COMMUNICATIONS IN COMPUTATIONAL PHYSICS
Vol. **2**, No. 3, pp. 545-576

Commun. Comput. Phys.
June 2007

Use of the Spatial *k*D-Tree in Computational Physics Applications

A. Khamayseh[1,*] and G. Hansen[2]

[1] *Computer Science and Mathematics Division, Oak Ridge National Laboratory, Oak Ridge, TN 37831, USA.*
[2] *Multiphysics Methods Group, Idaho National Laboratory, Idaho Falls, ID, USA.*

Received 13 July 2006; Accepted (in revised version) 8 November 2006

Available online 4 December 2006

Abstract. The need to perform spatial queries and searches is commonly encountered within the field of computational physics. The development of applications ranging from scientific visualization to finite element analysis requires efficient methods of locating domain objects relative to general locations in space. Much of the time, it is possible to form and maintain spatial relationships between objects either explicitly or by using relative motion constraints as the application evolves in time. Occasionally, either due to unpredictable relative motion or the lack of state information, an application must perform a general search (or ordering) of geometric objects without any explicit spatial relationship information as a basis. If previous state information involving domain geometric objects is not available, it is typically an involved and time consuming process to create object adjacency information or to order the objects in space. Further, as the number of objects and the spatial dimension of the problem domain is increased, the time required to search increases greatly. This paper proposes an implementation of a spatial k-d tree (*sk*D-tree) for use by various applications when a general domain search is required. The *sk*D-tree proposed in this paper is a spatial access method where successive tree levels are split along different dimensions. Objects are indexed by their centroid, and the minimum bounding box of objects in a node are stored in the tree node. The paper focuses on a discussion of efficient and practical algorithms for multidimensional spatial data structures for fast spatial query processing. These functions include the construction of a *sk*D-tree of geometric objects, intersection query, containment query, and nearest neighbor query operations.

AMS subject classifications: 52B10, 65D18, 68U05, 68U07

Key words: Geometric query, bounding volume hierarchy, *sk*D-tree, containment query, mesh generation, *h*-refinement, remapping.

*Corresponding author. *Email addresses:* `khamaysehak@ornl.gov` (A. Khamayseh), `Glen.Hansen@inl.gov` (G. Hansen)

1 Introduction

Computational physics applications are rapidly increasing in complexity to address evolving requirements to include more realistic models and more detailed domain representations. Requirements often include the incorporation of more complex geometric forms of the parts and components within the model along with a larger number of parts and components being used to form the computational domain. Indeed, the simple two-dimensional models of the recent past that typically employed structured mesh discretizations in which geometric objects were represented by line segments, have been replaced by complex three-dimensional unstructured meshes containing general objects defined by compositions of parametric curves and surfaces.

Common across a wide variety of applications is the need to perform spatial queries involving the geometric objects contained within the domain with respect to computational abstractions employed within the simulation application. For example, it is necessary to track the movement of objects with respect to the elements in a transient Eulerian finite element analysis application. Many other applications, including biological population modeling, molecular dynamics, multiphase fluid dynamics, scientific visualization, and solid modeling also require spatial query capabilities. Typically, there is a spectrum of application requirements for spatial query functionality, which range from a very general geometry query to the need to perform highly localized searches of nearby objects. In the general problem, the state and relationship of the geometric objects are unknown. This query problem involves determining the relationship between the domain objects and simulation abstractions in the most general case. In the latter localized search, the state of the objects and their relationships to each other are typically known to some degree. Perhaps the spatial location of the objects were known at a previous time step, for example. For this application, it is typically more efficient to make use of this known state information to economize the spatial query processing; a general search each time step is usually too costly. However, a general search is usually needed when the application initializes to construct the local information.

It is always preferable to use spatial adjacency information if it is available, instead of general spatial searching. For example, the use of an *Eulerian Walk* [1] for the transfer of data from one distinct mesh to another in a multiphysics application has a complexity of $\mathcal{O}(m+n)$, where m is the number of elements in the *source* mesh, and n is the number of elements in the *target* mesh. Generalized spatial searching, such as the algorithm proposed here, can determine element intersection in $\mathcal{O}(m\log n)$ time. Clearly, as the number of objects in the source and target meshes increase, the time required to perform the general search increases at a faster rate when the general approach is used. This paper proposes an implementation of a general search method; a spatial k-d tree (*sk*D-tree) for use by various applications when a general domain search is required.

In the parlance of geometric modeling, objects are usually described by their associated spatial attributes (*e.g.* location). Within a given modeling configuration, objects may "intersect" each other, be "adjacent" to one-another, and may "contain" other objects.

These relationships are "spatial" in nature. Further, these objects may also have "aspatial" attributes (*e.g.* the object's name). They may be grouped into three generic spatial object classes; namely "point", "line", and "region". The classification of an object into the above three classes is closely related to the object's *extent* (*e.g.* convex hull). The extent, which is the area of interest, may vary according to the application. One of the most fundamental problems in computational geometry involves queries that identify the aspatial attributes of entities based on their spatial attributes and the relationships between the entities.

Forming and maintaining the connection between the mathematics and the geometry of a physics simulation often consumes a significant portion of the total computation time necessary to perform the simulation. This "connection" appears in two basic forms; calculating the spatial relationship between two geometric entities (*e.g.* containment or nearest neighbor relationships), and associating simulation data with a geometric query region (*e.g.* determining the temperature at a particular spatial location).

This paper presents a spatial search structure, called a spatial *kd*-Tree or *sk*D-tree, that is designed to handle both of these classes of queries. Also included in this discussion is a presentation of robust data retrieval algorithms based on the *sk*D-tree for nearest neighbor, point containment, line-tree intersection, box-tree intersection, and tree-tree intersection operations. The *sk*D-tree and the algorithms associated with it are dimensionally independent, have space requirements that are a linear function of the input size, and often exhibit logarithmic query behavior in time. The emphasis of this paper is on algorithms that are theoretically and practically efficient, both in execution time and in implementation complexity, and on the description of the use of these algorithms for selected applications including:

1. geometric intersection calculation,
2. point location testing,
3. calculating minimum distances, and
4. adaptive mesh refinement.

Experience with these applications show the *sk*D-tree to be robust and computationally efficient for general query use. These applications will be discussed in more detail in Section 6.

2 Overview of the *sk*D-tree

The *sk*D-tree is intended for use on a set $\mathcal{G}$ of geometric "entities," where an entity is any geometric object with finite spatial extent. Examples of entities include points, line segments, polygons, polyhedra, and parametric surface patches. Objects with infinite extent, such as planes and lines, are excluded from this set $\mathcal{G}$. Some of the *sk*D-tree support algorithms presume that there are relationships between the entities in $\mathcal{G}$. For example, a point-in-body algorithm might presume that $\mathcal{G}$ represents the (closed) boundary of a

solid object. However, the internal mechanics of the *sk*D-tree search structure itself make no such assumption. One limiting restriction to this implementation of the *sk*D-tree is that the members of $\mathcal{G}$ are not allowed to move relative to one another after the tree is built. Rigid body transformations on the entire set $\mathcal{G}$ is permissible, however.

One of the more expensive spatial query operations of interest to simulation applications is determining the closest boundary point to a given query point. To obtain the closest boundary point typically requires efficient methods to perform the following queries:

(i) Given a query point $\mathbf{q}$ and a surface $\partial\mathcal{S}$, what is a nearest point to $\mathbf{q}$ on $\partial\mathcal{S}$?

(ii) Given a query ray $\mathbf{r}$ and a surface $\partial\mathcal{S}$, what is the nearest point to $\mathbf{q}$ along $\mathbf{r}$, if any?

One class of search structures that have utility in the implementation of query capabilities similar to the above is known as a *bounding volume hierarchy* (BVH). Each node of a BVH structure contains a bounding volume for some subset of the boundary geometry. The nodes are generally arranged in an oriented tree, where child nodes bound non-empty subsets of their parents' geometry. The bounding volumes are selected to minimize the cost of query operations (e.g. proximity, intersection, or containment) while providing a close fit to the underlying geometry. The *sk*D-tree proposed in this paper is one implementation of a BVH search structure.

The top-down constructive definition of a BVH begins by enclosing $\mathcal{G}$ in a tight-fitting *bounding volume* (BV), such as a box or a sphere. This "top-level" BV forms the root of a search tree. The elements of $\mathcal{G}$ are then partitioned into subsets, and the process is repeated recursively, where the bounding volume of each subset becomes a child of the current BV node. Bounding volume hierarchies are characterized by the type of the BVs used, the number of children at each interior node, the partitioning strategy, the depth to which the recursion is carried out, and their ability to support update (insertions/deletions).

Bounding volume hierarchies come in many forms: Swept Spheres [2], OBB-Trees [3], Sphere Trees [4, 5], and spatial *k*D-trees [6] are all examples. Analyzing the asymptotic behavior of BVHs is typically a challenging exercise. In the best case, BVH queries may be answered in constant time. In the worst case, each node of the tree may have to be visited, resulting in $\mathcal{O}(2n\text{-}1)$ work for the example of a binary tree with n leaf nodes. On average, the effort for intersection testing queries appears to be $\mathcal{O}(\log n)$, while nearest feature queries appear to be $\mathcal{O}(\alpha n)$, where $\alpha \leq 1$.

The *sk*D-tree uses *isothetic* (*i.e.*, axis-aligned) bounding boxes, arranged in a binary tree, using a strategy that balances the number of nodes in each child. Further, the *sk*D-tree recurses completely. This construction results in a height-balanced binary tree, with bounding volumes at each interior node, and a single geometric element at each leaf node. The *sk*D-tree does not support insertions or deletions, so it is best suited for static data. *sk*D-tree construction will be detailed later in the paper in Section 3.1, and the various query algorithms will be discussed in Sections 4 and 5.

