

Boolean Operations on 3D Selective Nef Complexes: Data Structure, Algorithms, Optimized Implementation, Experiments, and Applications

Dissertation
zur Erlangung des Grades
des Doktors der Ingenieurwissenschaften (Dr.-Ing.)
der Naturwissenschaftlich-Technischen Fakultäten
der Universität des Saarlandes

von

Peter Hachenberger

Saarbrücken 2006

Datum des Kolloquiums: 1. Dezember 2006

Dekan der Naturwissenschaftlich-Technischen Fakultät I:
Prof. Dr. Thorsten Herfet

Vorsitzender des Prüfungsausschusses:
Prof. Dr.-Ing. Gerhard Weikum

Gutachter:
Dr. Lutz Kettner
Prof. Dr. Kurt Mehlhorn

Promovierter akademischer Mitarbeiter:
Dr. Rene Beier

Danksagung

An dieser Stelle möchte ich all denjenigen danken, die zum Entstehen dieser Arbeit beigetragen haben. An erster Stelle möchte ich mich bei meinem Betreuer Lutz Kettner bedanken für die Unterstützung, die konstruktive Zusammenarbeit und für alles was ich von ihm gelernt habe.

Ich danke Prof. Kurt Mehlhorn für die produktive und herzliche Atmosphäre in seiner Arbeitsgruppe am Max-Planck-Institut für Informatik. Ich habe mich am Max-Planck-Institut immer sehr wohl gefühlt und werde gerne Gelegenheiten nutzen wieder vorbeizuschauen.

Weiterhin danke ich denjenigen, die mir geholfen haben meinen Weg zurück zur universitären Laufbahn zu finden. Hier sind vor allem mein Vater Hans-Joachim Hachenberger und mein ehemaliger Arbeitskollege Frank Buschmann zu nennen.

Ein Dank gilt auch Daniel Bobbert, Andreas Meyer und Joachim Reichel, den fleissigen Korrekturlesern dieser Doktorarbeit, sowie Alantha Newman, Liz und Seth Pettie, die mir bei Fragen zur englischen Sprache geholfen haben.

Zu guter letzt danke ich allen die mir in den letzten dreieinhalb Jahren mit Rat und Tat zur Seite standen. Dies sind unter anderem Arno Eigenwillig, Stefan Funke, Joachim Giesen, Martin Kutz, Uli Meyer, Ralph Osbild und Joachim Ziegler.

Kurzzusammenfassung

Nef-Polyeder sind d -dimensionale Punktmengen, die durch eine endliche Anzahl boolescher Operationen über Halbräumen generiert werden. Sie sind abgeschlossen hinsichtlich boolescher und topologischer Operationen. Als Konsequenz daraus können sie nicht-mannigfaltige Situationen, offene und geschlossene Mengen und gemischt-dimensionale Komplexe darstellen. Die Allgemeinheit von Nef-Komplexen ist unentbehrlich für einige Anwendungen.

In dieser Doktorarbeit stellen wir eine neue Datenstruktur vor, die eine Randdarstellung von dreidimensionalen Nef-polyedern und Algorithmen für boolesche Operationen realisiert. Wir benutzen exakte Arithmetik um die bekannten Probleme mit Gleitkommaarithmetik und Degeneriertheiten zu vermeiden. Außerdem präsentieren wir wichtige Optimierungen der Algorithmen und bewerten die optimierte Implementierung an Hand umfassender Experimente. Weitere Experimente belegen die theoretische Laufzeitanalyse und vergleichen unsere Implementation mit dem kommerziellen CAD kernel ACIS. ACIS ist meistens bis zu sechs mal schneller, aber es gibt auch Beispiele bei denen ACIS scheitert.

Nef-Polyeder können bei einer Vielzahl von Anwendungen eingesetzt werden. Wir präsentieren einfache Implementationen zweier Anwendungen – von der visuellen Hülle und von der Minkowski-Summe zwei abgeschlossener Nef-Polyeder.

Abstract

Nef polyhedra in d -dimensional space are the closure of half-spaces under boolean set operations. Consequently, they can represent non-manifold situations, open and closed sets, mixed-dimensional complexes, and they are closed under all boolean and topological operations, such as complement and boundary. The generality of Nef complexes is essential for some applications.

In this thesis, we present a new data structure for the boundary representation of three-dimensional Nef polyhedra and efficient algorithms for boolean operations. We use exact arithmetic to avoid well known problems with floating-point arithmetic and handle all degeneracies. Furthermore, we present important optimizations for the algorithms, and evaluate this optimized implementation with extensive experiments. The experiments supplement the theoretical runtime analysis

and illustrate the effectiveness of our optimizations. We compare our implementation with the ACIS CAD kernel. ACIS is mostly faster, by a factor up to six. There are examples on which ACIS fails.

Nef polyhedra can be used in many a variety of applications. We present simple implementations of the visual hull, and of the Minkowski sum of two closed Nef polyhedra.

Zusammenfassung

Nef-Polyeder sind d -dimensionale Punktmenge, die durch eine endliche Anzahl boolescher Operationen über Halbräumen generiert werden. Sie sind abgeschlossen hinsichtlich boolescher und topologischer Operationen. Infolgedessen können sie nicht-mannigfaltige Situationen, offene und geschlossene Mengen und gemischt-dimensionale Komplexe darstellen. Nef-Polyeder wurden zuerst von W. Nef in seinem wegweisenden Buch über Polyeder von 1978 eingeführt. Die Allgemeinheit von Nef-Komplexen ist unentbehrlich für einige Anwendungen.

Unsere Implementation von dreidimensionalen Nef-Polyedern wurde im Dezember 2004 als Open Source als Teil der Computational Geometry Algorithm Library (CGAL) Release 3.1 herausgegeben und stößt seitdem auf großes Interesse. Unser wichtigstes Herausstellungsmerkmal ist die Verwendung exakter Arithmetik, mit deren Hilfe wir robuste Operationen und die Behandlung aller Degeneriertheiten realisieren konnten. Wir unterstützen die Konstruktion mannigfaltiger Körper gegeben im OFF Dateiformat, boolesche Operationen (Vereinigung, Schnitt, Komplement, Differenz, symmetrische Differenz), topologische Operationen (Innenraum, Rand, Abschluss, Regularisierung), starre affine Transformationen und Rotationen durch rationale Rotationsmatrizen.

Nef-Polyeder mit beschränktem Rand können eindeutig durch eine Repräsentation der lokalen Umgebungen ihrer Knoten dargestellt werden. Wir nutzen diese Eigenschaft, indem wir die lokale Umgebung eines Knotens durch ein auf der Kugeloberfläche eingebettetes zweidimensionales Nef-Polyeder repräsentieren. Die Darstellung eines dreidimensionalen Nef-Polyeders allein durch die Knoten und ihrer lokalen Umgebungen ist ausreichend, aber weder bequem noch effizient zu handhaben. Aus diesem Grund berechnen wir zusätzlich die folgenden Inzidenzen: Kanten, Facettenzyklen, die Verschachtelung der Facettenzyklen, Zusammenhangskomponenten und die Verschachtelung der Zusammenhangskomponenten. Alle Knoten, Kanten, Facetten und Volumen tragen eine Mengenzugehörigkeits-Markierung. Damit unterscheiden wir zur Punktmenge gehörige Objekte von rein begrenzenden Objekten. Durch einen Reduktionsmechanismus erweitern wir unsere Repräsentation auf allgemeine dreidimensionale Nef-Polyeder. Wir schneiden ins Unendliche laufende Kanten und Facetten an einem hinreichend großen, umschließenden, achsenparallelen Würfel ab. Dadurch entsteht ein Nef-Polyeder mit beschränktem Rand, welches wir wie oben beschrieben darstellen.

Um die wichtigsten Teilroutinen unserer binären Operationen über Nef-Polyedern zu beschleunigen, benutzen wir heuristische Suchdatenstrukturen. Mit

der Hilfe eines kd-Baumes berechnen wir den ersten Schnittpunkt eines Strahls mit dem Rand eines Nef-Polyeders und die Lage eines Punktes im Verhältnis zu einem Nef-Polyeder. Weiterhin schneiden wir die minimal umschließenden Boxen von Kanten und Facetten zweier Polyeder, um schnell eine kleine Obermenge aller Schnitte zwischen Kanten und Facetten der zwei Polyeder zu identifizieren.

Wir haben unsere Implementation an Hand umfassender Experimente getestet. Die Experimente untersuchen das Laufzeitverhalten unserer binären Operationen in speziellen Situationen. Dabei interessieren uns sowohl generische Situationen, wie z.B. die Subtraktion eines kleinen und simplen Objekts von einem großen und komplexen Objekt, als auch Situationen, die eine besonders schlechte Laufzeit der wichtigsten Teilschritte unseres Algorithmus bewirken. Außerdem bestätigen wir an Hand dieser Experimente auch die theoretisch berechnete Komplexität der binären Operationen und ihrer wichtigsten Teilschritte. Eine weitere Gruppe von Experimenten belegt den Nutzen von wichtigen Optimierungen. Die letzte Gruppe unserer durchgeführten Experimente vergleicht unsere Implementation mit dem kommerziellen CAD Kernel ACIS R13. Die beiden Systeme sind recht unterschiedlich, da beide über Fähigkeiten verfügen, die dem anderen fehlen. Während ACIS zusätzlich mit gekrümmten Objekten umgehen kann, verwendet es andererseits die auf dem Markt übliche Gleitkommaarithmetik und ist somit nicht robust. Die Ergebnisse zeigen, dass ACIS R13 im Allgemeinen bis zu sieben mal schneller ist. In manchen Szenarien ist ACIS jedoch langsamer oder schlägt sogar fehl.

Nef-Polyeder können bei einer Vielzahl von Anwendungen eingesetzt werden. Wir präsentieren einfache Implementierungen zweier Anwendungen – von der visuellen Hülle und von der Minkowski-Summe zwei abgeschlossener Nef-Polyeder. Die Implementierung der visuellen Hülle konnte schnell und ohne große Schwierigkeiten durchgeführt werden. Sie ist robust und umfassend, aber noch zu langsam für viele Anwendungen – vor allem für Echtzeitanwendungen. Minkowski-Summen können zur Bewegungsplanung von Robotern eingesetzt werden, die sich ausschließlich durch Translation fortbewegen. Eine exakte Berechnung der Minkowski-Summe ist von großem Interesse, wenn der Roboter sich durch eine enge Passage bewegen soll, also durch eine Passage, die genauso breit ist wie er selbst. Unsere Implementation ist die erste exakte Implementation der Minkowski-Summe auf nicht-konvexen dreidimensionalen Polyedern. Andererseits können wir noch nicht die Minkowski-Summe nicht-geschlossener Polyeder berechnen, und somit noch nicht mit engen Passagen umgehen.

Contents

1	Introduction	1
1.1	Motivation	2
1.2	Previous Work	6
1.3	Relation to Preceding Work	8
1.4	CGAL — A Generic Software Library	9
1.5	Outline	12
2	Nef’s Theory of Polyhedra	15
3	Representation schemes	19
3.1	Sphere Maps	20
3.2	Selective Nef Complex	21
3.3	Infimaximal Box - A Reduction to Finitely Bounded Polyhedra	23
4	Boolean and Topological Operations	29
4.1	Map Overlay on the Sphere	30
4.1.1	A Segment Sweep Algorithm.	31
4.1.2	A Generic Framework.	32
4.1.3	Overlay of Two Planar Nef polyhedra.	35
4.1.4	Segment Sweep on the Sphere.	39
4.2	Selection	42
4.3	Simplification on the Sphere	42
4.4	Candidate Sphere Maps	45

4.5	Simplifying the Selective Nef Complex	45
4.6	Synthesizing the SNC	46
4.6.1	Pairing up Halfedges.	47
4.6.2	Creation of Facet Cycles.	49
4.6.3	Creation of Facets.	51
4.6.4	Creation of Volumes.	51
4.7	Unary Operations	54
5	Search Data Structures	55
5.1	Kd-tree	55
5.2	Fast Box Intersection	58
5.2.1	Segment Trees.	59
5.2.2	Streaming.	61
5.2.3	Scanning.	61
5.2.4	The Hybrid Algorithm.	63
6	Additional Functionality	65
6.1	Constructors, Input and Output	65
6.2	Transformation	68
6.3	Visualization	69
7	Complexity	71
7.1	Kd-Tree	72
7.2	Box-Intersection Algorithm	73
7.3	Total Complexity	73
8	Software Design	77

9	Algorithm Engineering	83
9.1	Optimizations	84
9.1.1	Ray shooting and Point Location	85
9.1.2	Intersection	86
9.1.3	Half-sphere Sweep	88
9.1.4	Plane Sweep	91
9.2	General Runtime Behavior	92
9.2.1	Balanced Binary Operations	92
9.2.2	Binary Operation with Quadratic Result	93
9.2.3	A Complex Object Minus a Simple Object	95
9.3	Runtime Behavior in Complex Situations	96
9.3.1	Complex Facet	97
9.3.2	Complex Sphere Map	98
9.3.3	Kd-tree Construction and Queries	99
9.4	Comparison with ACIS	103
9.4.1	Balanced Binary Operations	103
9.4.2	Floating-Point versus Exact Arithmetic	104
9.4.3	A Complex Object Minus a Simple Object	106
9.5	Growth of Coordinate Representation	107
9.6	Résumé	111
9.6.1	Further Improvement on Point Location and Ray Shooting	111
9.6.2	Modification Operations	113
9.6.3	Exact Geometric Computing	113
10	Applications for Nef polyhedra	117
10.1	Visual Hull	117
10.2	Minkowski Sum of Two Nef polyhedra	119
10.2.1	The Minkowski sum of convex polyhedra	121
10.2.2	The Vertical Decomposition of a 3D Nef polyhedron.	123
10.2.3	Uniting a Set of 3D Polyhedra	130
10.2.4	Limitations and Future Work	132

11 Conclusion	133
11.1 Results	133
11.2 Future work	134
Bibliography	134

Chapter 1

Introduction

In this dissertation we consider a data structure for Nef polyhedra in three-dimensional space and algorithms for Boolean operations on them. Nef polyhedra were introduced by Walter Nef in his seminal book on polyhedra from 1978 [Nef78]. They are defined as a finite number of set intersection and set complement operations on half-spaces. In consequence, they are closed under Boolean and topological operations. Furthermore, they can represent non-manifold situations, open and closed boundaries, and mixed-dimensional features. As an example, the intersection of the six half-spaces defined by $x \leq 0.5$, $x \geq -0.5$, $y \leq 0.5$, $y \geq -0.5$, $z \leq 0.5$, and $z \geq -0.5$ forms the unit cube. Figure 1.1 shows a more complex example with non-manifold situations, selected and unselected boundary parts, and lower dimensional features.

Polyhedron modelers are useful tools for solving various problems in solid modeling, computer graphics, and computational geometry. The solid modeling community spent much effort on theoretical foundations and implementations of polyhedron modelers. Nef's approach to modeling polyhedra is mathematically well-founded and clean. No other approach is as general and comprehensive as Nef's. Still, nobody provided an implementation of his concept, yet. Most of the research on polyhedra was done more than twenty years ago. At that time, it probably was an obvious choice to implement models of a lesser generality and therefore a lesser complexity than Nef polyhedra. Nowadays, polyhedron modelers are still limited in their generality and the research is more interested in methods for modeling higher-order surfaces. Nevertheless, we argue in Section 1.1 that there is a need for a more powerful polyhedron implementation.

