King Abdullah University of Science and Technology KAUST

CS 380 - GPU and GPGPU Programming
Lecture 23: GPU Texturing, Pt. 5

Markus Hadwiger, KAUST

Reading Assignment #12 (until Nov 24)

Read (required):

- Look at Vulkan *sparse resources*, especially *sparse partially-resident images*
  - `https://docs.vulkan.org/spec/latest/chapters/sparsemem.html`
- Read about shadow mapping
  - `https://en.wikipedia.org/wiki/Shadow_mapping`
- Look at Unreal Engine 5 virtual texturing
  - `https://dev.epicgames.com/documentation/en-us/unreal-engine/virtual-texturing-in-unreal-engine/`
- Look at Unreal Engine 5 MegaLights
  - `https://dev.epicgames.com/documentation/en-us/unreal-engine/megalights-in-unreal-engine/`

Read (optional):

- CUDA Warp-Level Primitives
  - `https://developer.nvidia.com/blog/using-cuda-warp-level-primitives/`
- Warp-aggregated atomics
  - `https://developer.nvidia.com/blog/cuda-pro-tip-optimized-filtering-warp-aggregated-atomics/`

GPU Texturing

Interpolation #1

Interpolation Type + Purpose #1:

Interpolation of Texture Coordinates

(Linear / Rational-Linear Interpolation)

Homogeneous Coordinates (1)

Projective geometry

- (Real) projective spaces RP^n:

  Real projective line RP^1, real projective plane RP^2, ...

- A point in RP^n is a line through the origin (i.e., all the scalar multiples of the same vector) in an (n+1)-dimensional (real) vector space

Homogeneous coordinates of 2D projective point in RP^2

- Coordinates differing only by a non-zero factor λ map to the same point

  $(\lambda x, \lambda y, \lambda)$ dividing out the λ gives $(x, y, 1)$, corresponding to (x,y) in R^2

- Coordinates with last component = 0 map to “points at infinity”

  $(\lambda x, \lambda y, 0)$ division by last component not allowed; but again this is the same point if it only differs by a scalar factor, e.g., this is the same point as $(x, y, 0)$

Texture Mapping

2D (3D) Texture Space

Texture Transformation

2D Object Parameters

Parameterization

3D Object Space

Model Transformation

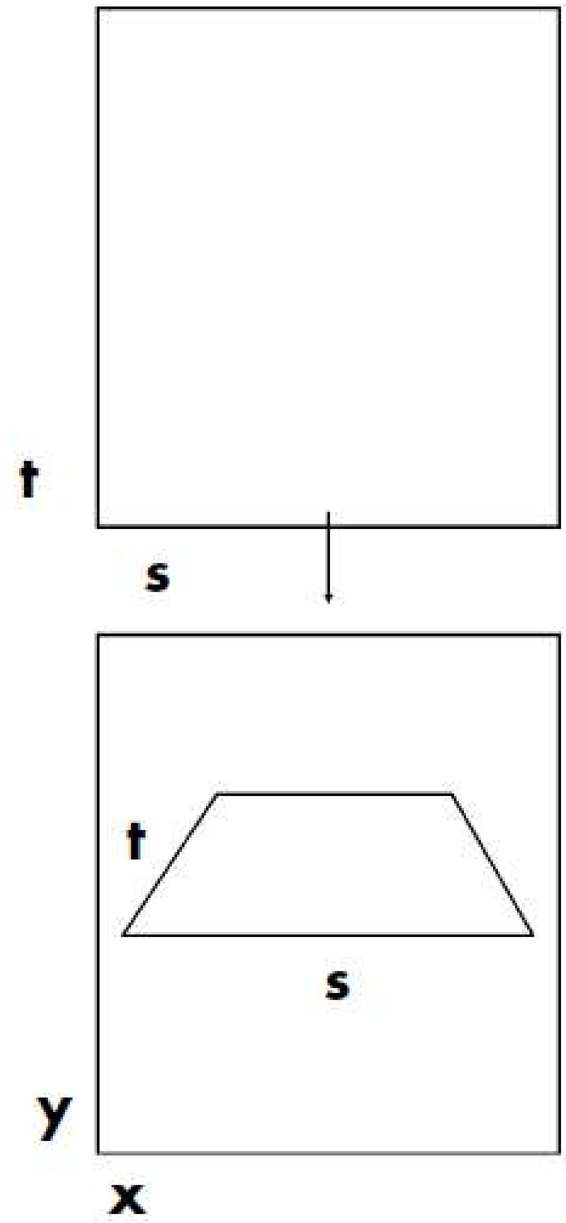
3D World Space

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

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

Kurt Akeley, Pat Hanrahan

Linear interpolation

Compute intermediate attribute value

- Along a line: $A = aA_1 + bA_2$, $a+b=1$
- On a plane: $A = aA_1 + bA_2 + cA_3$, $a+b+c=1$

Only projected values interpolate linearly in screen space (straight lines project to straight lines)

- x and y are projected (divided by w)
- Attribute values are not naturally projected

Choice for attribute interpolation in screen space

- Interpolate unprojected values
  - Cheap and easy to do, but gives wrong values
  - Sometimes OK for color, but
  - Never acceptable for texture coordinates
- Do it right

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

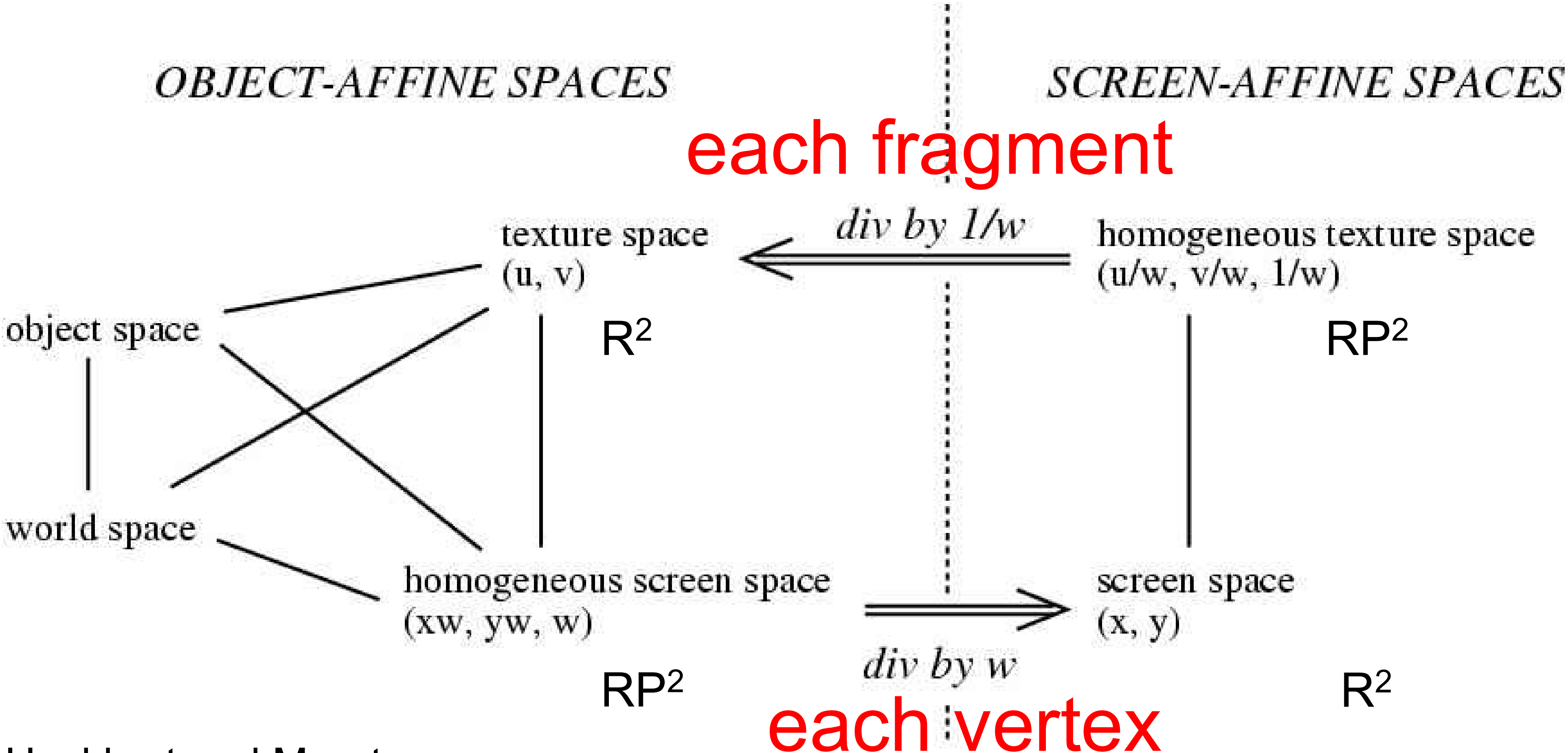
Barycentric triangle parameterization:

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

Kurt Akeley, Pat Hanrahan

Perspective Texture Mapping

- Solution: interpolate (s/w, t/w, 1/w)
- (s/w) / (1/w) = s etc. at every fragment



Heckbert and Moreton

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 4×4 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 $i \leq n$, $s_{n+1} = 1/w$.

