Computer Graphics (CS 543)
Lecture 7 (Part 1): Shadows and Fog

Prof Emmanuel Agu

Computer Science Dept.
Worcester Polytechnic Institute (WPI)
Introduction to Shadows

- Shadows give information on relative positions of objects

Use ambient + diffuse + specular components

Use just ambient component
Introduction to Shadows

- Two popular shadow rendering methods:
  1. Shadows as texture (projection)
  2. Shadow buffer
- Third method used in ray-tracing (covered later)
Projective Shadows

- Oldest method: Used in early flight simulators
- Projection of polygon is polygon called **shadow polygon**
Projective Shadows

- Works for flat surfaces illuminated by point light
- For each face, project vertices $V$ to find $V'$ of shadow polygon
- Object shadow = union of projections of faces
Projective Shadow Algorithm

- Project light-object edges onto plane

**Algorithm:**
- First, draw ground plane using specular+diffuse+ambient components
- Then, draw shadow projections (face by face) using only ambient component
Projective Shadows for Polygon

1. If light is at \((x_l, y_l, z_l)\)
2. Vertex at \((x, y, z)\)
3. Would like to calculate shadow polygon vertex \(V\) projected onto ground at \((x_p, 0, z_p)\)

Ground plane: \(y = 0\)
Projective Shadows for Polygon

- If we move original polygon so that light source is at origin
- Matrix $M$ projects a vertex $V$ to give its projection $V'$ in shadow polygon

$$M = \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & \frac{1}{-y_l} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$
Building Shadow Projection Matrix

1. Translate source to origin with \( T(-x_l, -y_l, -z_l) \)
2. Perspective projection
3. Translate back by \( T(x_l, y_l, z_l) \)

\[
M = \begin{bmatrix}
1 & 0 & 0 & x_l \\
0 & 1 & 0 & y_l \\
0 & 0 & 1 & z_l \\
0 & 0 & 0 & 1
\end{bmatrix}
\]

Final matrix that projects Vertex V onto V’ in shadow polygon
Code snippets?

- Set up projection matrix in OpenGL application

```c
float light[3]; // location of light
mat4 m;       // shadow projection matrix initially identity

M[3][1] = -1.0/light[1];
```

\[
M = \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & \frac{1}{y_l} & 0 & 0 \\
\end{bmatrix}
\]
Projective Shadow Code

- Set up object (e.g. a square) to be drawn

```cpp
point4 square[4] = {vec4(-0.5, 0.5, -0.5, 1.0),
                    {vec4(-0.5, 0.5, -0.5, 1.0),
                     {vec4(-0.5, 0.5, -0.5, 1.0),
                      {vec4(-0.5, 0.5, -0.5, 1.0)}}
```

- Copy square to VBO
- Pass modelview, projection matrices to vertex shader
What next?

- Next, we load model_view as usual then draw original polygon
- Then load shadow projection matrix, change color to black, re-render polygon
void display( )
{
    mat4 mm;
    // clear the window
    glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT);
    // render red square (original square) using modelview
    // matrix as usual (previously set up)
    glUniform4fv(color_loc, 1, red);
    glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
Shadow projection Display( ) Function

// modify modelview matrix to project square
// and send modified model_view matrix to shader
mm = model_view

* Translate(light[0], light[1], light[2]
* m
* Translate(-light[0], -light[1], -light[2]);
glUniformMatrix4fv(matrix_loc, 1, GL_TRUE, mm);

// and re-render square as
// black square (or using only ambient component)
glUniform4fv(color_loc, 1, black);
glDrawArrays(GL_TRIANGLE_STRIP, 0, 4);
glutSwapBuffers( );
Shadow Buffer Approach

- Uses second **depth buffer** called shadow buffer
- Pros: not limited to plane surfaces
- Cons: needs lots of memory
- Depth buffer?
OpenGL Depth Buffer (Z Buffer)

- **Depth**: While drawing objects, depth buffer stores distance of each polygon from viewer.
- **Why?** If multiple polygons overlap a pixel, only closest one polygon is drawn.

