Hidden-Surfaces: Combining BSP Trees with Graph-Based Algorithms

Timothy Miller

Department of Computer Science
Brown University
Providence, Rhode Island 02912

CS-96-15
Aprill 1996

Hidden-Surfaces: Combining BSP Trees with Graph-Based Algorithms

Timothy Miller

Brown University site
The Science and Technology Center
for Computer Graphics and Scientific Visualization

Abstract

BSP trees and *a priori* potential-occlusion-graph-based techniques may be unified to produce a superior algorithm for visible surface determination involving a hybrid data structure in which BSP nodes are used where the simple graph-based algorithm breaks down. In particular, empirical results show around two to three times fewer split polygons for this method, 20–40% less run-time space needed, and up to 34% faster display refresh rates *overall* (including significant computation not affected by the algorithm).

1 Introduction

This paper examines two priority-list algorithms for hidden-surface removal: BSP trees and graph-based schemes. A new algorithm is then presented that merges the two to produce a superior algorithm in which BSP nodes are used where the simple graph-based algorithm breaks down; supporting analysis and results are given. Empirically, the algorithm produces 40–64% fewer split polygons (resulting in 10–20% fewer polygons output overall), 20–40% less space needed at run-time, and a refresh rate speed up of up to 34% *overall* (including significant computation not affected by the algorithm) on the objects described in Section 5.

In many rendering applications, the scene to be rendered is composed of polygonal objects that are internally static (the relations between the geometrical components do not change, though the object as a whole may be passed through general linear transformations); it is also guaranteed that, for each view, if two objects' projections (relative to the viewer) overlap, then for each point of the overlap the corresponding points of the objects have the same relation with respect to the viewer (i.e., those from one object are always further from the viewer than those from the other object), and furthermore the directed graph induced by these relations is acyclic. The essential idea is that a linear ordering of the "cluster priorities" of [15] as described in [16] can be found for each view (where a "cluster" is what is called here an "object"). For instance, the application may be a flight simulation game in which the airplanes all have bounding spheres that, when they intersect, become a collision object drawn as an explosion, and the airplanes are all far enough away from the (static) terrain that they never interleave with terrain from an airplane's viewpoint. Another possibility is that of an architectural walkthrough of a static environment. In many such applications preprocessing time is irrelevant. On machines in which z-buffering is not implemented in hardware, such as many PCs, and for relatively low complexity scenes with the above properties, as in many games, it is better to do hidden-surface elimination using priority ordering than z-buffering [16]. Priority ordering may also be useful if the machine has hardware 2D acceleration but not 3D acceleration (because it may be faster to use the built-in scan-converter with a higher-level hidden-surface algorithm than to write a software z-buffering scan-converter), or if z-buffering is provided in hardware but the scene is relatively simple. Furthermore, z-buffering may not be preferable if transparency is used [7]. And even when z-buffering would be preferable, sorting front-to-back via a priority scheme in addition to doing z-buffering may produce a small run-time speedup over z-buffering alone, since no new z-buffer and pixel values are written for those pixels already written from the same object.

2 BSP tree summary

The BSP tree [3, 5, 11] is a recursive partitioning of a space by hyperplanes. To use it as a preprocessing speedup for hidden-surface removal in static polygonal scenes, the scene is hierarchically subdivided by planes until there is no more than one polygon in each leaf cell (a cell is the intersection of all the parent halfspaces). The planes chosen may or may not be planes of polygons present in the scene. At run-time, drawing is done by a modified inorder traversal of the tree, testing at each node which side of the splitting plane the viewpoint is on and traversing the branch on the other side first. If a polygon happens to be coplanar with a splitting plane, it is added to a third branch that is drawn between the two halfspace branches. If the viewpoint is on the splitting plane, then the order used doesn't matter except that the coplanar polygons need not be drawn. This approach has the drawback that it may require splitting a significant number of polygons, particularly since efficient algorithms for determining trees with minimal numbers of splits do not seem to be known, although work has been done on finding good trees efficiently [3, 5, 7, 9, 11, 14]. The variant run-time algorithm [6] that processes polygons nearer to the viewpoint first, combined with a different clipping and scan-conversion method, may also be applied with suitable modifications to the algorithm presented in this paper. BSP trees may also be used for many other purposes not considered here; for examples, see [1, 2, 5, 8–10, 12–14, 17].