(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 Map vs. Interpolation Recipe (1)

In general (see previous slides),
we had the projective map:

Backward transformation: linear projective map

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

Let's rename and rewrite this as:

$$\begin{bmatrix} s \\ t \\ q \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x_{world} \\ y_{world} \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \cdot w \\ y \cdot w \\ w \end{bmatrix}$$

For homogeneous points
we can also divide by w:

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

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

Coordinates on the right become
screen space coordinates!

Projective Map vs. Interpolation Recipe (2)

In general (see previous slides),
we had the projective map:

Backward transformation: linear projective map

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

Let's rename and rewrite this as:

$$\begin{bmatrix} s \\ t \\ q \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x_{world} \\ y_{world} \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \cdot w \\ y \cdot w \\ w \end{bmatrix}$$

For homogeneous points
we can also divide by w:

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

(special case $q = 1$)

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

Coordinates on the right become
screen space coordinates!

Projective Map vs. Interpolation Recipe (3)

In general (see previous slides),
we had the projective map:

Backward transformation: linear projective map

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

Now consider scanline interpolation:

(barycentric interpolation is linear along any line: here, horizontal line)

$$\begin{bmatrix} s/w \\ t/w \\ 1/w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x + \Delta x \\ y \\ 1 \end{bmatrix},$$

$$\begin{bmatrix} s/w \\ t/w \\ 1/w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} x \\ y \\ 1 \end{bmatrix} + \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} \Delta x \\ 0 \\ 0 \end{bmatrix}$$

$$\Delta_x \begin{bmatrix} s/w \\ t/w \\ 1/w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} \Delta x \\ 0 \\ 0 \end{bmatrix} = \begin{bmatrix} a \cdot \Delta x \\ d \cdot \Delta x \\ g \cdot \Delta x \end{bmatrix} = \begin{bmatrix} a \\ d \\ g \end{bmatrix}$$

$$(\Delta x = 1)$$

Interpolation #2

Interpolation Type + Purpose #2:

Interpolation of Samples in Texture Space

(Multi-Linear Interpolation)

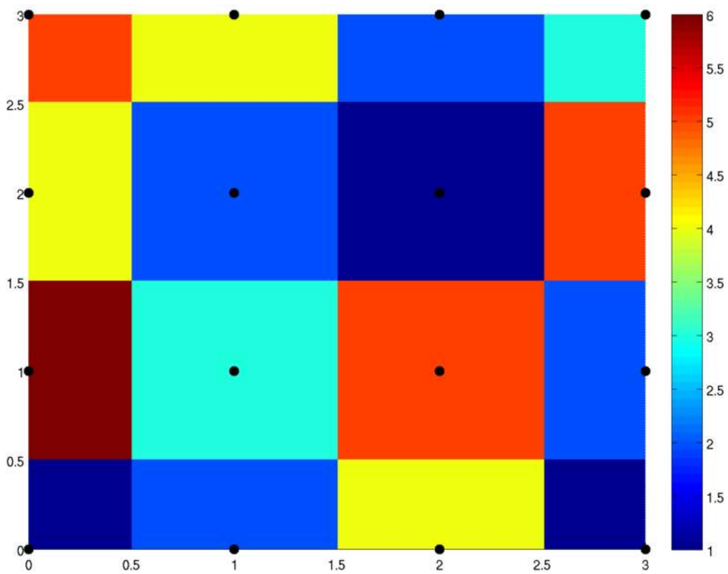
Magnification (Bi-linear Filtering Example)

