**Hidden Surface Removal**

Simone Gasparini  
<gasparini@elet.polimi.it>

---

**Outline**

- Introduction
- Hidden Surface Removal
- Back face culling
- Depth sort
- Z-buffer

---

**Introduction – Graphics pipeline**

- Clipping (field of view)
- Projection (3D → 2D)
- Hidden surface removal (only visible surfaces)
- Shading & Shadowing (according to light source(s))

---

**Introduction**

- Given a set of 3D objects and a viewing point determine which lines or surfaces of the objects are visible
- Main issues:
  - Back faces: invisible surfaces of the object
  - Hidden surfaces: (partially) invisible surfaces occluded by other surfaces of the object itself or by other objects
- It depends on the orientation of the object and on the mutual displacement of the objects w.r.t. the viewpoint
- Remember: time matters!
  - H.S.R. allows to avoid overdrawning
  - It has to be fast enough
Introduction

Preliminary notion on synthetic camera

- Viewing volume limited by:
  - Front clipping plane
    - Avoid to render object too close or behind the camera
  - Back clipping plane
    - Avoid to render objects too far and save rendering time

Objects outside the viewing volume is discarded
Objects intersecting viewing volume is clipped
Hidden surface removal applied only to:
- Objects completely inside the volume
- Clipped objects inside the volume

There are techniques for:
- Determining the visible surfaces
  - Visible surface determination
- Determining (and remove) hidden surfaces
  - Hidden surfaces removal

Two main approaches:
- Object space algorithms
  - Compare all scene objects in order to eliminate entire objects or portions of them
- Image space algorithms
  - For each pixel determine which object is visible

Image space

for each pixel in the image do
begin
  determine the object closest to the viewer that is pierced by the projector through the pixel
  Draw the pixel in the appropriate color
end
end for

For each image pixel consider the associated viewing ray
Intersect the viewing ray with each surface
Visible surface at pixel $p_i$ is the closest one (having closest intersection point)
Computational effort $O(p^*n)$ with $p \sim 10^6$
**Object space**

```plaintext
for each object in the world do
    begin
        determine those parts of the object whose view is
        unobstructed by other parts of it or any other object
        Draw those parts in the appropriate color
    end
end for

- Compare each of the n objects to itself and to the other objects
- Computational effort is $O(n^2)$
- Preferable if there are few objects
- The individual steps are complex and time consuming wrt image space
```

**Algorithms**

- Back face culling
- Depth sort
- Z-buffer

**Back face culling**

- Consider local properties of the objects
- Works independently of their mutual position in scene
- Consider objects approximated by a solid polyhedron
- The polygonal faces enclose its volume
- (Potentially) Visible faces are only those pointing towards the viewpoint
- Back faces are those pointing away from the observer
- Back face culling determines all non visible surfaces and
  - **All visible** surfaces in convex polyhedrons
  - **Potentially visible** surfaces in concave polyhedrons (occluded surfaces are not detected)
Back face culling

- A polygonal face is visible if the normal and the viewing ray form an angle less than 90°.

\[ \theta < 90^\circ \]

- It requires to check if the normal is pointing towards the viewpoint.

Let \( \theta \) the angle formed by the normal and the viewpoint.

- The polygon is a back face iff \(-90^\circ \leq \theta \leq 90^\circ\)
  \[ \cos \theta \geq 0 \]

- Check the dot product:
  - \( n \cdot v < 0 \) visible
  - \( n \cdot v = 0 \) "viewed on edge"
  - \( n \cdot v > 0 \) back face

- Just check the z-coordinate of the normal; if it is negative the face is visible.

Back face culling halves the number of polygon to be considered for each pixel in an image-space algorithm.

On average, about one-half of a polyhedron’s polygons are back facing.

Back face culling halves the number of polygon to be considered for each pixel in an object-space algorithm.

Computational effort \( O(n) \)

It can be speeded up by preprocessing (sublinear performances).

- For objects with small number of faces a table can be made up in order to determine visible surface according to viewpoint
  - E.g. Cube, for each octant there are at most three visible surface
- More complex objects require other tricks (graph-based representation), useful especially for dynamic scene

Open issues

- Back face culling only remove non visible surface
- If the polyhedron is concave it does not detect self-occlusion
- It does not detect mutual occlusion with other objects
**List priority algorithm**

- Determine a visibility ordering for objects
- Rendering objects in that order assures a correct picture

**Depth sort algorithm**
- Also known as “painter’s algorithm”
- Uses the frame buffer device
- Basic idea:
  - Paint the polygons into the frame buffer in order of decreasing distance from the viewpoint, possibly overwriting previously painted objects

**Depth sort algorithm**
- Given the set of polygons
- Order the polygon according to the decreasing distance from the viewpoint
- Paint:
  - First the farthest polygon
  - Then the closest polygons over the non visible part of farther objects