A similar search structure to the *sk*D-tree is the oriented bounding box tree (OBBTree) of Gottschalk [3]. The primary differences between the OBBTree and the *sk*D-tree is that

the OBBTree uses oriented (non-isothetic) bounding boxes and attempts to balance spatial extent rather than tree height. The skd-Tree of Ooi [6] is also similar to the *sk*D-tree presented in this paper, however it supports dynamic insertion and deletion and does not fully recurse. Most BVHs stem from the *kD*-tree originated by Bentley [7]. Many other examples BVHs of exist today, differing in all the parameters listed above (see, e.g., [2, 5, 8, 9]). The *sk*D-tree presented here is a competitor to these other approaches; this method may perform better for some applications and may not be competitive for others, depending on the nature and requirements of the final application. The *sk*D-tree is a good match for applications that do not require dynamic tree modification and that map well to the use of isothetic bounding volumes. Indeed, for these problems, the *sk*D-tree provides quick tree construction, is simple to implement, and has good search characteristics on uniformly-distributed elements, such as mesh volumes.

Both the tree construction and various query algorithms are theoretically and practically efficient. In particular, the geometric query algorithms based on the *sk*D-tree may generally be performed in logarithmic time. For an *sk*D-tree consisting of N elements, there are a total of $2N$-1 nodes (internal and leaf) in the tree. For this case, the tree may be constructed in $\mathcal{O}(N\log N)$ time. This paper also presents various retrieval algorithms based on the *sk*D-tree for nearest neighbor, point containment, line-tree intersection, box-tree intersection, and tree-tree intersection. Each of these algorithms requires approximately $\log N$ time.

3 Geometric partitioning using the *sk*D-tree

In the field of computational simulation and modeling, spatial data is characterized as geometric or mesh data. Geometric objects consist of points, curves, surfaces, and solids; where mesh data might be in the form of vertices, lines, triangles, quads, tetrahedra, hexahedra or any other type of polygonal or polyhedral elements in $\mathbb{R}^2$and $\mathbb{R}^3$, respectively. Most applications mandate that geometric query be performed in a fast and efficient manner; ideally with a logarithmic worst-time performance. However, spatial data access remains a challenge; there are often important requirements in addition to execution time minimization. Additional requirements might include the support for a variety (*i.e.* multiple) query operations and the storage efficiency of the algorithm.

To support simulation applications, mesh data typically involves the computational grid used in the physics simulation application. For the purposes of connecting mesh data to a general geometric query library, a *region* $\mathcal{R}=\bigcup_{i=1}^{n} E_i$, is composed of n geometric or mesh elements E_i. Further, this development assumes that $\mathcal{R}$ is not self-intersecting and oriented in a fashion consistent with the orientations of the individual geometric elements E_i.

It is often essential for the spatial query mechanism to support a variety of query operations. Several types of point, line, and range queries that might be applied to spatial data include:

Exact match query:	Find all geometric objects that have the same spatial extent as the target query object **q**.
Overlap or intersection query:	Find all geometric objects that have at least one point in common with the object **q**.
Enclosure query:	Find all geometric objects that enclose an object **q**.
Containment query:	Find all geometric objects that are enclosed by an object **q**.
Adjacency query:	Find all geometric objects that have common boundaries with an object **q**.
Nearest-neighbor query:	Find all geometric objects that have a minimum distance from an object **q**.

3.1 *sk*D-tree construction for general objects

This section presents the algorithm and a brief description of the method for constructing the *sk*D-tree for a set of N geometric objects. The *sk*D-tree construction algorithm produces a *link array* L and a *safety box array* S. These arrays are linear representations of binary trees, and each contain $2N$-1 nodes. For a node i, the link element L_i contains the index of the children of i, or the index of one of the geometric objects if node i is a leaf node. The safety box element S_i contains the "safety box" of all of the objects found below node i.

This derivation employs the term "safety box" rather than "bounding box" to indicate that the box is the *tightest practical* isothetic bounding box from an implementation perspective. Indeed, a safety box is the actual minimum bounding box of the entity, inflated by 2ϵ. This expansion allows the implementation to avoid various degenerate cases at the boundary of the box. For example, a line segment contained within a safety box is known not to intersect the walls of the box, which are at least ϵ away from the line segment.

Algorithm 3.1 is used to construct an *sk*D-tree containing general d-dimensional discrete objects. The creation of the tree uses a *stack* data structure (depth-first; last-in first-out), and the tree traversal may use either a stack or *queue* (breadth-first; first-in first-out) data structure. The approach computes the safety boxes for the individual geometric objects and stores them in a temporary array. For efficiency purposes, the centroid of each safety box is also calculated and stored in a separate array. This information could be computed on demand when required by the algorithm if space is a concern. An integer array *perm* is constructed, that contains a permutation of the integers $\{1,\cdots,N\}$. This permutation will be modified as the balanced binary tree is constructed; the final state of the permutation array indicates the order of the objects in the leaf nodes.

Algorithm 3.1: General object *sk*D-tree construction.

Input: d-dimensional tree elements T_i, $1 \le i \le N$
For each element T_i, $1 \le i \le N$
  $B_i \leftarrow$ safety box of T_i
  $\mathbf{c}_i \leftarrow$ centroid of B_i
  $perm(i) \leftarrow i$
Allocate tree safety box array $S_{1:2N}$, and tree link array $L_{1:2N}$
$S_1 \leftarrow$ safety box of $\bigcup_{i=1}^{N} B_i$
Push $\{$node 1, $perm(1:N)\}$ onto the stack.
$next \leftarrow 2$
Do until stack is empty
  Pop $\{parent$ node, $perm(imin:imax)\}$ off stack
  $imed \leftarrow \frac{imin+imax}{2}$
  If $(imax-imin)=0$
    $S_{parent} \leftarrow B_{perm(imed)}$
    $L_{parent} \leftarrow$ pointer to leaf node $T_{perm(imed)}$
  Else
    Choose longest dimension j of S_i $(1 \le j \le d)$
    Reorder $perm(imin:imax)$ so that
      $c^j_{perm(k)} \le c^j_{perm(imed)}$, $k \le imed$
      $c^j_{perm(imed)} \le c^j_{perm(k)}$, $imed+1 \le k$
    $L_{parent} \leftarrow next$
    $S_{next+0} \leftarrow \bigcup_{k=imin}^{imed} B_{perm(k)}$
    $S_{next+1} \leftarrow \bigcup_{k=imed+1}^{imax} B_{perm(k)}$
    Push $\{$node $next+0$, $perm(imin:imed)\}$ onto the stack
    Push $\{$node $next+1$, $perm(imed+1:imax)\}$ onto the stack
    $next \leftarrow next+2$
Output: *sk*D-tree consisting of $S_{1:2N}$ and $L_{1:2N}$

3.2 Details of the algorithm

Assume that the target query space consists of geometric objects that may be decomposed into N d-dimensional elements. In most physics applications, $d = 3$, and the discrete elements are typically restricted to points, lines, triangles, surface patches, quadrilatera, tetrahedra, hexahedra, or some other type of polyhedral element. To employ the *sk*D-tree, one associates with each element a representative point (or "key") which is the center of the bounding box of that element. By alternately locating the median coordinate of the keys in the three coordinate directions, one bisects the set of keys, producing a median *sk*D-tree. Constructed in this manner, the tree is a balanced binary tree with exactly $2N$-1 nodes on $\log N$ levels and allows geometric queries in logarithmic time. Each mesh element is associated with a unique leaf. The tree is constructed in $\mathcal{O}(N \log N)$ time.

The body of the construction algorithm begins by removing the current (node, range) pair from the stack. If the range is of length one, the node is a leaf, and the algorithm creates a link entry referring to the geometric object contained by the safety box describing this node. If the range contains more than one element, the node partitioning stage of the algorithm is invoked. The goal of this stage is to partition the objects contained in the current safety box into two distinct sets. This is accomplished by identifying the longest axis of the safety box, say the j axis. The median value of the j^{th} component of the centroids of the safety boxes is computed (see, e.g., [10]), then the permutation subset is partially ordered about this value. New safety boxes are created for the nodes in the left half and the right half of the permutation subset, and the respective node values are pushed on to the stack. The algorithm continues until the stack is empty.

The above discussion is valid independent of the actual type of objects that the algorithm is processing. It is probably easier to understand if the object type is restricted to triangles. For clarity, the remainder of this presentation assumes that the discrete elements that form the target query space are triangles, arbitrarily-oriented in three dimensional space.

The *sk*D-tree algorithm accepts a set of N triangles and produces an *sk*D-tree that is stored in a link array L_i of size $2N$-1. Leaf nodes in L each contain exactly one triangle index. Because of the possibility of triangle overlap, the algorithm also produces an array S of size $2N$-1 which contains the safety boxes. Associated with each node i of the tree is the safety box S_i which is the smallest isothetic rectangular box which fully encloses all the mesh elements in the subtree associated with node i.

- If node i is a leaf, S_i is the safety box of its associated element.
- S_1 (the box corresponding to the root node) is the safety box of the entire mesh.

Again, the collection $\{S_i\}$ of nested overlapping boxes guarantees that geometrical queries against the *sk*D-tree are rigorous.

The safety box associated with the root node S_1 contains all the triangle bounding boxes below it in the tree. The centroids of the triangle bounding boxes are stored in $\mathbf{c}_i$. If there is only one triangle in the tree, the root node is a leaf. In the actual implementation, the algorithm employs the convention that the value of the handle corresponding to a leaf is set to the negative of the unique triangle contained in that leaf. If the root node is a leaf, the tree is complete at this stage. An integer array *perm* is used as a stack to contain a permutation of the integers $\{1,\cdots,N\}$. The root node "1" is initially pushed onto the top of the stack. As other elements are added, the array subset of *perm* (*i.e.*, subset of triangles associated with this node) is formed. This permutation of elements will be altered as the balanced binary tree is constructed.

As the algorithm continues, the "cutting direction" for bisecting the set of triangles corresponding to this node is stored. This direction is one of the coordinate directions ($\hat{x}_j$, where $j=1,\cdots,d$), depending on which dimension of the bounding box is largest. As elements are added, nodes are popped off the stack to create children nodes, which are modified and then subsequently placed back onto the stack. This process continues

until the *sk*D-tree has been created and all target elements have been added. A detailed presentation of this portion of the algorithm begins with the median selection algorithm.

