ORTHANT NEIGHBORHOOD GRAPHS
A Decentralized Approach for Proximity Queries in Dynamic Point Sets

Tobias Germer and Thomas Strothotte
Department of Simulation and Graphics, Otto-von-Guericke University of Magdeburg, Germany

Keywords: Dynamic Point Sets, Proximity Queries, Range Searching, Geometric Spanners, Particle Systems.

Abstract: This work presents a novel approach for proximity queries in dynamic point sets, a common problem in computer graphics. We introduce the notion of Orthant Neighborhood Graphs, yielding a simple, decentralized spatial data structure based on weak spanners. We present efficient algorithms for dynamic insertions, deletions and movements of points, as well as range searching and other proximity queries. All our algorithms work in the *local neighborhood* of given points and are therefore independent of the global point set. This makes ONGs scalable to large point sets, where the total number of points does not influence local operations.

1 INTRODUCTION

In computer graphics, many methods rely on dynamic point sets. One example are particle systems, where individual particles can be considered as points moving through space (Reeves, 1983). A more complex example are multi-agent systems, where each object has some complex behavior (Schlechtweg et al., 2005). A common task in these systems is to find all neighbors in a defined neighborhood or the nearest neighbor for a particle or agent. This paper introduces a novel method to efficiently handle such *proximity queries* in dynamic point sets.

Our goal is to develop a simple and efficient data structure that maintains dynamic point sets. It should provide fast access to the local neighborhood for each point. Moreover, it should support big point sets commonly occurring in particle or agent systems. Popular approaches for this problem include (hierarchical) space partitioning techniques like octrees or bucket grids. However, these methods are either inflexible, do not perform well in dynamic settings, or do not scale well for large point sets.

We present an alternative paradigm which provides a flexible, decentralized approach for proximity queries in dynamic point sets. The main idea is to use a very simple, graph-based data structure with low memory footprint. We provide efficient algorithms which act in the local neighborhood of the points. This makes them input and output sensitive, as well as scalable to large point sets. Therefore, local changes in the point configuration only result in local changes in the data structure. Local operations like searching all neighbors in a given radius or moving a point a small (local) distance do not depend on the total size of the point set. This makes our approach suitable for dynamic particle or agent systems, where typical movements are relatively small and local.

Our approach is novel and poses many unresolved questions. The goal of this paper is to introduce the basic ideas and to describe the principles of our algorithms. Due to space constraints, we have to omit a thorough analysis here. We also have to leave an evaluation and the application of our approach for future work. Finally, we restrict our problem in this paper to the 2D case, i.e., to planar point sets.

2 BACKGROUND

Tasks like locating points, finding their neighbors or maintaining dynamic point sets are a common problem in computer graphics and computational geometry. Numerous approaches have been introduced and analyzed, making spatial data structures an elaborate area of research. We restrict our treatment of related work to the most established techniques used in computer graphics.

Uniform space subdivision: A simple way to speed up proximity queries (Schlechtweg et al., 2005) or collision detection (Kim et al., 1998) is to divide the

Germer T. and Strothotte T. (2007).
ORTHANT NEIGHBORHOOD GRAPHS - A Decentralized Approach for Proximity Queries in Dynamic Point Sets.
In *Proceedings of the Second International Conference on Computer Graphics Theory and Applications - GM/R*, pages 85-93
DOI: 10.5220/0002082800850093
Copyright © SciTePress

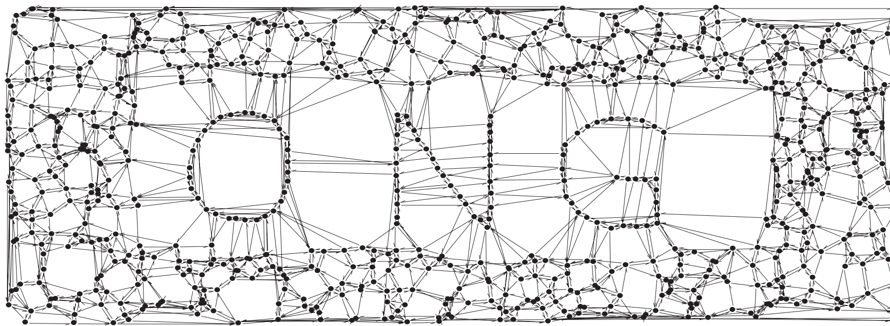


Figure 1: A complex example of an ONG for a point set with ca. 750 points.

space into equally sized buckets. Although simple and often effective, this technique has a relatively large memory footprint and doesn't work well in settings with varying search radius or inhomogeneous point distributions.

