CS 380 - GPU and GPGPU Programming
Lecture 16: GPU Texturing, Pt. 2

Markus Hadwiger, KAUST

Reading Assignment #10 (until Nov 6)

Read (required):

- MIP-Map Level Selection for Texture Mapping

  `https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=765326`

Read (optional):

- Vulkan Tutorial

  `https://vulkan-tutorial.com`

Next Lectures

Lecture 17: Tue, Nov 1 (make-up lecture; 16:00 – 17:15, room TBA)

Lecture 18: Wed, Nov 2

Quiz #2: Nov 9

Organization

- First 30 min of lecture
- No material (book, notes, ...) allowed

Content of questions

- Lectures (both actual lectures and slides)
- Reading assignments
- Programming assignments (algorithms, methods)
- Solve short practical examples

CUDA Update (11.8, updated)

CUDA SDK 11.8 and documentation now online

CUDA C Programming Guide

- New compute capability 9.0 (Hopper GPUs)
- New compute capability 8.9 (Ada Lovelace GPUs)

CUDA Binary Utilities

- New Hopper SASS (Ada Lovelace SASS is the same as Ampere)

CUDA NVCC Compiler Driver

- Support for cc 8.9 and 9.0 (PTX & cubin: sm_89, sm_90, compute_89, compute_90)

PTX ISA 7.8

- Support for cc 8.9 and 9.0 (sm_89 and sm_90) target architectures

Hopper Compatibility Guide, Hopper Tuning Guide

`https://developer.nvidia.com/blog/cuda-toolkit-11-8-new-features-revealed/`

Instruction Throughput

Instruction throughput numbers in CUDA C Programming Guide (Chapter 5.4)

	Compute Capability										
	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6	8.9	9.0
16-bit floating-point add, multiply, multiply-add	N/A		256	128	2	256	128	256 128 for __nv_bfloat16		128	256 128 for __nv_bfloat16
32-bit floating-point add, multiply, multiply-add	192	128		64	128		64		128		
64-bit floating-point add, multiply, multiply-add	64 8 for GeForce GPUs, except for Titan GPUs	4		32	4		32 2 for compute capability 7.5 GPUs	32	2	2	64

Instruction Throughput

Instruction throughput numbers in CUDA C Programming Guide (Chapter 5.4)

	Compute Capability										
	3.5, 3.7	5.0, 5.2	5.3	6.0	6.1	6.2	7.x	8.0	8.6	8.9	9.0
32-bit floating-point reciprocal, reciprocal square root, base-2 logarithm (`__log2f`), base 2 exponential (`exp2f`), sine (`__sinf`), cosine (`__cosf`)	32			16	32		16				
32-bit integer add, extended-precision add, subtract, extended-precision subtract	160	128		64	128		64				
32-bit integer multiply, multiply-add, extended-precision multiply-add	32	Multiple instruct.					64 32 for extended-precision				

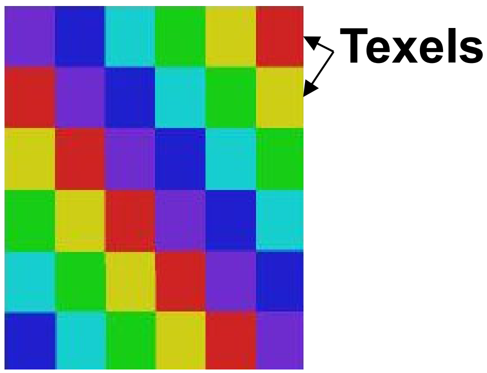
list continues…

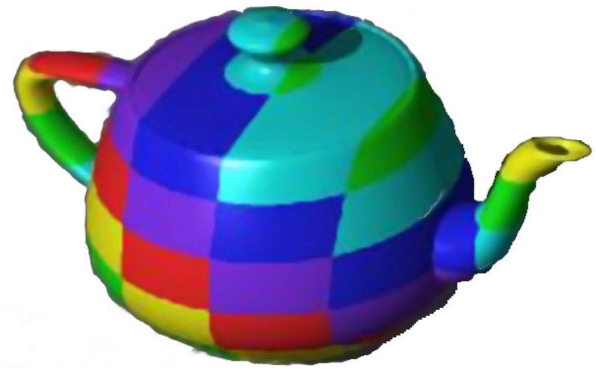
GPU Texturing

GPU Texturing

Rage / id Tech 5 (id Software)

Texturing: General Approach



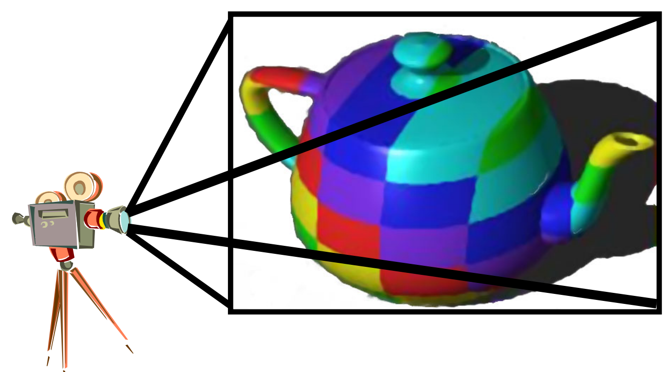


Texture space *(u,v)*

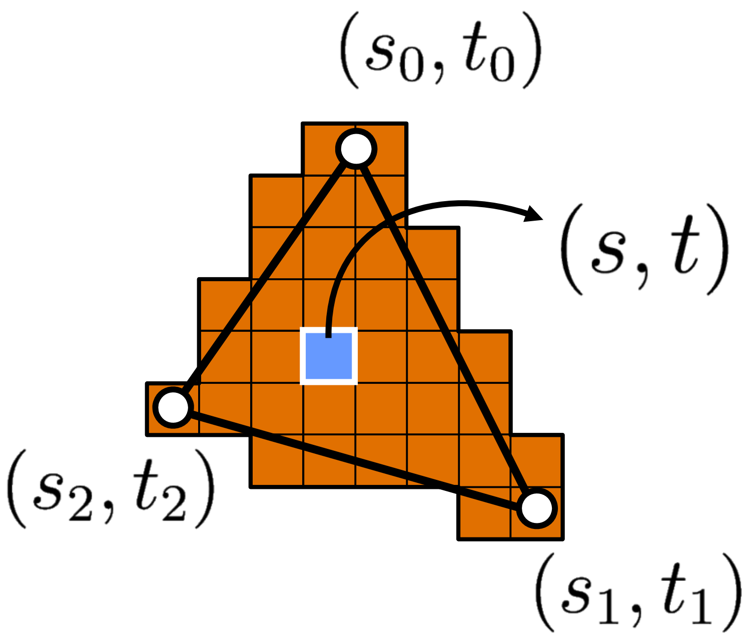
Object space (x_O,y_O,z_O)

Image Space (x_I,y_I)

