

CS 380 - GPU and GPGPU Programming

Lecture 24: Graphics Pipelines; GPU Texturing, Pt. 1

Markus Hadwiger, KAUST



Reading Assignment #9 (until Nov 4)

Read (required):

- Programming Massively Parallel Processors book, 4th edition
Chapter 11: Prefix Sum (Scan) – an introduction to work efficiency in parallel algorithms
- Warp Shuffle Functions
 - CUDA Programming Guide, Chapter 10.22 (pdf; 7.22 online)

Read (optional):

- Guy E. Blelloch: Prefix Sums and their Applications
 - [https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf/](https://www.cs.cmu.edu/~guyb/papers/Ble93.pdf)
- CUDA Cooperative Groups
 - CUDA Programming Guide, Chapter 11 (pdf; 8 online)
 - <https://developer.nvidia.com/blog/cooperative-groups/>
- Warp Matrix Functions (==tensor core programming)
 - CUDA Programming Guide, Chapter 10.24 (pdf; 7.24 online)



Next Lectures

Lecture 25: Mon, Nov 4

Lecture 26: Tue, Nov 5 (make-up lecture; 14:30 – 15:45)

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



What is in a GPU?

Lots of floating point processing power

- Stream processing cores
 - different names:
stream processors,
CUDA cores, ...
- Was vector processing, now scalar cores!

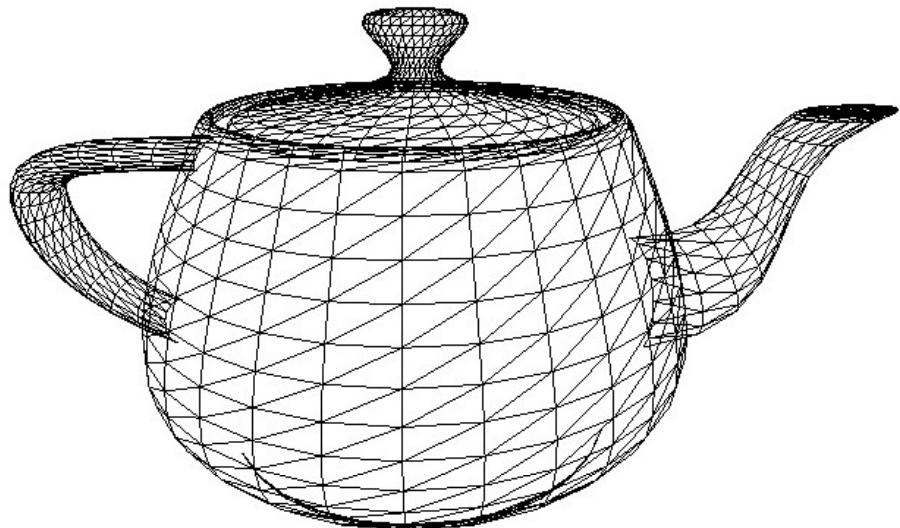


Still lots of fixed graphics functionality

- Attribute interpolation (per-vertex -> per-fragment)
- Rasterization (turning triangles into fragments/pixels)
- Texture sampling and filtering
- Depth buffering (per-pixel visibility)
- Blending/compositing (semi-transparent geometry, ...)
- Frame buffers



Real-time graphics primitives (entities)



Represent surface as a 3D triangle mesh

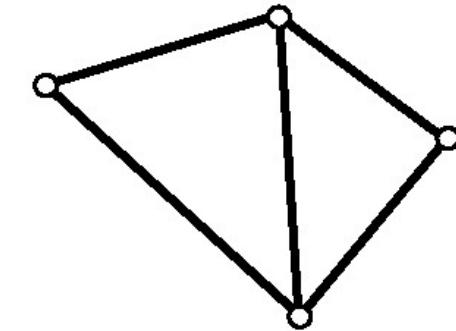
o 1

o 3

o 2

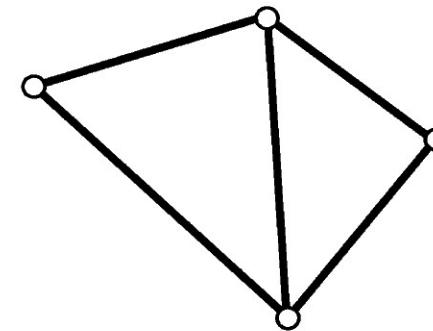
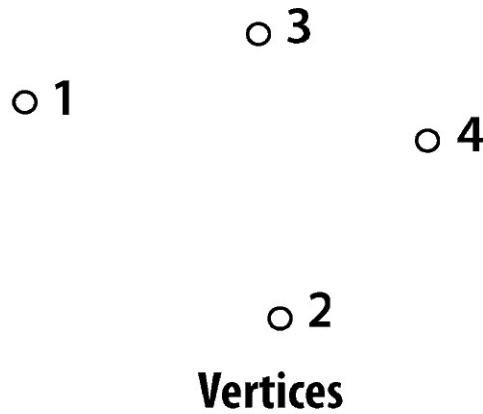
o 4

Vertices

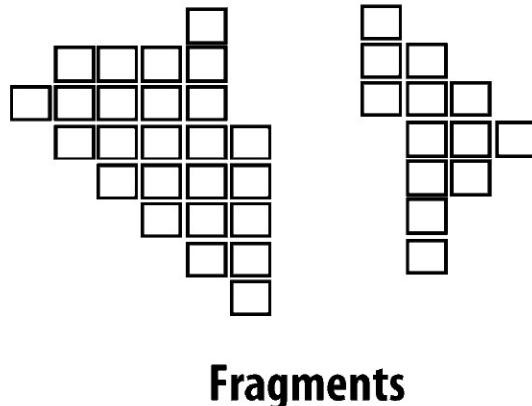


Primitives
(e.g., triangles, points, lines)

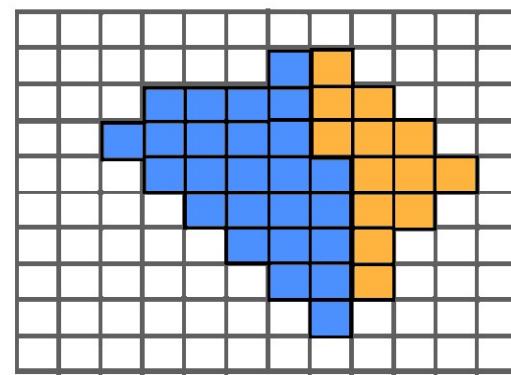
Real-time graphics primitives (entities)



Primitives
(e.g., triangles, points, lines)

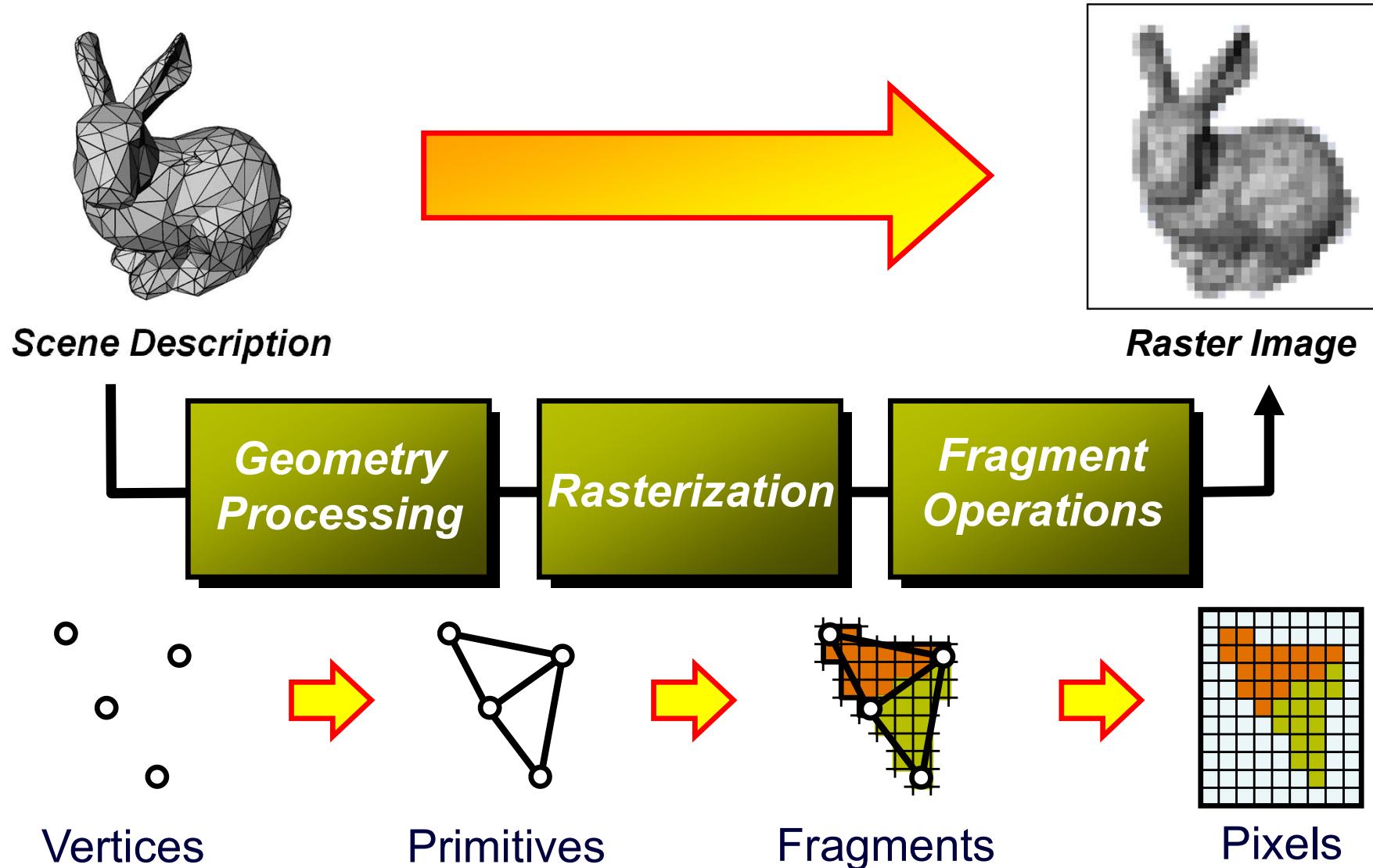


Fragments

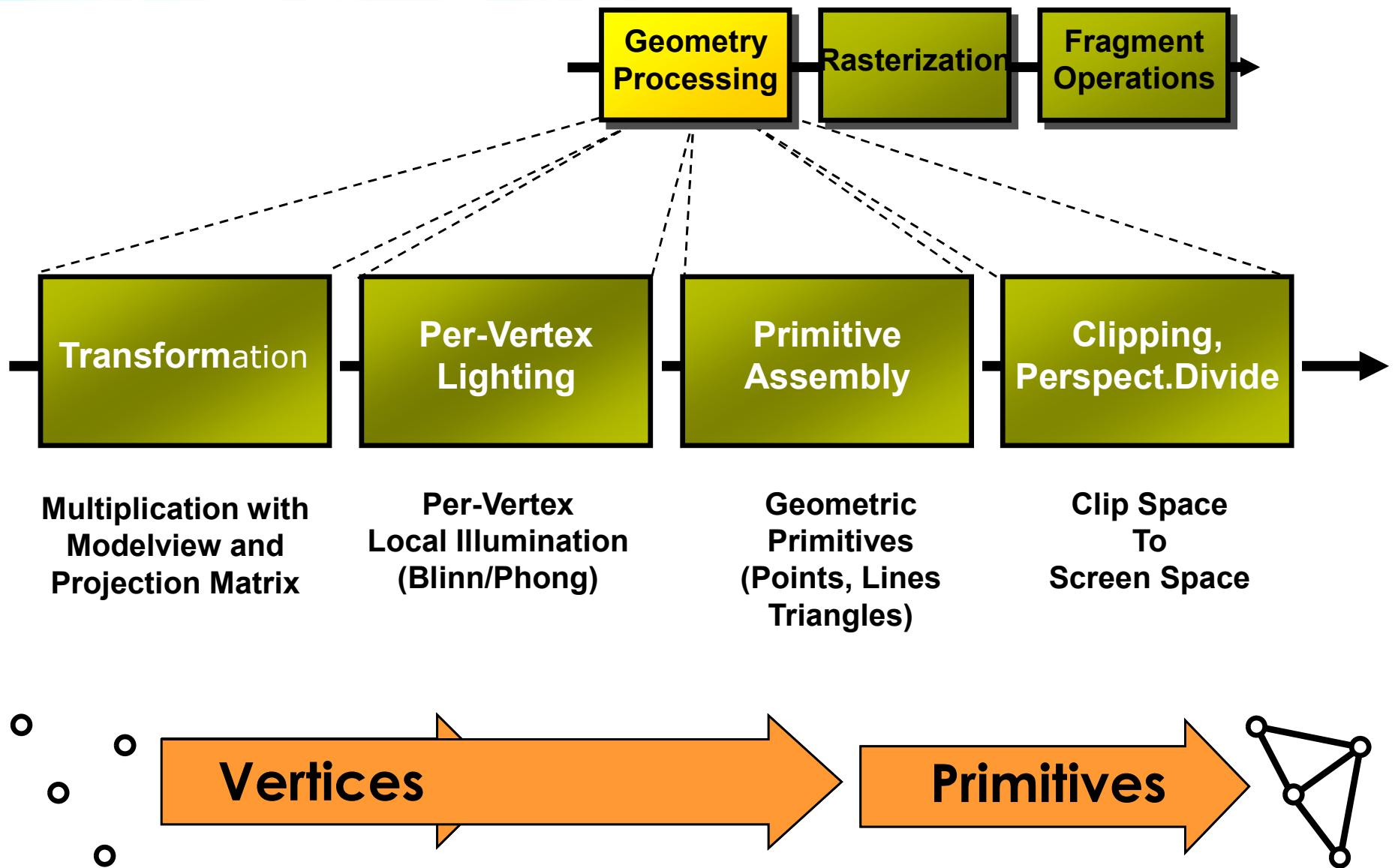


Pixels (in an image)

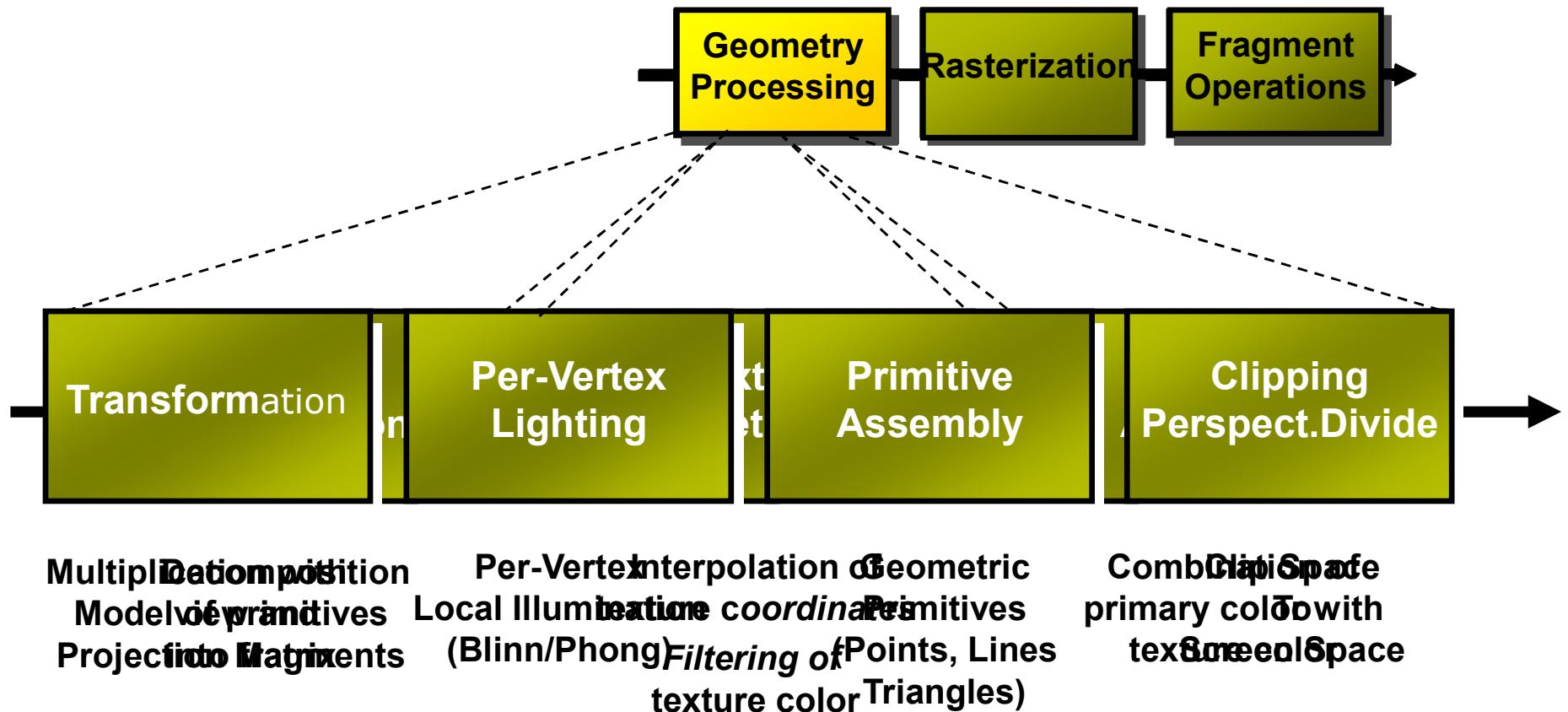
Traditional (GPU) Graphics Pipeline



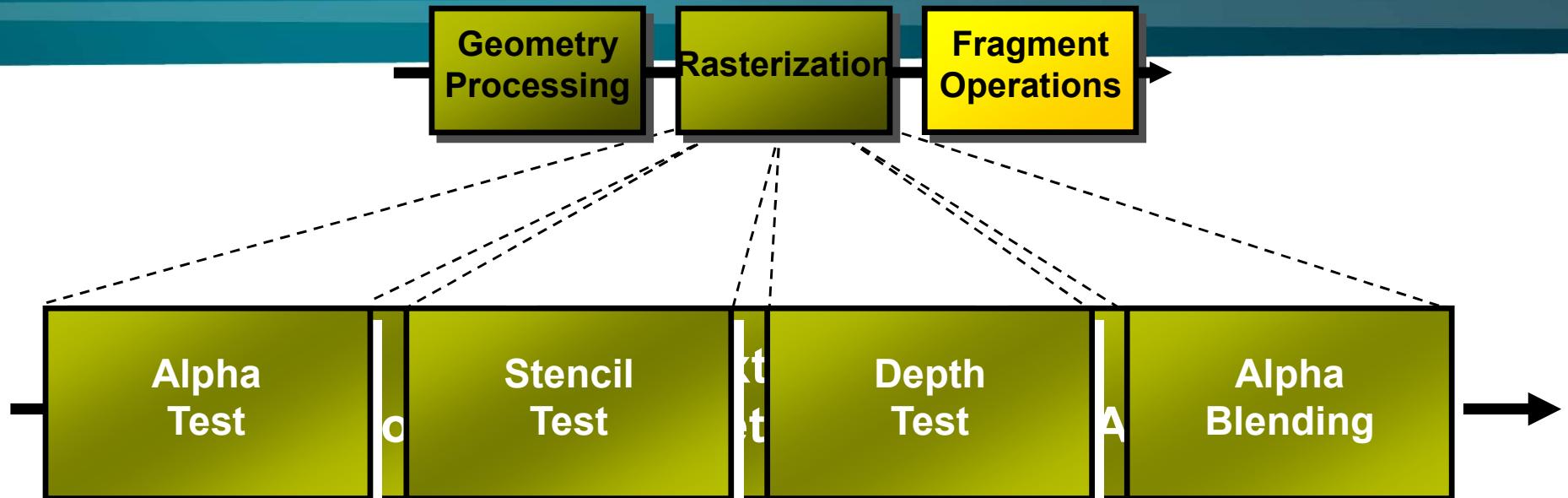
Geometry Processing



Rasterization Processing



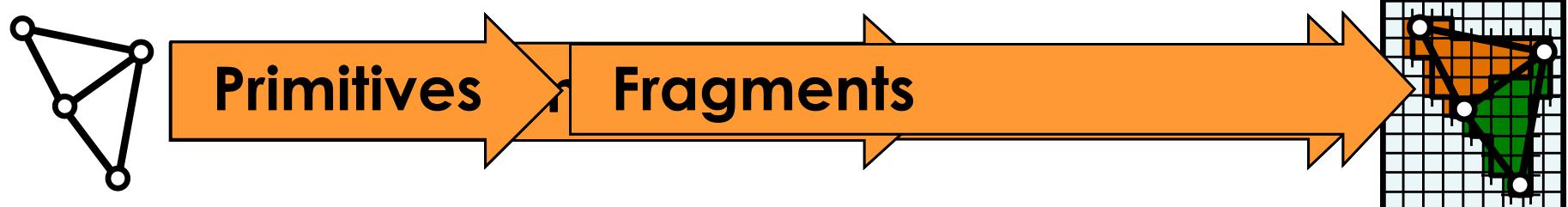
Rasterization Operations



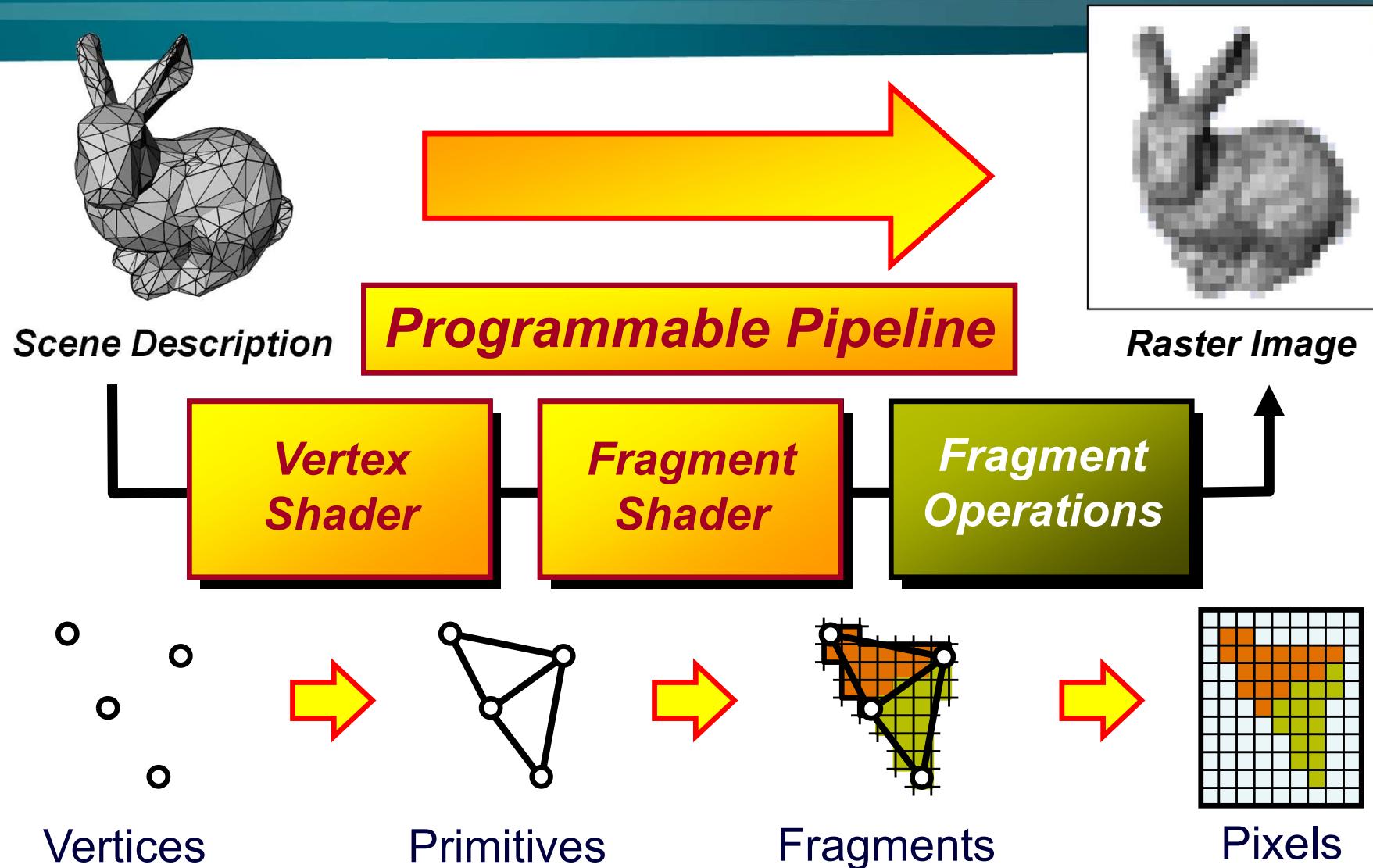
Discard all fragments with primitives
a deltafragments
alpha range

