Almost constant-time 3D nearest-neighbor lookup using implicit octrees

Bertram H. Drost¹ · Slobodan Ilic²

Received: 27 October 2016 / Revised: 22 August 2017 / Accepted: 20 October 2017 / Published online: 12 December 2017
© The Author(s) 2017. This article is an open access publication

Abstract A recurring problem in 3D applications is nearest-neighbor lookups in 3D point clouds. In this work, a novel method for exact and approximate 3D nearest-neighbor lookups is proposed that allows lookup times that are, contrary to previous approaches, nearly independent of the distribution of data and query points, allowing to use the method in real-time scenarios. The lookup times of the proposed method outperform prior art sometimes by several orders of magnitude. This speedup is bought at the price of increased costs for creating the indexing structure, which, however, can typically be done in an offline phase. Additionally, an approximate variant of the method is proposed that significantly reduces the time required for data structure creation and further improves lookup times, outperforming all other methods and yielding almost constant lookup times. The method is based on a recursive spatial subdivision using an octree that uses the underlying Voronoi tessellation as splitting criteria, thus avoiding potentially expensive backtracking. The resulting octree is represented implicitly using a hash table, which allows finding the leaf node a query point belongs to with a runtime that is logarithmic in the tree depth. The method is also trivially extendable to 2D nearest neighbor lookups.

Keywords Nearest neighbors · Voronoi cells · 3D point cloud processing

Mathematics Subject Classification 65D19

1 Introduction and overview

Quickly finding the point closest to some query point from a large set of data points in 3D is crucial for alignment algorithms, such as ICP [4], as well as industrial inspection and robotic navigation tasks. Most state-of-the-art methods for solving the nearest-neighbor problem in 3D are based on recursive subdivisions of the underlying space to form a tree of volumes. The various subdivision strategies include uniform subdivisions, such as octrees [21], as well as nonuniform subdivisions, such as k -d-trees [3] and Delaunay- or Voronoi-based subdivisions [10].

Tree-based methods require two steps to find the exact nearest neighbor. First, the query point descends the tree to find its corresponding leaf node. Since the query point might be closer to the boundary of the node's volume than to the data points contained in the leaf node, tree backtracking is required as a second step to search neighboring volumes for the closest data point.

The proposed method improves on both steps: the time for finding the leaf node is reduced by using a regular octree that is implicitly stored in a hash table, and the need for backtracking is eliminated by building the octree upon the Voronoi tessellation. The leaf voxel that contains the query point is found by bisecting the voxel level. For trees of depth L , this approach requires only $\mathcal{O}(\log(L))$ operations, instead of $\mathcal{O}(L)$ operations when letting the query point descend the tree. In addition, each voxel contains a list of all data points whose Voronoi cells intersect that voxel, such that no backtracking is necessary. By storing the voxels in a hash table and enforcing a limit on the number of Voronoi intersections per voxel, the total query time

✉ Bertram H. Drost
drost@mvtec.com

Slobodan Ilic
Slobodan.Ilic@siemens.com

¹ MVTec Software GmbH, Arnulfstr. 205, 80634 Munich, Germany

² Siemens AG, Otto-Hahn-Ring 6, 81739 Munich, Germany

is independent of the position of the query point and the distribution of data points. The query time is of magnitude $\mathcal{O}(\log(\log(N)))$, where N is the size of the target data point set.

The amount of backtracking that is required in tree-based methods depends on the position of the query point. Methods based on backtracking therefore have non-constant query times, making them difficult to use in real-time applications. Since the proposed method does not require backtracking, the query time becomes almost independent of the position of the query point. Further, the method is largely parameter free, does not require an a-priori definition of a maximum query range, and is straightforward and easy to implement.

We evaluate the proposed method on different synthetic datasets that show different distributions of the data and query point sets, and compare it to several state-of-the-art methods: a self-implemented k -d-tree, the *Approximate Nearest Neighbor* (ANN) library [22] (which, contrary to its name, allows also to search for exact nearest neighbors), the *Fast Library for Approximate Nearest Neighbors* (FLANN) [23], and the *Extremely Fast Approximate Nearest-Neighbor search Algorithm* (EFANNA) [15] framework. The experiments show that the proposed method is significantly faster for larger data sets and shows an improved asymptotic behavior. As a trade-off, the proposed method uses a more expensive preprocessing step.

We also evaluate an extension of the method that performs approximate nearest-neighbor lookups, which is faster for both the preprocessing and the lookup steps. Finally, we demonstrate the performance of the proposed method within two applications on real-world datasets, pose refinement and surface inspection. The runtime of both applications is dominated by the nearest-neighbor lookups, which is why both greatly benefit from the proposed method.

2 Related work

An extensive overview over different nearest-neighbor search strategies can be found in [25]. Nearest-neighbor search strategies can roughly be divided into tree-based and hash-based approaches. Concerning tree-based methods, variants of the k -d-tree [3] are state-of-the-art for applications such as ICP, navigation and surface inspection [14]. For high-dimensional datasets, such as images or image descriptors, embeddings into lower-dimensional spaces are sometimes used to reduce the complexity of the problem [20].

Many methods were proposed for improving the nearest-neighbor query time by allowing small errors in the computed closest point, *i.e.*, by solving the approximate nearest-

neighbor problem [1,8,18]. While faster, using approximations changes the nature of the lookup and is only applicable for methods such as ICP, where a small number of incorrect correspondences can be dealt with statistically. Fu and Cai [15] build a graph between nearest neighbors, allowing them to find approximate nearest neighbors using a graph search. Given a potential nearest neighbor, its neighbors are evaluated on the *the neighbor of my neighbor might be my neighbor* premise. This leads to highly efficient queries in higher dimensions, at the cost of preprocessing time. The iterative nature of ICP can be used to accelerate subsequent nearest-neighbor lookups through caching [17,24]. Such approaches are, however, only usable for ICP and not for defect detection or other tasks.

Yan and Bowyer [26] proposed a regular 3D grid of voxels that allow constant-time lookup for a closest point, by storing a single closest point per voxel. However, such fixed-size voxel grids use excessive amounts of memory and require a trade-off between memory consumption and lookup speed. The proposed multi-level adaptive voxel grid overcomes this problem, since more and smaller voxels are created only at the interesting parts of the data point cloud, while the speed advantage of hashing is mostly preserved. Glassner [9,16] proposed to use a hash table for accessing octrees, which is the basis for the proposed approach.