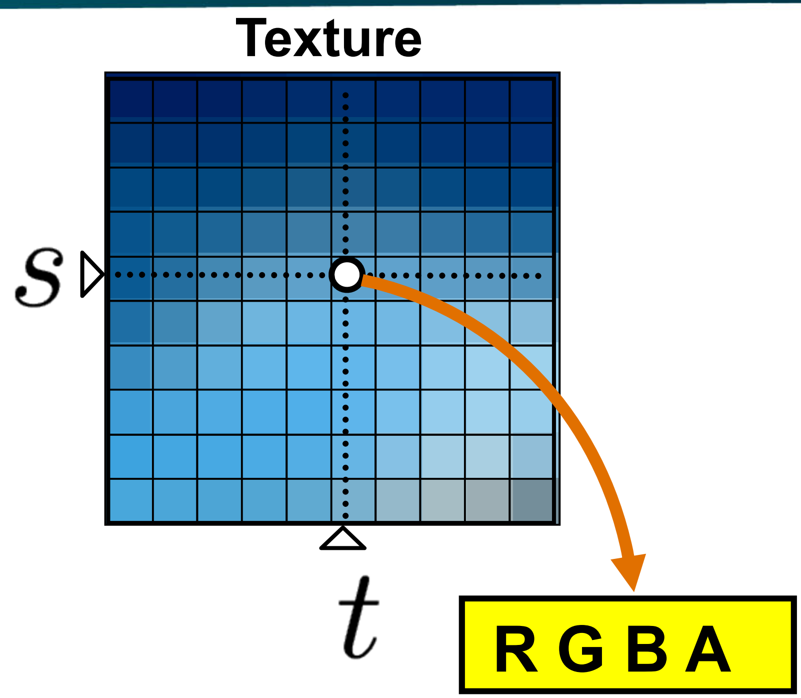
Parametrization → **Rendering (Projection etc.)**



2D Texture Mapping

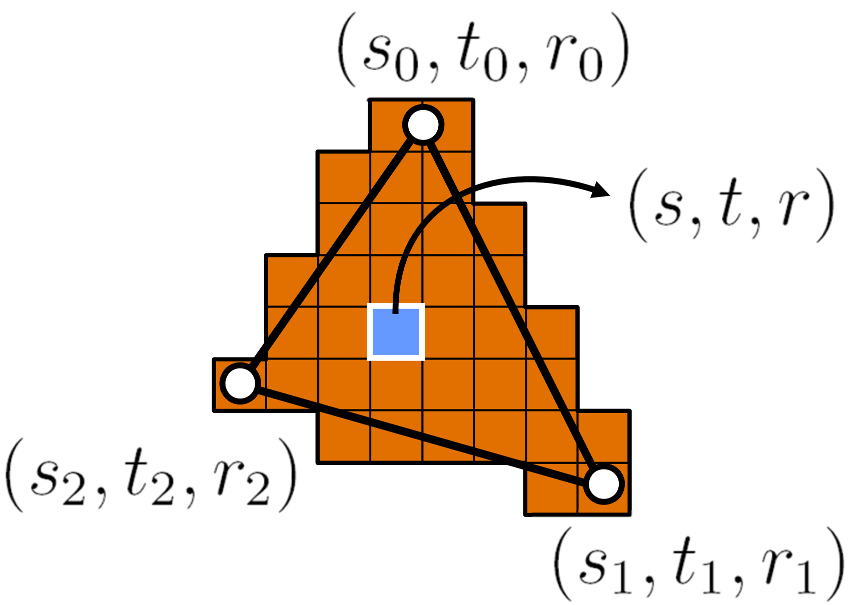


For each fragment:
interpolate the
texture coordinates
(barycentric)
Or:
Use arbitrary, computed coordinates

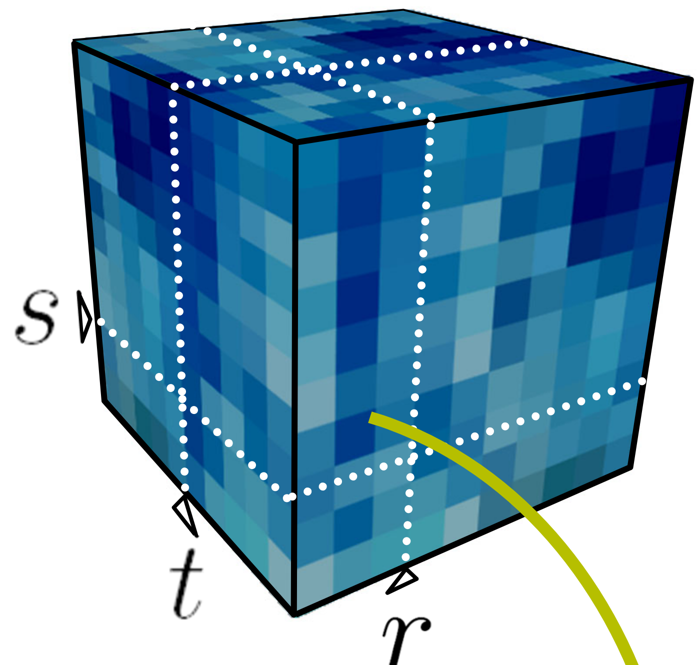


Texture-Lookup:
interpolate the
texture data
(bi-linear)
Or:
Nearest-neighbor for "array lookup"

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”