3 Graph-based summary

The idea of the graph-based approach [4, 11] to hidden-surface removal for static polygonal scenes with backface removal is to note the *potential-obscuration* relationship between two one-sided polygons, A and B, indicating whether there is some viewpoint from which A and B are both front-facing, A's projection onto a film plane overlaps B's, and at some point p of the overlap the line from the viewpoint through p intersects A first (which is to say that pixels at p should be drawn corresponding to A). This relation may be used to form a directed graph in which the polygons are nodes and a directed edge exists from node A to node B iff A

is potentially obscured by B. If this graph is acyclic it can be topologically sorted, and the polygons can be drawn back-to-front in the resulting linear order, independent of viewpoint, with backface-culling; in the resulting picture hidden surfaces are correctly removed in a manner similar to the painter's algorithm.

However, not all such graphs are acyclic. There are even scenes for which the corresponding graph is not acyclic even if the input polygons can be split arbitrarily [11]. Thus, to apply the method to general static one-sided-polygonal scenes, one must form the topological sort of the strongly connected components and then do something else with strong components of more than one polygon. Several possibilities for the "something else" are covered in the literature:

1. [4, 11] Don't handle them: restrict the input to those scenes without multipolygon strong components.
2. [4] Handle some of them: restrict the input to those scenes without multipolygon strong components even if polygons may be split by planes of other polygons.
3. [11] Use a non-graph algorithm like Newell's or Warnock's on the strong components.
4. [11] Remove all 2-cycles by splitting one of the member polygons and use a z-buffer-like algorithm on the priorities, per-pixel, as dynamically discovered at run-time from the graph.
5. [11, p. 53] Consider each of a potentially exponential number of simple cycles in the component and manipulate them by adding indicator nodes and possibly splitting polygons so that at run-time all that need be done is prune backfacing polygons from the component's subgraph and then topologically sort the subgraph.
6. [11, p. 67] Approximate the simple-cycle method (5) by a linear manipulation of the component as a whole instead of each cycle individually. The result is the same as in (5) except that it may contain more indicator nodes and split polygons.

4 New algorithm

Using the language of the previous sections, this paper presents a new algorithm (call it Q) for handling the strong components as follows:

- Pick a partitioning plane p, not necessarily that of a polygon in the component. Use essentially the same criteria for picking this plane as would be used in a BSP algorithm.
- Partition other polygons in the component by p (splitting if necessary) and collect the polygons in the component into three groups: one for each halfspace of p and one for those coplanar to p.
- Create a *BSP node* for p with the three groups as branches.
- For each of the groups for the two halfspaces, form the obscuration graph and recurse by topologically sorting into strong components and *only then* picking another partitioning plane if necessary. Obscuration relationships before splitting and partitioning can be reused to make forming the graph of the polygons in each halfspace efficient by limiting which polygons need to be checked for potential obscuration.

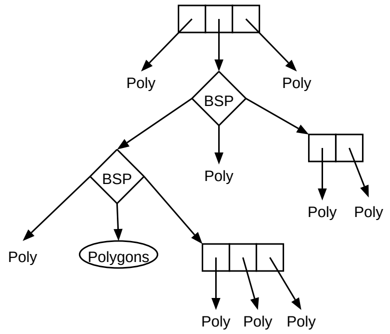

Figure 1: Example Q output diagram: sequences of squares are lists, arrows are pointers, "Poly" is a polygon, ovals are unordered sets of polygons, and diamonds are BSP nodes.

