Efficient Evaluation of the Field Functions of Soft Objects Using Interval Tree

Kyung-Ha Min[†], In-Kwon Lee[‡] and Chan-Mo Park[†]

[†]Department of Computer Science and Engineering
Pohang University of Science and Technology

[‡]POSTECH Information Research Laboratories
Pohang University of Science and Technology

Abstract

We present an algorithm to evaluate the field function of a soft object efficiently. Instead of using a global field function that is defined by the sum of all local field functions, we consider only the set of local field functions that affects a point at which we want to evaluate the field function. To find the affecting local field functions efficiently, we exploit a data structure called interval tree based on the bounding volume of the component corresponding to the primitives (skeletons) of a soft object. The bounding volume of each component is generated with respect to the radius of a local field function of the component, threshold value, and the relations between the components and other neighboring components. The proposed scheme of field function evaluation can be used in many applications for soft objects such as modeling and rendering, especially in interactive modeling process.

1. Introduction

A *soft object*, also known as a *blobby object* or *metaball*, is a kind of implicit surface that is a widely used model to represent smooth geometric objects [3, 15]. A soft object is defined by following parameters:

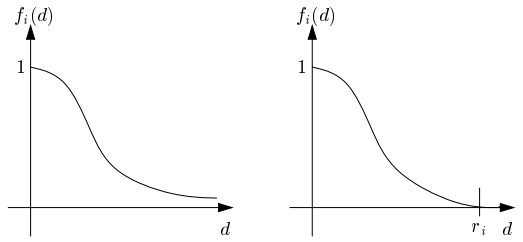
- *Primitives*:
  A primitive (also known as skeleton) p_i is usually a simple geometric object such as point and line segment. The set of the primitives determines the approximate shape of the soft object. In this paper, we mainly consider the convex primitives such as points, line segments, and triangles. The extension of our approach to any general type of geometric primitive is straightforward: for example, a concave polygon can be subdivided into a set of convex polygons, and a curve can be approximated by a piecewise linear curve.
- *Field function*:
  For each primitive p_i, a *local field function* f_i is given, which maps a point in R^3 into a value in [0..1]. Let p_i be a point primitive. The value of the local field function f_i at an arbitrary point $v = (x, y, z)$ is computed by $f_i(d)$, $d \geq 0$, where d is a Euclidean distance between v and p_i. Usually, $f_i(d)$ is a smoothly decreasing function (see Figure 1). The sum of all local field functions included in a soft object, $f = \sum f_i$, is called a *global field function* (simply *field function*) of the soft object.
- *Threshold*:
  A global constant T is given as a threshold value of a soft object. The surface of a soft object consisting of n primitives is defined by the set of points v satisfying

$$f(v) = \sum_{i=1}^{n} f_i(d_i) = T, \tag{1}$$

  where d_i is a distance between v and p_i.
- *Radius*:
  The local field function in Figure 1(a) has a positive value for infinite domain ($d \geq 0$), which can influences a point located very far from a primitive. To reduce the computational cost, a local field function like a Figure 1(b) is suggested, by which the influence of the local field function can be restricted to the domain, $0 \leq d < r$.

We denote a three-tuple $s_i = (p_i, f_i, r_i)$ a *component* of a soft object. A soft object, s, is specified by a set of n components $\{s_i = (p_i, f_i, r_i)\}$, $i = 1, \ldots, n$, and a threshold value T.

Many algorithms such as polygonization and ray tracing of a soft object include a number of evaluations of the field



(a) Infinite field function (b) Finite field function

Figure 1: *Field functions with (a) infinite domain, and (b) finite domain*

function of the soft object. Thus, speeding up the evaluation of the field function of a soft object has been considered as an important problem [15]. Let us assume that we want to evaluate the global field function $f(v)$ at an arbitrary point v. When we evaluate $f(v)$ using Equation (1), we do not have to consider all local field functions f_i, since $f_i(d_i)$ vanishes, if d_i is greater than the radius r_i. Thus, finding a set of primitives (called *influencing primitives*), whose distances from a given point v are less than the corresponding radii of the primitives, is a key problem of the fast evaluation of a field function.

In this paper, we suggest an efficient method to find the primitives that influence a given point using *interval tree*. The interval tree is a kind of binary search tree whose internal nodes store boundary values of the bounding volumes of the components and leaf nodes store a list of primitives that influence the corresponding interval. The bounding volume of a component is computed by considering the amount of the offset from the primitive in the component. The offset is determined to build compact bounding volumes in terms of the radius of the component and the relations between the neighboring components and the considering component. If the distance between a point and a primitive is less than the offset, then the point lies inside the bounding volume of the corresponding component of the primitive, while the inverse does not hold. Using the interval tree, the procedure of finding the influencing primitives for a point can be implemented by simple one dimensional searches in the interval tree. All distance computations are transferred to the preprocessing step that computes an interval tree, and thus, we do not have to compute the distance between a point and a skeleton in the process of the evaluation of a field function.

By using the interval tree, we can reduce the execution time of many applications such as polygonization and rendering of soft objects. In Section 6, we demonstrate the effect of our method by applying the method to the polygonization of soft objects. By the comparisons among our method, the simple evaluation method without any data structure such as interval tree, and other methods [15] for speeding up the evaluation of the field function, the efficiency of our method is proved.

This paper is organized as follows. In Section 2, the researches on the field function of a soft object, the methods of the efficient evaluation of field function, and the bounding volumes in collision detection and ray tracing are reviewed. The algorithm for computing the bounding volume of a component is proposed in Section 3 and the algorithm for building and traversing an interval tree is illustrated in Section 4. In Section 6, implementation details and performance comparison with the conventional method for field function evaluation are provided. Finally, we conclude the paper and suggest some future works in Section 7.

