Machine Vision and Applications (2022) 33:31
https://doi.org/10.1007/s00138-022-01279-w

ORIGINAL PAPER

Delaunay walk for fast nearest neighbor: accelerating correspondence matching for ICP

James D. Anderson[1] · Ryan M. Raettig[2] · Josh Larson[2] · Scott L. Nykl[2] · Clark N. Taylor[2] · Thomas Wischgoll[1]

Received: 14 April 2021 / Revised: 20 October 2021 / Accepted: 16 December 2021 / Published online: 15 February 2022

Abstract
Point set registration algorithms such as Iterative Closest Point (ICP) are commonly utilized in time-constrained environments like robotics. Finding the nearest neighbor of a point in a *reference* 3D point set is a common operation in ICP and frequently consumes at least 90% of the computation time. We introduce a novel approach to performing the distance-based nearest neighbor step based on Delaunay triangulation. This greedy algorithm finds the nearest neighbor of a query point by traversing the edges of the Delaunay triangulation created from a *reference* 3D point set. Our work integrates the Delaunay traversal into the correspondences search of ICP and exploits the iterative aspect of ICP by caching previous correspondences to expedite each iteration. An algorithmic analysis and comparison is conducted showing an order of magnitude speedup for both serial and vector processor implementation.

Keywords Nearest neighbor search · Delaunay triangulation · 3d point cloud processing · Point registration · Iterative Closest Point

1 Introduction

Point set registration presents a challenging problem for numerous applications including computer vision, pattern recognition, robotics, and image processing [42]. Registration finds a rigid transformation between two or more data sets—providing a rotation and translation that minimizes the pairwise Euclidean difference between corresponding pairs of matched points. These point clouds can be known a priori, generated from images, light detection and ranging (LIDAR), or other sensors.

A common point registration method, Iterative Closest Point (ICP), typically must execute in near real-time. In self-driving cars, on-board computers utilize the transformation found by ICP to detect objects within the car's local environment. The system may determine an adjacent vehicle is veering into its lane, creating a potential hazard and requiring immediate action. Thus, these applications require point-set registration processes be completed in a timely fashion. After investigating the steps of ICP, when utilizing a k-d tree, the nearest neighbor step consumes the majority of the time, as shown in Fig. 2. Our work has shown a nearly 90% speedup when compared to the traditional k-d tree as seen in Fig. 1. For this reason, our research focuses on the Euclidean distance-based pairwise nearest neighbor matching employed by point set registration methods such as Besl's ICP algorithm [10]. We invite the reader to watch a video overview of the work detailed in this paper at [5].

Nearest neighbor matching finds a candidate point in a *reference* point cloud that is nearest to a corresponding point in the *sensed* point cloud [29]. Doing this for each point in the *sensed* point cloud creates a mapping that associates each point in the *sensed* cloud to a point in the *reference* cloud. Naïvely, if both point sets are size n, nearest neighbor matching is an $\mathcal{O}(n^2)$ algorithm; therefore, accelerating an algorithm such as ICP requires minimizing the nearest neighbor matching time.

This research focuses on accelerating Euclidean distance-based pairwise point matching and is applicable to all registration algorithms utilizing nearest neighbor pairs. This paper makes the following contributions:

1. A novel, Delaunay-based, embarrassingly parallel, nearest neighbor 3D matching algorithm applicable to numerous applications requiring distance-based correspon-

✉ James D. Anderson
anderson.10@wright.edu

[1] Wright State University, Dayton, OH, USA

[2] Air Force Institute of Technology, Dayton, OH, USA

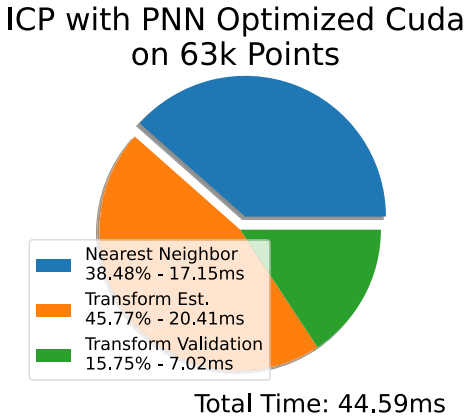


Fig. 1 Breakdown of ICP step times utilizing Previous Nearest Neighbor Delaunay Traversal

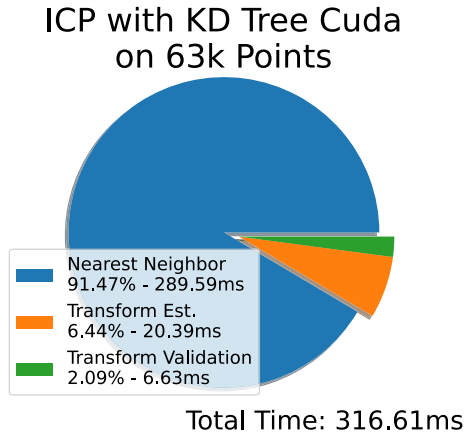


Fig. 2 Breakdown of ICP step times utilizing k-d tree Nearest Neighbor

dences including rigid and non-rigid point registration, pattern recognition and computer vision.
2. An algorithmic analysis of this Delaunay-based correspondence search.
3. ICP-based timing results showing reduced convergence time across common data sets compared to current methods.
4. A stereo vision-based application employing our algorithm in ICP.

The paper is arranged as follows. Section 2 presents related work. Section 3 introduces the *Delaunay Traversal* nearest neighbor algorithm. Section 4 describes variants of our proposed *Delaunay Walk*. Section 5 presents the experimental methodology and results. Sections 6 and 7 conclude the paper and discuss future work.

2 Related work

2.1 Point set registration

Paul Besl's ICP algorithm [10] is a ubiquitous point set registration process which depends on pairwise Euclidean distance correspondence. From the pair-wise correspondences, ICP registers two point clouds by computing a rotation $\mathbf{R}$ and translation $\mathbf{t}$. This is a rigid transformation, which minimizes the mean square point matching error (MSPME) between the two point clouds. This error is the average Euclidean distance between the point correspondences after the transformation is applied.

Consider 3D point sets $X = \{\mathbf{x_i}|\mathbf{x_i} \in R^3\}_{i=1}^{m}$ and $P = \{\mathbf{p_i}|\mathbf{p_i} \in R^3\}_{i=1}^{n}$, where $n = |X|$ and $m = |P|$. Additionally, each point $\mathbf{p_i}$ corresponds to point $\mathbf{x_i}$ with the same index. ICP executes to fit the source P onto the target X. From [42], ICP can be expressed as:

$$\arg\min_{\mathbf{R},\mathbf{t}} \left\{ \frac{1}{m} \sum_{i=1}^{m} |\mathbf{x_i} - (\mathbf{R}\mathbf{p_i} + \mathbf{t})| \right\} \tag{1}$$

ICP is broken into 3 major steps:

1. nearest neighbor search
2. transformation estimation
3. transformation validation

These three steps are repeated until either the error falls below a set threshold or the error between iterations is negligible.

Several variants exist for each of these steps to improve accuracy and speed, particularly transformation estimation. Where Besl utilized point-to-point correspondences, Chen modified the algorithm by generating point normals for the target 3D point set and matching the query point to the plane defined by these normals, typically cited as *point-to-plane* [13]. Since this method registers source points to the area around the target point instead of the target point itself, the algorithm's sensitivity to noise can be reduced [8]. Point-to-plane's transformation calculation takes longer per iteration, but the entire algorithm typically converges in fewer iterations and results in a similar timing to point-to-point [25]. Extending point-to-plane, generalized ICP describes a plane-to-plane method [38]. In plane-to-plane, normal vectors are calculated on both target and source point clouds. This method can be shown to be more robust; however, since surface normals need to be calculated, plane-to-plane is typically not utilized for real-time ICP applications.

