Perspective Projection and Texture Mapping

Computer Graphics
CMU 15-462/15-662, Fall 2016

Transforms: moving to 3D (and 3D-H)

Represent 3D transforms as 3x3 matrices and 3D-H transforms as 4x4 matrices

Scale:

3D

$$\mathbf{S_s} = \begin{bmatrix} \mathbf{S}_x & 0 & 0 \\ 0 & \mathbf{S}_y & 0 \\ 0 & 0 & \mathbf{S}_z \end{bmatrix}$$

3D-H

$$\mathbf{S_s} = \begin{bmatrix} \mathbf{S}_x & 0 & 0 & 0 \\ 0 & \mathbf{S}_y & 0 & 0 \\ 0 & 0 & \mathbf{S}_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Shear (in x, based on y, z position):

$$\mathbf{H}_{x,\mathbf{d}} = \begin{bmatrix} 1 & \mathbf{d}_y & \mathbf{d}_z \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

$$\mathbf{H}_{x,\mathbf{d}} = \begin{bmatrix} 1 & \mathbf{d}_y & \mathbf{d}_z & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

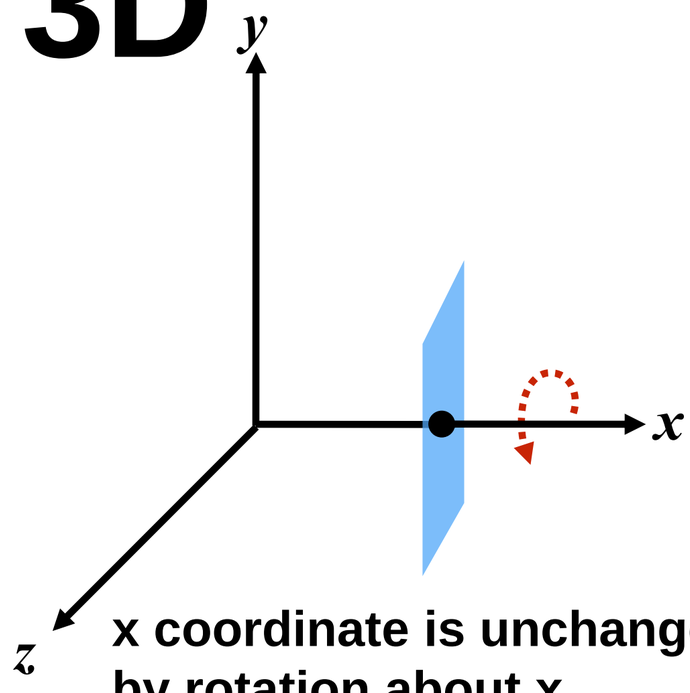
Translate:

$$\mathbf{T_b} = \begin{bmatrix} 1 & 0 & 0 & \mathbf{b}_x \\ 0 & 1 & 0 & \mathbf{b}_y \\ 0 & 0 & 1 & \mathbf{b}_z \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

Rotations in 3D

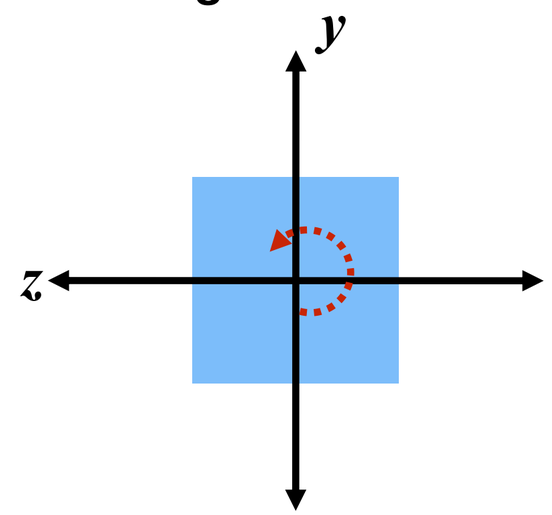
Rotation about x axis:

$$\mathbf{R}_{x,\theta} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos\theta & -\sin\theta \\ 0 & \sin\theta & \cos\theta \end{bmatrix}$$



x coordinate is unchanged by rotation about x

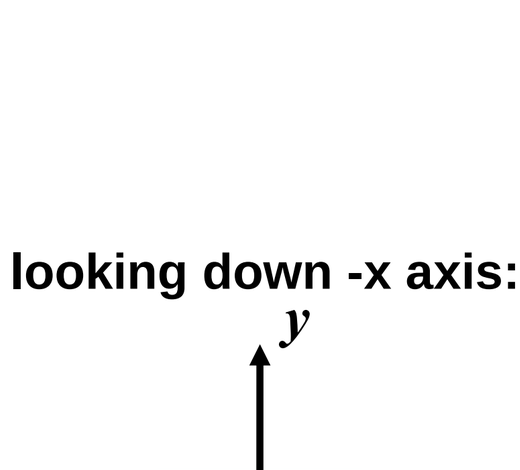
View looking down -x axis:



Rotation about y axis:

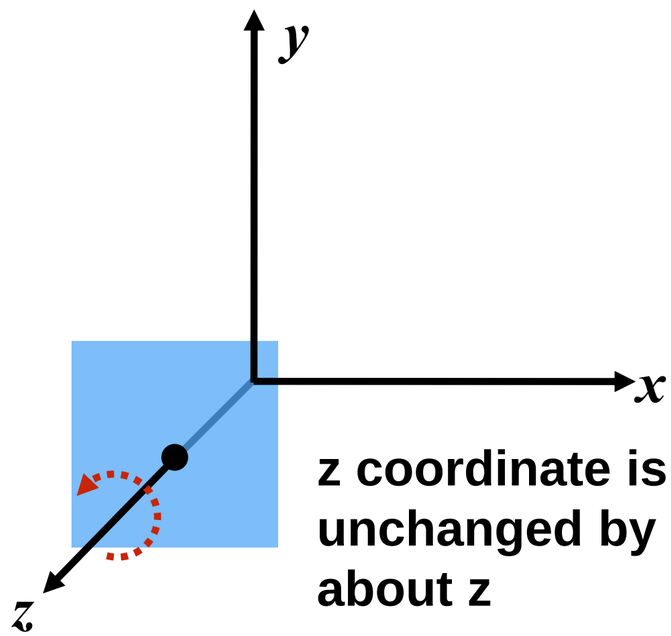
$$\mathbf{R}_{y,\theta} = \begin{bmatrix} \cos\theta & 0 & \sin\theta \\ 0 & 1 & 0 \\ -\sin\theta & 0 & \cos\theta \end{bmatrix}$$

View looking down -y axis:



Rotation about z axis:

$$\mathbf{R}_{z,\theta} = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



z coordinate is unchanged by rotation about z

Rotation about an arbitrary axis

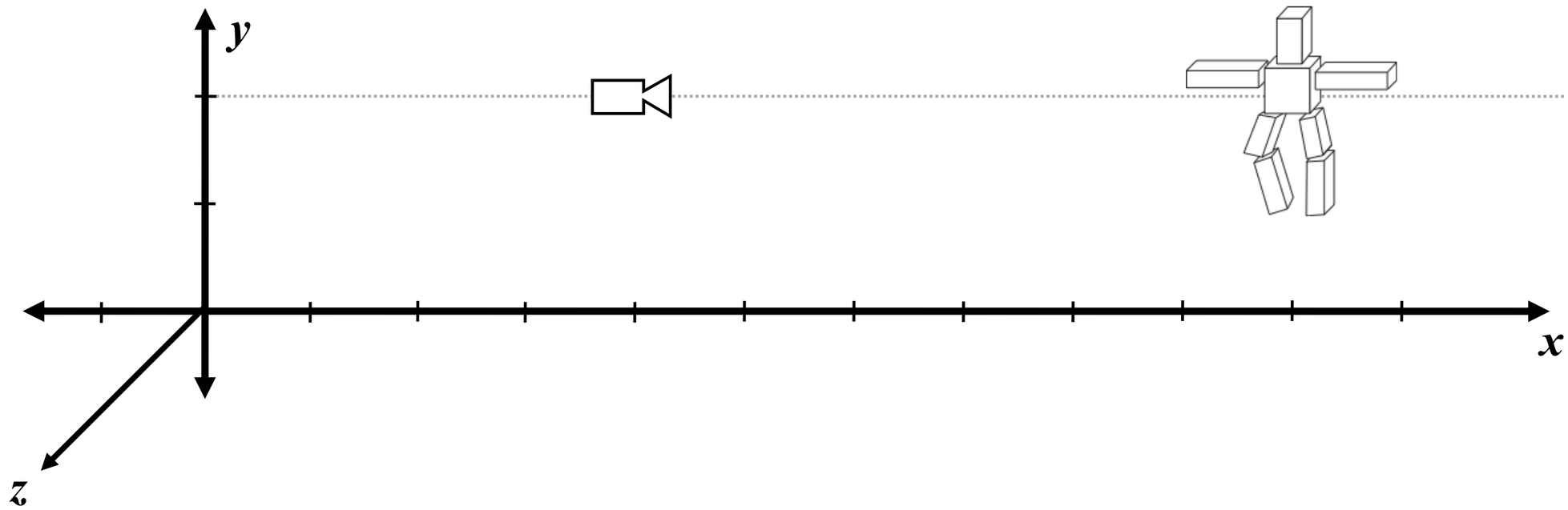
Q: Do you know how to derive it?

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

Just memorize this matrix! :-)

Review: simple camera transform

- Consider object in world at (10, 2, 0)
- Consider camera at (4, 2, 0), looking down x axis



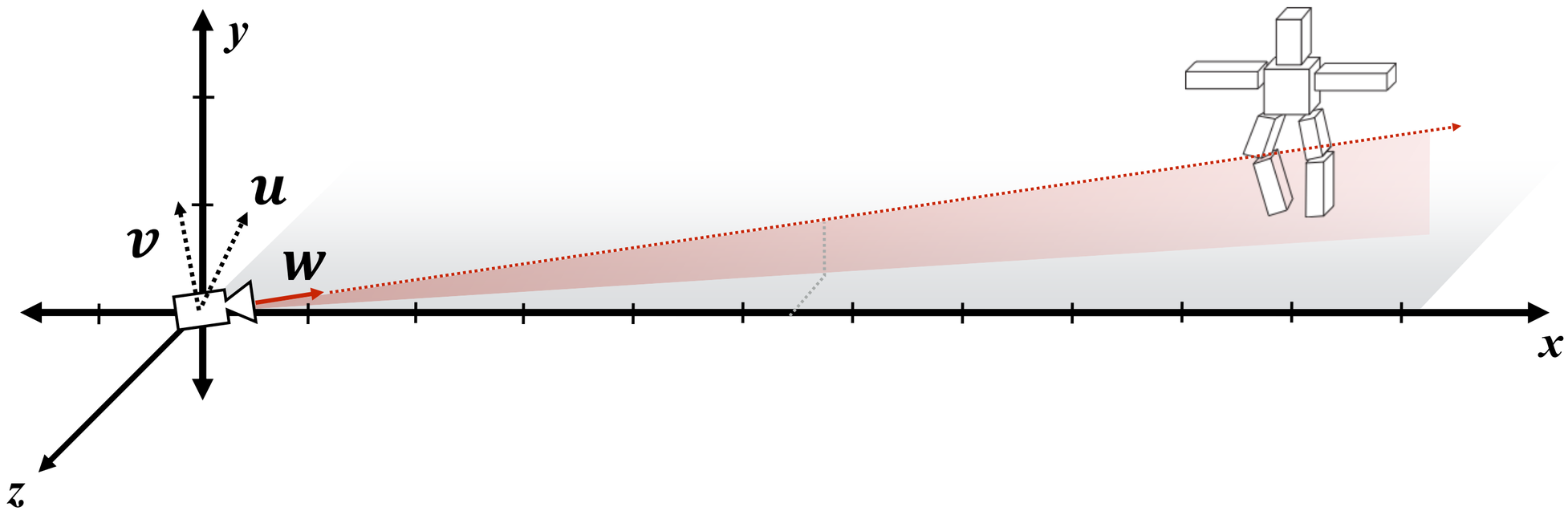
- How do you compute the transform from world to the camera coordinate system* (camera at the origin, looking down z axis)?
- Translating object vertex positions by (-4, -2, 0) yields position relative to camera.
- Rotation about y axis by $\pi/2$ gives position of object in coordinate system where camera's view direction is aligned with the -z axis

* The convenience of the camera coordinate system will become clear soon!

Camera with arbitrary orientation

Consider camera looking in direction w

What transform places the object in the camera coordinate system?



Form orthonormal basis around w : (see u and v)

Compute transformation that maps u to x-axis, v to y-axis, and w to -z axis

Answer: Consider rotation matrix:

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

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

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

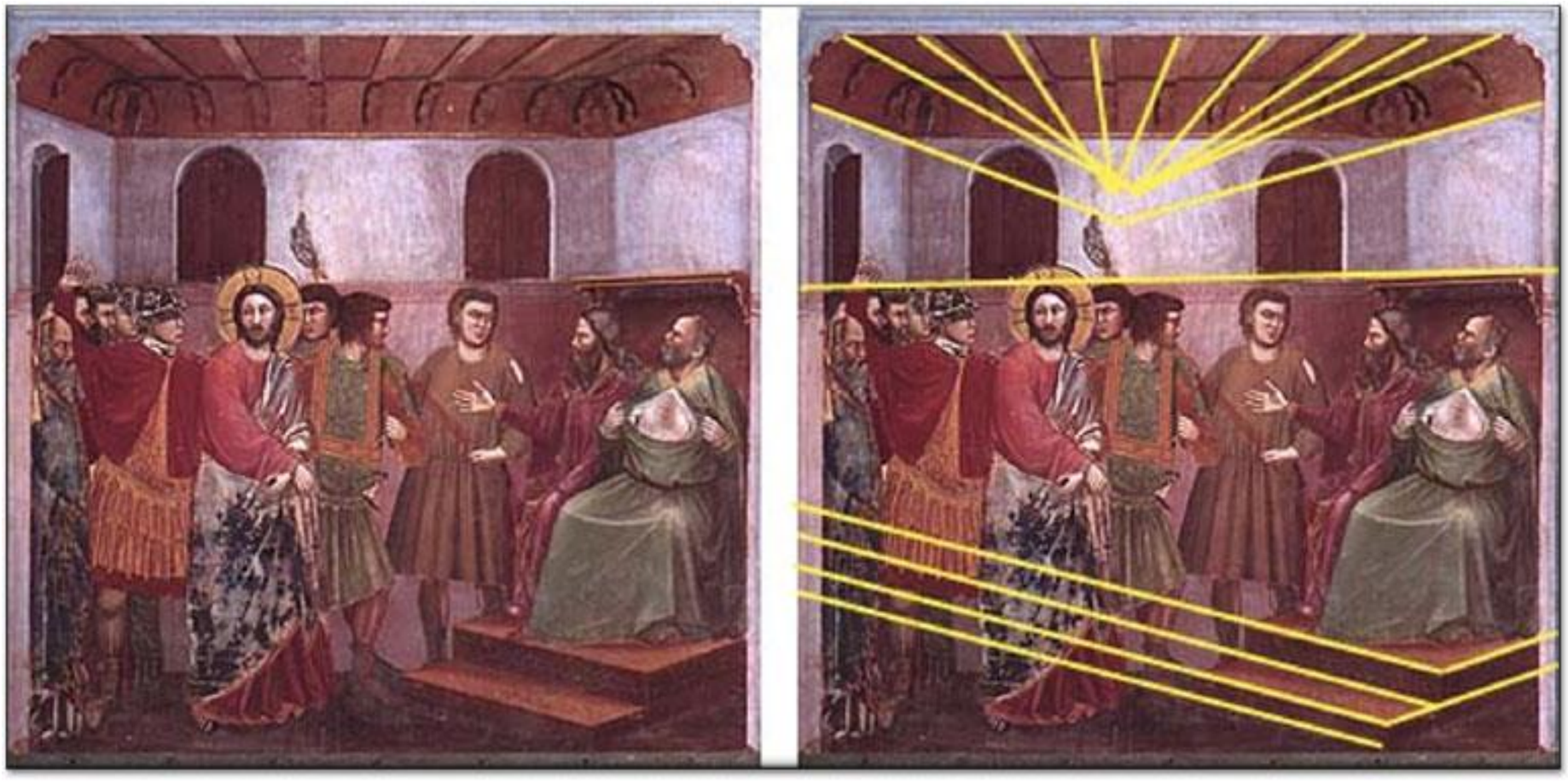
$$\mathbf{R}^T\mathbf{w} = \begin{bmatrix} \mathbf{u}\cdot\mathbf{w} & \mathbf{v}\cdot\mathbf{w} & -\mathbf{w}\cdot\mathbf{w} \end{bmatrix}^T = \begin{bmatrix} 0 & 0 & -1 \end{bmatrix}^T$$

Perspective projection

Early paintings: incorrect perspective

‘Jesus Before the Caïf’, by Giotto (1305)

Early paintings: incorrect perspective



‘Jesus Before the Caïf’, by Giotto (1305)

Geometrically correct perspective in art

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

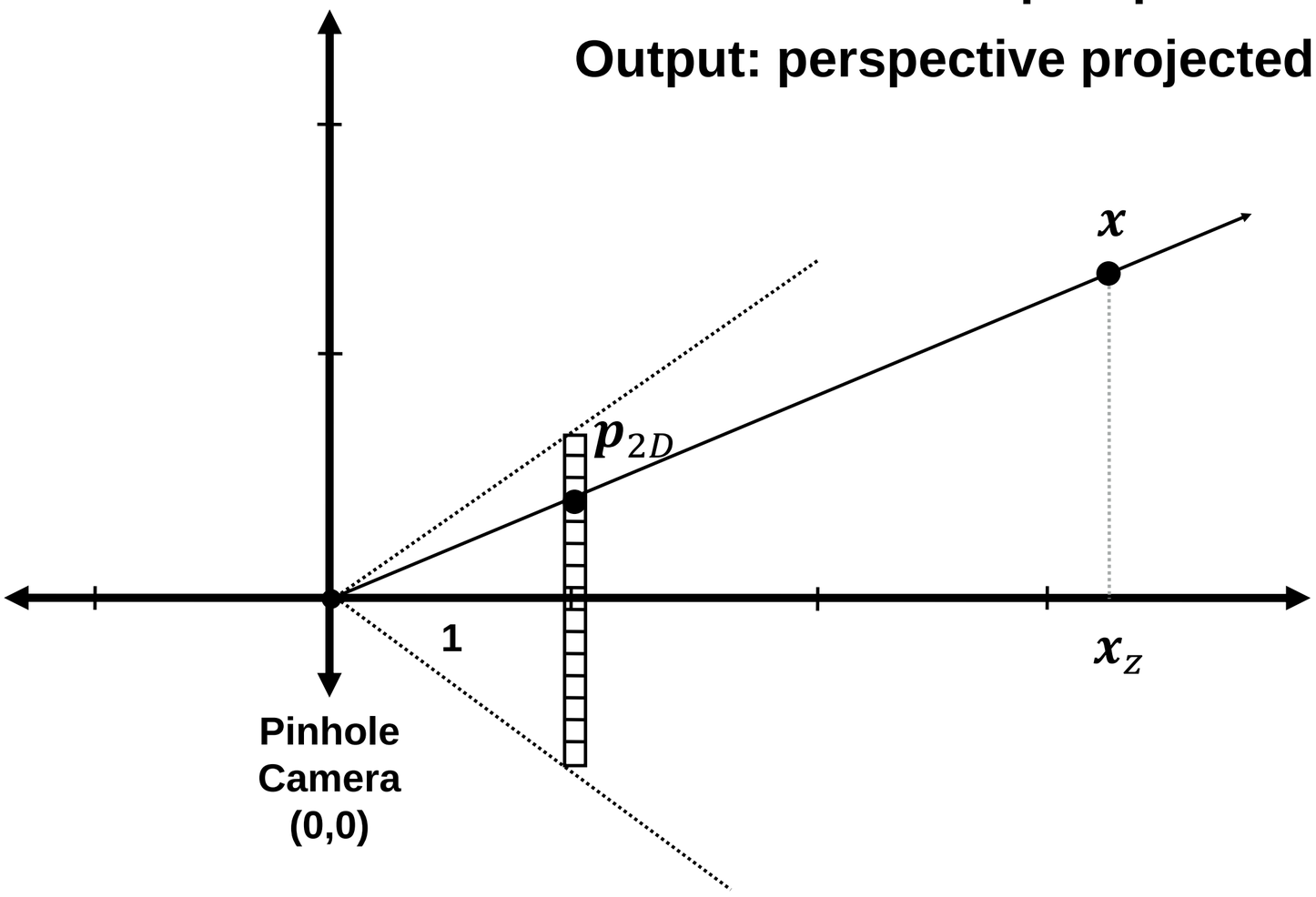
Masaccio – The Tribute Money c.1426-27 Fresco, The Brancacci Chapel, Florence

Later… rejection of proper perspective projection

Basic perspective projection

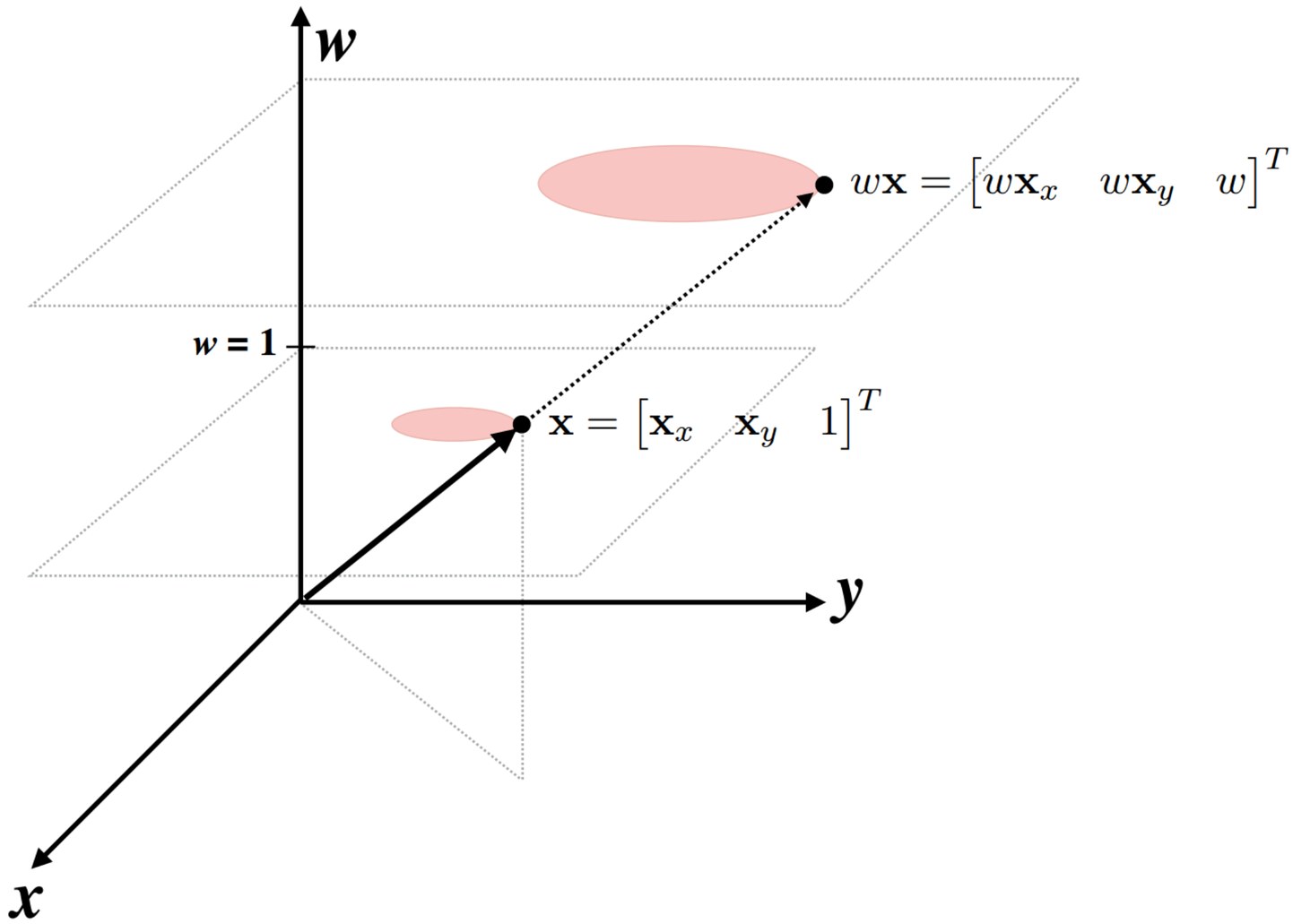
Input: point in 3D-H $\boldsymbol{x} = (\boldsymbol{x}_x, \boldsymbol{x}_y, \boldsymbol{x}_z, 1)$

Output: perspective projected point $\boldsymbol{p}_{2D} = (\boldsymbol{x}_x/\boldsymbol{x}_z, \boldsymbol{x}_y/\boldsymbol{x}_z)$



