

# Nearest Point Query on 184,088,599 Points in $E^3$ with a Uniform Grid

W. Randolph Franklin, *Member, IEEE*

**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 for NEARPT3, in addition to the space to store the points themselves, is only 4 bytes per grid cell plus 4 bytes per point. Running on a laptop computer, NEARPT3 can process these large datasets: the UNC complete powerplant ( $N_f = 5,413,053$  fixed points), Lucy ( $N_f = 14,012,961$ ), David ( $N_f = 28,158,109$ ) and St Matthew ( $N_f = 184,088,599$ ). Nonuniform data can be as quick to process as uniform data, although the optimal grid size is larger. NEARPT3 demonstrates that simple data structures and algorithms can be quite competitive for processing large datasets in  $E^3$ .

**Index Terms**—computational geometry, nearest point query, uniform grid, large geometric datasets, closest point query, nearest neighbor

## I. 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, [17], 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  $\theta(\log N)$ . Range trees, [17], cost  $\theta(N \log N)$  time to preprocess, with each query also costing  $\theta(\log N)$ . ANN (Approximate Nearest Neighbors), [2], 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. Murphy and Skiena's Ranger uses k-d trees, [15], [18]. Krelon also has a recent fast implementation with kd-trees, [11]. Finally, if successive queries are close, perhaps one can efficiently traverse the dataset from one answer to the next.

All those algorithms and data structures are more general, hence bigger, than NEARPT3, which is optimized specifically for the  $L_2$  metric in  $E^3$ , although its ideas generalize. NEARPT3, which uses a *uniform grid*, [1], [8], [9], 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  $\theta(1)$ . Finally, extreme data unevenness forces even hierarchical data structures to have many levels, so that they become slower.

The broader purpose of this paper is to demonstrate that simple data structures and algorithms may be competitive, perhaps even better than, more sophisticated solutions. Uniform grids have additional applications, such as finding the intersections in a large number of small edges, [1], mass properties of boolean combinations of polygons and polyhedra (implemented with unions of rectangles and cubes), including point location in a planar graph, [4], [5], [10], [16], and hidden surface algorithms, [3], [7], [9]

ECSE Dept, 6026 JEC, Rensselaer Polytechnic Institute, 110 8th St, Troy NY 12180, mail@wrfranklin.org, http://wrfranklin.org/

## II. ALGORITHM

0000 0000/00\$00.00 ©NEARPT3 has three stages, as follows.

### A. Antepreprocess

This step generates part of the source code that will be included in `nearpt.cc` when it is compiled. That is a table of cells sorted by distance from the origin.

- 1) Generate the coordinates  $(x, y, z)$  of all grid cells with  $0 \leq x \leq y \leq z \leq R$  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.

### B. 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  $0.5 \leq r \leq 3$ .
- 2) Allocate a uniform grid with one word per cell, to store a count of the number of 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 20% over the available real memory, even if its working set size is still smaller than the available real memory.)

(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.) (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.)

### C. 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, which is the usual case when the fixed and query points are selected from the same distribution, then do the following.
  - a) Find  $f$ , the closest fixed point in  $c$  to  $q$ .
  - b) Find the distance from  $q$  to the closest wall of  $c$ .
  - c) If  $f$  is closer than the closest wall, then return  $f$ .
  - d) Otherwise search every cell around  $c$  whose closest point is at least as close as  $f$ . (We actually search a rectangular block of cells, which is easier but may check a few cells that don't need it.)
- 3) If  $c$  did not contain at least one fixed point, then do the following.
  - a) Using the sorted cell list computed in the anteprocessing step, 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.)

- b) For each cell with coordinates  $(x, y, z)$  in the sorted cell list, derive up to 47 other reflected and rotated cells, 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 antepreprocessing 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.)

- c) 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.)  
 (On the average only a few cells are searched for each query; this spiraling is rarely necessary.)

### III. TIME ANALYSIS

This is simple. Assuming that the total number of grid cells is linear in the number of fixed points, the 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  $\theta(1)$ .