At run-time there is a data structure that can be defined recursively in terms of "elements": an element is a polygon, an ordered list of elements, an unordered set of elements, or a BSP node that points to three elements (one of which is guaranteed to be an unordered set of polygons). The resulting hierarchical structure essentially alternates between ordered lists and BSP nodes, with polygons and sets of polygons at the leaves (see Figure 1). A single object is represented as one element—the root of a tree—and drawing is done recursively: a polygon is simply drawn in the expected manner (with backface removal), an ordered list is drawn by drawing each of its members in order, an unordered set is drawn by drawing each of its members, and a BSP node is drawn in the same way as in a regular BSP tree (first the element on the other side of the node's splitting plane from the viewer is drawn, then the set of polygons coplanar to the splitting plane, and then the element on the same side as the viewer). The idea is that we want to have flat lists as much as possible, and only resort to BSP nodes where absolutely necessary.

The differences from the graph algorithms Naylor describes that seem closest to Q ((5) and (6)) are that Q does not need the obscuration graph edges at run-time, it does not need to do topological sorting at run-time, and it doesn't care about cycles themselves, only the component as a whole, so its preprocessing efficiency doesn't depend on the number of cycles any more than BSP-trees do. (This last is only a difference from (5).) Naylor observes that the BSP data structure "is on the order of the number of polygons remaining after the partitioning process while the above outlined approximation method using a directed graph is on the order of the number of edges between these polygons" [11, p. 74]. Algorithm Q also has run-time space requirements on the order of the number of resulting polygons. In the worst case, it uses exactly as much space per output polygon as the BSP variant in which splitting planes need not be those of input polygons; in one implementation this uses only about 30% more space per output polygon (assuming an average of four vertices per output polygon) in the worst case than the BSP variant where splitting planes must be those of input polygons.

For BSP trees, Naylor notes that the tree can be compressed when there is a chain of nodes with no children on one side (the same side for each node) [11]. The constant-order list L resulting from such a compression can be seen to be guaranteed to be part of a list from the graph algorithm given the polygons P of the BSP subtree rooted at the top node of the chain. Thus, if Q picks splitting polygons in the same way as the BSP preprocessor (assuming neither uses any information other than the polygons to be partitioned), eventually either it will pass the same polygons P to the topological sort routine, resulting in L being part of some list, or else some of the polygons in L will have already been output as part of some list. However, Q may sometimes produce lists that the compressed BSP tree wouldn't have, because it essentially actively seeks such lists at each level rather than just picking partitioning planes and hoping such lists appear. Simply drawing polygons in a list does not involve the additional recursion overhead of BSP traversal, even if an explicit stack is substituted for functional recursion, which saves both time and space. It also involves fewer conditional branches (and resulting pipeline-breaking in many architectures). It does, however, introduce the overhead of computed branches (i.e., branches to locations stored in a data structure) compared to the BSP variant that requires splitting planes to be those of input polygons. (This may seem obvious and trivial, but it was part of the motivation that led to this algorithm.) As a result, Q has the potential to be better than BSP at run-time even if the same number of polygon splits are made. Since Q's worst case occurs when every level is composed of one strong component, resulting in identical output to the BSP tree, Q can be no worse than the BSP algorithm at run-time: it stores no more data, as discussed above, and it has the potential to produce fewer BSP nodes and fewer split polygons.

5 Results

Algorithm Q has been implemented and tested on various models, including four models obtained from the internet whose results are presented here. The models were preprocessed by two algorithms: *BSP* is the plain BSP algorithm that allows splitting planes not to be those of input polygons, and Q is the algorithm described above.

The same criteria were used to determine partitioning planes in both algorithms, and in fact the code was the same, with a simple run-time switch to determine whether to attempt topological sorting at the appropriate stages. Developed in an attempt to balance between speed of preprocessing and optimality of result, the criteria include making various attempts at partitioning, using either just input polygon planes or all possible planes, and preferring either fewer split polygons or more balanced partitions, depending on the number of polygons to be split and how good the result of previous attempts has been. Details do not merit inclusion here as they could probably be improved a great deal (see [7] and also Section 6).