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

Review: homogeneous coordinates



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

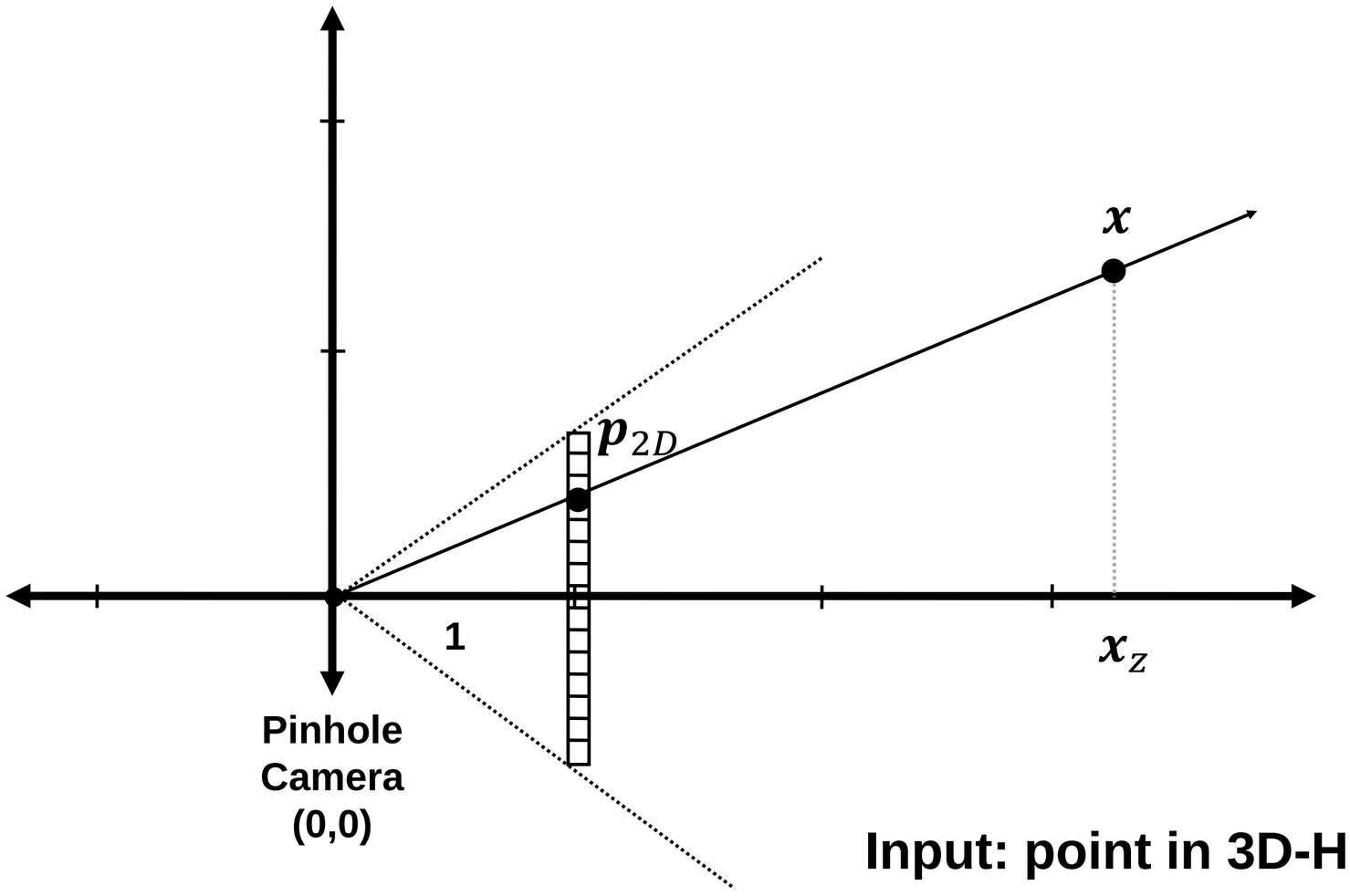
$\mathbf{x}$ and $w\mathbf{x}$ correspond to the same 2D point

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

Basic perspective projection

Desired perspective projected result (2D point):

$$\boldsymbol{p}_{2D} = (\boldsymbol{x}_x/\boldsymbol{x}_z,\ \boldsymbol{x}_y/\boldsymbol{x}_z)$$



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

Input: point in 3D-H $\boldsymbol{x} = (\boldsymbol{x}_x,\ \boldsymbol{x}_y, \boldsymbol{x}_z,\ 1)$

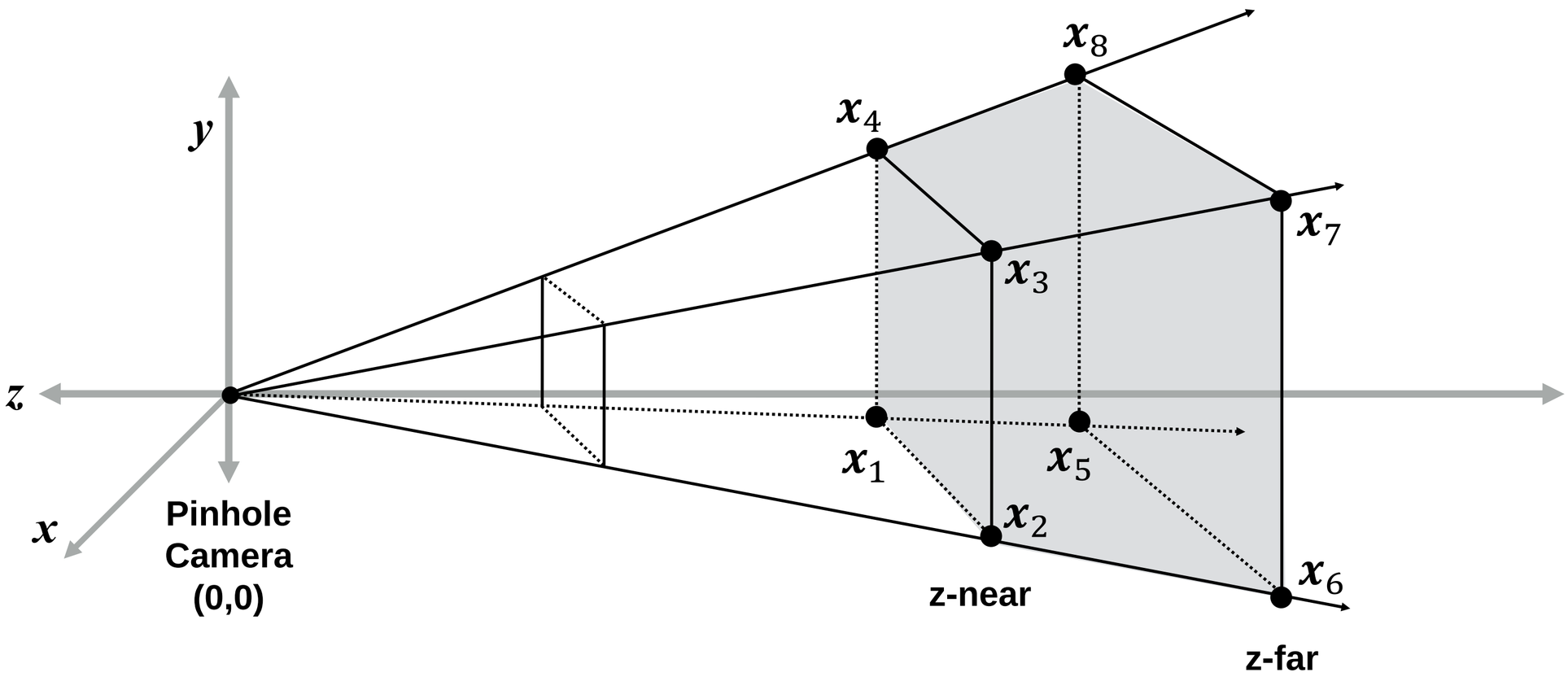
Applying map to get projected point in 3D-H $\boldsymbol{P}\boldsymbol{x} = (\boldsymbol{x}_x,\ \boldsymbol{x}_y, \boldsymbol{x}_z,\ \boldsymbol{x}_z)$

Point projected to 2D-H (drop z coord) $\boldsymbol{p}_{2D-H} = (\boldsymbol{x}_x,\ \boldsymbol{x}_y, \boldsymbol{x}_z)$

Point in 2D (homogeneous divide) $\boldsymbol{p}_{2D} = (\boldsymbol{x}_x/\boldsymbol{x}_z,\ \boldsymbol{x}_y/\boldsymbol{x}_z)$

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

The view frustum

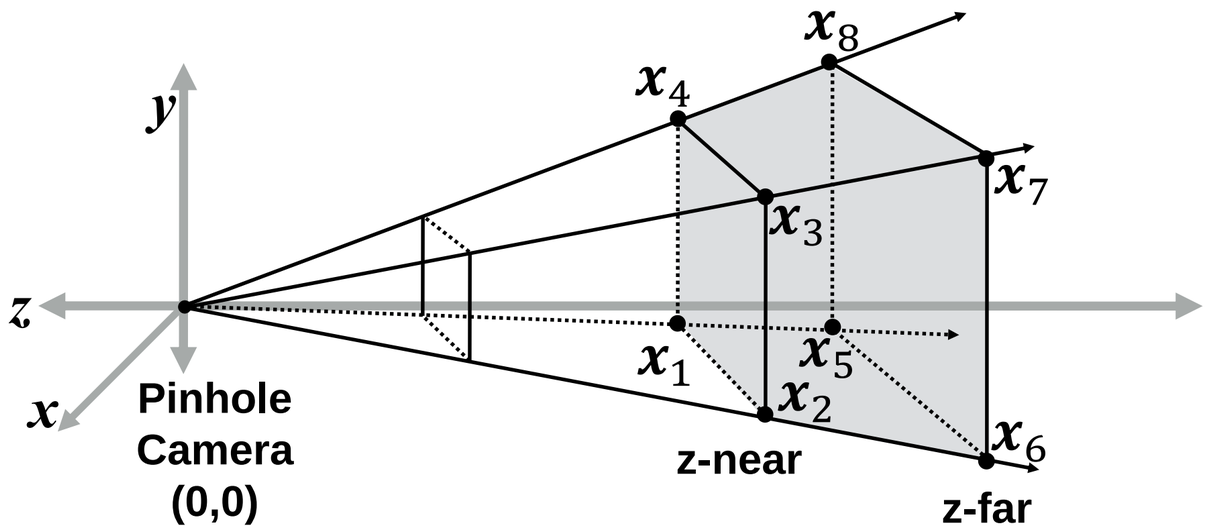


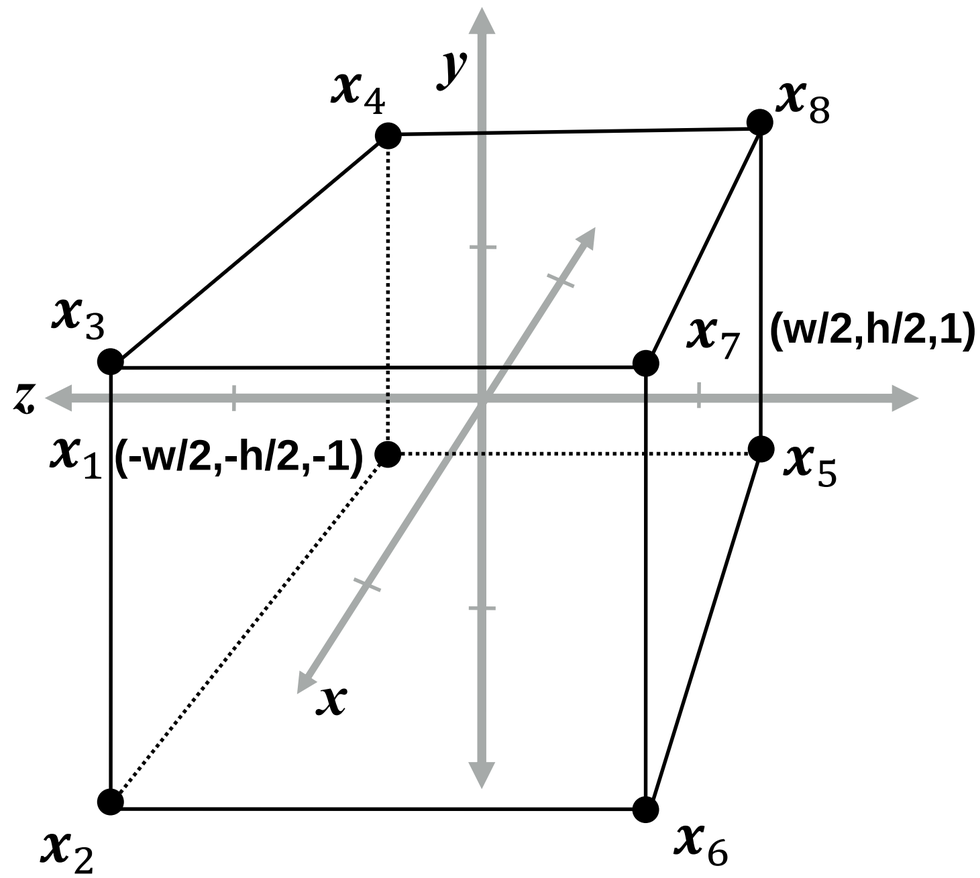
View frustum: region in space that may appear on the screen

How can we change the shape of the view frustum?

Want a transformation that maps view frustum to a unit cube (computing screen coordinates in that space is then trivial)

The view frustum





Projecting on virtual screen

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

$$\boldsymbol{Px} = (\boldsymbol{x}_x, \boldsymbol{x}_y, \boldsymbol{x}_z, \boldsymbol{x}_z)$$

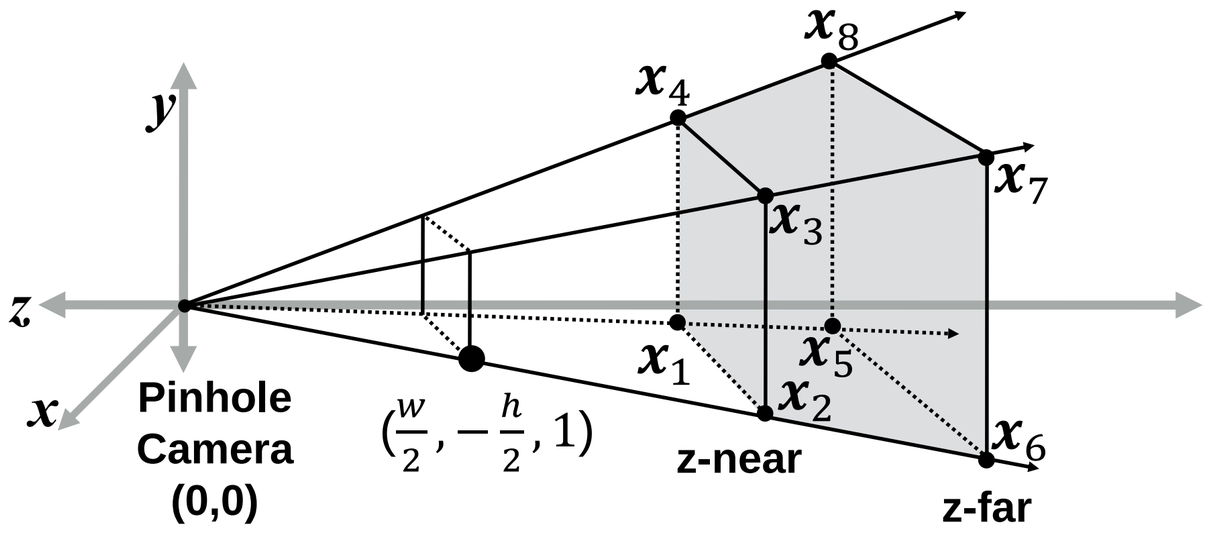
First step: apply perspective transform & normalize z-coord
What are the values of a and b?

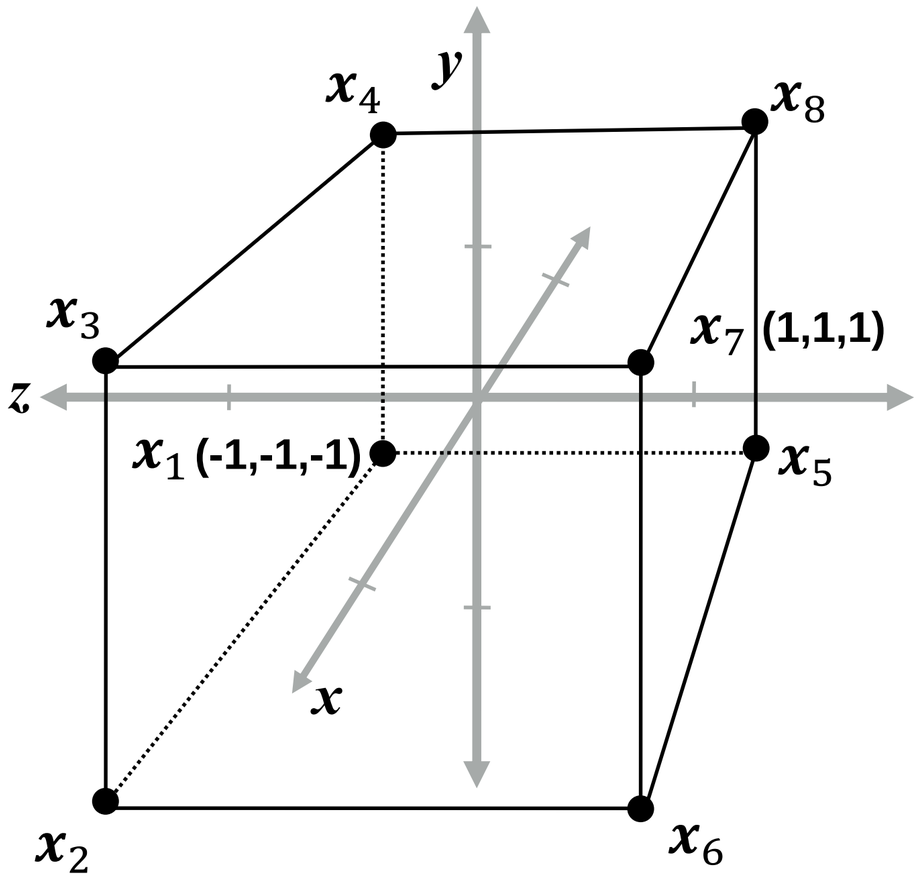
$$\mathbf{P}_z = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & a & b \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

$$\boldsymbol{x}_1 = (\boldsymbol{x}_{1x}, \boldsymbol{x}_{1y}, -zNear, 1)\ ;\ \boldsymbol{Px}_1 = (\boldsymbol{x}_{1x}, \boldsymbol{x}_{1y}, -zNear, zNear)$$

$$\boldsymbol{x}_7 = (\boldsymbol{x}_{7x}, \boldsymbol{x}_{7y}, -zFar, 1);\ \boldsymbol{Px}_7 = (\boldsymbol{x}_{7x}, \boldsymbol{x}_{7y}, zFar, zFar)$$

The view frustum





Changing the shape of the view frustum:

θ**: field of view in y direction (**$h = tan\,(\frac{\theta}{2})$**)**

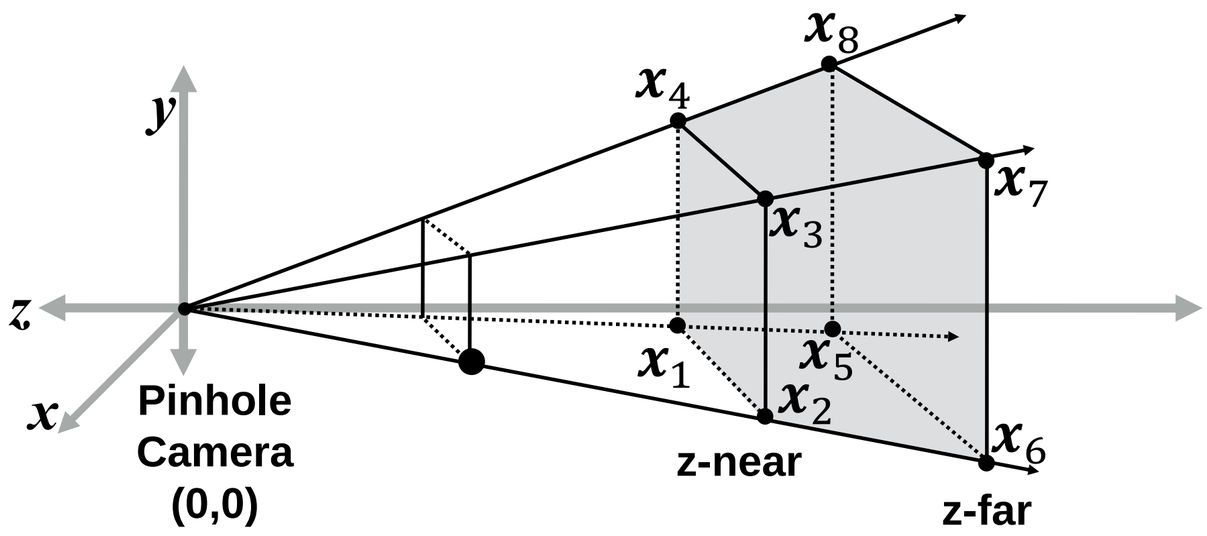
r = ***aspect ratio*** **= width / height**

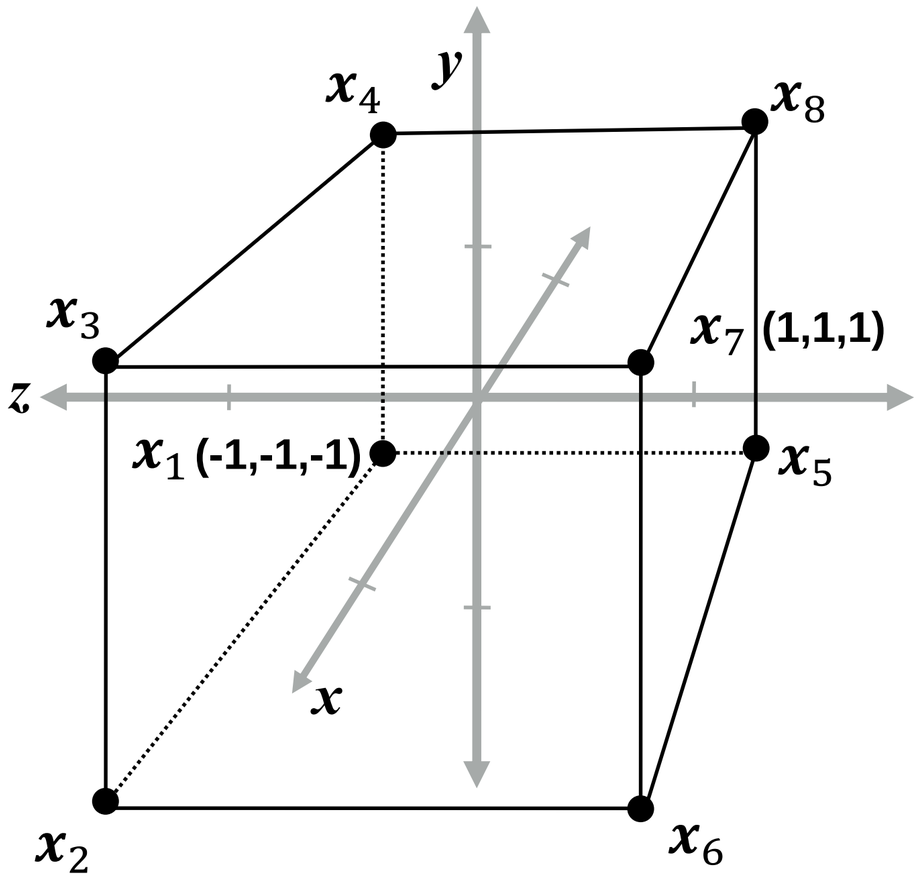
After normalizing z-component, rescale x and y components such that frustum maps to unit cube

$$\begin{bmatrix} \frac{f}{r} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

$$f = \mathbf{2}cot(\frac{\boldsymbol{\theta}}{\mathbf{2}})$$

The view frustum



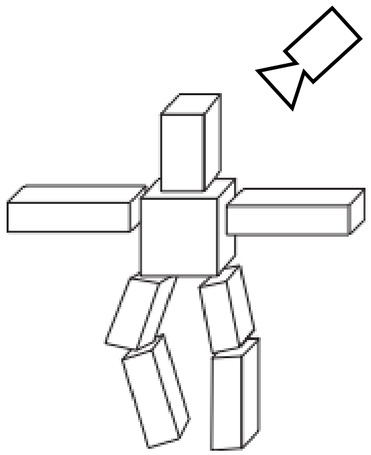


Putting it all together, transformation matrix that maps view frustum to unit cube:

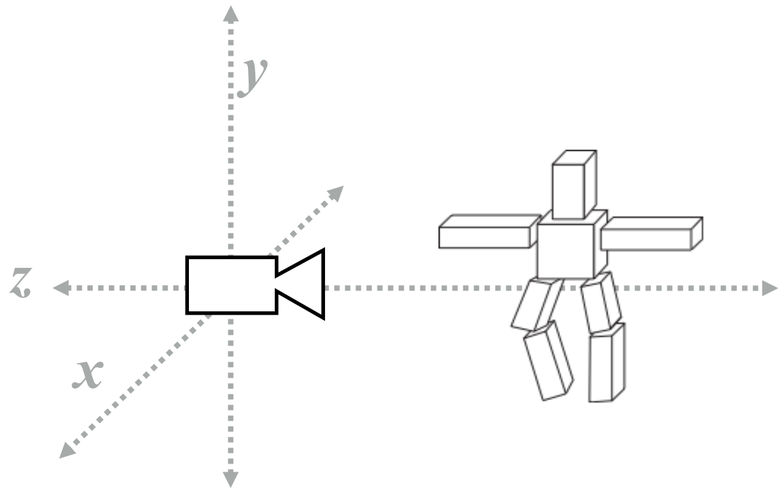
$$\mathbf{P} = \begin{bmatrix} \frac{f}{aspect} & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & \frac{zfar+znear}{znear-zfar} & \frac{2\times zfar\times znear}{znear-zfar} \\ 0 & 0 & -1 & 0 \end{bmatrix}$$

Q1: What kinds of features does this transformation preserve?
Q2: Is this transformation matrix invertible?

