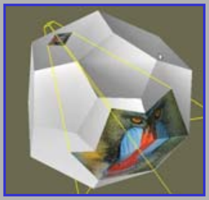
Shadow Mapping in OpenGL

What is Projective Texturing?

- An intuition for projective texturing
  - The slide projector analogy



Source: Wolfgang Heidrich [99]

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’)

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

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!

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

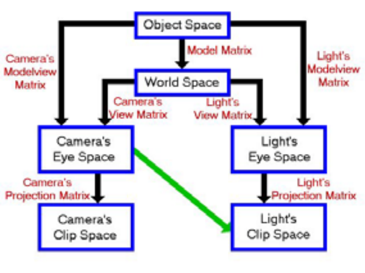
Back to the Shadow Mapping Discussion . . . Fixed Function

- 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

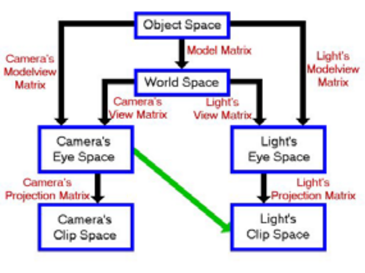
Back to the Shadow Mapping Discussion . . . Fixed Function

- ~~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~~

Tricks

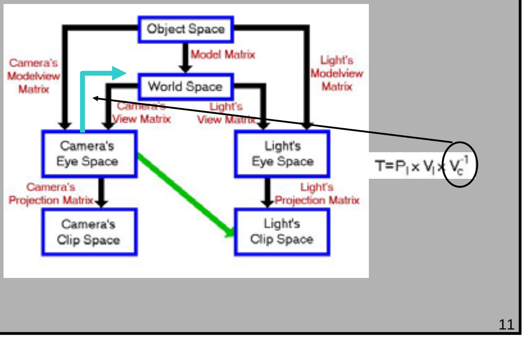


Tricks



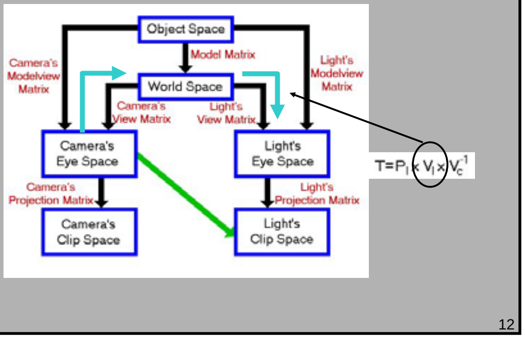
$T=P_l \times V_l \times V_c^{-1}$

Tricks



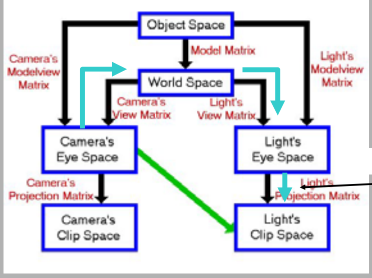
$T=P_l \times V_l \times V_c^{-1}$

Tricks



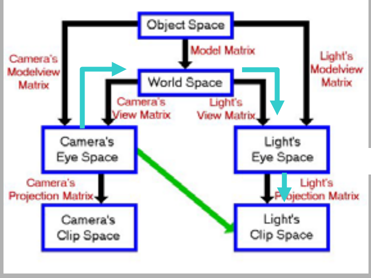
$T=P_l \times V_l \times V_c^{-1}$

Tricks



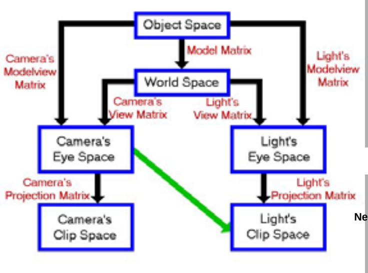
$T=P_l \times V_l \times V_c^{-1}$

Tricks



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

Tricks



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

Need to scale/bias too!

Shadow Map Eye Linear
Texgen Transform (Fixed Function)

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

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

Setting Up
Eye Linear Texgen (Fixed Function)

- 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!)

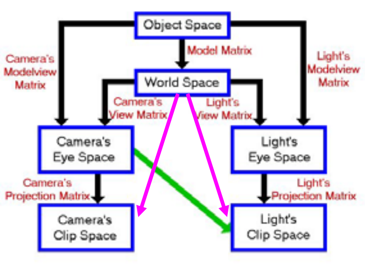
Eye Linear
Texgen Transform (Fixed Function)

- 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!

