4. Hidden Surface Algorithms and Ray Tracing

Reading

Recommended:

- H&B, Chapter 13 and Section 14.6
- Foley, Chapter 15 and Section 16.12
- 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.
- T. Whitted. An improved illumination model for shaded display. *Communications of the ACM* 23(6), 343-349, 1980.

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 problem."

(Or, sometimes, as the "visible surface problem.")

Overview

There are dozens of hidden surface algorithms.

They can be divided into three categories:

- Object-space algorithms
- Image-space algorithms
- List-priority algorithms

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

Object-space algorithms are not used much anymore, so we'll move right on. . .

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!

List-priority algorithms

- Algorithm — Kind of a hybrid:
  - Precompute a "priority" or "total visibility ordering" in object space
  - Rely on image space rendering to perform hidden surface
- Object-space character:
  - Depth overlap comparisons done in object space to high precision
- Image-space character:
  - Actual visibility resolved at each pixel

Outline of first part of lecture

- Image-space algorithms
  - Z-buffer
  - Z-buffer variations
  - Ray casting
- List-priority algorithms
  - Binary space partitioning (BSP) trees

Z-buffer

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

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

Description:

- Keep an extra buffer around for "Z" or "depth"
- Z-buffer in one-to-one correspondance with pixels of framebuffer

Today, Z-buffers are commonly implemented in hardware.

Z-buffer algorithm

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

@for@ each pixel (x, y) @do@
    $Z\text{-}buffer[x, y] \leftarrow 1.0$
    $Fb[x, y] \leftarrow \langle$background color$\rangle$
@end for@
@for@ each polygon P @do@
    Compute depth z and shade s of P at (x, y)
    @if@ $z < Z\text{-}buffer[x, y]$ @then@
        $Z\text{-}buffer[x, y] \leftarrow z$
        $Fb[x, y] \leftarrow s$
    @end if@
@end for@

Z-buffer algorithm, cont.

Note:

- The z value can be computed incrementally, like the shade s
- Efficiency is improved if polygons are drawn front-to-back

Z-buffer: Critique

Complexity:

Pros:

- Simple (very!)
  - · Easy to put in microcode
  - · Can be written as a very tight loop
- "On-line algorithm"
  - · Don't need all the polygons ahead of time
- Robust
  - · Doesn't explode for weird cases
  - · Works on any kind of object

Z-buffer: Critique, cont.

Cons:

- Memory intensive
- Inefficient in some respects:
  - Need to interpolate Z's
  - Need to do shading first
- No antialiasing:
  - Only allows one item visible per pixel
  - Loses surfaces that "fall between the cracks"
  - ("A-buffer" algorithm remedies these problems to some extent)
- No transparency:
  - Can sort polygons back-to-front
  - Fails if cyclic order or intersections

Variations on Z-buffer

A number of variations are possible — all image-space algorithms:

1. Scan-line algorithms

- Description:
  - Process each scan line ℓ top-to-bottom
  - Keep Z-buffer for polygons intersecting ℓ
  - Involves presorting polygons by $y_{\min}$
- Advantage:
  - Requires far less memory
- Disadvantages:
  - No longer an on-line algorithm
  - Need to presort polygons
  - Need to keep reinitializing buffers
  - More complex to implement

Variations on Z-buffer, cont.

2. Span coherence algorithms

- Description:
  - Process scan lines top-to-bottom
  - For each scan line, look at intersections of polygons with the xz-plane — called "spans"
  - Scan convert frontmost span in each overlapping region
- Advantages:
  - Eliminates Z-buffer altogether
  - Fewer depth comparison tests
  - Takes advantage of "area coherence"
- Disadvantages:
  - More complex — need to:
    - Break spans at every endpoint
    - Use intersection tests
  - Numerical difficulties possible

Variations on Z-buffer, cont.

3. Warnock's algorithm

- Description:
  - Recursively subdivide, until window is "simple":
    - Window is pixel-sized
    - No polygons in window
    - A single polygon covers window
  - Output contents of simple window
- Advantages:
  - Takes even better advantage of "area coherence"
  - Especially good for large polygons
- Disadvantages:
  - More complicated to implement
    - Have to clip polygons to windows
    - Have to keep track of polygons to test for each window
  - Does not produce output in scan-line order

Ray casting

Idea: For each pixel p,

- Send ray from eye, through center of p, into scene
- Intersect ray with each object
- Select nearest intersection

Ray casting, cont.

Implementation:

- Might parameterize each ray:

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

- 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: Critique

Complexity:

Pros:

Cons:

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

@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*({polygons on neg. side of P})
    $root.r \leftarrow$ *MakeBSPTree*({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: Critique

Complexity:

Pros:

Cons:

Ray tracing

(H&B, fig 14-49)

Idea: "Ray tracing" is like "ray casting," only you continue to bounce the ray around the scene.

