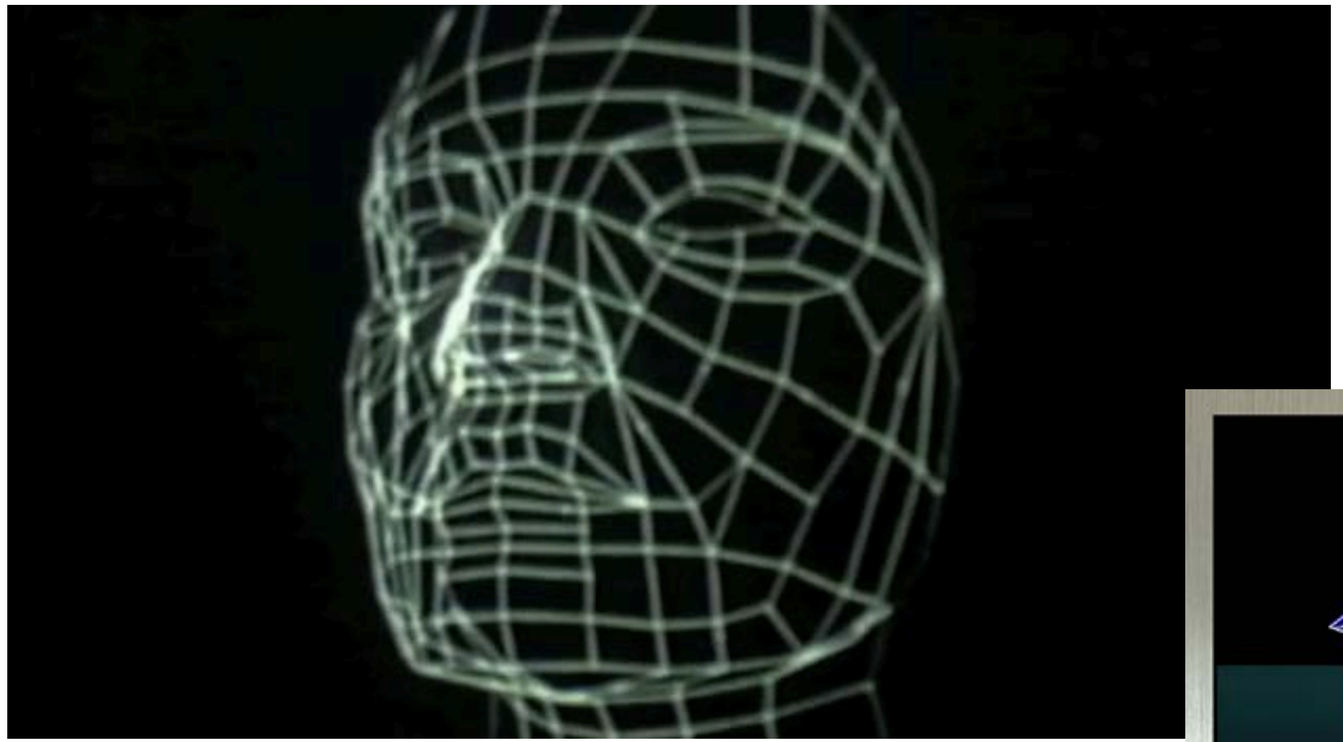
Ambrogio Lorenzetti
Annunciation, 1344

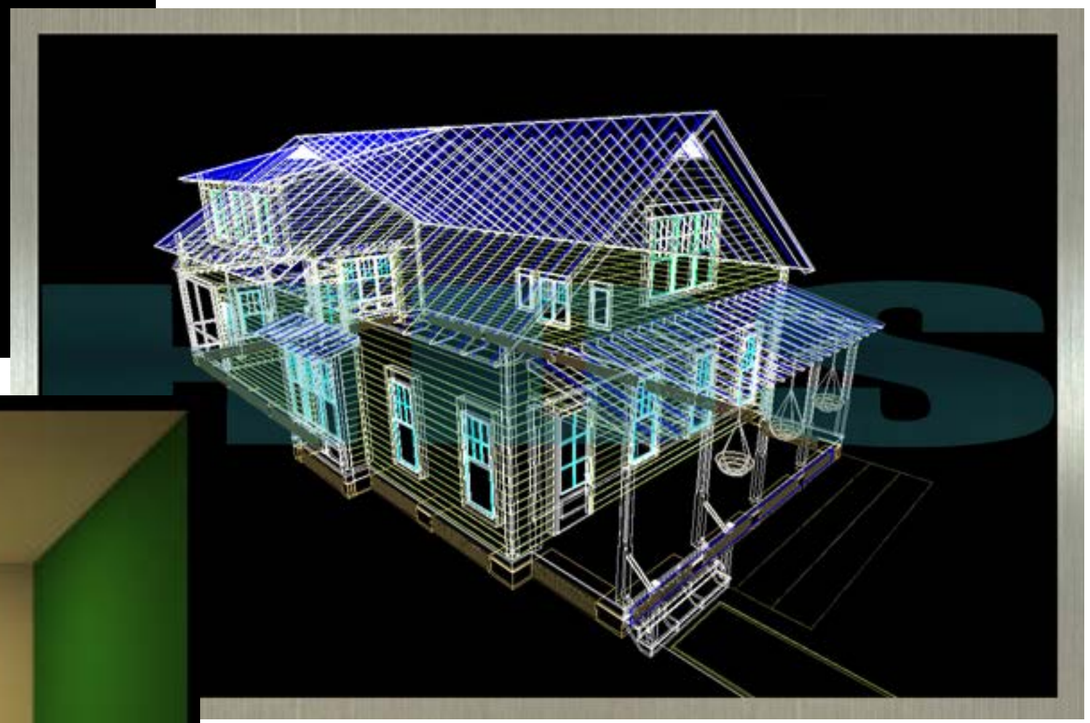
Brunelleschi, elevation of Santo Spirito, 1434-83, Florence

Masaccio – The Tribute Money c.1426-27
Fresco, The Brancacci Chapel, Florence

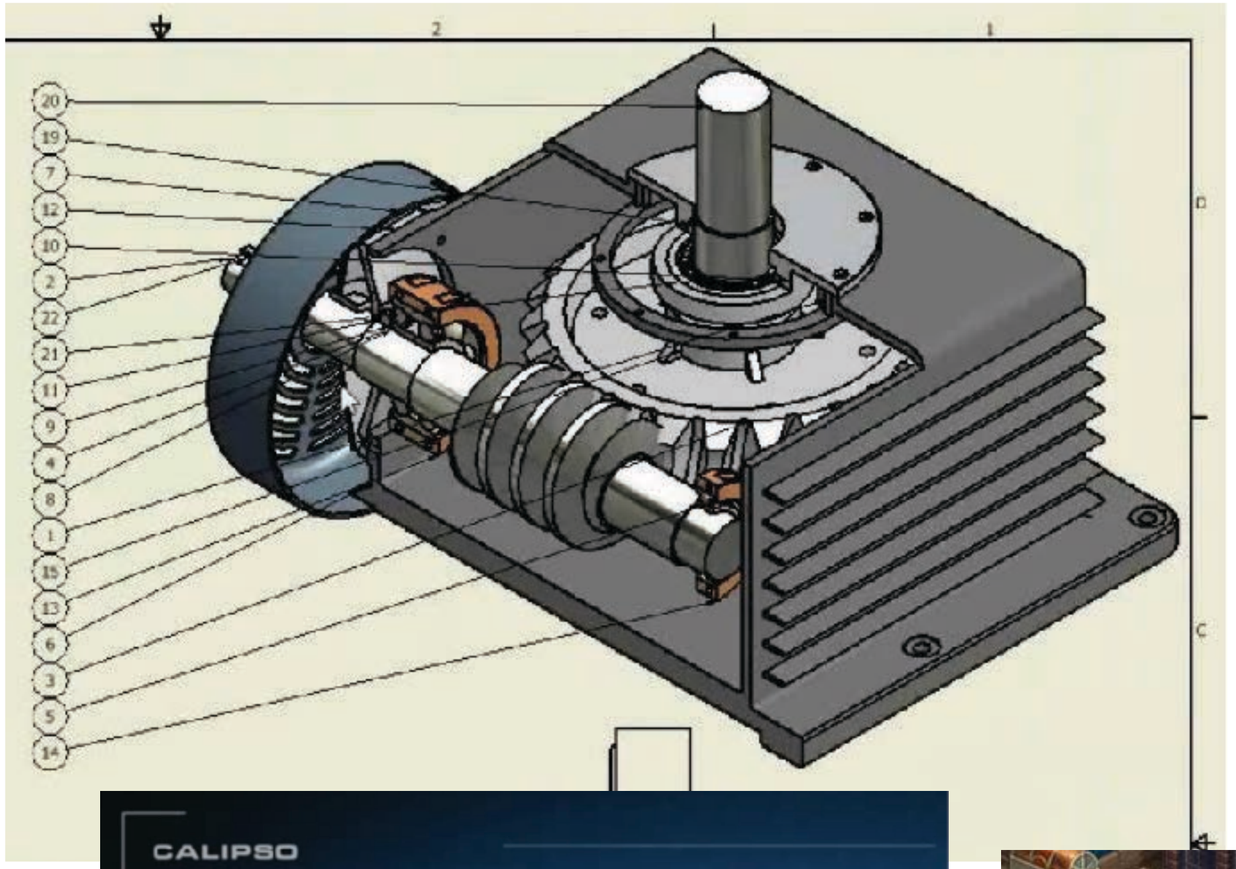
Later… rejection of proper perspective projection

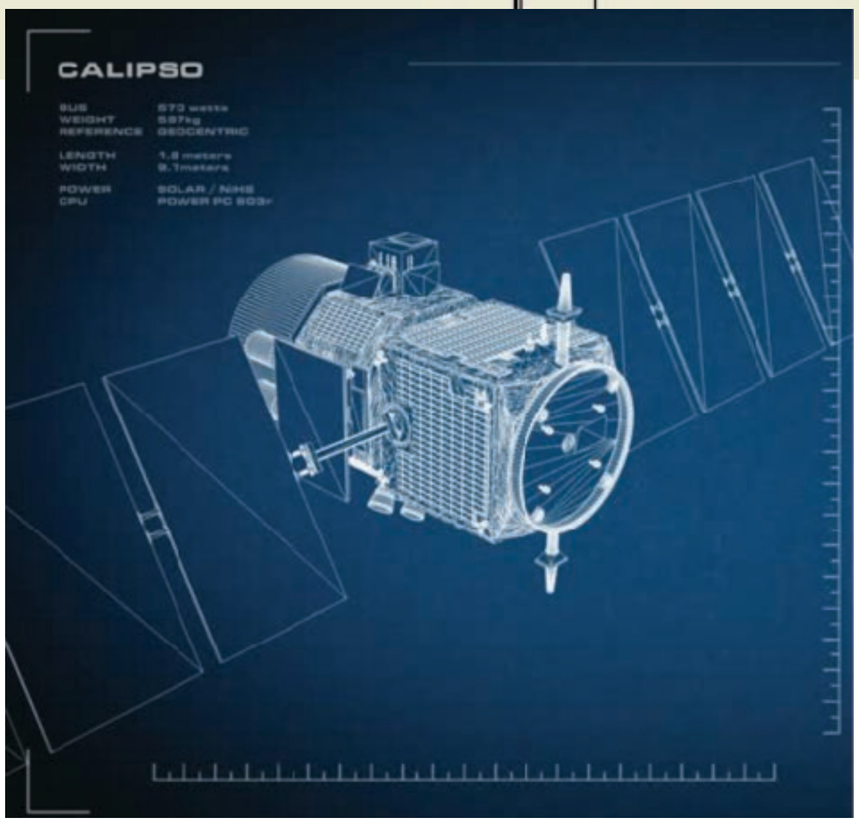
Return of perspective in computer graphics





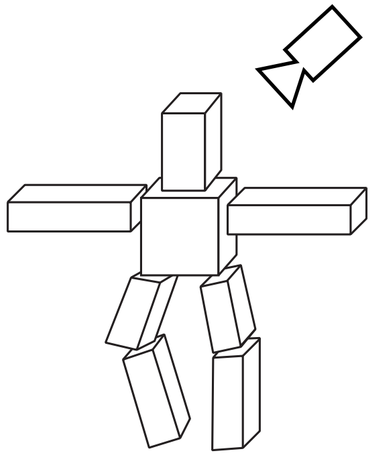
Rejection of perspective in computer graphics





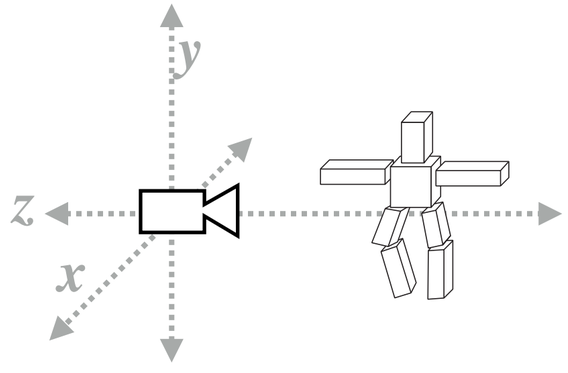
Transformations: From Objects to the Screen

[WORLD COORDINATES]



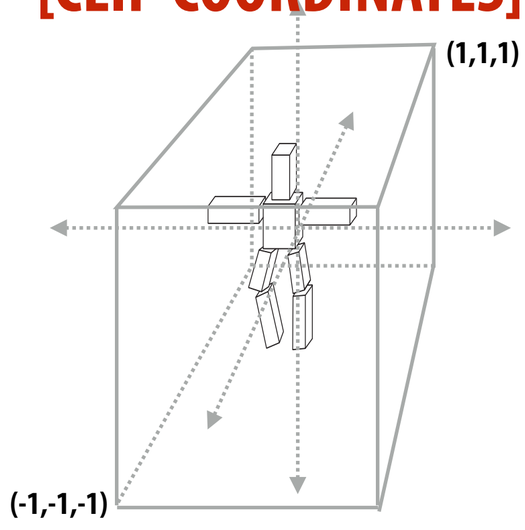
original description of objects

[VIEW COORDINATES]



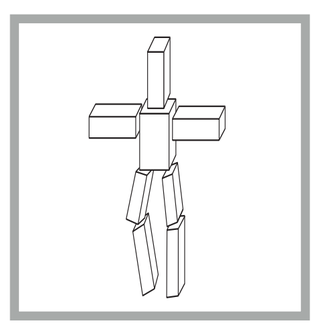
all positions now expressed relative to camera; camera is sitting at origin looking down -z direction

[CLIP COORDINATES]



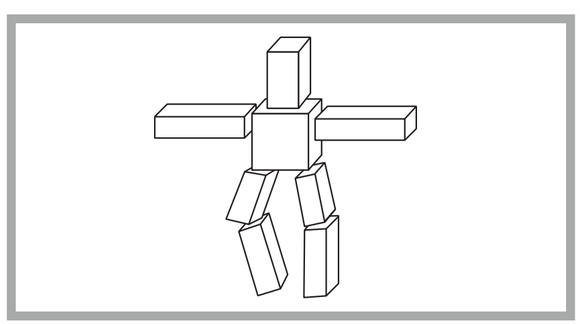
everything visible to the camera is mapped to unit cube for easy "clipping"

[NORMALIZED COORDINATES]



unit cube mapped to unit square via perspective divide

[WINDOW COORDINATES]



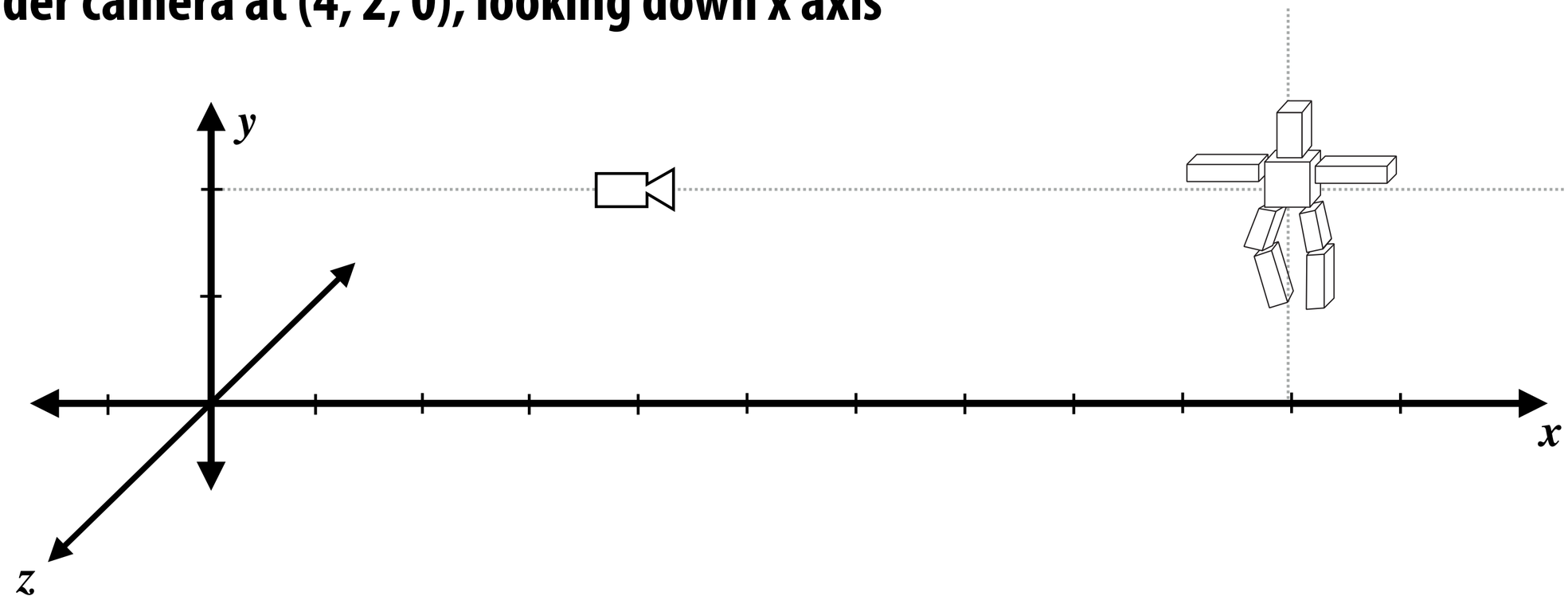
Screen transform:
objects now in 2D screen coordinates

primitives are now 2D and can be drawn via rasterization

Review: simple camera transform

Consider object positioned in world at (10, 2, 0)

Consider camera at (4, 2, 0), looking down x axis



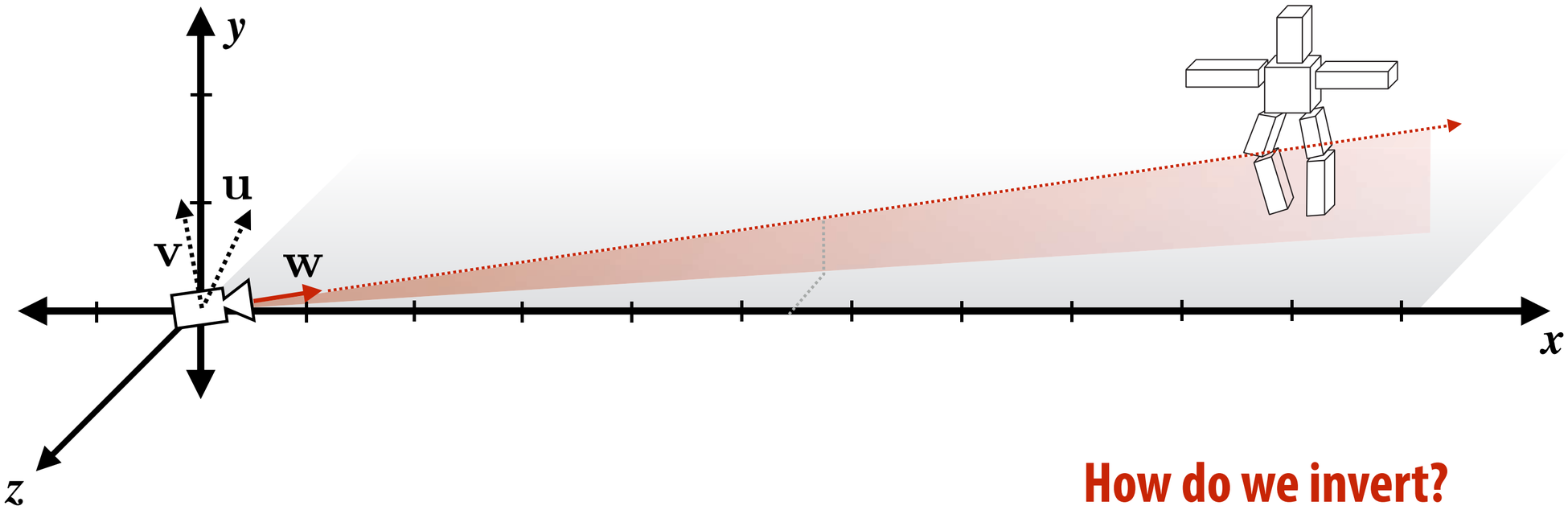
What transform places in the object in a coordinate space where the camera is at the origin and the camera is looking directly down the -z axis?

- **Translating object vertex positions by (-4, -2, 0) yields position relative to camera**
- **Rotation about y by $\pi/2$ gives position of object in new coordinate system where camera's view direction is aligned with the -z axis**

Camera looking in a different direction

Consider camera looking in direction $\mathbf{w}$

What transform places in the object in a coordinate space where the camera is at the origin and the camera is looking directly down the -z axis?



Form orthonormal basis around $\mathbf{w}$: (see $\mathbf{u}$ and $\mathbf{v}$)

Consider rotation matrix: $\mathbf{R}$

$$\mathbf{R} = \begin{bmatrix} \mathbf{u}_x & \mathbf{v}_x & -\mathbf{w}_x \\ \mathbf{u}_y & \mathbf{v}_y & -\mathbf{w}_y \\ \mathbf{u}_z & \mathbf{v}_z & -\mathbf{w}_z \end{bmatrix}$$

$\mathbf{R}$ maps x-axis to $\mathbf{u}$, y-axis to $\mathbf{v}$, z axis to $-\mathbf{w}$

How do we invert?

$$\mathbf{R}^{-1} = \mathbf{R}^T = \begin{bmatrix} \mathbf{u}_x & \mathbf{u}_y & \mathbf{u}_z \\ \mathbf{v}_x & \mathbf{v}_y & \mathbf{v}_z \\ -\mathbf{w}_x & -\mathbf{w}_y & -\mathbf{w}_z \end{bmatrix}$$

Why is that the inverse?

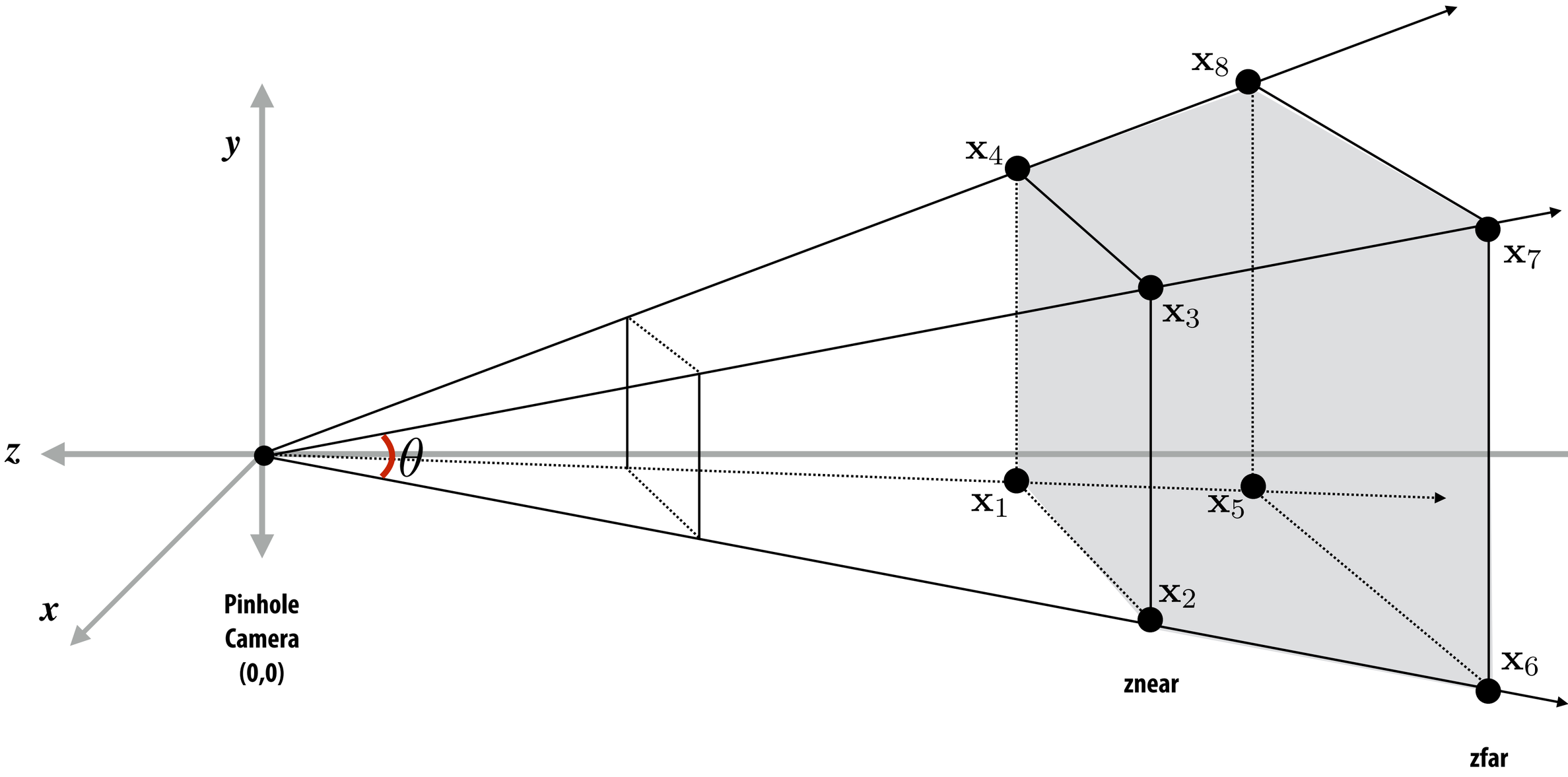
$$\mathbf{R}^T\mathbf{u} = \begin{bmatrix} \mathbf{u}\cdot\mathbf{u} & \mathbf{v}\cdot\mathbf{u} & -\mathbf{w}\cdot\mathbf{u} \end{bmatrix}^T = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}^T$$

$$\mathbf{R}^T\mathbf{v} = \begin{bmatrix} \mathbf{u}\cdot\mathbf{v} & \mathbf{v}\cdot\mathbf{v} & -\mathbf{w}\cdot\mathbf{v} \end{bmatrix}^T = \begin{bmatrix} 0 & 1 & 0 \end{bmatrix}^T$$

$$\mathbf{R}^T\mathbf{w} = \begin{bmatrix} \mathbf{u}\cdot\mathbf{w} & \mathbf{v}\cdot\mathbf{w} & -\mathbf{w}\cdot\mathbf{w} \end{bmatrix}^T = \begin{bmatrix} 0 & 0 & -1 \end{bmatrix}^T$$

View frustum

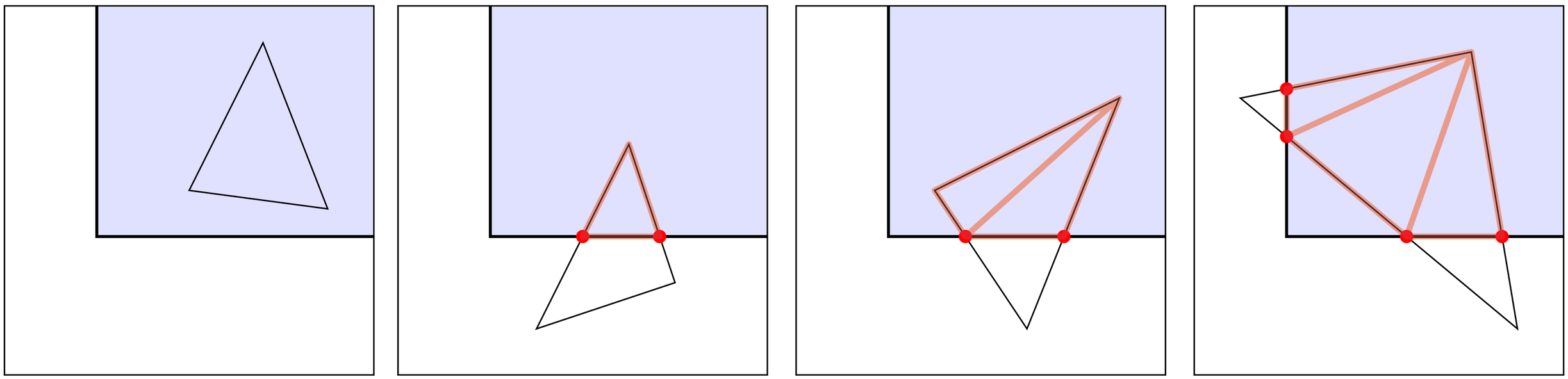
View frustum is region the camera can see:



- Top/bottom/left/right planes correspond to sides of screen
- Near/far planes correspond to closest/furthest thing we want to draw

Clipping

- In real-time graphics pipeline, "clipping" is the process of eliminating triangles that aren't visible to the camera
  - Don't waste time computing pixels (or really, *fragments*) you can't see!
  - Even "tossing out" individual fragments is expensive ("fine granularity")
  - Makes more sense to toss out whole primitives ("coarse granularity")
  - Still need to deal with primitives that are partially clipped…



