Z-buffer and Rasterization

Jian Huang
Visibility Determination

• AKA, hidden surface elimination
Hidden Lines

Wireframe
Hidden Lines Removed

Hidden Line Removal
Hidden Surfaces Removed

Hidden Surface Removal
Various Algorithms

- Backface Culling
- Hidden Object Removal: Painters Algorithm
- Z-buffer
- Spanning Scanline
- Warnock
- Atherton-Weiler
- List Priority, NNA
- BSP Tree
- Taxonomy
Where Are We?

- Canonical view volume (3D image space)
- Clipping done
- Division by \( w \)
- \( z > 0 \)
Back-face Culling

- Problems?
- Conservative algorithms
- Real job of visibility never solved
Back-face Culling

• If a surface’s normal is pointing to the same direction as our eye direction, then this is a back face
• The test is quite simple: if $\mathbf{N} \cdot \mathbf{V} > 0$ then we reject the surface
Painters Algorithm

- Sort objects in depth order
- Draw all from Back-to-Front (far-to-near)
- Is it so simple?

\[
\begin{array}{c}
\text{ at } z = 22, \\
\text{ at } z = 18, \\
\text{ at } z = 10,
\end{array}
\]
3D Cycles

- How do we deal with cycles?
- Deal with intersections
- How do we sort objects that overlap in Z?
Form of the Input

Object types: what kind of objects does it handle?

- convex vs. non-convex
- polygons vs. everything else - smooth curves, non-continuous surfaces, volumetric data
Object Space
- Geometry in, geometry out
- Independent of image resolution
- Followed by scan conversion

Image Space
- Geometry in, image out
- Visibility only at pixels

Precision: image/object space?

Form of the output

Object Space
- Geometry in, geometry out
- Independent of image resolution
- Followed by scan conversion

Image Space
- Geometry in, image out
- Visibility only at pixels
Object Space Algorithms

- Volume testing – Weiler-Atherton, etc.
  - input: convex polygons + infinite eye pt
  - output: visible portions of wireframe edges
Image-space algorithms

- Traditional Scan Conversion and Z-buffering
- Hierarchical Scan Conversion and Z-buffering
  - input: any plane-sweepable/plane-boundable objects
  - preprocessing: none
  - output: a discrete image of the exact visible set
Conservative Visibility Algorithms

- Viewport clipping
- Back-face culling
- Warnock's screen-space subdivision
Z-buffer

- Z-buffer is a 2D array that stores a depth value for each pixel.

- **InitScreen**: for \( i := 0 \) to \( N \) do 
  - for \( j := 1 \) to \( N \) do 
    - \( \text{Screen}[i][j] := \text{BACKGROUND\_COLOR}; \ Zbuffer[i][j] := \infty; \)

- **DrawZpixel** \((x, y, z, \text{color})\)
  - if \( (z \leq \text{Zbuffer}[x][y]) \) then 
    - \( \text{Screen}[x][y] := \text{color}; \ Zbuffer[x][y] := z; \)
Z-buffer: Scanline

I. for each polygon do
   for each pixel (x,y) in the polygon’s projection do
      \[ z := -(D + A \cdot x + B \cdot y) / C; \]
      DrawZpixel(x, y, z, polygon’s color);

II. for each scan-line y do
    for each “in range” polygon projection do
       for each pair (x₁, x₂) of X-intersections do
          for \( x := x₁ \) to \( x₂ \) do
             \[ z := -(D + A \cdot x + B \cdot y) / C; \]
             DrawZpixel(x, y, z, polygon’s color);

If we know \( z_{x,y} \) at (x,y) than: \[ z_{x+1,y} = z_{x,y} - A / C \]
**Incremental Scanline**

\[ Ax + By + Cz + D = 0 \]

\[ z = \frac{-(Ax + By + D)}{C}, C \neq 0 \]

On a scan line \( Y = j \), a constant

Thus depth of pixel at \( (x_1 = x + \Delta x, j) \)

\[ z_1 - z = \frac{-(Ax_1 + Bj + D)}{C} + \frac{-(Ax + Bj + D)}{C} \]

\[ z_1 - z = \frac{A(x - x_1)}{C} \]

\[ z_1 = z - \left( \frac{A}{C} \right) \Delta x \] , since \( \Delta x = 1 \),

\[ z_1 = z - \frac{A}{C} \]
Incremental Scanline (contd.)