Using Voronoi cells is a natural way to approach the nearest neighbor problem, since a query point is always contained in the Voronoi cell of its nearest neighbor. Boada et al [7] proposed an octree that approximates generalized Voronoi cells and that can be used to approximately solve the nearest-neighbor problem [6]. Their work also gives insight into the construction costs of such an octree. Contrary to the proposed algorithm, their work concentrates on the construction of the data structure and solves the nearest-neighbor problem only approximately. Additionally, their proposed octree still requires $\mathcal{O}(\text{depth})$ operations for a query, for an octree of average depth depth . However, their work indicates how the proposed method can be generalized to other metrics and to shapes other than points. Similarly, Har-Peled [19] proposed an octree-like approximation of the Voronoi tessellation. Birn et al [5] proposed a full hierarchy of Delaunay triangulations for 2D nearest-neighbor lookups. However, the authors state that their approaches are unlikely to work well in 3D and beyond.

This work extends our previous work [11], which describes the hash-implicit octree search. This paper additionally includes

- an approximate nearest-neighbor variant of the method;
- additional theoretical discussions regarding failure cases, search complexity, and extensions to higher dimensions;
- experiments regarding the influence of the different steps;

- comparisons to more related work, including FLANN and EFANNA.

3 Exact search

3.1 Notation and overview

We denote points from the target data set as $\mathbf{x} \in D$ and points of the query set $\mathbf{q} \in Q$. D contains $N = |D|$ points. Given a query point $\mathbf{q}$, the objective is to find a closest point

$$\text{NN}(\mathbf{q}, D) = \arg \min_{\mathbf{x} \in D} |\mathbf{q} - \mathbf{x}|_2. \quad (1)$$

The individual Voronoi cells of the Voronoi diagram of D are denoted $\text{voro}(\mathbf{x})$, which we see as closed set. Table 1 summarizes the notations.

Note that the nearest neighbor of $\mathbf{q}$ in D is not necessarily unique, since multiple points in D can have the same distance to $\mathbf{q}$. In many practical applications of this method, however, we are mostly interested in a single nearest neighbor. Additionally, considering rounding errors and floating point accuracy, it is highly unlikely for a measured point to actually have multiple nearest neighbors in practice. We will therefore talk of *the* nearest neighbor, even though this is technically incorrect.

The proposed method requires a pre-processing step where the voxel hash structure for the data set D is created. Once this data structure is precomputed, it remains unchanged and can be used for subsequent queries. The creation of the data structure is done in three steps: The computation of the Voronoi cells for the data set D , the creation

Table 1 Summary of notations used in this work

Notation	Description
$D \subset \mathbb{R}^3$	Target point cloud
$N = D $	Size of target point cloud
$\text{NN}(\mathbf{q}, D)$	Nearest neighbor of $\mathbf{q} \in \mathbb{R}^3$ in D
$\text{ANN}(\mathbf{q}, D)$	Approximate nearest neighbor
$\text{voro}(\mathbf{x})$	Voronoi cell of $\mathbf{x}$ in D : $\text{voro}(\mathbf{x}) = \{\mathbf{y} \in \mathbb{R}^3 : \forall \hat{\mathbf{x}} \in D : \mathbf{y} - \mathbf{x} \leq \mathbf{y} - \hat{\mathbf{x}} \}$.
$v \subset \mathbb{R}^3$	A voxel in 3D space
$l(v)$	Voxel level
$L(D, v)$	Set of all data points whose Voronoi cell intersects with voxel v
$M_{\max}$	Maximum list length
$L_{\max}$	Maximum voxel level
$\text{vol}(V)$	Volume of $V \subset \mathbb{R}^3$
$\text{diam}(V)$	Maximum diameter of $V \subset \mathbb{R}^3$: $\text{diam}(V) = \max\{ \mathbf{x} - \mathbf{y} : \mathbf{x}, \mathbf{y} \in V\}$

Fig. 1 Toy examples in 2D of the creation of the hierarchical voxel structure. For the data point set (left), the Voronoi cells are computed (center). Starting with the root voxel that encloses all points, voxels are recursively split if the number of intersecting Voronoi cells exceeds $M_{\max}$. In this example, the root voxel is split until each voxel intersects at most $M_{\max} = 5$ Voronoi cells (right)

of the octree and the transformation of the octree into a hash table.

3.2 Octree creation

Using Voronoi cells is a natural way to approach the nearest neighbor problem. A query point $\mathbf{q}$ is always contained within the Voronoi cell of its closest point, i.e.,

$$\mathbf{q} \in \text{voro}(\text{NN}(\mathbf{q}, D)). \quad (2)$$

Thus, finding a Voronoi cell that contains $\mathbf{q}$ is equivalent to finding $\text{NN}(\mathbf{q}, D)$. However, the irregular and data-dependent structure of the Voronoi tessellation does not allow a direct lookup. To overcome this, we use an octree to create a more regular structure on top of the Voronoi diagram, which allows to find the corresponding Voronoi cell quickly.

After computing the Voronoi cells for the data set D , an octree is created, whose root voxel contains the expected query range. Note that the root voxel can be several thousand times larger than the extend of the data set without significant performance implications.

Contrary to traditional octrees, where voxels are split based on the number of contained data points, we split each voxel based on the number of intersecting Voronoi cells: Each voxel that intersects more than $M_{\max}$ Voronoi cells is split into eight sub-voxels, which are processed recursively. Figure 1 shows a 2D example of this splitting. The set of data points whose Voronoi cells intersect a voxel v is denoted

$$L(D, v) = \{\mathbf{x} \in D : \text{voro}(\mathbf{x}) \cap v \neq \emptyset\}. \quad (3)$$

This splitting criterion allows a constant processing time during the query phase: For any query point $\mathbf{q}$ contained in a leaf voxel v_{leaf} , the Voronoi cell of the closest point $\text{NN}(\mathbf{q}, D)$ must intersect v_{leaf} . Therefore, once the leaf node voxel that contains $\mathbf{q}$ is found, at most M_{max} data points must be searched for the closest point. The given splitting criterion thus removes the requirement for backtracking.