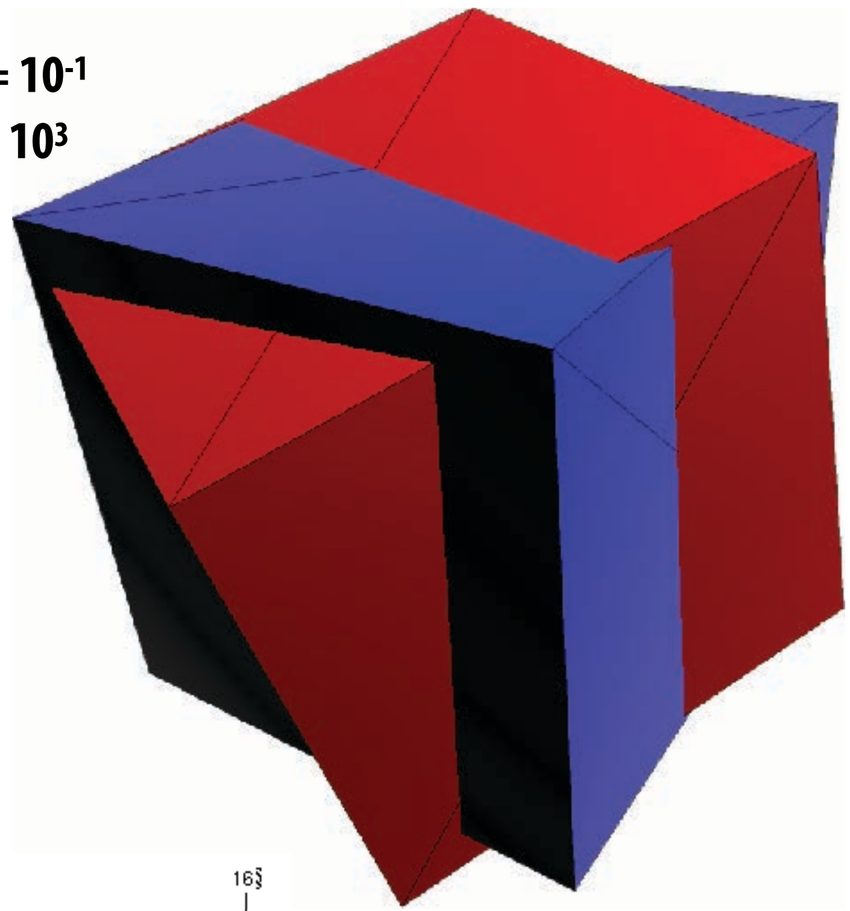
from: https://paroj.github.io/gltut/

Aside: Near/Far Clipping

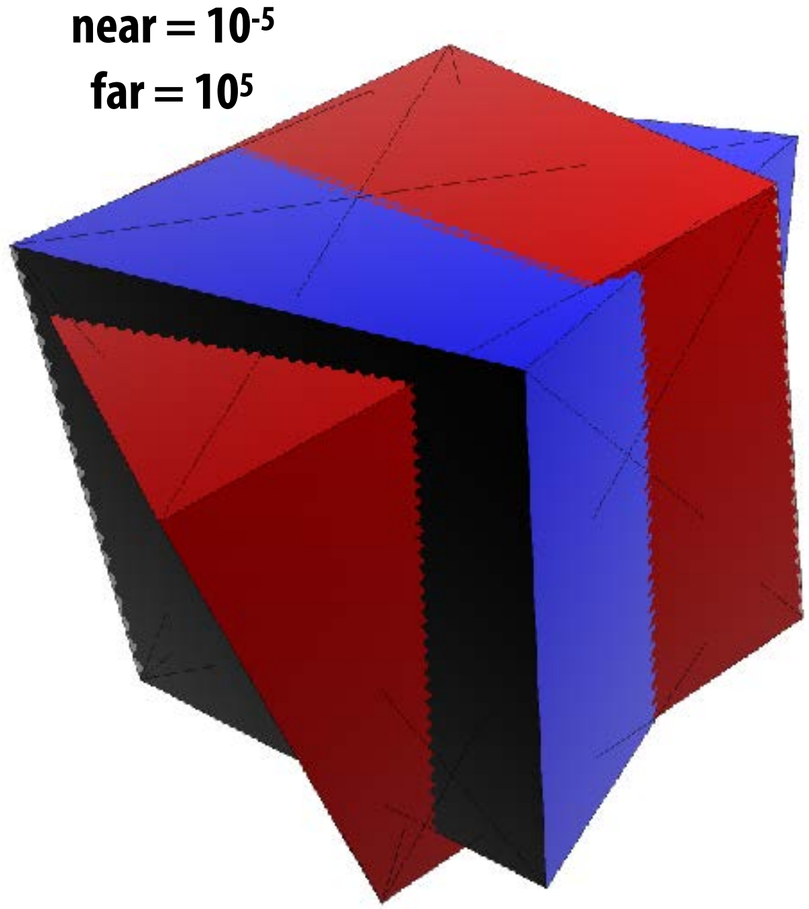
- But why *near/far* clipping?
  - Some primitives (e.g., triangles) may have vertices both in front & behind eye! (Causes headaches for rasterization, e.g., checking if fragments are behind eye)
  - Also important for dealing with finite precision of depth buffer / limitations on storing depth as floating point values

[DEMO]

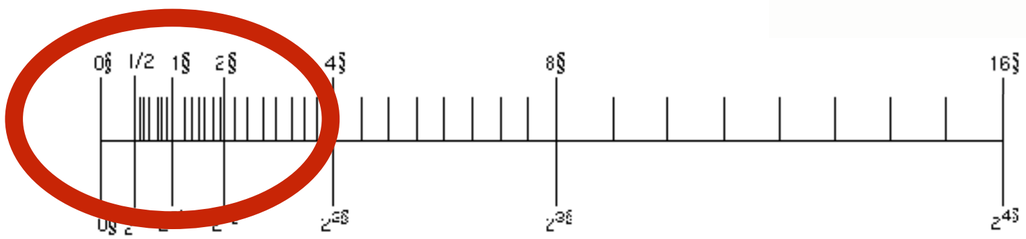
near = 10^{-1}
far = 10^{3}



near = 10^{-5}
far = 10^{5}



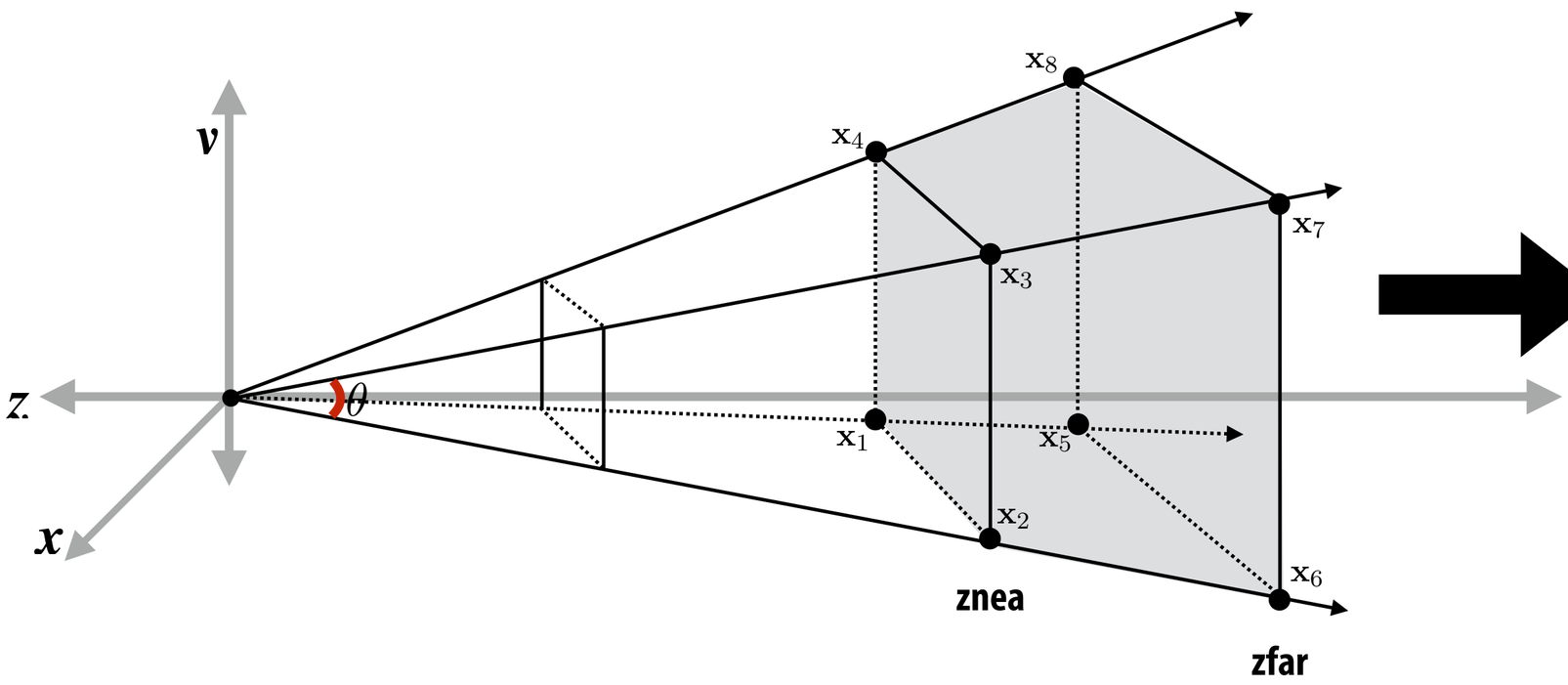
"Z-fighting"

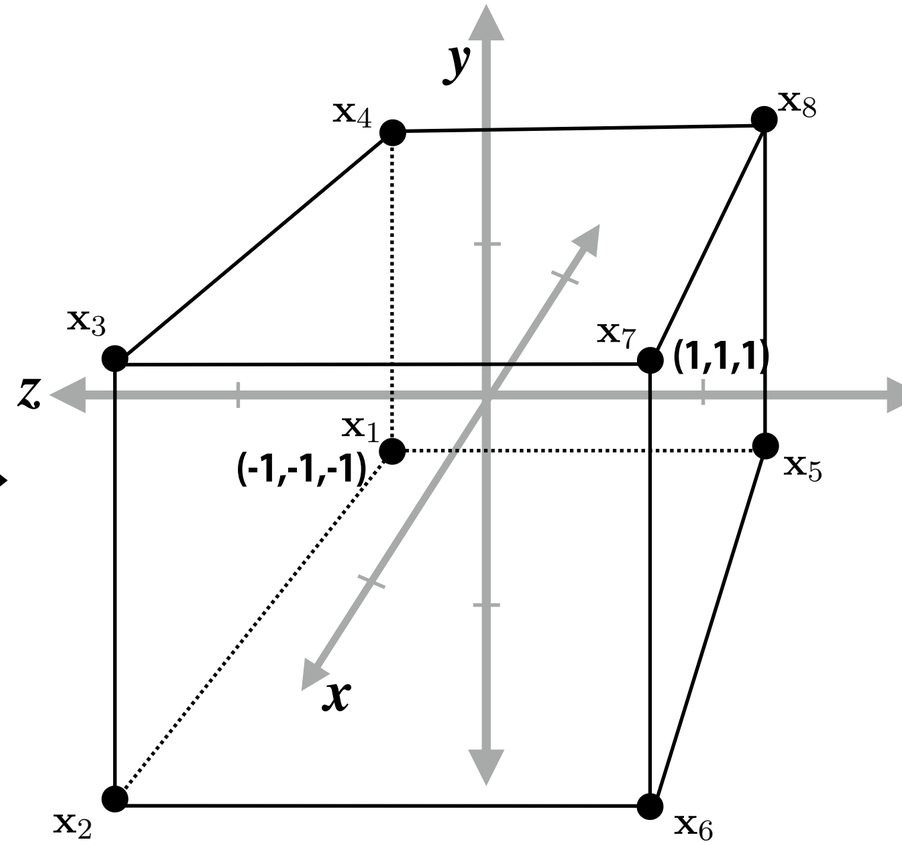


floating point has more "resolution" near zero—hence more precise resolution of primitive-primitive intersection

Mapping frustum to unit cube

Before mapping to 2D, map corners of frustum to corners of cube:





Why do we do this?

1. Makes *clipping* much easier!
   - can quickly discard points outside range [-1,1]
   - need to think a bit about partially-clipped triangles
2. Different maps to cube yield different effects
   - specifically perspective or orthographic view
   - perspective is transformation of homogeneous coords
   - for orthographic view, just use identity matrix!

Perspective:

Set homogeneous coord to "z"

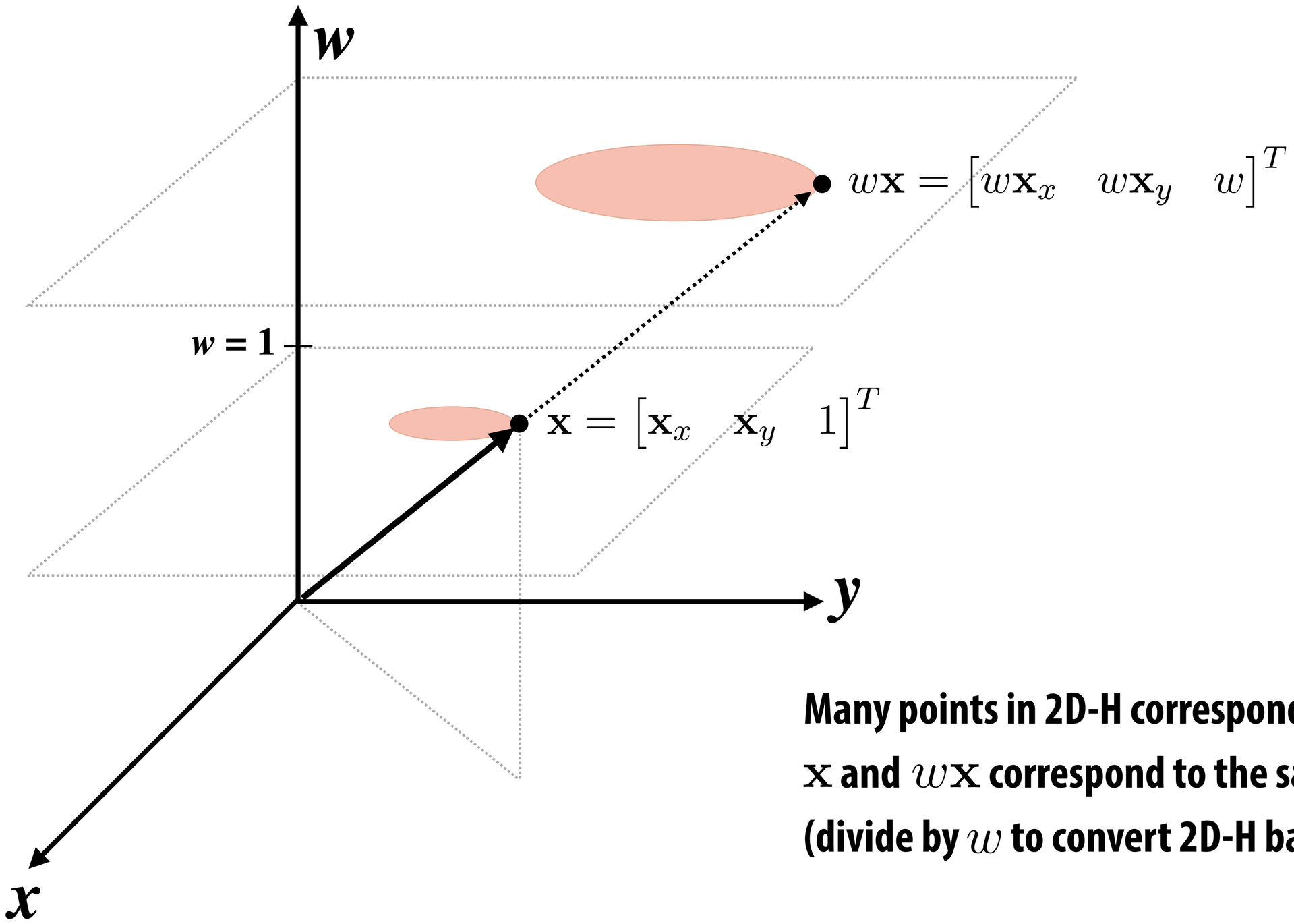
Distant objects get smaller

Orthographic:

Set homogeneous coord to "1"

Distant objects remain same size

Review: homogeneous coordinates



Many points in 2D-H correspond to same point in 2D

$\mathbf{x}$ and $w\mathbf{x}$ correspond to the same 2D point

(divide by w to convert 2D-H back to 2D)

Perspective vs. Orthographic Projection

- **Most basic version of perspective matrix:**

$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ z \end{bmatrix} \longmapsto \begin{bmatrix} x/z \\ y/z \\ 1 \\ 1 \end{bmatrix}$$

objects shrink in distance

- **Most basic version of orthographic matrix:**

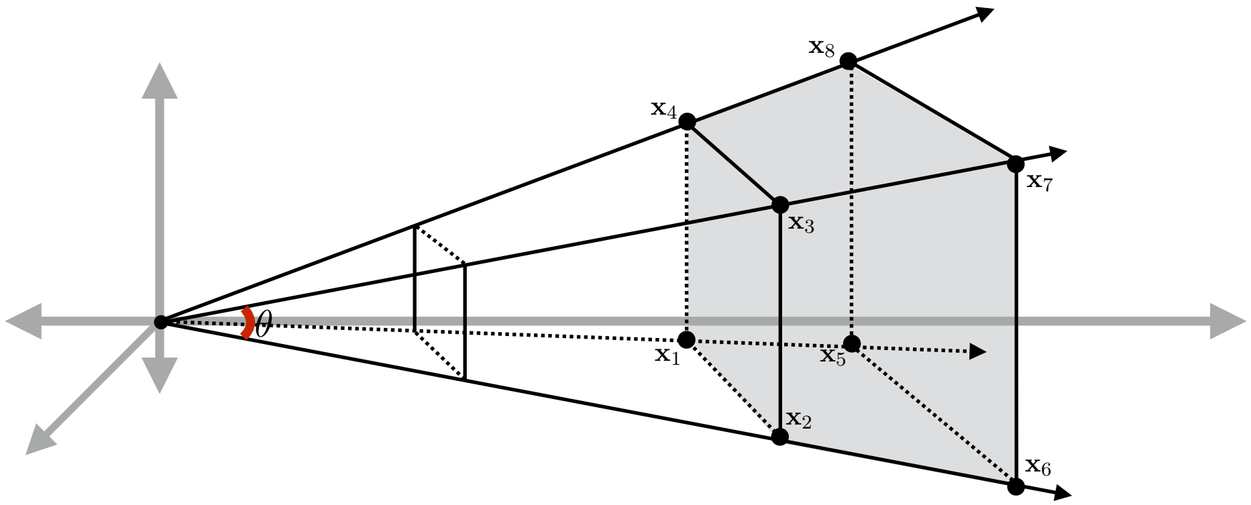
$$\begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} = \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \longmapsto \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix}$$

objects stay the same size

…real projection matrices are a bit more complicated! :-)

Matrix for Perspective Transform

Real perspective matrix takes into account geometry of view frustum:



$$\begin{bmatrix} \frac{2n}{r-l} & 0 & \frac{r+l}{r-l} & 0 \\ 0 & \frac{2n}{t-b} & \frac{t+b}{t-b} & 0 \\ 0 & 0 & \frac{-(f+n)}{f-n} & \frac{-2fn}{f-n} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

left (l), right (r), top (t), bottom (b), near (n), far (f)

For a derivation: http://www.songho.ca/opengl/gl_projectionmatrix.html

Review: screen transform

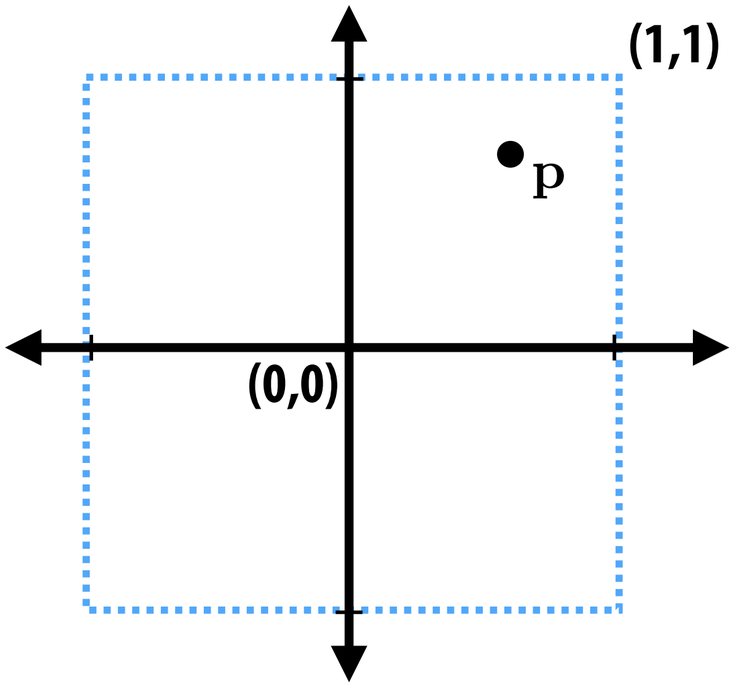
After divide, coordinates in "clip space" [-1,1] have to be stretched to fit the screen

Example:

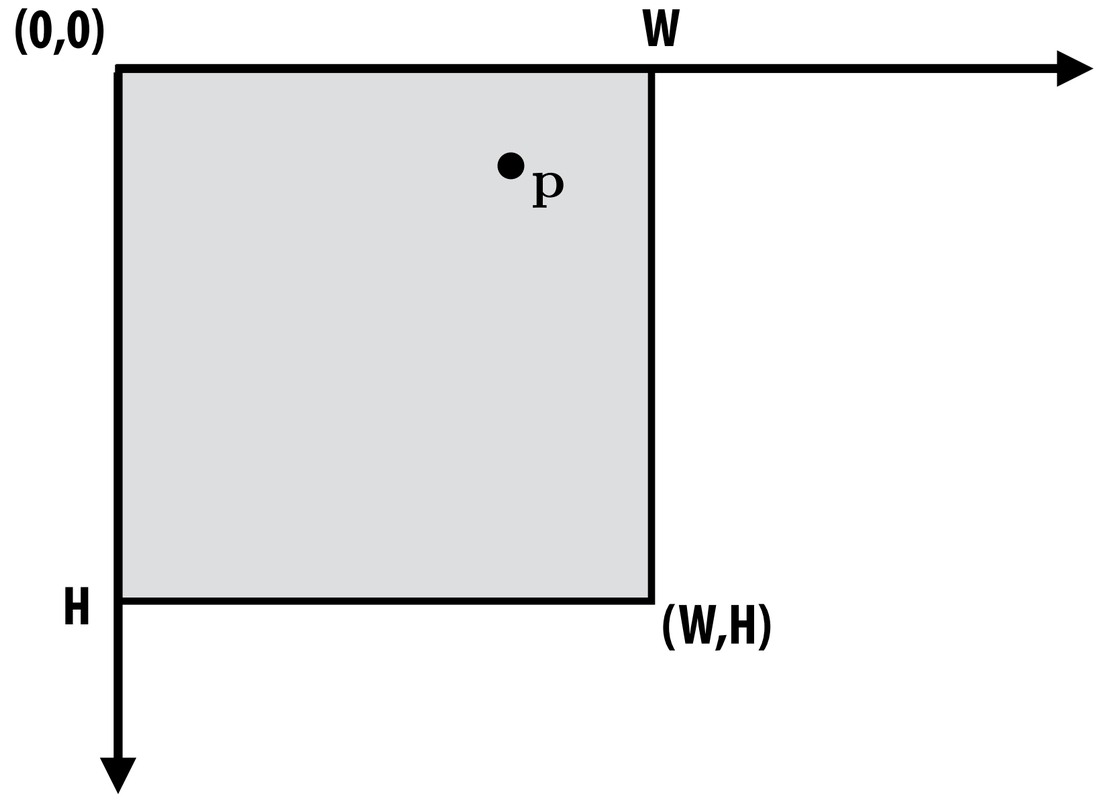
All points within (-1,1) to (1,1) region are on screen

(1,1) in normalized space maps to (W,0) in screen

Normalized coordinate space:



Screen (W x H output image) coordinate space:



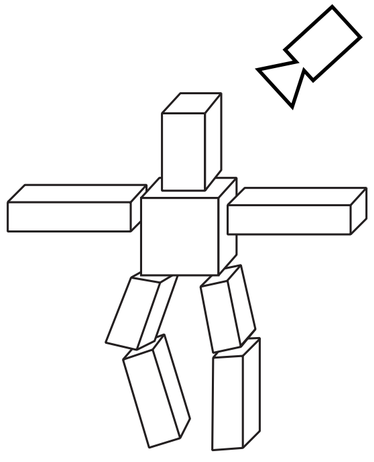
Step 1: reflect about x-axis

Step 2: translate by (1,1)

Step 3: scale by (W/2,H/2)

Transformations: From Objects to the Screen

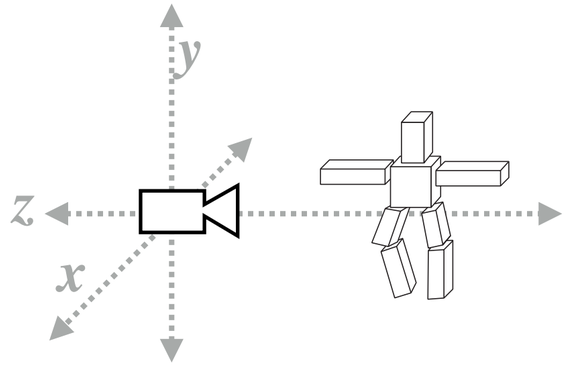
[WORLD COORDINATES]



original description of objects

view transform

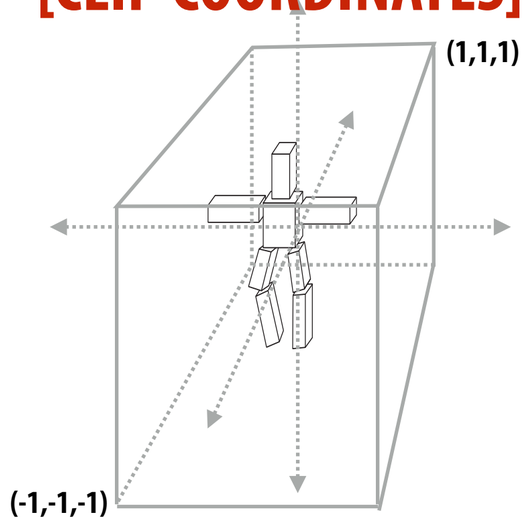
[VIEW COORDINATES]



all positions now expressed relative to camera; camera is sitting at origin looking down -z direction

projection transform

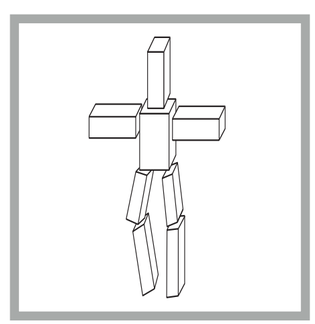
[CLIP COORDINATES]



everything visible to the camera is mapped to unit cube for easy "clipping"

perspective divide

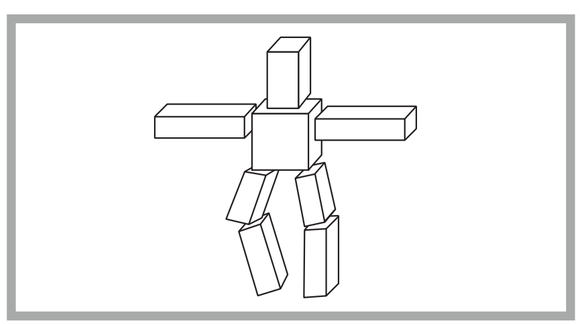
[NORMALIZED COORDINATES]



unit cube mapped to unit square via perspective divide

screen transform

[WINDOW COORDINATES]

