Query Sphere Indexing for Neighborhood Requests

Nicolas Brodu

June 2007

Abstract

This is an algorithm for finding neighbors for point objects that can freely move and have no predefined position. The query sphere consists of a center location and a given radius within which nearby objects must be found. Space is discretized in cubic cells. This algorithm introduces an indexing scheme that gives the list of all the cells making up the query sphere, for any radius and any center location. It can additionally take in account both cyclic and non-cyclic regions of interest. Finding only the K-nearest neighbors naturally benefits from the query sphere indexing by running through the list of cells from the center in increasing distance, and prematurely stopping when the K neighbors have been found.

1 Introduction

Finding the neighbors of a given point is a general problem that is still a major topic of research. There exists a large variety of application needs, which may be classified in the following categories: 1. The data points are static or dynamic. 2. The query is approximate or exact. 3. All points should be returned or just the nearest ones. Static problems are commonly found for example in geolocalization systems [1] and they allow pre-sorting techniques, usually tree-based [2]. Approximate queries are usually a compromise for handling large dimensional data sets [3], with an application domain in particular for computer vision [4]. Graphics applications like mesh reconstruction from point clouds heavily rely on finding the K-nearest neighbors for each point [5], while games may need to find all neighbors of a given player within a certain radius for messaging purposes. The algorithm presented here concentrates on dynamic objects, for which the positions cannot be known beforehand and change with time. A typical application example of the algorithm is a multi-agent simulation where all the agents may move, and the neighbors for each agent need to be found at run-time for the AI [6]. More generally the algorithm is designed to handle the case where database update operations are performed as frequently as or more than the query operations. In the previous example an update is necessary for each object movement, while a query is performed only for the AI. This situation is exactly the opposite as for the geolocalization case or for large database approximate queries, where the typical usage is to ask for nearby objects but these objects are themselves relatively static over time.

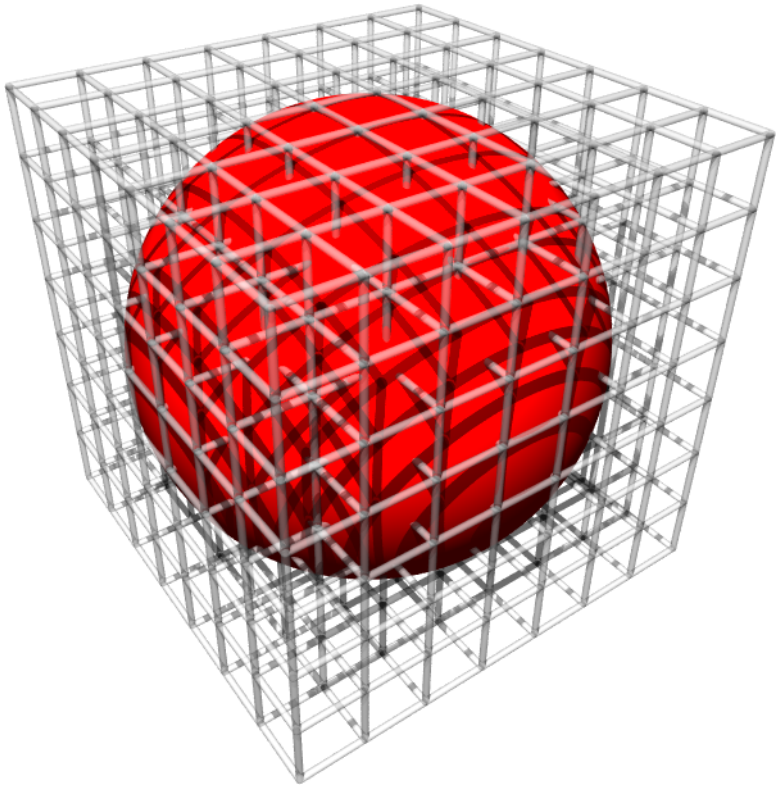
In such a scenario the tree-based techniques (like the kd-tree) perform badly: Changing the objects' positions degrades the tree properties, which then has to be rebuilt, and this is a costly operation (and more so than to the constant-time update presented below in any case). Maintaining the K-nearest neighbors list for mobile agents is still feasible with tree-based techniques, as in the algorithm proposed by Li *et al.* [7], though in this case the trajectories of each object within each other object's local basis needs to be computed. Even then tree updates are still unavoidable and certainly not constant-time. In fact, for dynamic objects situations, the bin-lattice spatial subdivision method has the best properties: constant-time update of the spatial database when an object moves, and amortized query time matching only a small fraction of all the objects on average [6].

