Binghamton University

EngiNet™

State University of New York

Thomas J. Watson

School of Engineering
and Applied Science

EngiNet™

WARNING

CS 460/560

Computer Graphics

Professor Richard Eckert

Lecture # 21

April 23, 2001

Lecture 21

- Hidden Surface Removal
  - Back Face Culling
- Review of Viewing/Projection Transformations
  - An Example Program Skeleton
- Z-Buffer Hidden Surface Removal
- Depth Sort Hidden Surface Removal

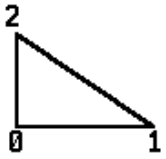
Simple Hidden Surface Removal (Back-Face Culling)

- For complex objects there are lots of polygons
- Many polygons not visible
  - Because they face away from observer
- More realistic, less complex image is produced if only visible polygons are drawn
  - So draw only those facing toward observer
- Technique of back-face culling determines if polygon is visible or not

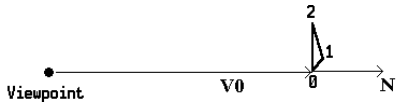
Back-Face Culling

- Define one side of each polygon to be the visible side
  - That side is the outward-facing side
- Defining each polygon in the polygons array:
  - Systematically number vertices in counter-clockwise fashion as seen from outside of the object

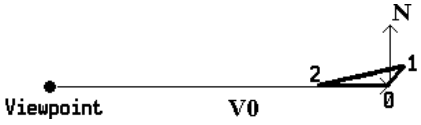
- Consider triangle with vertices 0, 1, 2
- Visible side of the triangle: 0,1,2
  - Vertices numbered in counter-clockwise order
  - Invisible side is: 0,2,1
    - (clockwise vertex ordering)



- Define vector N
  - Outward normal to triangle
- DefineVectorV0,
  - Vector from observer to vertex 0
- Some Cases:
  - N and V0 nearly parallel (V0 . N = 1)
  - Visible side of triangle 012 invisible to viewer



- Rotate triangle about side 01 by 90 degrees
  - Now N and V0 are perpendicular (V0 . N = 0)
  - Triangle is about to become visible
  - At all other points between these two orientations:
    - **V0 . N is positive**
    - **And triangle is invisible to viewer**



- Continue rotation about side 01
- Triangle becomes visible to the viewer
- 90 degrees more, N and V0 are antiparallel

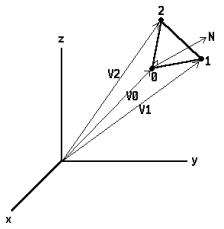
  V0 . N = -1

  Triangle facing toward viewer and is visible

  – At all intermediate orientations:
    - **Triangle is visible**
    - **And V0 . N is negative**

Criterion for Invisibility

- If V0 . N > 0, triangle 012 is invisible
- Now place triangle 012 in an arbitrary position relative to viewer V



- Outward normal N is vector (cross) product of V01 and V02

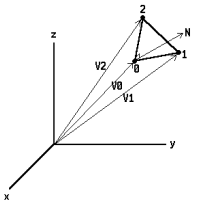
  V01 is vector from vertex 0 to vertex 1

  V02 is vector from vertex 0 to vertex 2
- So: N = V01 x V02
- Criterion for invisibility:

  V0 . (V01 x V02) > 0
- But:

  V01 = V1 - V0

  V02 = V2 - V0



- Substituting we get:

  V0 . [(V1 - V0) x (V2 - V0)] > 0, invisibility
- Expanding:

  V0 . (V1 x V2) - V0 . (V1 x V0)
  - V0 . (V0 x V2) + V0 . (V0 x V0) > 0
- Last Term = 0

  (Cross product of any vector with itself = 0)
- Middle two terms:

  Quantity inside ( ) is a vector perpendicular to V0

  So dot product of either vector with V0 is 0

So: V0 . (V1 x V2) > 0

– For right-handed coordinate system, triple product can be expressed as a determinant

$$V0 \cdot (V1 \times V2) = \begin{vmatrix} X0 & Y0 & Z0 \\ X1 & Y1 & Z1 \\ X2 & Y2 & Z2 \end{vmatrix}$$

