Hidden Surface Removal
a.k.a Visible Surface Determination
Reminder - Pipeline

3D Model  

Transformations

Polygon at [(2,9), (5,7), (8,9)]
Reminder - Pipeline

Object coordinates → Camera coordinates → Clip coordinates → Normalized device coordinates → Screen coordinates

\[
\begin{pmatrix}
  x \\
  y \\
  z \\
  1
\end{pmatrix} \xrightarrow{\text{Model-view}} \begin{pmatrix}
  x_p \\
  y_p \\
  z_p
\end{pmatrix} \xrightarrow{\text{Projection}} \begin{pmatrix}
  x_s \\
  y_s
\end{pmatrix}
\]
Hidden Surface Removal

- **Input**: Polygons in normalized device coordinates

- **Output**: 2D image of projected polygons containing only visible portions
Hidden Surface Removal Algorithms

- Back face culling
- Painter’s algorithm (depth sort)
- Z-Buffer
- Scan-line Z-Buffer
Back Face Culling
Object Space

In a closed polyhedron back faces are not visible
Back Face Culling
Determining back faces

- In a closed polyhedron back faces are not visible
- Assume normals point *out*

face visible if: \[ \frac{\pi}{2} \leq \theta \leq \frac{\pi}{2} \]

\[ \cos\theta \geq 0 \]

\[ v \cdot n = |v||n|\cos\theta \geq 0 \]
Back Face culling

When will it work?

Closed, convex  Open  Closed, non convex
Back Face culling

• Closed convex objects
  – Invisible back faces
  – All front faces visible
  → *Visibility problem solved*

• Open objects
  – Back faces possibly visible

• Closed non-convex objects
  – Invisible back faces
  – Front faces can be visible, invisible or partially visible
Painter’s Algorithm
Object space

• Which **color** to draw?
• Draw everything → which **order** to draw?
Painter’s Algorithm

- Sort polygons by depth values
- Paint back to front
- When will this fail?

Intersecting polygons

Cyclic occlusion
Painter’s Algorithm

• Works in special cases
  
  – E.g. polygons with constant z

  – Where do we have polygons with constant z?
Painter’s Algorithm

How do we fix it?

• Depth sort per **polygon** doesn’t work

• Depth sort per **pixel**
  – Image space algorithm
Z-Buffer Algorithm
Image Space

 Resolve visibility at the pixel level
 Store color + current z per pixel
 Put new color only if depth < than current z
Z-Buffer Algorithm
The Z-Buffer
Z-Buffer Algorithm

The Algorithm

For every pixel \((x, y)\)

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

For each polygon \(P\)

\[
Q = \text{project}(P)
\]

for each pixel \((x, y)\) in \(Q\)

\[
z = \text{depth}(Q, x, y)
\]

if \(z < \text{getZ}(Q, x, y)\)

\[
\text{putColor}(x, y, \text{col}(P))
\]

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

end

end

end
Z-Buffer Algorithm

The Algorithm

For every pixel \((x,y)\)
\[ \text{put}Z(x,y,\text{MaxZ}) \]

For each polygon \(P\)
\[ Q = \text{project}(P) \]
for each pixel \((x,y)\) in \(Q\)
\[ z = \text{depth}(Q,x,y) \]
if \(z < \text{get}Z(Q,x,y)\)
\[ \text{putColor}(x,y,\text{col}(P)) \]
\[ \text{put}Z(x,y,z) \]
end
end
end
Z-Buffer Algorithm
The Algorithm

For every pixel \((x, y)\)

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

For each polygon \(P\)

\[
Q = \text{project}(P)
\]

for each pixel \((x, y)\) in \(Q\)

\[
z = \text{depth}(Q, x, y)
\]

if \(z < \text{getZ}(Q, x, y)\)

\[
\text{putColor}(x, y, \text{col}(P))
\]

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

end
end
end
Computing $\text{depth}(Q, x, y)$

- Have z coordinate at 3 vertices
- How do we compute z at pre-image of projected point?
Computing $\text{depth}(Q, x, y)$

- We know
  - 3D coordinates at vertices
  - 2D coordinates at vertices
  - 2D coordinates at $p_s$

