Performance Analysis of Nearest Neighbor Algorithms for ICP Registration of 3-D Point Sets

Timo Zinßer*, Jochen Schmidt, Heinrich Niemann

Lehrstuhl für Mustererkennung, Universität Erlangen-Nürnberg
Martensstraße 3, 91058 Erlangen, Germany
zinsser@informatik.uni-erlangen.de

Abstract

There are many nearest neighbor algorithms tailor-made for ICP, but most of them require special input data like range images or triangle meshes. We focus on efficient nearest neighbor algorithms that do not impose this limitation, and thus can also be used with 3-D point sets generated by structure-from-motion techniques. We shortly present the evaluated algorithms and introduce the modifications we made to improve their efficiency. In particular, several enhancements to the well-known k-D tree algorithm are described. The first part of our performance analysis consists of experiments on synthetic point sets, whereas the second part features experiments with the ICP algorithm on real point sets. Both parts are completed by a thorough evaluation of the obtained results.

1 Introduction

The ICP (Iterative Closest Point) algorithm is the most popular algorithm for registration of 3-D point sets. A comprehensive summary of different variants of this algorithm can be found in [1]. The ICP main loop consists of two operations:

- matching points of the data point set to points in the model point set,
- computing the motion which best aligns the pairs of corresponding points.

When no additional information is available, the matching is based solely on the distance of points, and thus can be formulated as a nearest neighbor problem. As the nearest neighbor search is the single most time-consuming processing step of the ICP algorithm, improving the computation time of the nearest neighbor algorithm will also greatly increase the performance of the ICP algorithm.

The nearest neighbor search can be enhanced by using additional information. When the point sets are given as range images or triangle meshes, the implicit neighborhood relations of the points can be utilized to speed up the search [2]. Other methods take advantage of the fact that range images and triangle meshes also describe object surfaces [1]. As we focus on working with plain point sets, for example generated by structure-from-motion techniques (cf. e.g. [3]), such methods lie outside the scope of this work.

Recently, Greenspan *et al.* proposed TINN (Triangle Induced Nearest Neighbor), a novel nearest neighbor algorithm for small point sets [4]. Sorting the points by their distance to a reference point ensures that only a contiguous subset of points has to be considered. A similar algorithm was devised by Friedman *et al.* in 1975 [5]. We will carefully analyze the performance of both methods in this work.

The k-D tree algorithm introduced by Friedman *et al.* in [6] is widely used for nearest neighbor search in large point sets. Several refinements of this algorithm were described by Sproull [7] and by Arya and Mount [8]. Greenspan *et al.* proposed using a simple nearest neighbor algorithm like TINN for searching the leaf nodes of the k-D tree [4]. This extension should both speed up the search and reduce the memory requirements of the k-D tree algorithm. We will describe the k-D tree algorithm and its possible refinements, as well as the hybrid k-D tree with enhanced leaf node search. Furthermore, we will analyze the performance of the different approaches on synthetic point sets and with the ICP algorithm on real point sets.

Another algorithm often used for searching in large point sets is Elias' algorithm. It is for example examined by Cleary in [9]. As this algorithm uni-

*This work was partially funded by the European Comission's 5th IST Programme under grant IST-2001-34401 (project VAMPIRE). Only the authors are responsible for the content.

formly partitions the space occupied by the point set, it is not suited well for the irregular point sets produced by our structure-from-motion techniques. Therefore, Elias' algorithm will not be considered any further in this work.

Finally, we examine the benefits of the STCNN (Spherical Triangle Constraint Nearest Neighbor) enhancement for the k-D tree, a search algorithm specifically created for ICP by Greenspan *et al.* [10]. In a preprocessing step, a neighborhood of points is generated for each model point. If the motion update after an iteration of the ICP algorithm is small, it should be sufficient to search the neighborhood of the last nearest neighbor to find the new one. We will evaluate the efficiency of this algorithm with our optimized ICP algorithm [11].