The cost for this is a deeper tree, since a voxel typically intersects more Voronoi cells than it contains data points. The irregularity of the Voronoi tessellation and possible degenerated cases, as discussed below, make it difficult to give theoretical bounds on the depth of the octree. However, experimental validation shows that the number of created voxels scales linearly with the number of data points $|D|$ (see Fig. 6, Left).

3.3 Hash table

The result of the recursive subdivision is an octree, as depicted in Fig. 1. To find the closest point of a given query point $\mathbf{q}$, two steps are required: find the leaf voxel $v_{\text{leaf}}(\mathbf{q})$ that contains $\mathbf{q}$ and search all points in $L(D, v_{\text{leaf}}(\mathbf{q}))$ for the closest point of $\mathbf{q}$. The computation costs for finding the leaf node in an octree with average depth depth are on average $\mathcal{O}(\text{depth}) \approx \mathcal{O}(\log(|D|))$ when letting $\mathbf{q}$ descend the tree in a conventional way. We propose to use the regularity of the octree to reduce these costs to $\mathcal{O}(\log(\text{depth})) \approx \mathcal{O}(\log(\log(|D|)))$. For this, all voxels of the octree are stored in a hash table that is indexed by the voxel's level $l(v)$ and the voxel's integer-valued coordinates $\text{idx}(v) \in \mathbb{Z}^3$ (Fig. 2).

The leaf voxel $v_{\text{leaf}}(\mathbf{q})$ is then found by bisecting its level. The minimum and maximum voxel level is initialized as $l_{\min} = 1$ and $l_{\max} = \text{depth}$. The existence of the voxel with the center level $l_c = \lfloor (l_{\min} + l_{\max})/2 \rfloor$ is tested using the hash table. If the voxel exists, the search proceeds with the inter-

Fig. 2 The hash table stores all voxels v , which are indexed through their level l and their index idx that contains the integer-valued coordinates of the voxel. The hash table allows to check for the existence of a voxel in constant time

Fig. 3 Toy example in 2D of how to find the leaf voxel by bisecting its level. Finding the leaf node by letting the query point descend the tree would require $\mathcal{O}(\text{depth})$ operations on average, for an octree of depth depth (green path). Instead, the leaf node is found through bisection of its level. In each step, the hash table is used to check for the presence of the corresponding voxel. The search starts with the center level $l_1 = 5$ and, since the voxel exists, proceeds with $l_2 = 7$. Since the voxel at level l_2 does not exist, level $l_3 = 6$ is checked and the leaf node is found. This requires only three lookups, compared to 6 lookups in the green path (color figure online)

val $[l_c, l_{\max}]$. Otherwise, it proceeds to search the interval $[l_{\min}, l_c - 1]$. The search continues until the interval contains only one level, which is the level of the leaf voxel $v_{\text{leaf}}(\mathbf{q})$. Figure 3 illustrates this bisection on a toy example.

Note that in our experiments, tree depths were in the order of 20–40 such that the expected speedup over the traditional method was around 5. Additionally, each voxel in the hash table contains the minimum and maximum depth of its subtree to speedup the bisection. Additionally, the lists $L(D, v)$ are stored only for the leaf nodes. The primary cost during the bisection are cache misses when accessing the hash table. Therefore, an inlined hash table is used to reduce the average amount of cache misses.

3.4 Runtime complexity

The runtime complexity of the different steps for finding a nearest neighbor $\text{NN}(\mathbf{q}, D)$ in a set of $N = |D|$ points can be estimated as follows:

- Empirically, the depth of the octree is $\text{depth} = \mathcal{O}(N)$ (see Sect. 5.1)
- Using the bisection search, the leaf voxel v of $\mathbf{q}$ can be found in $\mathcal{O}(\log(\text{depth})) = \mathcal{O}(\log(\log(N)))$
- Since the number of points contained in the leaf voxel is bound by M_{max} , finding the closest point to $\mathbf{q}$ from that list can be done in $\mathcal{O}(1)$.

Fig. 4 Example of a degenerated point set (left) where many Voronoi cells meet at one point (center). In this case, the problem of finding the nearest neighbor is ill-posed for query points close to the center of the circle. To capture such degenerated cases, voxel splitting is stopped after $L_{\max}$ subdivisions (right). See the text for more comments on why such situations are not of practical interest

Thus, $\text{NN}(\mathbf{q}, D)$ can be computed on average in $\mathcal{O}(\log(\log(N)))$, which is almost constant.

3.5 Degenerated cases

For some degenerated cases, the proposed method for splitting voxels based on the number of intersecting Voronoi cells might not terminate. This happens when more than $M_{\max}$ Voronoi cells meet at a single point, as depicted in Fig. 4. To avoid infinite recursion, a limit $L_{\max}$ on the depth of the octree is enforced. In such cases, the query time for points that are within such an unsplit leaf voxel is larger than for other query points.

However, we found that in practice such cases appear only on synthetic datasets. Also, since the corresponding leaf voxels are very small (of size $2^{-L_{\max}}$ times the size of the root voxel), chances of a random query point to be within the corresponding voxel are small. Additionally, note the problem of finding the closest point is ill-posed in situations where many Voronoi cells meet at a single point and the query point is close to that point: small changes in the query point can lead to arbitrary changes of the nearest neighbor.

The degradation in query time can be avoided by limiting the length of $L(D, v)$ of the corresponding leaf voxels. The maximum error made in this case is in bound by the diameter of the voxel of level $L_{\max}$. For example, $L_{\max} = 30$ reduces the error to 2^{-30} times the size of the root voxel, which is already smaller than the accuracy of single-precision floating point numbers.

Summing up, the proposed method degrades only in artificial situations where the problem itself is ill-posed, but the method's performance guarantee can be restored at the cost of an arbitrary small error.

