Hidden Surface Removal

Chapter 7

Hidden Surface Removal

- Major research topic in CG
- Multiple algorithms – cover a few
- Algorithm types
  - Object space
  - Image space

Hidden Surface Removal for Polygonal Scenes

- Input: Set of polygons in three-dimensional space + viewpoint
- Output: Two-dimensional image of projected polygons, containing only visible portions

Back Face Culling (object space)

- In closed polyhedron you don’t see object “back” faces
- Assumption
  - Normals of faces point out from the object

Back Face Culling

- Determine back & front faces using sign of inner product \( \mathbf{n} \cdot \mathbf{v} \)
  \[
  \mathbf{n} \cdot \mathbf{v} = n_x v_x + n_y v_y + n_z v_z = |\mathbf{n}| |\mathbf{v}| \cos \theta
  \]
- In a convex object:
  - Invisible back faces
  - All front faces entirely visible \( \Rightarrow \) solves hidden surfaces problem
- In non-convex object:
  - Invisible back faces
  - Front faces can be visible, invisible, or partially visible

Copyright Alla Sheffer
UBC 2004
Hidden Surface Removal

Depth Sort (object space)
- Question: Given a set of polygons, is it possible to:
  - sort them (by depth)
  - then paint them back to front (over each other) to remove the hidden surfaces?
- Answer: No
- Works for special cases
  - E.g. polygons with constant z

Depth Sort by Splitting
- Given two polygons, P and Q, can order in z if:
  1. P and Q do not overlap in their x extents
  2. Or P and Q do not overlap in their y extents
  3. Or P is totally on one side of Q’s plane
  4. Or Q is totally on one side of P’s plane
  5. Or P and Q do not intersect in projection plane
- If neither holds, split P along its intersection with Q into two smaller polygons
- How does this apply to examples on previous slide?

BSP Trees
- Different use of tests 3 & 4 in Depth Sort method
- Define:
  - $S_p$ – set of polygons
  - $P \in S_p$
  - $N_p$ normal to P
  - $P$ in plane $L_p$
- Subdivide into 3 groups:
  - Polygons in front of $L_p$ ($N_p$ direction)
  - Polygons behind $L_p$
  - Polygons intersecting $L_p$
- Split polygons in class 3 along $L_p$, place pieces in first 2 groups

BSP Trees
- After subdivision
  - Polygons behind $L_p$ can’t obscure $P$ ⇒ draw first
  - $P$ can’t obscure polygons in front of $L_p$ ⇒ draw $P$
  - Draw polygons in front of $L_p$
- Recursively subdivide and draw front & back sets
  - BSP – Binary Space Partition

BSP Trees
- Convention: Right sibling in $N_p$ direction
- BSP Tree is vie\textit{\textbf{w independent}}
- Constructed using only object geometry
- Can be used in hidden surface removal from multiple views
- How to choose what is visible for given view?
Z-Buffer Algorithm (image space)

- Idea: Instead of always painting over pixel while scan-converting a polygon, do that only if polygon's depth is less than current depth at that pixel
- In each pixel save color and current depth \( z \)
- New color will replace current only if closer in \( z \)

Z-Buffer - Project(P)

- Use regular perspective – loose depth
- Need to store separately
- Alternative: perspective warp

\[
\begin{pmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & d & 0 \\
0 & 0 & 0 & d^{-1}
\end{pmatrix}
\begin{pmatrix} x \\ y \\ z \\ 1 \end{pmatrix} = \begin{pmatrix} x' \\ y' \\ z' \\ 1 \end{pmatrix}
\]

\( z' \) monotonic in \( z \) – use as depth to set order

Z-Buffer – Depth(Q,x,y)

\[
z_1 = z_1 + (1 - \alpha_1)z_1
\]

\[
z_2 = \alpha_1z_1 + (1 - \alpha_1)z_1
\]

Questions: How to compute Project(P) & Depth(Q,x,y)?

Z-Buffer Algorithm Properties

- Image space algorithm
- Data structure: Array of depth values
- Common in hardware due to simplicity
- Depth resolution of 32 bits is common

Scene may be updated on the fly adding new polygons

Copyright Alla Sheffer
UBC 2004

The Rendering Pipeline
Transparency/Object Buffer

- A-buffer - extension to Z-buffer
- Save all pixel values
- At the end – have list of polygons & depths (order) for each pixel
- Simulate transparency by weighting different list elements

Scan-Line Z-Buffer Algorithm

- In software implementations - amount of memory required for screen Z-buffer is prohibitive
- Scan-line Z-buffer algorithm:
  - Render image one line at a time
  - Take into account only polygons affecting this line
- Combination of polygon scan-conversion & Z-buffer algorithms
  - Only Z-buffer the size of scan-line is required
  - **Scene must be available apriori**
  - **Image cannot be updated incrementally**

Scan-Line Z-Buffer Algorithm

```
ScanLineZBuffer(Scene)
Scene-2D := Project(Scene);
Sort Scene-2D into buckets of polygons P in increasing order of YMin(P);
A := EmptySet;
Y := YMin(Scene-2D) to YMax(Scene-2D) do
  For each pixel (x,y) in scanline Y = y do
    PutZ(x, MaxZ);
    A := A + {P in Scene : YMin(P) = y};
  end;
end;
A := A - {P in A : YMax(P) < y};
For each polygon P in A
  For each pixel (x,y) in P’s spans on the scanline
    z1 := Depth(P, x, y);
    if (z1 < GetZ(x)) then
      PutZ(x, z1);
      PutColor(x, y, Col(P));
    end;
  end;
end;
```