HIDDEN SURFACE REMOVAL

An important problem in computer graphics, *hidden surface removal*. We are given collection of objects in 3-space, represented, say, by a set of polygons, and a viewing situation, and we want to render only the visible surfaces.

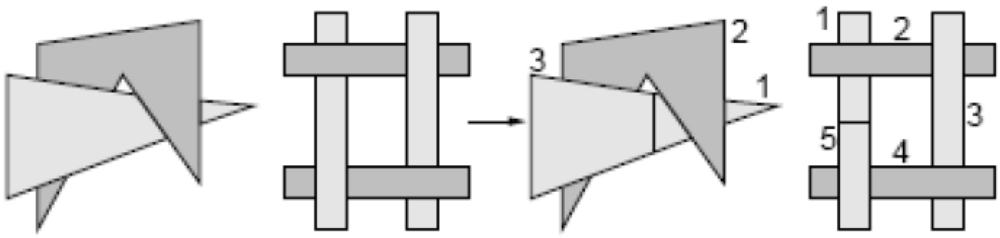
Algorithm Types

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 objects 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.

Back-face Removal:

This is a simple trick, which can eliminate roughly half of the faces from consideration. Assuming that the viewer is never inside any of the objects of the scene, then the back sides of objects are never visible to the viewer, and hence they can be eliminated from consideration.



For each polygonal face, we assume an outward pointing normal has been computed. If this normal is directed away from the viewpoint, that is, if its dot product with a vector directed towards the viewer is negative, then the face can be immediately discarded from consideration.

View Frustum Culling: If a polygon does not lie within the view frustum (recall from the lecture on perspective), that is, the region that is visible to the viewer, then it does not need to be rendered. This automatically eliminates polygons that lie behind the viewer. This amounts to clipping a 2-dimensional polygon against a 3-dimensional frustum. The Liang-Barsky clipping algorithm can be generalized to do this.

Visibility Culling: Sometimes a polygon can be culled because it is "known" that the polygon cannot be visible, based on knowledge of the domain. For example, if you are rendering a room of a building, then it is reasonable to infer that furniture on other floors or in distant rooms on the same floor are not visible. This is the hardest type of culling, because it relies on knowledge of the environment. This information is typically precomputed, based on expert knowledge or complex analysis of the environment.

Depth-Sort Algorithm:

A fairly simple hidden-surface algorithm is based on the principle of painting objects from back to front, so that more distant polygons are overwritten by closer polygons. This is called the *depthsort algorithm*.

This suggests the following algorithm: sort all the polygons according to increasing distance from the viewpoint, and then scan convert them in reverse order (back to front). This is sometimes called the ***painter's algorithm*** because it mimics the way that oil painters usually work (painting the background before the foreground). The painting process involves setting pixels, so the algorithm is an image precision algorithm. There is a very quick-and-dirty technique for sorting polygons, which unfortunately does not generally work.

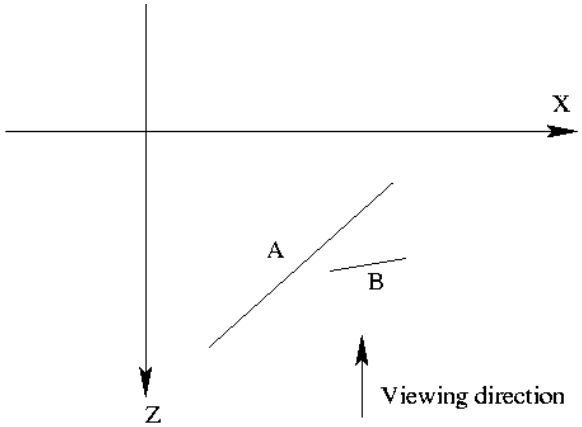
Compute a *representative point* on each polygon (e.g. the centroid or the farthest point to the viewer). Sort the objects by decreasing order of distance from the viewer to the representative point (or using the pseudodepth which we discussed in discussing perspective) and draw the polygons in this order. Unfortunately, just because the representative points are ordered, it does not imply that the entire polygons are ordered.

Worse yet, it may be *impossible* to order polygons so that this type of algorithm will work. The Fig. blow shows such an example, in which the polygons overlap one another cyclically.

