Binghamton University

EngiNet™

State University of New York

Thomas J. Watson

School of Engineering
and Applied Science

EngiNet™

WARNING

All rights reserved. No Part of this video lecture series may be reproduced in any form or by any electronic or mechanical means, including the use of information storage and retrieval systems, without written approval from the copyright owner.

©2001 The Research Foundation of the State University of New York

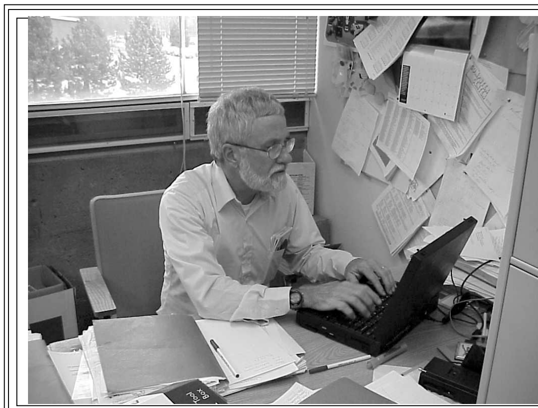
CS 460/560

Computer Graphics

Professor Richard Eckert

Lecture # 23

May 2, 2001



Lecture 23

- 3D Graphics using OpenGL
  - Illumination and Shading
- Hidden Surface Removal Revisited
  - Depth Sort
- Shadows

Illumination & Reflection in OpenGL

- Uses the Phong Illumination/Reflection Model

Final Phong Illumination/Reflection Model Result (Single Light Source)

- Three color intensity equations:

I(r,g,b) = Ambient + Point Diffuse + Point Specular

I(r,g,b) = kd(r,g,b)*Ia
+ Ip*kd(r,g,b)*(N.L)
+ Ip*ks*$(R.V)^n$

Illumination & Reflection in OpenGL

- Define Light Sources
- Define Material Properties
- Define Normal Vectors
- Specify Shading Model
- Enable Depth Testing (Z-Buffer)

Defining a Light Source

- Set up Arrays of lighting values

```
GLfloat ambLight0[ ] = {0.3f, 0.3f, 0.3f, 1.0f}; // R,G,B,α
GLfloat diffLight0[ ] = {0.5f, 0.5f, 0.5f, 1.0f};
GLfloat specLight0[ ] = {0.0f, 0.0f, 0.0f, 1.0f};
GLfloat posnLight0[ ] = {1.0f, 1.0f, 1.0f, 0.0f); // x,y,z,w
```

- Pass Arrays to OpenGL

```
glLightfv(GL_LIGHT0, GL_AMBIENT, ambLight0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffLight0);
glLightfv(GL_LIGHT0, GL_SPECULAR, specLight0);
glLightfv(GL_LIGHT0, GL_POSITION, posnLight0);
```

Enabling a Light Source

- Turning on the light source

```
glEnable(GL_LIGHTING);
glEnable(GL_LIGHT0);
```

Material Reflection Properties

- Set up Material Arrays
  - ambient/diffuse reflection coefficients

    GLfloat mat_ambdiff[ ] = {0.0f, 0.7f, 0.0f, 1.0f};
  - specular reflection coefficient

    GLfloat mat_spec[ ] = {1.0f, 1.0f, 1.0f, 1.0f};
- Pass Material Arrays to OpenGL

```
glMaterialfv(GL_FRONT,
   GL_AMBIENT_AND_DIFFUSE,
   mat_ambdiff);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_spec);
glMaterialf(GL_FRONT, GL_SHININESS, 20.0f);
```

Defining Normals

- Must compute normals for all polygons
- OpenGL has no function to do that
  - So write your own
- Assume the result is:

  double n[3];
- Use this when you define the polygon

```
glBegin(GL_POLYGON)
    glNormal3f((GLfloat)n[0], (GLfloat)n[1], (GLfloat)n[2]);
    // glVertex3f() calls here for polygon vertices
glEnd();
```

Specify a Shading Model & Enable Depth Testing

```
glShadeModel(GL_FLAT);  // use GL_SMOOTH
   for Gouraud shading
glEnable(GL_DEPTH_TEST);
glClear (GL_COLOR_BUFFER_BIT |
   GL_DEPTH_BUFFER_BIT);
      // clear frame buffer and z-buffer
```

