Best Journal Paper of 2007 Award

Available online at www.sciencedirect.com

ScienceDirect

Computers & Graphics 31 (2007) 157–174

COMPUTERS & GRAPHICS

www.elsevier.com/locate/cag

A fast all nearest neighbor algorithm for applications involving large point-clouds

Jagan Sankaranarayanan*, Hanan Samet, Amitabh Varshney

Department of Computer Science, Center for Automation Research, Institute for Advanced Computer Studies, University of Maryland, College Park, MD - 20742, USA

Abstract

Algorithms that use point-cloud models make heavy use of the neighborhoods of the points. These neighborhoods are used to compute the surface normals for each point, mollification, and noise removal. All of these primitive operations require the seemingly repetitive process of finding the k nearest neighbors (kNNs) of each point. These algorithms are primarily designed to run in main memory. However, rapid advances in scanning technologies have made available point-cloud models that are too large to fit in the main memory of a computer. This calls for more efficient methods of computing the kNNs of a large collection of points many of which are already in close proximity. A fast kNN algorithm is presented that makes use of the locality of successive points whose k nearest neighbors are sought to reduce significantly the time needed to compute the neighborhood needed for the primitive operation as well as enable it to operate in an environment where the data is on disk. Results of experiments demonstrate an *order* of magnitude improvement in the *time* to perform the algorithm and *several orders* of magnitude improvement in *work efficiency* when compared with several prominent existing methods.
© 2006 Elsevier Ltd. All rights reserved.

MSC: 68U05

Keywords: Neighbor finding; k Nearest neighbors; kNN Algorithm; All nearest neighbor algorithm; Incremental neighbor finding algorithm; Locality; Neighborhood; Disk-based data structures; Point-cloud operations; Point-cloud graphics

1. Introduction

In recent years there has been a marked shift from the use of triangles to the use of points as object modeling primitives in computer graphics applications (e.g., [1–5]). A point model (often referred to as a *point-cloud*) usually contains millions of points. Improved scanning technologies [4] have resulted in enabling even larger objects to be scanned into point-clouds. Note that a point-cloud is nothing more than a collection of scanned points and may not even contain any topological information. However, most of the topological information can be deduced by applying suitable algorithms on the point-clouds. Some of the fundamental operations performed on a freshly scanned point-cloud include the computation of *surface normals* in order to be able to illuminate the scanned object, applications of *noise-filters* to remove any residual noise from the scanning process, and tools that change the *sampling rate* of the point model to the desired level. What is common to all three of these operations is that they work by computing the k nearest neighbors for each point in the point-cloud. There are two important distinctions from other applications where the computation of neighbors is required. First of all, neighbors need to be computed for all points in the data set, potentially this task can be optimized. Second, no assumption can be made about the size of the data set. In this paper, we focus on a solution to the *k-nearest-neighbor* (kNN) problem, also known as the all-points kNN problem, which takes a point-cloud data set R as an input and computes the kNN for each point in R.

*Corresponding author. Tel.: +1 301 405 1769; fax: +1 301 314 9115.
E-mail addresses: jagan@cs.umd.edu (J. Sankaranarayanan), hjs@cs.umd.edu (H. Samet), varshney@cs.umd.edu (A. Varshney).
URLs: http://www.cs.umd.edu/~jagan, http://www.cs.umd.edu/~hjs, http://www.cs.umd.edu/~varshney.

0097-8493/$ - see front matter © 2006 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cag.2006.11.011

We start by comparing and contrasting our work with the related work of Clarkson [6] and Vaidya [7]. Clarkson proposed an $O(n\log\delta)$ algorithm for computing the nearest neighbor to each of n points in a data set S, where δ is the ratio of the diameter of S and the distance between the closest pair of points in S. Clarkson uses a PR quadtree (e.g., see [8]) Q on the points in S. The running time of his algorithm depends on the depth $d=\delta$ of Q. This dependence on the depth is removed by Vaidya who proposed using a hierarchy of boxes, termed a Box tree, to compute the *kNN*s to each of the n points in S in $O(kn\log n)$ time. There are two key differences between our algorithm and those of Clarkson and Vaidya. First of all, our algorithm can work on most disk-based (out of core) data structures regardless of whether they are based on a regular decomposition of the underlying space such as a quadtree [8] or on object hierarchies such as an R-tree [9]. In contrast to our algorithm, the methods of Clarkson and Vaidya have only been applied to memory-based (i.e., incore) data structures such as the PR quadtree and Box tree, respectively. Consequently, their approaches are limited by the amount of physical memory present in the computer on which they are executed. The second difference is that it is easy to modify our algorithm to produce nearest neighbors incrementally, i.e., we are able to provide a variable number of nearest neighbors to each point in S depending on a condition, which is specified at run-time. The incremental behavior has important applications in computer graphics. For example, the number of neighbors used in computing the normal to a point in a point-cloud can be made to depend on the curvature of a point.

The development of efficient algorithms for finding the nearest neighbors for a single point or a small collection of points has been an active area of research [10,11]. The most prominent neighbor finding algorithms are variants of depth-first search (DFS) [11] or best-first search (BFS) [10] methods to compute neighbors. Both algorithms make use of a search hierarchy which is a spatial data-structure such as an R-tree [9] or a variant of a quadtree or octree (e.g., [8]). The DFS algorithm, also known as branch-and-bound, traverses the elements of the search hierarchy in a predefined order and keeps track of the closest objects to the query point that have been encountered. On the other hand, the BFS algorithm traverses the elements of the search hierarchy in an order defined by their distance from the query point. The BFS algorithm that we use [10], stores both points and *blocks* in a priority queue. It retrieves points in an increasing order of their distance from the query point. This algorithm is *incremental* as the number of nearest neighbors k need not be known in advance. Successive neighbors are obtained as points are removed from the priority queue. A brute force method to perform the *kNN* algorithm would be to compute the distance between every pair of points in the data set and then to choose the top k results for each point. Alternatively, we also observe that repeated application of a neighbor finding technique [12] on each point in the data set also amounts to performing a *kNN* algorithm. However, like the brute-force method, such an algorithm performs wasteful repeated work as points in proximity share neighbors and ideally it is desirable to avoid recomputing these neighbors.

Some of the work entailed in computing the *kNN*s can be reduced by making use of the approximate nearest neighbors (ANNs) [12]. In this case, the approximation is achieved by making use of an error-bound ε which restricts the ratio of the distance from the query point q to an approximate neighbor and the distance to the actual neighbor to be within $1+\varepsilon$. When used in the context of a point-cloud algorithm, this method may lead to inaccuracies in the final result. In particular, point-cloud algorithms that determine local surface properties by analyzing the points in the neighborhood may be sensitive to such inaccuracies. For example, such problems can arise in algorithms for computing normals, estimating local curvature, as well as sampling rate and local point-cloud operators such as *noise-filtering* [3,13], *mollification* and removal of *outliers* [14]. In general, the correct computation of neighbors is important in two main classes of point-cloud algorithms: algorithms that identify or compute properties that are common to all of the points in the neighborhood, and algorithms that study variations of some of these properties.

An important consideration when dealing with point models that is often ignored is the size of the point-cloud data sets. The models are scanned at a *high fidelity* in order to create an illusion of a smooth surface. The resultant point models can be on the order of several millions of points in size. Existing algorithms such as normal computation [15] which make use of the suite of algorithms and data structures in the ANN library [12] are limited by the amount of physical memory present in the computer on which they are executed. This is because the ANN library makes use of in-core data structures such as the *k-d* tree [16] and the BBD-tree [17]. As larger objects are being converted to point models, there is a need to examine neighborhood finding techniques that work with data that is out of core and thus out-of-core data structures should be used. Of course, although the drawback of out-of-core methods is the incurrence of I/O costs, thereby reducing their attractiveness for real-time processing, the fact that most of the techniques that involve point-clouds are performed *offline* mitigates this drawback.

There has been a considerable amount of work on efficient disk-based nearest neighbor finding methods [10,11,18]. Recently, there has also been some work on the *kNN* algorithm [18,19]. In particular, the algorithm by Böhm [19], termed MuX uses the DFS algorithm to compute the neighborhoods of one block, say b, at a time (i.e., it computes the *kNN*s of all points in b before proceeding to compute the *kNN*s in other blocks) by maintaining and updating a best set of neighbors for each point in the block as the algorithm progresses. The rationale is that this will minimize disk I/O as the *KNN*s of points in the same block are likely to be in the same set of blocks. The GORDER method [18] takes a slightly different approach in that although it was originally designed for high-dimensional data-points (e.g., similarity retrieval in image processing applications), it can also be applied to low-dimensional data sets. In particular, this algorithm first performs a principal component analysis (PCA) to determine the first few dominant directions in the data space and then all of the objects are projected to this dimensionally reduced space, thereby resulting in drastic reduction in the dimensionality of the point data set. The resulting blocks are organized using a regular grid, and, at this point, a *kNN* algorithm is performed which is really a sequential search of the blocks.

Even though both the GORDER [18] and the MuX [19] methods compute the neighborhood of all points in a block before proceeding to process points in another block, each point in the block keeps track of its *kNN*s encountered thus far. Thus this work is performed independently and in isolation by each point with no reuse of neighbors of one point as neighbors of a point in spatial proximity. Instead, in our approach we identify a region in *space* that contains all of the *kNN*s of a *collection* of points (the space is termed *locality*). Once the best possible *locality* is built, each point searches only the locality for the correct set of k nearest neighbors. This results in large savings. Also, our method makes no assumption about the size of the data set or the sampling-rate of the data. Experiments (Section 6) show that our algorithm is faster than both the GORDER and the MuX methods and performs substantially fewer distance computations.

The rest of the paper is organized as follows. Section 2 defines the concepts that we use and provides a high level description of our algorithm. Section 3 describes the *locality* building process for *blocks*. Section 4 describes an incremental variant of our *kNN* algorithm, while Section 5 describes a non-incremental variant of our *kNN* algorithm. Section 6 presents the results of our experiments, while Section 7 discusses related applications that can benefit from the use of our algorithm. Finally, concluding remarks are drawn in Section 8.

2. Preliminaries

In this paper we assume that the data consists of points in a multi-dimensional space and that they are represented by a hierarchical spatial data structure. Our algorithm makes use of a disk-based quadtree variant that recursively decomposes the underlying space into blocks until the number of points in a block is less than some *bucket capacity B* [8]. In fact, any other hierarchical spatial data structure could be used including some that are based on object hierarchies such as the R-tree [9]. The blocks are represented as nodes in a tree access structure which enables point query searching in time proportional to the logarithm of the width of the underlying space. The tree contains two types of nodes: leaf and non-leaf. Each non-leaf node has at most 2^d non-empty *children*, where d corresponds to the *dimension* of the underlying space. A *child* node occupies a region in space that is fully contained in its parent node. Each leaf node contains a pointer to a *bucket* that stores at most B points. The *root* of the tree is a special block that corresponds to the entire underlying space which contains the data set. While the blocks of the access structure are stored in main-memory, the buckets that contain the points are stored on disk. In our implementation, a count is maintained of the number of points that are contained within the *subtree* of which the corresponding block b is the root and a *minimum bounding box* of the space occupied by the points that b contains.

