Hidden Surface Removal

- *Hidden surface removal* (HSR) determines which polygons are nearest to the viewer at a given pixel.

- Key criterion: a point $P$ *occludes* a point $Q$ (and thus $Q$ is “hidden”) if $P$ and $Q$ lie on the same ray (line) from the camera or eye and $P$ is between the camera location and $Q$.

- Calculating this ray is tough with a frustum, but normalizing that frustum to a cube (which the projection matrix does) transforms the oblique rays to straightforward parallelism with the $z$ axis.

- Thus, at the earliest, HSR happens after the projection matrix is applied — explaining the separation between the projection and viewport transformations!
Backface Culling

- An excellent preprocessing step to speed up hidden surface removal is \textit{backface culling} or \textit{backface removal}
- Backface culling checks the normal vector of every surface that we are rendering and throws away any surface whose vector points away from us (the viewer)
- This is easier (and faster) than it may seem (see below)
- This killer combination of speed and reduction is what makes backface culling work so well as an initial pass at hidden surface removal

Here’s the algorithm:

- Note how a polygon is front-facing if the angle $\theta$ between its normal and the vector toward the camera/viewer is between $-90$ and $90$ degrees. In other words, this means:

  $$\cos \theta \geq 0$$

- Recall from our study of vectors that $\cos \theta = \frac{e \cdot n}{||e|| \ ||n||}$. So, given $e$ is the vector toward the camera and $n$ is the normal vector, $\cos \theta \geq 0$ is the same as saying:

  $$e \cdot n \geq 0$$

- But, if we perform this calculation after transforming to normalized device coordinates (our $2 \times 2 \times 2$ cube), $e$ is merely $<0,0,1,0>$ in homogeneous coordinates

- Thus, backface culling is a matter of checking if the $z$ component of $n$ is greater than or equal to zero after NDC transformation!
Backface Culling == HSR
When...

- If everything we are only rendering a single, convex polyhedron, then backface culling is equivalent to HSR
- Backface culling is a straightforward switch in OpenGL — turn it on or off using `GL_CULL_FACE`
- Backface culling ≠ HSR if a polyhedron is not convex, or if there is more than one polyhedron involved:

![Diagram](image)

HSR Algorithms

- A number of HSR algorithms have been defined over the years — that’s what we’re describing today
- While there is no absolute “best” algorithm, there is a current prevailing “winner” due to the way it fits the cost/performance ratio of today’s technology
- The algorithms trade off on the following:
  1. Memory required
  2. Accuracy vs. speed
  3. Effect of increasing scene complexity on performance
  4. Hardware capabilities/limitations
Depth Sorting

• By Newell, Newell, & Sancha, 1972
• Paint each polygon in the scene in order, from the most distant to the nearest
• A “painter’s algorithm” that “naturally” performs HSR by progressively painting over the farthest polygons
• Two primary steps:
  ◦ Sort the polygons in occlusion-compatible order — that is, a sequence $P_i, P_2, P_3, \ldots, P_n$ such that for any polygon $P_i$, $1 \leq i \leq n$, $P_i$ hides (occludes) polygons $P_{i+1}$ to $P_n$
  ◦ Scan convert (paint) each polygon from $P_n$ down to $P_1$

• It’s all in the sorting!
  ◦ Useful preprocess: decompose polygons into triangles to simplify depth comparisons (tesselation) — because of polygons arranged like…

• A polygon sort algorithm would be:
  ◦ Determine a maximum $z$ for each polygon $P$
  ◦ Sort the polygons according to this maximum $z$
  ◦ For each polygon, make sure that all of the polygons that are “behind” it according to maximum $z$ are indeed hidden — we need to do this because sometimes maximum $z$ doesn’t imply occlusion:

Note how $P'$’s maximum $z$ is greater than $P''$’s but it is actually $P$ that is occluding $P''$
Overlap Testing for Depth Sort

- The catch in this algorithm is indeed in the overlap testing component — how do we do this accurately and quickly?

