King Abdullah University of Science and Technology KAUST

# CS 380 - GPU and GPGPU Programming
# Lecture 26: GPU Texturing, Pt. 3

Markus Hadwiger, KAUST

# Reading Assignment #10 (until Nov 11)

Read (required):

- Interpolation for Polygon Texture Mapping and Shading, Paul Heckbert and Henry Moreton

  `https://www.ri.cmu.edu/publications/interpolation-for-polygon-texture-mapping-and-shading/`

- Homogeneous Coordinates

  `https://en.wikipedia.org/wiki/Homogeneous_coordinates`

# Next Lectures

Lecture 27: Thu,  Nov 7: Vulkan tutorial #2

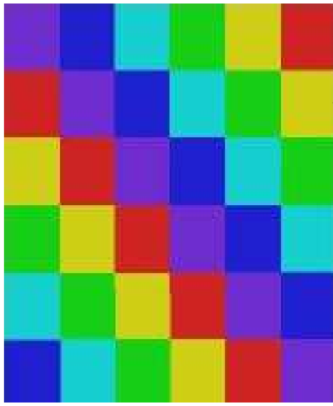
Lecture 28: Mon, Nov 11: 10:00-11:30 (on Zoom)

Lecture 29: Thu,  Nov 14: 10:00-11:30 (on Zoom)

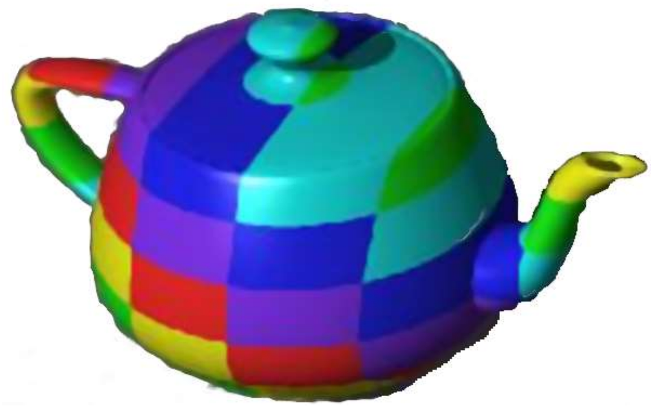
Lecture 30: Mon,  Nov 18: Quiz #3

# GPU Texturing

# Texturing: General Approach



Texels



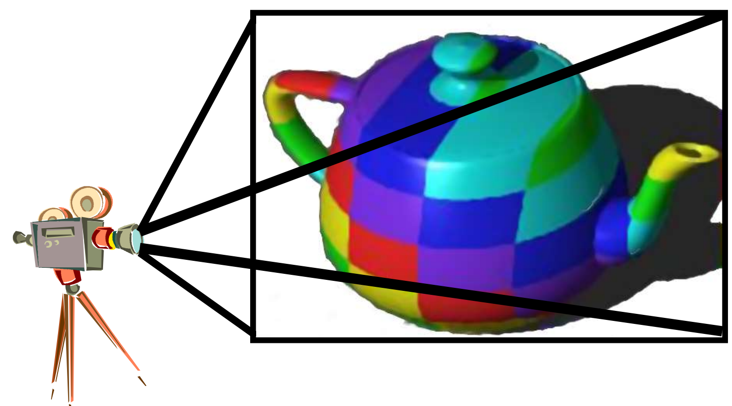
**Texture space** *(u,v)*

**Object space** $(x_O, y_O, z_O)$

**Image Space** $(x_I, y_I)$

**Parametrization**

**Rendering (Projection etc.)**

# Texture Mapping

2D (3D) Texture Space

  ↓ Texture Transformation

2D Object Parameters

  ↓ Parameterization

3D Object Space

  ↓ Model Transformation

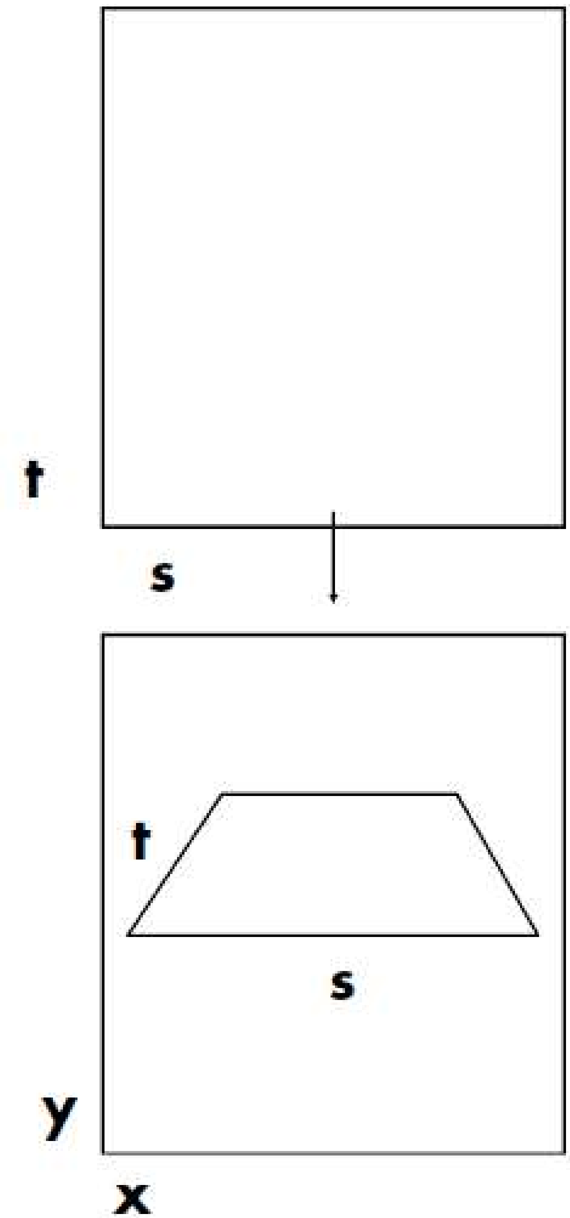
3D World Space

  ↓ Viewing Transformation

3D Camera Space

  ↓ Projection

2D Image Space




Kurt Akeley, Pat Hanrahan

# Texture Mapping Polygons

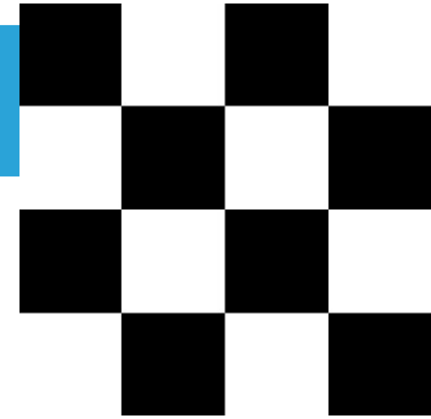
Forward transformation: linear projective map

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} s \\ t \\ r \end{bmatrix}$$

Backward transformation: linear projective map

$$\begin{bmatrix} s \\ t \\ r \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix}^{-1} \begin{bmatrix} x \\ y \\ w \end{bmatrix}$$

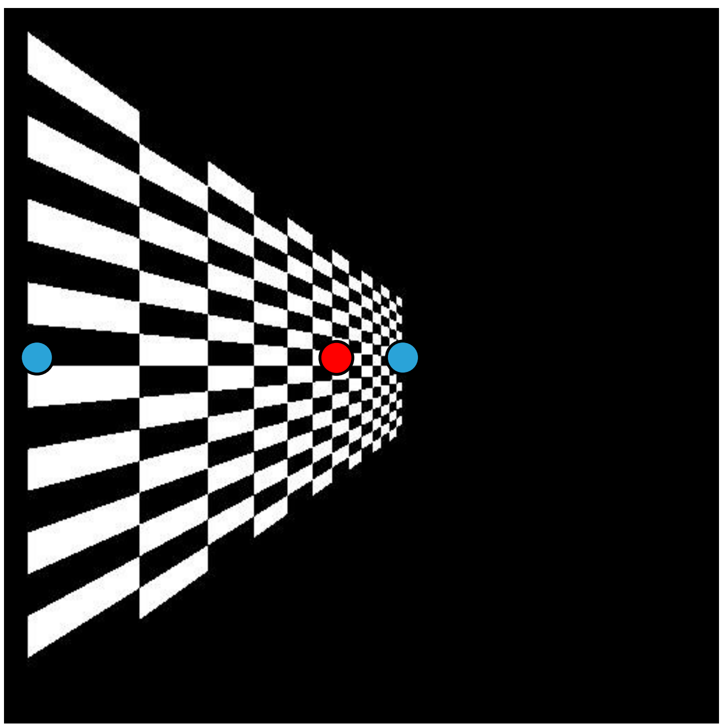
# Perspective Texture Mapping

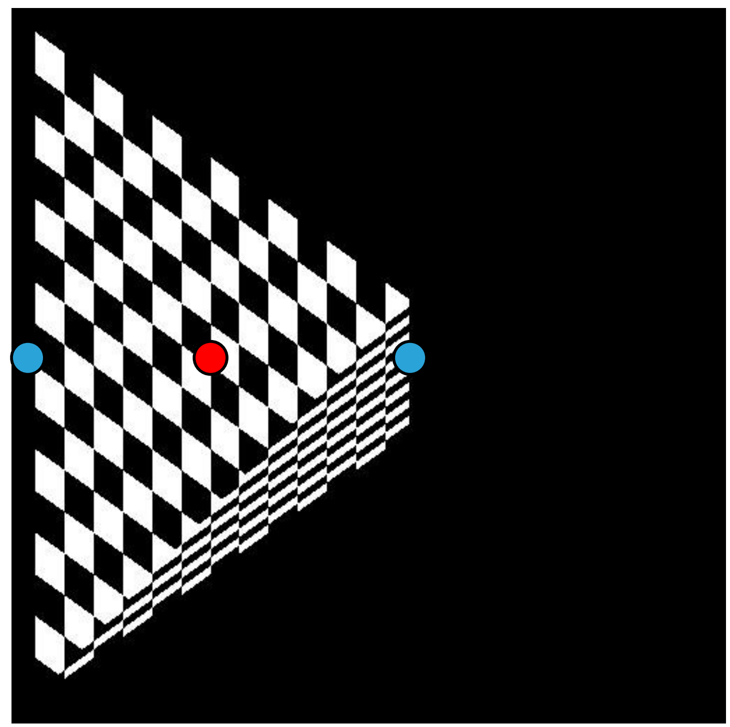


linear interpolation in object space

$$\frac{ax_1 + bx_2}{aw_1 + bw_2} \neq a\frac{x_1}{w_1} + b\frac{x_2}{w_2}$$

linear interpolation in screen space





$a = b = 0.5$

# Perspective-correct linear interpolation

Only projected values interpolate correctly, so project $A$

- Linearly interpolate $A_1/w_1$ and $A_2/w_2$

Also interpolate $1/w_1$ and $1/w_2$

- These also interpolate linearly in screen space

Divide interpolants at each sample point to recover $A$

- $(A/w) / (1/w) = A$
- Division is expensive (more than add or multiply), so
  - Recover $w$ for the sample point (reciprocate), and
  - Multiply each projected attribute by $w$

Barycentric triangle parameterization:

$$A = \frac{aA_1/w_1 + bA_2/w_2 + cA_3/w_3}{a/w_1 + b/w_2 + c/w_3} \qquad a + b + c = 1$$

Kurt Akeley, Pat Hanrahan

# Perspective-Correct Interpolation Recipe

$$r_i(x,y) = \frac{r_i(x,y)/w(x,y)}{1/w(x,y)}$$

(1) Associate a record containing the $n$ parameters of interest $(r_1, r_2, \cdots, r_n)$ with each vertex of the polygon.

(2) For each vertex, transform object space coordinates to homogeneous screen space using $4 \times 4$ object to screen matrix, yielding the values $(xw, yw, zw, w)$.

(3) Clip the polygon against plane equations for each of the six sides of the viewing frustum, linearly interpolating all the parameters when new vertices are created.

