Computer Graphics (CS 543)
Lecture 10 (Part 1): Shadows and Fog

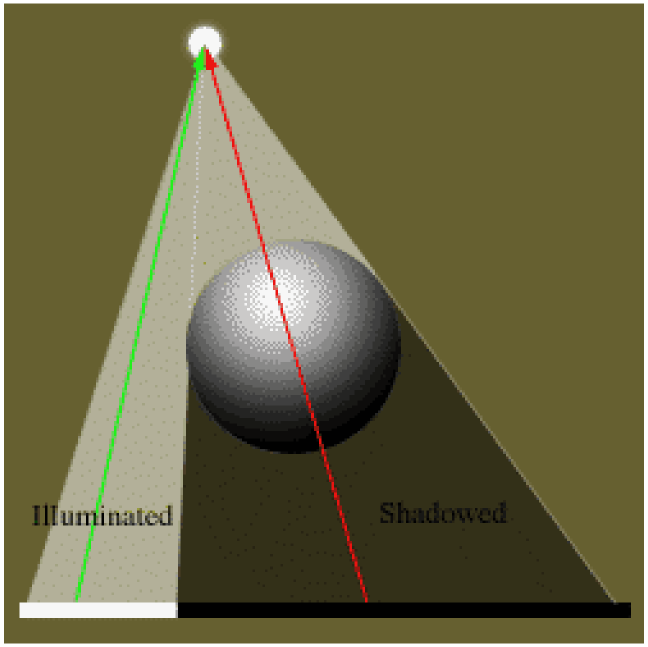
Prof Emmanuel Agu

Computer Science Dept.

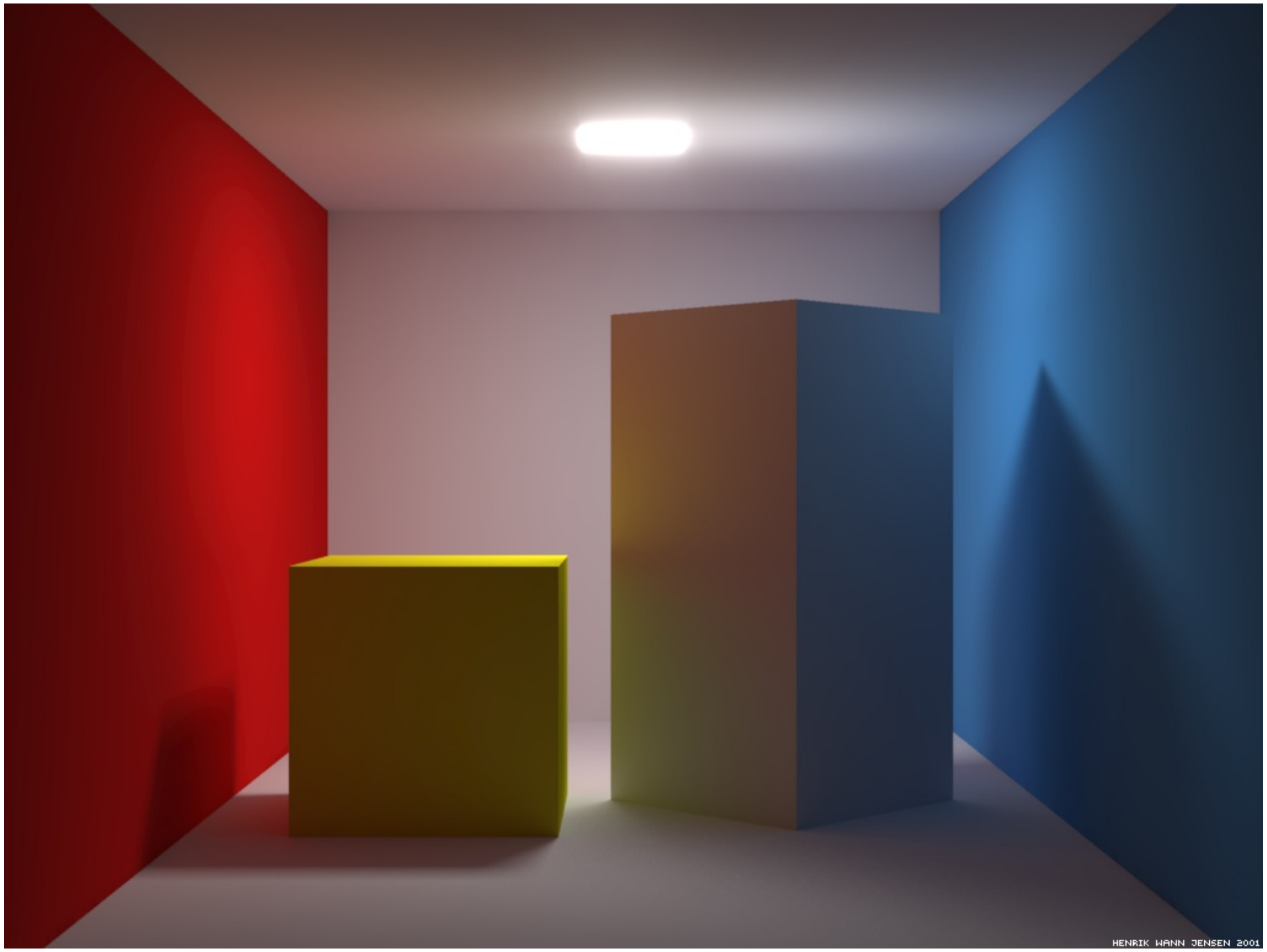
Worcester Polytechnic Institute (WPI)