We use the Euclidean metric (L_2) for computing distances. It is easy to modify our *kNN* algorithm to accommodate other distance metrics. Our implementation makes extensive use of the two distance estimates MINDIST and MAXDIST (Fig. 1). Given two blocks q and s, the procedure MINDIST(q, s) computes the minimum possible distance between a point in q to a point in s. When a list of blocks is ordered by their MINDIST value with respect to a reference block or a point, the ordering is called a MINDIST ordering. Given two blocks q and s, the procedure MAXDIST(q, s) computes the maximum possible distance between a point in q to a point in s. When a list of blocks is ordered by their An ordering based on MAXDIST is called a MAXDIST ordering. The *kNN* algorithm identifies the k nearest neighbors for each point in the data set. We refer to the set of *kNN*s of a point p as the *neighborhood* of p. While the neighborhood is used in the context of points, *locality* defines a neighborhood of blocks. Intuitively, the locality of a block b is the region in space that contains all the *kNN*s of all points in b. We make one other distinction between the concepts of neighborhood and locality. In particular, while neighborhoods contain no other points other than the *kNN*s locality is more of an approximation and thus the locality of a block b may contain points that do not belong to the neighborhood of any of the points contained within b.

Our algorithm has the following high-level structure. It first builds the locality for a block and later searches the locality to construct a neighborhood for each point contained within the block. The pseudo-code presented in Algorithm 1 explains the high level workings of the *kNN* algorithm. Lines 1–2 compute the *locality* of the blocks in the search hierarchy Q on the input point-cloud.

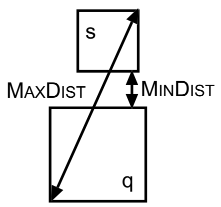

Fig. 1. Example illustrating the values of the MINDIST and MAXDIST distance estimates for blocks q and b.

Lines 3–4 build a neighborhood for each point in b using the locality of b.

Algorithm 1
Procedure $kNN[Q, k]$
Input: Q is the search hierarchy on the input point-cloud
(∗ high-level description of kNN algorithm ∗)

```
1.  for each block $b$ in $Q$ do
2.      Build locality $S$ for $b$ in $Q$
3.      for each point $p$ in $b$ do
4.          Build neighborhood of $p$ using $S$ and $k$
5.      end-for
6.  end-for
7.  return
```

3. Building the locality of a block

As the locality defines a region in space, we need a measure that defines the extent of the locality. Given a *query block*, such a measure would implicitly determine if a point or a block belongs to the locality. We specify the extent of a locality by a distance-based measure that we call PruneDist. All points and blocks whose distance from the query block is less than PruneDist belong to the locality. The challenge in building localities is to find a good estimate for PruneDist. Finding the smallest possible value of PruneDist requires that we examine every point which defeats the purpose of our algorithm which is why we resort to estimating it.

We proceed as follows. Assume that the query block q is in the vicinity of other blocks of various sizes. We want to find a set of blocks so that the total number of points that they contain is at least k, while keeping PruneDist as small as possible. We do this by processing the blocks in increasing order of their MaxDist order from q and adding them to the locality. In particular, we sum the counts of the number of points in the blocks until the total number of points in the blocks that have been encountered exceeds k and record the current value of MaxDist as the value of PruneDist. At this point, we process the remaining blocks according to their MinDist order from q and add them to the locality until encountering a block b whose MinDist value exceeds PruneDist. All remaining blocks need not be examined further and are inserted into list PrunedList. Note that an alternative approach would be to initially process the blocks in MinDist order, adding them to the locality, and set PruneDist be the maximum MaxDist value encountered so far and halting once the sum of the counts is greater than k to prune every block whose MinDist value is greater than PruneDist. This approach does not yield as tight an estimate for PruneDist as can be seen in the example in Fig. 2.

The pseudo-code for obtaining the locality of a block is given in Algorithm 2. The inputs to the BuildLocality algorithm are the query block q, a set of blocks Q corresponding to the partition of the underlying space into a set of blocks, and the value of k . Using these inputs, the algorithm computes the locality S of q. The while-loop in lines 1–7 visits blocks in Q in an increasing MaxDist ordering from q and adds them to S. The loop terminates when k or more points have been added to S, at which point the value of PruneDist is known. Lines 8–14 of the algorithm now add blocks in Q to S, whose MinDist to q is lesser than the PruneDist value. Line 17 returns the locality S of q, a set PrunedList of blocks in Q that does not belong to S, and the value of PruneDist.

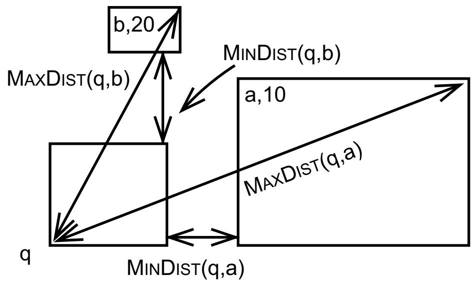

Fig. 2. Query block q in the vicinity of two other blocks a and b containing 10 and 20 points, respectively. When k is 10, choosing a with a smaller MinDist value does not provide the lowest possible PruneDist bound.

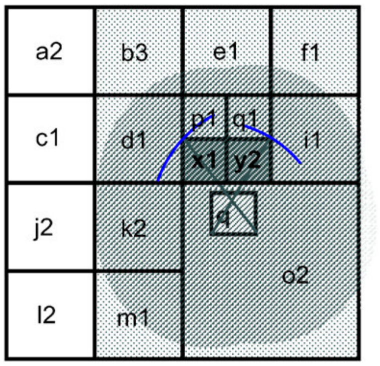

Fig. 3. Illustration of the workings of the BuildLocality algorithm. The labeling scheme assigns each block a label concatenated with the number of points that it contains. q is the query block. Blocks x and y are selected based on the value of MaxDist, while blocks b, e, f, i, d, p, q, k, m, and o are also selected as their MinDist value from $q \leqslant$ PruneDist.

The mechanics of the algorithm are illustrated in Fig. 3. The figure shows q in the vicinity of several other blocks. Each block is labeled with a letter and the number of points that it contains. For example, suppose that $k = 3$, and let Q = {a, b, c, d, e, f, i, j, k, l, m, o, p, q, x, y} be a decomposition of the underlying space into a set of blocks. The algorithm first visits blocks in a MaxDist ordering from q, until 3 points are found. That is, the algorithm adds blocks x and y to S and PruneDist is set to MaxDist(q, y). We now choose all blocks whose MinDist from q is less than PruneDist resulting in blocks b, e, f, i, d, p, q, k, m, and o being added to S.

Algorithm 2

```
Procedure BuildLocality[$q, Q, k$]
Input: $q$ is the query block
Input: $Q$ is a set of blocks; decomposition of underlying
    space
Output: $S \leftarrow$ set of blocks, initially empty; locality
    of $q$
Output: PrunedList $\leftarrow$ set of blocks, initially empty;
    $\forall b \in Q$ s.t., $b \notin S$
Output: PruneDist $\leftarrow$ size of the locality; initially 0
    (* count($b$) is the number of points contained in
    the block $b$ *)
    (* integer total $\leftarrow k$ *)
    (* block $b \leftarrow$ NULL *)
1.  while (total $\geqslant 0$) do
2.      $b \leftarrow$ NextInMaxDistOrder($Q$, $q$)
3.      (* Remove $b$ from $Q$ *)
4.      PruneDist $\leftarrow$ MaxDist($q$, $b$)
5.      total $\leftarrow$ total $-$ count($b$)
6.      Insert($S$, $b$)
7.  end-while
8.  while not (IsEmpty($Q$)) do
9.      $b \leftarrow$ NextInMinDistOrder($Q$, $q$)
10.     (* Remove $b$ from $Q$ *)
11.     if (MinDist($q$,$b$) $\leqslant$PruneDist) then
12.         Insert($S$, $b$)
13.     else
14.         Insert(PrunedList, $b$)
15.     end-if
16. end-while
17. return ($S$, PrunedList, PruneDist)
```

3.1. Optimality of the BuildLocality algorithm

In this section, we present few interesting properties of the BuildLocality algorithm. The discussion below is based on [20].

Definition 1 (*locality*). Let Q be a decomposition of the underlying space into a set of blocks. The locality S of a point q is defined to be a subset of Q, such that all of the k nearest neighbors of q are contained in S. The locality S of a block b is defined to be a subset of Q, such that all the *kNN*s of all the points in b are contained in S.

Definition 2. Given a point q, let n_i^q be the ith nearest neighbor of q at a distance of d_i^q. Let b_i^q be a block in Q containing n_i^q.

Definition 3 (*kNN-hyper-sphere*). Given a point q, the *kNN-hyper-sphere* $H(q)$ of q is a hyper-sphere of radius r_q centered at q, such that $H(q)$ completely contains all the blocks in the set $L = \{b_i^q | i = 1, \ldots, k\}$.

Corollary 4. *The number of points contained in the kNN-hyper-sphere $H(q)$ of a point q is $\geqslant k$.*

Definition 5 (*Optimality*). The locality S of a point q is said to be *optimal*, if S contains only those blocks that intersect with $H(q)$.

The rationale behind the definition of optimality is explained below. Let us assume that an optimal algorithm to compute the locality of q consults an *oracle*, which reveals the identify of the set of blocks $L = \{b_1^q, b_2^q \ldots b_q^k\}$ in Q containing the k nearest neighbors of a point q (as shown in Fig. 4). Given such a set L by the oracle, the optimal algorithm would still need to examine the blocks in the hyper-region $H(q)$ in order to verify that the points in L are indeed the k closest neighbors of q. We now show that our algorithm is optimal—that is, in spite of not using an oracle, the locality of q computed by our algorithm is always optimal.

Lemma 6. *Given a space decomposition Q into set of blocks, the locality of a point q produced by Algorithm 2 is optimal.*

Proof. Algorithm 2 computes the locality S of a point q by adding blocks from it Q to S in an increasing MaxDist ordering from q, until S contains at least k points. At this point, let PruneDist be the maximum value of MaxDist encountered so far (i.e., to the last block in the MaxDist ordering that was added to S). Next, the algorithm adds all blocks whose MinDist value is less than the PruneDist. We now demonstrate that the locality S is optimal by showing that a block that does not intersect with the *kNN*-hyper-sphere $H(q)$ of q cannot belong in S. Suppose that $b \in S$ is a block that does not intersect $H(q)$ of radius r_q,—that is, by definition

$$r_q < \text{MinDist}(q, b) \leqslant \text{PruneDist}. \quad (1)$$

From Corollary 4, we know that $H(q)$ contains at least k points.

Hence,

$$\text{PruneDist} \leqslant r_q. \quad (2)$$

Combining Eqs. (1) and (2), we have

$$\text{PruneDist} \leqslant r_q < \text{MinDist}(q, b) \leqslant \text{PruneDist},$$

which is a contradiction.

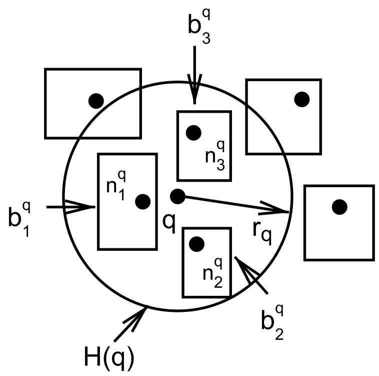

Fig. 4. Figure shows the *kNN*-hyper-sphere $H(q)$ of a point q when $k = 3$. Note that $H(q)$ completely contains the blocks b_1^q, b_2^q and b_3^q.

