from Computer Graphics, Vol. 14 No. 3 (July, 1980)
SIGGRAPH '80 Conference Proceedings

ON VISIBLE SURFACE GENERATION BY A PRIORI TREE STRUCTURES*

Henry Fuchs
University of North Carolina at Chapel Hill
Zvi M. Kedem
The University of Texas at Dallas
Bruce F. Naylor
The University of Texas at Dallas

ABSTRACT

This paper describes a new algorithm for solving the hidden surface (or line) problem, to more rapidly generate realistic images of 3-D scenes composed of polygons, and presents the development of theoretical foundations in the area as well as additional related algorithms. As in many applications the environment to be displayed consists of polygons many of whose relative geometric relations are static, we attempt to capitalize on this by preprocessing the environment's database so as to decrease the run-time computations required to generate a scene. This preprocessing is based on generating a "binary space partitioning" tree whose inorder traversal of visibility priority at run-time will produce a linear order, dependent upon the viewing position, on (parts of) the polygons, which can then be used to easily solve the hidden surface problem. In the application where the entire environment is static with only the viewing-position changing, as is common in simulation, the results presented will be sufficient to solve completely the hidden surface problem.

INTRODUCTION

One of the long-term goals of computer graphics has been, and continues to be, the rapid, possibly real-time generation of realistic images of simulated 3-D environments. "Real-time," in current practice, has come to mean creating an image in 1/30 of a second--fast enough to continually generate images on a video monitor. With this fast image generation, there is no discernable delay between specifying parameters for an image (using knobs, switches, or cockpit controls) and the image's appearance on the monitor's screen. Systems which can achieve this kind of performance are currently so expensive ($1M and up) that very few users can afford them. Users with more modest budgets have to be content with severely more limited performance--either a lower quality image ("wire frame" instead of solid-object modeling) or slower interaction (a time lag of several seconds to several minutes for a solid-object image).

*This research was partially supported by NSF under Grants MCS79-00168 and MCS79-02593, and was facilitated by the use of Theory Net (NSF Grant MCS78-01689).

PROBLEM STATEMENT

The problem to be solved is:

Given

1. a data base describing a 3-D environment in terms of, say, a few thousands tiles (polygons) describing the surfaces of the various objects in the environment, one or more light sources and

2. the (simulated) viewing position, orientation , and field of view,

Generate a color video image of the environment as it would appear from the given viewing position and orientation.

This image generation task consists, broadly, of the following three steps:

1. transforming points into the image space,

2. clipping away polygons outside the field of view,

3. generating the image from the polygons that remain. Generating the image consists of determining the proper color (intensities of red, green, and blue) for each of perhaps 250,000 picture elements (approximately 500 rows of dots, with 500 dots in each row). For each picture element ("pixel"),

   a) find the polygon closest to the viewing position. (This will be the visible polygon at this pixel, the

polygon which obstructs all others.)

b) given the visible polygon, determine the proper color for the pixel by evaluating a lighting model formula, see, e.g., (Newman and Sproull, 1979).

PROPOSED SOLUTION

Since current moderately-priced ($40-80k) real-time line-drawing systems (e.g. Evans and Sutherland Picture System 2, Vector General Model 3404) can easily perform steps 1 and 2, we shall concentrate on solutions to step 3. New solutions to this remaining step could then be combined with already available solutions to produce a complete system. Further, we believe step 3b can be effectively solved by distributing the individual pixel calculations among many small processors (Fuchs and Johnson, 1979). We thus concentrate in this paper on step 3a, determining the visible polygon at each pixel.

We propose an alternative solution to an approach first utilized a decade ago (Schumaker et al., 1969) but due to a few difficulties, not widely exploited. The general approach is based on the observation that in a wide variety of applications many images are generated of the same environment with only a change in the viewing position and orientation, but no change in the environment. For example, pilots in a simulator may practice many different landings at the same airport, with each landing generating thousands of new images. Similarly, an architect may "walk" through a newly designed house or housing development; a biochemist may rotate or move about a complicated protein molecule. To take advantage of such static environments, the data base is preprocessed once (for all time, or until the data base is changed) before any images are generated. In this preprocessing stage, certain geometric relationships are extracted which can then be used to speed up the visible polygon determination for each pixel, for all possible images.

It is important to note that although the development here is given only rigid objects and environments, these concepts can be extended to handle environments with some moving objects.

SOLUTION OVERVIEW

In order to determine the visible surface at each pixel, traditionally the distance from the viewing position to each polygon which maps onto that pixel is calculated. Most methods attempt to minimize the number of polygons to be so considered. Our approach eliminates these distance calculations entirely. Rather, it transforms the polygonal data base (splitting polygons when necessary) into a binary tree which can be traversed at image generation time to yield a visibility priority value for each polygon. These visibility priorities are assigned in such a way that at each pixel the closest polygon to the viewing position will be the one with the highest visibility priority. As we shall see, the visibility priorities are a function of the viewing position; they remain constant for all pixels in every image generated from the same viewing position. In cases for which these visibility priority numbers cannot be assigned to the original polygons (see, e.g., fig. 6) and some polygons need to be split, the splitting is done only once -- during the preprocessing phase -- never at image generation time.

