Query Driven BSP Trees *

Sigal Ar and Ayellet Tal
Department of Electrical Engineering
Technion – Israel Institute of Technology
{ar, ayellet}@ee.technion.ac.il

One of the fundamental problems in computational geometry, computer animation, solid modeling, robotics and virtual reality, is that of collision detection. Since it is needed in such a wide variety of applications, many collision detection methods have been proposed. Of the many algorithms and data structures used, we choose to focus on the practically popular Binary Space Partitioning (BSP) trees [1, 3, 6, 7, 8], which represent a data structure with a recursive search algorithm embedded in it. We explore BSP trees for collision detection in a new way.

Traditionally, BSP trees are kept small under the assumption that this will help keep traversal times low. Therefore, heuristics for choosing cutting planes aimed at that goal. One of the most common methods, used by what we call the *standard* BSP trees of the Graphics GEMS [5], is to choose the cutting plane so as to attempt to minimize the number of intersections of cutting planes with each other.

Constructing the tree using information about actual access patterns, rather than just heuristics, means that tree size is not as critical to efficiency. We base tree configuration on the assumption that events are more likely to happen in the future, if they have already happened in the past and incorporate information about likely queries. That is, rather than base BSP tree construction only on the properties of the scene, we gather information about how the tree is used to service ray-shooting queries and use this information to determine its structure.

With *self-customized* BSP trees [2], we first collect statistics about the expected ray-shooting queries, and then use this information to configure the tree. The idea is to hypothesize a probabilistic distribution of requests, based on a log of client usage. That information is then used to configure the data structure to improve its performance. This is done in two steps: (i) A learning step: Keeping a sampled log of past client requests and inferring from it a probability distribution over the request domain; (ii) A configuring step: Constructing the BSP tree to optimize its request servicing costs, given the inferred distribution.

When we do not have any, or enough, information to determine what a good tree is, we can use a different approach, based on ideas presented in [4], involving dynamic or query driven structuring. We *defer* full configuration and use a partially constructed tree to respond to queries, while continuing information collection. The statistics are collected in the (partial) tree itself. When it is determined that enough information has been collected, some more of the tree is constructed, based on this information. Thus, as with self-customized BSP trees, the structure of the *deferred, self-organizing* BSP tree reflects the way we expect to use it when answering queries.

*Fuller version as Technion technical report: Deferred, Self-Organizing BSP Trees for Collision Detection by S. Ar and A. Tal.

When using deferred data structures, there have to be criteria to decide when to continue configuration. In the case of deferred BSP trees, that means deciding which nodes, of those that can be, will be split by a cutting plane and when, to create more of the tree structure. Obviously, a node that is never accessed will not be considered for further splitting. Creating tree structure which is never used to respond to queries is wasted effort. Conversely, a node that is accessed frequently, and is one that may be further split, indicates a likely location where more BSP tree structure will be beneficial to better query processing times.

As indicated, when in use, the deferred BSP tree construction is only started. That is, the tree is only a root with a list of all scene faces, each corresponding to a potential cutting plane. Then, as it is used to respond to ray-shooting queries, accesses to nodes are counted. When a node is accessed and has reached the user-defined threshold for number of accesses, it is split into two nodes with a cutting plane chosen from its list of polygons, or faces, defining potential cutting planes.

This leads us to the question of maintaining the lists so that when the time comes, we choose a good cutting plane and we do so quickly. Every time a scene face, or polygon, is checked for collision, it is accessed. Thus, it would be advantageous to maintain the polygon lists so as to keep frequently accessed ones close to the head of the list (and thus also closer to the root of the tree).

There are some well-known, deterministic online algorithms used for list update, or reorganization. We have experimented with some of the most common ones: Move-To-Front, Transpose and Frequency-Count. In all three list maintenance methods, the lists are reorganized to bring frequently accessed polygons close to the head of the list. Thus, choosing the first face on the list as the one to define the next cutting plane is a good choice, and is obviously, the quickest.

Results

Variants of the algorithms involved in our various BSP tree constructions were implemented. We compared BSP trees produced using deferred, self-organizing methods to some highly optimized BSP trees. We first compared self-customized BSP trees to what we call *the standard* BSP trees of the Graphics Gems [5], Then, we compared both to our deferred self-organizing BSP trees.

To compare the performance, we counted the number of BSP tree nodes traversed, or *opened* (for all the rays) in each case - the Standard BSP vs. the Self-Customized BSP vs. the Deferred, Self-Organizing BSP trees. More importantly, we counted the total number of scene faces checked for collision in each case, since this is where most of the search work is done. In the Standard and Self-Customized trees, these are the faces defining the cutting planes of tree nodes encountered during the search. In the deferred, self-organizing trees, since they are only partially built, these may include both faces that define cutting planes of configured tree nodes and may also include lists of faces corresponding to candidate cutting planes, when the search is at a node that is not "split".

Figures 1-3 below present results of our experiments. Next to each object name, we indicate the number of faces comprising it. Each table header lists the total number of rays and the number of ray classes. Note that the classes may have different sizes. The table has a line for each of several runs: the standard BSP tree construction, the self-customized BSP tree construction, and two lines for deferred tree construction runs. The lines for the deferred runs each indicate the threshold type used to determine when to split a node. This may be either a constant threshold, or a "history" threshold.

