2012 IEEE International Conference on Robotics and Automation
RiverCentre, Saint Paul, Minnesota, USA
May 14-18, 2012

High Quality Conservative Surface Mesh Generation for Swept Volumes

Andreas von Dziegielewski, Michael Hemmer and Elmar Schömer

***Abstract*— We present a novel, efficient and flexible scheme to generate a high quality mesh that approximates the outer boundary of a swept volume. Our approach comes with two guarantees. First, the approximation is conservative, i.e., the swept volume is enclosed by the generated mesh. Second, the one-sided Hausdorff distance of the generated mesh to the swept volume is upper bounded by a user defined tolerance. Exploiting this tolerance the algorithm generates a mesh that is adapted to the local complexity of the swept volume boundary, keeping the overall output complexity remarkably low. The algorithm is two-phased: the actual sweep and the mesh generation. In the sweeping phase we introduce a general framework to compute a compressed voxelization. The phase is tailored for an easy application of parallelization techniques. We show this for our exemplary implementation and provide a multi-core solution as well as a GPU based solution using CUDA. The meshing phase utilizes Delaunay refinement which we carefully modified such that required guarantees are met. The approach is able to handle inputs of very high complexity at desired precision, which we demonstrate on real industrial data sets.**

I. Introduction

The swept volume SV is defined as the set of all points touched by a solid (the generator) while performing a motion. Its computation plays an important role in computer aided design (CAD), numerically controlled (NC) machining verification and graphical modeling. In robotics and motion planing the swept volume can be used for the determination of workspace or to efficiently verify that an already computed path remains valid while other parts of the design may change.

Most applications demand a mesh $\mathcal{M}$ that *conservatively* approximates the outer boundary of SV, that is, SV should be enclosed by $\mathcal{M}$. At the same time, the approximation error of $\mathcal{M}$ to SV should be globally bounded. An appropriate measure for this error is the one-sided Hausdorff distance, that, for two compact sets $\mathcal{A}, \mathcal{B} \subset \mathbb{R}^3$, is defined as $h(\mathcal{A}, \mathcal{B}) = \max_{a \in \mathcal{A}} \min_{b \in \mathcal{B}} d(a, b)$, where $d(\cdot, \cdot)$ is the Euclidean distance. For instance, assuming that $h(\mathcal{M}, SV) = \delta$, one can bound the Euclidean distance Δ of SV to an obstacle $\mathcal{O}$ by $d(\mathcal{O}, \mathcal{M}) \leq \Delta \leq d(\mathcal{O}, \mathcal{M}) + \delta$, which is crucial for clearance checks, maintainability analysis, and similar tasks.

This work has been supported in part by the 7th Framework Programme for Research of the European Commission, under FET-Open grant number 255827, CGL, Computational Geometry Learning.

Andreas von Dziegielewski is a PhD student at the faculty of Physics, Mathematics and Computer Science at the Johannes Gutenberg-university of Mainz dziegiel@uni-mainz.de

Michael Hemmer is with the School for Computer Science, Tel Aviv University mhsaar@gmail.com

Elmar Schömer is with the faculty of Physics, Mathematics and Computer Science at the Johannes Gutenberg-university of Mainz schoemer@uni-mainz.de

Furthermore, it is important to keep the complexity of $\mathcal{M}$ low. This is becoming more and more relevant due to increasing model complexity and high demands concerning error tolerance. Moreover, a practical algorithm should be tolerant towards topologically inconsistent input data and should preferably impose no restrictions on the input models.

A. Previous and related work

Mathematical formulations describing the swept volume include Jacobian rank deficiency methods, sweep differential equations and envelope theory; for a survey see the work of Abdel-Malek et al. [1]. For the special case of sweeping a polyhedral object, Weld and Leu [13] have shown that it suffices to compute the swept volume of the polygonal faces, which leads to ruled and developable surfaces and polygons of the generator itself. Polyhedral approximations of these surfaces were proposed by Abrams and Allen [2]. A similar approach was recently presented by Campen and Kobbelt [5], who apply local culling criteria to rule out triangles that will not contribute to the outer boundary of the swept volume approximation. For both approaches the resulting mesh is non-adaptive and possibly very complex since it is formed by all (portions of) triangles that are reachable from the outside.