### IV. TESTS

We implemented NEARPT3 in C++ under SuSE Linux. The environment was a 2005-vintage IBM T43p Thinkpad laptop computer with 2GB of memory, a 2GHz Pentium M processor, and Intel's icpc 9 C++ compiler, with aggressive optimizations enabled. g++ also works. Our code is available for nonprofit research and education, [6]; comments are welcome.

Our test data has many original sources, which we gratefully acknowledge below, and was often

distributed by the Georgia Institute of Technology's Large Geometric Models Archive, [19]. We report on every large dataset that we tested; there are no bad cases omitted.

- 1) uniXXX: Our uniformly and independently distributed sets of  $10^5$  to  $10^8$  random points.
- 2) bone6, the Visible Human Project, and William E. Lorensen, via Georgia Tech.
- 3) dragon: Brian Curless, via Stanford and Georgia Tech.
- 4) blade: Visualization Toolkit (VTK), via Georgia Tech.
- 5) hand: Clemson's Stereolithography Archive, via Georgia Tech.
- 6) powerplant: The complete powerplant from the University of North Carolina's UNC Chapel Hill Walkthru Project, [20].
- 7) bunny, lucy: Stanford University Computer Graphics Laboratory, [13].
- 8) david, stmatthew: The Stanford Digital Michelangelo Project Archive, [12]. We used the ply files, and are now working to use the raw scan data.

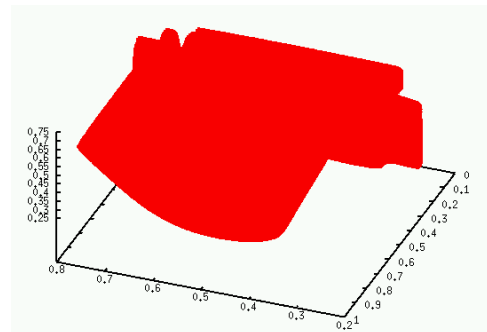


Fig. 1. Blade

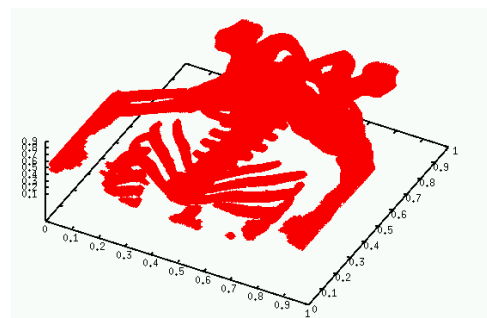


Fig. 2. Bone6

Fig. 1 to Fig. 9 show the datasets, and illustrate how nonuniform they are. In most cases, a random

TABLE I  
TEST RESULTS

<b>data</b>	$N_f$	$G_{fac}$	<b>G</b>	<b>total time (sec)</b>	<b>init time (sec)</b>	<b>pre- pro- cessing time (sec)</b>	<b>query time (sec)</b>	<b>pre- pro- cessing time per fixed pt (<math>\mu</math>sec)</b>	<b>query time per query pt (<math>\mu</math>sec)</b>
bunny	25,947	1.00	29	0.02	0.	0.01	0.01	0.38	1.
uni100k	100,000	0.50	23	0.06	0.	0.04	0.02	0.40	2.
uni300k	300,000	0.42	28	0.17	0.03	0.11	0.03	0.36	3.
hand	317,323	1.68	114	0.18	0.02	0.15	0.01	0.47	1.
dragon	427,645	1.19	89	0.21	0.03	0.17	0.01	0.39	1.
bone6	559,636	1.10	90	0.28	0.04	0.23	0.01	0.41	1.
blade	872,954	1.41	135	0.45	0.08	0.36	0.01	0.41	1.
uni1m	1,000,000	0.42	42	0.61	0.09	0.48	0.04	0.48	4.
uni3m	3,000,000	0.50	72	2.04	0.28	1.71	0.05	0.57	5.
powerplant	5,413,053	4.50	790	12.8	0.51	12.1	0.15	2.24	15.
uni10m	10,000,000	0.50	107	7.72	0.94	6.74	0.04	0.67	4.
lucy	14,012,961	3.36	810	23.3	1.33	21.9	0.03	1.56	3.
david	28,158,109	2.38	723	20.5	2.67	17.7	0.06	0.63	6.
uni30m	30,000,000	0.50	155	24.6	2.84	21.7	0.05	0.72	5.
uni100m	100,000,000	0.50	232	86.5	9.49	76.9	0.06	0.77	6.
stmatthew	184,088,599	0.71	398	85.8	18.1	67.1	0.53	0.36	53.

