NEARPT3— Nearest Point Query in E3 with a Uniform Grid

W. Randolph Franklin

Rensselaer Polytechnic Institute

geom@wrfranklin.org

http://wrfranklin.org/

—What?—

1. Preprocess N fixed points, p_i, in E^3.

2. Repeat:

   (a) Read a query point, q_j.

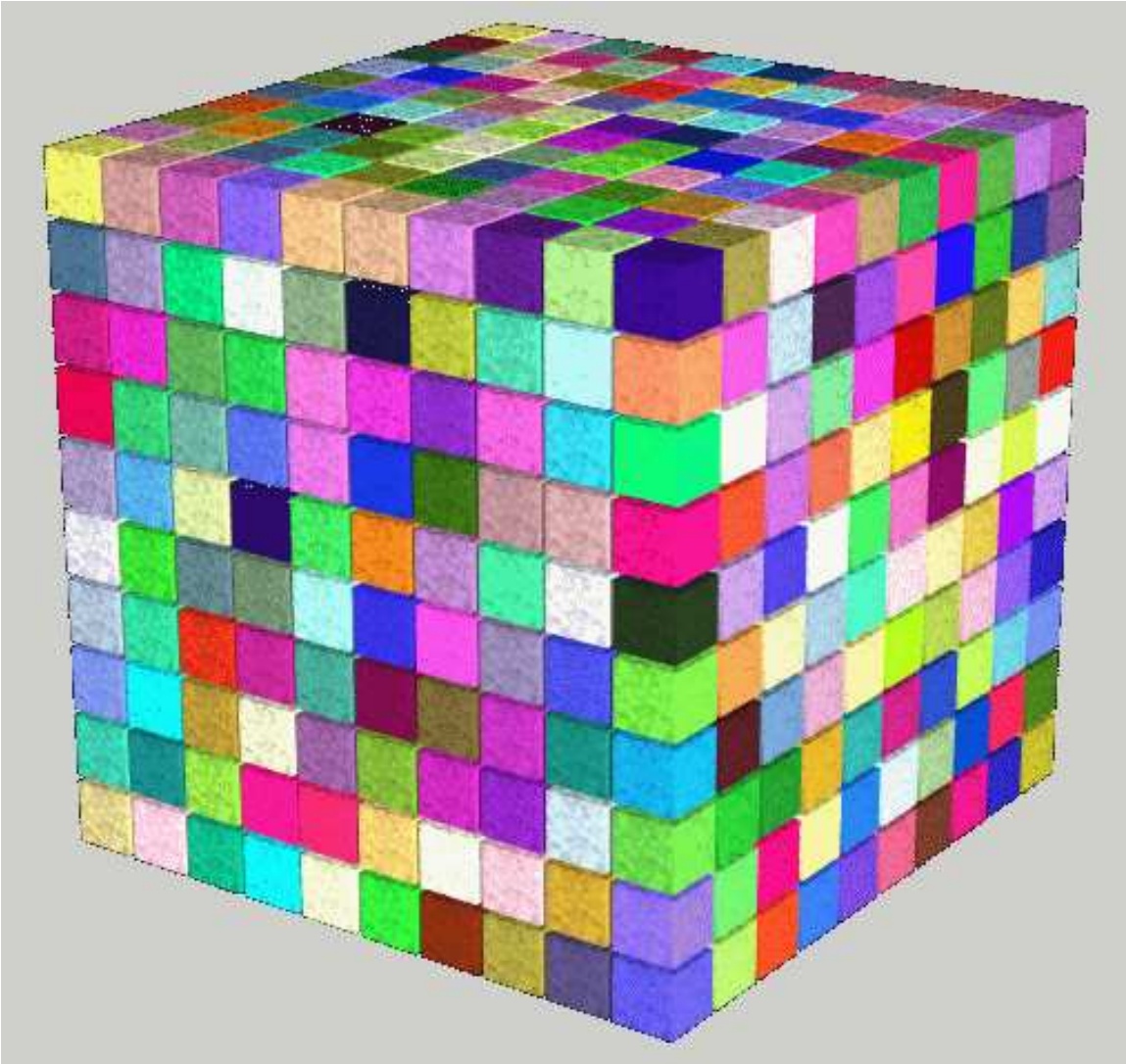
   (b) Find the closest p_{c_j}.

