High Precision Swept Volume Approximation with Conservative Error Bounds

Dissertation
zur Erlangung des Grades
"Doktor der Naturwissenschaften"

am Fachbereich Physik, Mathematik und Informatik
der Johannes Gutenberg-Universität in Mainz,

vorgelegt von
Andreas von Dziegielewski
geboren in Mainz

Mainz, den 26.06.2012

Berichterstatter:

D77

Abstract

In technical design processes in the automotive industry, digital prototypes rapidly gain importance, because they allow for a detection of design errors in early development stages. The technical design process includes the computation of swept volumes for maintainability analysis and clearance checks.

The swept volume is very useful, for example, to identify problem areas where a safety distance might not be kept. With the explicit construction of the swept volume an engineer gets evidence on how the shape of components that come too close have to be modified.

In this thesis a concept for the approximation of the outer boundary of a swept volume is developed. For safety reasons, it is essential that the approximation is conservative, i.e., that the swept volume is completely enclosed by the approximation. On the other hand, one wishes to approximate the swept volume as precisely as possible. In this work, we will show, that the one-sided Hausdorff distance is the adequate measure for the error of the approximation, when the intended usage is clearance checks, continuous collision detection and maintainability analysis in CAD.

We present two implementations that apply the concept and generate a manifold triangle mesh that approximates the outer boundary of a swept volume. Both algorithms are two-phased: a sweeping phase which generates a conservative voxelization of the swept volume, and the actual mesh generation which is based on restricted Delaunay refinement. This approach ensures a high precision of the approximation while respecting conservativeness.

The benchmarks for our test are amongst others real world scenarios that come from the automotive industry.

Further, we introduce a method to relate parts of an already computed swept volume boundary to those triangles of the generator, that come closest during the sweep. We use this to verify as well as to colorize meshes resulting from our implementations.

Zusammenfassung

Bei technischen Designprozessen in der Automobilindustrie kommen vermehrt digitale Prototypen zum Einsatz um anhand von CAD-Modellen frühzeitig Fehler zu erkennen. Darunter fallen auch Bauraum- und Wartbarkeitsuntersuchungen durch die Berechnung des überstrichenen Volumens (Swept Volumes).

Das Swept Volume ist beispielsweise sehr hilfreich, um Problemzonen zu identifizieren, in denen eine Unterschreitung von einzuhaltenden Sicherheitsabständen droht. Durch die explizite Konstruktion des Swept Volumes erhält der Ingenieur klare Hinweise darauf, wie die Form von Bauteilen, die sich zu nahe kommen, zu modifizieren ist.

In dieser Arbeit wird zunächst ein Konzept erarbeitet, das der Approximation des äußeren Randes eines von einem Generator überstrichenen Volumens dient. Es ist hierbei aufgrund von Sicherheitsaspekten unerlässlich, dass die Approximation konservativ ist, das eigentliche Volumen also komplett umschlossen wird. Auf der anderen Seite möchte man den maximalen Fehler der Approximation möglichst klein halten, also eine möglichst hohe Präzision erreichen. Es wird in dieser Arbeit gezeigt, dass der einseitige Hausdorff-Abstand ein sinnvolles Maß für den Fehler der Approximation des Swept Volumes ist, wenn man diese für die Überprüfung von Sicherheitsabständen oder Wartbarkeitsuntersuchungen verwenden möchte.

Es folgen zwei Implementierungen, die unter Anwendung des Konzeptes Approximationen des äußeren Randes des Swept Volumes in Form von Dreiecksnetzen berechnen.

Beide Implementierungen sind in zwei Phasen unterteilt. In einer ersten Phase wird jeweils eine konservative Voxelisierung des überstrichenen Volumens erzeugt, gefolgt von einer zweiten Phase, in der das approximierende Delaunay-Dreiecks-Mesh erzeugt wird. Dieses Vorgehen sichert unter Beachtung der Konservativität eine hohe Präzision der resultierenden Approximation.

Als Benchmarks für die Implementierungen dienen reale Beispiel-Szenarien aus der Automobilindustrie.

Abschließend wird ein Verfahren präsentiert, mit dem sich herausfinden lässt, welche Teile des Generators für bestimme Bereiche einer Swept Volume Approximation verantwortlich sind. Somit ist man in der Lage, Teile des approximierenden Dreiecksnetzes entsprechend den Farben der verantwortlichen Generatorteile einzufärben. Auch kann man dieses Verfahren zur Verifikation bereits berechneter Swept Volume Approximationen verwenden.

Acknowledgement

I most cordially thank my advisor for his guidance and the strong support throughout these studies.

Further, I want to thank the second reader of this thesis.

I would like to express my sincere thanks to the co-authors of the publications accompanying this thesisfor a valuable and pleasant cooperation.

Further, I would like to thank all my colleagues from the Johannes Gutenberg-Universität Mainz for the kind, personable and congenial working atmosphere.

Last but not least, my deepest gratitude goes to my wife and my family for their strong support, that made this work possible in the first place.

Contents

Chapter 1

Introduction

The swept volume (SV) is the entity of all points touched by a solid (the generator) under the transformations of either a continuous or discrete trajectory. It plays an important role in NC machining verification, robot analysis and graphical modeling. This work was, to a large extend, motivated by the concrete demand of a large German car manufacturer where the swept volume is used for clearance checks, continuous collision detection and maintainability analysis. For instance, the swept volume may be used to verify that an already computed assembly path remains valid while other parts of the design may change.

Due to increasing model complexity and high demands concerning error tolerance, algorithms in the fields of computer aided design (CAD) and digital mock up (DMU) have to process masses of data. Since the input models often lack topological qualities, a practical algorithm has to be stable and preferably imposes no restrictions on the input models. For safety reasons, it is furthermore important that an algorithm never underestimates the actual SV, i.e., the approximation has to be conservative.

Lastly, since the output mesh that approximates the swept volume boundary is intended for further usage in CAD or for subsequent distance computations, it is highly desirable that the mesh is of high quality, i.e., it should be closed, not have acute angled triangles and most important, the complexity (triangle count) should be as low as possible.

In most cases the generator (object to be swept) is given as a (often malformed) triangle mesh and the motion is given by a discrete trajectory, i.e., a sequence of rigid body transformations $(\mathcal{R}_i)_{i=1,\dots,m}$. One can think of the latter as a

densely sampled continuous motion. For example we were supplied with a set of vibrations from an engine of a car maneuvering through a test field. The position of the engine was sampled every 5 milliseconds.

1.1 Motivation

In late 2007, we were posed the following problem by a car manufacturer: To measure the movement of an engine during test drives, nine displacement sensor pairs were placed onto the motor compartment of a car as depicted in Figure 1.1. Every sensor was attached to one point on the engine ($a_i \in \mathbb{R}^3$) as well as to one

Figure 1.1: Sensor installation mounted on chassis to record movement of engine.

point on a metal frame that was installed on the carriage ($b_i \in \mathbb{R}^3$), cf. Figure 1.2. The sensors recorded the distances ($l_i \in \mathbb{R}$) between these two points every 5 milliseconds (for $i = 0, \ldots, 8$). Given the positions a_i, b_i, $i = 0, \ldots, 8$, as well as l_i at m time steps, we were asked to compute the actual transformations $(\mathcal{R}_i)_{i=1,\ldots,m}$ of the engine, specifically, the sequence of rigid transformations relative to the chassis.

Figure 1.2: Close up image of the displacement sensors fixated on chassis (red) and on the engine (blue).

Theoretically speaking, this problem can be described as the forward kinematics of a (redundant) Stewart Gough platform, with the peculiarity that all above data is afflicted with some measuring error.

We found an appropriate formulation of the problem that can cope with inexact data in the form of a non-linear optimization problem:

$$\min_{\mathcal{R}} \sum_{i=0}^{8} \left(\|\mathcal{R}(a_i) - b_i\| - l_i \right)^2, \tag{1.1}$$

or equivalently:

$$\min_{A \in SO(3),\, c \in \mathbb{R}^3} \sum_{i=0}^{8} \left(\|(Aa_i + c) - b_i\| - l_i \right)^2. \tag{1.2}$$

Due to the very small sampling interval of 5 ms, the engine does not move too much in between consecutive time steps. Therefore, the solution of the previous time step $\mathcal{R}_{i-1}$ is a perfect starting point for a local optimization algorithm

for the i-th time step. For this local optimization, we chose the Downhill Simplex Algorithm as proposed by Nelder and Mead [31], because it is very robust and treats every dimension of the search domain independently[1]. The latter is particularly important as we parameterized $SO(3)$ with unit quaternions.

We implemented the above method and shipped it to the manufacturer in 2008.

The reason for these test drives was to examine clearance checks between parts of the engine and other components that were fixated on the chassis, e.g., the oil pan, and the neighboring components, cf. Figure 1.3.

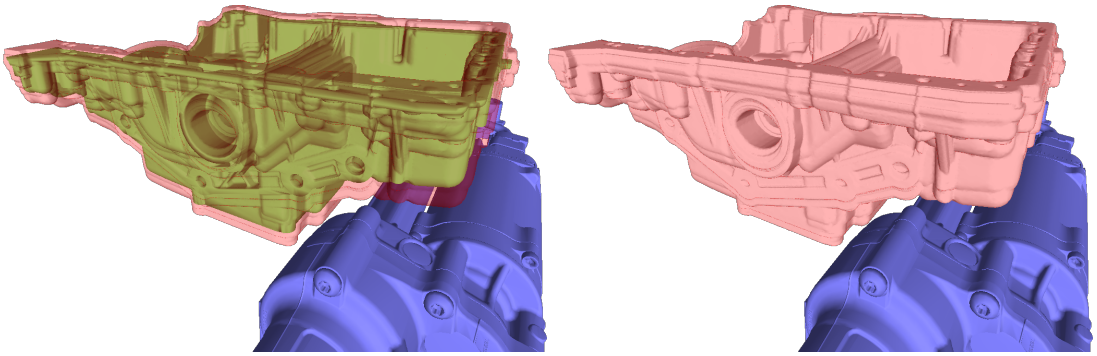

Figure 1.3: Swept volume boundary (red) of the bottom part of the engine (oil pan, green) under the transformations of the recorded vibration to verify that a safety distance to the neighboring component (blue) is kept.

A subsequent question to the problem above, given a model in CAD and a computed trajectory, is: Which volume in space does the model occupy under all motions of the trajectory? This question is the motivation of the following thesis. We present methods that approximatively compute the outer boundary of this volume, that is also called the swept volume of the generator model under the sweep-trajectory.

The trajectory can, of course, also come from CAD. The scenario depicted in Figure 1.4 shows a user-created path during maintainability analysis.

[1]This especially implies that the algorithm works scale-invariant for every dimension.

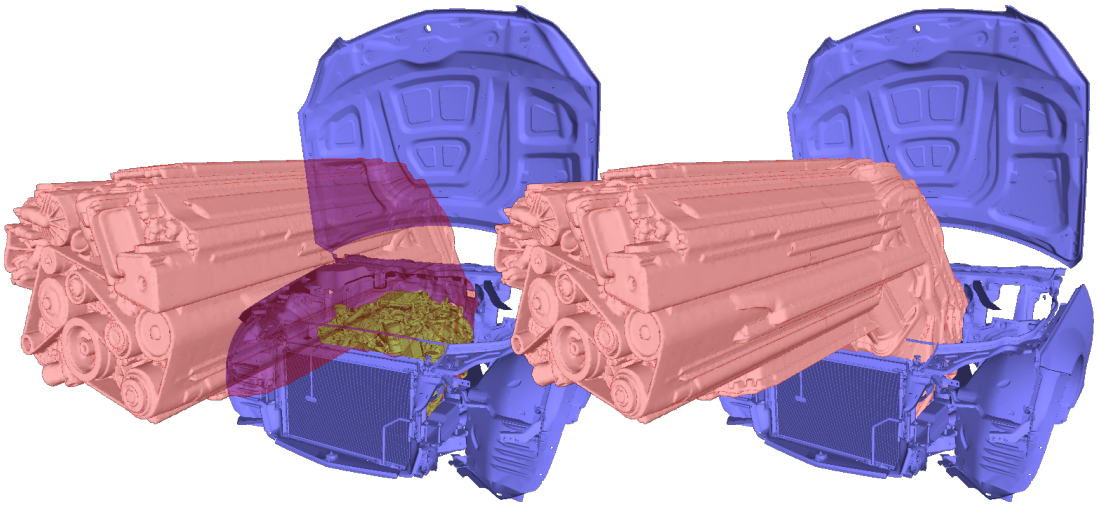

Figure 1.4: Example of a CAD user generated swept volume for maintainability analysis. Left: transparent swept volume boundary (red) with engine (green) and chassis (blue). Right: non-transparent swept volume boundary (red) and chassis (blue).

The swept volume (or rather its outer boundary) can be used to verify that the path is collision free and that spatial tolerances are not violated.

1.2 Previous Work

Mathematical formulations describing the swept volume includes: Jacobian rank deficiency methods (singularity theory) [3, 2], sweep differential equations [7, 18] and envelope theory [35, 51]. For a survey see [1].

For the special case of sweeping a polyhedral object, Weld and Leu [48] have shown that it suffices to compute the swept volume of the polygonal faces which leads to ruled and developable surfaces, arising from sweeping polyhedral surface primitives on a parametric trajectory and polygons of the generator itself. Polyhedral approximations of these surfaces were proposed by Abrams and Allen [4]. The authors compute the arrangement of these tessellated surfaces and then extract the outer boundary using the gift-wrapping algorithm. A similar approach

was recently presented by Campen and Kobbelt [12] and extended to Minkowski sums. Similar to [4], they apply local culling criteria to rule out triangles that will not contribute to the outer boundary of the swept volume approximation and propose the use of a local BSP trees to circumvent robustness issues when intersecting remaining triangles. Since these approaches aim to compute the exact boundary of the polygonal SV approximation, the output can become very complex. The authors in [12] are able to produce a conservative approximation, however their output mesh is non-adaptive and possibly highly over-tessellated, and further simplification in convex regions would violate conservativeness.

Numerical approximation of swept volume boundaries utilizing implicit modeling was introduced in [40]. The minimal distance to the swept generator is stored using a signed distance field, from which an approximation to the SV can be extracted as an isosurface.

In [27, 20] this implicit approach is extended to computing a directed distance field relative to the tessellated ruled and developable surfaces, using the graphics processing unit (GPU). The authors compute this directed distance field utilizing the GPU and extract the SV via the extended marching cubes algorithm [28]. However, the massive memory demand for storing a full volumetric distance field limits the geometric accuracy of the approximation. Further, as they rely on GPU rasterization, they cannot give geometrical guarantees because sharp features can be missed.

In a recent work [53], the authors propose a reduced memory consuming adaptive signed distance field. The authors claim to be able to achieve arbitrarily tight error-bounds and that the output mesh has the same topology as the exact SV. They give geometric bounds in terms of the (two-sided) Hausdorff-distance (although they do not regard conservativeness). However, since our output meshes serve for clearance and distance checks, the one-sided Hausdorff-distance is the appropriate (less restrictive) choice. Unfortunately, they do not give results for a priori error bounds.

Using a hierarchical spatial structure to store a volumetric approximation of the swept volume, was also presented by Schwanecke and Kobbelt [41]. The authors dynamically update an octree while sweeping the generator in an implicit representation. Unfortunately their implementation is restricted to spheres.

Another approach for reducing memory consumption is to store only 2D images of the sweep and reconstruct an approximation of the SV from them. These image based SV-approximations, that often utilize the depth buffer, were first introduced in [21]. They were used for displaying purposes [25, 24, 50] as well as for 3D mesh extraction [5]. In [5] the authors merge depth buffer images from arbitrarily positioned cameras to generate a point cloud of the swept volume boundary from which a mesh can be extracted.

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

The following thesis is based on work we published and presented at the 26th and 27th European Workshop on Computational Geometry (EuroCG) in 2010 and 2011 [46, 45], at the Symposium of Solid and Physical Modeling (SPM) in 2010 [44] and at the IEEE International Conference on Robotics and Automation (ICRA) in 2012 [47].

Chapter 2

A Concept of Error Bounded Approximations of Swept Volumes

In the following chapter we will present a concept to generate a closed surface approximating the outer boundary of a swept volume with appropriate error bounds.

We assume an object $\mathcal{G}$ (generator) travels in space along a path $(\mathcal{R}_t)_{t\in[0,1]}$ (trajectory) and passes one or more obstacles Ω_i. For now, let us assume that $\mathcal{G}, \Omega_i \subset \mathbb{R}^3$ are compact sets and that $(\mathcal{R}_t)_{t\in[0,1]}$ is a continuous one-parameter rigid body motion:

$$\begin{aligned} \mathcal{R}_t(x) =& A(t)x + c(t), \\ A : [0,1] &\longrightarrow SO(3), \\ c : [0,1] &\longrightarrow \mathbb{R}^3, \end{aligned}$$

where $A(t)$ as well as $c(t)$ are continuous.

The questions are:

- Does the generator collide with obstacle Ω_i?
- In the case of no collision, how close (Euclidean distance) does the generator get to Ω_i?

To answer these questions one can compute the minimal distance between generator and each obstacles during the path

$$\min_{t\in[0,1]} \min_{x\in\mathcal{G}} d(\mathcal{R}_t(x), \Omega_i) = \min_{t\in[0,1]} d(\mathcal{R}_t(\mathcal{G}), \Omega_i),$$

where $\mathcal{R}_t(\mathcal{G}) = \bigcup_{x\in\mathcal{G}} \mathcal{R}_t(x)$.

A different approach is to compute the swept volume generated by $\mathcal{G}$ during the movement. The swept volume is defined as the set of points touched by the generator under the transformations of the trajectory:

$$SV = \left\{ y \in \mathbb{R}^3 \,\middle|\, \exists\, x \in \mathcal{G}, \exists\, t \in [0,1], y = \mathcal{R}_t(x) \right\}.$$

Having determined the swept volume, the minimal distance problem boils down to computing the distance between the swept volume (SV) and the obstacle

$$\min_{t\in[0,1]} d(\mathcal{R}_t(\mathcal{G}), \Omega_i) = d(SV, \Omega_i).$$

We note that the second approach needs only a static distance computation per obstacle, in contrast to the continuous minimal distance computation problem in the first approach, but of course requires a prior swept volume computation. Clearly the second approach is especially useful when obstacle geometries or positions are expected to change frequently, since the swept volume once computed can be reused.

As determining the exact swept volume is too costly in most our applications, we aim for the generation of an error bounded approximation that we can use for all following computations.[1] Furthermore, for our intended usage of swept volumes in collision checking and distance queries, internal voids of the swept volume are of no interest. We therefore wish to approximate the outer boundary of the SV with a closed surface $\mathcal{M}$.

For safety reasons, it is important that we still can reliably decide whether collision occurred. Therefore, we demand that the approximation is conservative, i.e., the approximation $\mathcal{M}$ must not intersect SV. Second, we want to bound

[1] This approach is even more encouraged by the fact, that in our target applications the generators and obstacles as well as the trajectories stem from CAD data and are already afflicted with some error.

the deviation of the minimal distance between the generator and obstacles from the distance between $\mathcal{M}$ and the obstacles a priori:

$$d(\mathcal{M}, \Omega_i) \leq \min_{t\in[0,1]} d(\mathcal{R}_t(\mathcal{G}), \Omega_i) = d(SV, \Omega_i) \leq d(\mathcal{M}, \Omega_i) + \varepsilon. \tag{2.1}$$

The first inequality is a direct result of the conservativeness demand. The second inequality can be achieved by bounding the deviation between SV and the approximation $\mathcal{M}$ in terms of the directed (or one-sided) Hausdorff distance. The **directed Hausdorff distance** (from $\mathcal{A}$ to $\mathcal{B}$) is the maximum Euclidean distance from points in $\mathcal{A}$ to the set $\mathcal{B}$

$$h(\mathcal{A}, \mathcal{B}) = \max_{a\in\mathcal{A}} d(a, \mathcal{B}) = \max_{a\in\mathcal{A}} \min_{b\in\mathcal{B}} d(a, b).$$

Lemma 1. *Assume $h(\mathcal{M}, SV) \leq \varepsilon$, then the second inequality in (2.1) holds.*

Proof. Let $h(\mathcal{M}, SV) \leq \varepsilon$ and let $m_i \in \mathcal{M}$ being closest to Ω_i, i.e., $d(\mathcal{M}, \Omega_i) = d(m_i, \Omega_i)$. Then with the triangle inequality it holds

$$\begin{aligned} d(SV, \Omega_i) &\leq d(m_i, \Omega_i) + d(m_i, SV) \\ &\leq d(\mathcal{M}, \Omega_i) + \max_{a\in\mathcal{M}} \min_{b\in SV} d(a, b) \\ &\leq d(\mathcal{M}, \Omega_i) + \varepsilon. \end{aligned}$$

□

Hence, as we stated in [44], when approximating the outer swept volume boundary for maintainability analysis and clearance checks, it is desirable that the approximating output surface $\mathcal{M}$ meets two important demands:

1. Conservativeness: The approximation must be conservative, i.e., the swept volume has to be completely enclosed by the output surface:

   $$\mathcal{M} \cap SV = \emptyset.$$

2. Precision: The overestimation of the swept volume boundary is bounded in terms of the one-sided Hausdorff distance

   $$h(\mathcal{M}, SV) \leq \varepsilon,$$

   with ε being an a priori user defined tolerance.

Note that the demand of conservativeness ensures the first inequality in (2.1).

To achieve the two above demands, we proposed in [44] the definition of a volume that forms a layer coating the outer boundary of the SV. We then carefully extracted a triangle mesh ensuring that it completely lies inside this volume. In contrast to our approach, the volumes defined in [52, 10] do not regard conservativeness. Further, they use unnecessary involved data structures [10] and error metrics that are too restrictive for our purpose [52].

In Section 2.1 we will recapitulate this method and develop a generalized concept. We will show that this volume, called **tolerance hull** in the following section, meets the above demands. We then, in Section 2.2 develop a method to generate a mesh that lies completely inside the tolerance hull, thus meeting the demands that it conservatively approximates the underlying set and that its precision (in terms of the one-sided Hausdorff distance) can be controlled a priori by the user. Building up the mesh is based on restricted Delaunay mesh generation.

Finally, in Section 2.3, we apply the concepts of the tolerance hull and the mesh extraction to our purpose: swept volumes.

2.1 Tolerance Hull

Since the concept presented in this chapter can be used for applications beyond swept volume computation[2], in this section we consider a general compact set $\mathcal{A} \subset \mathbb{R}^3$ with an outer boundary $\overline{\partial}\mathcal{A}$ that we want to approximate with a closed surface $\mathcal{M}$. In the special case of swept volume approximation, we set $\mathcal{A} = SV$. The outer boundary of a compact set $\mathcal{A}$ is the part of the boundary $\overline{\partial}\mathcal{A} \subset \partial\mathcal{A}$ that is reachable from infinity without intersecting $\mathcal{A}$.

We define the (δ-)**offset** $\mathcal{O}_\delta(\mathcal{A})$ of $\mathcal{A}$ to be the Minkowski sum of $\mathcal{A}$ and the solid ball $\mathcal{B}_\delta(0)$ centered at the origin

$$\mathcal{O}_\delta(\mathcal{A}) = \mathcal{A} \oplus \mathcal{B}_\delta(0) = \left\{ x \in \mathbb{R}^3 \,\middle|\, \|x - a\|_2 \leq \delta, a \in \mathcal{A} \right\},$$

with its outer boundary $\overline{\partial}\mathcal{O}_\delta(\mathcal{A})$. We call δ **offset radius**.

[2]For example, we have successfully applied the concept to conservative error bounded remeshing of prohibitively complex or topologically malformed triangle meshes.

We will make use of the following result on the additivity of offsets.

Lemma 2. *For $\mathcal{A} \subset \mathbb{R}^3$ and $\delta, \varepsilon > 0$ it holds that*

$$\mathcal{O}_\delta(\mathcal{O}_\varepsilon(\mathcal{A})) = \mathcal{O}_{\delta+\varepsilon}(\mathcal{A}).$$

Proof. Due to the associativity of the Minkowski sum we have

$$\begin{aligned} \mathcal{O}_\delta(\mathcal{O}_\varepsilon(\mathcal{A})) &= (\mathcal{A} \oplus \mathcal{B}_\delta(0)) \oplus \mathcal{B}_\varepsilon(0) \\ &= \mathcal{A} \oplus (\mathcal{B}_\delta(0) \oplus \mathcal{B}_\varepsilon(0)) \\ &= \mathcal{O}_{\delta+\varepsilon}(\mathcal{A}), \end{aligned}$$

where we used $\mathcal{B}_\delta(0) \oplus \mathcal{B}_\varepsilon(0) = \mathcal{B}_{\delta+\varepsilon}(0)$. □

Given a set $\mathcal{A}$, we want to define a volume around $\mathcal{A}$ that neither deviates too far from, nor intersects it. Later, this will be the volume that our output mesh $\mathcal{M}$ must lie within.