TABLE II  
FIXED POINT DISTRIBUTION FOR THE UNI100M DATA

$K$	$C_K$	$K$	$C_K$	$K$	$C_K$	$K$	$C_K$
0	183890	7	1623145	14	244371	21	1097
1	79760	8	1670826	15	134200	22	444
2	181618	9	1531200	16	69635	23	153
3	363227	10	1264189	17	33811	24	66
4	654355	11	947504	18	15352	25	23
5	1028138	12	651571	19	6740	26	6
6	1384716	13	414370	20	2757	27	3

subset of the points was plotted. No surface facets or other topology are shown as it is irrelevant. The

process for the test runs went as follows.

- 1) (Usually) sort the data and delete duplicate

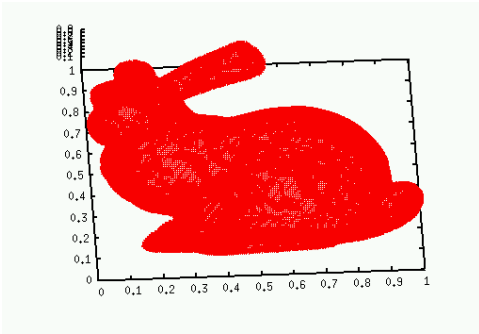


Fig. 3. Bunny



Fig. 4. David

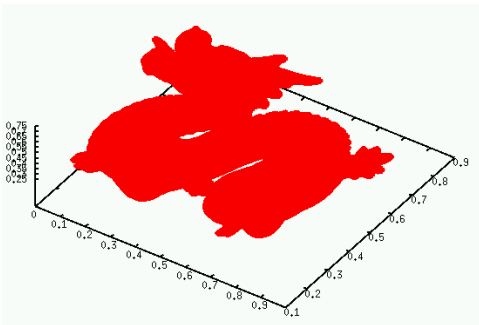


Fig. 5. Dragon

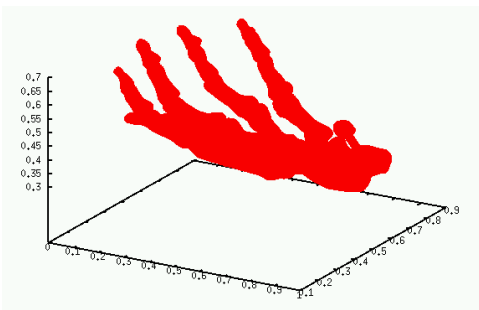


Fig. 6. Hand

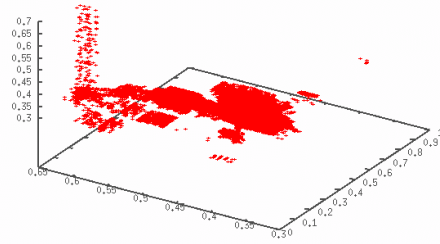


Fig. 7. Powerplant

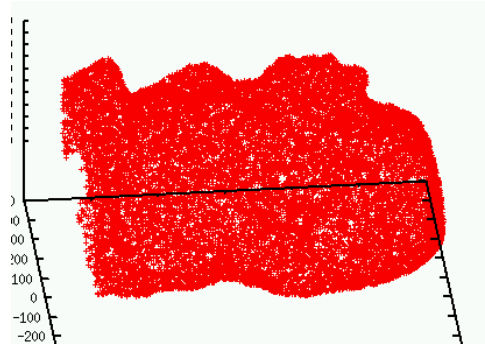


Fig. 8. St. Matthew

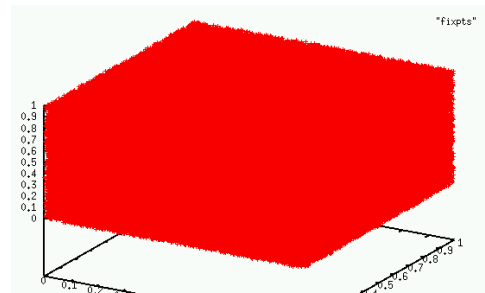


Fig. 9. Uniform i.i.d

points. (Some PLY files store a separate copy of a point for each incident triangle. Deleting duplicates made the program slower since it meant that the closest fixed point could never be the same as the query point.)

- 2) Uniformly scale the data into a  $[0, 1]^3$  cube. (If the dataset's extent in one dimension is greater than in another, then many grid cells will be empty. Here is a future possible storage optimization.)
- 3) Select 10,000 points with indices equally spaced throughout the data as query points, and use the rest as fixed points.
- 4) Store the data as 2-byte binary unsigned short ints, which is sufficient resolution for a laser scanner. (If the points had been stored in

ASCII, then reading them would have taken much more time than the nearest point determination.)

Table I shows the data we tested, with the best grid size, and resulting times.  $N_f$  is the number of fixed points.  $G$  is the number of grid cells on each side of the uniform grid.  $G_{fac} = \frac{G}{\sqrt[3]{N_f}}$  is an intuitive way to specify  $G$ . The query time is always an integral number of microseconds because the clock ticked every 0.01 second, and we did 10,000 queries for each dataset.