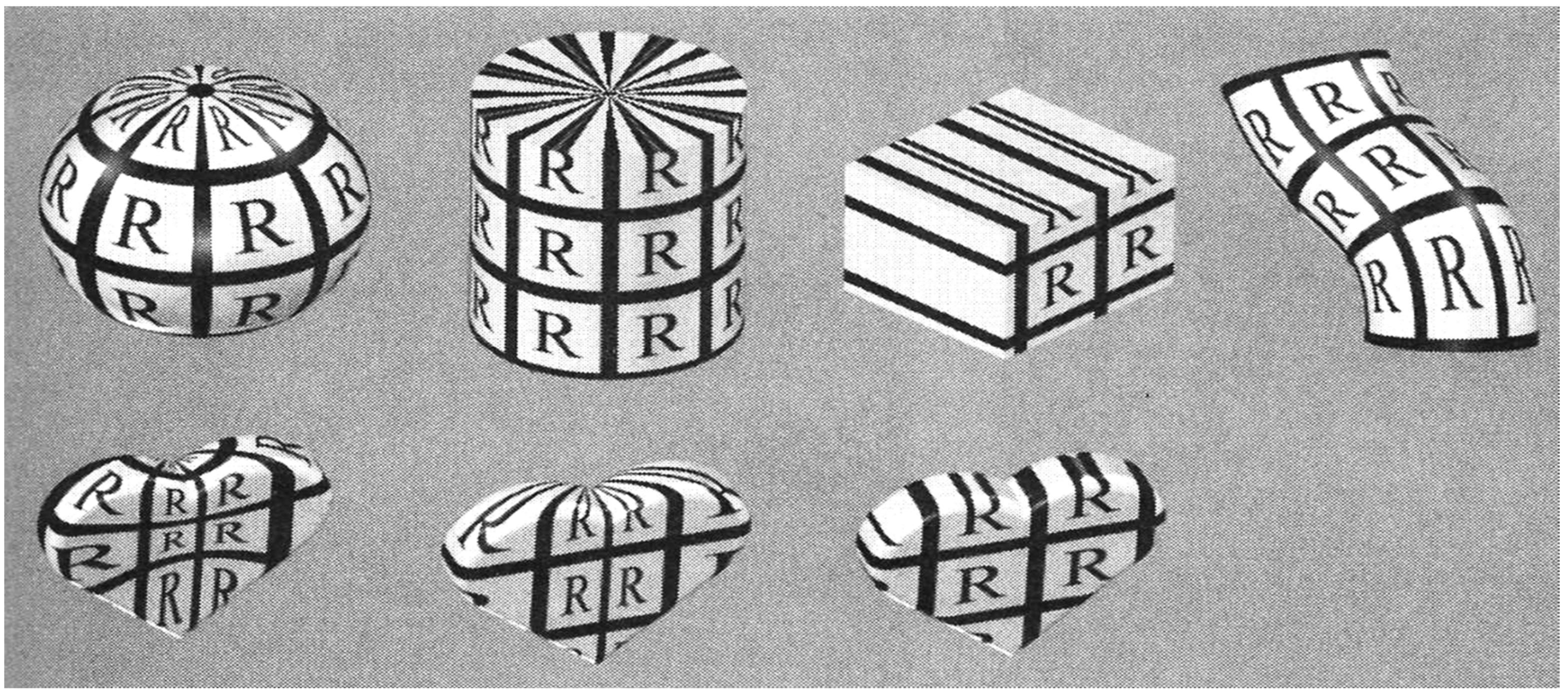
RGBA

Texture Projectors

Where do texture coordinates come from?

- Online: texture matrix/texcoord generation
- Offline: manually (or by modeling program)

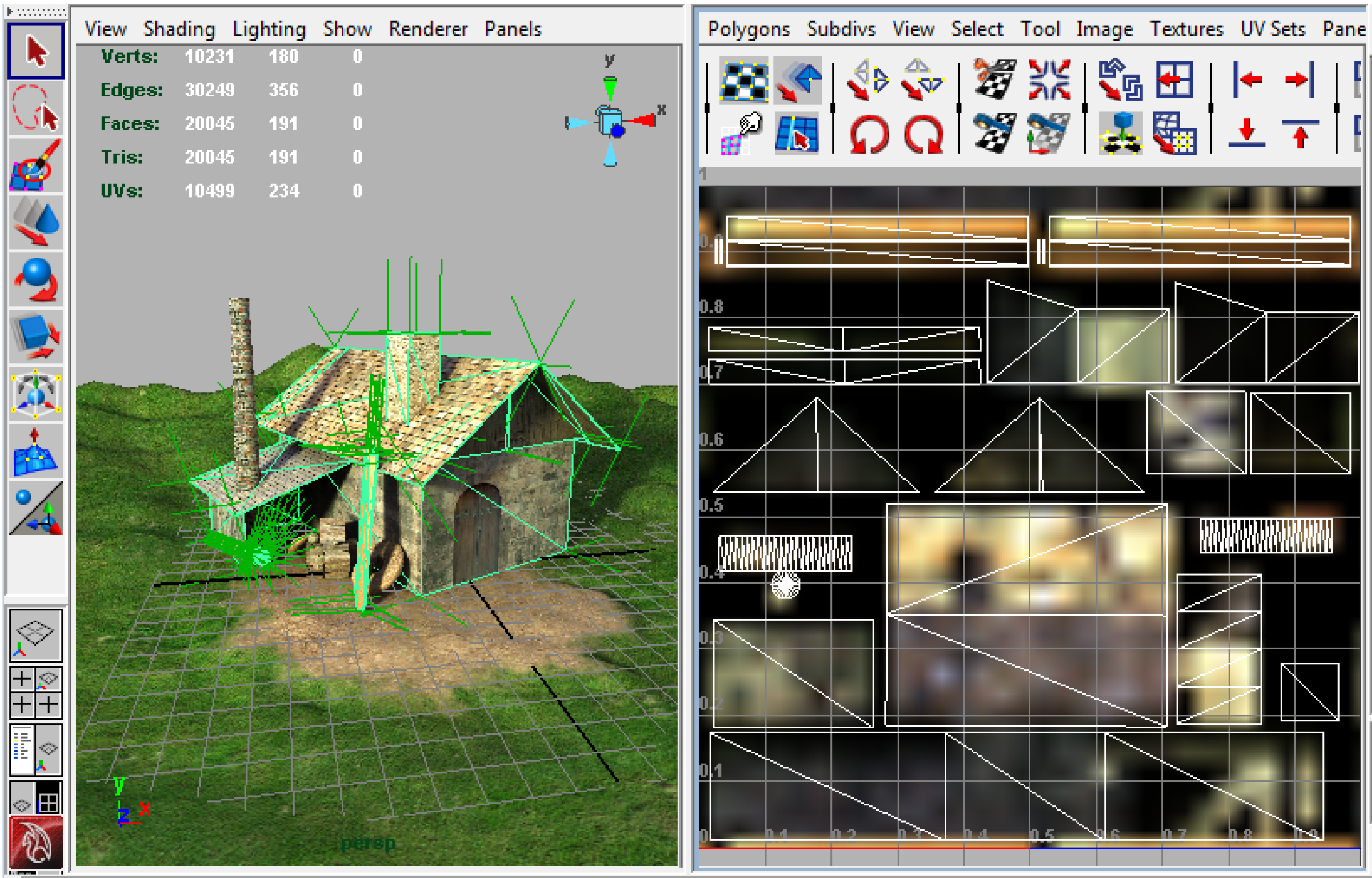
spherical *cylindrical* *planar* *natural*



Texture Projectors

Where do texture coordinates come from?

