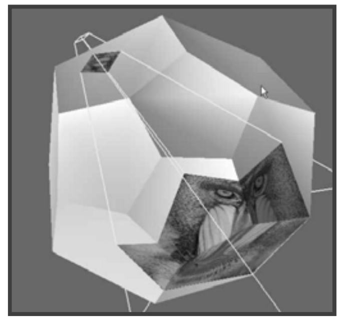
Shadow Mapping in OpenGL

Render Scene and Access the Depth Texture

- Realizing the theory in practice
  - Fragment's light position can be generated using eye-linear texture coordinate generation
    - specifically OpenGL's GL_EYE_LINEAR texgen
    - generate homogenous (s, t, r, q) texture coordinates as light-space (x, y, z, w)
    - T&L engines such as GeForce accelerate texgen!
    - relies on <u>projective texturing</u>

What is Projective Texturing?

- An intuition for projective texturing
  - The slide projector analogy



Source: Wolfgang Heidrich [99]

3

About Projective Texturing (1)

- First, what is perspective-correct texturing?
  - Normal 2D texture mapping uses (s, t) coordinates
  - 2D perspective-correct texture mapping
    - means (s, t) should be interpolated linearly in eye-space
    - so compute per-vertex s/w, t/w, and 1/w
    - linearly interpolate these three parameters over polygon
    - per-fragment compute s’ = (s/w) / (1/w) and t’ = (t/w) / (1/w)
    - results in per-fragment perspective correct (s’, t’)

4

About Projective Texturing (2)

- So what is projective texturing?
  - Now consider homogeneous texture coordinates
    - (s, t, r, q) --> (s/q, t/q, r/q)
    - Similar to homogeneous clip coordinates where (x, y, z, w) = (x/w, y/w, z/w)
  - Idea is to have (s/q, t/q, r/q) be projected per-fragment
  - This requires a per-fragment divider
    - yikes, dividers in hardware are fairly expensive

5

About Projective Texturing (3)

- Hardware designer's view of texturing
  - Perspective-correct texturing is a practical requirement
    - otherwise, textures "swim"
    - perspective-correct texturing already requires the hardware expense of a per-fragment divider
  - Clever idea [Segal, et.al. '92]
    - interpolate q/w instead of simply 1/w
    - so projective texturing is practically free if you already do perspective-correct texturing!

6

About
Projective Texturing (4)

- Tricking hardware into doing projective textures
  - By interpolating q/w, hardware computes per-fragment
    - (s/w) / (q/w) = s/q
    - (t/w) / (q/w) = t/q
  - Net result: projective texturing
    - OpenGL specifies projective texturing
    - only overhead is multiplying 1/w by q
    - but this is per-vertex

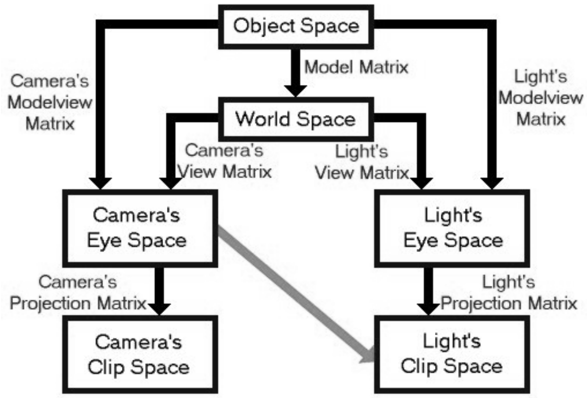
7

Back to the Shadow
Mapping Discussion . . .

- Assign light-space texture coordinates via texgen
  - Transform eye-space (x, y, z, w) coordinates to the light's view frustum (match how the light's depth map is generated)
  - Further transform these coordinates to map directly into the light view's depth map
  - Expressible as a projective transform
    - load this transform into the 4 eye linear plane equations for S, T, and Q coordinates
  - (s/q, t/q) will map to light's depth map texture