We did not try values of  $G$  that would put the program's memory over about 2.1GB, since above that, NEARPT3 started badly thrashing the memory. This affected the datasets *lucy*, *powerplant*, *david*, and *stmatthew*. For them, the query time was still decreasing when we stopped increasing  $G$ .

The big question with a uniform data structure is its performance on nonuniform data. We histogrammed the following statistics:

- 1)  $C_K$ , the number of grid cells containing  $K$  points, for various  $K$ , and
- 2)  $Q_K$ , the number of queries that had to search  $K$  cells to find the closest point, for various  $K$ .

Since the numbers cover such a wide range, tables present them better than graphs. Table II shows  $C_K$  for the *uni100m* dataset with  $N_f = 100,000,000$  and  $G = 232$ . Table III shows  $Q_K$ . The average number of fixed points per cell was 8.0; the maximum was 27. The average number of cells checked to resolve one query was 4.9; the maximum was 401.

Table IV and Table VI present  $C_K$  and  $Q_K$  for *david*, with  $N_f = 28,158,109$  and  $G = 724$ . 99.9% of the cells were empty, the average number of fixed points per cell was 0.074 and the maximum 231. This shows the extreme unevenness of the data. In 65% of the queries, only the cell containing the query needed to be checked. The average number of cells that one query searched to find the closest point was 1.75 and the max was 401. That is, even this quite even data was easy to process, apart from the large grid needed.

## V. VARYING THE GRID SIZE

What is the appropriate grid size to use? Determining the optimum is an area for research, however here are some observations.

TABLE III  
QUERY PERFORMANCE FOR THE *uni100m* DATA

$K$	$Q_K$	$K$	$Q_K$	$K$	$Q_K$
1	1477	18	67	233	1
2	2451	24	1	237	1
4	2577	27	8	241	3
8	3087	36	2	301	2
12	319	48	2	401	2

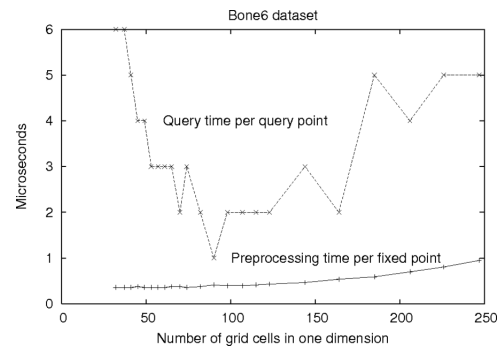


Fig. 10. Varying Grid Size on the *Bone6* Dataset

- 1) The answer is not that critical since the query time changes only slowly as a function of  $G$ . Fig. 10 shows the *bone6* dataset run with many grid sizes.
- 2) The user may tradeoff speed and memory by using a smaller  $G_{fac}$  than optimal. Indeed, the largest datasets demonstrate this. Our test machine had insufficient memory to use the optimal  $G_{fac}$ , but NEARPT3 still ran.