Let $\mathcal{A} \subset \mathbb{R}^3$ be a compact set and let $\delta > \varepsilon > 0$. We define a **(δ-ε-)tolerance hull** $\mathcal{H}_{\varepsilon,\delta}(\mathcal{A})$ of $\mathcal{A}$ to be the set

$$\mathcal{H}_{\varepsilon,\delta}(\mathcal{A}) = \mathcal{O}_\varepsilon(\overline{\partial}\mathcal{O}_\delta(\mathcal{A})). \tag{2.2}$$

Before we can show that the tolerance hull satisfies the demands (conservativeness and boundedness) stated above, we need the following rather intuitive result

Lemma 3. *Let $x \in \mathbb{R}^3$, $\mathcal{A} \subset \mathbb{R}^3$ compact, $\delta_1 > \delta_2 > 0$ and $d(x, \mathcal{A}) \geq \delta_1$. Then it holds that*

$$d(x, \mathcal{O}_{\delta_2}(\mathcal{A})) \geq \delta_1 - \delta_2.$$

Proof. Let $a \in \mathcal{O}_{\delta_2}(\mathcal{A})$, then there exist $a_1 \in \mathcal{A}$, $a_2 \in \mathcal{B}_{\delta_2}(0)$ such that $a = a_1 + a_2$. It holds

$$d(x, \mathcal{O}_{\delta_2}(\mathcal{A})) = \|x - a\|_2 = \|x - a_1 - a_2\|_2 \geq \big|\|x - a_1\|_2 - \|a_2\|_2\big| \geq \delta_1 - \delta_2.$$

□

This enables us to state the main result of this section:

Theorem 1. *Let $\mathcal{A} \subset \mathbb{R}^3$ be a compact set and $\mathcal{M} \subset \mathcal{H}_{\varepsilon,\delta}(\mathcal{A})$, with $\delta > \varepsilon > 0$. The following assertions hold:*

1. *$\mathcal{M}$ does not intersect the set itself,*

$$\mathcal{M} \cap \mathcal{A} = \emptyset.$$

2. *No point in $\mathcal{M}$ deviates from the set more than $\varepsilon + \delta$,*

$$h(\mathcal{M}, \mathcal{A}) \leq \varepsilon + \delta.$$

Proof. 1. Let $a \in \mathcal{A}$, then it holds $d(a, \overline{\partial}\mathcal{O}_\delta(\mathcal{A})) \geq \delta$. Since $\overline{\partial}\mathcal{O}_\delta(\mathcal{A})$ itself is compact, we can apply Lemma 3:

$$d(a, \underbrace{\mathcal{O}_\varepsilon(\overline{\partial}\mathcal{O}_\delta(\mathcal{A}))}_{=\mathcal{H}_{\varepsilon,\delta}(\mathcal{A})}) \geq \delta - \varepsilon > 0.$$

As $\mathcal{M} \subset \mathcal{H}_{\varepsilon,\delta}(\mathcal{A})$ and because we posed no restrictions on a, it holds

$$\mathcal{M} \cap \mathcal{A} = \emptyset.$$

2. Let $m \in \mathcal{M}$. Lemma 2 yields

$$\mathcal{M} \subset \mathcal{H}_{\varepsilon,\delta}(\mathcal{A}) \subset \mathcal{O}_\varepsilon(\mathcal{O}_\delta(\mathcal{A})) = \mathcal{O}_{\varepsilon+\delta}(\mathcal{A}),$$

hence it exists $a \in \mathcal{A}$ such that $m \in \mathcal{B}_{\delta+\varepsilon}(a)$ and $d(m, \mathcal{A}) \leq d(m, a) \leq \varepsilon+\delta$. Because we did not impose any restrictions on the selection of m other than $m \in \mathcal{M}$, it holds that

$$h(\mathcal{M}, \mathcal{A}) \leq \varepsilon + \delta.$$

□

Theorem 1 states that once we have computed the tolerance hull $\mathcal{H}_{\varepsilon,\delta}(\mathcal{A})$ of $\mathcal{A}$ and we approximate the outer boundary $\overline{\partial}\mathcal{A}$ with a closed surface $\mathcal{M} \subset \mathcal{H}_{\varepsilon,\delta}(\mathcal{A})$, this surface meets the conservative and boundedness conditions from Equation (2.1) with $\varepsilon := \varepsilon + \delta$.

If $\mathcal{A}$ is connected, then it is guaranteed that it is completely encased by the closed surface $\mathcal{M}$ and Equation (2.1) can be used for clearance checks. Figure 2.1 illustrates the concepts presented in this section.

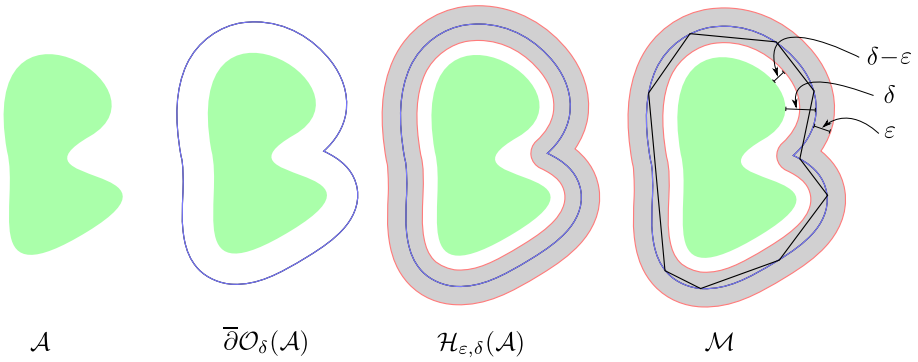

Figure 2.1: Concept of the tolerance hull.

2.2 Restricted Delaunay Mesh Generation

It is convenient to represent the output surface $\mathcal{M}$, from the previous section, that approximates the outer boundary of the SV, in form of a closed triangle mesh.

We presented in [46] a method to extract a mesh from a tolerance hull using iterative triangle refinement. The implementation of this method will be presented in Chapter 4. In this section we develop a generalized form, suitable to the abstract concept of the tolerance hull.

More precisely, we propose to use the Delaunay refinement [8] algorithm with some modifications so that the final mesh $\mathcal{M}$ is guaranteed to be contained in a given tolerance hull. This will allow us to ensure conservativeness and a bound on the one-sided Hausdorff distance while keeping the complexity of $\mathcal{M}$ low. We first give a brief review of Delaunay refinement.

For a given compact domain $\mathcal{D} \subset \mathbb{R}^3$, the Delaunay refinement algorithm produces a mesh $\mathcal{M}$ that approximates the boundary of that domain, $\partial\mathcal{D}$. Starting from an initial point set E on $\partial\mathcal{D}$, the process maintains a Delaunay triangulation $\mathrm{Del}(E)$ of this point set.[3] This is a 3D complex, i.e., a set of faces with dimension 0 (vertices), 1 (edges), 2 (facets/triangles) and 3 (cells/tetrahedra), such that all faces are pairwise interior disjoint, and the boundary of each face

[3]Strictly speaking $\mathrm{Del}(E)$ is a Delaunay tetrahedralization.

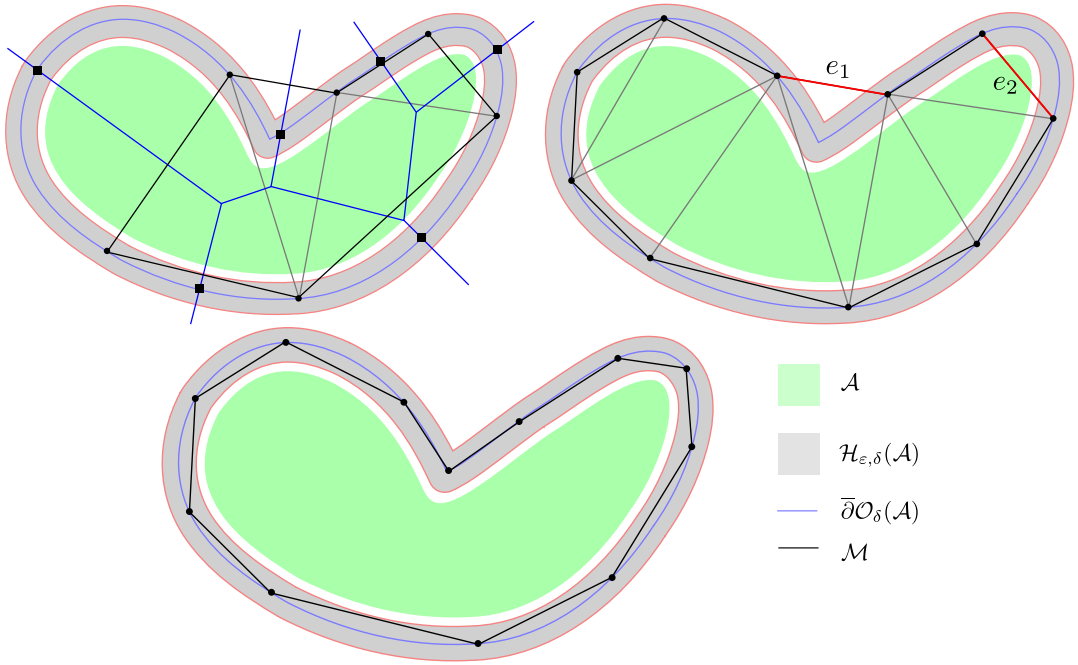

Figure 2.2: Illustration of Delaunay refinement for our approach in 2D. Top left: A possible initial state. An initial set of sample points (on $\overline{\partial}\mathcal{O}_\delta(\mathcal{A})$) induces a Delaunay triangulation and the corresponding Voronoi diagram. Possible refinement points are indicated as black squares. Top right: e_1 and e_2 are scheduled for refinement since they leave $\mathcal{H}_{\varepsilon,\delta}(\mathcal{A})$. Bottom: A final mesh, all boundary edges (triangles in 3D) are inside the hull.

of the complex is the union of faces of the complex. Thereby, the algorithm classifies each tetrahedron as interior or exterior according to the position of the center of its circumscribing ball.

The mesh $\mathcal{M}$ is now defined as the Delaunay triangulation of E restricted to $\partial\mathcal{D}$

$$\mathcal{M} := \mathrm{Del}_{|_{\partial\mathcal{D}}}(E),$$

that is, the subcomplex of $\mathrm{Del}(E)$ containing those facets of $\mathrm{Del}(E)$ whose dual Voronoi edges intersect $\partial\mathcal{D}$, or equivalently, a facet belongs to the mesh if the classification of the two neighboring tetrahedra differ. [8]

Such a boundary surface facet f can be refined by inducing a new point, namely an intersection point of its Voronoi edge $f^\star$ (the dual of f) with $\partial\mathcal{D}$. [4]

We use this Delaunay refinement algorithm to sample points on the outer boundary of the *delta*-offset of $\mathcal{A}$, thus we set

$$\mathcal{D} = \overline{\partial}\mathcal{O}_\delta(\mathcal{A}).$$

The refinement process successively refines boundary facets that are classified as bad facets, e.g., triangles that are considered too large or whose minimal angle is too small. The latter criterion ensures well formed triangles in the resulting mesh. Note that, giving a maximal facet size would already ensure a bound on the one-sided Hausdorff distance of $\mathcal{M}$ to $\mathcal{A}$. However, due to the high precision requirements in our setting, this would result in a uniform highly tessellated mesh with unacceptable complexity.

We therefore introduce a predicate that indicates a facet as being bad if

$$f \cap \mathcal{H}_{\varepsilon,\delta}(\mathcal{A}) \neq f. \tag{2.3}$$

This allows us to relax or even drop the bound on the maximum facet size. With this bad facet criterion, a resulting Delaunay triangle mesh is obviously guaranteed to be inside the tolerance hull. To increase triangle quality, further criteria (e.g., a bound on maximum angle or triangle size) can be added.

An illustration of the overall process is given in Figure 2.2.

[4] On the duality of the Delaunay triangulation and the Voronoi diagram see for instance [15].

The Delaunay refinement algorithm itself is described in [32, 8], we present it here in a simplified version.

Starting from a small initial point sample E of $\partial\mathcal{D}$, the restricted Delaunay triangulation $\mathrm{Del}_{|\partial\mathcal{D}}(E)$ is computed. At each iteration, the point of intersection of $\partial\mathcal{D}$ with a Voronoi edge of a bad facet is inserted into E. The triangulation $\mathrm{Del}(E)$, the restricted Delaunay triangulation $\mathrm{Del}_{|\partial\mathcal{D}}(E)$, as well as the Voronoi diagram are updated after each insertion.

Given E, $\mathrm{Del}(E)$, $\mathrm{Del}_{|\partial\mathcal{D}}(E)$ and a list L storing the bad facets, the algorithm works as shown in Algorithm 1.

Algorithm 1 Delaunay Refinement

while there exists a bad facet $f \in L$ **do**
  refine f by computing the intersection p of $f^\star$ with $\partial\mathcal{D}$;
  add p to E, remove f from L;
  update $\mathrm{Del}(E)$;
  update $\mathrm{Del}_{|\partial\mathcal{D}}(E)$ w.r.t. the local modifications of the Voronoi diagram:
    delete all facets whose dual Voronoi edge no longer intersects $\partial\mathcal{D}$;
    add every newly created facet whose dual Voronoi edge intersects $\partial\mathcal{D}$;
  update L by deleting the facets that are no longer in $\mathrm{Del}_{|\partial\mathcal{D}}(E)$;
    add every newly created facet to L that is a bad facet.
end while

2.2.1 Correctness and Termination

We will later approximate $\overline{\partial}\mathcal{O}_\delta(\mathcal{A})$ with the non-smooth surface of a conservative voxelization, i.e., a polygonal surface where every pair of non-disjoint triangles have a dihedral angle of either $0°$ or $90°$.

We note that even the surface that we approximate by the voxelization boundary, the surface $\overline{\partial}\mathcal{O}_\delta(\mathcal{A})$ is not necessarily in $\mathcal{C}^1$ for non-convex $\mathcal{A}$.

The proofs of correctness and termination of the restricted Delaunay refinement algorithm are shown for $\mathcal{C}^2$-surfaces in [8], and for a special subset of Lipschitz-surfaces in [9]. For the case of re-meshing a polygonal surface, correctness and

termination is shown under the assumption that the dihedral angle of every pair of non-disjoint triangles is $< 10.90°$. [32, § 4.5]

However, experimental results have shown that this boundary is pessimistic, since the algorithm works well for dihedral angles up to 90°, i.e., the algorithm outputs a manifold mesh that reproduces the topology. [32, § 7.4]

We will now give proof that the additional bad facet criterion (2.3) does not hinder the algorithm from terminating.

Theorem 1. *The additional bad facet criterion (2.3) only holds for a finite number of facets generated by the restricted Delaunay refinement algorithm.*

Proof. Let $v_0, \ldots, v_{n-1}$ be the vertices of $\mathcal{M}$. Since all inserted vertices lie on $\overline{\partial}\mathcal{O}_\delta(\mathcal{A})$, a facet f that is bad in terms of (2.3) must have at least one edge with length $> \varepsilon$. Since the newly inserted vertex v_n lies on the intersection of the Voronoi edge of f with $\overline{\partial}\mathcal{O}_\delta(\mathcal{A})$, it holds $\|v_n - v_i\|_2 > \frac{\varepsilon}{2}$, for all $i = 0, \ldots, n-1$. Hence the open balls $\mathring{\mathcal{B}}_{\varepsilon/4}(v_i)$ are pairwise disjoint for all i. Since $\overline{\partial}\mathcal{O}_\delta(\mathcal{A})$ is bounded, there can only exist a finite number of such balls. We conclude that criterion (2.3) can only declare a finite number of facets as bad. □

2.3 Application to Swept Volumes

In this section, we will apply the concept of the tolerance hull to the special case of sweeping a polyhedral generator along a discrete trajectory. Further we will discuss volumetric representations for error bounded approximations of the offset as well as of the tolerance hull while bearing in mind the high precision demands posed by our real world scenarios.

The trajectories in our applications are given in form of a sequence of rigid body transformations that arose from sampling a continuous and smooth motion. We mainly regard two different kind of real world scenarios:

- The vibration trajectories resemble the actual motions of moving engine parts and were recorded during a real test drive in temporal sampling interval of 5 ms, cf., Figure 1.3.

- The maintainability paths describe significant motions and are usually generated using CAD software, cf., Figure 1.4. Thus, it is up to the user to control the sampling density of these motions.

The generator meshes stem from CAD data and often run through simplification processes, and are usually topologically malformed. Thus we do not pose any demands on the generator mesh. The methods proposed work with any triangle soup.

We define a **trajectory** to be a sequence of m rigid body transformations $(\mathcal{R}_i)_{i=1,\dots,m}$. We want to sweep a **generator** $\mathcal{G}$, a triangle mesh (or soup) consisting of N triangles.

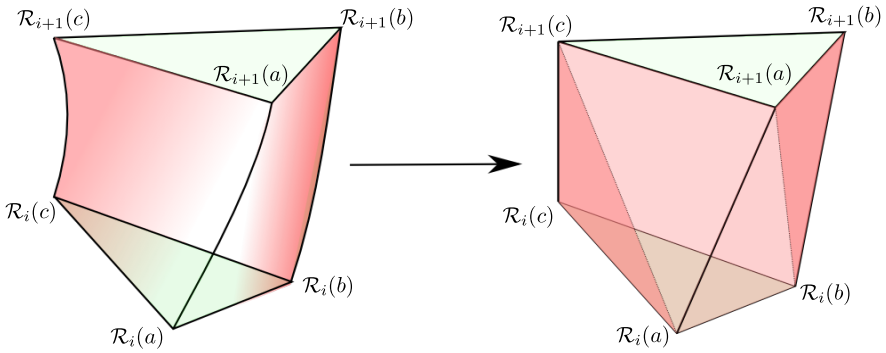

Figure 2.3: Sweeping the edges of a triangle yields three ruled surfaces, each of which is approximated and tessellated by inserting a diagonal.

As shown by Weld and Leu [48]; in order to compute the volume swept by $\mathcal{G}$ along a smooth motion, it suffices to compute the volume swept by its boundary, i.e., the volume swept by its triangles. Sweeping a triangle results in a volume that is bounded by the triangle in its start and end position (*facets*), three ruled surfaces (*patches*) arising from the swept edges, as well as a developable surface from the interior points of the triangle.

A ruled surface is per definition of the form

$$p(t, v) = q(t) + v \cdot r(t),$$

with the directrix $q \in \mathbb{R}^3$ and the ruling line generator $r \in \mathbb{R}^3$. The location of the directrix as well as the direction of the ruling line are both parameterized

by $t \in \mathbb{R}$ and the position along the line is given by $v \in \mathbb{R}$. A developable surface is a ruled surface with the special property, that the surface normal is constant along each of the ruling line generators [48].

It was proposed by Abrams and Allen [4] to approximate the ruled and developable surfaces with polygons. Campen and Kobbelt [12] proposed to tessellate the ruled surfaces emerging from the edges with two triangles by inserting a diagonal, as depicted in Figure 2.3. We call such a tessellated prism a **deformed prism**. According to [4] the developable surfaces can be approximated by the generator under the transformations of a densely sampled trajectory, which is the case in our target application. Note that the triangles bounding the prisms are exactly those triangles of the generator under the transformations of the discrete trajectory.

We define a **polyhedral approximation** $\mathcal{SV}$ to the actual SV to be the union of all deformed prisms arising from all triangles of $\mathcal{G}$ and all consecutive time steps of the discrete trajectory $\mathcal{R}_i$, $\mathcal{R}_{i+1}$, $i = 1, \ldots, m-1$, cf., Figure 2.3
If two triangles share an edge, i.e., the mesh is locally closed, the patch needs only to be tessellated once.

A major challenge is the high number of triangles arising from these prisms. Given a generator with N triangles $|\mathcal{G}| = N$ and a discrete trajectory $(\mathcal{R}_i)_{i=1,\ldots,m}$ with m transformations, this number is in $O(N \cdot m)$. In our scenarios, we sweep generators of several hundred thousand triangles along trajectories of several thousands of transformations. Hence, computing and storing these triangles (as proposed, for example, by Campen and Kobbelt [12]) is not possible. We therefore proposed in [44, 46] to compute the triangles on the fly and to store them (or their offsets) in volumetric voxel representations, whose memory consumption is mostly dependent on spatial resolution n, but independent from input model complexity.

Based on the voxel representation, we then construct an approximation to the offset $\mathcal{O}_\delta(\mathcal{SV})$ as well as an approximation to the tolerance hull $\mathcal{H}_{\varepsilon,\delta}(\mathcal{SV})$.

Typical tolerance demands (ε from the last section) are in the region of submillimeters, where the models (e.g. the engine) have dimensions of meters. This demands for a high resolution n, which in turn means a high number of voxels have to be stored. As an example, storing a voxel grid with a reasonable resolution of $n = 2^{12}$ causes a memory consumption of over 8 GB even when

the information occupied/non-occupied is encoded in a single bit per cell. Thus storing a full grid representation is not adequate.

For the purpose of storing the volumetric data of a swept volume, the usage of a distance field was proposed in [40]. This concept was extended to directed distance fields [27, 20] and adaptive distance fields [53]. However, for our intended use, all proposed data structures have an unnecessarily high memory consumption.

We now give a brief overview on voxelizations and the terminology used in the following thesis. Further we show some properties that will be necessary in later chapters.

In the context of this work, a voxel-grid with resolution $n \in \mathbb{N}$ is a uniform segmentation of the unit cube $[0,1]^3$ into n^3 small cubes (cells). Each cell is assigned a boolean value representing its state occupied/non-occupied. Occupied cell are called voxels.

Given $\mathcal{A} \subset [0,1]^3$, closed, and grid resolution n, we call a set of voxels $\mathcal{V}$ a **voxelization of** $\mathcal{A}$ if

$$\text{every cell whose center is covered by } \mathcal{A} \text{ is a voxel in } \mathcal{V}. \tag{2.4}$$

We say a voxelization $\mathcal{V}$ is **minimal** if

no voxels can be removed from $\mathcal{V}$ without violating condition 2.4.

We call a voxelization $\mathcal{V}$ **conservative** if

$$\text{every cell whose intersection with } \mathcal{A} \text{ is non-empty is a voxel in } \mathcal{V}. \tag{2.5}$$

Finally, a conservative voxelization $\mathcal{V}$ is said to be **minimal-conservative** if

$$\text{no voxels can be removed from } \mathcal{V} \text{ without violating (2.5).} \tag{2.6}$$

We now turn towards the deviation measured in terms of the directed Hausdorff metric between a set and its minimal voxelization.

Lemma 4. *Given a closed set $\mathcal{B} \subset [0,1]^3$, a grid resolution n and let $\mathcal{V}$ be the minimal voxelization of $\mathcal{B}$. Then it holds that*

$$h(\partial\mathcal{V}, \partial\mathcal{B}) \leq \frac{\sqrt{3}}{2n}.$$

Proof. Assume $v \in \partial\mathcal{V}$. Then v must lie on a voxel-face shared by an inside-voxel and an outside voxel. Let g_i and g_o be the midpoints of these voxels (resp.). The closed surface $S = \partial\mathcal{B}$ must intersect $\overline{g_i g_o}$ at least once and we call one of these intersection-points u and it holds that

$$d(v, u) \leq \frac{\sqrt{3}}{2n}.$$

Since no restrictions were imposed on the choice of v, this implies the Lemma. □

A similar result holds for the case of minimal-conservative voxelizations.

Lemma 5. *Given a closed set $\mathcal{B} \subset [0,1]^3$, a grid resolution n and let $\mathcal{V}$ be the minimal-conservative voxelization of $\mathcal{B}$. Then it holds that*

$$h(\partial\mathcal{V}, \partial\mathcal{B}) \leq \frac{\sqrt{3}}{n}.$$

In the following two chapters, we propose two different implementations that both choose a volumetric representation based on binary voxels, but they differ in their ways to store the voxelization as well as in the way they represent the tolerance hull.

Both approaches utilize the GPU and benefit from their massively parallel compute capabilities. The first approach that is presented in Chapter 3 is based on GPU depth buffer-based voxelization and makes heavy use of the programmable rendering pipeline. We present a special shader program that applies offsets to primitives on the fly and closely interacts with a Cuda kernel. The offset hull is implicitly defined by a sampling based inclusion test.

The second implementation (Chapter 4) uses a conservative voxelization algorithm and stores the resulting voxels in an incrementally growing octree data structure in CPU memory. We provide parallel implementations for GPUs as well as for multi-core CPUs. The offset hull is represented via additional voxel layers on the outer boundary of the initial voxelization.

In both implementations, the generation process of the final mesh is based on CGAL's Delaunay refinement that we extended by an additional bad facet criterion that is tailored to the respective tolerance hull representations.

CGAL follows the *generic programming paradigm*, that is, algorithms are formulated and implemented such that they abstract from the actual types, constructions and predicates. Using the C++ programming language, this is realized by means of class and function templates, respectively. As most other high level packages of CGAL, the meshing package is written such that it takes one major template argument, in this case, the class representing the domain. This class provides all other required types and predicates on them. This gives a tremendous flexibility in using the package, in particular, the user can implement his own intersection of the Voronoi edges with the boundary of the domain. Moreover, it is possible to provide additional criteria to classify triangles as bad, and thus schedule them for refinement.

