Florida Tech

CSE 5255

INTRODUCTION TO COMPUTER GRAPHICS CLASS NOTES

Dr. William D. Shoaff

Spring 1996

Table of Contents

Hidden Object Removal

- Hidden Object Algorithm Basics
- Back-face Culling
- Floating Horizon Algorithm
- z-Buffer Algorithm
- Scan-line Algorithm
- Painter's Algorithm
- Binary Space-Partition Trees
- Warnock's Algorithm

Hidden Object Algorithm Basics

- Hidden object algorithms involve (explicit or implicit) depth sorting (depth priority)
- *Coherence* – the tendency of characteristics in a scene to be locally constant can be used to increase the efficiency of hidden object algorithms
  - *span coherence* — scan lines contain runs of constant intensity
  - *scan-line coherence* — patterns are similar from scan line to scan line
  - *edge coherence* — edge changes visibility only where it crosses a visible edge or penetrates a face
  - *depth coherence* — adjacent parts on the same surface are typically close in depth
  - *frame coherence* — scenes are similar from frame to frame

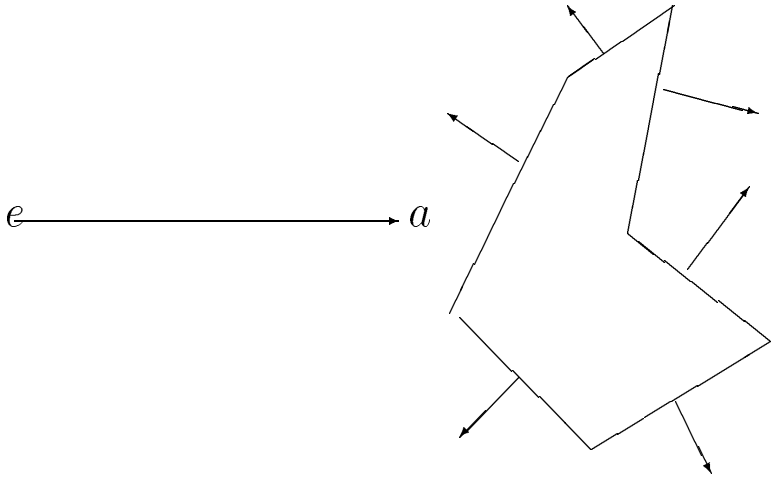
- Some hidden object algorithms are *object space* algorithms that work in floating point world coordinates
  - Object space algorithms are called *continuous*; they perform visibility determination over continuous areas giving precise results
  - Images can be enlarged without recomputation
  - Useful in precise engineering applications
  - A general rule-of-thumb is that object space algorithms are $O(n^2)$ where n is the number of objects in the scene
  - Complexity of algorithms is high, few primitives are supported, and special effects often are difficult to implement
  - Hidden-line algorithms are often implemented in object space

- Some hidden object algorithms are *image space* algorithms that work in integer device coordinates
  - Image space algorithms are called *point sampling*; they perform visibility determination over discrete areas giving crude results
  - Enlarged scenes are unacceptable
  - A general rule-of-thumb is that image space algorithms are $O(nN)$ where n is the number of objects in the scene and N is the number of pixels
  - Allow more advanced effects and complex models
  - Many hidden-surface algorithms are implemented in image space
- List-priority algorithms are partially implemented in both coordinate systems

Backface Culling

- Make a rule that solid objects are made of polygons, closed, and surface normals point into open space (outward)
- If a polygon is drawn in a counterclockwise direction, so its interior is on the left, then the normals calculated by cross products or Newell's method will point out from the visible side
- Let $\vec{N}$ be a polygon's surface normal and let $\vec{V} = \langle p_x - e_x,\, p_y - e_y,\, p_z - e_z \rangle$ be the direction vector from the eye to any point p on the polygon
- Since we are only interested in direction, neither $\vec{N}$ nor $\vec{V}$ need to be unit vectors

- If the inner product $(\vec{N} \cdot \vec{V}) \geq 0$, then the polygon's visible face can not be seen by the viewer
- If the scene is in eye coordinates, so that $\vec{V} = \langle 0,\ 0,\ 1 \rangle$, then the polygon's visible face can not be seen by the viewer if the z component N_z of the surface normal is greater than or equal to zero