- We need
  - 3D coordinates at $p$
Computing $\text{depth}(Q, x, y)$

- Linear transformations preserve straight lines

1. Compute $Z_i$

2. Compute $Z_p$
Linear Interpolation
On a line

- **Input:** 2 points, 2 values
- **Output:** value at any point \( p \) on the line \( L(t) \) between them

\[
L(t) = tp_1 + (1 - t)p_0
\]

\[
L(0) = p_0
\]
\[
L(1) = p_1
\]
\[
L(0.5) = \frac{p_0 + p_1}{2}
\]

\[
p_1, f_1, p_0, f_0, p \]
\[
t = \frac{||p - p_0||}{||p_1 - p_0||}
\]

\[
f(p) = f(t) = tf_1 + (1 - t)f_0
\]
Computing $\text{depth}(Q, x, y)$

- Linear transformations preserve straight lines

\[ z_i = t_i z_2 + (1 - t_i) z_1 \]

\[ t_i = \frac{|p_i - p_1|}{|p_2 - p_1|} \]

\[ z_p = t z_3 + (1 - t) z_i \]

\[ t = \frac{|p_s - p_3|}{|p_i - p_3|} \]
Linear Interpolation  
On a triangle

• Input: 3 points, 3 values

• Output: value at any point $p$ in the triangle they generate

\[ p = \alpha_1 p_1 + \alpha_2 p_2 + \alpha_3 p_3 \]

\[ \alpha_1 + \alpha_2 + \alpha_3 = 1 \]
\[ \alpha_i \geq 0 \]

\[ p_1, f_1, p_2, f_2, p_3, f_3, p \]

\[ \alpha_i = ? \]

\[ f(p) = f(\alpha_1, \alpha_2, \alpha_3) = \alpha_1 f_1 + \alpha_2 f_2 + \alpha_3 f_3 \]
Barycentric Coordinates

\[
\alpha_i = \frac{A_i}{A_1 + A_2 + A_3}
\]

Barycentric coordinates of \( p = (x, y, z) \) are unique.
B.C. of all interior points are \( \geq 0 \).
Triangle centroid = \((1/3,1/3,1/3)\).
\[ f(t) = tf(0) + (1-t)f(1) \]

\[ f_4 = t_1 f_1 + (1-t_1) f_2 \]

\[ f_5 = t_2 f_1 + (1-t_2) f_3 \]

\[ f(x, y) = t_3 f_4 + (1-t_3) f_5 \]

\[ = t_3 (t_1 f_1 + (1-t_1) f_2) + (1-t_3)(t_2 f_1 + (1-t_2) f_3) \]

\[ = \alpha_1 f_1 + \alpha_2 f_2 + \alpha_3 f_3 \]
Z-Buffer - Project(P)

\[
(x, y, z, 1) \begin{bmatrix}
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & \frac{d}{d-\alpha} & \frac{1}{d} \\
0 & 0 & -\alpha d & 0 \\
\end{bmatrix} = (x, y, \frac{(z-\alpha)d}{d-\alpha}, \frac{z}{d})
\]

\[
(x_p, y_p, z_p) = \left( \frac{x}{z/d}, \frac{y}{z/d}, \frac{d^2}{d-\alpha} \left( 1 - \frac{\alpha}{z} \right) \right)
\]

- \(z_p\) monotone with respect to \(z\) – use as depth to determine order
Z-Buffer Algorithm

• Image space algorithm

• Data structure: Array of depth values

• Implemented in hardware due to simplicity

• Depth resolution of 32 bits is common

• Scene may be updated on the fly, adding new polygons
Z Fighting

When Z-buffer has low precision and/or \( \alpha \) is not chosen correctly
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 Graphics Pipeline

• Hardware implementation of screen Z-buffer:
  – Polygons sent through pipeline one at a time
  – Display updated to reflect each new polygon
Scan-Line Z-Buffer Algorithm
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)

1. Scene2D := Project(Scene);
2. Sort Scene2D into buckets of polygons P in increasing YMin(P) order;
3. A := EmptySet;
4. for y := YMin(Scene2D) to YMax(Scene2D) do
   1. for each pixel (x, y) in scanline Y=y do PutZ(x, MaxZ);
   2. A := A + {P in Scene : YMin(P)<=y};
   3. A := A - {P in A : YMax(P)<y};
   4. for each polygon P in A
      1. for each pixel (x, y) in P's span(s) on the scanline
         1. z := Depth(P, x, y);
         2. if (z<GetZ(x)) then
            1. PutColor(x, y, Col(P));
            2. PutZ(x, z);
         end;
      end;
   end;
end;

---

A={ }  
A={a,d}  
A={c,d,b}  
A={b}  
A={ }