Some sample code (view class: OnDraw)

```
glShadeModel(GL_SMOOTH);
glEnable(GL_DEPTH_TEST);
glClearColor(1.0f, 1.0f, 1.0f, 1.0f);
glClear(GL_COLOR_BUFFER_BIT |GL_DEPTH_BUFFER_BIT);
glMatrixMode(GL_MODELVIEW);  glLoadIdentity();
glMaterialfv(GL_FRONT, GL_AMBIENT_AND_DIFFUSE,mat_ambdiff);
glMaterialfv(GL_FRONT, GL_SPECULAR, mat_spec);
glMaterialf(GL_FRONT, GL_SHININESS, 20.0f);
glLightfv(GL_LIGHT0, GL_AMBIENT, ambLight0);
glLightfv(GL_LIGHT0, GL_DIFFUSE, diffLight0);
glLightfv(GL_LIGHT0, GL_SPECULAR, specLight0);
glLightfv(GL_LIGHT0, GL_POSITION, posnLight0);
glEnable(GL_LIGHTING); glEnable(GL_LIGHT0);
DrawCube();
glFlush();
```

Code from DrawCube() function

```
glTranslatef(0.0f, 0.0f, -3.0f);    // position cube
glRotatef(20.0f, 1.0f, 0.0f, 0.0f);
glRotatef(20.0f, 0.0f, 1.0f, 0.0f);
// Draw the polygons of the cube, only one face given here:
double p1[] = {-0.5, 0.5, 0.5}; double p2[] = {-0.5, -0.5, 0.5};
double p3[] = {0.5, -0.5, 0.5}; double n[3];
CalcNormal(p1, p2, p3, n);
glBegin(GL_POLYGON); // only 1 face here, other 5 must be defined
      glNormal3f((GLfloat)n[0], (GLfloat)n[1], (GLfloat)n[2]);
      glVertex3f(-0.5f, 0.5f, 0.5f);
      glVertex3f(-0.5f, -0.5f, 0.5f);
      glVertex3f(0.5f, -0.5f, 0.5f);
      glVertex3f(0.5f, 0.5f, 0.5f);
glEnd();
```

Code from CalcNormal(double *p1, double *p2, double *p3, double *n)

```
// Form two vectors from the points.
double a[3], b[3];
a[0] = p2[0] - p1[0];    a[1] = p2[1] - p1[1];    a[2] = p2[2] - p1[2];
b[0] = p3[0] - p1[0];    b[1] = p3[1] - p1[1];    b[2] = p3[2] - p1[2];
// Calculate the cross product of the two vectors.
n[0] = a[1] * b[2] - a[2] * b[1];
n[1] = a[2] * b[0] - a[0] * b[2];
n[2] = a[0] * b[1] - a[1] * b[0];
// Normalize the new vector.
double length = sqrt(n[0]*n[0]+n[1]*n[1]+n[2]*n[2]);
n[0] = n[0] / length;
n[1] = n[1] / length;
n[2] = n[2] / length;
```

Back to Hidden Surface Removal

- The Depth Sort Technique

Depth Sort Hidden Surface Removal

- Basic Idea
  - Order Polygons according to how far away from observer
  - Then “paint” them into picture, farthest first, closest last
    - Far surfaces overwritten by near surfaces
  - The way artists sometimes paint scenes

Algorithm

1. Remove back faces (preprocessing step)
2. Decompose remaining polygons into triangles
   - Depth determination easiest for triangles
3. Sort triangles into depth order
4. Apply Painters Algorithm

The Depth Sort (step 3)

For each triangle

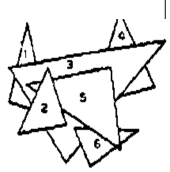
Create a linked list with pointers to all triangles in front of the triangle

And a counter of all triangles in back of the triangle

Gives an array of linked lists

An Example

triangle	counter	list
1	0	3 5 2
2	3	
3	2	2 5 6
4	0	3 5
5	4	2
6	1	5



Painter’s Algorithm (step 4)

Repeat

Go through array & draw any triangle whose counter = 0 // it’s the farthest

Decrement counter of all triangles in list of drawn triangle // since there’s 1 less in back

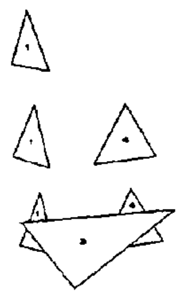
Mark counter of drawn triangle as finished

Until all triangles have been drawn

Painter's Algorithm in Action (A)

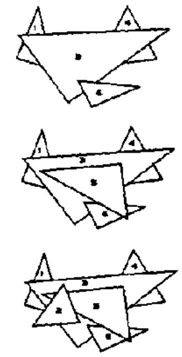
Triangle	List	Counters	Counters after
1 draw	3 5 2	0	•
2		3	2
3	2 5 6	2	1
4	3 5	0	0
5	2	4	3
6	5	1	1
1	3 5 2	•	•
2		2	2
3	2 5 6	1	0
4 draw	3 5	0	•
5	2	3	2
6	5	1	1
1	3 5 2	•	•
2		2	1
3 draw	2 5 6	0	•
4	3 5	•	•
5	2	2	1
6	5	1	0

