Visible Surface Determination

- Problem: Given a set of 3D objects and a viewing specification, determine which lines or surfaces are visible
- This is called visible surface(line) determination, or hidden surface(line) removal
- Two general approaches:
  - Image precision
  - Object precision

Image Precision HSR

- Brute force approach:
  - For each pixel, examine all \( n \) objects, find which is closest
- \( O(np) \) where \( p = \#\text{pixels} \)
- Needs to be redone if the image is resized
Object Precision HSR

- For each object in the world
  - determine the parts of the object that
  - are unobscured by other parts
  - of it or other objects
  - draw those parts in the appropriate color

- Brute force approach:
  - For each object, examine all $n$ objects
- $O(n^2)$
- $n << p$, so this seems good…
- But each computation much more expensive
- Tends to be slower in practice

Coherence

- Slowly varying object properties: *coherence*
  - Object, Face, Edge, Scanline, Area, Depth, Frame
- Exploit coherence whenever possible
  - Reuse calculations
  - Complex math $\rightarrow$ simple differences

Perspective Transformations

- Depth comparisons are typically performed after the perspective transformation
- Since we’re looking down the $z$ axis, two points are now on the same ray if $x_1=x_2$ and $y_1=y_2$
- The perspective transformation preserves
  - Relative depth
  - Straight lines
  - Planes

Extents

- An object’s *screen extent* is an axis aligned rectangle
Bounding Volumes

- Can be used to trivially reject objects
  - If bounding volumes don’t intersect, objects don’t either
  - If bounding volumes do intersect, more work is required
- Bounding volumes don’t have to be rectangular; spheres are common as well
- How “good” is a particular bounding volume?

Back-face detection

Q: When does this method break down?
A: Object not closed. What about interreflections?

Q: How do we test for back-facing polygons?
A: Dot product of the normal and view directions.
A2: Negative ‘z’ coordinate of normal in eye space

2D Backface Example

Red edges are not drawn
Blue edges are drawn

Hierarchical Objects

- Objects that can neatly be divided into hierarchies have implicit extents and object coherence
  - Building
    - Floor 1
    - Floor 2
    - Floor 3
    - Floor 4
  - Room 1
  - Room 2
  - Room 3
- If two hierarchical objects fail to intersect, no need to test their children
Visible Line Determination

- Most VLD algorithms operate in object space
- Input: description of a 3D scene
- Output: a list of visible line segments ready for display

Robert’s Algorithm

- Assume convex polyhedra
- Perform backface culling
- Test each remaining edge against all other polyhedra
  - Use extent testing to trivially reject most polyhedra
  - Each polyhedron either completely obscures the line, or splits it into 1 or 2 lines
- Use linear programming to figure out where a ray from the eye to the line in question grazes each polyhedron

Appel’s Algorithm

- Relax the assumption about convex polyhedra
- Define *quantitative invisibility* (QI) of a point
  - # of front-facing polygons that obscure it
- When a line passes behind a front facing polygon, its QI is increased by one, and when it passes out it is decreased by one

Contour Lines

- *contour line*:
  - An edge shared between a front and back facing polygon
  - The unshared edge of a front facing polygon
- QI changes when lines pass behind a contour edge
- Intersect the contour edge with the triangle formed by the endpoints of the line and the viewpoint
The Z Buffer Algorithm

- Image precision hidden surface removal algorithm
- Very simple to implement
- Use an additional buffer to hold depth values:
  - Render primitives in arbitrary order
  - Record their depths in the depth buffer
  - If the depth of a pixel about to be drawn is greater than what's already there, throw it away

Why Is Z So Popular?