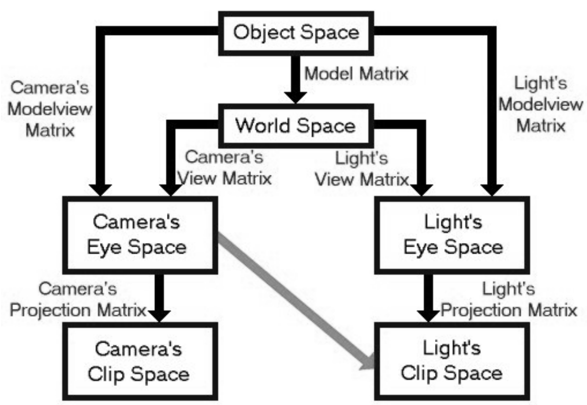
8

Tricks



9

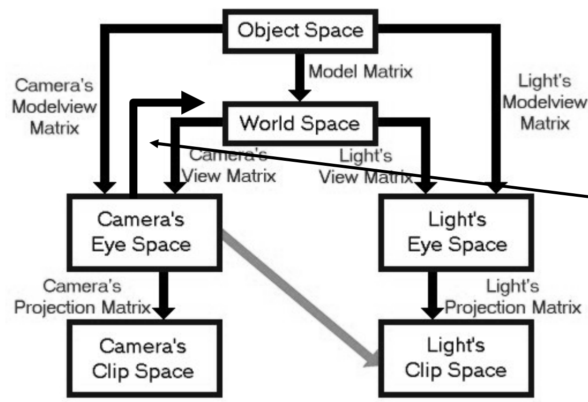
Tricks



$T = P_l \times V_l \times V_c^{-1}$

10

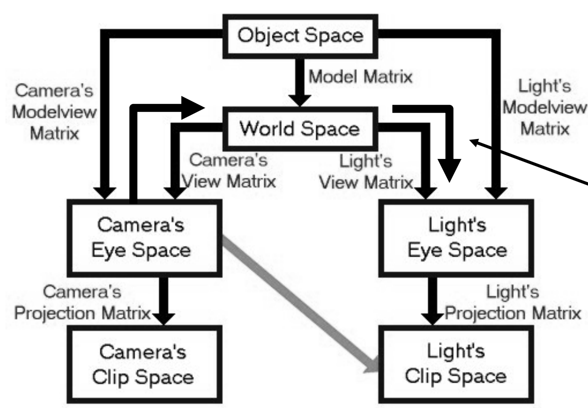
Tricks



$T = P_l \times V_l \times V_c^{-1}$

11

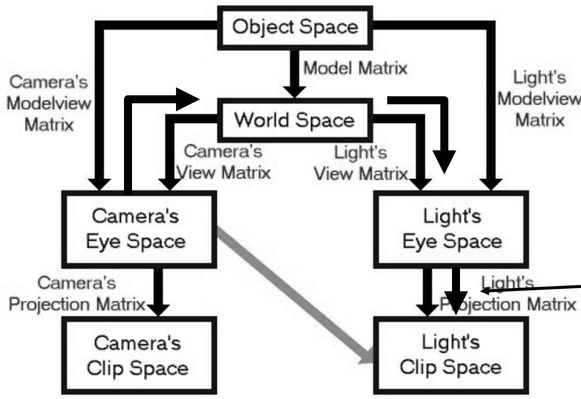
Tricks



$T = P_l \times V_l \times V_c^{-1}$

12

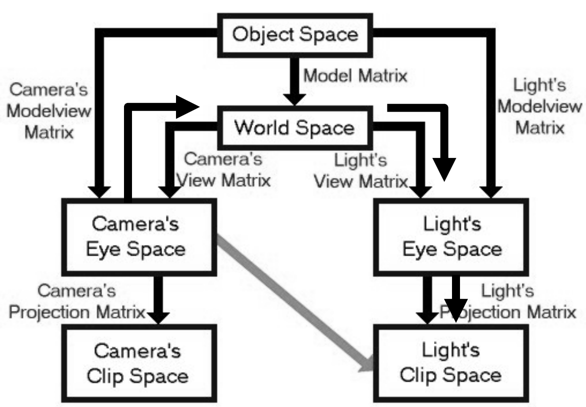
Tricks



$T = P_l \times V_l \times V_c^{-1}$

13

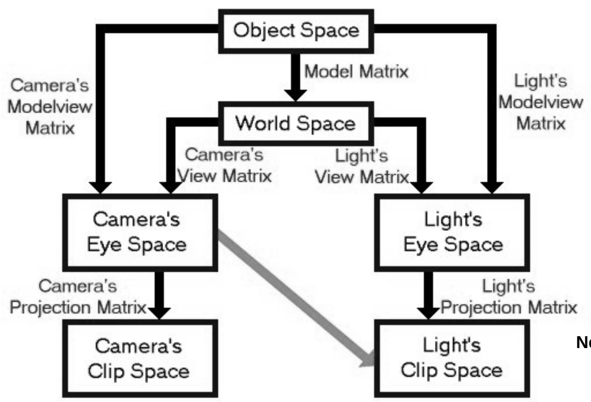
Tricks



$T = P_l \times V_l \times V_c^{-1}$

14

Tricks



$T = P_l \times V_l \times V_c^{-1}$

Need to scale/bias too!

15

Shadow Map Eye Linear Texgen Transform

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

glTexGen **automatically applies this when modelview matrix contains just the eye view transform**

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \underbrace{\begin{bmatrix} 1/2 & & & 1/2 \\ & 1/2 & & 1/2 \\ & & 1/2 & 1/2 \\ & & & 1 \end{bmatrix} \begin{bmatrix} \textit{Light frustum (projection) matrix} \end{bmatrix} \begin{bmatrix} \textit{Light view (look at) matrix} \end{bmatrix}}_{\text{Supply this combined transform to } \textit{glTexGen}} \overbrace{\begin{bmatrix} \textit{Inverse eye view (look at) matrix} \end{bmatrix}} \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$

16

Setting Up
Eye Linear Texgen

- With OpenGL

```
GLfloat Splane[4], Tplane[4], Rplane[4], Qplane[4];
glTexGenfv(GL_S, GL_EYE_PLANE, Splane);
glTexGenfv(GL_T, GL_EYE_PLANE, Tplane);
glTexGenfv(GL_R, GL_EYE_PLANE, Rplane);
glTexGenfv(GL_Q, GL_EYE_PLANE, Qplane);
glEnable(GL_TEXTURE_GEN_S);
glEnable(GL_TEXTURE_GEN_T);
glEnable(GL_TEXTURE_GEN_R);
glEnable(GL_TEXTURE_GEN_Q);
