9. Hidden Surface Algorithms

Optional reading

- Foley, van Dam, Feiner, Hughes, Chapter 15
- I. E. Sutherland, R. F. Sproull, and R. A. Schumacker, A characterization of ten hidden surface algorithms, *ACM Computing Surveys* 6(1): 1–55, March 1974.

Introduction

Once we transform all the geometry into screen space, we need to decide which parts are visible to the viewer.

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

There are dozens of hidden surface algorithms.

They can be characterized in at least three ways:

- Object-resolution vs. image-resolution (a.k.a, object-space vs. image-space)
- Object order vs. image order
- Sort first vs. sort last

Object-precision algorithms

- Basic idea:
  - Operate on the geometric primitives themselves
  - Objects typically intersected against each other
  - Tests performed to high precision
  - Finished list of visible objects can be drawn at any resolution
- Complexity:
  - Related to number of objects n
  - Typically $O(n^2)$
- Implementation:
  - Difficult to implement
  - Can get numerical problems

Image-precision algorithms

- Basic idea:
  - Find the closest point as seen through each pixel
  - Calculations performed at display resolution
  - Does not require high precision
- Complexity:
  - Related to resolution (number of pixels), R, of display
  - One measure is depth complexity, d – average number of objects along a pixel. Gives algorithm complexity of $O(dR)$.
- Implementation:
  - Very simple to implement!

Used a lot in practice!

Object order vs. image order

Object order:

- Consider each object only once, draw its pixels, and move on to the next object
- Might draw the same pixel multiple times

Image order:

- Consider each pixel only once, find nearest object, and move on to the next pixel
- Might compute relationships between objects multiple times

Sort first vs. sort last

Sort first:

- Find some depth-based ordering of the objects relative to the camera, then draw back to front
- Build an ordered data structure to avoid duplicating work

Sort last:

- Sort implicitly as more information becomes available

Outline of lecture

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

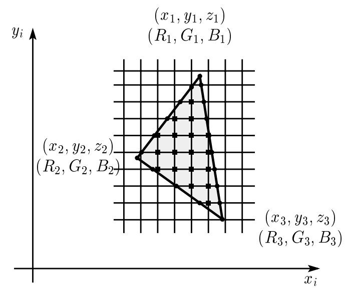
Z-buffer

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:

for each pixel (x, y) **do**
    Z-buffer$[x, y] \leftarrow$ -FAR
    $Fb[x, y] \leftarrow \langle$background color$\rangle$
end for
for each polygon P **do**
    for each pixel in P **do**
        Compute depth z and shade s of P at (x, y)
        if $z >$ *Z-buffer*$[x, y]$ **then**
            Z-buffer$[x, y] \leftarrow z$
            $Fb[x, y] \leftarrow s$
        end if
    end for
end for

Z-buffer, cont'd

The z value can be computed incrementally, like the shade s.



Curious fact:

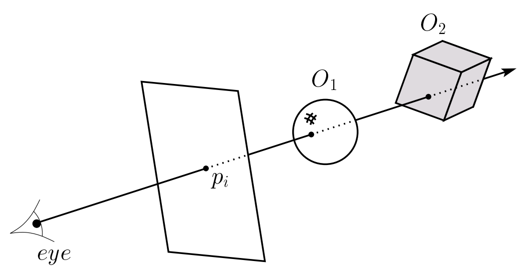
- Described as the "brute-force image space algorithm" by [SSS]
- Mentioned only in Appendix B as a point of comparison for huge memories, but written off as totally impractical

Today, Z-buffers are commonly implemented in hardware.

Z-buffer: Analysis

- Classification?
- Easy to implement?
- Hardware implementible?
- Incremental drawing calculations (uses coherence)?
- Memory intensive?
- Pre-processing required?
- On-line (doesn't need all objects in advance)?
- Handles transparency?
- Handles refraction?
- Polygon-based?
- Extra work for moving objects?
- Extra work for moving viewer?
- Efficient shading (doesn't compute colors of hidden surfaces)?
- Handles cycles and self-intersections?