Chapter 3

A Depth Buffer Based Concept Implementation

In this chapter, we present the first implementation of the concept presented in Chapter 2. We describe a Cuda kernel that closely interacts with the rendering pipeline to compute and directly render each triangle of $\mathcal{SV}$. We use a special shader program that creates offset geometry of each triangle on the fly, thus guaranteeing a conservative rasterization and correct depth values. Utilizing the depth buffer we then get a conservative voxelization of $\overline{\partial}\mathcal{O}_\delta(\mathcal{A})$ and can extract a triangle mesh from its surface. The offset hull is implicitly defined by a sampling based inclusion test.

We begin in Section 3.1 with a fast voxelization technique using the depth buffer of graphics hardware, together with the integration of our shader program and the Cuda integration. In Section 3.2, we introduce a method on how to extract a mesh from a tolerance hull around the depth buffer voxelization.

A second mesh extraction method (the one we originally used in [44]) is presented in Section 3.3.

It follows practical results in Section 3.4 and a conclusion in Section 3.5.

3.1 Depth Buffer Based Voxelization

Karabassi et al. [26] describe an efficient depth buffer based voxelization algorithm that relies on rasterization mechanisms of graphics hardware. The main

idea behind their approach is to generate one pair of depth buffer images from the object from each of the three principal axes and then extract those voxels that are inside the bounds given through the six images.

Assuming the object to be voxelized fits into the unit cube $[0, 1]^3$, we render the object with orthographic projection from each of the three principal axes twice (into an offscreen buffer). In the first pass, the depth buffer stores for every pixel the z value closest to the viewing plane and in the second pass, the farthest z values. The viewport is set to match the faces of the unit cube and to have $n \times n$ pixels. The depth values are stored in the RGB color channels of a texture with 32 bits per channel.

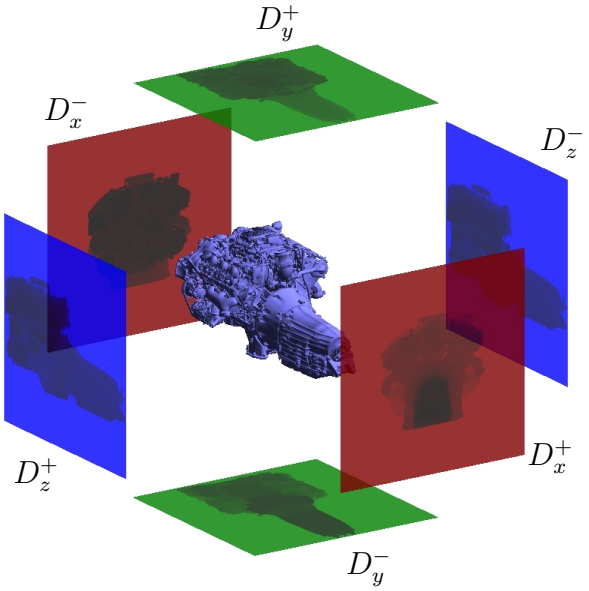

Figure 3.1: Three pairs of depth buffers images hold directed distances to the rendered geometry.

The z-buffer images $\{D_x^+, D_x^-, D_y^+, D_y^-, D_z^+, D_z^-\}$ now hold information on the regularly sampled directed distance values [28] from each unit cube-surface to the object's surface in the respective direction (cf. Figure 3.1). This defines a voxelization of the rendered object on a voxel grid with resolution n: a cell with

midpoint v is occupied (a voxel) iff the following three conditions hold:

$$\begin{aligned} D_x^-(k,\, j) \leq v_x \leq D_x^+(k,\, j) \\ D_y^-(i,\, k) \leq v_y \leq D_y^+(i,\, k) \\ D_z^-(i,\, j) \leq v_z \leq D_z^+(i,\, j). \end{aligned} \tag{3.1}$$

The method has two major drawbacks:

- First, only convex objects and a subset of concave objects can be voxelized correctly, since a concavity is only present in the voxelization if it is seen from at least one of the six directions. Attempts to overcome this problem have been made [5, 33]. They are primarily based on the use of more depth buffer images from other, arbitrary viewing directions and adding more depth buffer images for each direction to allow for a a higher depth complexity. Unfortunately, it causes the algorithmic complexity to become proportional to the depth complexity of the generator rendering this approach impractical for our applications with highly complex meshes.
- The second drawback is that, like many voxelization algorithms utilizing graphics hardware, this algorithm heavily relies on the rasterization mechanism that is not conservative [19], hence, we cannot expect the resulting voxelization to be conservative. According to the OpenGL specifications [42], a pixel is rasterized if its midpoint is occluded by the geometry, therefore, the resulting voxelization is a minimal voxelization of a superset of the rendered model that misses all concavities not visible to at least one depth buffer image.

We will now see that a conservative voxelization of a set $\mathcal{A}$ is given through the minimal voxelization of the offset $\mathcal{O}_\delta(\mathcal{A})$. If we render not the object itself to the depth buffers, but an adequate offset, we can ensure a conservative voxelization.

Theorem 2. *Given a closed set $\mathcal{A} \subset [0,1]^3$, a voxel grid with resolution n, and an offset radius $\delta \in \mathbb{R}$ greater than half the length of a voxel-diagonal $\delta > \frac{\sqrt{3}}{2n}$. Then*

1. *the minimal voxelization* $\mathcal{V}$ *of* $\mathcal{O}_\delta(\mathcal{A})$ *is a conservative voxelization of* $\mathcal{A}$ *and*

2. *it holds that*

$$h(\partial\mathcal{V}, \partial\mathcal{A}) < \frac{\sqrt{3}}{2n} + \delta.$$

We note that in general, $\mathcal{V}$ is not the minimal-conservative voxelization of $\mathcal{A}$. For a counterexample see [23].

Proof. 1. Assume $\mathcal{V}$ is not a conservative voxelization of $\mathcal{A}$, i.e. there exists an $a \in \mathcal{A}$ that is not contained by any voxel. Since $a \in [0,1]^3$ there must be some unoccupied cell containing a. Let v be the midpoint of that cell. Due to the offset condition $\mathcal{B}_\delta(a) \subset \mathcal{O}_\delta(\mathcal{A})$ and $\delta > \frac{\sqrt{3}}{2n} \geq d(a, v)$, the point v must fulfill $v \in \mathcal{B}_\delta(a)$ and so $v \in \mathcal{O}_\delta(\mathcal{A})$. Because $\mathcal{V}$ is assumed to be the minimal voxelization of $\mathcal{O}_\delta(\mathcal{A})$, the cell has to be occupied, hence, a contradiction.

2. With the triangle inequality for the directed Hausdorff distance we have

$$h(\partial\mathcal{V}, \partial\mathcal{A}) \leq \underbrace{h(\partial\mathcal{V}, \partial\mathcal{O}_\delta(\mathcal{A}))}_{1} + \underbrace{h(\partial\mathcal{O}_\delta(\mathcal{A}), \partial\mathcal{A})}_{2}.$$

   In Lemma 4, the first term was shown to be smaller than or equal to $\frac{\sqrt{3}}{2n}$. For the second term we state that for every $s \in \partial\mathcal{O}_\delta(\mathcal{A})$ there must be an $a \in \mathcal{A}$ such that $s \in \mathcal{B}_\delta(a)$, hence $d(s, a) \leq \delta$. Since $\mathcal{A}$ is closed in $[0,1]^3$, we can assume a to lie on the boundary of $\mathcal{A}$. As we did not put any restrictions on the selection of s, it holds that $h(\partial\mathcal{O}_\delta(\mathcal{A}), \partial\mathcal{A}) \leq \delta$.

□

In the following section, we present a new rendering technique that can render arbitrary offsets of triangle meshes, which we utilize to overcome the problem of non-conservative rasterization. We will give proper bounds on the deviation from each produced fragment to the original geometry.

3.1.1 Rendering Offsets on the Fly

Previous work on offset surfaces mainly addresses the computation and extraction of offset geometry, therefore these existing methods are not applicable for real-time offset rendering. In one of the few publications on offset visualization [22], the authors propose to generate a complete distance field of the mesh and to extract a point based contour, that is, rendered using splatting. In contrast to our approach, the time consuming process of creating a distance field is needed. Our approach follows the general idea of [38, 16, 29]. They use the fact that an offset surface of a triangle is the surface of the union of three spheres, three cylinders and a prism, trimmed at their intersection points. Our method circumvents this difficult process of trimming. As we are only interested in rendering the offset surface, it is implicitly done by the depth buffer. A method for extracting offset geometry is presented in the recent work of [34]. The authors compute an adaptive distance field of the input geometry, storing the minimum and an approximated maximum distance to each triangle using special distance-functions (ball, cylinder prism). They generate the offset in an isosurface extraction process followed by feature reconstruction and mesh simplification. Offsetting can be treated as a special case of a Minkowski sum between a polyhedral object and a tessellated sphere. Due to tessellation errors, this yields only an approximation. For recent work see [12, 30]. Attempts to overcome the problem of nonconservative rasterization have been made [19]. Similar to our work, a combination of a geometry and a fragment shader are used to achieve conservativeness, but their method only applies rather coarse depth values and their computation highly overestimates depth values for triangles with normals perpendicular to the viewing plane. Further, their method cannot render arbitrary offsets.

In [44], we presented a method to overcome this inaccurate depth value estimation and therefore are able to give proper bounds on the deviation from each produced fragment to the original geometry. We recapitulate the method in the following section. The key idea is that the geometry shader emits triangles, that, after projection onto the viewing plane, form a bounding geometry of the projection of the offset geometry onto the viewing plane. For every fragment of the bounding geometry, the fragment shader casts a ray in viewing direction and intersects this ray with the offset geometry.

Given a, b, $c \in \mathbb{R}^3$ we denote $T_{(a,b,c)}$ to be the **triangle** with its **vertices** a, b, c

and **edges** e_{ab}, e_{bc}, e_{ac}.

For an edge e_{ab} and $\delta > 0$ we define the **solid cylinder** $C_\delta(e_{ab})$ with axis e_{ab} and radius δ to be the set

$$C_\delta(e_{ab}) = \Big\{ x \in \mathbb{R}^3 \;\Big|\; \Big\| (x-a) \times \frac{b-a}{\|b-a\|_2} \Big\|_2^2 \leq \delta^2 \wedge \quad 0 \leq \frac{(x-a)\cdot(b-a)}{\|b-a\|_2^2} \leq 1 \Big\}.$$

For a given triangle $T_{(a,b,c)}$ with **normal** $n = \frac{(b-a)\times(c-a)}{\|(b-a)\times(c-a)\|_2}$ we define a (δ-) **offset-prism** of $T_{(a,b,c)}$ to be the convex hull of the two shifted triangles $T_{(a+\delta n, b+\delta n, c+\delta n)}$ and $T_{(a-\delta n, b-\delta n, c-\delta n)}$:

$$P_\delta(T_{(a,b,c)}) = \mathrm{CH}\big\{ a+\delta n,\, b+\delta n,\, c+\delta n,\, a-\delta n,\, b-\delta n,\, c-\delta n \big\}.$$

The offset to a triangle $T_{(a,b,c)}$ equals the union of three spheres, three cylinders and a prism:

$$\mathcal{O}_\delta(T_{(a,b,c)}) = P_\delta(T_{(a,b,c)}) \cup C_\delta(e_{ab}) \cup C_\delta(e_{bc}) \cup C_\delta(e_{ac}) \cup \mathcal{B}_\delta(a) \cup \mathcal{B}_\delta(b) \cup \mathcal{B}_\delta(c).$$

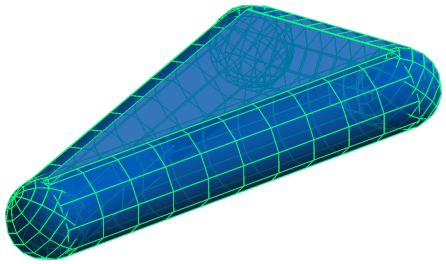

Figure 3.2: Tessellated offset geometry of a triangle.

In the case of a high triangle count, the straightforward method, to compute and tessellate this geometry for every triangle (cf., Figure 3.2), is not practical.

We therefore present a new approach to efficiently compute and render the exact offset geometry up to pixel resolution. A combination of a geometry shader [43] and fragment shader [39] computes the depth-values of fragments of the offset boundary on the fly, hence no extra storage is needed.

With our new method, every triangle is processed by the following rendering pipeline: First the vertex shader passes on the vertices of the triangle to the geometry shader stage without any change. The geometry shader creates a patch for each edge of the triangle in the xy-plane and also emits the two triangles of the offset prism in 3D. The latter become rasterized with the correct depth value and the fragments can be written to the depth buffer directly. The rectangular patches of the edges however still have a depth value according to $z = 0$, hence we calculate the correct depth value for each patch-fragment with a ray casting method, as shown in Figure 3.3.

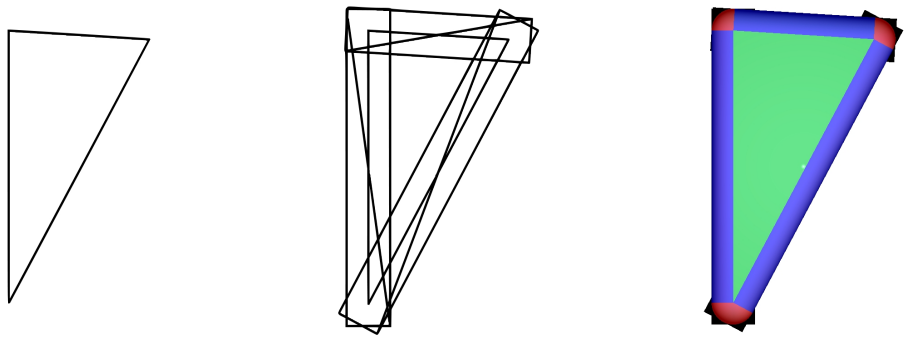

Figure 3.3: For each edge of the triangle projected to the xy-plane, a patch is constructed in the geometry shader stage and the appropriate depth values are computed in the fragment shader (blue, red). Together with the shifted triangle (green), the final offset surface is rendered, and dispensable fragments (black) are clipped.

We now give a detailed description of the shaders. We used the OpenGL Shading Language (GLSL) [39].

Vertex stage

In the rendering pipeline, the vertex shader is responsible for applying any kind of transformations on the vertex data. In our case, the vertex shader simply passes on the position values for each vertex to the geometry shader.

Geometry stage

The geometry shader [43] is able to create new primitives (in our case triangles) that are passed on to the rasterization stage just as primitives directly rendered by OpenGL. In contrast to the previous stage where each vertex was treated independently, now, information on vertex-connectivity is provided, i.e. we can work on the input triangles. In the general case we emit eight triangles per input triangle: two for each edge-patch and two triangles for the offset prism.
For each edge e_{ab} of the triangle we compute a rectangular patch that is a bounding box of the projection of the δ-balls $\mathcal{B}_\delta(a)$, $\mathcal{B}_\delta(b)$ and the δ-cylinder $C_\delta(e_{ab})$ on the viewing plane. The geometry shader computes and emits the triangles of the tessellated patch

$$\{s,\, t,\, v,\, u\} = T_{(s,t,u)} \cup T_{(t,v,u)},$$

as shown in figure 3.4. The dimensions of the patch are given as

$$d_1 = \frac{\delta \cdot h}{\|h\|_2},\; d_2 = \begin{pmatrix} -(d_1)_y \\ (d_1)_x \\ 0 \end{pmatrix},\; h = \begin{pmatrix} (p_z(b-a))_x \\ (p_z(b-a))_y \\ 0 \end{pmatrix},$$

where $p_z(\cdot)$ is the orthogonal projection onto the viewing plane.

The offset triangles of the δ-offset prism can be computed by shifting the vertices a, b, c by δ once in normal direction and once in the opposite direction. The rectangular faces of the prism are not needed since they are completely occluded by the cylinders.
All emitted triangles are now rasterized and cut into loose fragments. To be able to relate the fragments to their edges or offset triangles, we pass on two 4-vectors (to the fragment shader) holding all information needed. The coordinates of the vertices of the edge are passed as color values and the information, whether the fragment belongs to an offset triangle or a patch, is encoded in the alpha channels.

Fragment stage

The fragment shader now has to cope with the following three tasks:
Firstly, it must decide whether the fragment $f = (f_x, f_y, f_z)$ belongs to an offset

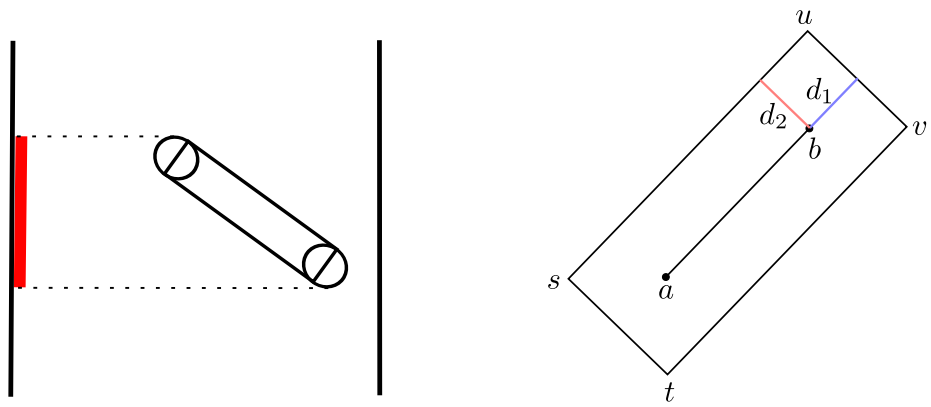

Figure 3.4: Left: For orthogonal projection it suffices to emit the bounding rectangle of the capsule in the viewing plane. Right: The bounding rectangle for the edge e_{ab}.

triangle or an edge-patch. Second, it must discard the fragments that are outside the offset and finally, adjust the correct depth values.
The first task can be completed easily by checking the alpha value of the variables received from the geometry shader. In the case of an offset triangle, the z coordinate of the built-in fragment coordinates can be mapped directly to the output depth value just like in the fixed function pipeline.
For the computation of the correct depth value for a patch-fragment, we define the line $\tilde{e}_{ab}$ as the edge e_{ab} extended to infinity and the endless cylinder $C_\delta(\tilde{e}_{ab}) = \mathcal{O}_\delta(\tilde{e}_{ab})$, given in the implicit form:

$$C_\delta(\tilde{e}_{ab}) = \left\{ p \in \mathbb{R} \,\middle|\, \|(p-a) \times (b-a)\|_2^2 \leq \delta^2 \|b-a\|_2^2 \right\}.$$

The intersection of the ray

$$r(\lambda) = \begin{pmatrix} f_x \\ f_y \\ 0 \end{pmatrix} + \lambda \begin{pmatrix} 0 \\ 0 \\ -1 \end{pmatrix}, \quad \lambda > 0$$

with $\partial C_\delta(\tilde{e}_{ab})$ yields the quadratic equation (with $\hat{e}_z = (0,0,1)^T$)

$$\begin{aligned} & \Big\| (f-a-\lambda\hat{e}_z) \times (b-a) \Big\|_2^2 = \delta^2 \|b-a\|_2^2 \\ \Leftrightarrow \quad & \Big\| \underbrace{(f-a) \times \frac{(b-a)}{\|b-a\|_2}}_{=:c} - \lambda \big(\underbrace{\hat{e}_z \times \frac{(b-a)}{\|b-a\|_2}}_{=:d} \big) \Big\|_2^2 = \delta^2 \end{aligned}$$

that we can solve for λ :

$$\lambda_{1,2} \;= \tfrac{c\cdot d}{d\cdot d} \pm \sqrt{\big(\tfrac{c\cdot d}{d\cdot d}\big)^2 - \tfrac{c\cdot c - \delta^2}{d\cdot d}}.$$

Of the two possible values, we chose the greater one (λ_1) and set $f_z = -\lambda_1$, since we are interested in the intersection closer to the viewing plane.

So far we have only considered an endless cylinder with axis e_{ab}. To acquire the correct offset $\partial\mathcal{O}_\delta(e_{ab})$, we need to clip the cylinder at a and b, i.e. all fragments not fulfilling

$$0 \leq (f-a)\cdot(b-a) \leq (b-a)\cdot(b-a)$$

have to be either projected on

- the ball $\mathcal{B}_\delta(a)$ for $(f-a)\cdot(b-a) < 0$ or
- the ball $\mathcal{B}_\delta(b)$ for $(f-a)\cdot(b-a) > \|b-a\|_2^2$.

For this we just have to intersect $r(\lambda)$ with the respective ball to get the correct depth value. In the case of no intersection, we discard the fragment. This only happens in the corners of the patch, where it is outside the projected balls around a and b, as shown in Figure 3.3.
For example, in the case of $\mathcal{B}_\delta(a)$, the intersection yields

$$f_z = -\lambda = a_z - \sqrt{\delta^2 - (f_x - a_x)^2 - (f_y - a_y)^2}.$$

An offset of an engine rendered in real time with the presented shader program is shown in Figure 3.5. The offset radius δ is set very high and per pixel shading is applied for demonstration purposes.

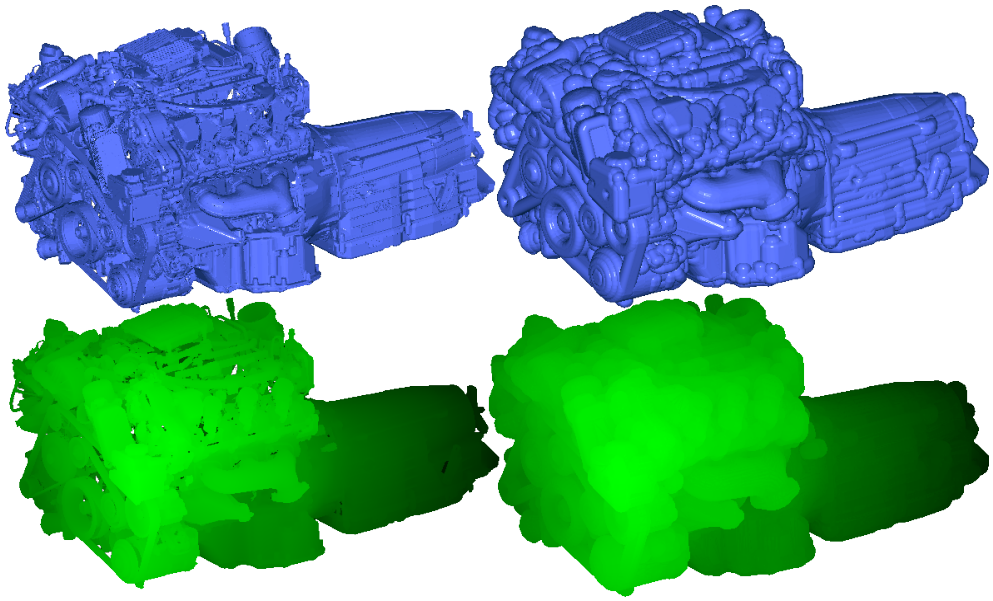

Figure 3.5: Rendered original model (left) and offset geometry (right) of a car engine consisting of ca. 330k triangles. Top: per pixel shading applied. Bottom: depth buffer images.

Performance of the Offset Shader

When programming shaders it is very important, performance-wise, to minimize shader divergence. In the offset shader presented here the computational bottleneck clearly lies in the fragment shader. Every fragment is assigned a thread executing the shader program with the attributes (input variable data) of that fragment. Threads from neighboring fragments (in screen space) are grouped and executed in parallel. Hence, strictly speaking, it is important to minimize shader divergence in these groups!

Although the fragment shader presented here takes completely different code paths depending on the geometric primitive of the fragment (ball, cylinder, triangle), divergence will only be present in those groups that have different primitive types. This is only the case in regions where the primitives intersect each other, therefore, we can expect most of the groups to be homogeneous.

3.1.2 Incremental Voxelization Pipeline

For a conservative swept volume voxelization using the aforementioned shader pipeline, we need to compute and render the offset $\mathcal{O}_\delta(\mathcal{SV})$, which is problematic, since we cannot hold all triangles of $\mathcal{SV}$ in memory at once.

The solution lies in the definition of the offset and the distributivity of the Minkowski sum:

$$\mathcal{O}_\delta(\mathcal{SV}) = \mathcal{SV} \oplus \mathcal{B}_\delta(0) = \bigcup_{T \in \mathcal{SV}} T \oplus \mathcal{B}_\delta(0) = \bigcup_{T \in \mathcal{SV}} \Big(\mathcal{O}_\delta(T)\Big).$$

Therefore, to render an offset of $\mathcal{SV}$ we can render the offsets of its triangles (in an arbitrary order) and accumulate the offsets of all triangles in the depth buffers, cf. Figure 3.6.

