Hidden Surface Removal

Hidden Surface Removal for Polygonal Scenes
- **Input**: Set of polygons in three-dimensional space + a viewpoint
- **Output**: A two-dimensional image of projected polygons, containing only visible portions

The Normal Vector

\[ n = (v_2 - v_1) \times (v_3 - v_1) \]

\[ n(1,2,3) = n(2,3,1) = -n(2,1,3) \]

Barycentric Coordinates

\[ (x, y) = v_{\alpha_i} \]

\[ x = \sum_{i=1}^{3} \alpha_i (x_i, y_i) \]

\[ \sum_{i=1}^{3} \alpha_i = 1 \]

Barycentric coordinates of \( v = (\alpha_1, \alpha_2, \alpha_3) \)
- B.C. are unique.
- B.C. of all interior points are \( \geq 0 \).
- Triangle centroid = \((1/3,1/3,1/3)\).

Linear Interpolation

\[ f(x_1, y_1) \]

\[ f(x_2, y_2) \]

\[ f(x_3, y_3) \]

\[ f(x_i, y_i) = \sum_{i=1}^{3} \alpha_i f(x_i, y_i) \]

\[ \sum_{i=1}^{3} \alpha_i = 1 \]

Back Face Culling (object space)
- In closed polyhedron you don’t see object “back” faces
- Assumption
  - Normals of faces point out from the object
- Object space algorithm

Copyright  Gotsman, Elber, Barequet, Karni, Sheffer
Computer Science, Technion
Back Face Culling
- Determine back & front faces using sign of inner product \( \langle n, v \rangle \):
  \[
  \langle n, v \rangle = n_i v_i + n_j v_j + n_k v_k = ||v|| \cos \theta
  \]
- In a convex object:
  - Invisible back faces
  - All front faces entirely visible \( \Rightarrow \) solves hidden surfaces problem
- In a non-convex object:
  - Invisible back faces
  - Front faces can be visible, invisible, or partially visible

Depth Sort (object space)
- Will fail for:
  - Intersecting polygons
  - Mutually occluding polygons

Absence of hidden surfaces

Plane containing \( P \)
- Since every polygon is planar, we can talk about the plane \( h \) of a polygon \( P \).
- \textbf{Observation:} If polygon \( Q \) does not intersect \( h \), then
  - If \( Q \) is in the side of \( h \) containing the viewpoint, then during the painter algorithm, we can draw \( P \) before drawing \( Q \)
  - Otherwise \( P \) can be drawn before \( Q \)

Depth Sort by Splitting
- Given two polygons, \( P \) and \( Q \), we can order them in \( \mathcal{Z} \) 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
- Can we always resolve the relation between \( P \) and \( Q \) using steps 1-5?

Depth Sort by Splitting
- What steps 1-5 all fail?
- Split \( P(Q) \) along:
  - the intersection with \( Q \) (resp \( P \)) into two smaller polygons – (how could one compute this intersection?)
  - the intersection of \( P \) (\( Q \)) with the plane containing \( \mathcal{Q} \)

Object space algorithm
Constructing tree:
- choose polygon (arbitrary)
- split its cell using plane on which polygon lies
- continue until each cell contains only one polygon

Advantages:
- recursive descent
- render back, node polygon, front
- back/front is determined by what side of the plane the camera is on
- data structure is worth knowing about

Disadvantages:
- many small pieces of polygon (more splits than depth sort)
- over rendering (does not work well for complex scenes with lots of depth overlap)
- expensive to get approximately optimal tree, but for many applications this can be "off-line" in a pre-processing step.

2D version for illustration
Z-Buffer Algorithm (image space)

- Basic Idea: resolve the visibility at the pixel level, using depth sort.
- For each image pixel - store both the color and the current $z$ depth
- Instead of always painting the pixels while scan-converting a polygon, do so only if polygon’s depth is less than current $z$ depth at that pixel

Z-Buffer

For every pixel $(x,y)$ do

\[
\text{PutZ}(x,y,\text{MaxZ});
\]

For each polygon $P$ in Scene do

\[
\text{Q} := \text{Project}(P);
\]

For each pixel $(x,y)$ in $Q$ do

\[
\text{z} := \text{Depth}(Q,x,y);
\]

if $(z < \text{GetZ}(x,y))$ then

\[
\text{PutZ}(x,y,z);
\]

\[
\text{PutColor}(x,y,\text{Col}(P));
\]

end;

end;

end;
In most cases, polygons are given by specifying their vertices. For the Z-buffer, we need to find the depth of two triangles in the same pixel. Linear interpolation will do.

\[
\text{Depth}(Q, x, y) = \alpha_1 z_1 + (1 - \alpha_1) z_3
\]
**Z-Buffer Algorithm**
- 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

**The Graphics Pipeline**
- Hardware implementation of screen Z-buffer:
  - Polygons sent through pipeline one at a time
  - Display updated to reflect each new polygon

**Transparency Z-Buffer**
How can we emulate transparent objects?

**Transparency Buffer**
- Extension to the basic Z-buffer algorithm
- Save all pixel values
- At the end – have list of polygons & depths (order) for each pixel
- Simulate transparency by weighting the different list elements, in order

**The A - buffer**
- For transparent surfaces and filter based anti-aliasing:
  - Algorithm (1): filling buffer
    - at each pixel, maintain a pointer to a list of polygons sorted by depth.
    - when filling a pixel:
      - If polygon is opaque and covers pixel, insert into list, removing all polygons farther away
      - If polygon is opaque and only partially covers pixel, insert into list, but don’t remove farther polygons
  - Algorithm (2): rendering pixels
    - at each pixel, traverse buffer using brightness values in polygons to fill.
    - values are used for either for calculations involving transparency or for filtering for aliasing

---

Copyright Gotsman, Elber, Barequet, Karni, Sheffer
Computer Science, Technion
Scan-Line Z-Buffer Algorithm

- In software implementations - amount of memory required for screen Z-buffer may be prohibitive
- Scan-line Z-buffer algorithm:
  - Render the 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.
- Entire scene must be available a-priori
- Image cannot be updated incrementally

Scan-Line Z-Buffer Algorithm

ScanLineZBuffer(Scene)
Scene2D := Project(Scene);
Sort Scene2D into buckets of polygons P in increasing YMin(P) order;
A := EmptySet;
for y := YMin(Scene2D) to YMax(Scene2D) do
  for each pixel (x, y) in scanline Y=y do
    PutZ(x, MaxZ);
    A := A + {P in Scene : YMin(P)<=y};
  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
      z := Depth(P, x, y);
      if (z<GetZ(x)) then
        PutZ(x, z);
        PutColor(x, y, Col(P));
      end;
  end;
end;