After a short description of the nearest neighbor problem in the next section, we will describe each examined nearest neighbor algorithm in its own section. There are two sections for the performance analysis. In Sect. 7, we will demonstrate the results of all experiments on synthetic point sets, and in Sect. 8, the experiments with the ICP algorithm on real point sets will be evaluated. We finish this work with a conclusion in Sect. 9.

2 Problem Definition

The nearest neighbor problem can easily be stated in a mathematical way. Let P be a set of n points $\boldsymbol{p}_i \in \mathbb{R}^3$, and let $\boldsymbol{q} \in \mathbb{R}^3$ be a query point. Then the nearest neighbor problem for point set P and query point $\boldsymbol{q}$ is to find the point $\boldsymbol{p}_c \in P$ with

$$\|\boldsymbol{q} - \boldsymbol{p}_c\| \leq \|\boldsymbol{q} - \boldsymbol{p}_i\| \quad \forall\, \boldsymbol{p}_i \in P\ . \qquad (1)$$

A simple solution for the nearest neighbor problem is the exhaustive search algorithm. For all n points $\boldsymbol{p}_i \in \mathbb{R}^3$, the distance to the query point $\boldsymbol{q}$ is computed, and the point $\boldsymbol{p}_c$ with the smallest distance is returned. In the implementation of this simple algorithm, two optimizations can decrease the required computation time. Firstly, the time-consuming evaluation of square roots can be circumvented by working with squared distances. Secondly, the evaluation of the sum

$$\|\boldsymbol{q} - \boldsymbol{p}\|^2 = \sum_{j=1}^{3} (q_j - p_j)^2 \qquad (2)$$

can be aborted if a partial sum is already greater than the smallest squared distance of the current nearest neighbor point. Both optimizations are also used in the other algorithms described in this paper.

Ultimately, all efficient nearest neighbor algorithms are based on the same idea. A suitable preprocessing step for point set P is employed to reduce the number of points that have to be examined during the nearest neighbor search. Thereby, additional operations must not take longer than the distance calculations that are superseded by them.

For the evaluation of different nearest neighbor algorithms, several criteria have to be considered:

- memory requirements,
- computation time for initialization,
- computation time for nearest neighbor search,
- computation time for adopting changes in P.

These criteria must be judged with respect to the application area of the algorithm.

As we concentrate on nearest neighbor algorithms for ICP with 3-D point sets, some conclusions can already be drawn. When working with moderately sized point sets, the memory requirements of the nearest neighbor algorithm should not be a problem. The computation time for nearest neighbor search is most important, because a search must be performed for each data point at each iteration of the ICP algorithm. In contrast, the initialization only has to be done once, and the model point set does not change during one run of ICP.

3 The TINN Algorithm

The TINN (Triangle Induced Nearest Neighbor) algorithm was proposed by Greenspan *et al.* in [4]. During the initialization, all points in point set P are sorted by their distance to a reference point $\boldsymbol{r}$, which can be chosen arbitrarily. Therefore, the time complexity of the initialization is $\mathrm{O}(n \log n)$. The memory requirements are also very modest, only one extra float value has to be stored for each point.

During the nearest neighbor search, a contiguous subset of points is searched. At first, the distance d_q of query point $\boldsymbol{q}$ from reference point $\boldsymbol{r}$ is calculated. Then, the point with the most similar distance from $\boldsymbol{r}$ is chosen as starting point $\boldsymbol{p}_s$. This can be implemented efficiently using a binary search over the sorted point set.

After checking the starting point, the points on the left and on the right of the starting point are searched for the nearest neighbor of $\boldsymbol{q}$. There are several possibilities for the search order:

- search on one side of $\boldsymbol{p}_s$, then on the other,
- search alternately on both sides of $\boldsymbol{p}_s$,
- search the next remaining point with a stored distance most similar to d_q.

When the numbers of searched points are considered, none of the given alternatives has a clear advantage over the others. Consequently, the third method should not be used, as it requires additional operations for determining the next point.