Discard interpolation
fragment texture coordinates
the stencil buffer is set
Filtering offragments
texture color

Combine the fragment
principally with the color
texture and polar
frame buffer

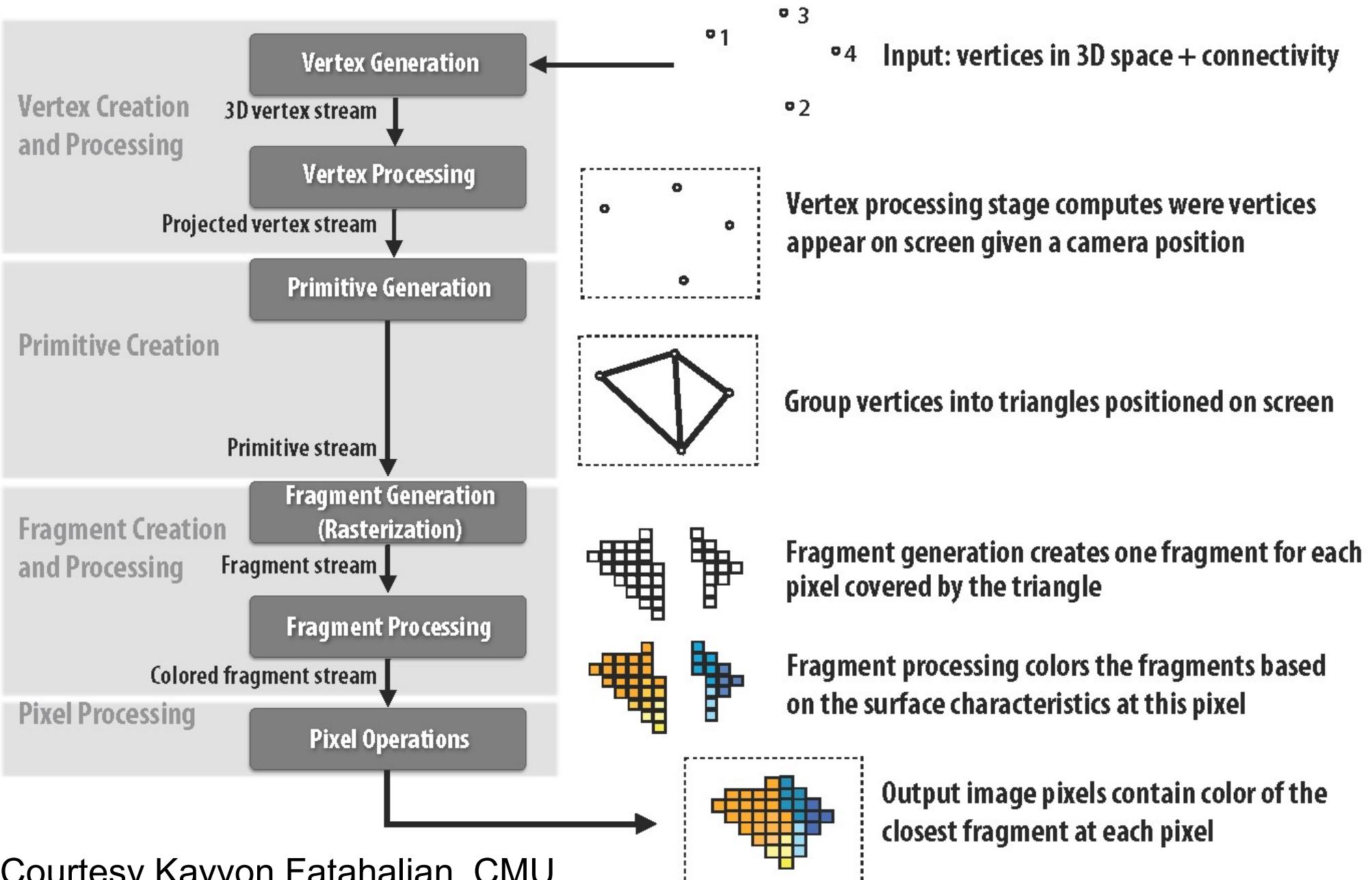


Programmable Pipeline: Shaders



Graphics pipeline architecture

Performs operations on vertices, triangles, fragments, and pixels



Courtesy Kayvon Fatahalian, CMU

CMU 15-418, Spring 2015

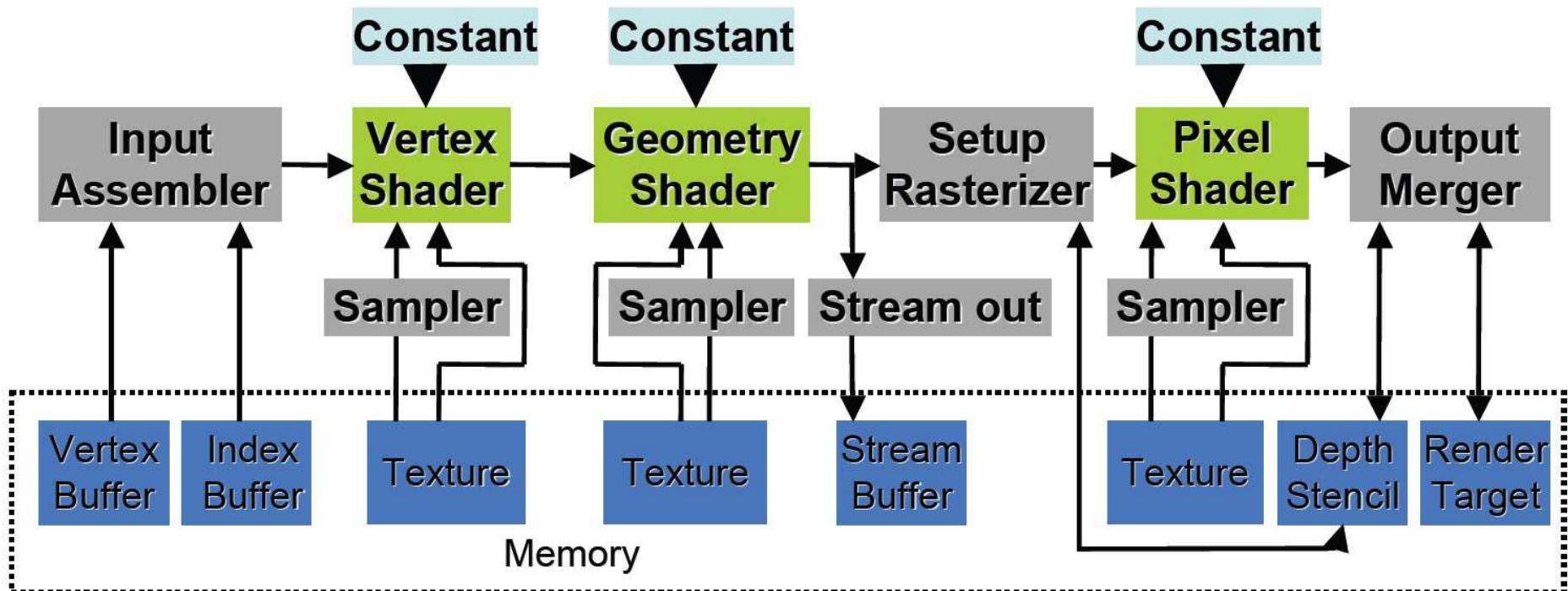
Direct3D 10 Pipeline (~OpenGL 3.2)



New geometry shader stage:

- Vertex -> geometry -> pixel shaders
- Stream output after geometry shader

■ fixed
■ programmable
■ memory

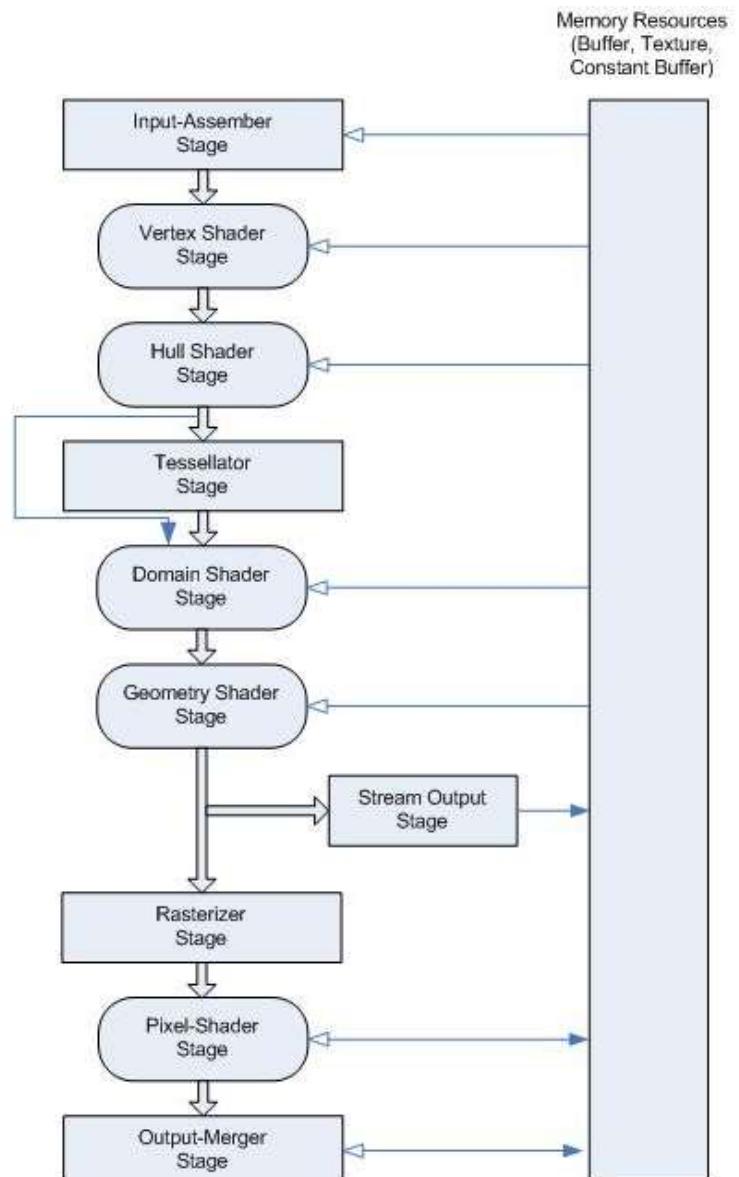




Direct3D 11 Pipeline (~OpenGL 4.x)

New tessellation stages

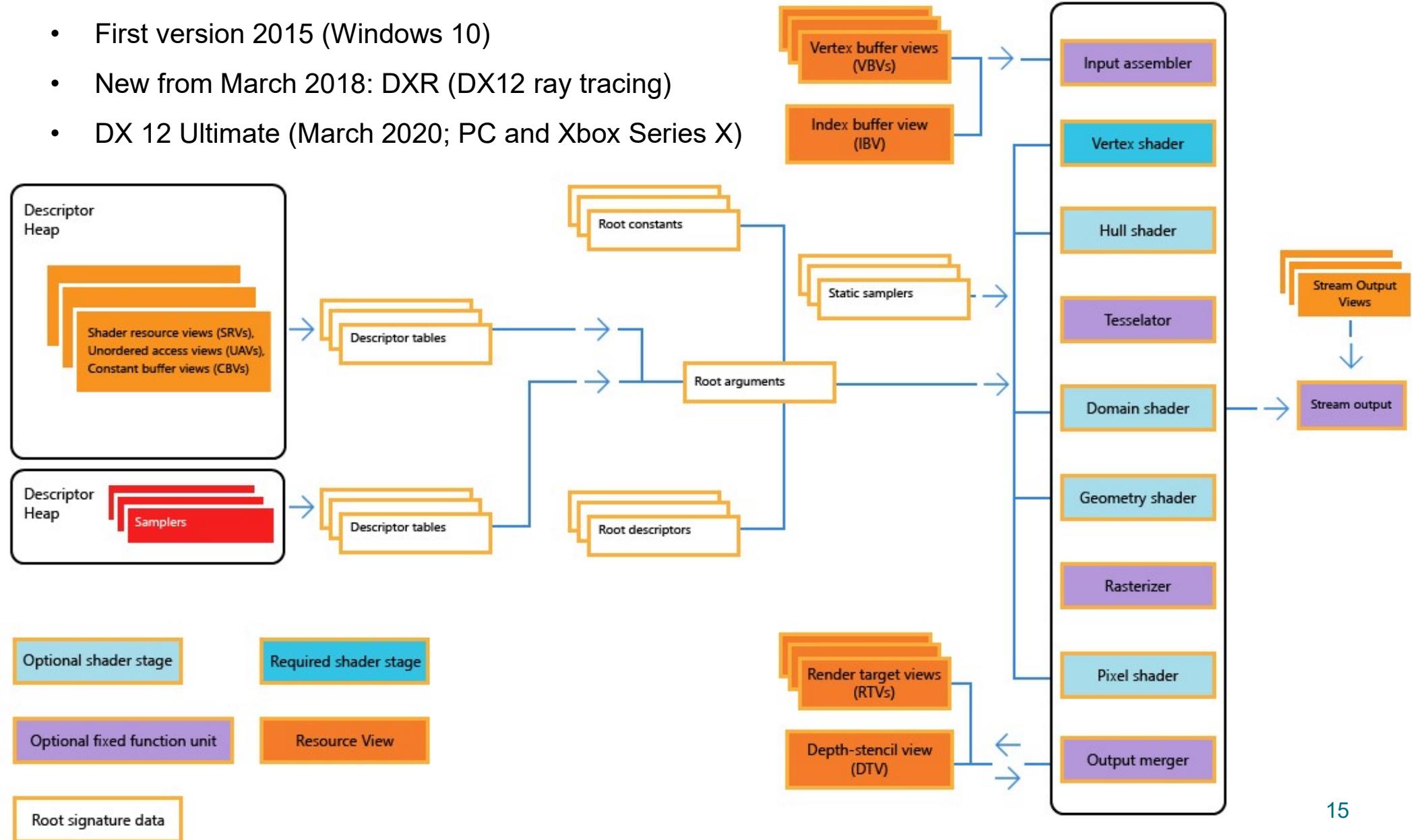
- Hull shader
(OpenGL: *tessellation control*)
- Tessellator
(OpenGL: *tessellation primitive generator*)
- Domain shader
(OpenGL: *tessellation evaluation*)
- In future versions, there might be yet more stages, but for some time now all additions were outside this pipeline:
 - Compute shaders
 - Vulkan
 - Ray tracing cores



Direct3D 12 Geometry Pipeline (Traditional)



- First version 2015 (Windows 10)
- New from March 2018: DXR (DX12 ray tracing)
- DX 12 Ultimate (March 2020; PC and Xbox Series X)

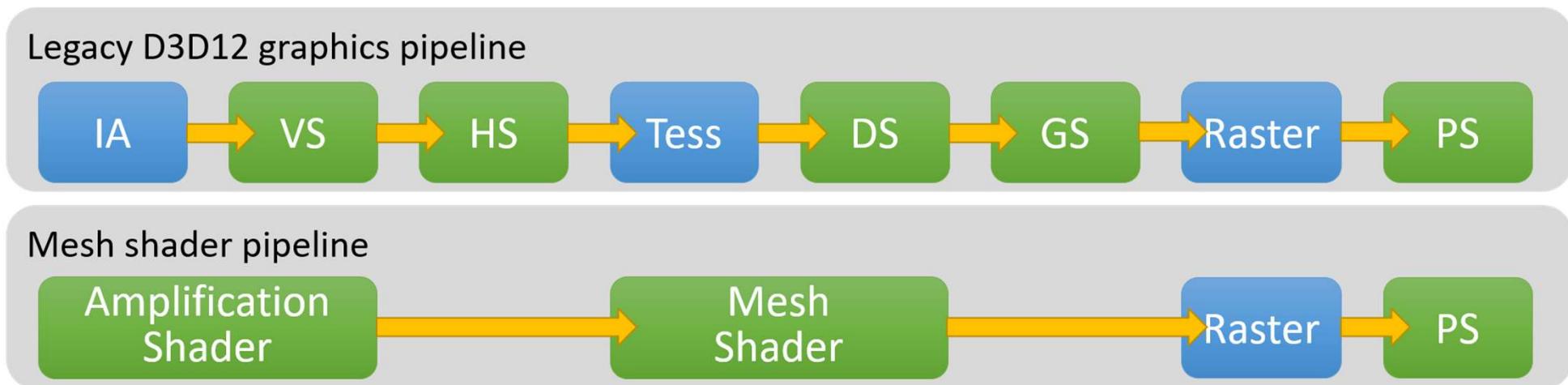




Direct3D 12 Mesh Shader Pipeline

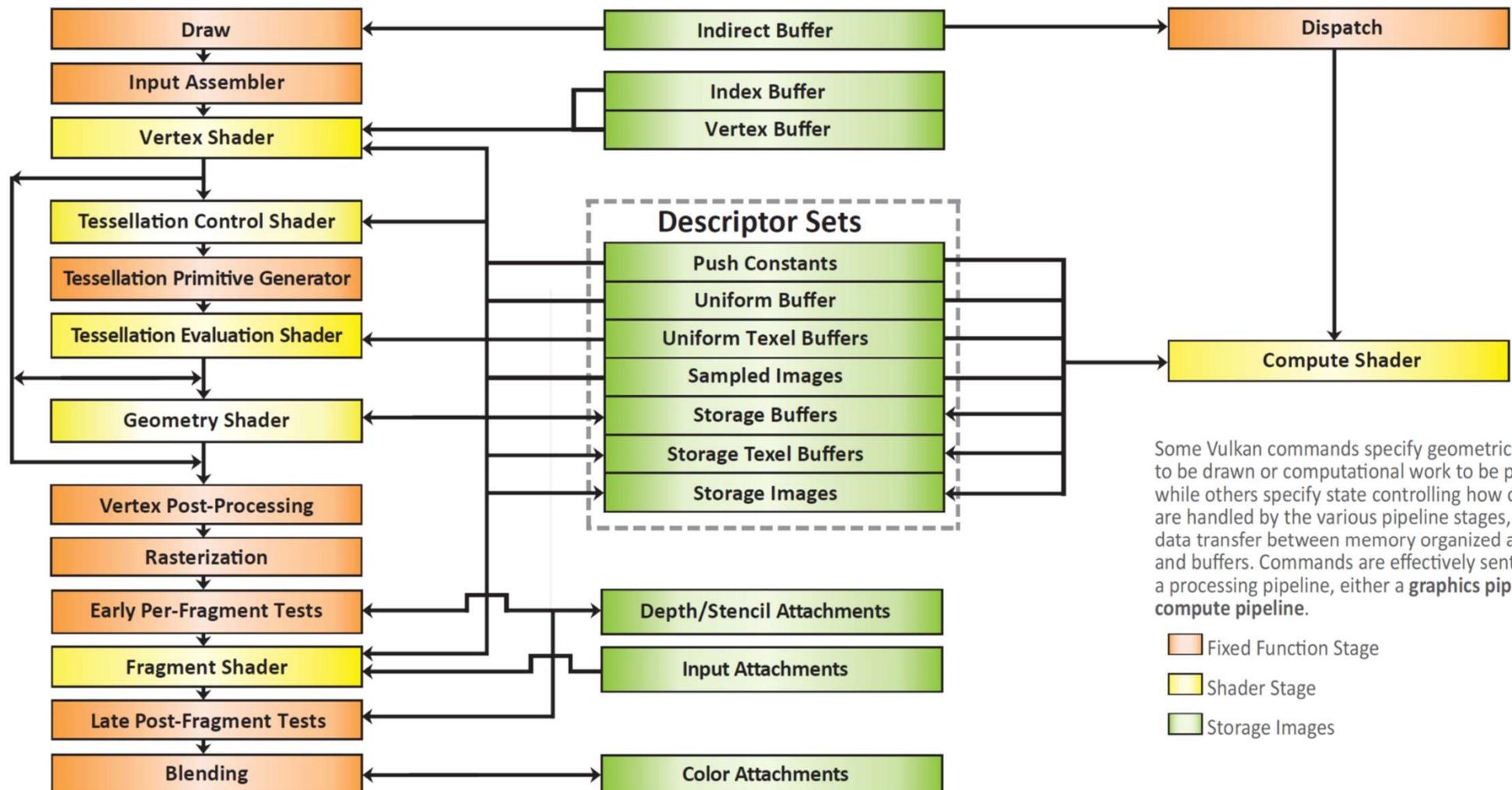
Reinventing the Geometry Pipeline

- Mesh and amplification shaders: new high-performance geometry pipeline based on compute shaders (DX 12 Ultimate / feature level 12.2)
- Compute shader-style replacement of IA/VS/HS/Tess/DS/GS



See talk by Shawn Hargreaves: <https://www.youtube.com/watch?v=CFXKTXtil34>

Vulkan (1.3) Pipeline (Traditional)



Some Vulkan commands specify geometric objects to be drawn or computational work to be performed, while others specify state controlling how objects are handled by the various pipeline stages, or control data transfer between memory organized as images and buffers. Commands are effectively sent through a processing pipeline, either a **graphics pipeline** or a **compute pipeline**.

