2020 IEEE International Conference on Robotics and Automation (ICRA)
31 May - 31 August, 2020. Paris, France

Voxel-based General Voronoi Diagram for Complex Data with Application on Motion Planning

Sebastian Dorn[1], Nicola Wolpert[2], and Elmar Schömer[3]

Abstract— **One major challenge in Assembly Sequence Planning (ASP) for complex real-world CAD-scenarios is to find appropriate disassembly paths for all assembled parts. Such a path places demands on its length and clearance. In the past, it became apparent that planning the disassembly path based on the (approximate) General Voronoi Diagram (GVD) is a good approach to achieve these requirements. But for complex real-world data, every known solution for computing the GVD is either too slow or very memory consuming, even if only approximating the GVD.**

We present a new approach for computing the approximate GVD and demonstrate its practicability using a representative vehicle data set. We can calculate an approximation of the GVD within minutes and meet the accuracy requirement of some few millimeters for the subsequent path planning. This is achieved by voxelizing the surface with a common error-bounded GPU render approach. We then use an error-bounded wavefront propagation technique and combine it with a novel hash table-based data structure, the so-called *Voronoi Voxel History* (VVH). On top of the GVD, we present a novel approach for the creation of a General Voronoi Diagram Graph (GVDG) that leads to an extensive roadmap. For the later motion planning task this roadmap can be used to suggest appropriate disassembly paths.

I. Introduction

Assembly Sequence Planning (ASP) is an important task in theory and practice [1], [2]. The subject of ASP is to find at least one feasible assembly sequence for all assembled parts of an input assembly. For a general overview of ASP we refer to [3]. One of the main tasks in ASP is to find an appropriate disassembly path for each assembled part to a possible destination. An appropriate disassembly path is short and has enough clearance to its surroundings [4]. Moreover, a fast computation is crucial due to the numerous disassembly path queries performed in ASP. It turned out that motion planning based on the General Voronoi Diagram (GVD) is a good approach to achieve these requirements [5], [7]. We consider the GVD in the three dimensional workspace because the one-time calculated GVD is the same for all assembled parts. Thus it can be used for the assembly of every part. Each part is an obstacle until it is going to be disassembled. Then it changes its role from obstacle to robot. After one part has been disassembled the GVD just needs a simple update by refilling the Voronoi cell of the removed part. So the runtime bottleneck of GVD-based disassembly planning, the calculation of the GVD [7], is distributed to all assembled parts.

The ordinary Voronoi Diagram (VD) is an important geometric data structure with a lot of applications [9]. For a given set of sites the VD partitions the plane into regions where every point in a region has the same nearest input site. In the ordinary case sites are points. We generalize the VD by regarding arbitrary 3D meshes as input sites. Our algorithm assumes the Euclidean metric, but works with minor changes for all L^p-norms.

Calculating the exact GVD is a very time consuming task [10]. Since we focus on industrial data with millions of triangles, we consider its approximation. The main criteria concerning practicability are calculation time and memory usage.

The state of the art algorithms for approximating the GVD can be divided into two main classes. The first works with the exact input data but approximates the GVD. We can subclassify this in **render**- and **octree**-based algorithms. The second class approximates the input data with voxels and calculates the **distance field** that implicitly produces the GVD.

Render-based approximation algorithms cut the scene in equidistant 2D slices, calculate the GVD for each slice and stitch them together for achieving the three-dimensional GVD [23]. The GVD for one slice is calculated as follows: For every site of the input meshes render the graph of a specific distance function - that is a hyperbolic function for a point, a cone for a line and a plane for the inner triangle - and read out the z-Buffer. But rendering the graph of one of the distance functions requires a facetted version of this graph. This makes the approach too time consuming for an input with millions of triangles.

Another approach is based on octrees [20]. The main idea starts with one octree which includes the whole scene. The octree recursively subdivides itself into eight descendant octrees if it includes more than one site or a neighbor octree has a different nearest site. This proceeding terminates if the desired resolution is achieved. It results in a finer resolution at the Voronoi boundaries and a coarser one inside the Voronoi cells. However, each time an octree is subdivided into eight smaller ones, the nearest site must be calculated for each descendant. This results in a lot of distance queries, which makes this approach too slow for large input data.

Distance field approaches start with a uniform 3D voxel grid and use a scanning [11] or neighbor propagation technique [12]. These algorithms are fast because they have linear computational costs in the number of voxels, but in

[1]Digital Factory, Daimler AG, Germany
sebastian.s.dorn@daimler.com
[2]Department: Geomatics, Computer Science and Mathematics, University of Applied Science Stuttgart, Germany
nicola.wolpert@hft-stuttgart.de
[3]Department: Physics, Mathematics and Computer Science, Johannes Gutenberg - University Mainz, Germany
schoemer@informatik.uni-mainz.de

978-1-7281-7395-5/20/$31.00 ©2020 IEEE

the 3D case, the memory usage is cubic. For a sufficient resolution, this leads to an extensive memory usage.