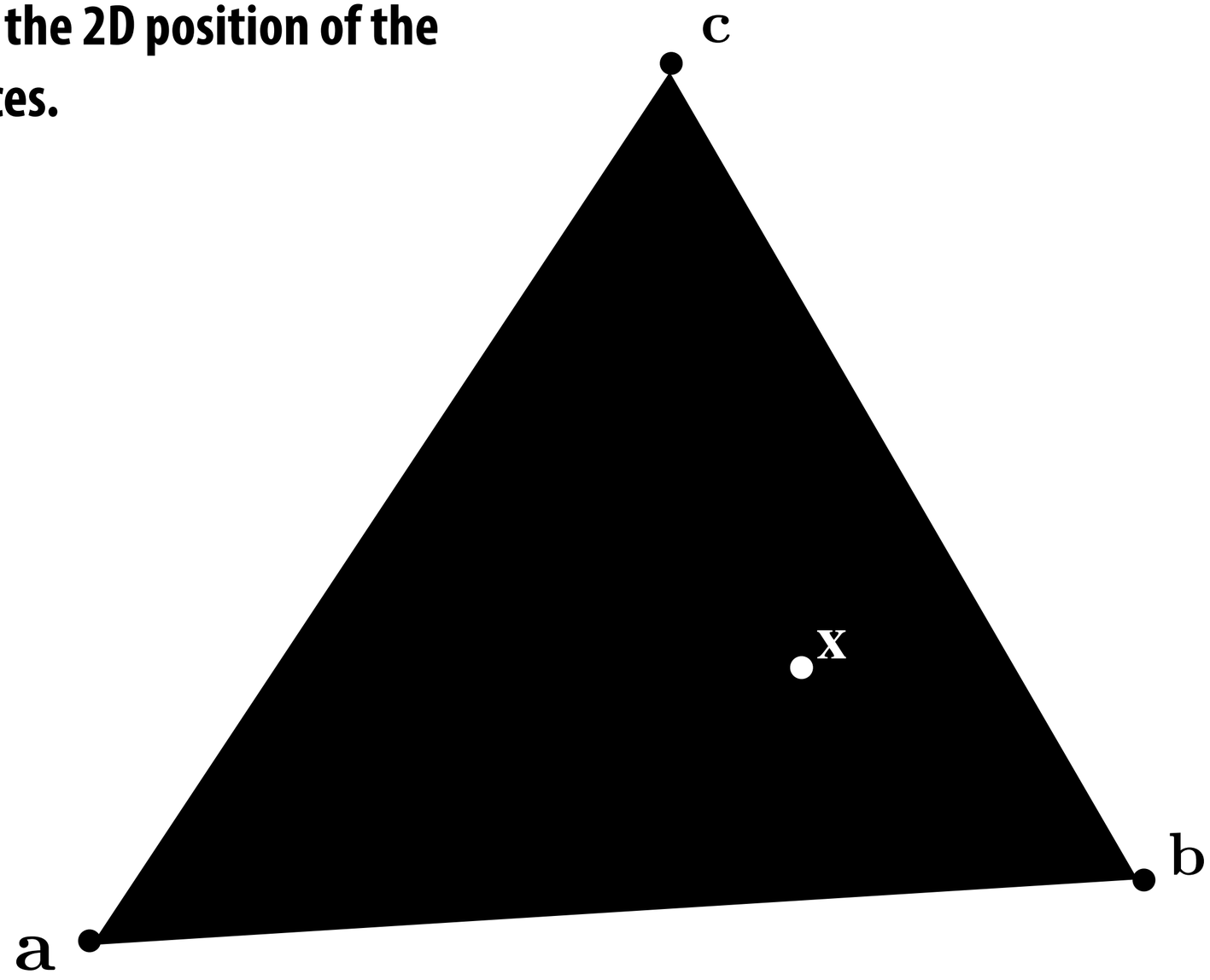


Screen transform: objects now in 2D screen coordinates

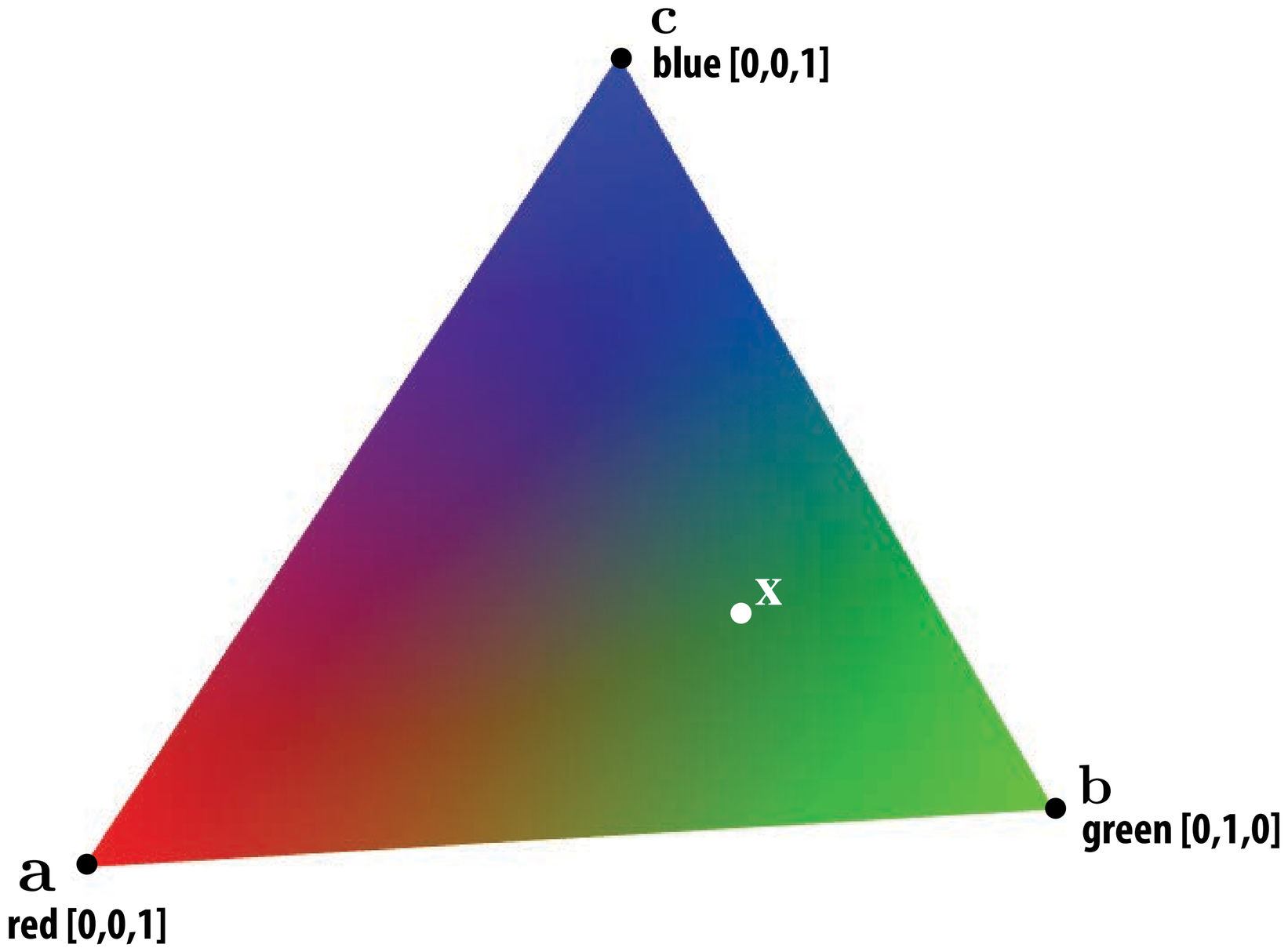
primitives are now 2D and can be drawn via rasterization

Coverage(x,y)

In lecture 2 we discussed how to sample coverage given the 2D position of the triangle's vertices.



Consider sampling color(x,y)



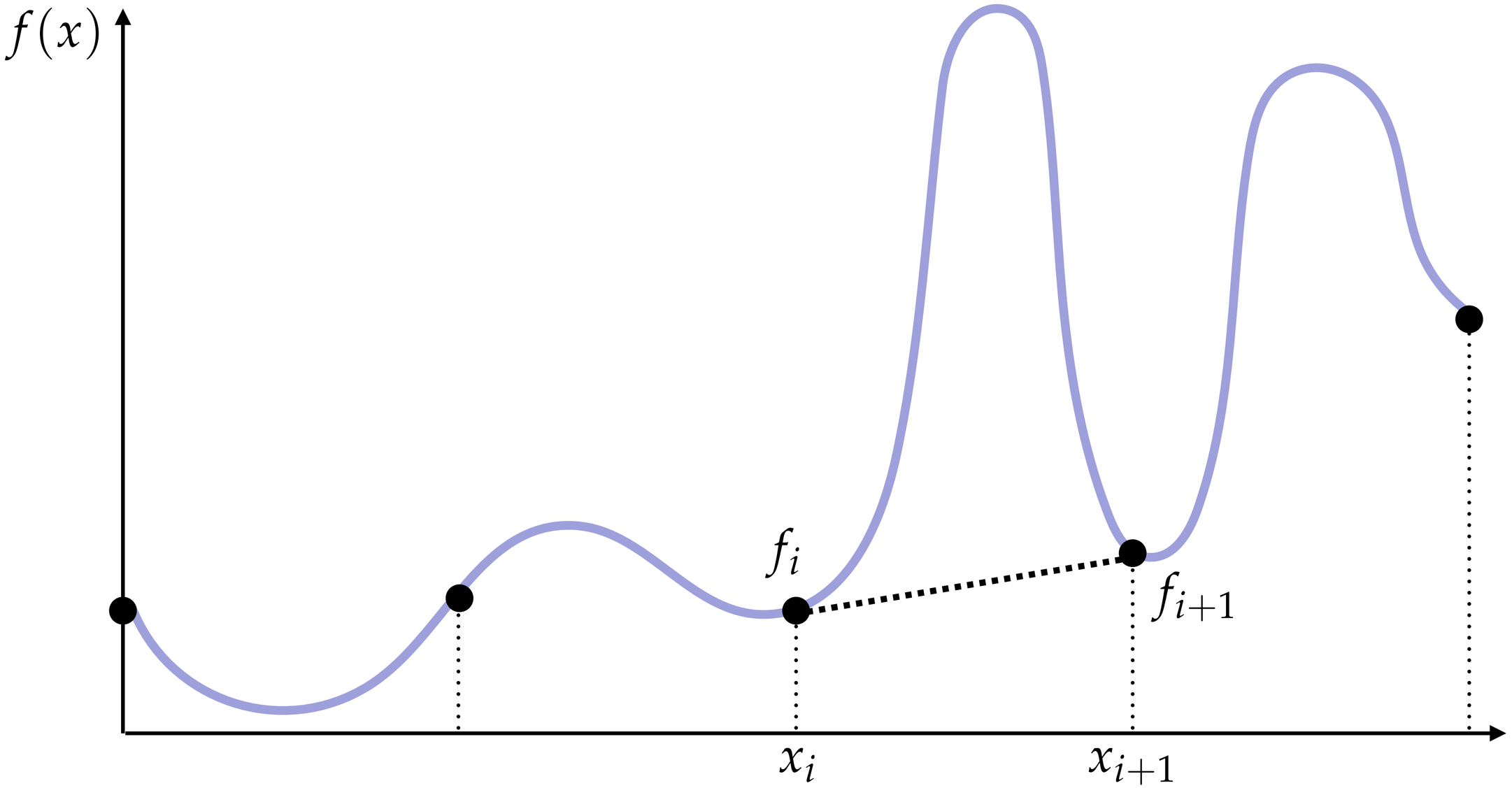
What is the triangle's color at the point x ?

Standard strategy: interpolate color values at vertices.

Linear interpolation in 1D

Suppose we've sampled values of a function f(x) at points x_i, i.e., $f_i := f(x_i)$

Q: How do we construct a function that "connects the dots" between x_i and x_{i+1}?



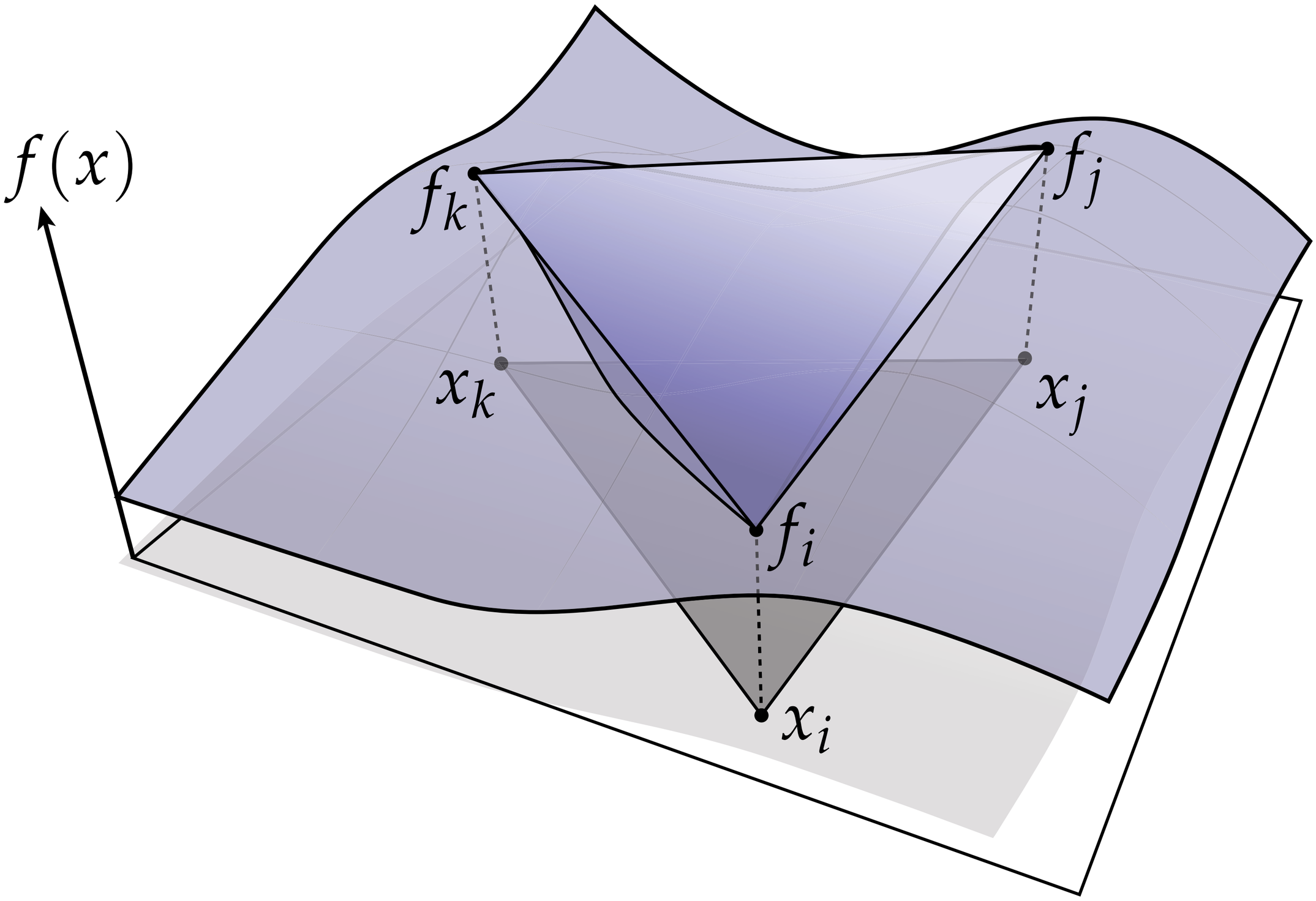
$$t := (x - x_i)/(x_{i+1} - x_i) \in [0, 1]$$

$$\hat{f}(t) = f_i + t(f_{i+1} - f_i) = (1 - t)f_i + tf_{i+1}$$

Linear interpolation in 2D

Suppose we've likewise sampled values of a function f(x) at points x_i, x_j, x_i in 2D

Q: How do we "connect the dots" this time? E.g., how do we fit a plane?



Linear interpolation in 2D

- Want to fit a linear (really, *affine*) function to three values
- Any such function has three unknown coefficients a, b, and c:

$$\hat{f}(x,y) = ax + by + c$$

- To interpolate, we need to find coefficients such that the function matches the sample values at the sample points:

$$\hat{f}(x_p,y_p) = f_p,\ p \in \{i,j,k\}$$

- Yields three linear equations in three unknowns. Solution?

$$\begin{bmatrix} a \\ b \\ c \end{bmatrix} = \frac{1}{(x_j y_i - x_i y_j) + (x_k y_j - x_j y_k) + (x_i y_k - x_k y_i)} \begin{bmatrix} f_i(y_k - y_j) + f_j(y_i - y_k) + f_k(y_j - y_i) \\ f_i(x_j - x_k) + f_j(x_k - x_i) + f_k(x_i - x_j) \\ f_i(x_k y_j - x_j y_k) + f_j(x_i y_k - x_k y_i) + f_k(x_j y_i - x_i y_j) \end{bmatrix}$$

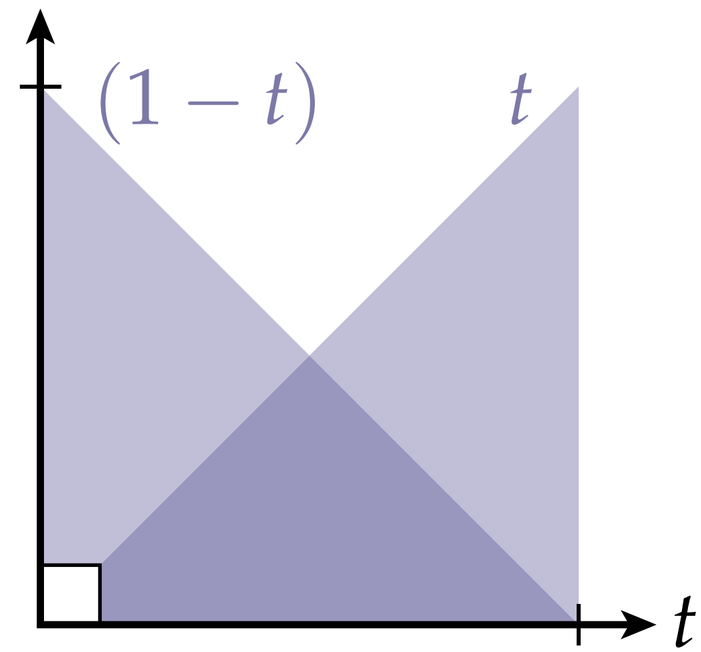
There has to be a better way to think about this. :-)

1D Linear Interpolation, revisited

- Let's think about how we did linear interpolation in 1D:

$$\hat{f}(t) = (1-t)f_i + tf_j$$

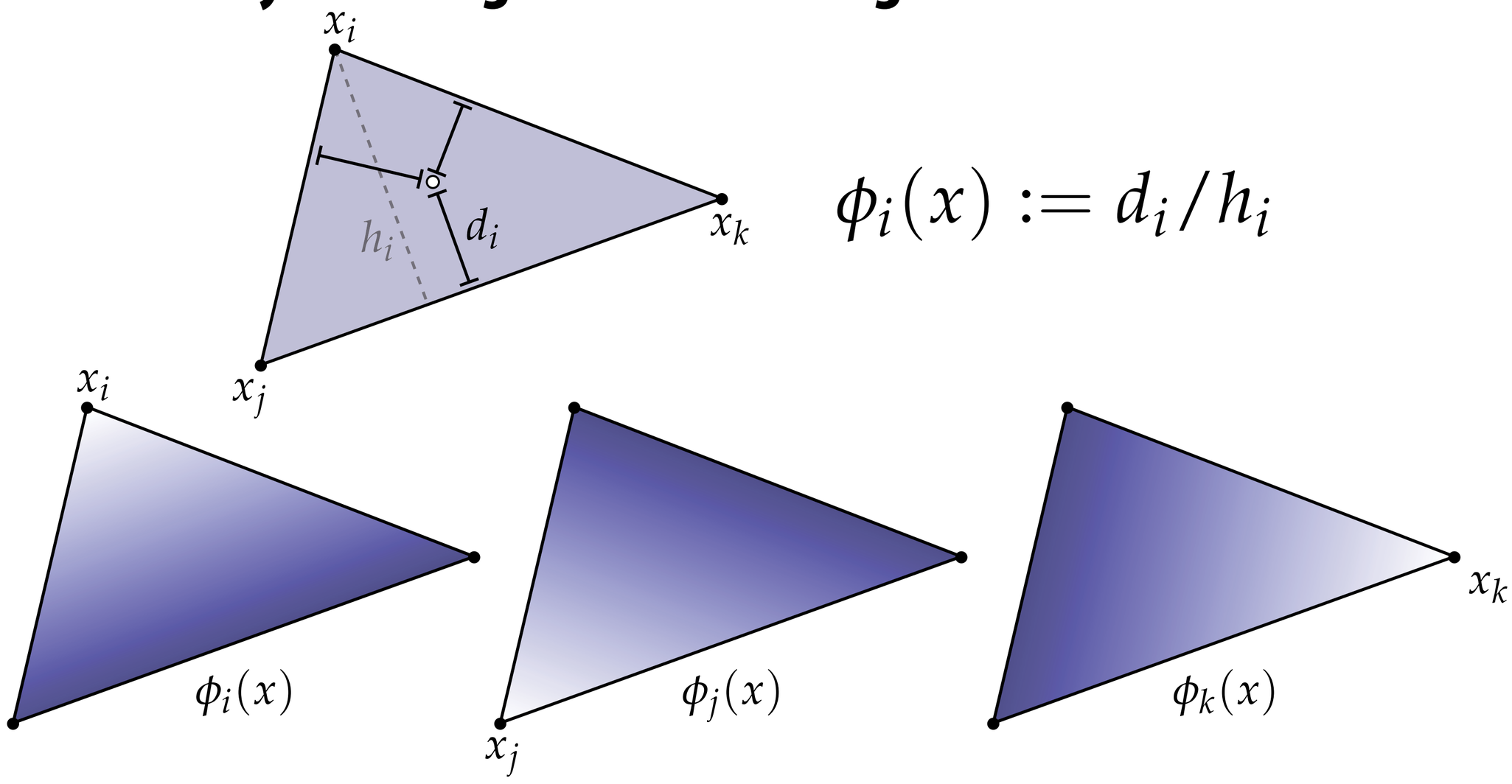
- Can think of this as a linear combination of two functions:



- As we move closer to t=0, we approach the value of f at x_i
- As we move closer to t=1, we approach the value of f at x_j

2D Linear Interpolation, revisited

- We can construct analogous functions for a triangle
- For a given point x, measure the distance to each edge; then divide by the height of the triangle:

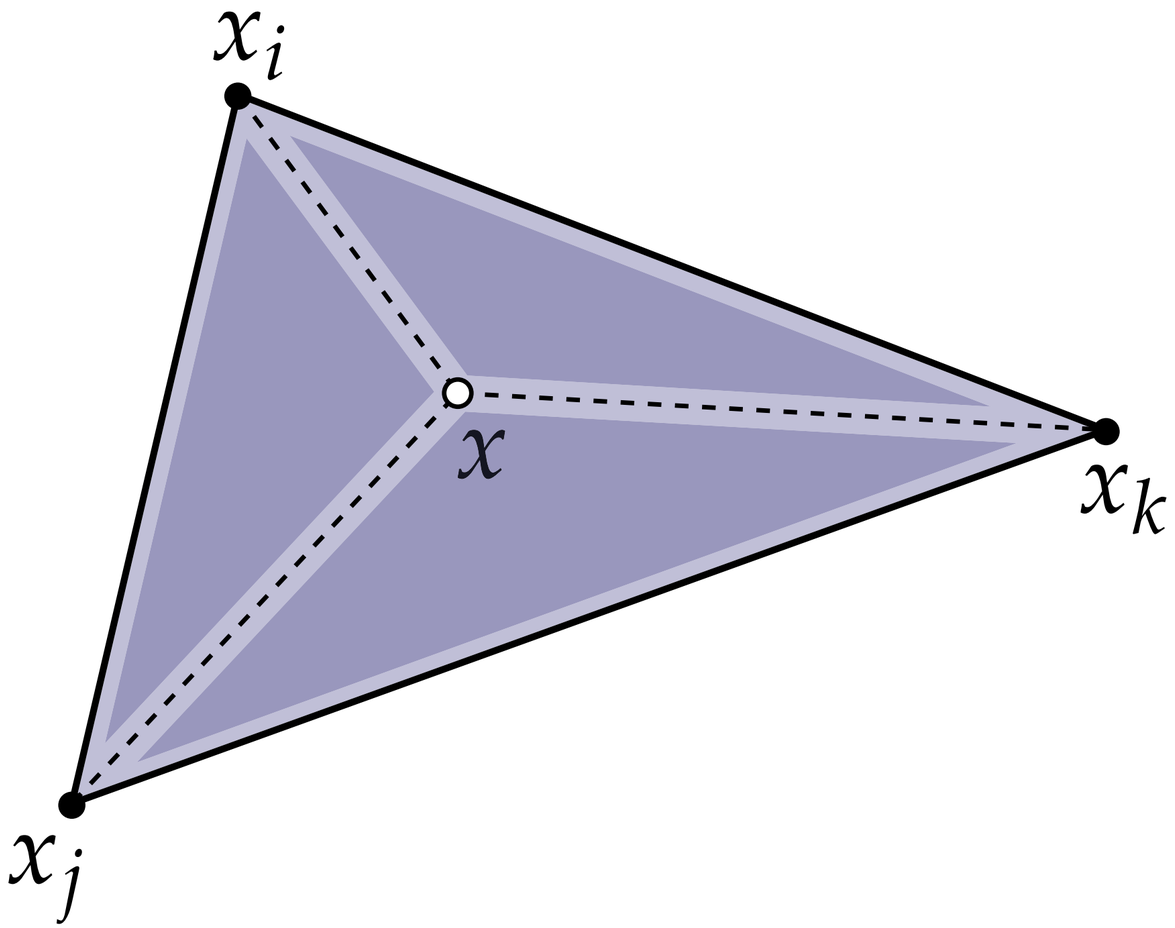


Interpolate by taking linear combination: $\hat{f}(x) = f_i\phi_i + f_j\phi_j + f_k\phi_k$

Q: Is this the same as the (ugly) function we found before?

2D Interpolation, another way

- I claim that we can get the same three basis functions as a ratio of triangle areas:



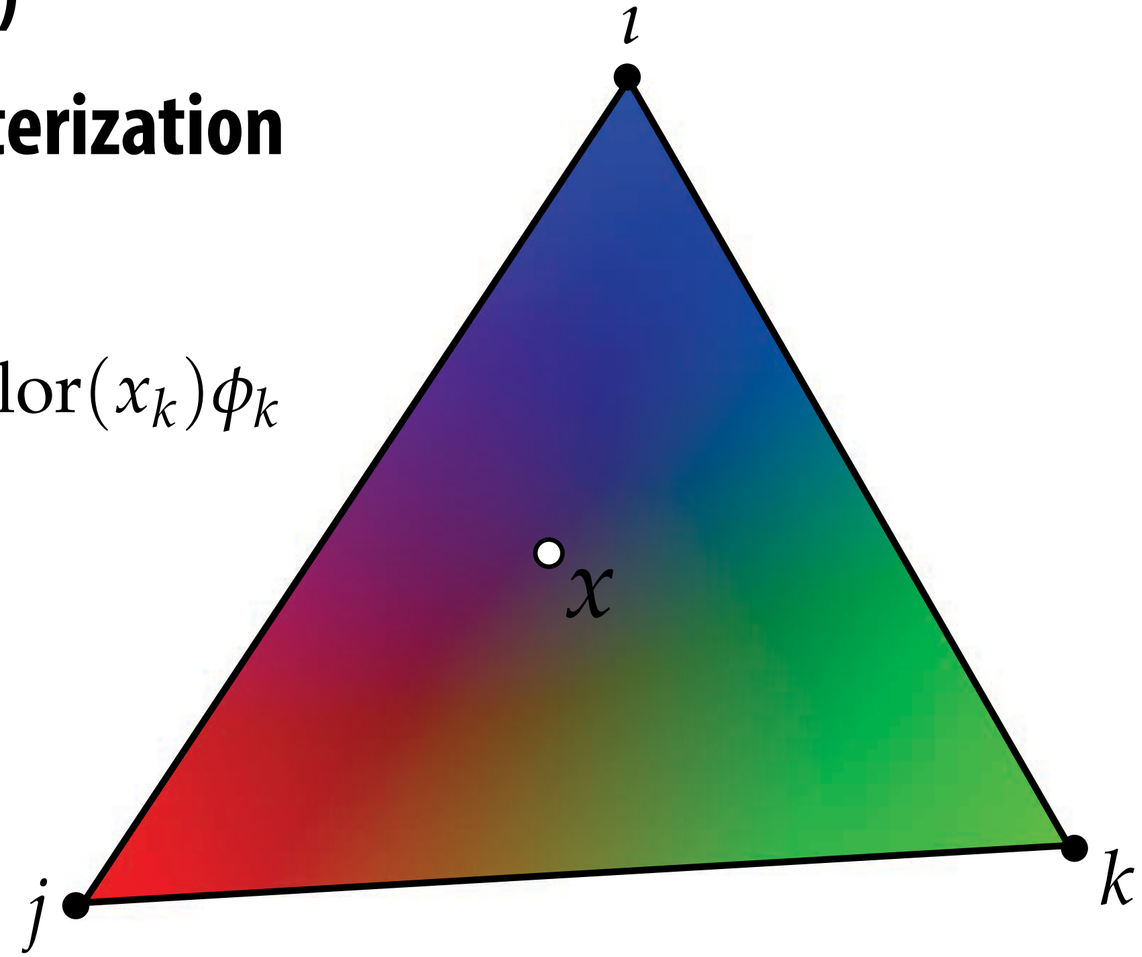
$$\phi_i(x) = \frac{\text{area}(x, x_j, x_k)}{\text{area}(x_i, x_j, x_k)}$$

Q: Do you buy it? :-)

Barycentric Coordinates

- No matter how you compute them, the values of the three functions $\phi_i(x)$, $\phi_j(x)$, $\phi_k(x)$ for a given point are called barycentric coordinates
- Can be used to interpolate any attribute associated with vertices (color, texture coordinates, etc.)
- Importantly, these same three values fall out of the half-plane tests used for triangle rasterization! (Why?)
- Hence, get them for "free" during rasterization

$$\text{color}(x) = \text{color}(x_i)\phi_i + \text{color}(x_j)\phi_j + \text{color}(x_k)\phi_k$$

