Hidden Surface Removal/ visual surface Detection

When you draw a scene composed of three-dimensional objects, some of them might obscure all or parts of others. Changing your viewpoint can change the obscuring relationship. For example, if you view the scene from the opposite direction, any object that was previously in front of another is now behind it. To draw a realistic scene, these obscuring relationships must be maintained.

Culling Polygon Faces

When drawing a closed polyhedral object, those faces that are facing away from the viewer will be obscured and hence will not be visible. On average half the faces in the object will fall into this category. If these faces can be quickly and easily identified then approximately half the effort in hidden surface removal and rendering of these polygons can be avoided.

Most objects specify their polygons in such a way that the normal vector points toward the outside of the object. Remember there are two directions that the normal can point (in or out), and one can be selected by choosing a particular ordering for the vertices of the polygon (clockwise or anti-clockwise).

Hidden surface Removal

The elimination of parts of solid objects that are obscured by others is called hidden-surface removal. (Hidden-line removal, which does the same job for objects represented as wireframe skeletons, is a bit trickier.
Methods can be categorized as:

Object Space Methods

These methods examine objects, faces, edges etc. to determine which are visible. The complexity depends upon the number of faces, edges etc. in all the objects.

Image Space Methods

These methods examine each pixel in the image to determine which face of which object should be displayed at that pixel. The complexity depends upon the number of faces and the number of pixels to be considered.

Z Buffer

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.

If the new pixel is closer to the eye than what's there, the new pixel's colour and depth values replace those that are currently written into the pixel. If the new pixel's depth is greater than what's currently there, the new pixel would be obscured, and the colour and depth information for the incoming pixel is discarded.

Since information is discarded rather than used for drawing, hidden-surface removal can increase your performance.

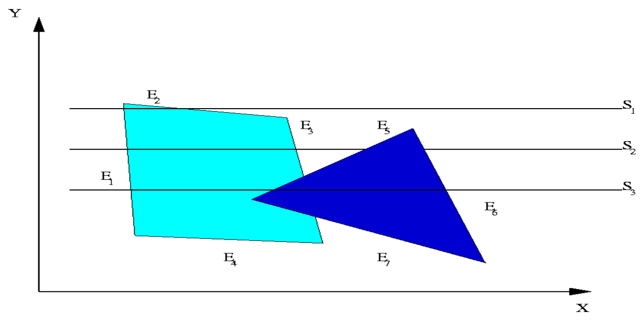
Scan-Line Algorithm

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 scan-line 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 S_1 must deal only with the left-hand object. S_2 must plot both objects, but there is no depth conflict. S_3 must resolve the relative z-depth of both objects in the region between edge E_5 and E_3 . The right-hand object appears closer.



The active edge list for scan line S_1 contains edges E_1 and E_2 . From the left edge of the viewport to edge E_1 , the beam paints the background colour. At edge E_1 , the IN flag goes up for the left-hand polygon, and the beam switches to its colour until it crosses edge E_2 , at which point the IN flag goes down and the colour returns to background.

For scan-line S_2 , the active edge list contains E_1 , E_3 , E_5 , and E_6 . 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 S_3 , the active edge list contains the same edges as for S_2 , but the order is altered, namely E_1 , E_5 , E_3 , E_6 . Now the question of relative z-depth first appears. The IN flag goes up once when we cross E_1 and again when we cross E_5 , indicating that the projector is piercing two polygons. Now the coefficients of each plane and the (x,y) of the E_5 edge are used to compute the depth of both planes. In the example shown the z-depth of the right-hand plane was smaller, indicating it is closer to the screen. Therefore the painting colour switches to the right-hand polygon colour which it keeps until edge E_6 .

Note that the technique is readily extended to three or more overlapping polygons and that the relative depths of overlapping polygons must be calculated only when the IN flag goes up for a new polygon. Since this occurrence is far less frequent than the number of pixels per scan line, the scan-line algorithm is more computationally efficient than the z-buffer algorithm.
The scan-line hidden surface removal algorithm can be summarized as:

1. Establish the necessary data structures.
    a. Polygon table with coefficients, colour, and edge pointers.
    b. Edge table with line end points, inverse slope, and polygon pointers.
    c. Active edge list, sorted in order of increasing x.
    d. An IN flag for each polygon.
2. Repeat for all scan lines:
    a. Update active edge list by sorting edge table against scan line y value.
    b. Scan across, using background colour, until an IN flag goes on.
    c. When 1 polygon flag is on for surface P , enter intensity (colour) into refresh buffer.
    d. When 2 or more surface flags are on, do depth sort and use intensity I_n for surface n with minimum z-depth.
    e. Use coherence of planes to repeat for next scan line.

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.

Painter's algorithm

The idea behind the Painter's algorithm is to draw polygons far away from the eye first, followed by drawing those that are close to the eye. Hidden surfaces will be written over in the image as the surfaces that obscure them are drawn.

The concept is to map the objects of our scene from the world model to the screen somewhat like an artist creating an oil painting. First she paints the entire canvas with a background colour. Next, she adds the more distant objects such as mountains, fields, and trees. Finally, she creates the foreground with "near" objects to complete the painting. Our approach will be identical. First we sort the polygons according to their z-depth and then paint them to the screen, starting with the far faces and finishing with the near faces.

