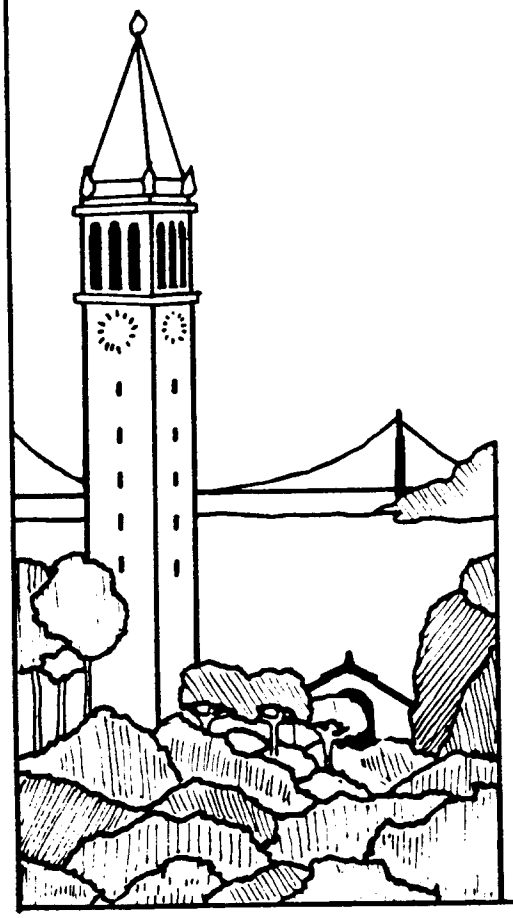
Binary Space Partitioning for Previewing UNIGRAFIX Scenes

Ziv Gigus



Report No. UCB/CSD 86/280

December 1985

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Binary Space Partitioning for Previewing UNIGRAFIX Scenes

Ziv Gigus

Master's Project Report
Under Direction of
Prof. Carlo H. Séquin

Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
December, 1985

ABSTRACT

The binary space partition algorithm by Fuchs et. al.[1,2] has been implemented for interactive display of scenes described in the UNIGRAFIX language. The scene is displayed using the painter's algorithm on a device that draws only convex polygons, thus an algorithm for decomposing concave polygons into convex pieces has also been implemented.

The program provides fast, near real time interaction for modifying the viewing parameters. The user can move through the scene, examine its content and the relation between the objects in the scene. In this way, one can easily pick the "best" viewing parameters for rendering the scene on hard copy devices, using the relatively slow scan-line based renderers provided by the UNIGRAFIX system.

1. Introduction

In many computer graphics applications we need to provide images of a world model that changes less frequently than the viewpoint, the viewing angle or the viewing direction. Schumacker et. al.[3] observed that in such situations, preprocessing the scene can reduce the amount of computation required during the image generation phase. However, the algorithm they presented for such situations depends on manual intervention in building the internal data structures. This makes it difficult to generate new databases and thus limits its general usefulness. Most of the visible surface algorithms developed in recent years[4,5,6] tackle the problem of displaying the correct image when given a fixed set of viewing parameters. When these parameters change, all the work has to be redone, and there is no use of the computation done for previous frames. Thus, most real-time interactive applications (e.g flight simulators) rely on special purpose hardware to achieve high display rates. However, for a general purpose graphics system, the Binary Space Partitioning algorithm[1,2] is the only known algorithm that is designed for fast redisplay of the same scene from different viewpoints, doing most of the computation during a preprocessing stage, and little computation per frame. This algorithm is particularly attractive because of recent fast graphics terminals and workstations (eg. the Silicon Graphics IRIS workstation[7]). These systems provide fast rendering of *3D* polygons, given the viewing transformations, but do not provide hidden-surface elimination.

The UNIGRAFIX system provides the user with a terse and powerful language for defining scenes made of polyhedral objects (for more detail see UNIGRAFIX user manual[8]) , it also provides several tools for generating and modifying objects.[9,10] Currently there are three different renderers provided for displaying scenes on different terminals and hard-copy devices. These renderers use different scan line based algorithms for generating the images. While these are suitable for producing the final pictures, we, as users of the system, found that often we would like to have a fast

previewing tool that gives us the means for moving through the scene, examining it latter with various viewing parameters. When the scene constructed with UNIGRAFIX is complex, predicting the "best" viewpoint for a given scene is difficult. At other times we would also like to take a detailed look at the scene, to examine the exact structure of the objects we created (either manually or automatically). From our experience, performing these tasks using the scan line based renderers, even those that provide limited interactivity,[11] is cumbersome and time consuming. Because of the long image generation time, the type of interactivity needed for these tasks can not be provided by programs built on top of these renderers. Thus, we needed a fast algorithm as a basis for an interactive program that will provide these capabilities.

We chose to use the *binary space partitioning* algorithm on a Silicon Graphics IRIS workstation[7] as the host machine. It performs all the geometric operations (transformations and clipping) at a high rate, so we only had to worry about the hidden-surface removal problem, which is not addressed by our IRIS.

The program consists of two independent parts: the scene preprocessor and the display program. The preprocessor reads the UNIGRAFIX scene description, builds the *binary space partitioning* tree and produces a file describing this tree. The preprocessor can be run off-line on any machine (not necessarily the IRIS). Therefore, one can use a fast host to perform the computation intensive preprocessing phase. The display program is run on the IRIS (or can be written to run on any machine with similar capabilities). It reads in the tree description, and provides the user with interactive means of moving through the scene, providing fast frame generation rate. As UNIGRAFIX supports arbitrary polygons (with concave contours and holes) and the IRIS can only handle simple polygons, an algorithm for decomposing arbitrary polygons into convex parts was also implemented. This algorithm is incorporated into the preprocessing program.

1.1. Paper Organization

Chapter 2 describes two previous algorithms that have influenced the development of the *binary space partitioning* algorithm. The *binary space partitioning* algorithm is presented in chapter 3, where we also discuss the effect of the scene structure on the size of the final tree. The algorithm used for dividing an arbitrary polygon along a plane is presented briefly in section 4. In this section I also discuss briefly the problems I had implementing this algorithm. Chapter 5 describes the algorithm used for decomposing arbitrary polygons into convex parts. In chapter 6 the user interface of the display program is described. Chapter 7 draws conclusions about this work. Appendix I contains manual pages for the programs discussed in this report.

2. Historical Background

In this section I briefly present two earlier algorithms, that have influenced the development of the *binary space partitioning* algorithm.

2.1. List-Priority Algorithms.

The hidden-surface elimination problem can be roughly viewed as follows:

Given the world model, a viewpoint location, a viewing direction and viewing angle, order the objects in the scene such that objects with lower priority can not obscure those with higher priority.

Having sorted the objects by this order, when two objects are candidates for occupying the same pixel in the picture, the one with the higher priority will be visible.

Sutherland et. al.[4] categorize the class of algorithms that establish this ordering under *list-priority* algorithms. The *binary space partitioning* algorithm described in section 3 belongs to this category. The next two subsections briefly describe two earlier algorithms of the same category, pointing out features of these algorithms that have influenced the design of the *binary space partitioning* algorithm.

2.2. Painter's Algorithm and Face Cutting - Newell's Algorithm

Newell[12,4] introduced the concept of "overwriting" objects to achieve hidden-surface elimination and transparency effects. Instead of using the priority list for determining the visible object (polygonal face in this case) at a given pixel, the list can be used in a different manner. One can just draw the faces into the frame buffer in increasing (back to front) order, such that faces with higher priority overwrite those with lower priority. After the entire list has been processed, a correct hidden-surface image is produced. Furthermore, if faces are allowed to partially overwrite the ones "underneath" them, a transparency effect is achieved. This hidden-surface elimination method is often referred to as *the painter's algorithm.*

This algorithm is particularly attractive in view of recent raster devices. Most medium-cost devices provide fast polygon filling and coordinate transformations. Hidden surface elimination, on the other hand is provided only by the more expensive ones and it is an extra feature.

The heart of Newell's algorithm is a priority ordering procedure that Newell referred to as a "priority computer". This procedure sorts a set of faces into a priority order. We are not going to describe this procedure in detail. However, one step of this procedure that has influenced the design of the *binary space partitioning* algorithm and is described below.

Given the task of ordering a set of objects into a priority list, one may find that such an ordering does not exist. This is the case when two objects mutually overlap in depth leading to an ordering conflict. This conflict is often called *cyclic overlap* conflict (see Figure 2.1). Newell observed that such conflict can be resolved by dividing one face along the plane of the other face, producing two faces instead of the one that has been divided (see Figure 2.1).

2.3. Space Division and Fixed Face Priority - Schumacker's Algorithm

Developing a hidden-surface algorithm for a flight simulator, Schumacker et. al[3,4] observed that they were facing a world model that rarely changes, while the viewpoint, viewing direction and viewing angle change frequently. Given these properties, they looked for a method of preprocessing the scene description to reduce the computation required for regenerating the image as the

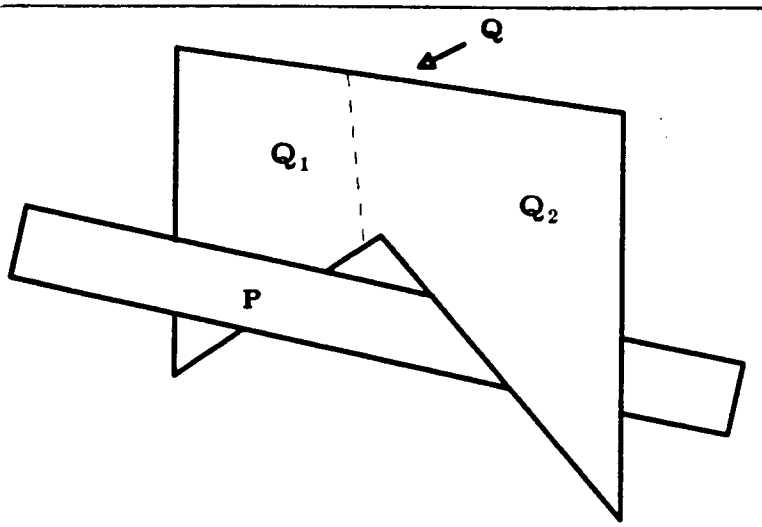


Figure 2.1.

Cyclic overlap: faces P and Q can not be placed in priority order. However if Q is divided into two faces Q_1 and Q_2 by the plane of P then the order is (Q_1, P, Q_2).

viewing parameters change.

To achieve the above goal, Schumacker introduced the concepts of *clustering* and *fixed face priority*.

Schumacker observed that given a viewpoint, the computation of face priority can be divided into two stages. In the first stage the world model is divided into clusters, where each cluster contains a set of objects (a set of faces in this case). Within each cluster every object is compared to every other object in the cluster to compute the object priority. The clusters in this algorithm are constructed manually so that within a cluster the priority of the objects (faces) can be determined independently of the viewpoint (hence, it is called *fixed face priority*). Thus, priority computation can be done once at the preprocessing stage and need not be recomputed when the viewpoint changes (see Figure 2.2).

In the second stage, the algorithm computes the relative priorities of the clusters. Once these are established, the scene can be rendered as follows: if cluster A is closer to the viewpoint

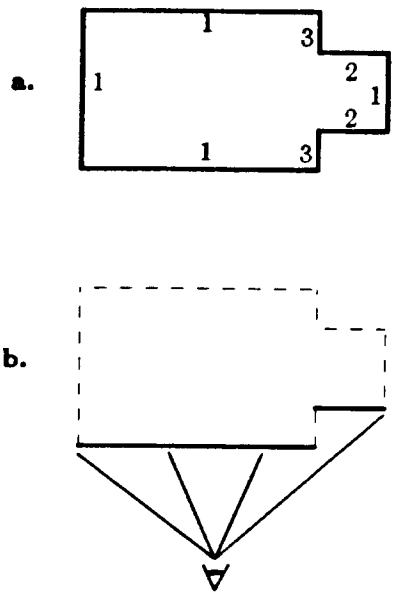


Figure 2.2.

Face priority. **a.** *Top view of an object with face priority numbers assigned.* **b.** *The same object with a specific viewpoint located. Dashed lines show back faces.*

than cluster B, then all the objects in cluster A are closer to the viewpoint and have higher priority then those in cluster B. This observation does not hold in general, unless the clusters are convex. Schumacker used planes, that were introduce manually, to separate the clusters. These planes are used to create a binary tree, that given a viewpoint has to be traversed to find out the correct cluster priority. The calculation of cluster priority is illustrated in Figure 2.3.

Schumacker's algorithm tolerates motion of clusters in the environment if they remain linearly separable. The cluster priority, recomputed every frame, correctly accounts for this motion. Using special-purpose hardware, the algorithm was able to produce real-time animation for the flight simulator (for a detailed description of the hardware see (3,4)).

