Visible-Surface Detection Methods
Chapter 9

Slides are taken from Robert Thomsons notes.
Rendering: creating an image from a model

- The model is a description of three dimensional objects in a strictly defined language or data structure.
- Model would contain geometry, viewpoint, texture and lighting information.
- The image is a digital image or raster graphics image.

A photorealistic rendered image created by using POV-Ray and other programs
Hidden surface removal

- Developing of rendering software and hardware led to geometric models, a call for more realism and therefore lots of polygons.
- All this led to intense interest in finding efficient algorithms for hidden surface removal.
- Some algorithms are more correctly called visible surface algorithms but the two names are used interchangeably.
- Many were generated during the period of intense interest (around late 60s to late 70s).
• Historical note
  – Problem first posed for wireframe rendering

  canonical house

  – Solution called “hidden-line removal”
    • note: lines themselves don’t hide lines. Lines must be edges of opaque surfaces that hide other lines
    • some techniques show hidden line segments as dotted or dashed lines
Visibility of primitives

• We don’t want to waste time rendering primitives which don’t contribute to the final image.
• A scene primitive can be invisible for 3 reasons:
  – Primitive lies outside field of view
  – Primitive is back-facing (under certain conditions)
  – Primitive is occluded by one or more objects nearer the viewer
• How do we remove these efficiently?
• How do we identify these efficiently?
The visibility problem.

Clipping we have covered
Two problems remain:
   Removal of faces facing away from the viewer.
   Removal of faces obscured by closer objects.
What do we want as the output of a hidden-surface procedure? -- There are generally two options.

**Object precision**: The algorithm computes its results to machine precision (the precision used to represent object coordinates).
- The resulting image may be enlarged many times without significant loss of accuracy.
- The output is a set of visible object faces, and the portions of faces that are only partially visible.

**Image precision**: The algorithm computes its results to the precision of a pixel of the image.
- Thus, once the image is generated, any attempt to enlarge some portion of the image will result in reduced resolution.
- In spite of this obvious drawback, the fastest and simplest algorithms usually operate by this approach.
Visible surface algorithms.

Hidden/Visible Surface/Line Elimination/Determination Requirements
  – Handle diverse set of geometric primitives
  – Handle large number of geometric primitives

• Efficiency – it is slow to overwrite pixels, or scan convert things that cannot be seen
• Accuracy - answer should be right, and behave well when the viewpoint moves
• Must have technology that handles large, complex rendering databases
• In many complex worlds, few things are visible
  – How much of the real world can you see at any moment?
• Complexity - object precision visibility may generate many small pieces of polygon
Visible surface algorithms.

• All the spaces in the viewing pipeline maintain depth, so we can work in any space
  – World, View and Canonical Screen spaces might be used
  – Depth can be updated on a per-pixel basis as we scan convert polygons or lines

Classification of Methods
(Sutherland, Sproull, Schumacher, 1974):
• Object Space
  – Geometric calculations involving polygons
  – Floating point precision: Exact
  – Often process scene in object order
• Image Space
  – Visibility at pixel samples
  – Integer precision
  – Often process scene in image order
[9.2] Back-face elimination (culling)

Idea: compare view direction with face normal for each face $F$ of each object

- $N =$ outward-pointing normal of $F$
- $V =$ vector in viewing direction

if $N \cdot V > 0$, throw away the face
A polygon surface with plane parameter $C < 0$ in a right-handed viewing coordinate system is identified as a back face when the viewing direction is along the negative $z_v$ axis.
Back face culling (removal)

- The vertices of polyhedra are oriented in an anticlockwise manner when viewed from outside – surface normal $N$ points out.
- If polygon is in view coordinates
  - Test $z$ component of surface normal. If negative – cull, since normal points away from viewer.
- Otherwise, if $N \cdot V > 0$ we are viewing the back face, so polygon is obscured.
- Only works for convex objects without holes, (ie. closed orientable manifolds).
How do we handle overlapping?

Can we draw the polygons in the “right order” so that we get the correct result (e.g. blue, then green, then pink)?

