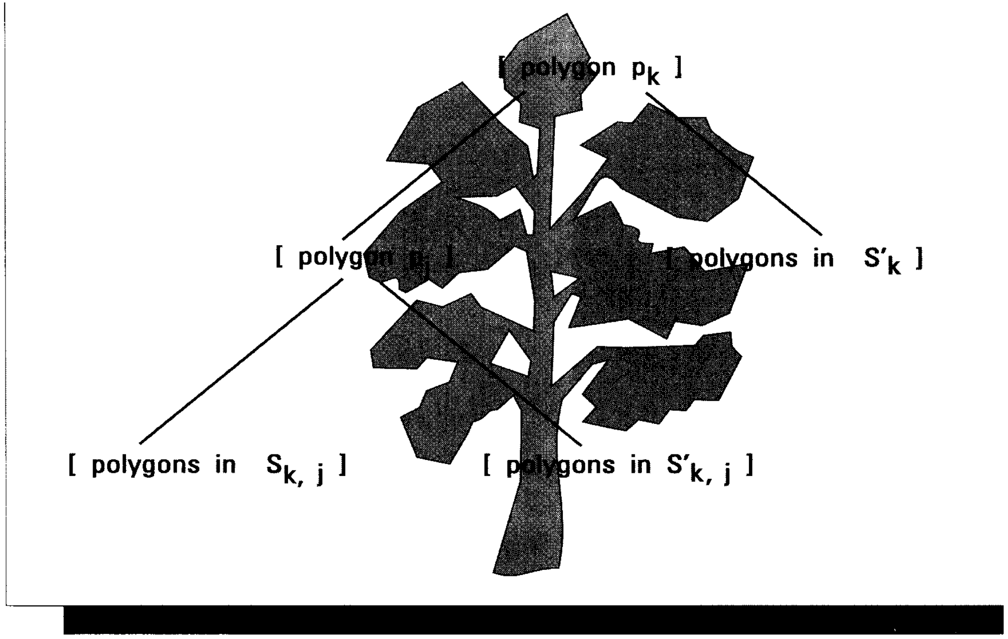


Front-to-Back Display of BSP Trees

Dan Gordon and Shuhong Chen
Texas A&M University

Combining polygon scan-conversion with a dynamic screen data structure led to a front-to-back approach for displaying BSP trees. This technique improves significantly over the back-to-front method.

We developed a technique to display binary space partitioning (BSP) trees that is faster than the usual back-to-front display method. Our technique—a front-to-back approach—provides significant speed-up in the display time of polygonal scenes that depend on BSP trees, especially in cases where the number of polygons is large.

BSP trees[1] are one of the most useful data structures for polygonal scenes. The structure of BSP trees embodies the geometrical priorities of a set of polygons in 3D space. Independent of any viewpoint, this structure (once constructed) enables fast display with hidden surface removal. It proves especially useful in applications with a fixed environment where we need to generate many images from different viewpoints—in flight simulators, for example.

Once we have determined a viewpoint, we can display the scene by traversing the BSP tree in back-to-front order and scan-converting each polygon. Since polygons closer to the viewpoint will be displayed later than further ones, they will overwrite the further ones, producing correct hidden surface removal.

0272-17-16/91/0900-0079$01.00 ©1991 IEEE

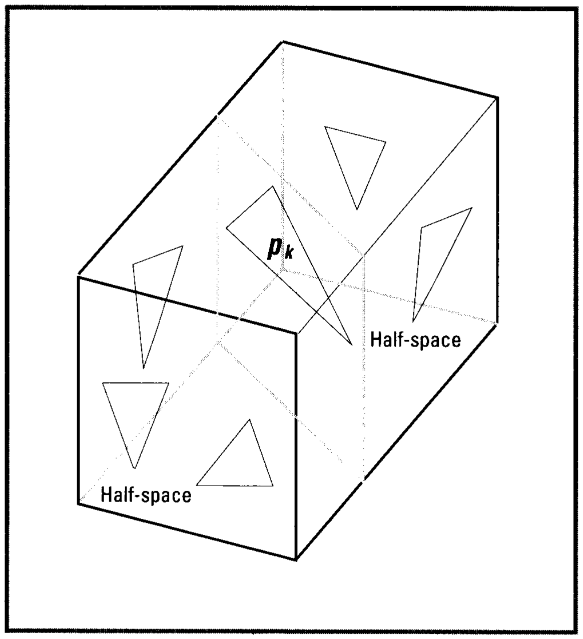


Figure 1. Environment split by plane of p_k.