Introduction to Shadows

Basic idea:



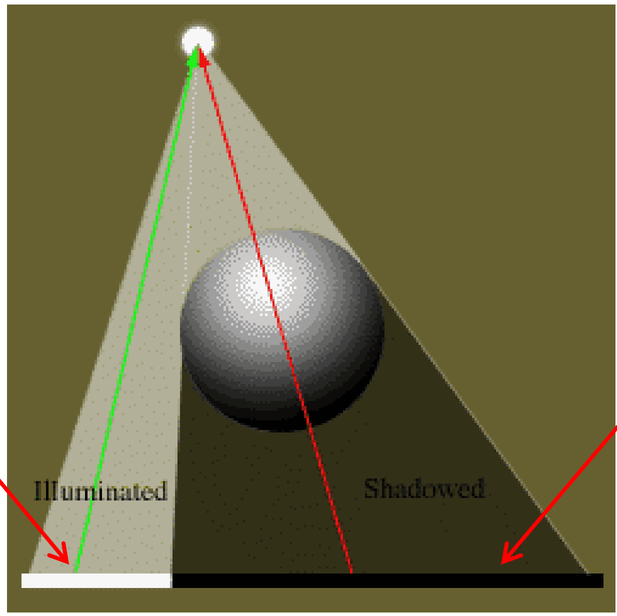
Introduction to Shadows



Introduction to Shadows

- Shadows make image more realistic
  - Important visual cues on relative positions of objects
- Lighting calculations for rendering shadows:
  - Points in shadow: only ambient component
  - Points NOT in shadow: ambient + diffuse + specular

Use ambient + diffuse + specular components



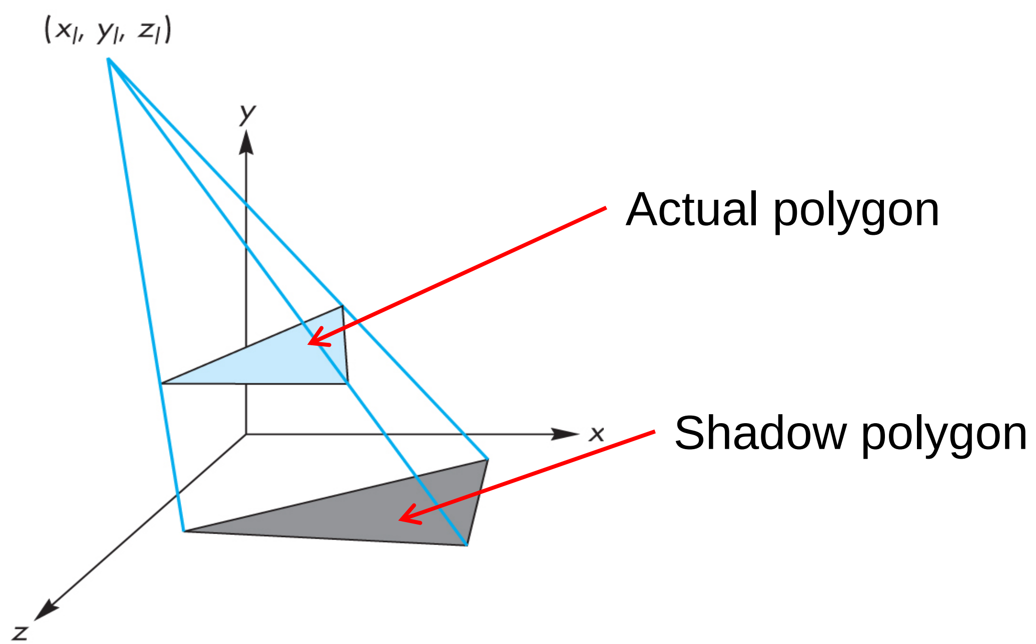
Use just ambient component

Introduction to Shadows

- Simple illumination models == simple shadows
- Two popular shadow rendering methods:
  1. Shadows as texture (projection)
  2. Shadow buffer
- Third method used in ray-tracing (covered in grad class)

