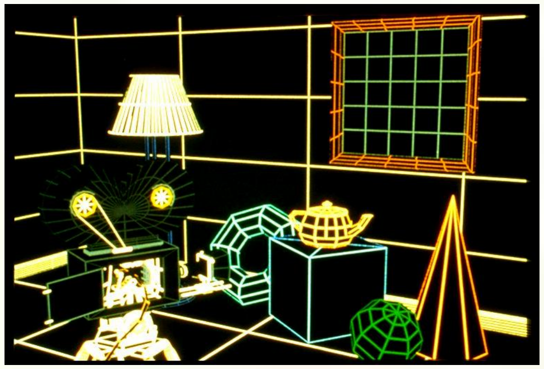
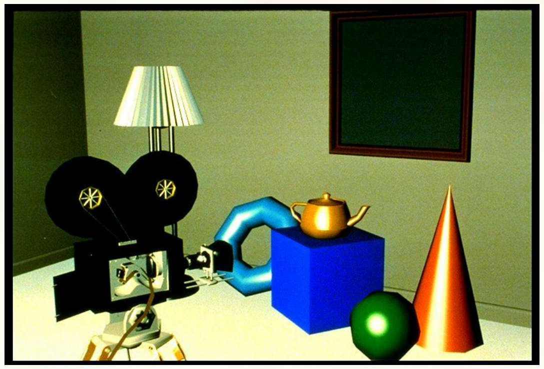
Texture Mapping

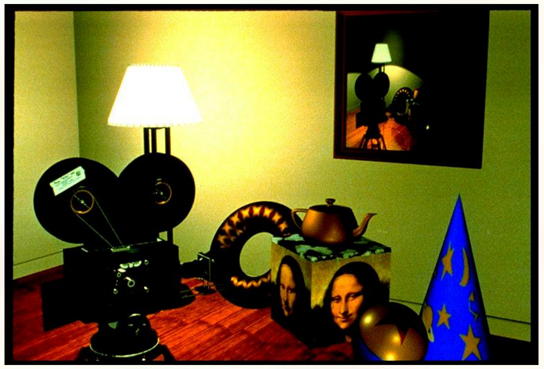
What adds visual realism?



Geometry only

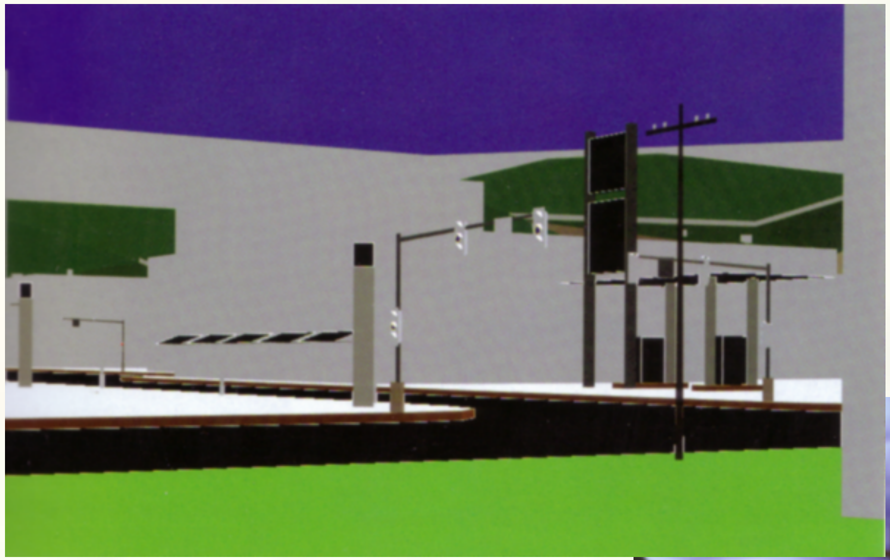


Phong shading

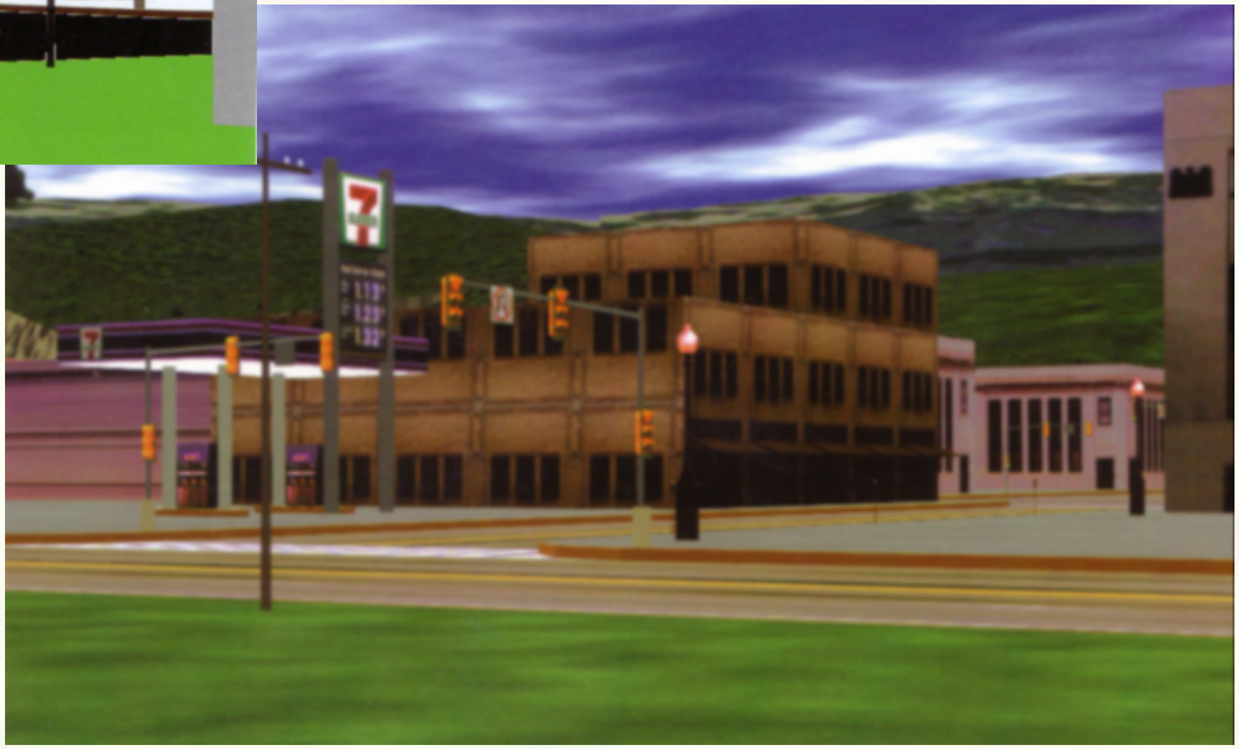


Phong shading + Texture maps

Textures Supply Surface Detail



Without texture



With texture

Textures Make Graphics Pretty

Sacred 2

Unreal Tournament

Microsoft Flight Simulator X

Texture → detail,
detail → immersion,
immersion → fun

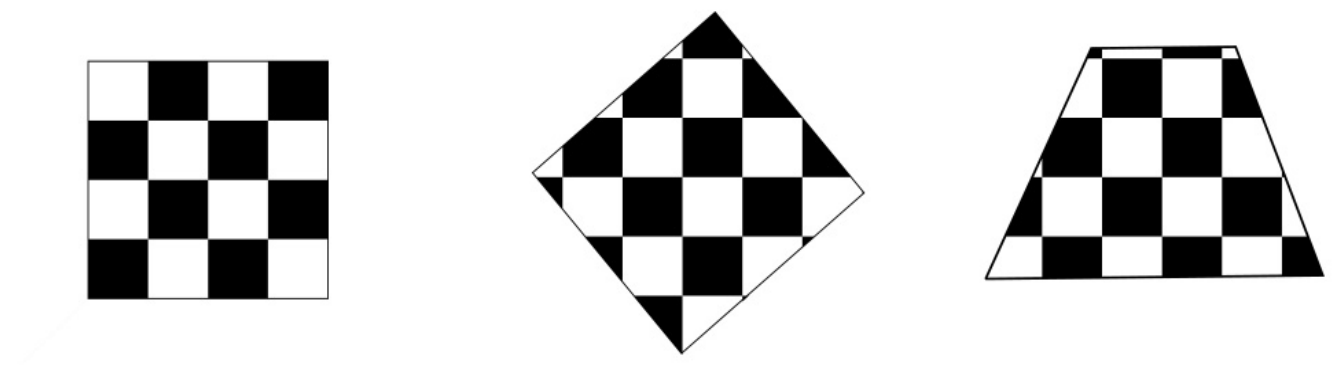
Texture mapping

IRIS Performer

Texture mapping (Woo et al., fig. 9-1)

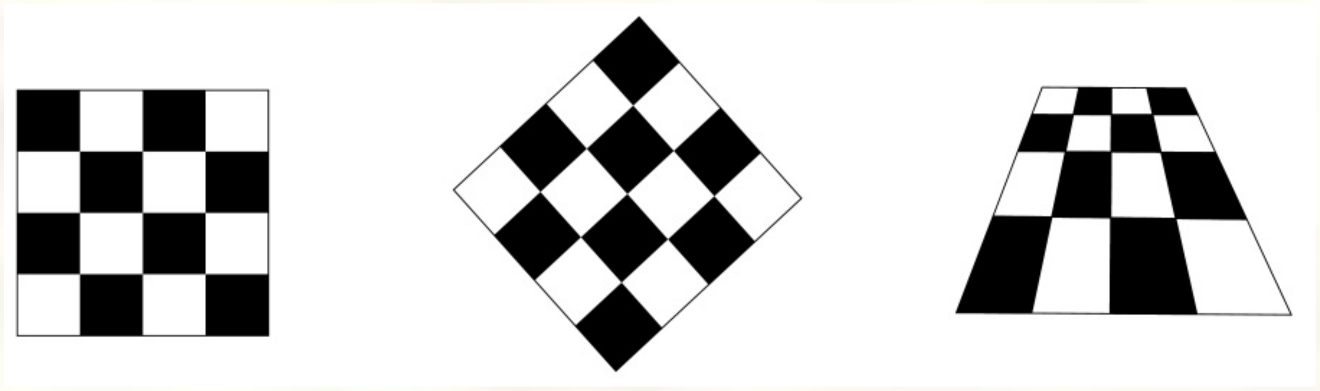
- Texture mapping allows you to take a simple polygon and give it the appearance of something much more complex.
  - Due to Ed Catmull, PhD thesis, 1974
  - Refined by Blinn & Newell, 1976
- Texture mapping ensures that "all the right things" happen as a textured polygon is transformed and rendered.

Non-parametric texture mapping



- With "non-parametric texture mapping":
  - Texture size and orientation are fixed
  - They are unrelated to size and orientation of polygon
  - Gives cookie-cutter effect

Parametric texture mapping

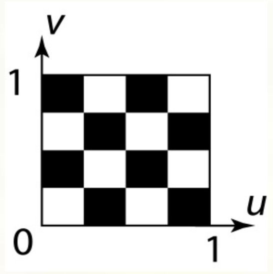


- With "parametric texture mapping," texture size and orientation are tied to the polygon.
- Idea:
  - Separate "texture space" and "screen space"
  - Texture the polygon as before, but in texture space
  - Deform (render) the textured polygon into screen space

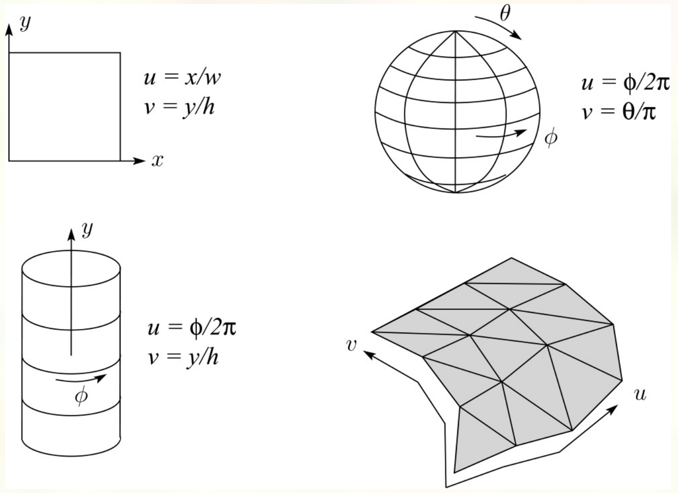
- A texture can modulate just about any parameter – diffuse color, specular color, specular exponent, …

Implementing texture mapping

- A texture lives in it own abstract image coordinates parameterized by (u,v) in the range ([0..1], [0..1]):



- It can be wrapped around many different surfaces:



- Computing (u,v) texture coordinates in a ray tracer is fairly straightforward.
- Note: if the surface moves/deforms, the texture goes with it.

