Visibility Algorithms

Roger Crawfis

CIS 781

This set of slides reference slides used at Ohio State for instruction by Prof. Machiraju and Prof. Han-Wei Shen.

Visibility Determination

Visibility Algorithms

Hidden Lines Removed

Hidden Lines

Hidden Lines Removed
Where Are We?

Canonical view volume (3D image space)

Back-face Culling

Problems?

Conservative algorithm

Real job of visibility never solved

Taxonomy

BSP Tree

List Priority, NNA

Warnock

Warnock-Weiler

Spanning Scanline

Z-buffer

Hidden Object Removal: Painters Algorithm

Backface Culling

Topics

Hidden Surfaces Removed
Back-face Culling

- If a surface's normal is pointing in the same general direction as our eye, then this is a back face.
- The test is quite simple: if \( N \cdot V > 0 \) then we reject the surface.

- If test is in eye-space, then if \( N_z > 0 \) reject.

Back-face Culling

- Only handles faces oriented away from the viewer:
  - Closed objects
  - Near clipping plane does not intersect the objects
  - Nearest clipping plane does not intersect the objects
  -Closest objects away from the viewer

Point sorting vs Polygon sorting

- What does it mean to sort two line segments?
- Z min?
- Z max?
- Length?
- Slope?

Painters Algorithm

- Sort objects in depth order
- Draw all from back-to-front (far-to-near)
- Simply overwrite the existing pixels.

- Simply overwrite the existing pixels.
- Is it so simple?

Back-face Culling

- The test is quite simple: if \( \Lambda \cdot N \neq 0 \) then we reject the surface.
- In eye-space, then if \( \Lambda \cdot N \neq 0 \) then we reject the surface. If \( \Lambda \cdot N = 0 \) then this is a back face.
- The test is quite simple: if \( \Lambda \cdot N = 0 \) then we reject the surface.

- Simply overwrite the existing pixels.
- Is it so simple?
How do we deal with cycles?

How do we deal with intersections?

How do we sort objects that overlap in Z?

**Form of the Input**

- Object types: what kind of objects does it handle?
  - convex vs. non-convex
  - polygons vs. everything else - smooth curves, non-
  - continuous vs. everything except non-
  - convex vs. non-convex

- Object types: what kind of objects does it handle?

**Object Space Algorithms**

- Volume testing – Weiler-Atherton, etc.
  - input: convex polygons + infinite eye pt
  - output: visible portions of wireframe edges

**Form of the Output**

- Output: visible portions of wireframe edges
- Input: convex polygons + infinite eye pt
- Volume testing – Weiler-Atherton, etc.

**Object Space**

- Geometry in, geometry out
- Independent of image resolution
- Followed by scan conversion

**Image Space**

- Geometry in, geometry out
- Image in, geometry out
- Image in, geometry out

**Object Space**

- Geometry in, geometry out
- Independent of image resolution
- Followed by scan conversion

**Image Space**

- Geometry in, geometry out
- Image in, geometry out
- Image in, geometry out

**Precision: Image/Object Space?**

**Input of the Output**

- Continuous surfaces, volometric data
- Polygons vs. everything else - smooth curves, non-
- Convex vs. non-convex

**3D Cycles**

- How do we sort objects that overlap in Z?
- How do we deal with intersections?
- How do we deal with cycles?
Image-space algorithms

Traditional Scan Conversion and Z-buffering

Hierarchical Scan Conversion and Z-buffering

input: any plane-sweepable/plane-boundable objects
preprocessing: none
output: a discrete image of the exact visible set

Conservative Visibility Algorithms

Viewport clipping
Back-face culling
Warnock's screen-space subdivision

Z-buffer

Z-buffer is a 2D array that stores a depth value for each pixel.