Original image

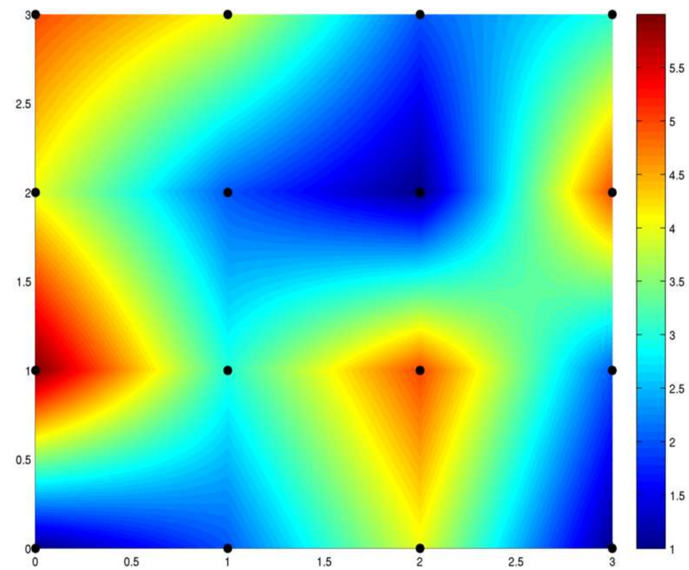
Nearest neighbor

Bi-linear filtering

Nearest-Neighbor vs. Bi-Linear Interpolation

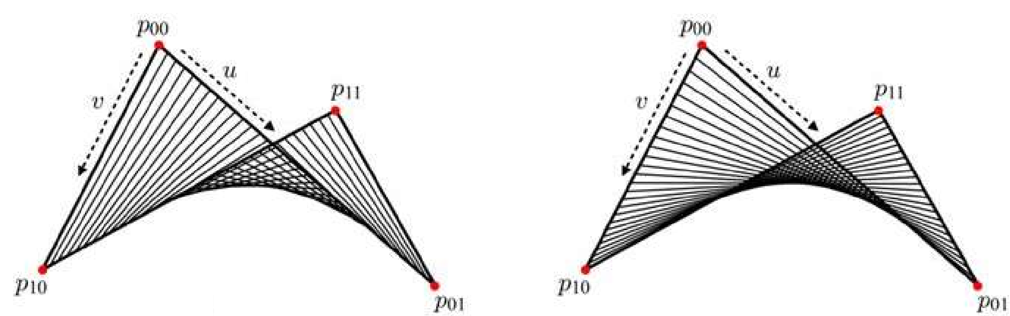


nearest-neighbor



bi-linear

wikipedia

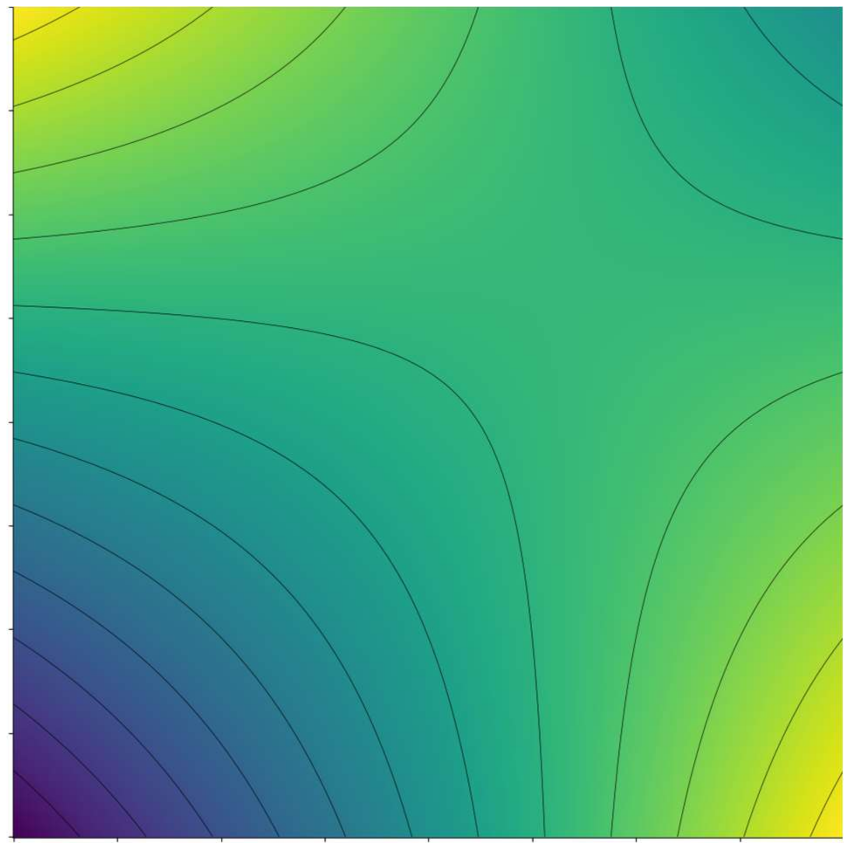


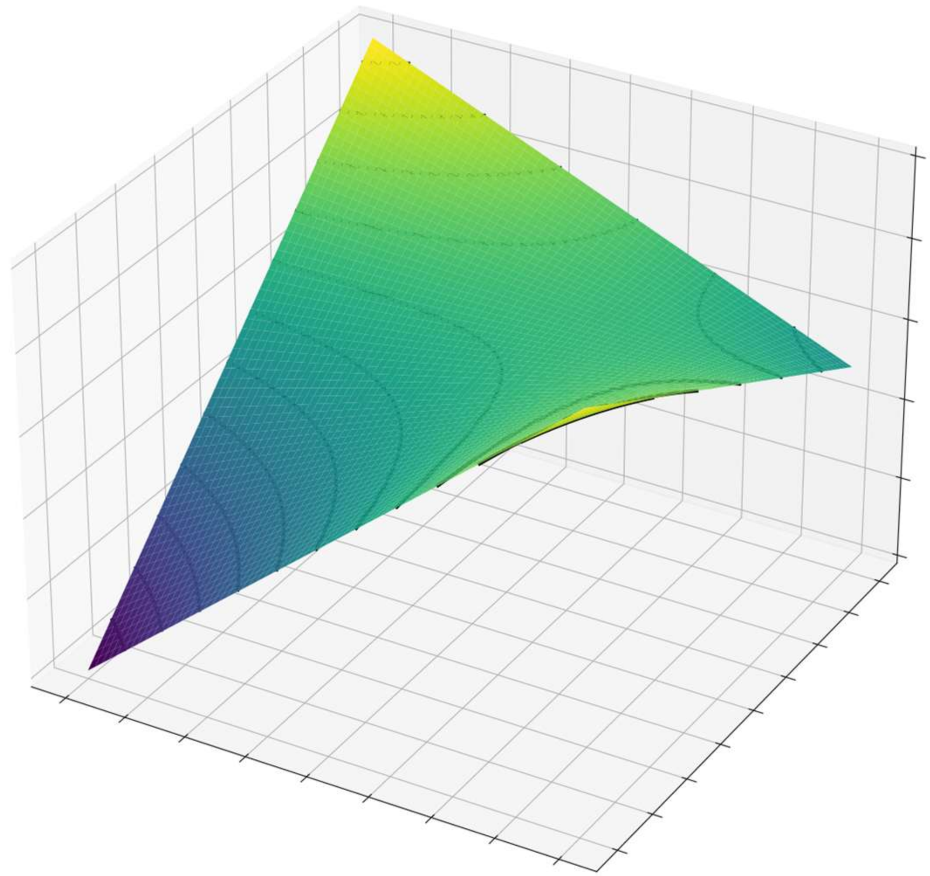
Bilinear patch (courtesy J. Han)

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):

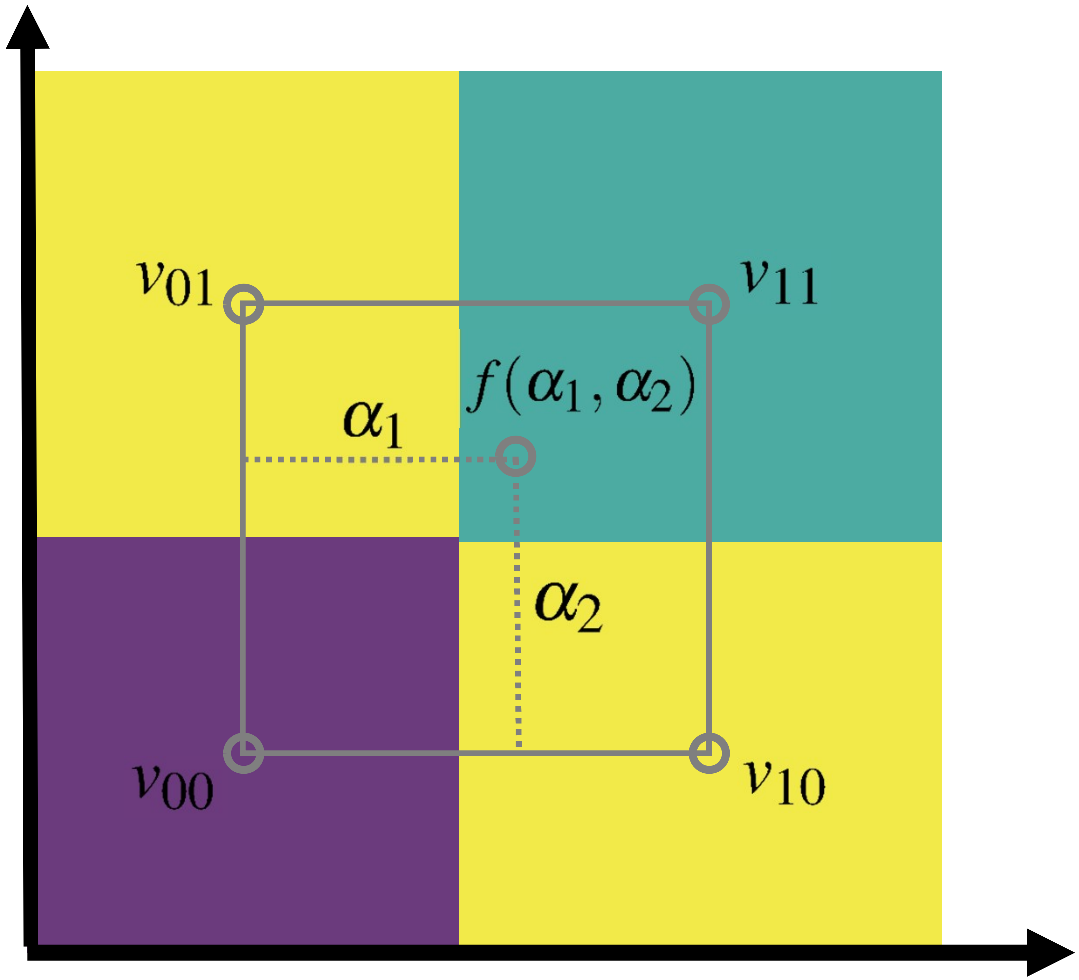
Given any (fractional) position

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

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):

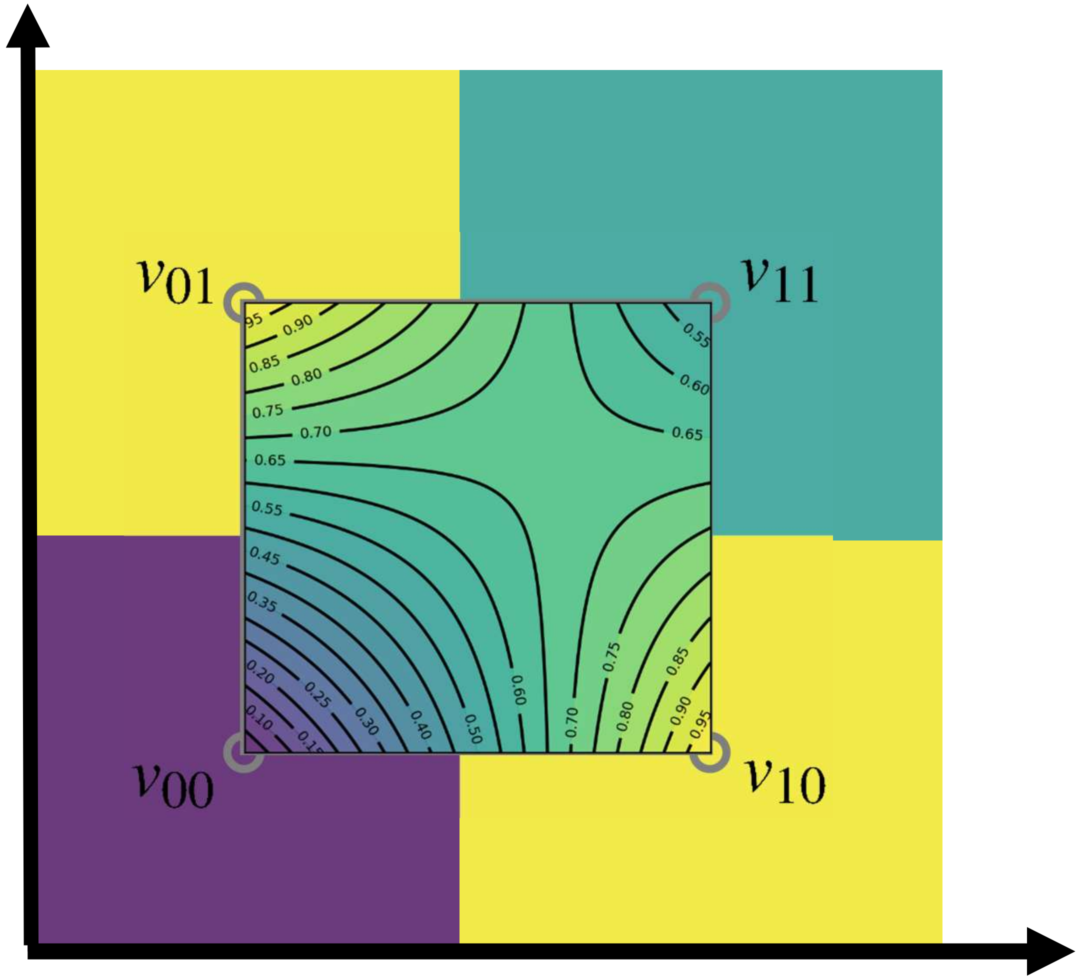
Given any (fractional) position

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

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 (α_1, α_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 (α_1, α_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 (α_1, α_2) :