Our Contribution: Until now there is no runtime **and** memory efficient algorithm for computing an approximate GVD for complex real-world CAD-scenarios. Our approach uses the runtime efficient concepts of voxelizing the input data and fast neighbor propagation. With our novel data structure, the *Voronoi Voxel History (VVH)*, we keep the memory utilization low. The idea is to store only the voxels needed for the current propagation step. These are mainly the voxels of the current wavefront which reduces memory usage for most practical applications from cubic to quadratic. Our algorithm is robust in the sense that it works with arbitrary meshes in arbitrary positions and is easy to implement. We also prove that the error of our approximation in the Euclidean metric is at most 2.232 times the voxel size. In addition, we present the General Voronoi Diagram Graph (GVDG) which is extracted from the GVD. It is an extensive roadmap which can be used as a basis to estimate reasonable disassembly paths for the later motion planning task. At last, we test our proposed algorithms with different resolutions on a complex real-world data set from the automotive industry.

II. Related Work

In the last decades many approaches for approximating the GVD were presented. We give a comprehensive insight into the related work of the mentioned areas of render-, octree- and distance field-based approaches.

The **render-based** approach [23] is used for many motion planning algorithms [7], [6], [5]. Based on this work the authors from [24] present an approximation for the distance field using arbitrary metrics for 2D data. In [25] there are some optimizations for the GVD visualization and shortest path queries. However, these works cannot overcome the runtime explosion for complex input data.

An extension of the basic work [20] on **octrees** is presented in [21]. The octree propagates the nearest site from the parent to the descendant octrees. This leads to smarter nearest neighbor queries, where not all input sites have to be checked. But for general input sites, it is still computationally intensive. The focus in [22] is to compute the GVD for closely spaced objects. The octree data structure is optimized for this situation and works on arbitrary 3D meshes. This approach cannot overcome the runtime problem with the nearest neighbor query at each subdividing step.

For the related work on **distance fields** until 2006 we refer to the extensive survey in [13]. The main techniques use distance templates, space scanning [11] or different voxel propagation methods [12]. After this survey, some GPU-based propagation methods were introduced [14], [18], [15], so that the propagation can be done in parallel which results in a significant speed up. The authors from [17] present a divide and conquer algorithm on the GPU. There is also a CPU based propagation algorithm with focus on fewer distance calculations [16]. But all of the presented algorithms still store the entire distance field. In [19] a memory saving version of the *Jump Flooding Algorithm* [14] is presented. In the distance field, only the nearest site is stored as a short integer and therefore the memory saving factor of 6 is achieved. But this only works for specific input sites and still cannot eliminate the cubic memory usage of distance fields.

III. General Voronoi Diagram

A. Setup

We assume for the rest of this work that the input sites are 3D meshes, each representing the body shell or one of the assembled parts. All voxels v are positioned at $v.pos$ in a $\mathbb{Z}^3$ grid. We use the Euclidean metric and define the neighborhood $N(v)$ of a voxel v as the union of the 26-neighborhood and the voxel itself. The discrete distance $d : \mathbb{R}^3 \times \mathbb{R}^3 \to \mathbb{N}_0$ between two vectors $a, b \in \mathbb{R}^3$ is defined as $d(a, b) := \lceil \|a - b\|_2 \rceil$ and the discrete distance of two voxels v_1, v_2 as $d(v_1, v_2) := d(v_1.pos, v_2.pos)$.

B. Main Idea

The main idea of our solution is to use the speed advantage of a propagation method without storing the complete distance field. The basic task during the propagation process is to find and check neighbors. In a complete distance field with a 3D array representation, this can be done with a simple index shift. To compensate the missing complete 3D array, we introduce our new data structure called Voronoi Voxel History (VVH). The VVH stores the voxels of the latest few propagation steps in several hash tables. The access time of a 3D array is $\mathcal{O}(1)$ and on average this holds for the VVH too. Our experiments show that the access time of our VVH increases just by a constant factor of 8 compared to a the access time of a distance field.

C. The Algorithm

Our algorithm works in two steps. In the first step we use the technique from [8] to voxelize the scene with a voxel resolution of $\lambda > 0$. We render the scene in 2D slices, each of width λ, and interpret each colored pixel as a voxel that belongs to a site. Different colors represent the different input sites. We call the colored voxels *startingVoxels*. They get stored in a first hash table.