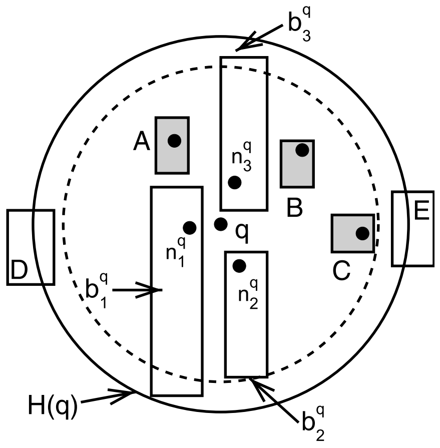

Fig. 5. The locality S of a point q computed by Algorithm 2 ($k = 3$) initially adds A, B, C to the locality of q, thus satisfying the initial condition that the number of points in S be equal to 3. Now PruneDist is set to MaxDist(q, C). Next, we add blocks whose MinDist to q is less than the PruneDist, thus adding the blocks b_1^q, b_2^q, and b_3^q to S. Note that the locality of q computed by Algorithm 2 may not contain all the blocks that intersect with $H(q)$ i.e., blocks D and E intersect with $H(q)$, but are not in S.

Note however that not all the blocks that intersect with $H(q)$ must be in S, as shown in Fig. 5, where D and E intersect $H(q)$ while not being in S.

We now show that the locality of a block b that is computed by Algorithm 2 is also optimal.

Definition 7 (*kNN-hyper-region*). Given a block b, let L be the subset of blocks in Q such that any block in L contains at least one of the k nearest neighbors of a point in b. The *kNN-hyper-region* $H(b)$ of b is a hyper-region R, such that any point contained in R is closer to b than the block r containing the farthest possible point from b in L—that is, r is the farthest block in L, if $\forall b_i \in L$, MaxDist$(r, b) \geqslant$ MaxDist(b_i, b). Now, $H(b)$ is a hyper-region R, such that the minimum distance of a point in R to b is less than or equal to MaxDist(r, b).

Definition 8 (*Optimality*). The locality S of a block b is said to be *optimal*, if S contains only those blocks that intersect with the *kNN*-hyper-region of b.

The rationale behind the definition of the optimality of a block is the same as that for a point. Even if our algorithm is provided with an *oracle*, which identifies the subset of blocks in Q containing at least one of the *kNN*s of a point in b, the blocks that intersect with $H(b)$ must be examined in order to prove correctness of the result.

Corollary 9. *The number of objects contained in the kNN-hyper-sphere $H(b)$ of a block b is $\geqslant k$.*

Lemma 10. *Given a space decomposition Q into set of blocks, the locality of a block b produced by Algorithm* 2 *is optimal.*

Proof. Follows from Lemma 6.

Note however, that the algorithm is optimal with respect to the given space decomposition Q. That is, the BuildLocality algorithm will never add a block b to the locality that cannot contain a nearest neighbor to any point contained in b, although, depending on the nature of the decomposition, the size of the locality may be large.

4. Incremental *kNN* algorithm

We briefly describe the working of a incremental variant of our *kNN* algorithm. This algorithm is useful when variable number of neighbors are required for each point in the data set. For example, when dealing with certain point-cloud operations, where the number of neighbors required for a point p is a function of its local characteristics (e.g., curvature), the value of k cannot be pre-determined for all the points in the data set, i.e., a few points may require more than k neighbors. The incremental *kNN* algorithm given in Algorithm 3 can produce as many neighbors as required by the point-cloud operation. This is contrast to the standard implementation of the ANN algorithm [12], where retrieving the $k + 1$th neighbor of p entails recomputing all of the first $k + 1$ neighbors to p.

Algorithm Inc*kNN* computes the nearest neighbors of a point p incrementally. The inputs to the algorithm are the point p whose nearest neighbors are being computed, the leaf block b containing p and the locality S of b. A priority queue Q in line 1 retrieves elements in increasing MinDist ordering from p. Initially, the locality S of b is *enqueued* in Q in line 2. At each step of the algorithm the top element e in Q is retrieved. If e is a Block, then e is replaced with its children blocks (lines 15–16). If e is a point, it is reported (line 18) and the control of the program returns back to the user. Additional neighbors of p are retrieved by making subsequent invocations to the algorithm. Note that S is guaranteed to only contain the first *kNN*s of p, after which the PrunedList of the parent block of b (subsequently, an ancestor) in the search hierarchy is enqueued into Q, as shown in lines 6–12.

Algorithm 3

```
Procedure InckNN[p, b, S]
Input: b is a leaf block
Input: p is a point in b
Input: S is a set of blocks; locality of b
  (* FindPruneDist(b) returns the PruneDist of
   the block b *)
  (* FindPrunedList(b) returns the PrunedList
   of the block b *)
  (* Parent(b) returns the parent block of b in the
   search hierarchy *)
  (* element e *)
  (* priority_queue Q ← empty; priority queue of
   elements *)
  (* float d ← FindPruneDist(b) *)
1.  Init: InitQueue(Q)
            (* MinDist ordering of elements in Q from p *)
```

```
2.      EnQueue($Q, S$)
3.   End-Iinit
4.   while not (IsEmpty($Q$)) do
5.      $e \leftarrow$ DeQueue($Q$)
6.      if (MinDist($e, b$) $\geqslant d$) then
7.         if ($b$ = root) then
8.            $d \leftarrow \infty$
9.         else
10.           $b \leftarrow$ Parent($b$)
11.           EnQueue($Q$, FindPrunedList($b$))
12.           $d \leftarrow$ FindPruneDist($b$)
13.        end-if
14.     end-if
15.     if ($e$ is a block) then
16.        EnQueue($Q$, Children($e$))
17.     else (* $e$ is a Point *)
18.        report $e$ as the next neighbor (and return)
19.     end-if
20.  end-while
```

5. Non-incremental *kNN* algorithm

In this section, we describe our *kNN* algorithm that computes the *kNN*s of each point in the data set. A point x whose k neighbors are being computed is termed the *query point*. An ordered set containing the k nearest neighbors of x is termed the *neighborhood* $n(x)$ of x. Although the examples in this section assume a two-dimensional space, the concepts hold true for arbitrary dimensions. Let $n(x) = \{q_1^x, q_2^x, q_3^x, \ldots, q_k^x\}$ be the neighborhood of point x, such that q_i^x is the ith nearest neighbor of x, $1 \leqslant i \leqslant k$ with q_1^x being the closest point in $n(x)$. We represent the L_2 distance of a point $q_i^x \in n(x)$ to x as $L_2^x(q_i) = \|q_i - x\|$ or d_i^x. Note that all points in the neighborhood of x are drawn from the *locality* of the leaf block containing x. The L_∞ distance between any two points u and v is denoted by $L_\infty^u(v)$.

The neighborhood of a succession of query points is obtained as follows. Suppose that the neighborhood $n(x)$ of the query point x has been determined by a search process. Let q_k^x be the farthest point in $n(x)$, such that the k nearest neighbors of x are contained in a circle (a hyper-sphere in higher dimensions) of radius d_k^x centered at x. Let y be the next query point under consideration. As mentioned earlier, the algorithm benefits from choosing y to be close to x. Without loss of generality, assume that y is to the *east* and *north* of x as shown in Fig. 6a. As both x and y are spatially close to each other, they may share many common neighbors and thus we let y use the neighborhood of x as an initial estimate of y's neighborhood, termed the *approximate neighborhood* of y and denoted by $n'(y)$, and then try to improve upon it. At this point, let d_k^y record the distance from y to the farthest point in the approximate neighborhood $n'(y)$ of y.

Of course, some of the points in $n'(y)$ may not be in $n(y)$. The fact that we use the L_2 distance metric means that the

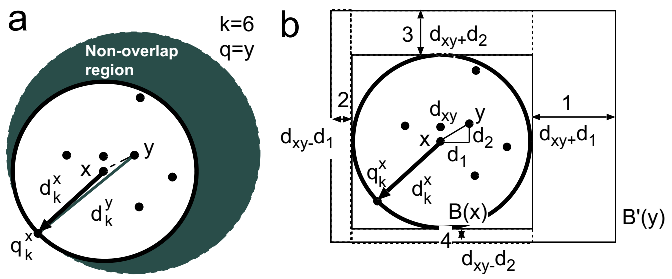

Fig. 6. (a) Searching the shaded region for points closer to y than q_k^y is sufficient; (b) to compute $B(y)$ from $B(x)$ requires four simple region searches. Compared to searching the crescent shaped region, these region searches are easy to perform.

region spanned by $n(x)$ is of a circular shape. Therefore, as shown in Fig. 6a, we see that some of the *kNN*s of y may lie in the shaded crescent-shaped region formed by taking the set difference of the points contained in the circle of radius d_k^y centered at y and the points contained in the circle of radius d_k^x centered at x. Thus, in order to ensure that we obtain the *kNN*s of y, we must also search this crescent-shaped region whose points may displace some of the points in $n'(y)$. However, it is not easy to search such a region due to its shape, and thus the *kNN* algorithm would benefit if the shape of the region containing the neighborhood could be altered to enable efficient search, while still ensuring that it contains the *kNN*s of y; although it could contain a limited number of additional points.

Let $B(x)$ be the *bounding box* of $n(x)$, such that any point p contained in $B(x)$ satisfies the condition $L_\infty^x(p) \leqslant d_k^x$, i.e., $B(x)$ is a square region centered at x of width $2 \cdot d_k^x$, such that it contains all the points in $n(x)$. Note that $B(x)$ contains all the k nearest neighbors of x and additional points in the region that does not overlap $n(x)$. While estimating a bound on number of points in $B(x)$ is difficult, at least in two-dimensional space we know that the ratio of the non-overlap space occupied by $B(x)$ to $n(x)$ is $(4 - \pi)/\pi$. Consequently, the expected number of points in $B(x)$ is proportionately larger than $n(x)$.

Once we have $B(x)$ of a point x, we obtain a rectangular region $B'(y)$, termed *approximate bounding box* of $n(y)$, such that $B'(y)$ is guaranteed to contain all the points in $n(y)$. This is achieved by adding four simple rectangular regions to $B(x)$ as shown in Fig. 6b. In general for a d-dimensional space, 2^d such regions are formed. Although, this process is simple, it may have the unfortunate consequence that its successive application to query points will result in larger and larger bounding boxes—that is, $B'(y)$ computed using such a method is larger than $B(y)$. We avoid this repeated growth by following the determination of d_k^y using $B'(y)$ with a computation of a smaller $B(y)$ with a width of $2 \cdot d_k^y$.

Algorithm 4 takes a leaf block b and the locality S of b as input and computes the neighborhood for all points in b. First of all, the points in b are visited in some predetermined sequence (line 1), usually the ordering of points is established using a space-filling curve [8].

The neighborhood $n(u)$ of the first point u in b (lines 5–8) is computed by choosing the k closest points to u in S. This is done by making use of an incremental nearest neighbor finding algorithm such as BFS [10]. Note that at this stage, we could also make use of an approximate version of BFS as pointed out in Section 1. Once the k closest points have been identified, the value of d_k^u is known (line 9). At this point we add the remaining points in $B(u)$ as they are needed for the computation of the neighborhood of the next query point in b. In particular, $B(u)$ is constructed by adding points $o \in S$ to $n(u)$ such that they satisfy the condition $L_\infty^u(o) \leqslant d_k^u$ (lines 10–15). Subsequent points in b are handled in lines 16–21. The points in the bounding box $B'(u)$ of u is computed by using the points in the bounding box $B(u)$ of the previous point p and then making 2^d region searches on S as shown in Fig. 6b (line 18). Finally, $B(u)$ is computed by making an additional region search on $B'(u)$ as shown in line 21.