In this thesis, we describe data structures capable of modeling 3D Nef polyhedra completely. Also we provide algorithms for performing Boolean and topological operations on Nef polyhedra. What is more, all our algorithms and data

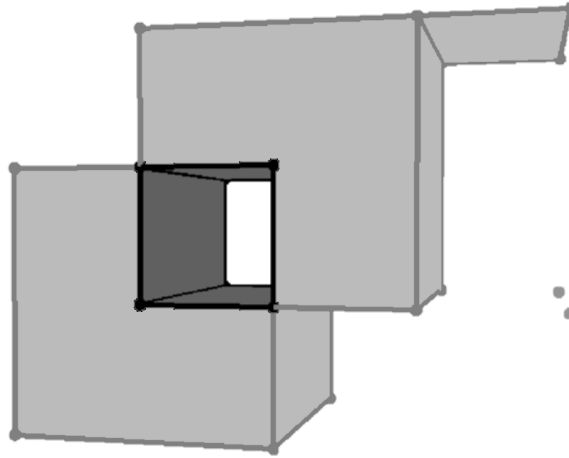


Figure 1.1: A Nef polyhedron with non-manifold edges, a dangling facet, two isolated vertices, and an unselected boundary in the tunnel.

structures work with exact arithmetic instead of floating-point arithmetic. Exact arithmetic is often regarded as slow. We compare our implementation with the ACIS CAD kernel and demonstrate the power and cost of exact arithmetic in near-degenerate situations. As far as we know our implementation is the only polyhedron modeler on the market that uses exact arithmetic.

In December 2004, our implementation was released as Open Source software in the Computation Geometry Algorithm Library (CGAL) release 3.1. It supports the construction of Nef polyhedra from half-spaces and manifold solids, Boolean operations (union, intersection, complement, difference, symmetric difference), topological operations (interior, closure, boundary), rotation by rational rotation matrices (arbitrary rotation angles are approximated up to a specified tolerance [CDR92]), translation and scaling.

In order to demonstrate the opportunities of our polyhedron modeler, we also examine and solve two applications: the computation of the visual hull from a set of two-dimensional shapes, and the Minkowski sum of two Nef polyhedra.

1.1 Motivation

Most of the professional polyhedron modelers serve as a kernel for computer-aided design applications. They share two problems: completeness and exactness. Our implementation of Nef polyhedra deals with these problems. Nef polyhedra are

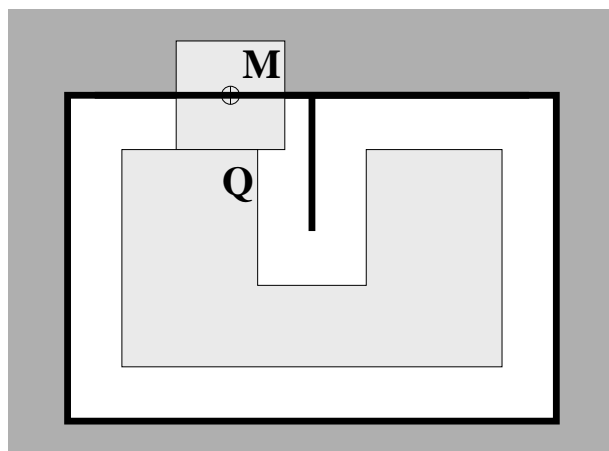


Figure 1.2: The width of the cutter M is equal to the width of the cavity in Q . The boundary of the region of legal placements is shown in bold. It is an unbounded polygon with a dangling edge.

more complete than any other polyhedron model; they can model non-manifold solids, unbounded solids, lower dimensional features, and infinite boundaries. Additionally we use exact arithmetic instead of floating-point arithmetic. These features are often regarded as unnecessary. We disagree.

Nef polyhedra are the smallest family of solids containing the half-spaces and being closed under Boolean operations. Without a doubt, a closed modeling space is desirable. We want to discuss two subsets of Nef polyhedra which also provide a closed modeling space: Regularized sets and finitely bounded Nef polyhedra. As described above, regularized sets are closed under regularized set operations, but Middleditch [Mid94] argues that we need more than regularized set operations. We need to concurrently model objects of different dimensionality, or objects with open and closed boundaries. One of his examples occurs in machine tooling. We may want to generate a polyhedron Q by a cutting tool M , as shown in Figure 1.2. When the tool is placed at a point p in the plane, all points in $p + M$ are removed. The set of legal placements for M is called the *configuration space* of M . It is defined as the set $C = \{p; (p + M) \cap Q = \emptyset\}$. The set C may contain lower dimensional features. In the context of robot motion planning a lower dimensional feature in the configuration space is referred to as a *tight passage*. In order to identify and handle tight passages, it is necessary to allow open and closed polyhedra. If M and Q are both modeled as closed sets, C cannot contain lower dimensional features, i.e., tight passages cannot be identified. Then again, if either of the two

is open, then C will be closed. See [Hal02] for the case of planar configuration spaces.

Our implementation can be used in two modes. It can either represent the full modeling space of 3D Nef polyhedra, or the user may decide to limit the modeling space to a specific subclass of 3D Nef polyhedra because of efficiency reasons. In order to describe this subclass, we introduce the notion of finitely and infinitely bounded Nef polyhedra. Usually, polyhedra are classified as bounded and unbounded, i.e., a polyhedron either has a finite or an infinite volume. We need a slightly different classification. Since our data structures are a boundary representation of Nef polyhedra, we are more concerned whether the boundary itself is bounded, and not whether the complete polyhedron is bounded. As an example, neglecting selection marks a cube and its complement have the same boundary representation, although the cube is bounded and its complement is not. Therefore, we denote a polyhedron as *finitely bounded*, if each bounding edge has a finite length, and each bounding facet covers a finite surface area. Otherwise, it is denoted as *infinitely bounded*. Note that finitely bounded polyhedra either have no boundary or a finite boundary; they can be both, bounded or unbounded. Finitely bounded Nef polyhedra are a subset of Nef polyhedra, but are also closed under Boolean and topological operations.

We do not know any other polyhedron modeler that supports infinitely bounded polyhedra. For most applications it suffices to only have finitely bounded polyhedra available. Furthermore, offering the whole modeling space of Nef polyhedra is more complicated and therefore less efficient. Still, infinitely bounded Nef polyhedra are meaningful for some applications. As an example, one of CGAL's evaluators uses infinitely bounded Nef polyhedra to create three-dimensional Voronoi diagrams. The resulting polyhedron reasonably approximates the requirements of a specific space partition needed by the evaluator. We offer both: the limited modeling space of finitely bounded Nef polyhedra for fast applications that do not necessarily need infinite boundaries, and the full modeling space as an alternative. Both modes use the same data structures and algorithms. The user decides about the modeling space by providing a proper template argument at compile time. The mechanism is explained in Section 3.3.

The lack of exactness caused by floating-point arithmetic is a well known problem. The robustness example shown in Figures 1.3 and 1.4 illustrates the problems nicely. We intersect two equal cubes, where the second is rotated by a small angle α around each coordinate axis. The first picture shows the result for α being five degrees. Neglecting minor deviations, the result looks like a cube. We can see three vertices in the upper front corner lying very close to each other. Also we can see diagonals on each side indicating the non-planarity of the sides. If α decreases

1.1. MOTIVATION

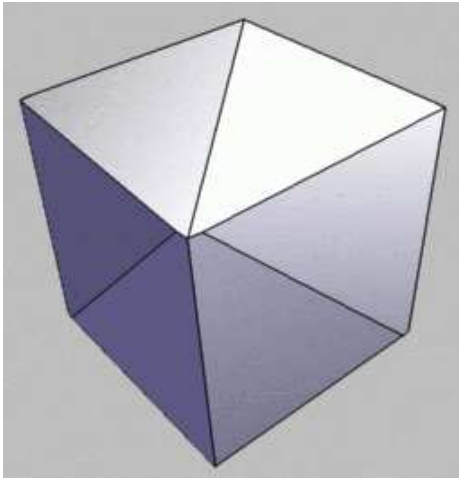


Figure 1.3: A robustness example showing the intersection of 2 cubes where one is rotated by five degrees around each coordinate axis.

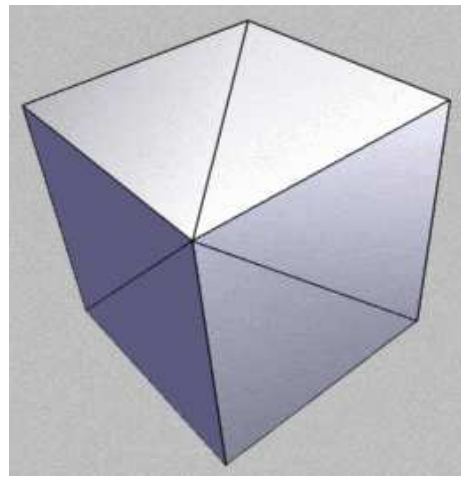


Figure 1.4: As Figure 1.3 but with 0.01 degree rotations. Vertices are not separable in the drawing, but the edges illustrate the solution.

continuously, the three vertices in the front corner move closer together and the sides become nearer to coplanar. Using floating-point arithmetic, it is not possible for small angles to distinguish the three vertices, and to decide whether the sides are coplanar or not. Exactness becomes crucial for many predicates. For example, we may want to perform point location queries. With floating-point arithmetic it is often impossible to get a correct solution if the queried point lies near to or even on a line or plane.

Yap gives an elaborate discussion about exact computation in computational geometry [Yap97]. He defines the term *exact computation* as a computation that

1. represents the underlying mathematical objects in an exact manner, and
2. in the course of computation, never makes an error in its decision.

One goal of his discussion is to “study the inherent tradeoffs between speed and precision, between fixed-precision and exact computation.” Amongst others, he names the following advantages of exact computation:

- Arithmetic robustness is a non-issue.

- Classical geometric concepts and algorithms are often formulated in exact terms. Providing exact computation preserves those concepts and algorithms. Using fixed-precision arithmetic with these concepts and algorithms can lead to major robustness problems or extensive workarounds.
- A major technique for handling degeneracies is symbolic perturbation. This method is only meaningful with exact computation.

The main disadvantage of exact computation is its lack of speed compared to fixed-precision arithmetic. Reflecting on approaches by Fortune and Van Wyk [FW93], and by Karasick, Lieber and Nackman [KLN91], Yap thinks that with careful work, exact geometric primitives should be at most ten times slower than their floating-point counterparts.

Test series have confirmed the robustness issues arising in geometric algorithms. In [KMP⁺04] the authors show that rounding errors in basic geometric predicates can lead to severe errors in geometric applications like the computation of the convex hull. Still, exact computation is unpopular in many areas. In addition, the floating-point community enjoys a huge infrastructural support that helps to manifest its predominance. Yap [Yap97] names robust algorithms for performing Boolean operations on solids as “fundamental in the field of solid modeling.” Yet, exact solid modelers are non-existent. We will show, that our exact solid modeler can compete with professional CAD kernels. In comparison with the ACIS CAD kernel, we achieve a significantly better result than the factor ten demanded by Yap.

1.2 Previous Work

Data structures for solids and algorithms for Boolean operations on geometric models are among the fundamental problems in solid modeling, computer aided design, and computational geometry [Hof89, Män88, RR, HSW01, For97]. In their seminal work, Nef and, later, Bieri and Nef [Nef78, BN88] developed the theory of Nef polyhedra. Dobrindt, Mehlhorn, and Yvinec [DMY93] consider Nef polyhedra in three-space and give an $O((n+m+s)\log(n+m))$ algorithm for intersecting a general Nef polyhedron with a convex one; here n and m are the sizes of the input polyhedra and s is the size of the output. The idea of the sphere map is introduced in their paper (under the name local graph). They do not discuss implementation details. Seel [See01a, See01b] gives a detailed study of planar Nef polyhedra. Our implementation is based on his work. We closely investigate his contribution in the following section.

1.2. PREVIOUS WORK

In the following, we shortly introduce other approaches to non-manifold geometric modeling, and identify the major differences to our approach:

Rossignac and O'Connor describe modeling by so-called *selective geometric complexes*. The underlying geometry is based on algebraic varieties. The corresponding point sets are stored in selective cellular complexes. Each cell is described by its underlying extent, and by a subset of cells of the complex that constitute its boundary. The non-manifold situations that occur are modeled via the incidence links between cells of different dimension. The incidence structure of the cellular complex is stored in a hierarchical but otherwise unordered way. No implementation details are given.

Weiler's radial-edge data structure [Wei88] and Karasick's star-edge boundary representation [Kar89] are centered around the non-manifold situation at edges. Both present ideas about how to incorporate the topological knowledge of non-manifold situations at vertices; their solutions, however, do not completely cover all incidences [GCP90]. If a vertex is incident to multiple volumes, their representation does not store data that resolves the nesting structure of their shells. The missing data must be computed from geometric information if needed. Gursoz, Choi and Prinz [GCP90] extend the ideas of Weiler and Karasick and center the design of their non-manifold modeling structure around vertices. They introduce a cellular complex that subdivides space and that models the topological neighborhood of vertices. The topology is described by a spatial subdivision of an arbitrarily small neighborhood of the vertex. Their approach gives thereby a complete description of the topological neighborhood of a vertex.

Fortune's approach [For97] centers around plane equations and uses symbolic perturbation of the planes' distances to the origin to eliminate non-manifold situations and lower-dimensional faces. Here, a two-manifold representation is sufficient. The perturbed polyhedron still contains the degeneracies, now in the form of zero-volume solids, zero-length edges, etc. Depending on the application, special post-processing of the polyhedron might be necessary, for example, to avoid meshing a zero-volume solid. Post-processing is not discussed in the paper and it is not clear how expensive it would be. The direction of the perturbation, i.e., towards or away from the origin, can be used to model open and closed sets.

We improve the structure of Gursoz et al. with respect to storage requirements and provide a more concrete description with respect to the work of Dobrindt et al. as well as a first implementation. Our structure provides maximal topological information and is centered around the local view of vertices of Nef polyhedra. We detect and handle all degenerate situations explicitly, which is a must given the generality of our modeling space. The clever structure of our algorithms helps to avoid

the combinatorial explosion of special case handling. We use exact arithmetic to achieve correctness and robustness.

The fact that we can quite naturally handle all degeneracies, including non-manifold structures, as well as unbounded objects and always produce the correct mathematical result differentiates us from other approaches. Previous approaches using exact arithmetic [AR94, BR96, BMP94, For97, KKM97] work in a less general modeling space, some unable to handle non-manifold objects and none able to handle unbounded objects.

1.3 Relation to Preceding Work

Following [BN88], Nef polyhedra can be represented by modeling the local neighborhood of its vertices. Later, Dobrindt, Mehlhorn and Yvinec [DMY93] proposed to realize the local neighborhood of a vertex by a symbolic intersection of the polyhedron with an ε -sphere around the vertex. The resulting surface is a planar Nef polyhedron embedded on the sphere. We follow this approach with our implementation.

Our work on Nef polyhedra in three-dimensional space is based upon the work of Michael Seel. First, Michael Seel implemented planar Nef polyhedra [See01a, See01b]. His implementation was released as part of CGAL 2.3 in August 2001. Then he started to implement three-dimensional Nef polyhedra, too. As an intermediate step, he adopted his implementation of planar Nef polyhedra for spherical surfaces. In the context of three-dimensional Nef polyhedra, this code can be reused for the representation of a so called *sphere map*, i.e., an ε -sphere intersecting the polyhedron around the vertex, together with the vertex in its center, and the label of the vertex. With most of the functionality of the spherical Nef polyhedra completed, and a good deal of the three-dimensional version realized, Michael Seel quit his academic work and went to industry. Afterwards, Miguel Granados, who visited the Max-Planck-Institut for half a year, continued Michael Seel's work during his stay.

