
Visible Surface Determination

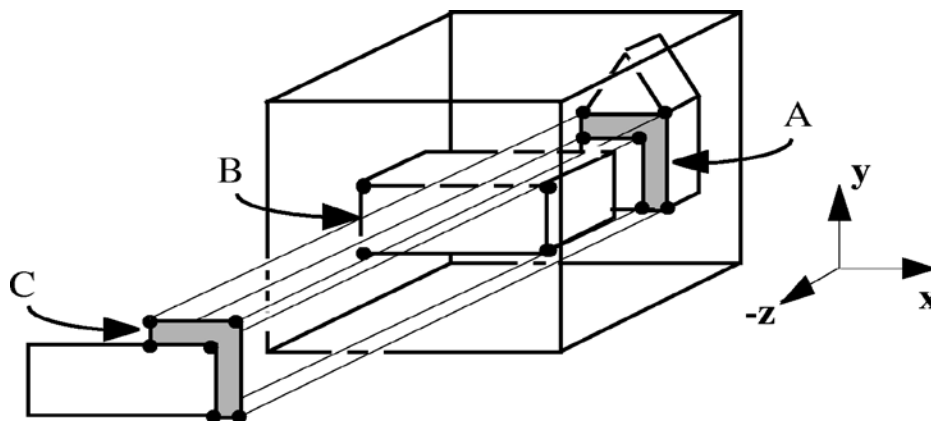
Definition

- 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



Polygon A is clipped by B which is in front of it. A new sub-polygon, C, is created.

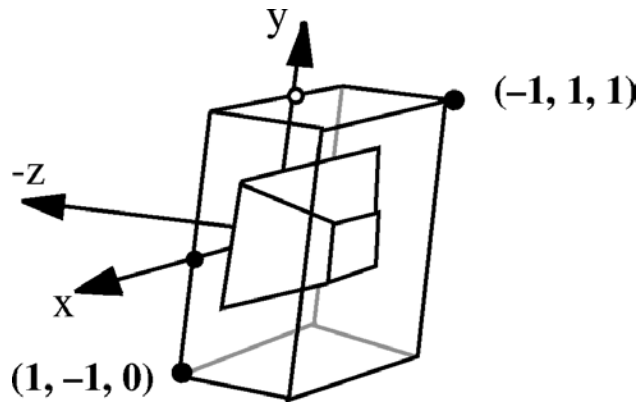
Painter's Algorithm – Image Precision

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?

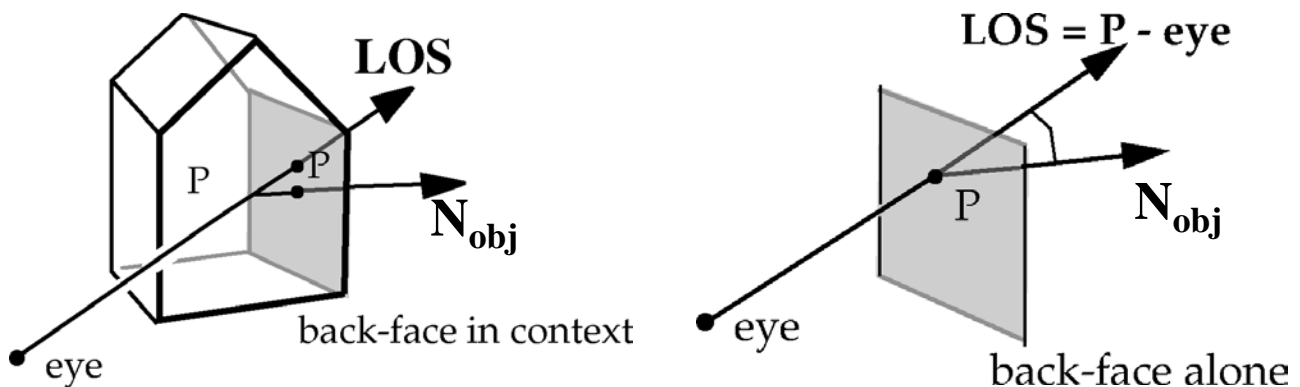
Hardware Scan Conversion: Visible Surface Determination (1/4)

- First apply perspective transformation on vertices.