- All that was about increment for pixels on each scanline.
- How about across scanlines for a given pixel?
- Assumption: next scanline is within polygon

\[
\begin{align*}
  z_1 - z &= \left(-\frac{Ax + By_1 + D}{C}\right) + \left(\frac{Ax + By + D}{C}\right) \\
  z_1 - z &= \frac{A(y - y_1)}{C} \\
  z_1 &= z - \left(\frac{B}{C}\right)\Delta y, \text{ since } \Delta y = 1, \\
  z_1 &= z - \frac{B}{C}
\end{align*}
\]
Non-Planar Polygons

Bilinear Interpolation of Depth Values

\[
\begin{align*}
    z_a &= z_1 + (z_4 - z_1) \frac{(y_1 - y_s)}{(y_1 - y_4)} \\
    z_b &= z_1 + (z_2 - z_1) \frac{(y_1 - y_s)}{(y_1 - y_2)} \\
    z_p &= z_a + (z_b - z_a) \frac{(x_a - x_p)}{(x_a - x_b)}
\end{align*}
\]
Z-buffer - Example

Z-buffer

Screen
Non Trivial Example?

Figure 4-57 Penetrating triangle. (a) Three-dimensional view; (b) two-dimensional projection.

Rectangle: P1(10,5,10), P2(10,25,10), P3(25,25,10), P4(25,5,10)

Triangle: P5(15,15,15), P6(25,25,5), P7(30,10,5)

Frame Buffer: Background 0, Rectangle 1, Triangle 2

Z-buffer: 32x32x4 bit planes
Example
Z-Buffer Advantages

- Simple and easy to implement
- Amenable to scan-line algorithms
- Can easily resolve visibility cycles
Z-Buffer Disadvantages

- Does not do transparency easily

- Aliasing occurs! Since not all depth questions can be resolved

- Anti-aliasing solutions non-trivial

- Shadows are not easy

- Higher order illumination is hard in general
Scanline Rasterization

- Polygon scan-conversion:
- Intersect scanline with polygon edges and fill between pairs of intersections

For $y = y_{\text{min}}$ to $y_{\text{max}}$

1) intersect scanline $y$ with each edge
2) sort intersections by increasing $x$
3) fill pairwise ($p_0 > p_1$, $p_2 > p_3$, ....)
Scanline Rasterization Special Handling

• Make sure we only fill the interior pixels
  – Define interior: For a given pair of intersection points \((X_i, Y), (X_j, Y)\)
  – Fill ceiling\((X_i)\) to floor\((X_j)\)
  – Important when we have polygons adjacent to each other

• Intersection has an integer X coordinate
  – If \(X_i\) is integer, we define it to be interior
  – If \(X_j\) is integer, we define it to be exterior
  – (so don’t fill)
Scanline Rasterization Special Handling

- Intersection is an edge end point, say: \((p0, p1, p2)\) ??
- \((p0,p1,p1,p2)\), so we can still fill pairwise
- In fact, if we compute the intersection of the scanline with edge e1 and e2 separately, we will get the intersection point \(p1\) twice. Keep both of the \(p1\).
Scanline Rasterization Special Handling

- But what about this case: still \((p_0, p_1, p_1, p_2)\)
Rule

• Rule:
  – If the intersection is the y-min of the edge’s endpoint, count it. Otherwise, don’t.
• Don’t count p1 for e2
Performance Improvement

• The goal is to compute the intersections more efficiently. Brute force: intersect all the edges with each scanline
  – find the ymin and ymax of each edge and intersect the edge only when it crosses the scanline
  – only calculate the intersection of the edge with the first scan line it intersects
  – calculate dx/dy
  – for each additional scanline, calculate the new intersection as \( x = x + dx/dy \)
Data Structure

• Edge table:
  – all edges sorted by their ymin coordinates.
  – keep a separate bucket for each scanline
  – within each bucket, edges are sorted by increasing x of the ymin endpoint
Edge Table

- Edge structure: ymax, xmin, dx/dy, next

AB:

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>6</td>
<td>-5/2</td>
</tr>
</tbody>
</table>
Active Edge Table (AET)

- A list of edges active for current scanline, sorted in increasing $x$

$y = 9$

$y = 8$
Polygon Scan-conversion Algorithm

Construct the Edge Table (ET);
Active Edge Table (AET) = null;
for y = Ymin to Ymax
    Merge-sort ET[y] into AET by x value
    Fill between pairs of x in AET
    for each edge in AET
        if edge.ymax = y
            remove edge from AET
        else
            edge.x = edge.x + dx/dy
    sort AET by x value
end scan_fill