Is it just a sorting problem?

Yes it is for 2D, but in 3D we can encounter intersecting polygons or groups of non-intersecting polygons which form a cycle where a valid ordering is impossible. Then we must determine which portion of each polygon is visible to the eye.
Painters Algorithm (Object space)

- **Algorithm:**
  - Choose an order for the polygons based on some choice (e.g. depth to a point on the polygon)
  - Render the polygons in that order, deepest one first
- **This renders nearer polygons over further polygons**
- **Difficulty:**
  - works for some important geometries (2.5D - e.g. VLSI)
  - doesn’t work in this form for most geometries - need at least better ways of determining ordering

Which point for choosing ordering?
• Review of Buffers
  – Monitor screen must be frequently refreshed (at least around 60 times/second) to maintain a constant image without artifacts
  – Screen is refreshed one scan line at a time from pixel information held in a refresh or frame buffer
  – Additional buffers can be used to store other pixel information. For example, we will use a z-buffer in which $z$-values (depth of points on a polygon) are stored to do visible surface detection (VSD)
Figure 9-4

Three surfaces overlapping pixel position \((x, y)\) on the view plane. The visible surface, \(S_1\), has the smallest depth value.
[9.3] **z-buffering (or depth buffering).**

Early algorithms were proposed when memory was very expensive (a 512 x 512 framebuffer could cost more than $50 000).

In the mid-70's, Ed Catmull had access to a framebuffer and proposed a radically new visibility algorithm: **z-buffering**.

The idea of z-buffering is to resolve visibility *independently at each pixel*.

• We have seen how to rasterize polygons into a discrete 2D viewport:
Question: what if multiple primitives occupy the same screen pixel?

Which one should be allowed to "paint" the pixel?
Retain depth through transformation to screen space (simply transform $z_{\text{NDC}}$)

- Note that both perspective and viewport transformations preserve depth order.

- In a confusing terminology, an $x,y$ pixel with associated color and depth $z$ has come to be known as a *fragment*.

Augment framebuffer with depth value at each pixel ($b$ bits, can represent $0 .. 2^b - 1$ values).
The Z-buffer (1) (Image Precision)

- For each pixel on screen, have at least two buffers
  - Color buffer stores the current color of each pixel
    - The thing to ultimately display
  - Z-Buffer stores at each pixel the depth of the nearest thing seen so far
    - Also called the depth buffer
- Initialize this buffer to a value corresponding to the furthest point
- As a polygon is filled in, compute the depth value of each pixel that is to be filled
  - if depth < z-buffer depth, fill in pixel color and new depth
  - else disregard
The Z-buffer (2)

• Advantages:
  – Simple
    • A z-buffer is part of what makes a graphics card “3D”
  – Computing the required depth values is simple

• Disadvantages:
  – Over-renders - worthless for very large collections of polygons
  – Depth quantization errors can be annoying
  – Can’t easily do transparency or filtering for anti-aliasing (Requires keeping information about partially covered polygons)
Z-Buffer and Transparency

• Must render in back to front order
• Otherwise, would have to store the opaque polygon most closely behind transparent one

Partially transparent

Opaque

Opaque

Front
Simple Z-buffering

• Simple to include in scanline algorithm.
• Interpolate z during scan conversion.
• Maintain a depth (range) image in the frame buffer (16 or 24 bits common).
• When drawing, compare with the currently stored z value.
• Pixel is given the intensity of nearest polygon.
Implementation.

• Initialise frame buffer to background colour.
• Initialise depth buffer to $z = \text{max. value for far clipping plane}$, i.e. using left-handed coordinate system.
• Need to calculate value for $z$ for each pixel – But only for polygons intersecting that pixel. – Could interpolate from values at vertices.
• Update both frame and depth buffer.
Determining depth.