ICP handles rigid point registration with little to no scaling or shearing. Other non-rigid point registrations exist to determine an affine transformation when scaling or shearing are present. Several of these approaches utilize distance-based correspondences. For example, coherent point drift (CPD) [31] utilizes the centroids from Gaussian mixture model (GMM) of the source dataset and find an affine transformation alining those centroids to the expected data. The Delaunay-based nearest neighbor algorithm presented in this

paper can be utilized in non-rigid registrations so long as the correspondences are found with a distance-based metric.

2.2 Nearest neighbor

Nearest neighbor applications span numerous fields, including pattern recognition [14], machine learning [40], and robotics [35]. Some of these require an ordered list of similar data, while others only require a single closest point. Depending on the metric used for similarity, the closest point may or may not be unique. In pairwise nearest neighbor, some distance metric such as Euclidean distance, Manhattan distance, squared distance, or other calculation is typically utilized depending on the application. This research employs pairwise Euclidean distance nearest neighbor matching.

Brute force is the generic method for determining nearest neighbors. Again, consider 3D point sets X and P. The nearest neighbor of $\mathbf{p_0} \in P$ is

$$\arg\min_{\mathbf{x}\in X} |\mathbf{p_0} - \mathbf{x}_i| \tag{2}$$

To match all points P to X, Eq. 2 must be evaluated for every point inside P. Again taking $n = ||X||$ and $m = ||P||$, a brute force approach results in a time complexity of $\mathcal{O}(n)$ to match a single point and $\mathcal{O}(\mathrm{nm})$ to match all points.

In order to reduce the time complexity for a nearest neighbor search, applications typically utilize a space-partitioning structure such as a k-d tree. Introduced by Bentley [9], a k-d tree is a space partitioning k-dimensional binary tree that organizes a data set by dividing each level by a hyperplane created from the median of the given dimension. A k-d tree is built by splitting space into a set of alternating hyperplanes placed at the median of a first dimension. The points are split along this partition and then recursively divided based on the next dimension's hyperplane. To query the k-d tree for the nearest neighbor to a given point, the algorithm begins at the root node and traverses down the tree, choosing which child node to visit based on which side of the hyperplane the query point exists [17]. After reaching a leaf, the algorithm back-traces up the tree checking the distance to each hyperplane to determine if the adjacent volume needs to be checked for a closer point. Since a binary tree has $\log_2(n)$ levels, the average query time is proportional to $\mathcal{O}(\log(n))$ with a worst-case of having to search the entire tree being $\mathcal{O}(n)$. However, when numerous queries to the dataset are required, construction of a k-d tree proves a benefit to the alternative of the brute force approach since not every point needs to be compared to the query points.

Other tree-based data structures, including R-trees [19] and octrees [27], have been developed to accelerate nearest neighbor searches. Both structures partition the space of the data comparable to a k-d tree. For R-trees, rectangles create bounds between clusters of data points with each higher level combining clusters into their minimal spanning rectangle. This approach increases spatial locality of nearby points as rectangles lower in the hierarchy can be stored in nearby memory locations. While the innate structure of the data dictates the regions of an R-tree, an octree partitions the space uniformly. These methods present challenges for beginning at an arbitrary data point since the closet point may not be within the region but in an adjacent one. Searches of tree structures typically begin at the root of the tree. The work presented in this paper allows searches to begin from any arbitrary point within the data set, allowing for extremely fast results when subsequent searches are similar to the previous ones.

One method utilizing octrees described in [16] utilizes previous results to facilitate searches. The tree is constructed on the Voronoi cells computed from the original points rather than the points themselves. Regions of the octree, voxels, contain at most M_{max} intersecting Voronoi cells. These voxels are then accessed through a hash table where each entry is indexed through its level in the tree. The search may need to backtrack up the tree to search adjacent regions. Since the structure of the work presented in this paper does not split the data into regions, utilizing a result for subsequent searches only requires searching the previous result's Delaunay neighbors.

With respect to point registration, methods in [13,38] focus on the accuracy of the second step of Besl's ICP. However, this step of the algorithm executes relatively quickly when compared to the nearest neighbor step. Caching the previous ICP iteration's nearest neighbor correspondences is an approach to accelerate the nearest neighbor phase. A cached k-d tree is presented in [32], where a pointer to the node within the tree is utilized as the starting node for subsequent iterations. Similarly, in [18], if the previous correspondence meets certain geometric constraints, it is used to expedite the nearest neighbor search. This research uses the previous ICP iteration's nearest neighbor correspondence to accelerate a nearest neighbor search using the Delaunay triangulation.

While some applications require an exact nearest neighbor, others only necessitate an approximate match. Typically, an approximate neighbor search returns a result faster than an exact search. A straightforward approach utilizing a k-d tree is to simply traverse from the root to the leaf node containing the query point [18]. Additionally, this method lessens the space complexity required for storing the data structure, as only the leaf nodes and median values for each hyperplane axis need to be stored. In [28], a method is presented that can configure itself to the desired degree of accuracy required for the application, allowing the user to balance between precision and speed. In [34], the k-nearest approximate neighbors are calculated in constant time by utilizing locally sensitive hashing to partition datasets into clusters. While some meth-

ods employ an approximate neighbor to accelerate ICP, as shown in Marden [26], the work in this paper only utilizes approximate neighbors as a means to accelerate the exact neighbor search. Thus, the traditional ICP algorithm from Besl can be utilized to verify correctness.

2.3 Delaunay triangulation

In 2D, a Delaunay triangulation of a point set is a set of triangles where the vertices are members of the point set and no points are located within the circumcircles created by the vertices of each triangle. When extended to 3D, the triangle simplex generalizes to a tetrahedron simplex, and the circumcircles become circum-spheres. Importantly, the condition of no points being located in each circum-sphere still holds. The concept can be extended to d-dimensions [15]; however, the work presented here focuses on three-dimensions for 3D sensing-based applications. Figure 3 depicts the full Delaunay triangulation of a 3D point set. Figure 4 visualizes a closer view of all the edges connected to a single point (shown in pink) connecting to the neighbor points (shown in blue).

Delaunay triangulations have many applications including path planning [6,23,39], surface reconstruction [4,12,21], and many others. Mulchrone [30] uses Delaunay triangulation to generate nearest neighbors within a data set with the goal of calculating strain. Mulchrone subsequently removes the edges that form the convex hull; the remaining edges connect nearest neighbors. In contrast to Mulchrone, the work presented in this paper utilizes all edges of a Delaunay triangulation. Keeping all edges allows the neighbor search to quickly traverse the data set, whereas removing edges from the convex hull limits the distance between vertices within the graph. Similarly, in [36] Delaunay triangulations are utilized to cluster points by removing undesired and redundant edges. While clustering may be utilized in nearest neighbor searches to determine the starting vertex of a traversal, the work in this paper utilizes other methods for determining starting nodes as presented in Sect. 4. These examples show Delaunay triangulation can be used to efficiently assign a variety of point correspondences.