I took over the package in August 2002. At this time, the basic functionality of two-dimensional Nef polyhedra embedded on the sphere and three-dimensional Nef polyhedra was complete, but had some major bugs. Also, ray shooting, point location and intersection finding were implemented by trivial brute-force solutions. As a first step, I created a running version by correcting the main bugs. This first running version worked with finitely bounded polyhedra only. We applied the technique of infimal boxes to remove this restriction. In contrast to Seel's implementation of planar Nef polyhedra, it is possible to choose between the full

modeling space or to limit it to finitely bounded Nef polyhedra at compile time. For infinitely bounded Nef polyhedra, we use an infimal box, which implies using linear polynomials as coordinates. Exchanging the linear polynomials with constants limits the modeling space to finitely bounded polyhedra, but improves the performance significantly.

Then we completed the functionality by adding transformations for arbitrary Nef polyhedra. Rotations of infinitely bounded polyhedra are especially complicated for our approach. We discuss this problems extensively in Section 6.2.

Having a complete functionality, we accomplished two further goals. First, we turned our implementation into a proper CGAL package. This includes a comprehensive test suite and a documentation. Furthermore, we refactored large parts of the code. Michael Seel did not reuse the code of the Nef polyhedra embedded on the sphere, but duplicated and adapted it. We consolidated the code for better maintainability. As a consequence it is now possible to obtain a sphere map as a two-dimensional Nef polyhedron embedded on the sphere, i.e., each algorithm created for Nef polyhedra embedded on the sphere also works on sphere maps without special adaptation.

As a second step, we optimized our implementation for efficiency. On a second visit Miguel Granados implemented a kd-tree to speed up ray shooting and point location. Another student, Andreas Meyer, implemented fast box intersection as described in [ZE02]. Andreas Meyer's work was supervised by Lutz Kettner, Miguel Granados' work was supervised by Lutz Kettner and me. Here, our test suite helped us to identify and remove errors early. Additionally, I performed a great number of experiments and benchmarks to find and remove the main bottlenecks. As a result we can compete with professional software for Computer-Aided Design.

With the packages for spherical Nef polyhedra and three-dimensional Nef polyhedra completed, we implemented applications based upon them. We have already realized an algorithm for the computation of visual hulls, and intend to solve Minkowski sums on arbitrary Nef polyhedra. As a first step of the Minkowski sum, we implemented the Minkowski sum on convex polyhedra.

1.4 CGAL — A Generic Software Library

Nef polyhedra in three-dimensional space were released as a package of the Computational Geometry Algorithm Library (CGAL) [CGA, FGK⁺00] in December 2004. CGAL is a collaborative effort of several sites in Europe and Israel. The goal

is to make the most important of the solutions and methods developed in computational geometry available to users in industry and academia in a C++ software library.

CGAL is designed as a C++ library. Following the generic programming paradigm, CGAL is particularly efficient and flexible. Here is one definition of the generic programming paradigm [JLM00]:

Generic programming is a sub-discipline of computer science that deals with finding abstract representations of efficient algorithms, data structures, and other software concepts, and with their systematic organization. The goal of generic programming is to express algorithms and data structures in a broadly adaptable, inter-operable form that allows their direct use in software construction. Key ideas include:

- Expressing algorithms with minimal assumptions about data abstractions, and vice versa, thus making them as inter-operable as possible.
- Lifting of a concrete algorithm to as general a level as possible without losing efficiency; i.e., the most abstract form such that when specialized back to the concrete case the result is just as efficient as the original algorithm.
- When the result of lifting is not general enough to cover all uses of an algorithm, additionally providing a more general form, but ensuring that the most efficient specialized form is automatically chosen when applicable.
- Providing more than one generic algorithm for the same purpose and at the same level of abstraction, when none dominates the others in efficiency for all inputs. This introduces the necessity to provide sufficiently precise characterizations of the domain for which each algorithm is the most efficient.

The generic programming paradigm is realized in C++ by *class templates* and *function templates*. Templates are incompletely specified components, i.e., some types are only identified by formal placeholders, the *template arguments*. The compiler generates a separate translation of the code for each instantiation of a template argument. The requirements that are needed to obtain a correct instantiation of a template argument are not defined explicitly. Syntactical requirements are defined by calls on instantiations of the templated type. Surely the assigned type must also

meet the semantical requirements as intended - and hopefully well specified - by the developer. Note that an actual type must only fulfill the requirements of the actually called functions. This enables the design of class templates with optional functionality.

The main advantage of generic programming lies in its flexibility. In contrast to object-oriented programming, polymorphism is available without the restrictions of inheritance. Not relying on inheritance implies benefits in efficiency, because inheritance requires extra memory and call indirections for virtual functions.

In the overall design of CGAL two major layers can be identified, the layer of algorithms and data structures and the geometric-kernel layer. The concept of a geometric kernel comprises all basic geometric data types and algorithms abstracted from number types and the choice whether to use Cartesian or homogeneous coordinates. Thereby it bundles several concepts into one large unit. As a result, the geometric kernel concept can be used with any CGAL algorithm.

CGAL provides further concept bundles for specific problems. This concept is denoted as a *traits class*. The notion of traits classes has evolved in the recent years in the domain of software libraries. Originally, the concept of traits classes [Mye95] was developed to associate related types, constants, and functions to built-in types. This is achieved by specialization of a general traits template.

For the access of geometry and incidence objects, CGAL uses *iterators*, which are a generalization of pointers. Iterators decouple the storage of data from its usage. Programmers may use different data structures, but use them in a uniform manner without even knowing the data structure. An iterator is a concept that specifies a set of requirements. A type is an iterator if it satisfies those requirements. In this sense, a pointer to an element of an array is an iterator.

Iterators are often used to define *iterator ranges*. Two iterators *first* and *last* define a valid iterator range $[first, last)$, if both point to an element in the same data structure, and the elements of the data structure are ordered in such a way, that *last* is a successor element of *first*. Then, the iterator range refers to the iterators $first, first + 1, \dots, last - 1$, as well as to the elements $*first, *(first + 1), \dots, *(last - 1)$. The iterator *last* is not part of the range, but indicates the end of the range. This way, even empty ranges can uniformly be represented.

The standard template library (STL) defines five iterator concepts: input iterator, output iterator, forward iterator, bidirectional iterator, and random access iterator. Each of these concepts defines a subset of the pointer functionality typically needed for the access of a data structure. An input iterator allows to read from a data structure in consecutive order, an output iterator overwrites the elements in consecutive order, a forward iterator has read and write access to a data structure,

but can only process its elements in consecutive order, while a bidirectional iterator can additionally process them in reverse order. Finally, a random access iterator has the full flexibility of a pointer, which also includes pointer arithmetic.

CGAL defines two additional iterator concepts. A handle, also known as trivial iterator, does not support an iteration over the data structure. It only points to some element that can be accessed for reading and writing. The circulator concept extends the iterator concept for iteration on circular data structures. Analogous to iterators, CGAL defines multiple circulator concepts: forward circulator, bidirectional circulator, and random access circulator.

1.5 Outline

The organization of the remaining chapters is as follows:

- In Chapter 2 we repeat Nef's definitions of polyhedra, their faces, and their incidence structure.
- In Chapter 3 we introduce the three basic concepts that we use for the representation of 3D Nef polyhedra. Sphere maps model the local neighborhood of vertices. With a sphere map for each vertex, we can model a finitely bounded Nef polyhedra. The selective Nef complex adds further incidences for a more convenient and faster usage. With the infimal box we reduce infinitely bounded Nef polyhedra to finitely bounded Nef polyhedra.
- In Chapter 4 we put down Boolean and topological operations on three-dimensional Nef polyhedra to Boolean and topological operations on two-dimensional Nef polyhedra embedded on the sphere. Also we describe the construction of the selective Nef complex.
- During Boolean operations we use fast box intersection in order to find all edge–edge and edge–facet intersections, and we use a kd-tree for efficient ray shooting and point location. We describe those two search structures in Chapter 5.
- Chapter 6 shortly introduces the remaining functionality provided by our implementation.
- Chapter 7 discusses the worst-case and expected average-case runtime of our algorithms.

1.5. OUTLINE

- In Chapter 8 we describe the design of our two software packages `Nef_polyhedron_3` and `Nef_polyhedron_S2`.
- In Chapter 9 we investigate the performance of our implementation and the means used to guarantee its efficiency. We perform several test series in order to examine the runtime behavior of binary operations and their major subroutines. With the results of the experiments, we motivate several optimizations and confirm their benefit. Also, we compare our implementation to the professional CAD-kernel ACIS.
- In Chapter 10 we present basic implementations of two applications: the computation of the visual hull from a set of two-dimensional shapes, and the Minkowski sum of two Nef polyhedra.
- Chapter 11 summarizes the main results and discusses opportunities for future research based upon 3D Nef polyhedra.

Chapter 2

Nef's Theory of Polyhedra

Partitions of three-space into cells are a common theme of solid modeling and computational geometry. The two major representation schemes were developed in the solid modeling community, which is the older of the two communities. Those schemes are: *constructive solid geometry* (CSG) and *boundary representations* (B-rep). Both have inherent strengths and weaknesses, see [Hof89] for a detailed discussion.

In CSG, a solid is represented as a set-theoretic Boolean combination of primitive solid objects, such as blocks, prisms, cylinders, or tori. The Boolean operations are not computed explicitly. Instead, objects are represented implicitly with a tree structure; leaves represent primitive objects and interior nodes represent Boolean operations or rigid motions, e.g., translation and rotation. Algorithms on such a CSG-tree first identify properties of the primitive objects and propagate the results using the tree structure.

A B-rep describes a solid by the incidence structure and the geometric properties of its boundary. Surfaces are oriented to decide between the interior and exterior of a solid.

The class of representable objects in a CSG is usually limited by the choice of the primitive solids. A B-rep is usually limited by the choice for the geometry of the supporting curves for edges and the supporting surfaces for surface patches, and, in addition, the connectivity structure that is allowed. In particular, a B-rep is not always closed under Boolean set operations. As an example, the class of orientable two-manifold objects is a popular and well understood class of surfaces commonly used for B-reps. They can be represented and manipulated efficiently, the data structures are compact in storage size, and many algorithms are simple. On the other hand, this object class is not closed under Boolean set operations. The

object in Figure 1.1 can be generated by applying Boolean set operations on several cubes. Cubes are orientable two-manifold object, but the object in Figure 1.1 is not a two-manifold. The vertices bounding the tunnel, or the edge connecting the “roof” with the cube are non-manifold situations.

Because manifolds are not closed under Boolean operations, Requicha proposed *regularized set operations* [KM76, Req80]. A set is *regular*, if it equals the closure of its interior. A regularized set operation is defined as the standard set operation followed by a regularization of the result, i.e., after the standard set operation, the closure and the interior operation are consecutively applied to the result. Regularized sets are closed under regularized set operations. As they exclude lower dimensional features and the boundary belongs to the point set, they are considered to reflect the nature of physical solids closely.

Nef took a different approach. Instead of finding new kinds of operations to establish a closed modeling space, he adapted the definition of polyhedra. His seminal book from 1978 provided clean mathematical definitions for d -dimensional polyhedra [Nef78]. Some basic notions of polyhedra, like the notion of a face, where properly defined for the first time. The new theory includes some very nice properties. For instance, each face of a polyhedron is a polyhedron itself.

We repeat a few definitions and facts about Nef polyhedra [Nef78] that we need for our data structures and algorithms. The definitions are presented for arbitrary dimensions, but in the sequel we restrict ourselves to three dimensions.

Definition 2.1 (Nef polyhedron). A *Nef-polyhedron* in dimension d is a point set $P \subseteq \mathbb{R}^d$ generated from a finite number of open half-spaces by set complement and set intersection operations.

Set union (\cup), difference (\setminus), and symmetric difference (Δ) can be reduced to set intersection (\cap) and set complement (!) as follows:

$$\begin{aligned} P_1 \cup P_2 &= !(P_1 \cap !P_2) \\ P_1 \setminus P_2 &= P_1 \cap !P_2 \\ P_1 \Delta P_2 &= (P_1 \setminus P_2) \cup (P_2 \setminus P_1) = !(P_1 \cap !P_2) \cap !(P_2 \cap !P_1) \end{aligned}$$

Set complement changes between open and closed half-spaces, thus the topological operations *boundary*, *interior*, *exterior*, *closure* and *regularization* are also in the modeling space of Nef polyhedra. In what follows, we refer to Nef polyhedra whenever we say polyhedra.

A face of a polyhedron is defined as an equivalence class of *local pyramids* that are a characterization of the local space around a point.

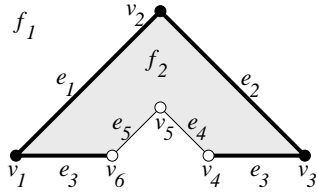


Figure 2.1: Planar example of a Nef polyhedron. The shaded region, bold edges and black nodes are part of the polyhedron, thin edges and white nodes are not.

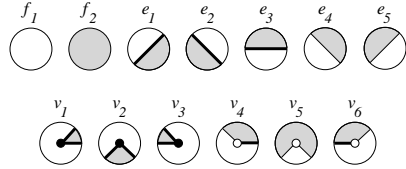


Figure 2.2: Sketches of the local pyramids of the planar Nef polyhedron example. The local pyramids are indicated as shaded in the relative neighborhood in a small disc.

Definition 2.2 (Local pyramid). A point set $K \subseteq \mathbb{R}^d$ is called a *cone with apex 0*, if $K = \mathbb{R}^+ K$ (i.e., $\forall p \in K, \forall \lambda > 0 : \lambda p \in K$) and it is called a *cone with apex x* , $x \in \mathbb{R}^d$, if $K = x + \mathbb{R}^+(K - x)$. A cone K is called a *pyramid* if K is a polyhedron.

Now let $P \in \mathbb{R}^d$ be a polyhedron and $x \in \mathbb{R}^d$. There is a neighborhood $U_0(x)$ of x such that the pyramid $Q := x + \mathbb{R}^+(P \cap U(x) - x)$ is the same for all neighborhoods $U(x) \subseteq U_0(x)$. Q is called the *local pyramid* of P in x and denoted by $\text{Pyr}_P(x)$.

Definition 2.3 (Face). Let $P \in \mathbb{R}^d$ be a polyhedron and $x, y \in \mathbb{R}^d$ be two points. We define an equivalence relation $x \sim y$ iff $\text{Pyr}_P(x) = \text{Pyr}_P(y)$. The equivalence classes of \sim are the *faces* of P . The dimension of a face s is the dimension of its affine hull, $\dim s := \dim \text{aff } s$.

In other words, a *face s* of P is a maximal non-empty subset of \mathbb{R}^d such that all of its points have the same local pyramid Q denoted by $\text{Pyr}_P(s)$. This definition of a face partitions \mathbb{R}^d into faces of different dimension. A face s is either a subset of P , or disjoint from P . We use this later in our data structure and store a selection mark in each face indicating its set membership.