The algorithm presented here can be seen as an improvement over the bin-lattice spatial subdivision method [6]. The case for three dimensions is described in this article and handled by the reference implementation, though the technique presented in this document is generic and applicable to other contexts as well. Source code is available online at the address listed at the end of this article.

2 Improving The Bin-Lattice Method

The neighborhood query problem consists in finding the objects within a given radius from a given center location, either all of them or only the K nearest. This defines what is called the query sphere in this document. The brute-force algorithm to answer the neighborhood query is $O(N)$: run through the list of all objects and compute their distance, then compare with the query radius to find the neighbors. Unfortunately when the query is repeated for each object, for example in the case of a multi-agent simulation where each agent wants to find its neighbors, then the brute-force algorithm becomes $O(N^2)$ and doesn't scale.

Let's now consider as in Fig. 1 a discretization of space consisting of a regular lattice of cubic cells. With the assumption that objects are represented by their position in space, each point object will be assigned to one cell, and only one. Each cell may contain as many as all the objects, or it can be empty. The idea behind the bin-lattice spatial subdivision method is to quickly eliminate all cells that are beyond range, with the con-



The center is the location for which neighbor objects should be found within a given radius. In this example 68 cells out of 343 do not intersect the query sphere and should ideally not be considered.

Figure 1: Query sphere intersection with a discretized space.

sequence that all objects within these cells are eliminated without computing their distance to the query center. Only the cells near the query center need to be processed. With the bin-lattice method, cells beyond the norm-1 cube (see Fig. 1) are not considered.

But the query sphere volume is $\frac{4}{3}\pi r^3$, and its bounding cube volume is $(2r)^3$, so the sphere fills only about 52% of the cube. This ratio is also the limit of the number of cells intersecting the sphere over the number of cells within the cube as the discretization size tends to 0. For a large distance query with respect to the cell discretization, up to nearly half the cells could thus be rejected. For higher dimensions, these sphere / cube volume ratios tend to decrease quickly, from about 31% in four dimensions to less than 1% for dimensions 9 and above [8]. The idea with the query sphere indexing technique is to not even consider the extra cube cells that do not intersect the query sphere, resulting in a gain of 68 cells out of 343 in Fig. 1. There is a final gain when the cost to set up the indexing scheme is lower than the cost of considering cells outside the sphere. Fortunately the run-time usage of the indexes is efficient by means of a few bit masking and shifting operations, and the list of the cells to process is implemented as a precomputed sorted array.

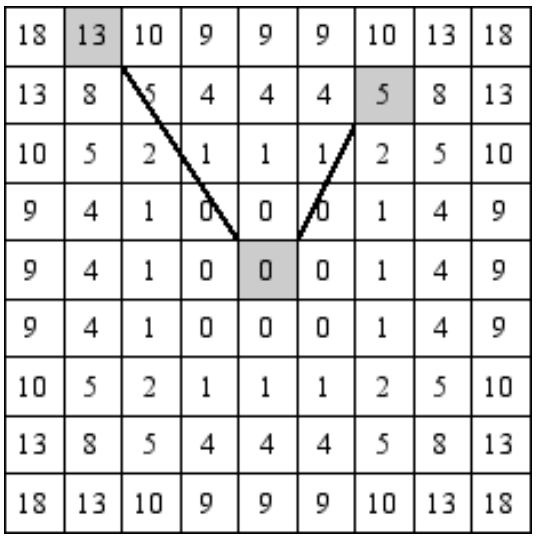
3 Query Sphere Indexing Scheme

3.1 Indexing cells by distance

Let's note the query center C, the query radius d, and the distance between C and an object X by x. The objects X in neighborhood are thus those for which $x \leq d$. With these notations, the rejected cells are at least strictly d away from C, with the distance between C and a cell defined as the minimum distance between C and the points in that cell. The other cells are either intersecting the sphere boundary or completely inside it.

The first step consists in building a lookup table based on the minimum distance two points in different cells can be. This table is precomputed only once at program startup, or it may be loaded from an external file. To build the table, let's consider an arbitrary cell L as the starting point, and assume for now this cell contains the query center location. If for a cell E, $\forall P \in E, \forall C \in L, \|P - C\| > d$, then the cell E can be rejected. By looking at the minimum squared distance between cells it is thus possible to pre-exclude some cells. Fig. 2 shows the minimum between cells in two dimensions, but the process can be extended to higher dimensions easily: In three dimensions there are 27 cells at distance 0, 54 at $d^2 = 1$, 36 at $d^2 = 2$, 8 at $d^2 = 3$, 54 at $d^2 = 4$, and so on. Figure 3 details the table building process.