<table>
<thead>
<tr>
<th></th>
<th>1.0</th>
<th>1.0</th>
<th>1.0</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.3</td>
<td>0.3</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>0.5</td>
<td>0.3</td>
<td>0.3</td>
<td>1.0</td>
<td></td>
</tr>
<tr>
<td>0.5</td>
<td>0.5</td>
<td>1.0</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>

Depth

- Z = 0.5
- Z = 0.3
Setting up OpenGL Depth Buffer

- **Note:** You did this in order to draw solid cube, meshes

1. `glutInitDisplayMode(GLUT_DEPTH | GLUT_RGB)` instructs OpenGL to create depth buffer

2. `glEnable(GL_DEPTH_TEST)` enables depth testing

3. `glClear(GL_COLOR_BUFFER_BIT | GL_DEPTH_BUFFER_BIT)`
   
   Initializes depth buffer every time we draw a new picture
Shadow Buffer Theory

- Along each path from light
  - Only closest object is lit
  - Other objects on that path in shadow
- Shadow buffer stores closest object on each path
Shadow Buffer Approach

- Rendering in two stages:
  - Loading shadow buffer
  - Render the scene
Loading Shadow Buffer

- Initialize each element to 1.0
- Position a camera at light source
- Rasterize each face in scene updating closest object
- Shadow buffer tracks smallest depth on each path
Shadow Buffer (Rendering Scene)

- Render scene using camera as usual
- While rendering a pixel find:
  - pseudo-depth $D$ from light source to $P$
  - Index location $[i][j]$ in shadow buffer, to be tested
  - Value $d[i][j]$ stored in shadow buffer
- If $d[i][j] < D$ (other object on this path closer to light)
  - point $P$ is in shadow
  - set lighting using only ambient
- Otherwise, not in shadow
Loading Shadow Buffer

- Shadow buffer calculation is independent of eye position
- In animations, shadow buffer loaded once
- If eye moves, no need for recalculation
- If objects move, recalculation required
Other Issues

- Point light sources => simple hard shadows, unrealistic
- Extended light sources => more realistic
- Shadow has two parts:
  - Umbra (Inner part) => no light
  - Penumbra (outer part) => some light
Fog is atmospheric effect
- Better realism, helps determine distances
Fog

- Fog was part of OpenGL fixed function pipeline
- Programming fixed function fog
  - **Parameters:** Choose fog color, fog model
  - **Enable:** Turn it on
- Fixed function fog **deprecated!!**
- Shaders can implement even better fog
- **Shaders implementation:** fog applied in fragment shader just before display
Rendering Fog

- Mix some color of fog: \( c_f \) + color of surface: \( c_s \)

\[
c_p = f c_f + (1 - f) c_s \quad f \in [0,1]
\]

- If \( f = 0.25 \), output color = 25% fog + 75% surface color
  - \( f \) computed as function of distance \( z \)
  - 3 ways: linear, exponential, exponential-squared
  - Linear:

\[
f = \frac{Z_{\text{end}} - Z_p}{Z_{\text{end}} - Z_{\text{start}}}
\]
Fog Shader Fragment Shader Example

\[ f = \frac{Z_{\text{end}} - Z_p}{Z_{\text{end}} - Z_{\text{start}}} \]

float dist = abs(Position.z);
Float fogFactor = (Fog.maxDist - dist)/
    Fog.maxDist - Fog.minDist);
fogFactor = clamp(fogFactor, 0.0, 1.0);

vec3 shadeColor = ambient + diffuse + specular
vec3 color = mix(Fog.color, shadeColor,fogFactor);
FragColor = vec4(color, 1.0);

\[ \mathbf{c}_p = f \mathbf{c}_f + (1 - f) \mathbf{c}_s \]
Fog

- Exponential: \( f = e^{-d_{fp}} \)
- Squared exponential: \( f = e^{-(d_{fp})^2} \)
- Exponential derived from Beer’s law
  - **Beer’s law:** intensity of outgoing light diminishes exponentially with distance

![Graph showing fog factor equations](attachment:graph.png)
Fog

- $f$ values for different depths ($z_p$) can be pre-computed and stored in a table on GPU
- Distances used in $f$ calculations are planar
- Can also use Euclidean distance from viewer or radial distance to create *radial fog*
References

- Real Time Rendering by Akenine-Moller, Haines and Hoffman