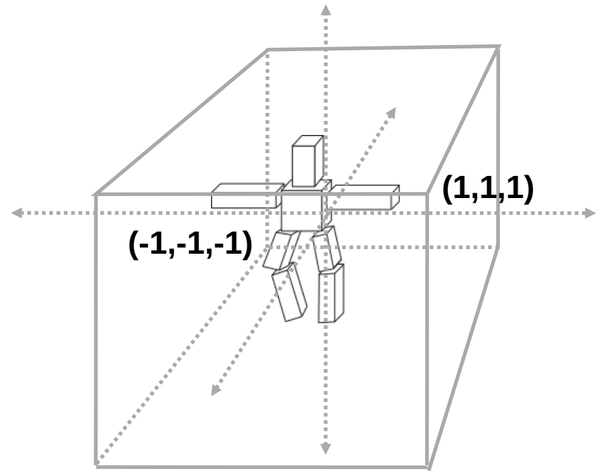
Transformations recap



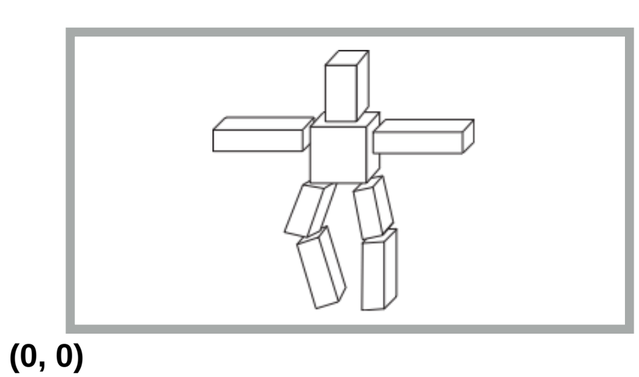
Modeling transforms:
Position object in scene



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

Review exercise: screen transform *

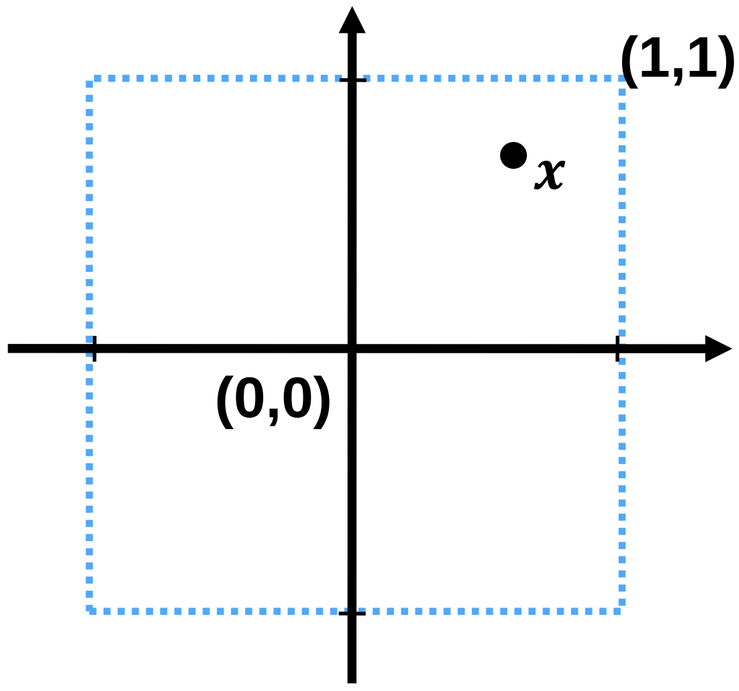
Convert points in normalized coordinate space to screen pixel coordinates

Example:

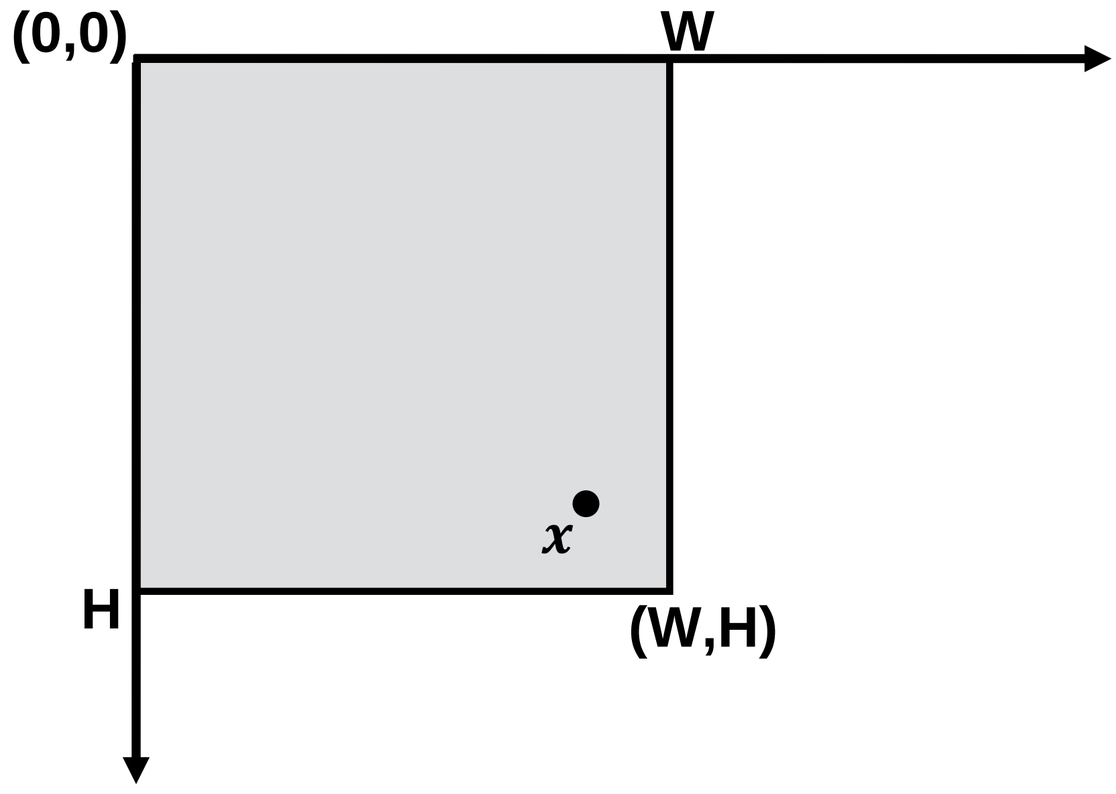
All points within (-1,1) to (1,1) region are on screen

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

Normalized coordinate space:



Screen (W x H output image) coordinate space:



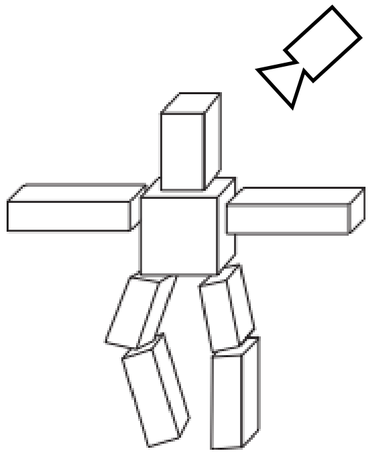
Step 1: reflect about x

Step 2: translate by (1,1)

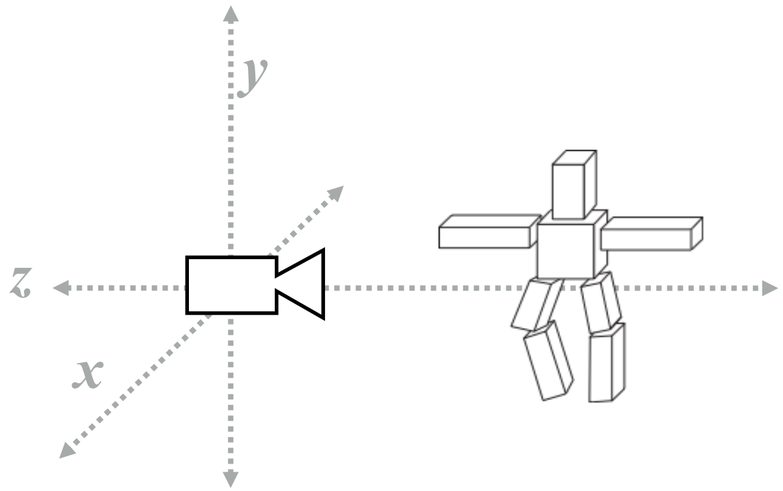
Step 3: scale by (W/2,H/2)

*** Adopting 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?

Transformations recap

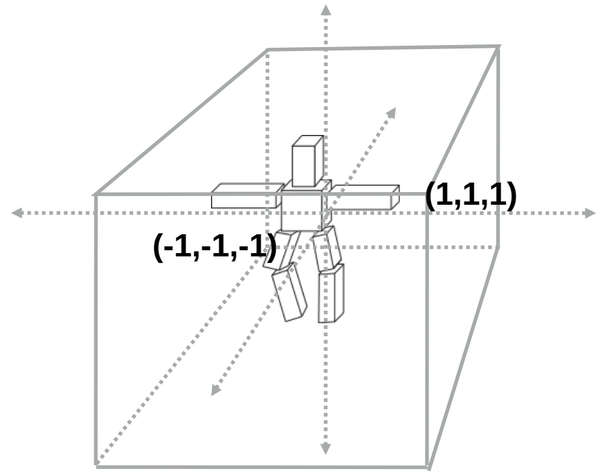


Modeling transforms:
Position object in scene



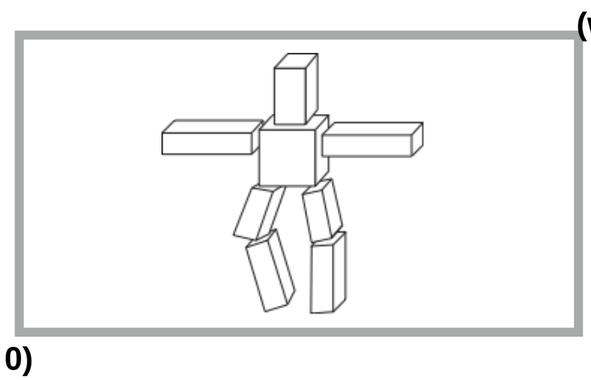
Viewing (camera) transform:
positions objects in coordinate space relative to camera

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

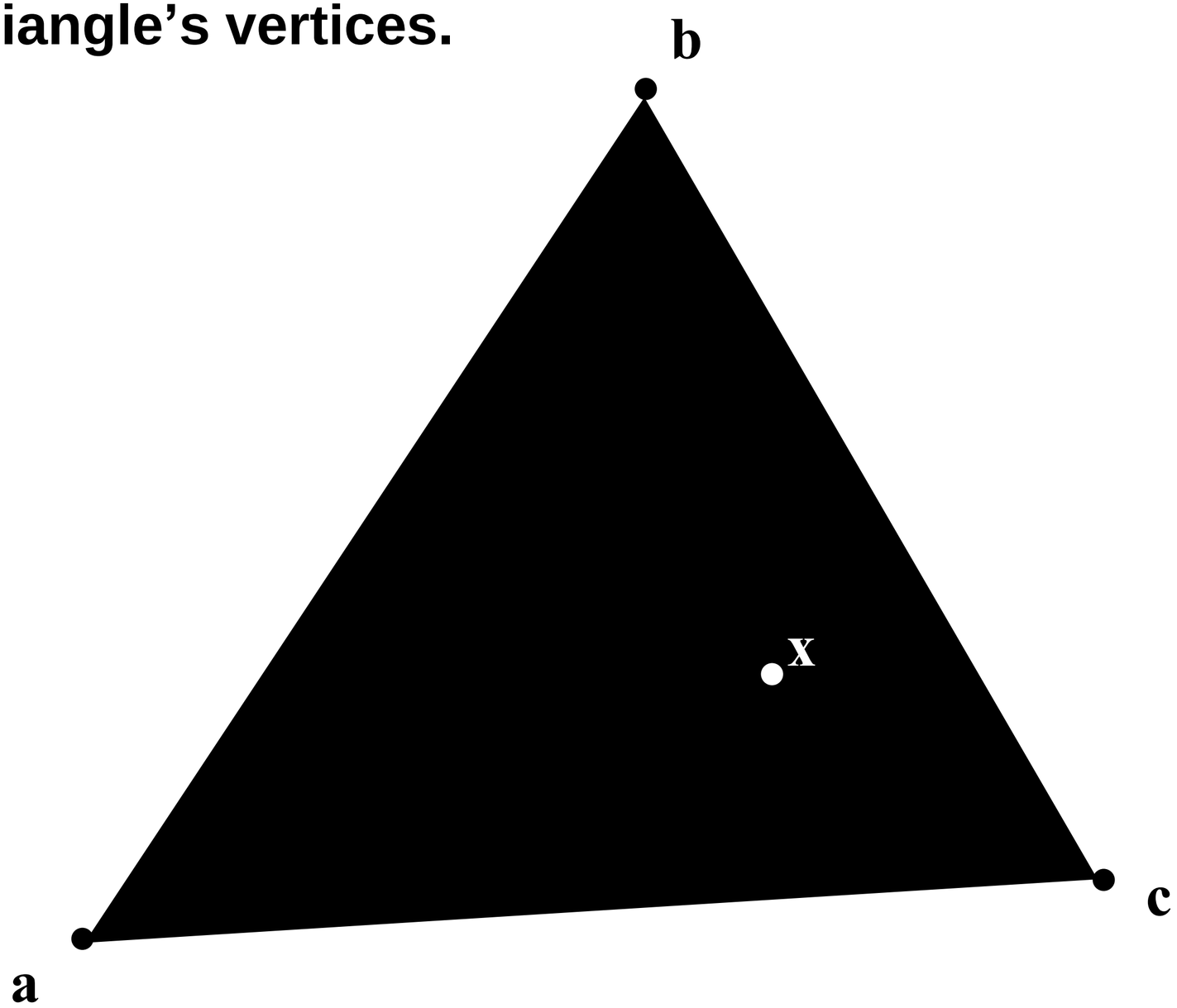


Screen transform:
objects now in 2D screen coordinates

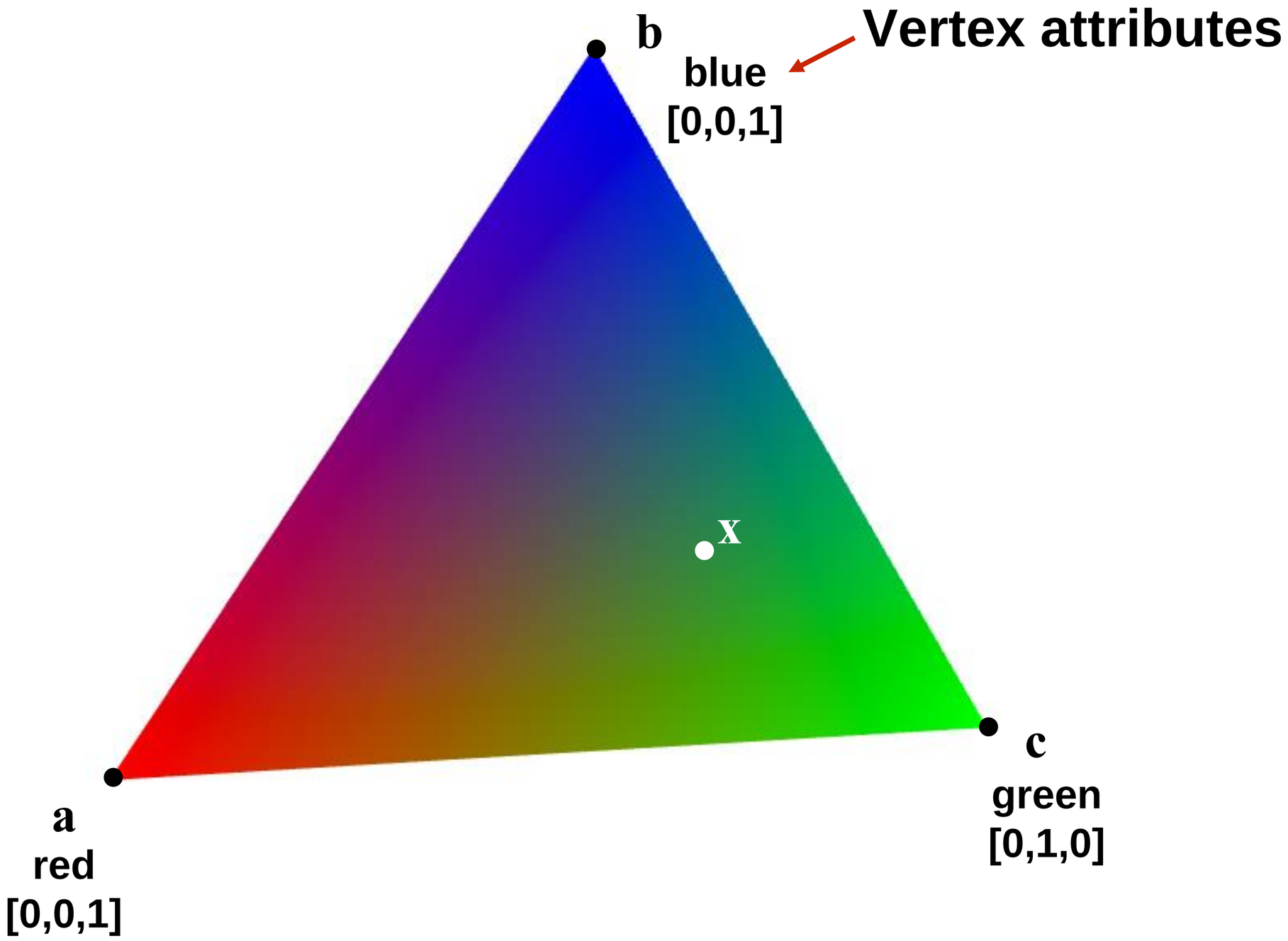
Compute
screen coverage from 2D object position

Coverage(x,y)

A few lectures ago 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 point x?

Review: interpolation in 1D

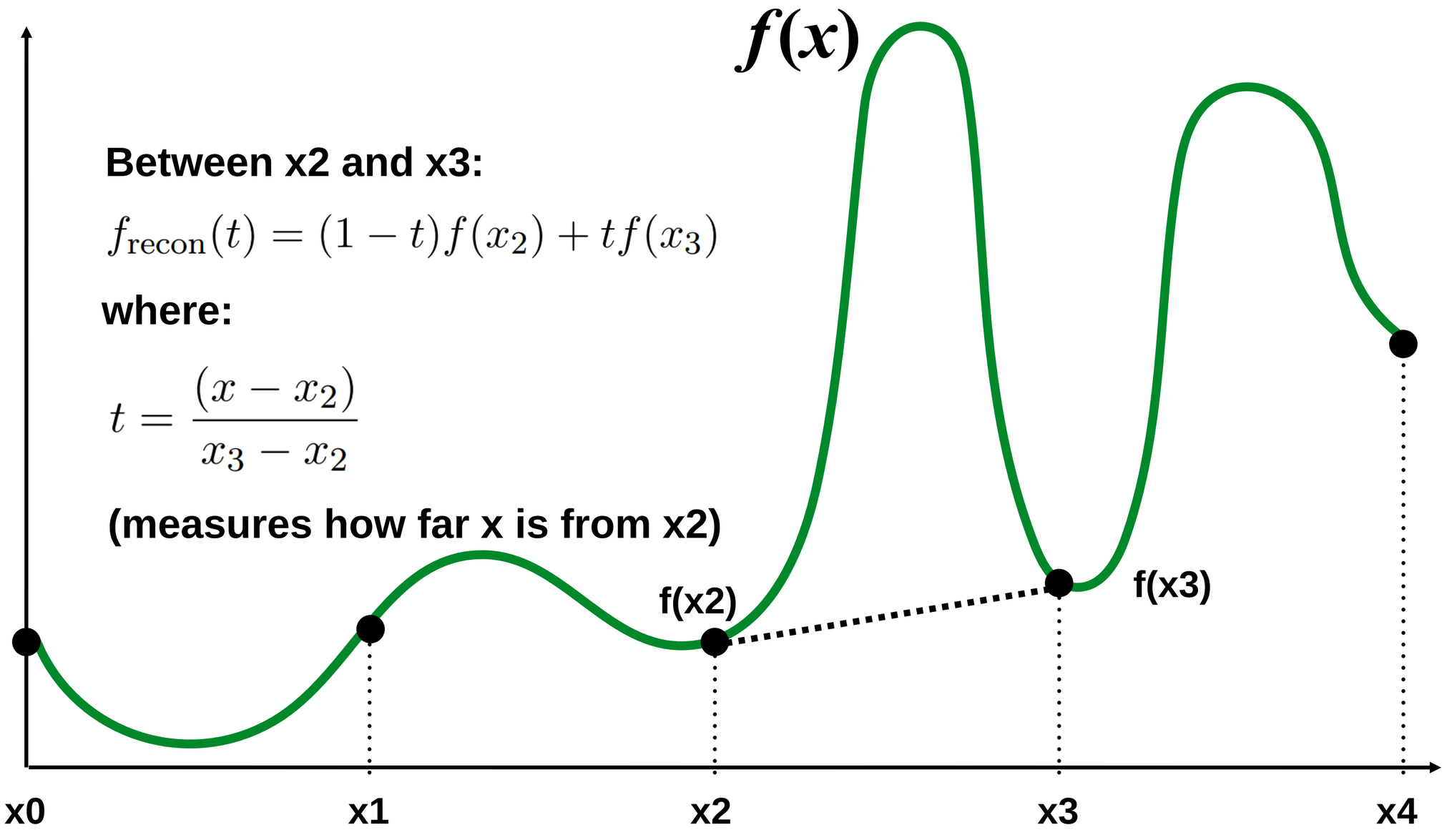
Between x2 and x3:

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

where:

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

(measures how far x is from x2)

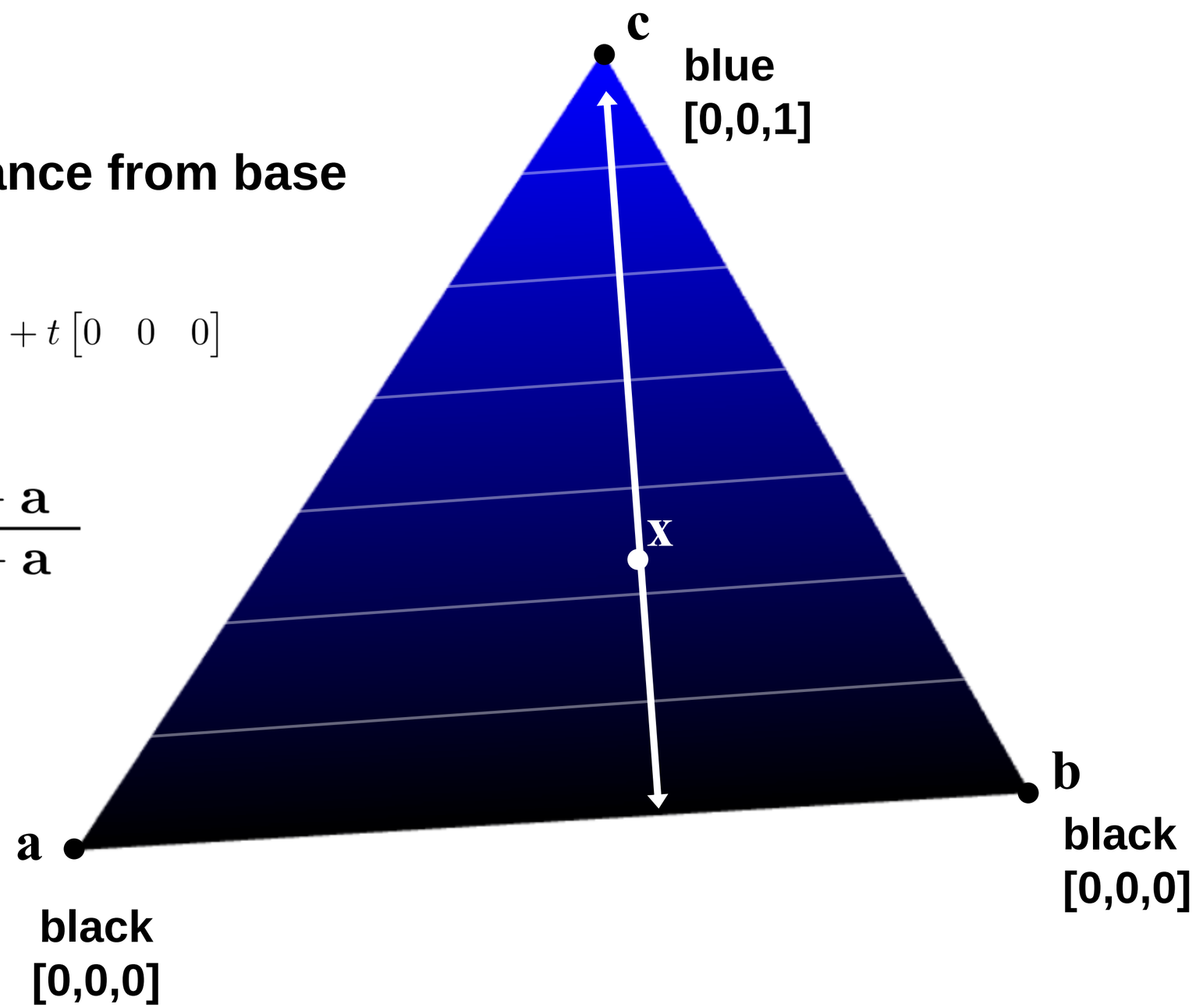


Consider similar behavior on triangle

Color depends on distance from base

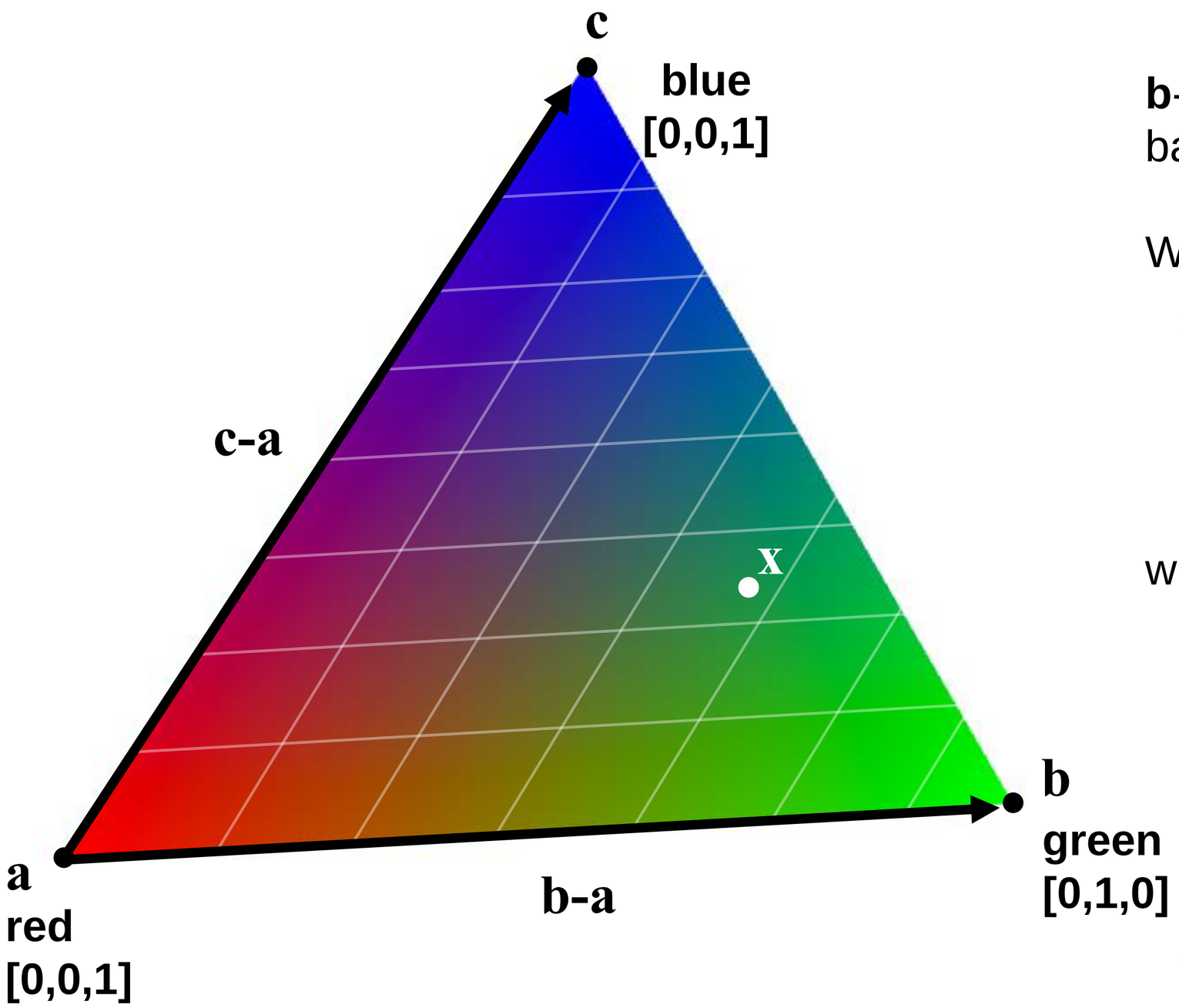
color at $\mathbf{x} = (1-t)\begin{bmatrix}0 & 0 & 1\end{bmatrix} + t\begin{bmatrix}0 & 0 & 0\end{bmatrix}$

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



How can we interpolate in 2D between three values?