Kim et al. [8] as well as Himmelstein et al. [7] compute a directed distance field to the tessellated ruled and developable surfaces utilizing the GPU. The resulting mesh is then extracted via isosurface reconstruction. However, the massive memory demand for storing a full volumetric distance field limits the geometric accuracy of the approximation. Furthermore, as they rely on GPU rasterization, they cannot give geometrical guarantees because sharp features can be missed. Recently Zhang [14] et al. extended the approach of [8] to give topological guarantees but do not regard conservativeness. The approach is able to overcome the memory issues of [8] by using a memory friendly adaptive distance field, but this is at the price of increased run time, which finally limits the applicability of their approach. Schwanecke and Kobbelt [11] used an octree to store a volumetric approximation of the swept volume, unfortunately their method restricts the generator to a set of spheres.

The only method that regards conservativeness is discussed in [12], but since the approach uses a depth buffer voxelization it can not give guarantees for concave regions. Moreover, it generates a very complex intermediate mesh, which limits the applicability of the approach.

In summary, all listed approaches have the deficiency that they produce a highly over-tessellated mesh in the first place, which usually requires a post processing step that may annihilate possible guarantees. One can expect that most

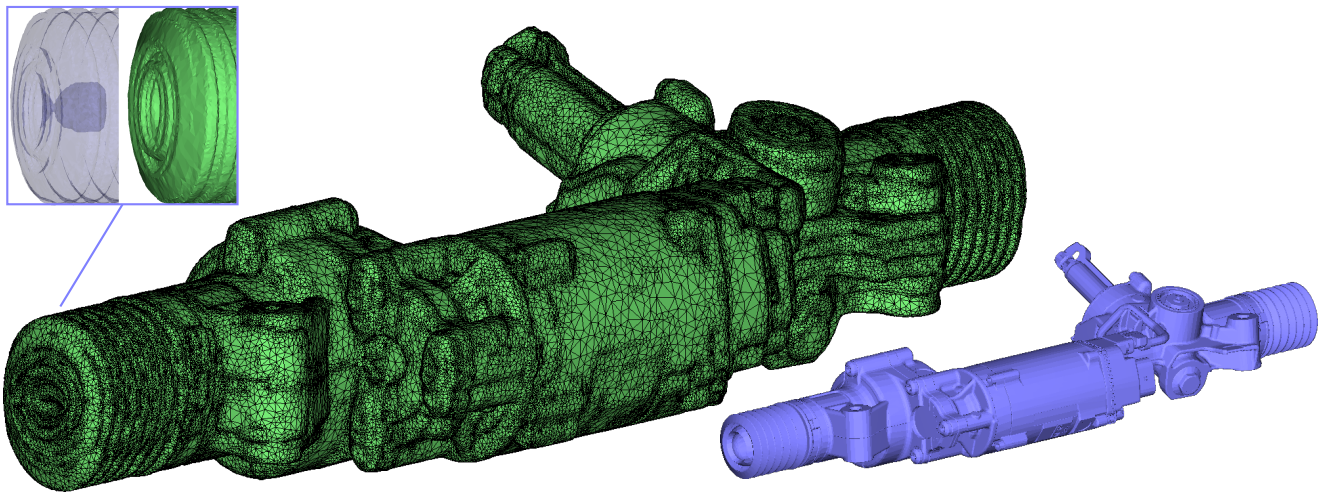

Fig. 1. Steering gear scenario. The input trajectory consists of 27k transformations. It was obtained by a dense sample of the displacement motion during a test drive. Lower right: The original component. Middle: swept volume boundary approximation with only 28k triangles. Upper left: the algorithm explores a concavity formed by a hollow part of the generator.

of them [2], [5], [8], [7], [14] are unable to handle very complex input data at reasonable computational time (as required by our applications), or they cannot give global error bounds [12].

B. Our contribution

As discussed above, a major problem of previous approaches is the high complexity of the output or at least of intermediate stages. The scheme presented in this paper proposes an initial sweeping phase (Section II), which accumulates only the necessary information in a compressed voxelization $\mathcal{V}_0$. The subsequent meshing phase (Section III) applies a carefully modified Delaunay refinement [4] and provides a non-uniform mesh $\mathcal{M}$ that conservatively approximates $\mathcal{V}_0$ while simultaneously bounding the one-sided Hausdorff distance of $\mathcal{M}$ to $\mathcal{V}_0$. The resulting mesh is of remarkably low complexity as it nicely adapts to the local complexity of the swept volume boundary as shown in Figure 1.