2. Previous Work

In 1982, Blinn proposed a soft object, a new modeling paradigm using distribution function [3]. The distribution function, known as field function or potential function, has played a major role in modeling and rendering models represented using soft objects. The first field function of a soft object, proposed by Blinn [3], is defined as follows:

$$f(d) = \frac{1}{2}\exp(\alpha - 4\alpha d^2),$$

where d is the distance between a vertex and a primitive and α is a hardness factor that controls the slope of the function at $d = 0.5$. By this function, a vertex located very far from a primitive is still influenced by the primitive.

The inefficiency of the Blinn's distribution function of infinite domain was improved by the field function having finite domain, proposed by Nishimura et al. [13]. Nishimura's field function is defined by

$$f(d) = (d < 1)\ ?\ (d < \frac{1}{3})\ ?\ \frac{4}{3} - 4d^2\ :\ 2(1-d)^2\ :\ 0,$$

where d is a normalized distance by the radius of the primitive. Using this function, the function evaluations for the vertices whose normalized distance is greater than 1 are avoided. Note that the function is C^2 continuous everywhere.

Nishimura's function, however, possess inefficiency by computing square roots in its formula. Improved field functions in polygonal formula are proposed by Wyvill et al. [15] and Murakama [12]. These finite field functions, however, do not include the hardness factor that controls the slop of the function. Gascuel [5], Kacic-Alesic and Wyvill [8], and Blanc and Schlick [2] proposed finite field functions with the hardness factor.

Wyvill et al. [15] also presented some techniques with data structures for the fast evaluation of a field function. In their method, the volume of space of the whole scene is divided into cubes of fixed size. Each of these cubes is considered as an entry of a hash table, and the entry has a list of pointers to the primitives that influence the cube. Since the method decomposes a space, the method can be considered as one of

image-space approaches. When a field function is evaluated at a given point p, the cube containing p can be found easily from the coordinates of p (they explained this in terms of hashing), and thus, we can easily access to the primitives that influence p.

Nevertheless, Wyvill et al.'s method has some disadvantages. First, it is not easy to determine the optimal size of a cube. Especially, when a number of components lie on a very small region, the components form a cluster in the hash table. Consequently, the method cannot effectively solve the problem of the fast evaluation of field function [15]. If the size of a cube is too small, we need much amount of storages and the construction of the hash table structure takes long time. Second, the construction of the hash table structure is not easy especially for the primitives that are not points. For example, it is not easy to check exactly whether a line segment primitive influence a cube or not. The exact influence field of a line segment primitive may be the volume of a cylinder with arbitrary orientation. The intersection check between the cylinder and the cube may be cumbersome in the construction process. Thus, a simple approximation (such as axis-parallel hexahedron or sphere) of the exact influence field is often used. Third, Wyvill et al. did not consider the dynamic modification of the data structure. The insertion of a component is straightforward. However, if a component needs to be deleted from a soft object, the whole hash table must be scanned in the worst case.

Being compared with the Wyvill et al.'s method, our method using the interval tree is an *object-space* approach. Instead of decomposing the space, we build an interval tree based on the bounding volume of each component. This object-space based approach guarantees that the method can effectively handle the situation where many components form a cluster. In our method, the bounding volume can be defined according to more than 3 principle axes. Thus, the bounding volumes can be more exact approximations of the exact influencing fields. Furthermore, we seriously consider the dynamic update of tree interval tree to support the interactive modeling process. The insertion/deletion of a component into/from an interval tree can be easily implemented using simple algorithms for the binary search tree. The balancing based on AVL tree always maintain the interval tree to have an optimal height without any serious computation.

Researchers on collision detection and ray tracing have been traced a slightly different problem:

"Given two sets of objects, find intersecting objects without testing all pairs of the objects."

Among the various methods proposed to solve the problem, a bounding volume hierarchy tree [1, 7, 6] is known to provide an efficient solution. A bounding volume hierarchy tree is a tree whose leaf node stores an individual object and its bounding volume. A parent node of the tree stores a bounding volume of the bounding volumes of its child nodes. The bounding volume hierarchy tree can be efficiently used specially in interactive applications to detect the collisions among the moving objects. The hierarchy of the tree can be used the level of collision detection with respect to the speed of the execution of the application.

3. Computing Bounding Volumes of an Object

3.1. Backgrounds

Various schemes for computing bounding volumes of geometric objects have been proposed in collision detection and ray tracing [1, 7, 6]. The proposed bounding volumes are classified according to the complexity of the bounding volume geometry. The bounding volumes defined by some simple geometries, such as bounding sphere or axis-aligned bounding box, provide easy computation and fast comparison, while they are bad approximations by leaving large empty corners. On the contrary, bounding volumes using complex geometries, such as oriented bounding box or convex hull are tighter approximations than the simple ones, while they show poor performance in construction and checking intersections. To provide the benefits of the both types of the schemes, a hybrid approach, known as bounding slab [9], finite direction hull [16], or k-DOPs [10], have been proposed. The methods of this scheme computes tighter approximations than the simple ones and provide more efficient performance than the complex ones. In this paper, we build the bounding volumes of the components of a soft object based on this hybrid approach.

