Dynamic Distance Analysis

New Geometric Data Structures and Algorithms for the Real-Time Calculation of Tolerance Violating Regions in the Digital MockUp Process

Dissertation
zur Erlangung des Grades
"Doktor der Naturwissenschaften"

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

vorgelegt von
Anja Mantel
geboren in Mutlangen

Mainz, den 31.03.2015

Berichterstatter:

Tag der mündlichen Prüfung: 22.05.2015

D77

GEFÖRDERT VOM

Bundesministerium
für Bildung
und Forschung

Abstract

In many branches of industry, for example in the automobile industry, digital mockups are used to inspect the construction and the functionality of a product by using virtual prototypes. One application case is the inspection of safety distances between components. This is called the distance analysis. Engineers detect whether the safety distance between a component and its environment is kept in its resting position and during motion. In case the safety distance is violated, the geometry of the components have to be reengineered or the components have to be placed to another position. Knowing the regions of the components that violate the safety distance is important for this task.

In this work we present a real-time solution to calculate the regions between two geometric objects, which violate the safety distance. The objects are given as a set of primitives (e.g. triangles). At any time we calculate the set of all primitives that violate the safety distance for a given transformation. This set is called the set of all tolerance violating primitives. We present in this work a holistic solution which is divided into the following three big topics.

In the first part of this work we consider algorithms to test two triangles on tolerance violation. We present different approaches of triangle-triangle tolerance tests. Thereby we show that triangle-triangle tolerance tests achieve a significantly better performance than previously used distance calculations. The focus of our work is a novel tolerance test that works in dual space. Our novel approach proves to be the most efficient approach in all our benchmarks to calculate all tolerance violating triangles.

The second part of this work is focused on data structures and algorithms to calculate all tolerance violating primitives between two geometric objects in real-time. We develop a new data structure that combines a flat hierarchical part and some uniform grids. In order to ensure high performance, it is important to consider the required safety distance carefully in both, the design of the data structure and the design of the query algorithms. Therefore we present solutions that quickly obtain the pairs of primitives that have to be tested. Moreover, we develop strategies to detect primitives as tolerance violating without even calculating the costly primitive-primitive tolerance test. In our benchmarks we use complex geometric objects consisting of many hundreds of thousands primitives. We show with these benchmarks, that our solution is able to calculate all tolerance violating primitives in real-time.

In the third part we present a novel memory optimized data structure that maintains the cell contents of the uniform grids used before. We call this data structure shrubs. Previous approaches to reduce the memory consumption of uniform grids are related to hashing methods. But hashing methods do not reduce the memory consumption of the cell contents. In our application case the cell content of neighboring cells is often similar. Based on this, our approach reduces the memory consumption of the cell contents of a uniform grid. It is able to compress the memory lossless until one fifth of its original size and to decompress it at runtime.

Finally we show how our solution to calculate all tolerance violating primitives finds application in practice. Beside the pure distance analysis, we show that our solution is applied to different problems in the field of motion planning.

Zusammenfassung

In vielen Industriezweigen, zum Beispiel in der Automobilindustrie, werden Digitale Versuchsmodelle (Digital MockUps) eingesetzt, um die Konstruktion und die Funktion eines Produkts am virtuellen Prototypen zu überprüfen. Ein Anwendungsfall ist dabei die Überprüfung von Sicherheitsabständen einzelner Bauteile, die sogenannte Abstandsanalyse. Ingenieure ermitteln dabei für bestimmte Bauteile, ob diese in ihrer Ruhelage sowie während einer Bewegung einen vorgegeben Sicherheitsabstand zu den umgebenden Bauteilen einhalten. Unterschreiten Bauteile den Sicherheitsabstand, so muss deren Form oder Lage verändert werden. Dazu ist es wichtig, die Bereiche der Bauteile, welche den Sicherhabstand verletzen, genau zu kennen.

In dieser Arbeit präsentieren wir eine Lösung zur Echtzeitberechnung aller den Sicherheitsabstand unterschreitenden Bereiche zwischen zwei geometrischen Objekten. Die Objekte sind dabei jeweils als Menge von Primitiven (z.B. Dreiecken) gegeben. Für jeden Zeitpunkt, in dem eine Transformation auf eines der Objekte angewendet wird, berechnen wir die Menge aller den Sicherheitsabstand unterschreitenden Primitive und bezeichnen diese als die Menge aller toleranzverletzenden Primitive. Wir präsentieren in dieser Arbeit eine ganzheitliche Lösung, welche sich in die folgenden drei großen Themengebiete unterteilen lässt.

Im ersten Teil dieser Arbeit untersuchen wir Algorithmen, die für zwei Dreiecke überprüfen, ob diese toleranzverletzend sind. Hierfür präsentieren wir verschiedene Ansätze für Dreiecks-Dreiecks Toleranztests und zeigen, dass spezielle Toleranztests deutlich performanter sind als bisher verwendete Abstandsberechnungen. Im Fokus unserer Arbeit steht dabei die Entwicklung eines neuartigen Toleranztests, welcher im Dualraum arbeitet. In all unseren Benchmarks zur Berechnung aller toleranzverletzenden Primitive beweist sich unser Ansatz im dualen Raum immer als der Performanteste.

Der zweite Teil dieser Arbeit befasst sich mit Datenstrukturen und Algorithmen zur Echtzeitberechnung aller toleranzverletzenden Primitive zwischen zwei geometrischen Objekten. Wir entwickeln eine kombinierte Datenstruktur, die sich aus einer flachen hierarchischen Datenstruktur und mehreren Uniform Grids zusammensetzt. Um effiziente Laufzeiten zu gewährleisten ist es vor allem wichtig, den geforderten Sicherheitsabstand sinnvoll im Design der Datenstrukturen und der Anfragealgorithmen zu beachten. Wir präsentieren hierzu Lösungen, die die Menge der zu testenden Paare von Primitiven schnell bestimmen. Darüber hinaus entwickeln wir Strategien, wie Primitive als toleranzverletzend erkannt werden können, ohne einen aufwändigen Primitiv-Primitiv Toleranztest zu berechnen. In unseren Benchmarks zeigen wir, dass wir mit unseren Lösungen in der Lage sind, in Echtzeit alle toleranzverletzenden Primitive zwischen zwei komplexen geometrischen Objekten, bestehend aus jeweils vielen hunderttausend Primitiven, zu berechnen.

Im dritten Teil präsentieren wir eine neuartige, speicheroptimierte Datenstruktur zur Verwaltung der Zellinhalte der zuvor verwendeten Uniform Grids. Wir bezeichnen diese Datenstruktur als Shrubs. Bisherige Ansätze zur Speicheroptimierung von Uniform Grids beziehen sich vor allem auf Hashing Methoden. Diese reduzieren aber nicht den Speicherverbrauch der Zellinhalte. In unserem Anwendungsfall haben benachbarte Zellen oft ähnliche Inhalte. Unser Ansatz ist in der Lage, den Speicherbedarf der Zellinhalte eines Uniform Grids, basierend auf den redundanten Zellinhalten, verlustlos auf ein fünftel der bisherigen Größe zu komprimieren und zur Laufzeit zu dekomprimieren.

Abschießend zeigen wir, wie unsere Lösung zur Berechnung aller toleranzverletzenden Primitive Anwendung in der Praxis finden kann. Neben der reinen Abstandsanalyse zeigen wir Anwendungen für verschiedene Problemstellungen der Pfadplanung.

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die zum Gelingen dieser Arbeit beigetragen haben.

Zu aller erst möchte ich meinem Betreuer und Doktorvater für sein in mich gesetztes Vertrauen danken. Außerdem danke ich ihm für seine Anleitung, seinen fachkundigen und weitsichtigen Rat zu allen Fragestellungen und für die Diskussionsrunden in Mainz, in denen immer wieder neue Ideen entstanden sind.

Genau so sehr möchte ich meiner Betreuerin und Zweitgutachterin danken. Die vielen gemeinsamen Diskussionen und ihr Engagement für unsere Arbeitsgruppe gingen weit über das Selbstverständliche hinaus. Insbesondere möchte ich ihr aber auch für die freundschaftliche Zusammenarbeit danken.

Außerdem bedanke ich mich auch bei einer weiteren Professorin. Ohne sie und ihr Vertrauen in mich wäre meine Promotion erst gar nicht zustande gekommen.

Unseren Projektpartnern danke ich für die Vermittlung fachlicher Hintergründe und für die Darlegung von Problemstellungen aus der Praxis.

Ein ganz herzlicher Dank geht an meine Kollegen in Stuttgart und Mainz. Die Zusammenarbeit hat immer viel Spaß gemacht. Den Stuttgarter Kollegen danke ich für den Gedankenaustausch sowie für die Unterstützung vor allem in der Endphase meiner Promotion. Den Mainzer Kollegen danke ich für die fachlichen Anregungen und dafür dass sie mich so selbstverständlich aufgenommen haben.

Nicht zuletzt möchte ich mich bei meiner Familie bedanken. Insbesondere bei meinem Partner, der mir die Freiheit gegeben hat meinen eigenen Weg zu gehen, mich stets unterstützt hat und damit diese Arbeit erst ermöglichte.

Contents

Nomenclature

$A \oplus S_\delta$	The δ-offset of A, which is the Minkowski-sum of a set $A \subset \mathbb{R}^3$ with a closed ball S_δ in the origin and radius $\delta \geq 0$ (Definition 1.3 and 1.4).
$A(\mathsf{N})$	The set of ancestor nodes of a node N in the shrubs (Definition 5.1).
$A^+(\mathsf{N})$	The set of ancestor nodes of a node N in the shrubs including the node N itself (Proposition 5.2).
A^*	The polar dual of a set A (Definition 2.1).
$\mathring{A}$	The inner of the set A, defined as $A \setminus \partial A$.
$\mathring{A}^*$	The inner of the set A^*, defined as $A^* \setminus \partial A^*$.
∂A	The boundary of the set A.
$\mathcal{C}$	A cell in a grid (Section 3.3.3).
$\mathfrak{D}$	A dynamic object (which is a complex object according to Notation 1.1). We apply a transformation $\mathbf{T}$ on $\mathfrak{D}$ for the transformation.
$\mathfrak{D}_\mathcal{C}$	The primitives in $\mathfrak{D}$ that are referenced by the cell $\mathcal{C}$ (Section 3.3.3 and Definition 3.2).
$\mathcal{G}$	A uniform grid with cell width w (Notation 3.1).
$\mathcal{G}(\mathfrak{O})$	The uniform grid of a complex object $\mathfrak{O}$ with cell width w. The primitives of $\mathfrak{O}$ are referenced by the cells in $\mathcal{G}(\mathfrak{O})$ (Notation 3.1).
$\mathcal{H}$	The home cell of a cell $\mathcal{C}$ (Section 3.3.3). This is the cell in a grid, where the center point of a query cell $\mathcal{C}$ is localized.
$H_\mathbf{a}$	The hyperplane $\langle \mathbf{x}, \mathbf{a} \rangle = 1$ (Definition 2.1).
$H_\mathbf{a}^-$	A closed halfspace that contains the origin and that is restricted by the hyperplane $H_\mathbf{a}$ (Definition 2.1).

$\mathsf{N}_{\mathcal{C}}$	The directly associated node in the shrubs of the cell $\mathcal{C}$ in a TD-grid (Definition 5.1).
$\mathfrak{O}_{\mathcal{C}}$	The primitives in the complex object $\mathfrak{O}$ that are referenced by the cell $\mathcal{C}$ (Definition 3.2).
$\mathfrak{O}_{\mathsf{N}}$	The content of a node N in the shrubs, thus, the primitives in the complex object $\mathfrak{O}$ that are referenced by the node N (Definition 5.2).
$\mathfrak{O}$, $\mathfrak{S}$, $\mathfrak{D}$, ...	Complex objects, which are defined as a set of primitives, for example, as a set of triangles (Notation 1.1). We use $\mathfrak{S}$ to denote a static and $\mathfrak{D}$ to denote a dynamic object.
$\mathfrak{P}_{\mathcal{C}}$	The intersection of all primitives in $\mathfrak{D}_{\mathcal{C}}$ with the cell $\mathcal{C}$ (Section 3.3.3).
$\mathcal{S}(\mathfrak{O})$	The shrubs data structure of a complex object $\mathfrak{O}$ for a given cell width w and tolerance value δ (Notation 5.1).
$\mathfrak{S}$	A static object (which is a complex object according to Notation 1.1).
$s(r, \mathbf{m})$	A sphere with radius r and center $\mathbf{m}$.
$S(r, \mathbf{m})$	A closed ball, thus, the closed inner of a sphere, with radius r and center $\mathbf{m}$.
S_{δ}	A closed ball, thus, the closed inner of a sphere, with radius δ and center in the origin.
$T_{\delta}(\mathfrak{O}_{\mathcal{A}}, \mathfrak{O}_{\mathcal{B}})$	The set of all tolerance violating primitives between $\mathfrak{O}_{\mathcal{A}}$ and $\mathfrak{O}_{\mathcal{B}}$ respective to the tolerance value δ (Definition 1.5).
$\mathbf{T}$	A transformation, which is given as 4×4 matrix.
$Z(\mathsf{N})$	The set of associated cells in a TD-grid of a node N in the shrubs (Definition 5.1).
Θ_M	The compression ratio of the grid memory due to the of the shrubs data structure (Definition 5.8). For example a compression ratio of 0.6 means that the date is compressed to 60% of the original size.
Θ_P	The compression ratio of primitive references due to the use of the shrubs data structure (Definition 5.4). For example a compression ratio of 0.6 means that the date is compressed to 60% of the original size.

κ_S	The complexity of the scenario S (Appendix A.5).
$\oplus$	The operator that build the Minkowski-sum (Definition 1.3).
$d(A, B)$	The Euclidean distance between A and B (Definition 1.1).
r	Dependent on the context either the cell radius of a grid cell which is half the diagonal (Section 3.3) or the radius of a sphere or a ball.
w	The cell width of a grid (Notation 3.1).
δ	The tolerance value, i.e. the safety distance. It is a real value ≥ 0.
Asymmetric approach	Our broad phase algorithm that uses TI-grids to store the dynamic object and TD-grids to store the static object (Section 3.3.2).
BVH	Bounding volume hierarchy (Section 3.1.1).
Candidate cells	Grid cells that contain candidate primitives for the primitives in a query cell (Section 3.3.3).
Candidate primitives	Candidate primitives are primitives, which are possibly tolerance violating (Section 3.3.3).
Complete result	A result that provides all affected (e.g. tolerance violating or intersecting) primitives or pairs of primitives and not only one witness pair or a Boolean answer (Section 1.2).
Compressed TD-grid	A TD-grid that uses the shrubs data structure to maintain the cell content of all grid cells (Notation 5.1).
Core-primitives	A primitive in the static object $\mathfrak{S}$ that is identified as definitely tolerance violating with the primitives of a query cell (Section 3.3.4).
DMU	Digital mockup
Offline reported result	A result is reported offline, if it is provided after all transformation steps are calculated (Section 1.2).
Online reported result	A result is reported online, if the result of each single transformation step is provided as soon as this transformation step is calculated (Section 1.2).
Pair-result	The pair-result consists of all tolerance violating primitive pairs between two complex objects (Section 1.2).

Set-result	The set-result is the set of all tolerance violating primitives between two complex objects (Section 1.2).
Symmetric approach	Our broad phase algorithm that uses TI-grids to store the dynamic object and further TI-grids to store the static object (Section 3.3.2).
TD-grid	A uniform grid with cells that reference primitives with distance $\leq D(\delta)$ (Definition 3.3). $D(\delta)$ depends on δ and we set $D(\delta) = \delta + r$ (Section 3.3.3).
TI-grid	A uniform grid with cells that reference intersecting primitives (Definition 3.3).
tps	Tests per second (Section 4).

1. Introduction

The virtual validation of the components of a car within the digital mockup process (DMU) is very important to avoid construction failures early before the production of the car. The components of a virtual prototype have to be checked to hold a given safety distance to the respective surrounding constructed space. For example, in resting position, while driving maneuver, or within the assembly process. Often components do not comply with the required safety distance and the engineers want to know all conflicting regions to be able to revise the construction. For a better understanding of the situation, the engineers additionally want to be able to move the component interactively and see which parts of the geometry violate the safety distance during motion. There are software tools that calculate for two static objects a so called distance band. However, this calculation is far away from real-time calculation and not useable for interactive applications. With our work we present a solution to calculate for two components all regions that violate the safety distance in real-time.

Geometrically this means that we are given two complex objects composed of primitives, e.g., two triangular meshes, an online sequence of transformations, and a tolerance value δ, which is the given safety distance. For every single transformation step we apply the transformation to one of the objects and want to calculate in real-time all primitives that are closer than the tolerance value to the other object. We call these primitives the tolerance violating primitives. Figure 1.1 shows an example of a situation with tolerance violating primitives.

The information which primitives are tolerance violating is necessary to modify the construction of the components or the motion. Beside this application, this information

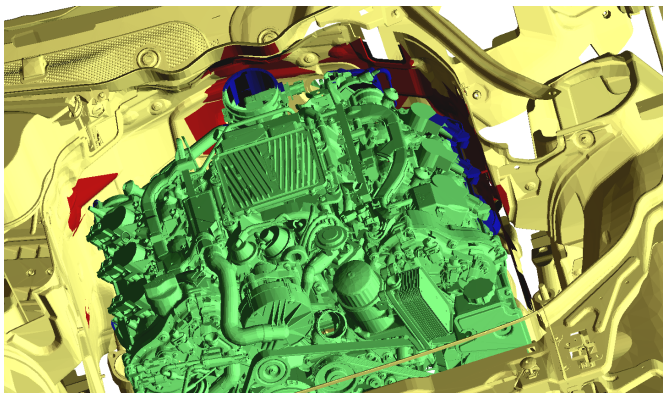


Fig. 1.1: Tolerance violating triangles between the front part of a car's bodyshell (yellow) and parts of the engine (green). The tolerance violating triangles are displayed in red and in blue.

can be further processed. For example, a retraction movement can be derived, which immediately retracts the movable object back to a legal not tolerance violating position. This is also an application within the DMU-process that helps engineers to define manually motion paths without tolerance violations in a virtual environment. Beside this application, the calculation of all tolerance violating primitives can find application within automatic motion planning tasks. One use-case is the motion planning with non-collision free start position. Common motion planners are not able to handle such a situation. The set of tolerance violating primitives can be used to derive a feasible disassembly direction into a free position. Another use-case is the motion planning under consideration of safety distances. Common motion planners plan only collision free paths, but for many industrial applications it is desired to plan a path that always guarantees a safety distance between the robot and the obstacles. The consideration of safety distances, i.e. of tolerances, decreases the free space and therefore makes the motion-planning task much more difficult. A decreasing free space means that the obstacles come closer and that, possibly, the only valid path passes so called narrow passages where obstacles are very close to another. In this case the set of all tolerance violating primitives can be used again to derive a retraction motion that retracts the robot into a free position, which can help to guide the robot through narrow passages.

1.1. Our Contribution

To the best of our knowledge, there is no efficient solution in the literature to calculate all tolerance violating primitives per motion step for interactive applications. We present in this work new data structures and algorithms to close this gap. Thereby we take special care to consider the tolerance value δ in the design of the data structures and the algorithms.

Our problem statement implicates that many pairs of primitives have to be tested on tolerance violation. Intersection and distance tests between primitives are well studied topics. Tolerance tests are not yet focused in that intensity. We will formulate and analyse primitive-primitive tolerance tests theoretically in Chapter 2 and present our developed approaches. Thereby we focus on triangles as primitives. Beside known techniques as utilizing separating planes, we present a completely new approach by translating the problem into dual space.

Since the calculation of all tolerance violating primitives per motion step is not yet considered by others, we analyze popular data structures with respect to their sufficiency. We designed a data structure and algorithms that are optimally capable for the problem statement. Our data structure combines a hierarchical part with several uniform grids in order to use the advantages of both types of data structures. A central issue within the design of our data structure and within the design of our algorithms is the consideration of the tolerance value δ. Therefor we present different approaches. One approach bases on a classical definition of a uniform grid and considers the tolerance value δ only at query time. Another approach bases on a specially developed tolerance

value depended grid that is highly optimized on fast calculations. Thereby the consideration of δ is partially sourced out to the preprocessing. To speed up the calculation especially for large δ and many tolerance violating primitives, we developed the idea of core-primitives in order to save some costly primitive-primitive tolerance tests. Our data structures and algorithms are presented in Chapter 3 and further implementation details and benchmarks are reported in Chapter 4. Our approach is not restricted to a special type of primitive. Nevertheless the reader can always imagine triangles as primitives. Our examples and benchmarks are also focused on triangles.

We present in this work a new approach to compress the memory consumption of a uniform grid that stores partially identical data in adjacent cells. Usually the memory consumption of grids is reduced in the context of reducing the number of stored cells by, for example, hashing methods. Our approach considers moreover the content of each cell and compresses the total memory by compressing the redundant content of adjacent cells. We call this new data structure *shrubs* and present it in Chapter 5.

Finally, we have implemented an environment for interactive real-time visualizations. There the tolerance violating primitives between two complex objects are visualized or are used to calculate a retraction movement to keep the objects always free from tolerance violation. As introduced in the beginning, this application can be used by engineers to inspect their construction. Beside this, we made experimental studies to use tolerance violating triangles for the above introduced motion planning problems. This four application cases and our solutions are outlined in Chapter 6.

1.2. Proximity Queries

There are different types of proximity queries between two complex objects. Some important are:

Types of Proximity Queries

- Collision tests
- Tolerance tests
- Distance computation
- Penetration depth computation

Collision detection and distance computation are probably the most common proximity queries. A collision test checks whether two objects are intersecting or not. Usually, within distance computation the smallest Euclidean distance between two objects is determined. In the case of two intersecting objects, their smallest distance is always zero. For such cases the penetration depth becomes interesting. The penetration depth is the smallest translational distance that would separate the two objects.

The focus of this work is on tolerance tests. Tolerance tests are a generalization of collision tests. A tolerance test checks whether the smallest distance between two objects is greater or smaller than a given constant tolerance value. In the special case the tolerance value is zero, we talk about a collision test.

It is obvious that the calculation of the distance between two objects is more complicated than a collision test. The effort for a tolerance test is ranked between the effort for a collision test and the effort for a distance calculation. A naive approach for a tolerance test is to calculate the distance between the objects and finally compare it with the tolerance value. In this case the effort is equal to the one for a distance calculation. During the distance calculation process many intermediate results are calculated which are always an upper bound on the final distance value. Steadily comparing the tolerance value with these intermediate results improves the naive approach. The calculation is finished as soon as the intermediate distance result is smaller than the tolerance value or as soon as it can be guaranteed that the distance between the objects is surely greater than the tolerance value. Thus, the effort for a tolerance test is smaller or equal than the effort for a distance calculation. It is obvious that an intersection test is less effort than a tolerance test with the following illustration: Consider two polytopes for a tolerance test. We can express the tolerance test as in intersection test by replacing one of the polytopes by its offset surface with the offset value equal to the tolerance value. This offset surface is not any more composed out of planar faces. It consists of plane faces, sphere- and cylinder-patches. To compute whether a polytope intersects or is inside of the offset surface of another polytope is surely more effort than computing the intersection between the two original polytopes.

Results of Tolerance Tests

The standard result of a tolerance test (including the special case of a collision test) between two complex objects consisting of several primitives is a Boolean answer: "True" for tolerance violation and "false" for no tolerance violation. But also other, additional and more detailed results, are possible. The demanded result of a query can be for example:

- Boolean (tolerance violating "true" or "false")
- One witness pair of tolerance violating primitives
- The set of all tolerance violating primitives (abbreviated *set-result*)
- All pairs of tolerance violating primitives (abbreviated *pair-result*)

The effort for getting one witness pair of primitives is equal to a Boolean answer. But the effort for calculating a set-result or a pair-result is significantly higher. Naturally, for the first two types of results the calculation is finished as soon as the first witness pair of primitives is detected. For the other two types of results the calculation goes on until all primitives or all pairs of primitives are found. Therefore, we denote these two result types *complete results*. The effort for the calculation of a complete result strongly depends on the number of reported primitives.

Calculating a set-result is less effort than the calculation of a pair-result. Although in both cases all primitive pairs have to be processed, in the case of a set-result the primitive test for some primitive pairs can be saved. In the case that a primitive pair has to be tested and both primitives had been marked as tolerance violating earlier in the calculation, the primitive test for this pair is not necessary anymore. More on this issue is given in Section 3.3.5. This work is focused on the calculation of set-results.

Nevertheless, the presented data structures and algorithms can be used with only little modifications for the calculation of pair-results.

Motion

In case proximity queries are applied on two moving objects, the motion of one object can always be described in relation to the local coordinate system of the other object. Hence, one of the objects can be considered as a static object and the other object as a dynamic object. One can distinguish between methods that handle continuous motions and methods that handle motions given as a sequence of many discrete motion steps. In this work we will consider only the latter as it is most usual in practical applications. The desired result for a sequence of discrete motion steps depends on the use-case and can be:

- One result for all motion steps *altogether*
- One result for each single motion step reported *offline*
- One result for each single motion step reported *online*

In the case of set-results, the first one differs from the other two as it provides all primitives between two objects that are tolerance violating in at least one of the given transformation steps. The other two give a much more detailed information as they provide the tolerance violating primitives in each single step. Therefore the calculation effort is usually higher for them. The difference between the second and the third one is the point in time when the results are computed and reported. Offline reported means that the result of all steps is provided after all motion steps are calculated. This is satisfactory in the case a given motion path has to be verified. Online reported means that for each single motion step the result is calculated and reported immediately as the motion step is done. This is required, for example, in interactive simulations. We will present in this work an online approach. The calculation effort for an offline and an online calculation is principally equal. But especially in case of parallel algorithms, offline algorithms have a big advantage in the distribution of the parallel tasks. In an online approach the work for the calculation of only one single motion step is distributed among the available threads. In an offline approach the work to calculate all motion steps is distributed among all threads, which has much more potential for an evenly work load balancing.

1.3. Related Work

In the literature lots of methods for efficient collision tests and for distance calculations are described. However, methods for efficient tolerance tests are not much considered in research, yet.

Tolerance Tests

There is one tolerance test for complex objects provided by the proximity query package PQP [27]. It is implemented by using bounding volume hierarchies (hereinafter referred to as BVH) of oriented bounding boxes as well as of rectangular swept spheres, which are presented by Gottschalk et al. [13] and by Larsen et al. [28, 29]. It is an algorithm with a Boolean result, thus the calculation is finished as soon as the first witness pair

of tolerance violating triangles is detected. This is different to our task as we want a complete output of all tolerance violating primitives.

Complete Result Outputs

Algorithms for complete results have to inspect a wider region compared to algorithms for Boolean results. Therefore they are not comparable in the calculation effort, thus, they are not comparable in the challenges for the algorithms design. In literature, complete results are often examined for the calculation of all intersecting primitives, i.e. in the special case $\delta = 0$. For example there are many publications concerning this issue in the field of parallel collision detection for the simulation of deformable models and N-body simulation. We will reflect some of these works in the following. Although we have in common the complete result calculation, they have further challenges like updating the data structure, which is not necessary in our case. However, we have the further challenge of performing tolerance tests.

Tang et al. [50, 49] and Lauterbach et al. [30] present parallel collision detection algorithms for deformable models. For the deformation they have to calculate all intersecting triangles or, in case of a continuous treatment, all intersecting triangles along a linear interpolation between two time steps. Therefor they present parallel BVH traversal algorithms. In [50] a workload balancing is presented that bases on the maintenance of a front of node-pairs in the bounding volume test tree. The term bounding volume test tree (BVTT) was originally introduced by Larsen et al. [29] and represents the sequence of node-pairs that have to be tested during the whole BVH traversal. Starting from the node-pairs of this front, the remaining tree is traversed in order to find all intersections. The algorithm in [50] is parallelized over the node-pairs of the front in the BVTT by using OpenMP. In [30] all calculations are performed on the GPU. Therefor a light-weight work balancing is proposed, where the threads maintain private work queues with node-pairs that have to be tested and shared work queues for the work balancing between the threads. Also the presented approach in [49] calculates all steps of the algorithm on the GPU. In contrast to previous work they present streaming technique to utilize the GPU. Kim et al. [23] developed also an approach for continuous collision detection for deformable models, where they developed a CPU/GPU hybrid approach. They parallelize a BVH traversal and update on the CPU and calculate the elementary tests on the GPU.

The work presented by Figueiredo and Fernando [6] is in the same context as our work, thus also motivated by industrial applications and virtual prototyping. They calculate the set of all intersecting surfaces between two rigid objects and use therefor the surface information of CAD-data. Optionally they also calculate the set of all intersecting triangles, similar as in our problem statement. However, they also do not consider tolerance violations. The algorithm is based on the calculation of the overlap of axis aligned bounding boxes in order to consider only surfaces that intersect the overlapping region. Their algorithm is parallelized by using OpenMP.

Some more related work that considers complete result outputs and uses uniform grids as underlying data structures are given in the following.

Only Erbes [3] provides a solution for the calculation of all tolerance violating primitives. Although his work is mainly focused on efficient collision and tolerance tests with

Complete Result Outputs for Tolerance Queries

a Boolean answer, the software he provides has the opportunity to compute extended outputs like all tolerance violating triangles. He uses therefor a BVH of orientated bounding boxes each of which is described in relation to its parent bounding volume. However, his approach is an offline calculation, which is different to our problem as we want a complete output of all tolerance violating primitives for interactive applications online. To the best of our knowledge, there is no solution in the literature for our application case.

Combined Data Structure

As mentioned at the beginning of the introduction, we use a combined data structure for our calculations. A combined data structure is also used for example by Pan and Manocha [43], but in another context as ours, namely for local sensitive hashing for k-nearest neighbor search. They present a random projection tree composed with several hash tables per leaf, which are constructed on the base of Morton curves.

Grid Query

Recent work that uses uniform grids is mainly focused on memory efficiency (hashing methods) and parallelization and is mainly proposed in the context of multiple particle simulation or the simulation of deformable models. A selection, which also provides complete result outputs, is given in the following. For example Le Grand [14] introduces a parallel implementation of the collision detection between multiple particles by using the GPU. It is based on spacial hashing and sorting in order to detect grid cells that reference multiple particles. Then the primitives that are referenced by the same cell are tested on mutual collisions. Pabst et al. [42] propose a CPU/GPU hybrid collision detection for deformable and for rigid models as well as for multiple particle simulation. Similar as in [14] their method is based on spatial hashing and sorting but their algorithms consider triangulated surfaces. A collision detection approach for deformable volumetric models, which are presented by tetrahedrons sets, is presented by Teschner et al. [51]. Their approach bases also on spatial hashing. Gissler et al. [11] continue this work and propose parallel algorithms for efficient construction, update and query, which are performed on the CPU.

The mentioned work all use one uniform grid where the particles or the deformable surface parts are stored and updated in order to test them for collisions. Having two complex rigid objects is something different, as it is not recommendable to update all primitives of the dynamic object in every motion step within one grid. An approach that considers two rigid objects is the so called *voxmap-pointshell* approach, which was introduced by McNeely et al. [36] and extended by Renz et al. [45]. It is originally an approximating approach and was utilized for haptic interaction in virtual environments. Both papers consider a static and a dynamic object. The surface of the dynamic object is sampled by a point cloud with surface normals. Usually this point cloud is created by voxelizing the surface of the object and projecting the voxel centers onto the surface. The static object is represented by using a uniform grid, called the voxmap. A grid cell of the voxmap is called occupied, if the cell references to some primitives of the static object or is inside the static object. At query time all points, which represent the dynamic object, are located in the voxmap. A collision is detected when one point of the pointshell is found that is located in an occupied cell of the voxmap. The voxmap-pointshell algorithm is extended by Reithmann [44] and Erbes [3] who do not approximate their objects. Their objects are given as triangle sets. They reference in

each voxel the intersecting triangles of their objects and use these triangles for collision tests. Like Reithmann and Erbes we require a none-approximating calculation and therefore consider the primitives of our complex objects. But in contrast to their work, we have to obtain all tolerance violating primitives and not only one pair of colliding triangles. Summarized, the ideas we use from the voxmap-pointshell algorithm are only that each object is stored in a separate grid and that for the grid query the center point of a grid cell of the dynamic object is located in the grid of the static object.

Primitive-Primitive Tolerance Tests

There is a lot of literature on collision and distance tests between primitives – but less on tolerance tests. The proximity query package PQP [27] uses within their tolerance test between complex objects an efficient triangle-triangle distance test. Erbes [3] and Erbes, Mantel, Schömer, and Wolpert et al. [4] propose algorithms for fast triangle-triangle tolerance tests.

1.4. Definitions and Problem Statement

Within this section we introduce the most important definitions and give a formal notation of our problem statement.

Notation 1.1 (Primitives and Complex Object): *We denote a geometric primitive like a triangle, a segment, a tetrahedron etc. by $\mathcal{P} \subset \mathbb{R}^3$ and consider a complex object $\mathfrak{O} \subset \mathbb{R}^3$ as a set of primitives:*

$$\mathfrak{O} := \{\mathcal{P}_1, \mathcal{P}_2, ...\mathcal{P}_n\} .$$

The most common representation of a complex object is a surface representation by a set of triangles. In the case of a volumetric object, it is common to represent the object by a set of tetrahedrons.

For the tolerance test between two complex objects, the Euclidean distance is fundamental.

Definition 1.1 (Euclidean Distance): *Given a point $\boldsymbol{a} \in \mathbb{R}^3$ and a point $\boldsymbol{b} \in \mathbb{R}^3$ then*

$$d(\boldsymbol{a}, \boldsymbol{b}) := \|\boldsymbol{a} - \boldsymbol{b}\| = \sqrt{(a_x - b_x)^2 + (a_y - b_y)^2 + (a_z - b_z)^2}$$

is the Euclidean distance, or briefly the distance, between the points $\boldsymbol{a}$ and $\boldsymbol{b}$. The distance between two primitives $\mathcal{P}_\mathcal{A}$ and $\mathcal{P}_\mathcal{B}$ is defined as

$$d(\mathcal{P}_\mathcal{A}, \mathcal{P}_\mathcal{B}) := \min\{d(\boldsymbol{a}, \boldsymbol{b}) \mid \boldsymbol{a} \in \mathcal{P}_\mathcal{A}, \boldsymbol{b} \in \mathcal{P}_\mathcal{B}\}$$

and the distance between two complex objects $\mathfrak{O}_\mathcal{A}$ and $\mathfrak{O}_\mathcal{B}$ is defined as

$$d(\mathfrak{O}_\mathcal{A}, \mathfrak{O}_\mathcal{B}) := \min\{d(\mathcal{P}_\mathcal{A}, \mathcal{P}_\mathcal{B}) | \mathcal{P}_\mathcal{A} \in \mathfrak{O}_\mathcal{A}, \mathcal{P}_\mathcal{B} \in \mathfrak{O}_\mathcal{B}\} .$$

Tolerance tests check whether a given safety distance between two complex objects is hold. We denote this given safety distance as the tolerance value δ and claim it to be ≥ 0. Given the tolerance value, we can formulate the central query of this work, which is the tolerance test between two objects A and B. A and B can be complex objects, primitives or points.

Definition 1.2 (Tolerance Violation): *Given two objects as sets A, B with Euclidean distance $d(A, B)$ and a tolerance value $\delta \geq 0$, then we define that*

$$A \text{ and } B \text{ are tolerance violating} :\Leftrightarrow d(A, B) \leq \delta \ .$$

The Euclidean distance is essential for the above definition. Instead of formulating the necessity and sufficiency condition for a tolerance test by the consideration of the distance between A and B, it is also possible to formulate it in terms of a intersection test. Suppose that we inflate A by δ such that all points $\mathbf{p} \in \mathbb{R}^3$ with $d(A, \mathbf{p}) \leq \delta$ build the inflated shape of A. This inflated shape of A complies with the Minkowski-sum of A with a closed ball S_δ that is centered in the origin and has the radius δ.

Definition 1.3 (Minkowski-Sum): *Given two arbitrary sets $A, B \subset \mathbb{R}^n$ then the Minkowski-sum is defined as*

$$A \oplus B := \{a + b \,|\, a \in A, b \in B\} \ .$$

Using the Minkowski-sum, we can define the δ-offset of A.

Definition 1.4 (δ-Offset): *Let A be a set in $\mathbb{R}^3$ and S_δ a closed ball centered in the origin with radius $\delta \geq 0$.*

We will denote the Minkowski sum $A \oplus S_\delta$ as the δ-offset of A.

Given an object composed of polygons, the surface of the δ-offset of this object is composed of sphere patches, cylinders patches, and planer faces.

B intersects the δ-offset of A, iff there is a point $\mathbf{q} \in B$ with $d(A, \mathbf{q}) \leq \delta$. This is equivalent to $d(A, B) \leq \delta$. Thus, we have

$$d(A, B) \leq \delta \ \Leftrightarrow \ (A \oplus S_\delta) \cap B \neq \emptyset \ . \tag{1.1}$$

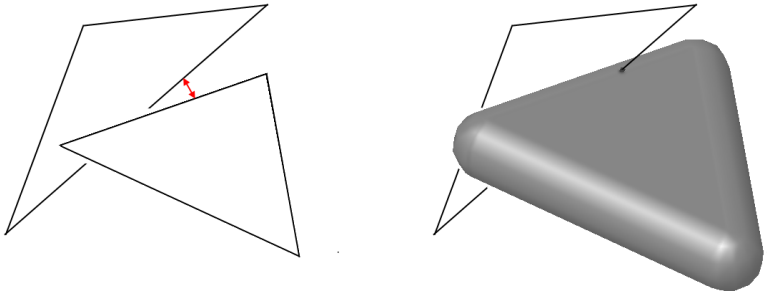


Fig. 1.2: Tolerance violating triangles. The minimal Euclidean distance between the two triangles is smaller or equal to δ (left) iff the δ-offset of one triangle intersects the other triangle (right) iff the triangles are tolerance violating triangles.

So we can give an alternative formulation for the tolerance violation as follows:

$$A \text{ and } B \text{ are tolerance violating } \Leftrightarrow \ (A \oplus S_\delta) \cap B \neq \emptyset \ . \tag{1.2}$$

Figure 1.2 illustrates the definition of a tolerance violation and the alternative formulation for the example of two triangles.

The issue of this work is the calculation of all tolerance violating primitives between two complex objects.

Definition 1.5 (Set of All Tolerance Violating Primitives): *We define the set of all tolerance violating primitives between a complex object $\mathfrak{O}_\mathcal{A}$ and a complex object $\mathfrak{O}_\mathcal{B}$ respective to a tolerance value δ as*

$$\begin{aligned} T_\delta(\mathfrak{O}_\mathcal{A}, \mathfrak{O}_\mathcal{B}) := &\{\mathcal{P}_\mathcal{A} \in \mathfrak{O}_\mathcal{A} \mid \exists \mathcal{P}_\mathcal{B} \in \mathfrak{O}_\mathcal{B} : \ d(\mathcal{P}_\mathcal{A}, \mathcal{P}_\mathcal{B}) \leq \delta\} \cup \\ &\{\mathcal{P}_\mathcal{B} \in \mathfrak{O}_\mathcal{B} \mid \exists \mathcal{P}_\mathcal{A} \in \mathfrak{O}_\mathcal{A} : \ d(\mathcal{P}_\mathcal{A}, \mathcal{P}_\mathcal{B}) \leq \delta\} \ . \end{aligned}$$

Based on the above definitions, the problem statement of this work can be given more formally:

Problem Statement:

We will consider two complex objects, a static object $\mathfrak{S}$ and a dynamic object $\mathfrak{D}$. In every time step a transformation $\mathbf{T}$ is applied to object $\mathfrak{D}$ resulting in the transformed object $\mathbf{T}\mathfrak{D}$. Our goal is to calculate the set of all tolerance violating primitives $T_\delta(\mathbf{T}\mathfrak{D}, \mathfrak{S})$ online for every time step.

2. Tolerance Tests for Primitives

Tolerance tests between complex objects base on tolerance tests between geometric primitives like triangles, boxes, spheres, tetrahedrons and others. According to Definition 1.2 the primitives $\mathcal{A}$ and $\mathcal{B}$ are tolerance violating in relation to the tolerance value δ iff

$$d(\mathcal{A}, \mathcal{B}) \leq \delta \tag{2.1}$$

or equivalent, according to Equation 1.2, iff

$$(\mathcal{A} \oplus S_\delta) \cap \mathcal{B} \neq \emptyset \, . \tag{2.2}$$

When two complex objects $\mathfrak{O}_\mathcal{A}$ and $\mathfrak{O}_\mathcal{B}$ are given and we are interested in all tolerance violating primitives $T_\delta(\mathfrak{O}_\mathcal{A}, \mathfrak{O}_\mathcal{B})$, many primitive-primitive tolerance tests have to be calculated. The amount of time spent on primitive tests is notable high for such a task - in contrast to, for example, the task where $\mathfrak{O}_\mathcal{A}$ and $\mathfrak{O}_\mathcal{B}$ are tested on collision.

Effort of Tolerance Tests

One reason is that much more pairs of primitives have to be tested than for the collision task. This is because the number of tolerance violating primitives of two objects $\mathfrak{O}_\mathcal{A}$ and $\mathfrak{O}_\mathcal{B}$ is naturally larger or equal than the number of intersecting primitives, i.e. $|T_0(\mathfrak{O}_\mathcal{A}, \mathfrak{O}_\mathcal{B})| \leq |T_\delta(\mathfrak{O}_\mathcal{A}, \mathfrak{O}_\mathcal{B})|$. So, the number of primitives that are possibly tolerance violating and therefore are tested on tolerance violation is always larger or equal to the number of primitives that are possibly intersecting. Another reason is that all primitives have to be tested on tolerance violation, thus we cannot stop the calculation after the first found tolerance violating witness pair. A third reason is that a single tolerance

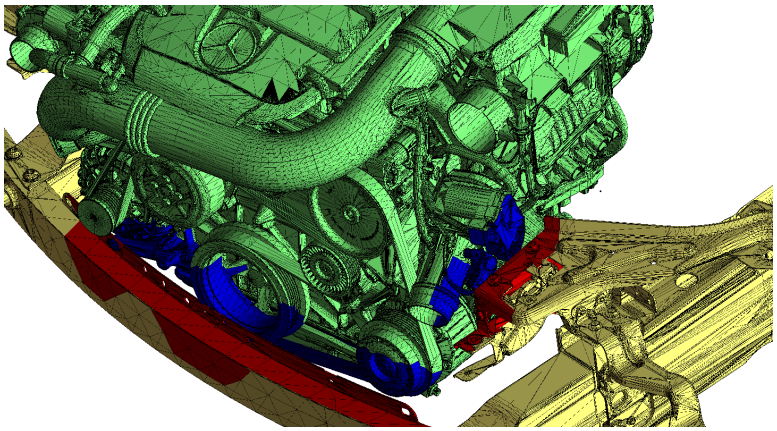


Fig. 2.1: All tolerance violating triangles between two complex objects (marked in red and blue color).

test for one pair of primitives is more effort than a single intersection test between the pair of primitives.

Figure 2.1 shows an example of the calculated result of all tolerance violating triangles between two complex objects. All tolerance violating triangles are colored in red and blue.

A self-evident approach to determine whether two primitives are tolerance violating is to calculate the shortest Euclidean distance between the primitives and then compare it with the tolerance value δ. In order to reduce this calculation effort we take advantage of the fact that we are not interested in the actual distance value and use so called *early outs*: As soon as we can be sure that the primitives are further apart than δ or as soon as we are sure that the primitives are not closer than δ, we are finished and can quit the tolerance test for this pair of primitives.

Early Out

The focus of this chapter is on triangle-triangle tolerance tests. This are the fundamental tests that have to be performed for tolerance tests of complex geometrical objects that are represented by a set of triangles.

Therefore we introduce in Section 2.1 elementary definitions and important properties and show some approaches of triangle-triangle tolerance tests. In Section 2.2 we present our new approach for a triangle-triangle tolerance test seen from another perspective as usual – we will consider triangle-triangle tolerance tests in the dual space. This dual approach proved to be the best approach for our application case to calculate all tolerance violating triangles. The benchmarks for all triangle-triangle tolerance tests are given in Section 2.3. Finally, in Section 2.4 a brief overview on tolerance tests for arbitrary primitives is given.

2.1. Triangle-Triangle Tolerance Tests

After we have introduced some basic definitions and some related work in 2.1.1 and 2.1.2, we will introduce the distance calculation between a pair of triangles in detail in 2.1.3 in order to give a deeper understanding on the numerous sub-issues that have to be considered for one tolerance tests. Based on this we will derive a straight forward tolerance test. In 2.1.4 we explain a popular technique for intersection tests – the separating axis theorem – and explain how this idea can be used for triangle-triangle tolerance tests. At last the ideas of the previous two approaches are combined for a third type of tolerance test in 2.1.5.

2.1.1. Elementary Examinations and Definitions

Triangles can be defined in various ways. One usual way is to define a triangle by its three corner points. Let $\mathbf{a}, \mathbf{b}, \mathbf{c} \in \mathbb{R}^3$ with $(\mathbf{b}-\mathbf{a}) \times (\mathbf{c}-\mathbf{a}) \neq \mathbf{0}$, then we can define a triangle as

$$\mathcal{T} := \{\mathbf{x} \in \mathbb{R}^3 \mid \mathbf{x} = \mathbf{a} + \lambda(\mathbf{b}-\mathbf{a}) + \mu(\mathbf{c}-\mathbf{a}), \quad \lambda, \mu \in [0,1] \subset \mathbb{R}, \quad \lambda + \mu \leq 1\} . \quad (2.3)$$

We will denote with $\mathbf{n} = (\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a})$ the normal of $\mathcal{T}$. By $(\mathbf{b} - \mathbf{a}) \times (\mathbf{c} - \mathbf{a}) \neq \mathbf{0}$ we exclude degenerated triangles.

Triangle Features

A triangle consists of seven features. These are the three vertices, the three edges and the triangular face. We define the features of a triangle $\mathcal{T}$ as

$$F(\mathcal{T}) := \{v_{\mathbf{a}}, v_{\mathbf{b}}, v_{\mathbf{c}}, e_{\mathbf{ab}}, e_{\mathbf{bc}}, e_{\mathbf{ca}}, f\}$$

where $v_{\mathbf{a}}, v_{\mathbf{b}}, v_{\mathbf{c}}$ are the vertices, $e_{\mathbf{ab}}, e_{\mathbf{bc}}, e_{\mathbf{ca}}$ the edges and f the face of the triangle with

$$\begin{aligned}
v_{\mathbf{a}} &= \{\mathbf{a}\}, \quad v_{\mathbf{b}} = \{\mathbf{b}\}, \quad v_{\mathbf{c}} = \{\mathbf{c}\} \\
e_{\mathbf{ab}} &= \{\mathbf{x} \in \mathbb{R}^3 \mid \mathbf{x} = \mathbf{a} + \lambda(\mathbf{b} - \mathbf{a}),\ \lambda \in (0,1) \subset \mathbb{R}\} \\
e_{\mathbf{bc}} &= \{\mathbf{x} \in \mathbb{R}^3 \mid \mathbf{x} = \mathbf{b} + \lambda(\mathbf{c} - \mathbf{b}),\ \lambda \in (0,1) \subset \mathbb{R}\} \\
e_{\mathbf{ca}} &= \{\mathbf{x} \in \mathbb{R}^3 \mid \mathbf{x} = \mathbf{c} + \lambda(\mathbf{a} - \mathbf{c}),\ \lambda \in (0,1) \subset \mathbb{R}\} \\
f &= \{\mathbf{x} \in \mathbb{R}^3 \mid \mathbf{x} = \mathbf{a} + \lambda(\mathbf{b} - \mathbf{a}) + \mu(\mathbf{c} - \mathbf{a}),\ \lambda, \mu \in (0,1) \subset \mathbb{R}, \lambda + \mu < 1\}\ .
\end{aligned}$$

Similar features are grouped together as follows:

$$F^V(\mathcal{T}) := \{v_{\mathbf{a}}, v_{\mathbf{b}}, v_{\mathbf{c}}\}, \quad F^E(\mathcal{T}) = \{e_{\mathbf{ab}}, e_{\mathbf{bc}}, e_{\mathbf{ca}}\}, \quad F^F(\mathcal{T}) := \{f\}$$

By the definition of the triangle and of the triangle features follows directly that the union of all sets of points in all triangle features is equal to the set of points in a triangle:

$$\bigcup_{\varphi \in F(\mathcal{T})} \varphi = \mathcal{T}\ . \tag{2.4}$$

For two triangles $\mathcal{A}$ and $\mathcal{B}$ we denote by

$$(\varphi_{\mathcal{A}}, \varphi_{\mathcal{B}}) \in F(\mathcal{A}) \times F(\mathcal{B})$$

a feature pair, where

$$d(\varphi_{\mathcal{A}}, \varphi_{\mathcal{B}}) = \min\{d(\mathbf{p}_{\mathcal{A}}, \mathbf{p}_{\mathcal{B}}) \mid \mathbf{p}_{\mathcal{A}} \in \varphi_{\mathcal{A}}, \mathbf{p}_{\mathcal{B}} \in \varphi_{\mathcal{B}}\}$$

is the Euclidean distance between the features $\varphi_{\mathcal{A}}$ and $\varphi_{\mathcal{B}}$.

Voronoi Regions of a Triangle

Given the triangle $\mathcal{T}$, we can sub-divide the $\mathbb{R}^3$ into regions such that for all points of one region the distance to one feature $\varphi \in F(\mathcal{T})$ is smaller than the distance to all other features. This regions are called Voronoi regions and are defined as

$$V(\varphi) := \{\mathbf{p} \in \mathbb{R}^3 \mid d(\mathbf{p}, \varphi) < d(\mathbf{p}, \tilde{\varphi}),\ \tilde{\varphi} \in F(\mathcal{T}) \setminus \varphi\} \tag{2.5}$$

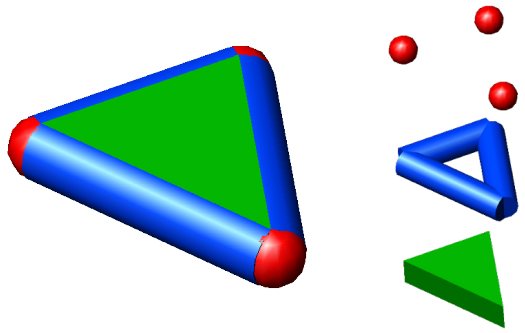


Fig. 2.2: δ-offset of a triangle, composed out of three balls, three solid cylinders, and one solid triangle prism.

The δ-offset of a Triangle

The δ-offset of a triangle $\mathcal{T}$ is given by $\mathcal{T} \oplus S_\delta$ (Definition 1.4) and is composed of

- three balls with radius δ that are centered at the vertices of the triangle $\mathcal{T}$,
- three solid right cylinders with radius δ, each with one edge of the triangle $\mathcal{A}$ as axis, and
- one solid right triangle prism with height δ and orthogonal cross section in the middle, which is the triangle face.

The δ-offset of a triangle and the respective components are displayed in Figure 2.2.

In the literature, the δ-offset of a primitive is also denoted as the swept sphere volume of a primitive. Swept sphere volumes were introduced by Larsen et al. [28] as a new type of bounding volumes. They consider in their work the swept sphere volumes of points, lines and rectangles, but not of triangles.

2.1.2. Previous Work

Fundamentals knowledge for the distance and the intersection calculation between primitives are given for example by Schneider and Eberly [48] and by Ericson [5].

Astonishingly little can be found in the literature about triangle-triangle tolerance tests. Only for the special case $\delta = 0$, the triangle-triangle intersection tests, there are many publications with efficient approaches.

Intersection Tests

A straight forward approach is to test whether at least one edge of one triangle intersects the other triangle. More sophisticated approaches are given for example by Möller [38] who first verifies whether the supporting plane of each triangle intersects the other triangle by a simple orientation test. Then he calculates the intersection interval of each triangle with the intersection line of the supporting planes. When both intervals are overlapping the triangles are intersecting. The approach of Held [17] is similar. He calculates the intersection segment of one triangle with the intersection line of

the planes and verifies whether this segment intersects an edge of the other triangle. Guigue and Devillers [1] present an approach which is based mainly on orientation tests by evaluating the signs of a small number of 4×4 determinants. Different from the mentioned approaches, Tropp et al. [52] propose an arithmetic approach by formulating a linear equation system and solving it by reusing already calculated intermediate results.

Tolerance Tests Based on Distance Tests

In general, a triangle-triangle tolerance test is easily formulated by the calculation of the distance between the two triangles and by the check whether the distance is $\leq \delta$. We will explain a straight forward distance calculation between a triangle pair in Section 2.1.3 on the basis of calculating the distances between pairs of triangle features. There are more sophisticated methods that avoid the distance calculation of some feature pairs by case decisions. Dependent in which Voronoi regions of one triangle the features of the other triangle are located, only some feature pairs may realize the smallest distance. For example Schneider and Eberly [48], Ericson [5], and especially Lin and Canny [32] use such strategies. The so called Lin-Canny algorithm is a general approach for all convex polyhedra. The GJK-algorithm, which was presented by Gilbert et al. [10], is a further popular distance calculation algorithm for convex polyhedra. These general distance tests can of course be used for the distance calculation of triangles.

The PQP Distance Test

The PQP library [27] provides an efficient implementation of a triangle-triangle distance test that is used for the tolerance test between two complex objects by algorithms of the PQP library. First, they calculate the shorted distance between every pair of edges like proposed by Lumelsky [35]. They define an edge of a triangle as the segment between two triangle vertices including the two vertices. For every pair of edges they define a slab by two planes. Each plane contains one of the calculated closest points between the edge pair and both planes are perpendicular to the direction of the connecting segment of the closest points. In the case the closest distance of the triangle is taken in such a pair of edges, the third vertex of both triangles must be outside of that slap. If this is not the case for any pair of edges, they search the closest point of the triangles between a vertex of one triangle and the other triangle's face by projections or check whether the triangles are intersecting or parallel.

Tolerance Tests

Triangle-triangle tolerance tests are presented by Erbes [3] and by Erbes, Mantel, Schömer, and Wolpert [4]. The approach presented in [3] will be explained briefly within Section 2.1.5. Three of the four approaches presented in [4] will be explained briefly in Section 2.1.3, 2.1.4, and 2.1.5. The fourth approach, which is the most interesting one, will be explained more detailed in Section 2.2.

2.1.3. Distance Calculation and the Feature Distance Approach

As mentioned in the beginning of this chapter, a triangle-triangle tolerance test can be calculated according to Equation 2.1, which bases on the calculation of the Euclidean distance between two triangles $\mathcal{A}$ and $\mathcal{B}$. The Euclidean distance $d(\mathcal{A}, \mathcal{B})$ between the two triangles can be attributed to the minimum of the Euclidean distances between all feature pairs.

Proposition 2.1: *The Euclidean distance $d(\mathcal{A}, \mathcal{B})$ between the triangles $\mathcal{A}$ and $\mathcal{B}$ can be calculated as*

$$d(\mathcal{A}, \mathcal{B}) = \min\{d(\varphi_\mathcal{A}, \varphi_\mathcal{B}) \mid (\varphi_\mathcal{A}, \varphi_\mathcal{B}) \in F(\mathcal{A}) \times F(\mathcal{B})\} \ .$$

Proof 1: *According to Definition 1.1 there exists a point $\boldsymbol{a} \in \mathcal{A}$ and a point $\boldsymbol{b} \in \mathcal{B}$ that realizes the minimal distance with $d(\mathcal{A}, \mathcal{B}) = d(\boldsymbol{a}, \boldsymbol{b})$. By Equation 2.4, the union of all points in all features $\varphi \in F(\mathcal{T})$ is equal to the set of points in $\mathcal{T}$.*
Because of that there must exist a feature $\varphi_\mathcal{A} \in F(\mathcal{A})$ with $\boldsymbol{a} \in \varphi_\mathcal{A}$ as well as there must exist a feature $\varphi_\mathcal{B} \in F(\mathcal{B})$ with $\boldsymbol{b} \in \varphi_\mathcal{B}$. Therefore the Euclidean distance of the feature pair $(\varphi_\mathcal{A}, \varphi_\mathcal{B})$ is $d(\varphi_\mathcal{A}, \varphi_\mathcal{B}) = d(\boldsymbol{a}, \boldsymbol{b})$, which is the minimal distance $d(\mathcal{A}, \mathcal{B})$. ■

From the proof follows directly:

Corollary 2.1: *There always exists a feature pair $(\varphi_\mathcal{A}, \varphi_\mathcal{B})$ that realizes the Euclidean distance between two triangles $\mathcal{A}$ and $\mathcal{B}$:*

$$\exists(\varphi_\mathcal{A}, \varphi_\mathcal{B}) \in F(\mathcal{A}, \mathcal{B}) : \quad d(\varphi_\mathcal{A}, \varphi_\mathcal{B}) = d(\mathcal{A}, \mathcal{B}) \ .$$

A straight forward approach for the distance calculation between two triangles is to calculate the distance between all feature pairs in order to finally find the minimum. As every triangle has 7 features, there are altogether 49 feature pairs. This means that 9 vertex-vertex distances, 18 edge-vertex distances, 9 edge-edge distances, 6 face-point distances, 6 face-edge distances and 1 face-face distance have to be calculated. Actually, there are some properties which allow to omit or reduce some of the 49 distance calculation.

Let $F(\mathcal{A}, \mathcal{B}) := F(\mathcal{A}) \times F(\mathcal{B})$ be the set of all feature pairs and let

$F^{VV}(\mathcal{A}, \mathcal{B}) := F^V(\mathcal{A}) \times F^V(\mathcal{B})$	the set of all vertex-vertex features
$F^{VE}(\mathcal{A}, \mathcal{B}) := F^V(\mathcal{A}) \times F^E(\mathcal{B}) \cup F^E(\mathcal{A}) \times F^V(\mathcal{B})$	the set of all vertex-edge features
$F^{VF}(\mathcal{A}, \mathcal{B}) := F^V(\mathcal{A}) \times F^F(\mathcal{B}) \cup F^F(\mathcal{A}) \times F^V(\mathcal{B})$	the set of all vertex-face features
$F^{EE}(\mathcal{A}, \mathcal{B}) := F^E(\mathcal{A}) \times F^E(\mathcal{B})$	the set of all edge-edge features
$F^{EF}(\mathcal{A}, \mathcal{B}) := F^E(\mathcal{A}) \times F^F(\mathcal{B}) \cup F^F(\mathcal{A}) \times F^E(\mathcal{B})$	the set of all edge-face features
$F^{FF}(\mathcal{A}, \mathcal{B}) := F^F(\mathcal{A}) \times F^F(\mathcal{B})$	the set of all face-face features.

For convenience we write F instead of $F(\mathcal{A}, \mathcal{B})$, if the context makes already clear that the feature pairs of the triangles $\mathcal{A}$ and $\mathcal{B}$ are meant. Analogous for F^{VV}, F^{VE}, F^{VF}, F^{EE}, F^{EF}, and F^{FF}.

Lemma 2.1: *For the calculation of the Euclidean distance between the two triangles $\mathcal{A}$ and $\mathcal{B}$ the face-face feature pair $(f_\mathcal{A}, f_\mathcal{B}) \in F^{FF}$ can be ignored, because*

$$d(f_\mathcal{A}, f_\mathcal{B}) = d(\mathcal{A}, \mathcal{B}) \quad \Rightarrow$$

$$\exists (\varphi_\mathcal{A}, \varphi_\mathcal{B}) \in F \setminus (f_\mathcal{A}, f_\mathcal{B}) : \; d(\varphi_\mathcal{A}, \varphi_\mathcal{B}) = d(f_\mathcal{A}, f_\mathcal{B})$$

Proof 2: *Let $\boldsymbol{a} \in f_\mathcal{A}$ and $\boldsymbol{b} \in f_\mathcal{B}$ the points that realizes the minimal distance. W.l.o.g. assume $\mathcal{A}$ is lying in the (x, y)-plane. Then the minimal distance arises as $|b_z| = d(\boldsymbol{a}, \boldsymbol{b})$, which is the absolute value of the z-coordinate of $\boldsymbol{b}$. Since $f_\mathcal{B}$ is an open subset of a plane we know that there are points in $\mathcal{B}$ with a smaller z-coordinate than b_z as well as points in $\mathcal{B}$ with a greater z-coordinate than b_z (Case 1) or that all points in $\mathcal{B}$ have the same z-coordinate equal to b_z (Case 2).*

- *Case 1: In this case the triangles are not parallel. Since $f_\mathcal{B}$ is an open subset of a plane, $d(\boldsymbol{a}, \boldsymbol{b}) = d(\mathcal{A}, \mathcal{B})$ with $\boldsymbol{b} \in f_\mathcal{B}$ can only be fulfilled iff $b_z = 0$. Thus, the triangles must be intersecting. We know that in this case either one edge of each triangle penetrates the other triangle or two edges of one triangle penetrate the other triangle or one vertex of one triangle contacts the other triangle. In the first two cases there exists $(\varphi_\mathcal{A}, \varphi_\mathcal{B}) \in F^{EF}$ and in the third case there exists $(\varphi_\mathcal{A}, \varphi_\mathcal{B}) \in F^{VF}$ with $d(\varphi_\mathcal{A}, \varphi_\mathcal{B}) = d(f_\mathcal{A}, f_\mathcal{B}) = 0$.*

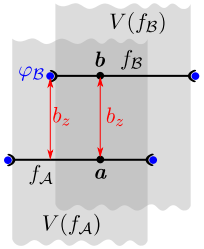


Fig. 2.3: 2d-sketch of case 2a in the proof of Lemma 2.1.

- *Case 2: In this case the triangles are parallel. According to the definition of a triangle feature's Voronoi region (Equation 2.5), it is $\boldsymbol{a} \in V(f_\mathcal{B})$ and $\boldsymbol{b} \in V(f_\mathcal{A})$. Therefore we have $f_\mathcal{A} \cap V(f_\mathcal{B}) \neq \emptyset$ and $f_\mathcal{B} \cap V(f_\mathcal{A}) \neq \emptyset$. We distinguish between the general case where the face feature of at least one triangle is not completely contained in the Voronoi region of the other triangles face feature (a) and the special case where the face feature of each triangle is completely contained in the Voronoi region of the other triangles face feature (b).*

  a) *For the general case let $f_\mathcal{A} \cap V(f_\mathcal{B}) \subsetneq f_\mathcal{A}$. Then we know that especially $\mathcal{B} \cap V(f_\mathcal{A}) \neq \emptyset$ and by this that there are other features $\varphi_\mathcal{B} \in F(\mathcal{B}) \setminus f_\mathcal{B}$ with $\varphi_\mathcal{B} \cap V(f_\mathcal{A}) \neq \emptyset$. Since for every point in $\mathcal{B}$ the distance to the (x,y)-plane is b_z, this is especially true for the points in $\varphi_\mathcal{B} \cap V(f_\mathcal{A})$. Thus, with $(f_\mathcal{A}, \varphi_\mathcal{B})$ exists a feature pair not in F^{FF} with $d(f_\mathcal{A}, \varphi_\mathcal{B}) = d(f_\mathcal{A}, f_\mathcal{B})$. Refer also to Figure 2.3.*

  b) *In the special case the triangles $\mathcal{A}$ and $\mathcal{B}$ must be congruent and $\mathcal{B}$ is equal to the translation of $\mathcal{A}$ in z-direction by b_z. Then for all points in $\mathcal{B}$ the distance to $\mathcal{A}$ is b_z. And especially there are 6 feature pairs $(\varphi_\mathcal{A}, \varphi_\mathcal{B}) \in \left(F^{VV} \cup F^{EE}\right)$ with $d(\varphi_\mathcal{A}, \varphi_\mathcal{B}) = d(f_\mathcal{A}, f_\mathcal{B}) = b_z$. Refer also to Figure 2.4.*

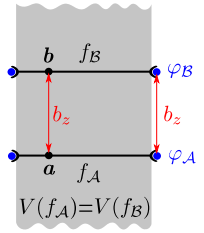


Fig. 2.4: 2d-sketch of case 2b in the proof of Lemma 2.1.

■

Lemma 2.2: *For the calculation of the Euclidean distance between the two triangles $\mathcal{A}$ and $\mathcal{B}$ the distance tests of the 6 face-edge feature pairs can be replaced by intersection tests, because*

$$\exists(\varphi_{\mathcal{A}}, \varphi_{\mathcal{B}}) \in F^{EF} : \; d(\varphi_{\mathcal{A}}, \varphi_{\mathcal{B}}) = d(\mathcal{A}, \mathcal{B}) > 0 \quad \Rightarrow$$
$$\exists(\tilde{\varphi_{\mathcal{A}}}, \tilde{\varphi_{\mathcal{B}}}) \in \left(F^{VV} \cup F^{VE} \cup F^{VF} \cup F^{EE}\right) : \; d(\tilde{\varphi_{\mathcal{A}}}, \tilde{\varphi_{\mathcal{B}}}) = d(\varphi_{\mathcal{A}}, \varphi_{\mathcal{B}})$$

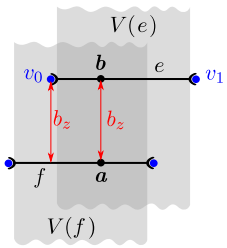


Fig. 2.5: 2d-sketch of case 2a in the proof of Lemma 2.2.

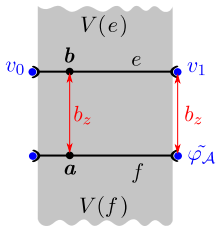


Fig. 2.6: 2d-sketch of case 2b.1 in the proof of Lemma 2.2.

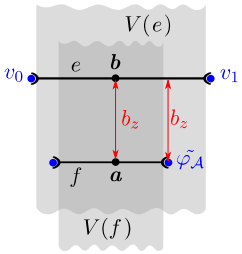


Fig. 2.7: 2d-sketch of case 2b.2 in the proof of Lemma 2.2.

Proof 3: *W.l.o.g., let $\boldsymbol{a} \in f \in F^F(\mathcal{A})$ and $\boldsymbol{b} \in e \in F^E(\mathcal{B})$ the points that realize the minimal distance with $d(\boldsymbol{a}, \boldsymbol{b}) = d(f, e) = d(\mathcal{A}, \mathcal{B}) > 0$. W.l.o.g. assume $\mathcal{A}$ is lying in the (x, y)-plane. Then the minimal distance arises as $d(\boldsymbol{a}, \boldsymbol{b}) = |b_z|$, which is the absolute value of the z-coordinate of $\boldsymbol{b}$. Since e is an open subset of a straight line l_e we know that there are points in e with a smaller z-coordinate than b_z as well as points in e with a greater z-coordinate than b_z (Case 1) or that all points in e have the same z-coordinate equal to b_z (Case 2).*

- *Case 1: In this case the e and f are not parallel. Since e is an open subset of the straight line l_e, $d(\boldsymbol{a}, \boldsymbol{b}) = d(f, e) = d(\mathcal{A}, \mathcal{B})$ with $\boldsymbol{b} \in e$ can only be fulfilled iff $b_z = 0$. But this is contradictory to the assumption $d(\mathcal{A}, \mathcal{B}) > 0$. Thus, in this case, the feature pair (f, e) never realize the minimal distance > 0.*
- *Case 2: In this case the edge e and the triangle $\mathcal{A}$ are parallel. According to the definition of a triangle feature's Voronoi region (Equation 2.5) we have $\boldsymbol{b} \in V(f)$ and therefore $e \cap V(f) \neq \emptyset$. Let $v_0, v_1 \in F^V(\mathcal{B})$ be the adjacent vertex features of the edge e. We distinguish between the following cases:*

  a) $\exists v_j \in \{v_0, v_1\}$ with $v_j \in V(f)$: Because v_j is an adjacent vertex feature of e, $v_j \in l_e$. Because l_e is parallel to the (x,y)-plane, the distance of v_j to f must be $|b_z|$. Thus, with (f, v_j) exists a feature pair $\in F^{VF}$ with $d(f, v_j) = d(f, e)$. Refer also to Figure 2.5.

  b) $v_0 \notin V(f)$ and $v_1 \notin V(f)$: We define the sets $e_{\mathcal{A}} := \{\boldsymbol{p} \in l_e \mid d(\mathcal{A}, \boldsymbol{p}) = |b_z|\}$ and $e_f := \{\boldsymbol{p} \in l_e \mid d(f, \boldsymbol{p}) = |b_z|\} = e \cap V(f)$. Because $\mathcal{A} = \mathsf{Closure}(f)$ we know that $e_f \subsetneq e_{\mathcal{A}}$ and $e_{\mathcal{A}} = \mathsf{Closure}(e_f)$. So there must exist a point $\bar{\boldsymbol{p}} \in e_{\mathcal{A}} \setminus e_f$ and a feature $\tilde{\varphi_{\mathcal{A}}} \in F(\mathcal{A}) \setminus f$ with $d(\varphi, \bar{\boldsymbol{p}}) = |b_z|$. We distinguish now between the following cases:

    1) $e \cap V(f) = e$: In this case the edge is completely inside of $V(f)$ but not the adjacent vertices v_j. So $\bar{\boldsymbol{p}}$ must be an adjacent vertex v_j. Thus, with $\tilde{\varphi_{\mathcal{A}}} \in F(\mathcal{A}) \setminus f$ there exists the feature pair $(\tilde{\varphi_{\mathcal{A}}}, v_j) \in \left(F^{VV} \cup F^{VE}\right)$ with $d(\tilde{\varphi_{\mathcal{A}}}, v_j) = d(f, e)$. Refer also to Figure 2.6.

    2) $e \cap V(f) \subset e$: In this case the edge passes through $V(f)$ and therefore there exists a $\bar{\boldsymbol{p}} \in e$. Thus, with $\tilde{\varphi_{\mathcal{A}}} \in F(\mathcal{A}) \setminus f$ there exists the feature pair $(\tilde{\varphi_{\mathcal{A}}}, e) \in \left(F^{VE} \cup F^{EE}\right)$ with $d(\tilde{\varphi_{\mathcal{A}}}, e) = d(f, e)$. Refer also to Figure 2.7.

Proposition 2.2: *Let $\mathcal{A}, \mathcal{B}$ be two triangles and $I(\mathcal{A}, \mathcal{B})$ the function*

$$I(\mathcal{A}, \mathcal{B}) := \begin{cases} 0 & \exists(\varphi_{\mathcal{A}}, \varphi_{\mathcal{B}}) \in F^{EF} : \ \varphi_{\mathcal{A}} \cap \varphi_{\mathcal{B}} \neq \emptyset \\ \infty & \textit{otherwise} . \end{cases}$$

The Euclidean distance $d(\mathcal{A}, \mathcal{B})$ between the triangles $\mathcal{A}$ and $\mathcal{B}$ can be calculated as

$$d(\mathcal{A}, \mathcal{B}) = \min \left\{ \{I(\mathcal{A}, \mathcal{B})\} \ \cup \ \{d(\varphi_{\mathcal{A}}, \varphi_{\mathcal{B}}) \mid (\varphi_{\mathcal{A}}, \varphi_{\mathcal{B}}) \in F \setminus (F^{FF} \cup F^{EF})\} \right\} .$$

Proof 4: *The proof follows directly by Lemma 2.1, 2.1 and 2.2.* ∎

Tolerance Test

A straight forward approach for a triangle-triangle tolerance test is to calculate the distance between all feature pairs one by one and compare each distance with the tolerance value δ. As soon as one distance is $\leq \delta$, computing the remaining feature distances can be omitted. Thus, we have an early out as soon as we know that the triangles are tolerance violating.

As explained above it is sufficient to calculated the 9 vertex-vertex distances, 18 edge-vertex distances, 9 edge-edge distances, 6 face-point distances and 6 face-edge intersections in order to determine the Euclidean distance. Naturally, this is also valid in the case of a tolerance test. Summarized we have to calculate at most 32 distance values and 6 intersection tests. We will call this approach the *feature distance test*.

2.1.4. The Separating Plane Approach

In contrast to the previous explained feature distance approach, the separating plane approach is motivated by an intersection test between the δ-offset of a triangle $\mathcal{A}$ and another triangle $\mathcal{B}$ (Equation 2.2). Two convex objects are disjoint iff there is a plane that separates the objects. Or, equivalently, iff there is a separating axis. A separating axis is a line on which the orthogonal projections of the two objects result in two non-overlapping intervals.

In order to test whether two objects are not intersecting, a separating axis must be found. Gottschalk et al. [13] use a separating axis test to check two oriented bounding boxes for intersection. They limit the number of axes to be investigated to only 15 based on the *separating axis theorem*. The separating axis theorem is derived by Gottschalk in [12] and also explained for example by Schneider and Eberly [48] and by Ericson [5]. It states that two convex polytopes are disjoint iff there is a separating axis perpendicular to one of the polytopes' faces or perpendicular to a pair of edges, with one edge from each polytope.

We cannot directly use this separating axis test for a triangle $\mathcal{A}$ and the δ-offset of a triangle $\mathcal{B}$, because the δ-offset of a triangle is no polytope. But we can limit the number of separating axis by inspecting only the directions which possibly connect the

closest points between the triangles $\mathcal{A}$ and $\mathcal{B}$. If we project the triangle $\mathcal{B}$ and the δ-offset of the triangle $\mathcal{A}$ on the axis that is parallel to the connection of the closest points, the intervals on the axis overlap iff the triangles are tolerance violating.