PREPROCESSING PHASE

Let us now consider the set of polygons $P = \{p_1, p_2, \ldots, p_n\}$ which define the 3-D environment. Choose an arbitrary (for now) polygon p_k from this set. We note that the plane in which this polygon lies partitions the rest of 3-space into two half-spaces--call these S_k and $S_{\bar{k}}$. The two half-spaces are identified with the positive and negative sides of the polygon p_k. If p_k was defined with a "front" side, then that side is considered as the positive one; otherwise, one of the sides is arbitrarily chosen at this time to be the positive side.

What can we say about visibility priorities of these polygons? We know that if the viewing position is in one half-space, say in S_k, that no polygon within $S_{\bar{k}}$ can obstruct either polygon p_k or any polygon in S_k (see figure. 1).

Therefore, we split each of the polygons in $P - \{p_k\}$ along the plane of p_k, putting the polygons (or parts of them) which lie in S_k into one set and polygons which lie in $S_{\bar{k}}$ into another set. (Polygons coplanar with p_k can be put into either set.) We can represent the results of this splitting process by a binary tree (we'll call it a Binary Space Partitioning, or "BSP" tree) in which the root contains p_k and each branch's subtree contains the set of polygons associated with one of the half-spaces (Fig. 2).

We next consider one of the two new sets of polygons, say the one in S_k. We remove a polygon, say p_i and split the remaining polygons in S along the plane of p_i, putting those polygons (or parts thereof) lying on the positive side in one

set ($S_{k,j}$) and those lying on the negative side in anotner set ($S_{k,r}$). The overall tree after this step is shown in Fig. 3.

To complete the construction of the BSP tree we continue splitting sets until no non-null sets remain.

The entire preprocessing phase, then, consists of transforming the entire polygonal data base into a BSP tree by the following recursive procedure (stated in a simple pseudo-PASCAL):

```
PROC Make_tree(pl:polygon_list): tree;

BEGIN
k=Select_polygon (pl);
pos_list := null; neg_list := null;
/* pos refers to positive parts
   neg refers to negative parts */
FOR i := 1 TO Size_of(pl) DO
  BEGIN
  IF i <> k THEN
   BEGIN
   Split_polygon(pl[i], pl[k],
                 pos_parts,neg_parts);
   Add (pos_parts, pos_list);
   Add (neg_parts, neg_list)
   END
  END;
RETURN   Combine_tree(Make_tree(pos_list),
                      pl(k),
                      Make_tree(neg_list))

END;
```

We note again that this process is only performed once for all possible images from all viewing positions; the tree remains valid as long as the scene doesn't change.

IMAGE GENERATION PHASE

Calculating the visibility priorities, once the viewing position is known, is a variant of an in-order traversal of the environment's BSP tree (traverse one subtree, visit the root, traverse the other subtree). We wish, for example, to have an order of traversal that visits the polygons from those farthest away to those closest to the current viewing position. At any given node, there are two possibilities: positive side subtree, node, negative side subtree or negative side subtree, node, positive side subtree. We choose one of these two orderings based on the relationship of the current viewing position to the node's polygon. Specifically, we are interested in the side (positive or negative) of the node's polygon where the current viewing position is located. Let's call the two sides the "containing" side and the "other" side. The traversal for a back-to-front ordering is 1) the "other" side, 2) the node, and 3) the "containing" side. (This side-or-node-polygon determination is, of course, just a check of the sign of the z component of the node polygon's normal vector after the usual transformation to the screen coordinate system.)

This notion of a traversal may be embodied in at least two different ways for visible surface image generation. One alternative is to assign priorities to polygons in the order that we visit them. Using the traversal order just described we will get a low-to-high visibility priority assignment. These values can then be used within a conventional visible surface display algorithm wherever visibility determinations need to be made. The other obvious alternative, which in fact is the one that we have implemented, does not assign explicit visibility priority values to polygons but uses the traversal to drive a "painter's" algorithm which paints onto the screen's image buffer each polygon as it is encountered in the traversal. Since higher priority polygons are visited later in the traversal and thus painted later, they will overwrite any overlapping polygons of lower priority. The following recursive procedure generates a visible surface image in the above-described manner.

```
PROC Back_to_front(eye:viewing_position;
                   t: BSP_tree);

BEGIN
IF Not_null (t) THEN
  IF pos_side_of (root [t],eye)
  THEN

    BEGIN
    Back_to_front (eye, neg_branch [t]);
    Display_polygon (root [t]);
    Back_to_front (eye, pos_branch[t])
    END

  ELSE

    BEGIN
    Back_to_front (eye, pos_branch[t]);
    Display_polygon (root[t]);
    Back_to_front (eye, neg_branch[t])
    END
END
```

Figures 4,5,and 6 illustrate this visible surface algorithm. Since the display used had only one bit per pixel, the procedure Display_polygon painted the interior of the polygon the background shade and painted the outline of the polygon in the other shade.