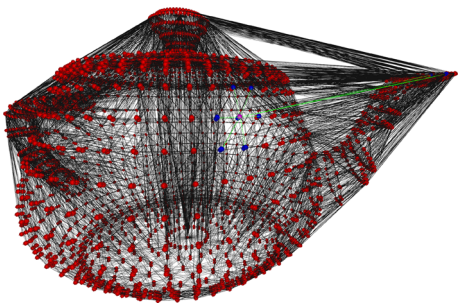


Fig. 3 An illustration of a Delaunay triangulation of a teapot 3D point set

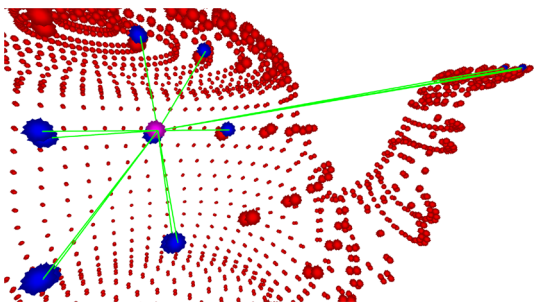


Fig. 4 A close-up view of Delaunay edges of point (pink) to neighbor points (blue)

The creation of the Delaunay triangulation structure can be completed offline. Lee [22] describes an iterative, divide-and-conquer algorithm to create the Delaunay triangulation structure. In this paper, we will be using the Open3D library [41] to create the Delaunay triangulation offline. Open3D uses the Qhull library [7] to compute the convex hull and stores the 3D triangulation as a series of 3D simplexes, or tetrahedrons.

While these methods identify neighbors within a data set, the Full Delaunay Hierarchies (FDH) [11] algorithm utilizes a Delaunay traversal to determine the nearest neighbor of a source point to a target dataset. The traversal in FDH only moves from vertices of lower index to higher index, disallowing traversal in both directions. This detail increases the complexity of implementing the algorithm. Additionally, the authors state their method cannot directly extend to greater than two dimensions. In contrast, this research focuses on traversal in both directions with three-dimensional point clouds.

3 Delaunay traversal

This section presents the novel pairwise Euclidean distance nearest neighbor matching algorithm, which utilizes the Delaunay triangulation of the target 3D point set. Once the Delaunay triangulation for a *reference* point set has been generated, the nearest neighbor of a query point is found by traversing the edges of the Delaunay graph. The algorithm utilizes a greedy approach where at each node of the Delaunay graph, if a connected node is closer than the current visiting node to the query point, the algorithm proceeds to check the nodes connected to the new current visiting mode. This process continues until there are no closer nodes to the query point connected to the current visiting node.

3.1 Notation

Given a point set $X = \{\mathbf{x}_1, \ldots, \mathbf{x}_n\}$ of n points of d dimensions, the nearest neighbor of a point $\mathbf{p} \in P$ is

$$NN(\mathbf{p}, X) = \arg\min_{\mathbf{x} \in X} |\mathbf{p} - \mathbf{x}_i|^2 \tag{3}$$

Squared distance is utilized to avoid a square root operation. Because multiple points within X may be equidistant to $\mathbf{p}$, the result of the nearest neighbor function might not be unique.

A Delaunay triangulation of a point set is a graph $G = (1..k, E)$ consisting of edges from each point $\mathbf{x}_i$ to the Delaunay neighbors $\mathbf{x}_{j \neq i}$. To find the nearest neighbor of a source point within the target point set utilizing the Delaunay triangulation, the algorithm can start at an arbitrary start point, $\mathbf{x}_0$.

To store the full Delaunay triangulation, for each point $\mathbf{x}_i$ in the data set there is a corresponding list of values:

$$\hat{e}_{i,j} = \frac{\mathbf{x}_j - \mathbf{x}_i}{|\mathbf{x}_j - \mathbf{x}_i|} \tag{4}$$

$$m_{i,j} = \frac{1}{2}|\mathbf{x}_j - \mathbf{x}_i| \tag{5}$$

$$j \tag{6}$$

In Eq. 4, $\hat{e}_{i,j}$ is the normalized Delaunay edge from point $\mathbf{x}_i$ to one of its Delaunay neighbors $\mathbf{x}_j$. In Eq. 5, $m_{i,j}$ is one half the distance, or the midpoint, between $\mathbf{x}_i$ and $\mathbf{x}_j$. Lastly, in Eq. 6, j is the reference index where point $\mathbf{x}_j$ is stored in the data set.

3.2 Delaunay creation

The Delaunay triangulation is pre-calculated offline when performing a pairwise Euclidean distance nearest neighbor matching with a known 3D point set. This step is done utilizing the Open3D library [41] discussed in Sect. 2.3. In order to prepare the Delaunay graph for the nearest neighbor traversal, the edges of each tetrahedron are extracted and grouped according to connections when visiting a point. Thus, each edge is represented twice. Although this factor increases required memory space, the graph becomes bidirectional and allows for an arbitrary start-point.

3.3 Traversal

Given an arbitrary query point $\mathbf{p}$, we may find $\mathbf{p}$'s nearest neighbor using a series of traversals, or *walks*, along the Delaunay graph. This sequence of walks is taken from a point $\mathbf{x}_0$ to point $\mathbf{x}$, where $\mathbf{x}$ satisfies Eq. 1. Figure 5 depicts an example of what the algorithm calculates at each node of the walk. In Fig. 5, $\mathbf{x}_j = \mathbf{x}$ *is* the nearest neighbor of query point $\mathbf{p}$. The first step in the algorithm is calculating

$$\mathbf{u} = \mathbf{p} - \mathbf{x}_i \tag{7}$$

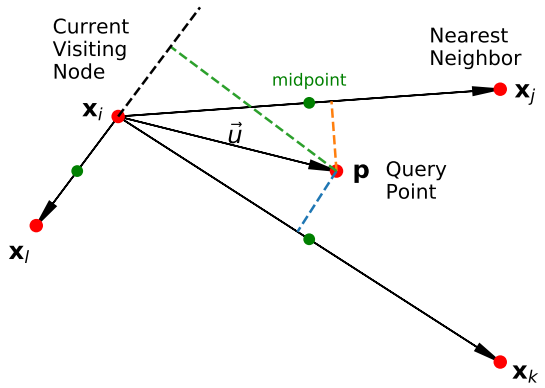


Fig. 5 A Delaunay Walk example. After creating $\mathbf{u}$ from the Current Visiting Node $\mathbf{x}_i$ and Query Point $\mathbf{p}$, the scalar component c is computed by projecting $\mathbf{u}$ onto the edges from $\mathbf{x}_i$ to $\mathbf{x}_j$, $\mathbf{x}_k$, and $\mathbf{x}_l$. The algorithm chooses the next node based on which c is greatest. Once no c is greater than the midpoints, the algorithm stops and returns the current node as the nearest neighbor

where $\mathbf{u}$ is the vector from $\mathbf{x}_i$ to $\mathbf{p}$. Next, the dot product

$$c = \mathbf{u} \cdot \hat{e}_{i,j} \tag{8}$$

between $\mathbf{u}$ and $\hat{e}_{i,j}$ is determined. In Fig. 5, unit vectors from $\mathbf{x}_i$ to $\mathbf{x}_j$, $\mathbf{x}_k$, and $\mathbf{x}_l$, each constitute a specific $\hat{e}_{i,j}$. Since $\hat{e}_{i,j}$ is a unit vector, the dot product c is the scalar component of $\mathbf{u}$ in the direction of $\hat{e}_{i,j}$. The dashed lines in Fig. 5 depict where the projections intersect each Delaunay edge. The scalar c can then be compared to the midpoint $m_{i,j}$ (green points in Fig. 5) from Eq. 5. If the scalar c is greater than $m_{i,j}$, $\mathbf{p}$ is closer to $\mathbf{x}_j$ than $\mathbf{x}_i$. Since the algorithm is seeking the point closest to $\mathbf{p}$, the point $\mathbf{x}_j$ corresponding to the largest c is the next node in the graph to visit. Thus, $\mathbf{x}_i$ is replaced by point $\mathbf{x}_j$, and the algorithm continues. However, if no c is greater than $m_{i,j}$, the algorithm stops and returns the current point $\mathbf{x}_i$ as the nearest neighbor. The pseudocode for the Delaunay traversal is shown in Algorithm 1. Section 4 covers selecting an efficient starting point, $\mathbf{x}_0$, to reduce the number of walks.