Interpolation via barycentric coordinates



b-**a** and **c**-**a** form a non-orthogonal basis for points in triangle.

We can therefore write:

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

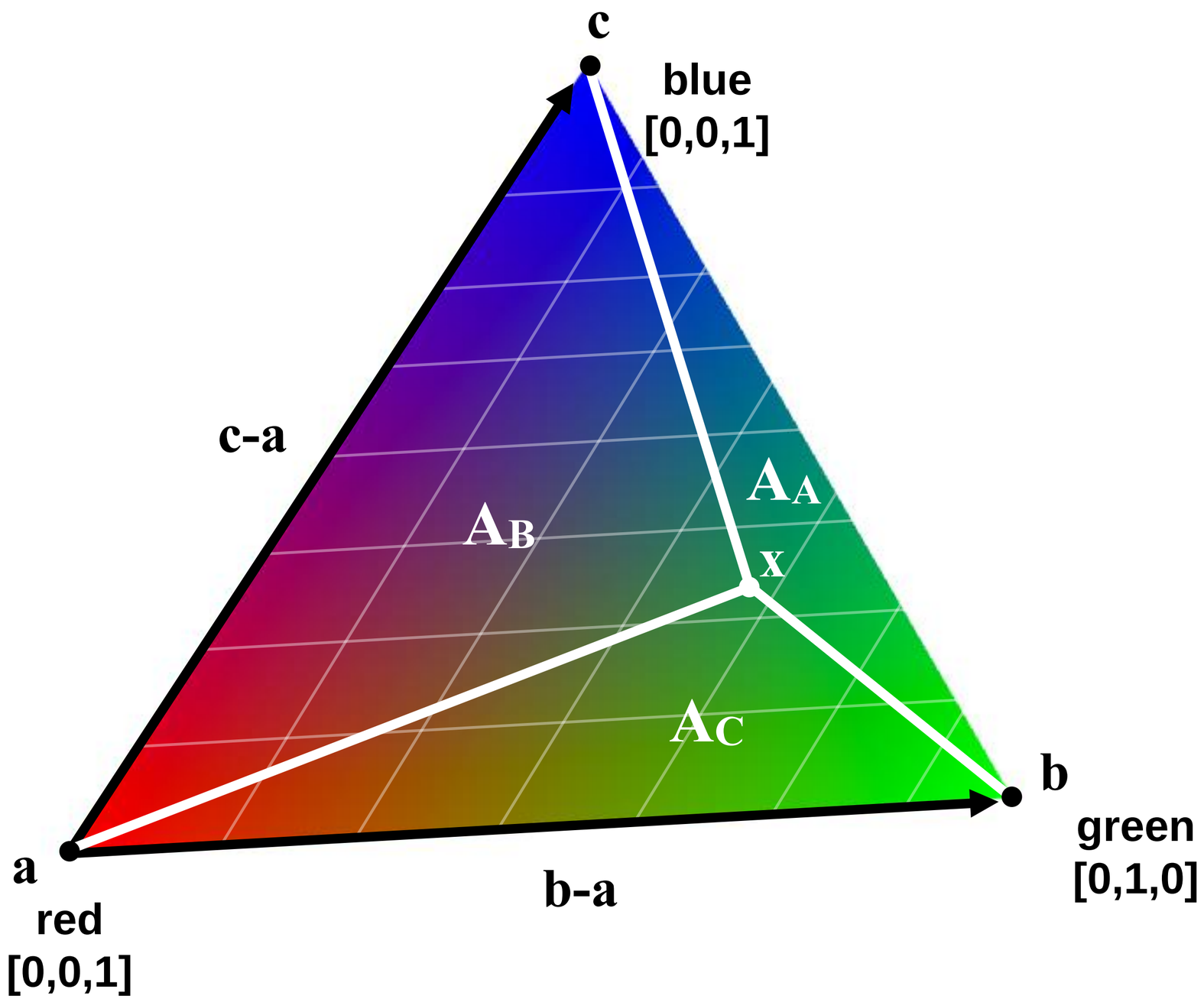
with

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

Color at **x** is **affine** 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 ratio of areas



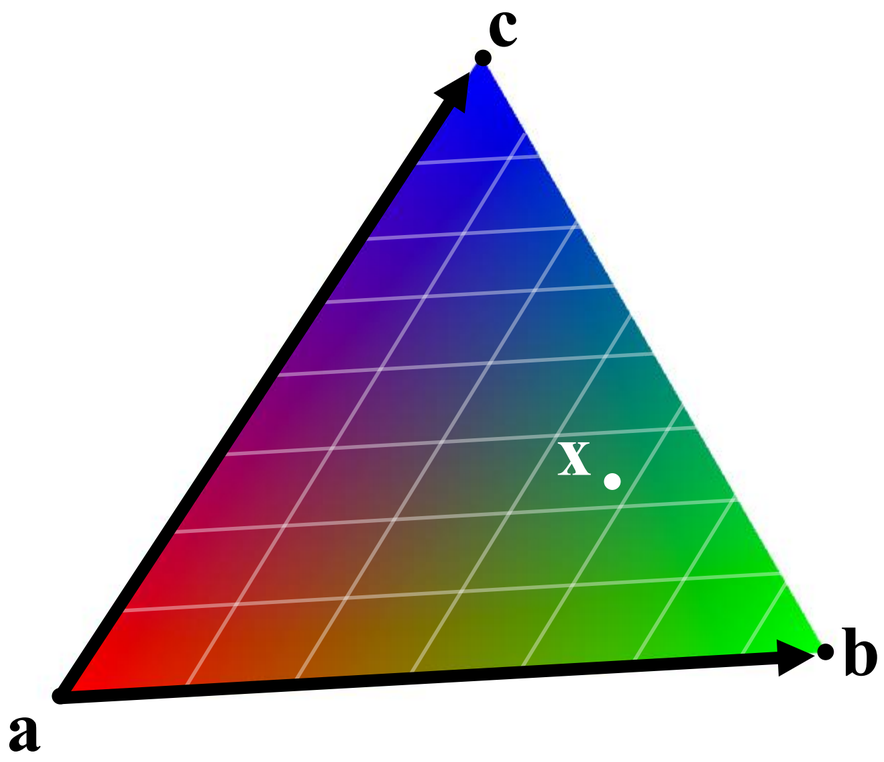
$$\alpha = A_A/A$$

$$\beta = A_B/A$$

$$\gamma = A_C/A$$

In the context of a triangle, Barycentric coordinates are also called areal coordinates

Barycentric interpolation as an affine map



$$f_x = \alpha f_a + \beta f_b + \gamma f_c$$

$$= [f_a \quad f_b \quad f_c] \begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix}$$

but

$$\begin{bmatrix} \alpha \\ \beta \\ \gamma \end{bmatrix} = A \begin{bmatrix} \boldsymbol{x}_x \\ \boldsymbol{x}_y \\ 1 \end{bmatrix}$$

so

$$f_x = Ax_x + Bx_y + C$$

Direct evaluation of surface attributes

For any surface attribute, value at $\boldsymbol{x}$ is

$$f_x = A\boldsymbol{x}_x + B\boldsymbol{x}_y + C$$

To find, A, B and C, plug in values at triangle vertices, where we know the values of the attribute (f_a, f_b, f_c)

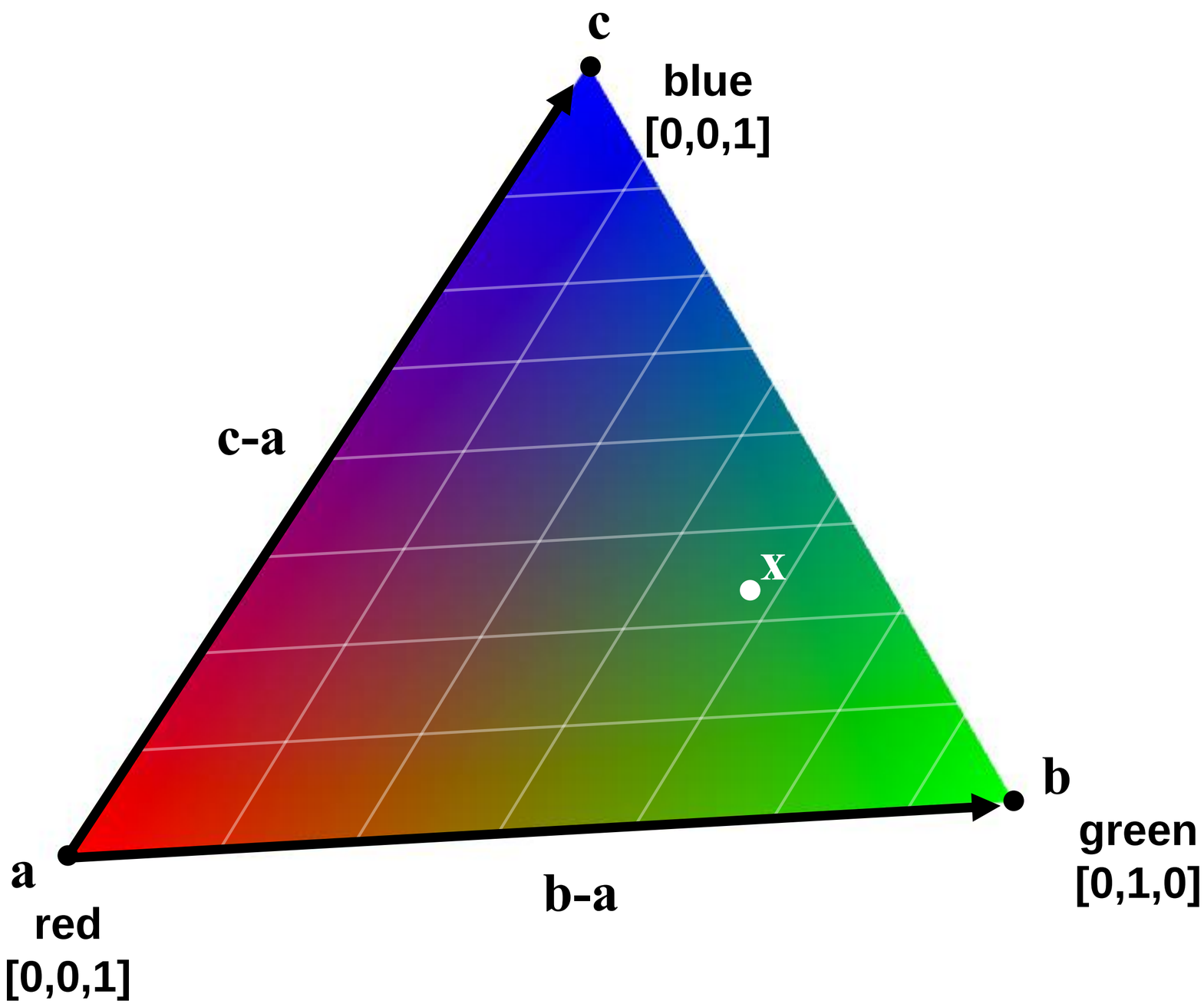
$$f_a = A\mathbf{a}_x + B\mathbf{a}_y + C$$

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

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

3 equations, solve for 3 unknowns (A, B, C)
Note: A, B and C will be different for different attributes

But what are we interpolating again?

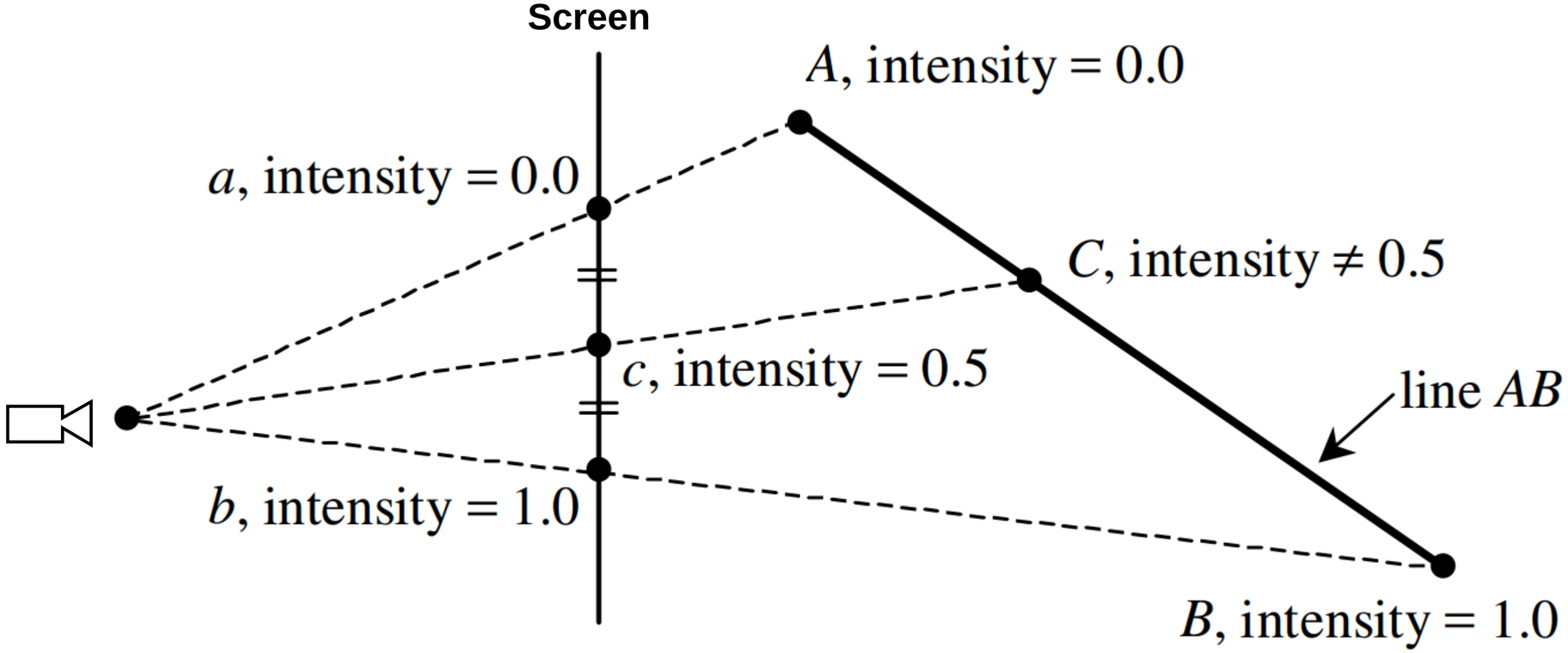


What are **a**, **b** and **c**?

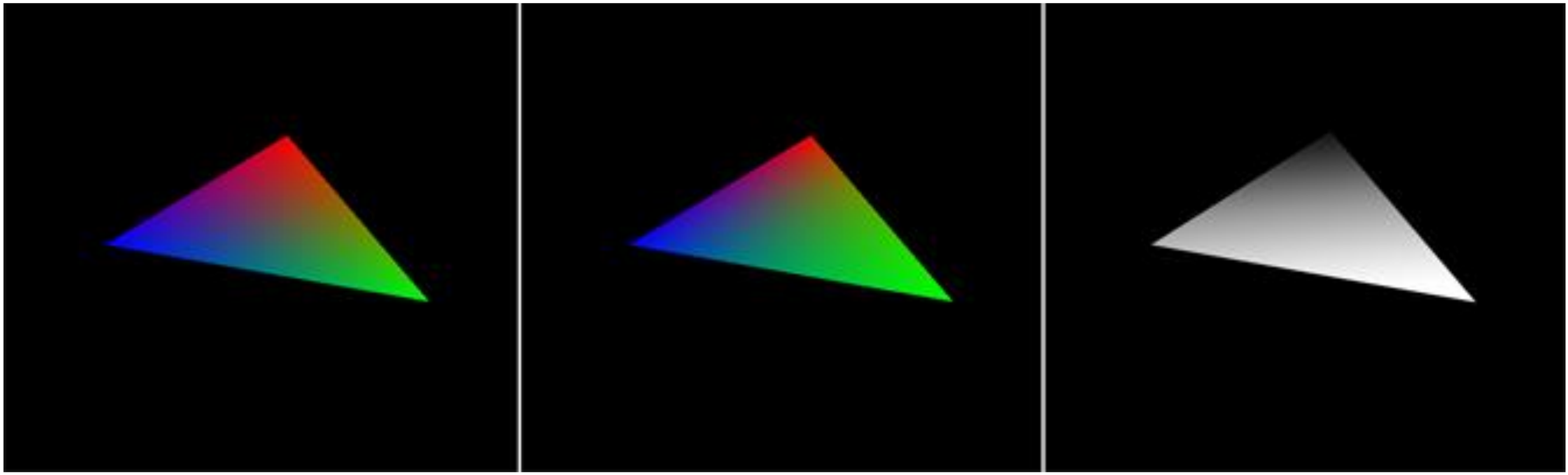
Perspective-incorrect interpolation

Due to projection, affine function (barycentric interpolation) in screen XY coordinates does not correspond to values that vary linearly on a triangle with vertices at different depths

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



An example: perspective-incorrect 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 therefore:**

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

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…**

Perspective-correct interpolation

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

Affine combinations of projected attribute values (f/w) correspond to linear interpolation in 3D

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

Consider the following example

Texture mapping

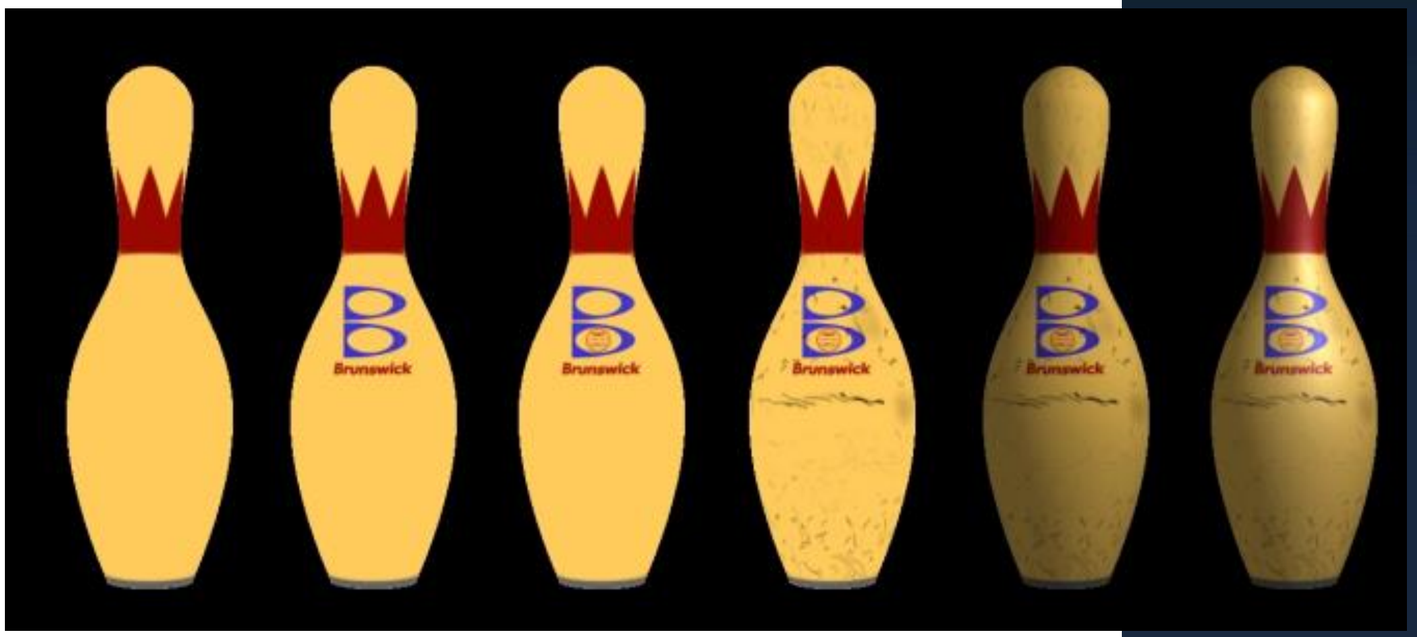
Many uses of texture mapping

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

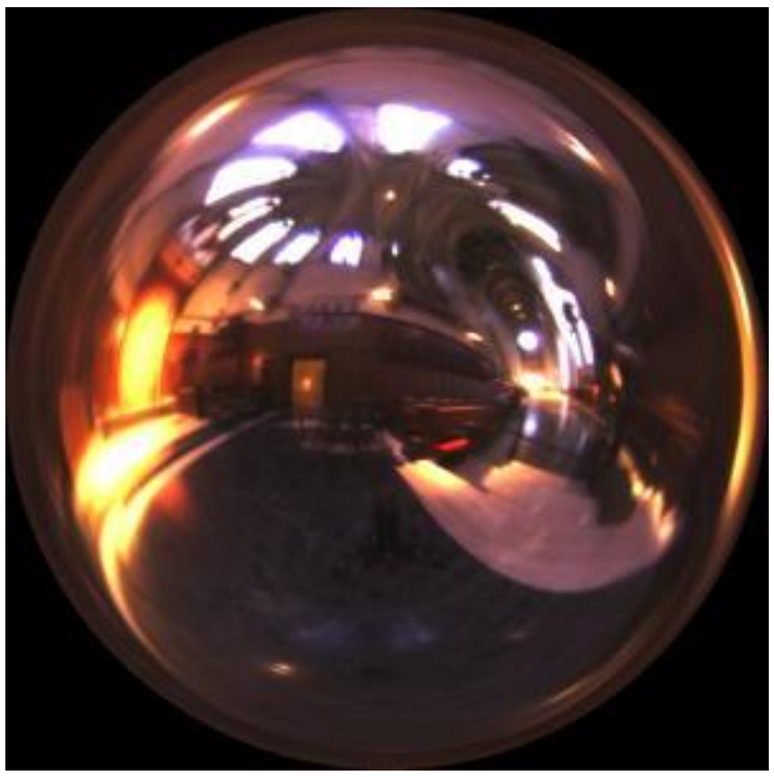
Rendering using high-resolution surface geometry (note bumpy silhouette and shadow boundary)

Represent precomputed lighting and shadows

Original model

With ambient occlusion

Extracted ambient occlusion map

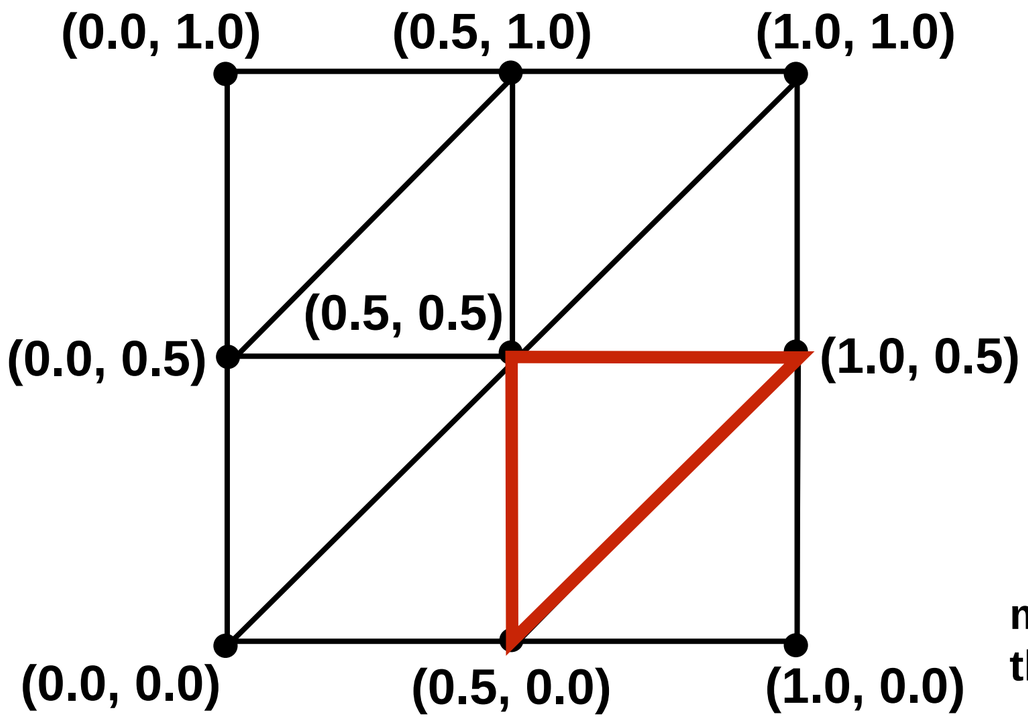


Grace Cathedral environment map

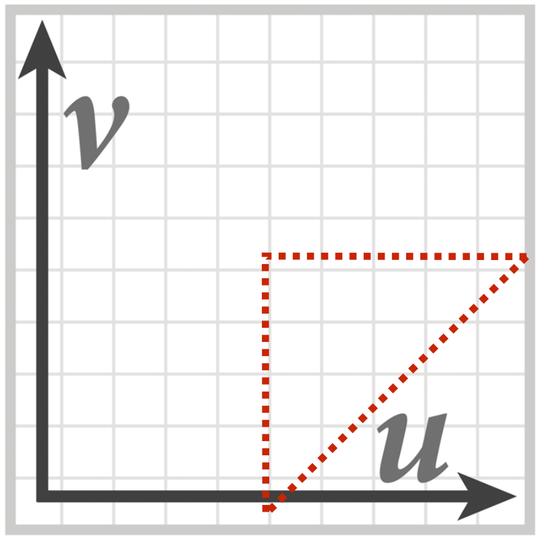
Environment map used in rendering

Texture coordinates

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



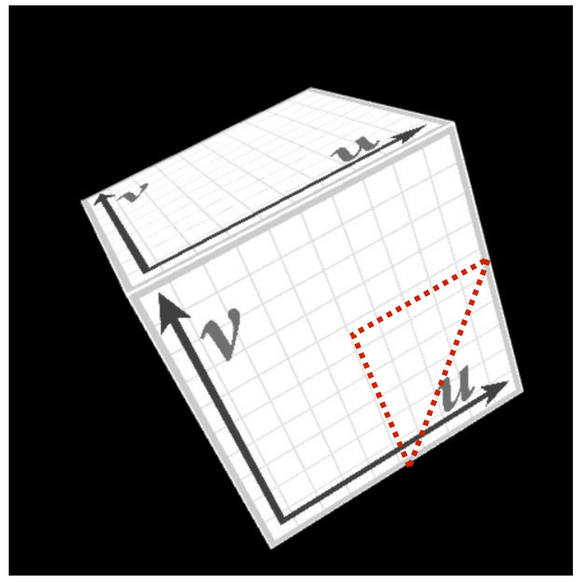
Eight triangles (one face of cube) with surface parameterization provided as per-vertex texture coordinates.