The basic algorithm:

1. Sort all polygons in ascending order of maximum z-values.
2. Resolve any ambiguities in this ordering.
3. Scan convert each polygon in the order generated by steps (1) and (2).

The necessity for step (2) can be seen in the simple case shown in following Figure.



Hard cases to depth-sort.

In these cases we may need to *cut* one or more of the polygons into smaller polygons so that the depth order can be uniquely assigned. Also observe that if two polygons do not overlap in *x; y* space, then it does not matter what order they are drawn in.

Here is a snapshot of one step of the depth-sort algorithm. Given any object, define its *z-extents* to be an interval along the *z*-axis defined by the object's minimum and maximum *z*-coordinates.

We begin by sorting the polygons by depth using farthest point as the representative point, as described above. Let's consider the polygon *P* that is currently at the end of the list.

Consider all polygons *Q* whose *z*-extents overlaps *P*'s. This can be done by walking towards the head of the list until finding the first polygon whose maximum *z*-coordinate is less than *P*'s minimum *z*-coordinate. Before drawing *P* we apply the following tests to each of these polygons *Q*. If any answers is "yes", then we can safely draw *P* before *Q*.

1. Are the x-extents of P and Q disjoint?
2. Are the y-extents of P and Q disjoint?
3. Consider the plane containing Q. Does P lie entirely on the opposite side of this plane from the viewer?
4. Consider the plane containing P. Does Q lie entirely on the same side of this plane from the viewer?
5. Are the projections of the polygons onto the view window disjoint?

In the cases of (1) and (2), the order of drawing is arbitrary. In cases (3) and (4) observe that if there is any plane with the property that P lies to one side and Q and the viewer lie to the other side, then P may be drawn before Q. The plane containing P and the plane containing Q are just two convenient planes to test. Observe that tests (1) and (2) are very fast, (3) and (4) are pretty fast, and that (5) can be pretty slow, especially if the polygons are nonconvex.

If all tests fail, then the only way to resolve the situation may be to split one or both of the polygons. Before doing this, we first see whether this can be avoided by putting Q at the end of the list, and then applying the process on Q. To avoid going into infinite loops, we mark each polygon once it is moved to the back of the list.

Once marked, a polygon is never moved to the back again. If a marked polygon fails all the tests, then we need to split. To do this, we use P's plane like a knife to split Q. We then take the resulting pieces of Q, compute the farthest point for each and put them back into the depth sorted list.

In theory this partitioning could generate $O(n^2)$ individual polygons, but in practice the number of polygons is much smaller. The depth-sort algorithm needs no storage other than the frame buffer and a linked list for storing the polygons (and their fragments). However, it suffers from the deficiency that each pixel is written as many times as there are overlapping polygons.

Depth-buffer Algorithm:

The depth-buffer algorithm is one of the simplest and fastest hidden-surface algorithms. Its main drawbacks are that it requires a lot of memory, and that it only produces a result that is accurate to pixel resolution and the resolution of the depth buffer. Thus the result cannot be scaled easily and edges appear jagged (unless some effort is made to remove these effects called "aliasing"). It is also called the z-buffer algorithm because the z-coordinate is used to represent depth. This algorithm assumes that for each pixel we store two pieces of information,

(1) the color of the pixel (as usual), and
(2) the depth of the object that gave rise to this color.
The depth-buffer values are initially set to the maximum possible depth value.
Suppose that we have a k-bit depth buffer, implying that we can store integer depths ranging from 0 to D = 2k − 1. After applying the perspective-with-depth transformation, we know that all depth values have been scaled to the range [−1; 1]. We scale the depth value to the range of the depth-buffer and convert this to an integer, e.g. $b(z + 1)=(2D)c$. If this depth is less than or equal to the depth at this point of the buffer, then we store its RGB value in the color buffer. Otherwise we do nothing.
This algorithm is favored for hardware implementations because it is so simple and essentially reuses the same algorithms needed for basic scan conversion.