Median selection is used to partition the triangle subset associated with the current node of the tree. Using the selected cutting direction, the *select* list is used to reorder *perm* such that the triangle with the median bounding box center coordinate is denoted T_{imed}, while the triangles $\{T_i,\ i<imed\}$ have smaller (or equal) bounding box coordinates, and the triangles with $\{T_i,\ i>imed\}$ have greater (or equal) bounding box coordinates. Floyd and Rivest [11], Hoare [12], and Knuth [10] present other efficient median selection approaches.

If the first child node's subset of triangles is a singleton, the child is a leaf. In this case, initialize the child's link to point to the negative of the triangle number and initialize the child's bounding box to be equal to the triangle's bounding box. In the case that the subset of triangles corresponding to the first child is more than one triangle, the child is not a leaf. For this outcome, the bounding box of this child is computed as the smallest box containing all the bounding boxes of the subordinate triangles.

Given the constructed child node, push the child onto the stack. Store the associated triangle subset in *imin* and *imax* and the cutting direction in *ict*. Repeat this process for each additional child node to complete the tree construction.

In practice, this *sk*D-tree construction technique often represents a good balance of tree construction time and query efficiency. Defining optimality for a BVH structure is challenging because of the relationship between the spatial distribution of the geometric elements in the BVH and the spatial distribution (and extent) of query elements. The construction algorithm presented above creates a height-balanced binary tree in the number of geometric elements: $2N$-1 nodes and a maximum path length of $\lceil \log N \rceil$. Balancing in terms of tree-height works well when the geometric elements are approximately equal in extent and distributed uniformly across space.

Algorithm 3.2 presents a modification to the basic tree construction method. The modification adds a convenience feature; the concept of a *stateless query*. For the purposes of this paper, a stateless query simply adds the capability to query the tree using the same function call as that used to build it. This algorithm is a blend between the tree construction method and an element query algorithm that is presented in the following section; its operation will become clear in the following discussion.

4 Proximity query against the *sk*D-tree

The basic algorithm for querying against the *sk*D-tree concerns the retrieval of geometric elements in the search tree that are guaranteed to contain the nearest point to a given query point. The nearest point query uses the *sk*D-tree structure corresponding to a geometric surface (the tree contains surface patches that span the surface of interest; stored in the tree using Algorithm 3.1) to accelerate finding the nearest point on the surface to the given query point $\mathbf{q}$. What is actually returned is a small subset of leaves (patches

Algorithm 3.2: Build-on-query *sk*D-tree construction.

Input: d-dimensional tree elements T_i, $1 \le i \le N$
d-dimensional query element Q
Do once on first tree query
For each element T_i, $1 \le i \le N$
$B_i \leftarrow$ safety box of T_i
$\mathbf{c}_i \leftarrow$ centroid of B_i
$perm(i) \leftarrow i$
Allocate tree safety box array $S_{1:2N}$, and tree link array $L_{1:2N}$
$S_1 \leftarrow$ safety box of $\bigcup_{i=1}^{N} B_i$
$next \leftarrow 2$
Push {node 1, $perm(1:N)$} onto the stack.
Do until stack is empty
Pop {$parent$ node, $perm(imin:imax)$} off stack
If Q intersects with S_{parent}
If L_{parent} indicates a leaf
Locate T_i pointed to by L_{parent}
Add $\{T_i\}$ to the collision list
Else
Do once on first node query
$imed \leftarrow \frac{imin+imax}{2}$
If $(imax - imin) = 0$
$S_{parent} \leftarrow B_{perm(imed)}$
$L_{parent} \leftarrow$ pointer to leaf node $T_{P(imed)}$
Else
Choose longest dimension j of S_i $(1 \le j \le d)$
Reorder $perm(imin:imax)$ so that
$c^j_{perm(k)} \le c^j_{perm(imed)}, \quad k \le imed$
$c^j_{perm(imed)} \le c^j_{perm(k)}, \quad imed+1 \le k$
$L_{parent} \leftarrow next$
$S_{next+0} \leftarrow \bigcup_{k=imin}^{imed} B_{P(k)}$
$S_{next+1} \leftarrow \bigcup_{k=imed}^{imax} B_{P(k)}$
$next \leftarrow next+2$
Push {node $L_{parent+0}$, $perm(imin:imed)$} onto the stack
Push {node $L_{parent+1}$, $perm(imed+1:imax)$} onto the stack
Output: Collision list of tree elements T_i and a subtree in $S_{1:2N}$ and $L_{1:2N}$

or individual triangles) that feasibly could contain the nearest point. The user must then perform the relevant geometric tests on this small subset to actually determine the nearest point as required by the ultimate application.

4.1 Point query

Consider a query originating at a point in space, with the goal of obtaining the closest point on a surface (actually the closest safety box contained in the tree) to the query point in d-dimensional space.

First, initialize the minimum distance $d_{\max}$ to the surface to be the distance from the query point to the point most distant in the collection of safety boxes forming the *sk*D-tree. Indeed, given even one element in the tree, $d_{\max}$ is always well-defined.

Given a d-dimensional point $\mathbf{q}=(q_1,q_2,\cdots,q_d)$ and an isothetic bounding box $B=\Pi_{i=1}^{d}[a_i,b_i]$ in Cartesian space, the minimum distance between $\mathbf{q}$ and B is computed using the generalized Pythagorean Theorem. The minimum distance from $\mathbf{q}$ to B will be the length of the straight line path from $\mathbf{q}$ to the nearest point $\mathbf{x}=(x_1,x_2,\cdots,x_d)\in B$. If one views this path as the sum of the subpaths parallel to each of the d coordinate directions $\hat{\mathbf{x}}_i$, then each subpath will be the shortest path between q_i and the interval $[a_i,b_i]$. That is, each subpath

$$\overrightarrow{\mathbf{qx}}=\sum_{i=1}^{d}(x_1,\cdots,x_{i-1},q_i,\cdots,q_d)(x_1,\cdots,x_i,q_{i+1},\cdots,q_d)$$

has the *shortest* length if

$$x_i=\begin{cases} a_i & \text{if } q_i\le a_i,\\ q_i & \text{if } a_i<q_i<b_i,\\ b_i & \text{if } b_i\le q_i.\end{cases}$$

Each of these possible subpaths have the minimum length; hence $\mathbf{x}$ is the closest point. Thus, the minimum distance between $\mathbf{p}$ and B may be computed using the expression

$$\begin{aligned} d_{\min}(\mathbf{q},B)&=\min_{\mathbf{x}\in\Pi_{i=1}^{d}[a_i,b_i]}\|\mathbf{q}-\mathbf{x}\|\\ &=\sqrt{\sum_{i=1}^{d}\min_{x_i\in[a_i,b_i]}(q_i-x_i)^2}=\sqrt{\sum_{i=1}^{d}(\max\{0,a_i-q_i,q_i-b_i\})^2}.\end{aligned}$$

Similar to the above, computing the the maximum distance between $\mathbf{p}$ and B is based on the observation that each subpath has the *longest* length if

$$x_i=\begin{cases} a_i & \text{if } |q_i-a_i|\ge|q_i-b_i|,\\ b_i & \text{if } |q_i-a_i|<|q_i-b_i|.\end{cases}$$

In this case, each possible subpath has the maximum length, and hence $\mathbf{x}$ is the farthest point from B. Thus, the maximum distance between $\mathbf{p}$ and B may be computed

using the expression

$$d_{\max}(\mathbf{q},B)=\max_{\mathbf{x}\in\Pi_{i=1}^{d}[a_i,b_i]}\|\mathbf{q}-\mathbf{x}\|$$
$$=\sqrt{\sum_{i=1}^{d}\max_{x_i\in[a_i,b_i]}(q_i-x_i)^2}=\sqrt{\sum_{i=1}^{d}(\max\{q_i-a_i,b_i-q_i\})^2}.$$

4.2 Sphere query

Given a d-dimensional sphere $S=\{\mathbf{c},r\}$ and an isothetic bounding box B in Cartesian space, where

$$\mathbf{c}=(c_1,c_2,\cdots,c_d)\quad\text{and}\quad B=\prod_{i=1}^{d}[a_i,b_i],$$

the minimum distance between S and B may be computed using the expression

$$d_{\min}(S,B)=\max\{0,d_{\min}(\mathbf{c},B)-r\},$$
$$d_{\max}(S,B)=d_{\max}(\mathbf{c},B)+r.$$

Given two d-dimensional axi-symmetric bounding boxes $B^\alpha=\prod_{i=1}^{d}[a_i^\alpha,b_i^\alpha]$ and $B^\beta=\prod_{i=1}^{d}[a_i^\beta,b_i^\beta]$ in Cartesian space, the minimum distance between B^α and B^β is computed using the generalized Pythagorean Theorem. The minimum distance from B^α to B^β is the length of the straight line path between the nearest two points $\mathbf{x^{ff}}=(x_1^\alpha,x_2^\alpha,\cdots,x_d^\alpha)\in B^\alpha$ and $\mathbf{x^{fi}}=(x_1^\beta,x_2^\beta,\cdots,x_d^\beta)\in B^\beta$.

