Nearest Point Query on 184M Points in E3 with a Uniform Grid

W. Randolph Franklin*†

Abstract

NEARPT3 is an algorithm and implementation to preprocess more than 10^8 fixed points in E^3 and then perform nearest point queries against them. With fixed and query points drawn from the same distribution, NEARPT3's expected preprocessing and query time are $\theta(1)$ per point, with a very small constant factor. The data structure is a uniform grid in E^3, typically with the same number of grid cells as points. The storage budget, in addition to the space to store the points themselves, is 4 bytes per grid cell plus 4 bytes per point. NEARPT3 has been tested on the UNC complete powerplant, on the largest ply datasets in the Georgia Tech Large Geometric Models Archive, and the Stanford Digital Michelangelo Project Archive. The examples with up to 30,000,000 points can be processed on a laptop computer, and the others, the largest of which is St Matthew with 184,088,599 points, on a Xeon.

1 Introduction

Finding the closest fixed point to a query is a common primitive operation in applications such as surface fitting and intersecting. The prior art includes various data structures and algorithms for variants of nearest neighbor searching. The cost of a Voronoi diagram in E^3 is data dependent, and runs from $\Omega(N \log N)$ to $O(N^2)$ in time and space for preprocessing, with each query costing $\theta(\log N)$, [9]. Range trees cost $\theta(N \log N)$ time to preprocess, with each query also costing $\theta(\log N)$. ANN (Approximate Nearest Neighbors) is a C++ library for approximate and exact nearest neighbor searching in E^d, allowing a variety of metrics, implemented with several different data structures, based on kd-trees and box-decomposition trees, [2]. Another method, which is also data dependent, assumes that successive queries are close, so that one can efficiently traverse the dataset to the next answer. [10] summarizes various methods. [5] has a recent implementation with kd-trees that may be even faster than the current version of NEARPT3.

More general algorithms and data structures tend to be bigger than NEARPT3, which is optimized specifically for the L_2 metric in E^3, although its ideas would generalize. NEARPT3, which uses a *uniform grid*, [1, 4], appears to be the only method that enthusiastically rejects hierarchical data structures and search techniques. Trees and subdivision searching are much more robust against the kind of adversarially chosen input that would force NEARPT3's query time up to $\theta(N)$. However, those data structures are so much larger that they cannot process the data sets used in this paper. Also, their $\theta(\lg N)$ query time makes them much slower for many large datasets where NEARPT3's query time is $\theta(1)$. Also, extreme data unevenness also forces hierarchical data structures to have many levels, so that they are also slower.

This paper is only a brief summary of NEARPT3. For more details of the implementation and test results, and the source, see [3].

2 Algorithm

NEARPT3 has three stages, as follows.

Antepreprocess: This step generates part of the source code that will be included in `nearpt.cc` when it is compiled. This is a table of cells sorted by distance from the origin. *(Step 1)* Generate the coordinates (x, y, z) of all grid cells with $0 \le x \le y \le z \le R$ for some fixed R, say 100. *(2)* Sort them by $\sqrt{x^2 + y^2 + z^2}$. *(3)* Pass down the list in order. For each cell c, find the last cell, s_c, whose closest point to the origin is at least as close as the farthest point of c. Call s_c the *stop cell* of c. Since the stop cells are monotonically increasing, all this requires only one pass down the cell list. The point is that if a point has been found in c, we have to continue searching as far as s_c to be sure of finding any closer points. *(4)* Write the sorted list of cells and stop cells, in the form of a C++ variable initialization, to a file that `nearpt.cc` includes when compiled.

Preprocess: Here the fixed points are built into the data structure. *(1)* Compute a uniform grid resolution, G from the number of fixed points, N_f or get it from the user. A reasonable value is $G = r \sqrt[3]{N_f}$, for $1 \le r \le 3$. The default is $r = 1.6$. *(2)* Allocate a uniform grid with one word per cell, to store a count of the number of

*Rensselaer Polytechnic Institute, Troy NY 12180, USA, mail@wrfranklin.org

†This research was supported by NSF grant CCR-0306502, and by DARPA and NGA under Geo*. We are grateful to be able to use datasets from the Stanford University Computer Graphics Laboratory, including the Stanford Digital Michelangelo Project Archive, Georgia Institute of Technology's Large Geometric Models Archive, and the University of North Carolina's UNC Chapel Hill Walkthru Project.