A few important points should be made about this algorithm. The use of *fixed face priority* constrains the structure of the objects in the environment. Only objects that have *fixed face priority* can be put into a cluster. The spatial relation between the objects is also constrained as

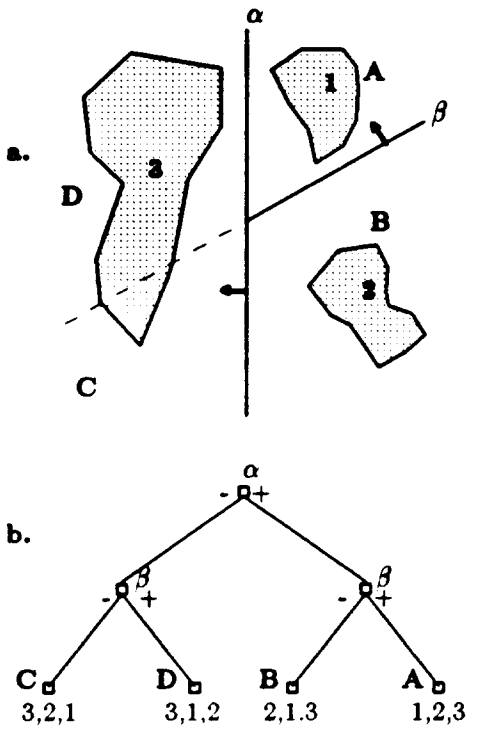


Figure 2.3.

Cluster priority. **a.** *Three clusters (1,2,3) are separated by two planes (alpha ,beta) (the arrows show the direction of plane normals). The viewpoint may be located in one of four areas (A,B,C,D).* **b.** *A tree structure for finding the cluster priority from the viewpoint location. At nodes labeled with planes, we take a branch depending on which side of the planes the viewpoint lies. At the leaves we get the cluster priority order.*

they have to be linearly separable. A fair amount of user intervention is required. The planes separating the clusters have to be introduced manually into the database as it is very hard to find the planes automatically. Furthermore, when a cluster that does not have fixed face priority is detected, it has to be manually subdivided into several "well behaved" clusters. These features are acceptable in a special purpose flight simulator, for which there is a limited number of possible environments. Experts can do the above tasks as a one time effort. However, these features prevent the algorithm from being used by ordinary end-users who want animated viewer motion through a general environment. Even if the user has the expertise required by the algorithm, the overhead in performing these steps, just for viewing one scene several times, is too high.

3. The Binary Space Partition Algorithm

Fuchs et. al.[1] looked for a way to use the preprocessing approach used in Schumacker's algorithm, while not having any user intervention in the process. They observed that if one were to use clusters made of single polygons, then given a viewpoint one can choose an arbitrary polygon A_k and assuming all other polygons lie on the negative or positive side of A_k, use the painter's algorithm in the following fashion:

1) Draw all the polygons in the scene that are on the far side of A_k, with respect to the viewpoint according to their priority.

2) If A_k is not a backface then draw A_k.

3) Draw all the polygons in the scene that are on the near side of A_k, according to their priority.

This process will draw the polygons according to their list-priority, resulting in the correct image with hidden surfaces removed. This approach is similar to the way the separating planes were used in Schumacker's algorithm, but instead of using artificial planes, the planes of the polygons themselves are being used. The problems of constructing the clusters and selecting the separating planes are automated and no user intervention is required.

The algorithm above assumes that given a polygon A_k, every other polygon in the scene is either on the far side or the near side of A_k. However, that might not be the case as some polygons may lie on both sides of A_k. Here, in a fashion similar to Newell's solution for the cyclic overlap conflict, we split these polygons along the plane of A_k.

3.1. Construction of the Binary Space Partitioning Tree

The algorithm, as stated above, fails to explain how the priority of the polygons on both sides of polygon A_k is determined. However, we can recursively repeat this algorithm for both sides (in a fashion similar to sorting with the *Quicksort* algorithm) , to get the correct list-priority ordering.

Fuchs et. al. observed that the basic step of the above algorithm is partitioning the *3D* space of the environment into two half-spaces along the plane of A_k: one half-space s_k contains the part of the environment (the polygons in this case) that lies on the positive side of A_k's plane, and the other half-space $\bar{s}_k$ contains the part of the environment that lies on the negative side of the plane. As this is a recursive binary subdivision, it can be described in a binary tree (Fuchs et. al. called it a *Binary Space Partitioning*, or "BSP" tree). The root of the tree contains A_k, the right subtree contains all the polygons in s_k and the left subtree contains all the polygons in $\bar{s}_k$. The tree construction step can be done at a preprocessing stage, once for all possible viewing positions. The tree remains the same as long as the model does not change. Procedure MakeTree gives the detailed algorithm in pseudo-code. Figure 3.1 illustrates this process and the resulting tree.

```
Procedure MakeTree(PolygonList) returns BspTree;

begin
  if PolygonList is empty then
    return NullTree;
  else begin
    Root := SelectPoly(PolygonList);
    NegativeList := [];
    PositiveList := [];
    foreach Polygon in PolygonList do
      if Polygon ≠ Root then begin
      if Polygon is on the negative side of Root then
        AddToList(Polygon,NegativeList);
      else if Polygon is on the Positive side of root then
        AddToList(Polygon,PositiveList);
      else begin
        SplitPoly(Polygon,Root,NegativePart,PositivePart);
        AddToList(NegativePart,NegativeList);
        AddToList(PositivePart,PositiveList);
      end
    end

    return CombineTree(MakeTree(NegativeList), Root, MakeTree(PositiveList));
  end
end
```

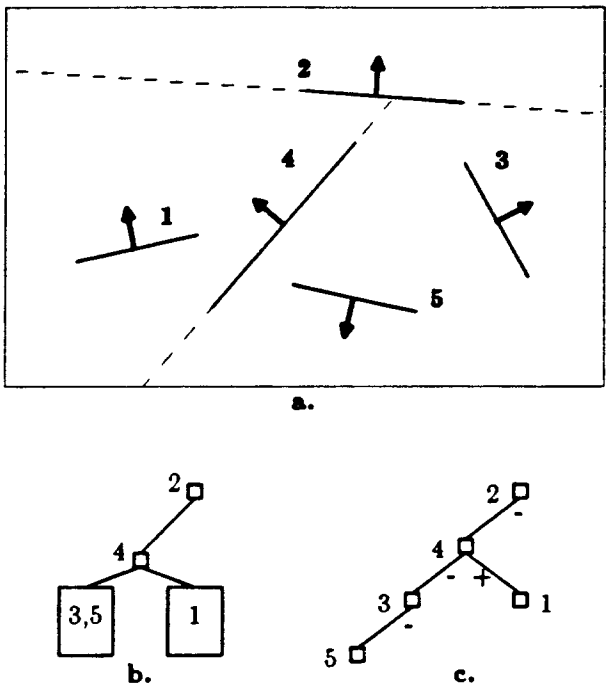


Figure 3.1.

Tree construction. **a.** *A top view of the scene. Arrow represent the direction of the plane normals. Dashed lines represent the cutting planes at first and second level of the recursion.* **b.** *After two levels of recursion, polygons 2 and 4 are chosen as the root at the first and second level respectively.* **c.** *The final tree.*

3.2. Use of the BSP Tree for Image Generation

Given the BSP tree, generating the image is simple. The tree is traversed in a special in-order fashion. Starting from the root, recursively for each node of the tree, we determine whether the viewpoint is on the negative or the positive side of the polygon. Then we first traverse the subtree that is "further" from the viewpoint, draw the polygon at the node (if it is not a back-face), and then traverse the "near" subtree. The traversal procedure is given below.

```
Procedure DisplayTree(Tree);
/* Procedure Display renders a polygon performing
 * the geometric transformations and clipping
 */
begin
  if Tree is empty then return
  else begin
    if viewpoint is on the positive side of Tree.Polygon then begin
      DisplayTree(Tree.NegativeSubtree);
```

```
      Display(Tree.Polygon);
      DisplayTree(Tree.PositiveSubtree);
    end
    else begin
      DisplayTree(Tree.PositiveSubtree);
      if back faces are to be visible then Display(Tree.Polygon);
      DisplayTree(Tree.NegativeSubtree);
    end
  end
end
```

3.3. Tree Size

The choice of the root polygon strongly influences the size of the tree. In the example illustrated in Figure 3.1 the resulting tree has 5 polygons, while in the example of Figure 3.2 the tree for the same scene has 10 polygons. Even in simple scenes, "bad" selections of the root polygon at many levels of the subdivision, may lead to trees that are much larger than the original data base. A tight upper bound for the size of tree has not been found yet.

We must try to keep the tree as small as possible. The size of the tree stands in almost direct relation to the frame generation time. The bigger the tree is, the more time it takes to traverse it. Although it takes less time to fill the smaller fragmented polygons, as this process is done by a special purpose processor, the increase in the tree traversal time becomes the dominant factor in displaying the tree. Furthermore, to get reasonable display rates, the tree has to be kept in main memory. When the tree grows beyond the size of the available memory, paging causes a sharp increase in image generation time.

Because of the prohibitive cost of computing the optimal tree, one must use a heuristic approach in selecting the root polygon at every level of the tree. As the behavior of the BSP tree is complex, selecting what heuristic to use is difficult. Selecting a polygon that cuts many other polygons in early stages of the algorithm, may reduce the fragmentation of the scene in later stages. The opposite also holds; a polygon that conflicts with only a few polygons may divide the half-space into two subspaces such that one or both of them contain many conflicts and result in significant fragmentation of the data base.

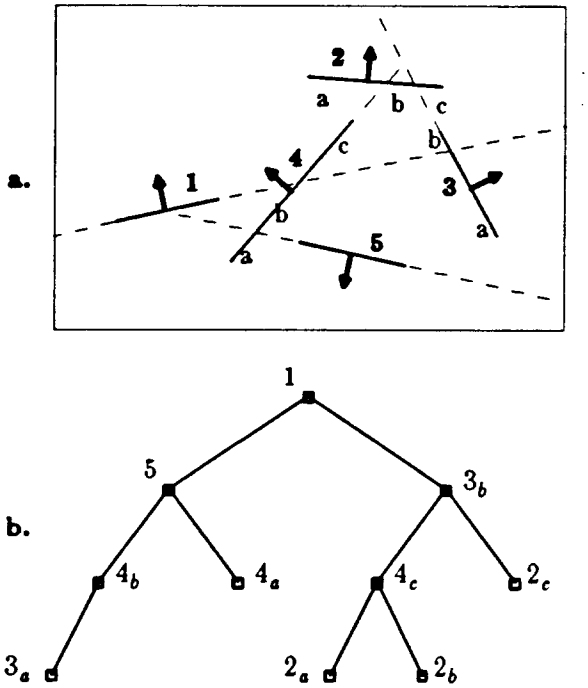


Figure 3.2.

Root selection. **a.** *Top view of the scene. The cutting planes at all level of the recursion are represented by dashed lines; the planes are shown only in the half-space they affect.* **b.** *The resulting tree.*

The heuristic suggested in (1) is: *maximize a weighted sum of the number of conflicts between the polygons in the positive half space with those in the negative half space minus the number of cuts produced by the root polygon.* The weighting function in this sum has to be selected empirically. This approach incurs a high computational cost. Every polygon of the given space has to be intersected with the plane of the candidate polygon, and then every polygon in one half-space has to be tested against the plane of every polygon of the other half-space. These tests have to be repeated for every polygon of the half-space of every level of the tree.

In a later paper,[2] Fuchs et. al. reported that a simpler heuristic produced very good results in most scenes. The heuristic was a pigeonhole approach: *at every level of the tree, select the polygon whose plane cuts the fewest other polygons.* On the scenes that they applied it to, this approach resulted in an increase of the size of the original data base by a factor of 2.33 at the

most. A further computational simplification of this approach was suggested by Z. Kedem. Instead of testing every polygon in the given half-space to select the best one, only a few randomly chosen candidates should be tested to select the one that cuts fewest polygons as the root at this level*. They reported that with about 5 polygons as candidates at each level, the results were nearly as good as their first approach.

I tried all these approaches and found that the last approach produced results that were almost as good as the more computationally expensive ones. In most scenes the number of polygons in the BSP tree was between 1 and 3 times the size of the original data base. The decrease in the size of the tree that was achieved when the other approaches were used did not justify the increase in computation time. The sizes of the trees produced for several scenes are given in table 3.1 as a function of the number of random trials in selecting the dividing polygon (these results represent the best of three runs for each entry), using Kedem's idea.

BSP tree size statistics					
Figure name (and number)	**Number of polygons in the model**	**Output for varying number of candidates for root selection**			
		1	**5**	**10**	**15**
Half of a Clockwork Orange (Figure 3.3)	*642*	*1079*	*1045*	*1003*	*963*
Clockwork Orange (Figure 3.4)	*1284*	*2972*	*1986*	*2070*	*1948*
Granny Knot (Figure 3.5)	*64*	*300*	*294*	*321*	*297*
Diamond Cell (Figure 3.7)	*483*	*2756*	*2655*	*2694*	*2587*
Rubik's Cube (Figure 3.7)	*162*	*174*	*162*	*162*	*162*
Interlocking Triangles (Figure 3.8)	*32*	*207*	*197*	*192*	*166*
Intersecting Tetrahedrons (Figure 3.9)	*20*	*256*	*250*	*253*	*253*

Table 3.1.