Algorithm 4

```
Procedure BuildNeighborhood[$b$, $S$]
Input: $b \leftarrow a$ leaf block
Input: $S \leftarrow$ set of blocks; locality of $b$
   (* point $p, u \leftarrow$ empty *)
   (* ordered_set $B_p, B_u, B'_u \leftarrow$ empty *)
   (* If $B$ is an ordered set, $B[i]$ is the ith element in $B$ *)
   (* integer count $\leftarrow 0$ *)
1.    for each point $u \in b$ do
2.       if ($p$ = empty) then
3.          (* compute the neighborhood of the first
            point in $b$ *)
4.          count $\leftarrow 0$
5.          while (count $< K$) do
6.             Insert($B_u$, NextNN($S$))
7.             count $\leftarrow$ count $+ 1$
8.          end-while
9.          $d_k^u \leftarrow L_2^u(B_u[k])$
10.         $o \leftarrow$ NextNN($S$)
11.         (* add all points that satisfy the $L_\infty$ criterion *)
12.         while ($L_\infty^u(o) \leqslant d_k^u$) do
13.            Insert($B_u, o$)
14.            $o \leftarrow$ NextNN($S$)
15.         end-while
16.      else (* $p \neq$ empty *)
17.         (* $2^d$ region searches as shown in
            Fig. 6b for a two dimensional case*)
18.         $B'_u \leftarrow B_p \bigcup$ RegionSearch($S, L_2^p(u), d_k^p$)
19.         $d_k^u \leftarrow L_2^u(B'_u[k])$
20.         (* Search a box of width $d_k^u$ around $u$ *)
21.         $B_u \leftarrow$ RegionSearch($B'_u, d_k^u$)
22.      end-if
23.      $d_k^u \leftarrow L_2^u(B'_u[k])$
24.      $p \leftarrow u$
25.      $B_p \leftarrow B_u$
26.   end-for
27.   return
```

6. Experimental comparison with other algorithms

A number of experiments were conducted to evaluate the performance of the *kNN* algorithm. The experiments were performed on a Quad Intel Xeon server running Linux(2.4.2) operating system with one gigabyte of RAM and SCSI hard disks. The data sets used in the evaluation consists of 3D scanned models that are frequently used in computer graphics applications. The three-dimensional point models range from 2k to 50 million points, including two *synthetic* point models of size 37.5 million and 50 million, respectively. We developed a toolkit in C++ using STL that implements the *kNN* algorithm. The performance of our algorithm was evaluated by varying a number of parameters that are known to influence its performance. We collected a number of statistics such as the time taken to perform the algorithm, the number of distance computations, the average locality size, page size, cache size, and the resultant number of page faults. The average size of the locality is the average number of blocks in the locality of all points in the data set.

A good benchmark for evaluating our algorithm is to compare it with a *sorting* algorithm. We make this unintuitive analogy with a sorting algorithm by observing that the work performed by the *kNN* algorithm in a one-dimensional space is similar to sorting a set of real numbers. Consider a data set S containing n points in a one-dimensional space as an input to a *kNN* algorithm. An efficient *kNN* algorithm would first sort the points in S with respect to their distance to some origin, thereby incurring $O(n \log n)$ distance computations. It would then choose the k closest neighbors to each point in the sorted list, thus, incurring an additional $O(kn)$ distance computations. We point out that it is difficult for any *kNN* algorithm in a higher dimensional space to asymptotically do better than $O(n \log n)$ as the construction of any spatial data structure is, in fact, an implicit sort in a high-dimensional space. We use the *distance sensitivity* [18], defined below,

$$\text{distance sensitivity} = \frac{\text{Total number of distance calculations}}{n \log n}$$

to evaluate the performance of our algorithm. Notice that the denominator of the above equation corresponds to the cost of a sorting algorithm in a one-dimensional space. A reasonable algorithm should have a low, and more importantly, a constant distance sensitivity value.

We evaluated our algorithm by comparing the execution time and the distance sensitivity of our algorithm with that of the GORDER method [18] and the MuX method [21]. We also compared our algorithm with traditional methods like the *nested join* [22] and a variant of the BFS algorithm [10]. We use both a bucket PR quadtree [8] and an R-tree [9] variant of the *kNN* algorithm in our study. Our evaluation was in terms of three-dimensional point models as we are primarily interested in databases for computer graphics applications. The applicability of our algorithm to data of even higher dimensionality is a subject for future research. We first discuss the effect of each of the following three variables on the performance of the algorithms.

(i) The size of the disk pages which is related to the value of the bucket capacity in the construction of the bucket PR quadtree (Section 6.1).
(ii) The memory cache size (Section 6.2).
(iii) The effect of the size of the data set (Section 6.3).

Once we have determined the effect of these variables on the algorithm, we choose appropriate values to compare our algorithm with the other methods in Section 6.4.

6.1. *Effect of bucket capacity (B)*

In this section, we study the effect of the bucket capacity B on the performance of our *kNN* algorithm. The bucket capacity B also corresponds to the size of the *disk page*. For a given value of k between 8 and 1024, the value of B was varied between 1 and 1024. The performance of our algorithm using a bucket PR quadtree was evaluated by measuring the execution time of the algorithm, the average number of blocks in the locality of the leaf blocks, and the resulting *distance sensitivity* of the algorithm. In this set of experiments, we made use of the Stanford Bunny model containing 35,947 points.

Fig. 7a shows the effect of B on the execution time of the *kNN* algorithm. Note that for smaller values of B ($\leqslant 16$), the *kNN* algorithm has a large execution time. However, it quickly decreases for slightly larger values of B . For values of B between 32 and 128, our *kNN* algorithm has some of the lowest execution times. Fig. 7b shows the average number of blocks in the *locality* of the leaf blocks of the bucket PR quadtree. When B is small, the size of the locality is large. As a result, for small values of B the algorithm has a higher execution time. However, as the value of B increases the size of the locality quickly reduces to a small constant value. For larger values of B , the increase in execution time can be attributed to a larger number of points stored in the blocks in the locality, even though the number of blocks in the locality remains almost the same. The sensitivity analysis shown in Fig. 7c is similar to Fig. 7a. To summarize, the *kNN* algorithm performs well for moderately small values of B , and in particular for the range of B between 32 and 128.

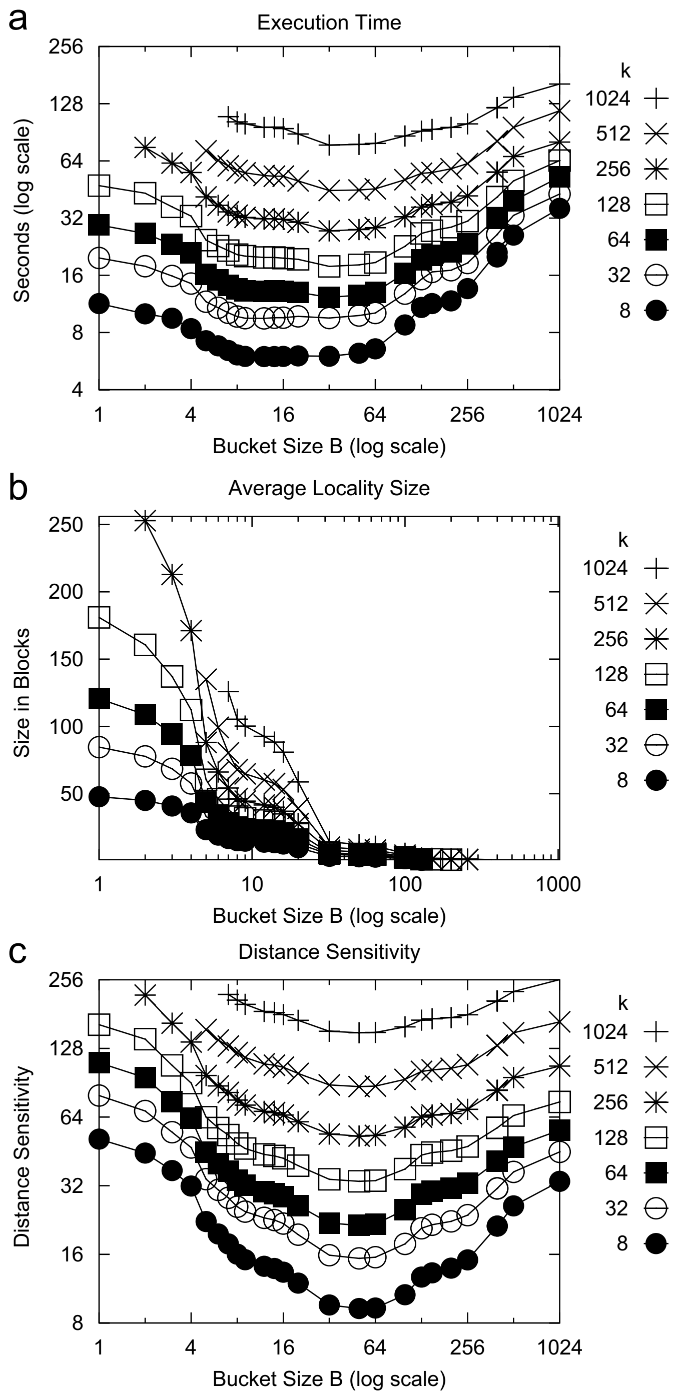

Fig. 7. Effect of Bucket capacity B on the: (a) execution time; (b) average size of the locality in blocks, and (c) distance sensitivity for different values of k for our *kNN* algorithm.

6.2. *Effect of cache size*

The next set of experiments examines the effect of the *cache size* on the performance of our *kNN* algorithm. The cache size is defined in terms of the number of leaf blocks that can be stored in the main memory. We use a *least recently used* (LRU) replacement policy on the disk pages stored in the cache. The size of each memory page is determined by the value of B . We record the effect of the size of the cache on the resulting number of page faults, and the time spent on I/O operations. Figs. 8a–b shows the result of the experiments for $B = 32$ and for varying values of k ranging between 8 and 1024. We observed high values for the I/O time and the number of page-faults for small ($\leqslant 32$) cache sizes, but these values quickly decreased when the cache size was increased beyond a certain value. This

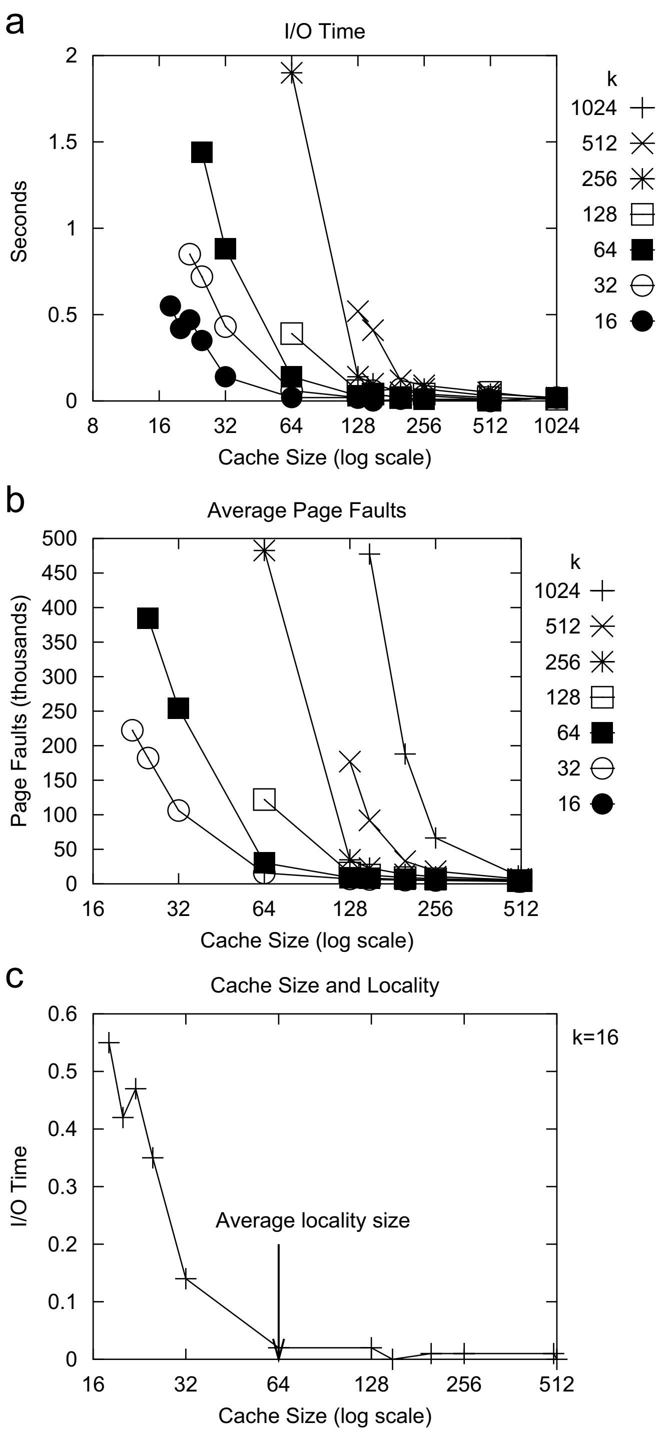

Fig. 8. Effect of cache size on: (a) the time spent on I/O; (b) the number of page faults; for varying values of k and $B = 32$; (c) a comparison between the cache size and the average size of the locality for $k = 16$.