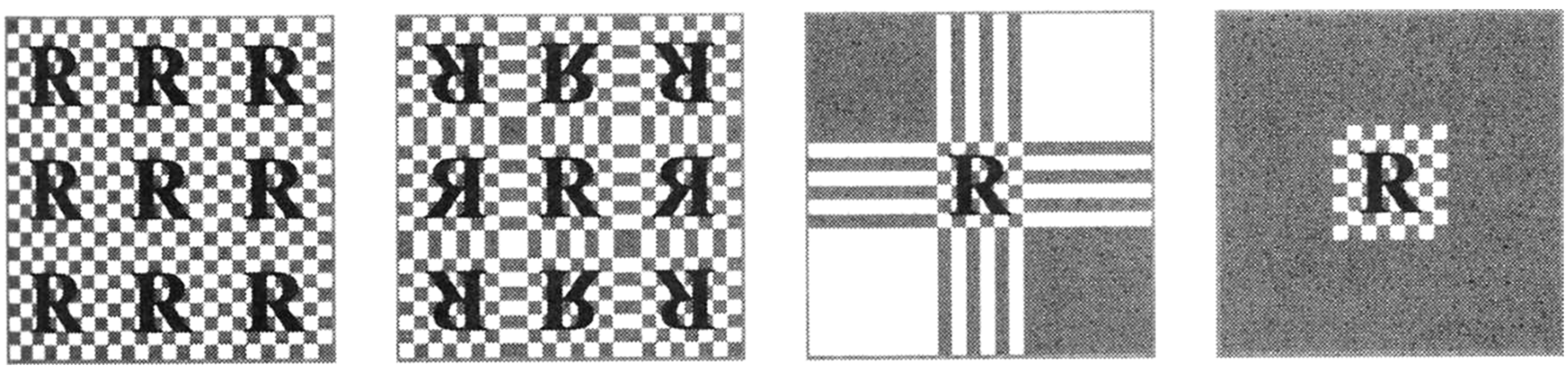
- Offline: manual UV coordinates by DCC program
- Note: a modeling problem!



Texture Wrap Mode

- How to extend texture beyond the border?
- Border and repeat/clamp modes
- Arbitrary (s,t,…) → [0,1] x [0,1] → [0,255] x [0,255]

repeat mirror/repeat clamp border



Interpolation #1

Interpolation Type + Purpose #1:

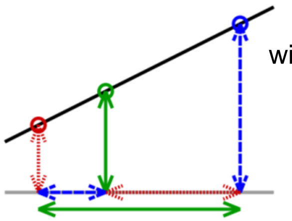
Interpolation of Texture Coordinates

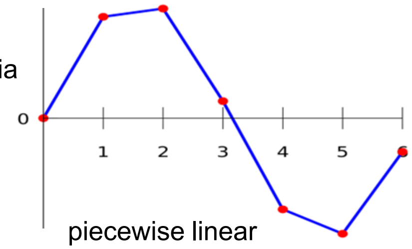
(Linear / Rational-Linear Interpolation)

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):

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

$$\alpha_1 + \alpha_2 = 1 \qquad \alpha = \alpha_2$$

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

Compare to line parameterization
with parameter t:

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

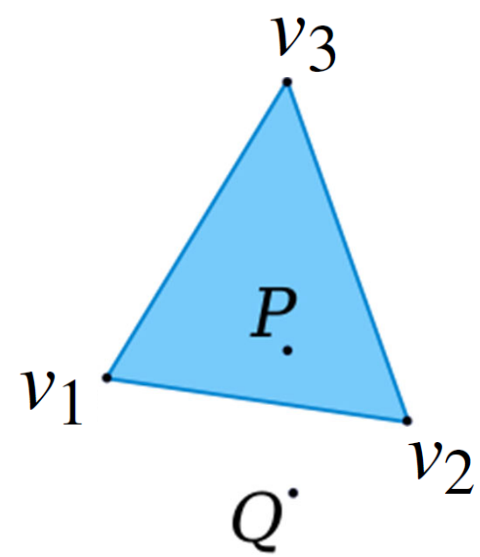
Linear Interpolation / Convex Combinations

Linear combination (n-dim. space):

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

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

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



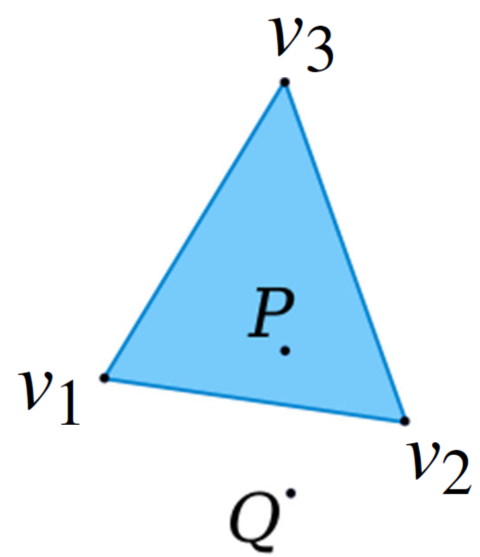
Convex combination: $\alpha_i \geq 0$

(restrict to simplex in subspace)

Linear Interpolation / Convex Combinations

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

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



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

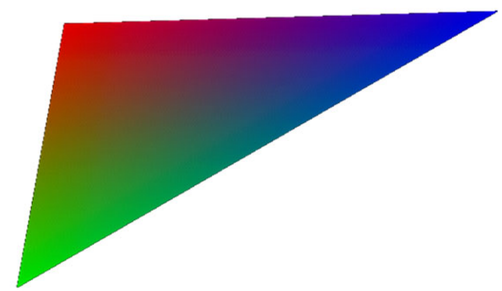
$\rightarrow$ linear attribute interpolation in simplex

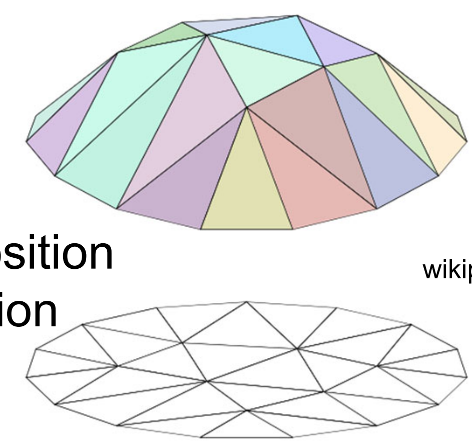
$$\alpha_1 v_1 + \alpha_2 v_2 + \ldots + \alpha_n v_n = \sum_{i=1}^{n} \alpha_i v_i$$

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

$$\alpha_i \geq 0$$

attribute interpolation





spatial position interpolation

wikipedia

Homogeneous Coordinates (1)

Projective geometry

- (Real) projective spaces 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 $(x, y, 0)$

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:

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

- 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

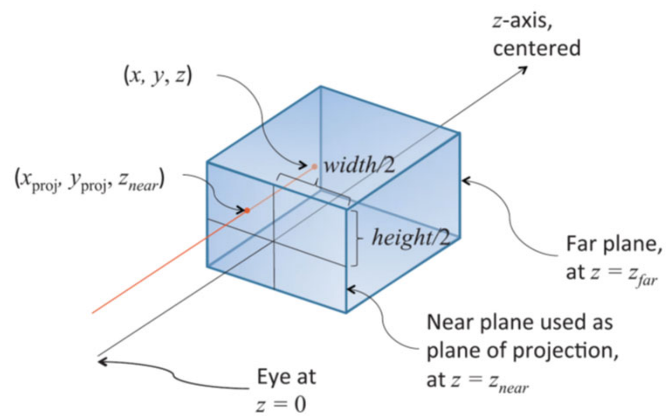
Homogeneous Coordinates (3)