Tricks



$T=P_l \times V_l$

Still Need to scale/bias!

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*

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

shadow API Usage

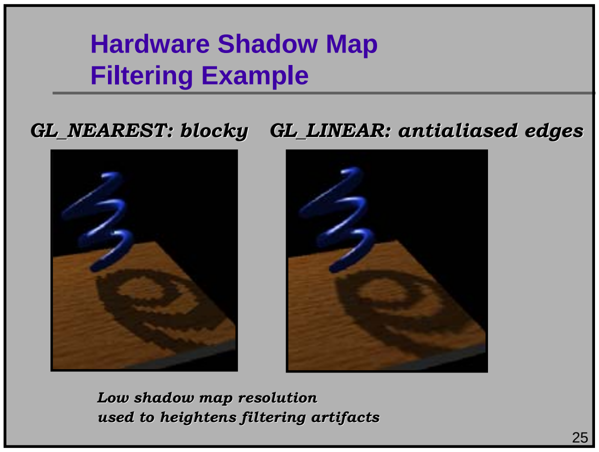
- Request shadow map filtering with glTexParameter calls
  - glTexParameteri(GL_TEXTURE_2D,
    GL_TEXTURE_COMPARE_MODE,
    GL_COMPARE_REF_TO_TEXTURE);
  - Default is GL_NONE for normal filtering
  - Only applies to depth textures
- Also select the comparison function
  - Either GL_LEQUAL (default)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

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)
- 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

Hardware Shadow Map Filtering

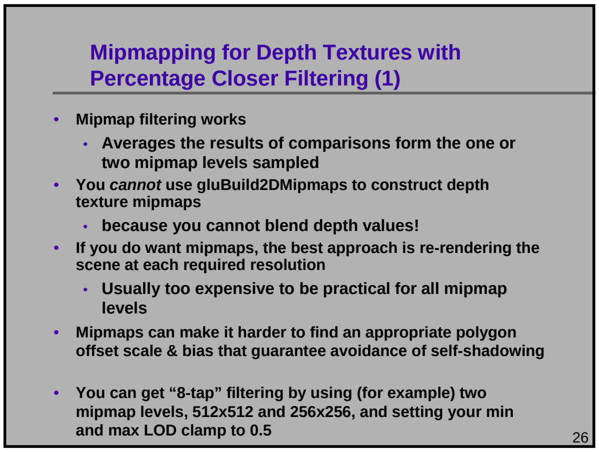
- "Percentage Closer" filtering
  - Normal texture filtering just averages color components
  - Averaging depth values does NOT work
  - Solution [Reeves, SIGGARPH 87]
    - Hardware performs comparison for each sample
    - Then, averages results of comparisons
  - Provides anti-aliasing at shadow map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



Hardware Shadow Map Filtering Example

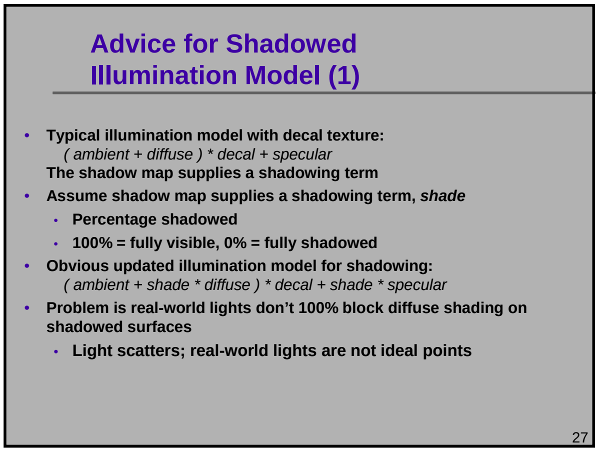
GL_NEAREST: blocky *GL_LINEAR: antialiased edges*