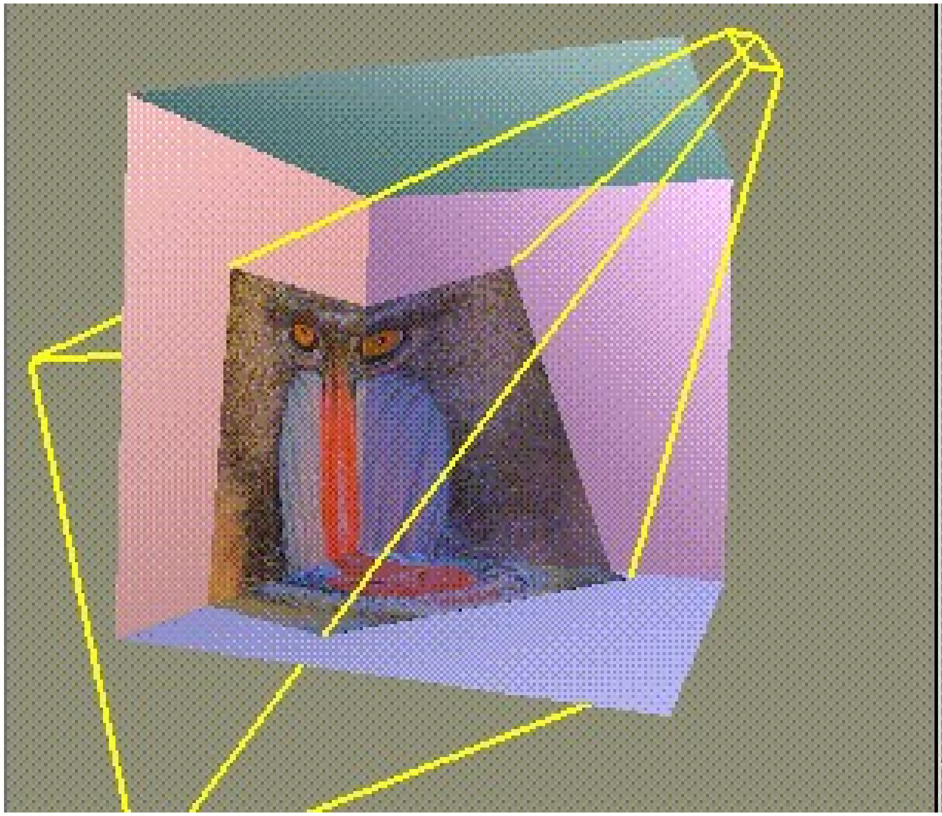
(4) At each vertex, divide the homogeneous screen coordinates, the parameters $r_i$, and the number 1 by $w$ to construct the variable list $(x, y, z, s_1, s_2, \cdots, s_{n+1})$, where $s_i = r_i/w$ for $i \le n$, $s_{n+1} = 1/w$.

(5) Scan convert in screen space by linear interpolation of all parameters, at each pixel computing $r_i = s_i/s_{n+1}$ for each of the $n$ parameters; use these values for shading.

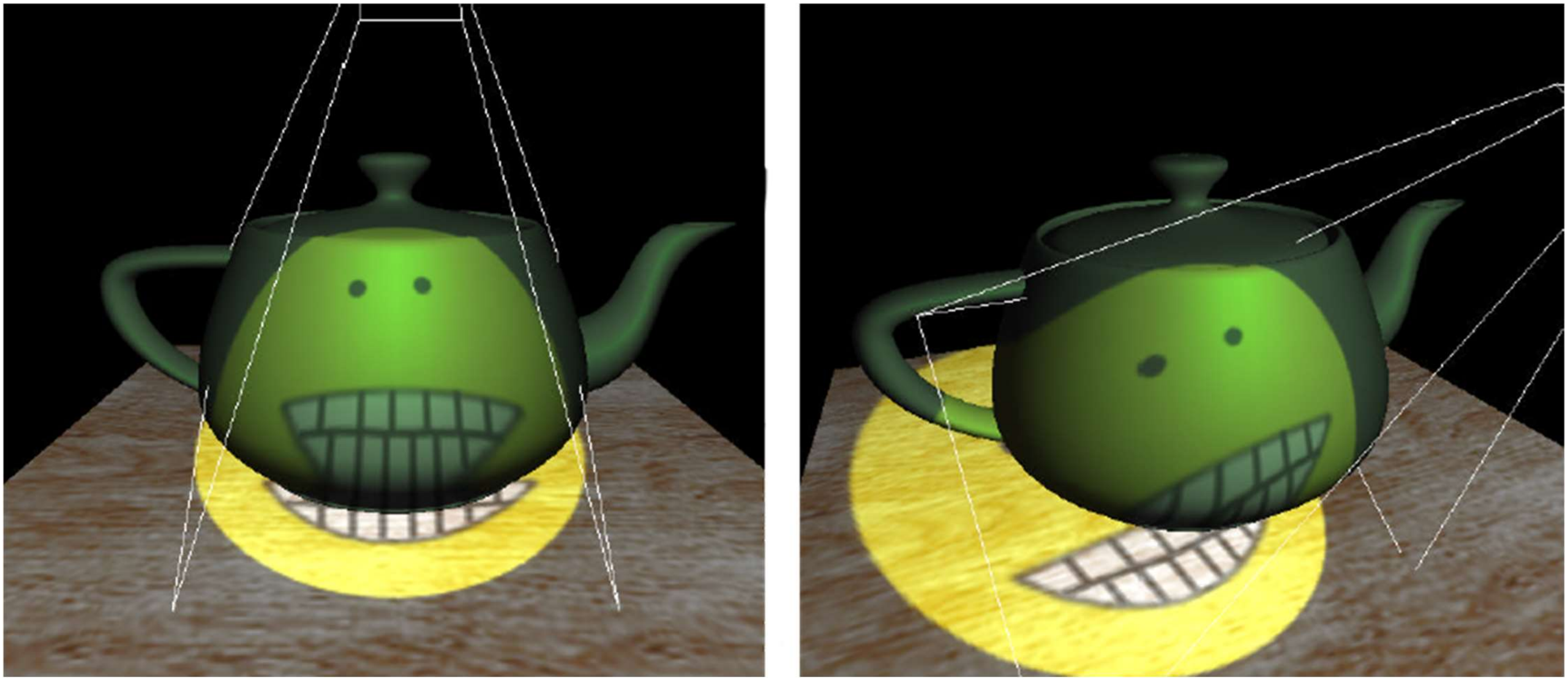
Heckbert and Moreton

# Projective Texture Mapping

- Want to simulate a beamer
  - … or a flashlight, or a slide projector
- Precursor to shadows
- Interesting mathematics: 2 perspective projections involved!
- Easy to program!



# Projective Texture Mapping



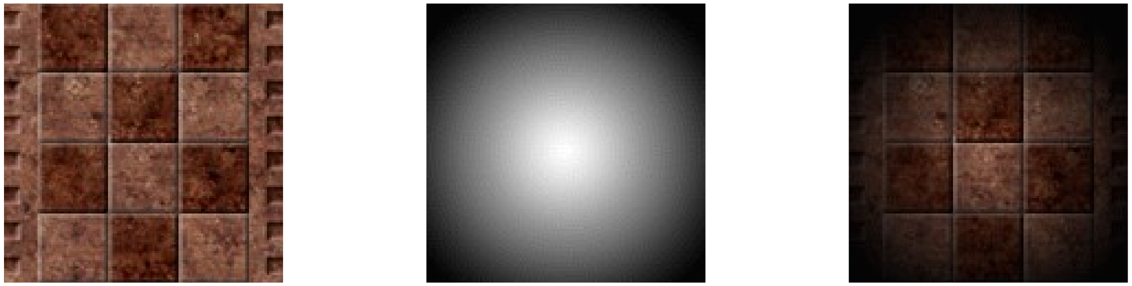
# Projective Shadows in Doom 3

# Projective Texturing

- What about homogeneous texture coords?
- Need to do perspective divide also for projector!
  - (s, t, q) → (s/q, t/q) for every fragment
- How does OpenGL do that?
  - Needs to be perspective correct as well!
  - Trick: interpolate (s/w, t/w, r/w, q/w)
  - (s/w) / (q/w) = s/q etc. at every fragment
- Remember: s,t,r,q are equivalent to x,y,z,w in projector space! → r/q = projector depth!

# Multitexturing

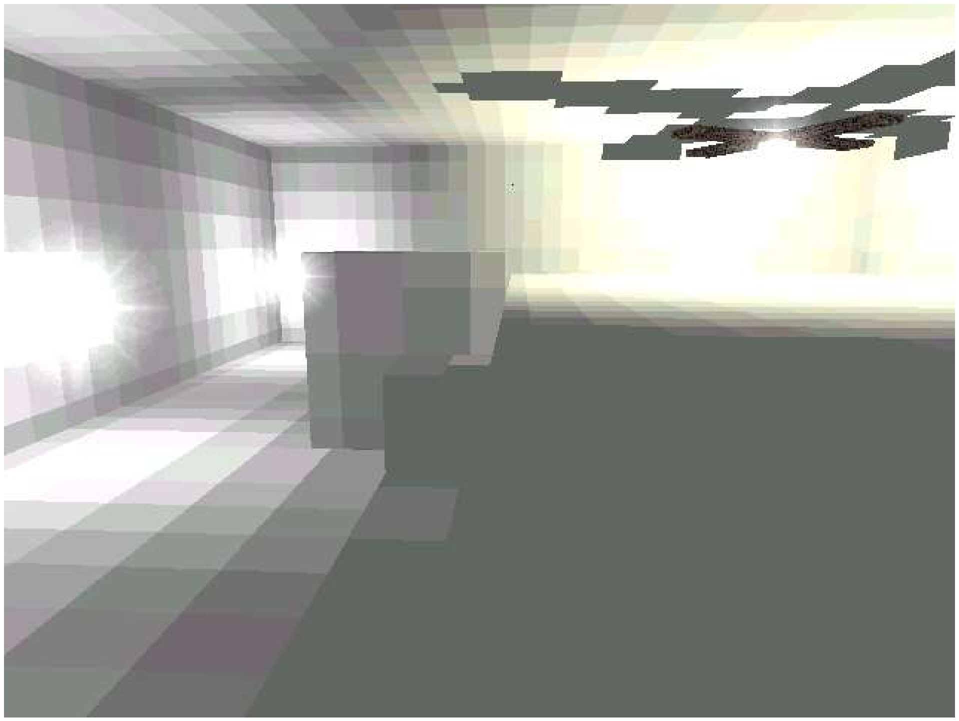
- Apply multiple textures in one pass
- *Integral* part of programmable shading
  - e.g. diffuse texture map + gloss map
  - e.g. diffuse texture map + light map
- Performance issues
  - How many textures are free?
  - How many are available



# Example: Light Mapping

- Used in virtually every commercial game
- Precalculate diffuse lighting on static objects
  - Only low resolution necessary
  - Diffuse lighting is view independent!
- Advantages:
  - No runtime lighting necessary
    - VERY fast!
  - Can take global effects (shadows, color bleeds) into account

# Light Mapping

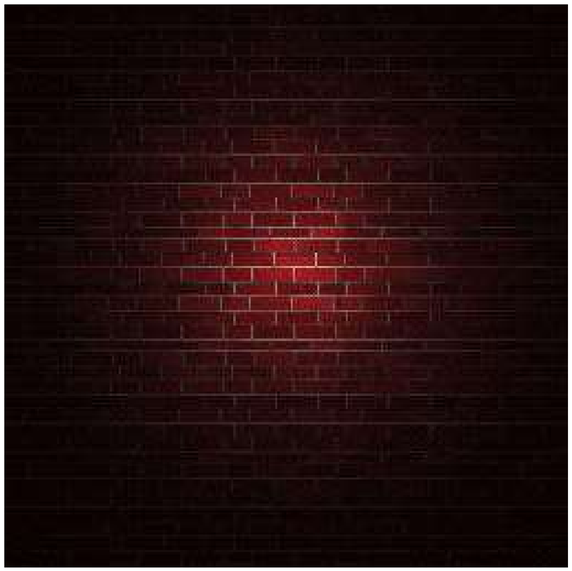


Original LM texels

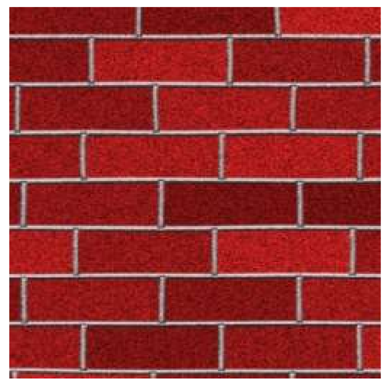
Bilinear Filtering

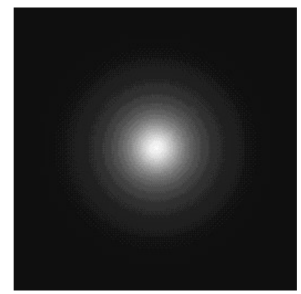
# Light Mapping Issues

- Why premultiplication is bad…



Full Size Texture
(with Lightmap)





