A Segment-Tree Based Kinetic BSP[*]

M. de Berg
Institute of Information and Computing Sciences
Utrecht University
P.O.Box 80.089
3508 TB Utrecht
the Netherlands

J. Comba
Department of Computer Science
Stanford University
Stanford, CA 94305
USA

L.J. Guibas
Department of Computer Science
Stanford University
Stanford, CA 94305
USA

ABSTRACT

We present a new technique to maintain a BSP for a set of n moving disjoint segments in the plane. Our kinetic BSP uses $O(n \log n)$ storage and it undergoes $O(n^2)$ changes in the worst case, assuming that the endpoints of the segments move along bounded-degree algebraically defined trajectories. The response time (the time needed to update the BSP when it undergoes a change) is $O(\log^2 n)$. A randomized variant achieves $O(\log n)$ expected response time, while the worst-case response time remains $O(\log^2 n)$.

The new BSP is based on a simple technique for maintaining a 1-d segment tree on a set of intervals on the real line with moving endpoints. The response time of the kinetic segment tree is $O(\log n)$ in the worst case, and $O(1)$ expected.

1. INTRODUCTION

Many geometric algorithms are based on spatial partitions. One of the most popular methods to obtain a decomposition of space is with a *Binary Space Partition*, or BSP for short. In a BSP, the space is split with a hyperplane into two subspaces, which are again split with a hyperplane, and so on. The splitting process continues recursively until the subspaces are intersected by only one of the objects in the scene. (Throughout the paper we assume that the objects in the scene do not intersect each other—otherwise we cannot require that each terminal subspace contains only one object.) Observe that the objects in the scene can be fragmented by the splitting process; the different fragments of an object will then end up in different leaves.

BSPs were introduced Fuchs et al. [14], who showed that they can be used for an efficient implementation of the so-called *painter's algorithm* [13]. Since then they have been used extensively, not only in computer graphics but also in other fields. Example applications are shadow generation [9, 10], set operations on polyhedra [19, 24], geometric data repair [16], visibility preprocessing for interactive walkthroughs [23], and cell decomposition methods in motion planning [4]. In all these applications it is important to keep the size of the BSP, that is the number of fragments generated, as small as possible. Hence, a lot of research has gone into designing BSPs of small size [6, 7, 18, 20, 21],

Computer games, simulation, animation, and other modern applications of computer graphics, are not static: the scene will contain moving objects, either following physical laws or being controlled by an external agent (which could be a software agent or a user). In such applications, a BSP must be updated over time, so that it always remains valid. This is usually achieved by time sampling: at regular time intervals the BSP is reconstructed by deleting the objects that changed position and reinserting them at the new location [11, 17, 25]. The main problem with this approach is the choice of time intervals. If the intervals are too large, important events will be missed and the BSP will be invalid for some time. If, on the other hand, the intervals are too small, a lot of computation is wasted on deleting and reinserting moving objects, even though the BSP is still valid.

An approach to overcome this problem is to apply the framework of so-called *Kinetic Data Structures*, introduced by Basch et al. [5]. A kinetic data structure (KDS) is a structure that maintains a certain 'attribute of interest' in a set of continuously moving objects—the convex hull of moving points, for instance, or the closest pair among moving objects. It consists of two parts: a combinatorial description of the attribute, and a set of *certificates*—elementary tests on the input objects—with the property that as long as the certificates remain vaild, the combinatorial structure of the attribute does not change. In other words, the set of certificates gurantees that the current combinatorial description of the attribute is still correct. It is assumed that each object follows a known flight path (which it can change at any time), so that one can compute the failure time of each certificate. The KDS therefore remains valid until the next time a certificate fails—to this end the failure times are stored in a priority queue—at which time the KDS and the set of certificates need to be updated.

The performance of a KDS is measured according to four criteria—*responsiveness*, *locality*, *compactness*, and *efficiency*—which we briefly discuss before stating our results.

[*]Part of the research by M. de Berg was done when he was visiting Stanford University. The work of J. Comba and L. Guibas was supported in part by NSF grants CCR-9623851 and CCR-9910633, and US Army MURI grant DAAH04-96-1-0007 and AASERT grant DAAG55-97-0218.

The worst-case time needed to update a KDS when a certificate fails is called its *response time*. If the response time is polylogarithmic, the KDS is called *responsive*.

A KDS is called *compact* if the number of certificates is always near-linear. It is called *local* if any object is involved in a small (polylogarithmic) number of certificates. This is important, because when an object changes its flight plan, one has to recompute the failure times of all certificates that object is involved in, and update the priority queue accordingly.

