Real-Time Rendering (Echtzeitgraphik)

Dr. Michael Wimmer
wimmer@cg.tuwien.ac.at

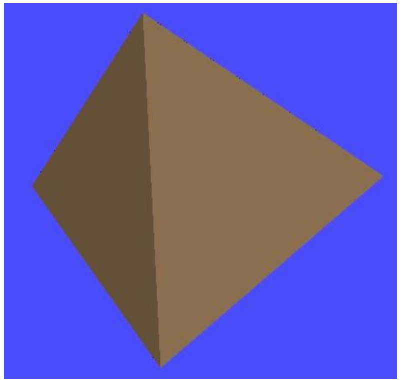
Texturing

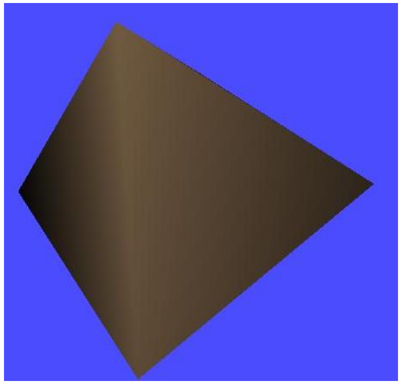
Overview

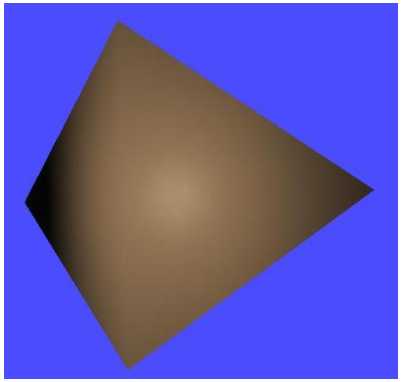
- OpenGL lighting refresher
- Texture Spaces
- Texture Aliasing and Filtering
- Multitexturing
  - Lightmapping
- Texture Coordinate Generation
- Projective Texturing
- Multipass Rendering

But Before We Start: Shading

- Flat shading
  - compute light interaction per polygon
  - the whole polygon has the same color
- Gouraud shading
  - compute light interaction per vertex
  - interpolate the colors
- Phong shading
  - interpolate normals per pixel
- Remember: difference between
  - Phong Light Model
  - Phong Shading

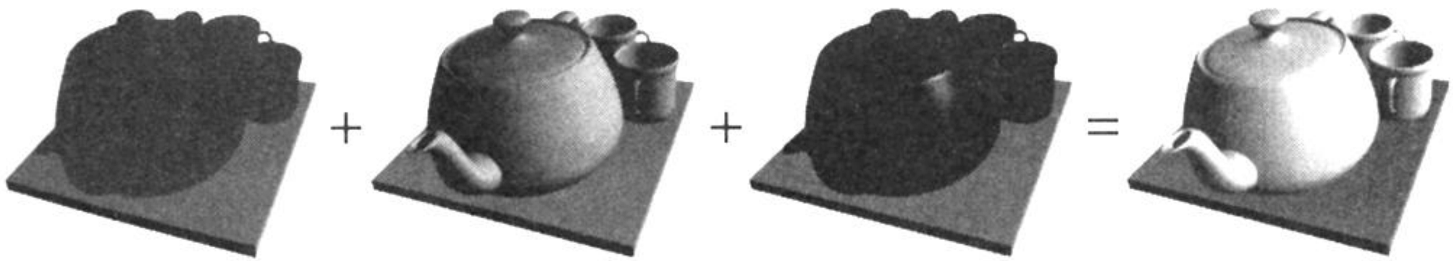






But Before We Start: OpenGL Lighting

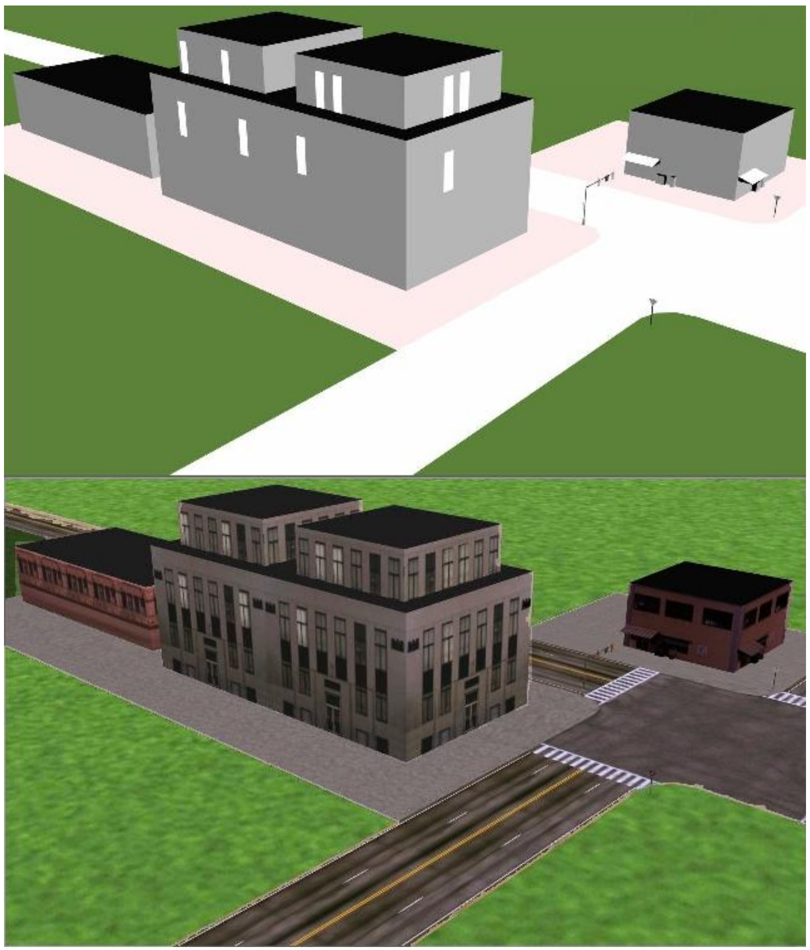
- Phong light model at each vertex (glLight, …)
- Local model only (no shadows, radiosity, …)
- ambient + diffuse + specular (glMaterial!)



- Fixed function: Gouraud shading
  - Note: need to interpolate specular separately!
- Phong shading: calculate Phong model in fragment shader

Why Texturing?

- Idea: enhance visual appearance of plain surfaces by applying fine structured details



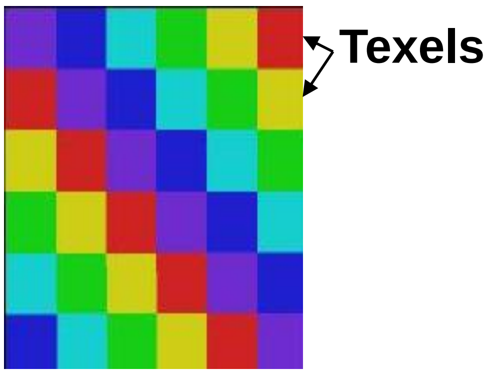
OpenGL Texture Mapping

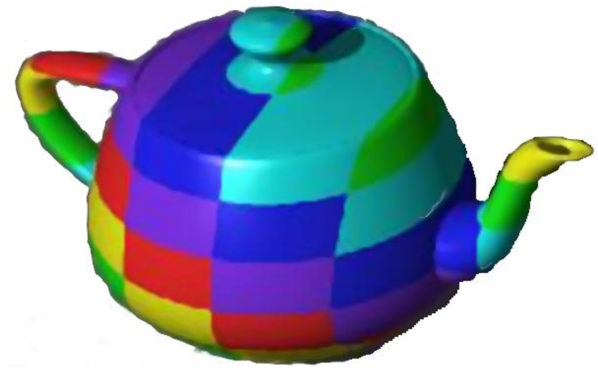
- Basis for most real-time rendering effects
- Look and feel of a surface
- Definition:
  - A *regularly sampled function* that is mapped onto every *fragment* of a surface
  - Traditionally an image, but…
- Can hold arbitrary information
  - Textures become general data structures
  - Will be interpreted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

Types of Textures

- Spatial Layout
  - 1D, 2D, 3D
  - Cube Maps
- Formats (too many), e.g. OpenGL
  - LUMINANCE16_ALPHA16: 32bit = 2 x 16 bit bump map
  - RGBA4: 16bit = 4 x 4 colors
  - RGBA_FLOAT32: 128 bit = 4 x 32 bit float
  - compressed formats, high dynamic range formats, …

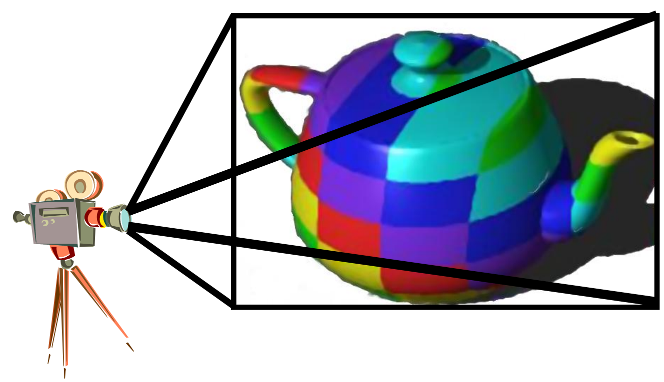
Texturing: General Approach



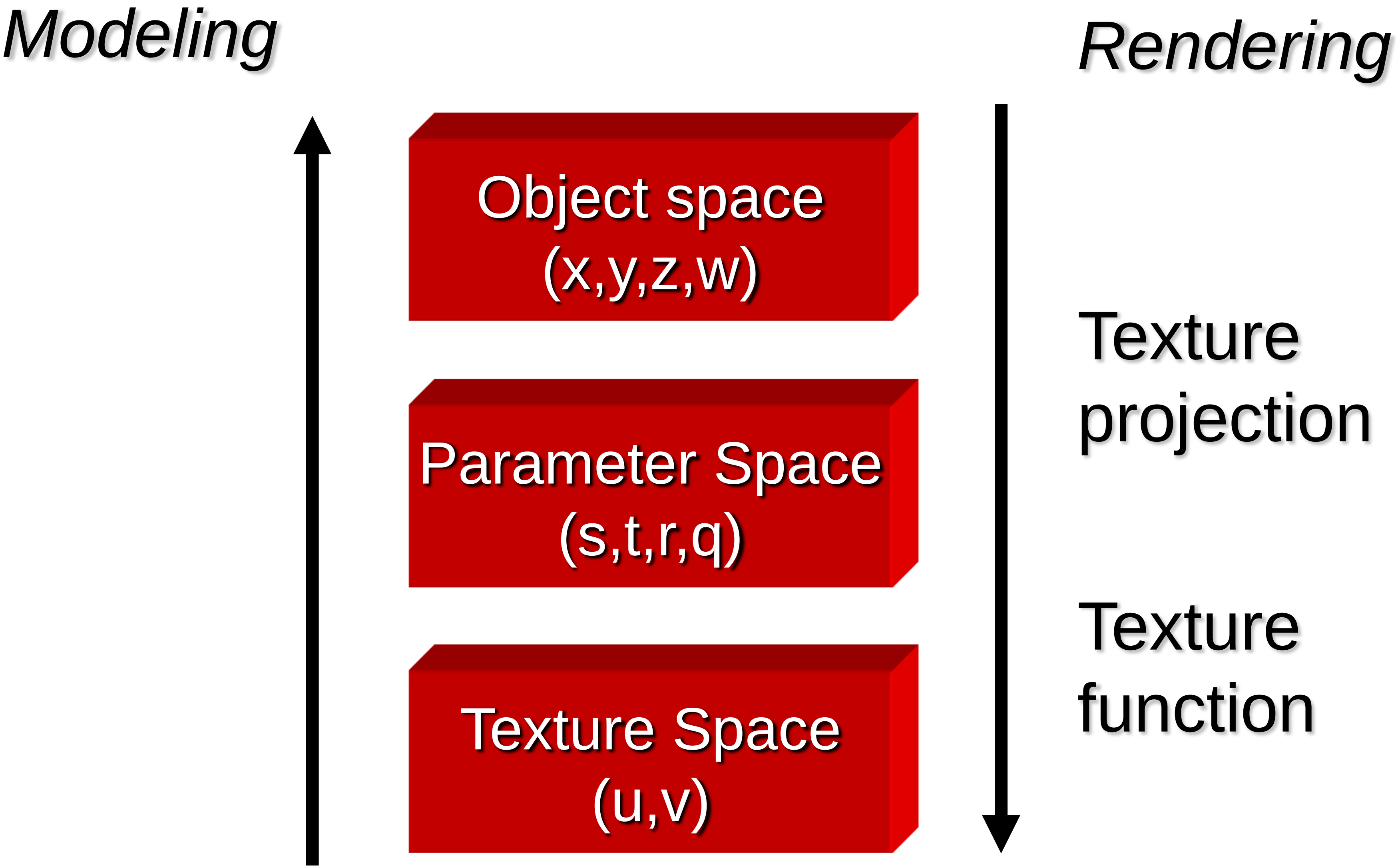


Texture space *(u,v)* **Object space** (x_O, y_O, z_O) **Image Space** (x_I, y_I)