The possible weakness of this approach is that the number of polygons in the tree may increase sharply. (Recall, every root polygon splits all crossing polygons in its list in order to put any polygon in one or the other of its

subtrees.) We have attempted to limit this increase by selecting the root polygon at each stage to be the one whose plane splits the minimum number of polygons in its list. Table 1 indicates the performance of the system in limiting the number of polygons in the BSP tree. Figures 7 and 8 show the BSP tree for the environments of Figures 4 and 6, respectively.

Fig. no.	No. of Original Polygons	No. of Polygons in BSP Tree
4	11	11
5	72	100
6	3	5

Table 1: Number of polygons in tree versus original data base

We are currently examining a more sophisticated strategy for minimizing the number of polygons in the BSP tree. In addition to the just-described criterion of choosing a node polygon as one that minimizes the number of polygons that are split, a second criterion is also considered. This one maximizes the number of "polygon conflicts" eliminated. We define a polygon conflict as an occurrence between two polygons in one list in which the plane of one polygon intersects the other polygon. The hope is that these eliminated polygon conflicts will reduce the number of polygons which will need to be cut in the descendant subtrees. More precisely, if P is the set of polygons, then form the sets S_1, S_2, S_3 for each polygon p P as follows:

$S_1 \equiv \{q \in P \mid q$ is entirely in the positive half space of p$\}$

$S_2 \equiv \{q \in P \mid q$ is intersected by the plane of p$\}$

$S_3 \equiv \{q \in P \mid q$ is entirely in the negative half-space of p$\}$

We define a function

$$f(s_i, s_j) = \begin{cases} 1; & \text{polygon } s_j \text{ and the plane of } s_i \text{ intersect} \\ 0; & \text{otherwise} \end{cases}$$

and

$$I_{m,n} = \sum_{s_i \in S_m} \sum_{s_j \in S_n} f(s_i, s_j)$$

We then select the p such that for $S_1(p), S_2(p), S_3(p)$ the expression $[I_{1,3} + I_{3,1} - (|S_2| * \text{weight})]$ which is maximal.

FORMAL DEVELOPMENT

Let us now examine the nature of the binary space-partitioning (BSP) tree more closely. The construction can be carried, in essentially identical manner, for any dimension; nonetheless, it is only the three-dimensional version that is of major interest to us here. However, it is easier to explain its nature in the two-dimensional setting, as the various geometric structures arising can be clearly drawn; thus the discussion of the properties of the tree will be presented assuming a two-dimensional universe. Nonetheless, we encourage the reader, as the next section of the paper is read, to extrapolate the three-dimensional interpretation. In the latter portion of the paper, where combinatorial issues are examined, the results will be given for both two and three dimensions, since combinatorial complexity is dimension dependent. We now begin with some (slightly non-standard) terminology.

Segment - an oriented closed convex subset of a line, i.e., a finite segment, a ray, or a line, with a direction associated with it.

Region - a closed convex set of points of a plane. (A region is normally defined as an open connected set.)*

Extension of a Segment in a Region - given a segment s and a region R, define the extension of s in R to be the intersection of the line on which s lies with the region R, obtaining the segment $\hat{s}_R$. Assign to $\hat{s}_R$ the direction induced by s (we indicate this by pointing an arrow to the right)

Note that a region can be unbounded (a plane) "partially bounded" (e.g., a half-plane), or (completely) bounded (e.g., a finite polygon). The motivation for defining regions and segments in this manner is that in general we have no interest in distinguishing between the bounded, partially bounded, and unbounded sets. The 3-space analogies to segments

*A set is open if there is an "implicit boundary" which is not in the set. Formally, a set of points R in the plane is open if and only if $\forall x \in R, e > 0$ such that $\forall y[|x-y|<e \Rightarrow y \in R]$. A closed set is the complement of an open set (if bounded, the set includes the boundary). Formally, a set of points R in a plane is closed iff for every converging sequence $x \to x$, $\forall n[x \in R \Rightarrow x \in R]$.

and regions are polygons (or alternately, regions) and sectors (or volumes) respectively. The orientation of the polygons corresponds to the usual notion of the front and back sides. We are now ready to examine the general algorithm for construction of a labeled binary space-partitioning tree.

Algorithm I: Construction of a (2-space) BSP tree

Input - a region R and a set of segments Σ lying in R

Output - A BSP Tree

Method - call the function, BSPT, with R and Σ as parameters and $\hat{\Sigma} \leftarrow \phi$.

