NearptD: A Parallel Implementation of Exact Nearest Neighbor Search using a Uniform Grid

David Hedin* W. Randolph Franklin†

Abstract

We present **NearptD**, a very fast parallel nearest neighbor algorithm and implementation, which has processed 10^7 points in E^6 and $184 \cdot 10^6$ points in E^3. It uses 1/5 the space and as little as 1/100 the preprocessing time FLANN (a well-known approximate nearest neighbor program). Up to E^4, its query time is also faster, by up to a factor of 100. NearptD uses Nvidia Thrust and CUDA in C++ to perform parallel preprocessing and querying of large point cloud data. Nearest neighbor searching is needed by many applications, such as collision detection, computer vision or machine learning. This implementation is an extension of the Nearpt3 algorithm performed in parallel on the GPU for a variable number of dimensions. NearptD shows that a uniform grid can outperform a kd-tree for preprocessing and searching large datasets.

1 Introduction

Nearest neighbor searching is an operation performed in many applications, in fields such as computer graphics, computer vision, statistics and machine learning. Nearest neighbor searching typically consists of preprocessing the data into a search structure to reduce the time needed for future queries on that dataset. A popular data structure for preprocessing spatial data is the kd-tree[4]. Kd-trees cost $\Theta(NlogN)$ time to preprocess N fixed points and an average $\Theta(logN)$ time per query.

NearptD uses a uniform grid[1], which stores the fixed points in a flat search structure. This reduces the time and space complexity of kd-trees. Because the uniform grid used by NearptD is not a hierarchical data structure, the data can be preprocessed, with the appropriate choice of grid size, into a grid with a cost of $\Theta(N)$. Because querying against the uniform grid does not involve traversing a tree structure, NearptD can obtain expected query times of $\Theta(1)$. It is possible for adversarial input to increase query times to $\Theta(N)$, but these inputs are not often found in real world datasets. Some of our test datasets have very unevenly spaced data, but NearptD still processed them quickly. These types of input would still induce many levels in a hierarchical data structure, which would slow them as well.

Exact nearest neighbor searching can be an expensive operation. It often requires continuing to search after one near neighbor has been found, to ensure that it is, in fact, the nearest neighbor. One method to increase the performance of many data structures is to perform approximate nearest neighbor search instead of exact[6]. This reduces the time required to ensure the validity of the output, only ensuring that it is "good enough" for the application. However, NearptD shows that it can perform exact nearest neighbor searching and still outperform approximate searching.

2 Related Work

This work is mainly based on Nearpt3[2], a nearest neighbor search algorithm which uses a uniform grid in 3 dimensions. NearptD extends Nearpt3 to allow the flexibility of any number of dimensions to be used, though practicality limits this to 6 dimensions.

A widely used nearest neighbor search library is the Fast Library for Approximate Nearest Neighbors (FLANN)[3], a part of the OpenCV library. FLANN is an approximate nearest neighbor library that utilizes kd-trees and k-means trees[5] to preprocess data into a search tree. The Computational Geometry Algorithms Library (CGAL)[7] also offers both approximate and exact nearest neighbor searching using kd-trees.

3 Parallel Programming in Geometry

NearptD executes in parallel on Nvidia GPUs. Perhaps 1/3 of all PCs have them, intended to accelerate graphics. However, they can also be used for general parallel programming. The low-level access is via CUDA, a small set of extensions to C++ together with a library. Higher level APIs like Thrust add more powerful tools like a functional language paradigm, at the cost of less low-level control. GPUs provide so much computing power that geometric algorithms that are not parallelizable are quite possibly obsolete. The challenge is that parallelizable algorithms require simple regular data structures and algorithms.

For parallel programming, multicore CPUs present an attractive alternative to GPUs. All modern pow-

*Rensselaer Polytechnic Institute, hedind@rpi.edu
†Rensselaer Polytechnic Institute, mail@wrfranklin.org

erful CPUs are multicore, even those in smart phones. The two types of parallel hardware have different capabilities. Thrust can also be compiled to use multicore CPUs, so that many algorithms can use either.

4 Algorithm

4.1 Antepreprocess

As described later, the query step will spiral out from the cell containing the query point.

A table of cells, called the *spiral order table*, containing the order in which to spiral out, cell is computed before preprocessing the data. This table does not depend on the data. It also contains the *stop cell*, which says many more cells to query after the first cell containing a fixed point is found. This is to ensure the nearest neighbor is, in fact, found, because a later cell in the spiral order might contain a closer point than the first point found. The spiral order table's size in the GPU memory depends only on the dimensionality of the data, shown in Table 1. The table is computed as follows.