So we have the following axis to test:

- For the case the minimal distance may take place between a face feature and another feature (this includes the case that the minimal distance is adopted between a vertex-face feature pair as well as the case that the triangles are intersecting), we have to test the axis that is parallel to the normal of both triangle faces.
- For the case the minimal distance may take place between a vertex-vertex feature pair, we have to test the axis that is parallel to the connection of that vertex-vertex feature pair.
- For the case the minimal distance may take place between a vertex-edge feature pair, we have to test the axis that is parallel to the direction of the orthogonal projection of the vertex onto the edge.
- For the case the minimal distance may take place between an edge-edge feature pair, we have to test the axis that is perpendicular to that pair of edges.

In summary, there are at most 38 axes to test. A single separating axis test projects both triangles onto the axis and tests whether the resulting intervals have a distance greater than δ. As soon as a separating axis is found, the remaining axes do not have to be tested any more. Thus, we have an early out as soon as we know that the triangles are not tolerance violating. We will call this approach the *separating plane test* (the separating plane is of course perpendicular to the separating axis found).

2.1.5. The Combined Approach

The above explained *feature distance test* iterates over the feature pairs of the two triangles to calculate the distance. It has an early out as soon as one distance is found to be $\leq \delta$, thus, as soon as it is known that the triangles are tolerance violating. The above explained *separating plane test* iterates over the feature pairs of the two triangles to perform separating axis tests. It has an early out as soon as a separating axis is found, thus, as soon as it is known that the triangles are not tolerance violating.

The combined approach combines the feature based tolerance test with the separating plane test and is presented by Erbes [3]. While iterating over the feature pairs, the distance is calculated and the direction of the closest points' connection is used as potential separating axis. Thus, for every feature pair there can be an early out when it is known that the triangles are tolerance violating or when it is known that the triangles are not tolerance violating. In detail, he starts with the vertex-vertex feature pairs, followed by the vertex-edge and the edge-edge feature pairs. He completes with two separating axes test with axes parallel to the triangle normals, which covers the remaining vertex-face, edge-face, and face-face feature pairs.

We implemented an own variant of the combined approach, where we use all separating axis tests and the vertex-vertex as well as the vertex-edge feature distance tests. In contrast to Erbes we start with the separating axis test with axes parallel to the triangle normals. In detail, we process the separating axis tests in the order given in Section 2.1.4. When computing the axes that are parallel to a vertex-vertex connection and when computing the axes that are parallel to a vertex-edge connection, we check the vertex-vertex and the vertex-edge distances without notable additional effort simultaneously. We will call this approach the *combined test*.

2.2. A Dual Approach for Triangle-Triangle Tolerance Tests

Sometimes it is advantageous to transfer a problem into dual space in order to analyze it under a different perspective. In this section the triangle-triangle tolerance test is considered by studying its polar duality. We are given two triangles $\mathcal{A}$ and $\mathcal{B}$ and want to determine whether they are tolerance violating according to Equation 2.2. This is the problem statement in the so called primal space. Now we will map the δ-offset of triangle $\mathcal{A}$, which is $\mathcal{A} \oplus S_\delta$, as well as the triangle $\mathcal{B}$ to the dual space and translate the tolerance test $(\mathcal{A} \oplus S_\delta) \cap \mathcal{B}$ into terms of duality.

Therefor we introduce in Section 2.2.1 some basics about polar duality. In Section 2.2.2 we investigate the dual of a triangle and in Section 2.2.3 the dual of the δ-offset of a triangle. We formulate a general intersection test between two convex objects in dual space in Section 2.2.4 and specialize it for the intersection of a triangle and the δ-offset of a triangle in Section 2.2.5. Finally, in Section 2.2.6, we present our dual approach of the triangle-triangle tolerance test.

2.2.1. Foundations of the Dual Consideration

There are different kinds of duality in Euclidean space. We will use in the following the *polar duality*. Here we introduce the most important definitions for our problem statement. For more details, more properties of polar duality, and especially for the respective proofs refer to Wolpert [55], Geismann et al. [8] and Eggleston [2].

Definition 2.1 (Polar Duality): *Let A be a set of points in $\mathbb{R}^d$. Then the polar dual of A is defined as*

$$A^* := \{\boldsymbol{x} \in \mathbb{R}^d \mid \langle \boldsymbol{x}, \boldsymbol{a} \rangle \leq 1 \quad \forall \boldsymbol{a} \in A\} = \bigcap_{\boldsymbol{a} \in A} H_{\boldsymbol{a}}^- \tag{2.6}$$

where $H_{\boldsymbol{a}}^- := \{\boldsymbol{x} \in \mathbb{R}^d \mid \langle \boldsymbol{x}, \boldsymbol{a} \rangle \leq 1\}$ is a closed halfspace that contains the origin and that is restricted by the hyperplane $H_{\boldsymbol{a}} = \{\boldsymbol{x} \in \mathbb{R}^d \mid \langle \boldsymbol{x}, \boldsymbol{a} \rangle = 1\}$.

A^* is for any set A a closed convex set that contains the origin. An important property for our application is that the polar dual of a set A is equal to the polar dual of the convex hull of the closure of A and the origin, i.e.

$$A^* = \mathsf{ConvexHull}(\mathsf{Closure}(A) \cup \{\mathbf{0}\})^* \,. \tag{2.7}$$

The dual can be be applied multiple times where $A^{**} = \mathsf{ConvexHull}(\mathsf{Closure}(A) \cup \{\mathbf{0}\})$ and $A^{***} = A^*$.

Definition 2.2 (Polar Hyperplane and Pole): *Let $\boldsymbol{p} \in \mathbb{R}^d$ be a point not equal to the origin $\boldsymbol{0}$. Then*

$$H_{\boldsymbol{p}} := \{\boldsymbol{x} \in \mathbb{R}^d \mid \langle \boldsymbol{x}, \boldsymbol{p} \rangle = 1\}$$

is the polar hyperplane of $\boldsymbol{p}$ and the point $\boldsymbol{p}$ is the pole of $H_{\boldsymbol{p}}$.

We will say that $\mathbf{p}$ is dual to $H_{\mathbf{p}}$ and vice versa. The duality between $\mathbf{p}$ and $H_{\mathbf{p}}$ is a geometric duality that is incidence and order preserving. Please note that Definition 2.2 and Definition 2.1 differ in the used equal and the used lower-or-equal sign. For example the polar dual of the set $\{\mathbf{p}\}$ is the closed halfspace $\langle \mathbf{x}, \mathbf{p} \rangle \leq 1$, whereas the polar hyperplane of the point $\mathbf{p}$ is the hyperplane with $\langle \mathbf{x}, \mathbf{p} \rangle = 1$.

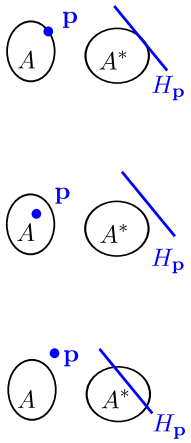


Fig. 2.8: Relationship of a set A and a point $\mathbf{p}$ (left) and relationship of the dual the set A^* and the dual hyperplane $H_{\mathbf{p}}$ (right).

One can derive the following relationship: If $A \subset \mathbb{R}^3$ is a closed convex set that contains the origin, the dual hyperplanes of all points on the boundary ∂A of A are the tangent hyperplanes of A^* and vice versa, i.e.

$$\mathbf{p} \in \partial A \;\Leftrightarrow\; H_{\mathbf{p}} \text{ is tangent to } A^* \,. \tag{2.8}$$

Moreover, the dual hyperplane $H_{\mathbf{p}}$ of a point $\mathbf{p} \in A$ does not intersect the interior $\mathring{A}^*$ of A^* and the dual hyperplane $H_{\mathbf{p}}$ of a point $\mathbf{p} \notin A$ intersects the interior $\mathring{A}^*$ of A^*, i.e.

$$\mathbf{p} \in A \;\Leftrightarrow\; \mathring{A}^* \cap H_{\mathbf{p}} = \emptyset \,. \tag{2.9}$$

This are central properties for deriving intersection tests. Refer also to Figure 2.8

Another important issue for our work is the conclusion by Wolpert [55] that the set of poles of all tangent planes on an ellipsoid describe an ellipsoid, an elliptic paraboloid or a two sheet hyperboloid in dual space. We are interested in the special case where the ellipsoid in primal space is a sphere. The set of poles of all tangent planes to a sphere $s(r, \mathbf{m})$ with center $\mathbf{m}$ and radius r is given by

$$s(r, \mathbf{m})^P = \{\mathbf{x} \in \mathbb{R}^3 \mid r^2 \langle \mathbf{x}, \mathbf{x} \rangle - (1 - \langle \mathbf{m}, \mathbf{x} \rangle)^2 = 0\} \,. \tag{2.10}$$

The set $s(r, \mathbf{m})^P$ is

- an ellipsoid in the case the origin is in the interior of $s(r, \mathbf{m})$,

- a paraboloid in the case the origin is on the boundary of $s(r, \mathbf{m})$ or
- a two sheet hyperboloid in the case the origin is not in $s(r, \mathbf{m})$.

Hung and Ierardi [19] as well as Wolpert [55] showed that the dual of the convex hull of a set of ellipsoids is the intersection cell that contains the origin in the arrangement of all ellipsoids' duals.

2.2.2. The Polar Dual of a Triangle

A triangle is defined by its three vertices as given in Equation 2.3. Let $\mathcal{B}$ be a triangle with vertices $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$. According to Equation 2.7 the dual of $\mathcal{B}$ is equal with the dual of a subset $X \subseteq \mathcal{B}$ with $\mathsf{ConvexHull}(X \cup \{\mathbf{0}\}) = \mathsf{ConvexHull}(\mathcal{B} \cup \{\mathbf{0}\})$. $X = \{\mathbf{u}, \mathbf{v}, \mathbf{w}\}$ is such a subset.

Thus, by Definition 2.1, the dual of the triangle $\mathcal{B}$ is given as

$$\mathcal{B}^* = X^* = \bigcap_{\mathbf{x} \in X} H^-_{\mathbf{x}} = H^-_{\mathbf{u}} \cap H^-_{\mathbf{v}} \cap H^-_{\mathbf{w}} \ . \tag{2.11}$$

2.2.3. The Polar Dual of the δ-offset of a Triangle

We know already that $\mathcal{A} \oplus S_\delta$, the δ-offset of a triangle $\mathcal{A}$, consists of three balls, three solid cylinders and a solid triangular prism. If we map $\mathcal{A} \oplus S_\delta$ into the dual space, the union of the balls, the solid cylinders and the solid triangular prism has to be mapped to dual space. This can be done by the aid of Equation 2.7 because $\mathcal{A} \oplus S_\delta$ is actually nothing else than the convex hull of the three spheres, i.e.

$$\mathcal{A} \oplus S_\delta = \mathsf{ConvexHull}\left(s(\delta, \mathbf{a}) \cup s(\delta, \mathbf{b}) \cup s(\delta, \mathbf{c})\right) \tag{2.12}$$

where $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$ are the vertices of the triangle $\mathcal{A}$ and $s(\delta, \mathbf{a})$, $s(\delta, \mathbf{b})$, and $s(\delta, \mathbf{c})$ are the spheres with center in $\mathbf{a}$, $\mathbf{b}$, and $\mathbf{c}$ and with radius δ.

W.l.o.g. we assume that $\mathcal{A} \oplus S_\delta$ contains the origin. This means that we have to translate a triangle pair $\mathcal{A}, \mathcal{B}$ prior to the calculation, such that $\mathcal{A} \oplus S_\delta$ contains the origin. We will translate it such that $\mathbf{a} = \mathbf{0}$.

So we get the polar dual of the δ-offset of a triangle $\mathcal{A}$ as

$$(\mathcal{A} \oplus S_\delta)^* = \mathsf{ConvexHull}\left(s(\delta, \mathbf{0}) \cup s(\delta, \mathbf{b}) \cup s(\delta, \mathbf{c})\right)^* \ . \tag{2.13}$$

We know from above that the dual of the convex hull of a set of ellipsoids is the intersection cell that contains the origin in the arrangement of all ellipsoids' duals. Since a closed ball is the convex hull of a sphere with the same radius and center, we know with Equation 2.7 that the polar dual of this sphere is equal to the polar dual of the closed ball. Thus, we get the intersection cell that contains the origin in the

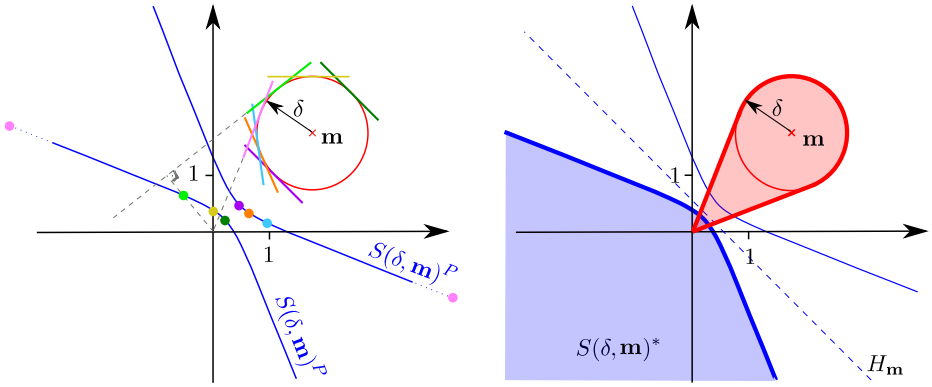


Fig. 2.9: The polar points of all tangent planes on a disk build a hyperbola in the dual space (left). The dual of a disk that does not contain the origin (right).

arrangement of $s(\delta, \mathbf{0})^*$, $s(\delta, \mathbf{b})^*$, and $s(\delta, \mathbf{c})^*$ as the intersection of polar duals of the balls $S(\delta, \mathbf{0})$, $S(\delta, \mathbf{b})$, and $S(\delta, \mathbf{c})$. So we have given the δ-offset of the triangle $\mathcal{A}$ as

$$(\mathcal{A} \oplus S_\delta)^* = S(\delta, \mathbf{0})^* \cap S(\delta, \mathbf{b})^* \cap S(\delta, \mathbf{c})^* \; . \tag{2.14}$$

It is easy to comprehend that the polar dual of the ball $S(\delta, \mathbf{0}) = \{\mathbf{x} \in \mathbb{R}^3 \mid \|\mathbf{x}\| \leq \delta\}$ is a ball with radius $1/\delta$ that is centered at the origin, thus

$$S(\delta, \mathbf{0})^* = S(1/\delta, \mathbf{0}) \; .$$

Let us now consider the general case of an arbitrary ball $S(\delta, \mathbf{m})$ in order to find $S(\delta, \mathbf{b})^*$ and $S(\delta, \mathbf{c})^*$. By Equation 2.10 the set of points that are poles of the tangent plane in every point in $\partial S(\delta, \mathbf{m}) = s(\delta, \mathbf{m})$ is $s(\delta, \mathbf{m})^P$. Further we know that the polar dual of an arbitrary closed set A is $A^* = \mathsf{ConvexHull}\,(A \cup \{\mathbf{0}\})^*$ (refer to Equation 2.7) and $A^{**} = \mathsf{ConvexHull}\,(A \cup \{\mathbf{0}\})$. Considering Equation 2.8 in dual space leads to

$$\mathbf{p} \in \partial(A^*) \;\Leftrightarrow\; H_\mathbf{p} \text{ is tangent to } A^{**}$$

and inserting $A^{**} = \mathsf{ConvexHull}\,(A \cup \{\mathbf{0}\})$ leads to

$$\mathbf{p} \in \partial(A^*) \;\Leftrightarrow\; H_\mathbf{p} \text{ is tangent to } \mathsf{ConvexHull}\,(A \cup \{\mathbf{0}\})$$

With this we know that the poles of all tangent planes to $\mathsf{ConvexHull}\,(A \cup \{\mathbf{0}\})$ build $\partial(A^*)$, the boundary of A^*. We will distinguish in the following between the case that $\mathbf{0} \in S(\delta, \mathbf{m})$ and that $\mathbf{0} \notin S(\delta, \mathbf{m})$.

Considering the case that $\mathbf{0} \in S(\delta, \mathbf{m})$, we have $\mathsf{ConvexHull}\,(S(\delta, \mathbf{m}) \cup \{\mathbf{0}\}) = S(\delta, \mathbf{m})$ and thus $\partial(S(\delta, \mathbf{m})^*) = s(\delta, \mathbf{m})^P$.

Considering the case that $\mathbf{0} \notin S(\delta, \mathbf{m})$, $\mathsf{ConvexHull}\,(S(\delta, \mathbf{m}) \cup \{\mathbf{0}\})$ is defined by the origin and all points on the boundary of the ball that are above the horizon seen from

the origin (refer to Figure 2.9).[1] We know that $s(\delta, \mathbf{m})^P$ is a two sheet hyperboloid in this case. The pole of a plane that contains the origin is a point at infinity (refer to the pink points in the left-hand image of Figure 2.9). Thus, the points on each side of the horizon of the ball's boundary lead to one sheet of the hyperboloid respectively. Since $\mathsf{ConvexHull}\,(S(\delta, \mathbf{m}) \cup \{\mathbf{0}\})$ contains the points of the ball's boundary above the horizon, the set of pole points of all tangent planes to $\mathsf{ConvexHull}\,(S(\delta, \mathbf{m}) \cup \{\mathbf{0}\})$ form the sheet of the hyperboloid whose convex hull contains the origin. The polar plane of the ball's center $\mathbf{m}$ is perpendicular to the axis of the hyperboloid and is lying between the two apex. Thus it does not intersect the hyperboloid and separates the two apex. So the poles of the tangent planes to $\mathsf{ConvexHull}\,(S(\delta, \mathbf{m}) \cup \{\mathbf{0}\})$ are

$$\partial(S(\delta, \mathbf{m})^*) = \{\mathbf{x} \in \mathbb{R}^3 \mid \mathbf{x} \in s(\delta, \mathbf{m})^P \ \wedge \ \langle \mathbf{x}, \mathbf{m} \rangle \leq 1\} \,. \tag{2.15}$$

In summary, for an arbitrary ball $S(\delta, \mathbf{m})$, which may contain the origin or not, the polar dual arise as

$$S(\delta, \mathbf{m})^* = \{\mathbf{x} \in \mathbb{R}^3 \mid \delta^2 \langle \mathbf{x}, \mathbf{x} \rangle - (1 - \langle \mathbf{m}, \mathbf{x} \rangle)^2 \leq 0 \ \wedge \ \langle \mathbf{x}, \mathbf{m} \rangle \leq 1\} \,. \tag{2.16}$$

By the intersection of the polar duals of the three balls we get the polar dual of the δ-offset of the triangle $\mathcal{A}$ as the set

$$\begin{aligned} (\mathcal{A} \oplus S_\delta)^* =& S(\delta, \mathbf{0})^* \cap S(\delta, \mathbf{b})^* \cap S(\delta, \mathbf{c})^* \\ =& \{\mathbf{x} \in \mathbb{R}^3 \mid \delta^2 \langle \mathbf{x}, \mathbf{x} \rangle - 1 \leq 0 \ \wedge \\ & \qquad \delta^2 \langle \mathbf{x}, \mathbf{x} \rangle - (1 - \langle \mathbf{b}, \mathbf{x} \rangle)^2 \leq 0 \ \wedge \ \langle \mathbf{x}, \mathbf{b} \rangle \leq 1 \ \wedge \\ & \qquad \delta^2 \langle \mathbf{x}, \mathbf{x} \rangle - (1 - \langle \mathbf{c}, \mathbf{x} \rangle)^2 \leq 0 \ \wedge \ \langle \mathbf{x}, \mathbf{c} \rangle \leq 1\} \end{aligned} \tag{2.17}$$

Some properties of the geometrical shape of $(\mathcal{A} \oplus S_\delta)^*$ can be easily derived as listed in the following. Refer additionally to Figure 2.10 which is an illustration of the correspondences between the δ-offset of a triangle $\mathcal{A} \oplus S_\delta$ in primal space and its dual $(\mathcal{A} \oplus S_\delta)^*$.

- Let $H_\mathbf{s}$ and $H_\mathbf{n}$ be the two tangent planes to the triangular faces of $\mathcal{A} \oplus S_\delta$. Let $\mathbf{s}$, $\mathbf{n}$ be the poles in dual space, respectively. Thus, the tangent plane to any point of the triangular faces of $\mathcal{A} \oplus S_\delta$ is dual to $\mathbf{s}$ or $\mathbf{n}$. We will denote $\mathbf{s}$ and $\mathbf{n}$ in the following as the south- and north-pole of $(\mathcal{A} \oplus S_\delta)^*$.
  $\rightarrow$ *The two triangle faces of $\mathcal{A} \oplus S_\delta$ in primal space correspond to the two points $\boldsymbol{s}$ and $\boldsymbol{n}$ in dual space.*

- The direction $(\mathbf{s} - \mathbf{n})$ will be denoted as the axis of $(\mathcal{A} \oplus S_\delta)^*$ and is parallel to the normal $\mathbf{b} \times \mathbf{c}$ of the triangle $\mathcal{A}$.

[1] For a convex object, the points on the horizon seen from the origin are the surface points with tangent planes though the origin, the points above the horizon are the surface points with a tangent plane for which the ball and the origin are on the same side, and the surface points below the horizon are the points with a tangent plane for which the ball and the origin are on different sides.

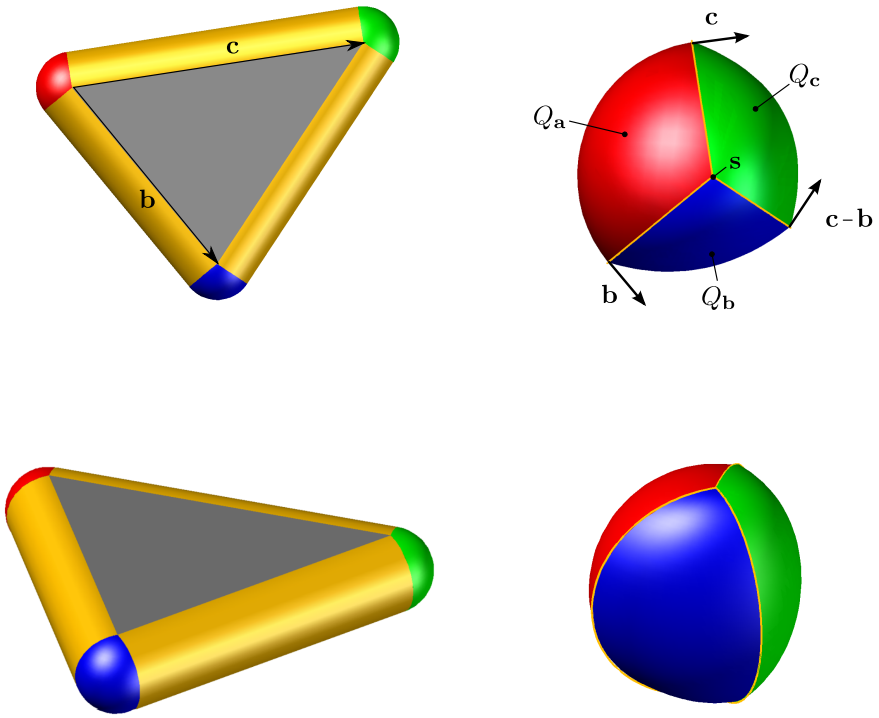


Fig. 2.10: The δ-offset of a triangle (left) with corner vertices $\mathbf{a} = \mathbf{0}, \mathbf{b}, \mathbf{c}$ and its polar dual (right), displayed stereoscopic (bottom) and displayed such that the normal of the triangle is projecting (top).

- All points in one straight line segment of a cylindrical part of $\mathcal{A} \oplus S_\delta$ have the same tangent plane. The poles of all tangent planes on one cylinder part build a planar curve in dual space that connects the south- and the north-pole $\mathbf{s}$ and $\mathbf{n}$ of $(\mathcal{A} \oplus S_\delta)^*$. For example we will denote the curve that is build by the poles of the tangent planes of the cylinder part between $\mathbf{a}$ and $\mathbf{b}$ by $c_{\mathbf{ab}}$. The curve $c_{\mathbf{ab}}$ is lying in the plane $\langle \mathbf{b}, \mathbf{x} \rangle = 0$. The curve $c_{\mathbf{bc}}$ is lying in the plane $\langle \mathbf{b} - \mathbf{c}, \mathbf{x} \rangle = 0$ and the curve $c_{\mathbf{ca}}$ is lying in the plane $\langle \mathbf{c}, \mathbf{x} \rangle = 0$. We will denote those planes as axis-planes.
  $\rightarrow$ *Each of the three cylinder patches of* $\mathcal{A} \oplus S_\delta$ *in primal space correspond to one of the three curves* $c_{\mathbf{ab}}$, $c_{\mathbf{bc}}$ *and* $c_{\mathbf{ca}}$ *in dual space.*
- The surface of $(\mathcal{A} \oplus S_\delta)^*$ consists of three surface patches, which we will denote as $Q_{\mathbf{a}}$, $Q_{\mathbf{b}}$ and $Q_{\mathbf{c}}$. The surface patches are restricted by the axis planes. Two surface patches contact in the curves that are lying in the axis planes. All three patches contact in the poles $\mathbf{s}$ and $\mathbf{n}$.
  - The poles of all tangent planes to the sphere patch of $\mathcal{A} \oplus S_\delta$ that is a subset of $s(\delta, \mathbf{0})$ build the surface patch $Q_{\mathbf{a}}$. $Q_{\mathbf{a}}$ is a subset of a sphere and is bounded by the curves $c_{\mathbf{ab}}$ and $c_{\mathbf{ca}}$, thus
    $$Q_{\mathbf{a}} = \{\mathbf{x} \in \ s(\delta, \mathbf{0})^P \mid \langle \mathbf{b}, \mathbf{x} \rangle \leq 0 \wedge \langle \mathbf{c}, \mathbf{x} \rangle \leq 0\} \,. \tag{2.18}$$
  - The poles of all tangent planes to the sphere patch of $\mathcal{A} \oplus S_\delta$ that is a subset of $s(\delta, \mathbf{b})$ build the surface patch $Q_{\mathbf{b}}$. $Q_{\mathbf{b}}$ can be the subset of an ellipsoid, a

paraboloid or a hyperboloid that is bounded by the curves $c_{\mathbf{ab}}$ and $c_{\mathbf{bc}}$, thus

$$Q_{\mathbf{b}} = \{\mathbf{x} \in \ s(\delta, \mathbf{b})^P \mid \langle \mathbf{b}, \mathbf{x} \rangle \geq 0 \wedge \langle \mathbf{c} - \mathbf{b}, \mathbf{x} \rangle \leq 0 \wedge \langle \mathbf{b}, \mathbf{x} \rangle \leq 1\} \ . \quad (2.19)$$

– Analogous to $Q_{\mathbf{b}}$ we can derive $Q_{\mathbf{c}}$, thus

$$Q_{\mathbf{b}} = \{\mathbf{x} \in \ s(\delta, \mathbf{c})^P \mid \langle \mathbf{c}, \mathbf{x} \rangle \geq 0 \wedge \langle \mathbf{c} - \mathbf{b}, \mathbf{x} \rangle \geq 0 \wedge \langle \mathbf{c}, \mathbf{x} \rangle \leq 1\} \ . \quad (2.20)$$

→ *Each of the three sphere patches of $\mathcal{A} \oplus S_\delta$ in primal space correspond to one of the three surface patches $Q_{\boldsymbol{a}}$, $Q_{\boldsymbol{b}}$ and $Q_{\boldsymbol{c}}$ in dual space.*

2.2.4. Dual Intersection Tests between Convex Objects

We will map the intersection test between arbitrary convex objects from primal space into dual space. The dual triangle-triangle tolerance test, which is an intersection test between two convex objects (refer to Equation 2.2), results from this directly.

We propose the following criterion:

Proposition 2.3 (Dual Intersection): *Given a closed convex set $A \subset \mathbb{R}^d$ that contains the origin and another closed convex set $B \subset \mathbb{R}^d$, then*

$$A \text{ and } B \text{ are intersecting } \Leftrightarrow \ \exists \boldsymbol{b} \in B : \ \mathring{A}^* \cap H_{\boldsymbol{b}} = \emptyset \ . \quad (2.21)$$

Proof 5: *The equivalence follows directly from Equation 2.9: In the case there is a point $\boldsymbol{b} \in B$ with $\mathring{A}^* \cap \ H_{\boldsymbol{b}} = \emptyset$, the pole $\boldsymbol{b}$ of $H_{\boldsymbol{b}}$ must be in A, thus, A and B are intersecting. In the case there is no $\boldsymbol{b} \in B$ with $\mathring{A}^* \cap \ H_{\boldsymbol{b}} = \emptyset$, we have for all $\boldsymbol{b} \in B$ that $\mathring{A}^* \cap \ H_{\boldsymbol{b}} \neq \emptyset$. Thus, all $\boldsymbol{b}$ are not in A, thus, A and B are not intersecting.* ∎

By Proposition 2.3 a dual predicate for an intersection test between convex object is given. As soon as one point $\mathbf{b} \in B$ is found with $\mathring{A}^* \cap H_{\mathbf{b}} = \emptyset$ we know that A and B are intersecting. This is the same kind of early out as in the presented *feature distance test* (refer to Section 2.1.3). In the following we want to derive a dual predicate for the separation of two convex object. By such a predicate the same kind of early out as in the presented *separating plane test* (refer to Section 2.1.4) can be used.

Lemma 2.3: *Let B be a convex set of points in $\mathbb{R}^d$ that does not contain the origin. Then the set of points in dual space for which the dual hyperplanes in primal space separate B from the origin is*

$$\hat{B}^* := \bigcap_{\boldsymbol{b} \in B} \mathring{H}^+_{\boldsymbol{b}} \ , \quad (2.22)$$

where $\mathring{H}^+_{\boldsymbol{b}} := \{\boldsymbol{x} \in \mathbb{R}^d \mid \langle \boldsymbol{x}, \boldsymbol{b} \rangle > 1\}$ is an open halfspace.

Proof 6: *For every* $\boldsymbol{p} \in \hat{B}^*$ *it is* $\boldsymbol{p} \notin H_{\boldsymbol{b}}^-$ *for all* $\boldsymbol{b} \in B$*. So we have* $\boldsymbol{p} \notin H_{\boldsymbol{b}}^- \Leftrightarrow \langle \boldsymbol{p}, \boldsymbol{b} \rangle > 1 \Leftrightarrow \boldsymbol{b} \notin H_{\boldsymbol{p}}^-$*. This holds true for every* $\boldsymbol{b} \in \hat{B}^*$ *and therefore all* $\boldsymbol{b} \in \hat{B}^*$ *are on the opposite side of the plane* $H_{\boldsymbol{p}}^-$ *than the origin. This means the polar hyperplanes* $H_{\boldsymbol{p}}$ *of all* $\boldsymbol{p}$ *separate* B *from the origin.* ∎

Using Lemma 2.3 we formulate the dual predicate for the separation of two convex objects.

Proposition 2.4 (Dual Intersection): *Given a closed convex set* $A \subset \mathbb{R}^d$ *that contains the origin and another closed convex set* $B \subset \mathbb{R}^d$*, then*

$$A \text{ and } B \text{ are not intersecting} \Leftrightarrow A^* \cap \hat{B}^* \neq \emptyset \,. \tag{2.23}$$

The set of points in the dual space with polar hyperplanes that separate A *and* B *in the primal space is given as the set* $A^* \cap \hat{B}^*$*.*

Proof 7: *We know that two convex objects are disjoint iff there exists a hyperplane* $H_{\boldsymbol{s}}$ *in the primal space that separates the two objects, thus*

$$\begin{aligned} A \text{ and } B \text{ are not intersecting} &\Leftrightarrow \exists H_{\boldsymbol{s}} \subset \mathbb{R}^d : \; A \subseteq H_{\boldsymbol{s}}^- \wedge B \subseteq \mathring{H}_{\boldsymbol{s}}^+ \\ &\Leftrightarrow \exists \boldsymbol{s} \in \mathbb{R}^d : (\langle \boldsymbol{s}, \boldsymbol{a} \rangle \leq 1 \; \forall \boldsymbol{a} \in A) \wedge (\langle \boldsymbol{s}, \boldsymbol{b} \rangle > 1 \; \forall \boldsymbol{b} \in B) \,. \end{aligned}$$

According to Definition 2.1, $\langle \boldsymbol{s}, \boldsymbol{a} \rangle \leq 1 \; \forall \boldsymbol{a} \in A$ *means in dual space that* $\boldsymbol{s}$ *must be* $\in A^*$*. According to Lemma 2.3,* $\langle \boldsymbol{s}, \boldsymbol{b} \rangle > 1 \; \forall \boldsymbol{b} \in B$ *means in dual space that* $\boldsymbol{s}$ *must be* $\in \hat{B}^*$*. Altogether* $\boldsymbol{s}$ *must be* $\in A^* \cap \hat{B}^*$*. Thus, the polar hyperplanes* $H_{\boldsymbol{s}}$ *of all* $\boldsymbol{s} \in A^* \cap \hat{B}^*$ *separate* A *and* B *in primal space.* ∎

2.2.5. Dualization of the Triangle Triangle Tolerance Test

We are given two triangle $\mathcal{A}$ and $\mathcal{B}$ in the primal space and want to determine whether they are tolerance violating for the given tolerance value δ, i.e.

$$\mathcal{A} \text{ and } \mathcal{B} \text{ are tolerance violating} \Leftrightarrow (\mathcal{A} \oplus S_\delta) \cap \mathcal{B} \neq \emptyset \,,$$

where $\mathcal{A} \oplus S_\delta$ is the δ-offset of triangle $\mathcal{A}$. W.l.o.g. we will assume in the following that $\mathcal{A} \oplus S_\delta$ contains the origin.

Because $\mathcal{A} \oplus S_\delta$ and $\mathcal{B}$ are both convex sets and $\mathcal{A} \oplus S_\delta$ contains the origin, we can transcribe our problem statement to the dual space by Proposition 2.3 as

$$\mathcal{A} \text{ and } \mathcal{B} \text{ are tolerance violating} \Leftrightarrow \exists \mathbf{b} \in \mathcal{B} : \; \left(\mathcal{A} \mathbin{\mathring{\oplus}} S_\delta\right)^* \cap H_{\mathbf{b}} = \emptyset \,, \tag{2.24}$$

or equivalent by Proposition 2.4 as

$$\mathcal{A} \text{ and } \mathcal{B} \text{ are tolerance violating} \Leftrightarrow (\mathcal{A} \oplus S_\delta)^* \cap \hat{\mathcal{B}}^* = \emptyset \,. \tag{2.25}$$

with $(\mathcal{A} \oplus S_\delta)^*$ as described already in Section 2.2.3 and with

$$\hat{\mathcal{B}}^* = \mathring{H}^+_{\mathbf{u}} \cup \mathring{H}^+_{\mathbf{v}} \cup \mathring{H}^+_{\mathbf{w}} \,, \tag{2.26}$$

where $\mathbf{u}$, $\mathbf{v}$, and $\mathbf{w}$ are the three vertices of the triangle $\mathcal{B}$. $\hat{\mathcal{B}}^* = \mathring{H}^+_{\mathbf{u}} \cup \mathring{H}^+_{\mathbf{v}} \cup \mathring{H}^+_{\mathbf{w}}$ can easily be derived from Lemma 2.3 for the dual of a triangle that is defined by $\mathcal{B}^* = H^-_{\mathbf{u}} \cap H^-_{\mathbf{v}} \cap H^-_{\mathbf{w}}$, as given in Section 2.2.2.

2.2.6. The Dual Approach

The basic idea of the *dual approach* is to improve the *separating plane test*, as presented in Section 2.1.4, by performing some of the separating plane tests in dual space. Therefor we consider Equation 2.25.

> *Annotation:* Within this section we will not distinguish strictly between the boundary and the inner of the sets according to Equation 2.8 and 2.9. Thus, we will be slightly inexact and say $\mathbf{p} \in A \;\Leftrightarrow\; A^* \cap H_{\mathbf{p}} = \emptyset$. The consequence is that we consider for Equation 2.25 the set $\hat{\mathcal{B}}^*$ as closed set with
>
> $$\hat{\mathcal{B}}^* = H^+_{\mathbf{u}} \cup H^+_{\mathbf{v}} \cup H^+_{\mathbf{w}} \,.$$
>
> (Compare with Equation 2.26). So we are not correct within contact situations – which is however not relevant when calculating with floating point arithmetic.

We showed in Section 2.1.4 which planes have to be tested for the *separating plane test*. In summary there are 38 planes that have to be tested in order to be sure that the triangles are tolerance violating. 29 of the 38 planes are parallel to at least one of the triangles' edges. For the dual approach we will map these 29 separating plane tests to the dual space as follows:

The set $\hat{\mathcal{B}}^*$ is the intersection of the three halfspaces $H^+_{\mathbf{u}}$, $H^+_{\mathbf{v}}$, and $H^+_{\mathbf{w}}$. In general, each two of the planes $H_{\mathbf{u}}$, $H_{\mathbf{v}}$, $H_{\mathbf{w}}$ intersect in a straight line.

Proposition 2.5: *Let $\boldsymbol{u}, \boldsymbol{v} \in \mathbb{R}^3$ and $H_{\boldsymbol{u}}, H_{\boldsymbol{v}}$ their polar planes that intersect in the straight line $\tilde{g}_{\boldsymbol{uv}}$. Then every point $\boldsymbol{p} \in \tilde{g}_{\boldsymbol{uv}}$ is the pole of a plane $H_{\boldsymbol{p}}$ in the sheaf of planes that share the line through the points $\boldsymbol{u}$ and $\boldsymbol{v}$ and vice versa.*

Proof 8: *If $\boldsymbol{p} \in \tilde{g}_{\boldsymbol{uv}}$, then $\boldsymbol{p} \in H_{\boldsymbol{u}} \wedge \boldsymbol{p} \in H_{\boldsymbol{v}}$. If $H_{\boldsymbol{p}}$ is the sheaf of planes that share the line through the points $\boldsymbol{u}$ and $\boldsymbol{v}$, then $\boldsymbol{u} \in H_{\boldsymbol{p}} \wedge \boldsymbol{v} \in H_{\boldsymbol{p}}$. Thus, the proposition holds true, iff $\boldsymbol{p} \in H_{\boldsymbol{u}} \wedge \boldsymbol{p} \in H_{\boldsymbol{v}} \Leftrightarrow \boldsymbol{u} \in H_{\boldsymbol{p}} \wedge \boldsymbol{v} \in H_{\boldsymbol{p}}$ is hold. This is true, as the right-hand side as well as the left-hand side of the equivalence lead to the term $\langle \boldsymbol{u}, \boldsymbol{p} \rangle = 1 \wedge \langle \boldsymbol{v}, \boldsymbol{p} \rangle = 1$.* ∎

We know by Proposition 2.4 that the set $(\mathcal{A} \oplus S_\delta)^* \cap \hat{\mathcal{B}}^*$ is the set of points in the dual space where each point has a polar plane in the primal space that separates the triangle $\mathcal{B}$ from $\mathcal{A} \oplus S_\delta$. Thus, with Proposition 2.5, every point $\mathbf{s}$ that is on the straight line $\tilde{g}_{\mathbf{uv}}$ and in $(\mathcal{A} \oplus S_\delta)^* \cap \hat{\mathcal{B}}^*$ has a polar plane in the primal space that contains the edge $e_{\mathbf{uv}}$ and that separates $\mathcal{B}$ from $\mathcal{A} \oplus S_\delta$, i.e.

$$\tilde{g}_{\mathbf{uv}} \cap (\mathcal{A} \oplus S_\delta)^* \cap \hat{\mathcal{B}}^* \neq \emptyset \Leftrightarrow \text{there is a separating plane that contains } e_{\mathbf{uv}} .$$

We distinguish now between the general case that the planes $H_{\mathbf{u}}$, $H_{\mathbf{v}}$, $H_{\mathbf{w}}$ intersect in one point and the special case that the planes $H_{\mathbf{u}}$, $H_{\mathbf{v}}$, $H_{\mathbf{w}}$ do not intersect in one point. $H_{\mathbf{u}}$, $H_{\mathbf{v}}$, and $H_{\mathbf{w}}$ intersect in one point, iff the plane that contains the points $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ in primal space does not contain the origin. This is the case iff $det(\mathbf{u}, \mathbf{v}, \mathbf{w}) \neq 0$.

In the general case, when $det(\mathbf{u}, \mathbf{v}, \mathbf{w}) \neq 0$, the set $\hat{\mathcal{B}}^*$ is an infinite pyramid. The lines $\tilde{g}_{\mathbf{uv}}$, $\tilde{g}_{\mathbf{vw}}$, $\tilde{g}_{\mathbf{wu}}$ contain the pyramid's edges.

Corollary 2.2: *Let $det(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) \neq 0$ and $\tilde{r}_{\boldsymbol{uv}}$ be the ray $\tilde{r}_{\boldsymbol{uv}} = \tilde{g}_{\boldsymbol{uv}} \cap \hat{\mathcal{B}}^*$. Then we know*

$$\tilde{r}_{\boldsymbol{uv}} \cap (\mathcal{A} \oplus S_\delta)^* \neq \emptyset \Leftrightarrow \textit{there exists a separating plane containing } e_{\boldsymbol{uv}} . \tag{2.27}$$

This is analogous for the edges $e_{\mathbf{vw}}$ and $e_{\mathbf{wu}}$.

In the special case, when $det(\mathbf{u}, \mathbf{v}, \mathbf{w}) = 0$, the set $\hat{\mathcal{B}}^*$ is an infinite prism. The planes $H_{\mathbf{u}}$, $H_{\mathbf{v}}$ and $H_{\mathbf{w}}$ intersect in two or three straight lines, thus one of the straight lines $\tilde{g}_{\mathbf{uv}}, \tilde{g}_{\mathbf{vw}}, \tilde{g}_{\mathbf{wu}}$ can be empty. A straight line $\tilde{g}_{\mathbf{uv}}$ is empty, iff $\mathbf{u} \times \mathbf{v} = \mathbf{0}$. The intersection between $\hat{\mathcal{B}}^*$ and a non-empty straight lines is either the straight line itself or empty.

Corollary 2.3: *Let $det(\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w}) = 0$, then*

$$\tilde{g}_{\boldsymbol{uv}} \cap \hat{\mathcal{B}}^* \neq \emptyset \ \wedge \ \tilde{g}_{\boldsymbol{uv}} \cap (\mathcal{A} \oplus S_\delta)^* \neq \emptyset \quad \Leftrightarrow \quad \textit{there exists a separating plane containing } e_{\boldsymbol{uv}} . \tag{2.28}$$

This is analogous for the edges $e_{\mathbf{vw}}$ and $e_{\mathbf{wu}}$.

Testing the criterion of Equation 2.27 or of Equation 2.28 for the edges $e_{\mathbf{uv}}$, $e_{\mathbf{vw}}$, and $e_{\mathbf{wu}}$ replaces all separating plane tests with planes parallel to the edges of triangle $\mathcal{B}$.

Since $(\mathcal{A} \oplus S_\delta) \cap \mathcal{B} = \mathcal{A} \cap (\mathcal{B} \oplus S_\delta)$ we can change the roles of the triangles $\mathcal{A}$ and $\mathcal{B}$. Therefore we transform the triangles such that $\mathbf{u} = \mathbf{0}$ and apply Equation 2.27 or Equation 2.28 for the edges $e_{\mathbf{ab}}$ with

$$\tilde{r}_{\mathbf{ab}} \cap (\mathcal{B} \oplus S_\delta)^* \neq \emptyset \Leftrightarrow \text{there exists a separating plane containing } e_{\mathbf{ab}}$$

or

$$\tilde{g}_{\mathbf{ab}} \cap \hat{\mathcal{A}}^* \neq \emptyset \ \wedge \ \tilde{g}_{\mathbf{ab}} \cap (\mathcal{B} \oplus S_\delta)^* \neq \emptyset \quad \Leftrightarrow \quad \text{there exists a separating plane containing } e_{\mathbf{ab}} .$$

and analogously for $e_{\mathbf{bc}}$ and $e_{\mathbf{ca}}$.

In summary we have 6 intersection tests of a ray or of a straight line with the dual of the δ-offset of a triangle in dual space instead of 29 separating plane tests between a triangle and the δ-offset of another triangle in the primal space.

By these 6 intersection tests the planes that contain the triangles $\mathcal{A}$ and $\mathcal{B}$ are tested each three times implicitly. This is because the plane that contains a triangle is in every sheaf of planes that contains an edge of the triangle. Nevertheless, our benchmarks showed that considering these two planes one more time separately in advance can improve the calculation performance.

Proposition 2.6: *Let $\boldsymbol{u}, \boldsymbol{v}, \boldsymbol{w} \in \mathbb{R}^3$ and $H_{\boldsymbol{u}}, H_{\boldsymbol{v}}, H_{\boldsymbol{w}}$ their polar planes that intersect in the point $\boldsymbol{f}$. Then $\boldsymbol{f}$ is the pole of a plane $H_{\boldsymbol{f}}$ that contains $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$.*

Proof 9: *If $\boldsymbol{f}$ is the point of intersection of the planes $H_{\boldsymbol{u}}$, $H_{\boldsymbol{v}}$, and $H_{\boldsymbol{w}}$, then $\boldsymbol{f} \in H_{\boldsymbol{u}} \wedge \boldsymbol{f} \in H_{\boldsymbol{v}} \wedge \boldsymbol{f} \in H_{\boldsymbol{w}}$. If $H_{\boldsymbol{f}}$ contains $\boldsymbol{u}$, $\boldsymbol{v}$, and $\boldsymbol{w}$, then $\boldsymbol{u} \in H_{\boldsymbol{f}} \wedge \boldsymbol{v} \in H_{\boldsymbol{f}} \wedge \boldsymbol{w} \in H_{\boldsymbol{f}}$. Thus, the proposition holds true, iff $\boldsymbol{f} \in H_{\boldsymbol{u}} \wedge \boldsymbol{f} \in H_{\boldsymbol{v}} \wedge \boldsymbol{f} \in H_{\boldsymbol{w}} \Leftrightarrow \boldsymbol{u} \in H_{\boldsymbol{f}} \wedge \boldsymbol{v} \in H_{\boldsymbol{f}} \wedge \boldsymbol{w} \in H_{\boldsymbol{f}}$. This is true, as the right-hand side as well as the left-hand side of the equivalence lead to the term $\langle \boldsymbol{u}, \boldsymbol{f} \rangle = 1 \wedge \langle \boldsymbol{v}, \boldsymbol{f} \rangle = 1 \wedge \langle \boldsymbol{w}, \boldsymbol{f} \rangle = 1$.* ∎

We know by Proposition 2.4 that the set $(\mathcal{A} \oplus S_\delta)^* \cap \hat{\mathcal{B}}^*$ is the set of points in the dual space where each point has a polar plane in the primal space that separates the triangle $\mathcal{B}$ from $\mathcal{A} \oplus S_\delta$. Thus, with Proposition 2.6, we have the following corollary:

Corollary 2.4: *Let $\boldsymbol{f} = H_{\boldsymbol{u}} \cap H_{\boldsymbol{v}} \cap H_{\boldsymbol{w}}$, then we know*

$$\boldsymbol{f} \in (\mathcal{A} \oplus S_\delta)^* \cap \hat{\mathcal{B}}^* \quad \Leftrightarrow \quad H_{\boldsymbol{f}} \text{ is a separating plane .} \tag{2.29}$$

With Corollary 2.2, 2.3, and 2.4 we have the foundations of our dual approach. In the following the algorithm of the dual approach is presented and it is explained more detailed how the predicates of the corollaries are calculated.

The Principal Algorithm

Algorithm 1 shows the basic work flow of the dual approach for testing the triangles $\mathcal{A}$ and $\mathcal{B}$ on tolerance violation.

As a first step we test whether the plane that contains triangle $\mathcal{B}$ is a separating plane (Line 1) and whether the plane that contains triangle $\mathcal{A}$ is a separating plane (Line 3). This test is performed in dual space and we denote it as the *dual point test*. The algorithm is given in Algorithm 2 and described more detailed later.

If none of the two planes is a separating plane, we perform distance tests between all vertex-vertex feature pairs and separating plane tests with planes perpendicular to the vertex-vertex directions in primal space (Line 5 to 9). This is similar to the *combined approach* presented in Section 2.1.5.

If this does not lead to the conclusion that the triangles are definitely tolerance violating or definitely not tolerance violating, the remaining separating planes have to be tested. Therefore the *dual edge test* is performed. It tests whether there is a separating plane parallel to the edges of triangle $\mathcal{B}$ (Line 10) and whether there is a separating plane parallel to the edges of triangle $\mathcal{A}$ (Line 12), based on the idea described above. The algorithm is given in Algorithm 3 and described more detailed later.

The two planes that contain the triangles $\mathcal{A}$ and $\mathcal{B}$ are tested multiple times. The first time they are tested by the dual point test in Line 1 and Line 3. Then they are tested implicitly with every edge test by the dual edge tests (Line 10 and 12). Thus, the dual point test is redundant and the algorithm is also correct without Line 1-4. However, our benchmarks showed in our application case a better performance when executing additionally Line 1-4. We implemented therefore two versions of the dual approach. One version is exactly according to Algorithm 1. The other version is according to Algorithm 1, but without the Lines 1-4.

Algorithm 1: dualTest($\mathcal{A}, \mathcal{B}, \delta$)

Data: $\mathcal{A}$ // a triangle with vertices **a**, **b** and **c**
$\mathcal{B}$ // a triangle with vertices **u**, **v** and **w**
δ // the tolerance value

Result: true // if $\mathcal{A}$ and $\mathcal{B}$ are tolerance violating
false // if $\mathcal{A}$ and $\mathcal{B}$ are not tolerance violating

```
1  if dualPointTest(A, B, δ) then
2      return false
3  if dualPointTest(B, A, δ) then
4      return false
5  for all pairs of vertices in primal space do
6      if the distance between the vertices is ≤ δ then
7          return true
8      if the difference vector between the vertices defines a separating axis then
9          return false
10 if dualEdgeTest(A, B, δ) then
11     return false
12 if dualEdgeTest(B, A, δ) then
13     return false
14 return true
```

The Dual Point Test

Algorithm 2 shows the work flow of the *dual point test*. It tests whether the plane $H_{\mathbf{f}}$ that contains the triangle $\mathcal{B}$ is a separating plane between the triangle $\mathcal{B}$ and the δ-offset $\mathcal{A} \oplus S_\delta$. In dual space this is tested by testing the criterion in Corollary 2.4. Out of performance reasons we check this in three steps. In each step the triangles are translated such that the origin is equal to one of the three vertices of triangle $\mathcal{A}$ (Line 1 and 2).

Let $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ be the vertices of the translated triangle $\mathcal{A}$ and with $\mathbf{a} = \mathbf{0}$ in the first step. In the second step we translate such that $\mathbf{b} = \mathbf{0}$ and in the third step such that $\mathbf{c} = \mathbf{0}$. In all three steps we denote the two vertices of triangle $\mathcal{A}$ that are $\neq \mathbf{0}$ by $\mathbf{p}$ and $\mathbf{q}$. Further let $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ be the vertices of the translated triangle $\mathcal{B}$ in every step.

The pole $\mathbf{f}$ of the plane $H_{\mathbf{f}}$ is computed as

$$\mathbf{f} = \frac{\mathbf{n}_{\mathcal{B}}}{\det(\mathbf{u}, \mathbf{v}, \mathbf{w})} \tag{2.30}$$

with $\mathbf{n}_{\mathcal{B}} = (\mathbf{v} - \mathbf{u}) \times (\mathbf{w} - \mathbf{u})$ being the normal of the triangle $\mathcal{B}$ (Line 5).

The denominator on the right-hand side of Equation 2.30 can be zero. This is the case iff the plane $H_{\mathbf{f}}$ goes through the origin (Line 3). In this case we know that $H_{\mathbf{f}}$ cannot separate $\mathcal{B}$ and $\mathcal{A} \oplus S_\delta$.

For every triple $(\mathbf{m}, \mathbf{p}, \mathbf{q})$ we test whether $\mathbf{f}$ is inside of a ball-slice $\mathscr{S}$ with

$$\mathscr{S} := \{\mathbf{x} \in \mathbb{R}^3 \;\mid\; \|\mathbf{x}\| \leq \frac{1}{\delta} \wedge \langle \mathbf{f}, \mathbf{p} \rangle \leq 0 \wedge \langle \mathbf{f}, \mathbf{q} \rangle \leq 0\} \,. \tag{2.31}$$

Algorithm 2: dualPointTest($\mathcal{A}, \mathcal{B}, \delta$)

Data: $\mathcal{A}$ // a triangle with vertices $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$
$\mathcal{B}$ // a triangle with vertices $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$
δ // the tolerance value

Result: true // if a separating plane between $\mathcal{A}$ and $\mathcal{B}$ is found
false // otherwise

```
1 foreach triple (m, p, q) ∈ {(a, b, c), (b, c, a), (c, a, b)} do
2     p, q, B ← translate p, q, and B by the vector −m
3     if the plane through triangle B contains 0 then
4         return false
5     f ← the pole of the plane through triangle B
6     if ⟨f, f⟩ ≤ 1/δ² ∧ ⟨f, p⟩ ≤ 0 ∧ ⟨f, q⟩ ≤ 0 then
7         return true
8 return false
```

Thus, we test whether $\mathbf{f} \in \mathscr{S}$ (Line 6). This means in primal space that the plane $H_{\mathbf{f}}$ separates the translated triangle $\mathcal{B}$ from the set $\tilde{\mathcal{A}} \oplus S_\delta$ with $\tilde{\mathcal{A}}$ an infinite triangle that can be given by the homogeneous coordinates $(\mathbf{0},1)^T, (\mathbf{p},0)^T, (\mathbf{q},0)^T$. In the case $H_{\mathbf{f}}$ separates $\mathcal{B}$ and $\tilde{\mathcal{A}} \oplus S_\delta$, it surely separates $\mathcal{B}$ and $\mathcal{A} \oplus S_\delta$.

In the case $H_{\mathbf{f}}$ separates $\mathcal{B}$ and $\tilde{\mathcal{A}} \oplus S_\delta$ in none of the three steps, $H_{\mathbf{f}}$ does not separate $\mathcal{B}$ and $\mathcal{A} \oplus S_\delta$.

The Dual Edge Test

Algorithm 3 shows the work flow of the *dual edge test*. It tests whether a plane parallel to an edge of triangle $\mathcal{B}$ is a separating plane between the triangle $\mathcal{B}$ and the δ-offset $\mathcal{A} \oplus S_\delta$. In dual space this is tested by testing the criterion in Corollary 2.2 or 2.3.

Testing for separating planes parallel with the edges of triangle $\mathcal{B}$, needs the precondition that $\mathcal{A}$ contains the origin (Line 1). We will translate both triangles such that one vertex of $\mathcal{A}$ is the origin. Let $\mathbf{a}$, $\mathbf{b}$, $\mathbf{c}$ be the vertices of the translated triangle $\mathcal{A}$ with $\mathbf{a} = \mathbf{0}$, and $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ be the vertices of the translated triangle $\mathcal{B}$.

We mentioned above that we have to distinguish between the general case where the plane $H_{\mathbf{f}}$, which contains the triangle $\mathcal{B}$, does not contain the origin and the special case where it contains the origin (Line 3).

General Case

In the general case the rays $\tilde{r}_{\mathbf{uv}}$, $\tilde{r}_{\mathbf{vw}}$ and $\tilde{r}_{\mathbf{wu}}$ are determined (Line 4). The polar planes $H_{\mathbf{u}}, H_{\mathbf{v}}, H_{\mathbf{w}}$ of $\mathbf{u}$, $\mathbf{v}$, $\mathbf{w}$ intersect in the point $\mathbf{f}$ as given already above by Equation 2.30. Consider for example the ray $\tilde{r}_{\mathbf{uv}}$. The direction of the ray is parallel to the vector $\mathbf{u} \times \mathbf{v}$. So the ray is determined by

$$\tilde{r}_{\mathbf{uv}}: \quad \mathbf{x}(\lambda) = \mathbf{f} + s\lambda(\mathbf{u} \times \mathbf{v}), \quad \lambda \in [0, \infty) \ ,$$

where $s = \operatorname{sgn}(\det(\mathbf{u}, \mathbf{v}, \mathbf{w}))$. The commitment for s is necessary to define the ray such that it contacts the set $\hat{\mathcal{B}}^*$, which is the case iff $\langle \mathbf{x}(\lambda), \mathbf{w} \rangle \geq 1$ is fulfilled.

The point $\mathbf{f}$ can be located relatively far way from the origin – depending on how close the plane $H_{\mathbf{f}}$ is to the origin. If $\mathbf{f}$ is far away from the origin this can cause numerical instabilities for the intersection computation of the ray with the set $\mathcal{A} \oplus \delta$, which contains the origin. An alternative representation of the ray is given by

$$\tilde{r}_{\mathbf{uv}}: \quad \mathbf{x}(\lambda) = \mathbf{l} + s\lambda(\mathbf{u} \times \mathbf{v}), \quad \lambda \in [\lambda_{min}, \infty) \ , \tag{2.32}$$

where $s = \operatorname{sgn}(\det(\mathbf{u}, \mathbf{v}, \mathbf{w}))$ and

$$\mathbf{l} = \frac{(\mathbf{v} - \mathbf{u}) \times (\mathbf{u} \times \mathbf{v})}{\|\mathbf{u} \times \mathbf{v}\|^2} \quad \text{and} \quad \lambda_{min} = \frac{1 - \langle \mathbf{w}, \mathbf{l} \rangle}{|\det(\mathbf{u}, \mathbf{v}, \mathbf{w})|} \ . \tag{2.33}$$

$\mathbf{l}$ is the dropped perpendicular foot of the origin on the straight line that contains the ray $\tilde{r}_{\mathbf{uv}}$. The formula to calculate $\mathbf{l}$ can be easily derived from the intersection of the three planes $H_{\mathbf{u}}, H_{\mathbf{v}}$, and $\langle \mathbf{u} \times \mathbf{v}, \mathbf{x} \rangle = 0$. λ_{min} is the parameter shift such that $\mathbf{x}(\lambda_{min}) = \mathbf{f}$. The formula to calculate λ_{min}, as well as the commitment of s can be

Algorithm 3: dualEdgeTest($\mathcal{A}, \mathcal{B}, \delta$)

Data: $\mathcal{A}$ // a triangle with vertices $\mathbf{a}$, $\mathbf{b}$ and $\mathbf{c}$
$\mathcal{B}$ // a triangle with vertices $\mathbf{u}$, $\mathbf{v}$ and $\mathbf{w}$
δ // the tolerance value

Result: true // if a separating plane between $\mathcal{A}$ and $\mathcal{B}$ is found
false // otherwise

```
1  A, B ← translate A and B by the vector −a
2  foreach pair (p, q) ∈ {(u, v), (v, w), (w, u)} do
3      if the plane through triangle B does not contain the origin then
4          x(λ) ← determine the ray r̃_pq
5      else
6          if g̃_pq exists and g̃_pq contacts B̂* then
7              x(λ) ← determine the straight line g̃_pq
8      if x(λ) is defined then
9          I_a, I_b, I_c ← intervals in λ, where x(λ) is in the scope of Q_a, Q_b and Q_c
10         foreach m ∈ {a, b, c} do
11             if I_m ≠ ∅ then
12                 q_m(λ) ← quadric equation with roots that solve x(λ) ∈ s(δ, m)^P
13                 if q_m(λ) = 0 has a solution ∈ I_m then
14                     return true
15 return false
```

derived from the condition that the ray has to contact the set $\hat{\mathcal{B}}^*$, which is the case iff $\langle \mathbf{x}(\lambda), \mathbf{w} \rangle \geq 1$ is fulfilled.

Special Case

In the special case the straight lines $\tilde{g}_{\mathbf{uv}}$, $\tilde{g}_{\mathbf{vw}}$ and $\tilde{g}_{\mathbf{wv}}$ are determined similarly (Line 7) – provided they exists and contact $\hat{\mathcal{B}}^*$ (Line 6). Then, considering for example the straight line $\tilde{g}_{\mathbf{uv}}$, $\tilde{g}_{\mathbf{uv}}$ is determined as

$$\tilde{g}_{\mathbf{uv}}: \quad \mathbf{x}(\lambda) = \mathbf{l} + \lambda \mathbf{n}_{\mathcal{B}}, \quad \lambda \in \mathbb{R}\,, \tag{2.34}$$

where $\mathbf{l}$ is calculated as above (refer to Equation 2.33). In this special case we can use $\mathbf{n}_{\mathcal{B}}$ as direction vector because $\mathbf{u} \times \mathbf{v} \parallel \mathbf{n}_{\mathcal{B}}$.

The straight line $\tilde{g}_{\mathbf{uv}}$ exists iff $\mathbf{u} \times \mathbf{v} \neq \mathbf{0}$. The straight line $\tilde{g}_{\mathbf{uv}}$ contacts $\hat{\mathcal{B}}^*$ iff it does not contact $\mathcal{B}^*$. Then is must be outside of the halfspace $H_{\mathbf{w}}$. Let $\mathbf{p}_w$ a point on the plane $H_{\mathbf{w}}$ then $\tilde{g}_{\mathbf{uv}} \cap \hat{\mathcal{B}}^* \neq \emptyset \Leftrightarrow \langle \mathbf{l} - \mathbf{p}_w, \mathbf{w} \rangle > 0$.

Intersection with $(\mathcal{A} \oplus S_\delta)^*$

In the case we have defined a ray or a straight line $x(\lambda)$ as described above (Line 8), we want to test whether it intersects the polar dual $(\mathcal{A} \oplus S_\delta)^*$. Let for the following consideration $\mathbb{D} = [\lambda_{min}, \infty)$ in the case we have defined a ray or $\mathbb{D} = \mathbb{R}$ in the case we have defined a straight line.

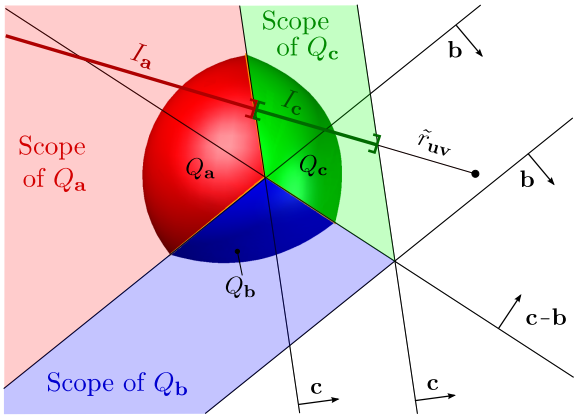


Fig. 2.11: Sketch of the scope of the three quadric surface patches of the polar dual of the δ-offset of a triangle and an intersecting ray with the respective intervals.

Our question is now whether there is a $\lambda_0 \in \mathbb{D}$ with $\mathbf{x}(\lambda_0) \in (\mathcal{A} \oplus S_\delta)^*$. Since the surface of $(\mathcal{A} \oplus S_\delta)^*$ is composed of three surface patches $Q_\mathbf{a}$, $Q_\mathbf{b}$, and $Q_\mathbf{c}$ and since we intersect with a ray or a straight line, there must be one or two intersections of the ray or the straight line with the quadric patches iff there exists a $\mathbf{x}(\lambda_0) \in (\mathcal{A} \oplus S_\delta)^*$. It is sufficient to test whether

$$\exists \lambda_0 \in \mathbb{D} : \quad \mathbf{x}(\lambda_0) \in Q_\mathbf{a} \ \vee \ \mathbf{x}(\lambda_0) \in Q_\mathbf{b} \ \vee \ \mathbf{x}(\lambda_0) \in Q_\mathbf{c} \ .$$

Each surface patch is a subset of a quadric (refer to the Equations 2.18, 2.19 and 2.20). We explained already that the surface patches are restricted by planes. These planes define the scope of a surface patch, which are

$$\begin{aligned}
\text{Scope of } Q_\mathbf{a} &:= \{ \ \mathbf{x} \in \mathbb{R}^3 \mid \langle \mathbf{b}, \mathbf{x} \rangle \leq 0 \wedge \langle \mathbf{c}, \mathbf{x} \rangle \leq 0 \} \\
\text{Scope of } Q_\mathbf{b} &:= \{ \ \mathbf{x} \in \mathbb{R}^3 \mid \langle \mathbf{b}, \mathbf{x} \rangle \geq 0 \wedge \langle \mathbf{c} - \mathbf{b}, \mathbf{x} \rangle \leq 0 \wedge \langle \mathbf{b}, \mathbf{x} \rangle \leq 1 \} \\
\text{Scope of } Q_\mathbf{c} &:= \{ \ \mathbf{x} \in \mathbb{R}^3 \mid \langle \mathbf{c}, \mathbf{x} \rangle \geq 0 \wedge \langle \mathbf{c} - \mathbf{b}, \mathbf{x} \rangle \geq 0 \wedge \langle \mathbf{c}, \mathbf{x} \rangle \leq 1 \} \ .
\end{aligned}$$

In order to test whether there is for example a $\lambda_0 \in \mathbb{D}$ with $\mathbf{x}(\lambda_0) \in Q_\mathbf{a}$, we test whether

$$\exists \lambda_0 \in I_\mathbf{a} : \ \mathbf{x}(\lambda_0) \in s(\delta, \mathbf{a})^P \ ,$$

where $I_\mathbf{a} \subset \mathbb{D}$ is the interval in λ where $\mathbf{x}(\lambda)$ is inside the scope of $Q_\mathbf{a}$.

We determine the intervals $I_\mathbf{a}$, $I_\mathbf{b}$ and $I_\mathbf{c}$ by computing and sorting the λ-values for which the five above planes intersect the ray or the straight line (Line 9). Figure 2.11 shows a sketch of the scope of the three quadric surface patches and the intersection intervals with a ray.

Finally, we have to test whether there is an intersection of the ray or the straight line with the quadric patch in the respective interval. For example for the patch $Q_\mathbf{a}$ we have to test whether there is a solution of the quadric equation

$$q_\mathbf{a}(\lambda) = \delta^2 \langle \mathbf{x}(\lambda), \mathbf{x}(\lambda) \rangle - (1 - \langle \mathbf{a}, \mathbf{x}(\lambda) \rangle)^2 \stackrel{!}{=} 0$$

in the interval $I_{\mathbf{a}}$ (Line 12 and 13). In summary, our question is now whether

$$(\exists \lambda_0 \in I_{\mathbf{a}} : \ q_{\mathbf{a}}(\lambda_0) = 0) \quad \vee \quad (\exists \lambda_0 \in I_{\mathbf{b}} : \ q_{\mathbf{b}}(\lambda_0) = 0) \quad \vee \quad (\exists \lambda_0 \in I_{\mathbf{c}} : \ q_{\mathbf{c}}(\lambda_0) = 0) \ .$$

In the case there exists such a λ we know that there is a separating plane and can have an early out (Line 14).

2.3. Benchmarks

We have implemented our approach and tested it with a wide range of different triangle pairs. We generated in summary 76 different types of test sets, each containing 10,000,000 triangle pairs. Using these test sets, we benchmarked all presented triangle-triangle tolerance tests as well as some triangle tests from literature. For our benchmark we used a consumer notebook with an Intel i7-4800MQ processor with a 2.70 GHz quad-core CPU.

2.3.1. Test Sets

The triangle pairs in our test sets are triangle pairs as they occur in application cases of tolerance tests between complex objects. Thus, we do not use pure randomly generated triangle pairs. This is important, because pure random triangles lie arbitrary in space and have arbitrary shapes. But the triangle pairs that has to be tested in the application case are actually not random. For example, triangles of complex object lie usually on a smooth surface. Thus, for tolerance tests, there is usually a larger portion on triangle pairs with a relatively small or medium angle between the triangle normals. Further, the portion on nearly degenerated triangles is usually relatively small. For this reason, we ran different types of application cases for tolerance calculations between complex objects and collect triangles pairs from these application cases.

Our main application is to compute the set of all tolerance violating triangles between two complex objects. We will denote this in the following as the calculation of *set-results*. Beside this, we consider the application case to compute all pairs of tolerance violating triangles between two complex objects. We will denote this in the following as the calculation of *pair-results*.

- **Set-Results:** To create one test set we ran the algorithm to calculate all tolerance violating triangles between two complex objects for a fixed tolerance value and for a sequence of random transformations that are applied on the dynamic object. The broad phase of the algorithm identifies candidate triangle pairs that might eventually be tolerance violating. In the case both of the triangles are not yet marked as tolerance violating, the triangle pair is tested on tolerance violation within the narrow phase. We have output the first 10,000,000 triangle pairs that are tested within the narrow phase to a file. The triangle pairs in this file represent one test set. Within our benchmarks we read the test set and test all 10,000,000 triangle pairs with the respective tolerance value δ on tolerance violation.

- **Pair-Results:** The process to create one test set is similar to the process to create a test set for pair-results, except for one issue: Other than for set-results no triangle pair is suspended because it is already marked as tolerance violating. Therefore the test sets for pair-results contain a much higher percentage of tolerance violating triangle pairs.

In order to create our test sets we use three different scenarios. Each scenario consists of a static object and a dynamic object. The scenarios and models are:

- **Scenario Engine:** The scenario consists of the front part of a car's bodyshell and a complete engine. For details and pictures refer to the Appendix A.1. Both models contain very different sized triangles. The different sized triangles make the scenario especially interesting for benchmarking the triangle-triangle tolerance tests on real-life test data.

- **Scenario Bunny:** The scenario consists of two instances of the Stanford Bunny. For details and pictures refer to the Appendix A.3. The Stanford Bunny consists of similar sized triangles. Lots of the triangles are nearly rectangular. The scenario is especially interesting as it uses well known academic models and as the triangles are well shaped.

- **Scenario Buddha:** The scenario consists of two instances of the Stanford's happy buddha model. For details and pictures refer to the Appendix A.4. In contrast to the bunny, the buddha consists of very small triangles. The triangles are different shaped, such that there are also acute and obtuse angles. The scenario is especially interesting as it uses well known academic models with additionally high challenges on the stability of the triangle tests.

The tolerance value that is used for the models of the *Engine* scenario in practical applications is about $\delta = 15$. For our benchmarks we consider a larger range of tolerance values with $\delta \in \{2.5, 5, 10, 15, 20, 25, 30\}$. For the academic models we convert these tolerance values in relation to the extension of the models. So we consider for the *Bunny* scenario the tolerance values $\delta \in \{0.0005, 0.001, 0.0015, 0.002, 0.0025, 0.003\}$ and for the *Buddha* scenario the tolerance values $\delta \in \{0.00005, 0.0002, 0.0004, 0.0006, 0.0008, 0.001\}$.

In all scenarios for all tolerance values we apply two types of transformations on the respective dynamic object, which are:

- **NoColTolVerl**: A set of transformations where the dynamic object and the static object have in every position no collisions but tolerance violations for a fixed tolerance value.

- **ColTolVerl**: A set of transformations where the dynamic object and the static object have in every position about 1,000 tolerance violating triangles for a fixed tolerance value.

For details on both transformation sets refer to the respective subsections in Appendix A.1, A.3, and A.4.

Summarized we get the following 76 test sets:

$$\left\{\begin{matrix}\text{Set-Results}\\ \text{Pair-Results}\end{matrix}\right\} \left\{\begin{matrix}\text{Engine}\\ \text{Bunny}\\ \text{Buddha}\end{matrix}\right\} \left\{\begin{matrix}\text{NoColTolVerl}\\ \text{ColTolVerl}\end{matrix}\right\} \left\{\text{varying } \delta\right\} . \tag{2.35}$$

2.3.2. Benchmark Process

We ran several triangle tests for each test set and output the total running time in seconds, that is required to test all 10,000,000 triangle pairs. Prior to this calculation, we normalize the scenario. This means that we scale all triangle pairs by the factor $1/\delta$ and adapt the respective tolerance value to be $\delta = 1$. This step is especially important, because the bunny model and especially the buddha model consists of very small triangles with average edge length 0.0015 and 0.0004. Scaling the scenario makes the results comparable as we calculate with floating point numbers in the same order of magnitude.

Within our benchmarks we consider the following triangle-triangle tolerance tests:

- **SepPlane:** Our implementation of the pure separating plane test (Section 2.1.4)
- **SepPlaneVV:** Our implementation of the pure separating plane test (Section 2.1.4), but with additional vertex-vertex distance test.
- **Dual:** Our dual approach as given in Algorithm 1 (Section 2.2.6).
- **Dual-P:** Our dual approach without the redundant separating plane tests parallel to the triangle faces, thus according to Algorithm 1, but without Line 1-4 (Section 2.2.6).
- **Combined:** Our implementation of the combined test (Section 2.1.5)
- **FeatDist:** Our implementation of the feature distance test (Section 2.1.3)

We compare our implementations with the following triangle tests from related work:

- **Erbes:** The implementation of the combined approach by Erbes [3].
- **PQP:** The implementation of the distance test of the PQP library [27].
- **Möller:** The triangle-triangle intersection test by Möller [38].