The notion of *efficiency* is slightly more complicated. A certificate failure does not automatically imply a change in the attribute being maintained; it could also be that there is only an 'internal change' in the KDS certificate set. Certificate failures where the attribute changes are called *external events*, other certificate failures are called *internal events*. The main difficulty in designing KDSs is to make sure that the worst-case total number of events (internal and external) is not much more than the worst-case number of external events. If this is the case then the KDS is called *efficient*. External events are only well defined if the attribute being maintained is unique for any given configuration of objects. This is the case for convex hulls or closest pair, but not for BSPs. However, Agarwal et al. [1] have shown that there are configurations of n moving segments, such that *any* BSPs must undergo $\Omega(n\sqrt{n})$ changes during some smooth motion.

BSPs have already been studied in a kinetic setting. For a set of n disjoint moving line segments in the plane, Agarwal et al. [3] present a kinetic BSP of expected size $O(n \log n)$ whose expected response time is $O(\log n)$. The expectation is with respect to a random order on the segments, which is fixed at the beginning. It is assumed that there is no correlation between the motion of the segments and the random order, so that the ordering 'behaves randomly' at all times. The $O(\log n)$ bound on the response time is for any given event. However, there are collections of moving segments for which there will always be *some* event that causes the kinetic BSP to spend linear time for the update. Which event takes linear time depends on the random order. So the expected *worst-case* response time over all events is not logarithmic, but linear. The structure is local—any segment is involved in $O(1)$ certificates—and the total number of events processed by their BSP is $O(n^2)$, assuming the endpoints of the segments follow trajectories of small algebraic degree.

Another paper by Agarwal et al. [2] extend this solution to set of intersecting segments, obtaining a kinetic BSP of expected size $O(n \log n + k)$, with k being the number of intersections, and with $O(\log n)$ expected response time. Again, the expected response time is not a worst-case bound over all events. They also extend the solution to 3-dimensional space.

Comba [12] presents another 3-dimensional BSP, which maintains the vertical decomposition of a set of non-intersecting triangles in 3-space, and gives a detailed description of its implementation.

In this paper we describe a new kinetic BSP, which improves in several ways over the approach of Agarwal et al. [3]. The main advantage is that the response time is $O(\log^2 n)$ in the worst case, compared to the $\Theta(n)$ worst-case response time for the previous approach. The expected response time remains $O(\log n)$. Another advantage is that the $O(n \log n)$ bound on its size is deterministic. Like the structure of Agarwal et al., ours is local and the number of events processed is $O(n^2)$.

Our kinetic BSP is a kinetic version of the deterministic BSP of Paterson and Yao [20]. As such, it based on a kinetic version of the segment tree. Our kinetic segment tree has $O(n \log n)$ size (like standard segment trees), and the response time is $O(\log n)$ in the worst case. A simple trick makes the expected response time $O(1)$, while keeping the same deterministic worst-case bounds on response time and size.

2. A KINETIC SEGMENT TREE

Let $\mathcal{S} := \{s_1, \ldots, s_n\}$ be a set of segments on the real line. The endpoints of the segments move, and segments can start or stop overlapping. The motion of the endpoints can also change the length of a segment. Our goal is to maintain a segment tree for $\mathcal{S}$. We assume the reader is familiar with segment trees [8].

The structure. The collection of nodes in the tree storing a given segment is determined by the ranks of its endpoints. So the idea is to replace the endpoints by their ranks—these can change during the motions, of course—and then maintain a segment tree on the normalized segments. (We give the smallest endpoint rank zero, not rank one. This will be convenient later.) We also need some additional structures for bookkeeping. More precisely, we maintain the following structures.

- We have a segment tree $\mathcal{T}$ whose leaves correspond to the elementary intervals $[i : i+1]$ for some integer i in the range $0..(2n-2)$. The leaf corresponding to $[i : i+1]$ is denoted by $\nu(i)$. The segment tree is augmented with parent pointers, and we maintain an additional array so that, for a given i, we have access to $\nu(i)$ in constant time.

  Each node ν of $\mathcal{T}$ stores its canonical subset in a doubly-linked list $\mathcal{S}(\nu)$. The canonical subsets are defined with respect to the (current) set of normalized segments.

- For each segment s, we have a doubly-linked list $\mathcal{L}(s)$ that links, in an ordered fashion, all occurrences of s in the various node lists $\mathcal{S}(\nu)$. We call $\mathcal{L}(s)$ the *fragment list* of s.