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



Texture Spaces

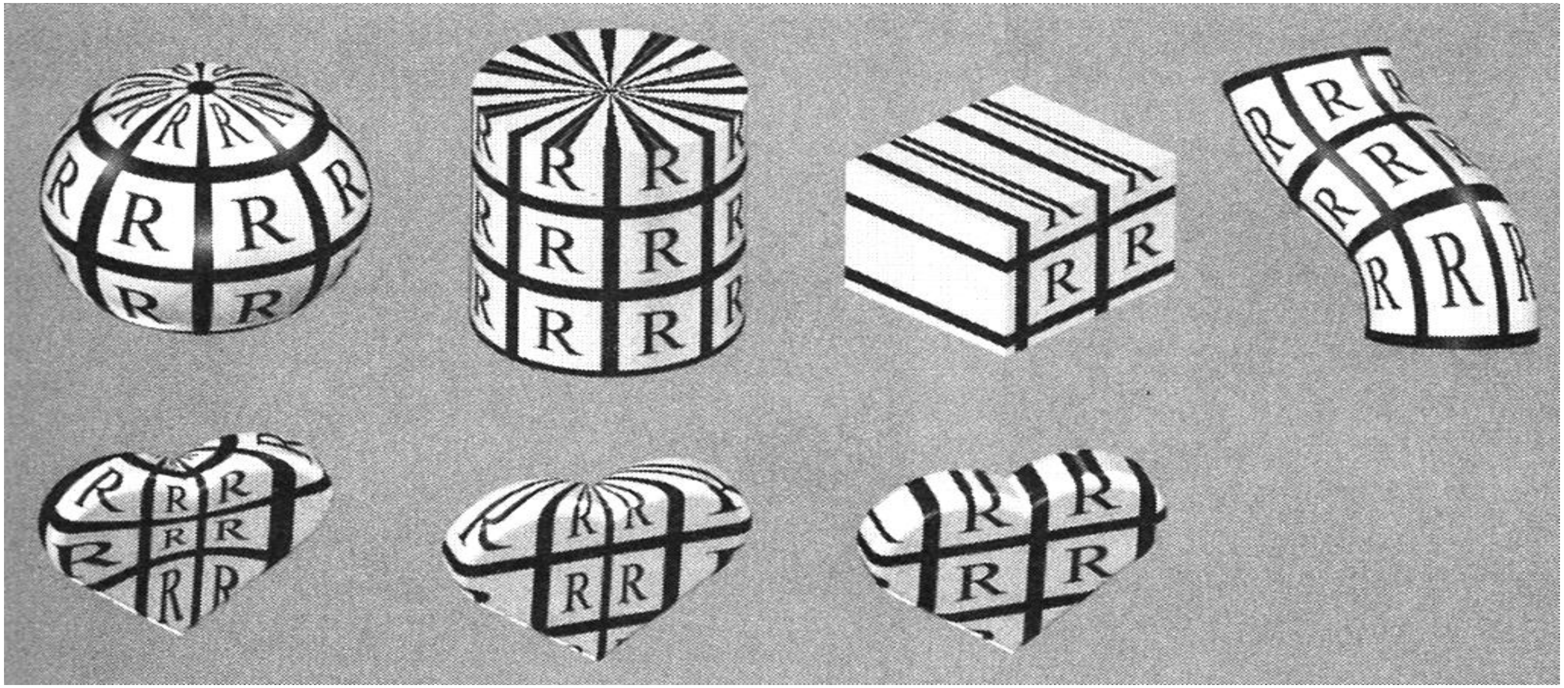


Texture Projectors

Where do texture coordinates come from?

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

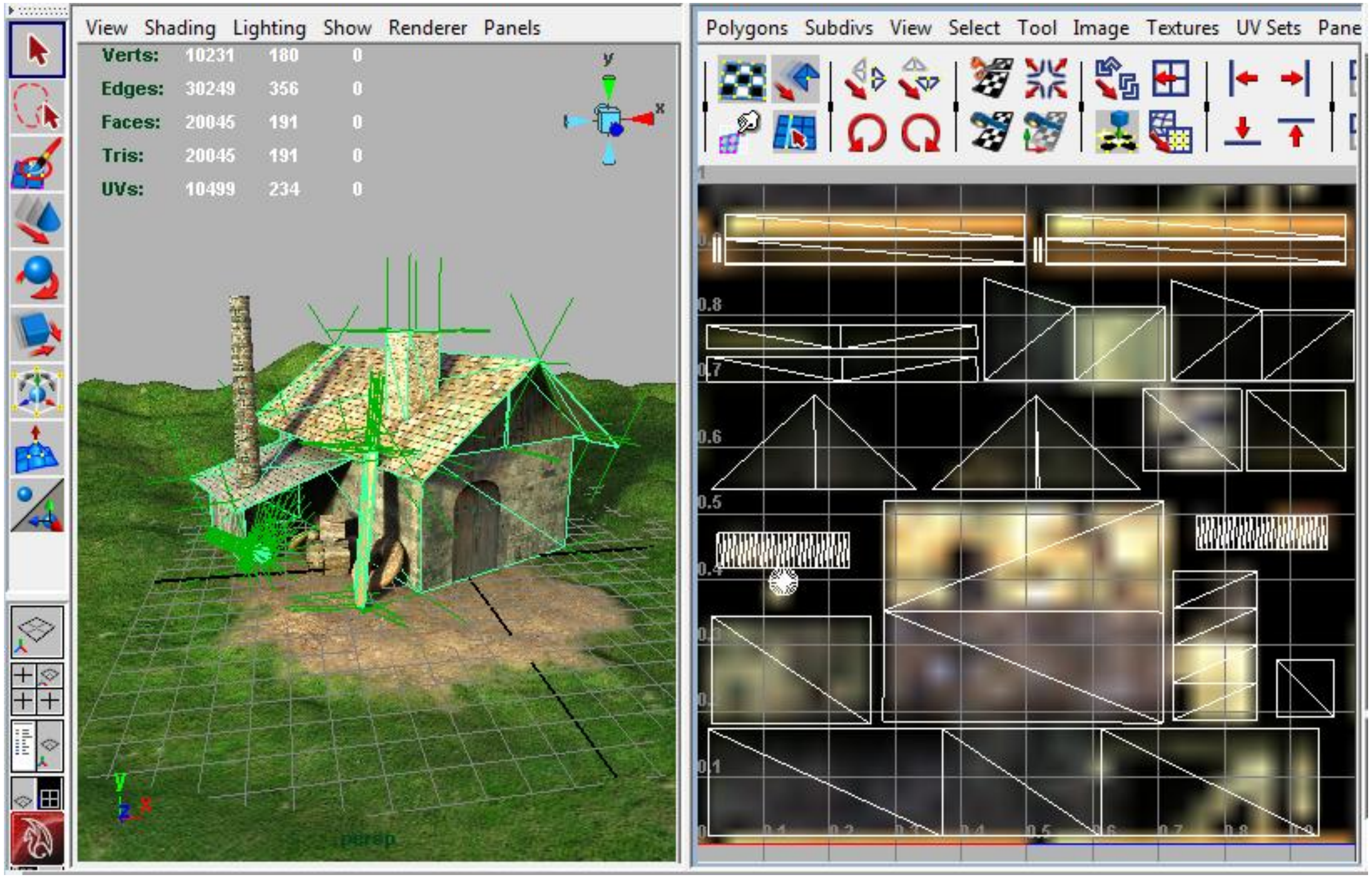
spherical cylindrical planar natural



Texture Projectors

Where do texture coordinates come from?

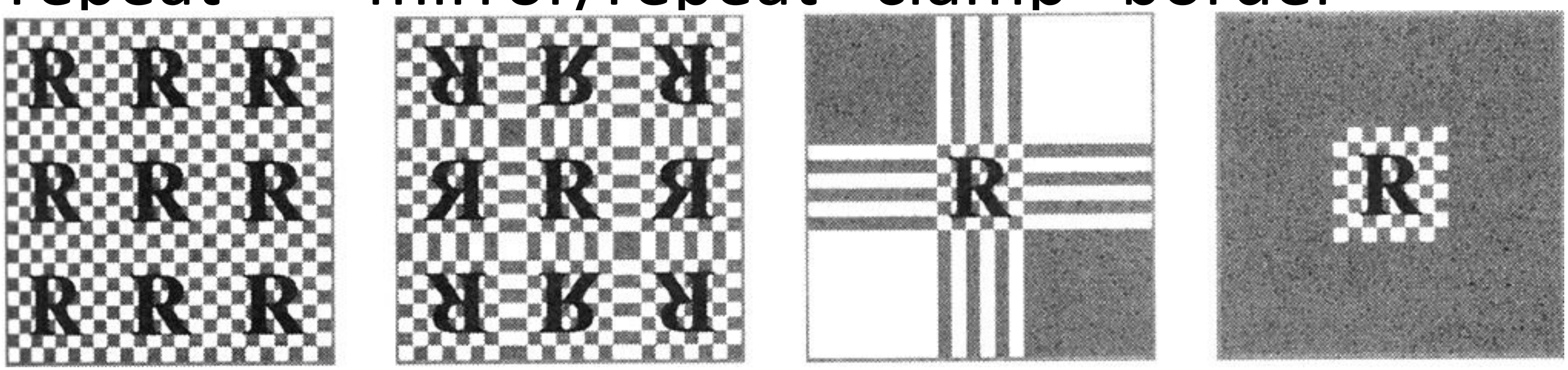
- Offline: manual UV coordinates by DCC program
- Note: a modeling Problem!



Texture Functions

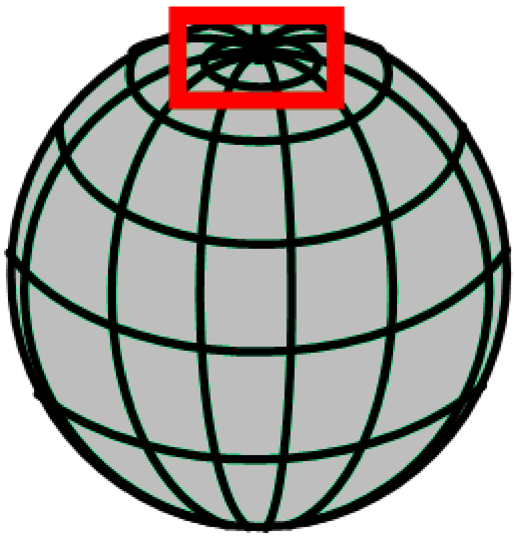
- How to extend texture beyond the border?
- Border and repeat/clamp modes
- Arbitrary (s,t,…) → [0,1] → [0,255]x[0,255]