Low shadow map resolution used to heighten filtering artifacts



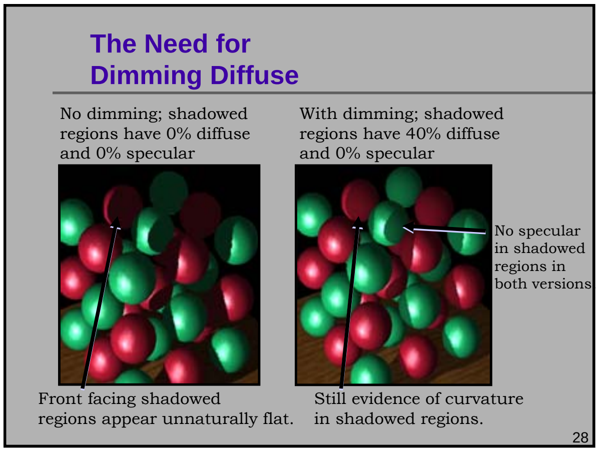
Mipmapping for Depth Textures with Percentage Closer Filtering (1)

- 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
  - 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
- 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



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



The Need for Dimming Diffuse

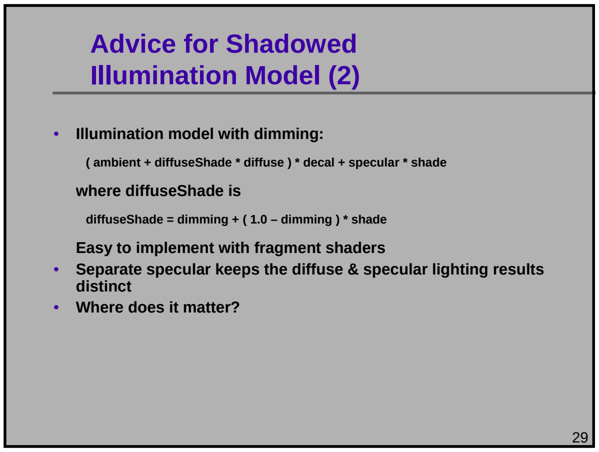
No dimming; shadowed regions have 0% diffuse and 0% specular

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.



Advice for Shadowed Illumination Model (2)

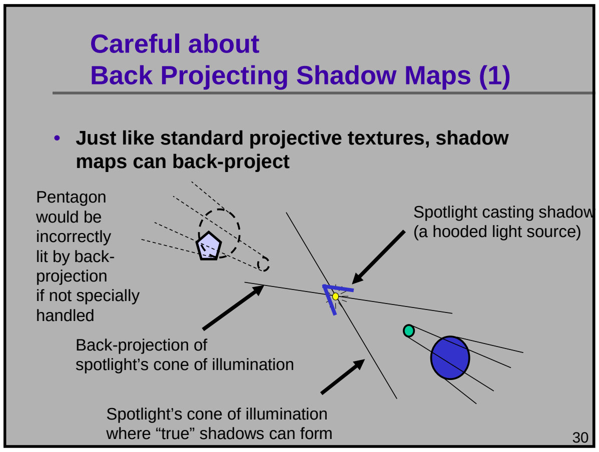
- Illumination model with dimming:

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

  where diffuseShade is

  diffuseShade = dimming + (1.0 – dimming) * shade

  Easy to implement with fragment shaders
- Separate specular keeps the diffuse & specular lighting results distinct
- Where does it matter?



Careful about Back Projecting Shadow Maps (1)

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

Pentagon would be incorrectly lit by back-projection if not specially handled

Spotlight casting shadow (a hooded light source)

Back-projection of spotlight's cone of illumination

Spotlight's cone of illumination where "true" shadows can form

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

Other OpenGL Extensions for Improving Shadow Mapping

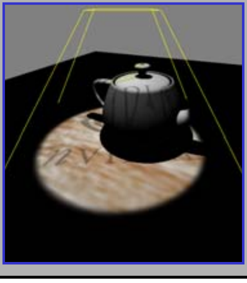
- **FBO** – create off-screen rendering surfaces for rendering shadow map depth buffers
  - Normally, you can construct shadow maps in your back buffer and copy them to texture
  - But if the shadow map resolution is larger than your window resolution, use pbuffers.

