Near Real-Time Shaded Display of Rigid Objects

Henry Fuchs
Gregory D. Abram
Eric D. Grant

University of North Carolina
at Chapel Hill

Abstract

Described is a visible surface algorithm and an implementation that generates shaded display of objects with hundreds of polygons rapidly enough for interactive use -- several images per second. The basic algorithm, introduced in [Fuchs, Kedem and Naylor, 1980], is designed to handle rigid objects and scenes by preprocessing the object data base to minimize visibility computation cost. The speed of the algorithm is further enhanced by its simplicity, which allows it to be implemented within the internal graphics processor of a general purpose raster system.

CR Categories and Subject Descriptors: I.3.3 [**Computer Graphics**]: Picture/Image Generation - Display Algorithms; Viewing Algorithms; I.3.7 [**Computer Graphics**]: Three-Dimensional Graphics and Realism - Visible Line/Surface Elimination

1. Introduction

The generation of realistic, colored images of 3D scenes has been a subject of much study for nearly twenty years. Many visible-surface algorithms have been developed for a variety of applications and machine environments (see, eg., [Sutherlánd, Sproull and Schumacker, 1974] or [Foley and Van Dam, 1982]). However, for real-time interactive applications, very expensive special purpose hardware is needed. Even if a much lower image generation rate, of perhaps one or two per second, is acceptable, we know of no (previously published) algorithm which can accomplish this on a general purpose graphics system.

It is not surprising that the generation of rendered color images is computationally expensive. Not only do the usual transformation, clipping and perspective steps need to be performed (the ones independent of the particular type of display or image generation algorithm) but visibility and rendering calculations must also be performed for every pixel in the image. General purpose algorithms may take several seconds (or much longer) to generate these images because they require either many calculations at each pixel or have significant overhead at a higher level in order to minimize the pixel calculations.

• This work was supported in part by NSF grant MCS79-02593 and in part by NIH grant NS/HL 16759-01.

Our own long-term goals are to have real-time 3D images generated in our laboratory, for use in a wide range of interactive applications. Although we are designing special purpose equipment for this [Fuchs, Poulton, Paeth and Bell, 1982], in the immediate future the images have to be generated with our current general purpose graphics system. To generate these images, we are willing to sacrifice update rate, down to even one or two per second. Further, we have found that many of our applications (as well as those of others) have the significantly simplifying property that the world model changes far less frequently than the viewing position [Schumacker, Brand, Gilliland and Sharp, 1969]. Our aim, then, is to cut image generation time by taking advantage of this simplification. We do so by by pushing much of the visible-surface overhead into a preprocessing phase, thereby greatly reducing the overhead at image-generation time. By doing so, we hope to make the generation of realistic, colored images for our class of applications fast enough to be useful in an interactive mode.

2. BSP-Tree Basics

This section briefly reviews the Binary Space Partitioning algorithm (BSP-tree) as introduced in [Fuchs, Kedem and Naylor, 1980].

2.1. Motivation

[Schumacker, Brand, Gilliland and Sharp, 1969] introduced the notion that the image generation can be simplified in situations where the world model changes less frequently than the viewpoint or direction of view of the observer. Many applications have this property: a biochemist studying a complex molecule, a physician examining an anatomical structure for signs of disease, an architect (or her client) walking through a planned house or subdivision, an engineer designing a mechanical part. However, the Schumacker approach's dependence on manual intervention in the building of the internal data structure made it difficult (and time consuming) to generate new databases, and thus limited its general usefulness.

Although the current implementation of the BSP-tree algorithm is limited to static world models (since whenever the world model changes, the preprocessing data restructuring step must be invoked) we hope to ease this restriction in the future.

In order for the algorithm to run fast, the entire BSP-tree should be in local memory. Although this is not an inherent restriction in the algorithm, the overall performance would be significantly degraded if parts of the tree had to be swapped from backing store. Our implementation (detailed below) uses 8 bit-planes of a 24-bit frame buffer to store a BSP-tree data structure with up to approximately 5000 polygons.

2.2. Description of the basic algorithm

The algorithm consists of two components:

- a one-time preprocessing module ("Make_tree") that converts the input polygon list into the BSP-tree structure, and
- an image-generation module ("Traverse") which traverses this structure and generates the polygons in a back-to-front order. (Strictly speaking, the order is not back-to-front, but is functionally equivalent to it.)