Cells are then represented by their offset in each dimension from the center cell (see the next section). The list of all offsets for each minimum squared distance is maintained. Given a target query radius d, there is an integer $n = \lfloor d^2 \rfloor = \text{floor}(d^2)$, such that $n \leq d^2 < n+1$. As aforementioned, all cells that are at least strictly d away from the query center are rejected. Therefore, all cell offsets at $n+1$ and above are rejected. Consequently, it is sufficient to truncate d^2 so as to get the



Consider that the query centre is in the center greyed cell. A point P in that cell may be at minimum distance 0 from the query centre if $P = C$. Points in the cells surrounding the centre one may mathematically be at ϵ minimal distance from C, with ϵ depending on the floating point precision. Equating ϵ to 0 just increases the risk (with very low probability) that the cell is uselessly included, which does not affect correctness. Minimal squared distances to other cells are given in this array, with examples for $5 = 1^2 + 2^2$ and $13 = 3^2 + 2^2$.

Figure 2: Minimum squared distances between cells

At precomputation time:

```
L := the cell at the origin (0,0,0)
A :=[ ]                                          # Array of sets of offsets
for each cell S                                  # Region of interest
  d := inf {x : ∀P ∈ S, ∀C ∈ L, ||P − C|| = x}   # See Fig. 2
  A[d²] ← A[d²] ∪ {offset(S)}                    # d² in cell units, integer
  G :=[ ]                                        # Global offset array
  D :=[ ]                                        # Distance array
  i := 0
  for n := 0 to max(d²)
    if A[n] = ∅                                  # Missing d², see main text
      D[n] ← D[n − 1]                            # A[0] ≠ ∅ by construction
    else
      for each offset f ∈ A[n]
        G[i] ← f
        i ← i + 1
      D[n] ← i                                   # Global offsets for d² > n
```

At run-time:

```
n := ⌊d²⌋                                        # Truncate query distance
for each 0 ≤ i < D[n]                            # All cells from centre to edge
  f := G[i]                                      # Offset of the cell
  S := translate(C, f)                           # Cell S at offset f from C
  process(S)                                     # See the next sections
```

Figure 3: Building and using the global offset array

table lookup entry corresponding to that distance. The next step is to organize the cell offsets for a $\lfloor d^2 \rfloor$ table entry contiguously in memory just after the largest non-empty entry below $\lfloor d^2 \rfloor$ (see Fig. 3). Thanks to this layout the indexes of all the cells making up the query sphere are available in a simple array, sorted by increasing distance. This is particularly useful for K-nearest neighbor queries, as is explained in Section 5. Figure 3 recapitulates the algorithm so far.

3.2 Representing cells by their offsets

The space discretization is assumed to be finite, defined over a region of interest. This section relies on a power-of-two sized discretization in each dimension, for maximal performance. Non-power-of-two sizes could be implemented by extension, but this limitation is usually acceptable, and well worth the optimization it brings.

Thanks to the power-of-two assumption, each cell can be given an absolute linear index within the region of interest, corresponding to its binary representation. As an example, let's consider sizes of respectively 32, 16 and 8 in X, Y, and Z. This setup uses respectively

$B_X = 5$, $B_Y = 4$, and $B_Z = 3$ bits to store the position of a cell in each dimension. A cell at position 22 in X, 10 in Y and 3 in Z would be given the absolute index (in binary): 011_1010_10110, in ZYX order and with underscores added for clarity. This linear index is also the position in memory of that cell in a large array containing all the cells in the region of interest. This index is called the *packed location* of the cell in this document.

An *unpacked location* format is also introduced. The Y component is shifted to the left by the total number of bits $B_X + B_Y + B_Z$, so as to allow simultaneous (parallel) operations on all three components without fear of overflows. Example: Let's find the cell at offset $(-5, +4, +3)$ from the center at $(22, 10, 3)$ using the unpacked format:

```
      1010_00000_011_0000_10110 Centre cell at (22, 10, 3)
+     0100_00000_011_0000_11011 Offset: (−5, +4, +3)
=     1110_00000_110_0001_10001 Note the overflow here
AND   1111_00000_111_0000_11111 Mask out the overflow bits
=     1110_00000_110_0000_10001 Unpacked result: (17, 14, 6)
```

Packing this result allows to give the final index of the cell in the large memory array with a shift to put Y back in place, a binary OR, and a final mask:

```
≫     0000_00000_000_1110_00000 Shifted version of the result
+     1110_00000_110_0000_10001 OR'd with the result itself
=     1110_00000_110_1110_10001
AND   0000_00000_111_1111_11111
=     0000_00000_110_1110_10001 Mem. address for (17, 14, 6)
```

All the offsets mentioned in the previous section are stored in unpacked format, in the G array built as in Fig. 3. For example, with $B_X = B_Y = 3$, G could start in a two-dimensional scenario (see Fig. 2) by [(0, 0), (1, 0), (1, 1), (0, 1), (−1, 1), ...] hence in unpacked format adapted to 2D: [000_000_000, 000_000_001, 001_000_001, 001_000_000, 001_000_111, ...].