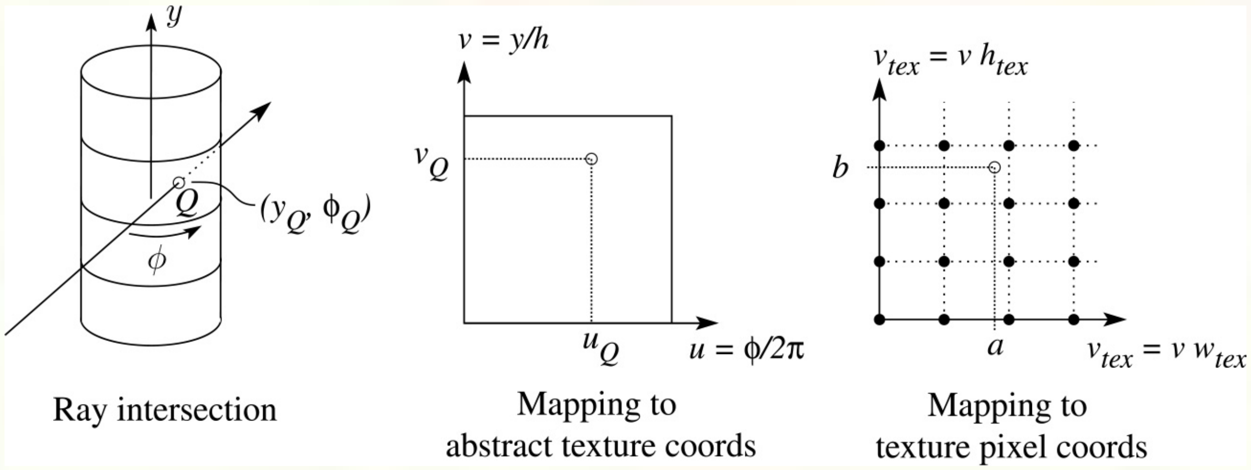
Mapping to texture image coords

- The texture is usually stored as an image. Thus, we need to convert from abstract texture coordinate:

  (u,v) in the range $([0..1], [0..1])$

  to texture image coordinates:

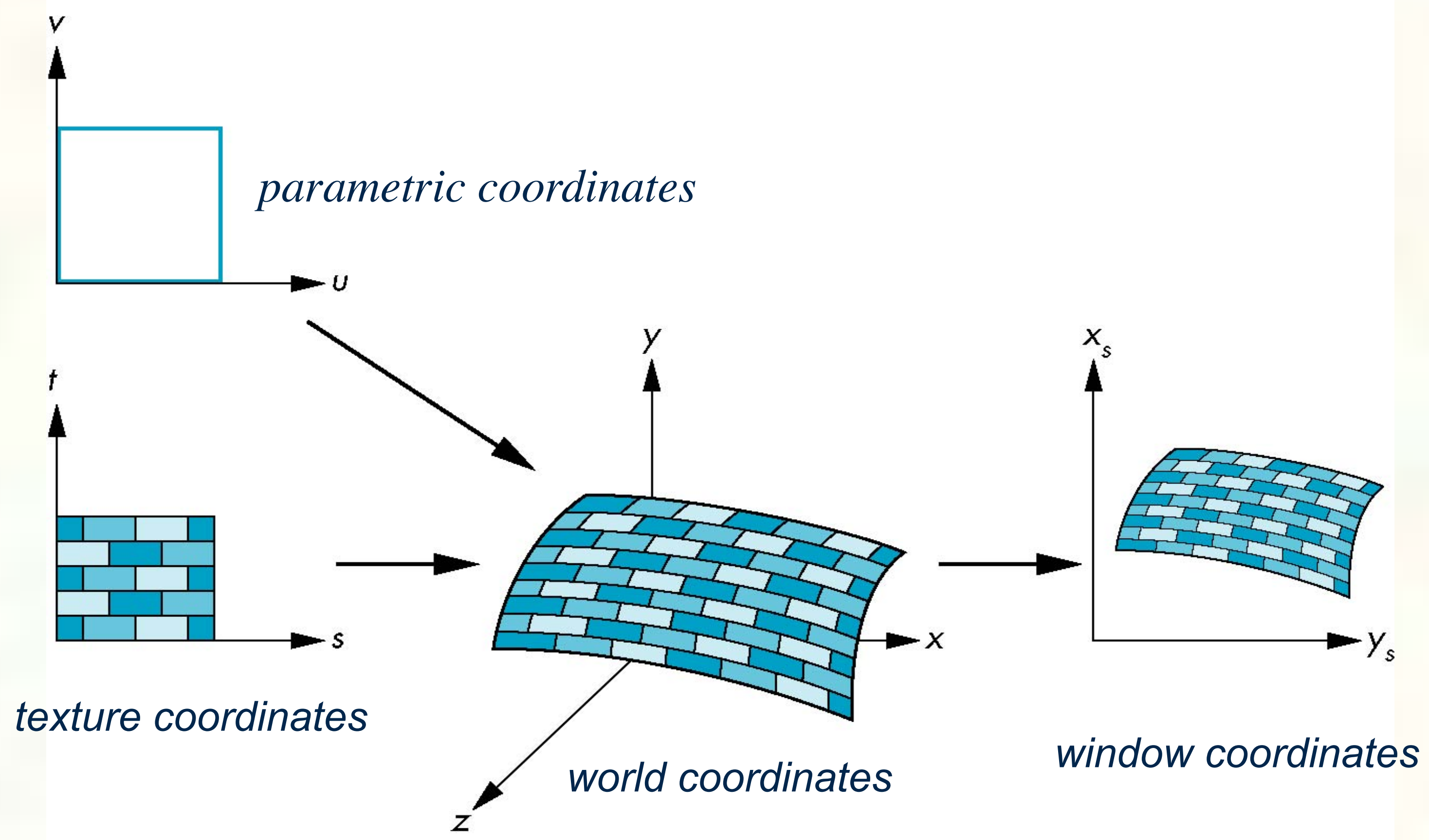
  (u_{tex}, v_{tex}) in the range $([0..\ w_{tex}], [0..\ h_{tex}])$



- **Q**: What do you do when the texture sample you need lands between texture pixels?

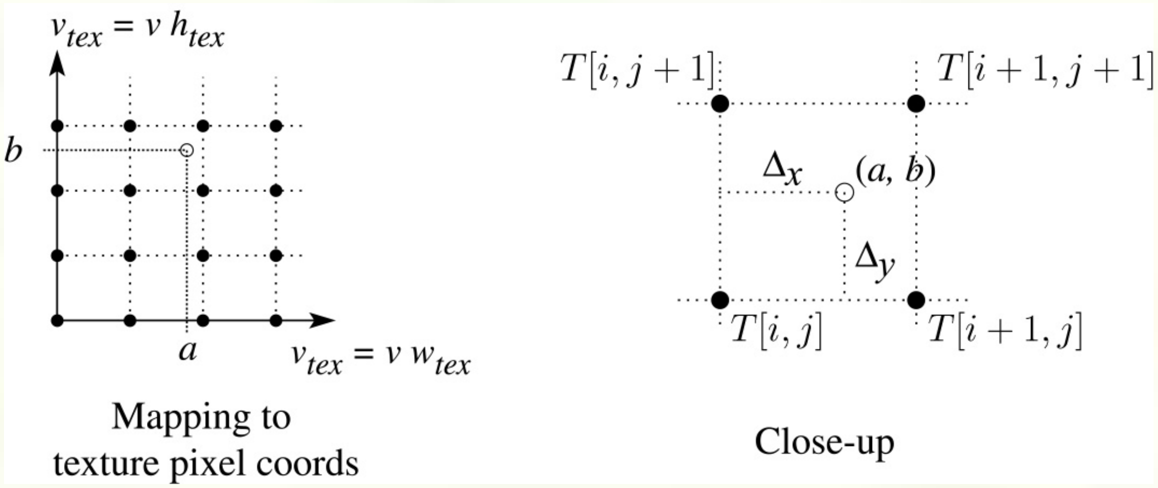
Transformed texture coordinates

- Interpolated over rasterized primitives



Texture resampling

- We need to resample the texture:

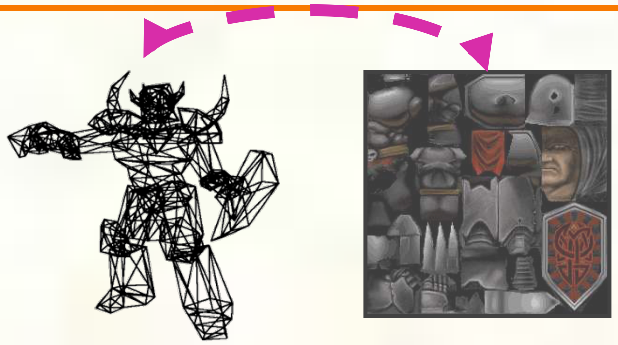


- A common choice is **bilinear interpolation**:

$$
\begin{aligned}
T(a,b) &= T[i+\Delta_x, j+\Delta_y] \\
&= (1-\Delta_x)(1-\Delta_y)T[i,j] + \Delta_x(1-\Delta_y)T[i+1,j] \\
&\quad + (1-\Delta_x)\Delta_y T[i,j+1] + \Delta_x\Delta_y T[i+1,j+1]
\end{aligned}
$$

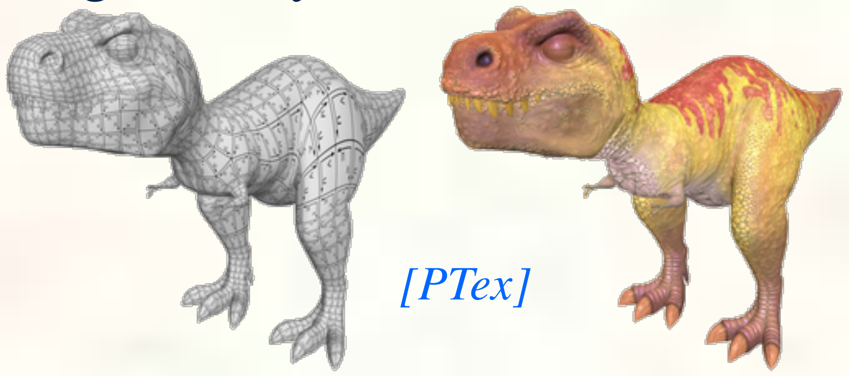
Source of texture coordinates?

- Assigned ad-hoc by artist
  - Tedious!
  - Has gift wrapping problem



- Computed based on XYZ position
  - Texture coordinate generation ("texgen")
  - Hard to map to "surface space"
  - Function maps (x,y,z) to (s,t,r,q)

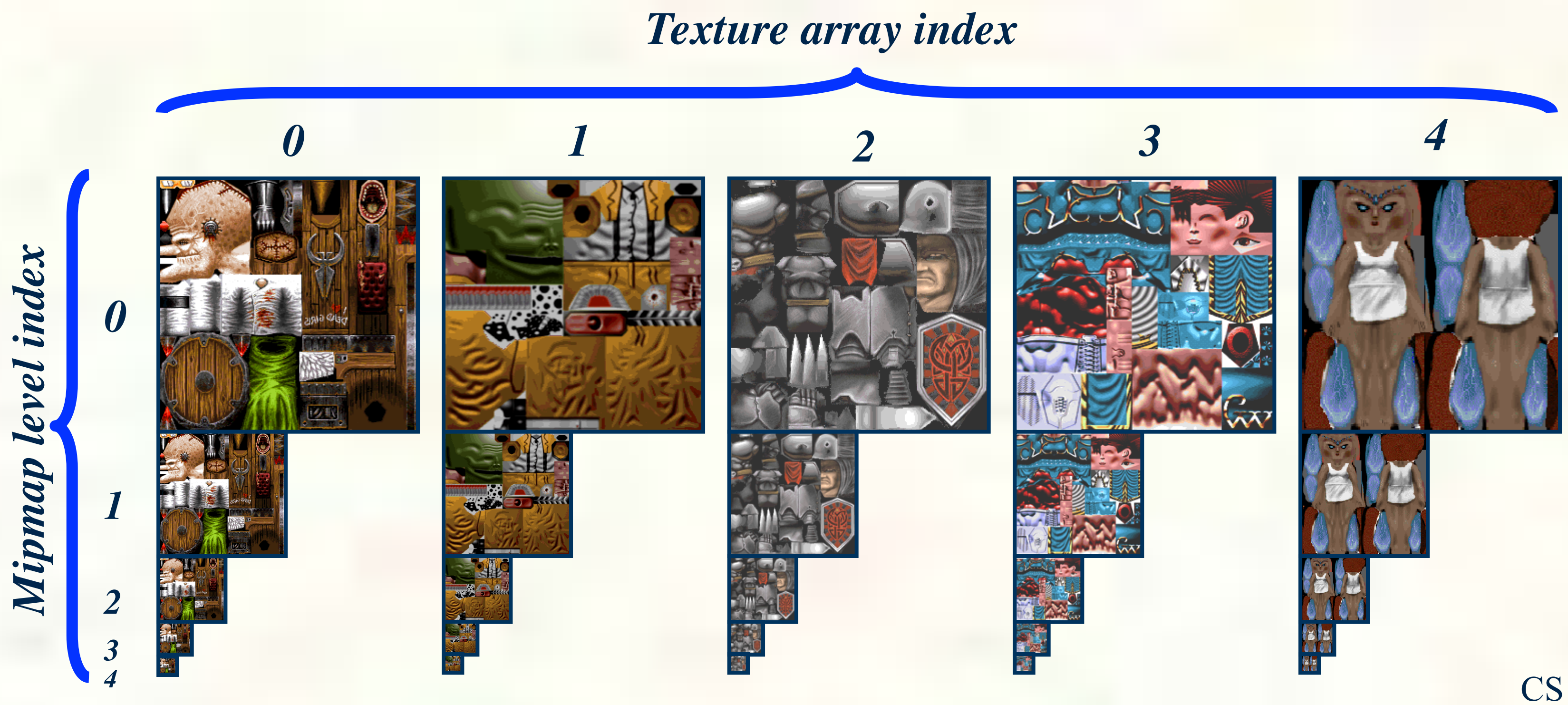
- From bi-variate parameterization of geometry
  - Good when geometry is generated from patches
  - So (u,v) of patch maps to (x,y,z) and (s,t)