\[ \frac{\partial}{\partial x} z = \frac{\partial}{\partial y} z = \frac{\partial}{\partial y} z = \frac{\partial}{\partial x} z = 0 \]

\[ z = z \]

\[ \text{if } z \leq \text{Zbuffer}[x][y] \]

\[ \text{then } \]

\[ \text{Screen}[x][y] = \text{color} \]

\[ \text{Zbuffer}[x][y] = z \]

\[ \text{DrawZpixel}(x, y, z, \text{color}) \]

\[ \text{Screen}[x][y] = \text{color} \]

\[ \text{Zbuffer}[x][y] = z \]

\[ \text{DrawZpixel}(x, y, z, \text{color}) \]

\[ \text{InitScreen} \]

\[ \text{for } i = 0 \text{ to } N \]

\[ \text{do } \]

\[ \text{for } j = 0 \text{ to } N \]

\[ \text{do } \]

\[ \text{Screen}[i][j] = \text{BACKGROUND_COLOR} \]

\[ \text{Zbuffer}[i][j] = \infty \]

\[ \text{for each polygon projection do} \]

\[ \text{for each } (x, y) \text{ in the polygon's projection do} \]

\[ z := -(D + A*x + B*y)/C \]

\[ \text{DrawZpixel}(x, y, z, \text{polygon's color}) \]

\[ \text{if } z \leq \text{Zbuffer}[x][y] \]

\[ \text{then } \]

\[ \text{Screen}[x][y] = \text{color} \]

\[ \text{Zbuffer}[x][y] = z \]

\[ \text{DrawZpixel}(x, y, z, \text{color}) \]

\[ \text{for each } (x_1, x_2) \text{ of } X \text{-intersections do} \]

\[ \text{for } x = x_1 \text{ to } x_2 \text{ do} \]

\[ z := -(D + A*x + B*y)/C \]

\[ \text{DrawZpixel}(x, y, z, \text{polygon's color}) \]

If we know \( z \) at \((x, y)\) then:

\[ z_{x+1, y} = \frac{z_x - A}{C} \]
Incremental Scanline

On a scan line \( Y = j \), a constant
Thus depth of pixel at \((x_1 = x + \Delta x, j)\)

\[
\frac{(x' - x)}{(x' - x)} (z' - z) + z = z
\]

\[
\frac{(\bar{y} - \bar{y})}{(\bar{y} - \bar{y})} (z' - z) + z = z
\]

\[
\frac{(\bar{y} - \bar{y})}{(\bar{y} - \bar{y})} (z' - z) + z = z
\]

- Incremental Scanline (contd.)
- Non-Planar Polygons
- Bilinear Interpolation of Depth Values

\[
\frac{b - z}{a} - z = z
\]

\[
' = ax \quad \text{Since } ax 'y'
\]

\[
\frac{(y - y)}{(y - y)} z - z
\]

\[
\frac{(x - x)}{(x - x)} z - z
\]

Thus depth of pixel at \((x, y)\)

On a scan line \( y \neq i \) a constant

\[
0 \neq \frac{c}{d + b + x} \quad \text{Thus depth of pixel at } (x, y)
\]

\[
4x + c = z
\]

\[
0 = d + b + c + a
\]

Incremental Scanline
Non Trivial Example?

Frame Buffer: Background 0, Rectangle 1, Triangle 2

Z-buffer: 32x32x4 bit planes

Rectangle: P1(10,5,10), P2(10,25,10), P3(25,10,25)
P4(25,5,10)

Triangle: P5(15,15,15), P6(25,25,5), P7(30,10,5)

Example
Z-Buffer Advantages

- Simple and easy to implement
- Amenable to scan-line algorithms
- Can easily resolve visibility cycles
- Handles intersecting polygons
- Can easily resolve visibility cycles
- Amenable to scan-line algorithms
- Simple and easy to implement

Z-Buffer Disadvantages

- Does not do transparency easily
- Aliasing occurs! Since not all depth questions can be resolved
- Anti-aliasing solutions non-trivial
- Spanning Scan-Line Algorithm

How do you deal with this – scan-conversion algorithm and a little more data structure.

Can we do better than scan-line Z-buffer?

Spanning Scan-Line

- Use no Z-buffer
- Each scan line is subdivided into several "spans".
- Determine which polygon the current span belongs to
- Shade the span using the current polygon's color
- Span coherence
- Scan-line coherence across multiple scan-lines
- Or span-coherence?
- Depth coherence
- How do you deal with this – scan-conversion algorithm and a little more data structure.

Spanning Scan-Line Algorithm

- Use no Z-buffer
- Each scan line is subdivided into several "spans".
- Determine which polygon the current span belongs to
- Shade the span using the current polygon's color
- Span coherence
- Scan-line coherence across multiple scan-lines
- Or span-coherence?