Examples of usage

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

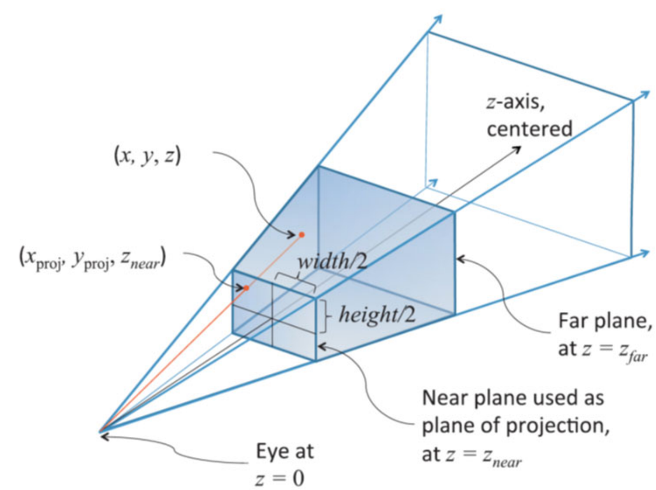
$$\begin{bmatrix} \frac{2}{\text{right}-\text{left}} & 0 & 0 & -\frac{\text{right}+\text{left}}{\text{right}-\text{left}} \\ 0 & \frac{2}{\text{top}-\text{bottom}} & 0 & -\frac{\text{top}+\text{bottom}}{\text{top}-\text{bottom}} \\ 0 & 0 & \frac{-2}{\text{far}-\text{near}} & -\frac{\text{far}+\text{near}}{\text{far}-\text{near}}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