3.2. Definition of the bounding volume

For a component $s_i = (p_i, f_i, r_i)$ of a soft object, *k bounding directions* are defined to compute the bounding volume of the component. In this paper, we use $k = 3$ or $k = 7$ for bounding directions. When $k = 3$, the three directions are $(1,0,0)$, $(0,1,0)$, and $(0,0,1)$. When $k = 7$, the seven directions are $(1,0,0)$, $(0,1,0)$, $(0,0,1)$, $(1,1,1)$, $(1,-1,1)$, $(-1,-1,1)$, and $(-1,1,1)$. In the case of $k = 3$, the resulting bounding volume coinsides with the axis-aligned bounding box. For each bounding direction $\vec{d}_j$, the two planes P^i_{j+} and P^i_{j-} that are orthogonal to the direction and circumscribed out the bounding volume are defined by

$$\begin{aligned} P^i_{j+} &= \{x \mid \vec{d}_j \cdot x = \tilde{v}_{max} \cdot \vec{d}_j + \delta^i_{j+}\}, \\ {}^i_{j-} &= \{x \mid \vec{d}_j \cdot x = \tilde{v}_{min} \cdot \vec{d}_j - \delta^i_{j-}\}, \end{aligned}$$

where $\tilde{v}_{max}$ and $\tilde{v}_{min}$ are vertices on p_i (the primitive of s_i) whose inner product with $\vec{d}_j$ is maximum and minimum, respectively. Figure 2 shows an example of the computation of P^i_{j+} and P^i_{j-} for various primitives with the definitions of $\tilde{v}_{min}$ and $\tilde{v}_{max}$.

δ^i_{j+} and δ^i_{j-} are offsets that indicate the range of influence

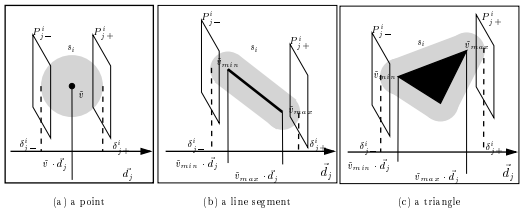


Figure 2: *The computation of P^i_{j+} and P^i_{j-}*

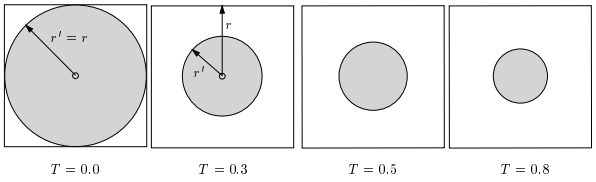


Figure 3: *Comparison of r'_r at various T*

of s_i to the direction of $\vec{d}_{j+}$ and $\vec{d}_{j-}$, respectively (see next subsection for the definition of offset). Consequently, the intersection of two halfspaces that enclose the object, denoted as h^i_j, is defined by

$$h^i_j = \{x \mid \tilde{v}_{min} \cdot \vec{d}_j - \delta^i_{j-} \leq \vec{d}_j \cdot x \leq \tilde{v}_{max} \cdot \vec{d}_j + \delta^i_{j+}\}.$$

Two values $b^i_{j-} = \tilde{v}_{min} \cdot \vec{d}_j - \delta^i_{j-}$ and $b^i_{j+} = \tilde{v}_{max} \cdot \vec{d}_j + \delta^i_{j+}$ are called the *bounding values* of h^i_j. Therefore, a bounding volume of the primitive p_i, denoted as H^i, using k bounding directions are defined as follows:

$$H^i = \bigcap_{j=1}^{k} h^i_j.$$

Suppose we test the intersection between the bounding volumes of two components, s_n and s_m. The corresponding bounding volumes, H^n and H^m are intersected, if h^n_j and h^m_j, are intersected, for all $j = 1, 2, \ldots, k$. Note that the intersection between h^n_j and h^m_j can be tested by comparing the boundary values.

3.3. Calculation of the offset

δ^i_{j+} and δ^i_{j-} are offsets of a component s_i in direction of $\vec{d}_j$ and $-\vec{d}_j$, respectively. Basically, since we define the bounding volume to find influencing primitives, the offset is to be defined as r_i, the radius of s_i. In many cases, however, the actual range of influence of s_i becomes $r'_i = f_i^{-1}(T)$, which is the actual radius of s_i. Note that $f_i(r'_i) = T$ is a boundary of influence. Figure 3 compares r'_i's for various T. Note that r'_i coincides with r_i at $T = 0$.

To build a compact bounding volume of an object, we compute the offset according to the actual range of influence of the object. Since the actual range of influence varies

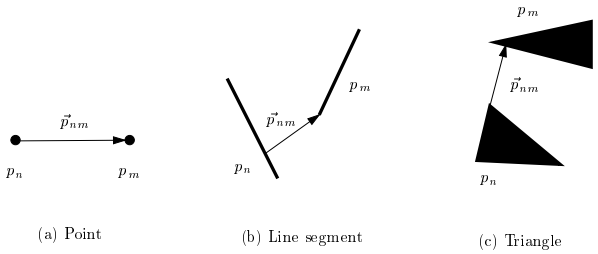


Figure 4: *The vector $\vec{p}_{nm}$ of the shortest distance between p_n and p_m for various types of primitives*

according to the threshold value and the relations between neighboring objects, we define the relation between two soft objects before defining the offset. Let $\vec{p}_{nm}$ denote a vector emanating from a point on a primitive p_n to a point on p_m, where $\|\vec{p}_{nm}\|$, the magnitude of $\vec{p}_{nm}$, is the shortest distance between p_n and p_m. Since we assume that the primitives are convex, at least one of the two end points of $\vec{p}_{nm}$ is a vertex of a primitive (see Figure 4). The relation between two components s_n and s_m can be classified according to $\vec{p}_{nm}$, r_n and r'_m as follows:

1. s_n is *related* to s_m, if $\|\vec{p}_{nm}\| \leq r_n + r'_m$.
2. s_n and s_m are *disjoint*, otherwise.