The second step is the propagation algorithm. We initially begin with the startingVoxels and radius 0 and propagate the voxels in discrete, radius increasing steps to their neighbors (for a graphical representation see Figure 1). This means the radius r is both distance (measured in voxel size λ) and time step. If a neighbor is visited the first time the propagating voxel forwards the information about its starting voxel ($voxel.start$), about the distance ($voxel.dist$) to its starting voxel, and about the nearest site it belongs to ($voxel.site$) to the neighbor. If the neighbor was visited before it is a candidate for the GVD.

We now give a detailed description of the proceeding. Also consider the pseudocode in the right column.

Voronoi Voxel History (VVH): Class 1 is our voxel managing data structure. The variable $data$ is a queue of 5 hash tables (see Theorem 1). The hash table at position $1 \leq i \leq 5$ stores for a given radius $r \in \mathbb{N}_0$ all voxels with distance $r + i - 4$. We call the hash table at index $i = 4$ *currentVoxels* and the hash table at index $i = 5$ *neighborVoxels* (for a graphical representation see Figure 1 a)). The hash tables use separate chaining for collision resolution. A hash key for a voxel at position $(x, y, z) \in \mathbb{Z}^3$ which, among others, worked well for our application is $(x^3+y^2+z+xyz)$ mod $hashTable.size()$. At the end of one propagation step, the function updateVVH (lines 2 - 4) clears the latest voxels (line 3), i.e. the voxels with distance $r-3$, and inserts a new $neighborVoxels$ hash table (line 4). This represents an "aging" of 1 for all hash tables. The function findVoxel(x, y, z) searches for the voxel with the input coordinates (x, y, z) in the 5 hash tables of $data$ and returns it. If the voxel is not stored yet it returns NULL.

calculateGVD: Algorithm 2 takes the $startingVoxels$ from the voxelization step as its input. We initialize the VVH (line 1), set the $currentVoxels$ to the $startingVoxels$ and the radius to $r = 0$ (lines 2, 3). The computation of the GVD is finished when there are no more $currentVoxels$ in the VVH. The size of the hash table $neighborVoxels$ is set to the number of voxels contained in $currentVoxels$ (line 5). Since a current voxel has at most 27 neighbors, this ensures a constant average query time for the new hash table. The following loop (lines 6, 7) propagates every voxel in $currentVoxels$, filling the $neighborVoxels$ hash table. When the propagation stops, propagation step r is finished and we update r and the VVH for the next step (lines 8, 9).

propagateVoxel: Algorithm 3 propagates a voxel to its neighbors. The position $nPos$ of the neighbor is set in line 2. If the distance d from the neighbor position $nPos$ to its starting voxel ($voxel.start$) is smaller than $r + 1$ we visited this neighbor before. If d is greater than $r + 1$ we would propagate too early. In both cases we discard this neighbor (lines 3, 4). Otherwise we search the neighbor n at position $nPos$ in $data$ and call handleNeighbor (lines 5, 6).

Note: In order to ensure that the propagation is always correct we, in detail, propagate a voxel to all of its neighbor voxels. If a neighbor voxel has a distance greater $r + 1$ we store its starting voxel but it does not participate in the next propagation step.

handleNeighbor: We first check whether the neighbor has not been visited before (line 1). If yes, create and insert it into the $neighborVoxels$ hash table of the VVH (line 2). If no, there are two different cases. First, if the nearest sites of the voxel and its neighbor differ (first condition in line 3) the condition for a GVD voxel is satisfied and we add both voxels to the GVD (line 4). Second, when the nearest sites ($voxel.start.site$, $neighbor.start.site$) are equal and the distance of their starting voxels is greater than a given parameter (second condition in line 3) we broaden the GVD by also adding both voxels lying on the medial axis of the (large) input site. Finally, we update the nearest starting voxel $n.start$ of n (line 5).

Class 1 VoronoiVoxelHistory

```
1: Queue<HashTable<Voxel>> data
2: function UPDATEVVH
3:     data.dequeue()                          ▷ delete oldest voxels
4:     data.enqueue()                          ▷ add empty neighborVoxels
5: function VOXEL FINDVOXEL(x, y, z)
6:     for 1 ≤ i ≤ 5 do
7:         if voxel(x, y, z) ∈ data[i][hashkey(x, y, z)] then
8:             return voxel
9:     return NULL
```

Algorithm 2 calculateGVD($startingVoxels$)

```
1: vvh ← new VoronoiVoxelHistory
2: vvh.currentVoxels ← startingVoxels
3: r ← 0                                       ▷ Initialize the radius
4: while vvh.currentVoxels.size() > 0 do
5:     vvh.setNeighborVoxelsSize()
6:     for all voxel ← vvh.currentVoxels do
7:         propagateVoxel(vvh, voxel, r)
8:     r ← r + 1
9:     vvh.updateVVH()
```