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

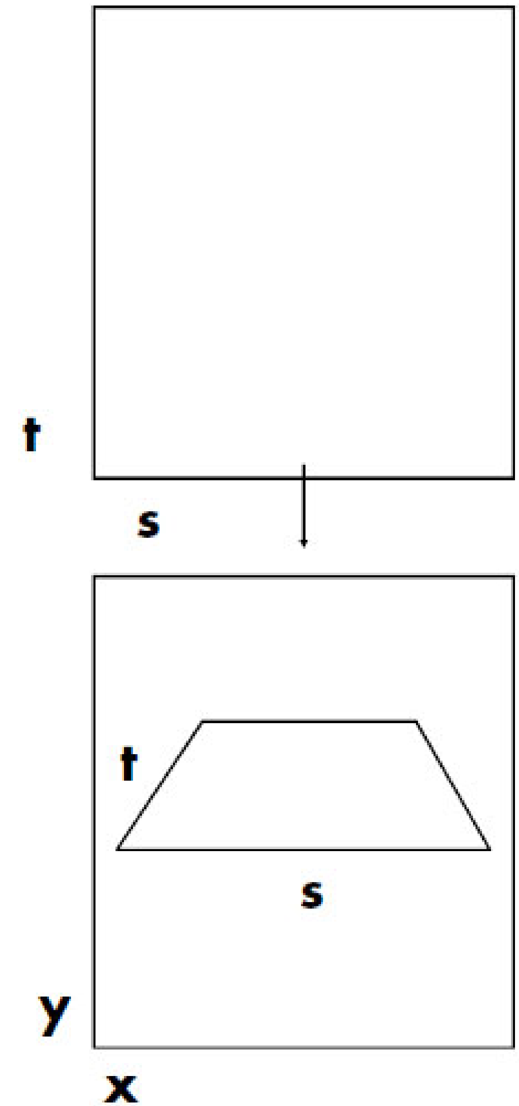
3D World Space

↓ Viewing Transformation

3D Camera Space

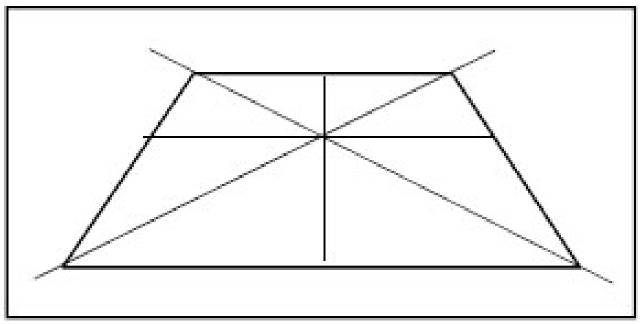
↓ Projection

2D Image Space

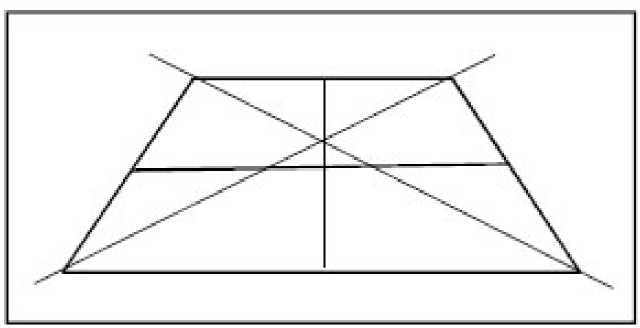


Kurt Akeley, Pat Hanrahan

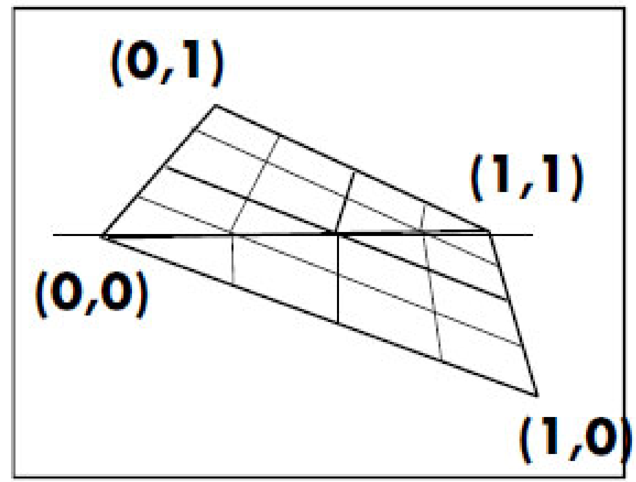
Linear Perspective



Correct Linear Perspective



Incorrect Perspective



Linear Interpolation, *Bad*

Perspective Interpolation, *Good*

Texture Mapping Polygons

Forward transformation: linear projective map

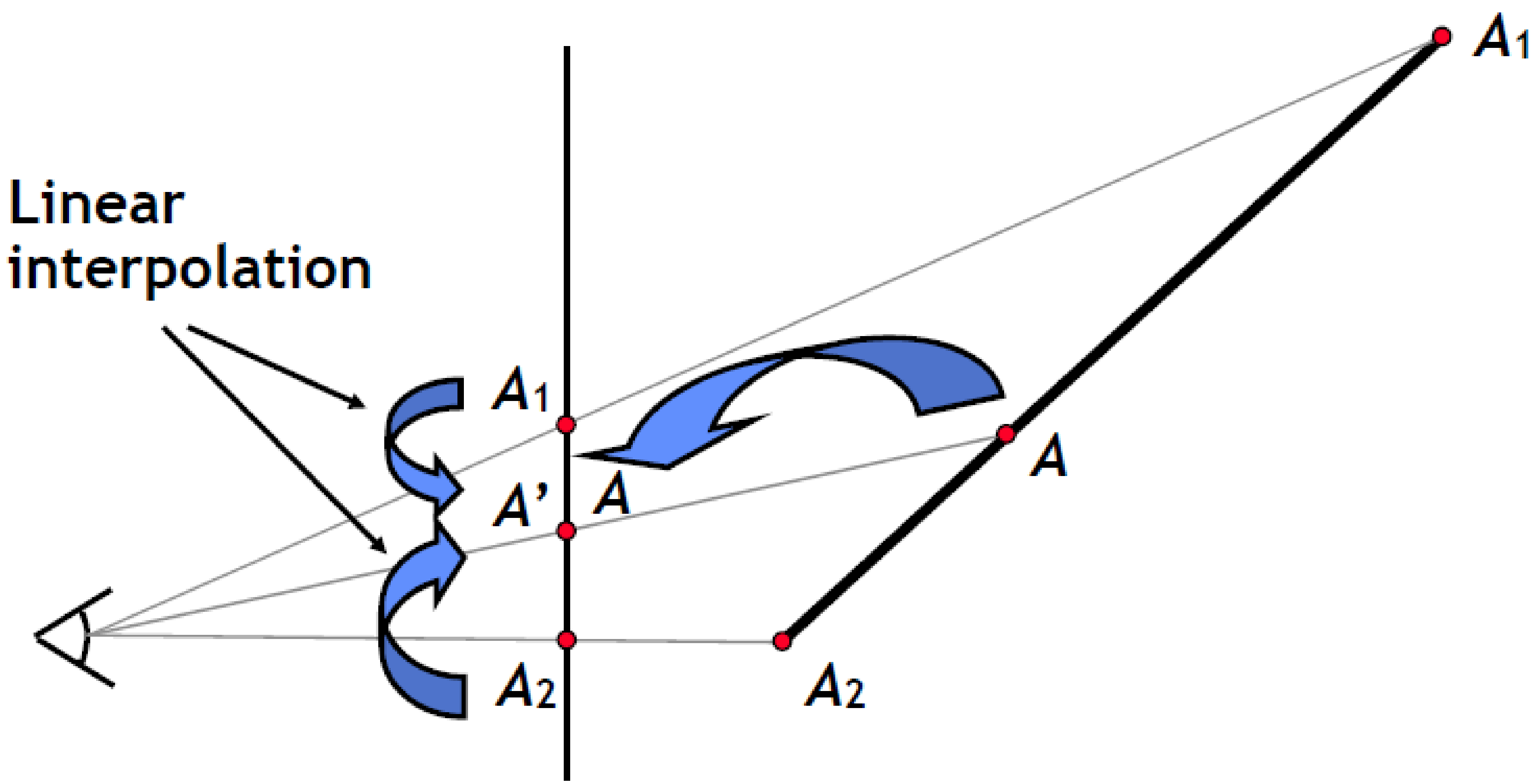
$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} a & b & c \\ d & e & f \\ g & h & i \end{bmatrix} \begin{bmatrix} s \\ t \\ r \end{bmatrix}$$

Backward transformation: linear projective map

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

Kurt Akeley, Pat Hanrahan

Incorrect attribute interpolation



$\boxed{A' \neq A\,!}$

Kurt Akeley, Pat Hanrahan

Linear interpolation

Compute intermediate attribute value

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

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

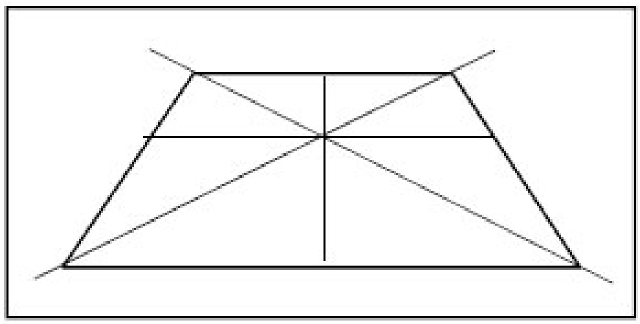
- x and y are projected (divided by w)
- Attribute values are not naturally projected

Choice for attribute interpolation in screen space

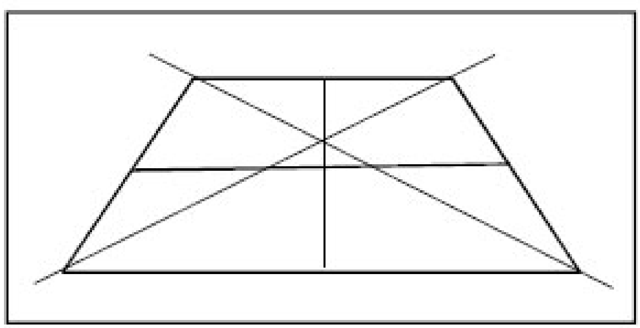
- Interpolate unprojected values
  - Cheap and easy to do, but gives wrong values
  - Sometimes OK for color, but
  - Never acceptable for texture coordinates
- Do it right

