Hidden Surface Algorithms

Ron Goldman
Department of Computer Science
Rice University
Polygonal Models

Standard Assumptions (Always)

1. Surfaces are planar polygons
2. Surfaces are opaque

Additional Assumptions (Sometimes)

3. Surfaces completely enclose a solid
4. Surfaces form a manifold -- No dangling polygons or edges

Objective

Display only those polygons visible to the eye
Cull Backfacing Polygons

Test

\[ N \cdot (E - Q) \leq 0 \]

- \( N \) = normal to polygon pointing out of the solid
- \( E \) = eye position
- \( Q \) = any point on the polygon (e.g. a vertex)

Remove backfacing polygons before

- Perspective (Pseudoperspective) -- need the eye point
- Proceeding with other algorithms -- faster
Hidden Surface Algorithms

Types

- Object Space = Model Space -- Before Perspective Projection
- Image Space = Pixel Space (Frame Buffer) -- After Perspective Projection

To Increase Speed

- Store additional information
- Use pixel coherence

Main Technique

- Sorting
**Object Space**

*Algorithm*

For each polygon
   Find unobstructed part
   Render it

*When*

Before perspective (pseudoperspective)

*Speed*

$O(n^2)$ -- $n =$ number of polygonal faces
**Image Space**

**Algorithm**
For each pixel
   Find the closest object
   Render it

**When**
After perspective (pseudoperspective)

**Speed**
$O(nN)$

- $N =$ number of pixels
- $n =$ number of polygonal faces

**Remark**
Most raster display algorithms work in image space.
Image Space Algorithms

1. Heedless Painter

2. Z-Buffer (Depth Buffer)

3. Scan Line Algorithm
Heedless Painter

Algorithm

- Paint polygons in order in which they occur in some list
- Overpaint preceding polygons

Problems

- Slow -- painting same pixel many times
- May be incorrect -- give an example
**Z-Buffer (Depth Buffer)**

**Data Structure**

Z-Buffer = Large Array

**Algorithm**

- For each pixel, store the following information
  - Current depth (Depth buffer)
  - Current color and intensity
- Use painter’s algorithm -- visit each face in turn
- Overwrite pixel only if $depth < current\ depth$
- Compute depth of each pixel in polygon incrementally

**Advantages**

- Simple to understand
- Easy to implement

**Disadvantages**

- Slow -- paints same pixel many times
- Memory intensive
Linear Interpolation

Derivation

\[ L(t) = (1 - t)P_1 + tP_2 \]
\[ L(t) = P_1 + t(P_2 - P_1) \]
\[ L(t + \Delta t) = P_1 + (t + \Delta t)(P_2 - P_1) \]
\[ \Delta L = \Delta t(P_2 - P_1) \]

- \[ \Delta x = \Delta t(x_2 - x_1) \]
- \[ \Delta y = \Delta t(y_2 - y_1) \]
- \[ \Delta z = \Delta t(z_2 - z_1) \]

\[ z_{\text{new}} = z_{\text{old}} + \Delta z \]
Incremental Depth Computation

Along a Scan Line

\[ P_1 = (x_1, y_1, z_1) \]
\[ (x, y, z) \]
\[ (x + \Delta x, y + \Delta y, z + \Delta z) \]
\[ P_2 = (x_2, y_2, z_2) \]

\[ \Delta x = 1 \Rightarrow \Delta t = 1/(x_2 - x_1) \quad \Delta y = 0 \]
\[ \Delta z = (z_2 - z_1) \Delta t = (z_2 - z_1)/(x_2 - x_1) \]

Next Scan Line

\[ P_1 = (x_1, y_1, z_1) \]
\[ (x, y, z) \]
\[ (x + \Delta x, y + \Delta y, z + \Delta z) \]
\[ P_2 = (x_2, y_2, z_2) \]

\[ \Delta y = 1 \Rightarrow \Delta t = 1/(y_2 - y_1) \]
\[ \Delta z = (z_2 - z_1) \Delta t = (z_2 - z_1)/(y_2 - y_1) \]
Scan Line Algorithm