Tiled Surface Texture
plus Lightmap

→ use tileable surface textures and low resolution lightmaps

# Light Mapping

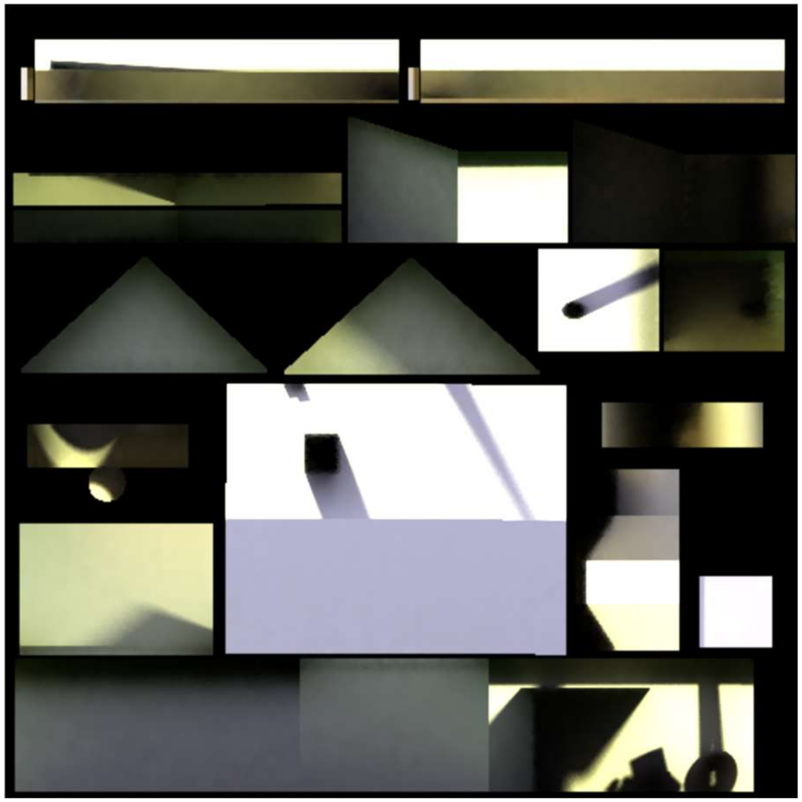
Original scene

Light-mapped

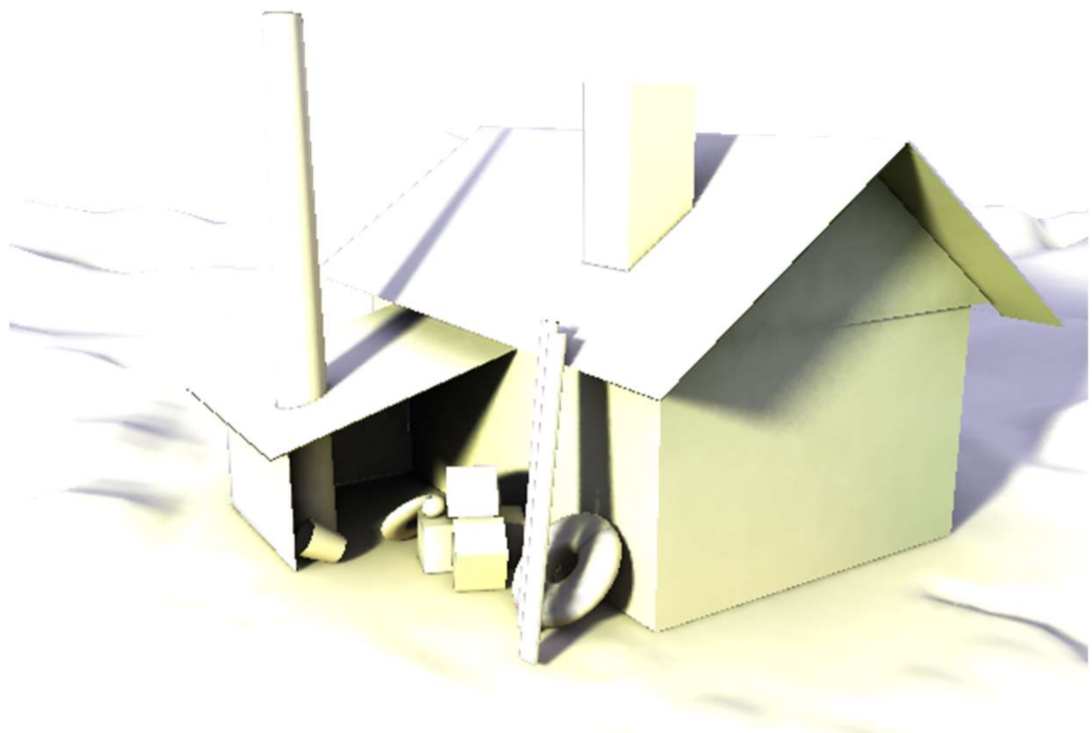
# Example: Light Mapping

- Precomputation based on non-realtime methods
  - Radiosity
  - Ray tracing
    - Monte Carlo Integration
    - Path tracing
    - Photon mapping

# Light Mapping

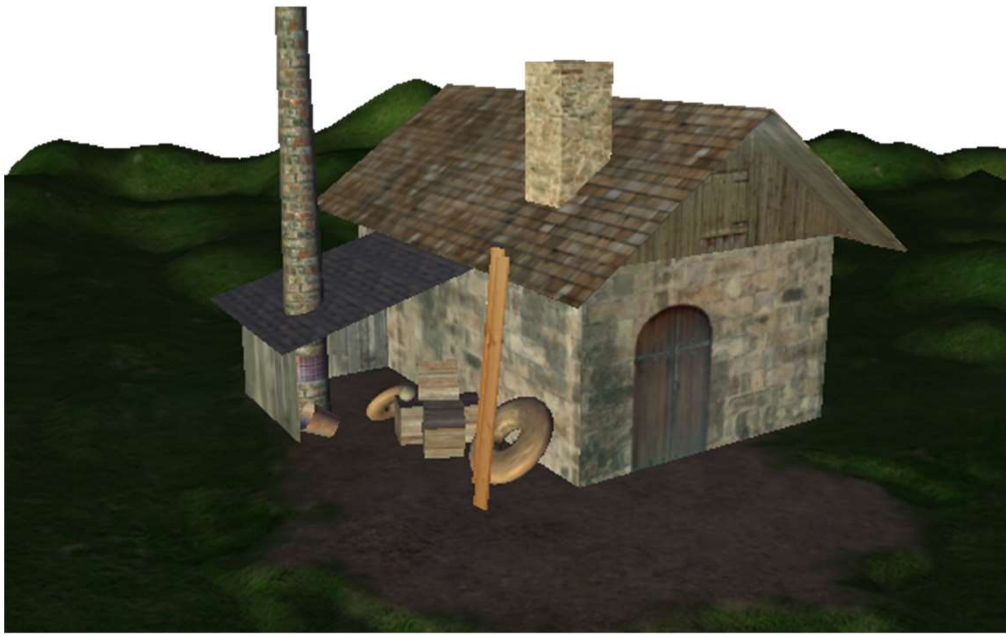


Lightmap



mapped

# Light Mapping



Original scene | Light-mapped

# Interpolation #2

Interpolation Type + Purpose #2:

**Interpolation of Samples in Texture Space**

*(Multi-Linear Interpolation)*

# Types of Textures

- Spatial layout
  - Cartesian grids: 1D, 2D, 3D, 2D_ARRAY, …
  - Cube maps, …

for Vulkan, see `vkImageView`

- Formats (too many), e.g. OpenGL
  - GL_LUMINANCE16_ALPHA16
  - GL_RGB8, GL_RGBA8, …: integer texture formats
  - GL_RGB16F, GL_RGBA32F, …: float texture formats
  - compressed formats, high dynamic range formats, …

- External (CPU) format vs. internal (GPU) format
  - OpenGL driver converts from external to internal

for Vulkan, see vkImage and `vkImageView`

use `VK_IMAGE_TILING_OPTIMAL` for `VkImageCreateInfo::tiling`

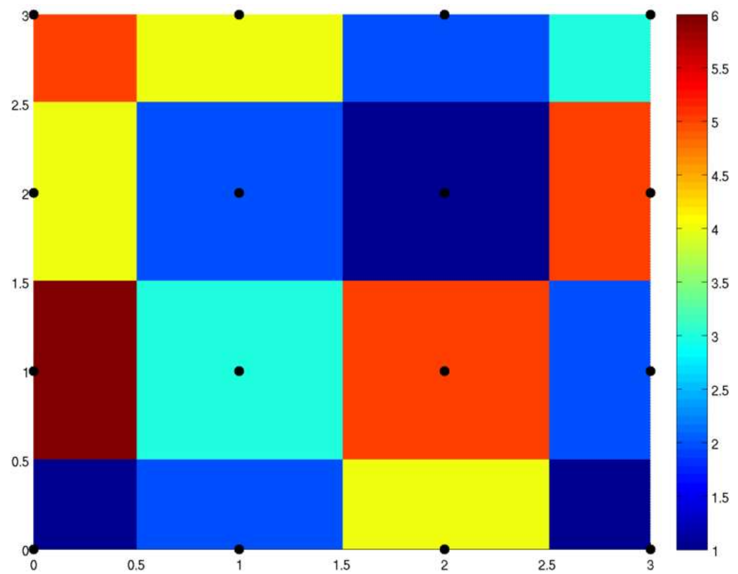
# Magnification (Bi-linear Filtering Example)