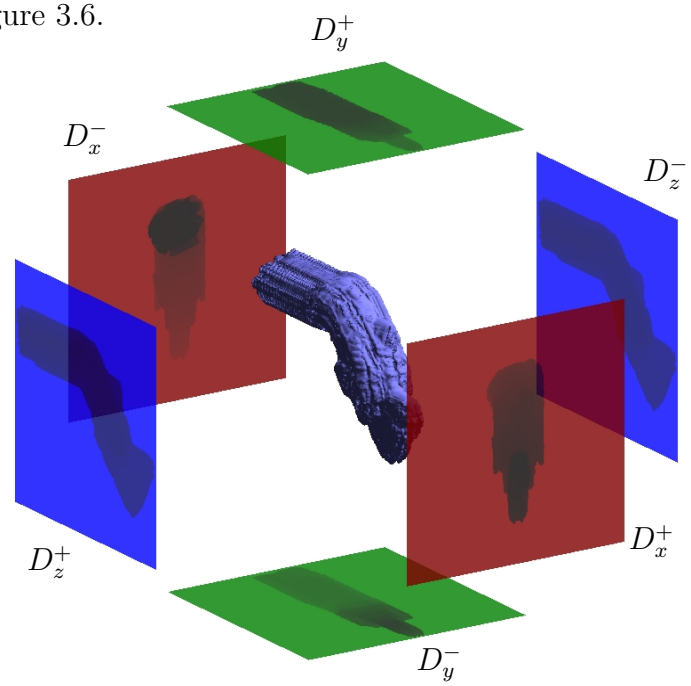

Figure 3.6: Three pairs of sweep depth buffers images.

However, computing them on the CPU, and directly rendering them on the fly with the offset pipeline described in the last section, causes an unbearable lot of CPU-GPU memory traffic causing the bottleneck of the algorithm.

We therefore propose to store the generator geometry and the transformations of the trajectory in GPU memory and to compute and render the triangles of $\mathcal{SV}$ by utilizing a collaboration of a Cuda kernel and the offset pipeline.

The OpenGL Vertex Buffer Objects (VBOs) provide a mechanism to upload data to video device memory. The data can be used for direct rendering which is faster than rendering with the immediate mode, but can still be easily modified. Further, Cuda offers the possibility to inter-operate with VBOs. Data in a VBO, that is registered with Cuda, can conveniently be modified in a Cuda kernel, which is launched in between rendering passes, cf. Figure 3.7.

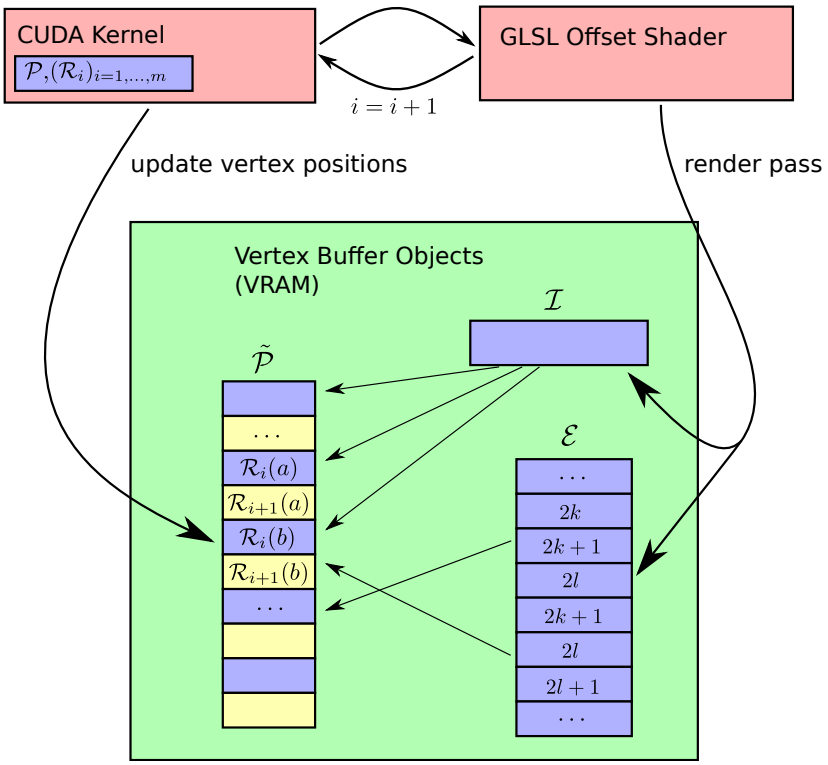

Figure 3.7: Interoperation between Cuda and VBOs in the incremental voxelization pipeline.

Assume the generator is given as a triangle mesh $\mathcal{G}$, consisting of N triangles stored in an indexed face set with vertices $\mathcal{P}$ and a list $\mathcal{I}$ representing the indices

of the triangles. We further assume having a list of indices representing the edges of $\mathcal{G}$. The trajectory $(\mathcal{R}_i)_{i=1,\dots,m}$ consists of m transformations.

We now interleave two copies of $\mathcal{P}$ such that every vertex is stored twice at consecutive locations and store this array $\tilde{\mathcal{P}}$ in form of a VBO in the GPU memory.

For every $i = 1, \dots, m-1$, we require this buffer to hold the coordinates of $\mathcal{P}$ transformed with $\mathcal{R}_i$ at odd index locations and the coordinates of $\mathcal{P}$ transformed with $\mathcal{R}_{i+1}$ at even index addresses. In principal, it is not necessary to update the whole buffer in between every two consecutive time steps i, $i+1$, since every entry at an odd position in $\tilde{\mathcal{P}}$ is just copied to the position of its predecessor at every time step. Hence, we could update even and odd positions only every other time (in turns). However, our tests have shown that the overhead in organizing even and odd turns is comparable to the overhead of updating the whole buffer at every time-step. For the sake of simplicity, we chose to update the whole buffer.

To do this task we use Cuda. The Cuda kernel only needs the transformations $(\mathcal{R}_i)_{i=1,\dots,m}$ and the vertex coordinates of the generator in its resting positions. This data remains unchanged and is copied to Cuda global memory only once. The Cuda kernel is called with a single parameter, the current time step i, and updates $\tilde{\mathcal{P}}$.

The data structure $\mathcal{E}$ holds the indices of the interpolated edges and its entries are computed in the following way: For every edge of $\mathcal{G}$ with two vertices $\mathcal{P}[k]$ and $\mathcal{P}[l]$ we store the following indices in $\mathcal{E}$:

$$2k, \quad 2k+1, \quad 2l, \quad 2k+1, \quad 2l, \quad 2l+1.$$

Since it holds that

$$\begin{aligned} \mathcal{R}_i(\mathcal{P}[k]) &= \tilde{\mathcal{P}}[2k] \\ \mathcal{R}_{i+1}(\mathcal{P}[k]) &= \tilde{\mathcal{P}}[2k+1] \\ \mathcal{R}_i(\mathcal{P}[l]) &= \tilde{\mathcal{P}}[2l] \\ \mathcal{R}_{i+1}(\mathcal{P}[l]) &= \tilde{\mathcal{P}}[2l+1] \end{aligned}$$

the array $\mathcal{E}$ now holds the indices of the triangles of the tessellated interpolated edges in a consecutive order, cf. Figure 3.8.[1]

[1] We disabled face culling so we do not cull falsely oriented triangles if the generator is mal-

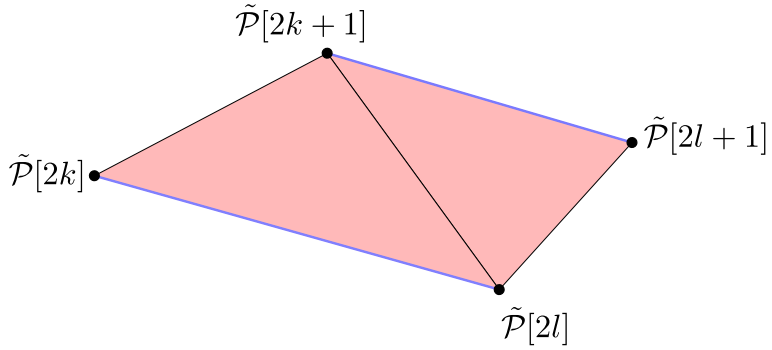

Figure 3.8: The indices of the vertices of the two triangles lie at consecutive buffer positions.

Thus, after the Cuda kernel returns control to the host (CPU) we now render the following VBOs with the offset shader applied to the OpenGL rendering pipeline:

- A VBO $\mathcal{I}$ holding the indices $\mathcal{I}$, multiplied by 2, each, that point to $\tilde{\mathcal{P}}$. This buffer is rendered at every time step: $1, \ldots, m-1$.
- The VBO $\mathcal{E}$ as previously defined, that points to $\tilde{\mathcal{P}}$. This buffer is rendered at time steps $1, \ldots, m-1$.

Lastly, the generator is rendered one last time in its end position $i = m$.

3.2 Sampling Based Delaunay Mesh Generation

The depth buffer images are created one after the other, since when using the maximum texture size[2] only one depth buffer image can reside in GPU memory at once.

When all triangles of $\mathcal{SV}$ are rendered to the depth buffer with the offset shader applied for one of the six directions, we download the depth buffer image to CPU

formed. Therefore we do not have to take care of triangle orientations.

[2]On the NVIDIA GTX480 that was used in the benchmarks this maximum is $2^{14} \times 2^{14}$. When using 32 bit floats, this results in 1 GB of data for each of the 6 images.

memory, clear the depth buffer, set the rendering pipeline according to the next direction and again render all triangles of $\mathcal{SV}$.

Thereafter, the directed distance values in the six depth buffer images, together with conditions (3.1) define a point membership test of a superset of the volume $\mathcal{O}_\delta(\mathcal{SV})$. We call this superset $\mathcal{SV}_{DB}$ as it only accounts for concavities seen by at least one depth buffer image.

These concepts are depicted for the two-dimensional case in Figure 3.9.

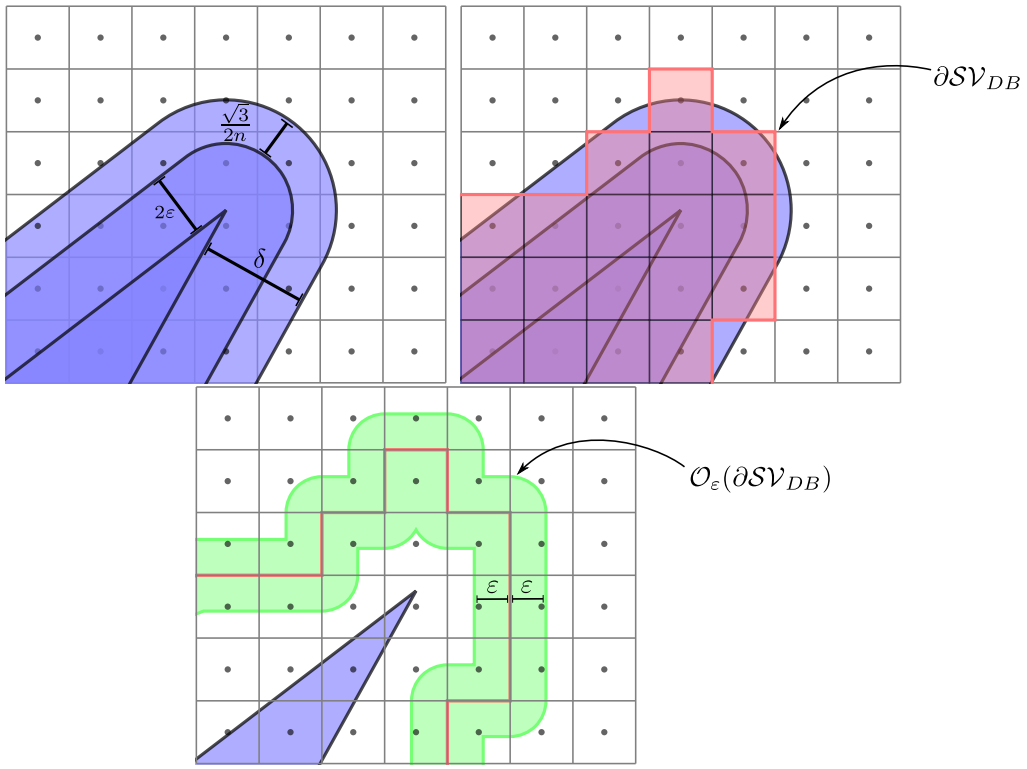

Figure 3.9: For a single triangle, a δ-offset is rasterized (top left), the depth buffers define a voxelization $\mathcal{SV}_{DB}$ (top right), around which the hull is defined (bottom).

To apply an error bounded mesh extraction algorithm we first need an appropriate offset hull as demanded in Chapter 2.

For this, we generate a set of sampling points $\mathcal{S}_\varepsilon$ on a sphere centered at the origin $\partial\mathcal{B}_\varepsilon(0)$. We choose a method that generates uniformly distributed samples on $[0, 2\pi] \times [0, \pi]$ and maps these to the sphere boundary via the standard spherical coordinate transformation.[3]

We define the set $\mathcal{H}_{DB} \subset [0, 1]^3$ via

$$\begin{aligned} \mathcal{H}_{DB} = &\{x \in \mathcal{SV}_{DB} : \exists y \in \mathcal{S}_\varepsilon,\, x + y \notin \mathcal{SV}_{DB}\} \cup \\ &\{x \notin \mathcal{SV}_{DB} : \exists y \in \mathcal{S}_\varepsilon,\, x + y \in \mathcal{SV}_{DB}\}\,. \end{aligned} \tag{3.2}$$

It holds that $\mathcal{H}_{DB} \subset \mathcal{O}_\varepsilon(\partial\mathcal{SV}_{DB})$ since for every $x \in \mathbb{R}^3$ the membership in $\mathcal{H}_{DB}$ is a sufficient condition that $d(x, \mathcal{SV}_{DB}) < \varepsilon$, as depicted in Figure 3.10.

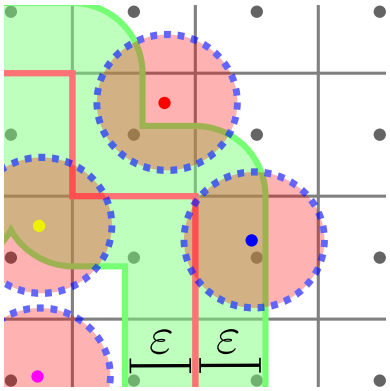

Figure 3.10: The hull is implicitly defined with a sufficient sampling based criterion: The blue point meets this criterion, since it is outside $\mathcal{SV}_{DB}$ and at least one of the sampling points is inside $\mathcal{SV}_{DB}$. The yellow point meets this criterion, since it is inside $\mathcal{SV}_{DB}$ and at least one of the sampling points is outside $\mathcal{SV}_{DB}$. The red point does not meet this criterion, since it is outside $\mathcal{SV}_{DB}$ and all sampling points are outside of $\mathcal{SV}_{DB}$. The purple point does not meet this criterion, since it is outside $\mathcal{SV}_{DB}$ and all sampling points are outside of $\mathcal{SV}_{DB}$.

As described in Section 2.2, we need an initial point set on the boundary of the domain to be approximated $\partial\mathcal{SV}_{DB}$. Any pixel in any depth buffer image holds

[3] $\mathcal{S}_\varepsilon$ can be any sampling with reasonably distributed samples.

the distance to a point on $\mathcal{SV}_{DB}$ in a given direction, hence it is a witness of a point on $\mathcal{SV}_{DB}$. Therefore the witness point of any random pixel that has a depth value > 0 can be chosen as an initial point.

The initial mesh is refined until all user defined criteria are met for all triangles, such as a bound on the maximum facet size or minimum angles.

The additional bad facet criterion for a facet f mentioned in (2.3) now translates to

$$f \cap \mathcal{H}_{DB} \neq f. \tag{3.3}$$

In practice we do not test the whole triangles for containment in $\mathcal{H}_{DB}$, but rather a subset, formed by uniformly sampling the triangles. We ensure that the triangle-sampling obeys a user defined sampling density σ_{Tri}, i.e., for every point on the triangle, there exists at least one sampling point that is closer than σ_{Tri}.

When a facet is declared bad it has to be refined as described in Section 2.2. For this the intersection point of a Voronoi edge with $\partial\mathcal{SV}_{DB}$ has to be computed. We implemented this intersection routine using a binary search, since checking points for membership in $\mathcal{SV}_{DB}$ is $O(1)$. The search stops at a user defined tolerance σ_{Bin}.

The process of a single Delaunay refinement step (for the two-dimensional case) is depicted in Figure 3.11.

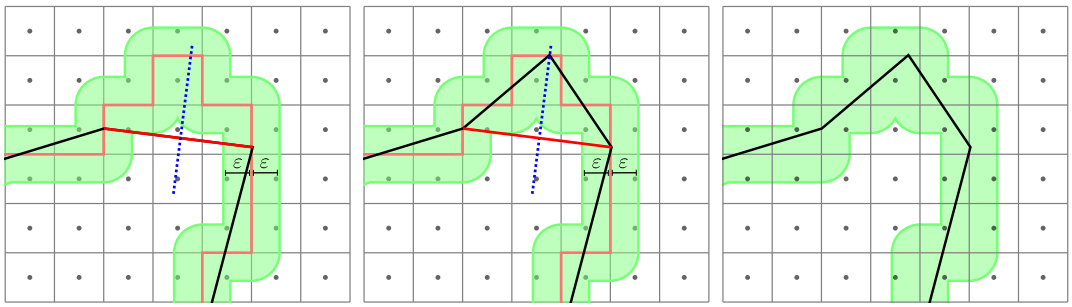

Figure 3.11: If at least one sampling point on a triangle is not inside the hull, the triangle is refined. The process terminates when all the sampling points of all triangles are inside the hull.

We chose to utilize the Delaunay refinement package [37] of CGAL[4]. We kept the option to control the overall quality of the resulting mesh and added the additional sampling based bad facet criterion as well as the intersection routine.

The definition of the hull introduces no further error, since the sampling based sufficient criteria actually shrinks the hull: $\mathcal{H}_{DB} \subsetneq \mathcal{O}_\varepsilon(\partial\mathcal{SV}_{DB})$. Only if we had a perfect sampling, meaning $\mathcal{S}_\varepsilon = \partial\mathcal{B}_\varepsilon(0)$, the above sets would be equal.

Also, the binary search for finding new vertices of the output mesh during the Delaunay refinement does not introduce an additional error. As long as the tolerance of the binary search, σ_{bin}, is smaller than two times the radius of the hull 2ε, we are guaranteed that all vertices are inside the hull.

However, we sampled the triangles when testing them for containment in the hull $\mathcal{H}_{DB}$, which introduces any additional error. We chose a sampling on a facet f that is σ_{tri}-dense. Hence, whenever a containment test returns positive, we can be sure that for every point $x \in f$, there exists (at least) one sample point s with

$$s \in \mathcal{H}_{DB}, \text{ and } \|x - s\|_2 \leq \sigma_{tri}.$$

This can lead to parts of the triangle that are outside the hull. But since their one-sided Hausdorff distance to the hull is bounded by σ_{tri}, this can easily be fixed by enlarging the offset radius. Incorporating this error leads to a required offset radius

$$\delta = \frac{\sqrt{3}}{2n} + \sigma_{tri} + \varepsilon,$$

where the tolerance radius ε of the hull remains unchanged. This guarantees conservativeness:

$$\mathcal{M} \cap \mathcal{SV} = \emptyset.$$

Unfortunately, this also means that the maximum deviation of the output mesh $\mathcal{M}$ to $\mathcal{SV}$, in terms of the one-sided Hausdorff distance, is now, according to Theorem 1, in all convex areas and visible concavities:

$$h(\mathcal{M}, \mathcal{SV}) \leq \varepsilon + \delta + \sigma_{tri} = 2\varepsilon + \frac{\sqrt{3}}{2n} + 2\sigma_{tri}.$$

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

As depicted in Figure 3.11 the hull is defined around the boundary of the voxelization $\mathcal{SV}_{DB}$, and not around $\overline{\partial}\mathcal{O}_\delta(\mathcal{SV})$. So, in the worst case, there is an additional error of $\frac{\sqrt{3}}{2n}$ according to Lemma 4. Thus, we have as a final result:

$$h(\mathcal{M}, \mathcal{SV}) \leq \frac{\sqrt{3}}{n} + 2\varepsilon + 2\sigma_{tri}, \tag{3.4}$$

with $\delta = \frac{\sqrt{3}}{2n} + \varepsilon + \sigma_{tri}$.

Hence, our method guarantees the resulting mesh to be a conservative approximation of $\mathcal{SV}$, that does not deviate more than $\frac{\sqrt{3}}{n} + 2\varepsilon + 2\sigma_{tri}$ from $\mathcal{SV}$ at all convex areas and visible concavities.

3.3 An Alternative Mesh Extraction Method

As an alternative to the top down Delaunay refinement, we presented in [44] a GPU based method to extract a triangle mesh from a tolerance hull in a bottom up fashion. We here give a brief overview.

We start with a corollary of Theorem 1, directly making use of Lemma 4.

Corollary 1. *Given a closed set $\mathcal{A} \subset [0,1]^3$ and $\delta > \epsilon > \frac{\sqrt{3}}{2n}$. Further let $\mathcal{V}$ be the minimal voxelization of $\mathcal{O}_\delta(\mathcal{A})$ Then it holds that*

$$\partial\mathcal{V} \subset \mathcal{O}_\epsilon(\partial\mathcal{O}_\delta(\mathcal{A})).$$

Corollary 1 justifies the use of the tessellated voxelization-surface as a starting mesh for further simplification steps. As long as we make sure, that during the simplification, we stay inside the conservative offset hull, Theorem 1 guarantees the desired conditions. We presented in [44] a GPU based method to extract a triangle mesh from the surface of a voxelization. The voxelization boundary consists of rectangular voxel-faces, each of which can be tessellated with 2 triangles (cf. Figure 3.12).

The mesh extracted from the voxelization does not self intersect, has correctly oriented triangles, and has neither holes nor T-junctions. However, in general, it is not 2-manifold since it lacks two topological properties: A vertex can be

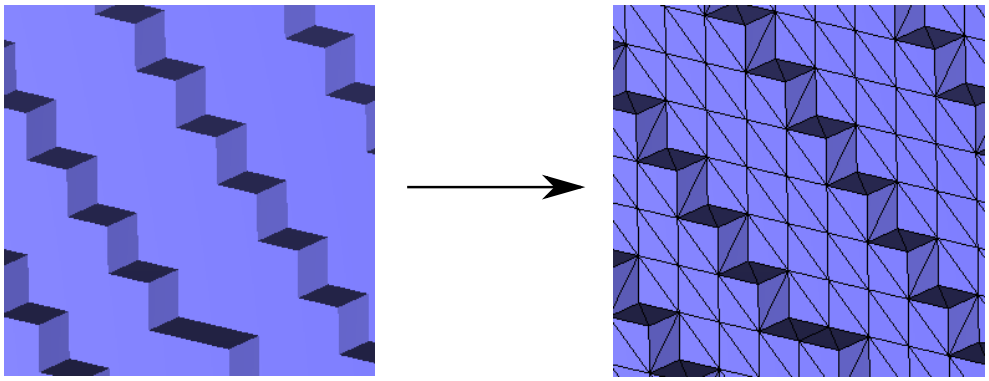

Figure 3.12: A triangle mesh is extracted from the boundary of a voxelization by tessellating each voxel face with two triangles.

adjacent to more than one triangle-fan and an edge can be adjacent to more than two triangles. Since in practice these degeneracies have shown to occur rarely, according to [14], the resulting mesh is termed *mostly manifold*.

We proposed a mesh-simplification method that is able to cope with the two degeneracies mentioned above and does not rip holes in the mostly manifold mesh using the edge collapse operator [14]. During an edge collapse, two adjacent vertices are merged and the two (or more in the case of degeneracies) triangles bounded by the edge are deleted. However, due to the locality of the operation, maintaining a global error bound during multiple edge collapses is challenging. We solved this by applying a simplification method that retains a one-sided Hausdorff distance error bound utilizing the programmable rendering pipeline.

Starting from the initial mesh extracted from the outer voxelization boundary, successive mesh simplification steps are applied, cf. Figure 3.13.

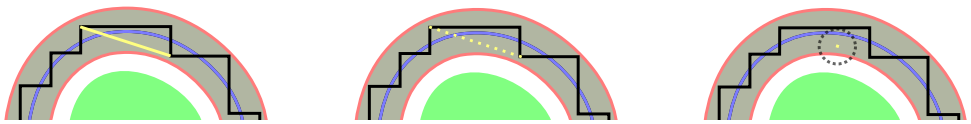

Figure 3.13: A simplification step is valid if all affected triangles are inside the tolerance hull.

All triangles affected by a candidate simplification step are rendered, sampled

by hardware rasterization, and a special fragment shader tests the fragments for containment in $\mathcal{H}_{DB}$. Only the fragments outside $\mathcal{H}_{DB}$ are written to the framebuffer. A simplification step is valid, if no fragments are written to the framebuffer.

OpenGL has a mechanism called occlusion queries that count the pixels that pass the depth test. We use this mechanism to detect if a step is valid. The benefit of this approach is that we do not have to check at every single pixel position of the rendered image if a fragment has been rendered.