Data Structure

- For each scan line, maintain an Active Edge List
- For each edge, store a pointer to its polygon
- For each face, store
  i. whether or not it covers the current pixel
  ii. depth at the current pixel -- computed incrementally

Active Edge List (AEL)

- \( \text{Edge Data} = (Y_{\text{max}}, \text{current } X_{\text{int}}, \text{current } Z_{\text{int}}, \Delta x, \Delta z) \)
- Introduce new edges as they become active -- see Edge Table
- Remove old edges as they become inactive -- see \( Y_{\text{max}} \)

Edge Table

- For each scan line, a list of those edges whose lower vertex lies on the line.
- Data -- Same as AEL
Polygon
Scan Line Algorithm (continued)

Algorithm

For each scan line:

• Update the AEL.
• Select the polygon corresponding to the first edge in the AEL.
• Fill with the current polygon color/intensity until a closer polygon is encountered along the scan line.
  
i. If the next polygon in the AEL has both end points behind the current polygon’s end points, then there is no need to switch polygons (look ahead in the AEL). Fill the run with the current polygon.
  
ii. If the next polygon to enter the AEL has an end point in front of one of the current polygon’s end points, then compute where the cross over occurs (see below) and switch polygons at the crossover.
Active Edge List (AEL)

Edge Data
$(Y_{\text{max}}, \text{current } X_{\text{int}}, \text{current } Z_{\text{int}}, \Delta x, \Delta z)$

Update Algorithm
For each edge in the active edge list,
  If $Y_{\text{scan line}} > Y_{\text{max}}$, delete the edge from the active edge list.
  Else update the values of $\text{current } X_{\text{int}}$ and $\text{current } Z_{\text{int}}$ using $\Delta x$ and $\Delta z$.
For each edge in the edge table for the current scan line ($Y_{\text{scan line}}$),
  Insert the edge to the active edge list.
  Sort the active edge list by increasing values of $\text{current } X_{\text{int}}$. 
Crossover

Scan Line

Scan Line
Crossover

Problem
Given the depths $z,z^*$ of 2 polygons at the same point along a scan line, find the pixel at which the polygons intersect.

Solution
\[ z + N\Delta z = z^* + N\Delta z^* \]
\[ N = \frac{z^* - z}{\Delta z - \Delta z^*} \]

- $z,z^*$ are known -- current depths
- $\Delta z, \Delta z^*$ are known -- see z-buffer algorithm
Scan Line Algorithm (continued)

Observations

• Only one crossover can occur per polygon pair, but a polygon may become inactive by ending.

• Whenever a covering polygon exits the active list, choose the polygon that is next closest.

• Look ahead in polygon list for intersections.

• Compute depth of each pixel in polygon incrementally.

• Fast because of scan line coherence.

• Integrates well with Gouraud and Phong shading algorithms.
Object Space Algorithms

1. Ray Casting
2. Depth Sort
3. BSP-Tree
Ray Casting

Algorithm

Through each pixel, fire a ray to the eye:

• Intersect ray with each polygonal plane.
• Reject intersections that lie outside the polygon.
• Accept the closest remaining intersection -- smallest $t$.

Advantages

• Can use with non-polygonal surfaces -- e.g. spheres.
• Requires only Line/Surface intersection algorithm.