2.2.1. Building the BSP-tree

The fundamental notion is one of a separating plane: that is, given a plane in the 3D scene and a viewing point, no polygon on the viewpoint side of the plane can be obstructed by *any* polygon on the far side. Of course, if the viewpoint should move to the other side of the plane, the obstruction priorities are reversed [Schumacker, Brand, Gilliland and Sharp, 1969].

The algorithm uses this simple notion to construct a binary tree of polygons from the original polygon list (see Fig. 1). A polygon is selected from the list and placed at the root of the tree. Each remaining polygon in the list is tested to determine the side of the root polygon in which it lies and is then placed in the appropriate descendent list. Any polygon which crosses the plane of the root polygon is split along that plane and each part put in the appropriate list (see polygon 5 in fig. 2). This procedure is repeated recursively in the following way: from each of these descendent lists, a polygon is selected to be the root of that subtree, and the remaining polygons in this list are split by the plane of the root of the new subtree (see polygon 2 fig. 3).

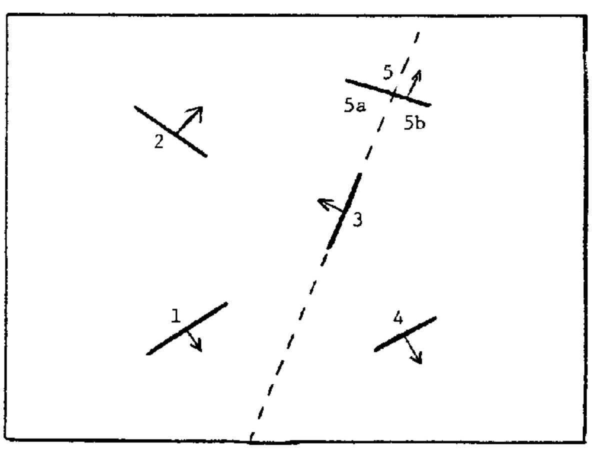


Figure 1: Top view of scene

```
PROC Make_tree (poly_list)
      returns (BSP_tree);

 if (poly_list is empty)
   return (NULL_TREE)

 else {
   root := select (poly_list);
   back_list := NULL;
   front_list := NULL;
   foreach (polygon in poly_list)
     if (polygon is not the root)
     {
       if (polygon in front of root)
         Addlist (polygon, front_list);
       else if (polygon is behind root)
         Addlist (polygon, back_list);
       else
         {
         Split_poly
           (polygon, root,
             front_part, back_part);
         Addlist (front_part, front_list);
         Addlist (back_part, back_list);
         }
     }

   return
    (Combine_tree (Make_tree (front_list),
          root,
          Make_tree (back_list)));
 }
END
```

The choice of the root polygon strongly influences the size of the tree. In the example illustrated in Figures 1-3, a better choice of the initial root would be a polygon *other* than 3, for example polygon 5. Figure 4 illustrates a BSP-tree with an initial choice of polygon 5. Note that the number of polygons in this tree is the same as the number in the input polygon list, while in the example of Figure 3, the tree is larger (6 polygons instead of the original 5). In reasonable-sized scene descriptions, the tree may grow substantially. In section 3.1 we discuss strategies to keep the tree small and give results. ([Naylor, 1981] develops bounds on the size of the BSP-tree and discusses many other related issues.)

2.2.2. Image generation

Once the BSP-tree has been constructed, generating an image from any point of view is simple. The tree is traversed in a special in-order fashion. At each node of the tree, we determine whether the eye is in front of or behind the node polygon. This result determines which subtree will be traversed first. The order is always the same: traverse the "other side" subtree, output and paint the node polygon, then traverse the "near side" subtree.

3. New Results

3.1. Tree size

When this algorithm was first introduced, there was concern that, in many cases, the tree would be significantly larger than the original polygon list. Indeed, there were fears that, given a list of N polygons, the BSP-tree may turn out to contain N^2 or more polygons! Although no tight bound has been yet been found, from our experience the trees derived from most world model databases are less than twice the size of the original

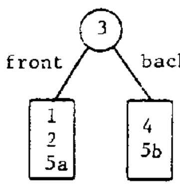


Figure 2:
After one level of recursion. Polygon 3 chosen as root.