Dimensions	Memory
2	8MB
3	10MB
4	12MB
d	$(2d+4)$MB

Table 1: GPU memory usage of cells array.

1. Generate coordinates $(x_1, x_2, \ldots, x_d)$ for all grid cells such that $0 \leq x_1 \leq x_2 \leq \ldots \leq x_d \leq R$ for some R, calculated to ensure the total number of cells will be less than 2^{20}.
2. Sort coordinates by $\sqrt{{x_1}^2 + {x_2}^2 + \ldots + {x_d}^2}$
3. For each cell, c, find its *stop cell*, whose closest point to the origin is at least as close as the farthest point in c.

The spiral order table can be computed by the programmer while developing NearptD, and distributed in a file to be read at run time.

4.2 Preprocess

The uniform grid will contain G cells in each dimension, for a total of G^d cells, where d is the dimensionality of the data. G is called the grid's *resolution*. Before preprocessing the data, G must be determined. We use $G = G_r \sqrt[d]{N_f}$, where N_f is the number of fixed points, and G_r is a scaling factor, usually $0.5 \leq G_r \leq 3$, though the ideal factor is another possible area of research. As G_r varies, parts of NearptD run faster, and others more slowly, so that the total time varies relatively little even as G_r varies a factor of two away from its optimum. However, larger values of G_r do increase the memory footprint.

Next, three arrays are allocated, as follows.

1. *cells*, an array of size N_f, to contain pointers to the points in each cell,
2. *base*, an array of size $G^d + 1$ for the indices of the start of each cell within *cells*, so that the *j-th* point of the *i-th* cell is point # `cells[base[i]+j]`, and
3. *index*, a temporary array of size N_f allocated to preprocess the points, and discarded afterwards.

Preprocessing the data into a uniform grid is done on the GPU in parallel using these three arrays, as follows.

1. Store a sequence from 0 to N_f in *index*, to maintain the initial order of the array.
2. Calculate the ID of the cell that each fixed point belongs to, and store it in *cells*.
3. Sort *cells* and *index* based on the calculated IDs, to group points together by cell.
4. Calculate the index of the start of each cell, and store it in *base*.
5. Scan across each cell to calculate the number of points in each cell, stored in *cells*.
6. Resort *cells* by the original *index* array, to restore the original order of the points.
7. Transform the *index* array to contain the offset of each point within its cell, calculated from the *cells* and *base* arrays.
8. Fill *cells* with a sequence from 0 to N_f to keep track of the order of the array.
9. Sort *index* and *cells* by the offset of each point.
10. *cells* now contains the index of each point, sorted by their position in the grid.

If only the points' coordinates are relevant, and not, say, their location in some other data structure, then *cells* could contain the points' coordinates themselves instead of pointers. In machine-level programming, this is called *immediate mode*. The benefits are decreased memory use and increased locality of memory reference. On either CPUs or GPUs, that can reduce memory access times by reducing cache misses.

4.3 Query

There are three possible types of query that can be performed for a given query point, q, denoted as the *fast case*, the *slow case*, and the *exhaustive case*. A *fast case* query is performed if c, the cell containing q, contains at least one fixed point. If c is empty, a *slow case* query is performed, spiraling out from c, checking if any nearby cells contain fixed points for querying against. If the *slow case* query fails, *exhaustive* querying is performed to query against every fixed point.

This implementation supports querying many points at once, which is performed in parallel on the GPU, returning a list of the index of the closest fixed point for each query point, as well as single point queries. In more detail:

1. Calculate the number of fixed points in the cell containing each query point.
2. For all queries that contain at least one fixed point in their cell, perform a *fast case* query.
3. For all queries that don't contain a fixed point in their cell, perform a *slow case* query.
4. If the slow case query failed to find a fixed point, perform an *exhaustive* query.

4.3.1 Fast Case Query

This query is only performed if there are fixed points in c. Each point in the query cell is tested and the closest fixed point, f, is found. It is possible, however, that a neighboring cell could contain a point closer to q than f, if, for instance, q was near a wall of c. So, calculate the nearby cells that could contain a point closer than f, and search them for the closest fixed point.