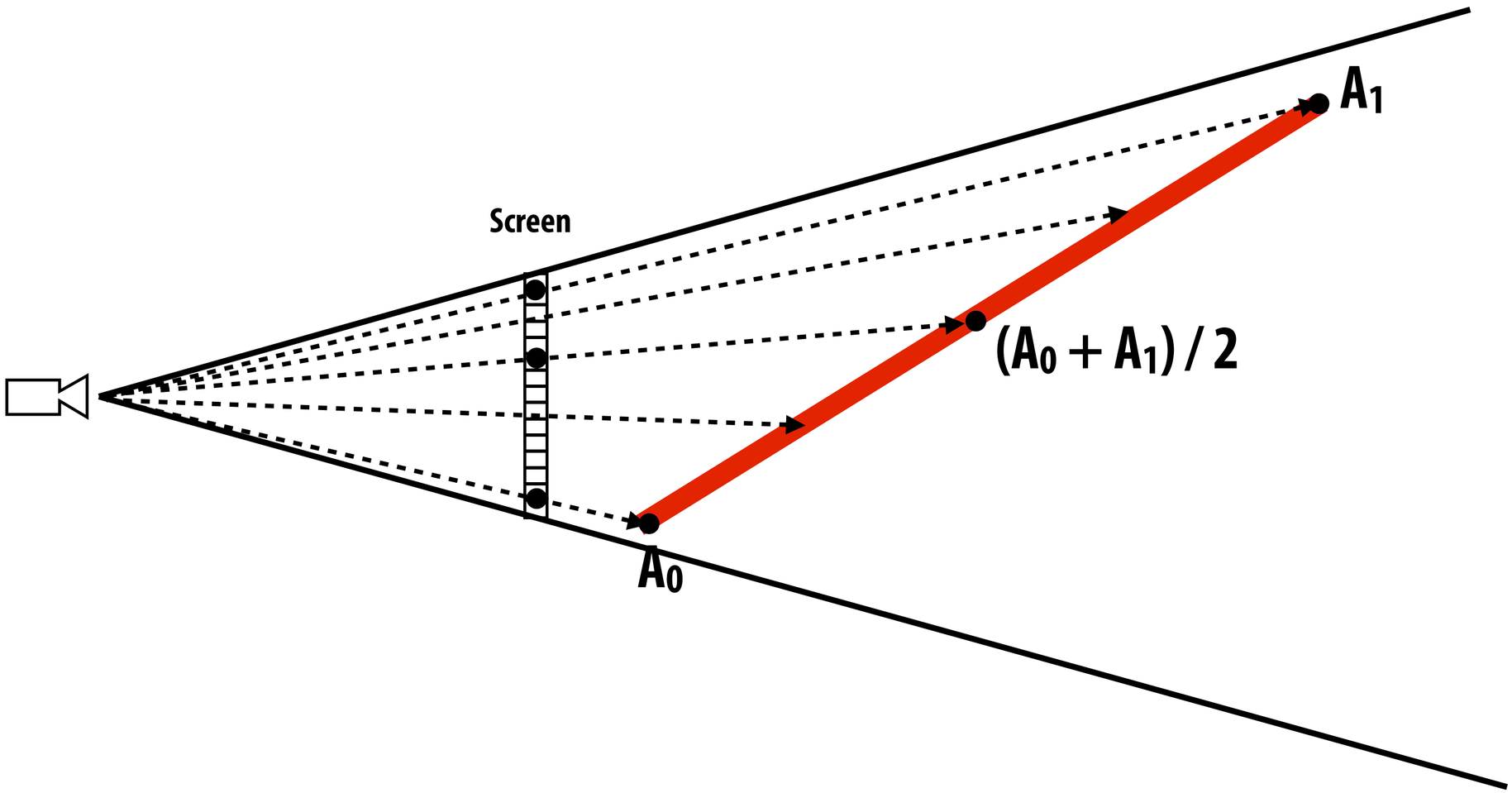


Note: we haven't explained yet how to encode colors as numbers! We'll talk about that in a later lecture…

Perspective-incorrect interpolation

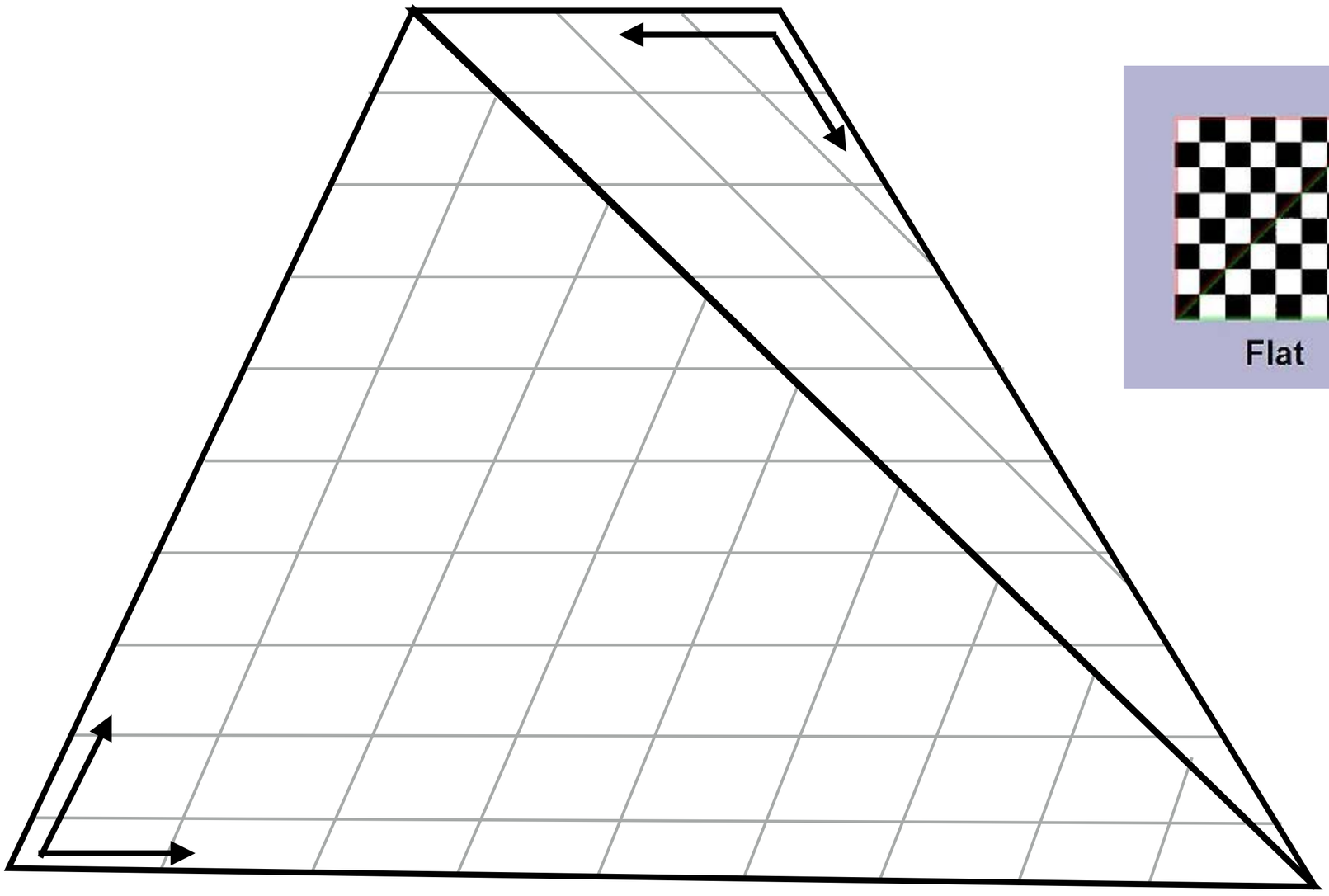
Due to perspective projection (homogeneous divide), barycentric interpolation of values on a triangle with different depths is not an affine function of screen XY coordinates.

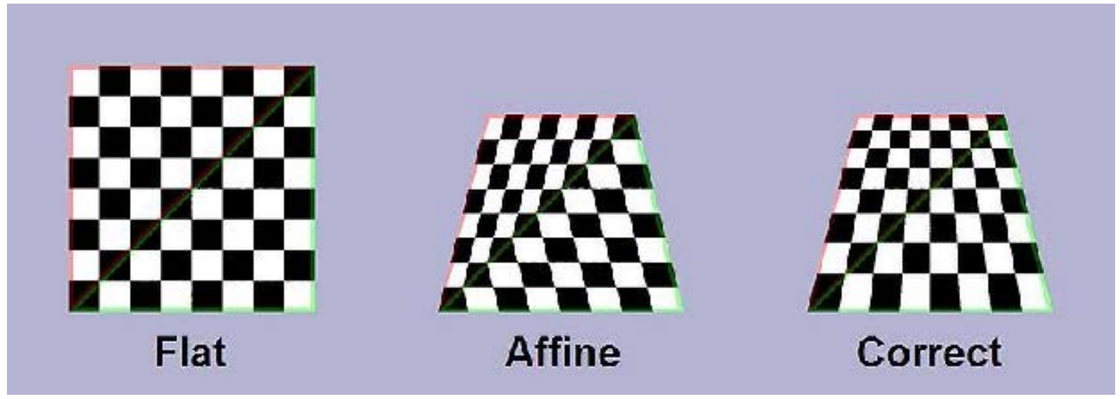
Attribute values must be interpolated linearly in 3D object space.



Example: perspective incorrect interpolation

Good example is quadrilateral split into two triangles:





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

Perspective Correct Interpolation

- Basic recipe:
  - To interpolate some attribute ϕ…
  - Compute depth z at each vertex
  - Evaluate $Z := 1/z$ and $P := \phi/z$ at each vertex
  - Interpolate Z and P using standard (2D) barycentric coords
  - At each *fragment*, divide interpolated P by interpolated Z to get final value

For a derivation, see Low, "Perspective-Correct Interpolation"

Texture Mapping

Many uses of texture mapping

Define variation in surface reflectance

Pattern on ball

Wood grain on floor

Describe surface material properties

Multiple layers of texture maps for color, logos, scratches, etc.

Normal & Displacement Mapping

normal mapping

Use texture value to perturb surface normal to "fake" appearance of a bumpy surface (note smooth silhouette/shadow reveals that surface geometry is not actually bumpy!)

displacement mapping

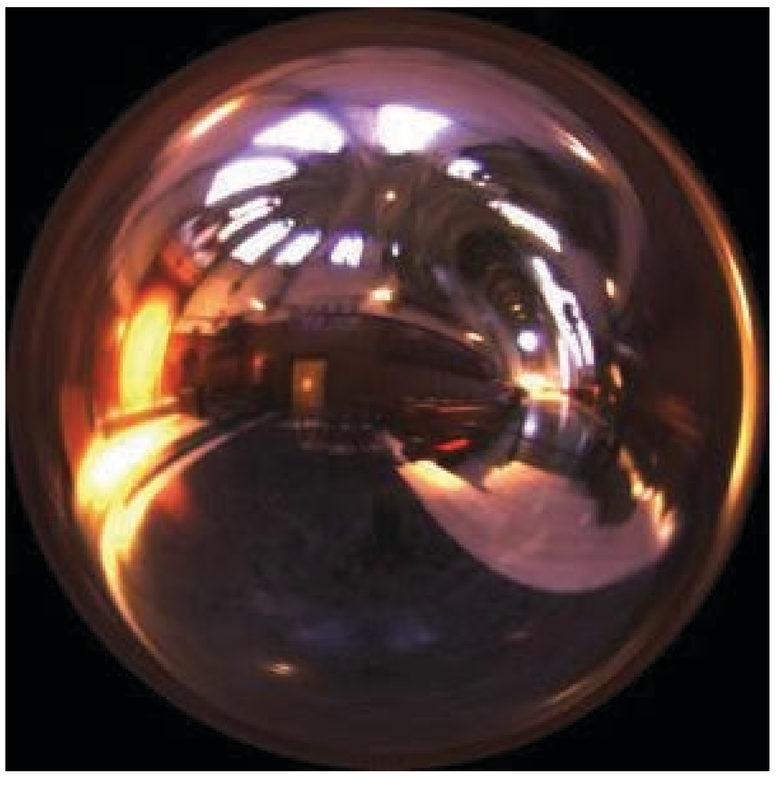
dice up surface geometry into tiny triangles & offset positions according to texture values (note bumpy silhouette and shadow boundary)

Represent precomputed lighting and shadows

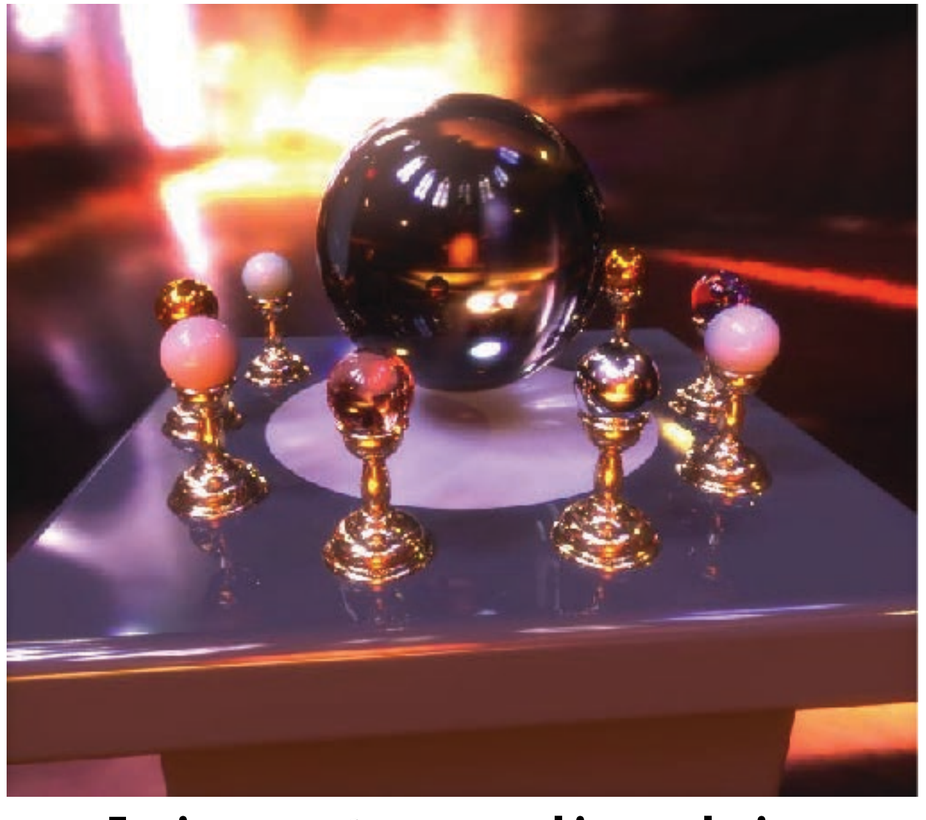
Original model

With ambient occlusion

Extracted ambient occlusion map



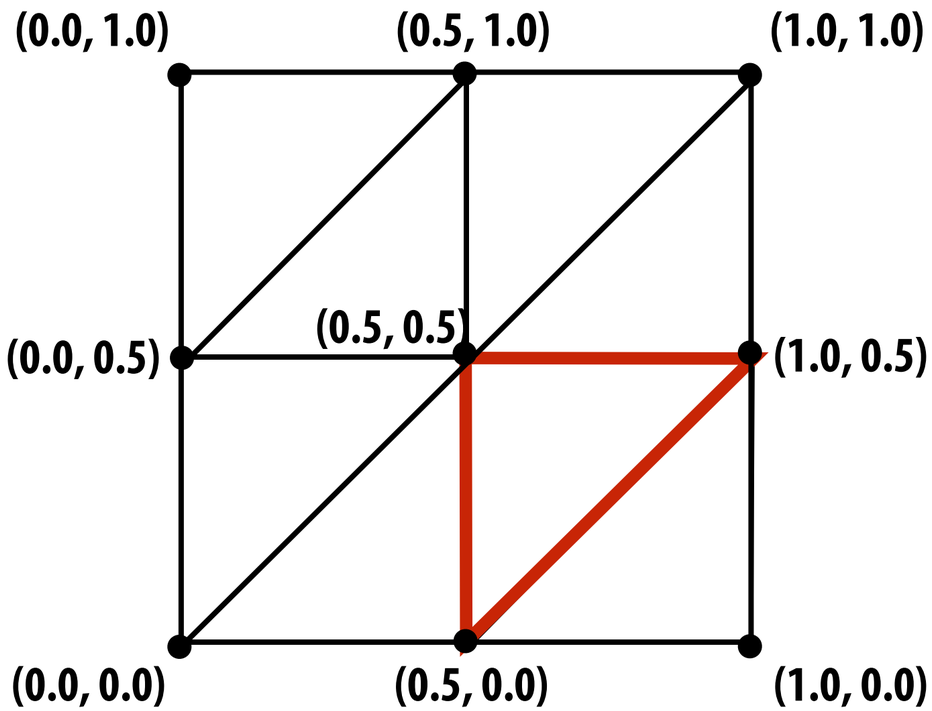
Grace Cathedral environment map



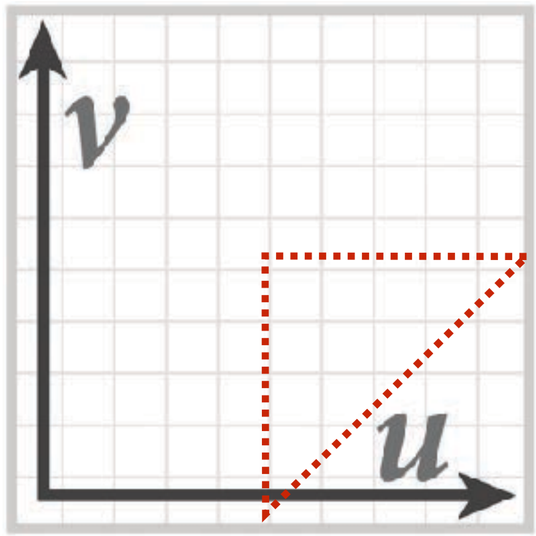
Environment map used in rendering

Texture coordinates

"Texture coordinates" define a mapping from surface coordinates (points on triangle) to points in texture domain.

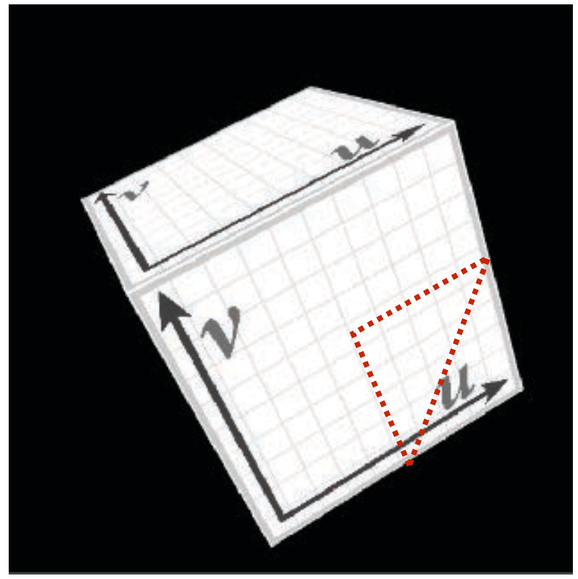


Eight triangles (one face of cube) with surface parameterization provided as per-vertex texture coordinates.



myTex(u,v) is a function defined on the $[0,1]^2$ domain (represented by 2048x2048 image)

Location of highlighted triangle in texture space shown in red.



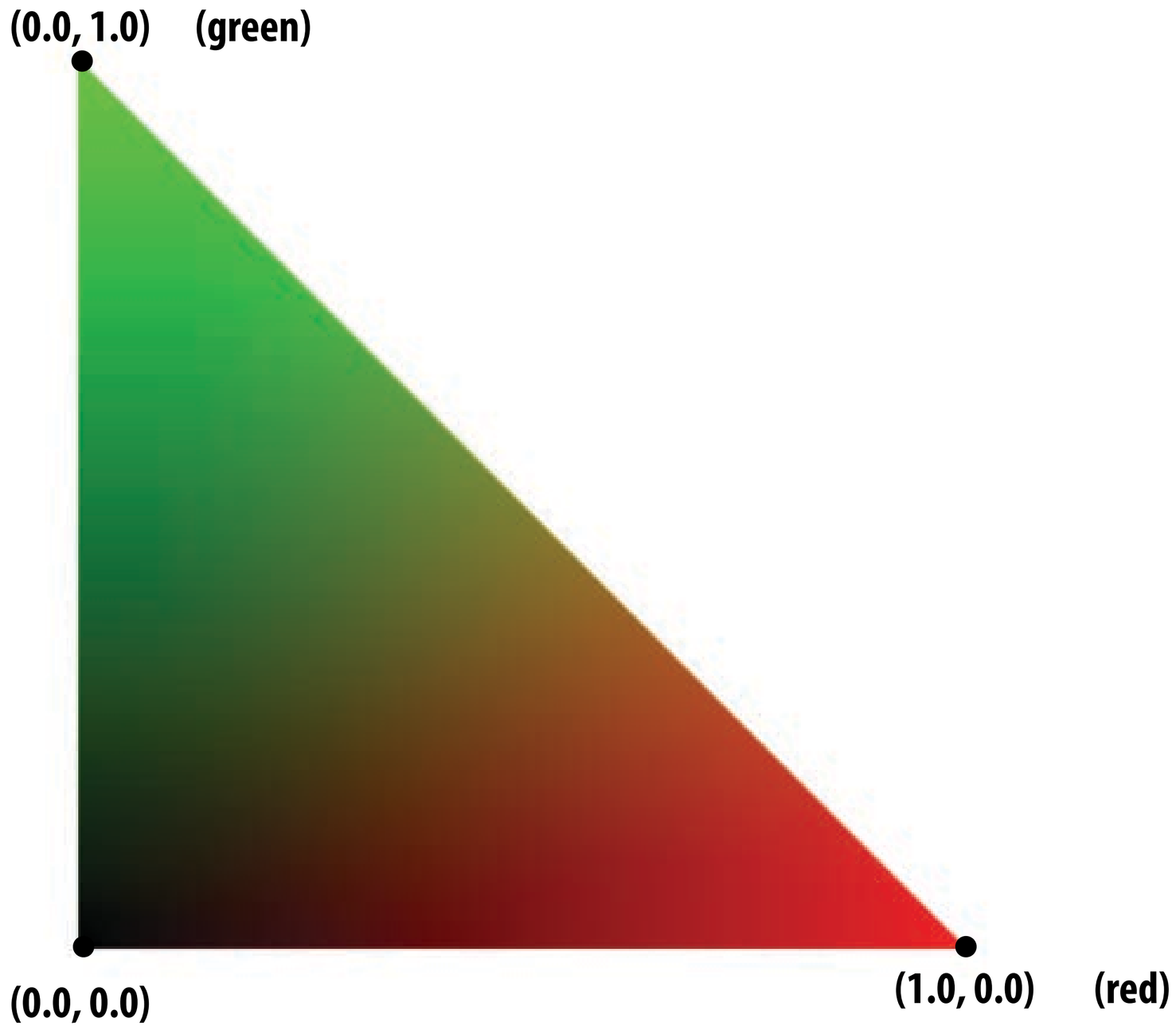
Final rendered result (entire cube shown).

Location of triangle after projection onto screen shown in red.

(We'll assume surface-to-texture space mapping is provided as per vertex values)

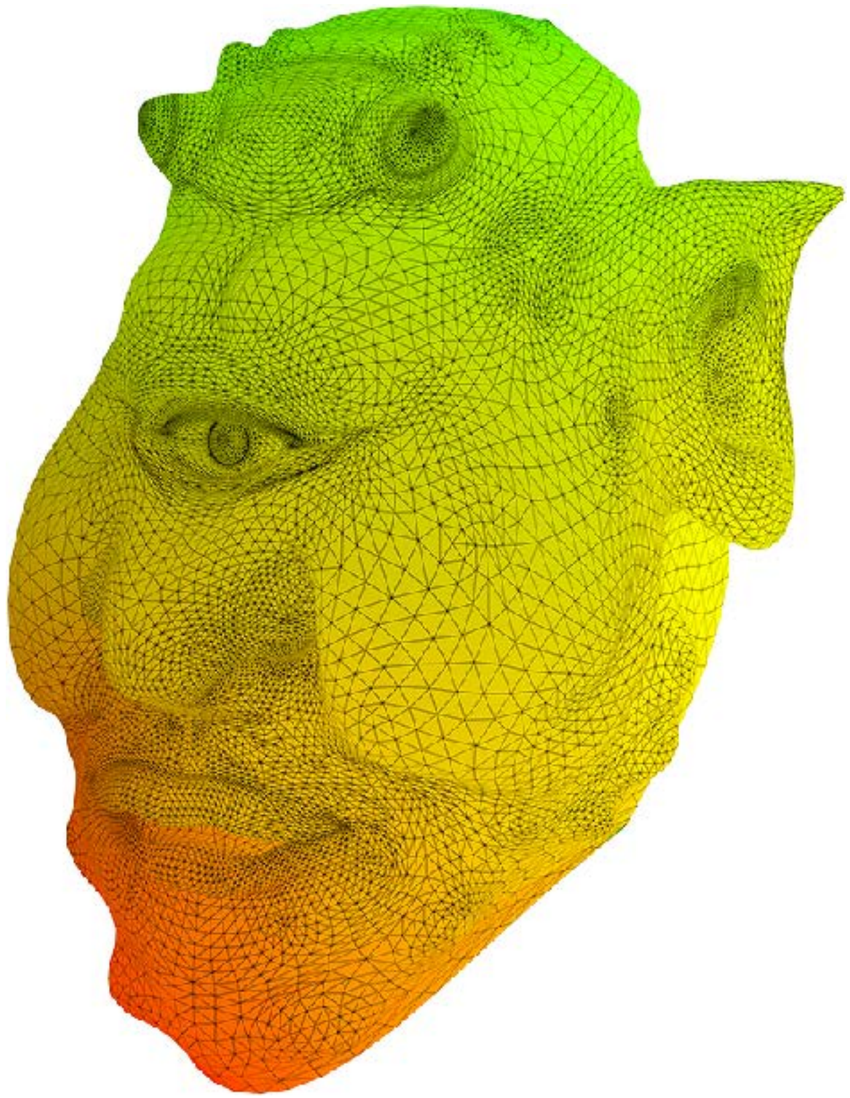
Visualization of texture coordinates

Texture coordinates linearly interpolated over triangle

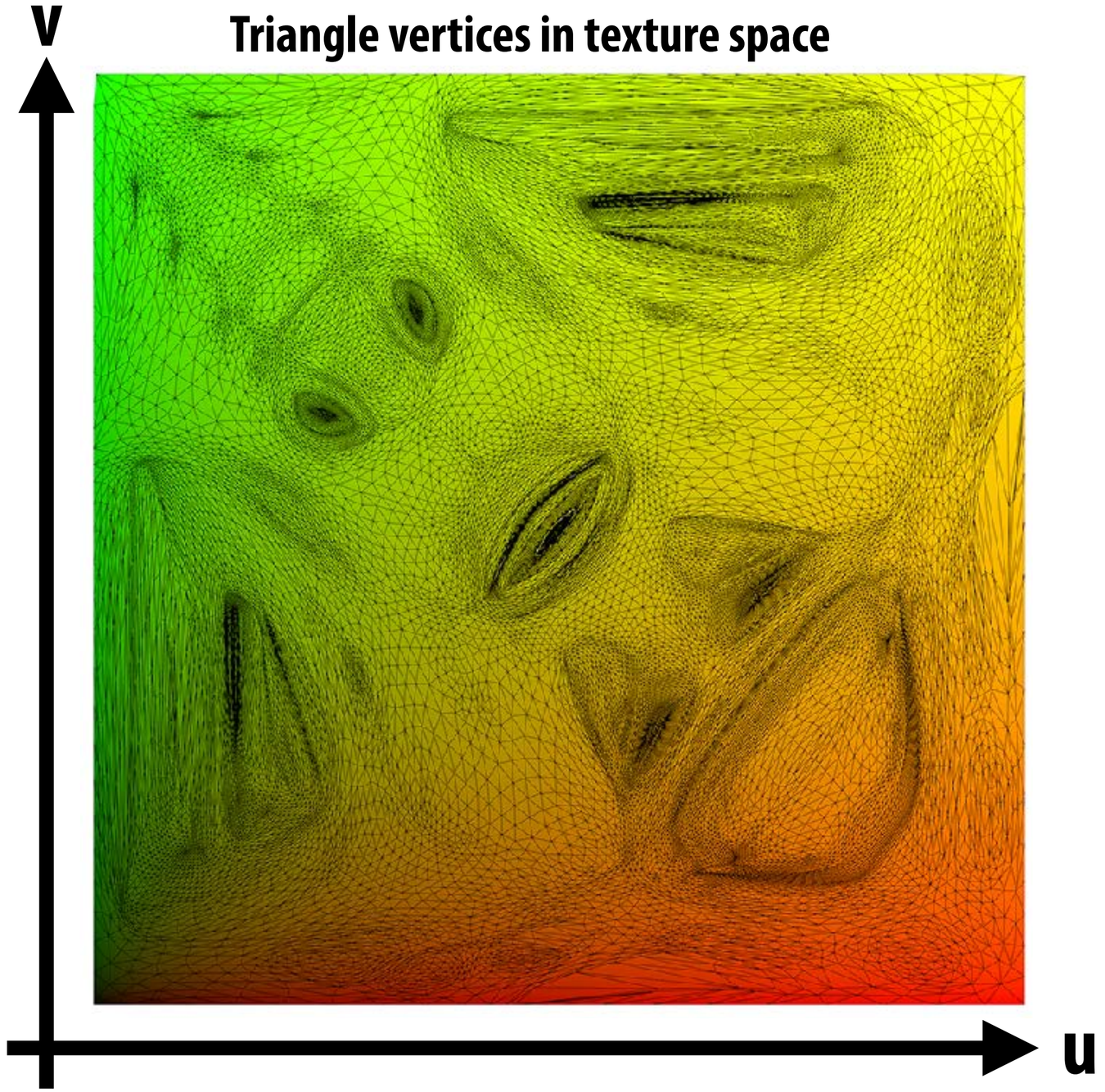


More complex mapping

Visualization of texture coordinates



Triangle vertices in texture space



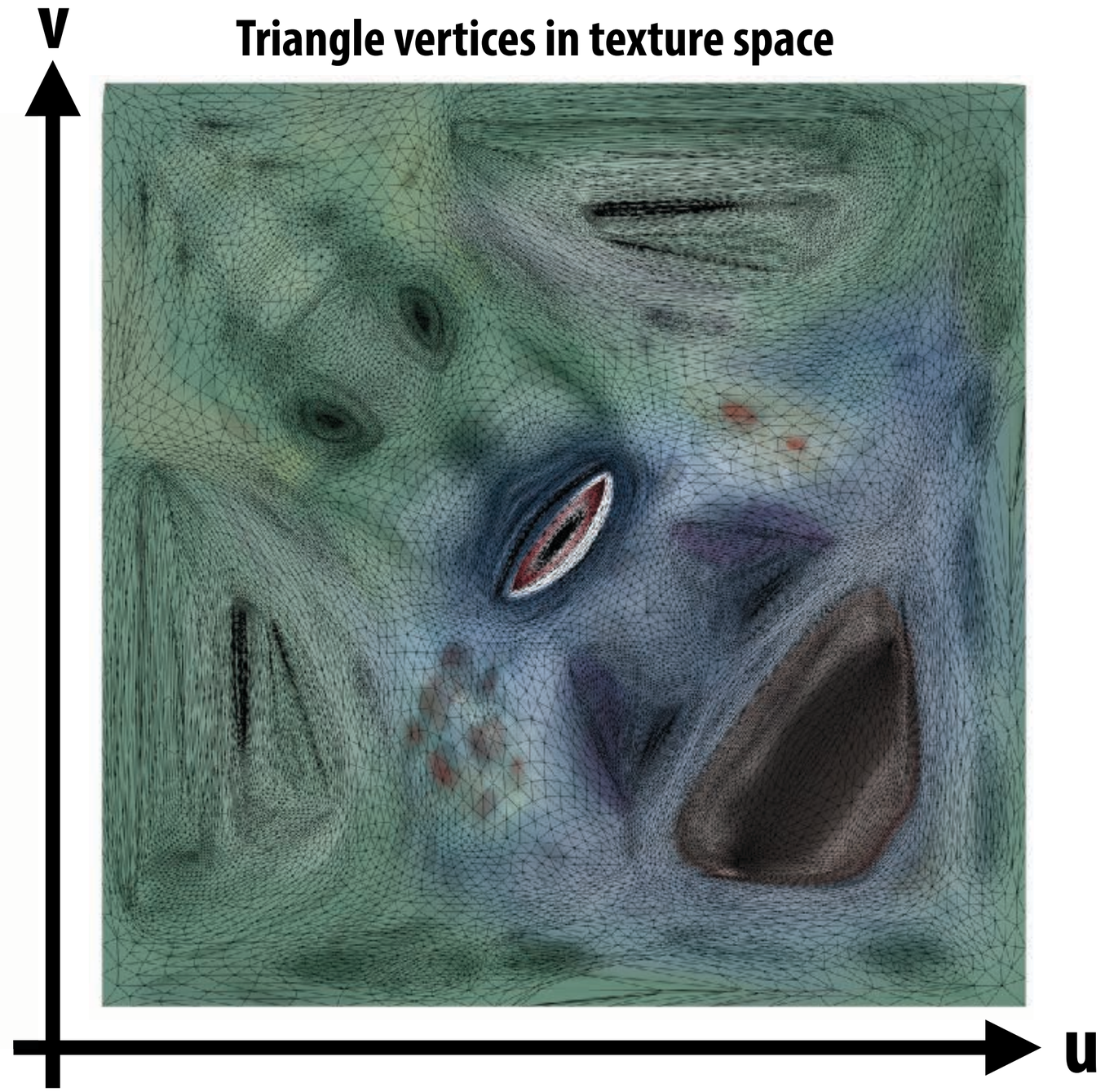
Each vertex has a coordinate (u,v) in texture space.