3.4 Space and time complexity

Seidel provides a proof illustrating $\mathcal{O}(n^{d/2})$ as the upper-bound for the number of faces within a Delaunay triangulation. With $d = 3$, a 3-dimensional object will have on the order $n^{3/2}$ faces. As such, each point will be a member of $\frac{n^{3/2}}{n} = n^{1/2} = \sqrt{n}$ faces. Thus, each point has on average $\sqrt{n}$ edges. Accordingly, the space complexity for the Delaunay triangulation of a dataset is on the order of $\mathcal{O}(n\sqrt{n})$.

In terms of time complexity to find the nearest neighbor of point $\mathbf{p}$ utilizing the Delaunay traversal algorithm, since each vertex connects to on the order $\sqrt{n}$ other vertices, and all but one of those are rejected at each step of the traversal, by the time $\sqrt{n}$ vertices have been visited, $\sqrt{n}(\sqrt{n} - 1) = n - \sqrt{n}$

Algorithm 1 Delaunay Traversal Pseudocode

Require: $X :=$ Source 3D Point Set
Require: $p :=$ Target Point
Require: $s_i :=$ Index to begin search
Require: $d_{edges} :=$ List of Delaunay edges
Require: $d_{mid} :=$ List of edge midpoints
function DELAUNAYTRAVERSAL(X, p, s_i, d_{edges}, d_{mid})
  $prev_{max} \leftarrow s_i$
  $curr_{max} \leftarrow s_i$
  repeat
    $\mathbf{u} \leftarrow \mathbf{p} - \mathbf{x}_i$
    $foundNN \leftarrow TRUE$
    $c_{max} \leftarrow 0.0$
    for each $\hat{e}, m \in d_{edges}[prev_{max}], d_{mid}[prev_{max}]$ **do**
      $c \leftarrow DotProduct(\mathbf{u}, \hat{e})$
      if $c > m$ **then**
        $foundNN \leftarrow FALSE$
        if $c > c_{max}$ **then**
          $c_{max} \leftarrow c$
          $curr_{max} \leftarrow \hat{e}.index()$
        end if
      end if
    end for
    $prev_{max} \leftarrow curr_{max}$
  until $foundNN == TRUE || prev_{max} == curr_{max}$
  return $X[prev_{max}]$
end function

vertices have been rejected, meaning the entire dataset has been visited. Thus, in the worst case, the algorithm will need to visit $\sqrt{n}$ vertices. However, because with each step of the traversal, the algorithm advances toward the neighbor and away from farther vertices, the worst case will rarely occur. In fact, we present heuristics in Sect. 4 to show how to greatly reduce the number of traversals required.

In contrast, a k-d Tree has $\log_2(n)$ levels and an average query time proportional to $\mathcal{O}(\log(n))$. However, since the search may involve the entire tree, the worst-case becomes $\mathcal{O}(n)$. Similarly, the average and worst-case for an octree are $O(\log(n))$ and $O(\log(n))$, respectively. Additionally, implementing a search of a tree structure typically utilizes recursion. While recursion may execute efficiently on a central processing unit (CPU), a graphics processing unit (GPU) will typically have a smaller stack size. For instance, the depth limit for Compute Unified Device Architecture (CUDA) kernels is 24 [33]. Thus, the search algorithm needs to be implemented iteratively rather than recursively. With respect to memory layout, since the Delaunay traversal structure is vectorized, it lends toward keeping cache coherency. Keeping the data structure in cache facilitates faster searches with less time copying data from main memory.

3.5 Degenerate cases

A few cases exist where the Delaunay triangulation produces suboptimal traversal results. 2D point sets result in edges isolated only to close neighbors. Similar to issues encountered in [16], which utilizes Voronoi cells to determine neighbors, a 2D circle results in each point being connected to only the two closest points. In Fig. 6, each vertex connects only to the two closest vertices. The edges highlighted in green are the edges connected to the pink vertex, which in turn connects to the blue vertices. Additionally, the Delaunay triangulation of a line or plane only connects neighbor points. Looking at Fig. 7, the highlighted edges in green connect the pink point only to the four closest points in blue. The main issue arising from this locality is the number of walks required to find the neighbor increases if the algorithm only can walk to points physically close.

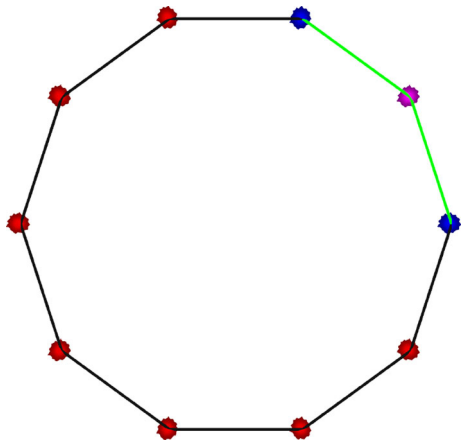


Fig. 6 Delaunay of a 2D circle

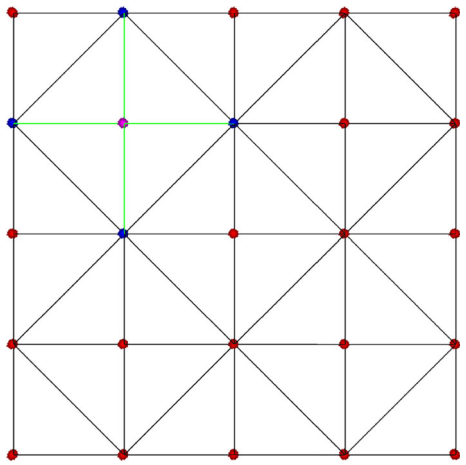


Fig. 7 Delaunay of a 2D plane

To effectively utilize the traversal algorithm on 1D or 2D point sets, virtual points can be introduced during the Delaunay creation. These virtual points exist in a higher dimension, and ideally at distances relatively far from the original points. This virtual bounding box introduces new points and facilitates in the Delaunay generation. After creation, the virtual points can be removed.

4 Delaunay walk variations

In Sect. 3, we covered how the Delaunay traversal algorithm identifies the nearest neighbor to a query point. Because the Delaunay triangulation forms a connected graph, the traversal algorithm can begin at an arbitrary point and be guaranteed to find the nearest point. Consequently, a heuristic or hysteresis can accelerate the correspondence search with the knowledge that there will be a path from the starting node to the nearest neighbor.

Leveraging the iterative aspect of ICP, the nearest neighbor search utilizing the *Delaunay Walk* can be further accelerated. With each iteration of ICP, the source cloud is incrementally transformed to align with the target. After the first few iterations, individual points move relatively small distances from their previous correspondence. Thus, if the correspondences of the previous iteration are cached, the nearest neighbor search can begin at a likely already close point and require fewer traversals to determine the new correspondence. To demonstrate, our research motivated variations focusing on the starting node of the traversal.

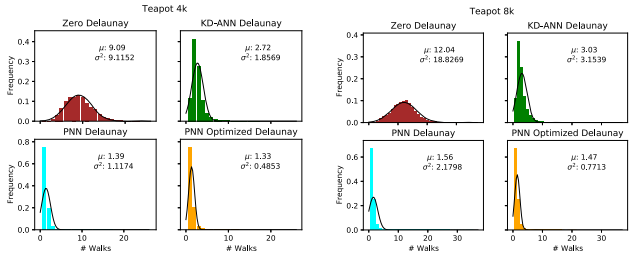
The *Zero Delaunay* walk serves as a benchmark, starting from an arbitrary mode for all traversals. The *k-d Approximate Neighbor (KD-ANN)* Delaunay walk originates from a k-d tree approximate neighbor search. The Previous Nearest Neighbor *(PNN) Delaunay* walk leverages prior nearest neighbor correspondences. Lastly, the *PNN Optimized Delaunay* walk combines both *KD-ANN* and *PNN* for an algorithmically efficient *Delaunay Walk*.