Table 2 shows statistics on the preprocessing output for the model of a mushroom. "BSPs", "Lists", "Sets" and "Polygons" give the number of the respective nodes in the output. "Splits" is the number of polygons split, and "List Length" and "Set Length" give statistics on the lengths of lists and sets, respectively, in the output. Finally, "Bytes" is the size in bytes that would be used at run-time by one implementation of the data structures, assuming floating-point values are represented as 8-byte doubles and type information is represented as a pointer to a table of functions. Similarly, Tables 3, 4, and 5 show statistics for models of an F-18 plane, a human head, and a polygonalized Utah teapot, respectively.

Run-time refresh-rate measurements were made using an HP 9000-735/125 workstation with CRX48Z graphics in a 512 by 512 8-bit-deep window. HP's Starbase was used to do scan-conversion (in 2D accelerated integer mode), and the program was run using HP's real-time scheduling facilities so that it was essentially the only user of the CPU. The object was drawn so that it took up most of the screen but did not need to be clipped, and was drawn either from the same viewpoint each frame (*static*) or rotating at about one revolution every 2π seconds (*rotating*). Every second, the frame rate over the last second was computed and the median of that reading and the previous two was placed in a ten-element circular buffer. Then, if at least thirteen such readings had been taken, the mean of the ten values in the buffer was recorded as one data point. Once 30 data points had been accumulated, the program wrote them out and exited. Several such runs were made for each condition, and graphs of the results were made and typical values determined. The results are in Table 1.

Model	Algorithm	Typical frame rates (static)	(rotating)
mushroom	BSP	32	33–40
	Q	36	45–53
F-18	BSP	36	35.4–35.9
	Q	36	36
head	BSP	10.3	8.6–10.3
	Q	10.3–11.5	10–12.3
teapot	BSP	2.9	2.8
	Q	3.9	3.7-3.9

Table 1: Run-time frame rates for the models under various conditions.

6 Future Work

The criteria used to select partitioning planes can have a significant effect on the results of both the BSP algorithm and Q. Experiments should be done to see to what extent this affects the practical improvement Q yields over plain BSP, in particular trying the heuristics in both [7] and [9].

7 Acknowledgements

Thanks to Steven Dollins, John Hughes, Robert Zeleznik, Katrina Avery, and Andries van Dam for reading this paper and making suggestions. This work was supported in part by grants from NSF, Microsoft, Sun Microsystems, Taco, and HP.

References

[1] Chin, N., and Feiner, S. Near real-time shadow generation using BSP trees. In *Computer Graphics (SIGGRAPH '89 Conference Proceedings)* (July 1989), ACM SIGGRAPH, pp. 99–106.

[2] Chin, N., and Feiner, S. Fast object-precision shadow generation for area light sources using BSP trees. In *1992 Symposium on Interactive 3D Graphics (Special Issue of Computer Graphics)* (1992), ACM SIGGRAPH, pp. 21–30, 220.

[3] Fuchs, H., Abram, G. D., and Grant, E. D. Near real-time shaded display of rigid objects. In *Computer Graphics (SIGGRAPH '83 Conference Proceed-*

Algorithm	BSPs	Lists	Sets	Polygons	Splits	List Length			Set Length			Bytes
						Min	Avg	Max	Min	Avg	Max	
Q	19	34	0	290	50	2	8.9	115				40916
BSP	344		5	378	138				2	2.4	3	66832

Table 2: Model of a mushroom: 226 vertices, 240 polygons, 10014 obscuration dependencies.

Algorithm	BSPs	Lists	Sets	Polygons	Splits	List Length			Set Length			Bytes
						Min	Avg	Max	Min	Avg	Max	
Q	62	71	11	401	47	2	5.1	29	2	3.4	10	57544
BSP	333		86	457	103				2	2.4	10	76856

Table 3: Model of an F-18: 330 vertices, 354 polygons, 25459 obscuration dependencies.