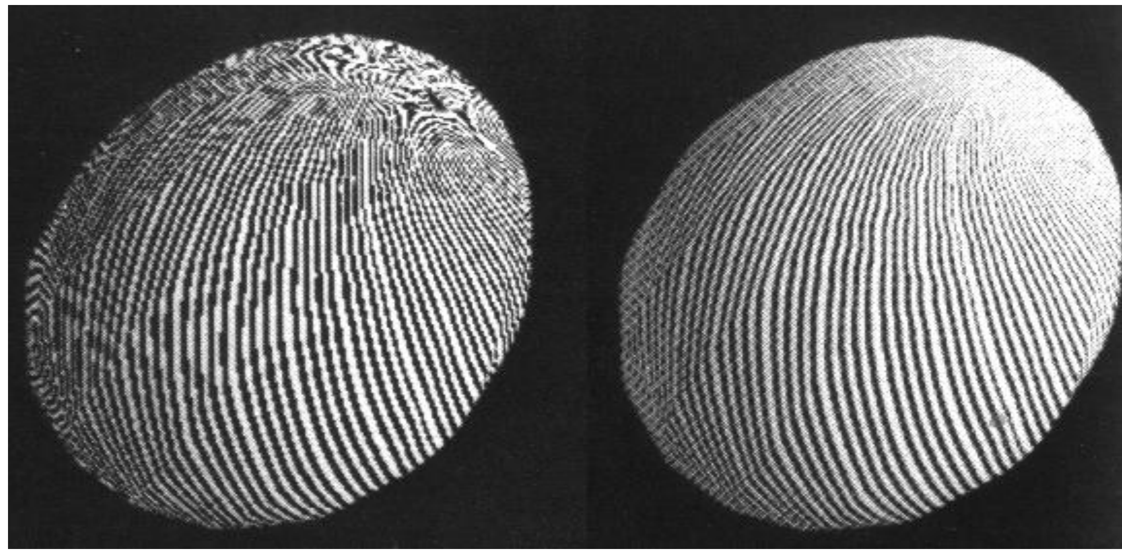
repeat mirror/repeat clamp border

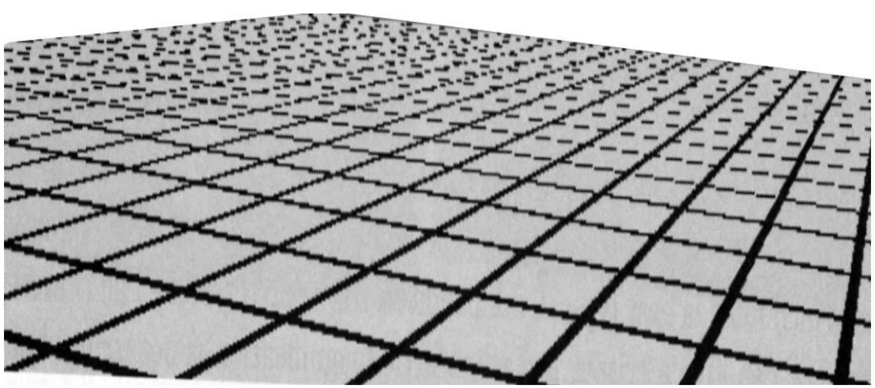


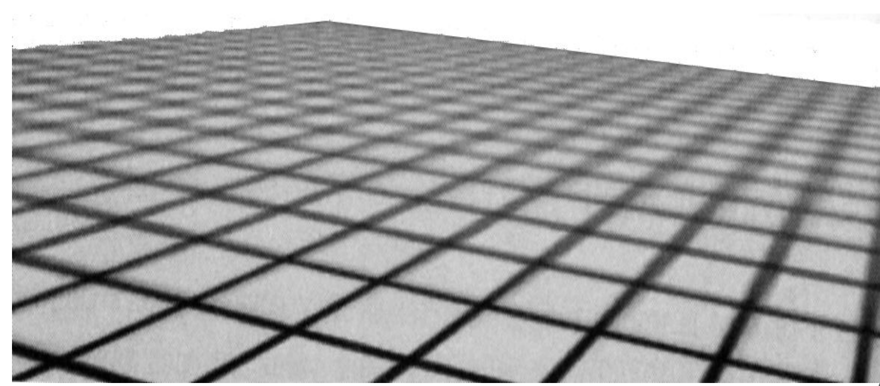
Texture Aliasing

- Problem: One pixel in image space covers many texels



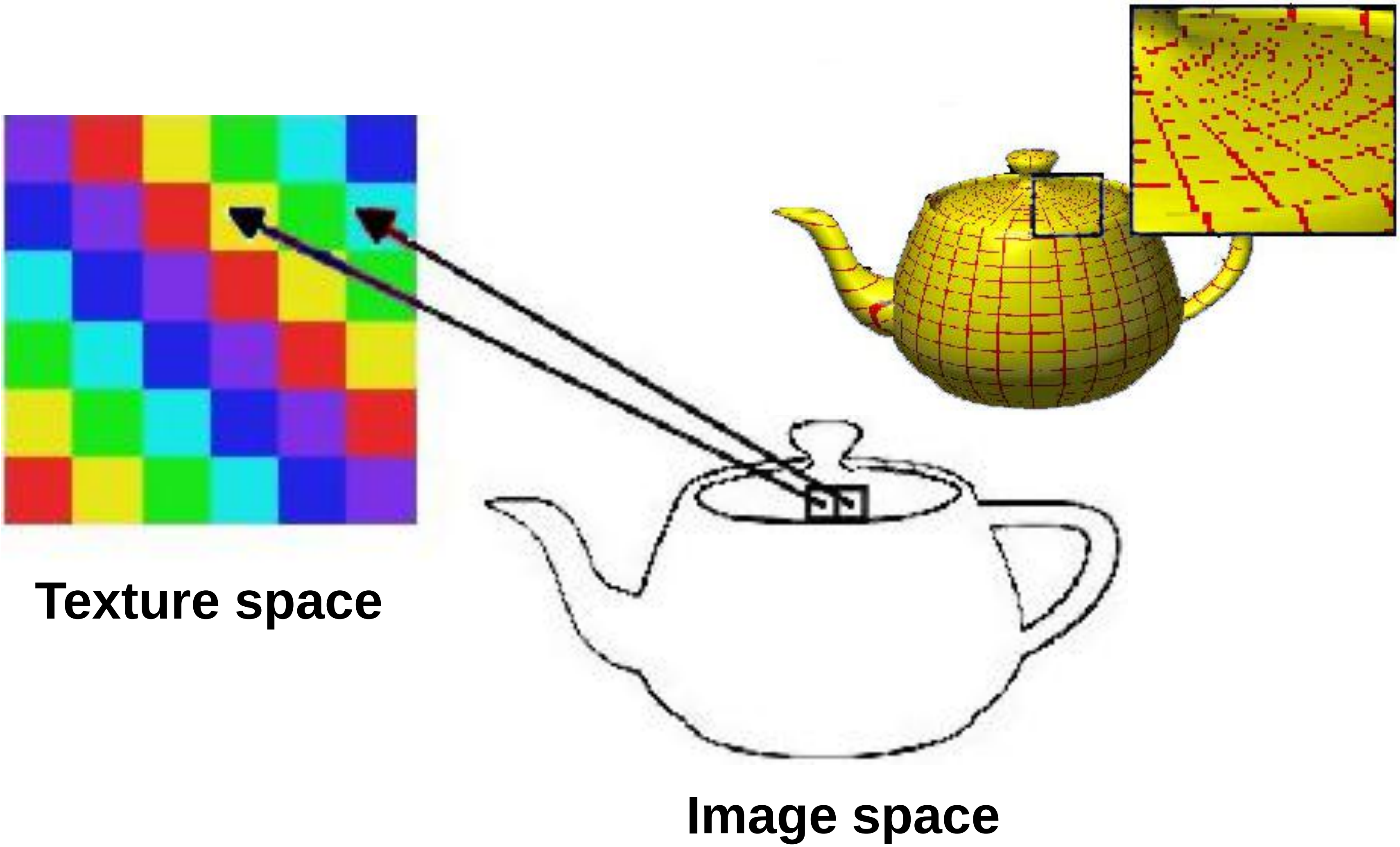






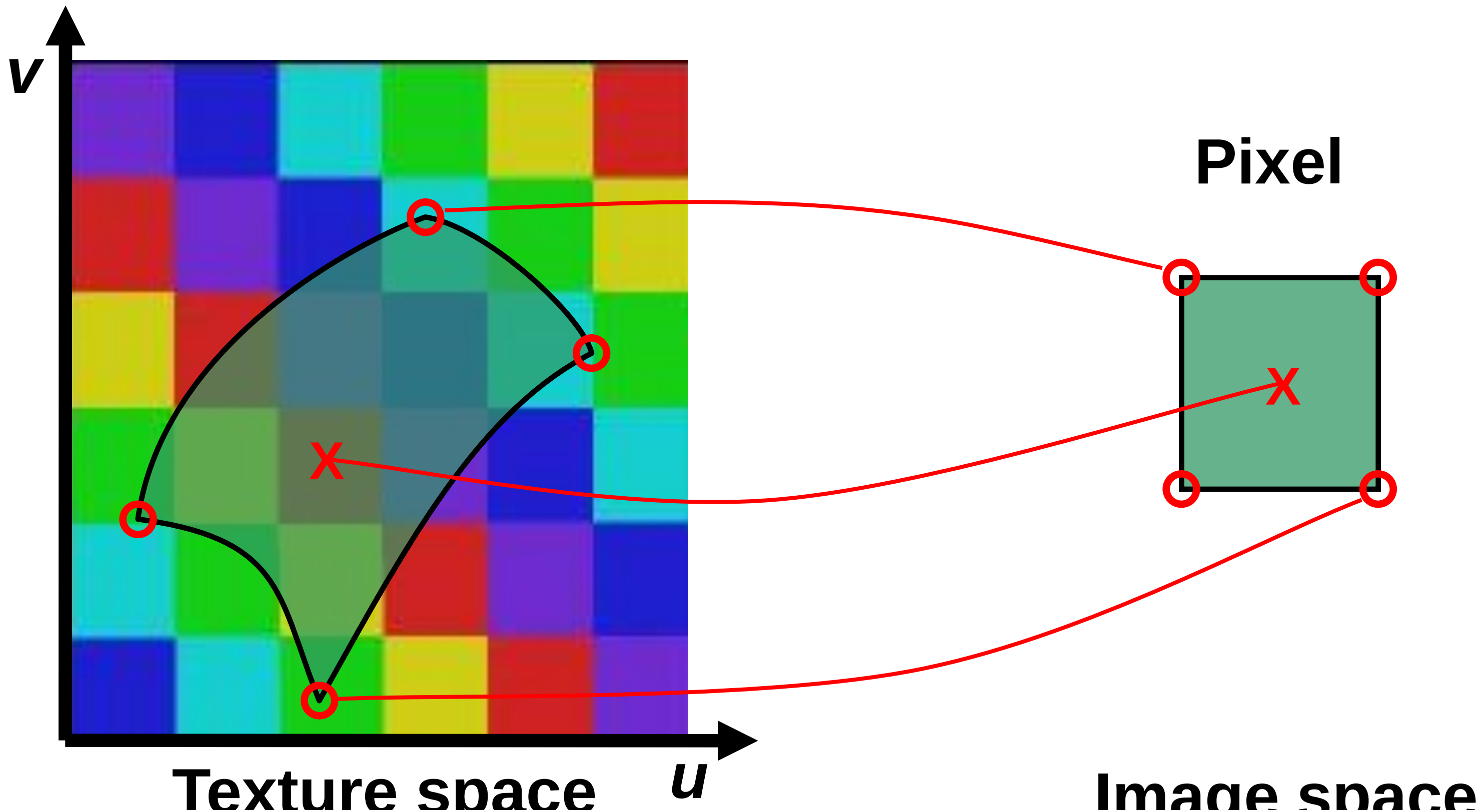
Texture Aliasing

- Caused by *undersampling*: texture information is lost



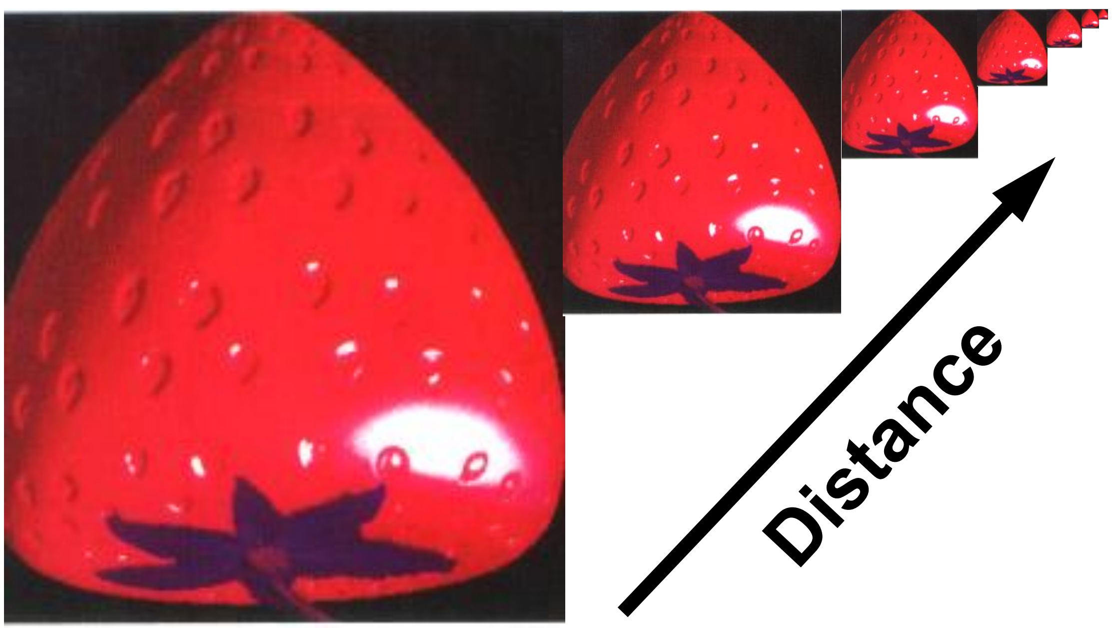
Texture Anti-Aliasing

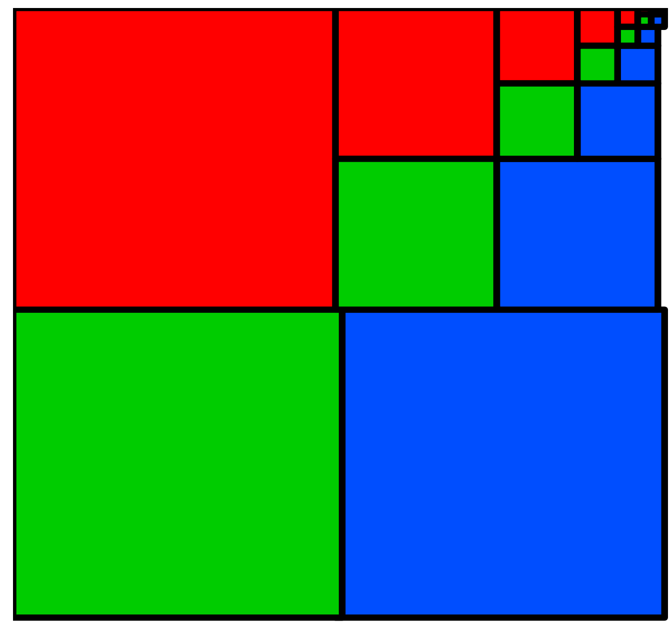
- A good pixel value is the weighted mean of the pixel area projected into texture space



Texture Anti-Aliasing: MIP Mapping

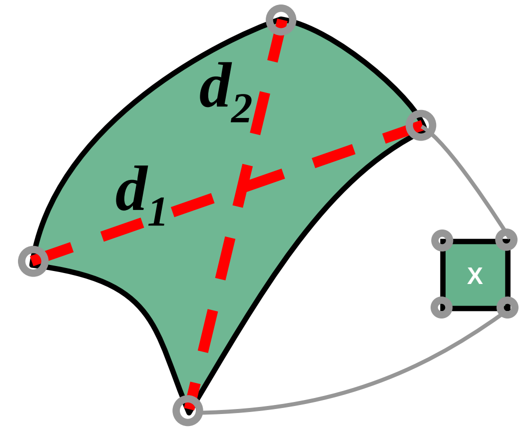
- MIP Mapping ("Multum In Parvo")
  - Texture size is reduced by factors of 2 (*downsampling* = "much info on a small area")
  - Simple (4 pixel average) and memory efficient
  - Last image is only ONE texel