3.6 Generalizations to higher dimensions

The proposed method theoretically can be generalized to dimensions $d > 3$. However, memory and computational

costs would likely render the method practically unusable in higher dimensions. This is due to several reasons:

- The branching factor 2^d of the corresponding hypercube tree leads to exponentially increasing memory and computation requirements, even for approximately constant average tree depths. For example, even a moderate dimension such as $d = 16$ has a branching factor of $2^{16} = 65536$, such that a tree of depth 3 would already have $(2^{16})^3 = 2^{48}$ nodes.
- Voronoi cells in higher dimensions are increasingly difficult to compute. Dwyer [13] showed that the geometric complexity of the Voronoi cells of n points in dimension d is at least¹

$$\mathcal{O}(nd^d) \quad (4)$$

- Due to the *curse of dimensionality*,² the distances between random points in higher dimensions tend to become more similar [2]. As one consequence, the number of Voronoi neighbors of each point increase, up to the point where almost all points are neighbors of each other. As another consequence, nearest-neighbor lookups for a random query point become ill-conditioned in the sense that a random query point will have many neighbors with approximately equal distance. Voxels are therefore likely to have very long lists of possible nearest neighbors, resulting in even deeper voxel trees.

4 Approximate search

4.1 Definition

Approximate nearest-neighbor methods are methods that return only an approximation of the correct nearest neighbor. Approximate methods often are significantly faster or require less memory than exact methods. For example, a simple approximate method is to use a k -d-tree without performing backtracking (see, for example, [22, 23]).

Given a query point $\mathbf{q}$ and a dataset D , we denote $\text{ANN}(\mathbf{q}, D)$ for an approximate nearest neighbor of $\mathbf{q}$ in D . We define the distance to the exact and the approximate nearest neighbor as

¹ Note that Dwyer showed that for a *fixed* dimension, the complexity is linear in the number of points. Refer to equation (3.3) in [13] and the following discussion for the result regarding 4.

² The term goes back to Richard E. Bellmann. It captures the fact that even for moderately larger dimensions, the volume of space increases drastically. This often results in counterintuitive effects if one keeps only 3D spaces in mind.

$$d_E = |\mathbf{q} - \text{NN}(\mathbf{q}, D)| \quad (5)$$

$$d_A = |\mathbf{q} - \text{ANN}(\mathbf{q}, D)| \quad (6)$$

with $d_A \geq d_E$.

4.2 Quality metrics

Several quantitative values can be used to describe the quality of an approximate method. The error probability p_{err} defines the probability for a random query point to not return the exact, but only an approximate nearest neighbor:

$$p_{\text{err}} = P(d_A > d_E) \quad (7)$$

The absolute error is given as

$$E_{\text{abs}} = |d_A - d_E| = d_A - d_E \quad (8)$$

Approximate methods are often classified according to the ε -criterion, which states that

$$d_A \leq (1 + \varepsilon) d_E \quad (9)$$

and thus puts an upper bound on the relative error.

Given some object M with a fixed, known size $\text{diam}(M)$, we will also measure the quality of an approximate nearest neighbor relative to the object's diameter:

$$e = E_{\text{abs}} / \text{diam}(M) = (d_A - d_E) / \text{diam}(M) \quad (10)$$

The proposed voxel hash method can easily be converted into an approximate method. We will combine two techniques that work at different steps of the method: List length limiting and explicit voxel neighborhood.

4.3 List length limiting

A straightforward way of reducing the complexity of both the offline and online phase is to limit the list lengths of each voxel. This is equivalent to storing, for each leaf node, only a subset of the intersecting Voronoi cells. We denote L_A for a subset of the correct list:

$$L_A(D, v) \subset L(D, v) \quad (11)$$

Several possibilities exist how L_A can be selected from L .

- *Minimize error probability:* Given a voxel v , the probability that an intersecting Voronoi cell $\text{voro}(\mathbf{x})$, $\mathbf{x} \in L(D, v)$ contains a query point $\mathbf{q} \in v$ is

$$P(\mathbf{q} \in \text{voro}(\mathbf{x}) \mid \mathbf{q} \in v) = \frac{\text{vol}(\text{voro}(\mathbf{x}) \cap v)}{\text{vol}(v)}. \quad (12)$$

Therefore, if $\mathbf{x}$ is removed from $L(D, v)$, the probability of making an approximation error when querying for $\mathbf{q}$ is $P(\mathbf{q} \in \text{voro}(\mathbf{x}) \mid \mathbf{q} \in v)$. In order to minimize the probability of making an error, the points in $L(D, v)$ can be removed based on the volume $\text{vol}(\text{voro}(\mathbf{x}) \cap v)$ of the intersection, removing cells with smaller intersection volumes first. Since the Voronoi cells are disjoint, the total probability of an approximation error is the sum of 12 over all removed entries.

If the approximation error probability shall be bounded, one can remove points from the lists $L(D, v)$ only until said probability is reached.

- *Minimize maximum absolute error:* The Voronoi cells intersecting a voxel can be removed such that some predefined maximum absolute error is maintained. Given some closed, convex, bounded volume $V \subset \mathbb{R}^3$, we define the maximum distance of a point inside that volume from the volume's boundary,

$$\text{maxdist}(V) = \sup_{\mathbf{v} \in V} \inf_{\mathbf{w} \in \mathbb{R}^3 \setminus V} |\mathbf{v} - \mathbf{w}| \quad (13)$$

If an entry $\mathbf{x} \in L(D, v)$ is removed from $L(D, v)$, the maximum absolute error possible is

$$\text{maxdist}(\text{voro}(\mathbf{x}) \cap v) \quad (14)$$

If multiple entries $\mathbf{x}_1, \mathbf{x}_2, \dots$ are removed, the maximum absolute error is

$$\max E_{\text{abs}} = \text{maxdist}\left(\bigcup_i (\text{voro}(\mathbf{x}_i) \cap v)\right) \quad (15)$$

This formula allows to remove points from $L(D, v)$ while keeping a bound on the maximum absolute error.

- *Greedy element selection:* Both methods above require an explicit computation of the Voronoi cells and their intersection with voxels. While elegant, such computations can be expensive.

A different strategy is to keep only a fixed number of vertices that are closest to the center of the voxel. This strategy is faster, since it does not require explicit computation of the intersection volumes. It is especially efficient in combination with the next step, which avoids constructing Voronoi cells all together.

4.4 Explicit voxel neighborhood

As shown in Sect. 5, using Voronoi cells as described leads to a potentially very time-consuming offline stage. Most of the runtime is spent in the creation of the Voronoi cells, and the intersection between Voronoi cells and voxels.