Quad- and octrees: A widely used technique to overcome these problems is to adaptively subdivide space by quad-trees (resp. octrees). An overview of different variations gives (Samet, 1990). (Boubekeur et al., 2006) present a new approach combining quad- and octrees.

BSP- and kd-trees are another class of popular techniques, supporting very flexible space subdivision and also working in higher dimensions (Bentley, 1990).

Bounding volume hierarchies: Instead of subdividing space, bounding volume hierarchies like OBB-Trees (Gottschalk et al., 1996) or BD-Trees (James and Pai, 2004) approximate the input data hierarchically to accelerate collision detection, for example.

Although providing fast (logarithmic) access to arbitrary leaf objects, all these tree-based approaches have a global (centralized) structure and therefore depend on the total number and structure of points. Instead, we seek for a data structure where local operations do not depend on the global structure. In general, tree-based approaches have also difficulties maintaining dynamic objects like moving point sets. Often the trees become inefficient or large parts have to be rebuild after a series of insertions or deletions.

3 ONGS

The main inspiration for our approach are the principles of swarm behavior and swarm intelligence (Bonabeau et al., 1999). In biology, there are various examples of large groups of animals like flocks of birds or schools of fish which rapidly move in global formations without collisions (Reynolds, 1987). However, a single animal has neither the mental nor physical ability to track all other animals and maintain a global view on the swarm as it steers through space. Instead, every animal only knows its *local neighborhood*, i.e., nearby animals and the local environment. Every animal acts solely based on this local information. However, the whole swarm is connected through various neighborhoods. This way, global information can be distributed using local structures. Therefore, global patterns can emerge.

We adopt this principle to build a data structure for point sets where each point tracks a limited, *local* neighborhood. This results in a decentralized data structure which provides local information. In addition, each point must have access to arbitrary large (global) neighborhoods, if needed. This way, every point *indirectly* knows the total point set. Thus, we have two main requirement:

- each point has a constant number of neighbors
- each point must have access to every other point

To meet the first requirement, we store pointers to all local neighbors for each point. The result is a directed geometric graph, where the vertices correspond to the points of the point set, and arcs represent the neighborhood relationships.

The second requirement implies that the graph has to be strongly connected. There must be a path (i.e., a chain of neighbors) connecting each vertex with every other vertex. To meet this requirement, we have to consider which vertices exactly are neighbors and how many neighbors are required for each vertex.

To answer this question we use results about "t-spanners" and "weak spanners" from (Fischer et al., 1997; Fischer et al., 1998; Fischer et al., 1999). We first review the relevant concepts. Geometric spanners are important data structures in computational geometry, because they approximate the complete graph using only $O(n)$ edges, where n denotes the number of vertices (Arya et al., 1995). In our context, this means that we can approximate global information about the

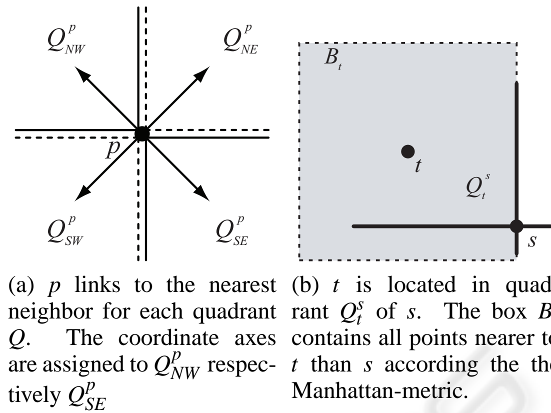
point set with local neighborhood relationships. A geometric graph $G = (V,E)$ is called *t-spanner*, if for each pair of vertices $(u,v) \in V$ there exists a path in E, which is no longer than t times the direct distance between u and v. Thus, the (relative) length of the path of any pair of vertices is bounded by the stretch factor t. The complete graph is obviously a t-spanner with $t = 1$. However, it has an out-degree of $n-1$ and therefore takes $O(n^2)$ space. Instead, we need a low and constant out-degree for our data structure to be output sensitive.

One way to construct a t-spanner is to divide the space around each vertex p into k cones and to create a directed edge from p to the closest vertex in each cone (Yao, 1982). It can be proved that the resulting graph is a t-spanner for $k > 6$ cones (Ruppert and Seidel, 1991). Fischer et al. improve this value to $k \geq 4$ by introducing “weak spanners”. However, these graphs only satisfy a weak spanner property. Here, not the path length between any vertex pair is bounded, but the distance from any vertex on the path to the start vertex. Note that this graph must be strongly connected. In the prove of this property for $k \geq 4$, Fischer et al. also show a way how to actually find a short path between any two vertices.