Again, if one views this path as the sum of the subpaths parallel to each of the d coordinate directions $\hat{\mathbf{x}}_i$; each subpath will be the shortest path between the intervals $[a_i^\alpha,b_i^\alpha]$ and $[a_i^\beta,b_i^\beta]$. That is, each subpath

$$\overrightarrow{\mathbf{x^{ff}}\mathbf{x^{fi}}}=\sum_{i=1}^{d}(x_1^\alpha,\cdots,x_{i-1}^\alpha,x_i^\beta,\cdots,x_d^\beta)(x_1^\alpha,\cdots,x_i^\alpha,x_{i+1}^\beta,\cdots,x_d^\beta)$$

has the *shortest* length if

$$x_i^\alpha=\begin{cases}a_i^\alpha & \text{if } a_i^\beta\le a_i^\alpha,\\ a_i^\beta & \text{if } a_i^\alpha<a_i^\beta<b_i^\alpha,\\ b_i^\beta & \text{if } a_i^\alpha<b_i^\beta<b_i^\alpha,\\ b_i^\alpha & \text{if } b_i^\alpha\le b_i^\beta,\end{cases}\quad\text{and}\quad x_i^\beta=\begin{cases}a_i^\beta & \text{if } a_i^\alpha\le a_i^\beta,\\ a_i^\alpha & \text{if } a_i^\beta<a_i^\alpha<b_i^\beta,\\ b_i^\alpha & \text{if } a_i^\beta<b_i^\alpha<b_i^\beta,\\ b_i^\beta & \text{if } b_i^\beta\le b_i^\alpha.\end{cases}$$

Each possible subpath has the minimum possible length, hence $\mathbf{x}^\alpha$ and $\mathbf{x}^\beta$ are the closest two points. Thus, the minimum distance between B^α and B^β may be computed using the

expression

$$d_{\min}(B^\alpha,B^\beta)=\min_{\substack{\mathbf{x}^\alpha\in\Pi_{i=1}^d[a_i^\alpha,b_i^\alpha]\\ \mathbf{x}^\beta\in\Pi_{i=1}^d[a_i^\beta,b_i^\beta]}}\|\mathbf{x}^\alpha-\mathbf{x}^\beta\|$$

$$=\sqrt{\sum_{i=1}^d\min_{\substack{x^\alpha\in[a_i^\alpha,b_i^\alpha]\\ x^\beta\in[a_i^\beta,b_i^\beta]}}(x_i^\alpha-x_i^\beta)^2}=\sqrt{\sum_{i=1}^d\Big(\max\{0,a_i^\alpha-b_i^\beta,a_i^\beta-b_i^\alpha\}\Big)^2}.$$

To compute the maximum distance between B^α and B^β requires that each subpath has the *longest* length if

$$x_i^\alpha=\begin{cases}a_i^\alpha & \text{if } |b_i^\beta-a_i^\alpha|\geq|a_i^\beta-b_i^\alpha|,\\ b_i^\alpha & \text{if } |b_i^\beta-a_i^\alpha|<|a_i^\beta-b_i^\alpha|,\end{cases}$$

and

$$x_i^\beta=\begin{cases}a_i^\beta & \text{if } |b_i^\alpha-a_i^\beta|\geq|a_i^\alpha-b_i^\beta|,\\ b_i^\beta & \text{if } |b_i^\alpha-a_i^\beta|<|a_i^\alpha-b_i^\beta|.\end{cases}$$

Each possible subpath has the maximum length; hence $\mathbf{x}^\alpha$ and $\mathbf{x}^\beta$ are the farthest two points. Thus, the maximum distance between $\mathbf{x}^\alpha$ and $\mathbf{x}^\beta$ may be computed using the expressions

$$d_{\max}(B^\alpha,B^\beta)=\max_{\substack{\mathbf{x}^\alpha\in\Pi_{i=1}^d[a_i^\alpha,b_i^\alpha]\\ \mathbf{x}^\beta\in\Pi_{i=1}^d[a_i^\beta,b_i^\beta]}}\|\mathbf{x}^\alpha-\mathbf{x}^\beta\|$$

$$=\sqrt{\sum_{i=1}^d\max_{\substack{x^\alpha\in[a_i^\alpha,b_i^\alpha]\\ x^\beta\in[a_i^\beta,b_i^\beta]}}(x_i^\alpha-x_i^\beta)^2}=\sqrt{\sum_{i=1}^d\Big(\max\{b_i^\beta-a_i^\alpha,b_i^\alpha-a_i^\beta\}\Big)^2},$$

$$d_{\min}(B^\alpha,B^\beta)=\sqrt{\sum_{i=1}^d\Big(\max\{0,a_i^\alpha-b_i^\beta,a_i^\beta-b_i^\alpha\}\Big)^2},$$

and

$$d_{\max}(B^\alpha,B^\beta)=\sqrt{\sum_{i=1}^d\Big(\max\{b_i^\beta-a_i^\alpha,b_i^\alpha-a_i^\beta\}\Big)^2}.$$

5 Nearest object query with respect to the *sk*D-tree

Algorithm 5.1 uses the *sk*D-tree structure to represent a geometric surface. The query algorithm attempts to locate all the bounding boxes representing surface elements (*i.e.*

Algorithm 5.1: Nearest object query against *sk*D-tree.

Input: d-dimensional tree elements T_i, $1 \le i \le N$
*sk*D-tree consisting of $S_{1:2N}$ and $L_{1:2N}$
d-dimensional query element Q
$d_{\text{known}} \leftarrow maxdist(\mathbf{q}, S_1)$
Push {node 1} onto the stack
Do until stack is empty
  Pop {*parent* node} off the stack
  $d_{\min} \leftarrow mindist(\mathbf{q}, S_{parent})$
  If $d_{\min} < (d_{\text{known}} + \epsilon)$
    $d_{\max} \leftarrow maxdist(\mathbf{q}, S_{parent})$
    $d_{\text{known}} \leftarrow \min(d_{\text{known}}, d_{\max})$
    If L_{parent} indicates a leaf
      Find T_i pointed to by L_{parent}
      Add $\{T_i, d_{\min}\}$ to the temporary collision list
    Else
      Add $\{L_{parent+0}\}$ to the stack
      Add $\{L_{parent+1}\}$ to the stack
For all entries j in the temporary collision list
  Examine j^{th} entry $\{T_i, d_{\min}\}$
  If $d_{\min} < (d_{\text{known}} + \epsilon)$
    Add *non-duplicate* $\{T_i\}$ to the final collision list
Output: Final collision list of tree elements T_i

triangles) that may contain the closest point to the query point. This list of boxes are candidates to contain the nearest point; one of them will, but the user must perform geometric tests on this small collection of objects to actually determine which one contains the closest point and what that location is.

The query simply returns the set of isothetic boxes in the tree that are "close to" the query box. The user must then determine the actual box that contains the closest point, and find the point on the actual object surface representation (not using the bounding box or discrete element approximation). One might initially question the advantages of using the *sk*D-tree at this point, if one must use a root finding (or optimization) approach to actually locate the closest point. Indeed, the query algorithm greatly localizes the search space that must be examined with the root finding algorithm. Thus, the combination of the query and the conventional root finding algorithm should generally be much faster than using the conventional approach alone.

To accelerate the tree traversal, a reference point $\mathbf{p}$ on the surface is often used to reduce the time spent searching the tree. It may be possible to "guess" this reference point rather easily, given the specifics of the objects stored in the tree and/or prior knowledge of the search process. The logic behind the reference point selection is to short-circuit the tree search by establishing a representative $d_{\min}$. This trick can significantly speed up the

search if **p** is *almost* the closest point. Caution is indicated however; if a reference point was chosen in error resulting in a $d_{\min}$ being less than the closest distance between the query point and the surface, the algorithm will fail.

Algorithm 5.1 details the general nearest-object query function. The search partially traverses the *sk*D-tree using a stack until completed. A tree node is only of interest if the minimum "optimistic" distance for the node is less than the known "pessimistic" distance for the whole surface. The use of a search ϵ insures a conservative process; it is necessary to avoid discarding a valid node because of machine precision issues.

The algorithm proceeds as follows. For each child of a given tree node, compute the pessimistic and optimistic distances. Reduce the global pessimistic distance $d_{\min}$ if appropriate. Order the children such that the child with the smaller optimistic distance will be placed above the other child when they are placed on the stack. Loop through the children and verify that all children remain candidates (ignore those whose optimistic distance is less than $d_{\min}$). If the child is a leaf, add it to triangle list, and add its optimistic distance to the leaf array maintaining these distances. If the child is not a leaf, place it on the stack.

Next, compress the list of candidate leaves by rejecting any leaves whose optimistic distance is less than $d_{\min}$ ($d_{\min}$ may have decreased since a particular leaf was added to the list). During the compression step, convert the leaf addresses to triangle indices simultaneously.

5.1 Line segment intersection query against the *sk*D-tree

The algorithm developed in this section uses the *sk*D-tree structure to represent a geometric surface. The query algorithm attempts to compute all intersections of the surface with an input ray or line segment. Again, the *sk*D-tree is traversed using a stack. The query algorithm returns a small subset of leaves (*i.e.* triangles) that may contain an intersection with the line segment. Using the *sk*D-tree, the ray/triangle intersection requires $\mathcal{O}(\log N)$ operations. As with the previous query algorithm, the user must perform geometric tests on the small subset of objects to actually determine the points of intersection of the underlying computational surface (if there indeed are any).

Algorithm 5.2 begins with the perturbation of the endpoints of the line segment such that the endpoints have a projection distance on the three coordinate axes of at least ϵ; this projection is compared with the tree. If the minimum line segment coordinate is greater than the maximum bounding box coordinate, or conversely, if the maximum line segment coordinate is less than the minimum bounding box coordinate, an intersection cannot exist between the objects.

Generally, there are two methods to test if a ray or line segment intersects a box. First, a bounding box may be created for the line segment and tested against the given box. Secondly, the line segment may be intersected with the six plane faces of the box. It is usually more efficient to use the bounding box approach. This technique is based on, for each of the x, y, and z directions, projecting the "safety box" of the node onto the

Algorithm 5.2: Line segment intersection query against the *sk*D-tree.