2.3.3. Benchmarks for Set-Results

Within this section we present the benchmark results for the 38 test sets with set-results. The following diagrams only show the results for the *NoColTolVerl* transformations, as the results for the *ColTolVerl* transformations are very similar. All our results are given completely in the Appendix B.1.

In the following we will use our dual approach *Dual* as represent of the dual approach, because in all test sets for set-results *Dual* is faster than *Dual-P*. The reason therefor

is, that the planes that contain the triangles are often separating planes. Thus, *Dual* has much earlier an early out than *Dual-P* and is therefore faster. Refer therefor also to the head maps in Appendix B.1.

Comparing the Dual Approach with an Equivalent Primal Approach: First of all we want to verify that the idea of mapping the triangle-triangle tolerance test to dual space works successfully. Therefore we developed an equivalent primal approach with the same types of early outs in the same order. The kind and order of early outs is important to verify whether there is really a benefit by calculating in dual space. The equivalent primal approach is the triangle-triangle tolerance test *SepPlaneVV*. It bases on the implementation of the pure separating plane test (refer to Section 2.1.4), but with additional vertex-vertex distance test. At first it tests, like *Dual* the planes that contain the triangles on separation. *SepPlaneVV* performs this step in primal space, *Dual* in dual space. Next the vertex-vertex distances and separating planes perpendicular to the vertex-vertex connections are tested. This step is performed identically by *SepPlaneVV* and *Dual* in primal space. Finally all separating plane tests with planes parallel to an triangle edge are tested. This step is done by *SepPlaneVV* in primal space and by *Dual* in dual space. The result of our benchmark is shown in Figure 2.12. It is obvious that the *Dual* approach is always faster and we conclude that it is worth to map the problem into dual space.

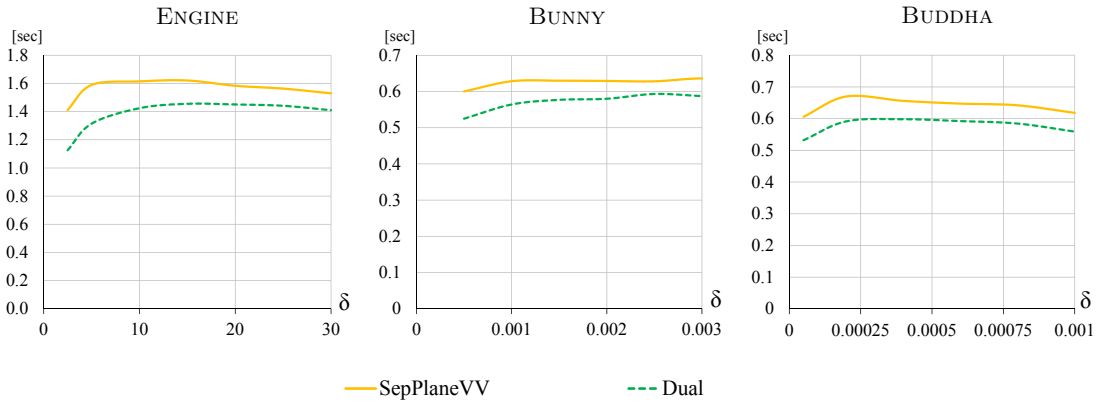


Fig. 2.12: Comparison of the dual approach with an equivalent primal approach for set-results and the *NoColTolVerl* transformations.

Comparing the Dual Approach with our other Approaches: The equivalent primal approach *SepPlaneVV* is not necessary our fastest primal approach. Thus, we compare *Dual* with our other implementations of primal triangle-triangle tolerance tests, which were presented in Section 2.1.4, 2.1.5, and 2.1.3. The diagrams in Figure 2.13 show the comparison of *Dual* with *SepPlane*, *SepPlaneVV*, and *Combined*. The *FeatDist* approach is not diagrammed because it is not competitive with our other implementations in the application case of set-results. *FeatDist* is always a factor 2.5 – 9.8 slower than the *Dual* approach (refer to Appendix B.1). Our result shows that using separating plane tests is principally a better strategy than using feature distance tests. However,

using some feature distance tests additionally to the separating planes tests is the best strategy, as the performance of *SepPlaneVV* and more over of *Combined* shows. In summary, *Combined* is our best primal approach – but *Dual* is in all test sets the fastest approach.

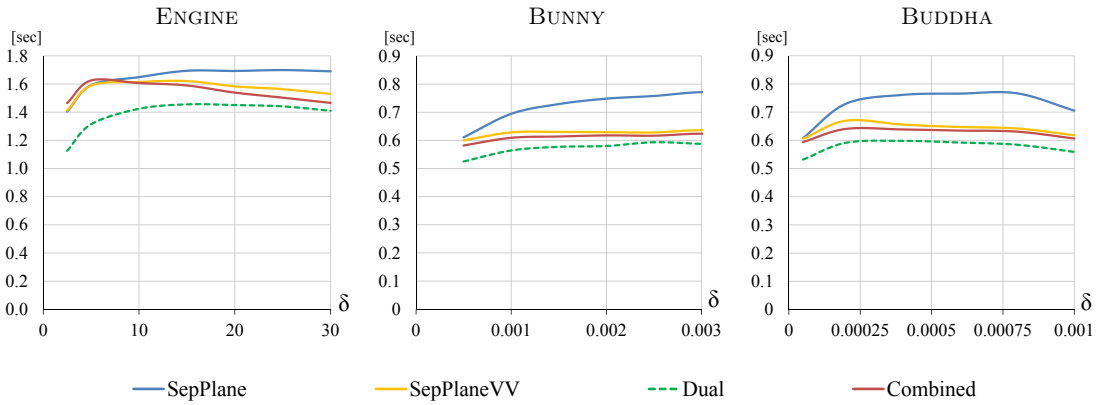


Fig. 2.13: Comparison of the dual approach with our other primal implementations for set-results and the *NoColTolVerl* transformations.

Comparing the Dual Approach with Previous Work: At last we compared the *Dual* approach with approaches from previous work. The diagrams in Figure 2.14 show the comparison of *Dual* with *Erbes* and *Möller*. The *PQP* approach is not diagrammed because it is not competitive with the other implementations. *PQP* is always a factor 2.6 – 9.5 slower than the *Dual* approach (refer to Appendix B.1). This is caused by the fact that it is a distance and not a tolerance test. One can see that the *Dual* approach is always faster than the solution of *Erbes*. Moreover it constantly approaches by a factor ≤ 4.5 (*Engine*) and ≤ 2 (*Bunny*, *Buddha*) the intersection test of *Möller* which is independent of δ and geometrically much easier.

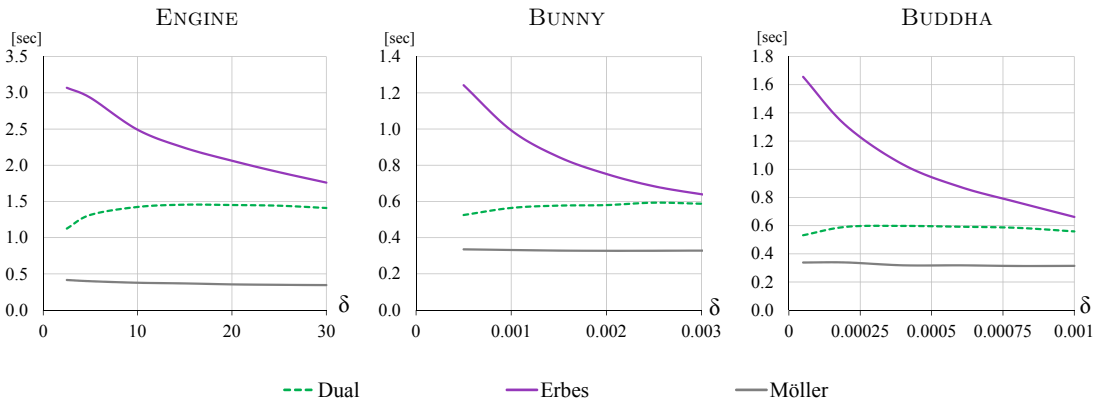


Fig. 2.14: Comparison of the dual approach with previous work for set-results and the *NoColTolVerl* transformations.

2.3.4. Benchmarks for Pair-Results

Within this section we discuss the benchmark results for the 38 test sets with pair-results. To interpret the results it is important to keep in mind that for pair-results no triangle pairs are suspended for the calculation, because they were already marked as tolerance violating. The consequence is that the test sets for pair-results have a much higher percentage on tolerance violating triangle pairs. Tables that display the percentage on tolerance violating triangle pairs for all our test sets are given in the Appendix B.3. Different percentages on tolerance violations require different kinds and orders of early outs. Nevertheless, we are interested on how our algorithms – which were developed for the application case of set-results – perform for pair-results.

The diagrams in Figure 2.15 summarize the most important results for pair-results. All our results are given completely in the Appendix B.1.

We display both dual approaches *Dual* and *Dual-P*. For small tolerance values *Dual* is faster than *Dual-P*. For large tolerance values it is vice versa. In contrast to set-results, the dual approaches are not always faster than all other approaches. But, however, there is no approach always the fastest. Sometimes *Combined* has a good performance, many times *Erbes*. The dual approaches are somehow averaging between *Combined* and *Erbes*. The *FeatDist* approach is competitive with the other approaches in the case of pair-results. In the *Engine* scenario it achieves even good results. In contrast, the *SepPlane* approach performs worse for pair-results (and is therefore not diagrammed). In summary, one can see that for pair-results there is no tolerance test which is clearly the best in all test sets, especially for the real-world data.

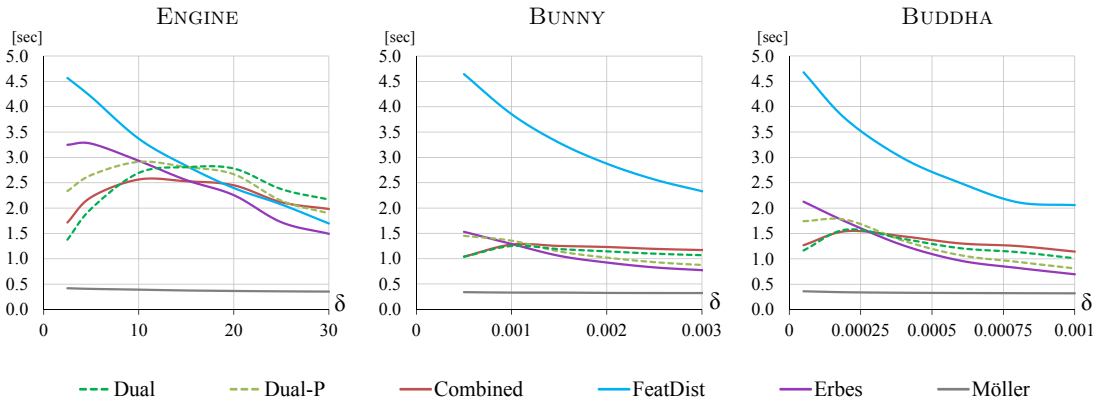


Fig. 2.15: Comparison of both dual approaches with other primal approach for pair-results and the *NoColTolVerl* transformations.

This mainly depends on the strong increasing percentage of tolerance violating triangle pairs. Every percentage needs a special treatment in the sense of specialized early outs. We have visualized the performance difference over the percentage of tolerance violating triangle pairs per test set in Figure 2.16. For every test set and every approach a point is drawn. The x-axis represents the percentage of tolerance violating triangle pairs per test set and the y-axis how much slower the approach is, compared to the fastest approach. We have not displayed the approaches that are never the fastest and

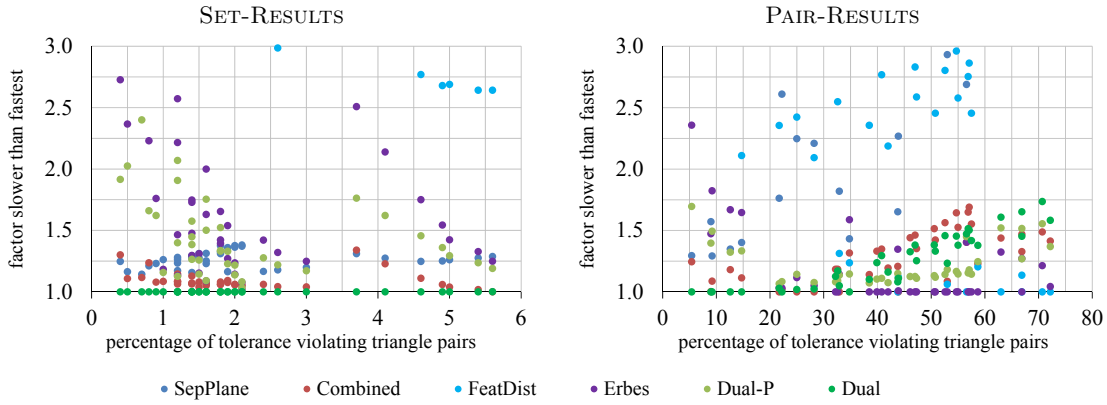


Fig. 2.16: Performance difference compared with the respectively fastest approach over the percentage of tolerance violating triangle pairs per test set.

we have not displayed the cases when an approach is slower than the factor 3. We can state that up to a percentage of 20% *Dual* is still the fastest. For large percentages *Dual-P* has a higher performance than *Dual*. For percentages $\leq 40\%$ the algorithm of *Erbes* is the best choice. But although *Dual-P* has much fewer early outs in the case of tolerance violations than Erbes, it is always competitive, even in the case of many tolerance violating triangle pairs.

2.3.5. Correctness

On the whole, we achieve in all our test sets the same results, thus, obtain the same tolerance violating triangle pairs. Of course there are small differences in the results due to floating point errors. Because of that we will analyse the correctness in the following. We have compared for each test set the result for every triangle pairs that is calculated with the results calculated by *SepPlane*. We use *SepPlane* for these verifications, since it is a numerically very stable implementation. All approaches achieve nearly no false results. If we compare, for example, the results computed with our other primal approaches, there are in average 0.00002% mismatches. The results computed by *PQP* have in average 0.00011% mismatches. *Dual* and by *Dual-P* are also numerically unproblematic. In average there are 0.0038% mismatches compared with *SepPlane*. For details, refer to the histogram in Figure 2.17.

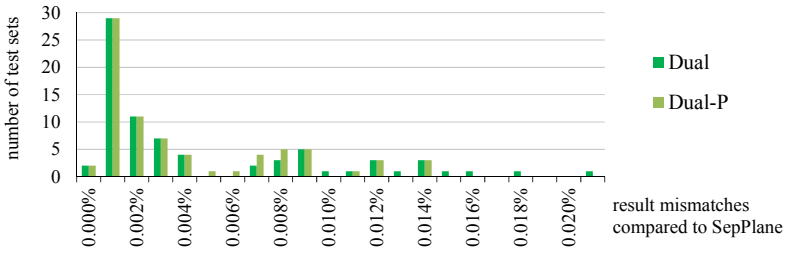


Fig. 2.17: Percentage on result mismatches per test set.

2.4. Tolerance Tests for General Geometric Primitives

All ideas of the presented triangle-triangle tolerance tests can be used to create tolerance tests for other types of convex primitives. For elementary primitives, like points and edges, the respective tolerance tests are already given as sub-parts of the triangle-triangle tolerance tests. For more complex primitives the tolerance tests can be composed out of parts of the triangle-triangle tolerance tests or extended by iterating over some more features of the primitives.

None-Volumetric Primitives

Considering other none-volumetric primitives, thus polygons, there are just some additional vertex and some additional edge features per primitive that must be considered. All the above introduced tolerance tests can be easily extended by considering the additional features and feature pairs. For example, if the primitives are rectangles, there is one more vertex feature and one more edge feature per primitive.

Based on the properties of the respective primitive, there are sometimes advantages that can be used. Considering again rectangles, some calculations will be redundant and can be saved during the calculation because of the right angles and the parallel edges of a rectangle. For example, if the separating plane approach is applied, there are actually 16 axes that are parallel to a pairs of edges – but in fact there are only 4 different axes parallel to all pairs of edges. In the dual approach one can profit because it is known that the quadric patches are bounded by two orthogonal planes instead of 4 arbitrary halfplanes. Tolerance tests for rectangles are already discussed in literature by Larsen et al. [28, 29] and by Erbes [3] where they use the swept sphere volume of a rectangle (RSS) as bounding volumes.

Volumetric Primitives

For volumetric primitives the tolerance tests will be more complex. In contrast to non-volumetric primitives, where we just have additional vertex-features and additional edge-features, volumetric primitives in addition have at least 3 face features and one new kind of feature, the volume feature that represents the interior of the primitive.

Tolerance tests that are based on feature distances additionally have to check whether a feature of primitive $\mathcal{A}$ is inside of the other primitive $\mathcal{B}$ and vice versa. This is only the case if one primitive is completely contained in the interior of the other primitive or when an edge of one primitive intersects a face or edge of the other primitive. All in all, except for the inclusion test, everything can be deduced from the already known feature-pair checks. Tolerance tests for volumetric primitives that are based on the presented separating plane approach can be handled completely the same as for polygons. The dual approach becomes more complicate. The δ-offset of the volumetric primitive is composed of several quadric surface patches (as many as the primitive has vertices), which are not anymore connected in only two poles. The dual of the second primitive is as complex as for triangles. It is still the intersection of several halfspaces (as many as the primitive has vertices). The ideas of Section 2.2 can all be transmitted or extended easily to convex polyhedra. But finally the intersection tests of the elementary primitives (points, lines and planes) with the complex δ-offset of the volumetric primitive is more effort.

3. Broad Phase for the Calculation of "All Tolerance Violating Primitives"

In this chapter we will introduce our approaches to determine all tolerance violating primitives between two complex objects. It is very common to divide proximity query approaches into two phases: The broad and the narrow phase. Within the broad phase, pairs of candidate primitives are obtained. In our case we obtain the pairs of primitives, one from each of the complex objects, which might be tolerance violating. Within the narrow phase the obtained pairs of candidate primitives are tested. In our case they are tested on tolerance violation.

We presented in the previous Chapter 2 how to perform fast triangle-triangle tolerance tests in detail. As already mentioned there, fast tolerance testes between two arbitrary polyhedral primitives can be performed similarly. In this chapter we will complete our approach by presenting the broad phase for the calculation of all tolerance violating primitives. Thus, we will introduce our broad phase data structures and algorithms. Later in Chapter 4 some further implementation details are discussed and our benchmarks are presented.

At the beginning of this chapter one general issue should be noted: The approach is not restricted to a special type of primitives like for example triangles. We will discuss our approaches in a general way in order to determine all tolerance violating primitives. Nevertheless the reader can always envision triangles standing for primitives and further, most examples are explained by using triangles.

3.1. Examination of Suitable Broad Phase Data Structures

An obvious approach to determine all tolerance violating primitives between two complex objects $\mathfrak{O}_A$ and $\mathfrak{O}_B$ without a broad phase is to perform a tolerance test of each primitive in $\mathfrak{O}_A$ with all primitives in $\mathfrak{O}_B$. Indeed, such an approach is impracticable, as the running time is quadratic in the number of primitives – or more precisely, it is $O(nm)$ with n the number of primitives in $\mathfrak{O}_A$ and m the number of primitives in $\mathfrak{O}_B$. Concerning this issue, broad phase data structures and algorithms are used to reduce the number of primitives that has to be tested. We will reflect some common broad phase data structures and algorithms in the following and evaluate them with regards to our problem statement.

3.1.1. Foundations

There are many possibilities to design an algorithm and to choose a data structure for the broad phase of proximity queries. A very good summary is given by Ericson in the book *Real-Time Collision Detection* [5]. Surveys on collision detection, proximity queries and possible data structures are for example given by Lin and Gottschalk [33], Jiménez et al. [21], by Figueiredo et al. [7] and by Lin and Manocha [34]. In the following a selection of the most promising data structures for our special task are introduced and common knowledge is refreshed and partially transferred to tolerance queries.

Bounding Volume Hierarchies (BVHs). One of the most popular data structures are bounding volume hierarchies (hereinafter referred to as BVHs). The idea of a bounding volume is to enclose a complex geometry in simple geometry. For example, a triangle can be enclosed within a sphere. The central property for bounding volumes is that in the case that two bounding volumes are not intersecting, the enclosed geometry cannot intersect. Of course this is also true for the tolerance query: If the distance between two bounding volumes is greater than a tolerance value δ, the distance between the enclosed geometry is surely greater than the tolerance value δ, too.

Enclosing each primitive of our complex objects in a bounding volume and pairwise testing would probably reduce the calculation time slightly, because many tolerance tests between primitives can be replaced by more simple bounding volume tests. But it would not reduce the calculation complexity. In order to reduce the calculation complexity, a hierarchical structure of bounding volumes is used – the BVH. A BVH is a tree of bounding volumes. Each node of the tree represents a bounding volume, which encloses the primitives of its child nodes. Thus, the root of the tree encloses all the primitives and a leaf node encloses usually just one primitive or only a few primitives. Figure 3.1 shows the structure of a BVH, for sake of illustration on a 2d-example. There, a BVH of axis aligned bounding boxes is built for a polygon which is composed out of seven segments.

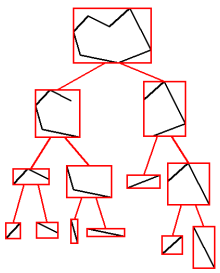


Fig. 3.1: BVH of a 2d-polygon as hierarchy of axis aligned bounding boxes.

There are various variants of BVHs. They vary mainly in the type of bounding volumes (spheres or axis aligned boxes are examples), in the degree (typical is the degree of two, thus binary-trees), in the way of construction and partitioning. The great advantage of BVHs comes with the query. In order to test, for example, whether a query primitive intersects a complex object of n primitives, the bounding volume of the query primitive and the BVH of the object are considered. The query starts at the root of the BVH. Only in the case the primitive's bounding volume intersects the bounding volume of a node in the BVH, the query is passed to the child nodes. And only in the case the query reaches a leaf node, a primitive-primitive test is performed. Provided the BVH is given as a balanced binary tree and provided the query primitive intersects only a constant number of bounding volumes in each level, such an intersection test can be done in $O(\log n)$ instead of $O(n)$ time in the naive approach.

Uniform Grids. BVHs are object related data structures. Beside object related data structures, also space partitioning data structures are common. Instead of partitioning the object into regions, in order to prune away negligible parts of the object (like it is done by traversing a BVH), the space is partitioned into regions and negligible parts of the space are pruned within a query. The elementary space partitioning data structure is the uniform grid. The uniform grid partitions the space into equal sized cells. Figure 3.2 shows a 2d-example of a polygon inside a uniform grid.

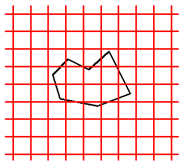


Fig. 3.2: Uniform grid with a 2d-polygon. Each segment is referenced by several cells.

One typical possibility to create a uniform grid for a complex object is to reference each primitive of the complex object by the cells that are intersected by the primitive. So each grid cell references to all its intersecting primitives. It is a notable difference compared to BVHs, where each primitive is referenced exactly once by a bounding volume of the leaf level. Later on, this issue will be discuss more detailed. A grid cell that references at least one primitive of the complex object is called *occupied*.

The central property for a uniform grid is that two primitives can only intersect in case that there exists a grid cell which references both primitives. In order to generalize this property for tolerance violations, Equation 1.2 is considered: A and B are tolerance violating $\Leftrightarrow \ (A \oplus S_\delta) \cap B \neq \emptyset$. Thus, only in the case that there exists a grid cell that refers the primitive B and the δ-offset $A \oplus S_\delta$ of a second primitive A, the two primitives A and B can be tolerance violating.

A great advantage of uniform grids is the simplicity of the data structure and the simplicity of a request. In order to test, for example, whether a query primitive intersects a complex object of n primitives, the query primitive has to be located in the uniform grid of the complex object. To be located means that the indices of the grid cells that intersect the query primitive have to be determined. One single point is easily located in a grid by simple integer divisions of the point's coordinates, which provides the cell indices. A primitive can be located, for example, by locating the corners of the primitive's axis aligned bounding box and performing cell-primitive intersection tests for all cells inbetween. The query primitive is tested on intersection with the primitives of the cells, where the query primitive has been located. For more details on this issue we refer to Section 3.3. Provided the access on a grid cell by its index is done in $O(1)$[1], a query primitive is located in only a constant number of cells and a cell references only a constant number of primitives, such an intersection test can be done in $O(1)$ instead of $O(n)$ time in the naive approach.

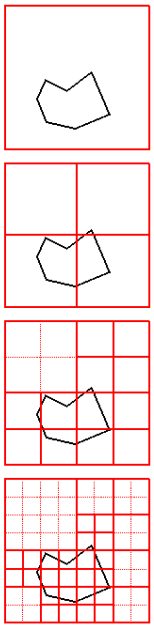


Fig. 3.3: 2d-polygon inside of a quadtree with 4 subdivisions respectively with 4 levels.

Multi-Layer-Grids and Octrees A uniform grid is a none-hierarchical space partitioning data structure. The multi-layer-grid (or short multi-grid) is the hierarchical extension of a uniform grid. As the name already says, it consists of some grid layers. From the top layer until the base layer, the cell width of each layer becomes smaller. A special case of a multi-grid is an octree. The top layer consists of one cell. Each following layer is build by splitting each cell into its eight octants. The 2d counterpart of an octree is a quadtree, which is constructed by splitting a cell in four quadrants. Figure 3.3 shows a 2d-example of a polygon inside a quadtree.

[1]This is possible, as we will explain in Section 4.2

Compared to uniform grids, multi-grids and octrees have two main advantages: First, they avoid the problem of finding a proper cell size and second, the hierarchical structure prunes away negligible parts during the query. The elementary publication is given by Samet [47].

3.1.2. Some Reflections on BVHs and Uniform Grids

Because BVHs are probably most popular for collision queries and because uniform grids are fundamental for this work, we will have a close look on these two data structures in the following.

As explained above, the effort for one query primitive to request a BVH is done in $O(\log n)$ and requesting a grid is done in $O(1)$.[2] By this consideration one could assume that it is obvious that grid based algorithms would perform best. But in fact, the decision for the most suitable data structure and the respective algorithms is not that simple. First, there are significant constants that have to be considered and second, we are not requesting with one single primitive – our request is for another complex object, which is composed out of multiple primitives. Hence, data structures for the static and the dynamic object as well as an appropriate query algorithm is required for our task.

Consider a BVH Approach

Using two BVHs is usually the best choice for collision tests if one just wants to know whether two objects are colliding. The algorithm that tests two bounding volume hierarchies against each other is called a BVH traversal. The BVH traversal gives a very fast answer in the case that the two objects are separated. Also in the case that the objects are not separated, they focus very fast on the most relevant bounding volumes and lead the query to the first pair of colliding primitives. Given two balanced bounding volume hierarchies, each of n primitives, the worst case running time of a bounding volume traversal is $O(n^2)$. The running time that is expectable in real life cases is usually much better and commonly supposed as logarithmic. In [54] an expected running time analysis is given for bounding volume hierarchies of axis aligned bounding boxes by Weller et al. They show that the average running time can be estimated based on the overlap of the root bounding volumes and on the diminishing factor of the hierarchies, which is a measure of the downsizing of a parent bounding volume to its child bounding volumes.

Very early Weghorst et al. [53] introduced the so called *cost function* for ray tracing and [13, 24, 15] enhanced it for collision queries. The running time for one bounding volume traversal is given as

$$T = N_v C_v + N_p C_p + N_u C_u + C_0,$$

[2] Under the assumptions described in Section 3.1.1: The BVH is given as a balanced binary tree and the query primitive intersects only a constant number or bounding volumes in each level. The query primitive is located only in a constant number of grid cells and a grid cell refers only a constant number of primitives.

where N_v is the number of all necessary bounding volume tests and C_v the effort for one bounding volume test, N_p is the number of all necessary primitive tests and C_p the effort for one primitive test, further N_u is the number and C_u the cost of updating the bounding volumes (for example, if a transformation has to be considered for one of the BVHs, C_u is the cost to apply the transformation to a bounding volume) and C_0 sums up the effort for operations that have to be done only once per traversal (for example initializations). Thus, the running time of a BVH traversal depends on several factors – unfortunately they are contradictory to optimize. For example tight fitting bounding volumes would reduce the number of bounding volume tests N_v and the number of primitive tests N_p (which reduces the calculation time) but they would also increase the cost of a single bounding volume test C_v (which increases the calculation time).

Consider a Grid Approach

In comparison, a simple grid approach locates all the primitives of the dynamic object in the uniform grid of the static object at query time. Provided that every primitive is located in a constant number of grid cells and every cell contains a constant number of primitives, each dynamic primitive is tested with a constant number of static primitives. The effort of such a collision test with n primitives, which are localized each in $O(1)$, has the complexity of $O(n)$.

We can give a similar cost function for the grid approach by

$$T = N_c C_c + N_p C_p + N_u C_u + C_0,$$

whereas N_c is the number of cell queries and C_c the effort of one cell query. The other parameters are identical to the ones of the cost function for BVH traversals. Again, the parameters influence each other. For example, the number of cell queries N_c depends on the cell width. Increasing the cell width implicates a smaller N_c. Thereby the number of primitive tests N_p is increased, because each cell covers a greater region. A special characteristic is that the number of primitive tests N_p is reduced until a certain degree by reducing the cell width of the grid and thus increasing the number of cell tests N_c. This is because the selection of possibly intersecting primitives is getting tighter. But after a certain degree, if the cell width becomes too small, N_c and N_p are both increasing. This is because primitives are referenced by an increasing number of cells and therefore considered multiple times for collision tests. More about this influence is given in 4.4.

3.1.3. Evaluation of Data Structures for the Problem Statement

Although BVH traversals are the most common approach for collision queries, for more complicate proximity queries, like the calculation of the set of tolerance violating triangles (refer to Section 1.2), it is not obvious which data structure might be the best choice.

The required data structures and algorithms should be well suited for calculations with large and complete result outputs. Complete means here that we do not stop as soon as we find the first witness pair but compute the complete set of all tolerance violating primitives. Therefore, for each primitive it has to be decided whether it is tolerance

violating or not. The size of the result dependents on the tolerance value δ. Therefore it is important that the influence of a large tolerance value δ will not result in an exploding calculation time – especially when the objects are in close proximity. For example within a BVH traversal, the number of bounding volume pairs that have to be tested becomes large. Using the above introduced notations, N_v becomes very large. It is the same for a uniform grid approach. Many grid cells are affected, thus N_c becomes large. At this point, we have to note that usually $C_c \ll C_v$, for example testing two orientated bounding boxes on tolerance violation is much more expensive than a grid cell location.

On the other hand, even when the objects are in close proximity, in practice there are still regions of both objects which are not in close proximity and which are definitely not participating on the result. Such regions should optimally be pruned away in order to concentrate the calculation power on the relevant regions. Hierarchical data structures, like BVHs and multi-grids or octrees, are perfectly able to prune away negligible regions.

Further, our intention is to parallelize the calculation of all tolerance violating primitives. The parallelization of a hierarchy traversal is not trivial (refer e.g. Tang et al. [50] and Kim et al. [23]). The parallelization of a uniform grid approach is more natural, due to the existence of many independent tasks (refer e.g. Papst et al. [42]). We desire a natural partition to sub-tasks with small additional administrative costs. This is especially important when we plan to parallelize relatively small calculation tasks for real time calculation and therefore we cannot afford high additional administrative costs.

Summarized Requirements

In summary, the requirement for our data structures and algorithms are the following:

- Suitable for calculations with large and complete result outputs
- Fast elimination of negligible regions
- Possibilities for parallelization without much additional administrative costs

3.2. Introduction to our Data Structure and Algorithms

Reconsidering the above introduced data structures, each of them has advantages and disadvantages for our task. None of them fulfills all the above three requirements perfectly. Thus, we are going to use the advantageous features of the data structures and create our data structure as described in the following.

Our data structure is a combination of some uniform grids with one arbitrary hierarchical data structure (for example a BVH or a multi-grid). A uniform grid is more suitable for calculations with large and complete results and for parallelization than a hierarchical date structure. But the disadvantage of a uniform grid is the lack of focussing on relevant regions and therefore many grid queries in regions that do not participate to the result of all tolerance violating primitives. For this reason our data structure consists of several uniform grids, which are organized under the hierarchical

Algorithm 4: Principal Algorithm

Data: δ // tolerance value
$\mathsf{HG}(\mathfrak{D})$ // data structure of the dynamic object
$\mathsf{HG}(\mathfrak{S})$ // data structure of the static object
$\mathbf{T}$ // transformation (of the dynamic object)

Result: T_δ // the set of all tolerance violating primitives

```
1  H_D ← get the hierarchical component of HG(𝔇)
2  H_S ← get the hierarchical component of HG(𝔖)
3  TH_D ← apply T on H_D
4  pairs ← get the intersecting / tolerance violating leafs between TH_D and H_S
5  foreach pair (first, second) in pairs do
6      G_D ← get the uniform grid in first
7      G_S ← get the uniform grid in second
8      foreach occupied cell C in G_D in parallel do
9          TC ← apply T on C
10         candidatCells ← obtain cells in G_S, which contain primitives that are
           possibly tolerance violating with the primitives in TC
11         foreach K in candidatCells do
12             foreach primitive A in C and foreach primitive B in K do
13                 TA ← apply T on A
14                 if TA and B are tolerance violating then
15                     mark A and B in T_δ as tolerance violating
```

data structure. This hierarchical component of the data structure focuses on the relevant regions and prunes away negligible regions. In contrast to typical hierarchical data structures, the hierarchy is very flat. The depth, for example of a BVH, is usually such that each leaf of the hierarchy contains just one primitive or just a few primitives. In contrast, our hierarchy has just a few levels and the regions of the leafs cover many primitives. The primitives, which are covered by a leaf of our hierarchy, are not stored in the leaf directly. Each leaf is associated with one uniform grid and the primitives are stored inside the respective uniform grids. Figure 3.4 shows a sketch of this combined data structure by using the simple 2d-example of the previous section. We use this data structure to store the static and to store the dynamic object.

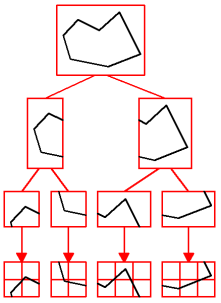


Fig. 3.4: 2d-polygon stored 4 disjoint uniform grids, which are stored in a 3-level BVH of axis aligned bounding boxes.

In order to calculate all tolerance violating primitives for the static and the dynamic object, in a *first step* the relevant regions are determined by using our flat hierarchy.

For all pairs of tolerance violating leafs the associated pairs of uniform grids are considered in a *second step*. For each pair of uniform grids, we process all occupied cells of the dynamic grid to obtain candidate primitives in the static object that are possibly tolerance violating with the dynamic cell and therefore with the primitives referenced in the dynamic cell.

Finally, in a *third step*, we perform the primitive tests for the primitives of a dynamic grid cell and the obtained candidate primitives.

We parallelize the calculation of the second and the third step such that one thread is responsible for one cell in the dynamic grid.

The principle of this algorithm is given more formally in Algorithm 4. The next sections of this chapter describe our data structure and the algorithm more detailed. Section 3.3 explains how the uniform grids are constructed and how the grid query works. Section 3.4 describes how we organize several uniform grids with a hierarchical data structure, how the hierarchical data structure is constructed, and how the query works.

3.3. The Uniform Grids

We will consider the uniform grids of our data structure in this section. For a better understanding, we will ignore the hierarchical component for the moment. Thus, within this section, we will consider only one uniform grid of the static and one uniform grid of the dynamic object. How several uniform grids are organized within the hierarchical data structure will be considered in the following Section 3.4.

Notation 3.1 (Uniform grid): *We will denote a uniform grid with cell width w as $\mathcal{G}$. By $\mathcal{G}(\mathfrak{O})$ we will denote the uniform grid of a complex object $\mathfrak{O}$, which references the primitive of $\mathfrak{O}$ in its cells.*

An elementary operation during the construction of a uniform grid $\mathcal{G}$ as well as during grid queries is to obtain the grid cell that contains a query point.

Definition 3.1 (Localization and look-up): *Let $\boldsymbol{p} \in \mathbb{R}^3$ be a query point and $\mathcal{C} \in \mathcal{G}$ the grid cell that contains $\boldsymbol{p}$. The determination of the cell index of $\mathcal{C}$ is denoted as localization of $\boldsymbol{p}$. The access on $\mathcal{C}$ by its index is denoted as look-up.*

Each grid cell is addressed by an integer triple $(i, j, k) \in \mathbb{N}^3$. The index of the grid cell, where a point $\mathbf{p} \in \mathbb{R}^3$ is located in $\mathcal{G}$, is determined by the map $\mathbb{R}^3 \to \mathbb{N}^3$:

$$localize: \quad \mathbf{p} \mapsto \left(\left\lfloor \frac{p_x}{w} \right\rfloor, \left\lfloor \frac{p_y}{w} \right\rfloor, \left\lfloor \frac{p_z}{w} \right\rfloor \right)$$

The set of points that is mapped to one cell index, is partially closed and partially open. Because it is cumbersome to differ for intersection, tolerance and distance calculations between the open and the closed regions, we will consider for these calculations the

closure of this point set instead. This enables the consideration of a cell by a cube with edge length w. However, this is a conservative simplification as we can never miss any primitives. Only in very special cases it is possible that we test a primitive multiple times due to this simplification. Since it is normal that we test primitives multiple times anyway due to the nature of the grid, this additional effort is of no consequence.

The localization of a point is done in $O(1)$ time. But the localization determines only the index of the grid cell. How long it takes to look-up the grid cell by its index depends on the data structure that manages the grid cells. In the following section one can imagine the data structure of a grid as a 3d-array, where the access is in fact $O(1)$. A detailed discussion about data structures, our implementation and benchmarks are given in Section 4.2.

After we have introduced the above basics, we will discuss the content of a grid cell and introduce our two variants of a uniform grid in Section 3.3.1. In Section 3.3.2 we discuss how to define the uniform grid of the dynamic and of the static object and introduce two different approaches: A symmetric and an asymmetric approach. The central sections are Section 3.3.3 and Section 3.3.4, where we explain our algorithms of the symmetric and the asymmetric approach in detail. We explain our algorithms for the grid query in order to obtain all tolerance violating primitives in Section 3.3.3. In Section 3.3.4 we discuss how to handle large tolerance values and introduce our idea of core-primitives. Finally, we introduce an early out for the processing of the primitive tests in Section 3.3.5.

3.3.1. The Content of a Grid-Cell

The content of a grid cell depends on two issues: Which type of primitives are referenced and which is the criterion to refer a primitive by a cell. We consider both issues within this section.

Definition 3.2 (Cell Content): *We denote the set of primitives of a complex object $\mathfrak{O}$ that are referenced by a grid cell $\mathcal{C}$ as the cell content of $\mathcal{C}$ and denote it by $\mathfrak{O}_{\mathcal{C}}$.*

Referencing to Primitives

It is most common that a grid cell references to its intersecting primitives. Another common possibility is that a grid cell references only those primitives, whose reference point is mapped to the cell. The reference point could be for example the center point of the primitive's smallest enclosing sphere or one corner point of the axis aligned bounding box. Figure 3.5 shows an example. The variants differ in the used memory as well as in the way how queries are processed and thus in the performance of a query.

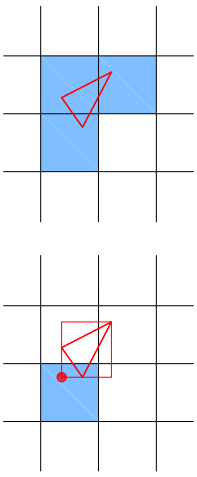


Fig. 3.5: A triangle referenced by all intersecting cells (top) and by the cell that contains its reference point (bottom).

We will denote in the following paragraphs the first variant as multi-reference variant and the second as single-reference variant.

Comparing both variants, the single-reference variant uses less memory than the multi-reference variant. On the other hand proximity queries are more complicated for the single-reference variant. As a basic sub-functionality during proximity queries, we need to know which primitives intersect a given cell. This request is easy to answer for the multi-reference variant – simply all primitives that are referenced. For the single-reference variant it is more effort to answer the query, because beside the cell of our interest, also some neighboring cells have to be queried for referenced primitives. Another basic sub-functionality during proximity queries is to determine the primitives that intersect some neighboring query cells. In this case the multi-reference variant has the disadvantage that primitives can be reported multiple times. On the other hand, the single-reference variant has the disadvantage that primitives that do not intersect any of the query cells are reported additionally. More details of both variants are given by Ericson [5]. Ericson's considerations are motivated by ray tracing and by collision detection of multiple objects. For tolerance queries we have to reconsider this issue.

Considerations for Tolerance Queries

The number of neighbor cells around a requested cell, which have to be queried additionally in the single-reference variant, depends on the width of the grid cells, on the size of the largest primitive and how the reference point is defined. Ericson suggests to set the cell width of the grid such that the largest object can be placed at any rotation into one grid cell. If the cell width is set like this, the additional cells are surely cells of the 1st-ring neighborhood of the requested cell. As we will see later in Section 3.3.3 we have to consider for tolerance queries primitives that intersect cells of a certain neighborhood that is enclosed by the R-ring neighborhood around a requested cell (R depends on the tolerance value δ). Thus, for a single-reference variant we would have to consider one additional ring and thereby the primitives that intersect cell of the $(R+1)$-ring neighborhood. We set our cell width usually smaller than Ericson's suggestion[3], thus, we will have more than one additional ring. As the number of cells grows cubically we should avoid this additional neighborhood ring(s) for performance reasons. Thus, possibly more than one additional neighborhood ring has to be considered. For this reasons we prefer the multi-reference variant.

TI-Grid and TD-Grid

We implement two variants of a grid. A *tolerance value independent grid* (TI-grid) and a *tolerance value dependent grid* (TD-grid). The TI-grid is a classical multi-reference grid, where each grid cell refers its intersecting primitives. We developed the idea of a TD-grid as a generalization of the classical TI-grid for more efficient tolerance queries. A grid cell of a TD-grid refers the primitives that have a distance smaller than $D(\delta)$ to the cell. The distance $D(\delta)$ depends on δ.

Definition 3.3 (TI-grid and TD-grid): *We denote a uniform grid with cells that reference the intersecting primitives as TI-grid and a uniform grid that reference the primitives with distance $\leq D(\delta)$ as TD-grid.*

[3] How we set our cell width is discussed in Section 4.4.2

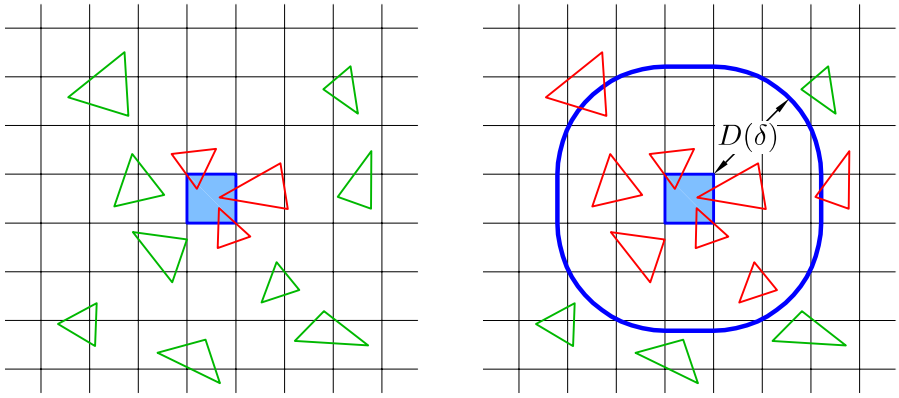


Fig. 3.6: Example of a cell in the TI-grid (a) and the TD-grid (b). The red triangles are referenced by the blue marked cell. These are the triangles that intersect the marked cell (a) and the triangles that have a smaller distance than $D(\delta)$ to the marked cell (b).

How we finally set $D(\delta)$ will be explained in the following Section 3.3.3. Both grids are illustrated in Figure 3.6.

The TD-grid obviously requires more memory per grid cell than a TI-grid because more primitives are referenced per cell. Moreover, more grid cells are occupied. This is clear because there are grid cells that do not intersect any primitive but maybe have a distance smaller than $D(\delta)$ to a primitive. Nevertheless, we will see later on the expense of this higher memory consumption the grid query to a TD-grid can be performed more efficient than to a TI-grid. How the query to a TI-grid or a TD-grid works is described in the next section.

Types of Primitives

A grid does not depend on the type of primitive that is referenced by the cells. Also all our algorithms are independent from the type of primitive. In order to use our solutions for a special type of primitives, still some basic tests are necessary, for example the primitive-primitive tolerance test for the tolerance query and further intersection and tolerance tests for the creation of the grid.

The issue of this work are fast and robust algorithms that can handle industrial geometry models. Most data from industry is available as unstructured set of triangles – often denoted as *triangle soup*. Such a triangle soup is represented in our work as a set of independent triangles. We do not require any topological attributes, especially we do not require a closed 2-manifold, which implies for a polyhedral mesh that each edge is adjacent to exactly two faces. This attribute is commonly denoted as watertight. As our complex objects are represented as sets of triangles, the type of primitives that are references by our grid cells are triangles.

3.3.2. The Grid of the Dynamic and the Grid of the Static Object

As mentioned in Section 3.2 we are using our combined data structure for the dynamic object as well as for the static object. But the two data structures differ in some small aspects. For example, as the dynamic object is the object we query with, the data structure of the static object should be efficient to answer queries, whereas this is not necessary for the dynamic object. Thus, the uniform grids of the static and of the dynamic object are not equal in detail. As it is sufficient to store the occupied grid cells of the dynamic grid in a simple list of cells, the grid of the static object is represented in a data structure with efficient cell look-up, e.g. in a complete 3d-array that allows a cell look-up in constant time. Refer for further discussions to Section 4.2.

Further, the uniform grids can also differ in the way the cell content is stored. Reconsider the definition of tolerance violation. Let A be a primitive of the static object $\mathfrak{S}$ and B be a primitive of the dynamic object $\mathfrak{D}$. A and B are tolerance violating iff

$$d(A,B) \leq \delta \quad \Leftrightarrow \quad (A \oplus S_\delta) \cap B \neq \emptyset \quad \Leftrightarrow \quad (A \oplus S_{\frac{\delta}{2}}) \cap (B \oplus S_{\frac{\delta}{2}}) \neq \emptyset \qquad (3.1)$$

where $A \oplus S_\delta$ is the δ-offset of A. $A \oplus S_{\frac{\delta}{2}}$ and $B \oplus S_{\frac{\delta}{2}}$ are the half-δ-offsets of A and B. The first equivalence is already know by Equation 1.1. The second equivalence can easily be derived. $(A \oplus S_{\frac{\delta}{2}}) \cap (B \oplus S_{\frac{\delta}{2}})$ is a symmetric and $(A \oplus S_\delta) \cap B$ an asymmetric formulation of the tolerance violation.

A Symmetric and an Asymmetric Approach

Based on the symmetric and the asymmetric formulation of the tolerance violation, we implement two approaches of the grid query, a symmetric and an asymmetric approach. They differ in the way the cell content is stored in the grids:

- Our first approach uses a TI-grid to store the static object as well as another TI-grid to store the dynamic object. This corresponds to the symmetric consideration of the tolerance violation $(A \oplus S_{\frac{\delta}{2}}) \cap (B \oplus S_{\frac{\delta}{2}})$ as we use the same way to reference primitives in both grids. We will denote this approach as the *symmetric approach*.
- Our second approach corresponds to the asymmetric consideration of the tolerance violation $(A \oplus S_\delta) \cap B$. As the δ-offset is only applied to the primitive A, we make only the grid that stores the static object responsible for the consideration of δ. Thus, for our second approach, we use a TD-grid to store the static object and a TI-grid to store the dynamic object. We will denote this approach as the *asymmetric approach*.

Both approaches will be presented in the following Sections 3.3.3 and 3.3.4.

3.3.3. The Basic Grid Query

Our grid query has similarity with the one in the voxmap-pointshell algorithm (refer to Section 1.3). A point or a voxel which represents a part of the dynamic object is located in a uniform grid which represents a static object. In contrast to McNeely et al.

[36] and Renz et al. [45] our intention is an exact calculation based on primitives.[4] In contrast to Reithmann [44] and Erbes [3] our grid query obtains all tolerance violating primitives and not only, in case of collision, one pair of colliding triangles.

Goal

Let us consider two complex objects: A static object $\mathfrak{S}$ and a dynamic object $\mathfrak{D}$. In every time step a transformation $\mathbf{T}$ is applied to object $\mathfrak{D}$ and our goal is to calculate the set of all tolerance violating primitives $T_\delta(\mathbf{T}\mathfrak{D}, \mathfrak{S})$ for every time step.

Cell-wise Processing

We calculate the set of all tolerance violating primitives $T_\delta(\mathbf{T}\mathfrak{D}, \mathfrak{S})$ by a cell-wise processing. Let $\mathcal{G}(\mathfrak{S})$ and $\mathcal{G}(\mathfrak{D})$ be the respective uniform grids of the static object $\mathfrak{S}$ and the dynamic object $\mathfrak{D}$ and w is the cell width of both grids. Let $\mathcal{C}$ be a cell in $\mathcal{G}(\mathfrak{D})$ and $\mathfrak{D}_\mathcal{C}$ the set of all primitives in $\mathfrak{D}$ that are referenced by $\mathcal{C}$ and $\mathfrak{P}_\mathcal{C} := \{P \cap \mathcal{C} \mid P \in \mathfrak{D}_\mathcal{C}\}$ the intersection of all primitives in $\mathfrak{D}_\mathcal{C}$ with the cell $\mathcal{C}$. The set of all tolerance violating primitives $T_\delta(\mathbf{T}\mathfrak{D}, \mathfrak{S})$ is equal to the union of all sets of tolerance violating primitives per cell $\mathcal{C}$

$$T_\delta(\mathbf{T}\mathfrak{D}, \mathfrak{S}) = \bigcup_{\mathcal{C} \in \mathcal{G}(\mathfrak{D})} T_\delta(\mathbf{T}\mathfrak{P}_\mathcal{C}, \mathfrak{S}) \,. \tag{3.2}$$

This is true, because

$$\mathfrak{D} = \bigcup_{\mathcal{C} \in \mathcal{G}(\mathfrak{D})} \mathfrak{P}_\mathcal{C} \,.$$

So we may calculate for each cell $\mathcal{C}$ the set of tolerance violating primitives $T_\delta(\mathbf{T}\mathfrak{P}_\mathcal{C}, \mathfrak{S})$ separately and finally build the union over all sets. Note, the primitives in $\mathfrak{P}_\mathcal{C}$ are generally not of the original type anymore. For example a triangle intersected with a cell is in general a polygonal face. However, we actually do not plan to calculate $\mathfrak{P}_\mathcal{C}$ or even perform any tolerance tests with these sectioned primitives. We just need this to estimate *candidate primitives*. Candidate primitives are primitives, which are possibly tolerance violating.

Let P be a primitive in $\mathfrak{D}_\mathcal{C}$ and Q a primitive in $\mathfrak{S}$. If $\mathbf{T}(P \cap \mathcal{C})$ is tolerance violating with Q then $\mathbf{T}P$ must also be tolerance violating with Q. Thus, if a primitive in $\mathfrak{S}$ is a candidate to be possibly tolerance violating with $\mathbf{T}(P \cap \mathcal{C})$, it must also be a candidate to be possibly tolerance violating with $\mathbf{T}P$. Because this is true for all $\mathcal{C} \in \mathcal{G}(\mathfrak{D})$ with $P \in \mathfrak{D}_\mathcal{C}$, we have

$$\{Q \in \mathfrak{S} | Q \text{ is candidate for } \mathbf{T}P\} = \bigcup_{\mathcal{C} \in \mathcal{G}(\mathfrak{D})} \{Q \in \mathfrak{S} | Q \text{ is candidate for } \mathbf{T}(P \cap \mathcal{C})\} \,.$$

This allows to iterate over all cells $\mathcal{C}$ in $\mathcal{G}(\mathfrak{D})$ in order to obtain the candidates for $\mathbf{T}(P \cap \mathcal{C})$ and test these candidates on tolerance violation with $\mathbf{T}P$. At the end $\mathbf{T}P$ is finally tested with all possible candidates in $\mathfrak{S}$.

[4] By *exact calculations based on primitives* we mean that the complex objects are not somehow overestimated or simplified. We do not mean numerically exact, although we endeavor to avoid floating point errors.

The Grid Query Algorithm

Algorithm 4 in Section 3.2 already showed our principal overall process. Now we will discuss the grid query (Line 8 to 15) more detailed.

Localization and Home Cell

For the grid query we iterate over all cells $\mathcal{C}$ of the dynamic grid $\mathcal{G}(\mathfrak{D})$. First, we localize $\mathbf{T}\mathcal{C}$ in the static grid: Let $\mathbf{c}$ be the center of $\mathcal{C}$. The cell index of the static cell, where $\mathbf{c}$ is located after the transformation, is obtained as

$$\left(\left\lfloor \frac{(\mathbf{Tc})_x}{w} \right\rfloor, \left\lfloor \frac{(\mathbf{Tc})_y}{w} \right\rfloor, \left\lfloor \frac{(\mathbf{Tc})_z}{w} \right\rfloor \right) .$$

We will denote the cell $\mathcal{H}$ in $\mathcal{G}(\mathfrak{S})$ with this cell index as the home cell of $\mathbf{T}\mathcal{C}$.

Candidate Cells and Candidate Primitives

For each cell $\mathcal{C}$ we obtain cells in $\mathcal{G}(\mathfrak{S})$ which contain *candidate primitives* for $\mathbf{T}\mathfrak{P}_\mathcal{C}$. We call these cells *candidate cells*. The determination of candidate primitives is a common broad phase task. The intention is to detect with little effort all the primitives that might be tolerance violating. This is done by using some simplifications and overestimation. We overestimate $\mathfrak{P}_\mathcal{C}$ by $\mathcal{C}$, because $\mathfrak{P}_\mathcal{C} \subseteq \mathcal{C}$. Hence, instead of searching for candidate primitives in $\mathfrak{S}$ that are tolerance violating with $\mathbf{T}\mathfrak{P}_\mathcal{C}$, we are searching for candidate primitives in $\mathfrak{S}$ that are tolerance violating with the transformed cell $\mathbf{T}\mathcal{C}$. Finally tolerance tests are performed between all primitives $\mathbf{T}\mathfrak{D}_\mathcal{C}$ and the obtained candidate primitives.

Primitive Tests

How the tolerance tests between all primitives $\mathfrak{D}_\mathcal{C}$ and all candidate primitives are processed is self-explanatory – provided the tolerance tests for the respective primitives are given (refer to Chapter 2). However, an additional note is given in Section 3.3.5.

To complete the grid query algorithm, we focus in the following on the determination of candidate primitives for $\mathbf{T}\mathcal{C}$. The algorithms how candidate primitives are determined is different for the symmetric and for the asymmetric approach. We describe both solutions in the following.

Determination of Candidate Primitives by the Symmetric Approach

Per definition, each cell in $\mathcal{G}(\mathfrak{S})$ references its intersecting primitives. As we have the same cell width for both grids, $\mathbf{T}\mathcal{C}$ partially overlaps its home cell $\mathcal{H}$ (or is identical). So candidate primitives in $\mathfrak{S}$ that are possibly tolerance violating with our query cell $\mathbf{T}\mathcal{C}$ are usually not only the primitives that are referenced in $\mathcal{H}$. Candidate primitives may be referenced by cells in a certain neighborhood of $\mathcal{H}$. This neighborhood depends on δ, the cell width w, and on the position of $\mathbf{T}\mathcal{C}$ in $\mathcal{G}(\mathfrak{S})$.

A primitive $Q \in \mathfrak{S}$ is tolerance violating with $\mathbf{T}\mathcal{C}$ iff their distance $d(Q, \mathbf{T}\mathcal{C}) \leq \delta$. In this case Q is a candidate to be tolerance violating with $\mathbf{T}\mathfrak{P}_\mathcal{C}$. We can express this by the terms of Equation 1.2 as

$$Q \cap (\mathbf{T}\mathcal{C} \oplus S_\delta) \neq \emptyset \;\Rightarrow\; Q \text{ is a candidate primitive} \tag{3.3}$$

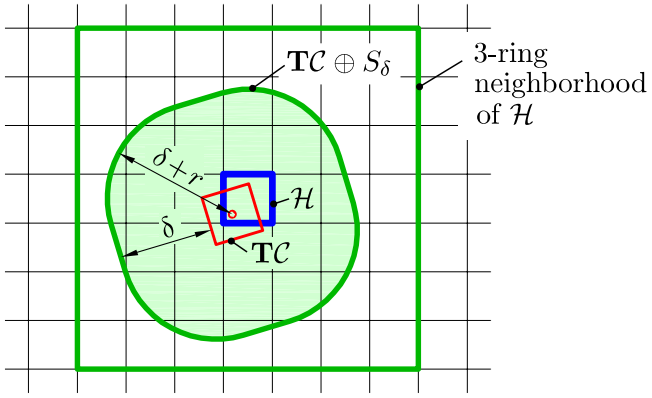


Fig. 3.7: Query cell $\mathbf{T}\mathcal{C}$ and its home cell $\mathcal{H}$ with offset $\mathbf{T}\mathcal{C} \oplus S_\delta$ inside of the 3-ring neighborhood of $\mathcal{H}$. All primitives in $\mathfrak{S}$ that intersect $\mathbf{T}\mathcal{C} \oplus S_\delta$ are tolerance violating with $\mathbf{T}\mathcal{C}$.

where $\mathbf{T}\mathcal{C} \oplus S_\delta$ is the δ-offset of $\mathbf{T}\mathcal{C}$. In Figure 3.7 $\mathbf{T}\mathcal{C} \oplus S_\delta$ is displayed as the green shaded region.

We overestimate the candidate primitives from Equation 3.3 by all the primitives which are referenced by a cell of the static grid that intersect $\mathbf{T}\mathcal{C} \oplus S_\delta$. As the center of $\mathbf{T}\mathcal{C}$ is located in $\mathcal{H}$ and as the maximum distance of a boundary point of $\mathbf{T}\mathcal{C} \oplus S_\delta$ to its center is exactly $\delta + r$ with $r = \frac{1}{2}\sqrt{3}w$ and w the cell width (see also Figure 3.7), $\mathbf{T}\mathcal{C} \oplus S_\delta$ can only intersect cells of the R-ring neighborhood of $\mathcal{H}$, where R is obtained as

$$R = \left\lceil \frac{\delta + r}{w} \right\rceil = \left\lceil \frac{\delta + \frac{1}{2}\sqrt{3}w}{w} \right\rceil = \left\lceil \frac{\delta}{w} + \frac{1}{2}\sqrt{3} \right\rceil . \tag{3.4}$$

The cells in the R-ring neighborhood build a superset of our candidate cells. As there are usually cells in the R-ring neighborhood of $\mathcal{H}$ that do not intersect $\mathbf{T}\mathcal{C} \oplus S_\delta$ and therefore do not contribute further candidate primitives, we reject some of them for

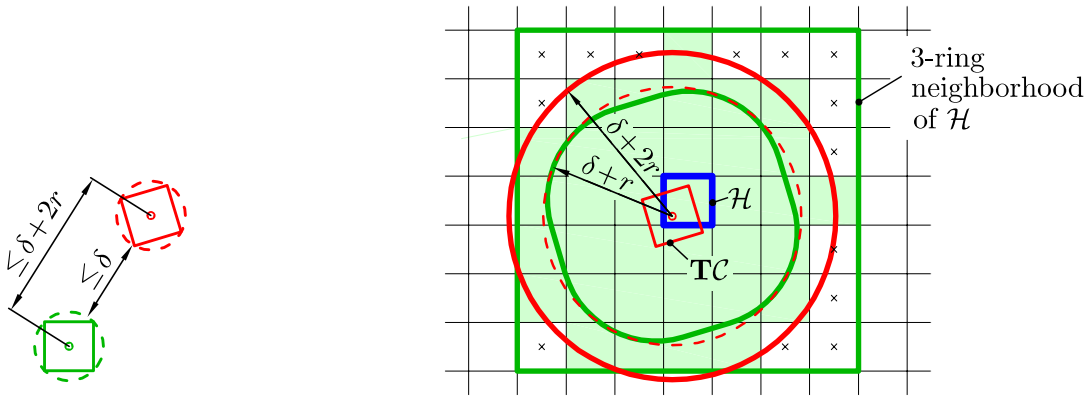


Fig. 3.8: Example on relevant candidate cells for TI-grids. The content of two cells can only be tolerance violating if the cell centers have a distance smaller than $\delta + 2r$ (left). So only the green shaded cells of the 3-ring neighborhood of $\mathcal{H}$ are candidate cells for the query cell $\mathbf{T}\mathcal{C}$ (right).

a tighter estimation of candidates. A cell $\mathcal{K}$ can be rejected, if $\mathbf{T}\mathcal{C}$ and $\mathcal{K}$ are not tolerance violating. A tolerance test between these two cells (one axis aligned and one arbitrary oriented cube) is too expensive – we require a much cheaper decision. On the expense of having some candidate cells that do not intersect $\mathbf{T}\mathcal{C} \oplus S_\delta$ we cover both cells by spheres with radius $r = \frac{1}{2}\sqrt{3}w$ and centers $\mathbf{Tc}$ and $\mathbf{k}$ respectively, where $\mathbf{k}$ is the center of the candidate cell. By this, $\mathcal{K}$ can be rejected, if the covering spheres are not tolerance violating, thus

$$\begin{aligned} &\mathcal{K} \in R\text{-ring neighborhood of } \mathcal{H} \ \wedge \ d(\mathbf{Tc}, \mathbf{k}) \leq \delta + 2r \\ &\Rightarrow \ \mathcal{K} \text{ is a candidate cell .} \end{aligned} \tag{3.5}$$

This issue is visualized in Figure 3.8. Only the green shaded cells in the 3-ring neighborhood of $\mathcal{H}$ are candidate cells for the query cell $\mathcal{C}$. The other cells of the 3-ring neighborhood of $\mathcal{H}$, which are marked with little black crosses, are rejected.

Summarized, the candidate primitives in $\mathfrak{S}$, which are possibly tolerance violating with

Algorithm 5: Grid Query in the Symmetric Approach

Data: δ // tolerance value
$\mathcal{G}(\mathfrak{D})$ // the grid of the dynamic object (TI-grid) with cell width w
$\mathcal{G}(\mathfrak{S})$ // the grid of the static object (TI-grid) with cell width w
$\mathbf{T}$ // transformation (of the dynamic object)

Result: T_δ // the set of all tolerance violating primitives

```
1  r ← ½√3 w, R ← ⌈½√3 + δ/w⌉
2  foreach occupied cell C in G(D) in parallel do
3      Tc ← apply T to the center of C
4      homeIdx ← localize Tc in G(S) and get respective cell index
5      for x = −R; x ≤ R; ++x do
6          for y = −R; y ≤ R; ++y do
7              for z = −R; z ≤ R; ++z do
8                  idx ← homeIdx + (x, y, z)
9                  K ← look-up for the cell with idx in G(S)
10                 if K is occupied then
11                     k ← get the center of K
12                     if d(Tc, k) ≤ δ + 2r then
13                         foreach primitive A in C do
14                             TA ← apply T on A
15                             foreach primitive B in K do
16                                 if A not marked in T_δ || B not marked in T_δ then
17                                     if d(TA, B) ≤ δ then
18                                         mark A and B in T_δ as tolerance violating
```

$\mathbf{T}\mathfrak{P}_{\mathcal{C}}$, are the primitives that are referenced by the obtained candidate cells based on Equation 3.5. This completes the grid query of the symmetric approach. The completed process is given in Algorithm 5. This code refines the lines 8 to 15 of Algorithm 4.

Determination of Candidate Primitives by the Asymmetric Approach

A bottleneck of the above explained grid query of the symmetric approach is that a cell $\mathcal{C}$ in $\mathcal{G}(\mathfrak{D})$ has many candidate cells in the R-ring neighborhood of its home cell $\mathcal{H}$ that have to be requested for candidate primitives. The number of candidate cells increases cubically in δ. Furthermore, candidate primitives can be obtained multiple times.

Our basic idea, within the design of the TD-grid that is used by the asymmetric approach to maintain the static object, was to avoid these problems. In Section 3.3.1 we define a TD-grid to reference in each cell the primitives that have a distance $\leq D(\delta)$ to the cell – but we leave open how $D(\delta)$ is set. Now we are going to fix $D(\delta)$ by the following demand: A query cell $\mathcal{C}$ in $\mathcal{G}(\mathfrak{D})$ should request only one candidate cell for candidate primitives, namely its home cell $\mathcal{H}$. Therefore we demand $\mathcal{H}$ to provide all candidate primitives for the ones in $\mathbf{T}\mathfrak{P}_{\mathcal{C}}$ for an arbitrary query cell $\mathcal{C}$ in $\mathcal{G}(\mathfrak{D})$. We fulfill our demand by setting $D(\delta)$ based on the following considerations of the grid query process:

The localized center point $\mathbf{Tc}$ can lie everywhere in $\mathcal{H}$. For an arbitrary position of $\mathbf{Tc}$ in $\mathcal{H}$ and an arbitrary orientation of $\mathbf{T}\mathcal{C}$, we have

$$\mathbf{T}\mathcal{C} \subset \mathcal{H} \oplus S_r \tag{3.6}$$

where $\mathcal{H} \oplus S_r$ is the Minkowski sum (see Definition 1.3) of the cell $\mathcal{H}$ and a ball S_r with radius $r = \frac{1}{2}\sqrt{3}w$ and center in the origin. $\mathcal{H} \oplus S_r$ is displayed in Figure 3.9 as the region surrounded by the red curve. Hence, an arbitrary point in an arbitrary query cell $\mathcal{C}$, is transformed to a point lying in $\mathcal{H} \oplus S_r$.

A primitive $Q \in \mathfrak{S}$ is tolerance violating with $\mathcal{H} \oplus S_r$ iff its distance $d(Q, \mathcal{H} \oplus S_r) \leq \delta$. In this case Q is a candidate to be tolerance violating with $\mathbf{T}\mathfrak{P}_{\mathcal{C}}$ for any orientation

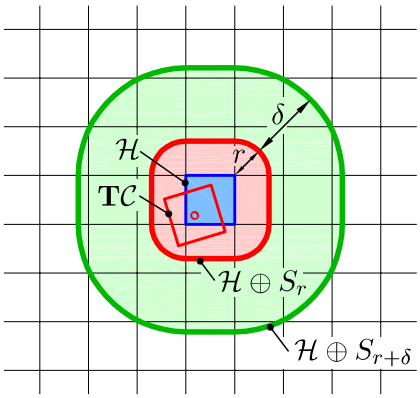


Fig. 3.9: Query cell $\mathbf{T}\mathcal{C}$ and its home cell $\mathcal{H}$ with offset regions $\mathcal{H} \oplus S_r$ and $\mathcal{H} \oplus S_{r+\delta}$.

Algorithm 6: Grid Query in the Asymmetric Approach

Data: δ // tolerance value
$\mathcal{G}(\mathfrak{D})$ // the grid of the dynamic object (TI-grid) with cell width w
$\mathcal{G}(\mathfrak{S})$ // the grid of the static object (TD-grid) with cell width w
$\mathbf{T}$ // transformation (of the dynamic object)

Result: T_δ // the set of all tolerance violating primitives

```
1  foreach occupied cell C in G(D) in parallel do
2      Tc ← apply T to the center of C
3      homeIdx ← localize Tc in G(S) and get respective cell index
4      H ← look-up for the cell with homeIdx in G(S)
5      if H is occupied then
6          foreach primitive A in C do
7              TA ← apply T on A
8              foreach primitive B in H do
9                  if A is not marked in T_δ || B is not marked in T_δ then
10                     if d(TA, B) ≤ δ then
11                         mark A and B in T_δ as tolerance violating
```

and position of $\mathbf{T}\mathcal{C}$ such that the center is located in $\mathcal{H}$. We can express this by the terms of Equation 1.2 as

$$Q \cap (\mathcal{H} \oplus S_{r+\delta}) \neq \emptyset \;\Rightarrow\; Q \text{ is a candidate primitive} \tag{3.7}$$

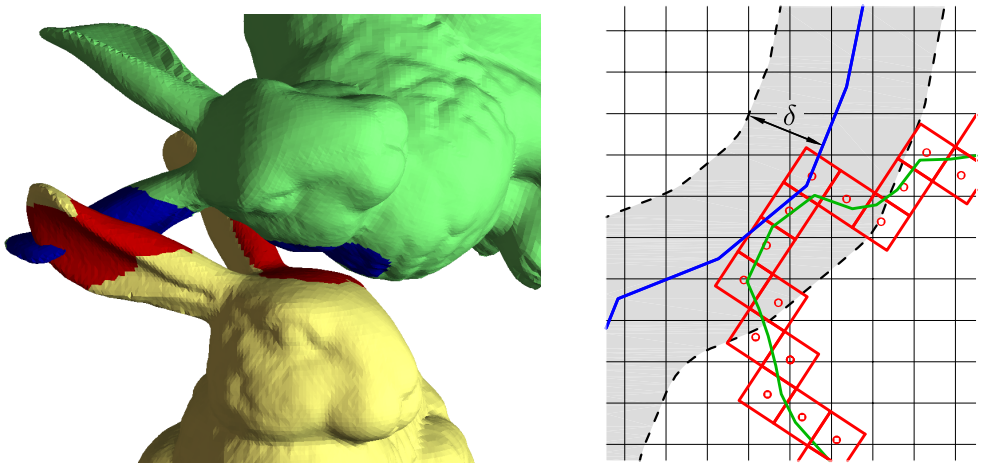
where $\mathcal{H} \oplus S_{r+\delta} = (\mathcal{H} \oplus S_r) \oplus S_\delta$ is the δ-offset of $\mathcal{H} \oplus S_r$. In Figure 3.9 the region surrounded by the green curve displays $\mathcal{H} \oplus S_{r+\delta}$.

Based on this we set $D(\delta) = r + \delta$ and thereby fulfill our demand that an arbitrary query cell $\mathcal{C}$ has only one candidate cell, its home cell $\mathcal{H}$, that provides all candidate primitives for $\mathbf{T}\mathfrak{P}_\mathcal{C}$ by just returning all its referenced primitives. Hence, no additional candidate cells in the neighborhood of $\mathcal{H}$ have to be queried and further, the prepared set of candidate primitives contains no multiple primitives. Both issues benefits the performance of the query algorithm.

The grid query is completed as given in Algorithm 6. Again, this code refines the lines 8 to 15 of Algorithm 4.

Evaluation and Anticipation of Experimental Results

Our Benchmarks, where the two above variants are compared, will show that the grid query of the asymmetric approach performs generally much better than the grid query of the symmetric approach. On the other hand, TI-grids are constructed faster than TD-grids, do not have to be re-constructed for queries with changing tolerance values δ



(a) (b)

Fig. 3.10: Large tolerance value δ. (a) Tolerance violating triangles (red and blue) between two bunnies. (b) 2d-example with a blue polygon with gray δ-offset inside of the black grid and a green polygon inside of red query cells.

and use less memory. Refer to Chapter 4 for further details and benchmarks. Further refer to Chapter 5 for a new approach how we reduce the memory consumption of TD-grids.

3.3.4. Core-Primitives: Handling Large Tolerance Violations

One issue of our data structures and algorithms is that we want them to be suitable for calculations with large and complete result outputs. This means, that even for large tolerance values δ we want all tolerance violating primitives to be determined efficiently. Of course the performance for such a calculation strongly depends on the size of the output, thus on the number of tolerance violating primitives. Nevertheless, we show within this section a method to reduce the calculation effort for large tolerance values.

Figure 3.10 (a) shows two tolerance violating Stanford bunnies with many tolerance violating triangles on the heads, throat and ears. In such a situation many primitive tests have to be calculated, but sometimes some primitive tests can be saved. Figure 3.10 (b) sketches such a situation with large tolerance value δ in 2d. The blue polygon represents the static object within a grid. The gray colored region is the δ-offset of the blue polygon. The green polygon represents the dynamic grid and the red cells the cells of the dynamic grid that reference the green polygon. Our idea is to find a simple criterion to decide on cell-basis if there are regions, where primitives are surly tolerance violating – without performing any expensive primitive tests.

Goal

Our goal is to calculate the set of all tolerance violating primitives $T_\delta(\mathbf{T}\mathfrak{D}, \mathfrak{S})$ for a static object $\mathfrak{S}$ and a transformed dynamic object $\mathbf{T}\mathfrak{D}$. Again, let $\mathcal{G}(\mathfrak{S})$ and $\mathcal{G}(\mathfrak{D})$ be the uniform grids of a static and a dynamic object respectively. The process, which

was presented in Section 3.3.3, iterates over all cells $\mathcal{C}$ in $\mathcal{G}(\mathfrak{D})$ in order to calculate the cell-wise set of tolerance violating primitives and build the union to obtain finally $T_\delta(\mathbf{T}\mathfrak{D}, \mathfrak{S})$. Here, in this cell-wise calculation, we integrate a criterion that decides whether all primitives $\mathbf{T}\mathfrak{D}_\mathcal{C}$ and some primitives in $\mathfrak{S}$ are definitely tolerance violating – without performing any primitive-primitive tolerance tests. Thereby $\mathfrak{D}_\mathcal{C}$ is again the set of all primitives in $\mathfrak{D}$ that are referenced by $\mathcal{C}$. We will denote a primitive in $\mathfrak{S}$ that is identified as definitely tolerance violating with $\mathbf{T}\mathfrak{D}_\mathcal{C}$ as a *core-primitive*.

