Hidden Surface Removal

- Image space vs. object space
- Efficiency methods (back-face culling, bounding objects, spatial partitioning ...)
- Painter's (reverse) algorithm
- Depth-sort Algorithm
- Z-buffer & A-buffer
- Scanline algorithm
- BSP-Trees

Visibility

polygonal model:
At the most basic level is a collection of vertices

Visibility (2)

- Using a line drawing routine, we can connect the dots
- The model tells which lines to draw to connect which vertices
- Helps, but still ambiguous

Visibility (3)

Visibility helps to resolve this ambiguity

Visibility (4)

And to get a better sense of the 3D model, we can use shading

Visibility (5)

Surfaces may
- be back-facing
- be occluding
- be overlapping
- be intersecting
Visibility (5)

Question:
Which surfaces are visible along direction of projection (parallel) or from center of projection (perspective)?

If two points \( p_1 \) and \( p_2 \) are on the same projector then the closer one obscures the other; otherwise it doesn’t.

Visibility (6)

Are two points on the same projector?

- Parallel projection:
  \[ \text{if} \ (x_1 = x_2 \ \text{and} \ y_1 = y_2) \]
- Perspective projection:
  \[ \text{if} \ (x_1/z_1 = x_2/z_2 \ \text{and} \ y_1/z_1 = y_2/z_2) \]

If Yes, choose the one which is closer.

Visible Surface Algorithms

Two Approaches:

- Image space (image precision) techniques:
  Determine visibility in each pixel

- Object space (object precision) techniques:
  Determine visibility for parts of objects

Image Space

for each pixel in the image {
  determine the object closest to the viewer pierced by the projector through the pixel;
  draw the pixel using the appropriate color;
}

works in the projected space

Object Space

for each object in the world {
  determine those parts of the object whose view is unobstructed by other parts of it or other object
  draw those parts in the appropriate color;
}

works in the model data space

Efficiency

Either approach is typically costly

- GOAL:
  - organize algorithm (objects/pixels) to speed things up
  - costly operations are performed as infrequently and efficiently as possible

Some techniques:
- perspective transformations
- bounding objects
- back-face culling (removal)
- spatial partitioning
- hierarchy
Perspective

- Parallel projection just check \((x, y)\) values
- Perspective needs more work

How can this be made more efficient?

- Bring canonical view-volume into screen coordinates

Perspective (2)

- Apply a transformation to the canonical view volume to transform it to 3D screen coordinates
- Then perform the same comparison as with the parallel projection

Bounding Objects

- Operations with objects are expensive!
- Can we do a quick test with an approximation of the object?
- Answer: yes!
- Technique - approximation through “bounding volumes” or “extents”
- avoid unnecessary clipping
- avoid unnecessary comparisons between objects or their projections

Bounding Objects (2)

Example: Projections

Bounding Objects (3)

- If the extents don’t overlap, the projections don’t need to be tested for clipping with one another

If the extents overlap then either one of the following cases will apply:

- rectangular extents \(\Rightarrow\) bounding boxes, or bounding volumes in 3D
- in general there are many possible bounding boxes
- want to choose one that is efficient for a particular application (i.e. axis-aligned)
- spherical extent \(\Rightarrow\) bounding sphere

Bounding Objects (4)
Bounding Objects (5)
- can be used in a single dimension
- min max testing:
  \[
  \text{if } (z_2_{\text{max}} < z_1_{\text{min}}) \text{ or } (z_1_{\text{max}} < z_2_{\text{min}}) \\
  \text{no overlap between two objects}
  \]

Back-face Culling
- An object is (approximated by) a solid polyhedron ⇒ faces completely enclose its volume
- If viewed from the outside, only its exterior will be visible and we will see only those faces whose surface normals point towards the observer

Back-face Culling (2)
- Testing:
  - Image space: when looking down the –z axis, examine the sign of the z component in the surface normal
  - Object space: A polygon is back-facing to the viewer if $\mathbf{V} \cdot \mathbf{N} > 0$
- can cull about half the polygons in a scene: Why?