Algorithm 3 propagateVoxel($vvh, voxel, r$)

```
1: for all (Δx, Δy, Δz) ∈ {−1, 0, 1}^3 do
2:     nPos ← voxel.pos + (Δx, Δy, Δz)
3:     if r + 1 ≠ d(nPos, voxel.start.pos) then
4:         continue
5:     n ← vvh.findVoxel(nPos)
6:     handleNeighbor(vvh, voxel, n, nPos)
```

Algorithm 4 handleNeighbor($vvh, voxel, n, nPos$)

```
1: if n == NULL then
2:     vvh.addNeighbor(voxel.start, nPos)
3: else if voxel.site ≠ n.site
           ∨ minDist < d(voxel.start, n.start) then
4:     GVD ← GVD ∪ {voxel, n}
5: n.start ← argmin_{v∈{voxel.start, n.start}} ‖n.pos − v.pos‖_2
```

From the approximate GVD obtained this way we can at the end derive a more specific approximation of the GVD by considering the intersections (see the dotted line $GVDintersection$ in Figure 1 b), which can be squares, lines or points in 3D) of all added voxel pairs $voxel$ and n.

D. Analysis

It is important to guarantee that the propagation waves from different starting voxels (as shown in Figure 1 b) for the 2D case) do not propagate through each other. We show in Theorem 1 that the VVH stores all needed voxels for this. A visual representation is shown in Figure 2 a).

Theorem 1: The VVH needs 5 hash tables for a correct propagation process.

Proof: Let $r \in \mathbb{N}_0$ be the current propagation step and v a current voxel with $v.start = A$ that propagates to its

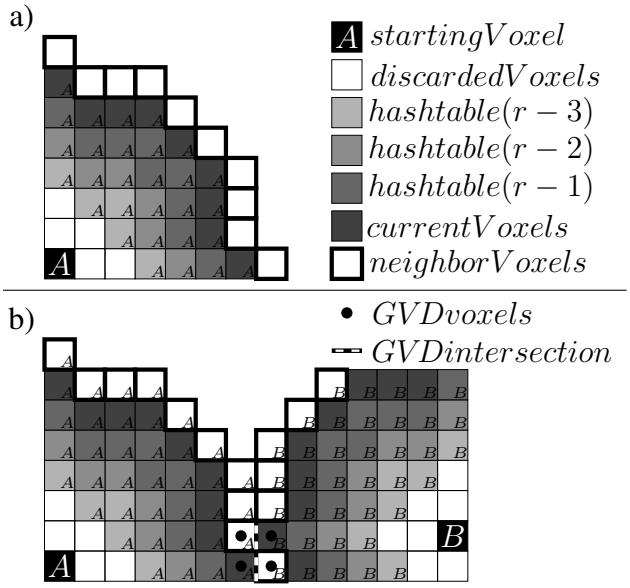

Fig. 1. a) An illustration of the data structure VVH. The A in the lower right corner symbols the starting voxel ($voxel.start$) of the associated voxel. The $discardedVoxels$ are not stored anymore and the $neighborVoxels$ are getting stored after the propagation step. b) The VVH data structure after the propagation step from the $currentVoxels$. The $neighborVoxels$ are now stored in the VVH. The $GVDvoxels$ result from voxels with different starting voxels (A and B) meeting during the propagation. The dashed line $GVDintersection$ between the $GVDvoxels$ represents the non-volumetric boundary of the Voronoi cells.

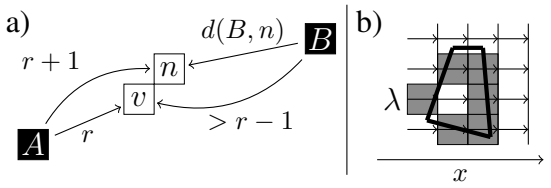

Fig. 2. a) Graphical representation of the voxel v which propagates to its neighbor n. b) The result of the render-based voxelization in $+x$ direction with voxel size λ. There are 4 slices with width λ and 4 pixels per slice, where each slice is visualized by a vertical line and the pixels by the arrows. If an arrow hits the mesh, the corresponding pixel gets colored.

neighbor n. This means that $d(A, v) = r$ and $d(A, n) = r+1$. Now assume that n was visited before with $n.start = B$.

We want to prove that the information that n was visited before is still stored in the VVH.

Since v has A as its nearest starting voxel it follows that $d(B, v) > r - 1$. This leads to

$$r - 1 < d(B, v) \leq d(B, n) + d(v, n) \leq d(B, n) + \left\lceil \sqrt{3} \right\rceil .$$

We derive $d(B, n) > r - 3$.

Under the assumption that we could compute the numerically correct distance between voxels it would be sufficient to store the $neighborVoxels$ $(r+1)$, the $currentVoxels$ (r) and the two previous hash tables $r-1$ and $r-2$. In order to deal correctly with numerical inaccuracies one additionally has to store the hash table $r - 3$. ■

Next, we prove the error-bound for the approximation of our GVD. We assume that the voxel size λ is sufficiently small so that every site and substantial subgeometry is intersected by the rasterization.