- Advantages
  - Simple to implement in hardware
  - One more interpolator
  - Chunk of memory
  - One more comparison
  - Supports non-polygonal primitives
  - Unlimited scene complexity
  - Depth values can be saved for later use, or other uses

- Disadvantages
  - Extra memory and bandwidth
  - Waste time drawing hidden objects
  - Z precision errors lead to depth aliasing

List Priority Algorithms

- *List priority algorithms* define a visibility ordering for objects, ensuring that a correct picture will result if the objects are drawn in this order
- If no objects overlap in z, just sort on z
- Even if they overlap, we may still be able to sort on z
- Problems for cyclical objects:
**Depth Sorting**

- The *Depth Sort algorithm* performs three conceptual steps:
  - Sort all polygons according to the smallest (farthest) $z$ coordinate of each
  - Resolve any overlapping $z$ extent ambiguities, splitting polygons if necessary
  - Scan convert each polygon in order of ascending $z$ (back to front)
- If we ignore the second step, this is called the *painter's algorithm*

**Resolving Depth Ambiguities**

- Call $P$ the polygon at the end of the list, and $Q$ the polygon about to be added behind $P$
- 5 questions, in order of complexity:
  - Do the polygons’ $x$ extents not overlap?
  - Do the polygons’ $y$ extents not overlap?
  - Is $P$ entirely on the opposite side of $Q$’s plane from the viewpoint?
  - Is $Q$ entirely on the same side of $P$’s plane from the viewpoint?
  - Do the projections of the two polygons onto the $(x,y)$ plane not overlap?

**These Five Questions…**

- If the answer to any of these five questions is true, $P$ does not obscure $Q$, and we can proceed
- If they all fail, assume for the moment that $P$ obscures $Q$, and see if $P$ and $Q$ need to be swapped
  - Ask questions 3 and 4 again, in the other order
- If either of these new tests succeeds, we put $Q$ at the head of the list, and proceed
- If these tests fail also, we split one polygon along the other’s plane and start over

**Some Examples**

- $P$ is entirely on the opposite side of $Q$’s plane from the viewpoint
- $Q$ is entirely on the same side of $P$’s plane as the viewpoint
Binary Space Partition Trees

- *BSP trees* provide a way to efficiently re-render a group of static polygons from any viewpoint
- Trade off large preprocessing cost for a cheap display algorithm
- BSP trees work well when the viewpoint is changing and the scene isn’t
- Incremental updates to the BSP tree are tricky

The Basic Idea

- Find a plane that separates the scene into two disjoint sets of polygons
- Polygons on the same side of that plane as the eye can obscure, but cannot be obscured by, polygons on the other side
- If we can recursively divide each group of polygons until we’re down to one polygon per region of space, we can draw everything from any viewpoint!
- We can represent this partitioning as a binary tree
- Scan convert one branch of the tree before the other branch

Building the BSP Tree

1. Select any polygon
2. Partition the remaining polygons into two groups, based on their position relative to the “splitting” polygon (use the surface normal to classify them)
3. If a polygon is in both groups, split it along the plane of the splitting polygon
4. Repeat on the “front” group
5. Repeat on the “back” group

BSP Tree Example
### BSP Tree Facts

- Starting at the root, we just need to make sure we draw everything in the right order.
  - Draw everything further from the eye
  - Draw the splitting polygon
  - Draw everything closer to the eye
- The BSP tree can also help in clipping
  - If a splitting polygon’s plane does not intersect the viewing volume, then one side of it will be completely clipped
- We really want to select a polygon that will split the least number of children

### Area Subdivision

- Divide the projection plane into smaller areas
- If it is easy to determine what to draw in an area, draw it
- Otherwise, subdivide the area further
- This exploits *area coherence*

### Polygon/Area Relationships

- **Surrounding**
- **Intersecting**
- **Contained**
- **Disjoint**

### Easy Area Decisions

1. All polygons are disjoint from the area. Fill with the background color
2. There is one intersecting or contained polygon. Fill with the background color and then scan convert the polygon into the area
3. There is one surrounding polygon. Fill with the polygon color
4. Multiple polygons surround, intersect, or are contained in the area, but a single surrounding polygon is in front of all the others. Fill with the polygon color
Area Subdivision Example

<p>| | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>