Introduction

So far we know how to construct a hierarchical 3D model and map points from 3D to 2D. Is that all?

Not every surface of an object is visible from a given camera viewpoint. We need an algorithm to determine which parts get drawn.

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

Hidden surface algorithms can be characterized in at least three ways:

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

Object-space algorithms

Basic idea: operate on 3D objects

- For each object (3D primitive) in the scene, compute which part is visible, then draw
- Objects typically intersected against each other
- Tests performed to high precision
- Resulting list of visible objects can be drawn at any resolution

Complexity:

- May have to compare every pair of objects, so for n objects, can take $O(n^2)$ time
- For an $m \times m$ 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-space algorithms

**Basic idea:** operate on pixels
- Find the closest point as seen through each pixel
- Calculations performed at display resolution
- Precision requirements typically not high

**Complexity:**
- Naïve approach checks all $n$ objects at every pixel. Then, $O(n)$. 
- 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, $O(d)$. 

**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**

Idea: along with a pixel’s red, green and blue values, maintain some notion of its depth

- An additional channel in memory, like alpha
- Called the depth buffer or Z-buffer

```c
void draw_node_setup( void ) {  
  ...  
  glEnable( GL_DEPTH_TEST );  
  ...  
}
```

When the time comes to draw a pixel, compare its depth with the depth of what’s already in the framebuffer. Replace only if it’s closer

Very widely used

**History**

- Originally described as “brute-force image space algorithm”, mentioned in an appendix
- Written off as totally impractical algorithm (for huge memories)
- Today, done easily in hardware

---

**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).

**Z value interpolation**

![Diagram of Z value interpolation](image)

After projective normalization, the z values may be linearly interpolated within the image

- \( z_a = z_1 - (z_1 - z_2) \frac{y_1 - y_a}{y_1 - y_2} \)
- \( z_b = z_1 - (z_1 - z_2) \frac{y_1 - y_b}{y_1 - y_3} \)
- \( z_p = z_1 - (z_1 - z_2) \frac{x_b - x_p}{x_b - x_a} \)
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

Idea: For each pixel center $P_i$

- Send ray from eye point (COP), $c$, through $P_i$ into scene.
- Intersect ray with each object.
- Select nearest intersection.

Ray casting, cont.

Implementation:

- Might parameterize each ray:
  \[ r(t) = c + t \left( P_i - c \right) \]
- Each object $O_k$ returns $t_k > 1$ such that first intersection with $O_k$ occurs at $r(t_k)$.

Q: Given the set \( \{t_k\} \) 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 (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(Polys on - side of A)
    node.pos ← MakeBSPTree(Polys on + 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
        DisplayBSPTree(T.____)
        Draw T.node
        DisplayBSPTree(T.____)
    else
        DisplayBSPTree(T.____)
        Draw T.node
        DisplayBSPTree(T.____)
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?

Visibility tricks for Z-buffers

Z-buffering is the algorithm of choice for hardware rendering, so let’s think about how to make it run as fast as possible...

What is the complexity of the Z-buffer algorithm?

What can we do to decrease the constants?

Summary

What to take home from this lecture:

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