Hidden Surface Algorithms

Brian Curless
CSE 457
Spring 2010

Reading

Reading:

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

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 <u>dozens</u> of hidden surface algorithms.

We look at three prominent ones:

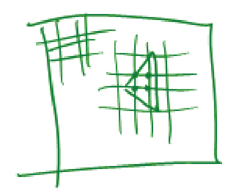
- Z-buffer
- Ray casting
- Binary space partitioning (BSP) trees

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:

```
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
```

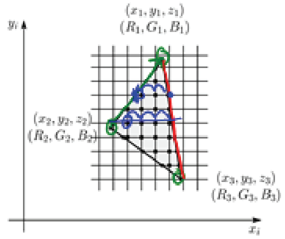


Q: What should FAR be set to? — BIG_NUMBER

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!).



<u>Curious fact:</u>

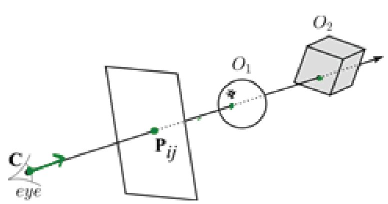
- Described as the "brute-force image space algorithm" by [SSS]
- Mentioned only in Appendix B of [SSS] as a point of comparison for <u>huge</u> memories, but written off as totally impractical.

Today, Z-buffers are commonly implemented in hardware.

Z-buffer: Analysis

- 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?

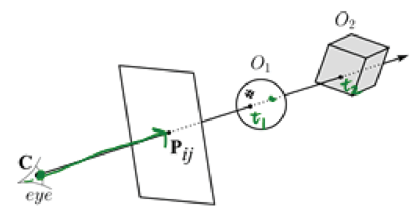
Ray casting



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:

  $\mathbf{r}(t) = \mathbf{C} + t\,(\mathbf{P}_{ij} - \mathbf{C})$

  where $t > 0$.

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

Q: Given the set $\{t_k\}$ what is the first intersection point?

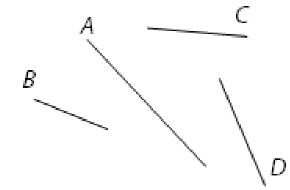
$t_n = \min_k \{t_k\}$

Note: these calculations generally happen in <u>world</u> 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?
- 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?

Binary-space partitioning (BSP) trees



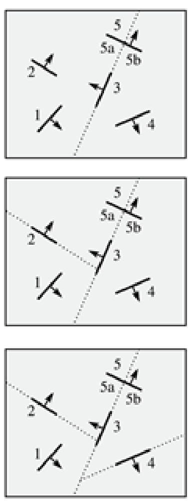
Idea:

- Do extra preprocessing to allow quick display from <u>any</u> 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 *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