Example 2.4. We illustrate the definitions with an example in the plane. Given the closed half-spaces

$$h_1 : y \geq 0, \quad h_2 : x - y \geq 0, \quad h_3 : x + y \leq 3, \quad h_4 : x - y \geq 1, \quad h_5 : x + y \leq 2,$$

we define our polyhedron $P := (h_1 \cap h_2 \cap h_3) - (h_4 \cap h_5)$. Figure 2.1 illustrates the polyhedron with its partially closed and partially open boundary, i.e., vertex v_4, v_5, v_6 , and edges e_4 and e_5 are not part of P . The local pyramids for the faces

are $\text{Pyr}_P(f_1) = \emptyset$ and $\text{Pyr}_P(f_2) = \mathbb{R}^2$. Examples for the local pyramids of edges are the closed half-space h_2 for the edge e_1 , $\text{Pyr}_P(e_1) = h_2$, and the open half-space that is the complement of h_4 for the edge e_5 , $\text{Pyr}_P(e_5) = \{(x, y) \mid x - y < 1\}$. The edge e_3 consists actually of two disconnected parts, both with the same local pyramid $\text{Pyr}_P(e_3) = h_1$. However, as explained later, in our data structure, we will represent the two connected components of the edge e_3 separately. Figure 2.2 illustrates all local pyramids for this example.

Faces do not have to be connected. There are only two full-dimensional faces possible, one whose local pyramid is the space \mathbb{R}^d itself and the other with the empty set as a local pyramid. All lower-dimensional faces form the *boundary* of the polyhedron. As usual, we call zero-dimensional faces *vertices* and one-dimensional faces *edges*. In the case of polyhedra in space we call two-dimensional faces *facets* and the full-dimensional faces *volumes*. Faces are *relative open* sets, e.g., an edge does not contain its end-vertices. The incidence relationship of faces in a Nef polyhedron is defined as follows:

Definition 2.5 (Incidence relation). In a polyhedron P , a face s is *incident* to a face t iff $s \subset \text{cl}st$. This defines a partial ordering \prec such that $s \prec t$ iff s is incident to t .

Bieri and Nef also proposed several data structures for storing Nef polyhedra in arbitrary dimensions. In the *Würzburg structure* [BN88], named after the workshop location where it was first presented, all faces are stored in the form of their local pyramids. The Würzburg structure is complete, but not convenient, since it misses the explicit representation of incidences between faces. They must be computed if needed. In the *extended Würzburg structure* these incidences are additionally stored. On the other hand, the Würzburg structure stores redundant information. It suffices to store only the local pyramids of the minimal elements in the incidence relation \prec , which is realized by the *reduced Würzburg structure* [Bie96]. For bounded polyhedra all minimal elements are vertices.

Chapter 3

Representation schemes

Either Würzburg structure supports Boolean operations on Nef polyhedra, neither of them does so in an efficient way. The reason is that Würzburg structures do not store enough information about the structure of the faces. For example, the extended Würzburg structure, which provides more information than the other two, records the facets incident to an edge, but it does not record the cyclic ordering of the edges around a facet.

Our data structures build upon the ideas of the reduced and the extended Würzburg structure. Like in the reduced Würzburg structure we want to represent only the local pyramids of the smallest elements of the incidence structure. But we also want to add the complete incidence structure and even further structural data to allow a convenient and efficient usage of our data structure.

We represent the local pyramid of a vertex by a planar Nef polyhedron embedded on a sphere. Together with the position of the vertex and a set-selection mark for the vertex, we call this structure the sphere map of the vertex. For finitely bounded Nef polyhedra, the sphere maps for all vertices are a sufficient representation, because the smallest elements of the incidence structure are only vertices. In the incidence structure of infinitely bounded polyhedra, edges and facets can also become the smallest elements. Applying the concept of infimal frames reduces the representation of arbitrary Nef polyhedra to finitely bounded Nef polyhedra, and therefore allows a simple, uniform representation with sphere maps. In addition to the sphere maps, we explicitly represent edges, facets and volumes together with their incidences, a set-selection mark, and further structural information. The final structure is a selective cell complex of the three-dimensional space.

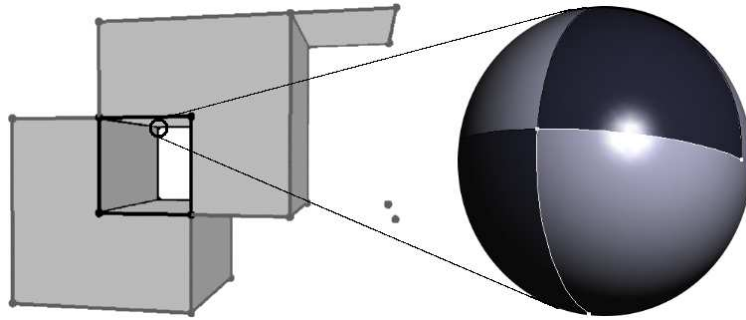


Figure 3.1: An example of a sphere map. The dark regions indicate selected sfaces.

3.1 Sphere Maps

We represent the local pyramid of a vertex by conceptually intersecting the local neighborhood of a vertex with an ε -sphere. This intersection forms a subdivision of the sphere (Figure 3.1) which we represent by a map. A *map* is a bidirected edge-paired graph, i.e., every edge $e = (v, w)$ has a reversal edge $e' = (w, v)$, and there exists a bijective mapping *twin* such that $\text{twin}(e) = e'$ and $\text{twin}(e') = e$. Together with a set-selection mark for each item, the map forms a two-dimensional Nef polyhedron embedded on the sphere. We add a set-selection mark for the vertex and call the resulting structure the *sphere map* of the vertex. Sphere maps were introduced in [DMY93].

We use the prefix *s* to distinguish the elements of the sphere map from the three-dimensional elements. An *svertex* corresponds to an edge intersecting the sphere. An *sedge* corresponds to a facet intersecting the sphere. Geometrically an sedge forms a great arc that is part of the great circle in which the supporting plane of the facet intersects the sphere. When there is a single facet intersecting the sphere in a great circle, we get an *sloop* going around the sphere without any incident vertex. There is at most one *sloop* per vertex because a second *sloop* would intersect the first. An *sfac*e corresponds to a volume. This representation extends the planar Nef polyhedron representation [See01a].

As incidence structure of the sphere maps, we adapt and extend the halfedge data structure provided by CGAL [Ket99]. For each sedge we store two oppositely oriented *shalfedges*. The opposite of an shalfedge is denoted as its *twin*. Each svertex stores a cyclic list of its outgoing shalfedges in counterclockwise order.

Figure 3.2 depicts the relationship between an shalfedge and its incident shalfedges, svertices, and sfaces on a sphere map. An shalfedge is an oriented

3.2. SELECTIVE NEF COMPLEX

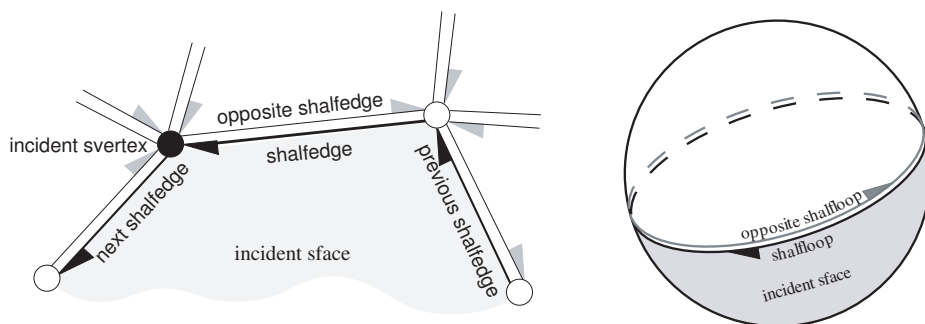


Figure 3.2: Incidences of shalfedges and shalfloops on a sphere map.

edge between two svertices. It is always paired with an shalfedge pointing in the opposite direction.

Note that sphere maps are capable to represent the local pyramid of every location in the three-dimensional space with respect to some Nef polyhedron. Thus, in addition to the locations of vertices, it can also represent locations on an edge, on a facet, or in a volume, no matter whether this volume is the interior of the polyhedron or the outer volume. For our data structure we only need sphere maps to represent the local pyramid of vertices, but during binary operations we also need sphere maps of other locations as an intermediate representation (see Section 4.4).

3.2 Selective Nef Complex

Having sphere maps for all vertices of a polyhedron is a sufficient but not easy accessible representation of finitely bounded Nef polyhedra. We enrich the data structure with more explicit representations of all the faces and incidences between them. We also depart slightly from the definition of faces in a Nef polyhedron; we represent the connected components of a face individually and do not implement additional bookkeeping to recover the original faces (e.g., all edges on a common supporting line with the same local pyramid) as this is not needed in our algorithms. We discuss features in the increasing order of dimension; see also Figure 3.3:

Edges: We store two oppositely oriented halfedges for each edge and link them by pointers from each halfedge to its opposite. Such a halfedge can be identified with an *svertex* in a sphere map; it remains to link one *svertex* with the corresponding opposite *svertex* in the other sphere map.

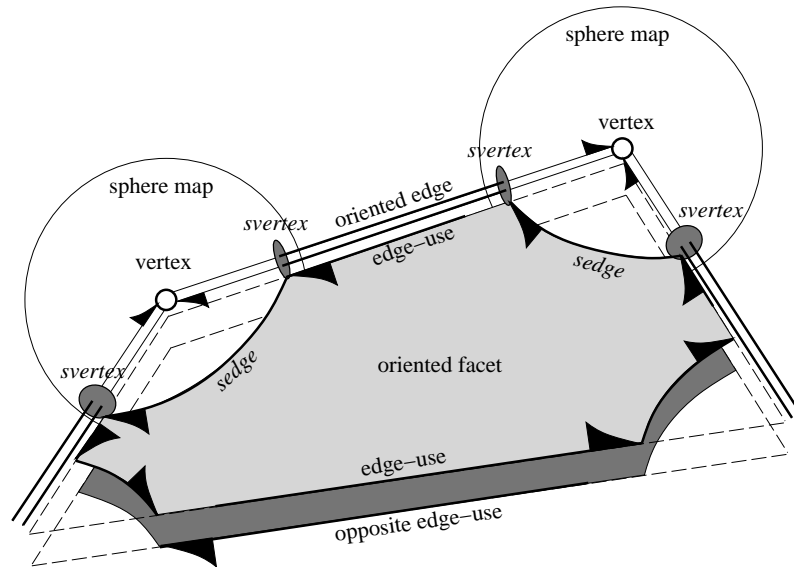


Figure 3.3: A selective Nef complex: We show one facet with two vertices, their sphere maps, the connecting edges, and both oriented facets. Shells and volumes are omitted.

Edge-uses: An edge can have many incident facets (non-manifold situation). We introduce two oppositely oriented edge-uses for each incident facet; one for each orientation of the facet. An edge-use points to its corresponding oriented edge and to its oriented facet. We can uniquely identify each edge use with an *shalfedge*, or, in the special case, also with an *shalfloop*.

Facets: We store oriented halffacets as boundary cycles of oriented edge-uses. We have a distinguished outer boundary cycle and several (or maybe none) inner boundary cycles representing holes in the facet. Boundary cycles are linked in one direction. We can access the other traversal direction when we switch to the oppositely oriented halffacet, i.e., by using the opposite edge-use.

Shells: The volume boundary decomposes into different connected components, the *shells*. They consist of a connected set of facets, edges, and vertices incident to this volume. Facets around an edge form a radial order that is captured in the radial order of *sedges* around an *svertex* in the sphere map. Using this information, we can traverse a shell completely starting at an arbitrary entry element with a graph search.

Volumes: A volume is defined by a set of shells, one outer shell containing the

3.3. INFIMAXIMAL BOX - A REDUCTION TO FINITELY BOUNDED POLYHEDRA

volume and several (or maybe none) inner shells excluding voids from the volume.

The point sets defined by the vertices, edges, facets, and volumes of a polyhedron form a cell complex of the three-dimensional space, i.e., a subdivision of \mathbb{R}^3 into 0, 1, 2, and 3-dimensional relative open sets. The notion of a cell complex is closely related to the notion of an arrangement. While a cell complex is purely topological, an arrangement is a cell complex that is induced by a set of geometric objects. Hence, we can also denote our data structure as an arrangement induced by a set of half-spaces.

For each cell of our cell complex, i.e., for each vertex, edge, facet and volume, we store a label. This label can be of an arbitrary type. To model Nef polyhedra, the labels are set-selection marks. A selected face indicate a point set that is part of the polyhedron, while an unselected face indicates a point set that is excluded, e.g., the outer volume or a facet that is only bounding the polyhedron, but is not a part of it. We call the resulting data structure *selective Nef complex*, *SNC* for short.

3.3 Infimaximal Box - A Reduction to Finitely Bounded Polyhedra

Nef polyhedra can be represented by the local pyramids of the minimal elements in the incidence relation \prec defined in Definition 2.5. For finitely bounded polyhedra those minimal elements are only vertices, but for infinitely bounded polyhedra this property does not hold. For example, the sole minimal element of a polyhedron representing a line, is an edge. In this section, we present a reduction from arbitrary to finitely bounded polyhedra. Applying the reduction, all minimal elements of the incidence structure are vertices. Hence, representing the local pyramids of all vertices by a sphere map becomes a sufficient representation of infinitely bounded Nef polyhedra, too.

As a reduction, we adapt infimaximal frames as presented in [MS03] for three-dimensional Nef polyhedra. Seel has already applied this approach for planar Nef polyhedra [See01a, See01b]. In three-dimensional space, the *infimaximal box* is a bounding volume of size $[-R, +R]^3$ where R represents a sufficiently large value to enclose all vertices of the polyhedron. The value of R is left unspecified as an *infimaximal number*, i.e., a number that is finite but larger than the value of any concrete real number. Clipping lines, rays, and planes at the infimaximal box leads to points and segments on the box, i.e., polyhedra become finitely bounded. As a result, we are left with only vertices as minimal elements of \prec .

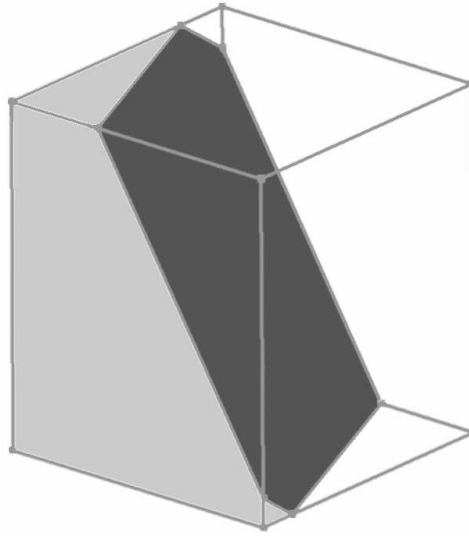


Figure 3.4: The half-space defined by the plane $3x + 5y + 7z + 9 = 0$ clipped at the infimal box. For visualization, R is set to a suitable finite value.

Mehlhorn and Seel argue that interpreting R as an infimal number instead of setting it to a large concrete number has several advantages, in particular increased efficiency and convenience [MS03]. Using a large concrete number requires the computation of a sufficiently large value for each performed operation. Then all rays and lines must explicitly be clipped at a frame of the computed size. Furthermore, a single point with large coordinates forces the usage of a large frame. In consequence, the endpoints of clipped rays and lines have very large coordinates, too. Large coordinates are a major efficiency problem of exact arithmetic. In Particular, floating-point filters are most effective when point coordinates are small [BFS98, FW96, MN94].

For projective geometry, Mehlhorn and Seel show that a plane sweep for segments cannot be generalized to inputs containing rays and lines [MS03]. We did not examine the applicability of projective geometry to our three-dimensional procedure, but it needs special treatment of rays, lines, and facets and shells bounded by rays and lines. This special treatment is not necessary with an infimal box. Instead, we can develop our algorithms and data structures uniformly for finitely bounded and arbitrary polyhedra.

