Visible Surface Determination

Definition
- Given a set of 3-D objects and a view specification (camera), determine which lines or surfaces of the object are visible
  - you’ve already seen a VSD step. computing smallest non-negative t value along a ray
  - why might objects not be visible? occlusion vs. clipping
  - clipping is one object at a time while occlusion is global

- Also called Hidden Surface Removal (HSR)

Object-Precision Algorithms

Historically first approaches
- Roberts ‘63 - hidden line removal
  - compare each edge with every object - eliminate invisible edges or parts of edges.
- Complexity: worse than $O(n^2)$ since each object must be compared with all edges
- A similar approach for hidden surfaces:
  - each polygon is clipped by the projections of all other polygons in front of it
  - invisible surfaces are eliminated and visible sub-polygons are created
  - SLOW, ugly special cases, polygons only

Painter’s Algorithm – Image Precision

Better way to resolve visibility exactly
- Create drawing order, each poly overwriting the previous ones, that guarantees correct visibility at any pixel resolution
- Strategy is to work back to front; find a way to sort polygons by depth (z), then draw them in that order
  - do a rough sort of polygons by smallest (farthest) z-coordinate in each polygon
  - scan-convert most distant polygon first, then work forward towards viewpoint (“painters’ algorithm”)
- We can either do a complete sort and then scan-convert, or we can paint as we go – see 3D depth-sort algorithm by Newell, Newell, and Sancha
- Any problems?

Hardware Scan Conversion: Visible Surface Determination (1/4)

- First apply perspective transformation on vertices.

  Canonical perspective-projection view volume with cube

- Perform backface culling
  - If normal is facing in same direction as LOS (line of sight), it’s a back face:
    - if $\overrightarrow{N_{obj}} \cdot \overrightarrow{LOS} < 0$, then polygon may be visible
  - If LOS $\overrightarrow{N_{obj}} \geq 0$, then polygon is invisible - discard
  - if $\overrightarrow{LOS} \cdot \overrightarrow{N_{obj}} < 0$, then polygon may be visible

- Next clip against normalized view volume ($-1 \leq x \leq 1$, $-1 \leq y \leq 1$, $0 \leq z \leq 1$)
So how do we compute this efficiently?

- Answer is simple: do it incrementally!
- Remember scan conversion/polygon filling? As we move along Y-axis, track x position where each edge intersects scan-line
- Do same thing for z coordinate using “remainder” calculations with v-z slope

\[
\begin{align*}
    z_2 &= z_1 - (t_1 - t_2) \frac{y_1 - y_2}{y_1 - y_2} \\
    z_3 &= z_1 - (t_1 - t_3) \frac{y_1 - y_3}{y_1 - y_2} \\
    z_0 &= z_1 - (t_1 - t_0) \frac{y_1 - y_0}{y_1 - y_2}
\end{align*}
\]

- Once we have \(z_2\) and \(z_3\) for each edge, can incrementally calculate \(z_0\) as we scan. Did something similar with calculating color per pixel... (Gouraud shading)

**Z-Buffer Algorithm (3/4)**

- Requires two “buffers”
  - Intensity Buffer
    - our familiar RGB pixel buffer
    - initialized to background color
  - Depth (“Z”) Buffer
    - depth of scene at each pixel
    - initialized to far depth = 255
- Polygons are scan-converted in arbitrary order. When pixels overlap, use Z-buffer to decide which polygon “gets” that pixel

**Z-Buffer Algorithm (4/4)**

**So how do we compute this efficiently?**

- Answer is simple: do it incrementally!
- Remember scan conversion/polygon filling? As we move along Y-axis, track x position where each edge intersects scan-line
- Do same thing for z coordinate using “remainder” calculations with v-z slope

\[
\begin{align*}
    z_2 &= z_1 - (t_1 - t_2) \frac{y_1 - y_2}{y_1 - y_2} \\
    z_3 &= z_1 - (t_1 - t_3) \frac{y_1 - y_3}{y_1 - y_2} \\
    z_0 &= z_1 - (t_1 - t_0) \frac{y_1 - y_0}{y_1 - y_2}
\end{align*}
\]

- Once we have \(z_2\) and \(z_3\) for each edge, can incrementally calculate \(z_0\) as we scan. Did something similar with calculating color per pixel... (Gouraud shading)