- Fixed Function Stage
- Shader Stage
- Storage Images

Vulkan (1.3) Pipelines



- Mesh and task shaders: new high-performance geometry pipeline based on compute shaders
(Mesh and task shaders also available as OpenGL 4.5/4.6 extension: GL_NV_mesh_shader)

TRADITIONAL PIPELINE



TASK/MESH PIPELINE

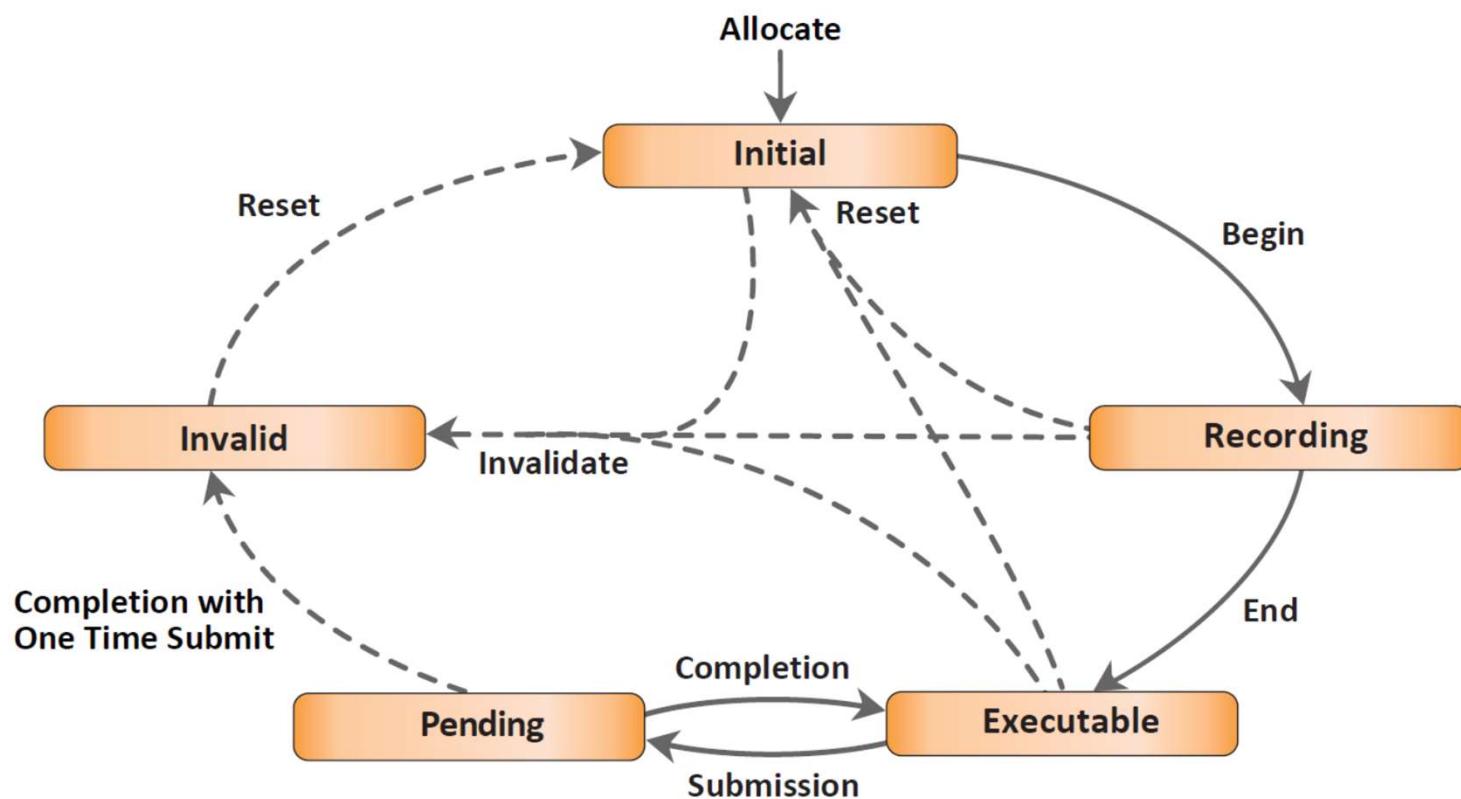


vulkan.org

github.com/KhronosGroup/Vulkan-Guide

<https://www.khronos.org/blog/mesh-shading-for-vulkan>

Vulkan Command Buffer Lifecycle



Initial state

The state when a command buffer is first allocated. The command buffer may be reset back to this state from any of the executable, recording, or invalid states. Command buffers in the initial state can only be moved to recording, or freed.

Recording state

`vkBeginCommandBuffer` changes the state from initial to recording. Once in the recording state, `vkCmd*` commands can be used to record to the command buffer.

Executable state

`vkEndCommandBuffer` moves a command buffer state from recording to executable. Executable command buffers can be submitted, reset, or recorded to another command buffer.

Pending state

Queue submission changes the state from executable to pending, in which applications must not attempt to modify the command buffer in any way. The state reverts back to executable when current executions complete, or to invalid.

Invalid state

Some operations will transition the command buffer into the invalid state, in which it can only be reset or freed.

GPU Texturing

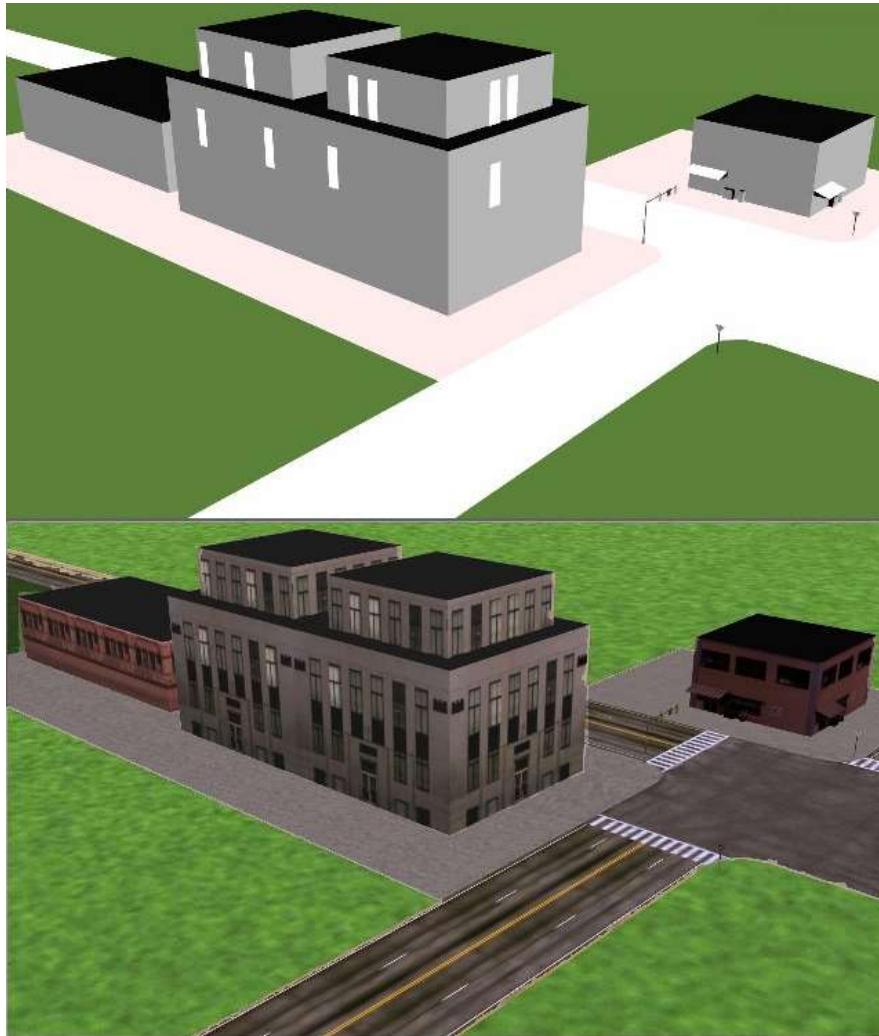
GPU Texturing



Rage / id Tech 5 (id Software)

Why Texturing?

- Idea: enhance visual appearance of surfaces by applying fine / high-resolution details



- Basis for most real-time rendering effects
- Look and feel of a surface
- Definition:
 - A *regularly sampled function* that is mapped onto every *fragment* of a surface
 - Traditionally an image, but...
- Can hold arbitrary information
 - Textures become general data structures
 - Sampled and interpreted by fragment programs
 - Can render into textures → important!



Types of Textures

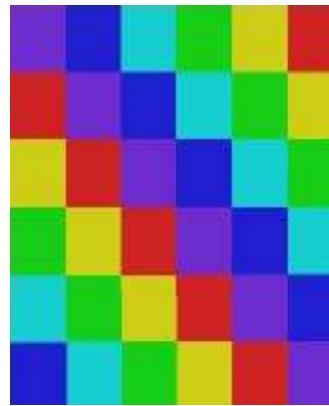
- Spatial layout
 - Cartesian grids: 1D, 2D, 3D, 2D_ARRAY, ...
 - Cube maps, ...
- Formats (too many), e.g. OpenGL
 - GL_LUMINANCE16_ALPHA16
 - GL_RGB8, GL_RGBA8, ...: integer texture formats
 - GL_RGB16F, GL_RGBA32F, ...: float texture formats
 - compressed formats, high dynamic range formats, ...
- 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`



Texturing: General Approach



Texture space (u, v)



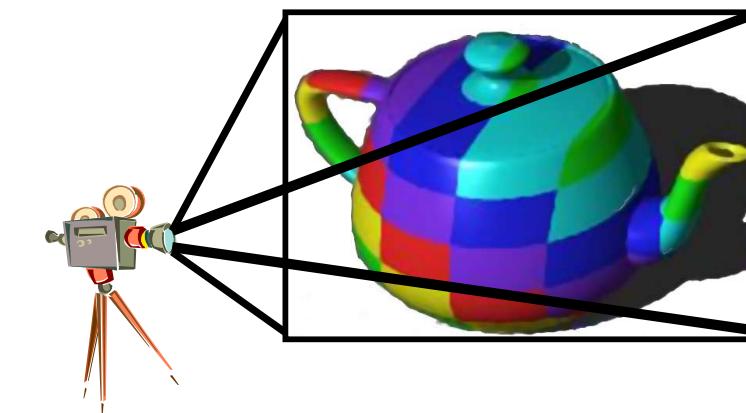
Object space (x_O, y_O, z_O)



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

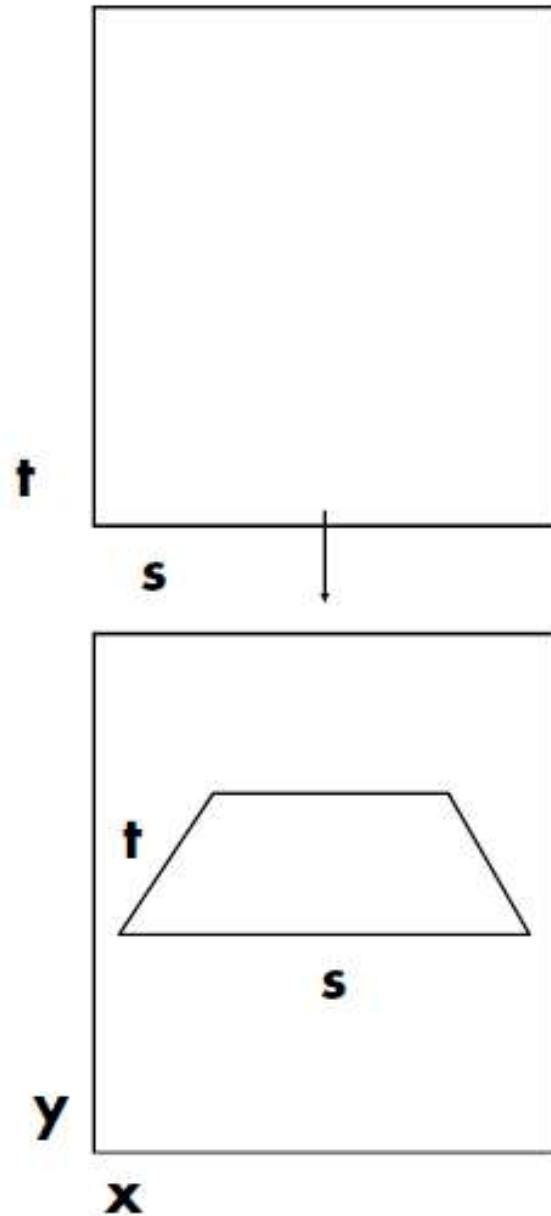
3D World Space

| Viewing Transformation

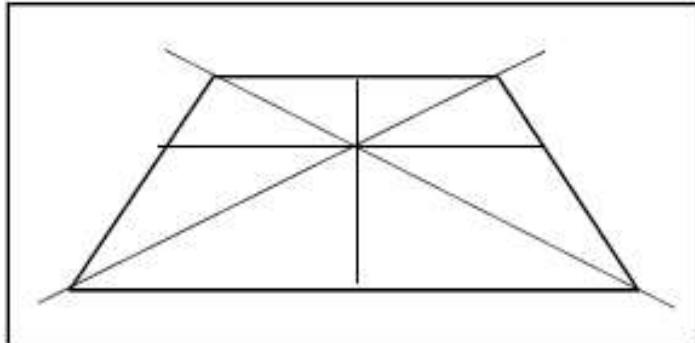
3D Camera Space

| Projection

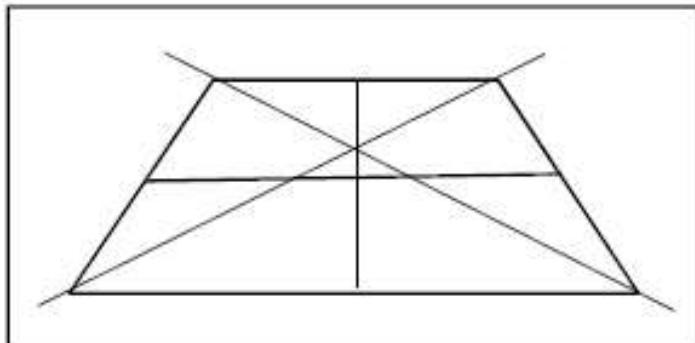
2D Image Space



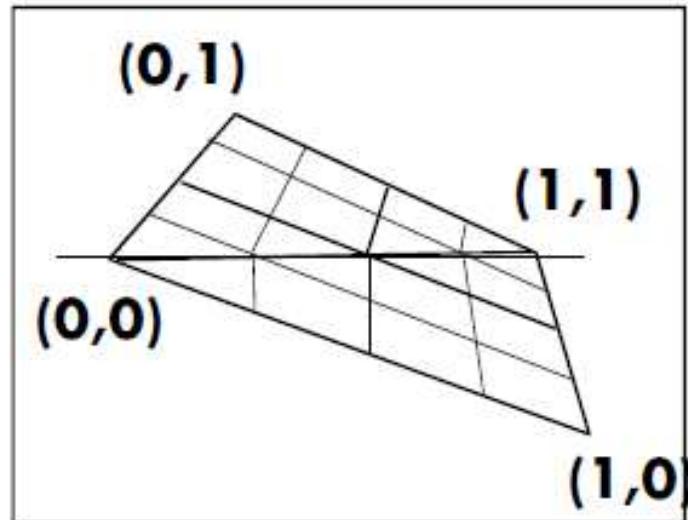
Linear Perspective



Correct Linear Perspective



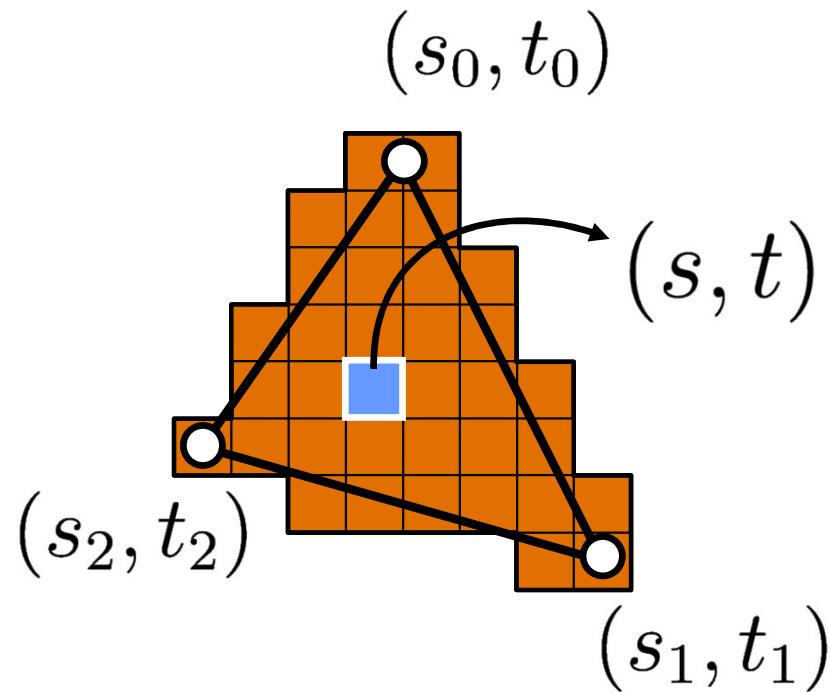
Incorrect Perspective



Linear Interpolation, Bad
Perspective Interpolation, Good

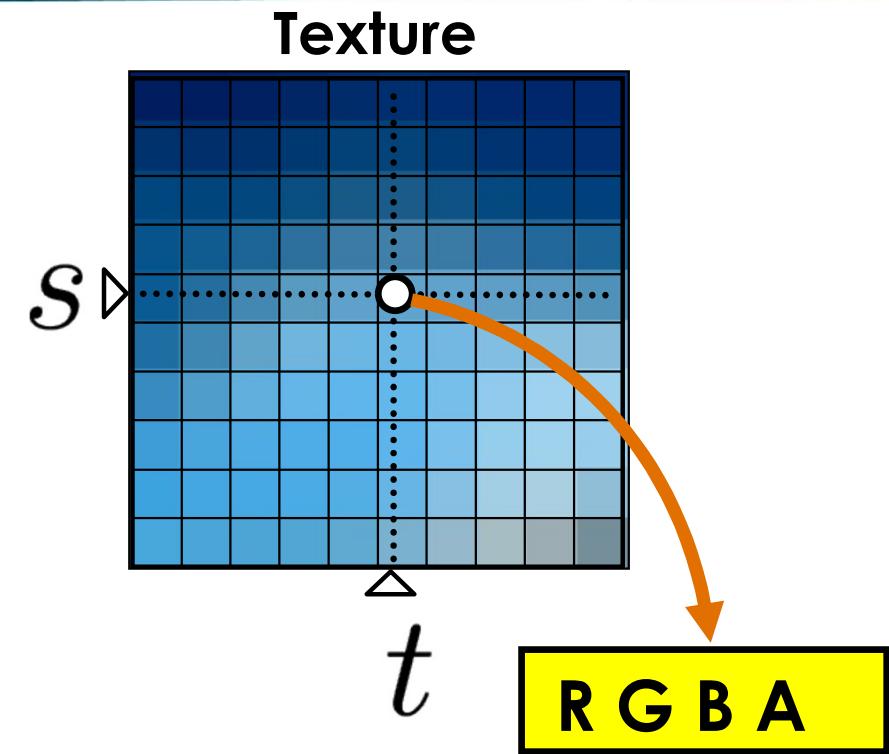


2D Texture Mapping



For each fragment:
interpolate the
texture coordinates
(barycentric)
Or:

Use arbitrary, computed coordinates

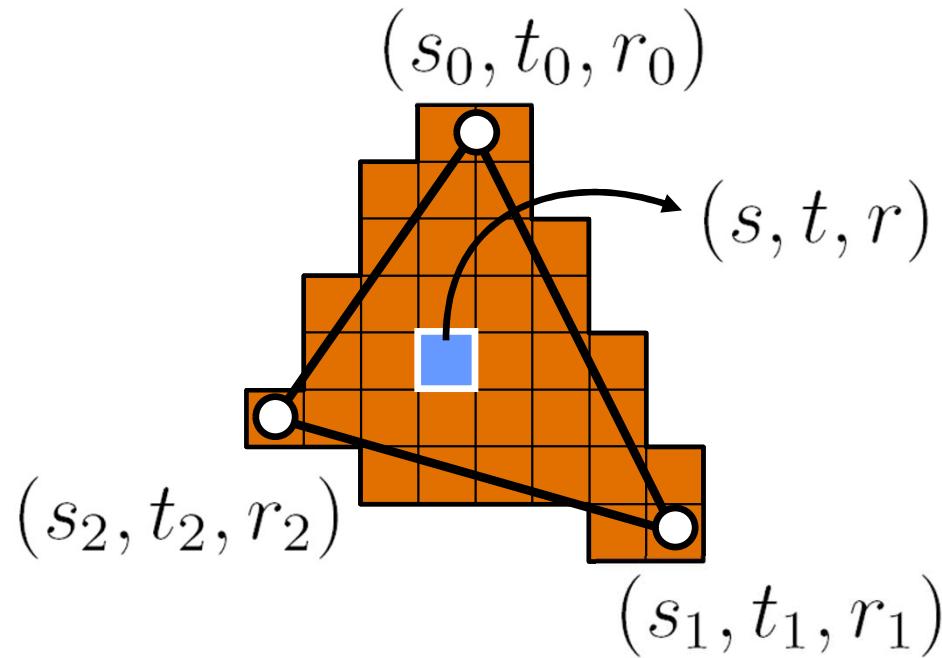


Texture-Lookup:
interpolate the
texture data
(bi-linear)
Or:

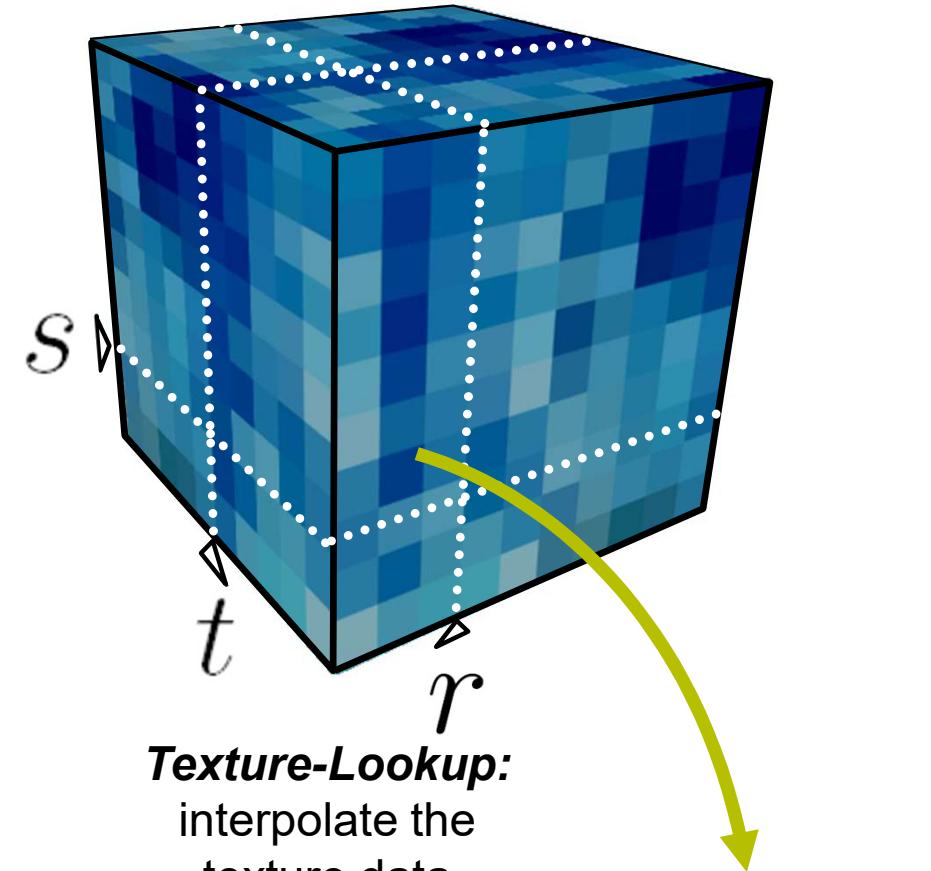
Nearest-neighbor for “array lookup”



3D Texture Mapping



For each fragment:
interpolate the
texture coordinates
(barycentric)
Or:
Use arbitrary, computed coordinates



Texture-Lookup:
interpolate the
texture data
(tri-linear)
Or:
Nearest-neighbor for “array lookup”

Interpolation #1



Interpolation Type + Purpose #1: **Interpolation of Texture Coordinates**

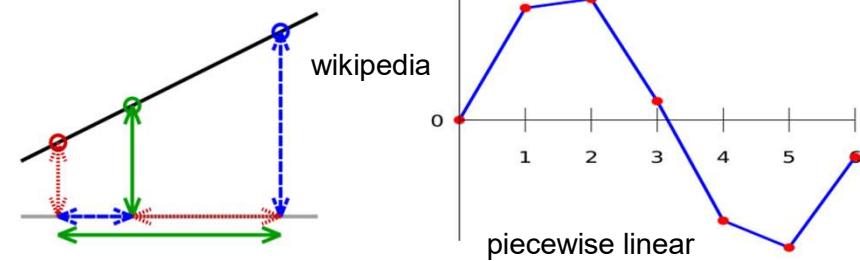
(Linear / Rational-Linear Interpolation)

Linear Interpolation / Convex Combinations



Linear interpolation in 1D:

$$f(\alpha) = (1 - \alpha)v_1 + \alpha v_2$$



Line embedded in 2D (linear interpolation of vertex coordinates/attributes):

$$f(\alpha_1, \alpha_2) = \alpha_1 v_1 + \alpha_2 v_2$$

$$\alpha_1 + \alpha_2 = 1$$

$$f(\alpha) = v_1 + \alpha(v_2 - v_1)$$

$$\alpha = \alpha_2$$

Line segment: $\alpha_1, \alpha_2 \geq 0$ (\rightarrow convex combination)

Compare to line parameterization
with parameter t:

$$v(t) = v_1 + t(v_2 - v_1)$$

Linear Interpolation / Convex Combinations

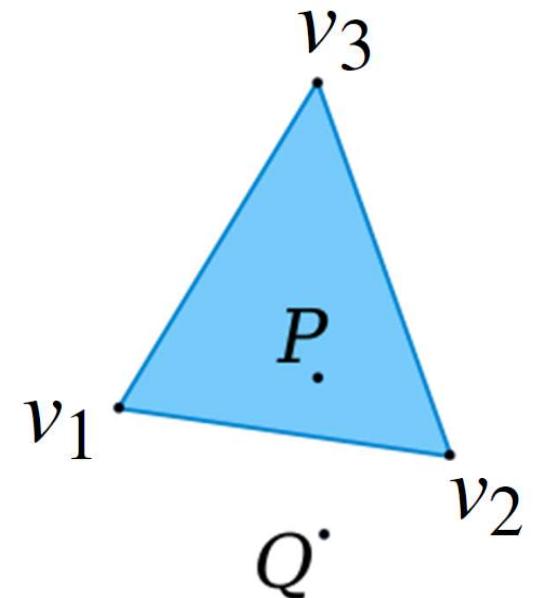


Linear combination (n -dim. space):

$$\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_n v_n = \sum_{i=1}^n \alpha_i v_i$$

Affine combination: Restrict to $(n - 1)$ -dim. subspace:

$$\alpha_1 + \alpha_2 + \dots + \alpha_n = \sum_{i=1}^n \alpha_i = 1$$



Convex combination: $\alpha_i \geq 0$

(restrict to simplex in subspace)

Linear Interpolation / Convex Combinations

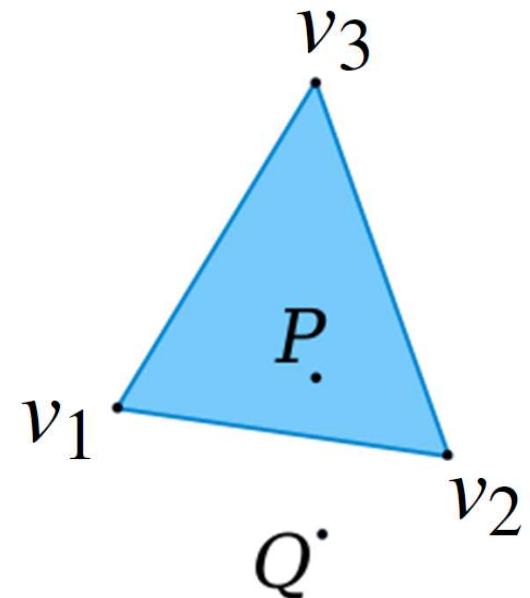


$$\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_n v_n = \sum_{i=1}^n \alpha_i v_i$$

$$\alpha_1 + \alpha_2 + \dots + \alpha_n = \sum_{i=1}^n \alpha_i = 1$$

Re-parameterize to get affine coordinates:

$$\begin{aligned}\alpha_1 v_1 + \alpha_2 v_2 + \alpha_3 v_3 &= \\ \tilde{\alpha}_1(v_2 - v_1) + \tilde{\alpha}_2(v_3 - v_1) + v_1 &\\ \tilde{\alpha}_1 &= \alpha_2 \\ \tilde{\alpha}_2 &= \alpha_3\end{aligned}$$



Linear Interpolation / Convex Combinations



The weights α_i are the (normalized) barycentric coordinates

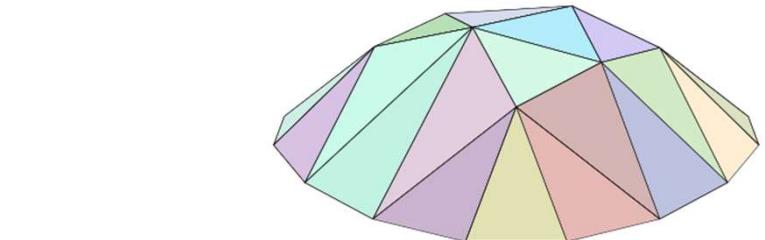
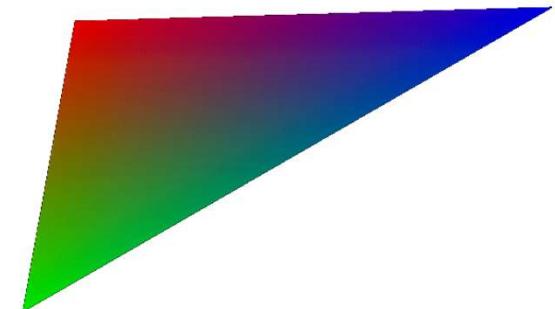
→ linear attribute interpolation in simplex

$$\alpha_1 v_1 + \alpha_2 v_2 + \dots + \alpha_n v_n = \sum_{i=1}^n \alpha_i v_i$$

$$\alpha_1 + \alpha_2 + \dots + \alpha_n = \sum_{i=1}^n \alpha_i = 1$$

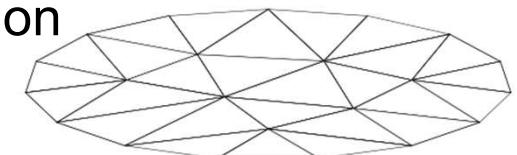
$$\alpha_i \geq 0$$

attribute interpolation



spatial position
interpolation

wikipedia



Homogeneous Coordinates (1)



Projective geometry

- (Real) projective spaces \mathbf{RP}^n :
Real projective line \mathbf{RP}^1 , real projective plane \mathbf{RP}^2 , ...
- A point in \mathbf{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 \mathbf{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 \mathbf{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 $(x, y, 0)$



Homogeneous Coordinates (2)

Examples of usage

- Translation (with translation vector \vec{b})
- Affine transformations (linear transformation + translation)

$$\vec{y} = A\vec{x} + \vec{b}.$$

- With homogeneous coordinates:

$$\left[\begin{array}{c} \vec{y} \\ 1 \end{array} \right] = \left[\begin{array}{cc|c} & A & \vec{b} \\ 0 & \dots & 0 \end{array} \right] \left[\begin{array}{c} \vec{x} \\ 1 \end{array} \right]$$

- Setting the last coordinate = 1 and the last row of the matrix to [0, ..., 0, 1] results in translation of the point \vec{x} (via addition of translation vector \vec{b})
- The matrix above is a linear map, but because it is one dimension higher, it does not have to move the origin in the $(n+1)$ -dimensional space for translation



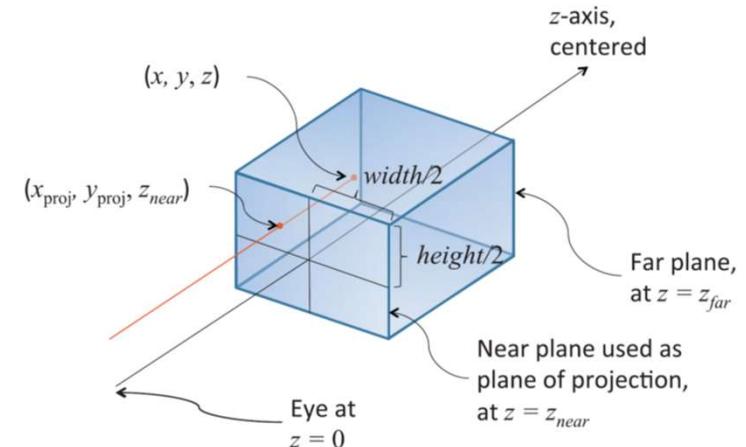
Homogeneous Coordinates (3)

Examples of usage

- Projection (e.g., OpenGL projection matrices)

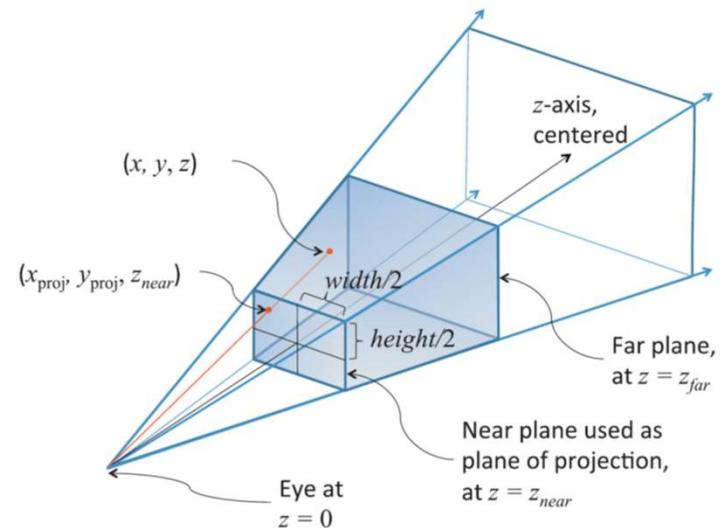
$$\begin{bmatrix} \frac{2}{right-left} & 0 & 0 & -\frac{right+left}{right-left} \\ 0 & \frac{2}{top-bottom} & 0 & -\frac{top+bottom}{top-bottom} \\ 0 & 0 & \frac{-2}{far-near} & -\frac{far+near}{far-near} \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

orthographic



$$\begin{bmatrix} \frac{z_{near}}{width/2} & 0.0 & \frac{left+right}{width/2} & 0.0 \\ 0.0 & \frac{z_{near}}{height/2} & \frac{top+bottom}{height/2} & 0.0 \\ 0.0 & 0.0 & -\frac{z_{far}+z_{near}}{z_{far}-z_{near}} & \frac{2z_{far}z_{near}}{z_{far}-z_{near}} \\ 0.0 & 0.0 & -1.0 & 0.0 \end{bmatrix}$$

perspective



Texture Mapping

2D (3D) Texture Space

| Texture Transformation

2D Object Parameters

| Parameterization

3D Object Space

| Model Transformation

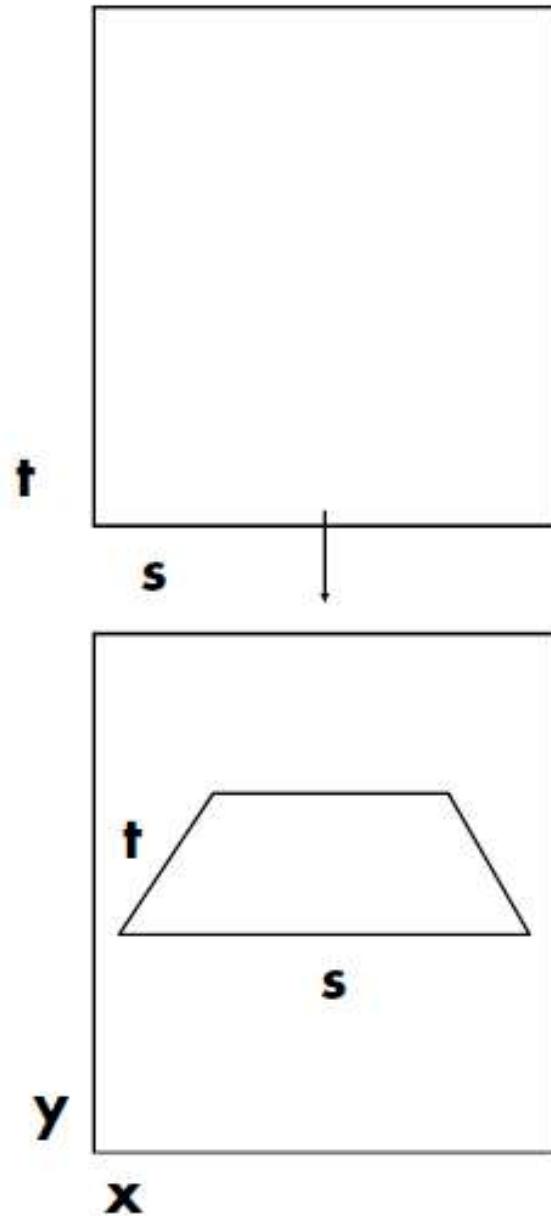
3D World Space

| Viewing Transformation

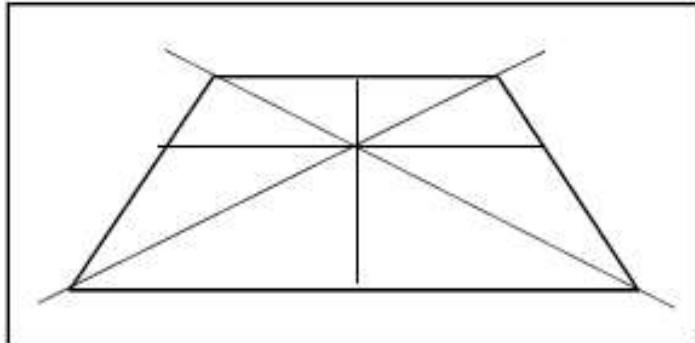
3D Camera Space

| Projection

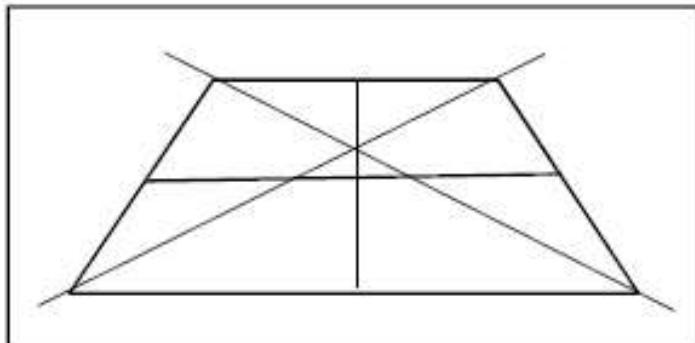
2D Image Space



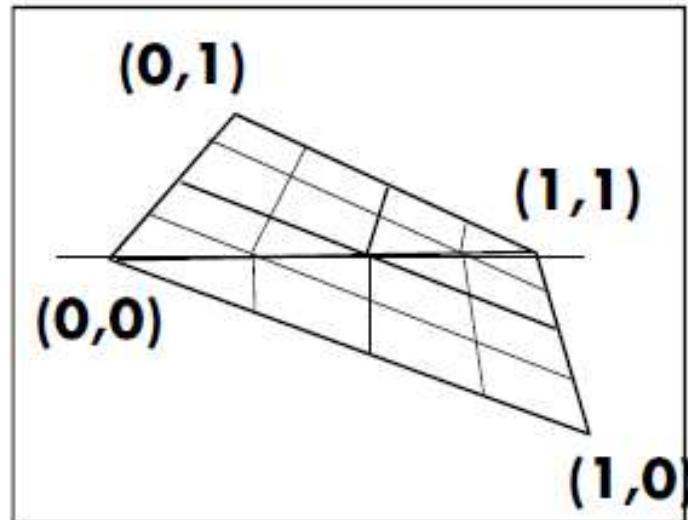
Linear Perspective



Correct Linear Perspective



Incorrect Perspective



Linear Interpolation, Bad
Perspective Interpolation, Good

Texture Mapping Polygons

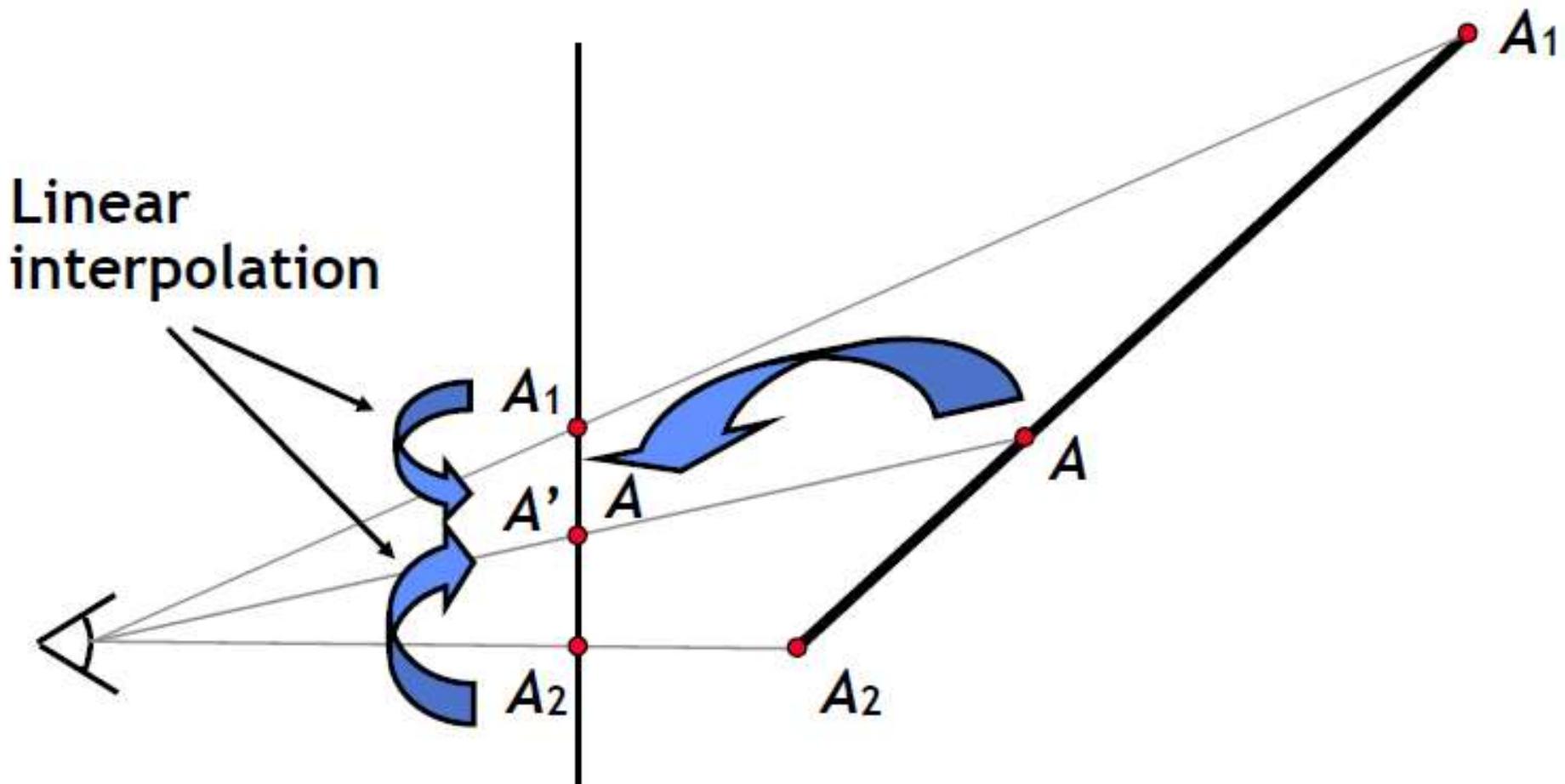
Forward transformation: linear projective map

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

Backward transformation: linear projective map

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

Incorrect attribute interpolation



$A' \neq A !$

Linear interpolation

Compute intermediate attribute value

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

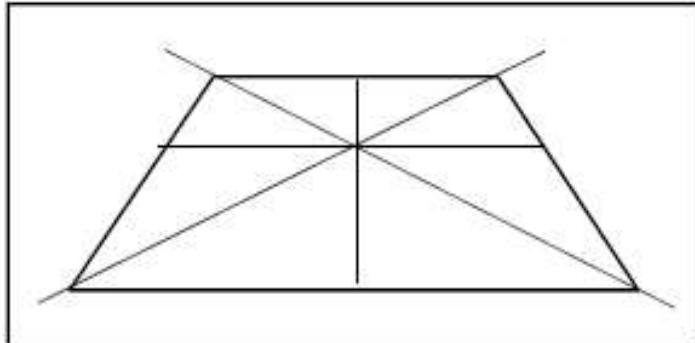
Only projected values interpolate linearly in screen space (straight lines project to straight lines)