* This simplification resembles, in some respects, the heuristic used in the pivot selection step of the Quicksort algorithm.

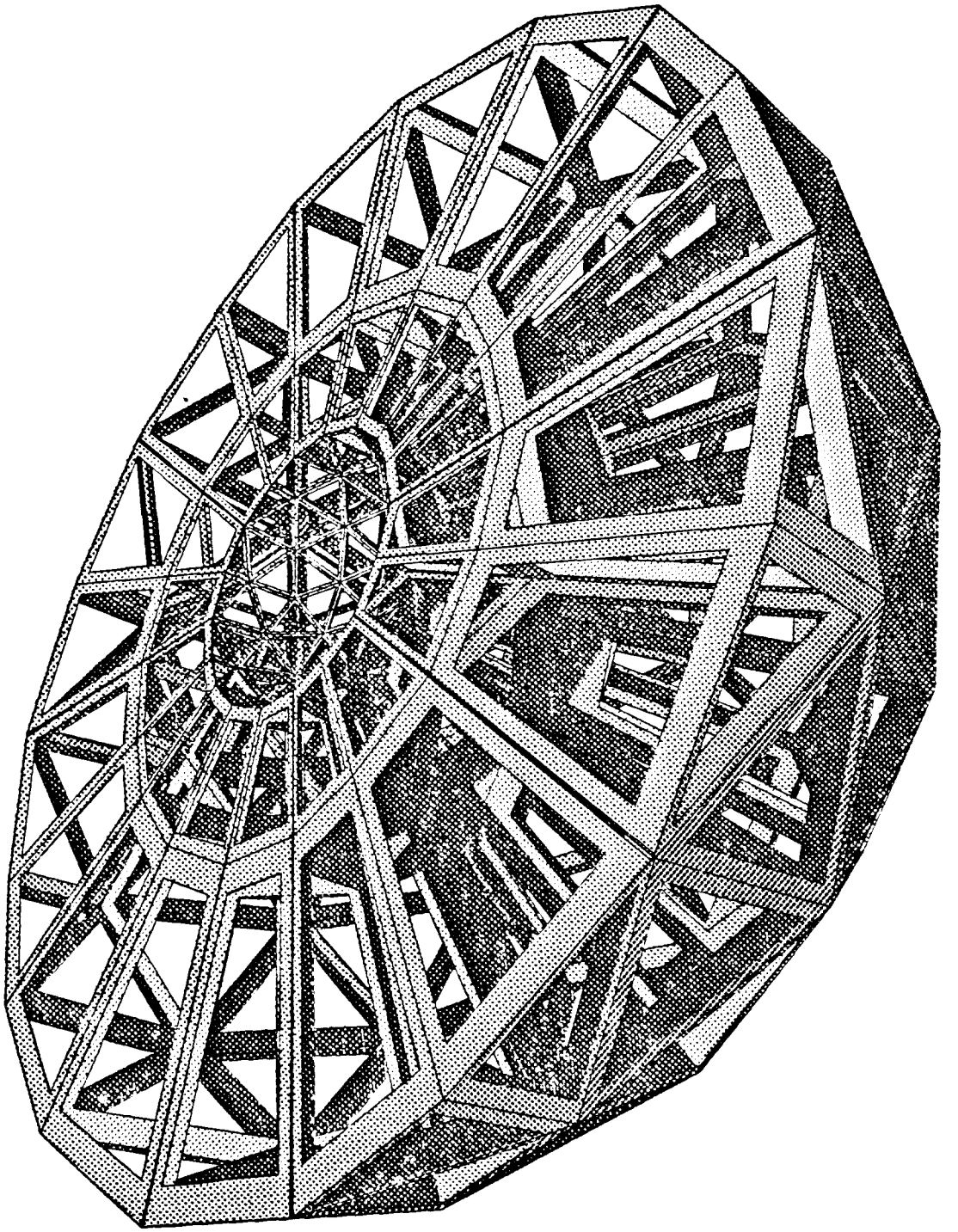


Figure 8.8. *Half of a Clockwork Orange.*

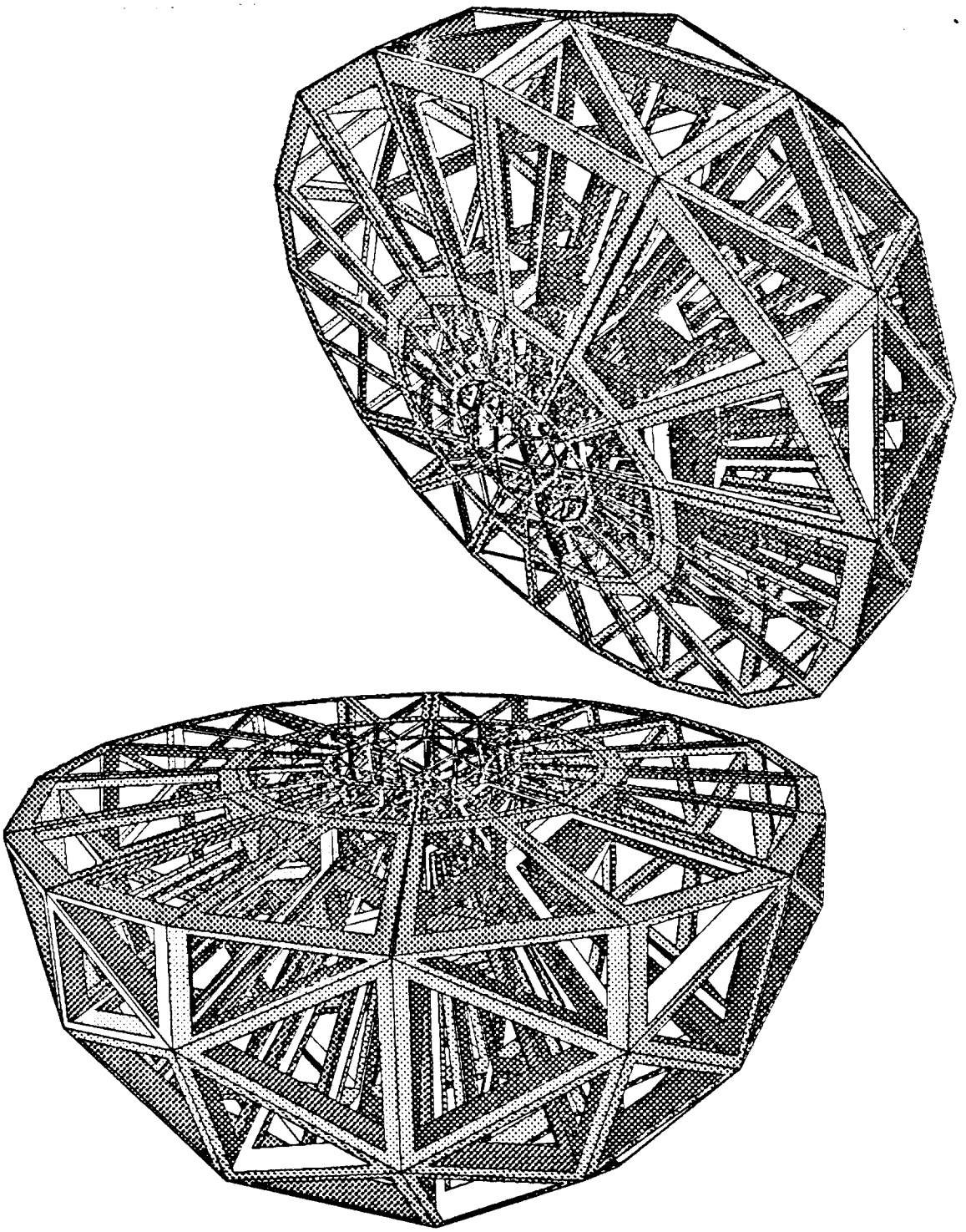


Figure 3.4. *Clockwork Orange.*

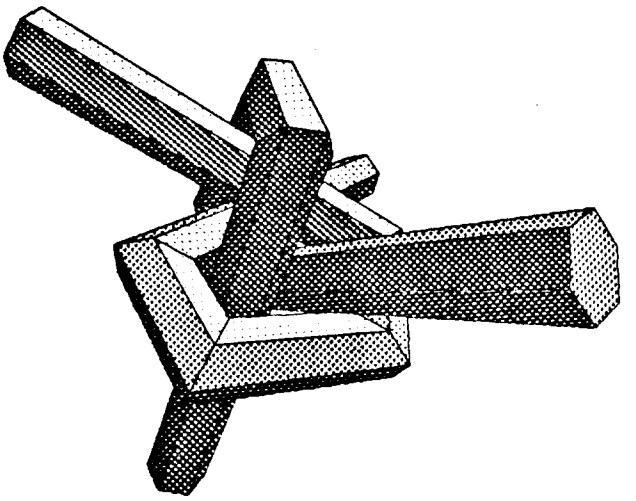


Figure 3.5. *Granny Knot.*

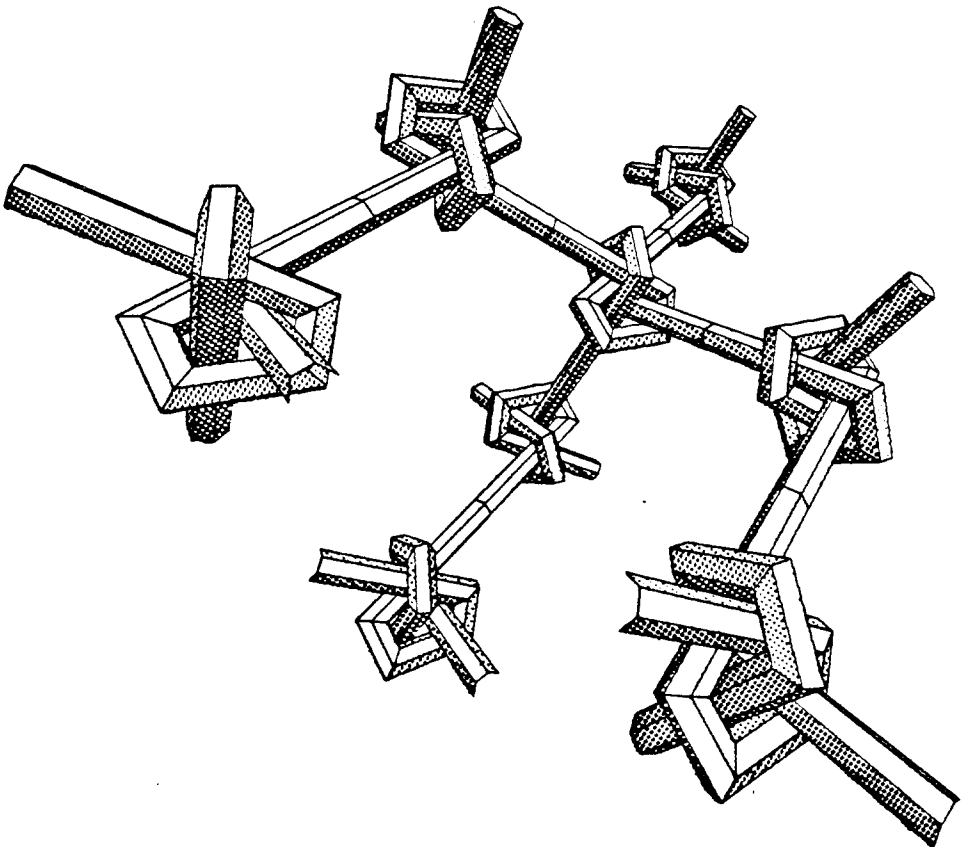


Figure 3.6. *Diamond Cell.*

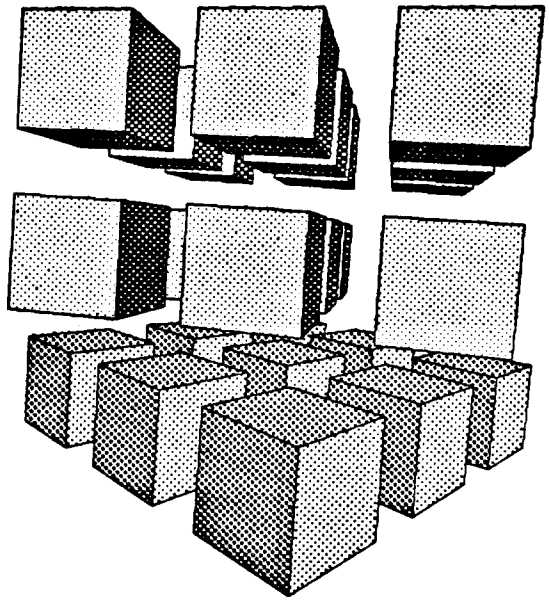


Figure 8.7. *Rubik's Cube.*

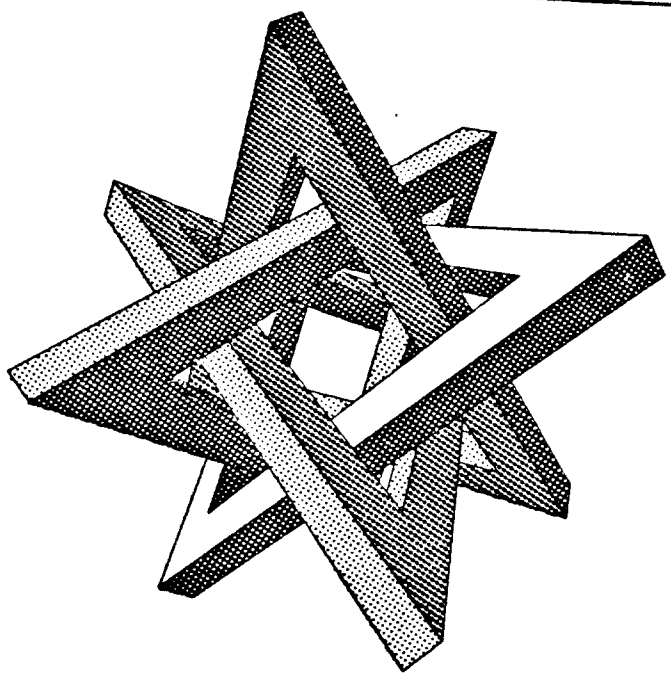


Figure 8.8. *Interlocking Triangles.*

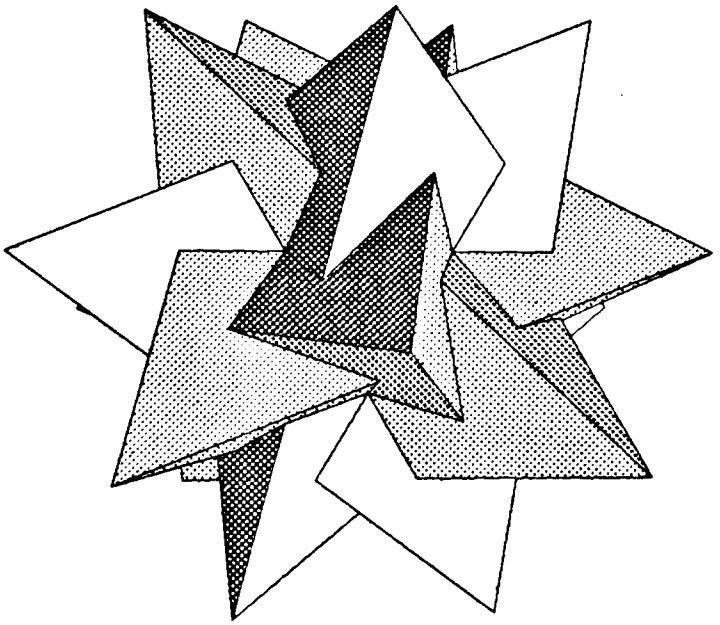


Figure 3.9. *Intersecting Tetrahedrons.*

3.3.1. Effect of the Model Structure on the Tree Size.

Examining table 3.1 we find some interesting results. Although the "Half of a Clockwork Orange" model (Figure 3.3) is a highly non-convex model made of 642 polygons, the size of the resulting tree is about 1.5 times the size of the original data base (when 15 candidates are used in the selection stage). However, the "Granny Knot" model (Figure 3.5), which is made of 64 polygons and seems much simpler then the "Clockwork Orange", results in a tree that is more than 4.6 times the size of the original data base. Even when all the polygons in the half-space are considered as candidates for the cutting polygon, the resulting tree has about the same size.

The "Diamond Cell" model (Figure 3.6), made of eight instances of the "Granny Knot", also exhibits similar results; the size of the BSP tree increases by about the same factor as does the tree for a single knot. The tree size for the "Clockwork Orange" model (Figure 3.4), on the other hand, grows by about the same factor as does the tree for one half of the model. Thus, we find that the structure of a single knot must be the dominant factor in the fragmentation of the scene.

Examining the structure of the "Granny Knot" and the "Half of a Clockwork Orange" models gives some insight as to the nature of these results.

In the "Granny Knot" all polygons are concentrated in a small space with very small distance between them. As the knot is tightly packed, most polygons, if extended by a small amount, will actually intersect a few other polygons in the model. Furthermore, every polygon has, at the most, one other polygon that is parallel to it and its plane cuts many of the other polygons in the model. Therefore, for several levels of the recursion, the plane of any polygon that we pick as the root of the subtree intersects several other polygons, and the resulting half-spaces still contain many conflicts. This behavior is further enhanced by the high degree of symmetry of the knot (it is actually made of four instances of the same model). The fragmentation of a single knot is shown in Figure 3.10, we can see that most of the cuts occur close to the center of the knot.

The "Half of a Clockwork Orange" model, on the other hand, is much more suited for the *binary space partitioning* process. Its outer "shell" can be peeled off without cutting any polygon at all. Many of the inward pointing "beams" are oriented in such a way that their plane cuts only a few polygons. At early stages of the recursion these beams divide the space into separate "slices" eliminating many conflicts at every subdivision. Therefore after just a few levels down the tree there are no more conflicts, and we get low fragmentation of the scene.

We conclude that the behavior of the algorithm on a particular model is affected by more properties than just the "convexity" of the model. We have to examine the separable clusters of polygons from which the model is made (such as the knots in the "Diamond Cell"). Polygons of one cluster cause very little, if any, fragmentation in other clusters. Most of the fragmentation occurs within the cluster. When the model has polygons that can divide it into clusters that exhibit low fragmentation, experimental result indicate that we can expect the size of the BSP tree to be less than twice the size of the original data base. However, when the model has volumes of space that have a concentration of many polygons that conflict with each other, we can expect a high increase in the size of the BSP tree.

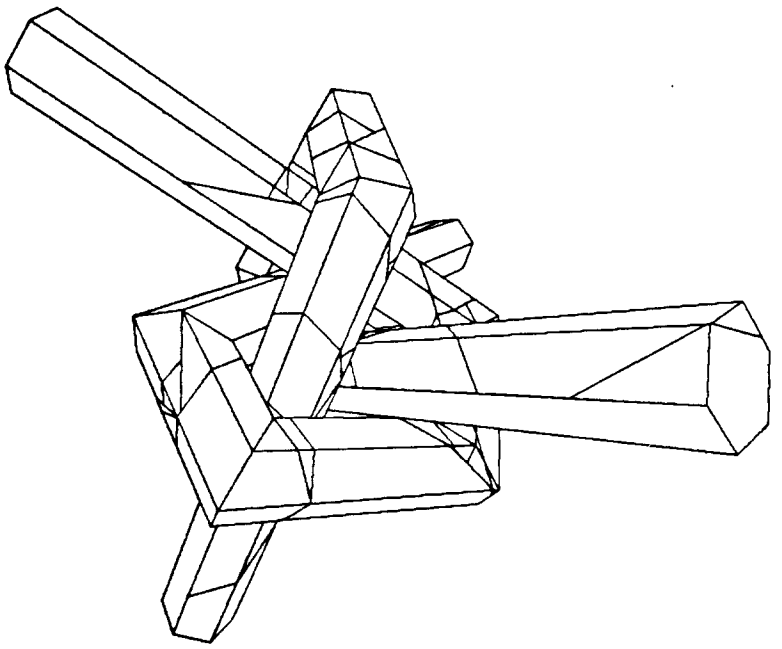


Figure 8.10.
Fragmentation of the model. After being processed by the binary space partitioning algorithm the "Granny Knot" shows a high level of fragmentation. Notice that most of the cuts occur near the center of the knot, where there is a high concentration of polygons.