- We have an array $R[0..(2n-1)]$ such that $R[i]$ stores the endpoint whose current rank is i. With this information we can search the segment tree in $O(\log n)$ time at any given moment, using the current values of the endpoints instead of the ranks whenever we do a comparison.

The certificates that we have (which are stored in a priority queue, as in any KDS) are simply comparisons of the form $R[i] < R[i+1]$, for all $0 \leq i < 2n-1$.

Updating the structure. Whenever two endpoints swap, a certificate fails and we have to update the structure. Updating the array R is trivial: we just have to swap two entries. Next we discuss how to update the segment tree itself; updating the fragment lists is easily done while the segment tree is being updated.

Swapping the endpoints of two segments only influences where these two segments are stored: the ranks of the endpoints of all other segments remain unchanged, so they will be stored at exactly the same nodes as before. Let s be a segment involved in the swap. Then the rank of one of its endpoints either increases by one, or it decreases by one: if before the swap $s = [i : j]$, then after the swap we have $s = [i-1 : j]$, or $s = [i+1 : j]$, or $s = [i : j-1]$, or $s = [i : j+1]$. We could update the segment tree in $O(\log n)$ time by simply deleting the segment $[i : j]$ and re-inserting the new segment. Such an approach will not work when we use the segment tree as the basis for a BSP, however, so next we describe an alternative approach. It has the additional advantage that a simple trick allows us to obtain $O(1)$ expected update time, while keeping the worst-case update time $O(\log n)$.

The change to s is always an addition or subtraction of an elementary interval.

To add an elementary interval $[j : j+1]$ to the segment s, we go to $\nu(j)$, the leaf representing this elementary interval. We then start the following procedure. Suppose we are at a node ν; initially ν will be $\nu(j)$. If ν is the root of the tree, we append s to $\mathcal{S}(\nu)$ and we are done. Otherwise we check whether $s \in \mathcal{S}(\mu)$, where μ is the sibling of ν. Using the fragment list $\mathcal{L}(s)$, this can be done in $O(1)$ time. If so, we delete s from $\mathcal{S}(\mu)$, we set $\nu := \text{parent}(\nu)$, and repeat. If $s \notin \mathcal{S}(\mu)$, we append s to $\mathcal{S}(\nu)$ and we are done.

Deleting an elementary interval $[j : j+1]$ from s is done in the reverse manner. Let ν be the node storing s such that $[j : j+1]$ is contained in the interval corresponding to ν. The node ν must be the first or last node in $\mathcal{L}(s)$, so we can find it in $O(1)$ time. We delete s from the list $\mathcal{S}(\nu)$. If ν is a leaf, we are done. Otherwise, we append s to the lists of the two children of ν. For one of the children, $[j : j+1]$ will be contained in the corresponding interval; we recursively delete $[j : j+1]$ from s at this child.

The time we spend at each node we visit in the recursive update procedures is $O(1)$. Hence, we get the following result.

Lemma 2.1. *Adding an elementary interval $[j : j+1]$ to (or subtracting it from) a segment s stored in the segment tree can be done in $O(h)$ time, where h is the height of the node storing s whose corresponding interval contains $[j : j+1]$ after the addition (or before the subtraction).*

Lemma 2.1 implies that the response time of our kinetic segment tree is $O(\log n)$. Next we describe a simple trick to reduce the expected response time to $O(1)$.

Suppose we have to add the elementary interval $[j : j+1]$ to a segment s. Lemma 2.1 states that this take $O(h)$ time, where h is the height of the node ν^* storing s whose corresponding interval contains $[j : j+1]$ after the addition. We can bound h as follows. Consider the leaf $\nu(j)$ corresponding to $[j : j+1]$. Then $\nu(j)$ must be the leftmost leaf in the subtree rooted at ν^* (in case $[j : j+1]$ extends s to the left) or it must be the rightmost leaf in this subtree (in case the extension is to the right). Define $h(j)$ to be the height of the highest node ν in $\mathcal{T}$ such that $\nu(j)$ is either the leftmost leaf or the rightmost leaf in the subtree rooted at ν. Clearly $h(j)$ is an upper bound on the height h of ν^*. The following lemma is easy to prove.