Similar to the determination of candidate primitives, which are *possibly* tolerance violating with $\mathbf{T}\mathfrak{P}_\mathcal{C}$ (refer to Section 3.3.3), we are deriving a criterion for core-primitives, which are *definitely* tolerance violating primitives. For the determination of candidate primitives we *overestimated* $\mathbf{T}\mathfrak{P}_\mathcal{C}$ by $\mathbf{T}\mathcal{C}$. In contrast, for the determination of core-primitives, we have to *underestimate* this set by determining primitives in $\mathfrak{S}$ that are tolerance violating with all points in $\mathbf{T}\mathcal{C}$ and therefore with all primitives in $\mathfrak{D}_\mathcal{C}$.

As we have presented two variants of a grid query in Section 3.3.3, we present in the following also the determination of core-primitives for the grid query in the symmetric and in the asymmetric approach.

Determination of Core-Primitives for the Symmetric Approach

A primitive $Q \in \mathfrak{S}$ is definitely tolerance violating with $\mathbf{T}\mathcal{C}$ iff the distance $d(Q, \mathbf{p}) \leq \delta$ for *all* points $\mathbf{p} \in \mathbf{T}\mathcal{C}$. Let $\mathbf{p}_{max} \in \mathbf{T}\mathcal{C}$ be the point with

$$d(Q, \mathbf{p}_{max}) = \max \{d(Q, \mathbf{p}) | \mathbf{p} \in \mathbf{T}\mathcal{C}\} \ .$$

Criterion for Definite Tolerance Violations

A criterion for definite tolerance violations arises as

$$d(Q, \mathbf{p}_{max}) \leq \delta \ \Rightarrow \ Q \text{ is a core-primitive} \ . \tag{3.8}$$

The maximal distance between Q and $\mathbf{T}\mathcal{C}$ can only take place between Q and one of the eight corner points of $\mathbf{T}\mathcal{C}$. To simplify the determination of definite tolerance violating primitives, we enclose $\mathbf{T}\mathcal{C}$ in a sphere with radius $r = \frac{1}{2}\sqrt{3}w$ and center $\mathbf{c}$. Thereby we know for $d(Q, \mathbf{Tc})$, which is the minimal distance between Q and $\mathbf{Tc}$, that

$$d(Q, \mathbf{p}_{max}) \leq d(Q, \mathbf{Tc}) + r \ .$$

A primitive Q that fulfills $d(Q, \mathbf{Tc}) + r \leq \delta$ surely fulfills $d(Q, \mathbf{p}_{max}) \leq \delta$. So we have

$$\{Q \in \mathfrak{S} \mid d(Q, \mathbf{Tc}) \leq \delta - r\} \ \subseteq \ \{Q \in \mathfrak{S} \mid d(Q, \mathbf{p}_{max}) \leq \delta\}$$

and on the expense of missing some core-primitives a simpler criterion but less tight criterion arises as

$$d(Q, \mathbf{Tc}) \leq \delta - r \ \Rightarrow \ Q \text{ is a core-primitive} \ . \tag{3.9}$$

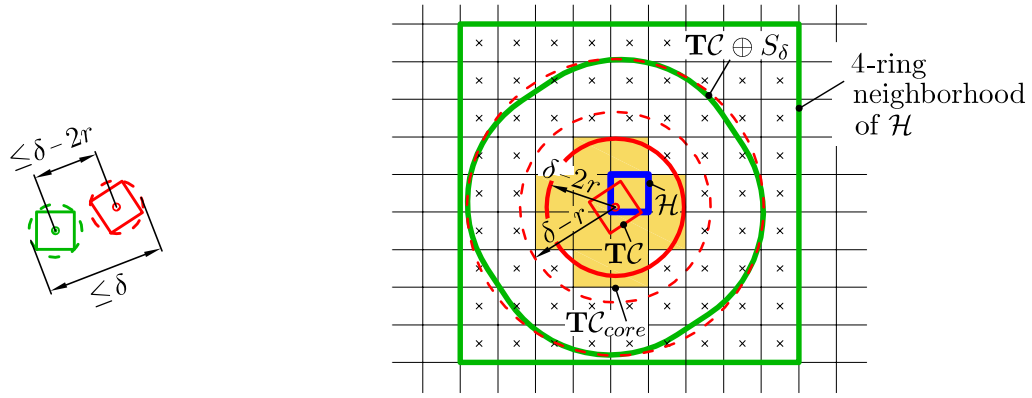


Fig. 3.11: Example on cells with definite tolerance violations for TI-grids. The content of two cells is completely tolerance violating if the cell centers have a distance smaller than $\delta - 2r$ (left). So the content of $\mathbf{T}\mathcal{C}_{core}$ (the orange shaded cells) is definitely tolerance violating with the content of $\mathbf{T}\mathcal{C}$ (right)

Similar to the determination of candidate primitives for possible tolerance violations on a simple cell-wise decision, we also want to determine surely tolerance violating primitives based on a simple cell-wise decision. Therefore we determine the cells $\mathcal{K}$ in the R-ring neighborhood of $\mathcal{H}$ (which we process while finding candidate primitives) that contain solely primitives that fulfill the above criterion. On the expense of missing some core-primitives, we cover $\mathcal{K}$ again by a sphere with radius $r = \frac{1}{2}\sqrt{3}w$ and center $\mathbf{k}$ (Section 3.3.3). So we have

Cell-wise Criterion for Definite Tolerance Violations

$$d(\mathbf{k}, \mathbf{Tc}) \leq \delta - 2r \Rightarrow \text{ all primitives in } \mathcal{K} \text{ are core-primitives .} \tag{3.10}$$

We denote the union of the cells that fulfill this cell-wise criterion as $\mathbf{T}\mathcal{C}_{core}$.

In Figure 3.11 the orange shaded cells in the 4-ring neighborhood of $\mathcal{H}$ visualize $\mathbf{T}\mathcal{C}_{core}$ and contain solely primitives that are definitely tolerance violating with any point in $\mathbf{T}\mathcal{C}$.

Integration to the Grid Query

The grid query, as presented at the beginning of in Section 3.3.3, processes over all cells to obtain candidate cells and thereby candidate primitives. Here in the cell-wise processing we introduce a check on definite tolerance violations per cell $\mathcal{C}$. The primitives in all candidate cells $\mathcal{K}$ with center $\mathbf{k}$ that fulfill $d(\mathbf{k}, \mathbf{Tc}) \leq \delta - 2r$ are marked as tolerance violating directly. In the case there was at least one primitive marked in that way, all primitives $\mathfrak{D}_\mathcal{C}$ are marked as tolerance violating, too. This is done prior to the narrow phase of the algorithm and thus, primitive-primitive tolerance tests can be saved.

Determination of Core-Primitives for the Asymmetric Approach

The determination of core-primitives in a TD-grid is completely different than for a TI-grid due to the different perspective. As we demand in Section 3.3.3 that a home

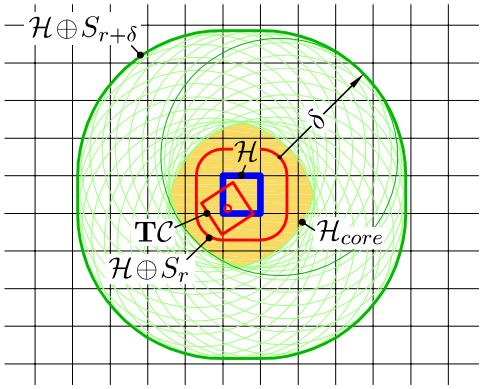


Fig. 3.12: Construction of $\mathcal{H}_{core}$ (orange shaded) in 2d as the intersection of all disks with radius δ and center in $\mathcal{H} \oplus S_r$. Any point inside of $\mathcal{H}_{core}$ is definitely tolerance violating with $\mathbf{T}\mathcal{C}$.

cell $\mathcal{H}$ has to provide all candidate primitives for the primitives in an arbitrary query cell $\mathbf{T}\mathcal{C}$, we demand now that the home cell $\mathcal{H}$ has to provide all core-primitives for an arbitrary query cell $\mathbf{T}\mathcal{C}$, too. It is clear that the set of all core-primitives (which are the definitely tolerance violating primitives) is always a subset of the set of all candidate primitives (which are the possibly tolerance violating primitives). So $\mathcal{H}$ already references the core-primitives for an arbitrary query cell. What we have to do is to label each primitive reference whether the primitive is a core-primitive or not. Therefore we need a criterion for definite tolerance violations.

In Section 3.3.3 we already showed that all points in $\mathbf{T}\mathcal{C}$ are definitely *not* tolerance violating, if there is no primitive of the static object that intersects $\mathcal{H} \oplus S_{r+\delta}$. Now we are deriving a similar criterion to decide whether all points in $\mathbf{T}\mathcal{C}$ are definitely tolerance violating. We know already by Equation 3.6 that for an arbitrary position of $\mathbf{Tc}$ in $\mathcal{H}$ and an arbitrary orientation of $\mathbf{T}\mathcal{C}$, we have $\mathbf{T}\mathcal{C} \subset \mathcal{H} \oplus S_r$.

Let $\mathbf{p}$ be a point in $\mathcal{H} \oplus S_r$. The point $\mathbf{p}$ is tolerance violating with all points in the set $\{\mathbf{q} \in \mathbb{R}^3 \mid d(\mathbf{p}, \mathbf{q}) \leq \delta\} = \mathbf{p} \oplus S_\delta$, which is a ball with center $\mathbf{p}$ and radius δ. Following, all points $\mathbf{p} \in \mathcal{H} \oplus S_r$ are tolerance violating with

$$\bigcap_{\mathbf{p} \in \mathcal{H} \oplus S_r} \mathbf{p} \oplus S_\delta \;:=\; \mathcal{H}_{core} \tag{3.11}$$

We denote $\mathcal{H}_{core}$ as the cell-core of $\mathcal{H}$. $\mathcal{H} \oplus S_r$ is displayed as the region, which is surrounded by the red curve in Figure 3.12. Some of the balls $\mathbf{p} \oplus S_\delta$ are visualized by green circles. Their intersection $\mathcal{H}_{core}$ is displayed by the orange shaded region.

Criterion for Definite Tolerance Violations

The criterion that a primitive Q in $\mathfrak{S}$ is definitely tolerance violation with every point $\mathbf{p} \in \mathcal{H} \oplus S_r$ arises as

$$Q \cap \mathcal{H}_{core} \neq \emptyset \;\Rightarrow\; Q \text{ is a core-primitive .} \tag{3.12}$$

Now we know that for any orientation and position of $\mathbf{T}\mathcal{C}$ with the center in $\mathcal{H}$, all primitives in $\mathfrak{S}$ that intersect $\mathcal{H}_{core}$ are definitely tolerance violating with all primitives in $\mathfrak{D}_\mathcal{C}$.

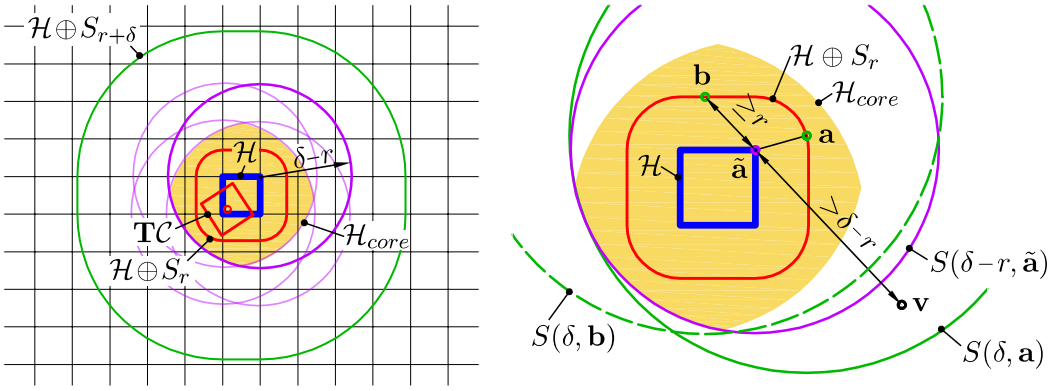


Fig. 3.13: Left: Construction of $\mathcal{H}_{core}$ (orange shaded) in 2d as the intersection of the 4 disks with radius $\delta - r$ and center in the cell's corner. Right: Proofing Equation 3.14.

Integration to the Grid Query

The grid query, as presented at the beginning of Section 3.3.3, processes over all cells $\mathcal{C}$ to obtain candidate primitives, which are provided by its respective home cell $\mathcal{H}$. Prior to the primitive-primitive tolerance tests of the narrow phase, we introduce a cell-core check. If $\mathcal{H}$ references at least one primitive which is labeled as core-primitive all core-primitives of $\mathcal{H}$ and all primitives in $\mathfrak{D}_\mathcal{C}$ are marked as tolerance violating directly. Thereby primitives can be marked as tolerance violating prior to the narrow phase and thus primitive-primitive tolerance tests can be saved.

Describing $\mathcal{H}_{core}$

Unfortunately, $\mathcal{H}_{core}$ is, as defined in Equation 3.11, unmanageable because infinitely many balls contribute to the definition of $\mathcal{H}_{core}$. So we will have a closer look on $\mathcal{H}_{core}$ in the following.

Since $\mathcal{H} \oplus S_r$ is a convex set, we can describe $\mathcal{H}_{core}$ as

$$\mathcal{H}_{core} = \bigcap_{\mathbf{p} \in \mathcal{H} \oplus S_r} \mathbf{p} \oplus S_\delta = \bigcap_{\mathbf{p} \in \partial(\mathcal{H} \oplus S_r)} \mathbf{p} \oplus S_\delta \,. \tag{3.13}$$

Consider two intersecting balls $S(\rho, \mathbf{c}_1), S(\rho, \mathbf{c}_2)$ with the same radius ρ but different centers $\mathbf{c}_1, \mathbf{c}_2 \in \partial(\mathcal{H} \oplus S_r)$. Any ball $S(\rho, \mathbf{c}_1 + \lambda(\mathbf{c}_2 - \mathbf{c}_1)), \lambda \in [0, 1]$, clearly contains $S(\rho, \mathbf{c}_1) \cap S(\rho, \mathbf{c}_2)$. A ball $S(\rho, \mathbf{c}_1 + \lambda(\mathbf{c}_2 - \mathbf{c}_1)), \lambda \in (0, 1)$ is superfluous for the intersection. So it is sufficient to intersect only the balls on the boundary $\partial(\mathcal{H} \oplus S_r)$ of the convex set $\mathcal{H} \oplus S_r$.

There are still infinitely many balls contributing to the definition of $\mathcal{H}_{core}$. Namely, each ball with the center on a curved part of the boundary of $\partial(\mathcal{H} \oplus S_r)$. Since $\mathcal{H}_{core}$ is the set of all points with distance $\leq \delta$ to every point in $\mathcal{H} \oplus S_r$ and $\mathcal{H} \oplus S_r$ is the set of points with distance $\leq r$ to $\mathcal{H}$, we have

$$\underbrace{\bigcap_{\mathbf{p} \in \partial(\mathcal{H} \oplus S_r)} \mathbf{p} \oplus S_\delta}_{(A)} = \underbrace{\bigcap_{\mathbf{p} \in \partial\mathcal{H}} \mathbf{p} \oplus S_{\delta - r}}_{(B)} \,. \tag{3.14}$$

For the equality $(A) = (B)$, consider that each ball $S(\delta, \mathbf{a}_0)$ with $\mathbf{a}_0 \in \partial(\mathcal{H} \oplus S_r)$ can be uniquely assigned to a ball $S(\delta - r, \tilde{\mathbf{a}})$, where $\tilde{\mathbf{a}}$ is the dropped perpendicular foot from

$\mathbf{a}_0$ onto $\partial\mathcal{H}$. The other way around, each ball $S(\delta - r, \tilde{\mathbf{a}})$ is assigned to one or more balls $S(\delta, \mathbf{a}_i)$ with $\mathbf{a}_i \in \partial(\mathcal{H} \oplus S_r)$, where $\tilde{\mathbf{a}}$ is the dropped perpendicular foot from each $\mathbf{a}_i$ onto $\partial\mathcal{H}$. Because $d(\tilde{\mathbf{a}}, \mathbf{a}_i) = r$, we have $S(\delta, \mathbf{a}_i) \supseteq S(\delta - r, \tilde{\mathbf{a}})$ for all i.

It is easy to see that $(A) \supseteq (B)$, because a point $\mathbf{v} \in (B)$ is inside of all $S(\delta - r, \tilde{\mathbf{a}})$ and therefore inside of all $S(\delta, \mathbf{a}_i)$ and therefore inside of (A).

For proving $(A) \subseteq (B)$ have a look at Figure 3.13 (b). Assumed $(A) \nsubseteq (B)$, then there is a point $\mathbf{v} \in (A)$ with $\mathbf{v} \notin (B)$. Then there must be a ball $S(\delta, \mathbf{a})$ with $\mathbf{v} \in S(\delta, \mathbf{a})$ and its assigned ball $S(\delta - r, \tilde{\mathbf{a}})$ with $\mathbf{v} \notin S(\delta - r, \tilde{\mathbf{a}})$. So we have $d(\tilde{\mathbf{a}}, \mathbf{v}) > \delta - r$. Consider the point $\mathbf{b} \in \partial(\mathcal{H} \oplus S_r)$ such that $\mathbf{v}$, $\tilde{\mathbf{a}}$ and $\mathbf{b}$ lie in this order on a common line. Due to the construction of $\mathcal{H} \oplus S_r$ every point in $\partial(\mathcal{H} \oplus S_r)$ has a distance $\geq r$ with any point in $\mathcal{H}$. Thus, we have $d(\tilde{\mathbf{a}}, \mathbf{b}) \geq r$. This leads to

$$d(\mathbf{v}, \mathbf{b}) \geq d(\mathbf{v}, \tilde{\mathbf{a}}) + d(\tilde{\mathbf{a}}, \mathbf{b}) > \delta - r + r = \delta \ .$$

Since $\mathbf{b} \in \partial(\mathcal{H} \oplus S_r)$, there is a ball $S(\delta, \mathbf{b})$. As we have $d(\mathbf{v}, \mathbf{b}) > \delta$ we conclude $\mathbf{v} \notin S(\delta, \mathbf{b})$. But this is contradictory to our assumption $\mathbf{v} \in (A)$, so $(A) \subseteq (B)$ and together $A = B$.

(B) is still the intersection of infinitely many balls, but the intersection is built over all points in $\partial\mathcal{H}$. Any point in $\partial\mathcal{H}$ can be described as a convex combination of the eight corner vertices $\mathbf{h}_i$ of $\mathcal{H}$. We know already that balls with center in points that can be described by the convex combination of other balls' centers are superfluous for the intersection. So we know that only the balls with center in the eight corner vertices of $\mathcal{H}$ contribute to the intersection. Thus

$$\bigcap_{\mathbf{p} \in \partial\mathcal{H}} \mathbf{p} \oplus S_{\delta - r} = \bigcap_{i=1}^{8} \mathbf{h}_i \oplus S_{\delta - r} \ . \tag{3.15}$$

Finally we have a simple description of $\mathcal{H}_{core}$ as intersection of only eight balls:

$$\mathcal{H}_{core} = \bigcap_{i=1}^{8} \mathbf{h}_i \oplus S_{\delta - r} \ . \tag{3.16}$$

Premise for Core-Primitives

Considering the symmetric approach, the size of $\mathbf{TC}_{core}$ depends on the size of the tolerance value δ and on the cell width w of the grid. Considering the asymmetric approach, the size of $\mathcal{H}_{core}$ depends on δ and w, too. In both cases it is possible that for some δ and w the criteria for core-primitives is never fulfilled. This takes place if

$$\delta - 2r < 0$$

with $r = 1/2\sqrt{3}w$.

Consider the symmetric approach: In the case of $\delta - 2r < 0$ the cell-wise criterion (Equation 3.10) is $d(\mathbf{k}, \mathbf{Tc}) \leq \delta - 2r < 0$, which can never be fulfilled.

Consider the asymmetric approach: In the case of $\delta - 2r < 0$ the intersection of the eight balls with radius $\delta - r$ in the corner vertices of $\mathcal{H}$ is empty and thus $\mathcal{H}_{core} = \emptyset$.

Table 3.1: The size and the behavior of S_{core} dependent on the tolerance value δ and the cell radius r where $\mathcal{X}$ stands for $\mathbf{T}\mathcal{C}$ in the case of a TI-grid and for $\mathcal{H}$ in the case of a TD-grid.

$\frac{\delta}{r}$	radius of S_{core} $= \delta - 2r$	behavior of S_{core}
< 2	< 0	no core-primitives possible, $S_{core} = \emptyset$
≥ 2	≥ 0	core-primitives possible, $S_{core} \neq \emptyset$
$\leq 2 + 1/\sqrt{3}$	$\leq 1/2\ w$	S_{core} is completely enclosed in $\mathcal{X}$
≥ 3	$\geq r$	S_{core} completely encloses $\mathcal{X}$
≥ 4	$\geq 2r$	S_{core} is large enough to omit intersection tests [5]
≥ 5	$\geq 3r$	S_{core} completely encloses the 1-ring neighborhood of $\mathcal{X}$

Estimating the Amount of Core-Primitives

S_{core}

The amount of core-primitives for a fixed cell width w increases with the tolerance value δ. The region that is considered to determine the core-primitives is different for the symmetric and the asymmetric approach. But in both cases the region can be approximated by a ball with radius $\delta - 2r$, which we denote as S_{core}:

- *Consider the symmetric approach*: The region that is considered to provide the core-primitives is $\mathbf{T}\mathcal{C}_{core}$ (Equation 3.10).
  The ball $S(\delta - 2r, \mathbf{Tc})$ with radius $\delta - 2r$ and the same center as the transformed query cell $\mathbf{T}\mathcal{C}$ approximates $\mathbf{T}\mathcal{C}_{core}$. This is because only cells with center inside of $S(\delta - 2r, \mathbf{Tc})$ are part of $\mathbf{T}\mathcal{C}_{core}$ according to the core-primitive criterion and because the radius of a cell is r. So all point in cells that fulfill the core-primitive criterion must be inside of the ball $S(\delta - r, \mathbf{Tc})$ and therefore we have $\mathbf{T}\mathcal{C}_{core} \subset S(\delta - r, \mathbf{Tc})$. And for the same reason, every point in the ball $S(\delta - 3r, \mathbf{Tc})$ must be inside of a cell that fulfill the core-primitive criterion and therefore we have $S(\delta - 3r, \mathbf{Tc}) \subset \mathbf{T}\mathcal{C}_{core}$.
  Together we have $S(\delta - 3r, \mathbf{Tc}) \subset \mathbf{T}\mathcal{C}_{core} \subset S(\delta - r, \mathbf{Tc})$. Thus, $\mathbf{T}\mathcal{C}_{core}$ can be approximated by the ball $S(\delta - 2r, \mathbf{Tc})$ and so by S_{core}. Refer to Figure 3.14 for an illustration.

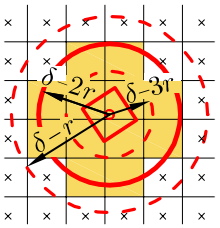


Fig. 3.14: Approximation of $\mathbf{T}\mathcal{C}_{core}$.

- *Consider the asymmetric approach*: The region that is considered to provide core-primitives in a TD-grid is $\mathcal{H}_{core}$ (Equation 3.12). The ball $S(\delta - 2r, \mathbf{h})$ with radius $\delta - 2r$ and the same center $\mathbf{h}$ as $\mathcal{H}$ is the largest enclosed ball in $\mathcal{H}_{core}$ and only slightly smaller than $\mathcal{H}_{core}$. Thus, $\mathcal{H}_{core}$ can be conservatively approximated by S_{core}.

Behavior of S_{core}

Table 3.1 shows the behavior of S_{core} depending on δ.

S_{cand}

Not only the amount of core-primitives increases with δ, also the amount of candidate primitives increases. Therefore the ratio of the region that is considered to provide the core-primitives and the region that is considered to provide the candidate primitives is interesting. The region that is considered to determine the candidate primitives is

Table 3.2: Portion of the radius $\delta - 2r$ of S_{core} on the radius $\delta + 2r$ of S_{cand} (left) and portion of the volume of S_{core} on the volume of S_{cand} when $r = 5$ and thereby $w = 10/\sqrt{3}$ (right).

δ	$\frac{\text{radius } S_{core}}{\text{radius } S_{cand}}$	δ	volume S_{cand}	volume S_{core}	$\frac{\text{volume } S_{core}}{\text{volume } S_{cand}}$
$2r$	0	10	33,510	0	0.00
$3r$	$1/5 = 0.2$	15	65,450	524	0.01
$4r$	$1/3 = 0.333...$	20	113,097	4,189	0.04
$5r$	$3/7 = 0.428...$	25	179,594	14,137	0.08
$6r$	$1/2 = 0.5$	30	268,083	33,510	0.13
$7r$	$5/9 = 0.555...$	35	381,704	65,450	0.17
$8r$	$3/5 = 0.6$	40	523,599	113,097	0.22
$14r$	$3/4 = 0.75$	70	2,144,661	904,779	0.42
$38r$	$9/10 = 0.9$	190	33,510,322	24,429,024	0.73

different in the symmetric and in the asymmetric approach. But in both cases the region can be approximated by a ball with radius $\delta + 2r$, which we denote as S_{cand}:

- *Consider the symmetric approach*: The region that is considered to provide the candidate primitives is the union of the cells that fulfill Equation 3.5. By a similar argumentation as above for S_{core}, we can show that this region can be approximated by the ball $S(\delta + 2r, \mathbf{Tc})$ with radius $\delta + 2r$ and the same center as the transformed query cell $\mathbf{T}\mathcal{C}$. So it can be approximated by S_{cand}.
- *Consider the asymmetric approach*: The region that is considered to provide candidate primitives is $\mathcal{H} \oplus S_{r+\delta}$ (Equation 3.7). The ball $S(\delta+2r, \mathbf{h})$ with radius $\delta + 2r$ and the same center $\mathbf{h}$ as $\mathcal{H}$ is the smallest enclosing ball around $\mathcal{H} \oplus S_{r+\delta}$ and only slightly larger than $\mathcal{H} \oplus S_{r+\delta}$. Thus $\mathcal{H} \oplus S_{r+\delta}$ can be conservatively approximated S_{cand}.

Ratio of S_{core} and S_{cand}

We estimate the ratio between the region which has to be considered to detect core-primitives and the region which has to be considered to detect candidate primitives by the ratio of S_{core} and S_{cand}. The larger the ratio, the greater is the portion on tolerance violating primitives that are marked directly by the detection as core-primitive and the the less is the portion on primitives that have to be tested by an expensive primitive-primitive tolerance tests.

A large ratio means also that the cell width is very small compared to δ. Choosing for a fixed δ intentionally a very small cell width when constructing the grids can be counterproductive as the cell width do not match the size of the primitives and the total calculation effort increases (refer to Section 4.4.2 for this issue).

Table 3.2 (a) shows the portion of the radius $\delta - 2r$ of S_{core} on the radius $\delta + 2r$ of S_{cand}. For increasing δ the ratio increases and approaches the limit 1 for an infinitely large δ where all candidate primitives are core-primitives.

[5] Refer to the next subsection *Special Benefit for the Remaining Primitive-Primitive Tolerance Tests* for an explanation.

The ratio of the radius is only an one dimensional consideration – the consideration of the volume relation is more meaningful. Table 3.2 (b) shows the portion of the volume of S_{core} on the volume of S_{cand} on the example that $r = 5$ and thereby $w = 10/\sqrt{3}$. Of course the behavior is similar to the 1-dimensional case, although the volume ratio of S_{core} and S_{cand} approaches the limit 1 slower than the radius ratio $(\delta - 2r)/(\delta + 2r)$.

Special Benefit for the Remaining Primitive-Primitive Tolerance Tests

As mentioned above, the detection of core-primitives saves some of the primitive-primitive tolerance tests, because some primitives are marked as tolerance violating without calculating the primitive-primitive tolerance test. Beside this big benefit, there also a benefit for the remaining primitive-primitive tolerance tests.

Consider the following fact: A primitive P in $\mathfrak{D}$ can only intersect primitives in $\mathfrak{S}$ in the case that at least one of the cells in the dynamic grid that references P intersects primitives in $\mathfrak{S}$. Within our cell-wise processing during the grid query, we consider per cell $\mathcal{C}$ the primitives $\mathfrak{D}_\mathcal{C}$ that are referenced by $\mathcal{C}$. We have denoted the intersections of all primitives in $\mathfrak{D}_\mathcal{C}$ with $\mathcal{C}$ as $\mathfrak{P}_\mathcal{C}$ (Section 3.3.3). The important fact is that an intersection between $\mathbf{T}\mathfrak{P}_\mathcal{C}$ and primitives in $\mathfrak{S}$ can only take place if $\mathbf{T}\mathcal{C}$ intersects primitives in $\mathfrak{S}$.

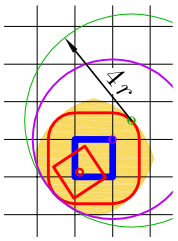


Fig. 3.15: $\mathcal{H}_{core}$ encloses every $\mathbf{T}\mathcal{C}$ for $\delta \geq 4r$.

Assumed that for every query cell $\mathbf{T}\mathcal{C}$ the primitives in $\mathfrak{S}$ that are intersected by $\mathbf{T}\mathcal{C}$ are core-primitives. In this case all candidate primitives that intersect $\mathbf{T}\mathcal{C}$ and the primitives $\mathfrak{D}_\mathcal{C}$ are always marked as tolerance violating and need not to be tested. Thus, we can premise for the remaining primitive-primitive tolerance tests that the primitives do not intersect. Thereby we can take advantage in the calculation effort for the primitive-primitive tolerance tests. For example in a triangle-triangle feature test the face-edge test needs not to be calculated. Especially interesting is this benefit for volumetric primitives like tetrahedrons.

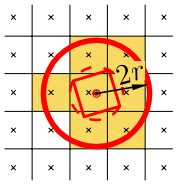


Fig. 3.16: $\mathbf{T}\mathcal{C}_{core}$ encloses always $\mathbf{T}\mathcal{C}$ for $\delta \geq 4r$.

We assumed that all primitives in $\mathfrak{S}$ that intersect a query cell $\mathbf{T}\mathcal{C}$ are core-primitives. In the following we will show under which conditions this can be guaranteed.

- *Consider the symmetric approach*: For an arbitrary position and orientation of the query cell $\mathbf{T}\mathcal{C}$ in its home cell $\mathcal{H}$ we always have $\mathbf{T}\mathcal{C} \subseteq \mathcal{H} \oplus S_r$ (refer Equation 3.6). So we want $\mathcal{H} \oplus S_r \subseteq \mathcal{H}_{core}$. This is fulfilled iff $\delta \geq 4r$. Refer to Figure 3.15.
- *Consider the asymmetric approach*: For $\delta \geq 3r$ we have $\mathbf{T}\mathcal{C} \subseteq S_{core}$. But S_{core} is only an approximation of $\mathbf{T}\mathcal{C}_{core}$ and we have to guarantee $\mathbf{T}\mathcal{C} \subseteq \mathbf{T}\mathcal{C}_{core}$ for an arbitrary position and orientation of $\mathbf{T}\mathcal{C}$. Therefore $\delta - 2r$ must be large enough to fulfill Equation 3.5 $d(\mathbf{Tc}, \mathbf{k}) \leq \delta - 2r$ for any cell $\mathcal{K}$ with center $\mathbf{k}$ that intersects $\mathbf{T}\mathcal{C}$. $\mathcal{K}$ and $\mathbf{T}\mathcal{C}$ can only intersect iff $d(\mathbf{Tc}, \mathbf{k}) \leq 2r$, thus we must have $\delta \geq 4r$. Refer to Figure3.16.

Summarized, for $\delta \geq 4r$ all primitives in $\mathfrak{S}$ that intersect an arbitrary query cell $\mathbf{T}\mathcal{C}$ are definitely detected as core-primitives and thereby all remaining primitive-primitive tolerance tests can premise that the primitives do not intersect.

3.3.5. Early Out for Known Tolerance Violations

One sub-task of the whole grid query (as described in the previous Section 3.3.3) is that tolerance tests between some candidate primitives and the primitives of a query cell have to be performed. In principle this is straight forward, except for one very significant issue: We introduced a so called *early out* for known tolerance violations.

The calculation of all tolerance violating primitives requires that we have to decide for each primitive whether it is tolerance violating or not. Thus, we cannot finish the calculation after the first tolerance violating pair of primitives is found. This kind of early out comes only into question for Boolean proximity queries, as we have explained in Section 1.2. For the query on the set of all tolerance violating primitives, we use another kind of early out.

Imagine two complex object $\mathfrak{O}_1$ and $\mathfrak{O}_2$ with the primitives $A, B \in \mathfrak{O}_1$ and $C, D \in \mathfrak{O}_2$ in the following situation: During our calculation, the pairs (A, C) and (B, D) were detected as tolerance violating. Later the pair (A, D) should be tested. Because we already know that A and D are tolerance violating, we can omit the tolerance test for the pair (A, D).

General Formulation

More generally, the tolerance test for a pair of primitives can be omitted, if both primitives are already known to be tolerance violating.

Please note, this early out can only be used in our case that we want to calculate the set of all tolerance violating primitives. For the calculation of all pairs of tolerance violating primitives this cannot be used. This is the main reason why the calculation of the set is less effort than the calculation of the pairs. However, our approach can be easily modified in order to calculate all pairs of tolerance violating primitives, by just deactivating this early out and output the pair-wise results.

This early out is already used in line 16 in Algorithm 5 and in line 9 in Algorithm 6. It proves to be very efficient and saves many primitive tests.

3.4. Combination with a Hierarchical Data Structure

This section describes how several uniform grids are organized within the hierarchical data structure. As mentioned in the introduction of this chapter, a simple uniform grid lacks of a hierarchical structure. According to our principal algorithm (refer to Algorithm 4), we have to iterate over all cells in the grid of the dynamic object and look-up for cells in the grid of the static object in order to finally find all tolerance violating primitives. The main problem in this procedure is the number of queries. Especially if the grid of the dynamic object is large, there are many cells to query with. Usually the dynamic and the static object are overlapping or tolerance violating only in some relatively small regions. Thus, many queries come to the same conclusion: There are no static grid cells for the query cell with candidate primitives (refer to line 10 in Algorithm 4).

In order to avoid many unnecessary cell queries, we are going to organize our grid cells within a hierarchical structure. Assumed, we are given the uniform grid $\mathcal{G}(\mathfrak{O})$ of a complex object $\mathfrak{O}$. The occupied cells of $\mathcal{G}(\mathfrak{O})$ are the cells that reference to primitives in $\mathfrak{O}$. They can be considered as a set of independent cells $C := \{\mathcal{C}_0, ..., \mathcal{C}_n\}$, which is geometrically a set of cubes.

Our intention is to create a hierarchical data structure over the cells in C. The cells in C are thereby covered by the regions of the leafs (for example in case of a bounding volume hierarchy, all cells in C are covered by the bounding volumes of the leafs). We assign the cells in C uniquely to the leafs. Let N be the number of leafs of the hierarchical structure and $C^{(i)} \subseteq C$ the subset of cells that are assigned to leaf i, where

$$\bigcup_{i=1}^{N} C^{(i)} = C$$

and for all $i, j \in \{1, ..., N\}$ with $i \neq j$

$$C^{(i)} \cap C^{(j)} = \emptyset \, .$$

We force the hierarchical structure to be flat enough to maintain enough cells per leaf in order to build a small uniform grid. This can be controlled by either restricting the number of hierarchy levels or by restricting the number of cells per leaf.

For each leaf i we create a uniform grid $\mathcal{G}^i(w)$ in the region that is covered by the leaf. The uniform grid $\mathcal{G}^i(w)$ is created such that the occupied cells of $\mathcal{G}^i(w)$ are exactly the cells in $C^{(i)}$.

Figure 3.17 gives an impression on such a combined data structure.

The first step in the whole tolerance query is to determine the pairs of intersecting or tolerance violating leafs between the hierarchical components of the dynamic and the static data structure (refer to Algorithm 4, Line 4). For our symmetric approach we calculate the list of all tolerance violating leafs pairs and for our symmetric approach we calculate the list of all intersecting leafs pairs. All these leaf pairs are then processed by the grid query (refer to Algorithm 4, Line 5).

How the hierarchical data structure is finally created and how the query will be implemented in detail, depends on the type of the hierarchy. We implemented a multi-grid and different variants of bounding volume hierarchies.

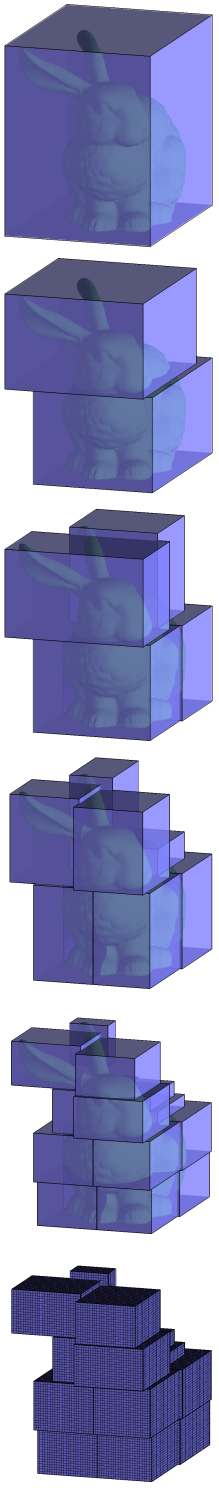


Fig. 3.17: 5-levels of a BVH with axis aligned boxes an 16 uniform grids in the base level.

Multi-Grid

The most natural way to organize uniform grids within a hierarchy are multi-grids. In order to use multi-grids for our task, we first create a uniform grid $\mathcal{G}(\mathfrak{O})$ that covers the complete object $\mathfrak{O}$. Let N_x, N_y, N_z be the number of grid cells in x-, $y-$ and z-direction of the grid $\mathcal{G}(\mathfrak{O})$. Then there is a $N \in \mathbb{N}$ with $2^{N-1} < \max(N_x, N_y, N_z) \leq 2^N$. We increase the original grid $\mathcal{G}(\mathfrak{O})$ to the extent of 2^N cells in every direction by just adding the respective number of (empty) cells to the borders of the original grid.

We consider this increased grid as all cells in the base layer of the multi-grid and define upper layer grids on its basis. All upper layer grids should match to the occupied cells C of the base layer grid – this means that the each cell $\mathcal{C}_i \in C$ can be uniquely assigned to a cell in the upper layer grid. The cell width of an upper layer is twice the cell size of its next lower layer. Blocks of eight cells, which can be merged to one cell of an upper layer, are called the children of this cell. If a cell in an upper layer has at least one occupied child, this cell is denoted as occupied too.

There are N possible upper layer grids. Using all $N + 1$ layer results into a complete octree. As our intention is to keep enough cells per leaf to build a small uniform grid, we use only the first few layers of this complete octree. Using the first $M + 1$ layers results in 2^{3M} leafs. For each leaf we define one uniform grid with $2^{3(N-M)}$ cells.

At query time we have to calculate all intersecting / tolerance violating leaf pairs between the multi grid of the dynamic and of the static data structure. As our hierarchical components are multi-grids, the principle is similar to the grid query which is described in the previous section. Each level of the hierarchy is thereby one uniform grid. Let us denote H_D^i the uniform grid of the i-th level in the multi-grid of the dynamic object and H_S^i respectively for the static object. For every level i, from the top to the bottom level, all occupied cells in H_D^i are localized in H_S^i. For each localized cell $\mathcal{C}$ in H_D^i the intersecting / tolerance violating cells in H_S^i have to be obtained. If one of the obtained cells in H_S^i is occupied, the children of $\mathcal{C}$ have to be checked within the next hierarchy level. In the base level the process is likewise, except that for each occupied intersecting / tolerance violating cell in H_S^i we build a pair with the localized cell $\mathcal{C}$ and push it to the result list of all leaf pairs.

BVH

The idea of using a BVH to focus on relevant parts of the grid is not that straight forward as using multi-grids. But, if we see the set of cells $C := \{\mathcal{C}_0, ..., \mathcal{C}_n\}$ of the original grid $\mathcal{G}(\mathfrak{O})$ more geometrically as a set of cubes, we see that it is natural and usual to create a BVH over the set of cubes – as it is usual for any kind of geometry.

We create the hierarchy by a top-down approach by a recursive process: Each node of the hierarchy is constructed by a set of cubes. First a bounding volume is calculated to cover all cubes, then the set of cubes is split into two parts. For each part the recursion is called again in order to create the child nodes. The recursion stops, if either a limit of minimal cubes per node is reached or a given number of levels is reached. We implemented the BVH with different bounding volumes: Axis aligned bounding boxes, orientated bounding boxes, and spheres.

At query time we have to calculate all intersecting / tolerance violating leaf pairs between the BVH of the dynamic and of the static data structure. This is done by a usual BVH traversal.

4. Implementation and Benchmarks

Within this chapter we benchmark our algorithms to calculate all tolerance violating triangles per motion step on real-life data and on academic models. Further, we will discuss some details of the implementation. We will first introduce our test environment in Section 4.1. Then we consider the data structure of the uniform grids in Section 4.2 and of the hierarchical part of our data structure in Section 4.3. After that we discuss in Section 4.4 the parameters of our data structure, which are the cell width of the grid and the depth of the hierarchical part. In Sections 4.5 and 4.6 we finally present the results of our symmetric approach (the grids of the dynamic as well as of the static object are TI-grids) and of our asymmetric approach (the grids of the dynamic object are TI-grids and the grids of the static object are TD-grids).

4.1. Test Environment

For the benchmarks of this chapter we work with the real-life scenarios *Engine* and *EngineBig* and with the academic scenarios *Bunny* and *Buddha*. The models of the *Engine* and *EngineBig* scenario are the front part of a car's bodyshell and a complete engine, respectively. The models of the academic scenarios are two models of the Stanford Bunny and two models of the Standford Happy Buddha. The scenarios are presented more detailed in the Appendix A.1, A.2, A.3, and A.4.

For our benchmarks we calculate all tolerance violating triangles for a number of different transformations of the dynamic object. The sets of transformations are also described in the appendix. For each of the above four scenarios we use three different types of transformation sets:

- *ColTolVerl*: A set of sampled random transformations such that in each transformation step we have surely tolerance violations and usually also collisions.
- *NoColTolVerl*: A set of sampled random transformations such that in each transformation step we have surely tolerance violations but no collisions.
- *DisassMotion*/ *ExtremeMotion*: A set of transformations that describe a smooth motion path of the dynamic object. *DisassMotion* describes a disassembly path of the engine out of the car's bodyshell. *ExtremeMotion* describes a fictional motion of one model, penetrating in almost every step the other model.

In summary we have 12 test cases. Refer to Table 4.1 for an overview.

	ColTolVerl	NoColTolVerl	DisassMotion	ExtremeMotion
Engine	X	X	X	
EngineBig	X	X	X	
Bunny	X	X		X
Buddha	X	X		X

Table 4.1: Used benchmark scenarios and transformation sets.

For each test case we calculate all tolerance violating triangles per motion step for varying tolerance values. We consider for the scenarios tolerance values in the following ranges:

- *Engine*: $\delta \in [5, 30]$
- *EngineBig*: $\delta \in [0.5, 3.0]$
- *Bunny*: $\delta \in [0.001, 0.004]$
- *Buddha*: $\delta \in [0.0002, 0.0006]$

The tolerance value that is used for the models of the *Engine* scenario in practical applications is about $\delta = 15$. The tolerance value that is used for the models of the *EngineBig* scenario in practical applications is about $\delta = 1.5$.[1] Thus, we verify our algorithms for tolerance values that include the one of real life applications as well as some smaller and some larger tolerance values.

The used tolerance ranges in the *Bunny* and the *Buddha* scenario are adapted from the ones in the *Engine* and the *EngineBig* scenario in relation to the scaling of the models.

All algorithms are compared by the achieved number of *tests per second* (tps), which is the number of positions divided by the total running time for all positions in seconds.

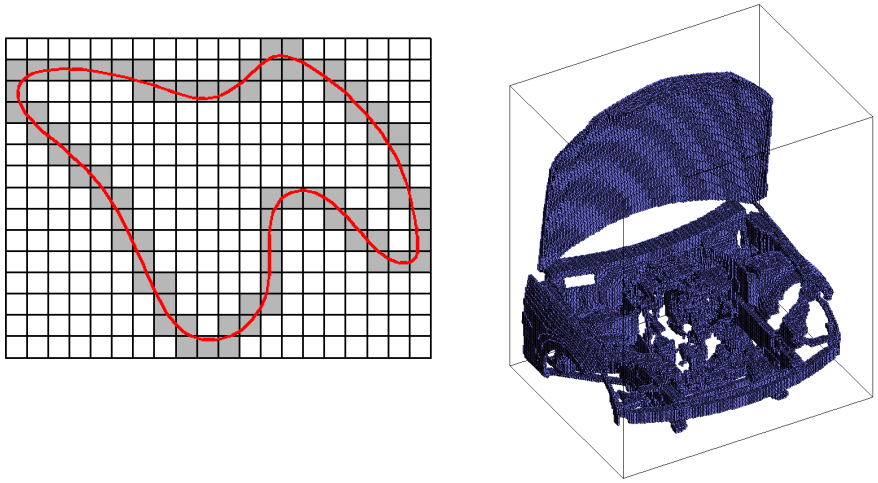
Due to lack of space, we will not diagram the results of all 12 test cases for every benchmark topic. Usually we will show for each benchmark topic one or two diagrams by example and explain in the text how the performance behaves in the other test cases. We will show tables with the results of all 12 test cases for most of our benchmark topics in the Appendix C.

It is our purpose that our algorithms find application in the daily work of the engineers. So we run our benchmarks on a consumer notebook with an Intel i7-4800MQ processor with a 2.70 GHz quad-core CPU. Our algorithms are implemented in C++ and parallelized by using OpenMP. Some algorithms in Sections 4.5.3 are parallelized also on the GPU using CUDA and benchmarked on different computers.

4.2. The Data Structure of the Grid

The most intuitive way to represent a uniform grid is by a 3d-array. In a 3d-array the access to one grid cell by its index is done in $O(1)$. Usually there are many grid cells

[1] Please note that the models of the *Engine* scenario were given in *mm* and the models in the *EngineBig* scenario were given in *cm*.



(a) (b)

Fig. 4.1: (a): 2d-example of occupied cells (gray) within a grid for a complex object (red). (b) 3d-example of occupied cells for the *Engine* bodyshell (the box around visualizes the grid extension) using a TI-grid with cell width 10. In both cases the grid is sparsely populated.

which are empty and only a few grid cells are occupied and store primitive references. Figure 4.1 shows a 2d- and a real life 3d-example. In both cases the grid is sparsely populated. In (a) a 20×15 grid with 300 cells has only 74 occupied grid cells. In (b) a $141 \times 178 \times 211$ grid with 5,295,678 cells has only 138,294 occupied grid cells, which are only 3% of all grid cells. For decreasing cell width w the number of all grid cells increases cubically. Because of that, a 3d-array data structure with constant access to a grid cell by its index has the drawback of a cubically increasing memory space to store all grid cells. It would be optimal to store mainly occupied grid cells. One simple solution to store only the occupied grid cells is to maintain the grid cells in a sorted list or in a simple unsorted list. The access on a grid cell by its index (denoted as look-up) in a sorted list is $O(\log n)$ and in an unsorted list $O(n)$, whereas n is the number of occupied grid cells. More advisable – and in fact the most common way for a memory efficient representation of occupied grid cells – is to use hash sets. For a suitable hash function the space of the hash map is still $O(n)$ and the access is in amortized constant time. In our problem statement the grid cells of each object are not changed during run time, so one can use perfect hashing. This means that a hash function can be used that maps the cells without collision to an array index. Thereby an $O(1)$ access can be guaranteed. A lot of research has been done on using hash maps to store occupied grid cells. An introduction to this issue and some related work how hash functions can be constructed is given by Ericson [5]. Some more publications concerning this issue were already introduced in the Related Work Section 1.3.

Our grid data structure consists of two individual parts:

- **The data structure to maintain the occupied grid cells:** An occupied cell is represented by its cell index and a reference to its cell content. Our grid query algorithms iterate over the occupied cells of the dynamic object's grids and look-

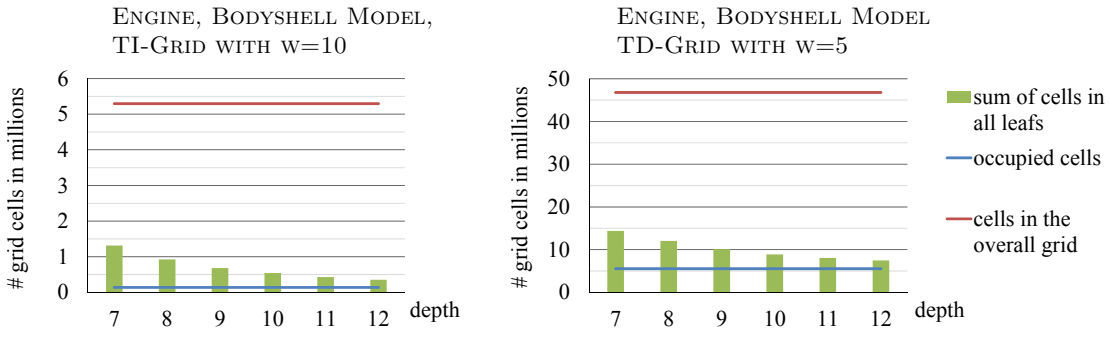


Fig. 4.2: Number of stored grid cells for varying depth of the hierarchy (green columns) in relation to the number of occupied grid cells (blue line) and the number of cells in a grid that contains the whole object (red line). The grids were calculated for the bodyshell in the *Engine* scenario with useful cell widths for a symmetric (left) and an asymmetric approach (right).

up for cells in the grids of the static object. So a grid of the static object must allow efficient grid queries, thus efficient cell look-ups. For a grid of the dynamic object this is not necessary. Thus, the data structure can be organized differently for the grids of the dynamic object and for the grids of the static object:

  - Dynamic grid: We use a simple array of occupied grid cells.
  - Static grid: We have implemented two variants to maintain the occupied cells. A classical variant using hash sets and a variant using 3d-arrays.

- **The data structure to maintain the cell content:** This data structure is based on a simple integer array. Each reference of an occupied cell points to the place where the cell content can be find. Each cell content consists of only one integer to denote the number of primitives and one integer per primitive reference.

In Figure 4.2 we show the number of occupied grid cells compared to the number of stored cells by using several 3d-arrays and compared with the number of grid cells in a single 3d-array that contains the whole object. Dependent on the depth of the hierarchical part of the combined data structure, the number of stored cells using 3d-arrays is close to the number of occupied cells and much smaller than the number of grid cells in a single 3d-array that contains the whole object.

All our benchmarks showed that using 3d-arrays is faster than using hash sets and that the number of stored cells never is a problem[2]. Refer to Figure 4.3 for two performance examples comparing 3d-arrays and hash sets. The benchmarks of our other test cases show a very similar behavior. The hash set we use is without perfect hashing. With perfect hashing the performance can probably be improved. But it cannot become noticeable faster than with 3d-arrays since using 3d-arrays can also be interpreted as a kind of perfect hashing. We decide to use 3d-arrays to maintain our grid cells.

[2]In fact, the number of occupied cells is not an issue – the number of primitive references for grids with small cell width is an issue. We will discuss this in Chapter 5.

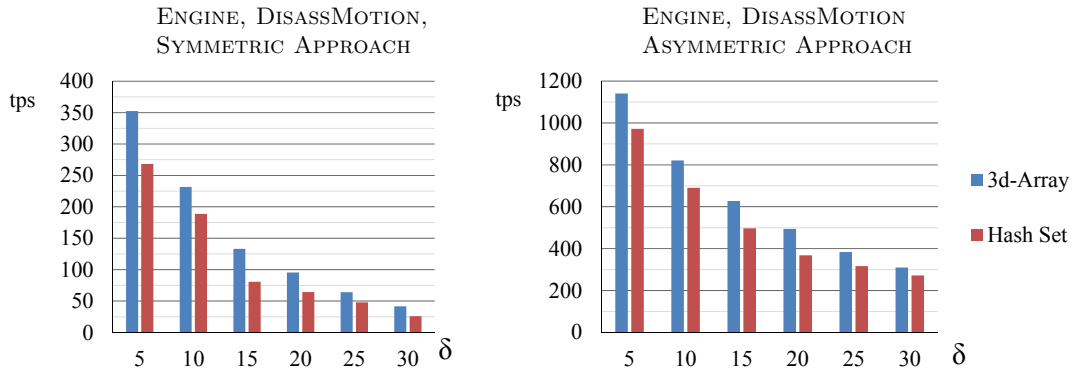


Fig. 4.3: Performance comparison of using 3d-arrays and hash sets to maintain the grid cells on the example of the *Engine* scenario with the transformation set *DisassMotion* for varying tolerance values.

4.3. The Data Structure of the Hierarchy

The advantage of the hierarchical part of our combined data structure is, beside the just seen possibility of using 3d-arrays for the grid, especially the limitation of the query cells. We implemented our combined data structure such that we are able to use different types of hierarchical data structures like different types of bounding volume hierarchies or multi-grids. An optimal hierarchical data structure is one with small and thigh fitting partitions such that it can be focused fast on the relevant leafs of the hierarchy. Another aspect is a fast test to decide which nodes are in conflict. Usually these two requirements are contradictory.

Our benchmarks showed that using BVHs of axis aligned bounding boxes (AABBs) as hierarchical data structure performs best compared with BVHs of orientated boxes (OBBs) and of spheres. Although the intersection or tolerance tests with spheres is faster than the ones for AABBs or OBBs, spheres fit very poor on subsets of cells that are aligned in a grid. In many applications OBBs are used as bounding volumes because they are tightest fitting. But the intersection or tolerance test for OBBs is more expensive than for AABBs. As our bounding volumes do not cover subsets of an

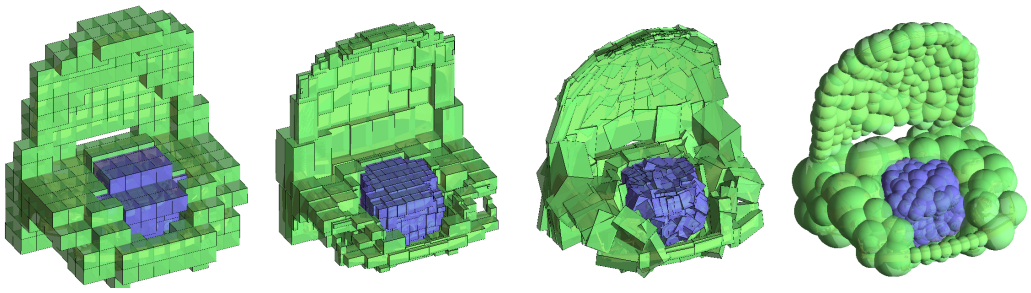


Fig. 4.4: Comparison of different hierarchical data structures on the example of the *Engine* scenario. From left to right: Multi-Grid, BVH of AABBs, BVH of OBBs, and BVH of spheres. The BVHs are all displayed at he same hierarchy level. The multi-grid is displayed on a level where the number of occupied cells in the level is similar to the number of bounding volumes of the displayed BVH levels.

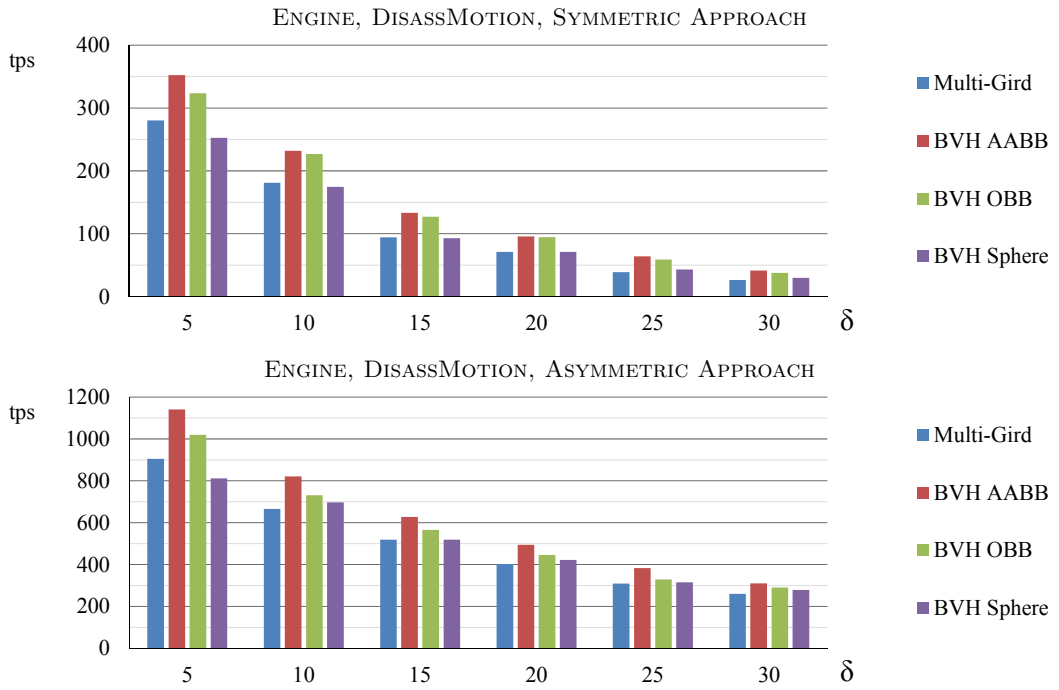


Fig. 4.5: Comparison of the performance for different hierarchical data structures on the example of the *Engine* scenario with the transformation set *DisassMotion* for grids with large cell width (top) and for a grids with small cell width (bottom).

arbitrary geometry but sets of axis orientated cubes, AABBs are also tight fitting. This explains why using AABBs is faster than using other bounding volumes. Figure 4.4 shows pictures of the three BVHs in order to give an impression on how the bounding volumes enclose the grid cells.

Our benchmarks showed also that BVHs with AABBs perform better than multi-grids. The test whether a node is in conflict is cheep for multi-grids. But the cells of the multi-grid usually cover much space without occupied grid cells (refer again to Figure 4.4). Thus, the multi-grid focuses not that fast on relevant parts. This causes that more tests whether a node is in conflict has to be executed and that the sum of selected leafs is larger than for BVHs with AABBs.

Thus, in summary, BVHs with AABBs perform best in our application cases. Figure 4.5 shows this behavior on the example of the *Engine* scenario for the transformation set *DisassMotion* using our symmetric and our asymmetric approach. For every δ and every hierarchical data structure we have calculated all tolerance violating triangles for the transformation set *DisassMotion* with varying hierarchy depth and display the fastest one in the diagram, respectively. Further benchmarks with other test cases show a very similar behavior. Thus, we decided always to use a BVH with AABBs as hierarchical data structure.

4.4. Parameters

> *In fact, for most applications although it is easy to pick bad parameter values it is equally easy to pick good ones.*
>
> C. Ericson, Real-Time Collision Detection, page 287 [5]

Our combined data structure is influenced by two parameters: The cell width w of the grid and the depth of the hierarchy. We will discuss these two parameters in the following. We ran our algorithms on several test cases of our benchmark data to calibrate for the optimal parameter setting and present some criteria how good parameters can be estimated.

4.4.1. Depth of the Hierarchy

The depth of the hierarchy influences the ratio of the hierarchy traversal and the grid query. As the hierarchy traversal is implemented for single core CPU and the grid query is parallelized, the depth of the hierarchy influences also the ratio of single core and multi core calculation.

Benchmark calibration: We ran our 12 test cases (refer to Table 4.1) with different tolerance values δ. Thereby we varied the cell width and the depth of the hierarchy. All our calibration results show the same behavior for varying depth of the hierarchy:

We observed that using a very flat hierarchy (e.g. a BVH with depth 1 to 4) usually results in a worse performance than using no hierarchical data structure. This is because the focus on relevant parts is not good enough and very often all leafs of the hierarchy must be considered. Thus, many grid cells of the dynamic grids are localized multiple times but never encounter occupied grid cells in a static grid. Then there is a range of moderate depth (e.g. a BVH with depth 5 to 17) where the performance is better than using no hierarchical data structure. Here the hierarchy is good enough to focus on the relevant parts. Thus, not that much grid cells have to be localized and the localized grid cells often encounter occupied grid cells in the static grids. A greater depth usually results in a too high degree on single core calculations and in too much multiple localizations per grid cell in a dynamic grid.

Recommendation: In summary, we recommend to set the depth of the hierarchy between 8 and 12, where a small depth is optimal for models with simple geometry (like the *Bunny* model) and a large depth is optimal for complex models (like the *Buddha* or the *EngineBig* models).

4.4.2. Cell Width of the Grids

The cell width is a very important factor of a grid. It influences the memory as well as the performance of a grid query:

- A too coarse cell width results in many primitives within one cell. This results in many pairs of primitives to be tested against each other in the grid query. The pre-selection of candidate primitives is not tight enough.

- A too fine cell width results in primitives which are referenced by many different cells. This results in many multiple primitive tests.

Cell width in the literature: For ray tracing applications the term grid density is often discussed in order to set the cell width of the grid. Refer for example to Lagae and Dutré [26] and references therein. The grid density is defined by $\rho = M/N$, where N is the number of objects that are referenced in the grid and M is the number of cells in the grid which is dependent on the cell width and the object extension. The value of the grid density ρ depends on the application case. In [26] it is suggested to be 4, in other work it is suggested to be between 1 and 10. For a defined grid density ρ the cell width can be computed. For collision detection, usually the cell width is set to be little larger than the largest primitive at any rotation in order to guarantee that each object overlaps at most 8 cells. Refer for example to Ericson [5] and to Section 3.3.1. However, both suggestions to set the cell width are made for special application cases, which are different from ours.

Benchmark calibration: A suitable cell width for our uniform grids depends on the size of the primitives and also on the tolerance value δ. We ran our 12 test cases (refer to Table 4.1) with different tolerance values δ. Thereby we varied the cell width and the depth of the hierarchy. We made this for our symmetric and for our asymmetric approach. Based on the obtained running times we analyzed the cell width ranges which achieved the best performances. Based on the ranges with good performance we were able to derive a common recommendation for setting the cell width in all benchmark scenarios.

If we would follow the suggestion in [5], we would choose the width as the sum of half the tolerance value δ plus the size of the largest primitive. We express the size l_{max} of the largest primitive by the largest edge length in the scenario (however, instead of the edge length other quantities are possible but behave similar). Choosing a cell width of $l_{max} + 1/2 \cdot \delta$ turned out as much too large, especially for our benchmarks from industrial applications since they contain very different sized triangles. But we can use this value as an upper bound for the cell width. So we have $w \leq l_{max} + 1/2 \cdot \delta$.

Very often the problem of different sized primitives is avoided by using multi-grids or octrees. But we desire to use a uniform grid as we want to parallelize over the grid cells as explained in the previous chapter. Thus, we have to handle different sized primitives. It is possible to split all primitives as long as they are almost equally sized. This will produce much more primitives and in case of tolerance violation much more primitive-primitive tolerance tests. Especially in the case of some very small primitives the number of total primitives may explode in case all primitives are split to the size of these smallest primitives. Moreover the number of grid cells will explode as they are set very small. Both will slow down the performance. But the size of the smallest primitive can give us a lower bound for the cell width. We express the size

of the smallest primitive by the smallest edge length l_{min} in the scenario and we have $w \geq l_{min}$.

Altogether we now have a very rough guess about the range where the appropriate cell widths can be found. We suggest to set the cell width w to a value in the interval $[l_{min}, l_{max} + 1/2 \cdot \delta]$. All our calibration results approved this bound. In the following we will tighten this estimation for our symmetric and for our asymmetric approach individually.

Benchmark calibration for the symmetric approach: All our calibration results showed that we can set the limits for the cell width for symmetric approaches more tightly to

$$l_{Q_{0.25}} \leq w \leq l_{Q_{0.75}} + \frac{1}{2}\ \delta\ , \tag{4.1}$$

where $l_{Q_{0.25}}$ is the 25th percentile and $l_{Q_{0.75}}$ is the 75th percentile of all edge lengths in the scenario. One can search for the optimum cell width within this interval for example by using the bisection method.

If we consider the cell width ranges with good performances in all our calibration results for the symmetric approach, we observed that

$$w = l_{median} + \frac{1}{10}\ \delta\ , \tag{4.2}$$

with l_{median} the median edge length in the scenario, is always inside these ranges with good performance and thus always a good choice to set the cell width.

Benchmark calibration for the asymmetric approach: All our calibration results showed that the optimal cell width for the asymmetric approach is significantly smaller than for the symmetric approach. We can give, based on our calibration results, a more precise lower and upper border for a good cell width by

$$l_{min} \leq w \leq l_{median} + \frac{1}{10}\ \delta\ . \tag{4.3}$$

A rough estimation for a useful cell width can be given as half of the optimal cell width in the symmetric approach:

$$w \approx \frac{1}{2}\ (l_{median} + \frac{1}{10}\ \delta)\ . \tag{4.4}$$

For complex objects (like in the *EngineBig* and in the *Buddha* scenarios) the optimal cell width is usually slightly larger than this approximation. So we weight the above approximation by a factor that describes the complexity of the scenario. Let κ be the factor that describes the complexity of the scenario. Then the cell width can be estimated with

$$w = \frac{1}{2}\ (l_{median} + \frac{1}{10}\ \delta)\ (1 + \frac{1}{100}\ \kappa)\ . \tag{4.5}$$

κ depends on the relation of the extension volume of the complex objects and the number of triangles. We will explain this in the Appendix A.5 more detailed and give the value of κ for each scenario. Again we consider the cell width ranges with good performances in all our calibration results, this time for the symmetric approach. We observed that the so calculated cell width w is always inside these ranges with good performance and thus always a good choice.

Recommendation: In summary, we suggest to set the cell width in the symmetric approach to $w = l_{median} + 0.1 \cdot \delta$ or to search for the optimal cell width by some calibration runs in the interval $[l_{Q_{0.25}}, l_{Q_{0.75}} + 1/2 \cdot \delta]$. For the cell width in the asymmetric approach we suggest to set $w = \frac{1}{2}\ (l_{median} + \frac{1}{10}\ \delta)\ (1 + \frac{1}{100}\ \kappa)$ or to search for the optimal cell width by some calibration runs in the interval $\left[l_{min}, l_{median} + \frac{1}{10}\ \delta\right]$.

4.5. The Symmetric Approach

Within this section we consider the performance of our symmetric approach that uses TI-grids to store the dynamic and to store the static object, respectively. We calculate for all 12 test cases all tolerance violating triangles in each step of the transformation set and compare the achieved number of tests per second (tps).

We compare the achieved performance of our symmetric approach with a standard BVH approach that is implemented by using OBBs as bounding volumes. Further we compare us with a BVH approach that was provided by Erbes [3]. Both BVH approaches are single core implementations.

The proximity queries presented by Erbes in [3] are mainly focused on the efficient calculation of many collision tests with a Boolean answer in parallel. Beside this, the software provided by Erbes also provides an algorithm that calculates all tolerance violating triangles. As his approach is an offline approach that calculates all transformation steps in parallel and outputs the results after all transformations are processed, we input not the whole transformation set at once. Instead, we input one transformation after the other in order to force his algorithm to perform an online calculation. The consequence is that there is no exploitation of parallel computing.

4.5.1. Performance of the Basic Symmetric Approach

First, we consider the basic symmetric approach. This means that we consider the symmetric approach without using core-triangles as discussed in Section 3.3.4.

In Figure 4.6 the performance of our symmetric approach is compared with the standard BVH approach and the single core variant of the BVH approach by Erbes for the *Engine* scenario and the transformation set *ColTolVerl* for varying tolerance values δ. Our approach is between 2 and 5 times faster than the standard BVH approach. The BVH approach by Erbes is well optimized and therefore significantly faster than the standard BVH approach (except for $\delta = 5$). Our symmetric approach is until a factor of 2.2 faster than the single core BVH approach.

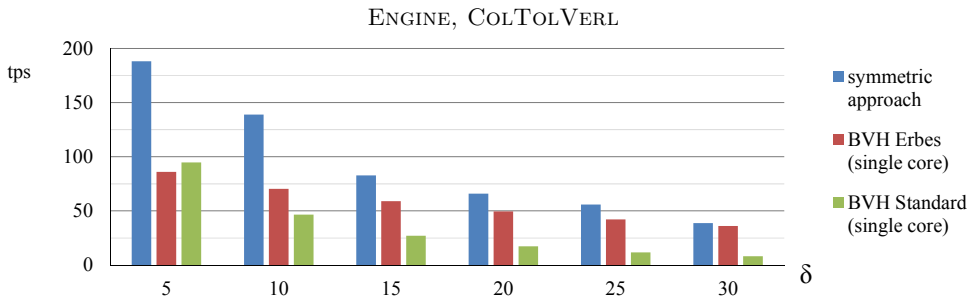


Fig. 4.6: Performance of the symmetric approach compared with a standard BVH approach and the approach by Erbes for the *Engine* scenario and the transformation set *ColTolVerl* for varying tolerance values δ.

The behavior is similar for the other 11 test cases. The symmetric approach is always significantly faster than the standard BVH implementation. Compared with the single core BVH approach of Erbes it is, depending on the test case and depending on the tolerance value, sometimes equal, in a few cases slower up to a factor of 0.8, and in all the other cases faster up to a factor of 2.5. In summary – over all test cases – our symmetric approach performs better than the single core variant of the BVH approach by Erbes. The complete results for all 12 tests cases are given in the Appendix C.

4.5.2. The Benefit of Using Core-Triangles

We have shown in Section 3.3.4 that core-triangles can only be used if $\delta \geq 2r$ with $r = 1/2\sqrt{3}w$. Using the empirical formula given by Equation 4.2 to estimate the cell width w in our scenarios, core-triangles exist in our scenarios only in the following cases:

- *Engine*: $\delta \geq 20.5$
- *Bunny*: $\delta \geq 0.0031$
- *EngineBig*: $\delta \geq 1.10$
- *Buddha*: $\delta \geq 0.00064$

Especially for the *Engine*, *Bunny*, and *Buddha* scenario the tolerance value for which we can have core-triangles is relatively large. In the *Buddha* scenario it is even larger than our considered tolerance range. Of course it is possible to use smaller cell widths to obtain core-triangles also for small tolerance values. However, our experiments showed that the cell width should not be chosen much smaller than the one that is suggested by the empirical formula given by Equation 4.2 (otherwise we will have too much multiple triangle tests to calculate, which slows down the performance). A slightly smaller cell width, however, is possible and often speeds up the calculation by using core-triangles.

We will analyse the benefit of using core-triangles on the example of the *Engine* scenario and the *EngineBig* scenario using the transformation set *ColTolVerl*, respectively. In Section 3.3.4 we have considered the relation between the tolerance value δ and the radius r of a grid cell. Further we have approximated by S_{core} the region that is considered to obtain core-triangles for the grid query of one dynamic cell. Table 4.4

Table 4.2: Relation of δ/r and the radius S_{core} in the example of the *Engine* scenario and the *EngineBig* scenario with cell widths chosen by the empirical formula.

	Engine						*EngineBig*					
δ	5	10	15	20	25	30	0.5	1.0	1.5	2.0	2.5	3.0
w	10.3	10.8	11.3	11.8	12.3	12.8	0.58	0.63	0.68	0.73	0.78	0.83
δ/r	<2	<2	<2	<2	2.35	2.71	<2	<2	2.55	3.16	3.70	4.17
radius S_{core}	0	0	0	0	3.70	7.83	0	0	0.32	0.74	1.15	1.56

shows the ratio δ/r for the two examples when the cell width is chosen by the empirical formula given in Equation 4.2. Further it shows the radius of S_{core}, respectively. Refer also to Table 3.1 in Section 3.3.4 in order to get an impression of the size of S_{core}.

The diagrams in Figure 4.7 display for the two examples the number of all tolerance violating triangles per transformation step and the number of tolerance violating triangles that can be detected by only marking core-triangles and without triangle-triangle tolerance tests. For smaller tolerance values than the one displayed in the diagrams there are no core-triangles since $\delta < 2r$. In the test case *Engine ColTolVerl* between 10% and 34% of all tolerance violating triangles can be detected by marking core-triangles. In the test case *EngineBig ColTolVerl* even between 27% and 48% of all tolerance violating triangles can be detected. Of course, the larger the quotient δ/r is, the more tolerance violating triangles can be detected by marking core-triangles without any triangle-triangle tolerance test.

The results in these two examples are theoretically good and show that the idea of marking core-triangles works well, even in those cases where δ/r is not much larger than 2 and S_{core} is small.