We denote the points on the box as *frame points* or *non-standard points* (compared to the regular *standard points* inside the box). The coordinates of such points are R or $-R$ for at least one coordinate axis, and linear functions $f(R)$ for the other

3.3. INFIMAXIMAL BOX - A REDUCTION TO FINITELY BOUNDED POLYHEDRA

coordinates. We use linear polynomials over R as coordinate representation for standard points as well as for non-standard points, thus unifying the two kinds of points in one representation, the *extended points*. In Lemma 3.1, we show that this representation is always sufficient, even in iterated constructions.

Analogous to non-standard and extended points, we can define *non-standard segments*, *extended segments*, *non-standard planes* and *extended planes*. Non-standard segments have at least one non-standard point as endpoint. They arise from clipping standard planes at the infimaximal box. Non-standard planes only occur as the supporting planes of the sides of the infimaximal box. Extended segments and planes are the unified representation of standard and non-standard segments and planes, respectively.

It is easy to compute predicates involving extended points. In fact, all predicates in our algorithms resolve to the sign evaluation of polynomial expressions in point coordinates. With the coordinates represented as polynomials in R , this leads to polynomials in R whose leading coefficient determines their signs.

We will also construct new points and segments. The coordinates of such points are defined as polynomial expressions of previously constructed coordinates. Fortunately, the coordinate polynomials stay linear even in iterated constructions.

Lemma 3.1. *The coordinate representation of extended points in three-dimensional Nef polyhedra is always a polynomial in R with a degree of at most one. This also holds for iterated constructions where new planes are formed from constructed (standard) intersection points.*

Proof. We show the second part of the Lemma first: In iterated constructions, the expression for computing new points, e.g., from intersecting planes and/or edges, is a rational expression where it is not obvious that it must simplify to a linear polynomial. On the other hand, the constructed point is either a standard point or the intersection of two extended segments. An extended segment results from clipping a facet supported by a standard plane at the infimaximal box. Hence, the intersection point of two extended segments results from clipping the intersection line of two facets at the infimaximal box. As a result, it is a non-standard point, i.e., it has a representation with coefficients linear in R . The rational expression must be equal to that representation and thus simplify.

We prove the first part of the Lemma: Frame points in a three-dimensional Nef polyhedron result from lines and rays clipped at the infimaximal box. Consider a line l defined by

$$l : \begin{pmatrix} x_1 \\ y_1 \\ z_1 \end{pmatrix} \lambda + \begin{pmatrix} x_0 \\ y_0 \\ z_0 \end{pmatrix}.$$

In the following, we list the endpoints p_1 and p_2 of l for every assignment of the values $x_i, y_i, z_i, i = 0, 1$. Because the infimaximal box is symmetric in all three dimensions, there are many symmetric cases in the listing of the endpoints. Without loss of generality we assume that $|x_1| \geq |y_1| \geq |z_1|$ and distinguish the following cases:

- $|x_1| > |y_1|$
 $p_1 = (R, \frac{R-x_0}{x_1}y_1 + y_0, \frac{R-x_0}{x_1}z_1 + z_0), p_2 = (-R, \frac{-R-x_0}{x_1}y_1 + y_0, \frac{-R-x_0}{x_1}z_1 + z_0)$

- $|x_1| = |y_1|, |x_1| > |z_1|$, without loss of generality $\frac{y_1}{x_1} = 1$
 - $x_0 - y_0 > 0$
 $p_1 = (R, R - x_0 + y_0, \frac{R-x_0}{x_1}z_1 + z_0), p_2 = (-R + x_0 - y_0, -R, \frac{-R-y_0}{y_1}z_1 + z_0)$

 - $x_0 - y_0 < 0$
 $p_1 = (-R, -R - x_0 + y_0, \frac{-R-x_0}{x_1}z_1 + z_0), p_2 = (R + x_0 - y_0, R, \frac{R-y_0}{y_1}z_1 + z_0)$

 - $x_0 = y_0$
 $p_1 = (R, R, \frac{-R-x_0}{x_1}z_1 + z_0), p_2 = (-R, -R, \frac{R-x_0}{x_1}z_1 + z_0)$

- $|x_1| = |y_1| = |z_1|$

We omit a detailed and straight forward discussion of the 28 sub-cases, here.

We now prove in detail that the listed endpoints for the case $|x_1| > |y_1|$ are correct. The proof of the other cases works analogously. We can verify, that the points $(R, R - x_0 + y_0, \frac{R-x_0}{x_1}z_1 + z_0)$ and $(-R + x_0 - y_0, -R, \frac{-R-y_0}{y_1}z_1 + z_0)$ lie on l . Also, we can see that they lie on the supporting planes of the sides of the infimaximal box in positive and negative x -direction, respectively. It is left to show that the point does not lie outside of the infimaximal box. They are on the boundary of the infimaximal box, if

3.3. INFIMAXIMAL BOX - A REDUCTION TO FINITELY BOUNDED POLYHEDRA

$$|(\pm R - x_0)\frac{y_1}{x_1} + y_0| \leq R \quad (3.1)$$

$$|(\pm R - x_0)\frac{z_1}{x_1} + z_0| \leq R \quad (3.2)$$

With $C = -\frac{y_1}{x_1}x_0 + y_0$, inequality 3.1 can be rewritten as follows:

$$\begin{aligned} & |\pm R\frac{y_1}{x_1} + C| \leq R \\ \Leftrightarrow & \quad \left|\frac{y_1}{x_1}\right|R + |C| \leq R \\ \Leftrightarrow & \quad \left|\frac{y_1}{x_1}\right|R - R \leq |C| \\ \Leftrightarrow & \quad \left(\left|\frac{y_1}{x_1}\right| - 1\right)R \leq |C| \end{aligned}$$

The final inequality is true, since we have an arbitrary large negative value on the left side, which is smaller than any constant value C . Inequality 3.2 follows analogously. Consequently, the given points are correct endpoints of l . \square

At compile time, we provide two modes for the work with Nef polyhedra, i.e., working with standard or extended geometry. The first template parameter of our main class `Nef_polyhedron_3` is used for specifying the underlying geometry. For this purpose we provide two *extended kernels*, namely `Extended_homogeneous` and `Extended_cartesian`. They differ from the standard CGAL kernels in using polynomials for representing coordinates. Extended kernels can represent extended points and segments and can therefore handle the full modeling space of Nef polyhedra.

We also offer the parameterization of `Nef_polyhedron_3` with a standard CGAL kernel. A standard kernel restricts the modeling space to finitely bounded Nef polyhedra. Still, the modeling space is closed under Boolean and topological operations. Standard kernels are considerably faster than extended kernels.

Note that we follow a different strategy than Seel's implementation of planar Nef polyhedra [MS03]. To date, planar Nef polyhedra only work with extended kernels, while the user is restricted only to work on standard geometry. Geometry is accessed by special functions, which interpret frame points as standard rays. Edges that completely lie on the infimaximal square—the two-dimensional equivalent of the infimaximal box—can be identified, but can not be accessed in any other way.

We offer the use of extended and standard kernels. Points, segments, and planes on the infimal box can be identified by special functions, too, but are not interpreted as standard geometry in any way.

Chapter 4

Boolean and Topological Operations

Following Rossignac and O’Conner [RO89], Boolean operations on planar Nef polyhedra work in three steps—overlay, selection, and simplification. The overlay step computes the conventional planar map overlay of the input polyhedra with a sweep-line algorithm [MN99, section 10.7]. The overlay is a combined arrangement of the two input polyhedra. For each face f_o in the overlay there is a face f_i in each of the input polyhedra, such that $f_o \subseteq f_i$. We call f_i the *support* f_o . The selection step computes the mark of each face in the overlay by applying the Boolean expression on the marks of the corresponding supports. This can be generalized to arbitrary functions on label sets. Finally, the simplification step cleans up the data structure and removes redundant representations. This scheme has already been used by Michael Seel on Nef polyhedra in the plane. We adopt it for planar Nef polyhedra embedded on the sphere.

The Boolean operations on spherical Nef polyhedra provide a basis for the Boolean operations on three-dimensional Nef polyhedra. As we pointed out in Chapter 3, as a representation scheme for Nef polyhedra in the three-dimensional space, it suffices to compute the sphere maps of the vertices. This can be done by Boolean operations on sphere maps, which are either provided by the input polyhedra, or are computed on the fly. Having the sphere maps of the result polyhedron, we synthesize a selective Nef complex. Our method may generate redundant sphere maps. We erase them in another simplification step.

The steps are described in detail in the following sections for the case of binary operations. Afterwards we discuss the differences for unary operations.

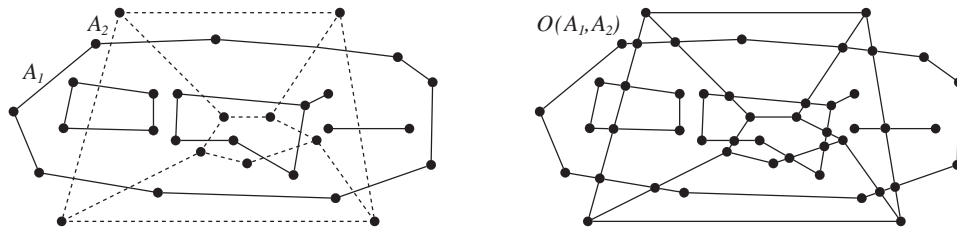


Figure 4.1: Two arrangements A_1 (solid edges) and A_2 (dashed edges), and their overlay $O(A_1, A_2)$.

4.1 Map Overlay on the Sphere

As the first step of a binary operation on spherical Nef polyhedra, we compute the overlay of the two input sphere maps. The overlay of two subdivisions S_1 and S_2 is defined as the subdivision $O(S_1, S_2)$ such that there is a face f in $O(S_1, S_2)$ if and only if there are faces f_1 in S_1 and f_2 in S_2 such that f is a maximal connected subset of $f_1 \cap f_2$. The faces f_1 and f_2 are called the support of $f_1 \cap f_2$ [dBvKOS97].

We are interested in the overlay of two planar arrangements of segments. Adapting our definition from Section 3.2, a planar arrangement of segments $A(S)$ is a subdivision of the plane into 0, 1, and 2-dimensional relative open sets, induced by set of segments S . Figure 4.1 illustrates the notion of an overlay for two planar arrangements A_1 and A_2 . Their overlay $O(A_1, A_2)$ is the arrangement induced by the edges from A_1 and A_2 . Likewise, the overlay of two arrangements of the sphere are induced by the edges and loops of the arrangements.

For planar Nef polyhedra, Seel realized the overlay of planar arrangements with the segment sweep algorithm by Bentley and Ottmann [BO79]. His implementation builds upon the solution described in the LEDA book [MN99], which includes the handling of all degeneracies, but is further generalized for flexible usage [See01b]. As a result, we could adopt his implementation for the overlay of two sphere maps.

In this section, we first give a short introduction to the segment sweep algorithm. Afterwards, we investigate the design of Seel's implementation of the segment sweep algorithm, and show how to compute an overlay of two planar arrangements with the help of a segment sweep algorithm. Finally, we adopted Seel's implementation for computing an overlay of two sphere maps.

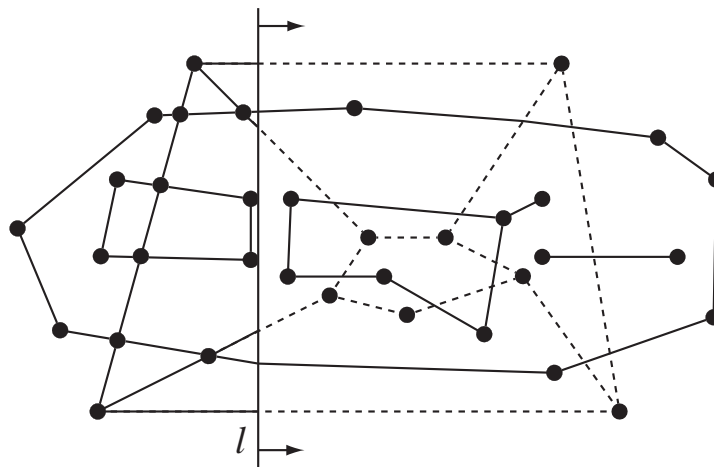


Figure 4.2: The name sweep stems from the image of sweeping a line l over the plane, starting from a position beyond all geometric items. A sweep line algorithm successively constructs the arrangement of the geometric items. As an invariant, the arrangement is always complete up to the current position of l .

4.1.1 A Segment Sweep Algorithm.

Given a set of segments in the plane S , a *segment sweep* algorithm computes the arrangement of the plane induced by the segments. The name sweep stems from the image of sweeping a line l over the plane, starting from a position beyond all segments. During the sweep, the algorithm keeps track of the structure induced by the swept segments and creates the arrangement (see Figure 4.2). Instead of a real line, we sweep an imaginary line and maintain the set of segments intersecting it. The set of intersecting segments is called the *status* of the sweep line. The status changes every time the sweep line reaches an event point, i.e., an endpoint of a segment or the intersection of at least two segments. In addition to the update of the status, each event point triggers an update of the arrangement. As the main invariant of the algorithm, the arrangement is always complete up to the current event point.

The algorithm described in the LEDA book [MN99] suggests to realize the segment sweep algorithm in the following way. The progressing of the sweep line is realized by a lexicographically sorted list containing the event points. Consequently, the sweep line is vertical and sweeps from left to right; if two event points have the same x -coordinate, the point with the lower y -coordinate is processed first. This list is denoted as the *x -structure*. At the beginning, the x -structure is initial-

ized with all endpoints of the segments in S . The intersection points are added successively during the sweep. Like the x -structure, the status is also represented by a sorted list. Because the segments in that list are sorted by the y -coordinate of their intersection point with the sweep line, it is denoted as the y -structure.

Seel follows the approach described in the LEDA book. Details, including the handling of degeneracies can be found in Seel's PhD thesis [See01b] and in the LEDA book [MN99].

4.1.2 A Generic Framework.

The LEDA segment sweep algorithm

```
void SWEEP_SEGMENTS(const list<SEGMENT>& S, GRAPH<POINT,SEGMENT>& G)
```

constructs the LEDA graph G , which represents the arrangement induced by the list of LEDA segments S . The graph G stores the incidence structure, the coordinates of each vertex and the supporting segment of each edge. LEDA provides two versions of the algorithm; one works with homogeneous coordinates represented by arbitrary precision integers, the other works with Cartesian coordinates represented by doubles.

Seel generalized the LEDA approach. His generic sweep framework includes two layers of abstraction. The first layer models the basic process flow of a sweep line algorithm, and delegates all basic operations to a traits class. It consists of a templated class `generic_sweep` with a single routine `sweep()`. The routine `sweep()` is shown in Figure 4.3.

The class `Segment_overlay_traits`, which is the second layer of the generic sweep framework, is a traits class for the class `generic_sweep`. This means that it realizes some specific task that can be accomplished by a sweep line algorithm by implementing the functions delegated by the first layer. In particular, it realizes the overlay of two planar arrangements of segments. The interface of the class `Segment_overlay_traits` also includes three template parameters, which allow an adaptation to diverse incidence structures and geometries. Seel defined the requirements of the templates parameters by the concepts `SegmentOverlayGeometry_2`, `SegmentOverlayInput`, and `SegmentOverlayOutput`.

The `SegmentOverlayGeometry_2` concept defines the geometric requirements of the algorithm. With this concept, we are not restricted to use specific points, segments or number types. It is designed for affine planar geometry, but we will see later in this section that it also works well with other geometric models.