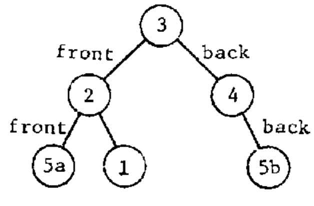


Figure 3:
A complete tree.

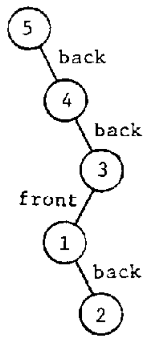


Figure 4:
An alternate tree with polygon 5 at root.

```
PROC Traverse_tree (tree)
/*
  Traverse an input BSP-tree and generate visible-
  surface image. The function Display handles
  transformation, clipping, perspective division,
  lighting and rendering. All but the rendering
  can be done elsewhere if deemed more efficient.
*/

 if (tree is empty) return
 else {
  if (eye is in front of tree.root.polygon) {
    Traverse_tree (tree.back_descendent);
    Display (tree.root.polygon);
    Traverse_tree (tree.front_descendent);
  }
  else {
    Traverse_tree (tree.front_descendent);
    Display (tree.root.polygon);
    /* if back-facing polygons are to
      be considered invisible,
     remove the previous line. */ •
    Traverse_tree (tree.back_descendent);
  }
}
END
```

polygon list; the largest found was 2.33 times. It should be noted that the image generation time does not increase by nearly this factor of two since it is dominated by pixel-painting time and the total pixel area of a model does not change as the polygons within it are split. Table 1 indicates the input polygon list and BSP-tree sizes for the various objects illustrated in Figures 6 - 10.

We use the heuristic of selecting a root at each step whose plane cuts the fewest other polygons in the list. In our first experiments, we made the selection after examining every polygon in the list as a candidate for root. We have since found, however, that selecting just a few candidates at random from the list (originally suggested by Zvi Kedem) gives nearly as good results. These results are shown in table 1, which indicates that near-minimal trees can be found by examining only about 5 candidate polygons at each level. The most startling result in this table is the example of the "Old Well", (the longstanding symbol of UNC - Chapel Hill). In this highly non-convex model of 356 polygons, this heuristic produces a tree which is *exactly* the same size as the input polygon list.

3.2. Simple implementation in a graphics processor

Although it may be clear from section 2.2 that the algorithm can be simply expressed in a high level language with recursion, what may be less obvious is that the image generation component is simple enough to be implemented entirely within a programmable graphics processor. Our implementation runs on an Ikonas RDS3000 raster graphics system, which has a programmable AM2900-based internal processor. The run-time component consists of 1309 64-bit microcode words, of which only 218 words implement the BSP-tree part of the image generation algorithm, while the rest (1091 words) implement the transformations, clipping, perspective and polygon painting routines that are needed in any 3D image generation system.

Figure name (and number)	**Size of input polygon list**	**Output for varying number of candidates** 1	3	5	15
Carotid artery (Fig.s 6a,b)	915	2421	1843	1720	1633
Old Well (Fig. 8) (low detail)	356	1125	384	378	356
SkullCT (Fig. 9)	988	3341	2361	2255	2053
Old Well (high detail)	1000	2426	1304	1176	1032
Space Shuttle	418	2092	1201	1095	972
3cubes (Fig. 10)	216	402	279	263	240
Ribbon plot	368	1478	929	797	860
Klein bottle	450	1475	1118	1035	990
Robot arm	268	975	650	587	440

Figure name	**Treemaking time for 5 candidates (seconds)**
Carotid artery	116.3
Old Well (low detail)	45.0
Old Well (high detail)	448.2
SkullCT	225.8
Space Shuttle	144.2
3cubes	15.4
Ribbon plot	86.4
Klein bottle	272.7
Robot arm	52.7

Table 1: Tree making statistics. The implementation is written in C under UNIX on a VAX 11/780.