Z-Buffering: Algorithm

The easiest way to achieve hidden-surface removal is to use the depth buffer (sometimes called a z-buffer). A depth buffer works by associating a depth, or distance from the viewpoint, with each pixel on the window. Initially, the depth values for all pixels are set to the largest possible distance, and then the objects in the scene are drawn in any order. Graphical calculations in hardware or software convert each surface that's drawn to a set of pixels on the window where the surface will appear if it isn't obscured by something else. In addition, the distance from the eye is computed. With depth buffering enabled, before each pixel is drawn, a comparison is done with the depth value already stored at the pixel.

allocate z-buffer; // Allocate depth buffer →Same size as viewport.
for each pixel (*x*,*y*) // For each pixel in viewport.
writePixel(*x*,*y*,backgrnd); // Initialize color.
writeDepth(*x*,*y*,farPlane); // Initialize depth (*z*) buffer.
for each polygon // Draw each polygon (in any order).
for each pixel (*x*,*y*) in polygon // Rasterize polygon.
***pz* = polygon's *z*-value at (*x*,*y*);** // Interpolate *z*-value at (*x*, *y*).
if (*pz* < z-buffer(x,y)) // If new depth is closer:
writePixel(x,y,color); // Write new (polygon) color.
writeDepth(*x*,*y*,*pz*); // Write new depth.
Note: This assumes' you've negated the z values!right edges.

Advantages:

Easy to implement in hardware (and software!)
Fast with hardware support Fast depth buffer memory
Hardware supported
Process polygons in arbitrary order

Handles polygon interpenetration trivially

Disadvantages:

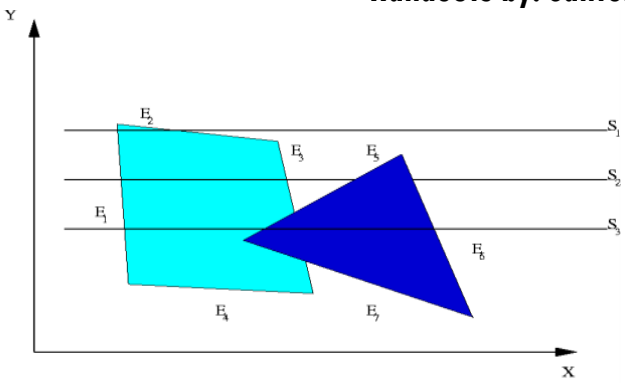
Lots of memory for z-buffer:
Integer depth values
Scan-line algorithm
Prone to aliasing
Super-sampling
Overhead in z-checking: requires fast memory

Scan-Line Algorithm

The scan-line algorithm is another image-space algorithm. It processes the image one scan-line at a time rather than one pixel at a time. By using area coherence of the polygon, the processing efficiency is improved over the pixel-oriented method. Using an active edge table, the scan-line algorithm keeps track of where the projection beam is at any given time during the scan-line sweep. When it enters the projection of a polygon, an IN flag goes on, and the beam switches from the background colour to the colour of the polygon. After the beam leaves the polygon's edge, the colour switches back to background colour. To this point, no depth information need be calculated at all. However, when the scan-line beam finds itself in two or more polygons, it becomes necessary to perform a z-depth sort and select the colour of the nearest polygon as the painting colour.

Accurate bookkeeping is very important for the scan-line algorithm. We assume the scene is defined by at least a polygon table containing the (A, B, C, D) coefficients of the plane of each polygon, intensity/colour information, and pointers to an edge table specifying the bounding lines of the polygon. The edge table contains the coordinates of the two end points, pointers to the polygon table to indicate which polygons the edge bounds, and the inverse slope of the x-y projection of the line for use with scanline algorithms. In addition to these two standard data structures, the scan-line algorithm requires an active edge list that keeps track of which edges a given scan line intersects during its sweep.

The active edge list should be sorted in order of increasing x at the point of intersection with the scan line. The active edge list is dynamic, growing and shrinking as the scan line progresses down the screen.