In voxel and octree environments, researchers have had some success with display algorithms that traverse the scene in front-to-back order (instead of back-to-front).[2,3] In principle, we can convert any back-to-front display method to a front-to-back one by a simple additional check: Before a pixel is set, it is checked to verify that it had not already been set. However, this simple modification has no advantage. Success lies in the use of certain data structures to represent the image during processing. These structures enable the algorithms to eliminate from consideration large chunks of pixels that have already been set.

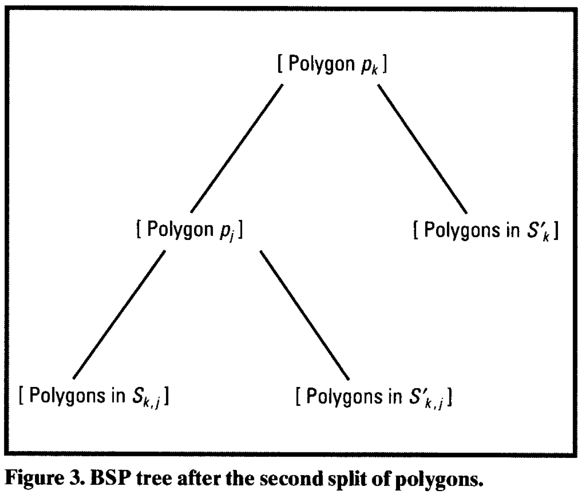


Figure 3. BSP tree after the second split of polygons.

In our method for displaying BSP trees in front-to-back order, we use the data structure introduced elsewhere[3] for front-to-back display of voxel-based objects. This data structure combines with polygon scan-conversion for efficient display of polygonal scenes. In principle, any priority-based algorithm (for example, the depth-sort algorithm[4,5]) that scan-converts polygons one at a time can be modified in this manner.

Back-to-front display

A BSP tree, which is a data structure for polygonal objects, enables easy traversal in any order relative to an observer.[1] Suppose the 3D environment is defined by a set of polygons $P = \{p_1, p_2, \ldots, p_n\}$. We choose an arbitrary polygon p_k from this set to serve as the root of the tree. The plane defined by p_k partitions the rest of the three-space into two half-spaces, as shown in Figure 1. Let S_k and S_k' denote the two half-spaces, which we can identify with the positive and negative sides of the plane. (If the plane equation is $ax + by + cz + d = 0$, then $S_k = \{(x, y, z) \mid ax + by + cz + d \geq 0\}$.)

p_k has two subtrees, called positive and negative, constructed as follows: If a polygon in $P-\{p_k\}$ lies entirely in S_k, then it is placed in the positive subtree. If it lies entirely in S_k', then it is

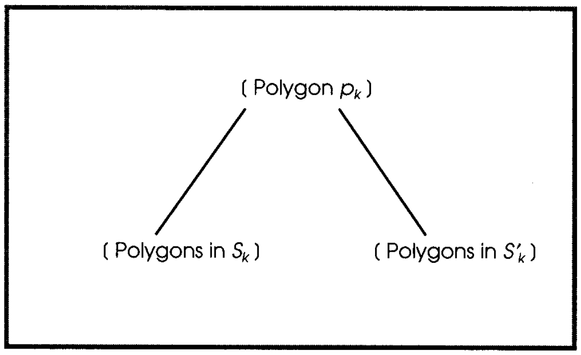


Figure 2. BSP tree after the first split of polygons.

placed in the negative subtree. If the partitioning plane cuts the polygon, then the polygon is split into two, with one part placed in S_k and the other in S_k'. Figure 2 illustrates this. In most normal scenes, most of the polygons will lie entirely in just one of the two half-spaces.

The process now continues recursively with each of the subsets of the polygons (see Figure 3). The resulting binary tree is a BSP tree. Note that after a BSP tree is formed, we can easily add new polygons to it. In fact, we can construct the BSP tree on an on-line basis, with each additional polygon inserted into the existing tree. Figure 4 shows an example of three polygons and the resulting BSP tree.