Next let us consider the number of all triangle-triangle tolerance tests that have to be calculated for the *ColTolVerl* transformation set in the *Engine* and in the *EngineBig* scenario. We calculate all tolerance violating triangles in all motion steps, once with our symmetric approach using core-triangles and once with our basic symmetric approach without using core-triangles. Interestingly, in both examples we can only save between 2% and 4% of all triangle-triangle tolerance tests by using core-triangles.

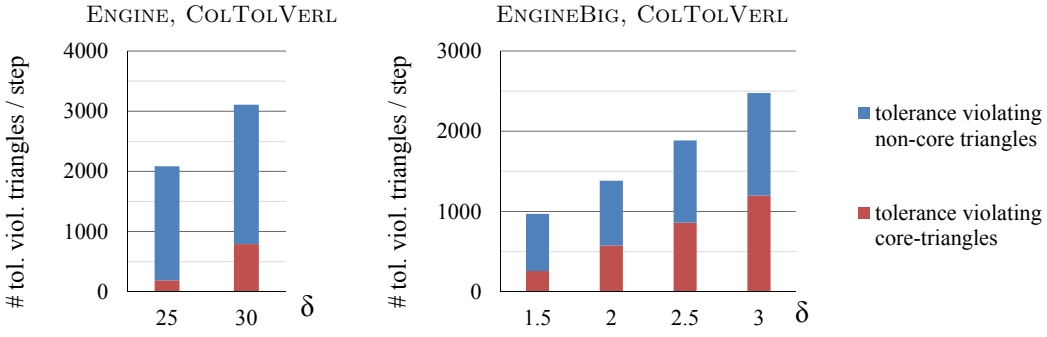


Fig. 4.7: Two examples for the portion of tolerance violating triangles (red+blue) that can be detected by just marking core-triangles (only red).

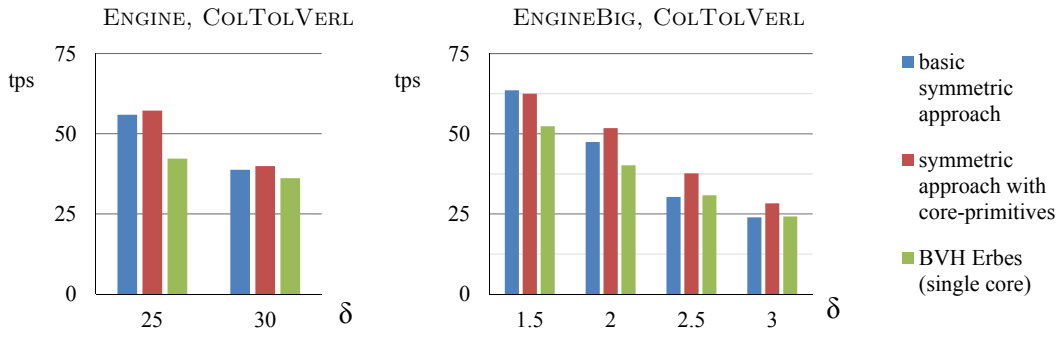


Fig. 4.8: Performance of the standard symmetric approach and the symmetric approach that uses core-triangles, compared with the BVH approach of Erbes.

Thus, marking core-triangles does not save many triangle-triangle tolerance tests – although many tolerance violating triangles can be found by marking core triangles. The reason for this behavior is that most of the triangle-triangle tolerance tests are spend on pairs of triangles that are not tolerance violating. Most tolerance violating pairs do not have to pass the triangle-triangle tolerance tests because they are already filtered out by our early out for known tolerance violations (refer to Section 3.3.5). Therefore marking core-triangles does not remarkably reduce the number of triangle-triangle tolerance tests.

We have benchmarked the performance of our symmetric approach that uses core-triangles on all our test cases and compared the performance with the one of the basic symmetric approach that does not use core-triangles and the single core BVH approach by Erbes. Figure 4.8 shows the achieved performance for the above already mentioned test cases. Although only a small portion on triangle-triangle tolerance test can be saved, the performance profits in most cases from using core-triangles.

In summary – over all 12 test cases – the performance of the symmetric approach that uses core-triangles is often slightly faster than the basic symmetric approach that does not use core-triangles. In some cases, especially when δ/r is small, the basic approach is faster. In average, over all 12 test cases and over all tolerance values, using core-triangles is about 7% faster. Refer to the Appendix C for all results in detail.

4.5.3. GPU Version of the Symmetric Approach

Until now we performed all calculations on the CPU and parallelized the grid query by using OpenMP. As we have many grid queries per transformation step, we have a lot of parallel tasks to calculate. The GPU is perfectly capable to calculate many similar tasks in parallel. Because of that we were interested how our algorithms perform by using the GPU. For the implementation of the GPU versions of our algorithms we used CUDA [40]. CUDA was developed by NVIDIA in order to execute code parts within e.g. a C++ program on a NVIDIA graphic card.

We will not give an introduction to CUDA and the special requirements for programming efficient CUDA code here. For an introduction please refer to the CUDA C Programming Guide [39].

The GPU Approach

We have implemented different GPU variants of our symmetric approach to calculate all tolerance violating triangles. They differ mainly by the algorithmic parts that are processed by CUDA and how a task for one thread is defined. This issue is very sensible as one bottleneck of CUDA is the access to the so called global memory and that warps of 32 threads should execute in parallel the same code on different data (SIMD = single instruction, multiple data).

Our best variant is a *concurrency hybrid approach*. We still calculate the traversal of the hierarchical part of our data structures on the CPU without any parallelization. The grid queries of all dynamic cells are performed on the CPU in parallel, except for the triangle-triangle tolerance tests. The result of the grid query of this approach is a list of cell pairs. Each pair consists of a dynamic query cell and a static candidate cell. The list of cell pairs is transferred to the GPU. The triangle-triangle tolerance tests are then performed on the GPU in parallel. One thread calculates all triangle tests that arise from one cell pair.

We denote our approach as hybrid approach as it calculates large parts of the whole algorithm on the CPU and large parts on the GPU. Further, we denote it as a concurrency approach, as we utilize the CPU and the GPU at the same time in parallel. A normal sequential process for one transformation step would process as follows:

1. First, the hierarchy traversal and the grid query, except for the triangle tests, are calculated on the CPU.
2. Then the list of cell pairs is transferred from the CPU memory to the GPU memory.
3. After that the GPU calculates the triangle tests.
4. Then the results, which identify all tolerance violating triangles, is transferred back from the GPU memory to the CPU memory.
5. Finally, the CPU outputs the result.

In such a sequential process the GPU is idle as long as the CPU is busy with step 2 and step 5. Further, the CPU is idle as long as the GPU is busy with step 3. In a concurrency process we try to keep both devices busy.

Assumed, the CPU has just finished step 1 of the 1st transformation. Then step 2 is executed and the GPU starts with step 3. Further assumed that the 2nd transformation is already available. While the GPU is busy with step 3 of the 1st transformation, the CPU can already start with step 1 of the 2nd transformation. As soon as the GPU has finished step 3 of the 1st transformation and the CPU has finished step 1 of the 2nd

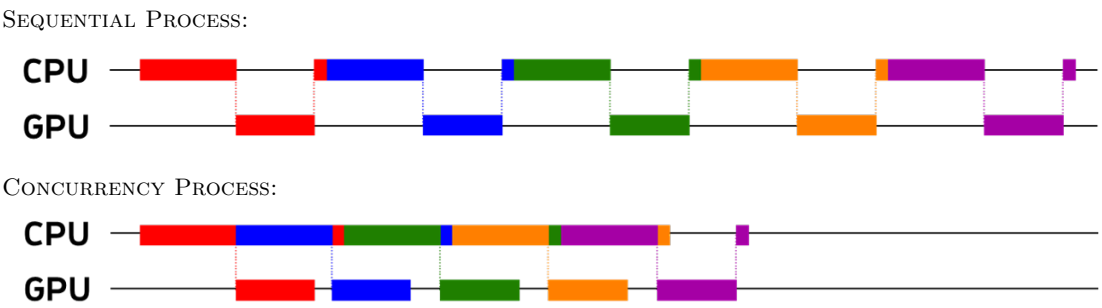


Fig. 4.9: The sequential and the concurrency process of the hybrid approaches. The color bars mark when which device is busy. The calculations for different transformations are visualized in different colors.

transformation, step 4 is executed for the 1st transformation and step 2 is executed for the 2nd transformation. After that the GPU can start with step 3 of the 2nd transformation and the CPU with the final step 5 of the 1st transformation. As soon as the CPU has finished step 5 of the 1st transformation, it starts with step 1 of the 3rd transformation. Refer to Figure 4.9 for a visualization of the sequential and the concurrency process.

In general the concurrency process works as follows:

- Assumed the CPU is busy with step 1 of the n-th transformation and the GPU is busy with step 3 of the (n-1)-th transformation.
- As soon as both are finished, step 4 of the (n-1)-th transformation and step 2 of the n-th transformation is executed.
- The GPU starts with step 3 of the n-th transformation and the CPU with step 5 of the (n-1)-th transformation and continuous with step 1 of the (n+1)-th transformation.

Usually the CPU and the GPU still have idle times. Depending on the length of step 3 and of step 1 either the GPU or the CPU has to wait for the other. But the total waiting times can be reduced significantly.

Benchmarks

Of course the performance of any algorithm finally depends on the hardware on which it is performed. A hybrid algorithm therefore depends on the used CPU as well as on the used GPU. A general statement about the speed-up one gets by using the hybrid version instead of the pure CPU version can never be made as the used hardware could be a combination of a *good or bad* CPU and a *good or bad* GPU. Therefore we benchmarked our concurrency hybrid version on different computers, which are configurated and denoted as follows:

C1: Intel(R) Core(TM) i7-740QM + GeForce GTX 460M (Fermi microarchitecture)

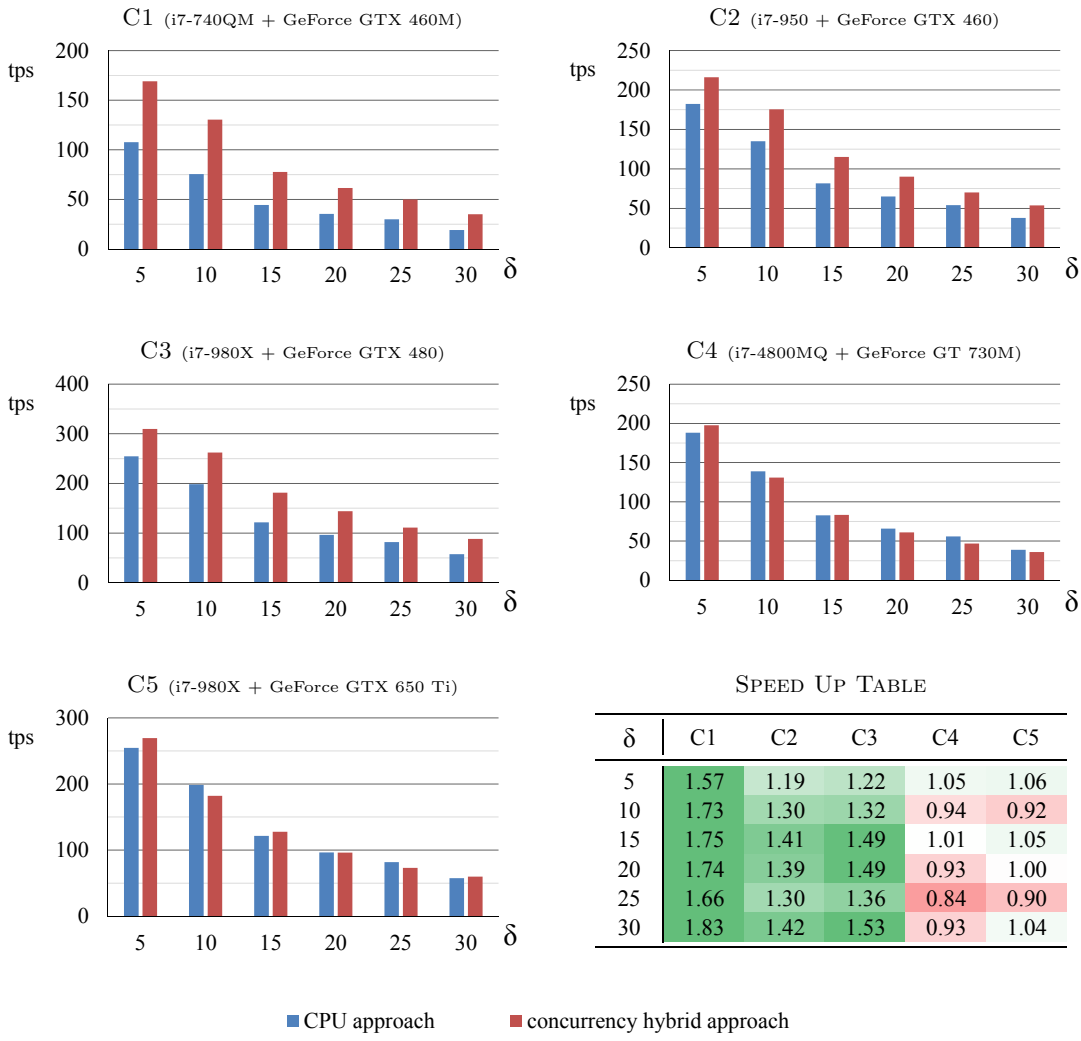
C2: Intel(R) Core(TM) i7-950 + GeForce GTX 460 (Fermi microarchitecture)

C3: Intel(R) Core(TM) i7-980X + GeForce GTX 480 (Fermi microarchitecture)

C4: Intel(R) Core(TM) i7-4800MQ + GeForce GT 730M (Kepler microarchitecture)

C5: Intel(R) Core(TM) i7-980X + GeForce GTX 650 Ti (Kepler microarchitecture)

Please note, the GPU implementation of our algorithms were an early part of this work. The implementation as well as our experience is related to the NVIDIA Fermi architecture. Refer to [41] for the Fermi whitepaper. Our implementation was developed on C1 and is thereby optimized for Fermi architecture. C2 and C3 also contain graphic cards with Fermi architecture. Nevertheless, we were also interested how our implementation performs on newer graphic cards. Thus, we used C4 and C5 for our benchmarks, too. Both graphic cards have the Kepler architecture.



Speed Up Table

δ	C1	C2	C3	C4	C5
5	1.57	1.19	1.22	1.05	1.06
10	1.73	1.30	1.32	0.94	0.92
15	1.75	1.41	1.49	1.01	1.05
20	1.74	1.39	1.49	0.93	1.00
25	1.66	1.30	1.36	0.84	0.90
30	1.83	1.42	1.53	0.93	1.04

Table 4.3: Comparison of the pure CPU approach (blue) with the concurrency hybrid approach (red) and a head map for the achieved speed-up.

We ran the test cases of the *Engine* scenario on all five computers and compared the pure CPU approach with the concurrency hybrid approach. The result of the *ColTolVerl* transformation set is given in Figure 4.3. For all results, please refer to the Appendix C.9.

One can see that the reached speed up by using the concurrency hybrid approach instead of the pure CPU approach behaves quiet different for different computers. Most significant is the different behavior of the computers with the Fermi architecture compared with those that have a Kepler architecture. The performance by running our GPU implementation directly on C4 and C5 is quiet disappointing as we have no or only a small speed-up or are even sometimes slower. Since we developed and optimized our algorithm for the Fermi architecture, the code runs not optimal on the Kepler architecture. But we are sure that it is possible to develop other GPU approaches to calculate all tolerance violating triangles per motion step for their hardware, such that we are able to achieve similar or even faster speed-ups as shown on C1, C2, and C3.

On C1, C2, and C3 we reach always a speed-up by using the concurrency hybrid approach instead of the pure CPU approach. The largest speed-up is reached on C1. Here the concurrency hybrid approach is until 1.8 times faster than the pure CPU approach.

In summary, we see very clearly that the performance is strongly hardware dependent. Further, of course, the algorithms can run only on CUDA capable graphic cards. Thus, we decided that it is not worth to develop highly optimized hardware dependent algorithms in order to achieve a speed up that is below 2, as it was in all our experiments. Because of that we have no longer examined algorithms for the GPU, but focused on our asymmetric approach on the CPU, which was clearly more promising.

4.6. The Asymmetric Approach

In this section we consider the performance of our asymmetric approach, where the grids of the static object are TD-grids and the grids of the dynamic object are TI-grids.

The symmetric approach spends relatively much time in querying candidate cells and candidate primitives for a dynamic grid cell. Especially for a large tolerance value and a small cell width of the grid there are many cells in the static grid to be queried for candidate triangles for one dynamic cell. Further, as neighboring cells have a similar content, a lot of candidate triangles are tested multiple times. Our asymmetric approach solves these two problems by preparing all possible candidate triangles in the cells of the static grid for an arbitrary querying dynamic grid cell in a pre-process. We expect the asymmetric approach to be faster than the symmetric approach. Refer also to Section 3.3.3.

Again, we have calculated all tolerance violating triangles in each transformation step for all 12 test cases and compared the number of tests per second (tps).

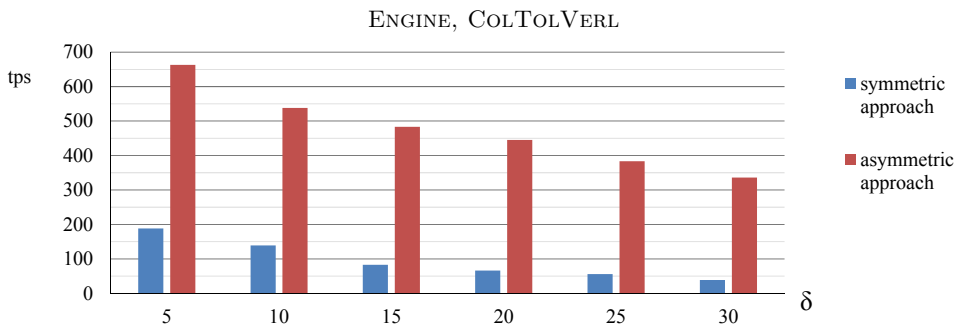


Fig. 4.10: Performance of the basic asymmetric approach compared with basic symmetric approach for the *Engine* scenario and the transformation set *ColTolVerl* for varying tolerance values δ.

4.6.1. Performance of the Basic Asymmetric Approach

At first we consider the basic asymmetric approach. This means that we consider the asymmetric approach without using core-triangles. We desire to prove our expectation that the asymmetric approach performs faster than the symmetric approach. So we compare the achieved performances of all our 12 test cases with the ones of the basic symmetric approach. The diagram in Figure 4.10 shows the performance of both approaches on the example of the *Engine* scenario and the transformation set *ColTolVerl* for varying tolerance values δ.

For the asymmetric approach we reach a significant speed up compared with the symmetric approach. For the smallest tolerance value in this example we already have a speed up of 3.5 and for the largest tolerance value we almost achieve a speed up factor of 9.

For our other 11 test cases the results are similar. We always reach a significant speed up with the asymmetric approach. Over all 12 tests it is between a factor of 2.5 and almost 10. Please refer to the Appendix C for all results.

4.6.2. The Benefit of using Core-Triangles

We already know that core-triangles can only be used if $\delta \geq 2r$ with $r = 1/2\sqrt{3}w$ (refer to Section 3.3.4). Using the empirical formula given by Equation 4.5 to estimate the cell width w for our scenarios, core-triangles exist in our scenarios only in the following cases:

- *Engine*: $\delta \geq 9.4$
- *Bunny*: $\delta \geq 0.0014$
- *EngineBig*: $\delta \geq 0.54$
- *Buddha*: $\delta \geq 0.00043$

Compared to the situation for the symmetric approach (refer to Section 4.5.2), the tolerance values for which we can have core-triangles is much smaller. This is because the cell width that is estimated by the empirical formula is much smaller for the asymmetric approach. So we can expect to have a larger amount of core-triangles for the asymmetric approach than for the symmetric approach.

Table 4.4: Relation of δ/r and the radius S_{core} in the example of the *Engine* scenario and the *EngineBig* scenario with cell widths chosen by the empirical formula.

	Engine						*EngineBig*					
δ	5	10	15	20	25	30	0.5	1.0	1.5	2.0	2.5	3.0
w	5.2	5.4	5.7	5.9	6.2	6.4	0.31	0.34	0.36	0.39	0.42	0.45
δ/r	<2	2.12	3.04	3.89	4.66	5.37	<2	3.42	4.75	5.90	6.90	7.77
radius S_{core}	0	0.6	5.1	9.7	14.3	18.8	0	0.42	0.87	1.32	1.77	2.23

Again we will analyse the benefit of using core-triangles on the example of the *Engine* scenario and the *EngineBig* scenario using the transformation set *ColTolVerl* respectively. In Section 3.3.4 we have considered the relation between the tolerance value δ and the radius r of a grid cell. Further we have approximated by S_{core} the region that is considered to obtain core-triangles for the grid query with one dynamic cell. Table 4.4 shows the ratio δ/r for the two examples when the cell width is chosen by the empirical formula given in Equation 4.2. Further it shows the radius of S_{core} respectively. Refer also to Table 3.1 in Section 3.3.4 in order to get an impression of the size of S_{core}.

The diagrams in Figure 4.11 display for the two examples the number of all tolerance violating triangles per transformation step and the number of tolerance violating triangles that can be detected by just marking core-triangles without computing any triangle-triangle tolerance tests. For smaller tolerance values than the one displayed in the diagrams there are no core-triangles since $\delta < 2r$.

A very large portion of tolerance violating triangles can be detected by marking core-triangles. In the test case *Engine ColTolVerl* the percentage is between 50% and 76%

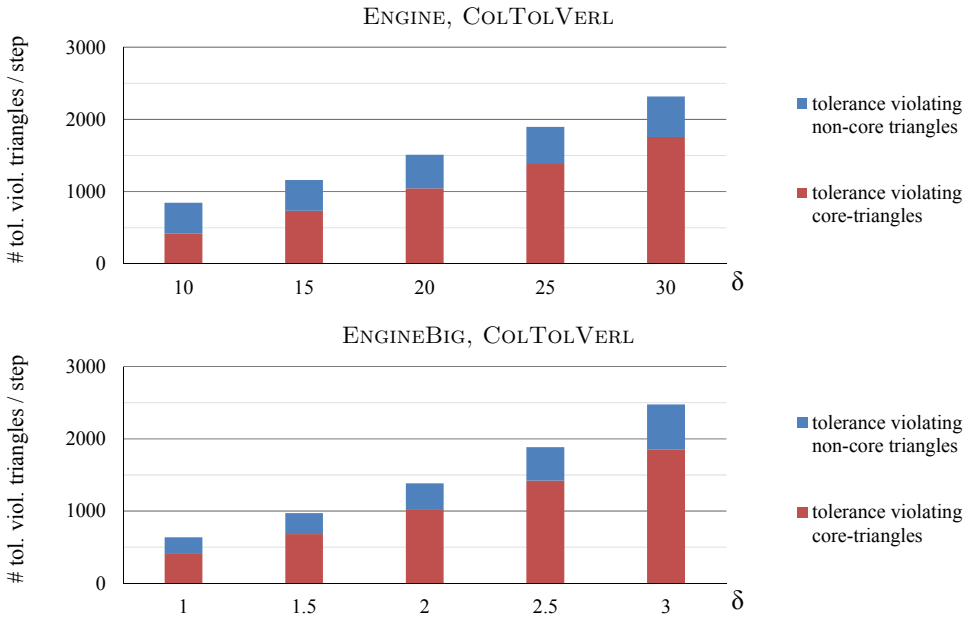


Fig. 4.11: Two examples for the portion of tolerance violating triangles (red+blue) that can be detected by just marking core-triangles (only red).

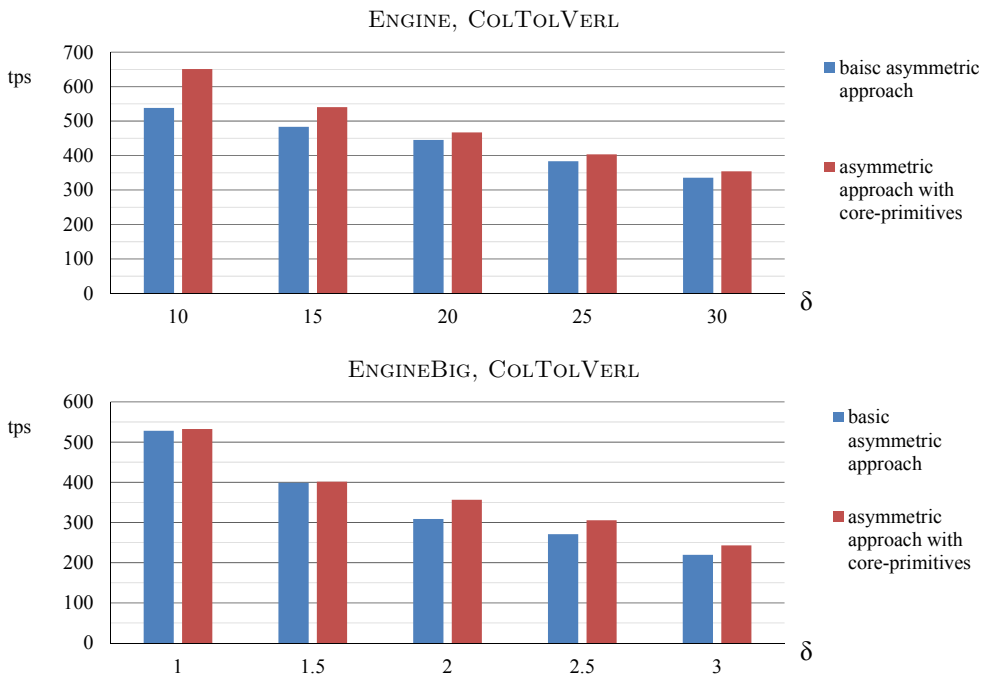


Fig. 4.12: Performance of the standard asymmetric approach and the asymmetric approach that uses core-triangles.

and in the test case *EngineBig ColTolVerl* it is between 64% and 75%. Especially for the large tolerance values most tolerance violating triangles are detected by marking core-triangles. More precisely, about 3/4 of all tolerance violating triangles are detected by marking core-triangles.

Also for our asymmetric approach the idea of marking core-triangles works theoretically very well. It even works much better than for the symmetric approach. However, the number of saved triangle-triangle tolerance tests due to marking core-triangles is again not that high – although larger than for the symmetric approach. We calculated all tolerance violating triangles in all motion steps once with our asymmetric approach by using core-triangles and once with our basic asymmetric approach that does not use core-triangles. In both examples we can save between 5% and 11% of all triangle-triangle tolerance tests by using core-triangles. It can be followed that (considering the large tolerance values) that 90% of all triangle tests are spent to find the remaining 1/4 of tolerance violating non-core-triangles. The reason is again that most triangle-triangle tolerance tests are spent on testing non-tolerance violating triangles pairs, which is caused by our early out for known tolerance violations (refer to Section 3.3.5 and also to Section 4.5.2).

We have benchmarked our asymmetric approach that uses core-triangles on all our 12 test cases and compared the performance with the one of the basic asymmetric approach that does not use core-triangles. Figure 4.12 shows the achieved performance for the two test cases *Engine ColTolVerl* and *EngineBig ColTolVerl*. The performance always profits from using core-triangles. We reach in these two examples a speed up to

a factor of 1.2.

Over all of our 12 test cases there are very few situations (7 out of 63) where the performance with marking core-triangles is slightly slower than without marking core-triangles. At this point we want to note that the implementation that is able to mark core-triangles is more complex than the basic implementation. For example, we have two parallel sections and between a barrier where threads have to wait for each other. Further we have to access grid cells that contain non-core triangles several times. Finally, the BVH traversal is more complex, since we output the colliding leaf pairs in a sorted order. Thus, the step of marking core-triangles has some cost overhead that must be first regained until we can profit. In almost all of our test cases we are able to regain the overhead of the more complex algorithm. In most cases we even achieve a speed-up. Please refer to the Appendix C for all our result in detail.

4.6.3. Comparison with an Offline Approach

Finally we want to compare our achieved results with results from related work. We have already noted in the beginning that our problem statement, the online calculation of all tolerance violating triangles per transformation step, is not yet considered by other researchers. A similar but offline problem was considered by Erbes in [3]. His work is mainly focused on efficient collision and tolerance tests with a Boolean answer. But his software also provides the opportunity for computing extended outputs like all tolerance violating triangles per transformation step. He calculates the tolerance tests for several transformations of the dynamic object in parallel and outputs all tolerance violating triangles per transformation step after all transformations are calculated. He uses a BVH with OBBs as bounding volumes. His implementation is realized like ours in C++ using OpenMP.

An offline approach has a big advantage in the distribution of the parallel tasks compared to an online approach. For a large set of transformations, each transformation trivially defines one task. So the set of all tolerance violating triangles can be calculated for all transformations in parallel. The performance of an offline approach depends on the number of transformation steps that can be calculated in parallel. Figure 4.13 gives an impression of that dependency. For the shown diagram we have used the *Engine ColTolVerl* test case and run the offline approach by Erbes to calculate all tolerance violating triangles for each of the 10,000 transformation steps. In order to show the dependency on the number of transformations that are calculated in parallel, we divide the 10,000 transformations into n packages of $10,000/n$ transformations and calculate one packages after the other by the approach of Erbes. The running times of all n packages is then summarized to the running time of all 10,000 transformation steps. It is obvious that the performance increases with decreasing n, thus, with the number of transformations that can be calculated in parallel.

In an online approach one has to calculate one transformation after the other. So one has to define parallel tasks within the calculation of only one transformation step. In our approach we define one grid query as one parallel task. So the tasks in an online

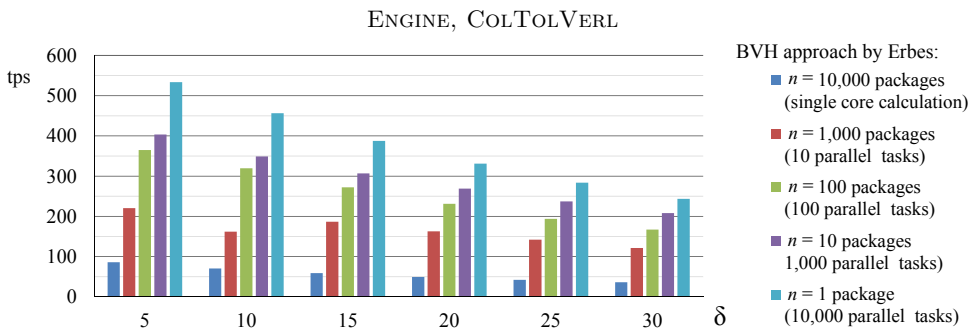


Fig. 4.13: Example of the dependency of an online approach on the number of transformation steps. The set of transformations *ColTolVerl* is divided to n packages of size $10,000/n$ and all n packages are calculated separately by the offline approach.

approach are much finer grained and have a much higher organizational effort. Further, as the tasks are not naturally given, this leads usually to some additional effort. In our case we have additional effort due to multiple calculations of some triangle-triangle tolerance tests.

Nevertheless, we compare us with the offline approach by Erbes and we will show that we are able to achieve the same or even in most cases a better performance with our online approach.

The diagram in Figure 4.14 compares the performance of our asymmetric approach with core-primitives and the offline approach by Erbes. Although we calculate our result online, we are even faster by a factor between 1.2 to 1.5 than the offline approach.

In our other 11 test cases we achieve similar good results. Our online approach always achieves comparable results with the offline approach. In all cases, except for two tolerance values in two test cases, we are even faster than the offline approach. In Figure 4.15 we summarize the performance of our online approach and the offline approach by Erbes for all 12 test cases. The respective speed ups are given, too. Please refer also to the Appendix C for all results in detail.

We are able to achieve these good results because our asymmetric approach is highly optimized to the task of obtaining all tolerance violating triangles for one given trans-

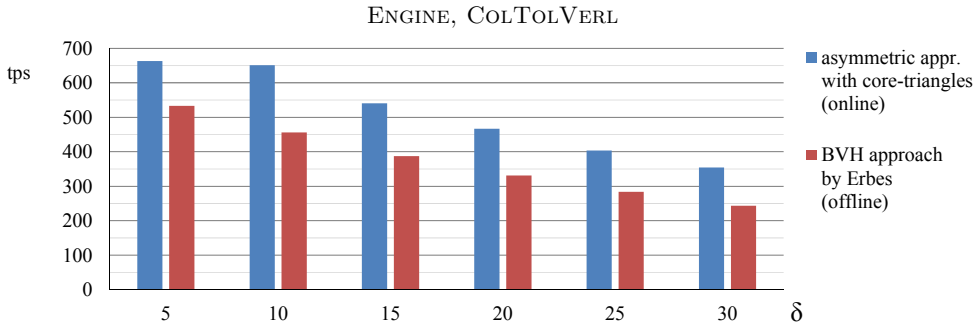


Fig. 4.14: Performance of our online approach and the offline approach by Erbes.

formation. The key-point is the design of the data structures – namely using TD-grids to store the static object. Since each cell in the TD-grid references all candidate triangles and core-triangles that are necessary for the cell query of an arbitrary dynamic grid cell, the performance of the grid query is very fast. This large performance benefit is achieved at the expense of a higher memory consumption of a TD-grid. The memory consumption and methods to compress the memory consumption of a TD-grid is the issue of the next chapter.

	Engine								
	ColTolVerl			NoColTolVerl			DisassMotion		
δ	BVH Erbes (offline)	asym. appr. (online)	speed-up	BVH Erbes (offline)	asym. appr. (online)	speed-up	BVH Erbes (offline)	asym. appr. (online)	speed-up
5	533	663	1.24	1,650	2,003	1.21	521	1,141	2.19
10	456	651	1.43	1,302	1,316	1.01	418	805	1.93
15	387	541	1.40	1,048	1,094	1.04	330	606	1.84
20	331	467	1.41	839	903	1.08	268	492	1.84
25	284	404	1.42	679	769	1.13	207	393	1.90
30	243	354	1.46	556	669	1.20	165	326	1.97

	EngineBig								
	ColTolVerl			NoColTolVerl			DisassMotion		
δ	BVH Erbes (offline)	asym. appr. (online)	speed-up	BVH Erbes (offline)	asym. appr. (online)	speed-up	BVH Erbes (offline)	asym. appr. (online)	speed-up
0.5	353	691	1.96	571	1,187	2.08	160	307	1.92
1.0	293	532	1.82	456	826	1.81	126	227	1.81
1.5	251	402	1.60	371	593	1.60	99	168	1.70
2.0	204	357	1.75	296	483	1.63	77	135	1.75
2.5	167	306	1.83	241	401	1.66	58	104	1.79
3.0	137	243	1.77	195	329	1.68	45	76	1.70

	Bunny								
	ColTolVerl			NoColTolVerl			ExtremeMotion		
δ	BVH Erbes (offline)	asym. appr. (online)	speed-up	BVH Erbes (offline)	asym. appr. (online)	speed-up	BVH Erbes (offline)	asym. appr. (online)	speed-up
0.001	1,109	942	0.85	1,844	1,318	0.71	177	182	1.03
0.002	809	788	0.97	1,161	1,130	0.97	126	154	1.22
0.003	593	629	1.06	777	867	1.12	91	118	1.29
0.004	433	512	1.18	565	697	1.23	68	96	1.42

	Buddha								
	ColTolVerl			NoColTolVerl			ExtremeMotion		
δ	BVH Erbes (offline)	asym. appr. (online)	speed-up	BVH Erbes (offline)	asym. appr. (online)	speed-up	BVH Erbes (offline)	asym. appr. (online)	speed-up
0.0002	252	444	1.76	279	534	1.91	117	144	1.23
0.0003	242	400	1.65	268	482	1.80	107	135	1.26
0.0004	230	393	1.71	255	457	1.79	98	118	1.20
0.0005	219	379	1.73	244	426	1.75	89	113	1.27
0.0006	208	349	1.68	230	385	1.67	81	101	1.24

Fig. 4.15: Overview over the performance of our online approach and the offline approach by Erbes in all 12 test cases.

5. Shrubs: The Compressed TD-Grid

In this chapter we present an approach to compress the memory consumption of TD-grids. Especially for large tolerance values and for a small cell width, the memory consumption of our TD-grid is high. We call our therefor developed data structure *shrubs*.

We will first analyze the problem of the high memory consumption of TD-grids and then introduce the idea of shrubs in Section 5.1. Details and definitions of shrubs are given in Section 5.2. In Section 5.3 and in Section 5.4 we present two completely different strategies to construct shrubs. Implementation details with regards to the total memory consumption of the compressed TD-grid are discussed in Section 5.5. Our results are presented and analyzed in Section 5.6.

5.1. Motivation and Idea

We showed in the previous chapter that for TD-grids usually the cell width is small compared to the tolerance value. A small cell width has advantages. These are:

- With a small cell width the selection of candidate primitives is more precise. Thus, we have less false positives.
- With a small cell width we detect more core-primitives. Thus, more primitives can be marked as tolerance violating without executing primitive tests.

On the other hand, we can observe an increasing memory consumption for a decreasing cell width, or likewise for an increasing tolerance value δ. This is especially notable in the case the static grid is a TD-grid.

Problem Analysis

A Cell's Zone of Influence

The number of primitives that are referenced by a cell of a TI-grid depends only on the geometry of the object and on the cell width w. The number of primitives that are referenced by a cell of a TD-grid depends on the geometry and on the cell width w, too – but additional on the tolerance value δ. We defined in Section 3.3.3 that a cell $\mathcal{X}$ in a TD-grid references the primitives that intersect $\mathcal{X} \oplus S_{r+\delta}$. A cell $\mathcal{Y}$ in a TI-grid references only the primitives that intersect $\mathcal{Y}$ itself. We will denote in the following $\mathcal{X} \oplus S_{r+\delta}$ and $\mathcal{Y}$ as the *zone of influence* of $\mathcal{X}$ and $\mathcal{Y}$, respectively.

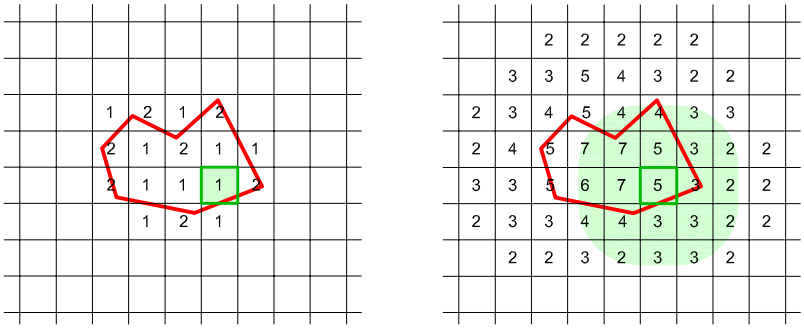


Fig. 5.1: Number of references per cell that are stored for a 2d-polygon in a TI-grid (a) and in a TD-grid (b). In summary there are 17 occupied cells and 24 references to store in (a) and 54 occupied cells and 176 references to store in (b). One cell and its zone of influence is highlighted by example.

In Figure 5.1 a 2d-polygon with seven segments is shown in a TI-grid and in a TD-grid with the same cell width. In (a) a TI-grid is used. 17 cells are occupied and in summary 24 segment-references have to be stored. In (b) a TD-grid is used. 54 cells are occupied and in summary 176 segment-references have to be stored. In both pictures one cell and its zone of influence is highlighted by example. It is obvious that the zone of influence in a TD-grid is much larger than in a TI-grid. This causes the considerable larger memory consumption of a TD-grid compared with a TI-grid.

The volume of the zone of influence of a cell $\mathcal{Y}$ in a TI-grid is equal to the volume of the cell and given as $V(\mathcal{Y}) = w^3$. The volume of the zone of influence of a cell $\mathcal{X}$ in a TD-grid is the union of the volume of eight 1/8 spheres, twelve 1/4 cylinders, six cuboids and one cube. It is given as $V(\mathcal{X} \oplus S_{r+\delta}) = 4/3\,\pi\,(\delta + r)^3 + 3\,\pi\,(\delta + r)^2\,w + 6\,(\delta + r)\,w^2 + w^3$ with $r = 1/2\sqrt{3}w$. The zone of influence of a cell in a TI-grid increases cubically in w, whereas the zone of influence of a cell in a TD-grid increases cubically in w and δ.

The Tables 5.1 and 5.2 show this behavior. In Table 5.1 some values of $V(\mathcal{Y})$ and $V(\mathcal{X} \oplus S_{r+\delta})$ are shown for varying δ and w. As expected, the volume rises quickly for increasing δ and w. Table 5.2 shows $V(\mathcal{X} \oplus S_{r+\delta})$ in relation to $V(\mathcal{Y})$ as ratio $V(\mathcal{X} \oplus S_{r+\delta})/V(\mathcal{Y})$ for equal cell width. Assumed, we have a grid of size $60 \times 60 \times 60$. If we use a cell width of $w = 3$, we have $20 \times 20 \times 20 = 8,000$ grid cells. The summarized volume of the zone of influence over all cells in a TI-grid of this size is $8,000 \times 27 = 216,000$. In a TD-grid with the same cell width and for the tolerance value $\delta = 5$ it is $8,000 \times 3,907 = 31,255,819$. In this case, it is 145 times larger for the TD-grid than for the TI-grid. This is exactly the factor given in the third row of the last column in Table 5.2.

Benchmark Example

As the summarized volume over the zone of influence increases, the number of primitives that possibly intersect a zone of influence increases and therefore the number of references in a cell increases. We will consider this issue for the example of one of our benchmark models, the front part of a car's bodyshell of the *Engine* scenario (Appendix A.1). In the diagrams of Figure 5.2 we compare for varying cell width TI-grids and TD-grids of the bodyshell. We always consider the TD-grid for the fixed tolerance

w	TI-grid	TD-grid $\delta = 0$	TD-grid $\delta = 1$	TD-grid $\delta = 2$	TD-grid $\delta = 3$	TD-grid $\delta = 4$	TD-grid $\delta = 5$
1	1	16	72	194	407	736	1,206
2	8	128	300	578	988	1,554	2,302
3	27	432	782	1,280	1,950	2,817	3,907
4	64	1,023	1,616	2,397	3,391	4,623	6,117
5	125	1,998	2,898	4,025	5,406	7,066	9,029

Table 5.1: The volume of the zone of influence of one grid cell in a TI-grid or in a TD-grid for varying tolerance value δ and cell width w.

w	TI-grid	TD-grid $\delta = 0$	TD-grid $\delta = 1$	TD-grid $\delta = 2$	TD-grid $\delta = 3$	TD-grid $\delta = 4$	TD-grid $\delta = 5$
1	1	16	72	194	407	736	1,206
2	1	16	37	72	123	194	288
3	1	16	29	47	72	104	145
4	1	16	25	37	53	72	96
5	1	16	23	32	43	57	72

Table 5.2: Relative consideration of the values in Table 5.1. The volume of the zone of influence of a TD-grid divided by the volume of the zone of influence of a TI-grid of the same cell width.

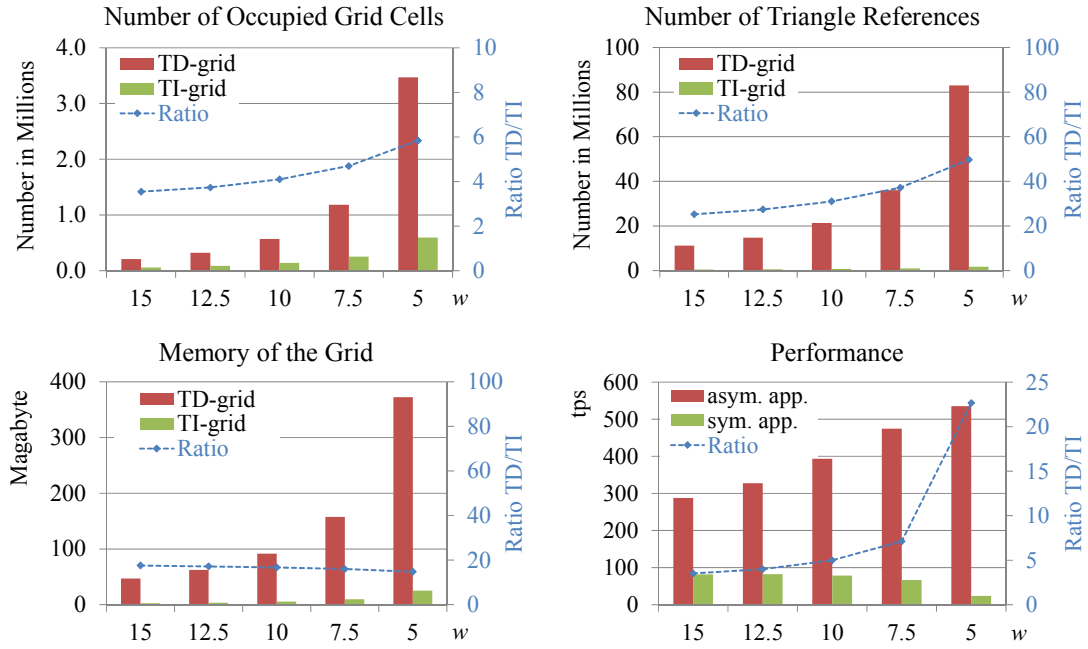


Fig. 5.2: Analysis of the *Engine* benchmark for varying cell width and for $\delta = 15$. Compared are a TD-grid (red) and a TI-grid (green) for the bodyshell. The blue curve displays the ratio between the values of the TD-grid and of the TI-grid. For the lower right diagram the transformation set *ColTolVerl* (Appendix A.1.1) is used.

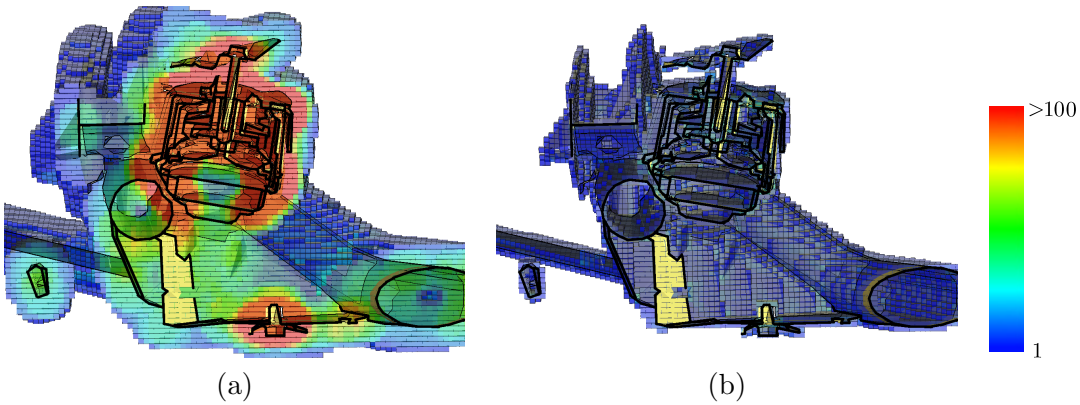


Fig. 5.3: Number of references per cell in a TD-grid (left) and in a TI-grid (right) shown with color-codes in a cross-section of a part of the bodyshell in the *Engine* scenario with cell width 5 and $\delta = 15$. The more red a cell is, the more triangle references are stored (the maximum in this picture is 835 references per cell).

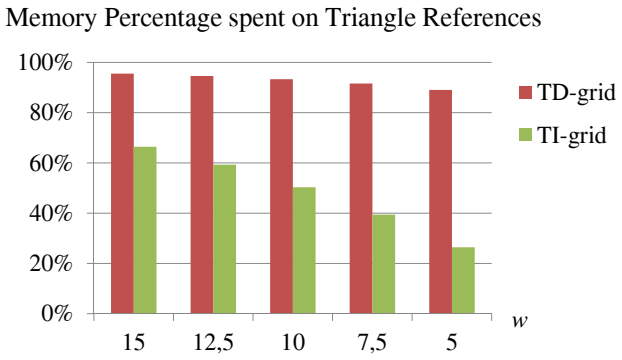


Fig. 5.4: The percentage on the total grid memory of a TD-grid (red) and a TI-grid (green) that is spent to store the triangle references. The memory is measured in the *Engine* benchmark for the bodyshell with $\delta = 15$ for varying cell width.

value $\delta = 15$. For larger δ the differences are more, for smaller δ the differences are less significant.

The upper left diagram in Figure 5.2 compares the number of occupied grid cells. It shows that the number of occupied grid cells for the TD-grid is always greater than for the TI-grid and increases with decreasing cell width. This is clear as we can consider the occupied grid cells of the TD-grid as a voxelization of the bodyshell's surface and the occupied grid cells in a TD-grid as a voxelization of the bodyshell's δ-offset.

The upper right diagram in Figure 5.2 shows the increasing number of triangle references for a decreasing cell width. The number of triangle references is significantly larger for the TD-grid than for the TI-grid, namely 25 times for $w = 15$ and even 50 times for $w = 5$. This is caused by the extremely increasing zone of influence as explained already above. Refer also to Figure 5.3, where the number of triangle references per cell is visualized by color codes. Each single cell in the TD-grid has much

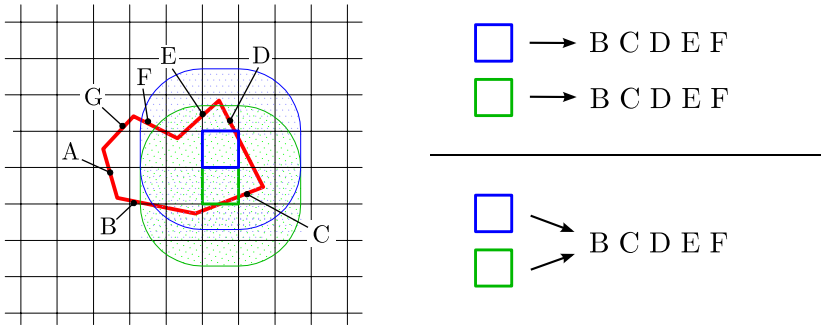


Fig. 5.5: Overlapping zone of influence of two neighboring cells with equal cell content (left) and the content of the two cells stored for each cell respectively (upper right) and stored only once (lower right).

more triangle references than a single cell in the TI-grid. This, together with the larger number of occupied cells, causes the high memory consumption of the TD-grid.

In the lower left diagram in Figure 5.2 the memory consumption of the grid data structures are displayed (how the memory consumption is composed will be explained later in Section 5.5). The diagram shows that the memory consumption of the TD-grid is between about 15 to 18 times the memory consumption of the TI-grid for all displayed cell widths.

Finally, the lower right diagram in Figure 5.2 displays the performance that is reached in the *Engine* scenario for the motion *ColTolVerl* (Appendix A.1.1) for our symmetric approach, which uses TI-grids to store the static object, and our asymmetric approach, which uses TD-grids to store the static object. It is obvious that the asymmetric approach is significantly faster than the symmetric approach, namely by a factor of 6.5 for the best displayed cell width, receptively (which is $w = 5$ for the asymmetric and $w = 12.5$ for the symmetric approach). So the better performance of the asymmetric approach is reached on the expense of a higher memory consumption.

The largest percentage of the higher memory consumption of the TD-grid is caused by the huge number of referenced primitives per cell. Figure 5.4 shows the percentage of the memory consumption that is spent to store the primitive references. It is interesting to see the difference between the TD-grid and the TI-grid. For the TD-grid almost the whole memory is used to store the primitive references. In the rest of this chapter we present a method for the lossless compression of the number of referenced primitives and thereby of the lossless compression of the memory consumption of a TD-grid.

The Idea to Compress the Number of Referenced Primitives

As the zones of influence of two neighboring cells overlap, the content of two neighboring cells probably overlaps, too. For example, the zones of influence of the blue and green marked cells in Figure 5.5 overlap and both cells reference the segments B, C, D, E and F, thus, their content is actually equal. A first idea to compress the number of referenced primitives is, to store the content of cells with equal content only once

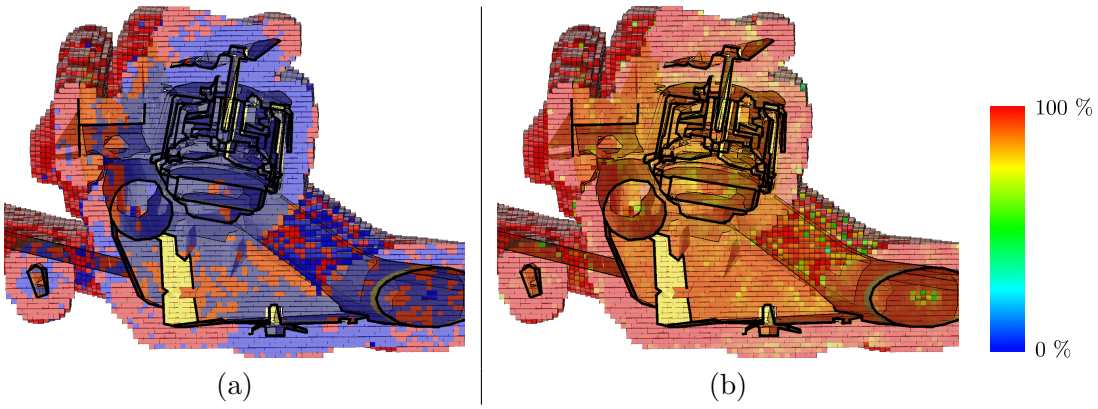


Fig. 5.6: Cells with at least one neighbor cell that has an identical cell content (red) and cells with solely neighbor cells that have a different content (blue) (a). Color code for the visualization of the similarity between a cell and its most similar neighbor cell (b). For the shown cross-section of a part of the static object in the *Engine* scenario we use a cell width of 5 and set $\delta = 15$.

and each of these cells references the common content (refer to the right-hand side of Figure 5.5).

In Figure 5.6 (a) cells which have at least one direct neighbor with identical cell content are visualized in red color. These red cells are cells near the boundary of all occupied grid cells. From Figure 5.3 (a) we know that the cells near the boundary have a relatively small number of referenced primitives. The cells with a large number of primitive references (refer again to Figure 5.3 (a)) are cells that have no direct neighbor cell with equal cell content (refer to Figure 5.6 (a)). However, cells with a large number of referenced primitives are especially interesting for data compression. Although these cells often have no direct neighbor cell with exactly the same content, there is usually a direct neighbor cell with a similar cell content. In Figure 5.6 (b) a color code visualization for the content similarity is given. We calculate the similarity for a cell $\mathcal{X}$ with its first ring neighbor cells $\mathcal{N}_i$ with $i \in [0, 25]$ as

$$\max_i \left\{ \frac{2\,|\mathfrak{O}_\mathcal{X} \cap \mathfrak{O}_{\mathcal{N}_i}|}{|\mathfrak{O}_\mathcal{X}| + |\mathfrak{O}_{\mathcal{N}_i}|} \right\} \cdot 100\% \; .$$

Thereby $|\mathfrak{O}_\mathcal{X}|$ and $|\mathfrak{O}_{\mathcal{N}_i}|$ is the number of primitives of an object $\mathfrak{O}$ that are referenced by the cell $\mathcal{X}$ and by the neighbor cell $\mathcal{N}_i$. $|\mathfrak{O}_\mathcal{X} \cap \mathfrak{O}_{\mathcal{N}_i}|$ is the number of primitives that are referenced by $\mathcal{X}$ as well as by the neighbor cell $\mathcal{N}_i$. Figure 5.6 (b) shows that the similarity between all cells is very high.

The Principal Idea of Shrubs

Our idea to reduce the memory consumption of a TD-grid is to compress the number of referenced primitives by utilizing the similarity of the cell content of neighboring cells. We store the content of all grid cells in several tree data structures. Each cell is associated with one node. The particular issue is that for two or more nodes with similar content a parent node is created that stores the common content of the nodes and each of the nodes stores its individual remaining content. Generally, not all cells have a common content. Thus, there is no common root for all cells and so our data structure is a forest of trees. As we usually have lots of small trees for our benchmarked

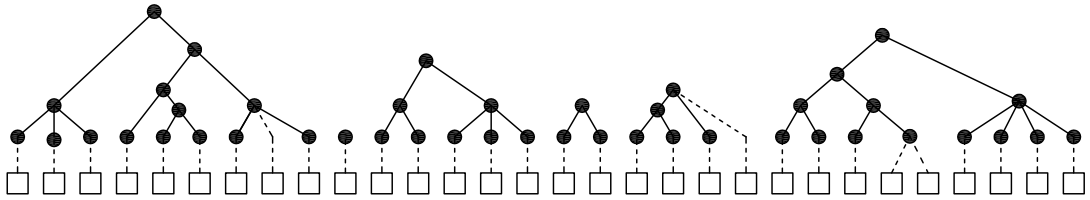


Fig. 5.7: Sketch of the shrubs data structure. The squares at the bottom represent the cells in a grid and the circles the nodes of the shrubs. The cells are associated to the nodes by the dashed lines.

objects, we denote our created data structure as *shrubs*. Figure 5.7 shows a sketch of the shrubs.

5.2. Fundamentals of the Shrubs Data Structure

Notation 5.1 (Shrubs): *We denote the shrubs data structure, or short the shrubs, of a complex object $\mathfrak{O}$ for a given cell width w and tolerance value δ by $\mathcal{S}(\mathfrak{O})$. The shrubs data structure maintains the cell content of all grid cells of a TD-grid $\mathcal{G}(\mathfrak{O})$ for the given tolerance value δ.*

Shrubs maintain the cell content of all grid cells more memory efficient than it is possible by using arrays, as explained in Section 4.2. Thus, a TD-grid that uses shrubs to maintain its cell content is denoted in the following as a *compressed TD-grid*. Each grid cell in the TD-grid references its primitives (according to Definition 3.3) indirectly by referencing a node in the shrubs. Knowing this node the cell content can be obtained. We define the properties of the shrubs as follows:

Definition 5.1 (Shrubs – Associated Cells and Associated Node): *Each occupied cell $\mathcal{C}$ in the TD-grid is directly associated with exactly one node in the shrubs which is denoted by $\mathsf{N}_{\mathcal{C}}$.*

A node in the shrubs can be directly or indirectly associated to an occupied grid cell in the TD-grid. We differ as follows:

- *Each leaf node in the shrubs is directly associated with one or more cells.*
- *An inner node in the shrubs can be directly associated with one or more cells.*
- *An inner node in the shrubs is indirectly associated with the those cells that are directly associated to one of its descendant nodes.*

In summary, every node in the shrubs is associated with its directly and indirectly associated cells. The set of all associated cells of a node N in the shrubs is denoted as $Z(\mathsf{N})$.

Definition 5.2 (Shrubs – Node Content): *Let $A(\mathsf{N})$ be the set of ancestor nodes of a node N in the shrubs. The content of N is the intersection of the cell content of all its associated cells without the content of all ancestor nodes:*

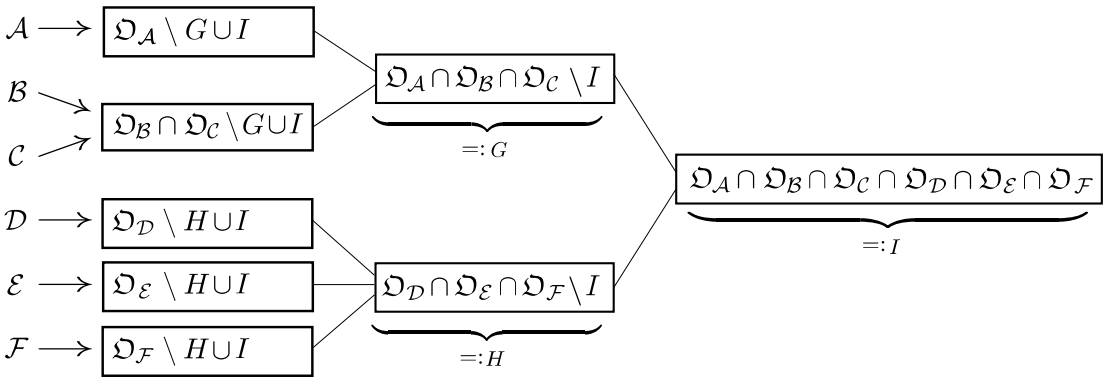
$$\mathfrak{O}_{\mathsf{N}} := \bigcap_{\mathcal{C} \in Z(\mathsf{N})} \mathfrak{O}_{\mathcal{C}} \quad \setminus \quad \bigcup_{\mathsf{A} \in A(\mathsf{N})} \mathfrak{O}_{\mathsf{A}} \tag{5.1}$$

In order to guarantee that the shrubs maintain the content of all grid cells, the content of the associated node $\mathsf{N}_{\mathcal{C}}$ of a grid cell $\mathcal{C}$ has to be

$$\mathfrak{O}_{\mathsf{N}_{\mathcal{C}}} = \mathfrak{O}_{\mathcal{C}} \quad \setminus \bigcup_{\mathsf{A} \in A(\mathsf{N})} \mathfrak{O}_{\mathsf{A}} \,. \tag{5.2}$$

The rules how we associate a cell to a node and how we connect nodes by a parent node will be the issue of Sections 5.3 and 5.4. First, we will have a closer look on some properties of the shrubs.

Consider a small example with the cells $\mathcal{A}, \mathcal{B}, \mathcal{C}, \mathcal{D}, \mathcal{E}$ and $\mathcal{F}$ in the uniform grid $\mathcal{G}(\mathfrak{O})$. $\mathfrak{O}_{\mathcal{A}}, \mathfrak{O}_{\mathcal{B}}, \mathfrak{O}_{\mathcal{C}}, \mathfrak{O}_{\mathcal{D}}, \mathfrak{O}_{\mathcal{E}}$ and $\mathfrak{O}_{\mathcal{F}}$ are the cell contents, respectively, and we have $\mathfrak{O}_{\mathcal{B}} = \mathfrak{O}_{\mathcal{C}}$. The shrubs can be built as displayed in the following:



In the case a cell is requested for its referenced primitives at query time, the primitives are obtained from the shrubs by the union of the associated node's content and the content of all ancestor nodes. In the above example, the primitives that are referenced by cell $\mathcal{A}$ are

$$(\mathfrak{O}_{\mathcal{A}} \setminus G \cup I) \;\cup\; G \;\cup\; I \;=\; \mathfrak{O}_{\mathcal{A}} \,.$$

Between the content of a node N and the content of one of its associated cells $\mathcal{C}$, there is the following relation:

Proposition 5.1: *Let* N *be a node in the shrubs and* $\mathcal{C} \in Z(\mathsf{N})$ *one of its associated cells. Then it is* $\mathfrak{O}_{\mathsf{N}} \subseteq \mathfrak{O}_{\mathcal{C}}$.

Proof 10:

$$\mathfrak{O}_{\mathsf{N}} = \bigcap_{\mathcal{X} \in Z(\mathsf{N})} \mathfrak{O}_{\mathcal{X}} \setminus \bigcup_{\mathsf{A} \in A(\mathsf{N})} \mathfrak{O}_{\mathsf{A}} \quad \subseteq \quad \mathfrak{O}_{\mathcal{C}} \setminus \bigcup_{\mathsf{A} \in A(\mathsf{N})} \mathfrak{O}_{\mathsf{A}} \quad \subseteq \quad \mathfrak{O}_{\mathcal{C}} \tag{5.3}$$

■

In general the referenced primitives of a grid cell are obtained from the shrubs as follows:

Proposition 5.2: *The referenced primitives of a cell* $\mathcal{C}$*, which are denoted as the cell content of* $\mathcal{C}$*, are obtained from the shrubs by*

$$\mathfrak{O}_{\mathcal{C}} = \bigcup_{\mathsf{A} \in A^+(\mathsf{N}_{\mathcal{C}})} \mathfrak{O}_{\mathsf{A}} \,, \tag{5.4}$$

with $A^+(\mathsf{N}_{\mathcal{C}}) = \{\mathsf{N}_{\mathcal{C}}\} \cup A(\mathsf{N}_{\mathcal{C}})$.

Proof 11:

$$\bigcup_{\mathsf{A} \in A^+(\mathsf{N}_{\mathcal{C}})} \mathfrak{O}_{\mathsf{A}} = \mathfrak{O}_{\mathsf{N}_{\mathcal{C}}} \bigcup \left(\bigcup_{\mathsf{A} \in A(\mathsf{N}_{\mathcal{C}})} \mathfrak{O}_{\mathsf{A}} \right) = \left(\mathfrak{O}_{\mathcal{C}} \setminus \bigcup_{\mathsf{A} \in A(\mathsf{N}_{\mathcal{C}})} \mathfrak{O}_{\mathsf{A}} \right) \bigcup \left(\bigcup_{\mathsf{A} \in A(\mathsf{N}_{\mathcal{C}})} \mathfrak{O}_{\mathsf{A}} \right) = \mathfrak{O}_{\mathcal{C}}$$

■

Obtaining the cell content of a cell $\mathcal{C}$ can be implemented efficiently by traversing the shrubs upwards from node $\mathsf{N}_{\mathcal{C}}$ visiting all ancestor nodes of $\mathsf{N}_{\mathcal{C}}$.

By using the shrubs data structure, we can reduce the number of primitive references. In the above example we would have to store

$$|\mathfrak{O}_{\mathcal{A}}| + |\mathfrak{O}_{\mathcal{B}}| + |\mathfrak{O}_{\mathcal{C}}| + |\mathfrak{O}_{\mathcal{D}}| + |\mathfrak{O}_{\mathcal{E}}| + |\mathfrak{O}_{\mathcal{F}}|$$

primitive references without using the shrubs. By using the shrubs we have

$$|\mathfrak{O}_{\mathcal{A}}| + |\mathfrak{O}_{\mathcal{B}}| + |\mathfrak{O}_{\mathcal{C}}| + |\mathfrak{O}_{\mathcal{D}}| + |\mathfrak{O}_{\mathcal{E}}| + |\mathfrak{O}_{\mathcal{F}}| - 5|I| - 2|G| - 2|H| - |\mathfrak{O}_{\mathcal{C}}|$$

primitive references to store.

In general, the number of stored primitives in a grid without shrubs is given naturally by

$$\sum_{\mathcal{C} \in \mathcal{G}(\mathfrak{O})} |\mathfrak{O}_{\mathcal{C}}| \,. \tag{5.5}$$

Proposition 5.3: *The number of stored primitives in a TD-grid $\mathcal{G}(\mathfrak{O})$ that uses the shrubs data structure $\mathcal{S}(\mathfrak{O})$ can be given as*

$$\sum_{\mathsf{N}\in\mathcal{S}(\mathfrak{O})} |\mathfrak{O}_{\mathsf{N}}| \;=\; \sum_{\mathcal{C}\in\mathcal{G}(\mathfrak{O})} |\mathfrak{O}_{\mathcal{C}}| \;-\; \sum_{\mathsf{N}\in\mathcal{S}(\mathfrak{O})} \left(|Z(\mathsf{N})|-1\right)|\mathfrak{O}_{\mathsf{N}}| \;. \tag{5.6}$$

Proof 12:

$$\begin{aligned}(5.6) \Leftrightarrow \sum_{\mathsf{N}\in\mathcal{S}(\mathfrak{O})} |\mathfrak{O}_{\mathsf{N}}| \;&=\; \sum_{\mathcal{C}\in\mathcal{G}(\mathfrak{O})} |\mathfrak{O}_{\mathcal{C}}| - \sum_{\mathsf{N}\in\mathcal{S}(\mathfrak{O})} |Z(\mathsf{N})|\,|\mathfrak{O}_{\mathsf{N}}| + \sum_{\mathsf{N}\in\mathcal{S}(\mathfrak{O})} |\mathfrak{O}_{\mathsf{N}}| \\ \Leftrightarrow \sum_{\mathcal{C}\in\mathcal{G}(\mathfrak{O})} |\mathfrak{O}_{\mathcal{C}}| \;&=\; \sum_{\mathsf{N}\in\mathcal{S}(\mathfrak{O})} |Z(\mathsf{N})|\,|\mathfrak{O}_{\mathsf{N}}|\end{aligned}$$

With Proposition 5.2 we have

$$(5.6) \Leftrightarrow \sum_{\mathcal{C}\in\mathcal{G}(\mathfrak{O})} \left| \bigcup_{\mathsf{N}\in A^+(\mathsf{N}_{\mathcal{C}})} \mathfrak{O}_{\mathsf{N}} \right| = \sum_{\mathsf{N}\in\mathcal{S}(\mathfrak{O})} |Z(\mathsf{N})|\,|\mathfrak{O}_{\mathsf{N}}| \;.$$

Since for every two nodes in $A^+(\mathsf{N}_{\mathcal{C}})$ the node content is disjoint (Definition 5.2), we have

$$(5.6) \Leftrightarrow \sum_{\mathcal{C}\in\mathcal{G}(\mathfrak{O})} \sum_{\mathsf{N}\in A^+(\mathsf{N}_{\mathcal{C}})} |\mathfrak{O}_{\mathsf{N}}| = \sum_{\mathsf{N}\in\mathcal{S}(\mathfrak{O})} |Z(\mathsf{N})|\,|\mathfrak{O}_{\mathsf{N}}|$$

On the left-hand side of the above equation we go over all cells $\mathcal{C}$ in $\mathcal{G}(\mathfrak{O})$. As the sets $A^+(\mathsf{N}_{\mathcal{C}})$ are often not disjoint for different cells $\mathcal{C}$ in $\mathcal{G}(\mathfrak{O})$, $|\mathfrak{O}_{\mathsf{N}}|$ can be counted multiple times for a node N. A node N has $|Z(\mathsf{N})|$ associated cells (Definition 5.1). Thus, $|\mathfrak{O}_{\mathsf{N}}|$ is counted exactly $|Z(\mathsf{N})|$-times on the left-hand side of the above equation. ∎

The proportional contribution of a single grid cell to the number of primitive references in the shrubs is sometimes interesting for the analysis of the compression quality.

Definition 5.3: *Let $\mathcal{C}$ be a grid cell in $\mathcal{G}(\mathfrak{O})$, then its proportional contribution to the number of primitive references in the shrubs $\mathcal{S}(\mathfrak{O})$ is given as*

$$\sum_{\mathsf{A}\in A^+(\mathsf{N}_{\mathcal{C}})} \frac{|\mathfrak{O}_{\mathsf{A}}|}{|Z(\mathsf{A})|} \;. \tag{5.7}$$

By this definition, the number of primitive references per node A is weighted by its number of associated cells. The sum over the proportional contribution of all grid cells in $\mathcal{G}(\mathfrak{O})$ has to be equal with the number of primitive references in the shrubs, thus

$$\sum_{\mathcal{C}\in\mathcal{G}(\mathfrak{O})} \sum_{\mathsf{A}\in A^+(\mathsf{N}_{\mathcal{C}})} \frac{|\mathfrak{O}_{\mathsf{A}}|}{|Z(\mathsf{A})|} \;=\; \sum_{\mathsf{N}\in\mathcal{S}(\mathfrak{O})} |\mathfrak{O}_{\mathsf{N}}| \;.$$

This is true because on the left side $\frac{|\mathfrak{O}_{\mathsf{A}}|}{|Z(\mathsf{A})|}$ is counted several times, once for each cell that has A as associated node or as ancestor node of its associated node. This is exactly $|Z(\mathsf{A})|$-times.

Definition 5.4 (Compression Ratio): *The compression ratio of primitive references is defined as*

$$\Theta_P := \frac{\sum\limits_{\mathsf{N}\in\mathcal{S}(\mathfrak{O})} |\mathfrak{O}_{\mathsf{N}}|}{\sum\limits_{\mathcal{C}\in\mathcal{G}(\mathfrak{O})} |\mathfrak{O}_{\mathcal{C}}|} \,. \tag{5.8}$$

Please note that following:

- The term compression ratio is often defined ambiguously in literature. With our definition we follow the one given by Salomon [46]. For example a compression ratio of 0.6 means that the data is compressed to 60% of the original size. Thus, 40% of the space is saved. The smaller the compression ratio, the better is the compression.
- Θ_P is related to the number of referenced primitives and not to the total memory consumption of the grid data structure. Later, in Section 5.5, we will consider the compression ratio of the total grid memory.

Achieving Good Compressions

In order to achieve a good compression ratio our aim is to create shrubs with a small Θ_P. A small Θ_P can be reached if the high level nodes in the shrubs have a large cell content. This can be reached if nodes with possibly similar content are connected by a parent node. We will present some different strategies to connect cells and nodes in order to create the shrubs in Section 5.3 and 5.4.

Related Work & Integration into the Field of Data Compression

Related Work on reducing the memory consumption of uniform grids focuses on hashing methods in order to store only the occupied grid cells. We discussed this issue already in Section 4.2. Our aim is not only to store less grid cells – moreover, our aim here is to use less memory to store the cell contents. To the best of our knowledge, this issue has not been considered before.

Lossy and Lossless Data Compression

Using less memory to store the cell content is actually a compression of the cell content, thus an issue of data compression. There are two principal ways how data compression is achieved: The lossy or the lossless compression. Lossy data compression removes less important data. Thus, one cannot reconstruct the original data from the lossy compressed data set. Because of that, lossy data compression is not suitable for compressing the cell content as we cannot miss any of the referenced primitives. Lossless data compression naturally does not achieve that high compression rate as lossy data

compression. But the original data set can be reconstructed from the compressed data set. Lossless data compression works with the utilization of redundant data. The content of one grid cell itself is never redundant, but we noticed within this section that the content of neighboring cells is usually highly redundant in TD-grids.