Input: d-dimensional tree elements $T_i, 1 \le i \le N$
*sk*D-tree consisting of $S_{1:2N}$ and $L_{1:2N}$
Insure width of L in each coordinate direction $\hat{x}_j$, $1 \le j \le d$, is at least ϵ
Push {node 1} onto the stack
Do until stack is empty
  Pop {*parent* node} off the stack
  For each coordinate direction $\hat{x}_j, 1 \le j \le d$
    $I_j \leftarrow$ projection of $(S_{parent} + 2\epsilon)$ along the $\hat{x}_j$-axis onto L
      (I_j is an interval in the parametric space of L)
  If $\left(([0,1] \bigcap_{j=1}^{d} I_j \right) \neq \emptyset$
    If L_{parent} indicates a leaf
      Find T_i pointed to by L_{parent}
      Add $\{T_i\}$ to the collision list
    Else
      Add $\{L_{parent+0}\}$ onto the stack
      Add $\{L_{parent+1}\}$ onto the stack
Output: Collision list of tree elements T_i

one-dimensional subspace spanned by the line segment as follows:

1. Define the bounding box $B = \Pi_{i=1}^{d} [a_i, b_i]$ and line segment $L = \{t\mathbf{u} + (1-t)\mathbf{v} | t \in [0,1]\}$, where $\mathbf{u} = (u_i, \cdots, u_d)$ and $\mathbf{v} = (v_1, \cdots, v_d)$.

2. For each coordinate direction $\hat{x}_i$, $1 \le i \le d$, set $I_i =$ projection of B along the $\hat{x}_i$-axis onto L (I_i is an interval in the parametric space of L).

3. The orthogonal projection $\mathcal{P}_L$ onto the one-dimensional subspace spanned by the line segment L, is $\mathcal{P}_L(B) = [0,1] \bigcap_{i=1}^{d} I_i$, where $I_i = \left[\frac{a_i - u_i}{v_i - u_i}, \frac{b_i - u_i}{v_i - u_i} \right]$.

In the above, the parameter for the one-dimensional subspace has (by convention) a value of zero at one endpoint and a value of one at the other endpoint of the segment. After obtaining three such projection intervals, one may conclude that the line segment intersects the box if the three intervals have a nonempty intersection which includes some point in $[0,1]$, *i.e.*, $L \cap B \neq \emptyset \Longleftrightarrow [0,1] \bigcap_{i=1}^{d} I_i \neq \emptyset$.

Again, all "safety boxes" are bounding boxes which have been 2ϵ inflated. This approach is required as: (1) it is necessary to verify that the original line segment intersects with an ϵ-inflated bounding box, and (2) the line segment endpoints being tested may differ by as much as $\epsilon/2$ in each coordinate direction from the original line segment, resulting in a maximum difference between the two segments of between $\sqrt{3}/2\epsilon$ and ϵ.

Algorithm 5.3: Bounding box intersection query against the *sk*D-tree.

Input: d-dimensional tree elements T_i, $1 \leq i \leq N$
skD-tree consisting of $S_{1:2N}$ and $L_{1:2N}$
d-dimensional query bounding box B
Push $\{$node 1$\}$ onto the stack
Do until stack is empty
Pop $\{parent$ node$\}$ off the stack
If $B \cap (S_{parent} + \epsilon) \neq \emptyset$
If L_{parent} indicates a leaf
Find T_i pointed to by L_{parent}
Add $\{T_i\}$ to the collision list
Else
Add $\{L_{parent+0}\}$ onto the stack
Add $\{L_{parent+1}\}$ onto the stack
Output: Collision list of tree elements T_i

5.2 Bounding box intersection query against the *sk*D-tree

The next algorithm presented (Algorithm 5.3) uses the *sk*D-tree structure to represent a surface. This routine is designed to calculate possible intersections of the surface geometric elements with a d-dimensional box B. The algorithm returns a subset of leaves (*e.g.* triangles) that may contain an intersection with the box. The user must perform further tests on this subset to actually determine the intersection set. This algorithm is similar to Algorithm 5.2; the details of its operation will not be presented here.

5.3 Tree Intersection Query Against the *sk*D-tree

The final query algorithm (Algorithm 5.4) uses the *sk*D-tree structure to determine the intersection with another input *sk*D-tree, to determine the intersections of the geometric elements between the trees. Again, this algorithm returns a subset of leaves that are candidate objects in which an intersection may occur. The user must perform further tests on this subset to actually determine the intersection set. This algorithm is similar to Algorithm 5.2; further details of operation will not be presented here.

6 Applications

The *sk*D-tree has been employed by the authors in several applications to date. The applications range from simple tools that respond with point location information to a comprehensive computational geometry capability in the form of a query library that provides a parallel adaptive mesh generation (AMR) capability to a large physics simulation code. This paper will provide an implementation overview to the use of the *sk*D-tree in the

Algorithm 5.4: *sk*D-tree intersection query against *sk*D-tree.

Input: d_α-dimensional tree elements T_i^α, $1 \le i \le N_\alpha$
*sk*D-tree consisting of $S^\alpha_{1:2N_\alpha}$ and $L^\alpha_{1:2N_\alpha}$
Query Object: d_β-dimensional tree elements T_j^β, $1 \le j \le N_\beta$
*sk*D-tree consisting of $S^\beta_{1:2N_\beta}$ and $L^\beta_{1:2N_\beta}$
Push $\{\text{node } 1, T^\beta\}$ onto the stack
Do until stack is empty
  Pop $\{parent \text{ node}\}$ off the stack
  If $\{parent \text{ node}\} \cap T^\alpha \neq \emptyset$
    If L^β_{parent} indicates a leaf
      Find T_j^β pointed to by L^β_{parent}
      Add pair T_j^β and $\{parent \text{ node}\} \cap T^\alpha$ to the collision list
    Else
      Add $\{L^\beta_{parent+0}\}$ onto the stack
      Add $\{L^\beta_{parent+1}\}$ onto the stack
Output: Collision list of pairs T_j^β and $T_j^\beta \cap T^\alpha$

following contexts:

1. a curve-curve intersection algorithm
2. a curve-surface intersection algorithm
3. a point in-out test
4. a minimum distance algorithm
5. an AMR mesh generation algorithm

To prepare for the presentation of the above approaches, it is necessary to make some definitions. These applications are based on computing the degree of overlap of the safety box of the *current object* against the safety boxes of *target objects* that have been stored in the *sk*D-tree. To state this differently, one wishes to calculate the potential volume of intersection, if any, between an arbitrary object (actually, its safety box) and any objects that have been previously placed in the *sk*D-tree. The geometric form and representation of the objects are not critical to the discussion, but are important to consider during the actual implementation of the algorithms to follow. With this in mind, this paper will supply details of the implementation to allow other researchers to duplicate the approaches if desired.

6.1 Curve-curve intersection

The curve-curve intersection problem is defined as an approach to determine if the *current curve*, $C_o(s) \rightarrow [x_o(s), y_o(s), z_o(s)]$ intersects one or more *target curves* in $\mathbb{R}^3$, represented

as $C_n(s) \to [x_n(s), y_n(s), z_n(s)]$, where $n = 1,2,\cdots,N$. Further, if the current curve does intersect one or more of the target curves, the application requires that the algorithm return the target curves that were intersected, and information about the region of intersection $R(x,y,z)$.

The algorithm proceeds as follows:

1. Select the curves $C_n(s), n = 1,2,\cdots,N$ that occupy the target space.
2. Discretize the curves [13]. Depending on the complexity of each curve, the discretization process will result in one, or perhaps a large number of line segments that approximate the curve. One desires that the discretization process be sufficiently fine such that the points on the curve are no further than some δ away from the corresponding location on the approximating line segment.
3. Create a safety box data structure for each line segment approximating each curve. The safety box must be at least 2ϵ larger than the line segment that it is enclosing. The safety box data structure should also contain an index to associate it with the particular line segment that it contains and a pointer to the particular curve to which it belongs.
4. Construct the *sk*D-tree, by placing the safety boxes of the line segments into the target tree. When all the data is stored, the tree may be finalized and the query process may begin.
5. Discretize the current curve, $C_o(s)$, such that the curve is no further than δ away from its corresponding line segment(s). Create a collection of safety boxes that are at least 2ϵ larger than the line segments.
6. For each of these safety boxes, query the *sk*D-tree to determine if there is an intersection between the safety box and any of the safety boxes currently contained in the target tree.

With the safety boxes of the discrete segments of the target curves entered into the *sk*D-tree, it is possible to query the safety boxes of the discrete segments of the current curve against the tree. As discussed earlier in the paper, the query function will return indices for all the safety boxes in the *sk*D-tree that *potentially intersect* with the current segment used in the query. Further, assuming that there are m segments from the target curves in the *sk*D-tree, and n segments that make up the current curve (thus n queries are made to the *sk*D-tree), the algorithmic complexity of the global search is $\mathcal{O}(n \log m)$.

The query function returns indices for all safety boxes contained in the tree that have *any* measure of intersection with the query box. If there is zero overlap between the query box and the *sk*D-tree, the query function does not return any indices. At this point, one is certain that there cannot be any intersection of the actual geometric entities involved, given an intelligent choice of the ϵ value used to construct the safety boxes. This is a very useful result; given a curve in space, it is possible to determine conclusively that the curve in question does not intersect any of the curves represented in the *sk*D-tree in

$\mathcal{O}(n\log m)$ operations. Indeed, for the general problem of curve-curve intersection in $\mathbb{R}^3$, this is by far the most common outcome of a query.

The other possibility is that the n^{th} safety box intersects one or more of the safety boxes contained in the *sk*D-tree. In this case, the query function will return the indices of all of the k safety boxes in the tree that are intersected by the query. Given this information, it is up to the user of the *sk*D-tree library to use this information to determine if the portions of the curve represented by each of the k safety boxes actually intersect the curve segment represented by the n^{th} safety box.

The authors typically use a two step process to determine if the curves actually intersect, and if so, to obtain the actual point(s) of intersection. This algorithm is as follows:

1. Develop a *tighter* set of bounding volumes for the line segments contained in the n^{th} and k^{th} isothetic safety boxes.
2. Determine if there is any intersection between these "tighter" boxes.
3. If so, use some root-finding algorithm appropriate for the representation of the actual curves to locate the actual point(s) of intersection if there are any.

