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-polygon 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
• 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
Setting Up Eye Linear Texgen (Fixed Function)

- With OpenGL
  - `GLfloat Splane[4], Tplane[4], Rplane[4], Qplane[4];`
  - `glTexGenf(GL_S, GL_EYE_PLANE, Splane);`
  - `glTexGenf(GL_T, GL_EYE_PLANE, Tplane);`
  - `glTexGenf(GL_R, GL_EYE_PLANE, Rplane);`
  - `glTexGenf(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 matrix (MISTAKE IN REDBOOK!)

Eye Linear Texgen Transform (Fixed Function)

- Plane equations form a projective transform
  
  \[
  \begin{bmatrix}
  s \\ t \\ r \\ q \\
  \end{bmatrix} = \begin{bmatrix}
  \end{bmatrix} \begin{bmatrix}
  x_o \\ y_o \\ z_o \\ w_o \\
  \end{bmatrix}
  \]

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

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 \(\text{texture}[s/q, t/q] \equiv r/q\) then not shadowed
    - If \(\text{texture}[s/q, t/q] < r/q\) 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
  - glTexParameter(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
    - glTexParameter(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 from the one or two mipmap levels sampled
- You cannot use gluBuild2DMipmaps to construct depth texture mipmap
  - because you cannot blend depth values!
- If you do want mipmap, 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:
  \[(\text{ambient} + \text{diffuse}) \cdot \text{decal} + \text{specular}\]
  The shadow map supplies a shadowing term
- Assume shadow map supplies a shadowing term, \(\text{shade}\)
  - Percentage shadowed
    - 100% = fully visible, 0% = fully shadowed
  - Obvious updated illumination model for shadowing:
    \[(\text{ambient} + \text{shade} \cdot \text{diffuse}) \cdot \text{decal} + \text{shade} \cdot \text{specular}\]
  - Problem is real-world lights don’t 100% block diffuse shading on shadowed surfaces
    - Light scatters; real-world lights are not ideal points

Advice for Shadowed Illumination Model (2)

- Illumination model with dimming:
  \[(\text{ambient} + \text{diffuseShade} \cdot \text{diffuse}) \cdot \text{decal} + \text{specular} \cdot \text{shade}\]
  where \(\text{diffuseShade} = \text{dimming} + (1.0 - \text{dimming}) \cdot \text{shade}\)
  - Easy to implement with fragment shaders
    - Separate specular keeps the diffuse & specular lighting results distinct
    - Where does it matter?

Advice for Shadowed Illumination Model (3)

- Just like standard projective textures, shadow maps can back-project
  - Pentagon would be incorrectly lit by back-projection if not specially handled
    - Back-projection of spotlight’s cone of illumination
    - Spotlight casting shadow (a hooded light source)
  - 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

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

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

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

Combine with Projective Texturing for Spotlight Shadows

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

Steps for Shadow Mapping (Fixed Function)

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

Example 7.10 Creating a Homeworld Object with a Depth Attachment

1. Create a depth anchor:
   - glDepthAnchor (FLOAT, GL_MAX_DEPTH_TEXTURE_DEPTH, 0.5, &depthAnch);
2. Draw scene with depth anchor
3. Modulate projected texture image with projected shadow map
4. Blend back-to-front shadowed slicing planes also modulated by projected texture image

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 buffer.
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 Setting up the Matrices for Shadow Mapping

```c
void setupLightMatrices(void) {
    if (numLights > 0) {
        // Set up light matrix for light 0
        light[0].matrix = glm::mat4(1.0f);
        light[0].matrix = glm::translate(light[0].matrix, light[0].position);
        light[0].matrix = glm::scale(light[0].matrix, light[0].radius);
        light[0].matrix = glm::rotate(light[0].matrix, light[0].rotation, light[0].up);
        // Multiply matrix by the view matrix to get the view matrix.
        light[0].matrix = glm::perspective(light[0].fov, glm::radians(light[0].aspect), light[0].near, light[0].far);
    }
}
```

7. Setup matrices for shadow mapping.
8. Render the scene with shadow mapping shaders.

Example 7.19 Matrix Calculations for Shadow Mapping

```c
void calculateShadowMatrix(void) {
    // Calculate the shadow matrix for each light.
    for (int i = 0; i < numLights; i++) {
        light[i].shadowMatrix = glm::inverse(light[i].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. Enable the depth buffer
4. Setup the light matrices
5. Render scene from the light
6. Enable depth testing
7. Render the scene with shadow mapping shaders
8. Render the scene with shadowmapping shaders

Whew!