Let $\boldsymbol{p}_{\min}$ be the best point found so far. Then the stopping criterion for searching the points on either side of point $\boldsymbol{p}_s$ can be derived from the triangle inequality:

$$\begin{aligned} &|\|\boldsymbol{q}-\boldsymbol{r}\| - \|\boldsymbol{p}_i-\boldsymbol{r}\|| > \|\boldsymbol{q}-\boldsymbol{p}_{\min}\| \\ &\quad\Rightarrow \|\boldsymbol{q}-\boldsymbol{p}_i\| > \|\boldsymbol{q}-\boldsymbol{p}_{\min}\| \,. \end{aligned} \tag{3}$$

As soon as one point $\boldsymbol{p}_i$ satisfies the inequality in the first line, all other points on the same side of point $\boldsymbol{p}_s$ also satisfy this inequality. Therefore, no other point on the same side of point $\boldsymbol{p}_s$ can be closer to $\boldsymbol{q}$ than $\boldsymbol{p}_{\min}$.

No expected time complexity is specified by Greenspan *et al.*, but experimental evaluation suggests a time complexity of $\mathrm{O}(n^{2/3})$ for three-dimensional data. Thus, TINN should only be used for small point sets of up to 500 points.

4 The CAS Algorithm

25 years before TINN, a similar algorithm was presented by Friedman *et al.* in [5]. We will refer to it as CAS (Coordinate Axis Sort) for brevity. Here, the points are sorted along one coordinate axis during initialization. Interestingly, CAS can be considered a specialization of TINN, because it is generated by moving the reference point to infinity along one coordinate axis.

Although TINN and CAS are very similar, there are some important differences. Firstly, there is no need to store an additional distance for each point. Secondly, by choosing the coordinate axis with the largest variance for sorting the points, the average number of searched points can easily be optimized. Last but not least, the stopping criterion is computationally less expensive, because only one component a of the point vectors has to be considered:

$$\begin{aligned} &|q_a - p_{i,a}| > \|\boldsymbol{q}-\boldsymbol{p}_{\min}\| \\ &\quad\Rightarrow \|\boldsymbol{q}-\boldsymbol{p}_i\| > \|\boldsymbol{q}-\boldsymbol{p}_{\min}\| \,. \end{aligned} \tag{4}$$

Taking these differences into account, it becomes obvious that TINN can only be faster than CAS if its sorting scheme excludes more points from the search. Whether this proposition holds will be evaluated experimentally in Sect. 7.

5 The k-D Tree Algorithm

The k-D tree nearest neighbor algorithm was introduced by Friedman *et al.* in [6]. Due to its immense popularity, many refinements have been developed since its conception, for example by Sproull [7] and by Arya and Mount [8].

The basic idea of the k-D tree algorithm is to recursively partition a point set P by hyperplanes, and to store the obtained partitioning in a binary tree. Sproull described several alternatives for choosing the orientation of the hyperplanes:

- The hyperplane is orthogonal to a coord. axis.
  - Use alternating coordinate axes.
  - Use coordinate axis where the current subset has the largest spread.
  - Use coordinate axis where the current subset has the largest variance.
- The hyperplane is orthogonal to the principal axis of the current subset.

We use the "coordinate axis / variance" method, as it constitutes a good compromise between computational cost and reduction of examined points.

According to Sproull, the position of the hyperplane can be defined by either

- the bisector of the range of coordinate values,
- the mean coordinate value,
- or the median coordinate value.

Choosing the median coordinate value of the current subset involves additional computational cost during the initialization, but allows us to build a perfectly balanced binary tree. Thus, in any layer of the tree, the number of points in the corresponding subsets differs at most by one.

Usually, each inner node of the tree stores one pointer to both of its children. However, we do not need these pointers, but store the inner nodes of the tree in an array and access them via their index. If the tree has l levels of inner nodes, it contains $2^l - 1$ inner nodes. Let the root node have index 0. When an inner node has index i, the indices of its children are $2i + 1$ and $2i + 2$.