Consider that, in the general case, it is statistically rare that the curves actually overlap when overlap is indicated between the isothetic safety boxes. Thus, it is worthwhile to eliminate as many of these non-overlapping cases as possible before proceeding to the root-finding approach. In the case of the isothetic safety boxes approximating a line segment which, in turn, approximates the true curve, one may first consider the problem of determining if there is an intersection between more representative bounding volumes surrounding the line segments. Recall that the line segments are no further than δ away from the true curve. Thus, if the line segments are represented by a bounding cylinder with a radius of $\delta+\epsilon$, where ϵ is again the safety parameter, the curve must be fully contained within the cylinder. Indeed, if this is the case, one may consider the problem of determining if there is a region of intersection between the safety cylinders of the line segments approximated by the isothetic safety boxes. Furthermore, one only needs to pursue further analysis of the case where the safety cylinders overlap.

For this final case, where the safety cylinders overlap, one typically pursues a root-finding approach to locate the point(s) of intersection between the native forms of the two curves, if any exist. The authors typically use a Newton method to locate the intersection, but there are many other alternatives that may be more appropriate for the application in question. The authors acknowledge that Newton's method is problematic for the general root finding problem given both the general nature of the curves and the need to supply an *initial guess* for the method that is within the radius of convergence of Newton's Method. In this application, these theoretical limitations are not generally encountered in practice. Indeed, Newton's method is not used to locate the point(s) of intersection (if any exist) between the endpoints of two general curves; it is used only on rather short segments of the curves defined by the endpoints of the line segment used to approximate the curve locally. Further, over these small regions of the curve that the

line segment approximates, the curve is not truly general. Indeed, the curve is well-approximated (within a tolerance δ) by a linear approximation. Given this tight bound on the local behavior of the curve, and using the approximating line segment endpoints as an initial guess for the Newton method, convergence is typically rapidly and reliably achieved if a single root exists. Further, divergence typically indicates that a root does not exist in the region. These two results, however, do not provide an indication of the existence of multiple roots if that information is required by the application.

This completes the discussion of the mechanics of a curve-curve intersection algorithm in $\mathbb{R}^3$. There are several other applications that may be developed using this approach as a base concept. Indeed, the next application to be considered (curve-surface intersection) is a straightforward generalization of the method presented above.

6.2 Curve-surface intersection

The curve-surface intersection problem is defined as an approach to determine if the *current curve*,

$$C_o(s) \rightarrow [x_o(s), y_o(s), z_o(s)],$$

intersects one or more *target surfaces* in $\mathbb{R}^3$, represented as

$$S_n(s,t) \rightarrow [x_n(s,t), y_n(s,t), z_n(s,t)],$$

where $n = 1,2,\cdots,N$. Further, if the current curve does intersect one or more of the target surfaces, the application requires that the algorithm return the target surfaces that were intersected, and information about the region of intersection $R(x,y,z)$.

The algorithm proceeds as follows:

1. Select the surfaces $S_n(s,t)$, $n = 1,2,\cdots,N$ that occupy the target space.
2. Discretize the surfaces. Generally, the surfaces may be rather complex in nature, and may actually wrap around to share a common edge, forming a closed volume. Indeed, if the domain geometry is obtained from a commercial solid-modeling package or computer-aided design (CAD) tool, the domain geometry may be represented in the form of *trimmed* parametric surfaces. For the purposes of the query, the surfaces (within the trimming region) are often triangulated to form a discrete approximation. This process typically results in the creation of a large number of triangles approximating each surface. Again, one desires that the discretization process be sufficiently fine such that the furthest points on the actual surface are no further than some δ away from the corresponding location on the approximating triangle.
3. Create a safety box data structure for each triangle approximating each surface. The safety box must be at least 2ϵ larger than the triangle that it is enclosing. The safety box data structure should also contain an index to associate it with the particular triangle that it contains and a pointer to the particular surface to which it belongs.

4. Construct the *sk*D-tree, by placing the safety boxes of the triangles into the target tree. When all the data is stored, the tree may be finalized and the query process may begin.
5. Discretize the current curve, $C_o(s)$, such that the curve in no further than δ away from its corresponding line segment. Create a collection of safety boxes that are at least 2ϵ larger than the line segments.
6. For each of these safety boxes, query the *sk*D-tree to determine if there is an intersection between the safety box and any of the safety boxes currently contained in the target tree.

Aside from the details involving the surface discretization, it is clear that the curve-surface intersection algorithm is very similar to the curve-curve algorithm considered in the previous section. As such, a detailed discussion of this method will not be repeated here.

6.3 Point in/out test

The point in/out test is a query to determine if a particular domain coordinate (x,y,z) lies inside or outside of a body contained within the computational domain. Alternatively, this test may be viewed as a query to determine which "part" within the domain contains a particular location (if any).

The mechanics of this test is based on the particular representation of the domain geometry. Two models are possible:

1. If regions are defined in terms of boundary-represented parts that do not have an explicit relationship to some underlying mesh discretizing the domain, it is necessary to use the part definition directly to determine the status of the query.
2. If the parts contained within the domain are explicitly associated with a computational mesh (*e.g.* a body-fitted mesh), the mesh elements contained within the body could share the aspatial attributes associated with the body.

For the first case, only the basic part definition may be used to determine where a query point lies. The general problem, in this case, exists when the domain contains an arbitrary number of parts. Each part is assumed to be represented by some boundary definition; usually a list of surfaces (perhaps trimmed parametric surfaces) that completely enclose the volume of the part. Further, the part definition is assumed to be *airtight*; the surfaces fit perfectly at seams such that there are no gaps or overlaps between surfaces and the volume is completely closed.

There are two possible states in which the query point may exist: (1) it is contained within one of the parts (that part, in turn, may be contained in another, and so on), or (2) the location is outside of all the domain parts, in the interstitial region that separates the parts. In either case, the algorithm to compute the status of the point is as follows:

1. Build the *sk*D-tree by discretizing the surfaces contained in the computational model.
2. Beginning at the (x,y,z) location of the query point, cast a ray in an arbitrary direction outward to the boundary of the problem domain.
3. Decompose the ray into a set of line segments, build safety boxes for the segments, and intersect these safety boxes against the *sk*D-tree. The number of line segments used to decompose the ray depend on the direction of the ray and other aspects of the application.
4. Determine the actual distinct bounding surfaces intersected by the ray. If the ray intersects an even number of distinct part surfaces as it reaches the boundary of the problem, the query point (x,y,z) was not interior to any of the parts in the domain (it is in the interstitial region). If an odd number of surfaces were intersected, the query point (x,y,z) is located inside of the part whose surface was intersected by the ray an odd number of times.

As introduced above, the number of line segments used to decompose the ray is a based on the ray direction and other application requirements. Given a ray cast in an arbitrary direction, there is a compromise between the desire to use only a few (or perhaps only one) line segment to decompose the ray for simplicity, and the use of a large number of segments to minimize the number of collisions returned by the *sk*D-tree that will require further (and slower) processing. Indeed, consider the case of a query point $(0,0,0)$, with the ray cast out the first octant equidistant from each of the principal axes. Further, only one line segment will be used to decompose the ray, resulting in a safety box that is the first octant. The *sk*D-tree query against the surface safety boxes will return all boxes that intersect with the first octant. This would not be a very useful result. Given a random ray cast direction, it is typically better to decompose the ray into a large number of segments to reduce the number of collisions returned by the tree (and the amount of further processing that is required). Note, however, that if the ray is cast along one of the coordinate directions, only one segment is required, which would result in a long, narrow safety box along that axis. Indeed, this safety box is *almost* one-dimensional; it is of thickness 2ϵ on two of its three dimensions. Clearly, casting rays parallel to the coordinate axes often provides excellent economy for point topological query operations without sacrificing robustness or accuracy. Unfortunately, one is limited to six rays for each query; this may not be a sufficient number of "random" raycasts for every application.

In a typical application, there are two issues that complicate this strategy. First, the boundary-represented parts are generally not airtight. Indeed, requiring the model to be airtight is not reasonable nor achievable, given that solid models are constructed on a computer. Secondly, the ray cast may intersect the surface at one point, as it passes through the surface moving from one part to another. The ray may encounter a large number (theoretically infinite) number of points if it encounters the surface nearly tangent and travels tangent to the surface for any distance. Additionally, the ray may only encounter the surface once, exactly tangent, and emerge back into the same part from

which it came. Given these issues, an implementation needs to cast many rays, in quasi-random directions, to statistically determine the number of surfaces that are encountered as the ray proceeds to the boundary from the point of origin (x,y,z). Typically, on the order of ten rays are needed to stochastically-determine the status of the query point.

The second form of model representation is one where the parts are associated explicitly with a computational mesh; typically called a body-fitted discretization. This problem is significantly less complex; it is only necessary to locate which mesh element contains the query point to know which part it is located in (or know that it is within the interstitial region). Such a spatial query requires a point-in-polyhedra query test. To locate the query point in this case, the following steps are used:

1. Load the *sk*D-tree with safety boxes enclosing the mesh elements.
2. Intersect the one dimensional query point against the *sk*D-tree.
3. Determine which element the query point lies in, given the set of safety boxes returned by the query.

Note that a safety box around the query point is not necessary in this application, as the query point is a one dimensional object that will lie in the safety box of one of the tree elements if there is any possibility that the point lies within an object contained within the tree.

This application requires a general spatial search if there is no a-priori knowledge about the spatial relationship of subsequent query locations (x,y,z). Often, these query locations may be spatially-related; for example, the first query may be a general query, but subsequent queries may involve points associated with the initial query location (i.e., they may be the endpoint of a line that begins with the initial query point). In this case, the use of an *Eulerian Walk* [1] may be more efficient than the *sk*D-tree for subsequent queries.

6.4 Minimum distance algorithm

The minimum distance query algorithm was discussed in detail earlier in this paper in Section 4. Many computational applications are interested in determining the closest boundary point to an interior location. The steps for using the minimum distance query algorithm are:

1. Discretize the boundary surfaces (or objects) of interest. Typically, the surfaces will be triangulated, but the actual form of the discretization is not important.
2. Construct safety boxes for the boundary objects (*i.e.* triangles).
3. Load the safety boxes into the *sk*D-tree.
4. Construct a safety box around the query location. Query the tree, specifying a reference point if desired.
5. Locate the closest point using the safety boxes returned in the collision list.