We use the weak spanner construction from Fischer et al. in a slightly adapted version. We divide the (planar) space around each point p into the four quadrants $Q_j^p, j \in \{NE,NW,SE,SW\}$ defined by the coordinate axes. We have to take care about the coordinate axes themselves and assign them to unique quadrants. Fischer et al. introduced a consistent scheme for this, as illustrated in figure 2(a). We assume that there are no coincident vertices. We use the Manhattan-metric $d_M(p,v) = |p_x - v_x| + |p_y - v_y|$ to find the nearest points $v_j \in Q_j^p$. The motivation to use the Manhattan-metric is explained in section 4.5. Finally, we store each v_j as a local neighbor for p. Figure 2(a) illustrates this concept. The resulting structure also generalizes to higher dimensions. Therefore, we call it “Orthant Neighborhood Graph” (ONG)[1]. This graph has appealing properties:

- *Constant Outdegree:* Each vertex has at most four local neighbors. A graph with n vertices has at most $4n$ edges.
- *Quadrant-based partition:* By aligning the cones with the four quadrants we can easily assign points to cones using coordinate comparisons. Employing only simple comparisons of constant numbers also makes our approach robust.
- *Simple metric:* The Manhattan-metric is simple and cheap to compute.
- *Weak spanner:* The resulting graph is strongly connected and has the weak spanner property.

[1]The concept of quadrants and octants generalized to arbitrary dimensions is called “Orthant”.



(a) p links to the nearest neighbor for each quadrant Q. The coordinate axes are assigned to Q^p_{NW} respectively Q^p_{SE}

(b) t is located in quadrant Q^s_t of s. The box B_t contains all points nearer to t than s according the the Manhattan-metric.

Figure 2: Basic construction of ONGs.

To verify that ONGs are strongly connected, we briefly sketch the main argument from the prove presented in (Fischer et al., 1998). Consider two vertices s and t as illustrated in figure 2(b). To construct a path from s to t, we consider the quadrant Q_t^s of s, to which t belongs. By definition, s must have a neighbor n for this quadrant. If this neighbor is t, we are done. Otherwise, there must be a neighboring vertex n closer to s than t. We show, that by recursively following the neighbor n, we incrementally get closer to t, until we reach t. Let $B_t = \{x \in \mathbb{R}^2 : d_{max}(t,x) \leq d_{max}(t,s)\}$ be the square defined by the maximum-metric $d_{max}(u,v) = max(|u_x - v_x|, |u_y - v_y|)$ (see figure 2(b)). Then, the neighbor n must be contained in B_t. There are three cases:

1. $d_{max}(t,n) < d_{max}(t,s)$: The neighbor is inside B_t and therefore closer to t. (see figure 3(a))
2. $d_{max}(t,n) = d_{max}(t,s)$ and $Q_t^s = Q_t^n$: s is on the border of B_t and t is in the same quadrant with respect to n_t^s. Then, n is still nearer to t according to the Manhattan-distance. (see figure 3(b))
3. $d_{max}(t,n) = d_{max}(t,s)$ and $Q_t^s \neq Q_t^n$: s is on the border of B_t and t has changed the quadrant with respect to n. Then, we do not come closer to t and B_t stays the same. However, in the next step of the path, the neighbor of n cannot be longer on the border of B_t, because our assignment of coordinate axes to quadrants does not permit this. Therefore, B_t will get smaller in the next step. (see figure 3(c))

This shows that the square B_t gets smaller or stays the same with each step along the path. However, cases 2 and 3 ensure that B_t can only stay constant for a finite number of steps. Therefore, the path will

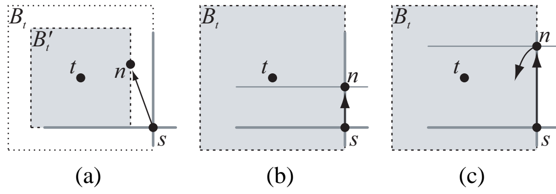


Figure 3: Cases for the location of neighbor n for vertex s.

finally reach t. We conclude this section with the following property of ONGs, which is important for the algorithms presented in the next section:

Corollary 1 *Given a vertex s in an ONG, any vertex t in this ONG can be reached by recursively following the neighbor of the quadrant, in which t is located.*

4 ALGORITHMS

Having introduced the fundamental structure of ONGs in the previous section, we now describe algorithms for the dynamic construction of ONGs. We first present the high level algorithms, then the low level procedures and finally an efficient algorithm for (localized) range searching using ONGs.

4.1 Insertion

First, we introduce two notations:

- $neigh_q^p$ denotes the neighbor of vertex p saved for quadrant q
- $quadrant_p(s)$ returns the quadrant of p in which vertex s is located

To insert a new vertex in an ONG, we have to insert and change certain arcs of the graph, so that the ONG stays consistent. We use the following algorithm:

Procedure `Insert(`*Vertex s, Vertex p*`)`.

