Lecture 4:

Perspective Projection and Texture Mapping

Computer Graphics
CMU 15-462/15-662, Fall 2015

Review from last time: screen transform *

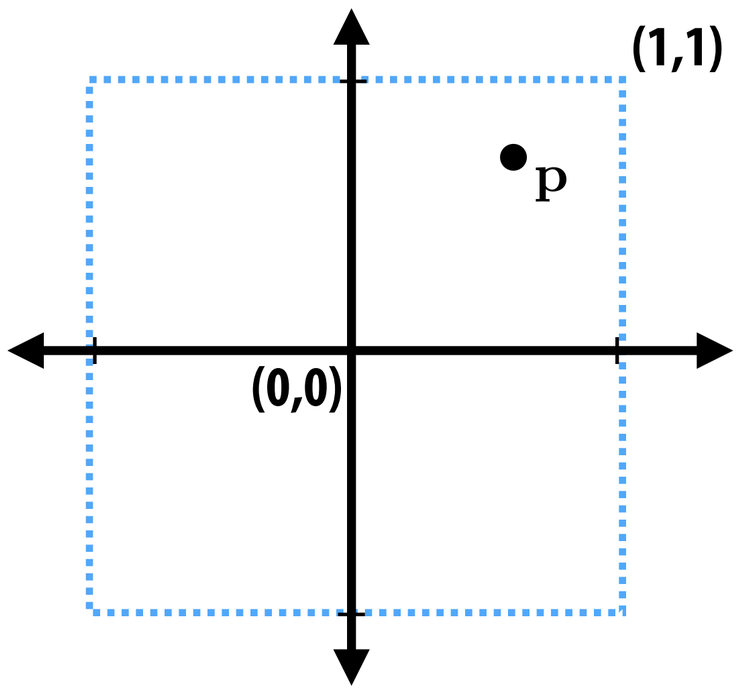
Convert points in normalized coordinate space to screen pixel coordinates

Example:

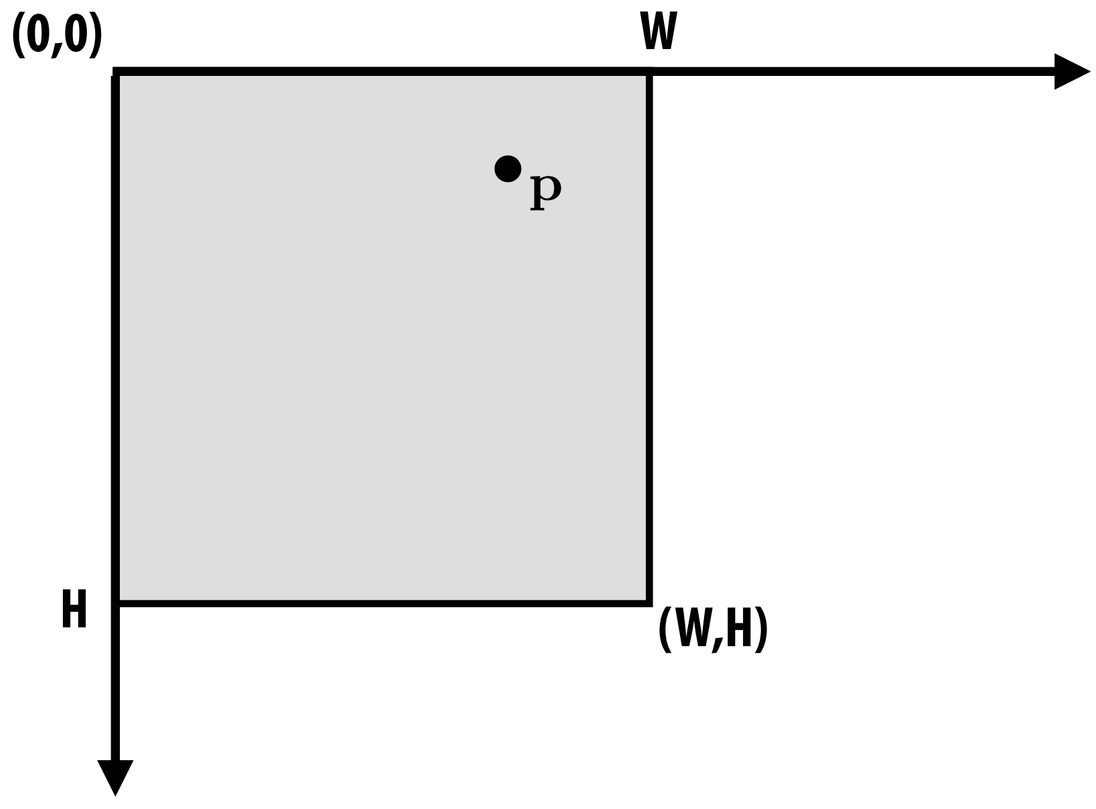
All points within (-1,1) to (1,1) region are on screen

(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)

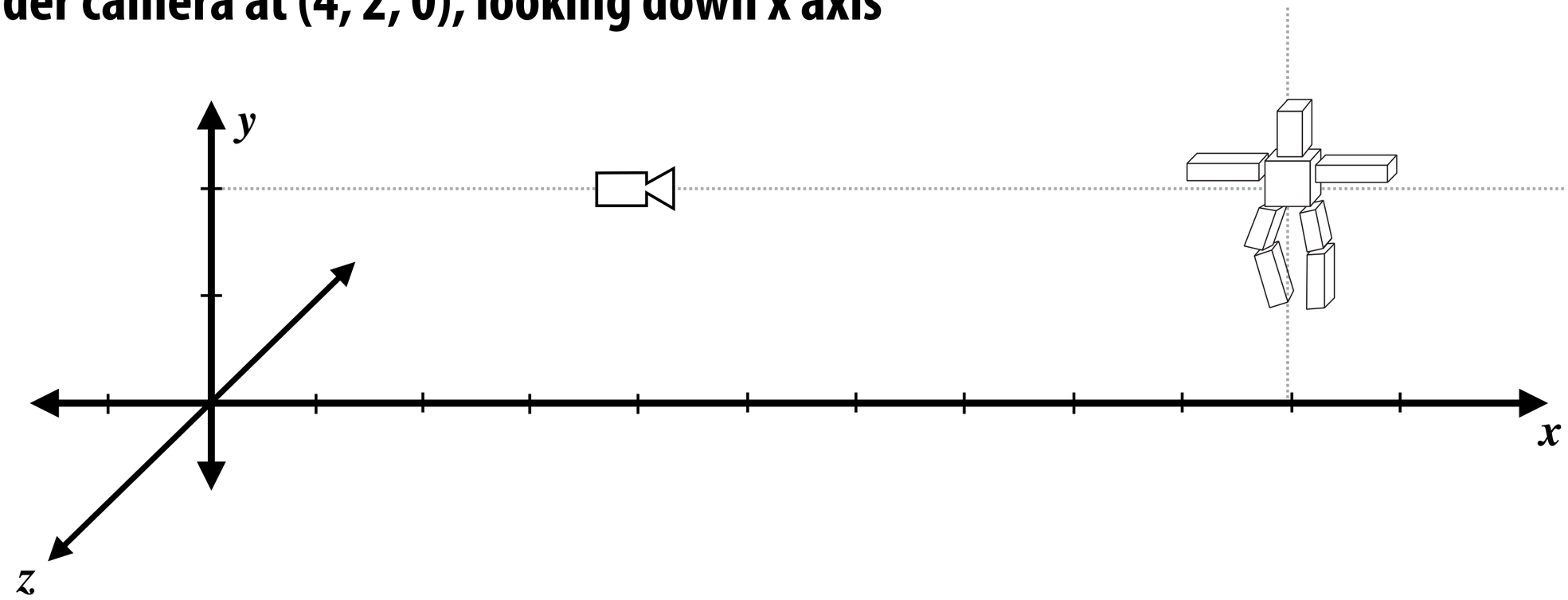
Step 3: scale by (W/2,H/2)

*** This slide adopts the convention that top-left of screen is (0,0) to match SVG convention in Assignment 1. Many 3D graphics systems like OpenGL place (0,0) in bottom-left. In this case what would the transform be?**

Review from last time: simple camera transform

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

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



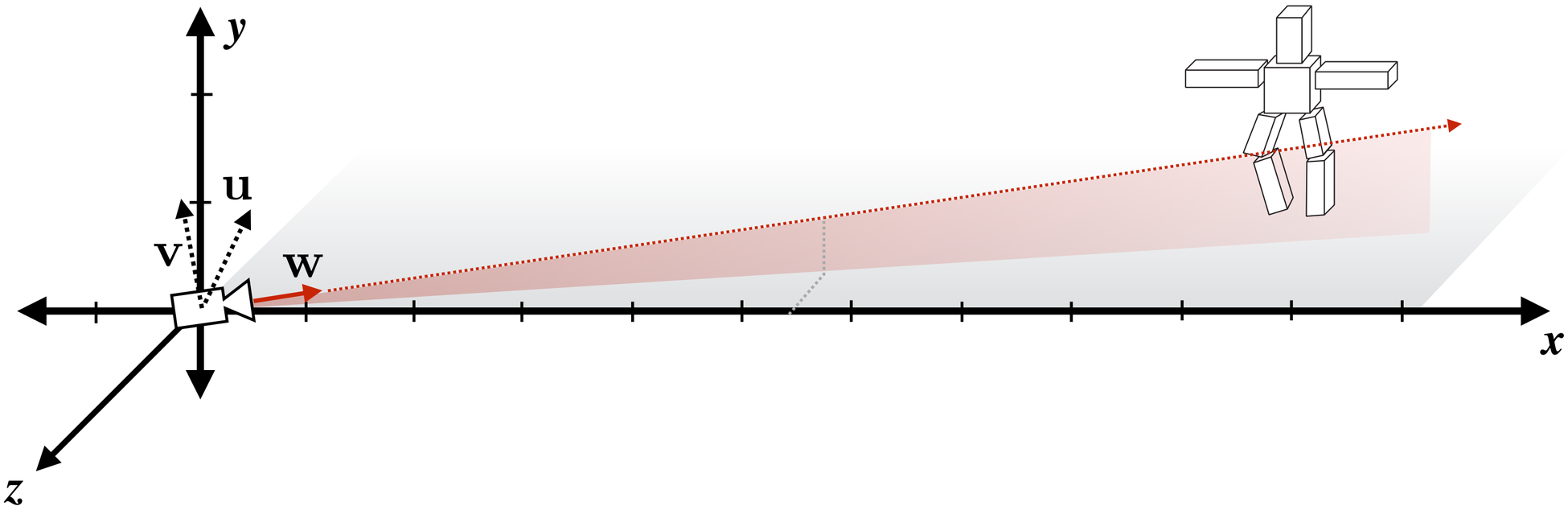
What transform places in the object in a coordinate space where the camera is at the origin and the camera is looking directly down the -z axis?

- **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 with arbitrary orientation

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 -$\mathbf{w}$

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

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

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

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

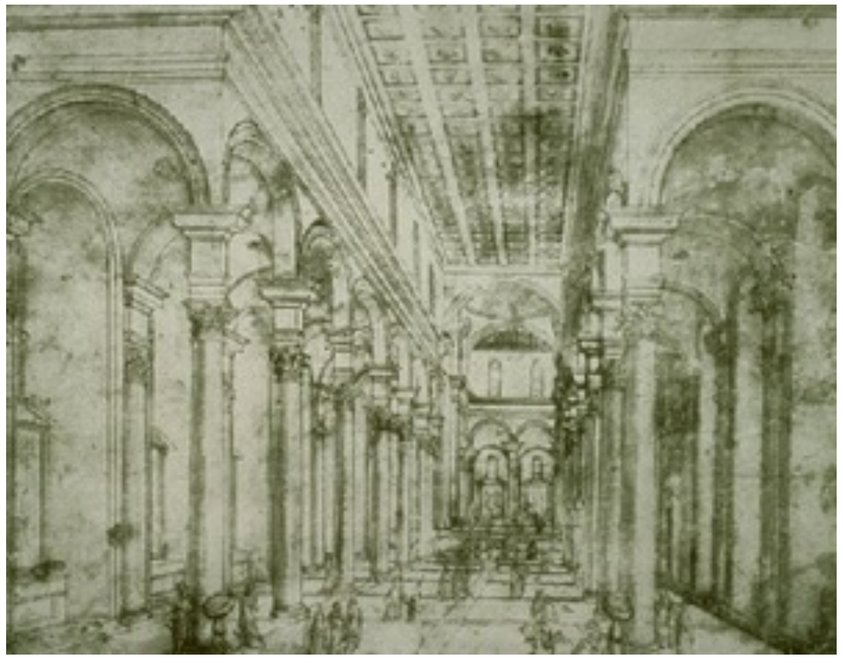
Perspective projection

Early painting: incorrect perspective

8-9th century painting

Geometrically correct perspective in art

Ambrogio Lorenzetti
Annunciation, 1344



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

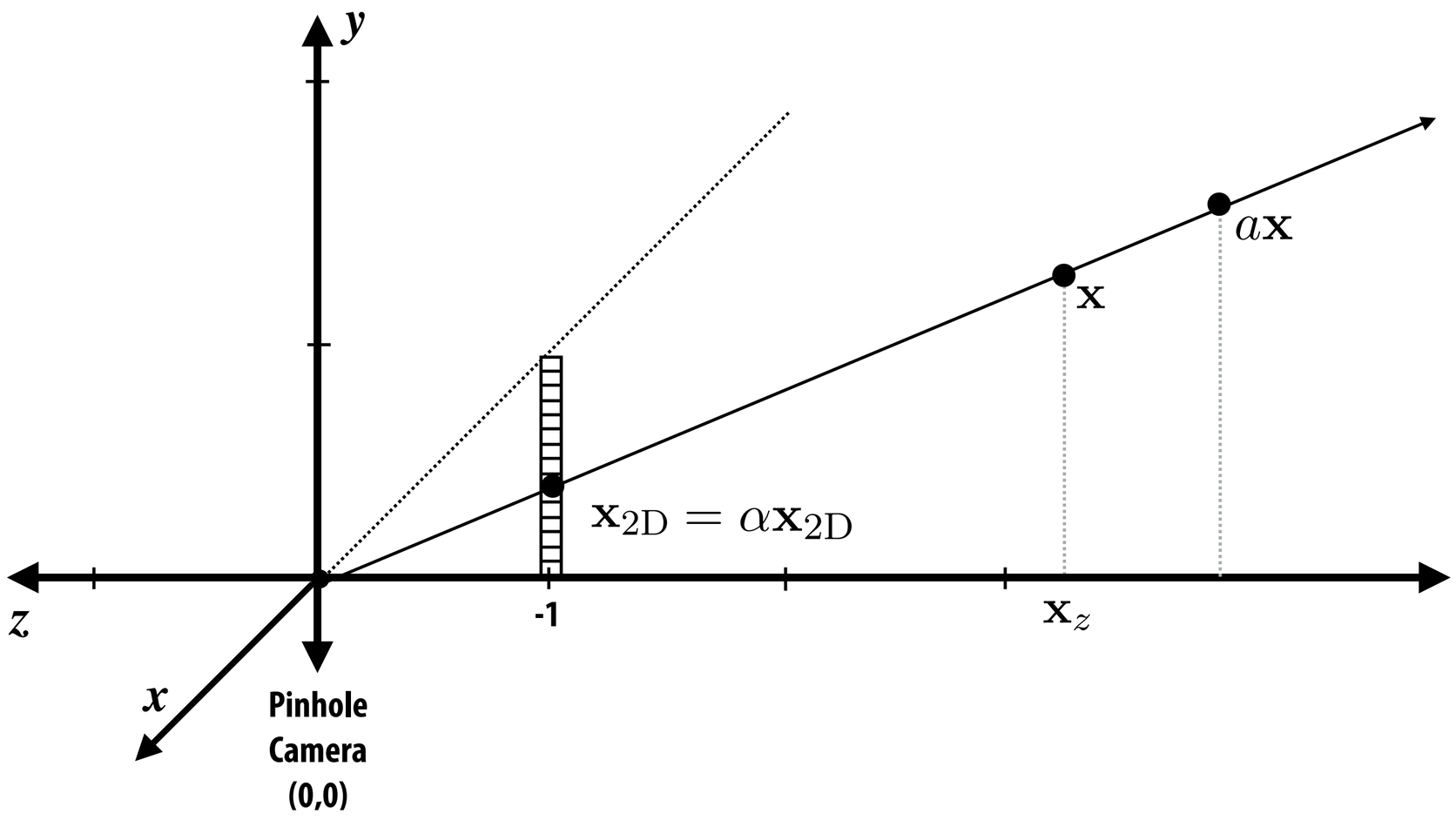
Masaccio – The Tribute Money c.1426-27
Fresco, The Brancacci Chapel, Florence

Later… rejection of proper perspective projection

Basic perspective projection

Input point in 3D-H: $\mathbf{x} = \begin{bmatrix} \mathbf{x}_x & \mathbf{x}_y & \mathbf{x}_z & 1 \end{bmatrix}^T$

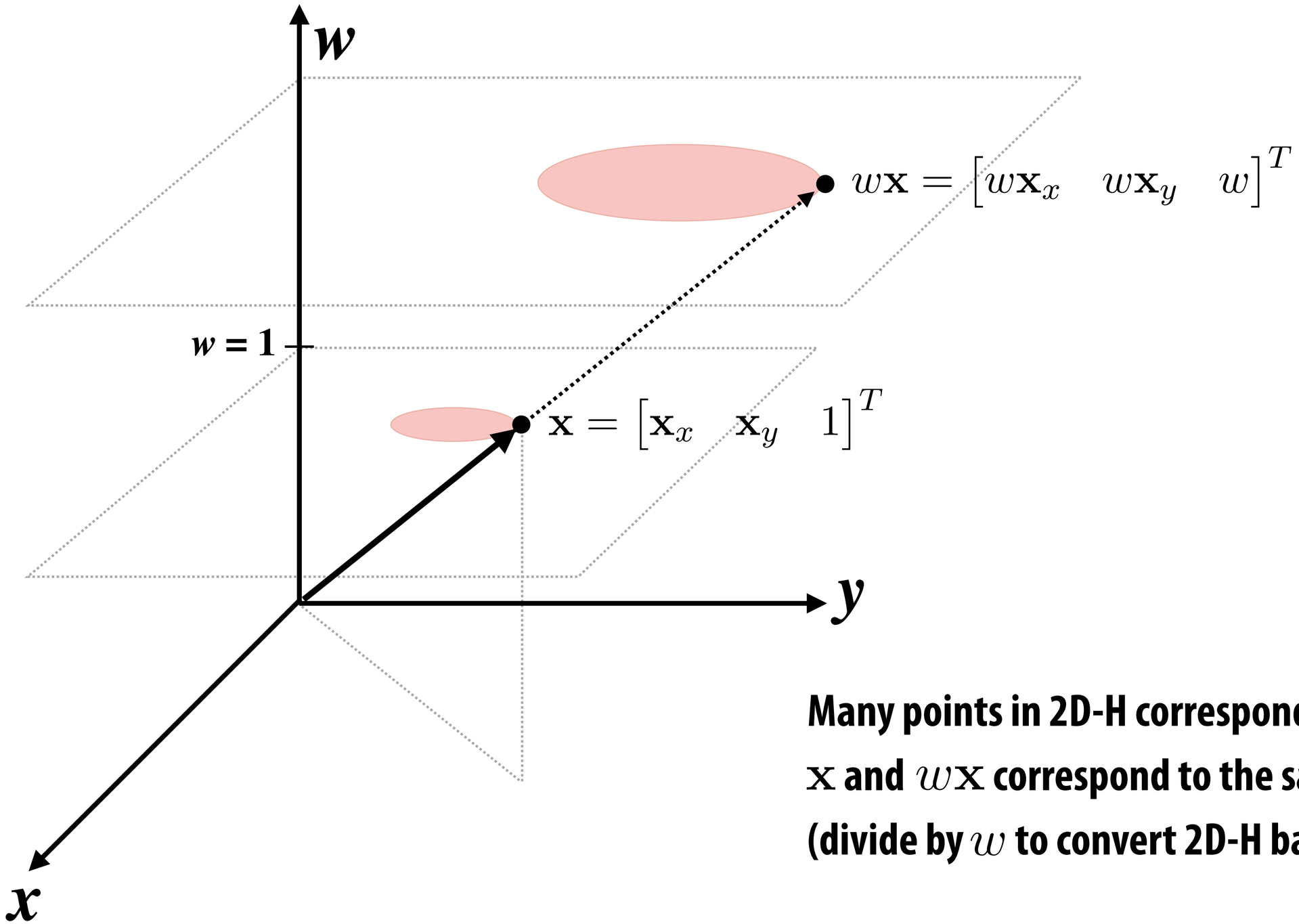
Perspective projected result (2D point): $\mathbf{x}_{2\mathrm{D}} = \begin{bmatrix} \mathbf{x}_x / -\mathbf{x}_z & \mathbf{x}_y / -\mathbf{x}_z \end{bmatrix}^T$



Assumption:

Pinhole camera at (0,0) looking down $-z$

Review: homogeneous coordinates



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

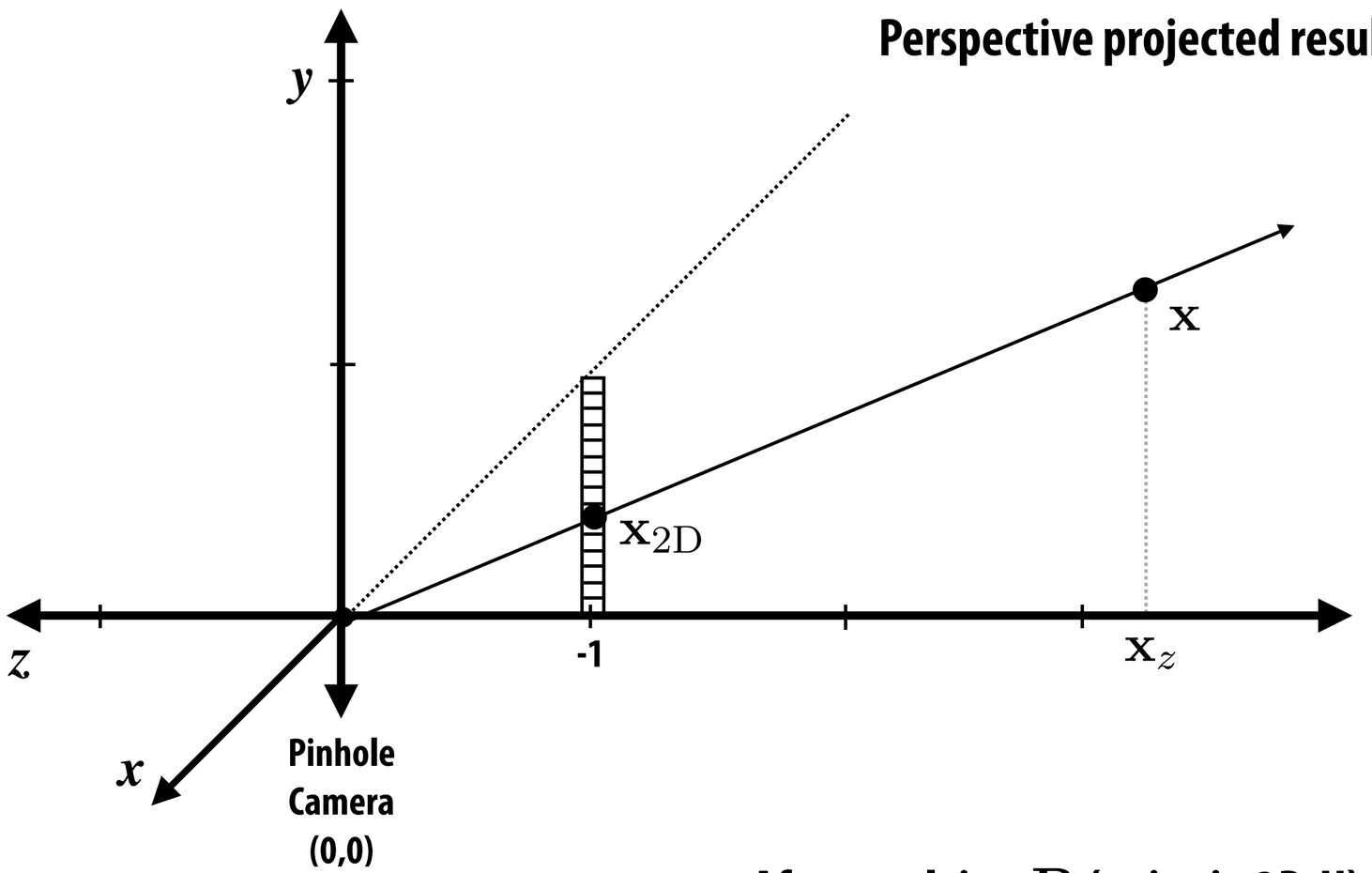
$\mathbf{x}$ and $w\mathbf{x}$ correspond to the same 2D point

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

Basic perspective projection

Input point in 3D-H: $\mathbf{x} = \begin{bmatrix}\mathbf{x}_x & \mathbf{x}_y & \mathbf{x}_z & 1\end{bmatrix}^T$

Perspective projected result (2D point): $\mathbf{x}_{2\mathrm{D}} = \begin{bmatrix}\mathbf{x}_x/-\mathbf{x}_z & \mathbf{x}_y/-\mathbf{x}_z\end{bmatrix}^T$



$$\mathbf{P} = \begin{bmatrix}1 & 0 & 0 & 0\\ 0 & 1 & 0 & 0\\ 0 & 0 & 1 & 0\\ 0 & 0 & -1 & 0\end{bmatrix}$$

After applying $\mathbf{P}$ (point in 3D-H): $\mathbf{P}\mathbf{x} = \begin{bmatrix}\mathbf{x}_x & \mathbf{x}_y & \mathbf{x}_z & -\mathbf{x}_z\end{bmatrix}^T$