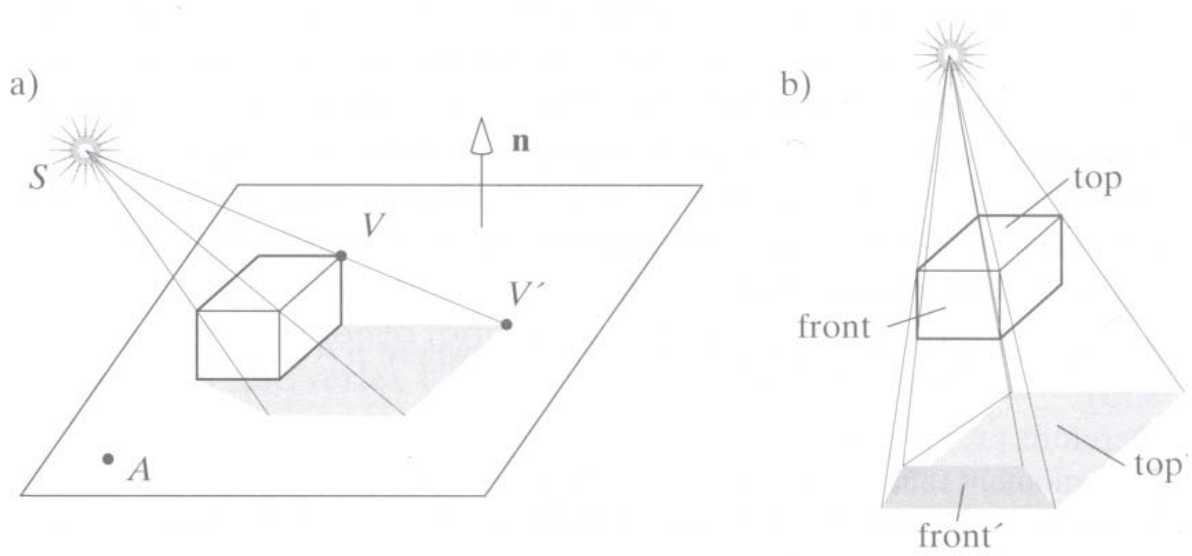
Projective Shadows

- Oldest methods: Used in early flight simulators
- Projection of a polygon is a polygon called a **shadow polygon**



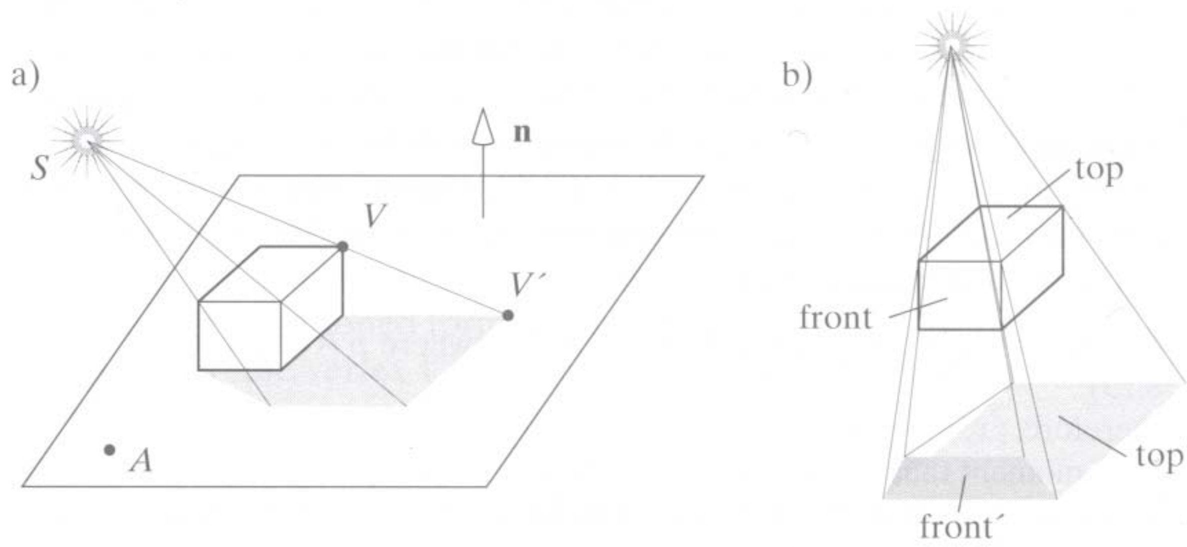
Projective Shadows

- Given a point light and a polygon, vertices of the shadow polygon (V’) are projections of original polygon’s vertices (V) from light source onto a surface



Projective Shadows

- Works for flat surfaces illuminated by point light
- Problem: compute shape of shadow
- For each face, project vertices, draw shadow polygon
- Shadow of entire object = union of projections of individual faces