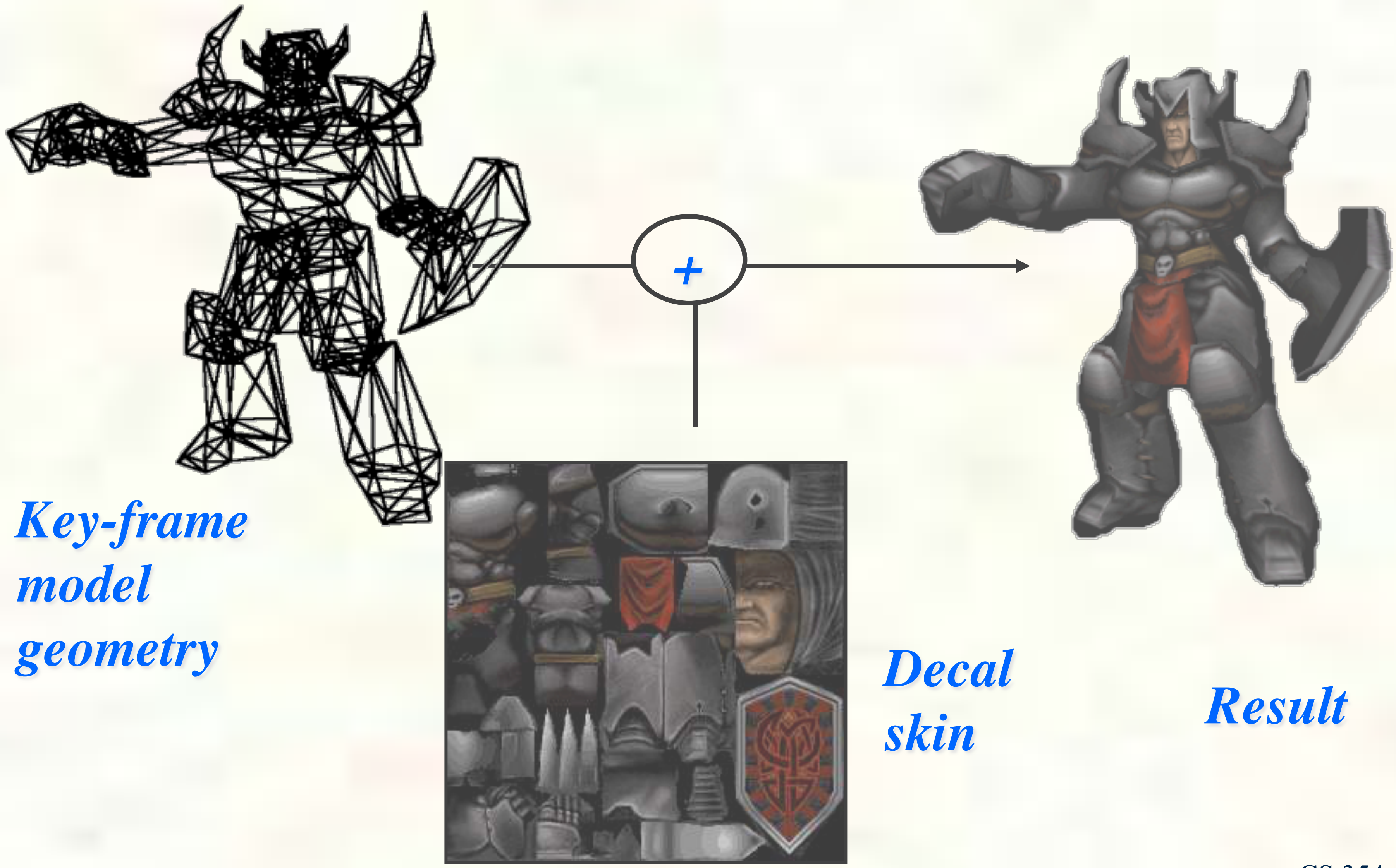
[PTex]

Texture Arrays

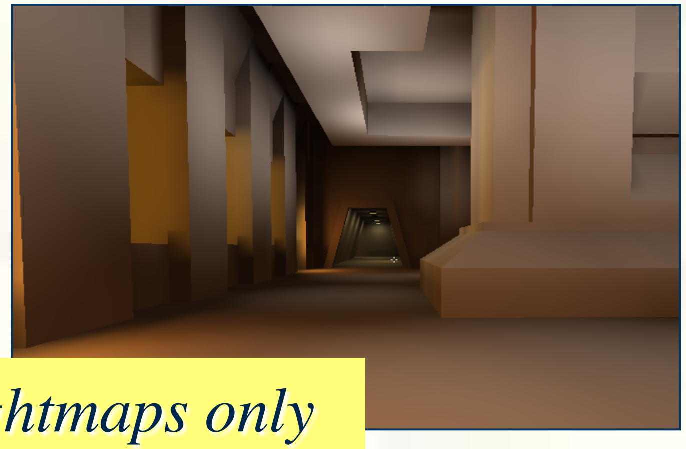
- **Multiple skins packed in texture array**
  - Motivation: binding to one multi-skin texture array avoids texture bind per object



Textured Polygonal Models



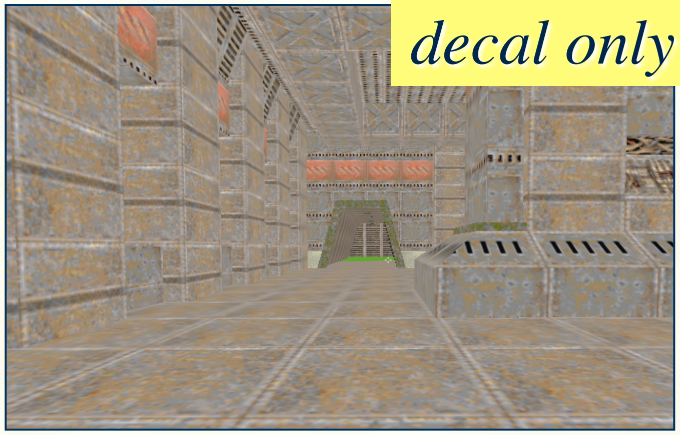
Multiple Textures



lightmaps only

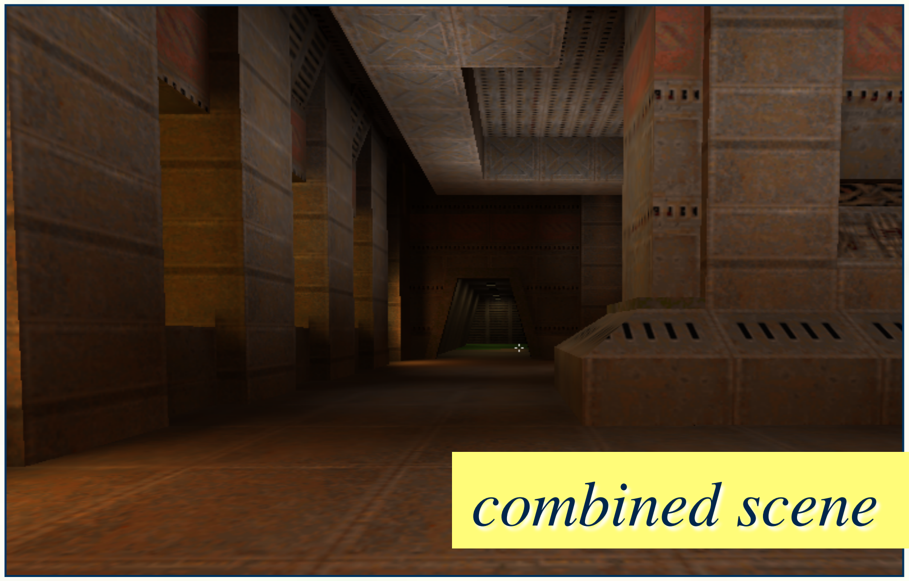
×

(modulate)



decal only

=



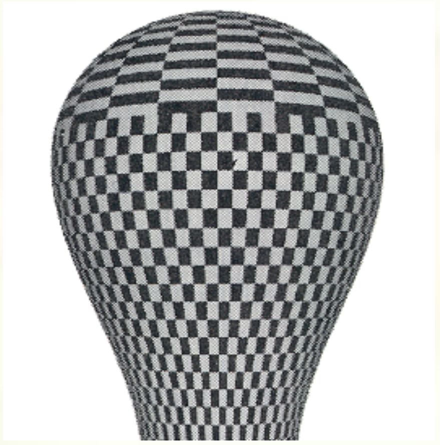
combined scene

* Id Software's Quake 2
circa 1997

Can define material by program

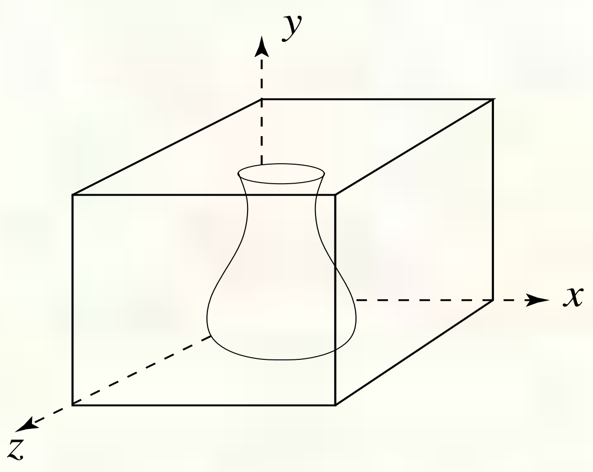
- A 'surface shader' computes the color of each ray that hits the surface.
- Example: Renderman surface shader

```
/*
 * Checkerboard
 */
surface checker(float Kd=.5, Ka=.1) {
  float smod = mod(10*s, 1);
  float tmod = mod(10*t, 1);
  if (smod < 0.5) {
    if (tmod < 0.5) Ci=Cs; else Ci=color(0,0,0);
  } else {
    if (tmod < 0.5) Ci=color(0,0,0); else Ci=Cs;
  }
  Oi = Os;
  Ci = Oi*Ci*(
          Ka*ambient() +
          Kd*diffuse(faceforward(normalize(N),I)));
}
```



Solid textures

- **Q**: What kinds of artifacts might you see from using a marble veneer instead of real marble?

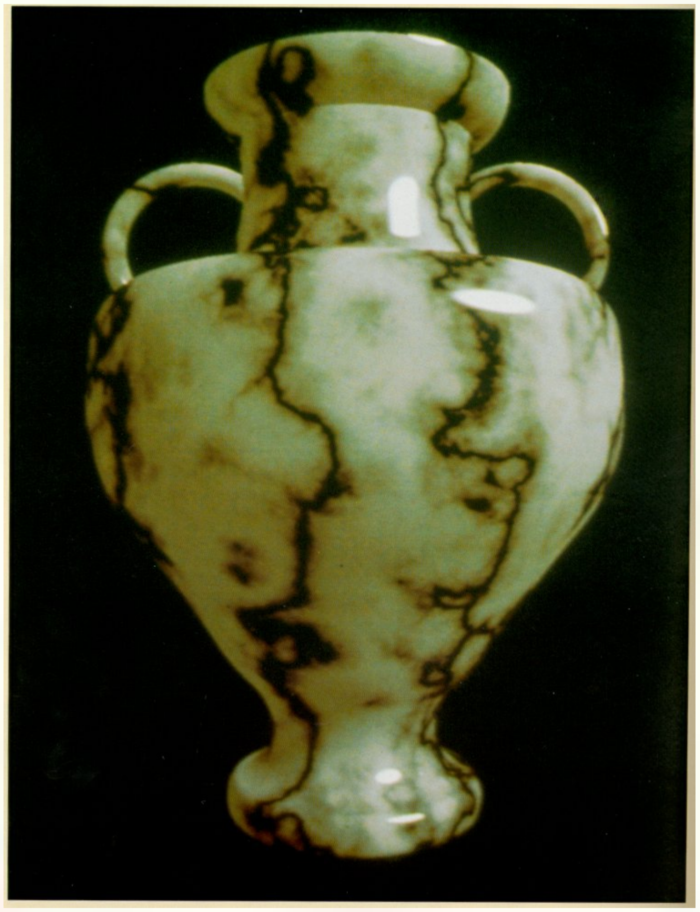


- One solution is to use **solid textures**:
  - Use model-space coordinates to index into a 3D texture
  - Like "carving" the object from the material

- One difficulty of solid texturing is coming up with the textures.

Solid textures (cont'd)

- Here's an example for a vase cut from a solid marble texture:



- *Solid marble texture by Ken Perlin, (Foley, IV-21)*

Interpolation in OpenGl

- OpenGl supports 2D, 3D and 2D perspective texturing by adding r and q parameters to s and t, r for 3D and q for perspective textures.
- First we need to interpolate (s,t,r,q)
- This is the `f[TEX3]` part of the TXP instruction
- Projective texturing means we want (s/q, t/q)
  - And possible r/q if shadow mapping
- In order to correct for perspective, hardware actually interpolates
  - (s/w, t/w, r/w, q/w)
- If not projective texturing, could linearly interpolate inverse w (or 1/w)
  - Then compute its reciprocal to get w
    - Since 1/(1/w) equals w
  - Then multiply (s/w,t/w,r/w,q/w) times w
    - To get (s,t,r,q)
- If projective texturing, we can instead
  - Compute reciprocal of q/w to get w/q
  - Then multiple (s/w,t/w,r/w) by w/q to get (s/q, t/q, r/q)
- Bottom line, for regular 2D perspective on triangles, set r=0, q=1 and let perspective correct interpolation of surface points handle it.

