Texturing

Course web page:

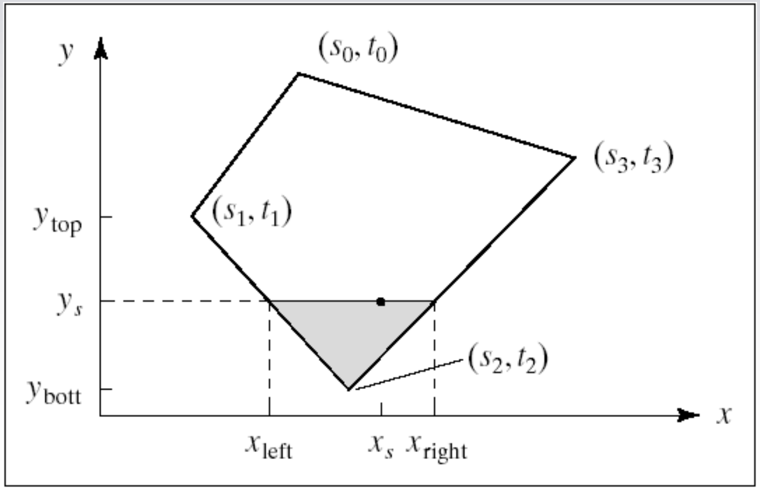
http://goo.gl/EB3aA

Outline

- Perspective-correct texture coordinate interpolation
- Environment mapping
- Shadow mapping
- Magnification/minification

Texture Rasterization

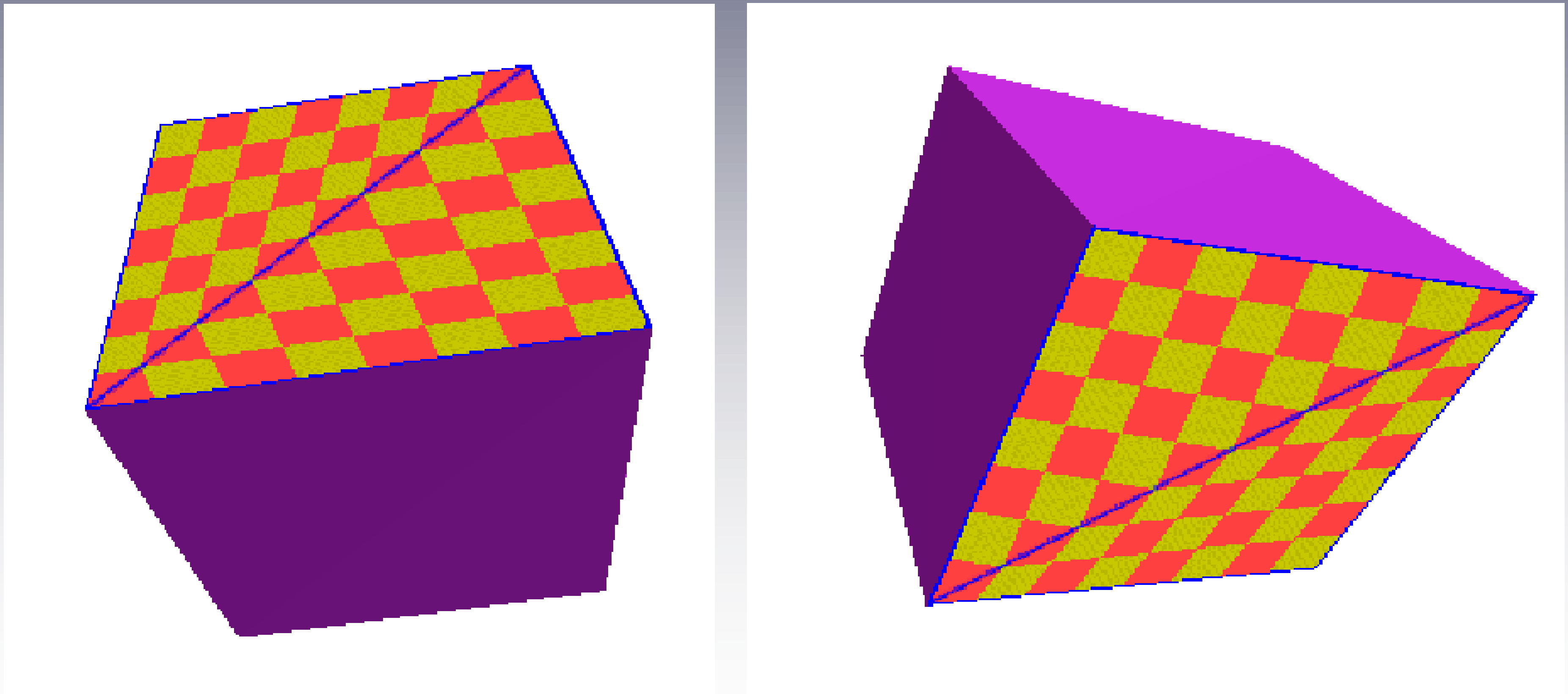
- Okay…we've got texture coordinates for the polygon vertices. What are (s, t) for the pixels inside the polygon?
- Use Gouraud-style linear interpolation of texture coordinates, right?
  - First along polygon edges between vertices
  - Then along scanlines between left and right sides



from Hill

Linear texture coordinate interpolation

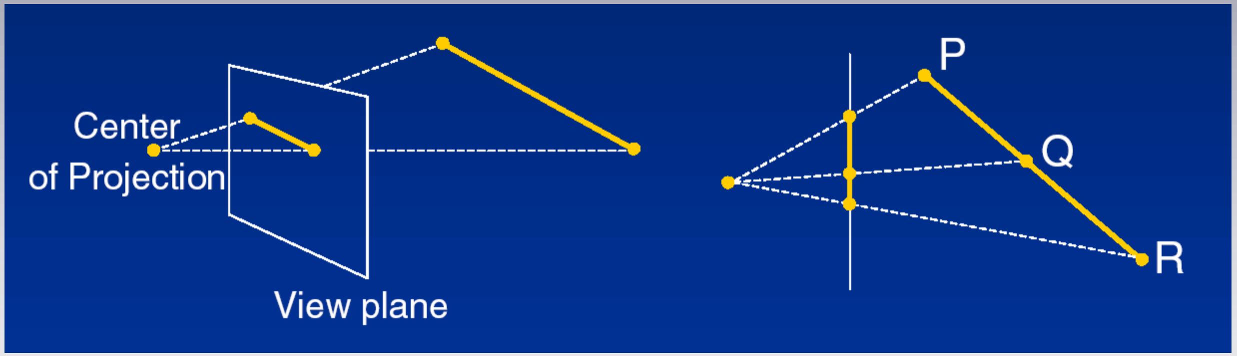
- But this doesn't work!