Fig. 5 2D-illustration of the fast, approximate voxel creation. Instead of computing and intersecting Voronoi cells, each point (black dot) is added to a $n \times n$ -neighborhood (here 3×3) of voxels, on each voxel level

A different approach allows a much faster assignment of points to voxels: instead of intersecting Voronoi cells with voxels, a point $\mathbf{x} \in D$ is added to the list of its neighboring voxels only. Figure 5 illustrates this: The given point is added to all voxels in its 3×3 (or, in 3D, $3 \times 3 \times 3$) neighborhood.

This technique is combined with the list length limiting by retaining only a few or even only one point closest to the voxel's center. The runtime for creating the voxel tree this way is linear in the number of points N and has a significantly smaller constant factor. In particular, no complex creation of Voronoi cells needs to be performed.

Note that both steps modify only the creation of the data structure; the lookup phase stays the same. The following algorithm summarizes the proposed approximate method.

```

Input: Dataset  $D$ 
      Voxel level range  $l_{\min}$  and  $l_{\max}$ 
      Maximum list length  $M_{\max}$ 

for  $\mathbf{x}$  in  $D$  do:
  for  $l$  from  $l_{\min}$  to  $l_{\max}$  do:
     $v \leftarrow$  voxel of level  $l$  containing  $\mathbf{x}$ 
    for  $v'$  in  $3 \times 3 \times 3$  neighborhood of  $v$ :
       $L(v') \leftarrow L(v') \cup \{\mathbf{x}\}$ 

for all voxels  $v$ :
  Sort  $L(v)$  by distance to center of  $v$ 
  Truncate  $L(v)$  at length  $M_{\max}$ 

Output: Set of voxel lists  $L$ 

```

Tree Depth For the exact methods, voxels were split based on the number of intersecting Voronoi cells. This provided a natural way of splitting voxels only where necessary. A downside of the proposed approximate method is that this automatic splitting no longer happens. As consequence, the range of levels must be specified a priori.

In the evaluation, we estimate the sampling density of the target point cloud D as d_{sampling} and use it as a lower bound on the voxel size. This typically leads to tree depths of 10–30.

Additionally, a post-processing step can be used to remove unnecessary voxels: if only a single point is stored for each voxel ($M_{\max} = 1$), and all existing child voxels of some voxel v store the same point, then all those child voxels can be removed without changing the result of the nearest-neighbor lookup. This effectively prunes the voxel tree at uninteresting locations.

5 Experiments

Several experiments were conducted to evaluate the performance of the proposed method in different situations and to compare it to the k -d-tree, the ANN library [22], the FLANN library [23] and the EFANNA method [15] as state-of-the-art methods. Note that the FLANN library returns an approximate nearest neighbor, while ANN was configured such that an exact nearest neighbor was returned. Both the k -d-tree and the voxel hash structure were implemented in C with similar optimization. The creation of the voxel data structure was partly parallelized, queries were not. All times were measured on an Intel Xenon E5-2665 with 2.4 GHz.

5.1 Data structure creation

Although the creation of the proposed data structure is significantly more expensive than the creation of the k -d-tree, the ANN library and the FLANN library, these costs are still within reasonable bounds. They are within the same order of magnitude as for EFANNA. Figure 6, right compares the creation times for different values of $M_{\max}$. The creation of the Voronoi cells is independent of the value of $M_{\max}$ and thus plotted separately.

Figure 6, right shows the number of created voxels. They depend linearly on the number of data points, while the choice of $M_{\max}$ introduces an additional constant factor. This shows empirically what is difficult to find analytically: The octree growth is of the same order as a k -d-tree and requires $\mathcal{O}(N)$ nodes. This leads to an average depth of the octree of $\text{depth} = \mathcal{O}(\log(N))$.

Note that the constant performance of the proposed method for fewer than 10^5 data points is based on our particular implementation, which is optimized for large data sets and requires constant time for the creation of several caches.

5.2 Influence of implicit octree

The proposed method consists of two improvements, tree building based on voronoi intersection and, on top of it, the implicit octree. To evaluate how much the implicit octree

Fig. 6 Construction and memory costs of the proposed data structure for the CLUSTER dataset. Left: the number of created voxels depends linearly on the size of the data cloud. As a rule of thumb, one voxel is created per data point. Right: the creation time of the voxel data structure. The creation of the Voronoi cells is independent of the value of $M_{\max}$ and its creation time is plotted separately. Although the creation of the voxel data structure is significantly slower than for the k -d-tree, the ANN and the FLANN libraries, and asymptotically at the same order

Table 2 Speedup of the implicit, hash-based octree w.r.t. to a traditional octree

Dataset	$ D $	Classic (s)	Implicit (%)	Speedup
ICP matching	990,998	0.34	0.25 s	26
Comparison	990,998	0.29	0.22 s	23
ICP room	260,595	0.07	0.06 s	17

helps in terms of speedup, we evaluated the Voronoi-based octree alone, letting query points descend the tree in a classic approach. The results, shown in Table 2, show that for datasets with around $|D| \approx 10^6$ 3D points, the runtime was reduced by around 25%. For $|D| \approx 2.5 \cdot 10^5$, the speedup was 17%.

This indicates that for larger datasets, and thus deeper octrees, the influence of the implicit octree increases. This is as expected from the theoretical analysis, since the influence of the implicit octree (search time of $\mathcal{O}(\log(\text{depth}))$ instead of $\mathcal{O}(\text{depth})$) becomes more prominent for larger depths.

5.3 Degenerated case

As discussed in Sect. 3.5, there exists degenerated cases where the octree creation based on Voronoi splitting would not terminate. As countermeasure, we used both a maximum tree depth $L_{\max}$ and a maximum list length $M_{\max}$. This was evaluated and compared to other methods on a synthetic dataset that consists of N points distributed equally on a sphere of radius 1. The query point is in the center of the sphere (Fig. 4).

of magnitude as EFANNA, the creation times are still reasonable for offline processing. Note that the constant performance of the proposed method for less than 10^5 data points is based on our particular implementation, which is optimized for large data sets and requires constant time for the creation of several caches. Overall, larger values of $M_{\max}$ lead to faster and less memory consuming data structure creation, at the expense of matching time (see Fig. 9)

As shown in Fig. 7, the non-approximate voxel-based methods have significant construction costs, but almost constant query times that are independent of the number of data points.