4.1 Zero Delaunay walk

In the *Zero Delaunay* walk variation, the traversal begins from a predetermined node each iteration. The walk originates with no search for a starting node. Each search can begin at the node closest to the center of mass, the most traveled node, or an arbitrary node. This variation requires the least memory and offline preparation; however, it requires the most number of traversals compared to the other variations as seen in Figs. 8 and 9. This variant has the largest mean and variance in the number of walks.

4.2 k-d approximate Delaunay walk

In the *KD-ANN Delaunay* walk variation, the search begins with an approximate nearest neighbor (ANN). The ANN is found by conducting a depth-first search of a k-d tree and returning the leaf node without back-tracing. The ANN returned is usually a close neighbor, but it can return the exact nearest neighbor. Figures 8 and 9 show this effect. By starting the walk from a close neighbor, the number of nodes in the walk decrease substantially. Since this algorithm monotonically converges, a close starting point can significantly shorten the *Delaunay Walk*. On the Dragon(62K) 3D point set, the mean number of walks reduce from around 26 to 5. Because this variation utilizes both a k-d tree and a Delaunay triangulation, it requires more memory than the *Zero Delaunay*.



(a) 4k **(b)** 8k

Fig. 8 Frequency of walks taken for 4k (**a**) and 8k (**b**) Teapot 3D point set

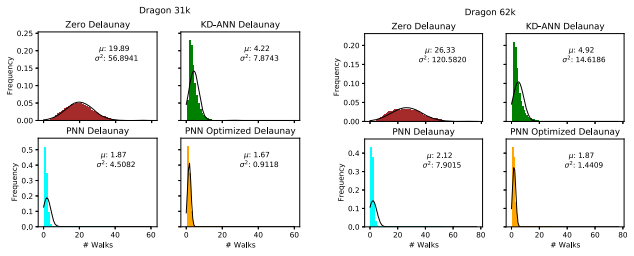
4.3 Previous nearest neighbor Delaunay walk

In the *PNN* Delaunay walk variation, the walk starts from the node found in the previous iteration of ICP. Since aligning the point clouds occurs iteratively, the previous nearest neighbor generally resides close to the current nearest neighbor. No additional computation is required; therefore, the search leverages the previous ICP iteration's work to make the current iteration more efficient. Additionally, increasing the size of the dataset has little effect on the number of walks required. When doubling the points in the Teapot model in Fig. 8a and b, the mean number of walks only increases from 1.39 to 1.56. However, the variances nearly double, indicating there are proportionally more walks of longer length in the larger dataset. Figure 9a and b depicts a result similar to the Teapot data set. The mean number of walks for the Dragon only increases from 1.87 to 2.12, and these values are not dissimilar to the Teapot. While the variances again nearly double, these values are still significantly less than the first two variants.

4.4 PNN optimized Delaunay walk

In the *PNN Optimized Delaunay* walk variation, the *KD-ANN Delaunay* walk is used for the first iteration and the *PNN Delaunay* walk is used in all subsequent iterations. By combining aspects of these variations, this walk variation is the most efficient. Looking at Figs. 8 and 9, this algorithm presents the smallest average number of walks as well as the smallest variance of all four walk variants.

In addition to requiring fewer walks, the two variants utilizing previous neighbor results gain an advantage from increased cache hits. Since the preceding iteration utilized the previous neighbor, it will likely remain in cache memory,



(a) 31k **(b)** 62k

Fig. 9 Frequency of walks taken for 31k (**a**) 62k (**b**) Dragon 3D point set

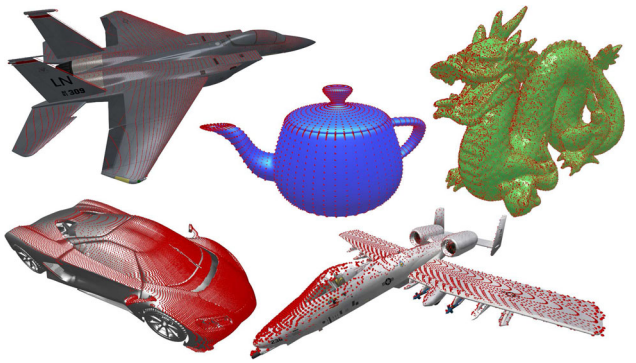
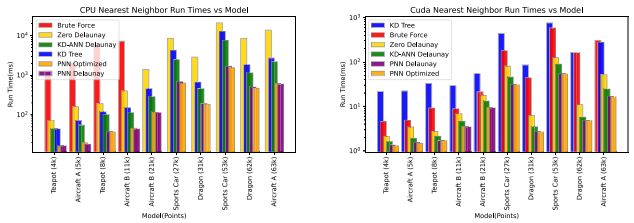


Fig. 10 3D point sets utilized to test the Delaunay Walk nearest neighbor algorithm



(a) CPU **(b)** GPU

Fig. 11 Runtime versus 3D point set on CPU (**a**) and GPU (**b**). Brute force method executed only on models of size <= 11k on CPU due to excessive runtimes

facilitating quick access. Consequently, the Delaunay edges of the previous neighbor likely persist in the cache, allowing the walk to initiate with no access to main memory.

5 Evaluation

5.1 Experiments

To demonstrate the speed and accuracy of the Delaunay Traversal, we conducted various experiments on several 3D point data sets: Teapot, Aircraft A, Aircraft B, Sports Car, and Dragon, all shown in Fig. 10. With these data sets, the runtimes of nearest neighbor methods presented in this paper were tested against the two traditional methods of *Brute Force* and *k-d Tree*. These tests involved matching the point sets to themselves with rotational offsets as well as adding artificial noise. In the second set of experiments, we integrated our ICP with the accelerated Delaunay Traversal into a stereo vision pipeline. The vision pipeline processed images from virtual and real cameras with truth data collected a motion capture system.

All algorithms were implemented in both C++ and CUDA to demonstrate execution on a serial and vector processor. The CPU utilized for these experiments is the AMD Ryzen Threadripper 3970X 32-Core processor [1] at 3.9 GHz with 64 GB of RAM. The GPU is the NVidia GeForce RTX 3080 [2]. Please see [37] for the full CUDA [33] GPU ICP implementation used in this paper.

5.2 Clean and noisy 3D point sets

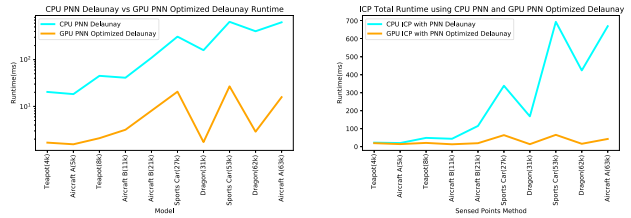
In these experiments, the 3D point sets were registered to themselves with ICP utilizing each nearest neighbor matching algorithm. The 3D point sets were evaluated at a base fidelity and downsampled by a factor of 2. The nearest neighbor algorithms compared are: *Brute Force*, *k-d Tree*, *Zero Delaunay*, *KD-ANN Delaunay*, *PNN Delaunay*, and *PNN Optimized Delaunay*.

ICP was executed with a maximum iteration count of 100 and an error of $1E-11$. The target point set was kept at identity, and the source points iterated through a series of rotations ranging from -20 to 20 degrees in 10 degree increments in roll, pitch, and yaw. This deterministically guarantees each algorithm receives the same registration inputs. Lastly, noise is added to the target points, as seen in Fig. 15a.