Once the viewing position and orientation are given, we can determine the visibility priorities of the polygons in the BSP tree. The calculation of the visibility priorities is a variant of an in-order traversal of a binary tree (traverse one subtree, visit the root, traverse the other subtree). For example, suppose we want an order of traversal that visits the polygons from those farthest away to those closest to the current viewing position. At any given node, there are two possible traversals: positive side subtree → node → negative side subtree, or negative side subtree → node → positive side subtree. We choose one of these orderings based on the relationship of the current viewing position to the node's polygon.

According to the viewing position, we can classify the two sides of a node's polygon as the "near" side and the "far" side. The near side is the half-space containing the viewpoint, while the far side is the other half-space. The traversal for a back-to-front ordering is simply the recursive implementation of the order: (1) the far side, (2) the node, (3) the near side. This is detailed in the procedure shown in Figure 5.

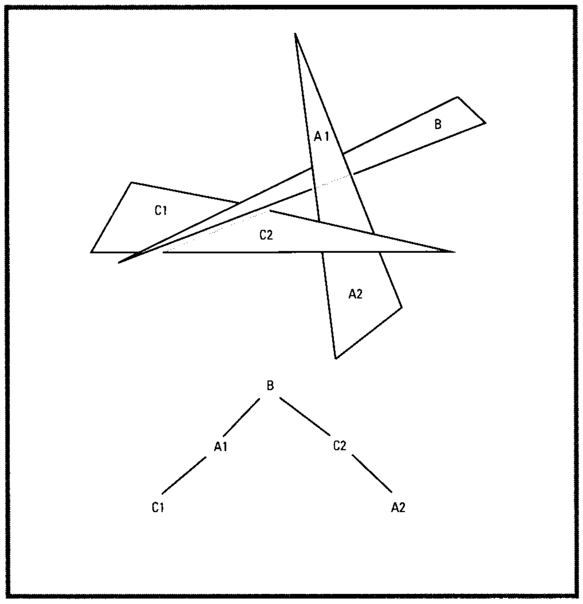


Figure 4. Example of a BSP tree.

Front-to-back order

The back-to-front traversal will clearly display the polygons correctly. However, we can also display the scene by traversing the tree in front-to-back order: (1) the near side, (2) the node, (3) the far side. The front-to-back traversal will only display the image correctly if, before a pixel is set, it is checked to verify that it had not already been set. Thus, a front-to-back traversal, if implemented in this naive fashion, will actually take longer than the back-to-front mode because of the extra work. The basic idea behind our technique is that large chunks of screen pixels can be checked quickly and eliminated using the "dynamic screen" data structure. The procedure shown in Figure 6 will display the scene in front-to-back order.

```
Procedure back_to_front(BSP_tree, view_point)
begin
  if ( BSP_tree != null )
    if ( positive_side_of( root(BSP_tree), view_point) )
      back_to_front(negative_branch(BSP_tree), view_point);
      display_polygon(root(BSP_tree));
      back_to_front(positive_branch(BSP_tree), view_point);
    else
      back_to_front(positive_branch(BSP_tree), view_point);
      display_polygon(root(BSP_tree));
      back_to_front(negative_branch(BSP_tree), view_point);
end
```

Figure 5. Traversal for a back-to-front ordering.

The dynamic screen technique was first introduced by Reynolds, Gordon, and Chen[3] for efficient front-to-back display of voxel-based objects. Basically, the dynamic screen represents unlit contiguous segments of pixels of each scan line. Combined with front-to-back traversal of the BSP tree, this enables us to set each pixel just once and eliminates the need to test individual pixels that have already been lit. This technique has the advantage of efficiently removing large chunks of lit pixels from consideration.

The data structure we use appears in many applications, for example, bucket sorting, graph representation, and run-length encoding of images. Assume the screen size is $N \times N$ pixels. An array of pointers $DS[0 \ldots N-1]$ is set up, each $DS[i]$ pointing to a linked list of the contiguous segments of unlit pixels at scan line i. Each list element has three fields: Min and Max for the minimum and maximum pixel coordinates of the unlit

```
Procedure front_to_back (BSP_tree, view_point)
  begin
    if ( BSP_tree != null )
      if ( positive_side_of( root(BSP_tree), view_point) )
        front_to_back(positive_branch(BSP_tree), view_point);
        display_polygon(root(BSP_tree));
        front_to_back(negative_branch(BSP_tree), view_point);
      else
        front_to_back(negative_branch(BSP_tree), view_point);
        display_polygon(root(BSP_tree));
        front_to_back(posive_branch(BSP_tree), view_point);
  end
```

Figure 6. Procedure to display the scene in front-to-back order.