Original image

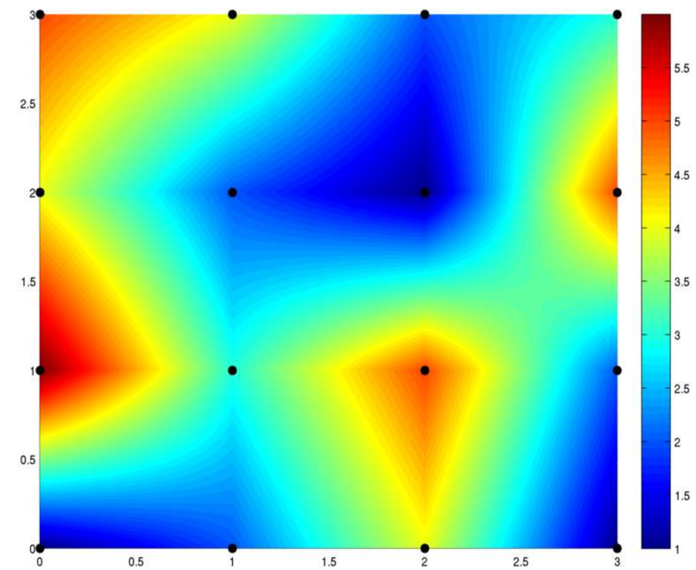
Nearest neighbor

Bi-linear filtering

# Nearest-Neighbor vs. Bi-Linear Interpolation

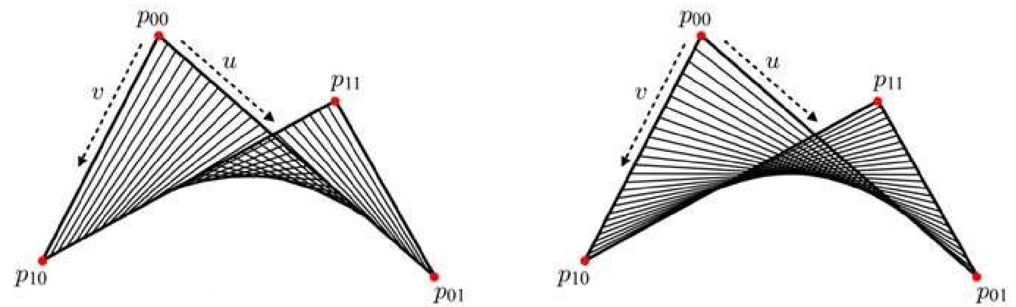


nearest-neighbor



bi-linear

wikipedia

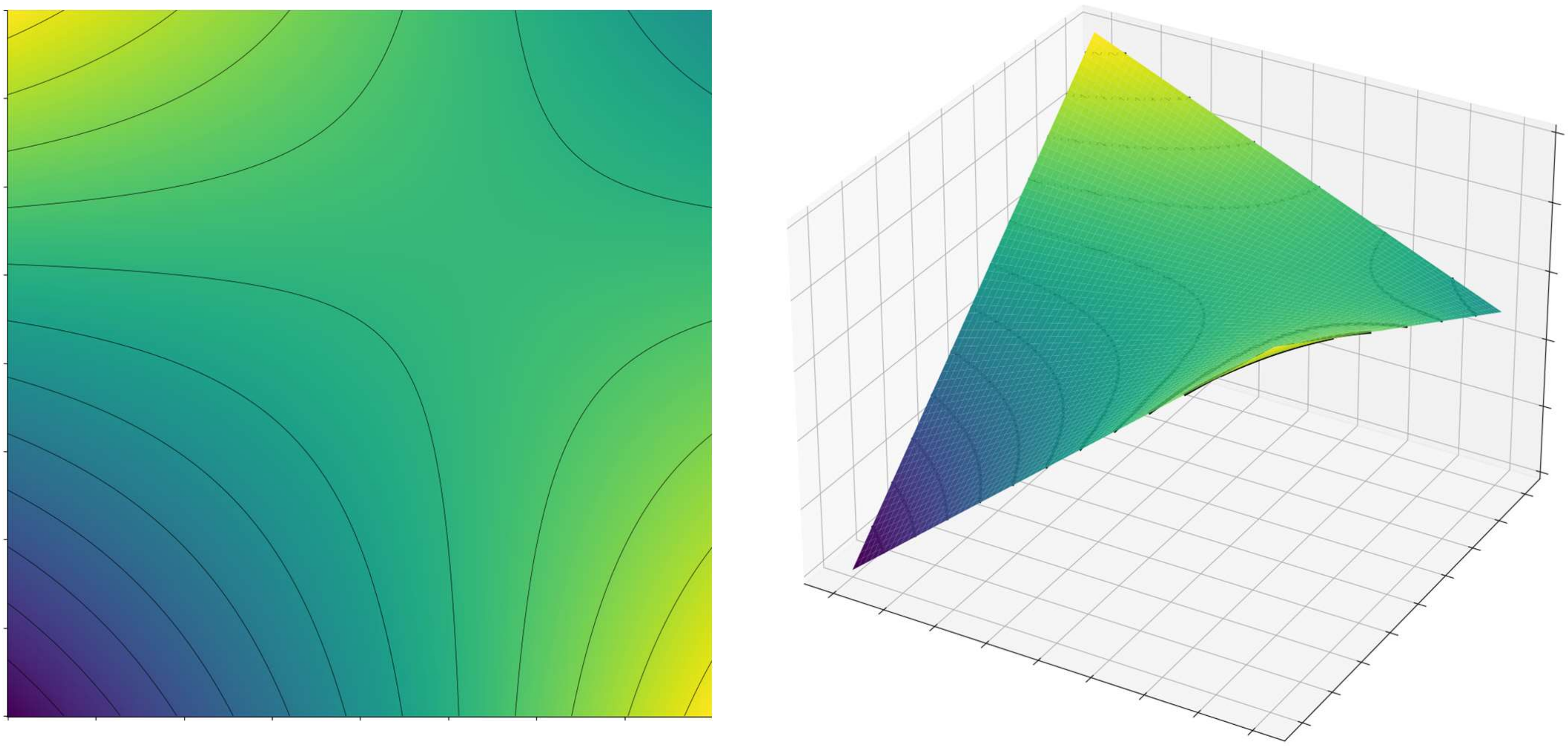



Bilinear patch (courtesy J. Han)

# Bi-Linear Interpolation

Consider area between 2x2 adjacent samples (e.g., pixel centers)

Example #2: 1 at top-left and bottom-right, 0 at bottom-left, 0.5 at top-right



# Bi-Linear Interpolation

Consider area between 2x2 adjacent samples (e.g., pixel centers):

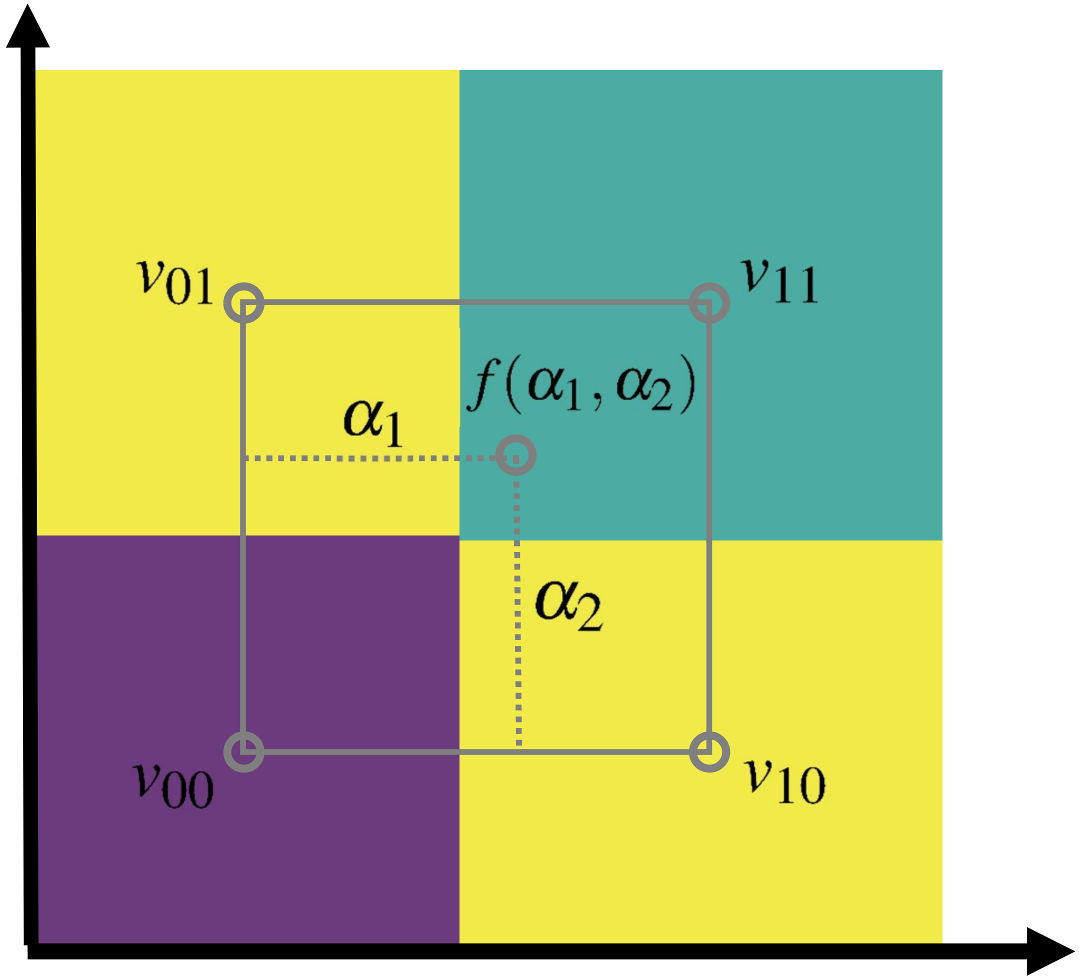
Given any (fractional) position

$$\alpha_1 := x_1 - \lfloor x_1 \rfloor \qquad \alpha_1 \in [0.0, 1.0)$$
$$\alpha_2 := x_2 - \lfloor x_2 \rfloor \qquad \alpha_2 \in [0.0, 1.0)$$

and 2x2 sample values

$$\begin{bmatrix} v_{01} & v_{11} \\ v_{00} & v_{10} \end{bmatrix}$$

Compute: $f(\alpha_1, \alpha_2)$



# Bi-Linear Interpolation

Consider area between 2x2 adjacent samples (e.g., pixel centers):

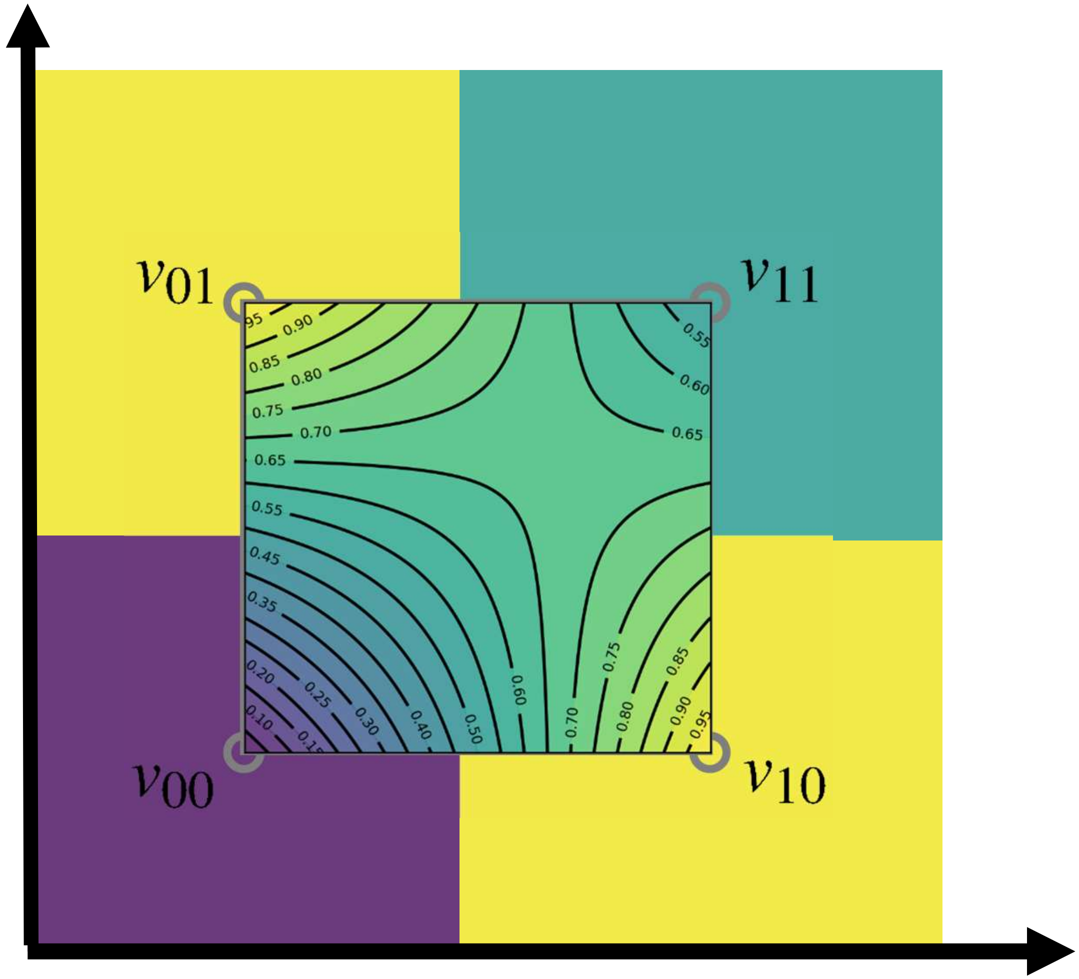
Given any (fractional) position

$$\alpha_1 := x_1 - \lfloor x_1 \rfloor \qquad \alpha_1 \in [0.0, 1.0)$$
$$\alpha_2 := x_2 - \lfloor x_2 \rfloor \qquad \alpha_2 \in [0.0, 1.0)$$

and 2x2 sample values

$$\begin{bmatrix} v_{01} & v_{11} \\ v_{00} & v_{10} \end{bmatrix}$$

Compute: $f(\alpha_1, \alpha_2)$



# Bi-Linear Interpolation

Weights in 2x2 format:

$$\begin{bmatrix} \alpha_2 \\ (1-\alpha_2) \end{bmatrix} \begin{bmatrix} (1-\alpha_1) & \alpha_1 \end{bmatrix} = \begin{bmatrix} (1-\alpha_1)\alpha_2 & \alpha_1\alpha_2 \\ (1-\alpha_1)(1-\alpha_2) & \alpha_1(1-\alpha_2) \end{bmatrix}$$

Interpolate function at (fractional) position $(\alpha_1, \alpha_2)$ :

$$f(\alpha_1, \alpha_2) = \begin{bmatrix} \alpha_2 & (1-\alpha_2) \end{bmatrix} \begin{bmatrix} v_{01} & v_{11} \\ v_{00} & v_{10} \end{bmatrix} \begin{bmatrix} (1-\alpha_1) \\ \alpha_1 \end{bmatrix}$$

# Bi-Linear Interpolation

Interpolate function at (fractional) position $(\alpha_1, \alpha_2)$ :

$$f(\alpha_1, \alpha_2) = \begin{bmatrix} \alpha_2 & (1-\alpha_2) \end{bmatrix} \begin{bmatrix} v_{01} & v_{11} \\ v_{00} & v_{10} \end{bmatrix} \begin{bmatrix} (1-\alpha_1) \\ \alpha_1 \end{bmatrix}$$

$$= \begin{bmatrix} \alpha_2 & (1-\alpha_2) \end{bmatrix} \begin{bmatrix} (1-\alpha_1)v_{01} + \alpha_1 v_{11} \\ (1-\alpha_1)v_{00} + \alpha_1 v_{10} \end{bmatrix}$$

$$= \begin{bmatrix} \alpha_2 v_{01} + (1-\alpha_2)v_{00} & \alpha_2 v_{11} + (1-\alpha_2)v_{10} \end{bmatrix} \begin{bmatrix} (1-\alpha_1) \\ \alpha_1 \end{bmatrix}$$