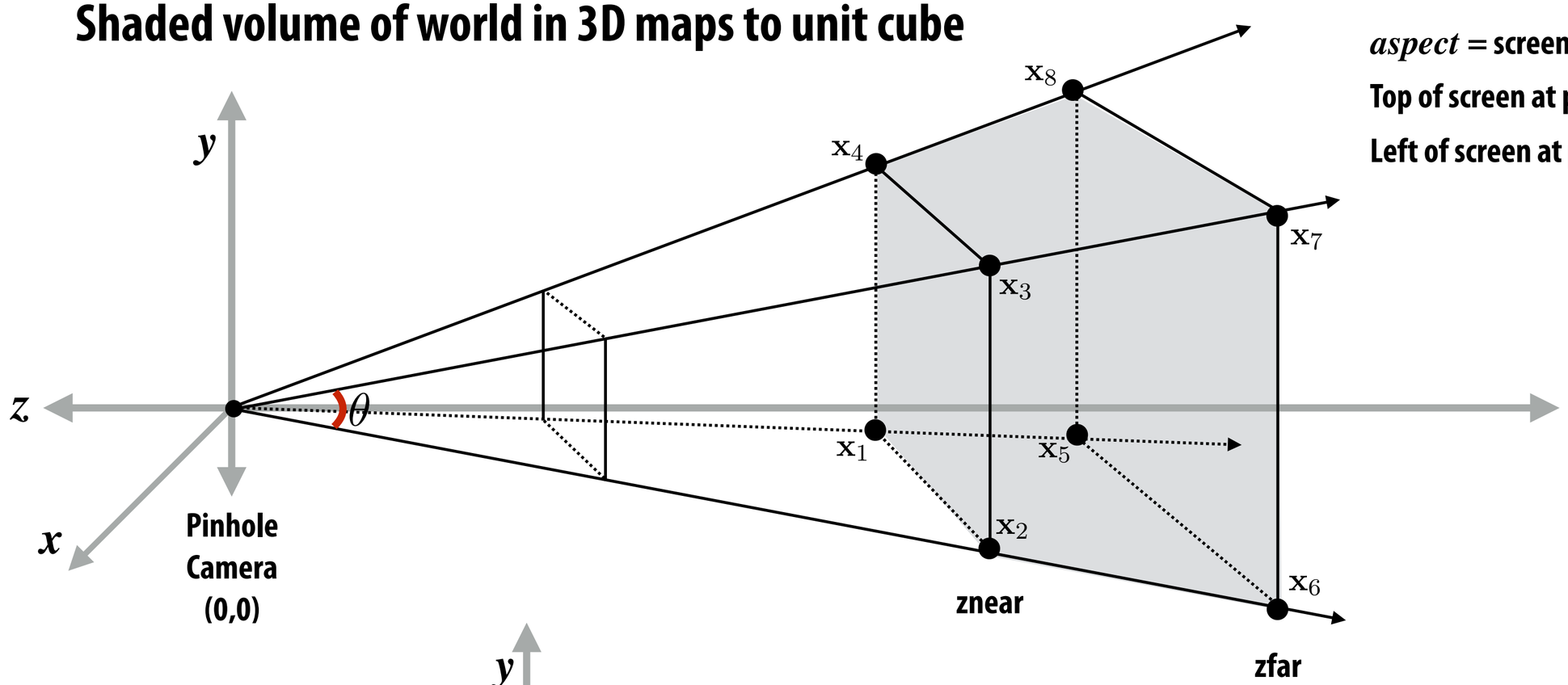
Point in 2D-H (drop z coord): $\mathbf{x}_{2\mathrm{D\text{-}H}} = \begin{bmatrix}\mathbf{x}_x & \mathbf{x}_y & -\mathbf{x}_z\end{bmatrix}^T$

Point in 2D (after homogeneous divide): $\mathbf{x}_{2\mathrm{D}} = \begin{bmatrix}\mathbf{x}_x/-\mathbf{x}_z & \mathbf{x}_y/-\mathbf{x}_z\end{bmatrix}^T$

Assumption:
Pinhole camera at (0,0) looking down $-z$

Projecting view frustum to unit cube

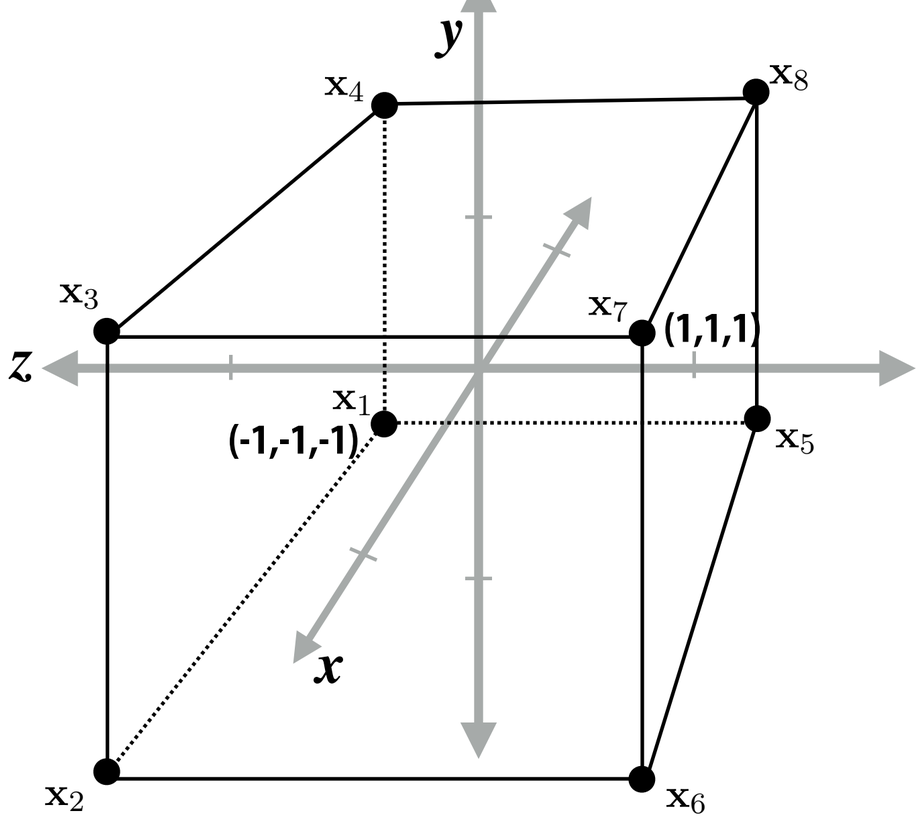
Shaded volume of world in 3D maps to unit cube



aspect = screen width / screen height

Top of screen at projection plane (z=-1): $\tan(\theta/2)$

Left of screen at projection plane: $aspect \times \tan(\theta/2)$



$$\mathbf{P} = \begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zfar+znear}{znear-zfar} & \frac{2\times zfar\times znear}{znear-zfar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

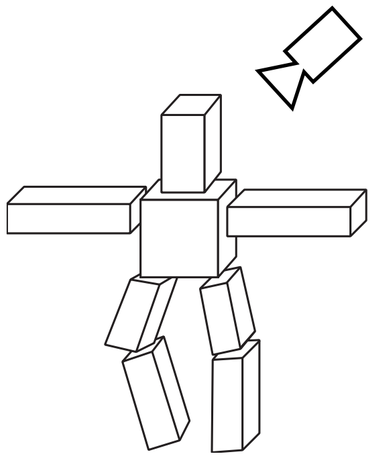
where: $f = \cot(\theta/2)$

Note treatment of z by $\mathbf{P}$:

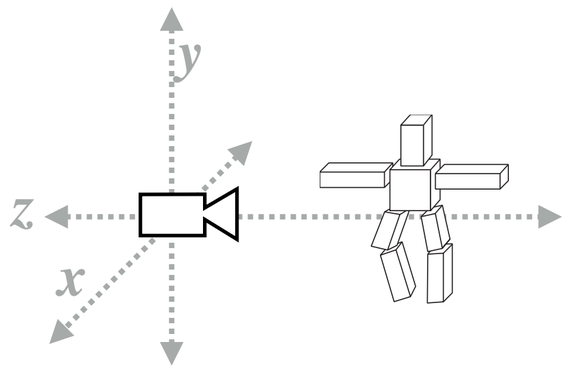
z = *-znear* maps to -z

z = *-zfar* maps to z

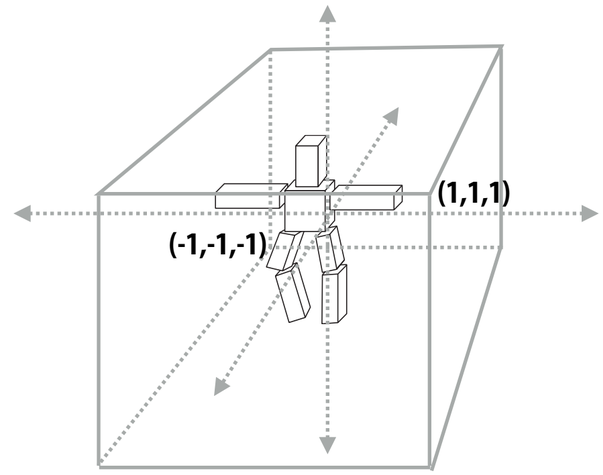
Transformations recap



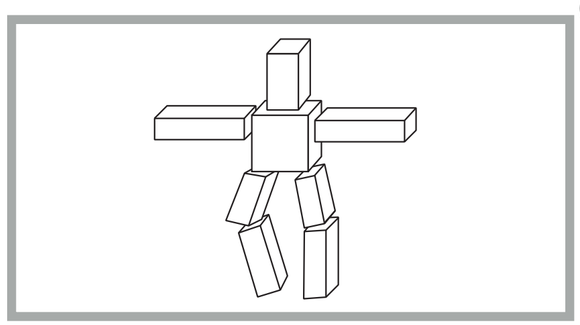
Modeling transforms:
Position object in scene



Viewing (camera) transform:
positions objects in coordinate space relative to camera
Canonical form: camera at origin looking down -z

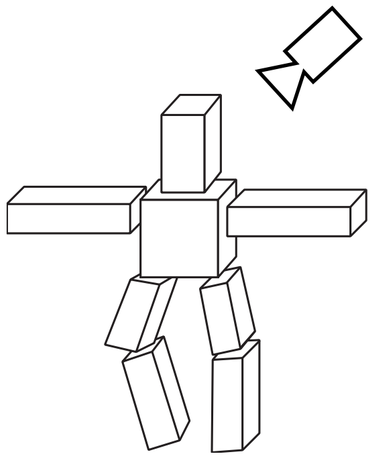


Projection transform + homogeneous divide:
Performs perspective projection
Canonical form: visible region of scene contained within unit cube

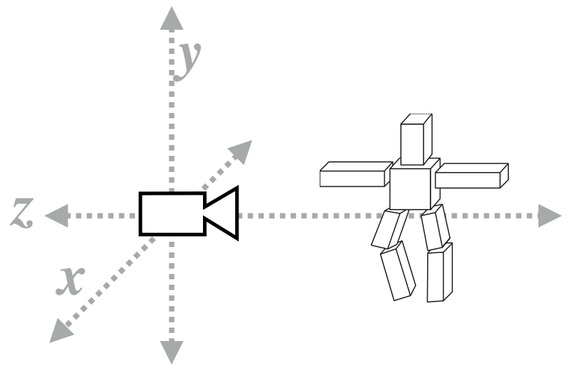


Screen transform:
objects now in 2D screen coordinates

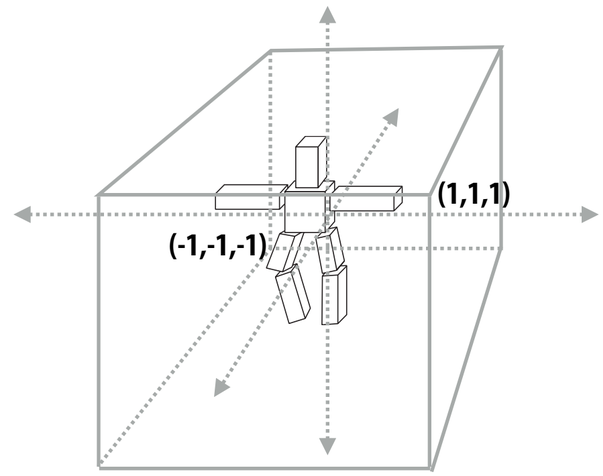
Transformations recap



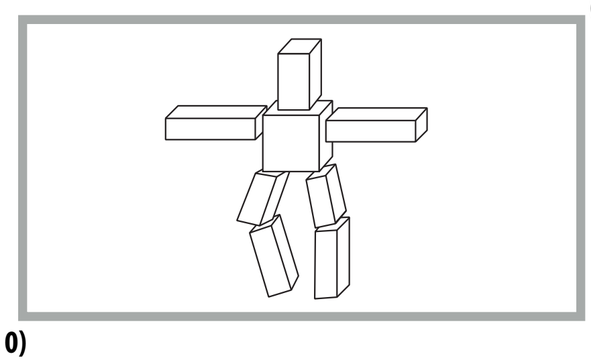
Modeling transforms:
Position object in scene



Viewing (camera) transform:
positions objects in coordinate space relative to camera
Canonical form: camera at origin looking down -z



Projection transform + homogeneous divide:
Performs perspective projection
Canonical form: visible region of scene contained within unit cube

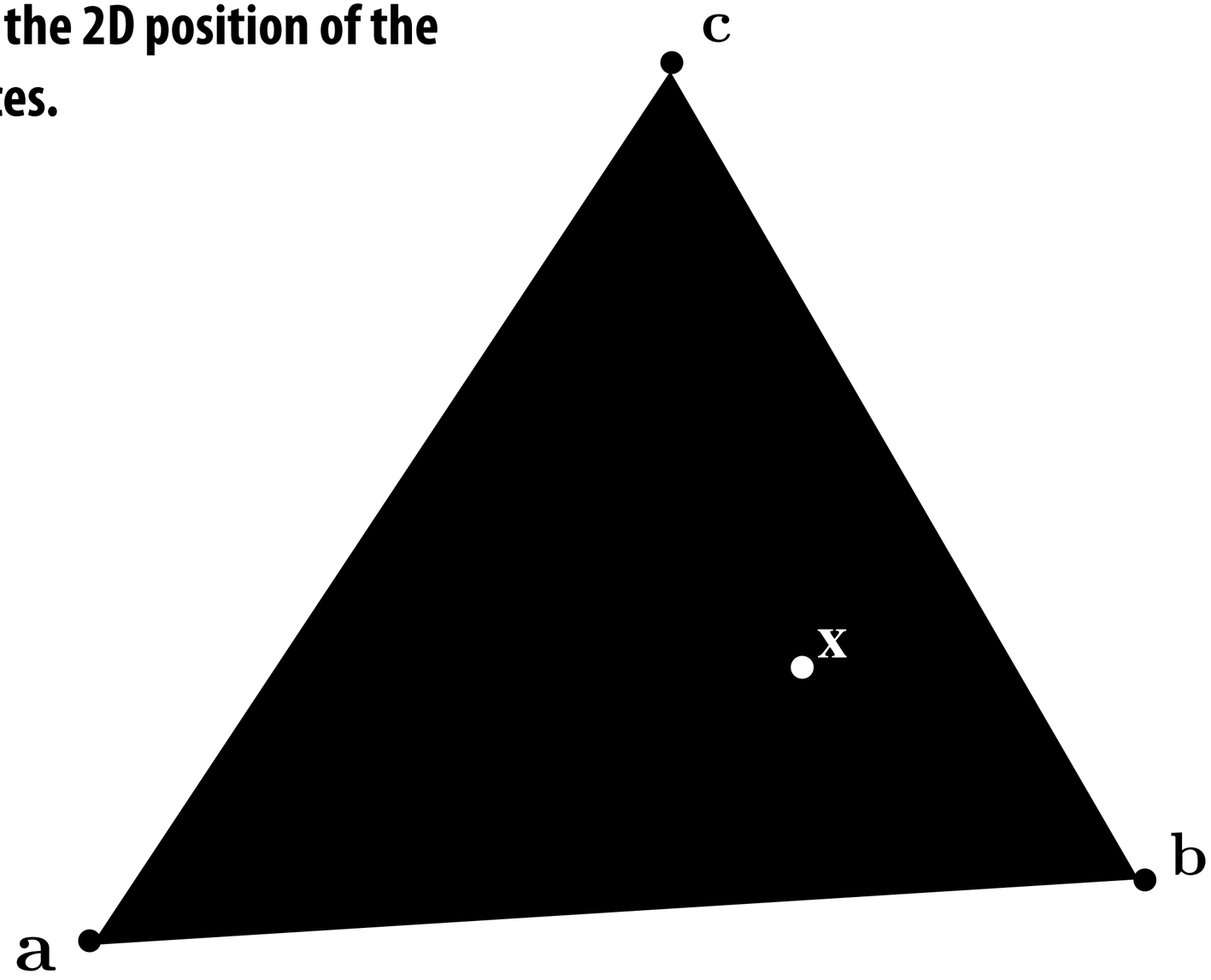


Screen transform:
objects now in 2D screen coordinates

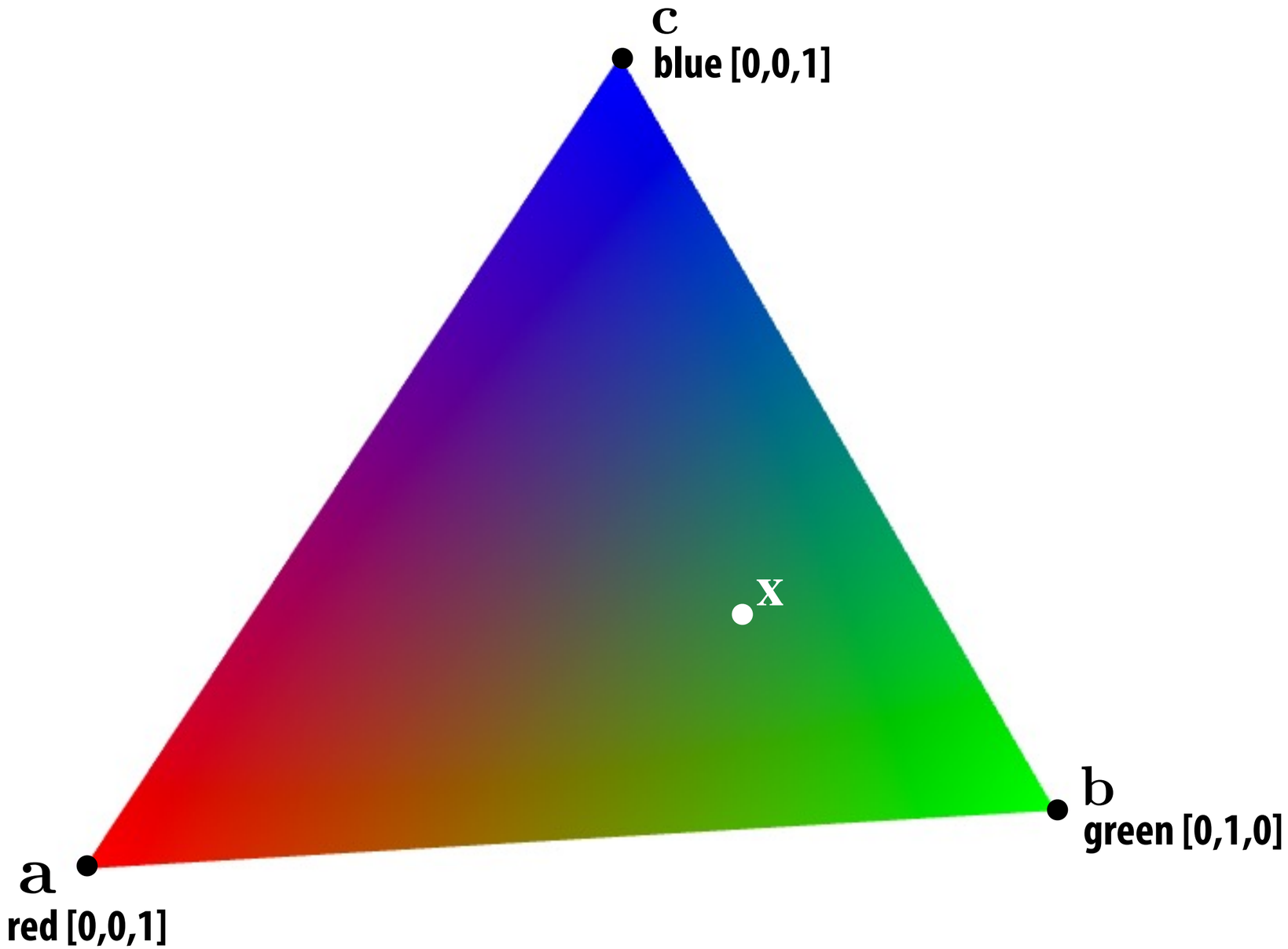
Compute screen coverage from 2D object position

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 ?

Review: interpolation in 1D

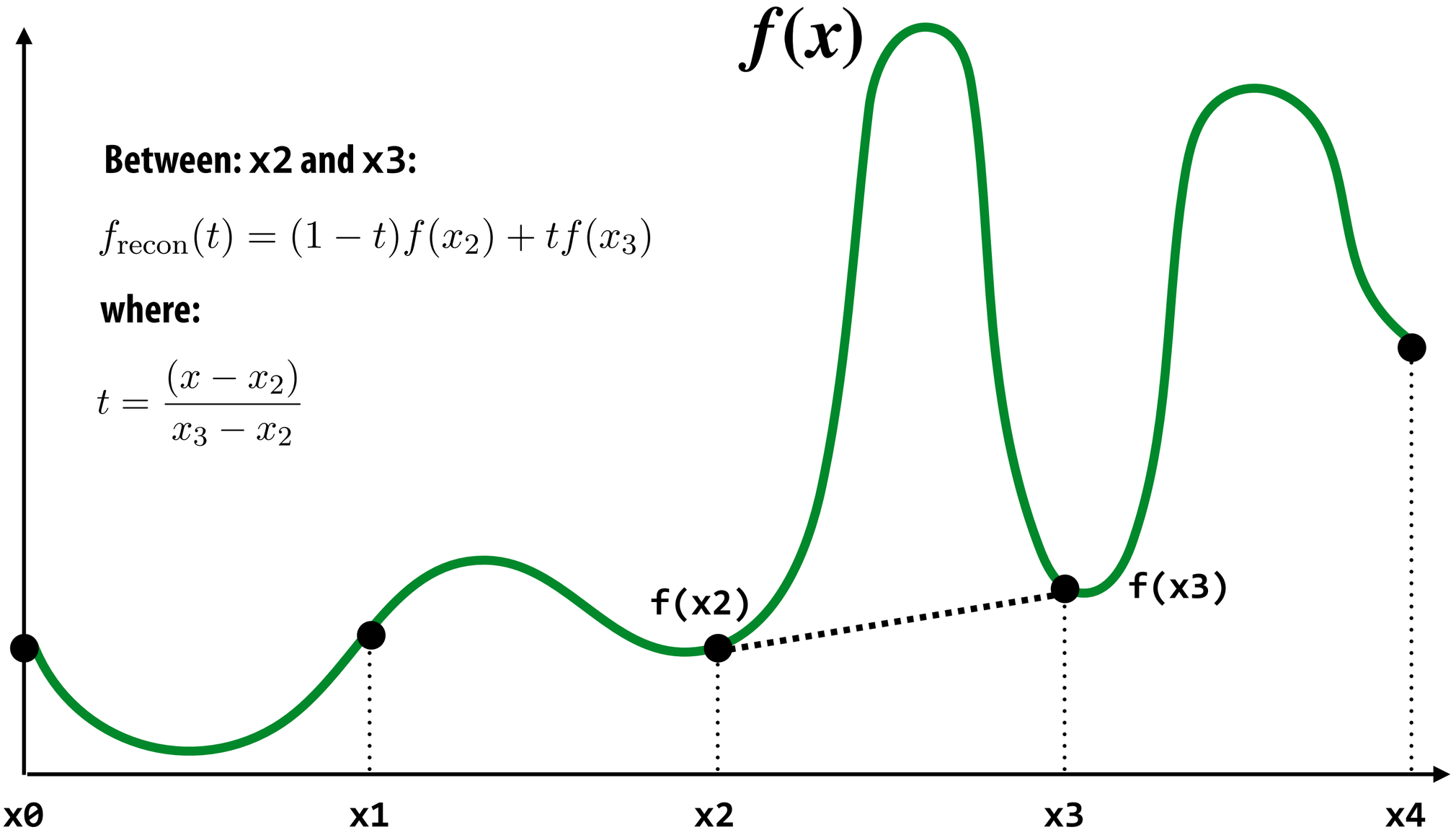
$f_{recon}(x)$ = linear interpolation between values of two closest samples to x

Between: x2 and x3:

$$f_{\text{recon}}(t) = (1-t)f(x_2) + tf(x_3)$$

where:

$$t = \frac{(x - x_2)}{x_3 - x_2}$$

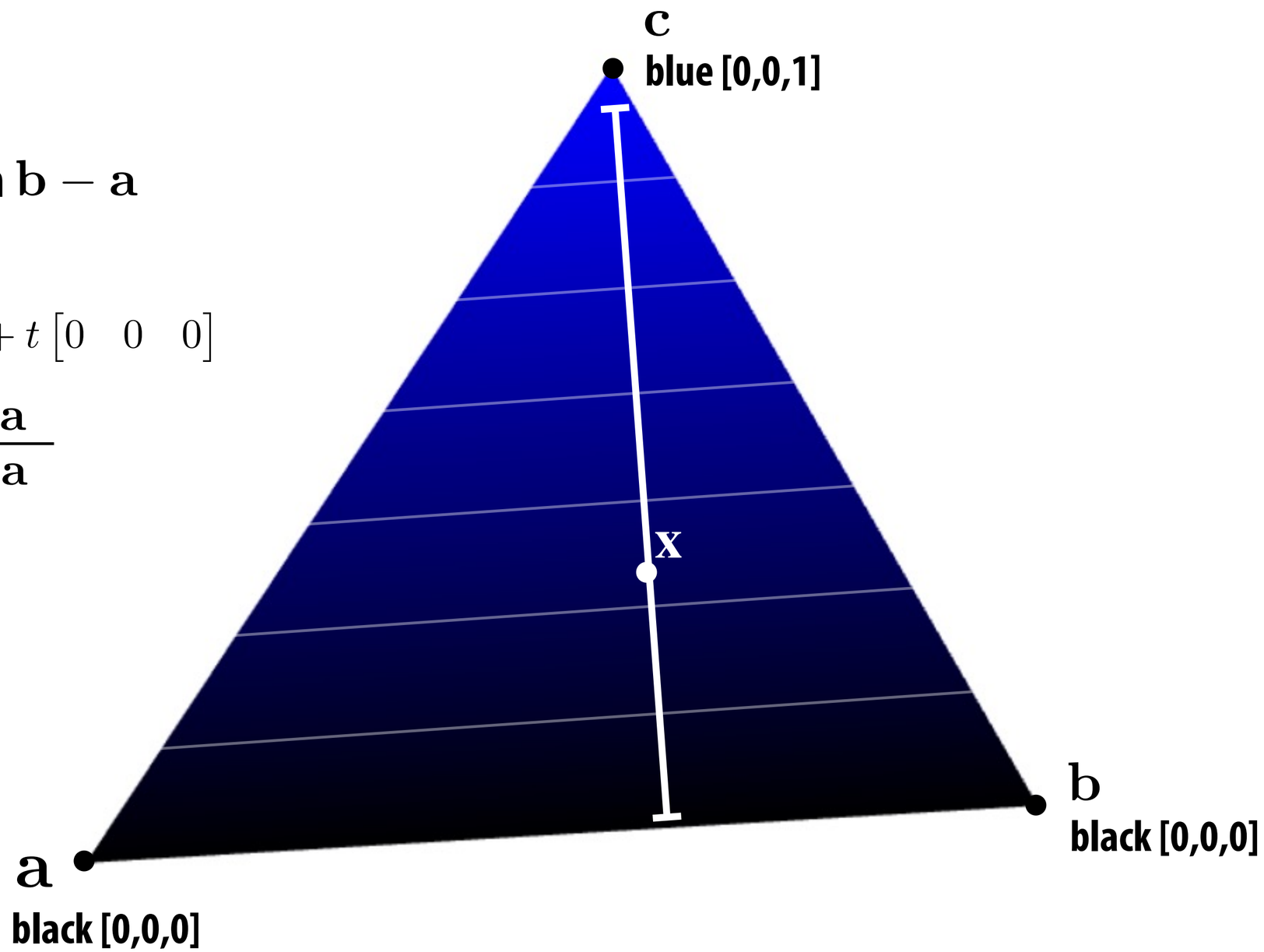


Consider similar behavior on triangle

Color depends on distance from $\mathbf{b} - \mathbf{a}$

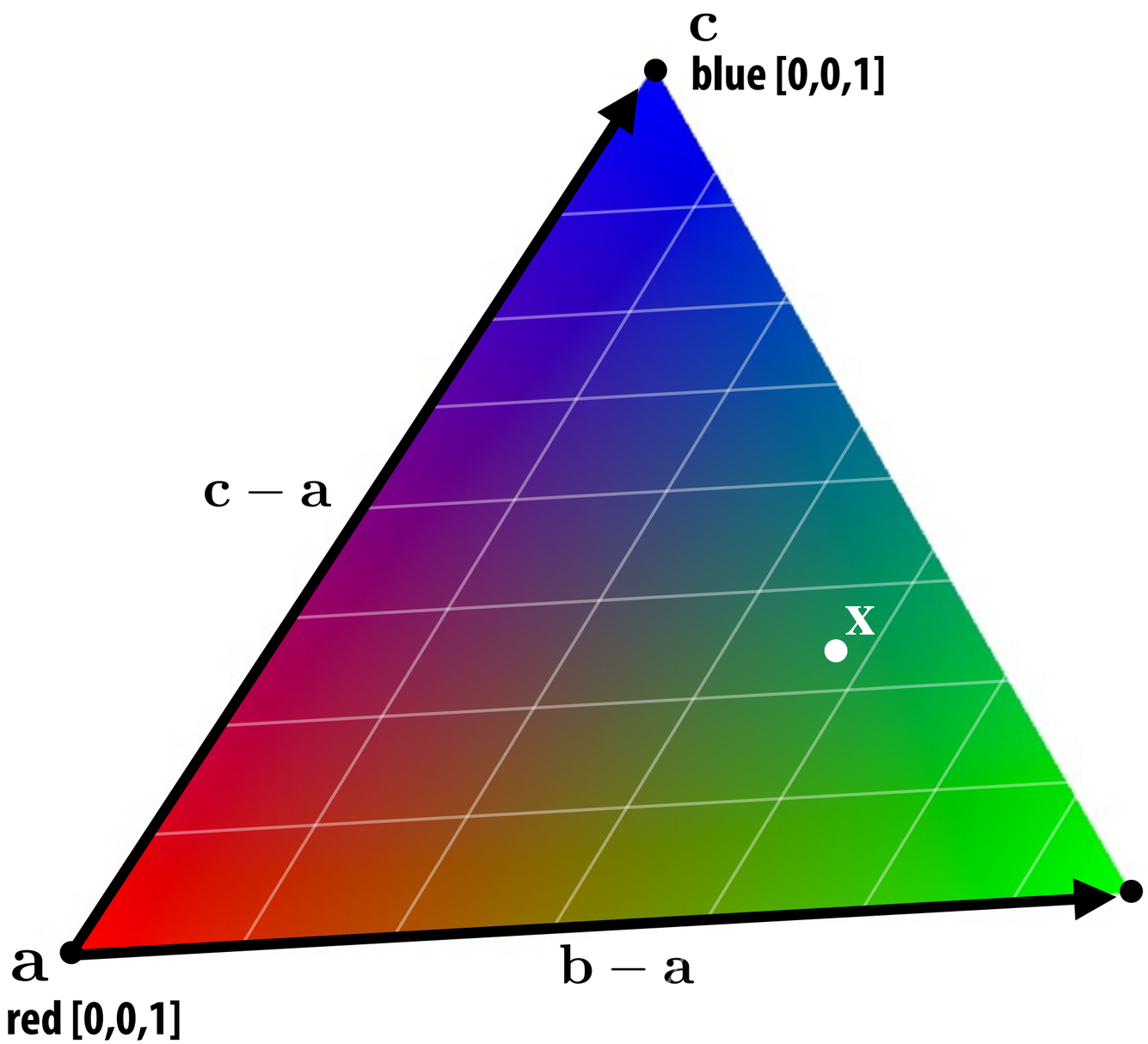
color at $\mathbf{x} = (1-t)\begin{bmatrix}0 & 0 & 1\end{bmatrix} + t\begin{bmatrix}0 & 0 & 0\end{bmatrix}$