Figure 5 shows the classification of types according to the radii of the components.

Initially, the offsets in all bounding directions of a component s_n are set to the actual radius r'_n of the object. Then, all components are tested pairwisely to check whether they are related or not. For two related objects s_n and s_m, δ^n_{j+}, the offset of s_n in the bounding direction $\vec{d}_j$, is set to r_n, if s_m is in the direction of $\vec{d}_j$ by testing $\vec{d}_j \cdot \vec{p}_{nm} > 0$. Otherwise, δ^i_{j-} is set to r_n. The algorithm that computes offsets for all objects and bounding directions are illustrated in Figure 6. In Figure 5, various shapes of bounding volumes determined from the offsets are illustrated.

4. Interval Tree

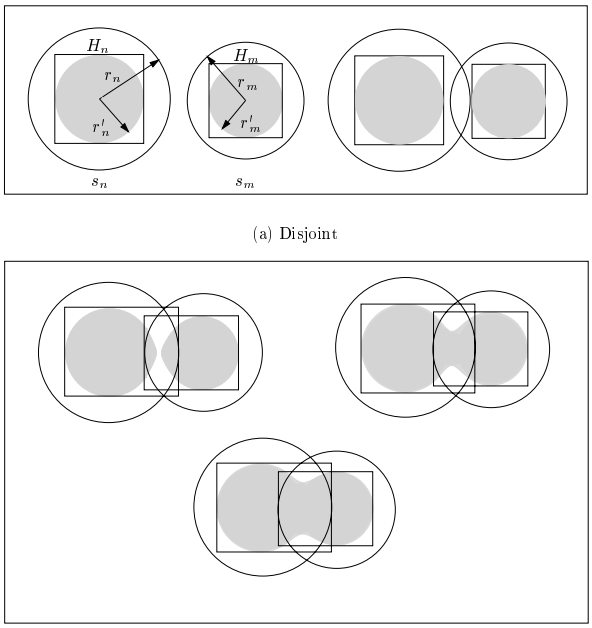
4.1. Definition of interval tree

Let s be a soft object consisting of N components s_i, $i = 1, \ldots, N$. The bounding volumes of of each component s_i is

$$H^i = \bigcap_{j=1}^{k} h^i_j,$$

where k is the number of bounding directions. As we have referred to in Section 3.2, the two bounding values of h^i_j are denoted by b^i_{j-} and b^i_{j+}, respectively.

For the soft object s, we construct k interval trees each of which corresponds to a single bounding direction. Let us



(a) Disjoint

(b) Related

Figure 5: *Relations of two components s_n and s_m according to their default and actual radii, and the shapes of bounding volumes H_n and H_m*

```
Compute offset ( Soft objects s_1, s_2, ..., s_N,
          bounding directions d_1, ..., d_k )
{
  for ( n = 1 to N ) {
    for (j = 1 to k ) {
      δ^n_{j+} ⇐ r'_n;
      δ^n_{j-} ⇐ r'_n;
    }
  }
  for ( n = 1 to N ) {
    for ( m = 1 to N ) {
      if ( n ≠ m ) {
        p_nm ⇐ the shortest
             distance vector from p_n to p_m;
        if ( ||p_nm|| ≤ r_n + r'_m ) {
          for ( j = 1 to k ) {
            if ( p_nm · d_j > 0 )
              δ^i_{j+} ⇐ r_n;
            else
                    δ^i_{j-} ⇐ r_n;
          }
        }
      }
    }
  }
}
```

Figure 6: *Algorithm for computing offsets*

consider an interval tree IT_j of a specific direction j such that $1 \le j \le k$. There are $2N$ boundary values b^i_{j-} and b^i_{j+} of h^i_j, for all $i = 1, ..., N$. For simplicity, we assume that all b^i_{j-} and b^i_{j+}, for $i = 1, ..., N$, are distinguished, and v_q's, $q = 1, ..., 2N$, denote a sorted list in ascending order of the boundary values. Then, IT_j is defined as follows:

1. The subset of IT_j including all internal (non-leaf) nodes and connecting edges between them is a one dimensional binary search tree. Thus, each internal node stores a bounding value v_q, $1 \le q \le 2N$.
2. Let l be a leaf node, and the parent of l be denoted by $p(l)$ storing a value v_q. The leaf node l has a set of influencing components denoted by $c(l)$ defined as follows:
   a. When l is a left child of $p(l)$,
      i. If $q = 1$, $c(l) = \emptyset$.
      ii. Otherwise, $c(l)$ is a set of the components influencing the region (v_{q-1}, v_q).
   b. When l is a right child of $p(l)$,
      i. If $q = 2N$, $c(l) = \emptyset$.
      ii. Otherwise, $c(l)$ is a set of the components influencing the region (v_q, v_{q+1}).