Higher order illumination is hard in general:

- Shadows are not easy
- Anti-aliasing solutions non-trivial
- Aliasing occurs! Since not all depth questions can be resolved
- Does not do transparency easily
Spanning Scan Line Algorithm

- A scan line is subdivided into a sequence of spans.
- Each span can be inside or outside polygons.
- If a span is inside one polygon, the pixels in the span will be drawn with the color of that polygon.
- If a span is inside more than one polygon, then we need to compare the z values of those polygons at their intersection point to determine the color of the scan line span.

When there are multiple polygons:
- Each polygon will have its own in/out flag.
- There can be more than one polygon having the in/out flag to be "in" at a given instance.
- We want to keep track of how many polygons the scan line is currently in.
- We need to perform z value comparison to determine the color of the scan line span.

When there is a single polygon:
- We use a "in/out" flag for each polygon to keep track of the current state.
- Initially, the in/out flag is set to be "outside".
- For a 1st time, the span becomes "inside" of the polygon.
- For a 2nd time, the span becomes "outside" of the polygon.
- When a scan line intersects an edge of a polygon, the span becomes "inside" of the polygon.
- Each time a scan line intersects an edge of a polygon, the span becomes "outside" of the polygon.
- When a scan line intersects an edge of a polygon, the span becomes "inside" of the polygon.
Z value comparison

- When the scan line intersects an edge, leaving the top-most polygon, we use the color of the remaining polygon if there is now only 1 polygon "in".
- If there is still more than one polygon with an "in" flag, we need to perform $z$ comparison, but only when the scan line leaves a non-obscured polygon.

Example: Spanning Scan-Line

- Use IPL as active In-Polygon List.
- Multiple polygons can have their flags set to true.
- When polygon is considered, Flag is true.
- Multiple polygons can have their flags set to true.
- Use IPL as active In-Polygon List.
- When polygon is considered, Flag is true.

Many Polygons:

Use a PT entry for each polygon:
- Color, In/Pt flag, poly-ID

When scan line intersects an edge, leaving the top-most polygon, we use the color of the remaining polygon.

If there is still more than one polygon, perform $z$ comparison with the remaining polygon.

When the scan line intersects an edge, leaving the remaining polygon, we use the $z$ value comparison.
Scan Line I: Polygon S is in and flag of S = true

Scan Line II: Both S and T are in and flags are disjointly true

Scan Line III: Both S and T are in simultaneously

Scan Line IV: Same as Scan Line II

Some Facts:

Penetrating Polygons

Depth Coherence

Penetrating Polygons

Depth Coherence

Spanning Scan-Line

False edges and new polygons!
**Warnock's Algorithm**

Starting with the entire display, we check the following four cases. If none hold, we subdivide the area and repeat, otherwise, we stop and perform the action associated with the case:

1. All polygons are disjoint, and none of the polygons overlap the area: draw the background color.
2. Only one intersection or contained polygon: draw the background and then draw the contained portion of the polygon.
3. There is a single surrounding polygon: draw the entire area.
4. There is more than one intersecting, contained, or surrounding polygon: draw the area that is not covered by the polygons.

At a Single Pixel Level:

When the recursion stops and none of the cases hold, we need to perform a depth sort and draw the polygon with the closest Z value. The algorithm is done at the object space level, except for scan conversion and clipping, which are done at the image space level.

**Warock's One Polygon**

If the polygon's color is not the background color, then:

- If it is not contained:
  - If it intersects:
    - Draw intersected portion of polygon.
  - Else:
    - Draw rectangle.
- Else:
  - Draw rectangle (poly-color).

The recursion stops at the pixel level if the area is contained in the polygon's color. Otherwise, the surrounding polygon is drawn first.
Warnock: Zero/One Polygons

```
warnock(rectangle, poly-list)
new-list := clip(rectangle, poly-list);
if length(new-list) = 0 then
draw_rectangle(BACKGROUND); return;
if length(new-list) = 1 then
draw_rectangle(BACKGROUND);
draw_poly(poly); return;
if rectangle size = pixel size then
poly := closest polygon at rectangle center
draw_rectangle(poly color); return;
warnock(top-left quadrant, new-list);
warnock(top-right quadrant, new-list);
warnock(bottom-left quadrant, new-list);
warnock(bottom-right quadrant, new-list);
```

### Zero-Polygons

- `clip(rectangle, poly-list)`
- `draw_rectangle(BACKGROUND)`
- `draw_poly(poly)`

### One-Polygons

- `closest polygon at rectangle center`
- `draw_rectangle(poly color)`
Area Subdivision 2

Weiler-Atherton Algorithm

Output – polygons of arbitrary accuracy

Like Warnock

Object space
Weiler-Atherton Clipping

• General polygon clipping algorithm
• Allows one to clip a concave polygon against another concave polygon.