This validation test is done in the fragment shader by testing uniformly distributed sample points on $\partial\mathcal{B}_\epsilon(p)$ similar to the one described in the last section. The six depth buffer images are copied to GPU texture memory and accessed by each fragment independently via OpenGL texture sampling. Unfortunately, sampling of textures introduces inaccuracies, since we have only the correct depth values for the texel-midpoints. An inner (outer) sample point can be falsely classified to be outside (inside), because the midpoint of its containing voxel (given through the 6 texels) is flagged outside. Therefore the maximum error introduced by texture-sampling is $\sigma_1 = \frac{\sqrt{3}}{2n}$, with n being the grid resolution.

To guarantee a uniform rasterization of the triangles for the fragment generation, each triangle is mapped to the standard triangle scaled with the length of the longest edge of the triangle. The texture coordinates are set accordingly to the actual positions of the vertices of the triangle in $[0,1]^3$ and passed on to the shaders. The sampling of the scaled standard triangle introduces further inaccuracies: the maximum spatial deviation from a point on the a triangle to its nearest sample-point is at most $\frac{\sqrt{2}}{2n}$. Altogether the overall maximum sampling error turns out to be $\sigma = \sigma_1 + \sigma_2 = \frac{\sqrt{2}+\sqrt{3}}{2n}$.

Unfortunately, the applicability of this top down approach is limited by the high memory demands of the highly over-tessellated intermediate mesh for fine voxelizations. We therefore propose to use the bottom up Delaunay mesh generation method described earlier.

3.4 Experimental Results of the Depth Buffer Based Approach

The overall error given by (3.2) in the last section depends on the 2 parameters: σ_{tri}, the sampling density of the triangles, and ε, the hull thickness. For the performance of the proposed method it is very important to choose these parameters carefully.

In the depth buffer image generation phase the rendering is highly dominated by the workload of fragment shaders. The bigger the offset radius $\delta = \frac{\sqrt{3}}{2n} + \varepsilon + \sigma_{tri}$, the more fragments are produced. Therefore the first part of the method greatly benefits from a small σ_{tri} and ε.

However there are two drawbacks in the meshing phase:

- A small σ_{tri} causes a lot of sample points on the triangles that need to be tested. In our experimental tests we found that the running time of the meshing phase is about quadratic in $1/\sigma_{tri}$. This is not surprising since during the meshing phase most of the time is spent on testing sampled triangles for containment in the hull, and the number of sampling points on each triangle grows with σ_{tri}^{-2}.
- A small hull thickness ε, i.e., a thin tolerance volume, causes a very fine tessellated output mesh in non-flat areas. This increases both running time of the meshing phase, as well as output complexity.

Depending on the scenario one can decide whether the depth buffer generation or the meshing dominates the overall running time and adjust the parameters accordingly. We found out that, for our scenarios, a sense output complexity as well as balanced running times of both phases can be achieved with setting

$$\varepsilon = \sigma_{tri} = \frac{\sqrt{3}}{n},$$

for $n = 2^{10}, \ldots, 2^{14}$ and $\delta = \frac{5\sqrt{3}}{2n}$.

Hence the overall error in our test scenarios is

$$h(\mathcal{M}, \mathcal{SV}) \leq \frac{5\sqrt{3}}{n}, \tag{3.5}$$

given that the sweep is scaled to fit into the $[0, 1]^3$ cube.

We demonstrate the capabilities of the method with scenarios from academia as well as real world scenarios. For the GPU version, we used a machine with an Intel Core i7 at 3.20GHz and 12GB RAM, under Ubuntu Linux using the GNU C++ compiler v4.4 with optimizations (`-O3`) and a NVIDIA GeForce GTX 480 with 1536MB RAM with Cuda version 3.2. In the first real world scenario, the movement of a steering gear relative to a neighboring engine part unit was recorded during a test-drive. The second is an engine assembly path that was planned with DMU. The *engine* scenario as well as the scenarios *bunny* and *dragon* describe significant[5] motions, whereas the *steering* scenario describes a vibration. The bunny scenario is depicted in Figure 6.1, the dragon is depicted in Figure 3.14, the *engine* in Figure 4.15 and the *steering* in Figures 4.14 and 3.15.

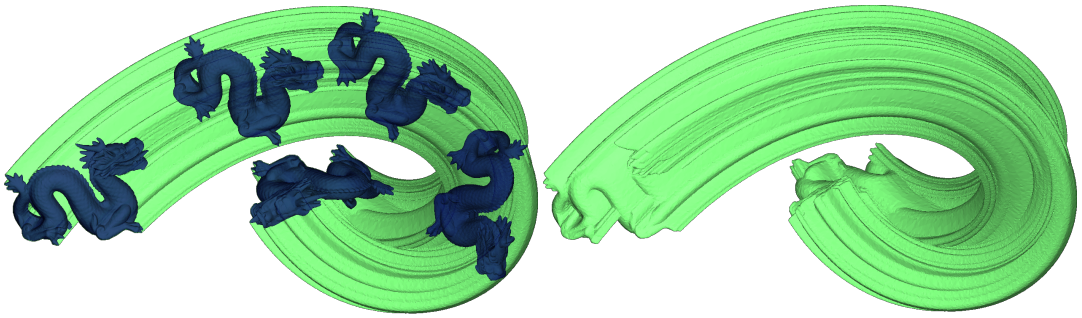

Figure 3.14: Model of a dragon (800k triangles) swept along a trajectory consisting of 129 transformations.

Since the depth buffer image generation phase and the subsequent mesh extraction phase are independent from each other, we split the timings and present them in Tables 3.1 and 3.2. Corresponding plots are shown for the depth buffer images generation in Figure 3.16 and for the meshing phase in Figure 3.17.

To interpret the plots in Figure 3.16, let us shortly recapitulate which geometry is rendered to the depth buffers and how far this geometry is dependent on the resolution n.

[5]We distinguish between significant motions and vibrations, in the sense that every motion that is not a vibration, is a significant motion.

model		trajectory	resolutions				
name	N	m	2^{10}	2^{11}	2^{12}	2^{13}	2^{14}
bunny	8100	129	3.27s	4.40s	8.10s	23.3s	93.7s
dragon	871k	129	353s	402s	521s	778s	1390s
engine	328k	191	192s	203s	239s	350s	724s
steering gear	261k	27k	5.21h	5.37h	6.19h	8.47h	14.4h

Table 3.1: Results for depth buffer images generation.

scenario		resolutions				
name		2^{10}	2^{11}	2^{12}	2^{13}	2^{14}
bunny		2.91s	9.94s	36.5s	137s	522s
	output complexity	14.3k	36.0k	87.9k	219k	549k
dragon		3.77s	13.6s	49.8s	177s	645s
	output complexity	26.1k	82.6k	235k	590k	1.40M
engine		3.65s	14.8s	61.1s	234s	924s
	output complexity	27.6k	106k	392k	1.25M	3.86M
steering gear		2.46s	8.74s	31.6s	117s	435s
	output complexity	15.0k	47.2k	142k	414k	1.17M

Table 3.2: Results for Delaunay mesh extraction.

In the depth buffer generation phase the workload (for high resolutions) lies in the fragment shader programs, since the geometry and vertex shader computations as well as the Cuda vertex buffer interoperation are independent of depth buffer resolution. Hence, if we just rendered the triangles of $\mathcal{SV}$ to the depth buffers (without offsetting), we would expect a growth of running time that is linear in the number of produced fragments, which again, is proportional to the number of pixels in the depth buffers (also called dexels). The number of pixels in the depth buffers is, of course, n^2 for each buffer.

However, we are rendering offset geometry with an offset radius proportional to $1/n$, which means, that the offset shrinks with higher resolutions. Thus we can expect the number of fragments (and running time) to grow less than quadratic in n, or equivalently, less than linear in the dexel count. This expected behavior

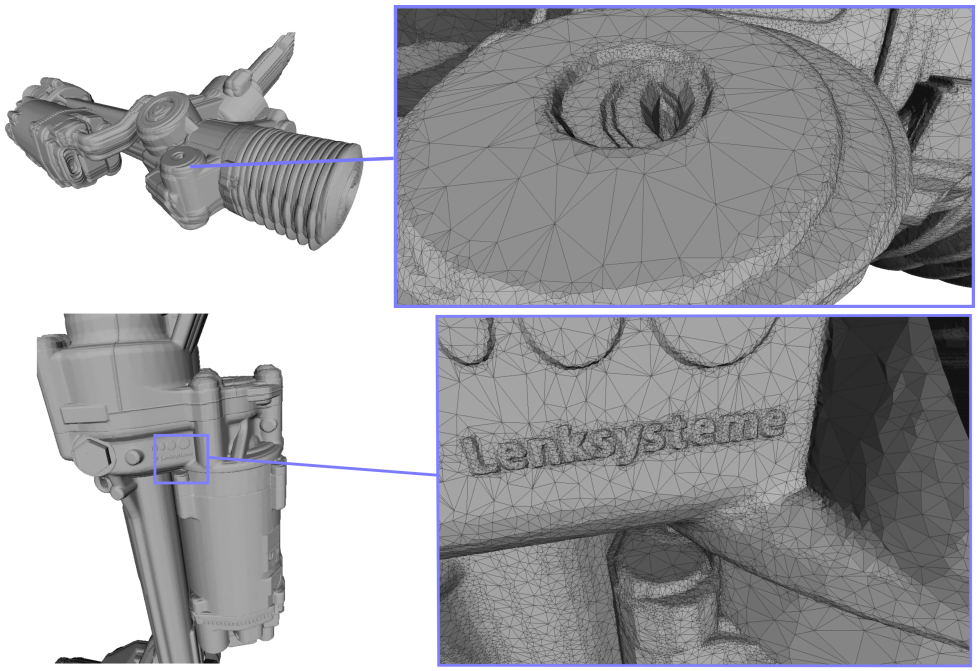

Figure 3.15: Resulting SV mesh of the steering gear scenario with the depth buffer based approach at the highest resolution ($n = 2^{14}$).

is reflected in the plots in Figure 3.16.

The fragments that are generated by the offset cylinders of edges and the offset balls of vertices follow shader code paths that are far more costly than the path followed by the fragments produced by the shifted triangles. Hence, the sub-linear growth behavior is even more noticeable for highly tessellated meshes, e.g., the dragon model.

The timings for the mesh extraction phase in Table 3.2 show similar running times and output triangle counts for the different scenarios at a fixed resolution, which is reasonable, since all information on the geometry is stored in the depth buffer images. The thickness of the tolerance hull ε is linear in $1/n$, we therefore expect a quadratic dependency of running time as well as output triangle count on n, or equivalently, we expect a quadratic dependency of running time as well as output triangle count on the inverse hull thickness ε.

The plots in Figure 3.17 show this quadratic dependency, since the axes have a logarithmic scaling and the data points of each scenario can be tightly interpolated with a line that has a slope of 2.

3.5 Conclusion for the Depth Buffer Based Approach

In this chapter, we have presented an implementation of the concept presented in Chapter 2. The main idea is to approximate the volumetric representation with six directed distance fields (depth buffers), which drastically reduces memory consumption.

Of course, this comes with the major drawback that only those parts of the geometry are present in the final approximation that were already present in the depth buffers in the first place. Unfortunately, this also means that the error bounds only hold in these regions as well, and therefore are not global, indicating that the resulting mesh might lack concavities in some areas, e.g., Figure 5.3. We present in Chapter 5 a method to detect these areas a posteriori.

However, since the voxel set in the depth buffers is a superset of the actual conservative voxelization, the conservative demand is met!

Although there might be applications where missing concavities are not acceptable, for the case of a vibration this should not be a great cutback. Moreover, the meshes we receive from the car manufacturer have internal geometry removed, such that only those triangles are kept that can be seen from one of the six major axes directions (positive and negative). For vibrations, this is a perfect match to our approach, since we align the depth buffers to the same axes.

As no requirements are posed to the input generator, the approach can cope with every type of malformed triangle meshes. Furthermore the sweeping phase makes heavy use of the GPU and we can expect it to scale well with future hardware generations.

With this implementation we can achieve very high voxel resolutions, which in turn, leads to very high approximation quality. This can be seen in Figure 3.15

for the case of the vibrating *steering* gear. The magnifications show very fine details caused by the generator at some peak of movement.

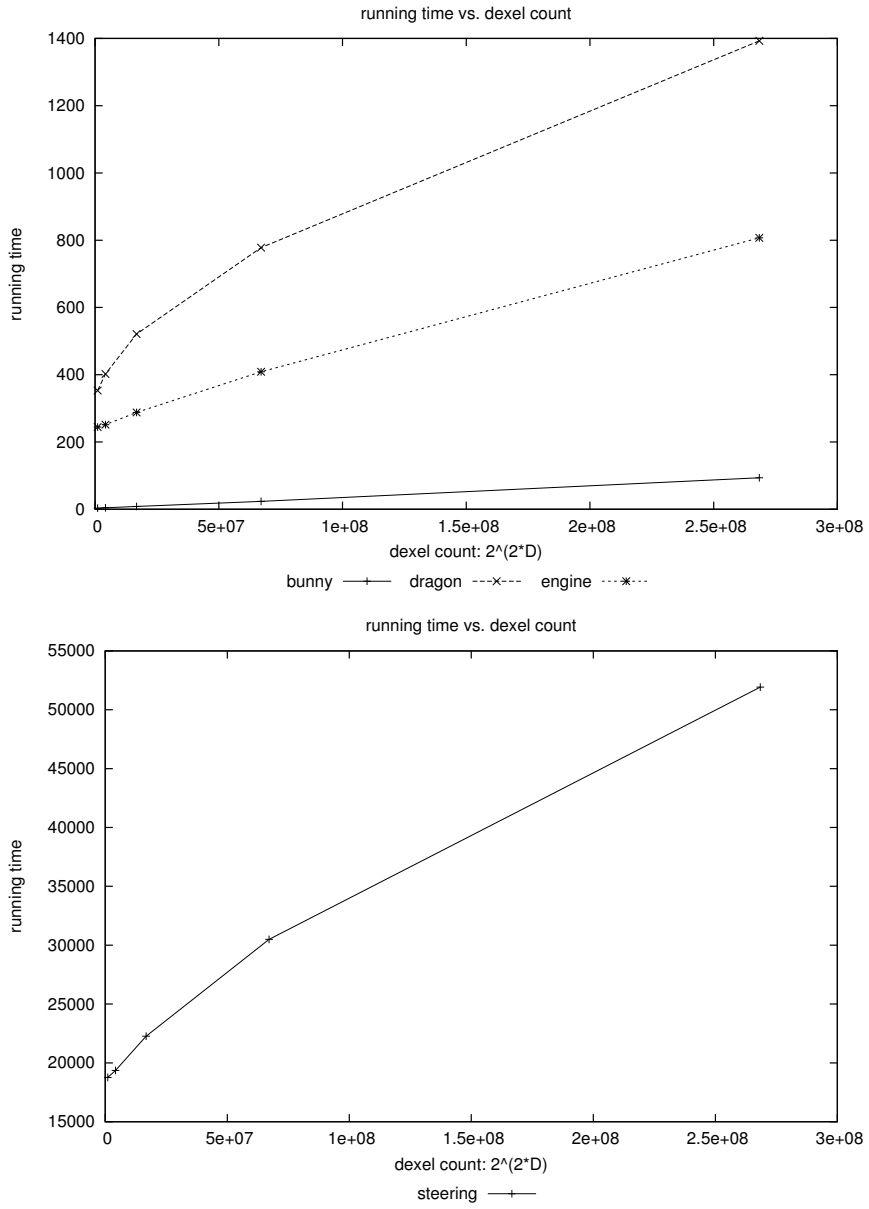

Figure 3.16: Depth buffer generation phase: Running time (in seconds) is plotted against the number of fragments in the depth buffer (dexel count) for the *bunny, dragon* and *engine* scenario (top) and for the *steering* scenario (bottom).

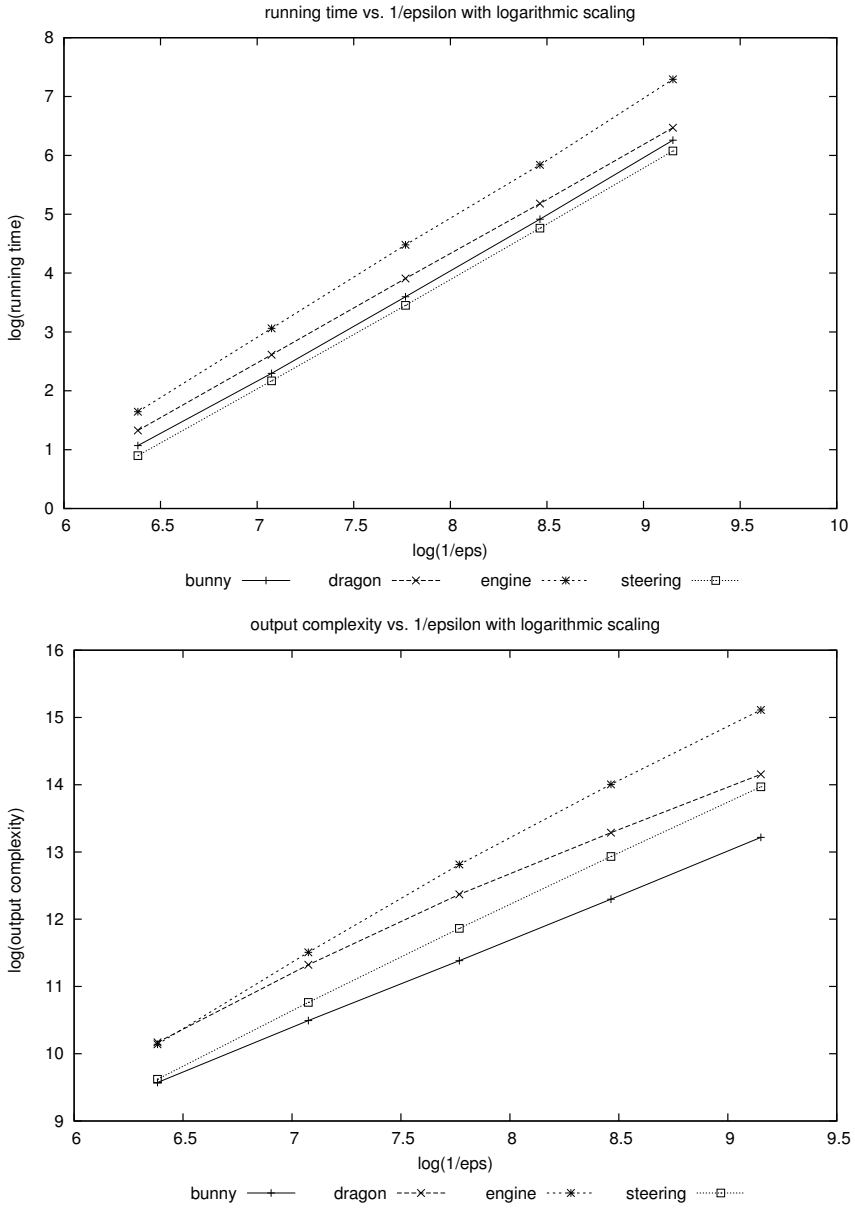

Figure 3.17: Meshing phase: Running time (top) and mesh output complexity (bottom) in logarithmic scaling are plotted against inverse hull thickness $1/\varepsilon$ in logarithmic scaling.

Chapter 4

An Octree Based Concept Implementation

In this chapter we present a second implementation of the concept presented in Chapter 2.

Similar to the depth buffer based approach, we store the volumetric approximation of the swept volume in form of a voxelization. In contrast to the depth buffer images, we utilize an octree to reduce memory consumption.

We transform the triangles of $\mathcal{SV}$ into a conservative voxelization and store the resulting voxels in an incrementally growing octree data structure in CPU memory. The offset hull is represented via additional voxel layers on the outer boundary of the initial voxelization.

Similar to the method presented in the previous chapter we distinguish between a sweeping phase and a subsequent meshing phase, for the former, we provide parallel implementations for GPUs as well as for multi-core CPUs. Clever memory management and heuristic adjustments of the octree enable us to significantly reduce time and memory consumption.

In order to reduce the number of triangles that actually need to be considered, we use a culling idea similar to that presented by Campen and Kobbelt [12]. A detailed discussion of the culling can be found in Section 4.1.

However, even if this process would be able to select only those triangles that actually contribute to the boundary of $\mathcal{SV}$, it would still be impossible to keep all of those triangles in memory at once. Instead, we immediately compute a voxelization for the required precision of the non-culled triangles as they appear and the resulting voxels are inserted into the octree. When memory becomes

scarce, we temporarily stop the sweep in order to fill up the volume that has so far been swept which further reduces the size of the octree. A full presentation of the sweeping phase, in particular of the parallelized versions for multi-core and Cuda, is given in Section 4.2.

In Section 4.3, we will adjust the concept of restricted Delaunay mesh generation for swept volumes, as presented in Section 2.2, to our octree data structure.

Practical results in Section 4.4, a conclusion in Section 4.5 and acknowledgements in Section 4.6 follows.

4.1 Polygonal Superset Computation

We are only interested in the outer boundary of $\mathcal{SV}$, which a lot of triangles of $\mathcal{SV}$, as defined in Section 2.3, do not contribute to. Hence, we follow a culling heuristic similar to the one proposed in [12] that tries to rule out those triangles (of the deformed prisms) that do not contribute to the boundary. This is based on local criteria of the mesh in relation to the current direction of the motion.

4.1.1 Culling of Facets

As mentioned earlier a triangle $T_{(a,b,c)}$ with vertices $a, b, c \in \mathbb{R}^3$ at two consecutive transformations $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$ forms, together with its interpolated edges, a deformed prism.

Each triangle of a deformed prism that will (at least in part) contribute to the outer boundary of $\mathcal{SV}$ has to be reached from the outside. As stated in [12], a sufficient condition to cull a triangle is that both its sides are covered by the prisms of the previous and following time steps, as depicted in Figure 4.1 on the left.

We here use a culling variant with the following observation: If the normal of the triangle points into the prism formed with same triangle at either the next time step or (non-exclusive) into the prism formed with same triangle at the previous time step it can impossibly be reached from the outside.

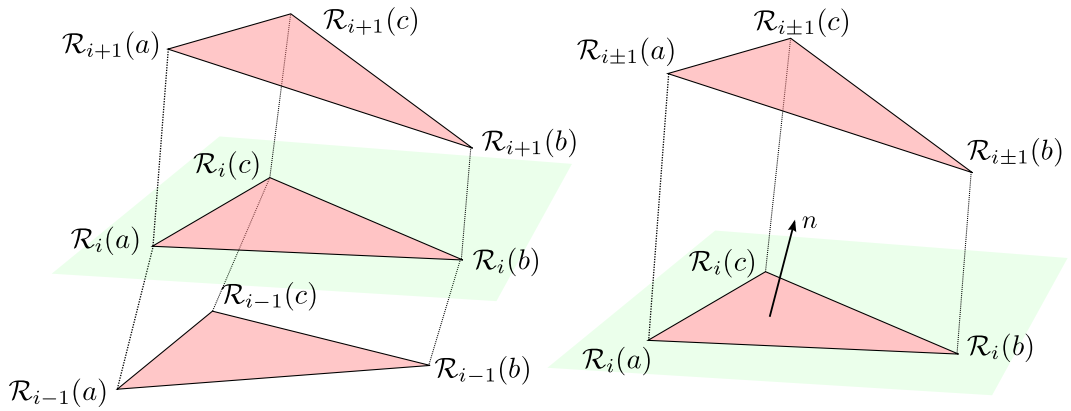

Figure 4.1: Two culling variants for facets. Left: Triangle has to be covered by prisms from both sides. Right: It suffices that only front side of triangle is covered (taking the normal into account).

Therefore, we can safely cull a triangle if at least one of the following conditions hold:

- The transformed vertices

$$\mathcal{R}_{i+1}(a), \mathcal{R}_{i+1}(b) \text{ and } \mathcal{R}_{i+1}(c)$$

  all lie on the side of the supporting plane of T, that the normal points to.

- The transformed vertices

$$\mathcal{R}_{i-1}(a), \mathcal{R}_{i-1}(b) \text{ and } \mathcal{R}_{i-1}(c)$$

  all lie on the side of the supporting plane of T, that the normal points to.

Both cases are depicted in Figure 4.1 on the right.

Note that the second variant rules out more triangles (hence is better) than the first, but requires the generator mesh to have meaningful normals.

4.1.2 Culling of Edge-patches

Following the proposition in [12] we tessellate each patch by choosing the diagonal with the lower dihedral angle. For example: Given the edge $e_{a,c}$ of a triangle

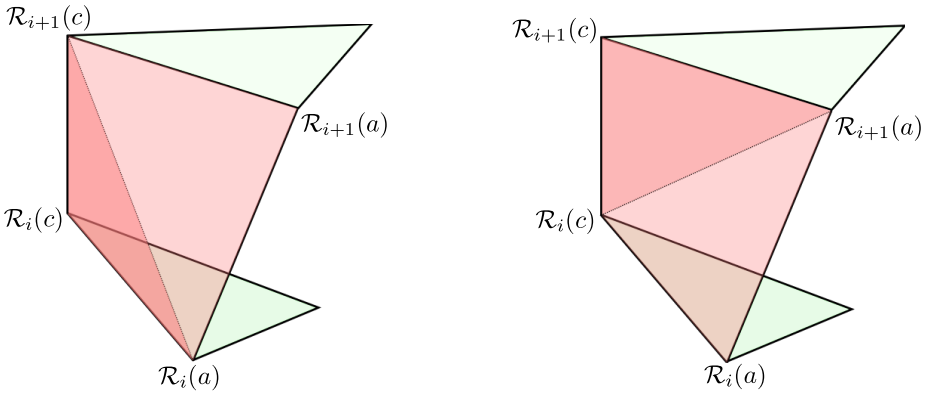

Figure 4.2: For each face there are two possible tessellations, yielding 2 different triangle pairs. The pair of triangles whose normals have the smaller angle is chosen.