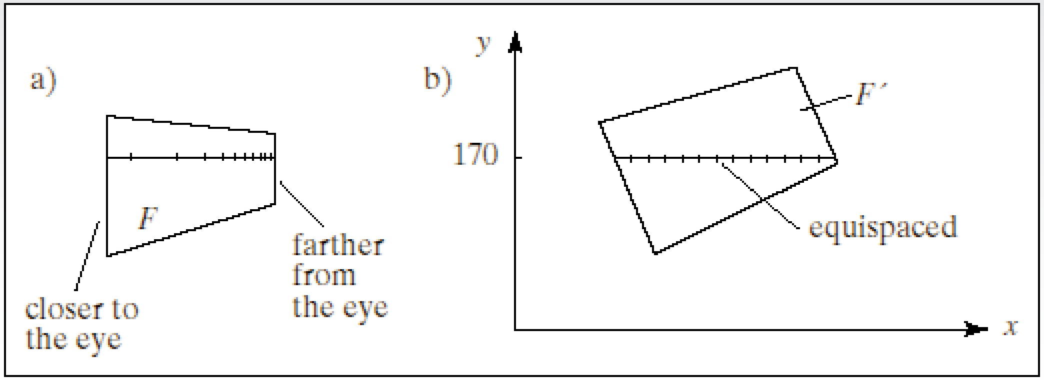
courtesy of H. Pfister

Why not?

- Equally-spaced pixels do **not** project to equally-spaced texels under perspective projection
  - No problem with 2-D affine transforms (rotation, scaling, shear, etc.)
  - But different depths change things due to **foreshortening**



courtesy of H. Pfister

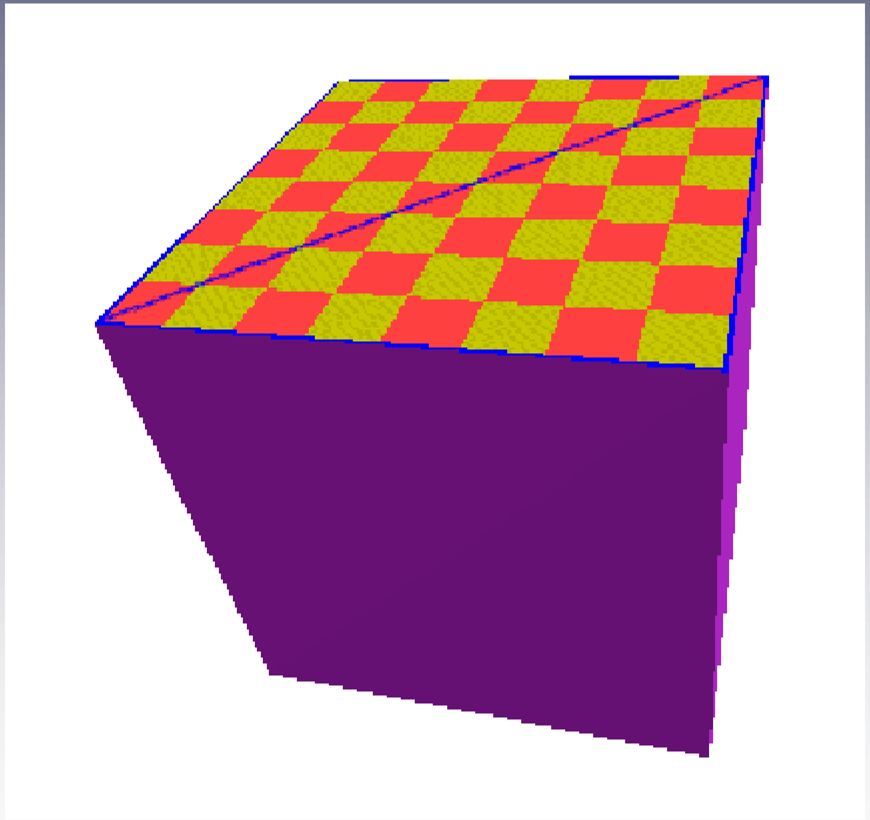


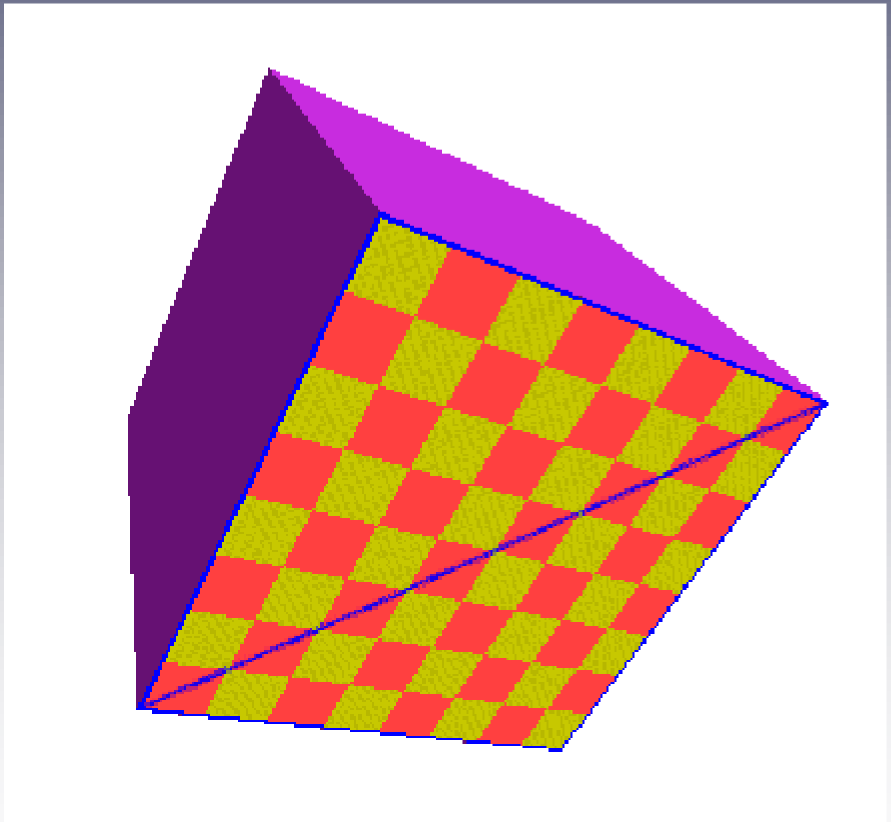
from Hill

Perspective-Correct Texture Coordinate Interpolation

- Compute at each vertex after perspective transformation
  - "Numerators" s/w, t/w
  - "Denominator" 1/w
- Linearly interpolate s/w, t/w, and 1/w across triangle
- At each pixel, perform perspective division of interpolated texture coordinates (s/w, t/w) by interpolated 1/w (i.e., numerator over denominator) to get (s, t)

