9. Hidden Surface Algorithms

Reading

Required:

- Hearn and Baker, 13.1–13.3, 13.6–13.7

Optional:

- 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 divided into three categories:

- Object-space vs. image-space
- Object order vs. image order
- Sort first vs. sort last

Object-space algorithms

- Algorithm:
  - · 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-space algorithms

- Algorithm:
  - Takes advantage of raster display
  - Tests performed in image space, at display resolution
  - Does not require high precision
  - Does not give exact results, but good enough for display

- Complexity:
  - Related to resolution of display
  - Typically, $O(nr)$, for n objects and r pixels/object
  - Often, $nr = O(R)$, for R pixels in display, since more objects generally means size of each object is smaller

- Implementation:
  - Very simple to implement!

Used a lot in practice!

Object vs. image order and sorting time

Object vs. image order

- Are objects drawn one after another?
- Or, do we compute visibility one pixel at time?

Sort first vs. sort last

- Do we sort the objects so that we can draw them back-to-front?
- Or, do we decide visibility as we draw?

Outline of lecture

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

Z-buffer

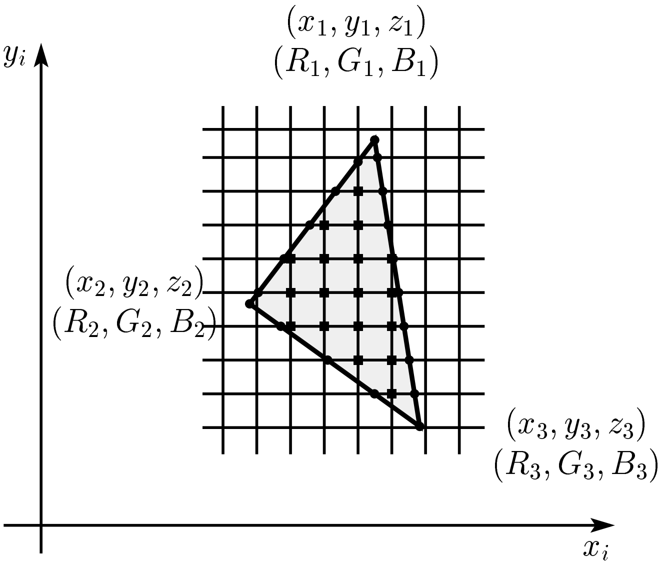
The "Z-buffer" or "depth buffer" algorithm [Catmull, 1974] is probably the simplest and most widely used.

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

> **for** each pixel (x, y) **do**
>   *Z-buffer*$[x, y] \leftarrow$ -FAR
> $Fb[x, y] \leftarrow$ ⟨background color⟩
> **end for**
> **for** each polygon 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**

Z-buffer, cont'd

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



Curious fact:

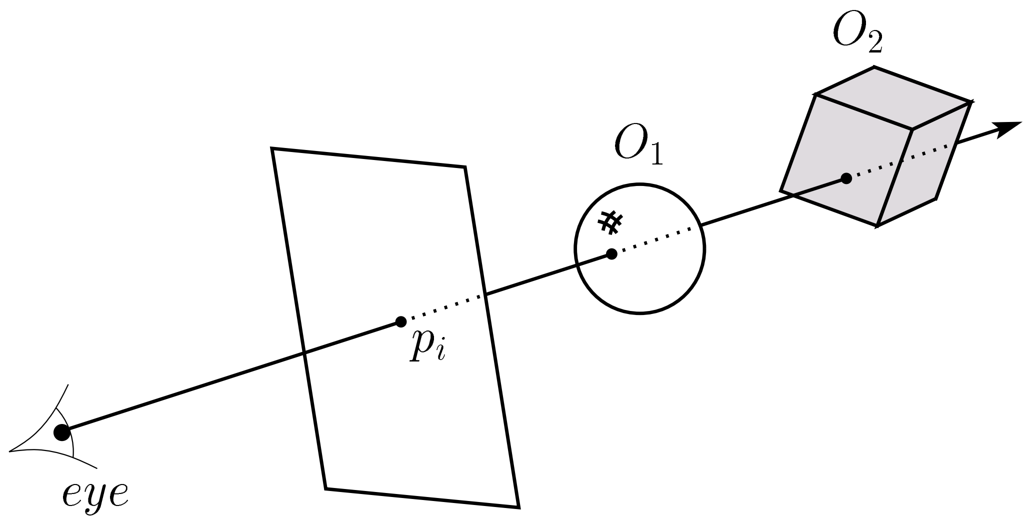
- Described as the "brute-force image space algorithm" by [SSS]
- Mentioned only in Appendix B as a point of comparison for huge memories, but written off as totally impractical

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?