$T_{(a,b,c)}$ under two consecutive transformations $\mathcal{R}_i, \mathcal{R}_{i+1}$, as depicted in Figure 4.2, we have the following two candidate tessellations:

$$T_1 := T_{(\mathcal{R}_i(a),\mathcal{R}_{i+1}(c),\mathcal{R}_i(c))} \cup T_2 := T_{(\mathcal{R}_i(a),\mathcal{R}_{i+1}(a),\mathcal{R}_{i+1}(c))}$$

with normals

$$n_1 = \frac{(\mathcal{R}_i(a) - \mathcal{R}_i(c)) \times (\mathcal{R}_{i+1}(c) - \mathcal{R}_i(c))}{\|(\mathcal{R}_i(a) - \mathcal{R}_i(c)) \times (\mathcal{R}_{i+1}(c) - \mathcal{R}_i(c))\|_2}$$

$$n_2 = \frac{(\mathcal{R}_{i+1}(c) - \mathcal{R}_{i+1}(a)) \times (\mathcal{R}_i(a) - \mathcal{R}_{i+1}(a))}{\|(\mathcal{R}_{i+1}(c) - \mathcal{R}_{i+1}(a)) \times (\mathcal{R}_i(a) - \mathcal{R}_{i+1}(a))\|_2}$$

as well as

$$T_3 := T_{(\mathcal{R}_i(c),\mathcal{R}_i(a),\mathcal{R}_{i+1}(a))} \cup T_4 := T_{(\mathcal{R}_i(c),\mathcal{R}_{i+1}(a),\mathcal{R}_{i+1}(c))}$$

with normals

$$n_3 = \frac{(\mathcal{R}_{i+1}(a) - \mathcal{R}_i(a)) \times (\mathcal{R}_i(c) - \mathcal{R}_i(a))}{\|(\mathcal{R}_{i+1}(a) - \mathcal{R}_i(a)) \times (\mathcal{R}_i(c) - \mathcal{R}_i(a))\|_2}$$

$$n_4 = \frac{(\mathcal{R}_i(c) - \mathcal{R}_{i+1}(c)) \times (\mathcal{R}_{i+1}(a) - \mathcal{R}_{i+1}(c))}{\|(\mathcal{R}_i(c) - \mathcal{R}_{i+1}(c)) \times (\mathcal{R}_{i+1}(a) - \mathcal{R}_{i+1}(c))\|_2}$$

If $n_1^T n_2 > n_3^T n_4$ we chose $T_1 \cup T_2$, otherwise $T_3 \cup T_4$.

Every manifold edge $e_{a,b}$ with vertices a, b is shared by exactly two triangles $T_{(a,c,b)}$ and $T_{(a,b,d)}$, see Figure 4.3. We will call these triangles **wings** of the edge and c and d **wing-tips** of the edge in the following.

We now want to examine an edge $e_{a,b}$ and its wings under two consecutive transformations: $\mathcal{R}_i$ and $\mathcal{R}_{i+1}$, as depicted in Figure 4.3. W.l.o.g. we assume $e_{\mathcal{R}_i(a),\mathcal{R}_{i+1}(b)}$ is the diagonal with the lower dihedral angle than $e_{\mathcal{R}_i(b),\mathcal{R}_{i+1}(a)}$. Hence, the candidate triangles are

$$T_1 := T_{(\mathcal{R}_i(a),\mathcal{R}_i(b),\mathcal{R}_{i+1}(b))}$$

and

$$T_2 := T_{(\mathcal{R}_i(a),\mathcal{R}_{i+1}(b),\mathcal{R}_{i+1}(a))}.$$

The tessellated patch $T_1 \cup T_2$ will not contribute to the outer boundary of $\mathcal{SV}$ if it is not reachable from the outside. This will be the case if both triangles T_1 and T_2 are occluded by the volumes swept by the wings of edge $e_{a,b}$. Thus, we can safely cull the patch if the following three conditions hold:

- The transformed wing tip $\mathcal{R}_i(c)$ and $\mathcal{R}_{i+1}(c)$ are on the same side of both, the supporting plane of T_1 and the supporting plane of T_2.
- The transformed wing tip $\mathcal{R}_i(d)$ and $\mathcal{R}_{i+1}(d)$ are on the same side of both, the supporting plane of T_1 and the supporting plane of T_2.
- The transformed wing tips $\mathcal{R}_i(c)$ and $\mathcal{R}_i(d)$ are on opposite sides of the supporting plane of T_1 and on opposite sides of the supporting plane of T_2.

Our test slightly differs from the one proposed in [12], since the authors only test one plane and do not state, how this is chosen in the general case, when T_1 and T_2 are not coplanar.

In general this method does not require the input triangle mesh to meet any topological requirements. However, for the local culling criteria to work properly (which is crucial to running time), most edges of the triangle mesh should be manifold, i.e., having exactly two neighboring triangles (wings). Patches emerging from swept non-manifold edges cannot safely be ruled out and all have to be voxelized.

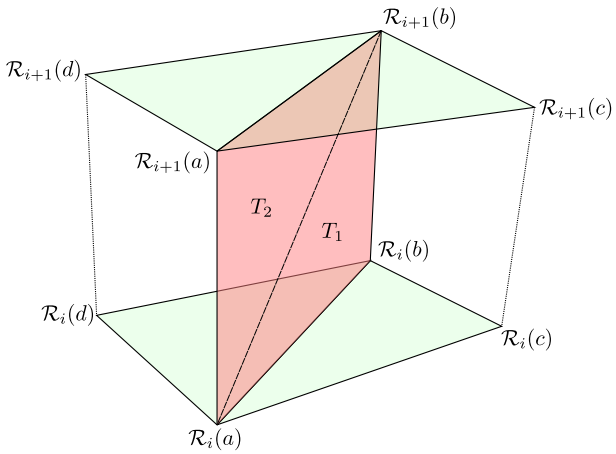

Figure 4.3: If the tessellated patch is covered by prisms from both sides it can safely be culled.

4.2 Octree Based Voxelization

An overview of the overall algorithm is given in Figure 4.4. Each step of the main loop corresponds to one discrete time step of the given trajectory $(\mathcal{R}_i)_{i=1,\dots,m}$. For each triangle (of each prism) we first check whether it may contribute to the outer boundary using the culling criteria discussed in Section 4.1. If the triangle is not culled, it is immediately voxelized and only the resulting voxels are stored.

Ideally, this would result in only those voxels that are required to cover the boundary of $\mathcal{SV}$. However, the culling is not perfect since it can only apply local criteria. As a consequence, a lot of inner voxels are occupied. Thus, in the worst case, the size of this set is proportional to the volume of $\mathcal{SV}$ and thus cubic in n.

In order to reduce memory usage it is rather common to organize such a voxelized volume in an octree [13, 53, 41]. Unfortunately, this is not directly applicable since a lot of voxels are not covered (we are only voxelizing the triangles and not the full prisms). Thus, if memory becomes scarce, we stop the actual sweep and compress the data structure by filling up the volume that was swept thus far using a special hierarchical method, see Section 4.2.1. Note that this may

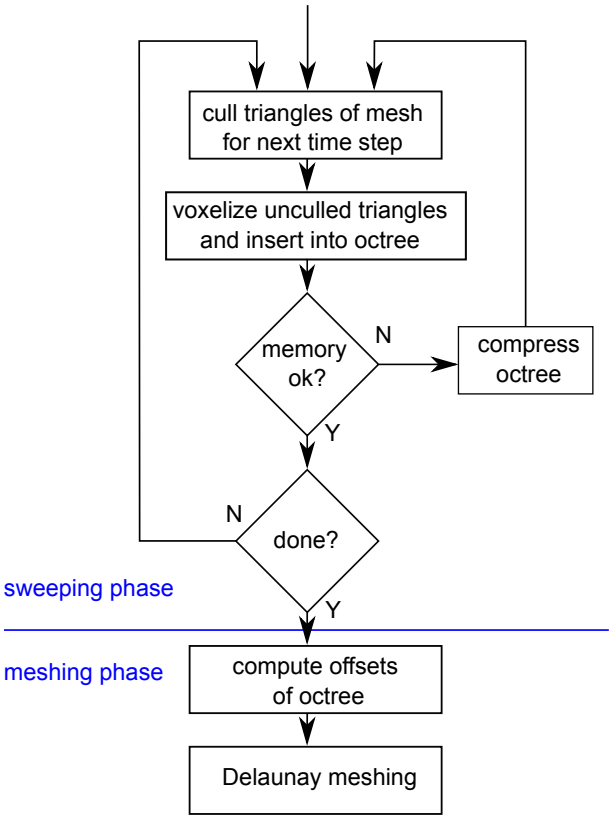

Figure 4.4: Overview of our approach (non-parallel).

also fill inner holes of $\mathcal{SV}$, which is perfectly fine, as we are only interested in the outer boundary.

Voxelization

The actual voxelization of each triangle is based on a simple recursive subdivision scheme. Starting from the initial $[0,1]^3$ cube, the cubes that still intersect the triangle are recursively subdivided into eight smaller cubes of equal size until the required resolution n is reached at the final recursion depth D, with $n = 2^D$.

As an intersection test, we use the separating axis theorem as it has been used, for

instance, in [6]. However, the code is explicitly designed to take advantage of the fact that all cubes share the same orientation. This implies that all 13 separating axes between the cubes and the triangle can be pre-calculated. Since only the positions and sizes of the cubes vary during the intersection computations, the necessary arithmetic overhead can be significantly reduced.

We note that the above method generates a minimal-conservative voxelization as defined in Section 2.3, (2.6).

Octree

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

4.2.1 Compression

The voxelization of the non-culled triangles yields a set of voxels $\mathcal{V}$ stored in an octree. Since the culling is only based on local criteria, internal geometry might still be present and therefore the octree has internal voxels which can lead to a memory consumption of $O(n^3)$. Thus, if we detect that memory becomes scarce at step j, we stop the sweeping and voxelize the triangles of the generator in its current position $\mathcal{R}_j(\mathcal{G})$. This is required so that the outer layer of $\mathcal{V}$ is a conservative voxelization of the boundary of the part of the sweep: $(\mathcal{R}_i(\mathcal{G}))_{i=1,\dots,j}$. This is depicted in Figure 4.5

In order to compress the octree, we now strive to fill up the interior holes of $\mathcal{V}$, which results in an octree whose size corresponds to the area occupied by the outer boundary of $\mathcal{V}$, which should be $O(n^2)$ for reasonable volumes.

[1] More precisely, we use a `boost::unordered_set`; `www.boost.org`.

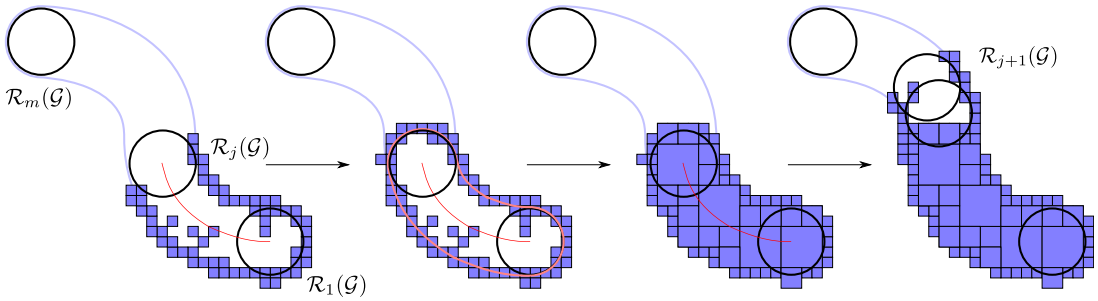

Figure 4.5: When memory becomes scarce, the sweep is stopped and the octree is filled to reduce memory consumption. Then the sweep continues.

A common approach to filling the holes of $\mathcal{V}$ is 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 $\mathcal{V}$. For this, we start from a voxel v_{out} that we know is not in $\mathcal{V}_0$. Then we move towards one of the voxels in $\mathcal{V}_0$ until we encounter the first voxel in $\mathcal{V}_0$. The last free voxel on the path is a voxel of the hull which starts the crawling. The crawling is implemented as a breadth first search (BFS) and the hull voxels are stored in a hash set. This is depicted in Figure 4.6. This results in a layer of voxels coating the outer boundary of $\mathcal{V}_0$.

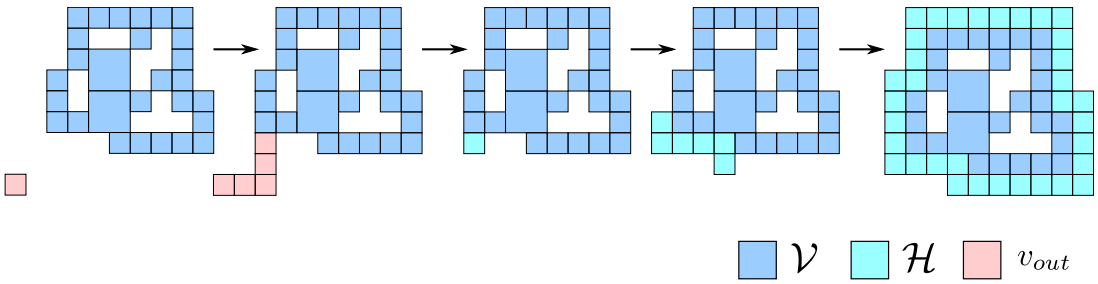

Figure 4.6: We move from an outside voxels towards $\mathcal{V}_0$. The last visited voxel starts the crawling.

This hull is then *flooded*, starting from an initial voxel (the **seed**) and recursively exploring all its neighbors that are not in $\mathcal{H}$ and storing them in the octree $\mathcal{V}_0$. Note that this guarantees exploration of every part of a conservative(!) $\mathcal{SV}$ voxelization as long as the generator geometry itself is connected. However, the

naive approach would touch all voxels in the volume and would take $O(n^3 \log n)$ time.

We here present a new filling method that carefully applies a hierarchical scheme without violating any of the above guarantees.

The hull $\mathcal{H}$ is itself represented as an octree with depth D representing a thin layer of voxels coating $\mathcal{V}$. We now coarsen $\mathcal{H}$ by applying the following rule. Every direct parent of a leaf of $\mathcal{H}$ 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 space of the actual volume $\mathcal{V}$. The parts of $\mathcal{V}$ that are left open on the coarse layers are exactly those that we want to fill first in the subsequent filling step. The idea of the hierarchical filling is depicted in Figure 4.7

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 coarse the given seed. In case the seed gets occluded by the inflating hull, we try to rescue the seed by moving it to a neighboring voxel. However, at some point the seed will die. At this point we start the flooding on the most coarse level that we could reach. We then use the resulting volume as seed for the next finer layer and so on. Thus, in the case that we were lucky, i.e., the seed did not die too early, we can expect that we only touch a quadratic number of cells, i.e., a running time of $O(n^2 \log n)$

In order to increase the success rate of this heuristic, and since the volume may also have several disconnected areas on coarse levels, we simply maintain several seeds simultaneously. In fact, this resulted in very reasonable running times for all tested scenarios.

An example of a hierarchical filling is depicted in Figure 4.8.

4.2.2 Parallelization Multi-core

The voxelizations of different triangles are tasks that obviously can be spread among several processors. For the multi-core version, each step $[\mathcal{R}_i, \mathcal{R}_{i+1}]$ of the trajectory is assigned to another core. This includes the actual voxelization as well as the culling of all generated triangles in this track. The resulting voxels are buffered in separate hash sets, each dedicated to one thread. However, since

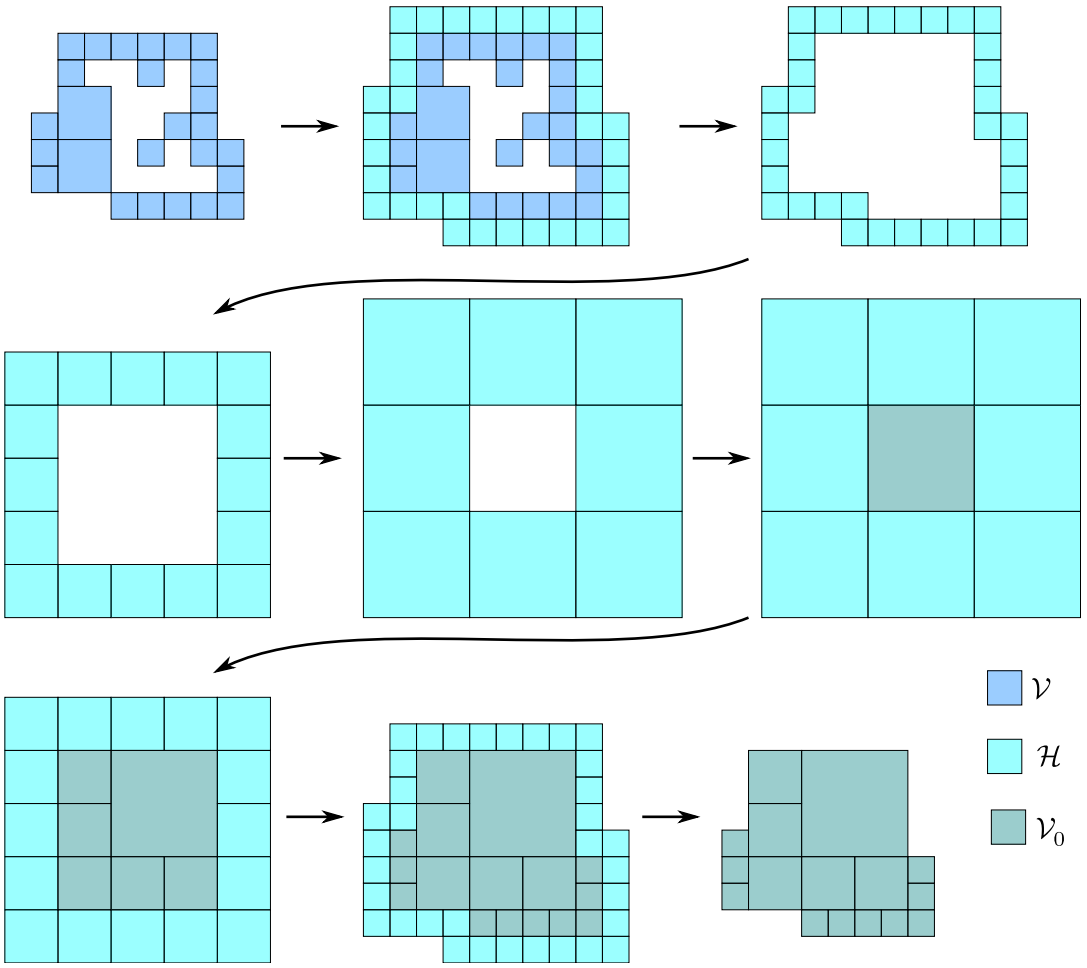

Figure 4.7: Hierarchical filling of an octree in order to reduce memory consumption.

most voxels are generated several times, it is not an option to keep all voxels in those buffers until the end since this would essentially multiply the memory consumption by the number of threads.

Therefore, we keep one separate (master) thread that inserts already generated voxels into the main octree. Since each thread can access one buffer at a time, we keep $2(t-1)+1$ buffers, where t is the total number of threads. That is,

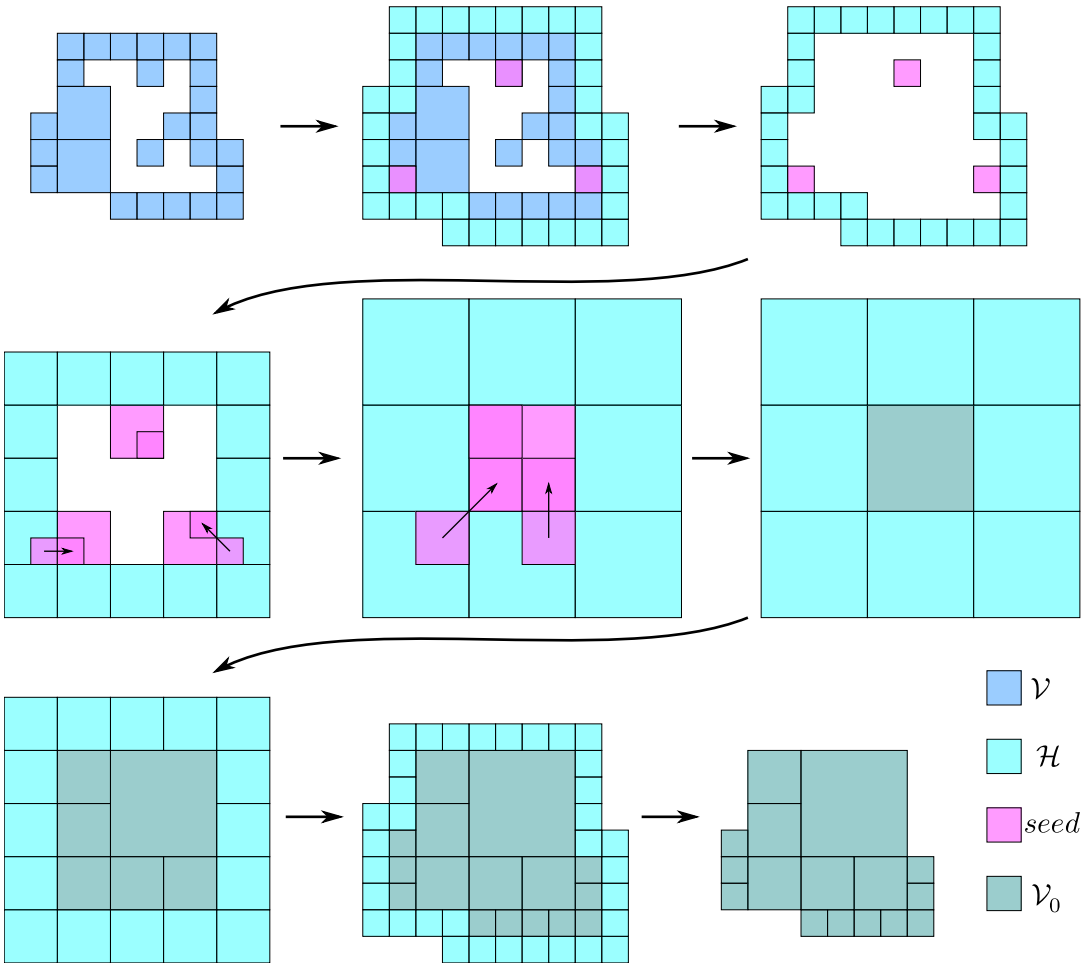

Figure 4.8: Hierarchical filling of an octree with seeds inside.

each thread has actually two buffers. One that it is currently being used to push voxels, while the other contains voxels from the last step and that are ready to be taken by the master thread. In turn, the master thread takes such a buffer just by swapping it with its own buffer and then, empties it by inserting all voxels into the main octree. This way, the interaction among threads (and thus time loss due to locks) is minimal, since it is essentially limited to the time required for the swapping which is implemented as a constant time operation.

Voxel Filter

The above mentioned scheme is very efficient since it keeps all threads constantly busy. A thread can take over the next step in the trajectory once it is done with its last one. Almost, no synchronization among threads is required. In fact, the actual bottleneck is not the generation of voxels but the insertion into the main octree, in particular, the time spent for memory allocation and organizing the hash set. Also, for each thread, the insertion of voxels into the buffers and the associated time for memory allocation contributes significantly to the consumed time.

In an optimal scenario, we would be able to rule out voxels that are already occupied by the main octree before we even put them into a buffer. However, the master thread is constantly changing this octree which blocks the access for the other threads. Therefore, we stop all threads (including the master thread) to make a copy of the main octree from time to time. Since this copy does not change, it can be used in a read only fashion by all threads in parallel for culling most useless voxels. Therefore, every time before a thread inserts a voxel into its local buffer, its is first checked for containment in the octree copy. If it is already present, it can be discarded.

The voxel filter concept is depicted in Figure 4.9.

Vibration Detection

The voxel filter becomes crucial if the motion remains rather local, as it is the case for vibrations. In this scenario, the amount of generated voxels that can be culled is significant and only a small fraction reaches the master thread. In order to boost this effect, we add the following heuristic: If less than one percent of the voxels of a buffer which was inserted by the master thread was not yet in the main octree we suspect that the motion is a vibration (since other threads already produced very similar voxel sets). In this case we even trigger a compression of the octree, that is, we fill up the volume even if the memory limit has not been. The reward for this vibration detection, despite the additional costs, is that many more voxels can be culled in the remaining runs. Moreover, the representation of the octree becomes much more compact at an early stage.