Use plane equation:

$$Ax + By + Cz + D = 0$$

$$z = \frac{-D - Ax - By}{C}$$

If at \((x, y)\), z value evaluates to \(z_1\), at \((x + \Delta x, y)\), new value of z is:

$$z_1 = z + \frac{A}{C} \Delta x$$

- Only one subtraction needed
- Depth coherence.
Why is z-buffering so popular?

Advantage
• Simple to implement in hardware.
  – Add additional z interpolator for each primitive.
  – Memory for z-buffer is now not expensive
• Diversity of primitives – not just polygons.
• Unlimited scene complexity
• Don’t need to calculate object-object intersections.

Disadvantage
• Extra memory and bandwidth
• Waste time drawing hidden objects
OpenGL Depth Buffer

• OpenGL defines a depth buffer as its visibility algorithm
• The enable depth testing: `glEnable(GL_DEPTH_TEST)`
• To clear the depth buffer:
  `glClear(GL_DEPTH_BUFFER_BIT)`
  – To clear color and depth:
    `glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)`
• The number of bits used for the depth values can be specified (windowing system dependent, and hardware may impose limits based on available memory)
[9.4] A-Buffer Method

An extension of the z-buffer idea
  – developed at LucasFilm Studios

• An antialiasing, area averaging, visibility-detection method

• Accumulation-buffer
  – stores a variety of surface data in addition to depth values
  – allows a pixel color to be computed as a combination of different surface colors for transparency and anti-aliasing effects
The A-buffer (Image Precision)

- Handles transparent surfaces and a form of anti-aliasing
- At each pixel, maintain a *list of polygons sorted by depth*, and a sub-pixel coverage mask for each polygon
  - Sub-pixel mask: Matrix of bits saying which parts of the pixel are covered by the polygon
- Algorithm: When drawing a pixel (first pass):
  - if polygon is opaque and covers pixel, insert into list, removing all polygons farther away
  - if polygon is transparent or only partially covers pixel, insert into list, but don’t remove farther polygons
A-Buffer Method

Each position in the A-buffer has two fields

• Depth field: stores surface data or a pointer
• Surface data field: stores surface data or a pointer
A-Buffer

Surface information in the A-buffer includes

- RGB intensity
- Opacity parameter (% of transparency)
- Depth
- Percent of area coverage
- Surface identifier
- other surface rendering parameters
The A-buffer

• Algorithm: Rendering pass
  – At each pixel, traverse buffer using polygon colors and coverage masks to calculate composite pixel color

• Advantage:
  – Can do more than Z-buffer
    • Anti-aliasing, transparent surfaces without ordering
  – Coverage mask idea can be used in other visibility algorithms

• Disadvantages:
  – Not in hardware, and slow in software
  – Still at heart a z-buffer: Over-rendering and depth quantization problems
[9.5] Scan Line Algorithm (Image Precision)

- Assume polygons do not intersect one another
  - Except maybe at edges or vertices
- Observation: across any given scan line, the visible polygon can change only at an edge
- Algorithm:
  - fill all polygons simultaneously
  - at each scan line, use edge information from polygon tables to find crossing points
  - keep record of current depth at current pixel - use to decide which is in front in filling span
Scan Line Algorithm (2)

Where polygons overlap, draw front polygon.
Scan Line Algorithm (3)

• Advantages:
  – Simple
  – Potentially fewer quantization errors (more bits available for depth)
  – Don’t over-render (each pixel only drawn once)
  – Filter anti-aliasing can be made to work (have information about all polygons at each pixel)

• Disadvantages:
  – Invisible polygons complicate matters
  – Non-intersection criteria may be hard to meet
Scan Line Algorithm

Figure 9-10

Scan lines crossing the view-plane projection of two surfaces, $S_1$ and $S_2$. Dashed lines indicate the boundaries of hidden surface sections.
Scan Line Algorithm

Procedure only works correctly if surfaces do not cut through or cyclically overlap each other. In such cases we could divide the surfaces to eliminate overlaps:

![Diagram showing subdivision of surfaces](image)

Figure 9-11
Intersecting and cyclically overlapping surfaces that alternately obscure one another.
[9.6] Depth Sorting

• The 'painter's algorithm'
• Sort polygons on depth of some point
• Render from back to front (modifying order on the fly)
• Uses both image-space and object-space operations
  • Surfaces are sorted in order of decreasing depth
  • Surfaces are scan-converted in order, starting with the surface of greatest depth
• Sorting carried out in image and object space, scan conversion done in image space
Depth-sort algorithm

- The idea here is to go back to front drawing all the objects into the frame buffer with nearer objects being drawn over top of objects that are further away.