Lemma 2.2. *The average value of $h(j)$ over all leaves $\nu(j)$, $0 \leq j < 2n-1$, is $O(1)$.*

In other words, the expected value of $h(j)$ is $O(1)$ if we choose j randomly. So what we do is choose a random integer r with $0 \leq r < 2n$, add it to all ranks (modulo $2n+1$), and then work with these shifted segments. This means that the expected response time for any given event is $O(1)$. (It can happen that if we apply this operation to a segment $[x : x']$ we get that $(x+r) \bmod (2n+1) > (x'+r) \bmod (2n+1)$. In this case we split the segment into two subsegments, namely $[(x+r) \bmod (2n+1) : 2n]$ and $[0 : (x'+r) \bmod (2n+1)]$. The algorithms are easily adapted to this. The fact that there may be many subsegments starting at zero is no problem for the analysis, because an endpoint of a subsegment that this is not an endpoint of the corresponding original segment does not move. An alternative approach would be to use a segment tree on the universe $0..4n$.)

Observe that every endpoint is involved in at most two certificates, so the structure is local. It is also efficient since there are no internal events; we only update the structure when two endpoints swap, which necessarily implies a change in the segment tree. We get the following result.

Theorem 2.1. *Let $\mathcal{S}$ be a set of n segments on the real line whose endpoints are moving. We can maintain a kinetic segment tree for $\mathcal{S}$ that uses the same amount of storage as an ordinary segment tree, namely $O(n \log n)$, and is local and efficient. The expected response time—that is, the expected time needed to update the segment tree when two endpoints swap—is $O(1)$, while the worst-case response time is $O(\log n)$.*

3. A KINETIC BSP

In this section we show how to kinetize the deterministic BSP described by Paterson and Yao [20]. (In fact, the structure dates back to Preparata [22]; his trapezoid method for point location is exactly the same as the BSP of Paterson and Yao.) The resulting structure will be responsive, efficient, local, and compact. In our BSP we will continue the splitting process until each subspace is empty—we do not stop when there is only one object left. This modification is possible because the objects we consider are 1-dimensional (that is, line segments), and it simplifies the description slightly.

We start by explaining the basic ideas behind the structure, which is most easily accomplished by viewing the BSP as a multi-way tree. Then we show how to transform the tree into a binary tree and we fill in the details of our approach.

3.1 The global approach

Let $\mathcal{S}$ be a set of segments in the plane. Before we describe how the kinetic BSP is constructed, we introduce some terminology. Any node ν in a BSP tree has an associated subspace, which is the intersection of the half-planes defined by nodes on the path from ν up to the root. We assume that the root has an associated subspace that is a bounding box for the entire scene. Because of the way we construct our BSP, the subspace of a node ν will always be a quadrilateral $\Delta(\nu)$ with two vertical sides. The two vertical sides are contained in splitting lines through segment endpoints. The top and bottom sides are contained in segments. Let $\mathcal{S}(\nu) := \mathcal{S} \cap \Delta(\nu)$ be the set of fragments of segments that have to be stored in the subtree rooted at ν. A segment in $\mathcal{S}(\nu)$ that intersects both the left and right boundary

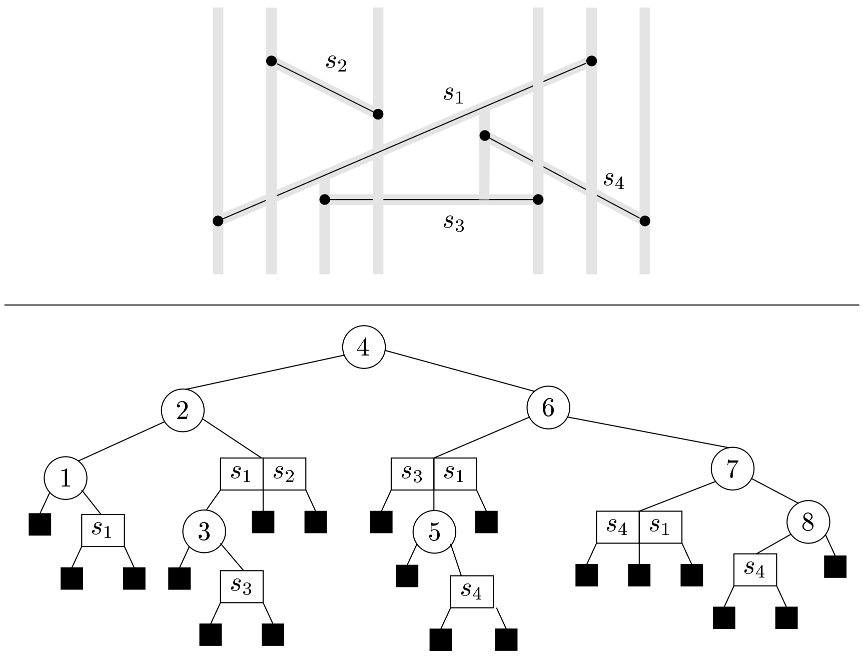

Figure 1: Example of the BSP; vertical nodes are depicted as circles, free nodes as rectangles, and leaves as solid squares.