As a result of storing unpacked offsets in the G array the translation operation in the run-time loop of Fig. 3 involves only two elementary operations (+, AND), and three more elementary operations (shift, OR, AND) to get back the offset of the cell in memory.

The previous explanation works well only if the world is cyclic along all three components, due to the masking operations. For non-cyclic regions of interest the above presentation fails: The offset for X in the example was interpreted as -5 but could as well be interpreted as $+27$ on the 5 bits two-complement arithmetic. For a cyclic world this doesn't matter but for a non-cyclic one both values are distinct and need to be represented. Indeed, while the cells themselves are always attributed positive coordinates in the region of interest, the offsets from the sphere center may be negative. The solution is simple: Encode each non-cyclic component using an extra bit, so as to allow for offsets with full-range precision in both negative and positive domains. When an overflow is still observed this means the cell falls outside the region of interest, so the cell is ignored and a flag is set for later processing the "outside" cell only once after the main loop. The reader is invited to consult the reference implementation for more details, a link is given at the end of this document.

4 Optimizations

4.1 Using the sub-cell center location information

Building the cell offsets array involves the computation of the minimal possible distance between cells, $d := \inf\{x : \forall P \in S, \forall C \in L, \|P - C\| = x\}$ (see Fig. 3). The justification is given in Fig. 2, with the result of having 0-distance cells surrounding the center cell as is apparent in Fig. 2. However if the query center is located at distance x from the border within its cell, it is at distance $1 - x$ from the other side. Hence the d estimate is too conservative and there still are uselessly included offsets in the G array. In order to overcome this problem the solution is to consider the location of the query center within its cell, but unfortunately this information is only known at run-time.

The solution is to build separate pre-computed distance tables for each possible situation corresponding to locations of the query centre within its cell. The correct table is then selected at run-time with minimal and constant cost, and then the list of offsets for that particular situation is handled as before without additional cost. But in order to build these tables we need to investigate the relations between the involved distances.

Figure 4 shows the relation between the query center position and the distances to other cells, along each dimension (left) and how to generalize to multiple dimensions (right). Along each dimension the target cell should be rejected if $C + d < \lceil C \rceil + \lfloor d \rfloor$, where $\lceil C \rceil = \text{ceil}(C)$ is the smallest integer above or equal to C. Thus the target cell may be safely omitted if $d - \lfloor d \rfloor < C - \lceil C \rceil$, or in other words, $frac(d) < \zeta$. This result is generalizable to negative directions, in which case the target cell is on the left in Fig. 4 and then $\zeta = \text{frac}(C)$. So far cells that are diagonally placed from the center have not yet been considered. Fig. 4 (right) shows the situation in two dimensions. The base distance of a target cell is b, with b^2 an integer that is also the distance entry for the index array. Let's note t the true distance from the query center C to the cell. The cell can be rejected if $t > d$, or equivalently:

$t^2 > d^2$, since both distances are positive.

$\sum_{i=x,y} (b_i + \zeta_i)^2 > d^2$, with b_i and ζ_i positive distances along each axis.

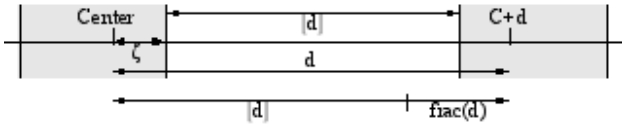
$\sum_{i=x,y} (b_i + \zeta_i)^2 > (\lfloor d \rfloor + f)^2$, with $f = \text{frac}(d)$.

$$b^2 + \zeta^2 + 2\sum_{i=x,y} b_i \zeta_i > \lfloor d \rfloor^2 + f^2 + 2 \lfloor d \rfloor f \qquad (1)$$