5.4 Synthetic datasets

We evaluate the performance on different datasets with different characteristics. Three synthetic datasets were used and are illustrated in Fig. 8. For dataset RANDOM, the points are uniformly distributed in the unit cube $[0, 1]^3$. For CLUSTER, points are distributed using a Gaussian distribution. For SURFACE, points are taken from a 2D manifold and slightly disturbed. For each data set, two query sets with 1,000,000 points each were created. For the first set, points were distributed uniformly within the bounding cube surrounding the data point set. The corresponding times are shown in the center column of Fig. 9. The second query set has the same distribution as the underlying data set, with the corresponding timings shown in the right column of Fig. 9.

The proposed data structure is significantly faster than the simple k -d-tree for all datasets with more than 10^5 points. The ANN library shows similar performance as the proposed method for $M_{\max} = 30$ for the RANDOM and CLUSTER datasets. For the SURFACE dataset, our method clearly outperforms ANN even for smaller point clouds. Note that the SURFACE dataset represents a 2D manifold and thus shows the behavior for ICP and other surface-based applications. Overall, compared to the other methods, the performance of the proposed method is less dependent on the distribution of

Fig. 7 Construction (left) and lookup times (right) for a degenerated dataset (as shown in Fig. 4). While the voxel-based methods take significantly longer to be created, their lookup times remains almost constant, even for larger datasets. Note that the ANN library switches to a different internal implementation at around 10^2 points, leading to a drop in construction time. Also note that the EFANNA implementation

failed to give results for less than 1000 data points, and that the voxel methods were evaluated only up to 10^5 data points due to excessive construction times. Finally, some methods (ANN, k -d-tree, and FLANN) are linear in the number of data points. This indicates that they fall back to a brute-force search in the degenerated case, for example, due to backtracking of the k -d-tree

Fig. 8 Datasets used for the synthetic evaluations. The datasets show different distributions of the target points in D : a uniform distribution in a unit cube (left), a Gaussian distribution forming a cluster of points

(center), and points sampled from a 2D manifold (right). Usually, distributions such as the SURFACE dataset are more relevant, since datasets typically contain points from the surface of 3D objects

data and query points. This advantage allows our method to be used in real-time environments.

5.5 Real-world datasets

Next, real-world examples were used for evaluating the performance of the proposed method. Three datasets were collected and evaluated.

ICP Matching: Several instances of an industrial object were detected in a scene acquired with a multi-camera stereo setup. The original scene and the matches are shown in Fig. 10. We found approximate positions of the target object using the method of [12] and subsequently used ICP for each match for a precise alignment. The nearest-neighbor lookups during ICP were logged and later evaluated with the available methods.

Comparison: We used the proposed method to find surface defects of the objects detected in the previous dataset.

For this, the distances of the scene points to the closest found model were computed. The distances are visualized in Fig. 10, right and show a systematic error in the modeling of the object.

ICP Room: Finally, we used a Kinect sensor to acquire two slightly rotated scans of an office room and aligned both scans using ICP. Again, all nearest-neighbor lookups were logged for later evaluation.

The sizes of the corresponding data clouds and the lookup times are shown in Table 3. For all three datasets, the proposed method significantly outperforms both our k -d-tree implementation and the ANN library by up to one order of magnitude.

5.6 Approximate method

We conducted several experiments to compare the proposed approach for turning the exact voxel hash method into an

Fig. 9 Query time per query point for different synthetic datasets and methods. Each row represents a different dataset. From top to bottom: RANDOM, CLUSTER, and SURFACE dataset. The x -axis shows the number of data points, i.e., $|D|$, the y -axis shows the average query time per query point. For the center column, query points were randomly selected from the bounding box surrounding the data. For the right column, query points were taken from the same distribution as

the data points. Compared to other methods, the query time for the proposed method depends little on the number of data points and is almost independent of the distribution of the data and query points. It is especially advantageous for very large datasets as well as for datasets representing 2D manifolds. The approximate significantly outperforms all other methods for all datasets

approximate method (see Sect. 4). We varied two parameters of the approximate nearest-neighbor structure: The number of voxels in the explicit voxel neighborhood, and the limit on the list length, $L(D, v)$. We allow a neighborhood radius of 1 (using a $3 \times 3 \times 3$ neighborhood of voxels on each voxel level) and 2 ($5 \times 5 \times 5$ neighborhood). We found that larger values have little benefit regarding accuracy but high computational costs. For the list lengths, we evaluated with limits of 1, 5 and 10. We denote the approximate methods with, for example, 2–5 for a voxel neighborhood of 2 and a list length limit of 5.

Table 4 compares the different exact and approximate methods regarding data structure creation time, nearest-neighbor lookup time and approximation errors. Figure 9 also includes the timings for the approximate method. In terms of nearest-neighbor lookup times, the proposed approximate method outperforms all other evaluated methods, sometimes by several orders of magnitude. It is the fastest method, we know for comparable error rates, and lookup times scale extremely well with the size of the dataset.

Fig. 10 Example applications for the proposed method. A 3D scan of the scene was acquired using a multi-camera stereo setup and approximate poses of the pipe joint were found using the method of [12]. Left: the poses were refined using ICP. The corresponding nearest-neighbor lookups were logged and used for the evaluation shown in Table 3.