Interpolation Operations

- Ax + By + C per scalar linear interpolation
  - 2 MADs
- One reciprocal to invert q/w for projective texturing
  - Or one reciprocal to invert 1/w for perspective texturing
- Then 1 MUL per component for s/w * w/q
  - Or s/w * w
- For (s,t) means
  - 4 MADs, 2 MULs, & 1 RCP
  - (s,t,r) requires 6 MADs, 3 MULs, & 1 RCP
- All floating-point operations

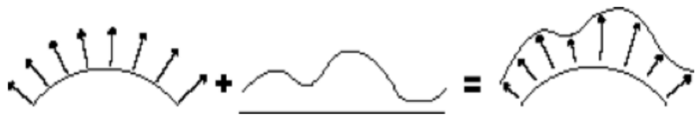
Texture Space Mapping

- Have interpolated & projected coordinates
- Now need to determine what texels to fetch

- Multiple (s,t) by (width,height) of texture base level
  - Could convert (s,t) to fixed-point first
    - Or do math in floating-point
  - Say based texture is 256x256 so
    - So compute (s*256, t*256)=(u,v)

Displacement and Bump Mapping

- Use surface offsets stored in texture
  - Perturb or displace the surface
  - Shade on the resulting surface normals



$$\mathbf{P}(u,v)$$

$$\mathbf{S}(u,v) = \frac{\partial \mathbf{P}(u,v)}{\partial u} \qquad \mathbf{T}(u,v) = \frac{\partial \mathbf{P}(u,v)}{\partial v}$$

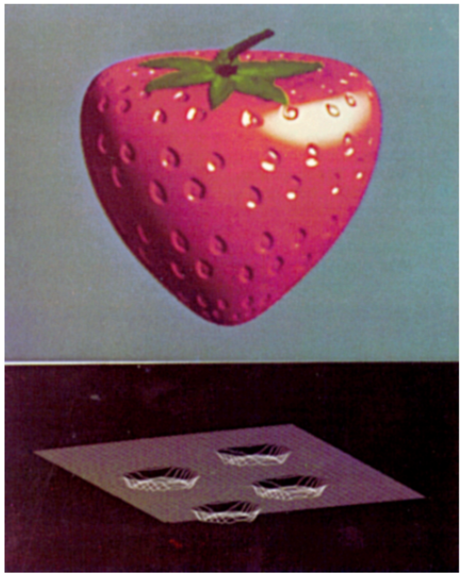
$$\mathbf{N}(u,v) = \mathbf{S} \times \mathbf{T}$$

- **Displacement**

$$\mathbf{P}'(u,v) = \mathbf{P}(u,v) + h(u,v)\mathbf{N}(u,v)$$

- **Perturbed normal**

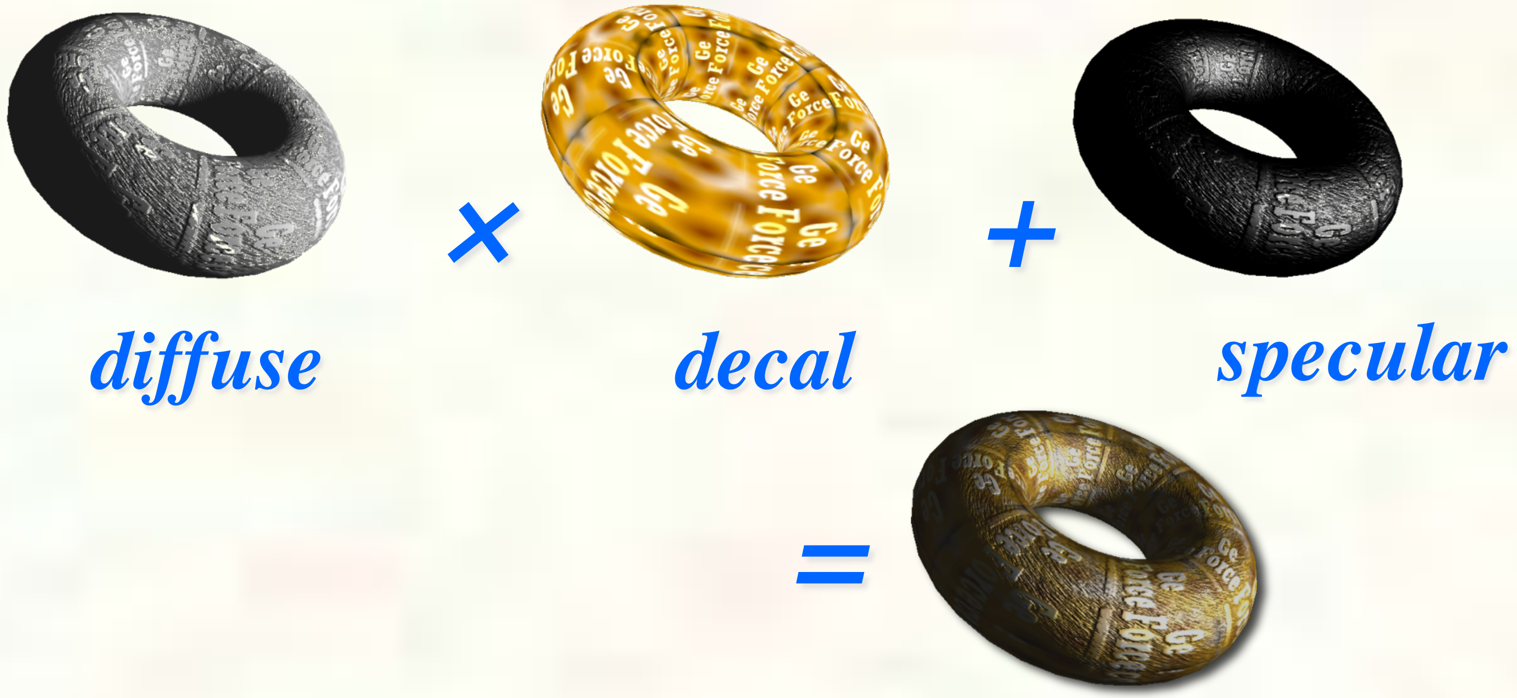
$$\mathbf{N}'(u,v) = \mathbf{P}'_u \times \mathbf{P}'_v$$
$$= \mathbf{N} + h_u(\mathbf{T} \times \mathbf{N}) + h_v(\mathbf{S} \times \mathbf{N})$$



From Blinn 1976

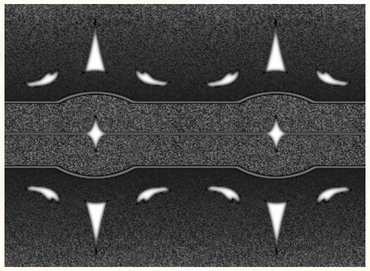
Normal Mapping

- Bump mapping via a normal map texture
  - Normal map – x,y,z components of actual normal
  - Instead of a height field 1 value per pixel
  - The normal map can be generated from the height field
  - Otherwise have to orient the normal coordinates to the surface

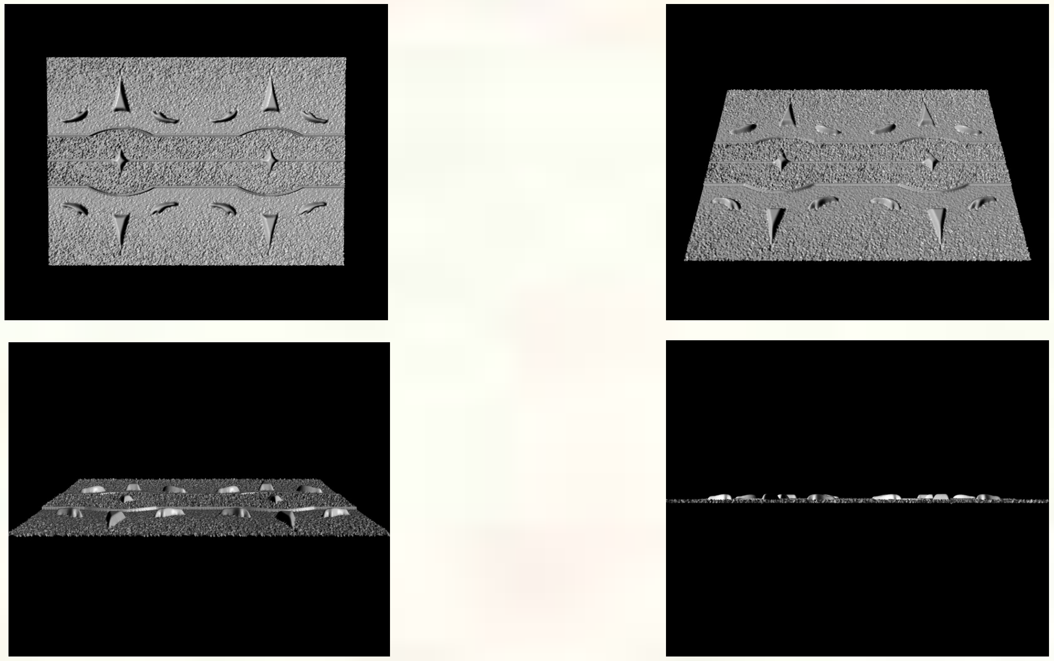


Displacement vs. bump mapping

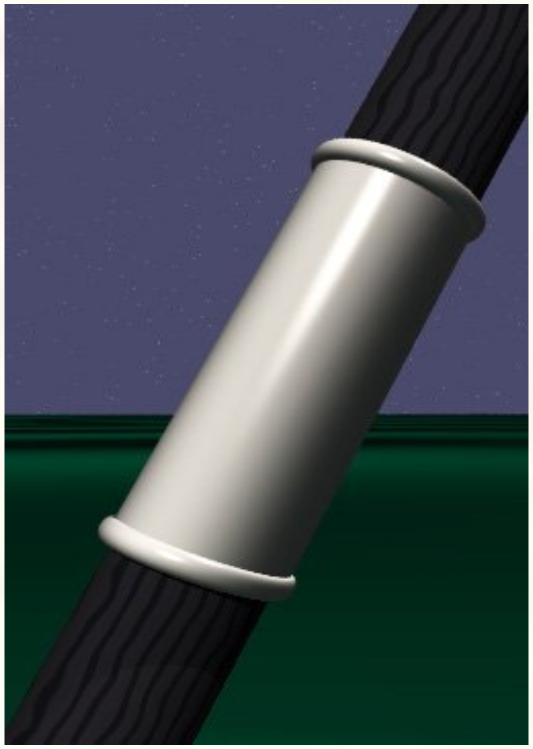
- Input texture

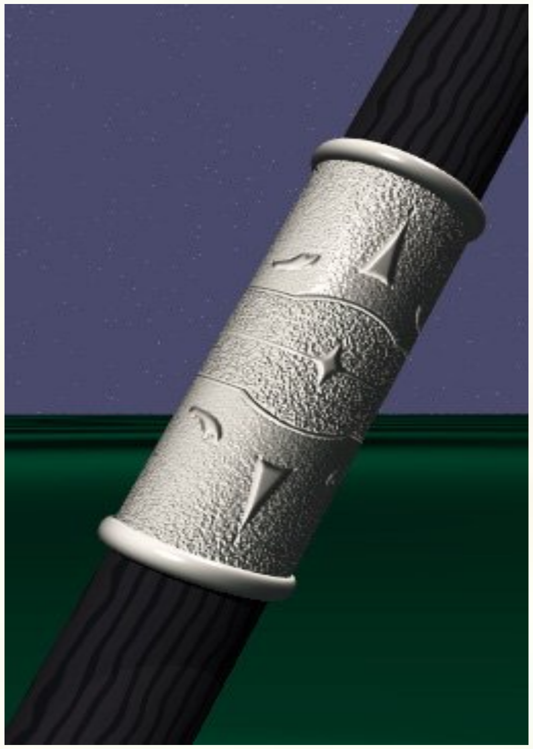


- Rendered as displacement map over a rectangular surface



Displacement vs. bump mapping (cont'd)


Original rendering

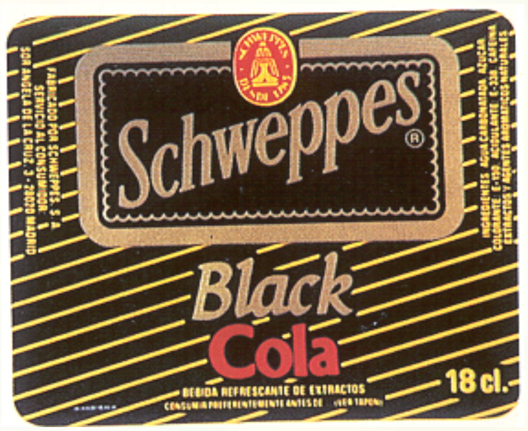

Rendering with bump map wrapped around a

cylinder

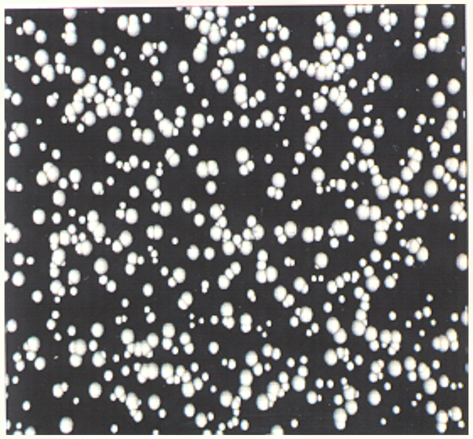
Bump map and rendering by Wyvern Aldinger

Bump mapping example

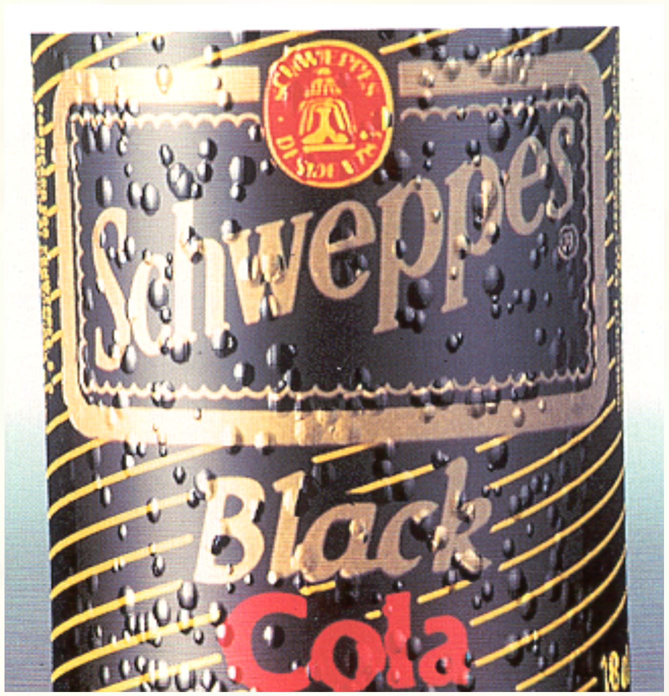
Texture #1
(diffuse color)



Texture #2
(bump map)