   ⋆ Optimize for $N \approx 10^7$.

   ⋆ Optimize for real examples from, e.g., Georgia Tech Large Geometric Models Archive, Stanford Michaelangelo David.

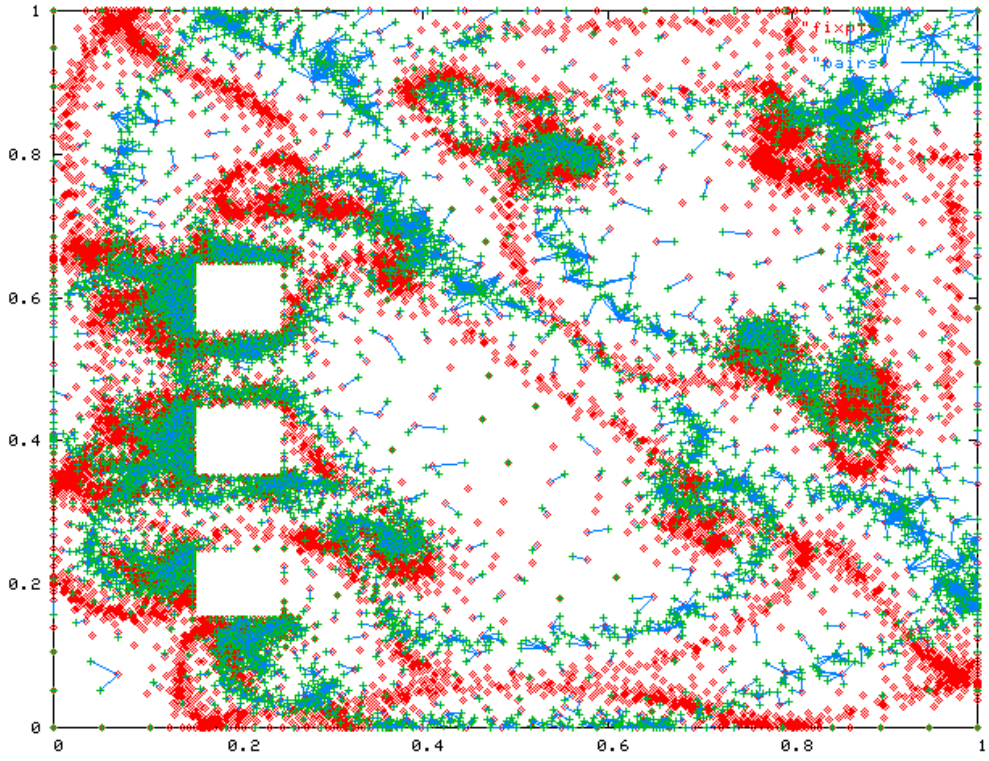
—How?—

- ⋆ **Insert points into a 3-D uniform grid of cells (buckets).**
- ⋆ **Spiral search out from the query cell.**