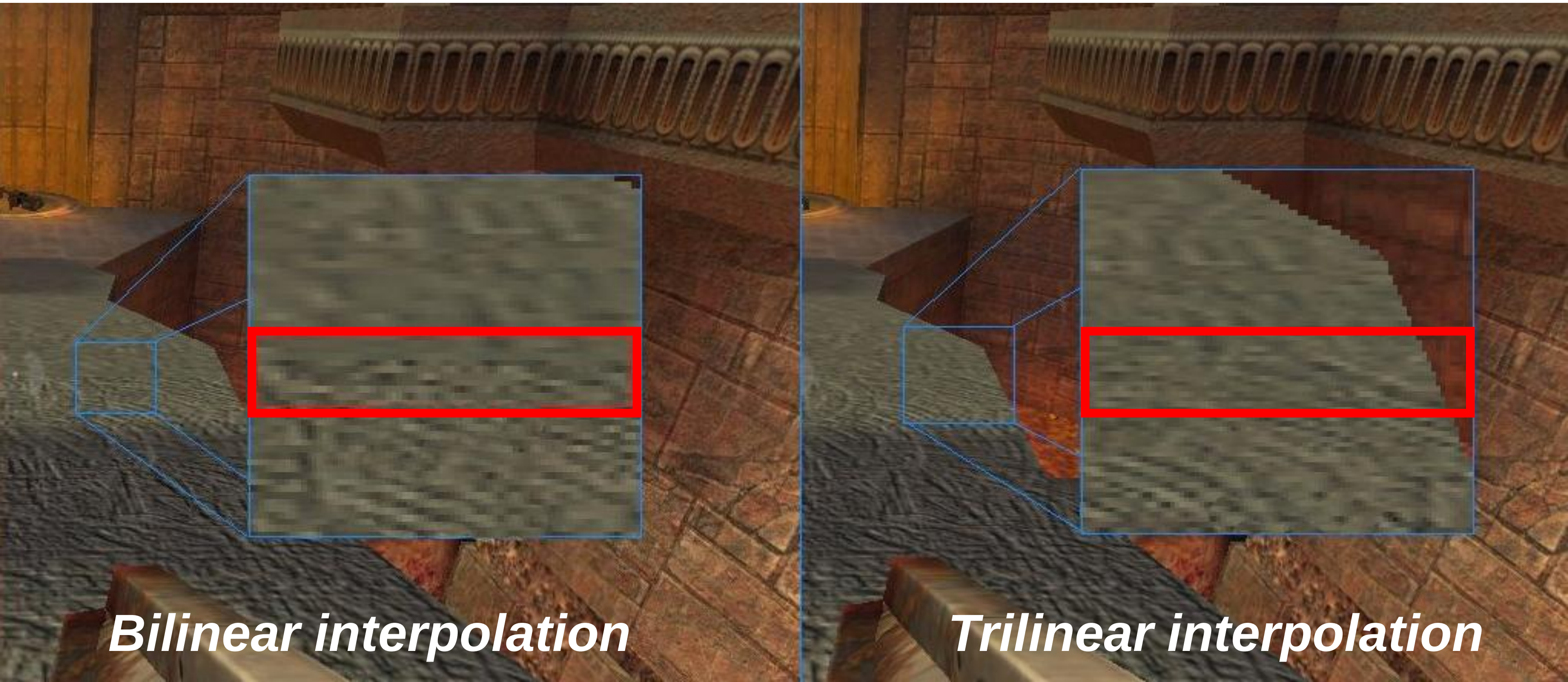




Texture Anti-Aliasing: MIP Mapping

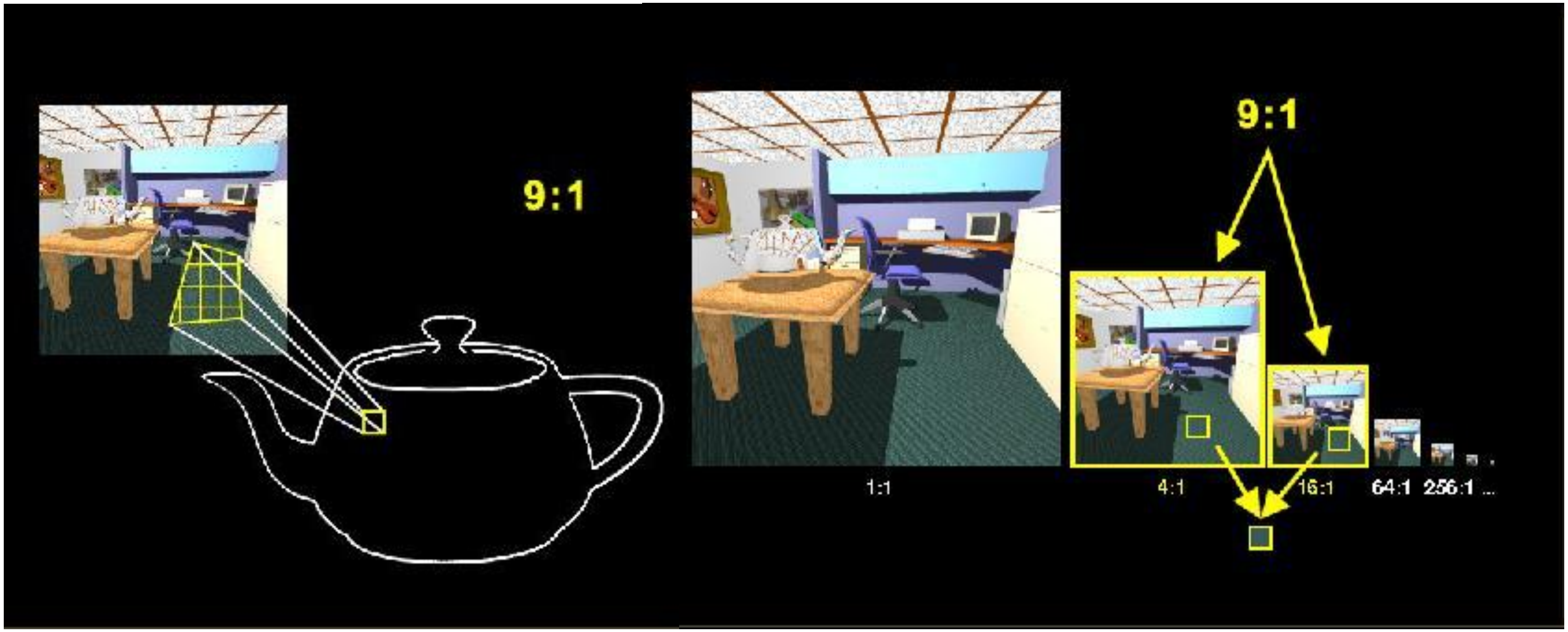
- MIP Mapping Algorithm
- $D := ld(max(d_1,d_2))$
- $T_0 :=$ value from texture $D_0 = trunc\ (D)$ "Mip Map level"
  - Use *bilinear interpolation*





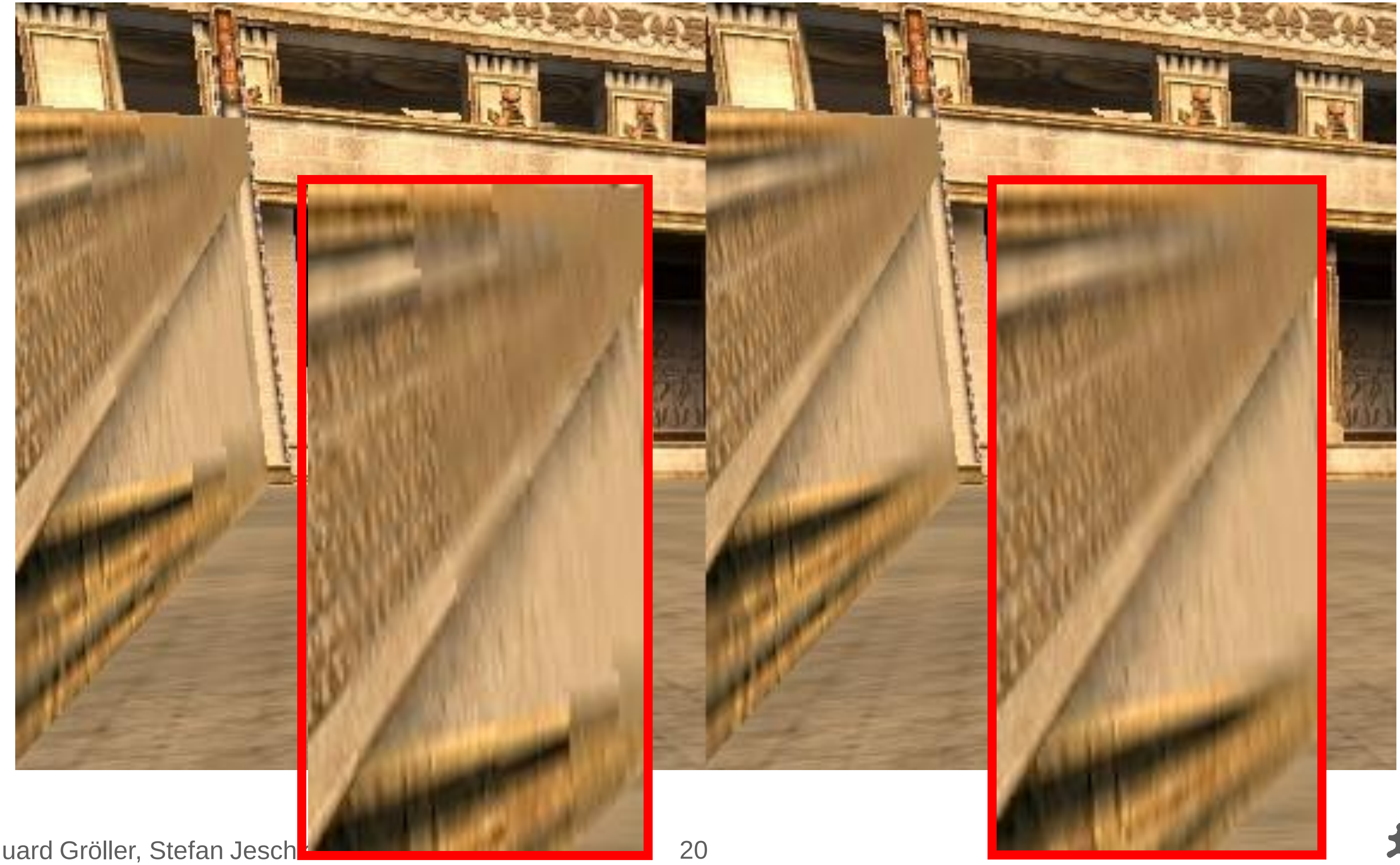
Texture Anti-Aliasing: MIP Mapping

- 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)



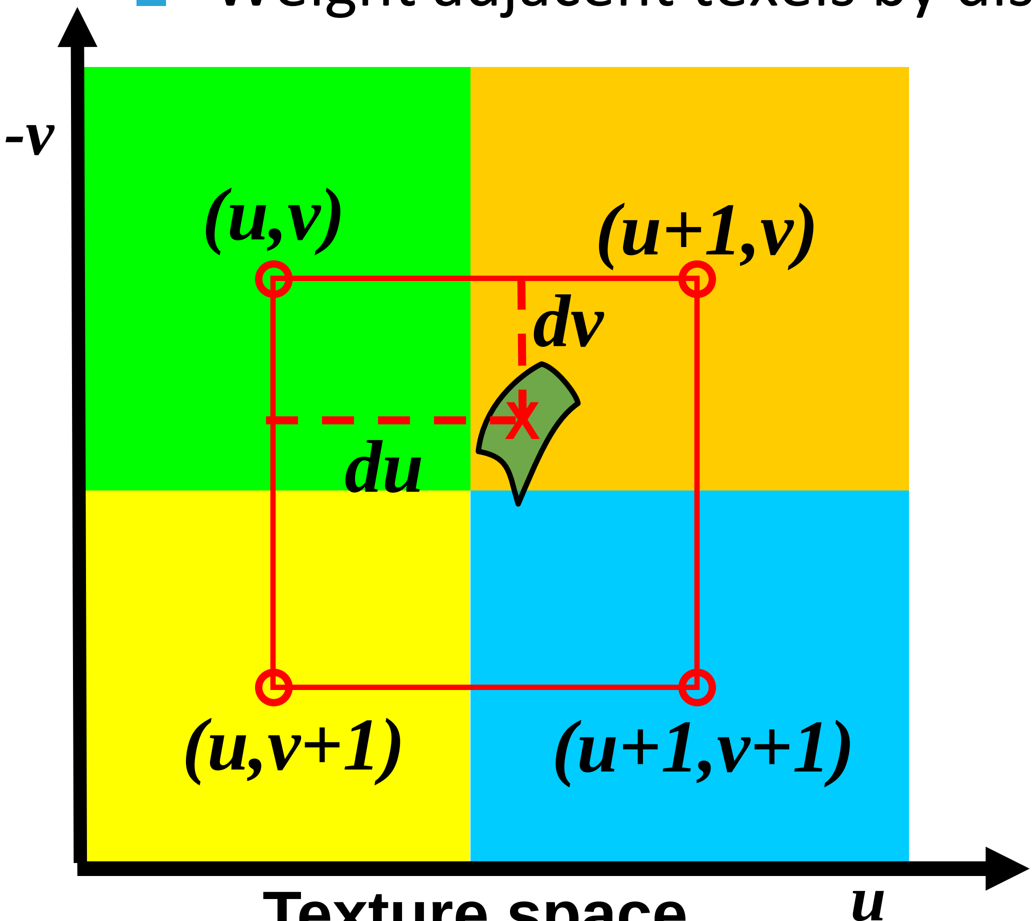
Texture Anti-Aliasing: Mip Mapping

- Other example for bilinear vs. trilinear filtering



Texture Anti-Aliasing

- Bilinear reconstruction for texture magnification ($D<0$) (*"upsampling"*)
  - Weight adjacent texels by distance to pixel position



$$
\begin{aligned}
&T(u+du,v+dv)\\
&= du\cdot dv\cdot T(u+1,v+1)\\
&+ du\cdot(1-dv)\cdot T(u+1,v)\\
&+ (1\text{-}du)\cdot dv\cdot T(u,v+1)\\
&+ (1\text{-}du)\cdot(1\text{-}dv)\cdot T(u,v)
\end{aligned}
$$

Anti-Aliasing (Bilinear Filtering Example)