$$t = \frac{\textbf{distance from } \mathbf{x} \textbf{ to } \mathbf{b}-\mathbf{a}}{\textbf{distance from } \mathbf{c} \textbf{ to } \mathbf{b}-\mathbf{a}}$$



How can we interpolate in 2D between three values?

Interpolation via barycentric coordinates



$\mathbf{b}-\mathbf{a}$ and $\mathbf{c}-\mathbf{a}$ form a non-orthogonal basis for points in triangle (origin at $\mathbf{a}$)

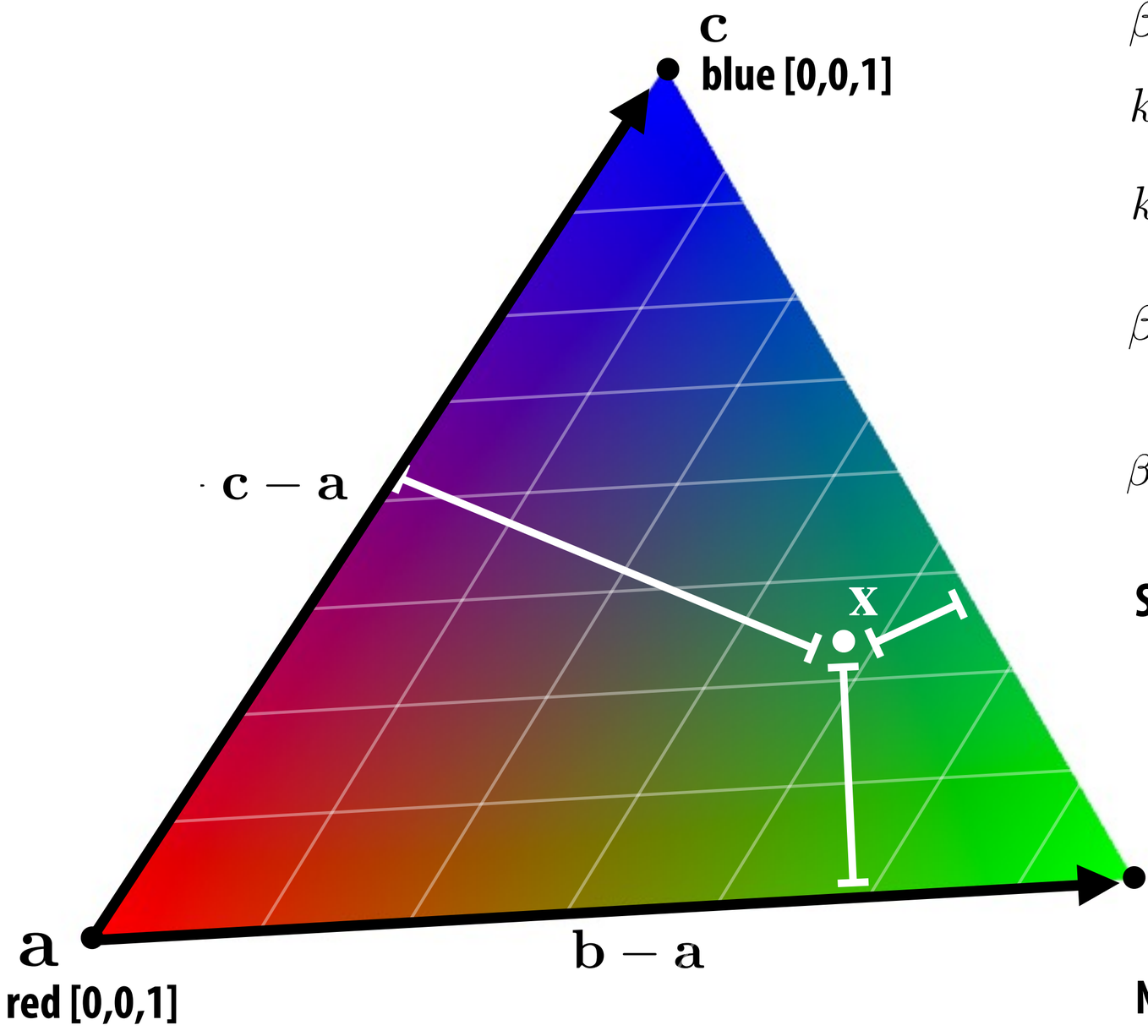
$$\begin{aligned}\mathbf{x} &= \mathbf{a} + \beta(\mathbf{b}-\mathbf{a}) + \gamma(\mathbf{c}-\mathbf{a}) \\ &= (1-\beta-\gamma)\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c} \\ &= \alpha\mathbf{a} + \beta\mathbf{b} + \gamma\mathbf{c}\end{aligned}$$

$$\alpha + \beta + \gamma = 1$$

Color at $\mathbf{x}$ is linear combination of color at three triangle vertices.

$$\mathbf{x}_{\text{color}} = \alpha\mathbf{a}_{\text{color}} + \beta\mathbf{b}_{\text{color}} + \gamma\mathbf{c}_{\text{color}}$$

Barycentric coordinates as scaled distances



β **proportional to distance from** $\mathbf{x}$ **to edge** $\mathbf{c} - \mathbf{a}$

$$kE_{\mathbf{ac}}(\mathbf{x}_x, \mathbf{x}_y) = \beta$$

$$kE_{\mathbf{ac}}(\mathbf{b}_x, \mathbf{b}_y) = 1$$

$$\beta = \frac{E_{\mathbf{ac}}(\mathbf{x}_x, \mathbf{x}_y)}{E_{\mathbf{ac}}(\mathbf{b}_x, \mathbf{b}_y)}$$

$$\beta = \frac{(\mathbf{a}_y - \mathbf{c}_y)\mathbf{x}_x + (\mathbf{c}_x - \mathbf{a}_x)\mathbf{x}_y + \mathbf{a}_x\mathbf{c}_y - \mathbf{c}_x\mathbf{a}_y}{(\mathbf{a}_y - \mathbf{c}_y)\mathbf{b}_x + (\mathbf{c}_x - \mathbf{a}_x)\mathbf{b}_y + \mathbf{a}_x\mathbf{c}_y - \mathbf{c}_x\mathbf{a}_y}$$

Similiar computation for γ

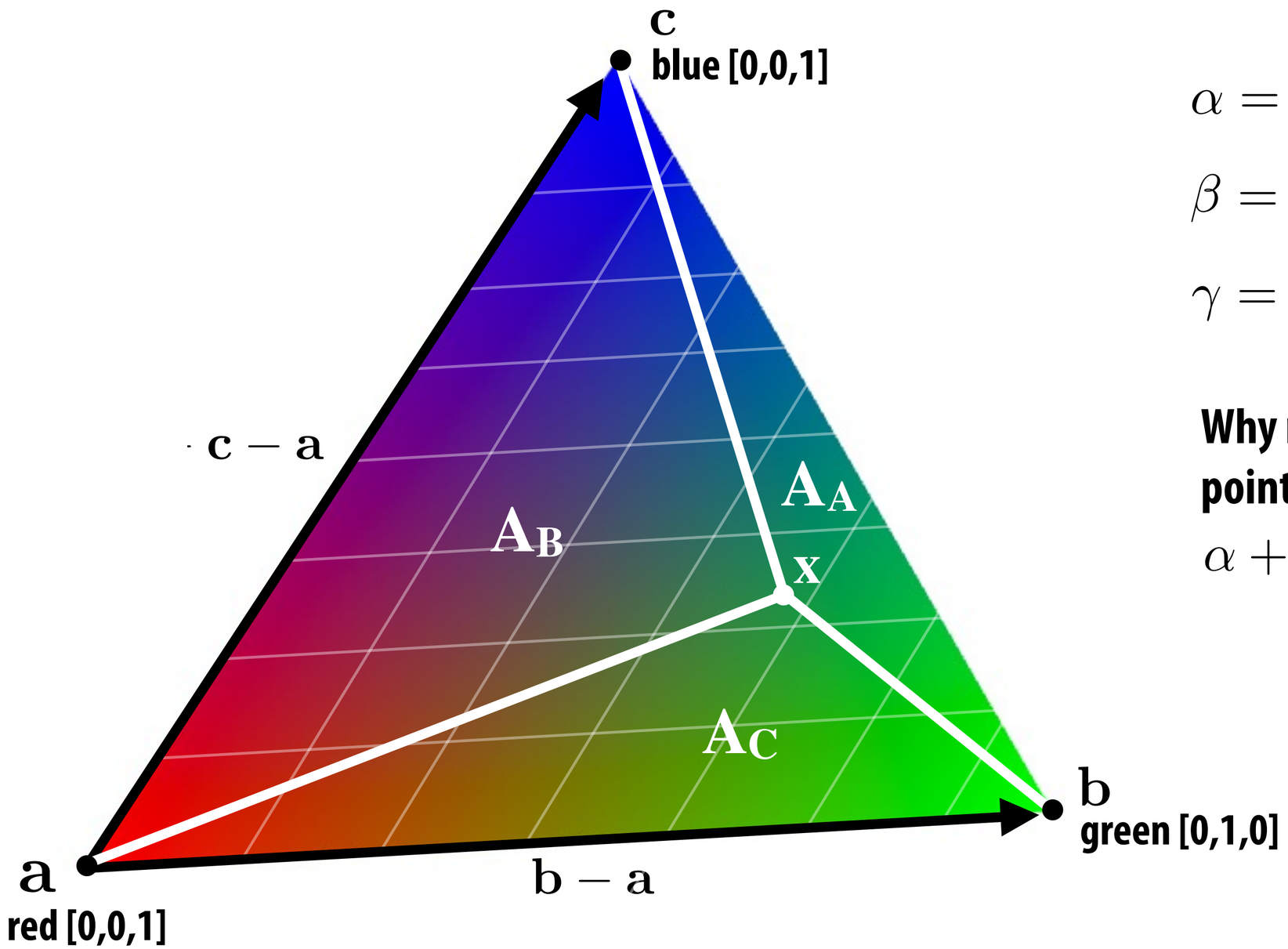
Note:

Barycentric coordinates are scaled distances →
Barycentric coordinates are affine function of $\mathbf{x}$

$$\mathbf{x}_{\text{color}} = \alpha\mathbf{a}_{\text{color}} + \beta\mathbf{b}_{\text{color}} + \gamma\mathbf{c}_{\text{color}}$$

$$\mathbf{x}_{\text{color}} = A\mathbf{x}_x + B\mathbf{x}_y + C$$

Barycentric coordinates as ratio of areas



$$\alpha = A_A / A$$

$$\beta = A_B / A$$

$$\gamma = A_C / A$$

Why must coordinates sum to one for points inside triangle?

$$\alpha + \beta + \gamma = 1$$

Also a valid interpretation of barycentric coordinates for a triangle in 3D

Direct evaluation of surface attributes

For any surface attribute (with value defined at triangle vertices as: f_a, f_b, f_c)

$$f_a = A\mathbf{a}_x + B\mathbf{a}_y + C$$

$$f_b = A\mathbf{b}_x + B\mathbf{b}_y + C$$

$$f_c = A\mathbf{c}_x + B\mathbf{c}_y + C$$

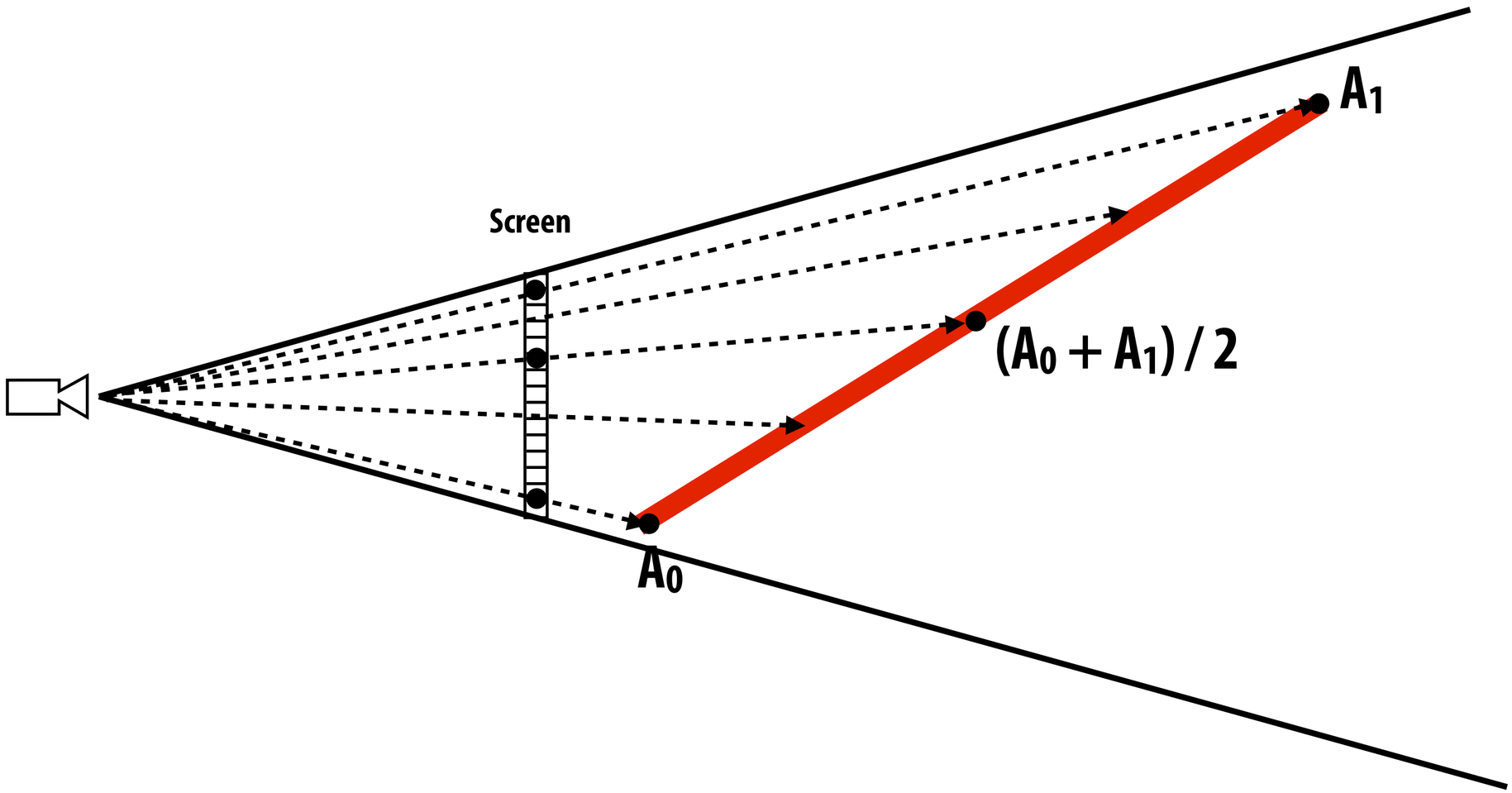
3 equations, solve for 3 unknowns (A, B, C)

3 equations, solve for 3 unknowns (A, B, C)

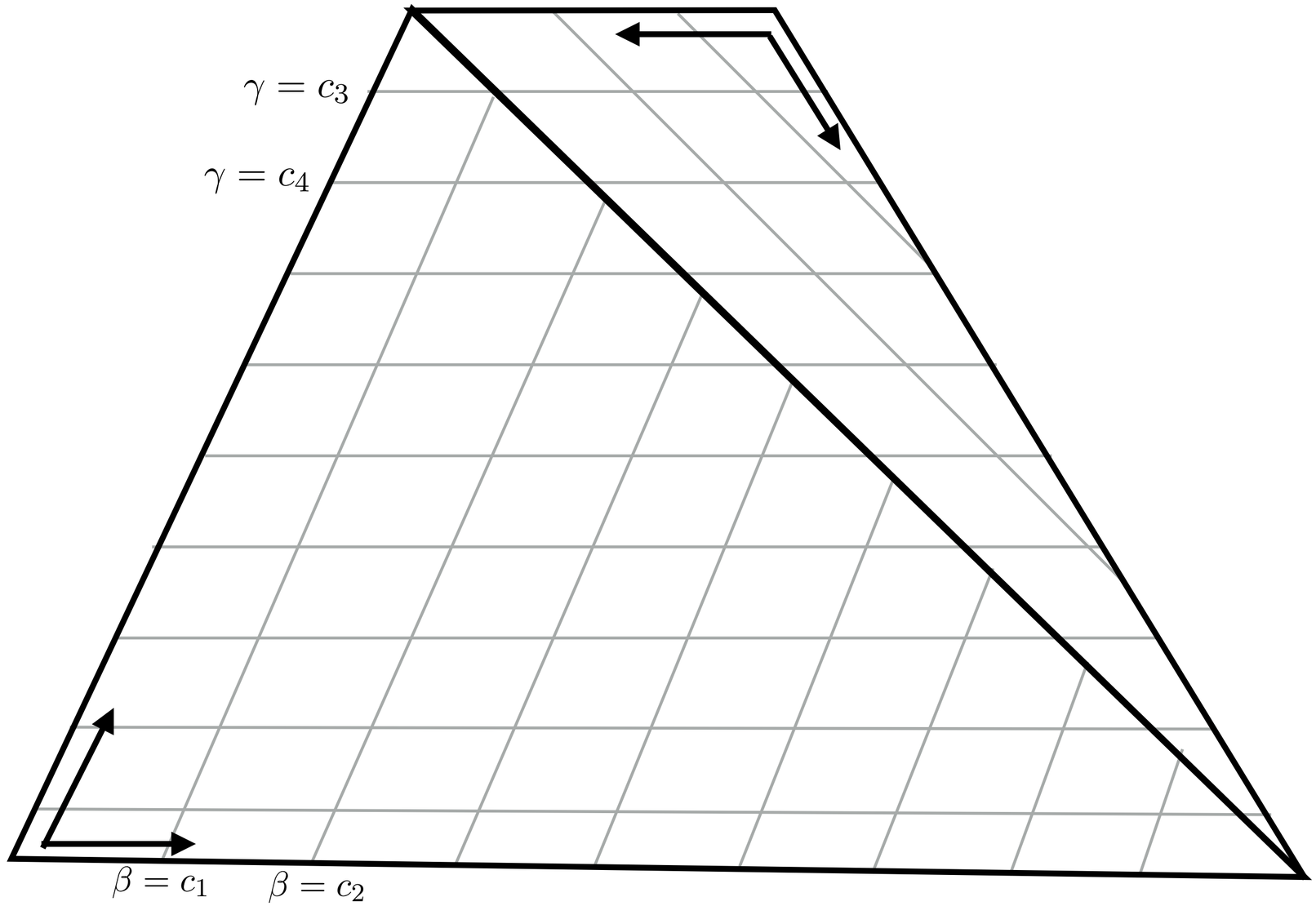
Perspective-incorrect interpolation

Due to projection, linear 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.



Another example: not perspective correct interpolation



Perspective-correct interpolation

Assume triangle attribute varies linearly across the triangle

Attribute's value at 3D (non-homogeneous) point $P = \begin{bmatrix} x & y & z \end{bmatrix}^T$ **is:**

$$f(x, y, z) = ax + by + cz$$

Project P**, get 2D homogeneous representation:** $\begin{bmatrix} x_{\text{2D-H}} & y_{\text{2D-H}} & w \end{bmatrix}^T = \begin{bmatrix} x & y & z \end{bmatrix}^T$

Rewrite attribute equation for f **in terms of 2D homogeneous coordinates:**

$$f = ax_{\text{2D-H}} + by_{\text{2D-H}} + cw$$

$$\frac{f}{w} = a\frac{x_{\text{2D-H}}}{w} + b\frac{y_{\text{2D-H}}}{w} + c$$

$$\frac{f}{w} = ax_{\text{2D}} + by_{\text{2D}} + c$$

Where $\begin{bmatrix} x_{\text{2D}} & y_{\text{2D}} \end{bmatrix}^T$ **are projected screen 2D coordinates (after homogeneous divide)**

So … $\frac{f}{w}$ **is affine function of 2D screen coordinates:** $\begin{bmatrix} x_{\text{2D}} & y_{\text{2D}} \end{bmatrix}^T$

Efficient perspective-correct interpolation

Attribute values vary linearly across triangle in 3D, but not in projected screen XY

Projected attribute values (f/w) are affine functions of screen XY!