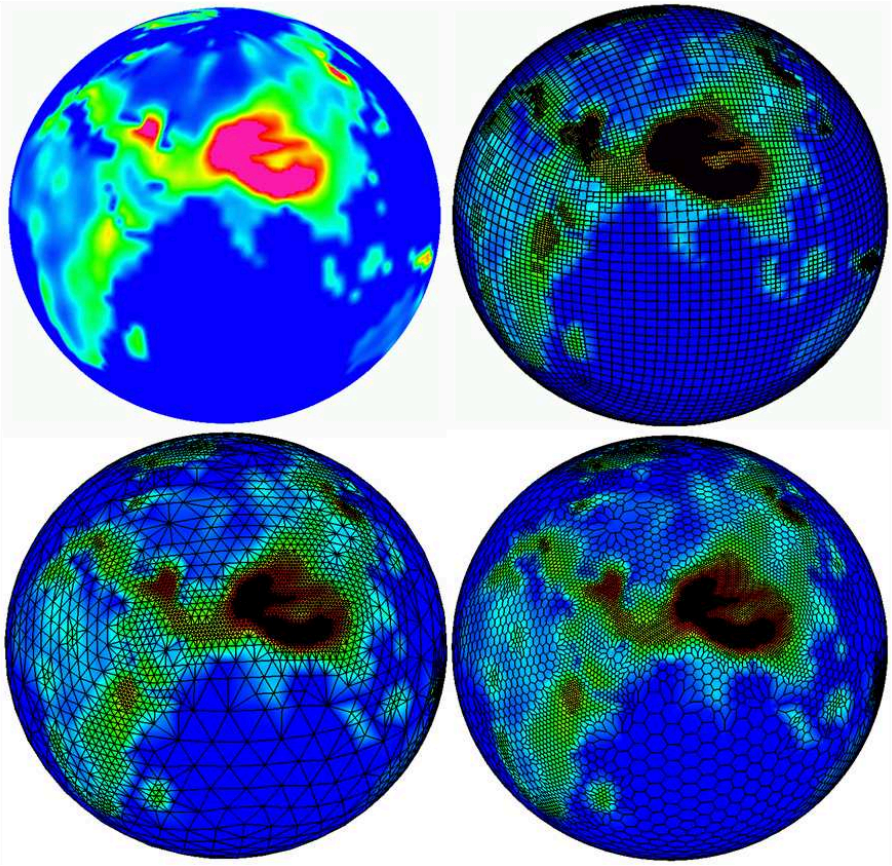

Figure 1: Example of the use of h-refined adaptive hybrid mesh generation for climate modeling. Note how refinement is used to capture altitude gradient detail in the region of the Alps and Himalayas mountains on this earth surface model shown in the upper-left figure. Moving clockwise, the upper-right figure shows an icosahedral mesh, the lower-right is a polygonal element mesh, followed by a triangle surface mesh in the lower-left figure.

6.5 Adaptive mesh refinement (AMR)

The adaptive mesh is growing in popularity for the simulation of complex multiphysics problems due to its ability to capture fine geometric detail in the problem specification simultaneously with resolving various length scales in the evolving physics simulation. Further, the tree-based adaptive mesh refinement method (AMR) is often automatic; the algorithm may be posed in such a way that it always terminates to yield a finished mesh. The same cannot be said for other mesh generation methods.

The AMR method may also be used to *refine* an existing mesh to capture fine geometric or solution detail present in the simulation. Fig. 1 illustrates the use of this approach to refine a mesh based on the height variation of mountains above a reference datum in an earth surface model.

The AMR approach begins with the definition of the basic mesh that spans the com-

putational domain. This mesh may be as simple as the user specifying a level of refinement in each of the three coordinate directions, along with the domain length, width, and height. A uniform, rectilinear base mesh can be generated directly from this information. Secondly, the user supplies a geometric problem statement, which details the components that define the simulation to be performed. The geometric model may be as complex as a complete CAD model of a detailed assembly. Lastly, the user supplies an initial statement of the physics state, which may include things like temperature profiles, the location of shock waves, or other solution features. More typically, however, the user just specifies the initial and boundary conditions for the simulation to begin at some initial state. The AMR method uses both the geometry statement, and any solution features (if supplied), to determine where AMR refinement must occur to capture the length scales important to the problem. The *sk*D-tree may be used quite effectively for the initial refinement calculation to match the geometric detail of interest. One approach is as follows:

1. Loop over all the parts of the geometric assembly, discretizing the surfaces (or other geometric objects). Construct safety boxes for these discrete elements, and insert them into the *sk*D-tree.
2. Loop over all the basic mesh elements. Treat each (axis-aligned) element as a safety box; intersect each of these boxes in turn with the *sk*D-tree.
3. If there is a potential intersection with one of the geometric objects in the tree and the mesh element, at least one collision will be returned. One could test at this point if there is truly an intersection between the mesh element and the geometry.
4. If there is an intersection, refine that particular mesh element. If there is not an intersection, select the next basic mesh element and repeat.
5. If there was an intersection, above, create a new set of safety boxes that represent the refined octants of the original base element. Intersect these with the *sk*D-tree to determine which of these to refine further. Continue to recurse in this manner until the maximum level of refinement is achieved (this is a user-supplied value).

This algorithm may be further improved by reconsidering the third step of the process, above. With some loss of rigor, the authors propose that if any collisions are returned from the mesh element/geometry intersection process, that the element be blindly subdivided in lieu of performing further (and more expensive) analysis to determine if the geometry actually bisects the element. Clearly, this approach will result in some excessive refinement at geometric interfaces, but it will save much time in more detailed analysis. The run-time penalty incurred by this refinement is typically much smaller than that required to perform the more detailed testing inherent in a higher precision intersection algorithm. Further, as the simulation using the adaptive mesh algorithm evolves, refinement will occur near geometric interfaces anyway due to boundary-layer resolution. Thus, any run-time penalty due to excessive refinement is typically only of concern for the first few time cycles of a simulation.

6.6 Multi-physics solution data transfer (remapping)

In addition to applications in mesh generation and refinement, the *sk*D-tree may be effectively applied to mapping a physics solution present on one mesh to another, topologically-distinct mesh. It is a common requirement in multiphysics and adaptive analysis applications to transfer solution fields and their derivatives between meshes and/or to other applications. This process requires the evaluation of field variables and their derivatives at particular target locations, based on data associated with the *donor* or source mesh. For each target location, the solution transfer process must (i) identify the mesh entities in the donor mesh from which data is required; (ii) determine the local coordinates of the target point on the donor mesh entities; and (iii) evaluate the required field components on the donor mesh at that location. In some cases, the field may be preprocessed on the donor mesh to improve the accuracy of this data, and/or the level of continuity between mesh entities. Solution transfer is performed by integrating aspects of geometry (mesh) sorting and searching, interpolation of kernels (fields), and various implementations of geometry, mesh, and field interfaces. As an aside, this solution transfer process is often called a *remapping* of the physics field data from one distinct mesh to another.

The literature is rich with discussion of methods to perform remapping; ranging from discussions of geometric approaches to calculating intersection volumes between the donor and target meshes, to approaches of accurately computing the data transfer integration process given these intersection regions. It is immediately clear that one must exercise care in hosting data from one mesh upon another; one should conserve the transferred physical quantities during the transfer process. Li [14] proposes a remapping method for multiphysics applications where the target mesh is topologically related to the donor. In this approach, both meshes share a base discretization level, where the donor and target meshes differ in the amount of inter-element refinement of this base connectivity reference. Li presents both solution criteria and geometric approaches used to implement the algorithm, along with several examples.

A second class of methods arise when the donor and target meshes are not obviously related; i.e., they are topologically distinct. Dukowicz [15], Ramshaw [16], and Miller *et.al.* [1] discuss approaches based on mapping of intensive physical values $q(\mathbf{x})$ from a donor mesh to a new field $q'(\mathbf{x})$ on the target mesh using an integration,

$$q'_k(\mathbf{x}) = \frac{1}{A_k}\int_{A_k} q(\mathbf{x})\,dA,$$

where k is the element index of the target element, and A_k is the area of the target element.

Jiao and Heath [17] provide a detailed comparison of several popular one- and two-dimensional remapping approaches on topologically-distinct donor and target meshes. They propose a method based on an intersection (common refinement) between the two meshes, and compare this to other methods. All of these methods require determining the geometric overlap (intersection) region between donor and target elements.

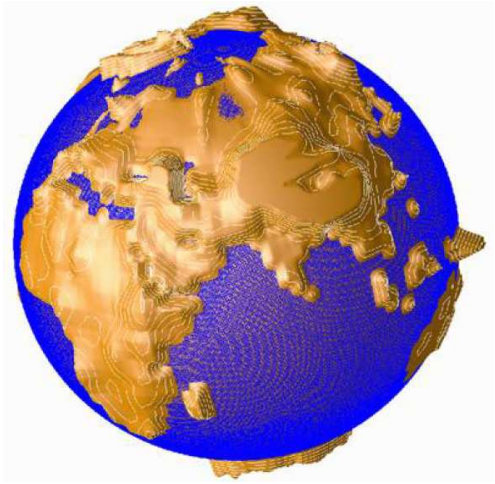

Figure 2: An orographic map provides an indication of the average height of land. This illustration depicts a contoured orographic relief of the Earth's surface.

When the meshes are truly distinct, a general search is required to locate the position of a selected object of the donor mesh in the space of the target, or vice versa. Typically, however, locality of data will allow an algorithm such as an Eulerian walk to be used in the integration method employed as part of the remap algorithm. Exceptions to this occur when an application requires the mapping of data with no clear spatial relationship to a target mesh. An example of this is the mapping of orographic field data (discrete altitude data) as shown in Fig. 2 to a distinct mesh to drive an *r*-adaptive mesh motion algorithm.

