Hidden Surface Removal

- Goal: Determine which surfaces are visible and which are not.
- Other names:
  - Visible-surface detection
  - Hidden-surface elimination
- Display all visible surfaces, do not display any occluded surfaces.
- We can categorize into
  - Object-space methods
  - Image-space methods

Visible surface determination

- An opaque cube has at most 3 sides visible

Problems to solve:
- Which sides are visible from a given viewpoint?
- Which are hidden?
- How to make the rendering efficient?
Cases for hidden surface removal

- Occluded surfaces (partially or fully)
- Back faces

Utah Teapot - wireframe

Utah Teapot – surface rendering - outside
Hidden Surface Elimination

• Object space algorithms: determine which objects are in front of others
  – Resize doesn’t require recalculation
  – May be difficult to determine
• Image space algorithms: determine which object is visible at each pixel
  – Resize requires recalculation

Hidden Surface Elimination Complexity

• If a scene has \( n \) surfaces, then since every surface may have to be tested against every other surface for visibility, we might expect an object precision algorithm to take \( O(n^2) \) time.
• On the other hand, if there are \( N \) pixels, we might expect an image precision algorithm to take \( O(nN) \) time, since every pixel may have to be tested for the visibility of \( n \) surfaces.
• Since the number of the number of surfaces is much less than the number of pixels, then the number of decisions to be made is much fewer in the object precision case, \( n << N \).

• Different algorithms try to reduce these basic counts.
• Thus, one can consider bounding volumes (or “extents”) to determine roughly whether objects cannot overlap - this reduces the sorting time. With a good sorting algorithm, \( O(n^2) \) may be reducible to a more manageable \( O(n \log n) \).
• Concepts such as depth coherence (the depth of a point on a surface may be predictable from the depth known at a nearby point) can cut down the number of arithmetic steps to be performed.
• Image precision algorithms may benefit from hardware acceleration.
Two classes of algorithms

- Object-space (OS)
  - Operates on 3D object entities (vertices, edges, surfaces)
- Image space (IS)
  - Operates on 2D images (pixels)
- Operations normally applied to polygonal representations of objects (e.g. triangulated surfaces)

Hidden surface removal

Object-space approach
Image-space approach

Backface removal
Depth-sorting methods
Scan-line methods
Depth buffer methods

Back face removal (Polygon culling): OS

Principles

- Remove all surfaces pointing away from the viewer
- Eliminate the surface if it is completely obscured by other surfaces in front of it
- Render only the visible surfaces facing the viewer

How to determine which surfaces face away from the viewer?

- Compute vector \( \mathbf{N} \) normal to a surface patch (e.g. a triangle)
- In the right-handed coordinate system a surface facing away from the viewer (i.e. not visible) will have negative value of z-coordinate of the normal vector: \( N_z < 0 \)

Details were discussed in lecture “Object rendering”
How to determine which surfaces face away from the viewer?

- Compute vector $\mathbf{N}$ normal to the face (normally a triangle)
- Surface facing away from the viewer will have negative value of $z$-coordinate of the normal vector: $N_z < 0$
- IMPORTANT: In the left-handed coordinate system vertices must be traversed counter-clockwise

How to determine which surfaces face away from the viewer?

- Based on the equation of a plane
  
  $Ax + By + Cz + D = 0$

- Vector normal to the plane: $\mathbf{N} = [A, B, C]$

- $C > 0$: surface facing towards the viewer
- $C < 0$: surface facing away from the viewer

How to determine which surfaces face away from the viewer?

Derivation: based on the equation of the plane passing through three points:

$P_1 = (x_1, y_1, z_1) \quad P_2 = (x_2, y_2, z_2) \quad P_3 = (x_3, y_3, z_3)$

\[
\begin{vmatrix}
  x - x_1 & y_1 - y_1 & z - z_1 \\
  x_2 - x_1 & y_2 - y_1 & z_2 - z_1 \\
  x_3 - x_1 & y_3 - y_1 & z_3 - z_1
\end{vmatrix} = 0
\]

Equivalent to:

$Ax + By + Cz + D = 0$

where

$C = (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)$

Back face removal (Polygon culling)

- Algorithm
  - Compute 3D coordinates of the object in the camera coordinate system
  - For each surface patch compute the normal vector $\mathbf{N}$
  - If the $z$-coordinate of the normal vector $N_z < 0$ the surface patch does not need to be displayed

- Advantages / problems
  - Speeds up rendering
  - Works only for convex objects
Back face removal (Polygon culling)