3.3. System speed

Table 2 indicates the time needed to generate the images illustrated in Figures 6 - 10. Because some of the speed of this implementation is due to a relatively fast graphics processor, it may be useful to analyze where the efficiencies are due to the algorithm itself (rather than the processor), to determine the utility of the algorithm independent of the processor on which it is implemented. To do this, we compare it with the most similar previous algorithm, the widely used Z (or depth) buffer algorithm (see, eg., [Foley and Van Dam, 1982]). The BSP-tree algorithm has a fixed per-polygon overhead of the tree traversal; this consists of a) the maintenance of a stack of tree return pointers for traversing the tree, and b) performing the inner product at each node to determine which way to turn next. The Z buffer, on the other hand, has none of this *polygon* overhead, but has the extra burden of calculating the Z, comparing, and possibly updating Z at *each pixel*. There is then a tradeoff between this per-polygon overhead and the per-pixel overhead, and it appears that until the average polygon size becomes very small (a few pixels) that the per-polygon overhead of the BSP-tree will be considerably less burdensome than the per-pixel

Figure name and number	Number of polygons in tree	Image generation rate (frames/second)
Carotid Artery (Fig. 6a,b)	1633	0.50 - 0.58
SkullCT (Fig. 9)	2255	0.40 - 0.48
3 atoms (Fig. 7)	1440	1.53 - 1.84
Old Well (Fig. 8) (low detail)	356	1.6 - 2.5
Space Shuttle	923	1.94 - 2.56
3 cubes (Fig. 10)	280	2.46 - 4.81
Old Well (high detail)	1005	1.05 - 1.58

Table 2: Image generation statistics.

course, is only valid for applications which can conform to the two limitations of the BSP-tree algorithm: a static world model and sufficient local memory to hold the BSP-tree.

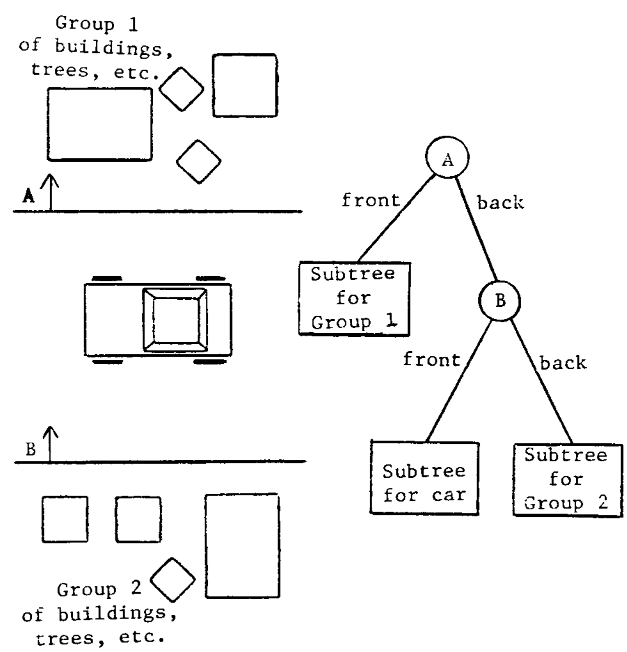
4. Future Work

4.1. Removing static world model restrictions

In the present implementation of the algorithm, whenever the world model changes, the entire BSP-tree must be rebuilt. From table 1, one can see that it takes a minute or two for this process. We are working on various alternatives to allow relaxing this restriction. We are considering the situation in which there is only limited change in the world model. One such example might be when the *range of motion* of the moving objects is known; such as airplanes which always fly above the airport and land only on runways, or automobiles which remain on the road, or parts of molecules which only move in certain restricted ways, or doors which only swing on their hinges. With this knowledge, we might be able to construct a series of convex regions, which always contain the moving object. The BSP-tree of the world model could be constructed such that no root polygons cuts this region. This causes all the polygons of the moving object to end up in their own subtree, which may then be transformed independently from the rest of the scene by using a nested transformation matrix at the root of this object's subtree (see Figure 5).