To evaluate surface attribute f at every covered sample:

- **Evaluate $1/w\,(x,y)$** **(from precomputed equation for $1/w$)**
- **Reciprocate $1/w\,(x,y)$ to get $w(x,y)$**
- **For each triangle attribute:**
  - **Evaluate $f/w\,(x,y)$** **(from precomputed equation for f/w)**
  - **Multiply $f/w\,(x,y)$ by $w(x,y)$ to get $f\,(x,y)$**

Works for any surface attribute f that varies linearly across triangle:

e.g., color, depth, texture coordinates

Texture mapping

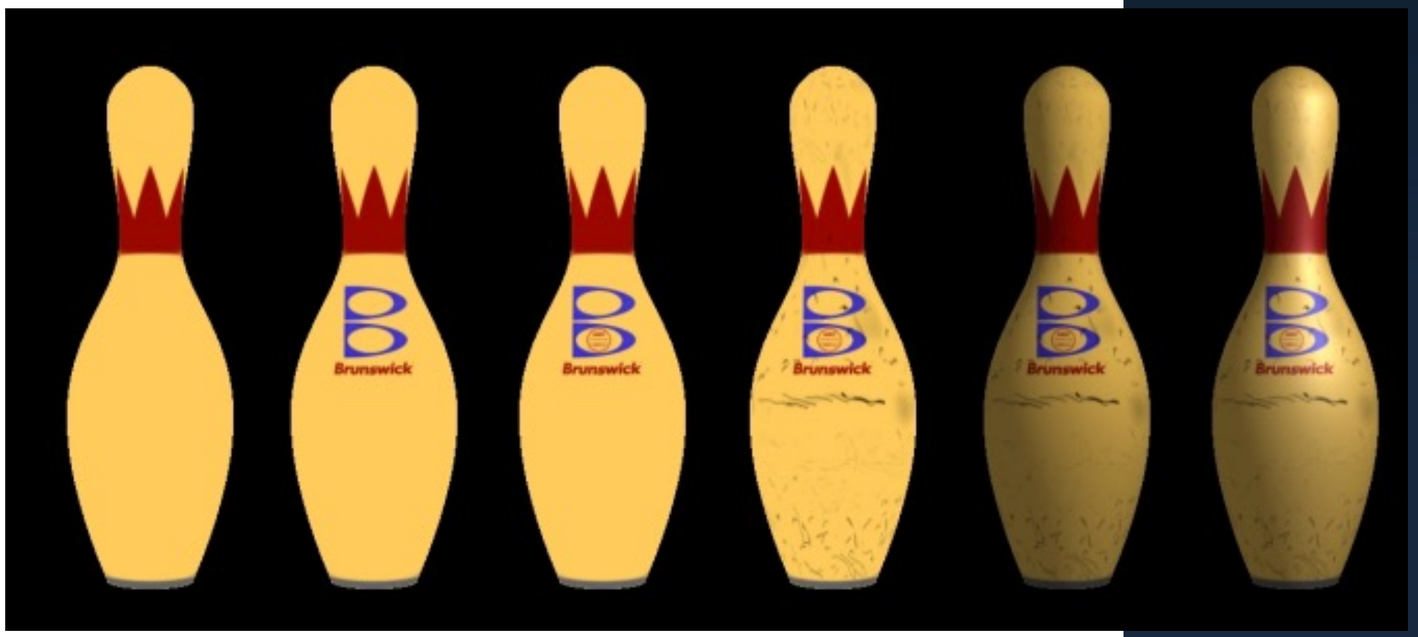
Many uses of texture mapping

Define variation in surface reflectance

Pattern on ball

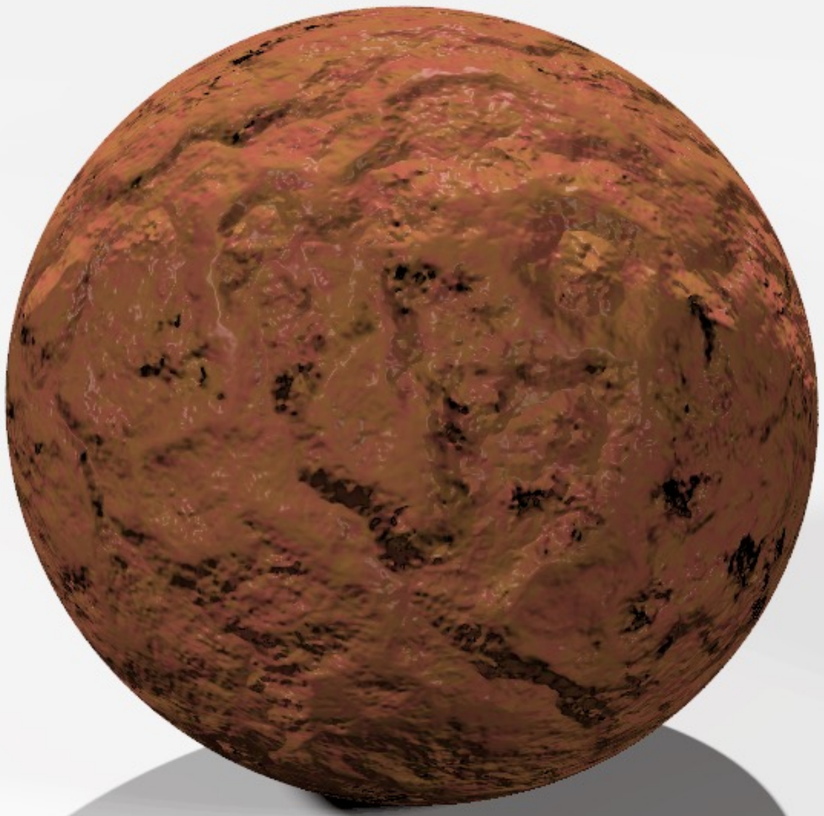
Wood grain on floor

Describe surface material properties



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

Normal mapping



Use texture value to perturb surface normal to give appearance of a bumpy surface
Observe: smooth silhouette and smooth shadow boundary indicates surface geometry is not bumpy

Rendering using high-resolution surface geometry (note bumpy silhouette and shadow boundary)

Represent precomputed lighting and shadows

Original model

With ambient occlusion

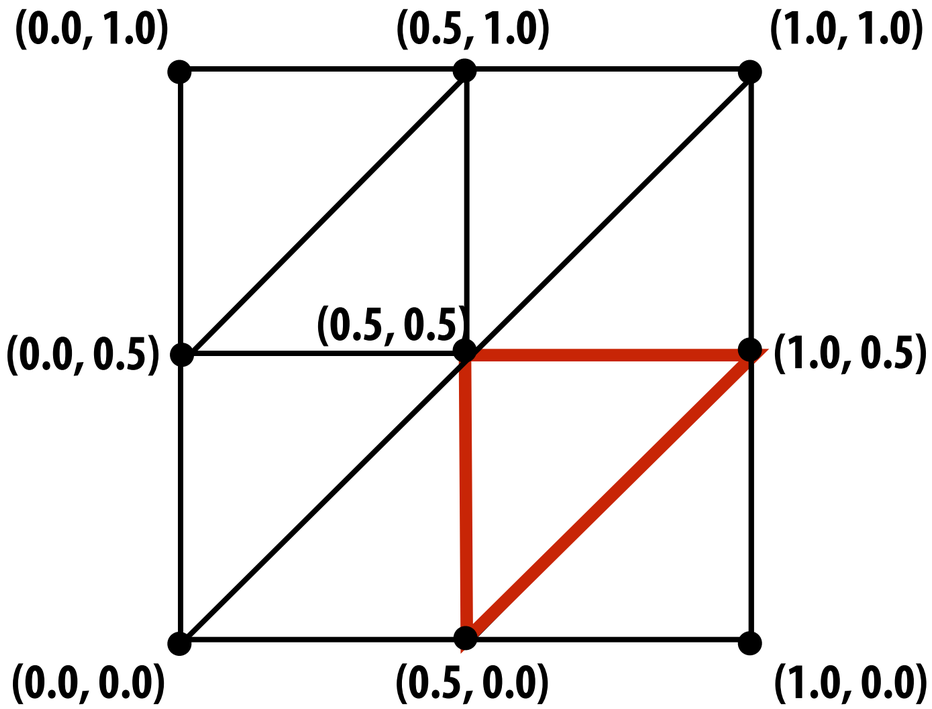
Extracted ambient occlusion map

Grace Cathedral environment map

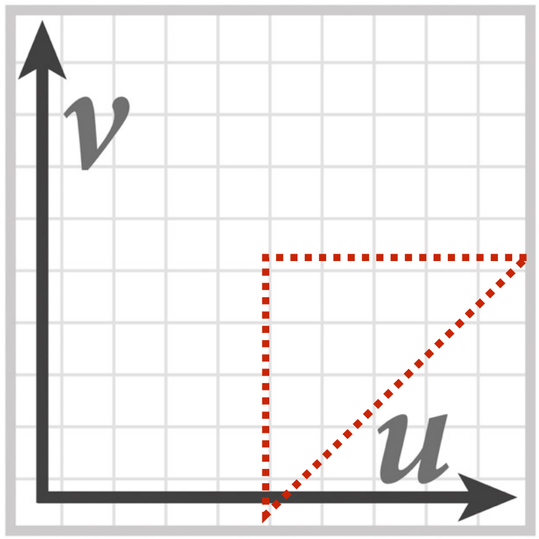
Environment map used in rendering

Texture coordinates

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



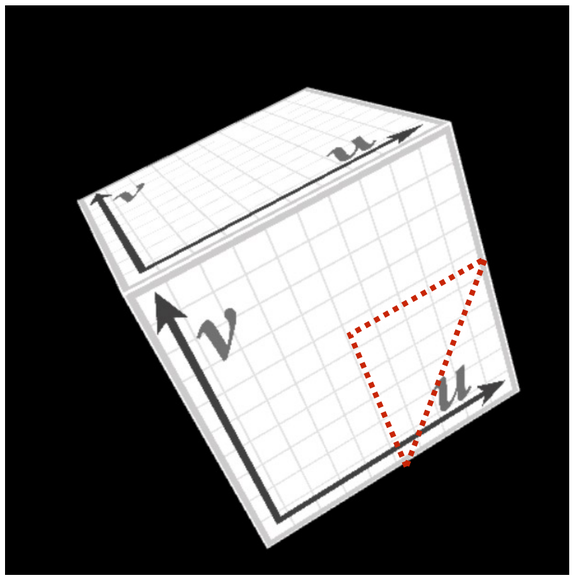
Eight triangles (one face of cube) with surface parameterization provided as per-vertex texture coordinates.



`myTex(u,v)` is a function defined on the $[0,1]^2$ domain:

myTex : $[0,1]^2 \rightarrow$ float3
(represented by 2048x2048 image)

Location of highlighted triangle in texture space shown in red.



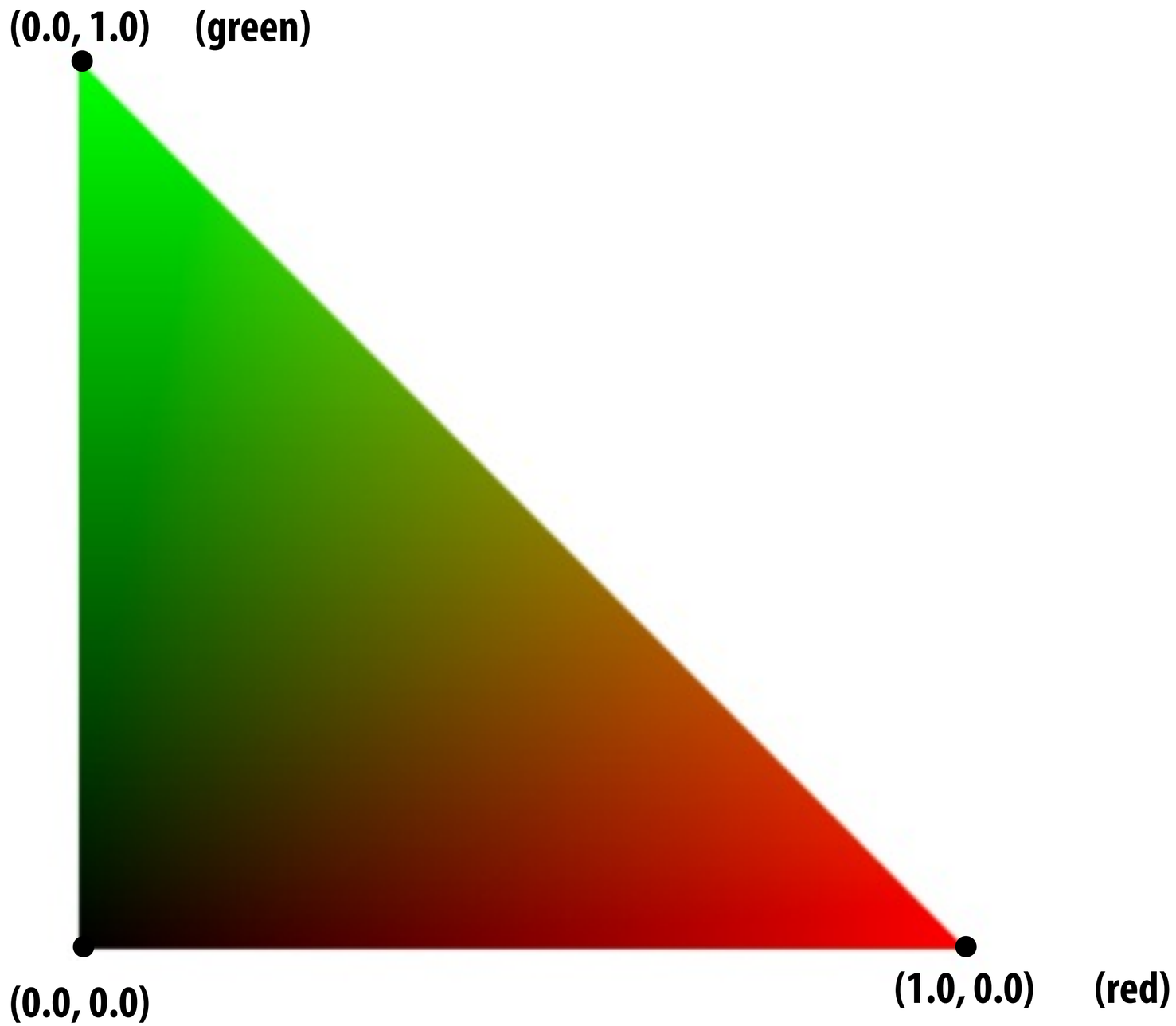
Final rendered result (entire cube shown).

Location of triangle after projection onto screen shown in red.

Today we'll assume surface-to-texture space mapping is provided as per vertex values (Methods for generating surface texture parameterizations will be discussed in a later lecture)

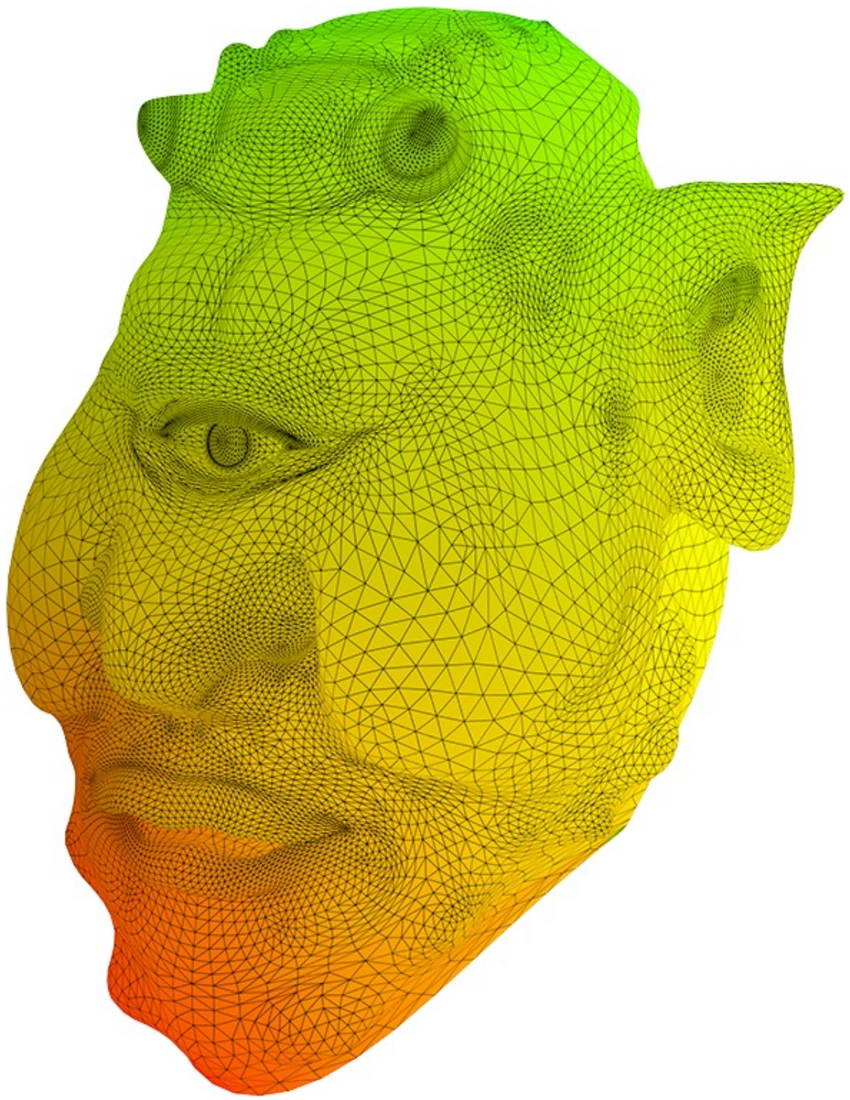
Visualization of texture coordinates

Texture coordinates linearly interpolated over triangle

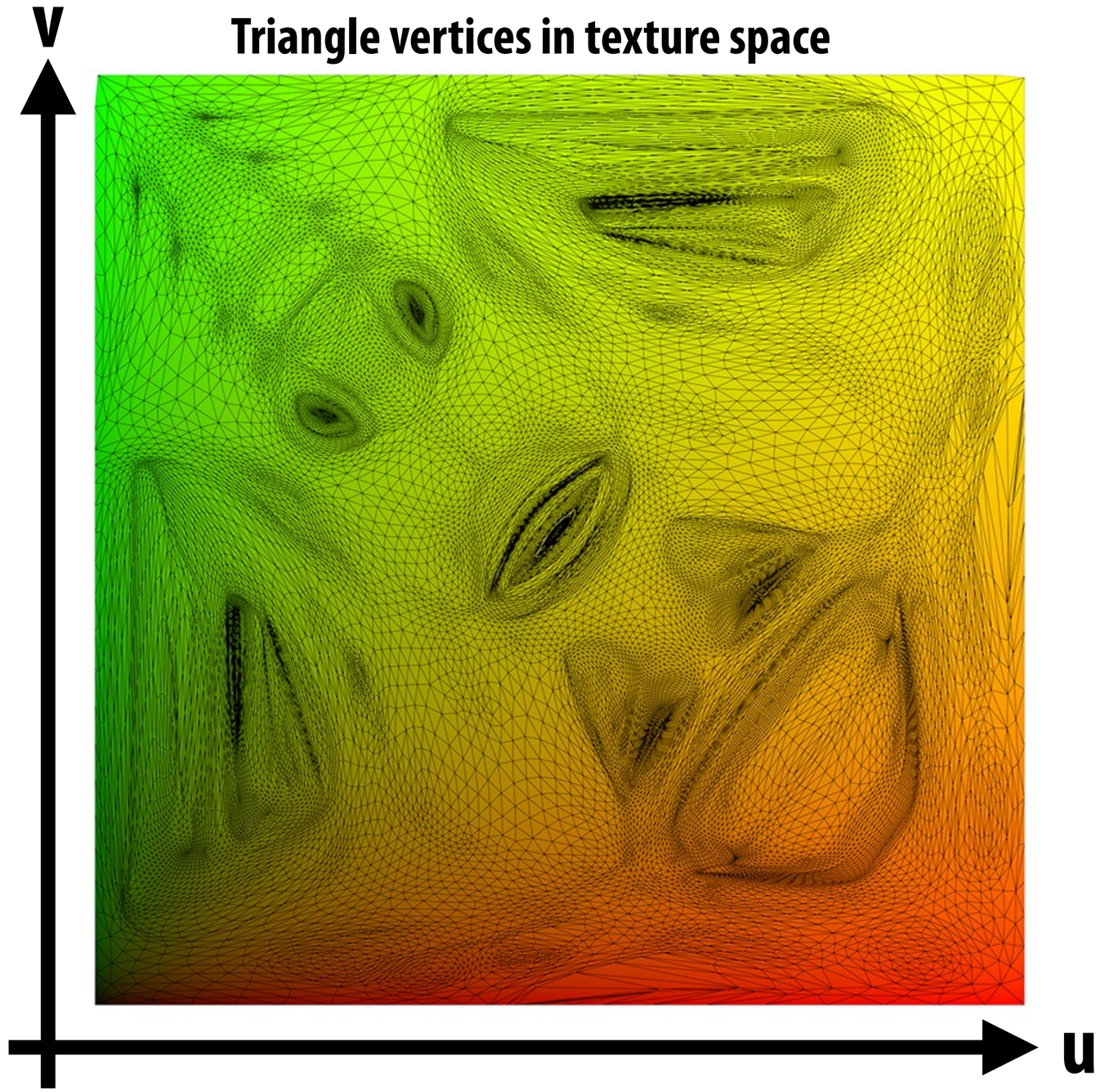


More complex mapping

Visualization of texture coordinates



Triangle vertices in texture space



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

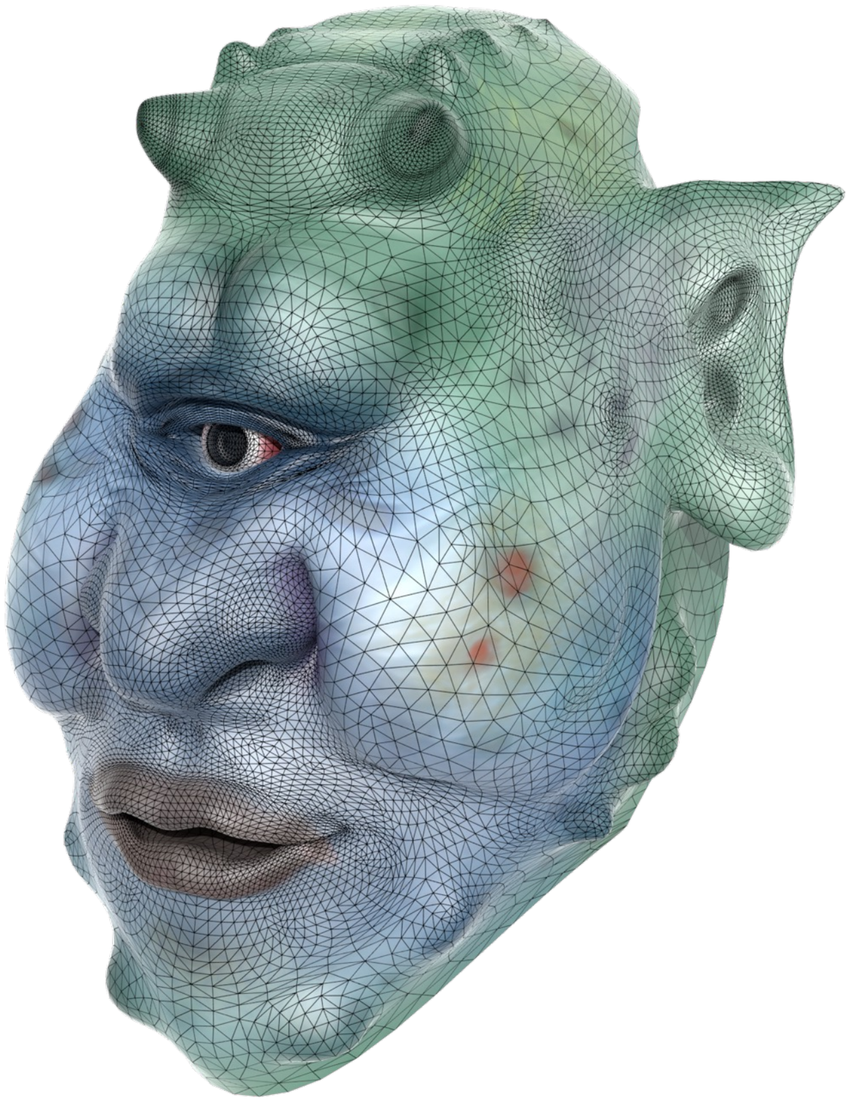
(Actually coming up with these coordinates is another story!)

Simple texture mapping operation

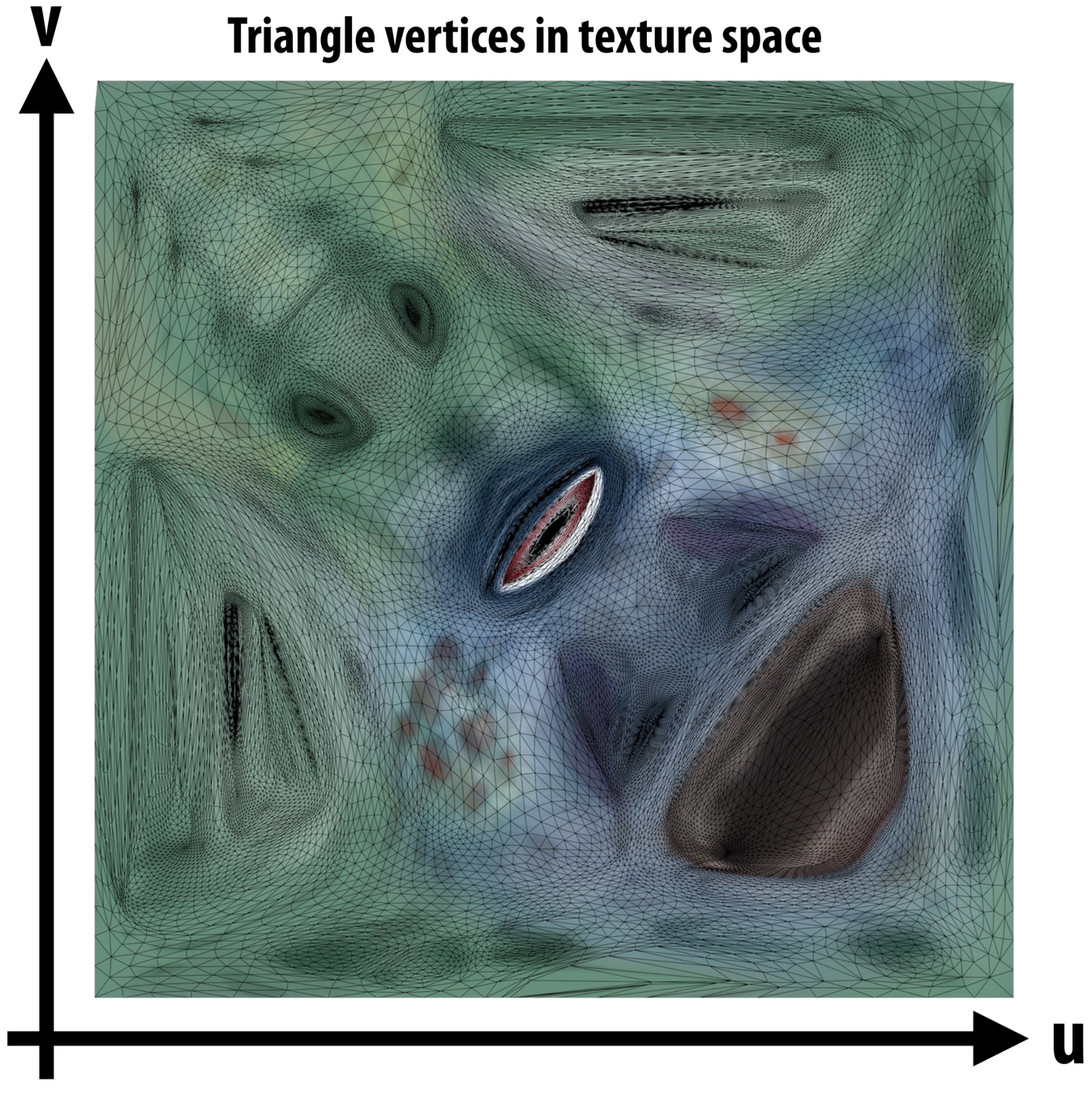
```
for each covered screen sample (x,y):
    (u,v) = evaluate texcoord value at (x,y)
    float3 texcolor = texture.sample(u,v);
    set sample's color to texcolor;
```