Section IV presents detailed benchmarks including highly complex data sets and compares a multi-core and GPU based versions of our implementation. The paper closes with conclusions and a discussion on further work.

II. Sweeping phase

For a given generator and trajectory, the sweeping phase computes a compressed voxelization of the swept volume. The side length of a voxel is $\varepsilon = 2^{-D}$, where D is provided by the user. It is assumed that the generator and its trajectory are scaled such that the swept volume fits into the $[0,1]^3$ cube.

The subsequent discussion of the sweeping scheme focusses on the memory efficient management of the generated voxels, which is essential to achieve the required precisions (see also Section IV). Thus, for now, the actual voxel generation method is treated as a black box. However, it is assumed that the chosen method is able to generate a watertight voxelization that covers at least the boundary of the swept volume that is defined by a requested portion of the trajectory.

A. Memory Management

1) Octree: In order to decrease the memory usage we store a voxelization in a pointer-less octree that is internally represented using a hash set.[1] A cell of the octree is encoded by a 4-tuple of (short) integers (i,j,k,ℓ), corresponding to the cube $[iL, iL+L]\times[jL, jL+L]\times[kL, kL+L]$, where $L = 2^{D-\ell}$. Thus, on level $\ell = D$ each cell exactly corresponds to one voxel. A cell is marked as occupied by its existence in the hash set. In case all 8 children of a cell exists the cell is created and the children are deleted. Thus, a voxel is not occupied if the voxel itself and none of its ancestors exist in the hash set. Obviously, the octree has $D = \log 1/\varepsilon$ levels. Thus, insertion (amortized) as well as testing containment is in $O(\log 1/\varepsilon)$.

2) Sweep: For efficiency reasons, we can assume that the chosen voxel generation method tries to generate only voxels that contribute to the outer boundary, i.e., by ruling out inner geometry using local culling criteria as, e.g, used in [2], [8], and [5]. However, since this local filtering can not be perfect, the complexity of the generated voxel set $\mathcal{V}$ is usually cubic. At the same time, the octree is hindered to combine voxels into larger cells since several voxels of the volume are missing. Thus, even with the octree, we would easily get a memory consumption of $O(1/\varepsilon^3)$.

Therefore, we propose to indeed structure the voxel generation in a sweep-like manner, that is, starting at the initial position we request the black box to generate only voxels up to a certain time step. While this time step progresses we keep track of the memory usage. As soon as we detect that memory becomes scarce, the sweep is interrupted and we start a compression phase that fills up all interior holes of $\mathcal{V}$.

[1] Using `boost::unordered_set`; `www.boost.org`.

This compression step, which we discuss in the next section, can be repeated during the sweep whenever it is necessary. In any case, we do at least one compression at the very end of the sweep. Since the compression fills all inner holes, we can expect that, for reasonable volumes, the size of the resulting octree is $O(1/\varepsilon^2)$, i.e., proportional to the area occupied by the outer boundary of $\mathcal{V}$.

3) Compression: A naive approach consists in applying a flood-fill method: One first computes a *hull* $\mathcal{H}$ by crawling along the outer boundary of $\mathcal{V}$ storing every outer voxel that has a neighbor in $\mathcal{V}$. This hull is then *flooded* starting from an initial *seed* voxel (any voxel that was generated before) and recursively exploring all its neighbors. The voxels generated this way replace the old set $\mathcal{V}$. However, this naive approach would touch all voxels in the volume and would take $O(\varepsilon^{-3} \log 1/\varepsilon)$ time. We here present a flooding method that carefully applies a hierarchical scheme. The idea is to fill up the large innermost cells of the octree first.

The hull $\mathcal{H}$ is computed as before and represented as an octree with depth D. It is a thin layer of voxels coating $\mathcal{V}$. We now incrementally coarsen $\mathcal{H}^0 = \mathcal{H}$ by applying the following rule: every direct parent of a leaf of $\mathcal{H}^{i-1}$ becomes a part of the new layer and is stored in $\mathcal{H}^i$ having only depth $D - i$. Note that this process inflates the hull, which also occupies parts of $\mathcal{V}$. The parts of the volume that are not occupied on the coarse layers are exactly those that we want to fill first in the subsequent filling step.