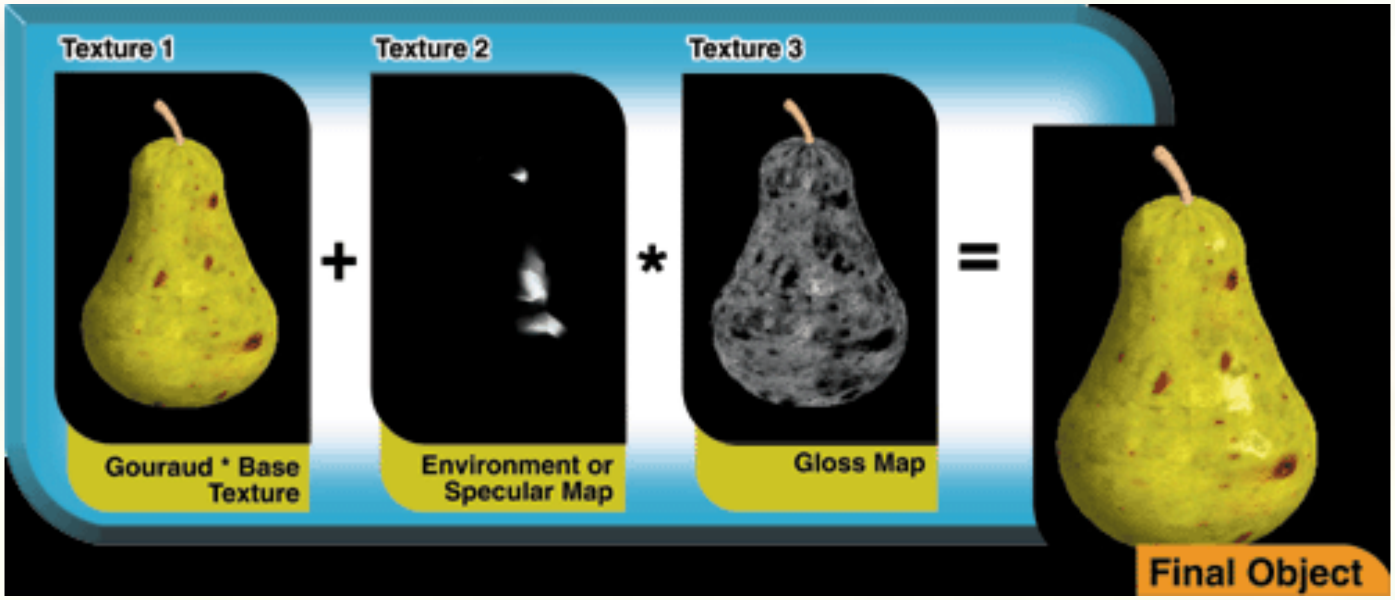


Rendered Image



Combining texture maps

- Using texture maps in combination gives even better effects.



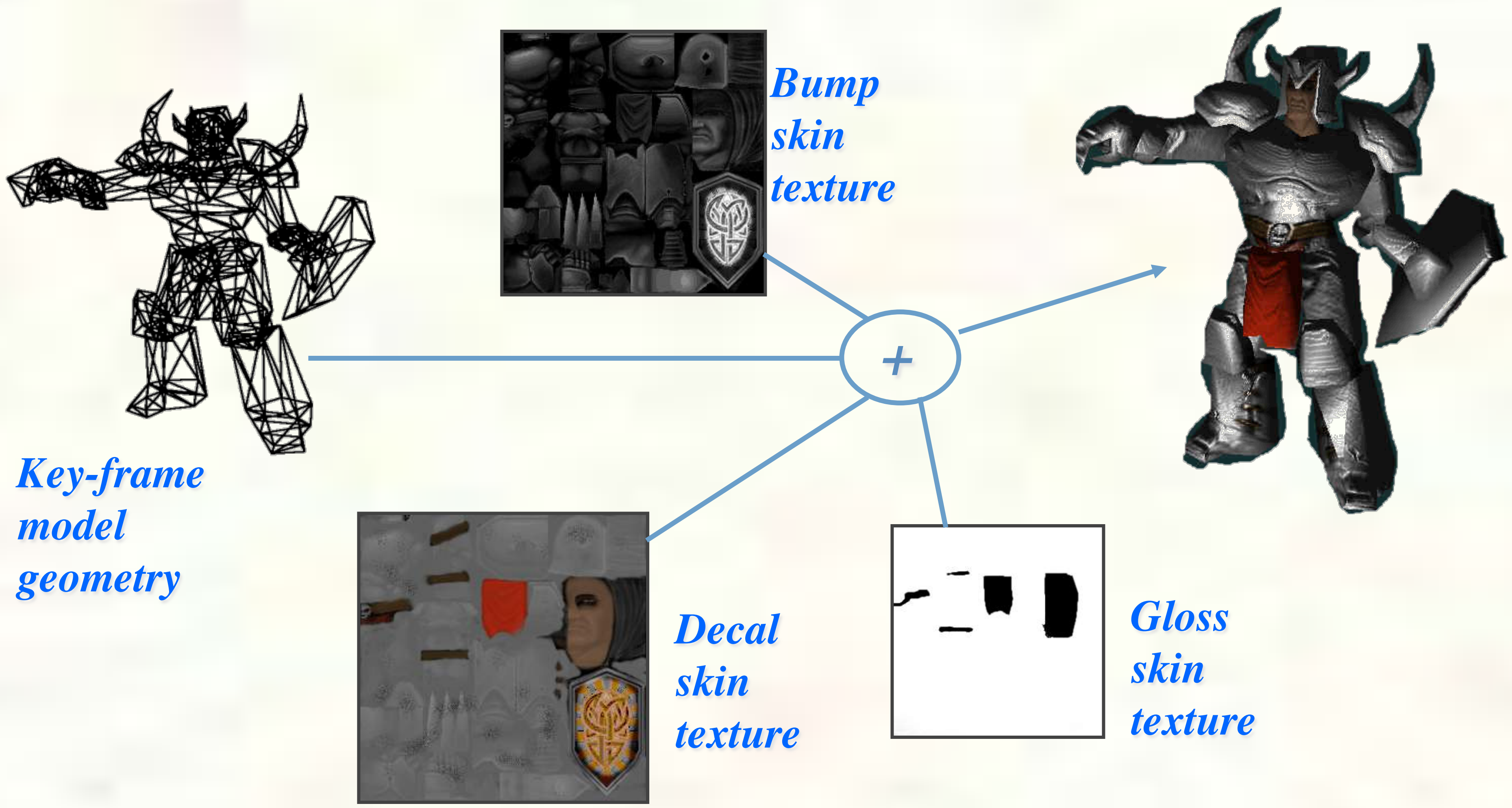
Diffuse color

Environment map (not necessary in ray tracer)

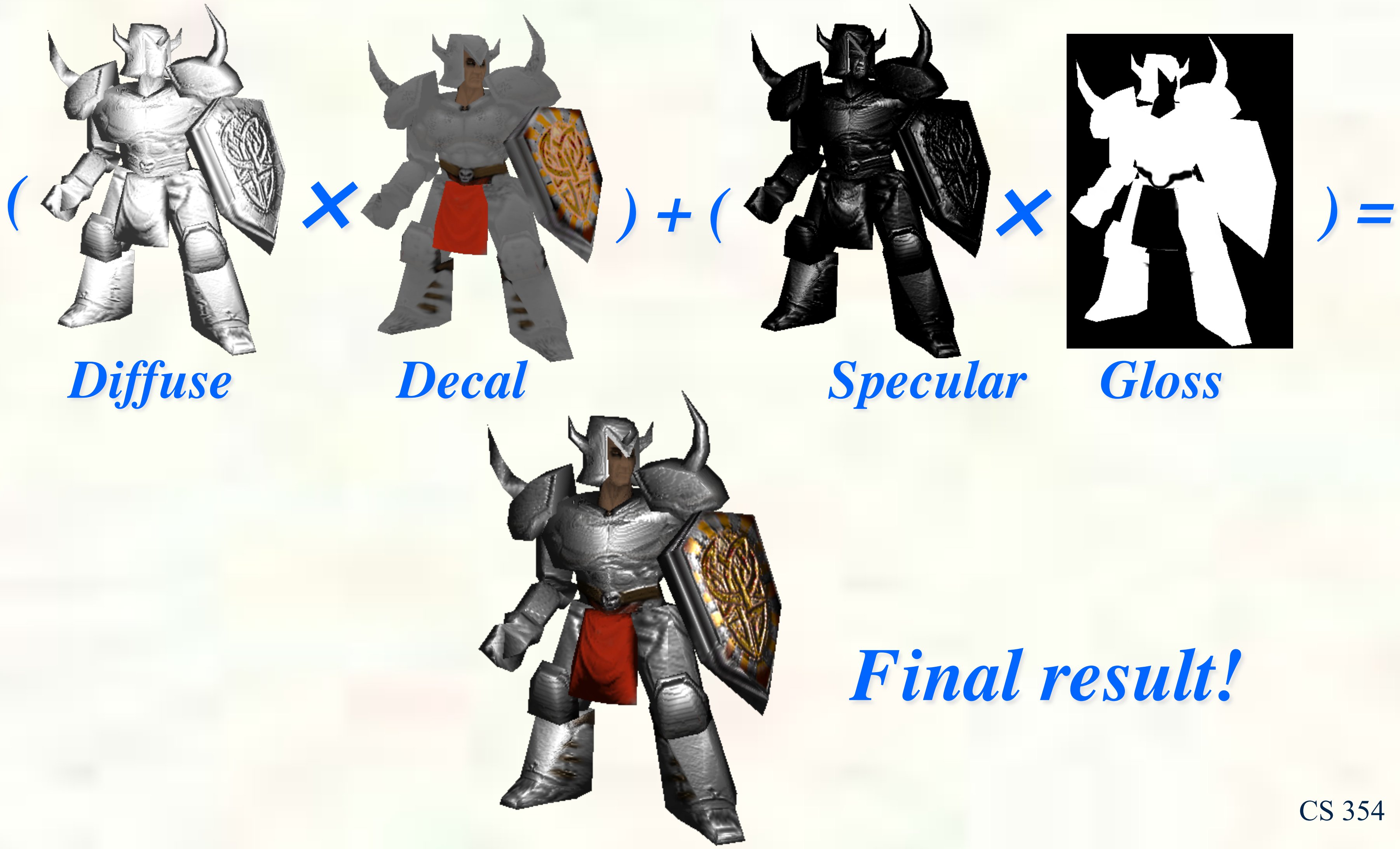
Specular coefficient

Material properties (coefficients in shading equation)

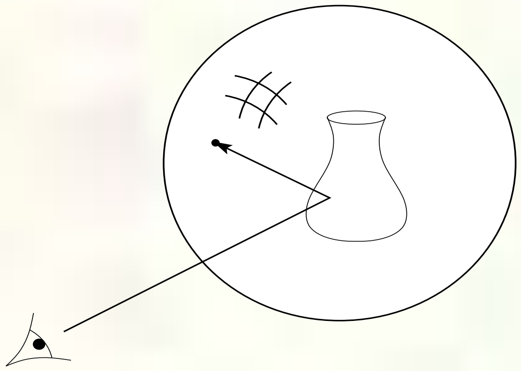
Multiple Textures



Multitexturing



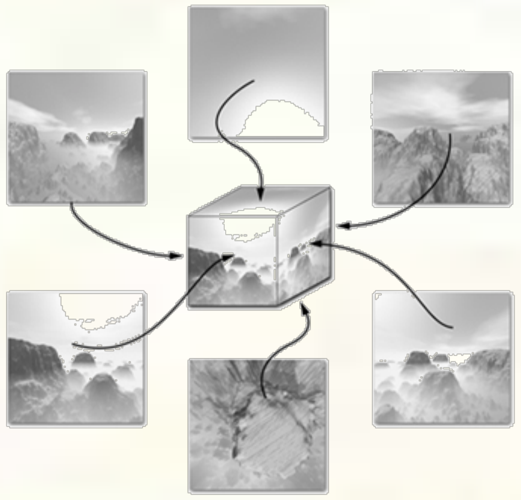
Environment mapping

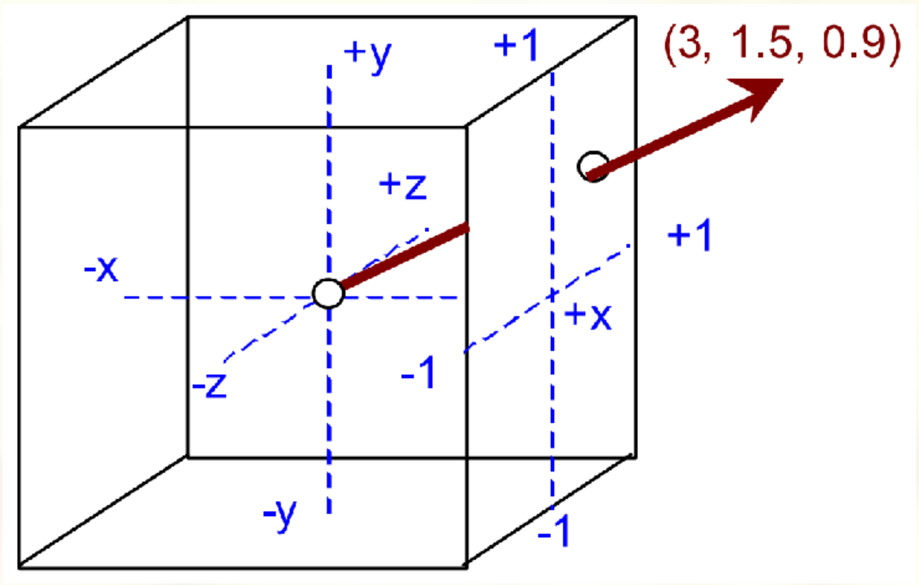


- In **environment mapping** (also known as **reflection mapping**), a texture is used to model an object's environment:
  - Rays are bounced off objects into environment
  - Color of the environment used to determine color of the illumination
  - Really, a simplified form of ray tracing
  - Environment mapping works well when there is just a single object – or in conjunction with ray tracing
- Under simplifying assumptions, environment mapping can be implemented in hardware.
- With a ray tracer, the concept is easily extended to handle refraction as well as reflection.