```

- Each eye plane equation is transformed by current inverse modelview matrix
  - Very handy thing for us; otherwise, a pitfall
  - Note: texgen object planes are *not* transformed by the inverse modelview (MISTAKE IN REDBOOK!)

17

Eye Linear
Texgen Transform

- Plane equations form a projective transform

$$\begin{bmatrix} s \\ t \\ r \\ q \end{bmatrix} = \begin{bmatrix} Splane[0] & Splane[1] & Splane[2] & Splane[3] \\ Tplane[0] & Tplane[1] & Tplane[2] & Tplane[3] \\ Rplane[0] & Rplane[1] & Rplane[2] & Rplane[3] \\ Qplane[0] & Qplane[1] & Qplane[2] & Qplane[3] \end{bmatrix} \begin{bmatrix} x_e \\ y_e \\ z_e \\ w_e \end{bmatrix}$$

- The 4 eye linear plane equations form a 4x4 matrix
  - No need for the texture matrix!

18

Shadow Map Operation

- **Automatic depth map lookups**
  - **After the eye linear texgen with the proper transform loaded**
    - **(s/q, t/q) is the fragment's corresponding location within the light's depth texture**
    - **r/q is the Z planar distance of the fragment relative to the light's frustum, scaled and biased to [0,1] range**
  - **Next compare texture value at (s/q, t/q) to value r/q**
    - **if texture[s/q, t/q] ≅ r/q then *not shadowed***
    - **if texture[s/q, t/q] < r/q then *shadowed***

19

Shadow Mapping Hardware Support (1)

- **OpenGL now has official standard shadow mapping extensions (in OpenGL 2.x)**
  - **Approved February 2002!**
  - depth_texture **– adds depth texture formats**
  - shadow **– adds "percentage closer" filtering for depth textures**
  - **The two extensions are used together**
- **Based on prior proven SGI proprietary extensions**
  - SGIX_depth_texture
  - SGIX_shadow

