EUROGRAPHICS 2002 / G. Dettrakis and H.-P. Seidel
(Guest Editors)

Volume 21 (*2002*), *Number 3*

Deferred, Self-Organizing BSP Trees

Sigal Ar, Gil Montag and Ayellet Tal†

Department of Electrical Engineering, Technion – Israel Institute of Technology, Haifa, Israel

Abstract

BSP *trees and KD trees are fundamental data structures for collision detection in walkthrough environments. A basic issue in the construction of these hierarchical data structures is the choice of cutting planes. Rather than base these choices solely on the properties of the scene, we propose using information about how the tree is used in order to determine its structure. We demonstrate how this leads to the creation of* BSP *trees that are small, do not require much preprocessing time, and respond very efficiently to sequences of collision queries.*

Categories and Subject Descriptors (according to ACM CCS): I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling I.3.6 [Computer Graphics]: Graphics data structures and data types, Interaction techniques I.3.7 [Computer Graphics]: Virtual reality

Keywords: Binary space partitioning, collision detection, ray-shooting, walkthrough, self-organizing data structures, deferred data structures.

1. Introduction

Ray-shooting and collision detection are fundamental problems in computer graphics. Of the many algorithms and data structures used [5, 7, 11, 12, 14, 15], we choose to focus on Binary Space Partitioning (BSP) trees [1, 9, 10, 13, 17, 19, 20, 23, 24, 25], which represent a data structure with a recursive search algorithm embedded in it. We explore BSP trees for collision detection in walkthrough environments and for ray shooting in a new way.

Obviously, there are many possible BSP trees for any given scene. The major consideration in the construction of a BSP tree is the choice of a "good" cutting plane. Traditionally, cutting planes are chosen so as to hopefully keep the BSP tree small under the assumption that this will help keep traversal times low. It is not known, however, how to find the smallest BSP tree for a given scene. It was shown that random constructions can limit the size to quadratic [19] and it seems much less in practice [8]. In actual applications, people employ greedy heuristics [18], which cannot guarantee small trees (nevertheless, often produce them). Moreover, as is shown in [3], tree size is not necessarily a significant factor in the time required to answer a query, for query types which need not explore the whole tree.

The expectation that small trees will keep query response times low amounts to attributing equal likelihood to each query. In other words, assuming a uniform distribution on the queries. However, in "real life", that is not the case. Three-dimensional scenes are not random; nor are ray-shooting queries or navigation paths in walkthrough systems. People, and even robots, rarely navigate randomly through a building in a walkthrough environment while aimlessly bouncing against the walls.

Our basic premise is that queries are typically dependent on previous ones. For instance, certain paths in a museum walkthrough are more popular than others. (Think, for example, of the room or hallway where Venus de Milo is displayed.) Therefore, our underlying assumption is that data structure construction should not depend solely on the static data set, but rather incorporate information about likely queries. That is, rather than base BSP tree construction only on the properties of the scene, as is customary, we propose gathering information about how the tree is used to service ray-shooting or collision-detection queries and use this information to determine the BSP tree's structure. This can be done if tree construction is delayed until there is some in-

† This research was supported by the fund for the promotion of research at the Technion.

formation about the nature of the queries it will be used to service.

Naylor [17] was the first to propose constructing a cost model for BSP trees based on the probability distribution of the input (i.e., the rays) or some estimate of this probability. In practice, it is assumed in [17] that the distribution is uniform, and thus the relative area can be used to estimate it.

As noted above, uniform distribution is not necessarily representative. Ar, Chazelle & Tal [3] presented the *self-customized BSP tree*, where tree construction is based on the same temporal coherence principle used in caches: Events are more likely to happen in the future, if they have already happened in the past. This is done by hypothesizing a probabilistic distribution of requests based on a log of recent client usage. After this learning stage, that information is used to configure the BSP tree to optimize its expected request-answering complexity. It has been demonstrated that using probabilistic cost models to guide BSP tree construction results in trees which outperform other forms of BSP trees for ray shooting queries.

But, what about situations where we do not have any, or enough information ahead of time to determine the distribution? We would still like to be able to service queries efficiently. This is the topic of the current paper.