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

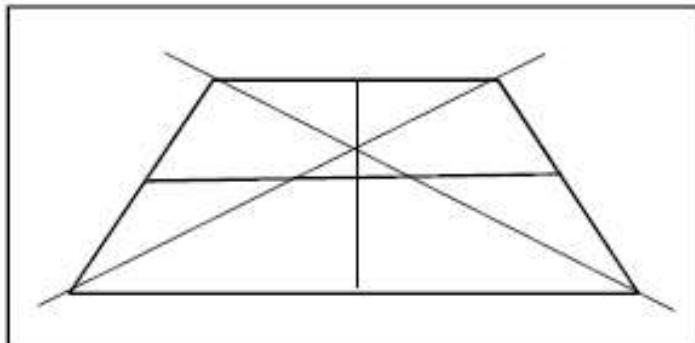
Choice for attribute interpolation in screen space

- 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

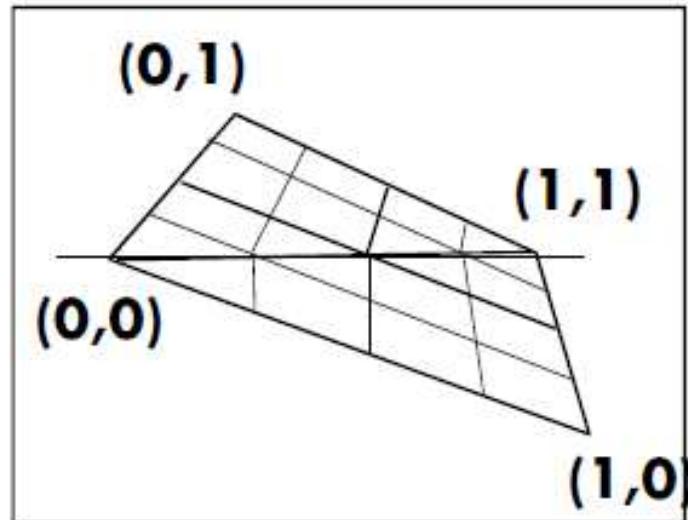
Linear Perspective



Correct Linear Perspective



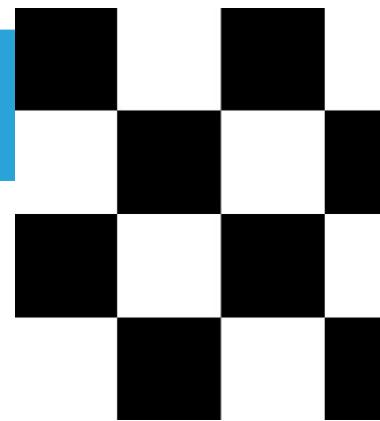
Incorrect Perspective



Linear Interpolation, Bad

Perspective Interpolation, Good

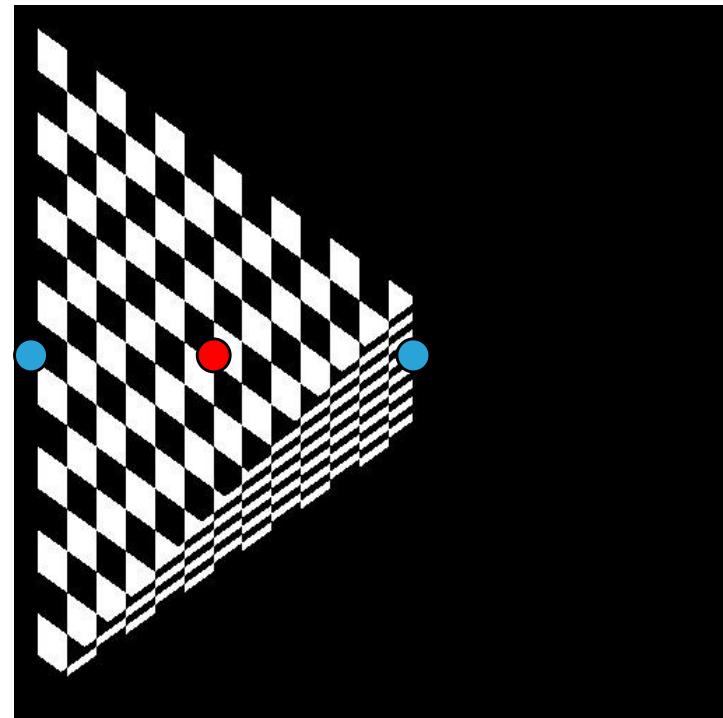
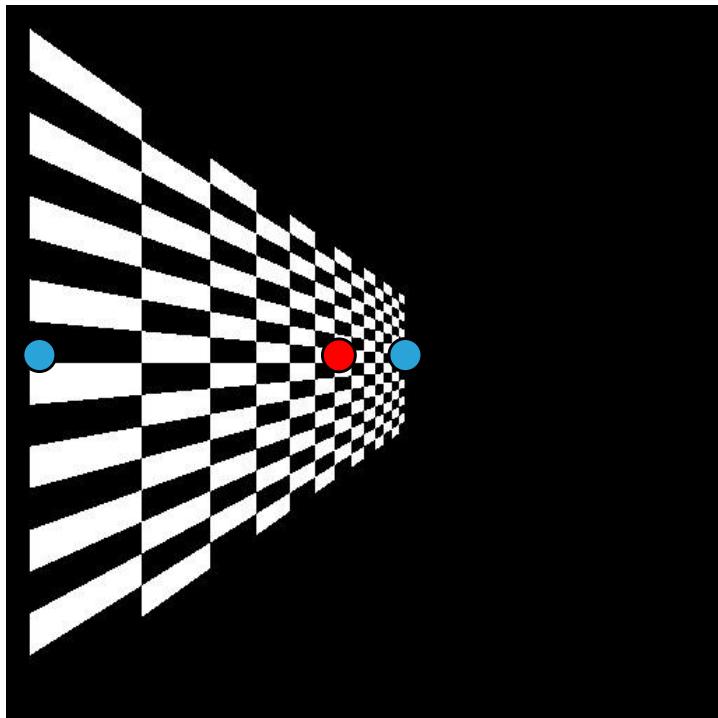
Perspective Texture Mapping



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



$$a = b = 0.5$$



Early Perspective Texture Mapping in Games



Ultima Underworld (Looking Glass, 1992)

Early Perspective Texture Mapping in Games



DOOM (id Software, 1993)

Early Perspective Texture Mapping in Games



Quake (id Software, 1996)

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

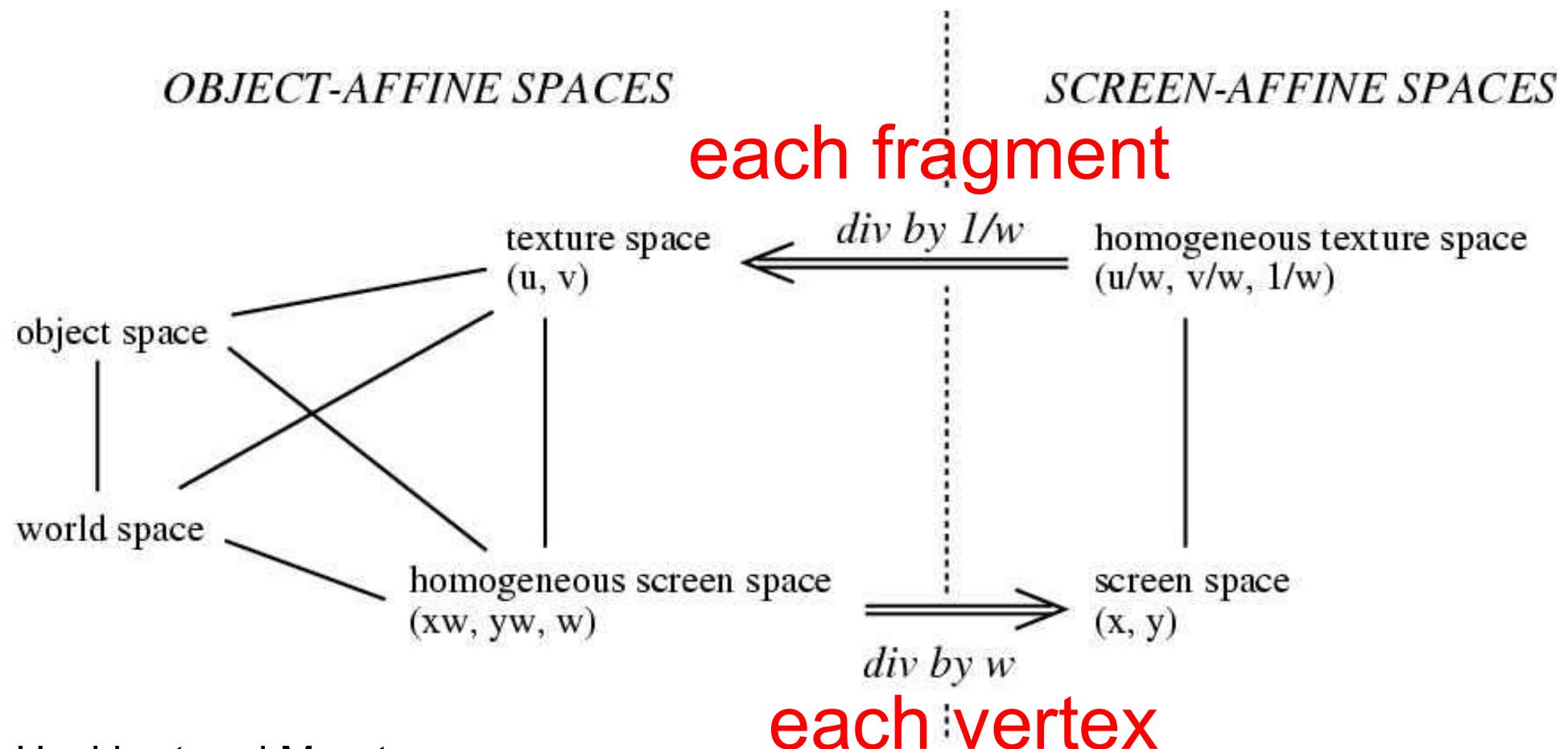
- $(A/w) / (1/w) = A$
- Division is expensive (more than add or multiply), so
 - Recover w for the sample point (reciprocate), and
 - Multiply each projected attribute by w

Barycentric triangle parameterization:

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

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

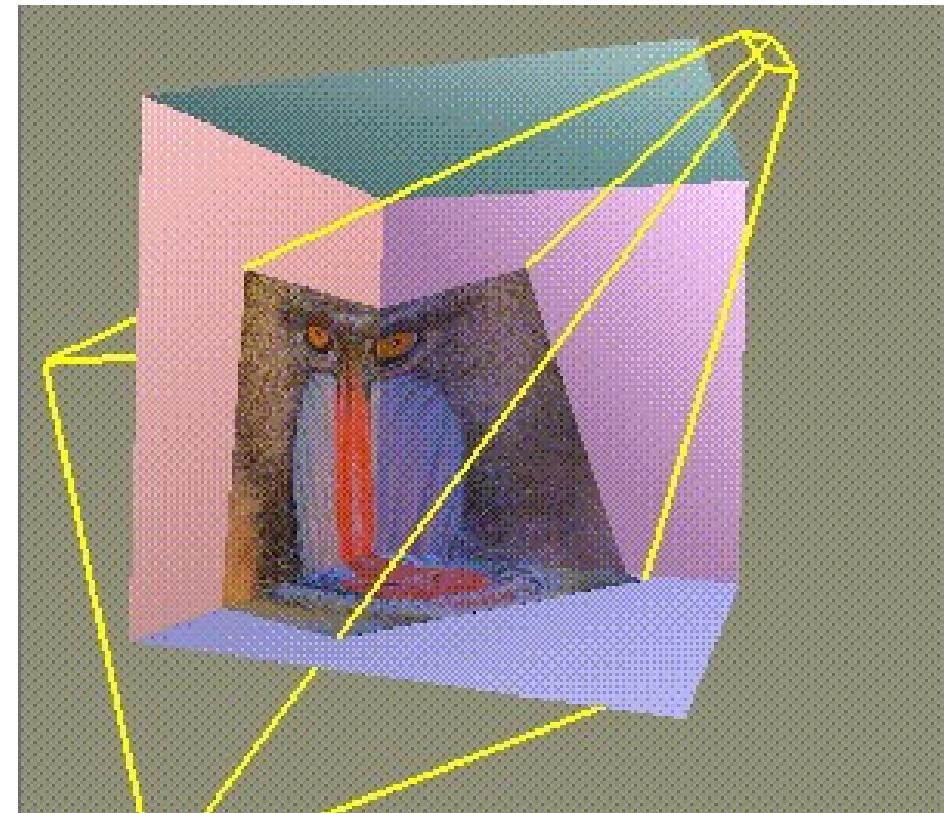


$$r_i(x, y) = \frac{r_i(x, y)/w(x, y)}{1/w(x, y)}$$

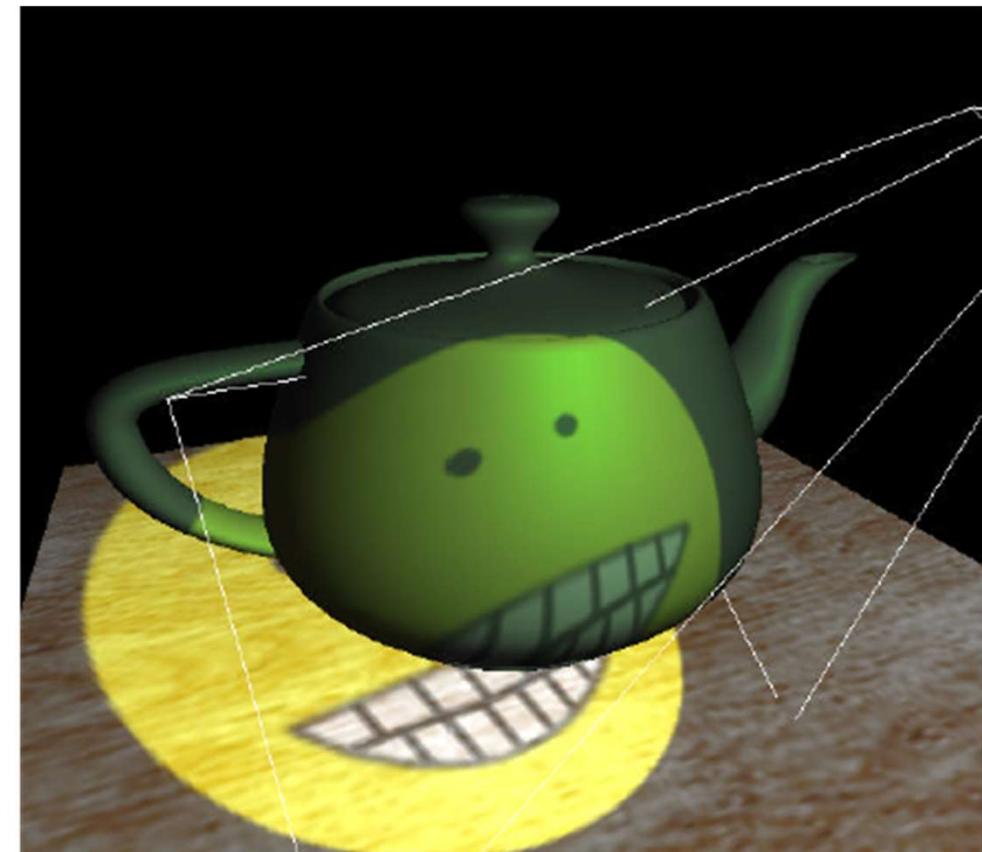
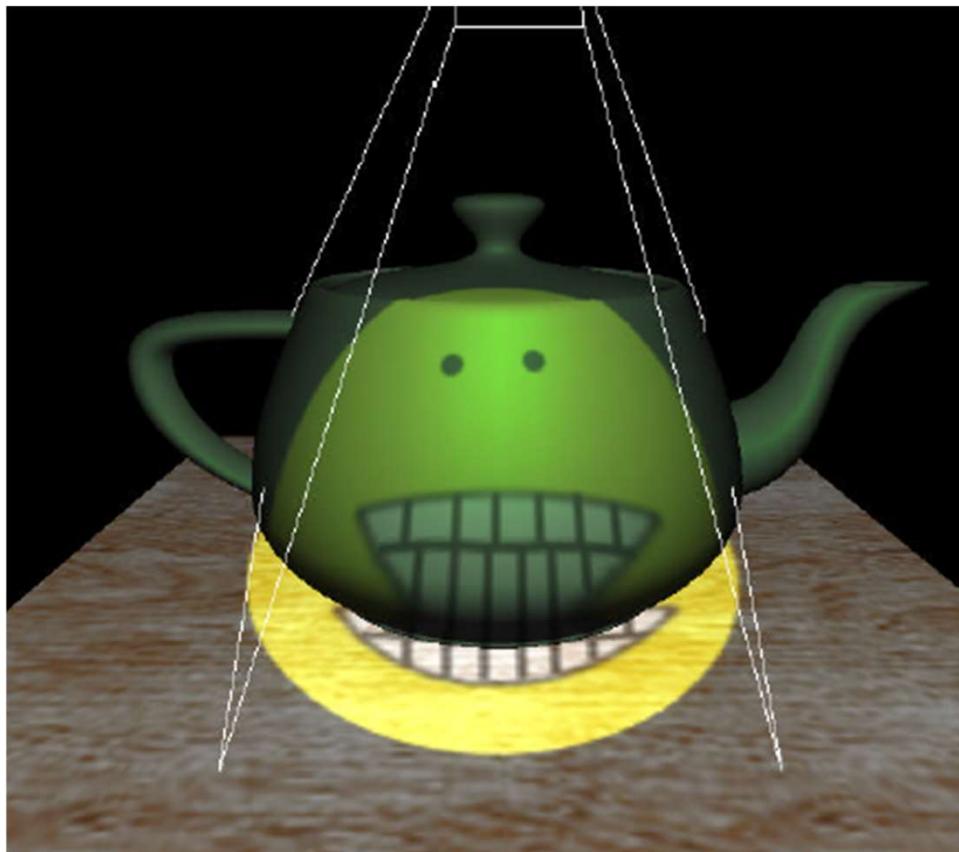
- (1) Associate a record containing the n parameters of interest (r_1, r_2, \dots, r_n) with each vertex of the polygon.
- (2) For each vertex, transform object space coordinates to homogeneous screen space using 4×4 object to screen matrix, yielding the values (xw, yw, zw, w) .
- (3) Clip the polygon 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 $(x, y, z, s_1, s_2, \dots, s_{n+1})$, 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.

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



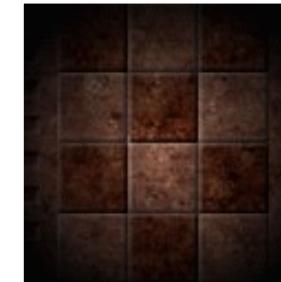
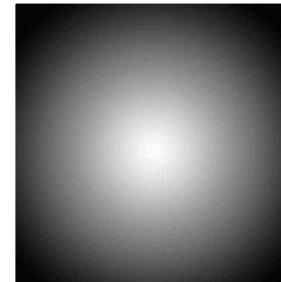
Projective Shadows in Doom 3



- What about **homogeneous texture coords**?
- Need to do perspective divide also for projector!
 - $(s, t, q) \rightarrow (s/q, t/q)$ 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! $\rightarrow r/q = \text{projector depth!}$



- 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

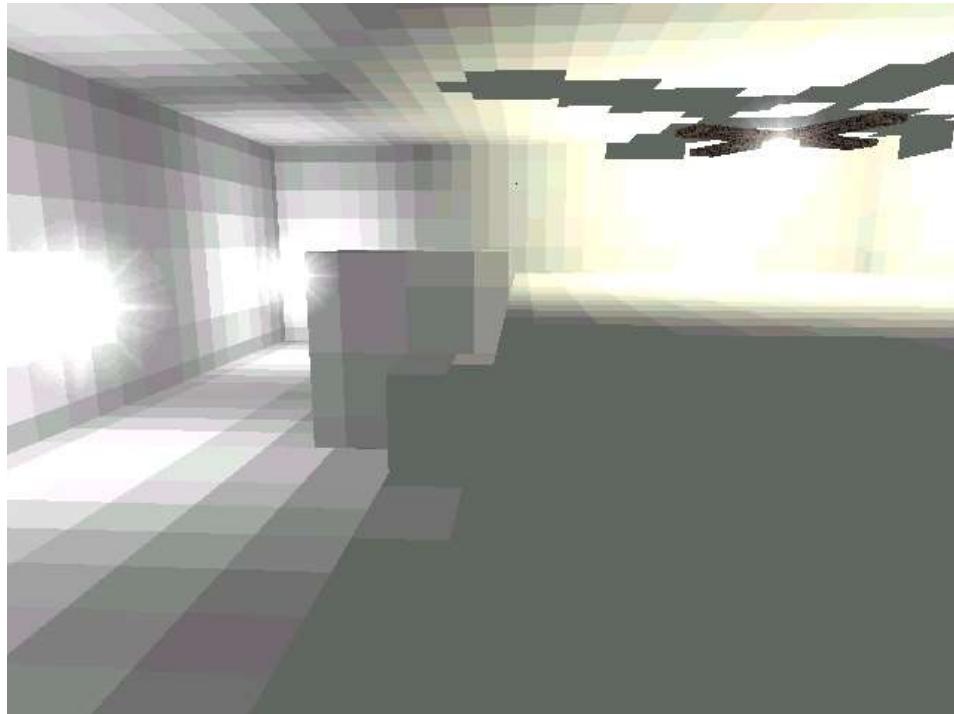


Example: Light Mapping

- 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



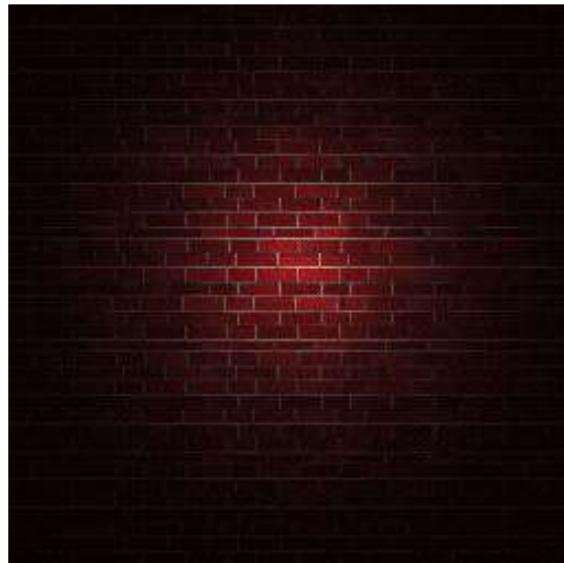
Original LM texels



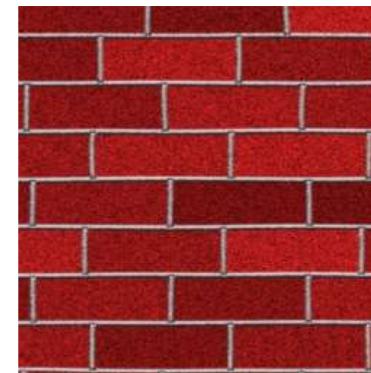
Bilinear Filtering



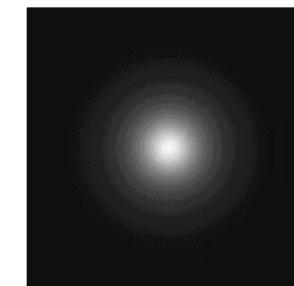
■ Why premultiplication is bad...