## VI. COMPARISON TO ANN

One classic nearest point algorithm in  $E^3$  is ANN 0.2 (Approximate Nearest Neighbors) [14]. ANN was compiled with the defaults, and run with the standard search method, which is faster than the priority method for this data. Running it required no data I/O since the input was randomly generated and the output not written. Table V 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  $\mu$ seconds. ANN required too much memory to process  $N_f \geq 3 \cdot 10^7$ .

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 much faster, and certainly

TABLE IV  
FIXED POINT DISTRIBUTION FOR THE DAVID DATA

$K$	$C_K$	$K$	$C_K$	$K$	$C_K$	$K$	$C_K$	$K$	$C_K$	$K$	$C_K$
0	379030902	34	2827	68	5375	102	3092	136	139	171	2
1	9903	35	2966	69	5106	103	2928	137	113	172	1
2	6682	36	3021	70	5158	104	2933	138	94	173	4
3	6817	37	2807	71	5432	105	2856	139	50	174	2
4	5446	38	2656	72	6108	106	2798	140	44	176	2
5	5303	39	2689	73	6373	107	2605	141	29	178	2
6	4653	40	3029	74	6681	108	2407	142	27	179	2
7	5096	41	2822	75	6365	109	2301	143	17	180	1
8	4484	42	2908	76	6107	110	2155	144	11	181	1
9	4093	43	2796	77	5902	111	2061	145	11	182	1
10	3791	44	2716	78	5611	112	1960	146	25	183	2
11	3612	45	2868	79	5587	113	1897	147	16	184	1
12	3515	46	2671	80	5912	114	1782	148	20	185	1
13	3510	47	2735	81	5998	115	1706	149	10	186	4
14	3875	48	2906	82	5948	116	1634	150	16	187	1
15	3471	49	3999	83	5742	117	1508	151	12	188	1
16	3729	50	2977	84	5484	118	1346	152	15	189	1
17	3339	51	2911	85	5203	119	1321	153	8	191	1
18	3181	52	2912	86	5083	120	1261	154	14	194	1
19	2996	53	2895	87	4859	121	1105	155	8	195	1
20	2985	54	2842	88	5045	122	1129	156	6	196	1
21	3395	55	2904	89	5069	123	967	157	6	197	1
22	3124	56	5985	90	4785	124	801	158	4	198	1
23	2952	57	3819	91	4725	125	777	159	10	205	2
24	3281	58	3910	92	4559	126	716	160	5	211	1
25	3067	59	3877	93	4225	127	686	161	6	212	1
26	2961	60	3830	94	4142	128	605	162	3	215	1
27	2875	61	3701	95	3880	129	587	163	5	217	1
28	2970	62	3684	96	3935	130	438	164	2	221	1
29	3020	63	4262	97	3794	131	369	165	2	224	1
30	2936	64	6405	98	3747	132	293	166	8	230	1
31	2799	65	5442	99	3593	133	236	168	4	231	1
32	3021	66	5720	100	3404	134	202	169	4		
33	2951	67	5623	101	3197	135	185	170	3		

TABLE V  
COMPARISON OF ANN AND NEARPT3

$N_f$	NEARPT3 $T_q$	ANN $T_q$	NEARPT3 Mem	ANN Mem
100K	$2\mu s$	$7\mu s$	6MB	10MB
300K	$3\mu s$	$7\mu s$	8MB	28MB
1M	$4\mu s$	$9\mu s$	15MB	92MB
3M	$5\mu s$	$10\mu s$	36MB	311MB
10M	$4\mu s$	$10\mu s$	105MB	916MB
30M	$5\mu s$	—	305MB	—
100M	$6\mu s$	—	1006MB	—

much smaller. Therefore it can process much larger datasets. As  $N_f \rightarrow \infty$ , its relative advantage increases.