Theorem 2: Let $\lambda > 0$ be the voxel size. Then the approximation of the GVD has an error of at most 2.232λ.

Proof: With a voxel size of λ and the render process in different directions (as described in [8]) we can guarantee that the voxelization produces an error of at most $\lambda/2$ (for a graphical representation see Figure 2 b)).

Let $a, b \in$ GVD be two neighbored voxels with different starting voxels. We guarantee in Algorithm 3 (lines 3, 4) that the discrete steps respect the Euclidean metric. So the exact GVD of the starting voxels $a.start$ and $b.start$ passes somewhere through the union of the voxels a and b. By setting the approximate GVD to the intersection of a and b (see the dotted line $GVDintersection$ in Figure 1 b)) and by having a space diagonal of length $\sqrt{3}\lambda$ for a voxel we get an error of at most $\sqrt{3}\lambda$.

This yields an overall error of at most $(1/2 + \sqrt{3})\lambda \approx 2.232\lambda$. ■

IV. General Voronoi Diagram Graph

As shown in [5], finding a disassembly path on a GVD-based roadmap, called *General Voronoi Diagram Graph* (GVDG), is runtime effective for finding short paths with sufficient clearance. The GVDG is an undirected graph $G := (V, E)$ where the set of nodes V as well as the set of edges E are part of the GVD. One straightforward approach to define nodes is to set $V := \{v \in \text{GVD} : \delta(v) \geq 4\}$. All voxels with degree $\delta(v) = 3$ belong to edges [7]. Here the degree $\delta(v) \in \mathbb{N}$ of a voxel $v \in$ GVD is defined by the number of neighbors $n \in N(v)$ with pairwise different nearest sites.

However, this approach ignores voxels v on the surfaces ($\delta(v) = 2$) and on the medial axis ($\delta(v) = 1$). Therefore, the GVDG misses important edges with respect to path length and clearance and *isolated sites* can occur. A site s is isolated if there is no path from boundary voxels of its Voronoi cell to other nodes in the GVDG. This happens if the GVDG has more than one connected component or a Voronoi cell has no voxel v with $\delta(v) \geq 4$. For a graphical representation of an isolated site in the two dimensional case see Figure 3. The shown site s has no voxel v on the boundary of its Voronoi cell with $\delta(v) \geq 3$ ($\delta(v) \geq 4$ in 3D). Thus no voxel of the Voronoi cell of s is an element of V. Even if there would be a node, e.g. v_2, without respecting the medial axis for the edges E the node v_2 would have no connection to the node $v_1 \in V$. It turns out that these cases often happen in complex real-world scenarios.

Unlike the straightforward approach, our approach handles isolated sites and provides a detailed roadmap. We set the nodes $V := \{v \in \text{GVD} : \delta(v) \geq 4\}$ just like in the straightforward approach. If a site s has no voxel v on the boundary of its Voronoi cell with $\delta(v) \geq 4$ we add some voxels from the boundary with $\delta(v) < 4$ to the nodes V.

To define the edges E, we start a simultaneous neighbor propagation on the GVD for every node $v \in V$. A voxel v is only allowed to propagate to the voxels $N(v) \cap$ GVD. If two

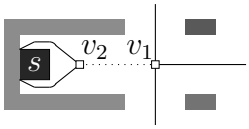

Fig. 3. A two dimensional assembly that shows four sites and the corresponding GVD. The solid line represents the boundary of the Voronoi cells and the dotted line the medial axis. The voxels v_1 and v_2 have degree $\delta(v_2) = 2$ and $\delta(v_1) = 3$.

propagating voxels $v_1, v_2 \in$ GVD meet each other and their starting voxels $v_1.start, v_2.start \in V$ have no connecting edge $e \in E$ yet the edge is created. Since we also stored the medial axis points in the GVD we can receive connections between non-adjacent Voronoi cells. In addition, every site s contributes at least one node $v \in V$. The GVDG provides a path for the site s starting from v to every $node$ in V. Here we exclude special cases like a site that is watertightly enclosed by other sites (e.g. a sphere in a greater sphere).

V. Evaluation

In this section we evaluate the GVD and the resulting GVDG for different voxel sizes λ. The focus for the GVD is on runtime and memory consumption. We compare the number of used voxels of our approach with the ones used by a complete distance field. In the analysis of the GVDG we consider the runtime, the size of the graph and the number of isolated sites and compare it with the mentioned straightforward approach [7]. In addition, paths proposed by a modified Dijkstra algorithm are evaluated.