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

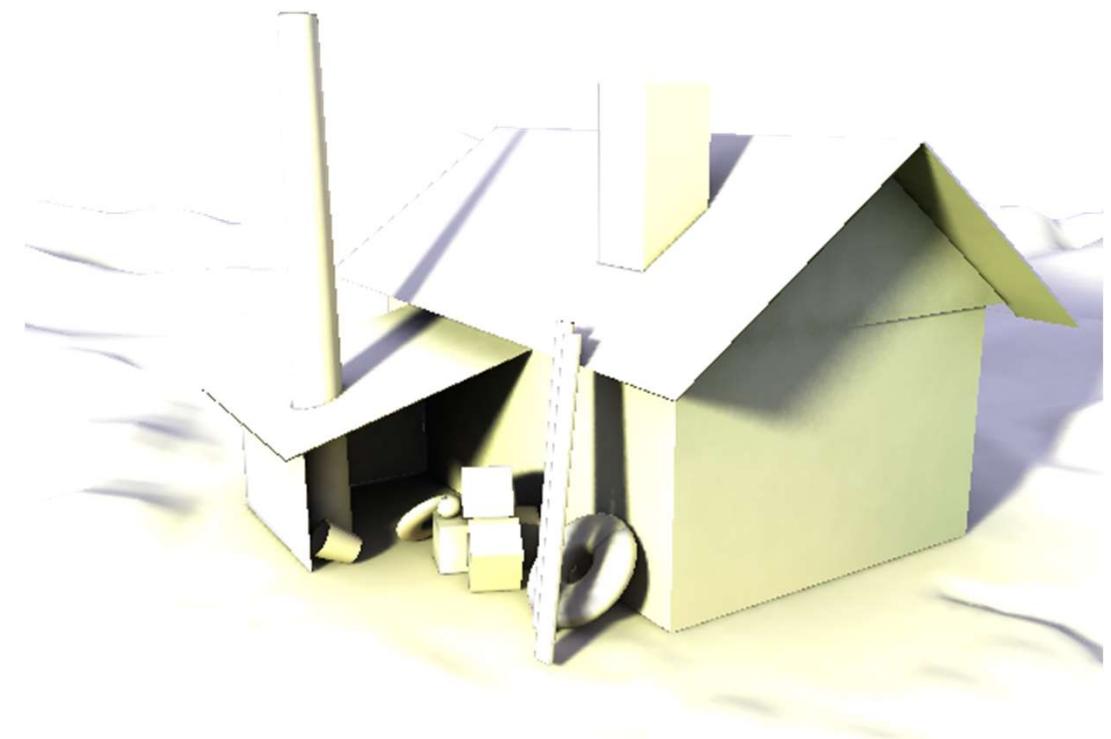
- Precomputation based on non-realtimemethods
 - Radiosity
 - Ray tracing
 - Monte Carlo Integration
 - Path tracing
 - Photon mapping



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, ...
- 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



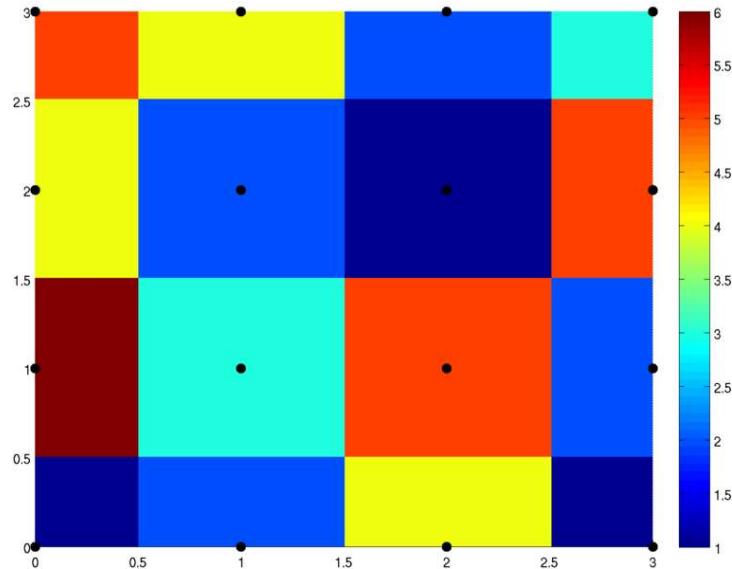
Nearest neighbor



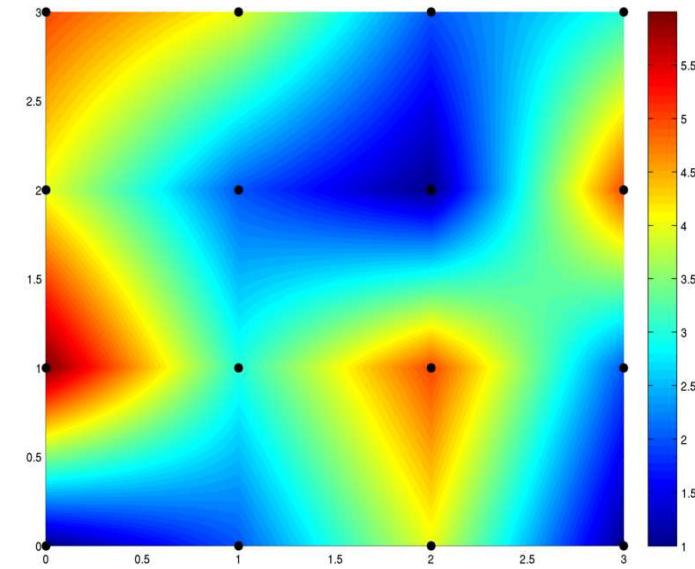
Bi-linear filtering



Nearest-Neighbor vs. Bi-Linear Interpolation

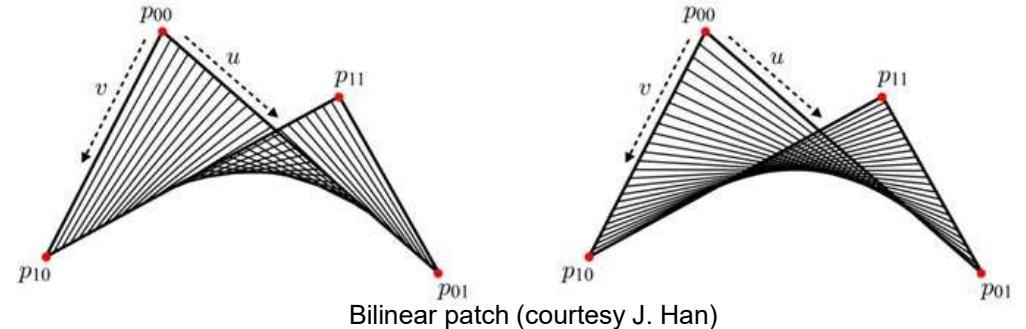


nearest-neighbor



wikipedia

bi-linear

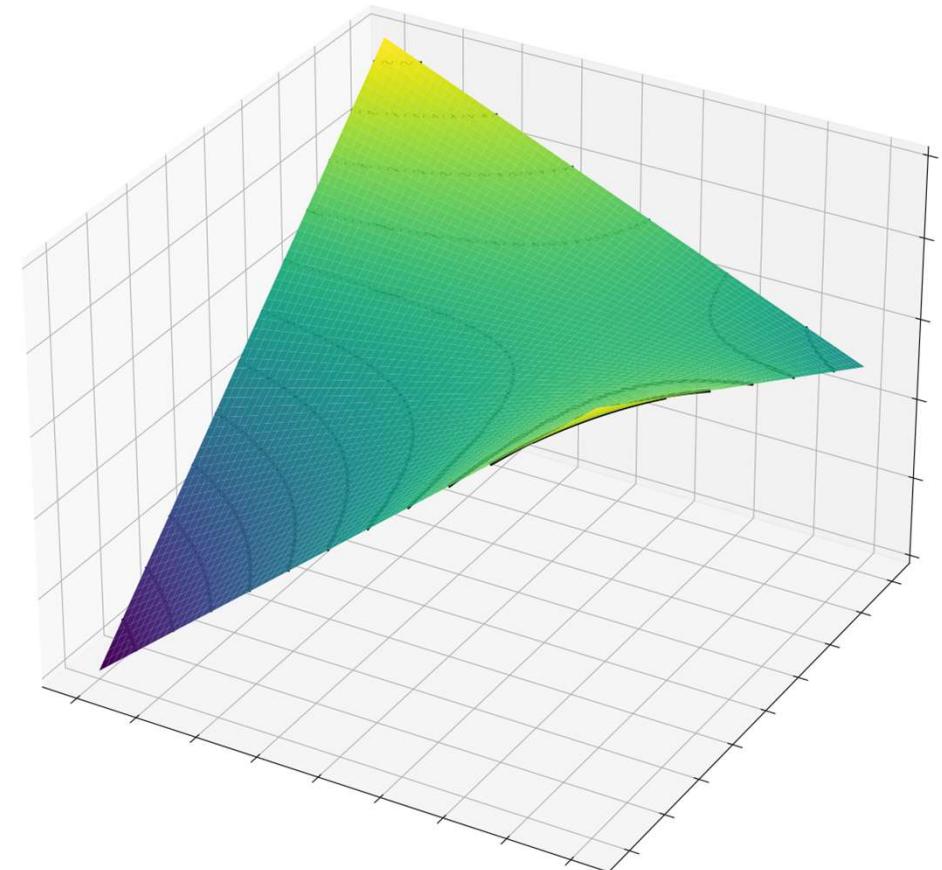
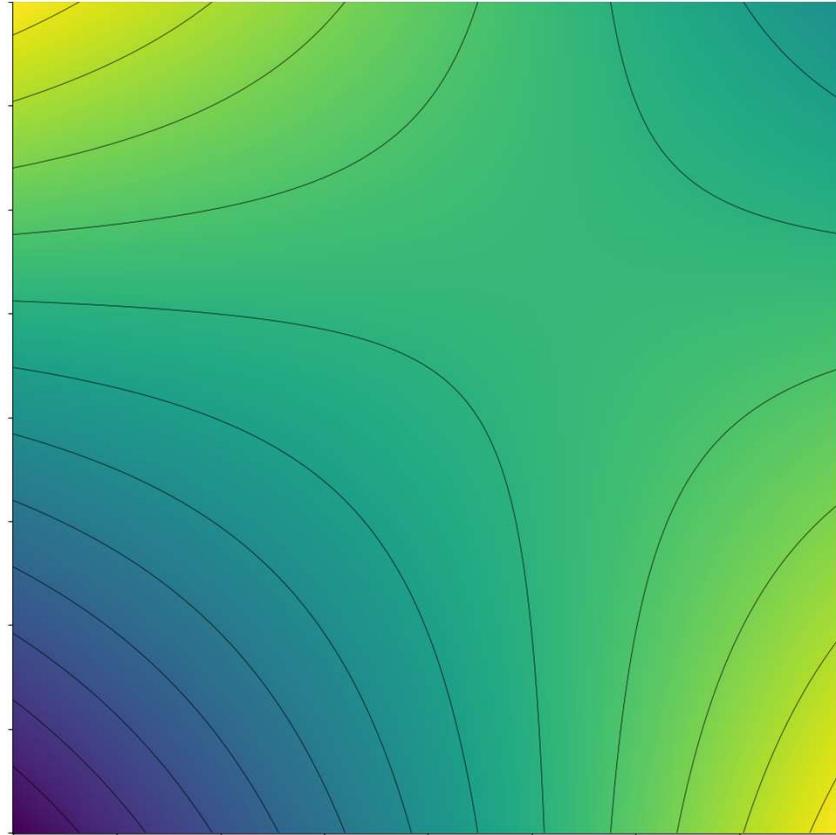




Bi-Linear Interpolation

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

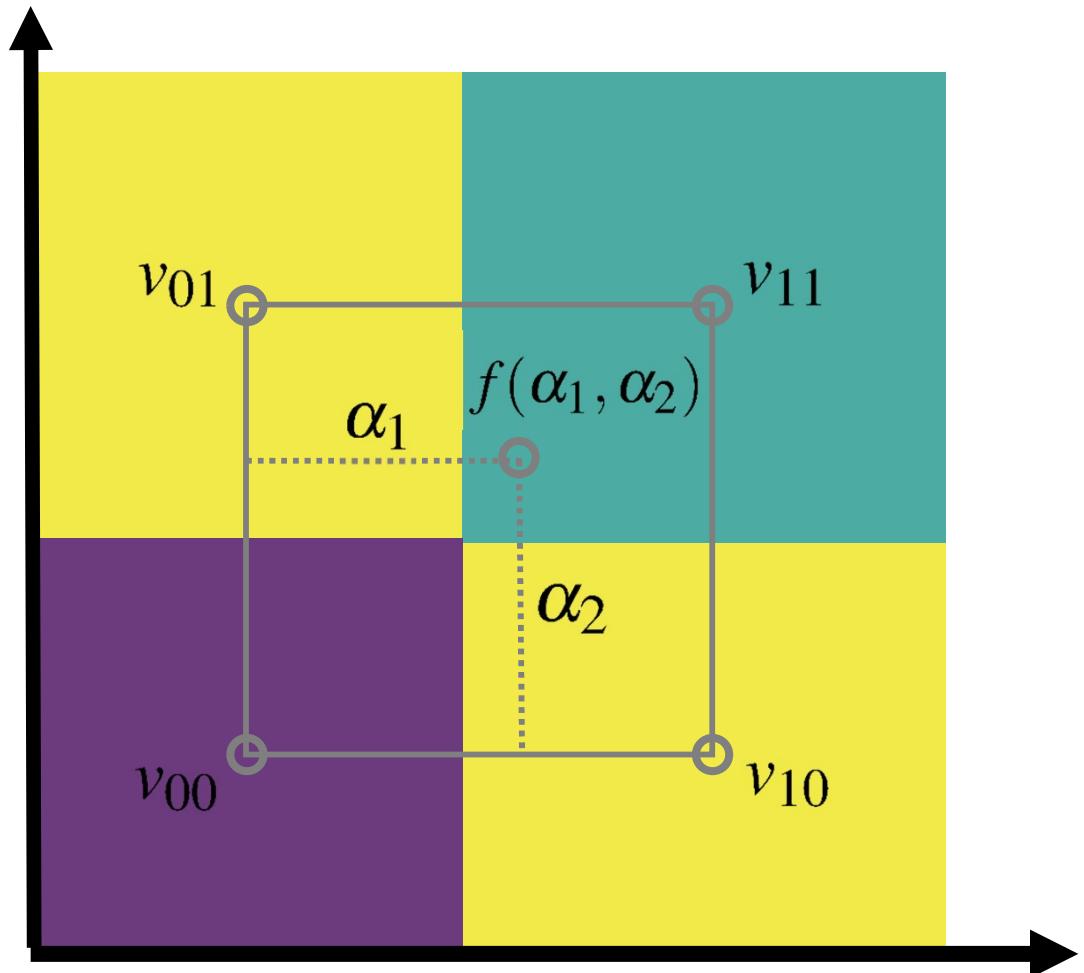
$$\alpha_1 := x_1 - \lfloor x_1 \rfloor \quad \alpha_1 \in [0.0, 1.0)$$

$$\alpha_2 := x_2 - \lfloor x_2 \rfloor \quad \alpha_2 \in [0.0, 1.0)$$

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

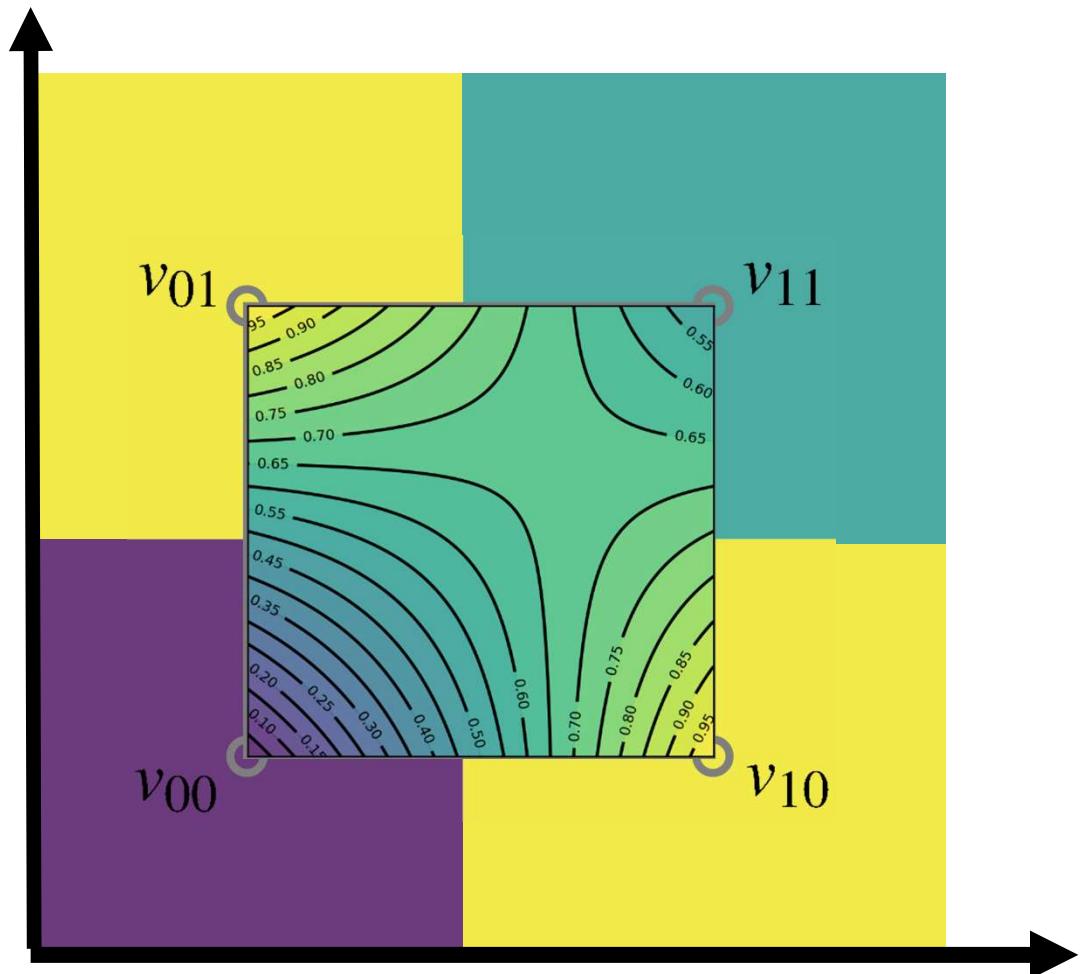
$$\alpha_1 := x_1 - \lfloor x_1 \rfloor \quad \alpha_1 \in [0.0, 1.0)$$

$$\alpha_2 := x_2 - \lfloor x_2 \rfloor \quad \alpha_2 \in [0.0, 1.0)$$

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) :

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



Bi-Linear Interpolation

Interpolate function at (fractional) position (α_1, α_2) :

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

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

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



Bi-Linear Interpolation

Interpolate function at (fractional) position (α_1, α_2) :

$$\begin{aligned} f(\alpha_1, \alpha_2) &= [\alpha_2 \quad (1 - \alpha_2)] \begin{bmatrix} v_{01} & v_{11} \\ v_{00} & v_{10} \end{bmatrix} \begin{bmatrix} (1 - \alpha_1) \\ \alpha_1 \end{bmatrix} \\ &= (1 - \alpha_1)(1 - \alpha_2)v_{00} + \alpha_1(1 - \alpha_2)v_{10} + (1 - \alpha_1)\alpha_2v_{01} + \alpha_1\alpha_2v_{11} \\ &= v_{00} + \alpha_1(v_{10} - v_{00}) + \alpha_2(v_{01} - v_{00}) + \alpha_1\alpha_2(v_{00} + v_{11} - v_{10} - v_{01}) \end{aligned}$$