4.1. MAP OVERLAY ON THE SPHERE

```
void sweep() {
    traits.initialize_structures();
    traits.check_invariants();
    post_init_hook(traits);

    while ( traits.event_exists() ) {
        pre_event_hook(traits);
        traits.process_event();
        post_event_hook(traits);
        traits.check_invariants();
        traits.proceed_to_next_event();
    }
    traits.complete_structures();
    traits.check_final();
    post_completion_hook(traits);
}
```

Figure 4.3: The first layer of Michael Seel’s generic sweep framework. The main routine `sweep()` structures the sweep into an initialization, a loop processing the events, and the completion of the output data structures. Additionally, it provides for the checking of invariants and for animation via a hook mechanism. The implementation of these steps is delegated to a `traits` class.

The geometric kernels of CGAL each provide a point and segment type, such that each of the kernels together with CGAL’s global functions works as a model of the `SegmentOverlayGeometry_2` concept without adaptation.

The `SegmentOverlayGeometry_2` concept asks for a segment and a point type, further on denoted as `Segment_2` and `Point_2`, together with the following functions and predicates:

Segment_2 construct_segment(Point_2 p1, Point_2 p2)

Constructs a segment with endpoints *p1* and *p2*.

Point_2 source(Segment_2 s)

Returns the source point of segment *s*.

Point_2 target(Segment_2 s)

Returns the target point of segment *s*.

bool is_degenerate(Segment_2 s)

Decides whether both endpoints of segment *s* are the same.

int compare_xy(Point_2 p1, Point_2 p2)

Compares points *p1* and *p2* lexicographically.

int orientation (Segment_2 s, Point_2 p)

Decides whether p lies on the line l through the endpoints of s (return value 0), or whether it lies on the right or left side of l (return value -1 or 1, respectively).

Point_2 intersection (Segment_2 s1, Segment_2 s2)

Returns the intersection point of the segments $s1$ and $s2$.

The `SegmentOverlayInput` concept asks for an iterator type, which is used for passing the input segments to the segment sweep as an iterator range of `Segment_2` objects. This way, the segment sweep becomes decoupled from the task of storing the input segments.

Finally, the `SegmentOverlayOutput` concept defines the requirements of a planar map as a generic output data structure. Many common data structures like the Halfedge Data Structure or the Directed Cyclic Edge List can be adapted as a model of the `SegmentOverlayOutput` concept. Besides creating the incidence structure, the concept is designed to associate each edge with its supporting input segment, as well as each vertex with the edge lying directly below it. With the first association it is possible to propagate data linked with the segments to the output structure. If the output structure comprises multiple connected components, the second association can be used to resolve their nesting structure.

The concept requires handle types for vertices and halfedges in the output structure, in addition to a point and an iterator type that must be the same as in the `SegmentOverlayGeometry_2` and the `SegmentOverlayInput` concepts. Also, three types of functions are defined. The first set of functions are used to construct the output structure.

Vertex_handle new_vertex (Point_2 p)

returns a new vertex created at point p .

Edge_handle new_halfedge_pair_at_source (Vertex_handle v)

returns a newly created edge inserted before the first edge in the adjacency list of v . Creates also a reversal edge whose target is v .

void link_as_target_and_append (Edge_handle e, Vertex_handle v)

completes the halfedge pair between the source of e and v by making v the target of e and appending the reversal of e to the adjacency list of v .

After the construction of an edge, each segment supporting the edge is identified. The sweep-line algorithm calls the function `supporting_segment` for each

4.1. MAP OVERLAY ON THE SPHERE

of these segments, and thereby indicates them as the edge's support. There are also four functions which indicate the support of newly created vertices. Each of them implies a different relation between the location of the new vertex and its supporting segment. The segment may start or end at the location, or pass through it. The fourth option is a trivial segment supporting the vertex.

void supporting_segment (Edge_Handle e, Iterator it)

Indicates **it* as a supporting segment of the edge *e*.

void starting_segment (Vertex_Handle v, Iterator it)

Indicates **it* as a supporting segment of the vertex *v* that starts at the location of *v*.

void passing_segment (Vertex_Handle v, Iterator it)

Indicates **it* as a supporting segment of the vertex *v* that passes through the location of *v*.

void ending_segment (Vertex_Handle v, Iterator it)

Indicates **it* as a supporting segment of the vertex *v* that ends at the location of *v*.

void trivial_segment (Vertex_Handle v, Iterator it)

Indicates **it* as a trivial segment supporting the vertex *v*.

Finally, for every newly created vertex the function `halfedge_below` reports the edge that lies directly below the new vertex, i.e., it returns the first edge hit by a ray shot from the vertex in negative *y*-direction. With the help of this information, it is possible to resolve the nesting structure of face cycles, as we will see later in this chapter.

void halfedge_below (Vertex_Handle v, Edge_Handle e)

Reports the edge *e* lying directly below vertex *v*.

4.1.3 Overlay of Two Planar Nef polyhedra.

The sweep yields the 1-skeleton of the common arrangement of two planar arrangements and additional information to complete the overlay. A *1-skeleton* of an arrangement only includes the 0 and 1-dimensional objects in the arrangement, i.e., of vertices and edges. At this point, our arrangement does not include the faces. If

the arrangement is represented by a halfedge data structure, as we do it for planar Nef polyhedra and Nef polyhedra embedded on the sphere, the representation includes items for vertices and edges together with proper incidences, but the items for the faces are yet missing. Therefore, we need to create face items and determine the boundary cycles of each face. To complete the overlay, we additionally determine the supports of each item in the overlay. With the help of the supports, we can access additional information stored with a face.

The sweep provides the following properties and informations for the completion of the overlay:

- The embedding of the 1-skeleton is *order-preserving*, i.e., for any vertex, the counterclockwise order of the outgoing edges agrees with the cyclic order of the adjacency list. As a result, the edges naturally form face cycles which agree with the faces of the embedding. They can be traversed easily by means of the twin relation and the order of the adjacency lists.
- For every vertex v we know the halfedge e_b lying directly below the vertex, i.e., e_b is the first edge hit by a ray shot from v in $-y$ direction.
- In case of a support by a vertex or an edge, the supported object knows its unique support from each each of the input arrangements. A vertex always supports a vertex. An edge can support multiple vertices and edges.

First, we create face items. Such a face item point to each of its boundary cycles. Faces are bound by at least one boundary cycle. We distinguish between outer and inner face cycles. Each face—except for the outer face—is enclosed by exactly one outer face cycle, which is a counterclockwise oriented cycle of halfedges. The outer face has no outer face cycle. Inner face cycles bound the holes in a face. There are trivial inner face cycles, which consist of a single isolated vertex, and there are clockwise oriented cycles of edges. In order to decide whether a cycle of halfedges is an outer or inner face cycle, we check the orientation of the cycle at its lexicographically smallest vertex. A left turn indicates an outer, a right turn an inner face cycle (see Figure 4.4). The opposite of a halfedge usually is a halfedge of some other face cycle. Then, the two cycles bound two adjacent or two nested faces. If the two faces are adjacent, both cycles are outer cycles; if the faces are nested, one of them is an inner cycle of the outer face, and the other is the outer cycle of an inner face. In degenerate situations, the opposite halfedge belongs to the same cycle, i.e., both halfedges are incident to the same face.

Figure 4.4 shows a face with two holes, where one of the holes includes two separate faces. All halfedge pairs bound an inner and an outer cycle, except for the

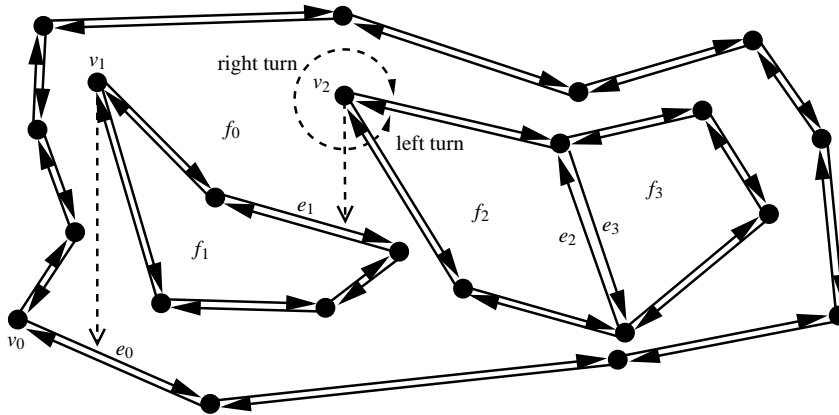


Figure 4.4: The orientation at the lexicographically smallest vertex in a face cycle shows whether the cycle is enclosing a face or is bounding a hole cut into the face. In the former case the cycle makes a left turn at that vertex, in the latter case it makes a right turn. The nesting of face cycles is resolved by recursively shooting a ray from the lexicographically smallest vertex of an inner face cycle until an outer cycle is found.

halfedge pair (e_2, e_3) , which bounds two different inner cycles. Because there are no lower dimensional features, there is no halfedge pair whose halfedges belong to the same cycle.

In order to create and link all face items properly, we start with the creation of the face item for the outer face. Then, we check each face cycle whether it is an outer or an inner cycle. If we identify an outer cycle that is not linked to a face item, we create a new face item and link it properly with the outer cycle. If we identify an inner cycle fc , and it is not already linked to the face item of the face f incident to fc , we obtain the proper face item from some other face cycle fc_{linked} incident to f that is already linked to a face item. The search for the face cycle fc_{linked} , is guided by the information provided by the sweep, i.e., we use the knowledge about the halfedge that lies below a vertex. We obtain the halfedge e_b below the lexicographically smallest vertex of fc . e_b is part of another face cycle fc' incident to f . If fc' has already been processed, fc' is already linked to f . Thus, we can obtain f 's face item and link fc to it. If fc' is an unprocessed outer face cycle, we create a new face item and link it with fc properly. If fc' is an unprocessed inner face cycle, we proceed recursively from fc' . The recursion must end, since the smallest vertex of the current face cycle becomes smaller with each recursion. Incidentally, the recursion will encounter an outer face cycle, or an inner face cycle whose smallest vertex has no halfedge below. In the former case, the final face

cycle encloses all face cycles found during the recursive search; in the latter case, all found cycles are incident to the outer face.

In Figure 4.4, there are four outer cycles. Therefore, we must create four facet items $f_0, f_1, f_2,$ and $f_3,$. Additionally there are three inner cycles, whose smallest vertex is $v_0, v_1,$ and $v_2,$ respectively. Supposing there already are face items for the outer cycles, the linking of the inner cycles works as follows: We arbitrarily start from the inner face cycle with the smallest vertex v_2 . From the orientation at v_2 we can see that the cycle performs a right turn and therefore indeed is an inner cycle. The halfedge-below relation gives us the edge e_1 , which also belongs to an inner cycle as we can conclude from the orientation at vertex v_1 . The halfedge below v_1 is e_0 . The facet cycle that contains e_0 is an outer cycle, since the cycle performs a left turn at vertex v_0 . As a result, we can determine the face item f_0 from that outer cycle and link two encountered inner cycles as holes of f_0 . There is no halfedge below the vertex v_0 . Thus, the inner face cycle that contains v_0 is linked as a hole of the outer face.

Having created and linked all face items, we identify the support of each item in the overlay. The sweep already provided all supports by vertices and edges. It remains to determine the supports by faces. We proceed in a sweep fashion, i.e., we handle the items in lexicographic order, and always finalize the support up to the current event point. The event points are the vertices of the combined arrangement. We maintain the invariant that at any event point, the support has already been computed for all svertices, shalfedges, and sfaces that have been swept at least partially. We start with the support of the outer face to fulfill the invariant at the first event point. Its support are the outer faces of the inputs arrangements.

At each event point, we first obtain the support of the vertex v at that position. Either it is supported by a vertex or edge, then we already know the support, or it is supported by a face. We obtain a face supporting v as the face incident to the halfedge below v .

The outgoing halfedges that lead to lexicographically smaller vertices have already been handled. The other so-called *forward* halfedges e_1, \dots, e_n constitute a single consecutive sequence of the adjacency list of v . They are incident to the faces f_1, \dots, f_n . Because of the invariant, we already know the supports of f_n and the face f_0 , which is the face incident to the twin of e_1 . We identify the supports in the order $e_1, f_1, e_2, \dots, f_{n-1}, e_n$. Knowing the support of f_i , we first check whether e_{i+1} is supported by an edge. If it is not supported by an edge, e_{i+1} and f_{i+1} are supported by the same face as f_i . If it is supported by an edge e_s , the support of f_{i+1} is the face incident to e_s . As a result, we have deduced the support of v and its incident edges and faces.

4.1. MAP OVERLAY ON THE SPHERE

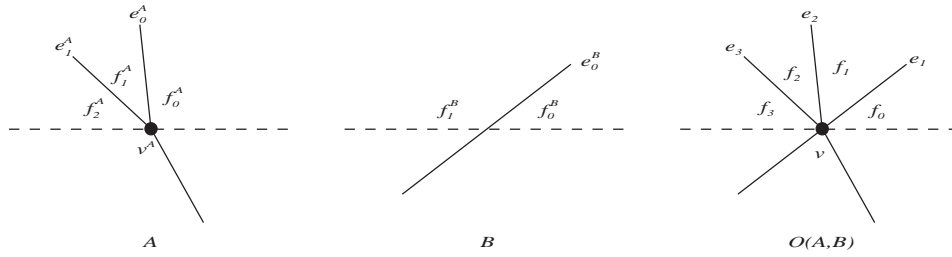


Figure 4.5: The face supports in an overlay $O(A, B)$ are determined in a sweep-line fashion. The vertices are processed in lexicographical order. As an invariant, the face supports are identified up to the current vertex. Processing a vertex v , the face supports of its outgoing forward edges and their incident faces are deduced in counterclockwise order.

The method is illustrated in Figure 4.5 by an example overlay $O(A, B)$. Looking at the supports from subdivision A , the invariant guarantees that the support of f_0 by f_0^A is already known. Since e_1 is not supported by an edge in A , f_0^A also supports e_1 and f_1 . e_2 is supported by e_1^A . As a consequence, f_2 is supported by the incident face of e_1^A , i.e., by f_1^A . The supports of e_3 and f_3 are known. Looking at the supports from B , the support of f_0 by f_0^B is known. As e_1 is supported by e_0^B , f_1 is supported by f_1^B , which is the face incident to e_1^B . Also e_2 , f_2 and e_3 are supported by f_1^B , since both edges are not supported by an edge.

Moving on to the next event point, we can easily see that all items below the sweep line, and all edges and faces crossing the sweep line are incident to some event point that has already been processed. Thus, the invariant holds.

4.1.4 Segment Sweep on the Sphere.

In order to adopt the segment sweep for the sphere, we have to resolve geometric and topological differences. Most important is the choice of a proper sweep line together with its progression. The sweep line must be a continuous curve, which continuously progresses over the sphere visiting every point exactly once. The order in which the points on the sphere are swept, is determined by the `compare_xy` function, which is part of some class that implements the `SegmentOverlayGeometry_2` concept. This function must resemble a proper sweep line, and we want it to be fast, i.e., we do not want to normalize vector coordinates or use sine and cosine functions. As sweep line we use a half-circle that is fixed at its endpoints and rotates around the sphere. It is convenient to use the two intersection points of a coordinate axis with the sphere as the fixed endpoints

of the sweep line. We choose the intersection points with the y -axis. To realize such a sweep line, the `compare_xy` function sorts points on the sphere by three-dimensional orientation tests. The orientation test decides whether a point lies on the left or on the right side of the sweep line. If two points lie on the same sweep line, we determine their relative position on the sweep line. For this purpose, we obtain the orientation of the second point with respect to a segment passing the first point orthogonal to the sweep line. The sweep line fulfills the stated requirements except for one. The two endpoints of the sweep line are swept in any position of the sweep line. We handle those endpoints separately; one of them is ranked by `compare_xy` as the smallest, and the other as the largest point of all. This way, they are processed once, as the first and last event point.