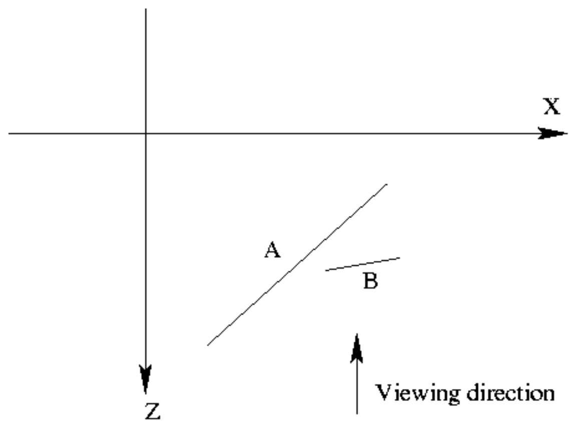
The algorithm initially sorts the faces in the object into back to front order. The faces are then scan converted in this order onto the screen. Thus a face near the front will obscure a face at the back by overwriting it at any points where their projections overlap. This accomplishes hidden-surface removal without any complex intersection calculations between the two projected faces.

The algorithm is a hybrid algorithm in that it sorts in object space and does the final rendering in image space.

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.



B precedes A in order of maximum z but A should precede B in writing order.

At step (2) the ordering produced by (1) must be confirmed. This is done by making more precise comparisons between polygons whose z-extents overlap.

Assume that polygon P is currently at the head of the sorted list, before scan converting it to the frame-buffer it is tested against each polygon Q whose z-extent overlaps that of P. The following tests of increasing complexity are then carried out:

1. If the x-extents of P and Q do not overlap then the polygons do not overlap, hence their ordering is immaterial.
2. If the y-extents of P and Q do not overlap then the polygons do not overlap, hence their ordering is immaterial.
3. If P is wholly on the far away side of Q then P is written before Q.
4. If Q is wholly on the viewing side of P then P is written before Q.
5. If the projections of P and Q do not overlap then the order P and Q in the list is immaterial.

If any of these tests are satisfied then the ordering of P and Q need not be changed. However if all five tests fail then it is assumed that P obscures Q and they are interchanged in the list. To avoid looping in the case where P and Q inter-penetrate Q must be marked as having been moved in the list.

When polygon Q which has been marked fails all tests again when tested against P then it must be split into two polygons and each of these polygons treated separately. Q is split in the plane of P.

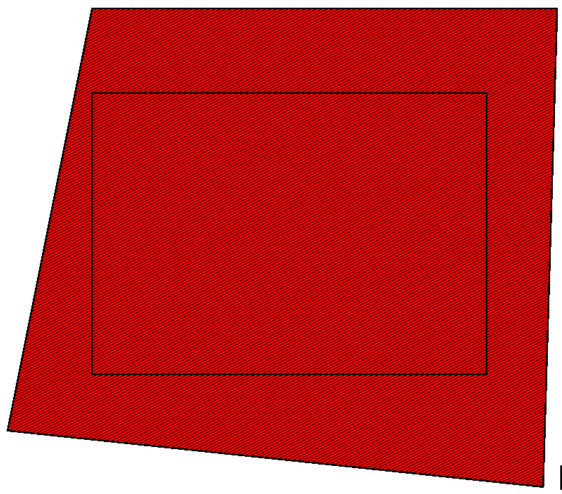
Often the last test will not be done because it can be very complex for general polygons.

Warnock's Area Subdivision Algorithm

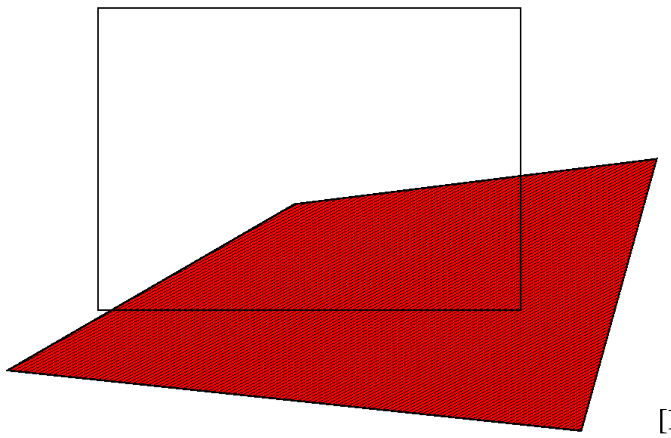
John Warnock proposed an elegant divide-and-conquer hidden surface algorithm. The algorithm relies on the area coherence of polygons to resolve the visibility of many polygons in image space. Depth sorting is simplified and performed only in those cases involving image-space overlap.

Warnock's algorithm classifies polygons with respect to the current viewing window into trivial or non-trivial cases. Trivial cases are easily handled. For nontrivial cases, the current viewing window is recursively divided into four equal subwindows, 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.