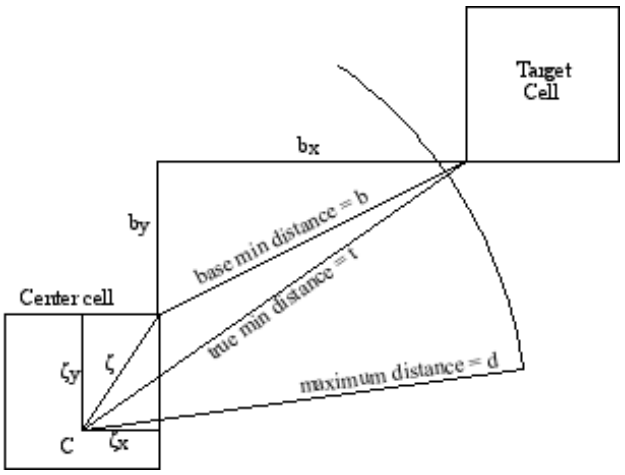
Re-using the previous condition for the rejection in one dimension, let's assume that $f < \zeta_i$ for each direction i. Let's additionally assume that $b \geq \lfloor d \rfloor$. Then by direct application of the assumptions in two dimensions:

$$b^2 + \zeta^2 + 2\sum_{i=x,y} b_i \zeta_i > \lfloor d \rfloor^2 + f^2 + 2f \sum_{i=x,y} b_i \qquad (2)$$

But, thanks to the triangular relation in Fig. 4 (right), $b_x + b_y \geq b$, and with the previous assumption $b_x + b_y \geq \lfloor d \rfloor$. Since $f > 0$ by definition, the set

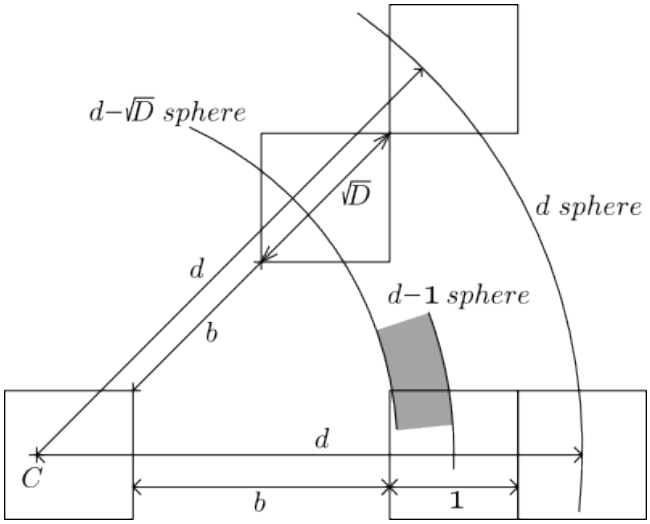


The center cell is on the left, the target cell on the right. d is the query distance. $\lfloor d \rfloor = \text{floor}(d)$ is the largest integer below or equal to d. It is also the distance between the two cells. $\text{frac}(d) = d - \lfloor d \rfloor$ is the fractional part of the distance. ζ is the distance from the center to the cell edge in the direction of the target cell.



Considering the query center position inside the center cell in multiple dimensions allows to reject cells at maximum distance when $b \leq d < t$.

Figure 4: Using the sub-cell location of the query center



Cells with $b < d - \sqrt{D}$, with D the dimension, are below one cube diagonal of the maximum distance and are always included. Cells below $d - 1$ along the main directions are also always included. In practice the run-time check for rejecting a cell is only applied for cells with $b > d - 1$: Some cells for which b is in the greyed zone escape rejection, but the test is not uselessly applied to the cells below $d - 1$ along each axis.

Figure 5: Included cells and tight query sphere bounds

of chosen assumptions makes Eq. 2 also satisfy Eq. 1 and the cell can be rejected. A similar argument holds in higher dimensions.

The specialized tables aforementioned may now be precomputed. In each direction i there is the possibility that $f < \zeta_i$ or not. This gives for three dimensions $2^6 = 64$ combinations, leading to as many specialized offset tables where the cells satisfying Eq. 2 are not included. These tables should only be used for $b \geq \lfloor d \rfloor$, but that's easy to ensure: the main table is run from sphere center to edge in increasing distance. The specialized tables are used as soon as $\lfloor d \rfloor$ is reached and no sooner. The precomputation can now rely on the fact that since the tables are only used for entries b^2 such that $d \geq b \geq \lfloor d \rfloor$, then by definition of $\lfloor d \rfloor = \text{floor}(d)$, $\lfloor d \rfloor = \lfloor \sqrt{b^2} \rfloor$ for each table entry b^2: $\lfloor d \rfloor$ is known at precomputation time even if d is not.