20

Shadow Mapping Hardware Support (2)

- SGIX_depth_texture & SGIX_shadow support
  - SGI's RealityEngine & InfiniteReality
  - Brian Paul's Mesa3D OpenGL work-alike
  - NVIDIA's GeForce3, GeForce4 Ti, and Quadro 4 XGL
    - Software emulation for GeForce1 & 2
- extensions now implemented
  - Latest NVIDIA drivers and Mesa 4.0

21

shadow Filtering Mode

- Performs the shadow test as a texture filtering operation
  - Looks up texel at (s/q, t/q) in a 2D texture
  - Compares lookup value to r/q
  - If texel is greater than or equal to r/q, then generate 1.0
  - If texel is less than r/q, then generate 0.0
- Modulate color with result
  - Zero if fragment is shadowed or unchanged color if not

22

shadow API Usage

- Request shadow map filtering with glTexParameter calls
  - glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_COMPARE_MODE,
    GL_COMPARE_R_TO_TEXTURE);
  - Default is GL_NONE for normal filtering
  - Only applies to depth textures
- Also select the comparison function
  - Either GL_LEQUAL (default) or GL_GEQUAL
  - glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);

23

New Depth Texture Internal Texture Formats

- depth_texture supports textures containing depth values for shadow mapping
- Three new internal formats
  - GL_DEPTH_COMPONENT16
  - GL_DEPTH_COMPONENT24
  - GL_DEPTH_COMPONENT32
    (same as 24-bit on GeForce3/4/Xbox)
- Use GL_DEPTH_COMPONENT for your external format
- Work with glCopySubTexImage2D for fast copies from depth buffer to texture
  - NVIDIA optimizes these copy texture paths

24

Depth Texture Details

- Usage example:
```
glCopyTexImage2D(GL_TEXTURE_2D, level=0,
   internalfmt=GL_DEPTH_COMPONENT,
   x=0, y=0, w=256, h=256, border=0);
```
- Then use glCopySubTexImage2D for faster updates once texture internal format initially defined
- Hint: use GL_DEPTH_COMPONENT for your texture internal format
  - Leaving off the “n” precision specifier tells the driver to match your depth buffer’s precision
  - Copy texture performance is optimum when depth buffer precision matches the depth texture precision

Depth Texture Copy Performance

- The more depth values you copy, the slower the performance
  - 512x512 takes 4 times longer to copy than 256x256
  - Tradeoff: better defined shadows require higher resolution shadow maps, but slows copying
- 16-bit depth values copy twice as fast as 24-bit depth values (which are contained in 32-bit words)
  - Requesting a 16-bit depth buffer (even with 32-bit color buffer) and copying to a 16-bit depth texture is faster than using a 24-bit depth buffer
  - Note that using 16-bit depth buffer usually requires giving up stencil

Hardware Shadow Map Filtering

- “Percentage Closer” filtering
  - Normal texture filtering just averages color components
  - Averaging depth values does NOT work
  - Solution [Reeves, SIGGARPH 87]
    - Hardware performs comparison for each sample
    - Then, averages results of comparisons
  - Provides anti-aliasing at shadow map edges
    - Not soft shadows in the umbra/penumbra sense