Spatial Partitioning
- break a larger problem down into smaller ones:
  - assign objects to spatially coherent groups as a preprocessing step
  - when projecting, only process objects lying within intersecting cells
- 2D: use a grid on the image plane
- 3D: use a grid over object space

Spatial Partitioning (2)
- adaptive partitioning techniques for irregularly distributed objects: size of the cells vary according to the distribution of objects in space
  - quadtrees
  - octrees
  - BSP-trees (binary space partition trees)

Hierarchy
- Use information in the modeling hierarchy to restrict the need for intersection tests at lower levels
  - example: the individual rooms don't need to be considered unless the building and floor are intersected
Hidden Surface Removal Algorithms

- Painter’s Algorithm
- Reverse Painter’s Algorithm
- Depth-Sort Algorithm
- Warnock’s Algorithm
- Z-Buffer
- A-Buffer
- Scan-line Algorithm
- Scan-line Z-Buffer & A-Buffer
- BSP Trees

Painter’s Algorithm

- Simple approach: Draw polygons as an oil painter might.
- Render polygons back to front, painting over previous polygons.
- Assumes that objects don’t overlap in both area and depth

Reverse Painter’s Algorithm

- Motivation:
  - Computing pixel colors may require expensive computations like shading
  - Don’t spend time on drawing pixels that will be overpainted
  - Simply process the objects in the reverse order, front to back
  - Don’t paint over pixels that already been painted

Problems with Painter’s/Reverse

- Sorting objects can be very expensive (depending on the scene)
- Intersecting objects cause problem
- Even non-intersecting polygons may cause problem if they overlap in both area and depth
Depth-Sort Algorithm

- An extension of the painter’s algorithm
- Performs a similar algorithm but attempts to resolve overlapping polygons
- Does some tests and resolves ambiguities caused by overlapping polygons by splitting polygons if necessary

Depth-Sort Algorithm (2)

Test for overlapping polygons:

Let \( P \) be the polygon furthest back in the sorted list.

For each polygon \( Q \) that \( P \) might obscure (have an overlap), do the following tests. As soon as one succeeds, there is no overlap, so stop.

1. Do the \( X \) extents of the polygons not overlap?
2. Do the \( Y \) extents of the polygons not overlap?
3. Is \( P \) entirely on the opposite side of \( Q \)'s plane from the viewpoint?
4. Is \( Q \) entirely on the same side of \( P \)'s plane as the viewpoint?
5. Do the projections of the polygons onto the \( (x, y) \) plane not overlap?

If all 5 tests fail, assume that \( P \) obscures \( Q \) and repeat tests 3 and 4.

If the new tests fail, one of the polygons must be split into multiple polygons and the tests are run again.

Warnock’s Algorithm

- Scheme based on a general approach common in graphics: if the situation is too complex, subdivide
- Start with a root viewport and a list of all primitives
- Then recursively:
  - Clip objects to viewport
  - Fill area if:
    - All surfaces are outside
    - Only one surface intersects area
    - One surface obscures other surfaces within area.
  - Otherwise, subdivide into smaller viewports, distribute primitives among them, and recurse

Warnock’s Algorithm (2)

Problems:

- Hard to embed hierarchical schemes in hardware
- Complex scenes usually have small polygons and high depth complexity
- Most screen regions come down to the single-pixel case

Warnock’s Algorithm (3)

Z-Buffer Algorithm

- Ed Catmull (mid-70s) proposed a radical new approach called z-buffering.
- One of the simplest and most widely used
- The big idea: resolve visibility independently at each pixel
- Store closest depth value at each pixel
- Works on image space and at image precision
Z-Buffer (2)

- Uses two image arrays
  - One image array stores color (in conventional way)
  - Second image array (depth buffer) stores depths associated with each image location
- Depth Buffer has same dimensions as image
- Memory requirements of method not likely to be an issue these days
- Today, done easily in hardware