Perspective-Correct Texture Coordinate Interpolation



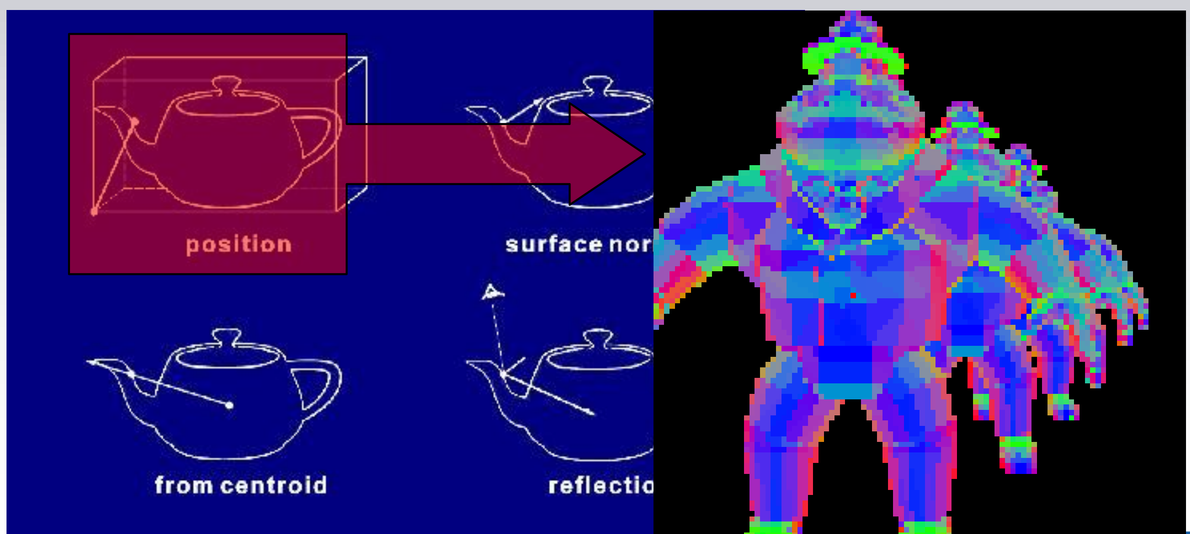


Perspective-Correct Interpolation: Notes

- But we didn't do this for the colors in Gouraud shading…
  - Actually, we should have, but the error is not as obvious
- Alternative: Use regular linear interpolation with small enough polygons that effect is not noticeable
- Linear interpolation for Z-buffering **is** correct

Projecting in non-standard directions

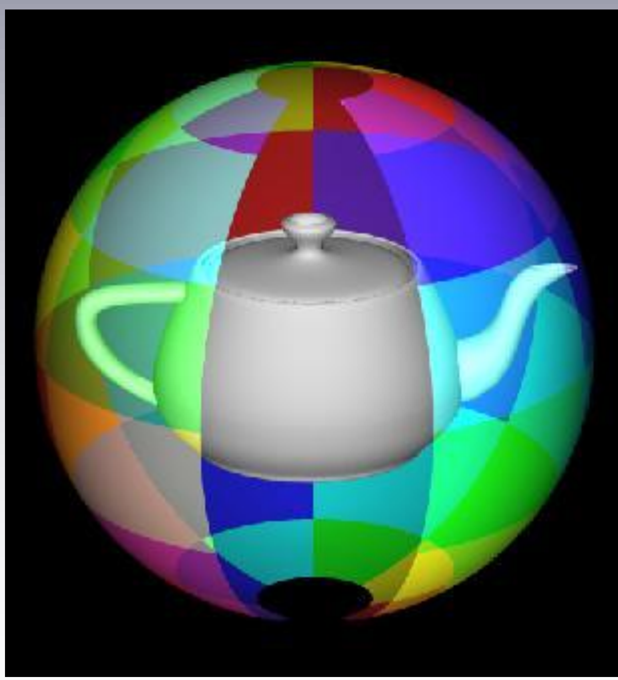
- Texture **projector** function doesn't have to project ray from object center through position (x, y, z)—can use any attribute of that position. For example:
  - Ray comes from another location
  - Ray is surface normal **n** at (x, y, z)
  - Ray is reflection-from-eye vector **r** at (x, y, z)
  - Etc.

