Hidden Surface Algorithms
Reading

Reading:

- Angel 5.6, 10.12.2, 13.2 (pp. 654-655)

Optional reading:

- Foley, van Dam, Feiner, Hughes, Chapter 15
Introduction

In the previous lecture, we figured out how to transform the geometry so that the relative sizes will be correct if we drop the z component.

But, how do we decide which geometry actually gets drawn to a pixel?

Known as the hidden surface elimination problem or the visible surface determination problem.

There are dozens of hidden surface algorithms.

We look at three prominent ones:

- Z-buffer
- Ray casting
- Binary space partitioning (BSP) trees
Z-buffer

The Z-buffer or depth buffer algorithm [Catmull, 1974] is probably the simplest and most widely used.

Here is pseudocode for the Z-buffer hidden surface algorithm:

\[
\text{for each pixel } (i, j) \text{ do}
\]
\[
\text{Z-buffer} [i, j] \leftarrow \text{FAR}
\]
\[
\text{Framebuffer}[i, j] \leftarrow \text{<background color>}
\]
\[
\text{end for}
\]
\[
\text{for each polygon } A \text{ do}
\]
\[
\text{for each pixel in } A \text{ do}
\]
\[
\text{Compute depth } z \text{ and shade } s \text{ of } A \text{ at } (i, j)
\]
\[
\text{if } z > \text{Z-buffer} [i, j] \text{ then}
\]
\[
\text{Z-buffer} [i, j] \leftarrow z
\]
\[
\text{Framebuffer}[i, j] \leftarrow s
\]
\[
\text{end if}
\]
\[
\text{end for}
\]
\[
\text{end for}
\]

Q: What should FAR be set to?  \(-\text{BIG NUMBER}\)
Rasterization

The process of filling in the pixels inside of a polygon is called **rasterization**.

During rasterization, the z value and shade s can be computed incrementally (fast!).

![Diagram](image)

**Curious fact:**

- Described as the "brute-force image space algorithm" by [SSS]
- Mentioned only in Appendix B of [SSS] as a point of comparison for huge memories, but written off as totally impractical.

Today, Z-buffers are commonly implemented in hardware.
Z-buffer: Analysis

- Easy to implement?
- Easy to implement in hardware?
- Incremental drawing calculations (uses coherence)?
- Pre-processing required?
- On-line (doesn’t need all objects before drawing begins)?
- If objects move, does it take more work than normal to draw the frame?
- If the viewer moves, does it take more work than normal to draw the frame?
- Typically polygon-based?
- Efficient shading (doesn’t compute colors of hidden surfaces)?
- Handles transparency?
- Handles refraction?
Ray casting

Idea: For each pixel center $P_{ij}$

- Send ray from eye point (COP), $C$, through $P_{ij}$ into scene.
- Intersect ray with each object.
- Select nearest intersection.
Ray casting, cont.

Implementation:

- Might parameterize each ray:
  \[
  r(t) = C + t(P_i - C)
  \]
  where \( t > 0 \).

- Each object \( O_k \) returns \( t_k > 0 \) such that first intersection with \( O_k \) occurs at \( r(t_k) \).

Q: Given the set \( \{t_i\} \), what is the first intersection point?

\[
\hat{t}_1 = \text{smallest} \{t_i\}
\]

Note: these calculations generally happen in world coordinates. No projective matrices are applied.
Ray casting: Analysis

- Easy to implement?
- Easy to implement in hardware?
- Incremental drawing calculations (uses coherence)?
- Pre-processing required?
- Online (doesn’t need all objects before drawing begins)?
- If objects move, does it take more work than normal to draw the frame?
- If the viewer moves, does it take more work than normal to draw the frame?
- Typically polygon-based?
- Efficient shading (doesn’t compute colors of hidden surfaces)?
- Handles transparency?
- Handles refraction?
Binary-space partitioning (BSP) trees

\( \text{sign}[(x - p) \cdot N] \)

\[(x - p) \cdot N = 0\]

Idea:
- Do extra preprocessing to allow quick display from any viewpoint.

Key observation: A polygon A is painted in correct order if
- Polygons on far side of A are painted first
- A is painted next
- Polygons on near side of A are painted last.
BSP tree creation
BSP tree creation (cont’d)

procedure MakeBSPTree:
  takes PolygonList L
  returns BSPTree
    Choose polygon \( A \) from \( L \) to serve as root
    Split all polygons in \( L \) according to \( A \)
    node ← \( A \)
    node.neg ← MakeBSPTree(Polygons on neg. side of \( A \))
    node.pos ← MakeBSPTree(Polygons on pos. side of \( A \))
  return node
end procedure

Note: Performance is improved when fewer polygons are split --- in practice, best of ~ 5 random splitting polygons are chosen.

Note: BSP is created in \textit{world} coordinates. No projective matrices are applied before building tree.
BSP tree display

procedure DisplayBSPTree:
Takes BSPTree T, Point COP

if T is empty then return

if COP is in front (on pos. side) of T.node

DisplayBSPTree(T.______)

Draw T.node

DisplayBSPTree(T.______)

else

DisplayBSPTree(T.______)

Draw T.node

DisplayBSPTree(T.______)

end if

end procedure
BSP trees: Analysis

- Easy to implement?
- Easy to implement in hardware?
- Incremental drawing calculations (uses coherence)?
- Pre-processing required?
- On-line (doesn’t need all objects before drawing begins)?
- If objects move, does it take more work than normal to draw the frame?
- If the viewer moves, does it take more work than normal to draw the frame?
- Typically polygon-based?
- Efficient shading (doesn’t compute colors of hidden surfaces)?
- Handles transparency?
- Handles refraction?
Summary

What to take home from this lecture:

- Understanding of three hidden surface algorithms:
  - Z-buffering
  - Ray casting
  - BSP tree creation and traversal