Z-Buffer (3)

initialize all pixels to background color, initialize depth-buffer to depth of back clipping plane
for each polygon {
    for each pixel in polygon’s projection {
        $pz = \text{depth of projected point}$
        if ($pz \geq \text{stored depth}$) {
            store depth
            draw pixel
        }
    }
}

Z-Buffer (4)

To compute z value, we can exploit the coherence available in planar polygons:

$$\begin{align*}
    z &= \frac{Ax + By + D}{C} \\
    z_0 &= z_{\text{pt}} \\
    z_{\text{np}} &= z_0 + \Delta z
\end{align*}$$

Z-Buffer (5)

If the polygon is not planar, or the surface is not defined, it is possible to interpolate between vertices and then along scanlines

A-Buffer

- Regular Z-Buffer does not support transparency and antialiasing
- A-Buffer is an extension to the z-buffer algorithm
- For each pixel, rather than keeping its current color and depth, keep a linked list of all colors, depths, and transparency values up until the closest completely opaque pixel
- Even more memory intensive
- Allows for proper handling of many advanced techniques (e.g. transparency, antialiasing, …)
- Widely used for high quality rendering
A-Buffer (2)

initialize all pixels linked list with background color,
initialize depth-buffer linked list with depth of back
clipping plane

for each polygon {
  for each pixel in polygon's projection {
    Add colors, depths, and transparency values to the
    proper linked list
  }
}

for each pixel in the screen {
  traverse each list starting at the most distal pixel and
  use the alpha-channel equation to update the pixel color in
  the resulting image
}

Scan-line Algorithm

- extend scan-line algorithm for single polygons to
  multiple polygons
- operate at image precision
- create one scan-line at a time
- maintains intersecting edges list for each scan-line
- determine visibility in each scan-line

Scan-line Algorithm (2)

Two polygons being processed by a scan-line algorithm

Scan-line Algorithm (3)

Maintain three data structures:
- Edge Table (ET): Stores sorted data on all edges for all polygons
- Polygon Table (PT): Stores parameters, attributes, and in-out flag
- Active Edge Table (AET): Stores sorted data for all polygon edges
  intersecting current scan-line

Scan-line Algorithm (4): ET

Entries in edge table are stored in buckets based on
each edge's smaller y coordinate
- Each Edge Table entry contains:
  - y_{max}, upper y co-ordinate of edge
  - x_{min}, lower x co-ordinate of edge
  - x increment, (dx = inverse slope of edge = 1/m), used in
    stepping from one scanline to next (above)
  - The polygon identification number indicating the polygon to
    which the edge belongs.
Scan-line Algorithm (5): ET Example

Scan-line Algorithm (6): PT
- One entry for each Polygon, stores:
  - Polygon ID
  - Polygon plane parameters
  - Attributes (color, shading, etc)
  - In/Out flag initialized to false (will become true when scan-line pixel enters polygon)

Scan-line Algorithm (7): AET
- Active Edge List: Stores data for all polygon edges intersecting current scan-line
- Always keep AET in order of increasing x
- maintain sorted list as scan line proceeds upwards
- active edges only change when a vertex is passed
  - If vertex is an upper vertex then delete 2 edges
  - If vertex is a lower vertex then insert 2 edges
  - If monotone vertex then replace edge

Scan-line Algorithm (8): AET
- Active Edge List: Stores data for all polygon edges intersecting current scan-line
- Always keep AET in order of increasing x
- Maintain sorted list as scan line proceeds upwards
- Active edges only change when a vertex is passed
  - If vertex is an upper vertex then delete 2 edges
  - If vertex is a lower vertex then insert 2 edges
  - If monotone vertex then replace edge