- (X0,Y0,Z0), (X1,Y1,Z1), (X2,Y2,Z2) are viewing coordinates (xv,yv,zv) of vertices 0, 1, and 2
- But viewing coordinate system is left-handed
- So sign of the determinant must be reversed

Final Criterion for Invisibility

$$\begin{vmatrix} X0 & Y0 & Z0 \\ X1 & Y1 & Z1 \\ X2 & Y2 & Z2 \end{vmatrix} < 0$$

- Result can be applied to any planar polygon
- Use viewing coordinates of three consecutive polygon vertices
- Could implement as a "visibility" function
  - Computes and returns value of determinant
    - Positive means visible, negative invisible

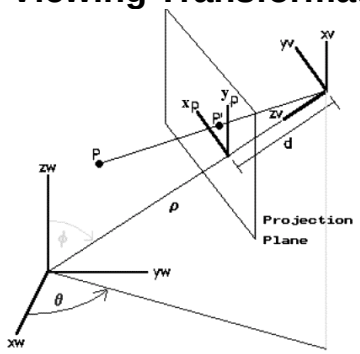
Review of 3D Viewing/Projection

3-D Viewing Transformation

- Converts world coordinates (xw,yw,zy) of a point to viewing coordinates (xv,yv,zv) of the point
  - As seen by a "camera" that is going to "photograph" the scene

(xw,yw,zw) ----------> (xv,yv,zv)
Viewing transformation

3-D Viewing Transformation

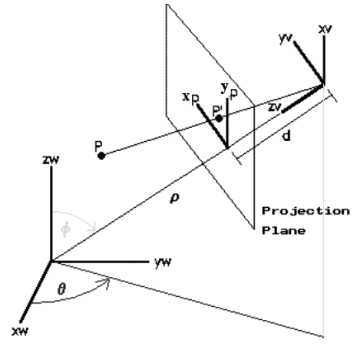


Projection Transformation

- Converts viewing coordinates (xv,yv,zv) of a point to 2-D coordinates (xp,yp) of point's projection onto a projection plane
- Think of projection plane as containing screen upon where image is to be displayed

(xv,yv,zv) ----------> (xp,yp)
Projection transformation

4-Parameter Viewing Setup



Composite Viewing Transformation Matrix

- Tv = T4*T3*T2*T1
- Result:

$$Tv = \begin{vmatrix} -\sin(\theta) & \cos(\theta) & 0 & 0 \\ -\cos(\phi)*\cos(\theta) & -\cos(\phi)*\sin(\theta) & \sin(\phi) & 0 \\ -\sin(\phi)*\cos(\theta) & -\sin(\phi)*\sin(\theta) & -\cos(\phi) & \rho \\ 0 & 0 & 0 & 1 \end{vmatrix}$$

Projection Transformation

- Look down xv axis at viewing setup:

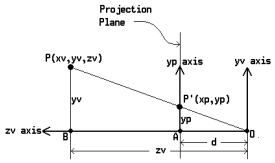
Triangles OAP' & OBP are similar

So set up proportion:

$$\frac{yp}{yv} = \frac{d}{zv}$$

Solve for yp:

$yp = (yv*d)/zv$



Look down yv axis for xp:

Result: $xp = (xv*d)/zv$

Plotting Points on Screen

- Get screen coordinates (xs,ys) from Projection Plane coordinates (xp,yp)
- Final Transformation:

  2D Window-to Viewport Transformation

  (xs,ys) <--- (xp,yp)

  See earlier notes
  - Replace xv,yv with xs,ys
  - Replace xw,yw with xp,yp

Skeleton Pyramid Program: Data Structures

```
// Build/display polygon mesh model of  4-sided pyramid
struct point3d {float x; float y; float z;};  // a 3d point
struct polygon {int n; int *inds;};
struct point3d w_pts[5];  // 5 world coord. vertices
struct point3d v_pts[5];   // 5 viewing coord. vertices
POINT s_pts[5];              // 5 screen coord. vertices
struct polygon polys[5];    // 5 polygons define pyramid
// global variables:
float v11,v12,v21,v22,v23,v31,v32,v33,v43; // view xform matrix elts
int screen_dist; float rho, theta, phi; // viewing parameters
int xmax,ymax;           // Screen dimensions
int num_vertices=5, num_polygons=5;
```