**Main Issues**
- Sorting polygons according to distance
  - Painter’s algorithm is easy because it draws objects lying on different z-plane
  - But if the z-coordinate of the objects overlap?
- Dealing with overlapping and intersecting objects

**Depth sort algorithm**
- Sort polygons according to distance of z-coordinate from the viewpoint
- Consider the extent of the polygon along z-direction
- The object is described by the pair of \((z_{\text{min}}, z_{\text{max}})\)
Depth sort algorithm

Easy case:
- Object A
- Compared to other objects $Z_{A_{\text{min}}} > Z_{A_{\text{max}}}$
- A is the first object to paint

Other cases:
- Which first?
- The depth extent is not a sufficient criterion
- Determine a depth order which grants correct picture
- Consider also $x$ and $y$

DEPTH-SORT ALGORITHM (Newell, Newell & Sancha, 1972)

3 main steps:
1. Sort all polygons according to the farthest z-coordinate (smallest z) of each
2. Resolve any ambiguities this may cause when the polygon’s z-extent overlap, possibly splitting polygons
3. Scan convert (rasterize) polygons in ascending order of smallest z coordinate (back to front)

Remark: the reference frame from now on is supposed to be placed on the back clipping plane with the z-axis pointing to the viewpoint. Hence the object closest to viewpoint has biggest z-coordinate and vice versa. The back clipping plane

Consider also $x$ and $y$
- Is it enough?
- Ambiguities for mutual intersecting/overlapping objects

Step 2: resolve any ambiguities
- Let P the polygon currently at the end of the sorted list
- Before scan converting P, test it against each polygon Q whose z-extent overlaps the extent Q, in order to prove that P cannot obscure Q and that P can be scan converted before Q.

Five tests to resolve ambiguities:
1. P and Q overlap in $x$?
2. P and Q overlap in $y$?
3. Is P entirely on the opposite side of Q’s plane from viewpoint?
4. Is Q entirely on the same side of P’s plane as the viewpoint?
5. Do the projections of the polygons onto the xy-plane not overlap?
### Depth sort algorithm

- **Test 3:** Is P entirely on the opposite side of Q’s plane from viewpoint?
  - [Diagram showing test result: OK!]

- **Test 4:** Is Q entirely on the same side of P’s plane as the viewpoint?
  - [Diagram showing test result: NO]

- If all tests fail, assume that P obscure Q and check whether Q can be scan converted before P.
- If all tests fail, one of the polygon as to be splitted by the plane of the other.
- This is necessary in case of cyclically overlapping polygons and intersecting polygons.

### Advantages
- Fast enough for simple scenes
- Fairly intuitive
- Great for transparent objects

### Disadvantages
- Slow for even moderately complex scenes
- Hard to implement and debug
- Lots of special cases

### Z-buffer

- Image-precision algorithm
  - Depends on image resolution
- Simplest visible-surface algorithm to implement in SW or HW
- Require a second buffer in addition to the frame buffer
- **Z-buffer** is a buffer of the same size of frame buffer
- Each entries of the buffer store the z-value for each (corresponding) pixel of frame buffer
  - Scan conversion computes the color value for each pixel of the frame buffer
  - Z-buffer store the relevant z-distance so that only closest objects are actually painted on the frame buffer.
- Z-buffer is initialized to background value (furthest plane of view volume = 1.0)
- Polygons are scan converted in arbitrary order
- For each object, z-values of all its sample points are compared to z-value in same (x, y) location in Z-buffer
  - Z could be determined by intersecting the ray with the plane containing the polygon.
  - In reality, calculate z at vertices and interpolate rest
  - If new point has z value greater 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

```c
void zBuffer()
{
    int x, y;
    for (y = 0; y < YMAX; y++)
        for (x = 0; x < XMAX; x++)
        {
            write_pixel(x, y, BACKGROUND_VALUE);
            write_z(x, y, 1);
        }
    for each polygon
        for each pixel in polygon’s projection
        {
            double pz = polygon’s Z-value at pixel (x, y);
            if (pz < read_z(x, y))
            {
                /* New point is closer to front of view */
                write_pixel(x, y, polygon’s color at pixel (x, y));
                write_z(x, y, pz);
            }
        }
}
```

Advantage
- Simplicity lends itself well to hardware implementations: FAST
  - used by all graphics cards
- Polygons do not have to be compared in any particular order: no presorting in z necessary
- Only consider one polygon at a time
  - ...even though occlusion is global problem!
  - brute force, but it is fast
- Can be used for non-polygonal surfaces, Constructive Solid Geometry (intersect, union, difference), and any \( z = f(x, y) \)
Z-buffer

Disadvantage
- Needs large memory to keep Z values
- It could not cover transparency