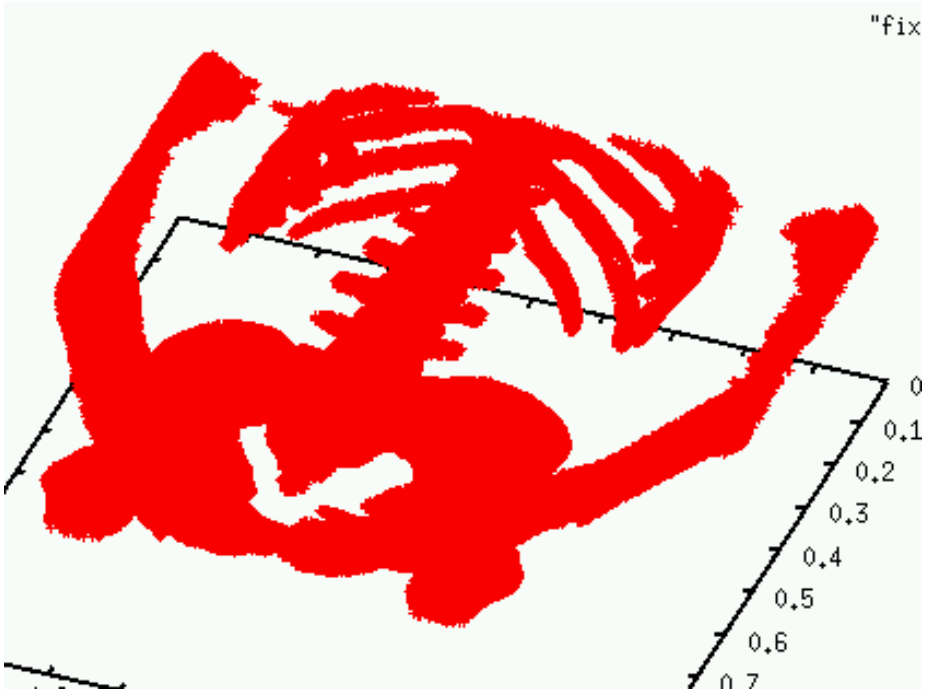


—E^2 Example—

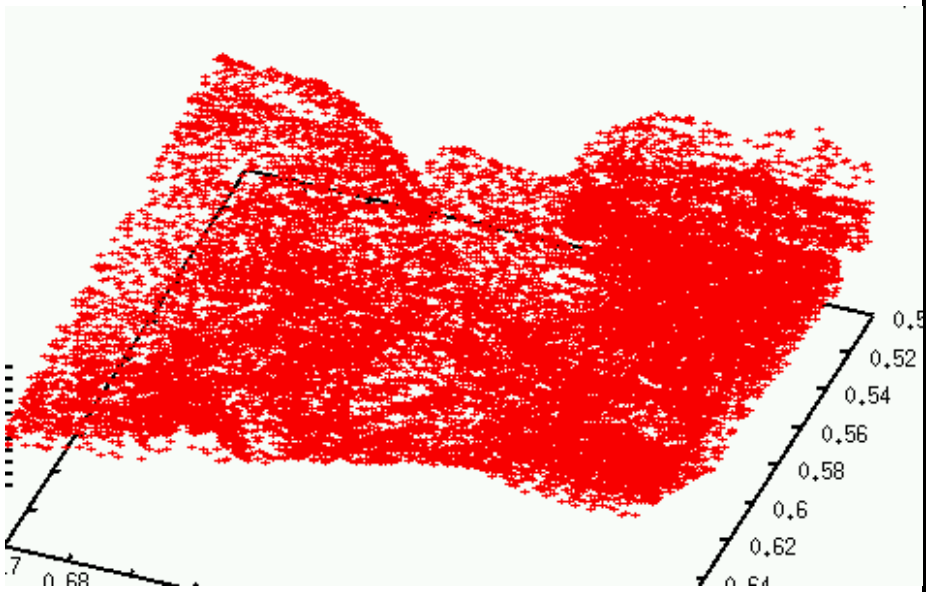
- ★ Points from non-uniform finite element mesh
- ★ red fixed points
- ★ green query points
- ★ blue line from each query to its closest fixed point



—E^3 Bone6 Example—



Whole dataset; note the nonuniformity.



Detail.

- ⋆ $10^{5.75}$ fixed points and 10^4 queries.
- ⋆ Total preprocessing and query time: 0.53 sec. on this 2002-vintage laptop.

—Prior Art— Data structures:

- ⋆ Voronoi diagram. $T_p = \Omega(N \log N)$ to $O(N^2)$ in time and space. $T_q = \theta(\log N)$.
- ⋆ Range tree. $T_p = \theta(N \log N)$. $T_q = \theta(\log N)$.

Implementations:

- ⋆ Approximate Nearest Neighbors (ANN).
- ⋆ CGAL.
- ⋆ Possible unpublished biologists' implementations.
- ⋆ Assume successive queries are close, and walk a planar graph from the last solution.

Comparison:

- ⋆ These may be more robust against uneven data.
- ⋆ However, very uneven data is also bad for hierarchies.
- ⋆ Bigger data structures; smaller max problem size.

