Visibility Computation

aka “The Hidden Surface Problem”

aka “The Visible Surface Problem”!

Assumptions

- Scene is a set of polygons
- Polygons have been perspective transformed
  - z-value has become “pseudo-depth”
    - Increases with distance from viewer
  - See 372 notes

Three classes of algorithm

- Image space methods
  - These answer the question: “What is visible at each pixel in the final image?”
  - e.g. depth buffer
  - Resolution dependent
- Object space methods
  - These answer the question: “What is the exact geometric description of what is visible?”
  - e.g. Weiler-Atherton clipper
  - Resolution independent
- Hybrids
  - Don’t fit either of the above descriptions!
  - e.g. polygon depth-sorting algorithms

The Depth Buffer Algorithm

```
float[][] d = new float[MAX_COLS][MAX_ROWS];
Colour[][] frameBuffer = new Colour[MAX_COLS][MAX_ROWS];
set all values of frameBuffer to background colour
set all values of d to infinity
for each face F in scene {
    for each pixel (x,y) covered by F {
        compute depth = depth of F at (x,y)
        if (depth < d[x][y]) {  // F is closest so far
            frameBuffer[x][y] = colour of F at (x,y)
            d[x][y] = depth
        }
    }
}
```

Notes on Depth Buffering

- Allows rendering of polygonal faces in any order
  - Fits the "pipeline" graphics rendering model well
- Is implemented in hardware on all modern graphics cards
  - Fill rates of up to 4 gigapixels per second (2003)
- Few disadvantages, except:
  - Gives wrong answers if depth resolution insufficient
    - As will ANY method!
  - Doesn't deal with transparency properly
    - Correct answers require depth ordering of faces at each pixel

Notes on Depth Buffering (cont'd)

- BUT depth buffering still requires that each polygon be transformed, lit, scan-converted.
  - Waste of time if polygon is occluded
- For high complexity scenes need to cull polygons *before* they enter the graphics pipeline.
  - Want to cull whole *groups* of polygons

List-Priority Methods

- Methods in which we draw the faces "back to front"
- Classic name: "painters algorithm"
  - Front polygons "painted over" back polygons
- But ….

What do we mean by "back to front"?

Heedless Painters Algorithm

- Three really bad answers:

  "Polygon A is in front of polygon B if its {minimum | maximum | average} depth is less".
- UDOO: sketch situations in which each of these fails.
- Algorithms based on this are called "Heedless Painters Algorithms" [by Hill]:

```
- Calculate depth measure of each face
- Sort faces in back-to-front order according to
  that depth measure
- Draw faces in back-to-front order
```

What "Back to Front" really means

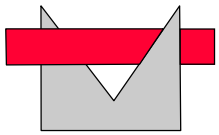
- We want a *partial ordering* with the property that Face A precedes Face B => !occludes(A,B)
  - where *occludes(A,B)* is a predicate that is true if any part of A occludes (i.e. "covers up") any part of B.
  - Heedless painters algorithm is based on false logic like:

```
boolean occludes(Face A, Face B) {
    return centroid(A).depth() < centroid(B).depth(); // "nearer"
}
```

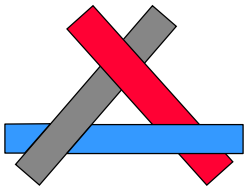
- This is at best a rough heuristic

But such an order may not exist!

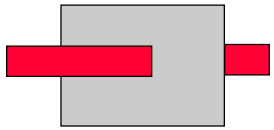
e.g.



concave polygon



cyclic overlap



interpenetration

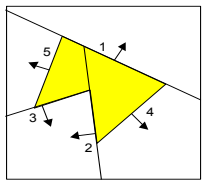
Improved (?) algorithm: Newell, Newell & Sancha Depth Sort

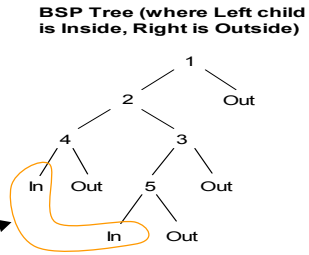
- Use simple depth sort as before.
- Then refine the order using such rules as:
  - if (x,y) bounding boxes of A and B are disjoint then
    occludes(B,A) = occludes(A,B) = false
  - if all vertices of A are in front of plane of B then
    occludes(B,A) = false
  - if the projections of A and B onto the viewplane are disjoint
    occludes(A,B) = occludes(B,A) = false
- But:
  - logic is difficult
  - still have failing cases when we have to clip polygons in two.