- Simple algorithm:
  - Sort all polygons based on their farthest z coordinate
  - Resolve ambiguities
  - Draw the polygons in order from back to front

- This algorithm would be very simple if the z coordinates of the polygons were guaranteed never to overlap. Unfortunately that is usually not the case, which means that step 2 can be somewhat complex.
Depth-sort algorithm

- First must determine z_extent for each polygon.
Depth-sort algorithm

- Ambiguities arise when the z-extents of two surfaces overlap.
Depth-sort algorithm

- All polygons whose z extents overlap must be tested against each other.
- We start with the furthest polygon and call it P. Polygon P must be compared with every polygon Q whose z extent overlaps P's z extent. 5 comparisons are made. If any comparison is true then P can be written before Q. If at least one comparison is true for each of the Qs then P is drawn and the next polygon from the back is chosen as the new P.
Depth-sort algorithm

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

If all 5 tests fail we quickly check to see if switching P and Q will work. Tests 1, 2, and 5 do not differentiate between P and Q but 3 and 4 do. So we rewrite 3 and 4 as:

3’. Is Q entirely on the opposite side of P's plane from the viewport?
4’. Is P entirely on the same side of Q's plane as the viewport?
• Testing for overlaps: Start drawing when first condition is met:
  – x-extents or y-extents do not overlap
  – S is behind the plane of S’
  – S’ is in front of the plane of S
  – S and S’ do not intersect in the image plane
Testing for overlaps: Start drawing when first condition is met:

(1)

x - extents not overlap?

if the extents do not overlap the test succeeds and we can proceed
Testing for overlaps: Start drawing when first condition is met:

(2)

$y$ - extents not overlap?

if they do, test fails
Testing for overlaps: Start drawing when first condition is met:

(3) Is \( P \) entirely behind the surface \( Q \) relative to the viewing position (i.e., behind \( Q \)'s plane with respect to the viewport)?

Test is true...
Testing for overlaps: Start drawing when first condition is met:

(4) Is Q entirely in front of P's plane relative to the viewing position (i.e., the viewport)?

Test is true...
Testing for overlaps: Start drawing when first condition is met:

(5) Do the projections of $P$ and $Q$ onto the $(x,y)$ plane not overlap?

Test is true...
Depth-sort algorithm

- If all tests fail...
  - ... then reverse P and Q in the list of surfaces sorted by maximum depth
  - set a flag to say that the test has been performed once.
  - If the tests fail a second time, then it is necessary to split the surfaces and repeat the algorithm on the 4 new split surfaces
• Example:
  - We end up processing with order Q2,P1,P2,Q1
Depth sorting

• Advantages:
  – Filter anti-aliasing works fine
    • Composite in back to front order with a sequence of over operations
  – No depth quantization error
    • Depth comparisons carried out in high-precision view space

• Disadvantages:
  – Over-rendering
  – Potentially very large number of splits - $\Omega(n^2)$ fragments from n polygons
Ray casting.

- Sometimes referred to as *Ray-tracing*.
- Involves projecting an imaginary ray from the centre of projection (the viewers eye) through the centre of each pixel into the scene.
Ray Casting Algorithm

- Algorithm:
  - Cast ray from viewpoint through each pixel to find front-most surface

It is like a variation of the depth-buffer algorithm, in which we proceed pixel by pixel instead of proceeding surface by surface.
Computing ray-object intersections.

- The heart of ray tracing.
  - e.g sphere (the easiest).