points in each cell. *(3)* Read the fixed points for the first time, determine which cell of the uniform grid each point would fall in, and update the counts. *(4)* Allocate a ragged array for the uniform grid, with just enough space in each cell for the points in that cell. (A ragged array contains storage for the points plus a dope vector pointing to the first point of each cell. The total variable storage is one word per cell, plus the storage for the points.) *(5)* Transform, in place, the array of point counts into the dope vector, so it can occupy the same space as the array of points count. *(6)* Read the fixed points a second time, again computing the cell that each falls into. This time, store each point in its proper cell. (The goal is to minimize both the storage used and the number of storage reallocations. Storage reallocations become especially costly as the program's virtual memory approaches the computer's available real memory. Our experience is that a program's performance drops off dramatically when its virtual memory size is even slightly over the available real memory, even if its working set size is still smaller.) (A possible alternative would be to use a linked list for the points in each cell. However, the space used for the pointers would be significant, and the points in each cell would be scattered throughout the memory, which might reduce the cache performance.) (Another alternative would be to use a C++ STL vector, which reallocates its storage as it grows. Our experience finds this to be very suboptimal. In addition, our version of STL restricts vectors to a maximum size of 2GB, which is inadequate.) (A better alternative for grids where almost every cell is empty would be a hash table. Then, an empty cell would occupy no space at all, so that larger grids would be feasible.)

Query: This reports the closest fixed point to q, a query point. *(1)* Determine which cell, c, contains q. *(2)* If c contains at least one fixed point, then check all cells in the $5 \times 5 \times 5$ block centered on c for the closest fixed point and return it. Since the cells in each row of the grid are stored contiguously, the fixed points in one row of 5 cells in the block are checked in one sweep, so that any empty cells in the row take no time. If the position of q inside c were taken into account, it would be unnecessary to check all 124 other cells in the block. That is a possible future optimization. If the fixed and query points are selected from the same distribution, then c almost always contains a fixed point. *(3)* If c did not contain at least one fixed point, then do the following. Use the precomputed sorted cell list to spiral out from c until a cell with at least one point is found. (In the rare case that no cell with any fixed point is found, then exhaustively check every fixed point.) For each cell with coordinates (x, y, z) in the sorted cell list, up to 47 other reflected and rotated cells are derived, such as $(-x, z, -y)$. If any coordinate is zero, or any two are equal, there will be fewer other cells. (It would be possible to do this reflection, rotation, and duplicate deletion in the prepreprocessing stage. This would cause the sort cell list to be almost 48 times as large. It might be expected that this would reduce the query time because that code would have fewer conditionals, which should make it more optimizable. However, when we tried this, the time did not change.) Stop spiralling out at c's stop cell. This spiralling process is overly conservative since it ignores the location of q inside c. That is another possible future optimization.

The time analysis is as follows. Assuming that the total number of grid cells is linear in the number of fixed points, preprocessing time per point is $\theta(1)$. Assuming that the query points are selected from the same distribution as the fixed points, the neighborhood of any query point is independent of the size of the dataset, and so the query time must also be $\theta(1)$.

3 Tests, Including Comparison to ANN

We implemented NEARPT3 in C++ under SuSE Linux. The hardware was either a 2002-vintage IBM T30 Thinkpad laptop computer with 768 MB of memory, a 1600 MHz Pentium 4 Mobile CPU, or else a 2.4GHz Xeon with 4GB of memory. The software was Intel's icpc 8.1 C++ compiler, with aggressive optimizations enabled. g++ also works. We report in [3] on every large dataset that we tested; there are no bad cases omitted.

Our test data included these datasets: `uniXXX`, uniform i.i.d. sets of 10^5 to 10^8 random points, `blade`, `bone6`, `dragon`, and `hand` from the Georgia Institute of Technology's Large Geometric Models Archive, [11], the complete powerplant from the University of North Carolina's UNC Chapel Hill Walkthru Project, [12], `bunny` from Stanford University Computer Graphics Laboratory, [7], and `david` and `stmatthew` from the Stanford Digital Michelangelo Project Archive, [6].