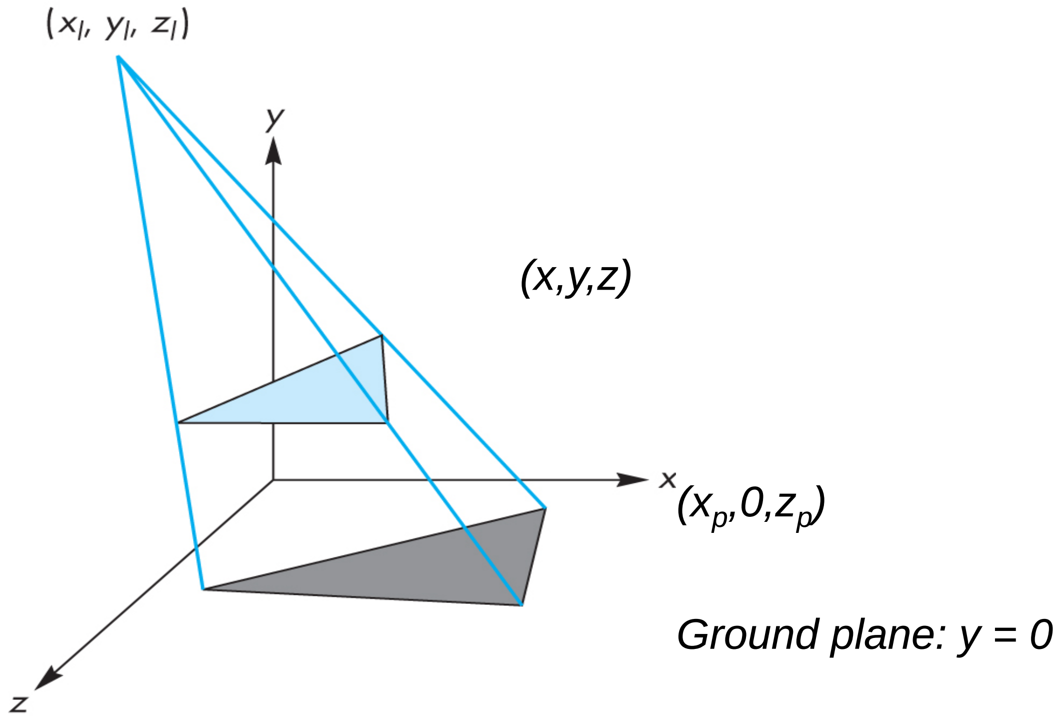


Projective Shadow Algorithm

- 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

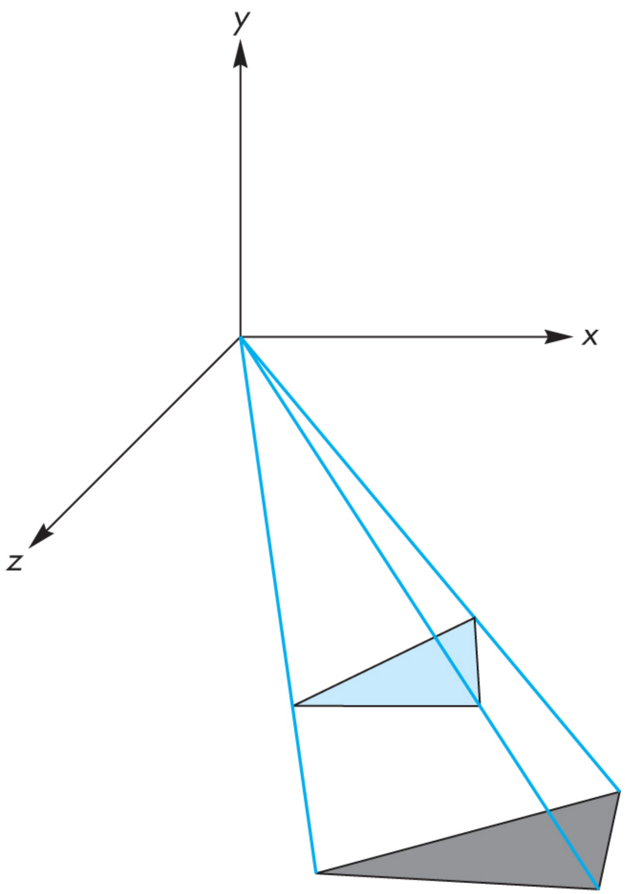
1. Source at (x_l, y_l, z_l)
2. Vertex at (x, y, z)
3. Would like to calculate shadow polygon vertex V projected onto ground at $(x_p, 0, z_p)$



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

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



Building Shadow Projection Matrix

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

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

Final matrix that projects
Vertex V onto V' in shadow polygon

Code snippets?