Input: a starting vertex s, already inserted
Input: the new vertex p
1 **if** *ONG is empty* **then**
2   insert p as the first vertex;
3   **return**
4 **for** *each quadrant q of p* **do**
5   s_q = search some point in q, starting at s;
6     $neigh_q^p$ = nearest neighbor in q, starting at s_q;
7   **if** $neigh_q^p \neq \emptyset$ **then** $s = neigh_q^p$;
8 search all vertices r_i for which p is the new nearest neighbor;
9 **for** *each* r_i **do**
10   Quadrant $q = quadrant_{r_i}(p)$;
11     $neigh_q^{r_i} = p$;

We first check if the ONG is empty. In this case, p is the only vertex and there is nothing to change (lines 1-3). Otherwise, we search the nearest neighbors for each quadrant of p using the algorithm of section 4.4, and save them as neighbors (lines 4-7). By searching the nearest neighbors, we actually *localize p*, i.e., we determine its local neighborhood. Note that we need a given vertex s, where we begin our search for start vertices s_q in every quadrant, which are then used to initialize the nearest neighbor search. If we found a nearest neighbor, we use it as the starting point for the next quadrant, because it is probably close to the nearest neighbor in this quadrant.

Afterwards, we have to find all vertices r_i in the ONG, which have p as their new nearest neighbor (line 8). Section 4.5 describes an algorithm for this. Finally, we update these vertices and store p as their nearest neighbor in the according quadrant. Note that we first find *all* vertices r_i before changing the topology of the existing graph. If we would change the topology (i.e., store p as the new nearest neighbor) immediately after we found one r_i, the topology wouldn't be consistent anymore, breaking the assumptions for subsequent queries. Therefore, we strictly separate queries using the ONG from changing the ONG.

4.2 Deletion

To delete a vertex p from the ONG, we have to change all arcs pointing to p. We do this with the following algorithm:

Procedure `Remove(`*Vertex p*`)`.

1 search all vertices r_i for which p is a nearest neighbor;
2 **for** *each* r_i **do**
3     $q_i = quadrant_{r_i}(p)$;
4   s_i = search second nearest neighbor in q_i;
5 **for** *each* r_i **do**
6     $neigh_{q_i}^{r_i} = s_i$;

Note that we don't have to localize p this time, because the local neighborhood (i.e., the nearest neighbors) are already known. We first find all vertices r_i, for which p is a nearest neighbor (line 1). This is similar to line 8 of the insert algorithm and also detailed in section 4.5. For all vertices r_i we have to remove p as a neighbor and store the second nearest neighbor instead. Again, we have to separate the queries for the second nearest neighbor (lines 2-4) from the change of topology (lines 5-6). This results in a consistent ONG where no arc points to p anymore. Therefore, p can be removed.

4.3 Movement

Our goal for ONGs was to design a spatial data structure which can be used to maintain particle or agent systems. A typical property of such systems is that the entities (i.e., the vertices) are moving. We handle this action by simply deleting the vertex and re-inserting it at the new position:

Procedure `Move(`*Vertex p, Vector v*`)`.

1 Vertex s = some neighbor from p;
2 `Remove` (p);
3 move p according to v;
4 `Insert` (s,p);

This algorithm benefits from small movements of vertices. We take some (old) neighbor of vertex p (line 1) as the starting point for the re-insertion in line 4. If the new position of p is relatively close to the old position, the localization step of the insert procedure will be cheap (see next section). This way, the algorithm becomes *input sensitive*: local movements only traverse and change local parts of the ONG. The downside is that chaotic, global movements traverse very large parts of the ONG, degrading performance. However, particle and agent systems mostly exhibit small movements. Therefore, ONGs will be suitable for such systems.

4.4 Neighbor Searching

One of the first steps of the insert algorithm is the localization of the new point p, where we search for the nearest neighbors for each of its quadrants. This section presents a simple and effective algorithm for this.

In contrast to centralized data structures like quad- or kd-trees, ONGs do not provide a mechanism to quickly locate arbitrary points in the point set. Instead, we have to iteratively traverse the local neighborhood of certain vertices until we find the nearest neighbor for p. The idea is to take a starting vertex s and then "walk across" the graph in the direction of p, until no closer vertex can be found anymore. We need two more concepts for this algorithm:

- *Search regions*: A search region is a rectangular region representing the "undiscovered" space. Only in the search region new results can be found. If the search region is empty, we have found all result points. We can find all vertices in a search region by using corollary 1: given a vertex p, we can find all vertices in the search region by recursively following all neighbors assigned to the quadrants which intersect the search region.
- *Vertex flags*: To avoid loops when traversing the graph, we mark visited vertices with a flag, denoted as $flag_p$ for vertex p. Before finishing the algorithm, we have to clear the flags again.