For each test, the statistics described at the top of Table 1 were recorded. We also recorded histograms of the number of points per cell and number of cells checked per query. Table 1 lists the tests run on a 2.4GHz Xeon with 4GB of memory, only 3GB of which could be used by any one process. The laptop could run examples up to $N_f = 30M$, using 2–3 times the CPU as the Xeon.

What is the appropriate grid size to use? The answer is not critical since the time changes only slowly. Figure 1 shows the `bone6` dataset run on the Xeon with many grid sizes. The program default is $1.6\sqrt[3]{559636} = 131$. The table shows that the optimal grid size depends on the number of queries to be performed.

The classic nearest point algorithm in E^3 is ANN 0.2 (Approximate Nearest Neighbors) [8]. ANN was compiled with the defaults. Running it required no data

data	N_f	G	total time (sec)	init time (sec)	pre-processing time (sec)	query time (sec)	pre-processing time per fixed pt (μsec)	time per query pt (μsec)
bunny	25947	46	0.08	0.	0.010	0.070	0.385	7.
hand	317323	108	0.50	0.050	0.150	0.300	0.473	30.
dragon	427645	120	0.53	0.060	0.220	0.250	0.514	25.
bone6	559636	131	0.53	0.070	0.250	0.210	0.447	21.
blade	872954	152	0.68	0.110	0.400	0.170	0.458	17.
uni1m	1000000	160	1.46	0.120	0.940	0.400	0.940	40.
powerplant	5413053	280	63.19	0.980	2.810	59.400	0.519	5940.
uni10m	10000000	344	12.83	1.270	11.130	0.430	1.113	43.
david	28158109	486	20.41	3.400	13.100	3.910	0.465	391.
uni30m	30000000	496	41.69	3.880	37.350	0.460	1.245	46.
uni100m	100000000	742	150.50	13.010	137.020	0.470	1.370	47.
stmatthew	184088599	568	160.08	23.750	128.710	7.620	0.699	762.

Table 1: Tests Run on the Dual 2.2 GHz Xeon with 4MB

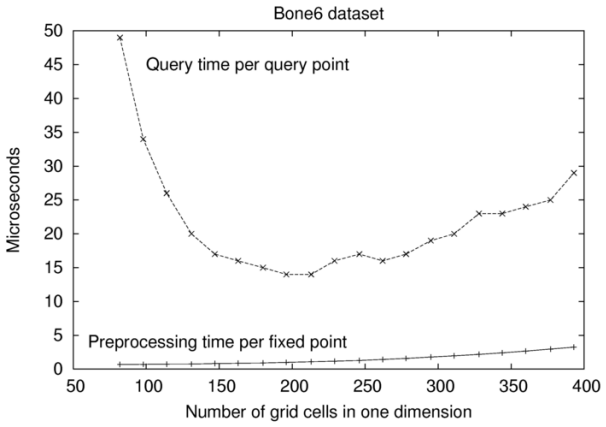

Figure 1: Varying Grid Size on the `Bone6` Dataset on the Xeon

N_f	Program	Time	T_q	Mem
$.1M$	NEARPT3	0.91	70	3.9M
	ANN	1.3	250	9.7M
$.3M$	NEARPT3	1.7	103	6.6M
	ANN	3.7	260	20.4M
$1M$	NEARPT3	3.7	118	16M
	ANN	15	270	94M
$3M$	NEARPT3	8.9	125	46M
	ANN	53	300	277M
$10M$	NEARPT3	28	136	140M
	ANN	—	—	—
$30M$	NEARPT3	82	159	328M
	ANN	—	—	—

Table 2: Comparison of ANN and NEARPT3

I/O since the input was randomly generated and the output not written. Table 2 shows the cost of performing 10,000 queries against N_f random fixed points, when run on the laptop. Time is the total CPU time in seconds. T_q is the query time per point in μseconds.

We killed the ANN run for $N_f = 10^7$ after 15 minutes of elapsed time, and didn't try $N_f = 3 \cdot 10^7$. While all such tests have the obvious limitations, and other datasets should also be tested, some points are clear. On this data, the uniform grid does not lose when compared to a hierarchical data structure. It is probably faster, and certainly smaller. Therefore it can process much larger datasets. As $N_f \to \infty$, it appears to get even better.