The results are benchmarked on a laptop with an Intel i7-6820HQ (2.70GHz) processor, 32 GB of RAM and an NVIDIA Quadro M2000M graphics card. The software (64bit binary) is written in C++ and compiled with MS Visual C++ 15.9. All algorithms use floating precision and are executed, except the rendering step, on a single core of the CPU.

A. Test Data

We evaluate our algorithms based on two subsets of a representative real-world data set which contains a high number of complex and differently positioned input sites. A graphical representation can be found in Figure 4. The attributes of the data are listed in Table I. Column "#sites" gives the number of input sites the body shell and all assembled sites consist of. "#triangles" shows the number of triangles of the whole assembly (bodyshell and assembled sites). The dimensions of the oriented bounding box containing the assembly are shown in "OBB size". Data set b) contains with 1728 sites consisting of over 12 million triangles almost all components of a complete vehicle. Data set a) is a subset of b) and is used to show the scalability of our algorithms. Since the slimmed down version is much more suitable for the graphical representation of our results the following images are based on data set a).

B. General Voronoi Diagram

The results of the tests for the GVD calculation are for both data sets and with different voxel sizes presented in

TABLE I
The attributes of the data sets.

Test Data	#sites		#triangles	OBB size [mm]		
data set	*body shell*	*assembled parts*	*assembly*	x	y	z
a)	50	137	885,000	2,623	1,786	1,292
b)	692	1,036	12,168,000	4,617	1,991	1,317

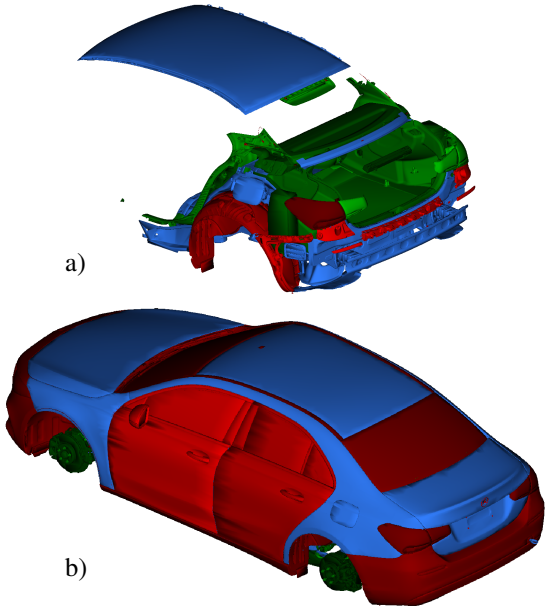

Fig. 4. The test data represent two different subsets of the Mercedes-Benz A-Class. The blue sites belong to the body shell and the different-colored ones are the assembled sites.

Table II. In addition, Figure 5 shows data set a) and our computed GVD. We set the parameter $minDist$ for the medial axis condition in Algorithm 4, line 3 to 50 mm. Column "t" contains the overall running time of our implementation for voxelizing the scene and calculating the GVD. The voxelization step needs only a small percentage (<5%) of the overall calculation time. We can compute the GVD for the almost complete car in dataset b) with a high resolution of $\lambda = 5$ mm in less than 25 minutes. Furthermore, the test on data set a) with $\lambda = 20$ mm and t = 8 seconds shows that our approach is downscalable. Our algorithm has its focus on the handling of very large and complex data but it is also usable for smaller datasets or coarser grids where the focus is on a fast calculation time.

The following two columns show the number of voxels needed by our GVD and the complete distance field (DF). Regarding the space consumption of a single voxel, a voxel in the VVH needs to store its position ($voxel.pos$). This is due to the unstructured ordering in the hash table and is not necessary for a distance field. But since in our application the GVD or DF computation is only a preprocessing step for the subsequent GVDG calculation, it is recommended to equally store for each voxel: its starting voxel ($voxel.start$), its position ($voxel.pos$), its clearance ($voxel.dist$) and a list of neighboring voxels with different nearest sites. This has the effect that the space consumption of a single voxel is almost identical for both data structures.

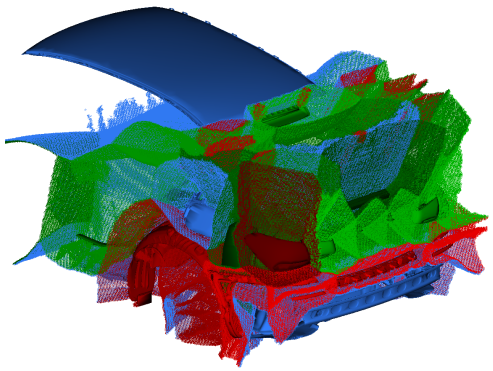

Fig. 5. Data set a) and its GVD with a voxel size of $\lambda = 5$ mm. The GVD is represented by the boundaries of all Voronoi cells which are colored the same as the corresponding input site (see Figure 4 a)).