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

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

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

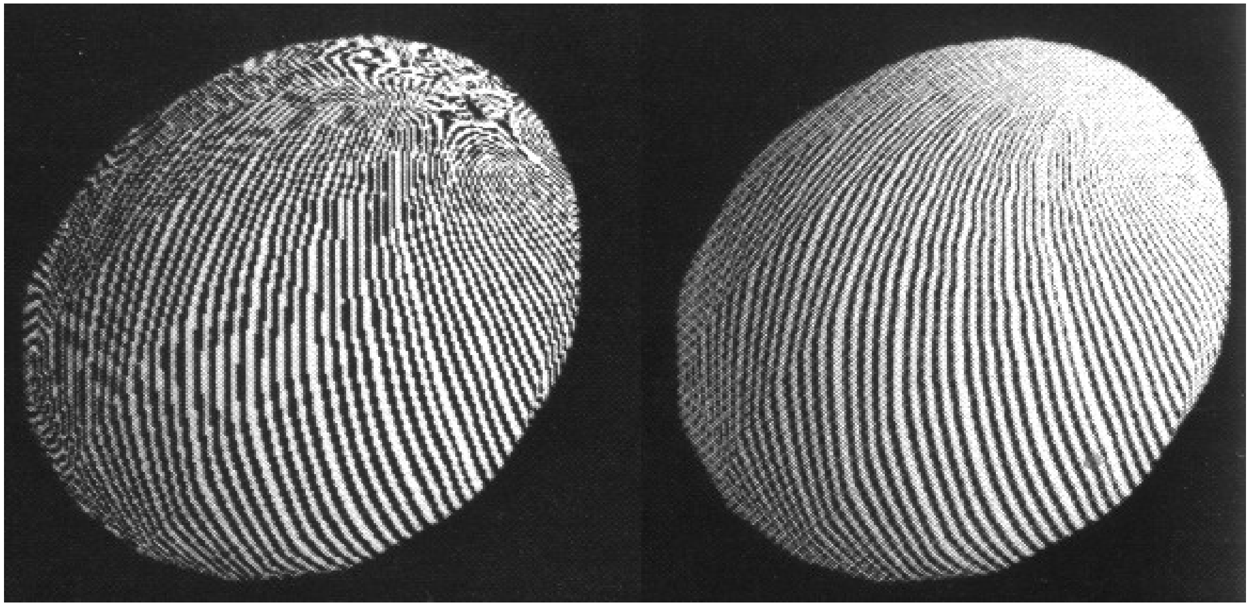
REALLY IMPORTANT:

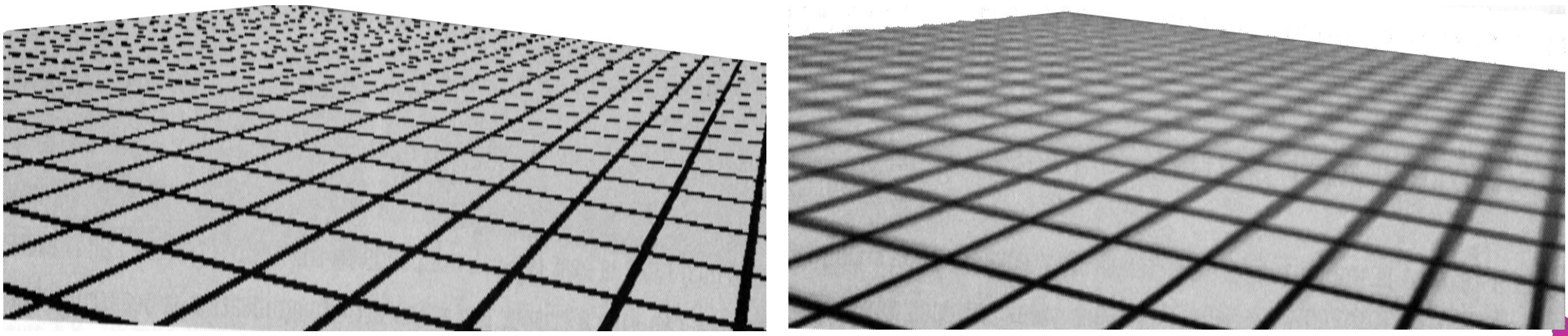
this is a different thing (for a different purpose) than the linear (or, in perspective, rational-linear) interpolation of texture coordinates!!

Texture Minification

Texture Aliasing: Minification

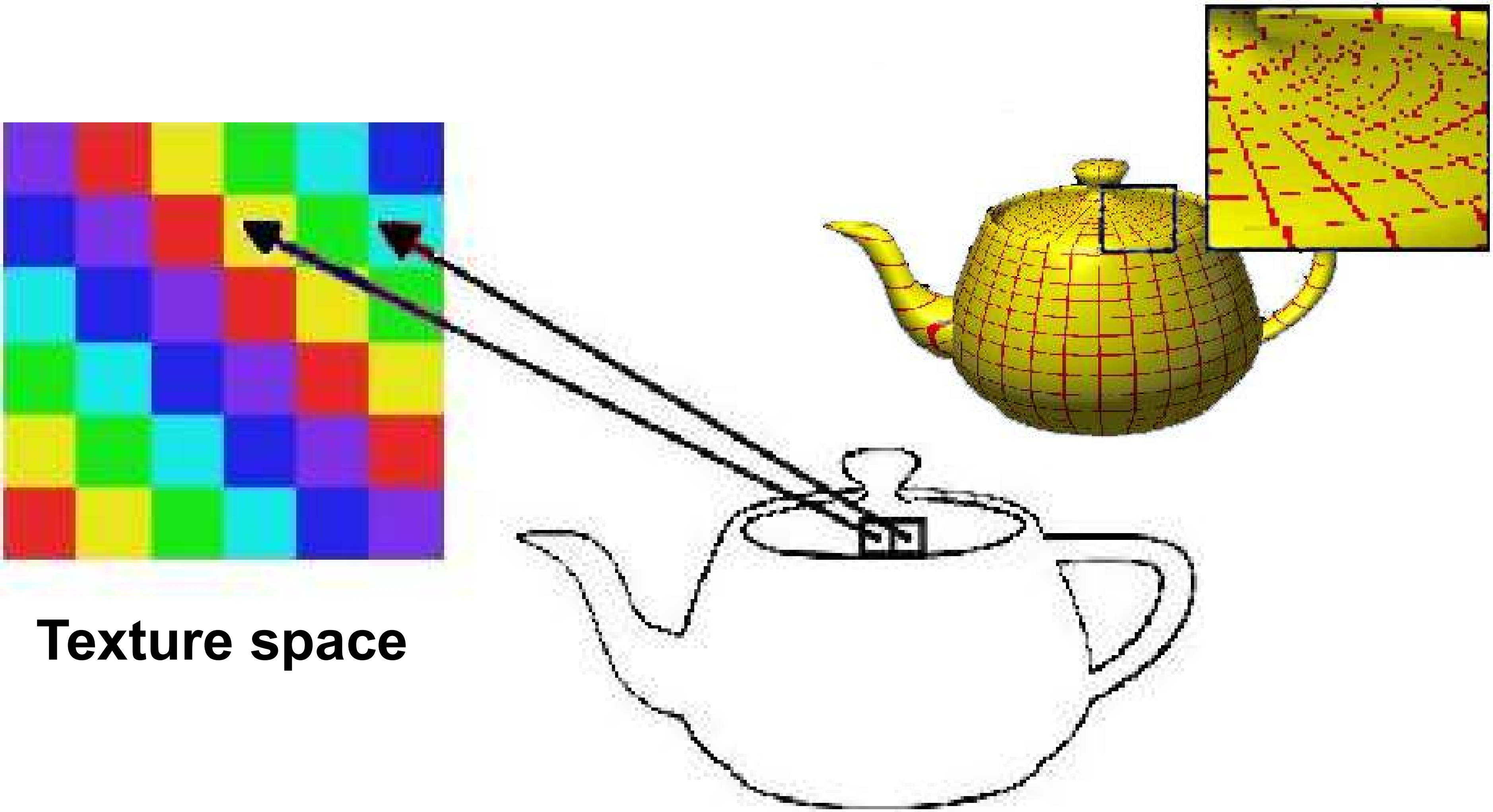
- Problem: One pixel in image space covers many texels