27

Hardware Shadow Map Filtering Example

GL_NEAREST: blocky

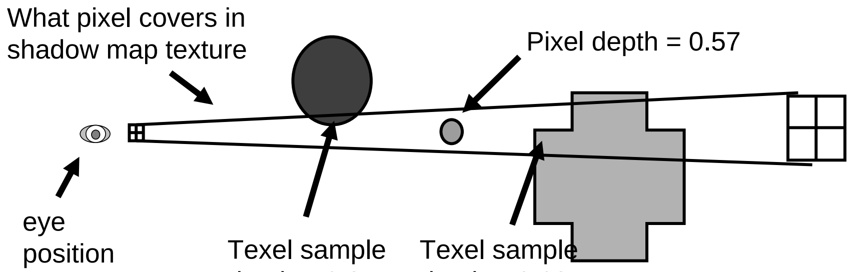
GL_LINEAR: antialiased edges

Low shadow map resolution
used to heightens filtering artifacts

28

Depth Values are not Blend-able

- **Traditional filtering is inappropriate**



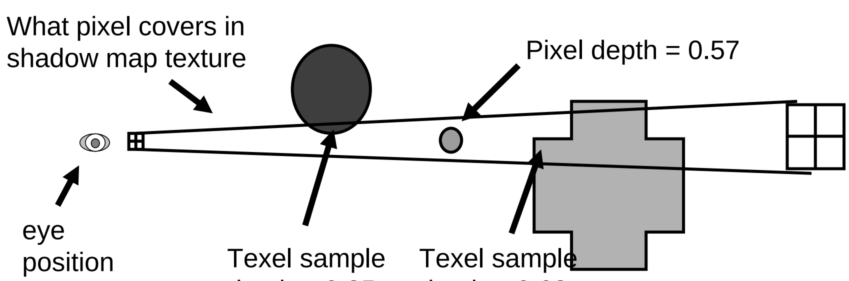
0.25	0.25
0.63	0.63

Average(0.25, 0.25, 0.63, 0.63) = 0.44
0.57 > 0.44 so pixel is **wrongly** “in shadow”
Truth: nothing is at 0.44, just 0.25 and 0.57

29

Percentage Closer Filtering

- **Average comparison *results*, not depth values**



Shadowed	
Unshadowed	

Average(0.57>0.25, 0.57>0.25, 0.57<0.63, 0.57<0.63) = 50%
so pixel is reasonably 50% shadowed
(actually hardware does weighted average)

30

Mipmapping for Depth Textures with Percentage Closer Filtering (1)

- **Mipmap filtering works**
  - **Averages the results of comparisons form the one or two mipmap levels sampled**
- **You *cannot* use gluBuild2DMipmaps to construct depth texture mipmaps**
  - **Again, because you cannot blend depth values!**
- **If you do want mipmaps, the best approach is re-rendering the scene at each required resolution**
  - **Usually too expensive to be practical for all mipmap levels**
  - **OpenGL 1.2 LOD clamping can help avoid rendering all the way down to the 1x1 level**

31

Mipmapping for Depth Textures with Percentage Closer Filtering (2)

- **Mipmaps can make it harder to find an appropriate polygon offset scale & bias that guarantee avoidance of self-shadowing**
- **You can get “8-tap” filtering by using (for example) two mipmap levels, 512x512 and 256x256, and setting your min and max LOD clamp to 0.5**
  - **Uses OpenGL 1.2 LOD clamping**

32

Advice for Shadowed Illumination Model (1)

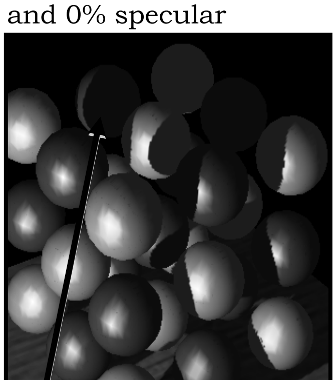
- **Typical illumination model with decal texture:**
  *(ambient + diffuse) * decal + specular*
  The shadow map supplies a shadowing term
- **Assume shadow map supplies a shadowing term, *shade***
  - **Percentage shadowed**
  - **100% = fully visible, 0% = fully shadowed**
- **Obvious updated illumination model for shadowing:**
  *(ambient + shade * diffuse) * decal + shade * specular*
- **Problem is real-world lights don't 100% block diffuse shading on shadowed surfaces**
  - **Light scatters; real-world lights are not ideal points**

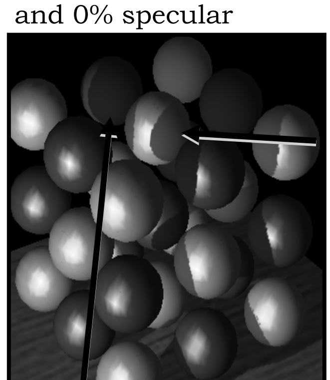
33

The Need for Dimming Diffuse

No dimming; shadowed regions have 0% diffuse and 0% specular

With dimming; shadowed regions have 40% diffuse and 0% specular





No specular in shadowed regions in both versions

Front facing shadowed regions appear unnaturally flat.

Still evidence of curvature in shadowed regions.