We now present our algorithm `NearestNeigh` for finding the nearest neighbor to a point p (which is not yet inserted) in the quadrant according to s, starting at s. First, we save the quadrant q, for which we search the nearest neighbor (line 1). Then, we save the distance d of the nearest neighbor yet found (which is s, at the beginning the starting point). Afterwards, we set up a search region R, where all possible nearer neighbors to p could be found (line 4). Note that $R \supset \{x : x \in q \wedge d_M(x,p) <= d_M(x,s)\}$. We then search for nearer neighbors by calling `nnInternal` for each quadrant q_i of s. Note that we don't have to search in quadrant q, because all points in this quadrant must have a greater distance to p than s. If `nnInternal` finds a new neighbor, it returns *true* and we start our search again with the new, reduced search region. If no new nearer neighbor could be found, we are finished and return the last result point s.

Function `NearestNeigh(`*Point p, Vertex s*`)`.

Output: nearest Vertex to p in the quadrant of s
1 Quadrant $q = quadrant_p(s)$;
2 float $d = d_M(p,s)$; `// init distance`
3 **repeat**
4 SearchRegion R = (square centered at p with side length $2d$) $\cap\, q$;
5 set $flag_s$;
6 **for** *each quadrant* $q_i \neq q$ **do**
7 Vertex $n = neigh^s_{q_i}$;
8 **if** `nnInternal` *(n)* **then break**; `// for`
9 clear all flags;
10 **until** *no nearer neighbor found* ;
11 **return** s

12 **Local Function** `nnInternal` (*Vertex n*):
13 **if** $n = \emptyset \vee flag_n$ *is set* **then return** *false*
14 Quadrant $q_n = quadrant_p(n)$;
15 **if** $q_n = q \wedge d_M(p,n) < d$ **then**
16 $s = n$; `// new nearest vertex`
17 $d = d_M(p,n)$;
18 **return** *true*
19 set $flag_n$;
20 **for** *each quadrant* $q_i \neq q_n$ **do**
21 **if** *R cuts* $q_i \wedge$ `nnInternal` *(*$neigh^n_{q_i}$*)* **then**
22 **return** *true*
23 **return** *false*

The function `nnInternal` first checks if the current vertex n is in the right quadrant and is nearer to p than the current nearest vertex s (lines 15-17). If this is not the case, it recursively performs a simple depth-first search to find other vertices. Note that we only have to search in quadrants that cut the search region.

The algorithm described above can be improved

by a simple heuristic. Given the current vertex s (or n in `nnInternal`), chances are high that a new nearest neighbor can be found by following the opposite quadrant of q_s (respectively q_n), because in this quadrant the target point p is located. Therefore, we first search in these quadrants before searching in the remaining ones. This way, we quickly reduce the search region and follow a roughly linear path to the target point p (see figure 4).

If the initial starting point s is already close to the nearest neighbor, this path will be very short and only points in a local neighborhood of s resp. p will be traversed. Thus, our algorithm benefits from small movements and local insertions.

In line 4 of the remove algorithm (section 4.2), we search for the *second* nearest neighbor in a given quadrant. This can be done with a slightly adapted version of `nnInternal`. We only have to extend the condition in line 15 to neither accept p nor $neigh_q^p$ as a new nearest vertex. This way, the search is aborted after the second nearest neighbor has been found.

We also use a similar algorithm to find (arbitrary) points in a given quadrant, as required in line 5 of the insert algorithm. In this case, we set the search region to match the quadrant and use a search similar to `nnInternal` to find a point in this region.

4.5 Reverse Neighbor Searching

One step of the insert algorithm of section 4.1 is to search for all vertices r_i, for which a point p is the new nearest neighbor. We say that the vertices r_i *will reference* to p after the insertion. We use a similar step in line 1 of the remove algorithm (section 4.2), where we search all vertices which *are referencing* to a given vertex p. In general, this problem is called "reverse nearest neighbor searching" (Maheshwari et al., 2002). Here, we adapt the problem to report all vertices r_i, which have p as their nearest neighbor in *one* of their quadrants.

In the context of ONGs, the number of such vertices can be arbitrary large and depends on the vertex

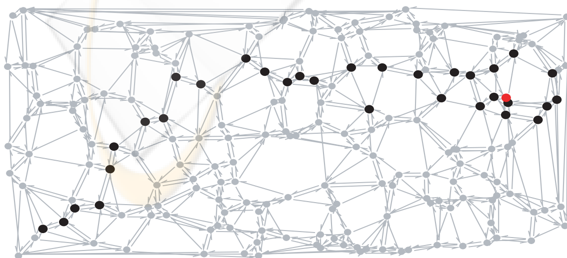


Figure 4: The black dots highlight the points visited during a nearest neighbor search. The path between the start point in the lower left and the target point in the upper right is roughly linear.