The query sphere has now been covered by cells with a much tighter bound than the norm-1 cube of the basic bin-lattice algorithm, and at precomputation time. However this covering is not yet optimal: it may be that some cells satisfy Eq. 1 but not Eq. 2. These cells will not be pre-excluded, but could still be rejected at run-time at the cost of an additional check. Fortunately, all the b_i distances in Eq. 1 correspond to the parts of the offset representation and they are thus available at run-time. It is then just a matter of adding the ζ_i and testing for $t^2 > d^2$ to decide whether to reject the cell or not. Figure 5 shows how this run-time test allows to tighten the query sphere coverage.

Then, as for the bin-lattice algorithm, objects in the cells that are still present so far are individually tested for rejection. This induces a cost that is proportional to the cell load: the average number of objects present in each cell. A final optimization is to unconditionally include all objects for cells below $d - \sqrt{D}$ (see Fig. 5). Indeed, in that case, the cells are entirely within the query sphere, and so are the objects within these cells.

5 Benchmarks And Influencial Factors

The benchmarks show the strengths and weaknesses of the new algorithm compared to competing techniques. The following methods are identified:

Cube — The bin-lattice algorithm described in [6] and which is further adapted to cyclic worlds.

Sphere — The query sphere indexing scheme presented in the first part of this document.

Non-empty cells list — The list of all non-empty cells is maintained and run through when looking for neighbors. Cells in that list are then tested for rejection because they are too far, quickly eliminating all objects in these cells in that case.

Brute-force — A loop through all objects is performed so as to find the neighbors of a given point. This method might still be occasionally faster than the others due to very low setup costs, good locality of references, and minimal run-time tests apart from the distance computations.

kd-tree — A three-dimensional kd-tree is used to handle the queries, so as to verify the claims in the introduction that a tree-based technique is not the best choice for this dynamic setup.

Space is discretized with $B_X = B_Y = B_Z = 4$ ($16 \times 16 \times 16$ cells). The number of objects and world wrapping conditions are variable. Objects are placed randomly in the region of interest. Each object is moved, and then each object finds all its neighbors within a predefined distance. This move/query operation is repeated 30 times and the performance of each method is measured over the whole operation. The original bin-lattice cube triple-loop performance is used as the baseline so the ratios of the other methods over that baseline are computed. This allows to show in a synthetic way where the new algorithm performs better than the other techniques.

Figure 6 shows the benchmark results for the non-wrapping world case. As is apparent on Fig. 6 the sphere indexing technique performance compared to the cube technique increases with the number of objects per cell. For small distances ($d = 0.8$ cell units) the sphere indexing does on average process marginally less cells than the cube. This gain outweights the sphere method setup costs when the cell load average is high enough, when there is an advantage of rejecting a cell without individually checking its objects. For large distances some cell offsets may fall outside the region of interest and are uselessly processed. Since the bin-lattice method always clip the query cube it doesn't have all these "outside" accesses, but nevertheless processes more "inside" cells than the sphere does. So depending on the load ratio, one or the other methods may be faster for large distances.

Running through the non-empty cells list allows to quickly reject some cells that are too distant. When there are many objects, these rejected cells give an advantage to the non-empty cells list over the brute-force method. When there are few objects, then the brute-force method low setup costs may be advantageous. Whether any of the brute-force or non-empty cells list methods has an advantage over the cube or the sphere method depends on the machine hardware specifics and the algorithms implementation, as well as the application. For example, when objects are concentrated on a few regions and not uniformly distributed as in the benchmarks, many of the cells considered by the bin-lattice cube or the indexed sphere techniques may be empty and then the non-empty cells list would cover less volume.

The kd-tree technique suffers from the object movements as mentioned in the introduction. Actually, for these tests, since all objects move it is faster to rebuild the whole tree after all objects have moved than to perform individual remove/insert on each position update. Even then the constant-time update cost for the move operation of the other methods makes the kd-tree behind in terms of performances.

The situation for wrapping worlds is even more advantageous to the new method. Indeed, there is no outside cell in an all-wrapping world, so the sphere method does not suffer from the aforementioned "outside index" penalty. This is apparent on Fig. 7 which reports the results of the same benchmarks as before but for a cyclic world in every dimension. The kd-tree technique has not been adapted to the cyclic case for these tests so the corresponding results are not available here. When the world is fully cyclic the gain of the new method over the cube increases with cube/sphere volume ratio, hence with the distance, up to the maximum of half the world size. At that point since the world is cyclic the query cube covers the whole world and its volume does not increase anymore, while the sphere covers more and more of this finite volume. In the end, when the query distance covers the whole world, it becomes more advantageous to use the non-empty cells list or the brute-force method due to their simplicity.