value, incidentally, corresponds to the average size of the locality, as seen in Fig. 8c. Moreover, this also explains the occurrence of large number of page faults when the size of the cache is smaller than the size of the locality. The rule of thumb is that the cache size should be at least as large as the average size of the locality.

6.3. *Effect of data set size*

Experiments were also conducted to evaluate the scalability of the algorithm as the size of the input data set is increased. We experimented with several three-dimensional point models ranging in size from 2k to 50 million points as shown in Fig. 9. The bucket size B and the cache-size were set to 32 points and 500 blocks, respectively. The results of the experiments are given in Figs. 10–11. Fig. 10a shows the effect of size on the time taken to perform the kNN algorithm. Fig. 10b records the distance sensitivity of the algorithm. As the distance sensitivity of our approach is almost linear, our algorithm exhibits $O(n \log n)$ behavior. Fig. 10c records the average size of the locality. We also note that the average locality size is almost constant for data sets of all sizes used in the evaluation. Also, the size of the locality showed only a slight increase even as the value of k is increased from 8 to 16. The I/O time and the resultant number of page faults are given in Fig. 11. Fig. 11a shows the effect of the *size* of the data set on the time spent by the algorithm on I/O operations. Fig. 11b shows the number of page faults normalized by size for data sets of various sizes. Both the time spent on I/O and the number of page faults exhibit linear dependence on the size of the data set.

6.4. *Comparison*

We evaluated our algorithm by comparing its execution time and distance sensitivity with that of the GORDER method [18] and the MuX method of Böhm et al. [19]. Our comparison also includes traditional methods like the *nested join* [22] and a variant of the BFS algorithm that invoked a BFS algorithm for each point in the data set. We used both a bucket PR quadtree and an R-tree variant of the algorithm in the comparative study. The R-tree implementation of our algorithm used a *packed* R-tree [23] with a bucket-capacity of 32 and a *branching factor* of 8. Note however, that the values of B and k are chosen independent of each other. We retained 10% of the disk pages in the main memory using a LRU based page replacement policy. For the GORDER algorithm, we used the parameter values that led to its best performance, according to its developers [18]. In particular, the size of a sub-segment was chosen to be 1000, the number of grids were set to 100, and the size of the data set buffers was chosen to be more than 10% of the data set size. For the MuX-based method, a page capacity of 100 buckets and a bucket capacity of 1024 points was adopted. There are a few differences between the MuX method as described in [19] and our implementation. In particular, we adapted our implementation into a three level structure with a set of *hosting pages* where each page contains several buckets with pointers to a disk-based store. Also, we did not use a *fractionated* priority-queue as described in [19] but replaced it with a heap-based priority queue. However, we did not take into the account the time taken to manipulate the

Model Name	Size (millions)	Model Name	Size (millions)
Bunny (B)	0.037	Femme (F)	0.04
Igea (I)	0.13	Dog (Do)	0.195
Dragon (Dr)	0.43	Buddha (Bu)	0.54
Blade (Bl)	0.88	Dragon (Ld)	3.9
Thai1 (T)	5.0	Lucy (L)	14.0
Syn-38 (S)	37.5	Syn-50 (M)	50.0

Fig. 9. Pseudo names of the point models and the corresponding number of points (in millions) used in the evaluation.

heap structure, thereby ensuring that these differences in the implementation do not affect the comparison results. Also, we only count the point–point distance computations in determining distance-sensitivity and disregard all other distance computations even though they form a substantial fraction of the execution time. We used a bucket capacity of 1024 for the BFS and nested join [22] methods. The results of our experiments were as follows.

(i) Our algorithm clearly out-performs all the other methods for all values of k on the Stanford Bunny model as shown in Fig. 12a–b. Our algorithm leads to at least an order of magnitude improvement in the distance sensitivity over the MuX, the GORDER , the BFS and the nested join techniques for smaller values of k ($\leqslant 32$) and at least 50% improvement for larger k (< 256) as seen in Fig. 12b. We observed an improvement of at least 50% in the execution time (Fig. 12a) over the competing methods.
(ii) However, as size of the input data set is increased the performance of the MuX algorithm was comparable to the nested, BFS and the GORDER based methods (Fig. 13a). Moreover, our method has an almost constant distance sensitivity even for large data sets. The distance sensitivity of the comparative algorithms are at least an order of magnitude higher for smaller data sets and up to several orders of magnitude higher for the larger datasets in comparison to our method (Fig. 13b). We observed similar execution time speedups as seen in Fig. 13a.
(iii) Fig. 13 shows similar performance for the R-tree and the quadtree variants of our algorithm.

7. Applications

Having established that our algorithm performed better than the GORDER and MuX methods, we next evaluated the use of our algorithm in a number of applications for different data sets that included both publicly available and synthetically generated point-cloud models. The size of the models ranged from 35,947 points (*Stanford Bunny* model) to 50 million points (*Syn-50* model). These applications include computing the surface normals to each point in the point-cloud using a variant of the algorithm by Hoppe et al. [2] and removing noise from the point surface using a variant of the *bilateral filtering* method [3,13]. Fig. 14 shows the time needed for these applications when incorporating an algorithm with a neighborhood of size $k = 8$ for each point in the point-cloud model. Fig. 14b shows that our algorithm results in *scalable* performance even as the size of the data set is increased so that it exceeds the amount of available physical memory in the computer by several orders of magnitude. The scalable nature of our approach is readily apparent from the almost uniform rate of finding the neighborhoods, i.e., 5900 neighborhoods/s for the Stanford Bunny model and 7779 neighborhoods/s for the Syn-50 point-cloud models.

In the rest of this section, we describe in greater detail how our algorithm can be used in these computer graphics applications, and give a qualitative evaluation of its use. In particular, we discuss its use in computing surface normals (Section 7.1), noise removal through mollification of surface normals and bilateral mesh filtering (Section 7.2), as well as briefly mentioning additional related applications (Section 7.3).

7.1. Computing surface normals

Point-cloud models are distinguished from other models by not containing any topological information. Thus, one of the initial preprocessing steps required before the point-cloud model can be successfully used is to compute the surface normal for each point in the model. Computing the surface normal is important for the proper display and rendering of point-cloud data. Using the surface normal information, other topological features of a point surface can be estimated. For example, we can estimate the presence of sharp corners on the point-cloud models with reasonable certainty. A sudden large deviation in the orientation of the surface normals within a small spatial distance may indicate the presence of a sharp corner. Many such local surface properties can be estimated by examining the surface normals and the neighborhood information.

One of the most prominent methods for computing surface normals for unorganized points is due to Hoppe et al. [2]. This method relies on computing the *kNN*s to

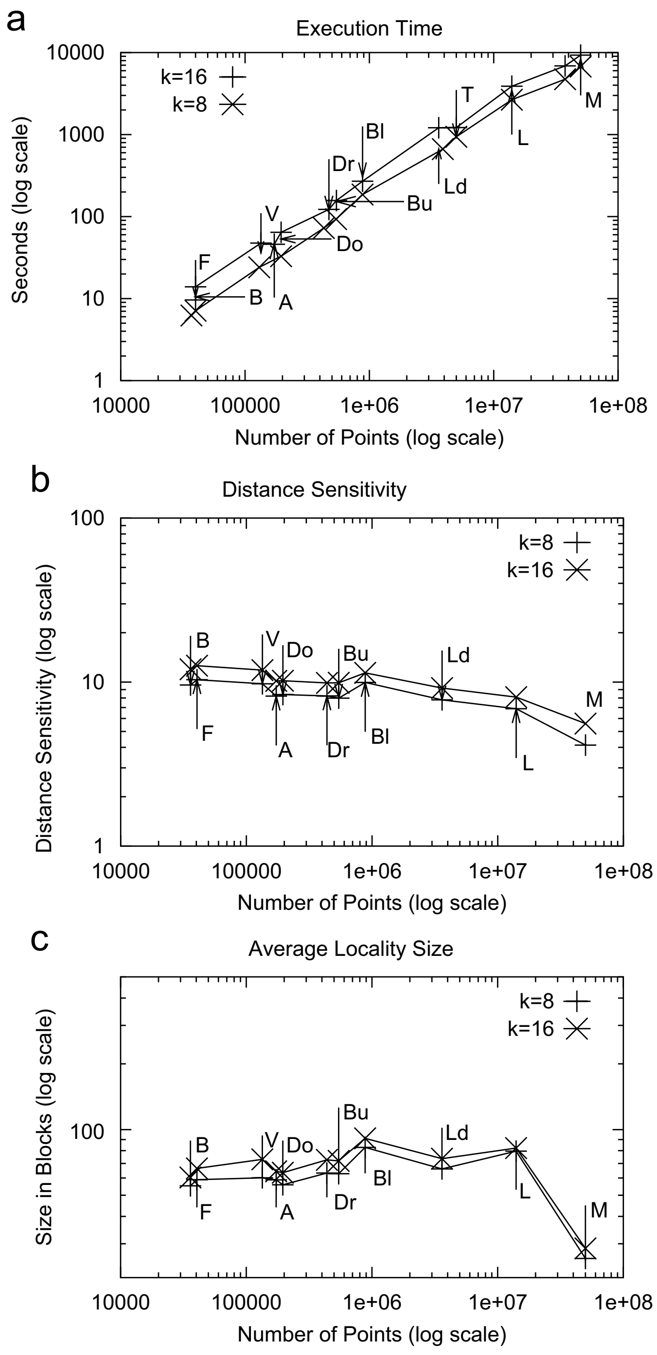

Fig. 10. Effect of the *size* of the data set on: (a) execution time; (b) distance sensitivity, and (c) average locality size for various point models with $B = 32$ and 500 blocks in the memory cache.