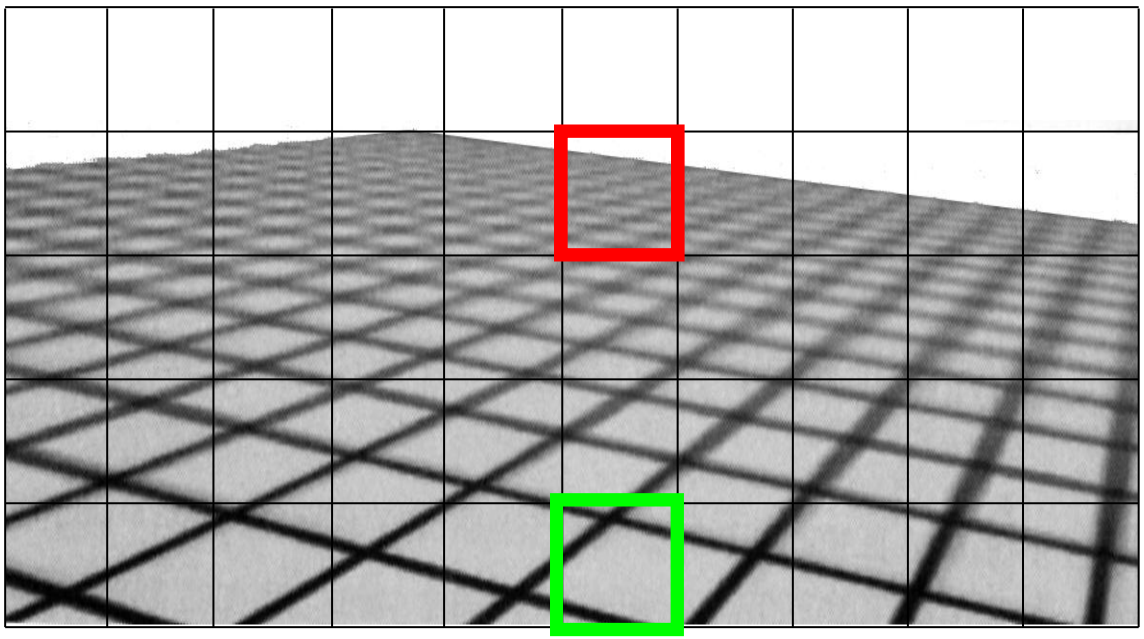
Original image

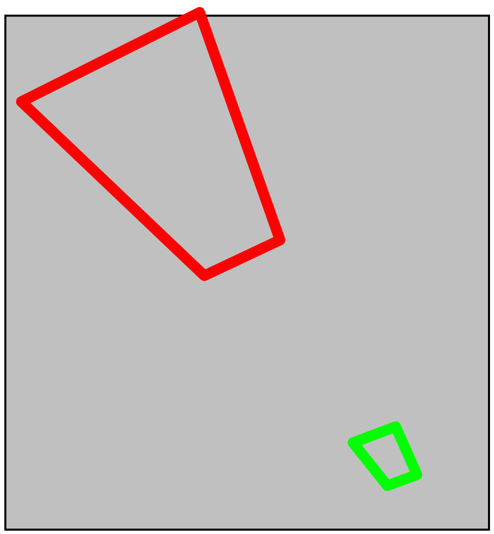
Nearest neighbor

Bilinear filtering

Anti-Aliasing: Anisotropic Filtering

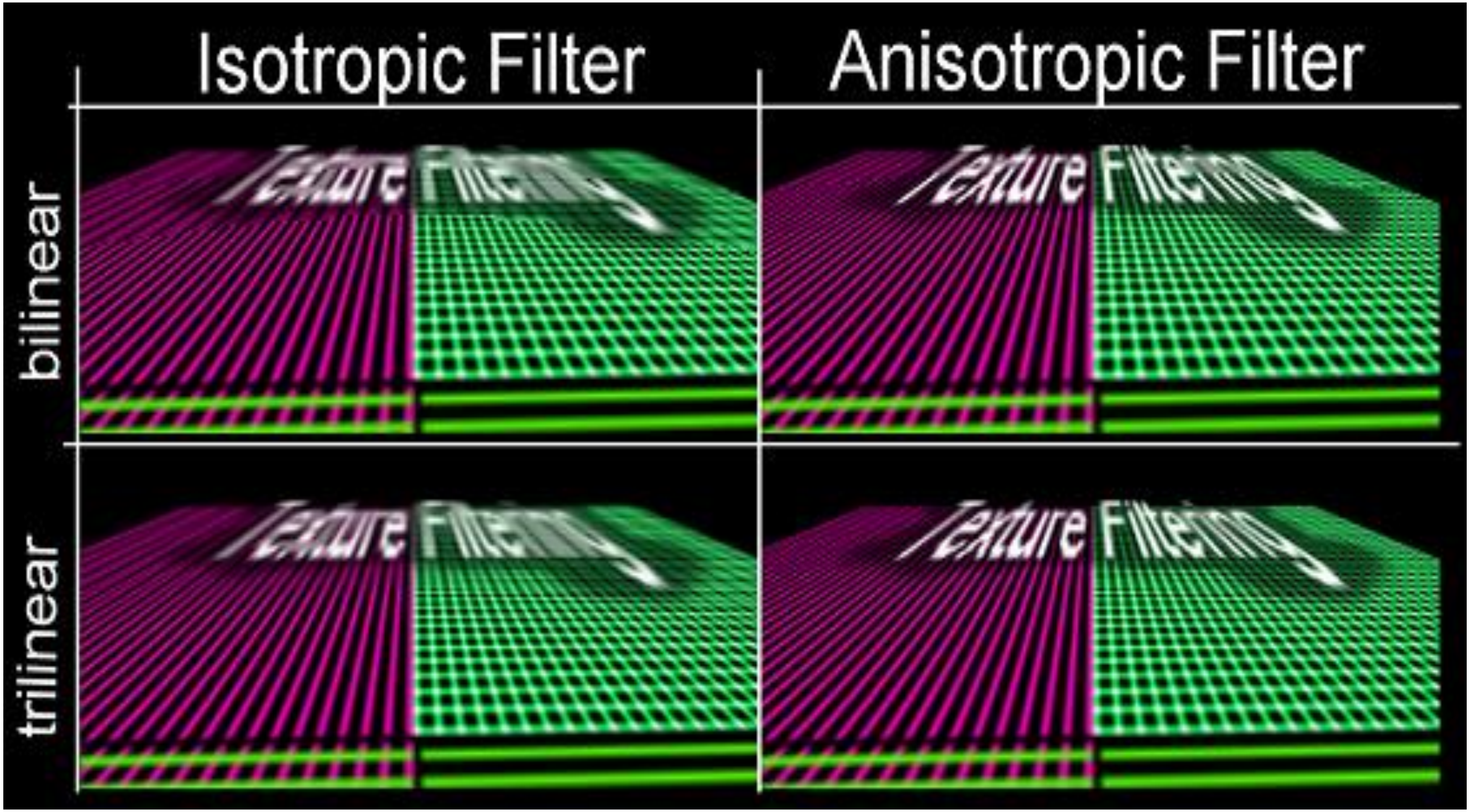
- Anisotropic Filtering
  - View dependent filter kernel
  - Implementation: *summed area table*, "*RIP Mapping*", "*footprint assembly*" , “***sampling***”





Texture space

- Example


Isotropic Filter
Anisotropic Filter
bilinear
trilinear
Texture Filtering

Texture Anti-aliasing

- Everything is done in hardware, nothing much to do!
- `gluBuild2DMipmaps()` generates MIPmaps
- Set parameters in `glTexParameter()`
  - `GL_LINEAR_MIPMAP_NEAREST`
  - `GL_TEXTURE_MAG_FILTER`
- Anisotropic filtering is an extension:
  - `GL_EXT_texture_filter_anisotropic`
  - Number of samples can be varied (4x,8x,16x)
    - Vendor specific support and extensions

Signal Theory

- Fourier Transform of signal → frequency space („spectrum“)
- Multiplication (mul) in primary space = Convolution (conv) in frequency space
- Typical signals and their spectra:
  - Box <-> sin(x)/x (=„sinc“)
  - Gaussian <-> Gaussian
  - Impulse train <-> Impulse train
  - Width inverse proportional!

CG Signal Pipeline: Overview

- Initial Sampling
- Resampling
- Display

CG Signal Pipeline: Initial Sampling

- Input: continuous signal
  - Nature or computer generated
- Bandlimiting: remove high frequencies
  - conv sinc <-> mul box
  - Happens in camera optics, lens of eye, or antialiasing (direct convolution, supersampling)
- Sampling:
  - mul impulse train <-> conv impulse train
  - Leads to replica of spectra!
- Result: image or texture

CG Signal Pipeline: Resampling

- Input: Samples = discrete signal (usually texture)
- Reconstruction:
  - conv sinc <-> mul box
  - „Removes“ replica of spectrum in sampled repr.
- Bandlimiting:
  - Only required if new sampling frequency is lower!
  - Typically through mipmapping
- Sampling
- Result: another texture or final image (=frame buffer)

CG Signal Pipeline: Display

- Input: Samples (from frame buffer)
- Reconstruction
  - Using display technology (e.g. CRT: Gaussian!)
- Result: continuous signal (going to eye)

CG Signal Pipeline: Observations

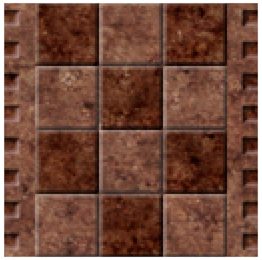
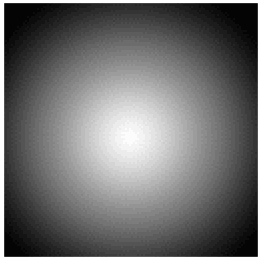
- Practice: substitute sinc by Gaussian
  - sinc has negative values
  - Gaussian can be cut off gracefully
- „Reconstruction“ is really an interpolation!
  - Reconstruction ≠ Antialiasing!
- Aliasing: overlap of signal replica in sampling
  - Bandlimiting = Antialiasing
- Magnification → reconstruction only
- Minification → bandlimiting + reconstruction

CG Signal Pipeline: Full Scene Antialiasing

- Supersamling
- Multisampling (MSAA): combines
  - Supersampling (for edges)
  - Texture filtering (for textures)
  - Only one shader evaluation per final pixel
- Morphological Antialiasing (FXAA, SMAA, ...):
  - Postprocess
  - Analyzes image, recovers edges, antialiases them

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


Multitexture – How?

- Simple(!) texture environment example:

```
glActiveTexture(GL_TEXTURE1);
glTexEnvi(GL_TEXTURE_ENV, …)
…     GL_TEXTURE_ENV_MODE, GL_COMBINE);
…     GL_COMBINE_RGB, GL_MODULATE);
…     GL_SOURCE1_RGB, GL_TEXTURE);
…     GL_OPERAND1_RGB, GL_SRC_COLOR);
…     GL_SOURCE2_RGB, GL_PREVIOUS);
…     GL_OPERAND2_RGB, GL_SRC_COLOR);
```

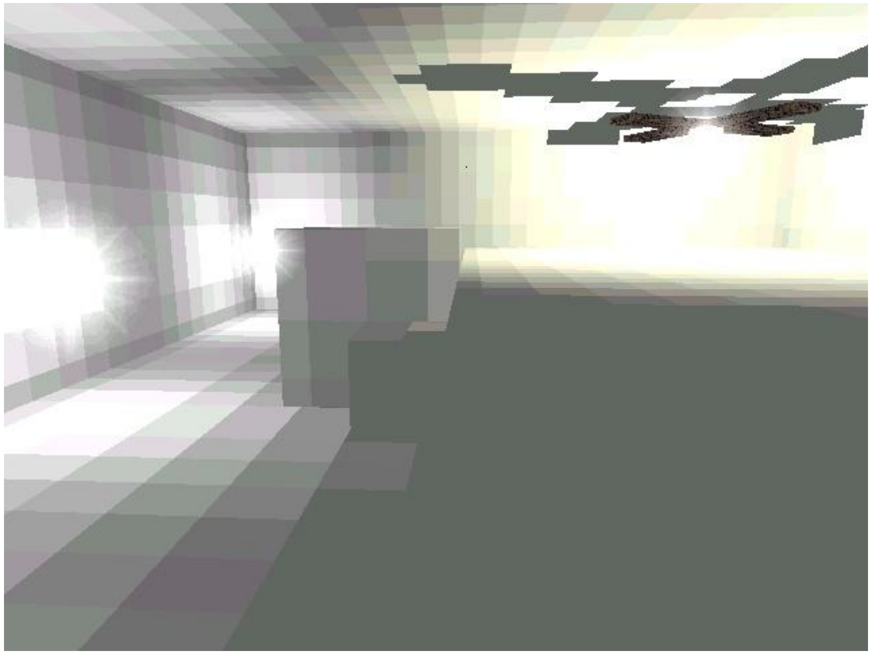
$$C = CT_1 \cdot CT_0$$

- Programmable shading makes this easier!

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

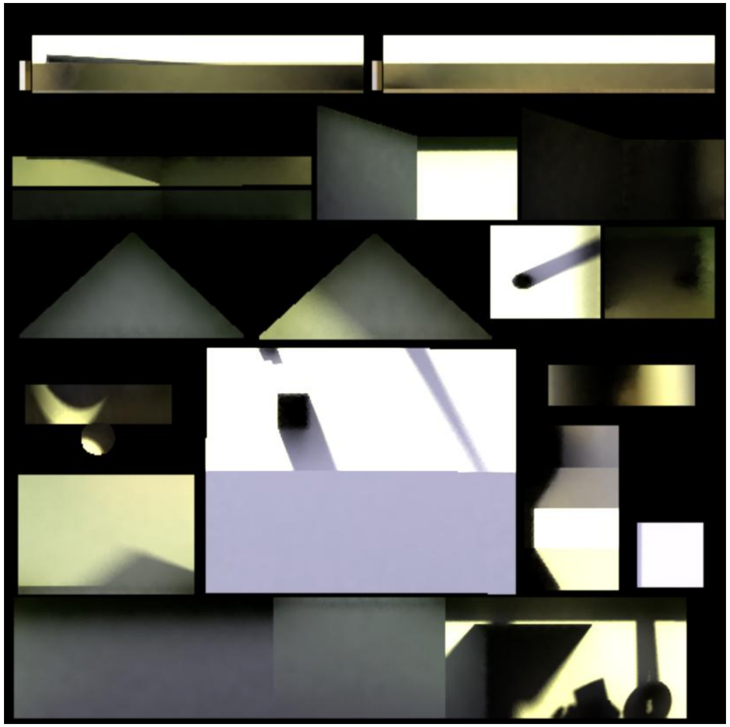
Light Mapping

Original scene Light-mapped

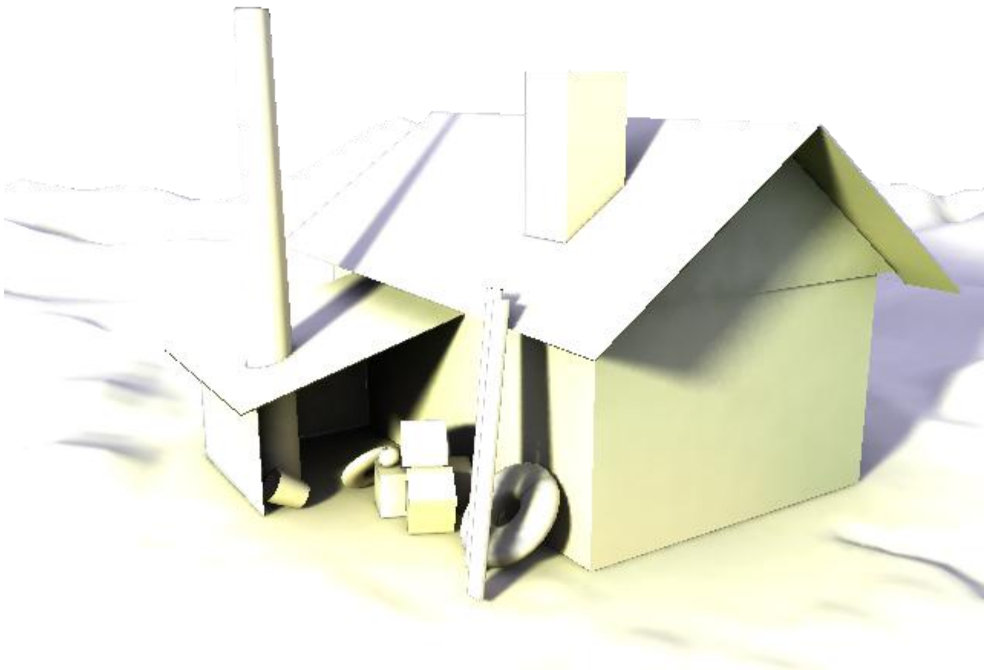
Example: Light Mapping

- Precomputation based on non-realtime methods
  - Radiosity
  - Raytracing
    - Monte Carlo Integration
    - Pathtracing
    - Photonmapping