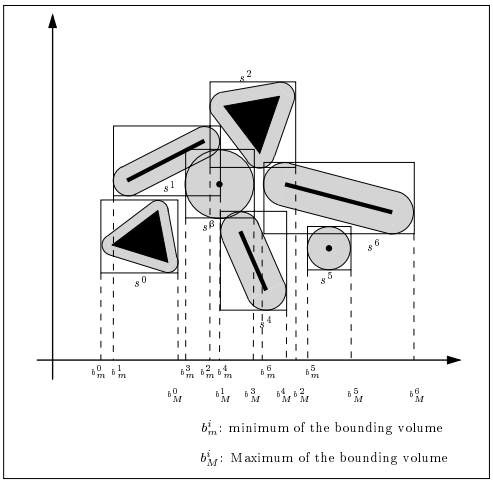
The data structure of an internal node is described in Figure 7. The `value` field stores the bounding value of the node. The `reference_count` field, which is initialized with 1, of a node stores the number of times of appearing the boundary value v_q in the list of v_q's, $q = 1, ..., 2N$. Thus, we do not have $2N$ internal nodes when any two or more boundary values are the same. If a `left` (resp. `right`) child of an internal node is a leaf node, `larray` (resp. `rarray`) pointer points to a leaf node, i.e., an array of component pointers. Thus, in this data structure, a leaf node l represents a set $c(l)$ directly. In Figure 4.1 an example interval tree from 7 objects is illustrated. In the figure, a component is indexed as s^i instead of s_i for the convenience of notation. Also, for simplicity, boundary values are denoted by b^i_m and b^i_M instead of b^i_{j-} and b^i_{j+}, respectively.

4.2. Building an Interval Tree

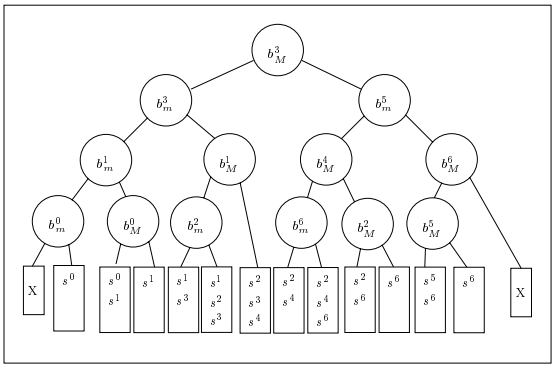
The algorithm for building an interval tree is composed of three phases: i) sorting all boundary values, ii) building a binary search tree from the sorted list of the boundary values, and iii) inserting each component to the leaf nodes that correspond to the intervals where the component influence. To build a balanced interval tree, the median value of the boundary values are selected and inserted to the interval tree recursively. When a value is inserted, if the `value` field of

```
struct Node
{
    float value;
    int reference_count;
    Node *left, *right;
    Object **larray, **rarray;
}
```

Figure 7: *Definition of an internal node*



(a) 7 primitives and minimum and maximum values of one bounding volume



(b) Interval tree built from the 7 primitives

captionAn example of interval tree

```
Algorithm Insert_Object ( Object s_i,
          Node *nd )
{
  if ( s_i.b_min < nd.value ) {
    if ( nd.larray = NULL )
      Insert_Object ( s_i, nd.left );
    else
      nd.larray ⇐ s_i;
  }
  if ( s_i.b_max > nd.value ) {
    if ( nd.rarray = NULL )
      Insert_Object ( s_i, nd.right );
    else
      nd.rarray ⇐ s_i;
  }
}
```

Figure 8: *Algorithm for inserting an object into an interval tree*

the visited node is identical to the value, then the inserting process stops by increasing the `reference_count` field of the node by 1. An interval tree from N components with $2N$ distinguished boundary values has $2N$ internal nodes and $2N+1$ leaf nodes. Note that the number of leaf nodes, which is the number of arrays, coincide with the number of intervals that divide space with $2N$ values. The correspondence between the interval and the leaf node must be maintained for all modifications of the interval tree.

The rule of inserting a component s_i having two boundary values b_{j-}^i and b_{j+}^i for a bounding direction j to an interval tree IT_j (the third phase of the building algorithm) is as follows. At an internal node having a `value` v_q, b_{j-}^i and b_{j+}^i are compared with v_q, and one of the following actions is taken.

- If $b_{j-}^i < b_{j+}^i \leq v_q$, then s_i is propagated down to the `left` child of the node.
- If $b_{j+}^i > b_{j-}^i \geq v_q$, then s_i is propagated down to the `right` child of the node.
- If $b_{j-}^i < v_q < b_{j+}^i$, s_i is propagated down to the both of `left` child and `right` child.

After reaching a leaf node, the pointer to s_i is inserted to the array in the leaf node. An algorithm of inserting a component to an interval tree is illustrated in Figure 8.

4.3. Field function evaluation using interval trees

When a field function is evaluated at a point p, p is projected onto each bounding direction $\vec{d}_j$. Then, the one dimensional projected coordinates, $\vec{d}_j \cdot p$, traverse down the corresponding interval tree IT_j to find an interval where the coordinates belong. An algorithm of searching the set of influenc-

```
Algorithm Search_IntervalTree
          ( float v_j, Node *nd )
{
  if ( v_j ≤ nd.value ) {
    if ( nd.larray ≠ NULL )
      return nd.larray;
    else
      Search_IntervalTree ( v_j, nd.left );
  }
  else {
    if ( nd.rarray ≠ NULL )
      return nd.rarray;
    else
      Search_IntervalTree ( v_j, nd.right );
  }
}

Algorithm Search_List ( Vertex ṽ )
{
  List l_1, l_2, ..., l_k;
  IntervalTree IT_1, IT_2, ..., IT_k;

  for ( j = 1 to k ) {
    l_j ⇐ Search_IntervalTree ( ṽ·d⃗_j, IT_j );
  }
  return Intersection ( l_1, l_2, ..., l_k );
}
```

Figure 9: *Algorithm for searching an object into an interval tree*

ing primitives for the projected coordinates is illustrated in Figure 9. For k bounding directions, k arrays are found, and the intersection of these k arrays are the final set of influencing primitives for the point.