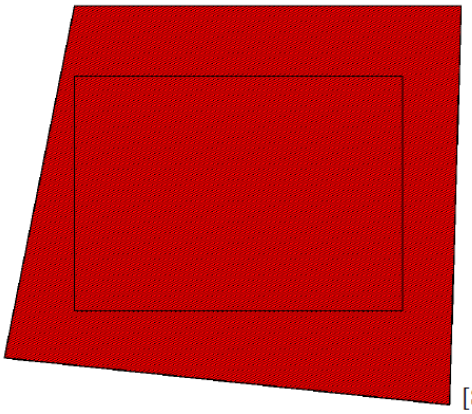


In the following Figure scan-line S1 must deal only with the left-hand object. S2 must plot both objects, but there is no depth conflict. S3 must resolve the relative z-depth of both objects in the region between edge E5 and E3 . The right-hand object appears closer. The active edge list for scan line S1 contains edges E1 and E2 . From the left edge of the viewport to edge E1 , the beam paints the background colour. At edge E1 , the IN flag goes up for the left-hand polygon, and the beam switches to its colour until it crosses edge E2 , at which point the IN flag goes down and the colour returns to background. For scan-line S2 , the active edge list contains E1 , E3 , E5 , and E6 . The IN flag goes up and down twice in sequence during this scan. Each time it goes up pointers identify the appropriate polygon and look up the colour to use in painting the polygon. For scan line S3 , the active edge list contains the same edges as for S2 , but the order is altered, namely E1 , E5 , E3 , E6 . Now the question of relative z-depth first appears. The scan-line algorithm for hidden surface removal is well designed to take advantage of the area coherence of polygons. As long as the active edge list remains constant from one scan to the next, the relative structure and orientation of the polygons painted during that scan does not change. This means that we can "remember" the relative position of overlapping polygons and need not recompute the z-depth when two or more IN flags go on. By taking advantage of this coherence we save a great deal of computation.

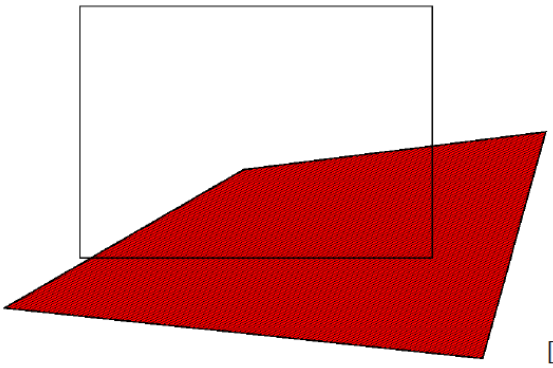
Area Subdivision Algorithm

John Warnock proposed an elegant divide-and-conquer hidden surface algorithm. The algorithm relies on the area coherence of polygons to resolve the visibility of many polygons in image space. Depth sorting is simplified and performed only in those cases involving image-space overlap. Warnock's algorithm classifies polygons with respect to the current viewing window into trivial or nontrivial cases. Trivial cases are easily handled. For nontrivial cases, the current viewing window is recursively divided into four equal sub-windows, each of which is then used for reclassifying remaining polygons. This recursive procedure is continued until all polygons are trivially classified or until the current window reaches the pixel resolution of the screen. At that point the algorithm reverts to a simple z-depth sort of the intersected polygons, and the pixel colour becomes that of the polygon closest to the

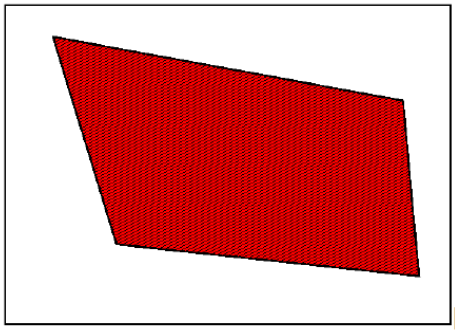
viewing screen All polygons are readily classified with respect to the current window into the four categories illustrated in following Figure.