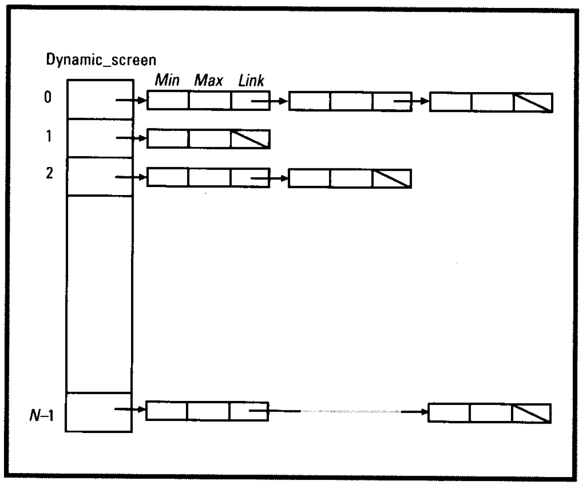


Figure 7. The dynamic screen data structure.

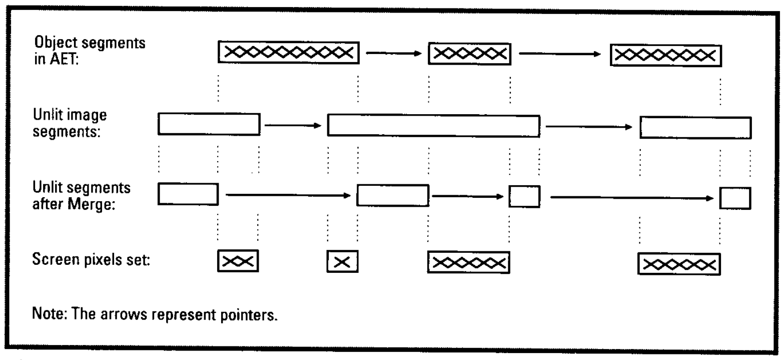

Object segments in AET:
Unlit image segments:
Unlit segments after Merge:
Screen pixels set:
Note: The arrows represent pointers.

Figure 8. The result of a merge operation.

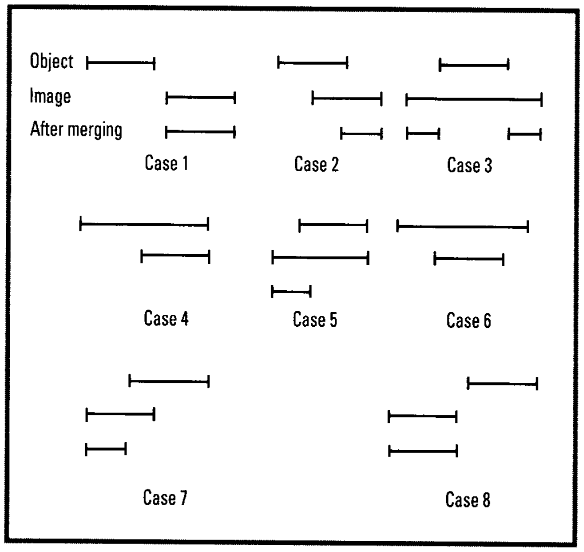


Figure 9. The eight cases arising from comparisons between a projected object segment and an unlit image segment, and the results of the merging operation.

segment, and Link, a pointer to the next list element (see Figure 7). For testing purposes, we chose $N = 1{,}500$. Initially, each list contained just one element representing the entire unlit scan line.

We used the standard polygon scan-conversion method for polygon display,[5] with each polygon displayed separately. Edges were inserted into an edge table in the usual way, then a linked list of the active edges—the active edge table (AET)—was swept through the scene in the usual manner.