Procedure - BSPT (R:region; Σ:set of segments) :node

Begin

If $\Sigma \neq \phi$ then

begin

choose $s \in \Sigma$ and form $\hat{s}_R$

$\hat{\Sigma} \leftarrow \hat{\Sigma} \cup \{ \hat{s}_R \}$

Partition R and Σ by $\hat{s}_R$ into R_s, $R_{\bar{s}}$, Σ_{R_s}, $\Sigma_{R_{\bar{s}}}$ defined as:

$R_s \equiv \{ p \in R \mid p \in \hat{s}_R$ or p lies to the right of $\hat{s}_R \}$

$R_{\bar{s}} \equiv \{ p \in R \mid p \in \hat{s}_R$ or p lies to the left of $\hat{s}_R \}$

$\Sigma_{R_s} \equiv \{ B \cap R_s \mid B \in \Sigma - \{s\} \}$

$\Sigma_{R_{\bar{s}}} \equiv \{ B \cap R_{\bar{s}} \mid B \in \Sigma - \{s\} \}$

Create a new node v

leftson(v) := BSPT($R_{\bar{s}}$, $\Sigma_{R_{\bar{s}}}$)

rightson(v) := BSPT (R_s, Σ_{R_s})

label(v) := s_R

return(v)

End

Else

Create a leaf l

label(l) := R

return(l)

End BSPT

Let us look at an example before examining the properties of this algorithm.

Let R be a square and $\Sigma \equiv \{ a,b,c \}$, as in figure 9a.

If a is chosen first, we get figure 9b which creates figure 9c. If, next, b_{R_a} is chosen before c, the final result will appear as in figure 10.

Consider now the set $\hat{\Sigma}$ of segments, which of course lies wholly within the original R. It is easily seen that it partitions R into convex regions (polygons). Each such region, together with its boundary, will be referred to as an area (volume for 3-space). The set of all the areas created by the algorithm will be referred to as a tessellation. The areas may be thought of as the intersection of half-planes (half-spaces for 3-D) created by the lines on which the elements of Σ (or $\hat{\Sigma}$) lie. The purpose of orientation of the segments is to distinguish between the two half-planes. The subscripts of each region, generated by algorithm I, indicate the half-planes whose intersection forms the region. As an example, refer to figure 11 which is a BSP tree for the tessellation in fig. 10 where parentheses are used to indicate subscripting of regions.

CHARACTERISTICS OF A BSP TREE

It should be clear by now that the algorithm performs a recursive partitioning of the plane by the segments lying in it. However, observe that given a set of segments , that more than one tessellations can be generated by the algorithm depending upon the order in which segments are selected. Observe that in fig. 9, had the order of selection been c, b, a, fig. 12 would have been produced, which not only looks different from fig. 10, but has four areas, as opposed to five. Since a tessellation is formed by the extended segments, as opposed to the segments themselves and the length of an extended segment is dependent on the size of the region containing it at the time it is extended, selecting segments in different orders produces different regions, and thus the dependence of the tessellation on the order of selection.

It is also possible to have, for a given set of segments, more than one tree which describes the same tessellation. Assume that at some stage of the construction of the tree, we are examining the region R_k and the associated set of segments $\Sigma_{R_k} = \{s_1, s_2, \ldots, s_m\}$. If $\bigcup_{i=1}^{m} s_i = \bigcup_{i=1}^{m} \hat{s}_i$ with respect to R_k, then every permutation π on i= 1,2,...,m will result in a different subtree, where the

subtree is generated by selecting segments in the order $s_{\pi(1)}, s_{\pi(2)}, \ldots s_{\pi(m)}$. Nonetheless, every subtree will describe the same tessellation of R_k. Consequently, there are distinct trees describing the same tessellation of the original region R. For example, in figure 13, either tree specifies the same tessellation.

An important special case occurs when the initial set of segments is equivalent to the extended set, i.e., $\cup\{s|s\in\Sigma\} = \cup\{\hat{s}|s\in\Sigma\}$. If in addition the initial region is a plane, all of the elements of Σ, would be lines. Since extension has no effect, the tessellation is fixed before the algorithm begins. We call such a tessellation a maximum tessellation because any set of segments lying on the same set of lines can produce only tessellations whose areas are the union of the areas of the maximum tessellation, as can be seen by comparing figures 10 and 12 with fig. 14. It follows that any set of segments has a corresponding maximum tessellation whose cardinality is the maximum of the number of areas produced by any tessellation resulting from the set. In general, the number of different tessellations that can be derived from a set Σ is, in some sense, the complement of the number of distinct trees which describe the same tessellations.

A BSP tree constructed by algorithm I contains nodes labeled with segments and nodes labeled with areas. The segment nodes are exactly the interior nodes of the tree and the "area" nodes are the leaves. The algorithm can be thought of as first generating a binary tree composed of only the segment nodes. There will then be segment nodes which have one or two empty sons. (Every node of a binary tree has potentially two sons, left and right. If a node does not have one or both sons, we refer to these as "empty sons.") At each empty son, an area node is added. The resulting tree is such that all segment nodes have both a left and a right son, either of which could be another segment node or an area node. Since binary trees of n nodes have n+1 empty sons, it follows that the number of area nodes is one more than the number of segment nodes, thus a tree of 2n-1 nodes is needed to represent a tessellation containing n areas.