(Actually coming up with these coordinates is another story!)

Texture mapping adds detail

Rendered result

Triangle vertices in texture space

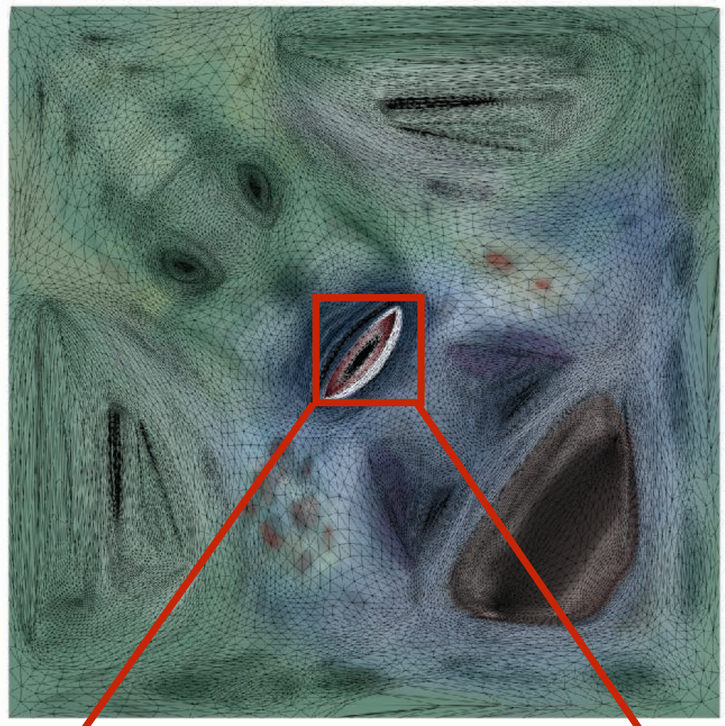


Texture mapping adds detail

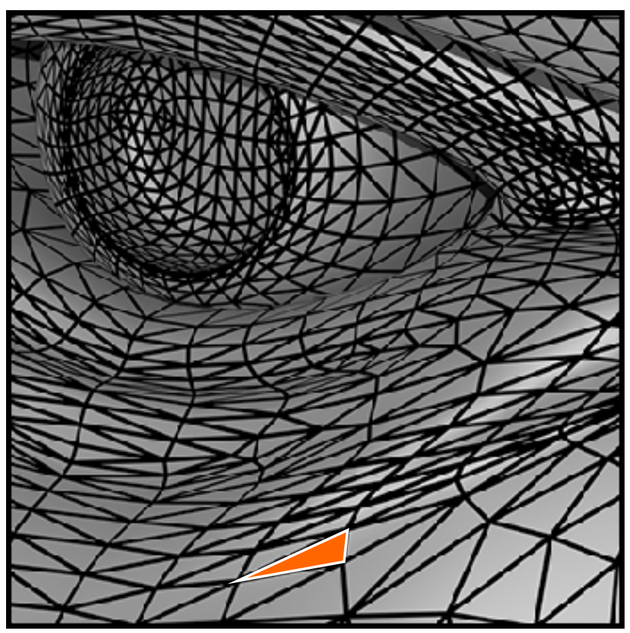
rendering without texture

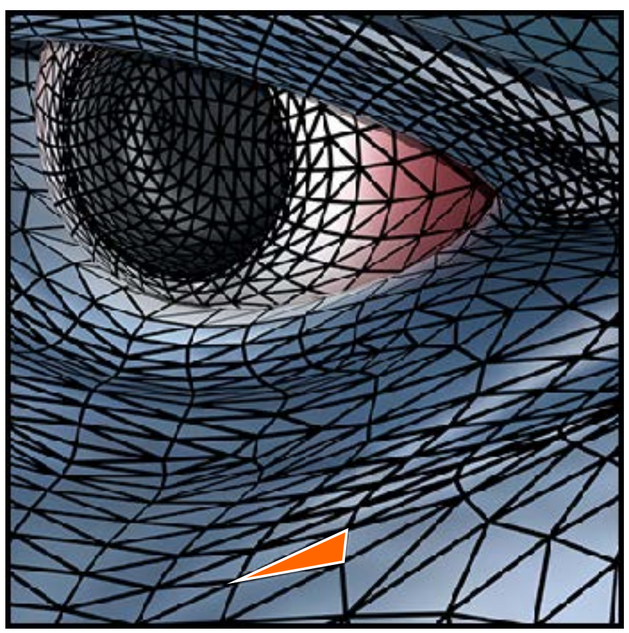
rendering with texture

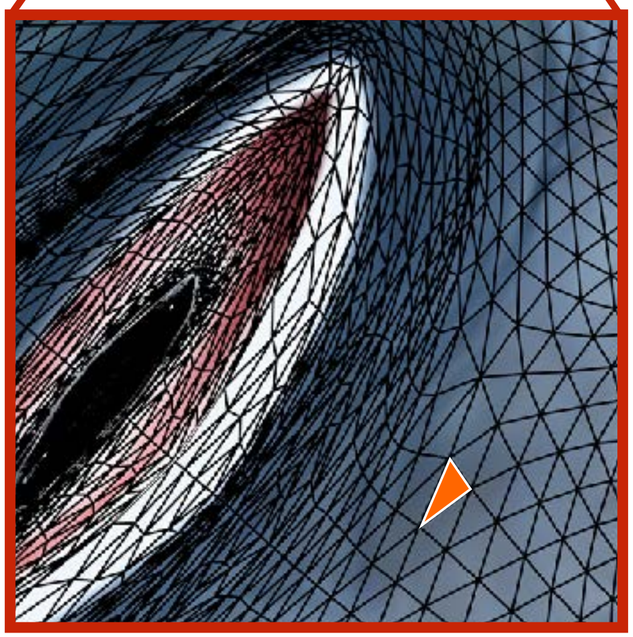
texture image



zoom

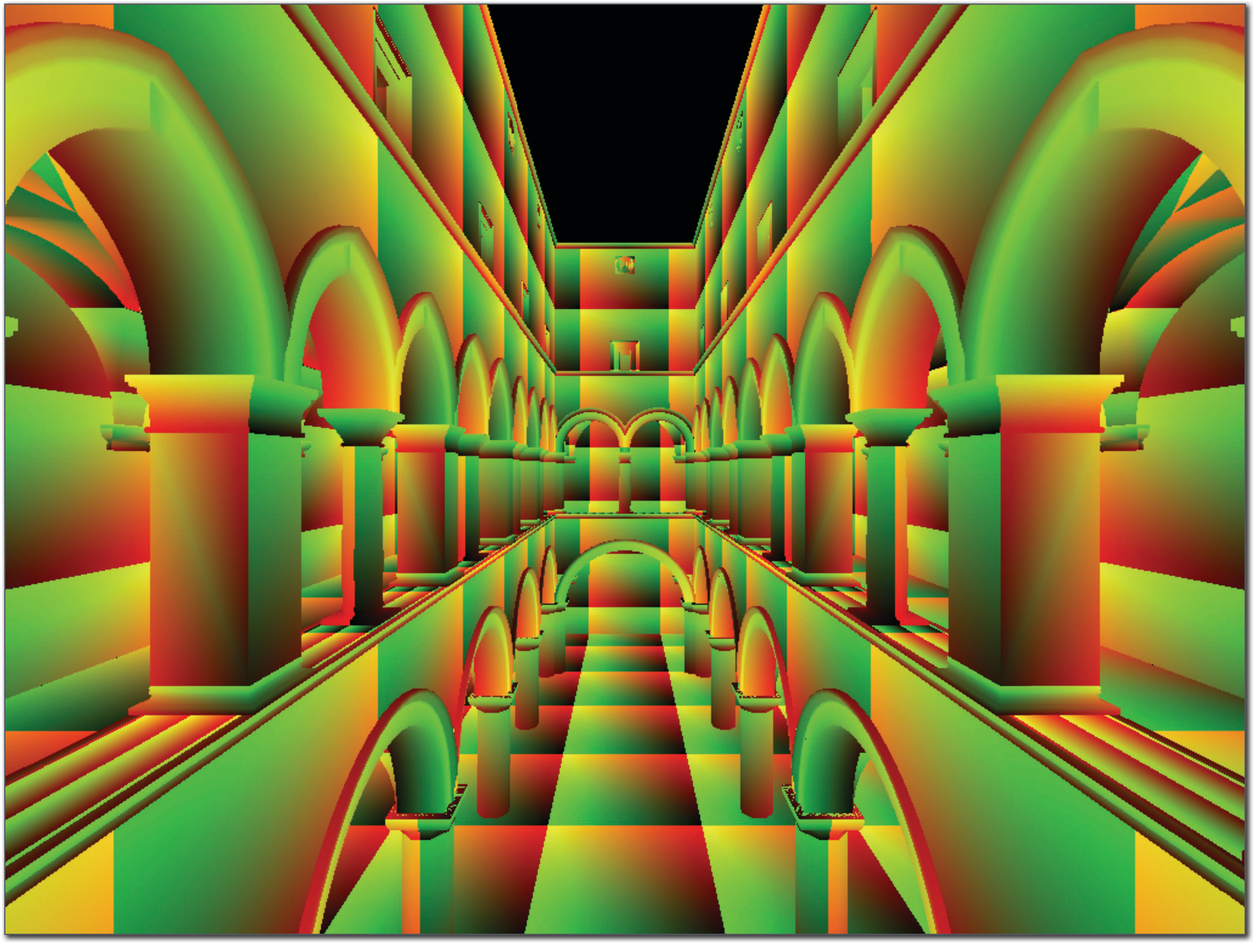






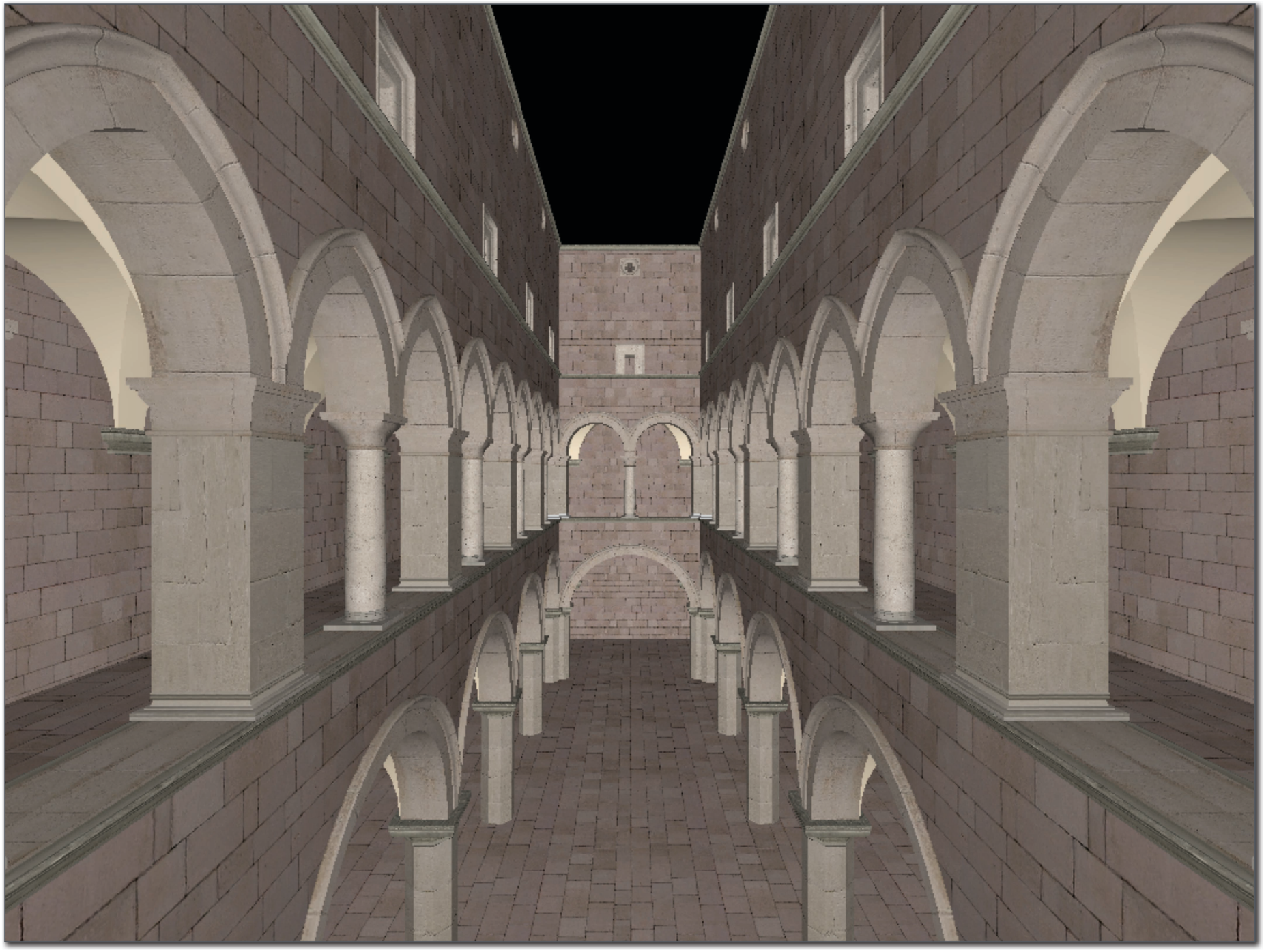
Each triangle "copies" a piece of the image back to the surface.

Another example: Sponza



Notice texture coordinates repeat over surface.

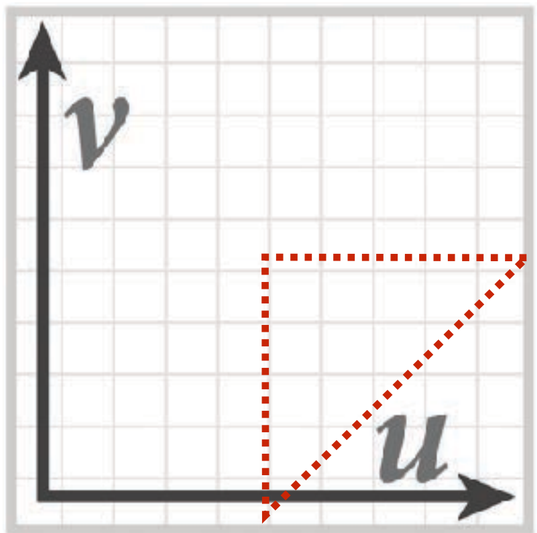
Textured Sponza

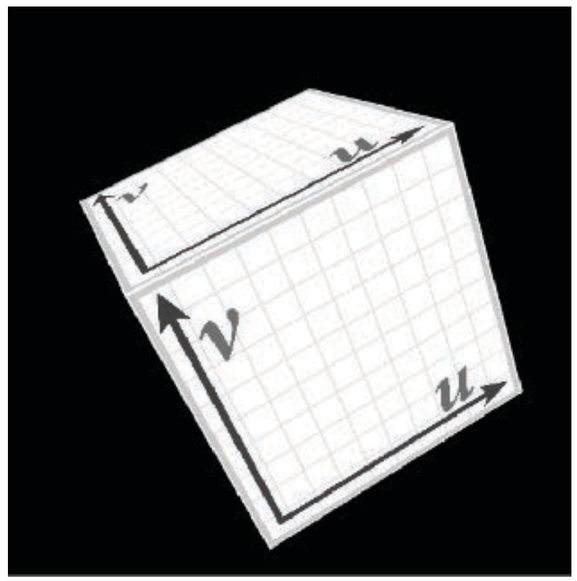


Example textures used in Sponza

Texture Sampling 101

- Basic algorithm for mapping texture to surface:
  - Interpolate U and V coordinates across triangle
  - For each fragment
    - Sample (evaluate) texture at (U,V)
    - Set color of fragment to sampled texture value

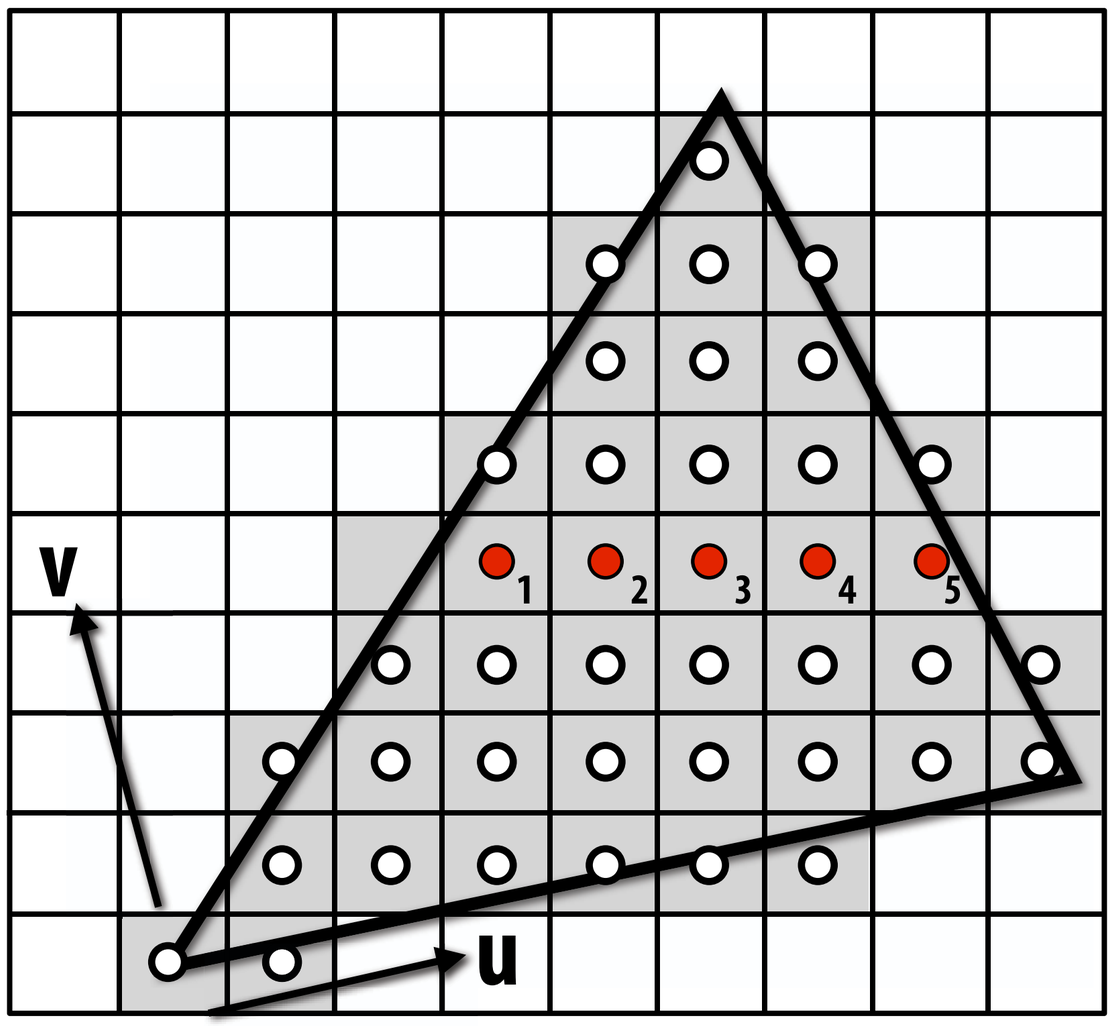




…sadly not this easy in general!

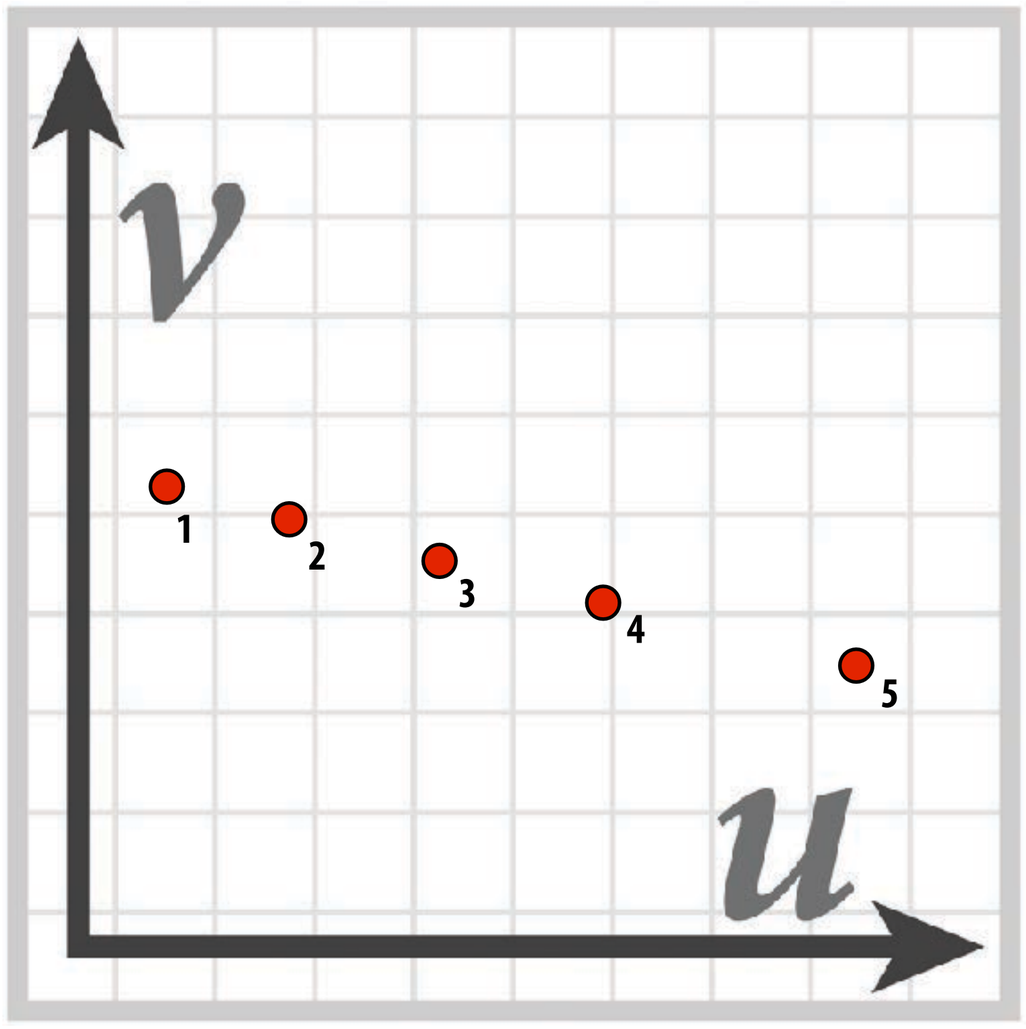
Texture space samples

Sample positions in XY screen space



Sample positions are uniformly distributed in screen space (rasterizer samples triangle's appearance at these locations)

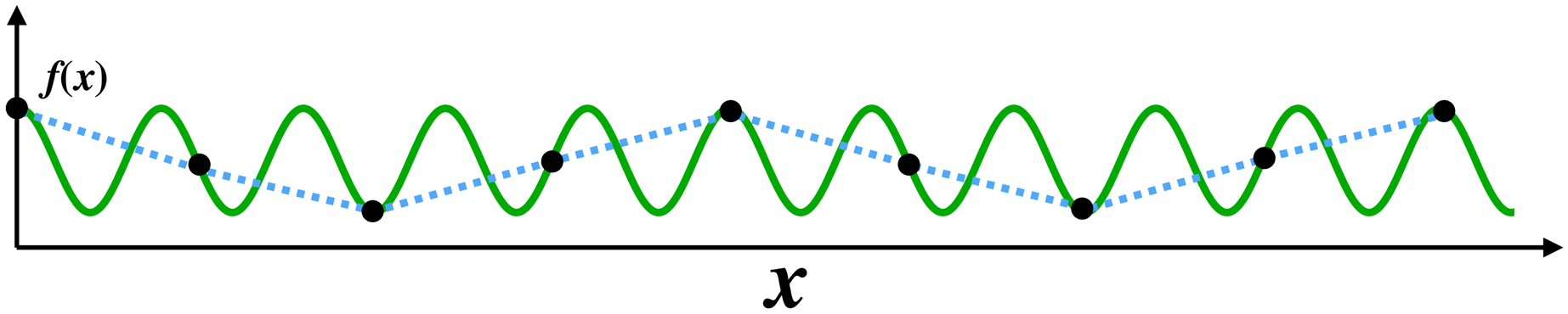
Sample positions in texture space



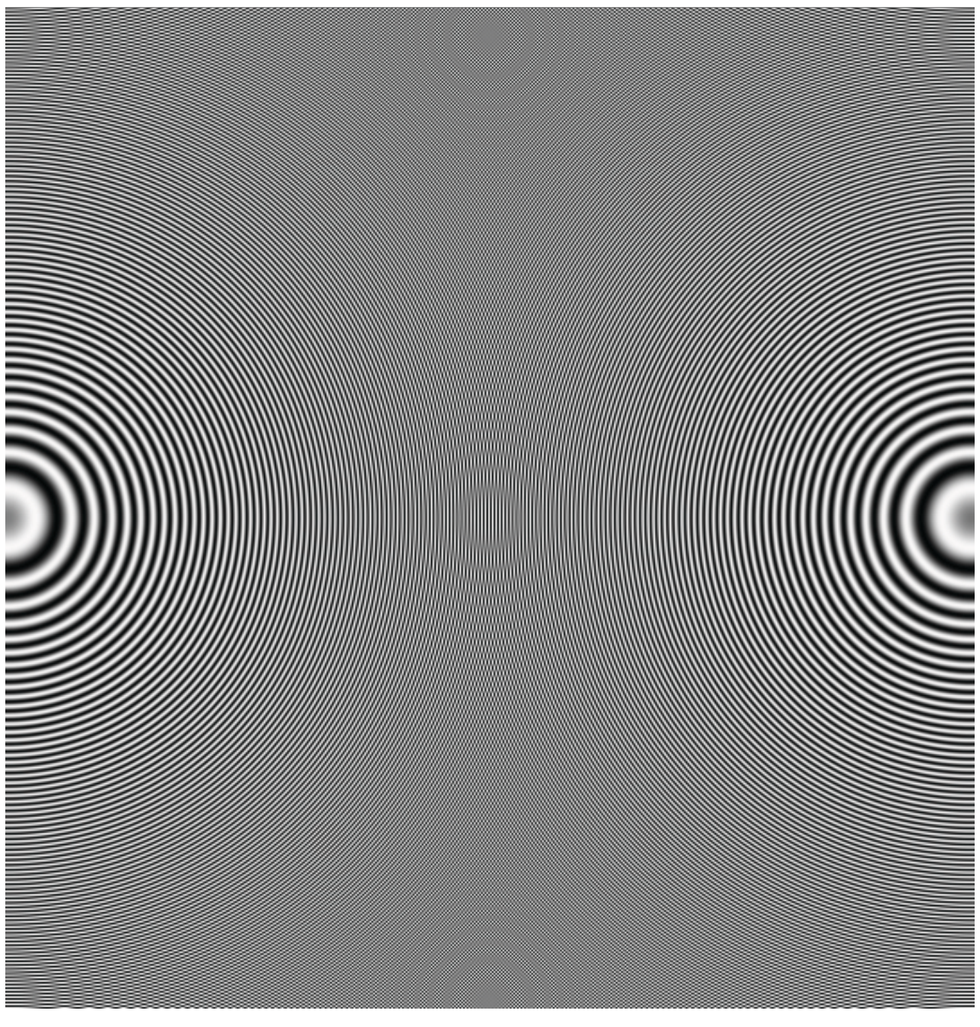
Texture sample positions in texture space (texture function is sampled at these locations)