Back-Face Culling Example

\[ \mathbf{n}_1 \cdot \mathbf{v} = (2, 1, 2) \cdot (-1, 0, -1) = \frac{2 - 2}{-2} = -4, \]
so \( \mathbf{n}_1 \cdot \mathbf{v} < 0 \)
so \( \mathbf{n}_1 \) front facing polygon

\[ \mathbf{n}_2 \cdot \mathbf{v} = (-3, 1, -2) \cdot (-1, 0, -1) = \frac{-3 + 2}{3} = 5 \]
so \( \mathbf{n}_2 \cdot \mathbf{v} > 0 \)
so \( \mathbf{n}_2 \) back facing polygon

Painter's algorithm (depth sorting method):

OS/IS

- Surfaces sorted in order of increasing depth in 3D
- Surfaces drawn starting with the surface of greatest depth (largest Z-coordinate)
- Surfaces with smaller depth are "painted over" surfaces that are already scan-converted.

Painter's algorithm

Example for a cube
**Painter’s Algorithm**

Object-space algorithm: Draw surfaces from back (farthest away) to front (closest):
- Sort surfaces/polygons by their depth (z value)
- Draw objects in order (farthest to closest)
- Closer objects paint over the top of farther away objects

---

**List Priority Algorithms**

- A visibility ordering is placed on the objects
  - Objects are rendered back to front based on that ordering
- Problems:
  - overlapping polygons

---

**Depth Sort Algorithm**

- An extension to the painter’s algorithm
  - Performs a similar algorithm but attempts to resolve overlapping polygons
- Algorithm:
  - Sort objects by their z value (farthest from the viewer)
  - Resolve any ambiguities caused by overlapping polygons, splitting polygons if necessary
  - Scan convert polygons in ascending order of their z values (back to front)

---

**Depth-Sort Algorithm**

- Depth-Sort test for overlapping polygons:
  - Let P be the most distant polygon in the sorted list.
  - Before scan converting P, we must make sure it does not overlap another polygon and obscure it
  - For each polygon Q that P might obscure, we make the following tests. As soon as one succeeds, there is no overlap, so we quit:
    1. Are there x extents non-overlapping?
    2. Are there y extents non-overlapping?
    3. Is P entirely on the other side of Q’s plane from the viewpoint?
    4. Is Q entirely on the same side of P’s plane as the viewpoint?
    5. Are their projections onto the (x, y) plane non-overlapping?
**Painter's algorithm**

**Steps of the algorithm**
- Order surfaces according to the z value
- Select surface with the greatest depth
- Select next greatest-depth surface
- If no depth overlap occurs - paint the surfaces

**Depth overlap**

**No depth overlap**

**Painter's algorithm**

- If depth overlap then carry out a number of tests to establish whether surface projections (X and Y) overlap

**Depth overlap**

**No X projection overlap**

**Depth-Sort Algorithm**

- Test 3 succeeds:
- Test 3 fails, test 4 succeeds:

Slide credit: www.cs.bath.ac.uk/~djp/30075-CG-12a-hiddenSurfaces.ppt
**Depth-Sort Algorithm**

- If all 5 tests fail, assume that P obscures Q, reverse their roles, and repeat steps 3 and 4.

- If these tests also fail, one of the polygons must be split into multiple polygons and the tests run again.

---

**Painter’s algorithm**

- **Advantages:**
  - Very good if a valid order is easy to establish; not so good for more complex surface topologies (e.g. presence of holes).
  - For simple cases very easy to implement.
  - Fits well with the rendering pipeline.

- **Problems:**
  - Not very efficient – all polygons are rendered, even when they become invisible.
  - Complex processing (sorting + tests) for complex objects.
  - There could be no solution for sorting order.
  - Possibility of an infinite loop as surface order is swapped.

---

**Z-buffer algorithm (IS)**

- Test visibility of surfaces **one point at a time**.
- The surface with the z-coordinate closest to VRP is visible (largest z in RH coordinate system; smallest z in LH coordinate system).

- Two storage areas required:
  - depth buffer - z value for each pixel at (x,y)
  - display buffer – pixel value (colour) for each pixel at (x,y).
fill z_buffer with "infinite" distance (in LH coordinate system)
for each polygon
compute 2D projections (rasterize)
for each pixel (x,y) inside the rasterized polygon
calculate z-value
if z(x,y) is closer than z_buffer(x,y)
set display_buffer(x,y)=polygon colour at (x,y)
set z_buffer(x,y)=polygon z(x,y)
end
end
end

Z-Buffering: Example

Scan convert the following two polygons.
The number inside the pixel represents its z-value.

Does order matter?