# Bi-Linear Interpolation

Interpolate function at (fractional) position $(\alpha_1, \alpha_2)$ :

$$f(\alpha_1, \alpha_2) = \begin{bmatrix} \alpha_2 & (1-\alpha_2) \end{bmatrix} \begin{bmatrix} v_{01} & v_{11} \\ v_{00} & v_{10} \end{bmatrix} \begin{bmatrix} (1-\alpha_1) \\ \alpha_1 \end{bmatrix}$$

$$= (1-\alpha_1)(1-\alpha_2)v_{00} + \alpha_1(1-\alpha_2)v_{10} + (1-\alpha_1)\alpha_2 v_{01} + \alpha_1\alpha_2 v_{11}$$

$$= v_{00} + \alpha_1(v_{10} - v_{00}) + \alpha_2(v_{01} - v_{00}) + \alpha_1\alpha_2(v_{00} + v_{11} - v_{10} - v_{01})$$

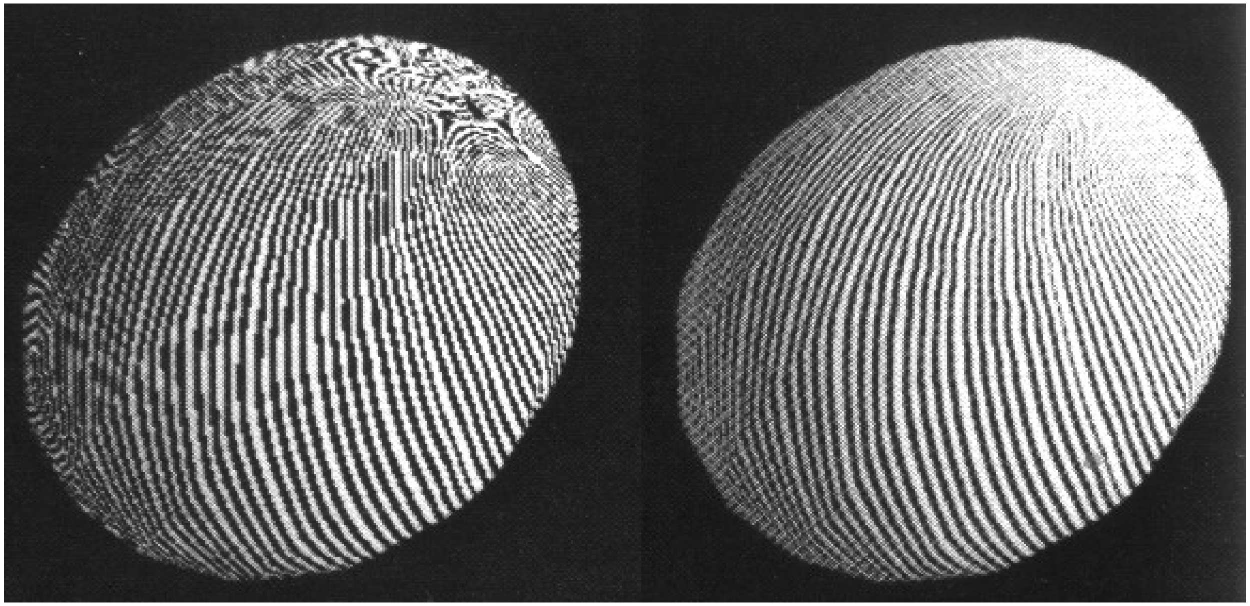
REALLY IMPORTANT:

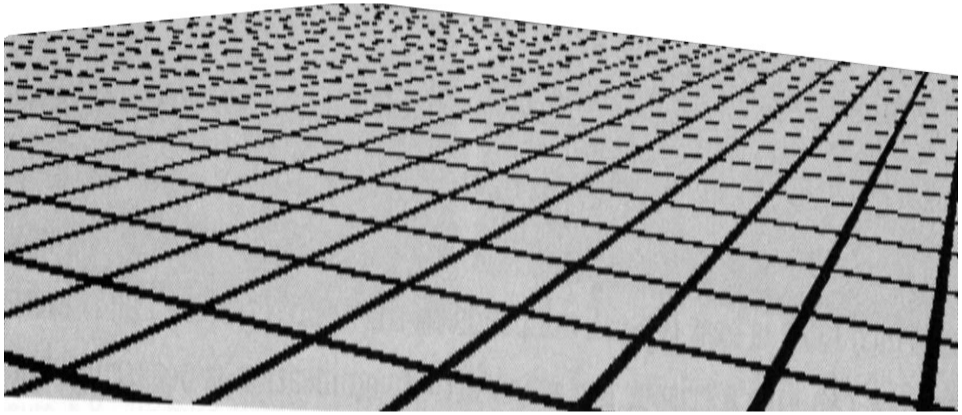
this is a different thing (for a different purpose) than the linear (or, in perspective, rational-linear) interpolation of texture coordinates!!