In order to distinguish the interior from the exterior volume it is also essential to have a seed inside, which starts the flooding. Therefore, we also coarsen the given seed. In case the cell of the seed gets occupied by the inflating hull we try to rescue it to a neighboring cell that is not yet occupied. However, since the inflating hull would eventually occupy the full cube, the seed must die at some point. Hence, we start the flooding on the most coarse level for which we were able to keep the seed. We then use the resulting volume as seed for the next finer layer and so on. Thus, in case the seed did not die too early, we can expect that we only touch a quadratic number of cells, which corresponds to a runtime of $O(\varepsilon^{-2} \log 1/\varepsilon)$

In order to increase the success rate of this heuristic, and since the volume may also have several disconnect regions on coarse levels, we simply maintain several seeds simultaneously. This strategy was sufficient in all tested scenarios.

B. Implementation Details

In our application we are given a discrete trajectory as the input is the result of a *densely* sampled continuous motion (Figure 1). Thus, it is reasonable to define a polyhedral approximation of the swept volume by linearly interpolating between time steps. That is, we tessellate the boundary of each swept edge with two triangles by inserting the diagonal with the lower dihedral angle, as proposed by [5]; see also Figure 2. However, in constrast to [5] these triangles are never stored since they are immediately voxelized.

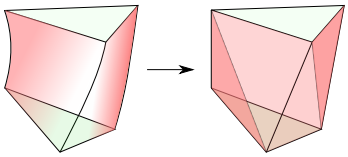

Fig. 2. Sweeping a triangle yields three ruled surfaces, each of which is approximated and tessellated by inserting a diagonal.

The actual voxelization of each triangle is based on a simple recursive subdivision scheme. Starting from the initial cube, the cubes that still intersect the triangle are subdivided into eight smaller cubes of equal size until the required resolution is reached. The intersection test is based on the separating axis theorem. However, similar to [3], the code is explicitly designed to take advantage of the fact that all cubes share the same orientation.

1) Parallelization multi-core: The voxelization of the different triangles is obviously a task that can be spread among several processors. However, it is crucial that all threads work as independent as possible from each other. In particular, only one thread can write into the octree at a time. We call this thread the master thread. All other threads are responsible for voxel generation and write these voxels into buffers that are local to each thread. The master thread then picks up these voxels from time to time and inserts them into the octree.

The above scheme is simple since it requires almost no synchronization among threads. However, already with a very small number of threads the actual bottleneck is the insertion of voxels into the octree. This is to a large extent caused by the fact that during the sweep most voxels are actually generated several times. Hence, the idea is to shift the work to the other threads and to *filter* out redundant voxels before they reach the master thread as much as possible. In order to do so we keep a copy of the main octree. Since this copy does not change it can be used by the other threads to discard already generated voxels. The copy is only updated from time to time. Therefore, we are able to keep the synchronization overhead at a very low level while only a few redundant voxels reach the master thread.

2) Parallelization GPU: The GPU version of the voxelization algorithm is implemented using CUDA. Instead of voxelizing each triangle separately, now chunks of triangles are collected and voxelized on the GPU in a single step. In order to reduce the number of redundant voxels we unify the generated voxels of each chunk on the GPU using thrust[2]. Thereafter, the voxels are downloaded to CPU memory and inserted into the octree.

III. Mesh generation phase

We compute the mesh $\mathcal{M}$ using Delaunay refinement [4]. However, we carefully modified the algorithm such that we can ensure that $\mathcal{M}$ is contained in a tolerance volume of the voxelization from the previous phase. This allows us to ensure conservativeness and a bound on the one-sided