Canonical perspective-projection view volume with cube

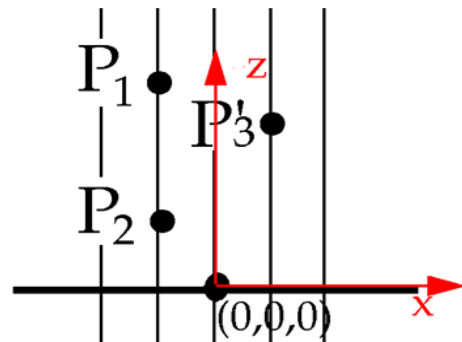
- Perform backface culling
 - If normal is facing in same direction as LOS (line of sight), it's a back face:
 - if $\text{LOS} \cdot \mathbf{N}_{\text{obj}} \geq 0$, then polygon is invisible – discard
 - if $\text{LOS} \cdot \mathbf{N}_{\text{obj}} < 0$, then polygon may be visible



- Next clip against normalized view volume
($-1 \leq x \leq 1$), ($-1 \leq y \leq 1$), ($0 \leq z \leq 1$)

Hardware Scan Conversion: Visible Surface Determination (2/4)

- Still need to determine object occlusion



P2 should occlude P1

How do we determine which point is closer?

The *Z-buffer* algorithm

- Z-buffer is initialized to background value (furthest plane of view volume = 1.0)
- As each object is traversed, z-values of all its sample points are compared to z-value in same (x, y) location in Z-buffer
- If new point has z value less than previous one (i.e., closer to eye), its z-value is placed in z-buffer and its color placed in frame buffer at same (x, y) ; otherwise previous z-value and frame buffer color are unchanged
- Can store depth as integers or floats or fixed points
 - i.e. for 8-bit (1 byte) integer z-buffer, set 0.0 \rightarrow 0 and 1.0 \rightarrow 255
 - Far plane and precision of z-buffer can have dramatic effect on rendered image

Z-Buffer Algorithm (3/4)

- Requires two “buffers”