34

Advice for Shadowed Illumination Model (2)

- **Illumination model with dimming:**

  (ambient + diffuseShade * diffuse) * decal + specular * shade

  where diffuseShade is

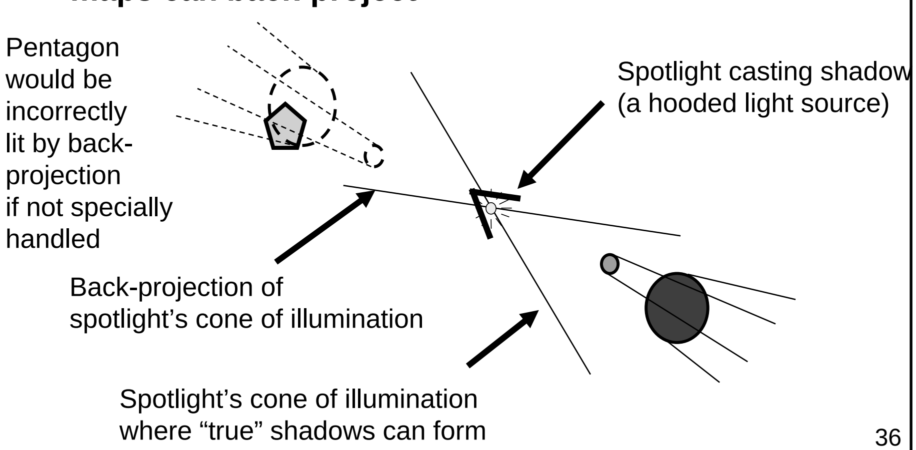
  diffuseShade = dimming + (1.0 – dimming) * shade

  Easy to implement with NV_register_combiners & OpenGL 1.2 "separate specular color" support
- **Separate specular keeps the diffuse & specular per-vertex lighting results distinct**
- **NV_register_combiners can combine the primary (diffuse) and secondary (specular) colors per-pixel with the above math**

35

Careful about Back Projecting Shadow Maps (1)

- **Just like standard projective textures, shadow maps can back-project**



36

Careful about Back Projecting Shadow Maps (2)

- **Techniques to eliminate back-projection:**
  - Modulate shadow map result with lighting result from a single per-vertex spotlight with the proper cut off (ensures light is “off” behind the spotlight)
  - Use a small 1D texture where “s” is planar distance from the light (generate “s” with a planar texgen mode), then 1D texture is 0.0 for negative distances and 1.0 for positive distances.
  - Use a clip plane positioned at the plane defined by the light position and spotlight direction
  - Use the stencil buffer
  - Simply avoid drawing geometry “behind” the light when applying the shadow map (better than a clip plane)
  - NV_texture_shader’s GL_PASS_THROUGH_NV mode

37

Other OpenGL Extensions for Improving Shadow Mapping

- **ARB_pbuffer** – create off-screen rendering surfaces for rendering shadow map depth buffers
  - Normally, you can construct shadow maps in your back buffer and copy them to texture
  - But if the shadow map resolution is larger than your window resolution, use pbuffers.
- **NV_texture_rectangle** – new 2D texture target that does not require texture width and height to be powers of two
  - Limitations
    - No mipmaps or mipmap filtering supported
    - No wrap clamp mode
    - Texture coords in [0..w]x[0..h] rather than [0..1]x[0..1] range.
  - Quite acceptable for for shadow mapping

38

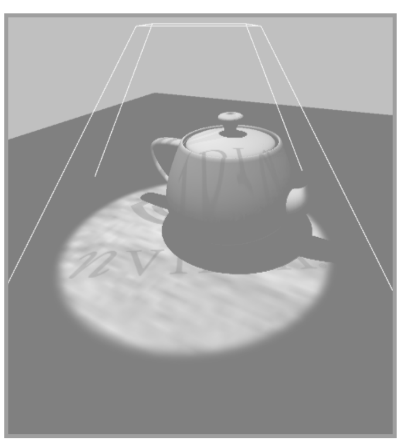
Combining Shadow Mapping with other Techniques