Skeleton Pyramid Program: Function Prototypes

```
void coeff (float r, float t, float p);  // calculates xform matrix elements
void convert (float x, float y, float z,
        float *xv, float *yv, float *zv,
        int *xs, int *ys);       // converts a 3D world coordinate pt to
                                 // 3D viewing & 2D screen coords
void build_pyramid (void);   // sets up pyr. points and polygons arrays
void draw_polygon (int p);   // draws polygon # p if polygon is visible
float visible(int p);        // returns a negative # if polygon p is visible
                             // or back-face culling is turned off
```

Skeleton Pyramid Program: Function Skeletons

```
//    Main Function--Called when pyramid is to be displayed
//     hide_flag determines if back-face culling is to be done
void main_ftn (int hide_flag)
{
// Set values of rho, theta, phi, and screen_dist here
build_pyramid ();       // build polygon model of the pyramid
coeff (rho,theta,phi);  // compute transformation matrix elements
for (int i=0; i<num_vertices; i++)
  { // Loop to convert polygon vertices from world coordinates
    // to viewing and screen coordinates; must call convert () }
for (int f=0; f<num_polygons; f++)
  { // Loop to draw each polygon if hide_flag is off or if polygon
    // is visible; must call visible () and draw_polygon ()}
}
```

```
void coeff (float r, float t, float p)
{ // Code to compute non-trivial viewing transformation matrix
  // elements: v11,v12,v21,v22,v23,v31,v32,v33,v34 }

void convert (float x, float y, float z,
              float *xv, float *yv, float *zv, int *xs, int *ys)
{ // Code to compute viewing coordinates and screen coordinates of
  // a point from its 3-D world coordinates. Must implement viewing,
  //  perspective, and window transformations described in class }

void build_pyramid (void)
{ // Code to define the pyramid by setting up w_pts & polys arrays }
```

```
float visible (int p)
{ // Code to compute and return value of visibility determinant for
  // polygon p. Negative means invisible, positive visible. }


void draw_polygon (int p)
{ // Code to draw polygon # p by obtaining its vertex numbers from
  // polys array, getting the screen coordinates of the vertices from
  // s_pts array, and making appropriate calls to the system
  // polygon-drawing primitive }
```

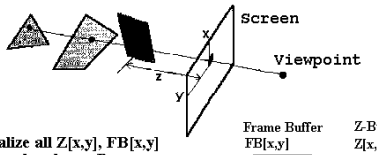
Z-Buffer Hidden Surface Removal Algorithm

- Basic Idea:
  - At a given pixel we want to plot color of closest surface that projects to that pixel
  - We're looking for minimum zv
  - Use a buffer (array) parallel to the frame buffer
    - Store minimum values of zv
    - One for every pixel
    - Called the Z-Buffer

Z-Buffer Technique Applied to a Polygon Mesh

- Initialize Z-Buffer and Frame Buffer
- Look at each polygon
  - Look at each point (xs,ys) projected to by the polygon
  - Compute zv of the point on the polygon
    - If zv is closer than value stored at [x,y] in Z-Buffer
      - Replace value in Z-Buffer with zv
      - Update corresponding element in frame buffer with color of the polygon

Z-Buffer Hidden Surface Removal



Initialize all Z[x,y], FB[x,y]
For each polygon P
For each pixel (x,y) covered by P
if (z < Z[x,y])
Z[x,y] = z
FB[x,y] = color of P

Frame Buffer FB[x,y]

b	b	b	b	b
b	b	b	b	b
b	b	b	b	b

b = background color initially

Z-Buffer Z[x,y]

i	i	i	i	i
i	i	i	i	i
i	i	i	i	i

i = infinity (largest value) initially

When completed, each position (pixel) in the frame buffer will contain the color of the closest polygon, and each position in the Z-buffer will contain the distance to the intersection with that polygon.