Texture mapping adds detail

Rendered result



Triangle vertices in texture space

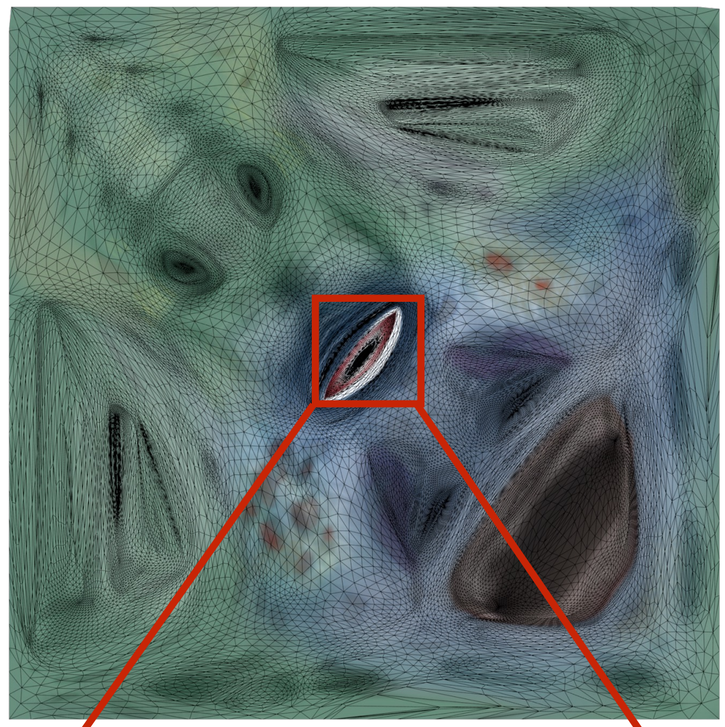


Texture mapping adds detail

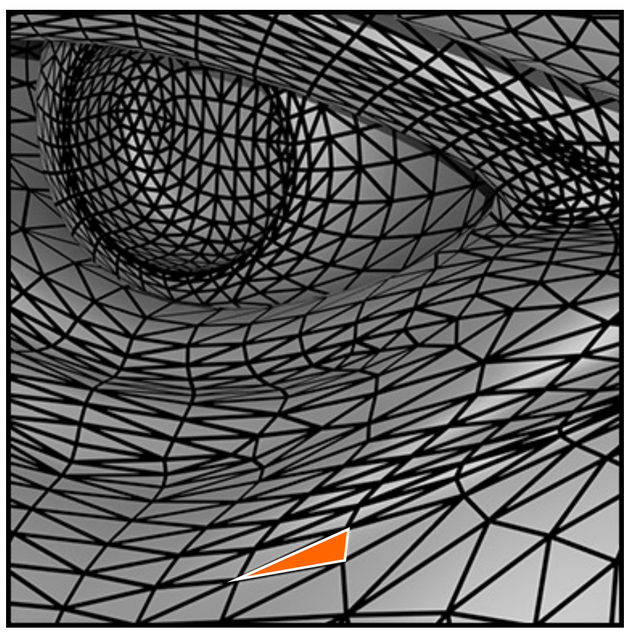
rendering without texture

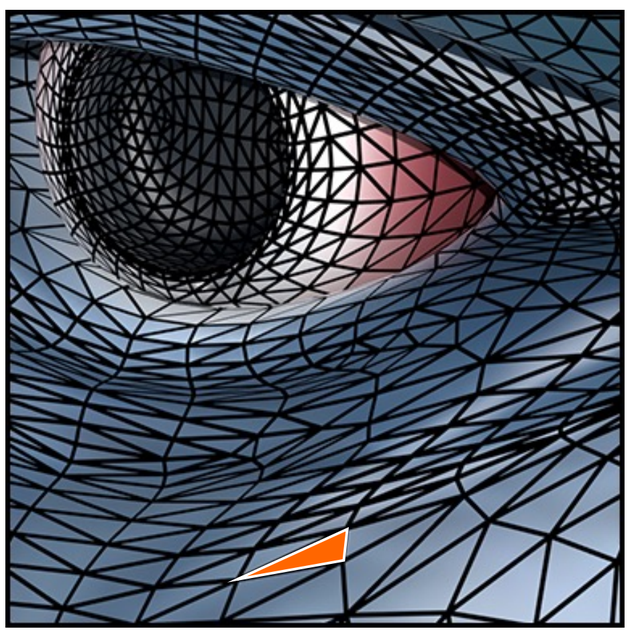
rendering with texture

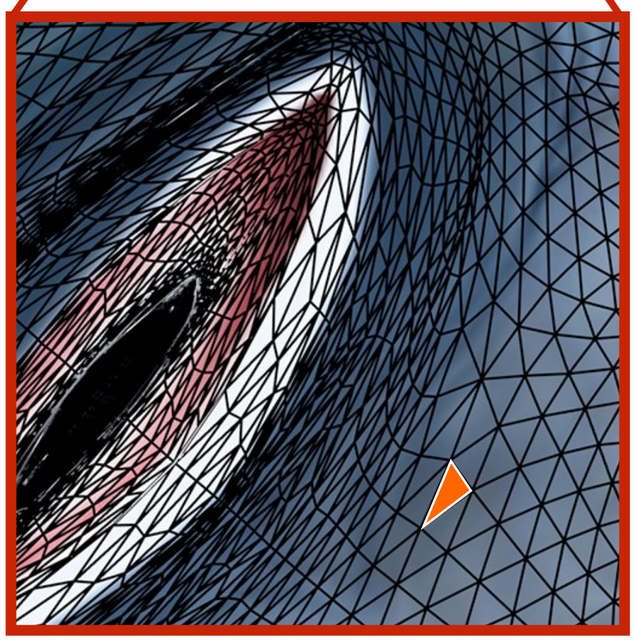
texture image



zoom

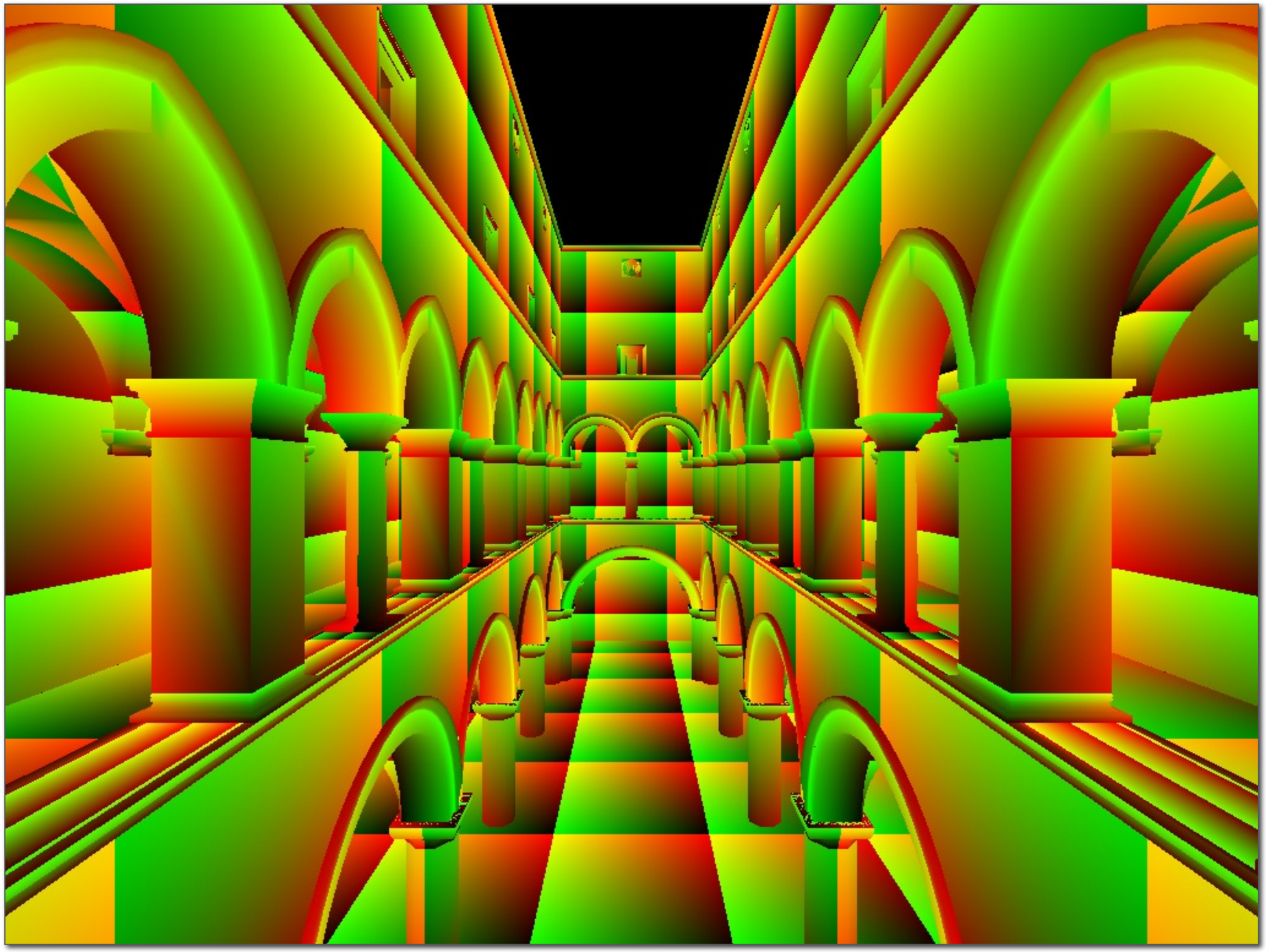






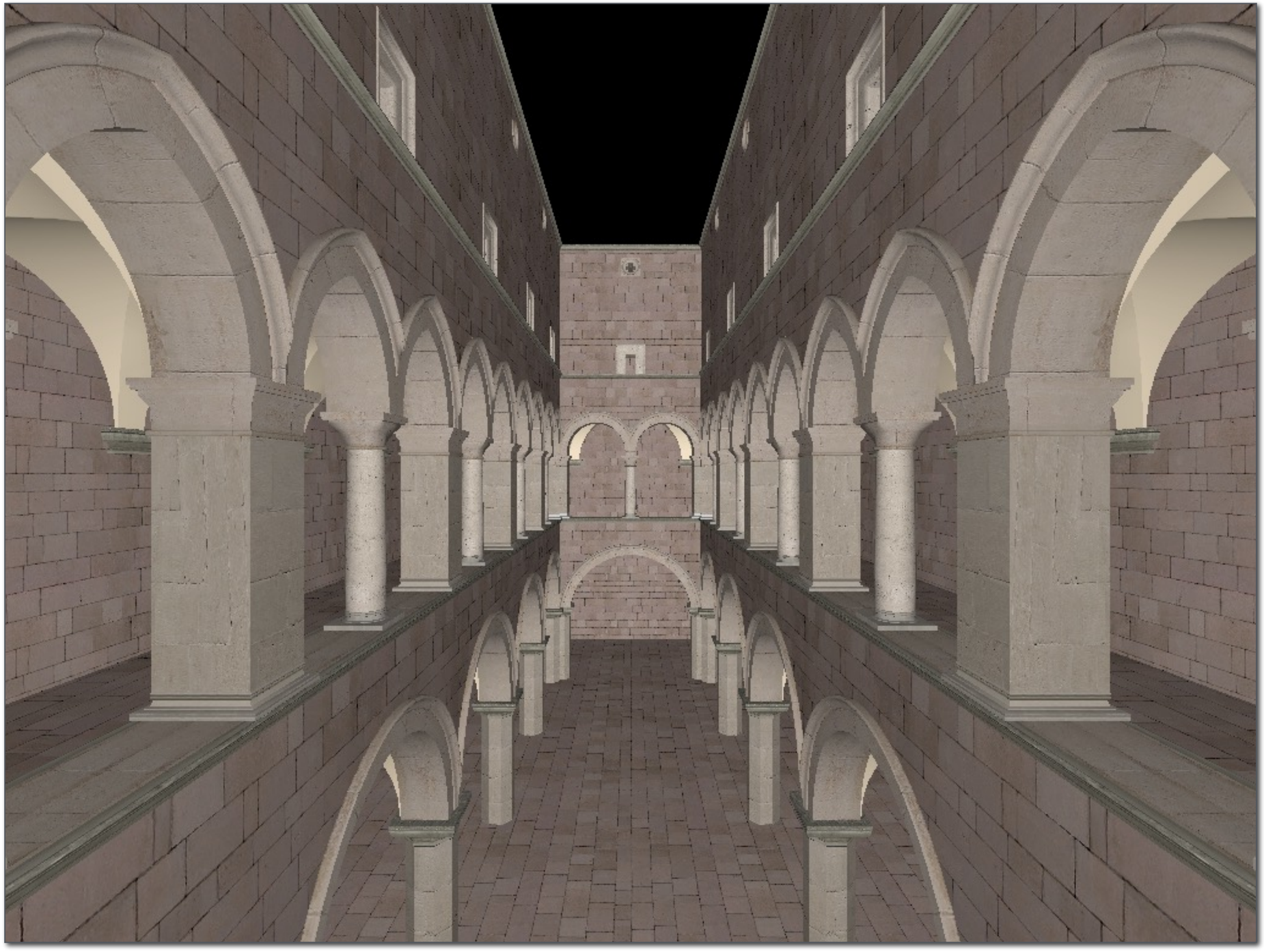
Each triangle "copies" a piece of the image back to the surface.

Another example: Sponza



Notice texture coordinates repeat over surface.

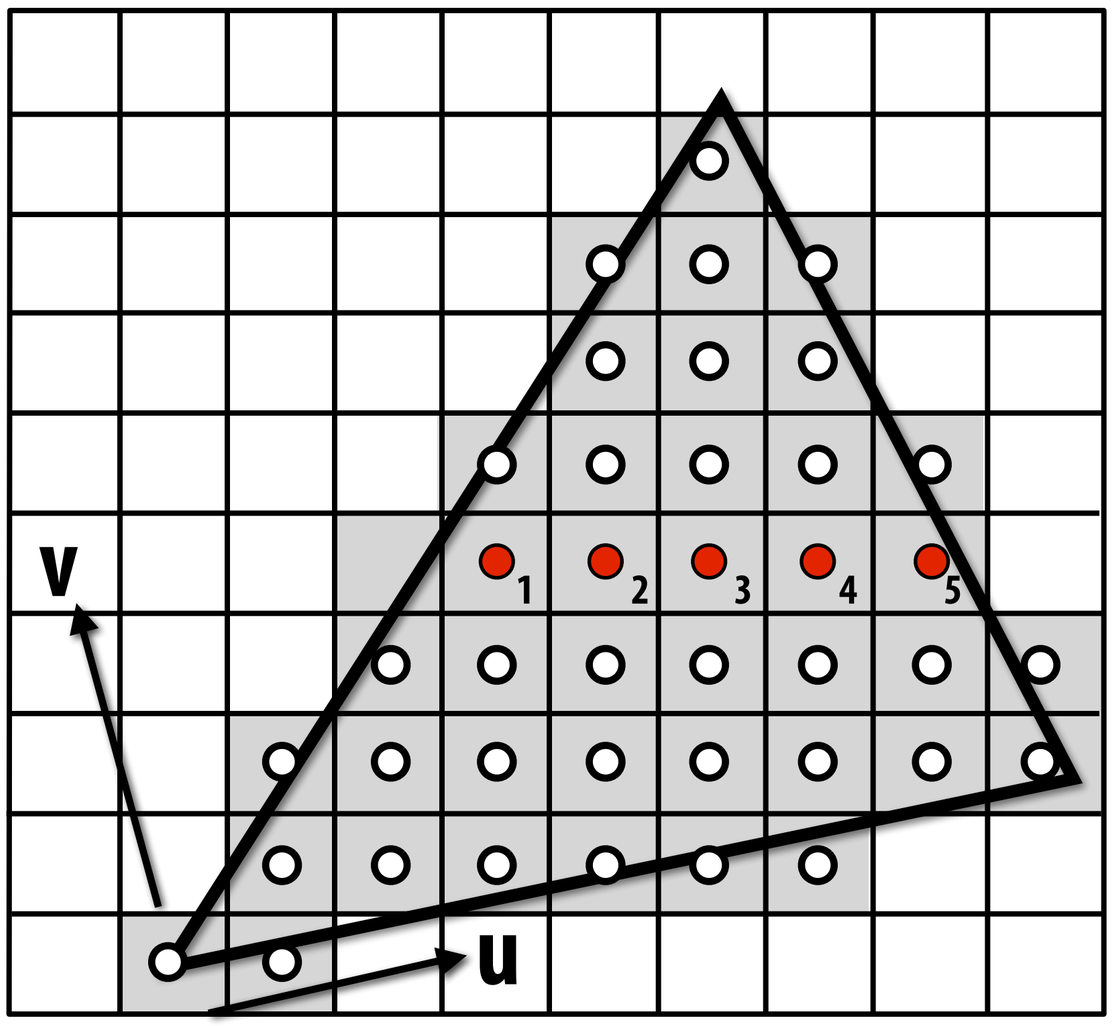
Textured Sponza



Example textures used in Sponza

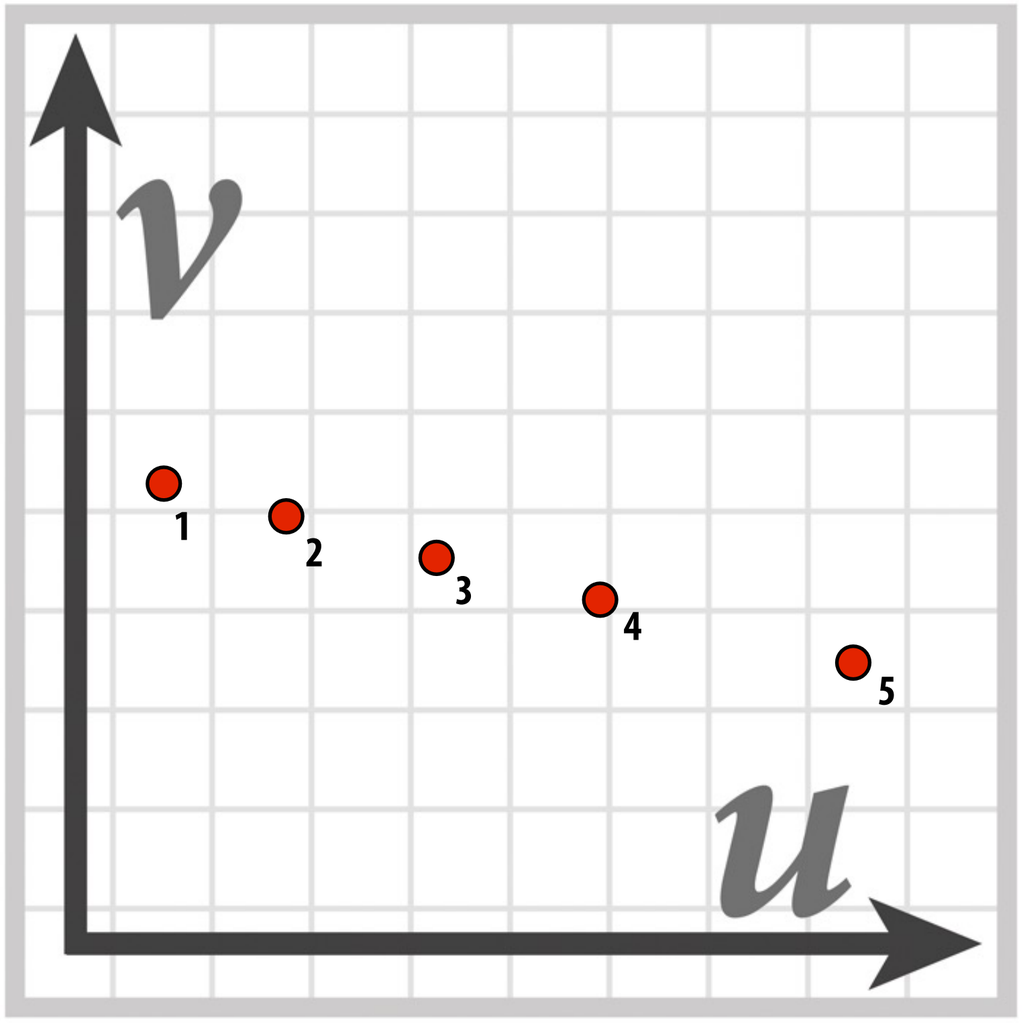
Texture space samples

Sample positions in XY screen space



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

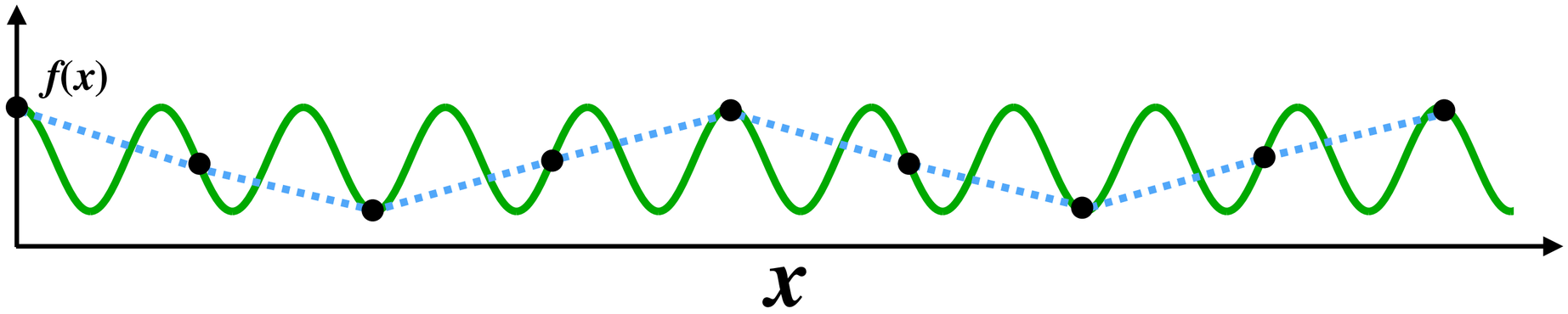
Sample positions in texture space



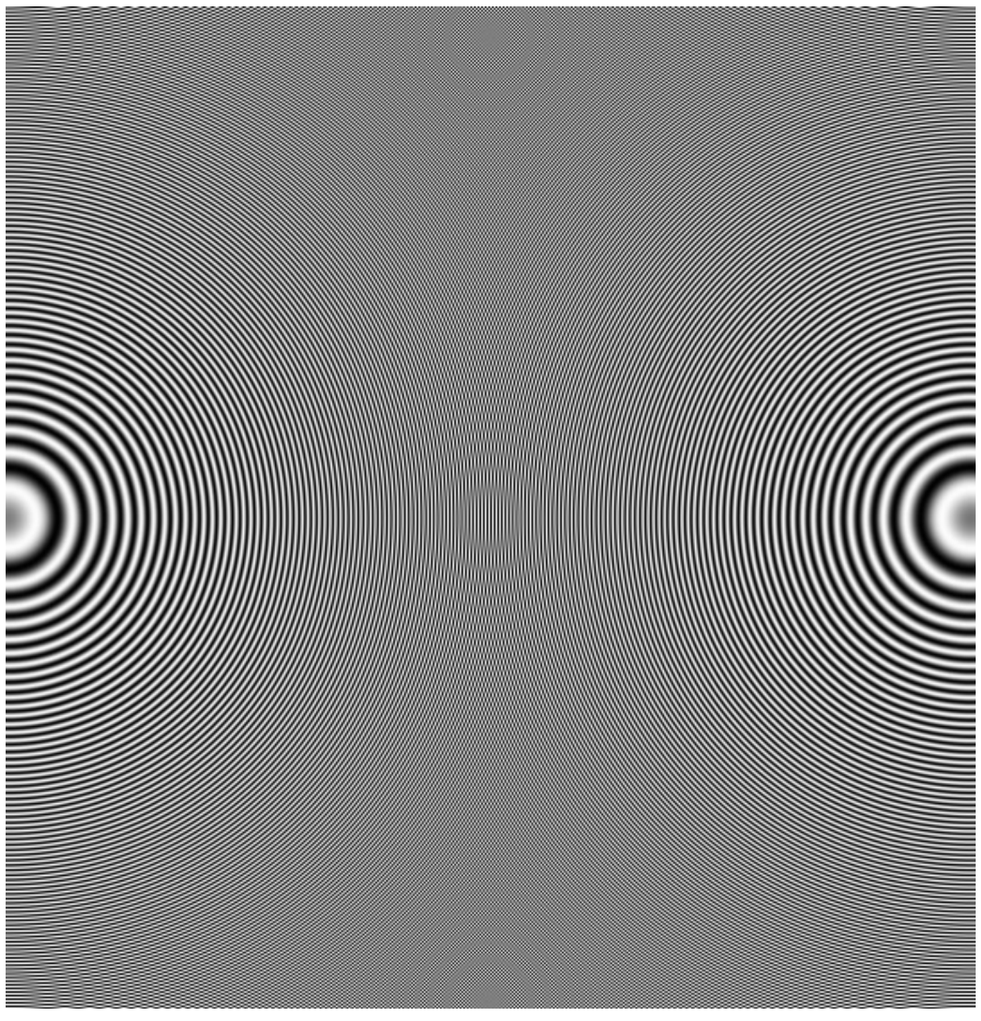
Texture sample positions in texture space (texture function is sampled at these locations)

Recall: aliasing

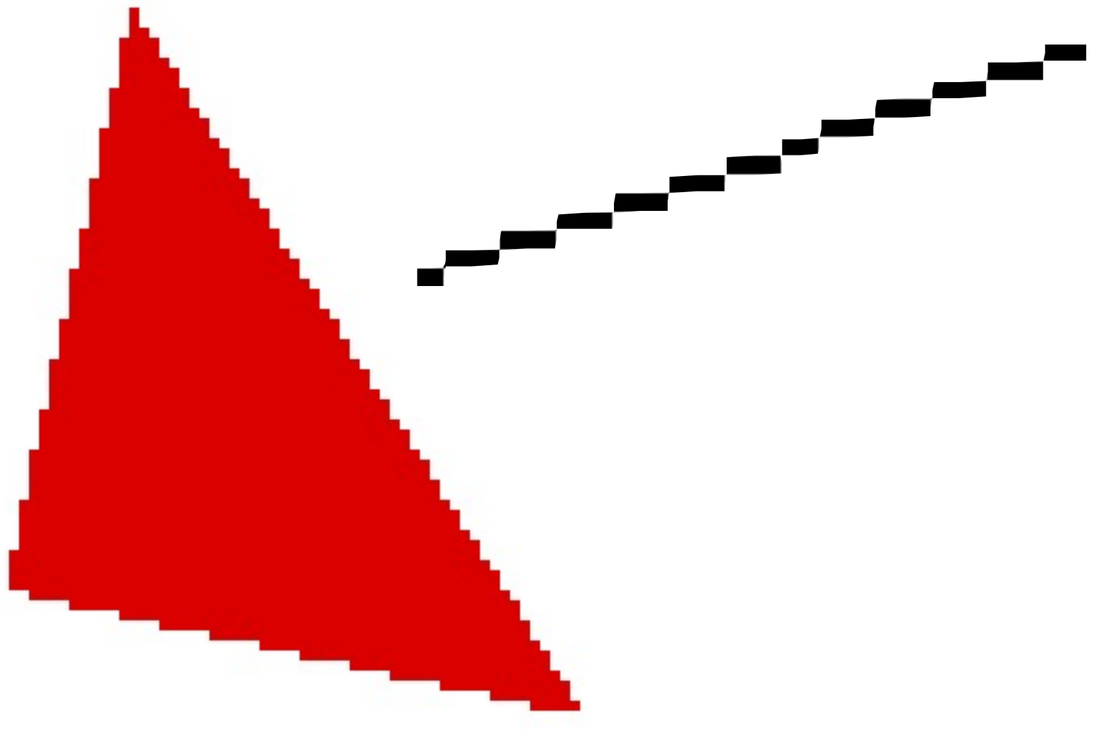
Undersampling a high-frequency signal can result in aliasing