Kurt Akeley, Pat Hanrahan

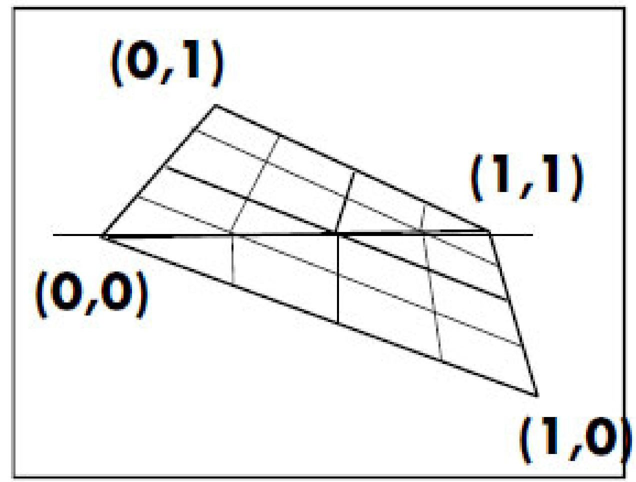
Linear Perspective



Correct Linear Perspective



Incorrect Perspective



Linear Interpolation, *Bad*

Perspective Interpolation, *Good*

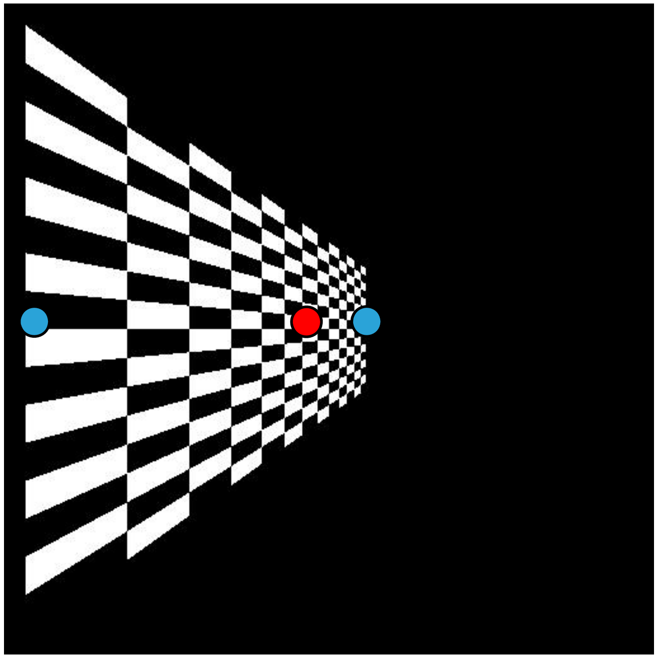
Kurt Akeley, Pat Hanrahan

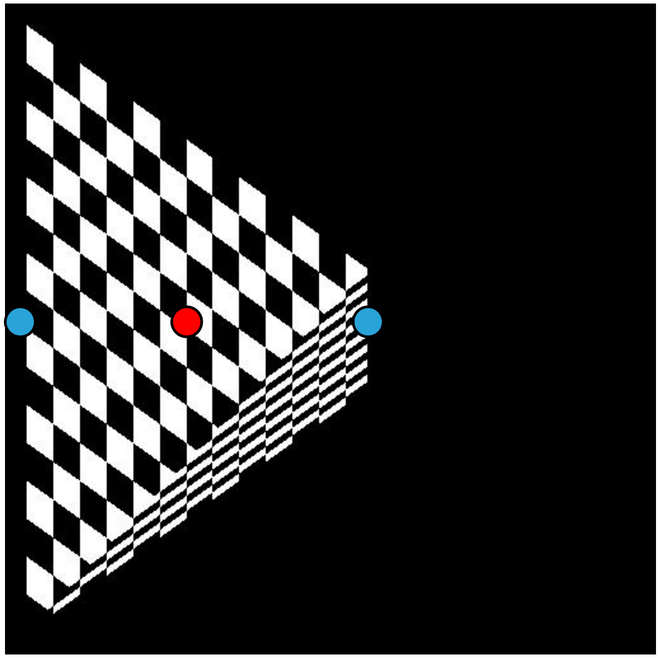
Perspective Texture Mapping

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

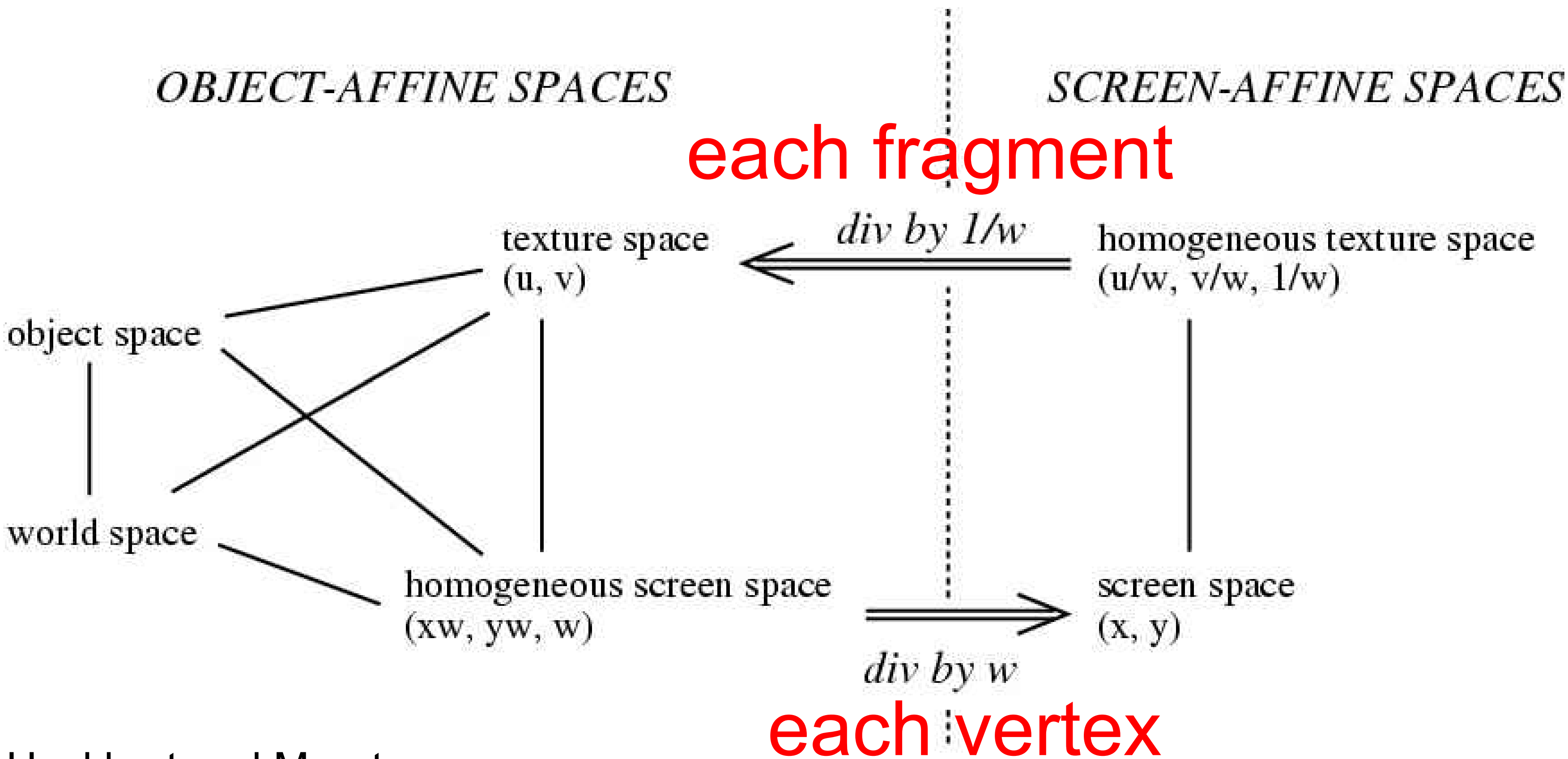
Barycentric triangle parameterization:

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

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.