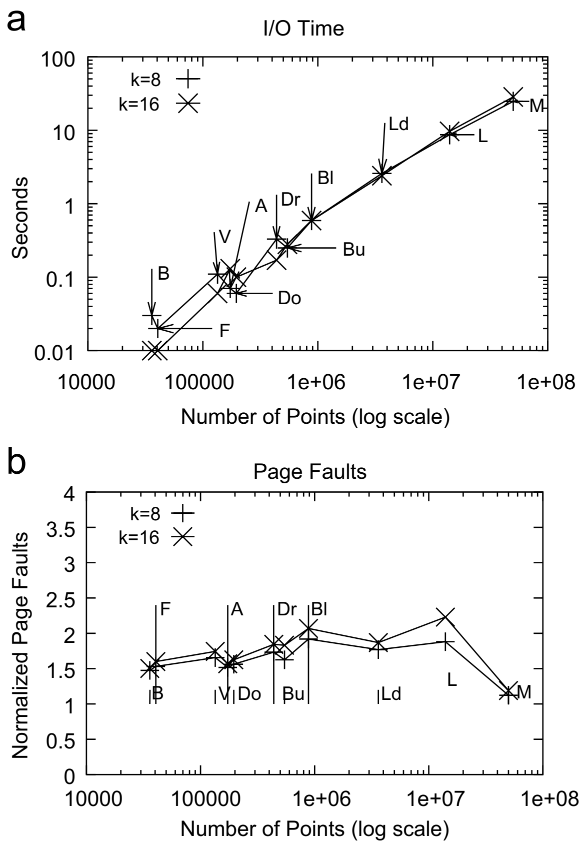

Fig. 11. Effect of size of the data set on: (a) time spent on I/O; (b) the number of page faults normalized by size for datasets of various sizes.

each point in the data set. The neighborhood is *fit* with a *hypothetical surface* which minimizes the sum of the squared distances from each point in the neighborhood to the hypothetical surface. A *covariance analysis* of the resulting neighborhood leads to the estimation of the normals to the surface and the query-point.

A more recent contribution is by Mitra et al. [15] which deals with the computation of the surface normals to a point-cloud in the presence of noise. This algorithm computes the neighborhood of points in the data set after taking into consideration the sampling density and the curvature of the neighborhood. There is also the alternative approach of Floater and Reimers [24] that *triangulates* the neighborhood and computes the surface normals from the resulting mesh surface.

The neighborhood finding algorithms used in these methods are as diverse as the methods themselves. The algorithm by Hoppe [2] assumes a uniform sampling of points in the point-cloud. This makes the computation of neighborhood almost trivial, although not realistic. Also, many algorithms use either an approximate brute-force method or compute the neighborhood by repeated computation of the neighborhood for one point at a time (e.g., see [15]).

We computed the surface normal information of several data sets using a method similar to that of Hoppe et al. [2]. We tabulated the time taken for data sets of different sizes and also recorded the effect of varying the size of the neighborhood on the resulting neighborhood calculation. The effect of varying the size of the data set when computing the surface normals is given by the appropriately labeled column in Fig. 14a. The main results of using our algorithm to compute surface normals are as follows:

(i) The *quality* of the surface normals depends on the size of the neighborhood as can be seen in Fig. 15. Using

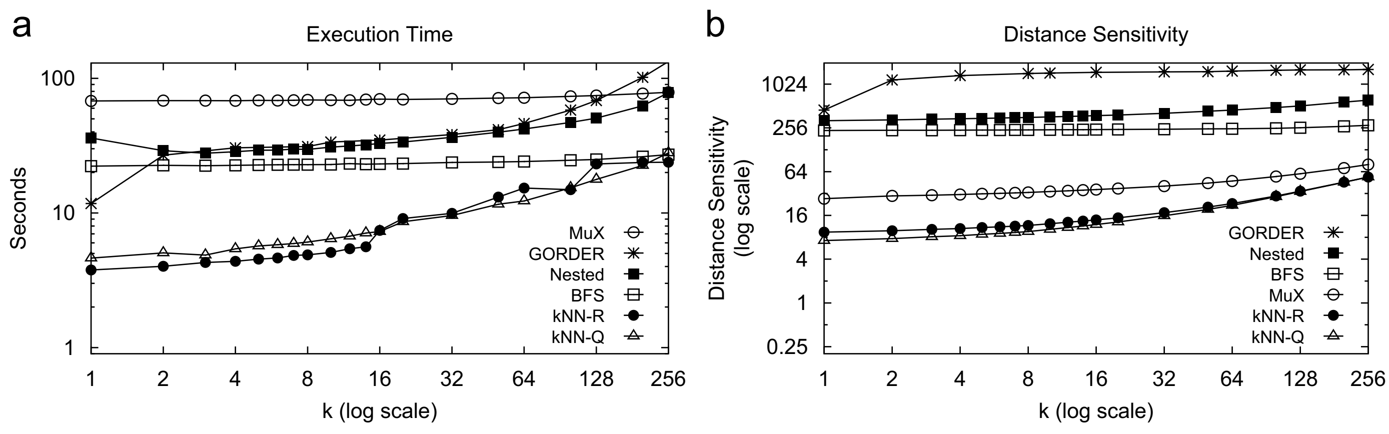

Fig. 12. Performance comparison of our *kNN* algorithm with the BFS, GORDER , MuX and the Nested join algorithms. '*kNN*-Q' and '*kNN*-R' refers to the quadtree and R-tree implementations of our algorithm respectively. Plots a–b show the performance of the techniques on the *Stanford Bunny* model containing 35,947 points for values of k ranging between 1 and 256; (a) execution time, and (b) distance sensitivity.

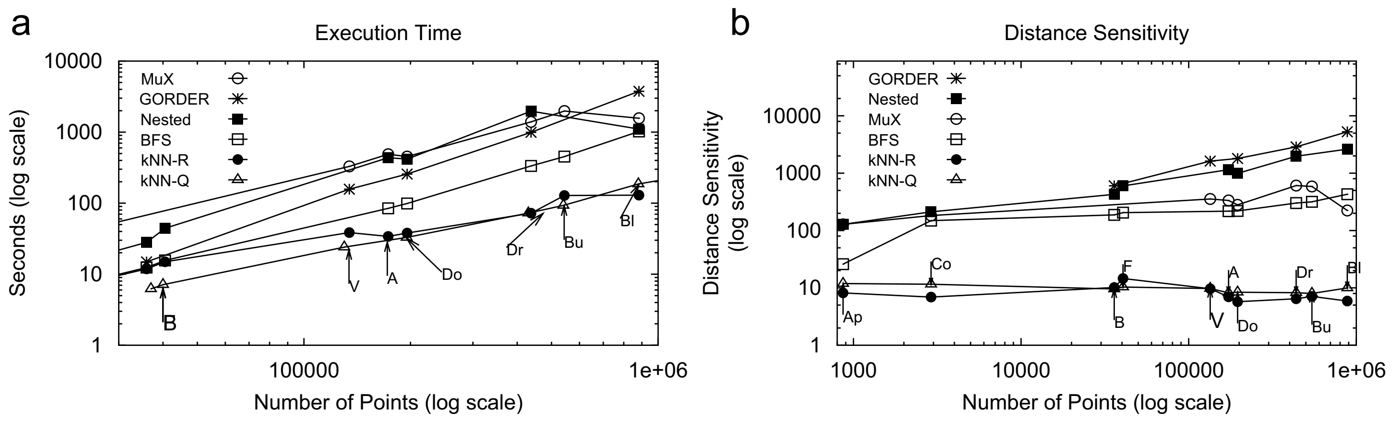

Fig. 13. Performance comparison of our *kNN* algorithm with the BFS, GORDER , MuX and the Nested join algorithms. '*kNN*-Q' and '*kNN*-R' refers to the quadtree and R-tree implementations of our algorithm, respectively. Plots a–b record the performance of all the techniques on data sets of various sizes for $k = 8$; (a) execution time, and (b) distance sensitivity.

the surface normals for $8 \leqslant k \leqslant 64$ retains the finer details on the surface (Figs. 15a–b). Using a larger neighborhood such as $k \geqslant 64$ leads to a loss of many of the finer surface details (Fig. 15c). This effect can be attributed to the *averaging* property of the neighborhood.

(ii) When dealing with noisy meshes, the surface normals computed using the topological information of the mesh are often erroneous as can be seen in the dragon model in Fig. 16a. In such cases, we can use our *kNN* algorithm to compute the surface normals by just using the neighborhood of the points and the result is relatively error-free as seen in Fig. 16b when using 8 neighbors. This leads us to observe that correct surface normals are important for the proper display of the point model, and that the normals computed by analyzing the neighborhood are resilient to noise, but result in a loss in surface details if an unsuitable value of k is used as seen in Fig. 15c.

7.2. Noise removal

With advances in scanning technologies, many objects are being scan converted into point-clouds. The objects are scanned at a high resolution in order to capture the surface details and to provide an illusion of a smooth compact surface by the close placement of the points comprising the point-cloud model. However, in reality, points in a freshly scanned point-cloud model are noisy due to environmental interference, material properties of the scanned object, and calibration issues with the scanning device. Often, an additional corrective procedure needs to be performed in order to account for the residual noise before the model can be successfully employed. In fact, such an unprocessed point-cloud model would have a *scarred appearance* as illustrated in Fig. 16a which has been obtained by adding a noise element to each of the points in the original model.

Noise is removed by applying a filtering algorithm to the points in the point-cloud model. *Bilateral mesh filtering* a

Model Name	Size (millions)	*kNN*	Surface Normals	Noise Removal
Bunny (Bu)	0.037	6.22	9.0	9.64
Femme (F)	0.04	7.13	10.5	13.9
Igea (I)	0.13	24.05	36.6	47.52
Dog (Do)	0.195	32.9	53.4	64.45
Dragon (Dr)	0.43	72.62	118.9	122.2
Buddha (Bu)	0.54	93.04	152.3	157.25
Blade (Bl)	0.88	185.92	304.2	270.0
Dragon (Ld)	3.9	663.84	900.0	1209.8
Thai (T)	5.0	940.04	1240.0	1215.7
Lucy (L)	14.0	2657.9	3504.0	3877.78
Syn-38 (S)	37.5	4741.79	-	-
Syn-50 (M)	50.0	6427.5	-	-

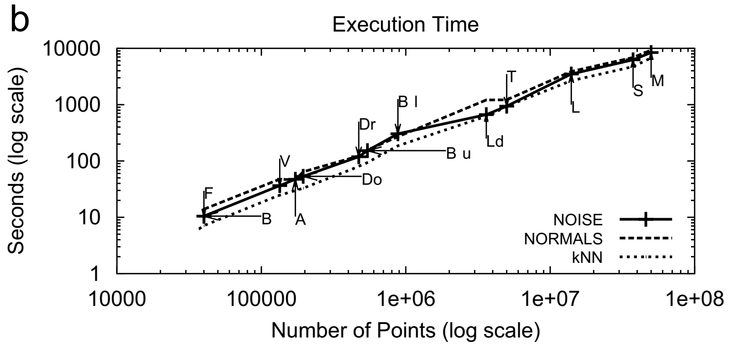

Fig. 14. (a) Tabular and (b) graphical views of the execution time of the *kNN* algorithm for different point models, and the time to execute a number of operations (i.e., normal computation and noise removal) using it. All results are for $k = 8$.