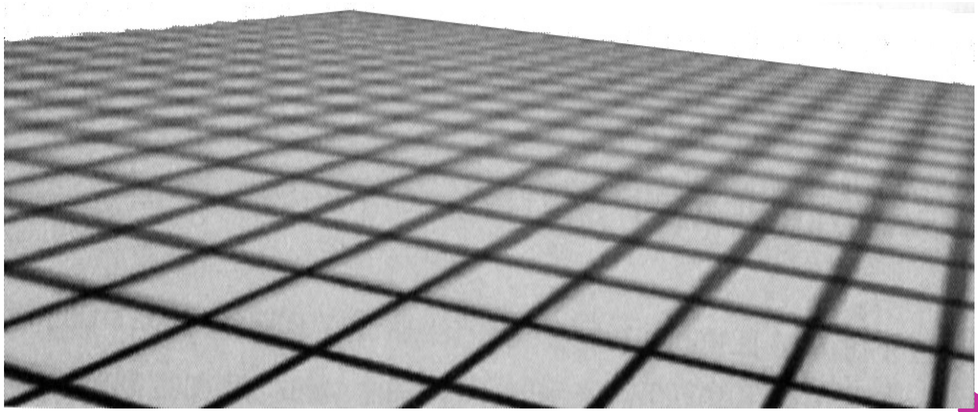
# Texture Minification

# Texture Aliasing: Minification

- Problem: One pixel in image space covers many texels

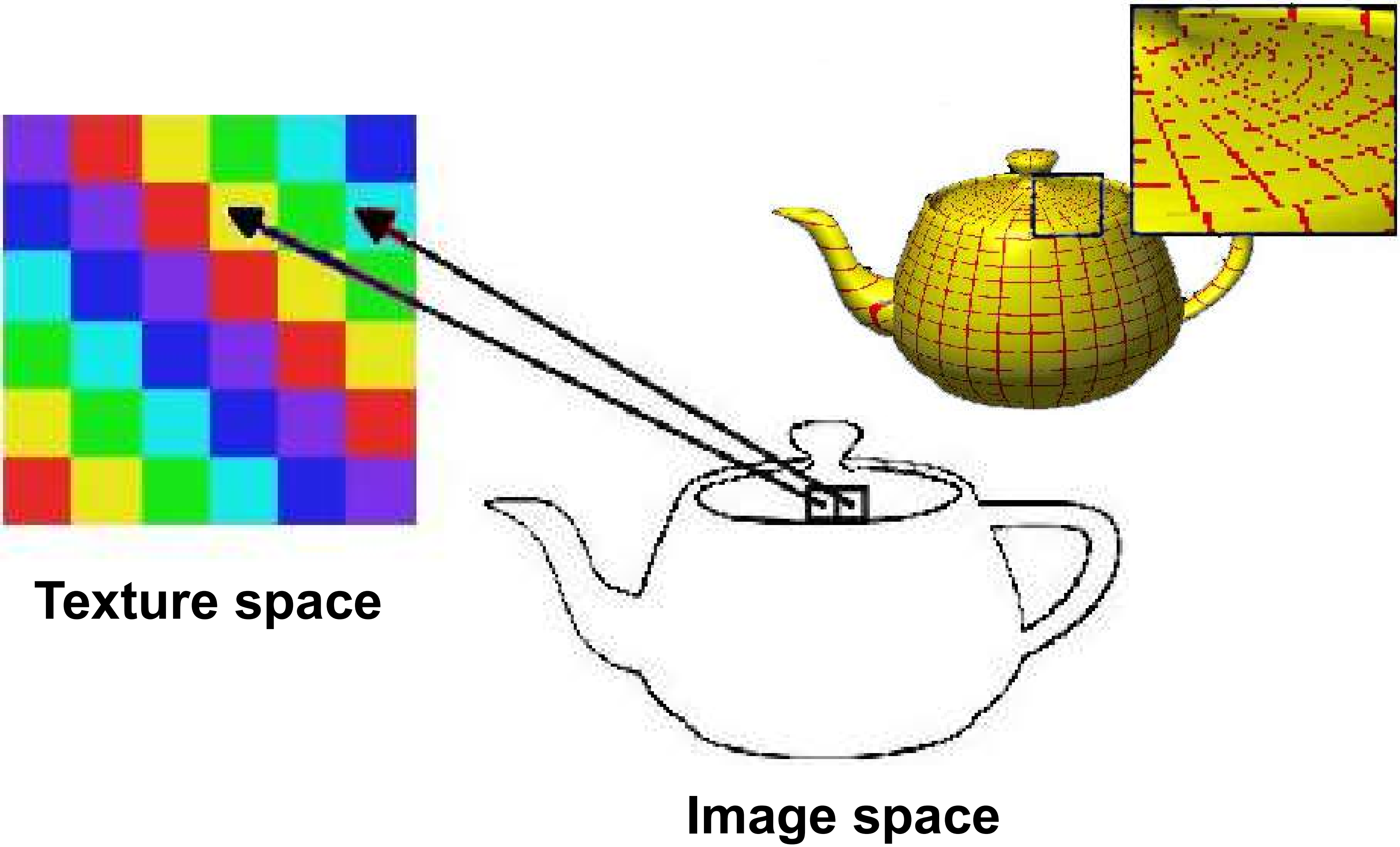






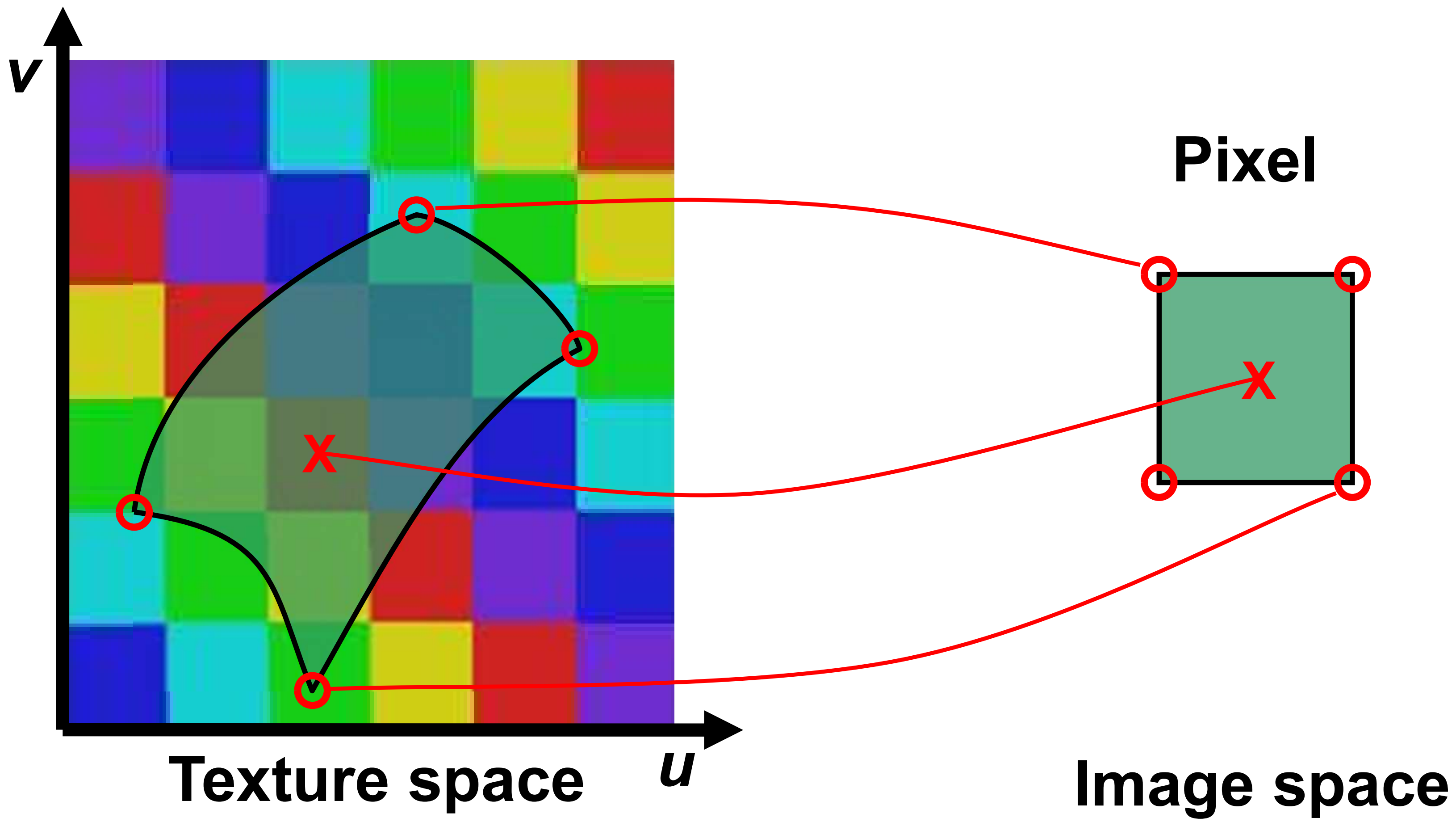
# Texture Aliasing: Minification

- Caused by *undersampling*: texture information is lost



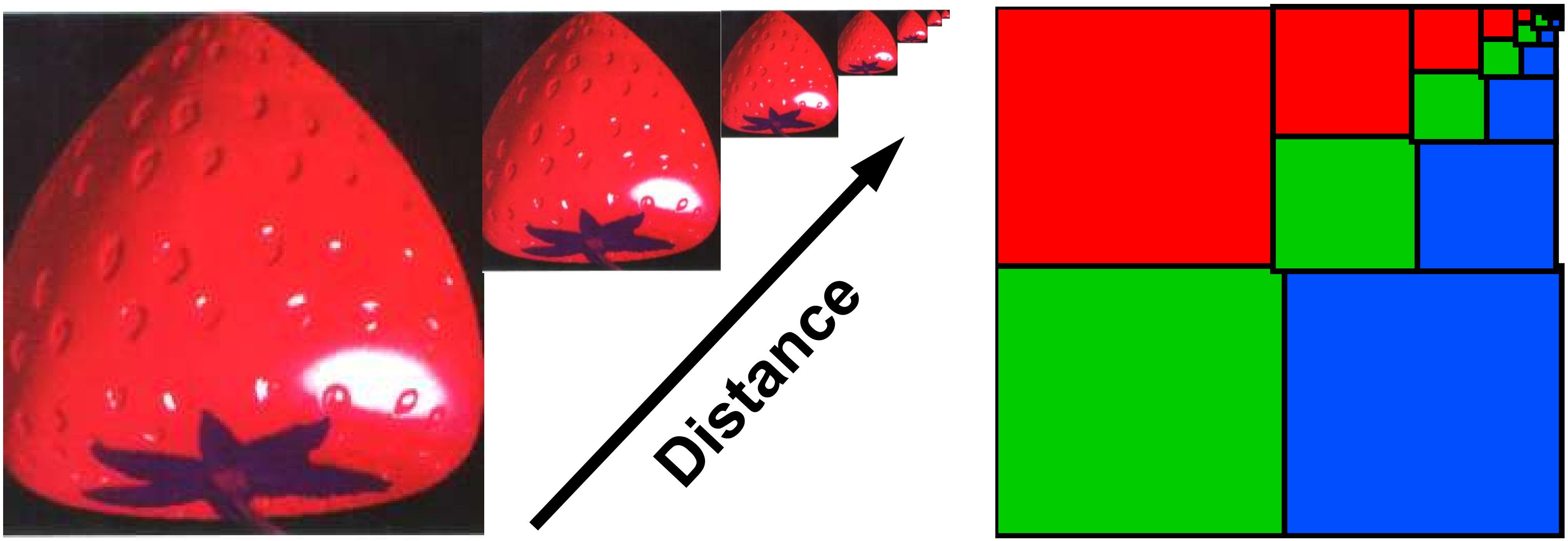
# Texture Anti-Aliasing: Minification

- A good pixel value is the weighted mean of the pixel area projected into texture space



# Texture Anti-Aliasing: MIP Mapping

- MIP Mapping ("Multum In Parvo")
  - Texture size is reduced by factors of 2 (*downsampling* = "many things in a small place")
  - Simple (4 pixel average) and memory efficient
  - Last image is only ONE texel

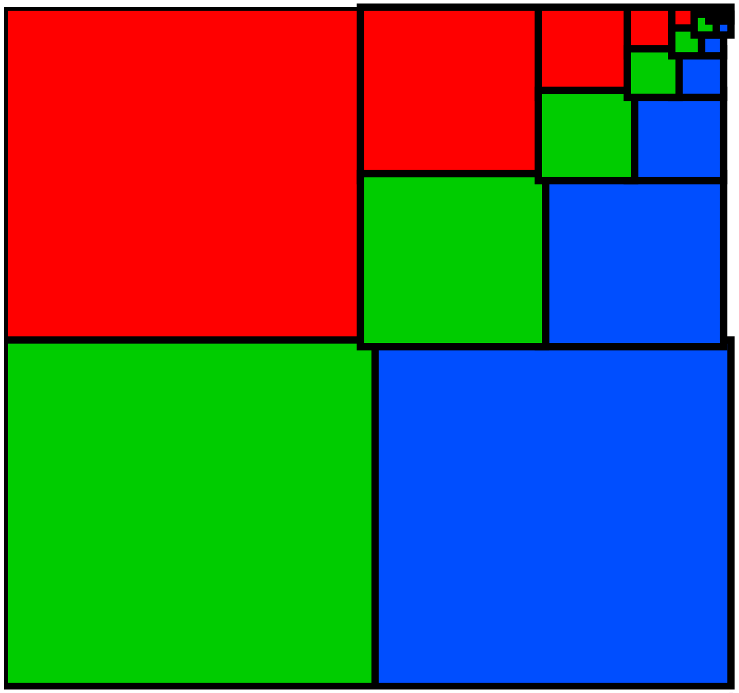


# Texture Anti-Aliasing: MIP Mapping

- MIP Mapping ("Multum In Parvo")
  - Texture size is reduced by factors of 2 (*downsampling* = "many things in a small place")
  - Simple (4 pixel average) and memory efficient
  - Last image is only ONE texel

geometric series:

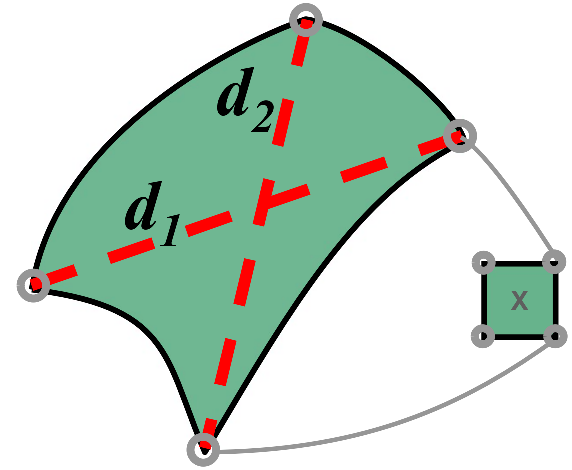
$$a + ar + ar^2 + ar^3 + \cdots + ar^{n-1} =$$
$$= \sum_{k=0}^{n-1} ar^k = a\left(\frac{1-r^n}{1-r}\right)$$

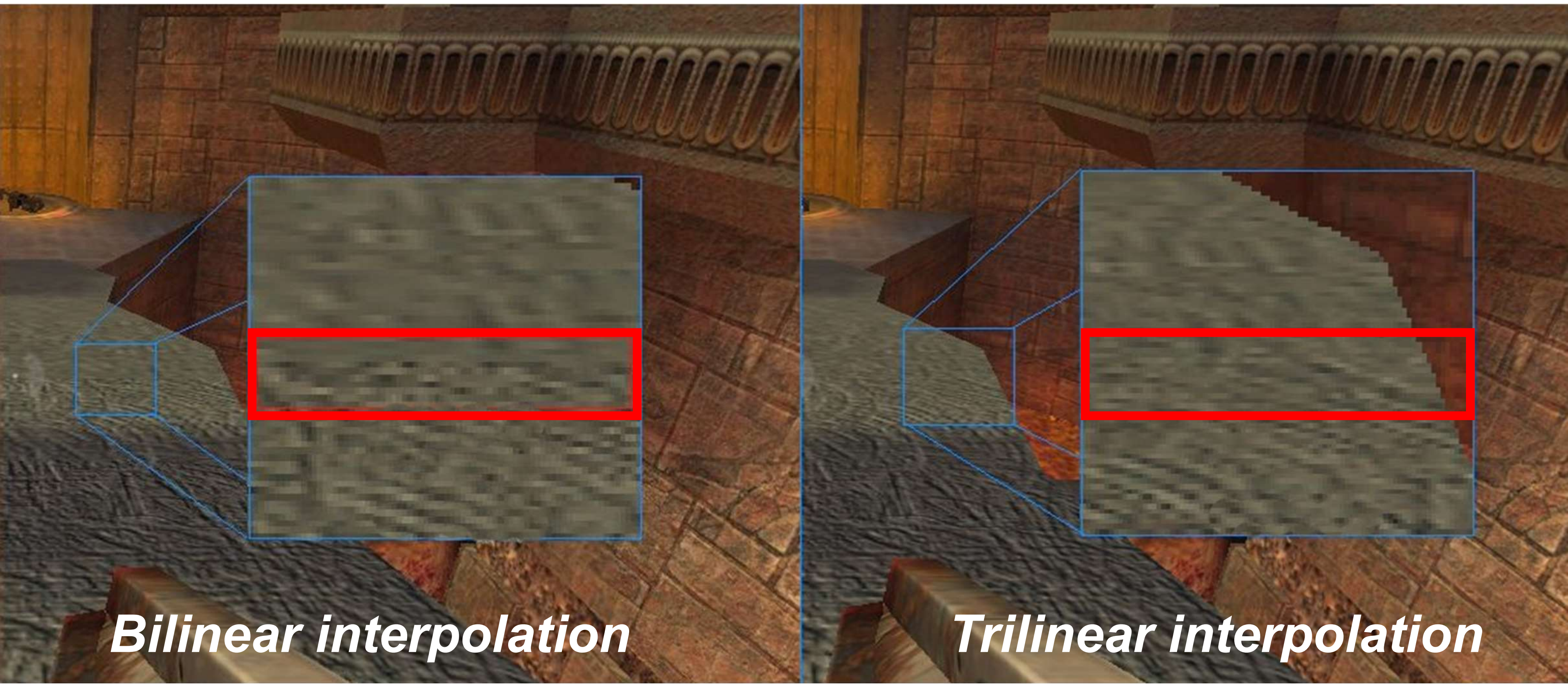


# Texture Anti-Aliasing: MIP Mapping

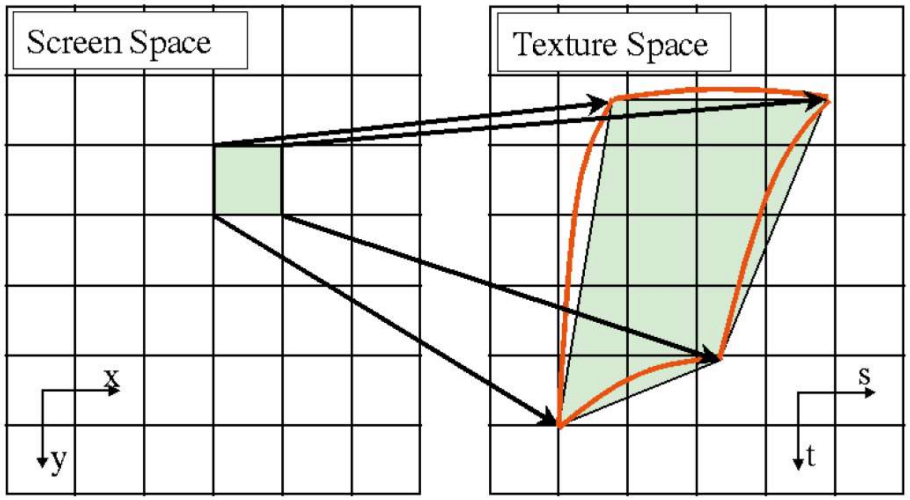
- MIP Mapping Algorithm
- $D := ld(max(d_1,d_2))$
- $T_0$ := value from texture $D_0 = trunc\ (D)$
  - Use *bilinear interpolation*