The only fundamental difference between the standard scan-conversion and our approach is that at each scan line i, we perform a merge-like process between the AET and the list $DS[i]$. This Merge process forms the core of our efficient display. It starts out by considering the first object segment of the AET and the first contiguous unlit image segment of $DS[i]$. The Merge operation then calls Merge_segment, which determines the action that needs to be taken, based on the relative position of the two segments. The result of the Merge operation is an updated list $DS[i]$, and, of course, some screen pixels may be set. (See Figure 8.)

The arguments to Merge_segment are an object segment from the AET, starting with pixel I and ending with pixel J, and an unlit image segment K in the list $DS[i]$, with initial pixel $K.Min$ and end pixel $K.Max$. Merge_segment compares I and J against $K.Min$ and $K.Max$. Depending on the outcome of these comparisons, there are eight distinct actions to take, as shown in Figure 9. Note that in Figure 9 the image segments are unlit segments of screen pixels.

The outcomes of these comparisons (together with the proper actions) fall into four different categories:

A. One segment completely precedes another (cases 1 and 8 in Figure 9). In these cases, no changes are made to the image segment, and nothing is drawn on the screen. In case 1 we advance (along the AET) to the next object segment, and in case 8 to the next unlit image segment (along the list $DS[i]$).

B. The object segment covers one end of the unlit image segment (cases 2, 5, and 7). The covered pixels are painted in and the image segment is shortened at one end by changing $K.Min$ or $K.Max$. These operations are performed by the procedures Paint_segment and Change_segment. In case 2, we advance to the next object segment, in case 7 to the next image segment, and in case 5 along both lists.

C. The entire image segment is covered by the object segment (cases 4 and 6). The image segment is painted in and deleted from the linked list by the procedure Delete_segment. In case 4 we advance along both lists, and in case 6 we advance to the next image segment.

D. The object segment falls strictly within the image segment (case 3). In this case the covered pixels are painted in and the

```
Procedure split_segment (I, J, K)
  (* K.Min < I ≤ J < K.Max — case 3 in Figure 9 *)
  (* split segment K into two segments *)
begin
  K.Max = I-1;
  new(K′);
  K′.Min = J+1;
  K′.Max = K.Max;
  K′.Link = K.Link;
  K.Link = K′;
end
```

Figure 10. The procedure to split a segment.

image segment splits into two segments (corresponding to the uncovered parts). The splitting operation is performed by the procedure Split_segment, which adds a new list element and updates the fields of the old ones. The Merge procedure advances in this case to the next object segment.

All the above procedures are quite simple, so we only describe one of them (see Figure 10).

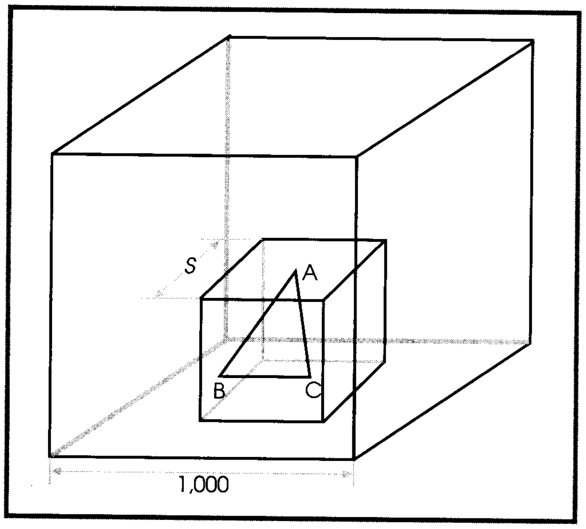


Figure 11. A randomly generated triangle (ABC) inside the universe cube.

Experimental results

To get some meaningful statistical data on the behavior of our technique, we applied it to randomly generated triangles. The universe is a 1,000 × 1,000 × 1,000 cube. Users specify the total number of triangles they want to generate and a number S (between 10 and 500) that controls the size of the triangles. Our program first generates three random numbers (C_1, C_2, C_3) inside the universe. These three numbers specify the center of a small cube of size $S \times S \times S$. Next, three points are randomly generated inside the small cube, giving the coordinates of the triangle (see Figure 11). Each coordinate is a random number, chosen uniformly between $C_i - S/2$ and $C_i + S/2$. The program was implemented on a Sun Sparc workstation.