of $\Delta(\nu)$ is called *long.* As in the kinetic segment tree, we will be working with the ranks of the x-coordinates of the endpoints. Let $x_\ell(\nu)$ and $x_r(\nu)$ denote the rank of the x-coordinate of the left and right side, respectively, of $\Delta(\nu)$; this is well defined since the x-coordinate of a vertical splitting line is always the x-coordinate of some endpoint. Finally, we use $\ell(x)$ to denote the vertical splitting line through the endpoint with rank x, and $\ell(s)$ to denote the splitting line through a segment s.

The construction

The recursive construction of the BSP at node ν is done as follows.

- If $\mathcal{S}(\nu)$ is empty, then ν is a leaf.

- If there are no long segments in $\mathcal{S}(\nu)$, then ν stores one splitting line, namely the vertical line $\ell(x_{\text{mid}})$, where $x_{\text{mid}} = \lfloor (x_\ell(\nu) + x_r(\nu))/2 \rfloor$. We call this a *vertical split*, and we call ν a *vertical node.*

  The left (right) child of ν is the root of a recursively defined BSP for the parts of the segments in $\mathcal{S}(\nu)$ lying to the left (right) of the splitting line.

- If there are long segments in $\mathcal{S}(\nu)$, then ν is a multi-way node. The splitting lines it stores are the lines through the long segments, ordered from bottom to top. Since the parts of these lines within $\Delta(\nu)$ are contained in their corresponding segments, these splitting lines do not cause any fragmentation. Hence, we call them *free splits* and we call ν a *free node.*

  The number of children of ν is one plus the number of free splits made at ν. The i-th child is the root of a recursively defined structure for the parts of the segments lying between the $(i-1)$-st free split and the i-th free split.

Figure 1 shows an example of a BSP constructed with this method.

One subtle but important difference between our BSP and the BSP as described by Paterson and Yao is how a vertical split is performed: we take a line through the median of the x-coordinates of *all* endpoints whose x-coordinate lies in $[x_\ell(\nu) : x_r(\nu)]$, whereas Paterson and Yao only consider endpoints in $\Delta(\nu)$. This change turns out to be crucial when we want to kinetize the structure. In our case a vertical split can be useless in the sense that all fragments in $\mathcal{S}(\nu)$ lie completely to the same side of the splitting line. In this case we could ignore the splitting line—that is, not store it. We will nevertheless leave it in, because then we will have to worry about less special cases. This does not influence the asymptotic size of the BSP.

To be able to maintain the structure efficiently when the segments move, we need to store the same extra information as in the segment-tree case. Later we will see that in fact we need some more additional information.

- There is an array $R[1..2n]$, such that $R[i]$ stores the endpoint whose x-coordinate has rank i.

- For each segment s, we have a *fragment list* $\mathcal{L}(s)$ that links all the nodes where s is stored, ordered from left to right.

The following lemma summarizes the properties of our structure. The proof of the lemma is rather straightforward—it is basically the same as the proof of Paterson and Yao on the size of their BSP—and therefore omitted.

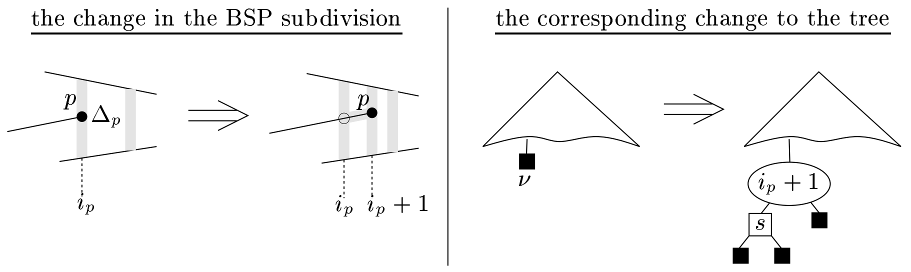
the change in the BSP subdivision
the corresponding change to the tree
p
Δ_p
i_p
p
i_p
i_p+1
ν
i_p+1
s

Figure 2: Updating the BSP at leaf level.