The K-nearest neighbors query problem is generally best handled by the sphere technique. As soon as K neighbors are found potentially closer candidates are necessarily within radius equal to the current furthest found neighbor distance d_K. Thanks to the sorted order of the offsets from center to query sphere edge, the sphere algorithm may prematurely stop as soon as d_K is reached: all cells above d_K are rejected. This is clearly apparent in Fig. 8 , taking the worst-case of Fig. 6, as the sphere technique maintains a constant processing time whatever the query distance once the neighbor is found. However for small distances not enough cells are rejected this way, so the situation is similar to the main benchmarks, and the cube method may be faster.

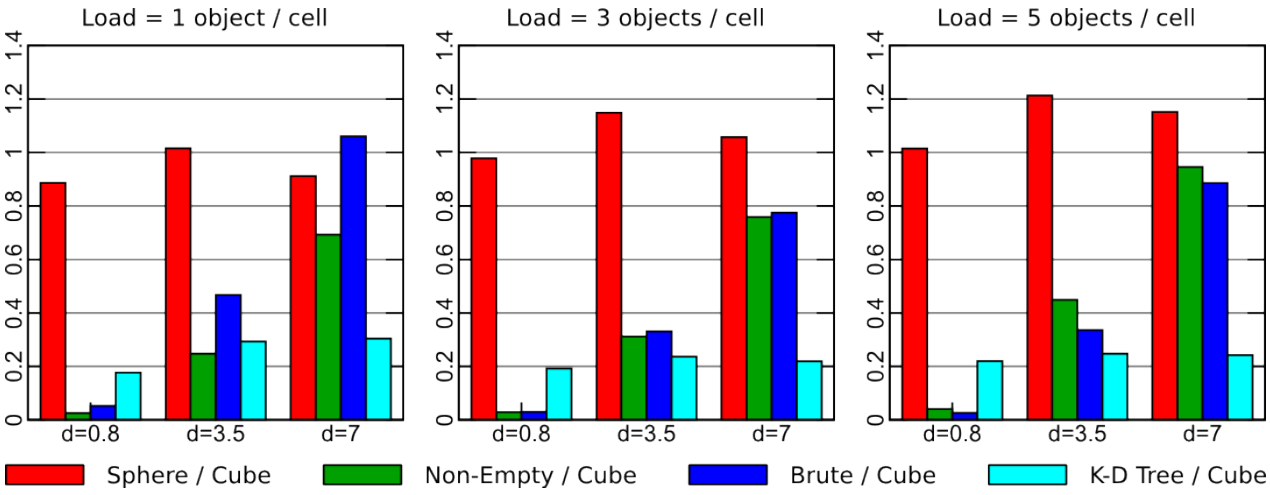


Figure 6: Non-wrapping world case, ratios of each query method performance over the bin-lattice cube one.

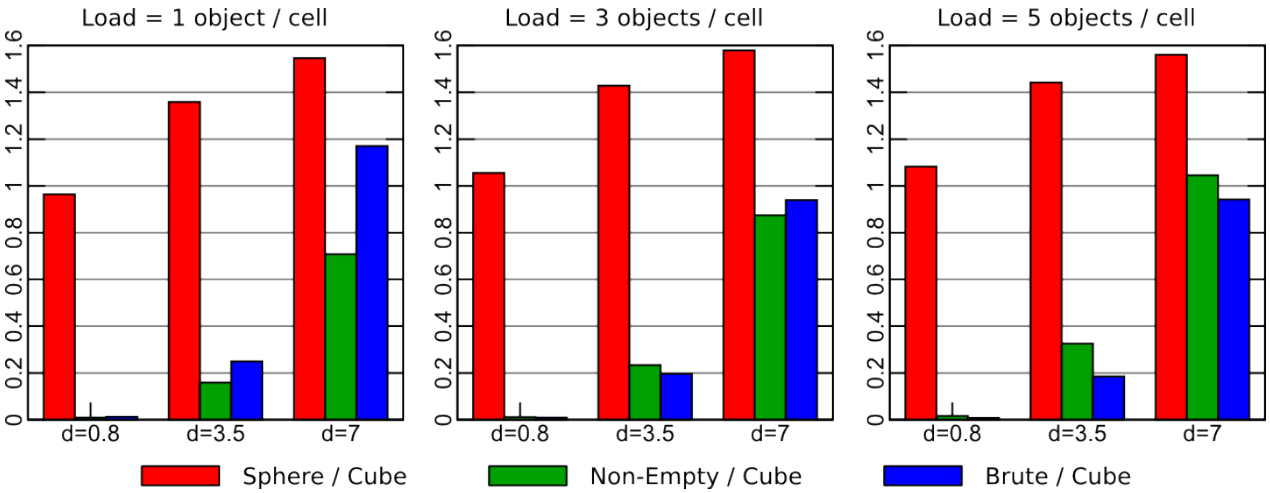
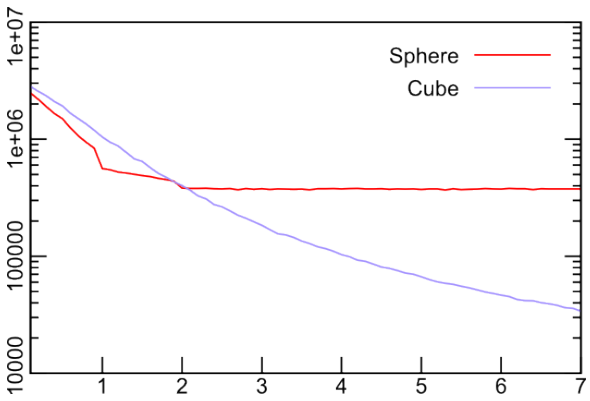