- Consider the example
  - Let $P_1 = (1,\, 2,\, 4), P_2 = (2,\, 1,\, 4), P_3 = (3,\, 3,\, 4)$ define a triangle
  - Newell's method gives a normal with components

$$\begin{aligned} N_x &= (2-1)(4+4) + (1-3)(4+4) + (3-2)(4+4) = 0 \\ N_y &= (4-4)(1+2) + (4-4)(2+3) + (4-4)(3+2) = 0 \\ N_z &= (1-2)(2+1) + (2-3)(1+3) + (3-1)(3+2) = 3 \end{aligned}$$

  - If $\vec{V} = \langle 0,\, 0,\, 1 \rangle$, then the face is visible
  - If $\vec{V} = \langle 0,\, 0,\, -1 \rangle$, then the face is not visible
- Note backface culling can not be used for open solids
- About a 50% savings in computation is expected by backface culling

The Visible Surface Problem

- The visible surface problem can be formulated as:
- Given —
  - A set of surfaces in three dimensions
  - A viewpoint
  - An oriented image (view) plane
  - A field of view
- For each point on the image plane within the field of view determine which point on which surface lies closest to the viewpoint along a line from the viewpoint through the point in the image plane

- Useful for rendering a mathematically defined surface
- The algorithm can be implemented in object space (with a rootfinding algorithm, which may be expensive,) or in image space (where aliasing may occur).
- Given a surface represented in the implicit form $F(x, y, z) = 0$
- Surface is represented by curves drawn on it, say curves of constant height $z = c$ for multiple values of c
- Convert the 3D problem to 2D by intersecting the surface with a series of cutting planes at constant values of z
- The function $F(x,\ y,\ z) = 0$ is reduced to a curve in each of these parallel planes,

  $$y = f(x,\ z)$$

  where z is constant for each of the planes

```
Floating-Horizon
{
  horizon[0.. max_x] = min_y;
  for z = near to far do
    yprev = f(0, z);
    for x = 1 to x = max_x do
      y = f(x, z);
      if y > horizon[x] then draw(x-1, yprev, x, y);
      yprev= y;
}
```

- The surface may dip below the closest horizon and the bottom of the surface should be visible
  - Keep two horizon arrays, one that floats up and one that floats down
- In image space, aliasing can occur when curves cross previously drawn curves
- Example:
  - Let curves at $z = 0$, $z = 1$ and $z = 2$ be defined by

$$y = \begin{cases} 20 - x & \text{for } 0 \leq x \leq 20 \\ x - 20 & \text{for } 20 \leq x \leq 40 \end{cases},$$

$$y = \begin{cases} x & \text{for } 0 \leq x \leq 20 \\ -x + 4020 & \text{for } 20 \leq x \leq 40 \end{cases},$$

$$y = 10,$$

respectively

- Set $h[0..40] = 0$ and for $z = 0,\ 1,\ 2$ compute the value of y saving higher y and drawing the curves

The z-Buffer (Depth-Buffer) Algorithm

- An image space, point-sampling method
- Polygons are tested one at a time
- At each pixel position on the view plane, the polygon with the closest z coordinate is visible
- Requires two buffers
  - A frame (refresh) buffer that store intensities
  - A z (depth) buffer stores z values for each pixel

z-buffer
{
  for *each pixel store* $\max_z$ *in depth buffer*;
  for *each pixel store background in frame buffer*;
  for *all objects*
    for *all covered pixels* (x, y)
      compute z *on object*;
      if z *is closer than stored* z **then**
        store z **and** *paint pixel the object's color*;
}

The z-Buffer (Depth-Buffer) Algorithm

- For each pixel position $(n,\ m)$ initialize depth buffer

$$\texttt{depth}(n, m) = \texttt{max\_depth}$$

- For each pixel position $(n,\ m)$ initialize frame buffer

$$\texttt{refresh(n, m)} = \texttt{background}$$

- For each pixel in polygon's projection calculate its z value at (n, m)
- If $z <$ depth(n, m), then

$$\text{depth}(n, m) = z,\ \text{refresh}(n, m) = I,$$

  where I is the intensity of the polygon at position $(n,\ m)$
- Let $Ax + By + Cz + D = 0$ be polygon's plane equation
- The depth is

$$z = -\frac{Ax + By + D}{C}$$

- Depth at next pixel $(x + 1,\ y)$ on scan line y is

$$z - \frac{A}{C}$$

- Depth at next scan line $(x,\ y - 1)$ is

$$z + \frac{B}{C}$$

The z-Buffer (Depth-Buffer) Algorithm

- The z-buffer algorithm does not require objects to be polygons (all we need to be able to determine is a shade and a z value)
- The z-buffer algorithm performs a radix sort in x and y, requiring no comparisons
- The z sort takes one comparison per pixel for each polygon containing the pixel
- The z-buffer algorithm requires a large amount of space