Because identical point clouds are registered to themselves, ICP returns the exact rotation and translation with little to no error. Additionally, as long as each nearest neighbor method provides accurate results, ICP will perform with the same accuracy. Each run of ICP in these experiments did return little to no error, demonstrating the accuracy and validity of each nearest neighbor method.

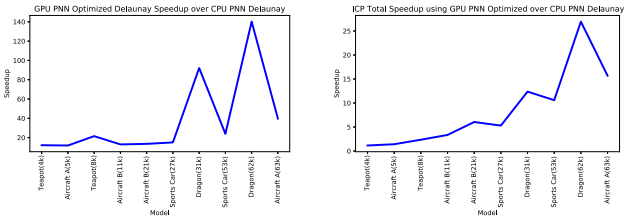
Figure 11a shows the runtime required for converging the source 3D point set onto the target 3D point set on the CPU. The *PNN Delaunay* nearest neighbor algorithm executed the fastest the CPU. Figure 11b shows the runtime required for converging on the GPU. The *PNN Optimized Delaunay* converged most quickly on the GPU. The experiments show the most efficient algorithms are the *PNN Delaunay* on the CPU and *PNN Optimized Delaunay* on the GPU. For this reason, these algorithms are compared against each other to compare performance between a CPU and GPU.

Figure 12a shows a runtime comparison of the CPU *PNN Delaunay* algorithm and the GPU *PNN Optimized Delaunay*



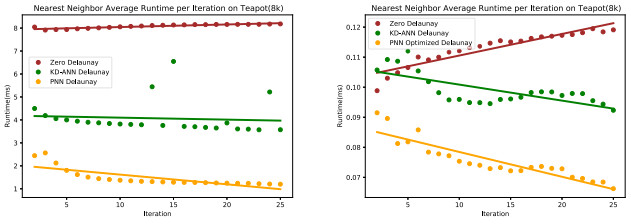
(a) Nearest Neighbor Runtime **(b)** Total ICP Runtime

Fig. 12 CPU versus GPU runtime on nearest neighbor step (**a**) and total ICP (**b**)



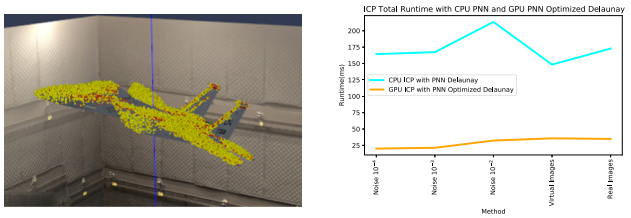
(a) Nearest Neighbor Speedup **(b)** ICP Speedup

Fig. 13 Speedup of GPU *PNN Optimized Delaunay* over CPU *PNN Delaunay* for nearest neighbor step (**b**) and entire ICP run (**a**)



(a) CPU **(b)** GPU

Fig. 14 Per iteration runtime iteration on the CPU (**a**) and GPU (**b**) running ICP on the Teapot(8k) 3D point set



CPU **(b)** GPU

Fig. 15 Aircraft B(21) 3D point set with noise added to yellow points (**a**). ICP run-times with artificial noise, virtual images and real images (**b**)

nearest neighbor algorithms. The GPU runtime is about an order of magnitude faster than the CPU. Figure 13a shows the GPU speedup over the CPU for these algorithms. The GPU achieves just under a 140× speedup on the *Dragon(62k)* 3D point set. Figure 12b shows the overall total ICP runtime and Fig. 13b shows the speedup. The GPU achieves about a whole order of magnitude in runtime and 25× speedup when comparing the most efficient algorithms.

Figure 14a shows the teapot(8k) 3D point set runtime per ICP iteration by the CPU *PNN Delaunay* algorithm. Figure 14b shows the same for the GPU *PNN Optimized Delaunay* algorithm. These are evidence the *PNN Delaunay* variants leverage the prior iteration's nearest neighbor. These algorithms increase in efficiency as iterations deepen. The negative slope in the fitted lines shows the increased efficiency.

When executing nearest neighbor in ICP, Figs. 8a, b and 9a, b show the walk frequencies of the Delaunay Walk variations on the teapot and dragon 3D point set fidelities. The *PNN Delaunay* and *PNN Optimized Delaunay* have the lowest average walks. PNN Optimized Delaunay has a lower average walk distance and variance than *PNN Delaunay* because the first iteration executes *KD-ANN Delaunay* to find a preferable starting location. Walk variance is important when executing on the GPU. If a single thread has a long walk distance, the whole thread block and possibly kernel will wait for the slowest thread to finish executing. This could also explain why the *PNN Optimized Delaunay* performs best on the GPU.

Figure 15b shows the ICP total runtime with adding noise, using virtual, and using real images. A dirty target point cloud does not negatively affect the runtime. This proves the robustness of the ICP and nearest neighbor algorithms.

5.3 Virtual and real stereo block matching

In order to demonstrate the accuracy and correctness of the Delaunay traversal, we integrated the Delaunay nearest neighbor method as step 1 of ICP. From the known geometry of our *reference* model, a 1/7th scale model aircraft, the Delaunay triangulation was generated. Then, a set of *sensed* points from stereo vision image processing were registered onto the *reference* 3D point set. With the truth position of the model aircraft from a motion capture system, we verified our Delaunay traversal by assessing the accuracy of the pose estimation.

In this experiment, stereo imagery was collected from simulating a leading aircraft with rear-facing stereo cameras mounted on its belly. The objective is to compute the relative pose between the leading and a trailing aircraft using only the stereo cameras. In doing so, the vision processing enables autonomous station keeping between two aircraft. This functionality has many applications, including formation flight and automated aerial refueling, among others. The stereo camera produces a 3D point set generated via stereo block matching [20].

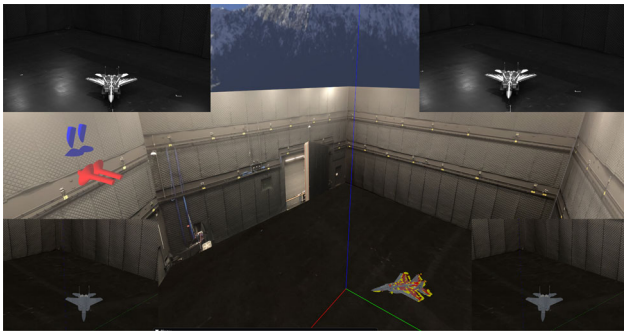


Fig. 16 ICP registering point clouds generated from both real and virtual stereo block matching. The red *reference* points are registered to the yellow *sensed* points. The real images are in the upper right and left virtual images are in the lower right and left

In terms of the experimental setup, the stereo cameras' resolution was 4096×3000 and field of view was 28.7 degrees. The baseline of the stereo system was 0.5 meters. A scale model of Aircraft B was used in an approach starting at a distance of approximately 20 meters to 14 meters from the cameras. The approach was captured with truth data recorded from a Motion Capture system [3]. Thus, the accuracy of ICP is able to be tested against the truth from the motion capture system ($<$ 1 cm, $<$ 0.1 deg). Lastly, to mimic a real-time application, ICP is executed with a maximum iteration count of 30 and *RMS* delta-error between iterations of $1E-6$. The video located at [5] shows a visualization of the motion capture environment.

In order to validate the correctness of the motion capture and stereo vision systems, truth data from the approach were replayed in a virtual simulation as a digital twin to the real images. Utilizing virtual images provides a baseline of accuracy for the motion capture and stereo vision systems. Assuming virtual images will produce less noise than real images, we utilize the virtual simulation to test and prototype various algorithms. Confidence in the system increases if processing real and virtual images produce similar results.