LEMMA 3.1. *The BSP described above together with the additional information uses $O(n \log n)$ storage, and its height is $O(\log n)$. Any segment is stored at $O(\log n)$ nodes.*

Updating the BSP

Since the segments remain disjoint during the motion, the only time the BSP changes is when two endpoints change their x-order. This means, of course, that we have to update the array R. This can easily be done in $O(1)$ time. Next we describe how to update the BSP itself; updating the fragments lists of the two segments involved in the update requires only simple pointer manipulations, so we omit the details. If there is only one segment involved—the event is a segment becoming vertical—we need not perform any actions besides updating R, so in the remainder we assume two segments are involved.

When two endpoints change x-order, the rank of one of them increases by one while the rank of the other decreases by one. Suppose the rank of the right endpoint p of a segment s increases by one. This essentially means that the segment is 'extended' to the right by adding an *elementary fragment*—a fragment whose normalized length is one—to it. Of course the reverse (a segment is shortened by subtracting an elementary fragment) can happen as well. We first discuss the addition of an elementary fragment.

Adding an elementary fragment. We assume the elementary fragment has to be added to the right of a segment s; adding an elementary fragment to the left of a segment is done in a similar manner. The addition of an elementary fragment has two steps.

Step 1 [The update at leaf level]. Let Δ_p be the cell in the BSP subdivision immediately to the right of p, and let ν be the leaf corresponding to Δ_p; thus $\Delta_p = \Delta(\nu)$. Let i_p be the rank of p before the addition of the elementary fragment. Note that $i_p = x_\ell(\nu)$. There are two cases.

Case (i): $x_r(\nu) > i_p + 1$. We need to split Δ_p into three subcells, by first applying a vertical split through p (with its new rank i_p+1) and then applying a free split in the new subcell to the left of p. Changing the BSP tree correspondingly is easily done in $O(1)$ time—see Figure 2.

Case (ii): $x_r(\nu) = i_p + 1$. We need to split Δ_p into two subcells, by applying a free split along s. Again it is easy to change the BSP tree correspondingly. In this case we also check whether the parent of the node ν' storing the new elementary fragment is a free node. If so, we have to merge ν' with its parent: we have to delete ν' and insert the elementary fragment at the correct position into the collection of fragments already stored at the parent.

Step 2 [Restoring the segment-tree property]. After Step 1 we have again a valid BSP: the subtree of every node ν stores exactly the fragments contained in the region $\Delta(\nu)$. However, the segment-tree property—a segment is used for a free split as soon as it becomes long—might be violated for the segment s. We restore this property with the following recursive procedure. Suppose we have just stored the segment s at a node ν; initially ν is the free node created in Step 1. We have to check whether we have to merge the fragment $f := s \cap \Delta(\nu)$ with its neighboring fragment. This has to be done if the neighboring fragment f'—there is only one, because f contains the endpoint p—is stored at a sibling of ν. Using the fragment list $\mathcal{L}(s)$ we can check this in $O(1)$ time. Merging f and f' amounts to some cut and link operations to the tree, as described next and illustrated in Fig. 3. Let μ be the parent of ν; μ must be a vertical node. Since f has to be merged with f', we have to apply the free split along $f \cup f'$ before applying the vertical split at μ. So we replace μ by three nodes: a free node μ' for $f' \cup f$, and two vertical nodes. The two vertical nodes use the same splitting value as μ used to have—rank i in Fig. 3. This makes it easy to assemble their subtrees from the subtrees we had before the merge: for each of the old subtrees depicted in Fig. 3, we know its position relative to $f' \cup f$ and its position relative to the vertical splitting line of μ, so a few pointer manipulations restore the order in the tree. (It can happen that one of the new vertical nodes does not have a vertex in its subspace. This happens if and only if both its children are leaves. In that case we replace the node and its children with a single leaf.) If the parent of μ' is a free node, we have have to merge μ' with its parent, that is, insert $f' \cup f$ into this (multi-way) node. We set ν to be the free node now storing $f' \cup f$, and repeat.

The process ends when we discover that the fragment of the current node ν does not need to be merged, either because ν's sibling does not store s or because ν is the root (and therefore has no sibling).

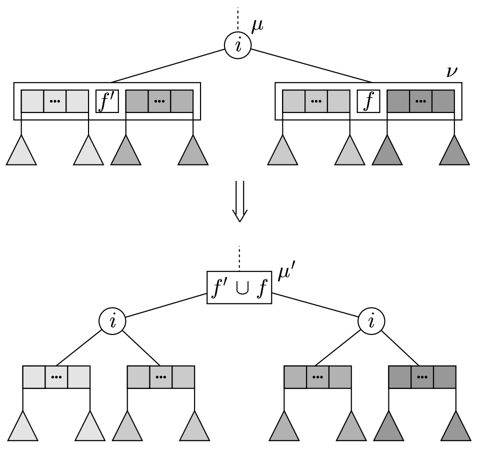

Figure 3: Merging two fragments.