- There are 5 cases to test — if any one of them succeeds, then we can leave the polygons in their current maximum z-based order

- Otherwise, maximum z did not correspond to occlusion, so we swap the polygons involved

1. **Minimax depth test:** Minimum z of one polygon is less than the maximum z of the other polygon

2. **Minimax x–y test:** Polygons do not overlap in the x and y directions

3. **Behind-plane test:** All vertices of one polygon are behind the plane defined by the other polygon (derive plane equation to do this)

4. **In-front-of-plane test:** All vertices of one polygon are in front of the plane defined by the other polygon (plane equation again)

5. **Full overlap test:** Check for overlap in either the x or y directions and determine the respective z values at the overlapping area
Warnock Algorithm

Takes advantage of area coherence: divide the display area into successively smaller rectangles until the entire rectangle can be filled with a single color.

Warnock Pseudocode

```plaintext
algorithm doWarnock(x1, y1, x2, y2) {
    if rectangle is a pixel then {
        if no polygons map to this pixel then {
            set pixel to background color
        } else {
            set pixel to the color of the polygon closest to this pixel
        } end if
    } else {
        if no polygons overlap this rectangle then {
            set rectangle to background color
        } else if polygon(s) completely overlap this rectangle then {
            set rectangle to the color of the closest of these polygons
        } else {
            doWarnock(x1, y1, (x1 + x2) / 2, (y1 + y2) / 2)
            doWarnock(x1, (y1 + y2) / 2, (x1 + x2) / 2, y2)
            doWarnock((x1 + x2) / 2, y1, x2, (y1 + y2) / 2)
            doWarnock((x1 + x2) / 2, (y1 + y2) / 2, x2, y1
        } end if
    } end if
}
```
Scan-Line Algorithm

• a.k.a. “scan coherence” algorithm — not to be confused with z-buffer on one scan line (we’ll talk about z-buffer next)

• Display-oriented: instead of traversing the list of polygons, we go through the display’s pixels and figure out which polygon is “on” the current pixel

• Most efficient algorithm prior to lower-cost memory and specialized graphics hardware

• Also benefits from sorted surfaces and tesselation

• Traverse the display device one scan line at a time, left-to-right, top-to-bottom

• Check polygon list to see which ones intersect the current pixel

◊ Once we are “in” a polygon, we know that we will stay “in” it until we hit another one of that polygon’s edges (this is the core of “scan coherence”)

• When “in” polygons > 1, perform a depth check, and paint the color of the polygon that “wins” that check
Incremental z Calculation

• Two observations/assumptions:
  ◦ Polygons are flat — i.e., they lie on a plane
  ◦ As we traverse a polygon one scan line at a time, the z coordinate at that pixel changes at a constant rate (since the polygon is flat)

• Thus, we can have incremental calculation of the current z coordinate, which is faster than calculating it analytically from the current x and y coordinates (another use of the coherence concept)

• Calculate the “slope” of the polygon upon entry (based on the plane’s equation $Ax + By + Cz + D = 0$ — note how $<A, B, C, D>$ is the normal vector, expressed in homogeneous coordinates)

• Calculate the initial value of $z_i$ at the entry point:
  $$z_i = (-Ax_i - By_i - D) / C$$

• Since we are scanning along the x-axis, going from the entry point to the next is just +1 to the x coordinate:
  $$z_{i+1} = (-A(x_i + 1) - By_i - D) / C = (-Ax_i - By_i - D - A) / C$$
  $$z_{i+1} = z_i - (A / C)$$

• $A / C$ is constant per polygon — so, calculating the next $z$ is a single addition!
Z-Buffer Algorithm

- Very general and powerful technique — works for all polygon and occlusion cases (including cyclical)
- The trick — maintain a separate, parallel buffer for the depths (z coordinates) of the closest polygon at that pixel (thus, the synonymous “depth buffer” moniker)