—The Three Stages of the Computation—

A. Antepreprocessing w/o data. Compute-bound work that is independent of the data.

B. Preprocessing of the fixed points.

C. Querying to find closest fixed point to each query.

Getting the details right is what makes it work fast.

—A. Antepreprocessing (w/o data)—

1. Sort the cells of a grid in E^3 by distance of the closest corner from $\mathcal{O}$.
2. For each cell, c, find its *stop cell*, the last cell in the list whose closest point is closer than the *farthest* point of c.
3. Result: C++ source code listing sorted cells with stop cells, included in `nearpt3.cc` when compiled.

(0,0,0),13	(0,1,2),33	(0,1,3),49	(0,0,4),61	(1,3,3),75
(0,0,1),18	(1,1,2),39	(1,1,3),51	(0,1,4),61	(0,2,4),75
(0,1,1),24	(0,2,2),42	(2,2,2),56	(2,2,3),72	(1,2,4),77
(1,1,1),31	(1,2,2),49	(0,2,3),56	(0,3,3),72	(2,3,3),86
(0,0,2),31	(0,0,3),49	(1,2,3),61	(1,1,4),72	(2,2,4),89

Notes:

1. Actually, sort only the cells in 1/48 of the sphere.
2. Why antepreprocess? It's surprisingly slow.

—B. Preprocessing (of the fixed points)—

1. Choose $\mathcal{G}$, the grid resolution, say $1.6\sqrt[3]{N_f}$.
2. Alloc a uniform grid with 1 word per cell, to store the number of points in each cell.
3. Read the points, determine which cell contains each point, and update the counts.
4. Morph the counts array into a dope vector.
5. Allocate a ragged array to store the points in the cells.
6. Read the points again, inserting into the cells.

—Paths Not Taken—

- ⋆ **Linked lists:** Pointers take too much space. Points in each cell are nonlocal.
- ⋆ **C++ STL `vector`**, which grows: Realllocations fragment memory. Max size is a puny 2GB.
- ⋆ **Hash table** of the nonempty cells: This would be better if most cells are empty.

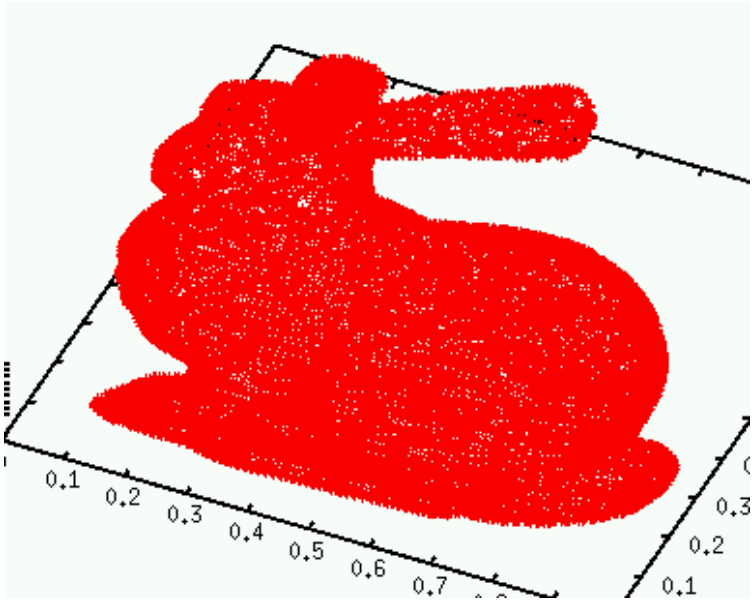
—Querying—

1. Locate the cell, c, containing the query point.
2. If c contains a fixed point, search a $5 \times 5 \times 5$ block of cells for the closest point.
3. Else, spiral out, following the antepreprocessed cell list.
4. For each (x, y, z) in list, also try $(y, -z, x)$ etc.
5. If the list is exhausted, test every fixed point.