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**

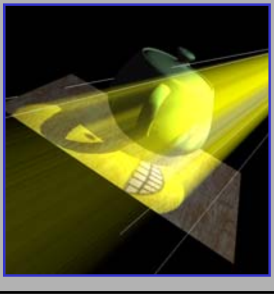
Combine with Projective Texturing for Spotlight Shadows

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



Combining Shadows with Atmospherics

- **Shadows in a dusty room**

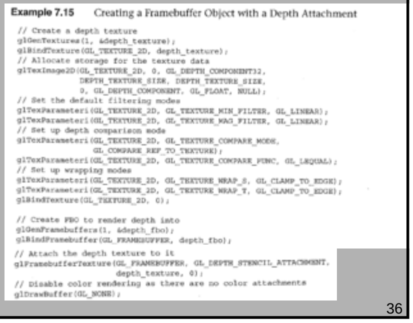


Simulate atmospheric effects such as suspended dust

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 (Fixed Function)

1. Create an empty depth texture
2. Set it up with an internal format of GL DEPTH COMPONENT
3. Set the texture parameters
4. Enable the depth buffer

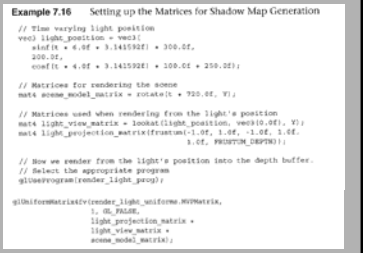


Example 7.15 Creating a Framebuffer Object with a Depth Attachment

```
// Create a depth texture
glGenTextures(1, &depth_texture);
glBindTexture(GL_TEXTURE_2D, depth_texture);
// Allocate storage for the texture data
glTexImage2D(GL_TEXTURE_2D, 0, GL_DEPTH_COMPONENT32,
             DEPTH_TEXTURE_SIZE, DEPTH_TEXTURE_SIZE,
             0, GL_DEPTH_COMPONENT, GL_FLOAT, NULL);
// Set the default filtering modes
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MIN_FILTER, GL_LINEAR);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_MAG_FILTER, GL_LINEAR);
// Set up depth comparison mode
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_MODE,
                GL_COMPARE_REF_TO_TEXTURE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_COMPARE_FUNC, GL_LEQUAL);
// Set up wrapping modes
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_S, GL_CLAMP_TO_EDGE);
glTexParameteri(GL_TEXTURE_2D, GL_TEXTURE_WRAP_T, GL_CLAMP_TO_EDGE);
glBindTexture(GL_TEXTURE_2D, 0);

// Create FBO to render depth into
glGenFramebuffers(1, &depth_fbo);
glBindFramebuffer(GL_FRAMEBUFFER, depth_fbo);
// Attach the depth texture to it
glFramebufferTexture(GL_FRAMEBUFFER, GL_DEPTH_STENCIL_ATTACHMENT,
                     depth_texture, 0);
// Disable color rendering as there are no color attachments
glDrawBuffer(GL_NONE);
```

Steps for Shadow Mapping (Fixed Function)

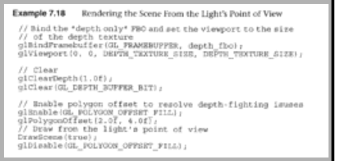
1. Create an empty depth texture
2. Set it up with an internal format of GL DEPTH COMPONENT
3. Set the texture parameters
4. Enable the depth buffer
5. Setup the light matrices


Example 7.16 Setting up the Matrices for Shadow Map Generation

```
// Time varying light position
vec3 light_position = vec3(
    sinf(t * 6.0f * 3.141592f) * 300.0f,
    200.0f,
    cosf(t * 4.0f * 3.141592f) * 100.0f + 250.0f);

// Matrices for rendering the scene
mat4 scene_model_matrix = rotate(t * 720.0f, Y);

// Matrices used when rendering from the light's position
mat4 light_view_matrix = lookat(light_position, vec3(0.0f), Y);
mat4 light_projection_matrix(frustum(-1.0f, 1.0f, -1.0f, 1.0f,
                                     1.0f, FRUSTUM_DEPTH));

// Now we render from the light's position into the depth buffer.
// Select the appropriate program
glUseProgram(render_light_prog);

glUniformMatrix4fv(render_light_uniforms.MVPMatrix,
                   1, GL_FALSE,
                   light_projection_matrix *
                   light_view_matrix *
                   scene_model_matrix);
```

Steps for Shadow Mapping (Fixed Function)

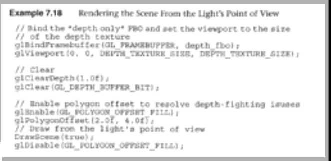
1. Create an empty depth texture
2. Set it up with an internal format of GL DEPTH COMPONENT
3. Set the texture parameters
4. Enable the depth buffer
5. Setup the light matrices
6. Render scene from the light