Additionally, we save memory by working with "logical" leaf nodes. Point set P is initially stored

in one large array. During the initialization, for each inner node the current subset is sorted along the chosen coordinate axis, so that both originating smaller subsets remain contiguous. The recursive search starts at the root node, which represents all points from indices 0 to $n-1$. When an inner node represents points from indices i to j, its children represent points from indices i to $\lfloor (i+j)/2 \rfloor$ and from indices $\lfloor (i+j)/2 \rfloor + 1$ to j accordingly. As the start and end index for each node can easily be calculated during the recursive search, no extra pointers and no physical leaf nodes are required.

Unlike Friedman, we do not store a bounds array for each node. Instead, our inner nodes only contain the index of the coordinate axis orthogonal to the hyperplane and the distance of the hyperplane to the origin. Arya and Mount show how to efficiently compute exact distances to the cells of the k-D tree in this case [8]. Also unlike Friedman, we do not use a "ball within bounds"-test when a new best point is encountered, because the computational cost for these tests is higher than that for the final backtracking to the root node.

On the whole, the recursive search procedure of our implementation is very lean. At first, the child that is on the same side of the partitioning hyperplane as the query point is examined by a new invocation of the procedure. When the search backtracks to the current node, the other child is only considered if its distance from the query point is smaller than that of the currently best point.

The memory requirements of our k-D tree algorithm are $\mathrm{O}(n)$, because a perfectly balanced tree with n leaf nodes has exactly $n-1$ inner nodes. The time complexity of the initialization is $\mathrm{O}(n \log^2 n)$ for our variant, due to the computation of the exact median of the point coordinates for splitting each inner node. Friedman proved that the expected search time for the k-D tree algorithm is proportional to $(\log n)$ [6].

It is of particular importance to identify the optimum value for the only parameter of the k-D tree, the maximum number of points in the leaf nodes b. It is obvious that the average number of examined points is minimal for $b = 1$. But although storing more points in each leaf node increases the average number of examined points, it also decreases the number of layers of the tree and the number of recursive calls of the search procedure. We will show how to determine b experimentally in Sect. 7.

The hybrid k-D tree algorithm was proposed by Greenspan *et al.* in [4] and uses the TINN algorithm for searching its leaf nodes. This enhancement should increase the optimum value for b, thereby reducing the levels of the tree and the memory requirements of the algorithm. Provided that TINN is better suited for searching small point sets than the k-D tree algorithm, the performance of the hybrid algorithm should also improve. We implemented the described hybrid k-D tree algorithm, but opted for the CAS algorithm instead of the TINN algorithm.

6 The STCNN / k-D Tree Algorithm

The STCNN (Spherical Triangle Constraint Nearest Neighbor) enhancement for the k-D tree algorithm was specifically developed by Greenspan *et al.* to speed up the nearest neighbor search of ICP [10]. It is based on the observation that the motion updates of the ICP algorithm become small after the first few iterations. Thus, the nearest neighbor points of the previous iteration often constitute a useful approximation for those of the current iteration.

During the initialization, a list of neighbor points is generated for each model point. Let $\boldsymbol{p}_c^{\mathrm{old}}$ be the nearest neighbor of the previous query point $\boldsymbol{q}^{\mathrm{old}}$. Furthermore, let $L = \{\boldsymbol{p}_1^{\mathrm{nb}}, \ldots, \boldsymbol{p}_k^{\mathrm{nb}}\}$ be the list of neighbor points for $\boldsymbol{p}_c^{\mathrm{old}}$, sorted by increasing distance from $\boldsymbol{p}_c^{\mathrm{old}}$. Then, the nearest neighbor search for query point $\boldsymbol{q}$ can be accelerated if the following inequation is met:

$$\begin{aligned} \|\boldsymbol{q} - \boldsymbol{p}_c^{\mathrm{old}}\| \;&\leq\; \frac{1}{2}\,\|\boldsymbol{p}_k^{\mathrm{nb}} - \boldsymbol{p}_c^{\mathrm{old}}\| \\ &\Rightarrow \boldsymbol{p}_c \in \{\boldsymbol{p}_c^{\mathrm{old}}\} \cup L\,. \end{aligned} \tag{5}$$

If the inequation is not met, a conventional search in the k-D tree has to be performed.