The Scan-line Algorithm

- An image space, point-sampling method
- For each scan line, all polygon surfaces intersecting the scan line are examined to determine which is visible at each pixel along the scan line
- Extension of polygon scan-conversion filling algorithm
- Uses scan-line coherence and edge coherence
- Extend node used in scan-line filling to include ID of polygon being filled

$$\boxed{x,\ \delta x,\ \delta y,\ \text{id}}$$

- Also require a polygon table that contains
  - Polygon's ID
  - Coefficient of the plane equation
  - Shading or color information
  - In-out boolean flag, initialize to FALSE

```
{
  sort objects by y;
  for all y
    sort objects by x;
    for all x
      compare z;
}
```

```
Scan-Line
{
  for each polygon P in the polygon list
    insert P in a y-bucket according
      to its maximum y-value;
  for scanline = top to bottom
    if y-bucket[scanline] is not empty
      for each P in y-bucket[scanline]
        calculate the endpoints of P that
          intersect scanline;
        insert the endpoints into the
          x-sorted list;
  for pixel = left-pixel to right-pixel
    construct z-depth list from the
      x-sorted list;
    construct visible element(s) from the
      z-depth list;
    determine the pixel intensities from the
      visible elements(s);
  update the x-sorted list for next scanline;
}
```

The Painter's (Depth-Sorting) Algorithm

- Sort surfaces in order of decreasing depth (in object space)
- Scan-convert surfaces in order, starting with surface of greatest depth

```
Painter
{
   sort objects by z;
   for all objects
      for all covered pixels (x, y)
         paint;
}
```

The Painter's (Depth-Sorting) Algorithm

- Sort polygons according to largest z value
- If surface of greatest depth does not overlap any other surface (in z) scan-convert it and proceed to polygon of next greatest depth
- If overlap in z
  - Do the two polygons overlap in x?
  - Do the two polygons overlap in y?
  - Is greatest depth polygon entirely on far side of other polygon's plane?
  - Is nearer polygon entirely on near side of deeper polygon's plane?
  - Do the projections of the polygons onto the (x, y) plane not overlap?

- If all five tests fail, we ask if nearer plane can be scan-converted before farther plane
  - Questions 1, 2, and 5 do not need to be asked again
  - Replace question 3 with: is nearer polygon entirely on far side of farther polygon's plane?
  - Replace question 4 with: Is farther polygon entirely on near side of nearer polygon's plane?
- If either of these two tests succeed, move nearer polygon to the end of the list and it becomes the new farthest polygon to test
- If all tests fail, split either the farther or nearer polygon by the plane of the other, discard the old polygon and put its pieces in order on the list
- Possible to have infinite loops when nearer polygons are cyclically moved to end of the list

Binary Space-Partition Tree

- Build a binary tree (BSP-tree) that determines the correct order (back-to-front) to scan convert polygons *from any view point*
- Pick any polygon to be displayed as the root of the tree
- Partition space into the *front* half-space containing all polygons in front of the root polygon (relative to its surface normal) and the *back* half-space containing all polygons behind the root polygon
- Recursively repeat the space partitioning until each node contains a single polygon
- Polygons may need to be split in two by the plane of a root polygon
- To display the scene, from the current viewpoint, use modified in-order tree traversal
- If the viewer is in the roots front half-space, visit the sub-tree in the back half-space, and vice versa

Pseudocode for Building BSP tree

```
struct BSP-tree {
   Polygon *polygon;
   struct BSP-tree *front, *back;
};
BSP-tree MakeTree(polylist)
{
   if polylist is empty return NULL;
   else
      root = select-and-remove-poly(polylist);
      front = NULL;
      back = NULL;
      for each p in polylist
         if p in front of root
            add-to-list(p, front-list);
         else if (pin back of root)
            add-to-list(p, back-list);
         else
            split-poly(root, p ,f-poly,b-poly);
            add-to-list(f-poly, front-list);
            add-to-list(b-poly, back-list);
   return combine-trees(MakeTree(front-list),
                        MakeTree(back-list));
}
```

Pseudocode for Displaying BSP tree

```
void DisplayTree(tree)
{
  if tree not empty
    if viewer in front of root
      DisplayTree(tree→back);
      DisplayPolygon(tree→polygon);
      DisplayTree(tree→front);
    else
      DisplayTree(tree→front);
      DisplayPolygon(tree→polygon);
      DisplayTree(tree→back);
}
```