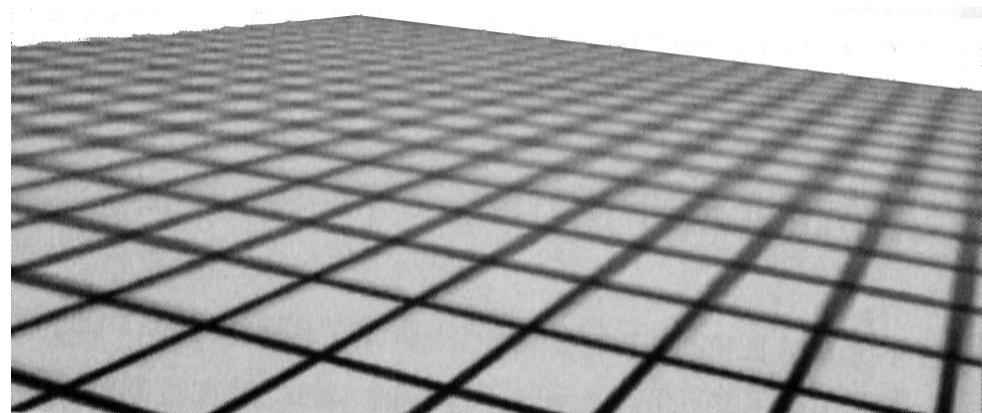
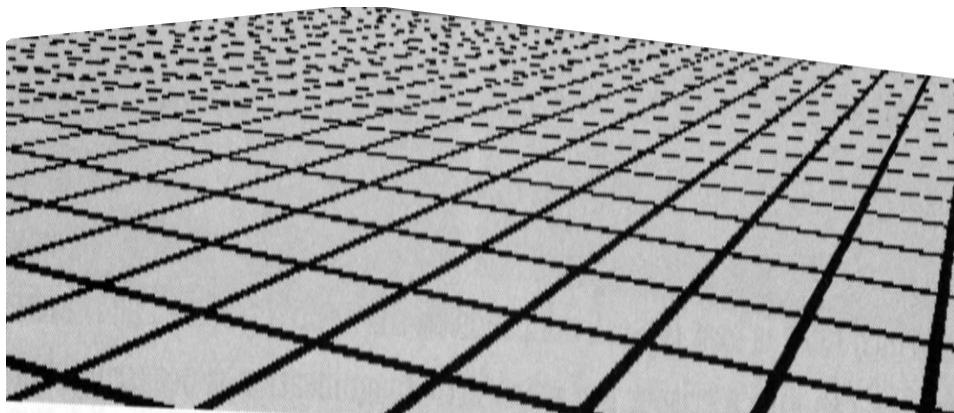
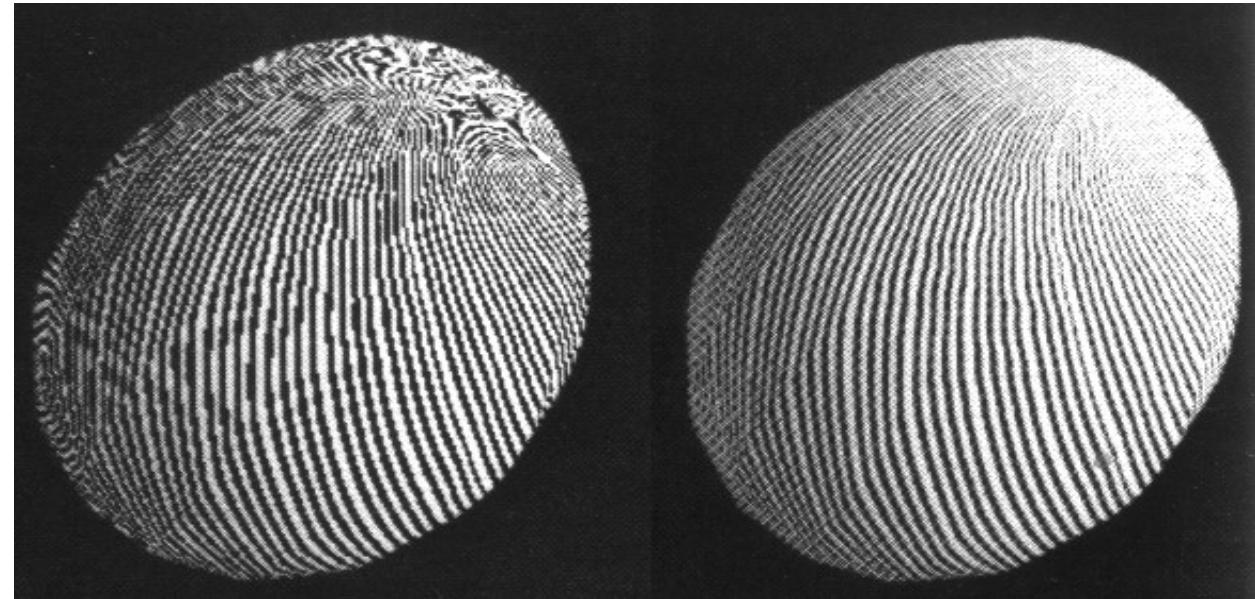
REALLY IMPORTANT:

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

Texture Minification

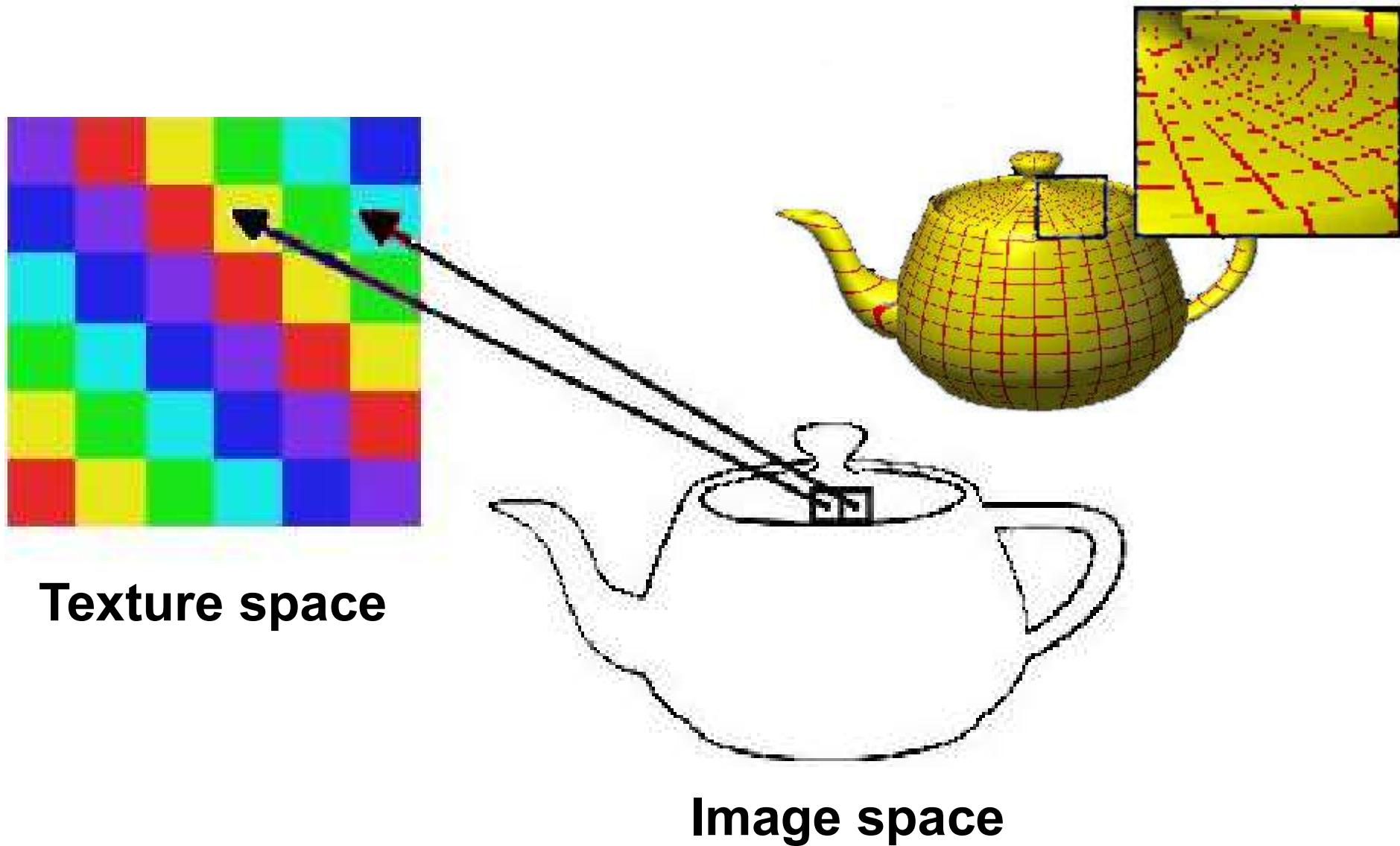
Texture Aliasing: Minification

- Problem: One pixel in image space covers many texels



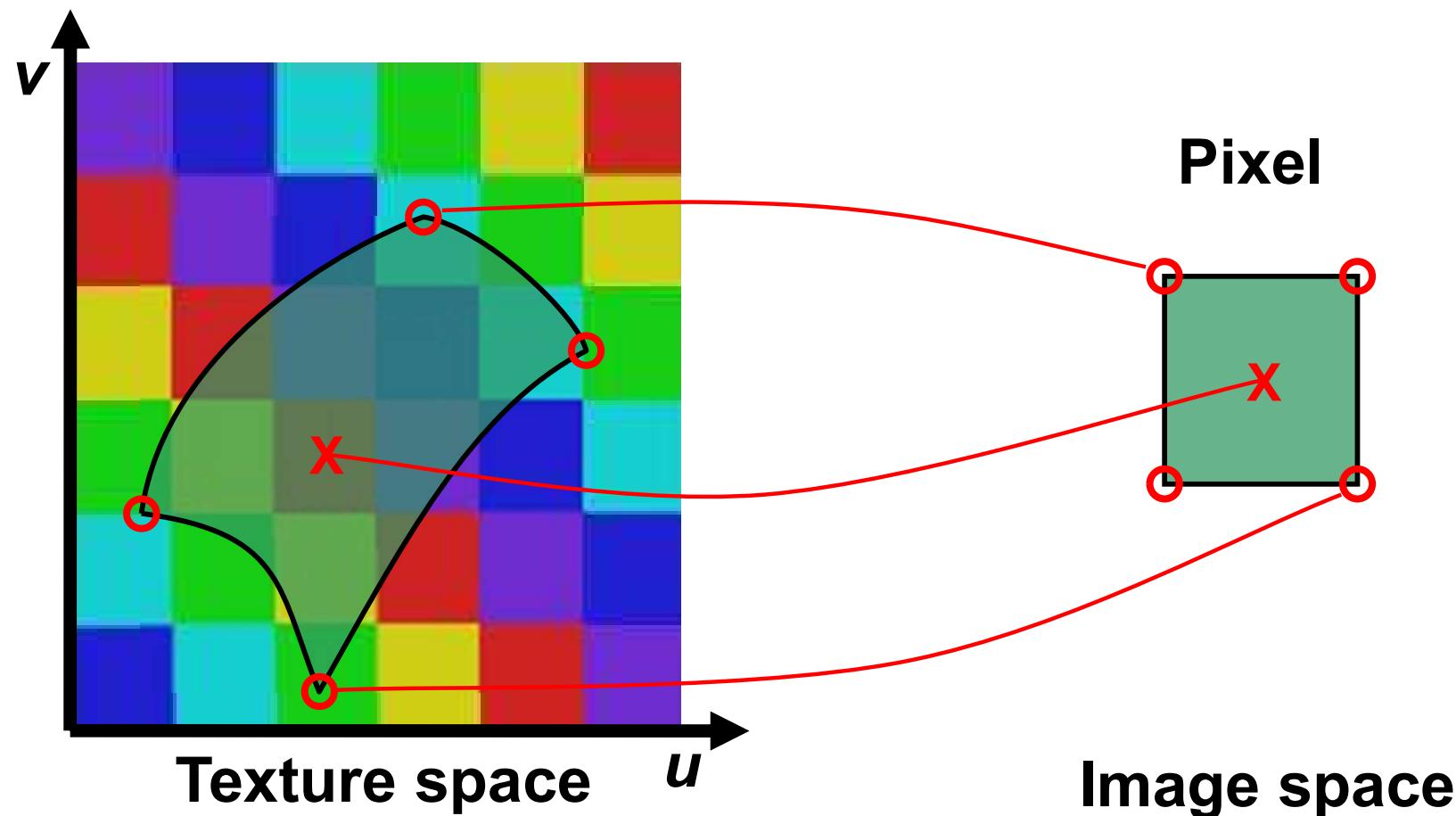
Texture Aliasing: Minification

- Caused by *undersampling*: texture information is lost



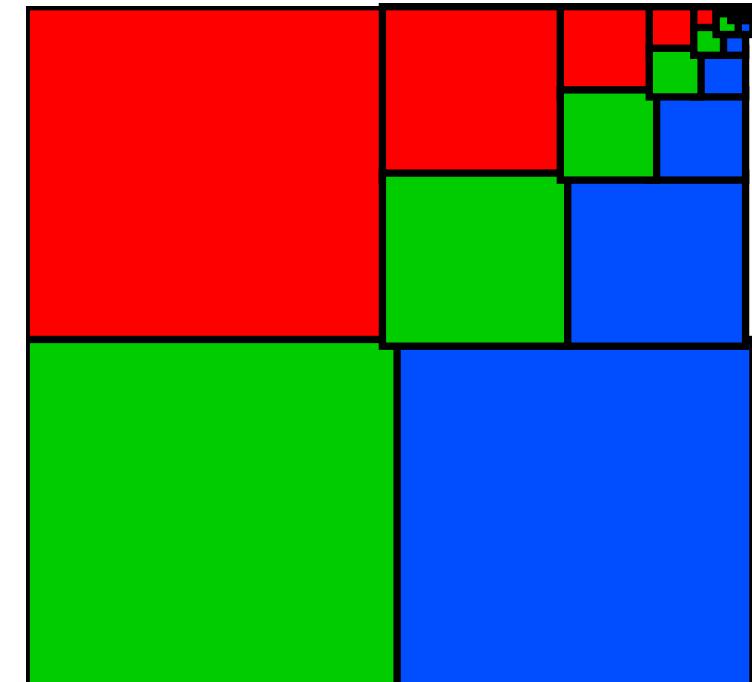
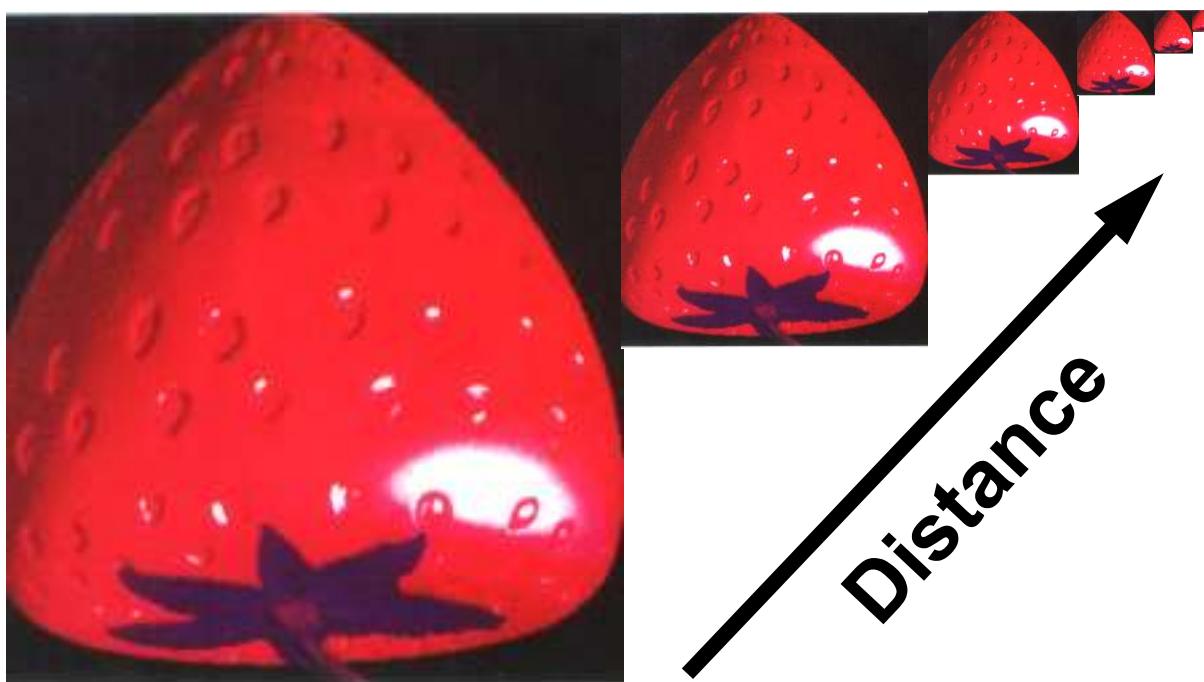
Texture Anti-Aliasing: Minification

- 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



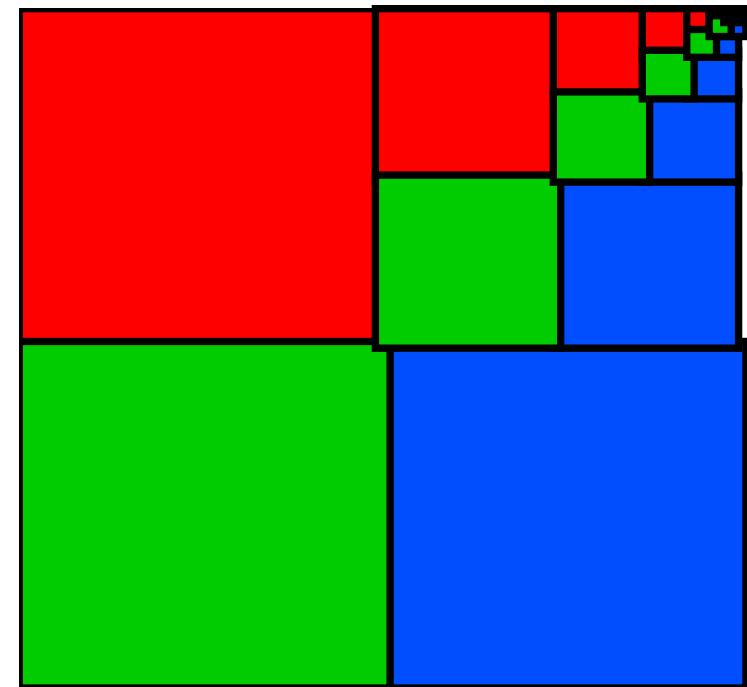
Texture Anti-Aliasing: MIP Mapping

- 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:

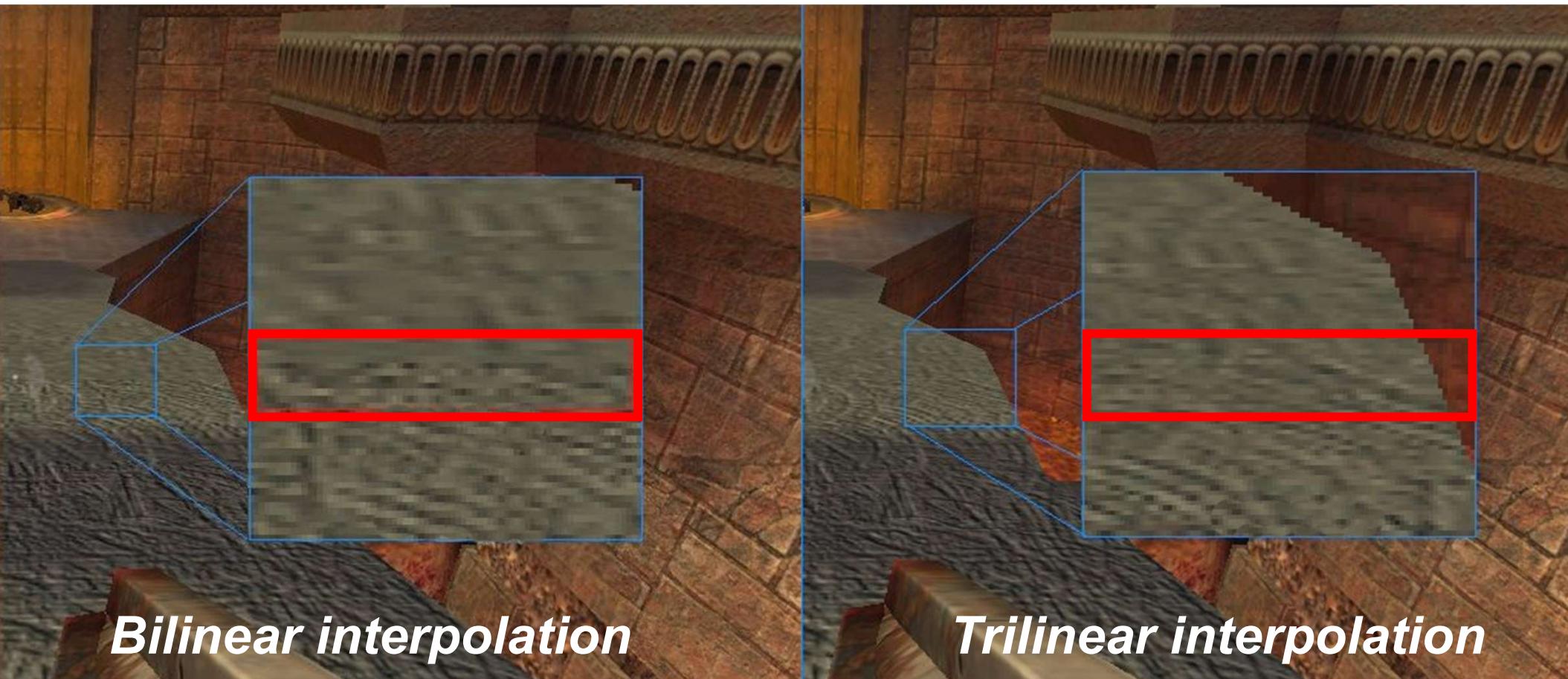
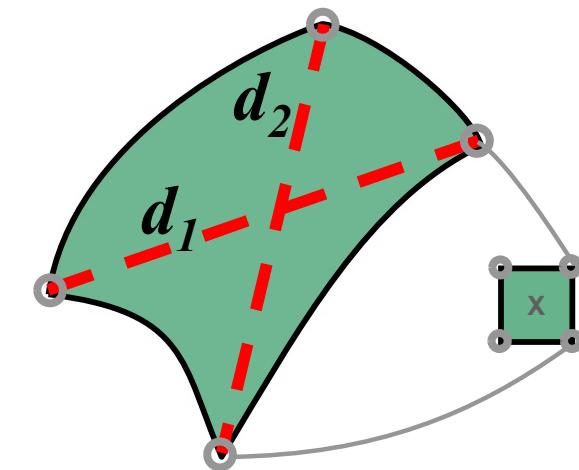
$$a + ar + ar^2 + ar^3 + \cdots + ar^{n-1} =$$

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



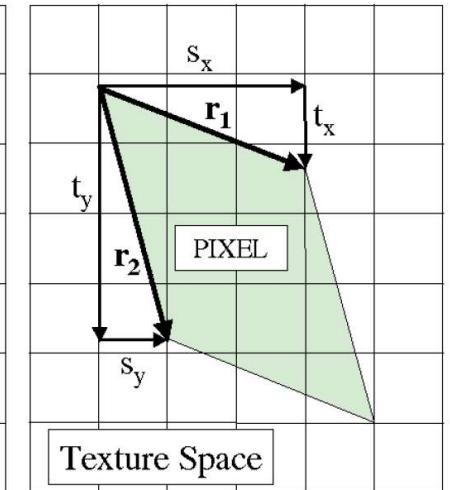
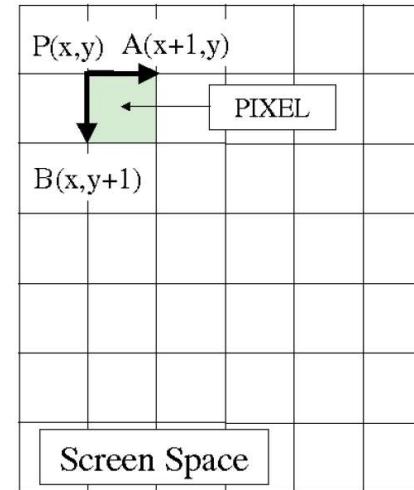
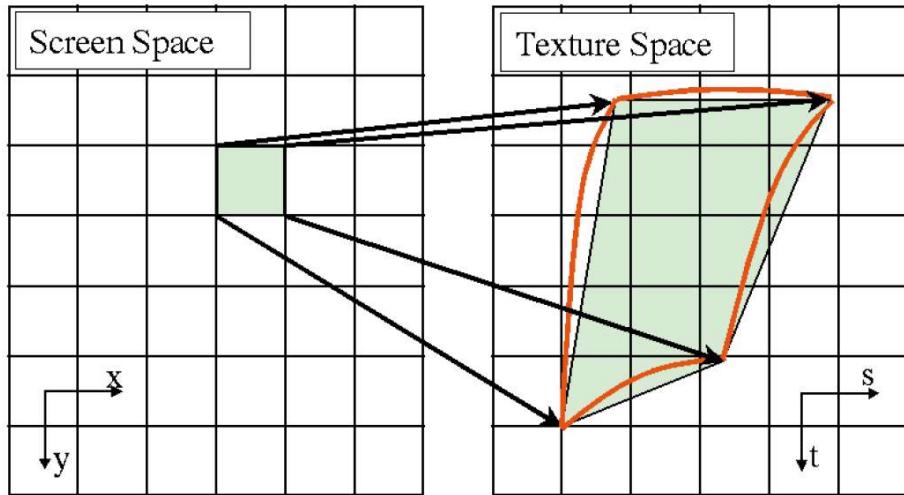
Texture Anti-Aliasing: MIP Mapping