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.



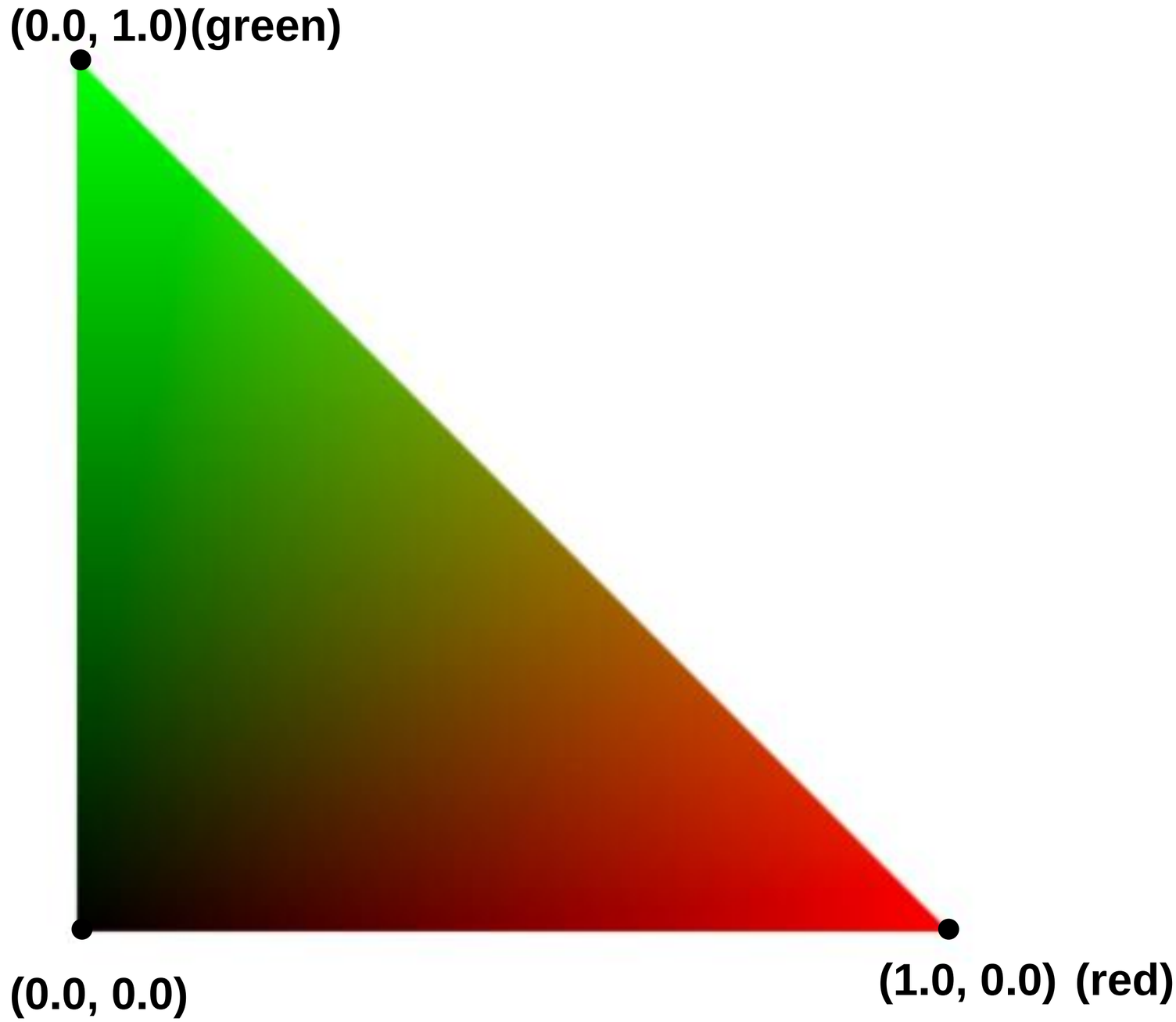
Final rendered result (entire cube shown).

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)

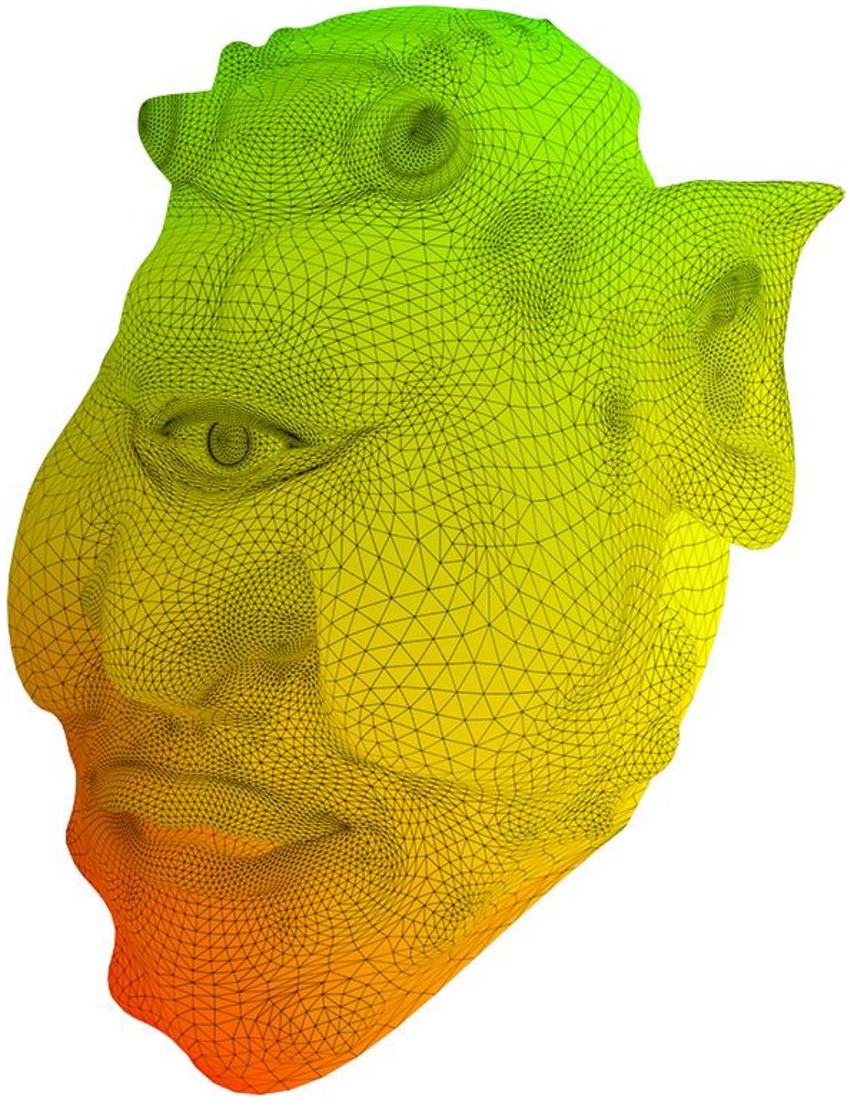
Visualization of texture coordinates

Texture coordinates linearly interpolated over triangle

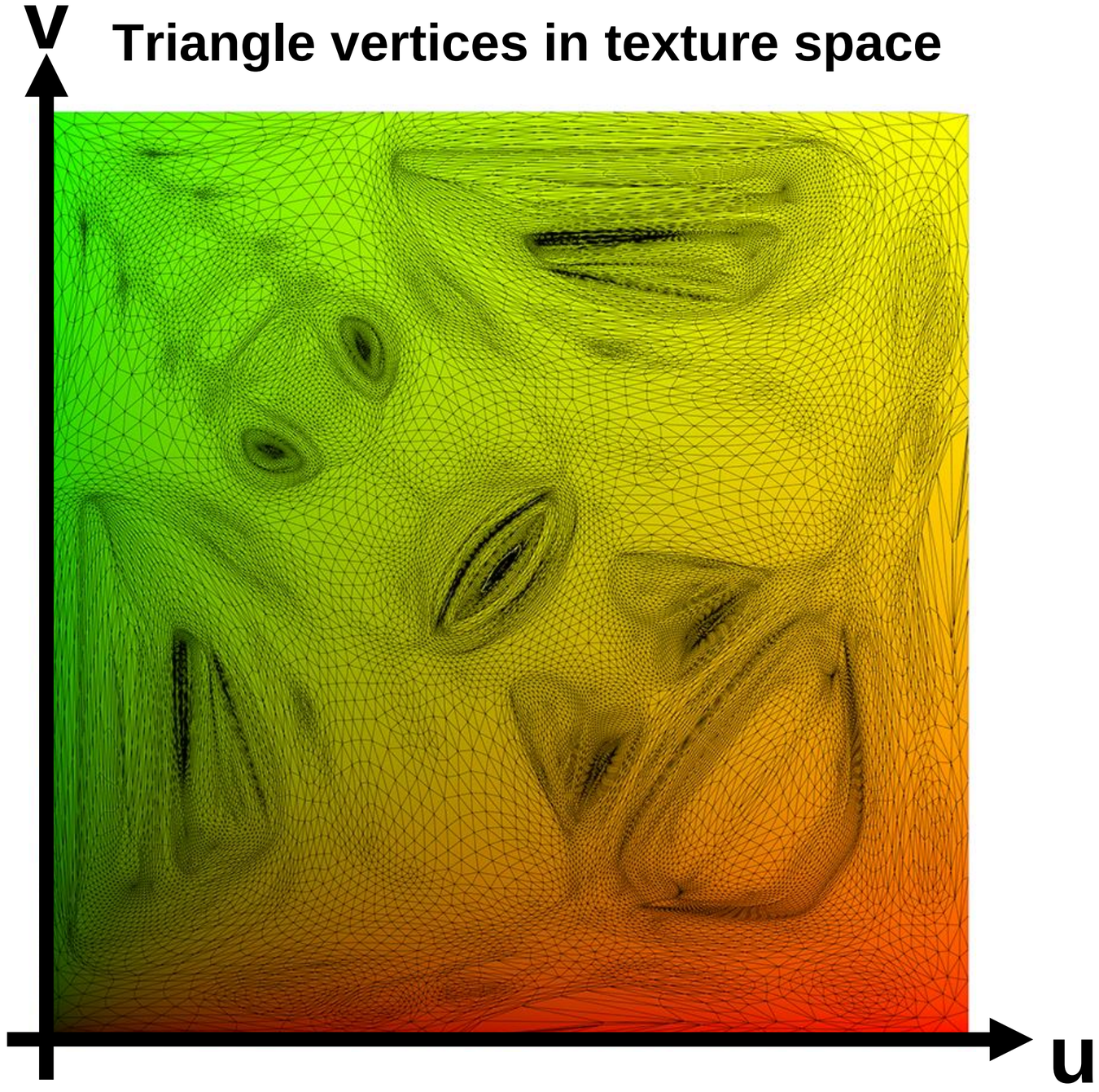


More complex mapping

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):

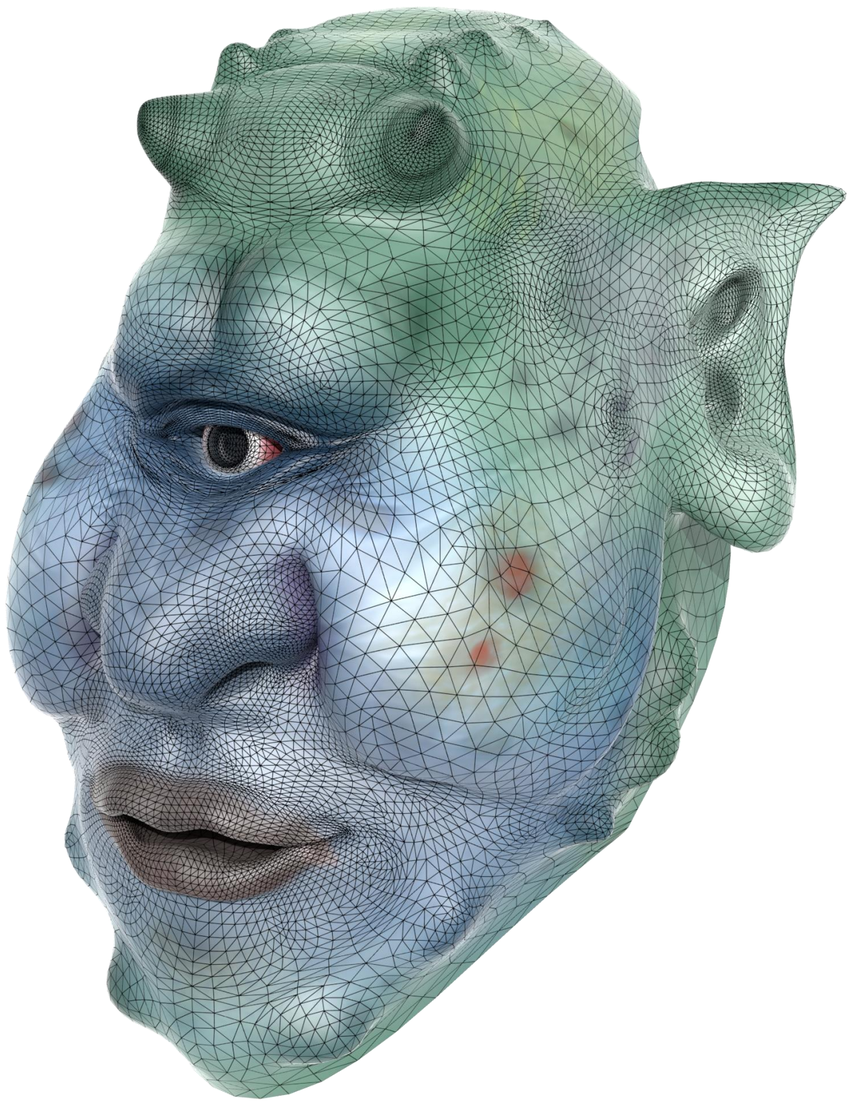
(u,v) = evaluate texcoord value at (x,y)

float3 texcolor = texture.sample(u,v); ← “just” an image lookup…

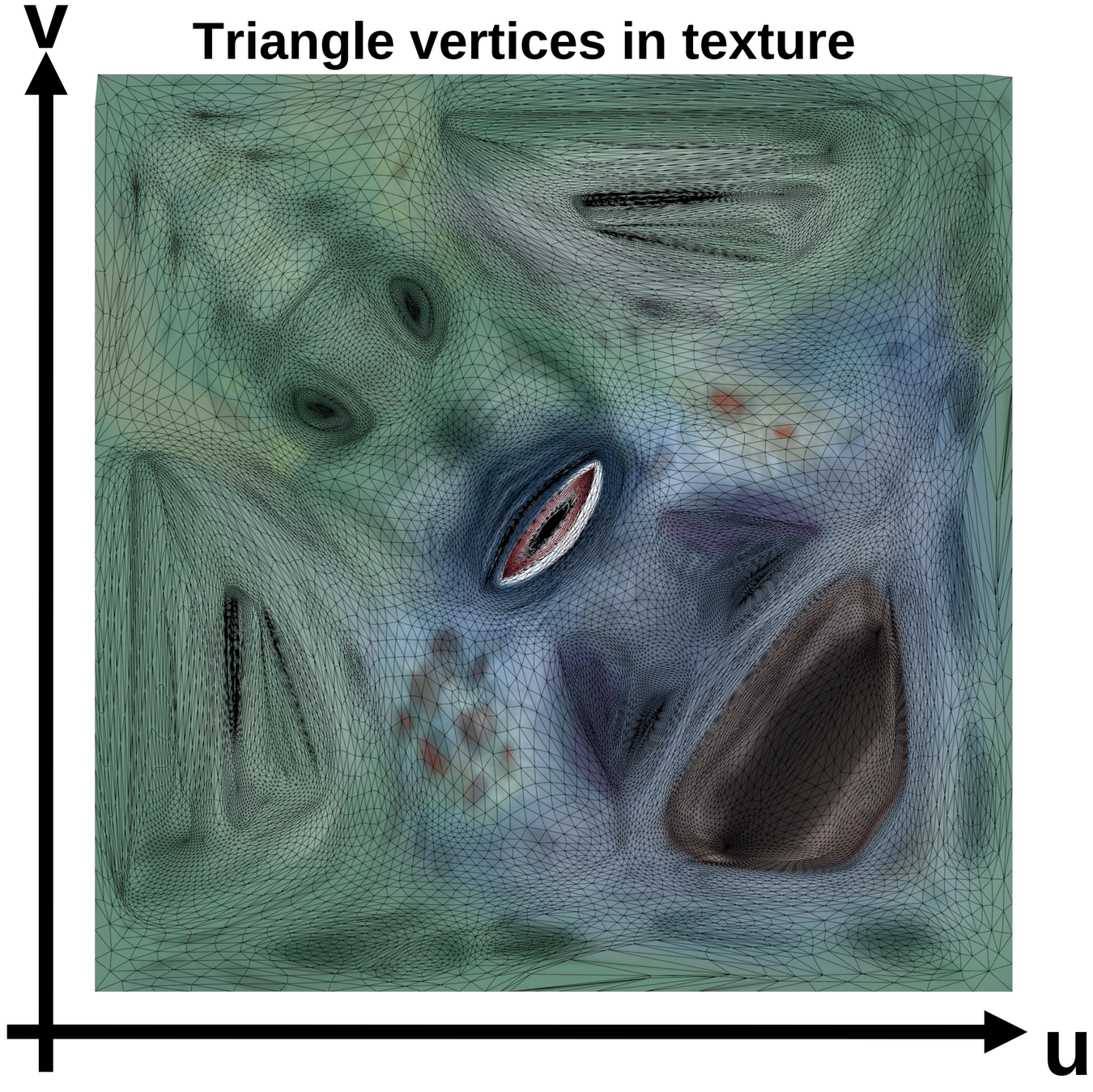
set sample’s color to texcolor;