The strategy we explore in this paper, is to *defer* full configuration and use a partially constructed tree to respond to queries, while continuing information collection. This leads us to propose a new BSP tree scheme – a *deferred, self-organizing* BSP tree. The statistics about the queries are represented in the structure of the (partial) tree itself. When it is determined that enough additional information has been collected, some more of the tree is constructed, based on this information. Thus, the structure of the deferred, self-organizing BSP tree evolves, reflecting the way it is expected to be used, when answering future queries. Obviously, the same scheme can be applied to a KD tree which is a special case of a BSP trees.

Our deferred, self-organizing BSP trees successfully attempt to construct trees that suit the client's usage, while putting an emphasis on keeping the trees small and saving on preprocessing time. We will show that our proposed partial structuring scheme 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. 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. In addition, since these trees are only partially constructed, they are much smaller than trees produced by other BSP tree construction methods.

Though small trees are not required for answering queries efficiently [3], there are still many advantages to reducing tree size. For example, think of a multi-user virtual reality walkthrough environment, where each user "walks" along a unique path in the environment. Different paths, on their own, do not require more than a single BSP tree for the whole system. However, different BSP trees can be beneficial for query processing when each tree fits the usage patterns of a specific user. Obviously, it would be advantageous if each of these trees did not require too much space.

The rest of this paper is organized as follows. In Section 2, we define *deferred, self-organizing* BSP trees and describe how they are constructed. In Section 3, we present our experiments and explain the results. We briefly conclude in Section 4.

2. Definitions and Tree Construction

A BSP tree is a method for partitioning n-dimensional space using $(n-1)$-dimensional hyperplanes. Once a space has been partitioned by a hyperplane, it is represented by two n-dimensional sub-spaces, one on each side of the partitioning hyperplane. These can now be recursively partitioned.

Our scene is given as a set S of disjoint polygons in $\mathbf{R}^3$. These are the *scene polygons.* The corresponding BSP tree is a binary tree, where each node v is associated with a partitioning plane π_v and a closed convex polyhedron C_v. The root's polyhedron is a large box enclosing the entire scene. If v is not a leaf, the plane π_v cuts C_v into the two convex regions associated with the children of v. We restrict ourselves to auto-partitioning BSP trees: ones where each cutting plane contains a scene polygon.

The basic question in the construction of BSP trees is how to choose the cutting planes. We answer this question by proposing a new BSP tree scheme – one which is both *Deferred* and *Self-Organizing.*

Deferred data structures, first suggested by Karp et al. [16], involve dynamic or query driven structuring. The idea is to process the data set only when doing so is required for answering a query, that is *during* the query processing phase. This contrasts with conventional data structures, that are fully configured prior to any query answering accesses.

A self-organizing data structure [2, 4, 6, 21, 22] is a conventional data structure with rules or algorithms for changing itself, potentially after each access. The rules are designed to respond to initially unknown properties of the input request sequence, and to get the data structure into a state that will take advantage of these properties and reduce the time per operation. Unlike deferred data structures, self-organizing data structures are fully configured ahead of time.

Both deferred and self-organizing data structures are designed to address the issue of responding efficiently to queries when there is not enough prior information about the query distribution. In what follows, we show how to combine these two notions in the context of BSP trees.

We define a *deferred, self-organizing* BSP tree as a partially constructed BSP tree. It will have a root node, and may have some of the tree structure below the root, but not all of its leaves will be proper BSP tree leaves. Each leaf of a partially constructed tree may have a list of scene faces, whose corresponding planes are the potential cutting planes for that BSP node. These face lists are managed with a set of rules for their potential re-organization with each tree access.

Note that the tree is used to respond to queries even before it is fully constructed. To do this, we start at the root of the tree and recurse downwards, checking for collisions against the polygons associated with the relevant tree nodes, in the standard way. When we get to a node that does not have a subtree configured below it, we check for collision by performing a sequential search through that node's list of potential cutting planes.

TREE CONSTRUCTION

A BSP tree is built incrementally by inserting cutting planes one at a time. At any time during the construction, i.e., at any "leaf", a list is kept. This is a list L of the scene planes (i.e., the planes coplanar with polygons of S) that are potential cutting planes for that node. When beginning tree construction, the single leaf-node is the root of the tree, and all scene polygons are on its list. To continue construction of the BSP tree at a given node, one of the potential cutting planes needs to be chosen, to "split" that node. The list of scene faces at that node then needs to be parceled out to its two children nodes. A fully constructed tree is obtained when this process can no longer continue.