4.3.2 Slow Case Query

If c does not contain any points, the next step is to begin searching around c for cells that may contain points. This is done by using the spiral order table computed in the antepreprocessing step to spiral out from c. For each cell in the table, we derive other reflected and rotated cells. For d dimensional data, there are up to $2^d d!$ possible reflections and permutations, although if some indices are zeros or repeated values, this number can be smaller. These could be pre-computed in the antepreprocessing stage, but this would require much more fixed memory. If a fixed point is found in one of these cells, we continue spiraling out until the *stop cell* is reached, ensuring that the closest point is found.

4.3.3 Exhaustive Query

Exhaustive queries are the worst case query performed if neither c nor any cells near c contain a fixed point. This is very rare, not happening once in any of the real 3D datasets that we tested. However an adversary could generate such a case. We exhaustively query by linearly searching all the fixed points, in parallel on the GPU.

5 Performance

All tests were run on an Intel i7-5820k with 32 GB of DDR4 memory and a Nvidia GTX 980Ti with 6GB of GDDR5 memory.

NearptD was implemented using Thrust 1.8.2 in C++ with CUDA 7.5, compiled using nvcc and clang++ 3.5 with level 3 optimization. (Thrust is an efficient API on top of C++ and CUDA that adds a functional programming paradigm.) For uniform datasets, a $G_r = 0.5$ was used, and for all other datasets $G_r = 1.0$.

Nearpt3 was compiled using clang++ 3.5 with level 3 optimization, using the same scheme to choose G_r as NearptD.

FLANN was compiled using clang++ 3.5 with level 3 optimization, preprocessing the data into a kd-tree with default parameters and performing a KNN search with default search parameters and $k = 1$.

The following datasets were used as real world comparisons of NearptD, Nearpt3 and FLANN in 3 dimensions. We are grateful to these projects for kindly making this data available.

- *uniXXX*: A uniformly and independently distributed set of 10^4 to 10^8 random points.
- *bunny* ($N_f = 35,947$): Stanford University Computer Graphics Laboratory[9].
- *hand* ($N_f = 327,323$): Clemson's Stereolithography Archive, via Georgia Tech[10].
- *dragon* ($N_f = 437,645$): Brian Curless, via Stanford and Georgia Tech.
- *bone6* ($N_f = 569,636$): The Visible Human Project, and William E. Lorensen, via Georgia Tech.
- *blade* ($N_f = 882,954$): Visualization Toolkit (VTK), via Georgia Tech.
- *powerplant* ($N_f = 5,423,053$): The complete powerplant from the University of North Carolina's UNC Chapel Hill Walkthru Project[11].
- *david* ($N_f = 28,168,109$), and
- *stmatthew* ($N_f = 184,098,599$): The Stanford Digital Michelangelo Project Archive[8].

NearptD has some fixed costs independent of data size, mainly the antepreprocessing step to create the cell search order necessary for slow case queries, as well

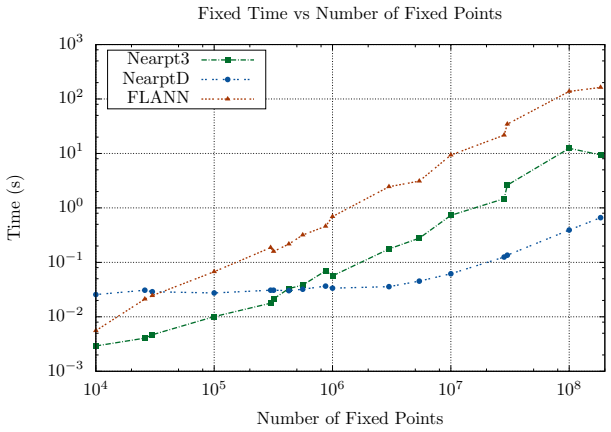

Figure 1: Time to preprocess fixed points into a search structure, not including I/O.

as overhead to run tasks on the GPU. For this reason, smaller datasets can take longer than existing programs, but on larger datasets, this cost is negligible. NearptD exhibits an order of magnitude speedup over Nearpt3 on larger datasets, and two orders of magnitude speedup vs FLANN. Figure 1 shows that NearptD becomes faster than both FLANN and Nearpt3 for preprocessing datasets of at least 10^6 points, with an order of magnitude speedup as the number of points increases.

Arguably the antepreprocessing costs should not be included any more than the compilation costs, since both are incurred only once, not once per dataset.