Example 7.18 Rendering the Scene From the Light's Point of View

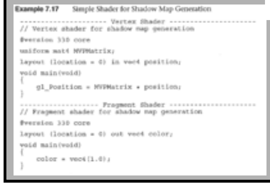
```
// Bind the "depth only" FBO and set the viewport to the size
// of the depth texture
glBindFramebuffer(GL_FRAMEBUFFER, depth_fbo);
glViewport(0, 0, DEPTH_TEXTURE_SIZE, DEPTH_TEXTURE_SIZE);

// Clear
glClearDepth(1.0f);
glClear(GL_DEPTH_BUFFER_BIT);

// Enable polygon offset to resolve depth-fighting issues
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(2.0f, 4.0f);
// Draw from the light's point of view
DrawScene(true);
glDisable(GL_POLYGON_OFFSET_FILL);
```

Steps for Shadow Mapping (Fixed Function)

1. Create an empty depth texture
2. Set it up with an internal format of GL DEPTH COMPONENT
3. Set the texture parameters
4. Enable the depth buffer
5. Setup the light matrices
6. Render scene from the light


Example 7.18 Rendering the Scene From the Light's Point of View

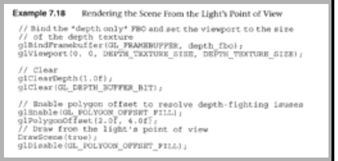
```
// Bind the "depth only" FBO and set the viewport to the size
// of the depth texture
glBindFramebuffer(GL_FRAMEBUFFER, depth_fbo);
glViewport(0, 0, DEPTH_TEXTURE_SIZE, DEPTH_TEXTURE_SIZE);

// Clear
glClearDepth(1.0f);
glClear(GL_DEPTH_BUFFER_BIT);

// Enable polygon offset to resolve depth-fighting issues
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(2.0f, 4.0f);
// Draw from the light's point of view
DrawScene(true);
glDisable(GL_POLYGON_OFFSET_FILL);
```


Example 7.17 Simple Shader for Shadow Map Generation

```
--------------------- Vertex Shader ---------------------
// Vertex shader for shadow map generation
#version 330 core
uniform mat4 MVPMatrix;
layout (location = 0) in vec4 position;
void main(void)
{
    gl_Position = MVPMatrix * position;
}
-------------------- Fragment Shader --------------------
// Fragment shader for shadow map generation
#version 330 core
layout (location = 0) out vec4 color;
void main(void)
{
    color = vec4(1.0);
}
```

Steps for Shadow Mapping (Fixed Function)

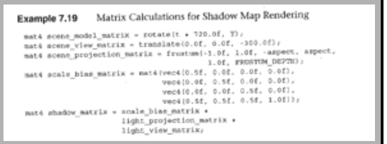
1. Create an empty depth texture
2. Set it up with an internal format of GL DEPTH COMPONENT
3. Set the texture parameters
4. Enable the depth buffer
5. Setup the light matrices
6. Render scene from the light


Example 7.18 Rendering the Scene From the Light's Point of View

```
// Bind the "depth only" FBO and set the viewport to the size
// of the depth texture
glBindFramebuffer(GL_FRAMEBUFFER, depth_fbo);
glViewport(0, 0, DEPTH_TEXTURE_SIZE, DEPTH_TEXTURE_SIZE);

// Clear
glClearDepth(1.0f);
glClear(GL_DEPTH_BUFFER_BIT);

// Enable polygon offset to resolve depth-fighting issues
glEnable(GL_POLYGON_OFFSET_FILL);
glPolygonOffset(2.0f, 4.0f);
// Draw from the light's point of view
DrawScene(true);
glDisable(GL_POLYGON_OFFSET_FILL);
```

Steps for Shadow Mapping (Fixed Function)

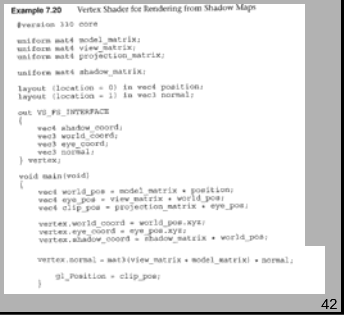
1. Create an empty depth texture
2. Set it up with an internal format of GL DEPTH COMPONENT
3. Set the texture parameters
4. Enable the depth buffer
5. Setup the light matrices
6. Render scene from the light
7. Setup matrices for shadowmapping


