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

$$n = (v_3 - v_1) \times (v_2 - v_1)$$

$$n(1,2,3) = n(2,3,1) = -n(2,1,3)$$

Barycentric Coordinates

$$\alpha = \frac{(x_1, y_1)}{\sum_{i=1}^{3} \alpha_i}$$

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

$$f(x_1, y_1) = \sum_{i=1}^{3} \alpha_i f(x_i, y_i)$$

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
Determine back & front faces using sign of inner product \(< n, V \>\):
\( n \cdot V = n_1 V_1 + n_2 V_2 + n_3 V_3 = ||n|| \cos \theta \)

In a convex object:
- Invisible back faces
- All front faces entirely visible \(\implies\) solves hidden surfaces problem

In 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

Plane containing \(P\)
- Since every polygon is planar, we can speak about the plane \(h\) of a polygon \(P\).
- Observation: if polygon \(Q\) does not intersect \(h\), then
  - If \(Q\) and the viewer lie on the same side of \(h\), then during the painter algorithm, we can render \(Q\) before rendering \(P\)
  - Otherwise \(P\) can be rendered before \(Q\)

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

**Painter’s algorithm revised**
- Given polygons $P_1, \ldots, P_n$ in 3D,
- For each $P_i$, find $h$ the plane containing $P_i$,
  - For each other (sub)polygons $Q_j$ intersecting $h$,
    - Split $Q_j$ into two smaller polygons, one on each side of $h$.

**Looks like lots of unnecessary chopping…**

**BSP - trees**
- Construct a tree that gives a rendering order
- Tree recursively splits 3D world into cells, each of which contain at most one piece of polygon.
- Constructing tree:
  - Choose polygon (arbitrary)
  - Split its cell using plane on which polygon lies
  - Continue until each cell contains only one polygon

**The Visibility Problem**

**Binary Planar Partitions**
- Draw everything below the line 0 (further from the eye) before drawing the ones closer to the eye. Continue recursively inside each subtree.
  - This structure, as any other hierarchical decomposition, is useful for range searching, point location etc.

**Painter’s Algorithm**

**Binary Planar Partitions**

---

Copyright  Gotsman, Elber, Barequet, Karni, Sheffer
Computer Science, Technion
**Hidden Surface Removal**

**Auto-partitions**

All cuts are through segments. The order has an impact. Here we first cut through a, then through c. Note: 4 nodes.

Different order: First cut through a, then through b. Note: 5 nodes.

**What is the complexity of BSP using auto-partitions?**

Worst case: 1 + 2 + 3 + ... + n = n(n+1)/2 = $\Omega(n^2)$

**BSP - trees**

2D version for illustration

Copyright  Gotsman, Elber, Barequet, Karni, Sheffer
Computer Science, Technion
**Computer Graphics**

**Hidden Surface Removal**

**BSP - trees**

- Rendering tree:
  - recursive descent
  - back/front polygon, front
- back/front is determined by what side of the plane the camera is on
- 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)
- Advantages:
  - one tree works for all focal points (good for cases when scene is static)
  - filter anti-aliasing works fine, as does transparency
  - data structure is worth knowing about
- Comment
  - expensive to get approximately optimal tree, but for many applications this can be "off-line" in a pre-processing step.

**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 scanning a polygon, do so only if polygon’s depth is less than current z depth at that pixel

**Z-Buffer**

```
ZBuffer(Scene)
For every pixel (x,y) do PutZ(x,y,MaxZ);
For each polygon P in Scene do
Q := Project(P);
For each pixel (x,y) in Q do
z := Depth(Q,x,y);
if (z < GetZ(x,y)) then
   PutZ(x,y,z);
   PutColor(x,y,Col(P));
end;
end;
end;
```

*Questions: How can one compute Project(P) and Depth(Q,x,y)?*

---

Copyright  Gotsman, Elber, Barequet, Karni, Sheffer
Computer Science, Technion
Computer Graphics

Hidden Surface Removal

Copyright Gotsman, Elber, Barequet, Karni, Sheffer
Computer Science, Technion
Z-Buffer – Depth(Q,x,y)

- 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.

\[
\begin{align*}
z_4 &= \alpha_0 z_1 + (1-\alpha_0) z_2 \\
\alpha_0 z_3 + (1-\alpha_0) z_4
\end{align*}
\]

% scanline Y=y

Depth(Q,x,y) = \alpha_0 z_1 + (1-\alpha_0) z_2

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

The A - buffer
- 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

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
- The list A contains the polygons intersecting the horizontal sweeping line, in their order of appearance along the line changes only at vertices.