Fortunately, there are many applications where the models have the properties that are necessary for a relatively low degree of fragmentation of the original data base. As memory is becoming cheaper, the increase in the size of the original data base can be tolerated in these applications. On the other hand, applications that have very large data bases and deal with ill-behaved models (made of densely packed objects that can not be separated into clusters that exhibit low level of fragmentation) are not suited for this algorithm.

4. Cutting an Arbitrary Polygon along a Plane

When constructing the *binary space partitioning* tree, polygons have to be cut along the plane of the root polygon. A UNIGRAFIX model can be made of arbitrary polygons (with concave contours, holes, and even several disjoint parts). The cutting module of the tree construction

program has to handle all such polygons, which is a much harder task than cutting convex polygons. The task can be simplified if the polygons are first decomposed into convex parts. However, this decomposition may increase the size of the data base significantly, resulting in an increase of the BSP construction time. Furthermore, when the display device can handle arbitrary polygons, the increase in the number of polygonal pieces will result in unnecessary increase in the frame generation time. Thus, we decided to implement a module for cutting an arbitrary polygon along a plane. This proved to be the most complicated part in the implementation of the *binary space partitioning* algorithm.

The algorithm used here was derived from the algorithm that was developed by Mark Segal for UGISECT,[13] that eliminates intersections between polygons by cutting every pair of polygons with respect to the other. The algorithm used here is a simplified version of the latter as we have to deal with dividing a polygon along a plane. The basic algorithm is simple:

1) Traverse the contours of the polygon and find the intersections of edges with the plane.

2) Compute the intersection line and sort the edge intersections along this line.

3) According to the above order, create a cut in the polygon between each consecutive even and odd intersections ($i_0 \rightarrow i_1, i_2 \rightarrow i_3, \ldots$).

However, the actual implementation is quite complex. As cuts are made, new contours have to be created and old ones have to be reconnected in the right order. Maintaining correct data structures for the polygons is a delicate task that is prone to error. When an edge has endpoints on different sides of the plane, there is no problem in detecting the intersection. However, when one of the vertices of the edge happens to lie on the cutting plane, it has to be handled as a special case. The polygon might be just touching the plane at this point without going through it, so there is no intersection. On the other hand, it can be a point where the polygon crosses from one side of the plane to the other, so it is an intersection. A third case is when the next vertex lies on the same side as the previous vertex but the angle at this point is concave, here two intersection points have to be created (see Figure 4.1).

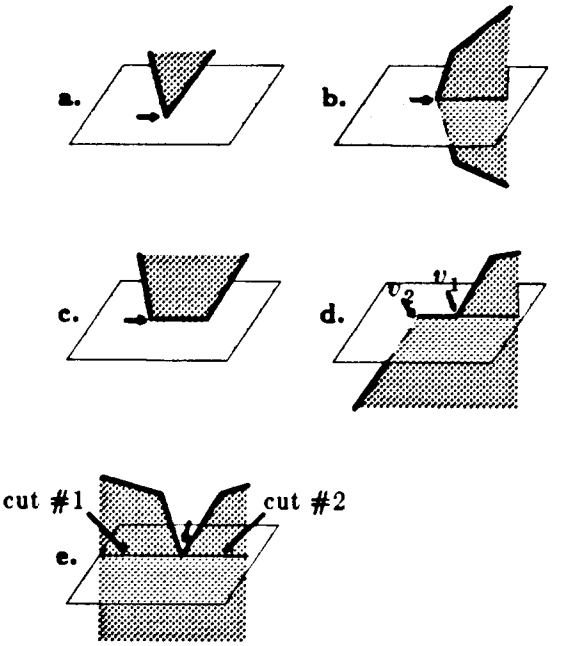


Figure 4.1.

A vertex of the polygon lies on the cutting plane (the polygon has counter clockwise-orientation). **a.** *The polygon just touches the plane - no intersection has to be created.* **b.** *At this vertex the polygon crosses from one side of the plane to the other - one intersection has to be created.* **c.** *The edge lies in the plane but the polygon does not intersect the plane - no intersection has to be created.* **d.** *The edge lies in the plane and the polygon intersects the plane - no intersection point is created at v_1 but one has to be created at v_2.* **e.** *The previous and the next vertices are on the same side of the plane and the inside angle is concave, this is a vertex where two cuts meet - two intersection points have to created.*

To differentiate between these cases, we need to establish the exact geometrical relation between the polygon and the plane, and determine on what side of the plane a vertex lies, as well as the inside angle at a vertex. Limited numerical accuracy results in apparent changes of these relations that may sometimes lead to incorrect results. Vertices that are very close to the cutting plane might be classified as lying on the plane resulting in degenerated geometrical situations (an example is illustrated in Figure 4.2). Substantial effort went into dealing with these problems and trying to minimize their effect. A more detailed discussion of these problems and possible solutions is presented by Mark Segal in (13,14).