Texture mapping adds detail

Rendered result



Triangle vertices in texture

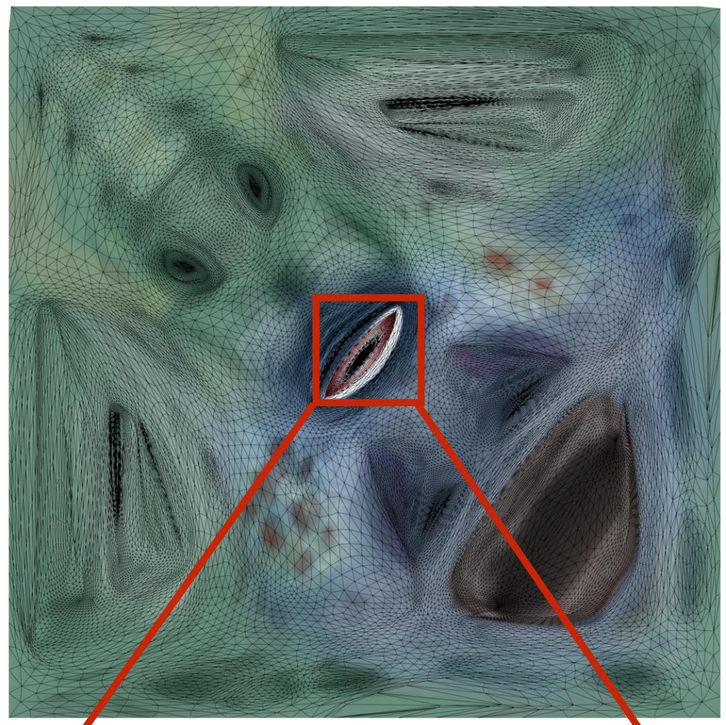


Texture mapping adds detail

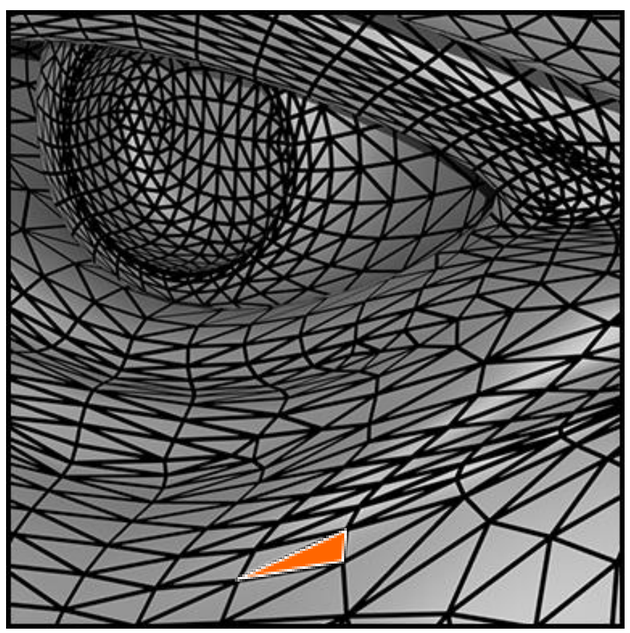
rendering without

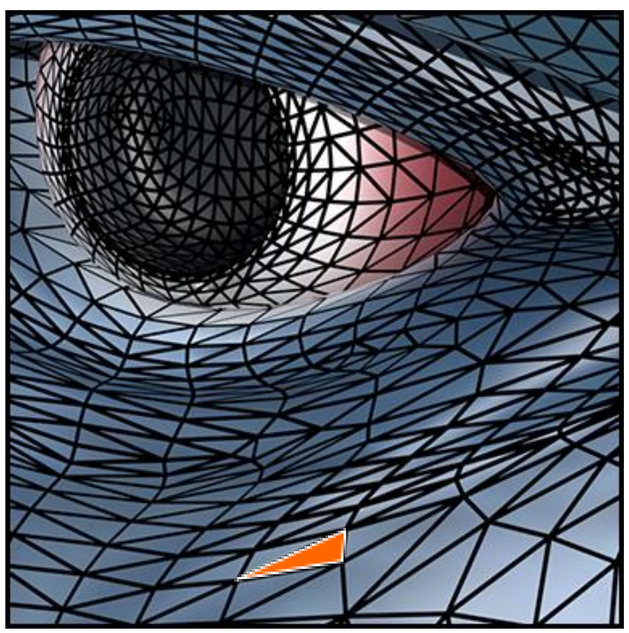
rendering with texture

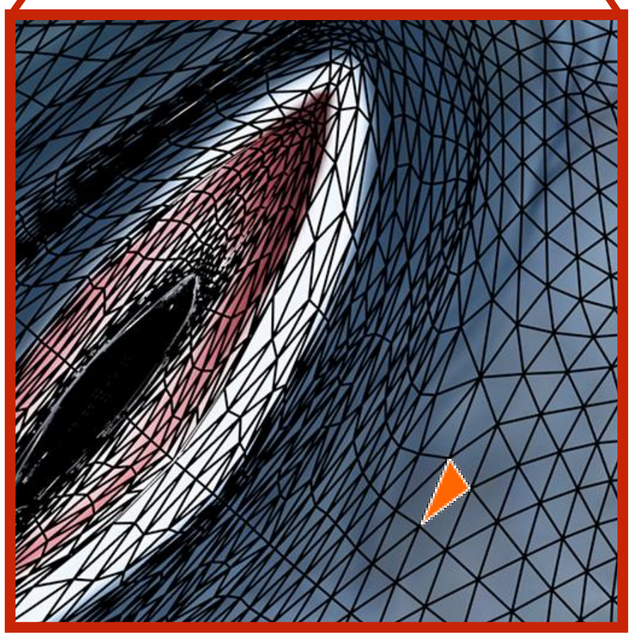
texture image



zoom

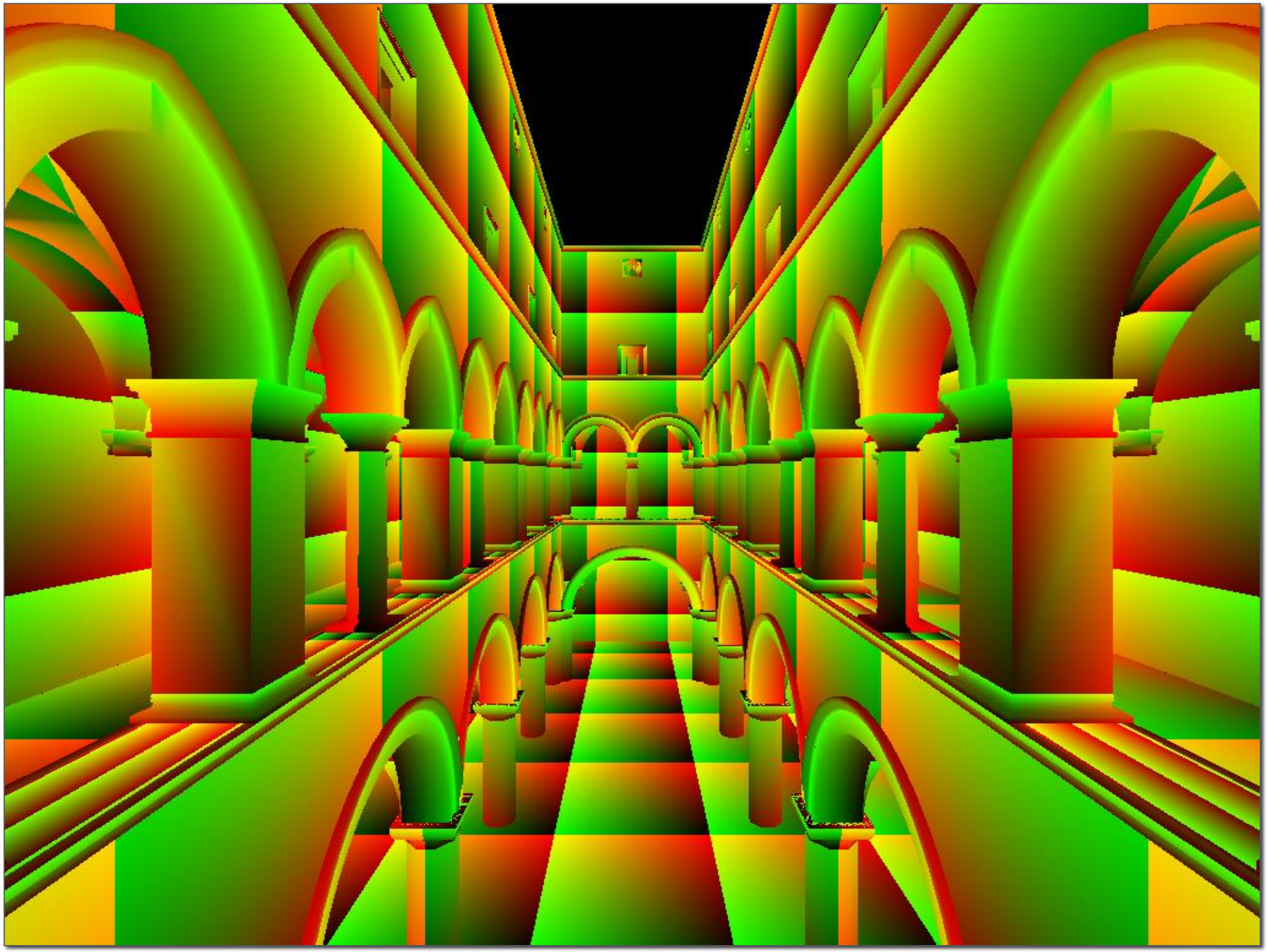






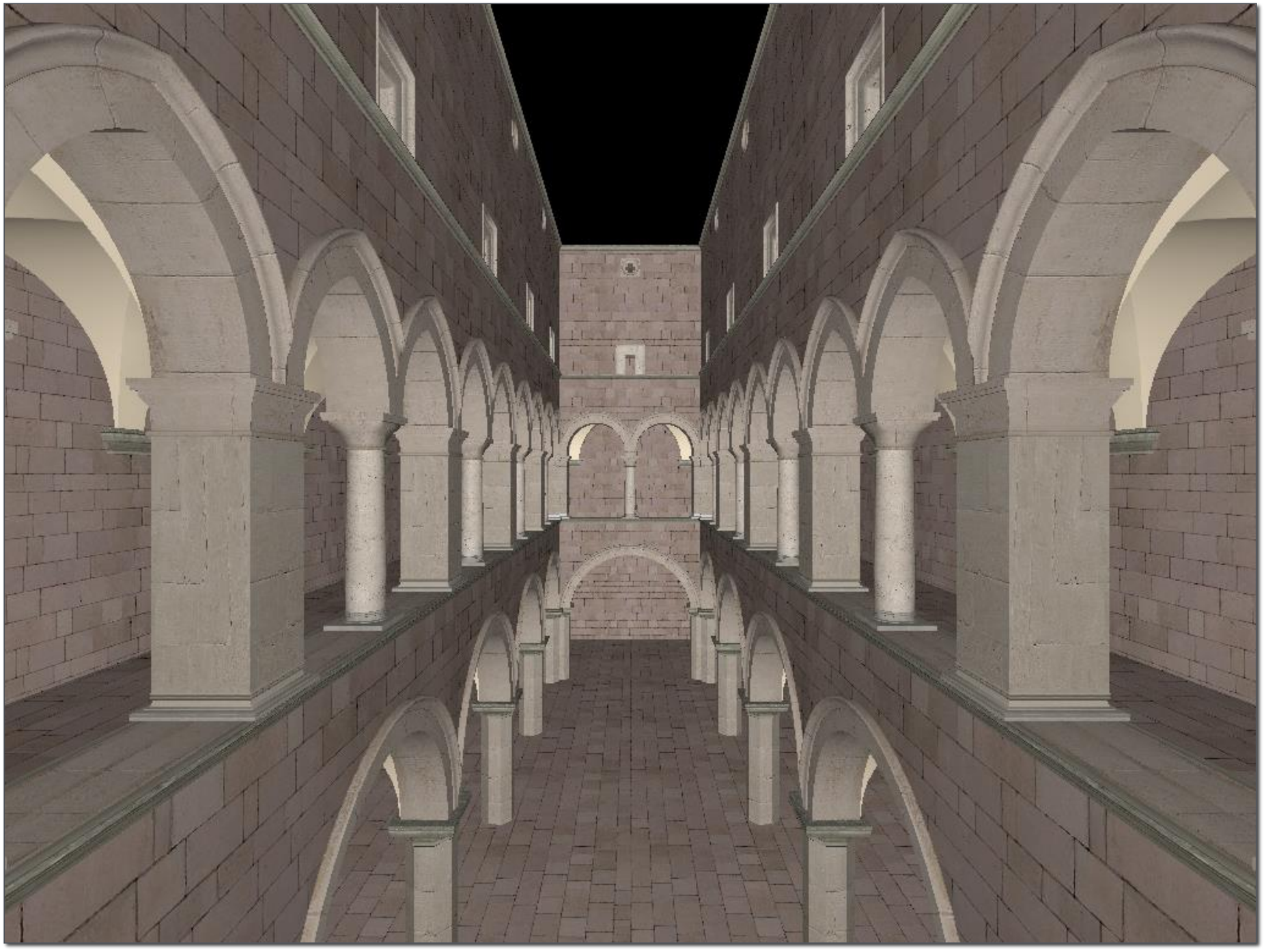
Each triangle “copies” a piece of the image back to the surface.

Another example: Sponza

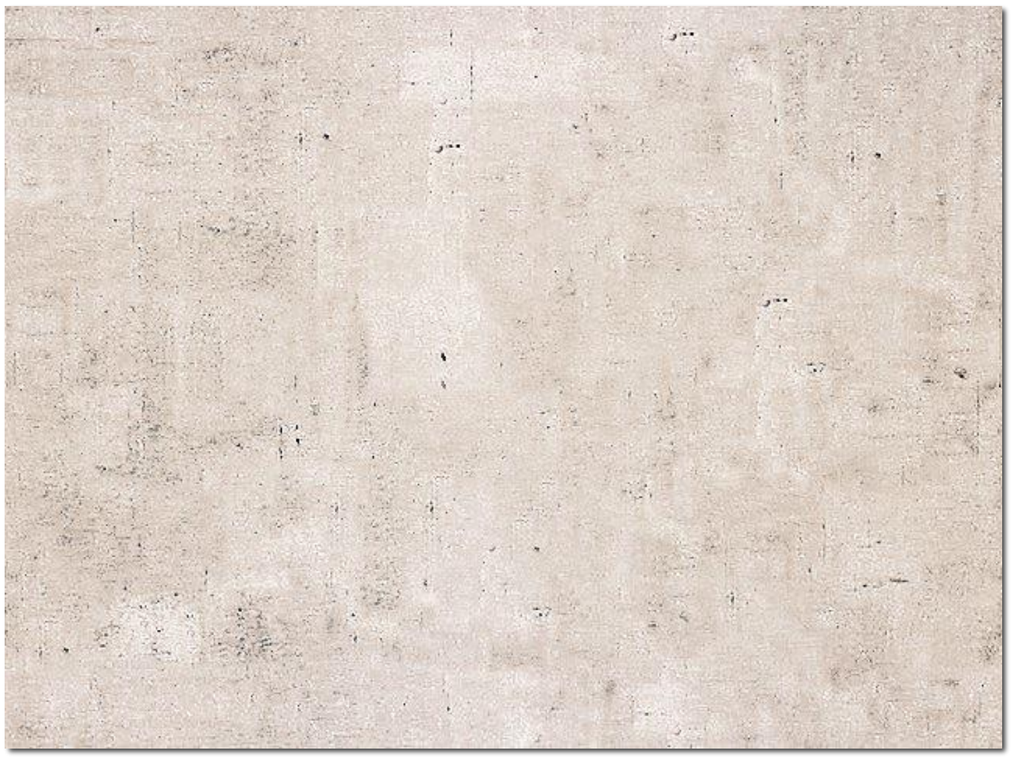


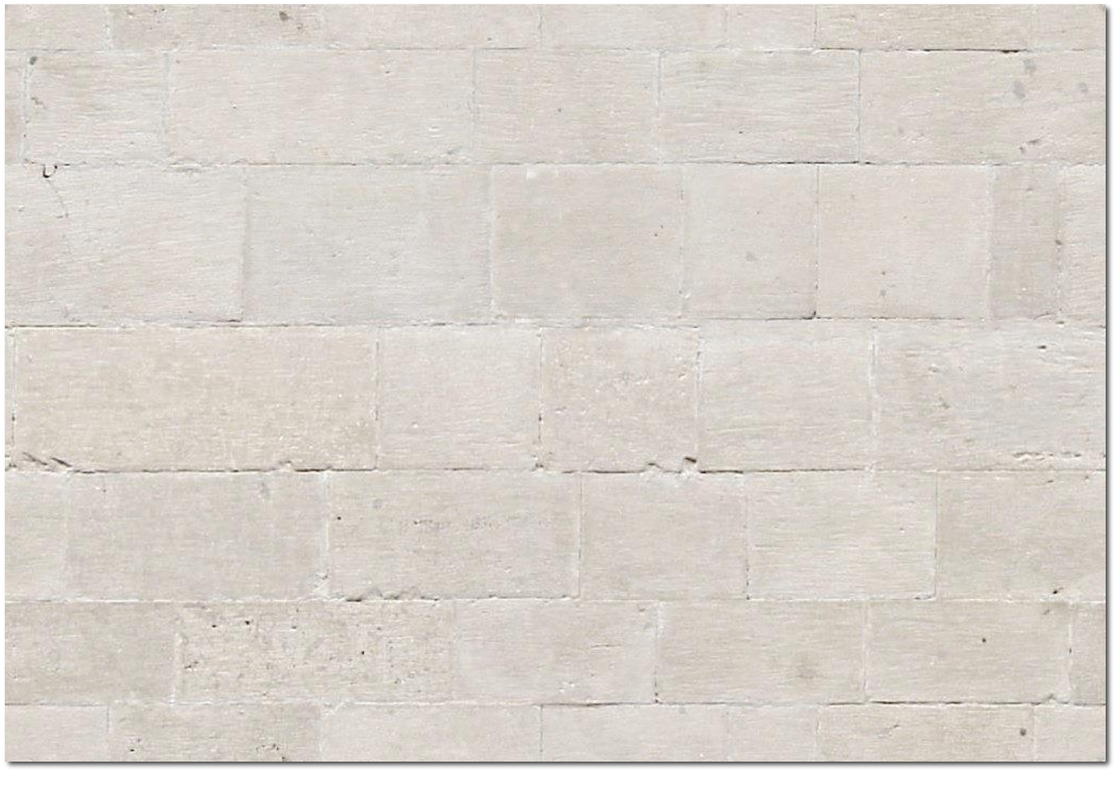
Notice texture coordinates repeat over surface.

Textured Sponza



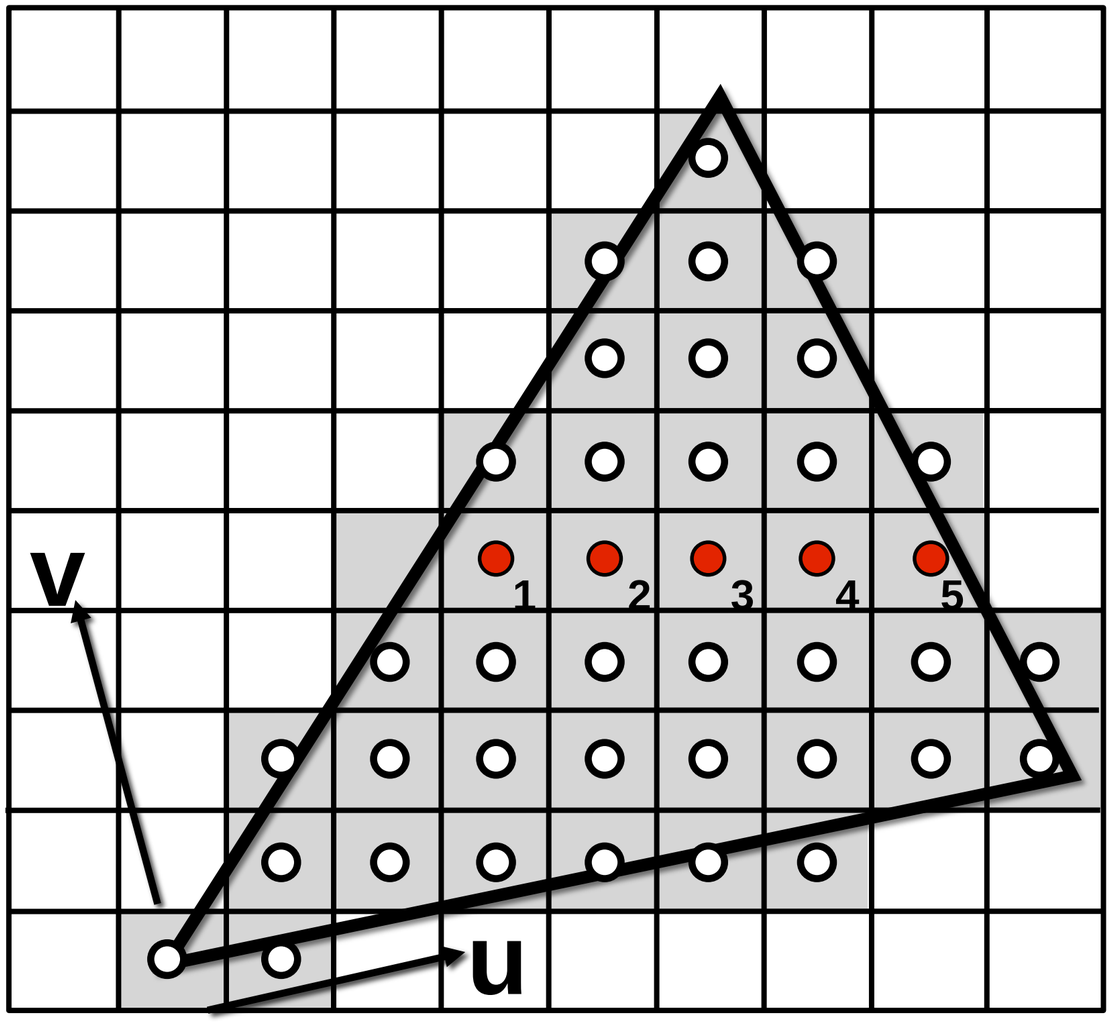
Example textures used in Sponza





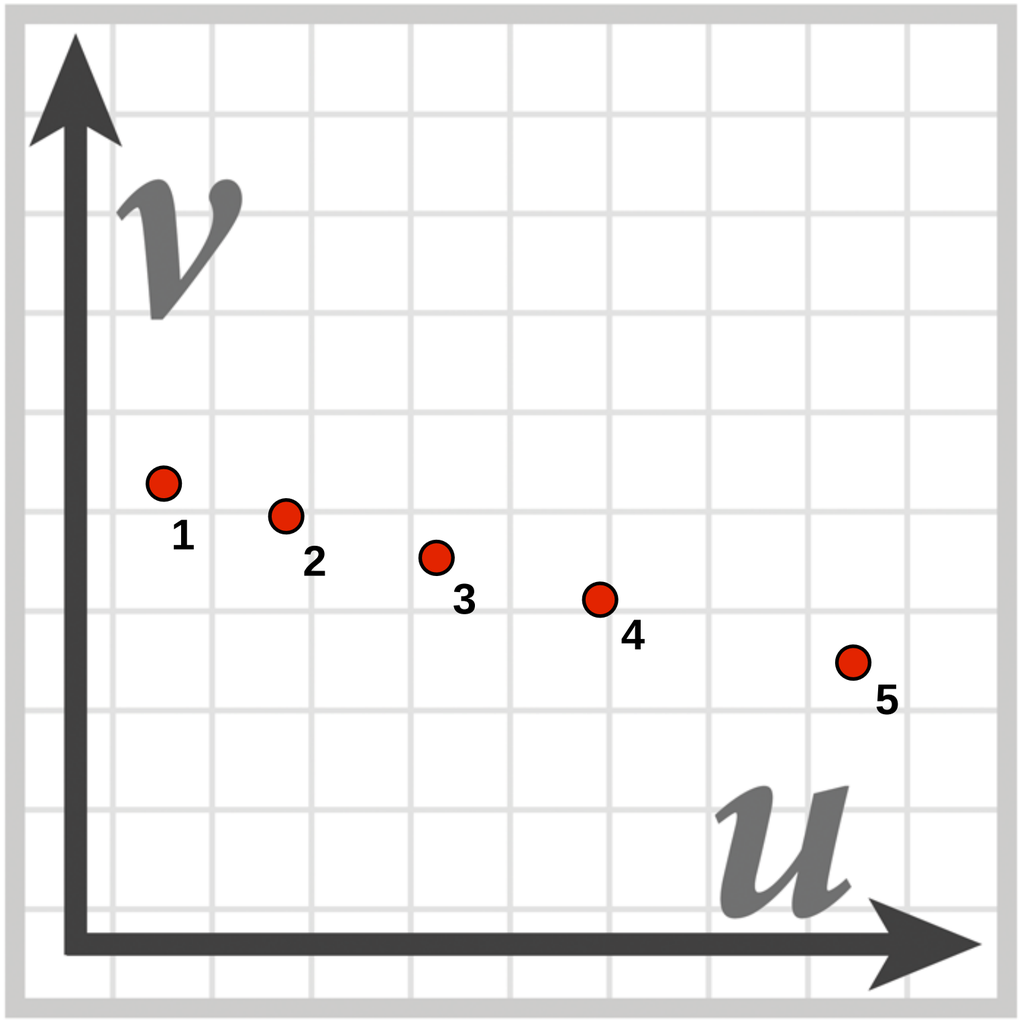
Texture space samples

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

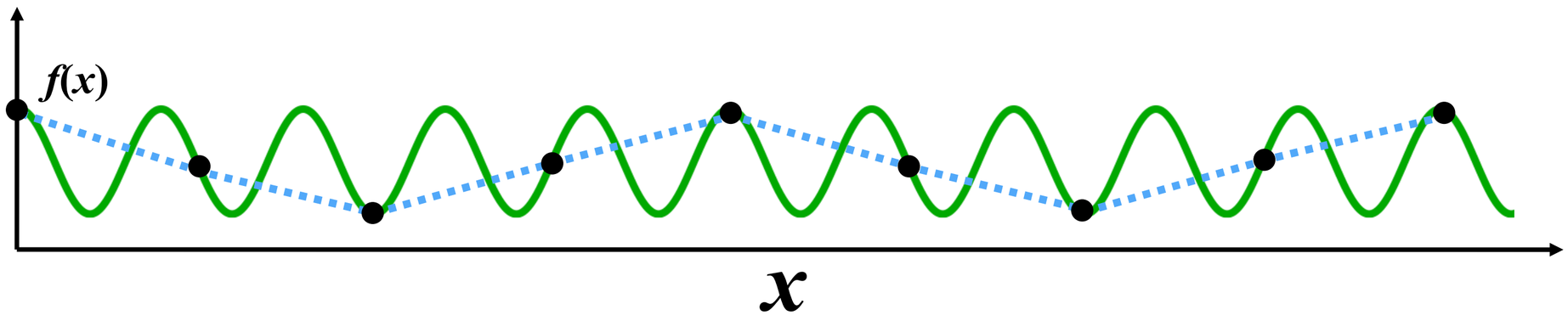


Texture sample positions in texture space (texture function is sampled at these locations)

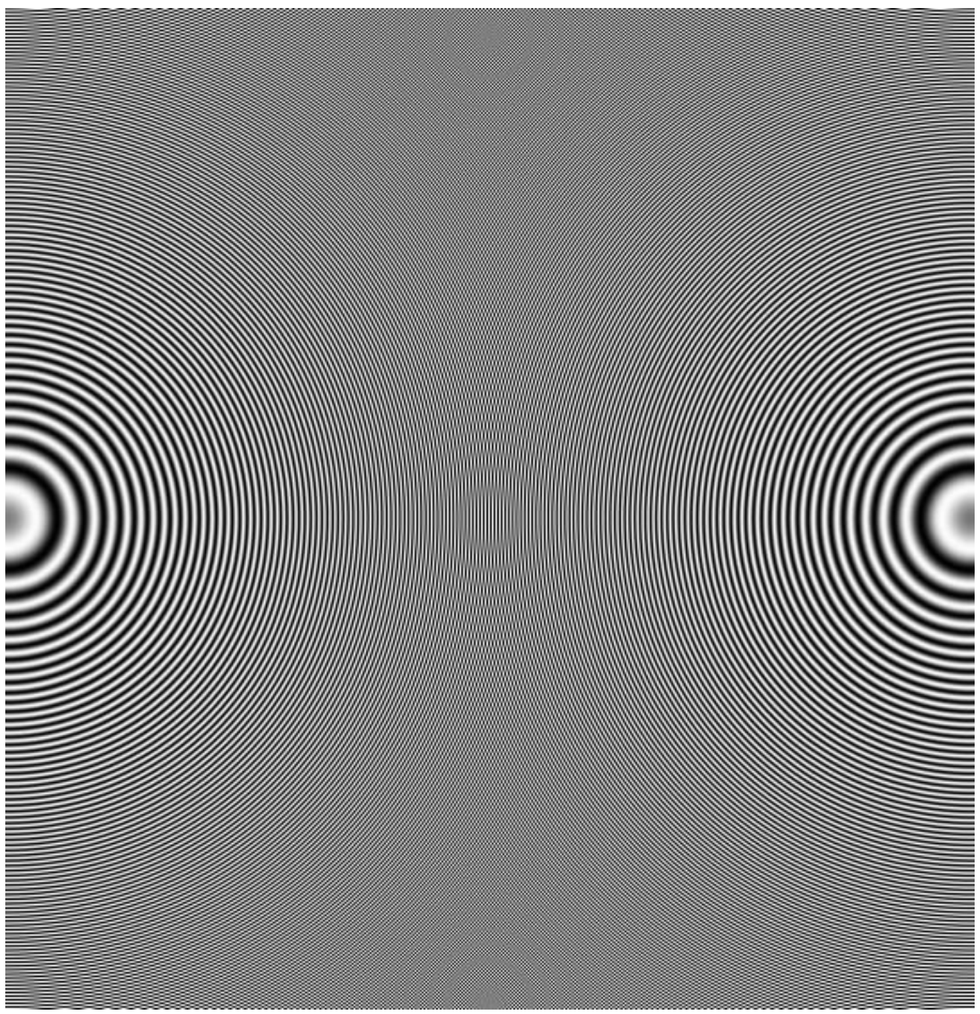
Q: what does it mean that equally-spaced points in screen space move further apart in texture space?

Recall: aliasing

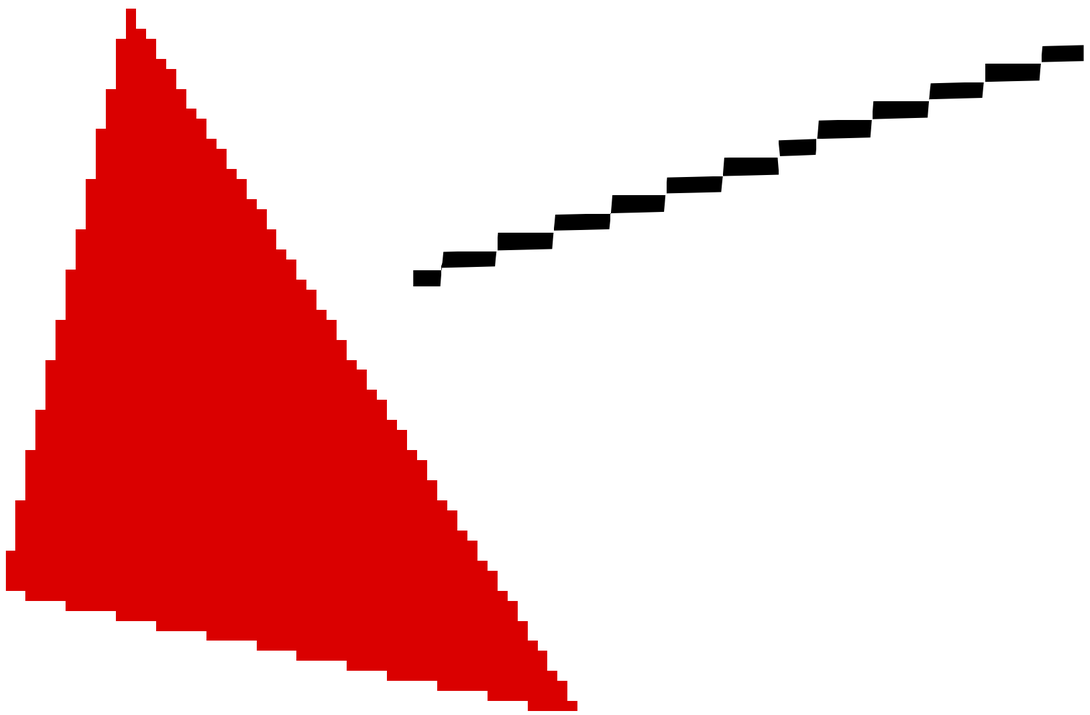
Undersampling a high-frequency signal can result in aliasing