Virtual cameras with parameters and pose set to correspond with their real counterparts captured the 3D environment. Figure 16 shows the virtual images in the lower left and right and the real images in the upper left and right, the *sensed* points in yellow, and the *reference* points in red. In the middle left of Fig. 16 the red tubes depict the location and view direction of the stereo cameras in the real environment and virtual environment—the real and virtual stereo cameras are co-incident and share the same coordinate frame with respect to the truth motion capture system.

As seen in Fig. 17a and b, when registering points generated from virtual images, the CPU and GPU implementations perform with the same accuracy. Furthermore, when the virtual images are replaced with real images, and the points generated from stereo block matching contain more noise,

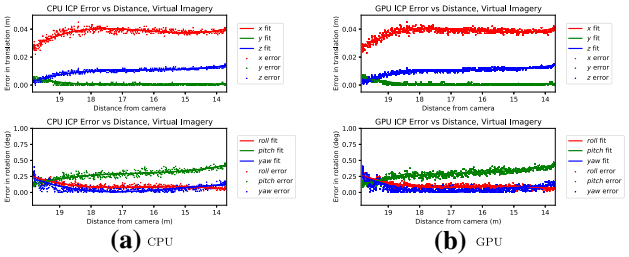


Fig. 17 Errors in translation and rotation from ICP on virtual imagery

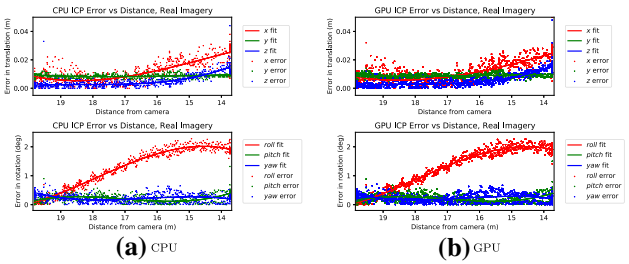


Fig. 18 Errors in translation and rotation from ICP on real imagery

Table 1 This shows the positional mean magnitude error in centimeters as well as the 99.5% confidence interval

Method	Positional mean magnitude error (cm)	99.5% confidence interval (cm) (±)
CPU	1.6373	0.0208
GPU	1.6081	0.0861

Table 2 This shows the rotational mean magnitude error in degrees as well as the 99.5% confidence interval

Method	Rotational Mean Magnitude Error (Degrees)	99.5% confidence interval (Degrees) (±)
CPU	1.2877	0.0183
GPU	1.2654	0.0695

both CPU and GPU implementations still perform with a similar level of accuracy, as is demonstrated in Fig. 18a and b.

Tables 1 and 2 show the positional and rotational mean magnitude error of ICP when using real images. The CPU and GPU have a very close error, the minor difference mostly due to hardware differences causing rounding differences. The CPU reports a slightly more accurate position, whereas the GPU reports a slightly more accurate rotation. Overall, there is less than a magnitude of 1.7 centimeters in positional error and a magnitude of 1.3 degrees of rotational error.

6 Conclusion

In this study, highly efficient and novel nearest neighbor matching algorithms were introduced and implemented based on a Delaunay triangulation. In comparison with the traditional k-d tree, the Delaunay traversal is shown to provide around 90% speedup. When integrated into ICP, the previous correspondences can cached to facilitate fewer traversals, and our research shows increasing the size of the 3D point-set has little effect on the time to find a correspondence. Finally, the accuracy of the Delaunay traversal was validated by experiments involving point registration with point-sets from real and virtual point sets.

7 Future work

Future work is planned to execute more real-time experiments with different 3D point sets. Additionally, research involving updating the *reference* point set to reflect changes in the viewpoint of the sensors is ongoing. If the Delaunay triangulation needs to be recomputed, parallel processes such as those presented in [24] may be utilized. Finally, it is planned to integrate the nearest neighbor algorithms into other applications, like point-to-plane ICP.

Acknowledgements The authors want to thank Dan Schreiter and AFRL/RQ Aerospace Systems Directorate for their help and support throughout this study. This research was supported in part by an appointment to the Postgraduate Research Participation Program at the US Air Force Institute of Technology (AFIT), administered by the Oak Ridge Institute for Science and Education through an interagency agreement between the US Department of Energy and AFIT.

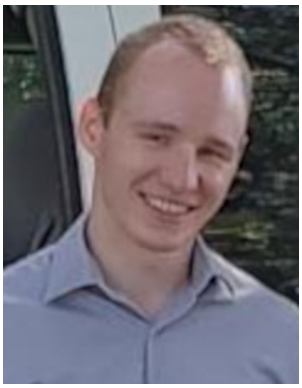
References

1. 3rd gen ryzen^{TM} threadripper^{TM} 3970x | desktop processor | amd. https://www.amd.com/en/products/cpu/amd-ryzen-threadripper-3970x. Accessed on 01/11/2021
2. Geforce rtx 3080 graphics card | nvidia. https://www.nvidia.com/en-us/geforce/graphics-cards/30-series/rtx-3080/. Accessed on 01/11/2021
3. Tracker documentation - tracker 3.9 documentation - vicon documentation. https://docs.vicon.com/display/Tracker39. Accessed on 01/13/2021
4. Amenta, N., Bern, M.: Surface reconstruction by Voronoi filtering. Discret. Comput. Geom. (1999). https://doi.org/10.1007/PL00009475
5. Anderson, J.: Delaunay walk for fast nearest neighbor matching. https://youtu.be/wI6MiRx-5Ik (2021). Accessed on 14/04/2021
6. Anderson, S.J., Karumanchi, S.B., Iagnemma, K.: Constraint-based planning and control for safe, semi-autonomous operation of vehicles. In: IEEE Intelligent Vehicles Symposium Proceeding (2012). https://doi.org/10.1109/IVS.2012.6232153
7. Barber, C.B., Dobkin, D.P., Huhdanpaa, H.: The quickhull algorithm for convex hulls. ACM Trans. Math. Softw. **22**(4), 469–483 (1996). https://doi.org/10.1145/235815.235821
8. Bellekens, B., Spruyt, V., Berkvens, R., Penne, R., Weyn, M.: A benchmark survey of rigid 3d point cloud registration algorithms. Int. J. Adv. Intell. Syst. **1**, 2015 (2015)
9. Bentley, J.L.: Multidimensional binary search trees used for associative searching. Commun. ACM (1975). https://doi.org/10.1145/361002.361007
10. Besl, P.J., McKay, N.D.: A method for registration of 3-D shapes. IEEE Trans. Pattern Anal. Mach. Intell. (1992). https://doi.org/10.1109/34.121791
11. Birn, M., Holtgrewe, M., Sanders, P., Singler, J.: Simple and fast nearest neighbor search. In: 2010 Proceedings of the 12th Workshop on Algorithm Engineering and Experiments. ALENEX 2010 (2010). https://doi.org/10.1137/1.9781611972900.5
12. Cazals, F., Giesen, J.: Delaunay Triangulation Based Surface Reconstruction: Ideas and Algorithms. INRIA Rapp, Rech (2004)
13. Chen, Y., Medioni, G.: Object modeling by registration of multiple range images. In: Proceedings of the IEEE International Conference on Robotics and Automation (1991). https://doi.org/10.1109/robot.1991.132043
14. Cover, T.M., Hart, P.E.: Nearest Neighbor Pattern Classification. IEEE Trans. Inf. Theory (1967). https://doi.org/10.1109/TIT.1967.1053964
15. Delaunay, B.: Sur la sphère du vide. Bull. l'Académie des Sci. l'URSS (1934)
16. Drost, B.H., Ilic, S.: Almost constant-time 3D nearest-neighbor lookup using implicit octrees. Mach. Vis. Appl. (2018). https://doi.org/10.1007/s00138-017-0889-4
17. Friedman, J.H., Bentley, J.L., Finkel, R.A.: An algorithm for finding best matches in logarithmic expected time. ACM Trans. Math. Softw. (1977). https://doi.org/10.1145/355744.355745
18. Greenspan, M., Yurick, M.: Approximate k-d tree search for efficient ICP. In: Proceedings of the International Conference on 3D Digital Imaging and Modeling 3DIM (2003). https://doi.org/10.1109/IM.2003.1240280
19. Guttman, A.: R-trees: a dynamic index structure for spatial searching. ACM SIGMOD Rec. **14**(2), 47–57 (1984). https://doi.org/10.1145/971697.602266
20. Kaehler, A., Bradski, G.: Learning OpenCV 3: computer vision in C++ with the OpenCV library. O'Reilly Media, Newton (2016)
21. Labatut, P., Pons, J.P., Keriven, R.: Robust and efficient surface reconstruction from range data. Comput. Graph. Forum (2009). https://doi.org/10.1111/j.1467-8659.2009.01530.x
22. Lee, D.T., Schachter, B.J.: Two algorithms for constructing a Delaunay triangulation. Int. J. Comput. Inf. Sci. **9**(3), 219–242 (1980)
23. Li, X.Y., Calinescu, G., Wan, P.J.: Distributed construction of a planar spanner and routing for ad hoc wireless networks. In: Proceedings of the IEEE INFOCOM (2002). https://doi.org/10.1109/INFCOM.2002.1019377