Problems arise from the cyclic nature of the sweep line. There is no natural beginning or ending position of the sweep line movement. Each initial sweep line might already intersect several segments. Therefore, it is not possible to state that the arrangement has completely been constructed for the swept area. Also, the three-dimensional orientation predicate is only suitable for half-sphere geometry. Using it in the `compare_xy` function as described above, it compares two points with respect to a full great circle instead of a half-circle. Also, the predicate has no means for deciding whether a point is lying before or after the initial sweep line.

Furthermore, we have to deal with loops and with the ambiguities that occur when we define a spherical segment by its two endpoints. Handling loops requires an extended incidence structure which is not supported by the `Segment_overlay_traits`. Constructing a segment on the sphere from two points as part of a great circle, there are always at least two possibilities (except for the trivial case). If the points are not opposite to each other, they define a distinct great circle, but the segment could go either way around the sphere. If the endpoints are opposite to each other, they do not even define a distinct great circle. As the `SegmentOverlayGeometry_2` concept does not allow additional information for the construction of a segment besides the two endpoints, we cannot deterministically construct the correct segment.

Because of these problems, we want to perform overlays on half-spheres rather than on a full sphere. On a half-sphere, there are no loops and no edges longer than a half-circle. Only the problems of handling half-circles remains open. We proceed as follows: We cut each segment at the xy -plane. Then we add an equator in the xy -plane by connecting the cut segments. Finally, we get rid of the remaining half-circles by cutting them in two halves. Figure 4.6 illustrates this process.

The two half-spheres can be swept separately. After the overlay of both halves have been completed—we discuss the overlay below—the half-spheres are re-joined. Note that redundant equator edges must be removed twice. First, the

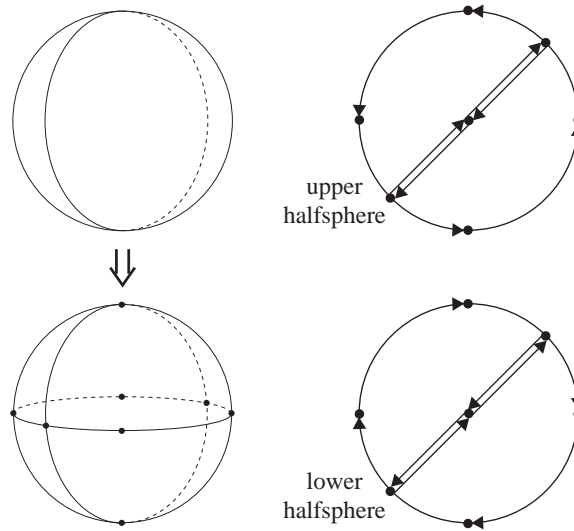


Figure 4.6: On the left side, a sphere map with a sloop is prepared for the half-sphere sweep: Equator edges are inserted and long sedges are cut into two halves. On the right side, the two half-spheres are shown separately and in detail for better illustration. Both are projected into the $z = 0$ plane and are viewed from the top.

sweep-line algorithm creates halfedge pairs for each equator edge on both half-spheres. Therefore, each equator halfedge exists twice and we have to erase one half of them during the rejoin. Second, most of the equator edges are redundant, but some of them might be necessary. We remove them later during the simplification process. For this purpose, we assign proper marks to each equator edge. The mark of a newly inserted equator edge coincides with the mark of the face that is divided by that edge. If it is still redundant after the sweep, the marks of the divided face and the equator edge will still be equal. As a result, the simplification step can remove the edge as we will see in Section 4.3.

The identification of the support in the half-sphere overlay includes one major difference to the planar version: a half-sphere has no outer sface. Instead, there are the sfaces of the other half-sphere that border on the sfaces of the currently processed half-sphere hs . Looking at the full sphere, the sface that is incident to the twin of the first forward shalfedge outgoing from the smallest svertex v_s in hs can be used as a replacement for the outer sface. We obtain the support of this sface by a point location query on the input sphere maps at the location of v_s . Either the query directly returns an sface, or we obtain the proper sface from the incidence structure of the returned item.

4.2 Selection

The overlay of two planar (or spherical) Nef polyhedra yields their combined arrangement and the support of each vertex, halfedge, and face. No matter which boolean operation is applied on the planar polyhedra, their combined arrangement is sufficient for the representation of the result polyhedron. We obtain the correct mark of an item i in the combined arrangement by applying the boolean operation upon the marks of i 's supports.

As a default, each item carries a set-selection mark of type `bool`. As an alternative, other labels can be used. The replacement label type needs to define at least one of the functions `operator&&`, `operator||`, and `operator-`, which are used by the selection step to combine the labels. With boolean labels, these functions are pre-defined as expected, and therefore realize the union, intersection, difference operation, respectively. The symmetric difference naturally combines the functions `operator&&`, `operator||`, and `operator!`. It can also be useful to replace the `operator!`, because it is also used in the negation operation.

In Section 10.2, we present an example for the use of different labels in the computation of the Minkowski sum of two convex polyhedra.

4.3 Simplification on the Sphere

According to Nef's theory [Nef78, Bie95], a face is a maximal set of points with the same local pyramid. As we pointed out in Chapter 2, we represent the connected components of each face separately. This representation is unique. During the overlay step, we compute a combined arrangement of multiple polyhedra. Independent of the concrete Boolean operation, this arrangement is suitable for representing the resulting Nef polyhedron. As a consequence, dependent on the concrete Boolean operation, items may be redundant. The uniqueness of the representation is restored by a simplification step, which identifies and erases all redundant items. In Figure 4.7 we see the combined arrangement $O(S, T)$ of a square S and a triangle T . In the arrangement $O(S, T)$, the triangle is subdivided into an upper and a lower part. In the difference between the square and the triangle the items of the lower part redundantly subdivides the outer face and therefore can be joined with it; in a union the top part redundantly subdivides the only face of the resulting polyhedron.

An item is redundant, if it is either surrounded by some higher-dimensional item, or separates two higher-dimensional items lying in a common hyper-plane. In both cases, the relevant items have the same mark. For example, we can erase a vertex lying on a face with the same mark. Surely this cannot be done, if there are

4.3. SIMPLIFICATION ON THE SPHERE

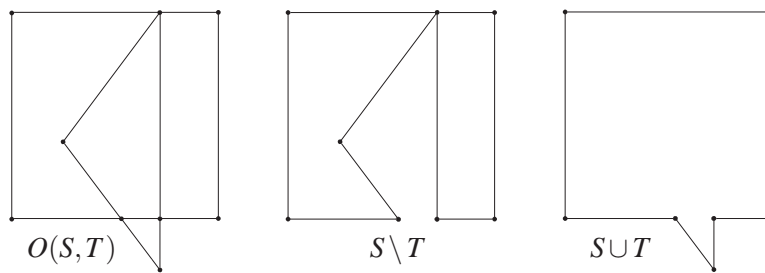


Figure 4.7: The combined arrangement of a square S and a triangle T , the arrangement of their difference $S \setminus T$, and the arrangement of their union $S \cup T$.

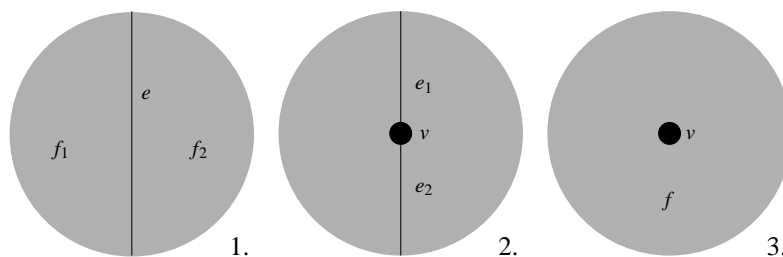


Figure 4.8: Basic situations that trigger a simplification.

edges incident to the vertex. Then we first have to find out whether those edges are redundant, too.

For planar polyhedra there are three basic situations (see Figure 4.8) that trigger a simplification:

1. An edge e , which separates two faces f_1 and f_2 . e , f_1 and f_2 have the same marks. f_1 and f_2 can be equal, i.e., e is surrounded by same face with the same mark. \Rightarrow Delete e . Unite f_1 and f_2 if necessary.
2. A vertex v , which separates two collinear edges e_1 and e_2 . v , e_1 and e_2 have the same marks. \Rightarrow Delete v . Unite e_1 and e_2 .
3. A vertex v without incident edges, which lies in a face with the same mark. \Rightarrow Delete v .

To clean up the structure completely and efficiently, we first check all edges for situation 1. As a result, no vertex can have redundant incident edges when checked for isolation in the next step. Afterwards, all vertices can then be checked for situations 2 and 3 simultaneously.

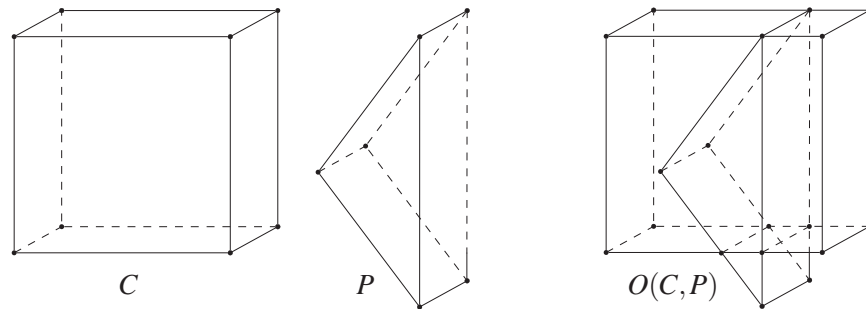


Figure 4.9: A cube C , a prism P , and their combined arrangement $O(C,P)$. The combined arrangement of multiple polyhedra exactly holds the vertices which may occur in the result of a Boolean operation on them. Which of these vertices are finally needed depends on the specific operation.

For the merge operation of faces in situation 1, we cannot afford to maintain an updated status of the face objects after every single simplification, as this would imply the repeatedly iteration of face cycles. We avoid a quadratic runtime by using a union–find data structure [CLR90]. Thus, situation 1 is handled as follows: While the separating edge is deleted directly, and the outer face cycle is concatenated properly, the face objects together with the mutual incidence with their boundary cycle objects remain untouched. Instead, the union–find data structure keeps track of the united faces. After all occurrences of situation 1 have been handled this way, the update of the face objects and their associated incidence pointers can be done in linear time.

The adaptation for sphere maps is simple. Identifying the situations needs incidence informations only, except for the collinearity test in situation 2. We adapt to spherical geometry by checking if the two sedges lie on the same great circle, instead. Additionally, we have to deal with sloops. They can be handled analogously to sedges in situation 1, i.e., if an sloop redundantly separates two faces, we can erase the sloop and unite the faces. But since we have not erased the redundant equator sedges before the simplification step, sloops are still cut into several sedges. On the other hand, the simplification routine might unite several sedges to an sloop. The conversion of an sedge with identical end points into an sloop is trivial.

4.4 Candidate Sphere Maps

The set of vertices in a polyhedron P_{res} resulting from an n -ary Boolean operation b on polyhedra P_1, \dots, P_n is a subset of the vertices in the combined arrangement of P_1, \dots, P_n . They are either located at the position of a vertex in any of the P_i 's, or at an intersection of P_i and P_j . For binary operations we must consider edge–edge and edge–facet intersections. These locations are the locations of the vertices in the overlay of the P_i s (see Figure 4.9).

As for the binary operations on two-dimensional Nef polyhedra, not the complete combined arrangement is needed for representing the result. We determine the sphere maps for all vertex locations of combined arrangement, first. Those sphere maps are a sufficient representation of the result polyhedron, but may include redundant sphere maps. We identify the redundant sphere maps and erase them. In contrast to our approach of binary operations on the two-dimensional Nef polyhedra, the overlay is not computed. Instead, the polyhedron can directly be synthesized from the sphere maps that remain after the simplification.

The sphere map of P_{res} at location l is calculated by applying b on the sphere maps representing the local neighborhood of l in each of the polyhedra P_i . We already have sphere maps at location l in all P_i s that have a vertex at l . For all other P_i s, the location l lies on an edge, on a facet, or in a volume. Remember that sphere maps can represent the local pyramid of every location in the three-dimensional space with respect to some given Nef polyhedron, not only the local pyramids of vertices. Thus, we can compute proper sphere maps for l on the fly, if it is necessary. In case of a volume, the sphere map that represents location l consists of a single sface with the same mark as the volume. If l lies on a facet, the sphere map has two sfaces separated by a sloop. The marks are taken from the facet and the incident volumes. In case of l lying on an edge, the sphere map has two opposite svertices with an sedge connecting them for each facet incident to the edge. The marks are taken from the edge, the incident facets and volumes.

4.5 Simplifying the Selective Nef Complex

Given two polyhedron P and Q , we created a set a sphere maps that is sufficient to represent the result polyhedron of a Boolean operation performed on P and Q in the previous step. In particular, we created sphere maps for all vertex locations of the combined arrangement of P and Q . The set of these vertex locations is a superset of the vertex locations in the result polyhedron. Depending on the specific Boolean operation, some of the vertices occur in the result, and some do not. In

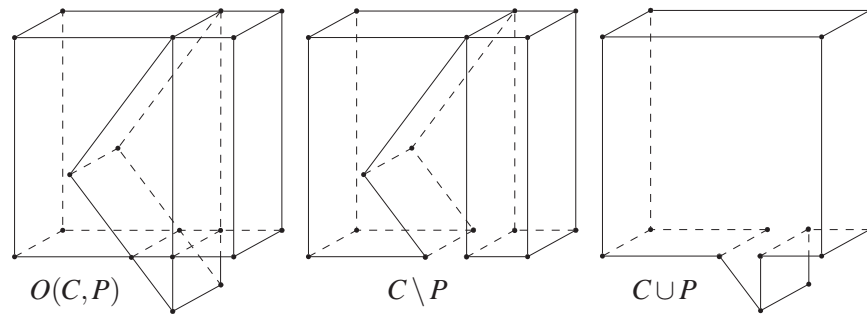


Figure 4.10: The combined arrangement $O(C,P)$ of a cube C and a prism P , and the arrangements of $C \setminus P$ and $C \cup P$. In the difference operation, two vertices of P are absorbed by facets, and two others by edges of C . In the union operation, the final two vertices of P get absorbed in the outer volume.

the latter case, the sphere maps represents a location on an edge, on a facet, or in a volume. This happens when for example the vertex of one input polyhedron is absorbed by an edge, facet, or volume of the other polyhedron. These sphere maps are redundant and will be erased in this step.

As an example, Figure 4.10 shows the combined arrangement $O(C,P)$ of a cube C and a prism P , together with the arrangements of $C \setminus P$ and $C \cup P$. The latter two arrangements include all vertices of $O(C,P)$. In $C \setminus P$, the two lower vertices of P have been absorbed into the outer volume. In $C \cup P$, the two left vertices of P have been absorbed into the volume of C , and the two upper vertices of P have become a part of the boundary of C .

As discussed in Section 4.4, sphere maps that represent a position on an edge, on a facet, or in a volume, have special structure. Thus, it is easy to identify them. Once such a redundant sphere map is identified, it can simply be deleted together with all its items. Since the SNC has not been synthesized yet, the sphere maps are not linked by any pointers. They are only implicitly linked by their geometric properties, which will later help us to perform the synthesis.