TABLE II

THE EVALUATION OF THE GVD CALCULATION.

data set	λ [mm]	GVD		DF	ratio
		t [s]	*#voxels*	*#voxels*	#voxels
a)	20	8	490,000	750,000	1.54
	10	72	2,280,000	6,050,000	2.65
	5	525	13,580,000	48,420,000	3.56
	2	11,687	68,478,000	756,575,000	11.04
b)	20	25	1,080,000	1,510,000	1.39
	10	160	7,050,000	12,100,000	1.71
	5	1,453	40,210,000	96,840,000	2.40

Therefore, the presented number of voxels for different resolutions in Table II shows the quadratic memory growth of our approach compared with the cubic growth of a complete distance field. As a consequence, the "ratio" between the number of voxels for a complete distance field and for our approach increases as the voxels become finer. The need for a voxel saving approach becomes apparent for data set a) with a voxel size of $\lambda = 2$ mm. Here the voxel ratio is already 11.04. The DF approach had a constant calculation speed up of approximately 8.

C. General Voronoi Diagram Graph

The results of the tests for computing the GVDG are shown in Table III. The column "t" shows the runtime for the GVDG calculation based on the GVD and a subsequent Dijkstra-based path search along the edges of the GVDG for each assembled and connected site. Whereby the path search for a site s starts at the nodes of the corresponding Voronoi cell and ends by a worker starting point. We thined out the set of nodes V by removing duplicate nodes, which have another node within a small distance depending on the voxel size. Therefore the number of nodes $|V|$ varies. Column "isoSites" shows the number of isolated sites of the straightforward approach. The high number shows that isolated sites often occur in complex scenarios. The number of isolated sites varies because sometimes very small sites are not voxelized depending on the voxel size λ. The last column "saveDis" shows the number of sites for which the path search has found a save disassembly path for which the clearance is greater than the diameter of the site. These sites can be disassembled with no further check. Here the motion planning in the close-up range from the installed position to a node of the GVDG is neglected.

TABLE III

THE EVALUATION OF THE GVDG CALCULATION.

data set	λ [mm]	t [s]	$\|V\|$	$\|E\|$	isoSites	saveDis
a)	20	10	963	9,896	68	85
	10	6	633	6,816	54	106
	5	15	318	3,598	50	108
b)	20	298	17,126	111,290	636	341
	10	703	13,784	84,892	574	343
	5	1463	9,339	48,574	601	400

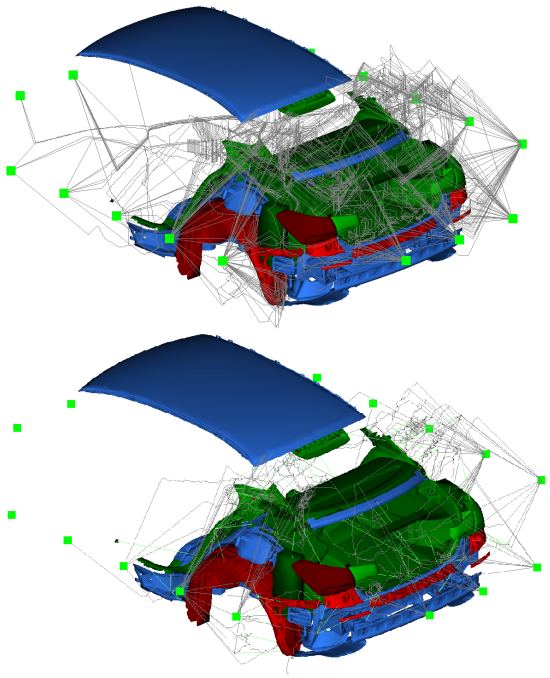

Fig. 6. Above: our computed GVDG. Below: the GVDG computed with the straightforward approach. The green squares represent example goals for disassembly paths.

A graphical representation of the GVDG is shown in Figure 6. As a comparison, a graph based on the straightforward approach is created. One can see that our proceeding provides a much more detailed roadmap which is able to reach all user-defined goal nodes. The goal nodes represent the worker starting points around the car.

VI. CONCLUSION AND FUTURE WORK

We presented an error-bounded approach for approximating the GVD of triangulated 3D CAD-data for all L_p-norms. Since it uses rendering and a fast propagation technique, it can compute results with a sufficient resolution and running time. The main contribution of our work is a data structure which reduces the previous cubic memory usage to quadratic for our practical applications. Our experiments showed that this makes the propagation approach workable for complex real-world data like a complete car. In addition, we proposed a detailed and connected roadmap, called GVDG, for subsequent motion planning. Based on our results we will in the future evaluate GVD-based motion planners for assembly sequence planning.