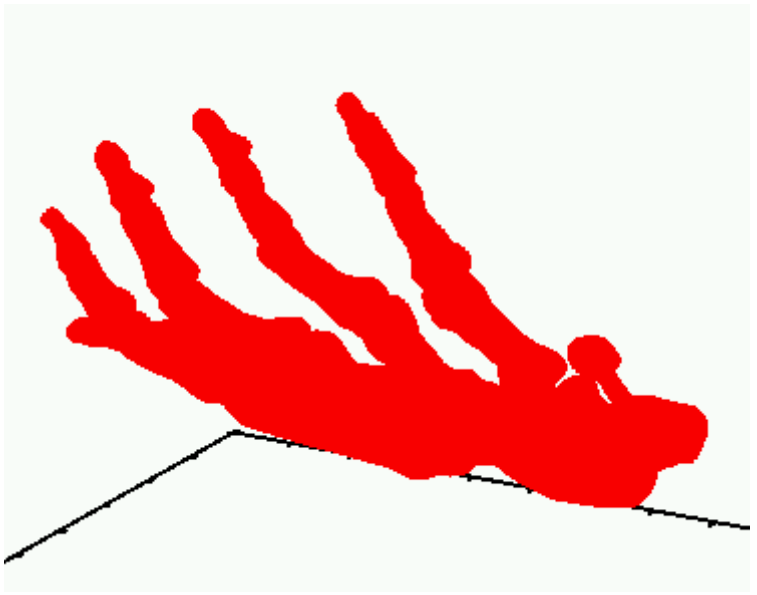
—Notes on Querying—

1. **This is optimized for query points having the same distribution as the fixed points.**
2. **The sorted cell list ignores where in its cell is the query, which seems suboptimal.**
3. **Searching the whole $5 \times 5 \times 5$ block is unnecessary.**
4. **However, testing fixed points is fast.**