Greenspan *et al.* limit the number of neighbor points by specifying their maximum distance from the considered model point. Unfortunately, the relation of maximum distance and average number of neighbor points is strongly affected by the shape of point set P. In order to be able to adjust the memory requirements of the algorithm more directly, we explicitly set the number of stored neighbor points. Whether the accelerated search of the STCNN / k-D tree algorithm can offset its more costly initialization will be evaluated experimentally in Sect. 8.

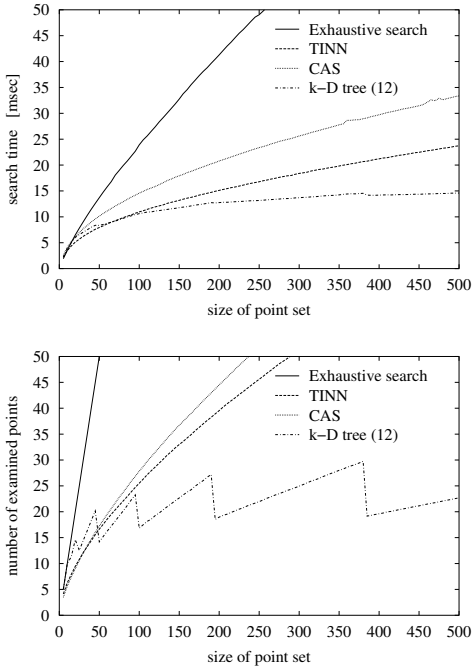

Figure 1: Comparison of nearest neighbor algorithms for small point sets. The maximum number of points in the leaf nodes for the k-D tree algorithm is given in parentheses.

7 Experiments on Synthetic Point Sets

In this section, we evaluate the performance of the described nearest neighbor algorithms on synthetic point sets, which were generated by uniformly distributing points in a unit cube. The query points were also uniformly distributed in the same unit cube. Each of the experiments performed in this section is illustrated by one figure with two charts. The upper chart shows the measured search times in milliseconds per 10000 query points. For the measurements, a PC with an Athlon XP 1500+ cpu and a Via KT266A chipset was used.

The lower chart displays the average number of examined points per query point. Hereby, a point is counted when its distance to the query point is evaluated. Consequently, computations that access control structures like the internal nodes of a k-D tree are not reflected in this number.

The results of the first experiment, which was conducted for small point sets of up to 500 points, are presented in Fig. 1. Even for very small point sets, the exhaustive search is clearly outperformed by all other algorithms. It is interesting to note that the search time of the exhaustive search does not increase linearly with the size of point set P. This effect is caused by the partial sum evaluation described in Sect. 2.

The TINN algorithm is slower than the older CAS algorithm for all sizes of point set P. As its search operations are more costly, only a reduction of the number of examined points could accelerate the TINN algorithm. However, it can clearly be seen in the lower chart of Fig. 1 that the sorting criterion of the CAS algorithm is more efficient for the given point sets.

Greenspan *et al.* show in [4] that TINN is faster than their implementation of the k-D tree algorithm for point sets of up to 275 points. In contrast to this, our adapted version of the k-D tree algorithm is faster than TINN for point sets of any size. Additionally, the k-D tree algorithm is only beaten by the CAS algorithm for point sets with less than 100 points. These results document the efficiency of our enhancements for the k-D tree algorithm.

In the second experiment, the performance of the nearest neighbor algorithms was evaluated for large point sets of up to one million points. We left out the exhaustive search, which is not very useful for large point sets. Instead, we included the hybrid k-D tree into the set of tested algorithms. The results of this experiment are shown in Fig. 2. Please take note that the horizontal axes of the charts are scaled logarithmically.