Light Mapping



Lightmap



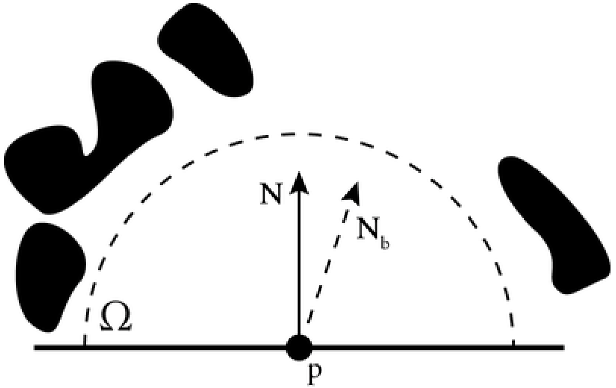
mapped

Light Mapping

Original scene　　　　Light-mapped

Ambient Occlusion

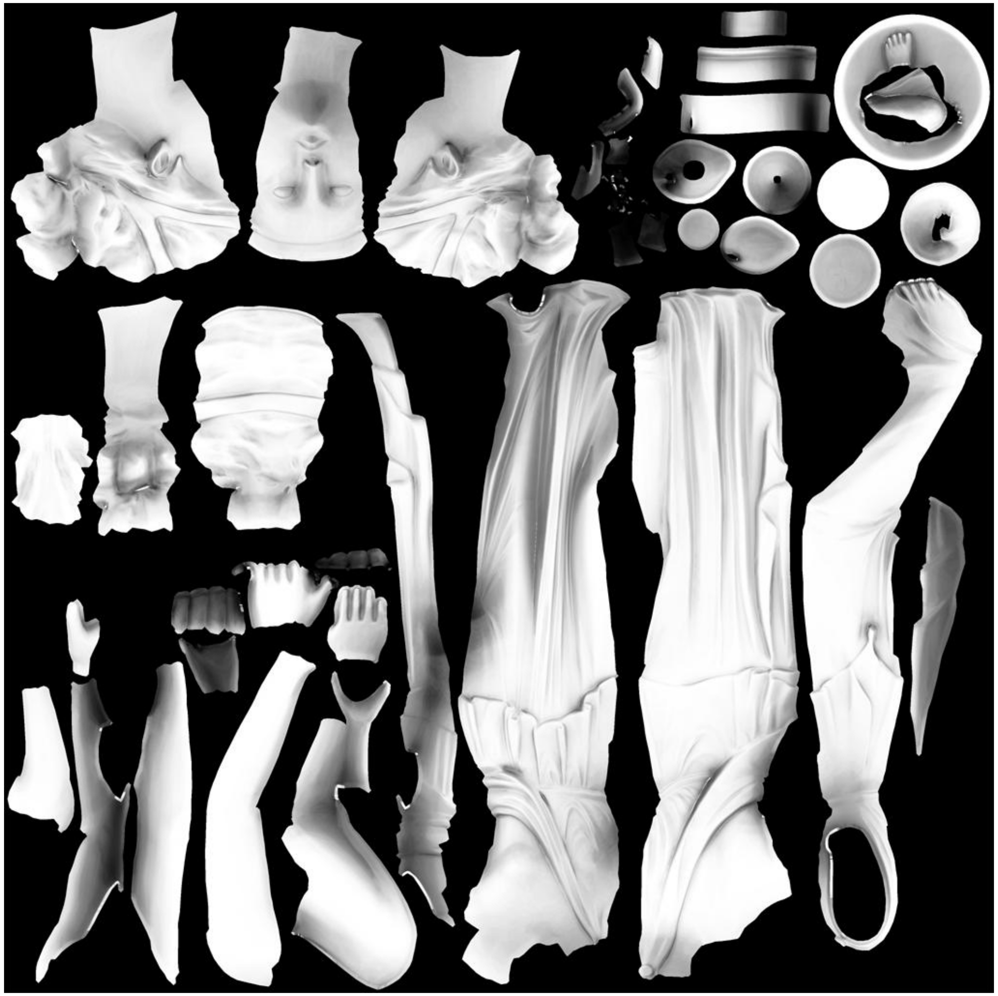
- Special case of light mapping
- Cos-weighted visibility to environment modulates intensity:



$$A_p = \frac{1}{\pi} \int_\Omega V_{p,\omega} (N \cdot \omega)\, d\omega$$

- Darker where more occluded
- „Soft shadow due to diffuse sky“
- Option: “per object” lightmap
  - Allows to move object

Ambient Occlusion



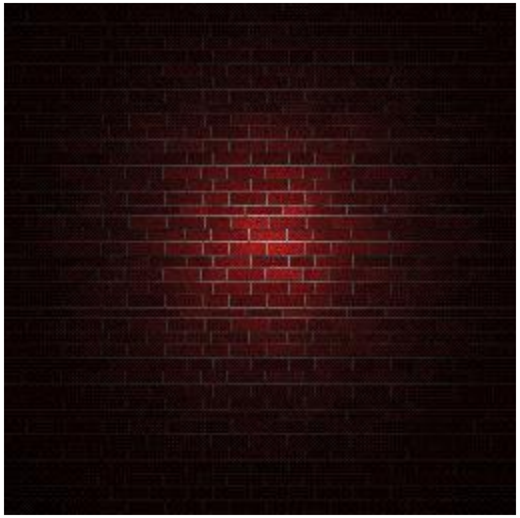
Model/Texture: Rendermonkey

Light Mapping Issues

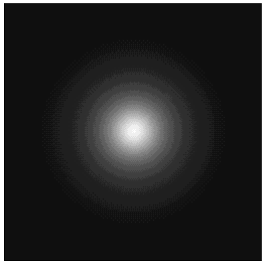
- Map generation:
  - Use single map for group of coplanar polys
    - Lightmap UV coordinates need to be in (0..1)x(0..1)
- Map application:
  - Premultiply textures by light maps
    - Why is this not appealing?
  - Multipass with framebuffer blend
    - Problems with specular
  - Multitexture
    - Fast, flexible

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/AO Toolset

- DCC programs (*Blender*, Maya…)
- Game Engines (Irrlicht)
- Light Map Maker (free)

- Ambient Occlusion:
  - xNormal

Texture Coordinates

- Specified manually (gl*Multi*TexCoord())
- Using classical OpenGL texture coordinate generation
  - Linear: from object or eye space vertex coords
  - Special texturing modes (env-maps)
  - Can be further modified with texture matrix
    - E.g., to add texture animation
  - Can use 3rd or 4th texture coordinate for projective texturing!
- Shader allows complex texture lookups!

Texture Coordinate Generation

- Specify a “plane” (i.e., a 4D-vector) for each coordinate (s,t,r,q)
- Example: $s = p_1\, x + p_2\, y + p_3\, z + p_4\, w$

```
GLfloat Splane[4] = { p1, p2, p3, p4 };
glTexGenfv(GL_S, GL_EYE_PLANE, Splane);
glEnable(GL_TEXTURE_GEN_S);
```

- Think of this as a matrix T with plane parameters as row vectors

Texture Coordinate Generation

- Object-linear:

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \mathbf{T} \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}_{object}$$

- Eye-linear:
  $T_e = T \cdot M^{-1}$
  (M…Modelview matrix at time of specification!)
- Effect: uses coordinate space at time of specification!
  - Eye: M=identity
  - World: M=view-matrix

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \mathbf{T}_e \begin{bmatrix} x \\ y \\ z \\ w \end{bmatrix}_{eye}$$

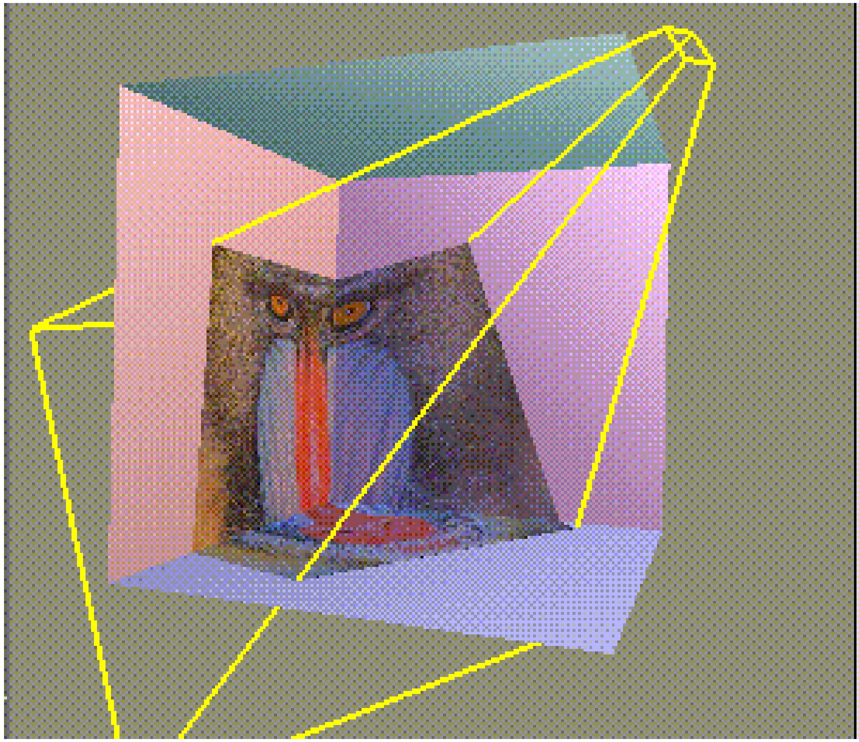
Texture Animation

- Classic OpenGL
  - Can specify an arbitrary 4x4 Matrix, each frame!
  - `glMatrixMode(GL_TEXTURE);`
  - There is also a texture matrix stack!
- Shaders allow arbitrary dynamic calculations with uv-coordinates
  - Many effects possible:
  - Flowing water, conveyor belts, distortions etc.