- Good in combination with techniques
  - Use stencil to tag pixels as inside or outside of shadow
  - Use other rendering techniques in extra passes
    - bump mapping
    - texture decals, etc.
  - Shadow mapping can be integrated into more complex multi-pass rendering algorithms
- Shadow mapping algorithm does not require access to vertex-level data
  - Easy to mix with vertex programs and such

39

Combine with Projective Texturing for Spotlight Shadows

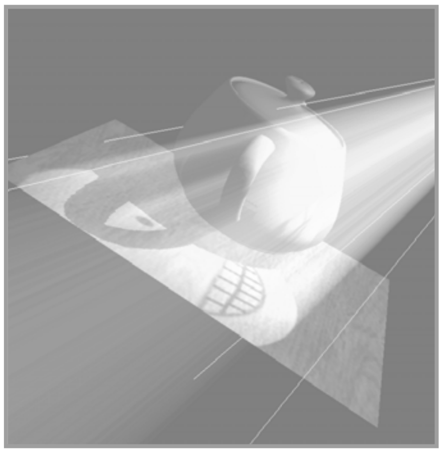
- Use a spotlight-style projected texture to give shadow maps a spotlight falloff



40

Combining Shadows with Atmospherics

- **Shadows in a dusty room**



Simulate atmospheric effects such as suspended dust

1) ***Construct shadow map***
2) ***Draw scene with shadow map***
3) ***Modulate projected texture image with projected shadow map***
4) ***Blend back-to-front shadowed slicing planes also modulated by projected texture image***

Steps for Shadow Mapping

1. Create an empty depth texture
2. Set it up with an internal format of GL DEPTH COMPONENT
3. Set the GL DEPTH TEXTURE MODE to GL LUMINANCE (so it stores the depth internally with a single luminance value) or to GL_INTENSITY
4. Enable the depth buffer
5. Render scene from the light
6. Copy the depth buffer into the texture using glCopyTexImage2D(...)
7. (Optional) Display texture to check that everything so far has worked
8. When we project the shadow map onto the scene, we need to compare the texture with the distance to the light we'll compute at each pixel. Tell OpenGL how to do this comparison by setting (using glTexEnvi(...)) the parameter GL TEXTURE COMPARE FUNC to GL LEQUAL.
9. Tell OpenGL what we want to compare on a per-pixel basis. Set GL TEXTURE COMPARE MODE to GL COMPARE R TO TEXTURE.
10. Tell OpenGL what sort of texture generation to use:
    glTexGeni(GL S, GL TEX GEN MODE, GL EYE LINEAR);
    glTexGeni(GL T, GL TEX GEN MODE, GL EYE LINEAR);
    glTexGeni(GL R, GL TEX GEN MODE, GL EYE LINEAR);
    glTexGeni(GL Q, GL TEX GEN MODE, GL EYE LINEAR);
11. Compute the matrix used to generate texture coordinates. This should be ($S \times P_l \times V_l$), where S is the scale/bias matrix, P_l is the light's projection matrix (the gluProjection(...) matrix you used when rendering the light view), and V_l is the light's view matrix (the gluLookAt(...) matrix you used when rendering the light view).
12. Inside your display function *right after you call gluLookAt(...) for your eye's viewpoint*, setup your texture planes using:
    glTexGenfv(GL S, GL EYE PLANE, plane s);
    glTexGenfv(GL T, GL EYE PLANE, plane t);
    glTexGenfv(GL R, GL EYE PLANE, plane r);
    glTexGenfv(GL Q, GL EYE PLANE, plane q);
    Define GLfloat plane s[4], plane t[4], plane r[4], plane q[4]; and initialize the planes as

$$SP_lV_l = \begin{pmatrix} plane\_s[0] & plane\_s[1] & plane\_s[2] & plane\_s[3] \\ plane\_t[0] & plane\_t[1] & plane\_t[2] & plane\_t[3] \\ plane\_r[0] & plane\_r[1] & plane\_r[2] & plane\_r[3] \\ plane\_q[0] & plane\_q[1] & plane\_q[2] & plane\_q[3] \end{pmatrix}$$

13. Enable texture generation for all four texture coordinates:
    glEnable(GL TEXTURE GEN S);
    glEnable(GL TEXTURE GEN T);
    glEnable(GL TEXTURE GEN R);
    glEnable(GL TEXTURE GEN Q);

Whew!

43