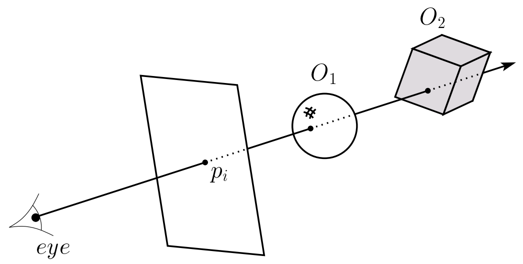
Ray casting



Idea: For each pixel p_i,

- Send ray from eye, through center of p_i, into scene
- Intersect ray with each object
- Select nearest intersection

Ray casting, cont.



Implementation:

- Might parameterize each ray:

$$R(t) \;=\; \mathbf{eye} + t(\mathbf{eye} - \mathbf{p_i})$$

- Each object O_i returns $t_i > 1$ such that first intersection with O_i occurs at $P_i = R(t_i)$.

- Foremost object is:

Ray casting: Analysis

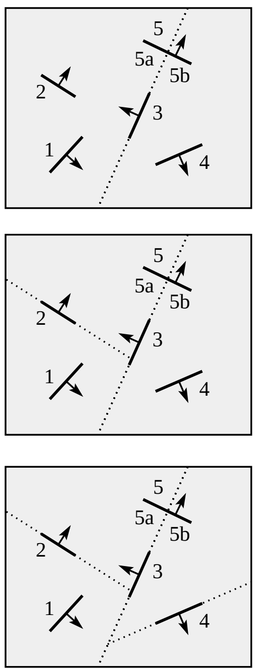
- Classification?
- Easy to implement?
- Hardware implementible?
- Incremental drawing calculations (uses coherence)?
- Memory intensive?
- Pre-processing required?
- On-line (doesn't need all objects in advance)?
- Handles transparency?
- Handles refraction?
- Polygon-based?
- Extra work for moving objects?
- Extra work for moving viewer?
- Efficient shading (doesn't compute colors of hidden surfaces)?
- Handles cycles and self-intersections?

Binary-space partitioning (BSP) trees

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

- Key observation: A polygon P is painted in correct order if
  - Polygons on far side of P are painted first
  - P is painted next
  - Polygons in front of P are painted last

BSP tree creation



BSP tree creation, cont'd

procedure *MakeBSPTree*:
takes *PolygonList* L
returns *BSPTree*
  Choose polygon P from L to serve as root
  Split all polygons in L according to P
  root $\leftarrow P$
  root.l $\leftarrow$ *MakeBSPTree*($\{$polygons on neg. side of $P\}$)
  root.r $\leftarrow$ *MakeBSPTree*($\{$polygons on pos. side of $P\}$)
  return *root*
end procedure

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

BSP tree display

procedure *DisplayBSPTree*:
takes *BSPTree* T
  if T is empty **then return**
  if viewer is in front of $T.root$ **then**
    DisplayBSPTree($T.l$)
    Draw $T.root$
    DisplayBSPTree($T.r$)
  else
    DisplayBSPTree($T.r$)
    Draw $T.root$
    DisplayBSPTree($T.l$)
  end if
end procedure

BSP trees: Analysis

- Classification?
- Easy to implement?
- Hardware implementible?
- Incremental drawing calculations (uses coherence)?
- Memory intensive?
- Pre-processing required?
- On-line (doesn't need all objects in advance)?
- Handles transparency?
- Handles refraction?
- Polygon-based?
- Extra work for moving objects?
- Extra work for moving viewer?
- Efficient shading (doesn't compute colors of hidden surfaces)?
- Handles cycles and self-intersections?

Summary

What to take home from this lecture:

1. Classification of hidden surface algorithms
2. Understanding of Z-buffer and ray casting hidden surface algorithms
3. Familiarity with BSP trees