We performed a series of experiments, with the size parameter S ranging from 50 to 500 and the number of polygons ranging from 100 to 500. Each BSP tree created was displayed in four different ways: back-to-front, front-to-back, constant shading, and intensity interpolation shading (see Foley et al.[5]). The results of these experiments are shown graphically in Figures 12 to 14 for small, medium, and large values of S. Note that the number of polygons specified in the figures is the number before the creation of the tree. Since many polygons split upon creation of the BSP tree, the final number of polygons is much higher (almost 10,000 polygons are created from the original 500 for $S = 500$).

Figure 12 shows no significant difference between the two methods for small polygons. While back-to-front is slightly bet-

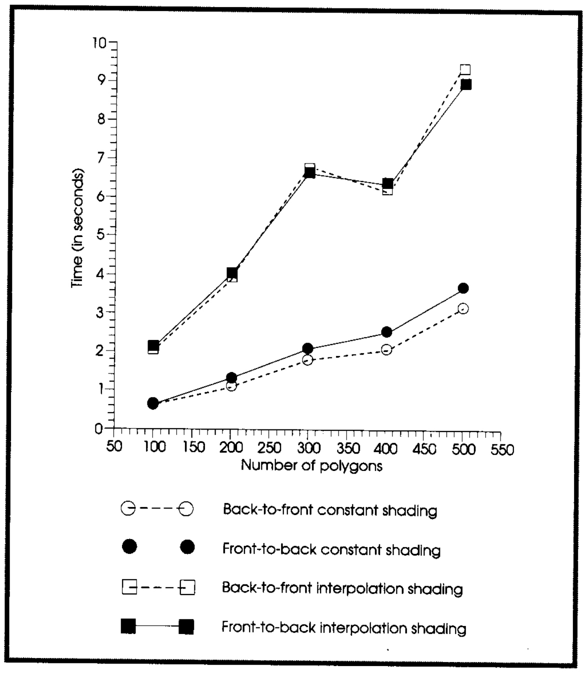


Figure 12. Experimental results for polygons of size 50.

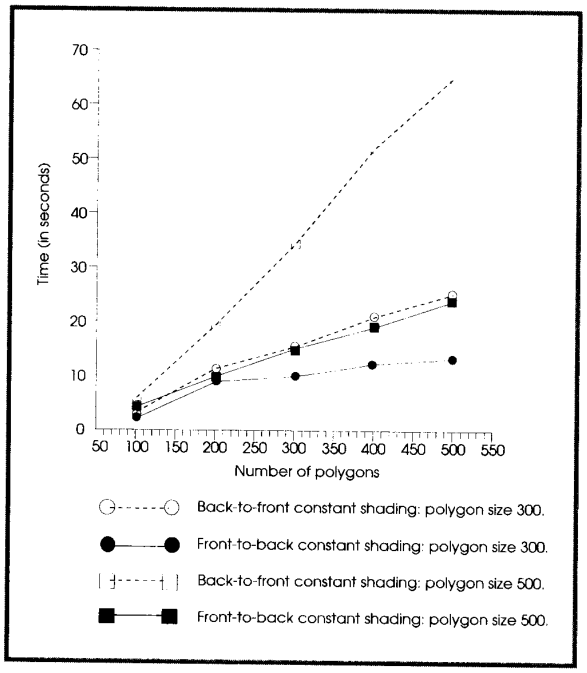


Figure 13. Experimental results for medium and large polygons using constant shading.

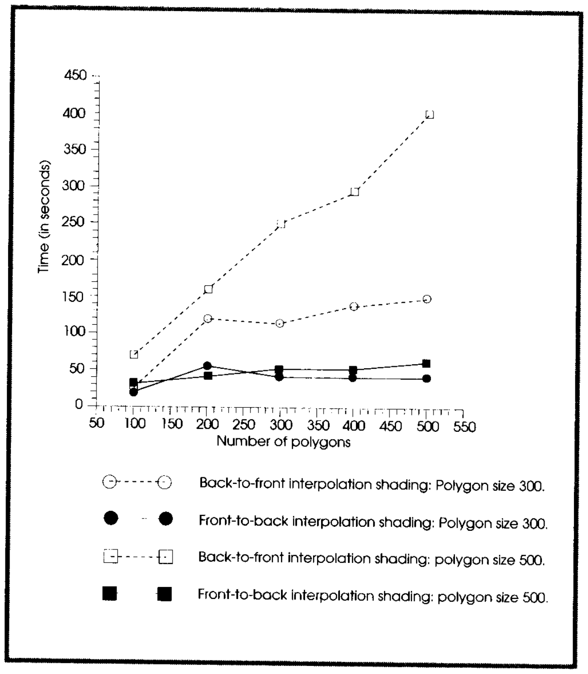


Figure 14. Experimental results for medium and large polygons using interpolation shading.