• First, find all of the intersection points between edges of the two polygons.

A
B
C
D
E
a
b
c
d
e
S
T

• Now, rebuild the polygons such that they include the intersection points in their clockwise ordering.

S: A,1,4,B,2,6,C,D,5,3,E
T: a,4,2,b,6,c,5,d,e,3,1

• Find an intersecting vertex of the polygon to be clipped that starts outside and goes inside the clipping region.

• Traverse the polygon until another intersection point is found.

• Then, find all of the intersection points between edges of the two polygons.

• Finally, rebuild the polygons such that they are in clockwise order.

• Now, repeat the process for the other polygon.

Weiler-Atherton Clipping
Weiler-Atherton Clipping

• Switch from walking around polygon 1, to walking around polygon 2, when an intersection is detected.
• Stop when we reached the initial point.

Algorithm:

1. Sort the polygons based on their minimum z distance
2. Choose the first polygon P in the sorted list
3. Clip all polygon fragments on the inside list that are behind P
4. All polygon fragments on the inside list that are in front of P are discarded. If there are polygons on the inside list that are in front of P, go back to step 3, use the 'offending' polygons as P
5. Display P and go back to step 3.

Weiler-Atherton Algorithm

\[
\text{WA}_\text{display}(\text{polys}):
\]
\[
\text{sort}_\text{by}_\text{minZ}(\text{polys});
\]
\[
\text{while } (\text{polys} \neq \text{NULL})
\]
\[
\text{WA}_\text{subdiv}(\text{polys}->\text{first}, \text{polys});
\]
\[
\text{end};
\]
\[
\text{WA}_\text{subdiv}(\text{first}:: \text{Polygon} ; \text{polys}:: \text{ListOfPolygons})
\]
\[
\text{inP}, \text{outP}:: \text{ListOfPolygons} := \text{NULL};
\]
\[
\text{for each } P \text{ in polys do}
\]
\[
\text{Clip}(\text{P}, \text{first}->\text{ancestor}, \text{inP}, \text{outP});
\]
\[
\text{for each } P \text{ in inP do if } P \text{ is behind (min z) first then }
\]
\[
\text{discard } P;
\]
\[
\text{for each } P \text{ in inP do if } P \text{ is not part of first then }
\]
\[
\text{WA}_\text{subdiv}(P, \text{inP});
\]
\[
\text{for each } P \text{ in inP do }
\]
\[
\text{display_a_poly}(P);
\]
\[
\text{polys} := \text{outP};
\]
\[
\text{end};
\]

Stop when we reached the initial point.

• Switch from walking around the polygon 1, to walking around polygon 2, when an intersection is detected.

Weiler-Atherton Clipping
List Priority Algorithms

• Find a valid order for rendering.
• Only consider cases where the sort matters.

How do we sort? — different algorithms differ

It is easy then to implement transparency.

Front (lowest priority) order (Painter's Algorithm).
Sorts by Z and rendered in back (highest priority) - to-
If they do not overlap in the Z dimension they can be
For hidden object removal process.
If objects do not overlap in X or in Y there is no need

List Priority Algorithms
Newell, Newell, Sancha Algorithm

1. Sort by [minz..maxz] of each polygon
2. For each group of unsorted polygons G
   resolve_ambiguities(G);
3. Render polygons in a back-to-front order.

resolve_ambiguities is basically a sorting algorithm that relies on
the procedure rearrange(P, Q):

\[
\text{rearrange}(P, Q, \text{flag})
\]

\[
\text{if (P and Q do not have overlapping x-extents, return P, Q)}
\]

\[
\text{if (P and Q do not have overlapping y-extents, return P, Q)}
\]

\[
\text{if all Q is on the opposite side of P from the eye return P, Q}
\]

\[
\text{if all P is on the same side of Q from the eye return Q, P}
\]

\[
\text{if not overlap-projection(P, Q)} \text{ (Q is on the opposite side of P)}
\]

\[
\text{split(P, Q, p1, p2); -- split P by Q}
\]

\[
\text{return (p1, p2, Q)};
\]

Newell-Newell-Sancha Sorting

\cdot Q is on the opposite side of P.
\cdot Means, all of Q's vertices are in front of the half-plane defined by P.
\cdot P is on the same side of Q.
\cdot Means, all of P's vertices are behind the half-plane defined by Q.