Texture Aliasing: Minification

- Caused by *undersampling*: texture information is lost

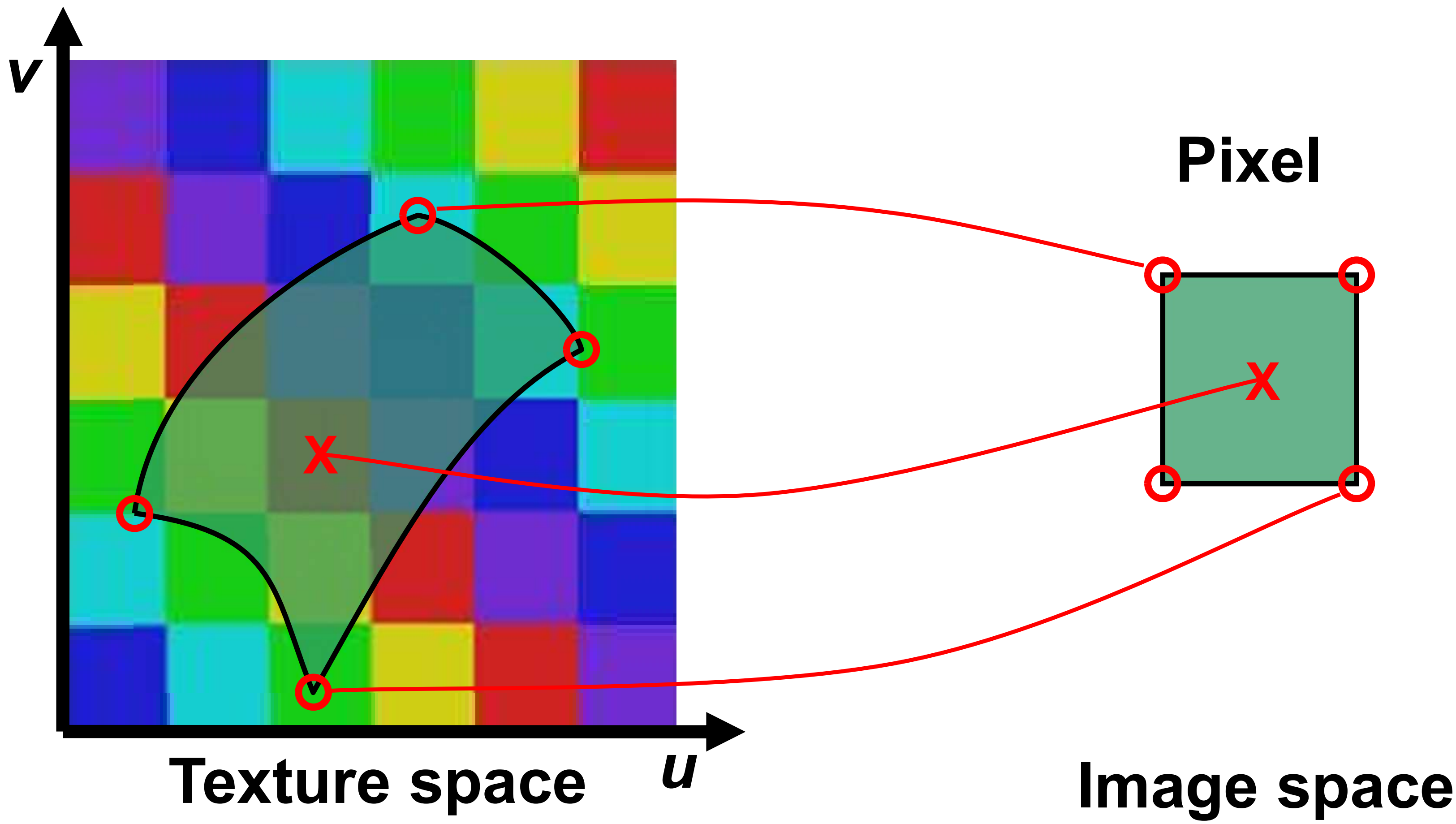


Texture space

Image space

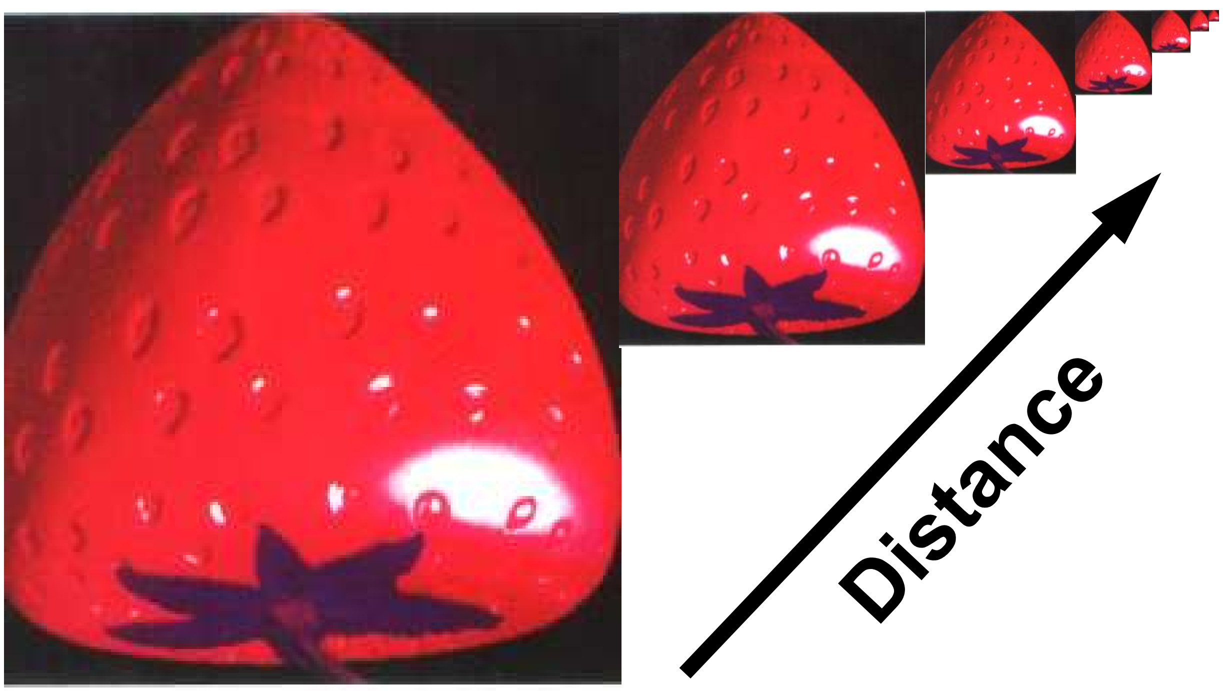
Texture Anti-Aliasing: Minification

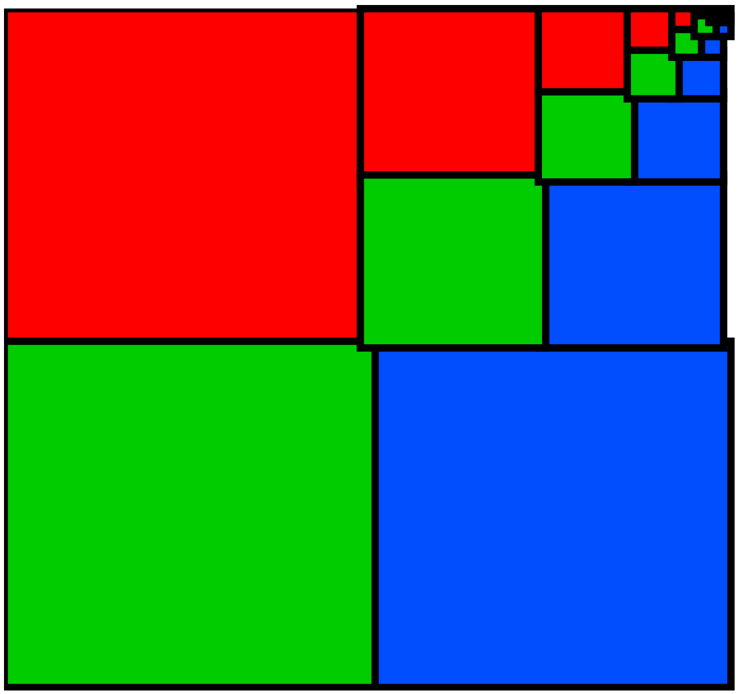
- A good pixel value is the weighted mean of the pixel area projected into texture space



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



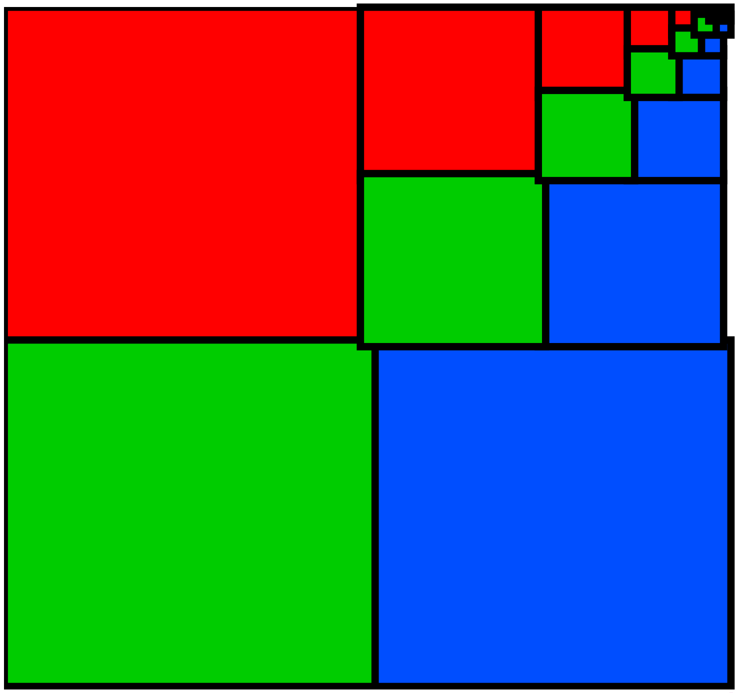


Texture Anti-Aliasing: MIP Mapping

- 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

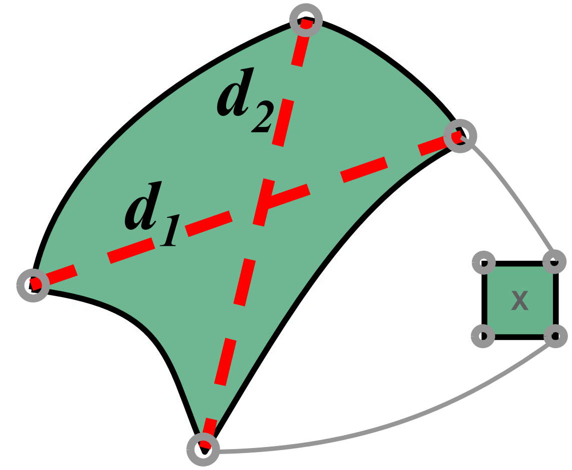
geometric series:

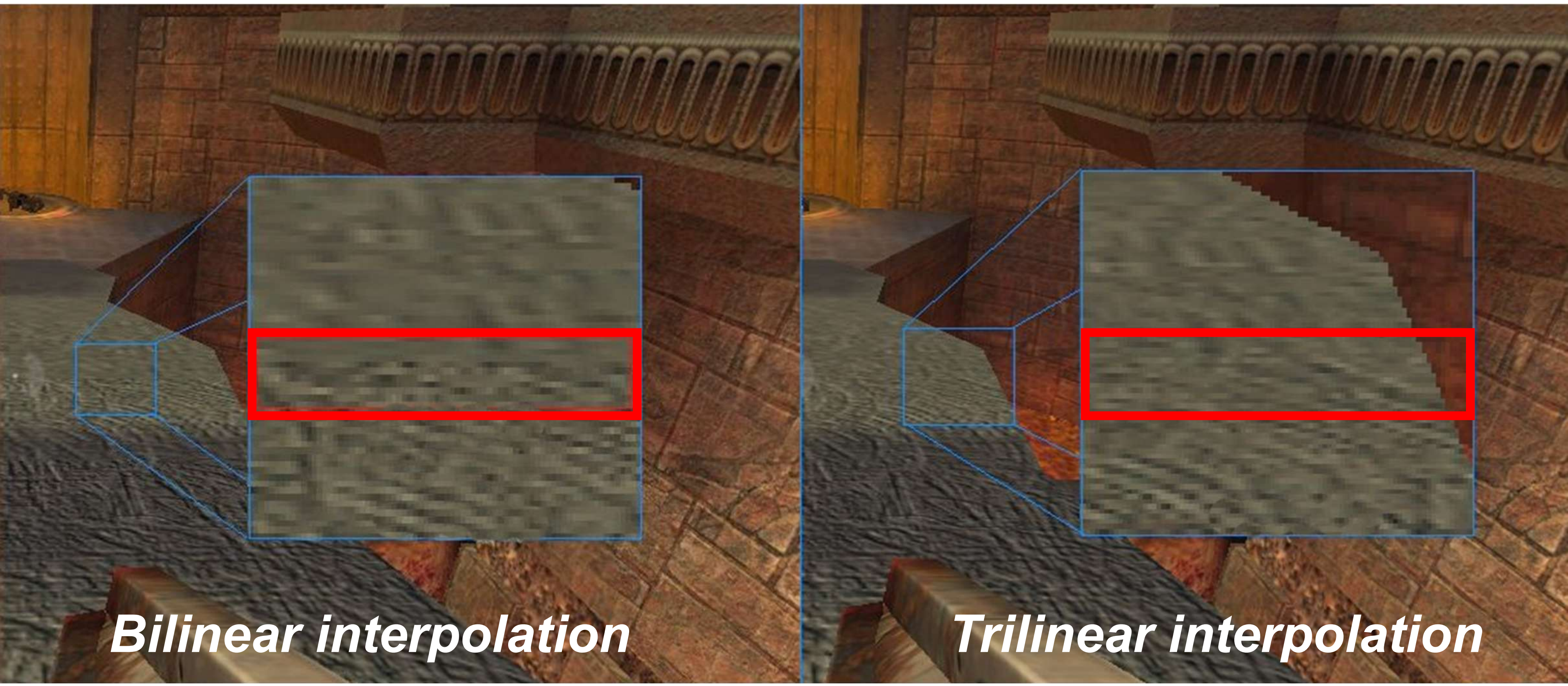
$$a + ar + ar^2 + ar^3 + \cdots + ar^{n-1} =$$
$$= \sum_{k=0}^{n-1} ar^k = a\left(\frac{1-r^n}{1-r}\right)$$



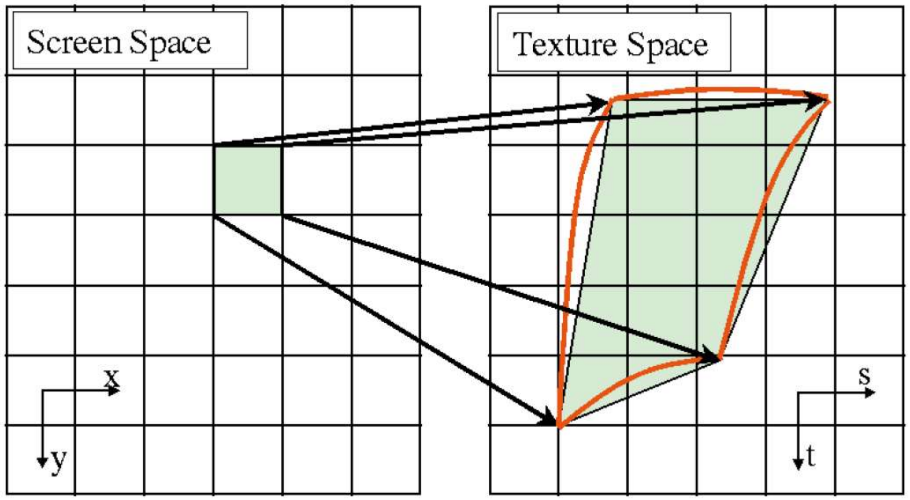
Texture Anti-Aliasing: MIP Mapping

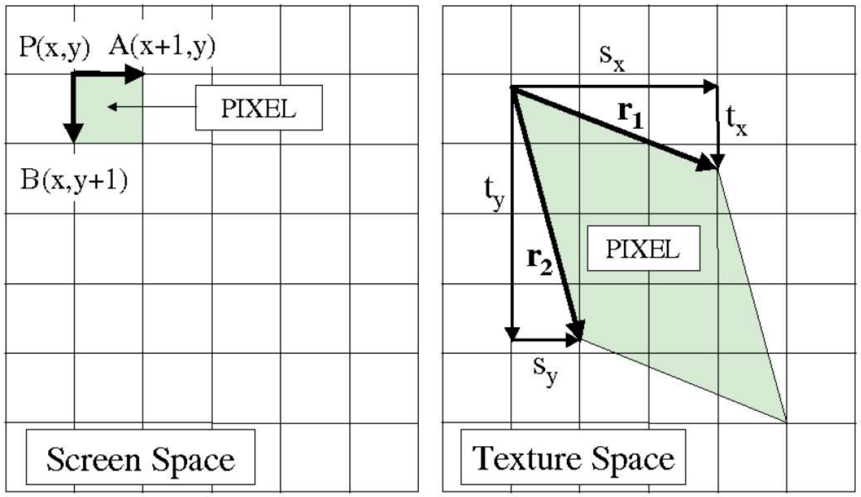
- MIP Mapping Algorithm
- $D := ld(max(d_1,d_2))$ "Mip Map level"
- $T_0 :=$ value from texture $D_0 = trunc\ (D)$
  - Use *bilinear interpolation*





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!)

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

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

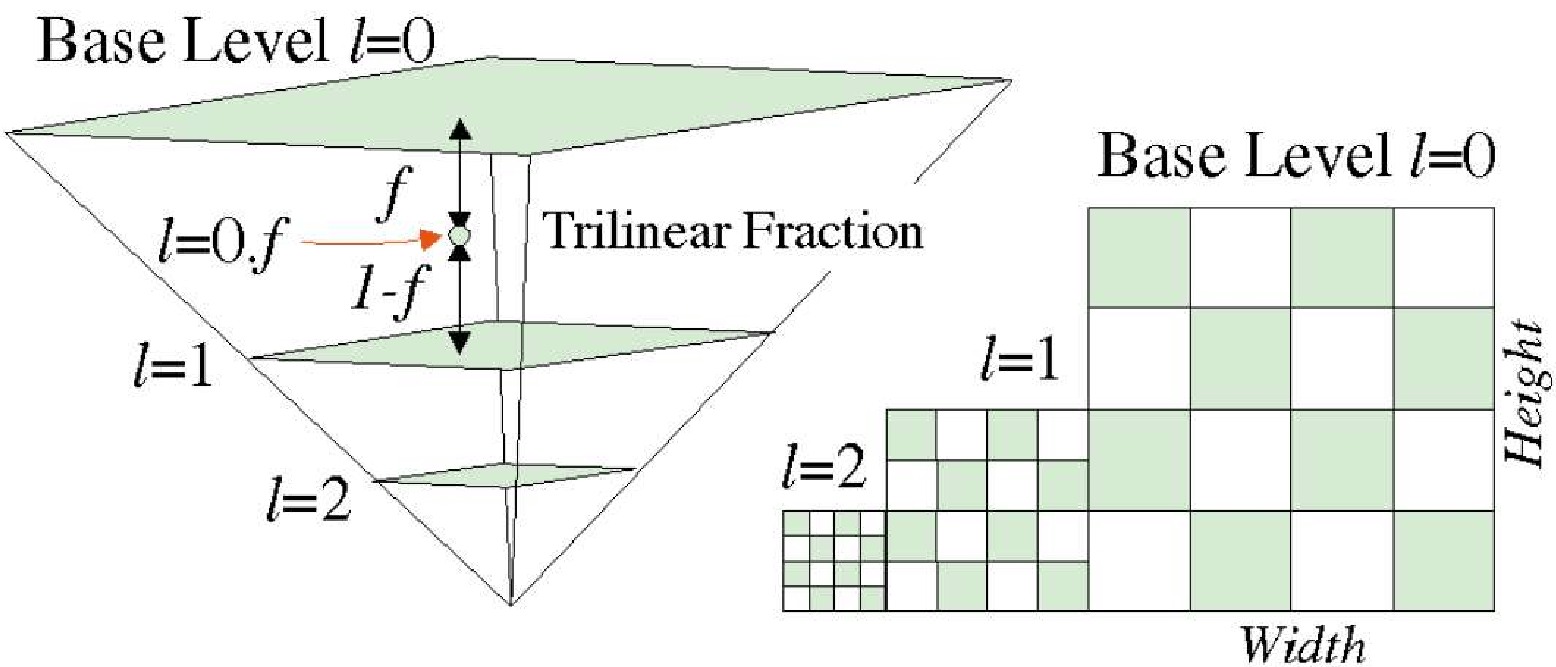
Does not use area of parallelogram but greater hypotenuse [Heckbert, 1983]