[2]http://code.google.com/p/thrust/

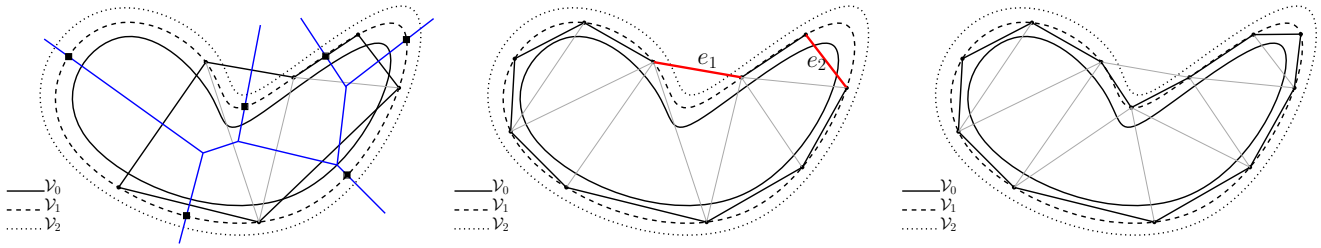

Fig. 3. Illustration of Delaunay refinement for our approach in 2D (voxelization omitted). To the left: A possible initial state. An initial set of sample points induces a Delaunay triangulation and the corresponding Voronoi diagram. Possible refinement points are indicated as black squares. In the middle: e_1 (e_2) is scheduled for refinement since it leaves $\mathcal{V}_2$ (intersects $\mathcal{V}_0$). To the right: A final mesh, all boundary edges (triangles in 3D) are in the desired tolerance region $\mathcal{V}_2 \setminus \mathcal{V}_0$.

Hausdorff distance while keeping the complexity of $\mathcal{M}$ low. We first give a brief review of Delaunay refinement.

A. Delaunay refinement

For a given domain $\mathcal{D}$, the algorithm produces a mesh $\mathcal{M}$ that approximates the boundary of the domain, $\partial\mathcal{D}$. Starting from an initial point set on $\partial\mathcal{D}$, the process maintains a Delaunay triangulation of this point set. This is a 3D complex, that is, a set of faces with dimension 0 (vertices), 1 (edges), 2 (facets/triangles) and 3 (cells/tetrahedra). Thereby, the algorithm classifies each tetrahedron as interior or exterior. A facet belongs to the mesh if the classification of the two neighboring tetrahedra differs. Such a boundary surface facet f can be refined by inducing a new point, namely an intersection point of its Voronoi edge (the dual of f) with $\partial\mathcal{D}$.[3] The refinement process successively refines boundary facets that are classified as bad facets, e.g., a facet may simply be considered too large. Note that giving a (very small) maximal facet size would already ensure a bound on the one-sided Hausdorff distance of $\mathcal{M}$ to $\mathcal{D}$. However, due to the high precision requirements in our setting, this would result in a uniform highly tessellated mesh with unacceptable complexity.

B. Modification

We start from the voxelization $\mathcal{V}_0$ that was obtained in the sweeping phase. To achieve conservativeness it is not an option to approximate the boundary of $\mathcal{V}_0$, since in convex regions of $\mathcal{V}_0$ the mesh would always intersect $\mathcal{V}_0$. Instead we compute two offsets $\mathcal{V}_1$ and $\mathcal{V}_2$ (one additional layer of voxels each) and set $\mathcal{D} := \mathcal{V}_1$. More precisely, we introduce two new predicates that declare a facet as bad if $f \cap \mathcal{V}_0 \neq \emptyset$ or $f \cap \mathcal{V}_2 \neq f$, respectively. An illustration of the overall process is given in Figure 3.

The process terminates since the mesh converges towards $\partial\mathcal{V}_1$ and since the minimal distance of $\partial\mathcal{V}_1$ to $\partial\mathcal{V}_0$ as well as the minimal distance of $\partial\mathcal{V}_1$ to $\partial\mathcal{V}_2$ is ε. The first predicate obviously ensures conservativeness while the second allows us to bound the Hausdorff distance: Since each point on $\mathcal{M}$ is within $\mathcal{V}_2$ it is at most two voxels away from $\mathcal{V}_0$. This bounds the one-sided Hausdorff distance of $\mathcal{M}$ to $\mathcal{V}_0$ to $2\sqrt{3}\varepsilon$, i.e., two voxel diagonals (see also Figure 4).

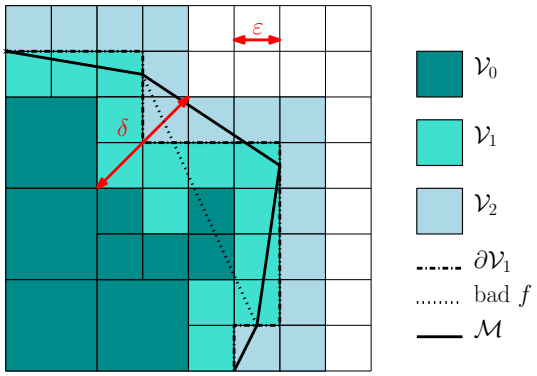

Fig. 4. A two-dimensional illustration of the voxelization $\mathcal{V}_0$ organized as an octree and its two offsets $\mathcal{V}_1$ and $\mathcal{V}_2$. Vertices of the mesh $\mathcal{M}$ are placed on the surface of $\mathcal{V}_1$. δ indicates the maximal distances of a point on $\mathcal{M}$ to $\mathcal{V}_0$.