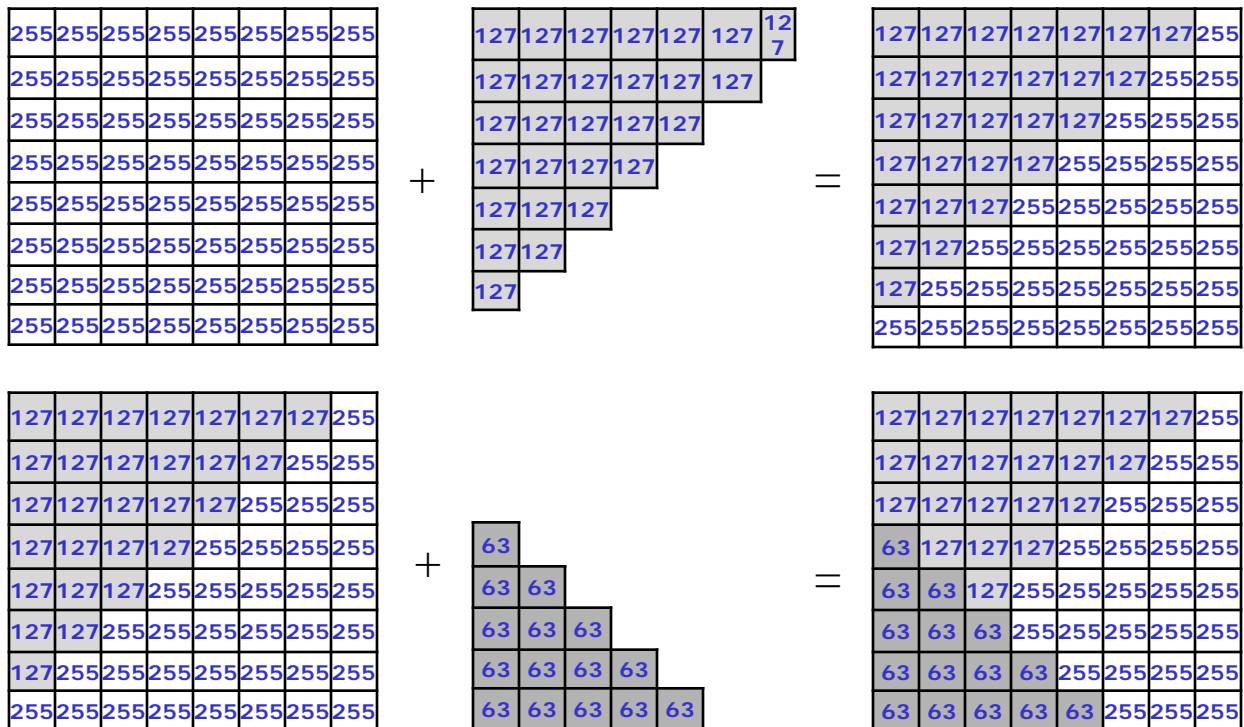
Intensity Buffer

- 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

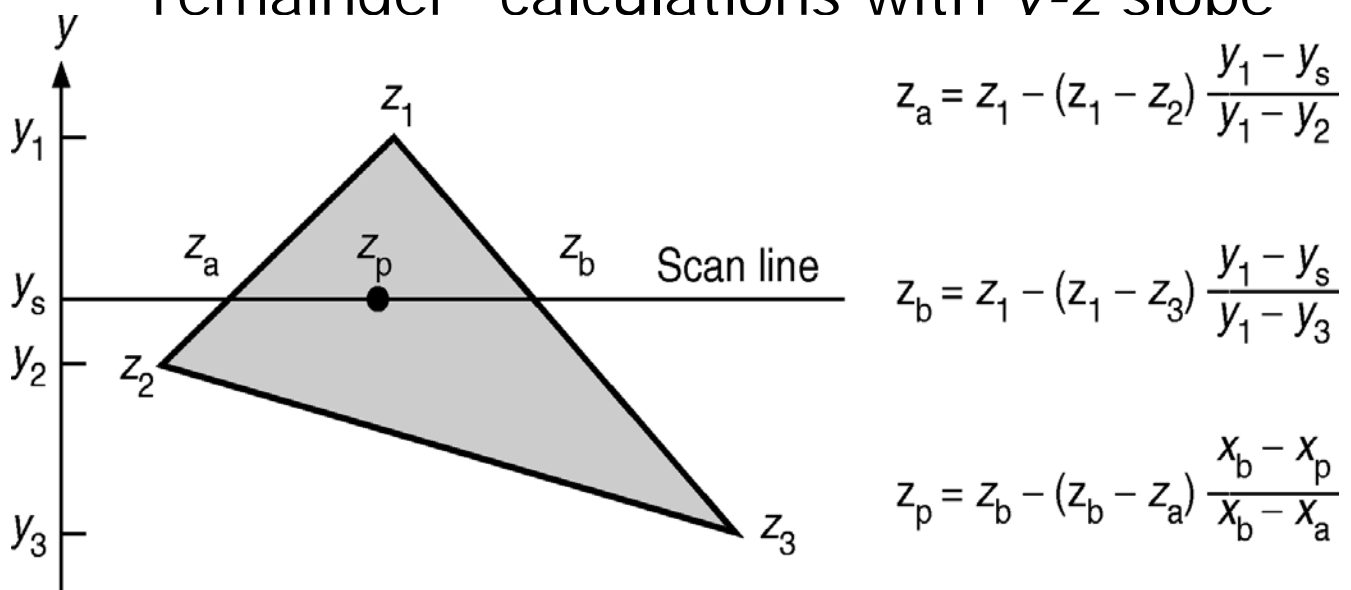


Above: example using integer Z-buffer with near = 0, far = 255

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



- Once we have z_a and z_b for each edge, can incrementally calculate z_p as we scan. Did something similar with calculating color per pixel... (Gouraud shading)