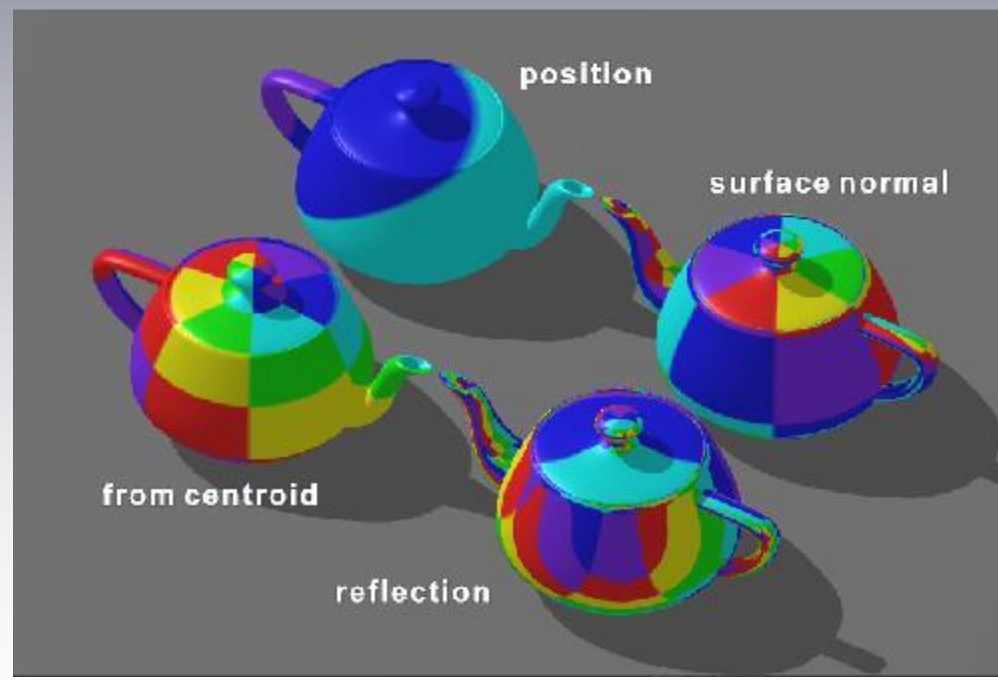


courtesy of R. Wolfe

Projecting in non-standard directions

- This can lead to interesting or informative effects



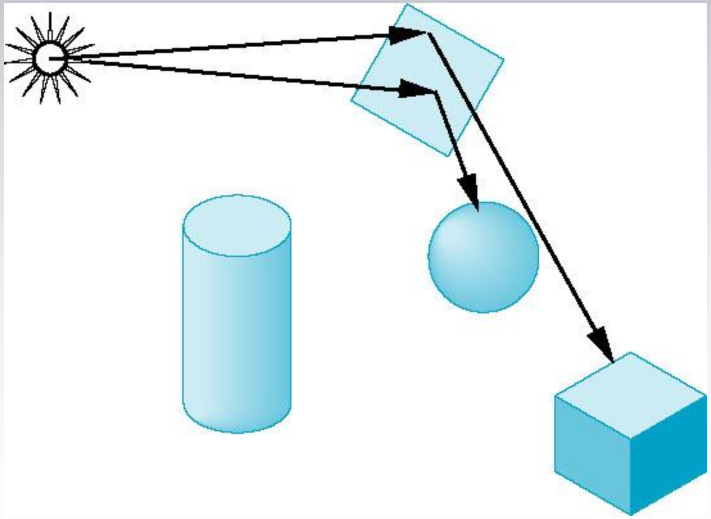


courtesy of R. Wolfe

Different ray directions for a spherical projector

Environment/Reflection Mapping

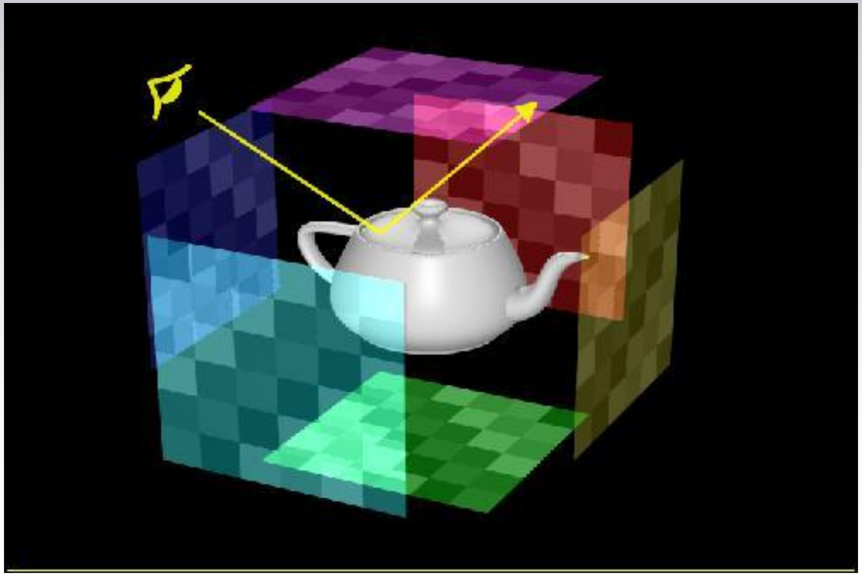
- Problem: To render pixel on mirrored surface correctly, we need to follow reflection of eye vector back to first intersection with another surface and get its color
- This is an expensive procedure with ray tracing
- Idea: Approximate with texture mapping



from Angel

Environment mapping: Details

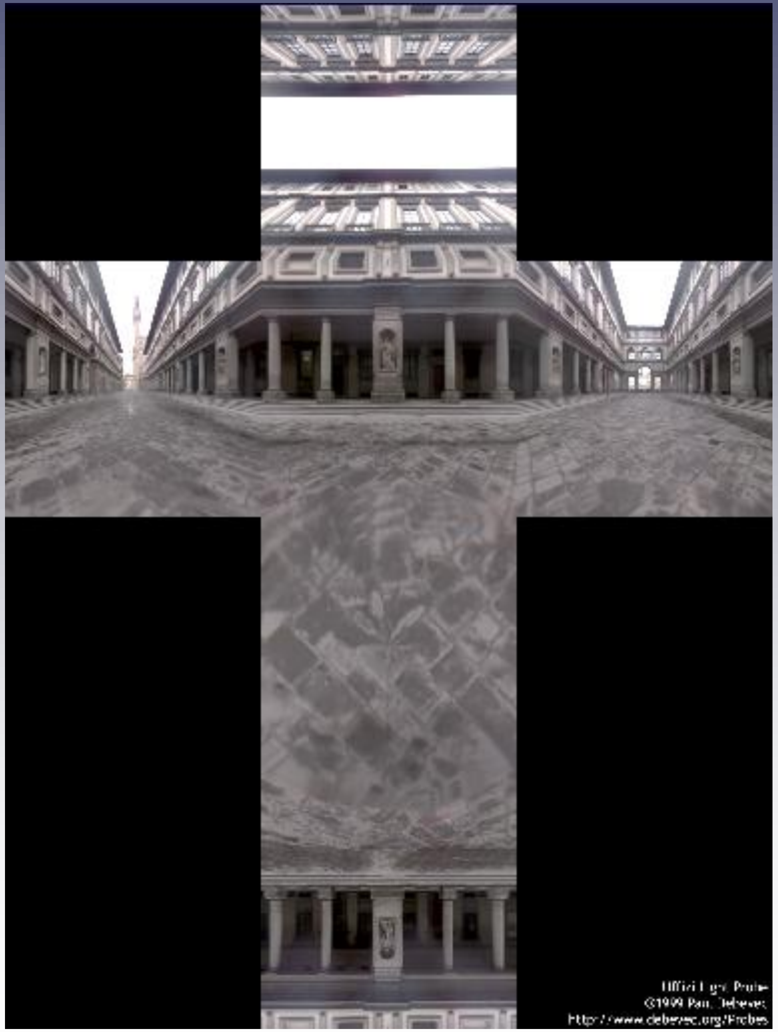
- Key idea: Render 360 degree view of environment **from center of object** with sphere or box as intermediate surface
- Intersection of eye reflection vector with intermediate surface provides texture coordinates for reflection/environment mapping



courtesy of R. Wolfe

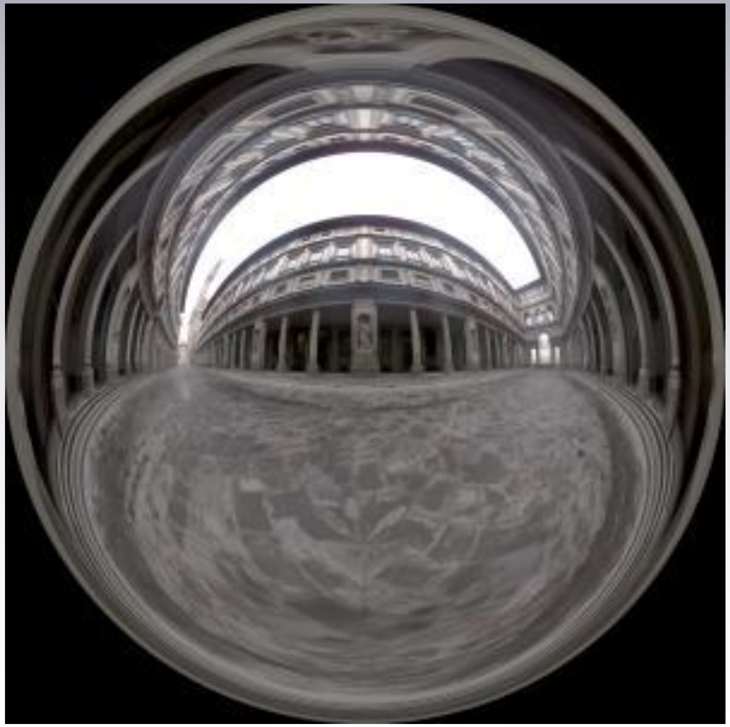
Making environment textures: Cube

- Cube map straightforward to make: Render/ photograph six rotated views of environment
  - 4 side views at compass points
  - 1 straight-up view, 1 straight-down view