As expected, the TINN and CAS algorithms cannot keep up with the k-D tree based algorithms for large point sets. Both k-D tree algorithms perform well, but the hybrid k-D tree can slightly surpass the standard k-D tree. Due to its larger leaf nodes, the hybrid k-D tree has fewer levels of inner nodes. The second chart of Fig. 2 also shows that the hybrid algorithm examines fewer points. Apparently, these advantages are almost compensated by the more complicated search procedure within the leaf nodes.

Because of the logarithmic scale of the horizontal axis, a linear graph should be expected for the k-D tree algorithm in the upper chart. This behavior can also be observed, except for the kink at approximately 10000 points, which can be explained by the internal structure of the central processing unit. When the point sets reach the size of 10000

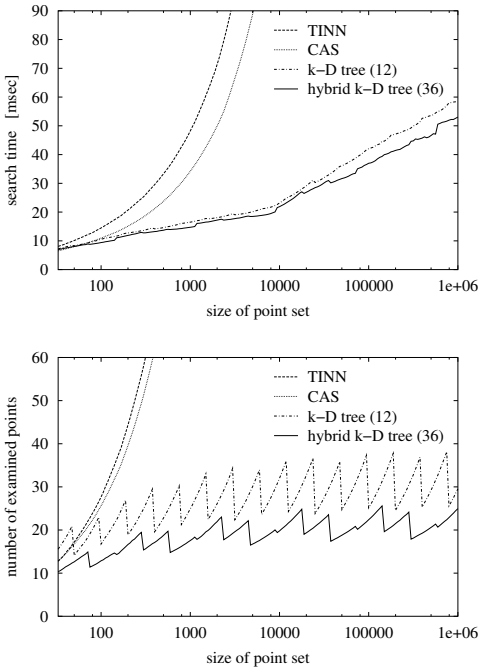

Figure 2: Comparison of nearest neighbor algorithms for large point sets. The maximum numbers of points in the leaf nodes for the k-D tree algorithm are given in parentheses.

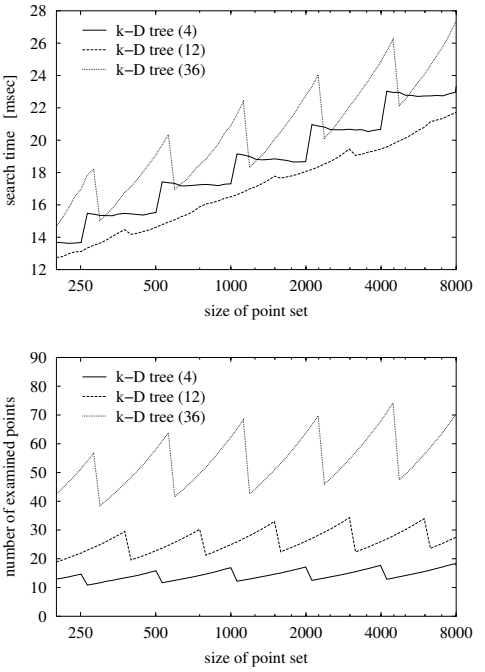

Figure 3: Comparison of k-D tree algorithms for different number of points in leaf nodes. The maximum numbers of points in the leaf nodes for the k-D tree algorithm are given in parentheses.

points, the data structures no longer fit into the internal cache of the cpu and have to be swapped to the slower main memory.

Whenever all leaf nodes are filled to the maximum, a new layer has to be added to the k-D tree, and the number of points in the leaf nodes is almost exactly halved. As a consequence, the number of examined points also decreases. This behavior can be observed for both k-D tree algorithms in the lower chart of Fig. 2. But every third time, the number of examined points decreases only slightly for the hybrid k-D tree. This phenomenon can be explained with the cubic shape of the synthetic point set and should not appear when working with real point sets.