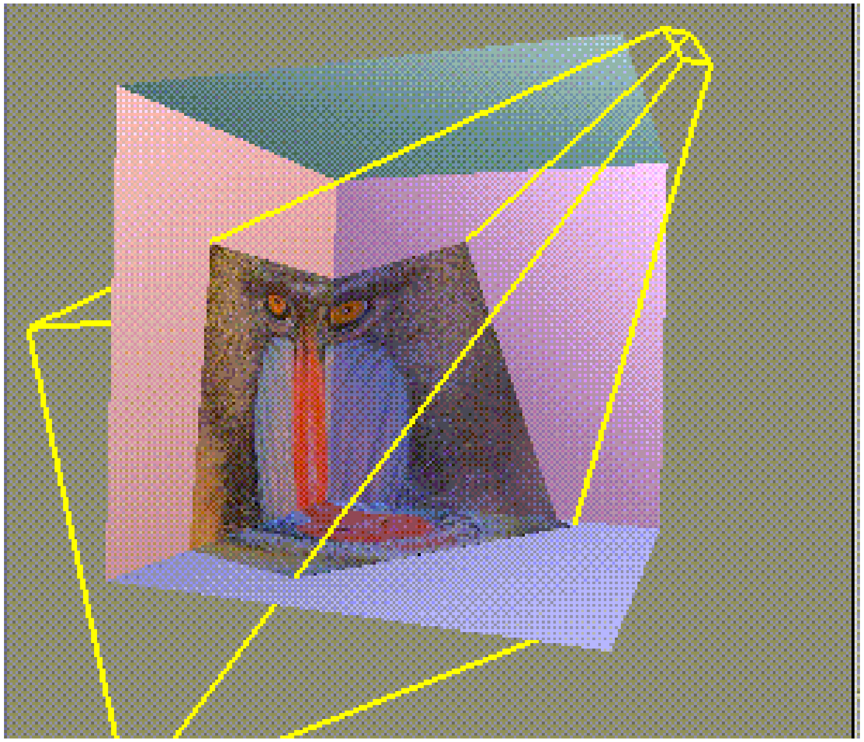
(4) At each vertex, divide the homogeneous screen coordinates, the parameters r_i, and the number 1 by w to construct the variable list $(x, y, z, s_1, s_2, \cdots, s_{n+1})$, where $s_i = r_i/w$ for $i \leq n$, $s_{n+1} = 1/w$.

(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.

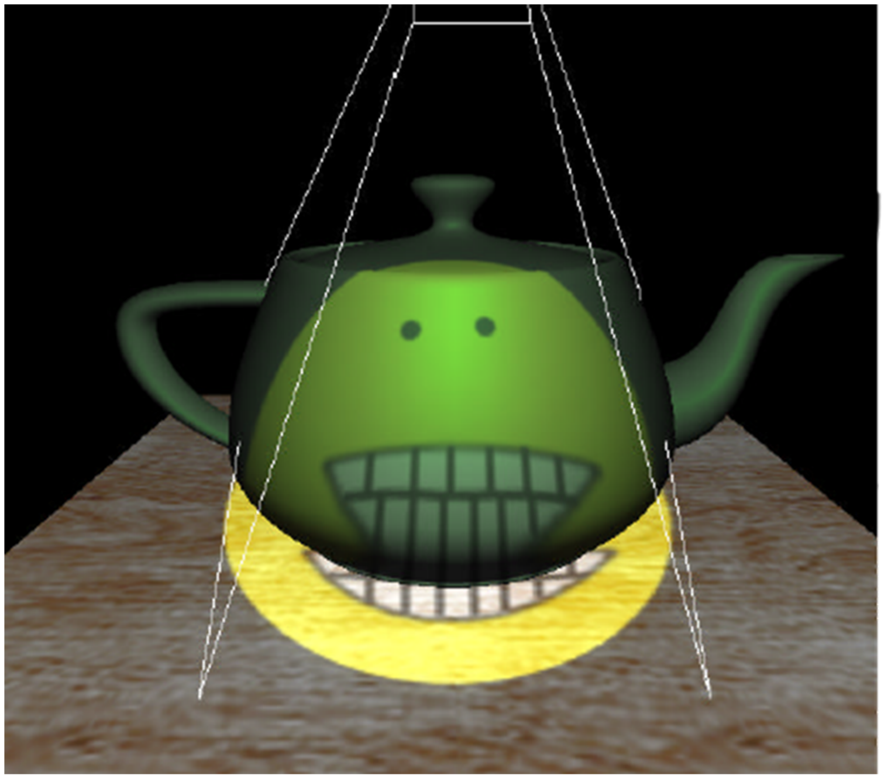
Heckbert and Moreton

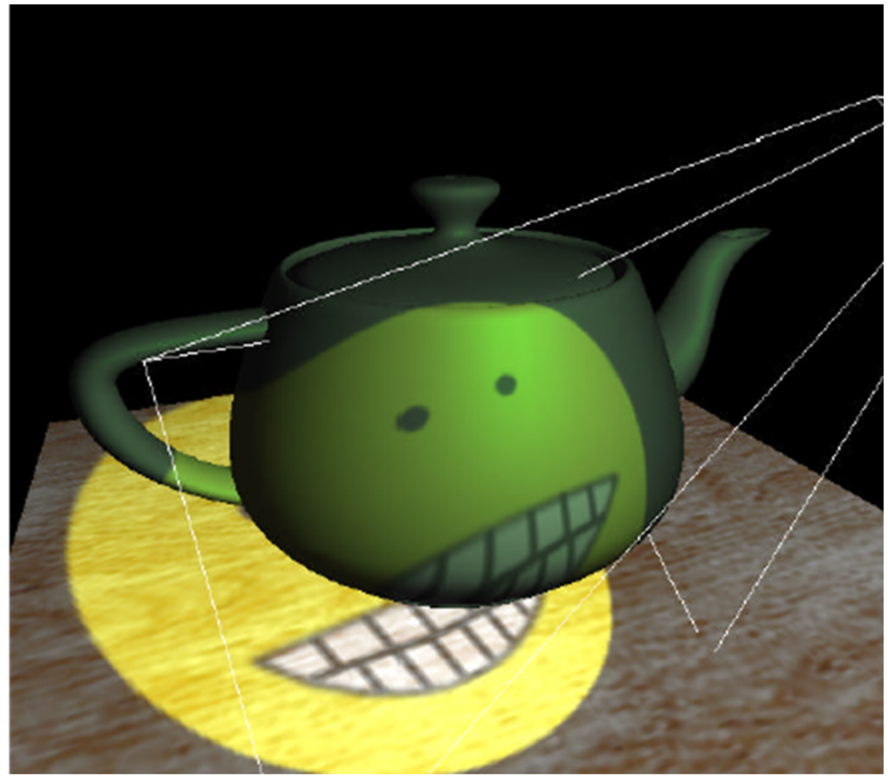
Projective Texture Mapping

- 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!

Thank you.