4.2. Anti-Aliasing

We are currently experimenting with adding anti-aliasing to the image generation using a sub-pixel mask similar to the latest Evans and Sutherland digital scene generator, the CT-5 [Schumacker, 1980]. This technique involves maintaining a binary mask of, say, 4x4 subpixels at each pixel. The polygons are painted front-to-back, and are sampled at the subpixel resolution, with the binary mask indicating the subpixel areas which have already been covered by a polygon. We note that the BSP-tree can generate a front-to-back (equivalent) order of polygons simply by reversing the order of the traversal (i.e., instead of far side; node polygon, near side, the order becomes near side, node polygon, far side). The contributions of the current polygon to a particular pixel's color are determined by the number of subpixels within that pixel of which this polygon is visible. This contribution is accumulated in the RGB pixel value in the image frame buffer.



Figures 5a and 5b: non-static scene handling

5. Summary and Conclusions

We have shown that the BSP-tree visible-surface algorithm generates images rapidly enough to be useful in interactive applications and that it can be easily implemented in a programmable graphics processor. Further, we have shown that in all cases encountered so far, that the tree size stays within reasonable bounds. We are currently using the system to study reconstructed surfaces of human arteries and density distributions of organic molecules. Our experience indicates that it is a viable (and faster) alternative to the commonly used Z buffer in many situations.

Finally, we note that this algorithm may be increasingly attractive as new raster graphics hardware systems become available. At least one new commercially available system (Megatek 7200 [Foley and Van Dam, 1982]) and several experimental designs ([Clark and Hannah, 1980] and [Fuchs, Poulton, Paeth and Bell, 1982]) concentrate on fast rendering of lines and polygons. It appears that these systems will most easily and directly generate realistic images of 3D scenes from a back-to-front ordered list of polygons. Since many can be expected to have transformation and clipping hardware, all that remains to be done in software is to generate the ordered list of polygons, something which can be achieved quite handily by the BSP-tree algorithm.

Acknowledgements

We would like to thank the many people at UNC-CH and elsewhere who have assisted us with this research: Gregg Podnar and Yehuda Kalay (CMU), Mike Connally (Yale), James R. Smith (NASA-Johnson Space Center), Richard A. Weinberg (Cray Research), and Sandy Bloomberg and Frank Moore (UNC) all for kindly allowing us the use of their graphic databases, Gary Bishop (UNC) for programming tools, and Bo Strain and Mike Pique (UNC) for photographic assistance.

References

Clark, J.H. and Hannah, M.R. Distributed Processing in a High-Performance Smart Image Memory", LAMBDA, 4th Quarter (1980), pp. 40-50.

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

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

Fuchs, H., Poulton, J., Paeth, A., Bell A. Developing Pixel-Planes, A Smart Memory-Based Raster Graphics System, Proceedings, Conference on Advanced Research in VLSI, Cambridge, Mass. January 25-27, 1982.

Naylor, B.F. "*A Priori* Based Techniques for Determining Visibility Priority for 3-D Scenes", Doctoral Dissertation, University of Texas at Dallas (1981).