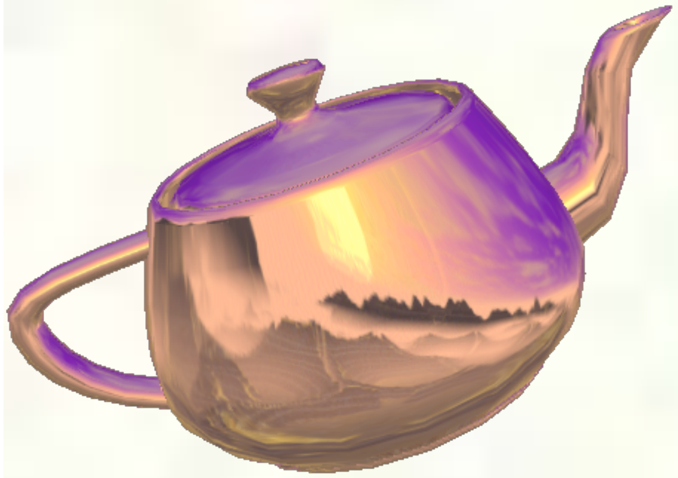
Cube Map Textures

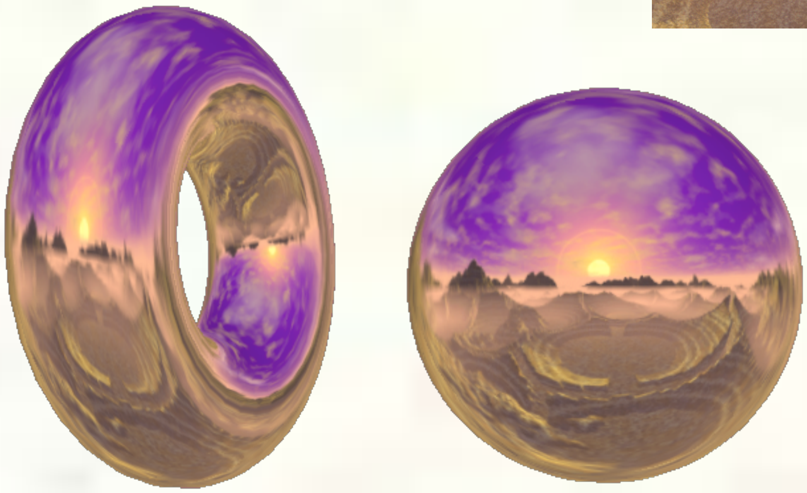
- Instead of one 2D images
  - Six 2D images arranged like the faces of a cube
    - +X, -X, +Y, -Y, +Z, -Z
- Indexed by 3D (*s*,*t*,*r*) un-normalized vector
  - Instead of 2D (*s*,*t*)
  - Where on the cube images does the vector "poke through"?
    - That's the texture result

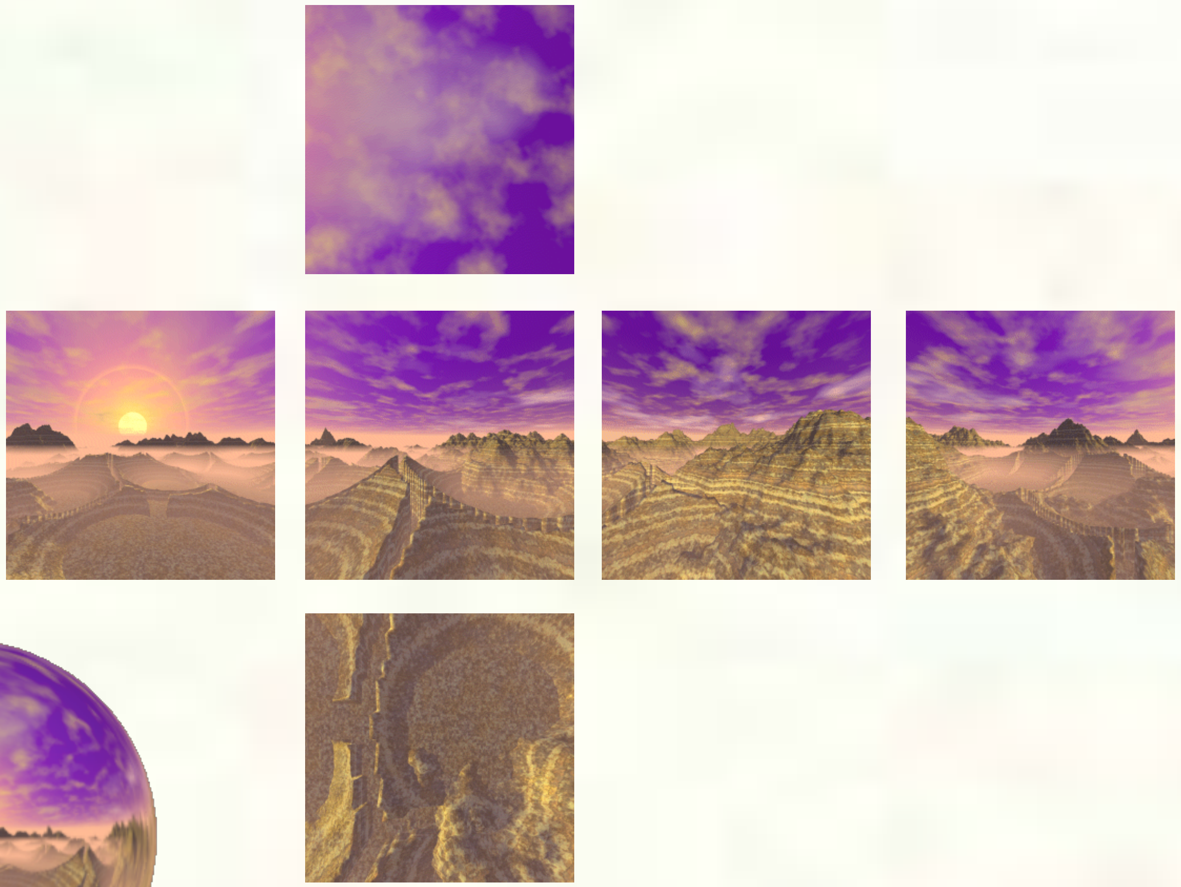




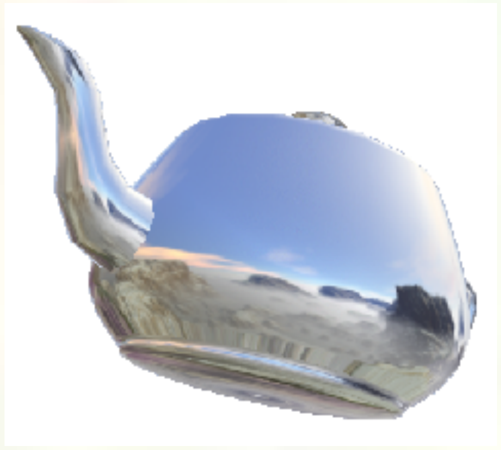
More Cube Mapping

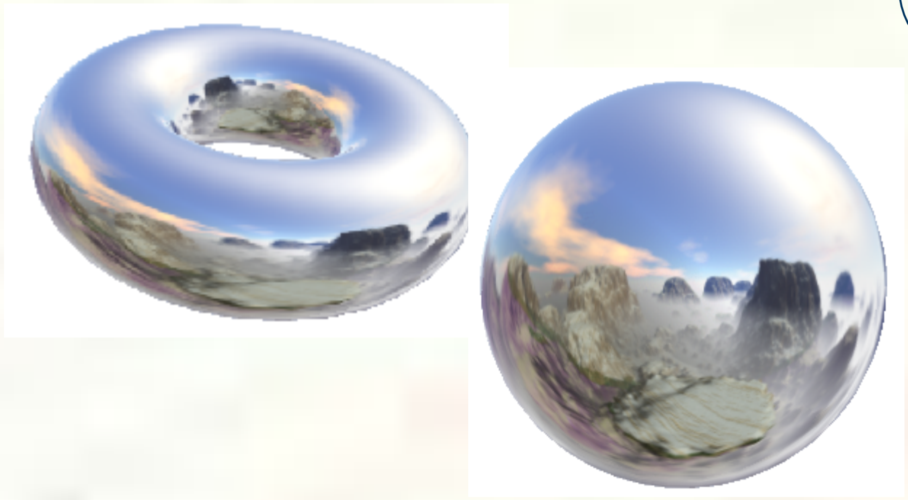


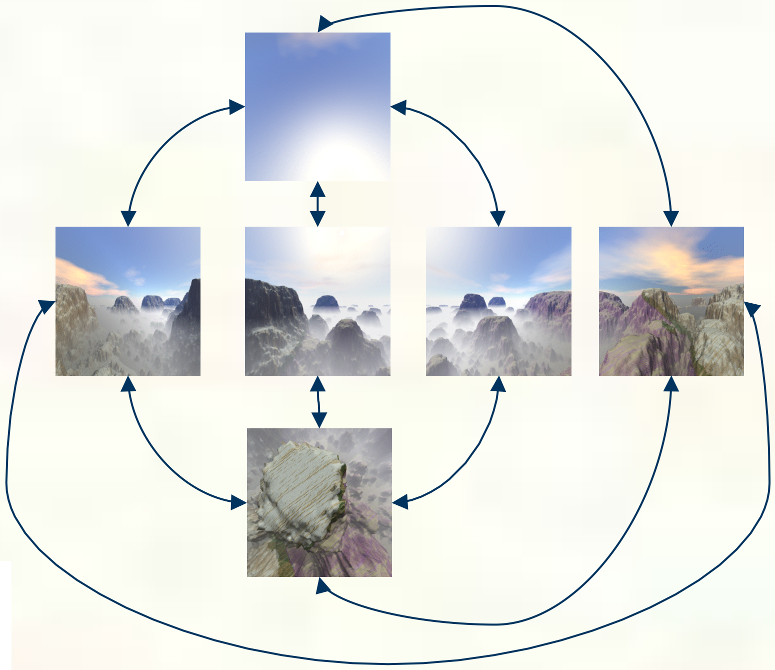




Omni-directional Lighting







Access texture by surface reflection vector

Dynamic Cube Map Textures

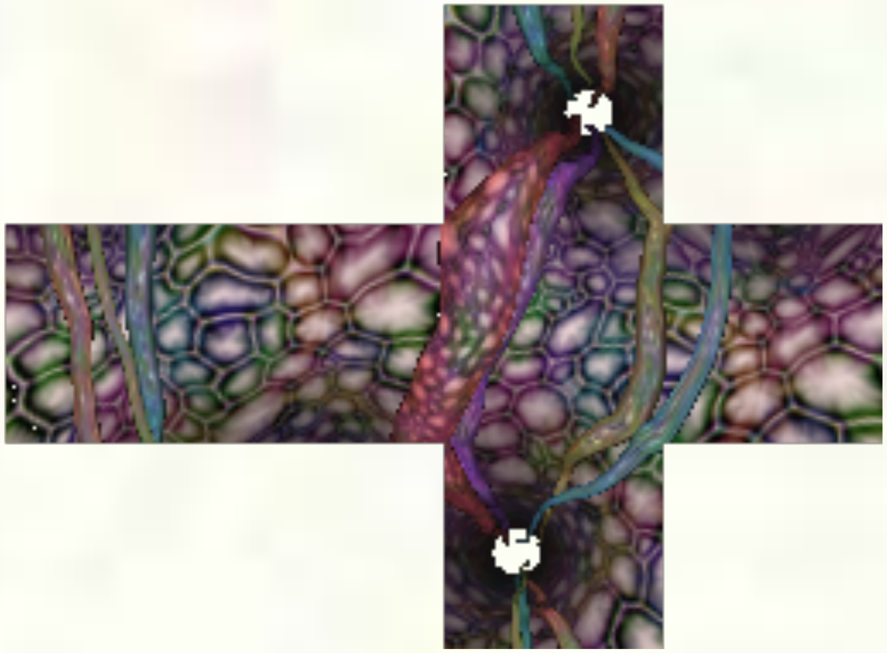
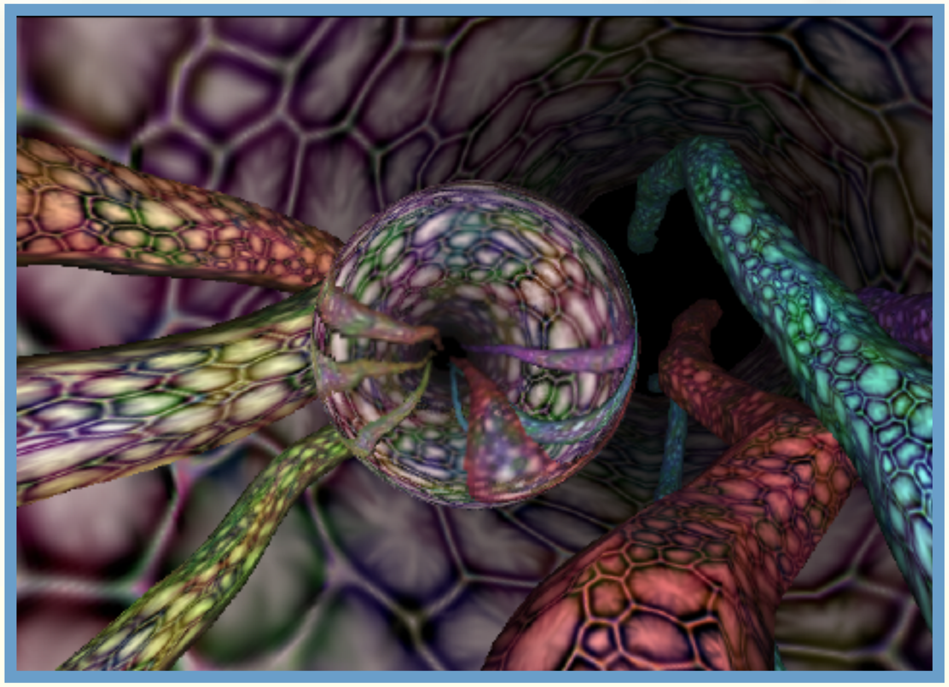


Image credit:
"Guts" GeForce 2 GTS demo,
Thant Thessman

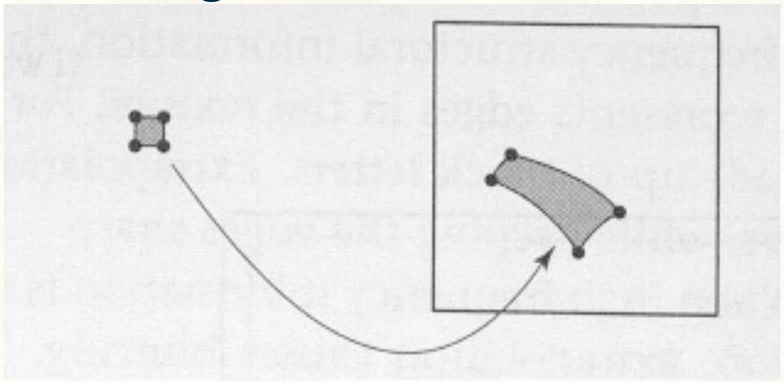
Rendered scene

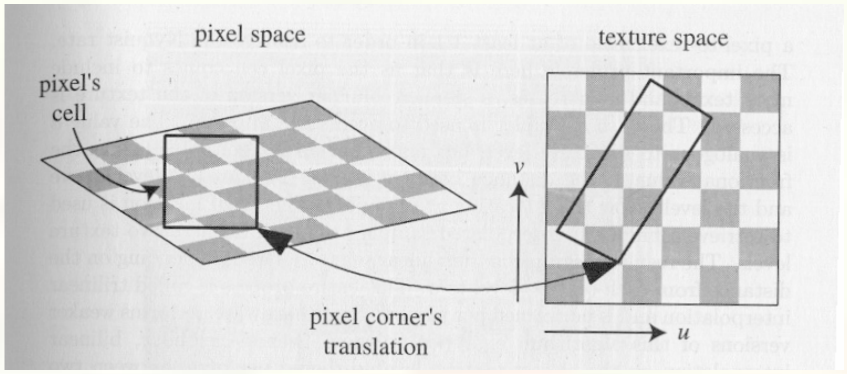


Dynamically created cube map image

How do we anti-alias textures?