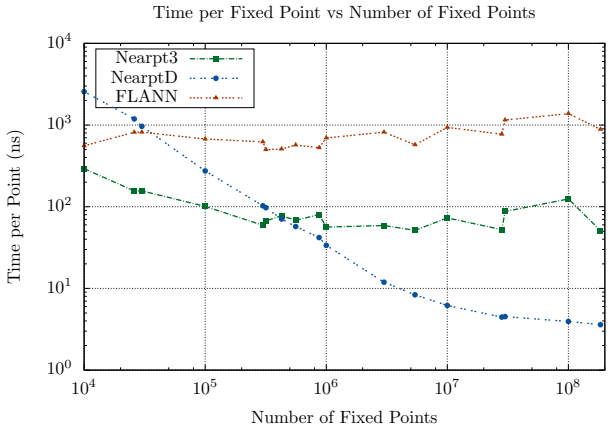

Figure 2: Per point time to preprocess fixed points into a search structure, not including I/O.

Figure 2 compares the time per point for each program to preprocess the fixed points into their respective search structures, and we see that NearptD benefits from its parallelism more on larger datasets. FLANN averages roughly $753ns$ per fixed point, and Nearpt3 averages around $95ns$ per fixed point. NearptD can take up to $2.5\mu s$ per fixed point for smaller datasets, but preprocesses *stmatthew* in just $3.5ns$ per fixed point.

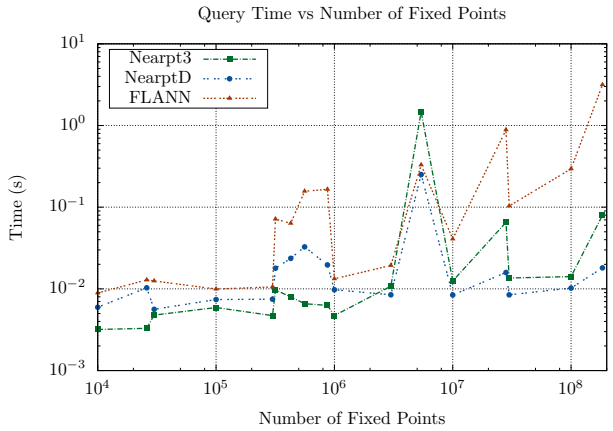

Figure 3: Time to complete 10^4 queries sampled from the fixed distribution.

To compare query times, each dataset had 10^4 points sampled from it to use as query points, with the rest preprocessed into a search structure. Figure 3 shows that as the number of fixed points increases, NearptD becomes faster than either existing program. For small datasets, it does not demonstrate any speedup, likely due to the GPU overheads necessary to compute these queries. Even for extremely adversarial data, such as the *powerplant* dataset (the large spike in Figure 3 at 5 million points), where 98% of fixed points are contained within one cell, NearptD still performs just well as FLANN. These results used an unoptimized grid resolution, however, as doubling the grid resolution reduces query time by 3 times, although it increases the memory usage.

The *powerplant* dataset is important because it disproves the notion that an adaptive dataset like the kd-tree will process uneven data better than the uniform grid. NearptD's query time is more than ten times slower than for a uniformly distributed dataset of the same size. FLANN's query time is also much slower, though by a smaller factor. However, NearptD started out faster, and the end result is that NearptD and FLANN have about the same query time here, with NearptD being a little faster.

There are two reasons. On such uneven data, the kd-tree has many levels and more of its cells are empty. These cells are allocated on the memory heap, whose time cost is superlinear (the more objects on the heap, the more time that allocating and freeing each object costs). Each query has to walk down the deep tree. In contrast, with the uniform grid, empty cells are almost free to allocate, since only the complete grid is allocated, and that in one step. Querying a grid with mostly empty cells is cheaper, the only unknown is how far we need

to spiral out.

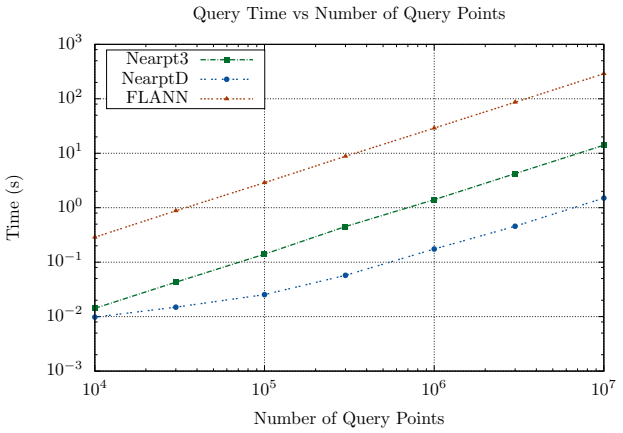

Figure 4: Time to complete queries on 10^8 fixed points vs number of queries.