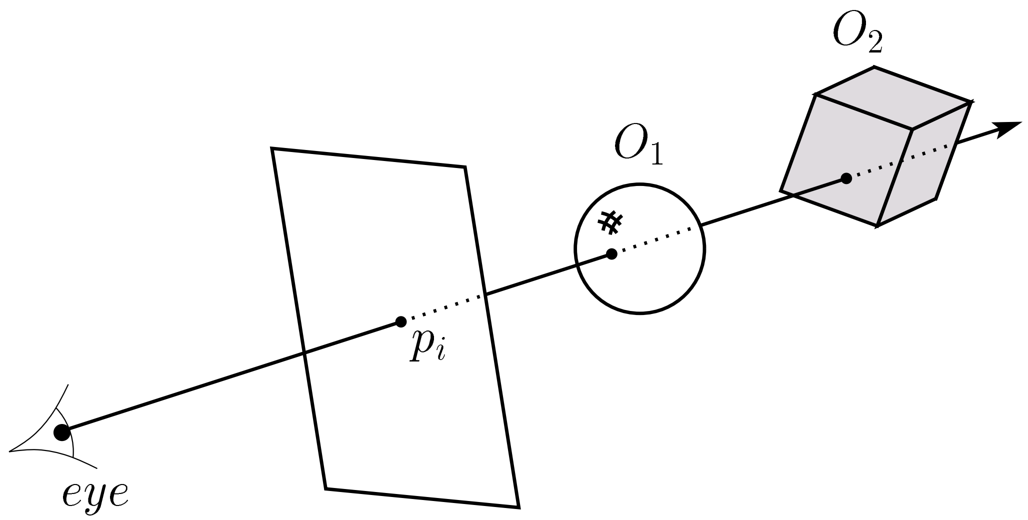
Ray casting



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:

$$R(t) \ = \ (1-t)\, eye \ + \ t\, p_i$$

- 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

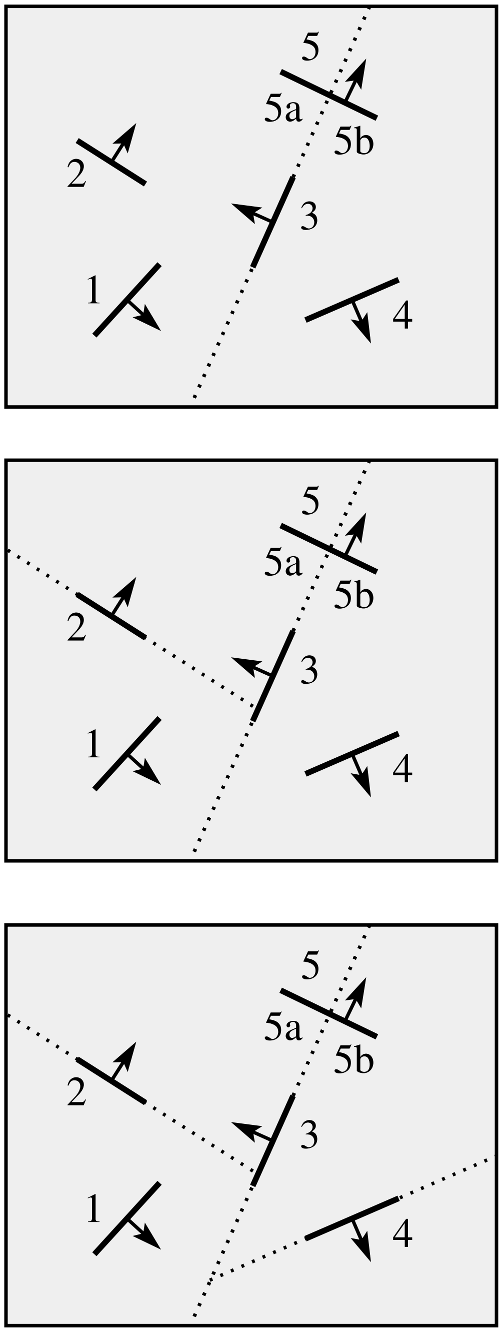
- Classification?
- Easy to implement?
- Hardware implementible?
- Incremental drawing calculations (uses coherence)?
- Memory intensive?
- Pre-processing required?
- On-line (doesn't need all objects in advance)?
- Handles transparency?
- Handles refraction?
- Polygon-based?
- Extra work for moving 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
    $root \leftarrow P$
    $root.l \leftarrow MakeBSPTree(\{\text{polygons on neg. side of } P\})$
    $root.r \leftarrow MakeBSPTree(\{\text{polygons on pos. side of } P\})$
    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