Recall: aliasing

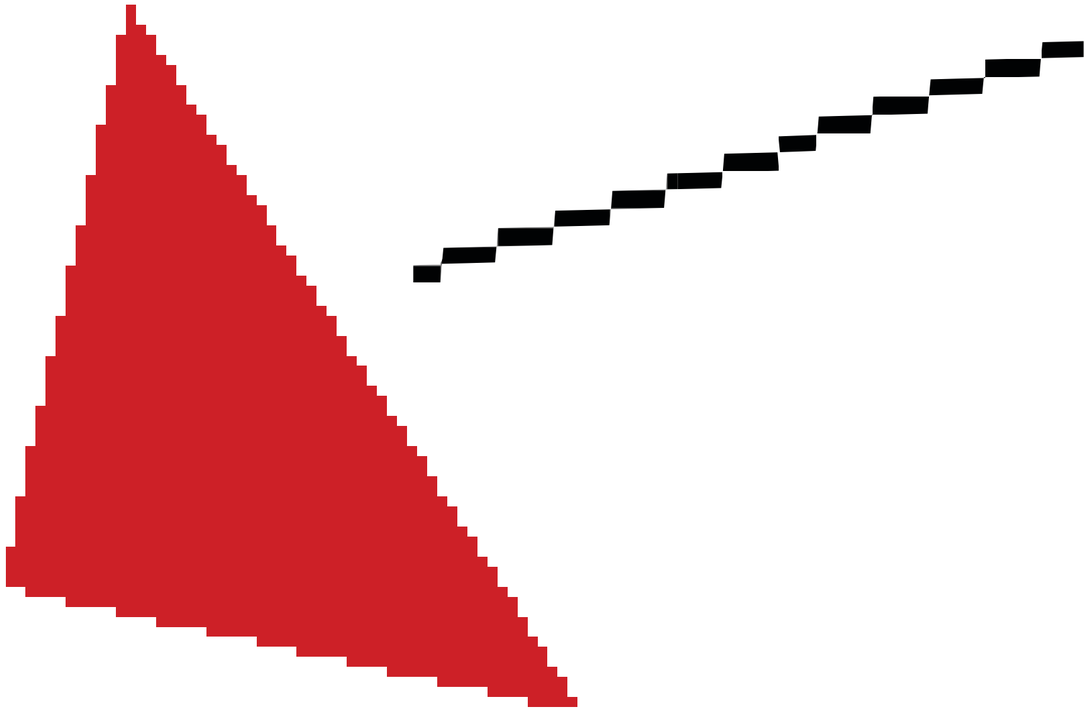
Undersampling a high-frequency signal can result in aliasing



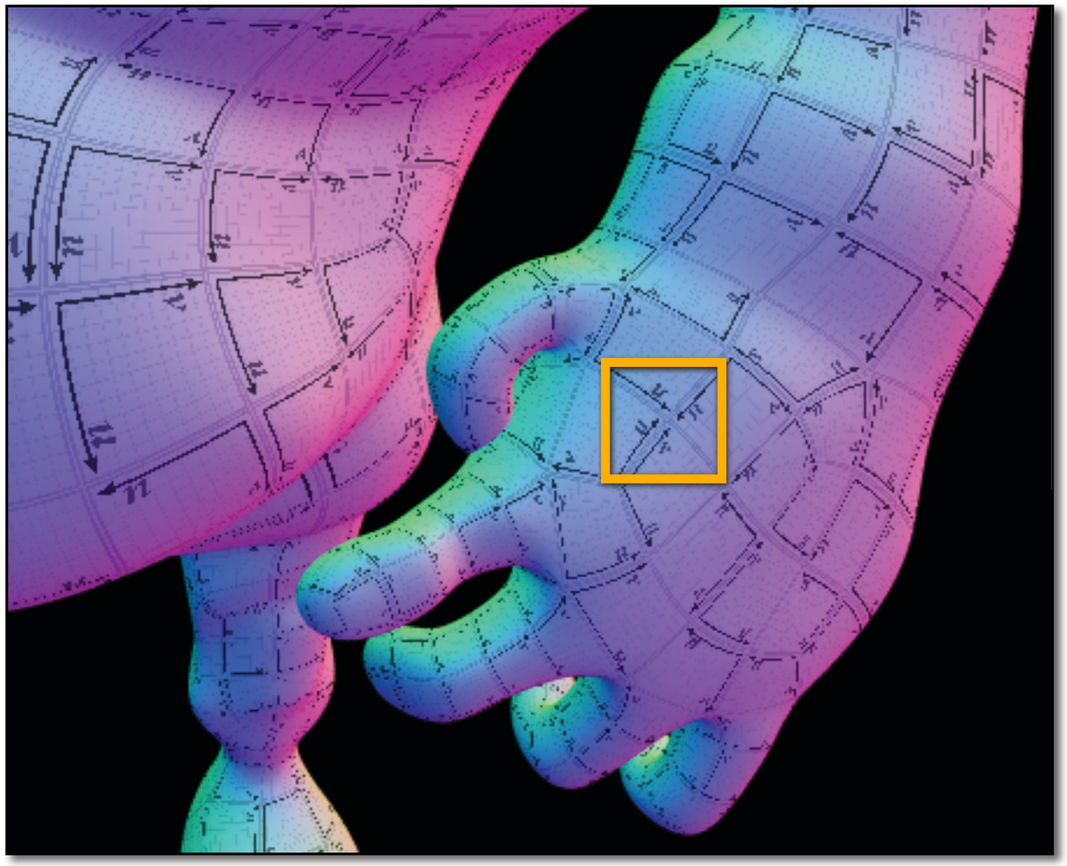
1D example



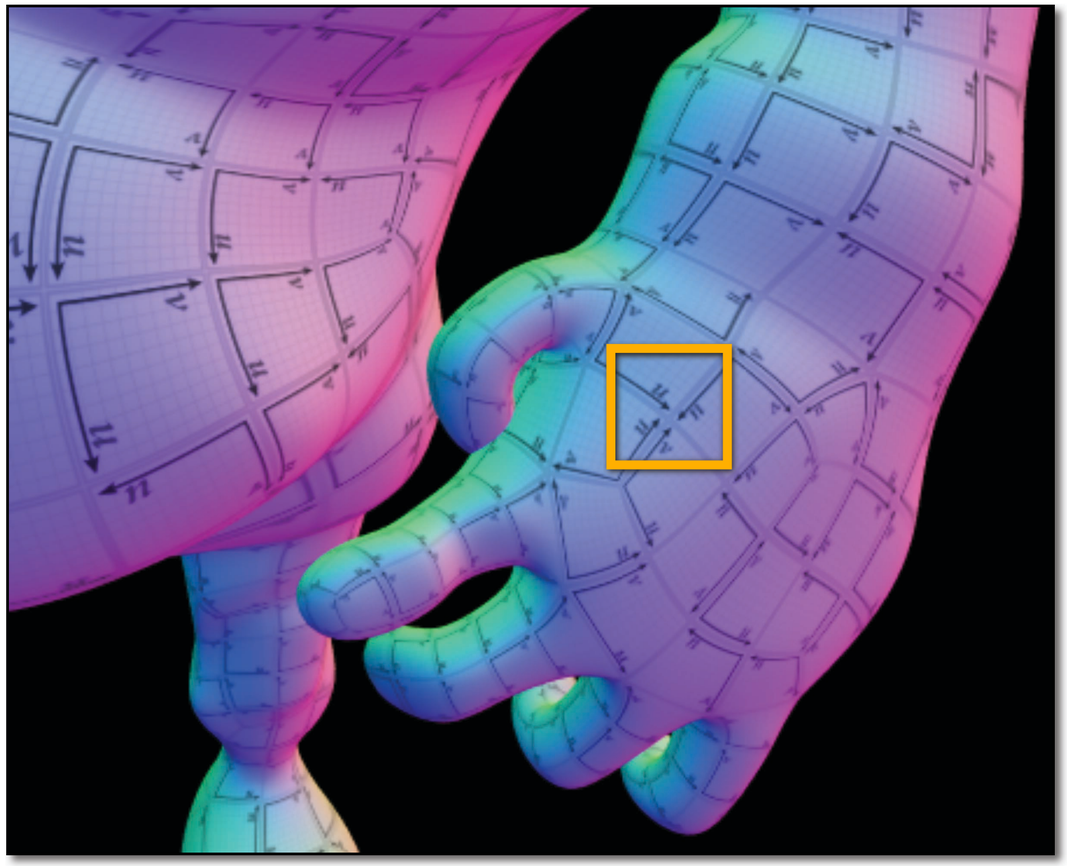
2D examples:
Moiré patterns, jaggies



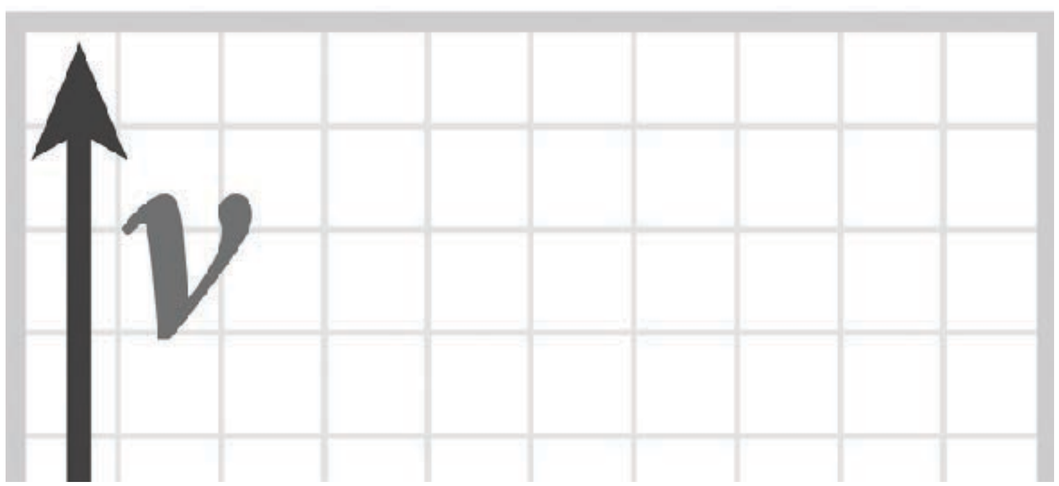
Aliasing due to undersampling texture



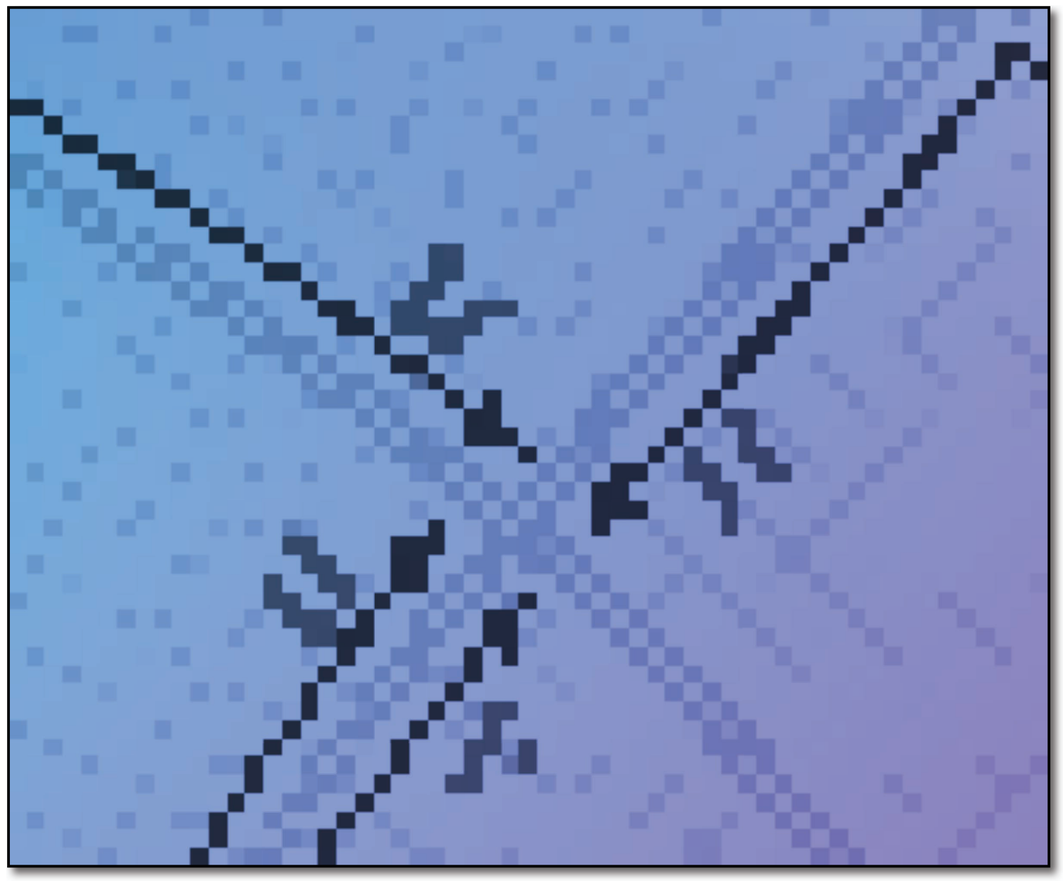
No pre-filtering of texture data
(resulting image exhibits aliasing)



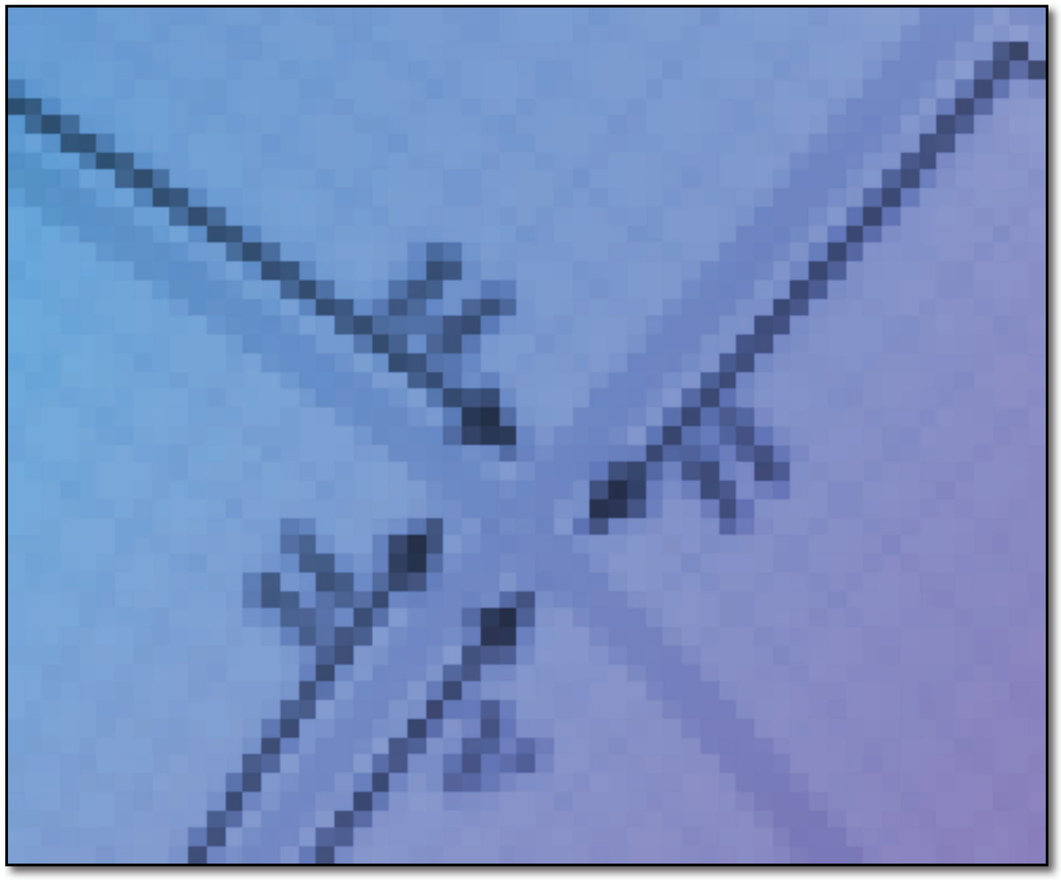
Rendering using pre-filtered texture data



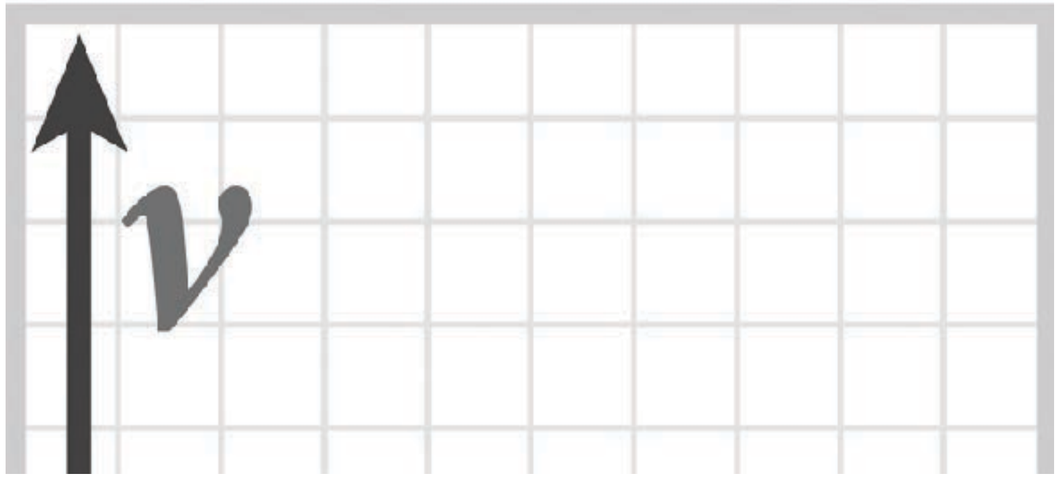
Aliasing due to undersampling (zoom)



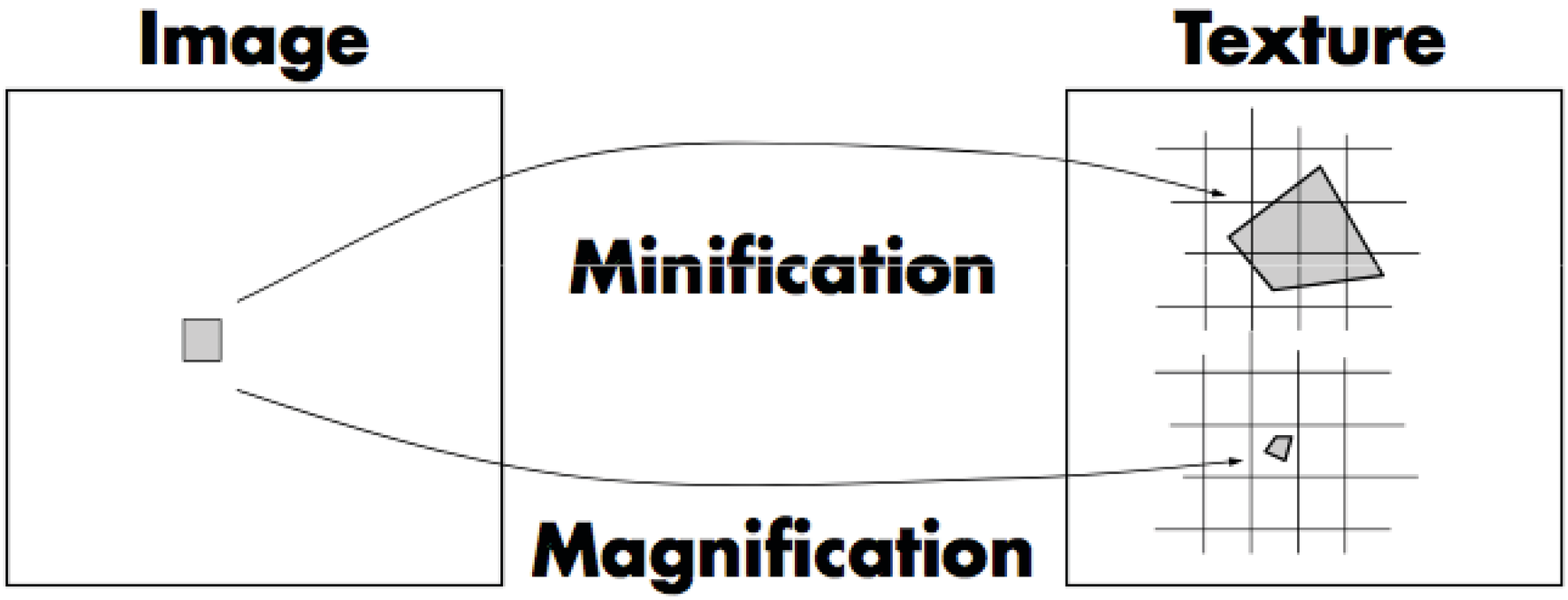
No pre-filtering of texture data
(resulting image exhibits aliasing)



Rendering using pre-filtered texture data



Filtering textures

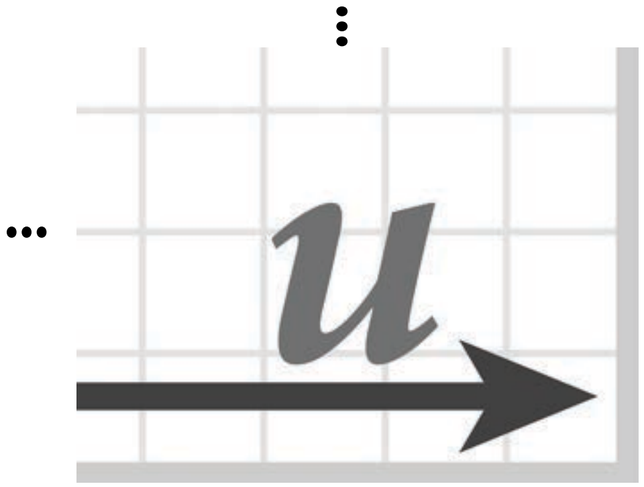


- **Minification:**
  - Area of screen pixel maps to large region of texture (filtering required -- averaging)
  - One texel corresponds to far less than a pixel on screen
  - Example: when scene object is very far away

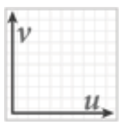
- **Magnification:**
  - Area of screen pixel maps to tiny region of texture (interpolation required)
  - One texel maps to many screen pixels
  - Example: when camera is very close to scene object (need higher resolution texture map)

Figure credit: Akeley and Hanrahan

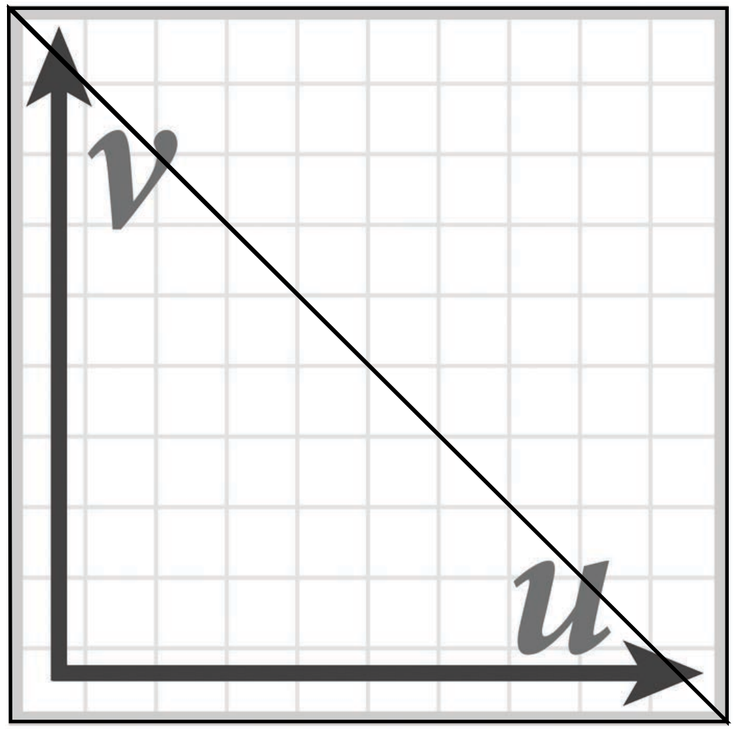
Filtering textures



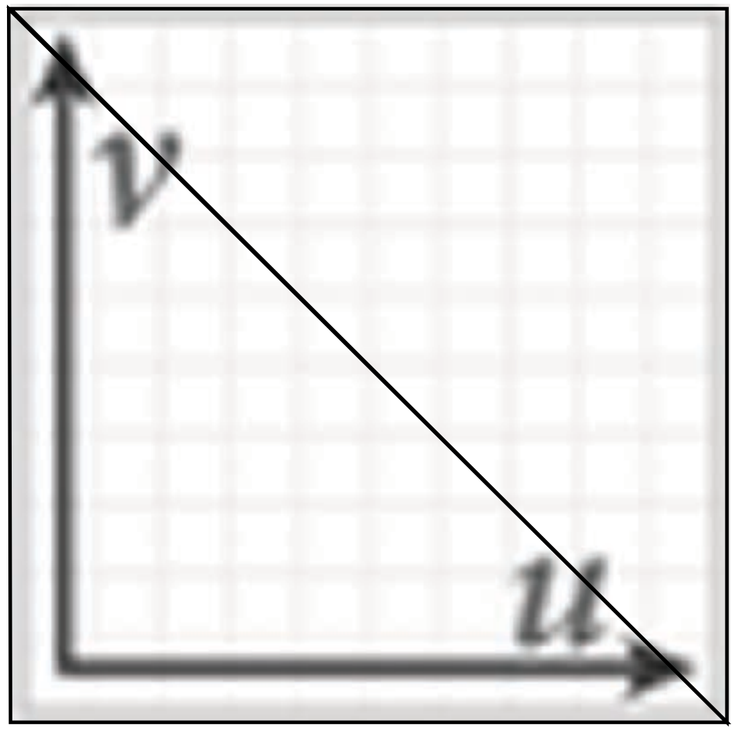
Actual texture: 700x700 image
(only a crop is shown)



Actual texture: 64x64 image

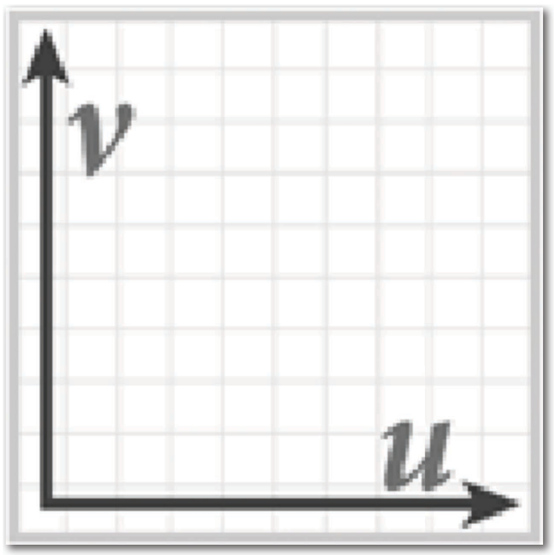


Texture minification

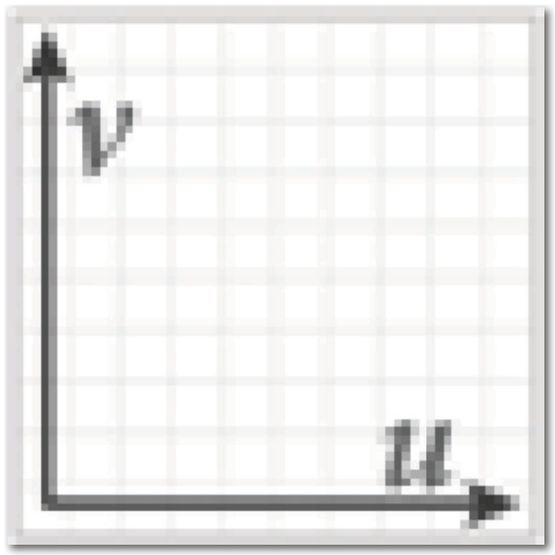


Texture magnification

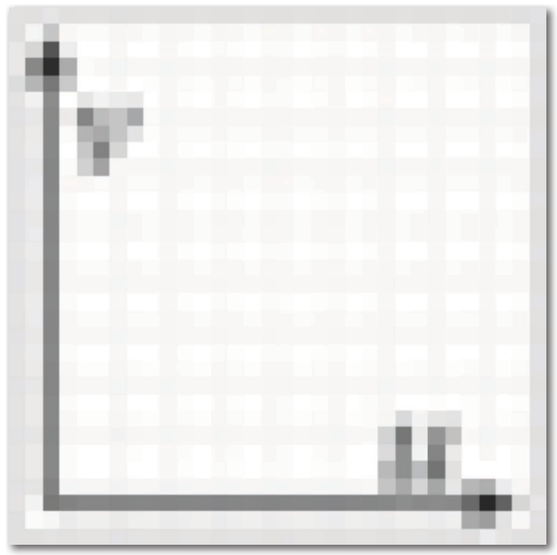
Mipmap (L. Williams 83)