4.4. Analysis of building and searching algorithms

An interval tree has $2N$ internal nodes and $2N-1$ leaf nodes. The size of the array in each leaf node can be N in the worst case. Thus, the interval tree takes $2N+(2N-1)N = O(N^2)$ storage. The construction time of an interval tree for N components is measured by separating the time for building the binary search tree (the first and second phases) and time for inserting the components into the arrays in the leaf nodes (the third phase). The first and second phases takes $O(N\log N)$. The third phase is equivalent to the one dimensional range searching problem: "given a set of discrete values and an interval, report all values that fall within the interval." The running time of the range searching problem is output-dependent. That is, if a component is inserted to α arrays, the insertion takes $O(\log N+\alpha)$ [4]. Therefore, inserting all N components to the interval tree takes $O(N\log N)$ and $O(N^2)$ time in the best and worst cases, respectively. In the evaluation of a field function, the interval tree that is completely balanced is traversed from the root to a leaf node, which takes $O(\log N)$ time.

5. Dynamic Update of Interval Tree

In interactive modeling process of a soft object, the components are inserted, deleted or editted dynamically. In such cases, the interval tree should be updated appropriately. We will consider the following topics for dynamic update of an interval tree:

- Insertion of a new component into an interval tree,
- Deletion of an existing component from an interval tree,
- Balancing an interval tree.

Note that the edition of a component can be implemented by the combination of a deletion and an insertion.

5.1. Inserting a new component

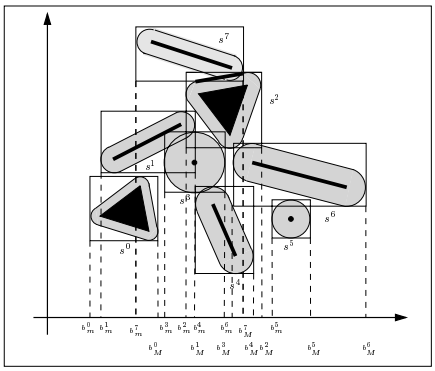
When a new component is inserted into an interval tree, the two boundary values of the component are inserted into the tree as two internal nodes. We can insert a bounding value using the conventional binary search algorithm utilized in the previous section. Let i be a new internal node that stores the inserted bounding value is created. Assume that i will be inserted as the left child of an existing node p that currently has a left child $l(p)$, where $l(p)$ is a leaf node. Then, the following steps are applied to insert i:

1. Disconnect $l(p)$ from p.
2. Make i be the left child of p.
3. Duplicate $l(p)$. Let $l'(p)$ be a copy of $l(p)$.
4. Make $l(p)$ and $l'(p)$ be the left and right child of i, respectively.

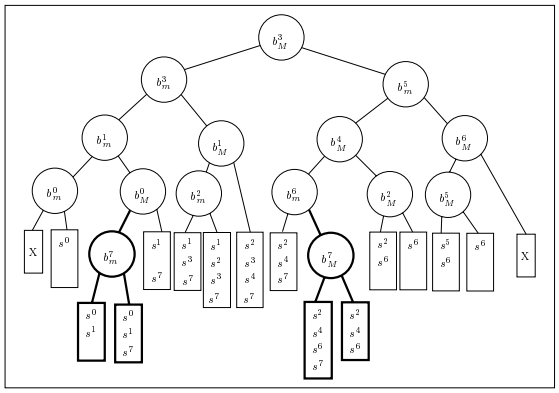
For the other case, when i will be the right child of p, the similar steps can be used. After two new internal nodes are created, the component itself is inserted to the components arrays in the leaf nodes of the interval tree using the algorithm in Figure 8. Figure 10 shows an interval tree after the insertion of new component s^7. The original interval tree is shown in Figure 4.1.

5.2. Deleting a component

When a component is deleted from a interval tree, first, the component pointers stored in the arrays of the leaf nodes are deleted. To delete the component pointers in the arrays, we apply an algorithm that is similar to the insertion algorithm in Figure 8. The only difference is that on reaching an array, the component pointer is deleted instead of inserted. Next, we traverse the tree to delete the internal nodes corresponding to two bounding values of the component. When we reach an internal node i of a bounding value, if i has `reference_count` > 1, then the traversal stops by decreasing the `reference_count` by 1. Otherwise, i is deleted and the tree is updated as follows.