Fig. 3 shows a computational mesh generated on the Earth surface model. In this illustration, there is no correspondence between the orography field data and the position or concentration of node points of the mesh; the mesh is equidistributed across the globe. The process of *r*-adaptation involves movement of the mesh nodes towards regions where additional refinement is desired to capture the orographic field shown by the coloration of the globe. Such *r*-adaptive algorithms require the transfer of solution data (in this case represented by the orographic field) to the nodes of the computational mesh. For this example, the *sk*D-tree is used to map the data onto the mesh. The mapping process is described by the following algorithm:

1. The orography field data is stored in the *sk*D-tree, using Algorithm 3.1.
2. For each element in the mesh, a query of the tree is performed, which returns all the field values that are contained within the element.
3. The mean orographic value for the element is calculated as the average of the values obtained from the query.
4. The *r*-adaptive algorithm is applied to move the mesh node points based on the mean orographic data within the elements.

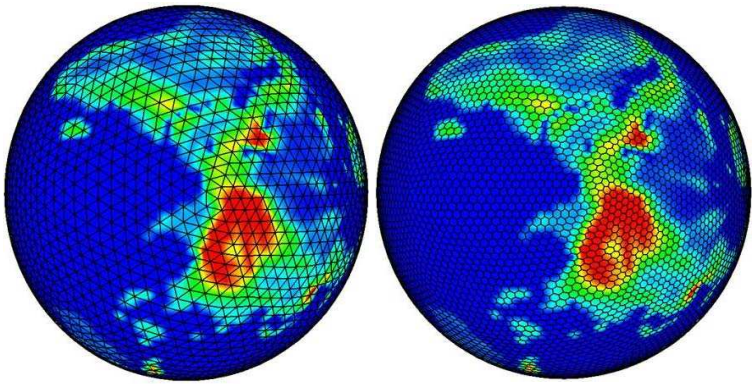

Figure 3: Examples of triangle and polygonal surface meshes superimposed on the Earth's surface. Globe coloration indicates orography.

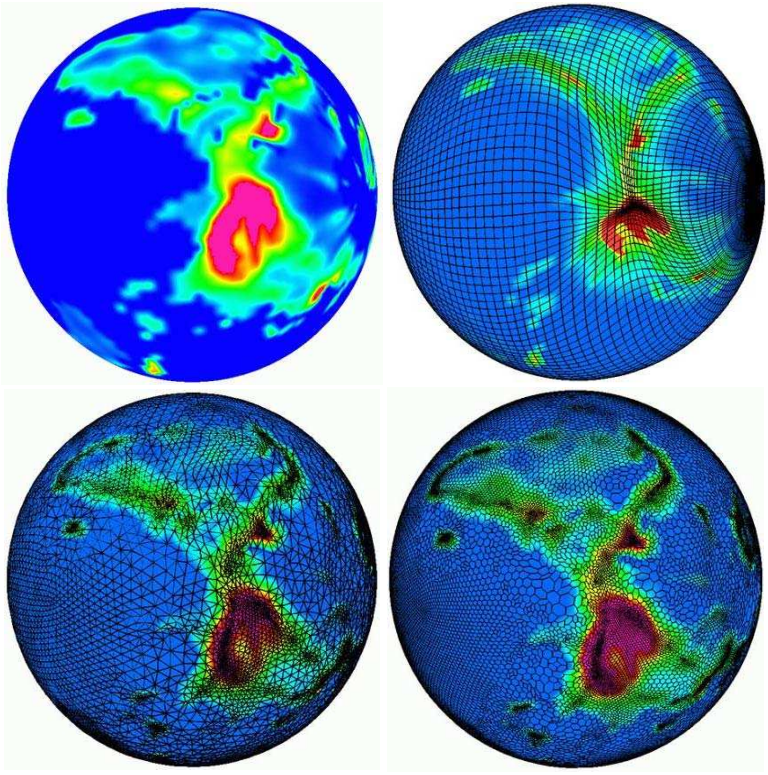

Figure 4: r-adapted surface mesh using the skD-tree spatial query to map discrete orography data to the mesh as it adapts to the data. Clockwise from the orography data shown in the upper-left figure is an adapted structured quadrilateral mesh, then a polygonal mesh, then finally an adapted triangular mesh.

5. As the elements now overlap with different orographic values (the element has moved as a result of the previous operation), it is necessary to refresh the mean orographic value for each element. Proceed back to step 2, and repeat until the algorithm converges.

Convergence of this algorithm results in the adapted meshes shown in Fig. 4. Note how the adapted mesh has moved to the areas of the globe where the change in elevation is the greatest. In this example, the *sk*D-tree provides an effective mechanism to transfer the orography data onto the evolving surface mesh to drive the *r*-adaptive process. This mapping mechanism may be readily applied to other solution transfer problems of a similar nature. Note that, although this is a dynamic application, the *sk*D-tree is formed only once. In this case, the orographic data is static but the mesh moves "across it" (as convergence proceeds, the elemental mean orographic value changes as the element moves with respect to the data). Each time the mesh moves, the bounding boxes that describe the elements must be re-constructed and queried against the static tree.

These example applications indicate the utility of searching and query using the *sk*D-tree. There are many other applications that could take advantage of efficient spatial search algorithms. To list a few possibilities:

- Graphics, interfaces, and scientific visualization - use of the mouse for picking and object selection, and collision detection.
- Point status - various point location and query activities, implementing particle tracers, point probe, and line probe functions. Providing arbitrary polyhedral cell support in particle tracer, line probe, and point probe methods.
- Geometric processing - intersection, projection, computational topology, and ray casting.
- Grid generation: point in body, AMR, volume fraction calculation, advancing front, duplicate feature removal, material interface resolution (cell splitting), and grid transformation.
- Other physics applications - Monte Carlo, smooth particle hydrodynamics (SPH), hydrodynamic front collision, and front propagation methods.

7 Conclusions

This paper develops a set of algorithms potentially useful for spatial searching in various computational physics applications. The data structure that is the basis for this work is an *sk*D-tree, which is a member of a family of methods based on bounding volume hierarchies (BVHs). The *sk*D-tree was introduced for its rapid general search behavior, its simplicity, and the ease with which it may be implemented and linked to various applications. However, one must carefully weigh the limitations of any method; the *sk*D-tree is best used with static data for general queries. If the target query set changes

during the query process, the *sk*D-tree is likely not the best choice. Further, if the search may be localized or if previous state information is known, a general search method like the *sk*D-tree may not be the most effective strategy for the particular application.

This work presents and discusses several algorithms that may be used to construct the tree, and subsequently query the tree for potential collisions. The collisions are only candidates for intersection; further (user-defined) processing is usually required to be certain of an intersection and learn of its true character. If zero candidates are returned, the user may be certain that an intersection does not exist if the strategy for construction of the "safety boxes" was followed correctly.

Much of this paper focused on discussing the use of the *sk*D-tree and various query algorithms for applications that support scientific simulation. Several searching and proximity query applications were discussed, along with the use of query for mesh generation and solution transfer (remapping). These are only a small selection of applications that can possibly use a general, efficient spatial search mechanism to good effect.

Acknowledgments

The submitted manuscript has been authored by contractors of the U.S. Government under Contract Nos. DE-AC05-00OR22725 and DE-AC07-05ID14517. Accordingly, the U.S. Government retains a non-exclusive, royalty-free license to publish or reproduce the published form of this contribution, or allow others to do so, for U.S. Government purposes.

References

[1] D. S. Miller, D. E. Burton and J. S. Oliviera, Efficient second order remapping on arbitrary two dimensional meshes, Technical Report UCRL-ID-123530, Lawrence Livermore National Laboratory, 1996.
[2] E. Larsen, S. Gottschalk, M. C. Lin and D. Manocha, Fast proximity queries with swept sphere volumes, Technical Report TR99-018, University of North Carolina, Chapel Hill, 1999.
[3] S. Gottschalk, M. C. Lin and D. Manocha, OBBTree: A hierarchical structure for rapid interference detection, in: Proceedings of ACM SIGGRAPH '96, 1996, pp. 171-180.
[4] P. M. Hubbard, Approximating polyhedra with spheres for time-critical collision detection, ACM T. Graphics, 15(3) (1996), 179-210.
[5] P. G. Xavier, A generic algorithm for constructing hierarchical representations of geometric objects, in: Proc. IEEE ICRA 1996, vol. 4, Minneapolis, 1996, pp. 3644-3651.
[6] B. C. Ooi, Efficient query processing in geographic information systems, in: Lecture Notes in Computer Science, vol. 471, Spring-Verlag, 1990.
[7] J. L. Bentley, Multidimensional binary search trees used for associative searching, Commun. ACM, 18 (1975), 509-517.
[8] P. M. Hubbard, Collision detection for interactive graphics applications, PhD thesis, Department of Computer Science, Brown University, 1995.

[9] T. Sellis, N. Roussopoulos and C. Faloutsos, The R^+-tree: A dynamic index for multi-dimensional objects, in: Proc. 13th Int. Conf. Very Large Data Bases, Brighton, 1987, pp. 507-518.
[10] D. E. Knuth, The Art of Computer Programming, vol. 3, 2 ed., Addison-Wesley, Reading, 1998.
[11] R. L. Floyd and R. W. Rivest, Expected time bounds for selection, Commun. ACM, 18(3) (1975), 165-172.
[12] C. A. R. Hoare, Quicksort, Comput. J., 5(1) (1962), 10-15.
[13] A. Khamayseh and A. Kuprat, Hybrid curve point distribution algorithms, SIAM J. Sci. Comput., 23 (2002), 1464-1484.
[14] R. Li, On multi-mesh h-adaptive methods, J. Sci. Comput., 24(3) (2005), 321-341.
[15] J. K. Dukowicz, Conservative rezoning (remapping) for general quadrilateral meshes, J. Comput. Phys., 54 (1984), 411-424.
[16] J. D. Ramshaw, Conservative rezoning algorithm for generalized two-dimensional meshes, J. Comput. Phys., 59 (1985), 193-199.
[17] X. Jiao and M. T. Heath, Common-refinement-based data transfer between non-matching meshes in multiphysics simulations, Int. J. Numer. Meth. Engrg., 61 (2004), 2402-2427.