C. Implementation details

Due to its flexibility, we use the Delaunay refinement package [10] of CGAL[4]. In particular, it was possible to provide our own type for $\mathcal{D}$ and our own set of criteria to classify triangles as bad. The two predicates are actually implemented at once: A facet is voxelized and declared bad if one of its voxels is contained in $\mathcal{V}_0$ or not contained in $\mathcal{V}_2$, respectively.

We remark that the above criteria alone are sufficient to achieve a mesh with the promised guarantees. However, in addition we also kept CGAL's criteria providing the possibility for the user to control the quality of the resulting mesh, i.e., it is possible to set an upper bound on the minimal angle and the size of triangles, respectively.

IV. Results

To a large extent, our work was motivated by the concrete demand of a large German car manufacturer that required a mesh of a swept volume with the above properties for two scenarios which differ in the motion that is performed by the solid, namely:

[3] On the duality of the Delaunay triangulation and the Voronoi diagram see for instance [6].

[4] Computational Geometry Algorithms Library, `www.cgal.org`.

			1-CPU		4-CPU		8-CPU			16-CPU			CUDA		
scenario	n	m	10	11	10	11	10	11	12	10	11	12	10	11	12
bunny	8100	129	25.2	96.8	24.5	99.6	22.5	96.9	434	22.6	95.5	429	20.2	83.7	372
dragon	871k	129	393	788	154	399	92.9	299	1745	83.5	309	1623	137	325	-
engine	328k	191	601	1239	225	562	127	475	3807	127	429	3669	122	405	-
steering	261k	27k	18k	27k	7216	10k	3453	4733	8725	3150	3807	7060	9200	15k	-

TABLE I

TOTAL RUNTIME (IN SECONDS) FOR DIFFERENT SCENARIOS IN DIFFERENT RESOLUTIONS COMPARING GPU WITH MULTI-CPU SOLUTION.

- the trajectory represents an assembly path,
- the trajectory represents a vibration.

For both scenarios the expected complexity n (#triangles) of the generator is rather high: for instance a mesh representing the main engine of a car or truck. In the vibration scenario also the complexity m (#samples) of the discrete trajectory is very high, since its transformations were recorded during a test drive with very small sampling intervals of 5 ms.

Benchmarks for the multi-core version were done on a machine with 16 Intel(R) Xeon(R) CPU X555 with 2.67GHz and 48GB main memory. For the GPU version we used a machine with an Intel Core i7 at 3.20GHz and 12GB RAM, and an NVIDIA GeForce GTX 480 with 1536MB RAM.

Table I shows the runtime of our scenarios for several resolutions, i.e., different depths of the octree. The scenarios *bunny*, *dragon* and *engine* all describe significant motions, whereas the movement of the steering gear is a vibration. The timings given are total, i.e., account for the sweeping and meshing phases. For complex scenarios most of the time is spent on the sweeping phase and the time consumption for meshing can be neglected. For the GPU version, we could not get timings for the scenarios at resolution 12 (other than the bunny) since the machine went out of memory (the compression scheme as discussed in Section II-A.3 is currently not implemented in our CUDA version).

The input complexity of the *bunny* scenario is rather low, hence the overhead for organizing the threads compensates for the parallel computation. The GPU is at its best for this scenario. Since the triangles as well as the edges of the bunny model are equally sized, not too small, and the motion is significant, the voxelization algorithm produces a massive amount of voxels per triangle, thus keeping all threads on the GPU equally busy.

For the *dragon* and the *engine* one can see a nice speedup when using multi-core. However, one can observe a stagnation, which starts at 8 cores. This is caused by the fact that the actual bottleneck is (despite the filtering for redundant voxels as discussed in Section II-B.1) the insertion of voxels into the octree by the master thread. The GPU version greatly improves the pure voxelization algorithm and thus can compete with or even beat the multi-core implementation on 16 CPUs.

The scenario of the *steering gear* is a vibration, which implies that the vast majority of voxels is generated several times. Hence the filtering scheme (Section II-B.1) boosts the performance of the multi-core implementation, since in most rounds only a few voxels actually reach the master thread.

The output complexities for the real world scenarios are comparable to those of the input generators (92k for *steering* at $D = 11$, 279k for $D = 12$, 252k for *engine* at $D = 11$, 1M at $D = 12$).