Newell-Newell-Sancha Algorithm
1. Sort by [minz..maxz] of each polygon G.
2. For each group of unsorted polygons G
   resolve_ambiguities(G);
3. Render polygons in a back-to-front order.
Spatial Subdivision

Sorting for Uniform Grid

Parallel to the viewing direction to traverse last.
- Better ordering would choose the axis most parallel, go forward for back-to-front sort.
- Look at the z-value of the transformed x-axis, then y.
- Backward on each axis.
- Simply need to decide whether to go forward or backward on each axis.
- Can always proceed along the x-axis, then y.
- Can always proceed along the x-axis, then y.

Back-to-front Traversals

For the first four, you can develop either a front-to-back or back-to-front traversal order explicitly.
- Thereby, solving the visibility sort efficiently.
- For the polyhedra, use a Newell-Newell-Sancha sort.

Sorting for Uniform Grid

- Parallel Projection
  - Can always proceed along the x-axis, then y.
  - Better ordering would choose the axis most parallel to the viewing direction to traverse last.

Spatial Subdivision

- Uniform grid
- Axis-Aligned Bounding Boxes (AABB's)
- Oriented Bounding Boxes (OBB's)
- Non-overlapping polyhedra
  - BSP-trees
  - K-d Trees
  - Octrees
  - Uniform grid

A characterization of 10 Hidden Surface Algorithms:

- A'priori
- Dynamic
- Roberts
- Newell
- Warnock
- Sutherland, Sproull, Schumaker (1974)

Image Space

Object Space

Volume

List Priority

Point

Are

Volume

Object Space

Spatial Subdivision

Back-to-front Traversals

Spatial Subdivision

Sorting for Uniform Grid
Sorting for Uniform Grid

- Perspective projection
  - May need to proceed forward for part of the grid and backwards for the other.

K-d Trees

- Alternate splits in each direction
- A subset of BSP-trees
- More efficient storage representation
- Sorting is the same

K-d Trees

- Extend to any dimension $d$
- In 3D, the splits are done with axis-aligned planes.
- Test is simple: is $x$-value (for nodes splitting the $x$-axis) greater than the node value?
- More efficient storage representation
- Sorting is the same
- A subset of BSP-trees

K-d Trees

- A subset of BSP-trees
- Sorting for Uniform Grid
- Perspective projection
  - May need to proceed forward for part of the grid and backwards for the other.

$\Delta \theta \theta$

$\theta \theta$

Sort for Uniform Grid
Binary Space-Partitioning Tree

Given a polygon p

Two lists of polygons:
- those that are behind(p): B
- those that are in-front(p): F

If eye is in-front(p), right display order is B, p, F
Otherwise it is F, p, B

Display a BSP Tree

struct bspnode {
  p: Polygon;  back, front : *bspnode;
} BSPTree;

BSP_display ( bspt )
BSPTree *bspt; {
  if (!bspt) return;
  if (EyeInfrontPoly( bspt->p )) {
    BSP_display(bspt->back); Poly_display(bspt->p); BSP_display(bspt->front);
  } else {
    BSP_display(bspt->front); Poly_display(bspt->p); BSP_display(bspt->back);
  }
}

Generating a BSP Tree

if (polys is empty) then return NULL;
rootp := first polygon in polys;
for each polygon p in the rest of polys do
  if p is infront of rootp then
    add p to the front list
  else if p is in the back of rootp then
    add p to the back list
  else
    split p into a back poly pb and front poly pf
    add pf to the front list
    add pb to the back list
  end_for;
bspt->back := BSP_gentree(back list); bspt->front := BSP_gentree(front list);
bspt->p = rootp; return bspt;

Generating a BSP Tree

If eye is in-front(d), right display order is B, p, F
- those that are in-front(d): F
- those that are behind(d): B
- Two lists of polygons:
  - Given a polygon p

Binary Space-Partitioning Tree