Making environment textures: Sphere

- Most often constructed with two photographs of mirrored sphere taken 90 degrees apart



courtesy of P. Debevec

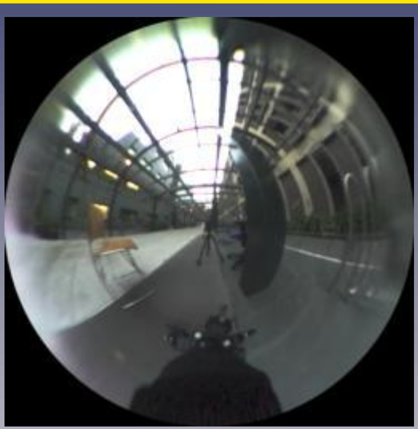
Environment mapping: Example

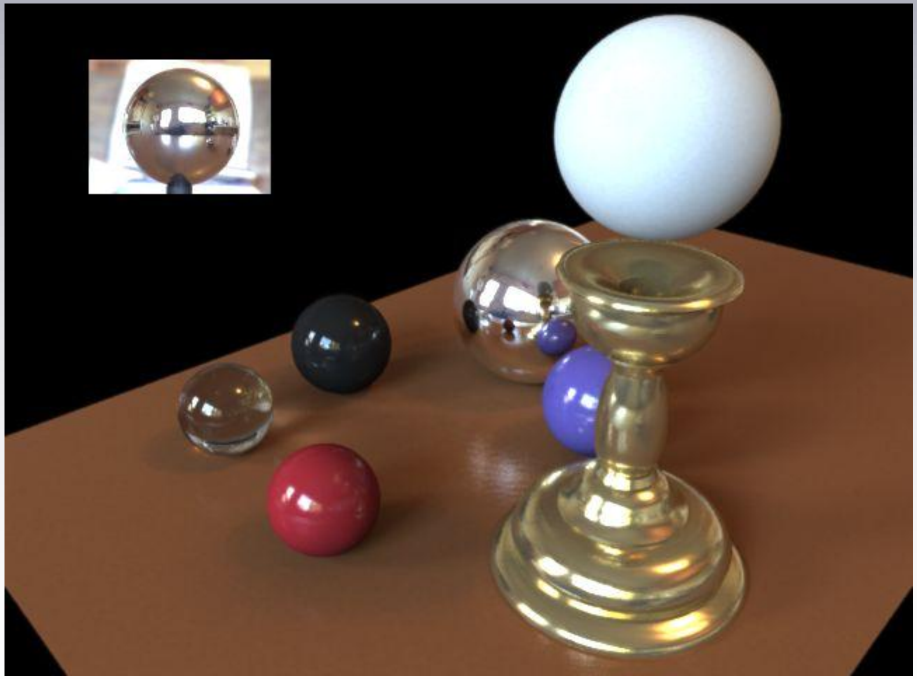
courtesy of G. Miller

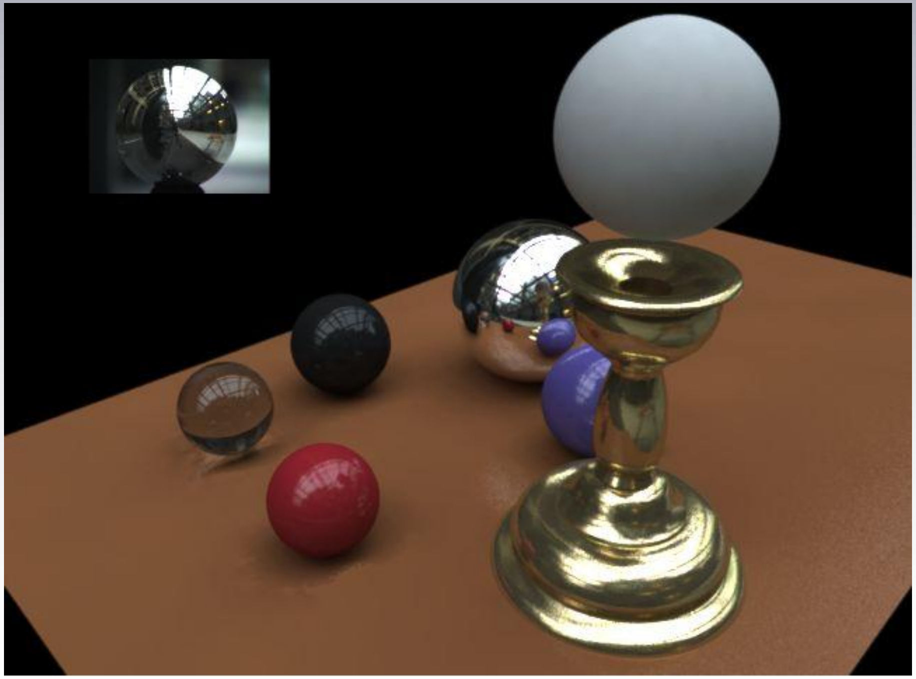
Environment mapping: Example

From "Terminator II"

Environment mapping example: Same scene, different lighting



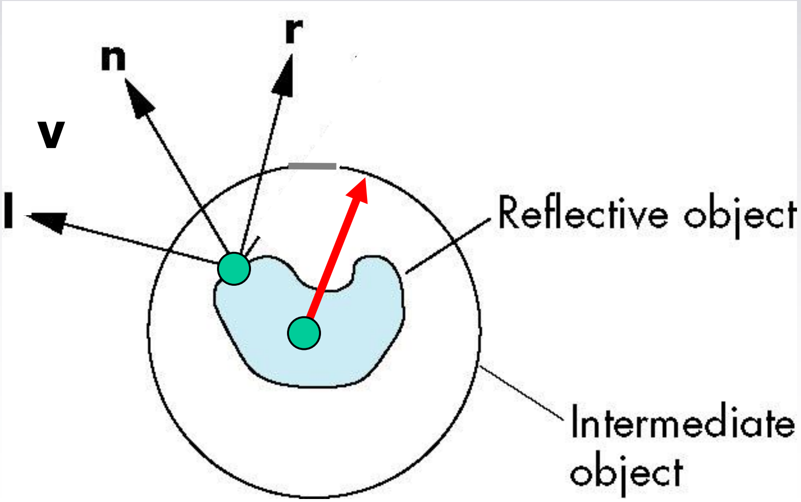




courtesy of P. Debevec

Environment mapping: Issues

- Only physically correct under assumptions that object shape is convex and radiance comes from infinite distance
  - Object concavities mean self-reflections, which won't show up
  - Other objects won't be reflected
  - Parallel reflection vectors access same environment texel, which is only a good approximation when environment objects are very far from object