- Set up projection matrix in OpenGL application

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

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

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

Projective Shadow Code

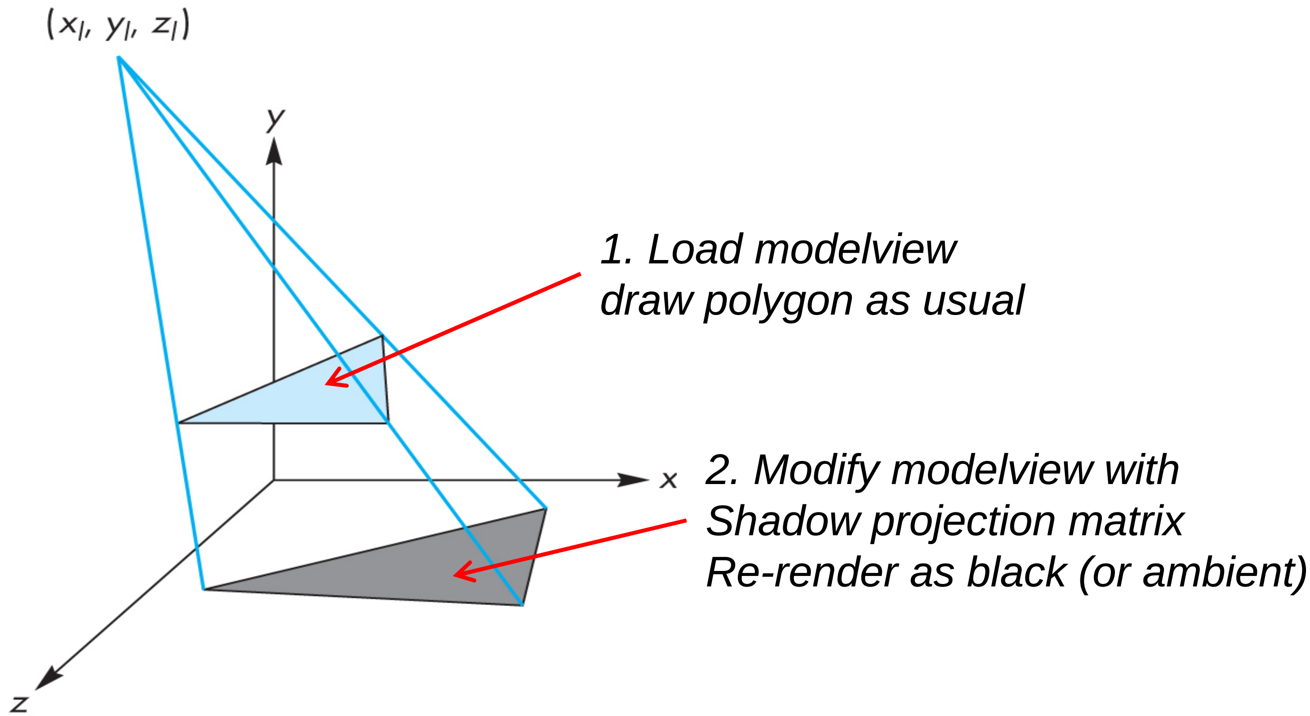
- Set up object (e.g a square) to be drawn

```
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}
```

- Set up VBO, copy square to VBO
- Set up modelview, projection matrices, pass 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



Shadow projection Display( ) Function

```
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( );
}
```

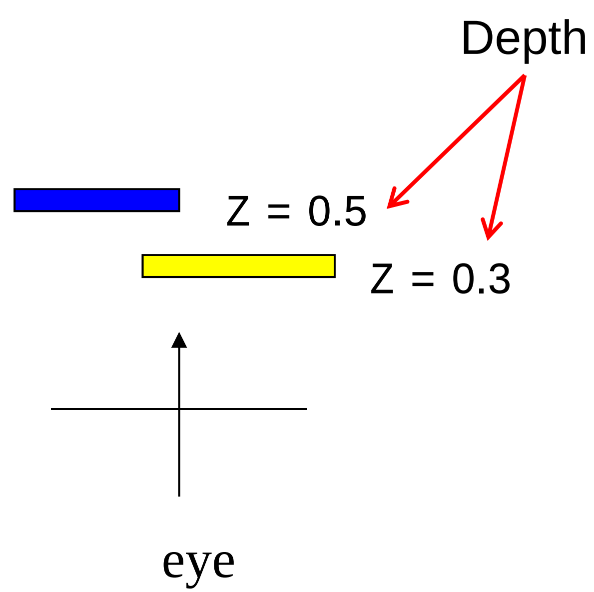
Shadow Buffer Approach

- 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

1.0	1.0	1.0	1.0
1.0	0.3	0.3	1.0
0.5	0.3	0.3	1.0
0.5	0.5	1.0	1.0



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)`

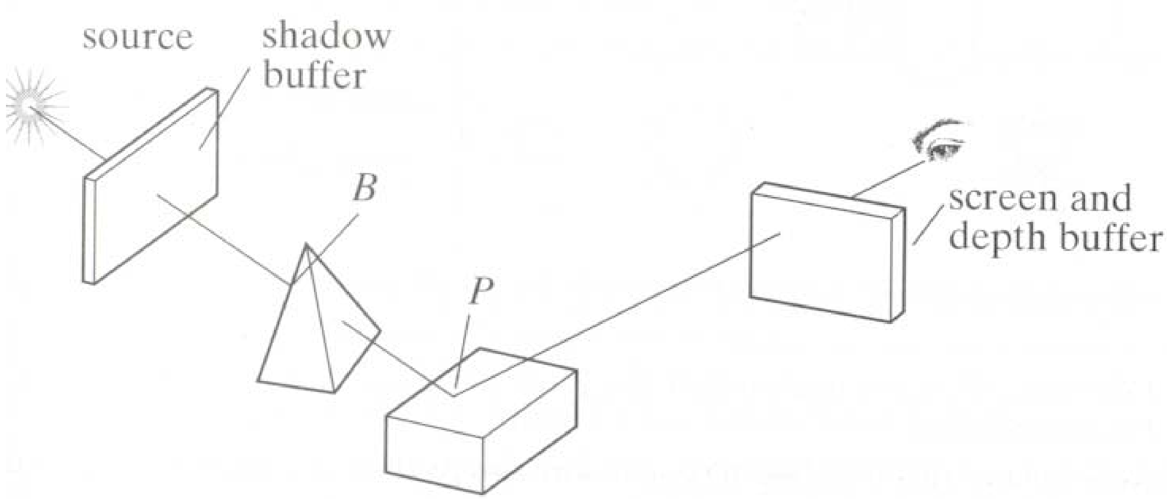
   Initializes depth buffer every time we draw a new picture