REFERENCES

[1] T. Ebinger, S. Kaden, S. Thomas, R. Andre, N. M. Amato, and U. Thomas, "A general and flexible search framework for disassembly planning," International Conference on Robotics and Automation (ICRA), pp. 1-8, 2018.

[2] I. Aguinaga, D. Borro, L. Matey, "Parallel RRT-based path planning for selective disassembly planning," International Journal of Advanced Manufacturing Technology, pp. 1221-1233, 2008.

[3] P. Jiménez "Survey on assembly sequencing: a combinatorial and geometrical perspective," Journal of Intelligent Manufacturing, pp. 235-250, April 2013.

[4] M. Davoodi, F. Panahi, A. Mohades, S. N. Hashemi, "Clear and smooth path planning," Applied Soft Computing, pp. 568-579, 2015.

[5] R. Geraerts, "Planning short paths with clearance using explicit corridors," International Conference on Robotics and Automation (ICRA), pp. 1997-2004, 2010.

[6] M. Garber, M. C. Lin, "Constraint-based motion planning using Voronoi Diagrams," International Conference on Robotics and Automation (ICRA), pp. 541-558, 2002.

[7] M. Foskey, M. Garber, M. C. Lin, D. Manocha, "A Voronoi-based hybrid motion planner," International Conference on Intelligent Robots and Systems, 2001.

[8] S. Fang, H. Chen "Hardware accelerated voxelization," Springer Tracts in Advanced Robotics, pp. 433-442, June 2000.

[9] A. Okabe, B. Boots, K. Sugihara, and S. N. Chiu "Spatial tessellations: concepts and applications of Voronoi Diagrams," 2nd ed., John Wiley, Hoboken, NJ, 2000.

[10] J.-D. Boissonnat, C. Wormser, and M. Yvinec, "Effective computational geometry for curves and surfaces," Springer, pp. 67-116, 2006.

[11] P.-E. Danielsson, "Euclidean distance mapping," Computer Graphics and Image Processing, pp. 227-248, 1980.

[12] I. Ragnemalm, "Fast erosion and dilation by contour processing and thresholding of distance maps," Pettern Recognition Letters 13, pp. 161-166, 1992.

[13] M. W. Jones, J. A. Bærentzen, and M. Sramek, "3D distance fields: a survey of techniques and applications," IEEE Transactions on Visualization on Computer Graphics, vol. 12, no. 4, pp. 581-599, July/August 2006.

[14] G. Rong, T. Tan, "Jump Flooding in GPU with applications to Voronoi Diagram and distance transform," Symposium of Interactive 3D graphics and games, pp. 109-116, 2006.

[15] J. Schneider, M. Kraus, and R. Westermann, "GPU-based real-time discrete euclidean distance transforms with precise error bounds," VISAPP, vol. 8, pp. 435-44, 2009.

[16] M. Velic, D. May, and L. Moresi, "A fast robust algorithm for computing discrete Voronoi Diagrams," Journal of Mathematical Modelling and Algorithms, pp. 343-355, 2009.

[17] T.-T. Cao, K. Tang, A. Mohamed, and T.-S. Tan, "Parallel banding algorithm to compute exact distance transform with the GPU," SIGGRAPH, pp. 83-90, 2010.

[18] L. Guo, F. Wang, Z. Huang, and N. Gu, "A fast and robust seed flooding algorithm on GPU for Voronoi Diagram generation," International Conference on Electrical and Control Engineering, pp. 492-495, 2011.

[19] Z. Yuan, G. Rong, X. Guo, and W. Wang, "Generalized Voronoi Diagram computation on GPU," International Symposium on Voronoi Diagrams in Science and Engineering, 2011.

[20] D. Lavender, A. Bowyer, J. Davenport, A. Wallis, and J. Woodwark, "Voronoi Diagrams of set-theoretic solid models," IEEE Comput. Graph, pp. 69-77, September 1992.

[21] I. Baston, N. Celes, "Approximation of 2d and 3d generalized Voronoi Diagrams," International Journal of Computer Mathematics, pp. 1003-1022, 2008.

[22] J. Edwards, E. Daniel, V. Pascucci and C. Bajaj, "Approximating the Generalized Voronoi Diagram of closely spaced objects," EUROGRAPHICS, vol. 34, no. 2, 2015.

[23] K. E. Hoff, J. Keyser, M. Lin, D. Manocha, and T. Culver, "Fast computation of generalized Voronoi Diagrams using graphics hardware," SIGGRAPH, pp. 277-286, 1999.

[24] R. Strzodka, and A. Telea, "Generalized distance transforms and skeletons in graphics hrdware," IEEE TCVG Symposium on Visualization, pp. 221-230, 2004.

[25] H. Hsieh, and W. Tai "A simple GPU-based approach for 3D Voronoi diagram construction and visualization," Simulation Modelling Pratice and Theory, pp. 681-692, September 2005.