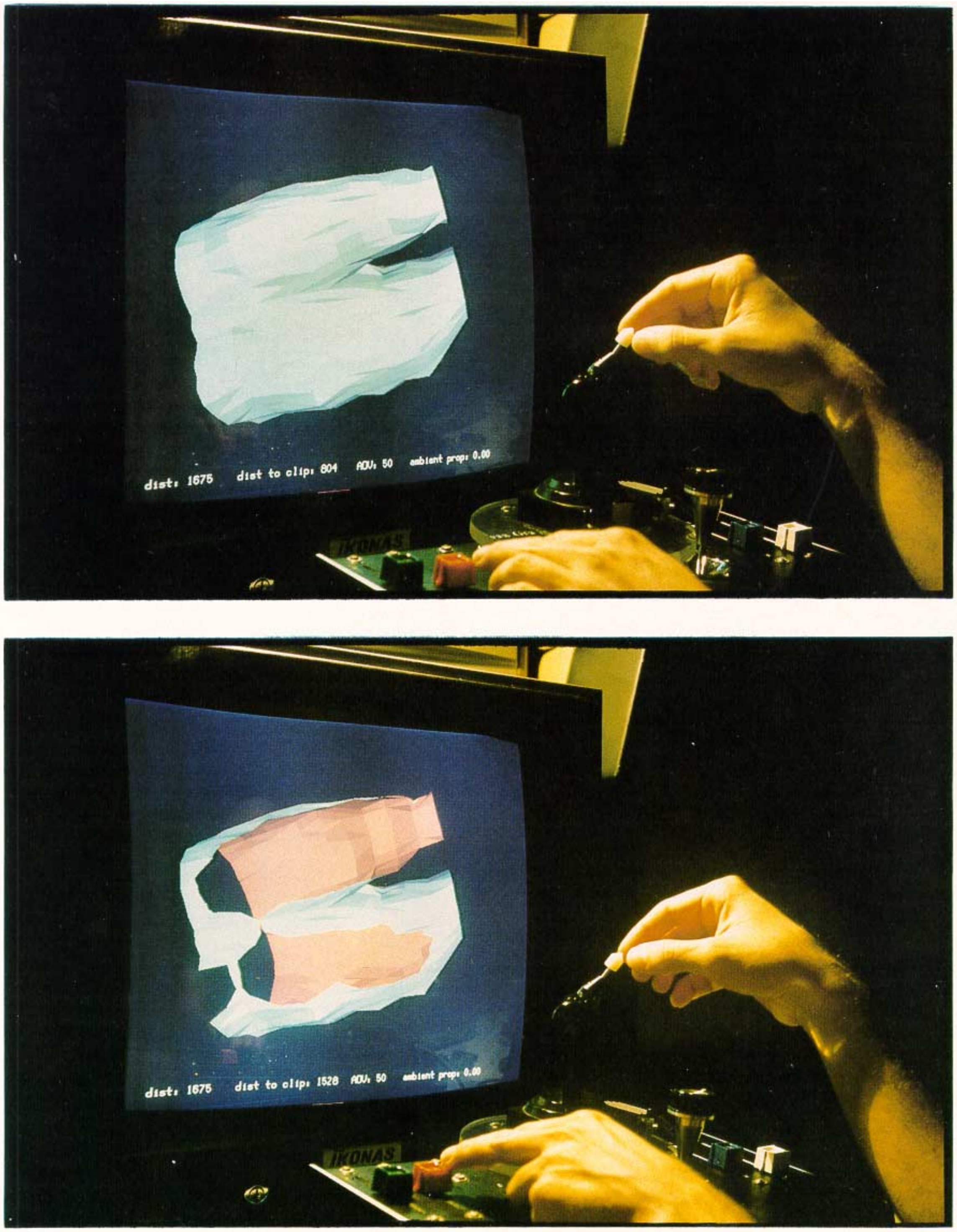
Newman, W.M. and Sproull, R.F. *Principles of Interactive Computer Graphics* (2nd. ed.), McGraw-Hill, New York, (1979).

Schachter B.J. Computer Image Generation for Flight Simulation, IEEE Computer Graphics and Applications, Vol. 1, No. 4, (October, 1981).·

Schumacker, R.A. "A New Visual System Architecture, Proc. Second Interservice/Industry Training Equipment Conf., Salt Lake City, Utah, (Nov. 1980), pp. 94-101.

Schumacker, R.A., Brand, B., Gilliland, M., and Sharp, W. "Study for Applying Computer Generated Images to Visual Simulation", Tech. Report No. AEHRL-TR-69-14, (AD 700375), US Air Force Human Resources Lab (1969).

Sutherland, I.E., Sproull, R.F. and R.A. Schumacker, R.A. A Characterization of Ten Hidden-Surface Algorithms", Computing Surveys, Vol. 6, No. 1 (1974).



Figures 6a,b:
Two images of a carotid artery reconstructed from CT data. A 1.7 second update time is achieved from a BSP-tree of 1633 polygons. The three-dimensional "toothpick", used for rotation, and the slide-pot box; for controlling zoom, clipping, angle-of-view and lighting effects are visible. Note the use of the clipping plane for cross-sectioning.