from Angel

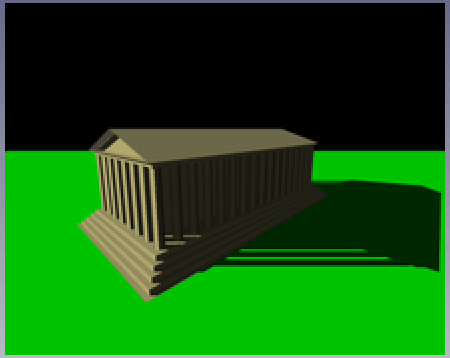
Environment Bump Mapping

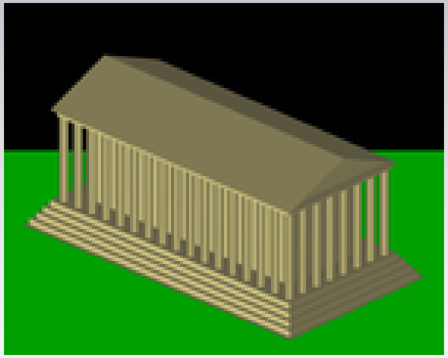
- Idea: Bump map perturbs eye reflection vector

from Akenine-Moller & Haines

Shadow Maps

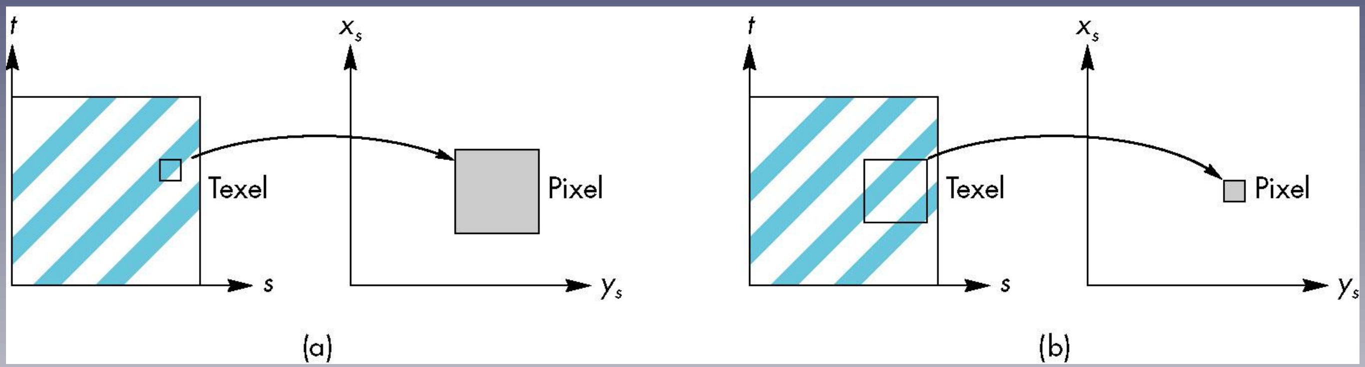
- Idea: If we render scene from point of view of light source, all visible surfaces are lit and hidden surfaces are in shadow
  - "Camera" parameters here determine spotlight characteristics
- When rasterizing scene from eye view, transform each pixel to get 3-D position with respect to the light
  - Project pixel to **shadow buffer** coordinates and compare to z-buffer depth there to see if it is visible
- Shadow edges have aliasing depending on shadow map resolution and scene geometry





University of Delaware

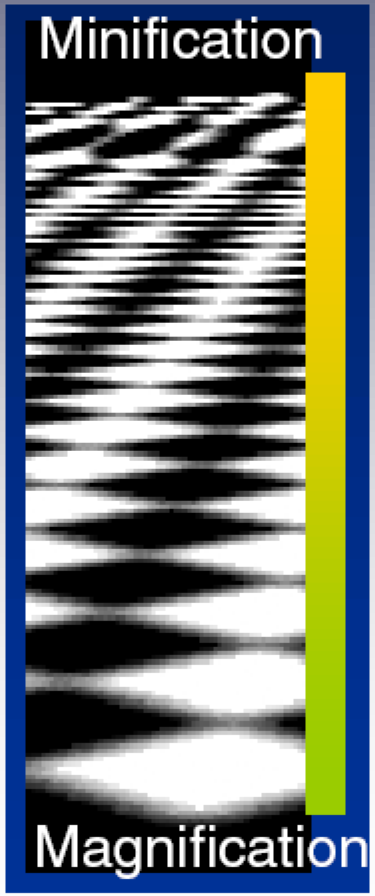
Magnification and minification



from Angel

Magnification

Minification



courtesy of H. Pfister

- **Magnification**: Single screen pixel maps to area less than or equal to one texel
- **Minification**: Single screen pixel area maps to area greater than one texel
  - If texel area covered is much greater than 4, even bilinear filtering isn't so great

Filtering for minification

- Aliasing problem much like line rasterization
  - Pixel maps to quadrilateral (**pre-image**) in texel space