Interpretation in Terms of Video Compression

Our task to compress the memory consumption of the cell content, can be expressed in terms of video compression. A cell content corresponds to one frame (image) in the video. In contrast to one image, which is a fixed sized 2-dimensional data field with color values, a cell content is a 1-dimensional data field of arbitrary many integer values. An image can be compressed separately, independent from other pictures, (for video compression it is often compressed with lossy techniques) whereas we cannot compress a cell content separately, independent from other cell contents. The one discrete time axis in a video corresponds to the three discrete space axes in our grid. In video compression techniques, the similarity between two consecutive frames is utilized for data compression. For example in the MPEG standard [20] for a sequence of frames, some frames are stored completely (denoted as I-frames), some frames store only the differences to the previous frame (denoted as P-frames) and some frames stored only the differences to a previous and to a following frame (denoted as B-frames). In order to compress the cell content in a grid, we utilize the similarity of neighboring cells. In contrast to video compression, where a frame has only two neighboring frames on the time axis, a cell in the grid has 27 neighbor cells (or 6 if we consider only face-face neighbors). Thus, which neighbor or neighbors are used to connect with, is one main question of our work.

Decoding at Query Time

It is very important that the decoding process takes almost no or only negligible time, because we have to obtain the cell content of queried cells from the shrubs at query time. Given the content of two neighboring nodes, we do not store one content completely (like it is done for an I-frame) and the differences of the other content separately (like it is done for a P- or B-frame). Assumed, we have two nodes in the shrubs with one common ancestor node. Then this ancestor node is an artificial construct (and can be seen like an I-frame), where each of the two nodes store the difference, which are actually only additions, to the ancestor node. Thus, nodes with non-empty parent node can be interpreted as P-frames and the nodes with empty parents can be interpreted as I-frames. Since we have only additions, we can output the cell content of a cell that is stored in the shrubs without additional effort for decoding.

Handling Different Types of Primitives

A grid cell may reference different types of primitives. In such cases there are two possibilities to use the shrubs data structure. One is to use one shrubs data structure for each type of primitive. The other is to use only one shrubs data structure, where each primitive reference is marked somehow in order to know of which type it is.

For example when creating a TD-grid that uses the idea of core-primitives (refer to Section 3.3.4), the referenced core-primitives in every cell must be distinguished from the referenced candidate primitives. Following the first possibility, two shrubs are

required – one to maintain the core-primitives and one to maintain the general candidate primitives. For the second possibility one must provide two IDs per primitive, one for its role as core-primitive and one for its role as general candidate primitive.

We will focus in the following the case that the grid cells reference only one type of primitives. But everything will be easily transferable for more types of primitives.

5.3. Schematic Construction Variants

Within this section we present three different schematic construction variants. First we will explain the common principle of all schematic variants and then address the individual variants. The first variant is the one where the basic idea of the schematic construction is most easy to comprehend. The second variant is an improvement with a slightly more complex construction algorithm, which also helps to understand our final variant. The third and final variant is the one with the most complex construction algorithm, but also the one with the best compression ratio.

The schematic construction variants have all in common that the shrubs as well as the grid are created at the same time by a recursive algorithm, which is given in Algorithm 7. Therefore we create a tree of nodes. We call this tree the *construction tree*. The construction tree is a superset of the shrubs. Each node N in the construction tree is responsible for a region of space R and finally stores a set of primitives, which is the node content $\mathfrak{O}_{\mathsf{N}}$. Usually the construction tree contains also nodes with empty node content. Such nodes do not contribute to the cell content of any associated cell (Proposition 5.2). Thus, these nodes are not necessary to store in the shrubs – they are only temporary required during the schematic construction of the shrubs. So, these nodes are nodes in the construction tree but not in the shrubs. For convenience we will not differ between the construction tree and the shrubs for the rest of this section. How to get the shrubs from the construction tree is focused in Section 5.5.

The shrubs and the grid are created for a given cell width w and a given tolerance value δ. Each recursion is called for a node N, with a set of primitives P and a region R that is a cuboid.

The first recursion is called for the root node of the construction tree, where the set of primitives P are all all primitives of the object $\mathfrak{O}$. The region R is a cube with edge length $2^N w$. We choose $N \in \mathbb{N}$ such that the cube is the smallest cube with edge length $2^N w$ that covers all primitives of the object $\mathfrak{O}$ including their $(\delta + r)$-offsets.

In each recursion step, as long as the region R is not a cube with edge length w, R is split to child-regions $\{R_0, ..., R_n\}$ (line 3). In how many child-regions R is split and, moreover, how R is split depends on the schematic variant and is explained later. The primitives in P are each assigned to one or more child-regions R_i. This means more precisely that the primitives in the input set P are tested for intersection with the $(\delta + r)$-offset $R_i \oplus \mathcal{S}_{r+\delta}$ of every child-region R_i. New primitive sets P_i are created such that each P_i contains all primitives that intersect $R_i \oplus \mathcal{S}_{r+\delta}$ (line 5). In case a set P_i

Algorithm 7: N.RecursiveBuild(R, P)

Data: R // the region for which N is responsible (given as cuboid)
P // a set of primitives (the primitives that are tolerance violating with R)

Result: I // a set of primitives (the intersection of the content of all cells in $Z(\mathsf{N})$)

```
1  I ← P
2  if R is not a cube with edge length w then
3      {R_0, ..., R_n} ← make the regions for the child nodes by splitting R
4      for i = 0 to n do
5          P_i ← all primitives in P that intersect R_i ⊕ S_{r+δ}
6          if P_i ≠ ∅ then
7              N_i ← create child node i
8              I_i ← N_i.RecursiveBuild(R_i, P_i)
9      foreach existing child node N_i do
10         I ← I ∩ I_i
11     foreach existing child node N_i do
12         Ø_{N_i} ← I_i \ I
13 else
14     C ← create grid cell for R and set N_C = N
15 return I
```

is not empty, a child node N_i of N is created and the recursion is called again for the child node N_i with P_i and R_i (line 8).

The recursion stops in the case R is a cube with edge length w. Then a grid cell $\mathcal{C}$ is created in the region of R with cell content P (but note, P will not be stored by $\mathcal{C}$ directly – the shrubs maintain the cell content and are created such that P can be obtained for $\mathcal{C}$). The node N is set as associated node of the cell $\mathcal{C}$, thus $\mathcal{C}$ will store a reference to N (line 14). Conversely, this means that $\mathcal{C}$ is the only directly associated cell to N, thus, $Z(\mathsf{N}) = \{\mathcal{C}\}$. This last recursion returns directly the set of primitives P.

When the recursion is called for an inner node, the recursion returns the primitives that are tolerance violating with all its child-regions $R_0, ..., R_n$. In general, the recursion that is called for a node N_i returns the primitives that are tolerance violating with all its associated cells. So we have after the recursion call (line 8)

$$I_i \;=\; \bigcap_{\mathcal{C} \in Z(\mathsf{N}_i)} \mathfrak{O}_{\mathcal{C}} \;.$$

After all recursive calls for one node N are finished (line 8), the remaining task of the function is to set the node content of its child nodes N_i (lines 9 - 12). We have defined the content of a node in the shrubs as the intersection of the cell content of all its associated cells without the content of all ancestor nodes (refer to Definition 5.2).

Accordingly, for a child node N_i of N we have

$$\mathfrak{O}_{\mathsf{N}_i} = \underbrace{\bigcap_{\mathcal{C} \in Z(\mathsf{N}_i)} \mathfrak{O}_{\mathcal{C}}}_{(A)} \setminus \underbrace{\bigcup_{\mathsf{A} \in A(\mathsf{N}_i)} \mathfrak{O}_{\mathsf{A}}}_{(B)} = I_i \setminus I \,.$$

This is true, since we know already from above that $I_i = (A)$. Further, $I = (B)$ because the content of I will be distributed along the content of the nodes in $A(\mathsf{N}_i)$ in the following closing recursion steps.

5.3.1. Octree Construction Schema

For the octree construction scheme, nodes are connected in an octree-like manner. Each node can have up to eight child nodes, thus, in Algorithm 7 we have $n = 8$. In every recursion step, R is split into its eight octants. Refer to Figure 5.8 that visualizes this scheme in a 2d-example.

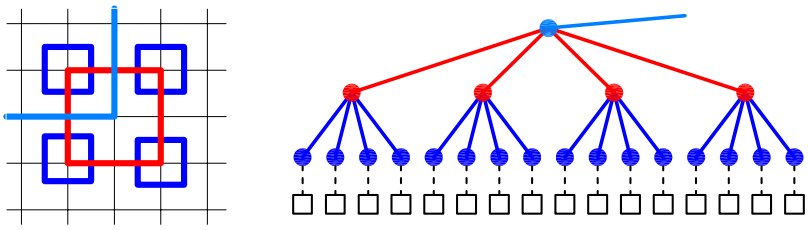


Fig. 5.8: 2d-example of the octree construction scheme, displayed as quadtree construction scheme.

5.3.2. Strict Binary Tree (SBT) Construction Schema

The SBT construction schema is similar to the octree construction scheme, where blocks of eight nodes are connected. The difference is that the SBT construction schema connects them in three steps. A node can only have up to two child nodes. In Algorithm 7 we have therefore $n = 2$. In every third recursion step, the cube is split by a plane with normal in x-direction through the cubes center point. Thereby two cuboids arise. In every subsequent recursion step each of the two cuboids is split by a plane with normal in y-direction trough the respective cuboid center. Thereby two cuboids arise

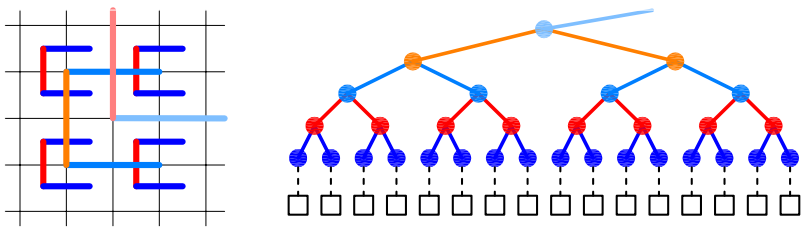


Fig. 5.9: 2d-example of the SBT construction scheme.

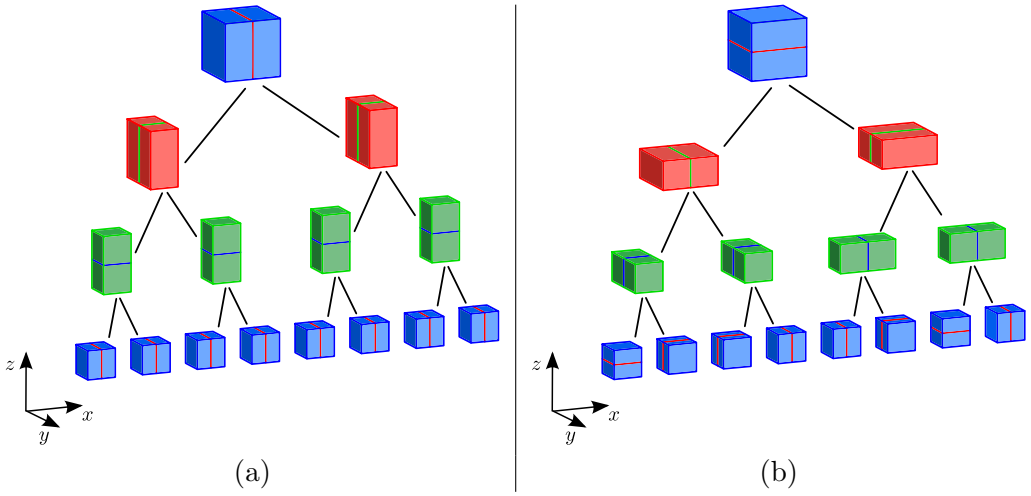


Fig. 5.10: 3d-example of the range splitting above four levels for the SBT construction scheme (a) and the OBT construction scheme (b).

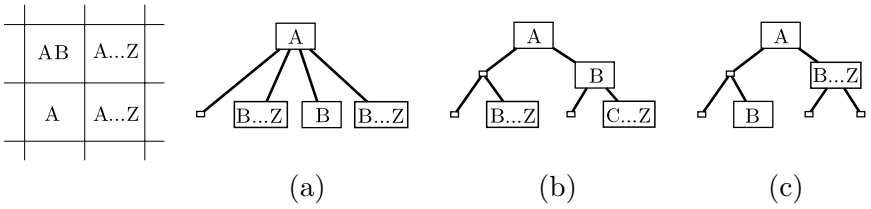


Fig. 5.11: 2d-example to connect a block of four cells with contents A, AB and twice A to Z (left) by the octree construction scheme (a), the SBT construction scheme (b) and the OBT construction scheme (c). The grid cells are not visualized in (a), (b) and (c).

respectively, thus in summary four cuboids. Finally, in every subsequent recursion step each of the cuboids is split by a plane with normal in z-direction trough the respective cuboid center. Thereby two cubes arise respectively, thus in summary eight cubes. These cubes correspond to the octants of the cube three recursion steps earlier. Refer to Figure 5.9 that visualizes this scheme in a 2d-example and refer to Figure 5.10 (a) that visualizes the splitting of the range over three recursions, starting with a cube as input range.

The tree that is constructed by the SBT construction scheme has three times the depth of the respective one constructed by the octree scheme. By the SBT construction scheme we expect a higher compression of the primitive references due to the additional levels in the constructed tree. Refer for an example to Figure 5.11 (a) and (b), where four cells are connected by the octree construction scheme and by the SBT construction scheme.

Algorithm 8: N.RecursiveBuild(R, P) for the Optimized Binary Tree Construction Schema

Data: R // the region for which N is responsible (given as cuboid)
P // a set of primitives (the primitives that are tolerance violating with R)

Result: I // a set of primitives (the intersection of the content of all cells in $Z(\mathsf{N})$)

```
1  I ← P
2  if R is not a cube with edge length w then
3      {R_0, ..., R_8} ← make the regions for the child nodes by splitting R to octants
4      for i = 0 to 8 do
5          P_i ← all primitives in P that intersect R_i ⊕ S_{r+δ}
6          if P_i ≠ ∅ then
7              N_i ← create great-grandchild node i
8              I_i ← N_i.RecursiveBuild(R_i, P_i)
9      for s = 0 to 12 do
10         T_s ← create root of a tree for every possible splitting sequence
11         I_s ← T_s.SplitSeqTree(s, R, 8, {P_0, ..., P_7})
12     T_best ← the tree T_s, s ∈ [0, 11] with the best compression ratio Θ_P
13     I ← the set of primitives I_s, s ∈ [0, 11] that was returned by T_best
14     copy the content of the 3^rd-level nodes of T_best to O_{N_i}
15     set the nodes N_i as child nodes of the 2^nd-level nodes of T_best
16     set the 1^st-level nodes of T_best as child nodes of N
17 else
18     C ← create grid cell for R and set N_C = N
19 return I
```

5.3.3. Optimal Binary Tree (OBT) Construction Schema

The OBT construction scheme is similar to the SBT construction scheme. As in the SBT scheme, the region R is a cube in every third level of the tree. But in contrast to the SBT scheme, the splitting sequence of a cube into its eight octants over three levels is optimized. For one cube there are 12 possible splitting sequences with planes containing the cuboids center point and normals parallel to the x-, y- or z-axis such that after three iterations the eight octants of the original cube arise. All possible splitting sequences over three levels are calculated and evaluated in order to choose for each cube the optimal splitting sequence. The 12 splitting sequences are:

1. The input cube is split with a plane trough the cube's center with normal parallel to the x-, y- or z-axis $\Rightarrow$ 3 possibilities
2. The two resulting cuboids are split with a plane trough the respective cuboids center with normal parallel to the x-, y- or z-axis receptively, but not parallel to the axis used before $\Rightarrow 2 \cdot 2$ possibilities

Algorithm 9: T.SplitSeqTree$(S, R, n, \{P_0, ..., P_{n-1}\})$

Data: S // identifies one of the 12 splitting sequences
R // the region where T is responsible for
n // number of corresponding octants for the region R
$\{P_0, ..., P_{n-1}\}$ // a set of sets of primitives (P_i belongs to octant i)

Result: I // a set of primitives

```
1  I ← P_0 ∪ ... ∪ P_{n-1}
2  if n > 1 then
3      {R^(0), R^(1)} ← split R according to S
4      {P^(0), P^(1)} ← assign sets in {P_0, ..., P_{n-1}} to P^(0) or P^(1) according to S
5      for i = 0 to 1 do
6          if at least one P_j ∈ P^(i) is not empty then
7              T^(i) ← create child node i
8              I^(i) ← T^(i).SplitSeqTree(S, R^(i), n/2, P^(i))
9      foreach existing child node T^(i) do
10         I ← I ∩ I^(i)
11     foreach existing child node T^(i) do
12         Ø_{T^(i)} ← I^(i) \ I
13 return I
```

3. The four resulting cuboids are split with a plane trough the respective cuboid's center with normal parallel to the respective remaining axis.

Figure 5.10 shows 2 of the 12 possible splitting sequences, where (a) is the special case of the SBT construction scheme.

With the OBT construction scheme higher compression rates can be expected than with the SBT construction scheme. The higher compression rate is achieved because for each node in every third level of the tree, the optimal splitting sequence for the next three levels is chosen. Refer for example to Figure 5.11 (b) and (c), where four cells are connected by the strict and by the OBT construction scheme. Nevertheless, the higher compression rate is achieved by a more complicated construction algorithm.

Algorithm 7 is slightly modified for the OBT construction scheme and shown in Algorithm 8. Going into the recursion is very similar to the octree construction scheme (line 3 to 8). The only difference is that in line 7 the great-grandchild nodes are created. Thus, the recursion goes in every recursion step three levels deeper.

After the recursion is calculated for a level l with $l\%3 = 0$, all possible splitting sequences of splitting the input cube R into its eight octants are calculated. Therefor 12 temporary binary trees $\{T_0, ..., T_{11}\}$, each with four levels, are created (line 10 and line 11 as well as Algorithm 9).

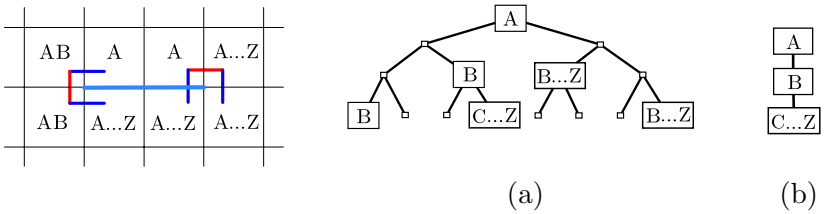


Fig. 5.12: 2d-example to connect two blocks of four cells with the displayed contents A, AB and A to Z (left) by the optimal binary tree construction scheme (a) and by the free construction (b). The grid cells are not visualized in (a) and (b).

Each leaf of a tree T_i corresponds to one of the eight octants R_i and therefore with one of the great-grandchild nodes N_i. The tree T_i with the best compression ratio Θ_P (Definition 5.4) is chosen as T_{best} in order to represent the child-, grandchild- and great-grandchild nodes of the node N (lines 14 to 16). This means in detail that the content of its 3^{rd} level nodes (leaf nodes) of T_{best} is copied to the content of the eight already created great-grandchild nodes N_i of N. The eight great-grandchild nodes N_i are set as child nodes of the 2^{nd} level nodes of T_{best} and the two 1^{st} level nodes of T_{best} are set as child nodes of the node N.

The principle of Algorithm 9 is similar to the one of the first if-case in Algorithm 7, except that the primitives must not be tested on tolerance with the regions (because it is already known) and that the condition that stops the recursion is different (it stops with the fourth iteration).

5.4. Free Construction

The connection of nodes in the previous presented schematic variants follows fixed rules that are independent of the individual cell contents. Following these rules, it frequently happens that neighboring cells with similar or equal content are associated to nodes that are in completely different branches of the shrubs. These nodes have no or only few common ancestor nodes. Further it happens, that nodes with less similar content are connected with a parent node although there would be other nodes with a greater similarity. Thereby the potential to compress the number of referenced primitives is not optimally exploited. Figure 5.12 (a) visualizes this problem. The compression by the binary tree construction scheme is not optimal. For example the cells with content A...Z are associated to different nodes in the shrubs.

For this reason we investigate and analyze another construction variant, the so called free construction. The basic idea is that each grid cell is individually considered and connected to the branch of the shrubs containing its best fitting node. The free construction primarily focuses to connect nodes with equal or similar content independent of any schematic rule. In the example shown by Figure 5.12 (b) the cells are connected by the free construction scheme much more optimal.

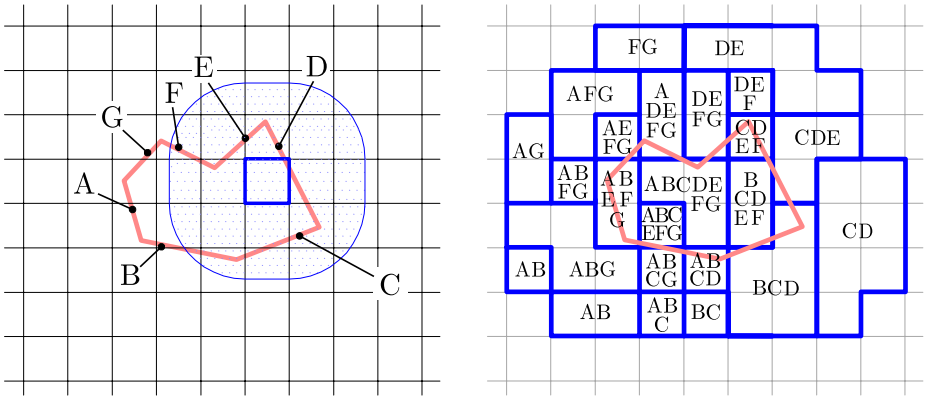


Fig. 5.13: 2d-example of a polygon within a TD-grid. One zone of influence (blue) and the naming of the primitives (A-G) are visualized on the left-hand picture. The cell contents are visualized on the right-hand picture.

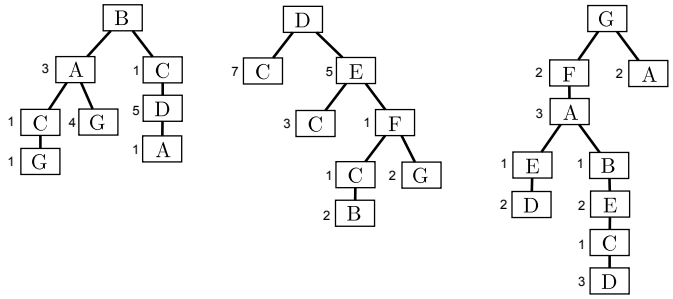


Fig. 5.14: An example of a free constructed shrubs for the grid of Figure 5.13. The grid cells are not visualized. The numbers beside the nodes give the number of directly associated cells of the respective node.

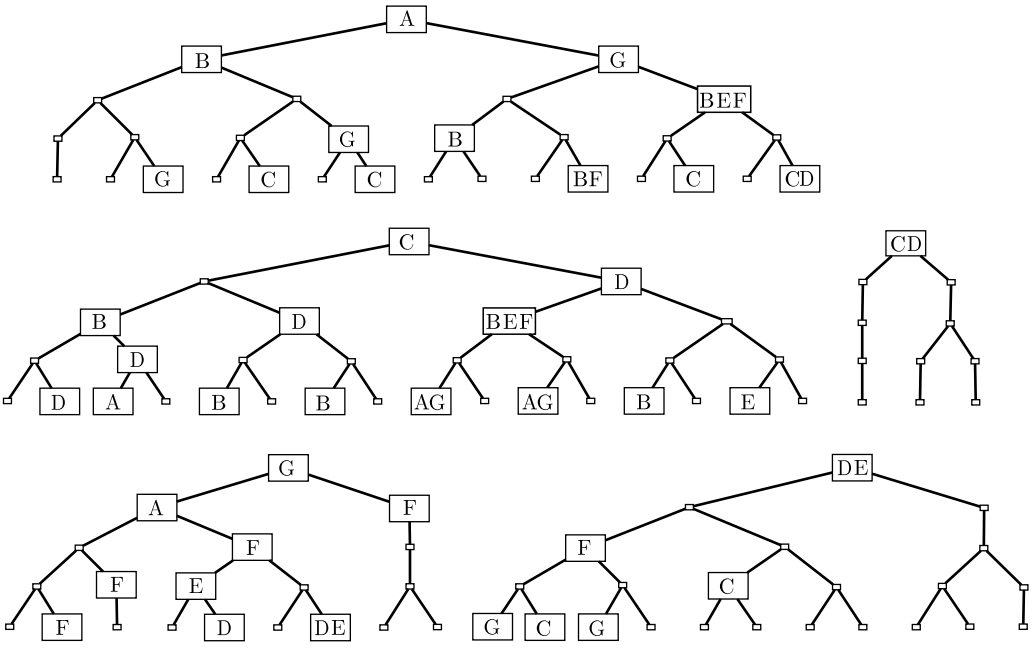


Fig. 5.15: An example of the shrubs, constructed by the SBT construction scheme, for the grid of Figure 5.13. The grid cells are not visualized. Each leaf node has one directly associated grid cell.

For a more comprehensive example refer to the Figures 5.13, 5.14 and 5.15. In Figure 5.13 a 2d-example of a polygon within a TD-grid is shown, where the referenced segments per cell are given in the right-hand picture. In the uncompressed TD-grid 176 segment references are required. Figure 5.14 shows how a free constructed shrubs could possibly look like. There are 26 segment references and thereby a compression ratio of $\Theta_P = 0.15$, thus 85% of all primitive references can be saved. For comparison Figure 5.15 shows the shrubs constructed by the SBT scheme. There are 53 segment references required, which achieves a compression ratio of $\Theta_P = 0.25$, thus 75% of all primitive references can be saved. In this example, the free construction achieves a better compression.

An Incremental Greedy Algorithm

Within this section we analyze how the free constructed shrubs can be created. The presented construction algorithm is an incremental greedy algorithm, that successively connects cells of the grid to the best fitting node in the shrubs data structure. The most important questions are:

- How to determine the best fitting node to connect a cell?
- How to connect a cell with the shrubs?
- In which order are cells connected to the shrubs?

All these issues will be discussed in the following sections.

5.4.1. Determining the Best Fitting Node

As concluded in Section 5.2, nodes with possibly similar content have to be connected by a parent node in order to achieve a good compression ratio. Our central assumption is that for a given grid cell the grid cell with the most similar content is a neighboring cell.

We build the shrubs incrementally by connecting in every iteration one occupied grid cell to the shrubs. Let $\mathcal{C}^{(t)}$ be an occupied grid cell and $\mathcal{S}^{(t)}(\mathfrak{O})$ be the shrubs at time $t \in \{0, 1, ..., n\}$. If $\mathcal{C}^{(t)}$ should be connected to the shrubs, we consider its neighboring cells that are already connected to the shrubs. A grid cell has at most 26 occupied neighboring cells. Let $\mathcal{N}_0^{(t)}, ..., \mathcal{N}_m^{(t)}$, $m < 26$ be the occupied neighboring cells of $\mathcal{C}^{(t)}$ that are connected to the shrubs $\mathcal{S}^{(t)}(\mathfrak{O})$.

There are different ways to measure the similarity between two grid cells. To determine the most similar neighbor cell to $\mathcal{C}^{(t)}$ we could determine the neighbor for which we have the largest cell content intersection, given as

$$\max_i \left\{ \left| \mathfrak{O}_{\mathcal{C}^{(t)}} \cap \mathfrak{O}_{\mathcal{N}_i^{(t)}} \right| \right\}$$

or the smallest cell content difference, given as

$$\min_i \left\{ \left| \mathfrak{O}_{\mathcal{C}^{(t)}} \setminus \mathfrak{O}_{\mathcal{N}_i^{(t)}} \right| + \left| \mathfrak{O}_{\mathcal{N}_i^{(t)}} \setminus \mathfrak{O}_{\mathcal{C}^{(t)}} \right| \right\} .$$

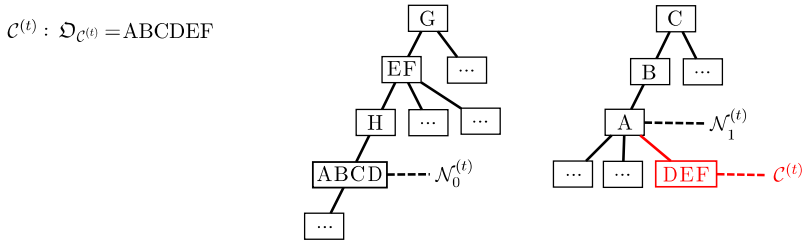


Fig. 5.16: Connecting the cell $\mathcal{C}^{(t)}$ with cell content $\mathfrak{O}_{\mathcal{C}^{(t)}}$ =ABCDEF to the shrubs. Although the neighbor cell $\mathcal{N}_0^{(t)}$ is the most similar neighbor cell of $\mathcal{C}^{(t)}$, the cell $\mathcal{C}^{(t)}$ cannot be connected under the associated node of $\mathcal{N}_0^{(t)}$ because G $\notin \mathfrak{O}_{\mathcal{C}^{(t)}}$ and H $\notin \mathfrak{O}_{\mathcal{C}^{(t)}}$.

Although both values are easy to compute, they are not sufficient to determine the best fitting node to connect $\mathcal{C}^{(t)}$ to the shrubs $\mathcal{S}^{(t)}(\mathfrak{O})$. For the best fitting the content of the associated node of a neighboring cell as well as all its ancestor nodes have to be considered. Refer for example to Figure 5.16. Although the neighboring cell $\mathcal{N}_0^{(t)}$ is the most similar neighboring cell of $\mathcal{C}^{(t)}$, the cell $\mathcal{C}^{(t)}$ cannot be connected to the associated node of $\mathcal{N}_0^{(t)}$ because G $\notin \mathfrak{O}_{\mathcal{C}^{(t)}}$ and H $\notin \mathfrak{O}_{\mathcal{C}^{(t)}}$, consider Proposition 5.2.

The overall goal when connecting a cell to the shrubs is to keep the total number of primitive references in the shrubs as small as possible. Thus, when connecting a cell to the shrubs, we want to have a least possible number of additional primitive references. Equivalently, we search for a large number of primitive references that are already contained in the shrubs and can be reused by the newly added cell.

Definition 5.5 (Node Chain Intersection): *Let $\mathcal{C}$ be a grid cell and N a node in the shrubs. Further let $A^+_{ord}(\mathsf{N}) = \{\mathsf{A}_0, ..., \mathsf{A}_m\}$ be the ordered set with $\mathsf{N} = \mathsf{A}_0$ and all ancestor nodes of N such that each A_{j+1} is the parent node of A_j in the shrubs for all $0 \leq j < m$. In the following we will denote $A^+_{ord}(\mathsf{N})$ as a node chain.*

Further let σ be the smallest index such that $\mathfrak{O}_{\mathsf{A}_j} \subset \mathfrak{O}_{\mathcal{C}} \quad \forall j \in \{\sigma, \sigma+1, ..., m\}$.

We define the node chain intersection as

$$\mathcal{C} \overset{*}{\cap} \mathsf{N} := \begin{cases} \bigcup_{j=0}^{m} \mathfrak{O}_{\mathsf{A}_j}, & \sigma = 0 \\ (\bigcup_{j=\sigma}^{m} \mathfrak{O}_{\mathsf{A}_j}) \cup (\mathfrak{O}_{\mathcal{C}} \cap \mathfrak{O}_{\mathsf{A}_{\sigma-1}}), & 0 < \sigma \leq m \\ \mathfrak{O}_{\mathcal{C}} \cap \mathfrak{O}_{\mathsf{A}_m}, & \mathfrak{O}_{\mathsf{A}_m} \not\subset \mathfrak{O}_{\mathcal{C}} \end{cases}$$

The primitives that are given by the node chain intersection are the primitives that can be reused when adding the cell $\mathcal{C}$ to the node chain $A^+_{ord}(\mathsf{N})$. Thus, the additional primitive references in the shrubs after adding $\mathcal{C}$ to the node chain $A^+_{ord}(\mathsf{N})$ are the primitive references $\mathfrak{O}_{\mathcal{C}} \setminus (\mathcal{C} \overset{*}{\cap} \mathsf{N})$.

Definition 5.6 (Best Fitting Node Chain): *The best fitting node chain to connect a cell $\mathcal{C}^{(t)}$ to the shrubs $\mathcal{S}^{(t)}(\mathfrak{O})$ is the node chain $A^{+}_{ord}(\mathsf{N}_{\mathcal{N}_i^{(t)}})$, for which we have the largest node chain intersection, given as*

$$\max_i \left\{ \left| \mathcal{C}^{(t)} \overset{*}{\cap} \mathsf{N}_{\mathcal{N}_i^{(t)}} \right| \right\} ,$$

where $\mathsf{N}_{\mathcal{N}_i^{(t)}}$ are the associated nodes of the already connected neighboring cells $\mathcal{N}_i^{(t)}$ of $\mathcal{C}$.

Definition 5.7 (Best Fitting Node): *The best fitting node to connect a cell $\mathcal{C}^{(t)}$ to the shrubs $\mathcal{S}^{(t)}(\mathfrak{O})$ in its best fitting node chain $A^{+}_{ord}(\mathsf{N}) = \{A_0, ..., A_m\}$ is*

$$\begin{array}{ll} \mathsf{A}_0 & \text{in the case } \sigma = 0 \\ \mathsf{A}_\sigma & \text{in the case } 0 < \sigma \leq m \ \wedge \ \mathfrak{O}_{\mathsf{A}_{\sigma-1}} \cap \mathfrak{O}_{\mathcal{C}} = \emptyset \\ \mathsf{A}_{\sigma-1} & \text{in the case } 0 < \sigma \leq m \ \wedge \ \mathfrak{O}_{\mathsf{A}_{\sigma-1}} \cap \mathfrak{O}_{\mathcal{C}} \neq \emptyset \\ \mathsf{A}_m & \text{in the case } \mathfrak{O}_{\mathsf{A}_m} \not\subset \mathfrak{O}_{\mathcal{C}^{(t)}} \end{array}$$

where σ is the smallest index for which $\mathfrak{O}_{\mathsf{A}_j} \subset \mathfrak{O}_{\mathcal{C}^{(t)}} \quad \forall j \in \{\sigma, ..., m\}$.

Reconsider Figure 5.16 and examine the node chain intersection of the cell $\mathcal{C}$ with the node chains of the associated nodes of its neighboring cells $\mathcal{N}_0^{(t)}$ and $\mathcal{N}_1^{(t)}$. We have $\mathcal{C}^{(t)} \overset{*}{\cap} \mathsf{N}_{\mathcal{N}_0^{(t)}} = \emptyset$ because G $\not\subset$ ABCDEF. Further, we have $\mathcal{C}^{(t)} \overset{*}{\cap} \mathsf{N}_{\mathcal{N}_1^{(t)}} =$ ABC, because C,B,A $\subset$ ABCDEF. So the best fitting node chain is $A^{+}_{ord}(\mathsf{N}_{\mathcal{N}_1^{(t)}})$. We get $\sigma = 0$ and $\mathsf{N}_{\mathcal{N}_1^{(t)}}$ as the best fitting node. The cell $\mathcal{C}^{(t)}$ is connected to the shrubs by adding a new node with content $\mathfrak{O}_{\mathcal{C}}^{(t)} \setminus (\mathcal{C}^{(t)} \overset{*}{\cap} \mathsf{N}_{\mathcal{N}_1^{(t)}})$, which is ABCDEF $\setminus$ ABC = DEF. How a cell is connected to the shrubs in general will be explained in the following section.

5.4.2. Connecting a Cell to the Shrubs

Connection Cases

A cell $\mathcal{C}$ is connected to the shrubs by connecting it to its best fitting node or by extending its best fitting node chain and connecting it to this extension or by creating a new shrub in the shrubs (thus, a new tree in the forest). So there are different cases to be considered:

- Case C1: No neighboring cell of the cell $\mathcal{C}$ has been added to the shrubs so far. In this case a new node $\mathsf{N}_{\mathcal{C}}$ without parent node is created and added to the shrubs. $\mathsf{N}_{\mathcal{C}}$ is set as the associated node of $\mathcal{C}$, with node content $\mathfrak{O}_{\mathsf{N}_{\mathcal{C}}} = \mathfrak{O}_{\mathcal{C}}$.

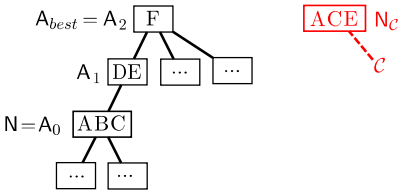


Fig. 5.17: Example for case C2: Connecting the cell $\mathcal{C}$ with $\mathfrak{O}_{\mathcal{C}}$=ACE to the shrubs.

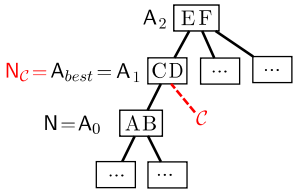


Fig. 5.18: Example for case C3: Connecting the cell $\mathcal{C}$ with $\mathfrak{O}_{\mathcal{C}}$=CDEF to the shrubs.

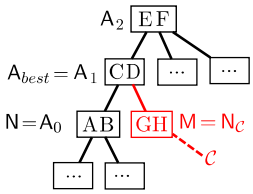


Fig. 5.19: Example for case C4: Connecting the cell $\mathcal{C}$ with $\mathfrak{O}_{\mathcal{C}}$=CDEFGH to the shrubs.

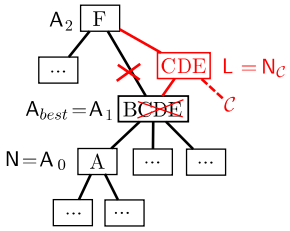


Fig. 5.20: Example for case C5: Connecting the cell $\mathcal{C}$ with $\mathfrak{O}_{\mathcal{C}}$=CDEF to the shrubs.

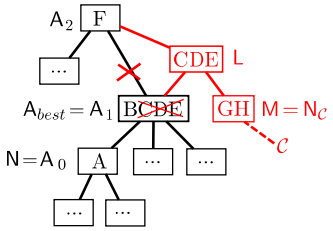


Fig. 5.21: Example for case C6: Connecting the cell $\mathcal{C}$ with $\mathfrak{O}_{\mathcal{C}}$=CDEFGH to the shrubs.

For the case one or more neighboring cells of the cell $\mathcal{C}$ have been added to the shrubs, let $A^{+}_{ord}(\mathsf{N})$ be the best fitting node chain of $\mathcal{C}$ and let A_{best} be the best fitting node of $\mathcal{C}$.

- Case C2: $\mathcal{C} \overset{*}{\cap} \mathsf{N} = \emptyset$
  In this case $\mathcal{C}$ cannot be connected to any node in $A^{+}_{ord}(\mathsf{N})$. A new node $\mathsf{N}_{\mathcal{C}}$ without parent node is created and added to the shrubs. $\mathsf{N}_{\mathcal{C}}$ is set as the associated node of $\mathcal{C}$, with node content $\mathfrak{O}_{\mathsf{N}_{\mathcal{C}}} = \mathfrak{O}_{\mathcal{C}}$. Refer to Figure 5.17 for an example.

In the case $\mathcal{C} \overset{*}{\cap} \mathsf{N} \neq \emptyset$, $\mathcal{C}$ can be connected to a node in $A^{+}_{ord}(\mathsf{N})$. The following cases are considered:

- Case C3: $\mathfrak{O}_{\mathcal{C}} \setminus (\mathcal{C} \overset{*}{\cap} \mathsf{N}) = \emptyset \ \wedge \ \mathfrak{O}_{\mathsf{A}_{best}} \subset \mathfrak{O}_{\mathcal{C}}$
  The node A_{best} can be set directly as the associated node of $\mathcal{C}$, thus $\mathsf{N}_{\mathcal{C}} = \mathsf{A}_{best}$. Refer to Figure 5.18 for an example.

- Case C4: $\mathfrak{O}_{\mathcal{C}} \setminus (\mathcal{C} \overset{*}{\cap} \mathsf{N}) \neq \emptyset \ \wedge \ \mathfrak{O}_{\mathsf{A}_{best}} \subset \mathfrak{O}_{\mathcal{C}}$
  A new node M is created and added to the shrubs. The parent node of M is A_{best}. The node content is $\mathfrak{O}_{\mathsf{M}} = \mathfrak{O}_{\mathcal{C}} \setminus (\mathcal{C} \overset{*}{\cap} \mathsf{N})$ and the associated node of the cell $\mathcal{C}$ is $\mathsf{N}_{\mathcal{C}} = \mathsf{M}$. Refer to Figure 5.19 for an example.

- Case C5: $\mathfrak{O}_{\mathcal{C}} \setminus (\mathcal{C} \overset{*}{\cap} \mathsf{N}) = \emptyset \ \wedge \ \mathfrak{O}_{\mathsf{A}_{best}} \not\subset \mathfrak{O}_{\mathcal{C}}$
  A new node L is created and added to the shrubs. The parent node of L is the parent node of A_{best} and L is set as the new parent node of A_{best}. The node content of A_{best} is replaced by $\mathfrak{O}_{A_{best}} \setminus \mathfrak{O}_{\mathcal{C}}$. The node content of L is $\mathfrak{O}_{\mathsf{L}} = \mathfrak{O}_{\mathcal{C}} \cap \mathfrak{O}_{A_{best}}$ and the associated node of the cell $\mathcal{C}$ is $\mathsf{N}_{\mathcal{C}} = \mathsf{L}$. Refer to Figure 5.20 for an example.

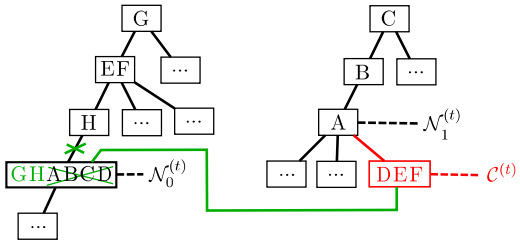


Fig. 5.22: Connecting the cell $\mathcal{C}^{(t)}$ with cell content $\mathfrak{O}_{\mathcal{C}^{(t)}}$ =ABCDEF to the shrubs (red) and reconnecting the neighbor cell $\mathcal{N}_0^{(t)}$ with cell content $\mathfrak{O}_{\mathcal{N}_0^{(t)}}$ =ABCDEFGH in order to increase the number of referenced primitives in the shrubs (green).

- Case C6: $\mathfrak{O}_{\mathcal{C}} \setminus (\mathcal{C} \overset{*}{\cap} \mathsf{N}) \neq \emptyset \ \wedge \ \mathfrak{O}_{\mathsf{A}_{best}} \not\subset \mathfrak{O}_{\mathcal{C}}$
  Two new nodes L and M are created and added to the shrubs. The parent node of L is the parent node of A_{best} and L is set as the new parent node of A_{best}. The node content of A_{best} is replaced by $\mathfrak{O}_{A_{best}} \setminus \mathfrak{O}_{\mathcal{C}}$. The node content of L is $\mathfrak{O}_{\mathsf{L}} = \mathfrak{O}_{\mathcal{C}} \cap \mathfrak{O}_{A_{best}}$. The parent node of M is L. The node content of M is $\mathfrak{O}_{\mathsf{M}} = \mathfrak{O}_{\mathcal{C}} \setminus (\mathcal{C} \overset{*}{\cap} \mathsf{N})$ and the associated node of the cell $\mathcal{C}$ is $\mathsf{N}_{\mathcal{C}} = \mathsf{M}$. Refer to Figure 5.21 for an example.

Dependency on the Connection Order

The free constructed shrubs depend on the insertion sequence of the cells. When a cell $\mathcal{C}^{(t)}$ is connected to the shrubs as explained above, only the node chains of the already connected neighboring cells at time t are considered for the determination of the best fitting node chain. For example, reconsider the situation given in Figure 5.16. The cell $\mathcal{C}^{(t)}$ is connected at time t and the neighboring cells $\mathcal{N}_0^{(t)}$ and $\mathcal{N}_1^{(t)}$ have been connected previously. Assumed, $\mathcal{N}_0^{(t)}$ would have been connected after $\mathcal{C}^{(t)}$ at time $t+\Delta$, the shrubs would look different. $\mathcal{C}^{(t)}$ would still be connected with a new node that becomes a child node of the associated node of its only already connected neighboring cell $\mathcal{N}_1^{(t)}$. Later $\mathcal{C}^{(t+\Delta)} = \mathcal{N}_0^{(t)}$ would be connected. One of its neighbor cell is $\mathcal{N}_0^{(t+\Delta)} = \mathcal{C}^{(t)}$. $\mathcal{C}^{(t+\Delta)}$ can be connected with a new node that becomes a child node of the associated node of $\mathcal{N}_0^{(t+\Delta)}$. So the shrubs would look different – and the total number of primitives would be smaller. In order to compensate the dependency on the connection order to a certain extent, we check after each connection of a cell $\mathcal{C}^{(t)}$ whether the shrubs can be improved by reconnecting the neighboring cells. The situation in Figure 5.16 can be improved as shown in Figure 5.22 in green color. How this works in general is presented in the following.

Reconnection

Let $\mathcal{F}$ be the last connected cell $\mathcal{C}^{(t)}$ or the last reconnected cell in the shrubs and let $\mathcal{R}$ be an already connected neighboring cell we want to reconsider.[1] The precondition for a reconnection is that the associated nodes $\mathsf{N}_{\mathcal{R}}$ and $\mathsf{N}_{\mathcal{F}}$ of the cells $\mathcal{R}$ and $\mathcal{F}$ are not equal and the node $\mathsf{N}_{\mathcal{R}}$ is no ancestor node of $\mathsf{N}_{\mathcal{F}}$, thus

$$\mathsf{N}_{\mathcal{R}} \notin A^{+}(\mathsf{N}_{\mathcal{F}}) \ . \tag{5.9}$$

[1] Notation hint: $\mathcal{F}$ stands for the ***fixed*** cell and $\mathcal{R}$ stands for the ***reconsidered*** cell.

The number of primitive references in the shrubs can be decreased, if

$$\left|\mathcal{R} \overset{*}{\cap} \mathsf{N}_{\mathcal{F}}\right| \; > \bigcup_{\mathsf{A} \in A(\mathsf{N}_{\mathcal{R}})} |\mathfrak{O}_{\mathsf{A}}| \; . \tag{5.10}$$

The left side of the inequality is the saving of primitives references after a reconnection of node $\mathsf{N}_{\mathcal{R}}$, the right side is the saving of primitives references for the current connection of $\mathsf{N}_{\mathcal{R}}$.

In this case this is fulfilled, the number of primitive references in the shrubs is decreased by $\left|\mathcal{R} \overset{*}{\cap} \mathsf{N}_{\mathcal{F}}\right| - \bigcup_{\mathsf{A} \in A(\mathsf{N}_{\mathcal{R}})} |\mathfrak{O}_{\mathsf{A}}|$ primitive references after reconnecting the node $\mathsf{N}_{\mathcal{R}}$.

Refer again to the situation in Figure 5.22. After the cell $\mathcal{C}^{(t)}$ is connected (red), we reconsider all already connected neighboring cells of $\mathcal{C}^{(t)}$. Using the notations above, we first consider $\mathcal{F} = \mathcal{C}^{(t)}$ and $\mathcal{R} = \mathcal{N}_1^{(t)}$. In this case Equation 5.9 is not fulfilled and $\mathsf{N}_{\mathcal{N}_1^{(t)}}$ is not reconnected. Considering the other neighboring cell $\mathcal{R} = \mathcal{N}_0^{(t)}$. Here Equation 5.9 and Equation 5.10 are fulfilled. For Equation 5.10 we have $6 > 4$. Thus, the node $\mathsf{N}_{\mathcal{N}_0^{(t)}}$ is reconnected as displayed with green color. By this reconnection the number of primitive references in the shrubs is decreased by 2.

Reconnection Cases

Similar to as for the connection of a single grid cell different cases to be considered for the reconnection. Again let $\mathcal{F}$ be the last connected cell or the last reconnected cell in the shrubs and let $\mathcal{R}$ be an already connected neighboring cell of $\mathcal{F}$ we want to reconsider. Further let A_{best} be the best fitting node of $\mathcal{R}$ in the node chain $A^{+}_{ord}(\mathsf{N}_{\mathcal{F}})$.

- Case R1: $\mathfrak{O}_{\mathcal{R}} \setminus (\mathcal{R} \overset{*}{\cap} \mathsf{N}_{\mathcal{F}}) = \emptyset \; \wedge \; \mathfrak{O}_{\mathsf{A}_{best}} \subset \mathfrak{O}_{\mathcal{R}}$
  The node A_{best} is set as the new parent node of $\mathsf{N}_{\mathcal{R}}$. The node content of $\mathsf{N}_{\mathcal{R}}$ is empty. Refer to Figure 5.23 for an example.

In this case the empty node $\mathsf{N}_{\mathcal{R}}$ could actually be deleted. Then A_{best} must be set as the new parent node of all child nodes of $\mathsf{N}_{\mathcal{R}}$ and further A_{best} must be set as the associated cell in all cells in $Z(\mathsf{N}_{\mathcal{R}})$. Our benchmarks showed that the case R1 is a rare case. Since we do not want to use bidirectional connections in the shrubs for memory reasons and since we do not want to search trough all nodes to find all child nodes of $\mathsf{N}_{\mathcal{R}}$ and through all cells to find all cells in $Z(\mathsf{N}_{\mathcal{R}})$, we prefer to have these few empty nodes within the shrubs.

- Case R2: $\mathfrak{O}_{\mathcal{R}} \setminus (\mathcal{R} \overset{*}{\cap} \mathsf{N}_{\mathcal{F}}) \neq \emptyset \; \wedge \; \mathfrak{O}_{\mathsf{A}_{best}} \subset \mathfrak{O}_{\mathcal{R}}$
  The node A_{best} is set as the new parent node of $\mathsf{N}_{\mathcal{R}}$. The node content of $\mathsf{N}_{\mathcal{R}}$ is $\mathfrak{O}_{\mathsf{N}_{\mathcal{R}}} = \mathfrak{O}_{\mathcal{R}} \setminus (\mathcal{R} \overset{*}{\cap} \mathsf{N}_{\mathcal{F}})$. Refer to Figure 5.24 for an example.

- Case R3: $\mathfrak{O}_{\mathcal{R}} \setminus (\mathcal{R} \overset{*}{\cap} \mathsf{N}_{\mathcal{F}}) = \emptyset \; \wedge \; \mathfrak{O}_{\mathsf{A}_{best}} \not\subset \mathfrak{O}_{\mathcal{R}}$
  The new parent node of $\mathsf{N}_{\mathcal{R}}$ is the parent node of A_{best} and $\mathsf{N}_{\mathcal{R}}$ is set as the new parent node of A_{best}. The node content of A_{best} is replaced by $\mathfrak{O}_{A_{best}} \setminus \mathfrak{O}_{\mathcal{R}}$. The node content of $\mathsf{N}_{\mathcal{R}}$ is $\mathfrak{O}_{\mathsf{N}_{\mathcal{R}}} = \mathfrak{O}_{\mathcal{R}} \cap \mathfrak{O}_{A_{best}}$. Refer to Figure 5.25 for an example.

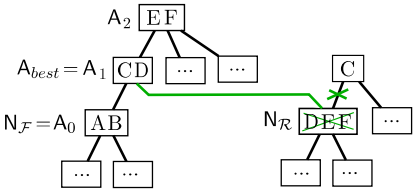


Fig. 5.23: Example for case R1: Reconnecting the associated node of cell $\mathcal{R}$ with $\mathfrak{O}_{\mathcal{R}}$ =CDEF to the node chain of its neighboring cell $\mathcal{F}$.

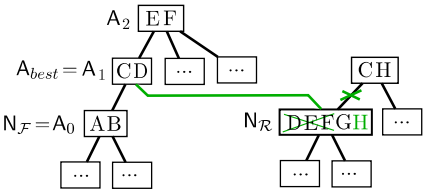


Fig. 5.24: Example for case R2: Reconnecting the associated node of cell $\mathcal{R}$ with $\mathfrak{O}_{\mathcal{R}}$ =CDEFGH to the node chain of its neighboring cell $\mathcal{F}$.

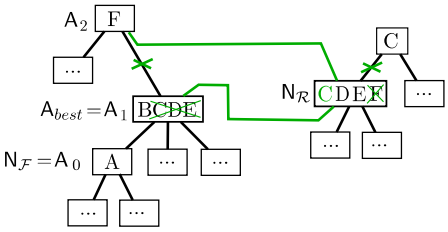


Fig. 5.25: Example for case R3: Reconnecting the associated node of cell $\mathcal{R}$ with $\mathfrak{O}_{\mathcal{R}}$ =CDEF to the node chain of its neighboring cell $\mathcal{F}$.

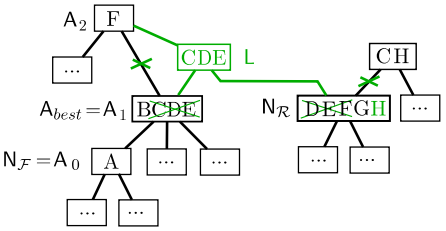


Fig. 5.26: Example for case R4: Reconnecting the associated node of cell $\mathcal{R}$ with $\mathfrak{O}_{\mathcal{R}}$ =CDEFGH to the node chain of its neighboring cell $\mathcal{F}$.

- Case R4: $\mathfrak{O}_{\mathcal{R}} \setminus (\mathcal{R} \overset{*}{\cap} \mathsf{N}_{\mathcal{F}}) \neq \emptyset \ \wedge \ \mathfrak{O}_{\mathsf{A}_{best}} \not\subset \mathfrak{O}_{\mathcal{R}}$
  A new node L is created and added to the shrubs. The parent node of L is the parent node of A_{best} and L is set as the new parent node of A_{best}. The node content of A_{best} is replaced by $\mathfrak{O}_{A_{best}} \setminus \mathfrak{O}_{\mathcal{R}}$. The node content of N_{L} is $\mathfrak{O}_{\mathsf{L}} = \mathfrak{O}_{\mathcal{R}} \cap \mathfrak{O}_{A_{best}}$. The parent node of $\mathsf{N}_{\mathcal{R}}$ is L. The node content of $\mathsf{N}_{\mathcal{R}}$ is $\mathfrak{O}_{\mathsf{N}_{\mathcal{R}}} = (\mathfrak{O}_{\mathcal{R}} \setminus \mathcal{R} \overset{*}{\cap} \mathsf{N}_{\mathcal{F}})$. Refer to Figure 5.26 for an example.

5.4.3. Connection Order and Process

A Greedy Construction Process with Local Optimization

We present an incremental greedy construction process with local optimization. The principle of the construction process is that all occupied grid cells of a TD-grid are connected incrementally to the shrubs. Our therefor presented connection process (Section 5.4.1 and 5.4.2) is a greedy approach. After a cell is connected, the already connected cells in its neighborhood are double checked whether their connection can be optimized by reconnections (Section 5.4.2).

Connection Order

Since the goal is to create the shrubs with least possible primitive references, the goal of the greedy algorithm is to add, in every single connection step, as least as possible primitive references to the shrubs. We assume, similar as for the schematic construction, that cells in close proximity have a similar cell content. If we start with an arbitrary cell in the grid, it is reasonable to continue with cells in close proximity to the cell we started with. Thus, it is reasonable to continue with a neighboring cell.

Another aspect is that it seems to be advantageous to connect cells with small cell content before cells with large cell content. This is because our connection process makes only local modifications to the shrubs. Connecting a cell with only a slightly larger content than the already connected cells is only a small extension and therefore usually a good choice for our greedy algorithm. Generally, cells with small cell content are easier to connect to the shrubs than cells with a large cell content.

Cells with a small cell content are usually cells near the boundary of all occupied grid cells. Cells with a large cell content are usually cells in the inner of all occupied grid cells (refer to Figure 5.27). Giving cells with small cell content priory to be connected next to the shrubs is like processing the cells on a wavefront that starts with the boundary cells and continuous towards the inner cells. Boundary cells have less occupied neighboring cells and more often neighboring cells with similar or even equal cell content. Thus, the chance for false decisions of the greedy connection process is not that high. Inner cells on the wavefront usually have one half of its neighboring cells already connected. Thus, about 13 neighboring cells can be evaluated to find the best fitting node chain. The number is even higher towards the end of the construction process, when the wavefront meets itself. So the chance to make false decisions is about 1/2 or smaller.

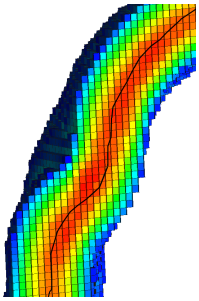


Fig. 5.27: Color code visualization of the number of referenced primitives per cell in an intersection through a 3d-surface (red = many, blue = less).

Comparing this approach with the strategy to connect cells with large cell content before cells with small cell content, we would process the cells on a wavefront from the inner of all occupied cells towards the boundary. Cells with large cell content usually have solely occupied neighbor cells, thus 26 possibilities to connect to the shrubs. In the beginning cells with large content would have no or only few already connected neighbor cells to evaluate the best fitting node chain and thereby have low chance to have the best possible neighbor to connect with. We will later show that the reconnection can repair false decisions until a certain degree – but it is always better to avoid failures in the beginning and connect in the best possible manner. Nevertheless, we also have implemented the variant where we connect cells with large content before cells with small cell content in order to prove the effectiveness of connecting cells with small cell content before cells with large cell content. Further, we implemented a third variant that randomly chooses the next cell to connect to the shrubs. We will show that the randomized variant is also able to produce good compression rates.

The Construction Algorithms

Algorithm 10 shows the free construction process with priority based cell selection. The free construction process with randomized cell selection is shown in Algorithm 11. Both algorithms start with a preprocessing where the TD-grid is created (line 1). But in contrast to the creation of the TD-grid as described in Chapter 3, the primitive references are not stored in the grid cells. After the shrubs are completed, each grid cell will reference one node in the shrubs which is its associated node. At query time the referenced primitives can be determined as given in Proposition 5.2. For the randomized construction, the grid cells of the constructed TD-grid are empty in the beginning. For the priority based construction the grid cells are empty, too, but the number of referenced primitives per cell has to be calculated additionally in order use it for the priory rule. This is an extra effort, because in line 9 of Algorithm 10 the referenced primitive per cell are determined again. If the TD-grid is small enough to be hold in memory without using shrubs, the cell content could be memorized by the preprocessing

and reused in line 9. Otherwise, the referenced primitives per cell are only determined to obtain the number of referenced primitives in the preprocessing (line 1) and later in line 9 calculated again in order to create the shrubs.

The priority based construction algorithm works with two priority queues because we have two priorities to combine:

- Cells with small(large) cell content have a high priority.
- Neighboring cells of already connected cells have a high priority.

The first priority queue uses a quantity priority which is unchanged during the whole construction process. The second priority queue uses a priority that depends on the local neighborhood of a cell and is frequently changed during the construction process.

We formulate our rule to select the next cell to be connected as follows:

> *Select from the cells, which have at least one already connected neighbor cell, the one with the smallest(largest) cell content – provided its cell content is smaller (larger) or equal to the smallest(largest) cell content of the cells, which have no connected neighbor cell so far. Otherwise select from the cells, which have no connected neighbor cell so far, the one with the smallest(largest) cell content.*

This selection is implemented by the two priorities queues PQ1, PQ2 and the doubled while-loop in Algorithm 10, line 4 and line 6. After a cell is processed, all its not yet connected neighbors are pushed to the priority queue PQ1 (line 24).

The randomized selection is much easier. There only one queue is used, where all cells are shuffled in the preprocessing before all cells in the queue are processed (refer to Algorithm 11, line 4.

The inner of the while loop(s) is identical for both algorithms. The pseudo code therefor is given in Algorithm 10 line 8 to line 22. For a processed cell $\mathcal{C}$ first the cell content $\mathfrak{O}_{\mathcal{C}}$ is obtained (line 9) as well as the set of all its already connected neighbor cells (line 10). The cell $\mathcal{C}$ is connected to the shrubs according to the distinction of cases given in Section 5.4.2. In the case there is no already connected neighbor cell, the cell is added to the shrubs according to case C1. Otherwise the maximal node chain intersection is determined (line 14 to 16). If the maximal node chain intersection is empty, the cell is added to the shrubs according to case C2 (line 17). Otherwise the cell is connected with the best fitting node chain according to case C3, C4, C5, or C6 (line 18). After the cell $\mathcal{C}$ is connected to the shrubs, a local optimization is performed (line 22).

The local optimization is given by Algorithm 12. The input cell $\mathcal{C}$ is directly pushed to the queue $Q_{\mathcal{F}}$ (line 1). For each cell $\mathcal{F}$ of the queue $Q_{\mathcal{F}}$ the associated nodes of all already connected neighboring cells of $\mathcal{F}$ are tested on reconnection. In the case that the preconditions for a reconnection are fulfilled (line 10 and 12) for a neighboring cell $\mathcal{R}$, its associated node $\mathsf{N}_{\mathcal{R}}$ is reconnected to the node chain of $\mathsf{N}_{\mathcal{F}}$ (line 13). The reconnection is done accordingly to the cases R1, R2, R3, and R4 as explained in Section 5.4.2. Finally, if $\mathsf{N}_{\mathcal{R}}$ was reconnected, $\mathcal{R}$ is pushed to $Q_{\mathcal{F}}$ (line 14) such that its neighbor cells can be tested in the same way.

Algorithm 10: Free Construction Process with Priority Based Cell Selection

Data: $\mathfrak{O}$,δ,w // the complex object, the tolerance value and the cell width

Result: $\mathcal{G}(\mathfrak{O})$ // the uniform TD-grid
$\mathcal{S}(\mathfrak{O})$ // the shrubs

1 $\mathcal{G}(\mathfrak{O}) \leftarrow$ create the TD-grid for the object $\mathfrak{O}$ with cell width w and tolerance value δ, where the cells store, instead of primitive references, only the number of primitive references
2 PQ1,PQ2 $\leftarrow$ create two priority queues for grid cells
3 push all occupied grid cells in $\mathcal{G}(\mathfrak{O})$ to PQ2
4 **while** PQ2 is not empty **do**
5 PQ1.push(PQ2.top)
6 **while** PQ1 is not empty $\wedge$ prio PQ1.top $\geq$ prio PQ2.top **do**
7 $\mathcal{C} \leftarrow$ PQ1.top
8 mark $\mathcal{C}$ as connected
9 $\mathfrak{O}_{\mathcal{C}} \leftarrow$ calculate all primitive references that must be referenced by $\mathcal{C}$
10 $\{\mathcal{N}_0, ..., \mathcal{N}_n\} \leftarrow$ get the already connected first ring neighbor cells of $\mathcal{C}$
11 **if** $\{\mathcal{N}_0, ..., \mathcal{N}_n\} = \emptyset$ **then**
12 connect $\mathcal{C}$ to $\mathcal{S}(\mathfrak{O})$ $\hookrightarrow$ case C1
13 **else**
14 **foreach** $\mathcal{N}_i \in \{\mathcal{N}_0, ..., \mathcal{N}_n\}$ **do**
15 $I_i \leftarrow$ calculate the node chain intersection $\mathcal{C} \overset{*}{\cap} \mathsf{N}_{\mathcal{N}_i}$
16 $best \leftarrow$ the smallest index i with $|I_i| = \max_i \{|I_i|\}$
17 **if** $|I_{best}| = 0$ **then**
18 connect $\mathcal{C}$ to $\mathcal{S}(\mathfrak{O})$ $\hookrightarrow$ case C2
19 **else**
20 $A^+_{ord}(\mathsf{N}_{\mathcal{N}_{best}}) \leftarrow$ the best fitting node chain
21 connect $\mathcal{C}$ to the node chain $A^+_{ord}(\mathsf{N}_{\mathcal{N}_{best}})$ $\hookrightarrow$ case C3 - C6
22 **LocalOptimization**($\mathcal{C}$, $\mathcal{S}(\mathfrak{O})$)
23 pop all already connected cells from the top of PQ1
24 push all not yet connected first ring neighbor cells of $\mathcal{C}$ on PQ1
25 pop all already connected cells from the top of PQ2

Algorithm 11: Free Construction Process with Randomized Cell Selection

Data: $\mathfrak{O}$,δ,w // the complex object, the tolerance value and the cell width

Result: $\mathcal{G}(\mathfrak{O})$ // the uniform TD-grid
$\mathcal{S}(\mathfrak{O})$ // the shrubs

1 $\mathcal{G}(\mathfrak{O}) \leftarrow$ create the TD-grid for the object $\mathfrak{O}$ with cell width w and tolerance value δ, where the cells store nothing
2 Q $\leftarrow$ create a queue
3 push all occupied grid cells in $\mathcal{G}(\mathfrak{O})$
4 shuffle elements in Q
5 **while** Q is not empty **do**
6 $\quad \mathcal{C} \leftarrow$ Q.top
7 $\quad$ ... // identical with Alg. 10 line 8 to line 22
8 $\quad$ pop all already connected cells from the top of Q

Algorithm 12: LocalOptimization($\mathcal{C}$, $\mathcal{S}(\mathfrak{O})$)

Data: $\mathcal{C}$ // a cell, which is already connected to the shrubs
$\mathcal{S}(\mathfrak{O})$ // the shrubs

Result: $\mathcal{S}(\mathfrak{O})$ // the modified shrubs

1 $Q_{\mathcal{F}} \leftarrow$ create queue and push $\mathcal{C}$ to $Q_{\mathcal{F}}$
2 **while** $Q_{\mathcal{F}}$ is not empty **do**
3 $\quad \mathcal{F} \leftarrow Q_{\mathcal{F}}$.front // the cell with whose node chain we compare
4 $\quad Q_F$.pop
5 $\quad \{\mathcal{N}_0, ..., \mathcal{N}_n\} \leftarrow$ get the already connected first ring neighbor cells of $\mathcal{F}$
6 $\quad Q_{\mathcal{R}} \leftarrow$ create queue and push all cells in $\{\mathcal{N}_0, ..., \mathcal{N}_n\}$ to Q_R
7 $\quad$ **while** $Q_{\mathcal{R}}$ is not empty **do**
8 $\quad\quad \mathcal{R} \leftarrow Q_{\mathcal{R}}$.front // the cell whose associated node is possibly reconnected
9 $\quad\quad Q_{\mathcal{R}}$.pop
10 $\quad\quad$ **if** $\mathsf{N}_{\mathcal{R}} \notin A^{+}(\mathsf{N}_{\mathcal{F}})$ **then**
11 $\quad\quad\quad I \leftarrow$ calculate the node chain intersection $\mathcal{R} \overset{*}{\cap} \mathsf{N}_{\mathcal{F}}$
12 $\quad\quad\quad$ **if** $|I| > \bigcup_{\mathsf{A} \in A(\mathsf{N}_{\mathcal{R}})} |\mathfrak{O}_{\mathsf{A}}|$ **then**
13 $\quad\quad\quad\quad$ reconnect node $\mathsf{N}_{\mathcal{R}}$ to the node chain $A^{+}_{ord}(\mathsf{N}_{\mathcal{F}}) \hookrightarrow$ case R1 - R4
14 $\quad\quad\quad\quad$ $Q_{\mathcal{F}}$.push($\mathcal{R}$)

5.5. Implementation Details with regards to the Memory Compression

In the beginning of this chapter we showed that the memory consumption of a TD-grid is many times higher than the one of a TI-grid of the same cell width. The purpose of the introduced shrubs data structure was to reduce the number of primitive references in a TD-grid. We showed in Section 5.1 that the largest memory portion of a TD-grid is spend on storing primitive references and that we are able to save primitive references by using the shrubs data structure. However, not only the compression ratio of primitive references is interesting – finally the compression ratio on the total memory consumption of the compressed TD-grid is interesting.

Definition 5.8 (Compression Ratio of the Compressed TD-Grid): *The compression ratio of the compressed TD-grid is defined as*

$$\Theta_M := \frac{\mathsf{Memory}_{Compressed}}{\mathsf{Memory}_{Uncompressed}} \,. \tag{5.11}$$

Please note that the term compression ratio is often defined ambiguously in literature. With our definition we follow the one given by Salomon [46]. For example a compression ratio of 0.6 means that the date is compressed to 60% of the original size. Thus, 40% of the space is saved. The smaller the compression ratio, the better is the compression.

Generally, the grid memory is composed out of

- the memory for the data structure that maintains the occupied cells
- and the memory for the data structure that maintains the cell content which is composed of
  - the memory for all primitive references, i.e. the memory of the cell contents,
  - and the overhead of the data structure itself.

Refer to Section 4.2 where we already have explained this data structures in general. The memory consumption of the above listed components is in detail as follows:

The memory for the data structure that maintains the occupied cells: For a given cell width w and a given tolerance value δ the memory consumption is equal for an uncompressed TD-grid and for a compressed TD-grid. The memory consumption depends on the type of data structure we use, as well as on the number of leafs of the hierarchical part of our combined data structure (refer again to Section 4.2). Since we are using 3d-arrays to maintain the occupied cells, 4 bytes per array entry are required plus a small overhead per 3d-array (in our implementation 10×4 bytes). The number of array entries is larger, but not much larger, than the number of occupied cells (refer for an example to Figure 4.2 in Section 4.2).

The memory for all primitive references: For every primitive reference 4 bytes are required. This is one integer value per primitive reference.

The memory overhead of the data structure that maintains the cell content: For an uncompressed TD-grid it consists of only 4 additional bytes per cell, if an array-based data structure is used (refer again to Section 4.2). For a compressed TD-grid we use the shrubs data structure, which is much more complicated as the simple data structure that is used by the uncompressed TD-grid. As it is the purpose of the shrubs data structure to save memory by saving stored primitive references, it is essential that the memory that is used by the shrubs data structure itself is possibly small. In order to achieve a memory space saving at all, the memory overhead of the shrubs data structure must not be larger than the saved memory. If one does not take care for a memory efficient data structure to implement the shrubs, it is possible that the memory overhead is larger than the saved memory due to saved primitive references. Thus, we focus in this section a memory efficient way to implement the shrubs.

The minimal requirement of a node in the shrubs is to save the primitives references and to save a pointer to the next node. Similar as the array-based cell data structure of an uncompressed grid, we use an array-based data structure for the compressed grid, too. Thereby we have to consider different issues for the schematic and for the free constructed variants of the shrubs.

Schematic Constructed Shrubs. As presented in Section 5.3, for the schematic constructed shrubs a complete tree is created. We called this tree the construction tree. Usually, the complete tree is not necessary to maintain the cell content of all grid cells because there are many nodes with empty content. Nodes that have an empty content do not contribute to the cell content of any grid cell (refer Proposition 5.2 and to the beginning of Section 5.3). Further, a typical data structure of a tree consists of nodes which points to their child nodes. For the grid query, where we request a cell content, we need the opposite direction. Thus, a node in the shrubs should point to its parent node in order to obtain the union of the contents of a cell's associated node and all its ancestor nodes.

For convenience we have not distinguished in Section 5.3 between the construction tree and the actual shrubs. Now we will precise this statement. For our memory efficient solution we build the construction tree and the shrubs as two separate data structures in parallel. Algorithm 7, 8 and 9 are specified more precisely as follows: The nodes that are created in these algorithms [2] are nodes of the construction tree. A node in the shrubs is a light weighted node that has only one pointer to its parent node and that stores the node content. A shrubs node is created when in Algorithm 7 a node of the construction tree is completed (line 12) and the set $I_i \setminus I$ is not empty. Instead of setting the content $I_i \setminus I$ as content of the construction tree node, it is set as the content of the newly created shrubs node. For the OBT created shrubs (Algorithm 8 and 9) it is similar but slightly more complex. Nodes of the shrubs are created after the nodes of T_{best} are integrated into the construction tree (line 16). For each 1^{st}, 2^{nd} and 3^{rd}-level

[2] Algorithm 7 line 7, Algorithm 8 line 7 and Algorithm 9 line 7

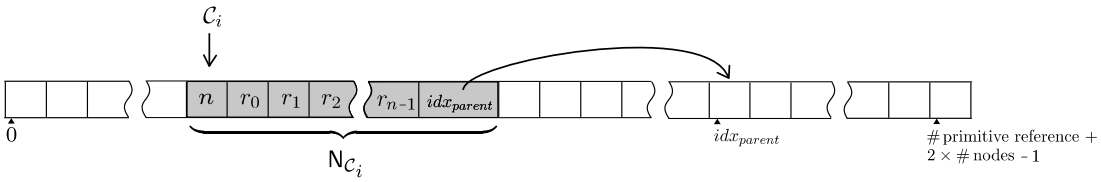


Fig. 5.28: Sketch of the array-based data structure to maintain the cell content of schematic constructed shrubs. A node consists of one integer n that denotes the number of primitive references that are stored in the node, an integer value for each of its n primitive references $r_0, ..., r_{n-1}$ and an integer value idx_{parent} for the array position where the parent node starts.

node in T_{best} that has a non-empty content a new shrubs node is created. After creating a new shrubs node, the construction tree data structure is used to find already created shrubs nodes that must have the new created shrubs node as parent node. After that, or when I is empty, all descendants of the current node of the construction tree are not needed anymore and are deleted. In this way we will never have the whole construction tree in memory. Actually, as our recursion build the shrubs in a depth-first manner, we have always only a very small part of the construction tree in memory.