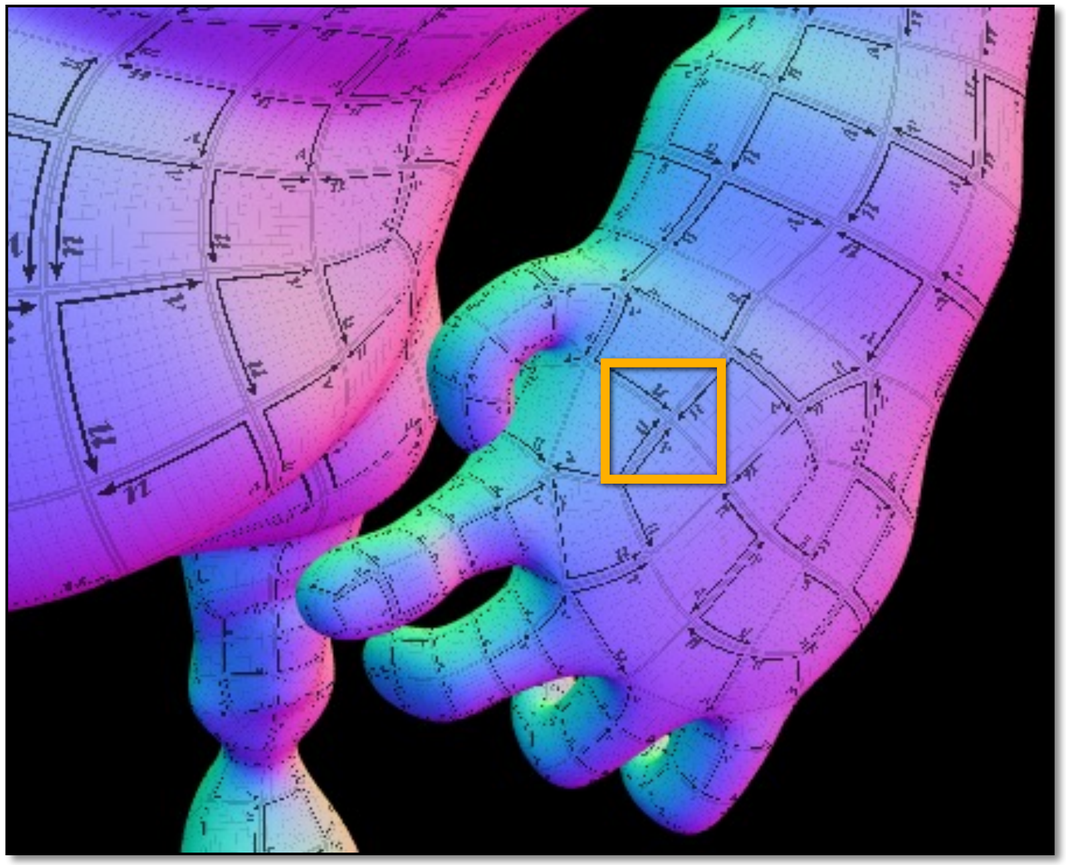
1D example



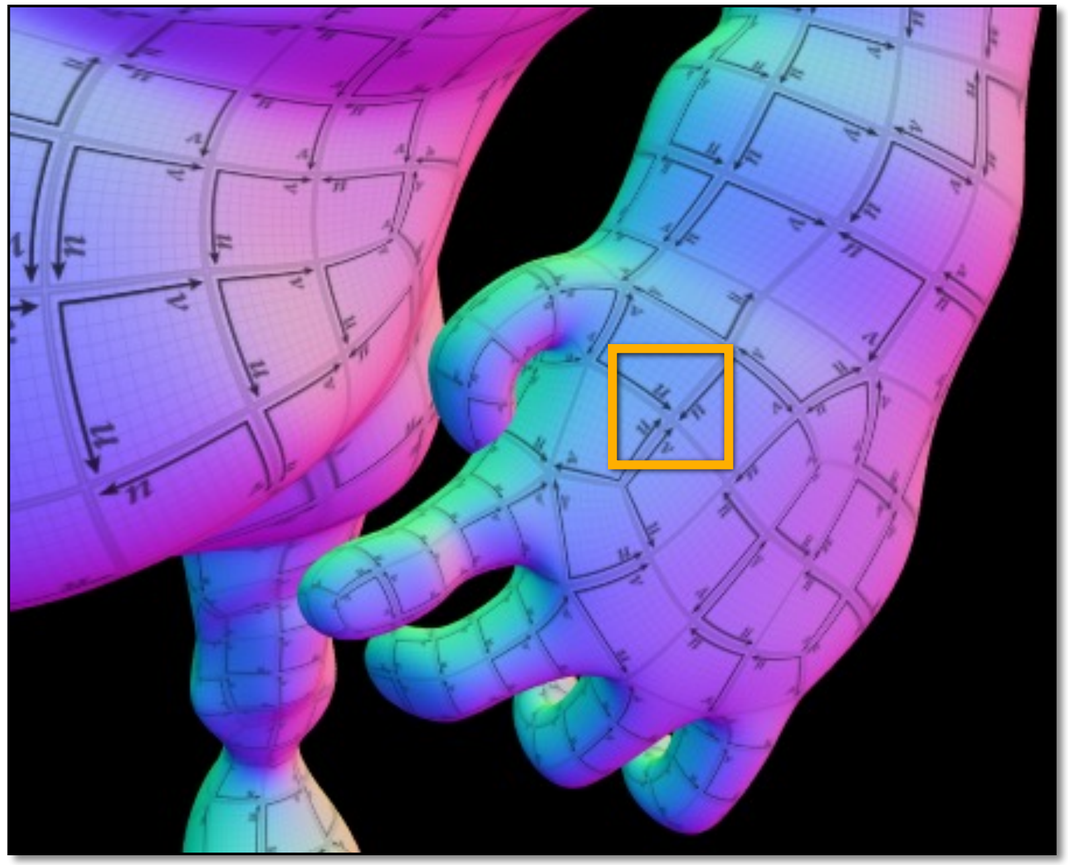
2D examples:
Moiré patterns, jaggies



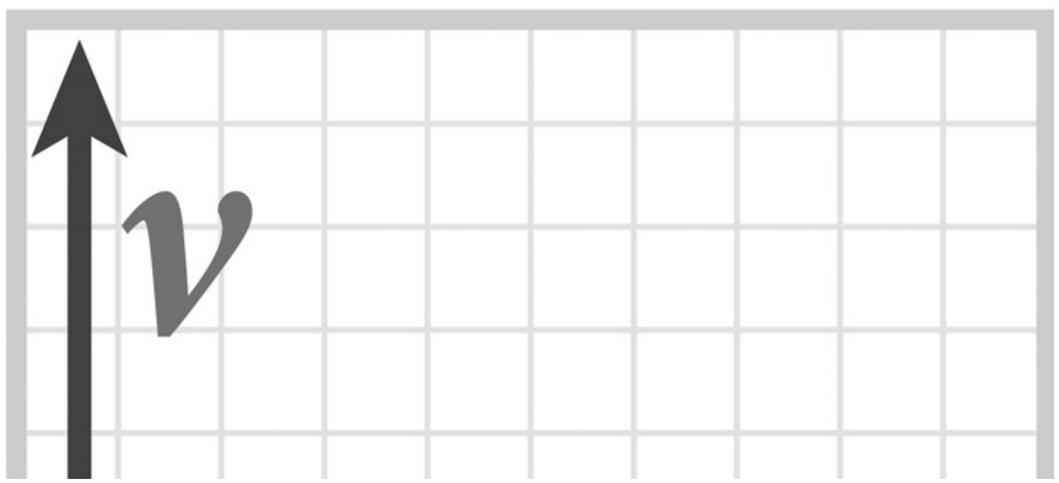
Aliasing due to undersampling texture



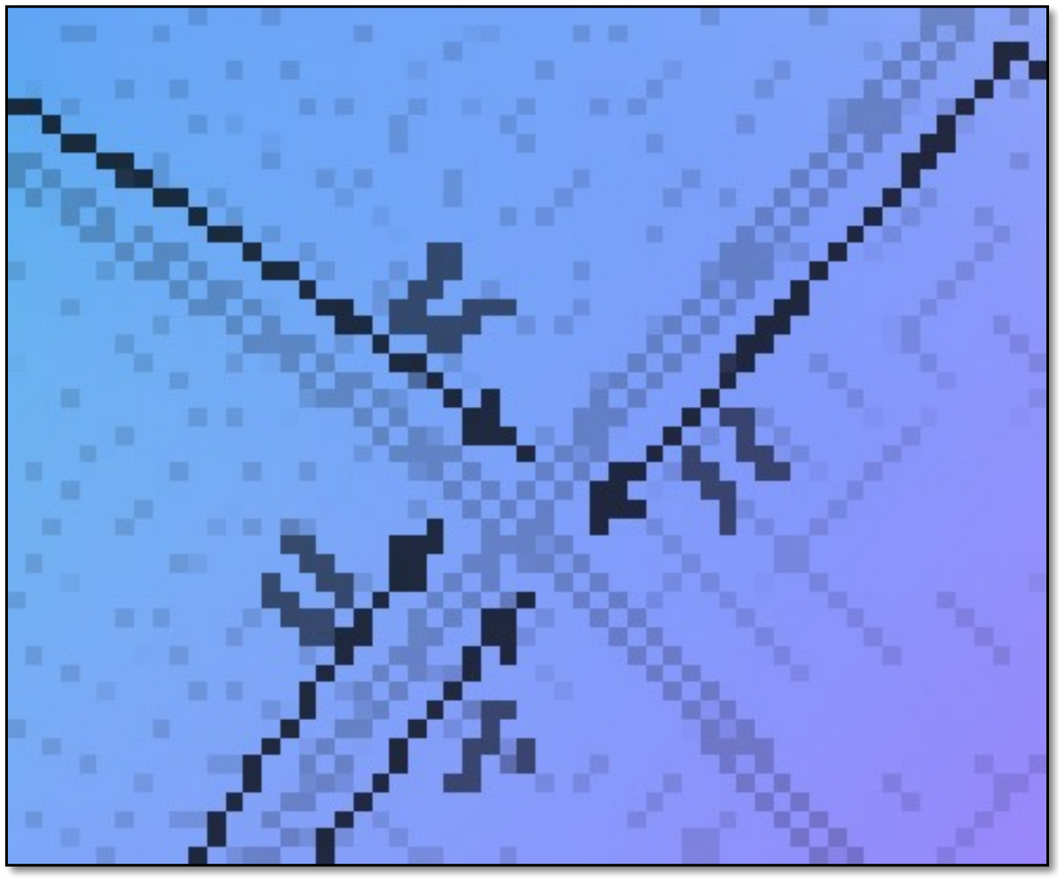
No pre-filtering of texture data
(resulting image exhibits aliasing)



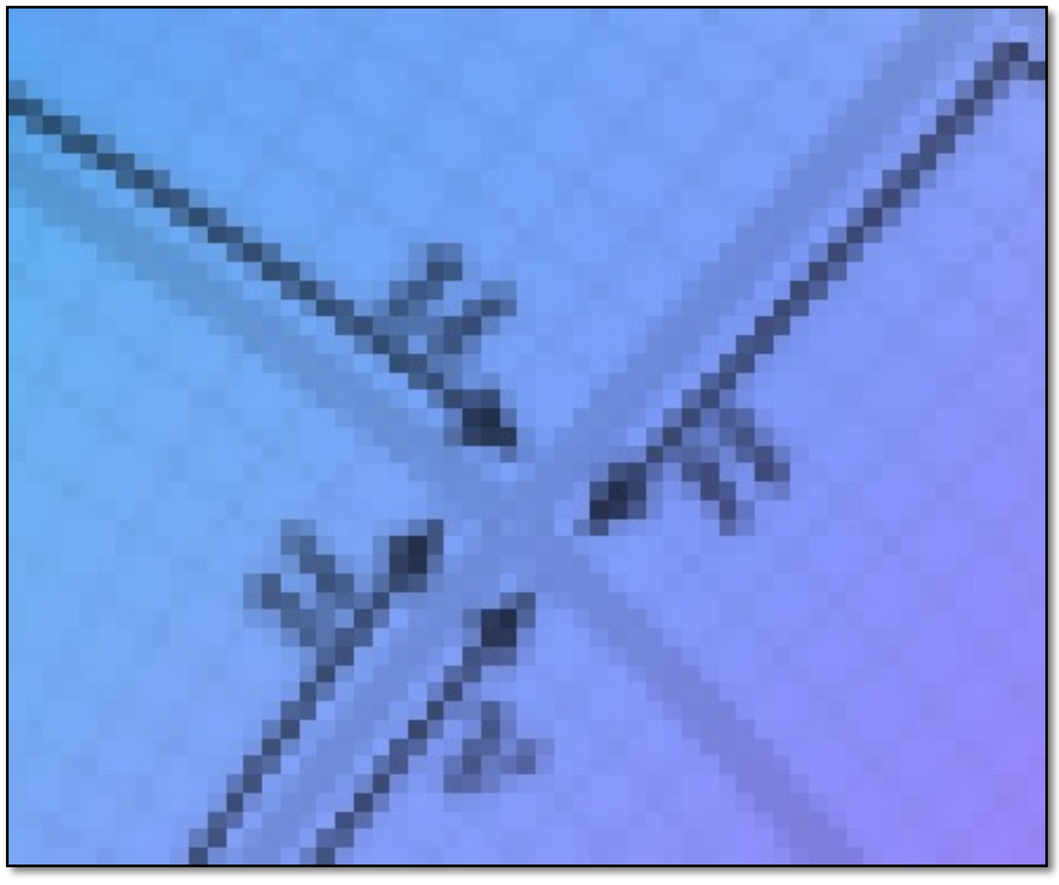
Rendering using pre-filtered texture data



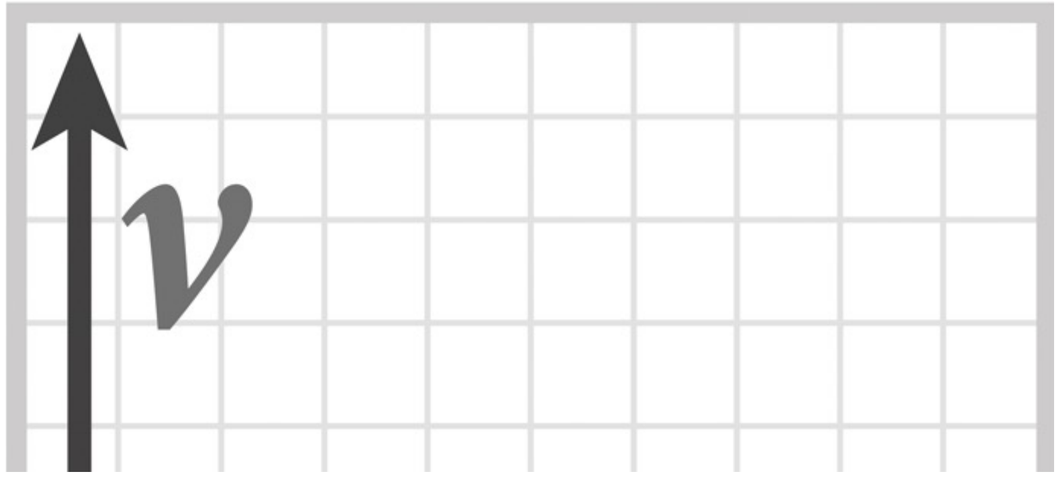
Aliasing due to undersampling (zoom)



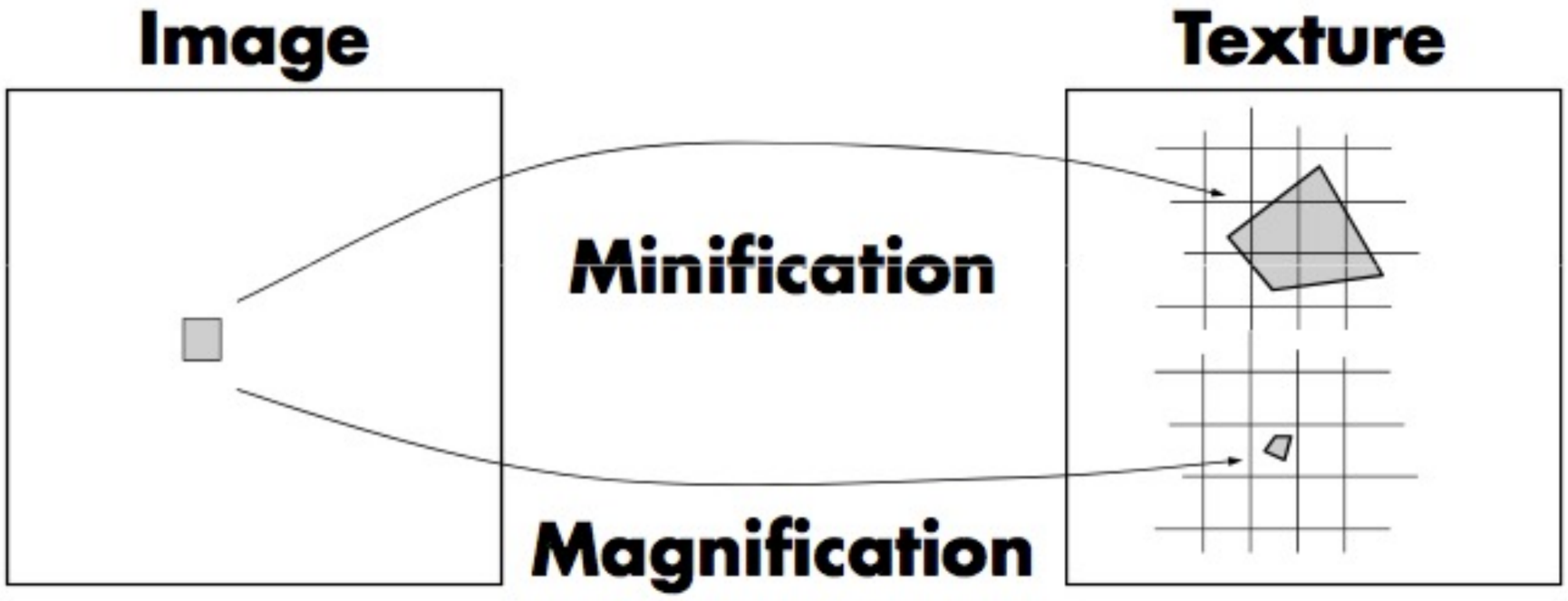
No pre-filtering of texture data
(resulting image exhibits aliasing)



Rendering using pre-filtered texture data



Filtering textures

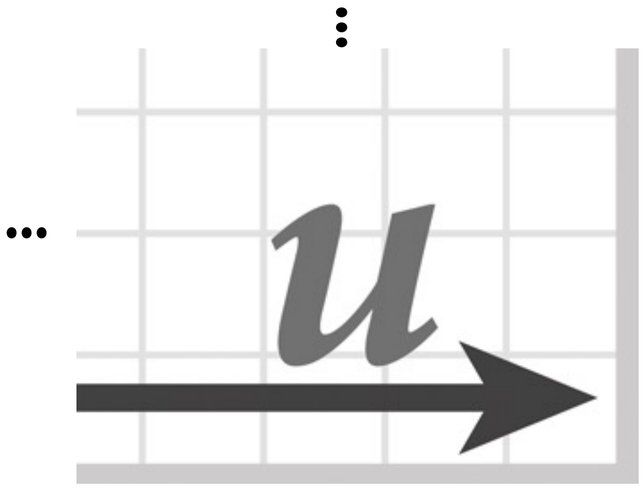


- **Minification:**
  - Area of screen pixel maps to large region of texture (filtering required -- averaging)
  - One texel corresponds to far less than a pixel on screen
  - Example: when scene object is very far away

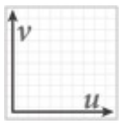
- **Magnification:**
  - Area of screen pixel maps to tiny region of texture (interpolation required)
  - One texel maps to many screen pixels
  - Example: when camera is very close to scene object (need higher resolution texture map)

Figure credit: Akeley and Hanrahan

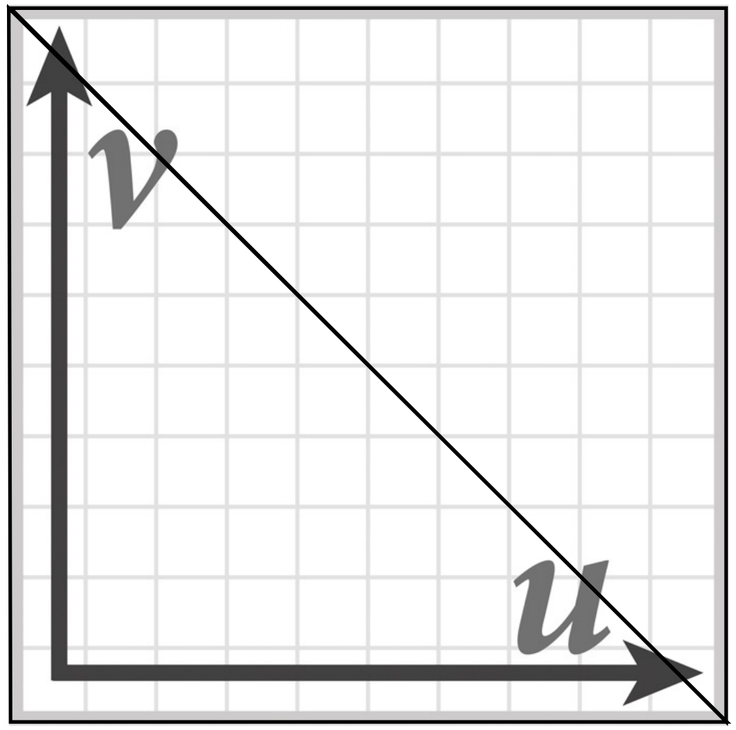
Filtering textures



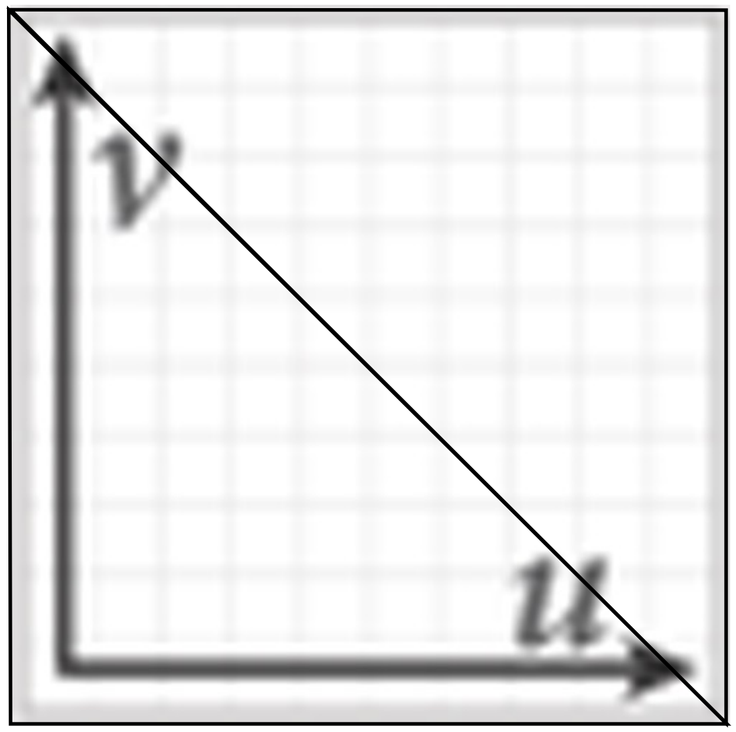
Actual texture: 700x700 image
(only a crop is shown)



Actual texture: 64x64 image

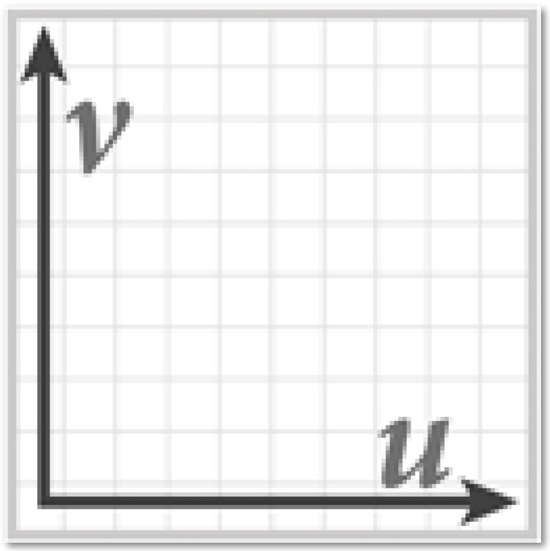


Texture minification

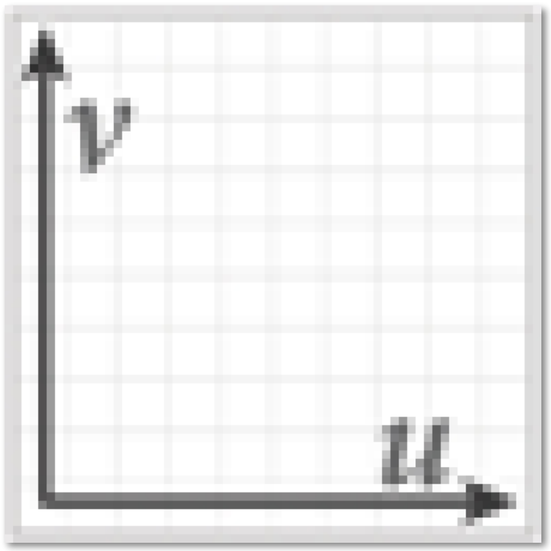


Texture magnification

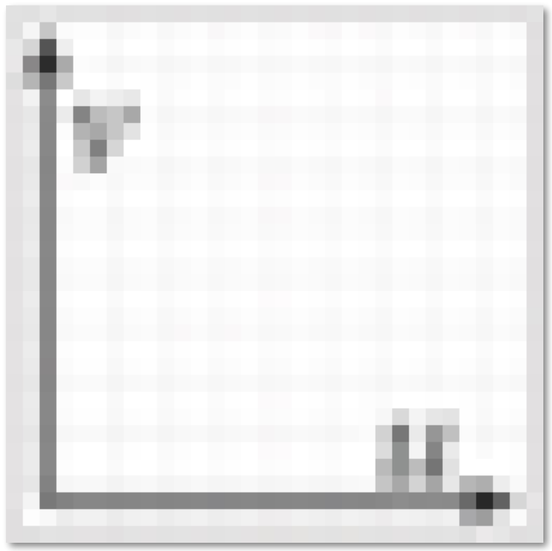
Mipmap (L. Williams 83)