- Approximation without square-roots

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

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

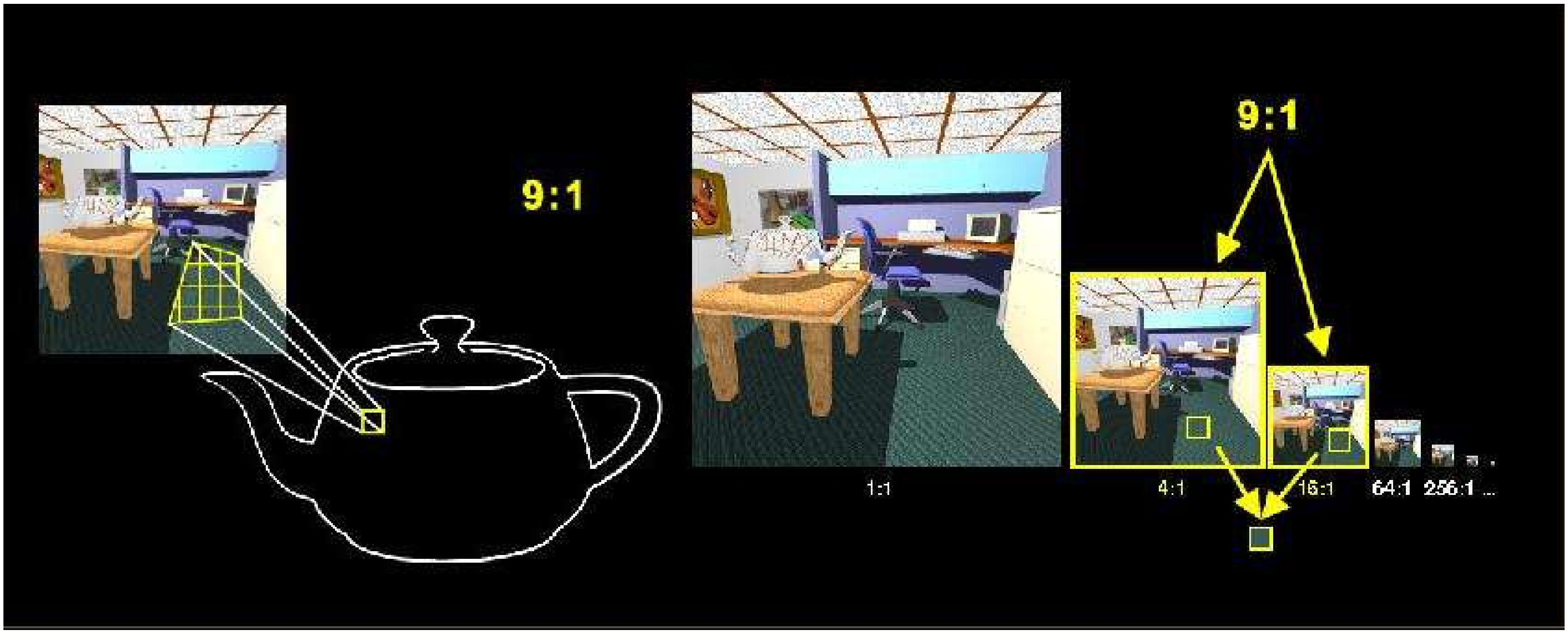
MIP-Map Level Interpolation



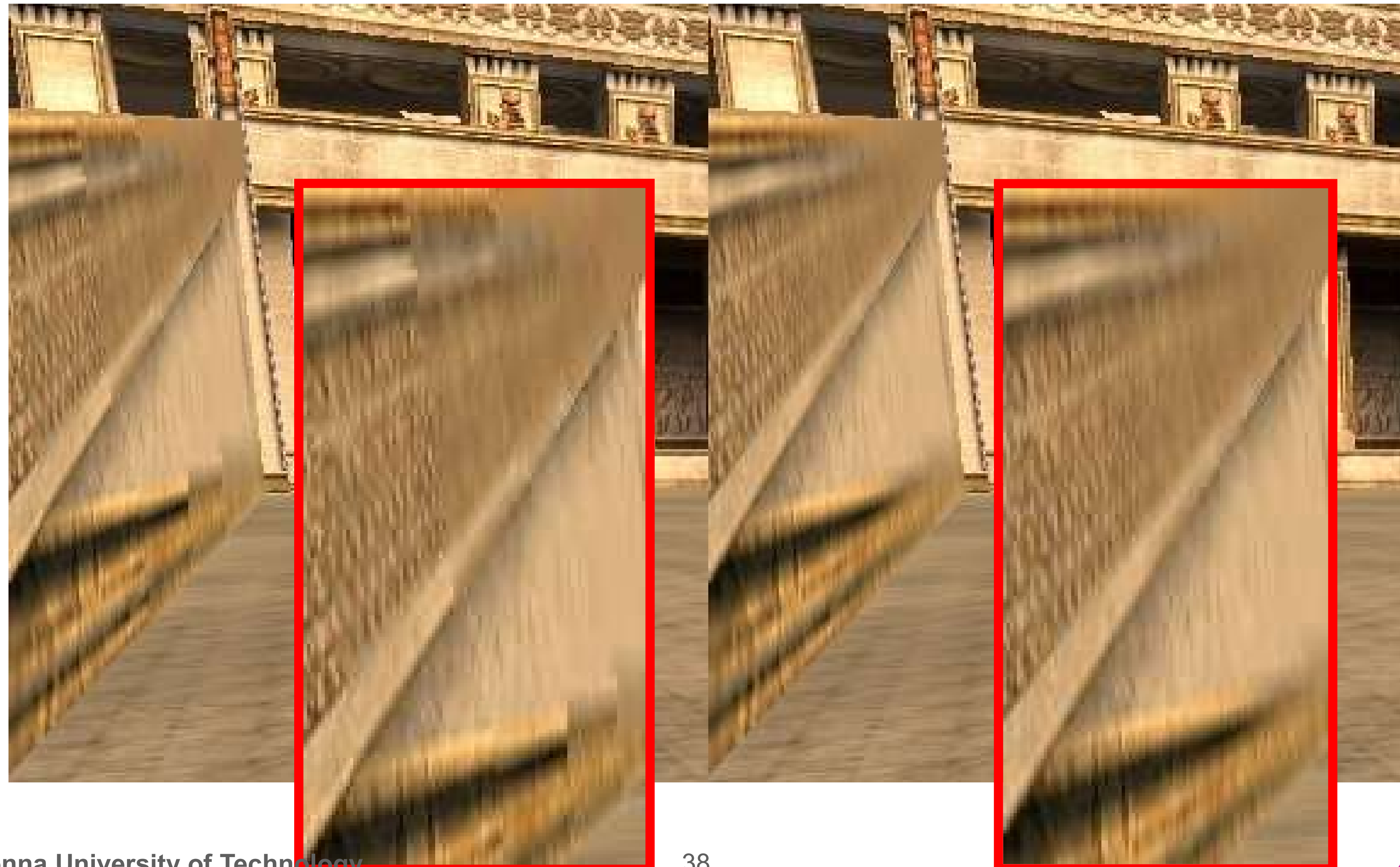
- Level of detail value is fractional!
- Use fractional part to blend (lin.) between two adjacent mipmap levels

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)