ter with constant shading, the additional computations required for interpolation shading make front-to-back slightly better.

In Figure 13 we begin to see some significant differences between the two methods, even with constant shading, especially for large polygons (size parameter $S = 500$). The difference between the two methods is much more pronounced in Figure 14 (intensity interpolation shading), where the time for front-to-back is about 14 percent of the time for back-to-front for 500 large polygons.

As mentioned, the actual number of polygons in the BSP tree can be much higher than the original number. To control this number, we also present data for parallel polygons, which do not split when a BSP tree is created. This is shown in Figures 15 and 16 for constant and interpolation shading. We see from this data that when the number of polygons exceeds 10,000, front-to-back with constant shading takes approximately 30 percent of the time of back-to-front. This percentage shrinks to less than 8 percent for interpolation shading.

The data for parallel polygons and interpolation shading (Figure 13) shows that the times for front-to-back are almost constant when compared to the times for back-to-front. This can be explained by the mode of operation of the front-to-back approach: Once (most of) the picture has been painted in, the time required to eliminate distant polygons is very small. This results from the efficient elimination of hidden object segments as implemented by the merging operation between a dynamic screen list and the AET. On the other hand, back-to-front laboriously spends equal time on all polygons, even if they are overwritten at a later stage.

Conclusions

Our experimental data confirm that our technique can provide a significant speed-up in the display time of BSP trees, especially in scenes with a large number of polygons. Also, we have seen that when the shading becomes more time-consuming, our method again has an advantage because no time is wasted in shading pixels that eventually get overwritten. We expect that with more sophisticated shading—such as normal interpolation (see Foley et al.[5])—the front-to-back method will be even more advantageous.

However, our method involves some overhead unjustified for simple images. The precise point at which our method begins to justify itself will vary with the system used, the quality of the image desired, and the complexity of the scene.

Although we have implemented our technique on BSP trees, the core of our method is the Merge operation between the active edge table of polygon scan-conversion and the list of unlit image segments in a scan line. Nothing in this Merge necessarily ties it to BSP trees, so we could use our method with any other list-priority algorithm,[5] such as depth-sort.[4]

Our technique also has a potential for parallel implementation on specialized graphics processors such as the Silicon

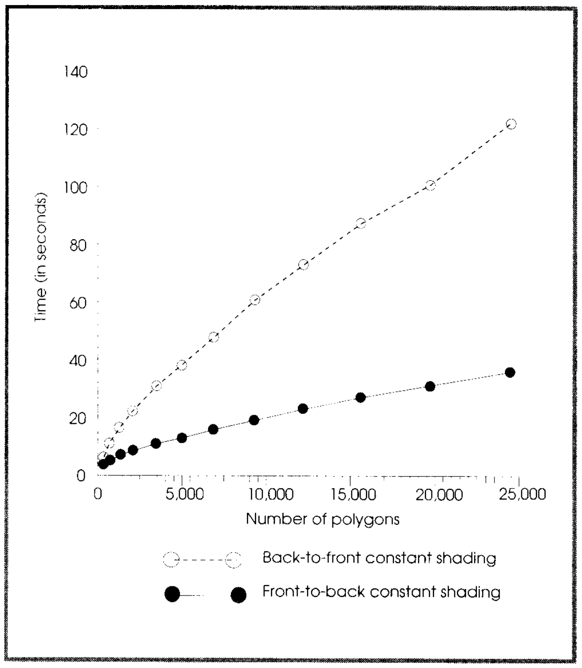


Figure 15. Experimental results for parallel polygons using constant shading.