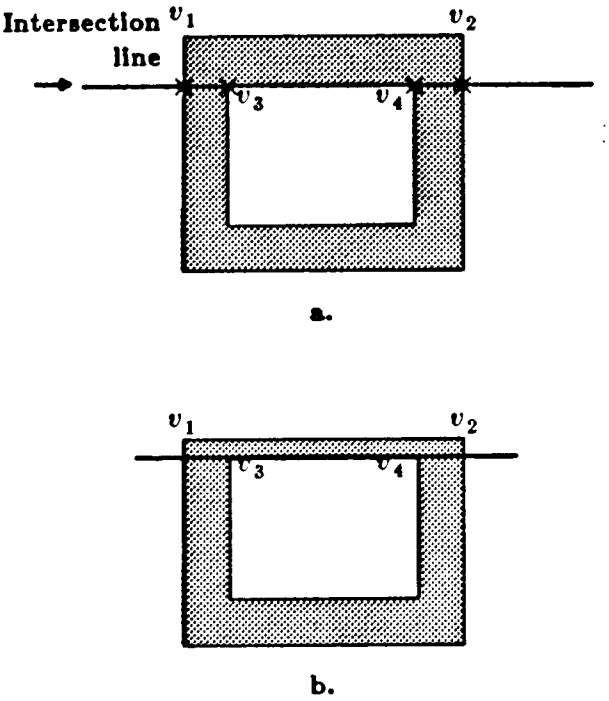


Figure 4.2.

Degeneration of polygons due to numerical inaccuracy. **a.** *The polygon has to be cut along the plane, four intersection points have to be created.* **b.** *Due to limited precision, all four vertices v_1 through v_4 appear to lie on the plane. The outer contour is found to be completely on one side of the plane and no intersections are detected. However, intersection points are detected at v_3 and v_4 resulting in an inconsistent situation for the cutting procedure.*

The *binary space partitioning* algorithm provided us with many test cases for the correctness of the polygon cutting algorithm. As a wide variety of relative constellations between a polygon and the cutting plane are encountered, errors in the cutting routine became readily apparent. Finding the cause of these errors, however, was often a less readily achievable task.

It turned out that the effort put into this part of the program also benefited the algorithm used by UGISECT. The variety of special cases, that have to be dealt with to be able to construct the *binary space partitioning* tree, provided us with insight into some of the special cases the algorithm used in UGISECT have to deal with. Thus, we could enhance (and sometimes fix) the later to deal with these cases.

5. Decomposing Arbitrary Polygons into Convex Parts

UNIGRAFIX tolerates arbitrary polygons (with concave contours and holes) as part of the world model. The Silicon Graphics IRIS workstation, used for displaying the *binary space partitioning* tree, can handle only convex polygons. Thus, arbitrary polygons must be decomposed into convex parts. This can be done either before or after the construction of the BSP tree. Decomposing the polygons before the *binary space partitioning* algorithm would simplify the polygon cutting routine of the algorithm but would significantly increase the time spent in testing a large number of polygons against the cutting plane. As this test is the most time consuming part of the algorithm, we decided to perform the convex decomposition *after* the BSP tree has been built.

To keep the size of the data base small, we looked for a convex decomposition algorithm that produces a small number of convex parts and uses existing vertices only. Another criterion in selecting the algorithm was its simplicity and speed. We were willing to accept an increase in the number of convex parts for enhanced speed. All the fast algorithms for this task produce nearly optimal solutions[15] (correct to within a constant factor). As all algorithms presented in the literature deal with simple polygons*, one has either to extend these algorithms to deal with holes (which often proves to be a complicated extension), or first decompose the polygon into simple polygons. We decided to take the second approach and used Greene's algorithm,[15] which decomposes monotone polygons into convex parts, because of its simplicity and relative speed.

The complete convex decomposition algorithm is divided into two stages. In the first stage (described in section 5.1) the polygon is decomposed into monotone parts; in the second stage (described in section 5.2) these polygons are decomposed into convex parts.

5.1. Decomposing Arbitrary Polygons into Monotone Parts

Let P be a simple *2D* polygon* with n vertices whose contour is $[\, v_0, v_1, \ldots, v_{n-1}]$, (v_i is a vertex of the polygon). Assume that no two vertices have the same Y coordinate. Let v_i be the vertex with the largest Y coordinate and let v_j be the vertex with the smallest Y coordinate.

* A simple polygon is made of one contour (no holes).

Then P is a *monotone* polygon in Y if and only if the sequence $[\ v_i, v_{i+1}, \ldots, v_j\ ]$ is monotone decreasing in the Y coordinate and the sequence $[\ v_j, v_{j+1}, \ldots, v_i\ ]$ is monotone increasing in the Y coordinate (the indices of the vertices are computed modulo n). The algorithm described below, which decomposes an arbitrary polygon into monotone parts, was suggested by T. Asano.[16] It is an extension of the algorithm presented in (17) for the decomposition of a simple polygon into monotone parts.

If we classify the vertices of a polygon according to the six classes presented in Figure 5.1, then it can be shown[18] that the polygon is *monotone* in Y if and only if it contains no vertex of type *UpConcave* or *DownConcave*. Therefore, to decompose a polygon into monotone parts it suffices to make cuts that eliminate vertices of these types. Such vertices can be eliminated by cuts that connect a *DownConcave* or *UpConcave* vertex to a visible† vertex below or above it respectively. These cuts do not intersect the edges of the polygon; if we also make sure that they do not intersect each other, then no new vertices will be introduced, and the resulting parts will be monotone. The algorithm described below performs such cuts.

The algorithm is based on a scan-line approach. The process is performed in the plane of the polygon. The scan-line sweeps from vertex to vertex in order of decreasing Y coordinates‡. Notice that the scan-lines that pass through an *UpConcave* vertex and a visible neighbour above it that is closest to it in the Y direction form a trapezoid with the edges immediately to the left and right of these vertices. The same applies to the scan-lines that pass through a *DownConcave* vertex and a visible vertex below it that is closest to it in the Y direction. If we keep track of these trapezoids, we can readily locate the nearest visible neighbour, which is necessary for performing the cuts. An *Active Edge List* of edges that intersect the current scan-line is used to keep track of these trapezoids. Each edge in the list points to the vertex that defines the top of the "current trapezoid" for that edge. The current scan-line forms the base of the "current trapezoid" for the edge. When an *UpConcave* vertex is encountered, a cut to the nearest visible vertex above it is

† Two vertices are *visible* to each other if they can be connect by a line segment that lies inside the polygon.
‡ Vertices with the same Y coordinate are processed in order of increasing X coordinate. This is equivalent to rotating the polygon by a very small angle clockwise to eliminate any equal Y coordinates.

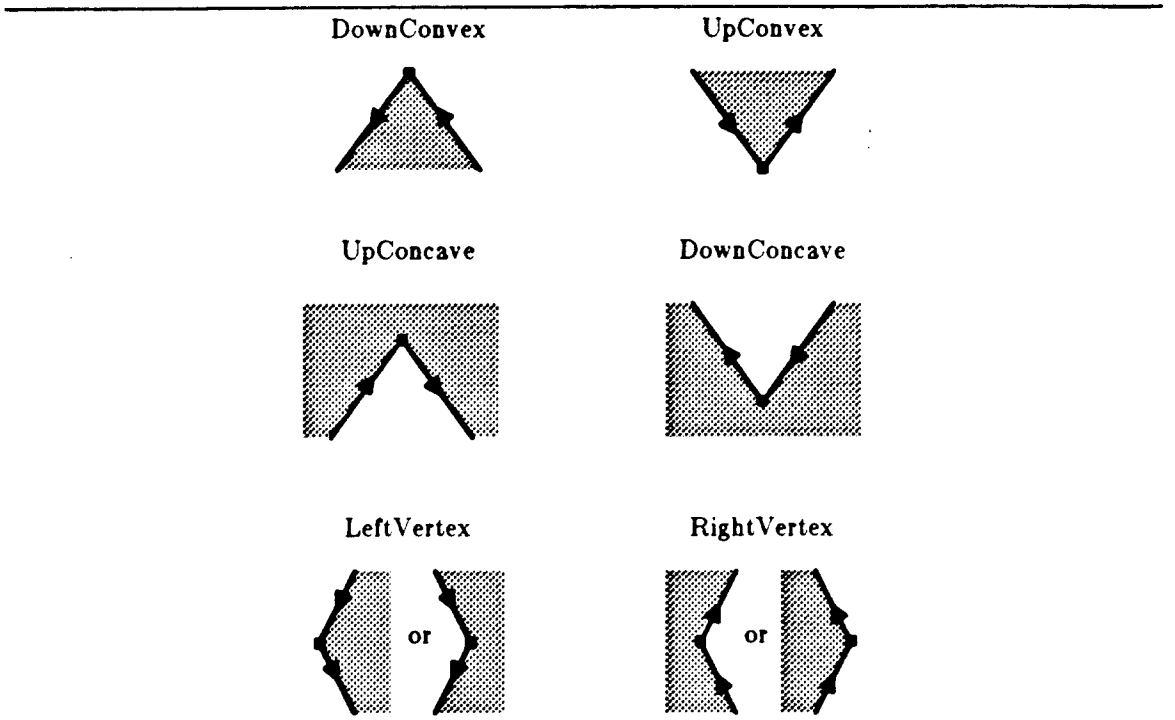
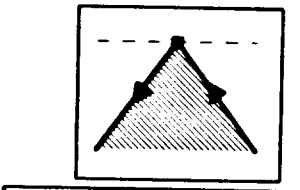


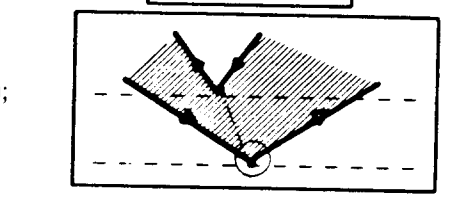
Figure 5.1. *Vertex classification for the monotonizing algorithm.*

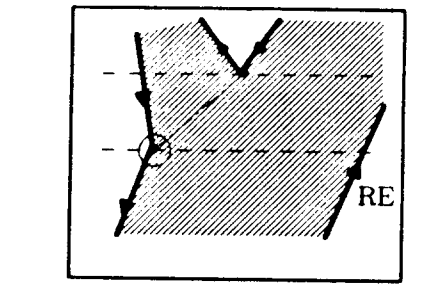
performed immediately. When a *DownConcave* vertex is encountered, we record it on the edges that are part of the trapezoid whose top is the current scan-line. When the vertex whose scan-line forms the bottom of this trapezoid is later encountered, a cut is made to that vertex.

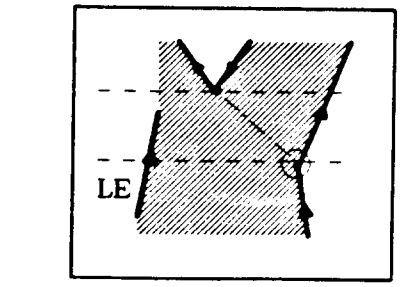
This algorithm may, in some cases, not decompose the polygon into the optimal number of monotone parts. However, it is simple, fast and yields a close to optimal decomposition in all cases.

```
Procedure MonotonizePolygon(Polygon)

  /*
   * Polygon has counter-clockwise orientation.
   *
   * AEL is short for Active Edge List.
   *
   * Procedures that maintain the AEL -
   *   • InitializeAEL() - Creates an empty AEL.
   *   • ReplaceInAEL(e1,e2) - Replaces edge e1 in the AEL with edge e1.
   *   • InsertTwoInAEL(e1,e2) - Searches for the correct location for e1
   *     in the AEL and then inserts e1 followed by e1 into the AEL.
   *   • DeleteTwoFromAEL(e1,e2) - Deletes the pair e1 followed by e2 from the AEL.
   */

begin
  VertexArray := SortByDecreasingY(Polygon);
  ClassifyVertices(VertexArray);
  InitializeAEL();
  for Vertex in VertexArray do
    begin
      InComingEdge := Vertex.InComingEdge;
      OutGoingEdge := Vertex.OutGoingEdge;
      case Vertex.Type of

        DownConvex:
          InsertTwoInAEL(OutGoingEdge,InComingEdge);
          InComingEdge.TopVertex := Vertex;
          OutGoingEdge.TopVertex := Vertex;
          InComingEdge.HasDownConcave := false;
          OutGoingEdge.HasDownConcave := false;

        UpConvex:
          DeleteTwoFromAEL(InComingEdge,OutGoingEdge);
          if Vertex.InComingEdge.HasDownConcave then
            MakeCut(Vertex,InComingEdge.TopVertex);

        LeftVertex:
          RightEdge := InComingEdge.NextInAEL;
          OutGoingEdge.TopVertex := Vertex;
          RightEdge.TopVertex := Vertex;
          OutGoingEdge.HasDownConcave := false;
          RightEdge.HasDownConcave := false;
          ReplaceInAEL(InComingEdge,OutGoingEdge);
          if InComingEdge.HasDownConcave then
            MakeCut(Vertex,InComingEdge.TopVertex);

        RightVertex:
          LeftEdge := OutGoingEdge.PreviousInAEL;
          InComingEdge.TopVertex := Vertex;
          LeftEdge.TopVertex := Vertex;
          InComingEdge.HasDownConcave := false;
          LeftEdge.HasDownConcave := false;
          ReplaceInAEL(OutGoingEdge,InComingEdge);
```