When executing a sequence of queries on a BSP tree, the total access time will be small if frequently accessed items are near the root of the tree. Thus, if we expect past queries to reflect likely future one, we want cutting planes at nodes near the root of the BSP tree to be ones corresponding to polygons likely to be hit. To avoid extensive log-keeping, the collected information is represented in the data structure, bypassing the need to create a probabilistic model based on the input.

Given a partially constructed tree, we may, at intervals, want to configure more of the tree. That is, we may want to "split" more of the tree nodes. There are several issues regarding splitting a node having a list of potential cutting planes. First, **when** do we split a node? Second, the closely related question of **which** node do we split? Third, there is of course the question of **how** to maintain the lists of potential cutting planes at nodes that may be split, so that choosing the actual cutting plane may be done quickly. In what follows, we will answer these three questions.

NODE SPLITTING

Intuitively, we would like to render more structure to a part of the tree that is used a lot for query processing. If we have reached a node frequently, and it may be split, that means useful information is sitting in the lists, rather than in BSP tree structure. We would like to put that information into the tree structure, by choosing as cutting plane one that will help reduce query processing time in the future.

When using deferred data structures, there needs to be a criterion for deciding 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.

Thus, we count accesses to nodes of the partially constructed BSP tree during the processing of queries. It is time to split a node when that node is accessed, and the count for that node has reached a pre-specified threshold. This threshold may be of one of two kinds: *constant* or *relative*. A constant threshold specifies the number of allowed accesses to a node before it is split to two nodes by a cutting plane. A relative threshold specifies the number of accesses to a node, as a fraction of total accesses to the BSP tree, before the node is split. Below, when a walkthrough application is discussed, another possibility will be described.

In actual use, the 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. When a node is accessed and has reached the threshold for number of accesses, it is split into two nodes with a cutting plane chosen from its list of polygons. If the polygon lists are maintained such that the frequently accessed polygons are brought close to the head of the list, choosing the first on the list is a good choice, and is obviously the quickest. Next we discuss list maintenance methods.

MAINTAINING FACE LISTS

The potential cutting planes maintained at each BSP tree node, need to be ranked based on query patterns. Rather than maintain an explicit score for each plane, we would like to keep the ranking information in the structure of the lists. This brings us to the list update problem of online algorithms, where a list is re-organized, potentially after each access, with the goal of speeding up query service time for future accesses.

In a partially constructed BSP tree, when the search for collision reaches a node with a polygon list, each list element is examined until a collision is found, or it is determined that there is none. 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. This will help keep query processing times low, and will keep those polygons readily available when it is time to choose a cutting plane.

There are some well-known, deterministic online algorithms used for list update, or re-organization. We experimented with the most common ones, which we list below.

- **Move To Front:** Move each accessed item to the front of the list.
- **Transpose:** Exchange the accessed item with the one immediately before it in the list.
- **Frequency Count:** Maintain a frequency count for each list item. Whenever an item is accessed, increase its frequency count by one. Maintain the list so that items are sorted in non-increasing order of their frequency counters.

Note that Move-To-Front and Transpose define memoryless algorithms, while Frequency-Count is a strategy that needs some additional memory in the data structure. However, in either case, the self-organizing rules can be viewed as putting the "memory" about the history of past requests into the data structure, by re-organizing it.

TREE RE-CONFIGURATION

There are times when access patterns change dramatically. If this situation arises and large parts of the tree are already configured, response to queries may not be as efficient as they could be. If usage indicates this situation, we choose to start the relevant sub-tree configuration from scratch.

3. Experimental Results

We have tested our deferred self-organizing BSP trees for two applications – ray shooting and collision detection in walkthrough environments. In this section we summarize our findings and analyze them.

RAY SHOOTING EXPERIMENTATION

Given a ray specified by a point p and a direction ℓ, and a scene S represented by a deferred self-organizing BSP tree, we can find the first polygon of S that the ray hits by recursing on the following process. Assume that the ray is known to cross C_v, the polyhedron associated with a tree node v and with a cutting plane π_v. If v is a BSP leaf, then we may compute the answer by exhaustive examination of all the scene polygons that intersect C_v; otherwise, check if the ray hits a scene polygon associated with node v. If it does, we have a collision. If there is no collision in v, recurse in either child whose polyhedron lies on the same side of π_v as p.