Subtracting an elementary fragment. Subtracting an elementary fragment from a segment s is done in the reverse manner, namely as follows. Let ν be the node storing the fragment f of s containing the elementary fragment. This node can be found in $O(1)$ time using the fragment list $\mathcal{L}(s)$.

If f is exactly the elementary fragment, then we are just above leaf level. In this case we apply the reverse of Step 1 of the addition procedure.

Otherwise, we apply the reverse of the merging operation described in Step 2 of the addition procedure. The fragment f is thus split into two sub-fragments that are stored two levels down. We recursively subtract the elementary fragment from the relevant sub-fragment.

3.2 The details

In the global description above, the free nodes where multi-way nodes. This is not permitted in a BSP. Below we show how to transform the BSP into a binary tree. We also discuss the extra information needed for an efficient implementation of the update procedures, and we analyze the time complexity of the updates.

Transforming the BSP into a binary tree. Consider a free node ν in the BSP. Following Paterson and Yao, we replace the multi-way node ν by a binary tree with one node per free split made at ν. Paterson and Yao use a weight-balancing scheme for this, where the weights correspond to the number of vertices in the subtrees of ν. This way they are able to keep the depth of the tree $O(\log n)$. Unfortunately we have not been able to apply this in our setting. (One problem is that we make vertical splits not at the median x-coordinate of the endpoints in a trapezoid $\Delta(\nu)$, but at the median x-coordinate of all endpoints in the vertical slab $[x_\ell(\nu) : x_r(\nu)] \times [-\infty : \infty]$.) Therefore we use an ordinary balanced tree, in particular a red-black tree [15]. The depth of our BSP will then be $O(\log^2 n)$.

Which operations do we need to perform on the free nodes and, hence, on the trees we replace them with? By inspecting the update algorithms described in the previous subsection, we see that we need to be able to insert and delete free splits into a free node, and we need to be able to split and concatenate free nodes—see Figure 3. These operations can be performed in $O(\log n)$ time on a red-black tree. There is one more operation that was more implicit in the description using multi-way nodes: we needed to go from a free node to its parent. This means we have to augment the red-black tree with parent pointers, so we can walk from a given node in it (corresponding to a free split) back to the root of the tree. In fact, we equip the entire BSP with parent pointers for this purpose. Stepping from a free multi-way node to its parent thus takes $O(\log n)$ time in the red-black tree representation.

We conclude that the operations we perform on a free node in the update algorithms can be implemented in $O(\log n)$ time, using a red-black tree representation.

Some additional information. We maintain the following extra information.

- For each right (left) endpoint p, we have a pointer to the leaf corresponding to the trapezoid Δ_p immediately to the right (left) of p.

- For each leaf ν, we have pointers to the leaves corresponding to all neighboring trapezoids separated from $\Delta(\nu)$ by a vertical splitting line. Under our general position assumptions, there are at most four such neighbors.

The trapezoid Δ_p is needed to perform in $O(1)$ time Step 1 of the procedure to add an elementary fragment to a segment. The neighboring trapezoids are needed to maintain Δ_p.

The analysis. From the discussion above we can conclude that the update time is $O(\log n)$ times the number of vertical nodes visited by the update algorithm. The latter number equals the number of nodes we would visit when we would update a kinetic segment tree on the x-projections of the segments. This number is obviously $O(\log n)$, but we have seen in Section 2 that a simple trick—adding a random integer between zero and $2n-1$ to all ranks—makes the expected number of visited nodes $O(1)$. This trick immediately carries over to the BSP. We get the following result.

THEOREM 3.1. *Let S be a set of n moving segments in the plane that are disjoint at all times. There is a kinetic BSP of worst-case size $O(n \log n)$ and worst-case depth $O(\log^2 n)$ that only changes when two endpoints change their x-order. The update for such an event takes $O(\log n)$ expected time and $O(\log^2 n)$ time in the worst case.*

4. CONCLUDING REMARKS

We have presented a kinetic version of a segment tree, and shown that the cost for maintaining it when two endpoints swap is $O(1)$ in the expected case, and $O(\log n)$ in the worst case. Based on this, we developed a kinetic version of the deterministic BSP of Paterson and Yao [20]. The worst-case size of our BSP is $O(n \log n)$, and the response time is $O(\log^2 n)$ in the worst case and $O(\log n)$ expected. This improves a result by Agarwal et al. [3] who describe a structure with $O(n \log n)$ expected size, $O(\log n)$ expected response time, and $\Theta(n)$ worst-case response time.