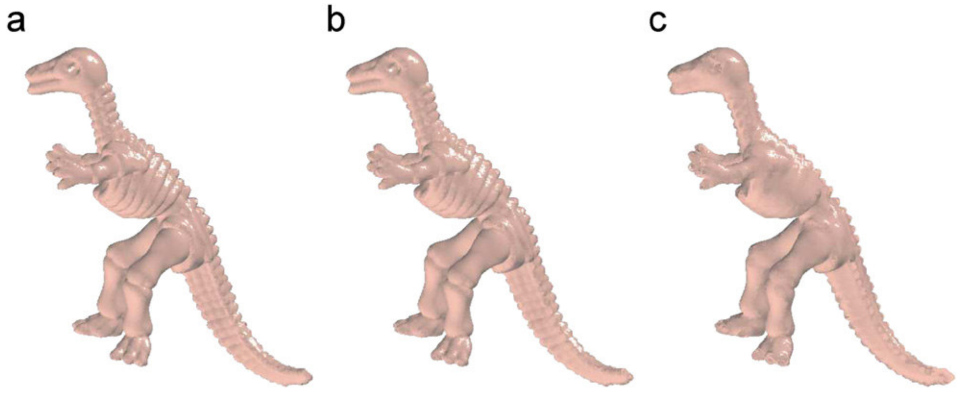

Fig. 15. Dinosaur point-cloud models displayed using surface normals computed with neighborhoods of (a) 16; (b) 64, and (c) 128 neighbors.

[3,13] and *mollification* [25] are two prominent techniques for removing noise from a mesh. While the bilateral mesh filtering algorithm attempts to correct the position of erroneous points, the mollification approach, instead, tries to correct the surface normals at the point. Bilateral mesh filtering is analogous to *displacement mapping* [26] and mollification is analogous to *bump mapping* [27], both of which are prominent texturing techniques that can be used to achieve the same result. In particular, displacement mapping relies on shifting the points themselves to bring about texturing of the surface, while bump-mapping modifies the surface normals at each vertex of the mesh surface. Bilateral filtering differs from another class of techniques, that include MLS noise removal [28], which

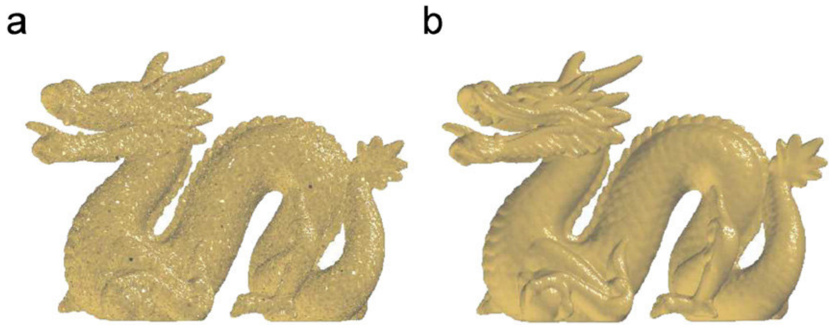

Fig. 16. (a) A noisy mesh-model of a dragon, and (b) the corresponding model whose surface normals were recomputed using our *kNN* algorithm. The algorithm took about 118 seconds and used eight neighbors.

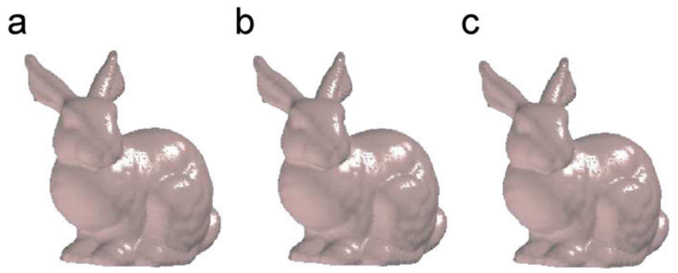

Fig. 17. Results of applying the neighborhood-based adaptation of the bilateral mesh filtering algorithm to the bunny model for Gaussian kernel pairs: (a) $\sigma_f = 2$, $\sigma_g = 0.2$; (b) $\sigma_f = 4$, $\sigma_g = 4$, and (c) $\sigma_f = 10$, $\sigma_g = 10$ for a neighborhood of size 8. The results are independent of the size of the Gaussian kernel that was chosen.

correct the points by reconstructing a smooth local surface and re-sampling points from the surface. In the rest of this section, we discuss the results of our application of both bilateral mesh filtering and mollification to remove noise in large point-cloud models.

We applied the bilateral mesh filtering algorithm in [3,13] to the point-cloud model as follows. We initially computed a neighborhood for each point in the model. Our adaptation of the bilateral filtering method assigns weights (an influence measure analogous to the Gaussian weights in the bilateral filtering method) to each point in the neighborhood in such a way that the computation becomes less *sensitive* to outlier points. Note that mollification corrects the normals instead of the point, but is similar in approach. Fig. 17 shows the results of applying our point-cloud model adaptation of the conventional bilateral mesh filtering algorithm to the bunny model (35,947 points) for different pairs of values of the Gaussian kernel. Note that the quality of the results when using our adaptation does not depend on the values of the Gaussian kernel.

As pointed out earlier, mollification is similar to bilateral mesh filtering with the difference being that instead of performing the filtering operation on the points, the filtering operation is applied to the original surface normals of the points. In order to evaluate the sensitivity of our filtering and surface computation methods to noise, we added Gaussian noise using the Box–Muller method [29] to a bunny mesh-model. We computed the surface normals at each vertex in the noisy mesh using the connectivity information contained in the mesh. The resultant mesh, disregarding the connectivity information, is a point-cloud (as shown in Fig. 19a) with noisy point positions and *noise-corrupted* normals. We use this approach to create the noisy point-clouds used in Figs. 18 and 19. Figs. 19b–d compare the result of using the mollification method (Fig. 19d) with the computation of surface normals as in Section 5.1 (Fig. 19b) and our adaptation of the bilateral mesh filtering method (Fig. 9c). All three methods were applied for 8 neighbors. From the figure, we see that when using our *kNN* method to compute the neighborhoods to be used in computing the surface normals, there is no perceptible difference between the three methods even in the case of noisy data.

7.3. Related applications

The most obvious application of the *kNN* algorithm is in the construction of *kNN* graphs [30]. *kNN* graphs are useful when repeated nearest neighbor queries need to be performed on a data set. The *kNN* algorithm may also be used in *point reaction-diffusion* [31] algorithms. Such algorithms mimic a physical phenomenon to uniformly distribute points on a given surface or space. Many of natural texture patterns encountered in nature can be recreated using this technique. The algorithm works as follows. Each point is assigned a unit positive charge. The resultant repulsion force acting on the point is computed using the *kNN*s at each point. Next, the point is moved along the direction of the force, and the *kNN* algorithm is repeatedly reinvoked at each iteration until an equilibrium condition is reached.

A recent contribution in the construction of approximate surfaces from point sets is the moving least squares (MLS) [28] method. Weyrich et al. [14] have identified useful point-cloud operations that use the MLS method. Of these operations, we believe that MLS *point-relaxation*, MLS *smoothing*, MLS based *upscaling* [28], and *downscaling* can all benefit when used in conjunction with the *kNN* algorithm.

Tools that perform *upscaling* [32,33] and downscaling [28] of point-clouds all use the *kNN* algorithm to generate

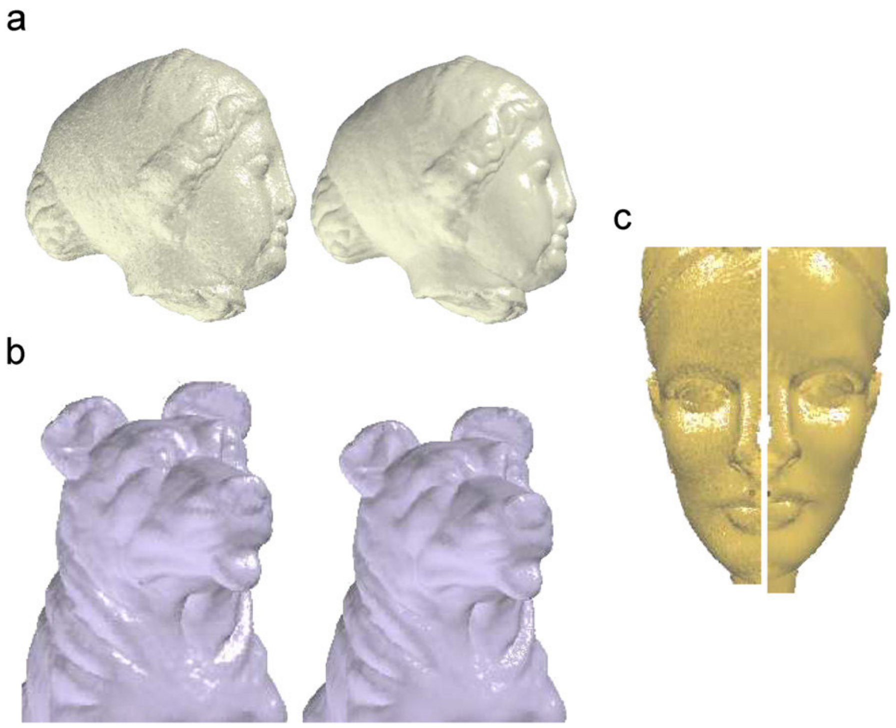

Fig. 18. Three noisy models which were de-noised using filtering and mollification techniques. In the pairs of figures shown for each of the models, the figure on the left is the noisy model, while the figure on the right is the corrected point model. The (a) *Igea* and (b) *dog* models were denoised with the filtering method, while the (c) *femme* model was denoised using the mollification technique.

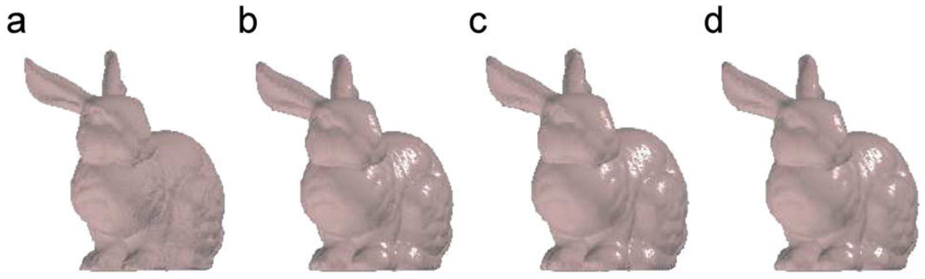

Fig. 19. (a) A bunny point-cloud model to which Gaussian noise was added, and the result of applying; (b) the surface normal computation method in Section 5.1; (c) our adaptation of bilateral mesh filtering, and (d) mollification.

varied *levels of detail* (*LOD*) [34] of point models. The *quadratic error simplification* method [32,33] simplifies a point-cloud by removing the points that make the least significant contribution to the surface details. We have built a sample tool that implements Garland's method [32] on point-clouds and have used it to generate Igea point models of different sampling-rate of as seen in Fig. 20. The Igea model of size 135k was reduced to smaller models of sizes 14k, 48k, 78k, 99k and 111k, the largest of which took less than 120 seconds to generate. The general quality of the reduced model produced by the tool however depends on the extent by which the models were reduced. For example, we can note some loss in facial features in Fig. 20 (14k) while Fig. 20 (111k) is almost identical to the original model.

A similar method to increase the point sampling uses a variant of MLS [28] to insert additional points in the neighborhood (termed *upscaling*). The algorithm computes the k nearest neighbors to each point using the *kNN* algorithm. Points are then evenly distributed [35] on the hypothetical surface that is fit through the points in the neighborhood. We built a variant of the algorithm which when applied to the *apple* model (Fig. 21a) containing 867 points resulted in a new point model containing 27,547 points (Fig. 21b) which took about 1.2 s to construct.