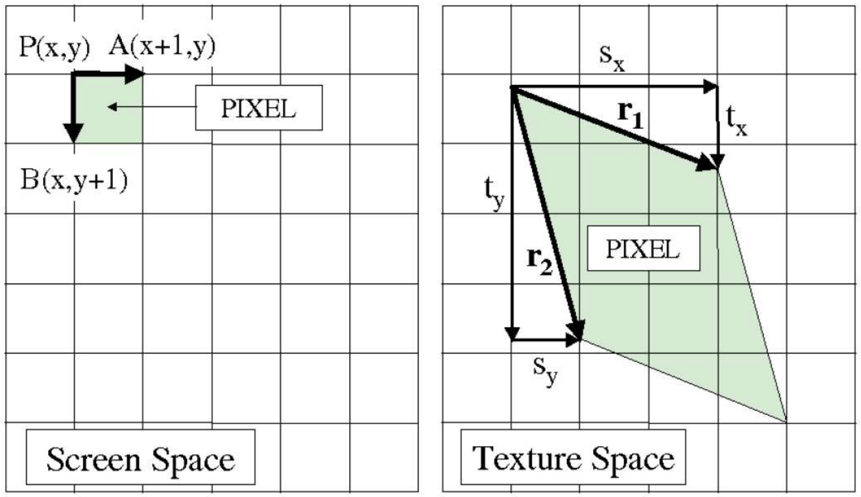
"Mip Map level"



# MIP-Map Level Computation




- Use the partial derivatives of texture coordinates with respect to screen space coordinates
- This is the Jacobian matrix

$$\begin{pmatrix} \partial u/\partial x & \partial u/\partial y \\ \partial v/\partial x & \partial v/\partial y \end{pmatrix} = \begin{pmatrix} s_x & s_y \\ t_x & t_y \end{pmatrix}$$

- Area of parallelogram is the absolute value of the Jacobian determinant (the *Jacobian*)

# MIP-Map Level Computation (OpenGL)

- OpenGL 4.6 core specification, pp. 251-264

(3D tex coords!)

$$\lambda_{base}(x, y) = \log_2[\rho(x, y)]$$

$$\rho = \max\left\{\sqrt{\left(\frac{\partial u}{\partial x}\right)^2 + \left(\frac{\partial v}{\partial x}\right)^2 + \left(\frac{\partial w}{\partial x}\right)^2}, \sqrt{\left(\frac{\partial u}{\partial y}\right)^2 + \left(\frac{\partial v}{\partial y}\right)^2 + \left(\frac{\partial w}{\partial y}\right)^2}\right\}$$

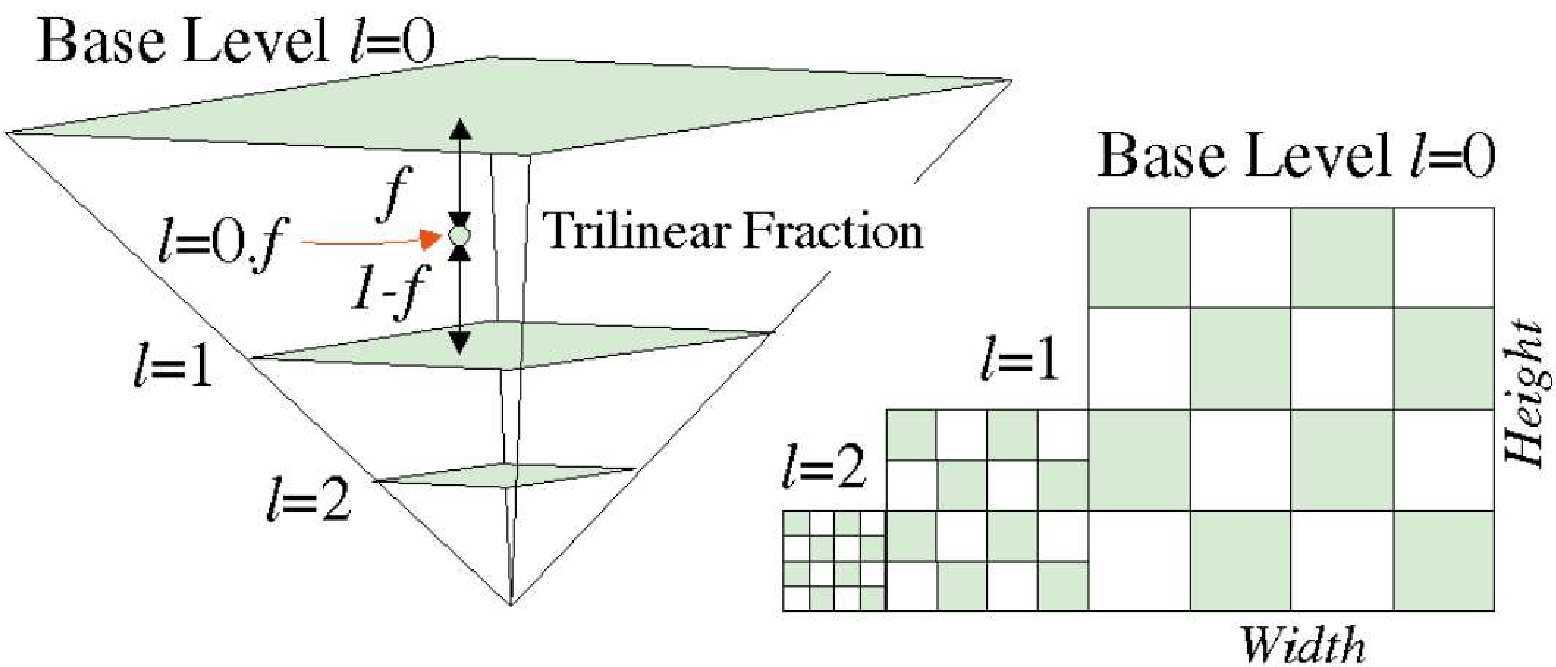
Does not use area of parallelogram but greater hypotenuse [Heckbert, 1983]

- Approximation without square-roots

$$m_u = \max\left\{\left|\frac{\partial u}{\partial x}\right|, \left|\frac{\partial u}{\partial y}\right|\right\} \quad m_v = \max\left\{\left|\frac{\partial v}{\partial x}\right|, \left|\frac{\partial v}{\partial y}\right|\right\} \quad m_w = \max\left\{\left|\frac{\partial w}{\partial x}\right|, \left|\frac{\partial w}{\partial y}\right|\right\}$$

$$\max\{m_u, m_v, m_w\} \leq f(x, y) \leq m_u + m_v + m_w$$

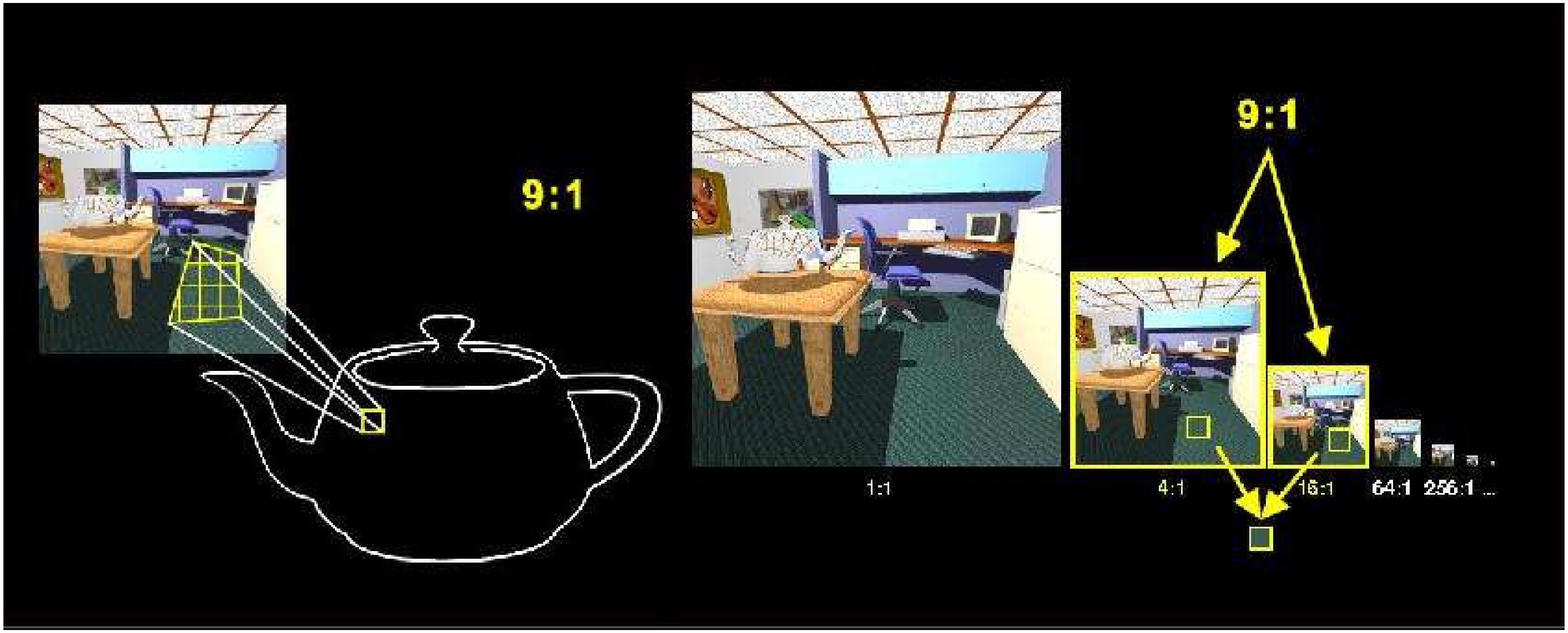
# MIP-Map Level Interpolation




- Level of detail value is fractional!
- Use fractional part to blend (lin.) between two adjacent mipmap levels

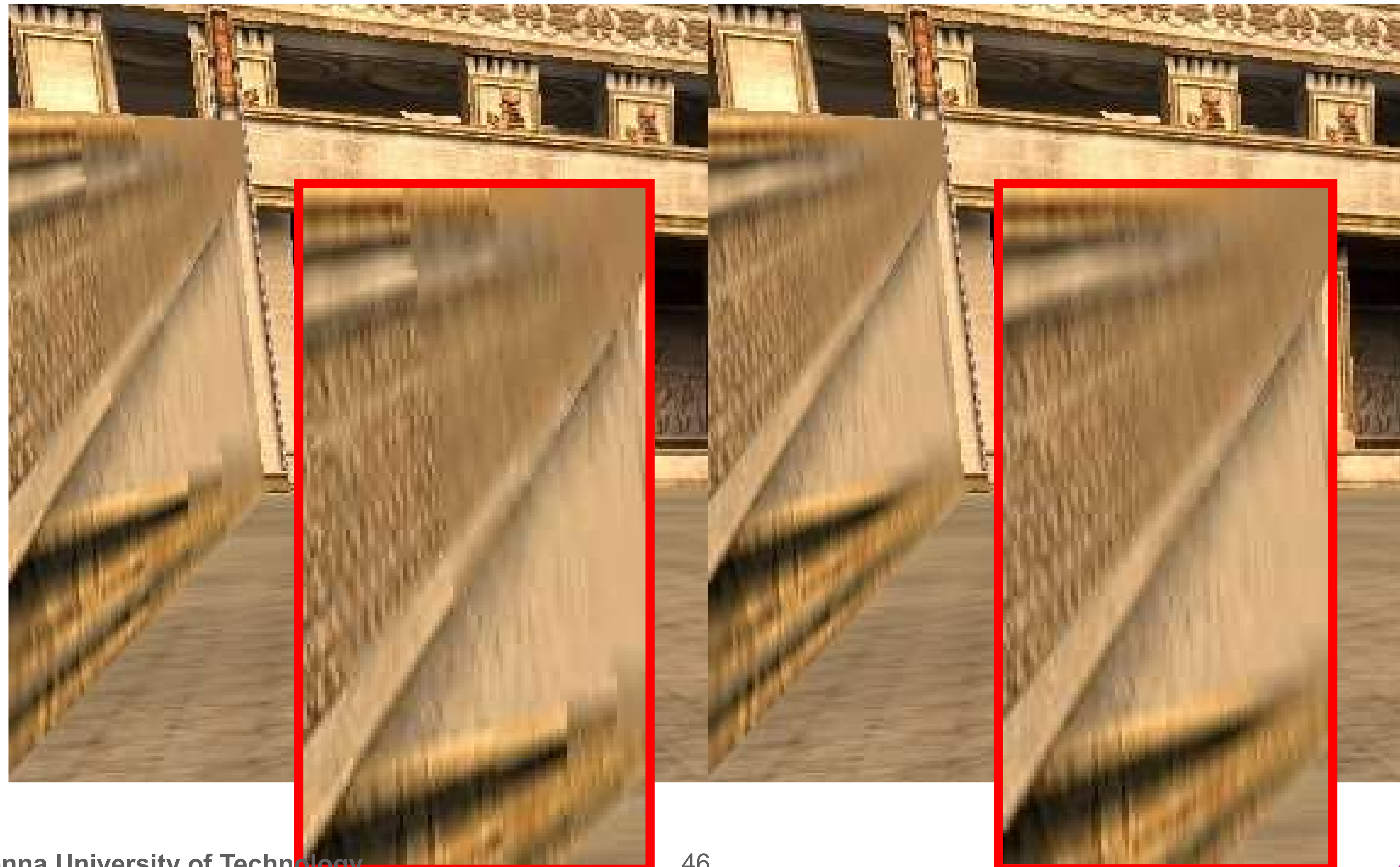
# Texture Anti-Aliasing: MIP Mapping

- Trilinear interpolation:
  - $T_1$ := value from texture $D_1 = D_0+1$ (bilin.interpolation)
  - Pixel value := $(D_1–D)\cdot T_0 + (D–D_0)\cdot T_1$
    - Linear interpolation between successive MIP Maps
  - Avoids "Mip banding" (but doubles texture lookups)