Right: for each scene point close to one of the detected objects, the distance to the object is computed and visualized. This allows the detection of defects on the surface of the objects. The lookups were again logged and used for the performance evaluation in Table 3

Table 3 Performance in the real-world scenarios

Dataset	$ D $	$ Q $	Voxel Hash, $M_{\max} =$			k -d-tree (s)	ANN (s)	FLANN (s)	EFANNA (s)
			30 (s)	60 (s)	90 (s)				
ICP matching	990,998	1,685,639	0.26	0.26	0.28	12.19	22.0	93.6	100.1
Comparison	990,998	2,633,591	0.22	0.25	0.30	10.62	232.1	146.3	130.3
ICP room	260,595	916,873	0.063	0.075	0.078	0.97	2.5	13.0	20.6

$|D|$ is the number of data points, $|Q|$ the number of query points. The proposed voxel hash structure is up to one order of magnitude faster than the compared methods, even for large values of $M_{\max}$

Table 4 Performance of exact and approximate methods on the real-world dataset *ICP Matching*

Method	$ D = 990,998$					$ D = 13,333$				
	t_{create} (s)	t_{lookup} (s)	Correct (%)	e_{mean}	e_{median}	t_{create}	t_{lookup} (s)	Correct (%)	e_{mean}	e_{median}
k -d-tree	0.16	12.19	100	0	0	1.3 ms	2.1	100	0	0
ANN	0.57	23	100	0	0	17 ms	26	100	0	0
FLANN	0.37	93	12	2.4%	0.73%	6.1 ms	2	20	2.7%	0.7%
EFANNA	225	97.6	89.2	0.03%	0	9.22 s	35.2	99.8	0	0
$M_{\max} = 30$	9400	0.24	100	0	0	170 s	0.18	100	0	0
$M_{\max} = 60$	5745	0.21	100	0	0	91 s	0.14	100	0	0
$M_{\max} = 90$	5405	0.20	100	0	0	68 s	0.13	100	0	0
Approx. 1–1	14	0.13	20	1.61%	0.60%	0.2 s	0.13	23	1.61%	0.58%
Approx. 1–5	28	0.17	34	1.34%	0.35%	0.4 s	0.15	39	1.26%	0.27%
Approx. 2–1	56	0.15	37	0.54%	0.20%	0.8 s	0.14	38	0.54%	0.19%
Approx. 2–5	108	0.19	54	0.39%	0	1.3 s	0.18	61	0.35%	0
Approx. 2–10	132	0.20	60	0.35%	0	1.5 s	0.20	69	0.29%	0

Two scenarios are tested: one with a very large data cloud D of approx. one million points (left), one with a much smaller cloud of approx. ten thousand points (right). t_{create} is the time required for the creation of the data structure, t_{lookup} is the time needed to perform around 1.6 million nearest-neighbor lookups. *Correct* is the ratio of points for which the returned nearest neighbor is correct and not just an approximation. Note that the approximate ANN library returned only correct results on this dataset. e_{mean} and e_{median} are the errors of the approximate nearest-neighbor search, relative to the diameter of the target object (an error of $e_{\text{mean}} = 1\%$ would indicate that the approximate nearest neighbors have a mean distance of 1% of the object's diameter to the exact nearest neighbor). Note that for the approximate method 1–1 with a total query time of 0.13 s, the query time per point was only 0.23 μ s, around 550 CPU cycles

Regarding construction times, the approximate voxel methods are much faster than the exact voxel methods, though still significantly slower than k -d-trees, ANN, FLANN, and EFANNA.

6 Conclusion

This work proposed and evaluated a novel data structure for nearest-neighbor lookup in 3D, which can easily be extended to 2D. Compared to traditional tree-based methods, backtracking was made redundant by building an octree on top of the Voronoi diagram. In addition, a hash table was used to allow for a fast bisection search of the leaf voxel of a query point, which is faster than letting the query point descend the tree. The proposed method combines the best of tree-based approaches and fixed voxel grids. We also proposed an even faster approximate extension of the method.

The evaluation on synthetic datasets shows that the proposed method is faster than traditional k -d-trees, the ANN library, the FLANN library and the EFANNA method on larger datasets and has a query time that is almost independent of the data and query point distribution. Although the proposed structure takes significantly longer to be created, these times are still within reasonable bounds. The evaluation on real datasets shows that real-world scenarios, such as ICP and surface defect detection, greatly benefit from the performance of the method. The evaluations also showed that the approximate variant of the method can be constructed significantly faster and offers unprecedented nearest-neighbor query times.

The limitations of the method are mostly in the dimension of the data. For more than three dimensions, the construction and storage costs increase more than exponentially, thus requiring additional work to make at least parts of the method available for such data. The method is thus not suitable for online applications, where the data must be processed immediately. In the future, we want to look into extensions to higher dimensions, additional speedups of the construction and online updates, for example, to extend datasets with additional points without completely re-computing the search structure.

Open Access This article is distributed under the terms of the Creative Commons Attribution 4.0 International License (<http://creativecommons.org/licenses/by/4.0/>), which permits unrestricted use, distribution, and reproduction in any medium, provided you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license, and indicate if changes were made.