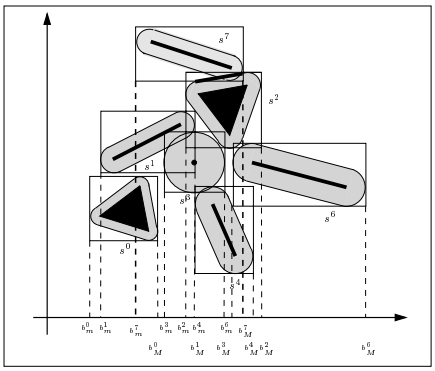


(a) New object s^7 is inserted

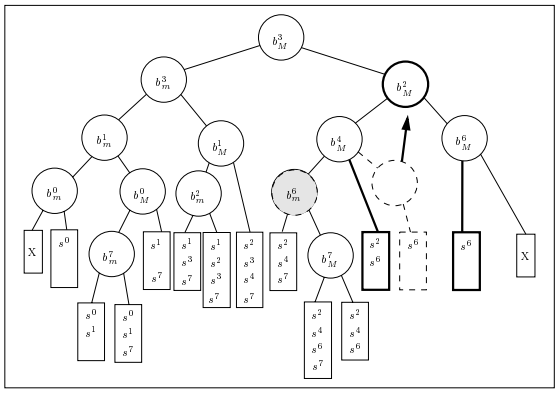


(b) Interval tree built after inserting s^7

Figure 10: *An example of inserting new object*



(a) Object s^5 is deleted



(b) Interval tree after deleting s^5

Figure 11: *An example of deleting an object*

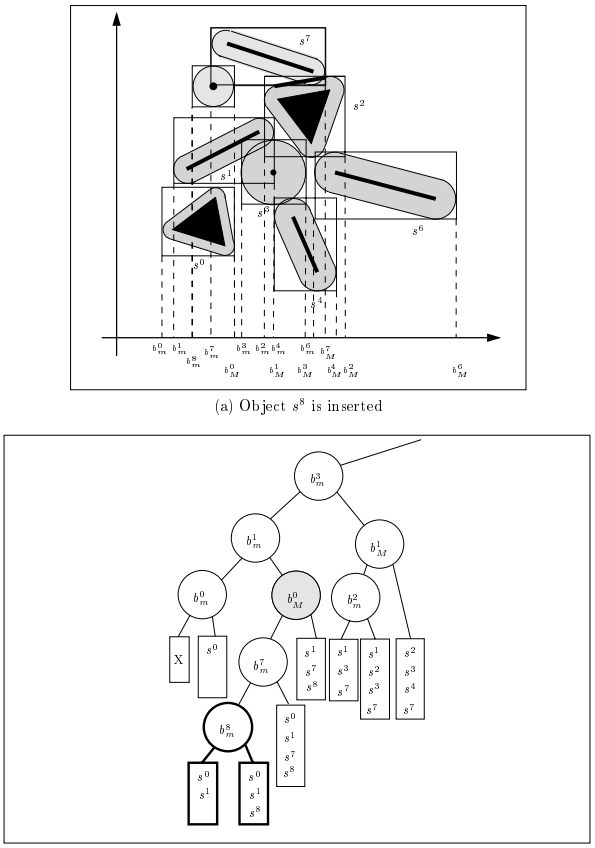
1. If both of two children $l(i)$ and $r(i)$ of i are leaf nodes,
   a. Merge the two arrays of component pointers in $l(i)$ and $r(i)$ and make a single leaf node nl storing the merged array.
   b. Substitute nl for i.
   c. Purge i.
2. Otherwise,
   a. If $l(i)$ is not a leaf node,
      i. Find an internal node j having the largest value in the left subtree of i.
      ii. Copy the value and the reference count of j into i.
      iii. If one of $l(j)$ and $r(j)$ is an internal node k, substitute k for j.
      iv. If both $l(j)$ and $r(j)$ are leaf nodes, make a merged leaf node nl as in Step 1(a), and substitute nl for j.
      v. Purge j.
   b. Otherwise, take a similar step to Step 2(a) with j = the internal node of the smallest value in the right subtree of i.

Note that in Step 2-(a)-iii, both of $l(j)$ and $r(j)$ cannot be internal nodes at the same time, since j must be the largest value in the left subtree of i. In Step 2-(a)-i, we only copy the value and the reference count of j to i. An elegant thing is that we do not have to modify the original child of the node i, even when the child is a leaf node. This is because we already pick out the component pointers to the deleted component before the deletion of internal nodes. That is, when both of the right child of i and j are leaf nodes, the arrays of the component pointers in the two leaf nodes are already the same after picking out the component pointers.

The above description shows that the correspondence between arrays and intervals are maintained after deleting a component from the tree. In Figure 11, we illustrate how an object, for example s^5, is deleted from the tree. Note that after deleting s^5, the interval tree becomes unbalanced.

5.3. Maintaining balance of the interval tree

Even though we built an initial balanced interval tree by inserting the median of the bounding values recursively, the dynamic modification may break the balance of the tree. We apply single rotation and double rotation, which are the techniques used in the AVL tree[14]. In Figure 11, since the unbalance is propagated in the alternating direction (b_m^7 is the right



(a) Object s^8 is inserted

(b) Left part of the interval tree after inserting s^8

Figure 12: *After inserting object s^8, the tree becomes unbalanced*

child of b_m^6 and b_m^6 is the left child of b_M^4), a double rotation is applied. After inserting an object s^8 to the resulting tree, another type of unbalance appears (See Figure 12). In this case, since the unbalance is propagated in the identical direction (b_m^8 is the left child of b_m^7 and b_m^7 is the left child of b_M^0), single rotation is applied. Through these methods, the balance of the interval tree is maintained.

5.4. Analysis of update algorithms

As well known, the balancing of AVL tree takes constant time [14]. By the balancing, we can always maintain the the height of an interval tree as $O(\log N)$, where N is the number of components. The running time of the insertion algorithm depends on the time of the algorithm described in Figure 8 for inserting a component pointers into the appropriate leaf nodes. As we referred to in Section 4.4, the algorithm takes $O(\log N + N) = O(N)$ times in the worst case. The running time of the deletion algorithm is $O(\log N)$, which is the same as that of the conventional deletion algorithm in binary search tree [14].

6. Implementation and Results

The proposed algorithm in this paper is implemented at a PC environment with Pentium-III CPU 500 MHz and 256 MB memories. The software environment is Visual C++ with MFC and OpenGL libraries. To measure the performance of the proposed algorithm to that of the conventional ones, we apply the algorithm for polygonizing soft objects. Marching cube-based conventional algorithm [11] is implemented for the polygonization. The marching cube method polygonizes the soft objects by decomposing space into small-sized cells and generating intersecting polygons with the cells and the objects. We designed two examples, each of which is composed of point skeletons and line segment skeletons, respectively. The first example is a smooth object composed of two loops. Each loop is approximated by line segments, which are applied as the primitives of soft objects. From this example, we can conclude that even though the number of generated polygons are nearly same, the number of primitives is an affecting factor for the performance. The measured performance and the statistics for the influencing primitives are illustrated in 13. Notice that the speed up of the proposed algorithm is due to the reduced number of influencing primitives. The second example is a mesh of point skeletons. In this example, we fix the number of primitives and change the number of generated polygons by controlling resolution of the cells. It is natural that the increase of the number of cells indicates the increase of the computation time. In this case, the computation time increases much faster for the conventional algorithms. Some of the blank column is due to the low performance of the platform, which prohibited us from achieving results. The shape of the soft object with 1210 point skeletons are suggested in accompanying file 1210.jpg. With the picture, we present two movie files to show the result of this paper. The first file, named example1.mpeg, compares the performance of polygonization using interval tree with the hash table method proposed by Wyvill [15]. The second file, named example2.mpeg, shows an interactive modeling environment of soft objects, which are polygonized with the proposed method.