Provides:

- Visible-surface detection
- Shadow effects
- Transparency
- Multiple light source illumination

Overview of rest of lecture

1. Basic ray-tracing algorithm
2. Ray–surface intersection calculations
   - Spheres
   - Polygons
3. Acceleration techniques
   - Hierarchical bounding volumes
   - Space-subdivision methods
   - Progressive methods
4. Antialiasing
   - Supersampling
   - Adaptive sampling
   - Distributed ray tracing

Ray-tracing algorithm

Algorithm:

1. For each pixel, trace a "primary ray" to the first visible surface (same as "ray casting").
2. For each intersection, trace "secondary rays":
   - "Reflected ray" in direction $\mathbf{R}$.
   - "Refracted ray" in direction $\mathbf{T}$.
   - "Shadow rays" in directions $\mathbf{L}_i$ to light sources.

Shading

Let $I(p, r)$ be the intensity seen from point p along direction r:

$$I(p, r) \;=\; I_{\text{direct}} \;+\; I_{\text{reflected}} \;+\; I_{\text{transmitted}}$$

where

- I_{direct} is computed from the Phong model
- $I_{\text{reflected}} \;=\; k_{\text{reflected}}\, I(P, \mathbf{R})$
- $I_{\text{transmitted}} \;=\; k_{\text{transmitted}}\, I(P, \mathbf{T})$

Reflected and transmitted directions

Law of reflection:

$$\theta_i \;=\; \theta_r$$

Snell's law of refraction:

$$\eta_i \sin\theta_i \;=\; \eta_t \sin\theta_t$$

where η_i, η_t are "indices of refraction."

The ray-tracing tree

As the recursively traced rays ricochet through the scene, they are added to a "ray-tracing tree":

The ray-tracing tree (H&B, fig 14-52)

Ray-tracing pseudocode

@function@ $RayTrace(p, r)$:
  $(P, Obj) \leftarrow RayCast(p, r)$
  $I \leftarrow Obj.k_a * I_a$
  @for@ each light source ℓ @do@:
    $(P', LightObj) \leftarrow RayCast(P, Dir(P, \ell))$
    @if@ $LightObj = \ell$ @then@:
      $I \leftarrow I +$ ⟨diffuse term⟩ $+$ ⟨spec term⟩
    @end if@
  @end for@
  $I \leftarrow I + Obj.k_{\text{reflected}} * RayTrace(P, \mathbf{R})$
  $I \leftarrow I + Obj.k_{\text{transmitted}} * RayTrace(P, \mathbf{T})$
  @return@ I
@end function@

Terminating recursion

Q: How do you bottom out of recursive ray tracing?

Possibilities:

Intersecting rays with spheres

Given:

- A ray $\mathbf{P}(s)$ through initial position $\mathbf{P}_0$ in direction $\mathbf{u}$ (a unit vector):

$$\mathbf{P}(s) \;=\; \mathbf{P}_0 \;+\; s\,\mathbf{u}$$

- A sphere S centered at $\mathbf{P}_c$ with radius r

Find: The intersection of $\mathbf{P}(s)$ with S.

Solution: (see also (14-58), p533)

Intersecting rays with polyhedra

Polyhedron enclosed by bounding sphere (H&B, fig 14-58)

To intersect a ray with a polyhedron:

1. Test intersection of ray with bounding sphere.
2. Locate the "front-facing" faces of the polyhedron with
$$\mathbf{u} \cdot \mathbf{N} < 0$$
3. Intersect the ray with each front face's supporting plane.
4. Use a point-in-polygon test to see if the ray is inside the face.
5. Sort intersections according to smallest s.

Acceleration techniques

Vanilla ray tracing is really slow!

In practice, some acceleration technique is almost always used:

1. Hierarchical bounding volumes

A group of objects within a bounding sphere (H&B, fig 14-61)

Acceleration techniques, cont.

Spatial subdivision (H&B, fig 14-62)

2. Space subdivision methods

Idea:

- Partition objects spatially.
- Trace ray through voxel array.

Partition can:

- Be uniform or adaptive
- Use octtrees or k-d trees

Acceleration techniques, cont.

3. Adaptive progressive refinement

Idea [Painter and Sloan]:

- Interpolate between ray traced samples.
- Only send rays where variance is high.

Antialiasing

One straightforward way to do antialiasing is to supersample, either uniformly or adaptively:

Supersampling by tracing rays through corners (H&B, figs 14-69, 14-70)

Distributed ray tracing

Jittered supersampling (H&B, fig 14-73)

Idea:

- Use non-uniform (jittered) samples.
- Replaces aliasing artifacts with noise.
- Provides additional effects by distributing rays in other dimensions:
  - Reflection and refractions
  - Texture mapping
  - Light source area
  - Camera lens area
  - Time

Distributed ray tracing, cont.

(H&B, fig 14-75)

Distributing rays over reflection and refraction directions gives:

Distributed ray tracing, cont.

(H&B, figs 14-76, 14-77)

Distributing rays over light source area gives:

Distributed ray tracing, cont.

(H&B, fig 14-74)

Distributing rays over a finite aperture gives:

Distributed ray tracing, cont.

(H&B, fig 14-78)

Distributing rays over time gives:

Summary

What to take home from this lecture:

1. Classification of hidden surface algorithms
2. Thorough understanding of Z-buffer and ray casting hidden surface algorithms (to the level of being able to implement)
3. Familiarity with Z-buffer variations and BSP trees
4. Enough to implement basic recursive ray tracing.
5. How reflection and transmission directions are computed.
6. How ray–object intersection tests are performed.
7. Basic acceleration strategies.
8. Concept of distributed ray tracing.