Scan-line Algorithm (9): AET Example
- Scanning scan-line b:
  - Active Edge List contains: E2 - E0 - E5 - E3
  - process E2: invert In/Out flag of Poly 1 (to true)
  - Only one polygon is "in" -> must be visible
  - Put color/shading attributes for Poly 1 into frame
  - Buffer until we reach edge E0 (span coherence)
  - invert In/Out for Poly 1
  - No visible polygons so:
    - Skip to next edge, E5
  - Invert In/Out flag of Poly 2 (to true)
  - Only one polygon is "in" -> must be visible
  - Put color/shading attributes for Poly 2 into frame
  - Buffer until we reach edge E3
  - Invert In/Out of Poly 2
  - No more entries in AEL: finished scan-line b, go to next scan-line (c)
Scan-line Algorithm (11): AET Example

Scanning scan-line c:
- Active Edge List contains E2,E4,E1,E3
- E2 in Poly 1 -> invert In/Out flag of Poly 1 (to true)
- Only one polygon is "in" -> must be visible
- Put color/shading attributes for Poly 1 into frame buffer until we reach edge E4
- E4 in Poly 2 -> invert In/Out for Poly 2
- Now we have two visible polygons so use plane parameters to determine which polygon is closest to Viewer (in this case P2)
- Put color/shading attributes for Poly 2 into frame buffer until we reach edge E1
- E1 in Poly 1 so invert In/Out flag of P1
- Now only one polygon is "in" -> must be visible
- Continue scan converting Poly 2 until edge E3
- Invert In/Out of Poly 2
- No polygons visible
- No more entries in AEL: scan line finished

Scan-line Algorithm (12): Special Cases

- Background color: Pixels without any polygons need to be set, too
- Initialize the frame buffer before scan conversion
- Or place a screen-sized rectangle behind all objects (back clipping plane)

Penetrating Polygons
- Either split objects to avoid piercing
- Or calculate a "false edge" where visibility may change
- Or find the point of penetration on a scan-line as the scan-line is processed

Scan-line Z-Buffer/A-Buffer

- Combines scan-line algorithm with z-buffer/a-buffer
- instead of storing an entire z-buffer what if we store only 1 scan-line at a time to save memory?
- Resolve visibility using z-buffer algorithm
- How does the algorithm have to change?

Properties:
- less memory so allows implementation for very high screen resolution
- good use of edge coherence
- flexible for anti-aliasing (a-buffer)
- not as easy as regular z-buffer to add more primitives

Binary Space Partitioning Trees

- Idea: divide space recursively into half-spaces by choosing splitting planes

BSP Trees (2)

Generating the BSP tree:
- select a polygon as the root (any choice works)
- the root polygon partitions the environment into two half-spaces:
  - one half-space contains all polygons in front of the root polygon (relative to its surface normal)
  - the other contains all polygons behind the root polygon
- any polygon lying on both sides of the root polygon's plane is split by the plan, and its front and back pieces are assigned to the appropriate half-space
- recursively subdivide the two half-spaces in the same manner
- the algorithm terminates when each node contains only a single polygon

BSP Trees (3): Example
BSP Trees (4): Example

[Diagram showing an example of a BSP tree with nodes labeled 1 to 9, indicating front and back regions]

BSP Trees (5): Example

[Diagram showing another example of a BSP tree with nodes labeled 1 to 9, indicating front and back regions]

BSP Trees (6): Rendering

Start from the root node:
- Recursively render right sub-tree: renders polygons behind root (cannot obscure the root polygon)
- Render the root polygon
- Recursively render left sub-tree: renders the polygons in front of the root (can obscure the root polygon)

BSP Tree (6): Rendering

/\square 6
Start from the root node:
- Recursively render right sub-tree: renders polygons behind root (cannot obscure the root polygon)
- Render the root polygon
- Recursively render left sub-tree: renders the polygons in front of the root (can obscure the root polygon)

BSP Trees (7)

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 ) );
}

showBSP( v: Viewer, T: BSPtree )
{
    if T is empty { 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 Trees (8): A Nice Demo!

http://symbolcraft.com/graphics/bsp/index.html

[Diagram showing a BSP tree example with a web link]