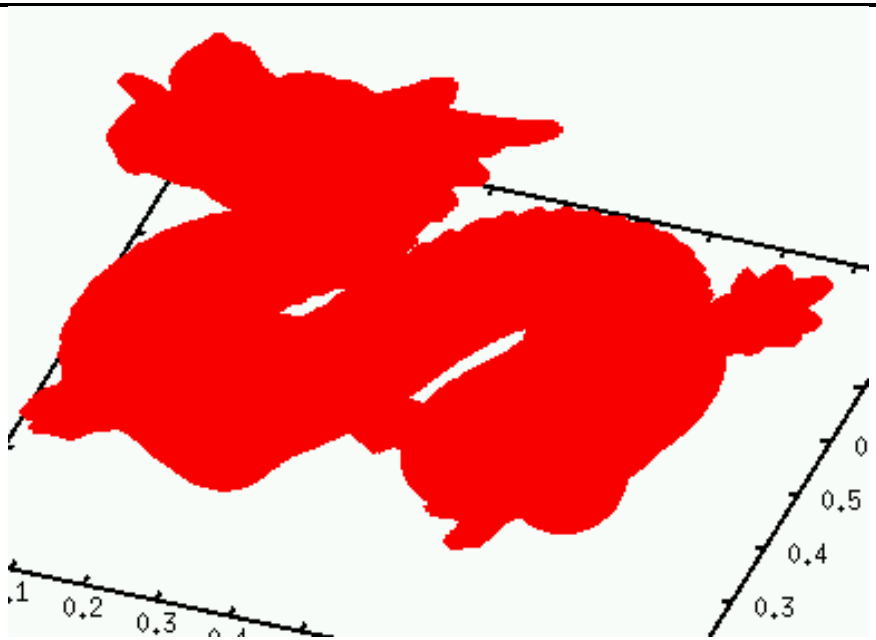
—Next Slides: Test Data—



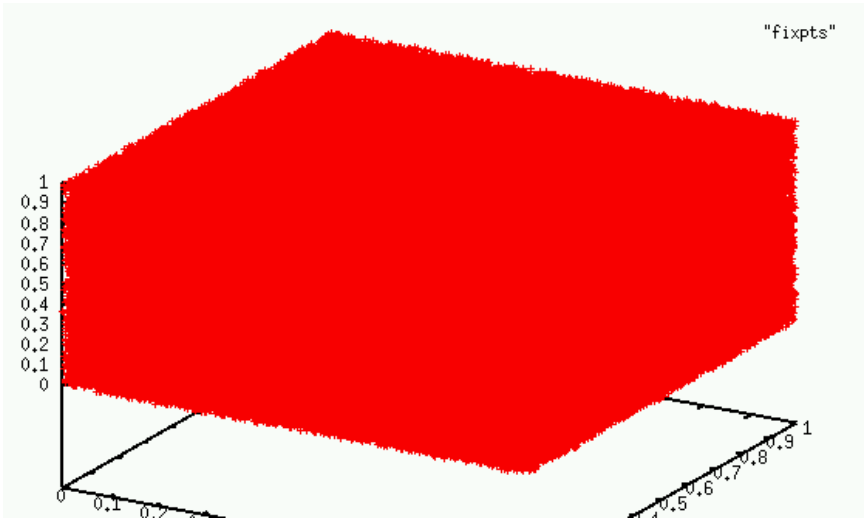
Bunny



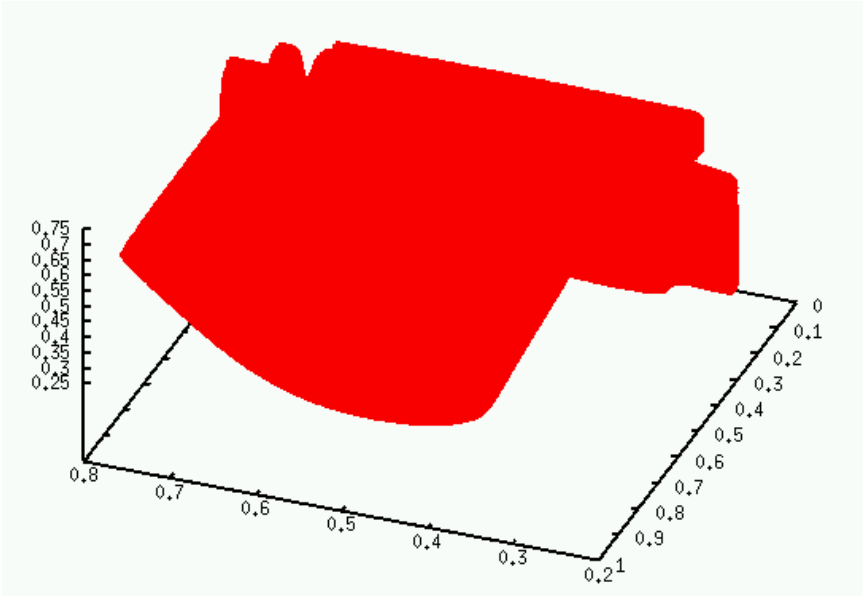
Hand



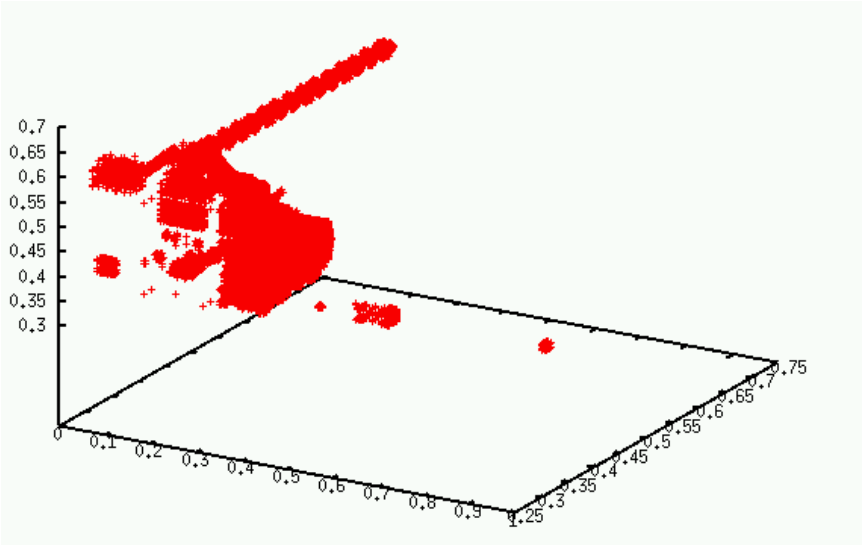
Dragon



Uniform

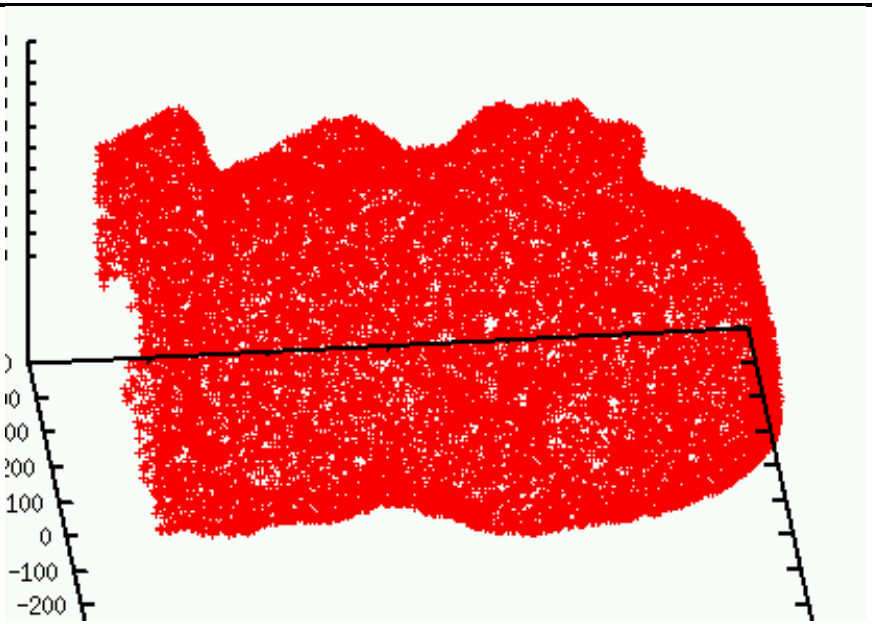


Blade



Complete Powerplant

Michaelangelo's David



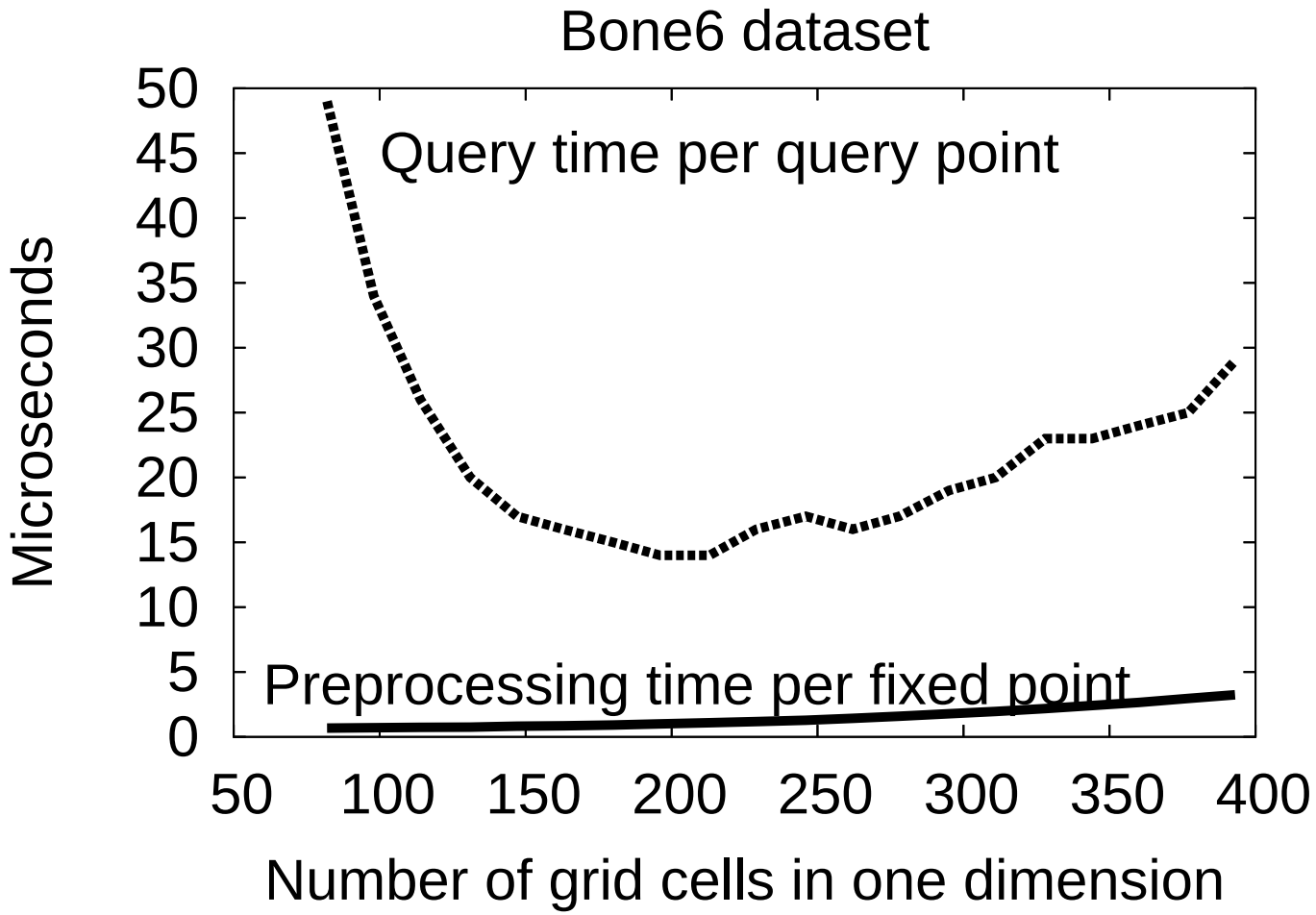
Michaelangelo's St. Matthew

—Results (on Xeon)—

Data	N_f	G	Total Time (seconds)	Init Time (seconds)	Prepr Time (μsec/pt)	Query Time (μsec/pt)
Bunny	25947	46	0.1	0.	0.39	7.
Hand	317323	108	0.5	0.05	0.47	30.
Dragon	427645	120	0.5	0.06	0.51	25.
Bone6	559636	131	0.5	0.07	0.45	21.
Blade	872954	152	0.7	0.11	0.46	17.
Uni1m	1000000	160	1.5	0.12	0.94	40.