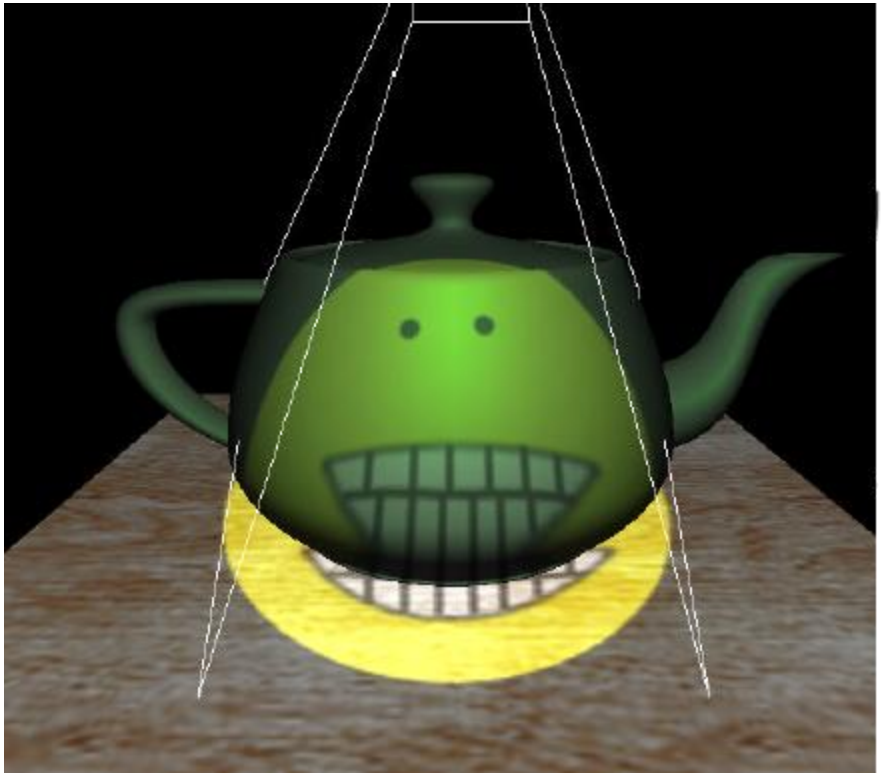
Projective Texturing

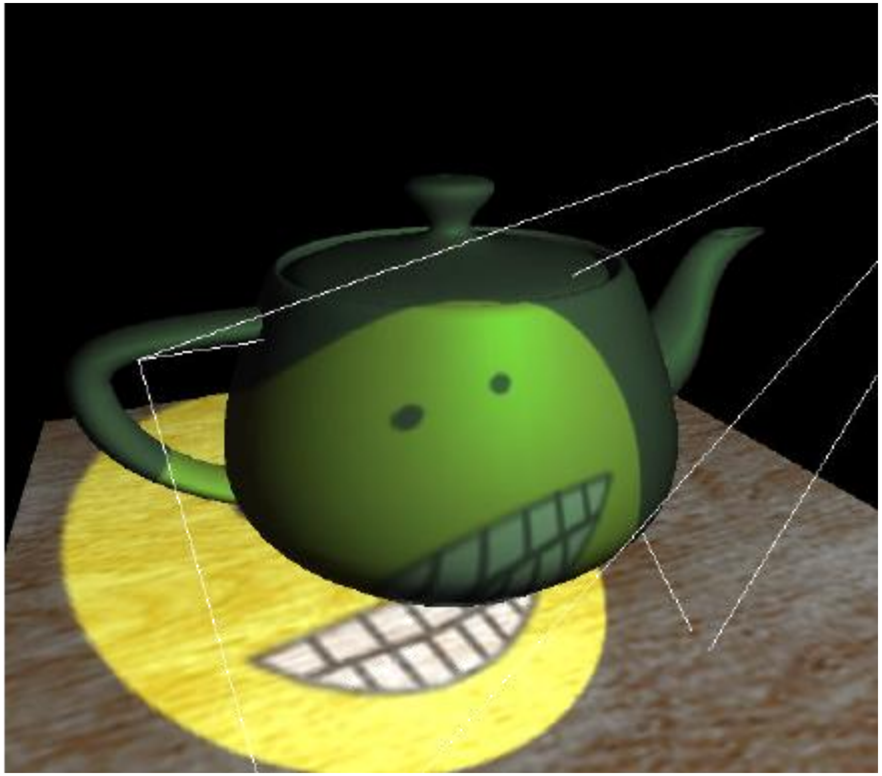
Projective Texture Mapping

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



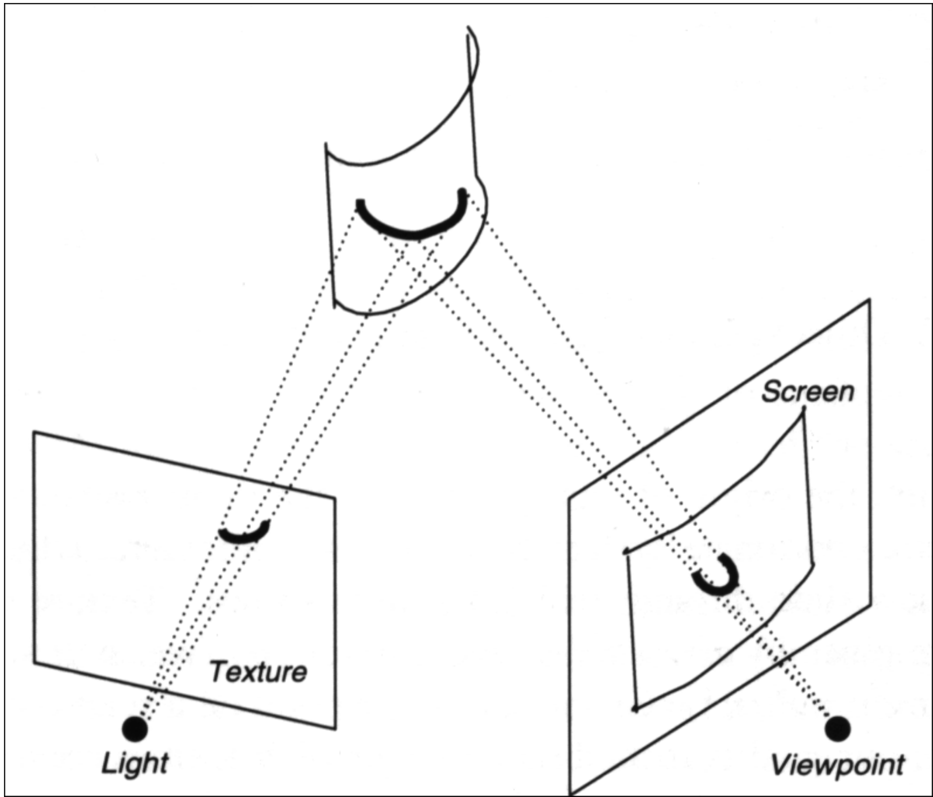
Projective Texture Mapping



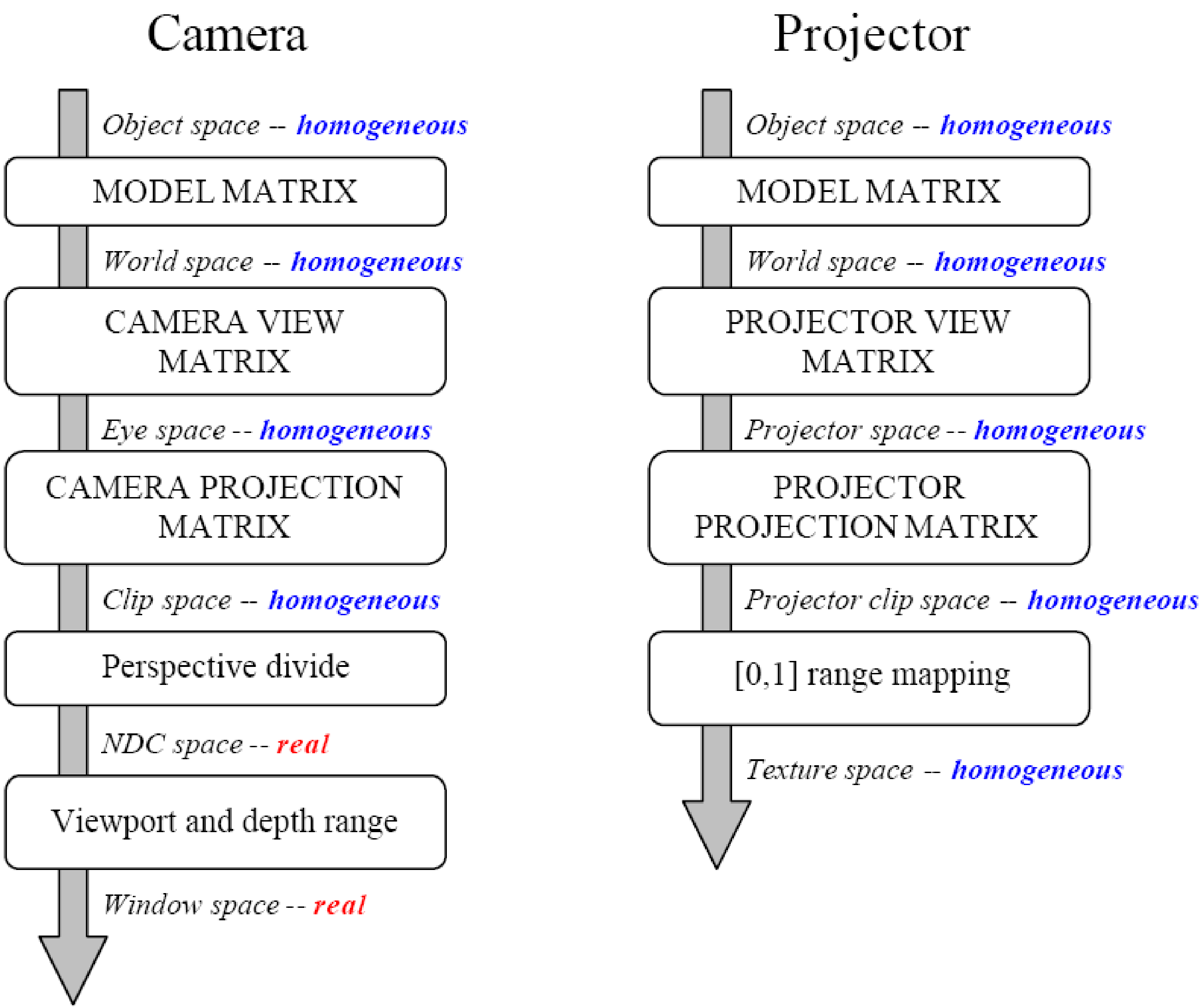


Projective Texture Mapping: Vertex Stage

- Map vertices to light frustum
  - Option 1: from object space
  - Option 2: from eye space
- Projection (perspective transform)



Spaces


Camera
Object space -- homogeneous
MODEL MATRIX
World space -- homogeneous
CAMERA VIEW MATRIX
Eye space -- homogeneous
CAMERA PROJECTION MATRIX
Clip space -- homogeneous
Perspective divide
NDC space -- real
Viewport and depth range
Window space -- real
Projector
Object space -- homogeneous
MODEL MATRIX
World space -- homogeneous
PROJECTOR VIEW MATRIX
Projector space -- homogeneous
PROJECTOR PROJECTION MATRIX
Projector clip space -- homogeneous
[0,1] range mapping
Texture space -- homogeneous

Projective Texture Mapping

- OpenGL does not store Modeling Matrix
- No notion of world space!

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} = \overbrace{\begin{bmatrix} \text{Camera} \\ \text{view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Modeling} \\ \text{matrix} \end{bmatrix}}^{\text{Modelview}} \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}$$

Camera Space

Object Space

Projective Texture Mapping

- Version 1: transforming object space coordinates
  - Disadvantage: need to provide model matrix for each object in shader!
  - Classic OpenGL: even more difficult!

$$\begin{bmatrix} s\\ t\\ r\\ q \end{bmatrix} = \underbrace{\begin{bmatrix} 1/2 & & & 1/2\\ & 1/2 & & 1/2\\ & & 1/2 & 1/2\\ & & & 1 \end{bmatrix} \begin{bmatrix} \text{Light} \\ \text{(projection)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Light} \\ \text{view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Modeling} \\ \text{matrix} \end{bmatrix}}_{T} \begin{bmatrix} x_o\\ y_o\\ z_o\\ w_o \end{bmatrix}$$

Map [-1..1] to [0..1]

Projective Texture Mapping

- Version 2: transforming eye space coordinates
  - Advantage: matrix works for all objects!

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \underbrace{\begin{bmatrix} 1/2 & & & 1/2 \\ & 1/2 & & 1/2 \\ & & 1/2 & 1/2 \\ & & & 1 \end{bmatrix} \begin{bmatrix} \text{Light} \\ \text{(projection)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} \text{Light} \\ \text{view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix}}_{T} \begin{bmatrix} \text{Inverse} \\ \text{eye} \\ \text{view} \\ \text{(look at)} \\ \text{matrix} \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$

Classic OpenGL TexGen Transform

$$\begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix} = \overbrace{\begin{bmatrix} \textit{Eye} \\ \textit{view} \\ \textit{(look at)} \\ \textit{matrix} \end{bmatrix} \begin{bmatrix} \textit{Modeling} \\ \textit{matrix} \end{bmatrix}}^{\textbf{Modelview}} \begin{bmatrix} x_o \\ y_o \\ z_o \\ w_o \end{bmatrix}$$

Automatically applied by TexGen (set Modeling matrix to eyeview)

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \underbrace{\begin{bmatrix} 1/2 & & & 1/2 \\ & 1/2 & & 1/2 \\ & & 1/2 & 1/2 \\ & & & 1 \end{bmatrix} \begin{bmatrix} \textit{Light} \\ \textit{frustum} \\ \textit{(projection)} \\ \textit{matrix} \end{bmatrix} \begin{bmatrix} \textit{Light} \\ \textit{view} \\ \textit{(look at)} \\ \textit{matrix} \end{bmatrix}} \begin{bmatrix} \textit{Inverse} \\ \textit{eye} \\ \textit{view} \\ \textit{(look at)} \\ \textit{matrix} \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$

Supply this combined transform to *glTexGen*

Projective Texture Mapping: Rasterization

- Problem: texture coordinate interpolation
  - Texture coordinates are homogeneous!
- Look at perspective correct texturing first!

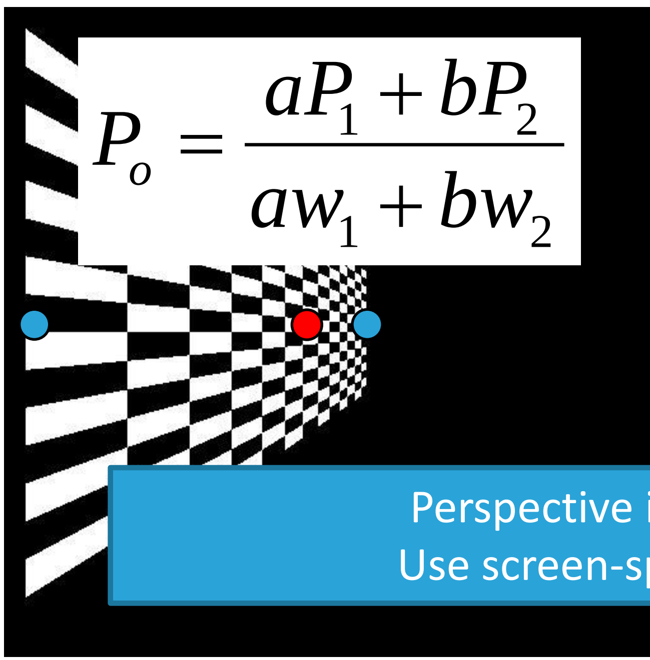
Perspective Texture Mapping

- Problem: linear interpolation in rasterization?