The array-based data structure for the shrubs is defined as an integer array. Refer to Figure 5.28 for a sketch. A node in the shrubs is represented by a sequence of array elements. Each grid cell stores an integer value that denotes the start position of its associated node within the array. A node consists of one integer value that denotes the number of primitive references that are stored in the node, an integer value for each of its stored primitive reference and an integer value for the array position where the parent node starts. In the case the node has no parent node, the value of this last integer is -1. The length of this array is the total number or primitive references plus two times the number of nodes in the shrubs.

Free Constructed Shrubs. The array-based data structure to maintain the cell content for the free constructed shrubs is very similar to the one of the schematic constructed shrubs. An additionally data structure during the construction process is not required here. The challenge is that we need a dynamic data structure that allows, beside adding new nodes, also to delete primitive references within nodes. This is necessary to handle the cell connection cases C5 and C6 as well as all reconnection cases R1-R4 (refer to Section 5.4.2).

Deleting one primitive reference of a node at position x means that all entries in the whole array after position x are moved by one positions towards the front of the array. This is problematic for the array-based data structure as introduced above for the schematic constructed shrubs. The reason is that the array stores entries indicating an array index. Each node in the shrubs uses such an index to denote the index where its parent node starts. All array entries that store an array index would become wrong in the case the index is larger than x. So we need an update mechanism that reduces the value of all those array entries that store an array index larger than x.

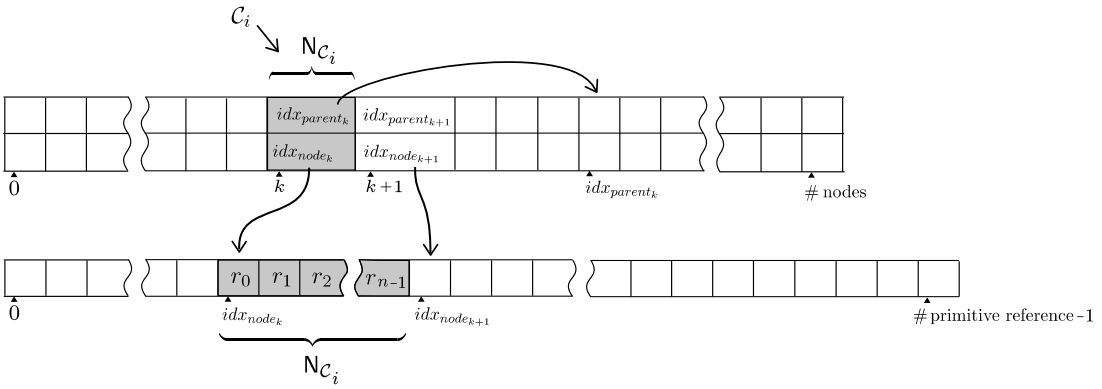


Fig. 5.29: Sketch of the array-based data structure to maintain the cell content of free constructed shrubs. A node consists of one integer idx_{node_k} that denotes the start position of the primitive references in another array, one integer idx_{parent_k} that denotes the start position of the parent node and an integer value for each of its n primitive references $r_0, ..., r_{n-1}$, where $n = idx_{node_k+1} - idx_{node_k}$.

This could be done by scanning the whole array and decreasing all array entries that store an array index larger than x. Alternatively, a second array can be introduced in the way as it is sketched in Figure 5.29. The shrubs are represented by a 1d- and a 2d-array, where the 2d-array has only two rows. A node in the shrubs is represented by a sequence of array elements in the 1D-array and a column in the 2d-array. Each grid cell stores an integer value that denotes the column of its associated node within the 2d-array. This could be done by scanning the whole array and decreasing all array entries that store an array index larger than x. Alternatively, a second array can be introduced in the way as it is sketched in Figure 5.29. The shrubs are represented by a 1d- and a 2d-array, where the 2d-array has only two rows. A node in the shrubs is represented by a sequence of array elements in the 1D-array and a column in the 2d-array. Each grid cell stores an integer value that denotes the column of its associated node within the 2d-array.

A node consists of a sequence of integer values within the 1d-array, one for each of its primitive references, and of the two integer values in one column of the 2d-array. The one in the first row denotes the column index of the node's parent node and the one in the second row denotes the start position of the primitive references in the 1d-array. The number of primitive references per node is given by the difference between the node's second-row-value and the following node's second-row-value. In the case the node has no parent node, its first-row-value is -1. The length of the 1d-array is the total number of primitive references. The length of the 2d-array is the number of nodes + 1. There is one additional column in order to calculate the number or primitive references per node uniquely for all nodes, thus, the second-row-value in the last column is the number of total primitive references.

When a primitive reference of a node at position x is deleted, all array entries in the 1d-array after position x are moved by one positions towards the front of the array. Then the second row of the 2d-array is scanned and all values that are larger than x are reduced by one.

Summarized. Essentially, the memory consumption of both shrubs variants is the memory consumption of all primitive references (4 bytes per primitive reference) plus the memory overhead to maintain the cell contents (2 times 4 bytes per node in the shrubs).

5.6. Results

In this section the compression results for the schematic constructed shrubs as well as for the free constructed shrubs are presented, analyzed and compared.

In order to verify the construction variants we use the bodyshell of the *Engine* benchmark (refer to Appendix A.1) and the Stanford *Bunny* (refer to Appendix A.3). Both models are quite different in their shape. The bunny consists of almost equal sized triangles, where the surface is almost a closed 2-manifold (there are only few holes). The bodyshell consists of varying sized triangles, has many holes, many loose triangles and some edges with more than two adjacent triangles. We want to see how the different construction variants perform on these two different types of models. Beside these benchmarks, we will also use some sub-parts of the bodyshell of the *Engine* benchmark (refer to Appendix A.1.4) for some deeper analysis.

For the created grids in the benchmarks, we use the cell width as they were used in the performance benchmarks in Chapter 4.

First, in Section 5.6.1 and Section 5.6.2 as well as in the beginning of Section 5.6.3 we compare the achieved compression ratio of primitive references Θ_P. So we have clean benchmarks to compare the compression quality of our approaches without considering the memory overhead of the data structure that maintains the cell content. In Section 5.6.3, where we compare our best schematic and our best free constructed variant, we finally consider the achieved memory compression ratio Θ_M.

5.6.1. Schematic Construction Variants

In this section we benchmark and compare the achieved compression ratio of primitive references Θ_P for the three schematic constructed variants, which were presented in Section 5.3: The octree constructed shrubs, the strict binary tree (SBT) constructed shrubs, and the optimal binary tree (OBT) constructed shrubs.

The top diagram in Figure 5.30 displays the compression ratio of referenced primitives that is achieved for the bodyshell in the *Engine* scenario. We are able to achieve for every tolerance value δ a compression ratio smaller than 0.2 – at most for δ =30 with 0.14. This means that we need less than 1/5 of the primitive references to store or, equally, that more than 80% of the original primitive references are saved.

Equally good compression ratios can be observed for the Stanford bunny in the bottom diagram of Figure 5.30. Here the compression ratio is always under 0.3 and at most 0.16.

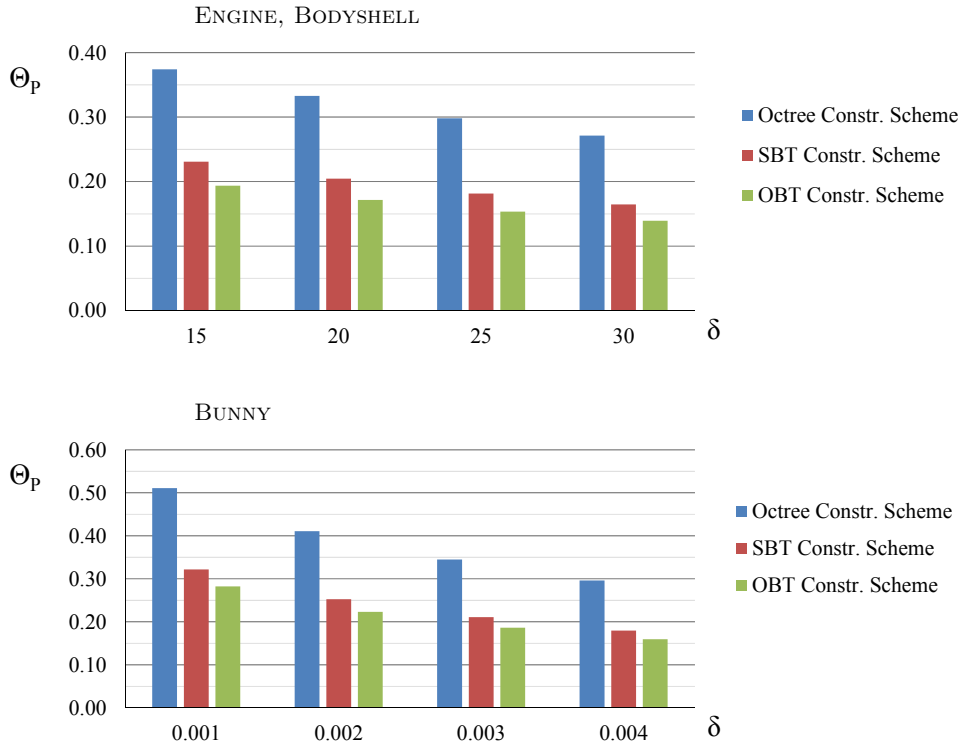


Fig. 5.30: Compression ratio of referenced primitives Θ_P with the schematic constructed shrubs for the bodyshell of the *Engine* (top) and for the Stanford Bunny (bottom) for varying tolerance value δ.

For all δ the OBT construction scheme is the one with the best compression ratio. Moreover, the ranking between the variants stays constant for all δ. The second best variant is the SBT construction scheme and the third best is the octree construction scheme. That the OBT construction scheme must have a better compression ratio than the SBT construction scheme is already clear by the definition of the OBT and the SBT construction schemes (refer also to Section 5.3.3). Further, the two binary tree construction schemes must achieve better compression ratio than the octree construction scheme because of the intermediate steps that connects two cells until a block of eight cells is reached. In each intermediate step there is an additional possibility to save primitive references – which is not given for the octree scheme (refer also to Section 5.3.2). Nevertheless, the octree scheme is surely the easiest to implement, directly followed by the SBT construction scheme. The OBT construction scheme is the most complicated to implement.

Almost similar compression ratios are achieved for the bodyshell of the *Engine* scenario and the Stanford bunny. This is a notable issue as the models are very different in shape. Although the results are not directly comparable, for our chosen tolerance values the compression ratio for the bodyshell is slightly better than for the one of the Stanford bunny. So we expect that for further models similar compression ratios can be achieved.

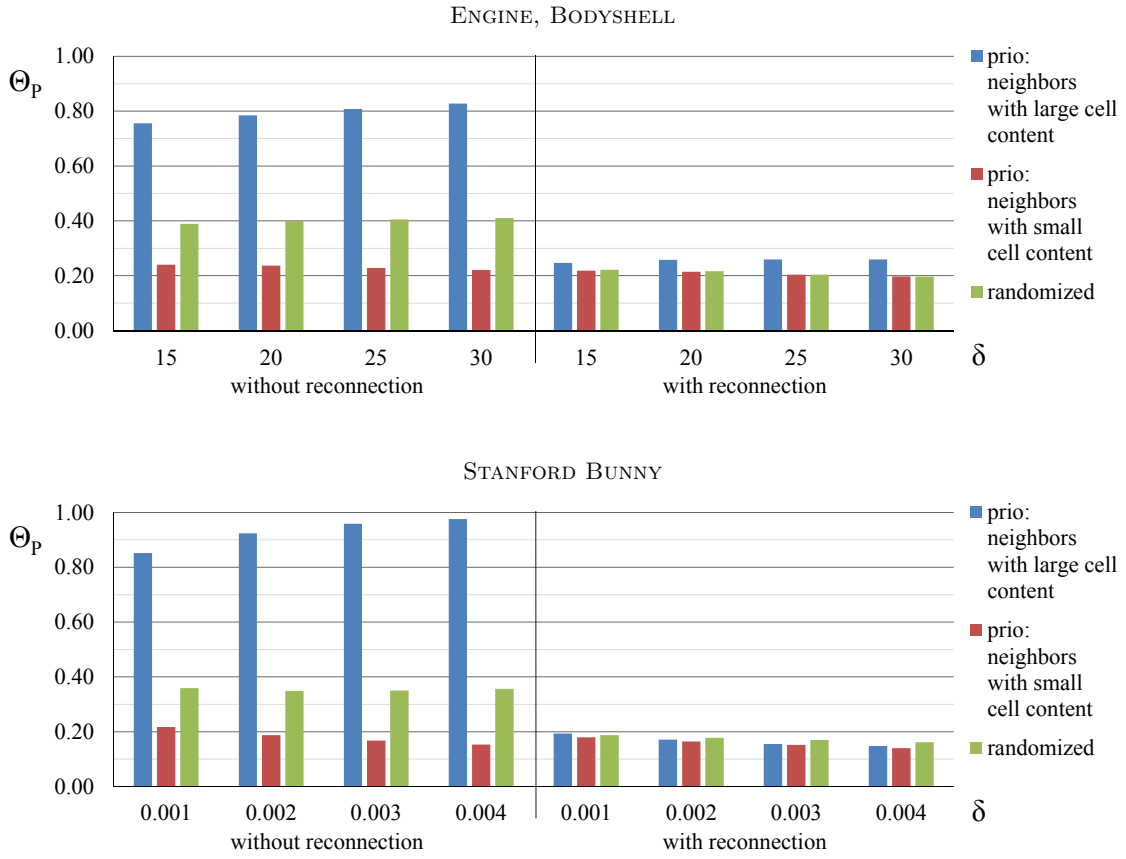


Fig. 5.31: Influence of the connection order and of the reconnection for the free constructed shrubs of the bodyshell of the *Engine* scenario (top) and the Stanford Bunny (bottom).

5.6.2. Free Construction

In this section we will mainly analyze different options and decisions we made for the free construction. Given the method to connect a cell to the shrubs, the most important issues for the quality of the constructed shrubs are the connection order and the reconnection of already connected cells. Figure 5.31 show the influence of the connection order and of the reconnection over a varying tolerance value δ. We show three different connection orders: Two variants are with priority based cell selection (blue and red) and one is with randomized cell selection order (green). The priority based cell selection gives neighbor cells with small cell content the higher priority (blue) or neighbor cells with large cell content the higher priority (red). We show these three variants with and without reconnection. Therefor the diagrams are split into two parts. For the results of the left side the reconnection is not applied, for the results of the right side the reconnection is applied.

Again, the compression results behave similar for the *Engine* bodyshell and the Stanford Bunny. The most significant observation is that the reconnection is essential for the priority based cell selection with large cell content priority and important for the

randomized cell selection order. Actually, the compression results of the priority based cell selection with large content priority are pretty bad without reconnection. By this choice of the priority rule, there are a lot of wrong decisions of our greedy cell connection. For the randomized cell selection the situation is similar, but not that extremely bad. Only when using the reconnection, the randomized cell selection and especially the priority based cell selection with large cell content priority achieve good results. Thus, on these examples one can see that the reconnection is a powerful method to revise false decisions during the cell connection.

The priority based cell selection that prefers cells with small cell content does not profit from the reconnection in that magnitude. This shows that this cell selection is a good choice as most decisions in the cell connection steps are obviously correct. So naturally, although the reconnection step can repair a lot of false decisions, it is useful to apply the best decision method in the cell connection step. This is clearly the priority based cell selection that gives neighbor cells with small cell content the highers priority.

However, it is notable that the randomized cell selection order achieves, by the aid of the reconnection step, almost similar good results as the priority based selection that gives neighbor cells with small cell content the priority. The algorithm to create the shrubs with randomized cell selection order is easier to implement than the one for the priority based selection. Further, the preprocessing step to calculate the number of primitive references per cell is not necessary. So the randomized cell connection order with reconnection is a useful alternative.

Summarized, the best connection order is the one that gives neighbor cells with small content the priority. Nevertheless, the reconnection step is able to repair the false decisions within the cell connection with randomized cell order such that similar results can be achieved for both methods. We achieve compression ratios between 0.22 and 0.14. Thus, we have a space saving between 78% and 86%.

The reasons can be balanced by the user whether an easier construction algorithm is desired (which is the case for the randomized connection order) are a more conservative method that achieves equal of slightly better results (which is the case for the priority based method that gives neighbor cells with small content priority). We prefer to work with the latter method and use it also the following benchmarks.

5.6.3. Overall Compression Results

In this section the best schematic construction variant (refer to Section 5.6.1) and the best free construction method (refer to Section 5.6.2) are compared and analyzed on our benchmarks. We will consider the compression ratio of primitive references Θ_P as well as the compression ratio of the memory Θ_M of a compressed TD-grid.

Space Saving of Primitive References. For the diagrams in Figure 5.32 the shrubs data structure is calculated for different benchmarks and for varying tolerance values δ. The diagrams display the achieved compression ratio of primitive references Θ_P. In

all cases we achieve a compression ratio of between about 0.3 and 0.1. Thus, we have a space saving between about 70% and 90%.

For the Stanford bunny the free constructed shrubs achieve better results than the free constructed shrubs. The schematic constructed variant stores between 10% and 60% more primitives than the free constructed variant. For the bodyshell of the *Engine* scenario it is reverse. There the free constructed variant stores between 10% and 40% more primitives.

For more detailed analysis of this divergent behavior we will next consider two sub-parts of the *Engine* bodyshell. One of the sub-parts contains only some tubes of the bodyshell, which is one of the most complex part of the bodyshell geometry. We call this sub-part the one with tubes. The other sub-part is is nearly the complete bodyshell, but without any complex geometry parts – we call it the one with less tubes. Refer to the Appendix A.1.4 for images of the two sub-parts.

For the sub-part with tubes the free constructed shrubs stores between 50% and 60% more primitives as the schematic constructed shrubs. For the one with less tubes the schematic schematic constructed variant stores between 40% and 60% more primitives. So we have for the two sub-parts the similar behavior as shown between the *Engine* bodyshell and the *Bunny*.

Schematic opposed Free Constructed Shrubs

We would like to formulate a general recommendation in which situation the free con-

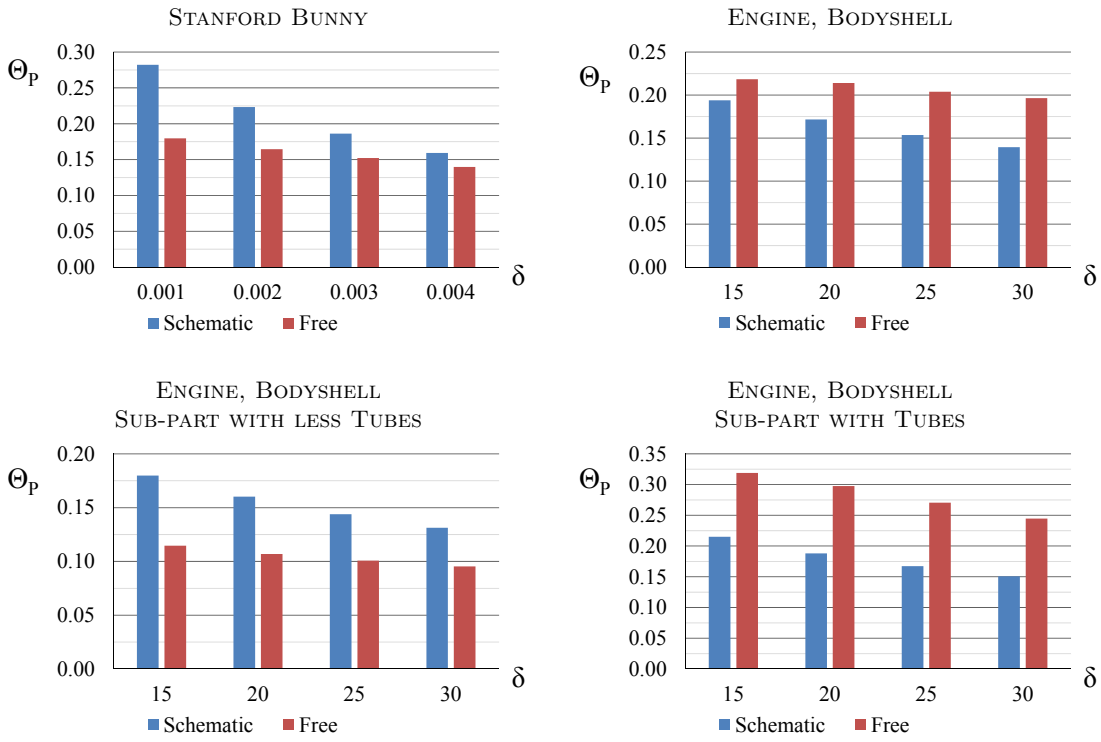


Fig. 5.32: Space saving of referenced primitive references Θ_P by the shrubs data structure of different models for varying tolerance value δ.

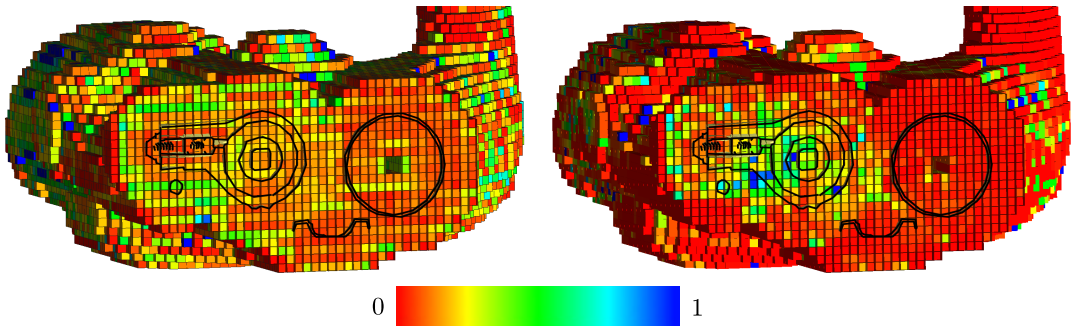


Fig. 5.33: Analysis of the *Engine* bodyshell sub-part with tubes on the sectional drawing trough some tubes. Color code visualization for the space compression ratio per cell for the schematic construction (left) and the free construction (right).

structed shrubs and in which situation the schematic constructed shrubs achieves better results. Comparing the two sub-parts of the *Engine* bodyshell leads to the assumption that the tube geometry is problematic for the free constructed shrubs. We will analyse the quality of the free and the schematic constructed shrubs on the example of the bodyshell's tube geometry more detailed in the following.

Refer to Figure 5.33 where a sectional drawing through the bodyshell sub-part with tubes and through the respective grid cells is shown. The figure displays a color code visualization of the compression ratio of primitive references per grid cell. The color code is calculated for a grid cell by the number of primitive references of the cell divided by the proportional contribution of the cell to the number of primitive references in the shrubs (which was given in Definition 5.3). The left picture in Figure 5.33 was generated by using schematic constructed shrubs, whereas in the right picture the free constructed shrubs are used. Both are calculated with tolerance value $\delta = 15$ and cell width $w = 5$. Considering again the lower left diagram in Figure 5.32 for this situation, we know that the schematic construction is more efficient with a compression ratio of 0.22, compared with 0.32 for the free construction. How this difference, by the factor 1.5, is established can be comprehended by the aid of the Figure 5.33.

We compare therefore the left geometry part and the right geometry part in each sectional drawing. It can be easily seen that the left part is much more dispersive and scattered than the one on the right side.

The cell contents of the right side in each sectional drawing can be easily described as follows: The cells that intersect the geometry have the most primitive references. Going from these cells outwards/inwards in normal direction, that cells have roughly a subset of the cell content of the previous one. This situation is perfectly to handle for the free construction method (refer to the respective color codes of the cells), since the best fitting neighbor is usually clear the newly added calls can either be connected to the shrubs without adding any new primitive references or by adding only few primitives references.

The cell content of the left-hand side cannot be described easily, as the geometry intersects many different cells irregularly in different regions of the grid. One cell

content is not that often a subset of a neighbor cell's content – especially when the cell has to reference primitives of different opposed located primitives. In such a situation it is difficult to decide which is the best fitting node in the already existing shrubs. This can be seen again by the color codes beautifully.

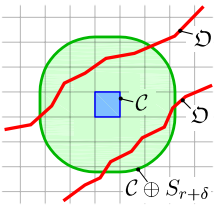


Fig. 5.34: Problem case in the free construction, where often non-optimal decisions are made to connect $\mathcal{C}$ to the shrubs.

Summarized we found out that not the tubes itself are problematic for the free construction method and therefore better to handle by the schematic construction method. Moreover the dispersive and scattered geometry is the issue. We will state this more precisely and general: Given a cell $\mathcal{C}$ and a complex object $\mathfrak{O}$. We know that $\mathcal{C}$ references all primitives of $\mathfrak{O}$ that intersect $\mathcal{C} \oplus \mathcal{S}_{r+\delta}$ (Section 3.3.1). Assumed $\mathcal{C}$ is located between two different surface regions of $\mathfrak{O}$ such that $\mathcal{C} \oplus \mathcal{S}_{r+\delta}$ is intersected on different sides by different sets of primitives of $\mathfrak{O}$ (refer to Figure 5.34). Grid cells in such a situation as cell $\mathcal{C}$ achieve usually not that good compression ratios by the free construction method. It is difficult to decide which is the best fitting neighboring cell of $\mathcal{C}$ as different neighboring cells contain mainly disjoint subsets of the cell content of $\mathcal{C}$. Thereby false decisions are made, which cannot always be corrected by the reconnection step.

Recommendation

We generally state: In the case the complex object consists of a lot of dispersive and scattered geometry where many surface parts are closer than $2(\delta + 2r)$, usually the schematic constructed shrubs achieve better compression results than the free constructed shrubs. In the case different surface parts of the complex object are farther apart, the free constructed shrubs achieve better compression results than the schematic constructed shrubs.

Summary

We can summarize that the free construction as well as the schematic construction of the shrubs data structure are able to achieve considerable compression results of about 0.2 to 0.1 compression ratio. This means that only about 10% to 20% of the original primitive references has to be stored and so we have a space saving of 80% to 90%.

Memory Consumption. Until now we have focused only on the compression rate of primitive references Θ_P. This was eligible as we showed in Section 5.1 that the largest memory portion of a TD-grid is spend on storing primitive references. But finally the memory compression that can be achieved by using the shrubs data structure is interesting. Therefore we calculate the memory consumption of an uncompressed TD-grid and compare it with the memory consumption of two compressed TD-grids. One is created by the schematic construction method and and the other by the free construction method. Further we compare these results with an uncompressed TI-grid. The most interesting question in this course is whether we can still achieve that good compression results. Naturally the compression ratio of the memory consumption Θ_M is not as good as the compression ratio of the primitive references Θ_P. The reason is that a constant memory portion is spent for the data structure of the grid keys and moreover the memory overhead that is used for the data structure that maintains the cell content. Surely the memory overhead that is used by the shrubs data structures is larger than the one be the uncompressed grid.

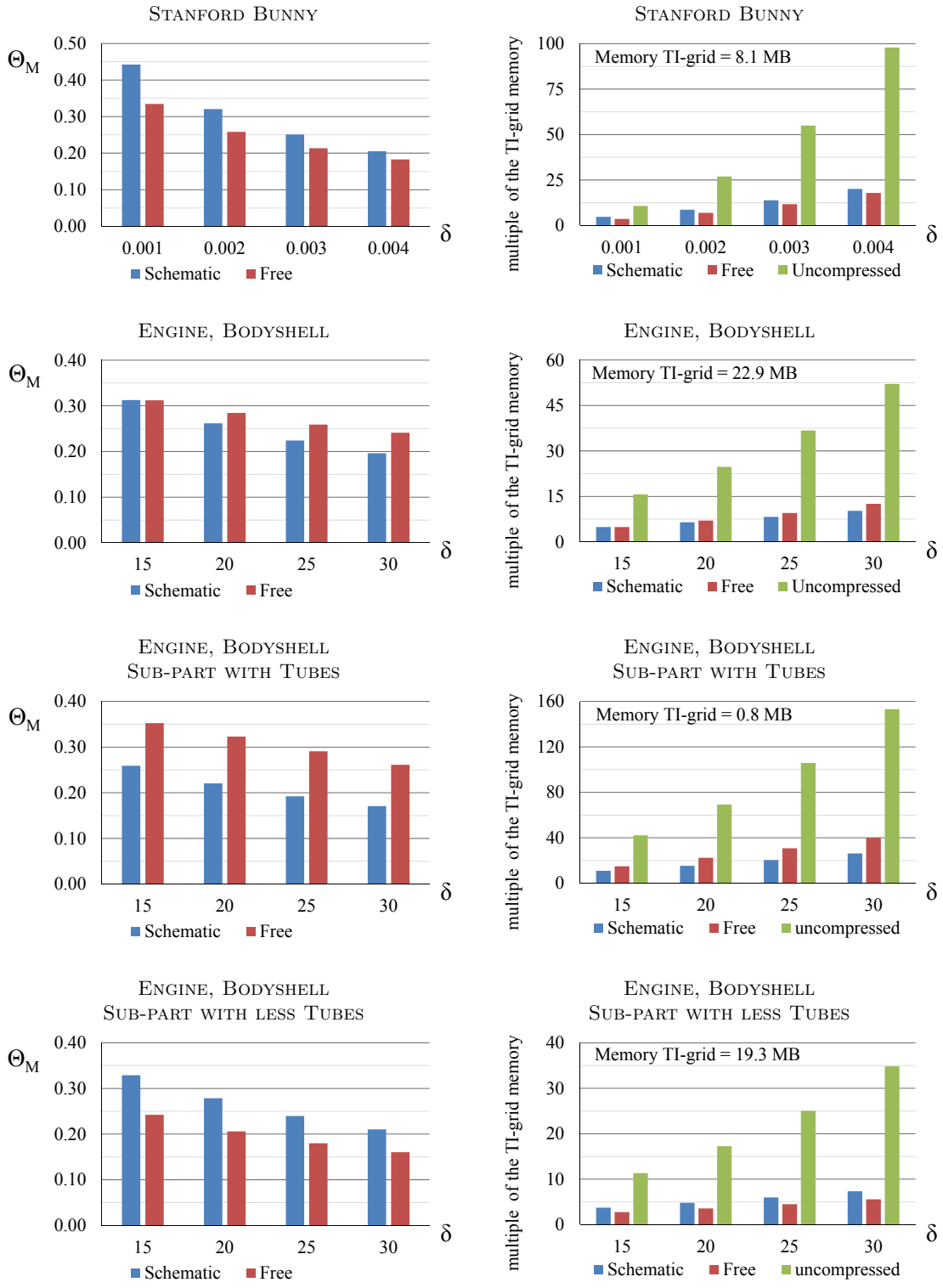


Fig. 5.35: The achieved memory compression ratio Θ_M by using the shrubs data structure for different models for varying tolerance value δ and fixed cell width (left column). The respective factor indicating how much more memory an uncompressed TD-grid, a compressed TD-grid that uses free constructed shrubs and a compressed TD-grid that uses schematic constructed shrubs use, compared to a TI-grid (right column).

The diagrams in the left column of Figure 5.35 show the memory compression ratio Θ_M that are achieved by the shrubs data structure. For all diagrams in Figure 5.35 the same scenarios are used as for the diagrams in Figure 5.32, which shows the compression ratio of the primitive references Θ_P. The space saving of the memory consumption Θ_M is calculated according to Definition 5.8. The memory of the uncompressed TD-grid and of the two compressed TD-grids are calculated as discussed in Section 5.5. For our benchmarks we used 3d-arrays to maintain the occupied cells. The 3d-arrays are referenced by the leafs of a 10-level hierarchical data structure.

We still achieve a very good compression ratio which is between 0.16 and 0.33. This is only slightly less than the achieved compression ratio of primitive references. Thus, the memory overhead of the shrubs data structure is not of big consequence.

The relation between the free and the schematic constructed shrubs is similar as it was shown in the benchmarks of Figure 5.32. Although, if we have a close look, the overhead of the free constructed shrubs is slightly smaller than the one of the schematic constructed shrubs. The reason is that the free constructed shrubs requires less nodes than the schematic (between 10% and 45% less).

The diagrams in the right column of Figure 5.35 compare the memory consumption of a TI-grid with the memory consumption of an uncompressed TD-grid, a compressed TD-grid that uses free constructed shrubs and a compressed TD-grid that uses schematic constructed shrubs. In each diagram the memory consumption of the TI-grid is given (refer to the upper left edge of each diagram) and the bars of the diagram display the factor how much more memory is required by the three TD-grids. Remember from the benchmarks of Chapter 4 that we achieve a much better performance for the calculation of all tolerance violating primitives when using a TD-grid for the static object instead of using a TI-grid. This performance benefit comes at the expense of a higher memory consumption. The diagrams in the right column of Figure 5.35 show how much more memory is required for an uncompressed TD-grid as well as for the two compressed TD-grids. The difference between the uncompressed and the compressed variants is obvious. For example for the Stanford bunny the TD-grid that uses the free constructed shrubs requires 18 times the memory of the TI-grid for the largest tolerance value, whereas the uncompressed TD-grid uses 98 times the memory of the TI-grid. In the case of the *Engine* bodyshell the TD-grid that uses the schematic construction requires 10 times the memory of the TI-grid for the largest tolerance value, whereas the uncompressed TD-grid uses 52 times the memory of the TI-grid.

Summary

Summarized, the compression ratio of the grid memory consumption achieves similar good results as the compression ratio of the primitive references that are analyzed above. Using a TD-grid for the calculation of all tolerance violating primitives speeds-up the performance on the expense of a higher memory consumption. The higher memory consumption of an uncompressed TD-grid is critical for large models. By the shrubs data structure the total memory consumption of the TD-grid can be compressed to about 1/5 of the uncompressed size. Thus, the memory compression of TD-grids that uses free constructed shrubs or of TD-grids that uses schematic constructed shrubs is remarkable.

6. Applications

In this chapter we outline some applications of our developed algorithms. We have implemented an environment for interactive real time visualizations (refer to Sections 6.1 and 6.2) and made experimental studies that use tolerance violating triangles for motion planning problems (refer to Sections 6.3 and 6.4).

6.1. Real Time Visualization of Tolerance Violations

Within the digital mockup (DMU) process components of a digital prototype are checked and eventually revised. One topic is the inspection of safety distances between several components of the digital prototype. Engineers want to know in which regions two components violate the safety distance given by the tolerance value δ. It is not only sufficient to inspect the compliance with the safety distance between components in their resting position but also in the dynamic process when the components are moving. In the case the engineers detect tolerance violating regions between the components, they have to revise the geometry of the components or have to revise the motion of the components such the tolerance violating regions are eliminated.

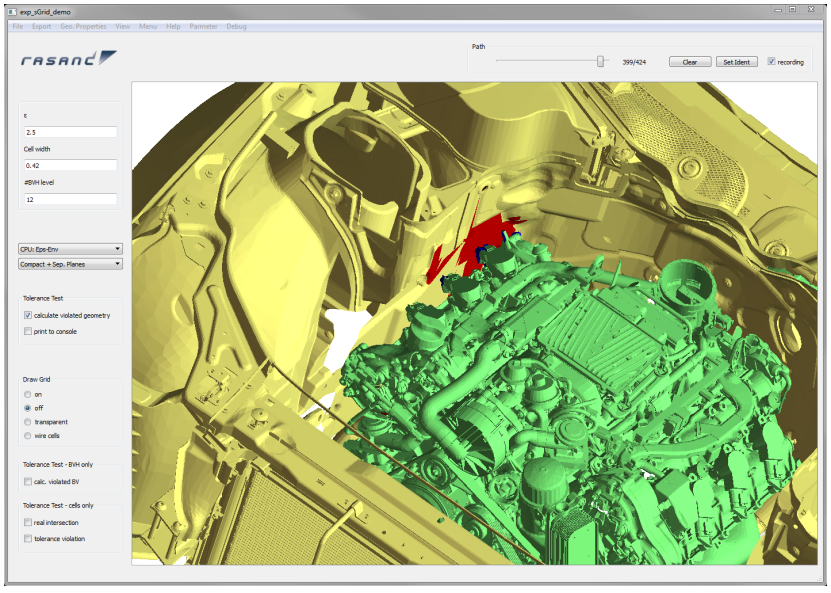


Fig. 6.1: GUI of the interactive application for the online visualization of all tolerance violating triangles. The green colored object can be moved interactively in space and the respective tolerance violating triangles are visualized in red and blue color in real time.

In the previous chapters we have presented our solution for the online calculation of all tolerance violating primitives between two complex objects. For the described inspection process in the DMU, we have further developed an interactive software tool. Engineers can load two triangulated complex objects into a graphical user interface, one movable geometry (the dynamic object) and one surrounding geometry (the static object). We then calculate and visualize all tolerance violating triangles between the movable geometry and the surrounding geometry for a given tolerance value δ. The engineers can translate and rotate the movable geometry interactively in space. For every position we display all tolerance violating triangles of the movable geometry and of the surrounding geometry in real time. The tolerance violating triangles as well as the respective transformation of the movable geometry can be output to a file. This is important for documentation and for further processing in other software tools where the geometry of the object can be re-engineered.

Figure 6.1 shows our GUI of the interactive software tool. The parameters of the data structures, the cell width of the grids as well as the number of hierarchy levels, can be set by the user or calculated by the empirical formula given in Section 4.4 or can be set after an automatic calibration process. Further, a once created data structure that was written to a file can be read again for a fast preprocessing.

6.2. Interactive Path Definition with Safety Distances

Another similar application case in the DMU process is the interactive path validation and definition under consideration of safety distances. Here the primary interest of the engineers is the information whether a component can be moved without violating safety distances to the surrounding geometry. For this question the engineers require an interactive tool that supports the definition of a path without tolerance violations. For example a component should be moved from an arbitrary position to its final fitting position. While the component is interactively moved by the engineer, every position is tested on tolerance violations. When a position causes a tolerance violation, the position is not accepted by the program and an internal retraction impulse is calculated in order to determine a nearby and physically plausible position instead. We have implemented this path definition process within our software tool.

The calculation of a retraction movement from a tolerance violating position to a valid position has not yet been discussed in the literature. But there has been a lot of research in the calculation of collision responses between rigid models, thus on the calculation of the retraction movement from a colliding position to a collision free position. For some examples refer to [16, 22, 3, 37]. Collision response is a basic functionality of most physic engines. We will denote the calculation of a retraction movement from a tolerance violating position to a not tolerance violating position as *tolerance violation response*. Calculating the tolerance violation response between two objects is equivalent to the problem of calculating the collision response between the first object and the δ-offset of the other object.

There are many possibilities to calculate a collision response, thus also to calculate a tolerance violation response. For our application it is not necessary to calculate a physically correct tolerance violation response. It is sufficient to have a plausible looking tolerance violation response which can be calculated easily and fast.

Let $\mathfrak{D}$ be the dynamic component and $\mathfrak{S}$ the static surrounding geometry, both given as triangle sets. When $\mathfrak{D}$ is set by the user to a position that causes tolerance violations with $\mathfrak{S}$, we apply a small translation and rotation movement to $\mathfrak{D}$. For this we use the information from the set of all tolerance violating triangles $T_\delta(\mathfrak{D},\mathfrak{S})$. We translate $\mathfrak{D}$ by $s\frac{\mathbf{t}}{\|\mathbf{t}\|}$, where $\mathbf{t}$ is the translation direction that is dependent on $T_\delta(\mathfrak{D},\mathfrak{S})$ and s is a constant factor that controls the translational step size. Further, we rotate $\mathfrak{D}$ around the axis $\mathbf{a}$ in the center of $\mathfrak{D}$ by an angle α, where $\mathbf{a}$ depends on $T_\delta(\mathfrak{D},\mathfrak{S})$ and α is a constant angle that controls the rotation step size. It is useful to set the factor s and the angle α relatively small and apply several retraction steps until $\mathfrak{D}$ and $\mathfrak{S}$ are not tolerance violating anymore. We set the factor s and the angle α dependent on the tolerance value δ and calculate the translation vector $\mathbf{t}$ and the rotation axis $\mathbf{a}$ by simple heuristics as follows:

We imagine that every tolerance violating point on $\mathfrak{S}$ gives an impulse on $\mathfrak{D}$ to repel $\mathfrak{D}$ in direction of the surface normal. Further we imagine that every tolerance violating point on $\mathfrak{D}$ gets pressure from outside and likes to evade in the opposite direction of its surface normal. In order to get one resulting translation direction we approximate this behavior by building the weighted sum over the normals of every tolerance violating triangle in $\mathfrak{S}$ and all reversely orientated normals of all tolerance violating triangles in $\mathfrak{D}$. The triangle normals are weighted by the respective triangle area. Thus, the translation direction is calculated as

$$\mathbf{t} = \sum_{\substack{\mathcal{T}\in T_\delta(\mathfrak{D},\mathfrak{S}) \\ \wedge\ \mathcal{T}\in\mathfrak{S}}} A_\mathcal{T}\ \mathbf{n}_\mathcal{T} \quad - \sum_{\substack{\mathcal{T}\in T_\delta(\mathfrak{D},\mathfrak{S}) \\ \wedge\ \mathcal{T}\in\mathfrak{D}}} A_\mathcal{T}\ \mathbf{n}_\mathcal{T}\ , \tag{6.1}$$

where $n_\mathcal{T}$ is the normal and $A_\mathcal{T}$ the area of a triangle $\mathcal{T}$.

Since we imagine that there is a pressure on all tolerance violating surface points of $\mathfrak{D}$, this pressure causes also a rotation movement of the object $\mathfrak{D}$ around its center of mass. We assume that for one tolerance violating surface point the rotation axis is perpendicular to the surface normal in this point and to the axis that connects the point with the center of mass of $\mathfrak{D}$. In order to get one resulting rotation axis we approximate this behavior by building the weighted sum of all such calculated axes for the triangle centers of all tolerance violating triangles in $\mathfrak{D}$. The triangle normals are weighted by the respective triangle area. The center of mass is approximated by the center $\mathbf{c}_\mathfrak{D}$ of the orientated bounding box of $\mathfrak{D}$. So the rotation axis is calculated as

$$\mathbf{a} = \sum_{\substack{\mathcal{T}\in T_\delta(\mathfrak{D},\mathfrak{S}) \\ \wedge\ \mathcal{T}\in\mathfrak{D}}} A_\mathcal{T}\ \mathbf{n}_\mathcal{T} \times (\mathbf{c}_\mathcal{T} - \mathbf{c}_\mathfrak{D})\ , \tag{6.2}$$

where $\mathbf{c}_\mathcal{T}$ is the center of a triangle $\mathcal{T}$.

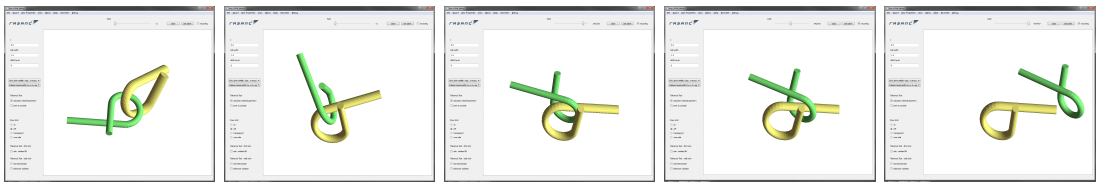


Fig. 6.2: Picture sequence to solve a nail puzzle with safety distances.

A precondition for this heuristic is that the triangle normals of $\mathfrak{S}$ and $\mathfrak{D}$ are well defined. This means that all normals point outwards from the object.

The calculation of $\mathbf{t}$ and $\mathbf{a}$ is integrated into the algorithm for calculating the set of all tolerance violating triangles and causes no notable additional calculation effort. We have integrated this functionality to our software application which was already presented in the previous Section 6.1. Although our method to calculate the tolerance violation response is not physically correct, our experiments showed that we are able to simulate a physical plausible behavior and our software can be a useful tool to support the interactive path definition by engineers.

In Figure 6.2 we show a picture sequence where a nail puzzle is manually solved by user interaction. We always ensure a safety distance between both nails by our tolerance violation response.

6.3. Release from Collision and Tolerance Violation

In the field of motion planning a collision free path of a movable component, called the robot, from a start position to a goal position has to be found through the so called obstacle geometry. There are standard algorithms that can find such a path automatically. For an introduction to motion planning refer for example to the book *Motion Planning Algorithms* by Steven M. LaValle [31].

In practice it is very often a problem that the robot and the obstacle are already colliding in the start position. Standard algorithms cannot be used in this case since they require a collision free starting position. For example, refer to Figure 6.3. In the

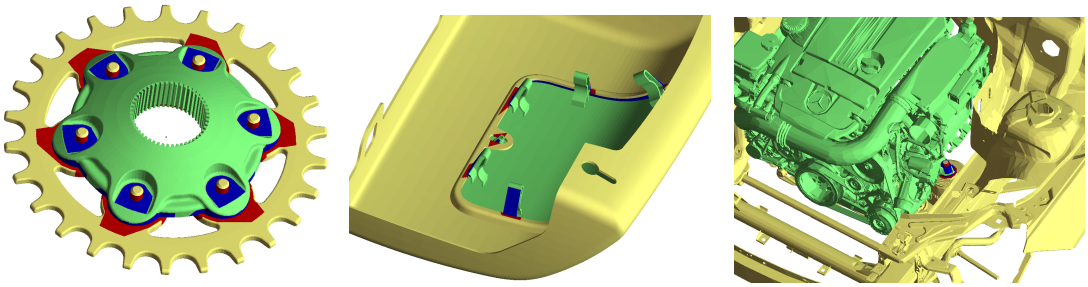


Fig. 6.3: Three motion planning examples with collision in the start positions. For a small tolerance value the tolerance violating triangles are highlighted red and blue.

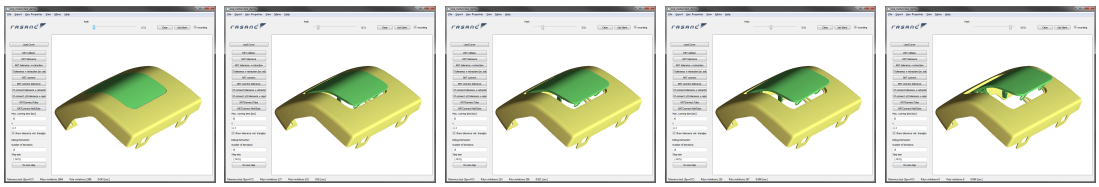


Fig. 6.4: Picture sequence to release the cover from its attached component by some steps of the tolerance violation response heuristic.

left picture a gear and a plate, where the gear is attached to, are shown. In the middle picture we show a cover that is fixed by several clips above a recess. In the right picture an engine and the front part of a car's bodyshell are shown. Currently engineers have to move colliding objects manually to nearby collision free positions. This is very work intensive if many components have to be checked in the DMU process. Our task is to release the colliding robot (here the plate, the cover, and the engine) from its contact component to a nearby collision free position and the releasing should look physically plausible. Then motion planning algorithms are able to plan a collision free path of the robot to the goal position. To release the colliding robot we use our approach of the tolerance violation response as presented in the previous Section 6.2. We define a tolerance value that dependents on the construction accuracy, e.g. two times the construction accuracy, and apply some steps of the tolerance violation response until the robot is collision free. In our experiments contact situations can be solved in a plausible way by our heuristic. Refer for example to Figure 6.4.

Our experiments showed that it is important to consider tolerance violating triangles instead of only colliding triangles. The reason is that the tolerance violating triangles provide more information about the near environment of colliding points. This is helpful to define a meaningful retraction motion. Further, since we work with triangulated data, there is always an approximation error of the triangulated surface with respect to the real smooth geometric shape. Usually the robot geometry is triangulated in a different way than the obstacle geometry. Even though the two original geometric shapes fit perfectly together and have a large contact area, the contact points in the triangulated geometry dependent on the respective triangulation. Provided a small tolerance value, the set of tolerance violating triangles corresponds usually more likely to the full contact area of the original geometric shapes than the set of colliding triangles.

6.4. Motion Planning with Safety Distances

Our approach to calculate all tolerance violating triangles finds application in the field of motion planning when safety distances have to be considered. Common motion planning algorithms find a collision free path of a movable component, the robot, through the obstacle geometry until a goal position is reached. For an introduction to motion planning refer for example to the book *Motion Planning Algorithms* by Steven M. LaValle [31]. But in practice, engineers desire to find motion paths such that in every position a given safety distance is hold. There are solutions that optimize the clearance

along a already calculated path, for example, by Geraerts and Overmars [9]. Another approach is to plan the path already by considering tolerances. This can be done by replacing the collision test in a standard motion planner by a tolerance test with a Boolean answer. For example, Erbes [3] and the PQP library [27] library provide such a tolerance test.

The consequence when planning with safety distances, thus with tolerances, is that the free space where the robot can be placed becomes smaller. Thus, a motion planning problem which might be easily solved when a collision free path is requested, can become arbitrarily difficult or even unsolvable – depending on the tolerance value δ. Thus, we face a motion planning problem with so called narrow passages. There has been recent work on such narrow passage problems, for example [3, 18, 56]. When planning with tolerances, we have to solve the narrow passage problems by considering tolerances, too. We follow the ideas presented by Erbes [3] and focus on sampling based single query motion planners.

Sampling based planners generate random configurations and try to connect each with an already existing valid neighboring configuration. A configuration is a translation and rotation that is applied to the robot. In our case a valid configuration is a configuration that is not tolerance violating. The step when a new sampled configuration C_{rand} is connected to a valid neighboring configuration C_{near} is called *local planning*. We interpolate with small discrete steps new configurations between C_{near} and C_{rand}. One interpolated configuration after the other is tested whether it is a valid one. Usually, when the first invalid configuration is reached, the last valid configuration is added to the set of valid configurations and a connection from this configuration to C_{near} is defined. Thereby a tree data structure is created. The strategy of Erbes to solve motion planning problems with narrow passages is to not stop after the first invalid configuration is reached. Instead, this invalid configuration is tried to be resolved by simulating a collision response. We generalized this idea for a motion planner with tolerances. As soon as an invalid configuration is reached within the local planning process, we try to resolve the invalid configuration by applying our tolerance violation response (refer to Section 6.2).

We already achieved good results in applying the tolerance violation response in motion planning problems with tolerances. We generalized an implementation of an RRTConnect (refer to Kuffner and LaValle [25] for details) by replacing all collision tests with tolerance tests and using the tolerance violation response in the local planning phase. Since one tolerance violation response is much more expensive than one tolerance test with Boolean answer, it is important to apply the tolerance violation response not in every local planning phase. In non-narrow regions the tolerance violation response should not be used, since it is much cheaper to explore the space by many additional random configurations. But in narrow regions using tolerance violation response is helpful to find a path, since additional random configurations in these regions are invalid with a high probability. We experimented with different strategies to distinguish between narrow and non-narrow regions. For example we distinguish depending on the number of interpolated configurations per local planning step, depending on how near a configuration in the active RRT is to the nearest configuration in the passive RRT, and

depending whether the two RRTs can be connected by a smaller tolerance value. These strategies already achieved promising results and are a good base for future research.

7. Conclusion and Future Work

The issue of this work was the calculation of all tolerance violating primitives (e.g. triangles) between two complex objects. The challenge thereby was to provide the results online for a sequence of transformations that is applied to one of the objects. This is required for example in the digital mockup process where engineers have to evaluate their constructed components in a dynamic process on compliance with a safety distance to the surrounding geometry. We have analyzed, developed, and benchmarked data structures and algorithms for the broad and the narrow phase as well as for the memory handling in order to solve this problem efficiently.

Narrow Phase

For the calculation of all tolerance violating primitives many primitive-primitive tolerance tests have to be executed. Because of that it is important to use an optimized primitive-primitive tolerance test. We focused in our work on triangle-triangle tolerance tests and worked out how such a tolerance test could be formulated. We showed different solutions in primal space and presented our new solution that works in dual space.

We derived the properties of a tolerance test in dual space between convex objects and between pairs of triangles. Based on this we presented our dual triangle-triangle tolerance test. Our main idea was to consolidate some feature tests. More precisely, we replaced 29 separating plane tests in primal space with 6 intersection tests in dual space of a ray/line and the dual of a triangle's δ-offset.

On a wide range of test sets we benchmarked all presented triangle-triangle tolerance tests as well as triangle tests from related work. We made two main conclusions. The first one is that the performance of a tolerance test is always significantly higher than the one of a distance test. So it is worth to study the until now rarely focused issue of tolerance tests. Secondly, we showed that our dual approach is faster than approaches from related work and faster than all our primal implementations. Thus, mapping the problem into dual space in order to consolidate feature tests proved to be efficient.

The most interesting issue for future work is to transfer our ideas for the triangle-triangle tolerance test to a tetrahedron-tetrahedron tolerance test. As soon as this primitive test is provided, one can directly calculate all tolerance violating tetrahedrons between two complex volumetric objects since all our broad phase data structures and algorithms are implemented generically by using templates.

Broad Phase

For the online calculation of all tolerance violating primitives between two complex objects it is important to have an efficient broad phase data structure and efficient query algorithms. We analyzed and benchmarked common data structures in order to weight advantages and disadvantages concerning our problem statement. Based on

this awareness we developed a combined data structure that consists of a flat bounding volume hierarchy of axis aligned bounding boxes and several uniform grids. Each leaf of the bounding volume hierarchy is associated with one uniform grid. The bounding volume hierarchy allows to focus fast on relevant geometry parts and further allows to maintain the uniform grids within 3d-arrays, which is most efficient for cell-look ups in the grid query. The grid data structure allows to subdivide the calculation into several independent tasks, which can be processed in parallel. One thread is thereby responsible for one query cell of the dynamic object. Usually it is an issue how to set suitable parameters of a data structure. Hence, we have developed a strategy how to set the parameters of our data structure in order to achieve fast running times.

We presented two different approaches for the grid query algorithm, denoted as the symmetric and the asymmetric approach. Our symmetric approach performs, in most cases, better than a pure bounding volume hierarchy traversal. The processing time for constructing the data structures is very fast (it takes under one second). So our asymmetric approach is ideal when engineers want to inspect the components in some positions for varying tolerance values. For our asymmetric approach we use a tolerance value dependent grid (TD-grid). The TD-grid considers the tolerance value already at construction time, thus, within the preprocessing. The time to construct a TD-grid is longer than the one for a tolerance value independent grid. It takes currently, dependent on the tolerance value, between one half and three minutes.[1] However, we have not yet deeply focused the performance of the construction process of a TD-grid and we are sure that there is potential for a more efficient construction time. So it is an interesting issue for future work. Since the asymmetric approach already considers the tolerance value in the preprocessing, it performs significantly faster than the symmetric approach. In our benchmarks it is between remarkable 2.5 and 10 times faster.

Using the strategy of core-primitives, we are able to mark many primitives as tolerance violating without performing an expensive primitive-primitive tolerance test. In our benchmarks we showed for example that we are able to mark 3/4 of all tolerance violating triangles by this technique in the asymmetric approach. Thereby we save about 10% of all triangle-triangle tolerance tests, leading to a speed up of a factor 1.2. Thus, using the strategy of core-primitives improves the performance.

We also presented a solution for the online calculation of all tolerance violating triangles that uses the GPU. It is a hybrid concurrency approach that utilizes the CPU as well as the GPU. We showed that this GPU implementation is extremely hardware dependent. In our benchmarks we reached a speed-up up to 2. Since we have developed our GPU approach on the Fermi generation of NVIDIA graphic cards, it might be interesting to see how our problem statement could be implemented by further algorithms that are optimized for newer generation NVIDIA graphic cards or for OpenCL. Further, the realization of GPU versions for the asymmetric approach could also be an issue for future work.

The most important success of our broad phase implementation is that we are able

[1] The construction time is measured for the tolerance values $\delta \in [5, 30]$ for the bodyshell of the *Engine* scenario.

to achieve with our asymmetric approach a similar and in most cases an even better performance than an offline approach from related work. In average we are 1.5 times and in altogether up to 2.2 times faster in all our benchmarks. This is notable, since we have with our online calculation more computation overhead in order to organize a parallel calculation than an offline calculation has.

Memory Issues

Since we observed that our TD-grid requires a lot of memory to store primitives, we developed a novel data structure to compress the memory consumption, called the shrubs. The shrubs maintain the cell content of all grid cells. The main idea of our data compression is that cells with similar content can share their common content. Thus, the shrubs are a forest of trees where each node stores some primitive references and each grid cell is associated with one node in the shrubs. The stored references of this node and of all its ancestor nodes together sum up to the cell content. Thus, at query time the content of a grid cell can be uncompressed easily and lossless by traversing a tree in the shrubs from the associated node of the grid cell to the root node.

We presented two types of shrubs, the schematic constructed shrubs and the free constructed shrubs. We found out that the schematic constructed shrubs perform better for complex objects with a lot of scattered and disperse geometry. Otherwise the free constructed shrubs perform better. With both variants we showed in our benchmarks that we are able to significantly compress the memory consumption of the uniform grids using only about 20% of the original size.

To the best of our knowledge, the idea to compress the cell content in a uniform grid has not been considered before in the literature. So this part of our work involves a new aspect to store uniform grids with highly redundant cell content more memory efficient.

Final Conclusion

In summary, we have presented two holistic approaches for the online calculation of all tolerance violating primitives between two complex objects. All aspects are well deliberated, studied, and benchmarked. We are able to compute in all our benchmark cases the results in real-time. More precisely, if an application desires to visualize 24 frames per second we require never more than 32% of the allowed time per frame in all our benchmarks. In average, over all benchmarks, we require only 4.5% of the allowed time per frame. Thus, there is definitely enough time for further processing per frame (like visualizations or computing reactions).

Our algorithms can be used for interactive applications where engineers inspect components on their violation of safety distances. By our asymmetric approach we are able to calculate and visualize in real-time all tolerance violating triangles between two complex objects with many hundreds of thousands of triangles (we benchmarked with models up to 1 million triangles). Even our slower symmetric approach can be perfectly used for detailed analysis where for a varying tolerance value the tolerance violations have to be displayed. We further proved the application of our algorithms in the simulation of retraction motions in interactive applications and also within motion planning tasks.

A. Benchmark Data

In the following sections we present our used benchmark data. We define four scenarios, each with two geometric models. For each scenario we present three transformation sets. All transformations are then successively applied to one of the two geometric models, denoted as the dynamic model.

A.1. Engine Scenario

The models of the *Engine Scenario* were provided to us by our industrial partner Daimler AG. The static object is the front part of a car's bodyshell and the dynamic object is a complete engine. The models consist originally out of free form surface patches. They are modeled not as volumetric solids, thus, sheets of metal are modeled as a surface. Because of that the models contain naturally holes. The models are output from the CAD-system CATIA as triangle sets with a used defined accuracy and are simplified for further processing. The final triangle set of the bodyshell consists of 92,671 and the one of the engine of 126,820 triangles. Due to the discretization and the simplification contain the triangle sets differently sized and not necessarily well shaped

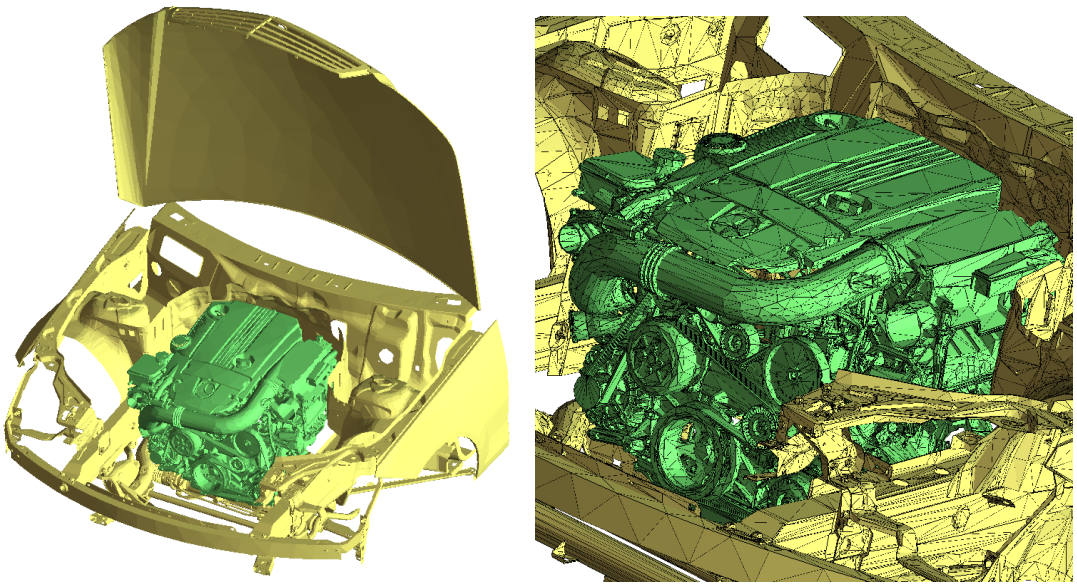


Fig. A.1: Models of the *Engine* scenario.

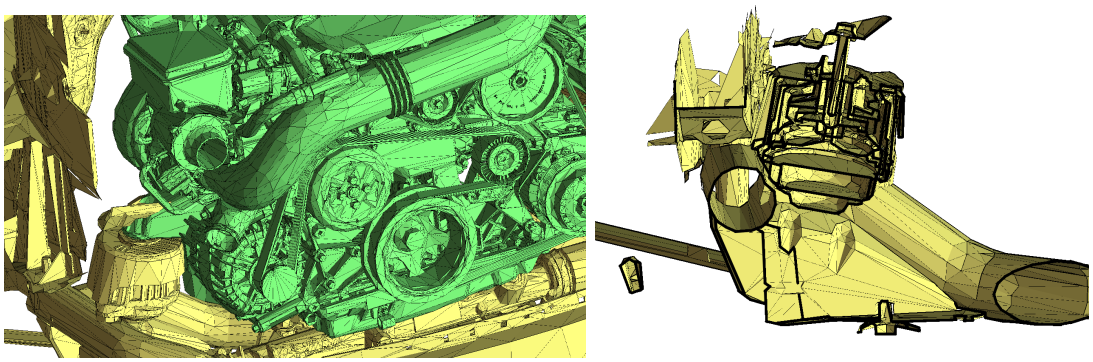


Fig. A.2: Close-up of the models of the *Engine* scenario and sectional drawing through the engine mount of the bodyshell to see the detailed modeling.

triangles and have partially self intersections. The models are displayed in Figures A.1 and A.2. Some more statistical quantities are given in Section A.6.

In order to benchmark our algorithms, we generated different sets of transformations of the engine as described in the following. Further, we created some sub-part models of the bodyshell model which are used for benchmarks on memory consumption. They are presented in Section A.1.4.

A.1.1. Transformation Set ColTolVerl

ColTolVerl is a set of 10,000 random transformations of the engine such that there are in average about 1,000 tolerance violating triangles per position for $\delta = 15$. More precisely, there are in average 1,162 tolerance violating triangles per position. Over all positions we have between 462 up to 1888 tolerance violating triangles for $\delta = 15$. Figure A.4 (top) shows some sample positions.

Figure A.3 shows the average number of tolerance violating triangles per position for *ColTolVerl* for a varying tolerance value δ. We have around 300 intersecting triangles per position. For $\delta = 15$ the about 1,000 tolerance violating triangles can be seen. Further, the number of tolerance violating triangles naturally increases with increasing tolerance values.

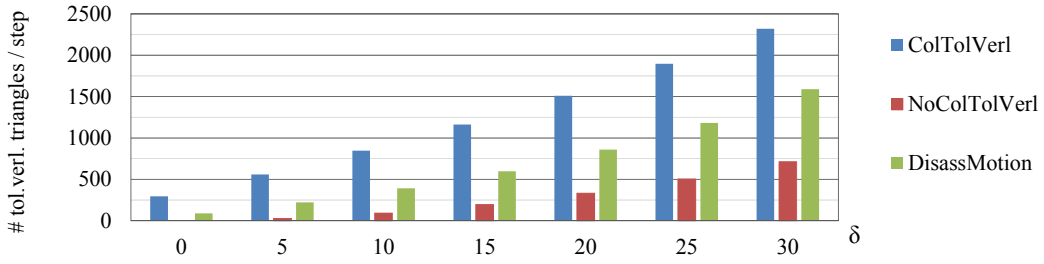


Fig. A.3: Average number of tolerance violating triangles per transformation in the three transformation sets of the *Engine* scenario.

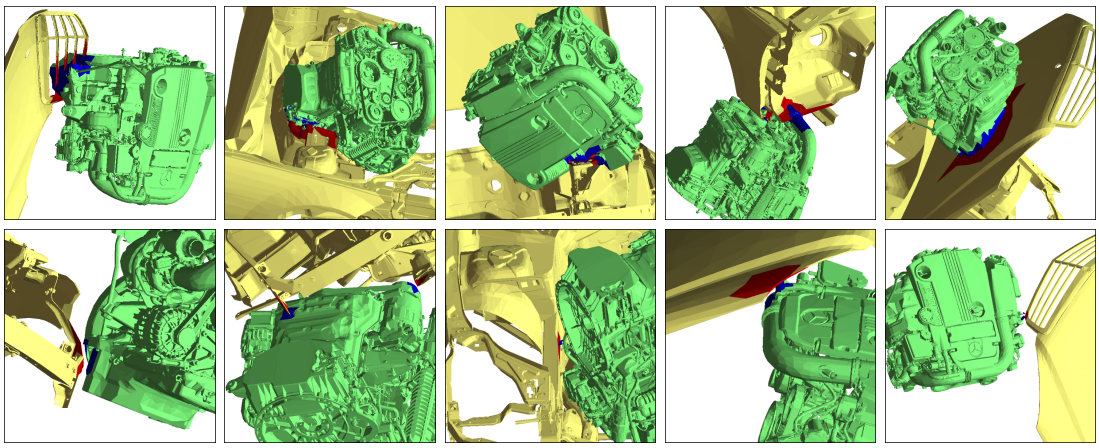


Fig. A.4: Sample positions of the engine in the transformation set *ColTolVerl* (top row) and *NoColTolVerl* (bottom row) with tolerance violating triangles displayed in red and blue color.

A.1.2. Transformation Set NoColTolVerl

NoColTolVerl is a set of 10,000 random transformations of the engine such that there are no colliding triangles between the models but definitely at last one tolerance violation for $\delta \geq 5$ in every position. Thus, the distance between the models in every position is larger than zero but smaller than or equal to five. Figure A.4 (bottom) shows some sample positions.

Figure A.3 shows the average number of tolerance violating triangles per position for *NoColTolVerl* for a varying tolerance value δ. Per definition, for $\delta = 0$ there are no tolerance violating triangles. Of course the number of tolerance violating triangles increases with increasing tolerance values.

In summary there are significantly less tolerance violating primitives per position in *NoColTolVerl* compared with *ColTolVerl*.

A.1.3. Transformation Set DisassMotion

DisassMotion is a motion path with 200 transformations that disassembles the engine from the bodyshell. This path is an example of a motion that demands corrections by the engineers as there are collisions and tolerance violations between the models. Figure A.5 shows some sample positions.

Figure A.3 shows the average number of tolerance violating triangles per position in *DisassMotion* for a varying tolerance value δ. For an impression on the number of tolerance violating triangles during the whole motion path refer to the diagram in Figure A.6. In the beginning of the motion we have the most tolerance violating and colliding triangles because the engine intersects the engine mount of the bodyshell in the initial position. Further, there are three periods without any tolerance violation.

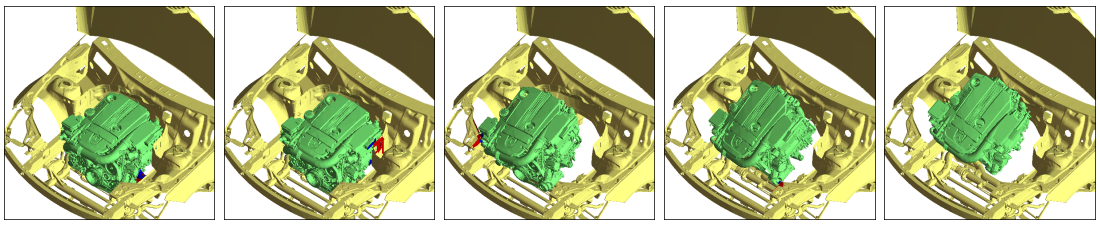


Fig. A.5: Sample positions of the engine in the transformation set *DisassMotion* with tolerance violating triangles displayed in red and blue color.

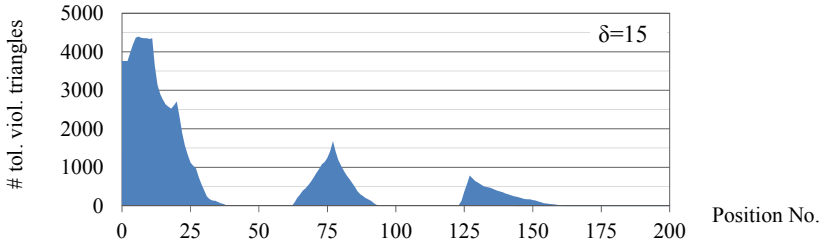


Fig. A.6: Number of tolerance violating triangles during several positions in the transformation set *DisassMotion* of the engine in the *Engine* scenario with $\delta = 15$.

A.1.4. Sub-Parts of the Engine Models

For a more detailed analysis we investigated two sub-parts of the bodyshell. One sub-part consists mainly of the tubes at the bottom of the bodyshell. They are visualized in the left-hand picture of Figure A.7. The other sub-part is the bodyshell without the tubes and without the engine mount, i.e. without the most complex geometry parts. This is visualized in the right-hand picture of Figure A.7. Statistical quantities of both models are given in Section A.6.

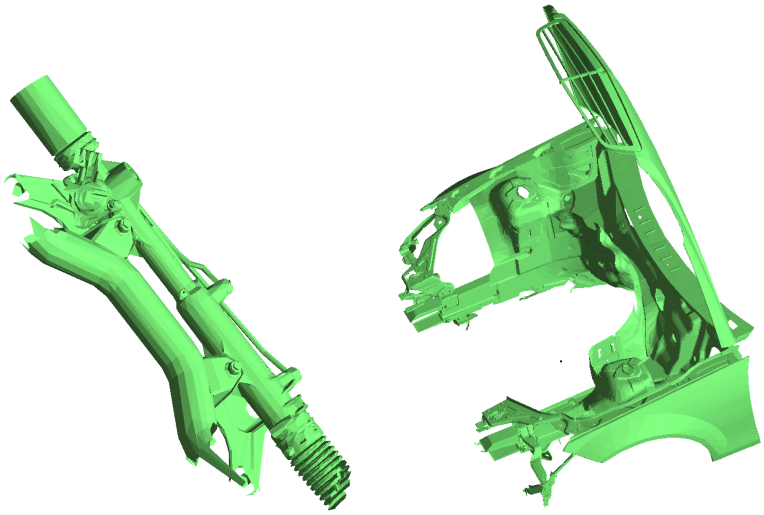


Fig. A.7: Sub-part of the bodyshell that contains the tubes in the middle of the bodyshell (left) and a sub-part of the bodyshell that contains less tubes.

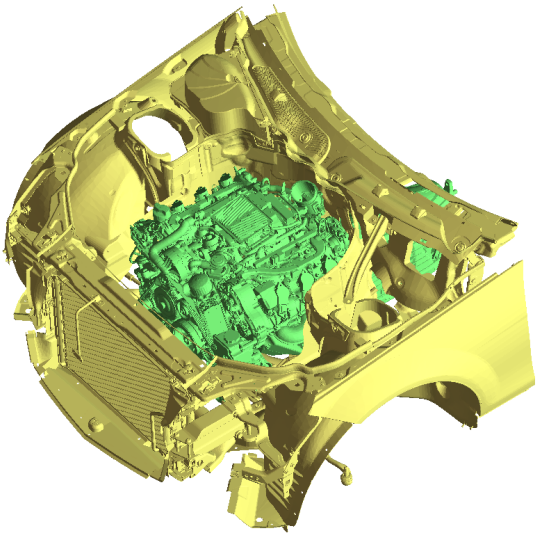


Fig. A.8: Models of the *EngineBig* scenario.

A.2. EngineBig Scenario

The models of the *EngineBig Scenario* were provided to us by our industrial partner Daimler AG. The static object is the front part of a car's bodyshell and the dynamic object is a complete engine. The models consist originally out of free form surface patches. They are modeled not as volumetric solids, thus, sheets of metal are modeled as a surface. Because of that the models contain naturally holes. The models are output from the CAD-system CATIA as triangle sets with a used defined accuracy and are simplified for further processing. The final triangle set of the bodyshell consists of 325,119 and the one of the engine of 519,266 triangles. Due to the discretization and the simplification contain the triangle sets differently sized and not necessarily well shaped triangles and have partially self intersections. The models are displayed in Figures A.8 and A.9. Some more statistical quantities are given in Section A.6. Compared with the *Engine* scenario, these models are much more detailed and complex.