Our experiments indicate that deferred BSP trees can offer stunning speedup factors, even over self-customized BSP trees. The marked advantage of the deferred BSP trees comes from several factors. First, using the actual queries to "train" the data structure for future queries, as is done with self-customizing. The learning process suggests picking the first BSP cutting plane as the one that the rays are actually expected to hit first. So, when the rays actually hit a polygon, it is hopefully near the root of the tree, and the hit is found quickly. Second, the partially constructed trees are small, so searches through the fully configured parts of the tree are quick. Third, the self organizing lists keep searches for collisions efficient in all tree parts, in spite of the lacking tree structure.

Note also that, though not shown in the tables, there are other obvious advantages to the deferred BSP trees being much smaller than both the standard BSP trees and the self-customized ones. Obviously, smaller trees mean less storage space. More importantly, little configuration ahead of use means big savings in pre-processing time. When tree configuration is fully deferred (none done ahead of query processing), there is virtually no preprocessing time. Although time is spent both building and re-organizing the deferred self-organizing tree (this time is interleaved with query processing), it is done only when deemed advantageous, in that it pays off in faster processing for future queries.

Deferred, self-organizing BSP trees successfully attempt to maintain the obvious advantages of self-customized BSP trees, while putting an emphasis on keeping the trees small and saving on preprocessing time. Partial structuring has several obvious advantages: Conveniently, there is no need for prior information. Partial tree construction can start, using any available information about queries, even if that information is not statistically significant. In addition, since they are only partially constructed, the BSP trees produced are much smaller. Most importantly, in spite of using only partial information and in spite of the data structure being very simple when not fully constructed, the response to queries is very cost effective. Deferred, self-organizing BSP trees are not only an improvement over standard BSP trees, but over self-customized BSP trees as well.

References

[1] Agarwal, P.K., Guibas, L.J., Murali, T.M., Vitter, J.S. *Cylindrical static and kinetic binary space partitions*, Proc. 13th Annu. Symp. Comput. Geom. (1997), 39-48.

[2] Ar, S., Chazelle, B., Tal, A. *Self-Customized BSP Trees for Collision Detection*, to be published in Computational Geometry: Theory and Applications.

[3] Fuchs, H., Kedem, Z.M., Naylor, B. *On visible surface generation by a priori tree structures*, Proc. SIGGRAPH '80, Comput. Graph., 14 (1980), 124–133.

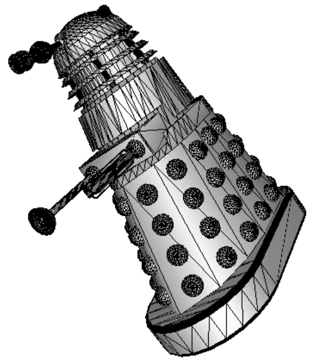
[4] Karp R.M., Motwani R. and Raghavan P. *Deferred Data Structuring*, SIAM Journal on Computing, 17 (1988), pp. 883-902

[5] Paeth A.W. *Graphics Gems V*, Academic Press, 1995.

[6] Paterson, M.S. and Yao, F.F. *Efficient binary space partitions for hidden-surface removal and solid modeling*, Disc. Comput. Geom., 5 (1990), 485–503.

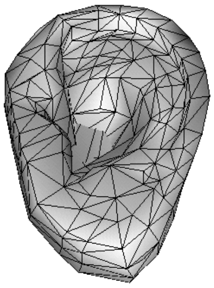
[7] Sung, K., Shirley, P. *Ray tracing with the BSP tree*, ed. David Kirk, Graphics Gems III, Academic Press Inc. (1992), 271–274.

[8] Teller, S.J., Sequin, C.H. *Visibility preprocessing for interactive walkthroughs*, Proc. SIGGRAPH '91, Comput. Graph., 25 (1991), 61–69.



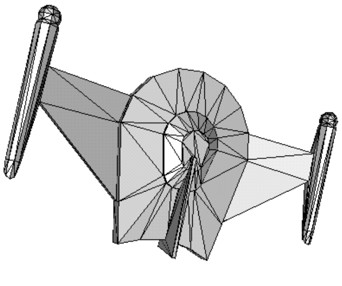
30,000 rays, two classes			
run type	# nodes	# faces	tree size
Standard	3995683	868538	68407
Self-Customized	780988	246709	86057
Deferred, Constant	67084	48926	247
Deferred, History	68270	44841	261

Figure 1: Dalek - 21,814 faces



60,000 rays, five classes			
run type	# nodes	# faces	tree size
Standard	8950467	1580270	1785
Self-Customized	4799768	759976	3139
Deferred, Constant	1166721	237213	169
Deferred, History	1130651	248176	127

Figure 2: Ear - 454 faces



30,000 rays, two classes			
run type	# nodes	# faces	tree size
Standard	1197159	445056	949
Self-Customized	392441	153367	1997
Deferred, Constant	61771	61851	7
Deferred, History	61775	61824	7

Figure 3: Romulan - 431 faces