Although we believe our result to be an improvement, it has some disadvantages as well. One is that we have to process all swaps of x-coordinates of endpoints, whereas Agarwal et al. process only a subset. So their structure may process fewer events, even though the worst-case number of events is asymptotically the same. Another disadvantage of our method is that the resulting BSP may have $O(\log^2 n)$ depth, whereas the depth of the BSP of Agarwal et al. is $O(\log n)$. (But note that our bound is deterministic while theirs is expected.) It would be interesting to find a BSP that combines the advantages of both approaches.

5. REFERENCES

[1] P.K. Agarwal, J. Basch, M. de Berg, L.J. Guibas, and J. Hershberger. Lower bounds for kinetic planar subdivisions. *Discrete Comput. Geom.*, To appear.

[2] P.K. Agarwal, J. Erickson, and L.J. Guibas. Kinetic binary space partitions for triangles. In *Proc. 9th ACM-SIAM Sympos. Discrete Algorithms*, pages 107–116, 1998.

[3] P.K. Agarwal, L.J. Guibas, T.M. Murali, and J.S. Vitter. Cylindrical static and kinetic binary space partitions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 39–48, 1997.

[4] C. Ballieux. Motion planning using binary space partitions. Technical Report Inf/src/93-25, Utrecht University, 1993.

[5] J. Basch, Leonidas J. Guibas, and J. Hershberger. Data structures for mobile data. In *Proc. 8th ACM-SIAM Sympos. Discrete Algorithms*, pages 747–756, 1997.

[6] M. de Berg. Linear size binary space partitions for uncluttered scenes. *Algorithmica*, To appear.

[7] M. de Berg, M. de Groot, and M. Overmars. New results on binary space partitions in the plane. *Comput. Geom. Theory Appl.*, 8:317–333, 1997.

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

[9] N. Chin and S. Feiner. Near real-time shadow generation using BSP trees. In *Proc. SIGGRAPH '89*, pages 99–106, New York, August 1989. ACM SIGGRAPH.

[10] N. Chin and S. Feiner. Fast object-precision shadow generation for areal light sources using BSP trees. *Comput. Graph.*, 25:21–30, March 1992. Proc. 1992 Sympos. Interactive 3D Graphics.

[11] Y. Chrysanthou. *Shadow Computation for 3D Interaction and Animation.* Ph.D. thesis, Queen Mary and Westfield College, University of London, 1996.

[12] J.L.D. Comba. *Kinetic Vertical Decomposition Trees.* Ph.D. thesis, Department of Computer Science, Stanford University, 2000.

[13] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics: Principles and Practice.* Addison-Wesley, Reading, MA, 1990.

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

[15] L. J. Guibas and R. Sedgewick. A dichromatic framework for balanced trees. In *Proc. 19th Annu. IEEE Sympos. Found. Comput. Sci.*, Lecture Notes Comput. Sci., pages 8–21. Springer-Verlag, 1978.

[16] T. M. Murali and T. A. Funkhouser. Consistent solid and boundary representations from arbitrary polygonal data. In *Proc. 1997 Sympos. Interactive 3D Graphics*, 1997.

[17] B. Naylor. Interactive solid geometry via partitioning trees. In *Proc. Graphics Interface '92*, pages 11–18, 1992.

[18] B. Naylor. Constructing good partitioning trees. In *Proc. Graphics Interface '93*, pages 181–191, 1993.

[19] B. Naylor, J. A. Amanatides, and W. Thibault. Merging BSP trees yields polyhedral set operations. *Comput. Graph.*, 24(4):115–124, August 1990. Proc. SIGGRAPH '90.

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

[21] M. S. Paterson and F. F. Yao. Optimal binary space partitions for orthogonal objects. *J. Algorithms*, 13:99–113, 1992.

[22] F. P. Preparata. A new approach to planar point location. *SIAM J. Comput.*, 10:473–482, 1981.

[23] S. J. Teller and C. H. Séquin. Visibility preprocessing for interactive walkthroughs. *Comput. Graph.*, 25(4):61–69, July 1991. Proc. SIGGRAPH '91.

[24] W. C. Thibault and B. F. Naylor. Set operations on polyhedra using binary space partitioning trees. *Comput. Graph.*, 21(4):153–162, 1987. Proc. SIGGRAPH '87.

[25] E. Torres. Optimization of the binary space partition algorithm (BSP) for the visualization of dynamic scenes. In *Eurographics '90*, pages 507–518. North-Holland, 1990.