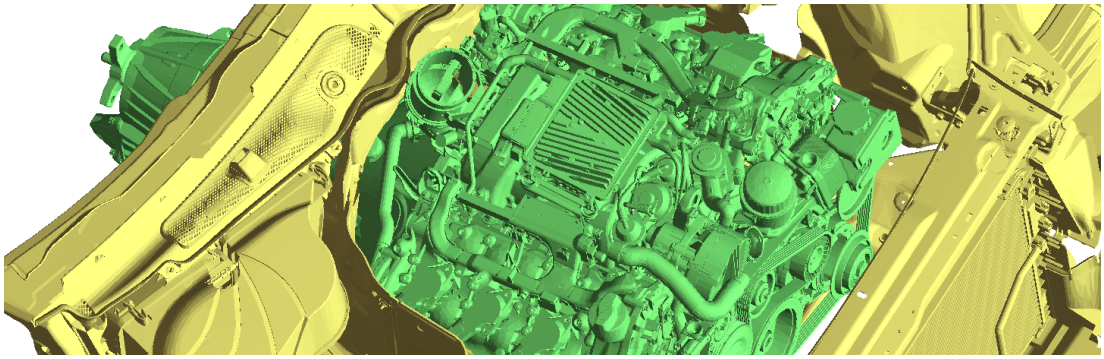


Fig. A.9: Close up of the models of the *EngineBig* scenario.

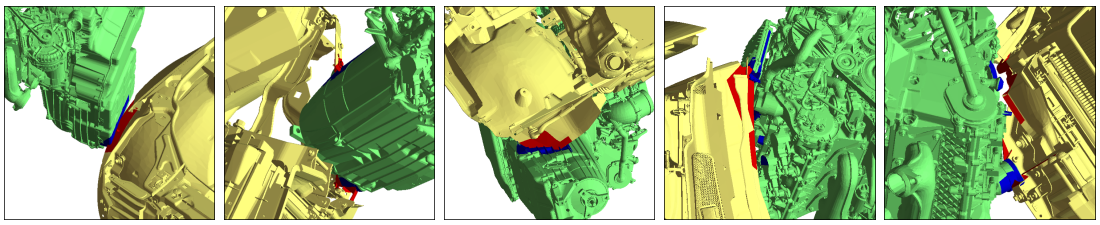


Fig. A.10: Sample positions of the engine in the transformation set *ColTolVerl* with tolerance violating triangles displayed in red and blue color.

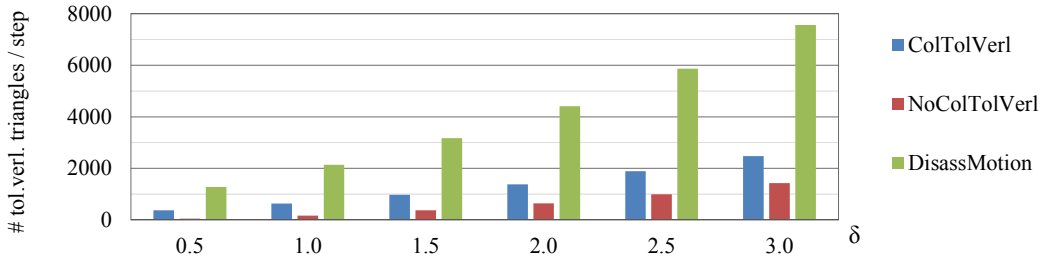


Fig. A.11: Average number of tolerance violating triangles per transformation in the transformation sets *ColTolVerl*, *NoColTolVerl*, and *ExtremeMotion* for a varying tolerance value δ.

In order to benchmark our algorithms to calculate all tolerance violating primitives, we generated different sets of transformations that were applied to the engine and are described in the following sub-sections.

A.2.1. Transformation Set ColTolVerl

ColTolVerl is a set of 1,000 random transformations of the engine such that there are in average about 1,000 tolerance violating triangles per position for $\delta = 1.5$. More precisely, there are in average 970 tolerance violating triangles per position. Over all positions we have between 500 up to 1,500 tolerance violating triangles for $\delta = 1.5$. Figure A.10 shows some sample positions.

Figure A.11 shows the average number of tolerance violating triangles per position in *ColTolVerl* for a varying tolerance value δ. We have around 164 intersecting triangles per position. For $\delta = 1.5$ the about 1,000 tolerance violating triangles can be seen. Further, that the number of tolerance violating triangles naturally increases with increasing tolerance values.

A.2.2. Transformation Set NoColTolVerl

NoColTolVerl is a set of 1,000 random transformations of the engine such that there are no colliding triangles between the models but definitely at least one tolerance violation for $\delta \geq 0.5$ in every position. Thus, the distance between the models in every position is larger than zero but smaller than or equal to 0.5. The top row in Figure A.12 shows some sample positions.

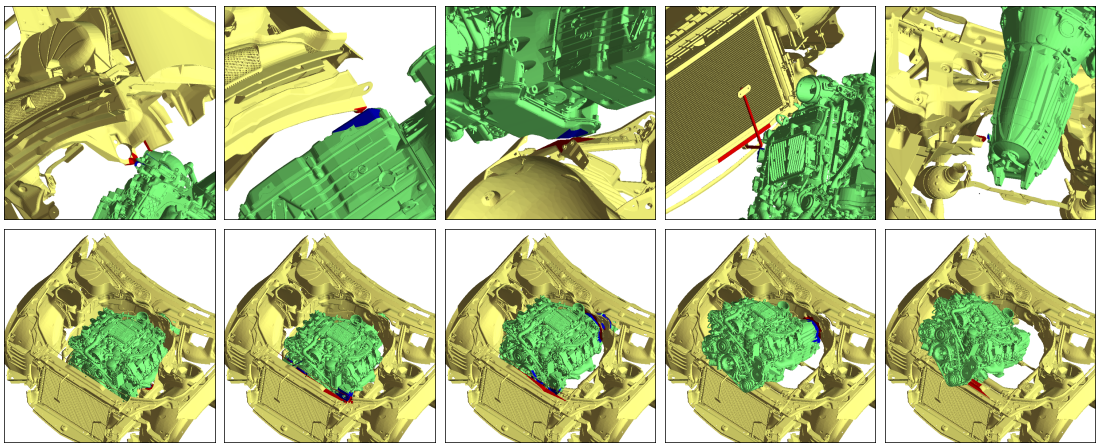


Fig. A.12: Sample positions of the engine in the transformation set *NoColTolVerl* (top row) and *DisassMotion* (bottom row) with tolerance violating triangles displayed in red and blue color.

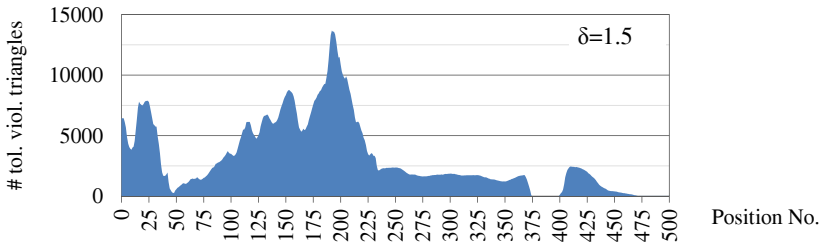


Fig. A.13: Number of tolerance violating triangles during several positions in the transformation set *DisassMotion* of the engine in the *EngineBig* scenario with $\delta = 1.5$.

The top row in Figure A.12 shows the average number of tolerance violating triangles per position in *NoColTolVerl* for a varying tolerance value δ. Per definition, for $\delta = 0$ there are no tolerance violating triangles. Of course, the number of tolerance violating triangles increases with increasing tolerance values.

A.2.3. Transformation Set DisassMotion

DisassMotion is a motion path with 500 transformations that disassembles the engine from the bodyshell. As in the *Engine* scenario, this path is an example of a motion that demands corrections by the engineers as there are collisions and tolerance violations between the models. The bottom row in Figure A.12 shows some sample positions.

Figure A.11 shows the average number of tolerance violating triangles per position in *DisassMotion* for a varying tolerance value δ. We can see that *DisassMotion* has the most tolerance violating triangles per step, compared with *NoColTolVerl* and *ColTolVerl*. For an impression of the number of tolerance violating triangles during the whole motion path refer to the bottom row diagram in Figure A.12.

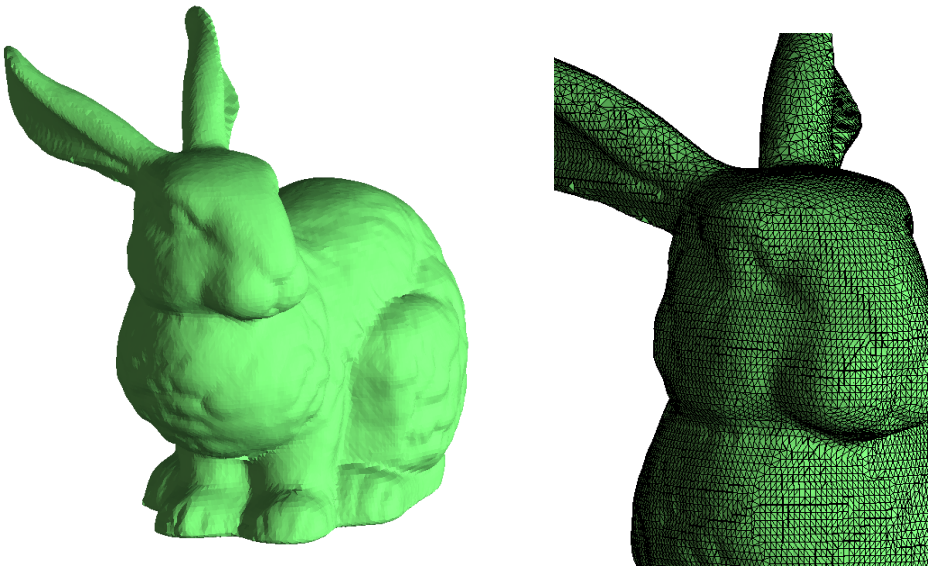


Fig. A.14: Stanford Bunny complete (left) and close up (right) with triangle edges displayed.

A.3. Bunny Scenario

The *Stanford Bunny* is a popular academic benchmark. It was obtained from 3d laser range data and consists of 69,451 triangles. The triangles are similar sized and lots of the triangles are nearly rectangular. The model contains only five holes in the bottom and no self intersections. Figure A.14 shows the bunny complete as well as a close up. Some statistical quantities are given in Section A.6.

In order to benchmark our algorithms to calculate all tolerance violating primitives, we use two bunny models. We mirrored one of them at the $yz-$plane and generated different sets of transformations that were applied to the mirrored bunny. The transformations sets are described in the following.

A.3.1. Transformation Set ColTolVerl

ColTolVerl is a set of 5,000 random transformations of the mirrored bunny such that there are in average about 1,000 tolerance violating triangles per position for $\delta = 0.002$.

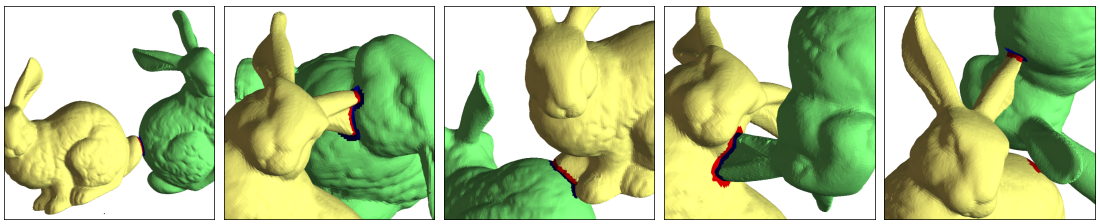


Fig. A.15: Sample positions for the mirrored bunny in the transformation set *ColTolVerl* with tolerance violating triangles displayed in red and blue color.

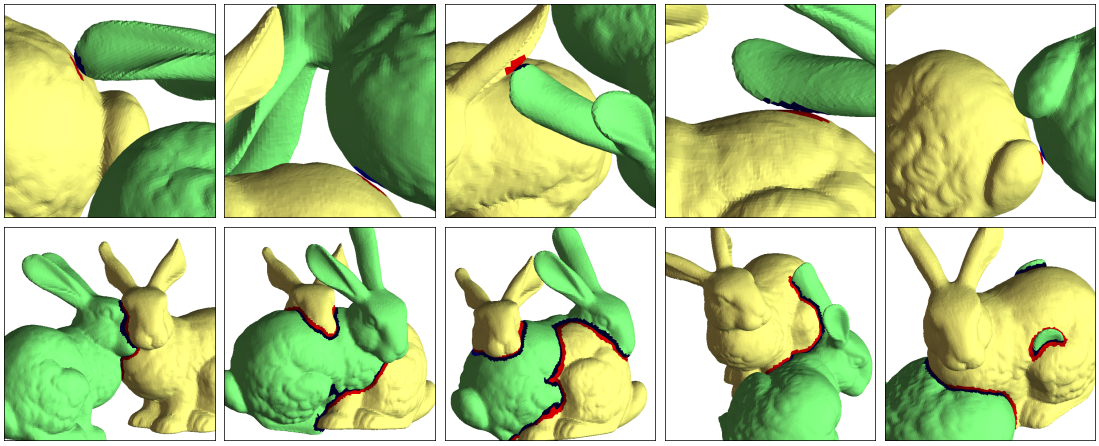


Fig. A.16: Sample positions for the mirrored bunny in the transformation set *NoColTolVerl* (top row) and *ExtremeMotion* (bottom row) with tolerance violating triangles displayed in red and blue color.

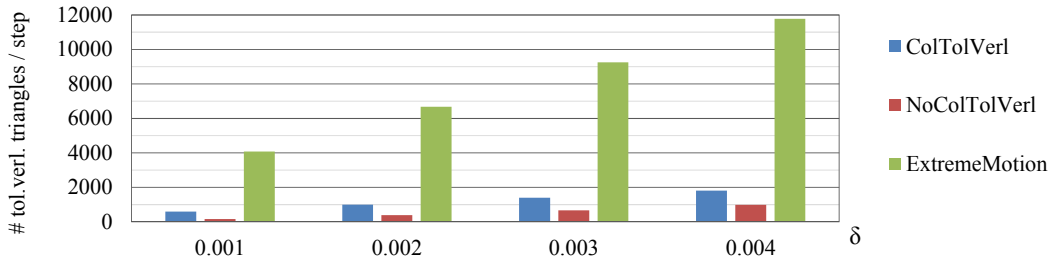


Fig. A.17: Average number of tolerance violating triangles per transformation in the transformation sets *ColTolVerl*, *NoColTolVerl*, and *ExtremeMotion* for a varying tolerance value δ.

More precisely, over all positions we have between 760 up to 1,250 tolerance violating triangles for $\delta = 0.002$. Figure A.15 shows some sample positions. Figure A.17 shows the average number of tolerance violating triangles per position in *ColTolVerl* for a varying tolerance value δ.

A.3.2. Transformation Set NoColTolVerl

NoColTolVerl is a set of 1,000 random transformations of the mirrored bunny such that there are no colliding triangles between the models but definitely at least one tolerance violation for $\delta \geq 0.005$ in every position. Thus, the distance between the models in every position is larger than zero but smaller than or equal to 0.005. The top row in Figure A.16 shows some sample positions and Figure A.17 shows the average number of tolerance violating triangles per position in *NoColTolVerl* for a varying tolerance value δ.

A.3.3. Transformation Set ExtremeMotion

ExtremeMotion is a motion path with 300 positions where both bunnies are penetrating in every step and there are many tolerance violating triangles in different spatial regions. The bottom row in Figure A.16 shows some sample positions and Figure A.17 shows the average number of tolerance violating triangles per position in *ExtremeMotion* for a varying tolerance value δ. As the bunny models penetrate each other, the number of tolerance violating triangles is extremely high for the positions in *ExtremeMotion*, compared with those in *ColTolVerl* and *NoColTolVerl*.

A.4. Buddha Scenario

The *Stanford Happy Buddha* is a popular academic benchmark. It was obtained from 3d laser range data and consist of 1,087,716 triangles. The triangles are very small and of different shapes. There are acute as well as obtuse angles. The model contains no holes, is a proper mesh, and has a large topological genus. Figure A.18 shows the buddha complete as well as a close up. Some statistical quantities are given in Section A.6.

In order to benchmark our algorithms to calculate all tolerance violating primitives, we used two instances of the buddha model and generated different sets of transformations that are applied to one of the buddha models. The transformation sets are described in the following.

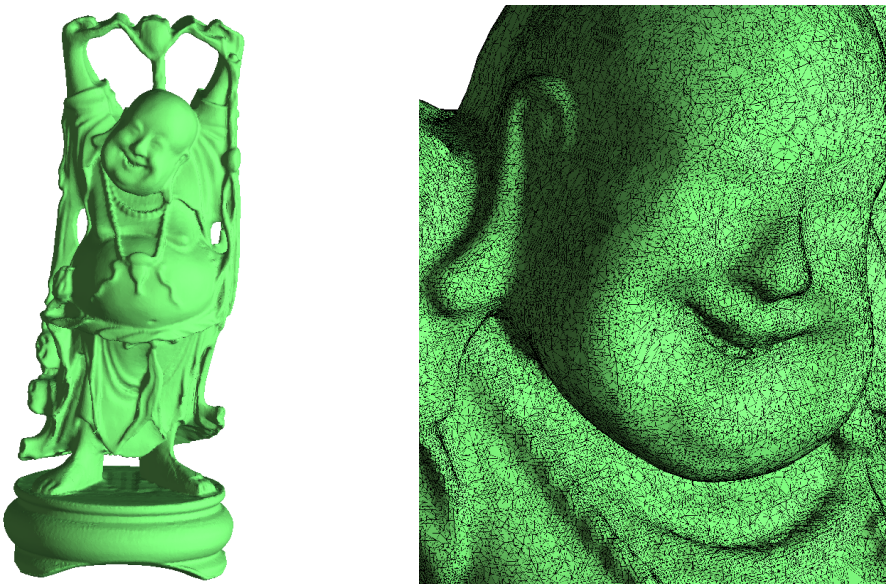


Fig. A.18: Stanford Happy Buddha complete (left) and close up (right) with triangle edges displayed.

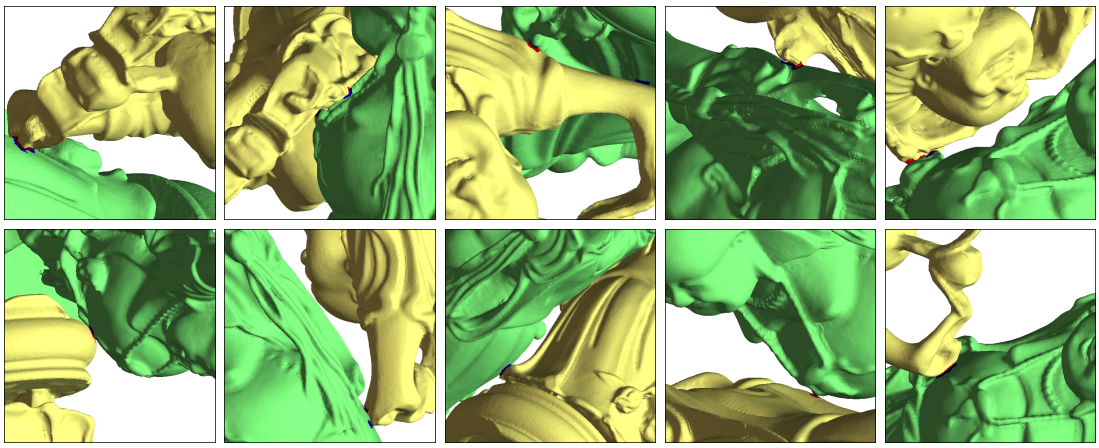


Fig. A.19: Sample positions for one buddha in the transformation set *ColTolVerl* (top row) and *NoColTolVerl* (bottom row) with tolerance violating triangles displayed in red and blue color.

A.4.1. Transformation Set ColTolVerl

ColTolVerl is a set of 1,000 random transformations of one buddha such that there are in average about 1,000 tolerance violating triangles per position for $\delta = 0.0005$. More precisely, over all positions we have between 500 up to 1,500 tolerance violating triangles for $\delta = 0.0005$. The top row in Figure A.19 shows some sample positions. Comparing these pictures with the pictures in the *Bunny* scenario for *ColTolVerl* (Figure A.16 top) gives the impression that there are less collisions for the buddha. This is true when comparing the total area of the tolerance violating triangles. But the number of tolerance violating triangles per position is similar as it is fixed to about 1,000 in both cases. Figure A.20 shows the average number of tolerance violating triangles per position in *ColTolVerl* for a varying tolerance value δ.

A.4.2. Transformation Set NoColTolVerl

NoColTolVerl is a set of 1,000 random transformations of one buddha such that there are no colliding triangles between the models but definitely at least one tolerance violation for $\delta \geq 0.0001$ in every transformation. Thus, the distance between the models in

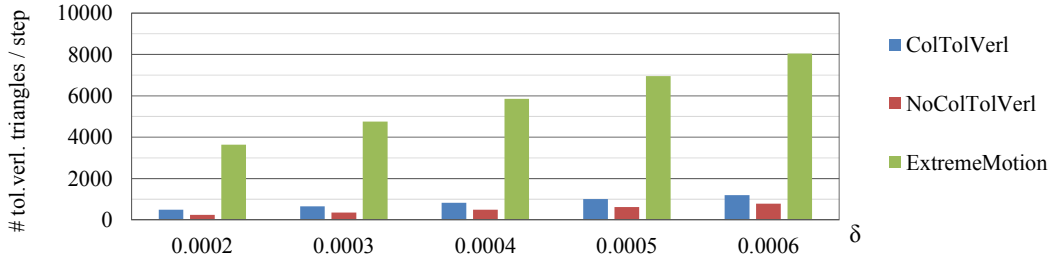


Fig. A.20: Average number of tolerance violating triangles per transformation in the transformation sets of the *Buddha* scenario.

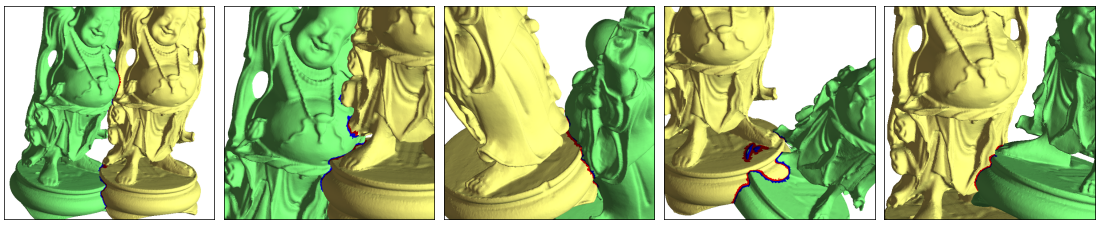


Fig. A.21: Sample positions for one buddha in the transformation set *ExtremeMotion* with tolerance violating triangles displayed in red and blue color.

every position is larger than zero but smaller than or equal to 0.0001. The bottom row in Figure A.19 shows some sample positions and Figure A.20 shows the average number of tolerance violating triangles per position in *NoColTolVerl* for varying tolerance value δ. Also for the *Buddha* scenario there are more tolerance violating primitives for *ColTolVerl* than for *NoColTolVerl*, but the difference between both transformation sets is not as high as in the other scenarios. The reason is that for the positions in *ColTolVerl* the penetration is not that deep because already small penetrations and contacts produce many tolerance violating triangles.

A.4.3. Transformation Set ExtremeMotion

ExtremeMotion is a motion path with 200 positions where both buddhas are penetrating in most steps and there are many tolerance violating triangles in different and large spatial regions. Figure A.21 shows some sample positions and Figure A.20 shows the average number of tolerance violating triangles per position in *ExtremeMotion* for a varying tolerance value δ. As the buddha models penetrate each other, the number of tolerance violating triangles is extremely high for the positions in *ExtremeMotion*, compared to those in *ColTolVerl* and *NoColTolVerl*.

A.5. Scenario Complexity

We define the complexity of a scenario depending on its normalized volumetric extension and the number of triangles in the models. As the models have completely different extensions, we normalize them such that each model fits just into a unit cube. More precisely, we scale each model such that the longest side of its axis aligned bounding box has the length 1. Let l_x, l_y, l_z be extensions of one model in $x-$, $y-$, $z-$direction and $l_{max} = \max(l_x, l_y, l_z)$, then the normalized volumetric extension is given as

$$V_{ext} \;=\; \frac{l_x}{l_{max}} \cdot \frac{l_y}{l_{max}} \cdot \frac{l_z}{l_{max}} \,.$$

Further let N be the number of primitives of the model, then the ratio of the model's normalized volumetric extension and the number of triangles is given as

$$\xi \;=\; \frac{V_{ext}}{N} \;=\; \frac{l_x \cdot l_y \cdot l_z}{N \cdot {l_{max}}^3} \,. \tag{A.1}$$

For a scenario that consists of two different models, ξ_{Scenario} is given as the average of $\xi_{\text{Model 1}}$ and $\xi_{\text{Model 2}}$. We define the *Engine* scenario as our reference scenario and set the complexity equal to 1. The complexity of all other scenarios is given in relation to the *Engine* scenario as

$$\kappa_{\text{Scenario}} = \frac{\xi_{Engine}}{\xi_{\text{Scenario}}} \,. \tag{A.2}$$

The following table shows the ratio of the volume extension and the number of triangles of all of our models and the complexity of all scenarios.

Scenario	Model	ξ	κ_{Scenario}
Engine	engine	$7.4 \cdot 10^{-6}$	
	bodyshell	$6.0 \cdot 10^{-6}$	
	whole scenario (engine + bodyshell)	$6.7 \cdot 10^{-6}$	1.0
EngineBig	engine	$1.0 \cdot 10^{-6}$	
	bodyshell	$0.7 \cdot 10^{-6}$	
	whole scenario (engine + bodyshell)	$0.8 \cdot 10^{-6}$	8.0
Bunny		$11.1 \cdot 10^{-6}$	0.6
Buddha		$0.2 \cdot 10^{-6}$	43.3

A.6. Summary of Statistical Quantities of Various Models

The following tables show and compare the most important statistical quantities of the models in our four scenarios. The first table lists the original statistical quantities. Since the models have completely different extensions, it is not easy to compare the statistical quantities directly. Because of that we have normalized the models and show the normalized quantities in the second and the third table. For the second table we have scaled the models such that the median edge length in every model is equal to 1. For the third table we have chosen from one model of the scenario one of its object extensions and defined it as the reference extension of the scenario. Then we scaled the models of the scenario such that this reference extension is equal to 100.

Original Quantities

Scenario	Model	Variant / Sub-Part	number of triangles	Object Extension (AABB)	defined reference extentsion
Engine	engine		126,820	728 x 732 x 690	1,770
	bodyshell		92,671	1,394 x 1,770 x 2,098	1,770
	bodyshell	sub-part with less tubes	46,618	1,316 x 1,770 x 2,100	1,770
	bodyshell	sub-part of tubes	11,803	264 x 767 x 143	1,770
	engine+bodyshell		219,491		
EngineBig	engine		325,119	124 x 71 x 68	177
	bodyshell		519,266	143 x 177 x 82	177
	engine+bodyshell		844,385		
Bunny			69,451	0.1556 x 0.1544 x 0.1206	0.1556
Buddha			1,087,716	0.0813 x 0.1980 x 0.0814	0.0813

Normalization: Median Edge Length = 1

Scenario	Model	Variant / Sub-Part	number of triangles	Object Extension (AABB)	defined reference extentsion
Engine	engine		126,820	84 x 85 x 80	205
	bodyshell		92,671	120 x 152 x 180	152
	bodyshell	sub-part with less tubes	46,618	101 x 136 x 162	136
	bodyshell	sub-part of tubes	11,803	26 x 75 x 14	174
EngineBig	engine		325,119	271 x 156 x 148	387
	bodyshell		519,266	250 x 309 x 142	309
Bunny			69,451	106 x 105 x 82	106
Buddha			1,087,716	267 x 649 x 267	267

Normalization: Reference Extension of the Scene = 100

Scenario	Model	Variant / Sub-Part	number of triangles	Object Extension (AABB)	defined reference extentsion
Engine	engine		126,820	41 x 41 x 39	100
	bodyshell	sub-part with less tubes	92,671	79 x *100* x 119	100
	bodyshell	sub-part of tubes	46,618	74 x *100* x 119	100
	bodyshell	fine resolution	11,803	15 x 43 x 8	100
EngineBig	engine		325,119	70 x 40 x 38	100
	bodyshell		519,266	81 x *100* x 46	100
Bunny			69,451	*100* x 99 x 78	100
Buddha			1,087,716	*100* x 244 x *100*	100

minimal edge length	maximal edge length	average edge length	**median edge length**	10% of all edge length smaller ($Q_{0.1}$)	10% of all edge length greater ($Q_{0.9}$)	25% of all edge length smaller ($Q_{0.25}$)	25% of all edge length greater ($Q_{0.75}$)
0.00099	358.0	14.05	**8.64**	3.11	28.20	5.02	15.67
0.00510	698.5	21.79	**11.66**	3.89	48.43	6.37	23.19
0.00510	545.3	25.23	**12.98**	3.92	58.13	6.31	30.04
0.17440	534.6	15.40	**10.18**	3.53	23.58	6.48	14.45
0.00099	698.5	17.31	**9.77**	3.35	36.36	5.50	18.30
0.00150	41.7	0.86	**0.46**	0.17	1.72	0.26	0.88
0.00010	78.3	1.41	**0.57**	0.18	2.61	0.30	1.18
0.00010	78.3	1.20	**0.53**	0.17	2.24	0.29	1.04
0.00018	0.00491	0.00147	**0.00147**	0.00101	0.00186	0.00110	0.00173
0.00000	0.00258	0.00036	**0.00030**	0.00010	0.00064	0.00025	0.00042

minimal edge length	maximal edge length	average edge length	**median edge length**	10% of all edge length smaller ($Q_{0.1}$)	10% of all edge length greater ($Q_{0.9}$)	25% of all edge length smaller ($Q_{0.25}$)	25% of all edge length greater ($Q_{0.75}$)
0.00011	41.4	1.63	**1.00**	0.36	3.26	0.58	1.81
0.00044	59.9	1.87	**1.00**	0.33	4.15	0.55	1.99
0.00039	42.0	1.94	**1.00**	0.30	4.48	0.49	2.31
0.01713	52.5	1.51	**1.00**	0.35	2.32	0.64	1.42
0.00327	91.1	1.87	**1.00**	0.36	3.75	0.58	1.93
0.00017	136.8	2.46	**1.00**	0.32	4.55	0.53	2.06
0.12131	3.35	1.00	**1.00**	0.69	1.27	0.75	1.18
0.00000	8.45	1.18	**1.00**	0.34	2.09	0.82	1.38

minimal edge length	maximal edge length	average edge length	**median edge length**	10% of all edge length smaller ($Q_{0.1}$)	10% of all edge length greater ($Q_{0.9}$)	25% of all edge length smaller ($Q_{0.25}$)	25% of all edge length greater ($Q_{0.75}$)
0.00006	20.2	0.79	**0.49**	0.18	1.59	0.28	0.89
0.00029	39.5	1.23	**0.66**	0.22	2.74	0.36	1.31
0.00029	30.8	1.43	**0.73**	0.22	3.28	0.36	1.70
0.00985	30.2	0.87	**0.58**	0.20	1.33	0.37	0.82
0.00085	23.6	0.48	**0.26**	0.09	0.97	0.15	0.50
0.00006	44.3	0.80	**0.32**	0.10	1.47	0.17	0.67
0.11438	3.16	0.95	**0.94**	0.65	1.19	0.71	1.11
0.00000	3.17	0.44	**0.37**	0.13	0.78	0.31	0.52

B. Benchmark Results of the Triangle-Triangle Tolerance Tests

The following Sections B.1 and B.2 show the complete benchmark results of the benchmarks described in Section 2.3. The head maps show the achieved running times in seconds for all triangle tests and for all our test sets. For every tolerance value δ the fastest result is highlighted in white color. In Section B.3 the percentage of tolerance violating triangle pairs is given for all test sets.

B.1. Head Maps Set-Results

	Car, set-results, NoCol							Car, set-results, Col						
	2.5	5	10	15	20	25	30	2.5	5	10	15	20	25	30
SepPlane	1.40	1.59	1.65	1.69	1.69	1.70	1.69	1.84	1.93	2.00	2.03	2.00	2.03	2.03
SepPlaneVV	1.41	1.59	1.61	1.62	1.58	1.56	1.53	1.84	1.91	1.91	1.87	1.79	1.77	1.73
Dual	**1.13**	**1.31**	**1.42**	**1.46**	**1.45**	**1.44**	**1.41**	**1.40**	**1.52**	**1.60**	**1.62**	**1.59**	**1.59**	**1.58**
Dual-P	2.16	2.18	2.06	1.94	1.85	1.76	1.65	2.47	2.46	2.34	2.20	2.05	1.97	1.88
Combined	1.46	1.63	1.61	1.59	1.54	1.50	1.47	1.88	1.86	1.78	1.72	1.65	**1.62**	**1.58**
FeatDist	4.72	4.67	4.55	4.45	4.37	4.30	4.24	4.73	4.60	4.44	4.34	4.26	4.21	4.16
Erbes	3.07	2.93	2.49	2.24	2.06	1.90	1.76	3.52	3.25	2.81	2.50	2.26	2.12	1.97
PQP	4.93	4.71	4.36	4.12	4.00	3.88	3.78	5.27	4.99	4.60	4.35	4.16	4.02	3.90
Möller	0.42	0.40	0.38	0.37	0.35	0.35	0.34	0.44	0.42	0.39	0.38	0.37	0.36	0.35

	Bunny, set-results, NoCol						Bunny, set-results, Col					
	0.0005	0.001	0.0015	0.002	0.0025	0.003	0.0005	0.001	0.0015	0.002	0.0025	0.003
SepPlane	0.61	0.69	0.73	0.75	0.76	0.77	0.83	0.86	0.86	0.85	0.84	0.83
SepPlaneVV	0.60	0.63	0.63	0.63	0.63	0.64	0.78	0.72	0.69	0.67	0.66	0.65
Dual	**0.53**	**0.56**	**0.58**	**0.58**	**0.59**	**0.59**	**0.67**	**0.65**	**0.63**	**0.62**	**0.61**	**0.61**
Dual-P	1.06	0.92	0.81	0.73	0.68	0.64	1.01	0.87	0.76	0.71	0.66	0.63
Combined	0.58	0.61	0.61	0.62	0.62	0.62	0.72	0.69	0.67	0.65	0.64	0.64
FeatDist	4.98	4.82	4.68	4.58	4.51	4.44	4.96	4.77	4.64	4.54	4.47	4.41
Erbes	1.24	0.99	0.85	0.75	0.68	0.64	1.10	0.90	0.78	0.71	0.66	**0.62**
PQP	4.90	4.56	4.30	4.12	3.97	3.84	4.85	4.47	4.25	4.07	3.94	3.82
Möller	0.34	0.33	0.33	0.33	0.33	0.33	0.37	0.35	0.35	0.34	0.34	0.34

	Buddha, set-results, NoCol						Buddha, set-results, Col					
	0.00005	0.0002	0.0004	0.0006	0.0008	0.001	0.00005	0.0002	0.0004	0.0006	0.0008	0.001
SepPlane	0.61	0.73	0.76	0.77	0.77	0.71	0.72	0.81	0.81	0.84	0.81	0.71
SepPlaneVV	0.61	0.67	0.66	0.65	0.64	0.62	0.72	0.72	0.68	0.68	0.66	0.61
Dual	**0.53**	**0.59**	**0.60**	**0.59**	**0.58**	**0.56**	**0.61**	**0.64**	**0.62**	**0.61**	**0.60**	**0.56**
Dual-P	1.28	1.13	0.94	0.82	0.74	0.65	1.27	1.13	0.94	0.81	0.73	0.63
Combined	0.59	0.64	0.64	0.63	0.63	0.61	0.70	0.68	0.66	0.66	0.64	0.60
FeatDist	4.94	4.72	4.55	4.44	4.35	4.18	4.96	4.71	4.53	4.41	4.34	4.13
Erbes	1.66	1.31	1.03	0.87	0.77	0.66	1.57	1.29	1.03	0.87	0.76	0.63
PQP	5.19	4.69	4.30	4.05	3.85	3.43	5.21	4.68	4.31	4.02	3.85	3.39
Möller	0.34	0.34	0.32	0.32	0.31	0.31	0.36	0.33	0.32	0.32	0.32	0.32

B.2. Head Maps Pair-Results

	Car, pair-results, NoCol							Car, pair-results, Col						
	2.5	5	10	15	20	25	30	2.5	5	10	15	20	25	30
SepPlane	1.78	2.79	4.67	5.74	6.61	7.04	7.70	4.54	5.48	6.87	7.58	7.94	8.34	8.37
SepPlaneVV	1.76	2.56	3.49	3.64	3.61	3.02	2.75	4.46	4.95	5.11	4.59	4.09	3.74	3.09
Dual	**1.38**	**1.99**	2.69	2.81	2.78	2.37	2.17	**3.17**	3.59	3.76	3.43	3.09	2.84	2.39
Dual-P	2.34	2.65	2.91	2.81	2.67	2.15	1.90	3.63	3.81	3.70	3.24	2.83	2.55	2.07
Combined	1.72	2.22	**2.56**	**2.53**	2.45	2.12	1.98	4.17	4.01	3.59	3.07	2.75	2.44	2.13
FeatDist	4.57	4.19	3.37	2.83	2.40	2.07	1.70	3.92	**3.32**	**2.56**	**2.13**	**1.87**	**1.64**	**1.51**
Erbes	3.25	3.27	2.93	**2.55**	**2.25**	**1.72**	**1.49**	5.03	4.47	3.59	2.82	2.38	1.99	1.57
PQP	4.98	4.81	4.53	4.33	4.17	3.88	3.76	5.98	5.70	5.26	4.90	4.67	4.42	4.14
Möller	0.42	0.41	0.39	0.37	0.36	0.36	0.35	0.50	0.48	0.44	0.41	0.40	0.39	0.37

	Bunny, pair-results, NoCol						Bunny, pair-results, Col					
	0.0005	0.001	0.0015	0.002	0.0025	0.003	0.0005	0.001	0.0015	0.002	0.0025	0.003
SepPlane	1.63	3.26	4.27	4.92	5.37	5.65	3.12	4.48	5.27	5.75	6.16	6.44
SepPlaneVV	1.32	1.53	1.39	1.32	1.26	1.22	2.39	1.89	1.58	1.43	1.35	1.30
Dual	**1.04**	**1.25**	1.19	1.14	1.10	1.07	**1.82**	1.57	1.37	1.26	1.20	1.14
Dual-P	1.45	1.36	1.14	1.02	0.93	0.88	1.89	1.46	1.17	1.02	0.94	0.87
Combined	**1.04**	**1.28**	1.25	1.23	1.19	1.17	**1.77**	1.58	1.42	1.33	1.28	1.24
FeatDist	4.64	3.85	3.29	2.88	2.56	2.33	4.17	3.41	2.92	2.58	2.30	2.08
Erbes	1.53	1.29	**1.06**	**0.92**	**0.83**	**0.77**	1.83	**1.34**	**1.06**	**0.91**	**0.82**	**0.75**
PQP	4.97	4.63	4.35	4.13	3.96	3.82	5.22	4.78	4.45	4.19	4.00	3.81
Möller	0.34	0.33	0.33	0.32	0.32	0.32	0.41	0.39	0.37	0.36	0.35	0.35

	Buddha, pair-results, NoCol						Buddha, pair-results, Col					
	0.00005	0.0002	0.0004	0.0006	0.0008	0.001	0.00005	0.0002	0.0004	0.0006	0.0008	0.001
SepPlane	1.51	3.47	4.89	5.61	6.18	5.94	1.88	3.83	5.29	6.01	6.46	6.19
SepPlaneVV	1.45	1.99	1.69	1.43	1.33	1.19	1.82	2.24	1.80	1.49	1.37	1.21
Dual	**1.17**	**1.58**	1.39	1.21	1.13	1.01	**1.40**	**1.78**	1.50	1.27	1.17	1.04
Dual-P	1.74	1.77	1.36	1.07	0.94	0.81	1.85	1.87	1.39	1.07	0.94	0.81
Combined	1.27	**1.55**	1.44	1.30	1.25	1.14	1.65	**1.73**	1.54	1.36	1.28	1.16
FeatDist	4.68	3.74	2.98	2.49	2.12	2.06	4.64	3.63	2.82	2.35	2.02	1.97
Erbes	2.13	1.73	**1.26**	**0.96**	**0.82**	**0.70**	2.33	1.82	**1.29**	**0.96**	**0.82**	**0.69**
PQP	5.34	4.90	4.44	4.10	3.83	3.53	5.49	5.02	4.52	4.11	3.90	3.61
Möller	0.36	0.34	0.33	0.33	0.32	0.32	0.40	0.37	0.35	0.34	0.33	0.33

B.3. Percentage on Tolerance Violating Triangle Pairs

		Engine						
		2.5	5	10	15	20	25	30
pair-results,	ColTolVerl	34.8	43.8	56.6	63.0	66.9	70.7	72.2
pair-results,	NoColTolVerl	5.4	14.7	32.9	43.9	53.0	58.7	66.9
set-results,	ColTolVerl	3.7	4.1	4.6	4.9	5.0	5.4	5.6
set-results,	NoColTolVerl	0.4	0.8	1.4	1.9	2.4	2.6	3.0

		Bunny					
		0.0005	0.001	0.0015	0.002	0.0025	0.003
pair-results,	ColTolVerl	21.7	32.6	40.8	47.0	52.6	56.9
pair-results,	NoColTolVerl	9.0	22.2	32.2	39.9	46.1	50.6
set-results,	ColTolVerl	1.6	1.8	2.0	2.0	2.1	2.1
set-results,	NoColTolVerl	0.5	0.9	1.2	1.4	1.5	1.6

		Buddha					
		0.00005	0.0002	0.0004	0.0006	0.0008	0.001
pair-results,	ColTolVerl	12.6	28.2	42.0	50.8	57.5	57.1
pair-results,	NoColTolVerl	9.2	25.0	38.5	47.3	55.0	54.7
set-results,	ColTolVerl	1.2	1.6	1.8	1.8	1.9	1.2
set-results,	NoColTolVerl	0.7	1.2	1.4	1.4	1.5	1.0

C. Benchmark Results of the Performance Tests

The following sections show the complete results of the benchmarks described in Sections 4.5 and 4.6. The head maps show the achieved tests per seconds (tps) for all considered approaches in all our 12 test cases. For every tolerance value δ the fastest result is highlighted in bold.

C.1. Results of the Engine Scenario

ColTolVerl

δ	average number tol. viol. triangles / step	online approches						offline appoch
		BVH OBB Standard (single core)	BVH OBB Erbes (single core)	our symmetric approach		our asymmetric approch		BVH OBB Erbes
				basic	using core-primitives	basic	using core-primitives	
5	560	95	86	188	-	**663**	-	533
10	846	47	70	139	-	538	**651**	456
15	1.162	27	59	83	-	483	**541**	387
20	1.511	17	50	66	66	445	**467**	331
25	1.897	12	42	56	57	384	**404**	284
30	2.318	8	36	39	40	336	**354**	243

NoColTolVerl

δ	average number tol. viol. triangles / step	online approches						offline appoch
		BVH OBB Standard (single core)	BVH OBB Erbes (single core)	our symmetric approach		our asymmetric approch		BVH OBB Erbes
				basic	using core-primitives	basic	using core-primitives	
5	27	735	400	609	-	**2.003**	-	1.650
10	93	298	274	386	-	**1.451**	1.316	1.302
15	195	145	196	206	-	1.043	**1.094**	1.048
20	332	79	148	149	147	853	**903**	839
25	504	47	113	113	112	726	**769**	679
30	712	30	90	74	74	633	**669**	556

DisassMotion

δ	average number tol. viol. triangles / step	online approches						offline appoch
		BVH OBB Standard (single core)	BVH OBB Erbes (single core)	our symmetric approach		our asymmetric approch		BVH OBB Erbes
				basic	using core-primitives	basic	using core-primitives	
5	223	114	141	352	-	**1.141**	-	521
10	392	48	101	232	-	**821**	805	418
15	598	26	75	133	-	**627**	606	330
20	861	15	57	95	91	**494**	492	268
25	1.180	10	47	64	60	383	**393**	207
30	1.588	7	37	42	42	310	**326**	165

C.2. Selected Speed-Ups in the Engine Scenario

Speed-Up in ColTolVerl

δ	basic sym. appr. / BVH Standard (single core)	basic sym. appr. / BVH Erbes (single core)	sym. appr. & core-tr. / BVH Erbes (single core)	sym. appr. & core-tr. / basic sym. appr.	basic asym. appr. / basic sym. appr.	asym. appr. & core-tr. / basic asym. appr.	basic asym. appr. / BVH Erbes (offline)	asym. appr. & core-tr. / BVH Erbes (offline)
5	1.98	2.19	-	-	3.53	-	1.24	-
10	2.98	1.97	-	-	3.87	1.21	1.18	1.43
15	3.06	1.40	-	-	5.84	1.12	1.25	1.40
20	3.82	1.33	1.32	1.00	6.75	1.05	1.34	1.41
25	4.78	1.32	1.35	1.02	6.86	1.05	1.35	1.42
30	4.67	1.07	1.10	1.03	8.66	1.06	1.38	1.46

Speed-Up in NoColTolVerl

δ	basic sym. appr. / BVH Standard (single core)	basic sym. appr. / BVH Erbes (single core)	sym. appr. & core-tr. / BVH Erbes (single core)	sym. appr. & core-tr. / basic sym. appr.	basic asym. appr. / basic sym. appr.	asym. appr. & core-tr. / basic asym. appr.	basic asym. appr. / BVH Erbes (offline)	asym. appr. & core-tr. / BVH Erbes (offline)
5	0.83	1.52	-	-	3.29	-	1.21	-
10	1.29	1.41	-	-	3.76	0.91	1.11	1.01
15	1.43	1.05	-	-	5.06	1.05	1.00	1.04
20	1.88	1.01	0.99	0.98	5.71	1.06	1.02	1.08
25	2.40	1.00	0.99	0.99	6.42	1.06	1.07	1.13
30	2.48	0.82	0.83	1.01	8.58	1.06	1.14	1.20

Speed-Up in DisassMotion

δ	basic sym. appr. / BVH Standard (single core)	basic sym. appr. / BVH Erbes (single core)	sym. appr. & core-tr. / BVH Erbes (single core)	sym. appr. & core-tr. / basic sym. appr.	basic asym. appr. / basic sym. appr.	asym. appr. & core-tr. / basic asym. appr.	basic asym. appr. / BVH Erbes (offline)	asym. appr. & core-tr. / BVH Erbes (offline)
5	3.09	2.50	-	-	3.24	-	2.19	-
10	4.79	2.29	-	-	3.54	0.98	1.96	1.93
15	5.16	1.78	-	-	4.71	0.97	1.90	1.84
20	6.19	1.67	1.59	0.95	5.17	1.00	1.85	1.84
25	6.45	1.37	1.29	0.94	5.98	1.02	1.85	1.90
30	6.21	1.13	1.13	1.00	7.46	1.05	1.87	1.97

C.3. Results of the EngineBig Scenario

ColTolVerl

δ	average number tol. viol. triangles / step	online approches						offline
		BVH OBB Standard (single core)	BVH OBB Erbes (single core)	our symmetric approach		our asymmetric approch		BVH OBB Erbes
				basic	using core-primitives	basic	using core-primitives	
0.5	378	93	89	204	-	**691**	-	353
1.0	638	40	68	112	-	528	**532**	293
1.5	970	20	52	64	63	399	**402**	251
2.0	1,384	12	40	47	52	309	**357**	204
2.5	1,885	7	31	30	38	271	**306**	167
3.0	2,476	5	24	24	28	220	**243**	137

NoColTolVerl

δ	average number tol. viol. triangles / step	online approches						offline
		BVH OBB Standard (single core)	BVH OBB Erbes (single core)	our symmetric approach		our asymmetric approch		BVH OBB Erbes
				basic	using core-primitives	basic	using core-primitives	
0.5	47	390	194	431	-	**1,187**	-	571
1.0	170	138	136	192	-	821	**826**	456
1.5	369	59	96	102	112	**609**	593	371
2.0	645	29	71	71	73	460	**483**	296
2.5	995	16	52	44	54	382	**401**	241
3.0	1,428	9	39	35	39	304	**329**	195

DisassMotion

δ	average number tol. viol. triangles / step	online approches						offline
		BVH OBB Standard (single core)	BVH OBB Erbes (single core)	our symmetric approach		our asymmetric approch		BVH OBB Erbes
				basic	using core-primitives	basic	using core-primitives	
0.5	1,275	28	31	76	-	**307**	-	160
1.0	2,140	12	24	40	-	221	**227**	126
1.5	3,173	6	19	22	25	163	**168**	99
2.0	4,416	3	15	15	16	118	**135**	77
2.5	5,865	2	12	9	12	88	**104**	58
3.0	7,562	1	9	7	8	68	**76**	45

C.4. Selected Speed-Ups in the EngineBig Scenario

Speed-Up in ColTolVerl

δ	basic sym. appr. / BVH Standard (single core)	basic sym. appr. / BVH Erbes (single core)	sym. appr. & core-tr. / BVH Erbes (single core)	sym. appr. & core-tr. / basic sym. appr.	basic asym. appr. / basic sym. appr.	asym. appr. & core-tr. / basic asym. appr.	basic asym. appr. / BVH Erbes (offline)	asym. appr. & core-tr. / BVH Erbes (offline)
0.5	2.19	2.29	-	-	3.38	-	1.96	-
1.0	2.81	1.66	-	-	4.72	1.01	1.80	1.82
1.5	3.11	1.21	1.19	0.98	6.27	1.01	1.59	1.60
2.0	4.06	1.18	1.29	1.09	6.51	1.16	1.51	1.75
2.5	4.23	0.98	1.22	1.24	8.95	1.13	1.62	1.83
3.0	5.15	0.99	1.17	1.18	9.17	1.11	1.60	1.77

Speed-Up in NoColTolVerl

δ	basic sym. appr. / BVH Standard (single core)	basic sym. appr. / BVH Erbes (single core)	sym. appr. & core-tr. / BVH Erbes (single core)	sym. appr. & core-tr. / basic sym. appr.	basic asym. appr. / basic sym. appr.	asym. appr. & core-tr. / basic asym. appr.	basic asym. appr. / BVH Erbes (offline)	asym. appr. & core-tr. / BVH Erbes (offline)
0.5	1.11	2.23	-	-	2.75	-	2.08	-
1.0	1.39	1.41	-	-	4.29	1.01	1.80	1.81
1.5	1.74	1.06	1.16	1.10	5.98	0.97	1.64	1.60
2.0	2.50	1.01	1.03	1.01	6.43	1.05	1.55	1.63
2.5	2.82	0.85	1.04	1.23	8.73	1.05	1.59	1.66
3.0	3.83	0.89	1.00	1.12	8.74	1.08	1.56	1.68

Speed-Up in DisassMotion

δ	basic sym. appr. / BVH Standard (single core)	basic sym. appr. / BVH Erbes (single core)	sym. appr. & core-tr. / BVH Erbes (single core)	sym. appr. & core-tr. / basic sym. appr.	basic asym. appr. / basic sym. appr.	asym. appr. & core-tr. / basic asym. appr.	basic asym. appr. / BVH Erbes (offline)	asym. appr. & core-tr. / BVH Erbes (offline)
0.5	2.69	2.43	-	-	4.06	-	1.92	-
1.0	3.44	1.68	-	-	5.56	1.03	1.76	1.81
1.5	3.70	1.16	1.31	1.13	7.48	1.03	1.64	1.70
2.0	4.59	1.03	1.05	1.02	7.62	1.15	1.52	1.75
2.5	4.57	0.80	1.01	1.26	9.33	1.18	1.52	1.79
3.0	5.49	0.77	0.87	1.12	9.23	1.13	1.50	1.70

C.5. Results of the Bunny Scenario

ColTolVerl

δ	average number tol. viol. triangles / step	online approches						offline
		BVH OBB Standard (single core)	BVH OBB Erbes (single core)	our symmetric approach		our asymmetric approch		BVH OBB Erbes
				basic	using core-primitives	basic	using core-primitives	
0.001	601	80	212	240	-	818	942	**1,109**
0.002	993	28	139	176	-	636	788	**809**
0.003	1,395	13	95	101	-	581	**629**	593
0.004	1,813	7	67	81	81	471	**512**	433

NoColTolVerl

δ	average number tol. viol. triangles / step	online approches						offline
		BVH OBB Standard (single core)	BVH OBB Erbes (single core)	our symmetric approach		our asymmetric approch		BVH OBB Erbes
				basic	using core-primitives	basic	using core-primitives	
0.001	156	263	531	438	-	1,698	1,318	**1,844**
0.002	392	73	300	291	-	1,137	1,130	**1,161**
0.003	671	30	185	154	-	809	**867**	777
0.004	988	16	121	117	115	619	**697**	565

ExtremeMotion

δ	average number tol. viol. triangles / step	online approches						offline
		BVH OBB Standard (single core)	BVH OBB Erbes (single core)	our symmetric approach		our asymmetric approch		BVH OBB Erbes
				basic	using core-primitives	basic	using core-primitives	
0.001	4,074	12	36	44	-	**198**	182	177
0.002	6,678	4	23	32	-	145	**154**	126
0.003	9,251	2	16	19	-	114	**118**	91
0.004	11,772	1	12	15	15	93	**96**	68

C.6. Selected Speed-Ups in the Bunny Scenario

Speed-Up in ColTolVerl

δ	basic sym. appr. / BVH Standard (single core)	basic sym. appr. / BVH Erbes (single core)	sym. appr. & core-tr. / BVH Erbes (single core)	sym. appr. & core-tr. / basic sym. appr.	basic asym. appr. / basic sym. appr.	asym. appr. & core-tr. / basic asym. appr.	basic asym. appr. / BVH Erbes (offline)	asym. appr. & core-tr. / BVH Erbes (offline)
0.001	3.01	1.14	-	-	3.40	1.15	0.74	0.85
0.002	6.30	1.27	-	-	3.62	1.24	0.79	0.97
0.003	7.57	1.06	-	-	5.77	1.08	0.98	1.06
0.004	10.97	1.21	1.21	1.00	5.79	1.09	1.09	1.18

Speed-Up in NoColTolVerl

δ	basic sym. appr. / BVH Standard (single core)	basic sym. appr. / BVH Erbes (single core)	sym. appr. & core-tr. / BVH Erbes (single core)	sym. appr. & core-tr. / basic sym. appr.	basic asym. appr. / basic sym. appr.	asym. appr. & core-tr. / basic asym. appr.	basic asym. appr. / BVH Erbes (offline)	asym. appr. & core-tr. / BVH Erbes (offline)
0.001	1.67	0.82	-	-	3.88	0.78	0.92	0.71
0.002	4.01	0.97	-	-	3.90	0.99	0.98	0.97
0.003	5.07	0.83	-	-	5.24	1.07	1.04	1.12
0.004	7.54	0.97	0.95	0.98	5.28	1.13	1.09	1.23

Speed-Up in ExtremeMotion

δ	basic sym. appr. / BVH Standard (single core)	basic sym. appr. / BVH Erbes (single core)	sym. appr. & core-tr. / BVH Erbes (single core)	sym. appr. & core-tr. / basic sym. appr.	basic asym. appr. / basic sym. appr.	asym. appr. & core-tr. / basic asym. appr.	basic asym. appr. / BVH Erbes (offline)	asym. appr. & core-tr. / BVH Erbes (offline)
0.001	3.58	1.22	-	-	4.55	0.92	1.12	1.03
0.002	7.44	1.38	-	-	4.59	1.06	1.15	1.22
0.003	9.26	1.21	-	-	6.07	1.03	1.26	1.29
0.004	13.43	1.28	1.28	1.00	6.04	1.03	1.38	1.42

C.7. Results of the Buddha Scenario

ColTolVerl

δ	average number tol. viol. triangles / step	online approches						offline
		BVH OBB Standard (single core)	BVH OBB Erbes (single core)	our symmetric approach		our asymmetric approch		BVH OBB Erbes
				basic	using core-primitives	basic	using core-primitives	
0.0002	489	99	91	181	-	**444**	-	252
0.0003	655	59	82	153	-	**400**	-	242
0.0004	828	38	76	115	-	363	**393**	230
0.0005	1,008	26	68	101	-	323	**379**	219
0.0006	1,196	19	62	90	-	288	**349**	208

NoColTolVerl

δ	average number tol. viol. triangles / step	online approches						offline
		BVH OBB Standard (single core)	BVH OBB Erbes (single core)	our symmetric approach		our asymmetric approch		BVH OBB Erbes
				basic	using core-primitives	basic	using core-primitives	
0.0002	242	161	108	228	-	**534**	-	279
0.0003	358	96	100	189	-	**482**	-	268
0.0004	488	61	91	127	-	431	**457**	255
0.0005	630	41	84	114	-	384	**426**	244
0.0006	786	29	76	101	-	335	**385**	230

ExtremeMotion

δ	average number tol. viol. triangles / step	online approches						offline
		BVH OBB Standard (single core)	BVH OBB Erbes (single core)	our symmetric approach		our asymmetric approch		BVH OBB Erbes
				basic	using core-primitives	basic	using core-primitives	
0.0002	3,642	17	30	43	-	**144**	-	117
0.0003	4,747	10	26	37	-	**135**	-	107
0.0004	5,856	6	22	23	-	**118**	-	98
0.0005	6,956	4	20	21	-	106	**113**	89
0.0006	8,042	3	18	19	-	97	**101**	81

C.8. Selected Speed-Ups in the Buddha Scenario

Speed-Up in ColTolVerl

δ	basic sym. appr. / BVH Standard (single core)	basic sym. appr. / BVH Erbes (single core)	sym. appr. & core-tr. / BVH Erbes (single core)	sym. appr. & core-tr. / basic sym. appr.	basic asym. appr. / basic sym. appr.	asym. appr. & core-tr. / basic asym. appr.	basic asym. appr. / BVH Erbes (offline)	asym. appr. & core-tr. / BVH Erbes (offline)
0.0002	1.83	1.98	-	-	2.46	-	1.76	-
0.0003	2.61	1.85	-	-	2.62	-	1.65	-
0.0004	3.03	1.50	-	-	3.16	1.08	1.57	1.71
0.0005	3.89	1.48	-	-	3.19	1.17	1.47	1.73
0.0006	4.82	1.44	-	-	3.20	1.21	1.39	1.68

Speed-Up in NoColTolVerl

δ	basic sym. appr. / BVH Standard (single core)	basic sym. appr. / BVH Erbes (single core)	sym. appr. & core-tr. / BVH Erbes (single core)	sym. appr. & core-tr. / basic sym. appr.	basic asym. appr. / basic sym. appr.	asym. appr. & core-tr. / basic asym. appr.	basic asym. appr. / BVH Erbes (offline)	asym. appr. & core-tr. / BVH Erbes (offline)
0.0002	1.41	2.11	-	-	2.34	-	1.91	-
0.0003	1.97	1.88	-	-	2.55	-	1.80	-
0.0004	2.08	1.39	-	-	3.40	1.06	1.69	1.79
0.0005	2.79	1.36	-	-	3.36	1.11	1.58	1.75
0.0006	3.51	1.33	-	-	3.32	1.15	1.46	1.67

Speed-Up in ExtremeMotion

δ	basic sym. appr. / BVH Standard (single core)	basic sym. appr. / BVH Erbes (single core)	sym. appr. & core-tr. / BVH Erbes (single core)	sym. appr. & core-tr. / basic sym. appr.	basic asym. appr. / basic sym. appr.	asym. appr. & core-tr. / basic asym. appr.	basic asym. appr. / BVH Erbes (offline)	asym. appr. & core-tr. / BVH Erbes (offline)
0.0002	2.56	1.43	-	-	3.34	-	1.23	-
0.0003	3.75	1.43	-	-	3.69	-	1.26	-
0.0004	3.70	1.05	-	-	5.06	-	1.20	-
0.0005	4.79	1.05	-	-	5.12	1.06	1.20	1.27
0.0006	6.02	1.04	-	-	5.13	1.05	1.19	1.24

C.9. GPU Results of the Engine Scenario

ColTolVerl

	C1		C2		C3		C4		C5	
	CPU: i7-740QM GPU: GeForce GTX 460M		CPU: i7-950 GPU: GeForce GTX 460		CPU: i7-980X GPU: GeForce GTX 480		CPU: i7-4800MQ GPU: GeForce GT 730M		CPU: i7-980X GPU: GeForce GTX 650 Ti	
δ	CPU	CPU/GPU	CPU	CPU/GPU	CPU	CPU/GPU	CPU	CPU/GPU	CPU	CPU/GPU
5	108	169	182	216	255	**310**	188	198	255	269
10	76	130	135	175	199	**262**	139	131	199	182
15	44	78	82	115	121	**181**	83	83	121	128
20	35	62	65	90	96	**144**	66	61	96	96
25	30	50	54	70	82	**111**	56	47	82	73
30	19	35	38	54	58	**88**	39	36	58	60

NoColTolVerl

	C1		C2		C3		C4		C5	
	CPU: i7-740QM GPU: GeForce GTX 460M		CPU: i7-950 GPU: GeForce GTX 460		CPU: i7-980X GPU: GeForce GTX 480		CPU: i7-4800MQ GPU: GeForce GT 730M		CPU: i7-980X GPU: GeForce GTX 650 Ti	
δ	CPU	CPU/GPU	CPU	CPU/GPU	CPU	CPU/GPU	CPU	CPU/GPU	CPU	CPU/GPU
5	295	395	543	415	**718**	547	609	698	**718**	689
10	204	305	362	334	**463**	445	386	380	**463**	444
15	111	194	196	240	279	**341**	206	240	279	306
20	84	138	144	172	207	**250**	149	154	207	197
25	64	95	111	122	158	**184**	113	104	158	156
30	42	71	72	101	105	**156**	74	78	105	117

DisassMotion

	C1		C2		C3		C4		C5	
	CPU: i7-740QM GPU: GeForce GTX 460M		CPU: i7-950 GPU: GeForce GTX 460		CPU: i7-980X GPU: GeForce GTX 480		CPU: i7-4800MQ GPU: GeForce GT 730M		CPU: i7-980X GPU: GeForce GTX 650 Ti	
δ	CPU	CPU/GPU	CPU	CPU/GPU	CPU	CPU/GPU	CPU	CPU/GPU	CPU	CPU/GPU
5	183	248	289	331	413	**441**	352	370	413	439
10	125	168	194	217	287	**312**	232	189	287	263
15	74	113	123	158	174	**230**	133	115	174	164
20	52	80	88	119	129	**181**	95	79	129	124
25	37	55	62	82	86	**127**	64	55	86	89
30	24	40	40	60	60	**96**	42	36	60	57

Bibliography

[1] O. Devillers and P. Guigue. Faster Triangle-Triangle Intersection Tests. Technical Report 4488, INRIA, 2002.

[2] H. Eggleston. *Convexity*. Cambridge Tracts in Mathematics. Cambridge University Press, 1958.

[3] R. Erbes. *Effcient Parallel Proximity Queries and an Application to Highly Complex Motion Planning Problems with Many Narrow Passages*. PhD thesis, Johannes-Gutenberg Universität Mainz, 2013.

[4] R. Erbes, A. Mantel, E. Schömer, and N. Wolpert. Triangle-Triangle Tolerance Tests. In *28th European Workshop on Computational Geometry*, pages 105–108, 2012.

[5] C. Ericson. *Real-Time Collision Detection*. Morgan Kaufmann, 2005.

[6] M. Figueiredo and T. Fernando. An Efficient Parallel Collision Detection Algorithm for Virtual Prototype Environments. In *Proc. Tenth International Conference on Parallel and Distributed Systems*, pages 249–256, 2004.

[7] M. Figueiredo, L. Marcelino, and T. Fernando. A Survey on Collision Detection Techniques for Virtual Environments. In *Proc. of V Symposium in Virtual Reality*, pages 285–307, 2002.

[8] N. Geismann, M. Hemmer, and E. Schömer. Computing a 3-dimensional Cell in an Arrangement of Quadrics: Exactly and Actually! In *Proc. 17th Annu. ACM Sympos. Comput. Geom.*, pages 264–271, 2001.

[9] R. J. Geraerts and M. H. Overmars. Creating High-quality Paths for Motion Planning. *International Journal of Robotics Research*, 26:845–863, 2007.

[10] E. G. Gilbert, D. W. Johnson, and S. S. Keerthi. A Fast Procedure for Computing the Distance Between Complex Objects in Three-Dimensional Space. In *Proc. IEEE Int. Conf. on Robotics and Automation*, page 18831889, 1987.

[11] M. Gissler, M. Ihmsen, and M. Teschner. Efficient Uniform Grids for Collision Handling in Medical Simulators. In *GRAPP*, pages 79–84, 2011.

[12] S. Gottschalk. *Collision Queries using Orientated Bounding Boxes*. PhD thesis, Chepal Hill, 2000.

[13] S. Gottschalk, M. C. Lin, and D. Manocha. OBBTree: A Hierarchical Structure for Rapid Interference Detection. In *Proc. of Conf. on Computer Graphics and Interactive Techniques*, pages 171–180, 1996.

[14] S. L. Grand. Broad-Phase Collision Detection with CUDA. In *GPU Gems 3*. Addison-Wesley Professional, 2007.

[15] T. He. Fast Collision Detection Using QuOSPO Trees. In *Proc. of the 1999 Symposium on Interactive 3D Graphics*, I3D '99, pages 55–62, 1999.

[16] C. Hecker. Physics, Part 3: Collision Response. *Game Developer Magazine, March 1997*, pages 11—18, 1997.

[17] M. Held. ERIT - A Collection of Efficient and Reliable Intersection Tests. *Journal of Graphics Tools*, 2:25–44, 1998.

[18] D. Hsu, T. Jiang, J. Reif, and Z. Sun. The Bridge Test for Sampling Narrow Passages with Probabilistic Roadmap Planners. In *IEEE Int. Conf. on Robotics and Automation*, pages 4420–4426, 2003.

[19] C.-K. Hung and D. Ierardi. Constructing Convex Hulls of Quadratic Surface Patches. In *Proc. of 7th CCCG (Canadian Conference on Computational Geometry)*, pages 255–260, 1995.

[20] ISO/IEC 11172-2:1993, Information technology – Coding of moving pictures and associated audio for digital storage media at up to about 1,5 Mbit/s – Part 2: Video, 1993.

[21] P. Jiménez, F. Thomas, and C. Torras. 3D Collision Detection: A Survey. *Computers and Graphics*, 25:269–285, 2000.

[22] L. Kavan. Rigid Body Collision Response. In *Proc. of the 7th Central European Seminar on Computer Graphics*, 2003.

[23] D. Kim, J.-P. Heo, J. Huh, J. Kim, and S.-E. Yoon. HPCCD: Hybrid Parallel Continuous Collision Detection using CPUs and GPUs. *Computer Graphics Forum (Pacific Graphics)*, 28(7), 2009.

[24] J. T. Klosowski, M. Held, J. S. Mitchell, H. Sowizral, and K. Zikan. Efficient Collision Detection Using Bounding Volume Hierarchies of k-DOPs. *IEEE Transactions on Visualization and Computer Graphics*, 4(1):21–36, 1998.

[25] J. J. Kuffner and S. M. LaValle. RRT-Connect: An Efficient Approach to Single-Query Path Planning. In *Proc. IEEE Int'l Conf. on Robotics and Automation*, pages 995–1001, 2000.

[26] A. Lagae and P. Dutré. Compact, Fast and Robust Grids for Ray Tracing. *Computer Graphics Forum (Proceedings of the 19th Eurographics Symposium on Rendering)*, 27(4):1235–1244, 2008.

[27] E. Larsen, S. Gottschalk, M. Lin, and D. Manocha. PQP - A Proximity Query Package, 1999. `http://gamma.cs.unc.edu/SSV/`.

[28] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha. Fast Proximity Queries with Swept Sphere Volumes. Technical Report TR99-018, University of North Carolina, Department of Computer Science, 1999.

[29] E. Larsen, S. Gottschalk, M. C. Lin, and D. Manocha. Fast Distance Queries with Rectangular Swept Sphere Volumes. In *Proc. of IEEE Int. Conference on Robotics and Automation*, pages 3719–3726, 2000.

[30] C. Lauterbach, Q. Mo, and D. Manocha. gProximity: Hierarchical GPU-based Operations for Collision and Distance Queries. *Comput. Graph. Forum*, 29(2):419–428, 2010.

[31] S. M. Lavalle. *Motion Planning Algorithms*. Cambridge University Press, 2006. `http://planning.cs.uiuc.edu/`.

[32] M. C. Lin and J. F. Canny. A Fast Algorithm for Incremental Distance Calculation. In *In IEEE International Conference on Robotics and Automation*, pages 1008–1014, 1991.

[33] M. C. Lin and S. Gottschalk. Collision Detection Between Geometric Models: A Survey. In *Proc. of IMA Conference on Mathematics of Surfaces*, pages 37–56, 1998.

[34] M. C. Lin and D. Manocha. Collision and Proximity Queries. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, page 787–808. CRC Press, Inc., 2003.

[35] V. J. Lumelsky. On Fast Computation of Distance Between Line Segments. *In Information Processing Letters*, 21:55–61, 1985.

[36] W. A. McNeely, K. D. Puterbaugh, and J. J. Troy. Six degree-of-freedom Haptic Rendering using Voxel Sampling. In *Proc. of Conf. on Computer Graphics and Interactive Techniques*, SIGGRAPH '99, pages 401–408, 1999.

[37] B. Mirtich. *Impulse-based Dynamic Simulation of Rigid Body Systems*. PhD thesis, University of California, Berkeley, 1996.

[38] T. Möller. A Fast Triangle-Triangle Intersection Test. *Journal of Graphics Tools*, 2:25–30, 1997.

[39] NVIDIA. CUDA C Programming Guide. `http://docs.nvidia.com/cuda/cuda-c-programming-guide/`.

[40] NVIDIA. CUDA Zone. `https://developer.nvidia.com/cuda-zone`.

[41] NVIDIA. Whitepaper: NVIDIA's Next Generation CUDA Compute Architecture: Fermi.

[42] S. Pabst, A. Koch, and W. Straßer. Fast and Scalable CPU/GPU Collision Detection for Rigid and Deformable Surfaces. *Computer Graphics Forum*, 29(5):1605–1612, 2010.

[43] J. Pan and D. Manocha. Bi-level Locality Sensitive Hashing for k-Nearest Neighbor Computation. In *Data Engineering (ICDE), 2012 IEEE 28th International Conference on*, pages 378–389, 2012.

[44] T. Reithmann. Schnelle Kollisionserkennung mit Hilfe des Voxmap-Pointshell-Algorithmus. Master's thesis (Diplomarbeit), Johannes-Gutenberg Universität Mainz / DaimlerChrysler AG Forschungszentrum Ulm, 2004.

[45] M. Renz, C. Preusche, M. Pötke, H.-P. Kriegel, and G. Hirzinger. Stable Haptic Interaction with Virtual Environments Using an Adapted Voxmap-Pointshell Algorithm. In *In Proc. Eurohaptics*, pages 149–154, 2001.

[46] D. Salomon. *Data Compression: The Complete Reference*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[47] H. Samet. *The Quadtree and Related Hierarchical Data Structures*. ACM Computing Surveys, 1984.

[48] P. J. Schneider and D. Eberly. *Geometric Tools for Computer Graphics*. Elsevier Science Inc., New York, NY, USA, 2002.

[49] M. Tang, D. Manocha, J. Lin, and R. Tong. Collision-streams: Fast GPU-based Collision Detection for Deformable Models. In *Symposium on Interactive 3D Graphics and Games*, I3D '11, pages 63–70, New York, NY, USA, 2011.

[50] M. Tang, D. Manocha, and R. Tong. Multi-core Collision Detection Between Deformable Models. In *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, SPM '09, pages 355–360, New York, NY, USA, 2009.

[51] M. Teschner, B. Heidelberger, M. Mueller, D. Pomeranets, and M. Gross. Optimized Spatial Hashing for Collision Detection of Deformable Objects. In *Proc. of Vision, Modeling, Visualization VMV'03*, pages 47–54, 2003.

[52] O. Tropp, A. Tal, and I. Shimshoni. A fast triangle to triangle intersection test for collision detection. *Journal of Visualization and Computer Animation*, 17:527–535, 2006.

[53] H. Weghorst, G. Hooper, and D. Greenberg. Improved Computational Methods for Ray Tracing. *ACM Transactions on Graphics*, pages 52–69, 1984.

[54] R. Weller, J. Klein, and G. Zachmann. A Model for the Expected Running Time of Collision Detection using AABB Trees. In R. Hubbold and M. Lin, editors, *Eurographics Symposium on Virtual Environments (EGVE)*, 2006.

[55] N. Wolpert. *An Exact and Efficient Approach for Computing a Cell in an Arrangement of Quadrics*. PhD thesis, Universität des Saarlandes, 2002.

[56] L. Zhang and D. Manocha. An Efficient Retraction-based RRT Planner. In *IEEE Int. Conf. on Robotics and Automation*, pages 3743–3750, 2008.