7. Conclusion and Future Work

In this paper, we proposed a new algorithm for evaluating field functions of soft objects. To remove the unnecessary distance computations of conventional field functions, we built the bounding volumes of the objects and decomposed space into intervals by designing interval tree. Using this tree, a set of influencing primitives for a vertex is found without computing the distances between the vertex and all primitives. We applied this new field function in polygonizing soft objects. The conventional marching cube-based polygonization method using the proposed field function shows faster and more consistent response time than that using the conventional field function.

We are investigating the problem of the localized polygo-

No. of objects	No. of cells	type of function	No. of polygons	Time (sec)
121	128	Old	58994	81.658
		New	58994	14.300
	64	Old	14242	20.560
		New	14242	2.944
	32	Old	3162	5.207
		New	3162	0.912
605	128	Old	294234	2054.980
		New	294234	110.188
	64	Old	68450	508.811
		New	68450	17.970
	32	Old	12920	107.524
		New	12920	15.102
1210	128	Old		
		New	649422	110.188
	64	Old		
		New	149758	79.525
	32	Old	28306	513.288
		New	28306	25.507

Figure 13: *Comparison of performance with example presented in 1210.jpb*

nization using interval tree, which is the application of our work suggested in this paper. When a soft object is updated, the polygonization of the soft object must be recomputed. Since the area where the field function changes can be classified using interval tree, we can repolygonize the soft object only in that classified area without repolygonizing the whole object. With this extension, the interval tree can be one of the solutions to the interactive modeling environment using soft objects.

References

1. Barequet, G., Chazelle, B., Guibas, L. J., Mitchell, J. S. B., and Tal, A., "BOXTREE: A Hierarchical Representaion for Surfaces in 3D," Proceedings of Eurographics '96, pp. 387-396, 1996. 3
2. Blanc, C. and Schlick, C., "Extended Field Functions for Soft Objects," Proceedings of Implicit Surfaces '95, pp. 21 - 35, 1995. 2
3. Blinn, J. F., "A Generalization of Algebraic Surface Drawing," ACM Transactions on Graphics, Vol. 1, No. 3, pp. 235 - 256, 1982. 1, 2
4. de Berg, M., van Kreveld, M., Overmars, M., and Schwarzkopf, O., Computational Geometry: Algorithms and Applications, Springer-Verlag, 1997. 7
5. Gascuel, M-P., "An Implicit Formulation for Precise Contact Modeling between Flexible Solids," Proceedings of SIGGRAPH '93, pp. 313-320, 1993. 2
6. Gottschalk, S., Lin, M. C., and Manocha, D., "OBB-Tree: A Hierarchical Structure for Rapid Interference Detection," Proceedings of SIGGRAPH '96, pp. 171-180, 1996. 3
7. Hubbard, P. M., "Approximating Polyhedra with Spheres for Time-Critical Collision Detection," ACM Transactions on Graphcs, Vol. 15, No. 3, pp. 179-210, 1996. 3
8. Kacic-Alesic, Z. and Wyvill, B., "Controlled Blending of Procedual Implicit Surfaces," Proceedings of Graphics Interface '91, pp. 236-245, 1991. 2
9. Kay, T. L. and Kajiya, J. T., "Ray trace tracing complex scenes," Proceedings of SIGGRAPH '86, pp. 269-278, 1986. 3
10. Klosowski, J. T., Held, M., Mitchell, J.S.B., Sowizral, H., and Zikan, K., "Efficient collision detection using bounding volume hierarchies of k-DOPs," IEEE Transactions on Visualization and Computer Graphics, Vol. 4, No. 1, pp.21 - 36, 1998. 3
11. Lorensen, W. E. and Cline, H. E., "Marching Cubes: A High Resolution 3D Surface Construction Algorithm," Computer Graphics, Vol. 21, No. 4, pp. 163 - 169, 1987. 9
12. Murakami, S. and Ichihara, H., "On a 3D Display Method by Metaball Technique," Electronics Communications, Vol. 70D, No. 8, pp. 1607-1615, 1987. 2
13. Nishimura, H., Hirai, M., Kawai, T., Kawata, T., Shirakara, I., and Omura, K., "Object Modelling by Distribution Functions," Electronics Communications, Vol. 64D, No. 4, pp. 718-725, 1986. 2
14. Preiss, B., Data Structure and Algorithms with Objected-Oriented Design Patterns in C++, John Wiley & Sons, 1999. 8
15. Wyvill, G., McPheeters, C., and Wyvill, B., "Data Structure for Soft Objects," The Visual Computer, Vol. 2, No. 4, pp. 246 - 259, 1986. 1, 2, 3, 9
16. Zikan, K. and Konecny, P., "Lower Bound of Distance in 3D," In Proceedings of WSCG '97, Vol. 3, pp. 640-649, 1997. 3