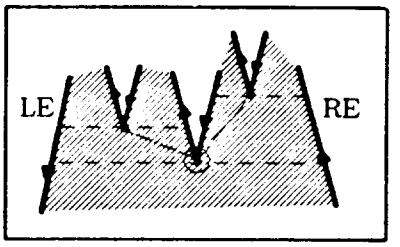


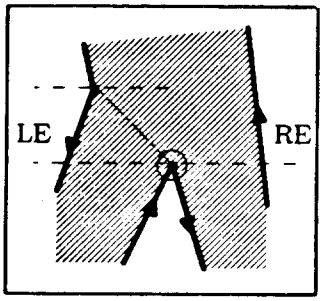






```
        if OutGoingEdge.HasDownConcave then
          MakeCut(Vertex,OutGoingEdge.TopVertex);

      DownConcave:
        LeftEdge := OutGoingEdge.PreviousInAEL;
        RightEdge := InComingEdge.NextInAEL;
        LeftEdge.TopVertex := Vertex;
        RightEdge.TopVertex := Vertex;
        LeftEdge.HasDownConcave := true;
        RightEdge.HasDownConcave := true;
        DeleteTwoFromAEL(OutGoingEdge,InComingEdge);
        if LeftEdge.HasDownConcave then
          MakeCut(Vertex,LeftEdge.TopVertex);
        if RightEdge.HasDownConcave then
          MakeCut(Vertex,RightEdge.TopVertex);

      UpConcave:
        InsertTwoInAEL(InComingEdge,OutGoingEdge);
        LeftEdge := InComingEdge.PreviousInAEL;
        RightEdge := OutGoingEdge.NextInAEL;
        MakeCut(Vertex,LeftEdge.TopVertex);
        InComingEdge.TopVertex := Vertex;
        OutGoingEdge.TopVertex := Vertex;
        LeftEdge.TopVertex := Vertex;
        RightEdge.TopVertex := Vertex;
        InComingEdge.HasDownConcave := false;
        OutGoingEdge.HasDownConcave := false;
        LeftEdge.HasDownConcave := false;
        RightEdge.HasDownConcave := false;
    end /* Case */
  end /* For */
end /* Procedure */
```





5.2. Decomposing Monotone Polygons into Convex Parts

The algorithm presented here is by D. Greene.[15] It is an extension of an algorithm for triangulating monotone polygons, which is presented in (18). This algorithm basically groups the triangles that would have been produced by the latter algorithm into the largest convex polygons that can be constructed from these triangles. The number of convex parts produced by this algorithm is within four times the smallest possible number of convex parts for a monotone polygon.

The algorithm sweeps through the vertices of the monotone polygon in the Y direction from top to bottom, in a similar fashion to the algorithm discussed in section 5.1. Figure 5.2 shows the basic data structures maintained by the algorithm. The *deque* is a collection of vertices whose angles are concave. The *front* and *rear chains* hang from front or rear of the *deque* respectively.

All the vertices of these chains have convex angles. They represent the outer boundary of the convex parts being constructed (connecting the ends of either chain results in a convex polygon). Cuts are always made from the head of the *front chain* to the front of the *deque*, or from the head of *rear chain* to the vertex that is second to the rear of the *deque*. When cuts are made, convex parts are pruned off the structure.

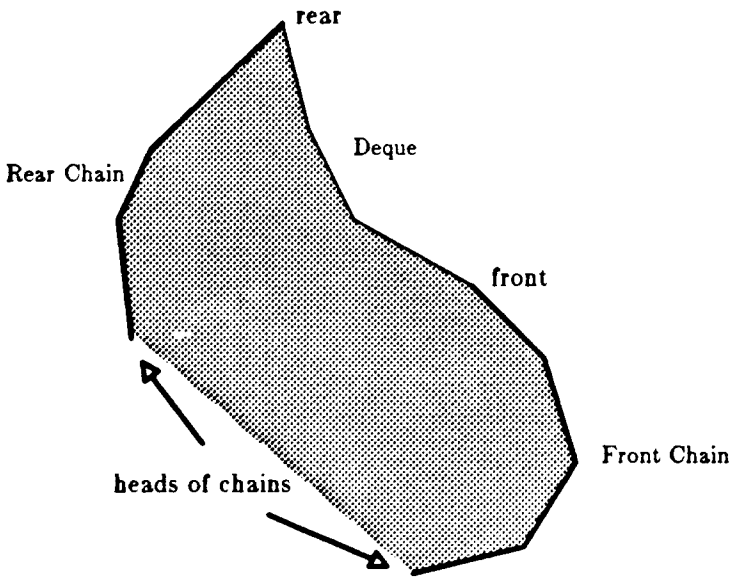


Figure 5.2.

Basic data structures used by the monotone to convex decomposition algorithm.

When the sweep encounters a vertex that is adjacent either to the front or the rear of the *deque*, it starts a new *front/rear chain*.

When the new vertex is adjacent to the *front chain*, the angle at the new vertex is tested. If the angle is convex, the new vertex becomes the head of the *front chain* and the sweep continues. If the angle is concave, then cuts are made from the new vertex to the elements of the *deque* starting at the second vertex from the front, working backwards through the *deque* until either:

- The angle at the new vertex becomes convex, so the new vertex can start a new *front chain* (Figure 5.3.a).

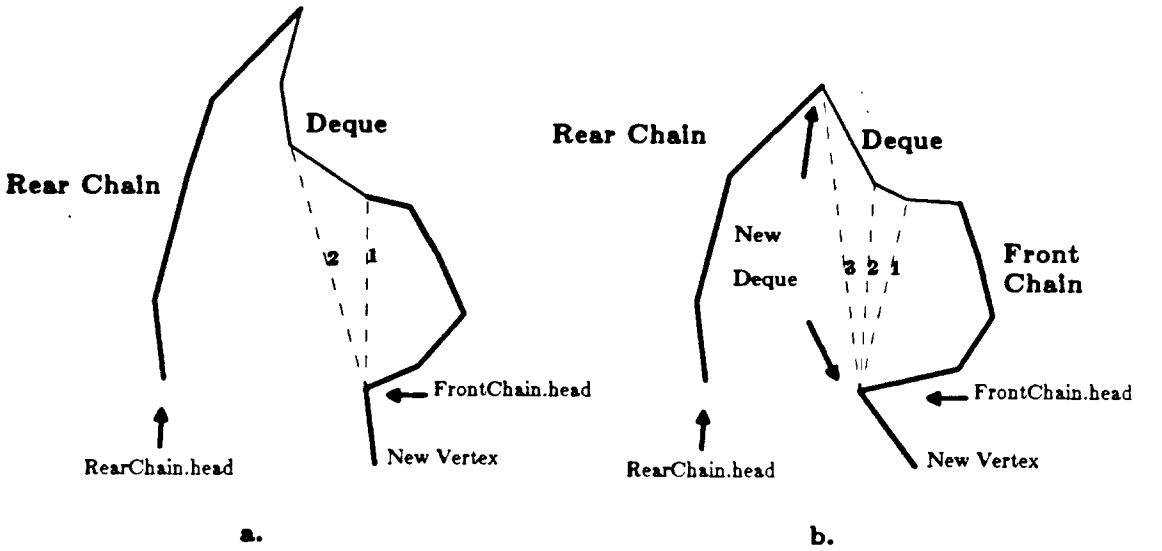


Figure 5.3.

Concave angles at the front chain. **a.** *Cuts are made to the deque until the angle becomes convex, or* **b.** *the angle at the connection to the degue is concave.*

- The angle at the cut with the *deque* is concave, or there is only one vertex left in the *deque* (Figure 5.3.b). In either of these cases the new vertex is added to the front of *deque*.

When the sweep encounters a vertex adjacent to the *rear chain*, it will become the new head of this chain. We have to ensure that a cut from the new vertex to the second vertex from the rear of the *deque* will produce a convex polygon. This property can be maintained if the new vertex is visible to the vertex at the rear of the *deque**. If the new vertex is not visible to the rear of the *deque*, the algorithm performs cuts from the head the *rear chain* to vertices in the *deque* working from the second vertex from the the rear of the *deque* forward, until the vertex at the rear of the *deque* is visible to the new vertex (Figure 5.4.a). In some cases, a cut to the *deque* may

* Note that we can test if the new vertex is visible to the vertex at the rear of the *deque* by testing if the angle < new vertex → head of *rear chain* → rear of *deque* > is convex.

results in a concave angle at the *deque*. Here, the vertex at the head of the *rear chain* is put at the rear of the *deque* (Figure 5.4.b). Another special case is when the *deque* is reduced to one vertex (Figure 5.4.c). Here, the head of the *rear chain* and the front of the deque form a new deque. Of these two, the one with the smallest Y coordinate becomes the front of the *deque*. When the later is the head of the *rear chain*, the *front chain* becomes the *rear chain* and the *front chain* becomes empty. Here, the *deque* also switches from being on one side of the polygon (left or right) to the other.

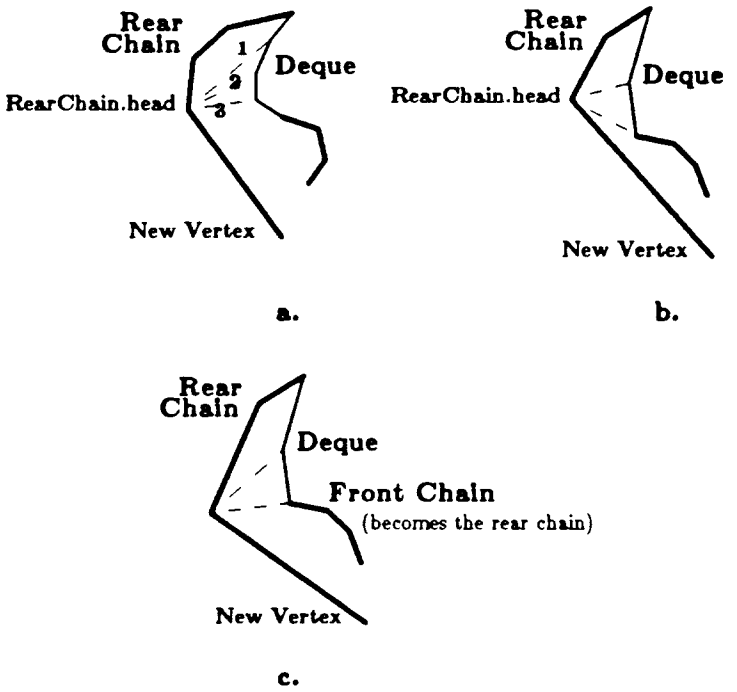


Figure 5.4.

Resolving visibility problems at the rear chain, **a.** *Cuts are made to the deque until the rear of the deque is visible to the new vertex (the numbers represent the sequence of the cuts).* **b.** *A cut results in a concave angle at the deque; the head of the rear chain is put at the rear of the deque.* **c.** *The deque is exhausted; the head of the rear chain and the front of the deque form a new deque. As here the head of the rear chain has smaller Y coordinate, it becomes the front of the deque and the front of the deque becomes its rear. This switches the front chain to be the rear chain and also moves the deque from the right side of the polygon to the left side.*

After the last operation, the new vertex may become adjacent to the front of the *deque* (this happens when the head of the *rear chain* becomes the front of the *deque*). When this happens, the new vertex will start a new *front chain*. If this is not the case, the new vertex becomes the head of the *rear chain*, and we test the angle at the new vertex. If it is concave, we make cuts to the *deque*, marching from the second vertex from the rear forward until the angle at the new vertex becomes concave (Figure 5.5.a). This may leave only one vertex in the *deque*, in which case the old *deque* front and the new vertex start a new *deque*, with the new vertex at the front of the *deque*.

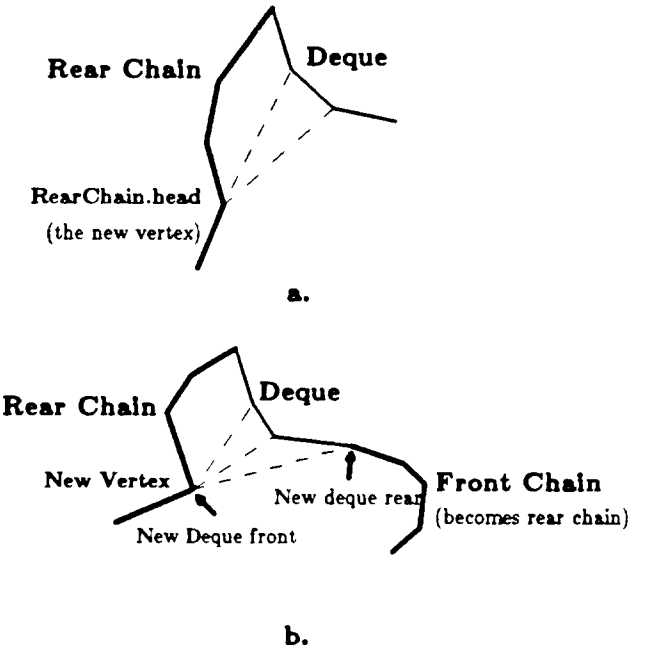


Figure 5.5.

