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

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 \((x_p, 0, z_p)\)

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

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

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

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

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

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

\[
\text{point4 square}[4] = \{\text{vec4}(-0.5, 0.5, -0.5, 1.0)\}
\]
\[
\quad \{\text{vec4}(-0.5, 0.5, -0.5, 1.0)\}
\]
\[
\quad \text{vec4}(-0.5, 0.5, -0.5, 1.0)
\]
\[
\quad \text{vec4}(-0.5, 0.5, -0.5, 1.0)
\]

- 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( );

\[
M = \begin{bmatrix}
1 & 0 & 0 & x_i \\
0 & 1 & 0 & y_i \\
0 & 0 & 1 & z_i \\
0 & 0 & 0 & 1
\end{bmatrix}\begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 1
\end{bmatrix}\begin{bmatrix}
1 & 0 & 0 & -x_i \\
0 & 1 & 0 & -y_i \\
0 & 0 & 1 & -z_i \\
0 & 0 & 0 & 1
\end{bmatrix}
\]
Fog
Fog example

- 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: $c_s$

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

- 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:

$$f = \frac{z_{end} - z_p}{z_{end} - z_{start}}$$
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);

\[
f = \frac{z_{end} - z_p}{z_{end} - z_{start}}
\]

\[
c_p = fc_f + (1 - f)c_s
\]
Fog

- Exponential: \( f = e^{-d_f z_p} \)
- Squared exponential: \( f = e^{-(d_f z_p)^2} \)
- Exponential derived from Beer’s law
  - **Beer’s law**: intensity of outgoing light diminishes exponentially with distance, similar to real life
Fog Optimizations

- $f$ 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