- MIP Mapping Algorithm
- $D := ld(\max(d_1, d_2))$ "Mip Map level"
- $T_0 := \text{value from texture } D_0 = \text{trunc}(D)$
 - 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
- Area of parallelogram is the absolute value of the Jacobian determinant (the *Jacobian*)

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



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]

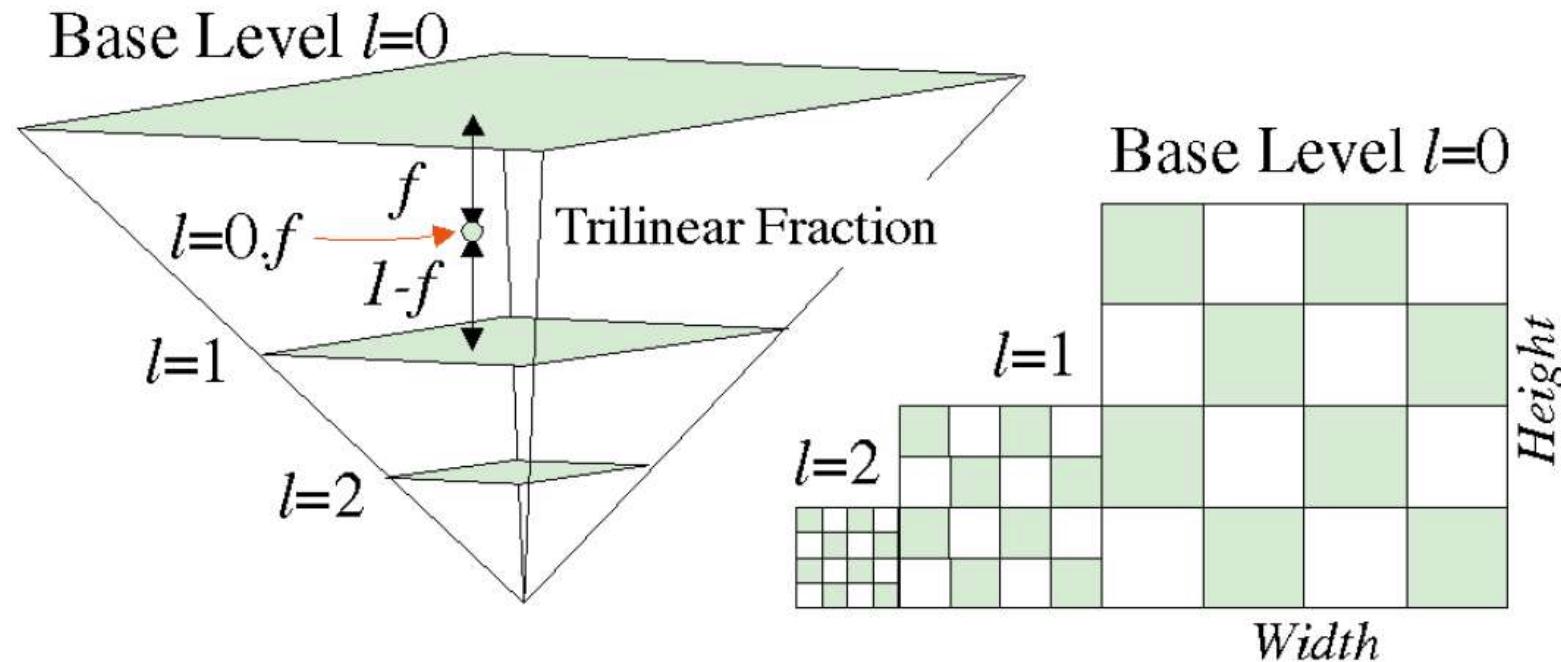
- Approximation without square-roots

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

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



MIP-Map Level Interpolation

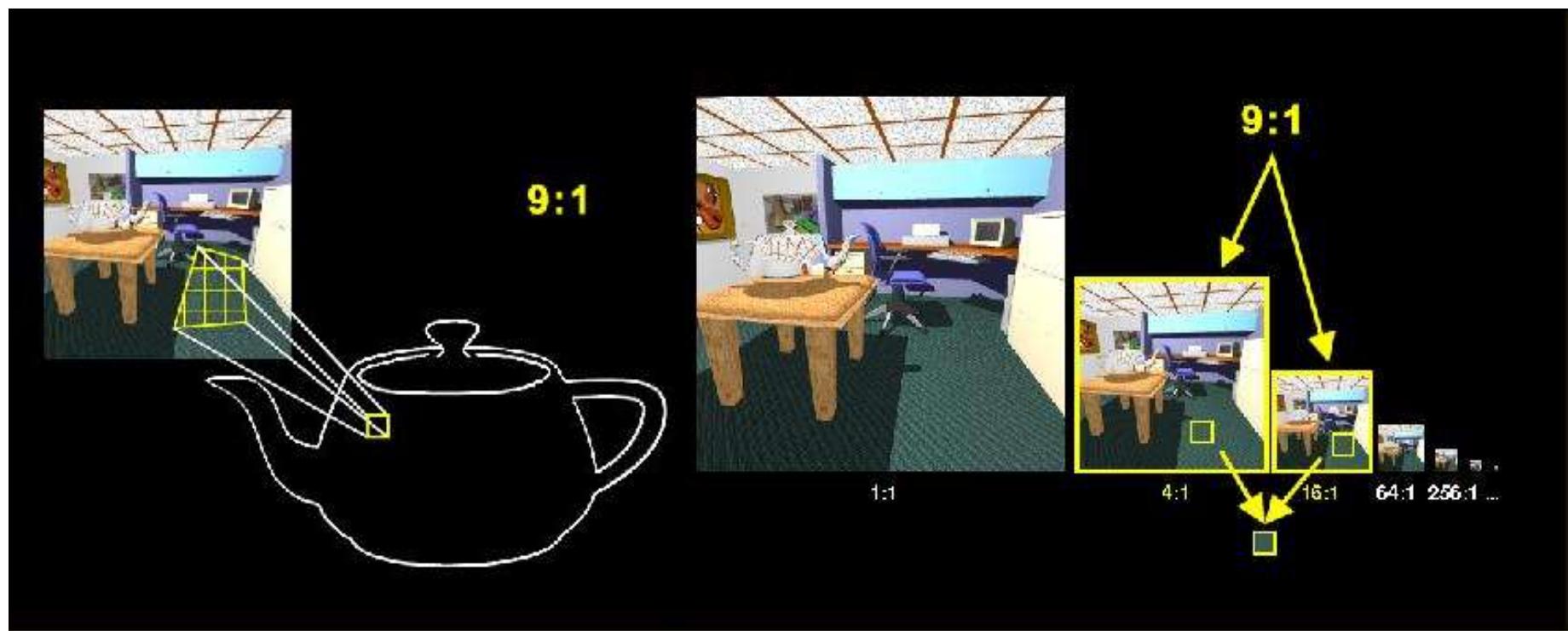


- Level of detail value is fractional!
- Use fractional part to blend (lin.) between two adjacent mipmap levels

Texture Anti-Aliasing: MIP Mapping

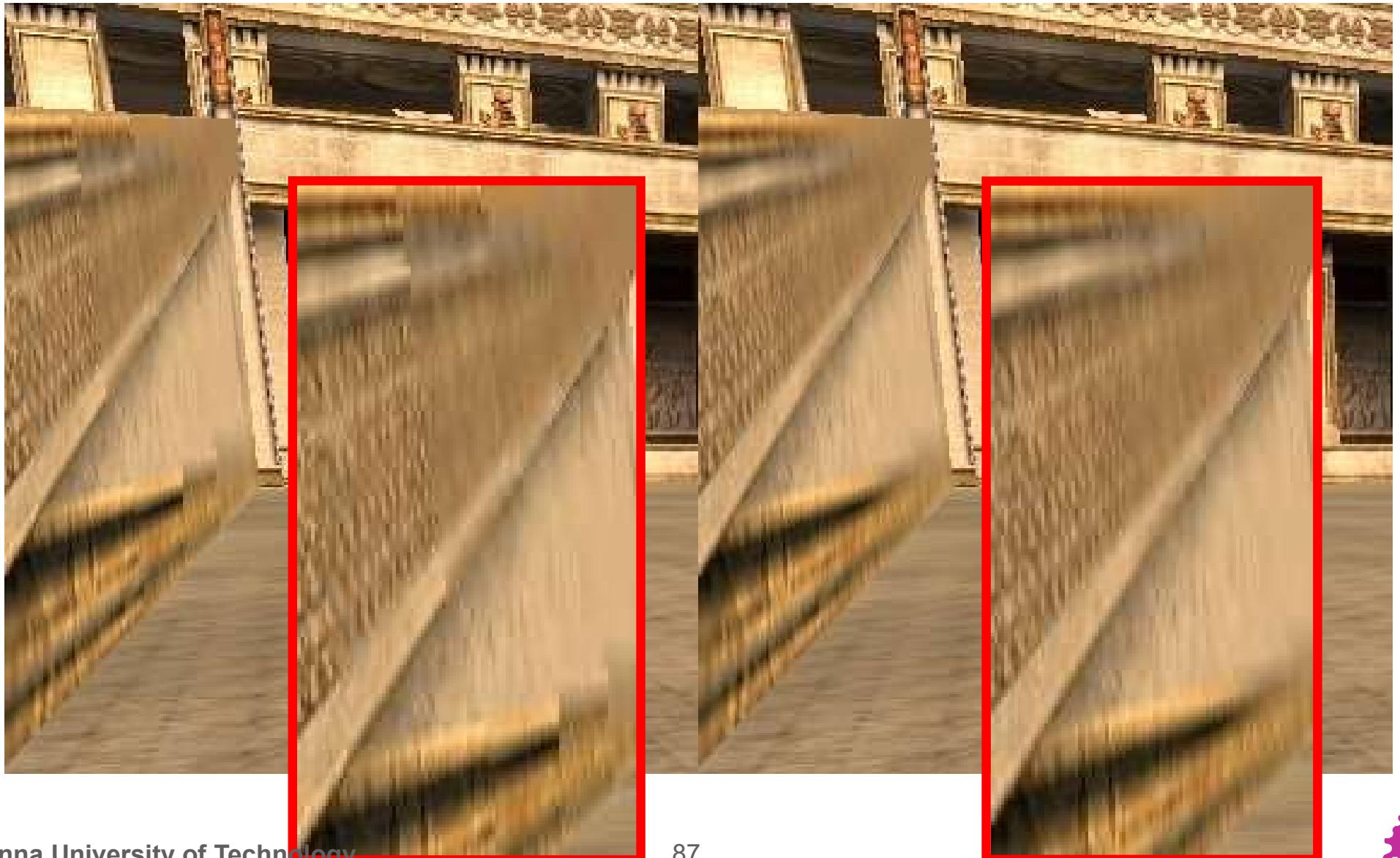
■ Trilinear interpolation:

- $T_1 :=$ 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)



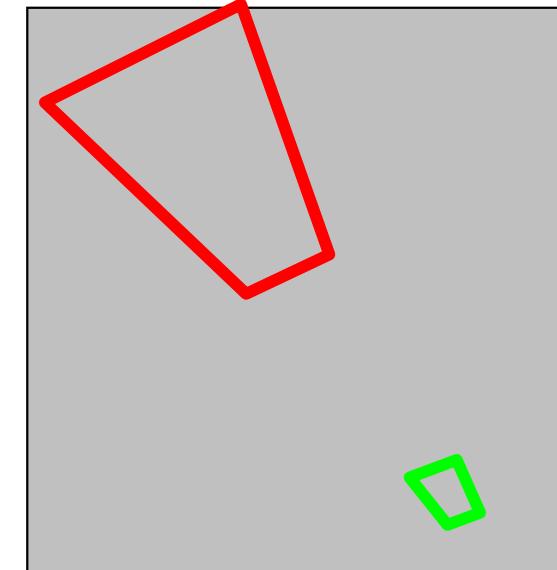
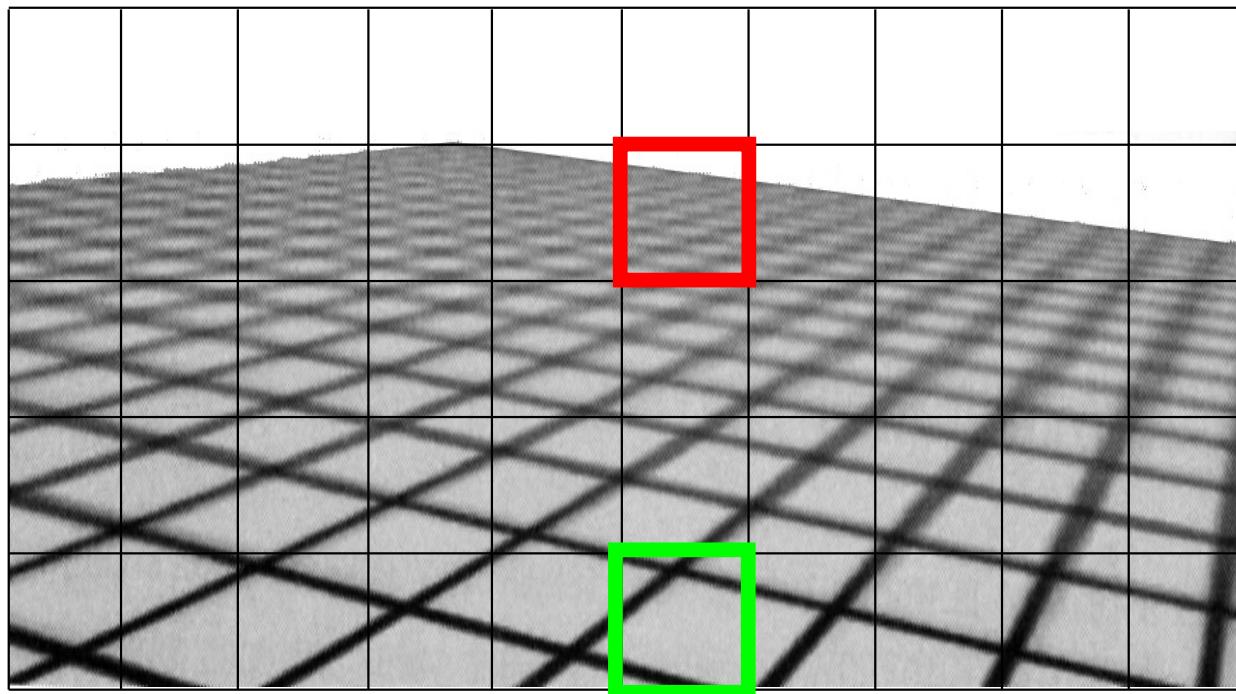
Texture Anti-Aliasing: MIP Mapping

- Other example for bilinear vs. trilinear filtering



Anti-Aliasing: Anisotropic Filtering

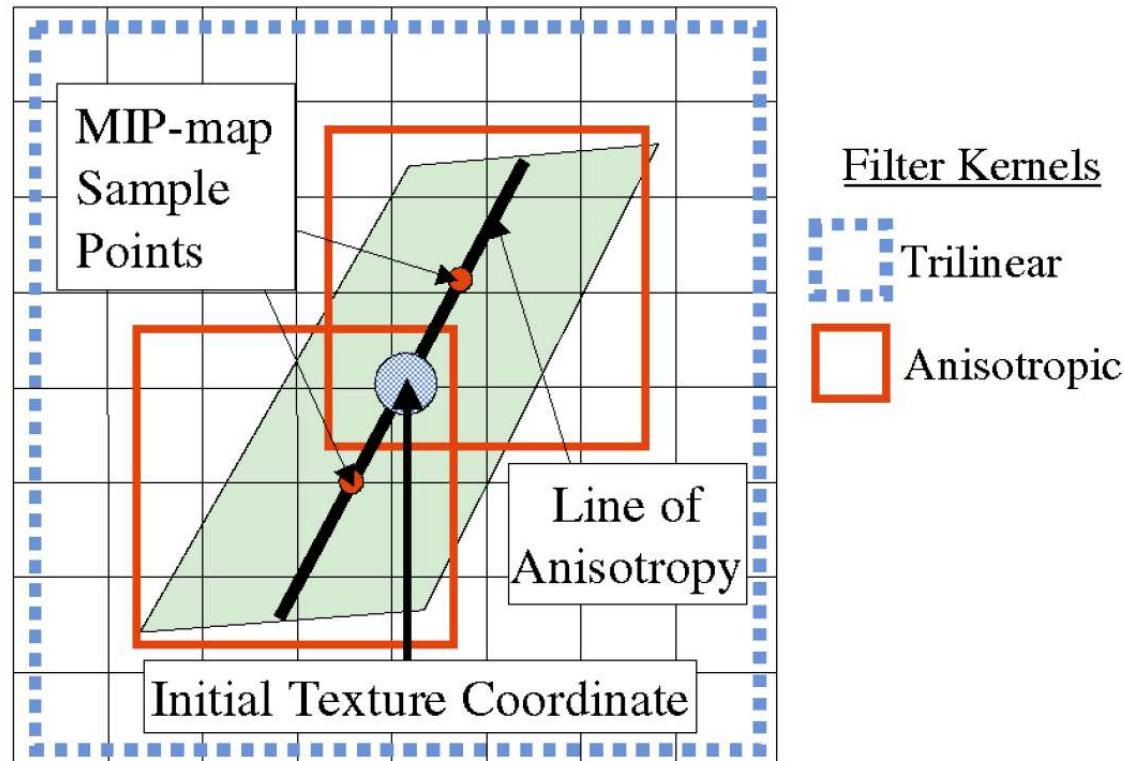
- Anisotropic filtering
 - View-dependent filter kernel
 - Implementation: *summed area table*, "*RIP Mapping*", *footprint assembly*, *elliptical weighted average* (EWA)



Texture space

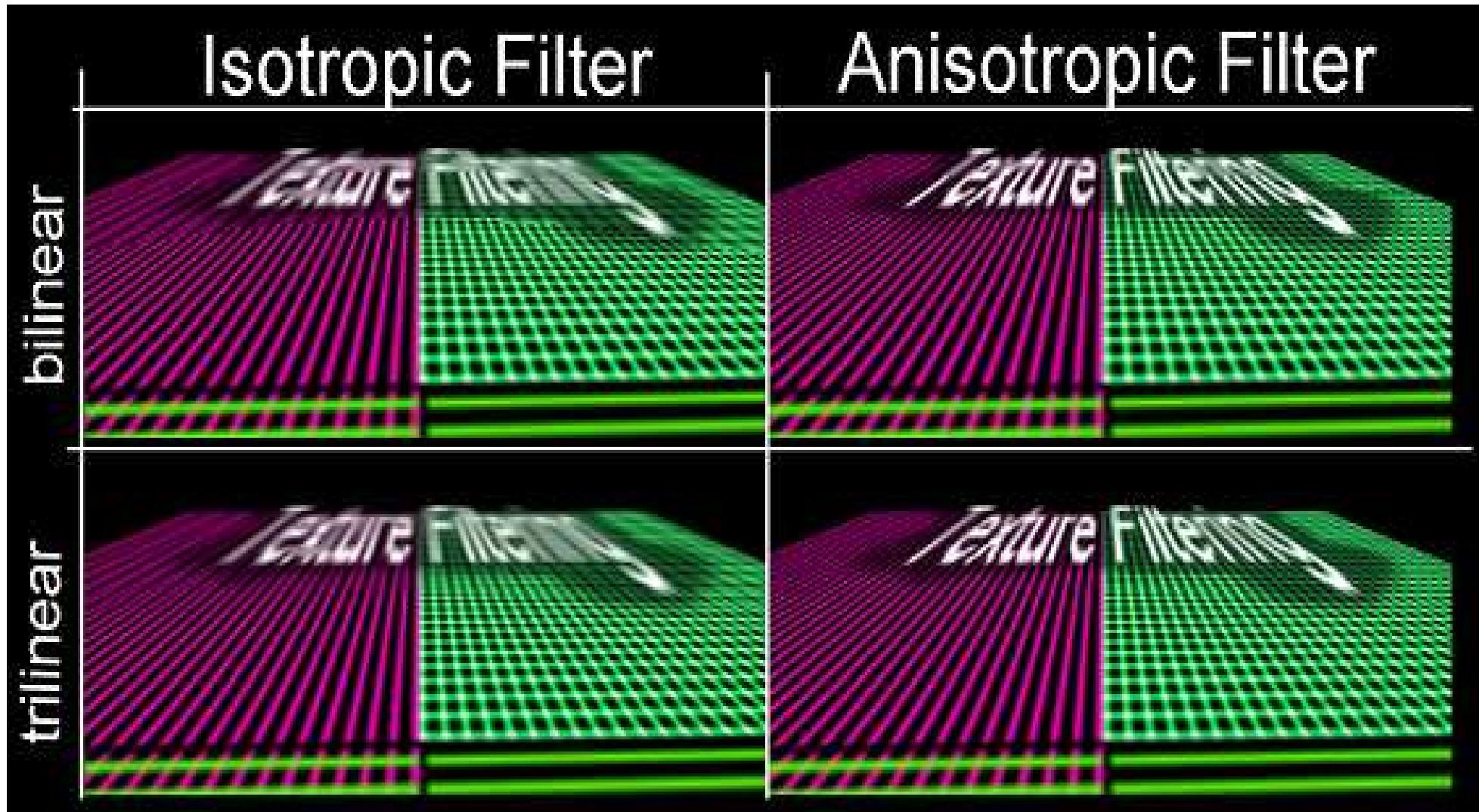


Anisotropic Filtering: Footprint Assembly



Anti-Aliasing: Anisotropic Filtering

■ Example



- 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.