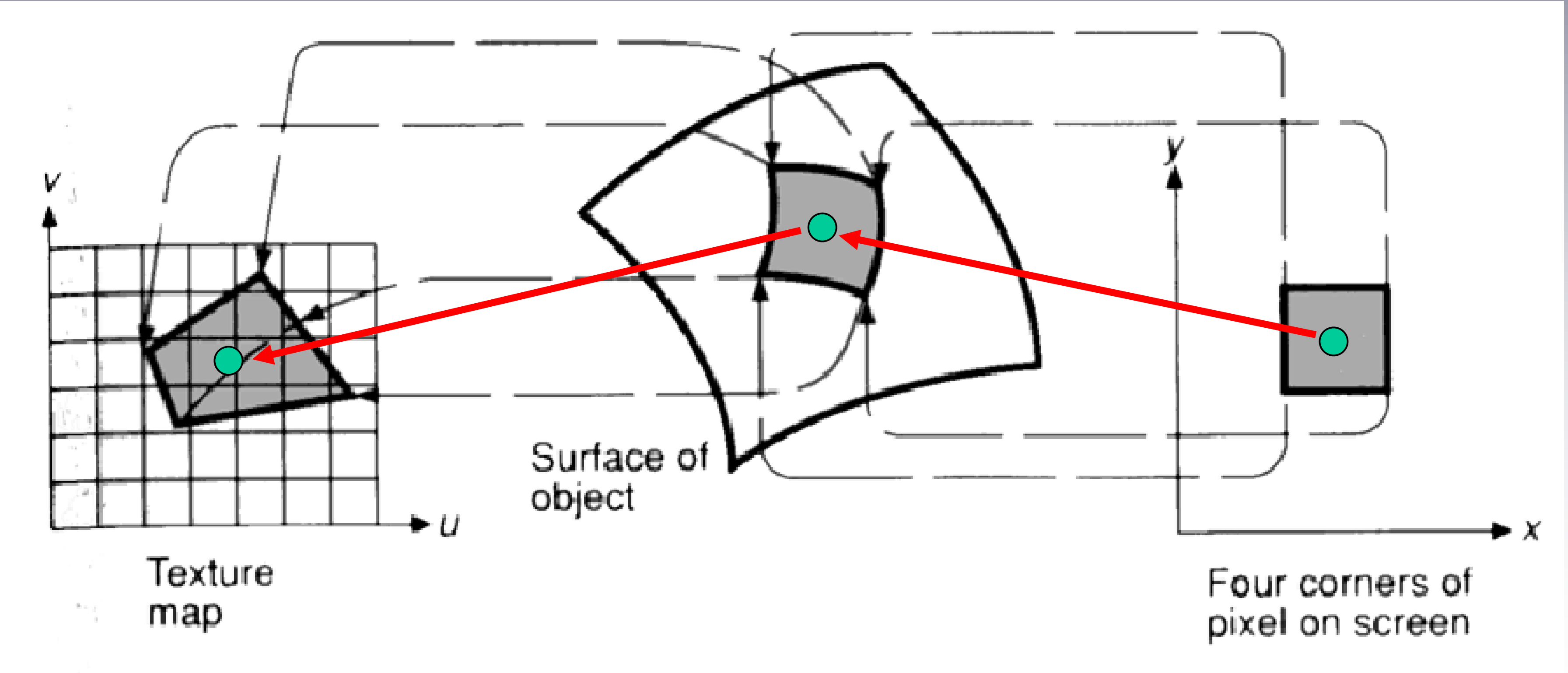
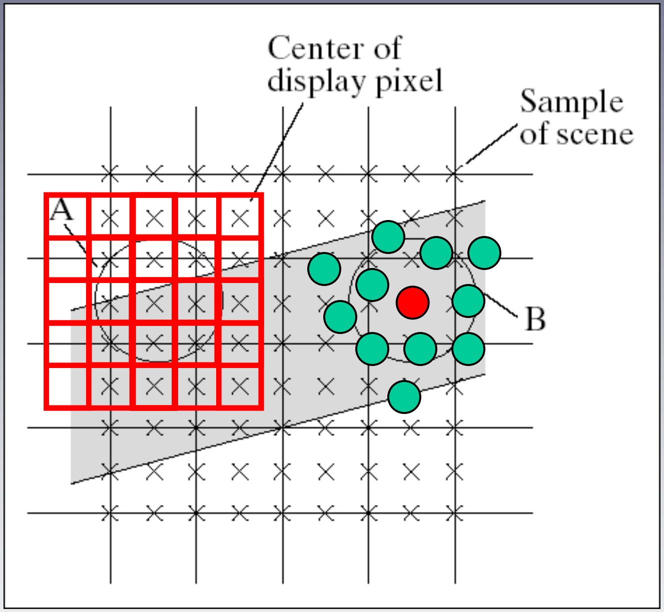


image courtesy of D. Cohen-Or

Supersampling: Using more than BLI's 4 texels

- Rasterize at higher resolution
  - Regular grid pattern around each "normal" image pixel
  - Irregular **jittered** sampling pattern reduces artifacts
- Combine multiple samples into one pixel via **weighted average**
  - "Box" filter: All samples associated with a pixel have equal weight (i.e., directly take their average)
  - Gaussian/cone filter: Sample weights inversely proportional to distance from associated pixel



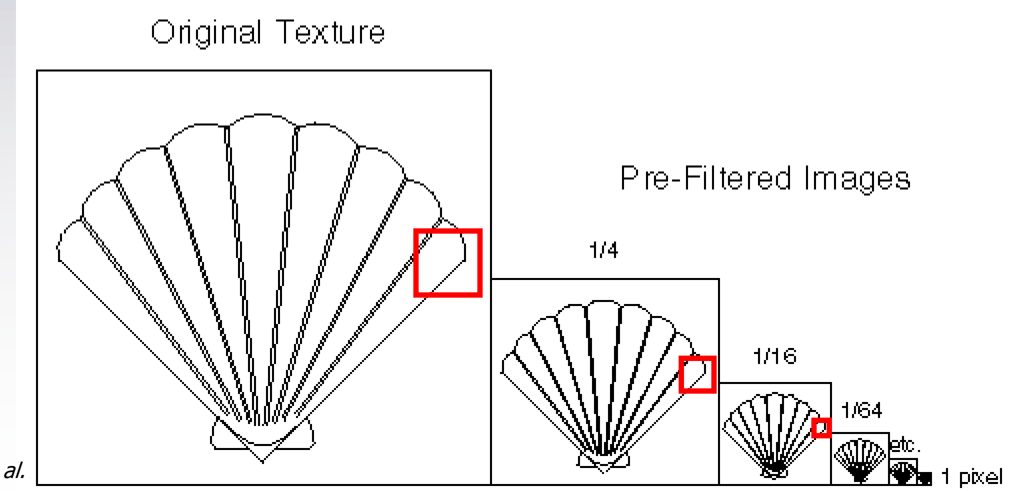
from Hill

Regular supersampling with 2x frequency

Jittered supersampling

Mipmaps

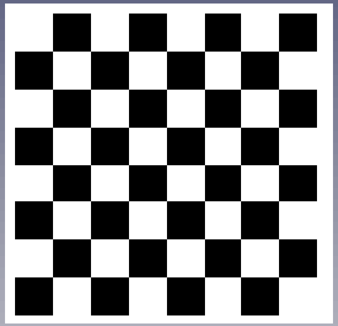
- Filtering for minification is expensive, and different areas must be averaged depending on the amount of minification
- Idea:
  - Prefilter entire texture image at different resolutions
  - For each screen pixel, pick texture in mipmap at **level of detail (LOD)** that minimizes minification (i.e., pre-image area closest to 1)
  - Do nearest or linear filtering in appropriate LOD texture image



from Woo, *et al.*

Create Texture Object

- From where?
  - Create programmatically (aka "procedurally" -- see Red Book Chap. 9 checker.c)
  - Load image from file (e.g., `load_ppm()` in Sprite.cpp)