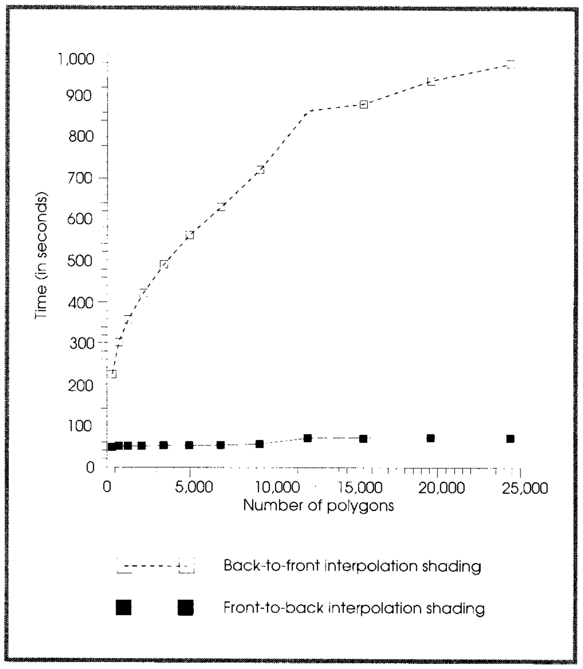


Figure 16. Experimental results for parallel polygons using interpolation shading.

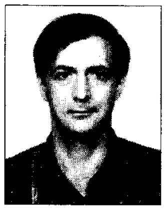
Graphics GTX workstations.[6] This potential arises because the merging operation of one scan line can be done independently of others, hence different processors can perform it on different scan lines. For example, in the scan-conversion subsystem of the Iris GTX,[6] five different "span" processors work on five different (interlaced) sets of scan lines. In principle, the number of processors can equal the number of scan lines. ❑

Acknowledgments

The authors would like to thank Nathan J. Pfluger for a preliminary version of a (back-to-front) BSP program. Under Dan Gordon's supervision, Shuhong Chen implemented the computer program for this research as a project for his Master of Computer Science degree at Texas A&M University.

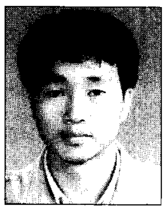
References

1. H. Fuchs, Z.M. Kedem, and B. F. Naylor, "On Visible Surface Generation by A Priori Tree Structures," *Computer Graphics* (Proc. Siggraph), July 1980, pp. 124-133.
2. D. Meagher, "Efficient Synthetic Image Generation of Arbitrary 3D Objects," *Proc. IEEE Computer Society Conf. Pattern Recognition and Image Processing*, June 1982, pp. 473-478.
3. R.A. Reynolds, D. Gordon, and L.-S. Chen, "A Dynamic Screen Technique for Shaded Graphics Display of Slice-Represented Objects," *Computer Vision, Graphics, and Image Processing*, Vol. 38, No. 3, June 1987, pp. 275-298.
4. M.E. Newell, R.G. Newell, and T.L. Sancha, "A Solution to the Hidden Surface Problem," *Proc. ACM Nat'l Conf.*, Aug. 1972, pp. 443-450.
5. J.D. Foley et al., *Computer Graphics: Principles and Practice*, 2nd ed., Addison-Wesley, Reading, Mass., 1990.
6. *Iris GTX: A Technical Report* (rev. 2) and *Iris GT Graphics Architecture* (tech. rept.), Silicon Graphics, Mountain View, Calif.



Dan Gordon is a senior lecturer of computer science at the University of Haifa, Israel. His research interests include computer graphics, data structures and algorithms, and VLSI theory. He has previously taught at Texas A&M University and at several other universities.

Gordon received BS and MS degrees in mathematics from the Hebrew University of Jerusalem and a DSc degree in mathematics from the Technion—Israel Institute of Technology.



Shuhong Chen is a development engineer with Dowell Schlumberger in Tulsa, Oklahoma. His research interests include computer modeling of complex dynamic systems and scientific visualization.

Chen received a BS in physics from Zhongshan University, China in 1982, an MS in physics from the University of Nebraska, Lincoln, in 1988, and an MS in computer science from Texas A&M University in 1990. He is a member of Upsilon Pi Epsilon, the computing science honor society.

Readers may contact Gordon at Dept. of Mathematics and Computer Science, University of Haifa, Haifa 31905, Israel.