Figure 7: Wrapping world case, ratios of each query method performance over the bin-lattice cube one.



The performances for the sphere and cube techniques are plotted against the query distance in cell units, for 1 object/cell in a non-wrapping world (see Fig. 6). Each object is asked to find its nearest neighbor. The Sphere technique may prematurely stop once the neighbor is found, and keeps a constant processing time whatever the query distance in that case.

Figure 8: Nearest neighbor query performance vs. query distance.

6 Discussion And Automatic Method Selection

The algorithm presented in this document is especially well adapted to situations including a large number of objects. The wrapping worlds that are common in multi-agent simulations would benefit most from this algorithm as well. It is also well suited to problems like signaling and communication, where all agents in sight must be contacted regardless of their distance. The K-nearest neighbors finding problem also benefits directly from the spherical indexing, with the ability to early stop when the neighbors are found. For static environments with fixed object positions, some other methods like the kd-tree may be more efficient. But the new algorithm may be a good choice for dynamic situations where the objects move and the tree-based techniques

require costly updates. To sum up, depending on the configuration, the new algorithm may provide appreciable gains over the competing techniques (ex: nearly 60% improvement in one of the above benchmarks). However it is worth checking for a particular application which of the techniques really performs better, especially for small query distances.

The reference implementation features an automatic method selection routine in order to try to detect the most efficient method to apply to a given situation. Each method cost is quickly estimated using the query distance, the cell load ratio, the world size and the number of non-empty cells. These cost estimates are further weighted by user-specified factors so as to allow tuning to a specific architecture or application. Once weighted, the method with least estimated cost is then selected and processed as usual. The run time taken by the automatic selection feature is low (precomputations can be done), but that cost can nonetheless be avoided by forcing the usage to one specific query method if needed. Used properly, the various query techniques and the automated selection tool allow to maximally benefit from the reference implementation.

Acknowledgments

Financial support was provided by the EADS Corporate Research Center, in cooperation with the French Ministry of Foreign Affairs.

References

[1] Jagan Sankaranarayanan, Houman Alborzi, Hanan Samet (2005), Efficient Query Processing on Spatial Networks. 13th ACM International Symposium on Advances in Geographic Information Systems.

[2] H. Samet (1995), "Spatial data structures", Modern Database Systems: The Object Model, Interoperability, and Beyond, W. Kim, Ed., Addison-Wesley/ACM Press, pp 361-385.

[3] P. Indyk and R. Motwani (1998). Approximate nearest neighbors: towards removing the curse of dimensionality. In STOC, pages 604-613.

[4] Ting Liu, Andrew Moore, Alexander Gray, Ke Yang (2004), An Investigation of Practical Approximate Nearest Neighbor Algorithms, NIPS conference.

[5] Jagan Sankaranarayanan, Hanan Samet, Amitabh Varshney (2007), A fast all nearest neighbor algorithm for applications involving large point-clouds. Computers & Graphics 31:157-174.

[6] C. W. Reynolds (2000), "Interaction with Groups of Autonomous Characters", Game Developers Conference 2000, proceedings, pp 449-460.

[7] Yifan Li, Jiong Yang, Jiawei Han (2004), Continuous K-nearest neighbor search for moving objects. IEEE International conference on Scientific and Statistical Database Management, pp 123-126.

[8] E. W. Weisstein "Hypersphere". MathWorld, http://mathworld.wolfram.com/Hypersphere.html

[9] http://en.wikipedia.org/wiki/SWAR, and http://aggregate.org/SWAR/

Web information

Project web site with source code: http://nicolas.brodu.free.fr/en/programmation/neighand/
Author email address: nicolas.brodu@free.fr