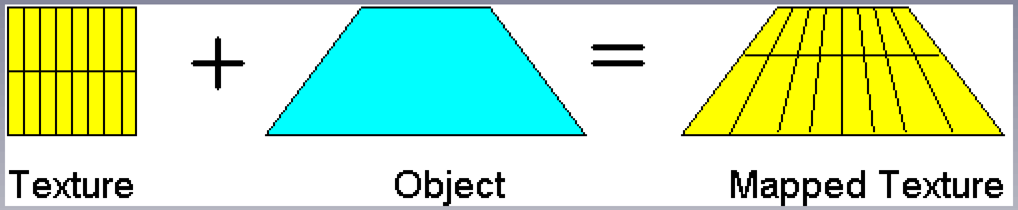
- Name it
  - // Get unused "names" – not mandatory
    glGenTextures(GLsizei *n*, GLuint **textures*)
  - // Create texture object w/ default params (or switch to existing one)
    glBindTexture(GLenum *target*, GLuint *texture*)

- // Store data in bound texture object (no ref because it's global)

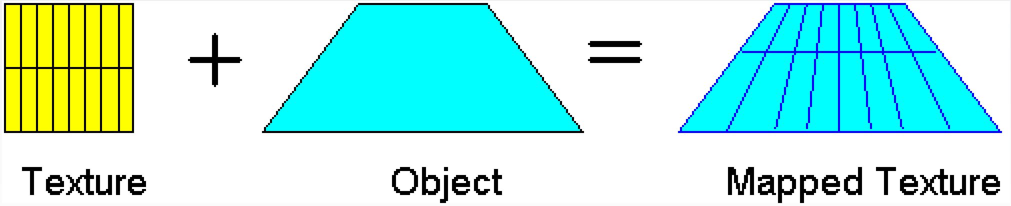
```
glTexImage2D(   GLenum target, GLint level,
                GLint internalFormat,
                GLsizei width, GLsizei height,
                GLint border, GLenum format,
                GLenum type,
                const GLvoid *pixels)
```

Rasterization: Texture application modes

- **decal**: Overwrite object pixel with texel



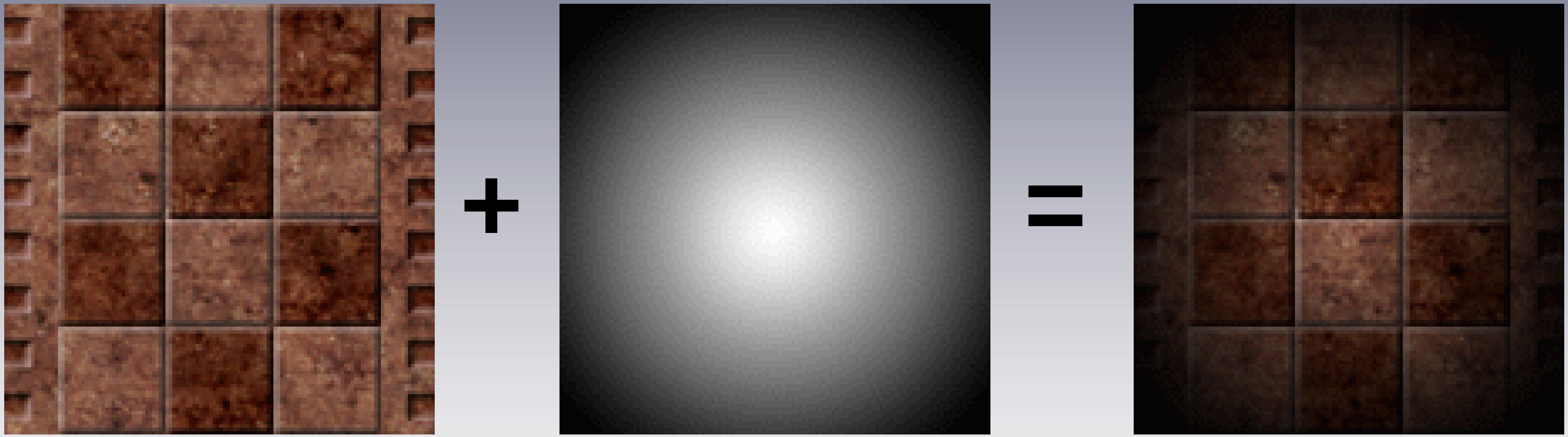
- **modulate**: Combine object pixel with texel via multiplication
  - Need this for multitexturing (i.e., lightmaps)



courtesy of Microsoft

University of Delaware

Texture mapping applications: Lightmaps



courtesy of K. Miller

Texture Application Modes

- **`glTexEnv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE, param)`**, where **`param`** is one of:
  - **`GL_REPLACE`**: Just overwrite surface pixel
  - **`GL_DECAL`**: Use alpha values of surface pixel and texel to blend in standard way
  - **`GL_MODULATE`**: Multiply surface pixel and texel colors
  - **`GL_BLEND`**: Blend surface and texel colors with **`GL_TEXTURE_ENV_COLOR`** (see **`glTexEnv()`** man page for details)
- One thing we're ignoring right now is **wrapping**—the idea of the texture being a repeating pattern

Texture Filtering Parameters

```
GLfloat border_color[ ] = {   1.00 ,   0.00 ,   0.00 ,   1.00 };
GLfloat env_color[ ] = {       0.00 ,   1.00 ,   0.00 ,   1.00 };
glTexParameterfv(GL_TEXTURE_2D, GL_TEXTURE_BORDER_COLOR, border_color);
glTexEnvfv(GL_TEXTURE_ENV, GL_TEXTURE_ENV_COLOR, env_color);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER,  GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER,  GL_NEAREST);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S,  GL_REPEAT);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T,  GL_REPEAT);
glTexEnvi(GL_TEXTURE_ENV, GL_TEXTURE_ENV_MODE,  GL_MODULATE);

glEnable(GL_TEXTURE_2D);
gluBuild2DMipmaps(GL_TEXTURE_2D, 3, w, h, GL_RGB, GL_UNSIGNED_BYTE, image);
glColor4f(   0.60  , 0.60  , 0.60  , 1.00  );
glBegin(GL_POLYGON);
glTexCoord2f( 0.0  , 0.0  );  glVertex3f( -1.0 , -1.0 , 0.0  );
glTexCoord2f( 1.0  , 0.0  );  glVertex3f( 1.0  , -1.0 , 0.0  );
glTexCoord2f( 1.0  , 1.0  );  glVertex3f( 1.0  , 1.0  , 0.0  );
glTexCoord2f( 0.0  , 1.0  );  glVertex3f( -1.0 , 1.0  , 0.0  );
glEnd();
```

Click on the arguments and move the mouse to modify values.

Texturing: Enabling and Drawing

- To draw textured shape, texturing must first be enabled: **`glEnable(GL_TEXTURE_2D)`**
- Load current texture image with **`glTexImage2D()`**
  - Width, height must be powers of 2 (plus 2 if border is used)
  - Only one texture current; faster to change textures by preloading all and switching with **`glBindTexture()`** rather than reloading each time (this is what Sprite.cpp does)
- Assign texture coordinates (s, t) to vertices with **`glTexCoord()`**
  - Similar to **`glColor()`** command—sets a property for subsequent vertices that holds until it is changed