24. Lo, S.H.: Parallel Delaunay triangulation in three dimensions. Comput. Methods Appl. Mech. Eng. **237–240**, 88–106 (2012). https://doi.org/10.1016/j.cma.2012.05.009
25. Low, K.: Linear Least-squares Optimization for Point-to-plane ICP Surface Registration. Univ. North Carolina, Chapel Hill (2004)
26. Marden, S., Guivant, J.: Improving the performance of ICP for real-time applications using an approximate nearest neighbour search. In: Australasian Conference on Robotics and Automation ACRA (2012)
27. Meagher, D.J.: Octree encoding: A new technique for the representation, manipulation and display of arbitrary 3-d objects by computer. Electrical and Systems Engineering Department Rensseiaer Polytechnic (1980)
28. Muja, M., Lowe, D.G.: Fast approximate nearest neighbors with automatic algorithm configuration. In: VISAPP 2009—Proceedings of the 4th International Conference on Computer Vision Theory and Applications (2009)
29. Muja, M., Lowe, D.G.: Scalable nearest neighbor algorithms for high dimensional data. IEEE Trans. Pattern Anal. Mach. Intell. **36**(11), 2227–2240 (2014). https://doi.org/10.1109/TPAMI.2014.2321376
30. Mulchrone, K.F.: Application of Delaunay triangulation to the nearest neighbour method of strain analysis. J. Struct. Geol. (2003). https://doi.org/10.1016/S0191-8141(02)00067-6
31. Myronenko, A., Song, X.: Point-set registration: coherent point drift. IEEE Trans. Pattern Anal. Mach. Intell. **32**(12), 2262–2275 (2010). https://doi.org/10.1109/TPAMI.2010.46. arXiv:0905.2635
32. Nüchter, A., Lingemann, K., Hertzberg, J.: Cached k-d tree search for ICP algorithms. In: 3DIM 2007—Proceedings of the International Conference on 3D Digital Imaging and Modeling (2007). https://doi.org/10.1109/3DIM.2007.15
33. NVIDIA: CUDA Toolkit Documentation v11.2.76. https://docs.nvidia.com/cuda/
34. Pan, J., Manocha, D.: Fast GPU-based locality sensitive hashing for k-nearest neighbor computation. In: GIS Proceedings of the ACM International Symposium on Geographic Information Systems (2011). https://doi.org/10.1145/2093973.2094002
35. Pomerleau, F., Colas, F., Siegwart, R.: A review of point cloud registration algorithms for mobile robotics. Found. Trends Robot. (2015). https://doi.org/10.1561/2300000035
36. Qiu, T., Li, Y.: Nonparametric Nearest Neighbor Descent Clustering based on Delaunay Triangulation. arXiv Preprint arXiv:1502.04837 (2015)
37. Raettig, R.M., Anderson, J.D., Nykl, S.L., Merkle, L.D.: Accelerated point set registration method (2020)
38. Segal, A., Hähnel, D., Thrun, S.: Generalized-icp (2009). https://doi.org/10.15607/RSS.2009.V.021
39. Stepanov, A., Smith, J.M.G.: Modeling wildfire propagation with Delaunay triangulation and shortest path algorithms. Eur. J. Oper. Res. (2012). https://doi.org/10.1016/j.ejor.2011.11.031
40. Weinberger, K.Q., Saul, L.K.: Distance metric learning for large margin nearest neighbor classification. J. Mach. Learn. Res. (2009). https://doi.org/10.1145/1577069.1577078
41. Zhou, Q.Y., Park, J., Koltun, V.: Open3D: A modern library for 3D data processing. arXiv:1801.09847 (2018)
42. Zhu, H., Guo, B., Zou, K., Li, Y., Yuen, K.V., Mihaylova, L., Leung, H.: A review of point set registration: From pairwise registration to groupwise registration. Sensors **19**(5), 1191 (2019)

Mr. James Anderson received his Master's degree in computer engineering in 2018 from Wright State University in Dayton, Ohio, and is working on his PhD at the same institution. His research work is in collaboration with researchers at the Air Force Institute of Technology (AFIT) on simulating and analyzing flights involving automated aerial refueling. His research involves real-time visualization and image processing, as well as virtual and augmented reality environments.

Lt. Ryan Raettig is a student at the Air Force Institute of Technology pursuing a M.S. degree in Computer Engineering. His areas of interest are parallel computing, computer vision, and alternative navigation. He received his B.S. degree in Electrical & Computer Engineering at the University of Arizona in 2019, summa cum laude.



Lt. Joshua Larson is an officer in the United States Air Force. He commissioned from the University of Minnesota with a degree in computer science and completed his Masters in computer science at the Air Force Institute of Technology. His areas of interest include computer networking, machine learning, computer vision, and computer security.

Dr. Scott Nykl is an Associate Professor of Computer Science at the Air Force Institute of Technology. His areas of interest are Real Time 3D Computer Graphics, computer vision, sensor fusion, parallel processing, Interactive Virtual Worlds, and computer networking. He authored a 3D Visualization Engine and created avionics-related visualizations earning several national awards including mention in Forbe's "The Greatest Young Inventors In America". He received his B.S. degree in Software Engineering at UW-Platteville in 2006 and his M.S. and Ph.D. degrees in Computer Science from Ohio University in 2008 and 2013, summa cum laude.

Dr. Clark Taylor is currently an Assistant Professor in Computer Engineering at the Air Force Institute of Technology. He received his Ph.D. degree in electrical and computer engineering from the University of California, San Diego, in 2004. From 2004–2010, he was an assistant professor in electrical and computer engineering at Brigham Young University, and from 2010–2018, he was a research electronics engineer with the Air Force Research Laboratory (AFRL). He has published over 100 papers in the fields of video processing for unmanned aerial vehicles (UAVs), estimation theory, video communication, and digital systems design.

Dr. Thomas Wischgoll received his Master's degree in computer science in 1998 from the University of Kaiserslautern, Germany, and his PhD from the same institution in 2002. He was working as a post-doctoral researcher at the University of California, Irvine, until 2005 and is currently an associate professor and the Director of Visualization Research at Wright State University. His research interests include large-scale visualization, flow and scientific visualization, as well as biomedical imaging and visualization. His research work in the field of large-scale, scientific visualization and analysis resulted in more than thirty peer-reviewed publications, including IEEE and ACM. Dr. Wischgoll is a member of ACM SIGGRAPH, IEEE Visualization & Graphics Technical Committee, and the IEEE Compute Society.