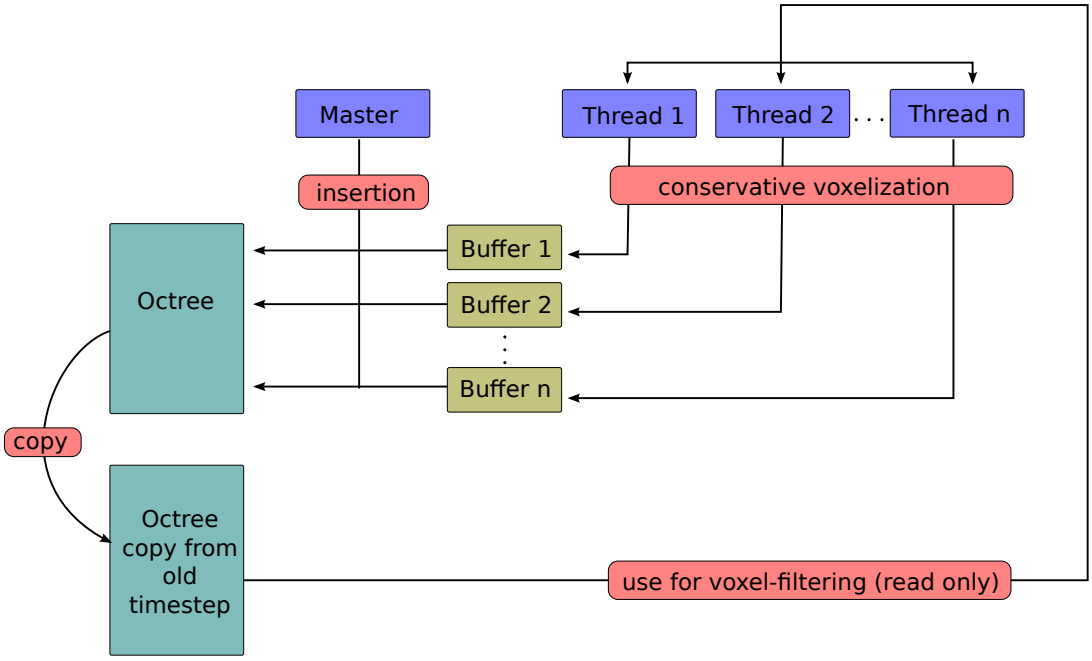

Figure 4.9: Diagram showing the concept of the voxel filtering by the parallel threads.

In order to ensure a compression is not triggered too often, we only allow a compression if the time spent for voxel generation exceeds 10 times the time spent for compressions so far. This way, the time spent in compression is usually, at most, 10% of the running time.

4.2.3 Parallelization GPU

Our implementation of the voxelization algorithm on the GPU is straightforward. Each triangle (emerging either from a swept edge or a facet) is completely voxelized (down to the required depth D) by one thread. For this, each thread has 2 stacks (called working stack and output stack) capable of storing axis aligned cubes. The voxelization is based on a simple subdivision scheme. Starting from the initial $[0,1]^3$ cube, all of the 8 sub-cubes that still intersect the triangle are

put onto the working stack. While the working stack is non-empty, the cube on top is taken off the stack, subdivided and again all of its 8 sub-cubes that still intersect the triangle are put onto the working stack. If a cube has the desired dimensions, i.e., 2^{-D} (thus being a voxel) it is written to the output stack. The intersection test is again based on the separating axes theorem tailored to the special case of axis aligned cubes. As we are refining the voxelization in a depth first manner, it is easy to see that the working stack of each thread cannot grow larger than $8 \cdot D$ (The worst case being all sub-cubes have to be put onto the working stack for each depth, which cannot happen for triangles). Determining the size of the output stack for each thread is more challenging. We use an intuitive heuristic based on the size of each triangle: We contemplate the ρ-offset of a triangle T,

$$\mathcal{O}_\rho(T) = \{x \in \mathbb{R}^3 \,\big|\, \|x - T\|_2 \leq \rho\}.$$

It is easy to see that an upper bound of the voxels occupied by a triangle N_T can be found if one chooses ρ to be the voxel diagonal: $\rho = \frac{\sqrt{3}}{n}$ and divides the volume of $\mathcal{O}_\rho(T)$ by the volume of a single voxel:

$$N_T < \frac{\text{vol}(\mathcal{O}_\rho(T))}{(1/n)^3}.$$

This is true, since all voxel are disjunct and every voxel, that intersects the triangle, must be completely occupied by $\mathcal{O}_\rho(T)$. The volume $\mathcal{O}_\rho(T)$ can be conservatively estimated by adding the volumes of the 3 spheres, the 3 solid cylinders and the prism, cf., Section 3.1.1.

The computation is done on the GPU in a kernel prior to the actual voxelization kernel. Its computational time can be neglected compared to the voxelization kernel.

Instead of voxelizing each triangle that could not be culled directly, as done in the non-parallel algorithm, now chunks of triangles are collected and voxelized on the GPU in a single step. Before downloading the voxels of the whole chunk of triangles back to CPU memory, we apply a unification using thrust[2] to reduce the number of voxels that have to be inserted into the octree.

Due to the great amount of memory used when demanding high resolutions, a sparse representation of a cube is crucial. Storing the integer coordinates of a

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

cube in only one `int` (with 32 bit), only allows resolutions up to 2^{10}. Since this is not enough, we encode each cube into two integers. On Cuda we used `int2`:

$$\underbrace{\underbrace{|\text{ x-coord }|\text{ y-coord }|}_{\texttt{int}}\underbrace{|\text{ z-coord }|\text{ depth }|}_{\texttt{int}}}_{\texttt{int2}}$$

thus allowing for resolutions up to 2^{16}. For example, the $[0,1]^3$-cube:

$$|0\ldots0|0\ldots0||0\ldots0|0\ldots01|$$

would be divided into the 8 sub-cubes:

$$|0\ldots0i|0\ldots0j||0\ldots0k|0\ldots010|,$$

for $i,j,k \in \{0,1\}$.

The overall parallelization concept for the GPU is depicted in Figure 4.10

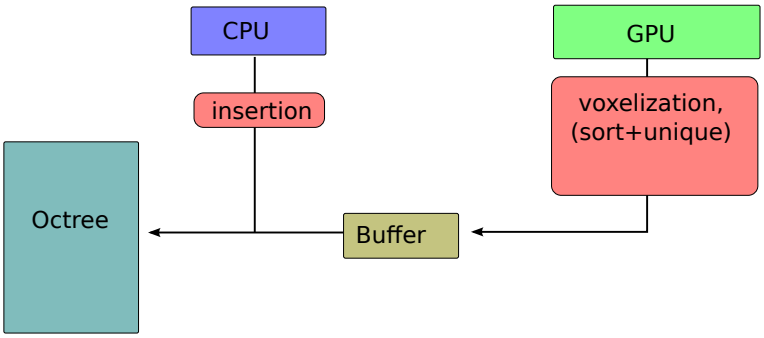

Figure 4.10: Diagram showing the parallelization concept for the GPU.

4.3 Voxel Based Delaunay Mesh Generation

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 $\mathcal{D} := \mathcal{V}_1$ in the notation of Section 2.2.

More precisely, we introduce two new predicates that declare a facet as bad if

$$f \cap \mathcal{V}_0 \neq \emptyset \text{ or } f \cap \mathcal{V}_2 \neq f,$$

respectively. This means that the voxel offset $\mathcal{V}_1$ takes over the role of the continuous offset $\mathcal{O}_\delta(\mathcal{V}_0)$ and that the surface, we restrict the Delaunay refinement to, is $\partial\mathcal{V}_1$. The tolerance hull is then $\mathcal{V}_2 \setminus \mathcal{V}_0$.

The first predicate obviously ensures conservativeness while the second allows us to bound the one-sided 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 $\frac{2\sqrt{3}}{n}$, i.e., two voxel diagonals (see also Figure 4.11).

Since $\mathcal{V}_0$ is a minimal-conservative voxelization of $\mathcal{SV}$, we have the desired conservativess

$$\mathcal{M} \cap \mathcal{SV} = \emptyset$$

and with Lemma 5 the bound on the one-sided Hausdorff distance

$$h(\mathcal{M}, \mathcal{SV}) \leq \frac{3\sqrt{3}}{n}. \tag{4.1}$$

4.4 Experimental Results of the Octree Based Approach

We use the same real world scenarios as in Section 3.4, namely:

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

For both scenarios, the expected complexity n of the generator is rather high, for instance, a mesh representing the main engine of a car or truck. However, in the vibration scenario, also the complexity m of the trajectory is very high. In the assembly scenario the trajectory is less complex but represents a significant motion.

		input complexity		output complexity (16-CPU)		
scenario	type	n	m	$D = 10$	$D = 11$	$D = 12$
bunny	motion	8100	129	23.7k	60.3k	157k
dragon	motion	871k	129	45.7k	147k	420k
engine	motion	328k	191	59.0k	252k	1.34M
steeringgear	vibration	261k	27k	28.7k	91.6k	279k

Table 4.1: Type and complexity of different scenarios.

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

Table 4.2: Total running time (in seconds) for different scenarios in different resolutions comparing GPU with Multi-CPU solution.

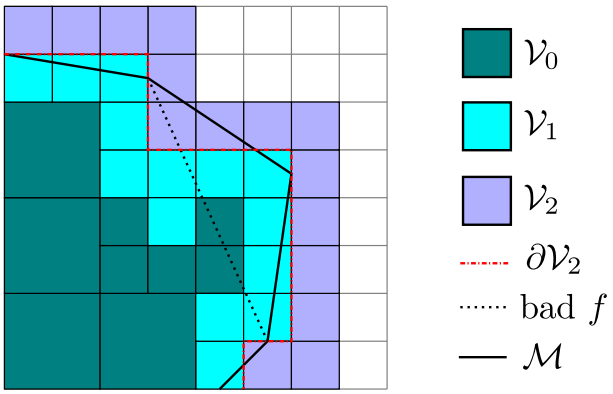

Figure 4.11: A two-dimensional illustration of the voxelization $\mathcal{V}_0$ organized as an octree and its two offsets $\mathcal{V}_1$ and $\mathcal{V}_2$. Vertices of the mesh $\mathcal{M}$ are placed on the surface of $\mathcal{V}_1$. δ indicates the maximal distances of a point on $\mathcal{M}$ to $\mathcal{V}_0$.

Benchmarks for the multi core version were produced on a machine with 16 Intel(R) Xeon(R) CPU X5550 with 2.67GHz and 8192 kB cache each and 48GB main memory, under Ubuntu Linux for amd64 using the GNU C++ compiler v4.4 with optimizations (`-O2`). For the GPU version we used a machine with an Intel Core i7 at 3.20GHz and 12GB RAM, under Ubuntu Linux using the GNU C++ compiler v4.4 with optimizations (`-O3`) and a NVIDIA GeForce GTX 480 with 1536MB RAM with Cuda version 3.2 and Thrust v1.3.0. Table 4.2 shows the runtime of our scenarios for several resolutions, with corresponding plots for resolution $D = 10$ and $D = 11$ in Figure 4.12 and for $D = 12$ in 4.13.

The first three scenarios, namely *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 phase. For the GPU version, we could not get timings for the scenarios at resolution 12 (other than the bunny) since the machine ran out of memory (octree compression 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 diminishes the speed up by 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 not too small, equally sized and the motion is significant, the voxelization algorithm produces a massive amount of voxels per triangle, thus keeping all threads on the GPU busy. Further the overestimation for the allocated GPU memory is lower for larger triangles. For example when choosing $D = 11$, the whole set of triangles from the sweep can be voxelized in only two voxelization kernel invocation.

For the *dragon* and the *engine* one can see a nice speedup when using multi-core, however the peak is reached at 8 cores. When switching to 16 cores the overhead for organizing as well as the greater amount of voxels handed to the main thread (local filters are smaller) diminish the speed up by doubling the cores. The GPU version greatly improves the pure voxelization algorithm and thus can compete with or even beat the 16 core implementation. However, as the dragon consists of very small triangles a lot of voxels are generated at least twice (due to the conservative voxelization each voxel occupied by an edge is computed at least twice). We can only overestimate the actual number of voxels produced in one kernel invocation, which in this scenario, is significantly higher than the actual number of voxels read back after unification. This causes a lot of memory traffic between CPU and GPU slowing the GPU version down.

In the scenario of the *steering gear*, the voxel filtering (Section 4.2.2) and vibration detection (4.2.2) have a great impact on the running time. The overhead of handling additional local octree copies pays off, since the local copies approximate the octree of the master thread quite well. Thus, a large number of voxels can be ruled out **in parallel** and do not need to be handed over to the main thread. Unfortunately, the voxel filtering and vibration detection features cannot be incorporated into the Cuda implementation. However, the GPU beats the comparable single-core version by a factor of 2.

As shown in Table 4.1, the output complexity (triangle count) is remarkably low for all scenarios. As expected, for the under-tessellated bunny model, the output triangle count is significantly higher than the generator triangle count. On the contrary, for the highly over-tessellated dragon model, the output complexity is about half the input complexity, even at the highest resolution.

For real world scenarios, one might want the triangle count of the output mesh to be in the same order as the number of triangles in the input generator. Approximating the swept volume of a lowly tessellated generator with a highly

tessellated mesh seems inadequate. This is the case for both our real world scenarios. Especially for the vibration scenario, the input generator triangle count and the triangle count of the swept volume approximation for $D = 12$ are about the same, which indicates that $D = 12$ is a reasonable choice for this scenario.

4.5 Conclusion for the Octree Based Approach

In this chapter we presented an implementation of the concept of Chapter 2 which stores the volumetric representation in an octree. In order to reduce memory consumption as well as running time several heuristic adjustments were applied to the octree data structure. Further a culling method is added prior to the actual voxelization.

We are aware of that the box-triangle intersection test of our voxelization might cause our voxelization to be non-conservative due to rounding errors. But, we feel that missing one or another voxel is not a severe problem. However, if we falsely cull a whole triangle, (of arbitrary size) we could rip significant holes in the voxelization. Thus we chose an exact implementation (`CGAL::orientation_3`, [11]) of the culling tests, since these are only based on point-plane-test. For this reason, we did not move the culling part to the GPU. Results for the different culling methods are shown in Table 4.3. Listed are the number of triangles that could be culled based on local criteria and the number of candidate triangles, i.e., the number of all triangles generated during the sweep, either by interpolating edges or by the generator itself. The culling rate is the ratio of these values. Further, the last column shows the computational time of the methods. It is the time spent merely on culling without voxelization on a single core.

The culling rates are in between 84-94% which is similar to those reported in [12]. One can see that rates for exact culling and non-exact culling are almost identical, where the exact version culls slightly more triangles for *bunny*, *dragon* and *steering* but less triangles for the engine scenario. The exact culling comes with a computational overhead.

Note that the approach ignores narrow tunnels to (possibly large) caves inside the voxelization. In case the tunnel has a diameter of around δ, the hull computation explores the cave, but it may appear as a closed void since the tunnel is closed

model	type	culled	candidate	rate	time (s)
bunny	non-exact	3872286	4155300	93%	1.13
	exact	3872574	4155300	93%	1.69
dragon	non-exact	419484337	446488822	93%	122
	exact	419743207	446488822	94%	156
engine	non-exact	200786335	235978007	85%	216
	exact	198871494	235978007	84%	421
steering	non-exact	25793875185	28734748224	89%	9576
	exact	25795597787	28734748224	89%	12886

Table 4.3: Comparison between exact and non-exact culling methods.

due to the subsequent offset computations. Note that this does not contradict our guarantees since we are only interested in the one-sided Hausdorff-distance. If the tunnel is larger than ε (as depicted in Figure 4.14) the cave is explored.

4.6 Acknowledgement to the Co-Authors

The work presented in this chapter is based on work published and presented at the 27th European Workshop on Computational Geometry (EuroCG) in 2011 [46] and at the IEEE International Conference on Robotics and Automation (ICRA) in 2012 [47]. The implementation of the octree, the multi-core parallelization and the exact version of the culling were done by the second author of the papers mentioned above.

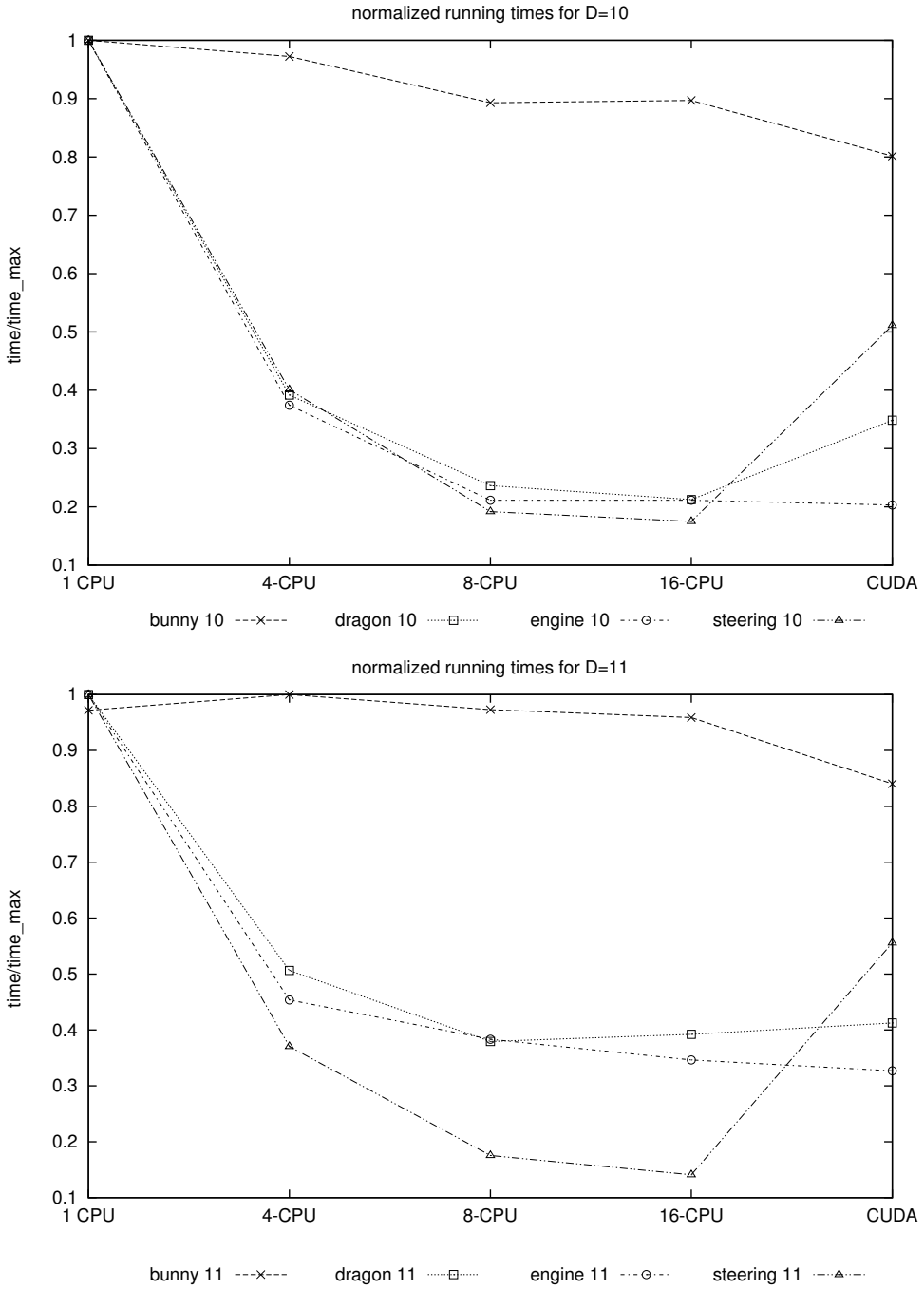

Figure 4.12: Performance of the Octree based approach on different machines for resolutions $D = 10, 11$.

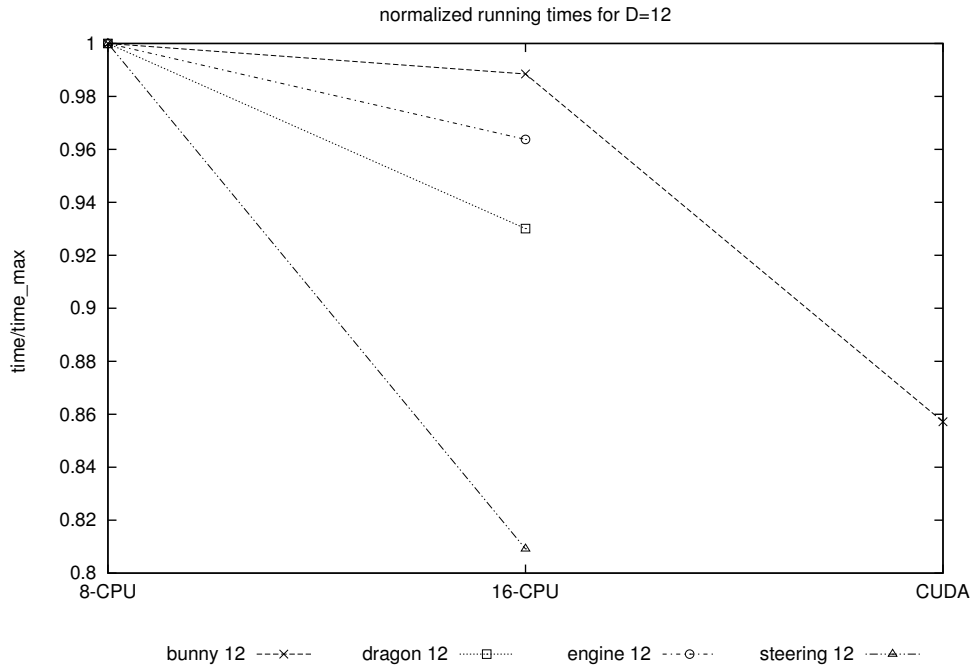

Figure 4.13: Performance of the Octree based approach on different machines for resolution $D = 12$.

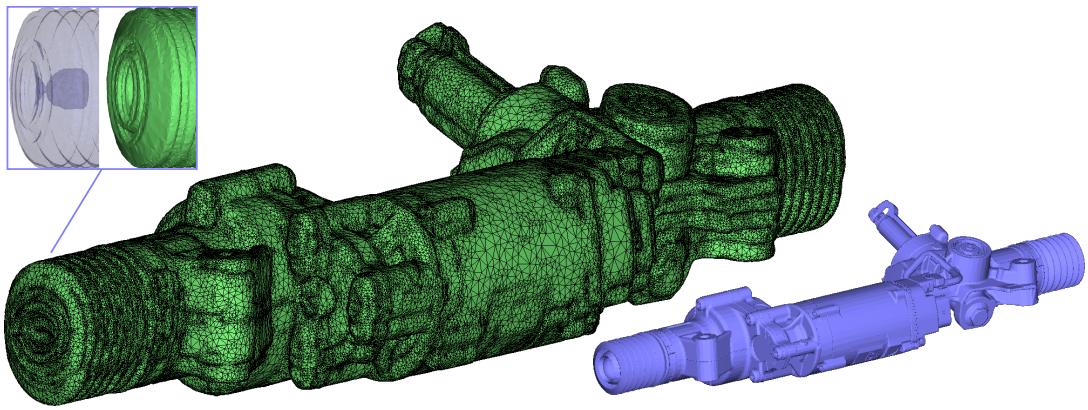

Figure 4.14: Steering gear scenario. Lower right: Input generator with 26k triangles swept along a trajectory consisting of 27k transformation. Middle: Swept volume boundary approximation with 28k triangles. Upper left: The algorithm explores a concavity formed by a hollow part of the generator (view from top).

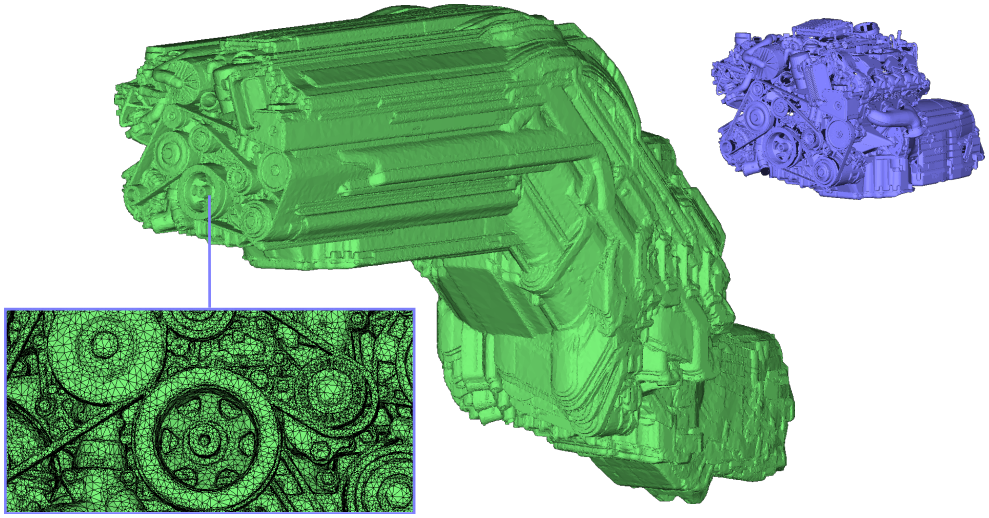

Figure 4.15: Swept volume of the engine scenario. Top right: the generator: engine model. Bottom left: The magnifications show the very fine details caused by the generator in its end position.