We compared BSP trees produced using deferred, self-organizing methods to two optimized BSP tree types. The first kind of BSP tree we compare against is what we call the *standard* BSP tree which is based on the BSP code presented in the Graphics Gems [18], and which we further optimized. The strategy used when constructing a standard BSP tree is to always choose a cutting plane which "cuts" the minimal number of scene faces. In other words, this is a greedy procedure with the goal of minimizing the number of intersections between cutting planes and thus minimizing the tree size. The second kind of BSP tree we compare against is the *self-customized* BSP tree [3], which bases the scoring of the cutting planes on the hypothesized probabilistic distribution of a history of queries.

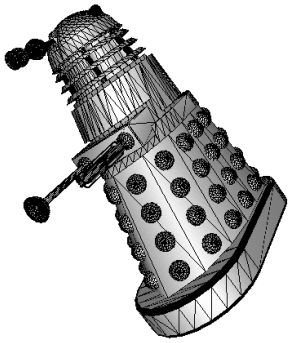
To compare the performance of these BSP schemes, we use two main measures for the traversal cost of a directed line in a BSP tree. We count the number of tree nodes accessed before the first collision is found. More importantly, we count the number of scene polygons that are checked for collision with the ray, before a collision is found, or it is determined that there is none. 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 yet "split". These checks consume the bulk of the computation time.

To benchmark the performance of any given BSP tree, we produced ray shooting queries assuming a multivariate normal distribution. Similar distribution parameters were used to simulate the learning process in the self-customized BSP trees [3].

Figures 1–2 present results of our experiments. Next to each object name, we indicate the number of faces comprising it. Every object is associated with a table. Each table header lists the total number of rays and the number of ray classes (of the multi-variate distribution). 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, self-organizing tree construction runs, indicating the threshold type used to determine when to split a node (constant or relative). The first column indicates the the number of tree nodes accessed before the first collision is found. The second column indicates the number of scene polygons that are checked for collision. The third column indicates the size of the BSP tree.

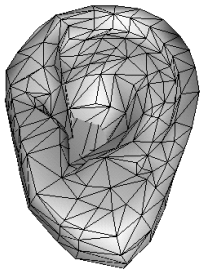
We show the results of only one run of each possible deferred threshold kind, although we performed many more. We varied the threshold for each type, and we experimented with all three methods of list maintenance. Our conclusion is that frequency count is the most cost effective list maintenance method.

Our experiments indicate that deferred, self-organizing BSP trees can offer stunning speedup factors. For instance, in the case of the Dalek with $30,000$ rays of two different classes, the deferred BSP tree checks for collision against



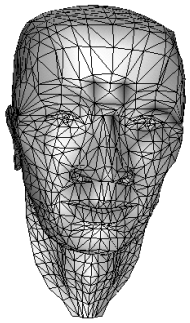
30,000 rays, two classes			
run type	# nodes	# faces	tree size
Standard	3995683	868538	68407
Self-Customized	780988	246709	86057
Deferred, Constant	67084	48926	247
Deferred, Relative	68270	44841	261

(a) 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, Relative	1130651	248176	127

(b) Ear - 454 faces



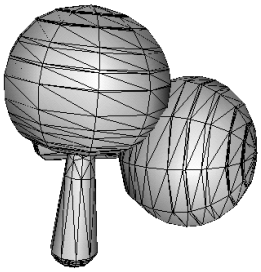
120,000 rays, five classes			
run type	# nodes	# faces	tree size
Standard	7473397	1169935	7161
Self-Customized	2854606	716882	15395
Deferred, Constant	1843284	487166	245
Deferred, Relative	1924076	511810	323

(c) Human Head - 1850 faces

Figure 1: *Ray shooting experimental results*

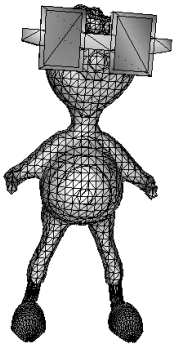
48,926 faces (1.63 faces per ray, on average), while the self-customized BSP checks 246,709 (8.22 faces per ray), and the standard BSP checks 868538 faces (28.95 faces per ray).