Powerplant	5413053	280	63.2	0.98	0.52	5940.
Uni10m	10000000	344	12.8	1.27	1.11	43.
David	28158109	486	20.4	3.40	0.47	391.
Uni30m	30000000	496	41.7	3.88	1.25	46.
Uni100m	100000000	742	151	13.01	1.37	47.
Stmatthew	184088599	568	160	23.75	0.70	762.

—Optimizing G—

1. **This is a very broad optimum.**
2. **Graph shows preprocessing and query times, per point, on bone6, on Xeon.**



—Comparison to ANN—

N_f fixed points, 10,000 queries, run on laptop.

N_f	Program	Time	Mem
$.1M$	NEARPT3	0.91	3.9M
	ANN	1.3	9.7M
$.3M$	NEARPT3	1.7	6.6M
	ANN	3.7	20.4M
$1M$	NEARPT3	3.7	16M
	ANN	15	94M
$3M$	NEARPT3	8.9	46M
	ANN	53	277M
$10M$	NEARPT3	28	140M
	ANN	—	—
$30M$	NEARPT3	82	328M
	ANN	—	—

—Extensions and Limitations—

- ⋆ Approximate nearest point query is even faster.
- ⋆ E^2 works even better than E^3.
- ⋆ Query distribution different from fixed distribution is bad.
- ⋆ Points locally nonuniform, e.g., locally E^2 or E^1 (embedded in E^3) is bad.
- ⋆ Several possible optimizations may reduce time and space; this is still immature.
- ⋆ Uniform grid extends to point location, polyhedron intersections, etc, etc.

—Summary incl. Broader Implication—

- ⋆ Simple data structures, e.g., uniform grid, often work, especially in E^3.
- ⋆ Data dependent performance: Worst case is bad, but real data sets are quite good.
- ⋆ Larger datasets will now fit in real memory
- ⋆ ... and be processed much more quickly.
- ⋆ External data structures are less necessary.