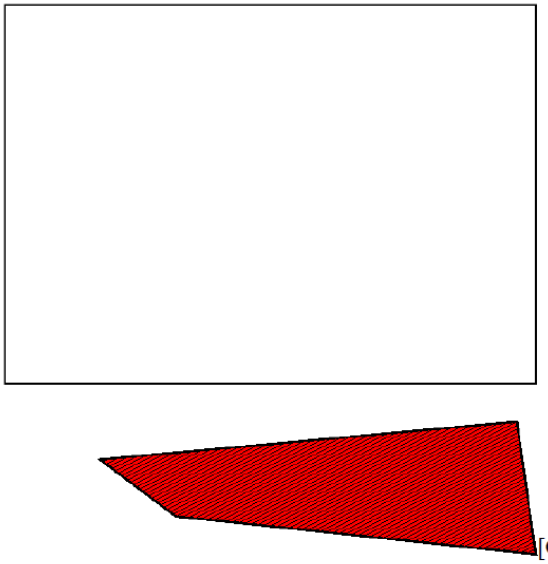
[Surrounding]



[Intersecting]



[Inside]



[Outside]

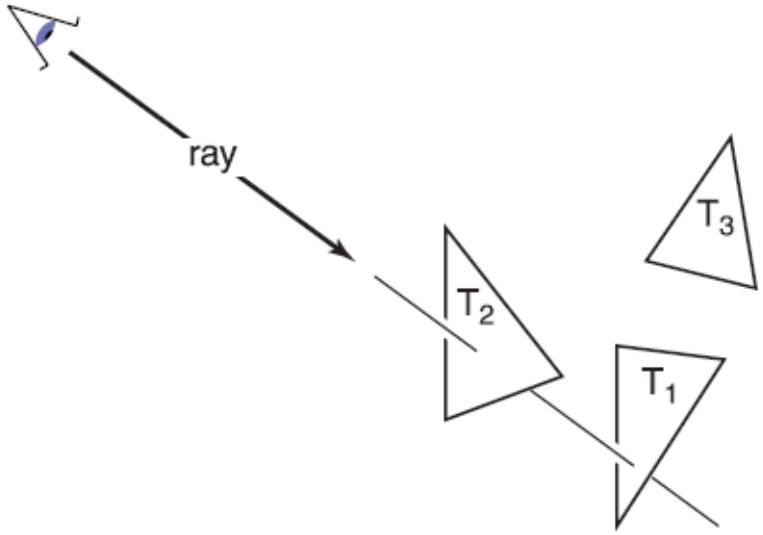
A noteworthy feature of Warnock's algorithm concerns how the divide-and-conquer area subdivision preserves area coherence. That is, all polygons classified as surrounding and outside retain this classification with respect to all sub-windows generated by recursion. This aspect of the algorithm is the basis for its efficiency. The algorithm may be classified as a radix four quick sort. Windows of 1024× 1024 pixels may be resolved to the single pixel level with only ten recursive calls of the algorithm.

While the original Warnock algorithm had the advantages of elegance and simplicity, the performance of the area subdivision technique can be improved with alternative subdivision strategies. Some of these include:

1. Divide the area using an enclosed polygon vertex to set the dividing boundary.
2. Sort polygons by minimum z and use the front polygon as the window boundary.

Ray-Tracing

A ray tracer works by computing one pixel at a time, and for each pixel the basic task is to find the object that is seen at that pixel's position in the image. Each pixel "looks" in a different direction, and any object that is seen by a pixel must intersect the *viewing ray*, a line that emanates from the viewpoint in the direction that pixel is looking. The particular object we want is the one that intersects the viewing ray nearest the camera, since it blocks the view of any other objects behind it. Once that object is found, a *shading* computation uses the intersection point, surface normal, and other information (depending on the desired type of rendering) to determine the color of the pixel. This is shown in Figure



Where the ray intersects two triangles, but only the first triangle hit, *T*2, is shaded.

A basic ray tracer therefore has three parts:

Ray generation, which computes the origin and direction of each pixel's viewing ray based on the camera geometry;

R***ay intersection***, which finds the closest object intersecting the viewing ray;

Ray shading, which computes the pixel color based on the results of ray intersection.

The structure of the basic ray tracing program is:

for *each pixel* ***do***
compute viewing ray
find first object hit by ray and its surface normal ***n***
set pixel color to value computed from hit point, light, and ***n***

The basic methods for ray generation, ray intersection, and shading, that are sufficient for implementing a simple demonstration ray tracer.

BSP Trees

A Binary Space Partitioning (BSP) tree represents a recursive, hierarchical partitioning, or subdivision, of n-dimensional space into convex sub-spaces. BSP tree construction is a process which takes a subspace and partitions it by any hyperplane that intersects the interior of that subspace. The result is two new subspaces that can be further partitioned by recursive application of the method.