Slide credit: www.cs.bath.ac.uk/~djp/30075-CG-12a-hiddenSurfaces.ppt
Z-buffer algorithm – computing depth

- Depth values can be evaluated iteratively using the plane equation \((Ax + By + Cz + D = 0)\)
  - for x increments, along each scan line 
    \((x+1, y)\):
    \[z' = z - \frac{A}{C}\]
  - for y increments, for each new scan line 
    \((x, y+1)\):
    \[z'' = z - \frac{(A/m + B)}{C}\]

Compute A, B and C from the plane equation

Construct the equation of the plane passing through three points:

\[P_1=(x_1, y_1, z_1), \quad P_2= (x_2, y_2, z_2), \quad P_3=(x_3, y_3, z_3)\]

\[
\begin{align*}
(x - x_1) & \quad (y - y_1) & \quad (z - z_1) \\
x_2 - x_1 & \quad y_2 - y_1 & \quad z_2 - z_1 \\
x_3 - x_1 & \quad y_3 - y_1 & \quad z_3 - z_1
\end{align*}
\]

Equivalent to: \(Ax + By + Cz + D = 0\)

**e.g.**
\[C = (x_2 - x_1)(y_3 - y_1) - (x_3 - x_1)(y_2 - y_1)\]

Z-buffer algorithm

- **Advantages**
  - Easy to implement
  - Fits well with the rendering pipeline
  - Can be implemented in hardware
  - Always correct results

- **Problems**
  - Some inefficiency as pixels in polygons nearer the viewer will be drawn over polygons at greater depth

- It is a standard in many graphics packages (e.g. Open GL)

Other methods

Scan-line method (IS)

- Each scan line is processed
- list of edges (of ALL polygons) crossing a current line sorted in order of increasing x
- on/off flag for each surface
- depth sorting is necessary if multiple flags are on
- uses coherence along the scan lines to prevent unnecessary sorting
Other methods

Scan-line method (IS)

Other methods

Scan-line method (IS)

Area subdivision method

• Locate view areas (normally squares or rectangles) which represent a part of a single surface
• This is done by successively dividing the total view area into smaller rectangles
• Stop dividing a given rectangle when it contains a single surface or visibility precedence can be easily determined
Warnock’s Algorithm

- An area-subdivision technique
- Idea:
  - Divide an area into four equal sub-areas
  - At each stage, the projection of each polygon will do one of four things:
    1. Completely surround a particular area
    2. Intersect the area
    3. Be completely contained in the area
    4. Be disjoint to the area

Initial scene

- Disjoint polygons do not influence an area.
- Parts of an intersecting polygon that lie outside the area do not influence that area
- At each step, we determine the areas we can color and color them, then subdivide the areas that are ambiguous.

- At each stage of the algorithm, examine the areas:
  1. If no polygons lie within an area, the area is filled with the background color
  2. If only one polygon is in part of the area, the area is first filled with the background color and then the polygon is scan converted within the area.
  3. If one polygon surrounds the area and it is in front of any other polygons, the entire area is filled with the color of the surrounding polygon.
  4. Otherwise, subdivide the area and repeat the above 4 tests.
Warnock’s Algorithm

First subdivision

Slide credit: www.cs.bath.ac.uk/~djp/30075-CG-12a/hiddenSurfaces.ppt

 Warnock’s Algorithm

Second subdivision

Slide credit: www.cs.bath.ac.uk/~djp/30075-CG-12a/hiddenSurfaces.ppt

Warnock’s Algorithm

Third subdivision

Slide credit: www.cs.bath.ac.uk/~djp/30075-CG-12a/hiddenSurfaces.ppt

Warnock’s Algorithm

Fourth subdivision

Slide credit: www.cs.bath.ac.uk/~djp/30075-CG-12a/hiddenSurfaces.ppt
Warnock’s Algorithm

- Subdivision continues until:
  - All areas meet one of the four criteria
  - An area is pixel size
    - in this case, the polygon with the closest point at that pixel determines the pixel color

Recommendations for hidden surface methods

<table>
<thead>
<tr>
<th>Surfaces condition</th>
<th>Method</th>
</tr>
</thead>
<tbody>
<tr>
<td>Surfaces are distributed in z</td>
<td>Depth sorting</td>
</tr>
<tr>
<td>Surfaces are well separated in y</td>
<td>Scan-line or area-subdivision</td>
</tr>
<tr>
<td>Only a few surfaces present</td>
<td>Depth sorting or scan-line</td>
</tr>
<tr>
<td>Scene with at least a few thousand surfaces</td>
<td>Depth-buffer method or area-subdivision</td>
</tr>
</tbody>
</table>

Next lecture

Texture mapping