- We could just super-sample.
- But textures (and shader programs) are a special case; we can use true area integration!

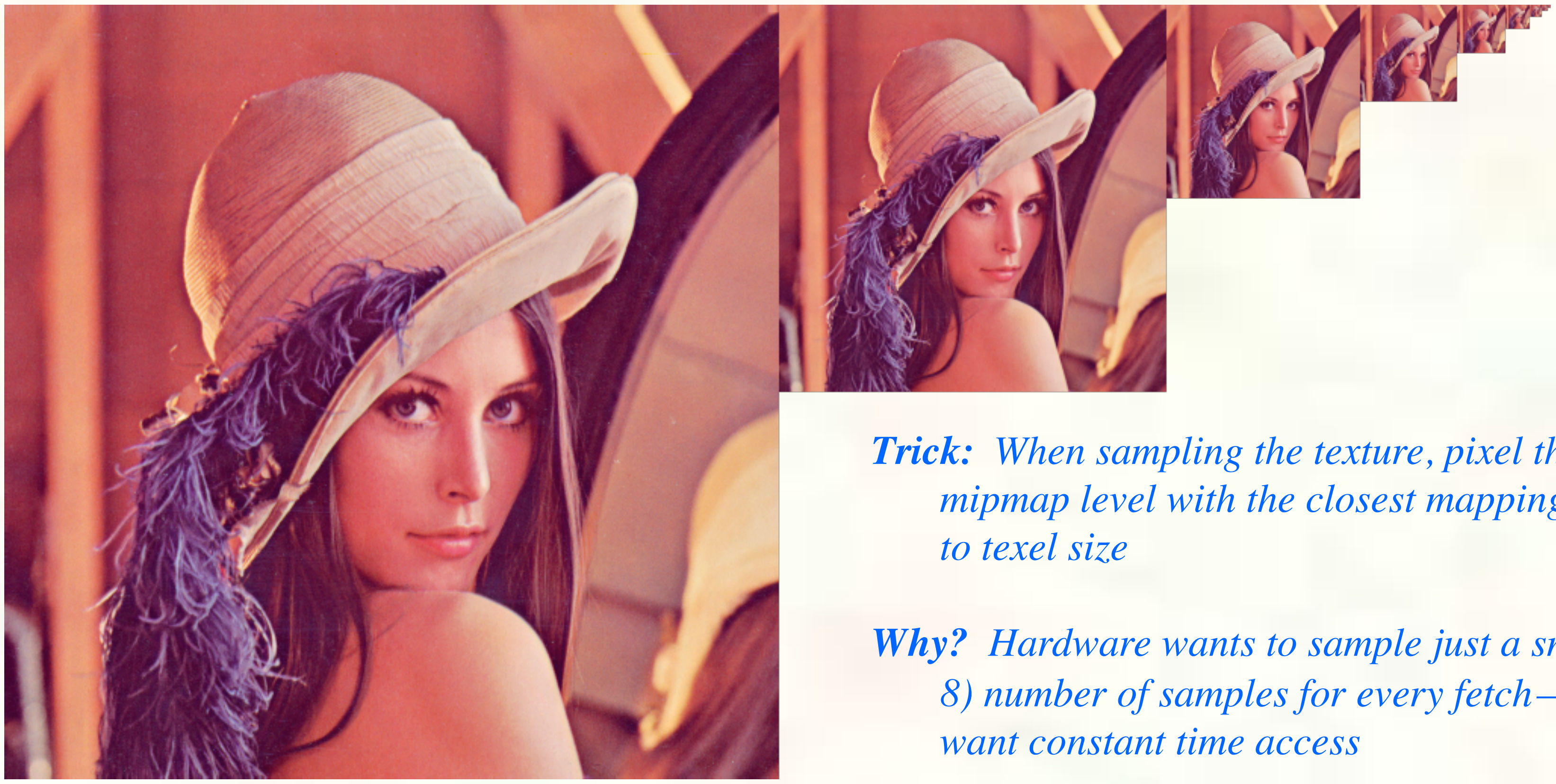




- *Approximate footprint as parallelogram*
- *Determine this approximate footprint using discrete differences*

Pre-filtered Image Versions

- Base texture image is say 256x256
  - Then down-sample 128x128, 64x64, 32x32, all the way down to 1x1

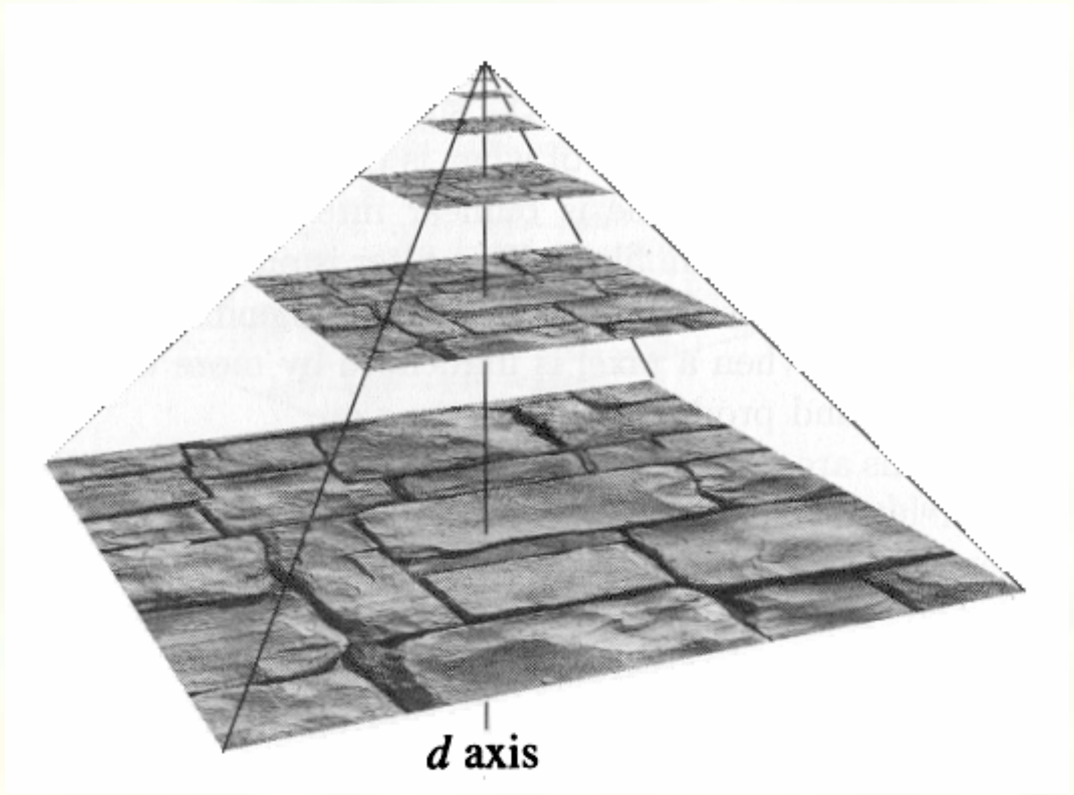


Trick: *When sampling the texture, pixel the mipmap level with the closest mapping of pixel to texel size*

Why? *Hardware wants to sample just a small (1 to 8) number of samples for every fetch—and want constant time access*

Cost of filtering can be reduced

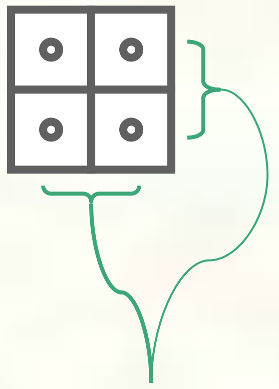
- Store a pyramid of pre-filtered images:



- During texture lookup, read from appropriate level of the pyramid.

Mipmap LOD Selection

- Tri-linear mip-mapping means compute appropriate mipmap level
- Hardware rasterizes in 2x2 pixel entities
  - Typically called quad-pixels or just *quad*
  - Finite difference with neighbors to get change in u and v with respect to window space
    - Approximation to $\partial u/\partial x$, $\partial u/\partial y$, $\partial v/\partial x$, $\partial v/\partial y$
    - Means 4 subtractions per quad (1 per pixel)
- Now compute approximation to gradient length
  - $p = \max(\mathrm{sqrt}((\partial u/\partial x)^2+(\partial u/\partial y)^2), \mathrm{sqrt}((\partial v/\partial x)^2+(\partial v/\partial y)^2))$



LOD Bias and Clamping

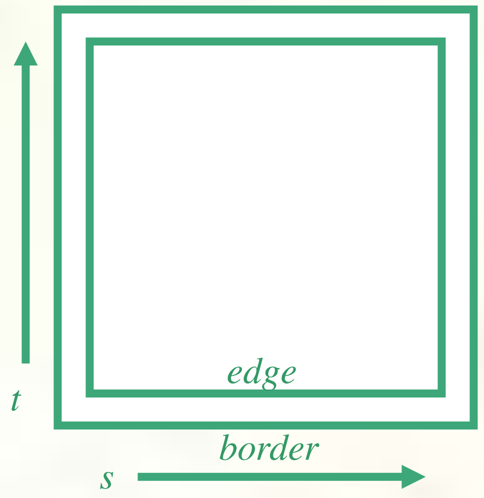
- Convert p length to power-of-two level-of-detail and apply LOD bias
  - $\lambda = \log2(p) + \text{lodBias}$
- Now clamp λ to valid LOD range
  - $\lambda' = \max(\text{minLOD}, \min(\text{maxLOD}, \lambda))$

Determine Levels and Interpolant

- Determine lower and upper mipmap levels
  - b = floor(λ')) is bottom mipmap level
  - t = floor(λ' +1) is top mipmap level
- Determine filter weight between levels
  - w = frac(λ') is filter weight

Determine Texture Sample Point

- Get (u,v) for selected top and bottom mipmap levels
  - Consider a level l which could be either level t or b
    - With (u,v) locations (ul,vl)
- Perform GL_CLAMP_TO_EDGE wrap modes
  - u_w = max(1/2*widthOfLevel(l), min(1-1/2*widthOfLevel(l), u))
  - v_w = max(1/2*heightOfLevel(l), min(1-1/2*heightOfLevel(l), v))
- Get integer location (i,j) within each level
  - (i,j) = (floor(u_w* widthOfLevel(l)), floor(v_w*))



Determine Texel Locations

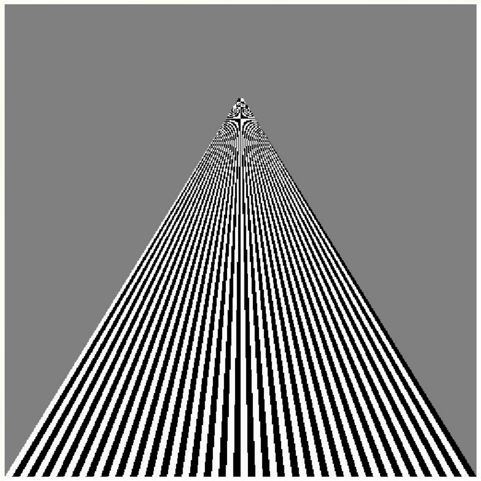
- Bilinear sample needs 4 texel locations
  - (i0,j0), (i0,j1), (i1,j0), (i1,j1)
- With integer texel coordinates
  - i0 = floor(i-1/2)
  - i1 = floor(i+1/2)
  - j0 = floor(j-1/2)
  - j1 = floor(j+1/2)
- Also compute fractional weights for bilinear filtering
  - a = frac(i-1/2)
  - b = frac(j-1/2)

Determine Texel Addresses

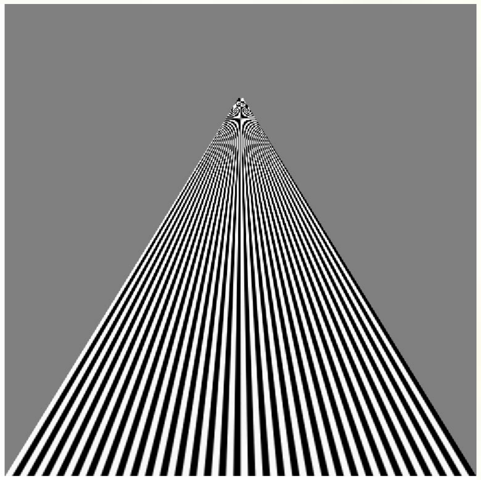
- Assuming a texture level image's base pointer, compute a texel address of each texel to fetch
  - Assume bytesPerTexel = 4 bytes for RGBA8 texture
- Example
  - addr00 = baseOfLevel(l) + bytesPerTexel*(i0+j0*widthOfLevel(l))
  - addr01 = baseOfLevel(l) + bytesPerTexel*(i0+j1*widthOfLevel(l))
  - addr10 = baseOfLevel(l) + bytesPerTexel*(i1+j0*widthOfLevel(l))
  - addr11 = baseOfLevel(l) + bytesPerTexel*(i1+j1*widthOfLevel(l))
- More complicated address schemes are needed for good texture locality!