The choice of the maximum number of points in the leaf nodes b is very important for the performance of the k-D tree algorithm. Therefore, we tested the k-D tree algorithm with different values for parameter b in our third experiment. As the actual number of points in the leaf nodes can lie between $b/2$ and b, depending on the size of the point set, we chose $b \in \{4, 12, 36\}$, so that different trees are built for each possible size of point set P. The obtained results for search time and number of examined points are depicted in Fig. 3.

The lower chart shows that the number of examined points increases with the size of the leaf nodes. But at the same time, the number of levels of the k-D tree decreases. Consequently, the fastest search time can be achieved by finding a compromise between point examination and tree traversal. This compromise depends on the implementation, the compiler, and the hardware and can only be determined experimentally.

In the upper chart, the k-D tree with $b = 4$ suffers a performance penalty whenever a new tree level is added. This clearly shows that new levels are added too early and that larger leaf nodes should be used. In contrast, the k-D tree with $b = 36$ gains much

Figure 4: Point set used for performance evaluation.

	A	B	C	D
exh. search	8170	6767	5297	4582
TINN	1362	1157	753	610
CAS	885	764	514	417
k-D tree	629	533	405	351
hyb. k-D tree	597	506	384	331
STCNN (5)	594	509	377	325
STCNN (10)	550	483	**353**	**306**
STCNN (15)	**547**	**479**	359	308
STCNN (20)	550	486	368	324
STCNN (25)	554	493	380	330

Table 1: Computation times of ICP algorithm for different nearest neighbor algorithms in msec. The numbers of neighbor points used for STCNN are given in parentheses.
A: standard ICP algorithm
B: A + motion extrapolation
C: B + point selection hierarchy with two levels
D: B + point selection hierarchy with four levels

speed when a new level is added to the tree. This behavior indicates that the new level should have been added earlier by using a smaller maximum number of points in the leaf nodes. Finally, the k-D tree with $b = 12$ does not show any abrupt changes in its average search time. Therefore, its maximum number of points in the leaf nodes is optimal.

8 Experiments with ICP

In this section, we evaluate the performance of all described nearest neighbor algorithms with an ICP algorithm on a real point set. We employ an ICP variant that can be accelerated by motion extrapolation and by hierarchical data point selection. The motion extrapolation aims at reducing the number of iterations needed for convergence of the algorithm. Simultaneously, the hierarchical data point selection can improve the computation time by using only every 2^i-th data point on the i-level of the hierarchy (see [11] for further details).

The point set in Fig. 4 contains 3600 points, has a height of approximately 2 units, and was reconstructed from a video by applying structure-from-motion techniques. It was directly used as the set of model points for the ICP algorithm. A set of data points was created by rotating this point set by 30 degrees, moving it by 0.17 units and adding Gaussian noise with a standard deviation of 0.01 units. With this configuration, the ICP algorithm reliably converges to the correct registration.

The results of our experiment are presented in Table 1. We used the hybrid k-D tree as the basis for the STCNN extension, because it has proven to be the fastest of our standard nearest neighbor algorithms. The given computation times also include the motion estimation of the ICP algorithm. For example, these computations take approximately 300 msec in column A and 180 msec in column D.

There are no surprises for the nearest neighbor algorithms already tested in the previous section. The CAS algorithm is considerably faster than TINN, and the hybrid k-D tree algorithm has a small lead over the standard k-D tree algorithm. As expected, the hybrid k-D tree algorithm is the fastest of the previously tested algorithms.

The STCNN extension manages to improve the results of the hybrid k-D tree algorithm. But the results indicate that the determination of the optimum number of neighbor points is difficult. In column A, initialization takes 40 msec and search takes 210 msec for 10 neighbor points. In contrast to this, for 20 neighbor points initialization takes 60 msec

and search takes 190 msec. Thus, computing more neighbor points slows down the initialization, but accelerates the search. As the number of iterations of the ICP algorithm is not known in advance, it is difficult to choose the optimum value for this important parameter.