Shadow Buffer Approach

- Theory:
  - Establish object-light path
  - Other objects in object-light path = object in shadow
  - Otherwise, not in shadow

Shadow Buffer Approach

- Shadow buffer records object distances from light source
- Shadow buffer element = distance of closest object in a direction

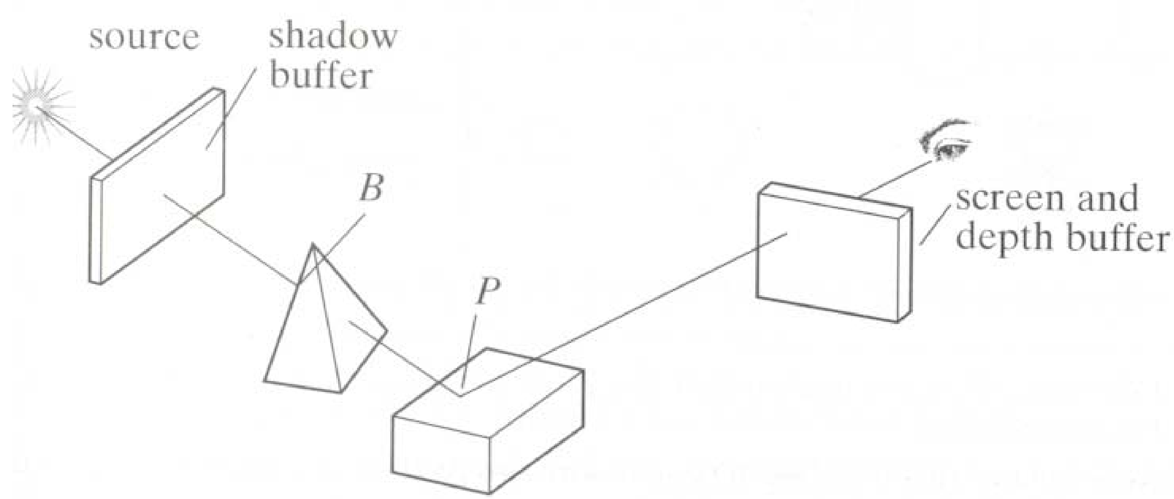


Shadow Buffer Approach

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

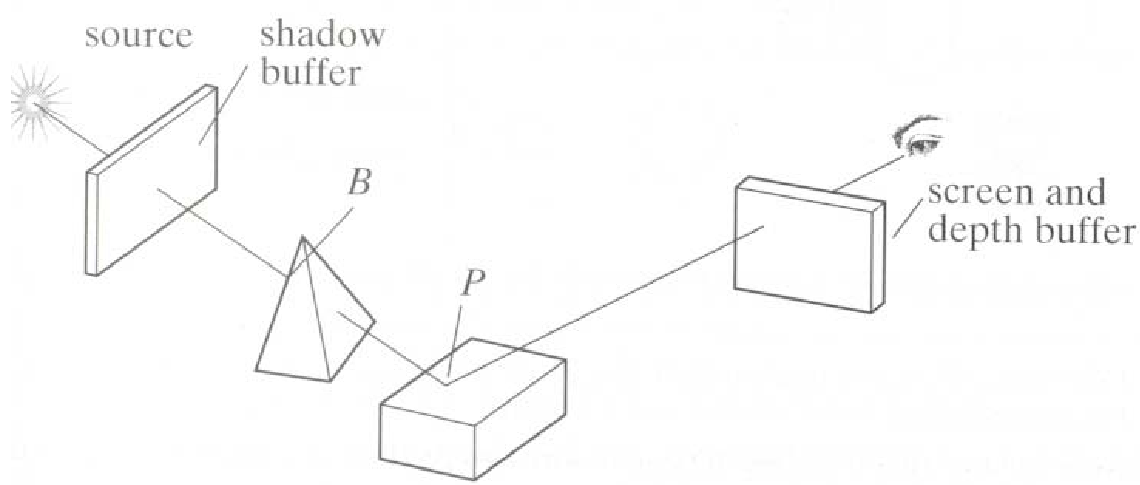
Loading Shadow Buffer

- Initialize each element to 1.0
- Position a camera at light source
- Rasterize each face in scene updating pseudo-depth
- Shadow buffer tracks smallest pseudo-depth so far