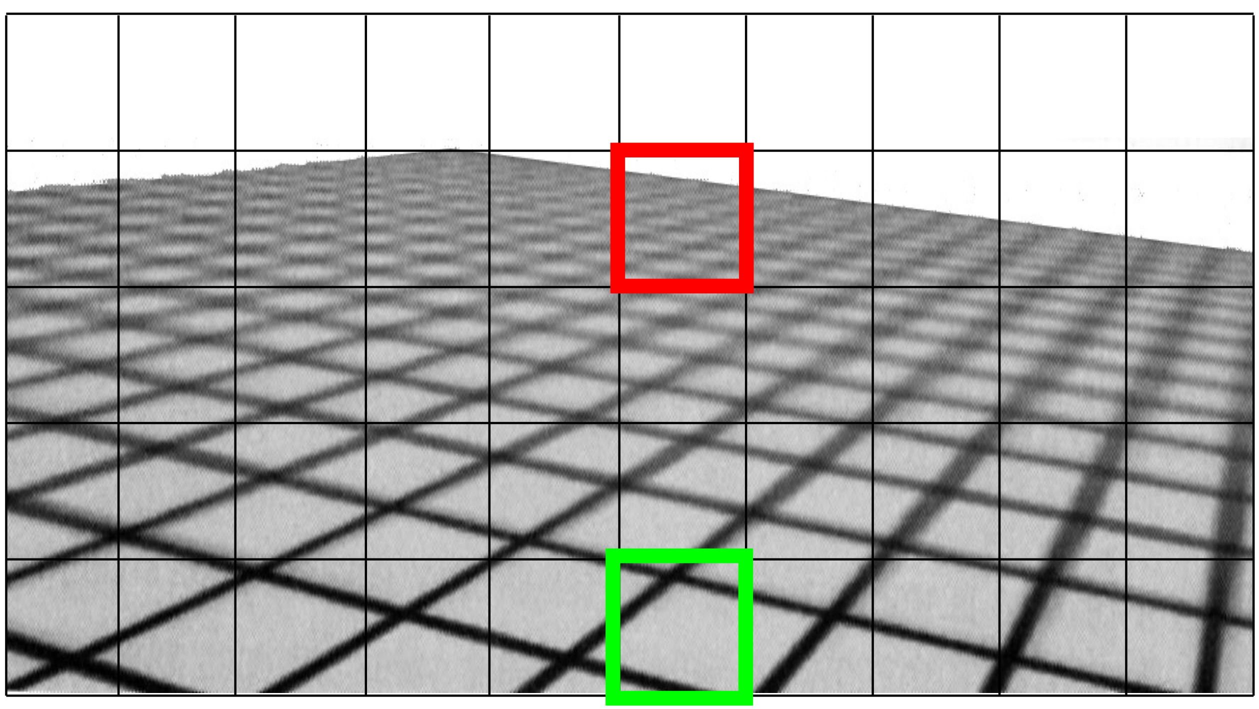


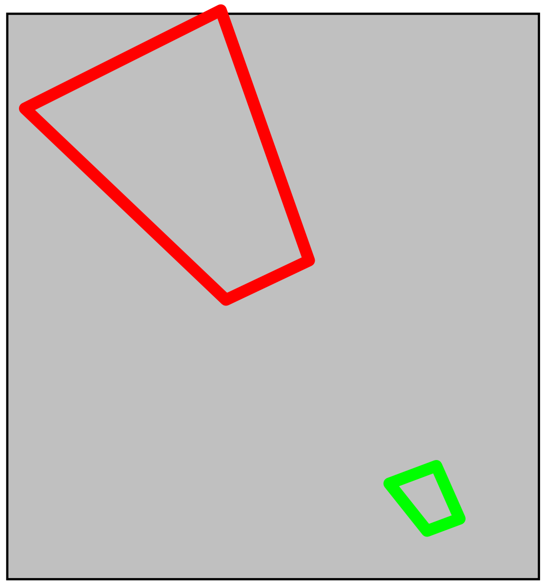
- Other example for bilinear vs. trilinear filtering



Anti-Aliasing: Anisotropic Filtering

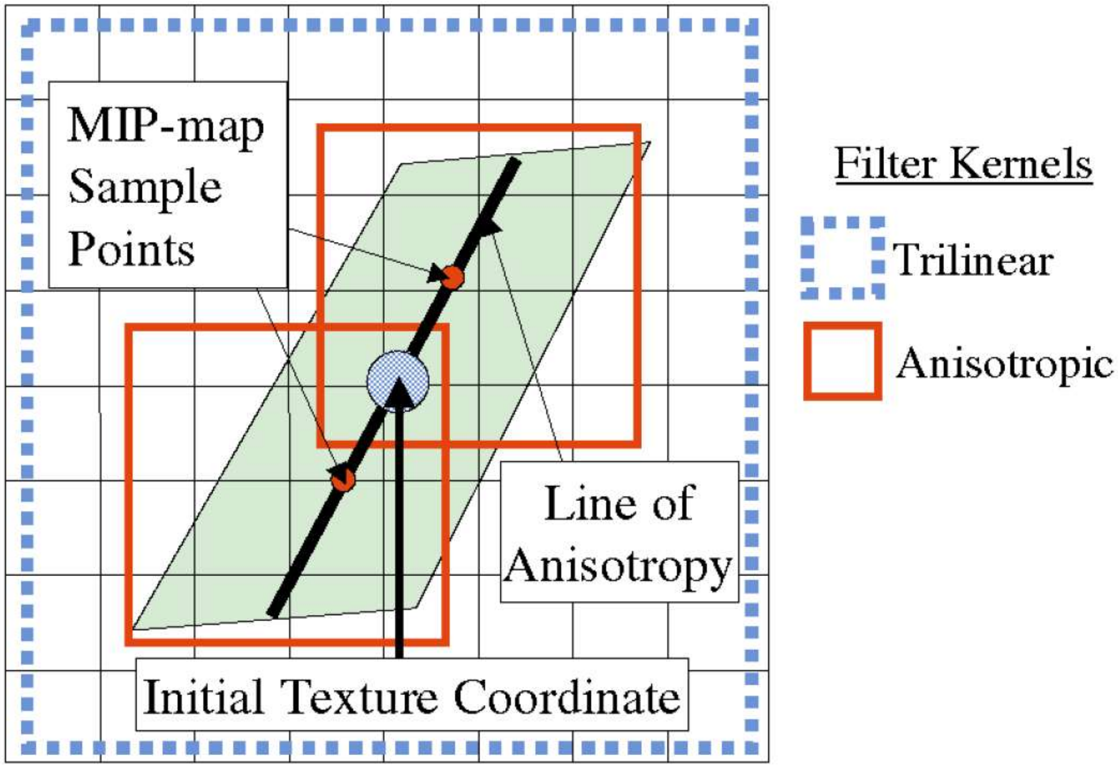
- Anisotropic filtering
  - View-dependent filter kernel
  - Implementation: *summed area table*, "*RIP Mapping*", *footprint assembly*, *elliptical weighted average* (EWA)





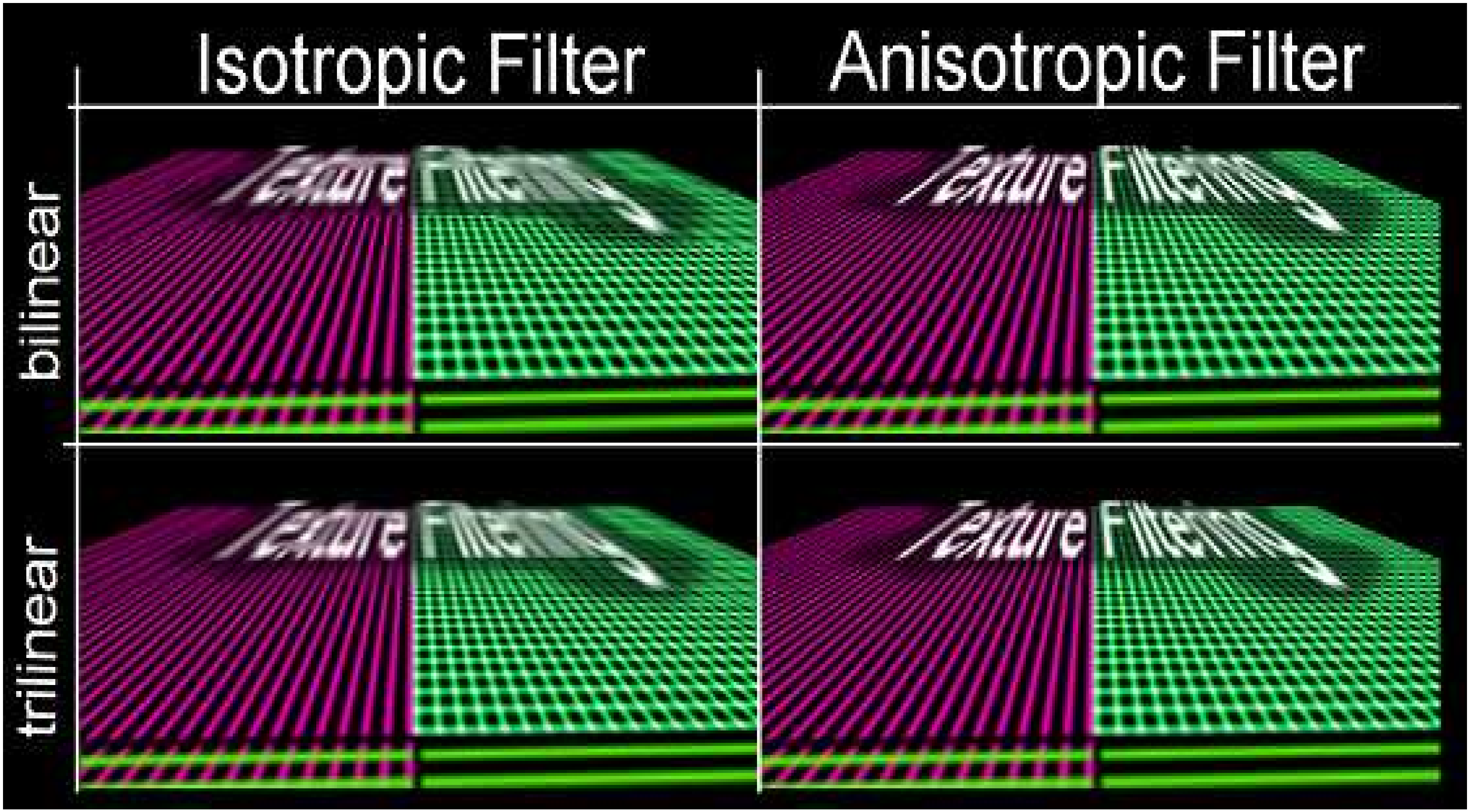
Texture space

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.