TABLE VI  
QUERY PERFORMANCE FOR THE DAVID DATA

$K$	$Q_K$
1	6483
2	2711
4	704
8	97
401	5

## VII. IMPROVEMENTS IN THIS VERSION

### VIII. DISCUSSION

- 1) This version has various algorithmic improvements compared to the first public version, which always searched at least  $5 \times 5 \times 5$  block of cells around the query point. E.g., the query time for `david`, albeit on a slightly slower machine, was 655 microseconds, but now it is only 6.
- 2) 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:
- 3) The time to preprocess the fixed points grows very slowly with  $N_f$ . (This is puzzling and needs study; our current guess is that accessing large arrays is less efficient, perhaps because of cache misses.)
- 4) The query time on uniform data also rises very slowly with  $N_f$ .
- 5) For nonuniform data, the optimal  $G$  is larger than for uniform data of the same size. Presumably, an optimal grid limits the maximum number of fixed points per cell.
- 6) For very nonuniform data, there may be insufficient memory for the optimal  $G$ . In that case, NEARPT3 will still run, but more slowly.
- 7) Apart from that, NEARPT3 can process nonuniform data as quickly as it can process uniform data. E.g., compare `david` and `uni30m` in Table I. They have about the same  $N_f$ . Because `david` is so nonuniform, its optimal grid has 100 times as many cells. However, its execution time is a little less.
- 8) Even the datasets produced by laser scanning an actual object, such as `david`, whose points' local topologies are two dimensional, are easy. This is in spite of most of the cells around each query being empty, while a few cells contain many points.
- 9) `powerplant` illustrates another way for a dataset to be particularly bad. There are a few outlying points, which force the vast majority of the points into a small part of the grid. The resulting optimal  $G$  would be too large to execute. In this case, hashing the nonempty grid cells would be a reasonable strategy.
- 10) 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.



- 11) 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 2.

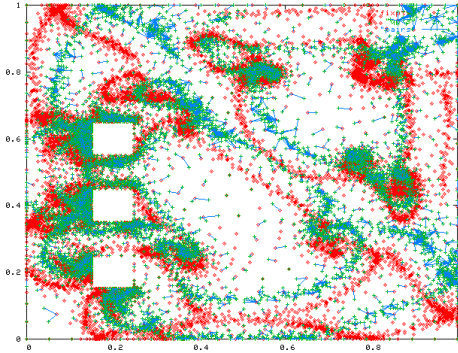


Fig. 11. Nearest Point Determination in  $E^2$

## IX. EXTENSIONS

- 1) NEARPT3 could return approximate nearest matches in much less time than exact nearest matches, since fewer cells would need to be searched.
- 2) Some simple optimizations are possible, such as not allocating large blocks of empty cells when the data's range in one dimension is smaller than its range in another dimension.
- 3) Profiling the code to identify execution hotspots could help. However, that is tricky because such hotspots probably occur in optimized inlined code originating from small C++ class routines that are called many times. That is, measuring complete routines' execution times is rather useless.
- 4) NEARPT3 could be extended to  $E^d$  for other  $d$ ; the cost of searching would be exponential in  $d$ , as for any search procedure.
- 5) A version of NEARPT3 for  $E^2$  runs much faster. Figure 11 shows one test case, a nonuniform mesh from Mark Shepherd with  $N_f = 10267$  and  $N_q = 8005$ . Some of the triangles in the mesh have aspect ratios of 100:1. The fixed points are red, the queries blue, and a green line connects each query to its answer. Excluding I/O, the total preprocessing and query time is  $0.01 \pm 0.01$  seconds, i.e., too small to measure.
- 6) 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.

- 7) 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 cell (for the cell location, pointer to its contents, and number of points in it, all multiplied by 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. The execution time might be slower, although that's not clear.

## X. SUMMARY

The broader lesson of NEARPT3 is that simple data structures like the uniform grid can be quite efficient in both time and space, especially in  $E^3$ , even for very nonuniform data.