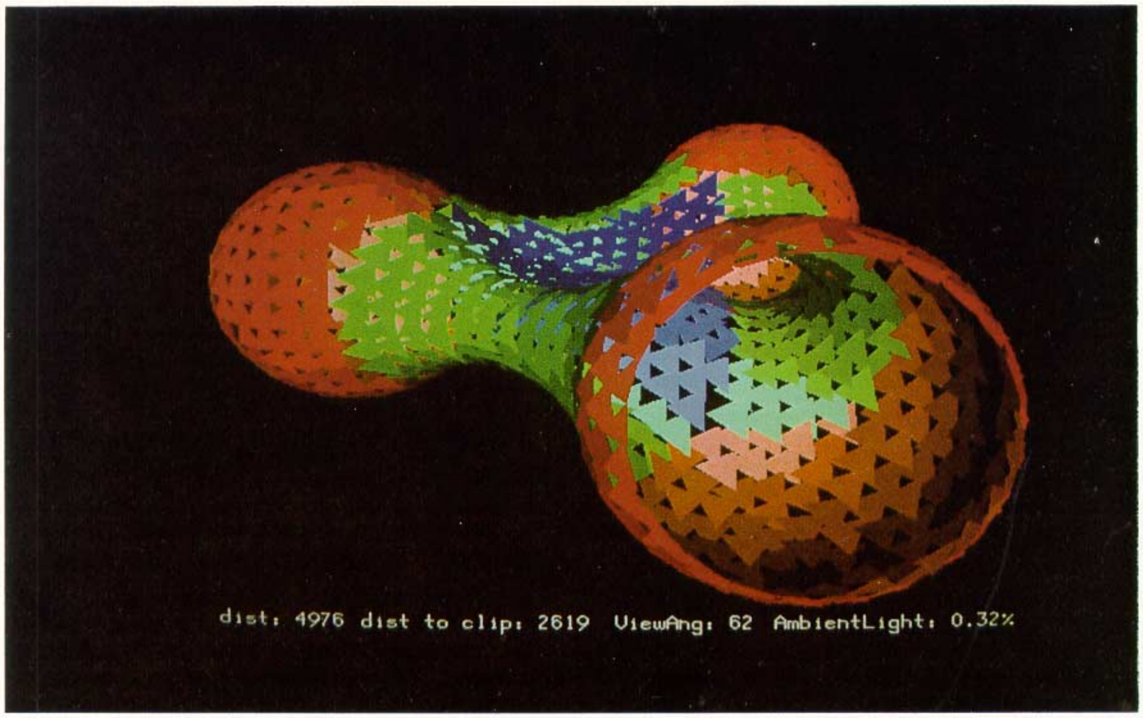


Figure 7:
Three simulated atoms, as created by Michael Connally of Yale. The BSP-tree has 1440 polygons, and we generate about 1.5-2 frames per second. The front of the object has been cut away by the interactive clipping plane.

Figure 8:
The small version of the Old Well. A BSP-tree of 356 polygons was created from 356 input polygons. We generate more than 2 images a second in most cases.