Algorithm	BSPs	Lists	Sets	Polygons	Splits	List Length			Set Length			Bytes
						Min	Avg	Max	Min	Avg	Max	
Q	95	115	34	1812	216	2	13.4	240	2	8.0	27	247100
BSP	782		434	2059	463				2	3.9	40	312880

Table 4: Model of a human head: 1434 vertices, 1596 polygons, 233513 obscuration dependencies.

Algorithm	BSPs	Lists	Sets	Polygons	Splits	List Length			Set Length			Bytes
						Min	Avg	Max	Min	Avg	Max	
Q	948	737	49	5611	1860	2	7.0	557	2	2.9	4	739568
BSP	4654		1269	6837	3086				2	2.8	6	1079172

Table 5: Polygonalized Utah teapot: 1976 vertices, 3751 polygons, 2625250 obscuration dependencies.

ings) (July 1983), P. Tanner, Ed., ACM SIGGRAPH, pp. 65–72.

[4] Fuchs, H., Kedem, Z. M., and Naylor, B. Predetermining visibility priority in 3-D scenes. In *Computer Graphics (SIGGRAPH '79 Conference Proceedings)* (August 1979), B. W. Pollack, Ed., ACM SIGGRAPH, pp. 175–181.

[5] Fuchs, H., Kedem, Z. M., and Naylor, B. F. On visible surface generation by a priori tree structures. In *Computer Graphics (SIGGRAPH '80 Conference Proceedings)* (July 1980), ACM SIGGRAPH, pp. 124–133.

[6] Gordon, D., and Chin, S. Front-to-back display of BSP trees. *IEEE Computer Graphics and Applications* (September 1991), 79–85.

[7] Morer, P., Garcia-Alonso, A. M., and Flaquer, J. Optimization of a priority list algorithm for 3-D rendering of buildings. *Computer Graphics Forum 14*, 4 (1995), 217—227.

[8] Naylor, B. SCULPT: An interactive solid modeling tool. In *Graphics Interface '90* (1990), pp. 138–148.

[9] Naylor, B. Constructing good partitioning trees. In *Graphics Interface '93* (1993), pp. 181–191.

[10] Naylor, B., Amanatides, J., and Thibault, W. Merging BSP trees yields polyhedral set operations. In *Computer Graphics (SIGGRAPH '90 Conference Proceedings)* (August 1990), ACM SIGGRAPH, pp. 115–124.

[11] Naylor, B. F. A Priori *Based Techniques for Determining Visibility Priority for 3-D Scenes.* PhD thesis, The University of Texas at Dallas, May 1981.

[12] Naylor, B. F. Interactive solid geometry via partitioning trees. In *Graphics Interface '92* (1992), pp. 11–18.

[13] Naylor, B. F. Partitioning tree image representation and generation from 3D geometric models. In *Graphics Interface '92* (1992), pp. 201–212.

[14] Paterson, M. S., and Yao, F. F. Efficient binary space partitions for hidden-surface removal and solid modeling. *Discrete & Computational Geometry 5*, 5 (1990), 485–503.

[15] Schumacker, R. A., Brand, R., Gilliland, M., and Sharp, W. Study for applying computer-generated images to visual simulation. Tech. Rep. AFHRL-TR-69-14, U.S. Air Force Human Resources Laboratory, September 1969.

[16] Sutherland, I. E., Sproull, R. F., and Schumacker, R. A. A characterization of ten hidden-surface algorithms. *Computing Surveys 6*, 1 (March 1974), 1–55.

[17] Thibault, W. C., and Naylor, B. F. Set operations on polyhedra using binary space partitioning trees. In *Computer Graphics (SIGGRAPH '87 Conference Proceedings)* (July 1992), ACM SIGGRAPH, pp. 153–162.