BSP Tree Method (depth sort done properly!)

Firstly: what is a BSP tree?

- A BSP tree is a recursive subdivision of space with planes (3D)/lines (2D) at internal nodes
  - Leaf nodes represent convex regions of space
  - Can store various extra info at nodes (depending on application)
- 2D example (unrelated to depth sorting):





Can represent arbitrary polygonal regions as a union of leaf nodes. Can classify any point by pushing it down the tree to a leaf.

Depth sorting with BSP trees

- Idea:
  - Goal is to find an ordering such that no polygon occludes any part of any polygon that comes later in the ordering.
  - Suppose the set of polygons can be divided into two distinct sets by a partitioning plane.
  - Then none of the polygons on the far side of the partitioning plane from the eye can possibly obscure any of the polygons on the near side.
  - Hence can "paint" far side first, then near side
  - BSP-tree allows us to do this recursively
  - Tree is valid for *any* viewpoint

Depth sorting with BSP trees (cont'd)

- How do we construct the BSP tree?
  - Use the planes of polygons within the scene as the partitioning planes
  - Each node in the tree contains (usually) a single polygon and two subtrees.
  - One sub-tree contains polygons that lie entirely behind the plane of the root polygon
  - The other sub-tree contains polygons that lie entirely in front of the plane of the root polygon.
  - In this application, leaves are empty (null) – all the scene polygons are stored in internal nodes.

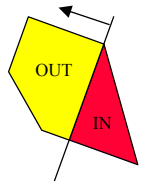
Algorithm to build a 3D BSP tree

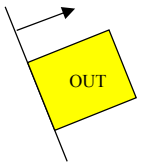
```
class BSPTree {
    Plane plane;                    // The plane that subdivides space
    List inPlanePolys = new List();  // All scene polygons lying on that plane
    BSPTree frontTree;              // A tree describing the world in front of that plane
    BSPTree backTree;               // A tree describing the world behind that plane

    BSPTree (PolygonList polyList) {
    // Constructor, given a non-empty list of scene polygons
        List frontList = new List(), backList = new List();  // Lists of polygons in front and back
        Polygon rootPoly = polyList.head();   // Use the first polygon to subdivide the world
        plane = rootPoly.plane();             // Get its plane as the plane of this node
        inPlanePolys.append(rootPoly);        // Store the polygon itself in this node
        for each polygon in polyList.tail() {   // Sort all the rest of the scene w.r.t. the plane
            if (polygon lies in rootPoly.plane) inPlanePolys.append (polygon);
            else {  polygonPair = clipInTwo(polygon, plane);
                    if (polygonPair.front != null) frontList.append(polygonPair.front);
                    if (polygonPair.backt != null) backList.append(polygonPair.back);
            }
        }
        frontTree = frontList.isEmpty() ? : null : new BSPTree(frontList);
        backTree = backList.isEmpty() ? : null : new BSPTree(backList);
}
```


Clipping a polygon in two with a plane

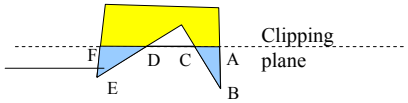
```
PolygonPair ClipInTwo(plane, polygon) {
// Uses the given plane to clip a polygon in 3-space in two.
    Polygon insidePoly = new VertexList();
    Polygon outsidePoly = new VertexList();
    for (each edge of polygon) {
        classify edge endpoints as inside, on, or outside the plane
        if (edge startpoint is on or inside plane)   insidePoly.append(startPoint);
        if (startpoint is on or outside plane)       outsidePoly.append(startPoint);
        if (edge crosses plane) {     // one vertex is inside and the other outside
            calculate crossing point Px;
            tag Px as an on vertex;
            append Px to both insidePoly and outsidePoly;
        }
    }
    if (insidePoly contains only on vertices)   insidePoly  = nil;
    if (outsidePoly contains only on vertices) outsidePoly = nil;
    return new PolygonPair(insidePoly, outsidePoly);
}
```





A Difficulty

Clipping algorithm yields a single polygon ABCDEF



3 ways of dealing with this, in increasing order of complexity are:

- Ignore it, and hope that the scan conversion algorithm does not display the line connecting the two component polygons.
  - This would be the case if, of example, the scan conversion were antialiased
- Split any concave polygons into convex polygons beforehand
- Detect the situation afterwards, and split the components into separate polygons.