V. CONCLUSION

We have introduced a framework that combines a sweeping phase, which computes a compressed voxelization of the swept volume with a subsequent meshing phase. The proposed sweeping scheme is independent from the used voxel generation approach and uses a new compression approach that is applied depending on the current memory consumption. The presented meshing scheme is conservative and guarantees a global error bound in terms of the one-sided Hausdorff distance to the provided voxelization.

By our exemplary implementation we have shown that our method lends itself perfectly to parallelization. The introduced concepts are part of the general framework and can be applied to any voxelization strategy. As the main bottleneck is the insertion into the octree, it should be possible to reach similar performance for even more involved voxelization approaches.

As demonstrated in Section IV, the approach is capable to handle very complex industrial data sets. In particular, we are not aware of any approach that could cope with such complex inputs while keeping the above mentioned guarantees. Using CGAL's Delaunay refinement, the resulting mesh is free of self intersections and of high quality (triangle size, minimal angle). Moreover, exploring the allowed tolerance the approach results in a mesh that adapts nicely to the local complexity of the approximated volume, which keeps the complexity of the resulting mesh remarkably low. Thus, the resulting mesh is well suited for further processing in industrial applications.

VI. FURTHER WORK

The currently implemented voxelization approach computes a conservative approximation to a tessellated swept volume boundary as proposed in [5]. This tessellation might not be conservative. Strictly speaking this could lead to occasional areas of non-conservativeness. Although this is not severe for the dense samplings in our applications, it might be for other applications.

However, if the used voxelization approach is conservative we can guarantee that the resulting mesh is conservative for all cases. Alternatively, if the voxelization approach can provide an error bound ε_v, we can easily incorporate this

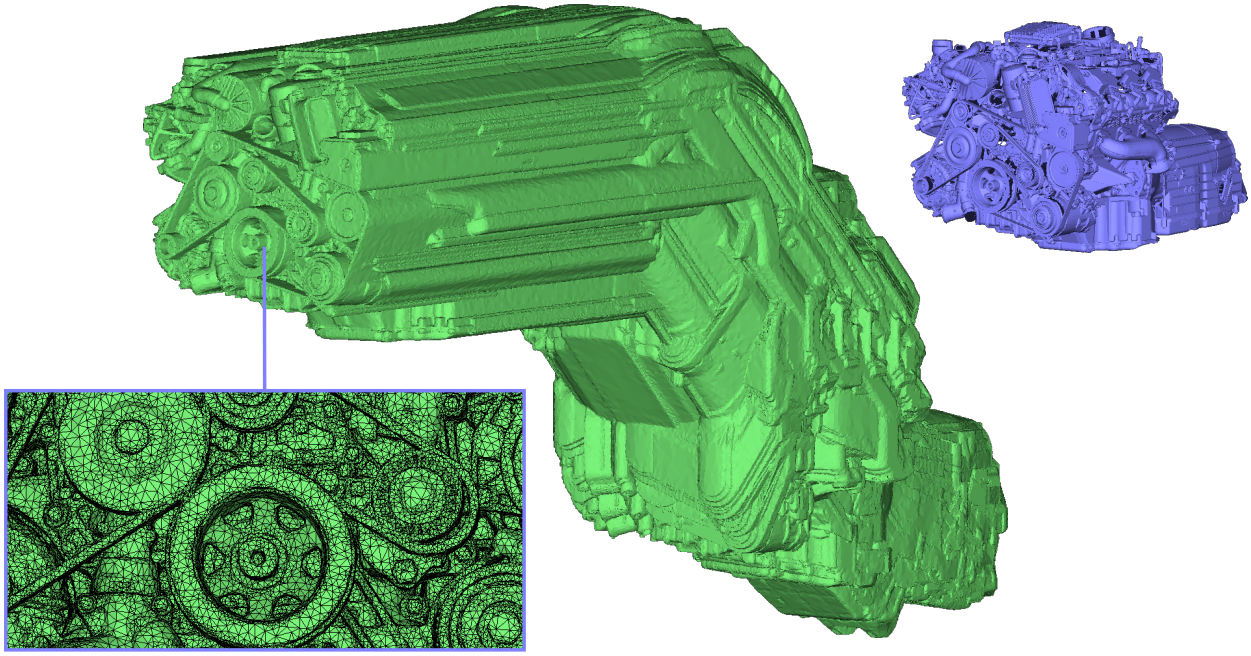

Fig. 5. Engine scenario. Despite the very high precision ($D = 12$), the resulting mesh has only roughly 1M triangles. The magnification shows the very fine details of the generator in its end position. The original component (blue) is a malformed mesh from CAD consisting of ca. 300k triangles.