The preprocessing time for the standard BSP may get quite substantial in the case of scenes made up of many faces. For instance, in the case with the Nerd, which has 7,312 faces, the preprocessing time for the standard BSP is 510,326,810 microseconds. In the same case, the preprocessing for the self-customized BSP is 76,606,230 microseconds. Obviously, there is virtually no preprocessing time for a deferred BSP tree. 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. This process is very quick since the self-organization of the lists means that the first face is always picked, and no complex computation is needed. Moreover, this is done very rarely. At the end of each run, very few nodes have actually been split, as can be seen with the final trees being very small. For instance, in the case of the Dalek, the standard BSP tree has 68407 nodes, the self-customized BSP tree has 86057 nodes, while the deferred self-organizing



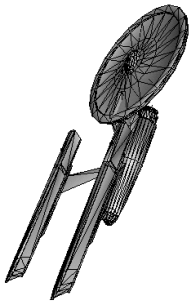
20,000 rays, one class			
run type	# nodes	# faces	tree size
Standard	8765706	1273440	17941
Self-Customized	3146212	554789	25009
Deferred, Constant	1281526	329239	843
Deferred, Relative	1336004	399571	883

(d) Moraccas - 3251 faces



30,000 rays, two classes			
run type	# nodes	# faces	tree size
Standard	6168995	689758	36199
Self-Customized	1168756	570634	52051
Deferred, Constant	1163687	496080	353
Deferred, Relative	1176089	492143	325

(e) Nerd - 7312 faces



45,000 rays, three classes			
run type	# nodes	# faces	tree size
Standard	3181464	973197	9085
Self-Customized	2734468	529039	18069
Deferred, Constant	1713029	428393	897
Deferred, History	1637907	383095	1039

(e) Enterprise - 1989 faces

Figure 2: *Ray shooting experimental results (Cont')*

BSP tree, which is only partially built, has only 247 (261) nodes.

WALKTHROUGH COLLISION EXPERIMENTATION

To achieve the feeling of presence in a walkthrough setting, one must address many issues, a key one among them is that of collision detection. The user should be kept from colliding with walls or other obstacles. In this section we describe our experiments using our deferred self-organizing BSP trees as our data structure for representing walkthrough scenes.

Recall that the BSP tree is built incrementally, by inserting one cutting plane at a time. In a walkthrough environment this is done while the user is "walking" within the given scene. The first walk can thus be considered as a "training" walk, and subsequent similar walks use the partial BSP trees constructed during the training walk (and can of course continue construction).

As before, there are three questions which need to be answered: When do we split a node? Which node do we split? How to maintain the lists of potential cutting planes at nodes that may be split?

In general, our goal is to construct a tree which optimizes collision detection queries for a specific walk. Obviously, faces the walker is more likely to collide with should be selected as cutting planes. It is a wasted effort to use faces which are far from the walker. In other words, the path the walker uses should determine the structure of the BSP tree.

On every step the walker makes, we consider splitting the BSP node the walker is currently in. Of all the faces which belong to this node's face candidate list, we wish to select the face closest to the walker as a cutting plane. However, if the closest face is still far away from the walker, it is not necessary to grow the tree at all, and no cutting plane should be chosen. This is somewhat different from the ray-shooting application, where a node is split when some threshold is reached.

To choose the cutting plane efficiently, the candidate list at every node should be sorted according to the distance from the walker. Of course, keeping theses lists updated at all times would be costly. Instead, we use self-organizing lists. Initially, all the faces are sorted according to their distance from the walker. Since the steps the walker makes are small, very few changes need to be done to the list at every step. In fact, only a few faces from the head of the list need to be checked. Moreover, when the list needs to be modified, one of the nodes at the head of the list should move forward. This brings us to the "move to front" method described above.

Obviously, only lists at the relevant nodes should be updated, and other nodes' lists are updated only when becoming relevant. At times, a whole list needs to be re-sorted. This happens either when many modifications are done to a list, or when the the current location of the walker is far from the location during previous re-sorting.