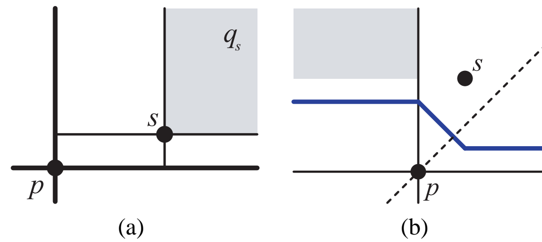


Figure 5: p cannot be the nearest neighbor for vertices in the shaded areas. The blue line in (b) marks the Voronoi-edge between s and p for the Manhattan-metric.

distribution in the local neighborhood around p (consider a vertex surrounded by a circle of other vertices). However, the average number of referencing vertices for each vertex in an ONG is (at most) four, since the out-degree of every vertex is (at most) four.

The idea for our reverse nearest neighbor algorithm is to use a breadth-first search (BFS) constrained by corollary 1 and the two following observations:

Corollary 2 *Let s be a vertex located in quadrant q_s of vertex p. Then, every vertex in the same quadrant q_s of s must be closer to s than to p (see figure 5(a)). Therefore, no vertex in this area can reference to p.*

Corollary 3 *Let s and p be two vertices and $v = s - p$ the difference between them. Without loss of generality, we assume that s is in the upper-right quadrant of p. If $|v_x| < |v_y|$, then all vertices in the upper-left quadrant of p above s are closer to s than to t. Analogously, if $|v_x| > |v_y|$, then all vertices in the lower-right quadrant of p on the right of s are closer to s than to t.*

This observation can be easily verified by considering the Voronoi diagram for the Manhattan-metric, as illustrated in figure 5(b), where all points above s are also above the Voronoi edge. Note that this observation is only possible with the Manhattan-metric, which was the main reason to use it for ONGs.

We can now present our reverse nearest neighbor algorithm (`ReverseNN`), reporting all vertices which have p as their nearest neighbor.

The algorithm begins by creating a search region for every quadrant (line 2). Initially, every search rectangle is equivalent to its quadrant and therefore on two sides open. Then, all neighbors of p are inserted into the BFS queue (lines 4-9). In addition, the search regions are reduced by calling the function `clipSearchReg` for every neighbor. This function employs corollary 3 to clip the search rectangles.

Afterwards, we process every vertex s from the queue (lines 10-21): If s has p as its nearest neighbor, it is reported as a result (line 14). Note that p can only

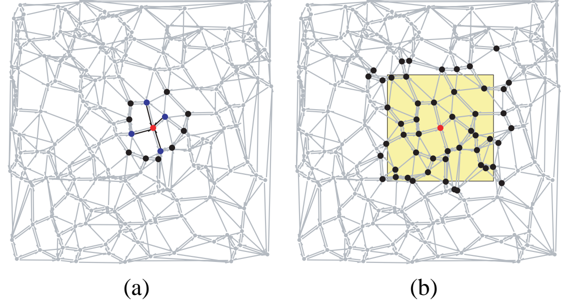
Procedure `ReverseNN(`*Vertex p*`)`.

```
1  List Q = empty vertex list;
2  SearchRegion R[4] = four open rectangles
   corresponding to the quadrants of p;
3  set flag_p;
4  for each quadrant q_i do
5      Vertex n = neigh^p_{q_i};
6      if n ≠ ∅ then
7          set flag_n;
8          clipSearchReg (n);
9          append n to Q;
10 repeat
11     Vertex s = pop front element from Q;
12     Quadrant q_s = quadrant_p(s);
13     Quadrant q̂_s = opposite(q_s);
14     if neigh^s_{q̂_s} = p then report s as a result;
15     for each quadrant q_i do
16         Vertex n = neigh^s_{q_i};
17         if n ≠ ∅ ∧ flag_n is not set
                   ∧ q_i ≠ q_s ∧ R[1...4] cuts q_i then
18             set flag_n;
19             clipSearchReg (n);
20             append n to Q;
21 until Q is empty ;
22 clear all flags;

23 Local Function clipSearchReg ( Vertex n ):
24   Quadrant q_n = quadrant_p(n);
25   Vector v = n − p
26   if |v_x| < |v_y| then
27       Quadrant q = q_n inverted in x-direction;
28       clamp R[q] in y-direction;
29   else
30       Quadrant q = q_n inverted in y-direction;
31       clamp R[q] in x-direction;
```

be the neighbor for s in the opposite quadrant of q_s. Then, we consider all neighbors of s which have not been visited yet and which correspond to quadrants intersecting one of the search regions ($R[1...4]$). Following corollary 2, we don't have to consider neighbors in q_s (line 17). After clipping the search regions, we include each such neighbor into the BFS queue (lines 18-20). We repeat these steps until the BFS queue is empty and no more referencing vertices could be found. The visited and resulting vertices for a typical example are illustrated in figure 6(a).