Example 7.19 Matrix Calculations for Shadow Map Rendering

```
mat4 scene_model_matrix = rotate(t * 720.0f, Y);
mat4 scene_view_matrix = translate(0.0f, 0.0f, -300.0f);
mat4 scene_projection_matrix = frustum(-1.0f, 1.0f, -aspect, aspect,
                                       1.0f, FRUSTUM_DEPTH);

mat4 scale_bias_matrix = mat4(vec4(0.5f, 0.0f, 0.0f, 0.0f),
                              vec4(0.0f, 0.5f, 0.0f, 0.0f),
                              vec4(0.0f, 0.0f, 0.5f, 0.0f),
                              vec4(0.5f, 0.5f, 0.5f, 1.0f));

mat4 shadow_matrix = scale_bias_matrix *
                     light_projection_matrix *
                     light_view_matrix;
```

Steps for Shadow Mapping (Fixed Function)

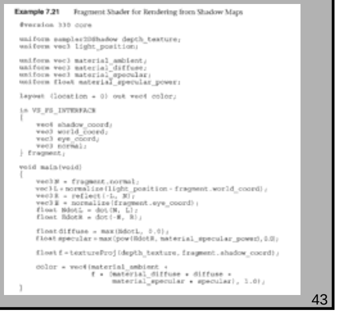
1. Create an empty depth texture
2. Set it up with an internal format of GL DEPTH COMPONENT
3. Set the texture parameters
4. Enable the depth buffer
5. Setup the light matrices
6. Render scene from the light
7. Setup matrices for shadowmapping
8. Render the scene with shadowmapping shaders


Example 7.20 Vertex Shader for Rendering from Shadow Maps

```
#version 330 core

uniform mat4 model_matrix;
uniform mat4 view_matrix;
uniform mat4 projection_matrix;

uniform mat4 shadow_matrix;

layout (location = 0) in vec4 position;
layout (location = 1) in vec3 normal;

out VS_FS_INTERFACE
{
    vec4 shadow_coord;
    vec3 world_coord;
    vec3 eye_coord;
    vec3 normal;
} vertex;

void main(void)
{
    vec4 world_pos = model_matrix * position;
    vec4 eye_pos = view_matrix * world_pos;
    vec4 clip_pos = projection_matrix * eye_pos;

    vertex.world_coord = world_pos.xyz;
    vertex.eye_coord = eye_pos.xyz;
    vertex.shadow_coord = shadow_matrix * world_pos;

    vertex.normal = mat3(view_matrix * model_matrix) * normal;

        gl_Position = clip_pos;
    }
```

Steps for Shadow Mapping (Fixed Function)

1. Create an empty depth texture
2. Set it up with an internal format of GL DEPTH COMPONENT
3. Set the texture parameters
4. Enable the depth buffer
5. Setup the light matrices
6. Render scene from the light
7. Setup matrices for shadowmapping
8. Render the scene with shadowmapping shaders



Example 7.21 Fragment Shader for Rendering from Shadow Maps

```
#version 330 core

uniform sampler2DShadow depth_texture;
uniform vec3 light_position;

uniform vec3 material_ambient;
uniform vec3 material_diffuse;
uniform vec3 material_specular;
uniform float material_specular_power;

layout (location = 0) out vec4 color;

in VS_FS_INTERFACE
{
    vec4 shadow_coord;
    vec3 world_coord;
    vec3 eye_coord;
    vec3 normal;
} fragment;

void main(void)
{
    vec3 N = fragment.normal;
    vec3 L = normalize(light_position - fragment.world_coord);
    vec3 R = reflect(-L, N);
    vec3 E = normalize(fragment.eye_coord);
    float NdotL = dot(N, L);
    float EdotR = dot(-E, R);

    float diffuse = max(NdotL, 0.0);
    float specular = max(pow(EdotR, material_specular_power),0.0);

    float f = textureProj(depth_texture, fragment.shadow_coord);

    color = vec4(material_ambient +
                 f * (material_diffuse * diffuse +
                      material_specular * specular), 1.0);
}
```

Whew!