Figure 3 demonstrates the construction of the BSP tree, as done during a walk in the scene. In Figure 3(a), the walkthrough scene is shown together with a specific walk (in a dashed line). In Figure 3(b), the BSP tree, as built for this walk, is shown. It can be seen that though the scene consists of 24 possible cutting planes, the constructed tree is very small (having only 9 internal nodes). Only relevant planes, those that are close to the walk, are used as cutting planes. Other planes are left un-explored.

As the walk begins, plane 1 is the closest to the walker, and thus is chosen as the first cutting plane, and is associated with the root of the tree. On the second step, plane 15 is chosen since it is almost as close. As the walker keeps walking, the tree need not be further constructed, since all relevant faces are still far. Once the walker gets closer to the first intersection, Plane 2 becomes sufficiently close and is chosen as the next cutting plane. Similarly, Plane 16 is used as a cutting plane. Next, Plane 5 becomes the closest and is chosen as a cutting plane. This is done first on the left branch of the root node, and after the walker turns right, on the right branch. As this walker is approaching the next intersection, Plane 4 is chosen as the next cutting line, followed by Plane 3 (on both branches of node 5.2). This ends the construction of the BSP node for this specific walk. It is not necessary to explore all the other cutting planes which are far from the walker.

To test our data structure, we let a user move through various walkthrough scenes, using both deferred, self organizing BSP trees and standard BSP trees. (Self-customized BSP trees are designed for ray shooting applications.) During the first walk, the deferred, self organizing BSP tree is built. During subsequent similar walks, this tree is used (and construction continues when necessary). Recall that the main advantage of our scheme is when it is used for these subsequent similar walks, since each tree is optimized for a specific walk.

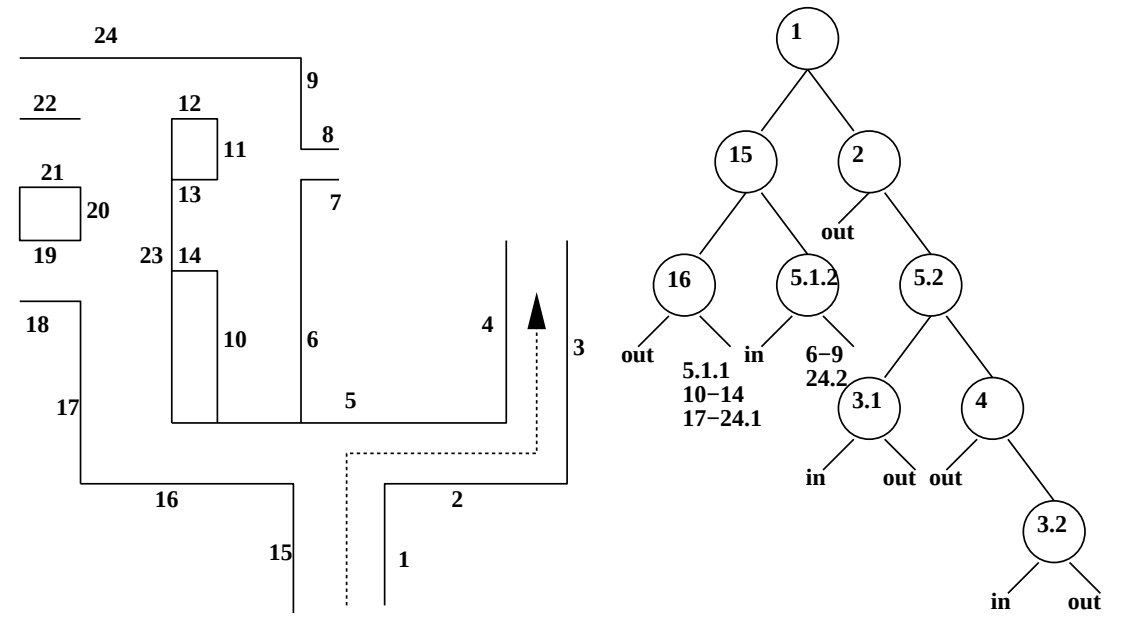
Figure 4 illustrates our results. Our deferred BSP tree is compared to a standard BSP tree. Each walkthrough scene is described by a table and is accompanied with an image of the model. See also the color section.