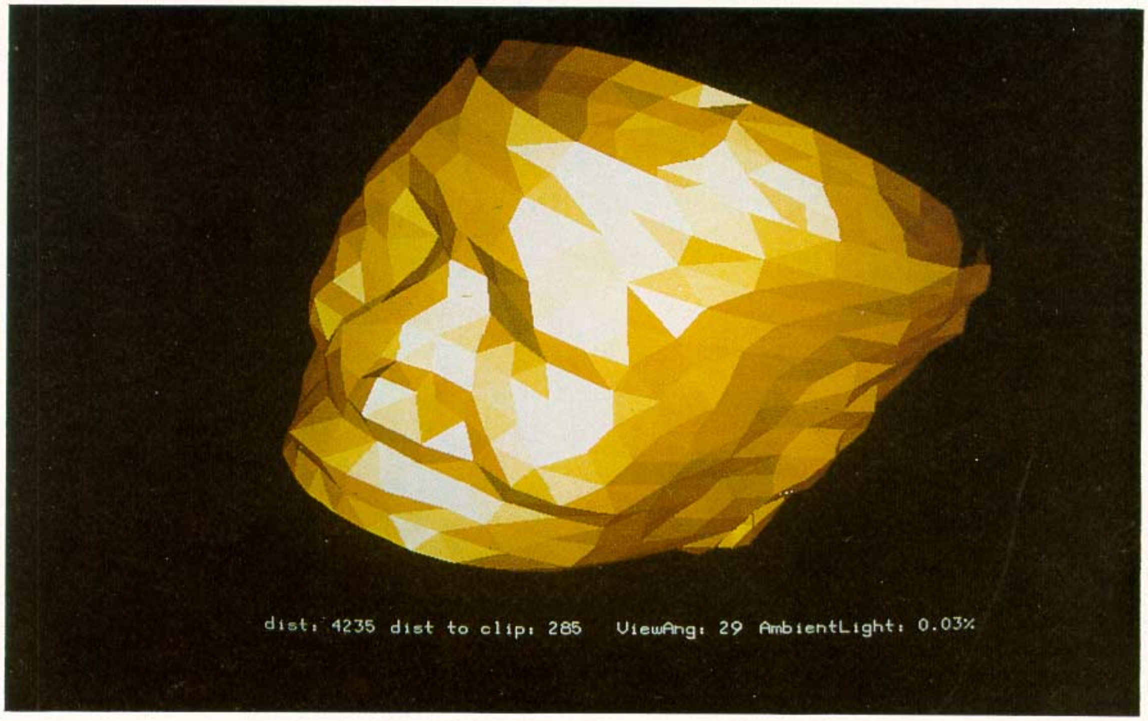


Figure 9:
A skull, reconstructed from CT scans. A single 8-bit intensity value is computed, rather than separate red, green and blue intensities as in the other images. This is then displayed using a heated object spectrum color map.

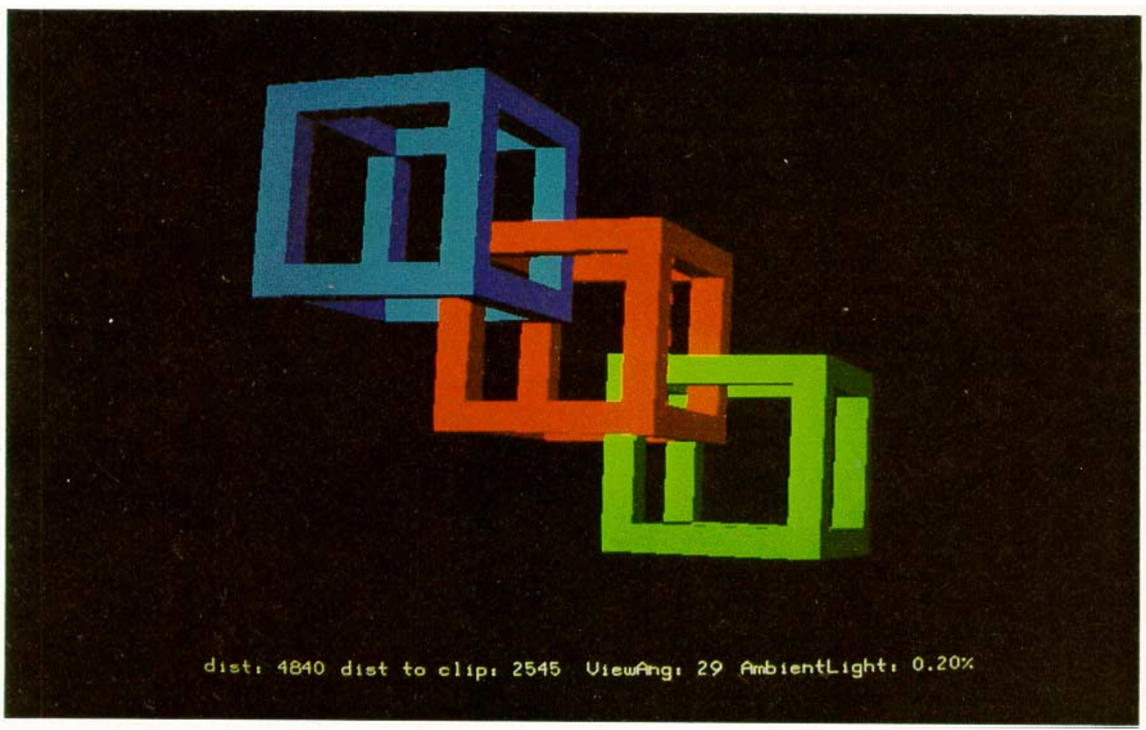


Figure 10:
3 interlocking cubes. The BSP-tree has 280 polygons. For this very simple object with low screen coverage we can generate up to 5 images per second.