Concave angles at the rear chain. **a.** *Cuts are made to the deque until the angle becomes convex.* **b.** *The deque is exhausted; the new vertex becomes the front of the deque and the front chain becomes the rear chain. The deque switches from the right side of the polygon to the left side.*

As the old *deque* front became its rear, the old *front chain* becomes the *rear chain* and the *front chain* becomes empty (Figure 5.5.b). Here also, the *deque* moves from one side of the polygon to the other. The complete algorithm is described below.

```
Procedure MonotoneToConvex(MonotonePolygon)

begin
  VertexArray := SortByDecreasingY(MonotonePolygon);
  InitializeDQ();
  AppendToFront(VertexArray[0]);
  AppendToFront(VertexArray[1]);
  FrontChain := RearChain := [];

  for Vertex := third to last of VertexArray do
  begin
    if Vertex is adjacent to RearChain.Head then
      Visible(Vertex);

    if Vertex is the last vertex in VertexArray then
      Make cuts from Vertex to all vertices in the
        interior of Deque and quit;

    if Vertex is adjacent to Deque.Front then
      if angle at Deque.Front is concave then
        AppendToFront(Vertex);
      else
        HandleFrontChain(Vertex);
    else if Vertex is adjacent to FrontChain.Head then
      HandleFrontChain(Vertex);
    else
      HandleRearChain(Vertex);
  end /* For Vertex Loop */
end /* Procedure MonotoneToConvex */


Procedure Visible(Vertex);

begin
  while rear of deque is not visible to Vertex do
  begin
    PopRear();
    MakeCut(RearChain.Head,Deque.Rear);
    if Deque has only one element in it then
    begin
      Form new Deque from Deque.Front and RearChain.Head;
        /* The vertex with the smallest Y coordinate
          is put at the front. */
      if RearChain.Head became Deque.Front then
        Swap FrontChain with RearChain; /* Front becomes empty */
      return;
    end /* If one in deque */
  end /* While */
end /* Visible */
```

```
Procedure HandleFrontChain(Vertex);
begin
  while true /* Exit from the loop is by return */
  begin
    if angle at Vertex is convex then
    begin
      make Vertex head of FrontChain;
      return;
    end
    PopFront();
    MakeCut(Vertex,Deque.Front);
    if only one vertex left in the deque or
        angle at front of deque is concave then
    begin
      AppendToFront(Vertex);
      FrontChain := [];
      return;
    end;
  end /* While */
end /* Procedure */


Procedure HandleRearChain(Vertex);
begin
  while true /* Exit from the loop is by return */
  begin
    if angle at Vertex is convex then
    begin
      make vertex head of RearChain;
      return;
    end;
    PopRear();
    MakeCut(Vertex,Deque.Rear);
    if only one vertex left in deque then
    begin
      AppendToFront(Vertex);
      RearChain := FrontChain;
      FrontChain := [];
      return;
    end;
  end /* While */
end /* Procedure */
```

5.3. Decomposing Arbitrary Polygons into Triangular Parts

The effort put into the decomposition of arbitrary polygons into convex parts had side benefits as well. This module was made into an independent program called UGTESS that decomposes arbitrary polygons into monotone, convex or triangular parts. The decomposition into triangular parts was a simple addition to this module, as the monotone to convex algorithm is an

enhancement of a monotone to triangles decomposition algorithm. Using the data structures and utility procedures (for making cuts, testing of convexity etc.) designed for the monotone to convex decomposition algorithm, it was a simple task to implement the triangular decomposition algorithm described in (18). UGTESS is a valuable tool as a filter for other programs where simple polygons or triangles are easier to handle (e.g for Gouraud shading in UGDISP[19]), or are required (e.g for the smoothing operation by UCI[20]).

6. Interactive User Interface on the IRIS

An interactive user interface is used for displaying the scene described by the BSP tree. This interface, implemented on the IRIS workstation, allows the user to vary the viewing parameters while viewing the resulting image.

The display program reads a special format ascii file that describes the BSP tree and the illumination sources, reconstructs the BSP tree and creates the graphical objects for the IRIS display list, then it invokes the interactive user interface module. This module goes into a loop where it processes input from the user, sets up the viewing transformation and the corresponding viewpoint, and invokes the tree traversal module for generating the new frame.

The method of communicating with the user and the setting of the viewing transformation is independent of the tree traversal module. Thus, it is fairly easy to experiment with different types of user interface paradigms. Of the ones that we have experimented with, we found two that are easy to use and give the user intuitive control in manipulating the scene. One interface uses a *flight* paradigm and the other one uses a *crystal ball* paradigm.

6.1. The Flight Interface

In this interface the user gets the feeling of being the pilot of a weightless helicopter that he can move through the model. The mouse serves as a rudder controlling the pitch (by y motion of the mouse) and the yaw (x motion). Roll is controlled by keyboard buttons. Forward and

backward motion is achieved by holding down the left (forward) and right (backwards) mouse buttons, and horizontal and vertical motion is accomplished by pressing the keypad buttons. The speed, the roll increment, and viewing angle can be adjusted interactively by the user.

We found that this interface is suitable for examining the model in detail. This paradigm provides intuitive feeling of traveling through the model and looking around. Thus, the user gets good perception of the structure of the model, and can easily move himself closer to model details that he wishes to examine. However, this paradigm is not that useful for selecting a good set of viewing parameter for producing hard-copy images of the model. Here, the user basically controls the movement of the eye coordinate system with respect to the world. Therefore, positioning the model at a certain orientation with respect to the viewer has to be achieved by the inverse operation, which is positioning the viewer with respect the object. This is an unintuitive operation that involves indirect movements and is hard to perform.

6.2. The Crystal Ball Interface

Here, the interface paradigm is one of enclosing the model in a transparent sphere with infinite radius, placing the model at an arbitrary location with respect to the center of the sphere. By rotating the sphere around its center, the user manipulates the model's orientation. The viewer is assumed to be at a position that does not change with the motion of the sphere, looking towards the center of the sphere. To be able to determine the exact position of the model with respect to the center of the sphere, the viewer can switch between two positions that are in the same plane but 90 degrees apart around the center of the sphere (this is equivalent to looking either down the Z axis or down the X axis).

The amount of x and y rotation of the sphere is directed by the movement of the mouse, which is reflected in the position of a cursor on the screen. The position of the cursor with respect to the center of the screen specifies the angle of rotation in either direction. Moving the cursor in the y/x direction result in a rotation around the x/y axis in the same direction as that of the cursor. As the cursor gets further away from the center, the angle of rotation increases quadratically

(this provides better control of the rotation angle). Rotation around the Z axis is accomplished by holding down keyboard buttons. This rotation is by a fixed angle that can be adjusted interactively. Lateral movements of the model inside the sphere are specified by holding down buttons. The left and right buttons of the mouse control the movement along the Z axis, and four keypad keys control the movements along the X and Y axes*. The size of these movements can be adjusted interactively. The viewer's distance from the center of sphere, as well as the viewing angle are also adjusted incrementally from the keyboard.

We found that this interface is particularly useful for positioning the model with respect to the viewer. The intuitive feeling provided by this interface is of holding the model and manipulating it in different directions. Thus, this is a good interface for picking viewing parameters for generating hard-copy images, and examining the overall appearance of the model. On the other hand, it does not provide the capabilities of the flight interface. Positioning the viewer at certain locations of the model, and looking around from that location to examine details of the model, is not easy. Here, the object has to "come to the viewer" rather then the viewer "going to the object", which is unintuitive and therefore cumbersome.

This interface method is actually a simplification of a more elaborate method we have experimented with. In the later, the surface of the screen is considered to be an imaginary tracker ball that is being manipulated by the motion of the mouse. The cursor serves as a finger that moves the ball around its center, in the same manner as a human manipulates a regular tracker ball. This scheme needs immediate feed-back to the motion of the mouse to make it really intuitive. The cursor location has to be sampled at a high rate to be able to follow the user's instructions. When the model is large, the relatively long frame generation rate prevents immediate feedback. Hence, we picked the simpler version described above.

* The three axes are fixed independent of the movement of the sphere. The X/Y plane is parallel to screen with the negative Z axis going into the screen. (Or we may switch to Y/Z plane parallel and X going into the screen).

6.3. Dealing with Relatively Long Image Generation Times

In case where the BSP tree for the model is bigger than 500 polygons, the frame generation rate is less than that required for for smooth animation. When this happens, manipulating the model becomes tiresome. Here, the user can get much better response using a wire frame option for fast redisplay. Wire frame images may, at times, be ambiguous or misleading. However, switching back and forth between wire frame and hidden surface mode, provides fast interaction while perception of the the model is maintained.

7. Conclusion

The combination of the *binary space partitioning* algorithm with the fast transformations provided by the IRIS workstation provided us with a tool for previewing and examining UNIGRAFIX models. Because of the high level of interactivity provided by this tool, it is easy to select a "good" viewing parameters for generating the final hardcopy pictures. The ease of moving through a model proved to be useful in examining the structure of the model and the relation between its components. Examining the model for features such as exact object placement and intersections, which is difficult using the other renderers, becomes easy when using the *binary space partitioning* display programs.

There are some weaknesses of the *binary space partitioning* algorithm that we hope to overcome by future programs of the same nature. The main weakness is the inability to handle changes in the model without reconstructing the complete tree. Thus, it can not be used in an interactive manner for constructing a model or even just moving existing objects in the model. Another weakness is the inability to handle all the models that can be rendered by the other batch oriented UNIGRAFIX renderers. Such models are either made of a large number of polygons to begin with, or their structure leads to a high level of fragmentation during tree construction. Here, the size of *binary space partitioning* tree exceeds the memory limit imposed by the system.

Overall, the efforts in creating this program resulted in a few good tools that enhanced the UNIGRAFIX environment. This preview program provides a counterpart to the interactive editor JESSIE,[10] which was also recently added to the UNIGRAFIX environment. Thus, we now have the interactive capabilities that have been so conspicuously missing in the UNIGRAFIX environment.

Acknowledgements

I would like to thank the people who helped me in this work. Mark Segal provided valuable help in the design and implementation of the polygon cutting routines. Dr. Tetsuo Asano suggested the algorithm for decomposing arbitrary polygons into monotone parts and pointed out Greene's algorithm. I would also like to thank my advisor Carlo Séquin who provided support and guidance throughout the course of this work.

Most of all, I would like to thank my wife Nirit who made many sacrifices and provided me with much needed support throughout the time I spent on this work.

This work was supported in part by the State of California Microelectronics Innovation and Computer Research Opportunities program, and the Semiconductor Research Corporation.

References

1. H. Fuchs, Z.M. Kedem, and B.F. Naylor, "On Visible Surface Generation by A Priori Tree Structures," *Computer Graphics (Proc. SIGGRAPH '80)*, vol. 14, no. 3, pp. 124-133, July 1980.

2. H. Fuchs, G.D. Abram, and E.D. Grant, "Near Real-Time Shaded Display of Rigid Objects," *Computer Graphics*, vol. 17, no. 1, pp. 65-69, July 1983.

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

4. I.E. Sutherland, R.F. Sproull, and R.A. Schumacker, "A Characterization of Ten Hidden Surface Algorithms," *ACM Computing Surveys*, vol. 6, no. 1, pp. 1-55, Mar. 1974.

5. J.D Foley and A. Van Dam, in *Fundamentals of Interactive Computer Graphics*, Addison Wesley, Reading, Mass., 1982.

6. G. Hamlin and C.W. Gear, "Raster-Scan Hidden Surface Algorithm Techniques," *Computer Graphics*, vol. 11, no. 2, pp. 2206-213, Summer 1977.

7. *IRIS User's Guide.*, Silicon Graphics Inc., Mountain View, CA., 1983.

8. M. Segal, C.H. Séquin, and Paul Wensley, "UNIGRAFIX 2.0 User's Manual and Tutorial," Tech. Report (UCB/CSD 83/161), University of California, Berkeley, December 1983.

9. "Creative Geometric Modeling with UNIGRAFIX," ed. C. H. Séquin, Tech. Report (UCB/CSD 83/162), December 1983.

10. H.B. Siegel, "Jessie: An Interactive Editor for Unigrafix," Master's Report, U.C. Berkeley, Dec. 1985.

11. N. Gal, *UGI - An Interactive Environment for UNIGRAFIX*, University of California, Berkeley, In preparation.

12. M. E. Newell, R. G. Newell, and T. L. Sancha, "A New Approach to the Shaded Picture Problem," *Proc. ACM National Conf.*, 1972.

13. M.G. Segal, "Partitioning Polyhedral Objects into Non-Intersecting Parts," Master's Report (in preparation), U.C. Berkeley, Spring 1986.

14. M.G. Segal, "Consistent Calculations for Solid Modeling"," *Proc. ACM Symposium on Computational Geometry*, June 1985.