Level 0 = 128x128



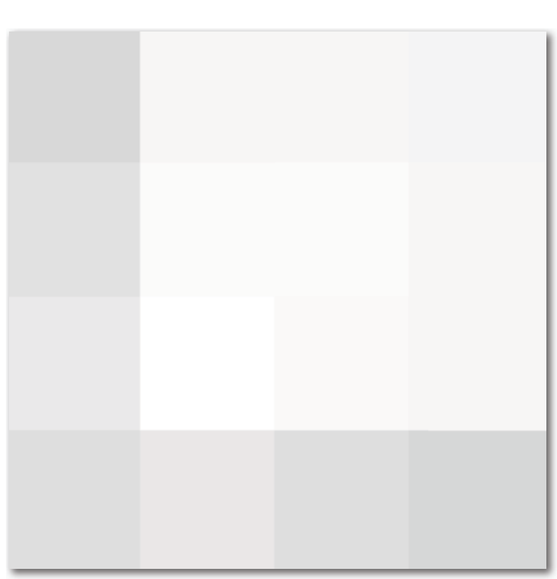
Level 1 = 64x64



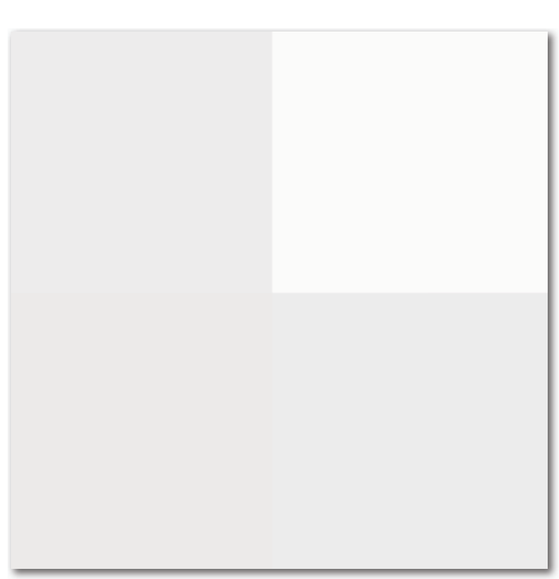
Level 2 = 32x32

Level 3 = 16x16

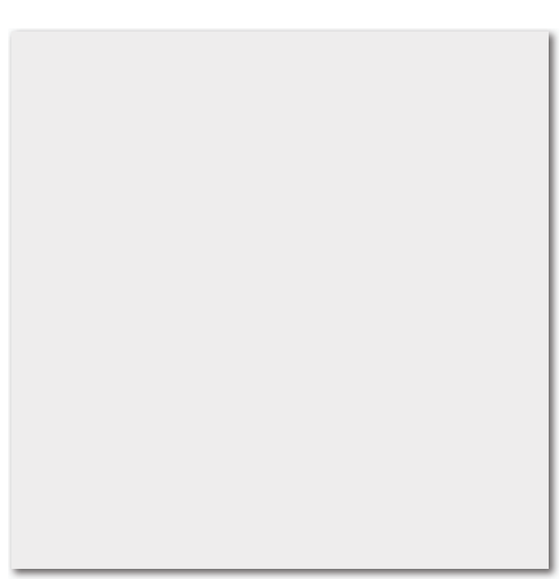
Level 4 = 8x8



Level 5 = 4x4



Level 6 = 2x2



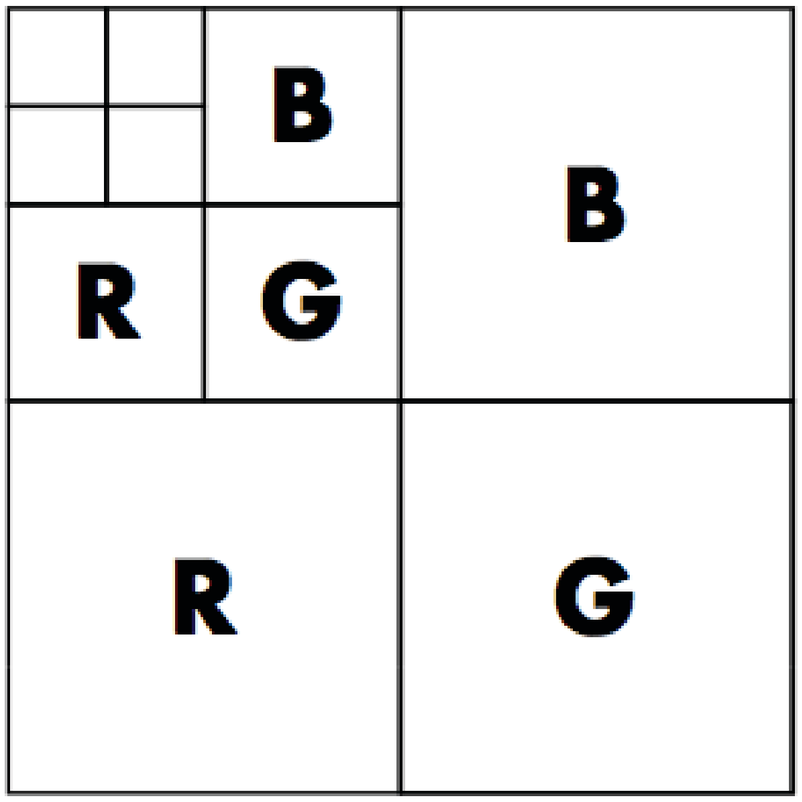
Level 7 = 1x1

Idea: prefilter texture data to remove high frequencies

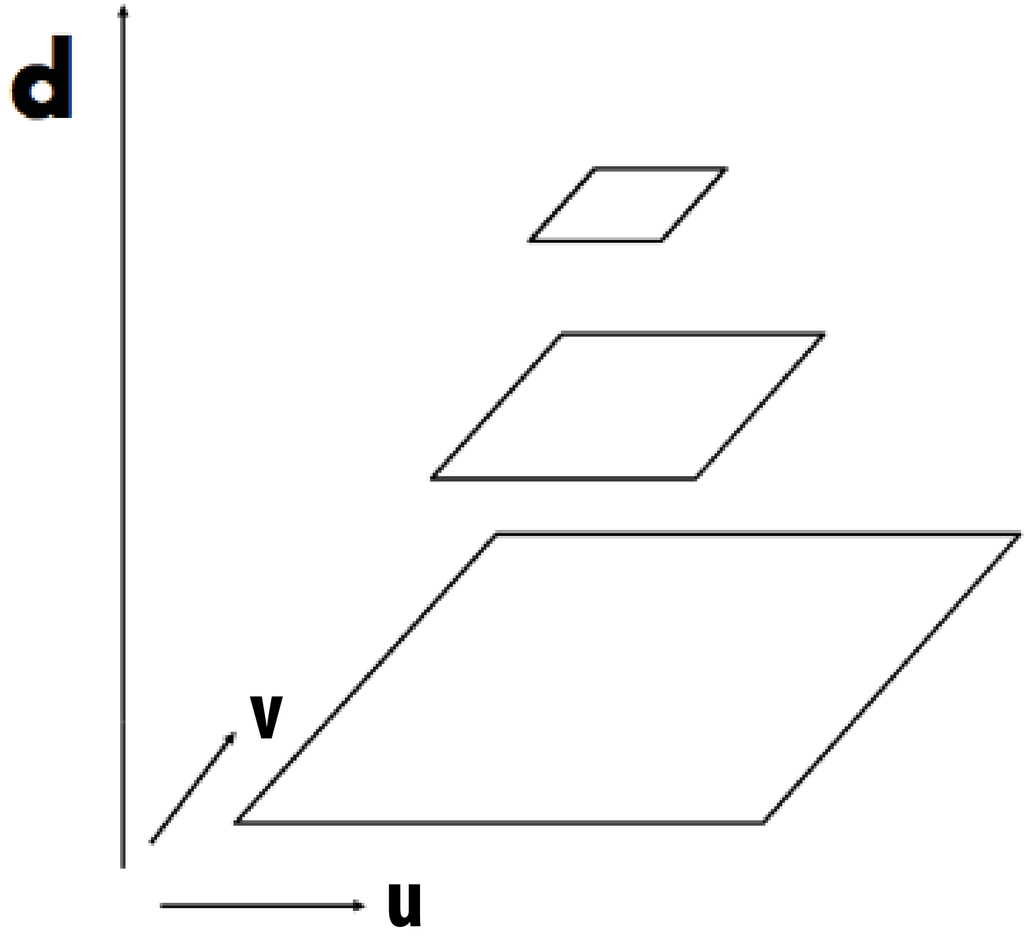
Texels at higher levels store integral of the texture function over a region of texture space (downsampled images)

Texels at higher levels represent low-pass filtered version of original texture signal

Mipmap (L. Williams 83)



Williams' original proposed
mip-map layout



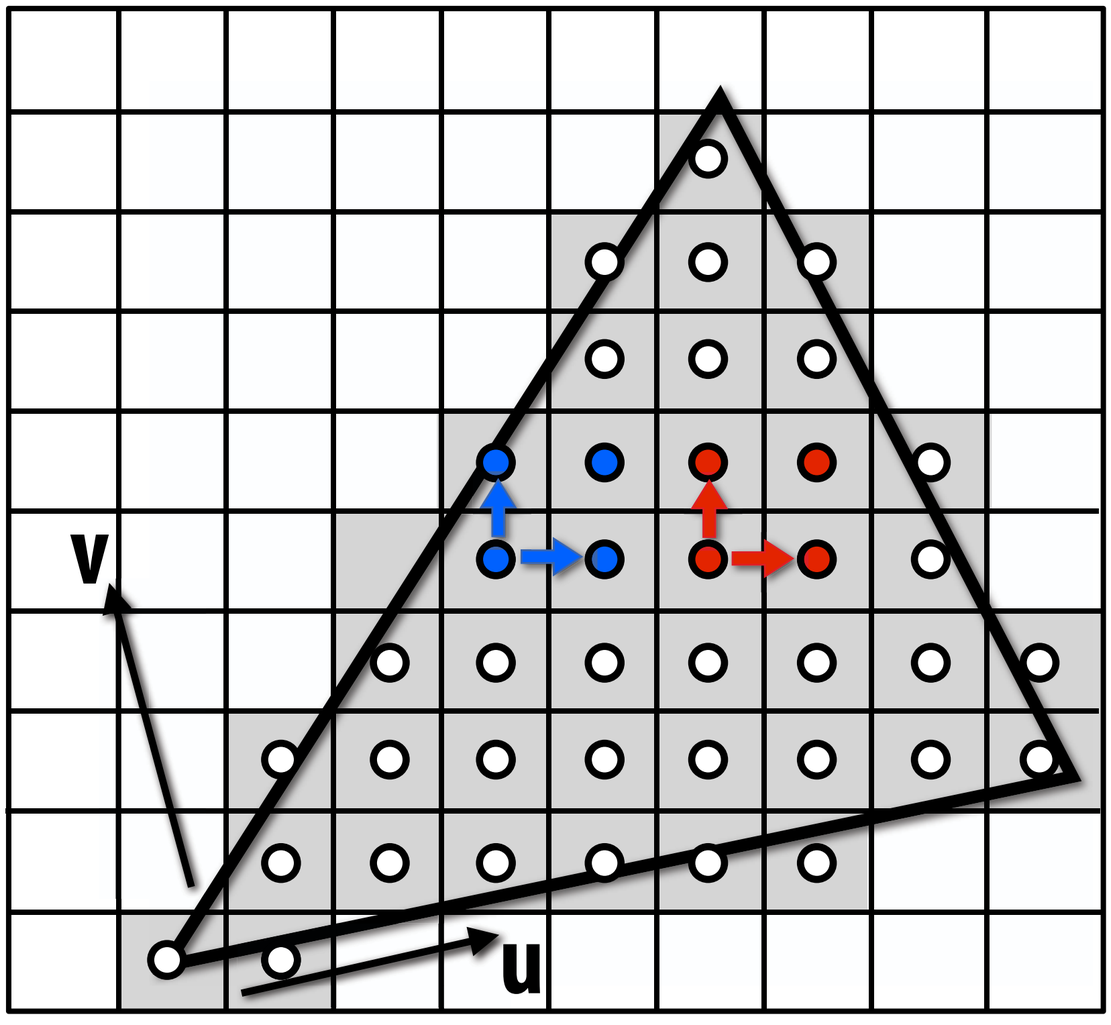
"Mip hierarchy"
level = *d*

What is the storage overhead of a mipmap?

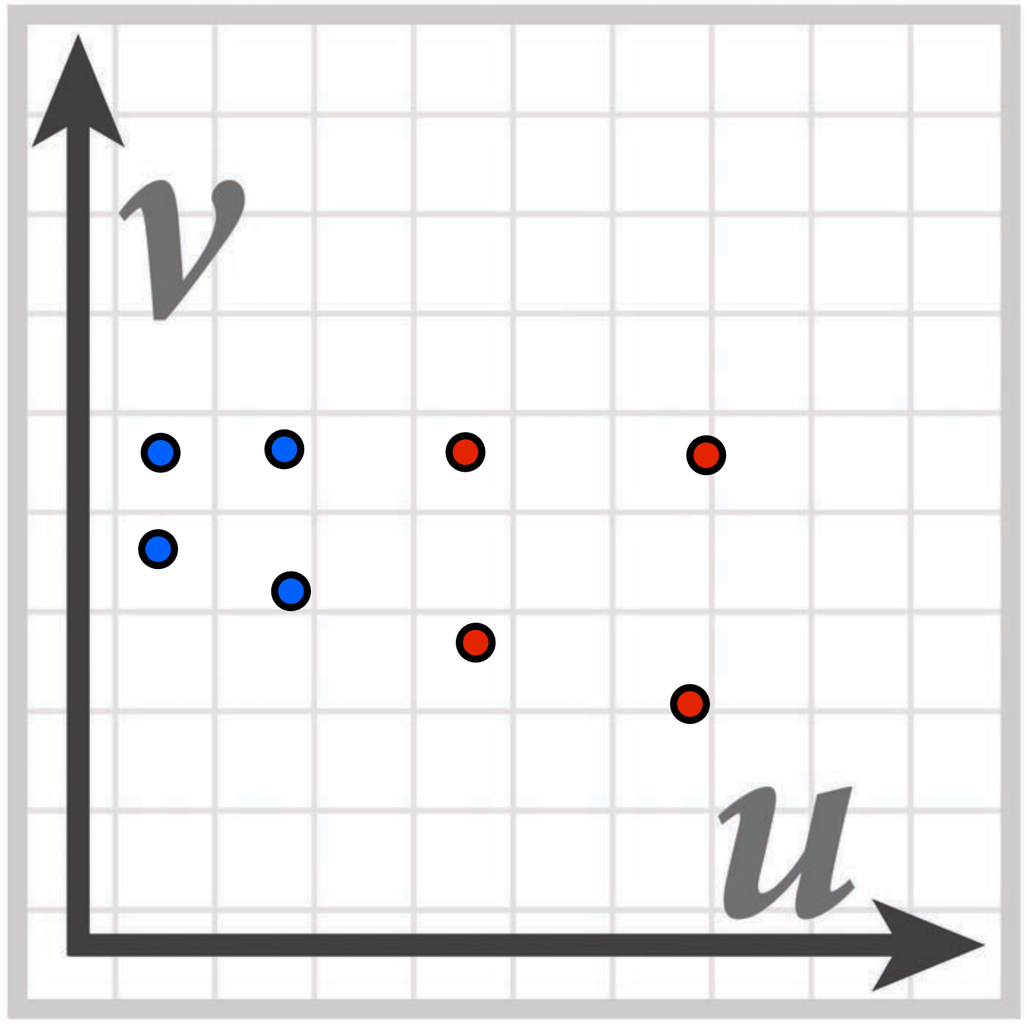
Slide credit: Akeley and Hanrahan

Computing Mip Map Level

Compute differences between texture coordinate values of neighboring screen samples



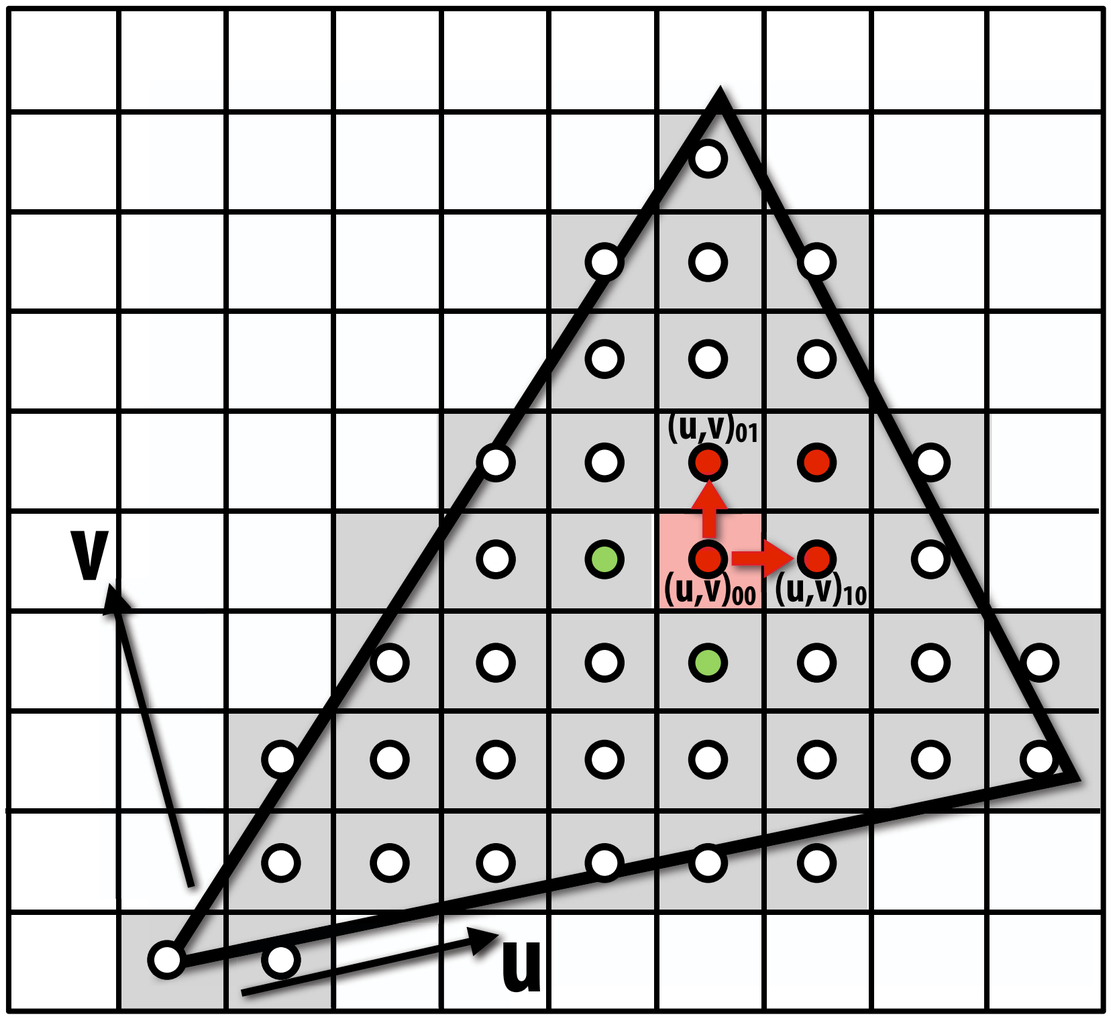
Screen space

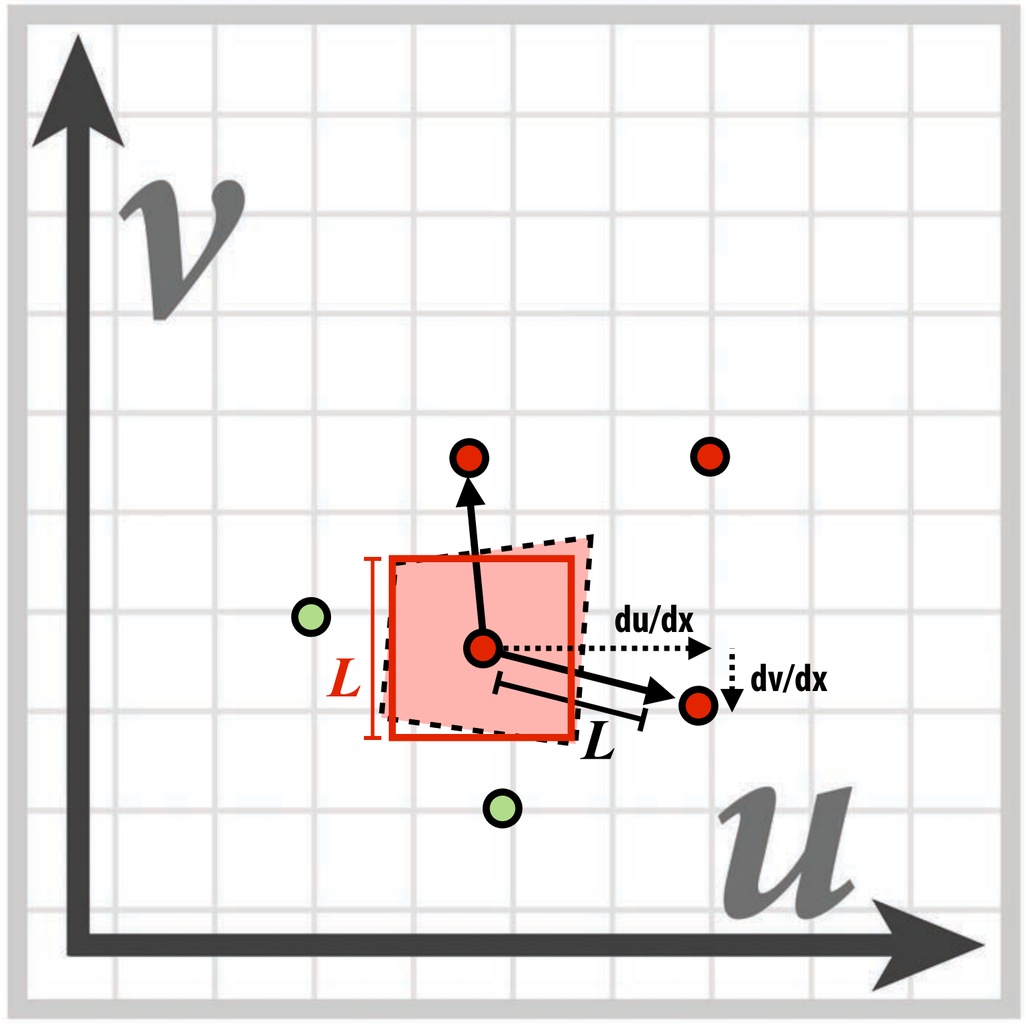


Texture space

Computing Mip Map Level

Compute differences between texture coordinate values of neighboring fragments





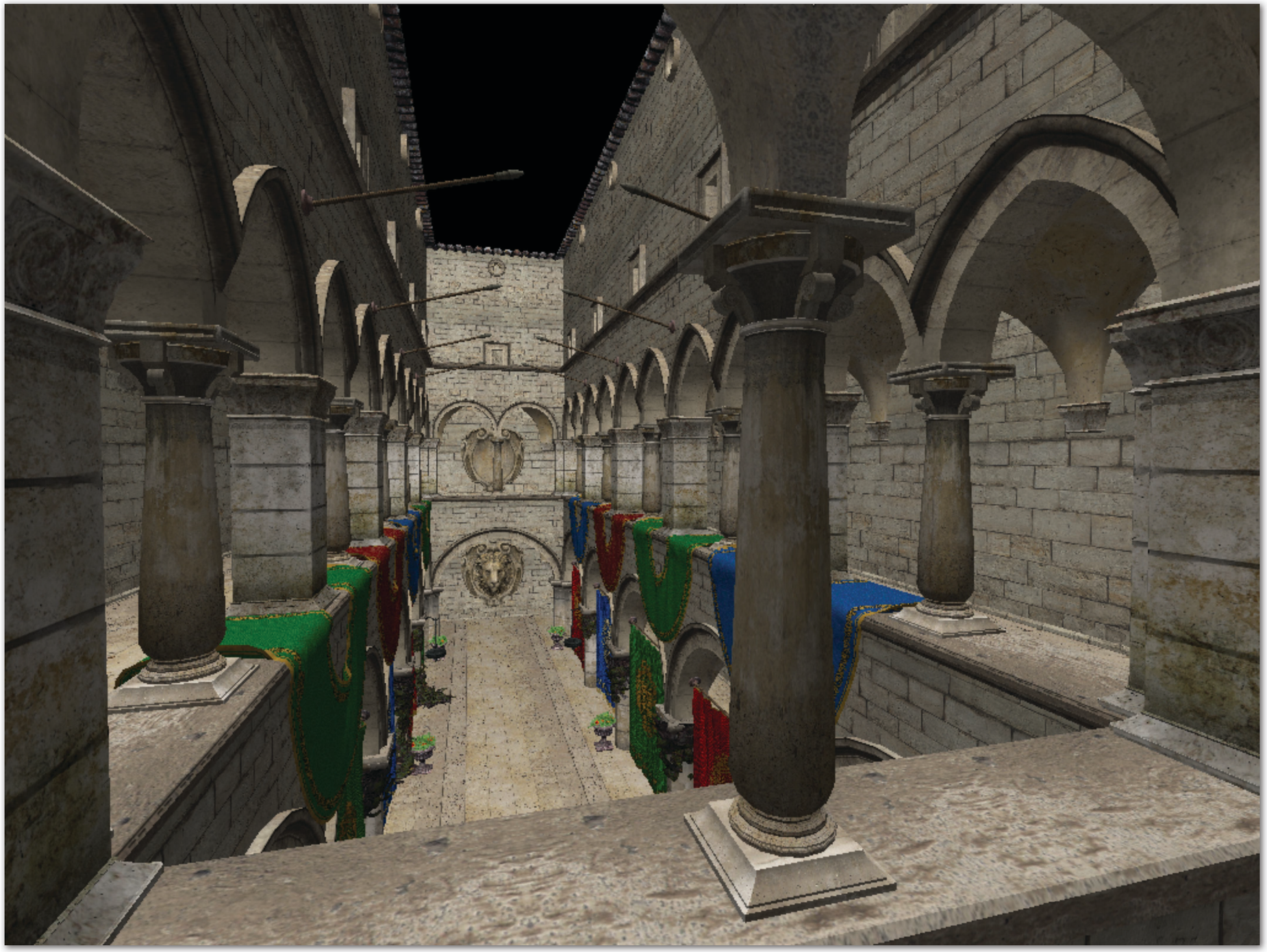
$du/dx = u_{10}-u_{00}$ $\quad dv/dx = v_{10}-v_{00}$

