Computer Graphics (CS 543)
Lecture 9b: Shadows and Shadow Maps

Prof Emmanuel Agu

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

- Shadows give information on relative positions of objects

Use just ambient component

Use ambient + diffuse + specular components

Use just ambient component
Why shadows?

- More realism and atmosphere

Image courtesy of BioWare

Neverwinter Nights
Types of Shadow Algorithms

- Project shadows as separate objects (like Peter Pan's shadow)
  - Projective shadows
- As volumes of space that are dark
  - Shadow volumes [Franklin Crow 77]
- As places not seen from a light source looking at the scene
  - Shadow maps [Lance Williams 78]
- Fourth method used in ray tracing
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/scene 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 \((x_p, 0, z_p)\)
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

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

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

$$M = \begin{bmatrix} 1 & 0 & 0 & x_l \\ 0 & 1 & 0 & y_l \\ 0 & 0 & 1 & z_l \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & -y_l & 0 & 0 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -x_l \\ 0 & 1 & 0 & -y_l \\ 0 & 0 & 1 & -z_l \\ 0 & 0 & 0 & 1 \end{bmatrix}$$

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];
```

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

1. Load modelview
   draw polygon as usual

2. Modify modelview with Shadow projection matrix
   Re-render as black (or ambient)
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 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 Map Illustrated**

- Second dept buffer called the **shadow map** is used.
- Point $v_a$ stored in element $a$ of shadow map: lit!
- Point $v_b$ **NOT** in element $b$ of shadow map: In shadow.

Not limited to planes.
Shadow Map: Depth Comparison

Render depth image from light

A fragment is in shadow if its depth is greater than the corresponding depth value in the shadow map.
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>0.5</th>
<th>0.3</th>
<th>0.3</th>
<th>1.0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.0</td>
<td>0.5</td>
<td>0.3</td>
<td>0.3</td>
<td>1.0</td>
</tr>
<tr>
<td>0.3</td>
<td>0.3</td>
<td>0.3</td>
<td>1.0</td>
<td></td>
</tr>
</tbody>
</table>
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 Map Approach

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

- Initialize each element to 1.0
- Position a camera at light source
- Rasterize each face in scene updating closest object
- Shadow map (buffer) tracks smallest depth on each path
Shadow Map (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
  - lighting = ambient
- Otherwise, not in shadow
  - Lighting = amb + diffuse + specular
Loading Shadow Map

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

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