The first column of the table shows the tree sizes. Though standard BSP trees optimize tree sizes, the deferred, self organizing BSP trees maintain only partial constructed trees, hence their significantly smaller sizes. The other three columns show times in milliseconds. In the "walk" column, the walkthrough time using a similar path to the training path is given. In the "construct" column" the construction time is given. Since in deferred BSP trees the construction time interleaves with the time of the first walk, we measure the construction time plus the walk time also for standard BSP trees. Finally, in the "track II" column the times to walk through a different path than the training path is measured.

It can be seen that the deferred BSP tree outperforms the standard BSP tree. For instance, in the case of the castle (Figure 4(a)), walking through the training path takes 30ms for deferred BSP trees and 2273ms for the standard trees. Moreover, using a different path (entering the castle through the bridge rather than through the lobby) takes 271ms using a deferred BSP tree. The training walk through the castle, during which the tree is built, takes 1793ms.

Analysis

The marked advantage of the deferred BSP trees comes from several factors. The first factor is the use of the actual queries to "train" the data structure for future queries. This training process suggests picking the BSP cutting planes as the ones that are expected to be hit (or close) in the future. Note that a similar coherence principle is used in the construction of self-customized BSP trees. However, in this case the data is used within a function that estimates the efficiency of candidate cutting planes. In the case of deferred BSP trees the choice of a cutting plane is more directly dependent on the actual use of the tree, and thus the advantage over self-customized BSP trees. The second factor is that the partially constructed trees are small, so searches through the fully configured parts of the tree are quick. Finally, the self-



(a) The walkthrough scene (b) The deferred self-organizing BSP tree

Figure 3: *The deferred self-organizing* BSP *as built throughout the walk*

organizing lists keep searches for collisions efficient in all tree parts, in spite of the lacking tree structure.

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. This saving can be significant in a walkthrough environment where the scene is quite large and therefore a fully configured BSP tree is likely to be very big. Small trees let each user maintain a tree that suits his or her access patterns.

Moreover, little configuration ahead of use means big savings in preprocessing time. In the standard BSP trees, preprocessing time is spent on calculating intersections of potential cutting planes. The bigger the scene the more time spent on this preprocessing. In the self-customized BSP trees, preprocessing time is spent on classification of the training data, later used to give scores to potential cutting planes. This time is independent of the number of scene polygons. Deferred, self-organizing BSP trees do not need pre-processing at all.

4. Conclusions

Traditionally, BSP trees are constructed without any consideration to their use, configuration being based only on the scene for which they are built. In this paper this approach has been challenged. Instead, tree access information has been utilized in the construction of *deferred, self-organizing* BSP trees.

With deferred, self-organizing BSP trees prior information is not necessary. Rather, the tree is partially constructed whenever enough information about its use is available, and whenever it is deemed beneficial to configure more of the tree. We show that while the trees are indeed kept small, due to being only partially constructed, the response to queries is very cost effective.

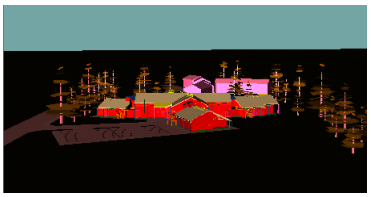
Although not necessary for processing queries efficiently, we show other advantages to keeping the trees small, most notably, saved preprocessing time and saved storage space.

To benchmark the performance of deferred, self-organizing BSP trees on collision detection or ray shooting queries, we compared the query answering costs of the deferred, self-organizing BSP trees to those of some public domain BSP trees. On the basis of our experimental results, it is clear that deferred data structuring, with re-organization can minimize both preprocessing time and storage requirements. Obviously, the main advantage is that this can be done while greatly improving query response time.

We deem the combination of deferred data structures with self-organizing data structures worthy of further investigation, perhaps for other data structures. Moreover, in the context of BSP trees, potential other domains where deferred, self-organizing BSP trees may prove beneficial include point location and range searching.