$du/dy = u_{01}-u_{00}$ $\quad dv/dy = v_{01}-v_{00}$

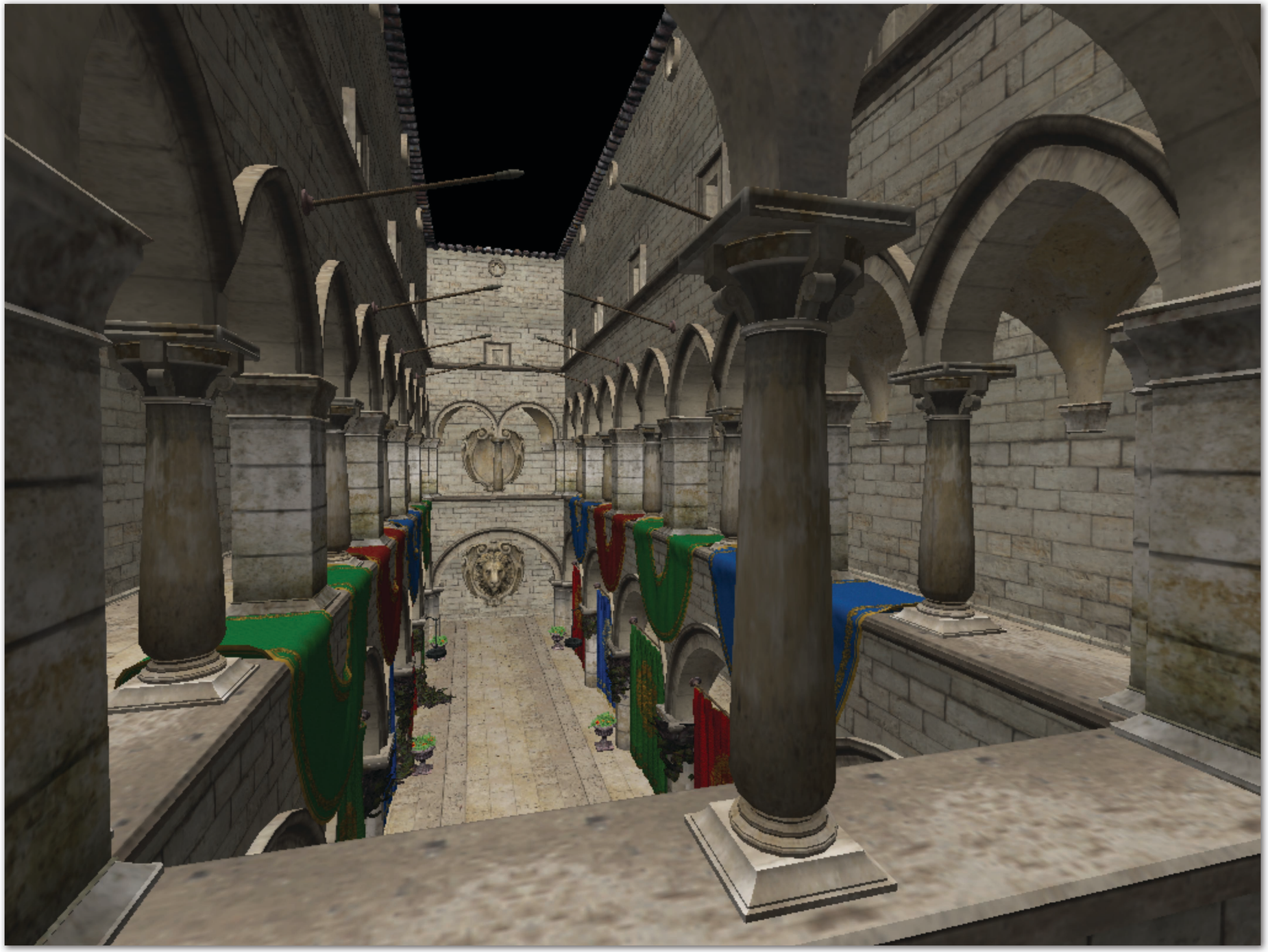
$$L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2+\left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2+\left(\frac{dv}{dy}\right)^2}\right)$$

$$mip\text{-}map\ d = \log_2 L$$

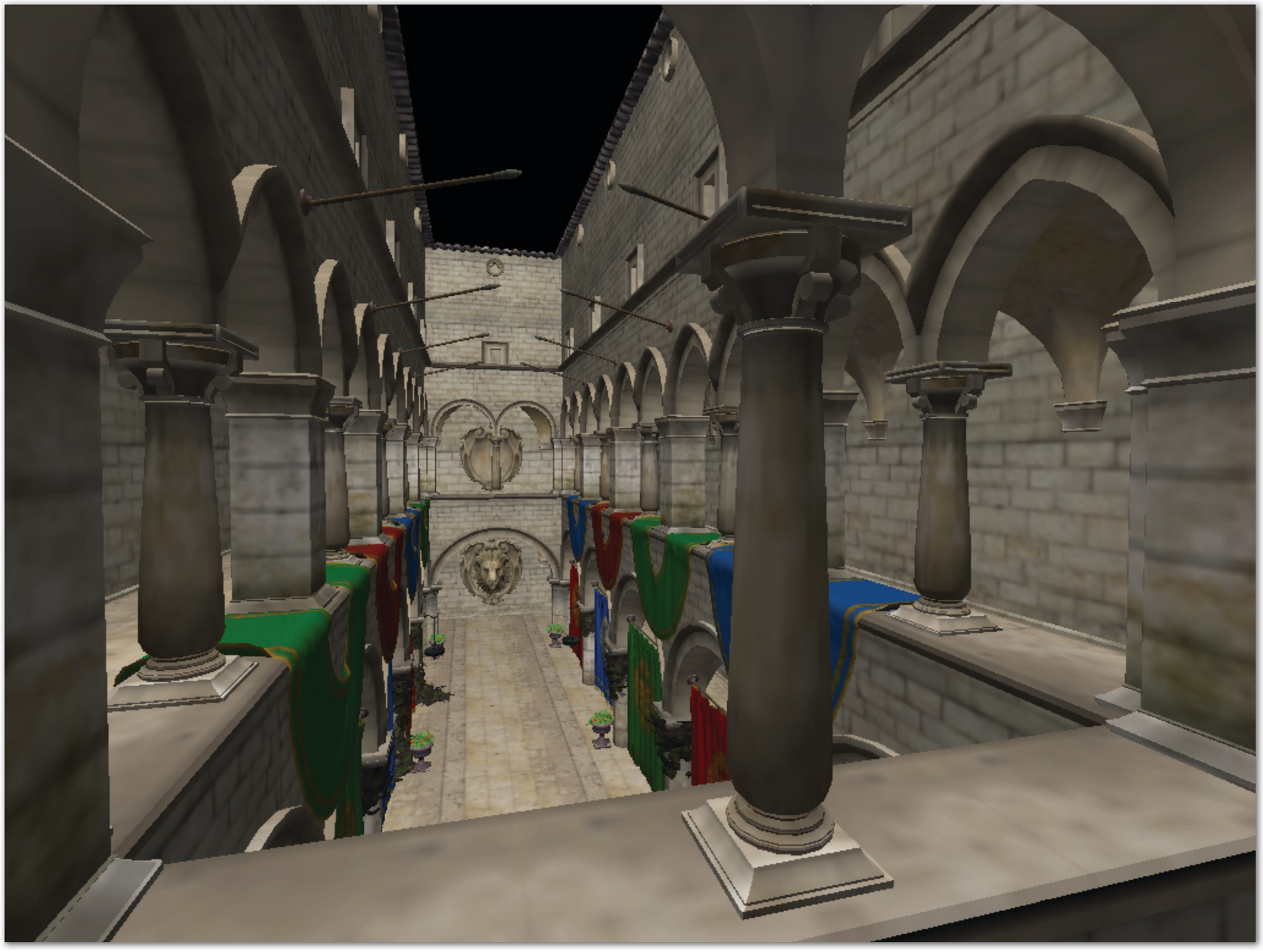
Sponza (bilinear resampling at level 0)



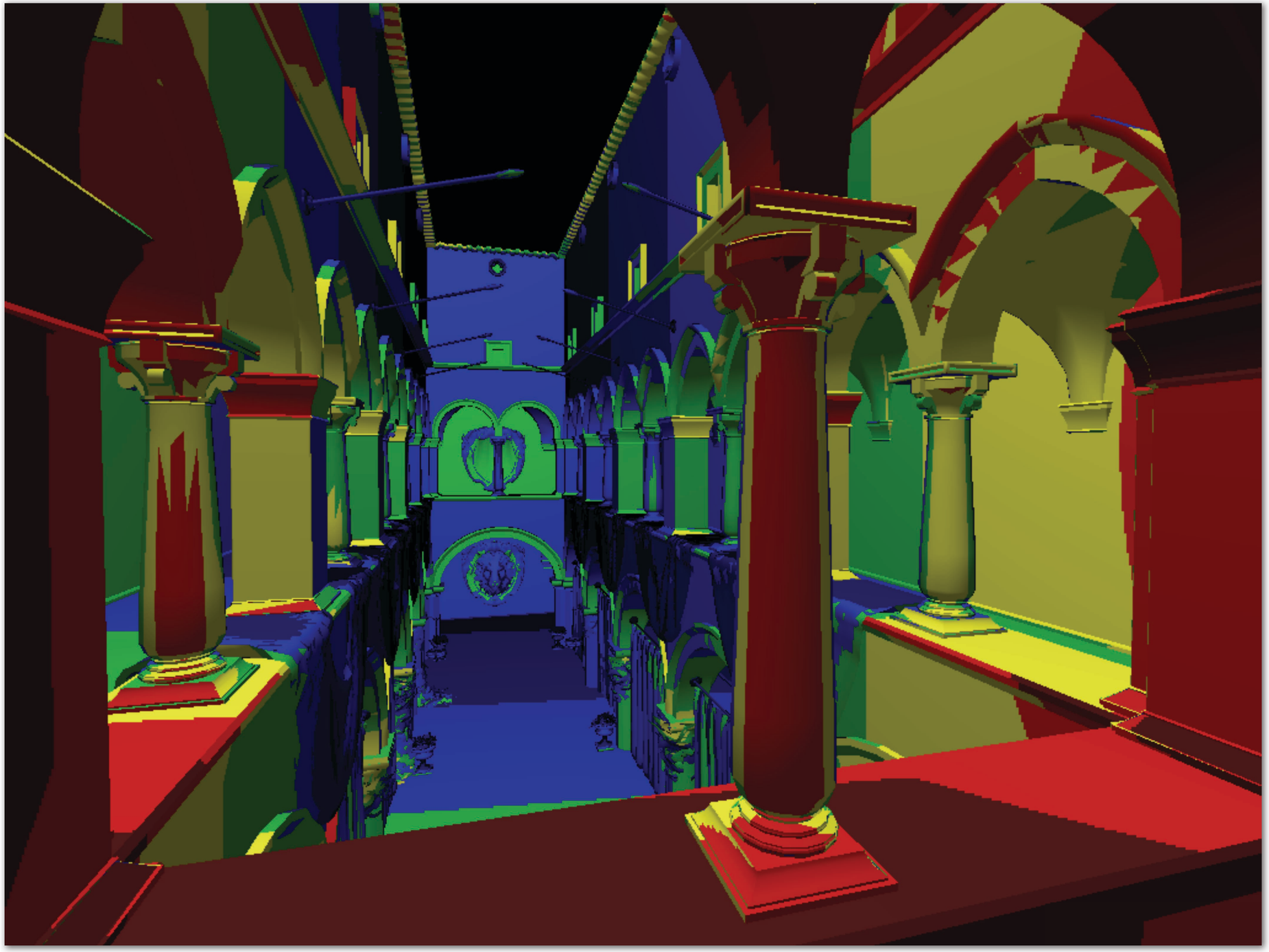
Sponza (bilinear resampling at level 2)



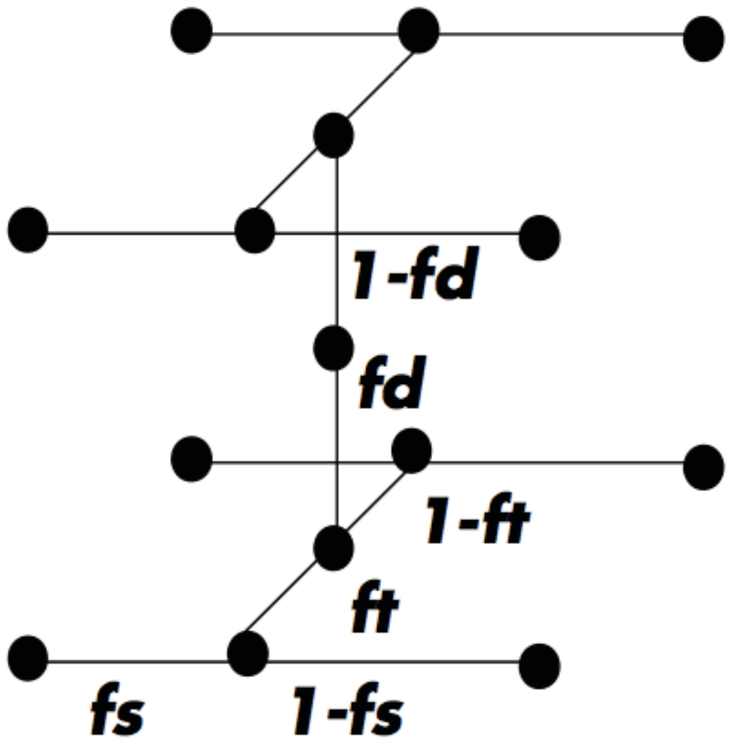
Sponza (bilinear resampling at level 4)



Visualization of mip-map level
(bilinear filtering only: *d* clamped to nearest level)



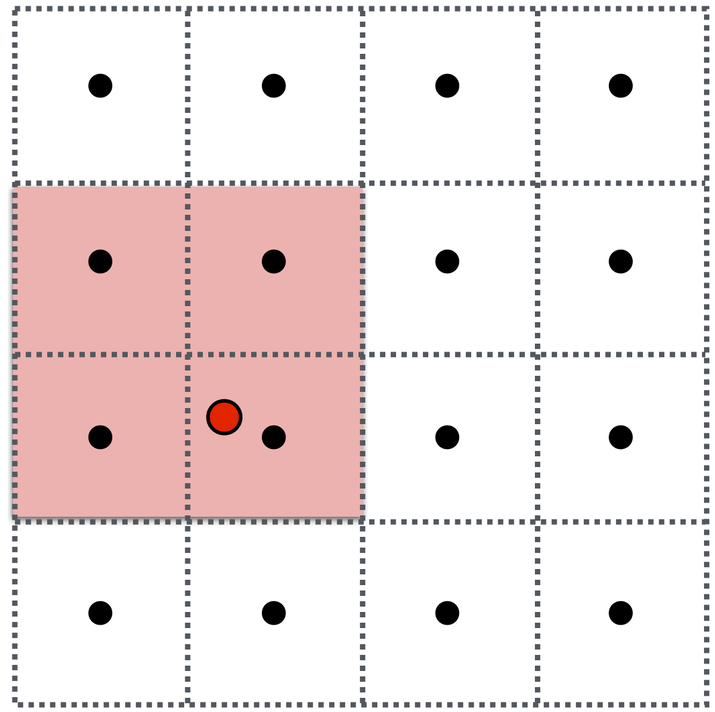
“Tri-linear” filtering



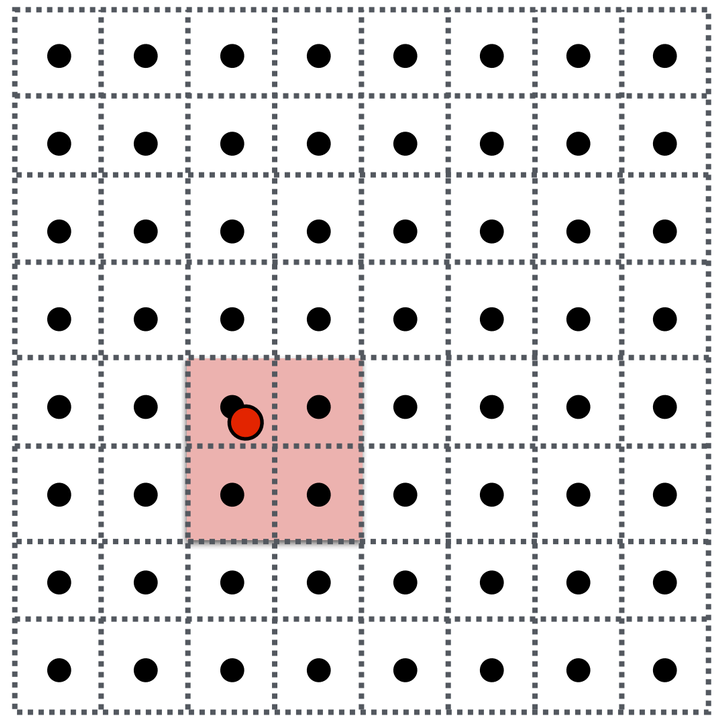
$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

Bilinear resampling:
- four texel reads
- 3 lerps (3 mul + 6 add)

Trilinear resampling:
- eight texel reads
- 7 lerps (7 mul + 14 add)



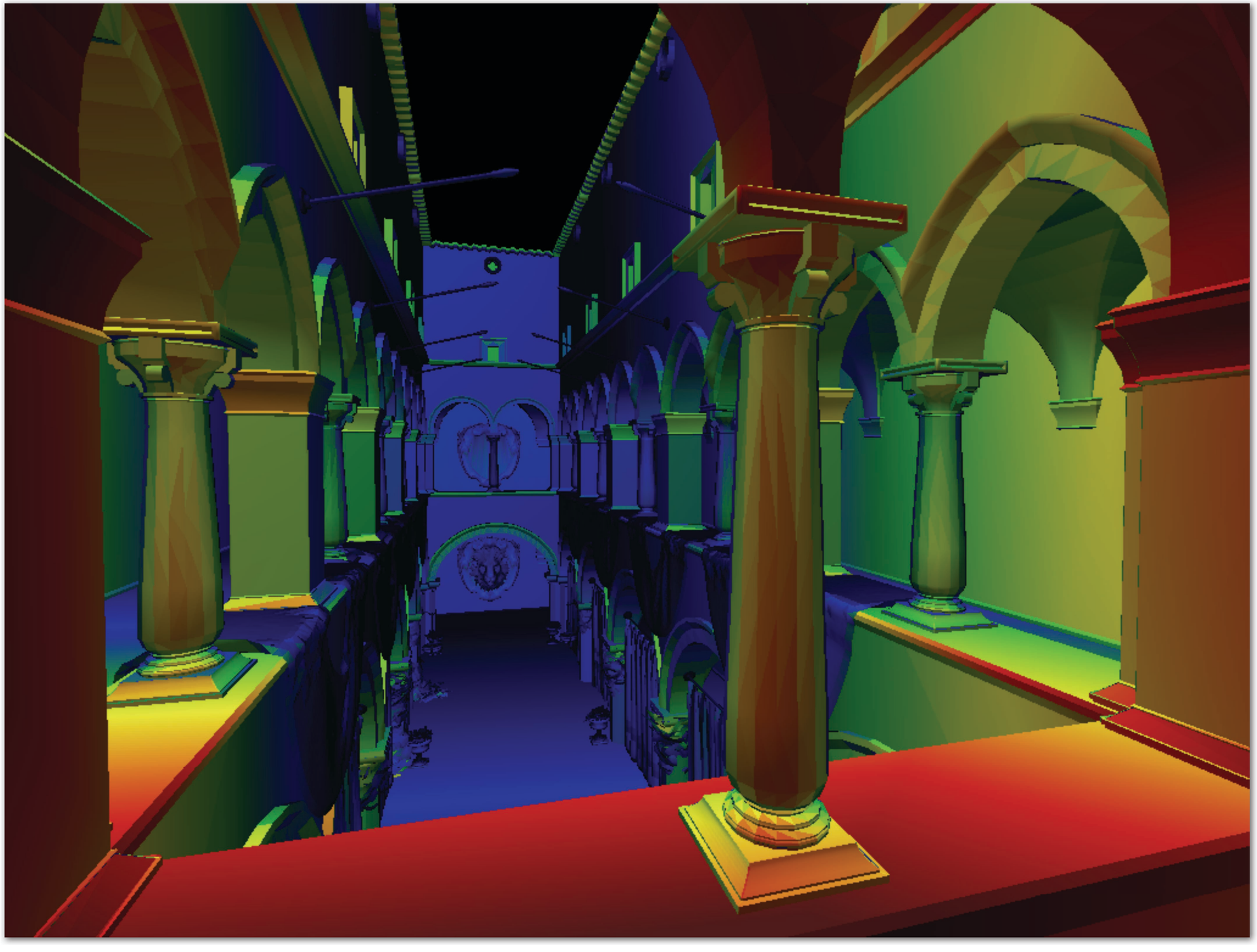
mip-map texels: level d+1



mip-map texels: level d

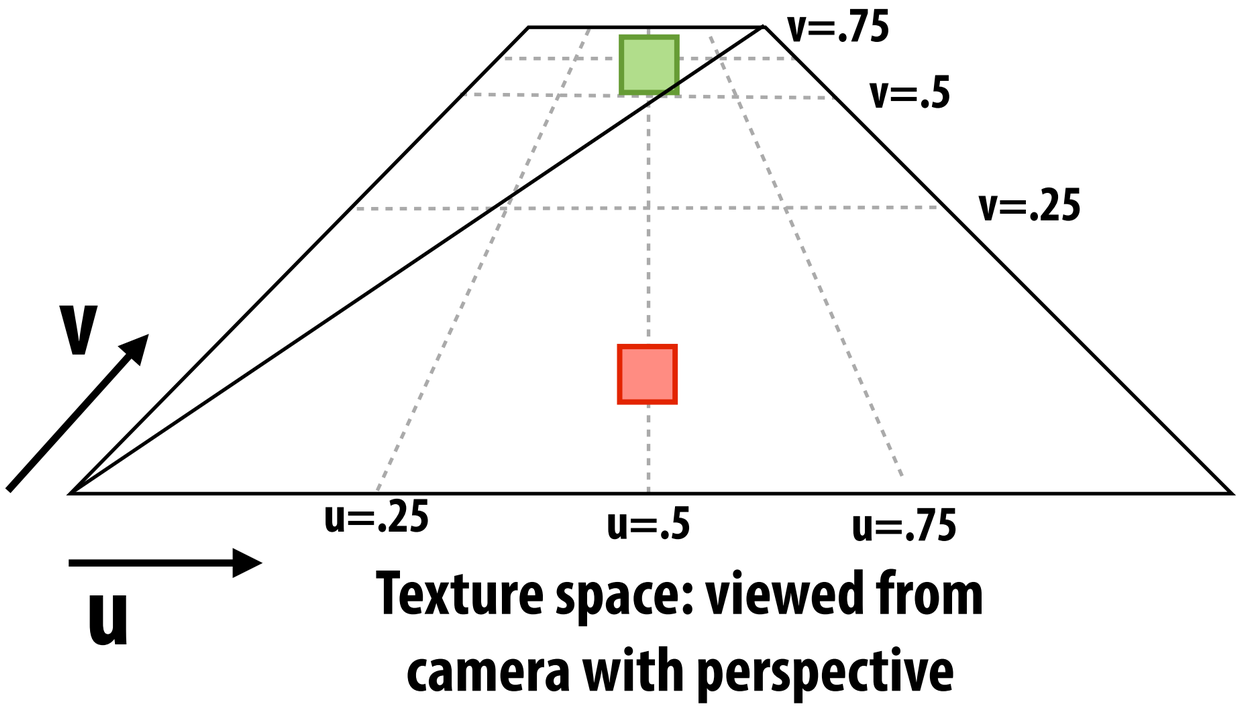
Figure credit: Akeley and Hanrahan

Visualization of mip-map level
(trilinear filtering: visualization of continuous *d*)

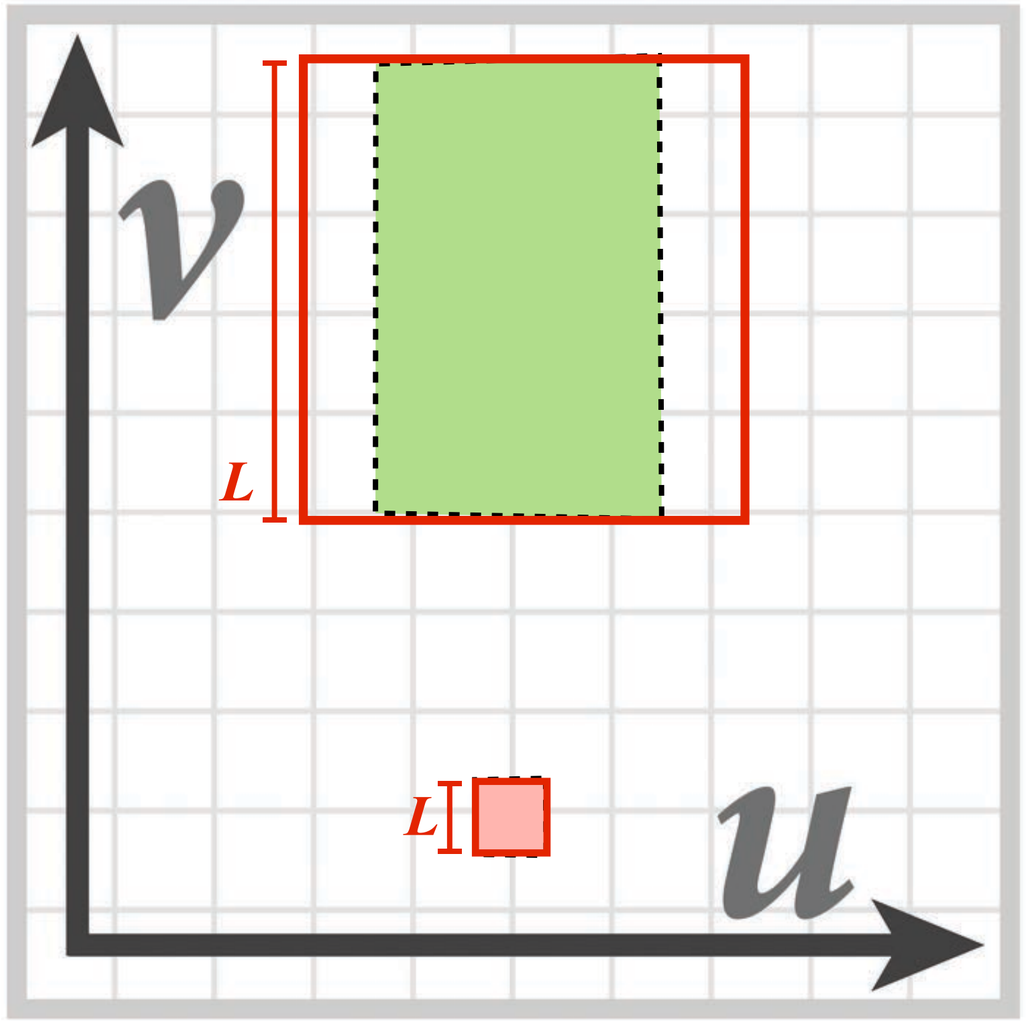


Pixel area may not map to isotropic region in texture

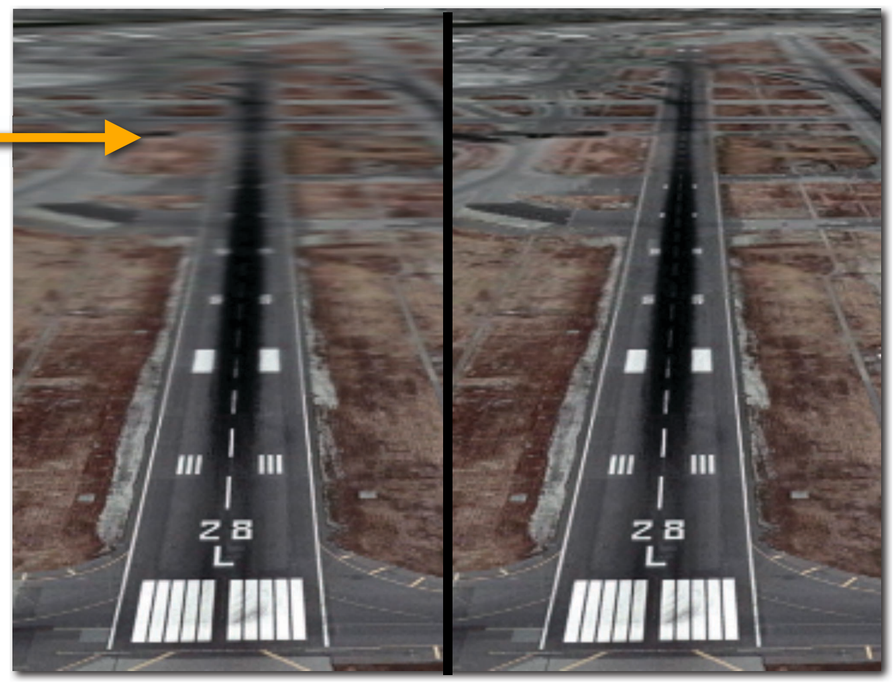
Proper filtering requires anisotropic filter footprint



Texture space: viewed from camera with perspective



Overblurring in *u* direction



Trilinear (Isotropic) Filtering

Anisotropic Filtering

(Modern solution: Combine multiple mip map samples)

Summary: texture filtering using the mip map

- **Small storage overhead (33%)**
  - Mipmap is 4/3 the size of original texture image

- **For each isotropically-filtered sampling operation**
  - Constant filtering cost (independent of mip map level)
  - Constant number of texels accessed (independent of mip map level)

- **Combat aliasing with *prefiltering*, rather than supersampling**
  - Recall: we used supersampling to address aliasing problem when sampling coverage

- **Bilinear/trilinear filtering is isotropic and thus will "overblur" to avoid aliasing**
  - Anisotropic texture filtering provides higher image quality at higher compute and memory bandwidth cost (in practice: multiple mip map samples)

"Real" Texture Sampling

1. Compute u and v from screen sample x,y (via evaluation of attribute equations)
2. Compute du/dx, du/dy, dv/dx, dv/dy differentials from screen-adjacent samples.
3. Compute mip map level *d*
4. Convert normalized [0,1] texture coordinate (u,v) to texture coordinates U,V in [W,H]
5. Compute required texels in window of filter
6. Load required texels (need eight texels for trilinear)
7. Perform tri-linear interpolation according to (U, V, d)

Takeaway: a texture sampling operation is not just an image pixel lookup! It involves a significant amount of math.

For this reason, modern GPUs have dedicated fixed-function hardware support for performing texture sampling operations.

Texturing summary

- Texture coordinates: define mapping between points on triangle's surface (object coordinate space) to points in texture coordinate space

- Texture mapping is a sampling operation and is prone to aliasing
  - Solution: prefilter texture map to eliminate high frequencies in texture signal
  - Mip-map: precompute and store multiple multiple resampled versions of the texture image (each with different amounts of low-pass filtering)
  - During rendering: dynamically select how much low-pass filtering is required based on distance between neighboring screen samples in texture space
    - Goal is to retain as much high-frequency content (detail) in the texture as possible, while avoiding aliasing