Figure 4 shows the speedup NearptD has as the number of query points increases, with 10^8 fixed points. FLANN and Nearpt3 take 28.9μs and 1.4μs per query point, respectively, while NearptD takes 1.0μs per query on small numbers of queries and 0.15μs per query on larger numbers of queries.

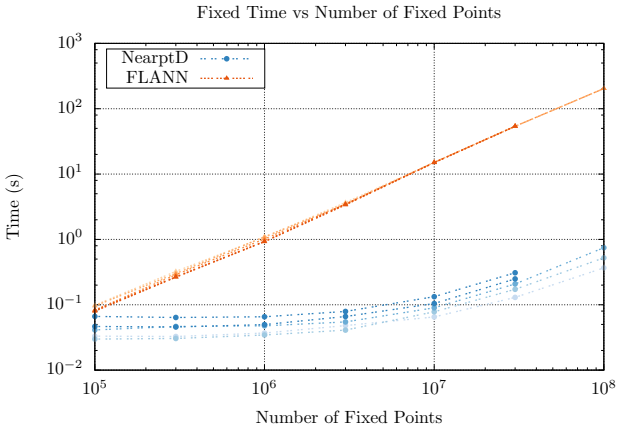

Figure 5: Time to preprocess fixed points into a search structure for varying number of dimensions. The darkest line is 6 dimensional data, with lighter colors indicating lower dimensions, down to 2.

For preprocessing fixed points, NearptD exhibits over two orders of magnitude speedup vs FLANN, even for higher dimensional data, as shown in Figure 5. At 5 and 6 dimensions, the 10^8 dataset did not fit into the GPU memory, so it is not shown. Preprocessing points into a uniform grid is largely independent of the dimensionality of the data, for both FLANN and NearptD.

Concerning the limited size of the GPU's memory: There will always be datasets too big to fit into the available memory. However, that occurs less often than is generally realized. Later in 2016, Nvidia GPUs with 32GB of memory are expected to become available. In addition, the bandwidth between the GPU and the CPU is increasing, so that the cost of the GPU accessing the CPU memory is shrinking. Indeed, one problem with our research into designing parallel algorithms on GPUs today, to process the large datasets expected in the future, is finding large real test datasets today.

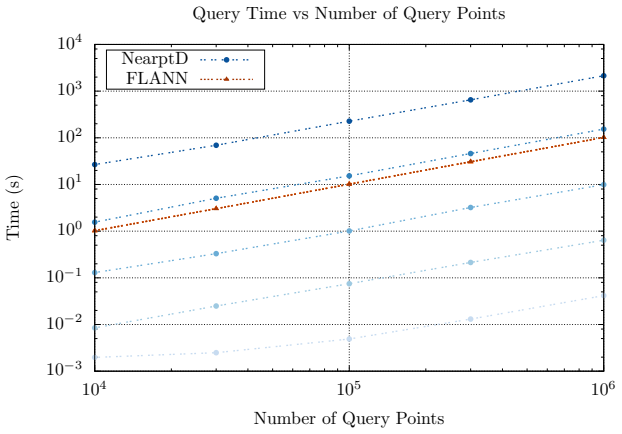

Figure 6: Time to perform queries on 10^7 fixed points for varying number of dimensions. The darkest line is 6 dimensional data, with lighter colors indicating lower dimensions, down to 2.

Figure 6 shows the time to perform queries against 10^7 fixed points for dimensions 2 to 6. NearptD exhibits an order of magnitude slowdown in query time for each extra dimension. While NearptD is significantly faster than FLANN in 2 to 4 dimensions, it performs worse as dimensionality increases. Query times for FLANN are completely independent of the dimensionality of the data.

6 Space Complexity

N_f	Fixed	NearptD	Nearpt3	FLANN
1M	5.7	115.9/137.7	36.9	263.8
10M	57.2	173.3/225.6	156.8	2594.1
100M	572.2	682.4/2223.6	1358.4	25897.6

Table 2: Comparison of total memory footprint of different programs, in MB, as well as the size of the fixed points. NearptD values are given as CPU/GPU memory usage.