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

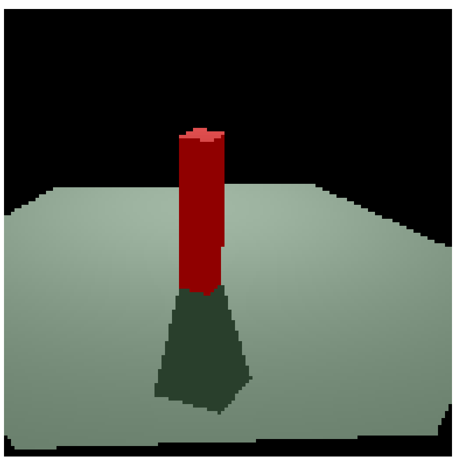


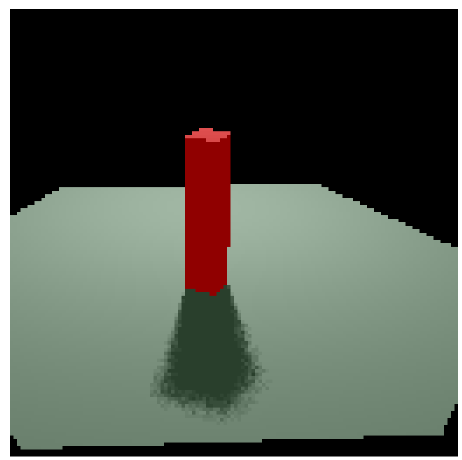
Shadow Buffer (Rendering Scene)

- 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

Other Issues

- Point light sources => simple but a little unrealistic
- Extended light sources => more realistic
- Shadow has two parts:
  - Umbra (Inner part) => no light
  - Penumbra (outer part) => some light

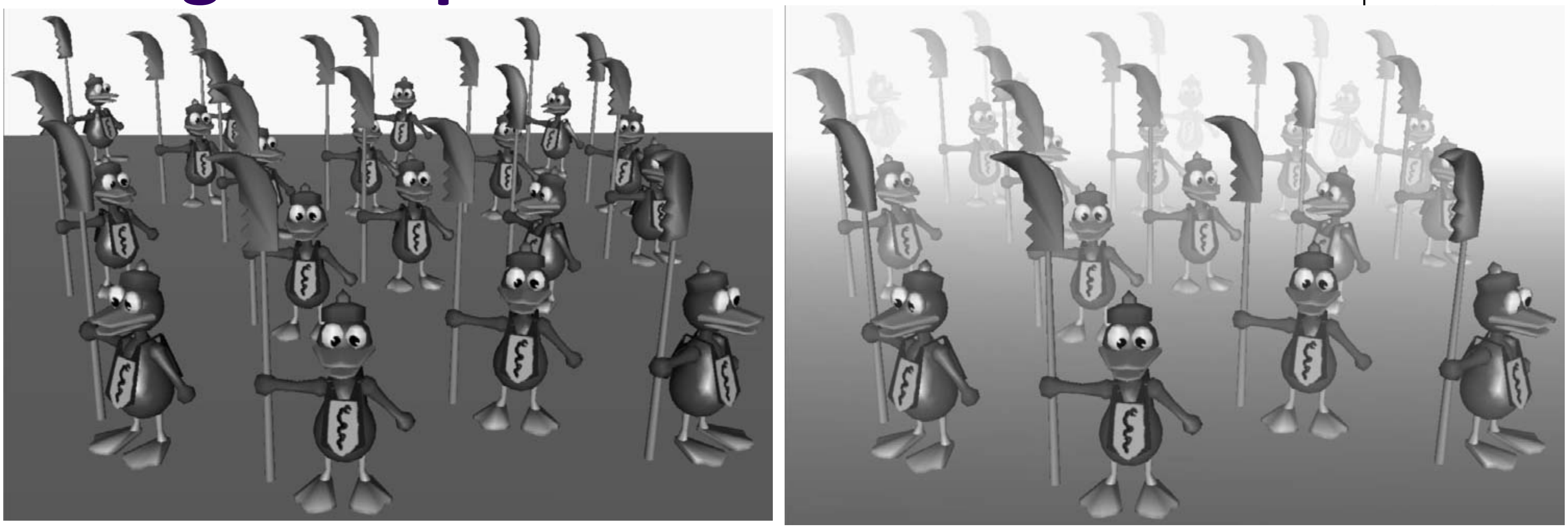




Fog

- Fog was part of OpenGL fixed function pipeline
- Using shaders, fog applied to scene just before display
- Shaders can generate more elaborate fog
- Fog is atmospheric effect
  - A little better realism
  - Help in determining distances

Fog example



- Often just a matter of
  - Choosing fog color
  - Choosing fog model
  - Turning it on

Rendering Fog

- Color of fog: $\mathbf{c}_f$ color of surface: $\mathbf{c}_s$

$$\mathbf{c}_p = f\mathbf{c}_f + (1 - f)\mathbf{c}_s \qquad f \in [0,1]$$

- How to compute *f* ?
- 3 ways: linear, exponential, exponential-squared
- Linear:

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

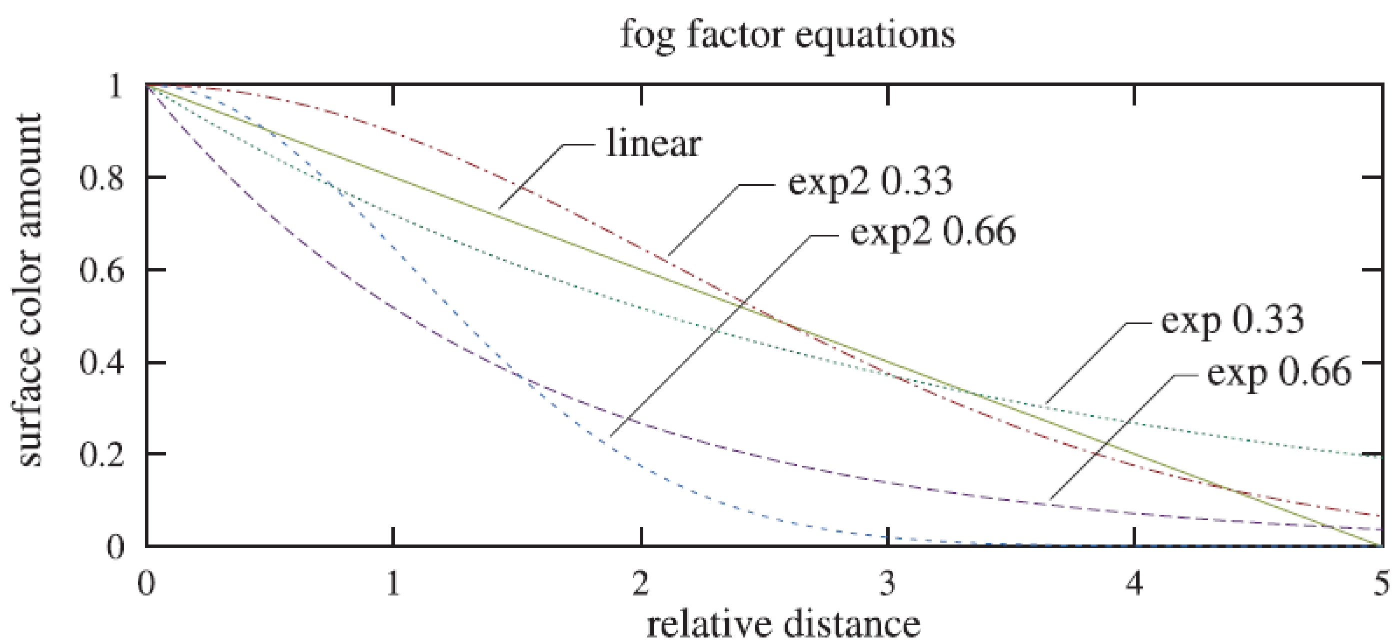
Fog Shader Fragment Shader Example

```
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);
```

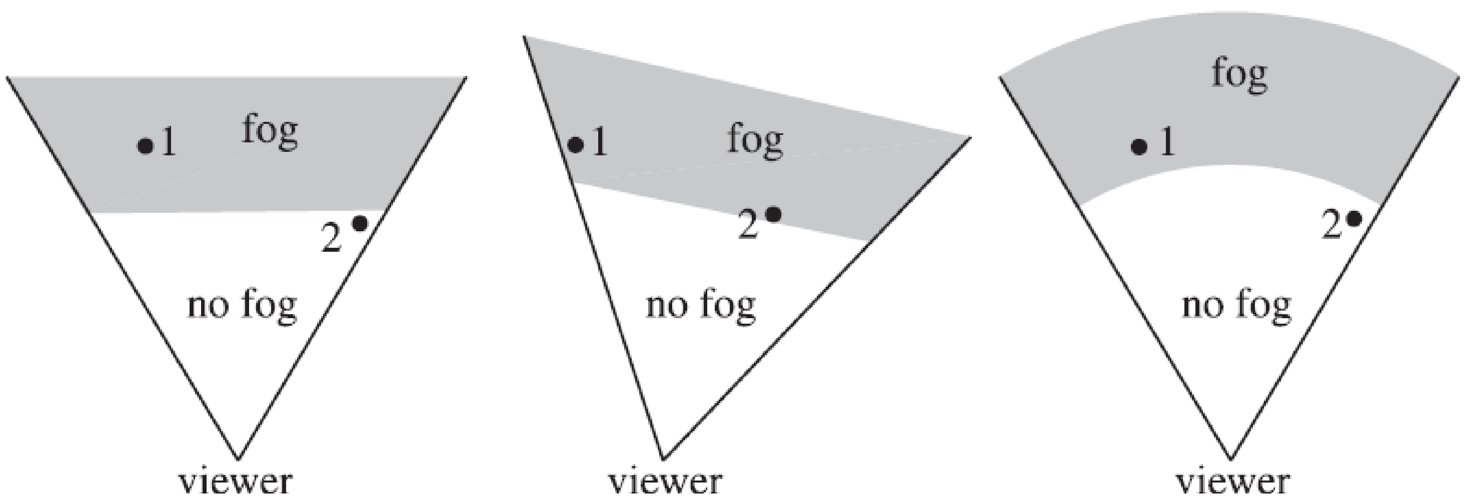
Fog

- 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



Fog

- *f* values for different depths 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

- Interactive Computer Graphics (6th edition), Angel and Shreiner
- Computer Graphics using OpenGL (3rd edition), Hill and Kelley
- Real Time Rendering by Akenine-Moller, Haines and Hoffman