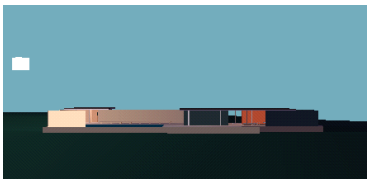
BSP type	tree size	walk (ms)	construct (ms)	track II (ms)
Standard	17442	2273	3025	390
Deferred	552	30	1793	271

(a) Castle - 4686 faces



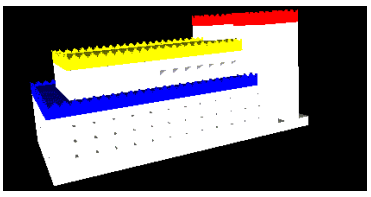
BSP type	tree size	walk (ms)	construct (ms)	track II (ms)
Standard	270455	221058	238423	22470
Deferred	811	261	160841	531

(b) Houses - 28995 faces



BSP type	tree size	walk (ms)	construct (ms)	track II (ms)
Standard	15165	1312	1883	3925
Deferred	822	20	1502	480

(c) Barcelona - 5338 faces



BSP type	tree size	walk (ms)	construct (ms)	track II (ms)
Standard	48374	15132	17335	9436
Deferred	116	40	15803	140

(d) Temple - 28995 faces

Figure 4: *Walkthrough collision experimental results*

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. Albers, S. and Westbrook, J. *Self-Organizing data Structures*, from Online Algorithms. The State of the Art, LNCS State of the Art Survey, Fiat, A. and Woeginger G. J., (editors). Springer.
3. Ar, S., Chazelle, B., Tal, A. *Self-Customized BSP Trees for Collision Detection*, "Computational Geometry: Theory and Applications", Vol. 15, February 2000, 91–102.
4. Arge, L. *The buffer tree: a new technique for optimal I/O-algorithms*, in "Algorithms and Data Structures," eds. Akl, S.G., Dehne, F., Sack J.-R., Santoro, N., Springer (1995), 334–345.
5. Aronov, B., Fortune, S. *Average-case ray shooting and minimum weight triangulations*, Proc. 13th Annu. ACM Symp. Comput. Geom. (1997), 203-211.

6. Bachrach, R., El-Yaniv, R. and Reinstadter, M., *On the competitive theory and practice of list accessing algorithms*, Submitted to Algorithmica.

7. Barequet, G., Chazelle, B., Guibas, L.J., Mitchell, J., Tal, A. *BOXTREE: a hierarchical representation for surfaces in 3D*, Graphics Forum, 15 (1996), C–387–396.

8. de Berg, M., van Kreveld, M., Overmars, M., Schwarzkopf, O. *Computational Geometry: Algorithms and Applications*, Springer, 1997.

9. Chrysanthou, Y., and Slater, M., *Computing dynamic changes to BSP trees*, Computer Graphics Forum (EUROGRAPHICS '92 Proceedings), 11 (1992), 321–332.

10. Chrysanthou, Y., and Slater, M., *Shadow Volume BSP Trees for Computation of Shadows in Dynamic Scenes*, ACM SIGGRAPH Symposium on Interactive 3D Graphics, (1995), 45-49.

11. Cohen, J., Lin, M., Manocha, D., Ponamgi, K. *I-COLLIDE: an interactive and exact collision detection system for large-scaled environments*, Proc. ACM Int. 3D Graphics Conf. (1995), 189–196.

12. Dobkin, D.P. and Kirkpatrick, D.G. *Fast detection of polyhedral intersection*, Theoret. Comput. Sci., **27** pp. 241–253 (1983).

13. 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.

14. Gottschalk S., Lin, M.C., and Manocha, D. *OBBTree: A Hierarchical Structure for Rapid Interference Detection*, Proc. SIGGRAPH '96, 171–180.

15. Held, J.T. Klosowski, and J.S.B. Mitchell, *Evaluation of collision detection methods for virtual reality fly-throughs*, Proc. 7th Canadian Conf. Computational Geometry, pp. 205–210 (1995).

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

17. Naylor, B. *Constructing Good Partitioning Trees*, Graphics Interface 1993, pp 181–191.

18. Paeth A.W. *Graphics Gems V*, Academic Press, 1995.

19. 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.

20. Samet H., *Spatial Data Structures: Quadtrees, Octrees, and Other Hierarchical Methods*, Addison-Wesley, Redding, Mass., 1989.

21. Sleator, D.S. and Tarjan, R.E. *Self-adjusting heaps*, SIAM J. Comput. 15 (1986).

22. Sleator, D.S. and Tarjan, R.E. *Self-Adjusting Binary Search Trees* JACM Volume 32, No 3, July 1985, pp 652-686.

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

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

25. Torres, E. *Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes*, EUROGRAPHICS '90, Eurographics Association (1990), 507–518.