Z-Buffer Algorithm Applied to Convex Polygons

Data Structures:

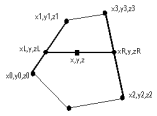
- For each polygon:
  - Polygon color
  - For each polygon vertex:
    - Edge table with vertex xs, ys, and zv
      - Note mixed coordinates
    - Active edge list (AEL) with active edges intersected by current scanline sorted on xs
      - (See scanline polygon fill notes)

Other Data Structures

- Frame Buffer fbuf[x][y]
  - Will store the color of each pixel (x,y)
- Z-Buffer zbuf[x][y]
  - Will store the zv distance of point on closest polygon that projects to pixel (x,y) on screen
- Initialize each element of fbuf[][] to background color
- Initialize each element of zbuf[][] to infinity (largest possible value)

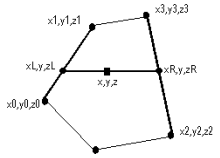
The Algorithm

For each polygon
  For each scanline y spanning the polygon
    Get left & right active edges from AEL
    Get x,z coordinates of endpoints from edge table
    Compute scanline intersection pts (xL,zL,xR,zR)
      (Use x-y & z-y interpolation)
    For (x=xL to xR)
      Compute z by z-x interpolation
      If (z < zbuf[x,y])
        zbuf[x,y] = z
        fbuf[x,y] = polygon color



Double Interpolation

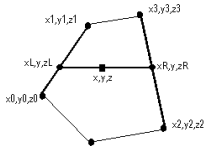
- We know (from Edge Table):
  - lower/upper vertices of left active edge:
    (x0,y0,z0) and (x1,y1,z1)
  - lower/upper vertices of right active edge:
    (x2,y2,z2) and (x3,y3,z3)
- Also know y of current scanline



x-y Interpolation:

- Find x coords of intersection pts (xL,xR)
- Left Edge:

  xL-x0 y-y0
  ------- = -------
  x1-x0 y1-y0



- Solving for xL:
- xL = (x1-x0)*(y-y0)/(y1-y0) + x0
- Similarly for xR on right edge:

  xR = (x3-x2)*(y-y2)/(y3-y2) + x2

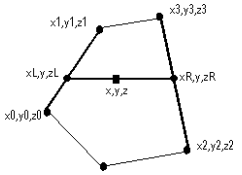
z-y Interpolation

- Find z coordinates of intersection points of scan line (y) with left and right edges
- Done the same way as x-y interpolation
- x coordinates replaced by z coordinates
- Results:
- zL = (z1-z0)*(y-y0)/(y1-y0) + z0
- zR = (z3-z2)*(y-y2)/(y3-y2) + z2

z-x Interpolation

- Find z value on polygon at pixel x on current scanline (y)
- Interpolate between the left and right edge intersection points:

  z-zL x-xL
  -------- = --------
  zR-zL xR-xL



Solving for z:

  z = (zR-zL)*(x-xL)/(xR-xL) + zL

Speeding up the Algorithm

- Do interpolations incrementally
  - Get new values from old values by adding correct increments
  - xL,xR,zL,zR (in the outer loop)
  - z (in the inner loop)
  - Avoids multiplications and divisions inside algorithm loops

Z-Buffer Performance

- Outer loop repeats for each polygon
- Complex scenes have more polygons
  - So complex scenes should be slower
- But: More polygons usually means smaller polygons
  - So inner loops (y and x) are faster
- For most real scenes, performance is approximately independent of scene complexity

Disadvantage of Z-Buffer

- Memory requirements
- Z-Buffer at least as big as the frame buffer
- For best results, need floating point or doubles for z values
- Example 1000 X 1000 resolution screen
  - Assume 8 bytes to store a double
  - 8 Megabytes required for Z-Buffer
- But memory has become cheap
- Z-Buffer used very commonly now!