A "hyperplane" in n-dimensional space is an n-1-dimensional object which can be used to divide the space into two half-spaces. For example, in three-dimensional space, the "hyperplane" is a plane. In two-dimensional space, a line is used. BSP trees are extremely versatile, because they are powerful sorting and classification structures. They have uses ranging from hidden surface removal and ray tracing hierarchies to solid modelling and robot motion planning.

BSP trees are closely related to Quadtrees and Octrees. Quadtrees and Octrees are space partitioning trees which recursively divide sub-spaces into four and eight new sub-spaces, respectively. A BSP Tree can be used to simulate both of these structures.

BSP Tree Construction

Given a set of polygons in three-dimensional space, we want to build a BSP tree which contains all of the polygons. For now, we will ignore the question of how the resulting tree is going to be used. The algorithm to build a BSP tree is very simple:

1. Select a partition plane.
2. Partition the set of polygons with the plane.
3. Recurse with each of the two new sets.

The choice of partition plane depends on how the tree will be used, and what sort of efficiency criteria you have for the construction. For some purposes, it is appropriate to choose the partition plane from the input set of polygons. Other applications may benefit more from axis aligned orthogonal partitions If all of the points lie to one side of the plane, then the entire polygon is on that side and does not need to be split.

If some points lie on both sides of the plane, then the polygon is split into two or more pieces. The basic algorithm is to loop across all the edges of the polygon and find those for which one vertex is on each side of the partition plane. The intersection points of these edges and the plane are computed, and those points are used as new vertices for the resulting pieces.

Classifying a point with respect to a plane is done by passing the (x,y,z) values of the point into the plane equation, $ax+by+cz+d=0$. The result of this operation is the distance from the plane to the point along the plane's normal vector. It will be positive if the point is on the side of the plane pointed to by the normal vector, negative otherwise. If the result is 0, the point is on the plane. For those not familiar with the plane equation, the values a , b , and c are the coordinate values of the normal vector. The value of d can be calculated by substituting a point known to be on the plane for x , y , and z into the plane equation.

Convex polygons are generally easier to deal with in BSP tree construction than concave ones, because splitting them with a plane always results in exactly two convex pieces. Furthermore, the algorithm for splitting convex polygons is straightforward and robust.

Hidden surface removal

Probably the most common application of BSP trees is hidden surface removal in three dimensions. BSP trees provide an elegant, efficient method for sorting polygons via a depth first tree walk. This fact can be exploited in a back to front "painter's algorithm" approach to the visible surface problem, or a front to back scan-line approach.

BSP trees are well suited to interactive display of static (not moving) geometry because the tree can be constructed during a preprocessing stage. Then the display from any arbitrary viewpoint can be done in linear time. One reason that BSP trees are so elegant for the painter's algorithm is that the splitting of difficult polygons is an automatic part of tree construction. To draw the contents of the tree, perform a back to front tree traversal. Begin at the root node and classify the eye point with respect to its partition plane.

Draw the subtree at the far child from the eye, then draw the polygons in this node, then draw the near subtree. Repeat this procedure recursively for each subtree.

When building a BSP tree specifically for hidden surface removal, the partition planes are usually chosen from the input polygon set. However, any arbitrary plane can be used if there are no intersecting or concave polygons.

Efficient BSP Trees

The upper bound on space and time complexity is $O(n^2)$ for n polygons. The expected case is $O(n)$ for most models. Construction of an optimal tree is an NP-complete problem. There are several strategies for making a BSP tree more efficient.

Minimizing splitting:

An obvious problem with BSP trees is that polygons get split during the construction phase, which results in a larger number of polygons. Larger numbers of polygons translate into larger storage requirements and longer tree traversal times. This is undesirable in all applications of BSP trees, so some scheme for minimizing splitting will improve tree performance.

Tree balancing:

Tree balancing is important for uses which perform spatial classification of points, lines, and surfaces. This includes ray tracing and solid modelling. Tree balancing is important for these applications because the time complexity for classification is based on the depth of the tree. Unbalanced trees have deeper subtrees, and therefore have a worse worst case. For the hidden surface problem, balancing doesn't significantly affect runtime.