Each subtree of a BSP tree represents some region R_i in the sense that the union of all the areas represented by the leaves of R_i equals R_i (the segments represented by the segment nodes of R_i are thus, also included in this union). For notational purposes we will designate the region represented by the entire tree as R_o. This, of course, is the original region from which the tessellation is formed. The extension of the segment s represented by the root q of a subtree partitions a region R_i, and the regions represented by the two subtrees of q are the two half-spaces formed from R_i by $\hat{s}$. If, upon traversing the tree one reaches q, then taking the left or right branch of q would have a geometric correspondence to selecting one of these two half-spaces. A path in the tree, then, reflects a successive selection of smaller and smaller portions of R_o. In fact the region represented by a subtree is the intersection of the half-spaces with respect to R_o formed by the extension of the segments which are on the path to the root of the subtree q (but not including q). It immediately follows that the area which is "added" at each empty son is exactly the intersection of the half-spaces with respect to R_o formed by the extension of segments whose nodes are on the path to the son.

It is easy to see how a BSP tree can be used to locate which area of the tessellation a point lies. Beginning at the root, determine on which side of a segment the point lies and proceed to the son representing the half-space corresponding to that side (points on the line being assigned arbitrarily to one of the two half-spaces). Repetition of this process will generate a path to a leaf node that represents the area in which the point lies, thus solving what might be called the "location problem" with respect to a tessellation.

BSP Tree Used for Priority Ordering

The ability of a BSP tree to be used for the generation of a priority ordering is based upon the principle that given in which half-space lies the point to which the ordering is relative (usually thought of as the "eye" or viewing position), all points in this same half-space will have priority over all points in the other half-space. Although this fact is fairly self evident for half-spaces, it is also true for any two convex regions.

To obtain a priority ordering from the tree, an inorder traversal is performed. The choice of taking the left or right branch of a node q representing segment s is always made in favor of the subtree which represents the region that is contained in the same half-space that the viewing position is in, this half-space having been formed by $\hat{s}$ with respect to R_o. It is easy to see that such a policy will result in the first area node to be reached being the one in which the viewing position lies, i.e. the solution to the location problem mentioned earlier. Priority is assigned to a node upon backing-up from it during the traversal.

Thus for each node q, all nodes of the chosen subtree receive higher priorities than q, and similarly, all nodes of the remaining subtree obtain a lower priority than q. The entire traversal of the tree will then produce a total ordering of the nodes, and this is precisely the visibility priority of the elements represented by the nodes. Note that it is requisite that R_0 be convex to guarantee this property. Since the partitioning of a convex object produces two convex objects, the convexity of R_0 implies the same property for all subsequent refinements of R_0 during the construction of the tree. Thus all areas are convex which is sufficient to guarantee the existence of a priority ordering of the areas.

Comparison of Uses of the BSP Tree

The first appearance of a BSP tree in the general literature was in Sutherland, et al. (1974) reviewing the work of Schumaker, et al. (1969), although the tree was not named and its general properties were not developed. The application presented was that in which invisible "dividing planes" were introduced to the data base. The method involved the designer of a simulation scene manually positioning "clusters" such as buildings, trees, mountains, etc., so that vertical dividing planes could be placed between the clusters to varying extents. This resulted, in terms of a BSP tree, in the generation of a tessellation of the surface by the dividing places which are represented by segment nodes, and each cluster was contained wholly within an area. Thus each cluster corresponded to an area node. A priority ordering could then be obtained on the clusters.

Additional power is available if the tessellation is a maximum tessellation. In this case, it is possible to compute off-line the priority ordering for each case of the viewing position being in a different area. This follows from the fact that since the areas are formed by a maximum tessellation, it is not possible for two different points in the same area to be on different sides of the extension of a segment with respect to R_0 (since in a maximum tessellation all segments are equal to their extensions with respect to R_0). Thus for each area the traversal of the tree is fixed. The Sutherland, et al., presentation suggests taking advantage of this by pre-computing and storing for each area its inherent priority ordering on the clusters (since the dividing planes are not part of the scene they need not be included in the ordering). It was then sufficient to solve the location problem in order to obtain the priority ordering. Since this method requires n^2 storage space (where n is the number of clusters) and the traversal of the tree is O(n), it is not clear whether this approach is advantageous. Also since a maximum tessellation is required the tree will be the largest possible for a given set of clusters.

The application of BSP trees introduced in this paper is something of a complement to that presented in Sutherland, et al. Here those objects represented by the segment (or polygon) nodes constitute the visible data while the areas of the tessellation are of no importance. In fact, the function Make_tree presented earlier produces only the segment nodes. The area nodes are only implied by the empty sons. Also, Make_tree forms a BSP tree for three dimensions while the former method, although working in 3-D, forms a BSP tree for two dimensions, and Make_tree's tessellation in general is not maximal. Clearly the BSP tree can be used with dividing planes to divide 3-space into volumes, and a hybrid of polygons and dividing planes could also be developed. For instance, each area node of a tree constructed with dividing planes could be replaced with a BSP tree constructed of polygons for the cluster contained in the area.

Combinatorics of the BSP Tree

We will now examine the size of the BSP trees. The previous discussion was presented, for simplicity's sake, for the 2-D case; here we will derive some formulas both for the 2-D and 3-D BSP trees. Although we are most interested in the 3-D case, 2-D is important in the special 3-D case in which all of the objects "sit" on the ground and can be separated by vertical planes. This is equivalent to a 2-D BSP tree corresponding to the 2-D scene obtained by projecting the objects and the separating plane on the ground plane.