Disadvantages

• Through each pixel, instead of for each polygon.
• No coherence -- dense sampling -- slow.
Ray Casting (continued)

Intersection of Line and Plane

Line Equation (Parametric): \( L(t) = E + tv \)
Plane Equation (Implicit): \( N \cdot (P - Q) = 0 \)
Solve Linear Equation:
\[
N \cdot (L(t) - Q) = N \cdot (E + tv - Q) = 0
\]
\[
t = \frac{N \cdot (Q - E)}{N \cdot v}
\]

Inside/Outside Test

- Convex Polygon -- Half plane test \( N \cdot (P - Q) > 0 \)
- Concave Polygon -- Fire ray in plane of polygon
  - # intersection even \( \Rightarrow \) outside
  - # intersection odd \( \Rightarrow \) inside
Point Inside Polygon

Test 1: (Convex Polygons)
\[(Q - P_i) \cdot N_{i,i+1} \geq 0\] for all inward normals $N_{i,i+1}$

Test 2: (Arbitrary Polygons)
\[\#(L \cap Polygon)\] is odd for any ray $L$ through $Q$
Ray Casting (continued)

Object Space Algorithm -- Before Perspective (Pseudoperspective)
• Ray from Eye in direction of Pixel
• Accept closest intersection to Eye inside the viewing frustum (yet another inside/outside test)

Image Space Algorithm -- After Pseudoperspective
• Ray from Pixel (near plane) in direction orthogonal to screen towards far plane
• Accept closest intersection to Pixel
**Depth Sort**

*Data Structure*
Sorted List of Polygons

*Algorithm*
Sort polygons by furthest vertex from screen
- For each polygon find the minimum and maximum values of $z$ at the vertices
- Sort the polygons in order of decreasing maximum $z$-coordinates

Resolve $z$-overlaps
Paint in order from farthest to closest

*Observations*
Without second step, this is Heedless Painter’s Algorithm
Sorting is done in object space; painting in image space
Resolving Overlaps

Algorithm
If z-extents of $P$ and $Q$ overlap, then perform 5 tests:

i. Do $x$–extents fail to overlap?

ii. Do $y$–extents fail to overlap?

iii. Does every vertex of $P$ lie on the far side of $Q$?

iv. Does every vertex of $Q$ lie on near side of $P$?

v. Do the $xy$-projections of $P$ and $Q$ fail to overlap (clipping)?

If any test succeeds, $P$ does not obscure $Q$, so draw $P$ then $Q$.
Otherwise we cannot resolve the conflict so split $Q$ (clip by $P$).

Cycle Problem
$P$ obscures $Q$, $Q$ obscures $R$, $R$ obscures $P$.
Avoid by marking and splitting.
Near Side and Far Side

Far Side

Screen

P

Q

\((V_P - V_Q) \cdot N_Q > 0\)

Near Side

Screen

Q

P

\((V_P - V_Q) \cdot N_P > 0\)
Binary Space Partitioning Trees (BSP–Trees)

Data Structure
Binary Tree
Nodes = Polygonal Faces

Algorithm for Generating the BSP–tree
Select a face (plane)
Partition all polygons to back (left subtree) or front (right subtree)
Split any polygons lying on both sides
Build binary tree recursively
BSP–Trees (continued)

Algorithm for Display

(In order tree traversal of BSP–tree)

If eye in front of root, then

   Display left subtree (behind)

   Display root

   Display right subtree (front)

If eye in back of root, then

   Display right subtree (front)

   Display root

   Display left subtree (back)

Advantages

Can use same BSP–tree for different eye points.
Summary

**Image Space Algorithms**
- Z-Buffer -- coherence for calculating depths
- Scan Line -- coherence for depth and sorting of edges (in AEL)
- Ray Casting -- no sorting, no coherence

**Object Space Algorithms**
- Ray Casting -- no sorting, no coherence
- Depth Sort -- sorting
- Binary Space Partitioning Tree -- tree sort

**Main Ideas**
- Coherence
- Sorting
Final Observations

1. We can speed up some of these algorithms by storing bounding boxes for each face and avoiding tests when the bounding boxes do not overlap

2. We can use the same algorithms for computing
   a. Hidden Surfaces -- Invisible to eye
   b. Shadows -- Invisible to light source

3. We must perform hidden surface algorithms twice
   a. Once for each light source -- shadows
   b. Once for eye -- hidden surfaces

4. We can reuse calculations
   a. Can move eye point and reuse shadow calculations
   b. Can move light source and reuse hidden surface calculations