4 Discussion, Extensions, & Summary

1. The absolute times can vary 20% when the same tests are rerun. The relative times of two different tests can vary a factor of two depending on the platform and compiler options. That said:

2. If we ignore *bunny*, which ran too fast to measure accurately, the time to preprocess the fixed points is basically constant. The time did not depend on the size of the dataset, but on how evenly distributed the points were. Uniform data sets are

good. The powerplant was particularly bad for us because there are a few outlying points, which force the vast majority of the points into a small part of the grid. That demonstrates the limitation of this method.

3. Even on the uniform random data, the query time rose slowly with N_f. This is puzzling and needs study; our current guess is that accessing large amounts of memory is slightly less efficient.

4. NEARPT3 would fail on query points that were from a very different distribution from the fixed points, so that the distance to the nearest point was large, and most of the cells had to be searched. However, many competing methods would also have difficulties.

5. NEARPT3's cost is affected by the grid resolution, however values within a factor of two of the optimum typically change the time less than a factor of two.

6. Why do the David and St Matthew datasets have such large query times? Their points' local topology is two dimensional. Therefore the distribution of points around each query is very nonuniform. Most of the cells around each query are empty, while a few contain many points. We have collected some statistics on this. For such datasets, a more sophisticated, probably hierarchical, data structure would allow faster queries, if it didn't bloat up the execution size so much that the data set couldn't be processed at all.

7. NEARPT3 could return approximate nearest matches in much less time than exact nearest matches, since then the spiral search could stop sooner.

8. NEARPT3 could be extended to E^d for other d; the cost of searching would be exponential in d, as for any search procedure.

9. How might the fixed point storage budget be reduced? If the user's program doesn't need a separate copy of the points, then we can store the points' coordinates in the grid, instead of storing the points' indexes. Also, knowing which cell contains point p tells us the high order bits of p's coordinates. For a $512 \times 512 \times 512$ grid, that would save 27 of the 48 bits.

10. How might the storage budget of 4 bytes per cell be reduced? Using a hash table keyed on the cell location would reduce that to 0 bytes per empty cell plus perhaps 16 bytes per occupied call (for the cell location, pointer to its contents, and number of points in it, all times 2 for a conservative hash table load factor). That would be a win for our nonuniform examples. Also, that would allow larger grids to be run on our laptop. However, the execution time might be slower, tho that's not clear.

NEARPT3 is still immature, with several obvious possible optimizations in time and space. Nevertheless, the general lesson is that simple data structures like the uniform grid can be quite efficient in both time and space, especially in E^3.

References

[1] V. Akman, W. R. Franklin, M. Kankanhalli, and C. Narayanaswami. Geometric computing and the uniform grid data technique. *Computer Aided Design*, 21(7):410–420, 1989.

[2] S. Arya and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*, pages 271–280, 1993.

[3] W. R. Franklin. Nearpt3 — nearest point query on 184M points in E^3 with a uniform grid. `http://wrfranklin.org/Research/nearpt3/`, 2005.

[4] W. R. Franklin and M. Kankanhalli. Parallel object-space hidden surface removal. In *Proceedings of SIGGRAPH'90*, volume 24, pages 87–94, Aug. 1990.

[5] O. Kreylos. Nearest-neighbor-lookup. `http://graphics.cs.ucdavis.edu/~okreylos/ResDev/NearestNeighbors/`, 2005.

[6] M. Levoy. The digital Michelangelo project archive of 3D models. `http://www-graphics.stanford.edu/data/mich/`, 2003.

[7] M. Levoy. The Stanford 3D scanning repository. `http://www-graphics.stanford.edu/data/3Dscanrep/`, 2005.

[8] D. Mount and S. Arya. ANN: library for approximate nearest neighbor searching version 0.2 (beta release). `http://www.cs.umd.edu/~mount/ANN/`, 1998.

[9] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[10] S. S. Skiena. The algorithm design manual — nearest neighbor search. `http://www.cs.sunysb.edu/~algorith/files/nearest-neighbor.shtml`, 2001.

[11] G. Turk and B. Mullins. Large geometric models archive. `http://www.cc.gatech.edu/projects/large_models/`, 2003.

[12] UNC Chapel Hill Walkthru Project. Complete power plant model. `http://www.cs.unc.edu/~geom/Powerplant/`, 1997.