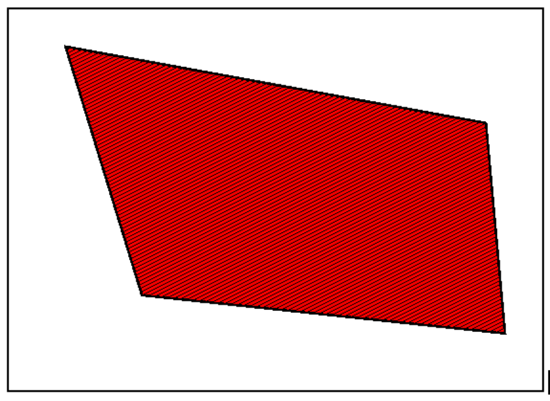
All polygons are readily classified with respect to the current window into the four categories illustrated in following Figure.



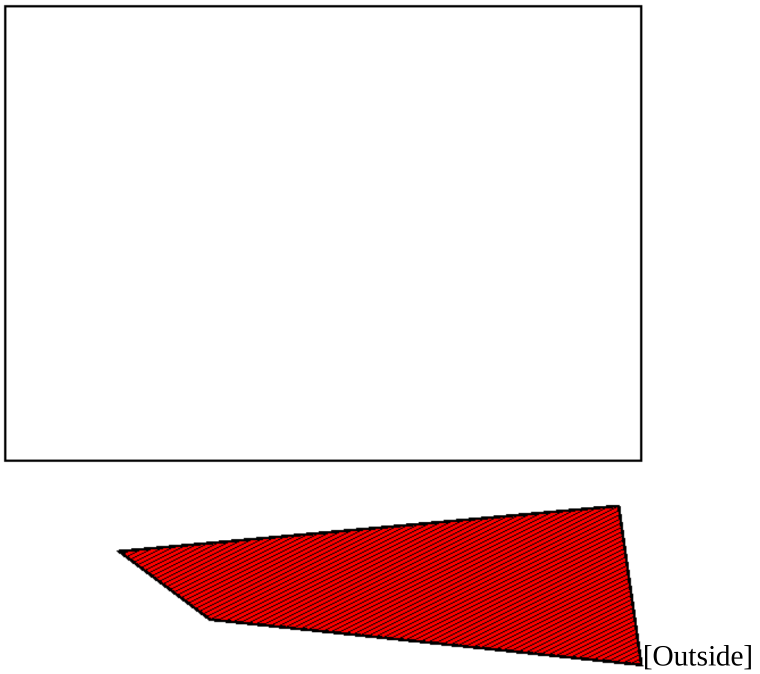
[Surrounding]



[Intersecting]



[Inside]



The classification scheme is used to identify certain trivial cases that are easily handled. These "easy choice tests" and the resulting actions include:
For polygons outside from the window, set the colour/intensity of the window equal to the background colour.

1. There is only one inside or intersecting polygon. Fill the window area with the background colour then render the polygon.
2. There is only one surrounding polygon. Fill the window with the polygon's colour.
3. If more than one polygon intersects, is inside, or surrounds, and at least one is a surrounding polygon.

    a. Is one surrounding polygon, P , in front of all others? If so, paint window with the colour of P . The test is: Calculate the z-depths for each polygon plane at the corners of the current window. If all four z-depths of the P plane are all smaller than any z-depths of other polygons in the window, then $P$is in front.

If the easy choice tests do not classify the polygon configuration into one of these four trivial action cases, the algorithm recurs by dividing the current window into four equal subwindows. Rather than revert to the complex geometrical tests of the Painter's algorithm, Warnock's algorithm simply makes the easy choices and invokes recursion for non-trivial cases.

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 subwindows 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.

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 sub-spaces 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 modeling 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. In any case, you want to evaluate how your choice will affect the results. It is desirable to have a balanced tree, where each leaf contains roughly the same number of polygons. However, there is some cost in achieving this. If a polygon happens to span the partition plane, it will be split into two or more pieces. A poor choice of the partition plane can result in many such splits, and a marked increase in the number of polygons. Usually there will be some trade off between a well balanced tree and a large number of splits.

Partitioning a set of polygons with a plane is done by classifying each member of the set with respect to the plane. If a polygon lies entirely to one side or the other of the plane, then it is not modified, and is added to the partition set for the side that it is on. If a polygon spans the plane, it is split into two or more pieces and the resulting parts are added to the sets associated with either side as appropriate.

The decision to terminate tree construction is, again, a matter of the specific application. Some methods terminate when the number of polygons in a leaf node is below a maximum value. Other methods continue until every polygon is placed in an internal node. Another criteria is a maximum tree depth.

Partitioning a polygon with a plane is a matter of determining which side of the plane the polygon is on. This is referred to as a front/back test, and is performed by testing each point in the polygon against the plane. 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.

If the eye point is classified as being on the partition plane, the drawing order is unclear. This is not a problem if the rendering routine is smart enough to not draw polygons that are not within the viewing frustum. It is possible to substantially improve the algorithm by including the viewing direction vector in the computation. You can determine that entire subtrees are behind the viewer by comparing the view vector to the partition plane normal vector.

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.