15. Daniel H. Greene, "The Decomposition of Polygons into Convex Parts," *Advances in Computing Research*, vol. 1, pp. 235-259, JAI Press Inc., 1983.

16. Tetsuo Asano, *Personal communication*, Summer 1984.

17. D. T. Lee and F. P. Preparata, "Location of a Point in a Planar Subdivision and its Applications," *SIAM J. Comput.*, vol. 6, no. 3, pp. 594-606, 1977.

18. M. R. Garey, D. S. Johnson, F. P. Preparata, and R. E. Tarjan, "Triangulating a Simple Polygon," *Information Processing Letters*, vol. 7, no. 4, pp. 175-179, June 1978.

19. N. Gal, "Hidden Feature Removal and Display of Intersecting Objects in UNIGRAFIX," Master's Report, University of California, Berkeley, Dec. 1985.

20. L. Longhi, "Interpolating Patches Between Cubic Boundaries," Master's Report, U.C. Berkeley, Dec. 1985.

Appendix I - Manual Pages

- **Ugbsp** - Creates a Binary Space Partion tree from UNIGRAFIX scene description.
- **BspBall,bspFly** - Interactive display programs for the *binary space partition tree.*

NAME

ugbsp - create a binary space partition tree from UNIGRAFIX scene description.

SYNOPSIS

ugbsp [options, arguments]

DESCRIPTION

Ugbsp is a preprocessor for the *binary space partition* interactive display programs. It reads a UNIGRAFIX scene description and generates a special format ascii file describing the binary space partition tree. The following arguments can be used:

-t *number of tries*
Number of candidates to try when selecting the root face of a subtree. Default is 15.

-nr Do not use the randomizing scheme when selecting the root for a subtree. Instead, all faces of the subtree are tested and the one that cuts the least faces is selected as the root. This option overrides the -t option.

-nt Do not write out the file describing the *binary space partition* tree. Used for gathering statistics and creating UNIGRAFIX descriptions of the scene after the *binary space partition* algorithm was run (to show the fragmentation of polygons).

-nc Do not decompose the faces of the *binary space partition* tree into convex parts. By default, after the *binary space partition* tree is created, the faces are decomposed into convex parts. This option should be used when the scene is to be displayed on devices that can handle arbitrary polygons or it is desirable not to have extra edges when producing a UNIGRAFIX description of the scene.

-w Write out a UNIGRAFIX description of the faces in the *binary space partition* tree. Convex decomposition of faces (if not disabled by **-nc**) is performed before the scene is written out.

-fi *filename*
Use *filename* as input file. Default is standard input.

-fo *filename*
Write the *binary space partition* tree description into *filename*. Default is standard output.

-fc *filename*
Use *filename* to find command-line options.

-fw *filename*
Write the UNIGRAFIX description of the scene after the *binary space partition* algorithm into *filename*. Implies **-w**. Default file is standard output.

-d{1,2,3}
Debugging option, recognized only when the program is compiled with the DEBUG flag. Turns on the debugging option. The numbers specify increasing levels of verbosity in debugging messages. Apart from the messages, when in debugging mode, the program generates long labels for vertices and faces. These labels make it easier to trace the ancestors of faces and the edge that was cut when a vertex was created.

As the program uses randomized root selection, it may produce trees of varying sizes at different runs. When the scene has a large number of faces or the size of the resulting tree is significantly larger than the original scene, several runs are recommended to try and get better results. Increasing the number of candidates for selecting the root of a subtree increases the computation time but will not necessarily improve the size of the tree. Therefore, experimentation is required here as well.

The cutting procedures as well as the convex decomposition procedure can not handle self intersecting contours, in most cases the program will crash when such contours are encountered.

EXAMPLE

ugbsp -fi infile -fo outfile -t 5

FILES

~ug/bin/ugbsp
~ug/src/ug2/bsp/MakeTree

SEE ALSO

bspBall (UG), bspFly (UG)

DIAGNOSTICS

Prints out the number original number of faces, the number of added faces and the total number of faces in the tree.

AUTHOR

Ziv Gigus

NAME

bspBall – interactive display program for scenes defined by a *binary space partition* tree

SYNOPSIS

bspBall [options] [bsp-tree-file]

DESCRIPTION

BspBall provides an interactive interface on a Silicon Graphics Iris work-station for viewing scenes defined by a *binary space partition* tree. It reads the scene description from *bsp-tree-file* that is a special format ascii file produced by *ugbsp*. *BspBall* gives the user the impression of manipulating the scene by enclosing it in a transparent crystal ball and rotating this ball around its center. The center of the ball is at a fixed location with its X and Y coordinates coinciding with the center of the view window, and the Z coordinate at a given distance from the view point. The viewing direction is fixed toward the center of the ball (this can be thought of as looking down the Z axis of a fixed coordinate system whose origin coincides with the center of the ball). The user can also switch to looking up the X axis of this coordinate system, to check the relation between the scene and the Z coordinate of the center of the ball.

This program is useful for manipulating the scene while viewing it from viewpoint and at a fixed viewing direction. Thus, it is suitable for selecting a set of viewing parameters for producing pleasing hard-copy images. However, moving around a scene in order to examine it in detail is not easy with this interface. **BspFly** is more appropriate for the latter task.

Rotating the Ball

Rotation of the ball around the X and Y axes is controlled by the y/x displacement of the cursor from the middle of the view window (indicated by a small cross). The further away the cursor is from the center, the faster is the rotation. X displacement controls the rotation around the Y axis; being to left/right of the middle of the window results in a left/right rotation of the ball. The Y displacement controls the rotation around the X axis; being above or below the middle of the window results in a rotation upwards or downwards, respectively. The middle button of the cursor zeros the X and Y rotations and moves the cursor to the neutral position at the cross (the middle of the window). **Rotation of the ball around the Z axis** is controlled by the left and right mouse buttons for counterclockwise and clockwise, respectively. The **speed of the rotation around the Z axes** is decreased and increased by the "[" and "]" keys, respectively. Each key push increases or decreases the previous speed by 25% or by 20%, respectively (so the end result of one increase command followed by one decrease command or vice versa is the original speed).

Positioning the Scene Inside the Ball

The scene can be moved inside the ball by translational motions. This motions are always defined with respect to a fixed coordinate system that has one of it axis perpendicular to the screen and the other two axes coinciding with the X and Y axes of the screen. All these motions are controlled by keypad keys. Moving the object along the axes that is perpendicular to the screen is controlled by the "7" and "9" keys for moving the object away and towards the viewpoint ("in and out of the screen"). The "4", "6", "8", and "2" keys control the left and right up and down movements, respectively.

The **speed of the translational motions** of the object is increased and decreased by the "=" and "-" keys, respectively (using the same +25%/-20% method described above for control of the speed of the Z rotation).

Positioning the view point

The **distance of the viewpoint** from the center of the ball is controlled by the "pf1" and "pf3" function keys for moving the viewpoint away or closer to the center of the ball, respectively. Switching between **looking down the Z axis or up the X axis** is controlled by the "'" (single quote) key that serves as a toggle switch.

Control of the Display Parameters

The **viewing angle** can be increased and decreased by the "z" and "x" keys, respectively.

Display of backfaces is switched on and off by the "d" key.

Edge enhancement is toggled by the "e" key. The type of edges that are enhanced is given by command line options (See Options Section).

Display of wire-frame or full hidden surface elimination is toggled by the "l" key. Display of wire frame image is faster than performing hidden surface elimination. This option is particularly useful in large scenes where the redisplay rate with hidden surface elimination is too slow. In wire-frame mode only the original edges of the scene are displayed.

Front and back clipping plane distance can be changed by pressing the "f" and "b" keys. The program prompts the user with the current distance of the relevant parameter and asks for the new value.

The color map can be toggled between a red, green and blue or a grey map by the "m" key.

Miscellaneous

Writing the viewing parameters is preformed when the "w" key is pressed. The user is prompted for the name of the file to print the parameters to. The parameters are written in a UNIGRAFIX renderers command line format. This file can be used directly as command line input to these renderers (using the "command file" option "-fc") to reproduce the picture that is displayed in the image window when the parameters are written. The user should be aware that the current UNIGRAFIX renderers do not perform front and back clipping and thus should avoid trying to render pictures where the eye point is inside the scene.

Help in the form of a short description of the different keys is provided when the "?" key is pressed.

Quitting the program is accomplished by pressing the "q" key.

OPTIONS

The following options are recognized on the command line:

- Use standard input for the bsp-tree-file. When this option is specified and a file is also specified the latter will be used.

-sb When edge enhancement is on, display edges that result from the *binary space partition* process as well as original edges. Default is original edges only.

-sa When edge enhancement is on, display edges that result from the *binary space partition* process and edges that resulted from the convex decomposition as well as original edges. Default is original edges only.

-h Help. Displays the possible command-line options and quits.

FILES

~ug/bin/ugBall
~ug/src/ug2/bsp/ShowTree

SEE ALSO

ugbsp (UG), bspFly (UG), ugplot (UG), ugdisp (UG), ugi (UG).

AUTHOR

Ziv Gigus

NAME

bspFly – interactive display program for scenes defined by a *binary space partition* tree

SYNOPSIS

bspFly [options] [bsp-tree-file]

DESCRIPTION

BspFly provides an interactive interface on a Silicon Graphics Iris work-station for viewing scenes defined by a *binary space partition* tree. It reads the scene description from bsp-tree-file that is a special format ascii file produced by *ugbsp*. *BspFly* gives the user the impression of piloting a massless helicopter moving through the scene and looking around.

This program is useful for examining details of a scene from various viewpoints, moving around inside the scene and getting closer to points of interest in the scene as well as looking at inner hidden structure. However, it is quite difficult to use this program for selecting a set of viewing parameters for producing pleasing hard-copy images. **BspBall** is more appropriate for the latter task.

Motion Control

Changes in the pitch/yaw of the craft are controlled by the y/x displacement of the cursor from the middle of the view window (indicated by a small cross). The further away the cursor is from the center, the faster is the rate of change. X displacement controls the yaw; being to left/right of the middle of the window results in a left/right change in yaw. The Y displacement controls the change in pitch; being above/below the cross raises/lowers the pitch. The middle button of the mouse zeros the pitch and yaw change and moves the cursor to the neutral position at the cross (the middle of the window). **Roll** is controlled by the "<" and ">" keys for counterclockwise and clockwise rotation, respectively. **Forward/backward** motion is controlled by the left/right mouse buttons. Four keypad keys control the **lateral motions**; the "4", "6", "8", and "2" keys control the left and right up and down motion, respectively.

The **speed of the linear motions** (backward, forward and lateral) is increased or decreased by the "=" and "-" keys, respectively. Each key push increases or decreases the previous speed by 25% or by 20%, respectively (so the end result of one increase command followed by one decrease command or vice versa is the original speed). The **angular velocity of the roll** is decreased and increased by "[" and "]" keys, respectively (using the same +25%/-20% method).

Control of the Display Parameters

The **viewing angle** can be increased or decreased by the "z" and "x" keys, respectively.

Display of backfaces is switched on and off by the "d" key.

Edge enhancement is toggled by the "e" key. The type of edges that are enhanced is given by command line options (See Options Section).

Display of wire-frame or full hidden surface elimination is toggled by the "l" key. Display of wire frame image is faster than performing hidden surface elimination. This option is particularly useful in large scenes where the redisplay rate with hidden surface elimination is too slow. In wire-frame mode only the original edges of the scene are displayed.

Front and back clipping plane distance can be changed by pressing the "f" and "b" keys. The program prompts the user with the current distance of the relevant parameter and asks for the new value.

The color map can be toggled between a red, green and blue (fixed different intensity level of each of these colors) or a grey map (that provides as many equally spaced grey levels as the color map can accomadate) by the "m" key.

Miscellaneous

Writing the viewing parameters is preformed when the "w" key is pressed. The user is prompted for the name of the file to print the parameters to. The parameters are written in a UNIGRAFIX renderers command line format. This file can be used directly as command line input to these renderers (using the "command file" option "-fc") to reproduce the picture that is displayed in the image window when the parameters are written. The user should be aware that the current UNIGRAFIX renderers do not perform front and back clipping and thus should avoid trying to render pictures where the eye point is inside the scene.

Help in the form of a short description of the different keys is provided when the "?" key is pressed.

Quitting the program is accomplished by pressing the "q" key.

OPTIONS

The following options are recognized on the command line:

- Use standard input for the bsp-tree-file. When this option is specified and a file is also specified the latter will be used.

-sb When edge enhancement is on, display edges that result from the *binary space partition* process as well as original edges. Default is original edges only.

-sa When edge enhancement is on, display edges that result from the *binary space partition* process and edges that resulted from the convex decomposition as well as original edges. Default is original edges only.

-h Help. Displays the possible command-line options and quits.

FILES

~ug/bin/bspFly
~ug/src/ug2/bsp/ShowTree

SEE ALSO

ugbsp (UG), bspBall (UG), ugplot (UG), ugdisp (UG), ugi (UG).

AUTHOR

Ziv Gigus