As noted previously, the BSP tree can be created by both infinite and finite objects. The infinite objects are planes for the 3-D case are lines for the 2-D case. The corresponding finite objects are non-intersecting convex polygons and segments. We will examine these two extremal cases in turn.

A d-dimensional BSP tree partitions the d-dimensional space by (d-1)-dimensional objects. We thus examine first, what is the maximum number $f_d(n)$ of volumes of a d-dimensional space that can be created by n (d-1)-dimensional planes. In the 2-D case we have been considering, this corresponds to the maximum tessellation of the plane using lines.

The general formula is

$$f_d(n) = \sum_{i=0}^{d} \binom{n}{i}.$$

As there is a one-to-one correspondence between the volumes and the leaves of the binary BSP tree, the number of the nodes of the BSP tree is $2f_d(n)-1$.

How many (d-1)-dimensional regions created from (d-1)-dimensional planes under the assumption that no 3 planes intersect along a single (d-2)-dimensional line? It can be shown that the number is

$$\sum_{i=0}^{d} i \binom{n}{i}.$$

In the other extremal case, where the objects given are n (d-1)-dimensional non-interpenetrating convex polygons, we examine the worst case, namely compute the maximum number of the (d-1)-dimensional regions that are obtained from the polygons by the intersection of the n (d-1)-dimensional planes on which they lie. It can be shown that the number is

$$\binom{n}{2} + n^{d-1}$$

We summarize the results in Table 2 for the two interesting cases d=2 and d=3.

	Volumes	Unbounded Objects	Bounded Objects
2-D:	$\frac{n^2+n}{2}+1$	n^2	$\frac{n^2+n}{2}$
3-D:	$\frac{n^3+5n}{6}+1$	$\frac{n^3-n^2+2n}{2}$	$\frac{n^3+3n^2+2n}{6}$

Table 2: Maximum possible nodes in BSP tree.

Conclusion

A solution has been presented to the visible surface problem which appears to be more efficient than previous solutions whenever many images are to be generated of the same static environment. The algorithm is easy to implement since both phases, the preprocessing and the image generation, can each be succinctly stated in a short recursive procedure. The major potential weakness, a large increase from the number of original polygons in the data base to the number in the BSP tree, has not occurred in any environment so far encountered.

Acknowledgement

We wish to thank Greg Abram for much needed and appreciated program development, Mike Cronin for numerous improvements to the narrative, and the referees for helpful and thorough reviews of the first draft of this paper.

References

Berman, G. and Fryer, K.D. Introduction to Combinatorics, (1972) Academic Press.

Fuchs, H. and Johnson, B. "An Expandable Multiprocessor Architecture for Video Graphics" Proc. 6th Annual Symp. on Computer Architecture, (1979) pp 58-67

Schumaker, R.A., Brand, P., Gilliland, M. and Sharp, W. "Study for Applying Computer-Generated Images to Visual Simulation," AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory (1969)

Sutherland, I.E., Sproull, R.F. and Schumaker, R.A. "A Characterization of Ten Hidden-Surface Algorithms", (1974) ACM Computing Surveys, 6 (1): 1-55