Choosing the Root Node

- The above algorithm took the head of the list of polygons as the root.
- Much better to choose the polygon whose plane intersects the fewest other polygons.
  - Less clipping
  - Simpler, shallower tree
    - Some algorithms much faster – O(log n) instead of O(n)
      - UDOO: which ones?!
  - BUT determining such a polygon is very expensive
- Choosing the best of a random sampling of five polygons is almost as good.

Traversing the BSP Tree

```
PolygonList  traverse(BSPTree root, Point3f viewpoint) {
// Traverses the BSP tree w.r.t. given viewpoint, using "otherside first" order.
// Returns a list of all polygons encountered, with the property that no
//  polygon in the list can obscure any part of any other polygon that comes
//  later in the list (when viewed from the viewpoint).
    if (tree == null)  return null;
    else if (viewpoint outside root.plane)
      return traverse(root.backTree, viewpoint) ++  root.inPlanePolys ++
             traverse(root.frontTree, viewpoint);
    else
      return traverse(root.frontTree, viewpoint) ++  root.inPlanePolys ++
          traverse(root.backTree, viewpoint);
}
```

List concatenation operator

Other uses of BSP Trees

- In 3D games like *Quake*
  - BSP tree is used to decompose the scene into a set of disjoint convex regions.
  - Set of all polygons potentially visible from each region is determined (PVS)
    - e.g. polygons inside the convex region plus any regions connected to it by a single open "portal"
  - Only the PVS of the region in which the viewer lies is rendered at each frame
- For set operations on polyhedra
  - e.g. do *intersection* by pushing one polyhedron into the BSP tree of the other, retaining only bits that land in *In* nodes

Other uses of BSP Trees (cont'd)

- For calculating shadows in polyhedral scenes
  - Calculate *shadow volume* as union of shadow volumes of each polygon in scene
- To provide a space-subdivision scheme for use in ray-tracing.

Scan-line Methods

- Obsolescent
- Were used when cost of depth-buffer was excessive
- Only advantage nowadays: can do transparency properly
  - Still used in some modellers for a "quality rendering" pass
    - Much faster than ray tracing
    - But ray tracing is much better quality

Scan-line algorithm (idea only)

- Pre-sort all polygon edges into an edge table with one entry per scan line
  - Associate each edge with its lowest scan line
- Initialize ActiveEdgeList to empty
- for each scan line
  - Delete "expired" edges
  - Add new edges from EdgeTable
  - Compute where each edge crosses scan line
  - Sort edges by crossing point (x value)
  - Fill *spans* of pixels between edge-crossing points, using the colour of whichever polygon is in front over that span

Area Subdivision Methods

Warnock's algorithm

- Divide and conquer method
- Of historical significance only.
- Warnock's algorithm idea:

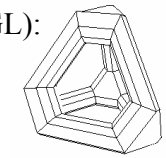
```
display(PolygonList polys, Window win) {
        clip all polys to the window win
        do simple depth sort
        if (polys.length() <= 1)  draw 0 or 1 polys;
        else if (polys.head surrounds window and is in front of all others)
                draw polys.head;
        else {
                subdivide win (into 2 or 4)
                recurse with each subdivided window
        }
}
```

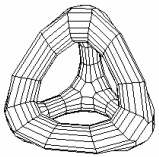
Weiler and Atherton algorithm

- An area subdivision algorithm in object space
- Similar to Warnock, but subdivide along polygon edges
- Have to clip polygons to arbitrary polygonal window
  - "Weiler and Atherton clipper"
  - Hard!
- Output is a list of fully-visible polygon fragments
  - Object space
- Next to impossible to get this working properly!
- Modern approach (?) – use 3D BSP trees to generate "front to back" sequence of output polygons then 2D BSP trees to handle 2D clipping
  - I'm not sure if anyone has actually done this!

"Hidden Line Removal"

- Another classic but rarely-useful algorithm domain
- Nowadays if we want line drawings we usually use hidden *surface* removal techniques, e.g. (OpenGL):
  - Turn on depth buffering
  - Set polygon mode to area fill
  - Draw object's polygons
  - Set polygon mode to line drawing
  - Call *glPolygonOffset* to "pull" output primitives forward at least 1 depth unit
  - Redraw object's polygons





- See http://www.opengl.org/developers/faqs/technical/polygonoffset.htm