Hidden Surfaces

The Quest for 3D

- Construct a 3D hierarchical geometric model
- Define a virtual camera
- Map points in 3D space to points in an image
- Produce a wireframe drawing in 2D from a 3D object
- Of course, there’s more work to be done…

Introduction

- Not every part of every 3D object is visible to a particular viewer. We need an algorithm to determine what parts of each object should get drawn.
- Known as “hidden surface elimination” or “visible surface determination”.
- Hidden surface elimination algorithms can be categorized in three major ways:
  - Object space vs. image space
  - Object order vs. image order
  - Sort first vs. sort last
- Still a very active research area
- Where would we use a hidden surface algorithm?

Reading

- Foley et al, Chapter 15
Object Space Algorithms

- Operate on geometric primitives
  - For each object in the scene, compute the part of it which isn’t obscured by any other object, then draw.
  - Must perform tests at high precision
  - Resulting information is resolution-independent

- Complexity
  - Must compare every pair of objects, so \(O(n^2)\) for \(n\) objects
  - Optimizations can reduce this cost, but...
  - Best for scenes with few polygons or resolution-independent output

- Implementation
  - Difficult to implement!
  - Must carefully control numerical error

Image Space Algorithms

- Operate on pixels
  - For each pixel in the scene, find the object closest to the COP which intersects the projector through that pixel, then draw.
  - Perform tests at device resolution, result works only for that resolution

- Complexity
  - Must do something for every pixel in the scene, so at least \(O(R)\).
  - Easiest solution is test projector against every object, giving \(O(nR)\).
  - More reasonable version only does work for pixels belonging to objects: \(O(nr)\), \(r\) is number of pixels per object
  - Often, with more objects, each is smaller, so we estimate \(nr \approx O(R)\) in practice

- Implementation
  - Usually very simple!

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 - draw part of an 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 from back to front
  - Build an ordered data structure to avoid duplicating work

- Sort last
  - Sort implicitly as more information becomes available
Important Algorithms

- Ray casting
- Z-buffer
- Binary space partitioning
- Back face culling

Ray Casting

- Partition the projection plane into pixels to match screen resolution
- For each pixel \( p_i \), construct ray from COP through PP at that pixel and into scene
- Intersect the ray with every object in the scene, colour the pixel according to the object with the closest intersection

Aside: Definitions

- An algorithm exhibits coherence if it uses knowledge about the continuity of the objects on which it operates
- An online algorithm is one that doesn’t need all the data to be present when it starts running
  - Example: insertion sort

Ray Casting Analysis

Category:
- Easy to implement?
- Hardware implementation?
- Coherence?
- Memory intensive?
- Pre-processing required?
- Online?
- Handles transparency?
- Handles refraction?
- Polygon-based?
- Extra work for moving objects?
- Extra work for moving viewer?
- Efficient shading?
- Handles cycles and self-intersections?
**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_mode_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.”
- Written off as impractical algorithm for huge memories.
- Today, done easily in hardware.

**Z-buffer Implementation**

```c
for each pixel p, {
    Z-buffer[ p ] = FAR
    Fb[ p ] = BACKGROUND_COLOR
}
```

```c
for each polygon P {
    for each pixel p, in the projection of P {
        if z < Z-buffer[ p ] {
            Z-buffer[ p ] = z
            Fb[ p ] = s
        }
    }
}
```

**Z-buffer Tricks**

- The shade of a triangle can be computed incrementally from the shades of its vertices.
- Can do the same with depth.

![Z-buffer Tricks Diagram](image_url)

**Depth Preserving Conversion to Parallel Projection**

- Conversion method preserves depth information.
- Perspective and parallel projection matrices.

![Depth Preserving Conversion Diagram](image_url)
Z value interpolation

\[
\begin{align*}
\zeta_a &= z_i - (z_1 - z_2) \frac{y_i - y_1}{y_i - y_2} \\
\zeta_b &= z_i - (z_1 - z_2) \frac{y_i - y_1}{y_i - y_3} \\
\zeta_p &= z_i - (z_3 - z_2) \frac{x_i - x_2}{x_i - x_3}
\end{align*}
\]

Z-buffer Analysis

Categorization:
- Easy to implement?
- Hardware implementation?
- Coherence?
- Memory intensive?
- Pre-processing required?
- Online?
- Handles transparency?
- Handles refraction?
- Polygon-based?
- Extra work for moving objects?
- Extra work for moving viewer?
- Efficient shading?
- Handles cycles and self-intersections?

Binary Space Partitioning

- Goal: build a tree that captures some relative depth information between objects. Use it to draw objects in the right order.
  - Tree doesn’t depend on camera position, so we can change viewpoint and redraw quickly
  - Called the binary space partitioning tree, or BSP tree
- Key observation: The polygons in the scene are painted in the correct order if for each polygon \( P \),
  - Polygons on the far side of \( P \) are painted first
  - \( P \) is painted next
  - Polygons in front of \( P \) are painted last

Building a BSP Tree (in 2D)
Alternate BSP Tree

BSP Tree Construction

```cpp
BSPtree makeBSP( L: list of polygons )
{
  if L is empty
  {
    return the empty tree
  }
  Choose a polygon P from L to serve as root
  split all polygons in L according to P
  return new TreeNode( P, makeBSP( polygons on negative side of P ), makeBSP( polygons on positive side of P ) )
}
```

• Splitting polygons is expensive! It helps to choose P wisely at each step.
  – Example: choose five candidates, keep the one that splits the fewest polygons

BST Tree Display

```cpp
showBSP( v: Viewer, T: BSPtree )
{
  if T is empty then return
  P := root of T
  if viewer is in front of P
  {
    showBSP( back subtree of T )
    draw P
    showBSP( front subtree of T )
  } else
  {
    showBSP( front subtree of T )
    draw P
    showBSP( back subtree of T )
  }
}
```

BSP Tree Analysis

Categorization:
• Easy to implement?
• Hardware implementation?
• Coherence?
• Memory intensive?
• Pre-processing required?
• Online?
• Handles transparency?
• Handles refraction?
• Polygon-based?
• Extra work for moving objects?
• Extra work for moving viewer?
• Efficient shading?
• Handles cycles and self-intersections?
Back Face Culling

- Can be used in conjunction with polygon-based algorithms
- Often, we don’t want to draw polygons that face away from the viewer. So test for this and eliminate (cull) back-facing polygons before drawing
- How can we test for this?

Summary

- Classification of hidden surface algorithms
- Understanding of Z-buffer
- Familiarity with BSP trees and back face culling