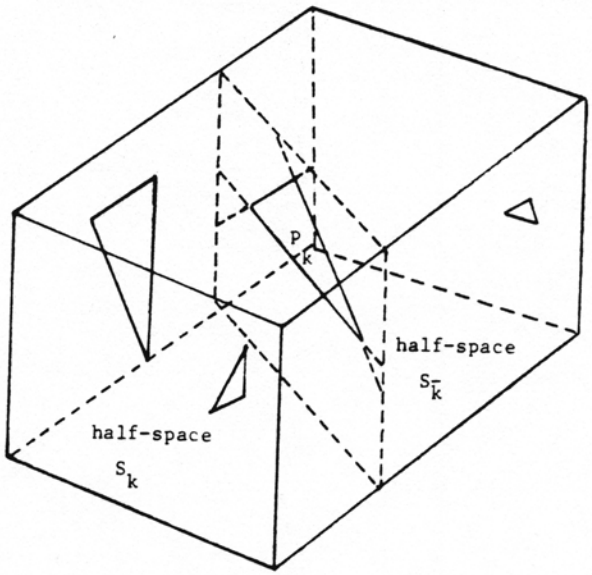


Figure 1: Environment split by plane of P_k

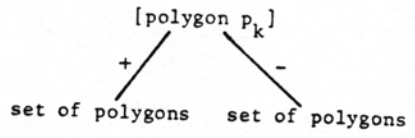


Figure 2: Beginning of BSP tree construction

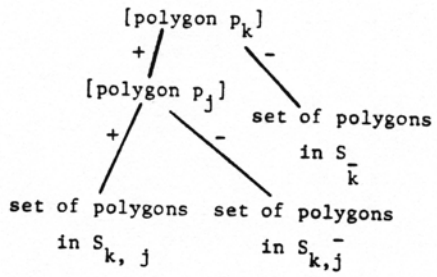


Figure 3: BSP tree after two steps

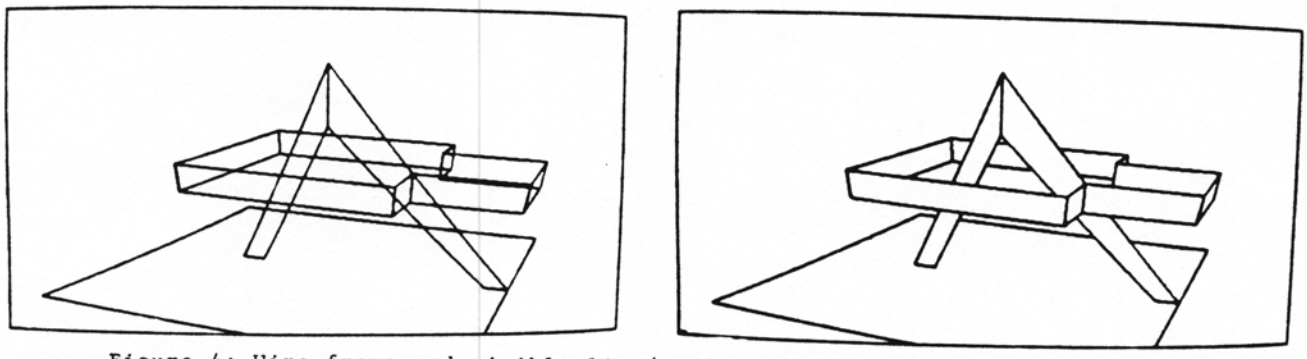


Figure 4: Wire-frame and visible line/surface images of same environment
(11 original polygons; 11 in BSP tree)

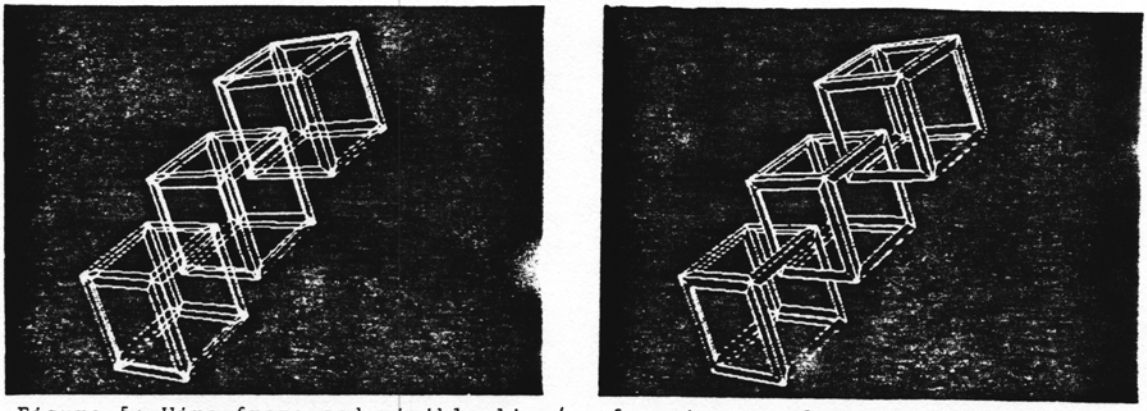


Figure 5: Wire-frame and visible line/surface images of same environment
(72 original polygons; 100 polygons in BSP tree)

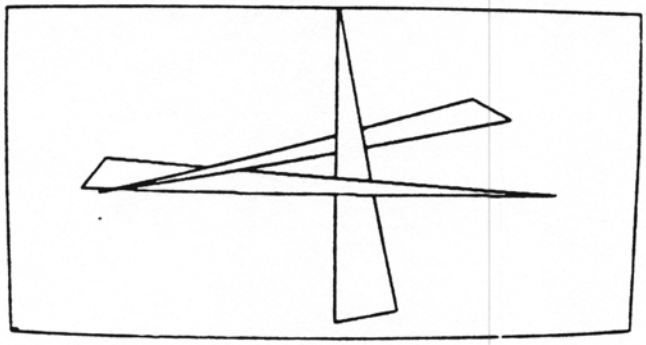
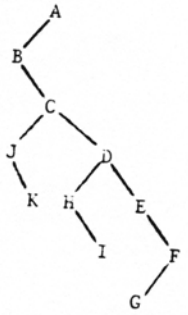
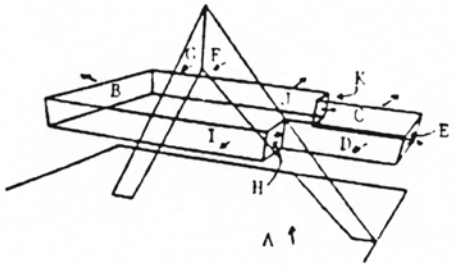


Figure 6: Visible line/surface image of simple object whose polygons cannot be directly assigned visibility priorities (some polygons here have been split during preprocessing)