We use a variation of the algorithm above to search all vertices which will reference to a new vertex p after insertion. The only differences are line 5, where we take the nearest neighbors for p found by `NearestNeigh`, and line 14, where we check, if p is nearer than the current neighbor.



(a) (b)

Figure 6: The black dots highlight the points visited during a reverse nearest neighbor search (a) and range searching (b) for the red point. The shaded square in (b) represents the query region.

4.6 Range Searching

Finally, the main purpose of ONGs is to provide *proximity queries*. The nearest neighbor for any vertex (according to the Manhattan metric) can be simply found by comparing the four neighbors of each quadrant. Another common query in particle or agent systems is to find all points in a given radius. We formulate this as a (circular) range searching problem: given a vertex p, find all vertices nearer than a certain distance r. We approximate this problem by finding all neighbors in a square centered at p and having a side length of $2r$:

Procedure `RangeSearch(`*Vertex p, float r*`)`.

```
1  List Q = empty vertex list;
2  SearchRegion R = square centered at p with side
   length 2r;
3  set flag_p;
4  while p ≠ ∅ do
5      if p ∈ R then report p as a result;
6      for each quadrant q_i do
7          Vertex n = neigh^p_{q_i};
8          if n ≠ ∅ ∧ flag_n is not set ∧ R cuts q_i then
9              set flag_n;
10             append n to Q;
11     p = pop front element from Q;
12 clear all flags;
```

This algorithm implements a simple breadth-first search (BFS), starting at p. Using BFS for range searching with (weak) spanners was introduced by (Fischer et al., 1998). They use the (weak) spanner property to constrain the BFS to a certain region around p. In contrast to this approach, we use corollary 1 to constrain the BFS: we only have to continue the search in quadrants which intersect the search region R. This results in much fewer vertices we have to visit. In typical examples, there are only few visited vertices which are not contained in the query rectangle (see figure 6(b)).

Table 1: Timings for experiments with ONGs for point sets of different size. The timings in ms are averaged over 10 000 iterations.

Point set	1 000	10 000	100 000
Incremental build	0.031 s	0.367 s	6.141 s
Remove all points	0.027 s	0.309 s	5.694 s
Point movement	0.055 ms	0.059 ms	0.058 ms
Range searching	0.109 ms	0.111 ms	0.110 ms

Using this algorithm, we can also provide proximity queries for other metrics. For example, the Euclidean nearest neighbor to p can be found by searching all vertices nearer than the nearest neighbor n_M based on the Manhattan metric. We do this by performing a range search around p with radius $d_M(p, n_M)$ and comparing the distances of the result points according to the Euclidean metric.

5 DISCUSSION

In the following we give preliminary results for our approach. We have implemented all presented algorithms and data structures in C++. Figure 1 shows a complex ONG generated with our system.

Table 1 gives timings of some experiments on a 3GHz PC using our prototypical implementation, where the code was not optimized for speed. First, we inserted a large number of points with random distribution in arbitrary order. The table shows that these can be quickly incrementally inserted into an ONG. The incremental removal of all points is even faster, because no localization step (as explained in section 4.4) is needed. In the next experiment we compare the cost of locally moving a point in small and large point sets consisting of 1 000, 10 000, and 100 000 points. The timings for all three cases are nearly equal. Finally, we do a range search in small and large point sets. We adapt the radius, so that 100 points are found each time. Again, the timings are nearly equal. Therefore, the total number of points in the ONG does not influence the cost of local operations like local movements or range searching. This confirms the scalability of ONGs.

5.1 Conclusion

We have introduced ONGs, a new spatial data structure that supports proximity queries in dynamic point sets. The basis for ONGs are weak spanners, which ensure that storing the nearest neighbor for each quadrant results in a strongly connected graph. ONGs are decentralized in that all information is distributed on the whole point set. We presented different algorithms that work on the local neighborhood of given points. This allows dynamic insertions, deletions and movements of points as well as range queries *independent* of the size of the point set. Our algorithms are input and output sensitive: The cost of moving a point is low for small movements, but grows as it moves farther. Also, the cost of range queries depends of the number and the neighborhood of the result points. These properties make ONGs applicable to systems consisting of a large number of points, like particle or agent systems.