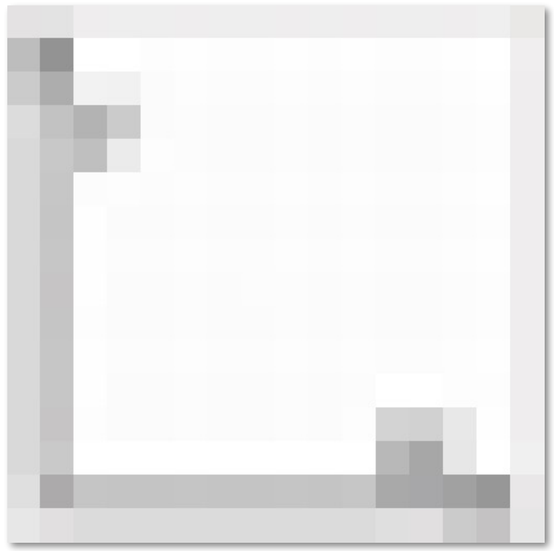
Level 0 = 128x128



Level 1 = 64x64

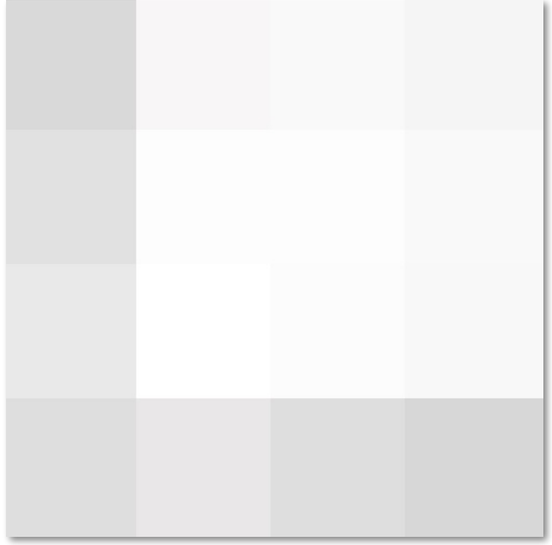


Level 2 = 32x32

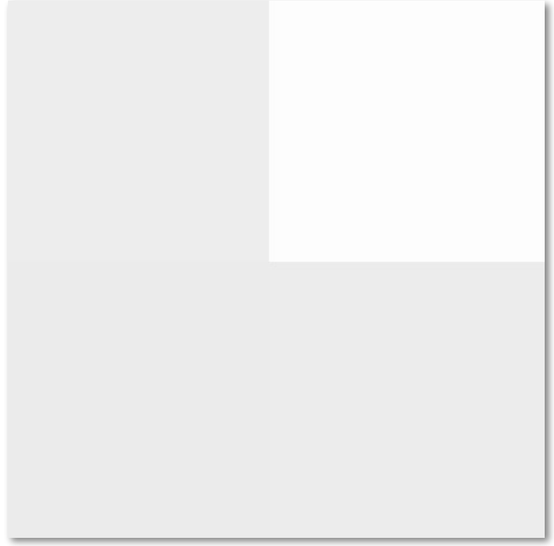


Level 3 = 16x16

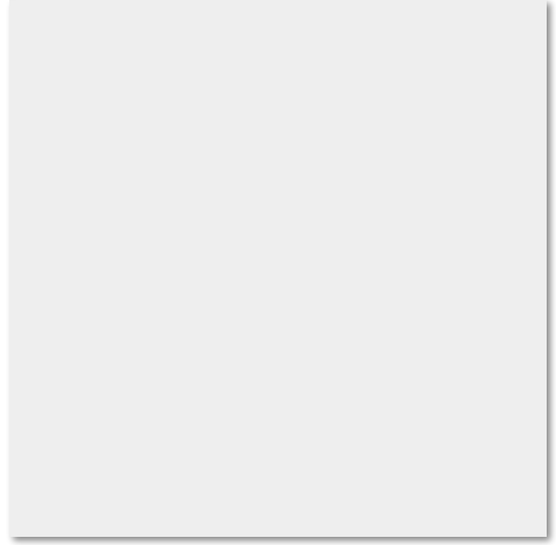
Level 4 = 8x8



Level 5 = 4x4



Level 6 = 2x2



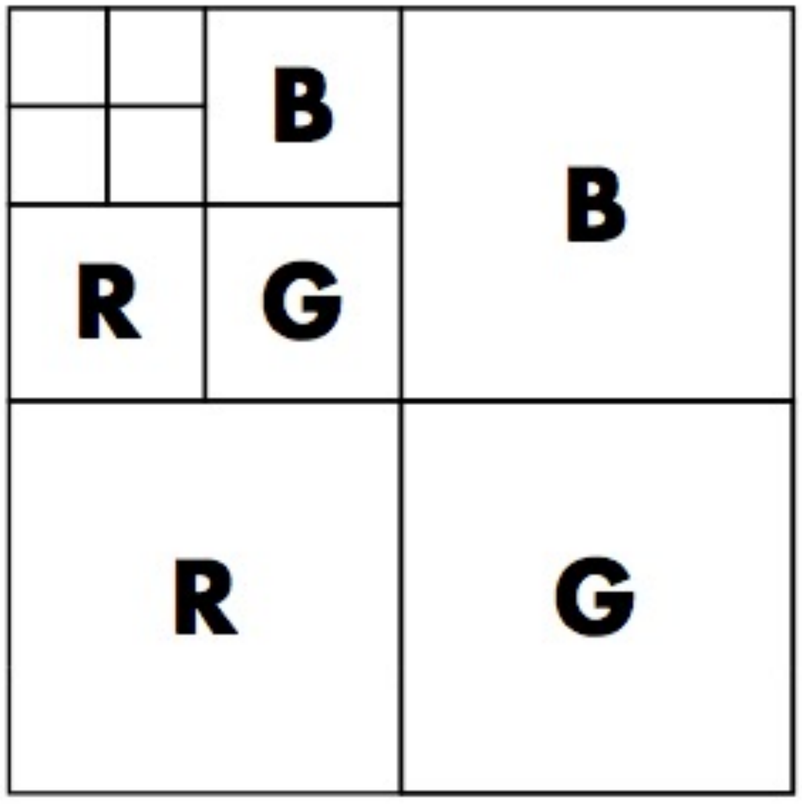
Level 7 = 1x1

Idea: prefilter texture data to remove high frequencies

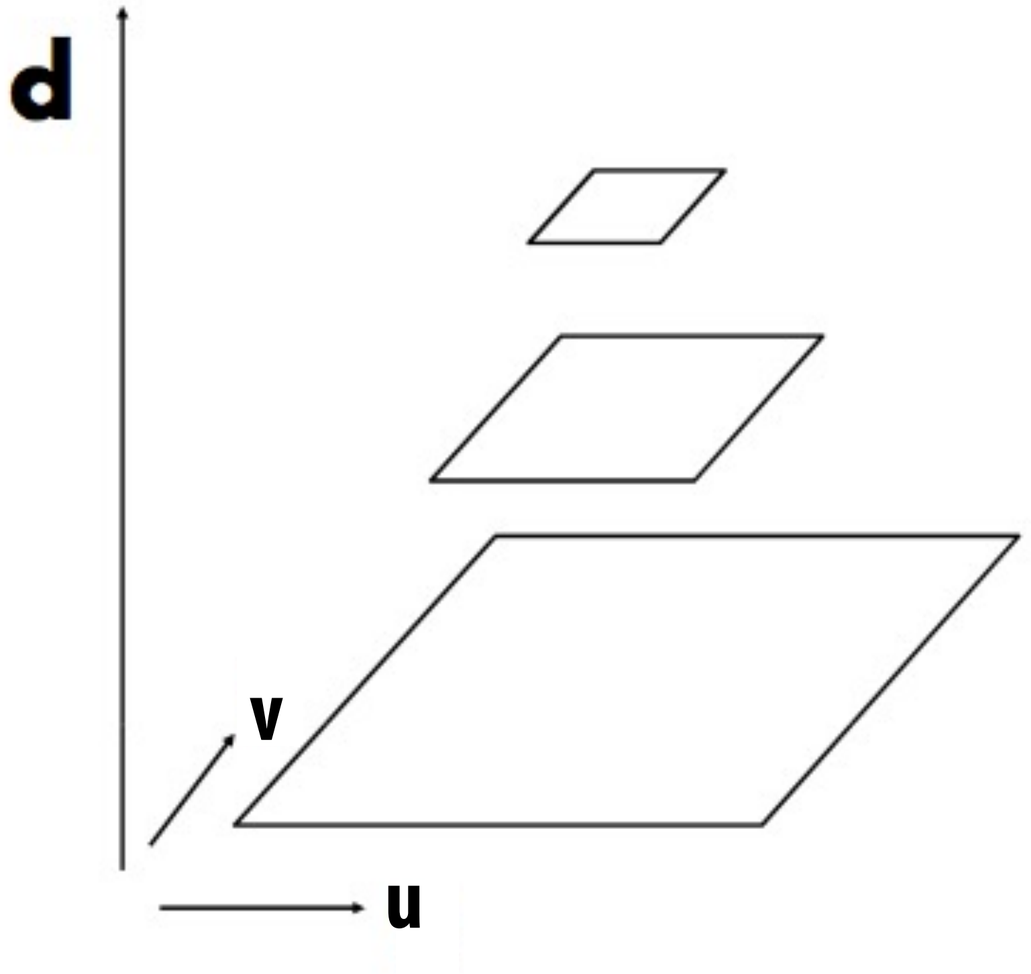
Texels at higher levels store integral of the texture function over a region of texture space (downsampled images)

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

Mipmap (L. Williams 83)



Williams' original proposed mip-map layout

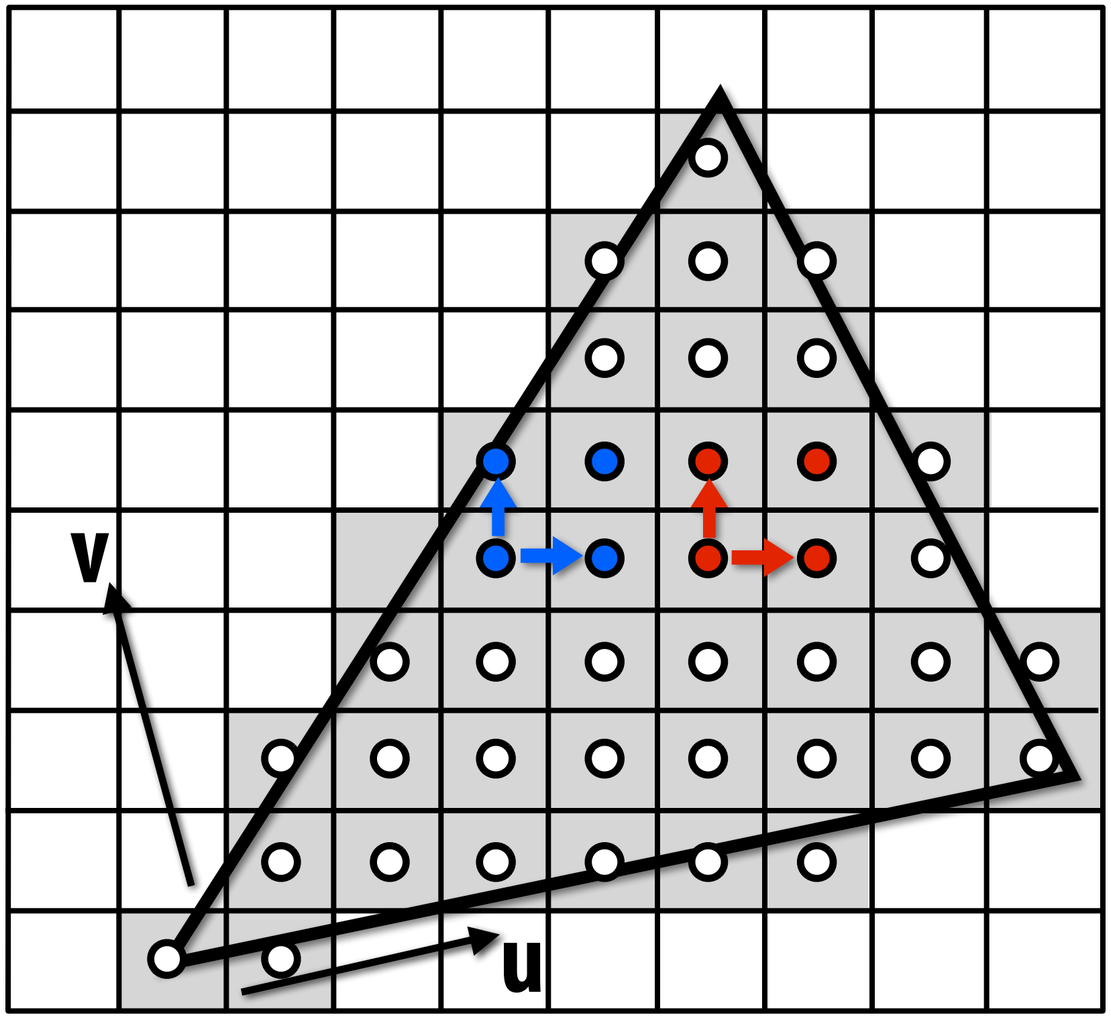


"Mip hierarchy" level = *d*

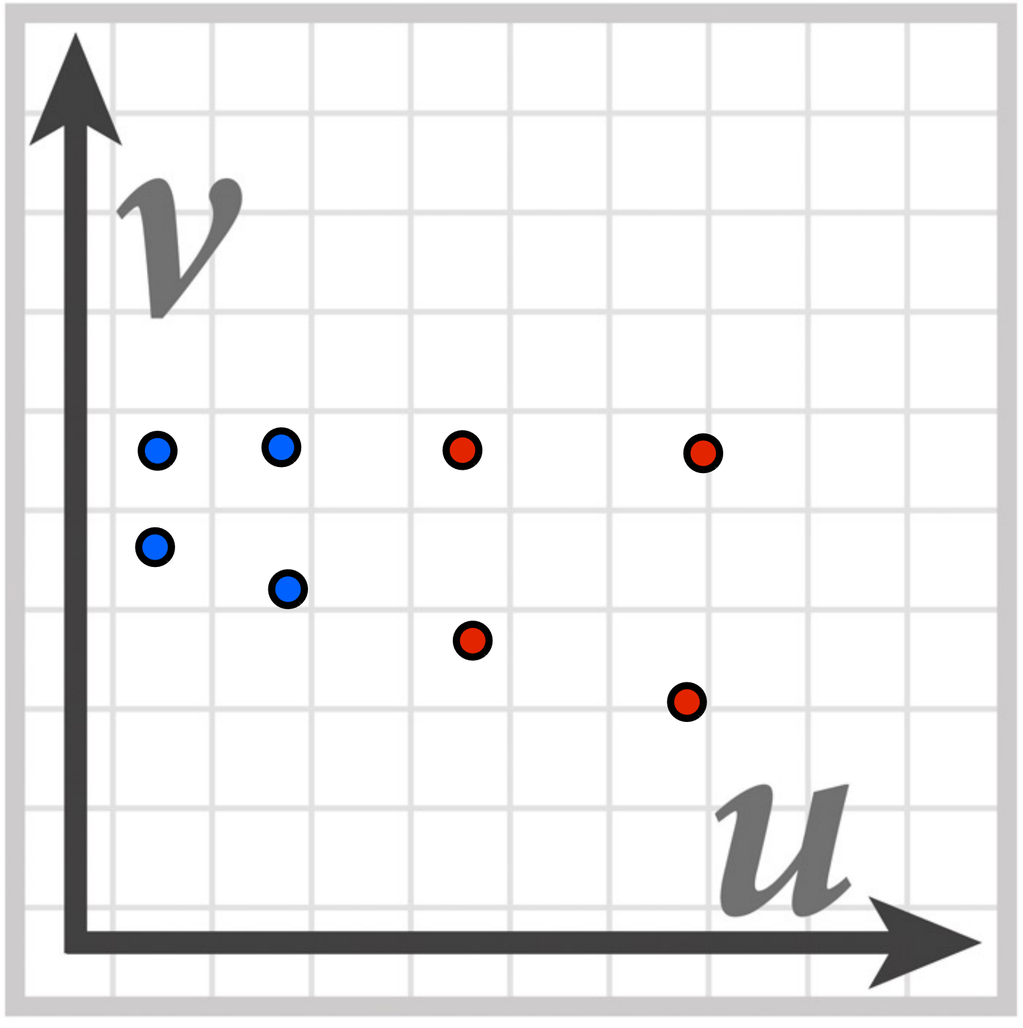
What is the storage overhead of a mipmap?

Computing *d*

Compute differences between texture coordinate values of neighboring screen samples



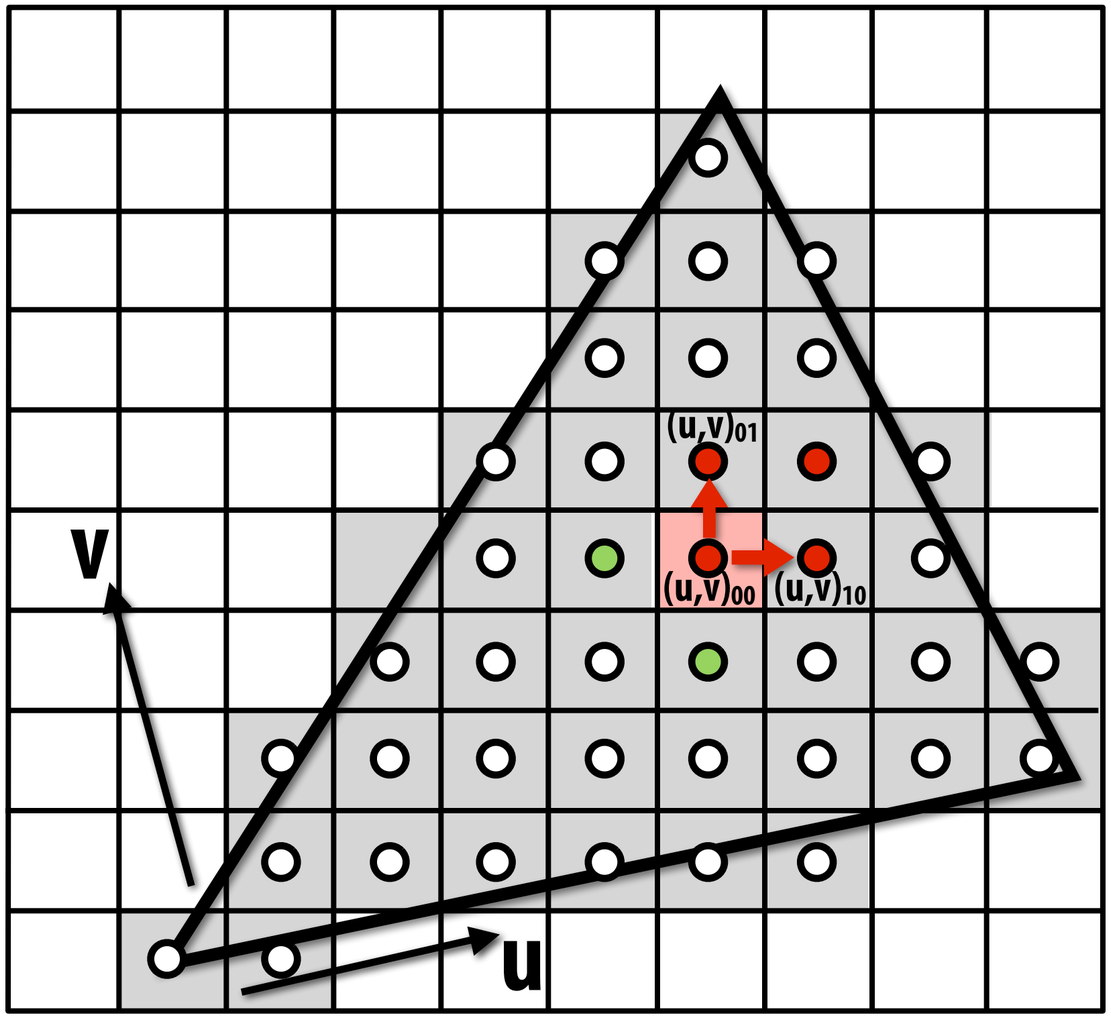
Screen space



Texture space

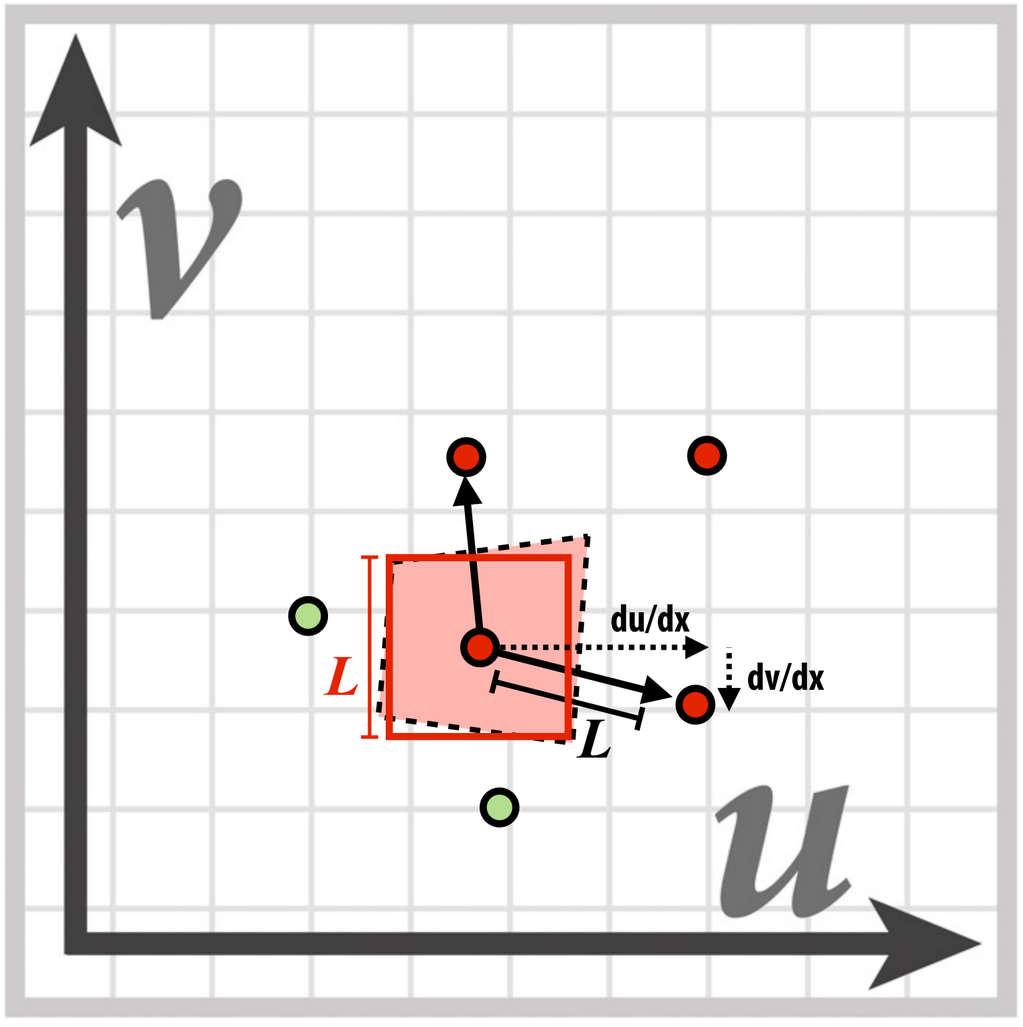
Computing *d*

Compute differences between texture coordinate values of neighboring fragments



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

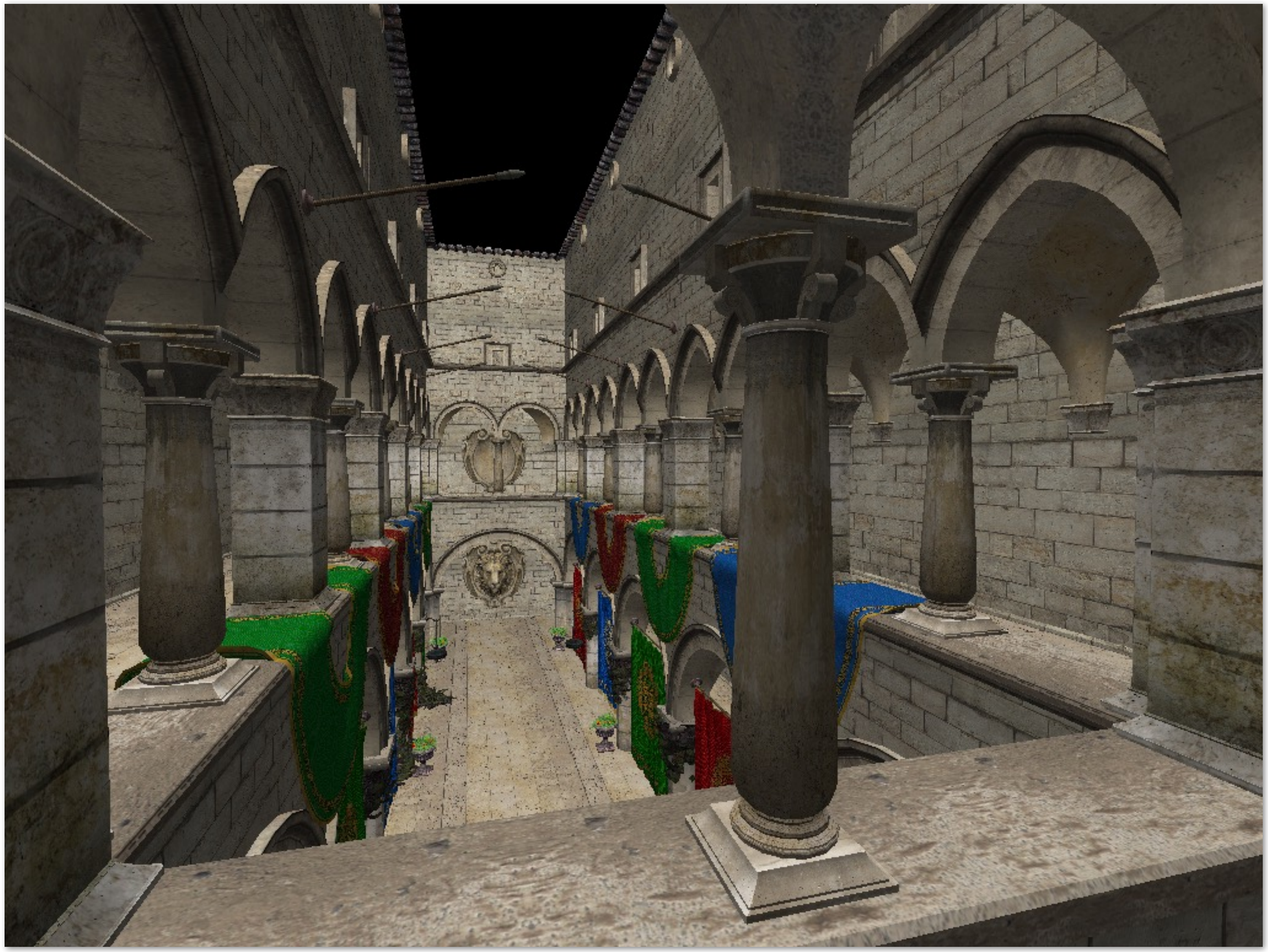
$du/dy = u_{01}-u_{00}$ $dv/dy = v_{01}-v_{00}$



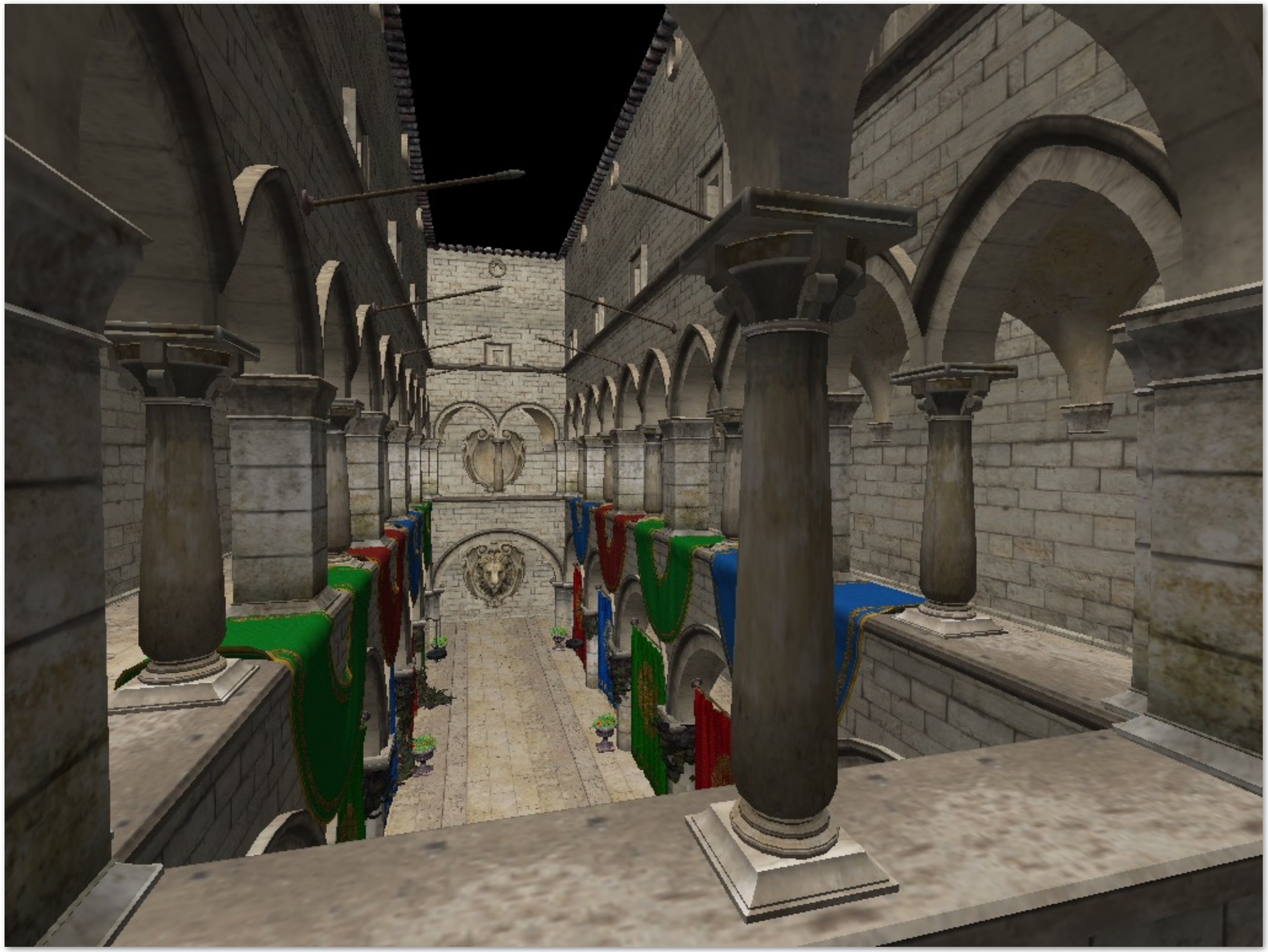
$$L = \max\left(\sqrt{\left(\frac{du}{dx}\right)^2 + \left(\frac{dv}{dx}\right)^2}, \sqrt{\left(\frac{du}{dy}\right)^2 + \left(\frac{dv}{dy}\right)^2}\right)$$