Express line in parametric form.
\[ x = x_0 + t\Delta x \quad ; \quad y = y + t\Delta y \quad ; \quad z = z + t\Delta z \]

Equation for a sphere:
\[ (x - a)^2 + (y - b)^2 + (z - c)^2 = r^2 \]

Expand, substitute for \(x, y, \& z\).
Gather terms in \(t\).
\[ \Rightarrow \text{Quadratic equation in } t. \]

Solve for \(t\).
- No roots – ray doesn’t intersect.
- 1 root – ray grazes surface.
- 2 roots – ray intersects sphere, (entry and exit)
Ray-polygon intersection.

- Easier
  1) Determine whether ray intersects polygon’s plane.
  2) Determine whether intersection lies within polygon.
- Easiest to determine with an orthographic projection onto the nearest axis and the 2D point-in-polygon test.
Ray casting.

• Easy to implement for a variety of primitives – only need a ray-object intersection function.
• Pixel adopts color of nearest intersection.
• Can draw curves and surfaces exactly – not just triangles
• Can generate new rays inside the scene to correctly handle visibility with reflections, refraction, etc – recursive ray-tracing.
• Can be extended to handle global illumination.
• Can perform area-sampling using ray super-sampling.
• But… too expensive for real-time applications.
Examples of Ray-traced images.
Binary Space Partitioning

- BSP tree: organizes all of space (hence *partition*) into a binary tree
  - Tree gives a rendering order: correctly traversing this tree enumerates objects from back to front
- Tree splits 3D world with planes
  - The world is broken into convex cells
  - Each cell is the intersection of all the half-spaces of splitting planes on tree path to the cell
    - Splitting planes can be arbitrarily oriented
Use of BSPs in VSD

• BSP trees will split up objects so that the painter's algorithm will draw them correctly without the need for a Z-buffer and eliminate the need to sort the objects:
  – *a simple tree traversal will yield the objects in the correct order.*

• It also serves as base for other algorithms, such as visibility lists, which seek to reduce overdraw.

• However, building the tree for scene pre-processing is time consuming, which makes it difficult and inefficient to directly implement moving objects into a BSP tree.

• This is often overcome by using the BSP tree together with a Z-buffer (using the Z-buffer to correctly merge movable objects onto the background scene).
BSP Tree.

- A lot of computation required at start.
  - Try to split polygons along good dividing plane
  - Splitting intersected polygon may be costly
- Inexpensive to check visibility once tree is set up.
- Can be used to generate correct visibility for arbitrary views.

⇒ Efficient when objects don’t change very often in the scene.
BSP-Tree Example

View of scene from above
Building BSP-Trees

- Choose polygon (arbitrary)
- Split its cell using plane on which polygon lies
  - May have to chop polygons in two (Clipping!)
- Continue until each cell contains only one polygon fragment
- Splitting planes could be chosen in other ways, but there is no efficient optimal algorithm for building BSP trees
  - Optimal means minimum number of polygon fragments in a balanced tree
BSP (Binary Space Partitioning) Tree.

• One of “list-priority” class algorithms – returns ordered list of polygon fragments for specified viewpoint (static pre-processing stage).

• Choose polygon arbitrarily

• Divide scene into front (relative to normal) and back half-spaces.

• Split any polygon lying on both sides.

• Choose a polygon from each side – split scene again.

• Recursively divide each side until each node contains only 1 polygon.
BSP Tree.

- Choose polygon arbitrarily

- Divide scene into front (relative to normal) and back half-spaces.

- Split any polygon lying on both sides.

- Choose a polygon from each side – split scene again.

- Recursively divide each side until each node contains only 1 polygon.
BSP Tree.

• Choose polygon arbitrarily

• Divide scene into front (relative to normal) and back half-spaces.

• Split any polygon lying on both sides.

• Choose a polygon from each side – split scene again.

• Recursively divide each side until each node contains only 1 polygon.
BSP Tree.

• Choose polygon arbitrarily

• Divide scene into front (relative to normal) and back half-spaces.

• Split any polygon lying on both sides.

• Choose a polygon from each side – split scene again.