## XI. ACKNOWLEDGEMENTS

This research was supported by NSF grant CCR-0306502. 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.

## REFERENCES

- [1] Akman, V., W. R. Franklin, M. Kankanalli, and C. Narayanaswami. Geometric computing and the uniform grid data technique. *Computer Aided Design* **21**(7), (1989), 410–420.
- [2] Arya, S. and D. M. Mount. Approximate nearest neighbor queries in fixed dimensions. In *Proc. 4th ACM-SIAM Sympos. Discrete Algorithms*. 271–280.
- [3] Franklin, W. R. An exact hidden sphere algorithm that operates in linear time. *Comput. Graph. Image Process.* **15**, (1981), 364–379.
- [4] Franklin, W. R. Analysis of mass properties of the union of millions of polyhedra. In Lucian, M. L. and M. Neamtu (eds.) *Geometric Modeling and Computing: Seattle 2003*. Nashboro Press, Brentwood TN. ISBN 0-0-9728482-3-1, 189–202.

- [5] Franklin, W. R. Mass properties of the union of millions of identical cubes. In Janardan, R., D. Dutta, and M. Smid (eds.) *Geometric and Algorithmic Aspects of Computer Aided Design and Manufacturing, DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 67. American Mathematical Society, 329–345.
- [6] Franklin, W. R. Nearpt3 — Nearest point query on 184M points in  $E^3$  with a uniform grid. <http://wrfranklin.org/Research/nearpt3/>, 2005.
- [7] Franklin, W. R. and V. Akman. Adaptive grid for polyhedral visibility in object space, an implementation. *Computer Journal* **31**(1), (1988), 56–60.
- [8] Franklin, W. R., N. Chandrasekhar, M. Kankanhalli, V. Akman, and P. Y. Wu. Efficient geometric operations for CAD. In Wozny, M. J., J. U. Turner, and K. Preiss (eds.) *Geometric Modeling for Product Engineering*. Elsevier Science Publishers B.V. (North-Holland), 485–498.
- [9] Franklin, W. R. and M. Kankanhalli. Parallel object-space hidden surface removal. In *Proceedings of SIGGRAPH'90*, volume 24. 87–94.
- [10] Kankanhalli, M. and W. R. Franklin. Area and perimeter computation of the union of a set of iso-rectangles in parallel. *J. Parallel Distrib. Comput.* **27**, (1995), 107–117.
- [11] Kreylos, O. Nearest-neighbor-lookup. <http://graphics.cs.ucdavis.edu/~okreylos/ResDev/NearestNeighbors/>, 2005.
- [12] Levoy, M. The digital Michelangelo project archive of 3D models. <http://www-graphics.stanford.edu/data/mich/>, 2003.
- [13] Levoy, M. The Stanford 3D scanning repository. <http://www-graphics.stanford.edu/data/3Dscanrep/>, 2005.
- [14] Mount, D. and S. Arya. ANN: library for approximate nearest neighbor searching version 0.2 (beta release). <http://www.cs.umd.edu/~mount/ANN/>, 1998.
- [15] Murphy, M. and S. S. Skiena. Ranger — nearest neighbor search in higher dimensions. <http://www.cs.sunysb.edu/~algorithm/implementation/ranger/implementation.shtml>, 1996.
- [16] Narayanaswami, C. and W. R. Franklin. Determination of mass properties of polygonal CSG objects in parallel. *Internat. J. Comput. Geom. Appl.* **1**(4), (1991), 381–403.
- [17] Preparata, F. P. and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.
- [18] Skiena, S. S. The algorithm design manual — nearest neighbor search. <http://www.cs.sunysb.edu/~algorithm/files/nearest-neighbor.shtml>, 2001.
- [19] Turk, G. and B. Mullins. Large geometric models archive. [http://www.cc.gatech.edu/projects/large\\_models/](http://www.cc.gatech.edu/projects/large_models/), 2003.
- [20] UNC Chapel Hill Walkthru Project. Complete power plant model. <http://www.cs.unc.edu/~geom/Powerplant/>, 1997.