Mipmap Texture Filtering

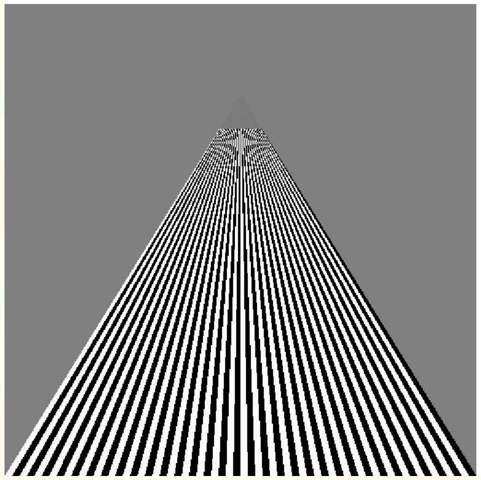
point sampling



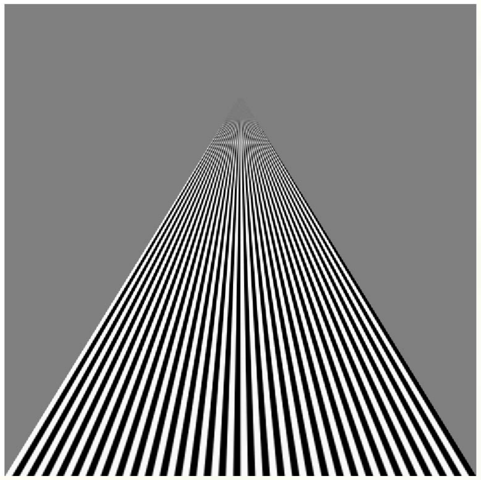
linear filtering



mipmapped point sampling

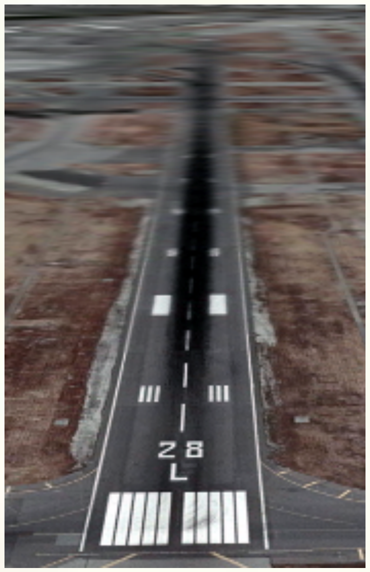


mipmapped linear filtering

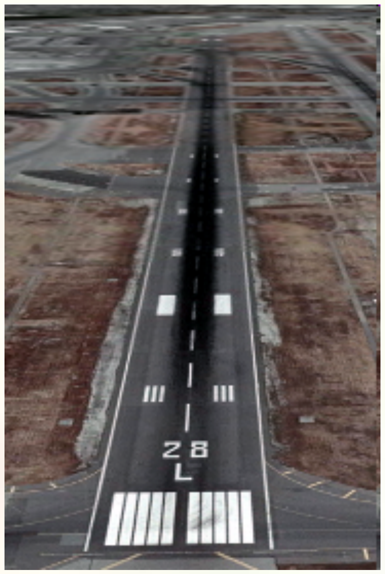


Anisotropic Texture Filtering

- Standard (isotropic) mipmap LOD selection
  - Uses magnitude of texture coordinate gradient (not direction)
  - Tends to spread blurring at shallow viewing angles
- Anisotropic texture filtering considers gradients direction
  - Minimizes blurring

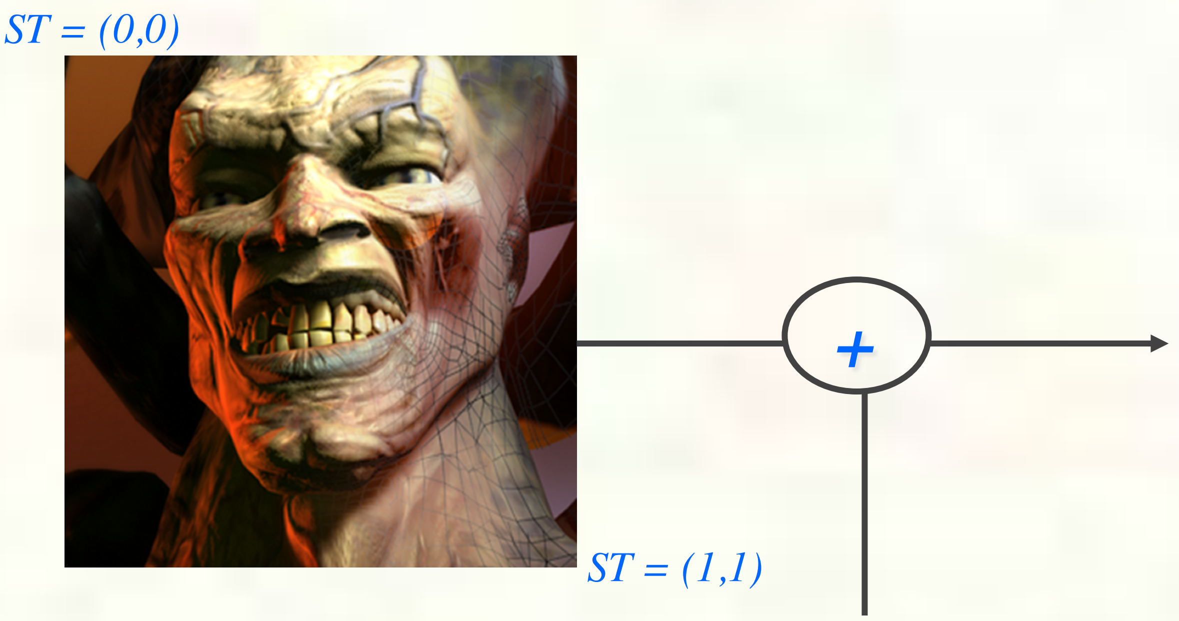


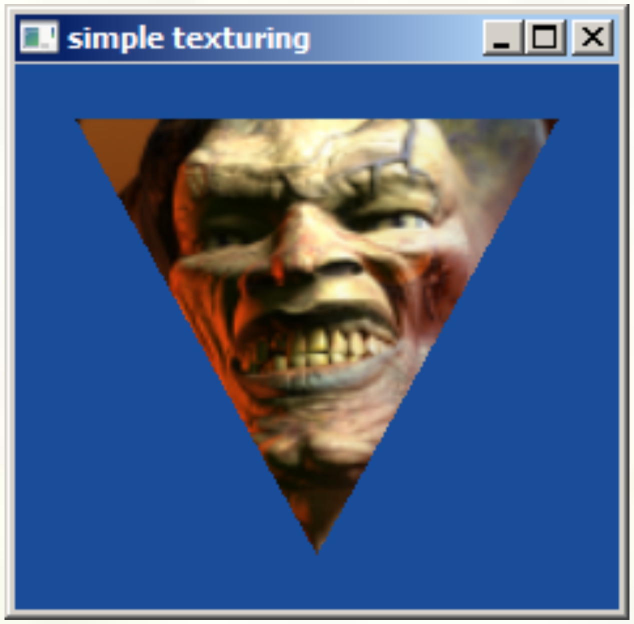
Isotropic



Anisotropic

Texture Mapping in OpenGL





```
glBegin(GL_TRIANGLES);
  glTexCoord2f(0, 0);
  glVertex2f(-0.8, 0.8);

  glTexCoord2f(1, 0);
  glVertex2f(0.8, 0.8);

  glTexCoord2f(0.5, 1);
  glVertex2f(0.0, -0.8);
glEnd();
```

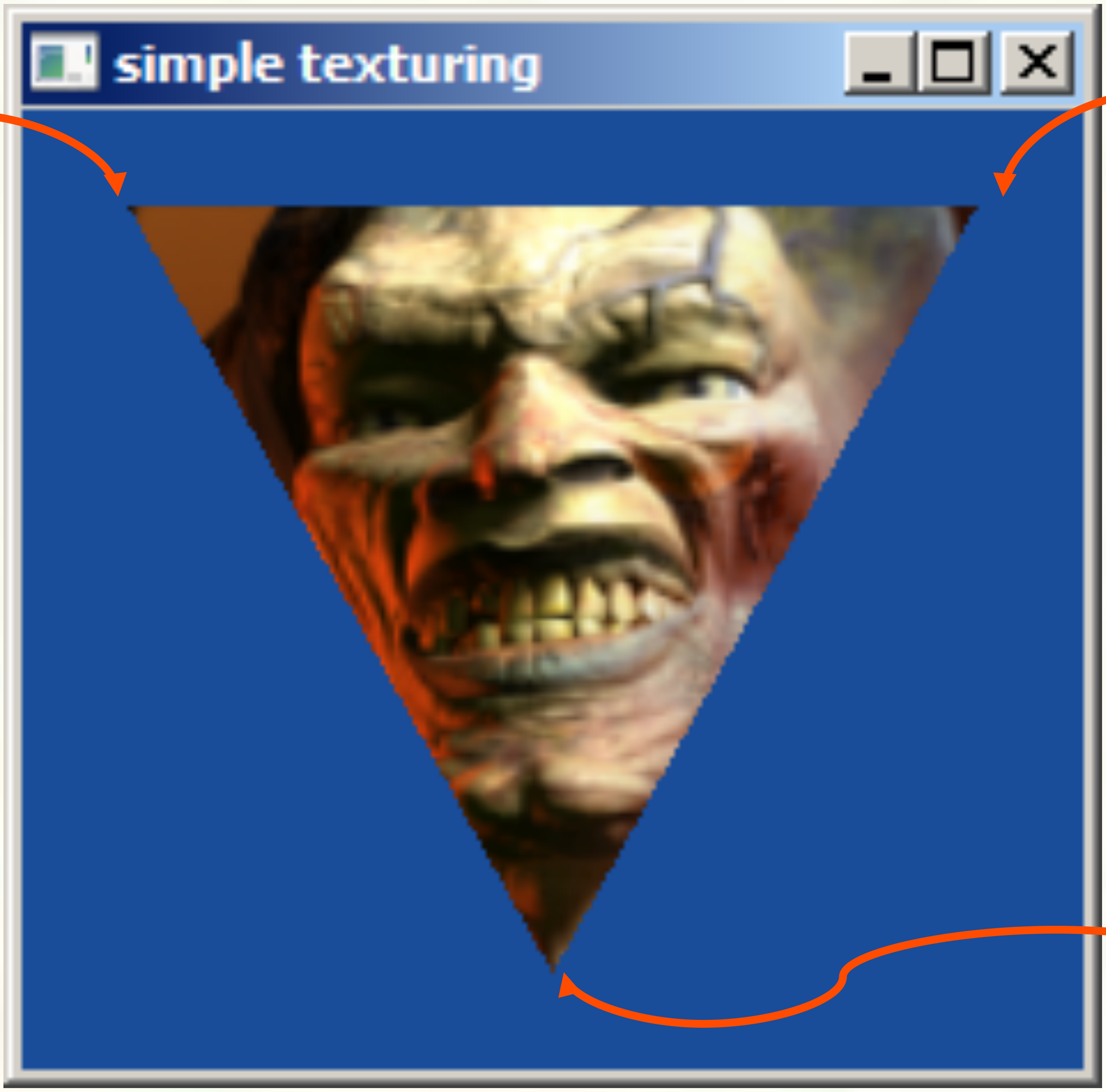
glTexCoord2f like glColor4f but sets "current" texture coordinate instead of color

glMultiTexCoord2f takes texture unit parameter so

glMultiTexCoord2f(GL_TEXTURE0, s,t) same as glTexCoord2f(s,t)

Vertex Texture Coordinates

XYZ = (-0.8,0.8)
ST = (0,0)



XYZ = (0.8,0.8)
ST = (1,0)

XYZ = (0,-0.8)
ST = (0.5,1)

Loose Ends of Texture Setup

- Texture object specification

```
static const GLubyte
myDemonTextureImage[3*(128*128)] = {
/* RGB8 image data for a mipmapped 128x128
    demon texture */
#include "demon_image.h"
};

/* Tightly packed texture data. */
glPixelStorei(GL_UNPACK_ALIGNMENT, 1);

glBindTexture(GL_TEXTURE_2D, 666);
/* Load demon decal texture with mipmaps. */
gluBuild2DMipmaps(GL_TEXTURE_2D, GL_RGB8,
  128, 128, GL_RGB, GL_UNSIGNED_BYTE,
    myDemonTextureImage);
glTexParameteri(GL_TEXTURE_2D,
  GL_TEXTURE_MIN_FILTER,
  GL_LINEAR_MIPMAP_LINEAR);
```

- Fixed-function texture binding and enabling

```
glActiveTexture(GL_TEXTURE0);
glTexEnvi(GL_TEXTURE_ENV,
  GL_TEXTURE_ENV_MODE, GL_REPLACE);
glEnable(GL_TEXTURE_2D);
glBindTexture(GL_TEXTURE_2D, 666);
```

gluBuild2DMipmaps calls glTexImage2D on image, then down-samples iteratively 64x64, 32x32, 16x16, 8x8, 4x4, 2x1, and 1x1 images (called mipmap chain)