• Recursively divide each side until each node contains only 1 polygon.
BSP Tree.

• Choose polygon arbitrarily

• Divide scene into front (relative to normal) and back half-spaces.

• Split any polygon lying on both sides.

• Choose a polygon from each side – split scene again.

• Recursively divide each side until each node contains only 1 polygon.

Alternate formulation starting at 5
Building a BSP

recursive algorithm:

BSPtree *BSPmaketree(polygon list) {
    choose a polygon as the tree root
    for all other polygons {
        if polygon is in front, add to front list
        if polygon is behind, add to behind list
        else split polygon and add one part to each list
    }
    BSPtree = BSPcombinetree(BSPmaketree(front list),
                            root,
                            BSPmaketree(behind list) )
}
Building Example

- We will build a BSP tree, in 2D, for a 3 room building
  - Ignoring doors
- Splitting edge order is shown
  - “Back” side of edge is side with the number
Building Example (1)

```
3a, 4a, 6

2, 3b, 4b, 5

3b

2

1

6

4b

4a
```
Building Example (2)
Building Example (3)
Building Example (Done)
Displaying a BSP tree.

- Once we have the regions – need priority list
- BSP tree can be traversed to yield a correct priority list for an arbitrary viewpoint.
- Start at root polygon.
  - If viewer is in front half-space, draw polygons behind root first, then the root polygon, then polygons in front.
  - If polygon is on edge – either can be used.
  - Recursively descend the tree.
- If eye is in rear half-space for a polygon – then we can back face cull.
Drawing with a BSP

```c
DrawTree(BSPtree) {
    if (eye is in front of root) {
        DrawTree(BSPtree->behind)
        DrawPoly(BSPtree->root)
        DrawPoly(BSPtree->front)
    } else {
        DrawTree(BSPtree->front)
        DrawPoly(BSPtree->root)
        DrawTree(BSPtree->behind)
    }
}
```
Suppose eye is positioned here

Determine object order - from back to front

Eye is in front of root (3) so draw all behind (3) then (3) then all in front of (3)

When drawing in front of (3) we see that eye is behind the subtree root (2)

So we draw all in front of (2), then (2), then behind (2)

.... drawing order is 4, 5b, 3, 5a, 2, 1

This way the later objects can be drawn over the earlier objects with correct results
Area Subdivision

- Successively divide the view plane into smaller rectangles until each rectangle contains the projection of part of a single visible surface, or contains no surface projections, or rectangle has reduced to single pixel size
Warnock’s Algorithm

- Elegant hybrid of object-space and image-space.
- Uses standard graphics solution: if situation too complex then subdivide problem.
- Start with root window (e.g. entire screen):
  - If this is "simple enough" then scan convert window
  - Else subdivide window as quadtree
  - Recurse until zero or one polygon, or some set depth
  - Depth may be pixel resolution, display nearest polygon
Is an area "simple enough"

- Given an area of interest, classify polygons as
  - Surrounding (completely containing the area)
  - Intersecting (intersecting the area)
  - Contained (completely inside the area)
  - Disjoint (completely outside the area)
- Disjoint polygons can be eliminated
- Intersecting polygons can be split into disjoint and contained polygons
- If there is only one contained polygon, fill area with background, then scan-fill polygon area
- IF there is a single surrounding polygon, and no intersecting or contained polygons, display the surrounding polygons color
- There is a surrounding polygon in front of all other polygons, display the surrounding polygons's color in the area
Warnock’s example
Example: Quake game engine.

- Calculates visibility separately for environment and objects.

- Environment
  - Use portals to determine potentially visible set
  - Use BSP-tree to order polygons front-back.
  - Scan-convert polygons maintaining order.
  - Maintain colour and z buffers.

- Objects.
  - Use Z-buffer from environment stage.
What about other applications?

- **Outdoor environments:**
  - Urban regions, forests, natural scenes in general

- Or very complex assemblies: mechanical CAD parts (Boeing 777 engine block)

- Molecular visualization

Very hard and still not solved problem