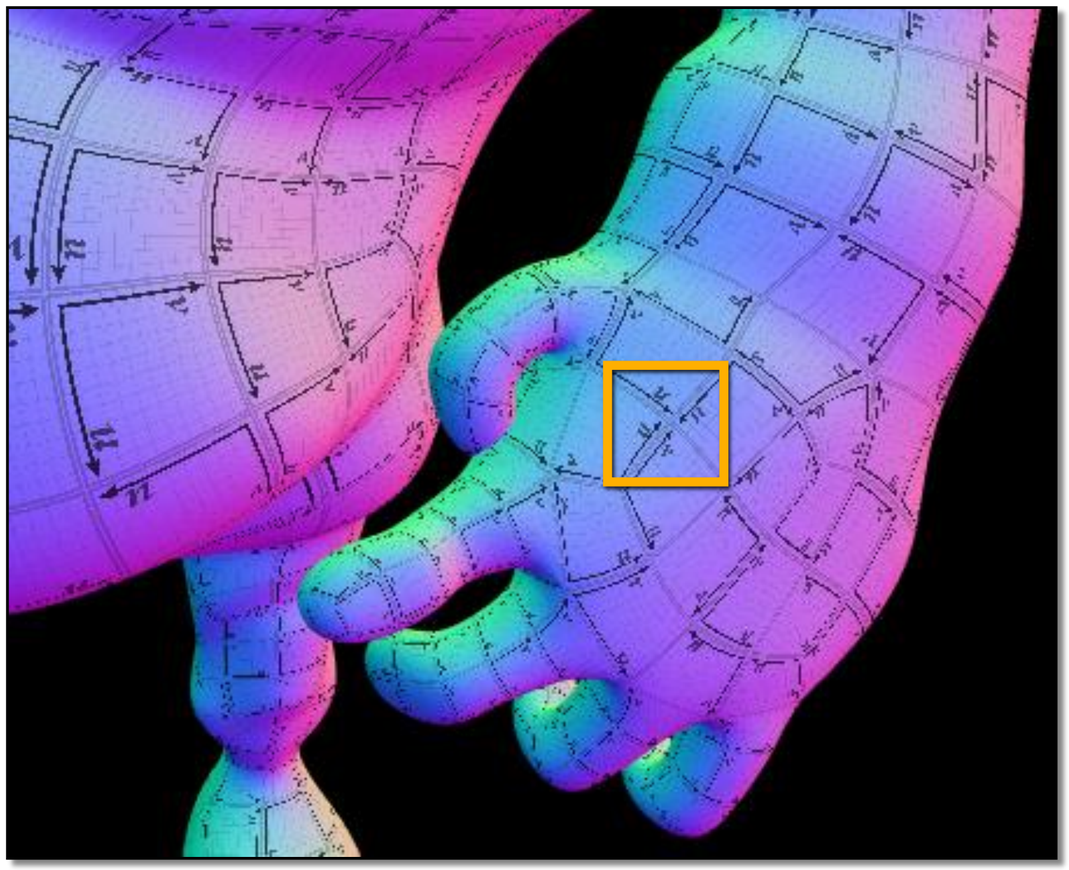
1D example



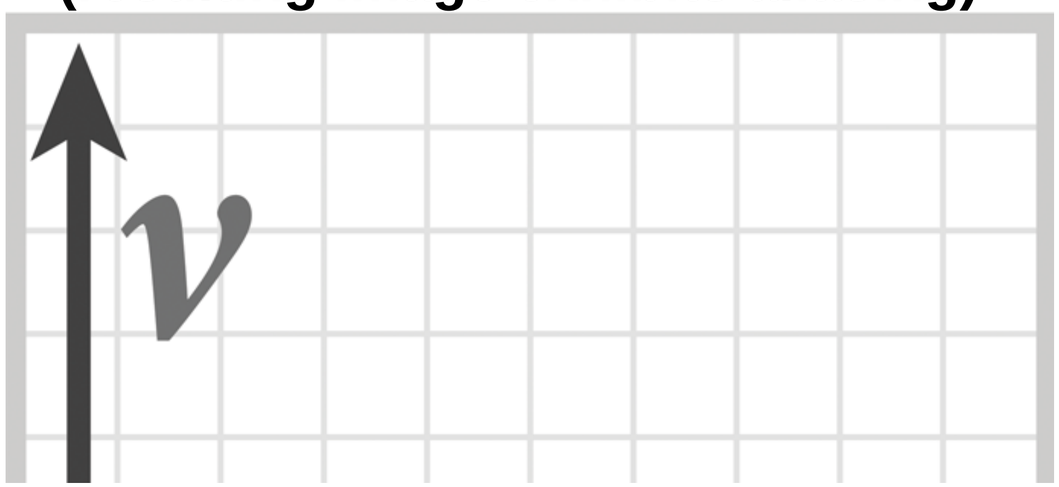
2D examples:
Moiré patterns,

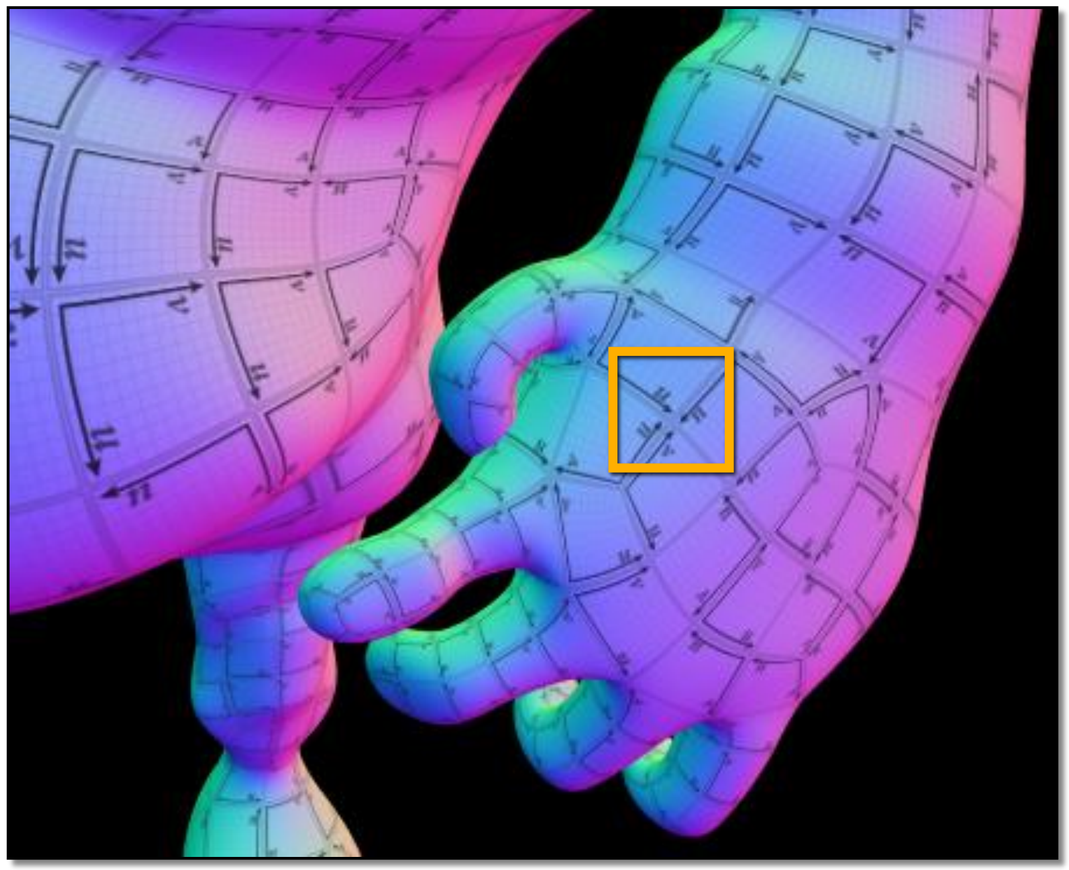


Aliasing due to undersampling texture



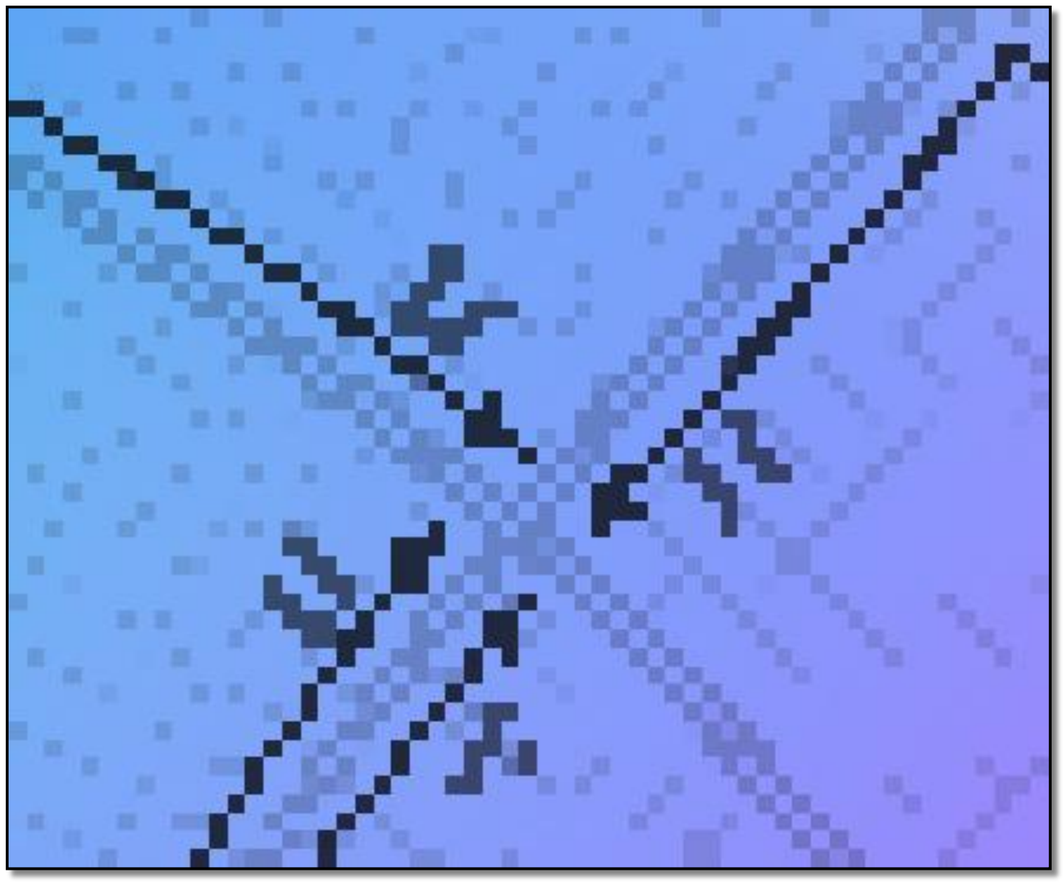
No pre-filtering of texture data
(resulting image exhibits aliasing)



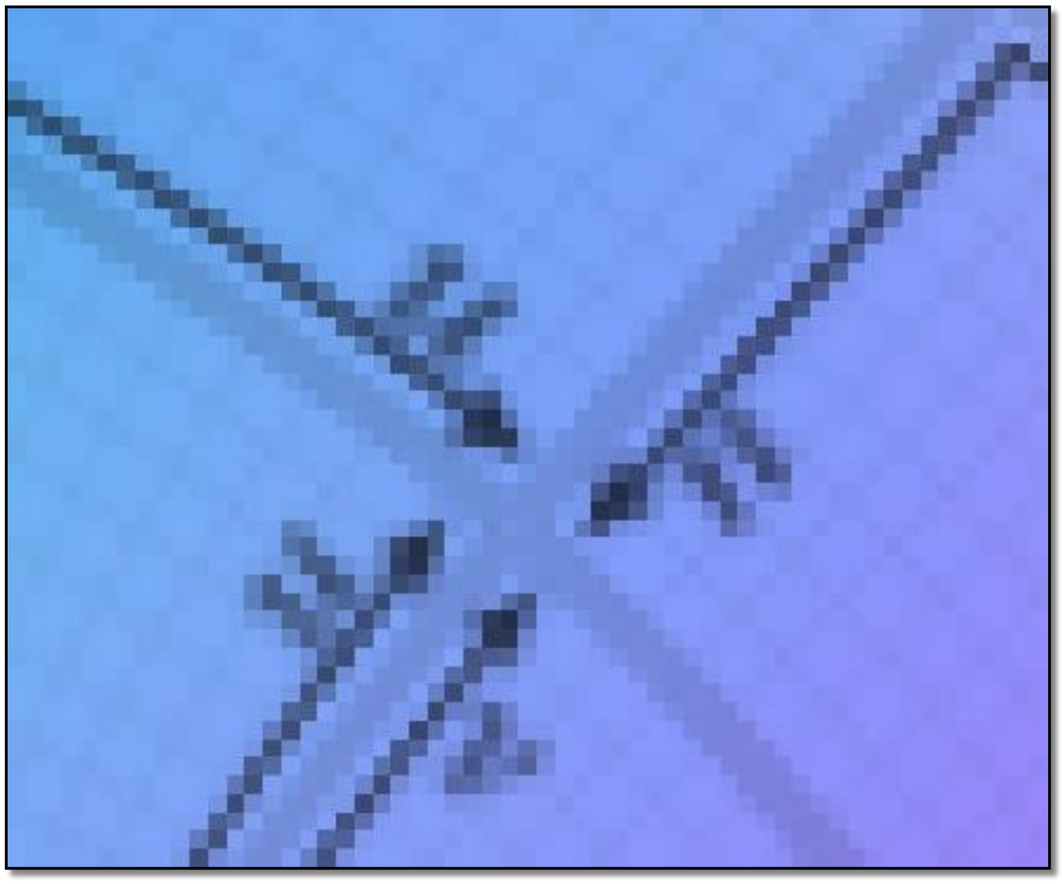


Rendering using pre-filtered
texture data

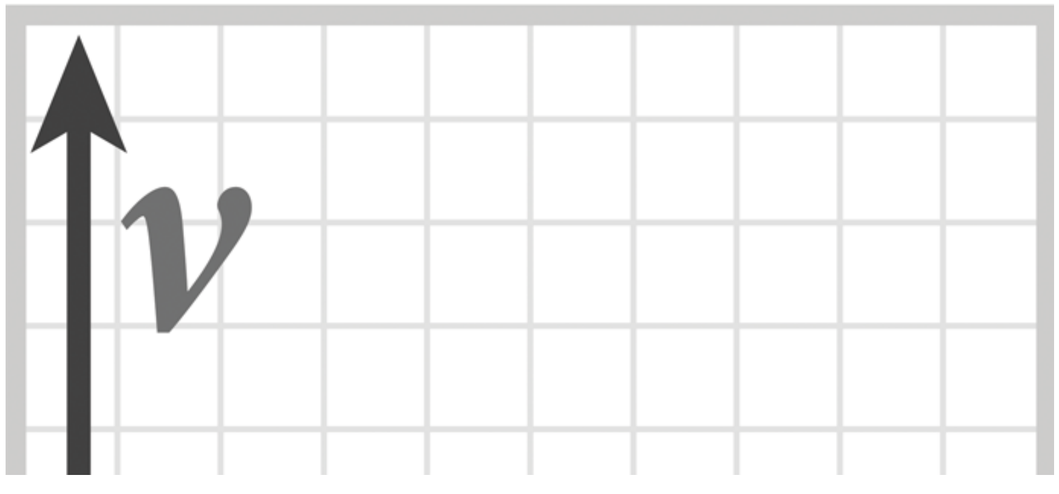
Aliasing due to undersampling (zoom)



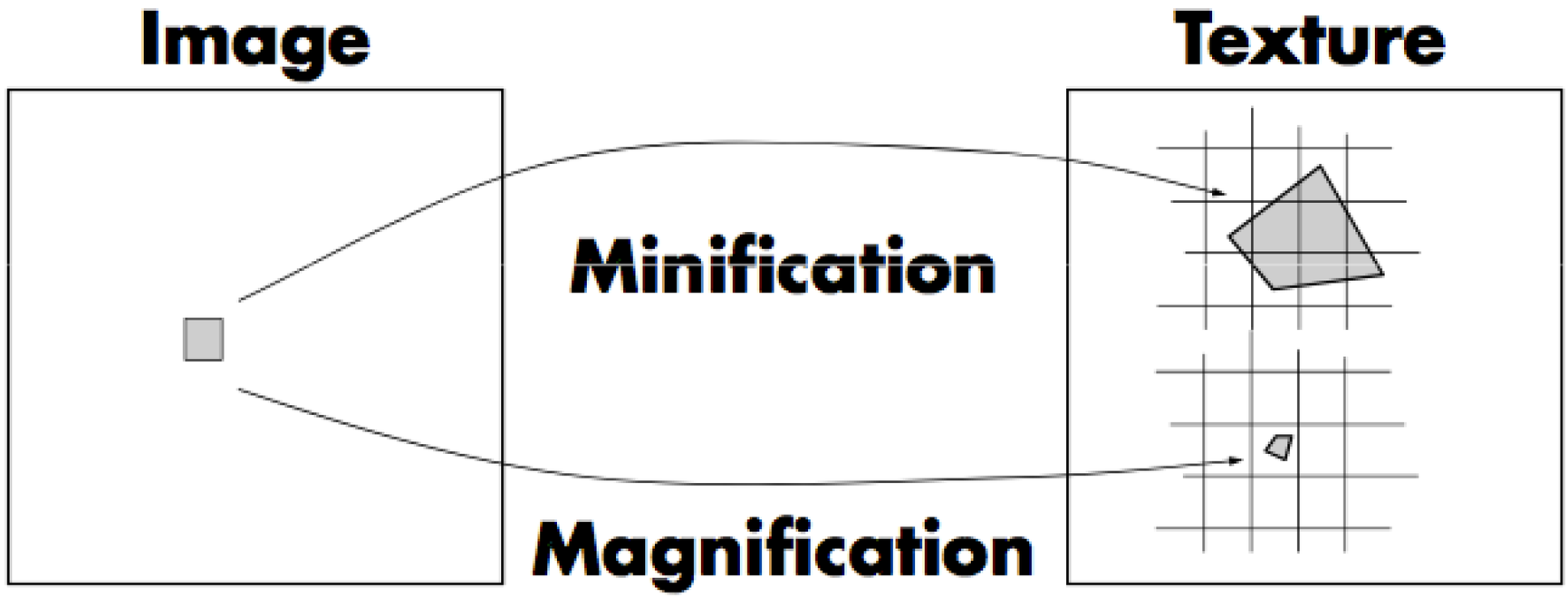
No pre-filtering of texture data



Rendering using pre-filtered texture data



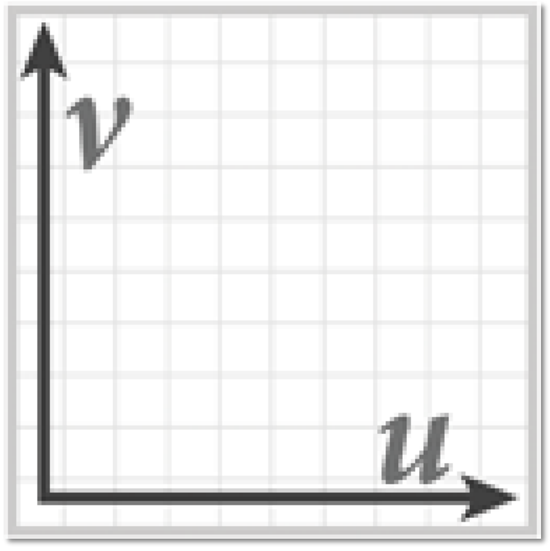
Filtering textures



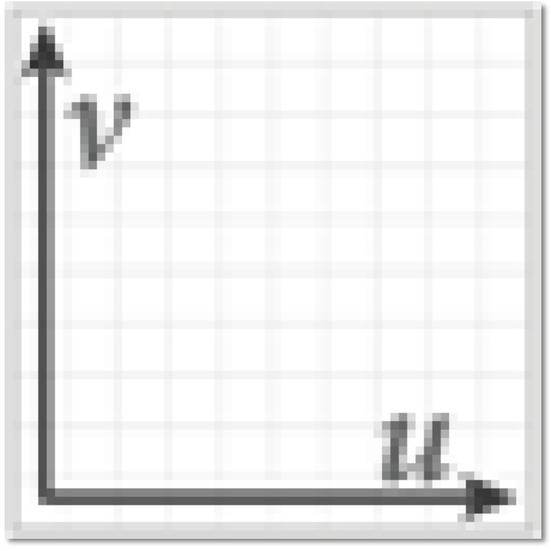
- **Minification:**
  - Area of screen pixel maps to large region of texture (filtering required -- averaging)
  - One texel corresponds to far less than a pixel on screen
  - Example: when scene object is very far away
  - Texture map is too large, it contains more details than screen can display
- **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
  - Texture map is too small

Figure credit: Akeley and Hanrahan

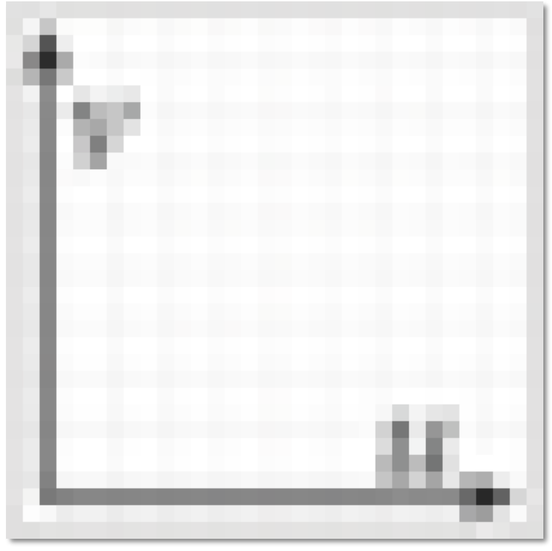
Mipmap (L. Williams 83)



Level 0 = 128x128



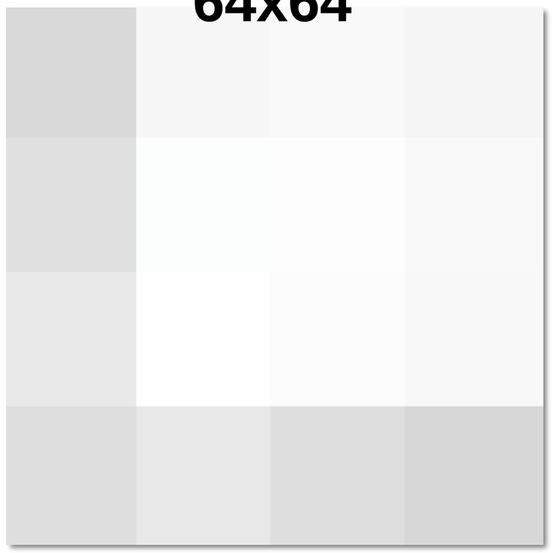
Level 1 = 64x64



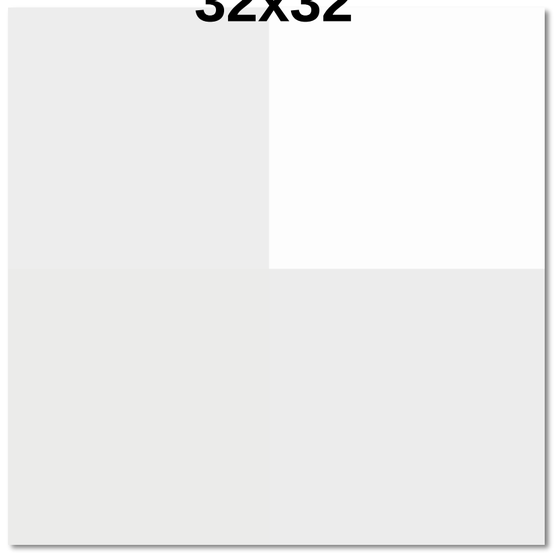
Level 2 = 32x32

Level 3 = 16x16

Level 4 = 8x8



Level 5 = 4x4



Level 6 = 2x2

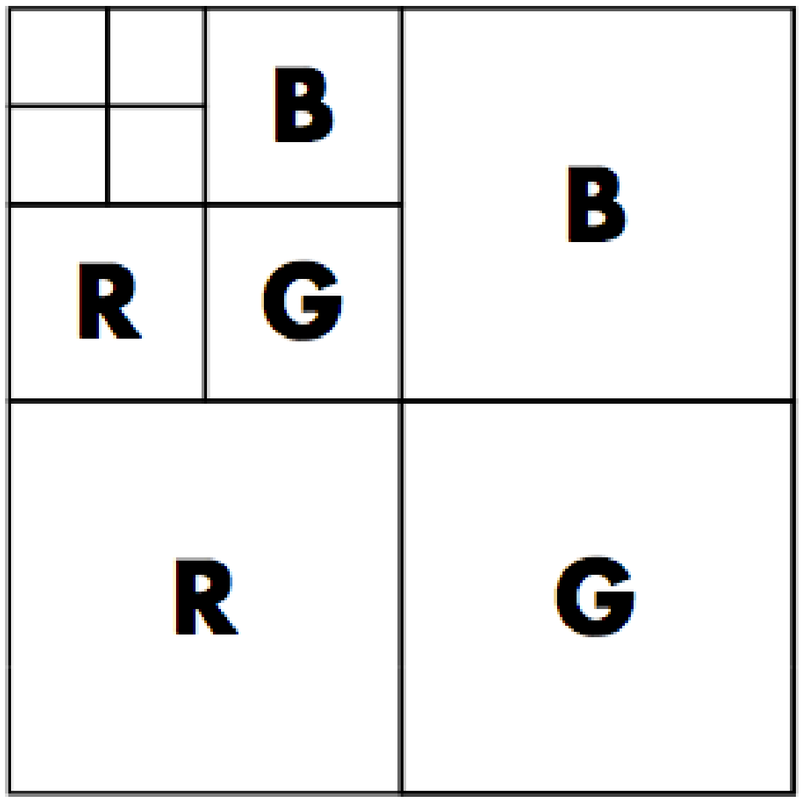
Level 7 = 1x1

Idea: prefilter texture data to remove high frequencies

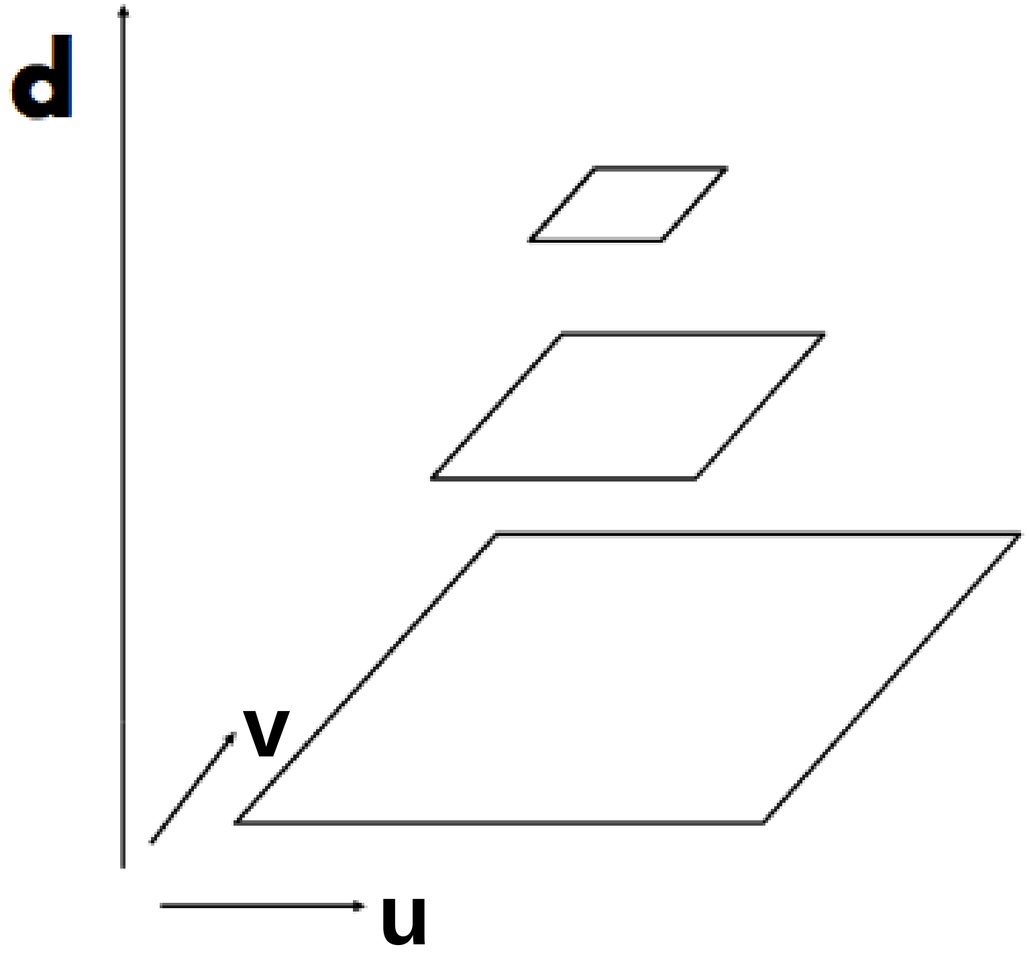
Texels at higher levels store integral of the texture function over a region of texture space (downsampled images)

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

Mipmap (L. Williams 83)



Williams' original proposed mip-map layout



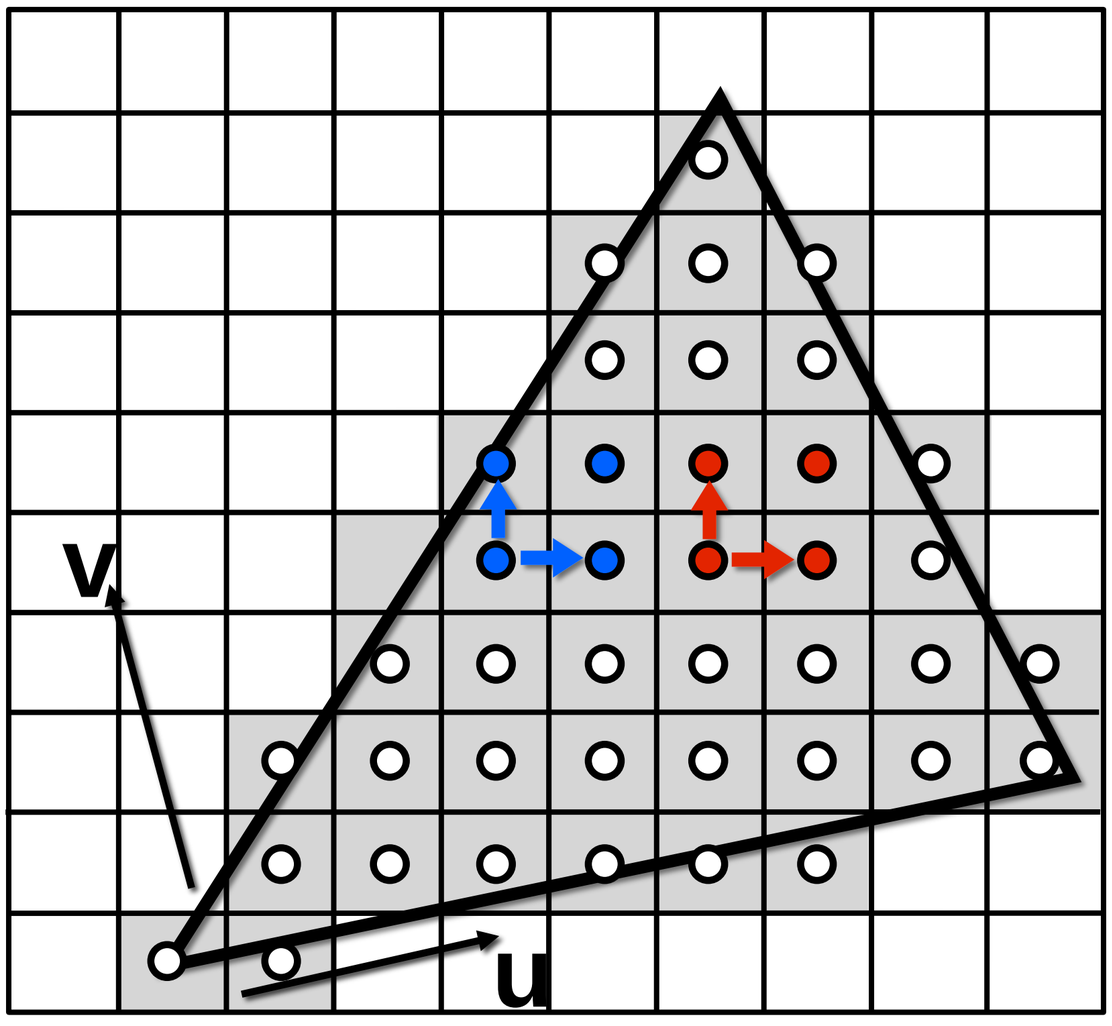
"Mip hierarchy" level = *d*

What is the storage overhead of a mipmap?