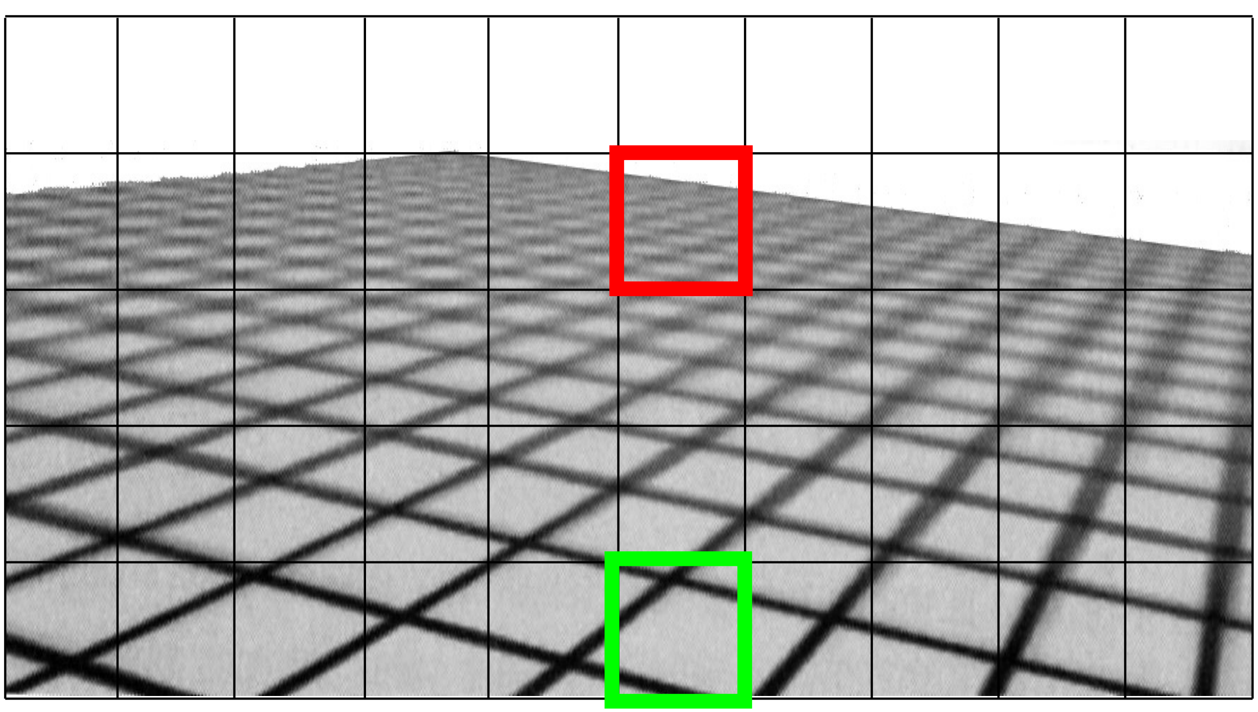
# Texture Anti-Aliasing: MIP Mapping

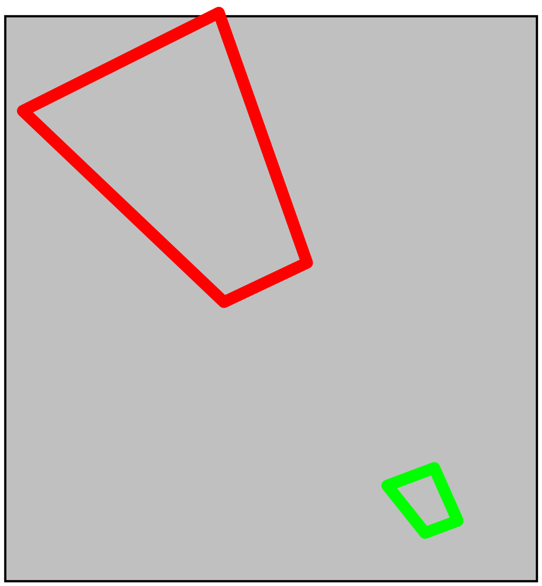
- Other example for bilinear vs. trilinear filtering



# Anti-Aliasing: Anisotropic Filtering

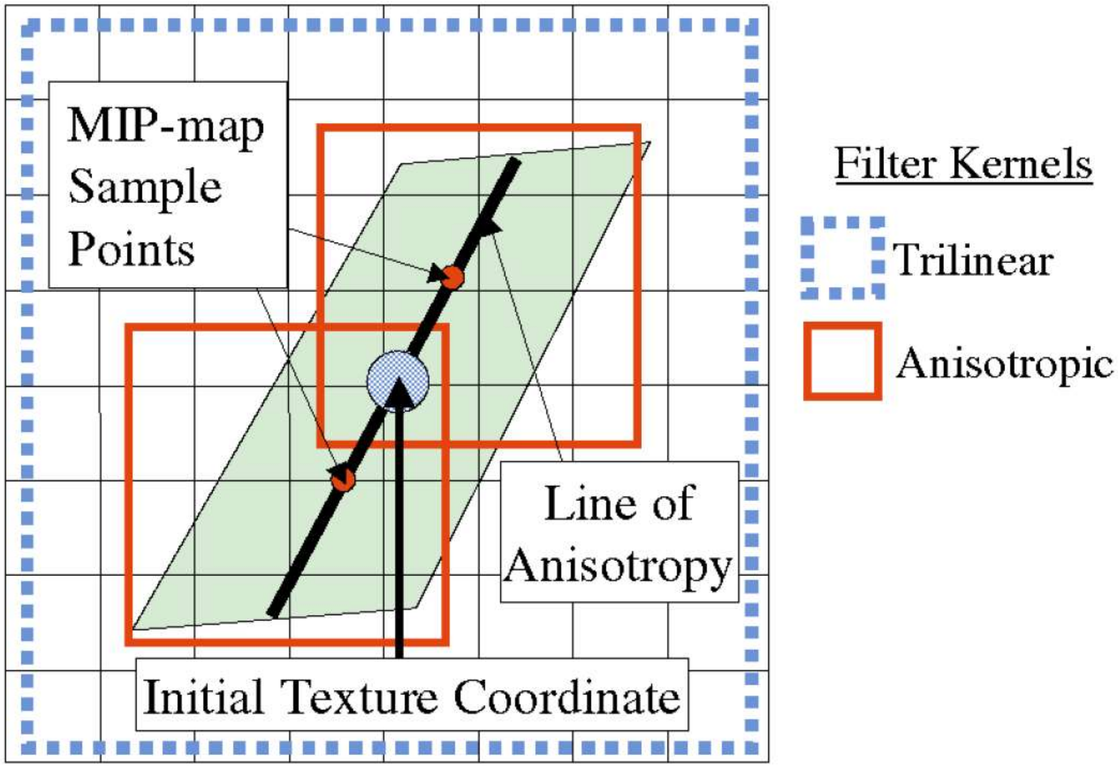
- Anisotropic filtering
  - View-dependent filter kernel
  - Implementation: *summed area table*, "*RIP Mapping*", *footprint assembly*, *elliptical weighted average* (EWA)





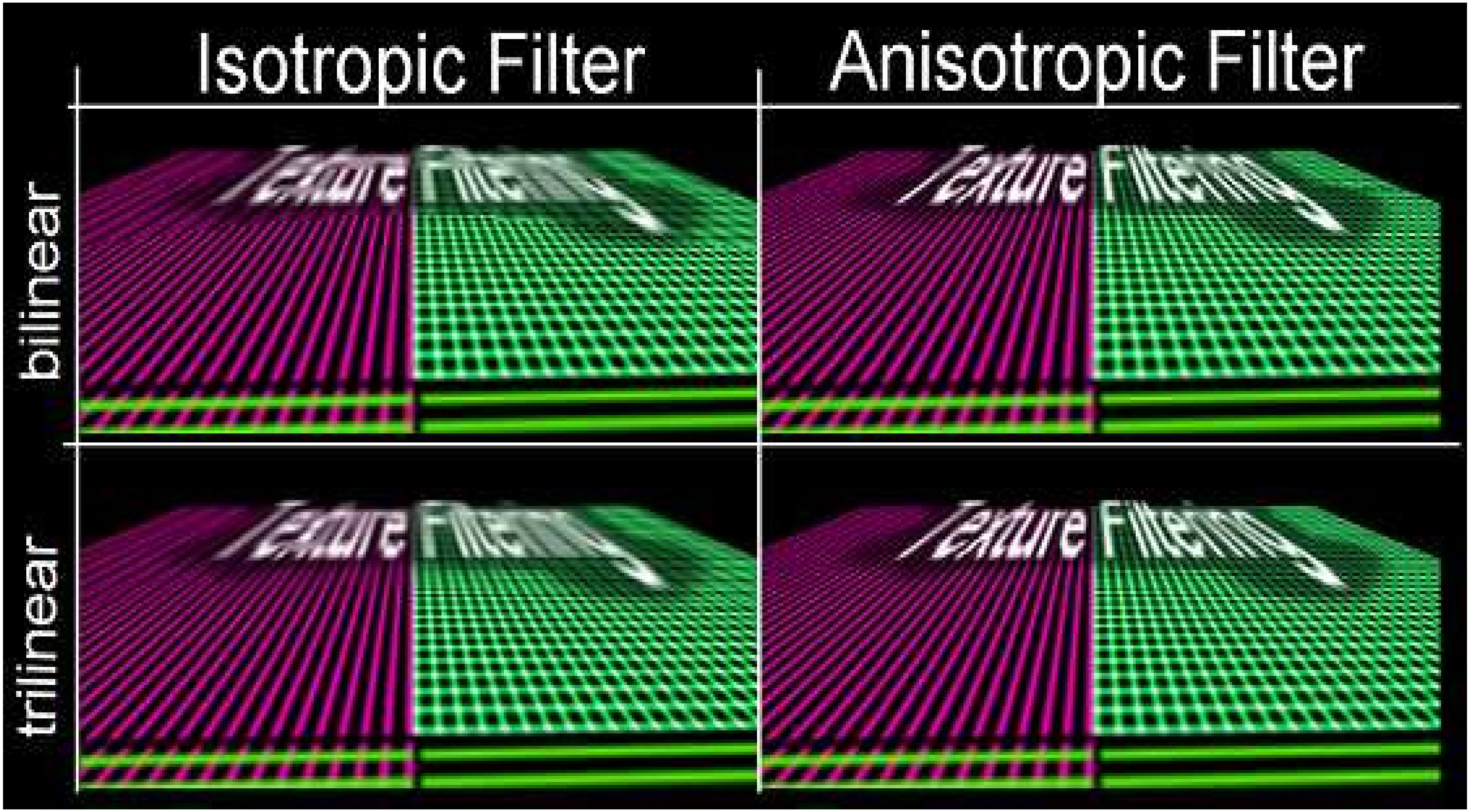
Texture space

# Anisotropic Filtering: Footprint Assembly



# Anti-Aliasing: Anisotropic Filtering

- Example



# Texture Anti-aliasing

- Basically, everything done in hardware
- `gluBuild2DMipmaps()` generates MIPmaps
- Set parameters in `glTexParameter()`
  - `GL_TEXTURE_MAG_FILTER: GL_NEAREST, GL_LINEAR, …`
  - `GL_TEXTURE_MIN_FILTER: GL_LINEAR_MIPMAP_NEAREST`
- Anisotropic filtering is an extension:
  - `GL_EXT_texture_filter_anisotropic`
  - Number of samples can be varied (4x,8x,16x)
    - Vendor specific support and extensions

for Vulkan, see `vkSampler`, `VkSamplerCreateInfo::magFilter`, `VkSamplerCreateInfo::minFilter`, `VK_FILTER_NEAREST`, `VK_FILTER_LINEAR`, `VkSamplerCreateInfo::mipmapMode`, `VK_SAMPLER_MIPMAP_MODE_NEAREST`, `VK_SAMPLER_MIPMAP_MODE_LINEAR`, …

Thank you.