5.2 Future Work

There are many areas of future work and open questions for ONGs. The approach and the presented algorithms have to be analyzed and evaluated in more detail. A comparison with other techniques (like quad- or kd-trees) will show the usability of ONGs.

We want to adopt ONGs to 3D and higher dimensions. In principle, the presented algorithms and data structures also work in higher dimensions. However, the number of orthants grows exponentially with the number of dimensions, resulting in drawdowns in performance. The lowest known bound of the number of neighbors required for weak spanners in 3D is 8. However, this bound is not tight (Fischer et al., 1999). ONGs would benefit from a scheme that requires less cones and still produces weak spanners.

Another direction of future work is the kinetization of ONGs. Kinetic data structures store dynamic objects and explicitly model their motion (Basch et al., 1997). The idea is to only change the underlying structure if certain predicates change. This could save unnecessary updates of ONGs for small motions which do not change the graph topology.

Finally, we want to apply ONGs to different problems in computer graphics. For example, higher dimensional ONGs could be used for the broad phase of collision detection.

REFERENCES

Arya, S., Das, G., Mount, D. M., Salowe, J. S., and Smid, M. (1995). Euclidean spanners: short, thin, and lanky. In *STOC '95: Proc. 27th annual ACM symposium on Theory of computing*, pages 489–498.

Basch, J., Guibas, L. J., and Hershberger, J. (1997). Data structures for mobile data. In *SODA '97: Proc. 8th annual ACM-SIAM symposium on Discrete algorithms*, pages 747–756.

Bentley, J. L. (1990). K-d trees for semidynamic point sets. In *SCG '90: Proc. 6th annual symp. on Computational geometry*, pages 187–197.

Bonabeau, E., Dorigo, M., and Theraulaz, G. (1999). *Swarm Intelligence : From Natural to Artificial Systems*. Oxford University Press, USA.

Boubekeur, T., Heidrich, W., Granier, X., and Schlick, C. (2006). Volume-surface trees. *Computer Graphics Forum (Proceedings of Eurographics 2006)*, 25(3):399–406.

Fischer, M., Lukovszki, T., and Ziegler, M. (1998). Geometric searching in walkthrough animations with weak spanners in real time. In *ESA '98: Proc. 6th Annual European Symposium on Algorithms*, pages 163–174.

Fischer, M., Lukovszki, T., and Ziegler, M. (1999). Partitioned neighborhood spanners of minimal outdegree. In *Proc. 11th Canadian Conf. on Computational Geometry (CCCG'99)*, pages 47–50.

Fischer, M., Meyer auf der Heide, F., and Strothmann, W.-B. (1997). Dynamic data structures for real-time management of large geometric scenes. In *ESA '97: Proc. 5th Annual European Symposium on Algorithms*, pages 157–170.

Gottschalk, S., Lin, M. C., and Manocha, D. (1996). OBB-Tree: A hierarchical structure for rapid interference detection. *Computer Graphics*, 30(Annual Conference Series):171–180.

James, D. L. and Pai, D. K. (2004). BD-Tree: Output-sensitive collision detection for reduced deformable models. *ACM Transactions on Graphics (SIGGRAPH 2004)*, 23(3):393–398.

Kim, D.-J., Guibas, L. J., and Shin, S. Y. (1998). Fast collision detection among multiple moving spheres. *IEEE Transactions on Visualization and Computer Graphics*, 4(3):230–242.

Maheshwari, A., Vahrenhold, J., and Zeh, N. (2002). On reverse nearest neighbor queries. In *Proc. 14th Canadian Conf. on Computational Geometry*, pages 128–132.

Reeves, W. T. (1983). Particle Systems – A Technique for Modeling a Class of Fuzzy Objects. *Computer Graphics (Proceedings of ACM SIGGRAPH 83)*, 17(3):359–376.

Reynolds, C. W. (1987). Flocks, Herds, and Schools: A Distributed Behavioral Model. *Computer Graphics (Proceedings of ACM SIGGRAPH 83)*, 21(4):25–34.

Ruppert, J. and Seidel, R. (1991). Approximating the d-dimensional complete euclidean graph. In *Proc. 3rd Canadian Conf. on Computational Geometry*, pages 207–210.

Samet, H. (1990). *The design and analysis of spatial data structures*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.

Schlechtweg, S., Germer, T., and Strothotte, T. (2005). RenderBots—Multi Agent Systems for Direct Image Generation. *Computer Graphics Forum*, 24(1):137–148.

Yao, A. C.-C. (1982). On constructing minimum spanning trees in k-dimensional spaces and related problems. *SIAM J. Comput.*, 11(4):721–736.

SciTePress
Science and Technology Publications