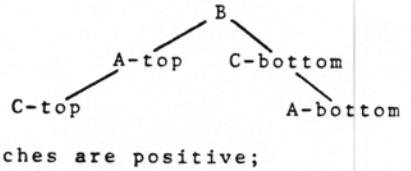


(left branches are positive)



(arrows on positive side of polygons)

Figure 7: BSP tree and polygons of Fig.4



(left branches are positive;
positive sides of all polygons are visible)

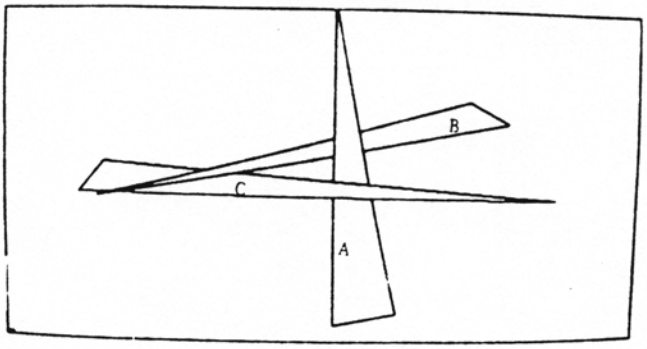


Figure 8: BSP tree and polygons of Fig. 6 (A and C have each been split into two parts by plane of polygon B)

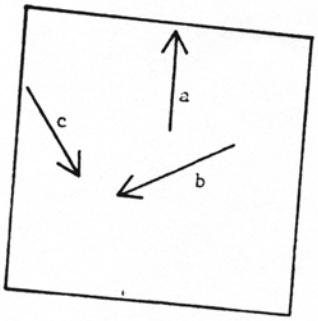


Figure 9a

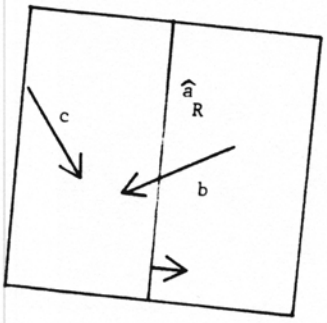


Figure 9b

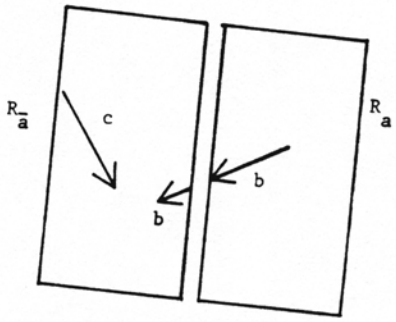


Figure 9c

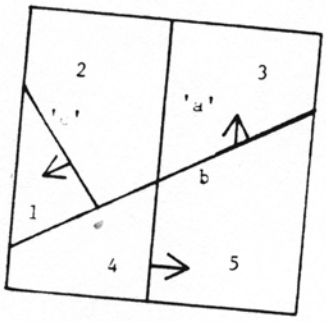


Figure 10

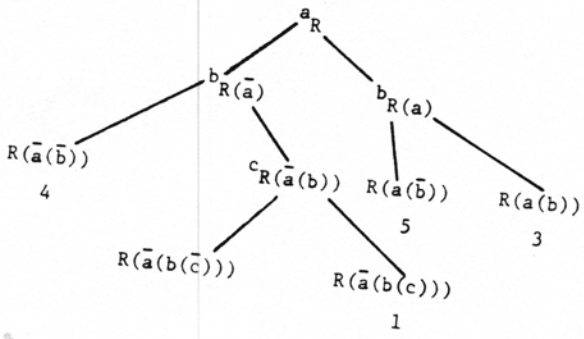


Figure 11

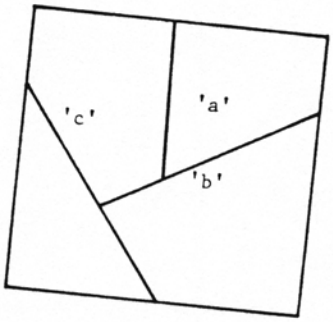
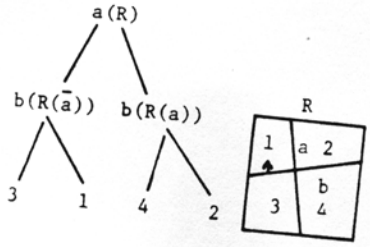


Figure 12



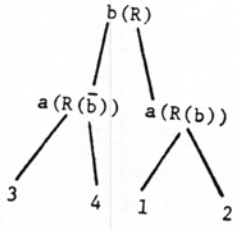
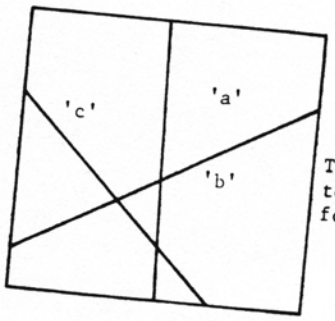


Figure 13



The maximum tessellation for Figure 9

Figure 14

(end)