Chapter 5

Verification and Coloring of Swept Volumes

In this chapter we describe a method to color a swept volume that was computed with any method that implements the concept described in Chapter 2.

The method can be used as well to verify an already computed swept volume boundary approximation.

We start with the description of the coloring and extend it to verification further in this chapter.

The method works per vertex on the output mesh $\mathcal{M}$ in a post process. We relate every vertex to the triangle that comes closest during the sweep. The color of that triangle is applied to that vertex. This triangle we call **witness triangle**.

Instead of storing just the color, it is also possible to store other data, such as the index of the witness triangle and the transformation under which the minimal distance is realized.

5.1 Distance Computation with Inverse Trajectories

For the computation of the colors, we look at the sweeping scenario from a different angle. Instead of moving the generator, we now interpret the generator

to be fixed in space and apply the transformations of the inverse trajectory

$$\mathcal{R}_i^{-1},\ i = m-1,\ldots,0$$

to all points in $[0,1]^3$. Therefore, applying the inverse trajectory to a point $p \in [0,1]^3$ yields a set of m points. We define the **inverse point-trajectory** of the point p to be the polygonal chain interpolating this set of points: $\mathcal{T}^{-1}(p)$.

To find the witness triangle we have to find the pair of line segment of $\mathcal{T}^{-1}(p)$ and triangle of the generator $\mathcal{G}$ that has the smallest distance.

5.1.1 Bounding Volume Hierarchies

To speed up this computation we used bounding volume hierarchies. For the static generator we use a bounding sphere hierarchy as proposed in [36], but with a top down splitting heuristic as proposed in [17], that, starting from the initial set of triangles, recursively uses the eigenvector of the covariance matrix with the largest eigenvalue as the splitting direction. The spheres bounding the triangles are computed with the smallest enclosing spheres algorithms proposed by Welzl [49]. The bounding sphere hierarchy is computed once and reused for every point-trajectory test.

For the point-trajectories, we implemented capsule trees, i.e., for every vertex v of the mesh a hierarchy of capsules bounding the polygonal chain $\mathcal{T}^{-1}(v)$ is computed.

For a polygonal chain with vertices $a_0,\ldots,a_i$ we compute a tight fitting capsule with the following heuristic: We choose the line passing through a_0 and a_i to be the axis of the capsule,

$$a_0 + \lambda \frac{a_i - a_0}{\|a_i - a_0\|_2}.$$

The radius r of the capsule is the maximum distance from $a_1,\ldots,a_{i-1}$ to that line. Having the radius, we find the centers of the hemispheres by intersecting balls with radius $r+\varepsilon$ centered at $a_0,\ldots,a_i$ with the line through the axis of the capsule, with a small $\varepsilon > 0$. This yields two intersection points for each vertex a_j with according parameter values $\lambda_j^{(1)}, \lambda_j^{(2)}$, w.l.o.g we assume $\lambda_j^{(1)} < \lambda_j^{(2)}$. The values

$$\lambda_a = \max_{j=0,\ldots,i} \lambda_j^{(1)}, \text{ and } \lambda_b = \min_{j=0,\ldots,i} \lambda_j^{(2)}$$

then determine the centers of the hemispheres of the capsule. This is depicted in Figure 5.1.

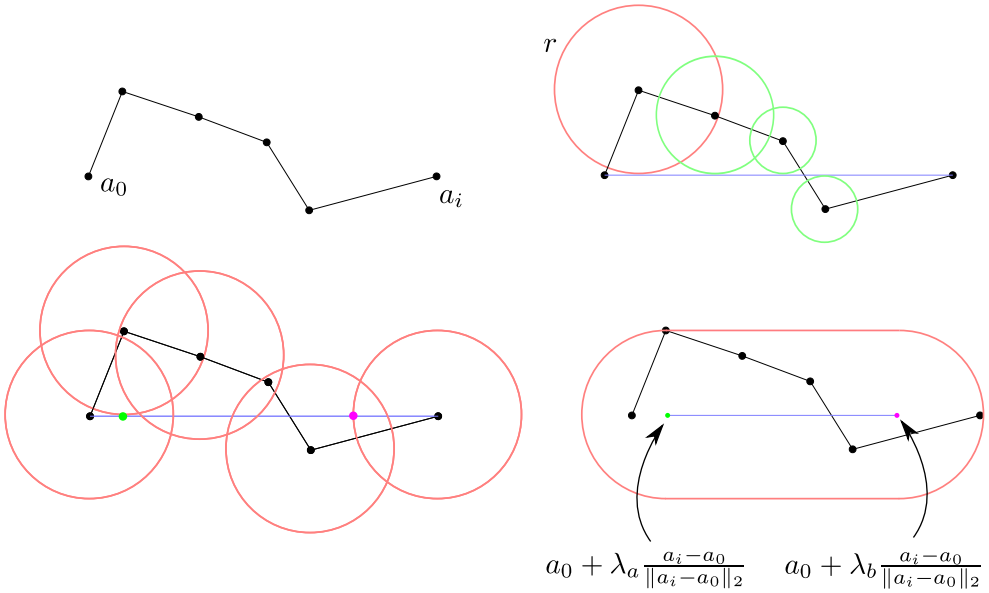

Figure 5.1: Computation of a tight fitting capsule. Top left: the polygonal chain. Top right: The distance from each vertex to the axis is computed and the maximum radius r (red circle) is stored. Bottom left: circles with radius $r + \varepsilon$ are intersected with the axis and the centers of the hemispheres are calculated. Bottom right: the final capsule bounding the polygonal chain.

The computation of a more tightly fitting capsule might be achieved, by taking into account eigenvectors of the covariance matrix, for example. However, one has to keep in mind that such a capsule hierarchy has to be computed for every inverse point trajectory of every vertex of the mesh $\mathcal{M}$.

Building one capsule tree in the above fashion costs $O(m \log m)$, and computing the distance between a polygonal chain and a triangle mesh with the aid of bounding volume hierarchies has an empirical running time of $O(\log N \cdot \log m)$. Hence coloring a mesh with V vertices (empirically) costs

$$O(V(m \log m + \log N \log m)). \tag{5.1}$$

The cost of the initial construction of the sphere tree from the generator is negligible.

Although it might seem as an obvious improvement, clustering trajectories of adjacent vertices is not advisable, since proximity on the swept volume boundary does not imply proximity of the generating parts of the model. Although two points on the swept volume boundary are close, they can be generated by two totally different parts of the generator.

5.2 Experimental Results

For the benchmarks we used a machine with an Intel Core i7 at 3.20GHz and 12GB RAM, under Ubuntu Linux using the GNU C++ compiler v4.4 with optimizations (`-O3`).

We only used meshes from scenarios with a significant motion, i.e., *bunny*, *dragon* and *engine*. We do not expect the coloring to work too well for vibrations, since the capsule tree will not group spatially coherent points it this case, which is crucial for running time.

generator		trajectory	swept volume		time
name	N	m	V	$\#tri$	
bunny	8100	129	11.9k	71.1k	128s
			78.4k	470k	840s
dragon	871k	129	23k	137k	608s
			73k	440k	1951s
engine	328k	191	30k	177k	1249s
			125k	755k	5233s

Table 5.1: Results for coloring swept volumes.

The results are shown in Table 5.1. For each scenario, running time grows linear with the number of vertices, as expected according to Equation 5.1. The average time consumption for one inverse trajectory test is: 10 ms for the bunny scenario, 26 ms for the *dragon* and 40 ms for the *engine*.

The bunny and *dragon* scenarios have identical trajectories and only differ in generator triangle count N. Although the dragon model has roughly 100 times more triangles than the bunny model, the average time for a test grows only by a factor of 2.6.

When comparing the running times of *dragon* and *engine* we can observe the super-linear dependency on m, since trajectory complexity m grows by 48% and average time consumption by 54%. Moreover, the engine model has far less triangles!

5.3 Verification of Swept Volumes

With the method presented in this chapter, we are able to relate parts of the final swept volume boundary approximation to the generating triangles, for example to apply colors a posteriori.

Although this works quite well in practice, we should keep in mind that this approach is not totally consistent with the concept presented in Chapter 2, since the point-wise linear approximation of the inverse trajectory is not consistent with $\mathcal{SV}$. Strictly speaking, they are consistent up to the point where we linearly interpolate the edges of the generator at consecutive time steps, but the subsequent tessellation of these patches introduces some error.

The method presented in this chapter can also be used to verify already computed swept volume approximations on a per vertex basis.

For example, we can easily modify the distance test between the inverse trajectory and the mesh to check if it comes closer to the mesh than $\delta + 2\varepsilon$, but does not intersect the mesh. This means we can test if a vertex is inside the hull!

An obvious question is: Why did we not use the modified inverse trajectory test as a predicate for the restricted Delaunay meshing in the first place? The answer is simple: It would be far too costly! The predicate is called during the binary search as well as in the sampling stage of the triangle verification. Hence, it can only be applied in an a posteriori process.

As already mentioned, the meshes from the depth buffer based approach presented in Chapter 3 might lack some concavities. With our verification method

Figure 5.2: Colored swept volume as well as colored generator of the *engine* scenario.

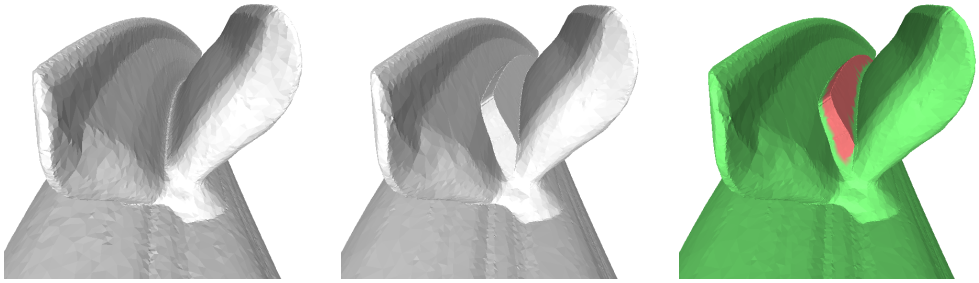

Figure 5.3: Part of the swept volume of the bunny scenario. Left: computed with octree based approach, Middle: computed with depth buffer based approach, Right: same as middle, with colors showing verification of hull containment.

we are now able to detect these concavities a posteriori, simply by testing for every vertex, if the inverse trajectory of that vertex comes closer to the generator than $\frac{5\sqrt{3}}{n}$. If this test returns negative, we can be sure to have hit a missing concavity. In the *bunny* scenario, there is such a missing concavity in a valley formed by the ears of the bunny model during the end of the sweep, as depicted in Figure 5.3. The valley formed by the ears of the bunny is only fully present in the swept volume computed with octree based method. Those vertices that did not pass the test, i.e., that are not in the hull defined in the above fashion, are colored in red, others in green.

Chapter 6

Conclusion and Future Work

6.1 Conclusion

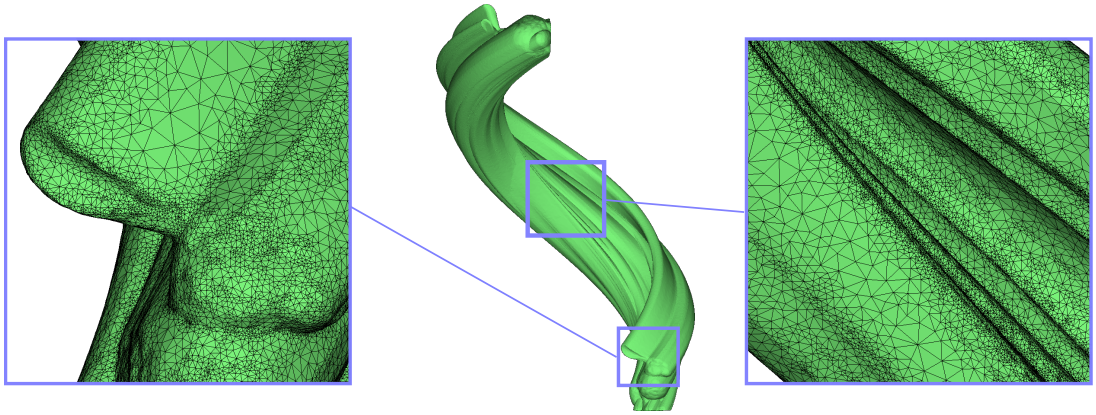

Figure 6.1: The Stanford bunny swept along a trajectory consisting of 192 transformations. The output mesh consists of 400k triangles. The magnifications show the local adaptivity and the high quality of the mesh.

We have presented a concept for the conservative and error bounded approximation of the outer boundary of a swept volume. The approximating mesh is of high quality and low triangle count. Due to the conservativeness and the error bounds in terms of the one-sided Hausdorff distance, the swept volume approximation is perfectly suited for clearance checks, continuous collision detection and maintainability analysis in CAD.

We presented two implementations that are able to cope with very high model and trajectory complexity and, concerning error tolerance, high demands. At the same time, both implementations do not impose any restrictions on the input models. They are scaleable, in the sense, that one can start with a coarse approximation of the swept volume (small D) and increase the precision in a subsequent computation, if needed. Further the implementations are very robust, even in the case of self intersecting sweeps, since there is no need for ill-conditioned trimming of intersecting triangles.

The depth buffer based approach presented in Chapter 3 and the octree based approach from Chapter 4 mainly differ in the way they represent the voxelization as well as the tolerance hull. Notice that the depth buffer based approach computes a voxelization of an offset of $\mathcal{SV}$ to get a conservative voxelization of $\mathcal{SV}$ itself, whereas the octree based approach directly computes a conservative voxelization of $\mathcal{SV}$.

In the first approach, the hull is implicitly defined, and containment is tested via a sampling based sufficient criterion. The second approach applies two voxel offsets to explicitly define the hull. Containment in the hull means containment in the two voxel layers, but not in the conservative voxelization of $\mathcal{SV}$ itself.

The dependency of the Hausdorff error bounds from the resolution are quite similar,

$$h(\mathcal{M}, \mathcal{SV}) \leq 5\frac{\sqrt{3}}{n}$$

in the case of the depth buffer based approach (cf., (3.5)) and

$$h(\mathcal{M}, \mathcal{SV}) \leq 3\frac{\sqrt{3}}{n}$$

for the octree based approach (cf., (4.1)).

On the hardware we used for benchmarking, the first approach was able to handle resolutions up to 2^{14}, whereas the second approach could only cope with resolutions of 2^{12}. The advantage of the first approach, of course, comes with the drawback that, the error bound does not hold globally, but only in those concavities that are present in at least one of the depth buffers.

Such high resolutions ensure that the approximation is of high precision, i.e., submillimeter tolerance demands for the presented real world scenarios can easily

be met. Despite of the high precision, the output complexity is remarkably low since the mesh is not uniform but locally adapts to the curvature of $\mathcal{SV}$. Both implementations have in common the very high quality of the output mesh (cf., Figure 6.1) because they utilize the CGAL Delaunay refinement implementation, where additional requirements can easily be posed to the resulting triangles.

With the a posteriori coloring method presented in Chapter 5, we are able to relate parts of the final swept volume boundary approximation to the generating triangles. We used this method to color meshes from the two approaches, as can be seen in Figure 5.2.

6.2 Future Work

Often the user of CAD software already knows interesting regions such as narrow passages and is only interested in parts of the swept volume that lie in the vicinity of those areas. In that case, it would be of great benefit to compute only a local swept volume approximation. One could use a smaller voxel grid for the same error bound, which of course means less memory consumption and less running time. The concept presented in this work is not directly applicable. Although the generator geometry is connected, the local swept volume might not be. Further, one can not easily decide if a boundary of the local swept volume is part of the outer boundary of the swept volume.

Our concept can be directly applied to Minkowski sums, edge-patch generation and local culling for Minkowski sums can be found in [30, 12].

Another question that directly follows, is the widening of the concept presented in this work to multi parameter sweeps or kinematic chains. An application of this, in automotive industry would be computing the volume that is swept by a car seat when taking into regard every possible configuration of distance to driving wheel, angle of the backrest and adjustment of the headrest.

Bibliography

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

[2] Karim Abdel-Malek and Harn-Jou Yeh. Geometric representation of the swept volume using jacobian rank-deficiency conditions. *Computer-Aided Design*, 29(6):457–468, 1997.

[3] Karim Abdel-Malek and Harn-Jou Yeh. On the determination of starting points for parametric surface intersections. *Computer-Aided Design*, 29(1):21–35, 1997.

[4] Steven Abrams and Peter K. Allen. Computing swept volumes. *Journal of Visualization and Computer Animation*, 11:69–82, 2000.

[5] Jaewoo Ahn and Sung Je Hong. Approximating 3D general sweep boundary using depth-buffer. In *ICCSA (3)*, pages 508–517, 2003.

[6] Tomas Akenine-Möller. Fast 3d triangle-box overlap testing. In *ACM SIGGRAPH 2005 Courses*, SIGGRAPH '05, New York, NY, USA, 2005. ACM.

[7] Denis Blackmore and M. C. Leu. Analysis of swept volume via lie groups and differential equations. *Int. J. Rob. Res.*, 11(6):516–537, 1992.

[8] Jean-Daniel Boissonnat and Steve Oudot. Provably good sampling and meshing of surfaces. *Graphical Models*, 67:405–451, 2005.

[9] Jean-Daniel Boissonnat and Steve Oudot. Provably good sampling and meshing of lipschitz surfaces. In *Proceedings of the twenty-second annual symposium on Computational geometry*, SCG '06, pages 337–346, 2006.

[10] Mario Botsch, David Bommes, Christoph Vogel, and Leif Kobbelt. GPU-based tolerance volumes for mesh processing. In *Pacific Conference on Computer Graphics and Applications*, pages 237–243, 2004.

[11] Herv'e Brönnimann, Andreas Fabri, Geert-Jan Giezeman, Susan Hert, Michael Hoffmann, Lutz Kettner, Sylvain Pion, and Stefan Schirra. 2D and 3D geometry kernel. In *CGAL User and Reference Manual*. CGAL Editorial Board, 3.8 edition, 2011.

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

[13] Homer H. Chen and Thomas S. Huang. A survey of construction and manipulation of octrees. *Computer Vision, Graphics, and Image Processing*, 43(3):409 – 431, 1988.

[14] Jonathan D. Cohen. Concepts and algorithms for polygonal simplification. *SIGGRAPH 99 Course Tutorial, Interactive Walkthroughs of Large Geometric Datasets*, 20:C1–C34, 1999.

[15] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer, 3rd edition, 2008.

[16] Mark Forsyth. Shelling and offsetting bodies. In *SMA '95: Proceedings of the third ACM symposium on Solid modeling and applications*, pages 373–381, New York, NY, USA, 1995. ACM.

[17] S. Gottschalk, M. C. Lin, and D. Manocha. OBB-tree: A hierarchical structure for rapid interference detection. *Computer Graphics*, pages 171–180, August 1996. Proc. SIGGRAPH'96.

[18] Wang Guoping, Sun Jiaguang, and Hua Xuanji. The sweep-envelope differential equation algorithm for general deformed swept volumes. *Computer Aided Geometric Design*, 17(5):399–418, 2000.

[19] Jon Hasselgren, Tomas Akenine-Möller, and Lennart Ohlsson. *GPU Gems 2*, chapter Conservative Rasterization, pages 677–690. Addison-Wesley Professional, 2005.

[20] Jesse C. Himmelstein, Etienne Ferre, and Jean-Paul Laumond. Swept volume approximation of polygon soups. In *ICRA*, pages 4854–4860, 2007.

[21] Tim Van Hook. Real-time shaded NC milling display. *SIGGRAPH Comput. Graph.*, 20(4):15–20, 1986.

[22] Jian Huang, Yan Li, Roger Crawfis, Shao Chiung Lu, and Shuh Yuan Liou. A complete distance field representation. In *VIS '01: Proceedings of the conference on Visualization '01*, pages 247–254, Washington, DC, USA, 2001. IEEE Computer Society.

[23] Jian Huang, Roni Yagel, Vassily Filippov, and Yair Kurzion. An accurate method for voxelizing polygon meshes. In *VVS '98: Proceedings of the 1998 IEEE symposium on Volume visualization*, pages 119–126, New York, NY, USA, 1998. ACM.

[24] Yunching Huang and James H. Oliver. NC milling error assessment and tool path correction. In *SIGGRAPH '94: Proceedings of the 21st annual conference on Computer graphics and interactive techniques*, pages 287–294, New York, NY, USA, 1994. ACM.

[25] K. C. Hui. Solid sweeping in image space application in NC simulation. *The Visual Computer*, 10(6):306–316, 1994.

[26] Evaggelia-Aggeliki Karabassi, Georgios Papaioannou, and Theoharis Theoharis. A fast depth-buffer-based voxelization algorithm. *J. Graph. Tools*, 4(4):5–10, 1999.

[27] Young J. Kim, Gokul Varadhan, Ming C. Lin, and Dinesh 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.

[28] Leif P. Kobbelt, Mario Botsch, Ulrich Schwanecke, and Hans-Peter Seidel. Feature sensitive surface extraction from volume data. In *SIGGRAPH '01: Proceedings of the 28th annual conference on Computer graphics and interactive techniques*, pages 57–66, New York, NY, USA, 2001. ACM.

[29] Sang Hun Lee. Offsetting operations on non-manifold topological models. *Comput. Aided Des.*, 41(11):830–846, 2009.

[30] Wei Li and Sara 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.

[31] J. A. Nelder and R. Mead. A Simplex Method for Function Minimization. *The Computer Journal*, 7(4):308–313, January 1965.

[32] Steve Oudot. Sampling and meshing surfaces with guarantees. In *These de doctorat en sciences, Ecole Polytechnique, Palaiseau, France*, 2005.

[33] G. Passalis, I. A. Kakadiaris, and T. Theoharis. Efficient hardware voxelization. *Computer Graphics International Conference*, 0:374–377, 2004.

[34] Darko Pavic and Leif Kobbelt. High-resolution volumetric computation of offset surfaces with feature preservation. *Comput. Graph. Forum*, 27(2):165–174, 2008.

[35] Martin Peternell, Helmut Pottmann, Tibor Steiner, and Hongkai Zhao. Swept volumes. *Computer-Aided Design Appl.*, 2:599–608, 2005.

[36] Sean Quinlan. Efficient distance computation between non-convex objects. In *In Proceedings of International Conference on Robotics and Automation*, pages 3324–3329, 1994.

[37] Laurent Rineau and Mariette Yvinec. *3D Surface Mesher*, 3.2 edition, 2006. CGAL User and Reference Manual.

[38] J R Rossignac and A A G Requicha. Offsetting operations in solid modelling. *Comput. Aided Geom. Des.*, 3(2):129–148, 1986.

[39] Randi J. Rost. *OpenGL Shading Language*, chapter 4.2: The Fragment Processor, page 104 ff. Addison-Wesley Longman, 2 edition, 2006.

[40] William J. Schroeder, William E. Lorensen, and Steve Linthicum. Implicit modeling of swept surfaces and volumes. In *VIS '94: Proceedings of the conference on Visualization '94*, pages 40–45, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

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

[42] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 2.1)*, chapter 3: Rasterization, page 122 ff. 2006.

[43] Mark Segal and Kurt Akeley. *The OpenGL Graphics System: A Specification (Version 3.2)*. 2009.

[44] 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.

[45] A. von Dziegielewski, R. Erbes, and E. Schömer. Real time offset surfaces. In *EuroCG Workshop on Computational Geometry*, 2010.

[46] A. von Dziegielewski, M. Hemmer, and E. Schömer. High quality surface mesh generation for swept volumes. In *EuroCG Workshop on Computational Geometry*, 2011.

[47] A. von Dziegielewski, M. Hemmer, and E. Schömer. High quality surface mesh generation for swept volumes. In *Proc. IEEE International Conference on Robotics and Automation (ICRA'12)*, St. Paul, May 2012.

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

[49] Emo Welzl. Smallest enclosing disks (balls and ellipsoids). In *Results and New Trends in Computer Science*, pages 359–370. Springer-Verlag, 1991.

[50] Andrew S. Winter and Min Chen. Image-swept volumes. *Comput. Graph. Forum*, 21(3), 2002.

[51] Zhiqi Xu, Zhiyang Chen, Xiuzi Ye, and Sanyuan Zhang. Approximate the swept volume of revolutions along curved trajectories. In *SPM '07: Proceedings of the 2007 ACM symposium on Solid and physical modeling*, pages 309–314, New York, NY, USA, 2007. ACM.

[52] Steve Zelinka and Michael Garland. Permission grids: Practical, error-bounded simplification. *ACM Transactions on Graphics*, 21:2002, 2002.

[53] Xinyu Zhang, Young J. Kim, and Dinesh 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.