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

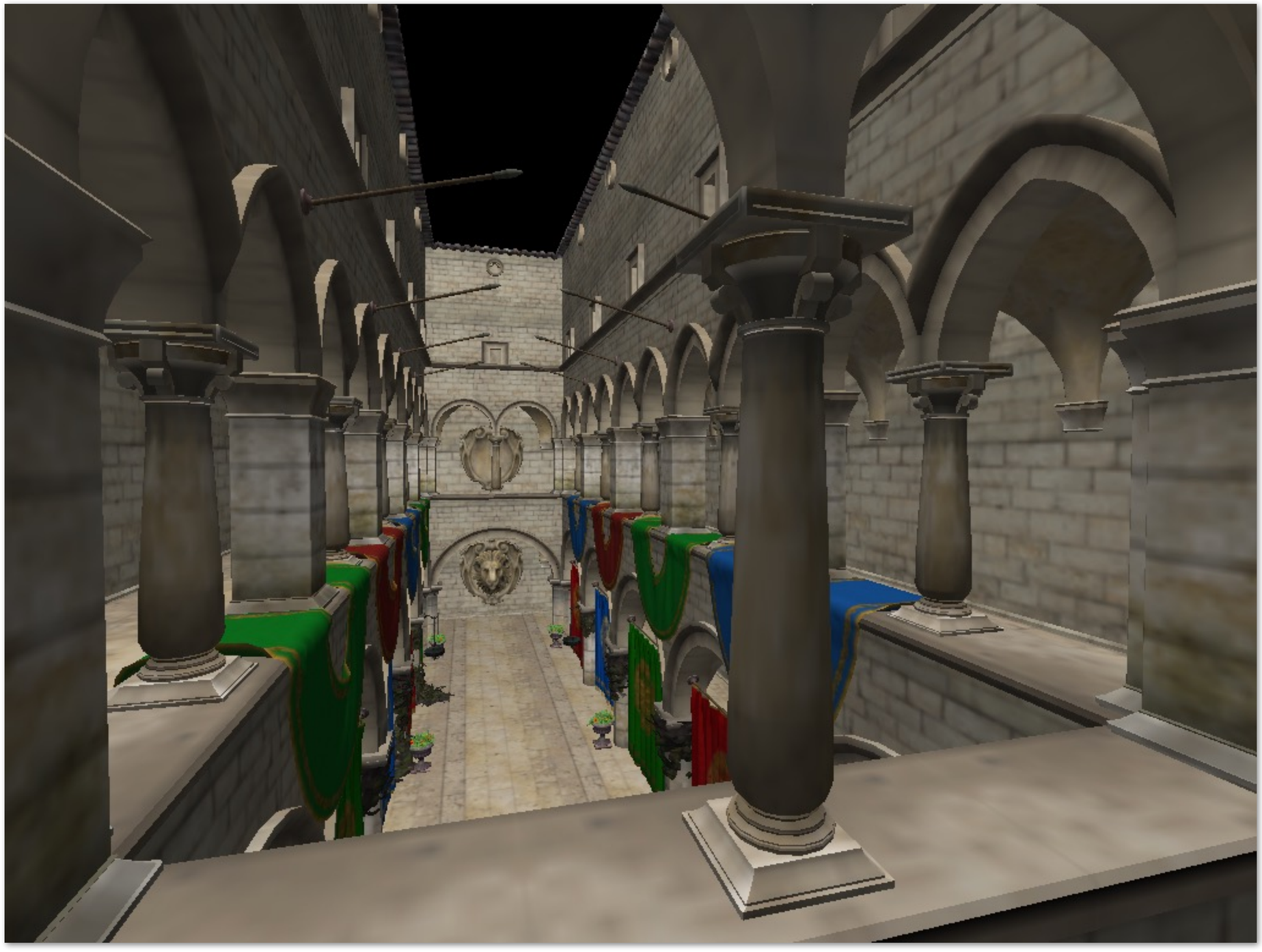
Sponza (bilinear resampling at level 0)



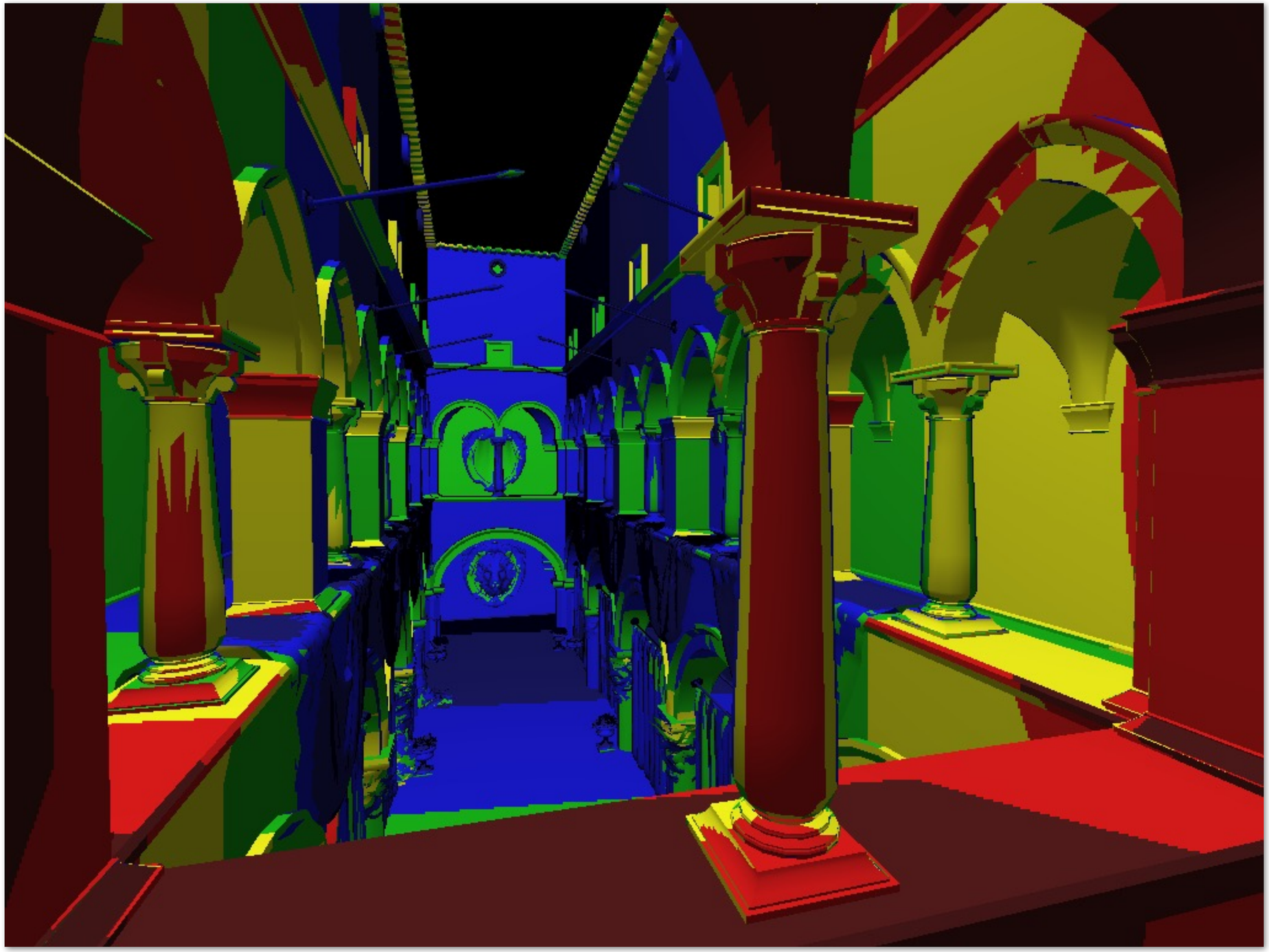
Sponza (bilinear resampling at level 2)



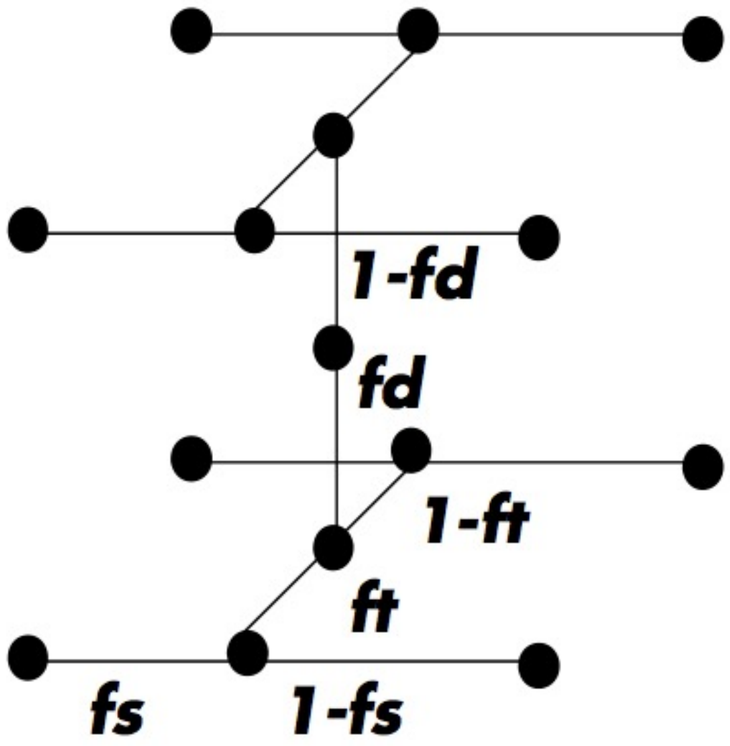
Sponza (bilinear resampling at level 4)



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



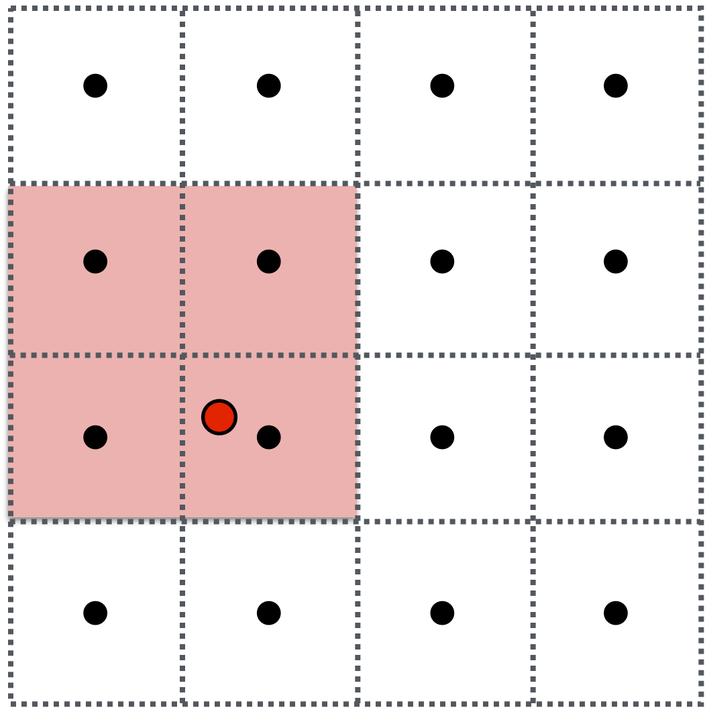
“Tri-linear” filtering



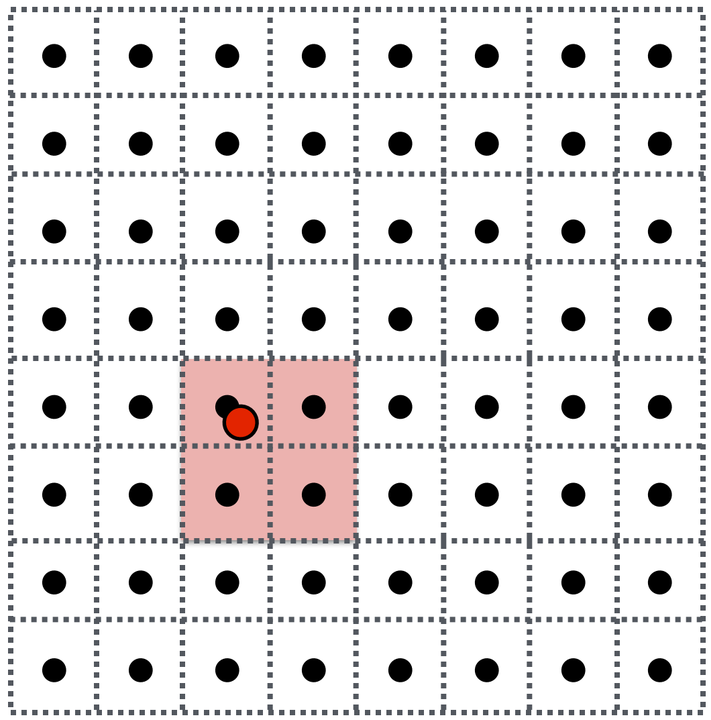
$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

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

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



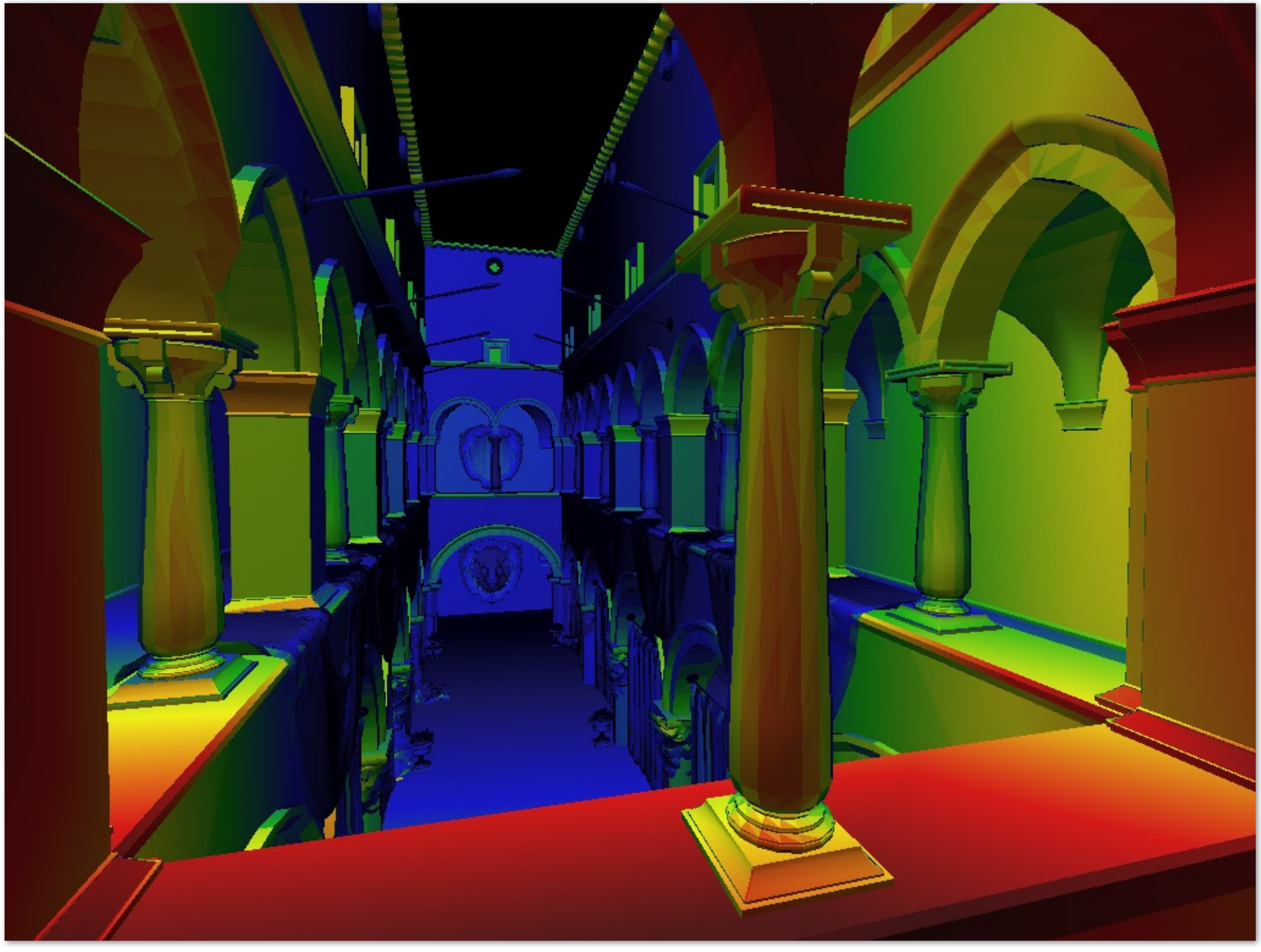
mip-map texels: level d+1



mip-map texels: level d

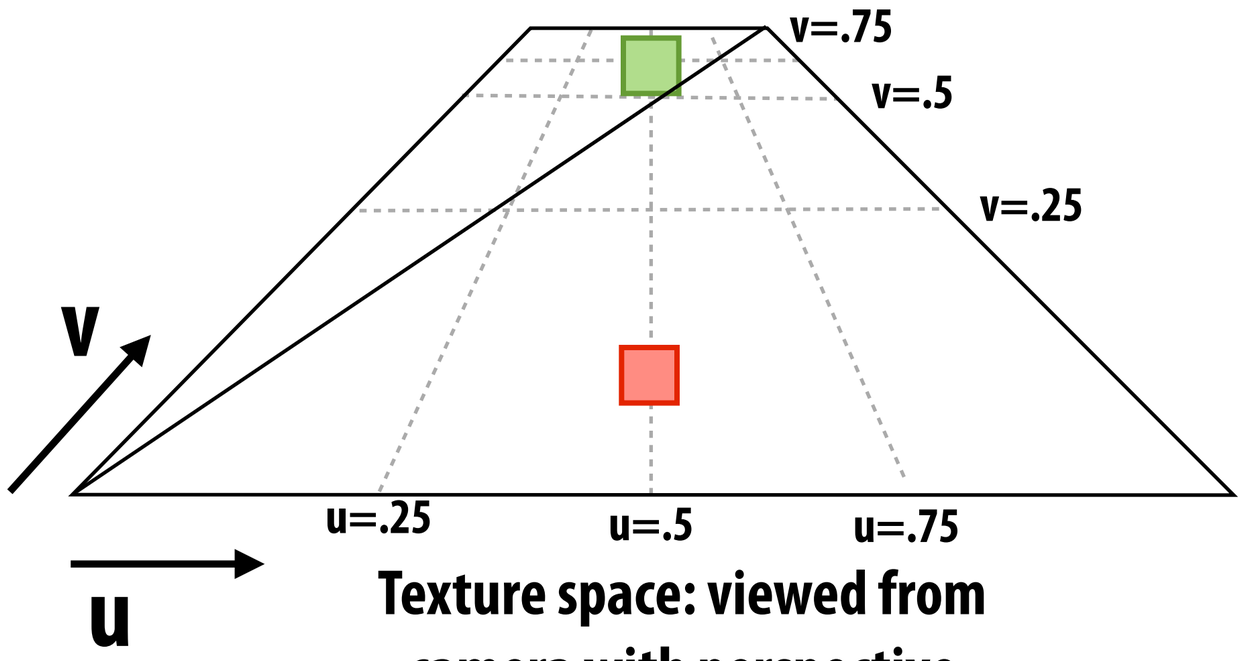
Figure credit: Akeley and Hanrahan

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

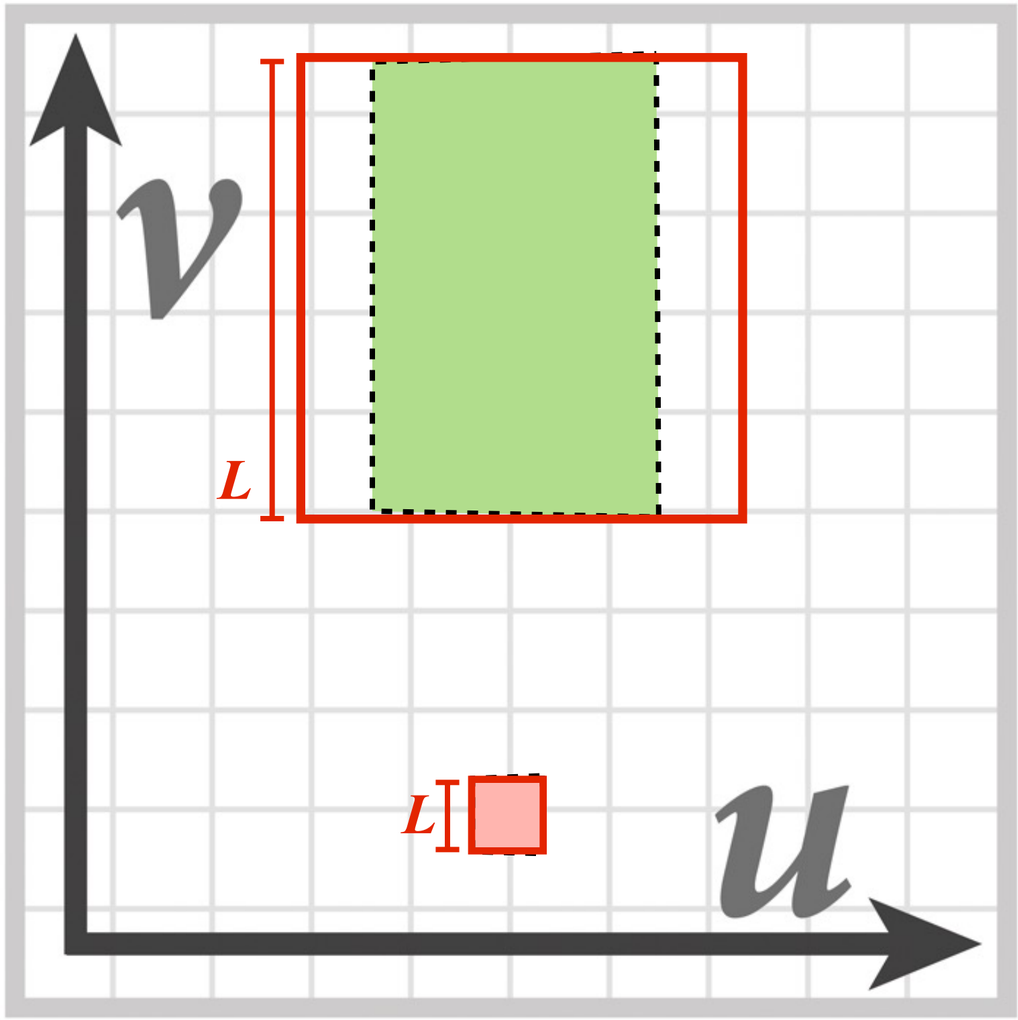


Pixel area may not map to isotropic region in texture

Proper filtering requires anisotropic filter footprint



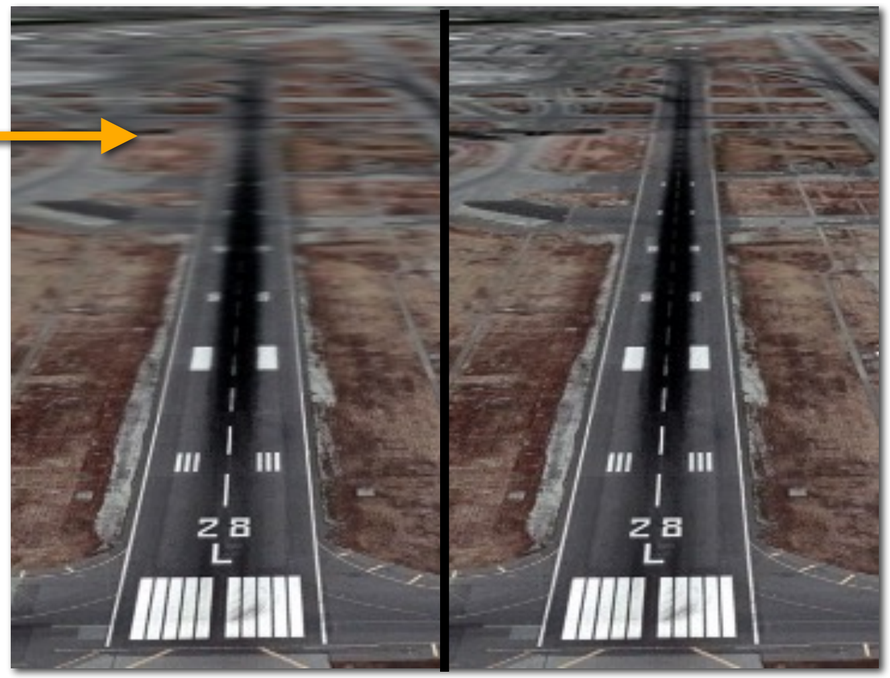
Texture space: viewed from camera with perspective



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

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

Overblurring in *u* direction



Trilinear (Isotropic) Filtering

Anisotropic Filtering

Principle of texture thrift

[Peachey 90]

Given a scene consisting of textured 3D surfaces, the amount of texture information minimally required to render an image of the scene is proportional to the resolution of the image and is independent of the number of surfaces and the size of the textures.

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 *d*)
  - Constant number of texels accessed (independent of *d*)

- **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

Summary: a texture sampling operation

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 *d*
4. Convert normalized texture coordinate (u,v) to texture coordinates texel_u, texel_v
5. Compute required texels in window of filter
6. Load required texels (need eight texels for trilinear)
7. Perform tri-linear interpolation according to (texel_u, texel_v, d)

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

All 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