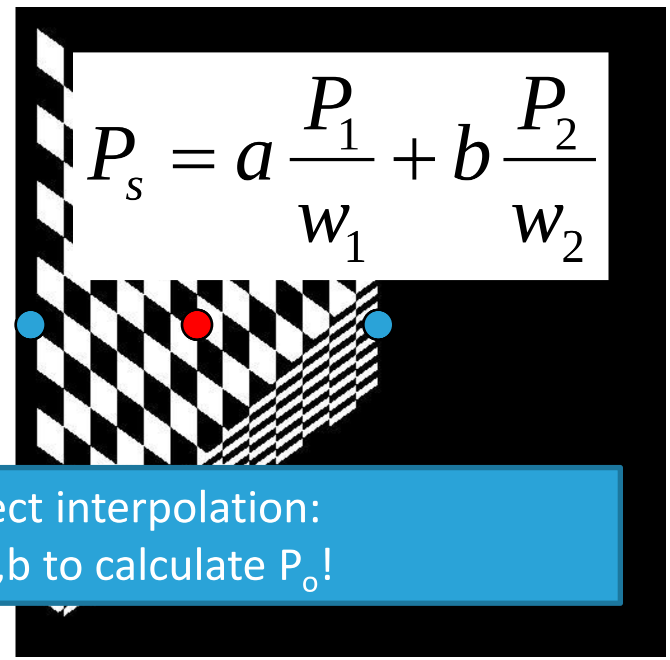
objectspace interpolation

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

screenspace interpolation



$$P_o = \frac{aP_1 + bP_2}{aw_1 + bw_2}$$



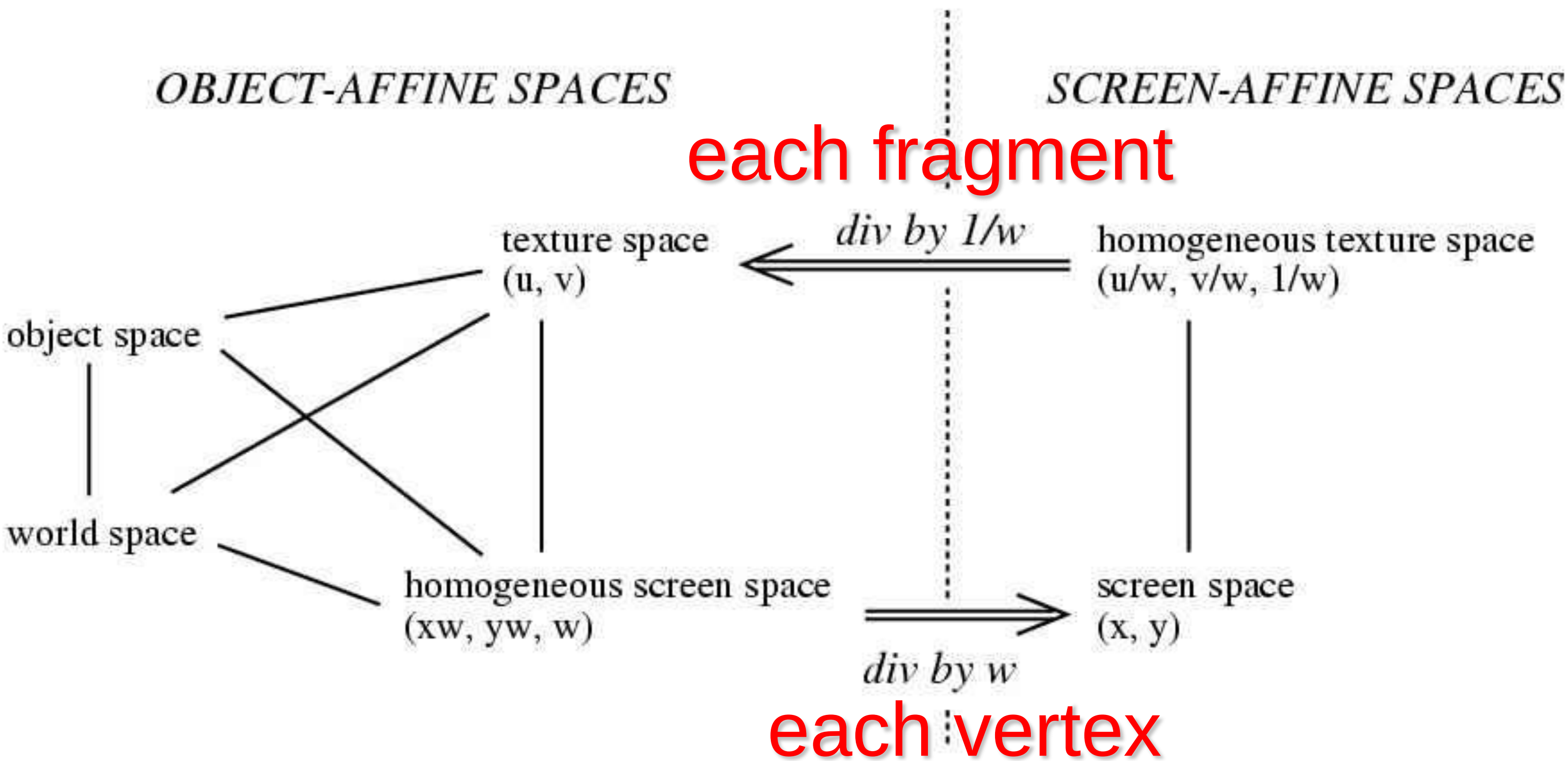
$$P_s = a\frac{P_1}{w_1} + b\frac{P_2}{w_2}$$

Perspective incorrect interpolation:
Use screen-space a,b to calculate P_o!

$$a = b = 0{,}5;\ P = (x, y, z, w, 1, u, v, \ldots)$$

Perspective Texture Mapping

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



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!

Homogeneous Perspective Correct Interpolation

- [x,y,z,1,r,g,b,a]
- texcoord generation → [x,y,z,1, r,g,b,a, s,t,r,q]
- Modelviewprojection → [x',y',z',w,1, r,g,b,a, s,t,r,q]
- Project (/w) →

  $[x'/w, y'/w, z'/w, 1/w, r,g,b,a, s/w, t/w, r/w, q/w]^{vert}$
- Rasterize and interpolate →

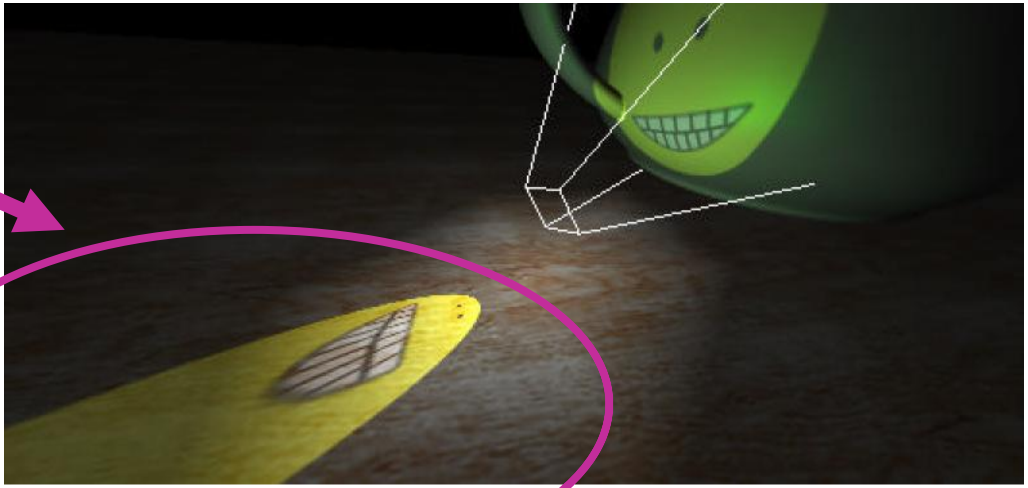
  $[x'/w, y'/w, z'/w, 1/w, r,g,b,a, s/w, t/w, r/w, q/w]^{frag}$
- Homogeneous: → texture project (/ q/w) →

  [x'/w,y'/w,z'/w,1/w, r,g,b,a, s/q,t/q,r/q,1]
- Or non-homogeneous: → standard project (/ 1/w) →

  [x'/w, y'/w, z'/w, 1/w, r,g,b,a, s,t,r,q] (for normals)

Projective Texture Mapping

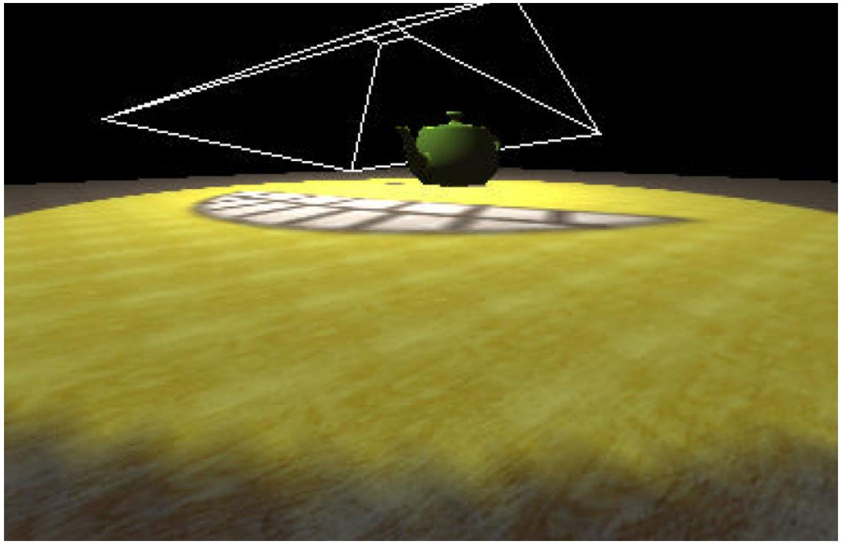
- Problem
  - reverse projection
- Solutions
  - Cull objects behind projector
  - Use clip planes to eliminate objects behind projector
  - Fold the back-projection factor into a 3D attenuation texture
  - Use to fragment program to check q < 0



Projective Texture Mapping

- Problems
  - Resolution problems
  - Projection behind shadow casters

  → Shadow Mapping!



Projective Texture Mapping Example

- Example shown in CG Shading Language
  - CG is proprietary to NVIDIA
  - C-like synthax
  - HLSL (DirectX shading language) nearly the same synthax
- Shading languages have specialized calls for projective texturing:
  - CG/HLSL: `tex2Dproj`
  - GLSL: `texture2DProj`
  - They include perspective division

CG Vertex Program

```
Input:   float4 position,
         float3 normal
Output:  float4 oPosition,
         float4 texCoordProj,
         float4 diffuseLighting
Uniform: float Kd,
         float4x4 modelViewProj,
         float3 lightPosition,
         float4x4 textureMatrix
```

CG Vertex Program

```
oPosition =
    mul(modelViewProj, position);
texCoordProj =
    mul(textureMatrix, position);
float3 N = normalize(normal);
float3 L = normalize(lightPosition
             - position.xyz);
diffuseLighting =
    Kd * max(dot(N, L),0);
```

CG Fragment Program

```
Input:   float4 texCoordProj,
         float4 diffuseLighting
Output:  float4 color
Uniform: sampler2D projectiveMap
float4 textureColor =
  tex2Dproj(projectiveMap,
            texCoordProj);
color = textureColor *
        diffuseLighting;
```

CG vs. Classic OpenGL

- Classic OpenGL:
  - Just supply correct matrix to glTexGen

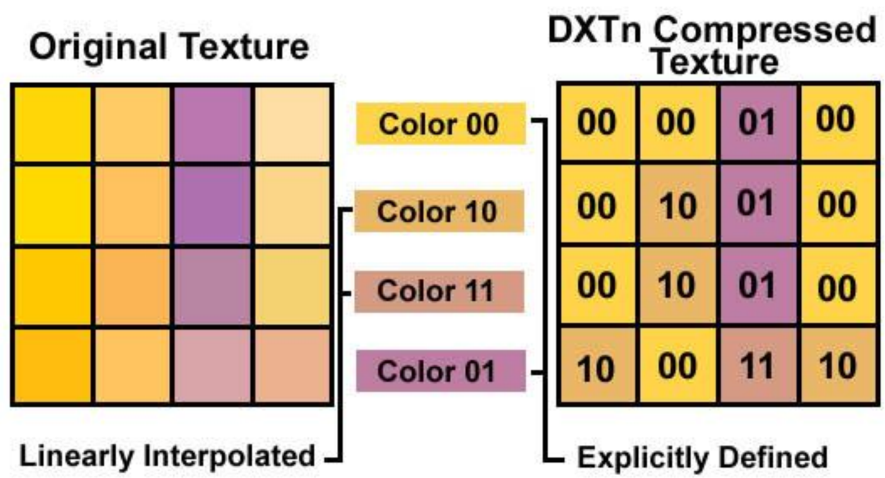
→ Projective texturing is easy to program and very effective method.

→ Combinable with shadows

Projective Shadow in Doom 3

Texture Compression

- S3TC texture compression (DXTn)
- Represent 4x4 texel block by two 16bit colors (5 red, 6 green, 5 blue)
- Store 2 bits per texel
- Uncompress
  - Create 2 additional Colors between c1 and c2
  - use 2 bits to index which color
- 4:1 or 6:1 compression



Multipass Rendering

Multipass Rendering

- Recall 80 million triangle scene
- Games are NOT using $a = 0.5$
  - at least not yet
- Assume $a = 32$, I = 1024x768, d=4
  - Typical for last generation games
  - F = I * d = 3,1 MF/frame,
  - T = F / a = 98304 T/frame
  - 60 Hz → **~189 MF/s, ~5,6 MT/s**

Do More!

- Hardware underused with standard OpenGL lighting and texturing

What can we do with this power?

- Render scene more often: **multipass rendering**
- Render more complex pixels: **multitexturing**
  - 2 textures are usually for free
- Render more complex pixels and triangles: **programmable shading**

Note

- Conventional OpenGL allows for many effects using multipass
  - Still in use for mobile devices and last gen consoles
  - Modern form: render to texture
    - Much more flexible but same principle

- Programmable shading makes things easier
  - Specialized calls in shading languages

Multipass Rendering: Why?

- OpenGL lighting model only
  - local
  - limited in complexity
- Many effects possible with multiple passes:
  - Dynamic environment maps
  - Dynamic shadow maps
  - Reflections/mirrors
  - Dynamic impostors
  - (Light maps)

Multipass Rendering: How?

- Render to auxiliary buffers, use result as texture
  - E.g.: environment maps, shadow maps
  - Requires pbuffer/fbo-support
- Redraw scene using fragment operations
  - E.g.: reflections, mirrors
  - Uses depth, stencil, alpha, … tests
- “Multitexture emulation mode”: redraw
  - Uses framebuffer blending
  - (light mapping)

Multipass Rendering: How?

(assume redraw scene…)

- First pass
  - Establishes z-buffer (and maybe stencil)
    `glDepthFunc(GL_LEQUAL);`
  - Usually diffuse lighting
- Second pass
  - Z-*Testing* only
    `glDepthFunc(GL_LEQUAL);`
  - Render special effect using (examples):
    - Blending
    - `glStencilFunc(GL_EQUAL, 1, 1);`

```
glEnable(GL_BLEND);
glBlendEquation(GL_FUNC_ADD);
```

weighting factors

result color

$$C = C_s\,S + C_d\,D$$

incoming (source) fragment color

framebuffer color

- Other equations: **`SUBTRACT`**, **`MIN`**, **`MAX`**

Multipass – Blending - Weights

```
glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA);
```

$$C = C_s \cdot \alpha + C_d \cdot (1- \alpha)$$

- Example: transparency blending (window)
- Weights can be defined almost arbitrarily
- Alpha and color weights can be defined separately
- `GL_ONE, GL_ZERO, GL_DST_COLOR, GL_SRC_COLOR, GL_ONE_MINUS_xxx`