9. Hidden Surface Algorithms

Reading

Reading

- Watt, 6.6 (esp. intro and subsubsections 1, 4, and 8-10), 12.1.4.

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

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.

They can be characterized in at least three ways:

- Object-precision vs. image-precision (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. (We'll use "object" and "primitive" interchangeably.)
- Objects typically intersected against each other
- Tests performed to high precision
- Finished list of visible objects can be drawn at any resolution

Complexity:

- For n objects, can take $O(n^2)$ time to compute visibility.
- For an $m x m$ display, have to fill in colors for m^2 pixels.
- Overall complexity can be $O(k_{obj}n^2 + k_{disp}m^2)$

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:

- Naïve approach checks all n objects at every pixel. Then, $O(n\, m^2)$.
- Better approaches check only the objects that *could* be visible at each pixel. Let's say, on average, *d* objects are visible at each pixel (a.k.a. depth complexity). Then, $O(d\, m^2)$.

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 to 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.
- Means building 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.

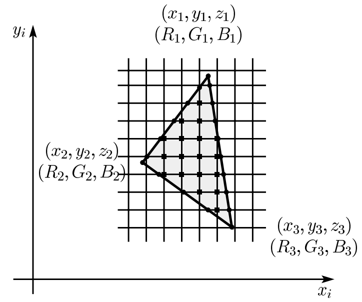
Here is pseudocode for the Z-buffer hidden surface algorithm:

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

Z-buffer (cont'd)

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!).



<u>Curious fact:</u>

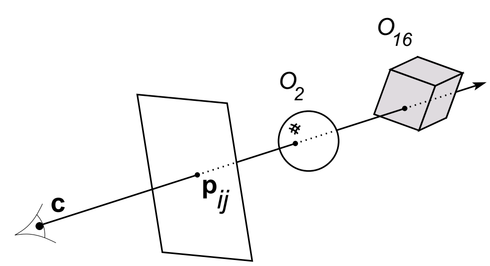
- Described as the ''brute-force image space algorithm" by [SSS]
- Mentioned only in Appendix B of [SSS] as a point of comparison for <u>huge</u> memories, but written off as totally impractical.

Today, Z-buffers are commonly implemented in hardware.

Z-buffer: Analysis

- Classification?
- 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 extra work than normal to draw the frame?
- If the viewer moves, does it take extra work than normal to draw the frame?
- Typically polygon-based?
- Efficient shading (doesn't compute colors of hidden surfaces)?
- Handles transparency?
- Handles refraction?

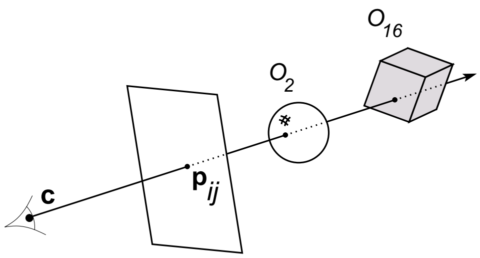
Ray casting



<u>Idea</u>: For each pixel center $\boldsymbol{p}_{ij}$

- Send ray from the eye point (COP), **c**, through $\boldsymbol{p}_{ij}$ into scene.
- Intersect ray with each object.
- Select nearest intersection.

Ray casting (cont'd)



Implementation:

- Might parameterize each ray:

  $\mathbf{r}(t) = \mathbf{c} + t\,(\mathbf{p}_{ij} - \mathbf{c})$

- Each object O_k returns $t_k > 1$ such that first intersection with O_k occurs at $\mathbf{r}(t_k)$.

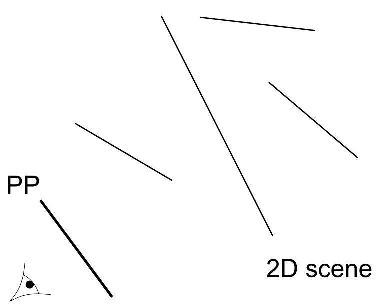
Q: Given the t_k what is the first intersection point?

Note: these calculations generally happen in world coordinates.

Ray casting: Analysis

- Classification?
- 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 extra work than normal to draw the frame?
- If the viewer moves, does it take extra 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



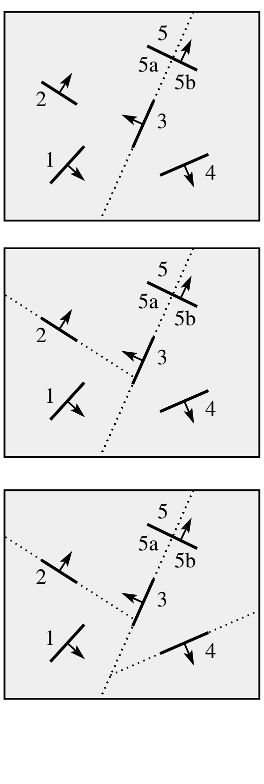
Idea:

- 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.
- P is painted next.
- Polygons in front 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 *world* coordinates.

BSP tree display

procedure *DisplayBSPTree*:

Takes *BSPTree T*

if *T* is empty **then return**

if viewer is in front (on pos. side) of *T.node* **then**

DisplayBSPTree(*T.neg*)

Draw *T.node*

DisplayBSPTree(*T.pos*)

else

DisplayBSPTree(*T.pos*)

Draw T.node

DisplayBSPTree(*T.neg*)

end if

end procedure

BSP trees: Analysis

- Classification?
- 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 extra work than normal to draw the frame?
- If the viewer moves, does it take extra 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:

- Classification of hidden surface algorithms
- Understanding of Z-buffer and ray casting hiddensurface algorithms
- Familiarity with BSP trees