additional error into our method by simply adding an appropriate amount of additional voxel layers to $\mathcal{V}_0$. Moreover, one can bound the deviation of $\mathcal{M}$ to the actual SV in terms of the one-sided Hausdorff distance. Let n_{add} be the number of additional voxel layers then we can bound the error by

$$h(\mathcal{M}, SV) \leq (2 + n_{add})\sqrt{3}\varepsilon + \varepsilon_v.$$

For previous approaches the computational bottleneck lies in the transformation of triangles into a volumetric representation, e.g. $> 99\%$ for the benchmarks in [14] and $75 - 88\%$ in [8]. These approaches could benefit from our presented framework, since the number of triangles that we can process per second is several orders of magnitude higher. In [14] triangles are inserted into an adaptive distance field with an error bound comparable to $D = 8$ with an average rate of roughly 1000 triangles/second (for $D = 9$ only roughly 10 triangles/second), whereas we can insert triangles into our octree data structure with $D = 12$ at an average rate of up to 10^6 triangles/seconds (steering scenario) on a machine with a CPU clocked as in [14] but 16 cores.

Thus, we plan to add further conservative voxelization algorithms for other swept volume boundary definitions, e.g., similar to the error bounded tessellation of ruled surfaces as used in [8] or even a direct voxelization of the occurring surfaces.

Moreover, we are confident that our method can be extended to approximate the outer boundary of a Minkowski sum of two triangular meshes. Culling criteria for Minkowski sum boundaries that will work well with our voxelization scheme can be found, e.g., in the recent work of Li and McMains [9] and in [5].

References

[1] K. Abdel-Malek, D. Blackmore, and K. Joy. Swept volumes: Foundations, perspectives, and applications. *International Journal of Shape Modeling*, 23(5):1–25, 2004.

[2] S. Abrams and P. K. Allen. Computing swept volumes. *Journal of Visualization and Computer Animation*, 11:69–82, 2000.

[3] T. Akenine-Möller. Fast 3d triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[4] J.-D. Boissonnat and S. Oudot. Provably good sampling and meshing of surfaces. *Graphical Models*, 67:405–451, 2005.

[5] M. Campen and L. Kobbelt. Polygonal boundary evaluation of minkowski sums and swept volumes. In *Eurographics Symposium on Geometry Processing (SGP 2010)*, 2010.

[6] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[7] J. C. Himmelstein, E. Ferre, and J.-P. Laumond. Swept volume approximation of polygon soups. In *ICRA*, pages 4854–4860, 2007.

[8] Y. J. Kim, G. Varadhan, M. C. Lin, and D. Manocha. Fast swept volume approximation of complex polyhedral models. In *SM '03: Proceedings of the eighth ACM symposium on Solid modeling and applications*, pages 11–22, New York, NY, USA, 2003. ACM.

[9] W. Li and S. McMains. A GPU-based voxelization approach to 3D minkowski sum computation. In *SPM '10: Proceedings of the 14th ACM Symposium on Solid and Physical Modeling*, pages 31–40, New York, NY, USA, 2010. ACM.

[10] L. Rineau and M. Yvinec. *3D Surface Mesher*, 3.8 edition, 2011. CGAL User and Reference Manual.

[11] U. Schwanecke and L. Kobbelt. *Approximate envelope reconstruction for moving solids*, pages 455–466. Vanderbilt University, Nashville, TN, USA, 2001.

[12] A. von Dziegielewski, R. Erbes, and E. Schömer. Conservative swept volume boundary approximation. In *SPM '10: Proceedings of the 14th ACM Symposium on Solid and Physical Modeling*, pages 171–176, New York, NY, USA, 2010. ACM.

[13] J. D. Weld and M. C. Leu. Geometric representation of swept volumes with application to polyhedral objects. *Int. J. Rob. Res.*, 9:105–117, September 1990.

[14] X. Zhang, Y. J. Kim, and D. Manocha. Reliable sweeps. In *SPM '09: 2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, pages 373–378, New York, NY, USA, 2009. ACM.