8. Concluding remarks

We have presented a new *kNN* algorithm that yields an improvement of several orders of magnitude for distance sensitivity and at least one order of magnitude improvement in execution time over an existing method known as GORDER designed for dealing with large volumes of data that are disk-resident. We have applied our method to point-clouds of varying size including some as large as 50 million points with good performance. A number of

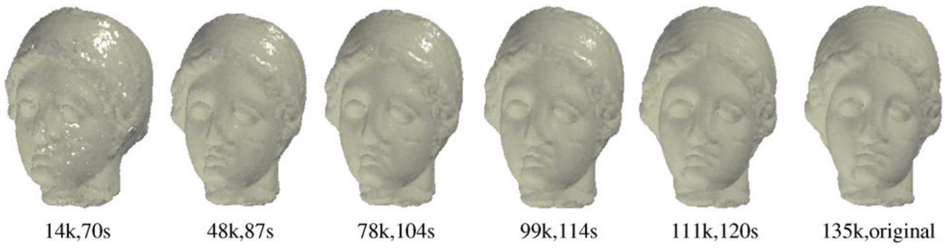

Fig. 20. Sizes and execution times for the result of applying a variant of the simplification algorithm [32] using the *kNN* algorithm to the *Igea* point model of size 135k.

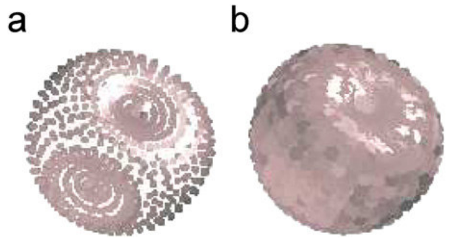

Fig. 21. (a) Initial apple model (867 points) and (b) the result of applying an upscaling algorithm to it using the *kNN* algorithm (27,547 points).

applications of the algorithm were presented. Below, we summarize a few interesting directions for future research.

(i) Although our focus was on the computation of the exact *kNN*s, our methods can also be applied to work with approximate *kNN*s by simply stopping the search for the *kNN*s when k neighbors of the query point within ε of the true distance of the kth neighbor have been found. An interesting problem is to devise an ε approximate locality L of a block b, such that L contains the ε-approximate *kNN*s of all the points contained in b.

(ii) We have shown that for a given subdivision of space, the BuildLocality algorithm is *optimal*, although, the time taken to perform the algorithm depends solely on our choice of the data structure. It is not difficult to see that certain point data set and data structure configurations may result in large localities of the points. An interesting direction of research is the design and analysis of a data structure that can ensure that the average size of the locality is small, thereby providing good performance.

(iii) Our *kNN* algorithm only requires the ability to compute MinDist, MaxDist, and the number of points contained in a block. An interesting study would be to examine if smaller localities can be built if additional statistics on the distribution of the points contained in a block, such as the MaxNearestDist estimator [8], were available to the algorithm.

(iv) Modify our *kNN* algorithm to provide k nearest neighbors that are radially well distributed around the query point. It is not clear if a locality L of a block b can be defined, such that all the radial neighbors of all the points in b are contained in L.

(v) Explore the applicability of some of the concepts discussed here to high-dimensional datasets using techniques such as those described in [36,18].

Acknowledgments

The authors would like to gratefully acknowledge the support of the National Science Foundation under Grants CCF-0515241 and EIA-00-91474, Microsoft Research, and the University of Maryland General Research Board. The authors would like to thank the anonymous reviewers for their useful comments and suggestions which helped improve the quality of this paper immensely. Special thanks are due to Chenyi Xia for providing us with the gorder source code. The point models used in the paper are the creations of the following individuals or institutions. The *Bunny*, *Buddha*, *Thai*, *Lucy* and the *Dragon* models are from the Stanford 3D Scanning Repository. The *Igea* and the *Dinosaur* models are from Cyberware Inc. The *Turbine blade* model was obtained from the CD-ROM associated with the Visualization Toolkit [37] book. The *Dog* model is from MRL-NYU. The *Femme* model is thanks to Jean-Yves Bouguet.

References

[1] Andersson M, Giesen J, Pauly M, Speckmann B. Bounds on the k-neighborhood for locally uniformly sampled surfaces. In: Proceedings of the eurographics symposium on point-based graphics, Zurich, Switzerland; 2004. p. 167–71.

[2] Hoppe H, DeRose T, Duchamp T, McDonald J, Stuetzle W. Surface reconstruction from unorganized points. In: Proceedings of the SIGGRAPH'92 conference. Chicago, IL: ACM Press; 1992. p. 71–8.

[3] Jones TR, Durand F, Desbrun M. Noniterative, feature-preserving mesh smoothing. In: Proceedings of the SIGGRAPH'03 conference, vol. 22(3). San Diego, CA: ACM Press; 2003. p. 943–49.

[4] Levoy M, Pulli K, Curless B, Rusinkiewicz S, Koller D, Pereira L, Ginzton M, Anderson S, Davis J, Ginsberg J, Shade J, Fulk D. The digital Michelangelo project: 3D scanning of large statues. In: Proceedings of the SIGGRAPH'00 conference, New Orleans, LA: ACM Press; 2000. p. 131–44.

[5] Pauly M, Keiser R, Kobbelt LP, Gross M. Shape modeling with point-sampled geometry. ACM Trans. Graph. 2003;22(3): 641–50.

[6] Clarkson KL. Fast algorithm for the all nearest neighbors problem. In: Proceedings of the 24th IEEE annual symposium on foundations of computer science, Tucson, AZ, 1983, p. 226–32.

[7] Vaidya PM. An $O(n \log n)$ algorithm for the all-nearest-neighbor problem. Discrete and Computational Geometry 1989;4(1):101–15.

[8] Samet H. Foundations of Multidimensional and Metric Data Structures, Morgan-Kaufmann, San Francisco, CA, 2006.

[9] Guttman A. R-trees: a dynamic index structure for spatial searching. In: Proceedings of the ACM SIGMOD'84 conference. Boston, MA: ACM Press; 1984. p. 47–57.

[10] Hjaltason GR, Samet H. Distance browsing in spatial databases. ACM Transactions on Database Systems 1999;24(2):265–318.

[11] Roussopoulos N, Kelley S, Vincent F. Nearest neighbor queries. In: Proceedings of the ACM SIGMOD'95 conference. San Jose, CA: ACM Press; 1995. p. 71–9.

[12] Mount DM, Arya S. ANN: a library for approximate nearest neighbor searching. In: Proceedings of the second annual center for geometric computing workshop on computational geometry, electronic edn., Durham, NC, 1997. URL ⟨http://www.cs.duke.edu/CGC/workshop97.html⟩.

[13] Fleishman S, Drori I, Cohen-Or D. Bilateral mesh denoising. In: Proceedings of the SIGGRAPH'03 conference, vol. 22(3). San Diego, CA: ACM Press; 2003. p. 950–3.

[14] Weyrich T, Pauly M, Heinzle S, Keiser R, Scandella S. Post-processing of scanned 3d surface data. In: Proceedings of eurographics symposium on point-based graphics 2004, Eurographics Association, Aire-La- Ville, Switzerland, 2004. p. 85–94.

[15] Mitra NJ, Nguyen A. Estimating surface normals in noisy point cloud data. In: Proceedings of the 19th ACM symposium on computational geometry. San Diego, CA: ACM Press; 2003. p. 322–8.

[16] Bentley JL. Multidimensional binary search trees used for associative searching. Communications of the ACM 1975;18(9):509–17.

[17] Arya S, Mount DM, Netanyahu NS, Silverman R, Wu A. An optimal algorithm for approximate nearest neighbor searching. In: Proceedings of the 5th annual ACM-SIAM symposium on discrete algorithms. Arlington, VA, 1994. p. 573–82 (also Journal of the ACM 1998;45(6):891–923).

[18] Xia C, Lu J, Ooi BC, Hu J. GORDER: an efficient method for KNN join processing. In: VLDB'04: Proceedings of the 30th international conference on very large data bases. Toronto, Canada: Morgan Kaufmann; 2004. p. 756–67.

[19] Böhm C, Krebs F. The *k*-nearest neighbor join: Turbo charging the KDD process. In: Knowledge and information systems (KAIS), vol. 6. London, UK: Springer; 2004. p. 728–49.

[20] Berchtold S, Böhm C, Keim DA, Kriegel H-P. A cost model for nearest neighbor search in high-dimensional data space. In: Proceedings of the 16th ACM SIGACT-SIGMOD-SIGART symposium on principles of database systems (PODS). Tucson, AZ, 1997. p. 78–86.

[21] Böhm C, Krebs F. Supporting KDD applications by the *k*-nearest neighbor join. In: Proceedings of international conference on database and expert systems applications (DEXA), Lecture Notes in Computer Science, Vol. 2736. Springer, Prague, Czech Republic, 2003. p. 504–16.

[22] Ullman JD, Garcia-Molina H, Widom J. Database systems: the complete book, Upper Saddle River, NJ: Prentice Hall PTR; 2001.

[23] Roussopoulos N, Leifker D. Direct spatial search on pictorial databases using packed R-trees. In: Proceedings of the ACM SIGMOD conference, Austin, TX, 1985. p. 17–31.

[24] Floater MS, Reimers M. Meshless parameterization and surface reconstruction. Computer Aided Geometric Design 2001;18(2):77–92.

[25] Murio DA. The mollification method and the numerical solutions of Ill-posed problems. New York, NY: Wiley; 1993.

[26] Cook RL. Shade trees. In: Proceedings of the SIGGRAPH'84 conference. New York, NY: ACM Press; 1984. p. 223–31.

[27] Blinn JF. Simulation of wrinkled surfaces. In: Proceedings of the SIGGRAPH'78 conference. Atlanta, GA: ACM Press; 1978. p. 286–92.

[28] Alexa M, Behr J, Cohen-Or D, Fleishman S, Levin D, Silva CT. Point set surfaces. In: VIS'01: Proceedings of the conference on visualization '01. San Diego, CA: IEEE Computer Society; 2001. p. 21–8.

[29] Weisstein EW. Box-Muller Transformation, Math World-A Wolfram Web Resource, Champaign, IL, 1999, ⟨http://mathworld.wolfram.com/Box-MullerTransformation.html⟩.

[30] Sebastian TB, Kimia BB. Metric-based shape retrieval in large databases. In: Kasturi R, Laurendau D, Suen C, editors. Proceedings of the 16th international conference on pattern recognition, vol. 3, Quebec City, Canada: IEE Computer Society; 2002. p. 291–6.

[31] Turk G. Generating textures on arbitrary surfaces using reaction-diffusion. In: Proceedings of the SIGGRAPH'91 conference. Las Vegas, NV: ACM Press; 1991. p. 289–98.

[32] Garland M, Heckbert P. Surface simplification using quadratic error metrics. In: Proceedings of the SIGGRAPH'97 conference. Los Angeles, CA: ACM Press; 1997. p. 209–16.

[33] Pauly M, Gross M, Kobbelt LP. Efficient simplification of point sampled surfaces. In: VIS'02: Proceedings of the conference on visualization'02. Boston, MA: IEEE Computer Society; 2002. p. 163–70.

[34] Luebke D, Reddy M, Cohen J, Varshney A, Watson B, Huebner R. Level of detail for 3D graphics. San Francisco, CA: Morgan Kaufmann; 2003.

[35] Lloyd S. Least squares quantization in PCM. IEEE Transactions on Visualization and Computer Graphics 1982;28(2):129–37.

[36] Datar M, Immorlica N, Indyk P, Mirrokni VS. Locality-sensitive hashing scheme based on p-stable distributions. In: SCG 04: Proceedings of the twentieth annual symposium on computational geometry, Brooklyn, NY, 2004. p. 253–62.

[37] Schroeder W, Martin K, Lorensen B. Visualization toolkit. Englewood Cliffs, NJ: Prentice-Hall; 1996.