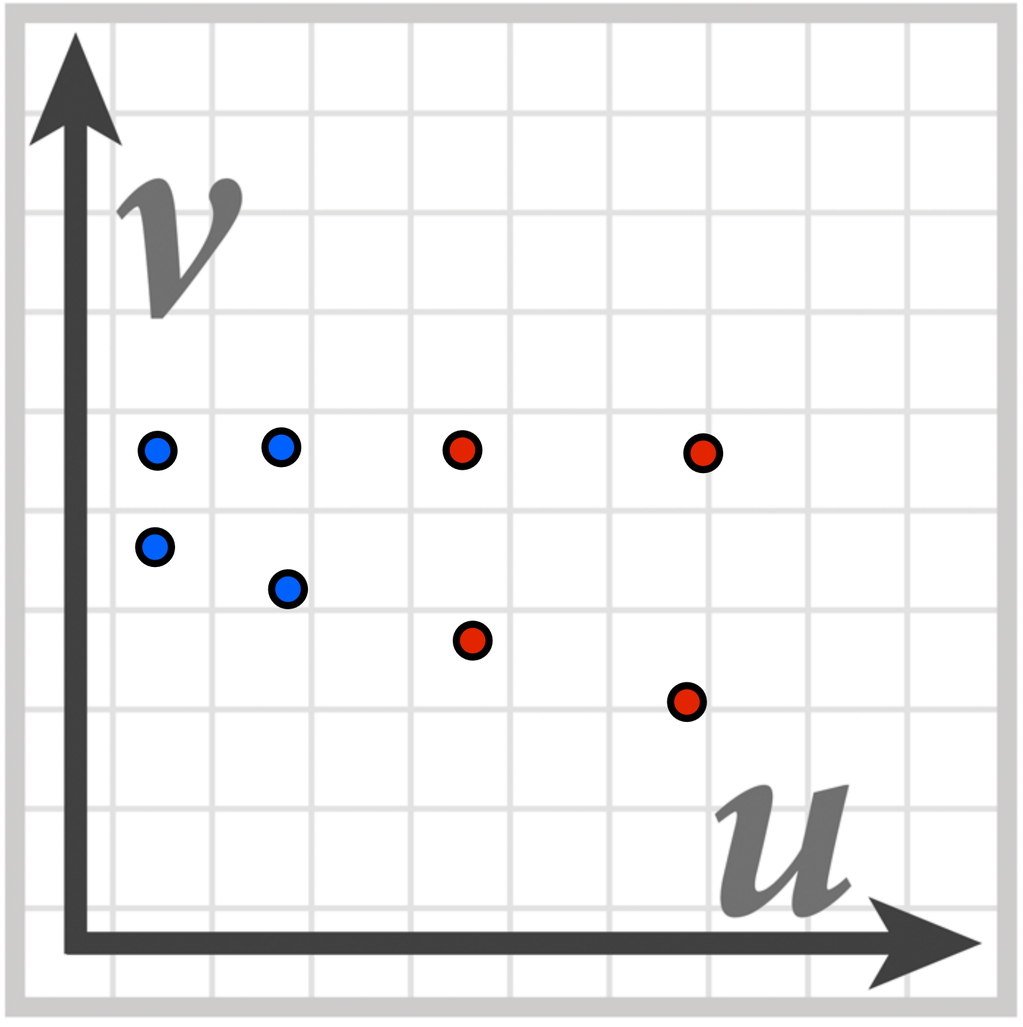
Slide credit: Akeley and Hanrahan

Computing *d*

Compute differences between texture coordinate values of neighboring screen samples



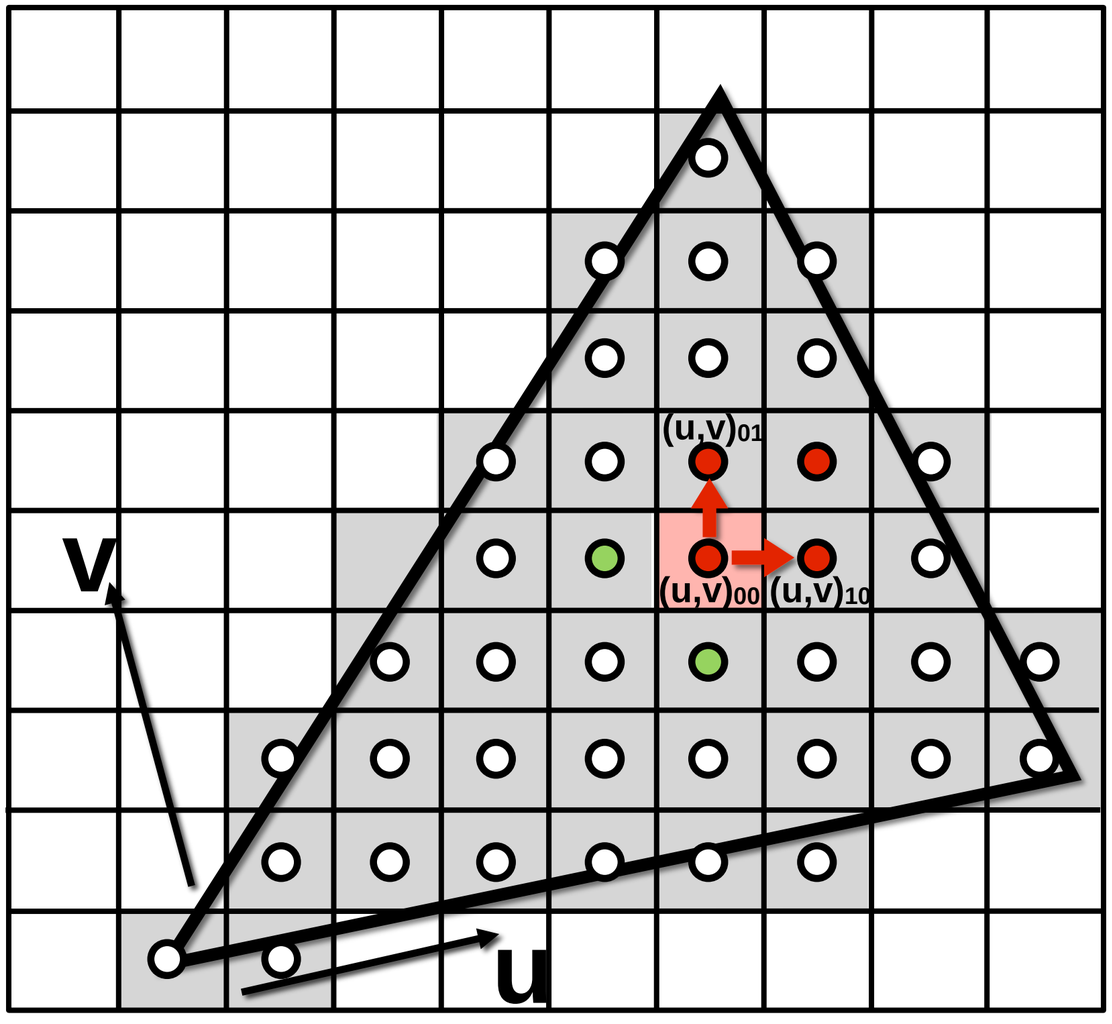
Screen space

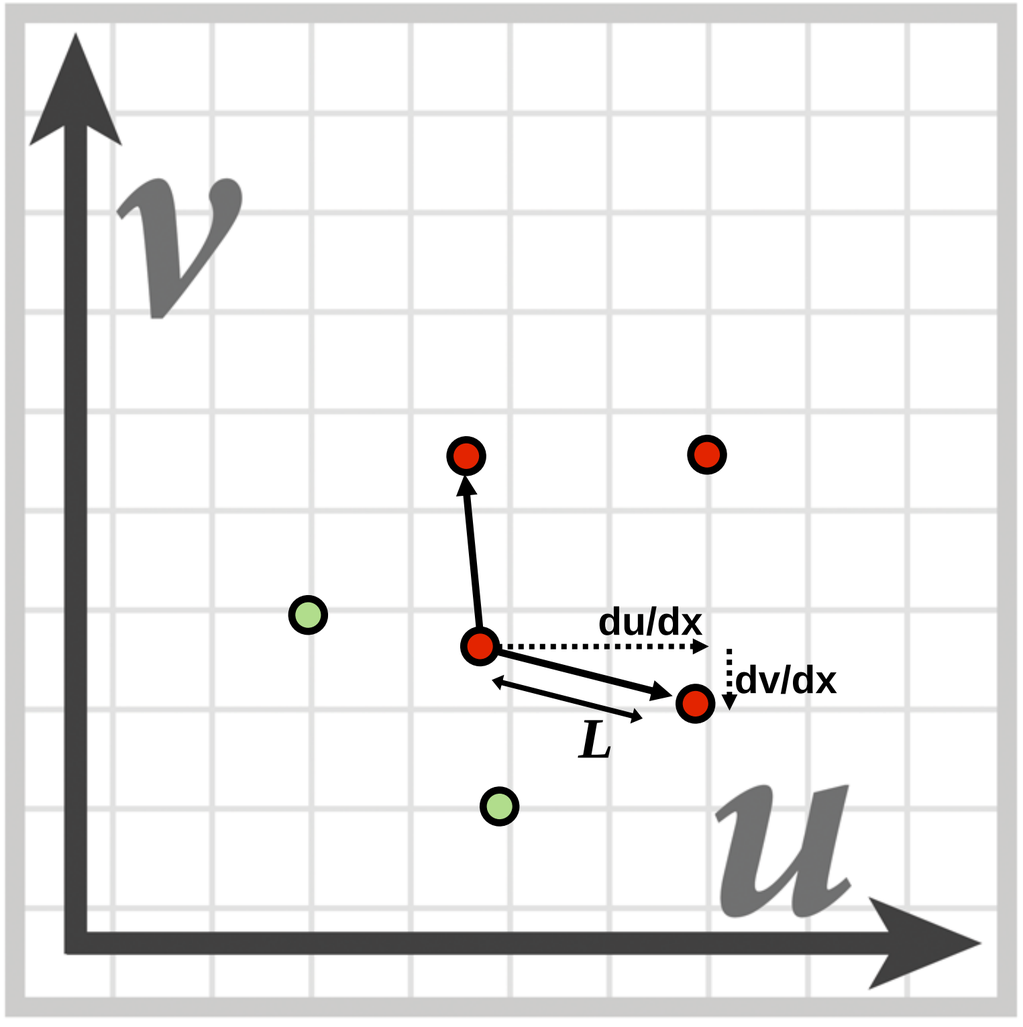


Texture space

Computing *d*

Compute differences between texture coordinate values of neighboring screen samples





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

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

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

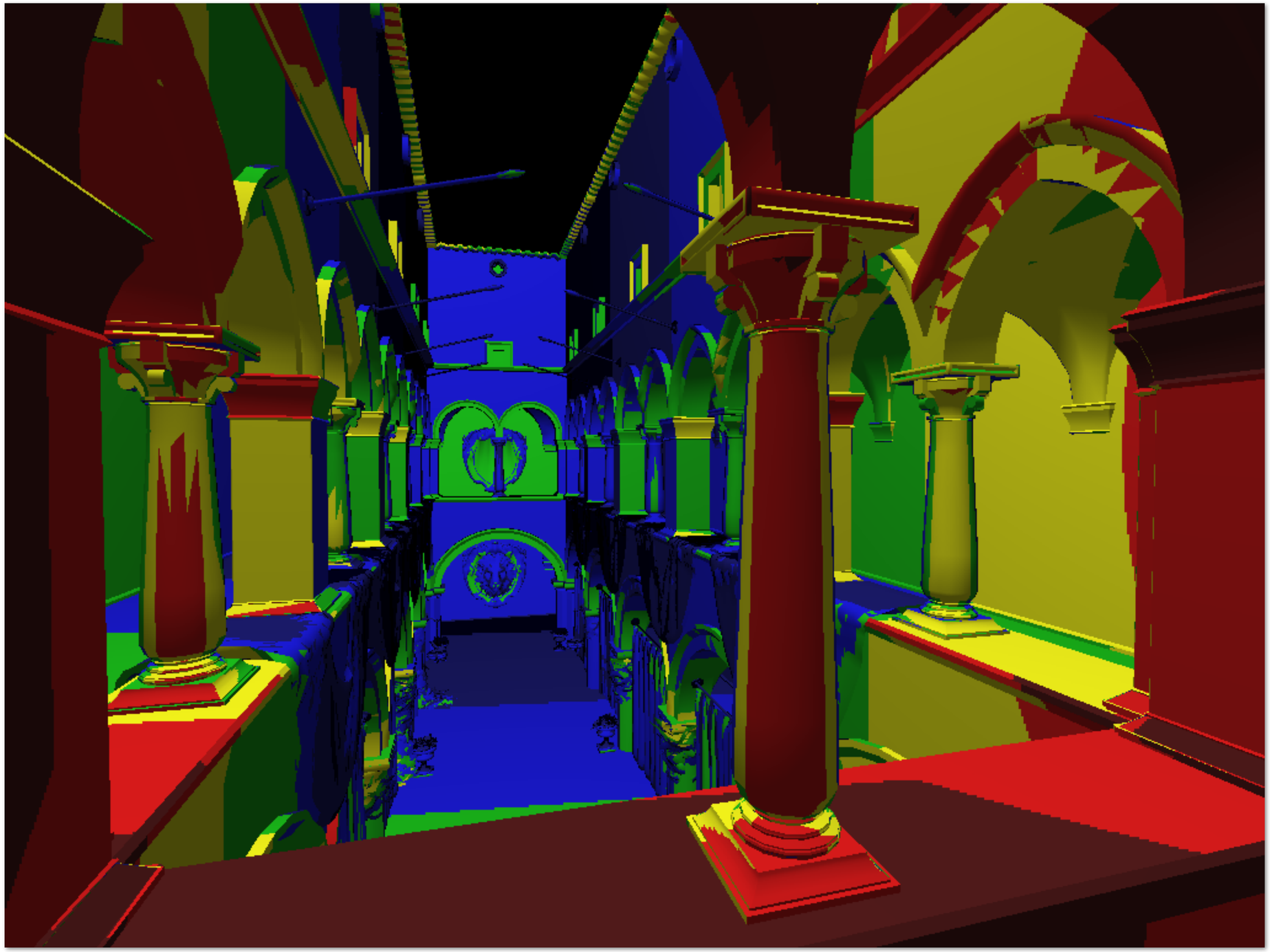
$$\textit{mip-map } d = \log_2 L$$

Sponza (bilinear resampling at level 0)

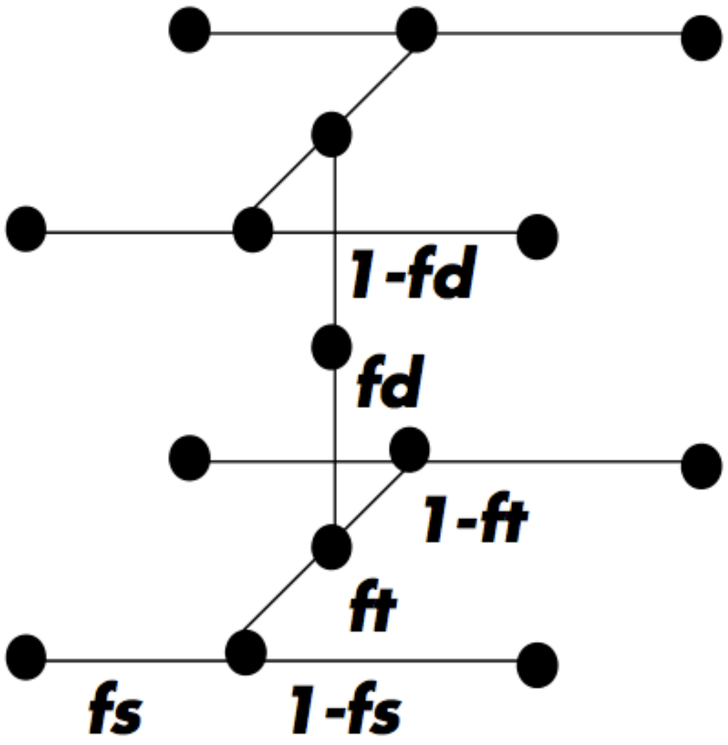
Sponza (bilinear resampling at level 2)

Sponza (bilinear resampling at level 4)

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



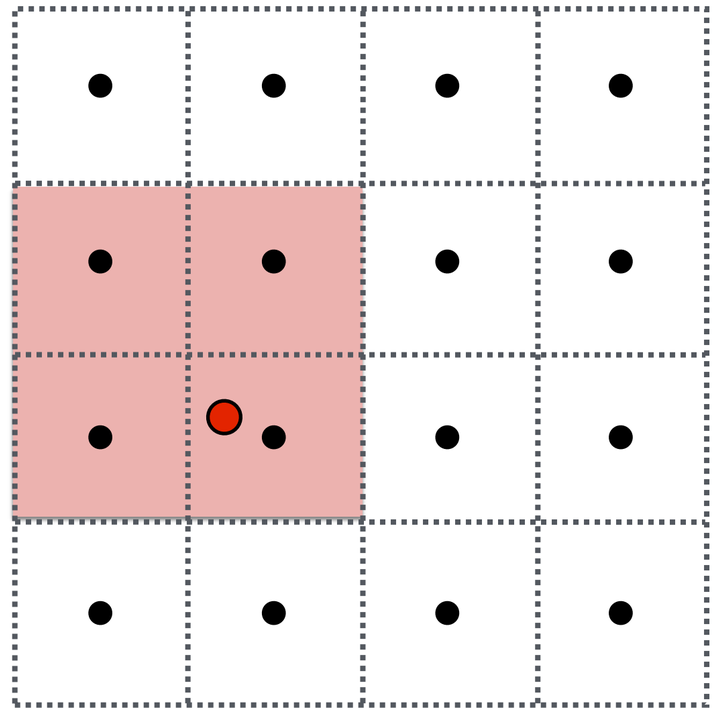
“Tri-linear” filtering



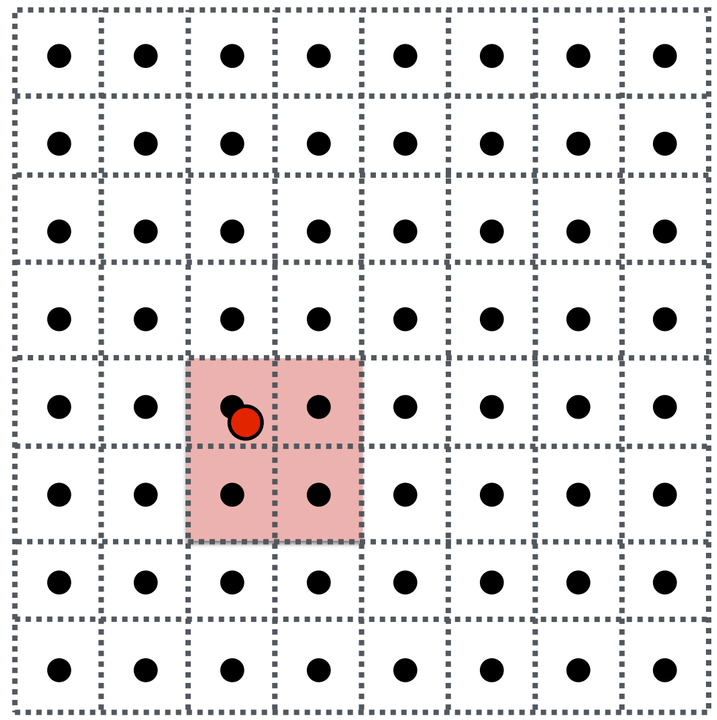
$$lerp(t, v_1, v_2) = v_1 + t(v_2 - v_1)$$

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

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



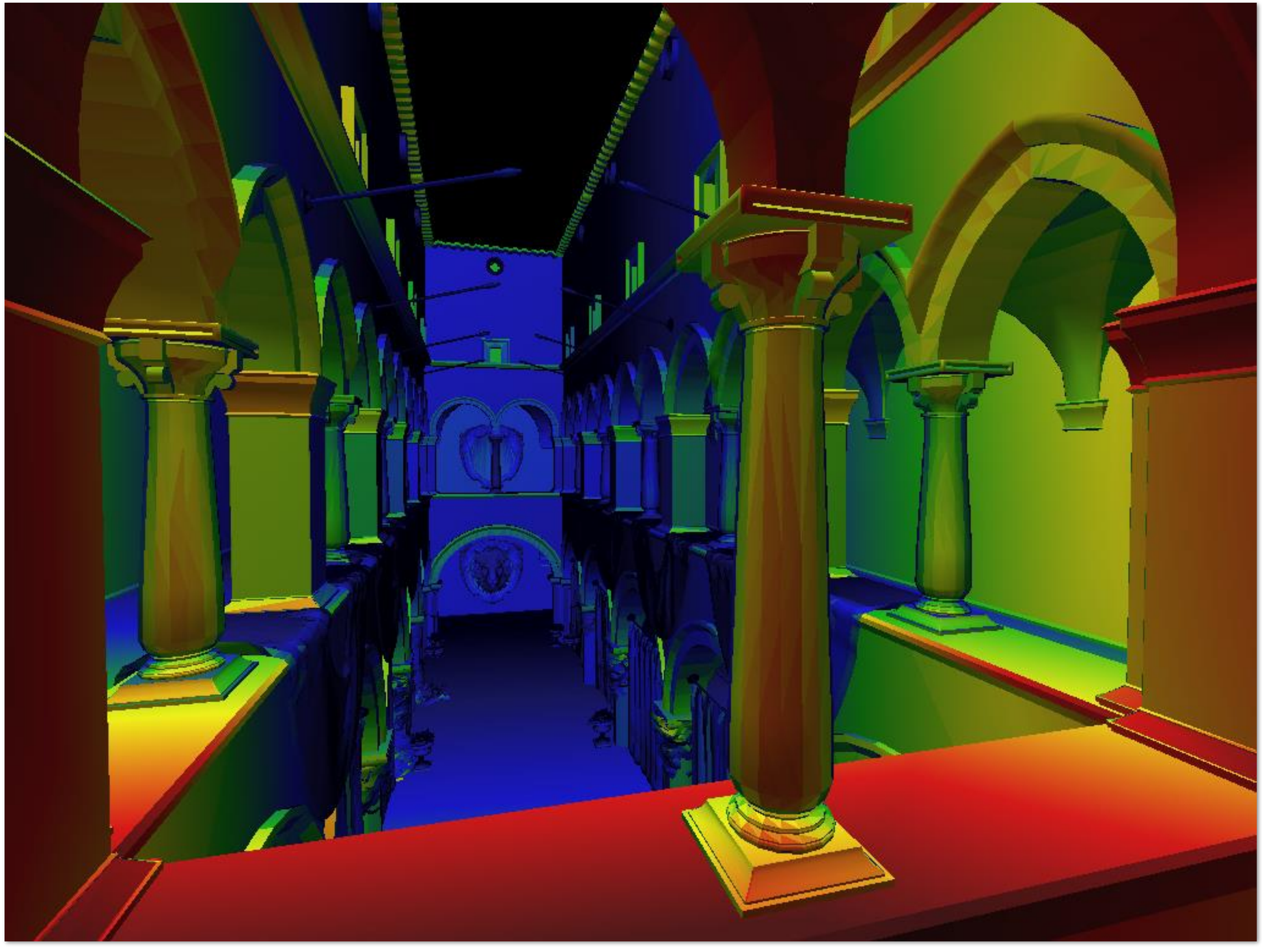
mip-map texels: level d+1



mip-map texels: level d

Figure credit: Akeley and Hanrahan

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



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 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