Picture



Painter's Algorithm in Action (B)

1	3 5 2	•	•
2		1	1
3	2 5 6	•	•
4	3 5	•	•
5	2	1	0
6 draw	5	0	•
1	3 5 2	•	•
2		1	0
3	2 5 6	•	•
4	3 5	•	•
5 draw	2	0	•
6	5	•	•
1	3 5 2	•	•
2 draw		0	•
3	2 5 6	•	•
4	3 5	•	•
5	2	•	•
6	5	•	•

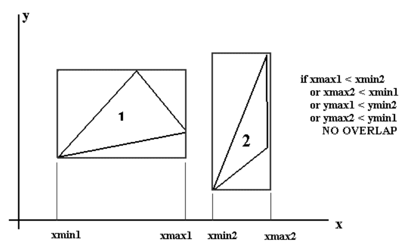


Depth Sort (Step 3) Details

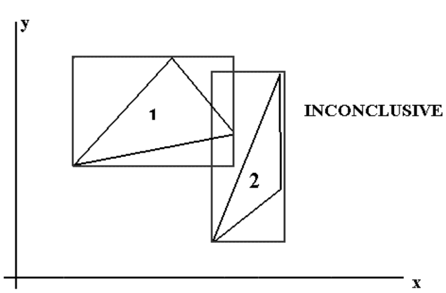
```
Initialize all triangle counters to 0
for i = 1 to n-1
    for j = i+1 to n
        if triangles i & j "overlap"
            if triangle j is "in front of" triangle i
                add triangle j to i's list
                ctr[j]++
            else
                add triangle i to j's list
                ctr[i]++
```

Determine if Triangles "overlap"

1. Mini-max test (inexpensive)
   - look at containing rectangles
   - if rectangles don't overlap, triangles don't

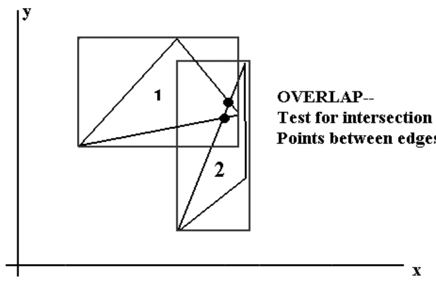


But if rectangles overlap, triangles may not



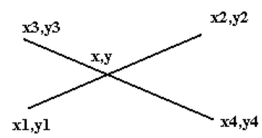
2. So check for intersections of pairs of edges
   - we're working with x-y projections
   - if projected edges intersect, triangles overlap
   - 9 tests in all
   - Can use parametric equations of edges to look for intersections

2. Test for Intersection Points between Edges



OVERLAP--
Test for intersection
Points between edges

Can Use Parametric Equations



x=x1+(x2-x1)t
y=y1+(y2-y1)t

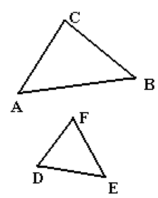
x=x3+(x4-x3)s
y=y3+(y4-y3)s

Equate & solve for s & t
If 0<=t<=1
& 0<=s<=1
the edges intersect

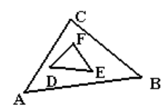
3. But if there are no intersections, triangles still could overlap

- One triangle's projection could contain the other's
- So test for containment

3. Test for Containment

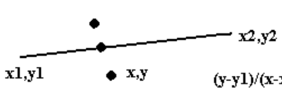


D,E,F not on same side
of AB as C ==>
DEF not contained in ABC



D on same side of BC as A
E on same side of BC as A
F on same side of BC as A
D on same side of AC as B
E on same side of AC as B
F on same side of AC as B
D on same side of AB as C
E on same side of AB as C
F on same side of AB as C
==> DEf is contained in ABC

Which Side of a Line is a Point on



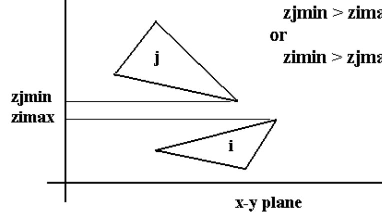
(y-y1)/(x-x1) = (y2-y1)/(x2-x1): on line
Or: (x-x1)(y2-y1) - (y-y1)(x2-x1) = 0

f(x,y) = (x-x1)(y2-y1) - (y-y1)(x2-x1)
f(x,y) = 0 ==> (x,y) on line
f(x,y) < 0 ==> (x,y) on one side of line
f(x,y) > 0 ==> (x,y) on other side of line