<table>
<thead>
<tr>
<th>red</th>
<th>blue</th>
<th>red</th>
<th>brown</th>
<th>black</th>
</tr>
</thead>
<tbody>
<tr>
<td>white</td>
<td>green</td>
<td>red</td>
<td>red</td>
<td>black</td>
</tr>
<tr>
<td>white</td>
<td>green</td>
<td>red</td>
<td>brown</td>
<td>black</td>
</tr>
<tr>
<td>blue</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
<td>blue</td>
</tr>
<tr>
<td>green</td>
<td>white</td>
<td>red</td>
<td>red</td>
<td>brown</td>
</tr>
</tbody>
</table>

frame buffer — display device

| 0.5 | 0.8 | 0.5 | 0.2 | 1.0 |
| 0.35 | 0.44 | 0.5 | 0.5 | 1.0 |
| 0.35 | 0.44 | 0.5 | 0.2 | 1.0 |
| 0.8 | 0.8 | 0.8 | 0.8 | 0.8 |
| 0.44 | 0.35 | 0.5 | 0.5 | 0.2 |

z or depth buffer

Z-Buffer Pseudocode

```
clear frame buffer viewport to background color
clear depth buffer zbuffer to 1.0
for each polygon P
    for each pixel (x_{ndc}, y_{ndc}) to which P projects
        if z_{ndc} < zbuffer[x_{ndc}, y_{ndc}] then
            zbuffer[x_{ndc}, y_{ndc}] := z_{ndc}
            viewport[x_{ndc}, y_{ndc}] := color of P at (x_{ndc}, y_{ndc})
        endif
    endfor
endfor
```

- Note how the initial value of the depth buffer is 1.0 because in NDC, that is the maximum z value
- We use the coordinates after the viewport transform (i.e., conversion of –1…1 to width and height)
Z-Buffer Implementation Notes

• Because z-buffer also calculates the z per polygon per scan line, we can use the same incremental z calculation optimization as the scan-line algorithm.

• Note that z-buffer uses significantly more memory than the other algorithms — it needs a buffer with the same width/height as the viewport!
  ◊ Actual memory used would be \( \text{width} \times \text{height} \times \text{sizeof(real)} \)
  ◊ These days, \( \text{sizeof(real)} \) is around 4 to 8 bytes — bigger than a typical RGB pixel.
  ◊ Thus, a z-buffer implementation immediately requires at least thrice the desired display resolution: 2 swappable buffers for animation, and a third buffer for depth.
  ◊ See why it’s easy to outgrow video memory now?

OpenGL Uses Z-Buffer

• Preprocess polygons with backface culling, then use z-buffer (if enabled):
  
  ```gl
  glEnable(GL_DEPTH_TEST);
  // --- Activates depth buffer HSR.
  ```

• Depth buffer management corresponds to frame buffer management — it is allocated at the same time:
  
  ```gl
  glutInitDisplayMode( ... | ... | GLUT_DEPTH);
  // --- Initializes depth buffer.
  ```

• …and also reset along with the frame buffer
  
  ```gl
  glClear(... | ... | GL_DEPTH_BUFFER_BIT);
  // --- Sets depth buffer to 1.0.
  ```
HSR Algorithm Comparison

Sutherland, Sproull, and Schumaker, ACM Computing Surveys, March 1974 (yes, 1974)

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>100</th>
<th>2500</th>
<th>60000</th>
</tr>
</thead>
<tbody>
<tr>
<td>Depth Sort</td>
<td>1</td>
<td>10</td>
<td>507</td>
</tr>
<tr>
<td>Scan-Line</td>
<td>5</td>
<td>21</td>
<td>100</td>
</tr>
<tr>
<td>Warnock</td>
<td>11</td>
<td>64</td>
<td>307</td>
</tr>
<tr>
<td>z-Buffer</td>
<td>54</td>
<td>54</td>
<td>54</td>
</tr>
</tbody>
</table>

HSR Algorithm Conclusions

- Depth sort is quickest for a small number of polygons, but slows down significantly as polygon count increases
- Scan-line and Warnock performance also depends on polygon count, but they don’t degrade as quickly
- z-buffer is virtually independent of the number of polygons; more polygons means smaller regions to test (since the frame buffer’s size is fixed), so the total number of calculations tends to even out
- Final conclusion: the main drawback of z-buffer is memory use, but that’s cheap today, so z-buffer “wins”