Besides speed, another benefit of a uniform grid is the relatively small memory footprint to maintain the search structure. As the number of fixed points grows, almost the entirety of host memory used by NearptD

is dedicated to simply holding the fixed points. The uniform grid is constructed entirely on the GPU, as well as a copy of the fixed points. FLANN requires an order of magnitude more memory to hold the kd-tree in memory, almost exceeding the amount of host memory available in the largest test case.

7 Future Work

The main limiting factor for the data size NEARPTD can handle is the memory of the GPU. While the largest real world dataset, *stmatthew*, fits into the 6GB of memory on the test machine, higher dimensions can reduce the effective maximum size of the data that can be processed on the GPU. Although GPU memory is constantly increasing, modifying the NEARPTD algorithm to preprocess the data in chunks that could fit into GPU memory would allow for arbitrarily large datasets to be processed, especially in higher dimensions. Another possible solution would be to utilize Unified Memory in CUDA along with Thrust to process datasets too big to fit into GPU memory, but this could lead to lower performance with the high cost of moving large amounts of data between the CPU and GPU. Extending NEARPTD to utilize multiple GPUs could also help in processing large datasets.

When the Thrust library implements C++11 variadic templates for tuples, NEARPTD could be refactored to implement some of the C++11 features and use variadic templates, which would remove the need for template specialization. This could reduce the compilation time and make the code more straightforward.

NEARPTD could also be extended to a k nearest neighbor search by simply extending the query scheme to continue searching until the k nearest neighbors are found. For fast case queries where the cell contains at least k fixed points, this does not increase the running time in any significant manner. If a cell is empty or contains less than k points, a slow case queries could be performed until k fixed points are found, falling back on exhaustive querying only when necessary.

8 Conclusion

This paper presented NEARPTD as an improvement on more common nearest neighbor libraries that utilize kd-trees to preprocess data. The uniform grid used by NEARPTD has lower time and space complexity compared to traditional kd-trees and by utilizing the GPU, NEARPTD exhibits an order of magnitude speedup for larger datasets over existing libraries for both preprocessing and querying. On a dataset with over 184 million points, each point can be preprocessed into a search structure in just $3.5ns$. When performing 10 million queries on 100 million points, queries completed in an average of $0.15\mu s$. NEARPTD shows that a non hierarchical search structure can enable exact nearest neighbor searching to outperform even approximate searching using kd-trees.

The broader lesson from NEARPTD is that, counterintuively, simple data structures work better to process large datasets in parallel. We have other algorithms that also demonstrate this. We intend to make this code freely available for nonprofit research and education.

9 Acknowledgments

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] Varol Akman, W. Randolph Franklin, Mohan Kankanhalli and Chandrasekhar Narayanaswami Geometric Computing and the Uniform Grid Data Technique *Computer Aided Design* 21(7):410-420, 1989.

[2] W. Randolph Franklin Nearest Point Query on 184M Points in E^3 with a Uniform Grid *Canadian Conference on Computational Geometry* 17:239-242, 2005.

[3] Marius Muja and David G. Lowe Scalable Nearest Neighbor Algorithms for High Dimensional Data *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 36, 2014.

[4] Jon Louis Bentley Multidimensional binary search tress used for associative searching *Communications of the ACM* 18(9):509-517, 1975.

[5] Fukunaga, Keinosuke, and Patrenahalli M. Narendra A branch and bound algorithm for computing k-nearest neighbors *Computers, IEEE Transactions on* 100(7):750-753, 1975.

[6] Ting Liu, Andrew W. Moore, Alexander Gray and Ke Yang An Investigation of Practical Approximate Nearest Neighbor Algorithms *Advances in neural information processing systems* 825-832, 2004.

[7] CGAL The CGAL home page *http://www.cgal.org/* 2016.

[8] Marc Levoy The Digital Michelangelo Project *http://graphics.stanford.edu/projects/mich/* 2003.

[9] Marc Levoy The Stanford 3D Scanning Repository *http://graphics.stanford.edu/data/3Dscanrep/* 2005.

[10] Greg Turk and Brendan Mullins Large Geometric Models Archive *http://www.cc.gatech.edu/projects/large_models/* 2003.

[11] UNC Chapel Hill Walkthru Project Complete Power Plant Model *http://www.cs.unc.edu/~walk/* 1997.