Warnock's Algorithm

- Object/Image space, continuous algorithm
- Uses area coherence
- If easy to decide what polygons are visible in an area, display them
- Otherwise, divide the area into smaller areas
- Given an area of interest, classify polygons to be displayed as
  - *Surrounding* completely containing the area
  - *Intersecting* intersecting the area
  - *Containing* completely inside the area
  - *Disjoint* completely outside the area
- Disjoint polygons can be eliminated from consideration
- Intersecting polygons can be split into disjoint and containing polygons

Warnock's Algorithm

- If all polygons are disjoint from the area, display the background color
- There is only one containing polygon (perhaps derived from an intersecting polygon), fill area with background, then scan convert containing polygon
- There is a single surrounding polygon, and no intersecting or containing polygons, display the surrounding polygon's color in the area
- There is a surrounding polygon in front of all other polygons, display the surrounding polygon's color in the area
- A surrounding polygon can be determined to be in front by computing the z coordinates of the surrounding, intersecting, and containing polygons at the four corners of the area

Warnock's Algorithm

- If we can not determine how to display the area, subdivide it into four sub-rectangles and repeat the process
- We can stop subdividing when the resolution of the display is reached and display the polygon with closest z value at the pixel (if none of the four cases have occurred)
- Alternatively, for antialiasing, we can subdivide further and use a color weighting
- Except when none of the four cases occur, the algorithm can be implemented in object space
- Jim Blinn — CG&A (January 1990, Vol 10, No. 1, pp 70 – 75) describes an efficient *triage table* for maintaining the lists of polygons

Problems

1. What is back-face culling?

2. How is testing for a back-face done?

3. Let $3x - 2y + 6z - 5 = 0$ be the equation of a plane viewed from eye position $e = (1, 1, 1)$, while looking at the point $a = (0, 0, 0)$. Is the front of the plane visible?

4. What is the difference between an image space and an object space hidden surface algorithm?

5. What is coherence?

6. Let curves at $z = 0$, $z = 1$ and $z = 2$ be defined by

$$y = (x - 20)^2 \quad \text{for } 0 \leq x \leq 40,$$

$$y = -(x - 20)^2 + 400 \quad \text{for } 20 \leq x \leq 40,$$

and

$$y = 300,$$

Show how the floating horizon algorithm would draw the surface.

7. Describe the basic flow of control for the z-buffer algorithm.

8. What incremental technique can be used in the z-buffer algorithm?

9. Suppose you are given a 2×2 frame buffer and z buffer initialized as shown to the background color and the far value of z.

$$F = \begin{bmatrix} 0.0 & 0.0 \\ 0.0 & 0.0 \end{bmatrix} \qquad Z = \begin{bmatrix} 1023 & 1023 \\ 1023 & 1023 \end{bmatrix}$$

Show the contents of both the frame buffer and the z buffer after processing polygons P_1, P_2 where

$$P_1 = \begin{bmatrix} (512, 0.4) & (128, 0.3) \\ (1023, 0.7) & (412, 0.5) \end{bmatrix} \qquad P_1 = \begin{bmatrix} (612, 0.8) & (72, 0.6) \\ (300, 0.5) & (418, 0.2) \end{bmatrix}$$

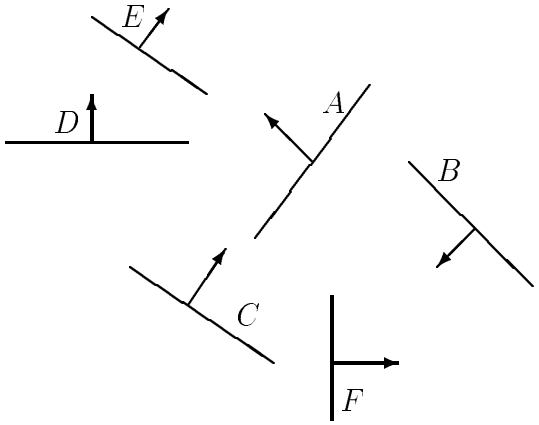
Here the notation (z, I) at each pixel indicates the z depth and the intensity of the polygon at the pixel.

Problems

10. What is the basic concept of the Painter's algorithm?

11. What is an important advantage of a binary space partition tree?

12. Show how to construct a binary space-partition tree from the polygons displayed below starting with polygon A as the root. Note the polygons are displayed as line segments (think of viewing them on their edges) with an arrow indicating the front side of the polygon. Clearly explain how your tree is built.



13. What are the basic tests in Warnock's algorithm?