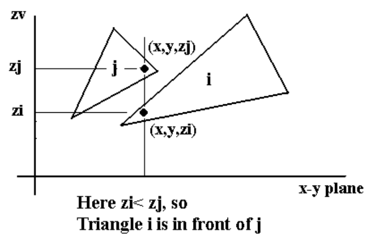
Determine which Triangle "in front"

1. Mini-max test
   - if all zv's of i's vertices < all zv's of j's, then i is in front of j

i in front of j if:
zjmin > zimax
or
zimin > zjmax

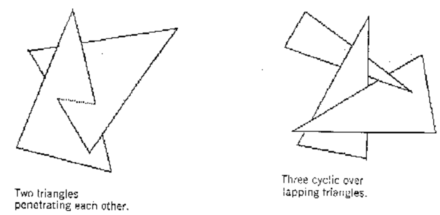


2. If test 1 inconclusive, find a point with same x,y on overlapping triangle projections & compute z for each
   - Triangle with smaller z is in front



Algorithm won't work for some cases

- Inter-penetrating triangles
- Cyclic overlapping triangles
- So decompose triangles into smaller ones

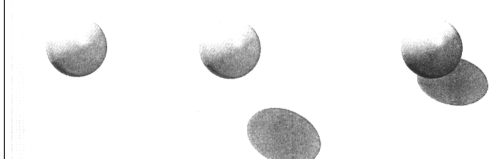


Performance of Depth Sort

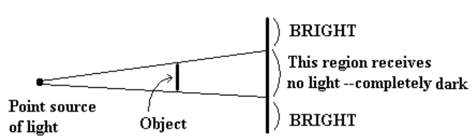
- Depends on number of polygons
- More polygons means slower
- And it's worse than linear

Shadows

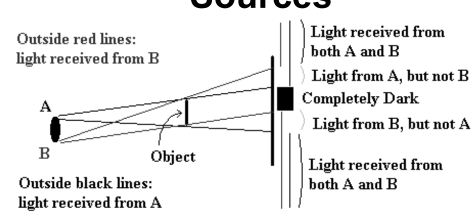
- Very important to our perception of depth
- Shadow position/orientation give information as to how objects relate to each other in space



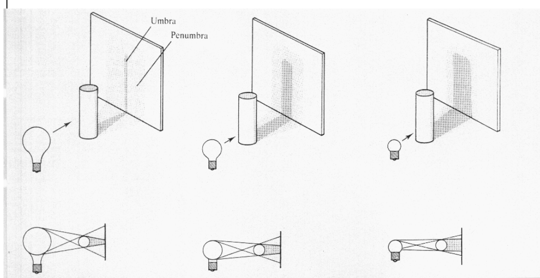
Sharp Shadows from Point Sources



Soft Shadows from Extended Sources



Umbra: central area that receives no light (complete shadow)
Penumbra: areas in partial shadow (receive light from part of source)



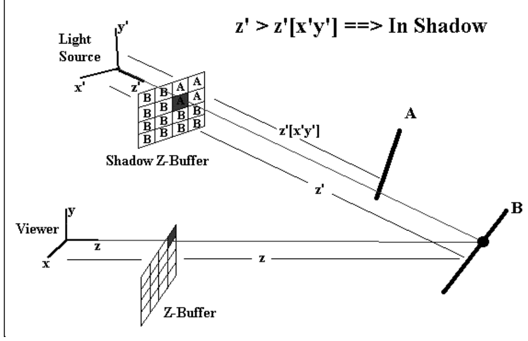
Shadows from Point Sources

- Look at shadows from point sources
- If a point is in shadow, set Phong Ip to 0
  - Source gets no light from point source
  - So no reflection from point source
  - Still must include ambient term
- Lots of algorithms
- One of simplest: Shadow Z-Buffer

Shadow Z-Buffer Algorithm

- A two-stage process
  1. Take source as viewpoint & compute depths
     - Store results in shadow Z-buffer Z'[x'y']
     - Each Z'[x',y'] will contain distance of closest surface to light source
  2. Normal Z-Buffer rendering
     - But if (x,y) is closest (visible), transform to light space coordinates (x',y',z')
     - If z' > Z'[x',y'] point is in shadow
       - Some object is closer to light & will block it
       - So only include ambient term in computation

Shadow Z-Buffer



Set up shadow Z-buf Z'[x',y'] using coordinate system whose origin is at light source

```
Z-buf[x,y]=infinity for all x,y
for each polygon
    for each pixel x,y
        calculate z
        if z < Z-buf[x,y]
            transform x,y,z to light coord space x',y',z'
            if z' > Z'[x',y']
                reduce intensity (include only ambient)
            Z-buf[x,y]=z; fb[x,y]=intensity
```