The different columns of Table 1 show that enhancing the ICP algorithm is also very important for the final speed of the algorithm. When employing the motion extrapolation and the point selection hierarchy, the computation time is almost halved.

9 Conclusion

In the first part of this work, we described several algorithms for efficient nearest neighbor search in three-dimensional point sets. In particular, we proposed to enhance the well-known k-D tree algorithm by creating a perfectly balanced tree structure. As a consequence, our variant of the k-D tree algorithm completely circumvents the use of pointers for connecting its nodes. Additionally, we presented a more direct way of controlling the memory requirements of the STCNN enhancement, which was specifically created for accelerating the nearest neighbor search of the ICP algorithm.

We analyzed the performance of the nearest neighbor algorithms by experiments with synthetic point sets. The old CAS algorithm due to Friedman *et al.* clearly outperformed the newer TINN algorithm proposed by Greenspan *et al.* For point sets with more than 100 points, our enhanced k-D tree algorithm is faster than both TINN and CAS. An additional performance increase can be achieved by using the hybrid k-D tree algorithm. As a part of our experimental evaluation, we showed how to find the optimum number of points in the leaf nodes for the k-D tree algorithm.

Finally, we tested the nearest neighbor algorithms with an ICP algorithm on a real point set. For the previously tested algorithms, the results of the synthetic tests could be verified. What is more, the STCNN enhancement for nearest neighbor search proved to further accelerate the ICP algorithm. But it must also be noted that directly enhancing the speed of the ICP algorithm by motion extrapolation and the use of a point selection hierarchy has a far stronger impact on the final speed of the algorithm than accelerating the nearest neighbor search alone.

References

[1] S. Rusinkiewicz and M. Levoy, "Efficient Variants of the ICP Algorithm", *Proceedings of the 3rd International Conference on 3-D Digital Imaging and Modeling*, Quebec City, Canada, May 2001, pp. 145–152.

[2] T. Jost and H. Hügli, "Fast ICP Algorithms for Shape Registration", *Pattern Recognition – 24th DAGM Symposium*, Zurich, Switzerland, September 2002, pp. 91–99.

[3] R. I. Hartley and A. Zisserman, "*Multiple View Geometry in Computer Vision*", Cambridge University Press, 2000.

[4] M. Greenspan, G. Godin, and J. Talbot, "Acceleration of Binning Nearest Neighbour Methods", *Vision Interface 2000*, Montreal, Canada, May 2000, pp. 337–344.

[5] J. H. Friedman, F. Baskett, and L. J. Shustek, "An Algorithm for Finding Nearest Neighbors", *ACM Transactions on Computers*, 1975, pp. 1000–1006.

[6] J. H. Friedman, J. L. Bentley, and R. A. Finkel, "An Algorithm for Finding Best Matches in Logarithmic Expected Time", *ACM Transactions on Mathematical Software*, vol. 3, no. 3, 1977, pp. 209–226.

[7] R. F. Sproull, "Refinements to Nearest-Neighbor Searching in k-Dimensional Trees", *Algorithmica*, vol. 6, 1991, pp. 579–589.

[8] S. Arya, D. M. Mount, "Algorithms for Fast Vector Quantization", *Proceedings of the IEEE Data Compression Conference*, Snowbird, Utah, 1993, pp. 381–390.

[9] J. G. Cleary, "Analysis of an Algorithm for Finding Nearest Neighbours in Euclidean Space", *ACM Transactions on Mathematical Software*, vol. 5, no. 2, 1979, pp. 183–192.

[10] M. Greenspan and G. Godin, "A Nearest Neighbor Method for Efficient ICP", *Proceedings of the 3rd International Conference on 3-D Digital Imaging and Modeling*, Quebec City, Canada, May 2001, pp. 161–168.

[11] T. Zinßer, J. Schmidt, and H. Niemann, "A Refined ICP Algorithm for Robust 3-D Correspondence Estimation", *Proceedings of the International Conference on Image Processing*, Barcelona, Spain, September 2003, to appear.