References

1. Arya, S., Mount, D.M., Netanyahu, N.S., Silverman, R., Wu, A.Y.: An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *JACM* **45**(6), 891–923 (1998). <https://doi.org/10.1145/293347.293348>
2. Bellman, R.E.: *Adaptive Control Processes: A Guided Tour*, vol. 4. Princeton University Press, Princeton (1961)
3. Bentley, J.L.: Multidimensional binary search trees used for associative searching. *CACM* **18**(9), 509–517 (1975). <https://doi.org/10.1145/361002.361007>
4. Besl, P.J., McKay, N.D.: A method for registration of 3-d shapes. *IEEE Trans. Pattern Anal. Mach. Intell.* **14**(2), 239–256 (1992). <https://doi.org/10.1109/34.121791>
5. Birn, M., Holtgrewe, M., Sanders, P., Singler, J.: Simple and fast nearest neighbor search. In: Blleloch, G.E., Halperin, D. (eds.) *Proceedings of the Twelfth Workshop on Algorithm Engineering and Experiments, ALENEX 2010, Austin, Texas, USA, 16 Jan 2010*, pp. 43–54. SIAM (2010). <https://doi.org/10.1137/1.9781611972900.5>
6. Boada, I., Coll, N., Madern, N., Sellarès, J.A.: Approximations of 3d generalized voronoi diagrams. In: (Informal) *Proceedings of the 21st European Workshop on Computational Geometry, Eindhoven, The Netherlands, 9–11 Mar 2005*, pp. 163–166. Technische Universiteit Eindhoven (2005)
7. Boada, I., Coll, N., Madern, N., Sellarès, J.A.: Approximations of 2d and 3d generalized voronoi diagrams. *Int. J. Comput. Math.* **85**(7), 1003–1022 (2008). <https://doi.org/10.1080/00207160701466362>
8. Choi, W., Oh, S.: Fast nearest neighbor search using approximate cached k -d tree. In: *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems, IROS 2012, Vilamoura, Algarve, Portugal, 7–12 Oct 2012*, pp. 4524–4529. IEEE (2012). <https://doi.org/10.1109/IROS.2012.6385837>
9. Cleary, J.G., Wyvill, G.: Analysis of an algorithm for fast ray tracing using uniform space subdivision. *Vis. Comput.* **4**(2), 65–83 (1988). <https://doi.org/10.1007/BF01905559>
10. Delaunay, B.: Sur la sphere vide. a la memoire de george voronoi. *Bulletin de l'Académie des Sciences de l'URSS. Classe des sciences mathématiques et na*, pp. 793–800 (1934)
11. Drost, B., Ilic, S.: A hierarchical voxel hash for fast 3d nearest neighbor lookup. In: Weickert, J., Hein, M., Schiele, B. (eds.) *Pattern Recognition—35th German Conference, GCPR 2013, Saarbrücken, Germany, September 3–6, 2013. Proceedings, Lecture Notes in Computer Science*, vol. 8142, pp. 302–312. Springer (2013). <https://doi.org/10.1007/978-3-642-40602-7>
12. Drost, B., Ulrich, M., Navab, N., Ilic, S.: Model globally, match locally: efficient and robust 3d object recognition. In: *The Twenty-Third IEEE Conference on Computer Vision and Pattern Recognition, CVPR 2010, San Francisco, CA, USA, 13–18 June 2010*, pp. 998–1005. IEEE Computer Society (2010). <https://doi.org/10.1109/CVPR.2010.5540108>
13. Dwyer, R.A.: Higher-dimensional voronoi diagrams in linear expected time. *Discrete Comput. Geom.* **6**, 343–367 (1991). <https://doi.org/10.1007/BF02574694>
14. Elseberg, J., Magnenat, S., Siegwart, R., Nuechter, A.: Comparison of nearest-neighbor-search strategies and implementations for efficient shape registration. *J. Softw. Eng. Robot.* **3**(1), 2–12 (2012)
15. Fu, C., Cai, D.: EFANNA: an extremely fast approximate nearest neighbor search algorithm based on kNN graph. *ArXiv* (2016). <http://arxiv.org/abs/1609.07228>
16. Glassner, A.S.: Space subdivision for fast ray tracing. *IEEE Comput. Graph. Appl.* **4**(10), 15–24 (1984)
17. Greenspan, M.A., Godin, G.: A nearest neighbor method for efficient ICP. In: *3rd International Conference on 3D Digital Imaging and Modeling (3DIM 2001), 28 May–1 June 2001, Quebec City, Canada*, pp. 161–170. IEEE Computer Society (2001). <https://doi.org/10.1109/IM.2001.924426>
18. Greenspan, M.A., Yurick, M.: Approximate K-D tree search for efficient ICP. In: *4th International Conference on 3D Digital Imag-*

- ing and Modeling (3DIM 2003), 6–10 Oct 2003, Banff, Canada, pp. 442–448. IEEE Computer Society (2003). <https://doi.org/10.1109/IM.2003.1240280>
19. Har-Peled, S.: A replacement for voronoi diagrams of near linear size. In: 42nd Annual Symposium on Foundations of Computer Science, FOCS 2001, 14–17 Oct 2001, Las Vegas, Nevada, USA, pp. 94–103. IEEE Computer Society (2001). <https://doi.org/10.1109/SFCS.2001.959884>
 20. Hwang, Y., Han, B., Ahn, H.: A fast nearest neighbor search algorithm by nonlinear embedding. In: 2012 IEEE Conference on Computer Vision and Pattern Recognition, Providence, RI, USA, 16–21 June 2012, pp. 3053–3060. IEEE Computer Society (2012). <https://doi.org/10.1109/CVPR.2012.6248036>
 21. Meagher, D.: Geometric modeling using octree encoding. *Comput. Graph. Image Process.* **19**(2), 129–147 (1982). [https://doi.org/10.1016/0146-664X\(82\)90104-6](https://doi.org/10.1016/0146-664X(82)90104-6)
 22. Mount, D.M., Arya, S.: ANN: a library for approximate nearest neighbor searching. <https://www.cs.umd.edu/~mount/ANN/>
 23. Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. In: Ranchordas, A., Araújo, H. (eds.) *VISAPP 2009—Proceedings of the Fourth International Conference on Computer Vision Theory and Applications*, Lisboa, Portugal, 5–8 Feb 2009, vol. 1, pp. 331–340. INSTICC Press (2009)
 24. Nüchter, A., Lingemann, K., Hertzberg, J.: Cached k-d tree search for ICP algorithms. In: *Sixth International Conference on 3-D Digital Imaging and Modeling, 3DIM 2007*, 21–23 Aug 2007, Montreal, Quebec, Canada, pp. 419–426. IEEE Computer Society (2007). <https://doi.org/10.1109/3DIM.2007.15>
 25. Samet, H.: *Foundations of Multidimensional And Metric Data Structures*. Morgan Kaufmann, Burlington (2006)
 26. Yan, P., Bowyer, K.W.: A fast algorithm for icp-based 3d shape biometrics. *Comput. Vis. Image Underst.* **107**(3), 195–202 (2007). <https://doi.org/10.1016/j.cviu.2006.11.001>

Bertram H. Drost is a senior research engineer at MVTec Software GmbH in Munich, Germany. He received a diploma in mathematics and, in 2016, a PhD in computer science from Technische Universität München (TUM). His current research interests include 2D and 3D object detection, 3D data and point cloud processing, 3D reconstruction and machine learning.

Slobodan Ilic is currently a senior key expert research scientist at Siemens Corporate Technology in Munich, Perlach. He is also a visiting researcher and lecturer at Computer Science Department of TUM and closely works with the CAMP Chair. From 2009 until the end of 2013, he was leading the Computer Vision Group of CAMP at TUM, and before that he was a senior researcher at Deutsche Telekom Laboratories in Berlin. In 2005, he obtained his PhD at EPFL in Switzerland under supervision of Pascal Fua. His research interests include 3D reconstruction, deformable surface modelling and tracking, real-time object detection and tracking, human pose estimation and semantic segmentation.