4.6 Synthesizing the SNC

Given the sphere maps of a particular polyhedron, it is still complex to solve interesting geometric queries on the polyhedron. For instance, it is very complicated to solve a point location query, because there are no edge, facet, or volume items that could be returned. As a result, a user must first define these object.

Providing the SNC is a necessity not only because of convenience but also because of efficiency reasons. Determining edges, facets, facet cycles, shells or volumes on demand can increase the complexity of algorithms performed on a polyhedron essentially. As an example, without precomputation the identification of the opposite endpoint of some edge needs at least time linear in the size of the polyhedron.

The selective Nef complex complements the information provided by the sphere maps. In this section, we describe how to synthesize the selective Nef complex. The synthesis works in order of increasing dimension.

4.6.1 Pairing up Halfedges.

Interpreting an svertex as a halfedge in a three-dimensional polyhedron, the center vertex of the sphere map becomes the source vertex of the halfedge. The direction of the halfedge is the direction from the center vertex to the svertex. A halfedge has a unique supporting line, which is defined by the position of its source vertex and its direction. Edges are identified by two svertices directly opposite to each other, i.e., they have the same supporting line, they are oppositely oriented, and there is no other vertex lying on the same supporting line between them.

We create halfedge pairs as follows: First, we compute a normalized line representation for each halfedge, and group halfedges that lie on the same supporting line in a common list. Then we sort each list such that consecutive halfedges can be linked as halfedge pairs.

To group halfedges with the same supporting line, we use normalized Plücker coordinates of the line [Sto91]. The Plücker coordinates of a line l can be determined from two distinct points p and q on l . For a halfedge with source vertex at location s and direction v , we set $p = s$ and $q = s + v$. The Plücker coordinates are a sixtuple, which represent the line defined by p and q uniquely up to a multiplicative factor. It is computed as follows:

$$\begin{pmatrix} p.x() * q.y() - p.y() * q.x(), \\ p.x() * q.z() - p.z() * q.x(), \\ p.y() * q.z() - p.z() * q.y(), \\ p.x() - q.x(), \\ p.y() - q.y(), \\ p.z() - q.z() \end{pmatrix} \quad \begin{pmatrix} p.hx() * q.hy() - p.hy() * q.hx(), \\ p.hx() * q.hz() - p.hz() * q.hx(), \\ p.hy() * q.hz() - p.hz() * q.hy(), \\ p.hx() * q.hw() - p.hw() * q.hx(), \\ p.hy() * q.hw() - p.hw() * q.hy(), \\ p.hz() * q.hw() - p.hw() * q.hz() \end{pmatrix}$$

Cartesian coordinates

homogeneous coordinates

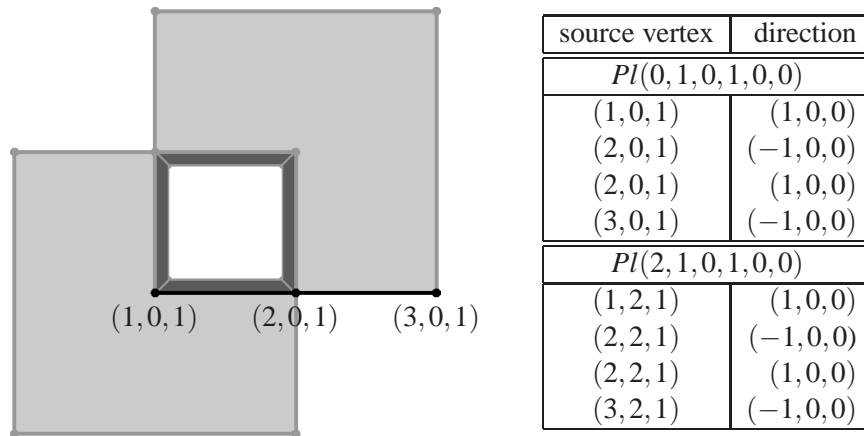


Figure 4.11: Symmetric Difference of two axis-aligned cubes with corners at $(0, 0, 0)$, $(2, 2, 2)$, and $(1, 0, 1)$, $(3, 0, 3)$: The table lists the position of the source vertex and the direction of the halfedges with the supporting lines $Pl(0, 1, 0, 1, 0, 0)$ and $Pl(2, 1, 0, 1, 0, 0)$. The halfedges are lexicographically sorted by the location of their source vertices. The direction breaks the tie.

Since we want to group all halfedges with the same supporting line in a common list, we need to normalize the Plücker coordinates. For homogeneous coordinates we achieve normalization by division with the common greatest divisor of all six Plücker coordinates and negate them if the first coordinate is negative; for Cartesian coordinates we divide all by the first Plücker coordinate.

Since the source vertices of twin halfedges must lie next to each other on their supporting line, we sort the lists lexicographically. The lexicographic order always coincides with the order of points on their common supporting line. There can be two oppositely oriented halfedges e^+ and e^- with the same source vertex lying on the same supporting line. Because of the lexicographic order, the twin halfedge of e^- , the one pointing in a direction of lexicographic smaller points, is sorted directly before e^- and e^+ . The twin halfedge of e^+ is sorted directly after them. Thus, we break the tie by sorting e^- before e^+ .

The method is illustrated by Figure 4.11. For a polyhedron constructed by the symmetric difference of two cubes, it shows two of the lists of halfedges with common Plücker coordinates. Each halfedge is identified by the homogeneous coordinates of its source vertex and its direction. The vertices $(1, 0, 1)$, $(2, 0, 1)$, and $(3, 0, 1)$ lie on a common supporting line with normalized Plücker coordinates $Pl(0, 1, 0, 1, 0, 0)$. Consequently, those are the normalized Plücker coordinates of the four halfedges connecting these vertices. There are two halfedges with source

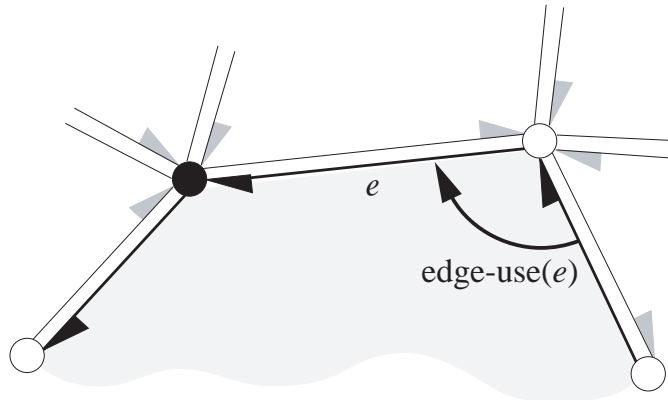


Figure 4.12: The shalfedge representing the edge-use of edge $e = (v_1, v_2)$ in half-facet f is the shalfedge in the sphere map around v_1 that lies in the supporting plane that is oppositely oriented to f and whose target svertex is e .

vertex $(2, 0, 1)$. The halfedge with direction $(-1, 0, 0)$ points into a direction with lexicographically smaller points, and is therefore sorted before the one with direction $(1, 0, 0)$. The first and second halfedge of the list are paired as a halfedge pair. So are the third and the fourth.

In case of extended points we proceed the same way. For supporting lines that do not lie completely on the infimal box, the normalized Plücker coordinates of the frame points are the same as for the non-frame points. For the lines lying completely on the infimal box, we get normalized Plücker coordinates whose three leading coordinates are each either zero, or a polynomial in R of degree one. The other three coordinates are constants. As an example, the edge from $(-R, -R, R)$ to $(R, 0, R)$ has the normalized Plücker coordinates $Pl(R, -2R, -R, -2, -1, 0)$. To sort the vertex coordinates of frame points, we use the notion of R as an infimal number, i.e., a number greater than any other.

4.6.2 Creation of Facet Cycles.

Facets are bounded by at least one boundary cycle. The outer cycle is obligatory and consists of edges. Inner cycles border holes. They either consist of a single vertex or of a cycle of edges. Since edges and vertices may occur in multiple facet cycles, we regard facet cycles as cycles of edge-uses rather than edges. Instead of introducing additional items for the edge-uses, we associate the edge-use of halfedge $e = (v_1, v_2)$ in halffacet f with the shalfedge se , such that se has the same oriented supporting plane as f and is part of the sphere map of v_1 (see Figure 4.12).

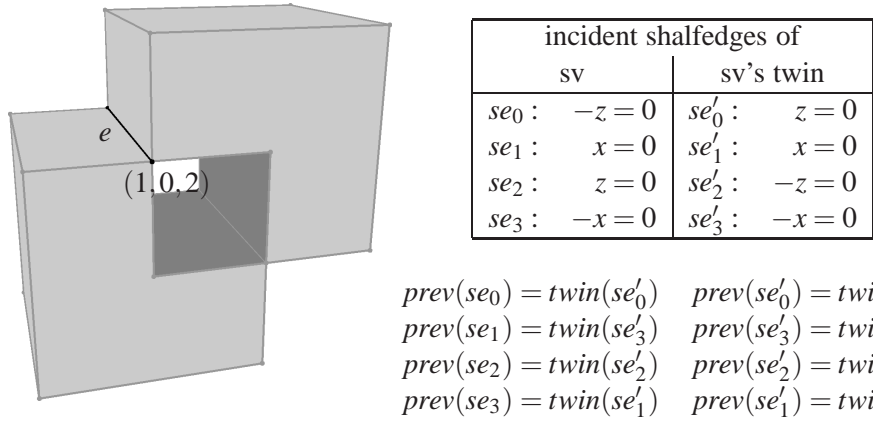


Figure 4.13: Symmetric Difference of two axis-aligned cubes with corners at $(0,0,0)$, $(2,2,2)$, and $(1,0,1)$, $(3,0,3)$: The table lists the shalfedges incident to the shalfedge pair $e = (sv, twin(sv))$, i.e., the svertex with source vertex $(1,0,2)$ and direction $(0,1,0)$, and its twin. The shalfedges are denoted by their supporting planes in the the sphere's coordinate system and are listed in counter-clockwise order. The new previous pointers are listed below the table. The next pointers link in the opposite direction.

A boundary cycle consisting of a single vertex is regarded as a trivial edge-use. We associate the trivial edge-use of vertex v on facet f with the shalfloop on the sphere map of v that has the same oriented supporting plane as f .

We link shalfedges to previous–next pairs, such that a boundary cycle of halfedges e_1, \dots, e_n is represented by a cycle of shalfedge se_1, \dots, se_n , where se_i represents the edge-use of e_i . For this purpose, we form facet cycles by linking together sedges that are incident to twin shalfedges and lie in the same supporting plane. To identify the previous–next pairs for all shalfedges adjacent to svertex sv , we first search the adjacency list of sv 's twin for an shalfedge se_2 lying on the oppositely oriented supporting plane than the first shalfedge se_1 outgoing from sv . The shalfedges se_1 and se_2 lie on oppositely oriented facet cycles passing along the same edges and vertices. In one of these two facet cycles the twin of se_1 is the predecessor of se_2 ; in the other facet cycle the twin of se_2 is the predecessor of se_1 . We link them as previous–next pairs, accordingly. We use the counter-clockwise order of the shalfedges outgoing from sv and its twin to link together the remaining sedges adjacent to sv . In this fashion we create all previous–next pairs, and consequently also form all facet cycles.

In Figure 4.13, we look at the svertex sv with source vertex $(1,0,2)$ and direc-

tion $(0, 1, 0)$. Its twin svertex has source vertex $(1, 2, 2)$ and direction $(0, -1, 0)$. The table lists the supporting planes of the outgoing shalfedges of both svertices in counter-clockwise order. Remember that in a sphere map the plane equalities are given with respect to the center of the sphere. The first outgoing shalfedge of sv lies in the plane $-z = 0$. We therefore match it with the outgoing shalfedge of sv 's twin that lies in plane $z = 0$. Traversing sv 's adjacency list in counter-clockwise order and the adjacent list of sv 's twin in clockwise order simultaneously, we obtain all the previous-next pairs.

4.6.3 Creation of Facets.

To resolve the nesting relationship of the boundary cycles of a halfacet, we can reuse the planar sweep line algorithm from Michael Seel [See01b]. From the discussion of the overlay in Section 4.1, we know that we only need the information which edge lies below each of the vertices to resolve the nesting relationship. Since we also do not need the sweep to create an output graph, most of the functions implementing the `SegmentOverlayOutput` concept are empty.

Our boundary cycles consist of shalfedges and shalfloops. For the sweep we reinterpret them as edge-uses, i.e., each shalfedge se is understood as the use of the halfedge from se 's center vertex to the center vertex of its successor shalfedge. Thus, for each shalfedge in a boundary cycle we create a segment between the locations of those vertices as an input for the sweep; for each shalfloop in a boundary cycle, we create the trivial segment of its center vertex.

With the sweep, we can process all halfacets lying in the same supporting plane at once. Also, we can conclude the nesting structure of a halfacet from its already processed twin. Consequently, it suffices to sort all shalfedges by their normalized oriented plane equation and perform one sweep per positively-oriented supporting plane.

4.6.4 Creation of Volumes.

Shells are identified with a graph traversal. As halfacets together with the shalfedges and shalfloops in their boundary cycles belong to exactly one shell, we can traverse facet and sface cycles to obtain further elements of the same shell. Starting at any sface or halfacet this method yields all items of the shell if there is no edge whose removal divides the shell into two separate parts. This situation is handled by traversing from an isolated svertex to its twin svertex.

Looking at our example with the symmetric difference of two cubes, we traverse the outer shell starting from the outer sface sf at vertex $(0, 0, 0)$ as follows.

```

class Smallest_vertex_visitor {
    bool first;
    Vertex_const_handle v_min;

public:
    Shell_explorer() : first(true) {}

    void visit(Vertex_const_handle v) {
        if(first ||
            CGAL::lexicographically_xyz_smaller(v->point(),
                                                v_min->point())) {
            v_min = v;
            first=false;
        }
    }

    void visit(Halfedge_const_handle e) {}
    void visit(Halffacet_const_handle f) {}
    void visit(SHalfedge_const_handle se) {}
    void visit(SHalfloop_const_handle sl) {}
    void visit(SFace_const_handle sf) {}

    Vertex_const_handle get_result() { return v_min; }
};

Vertex_const_handle get_smallest_vertex(const Nef_polyhedron& N,
                                       Shell_entry_const_iterator it) {
    Smallest_vertex_visitor S;
    N.visit_shell_objects(SFace_const_handle(it),S);
    return S.get_result();
}

```

Figure 4.14: The class `Smallest_vertex_visitor` implements a `Shell_visitor`, which obtains the smallest vertex of a shell. The function `get_smallest_vertex` starts the shell exploration and forwards the result.

We first traverse the only sface cycle of sf and find the three outwards oriented halffacets adjacent to vertex $(0,0,0)$. Traversing the boundary cycles of these halffacets yields the proper sfaces of the vertices $(0,2,0)$, $(0,2,2)$, $(0,0,2)$, $(2,0,0)$, $(2,2,0)$, $(0,0,2)$, $(1,0,2)$, $(1,0,1)$, and $(2,0,1)$. Going on obviously yields the remaining sfaces and halffacets of the shell.

The traversal is implemented using the visitor pattern [GHJV95], i.e., we offer a function `visit_shell_objects` which traverses the shell and reports each found item to a given `Shell_visitor`. The `Shell_visitor` concept specifies six visit functions for reporting the six item types that constitute shells: vertex, halfedge, halffacet, shalfedge, shalfloop, and sface. This way, we decouple the traversal of the shell from operations performed on the items. Figure 4.14 illustrates the usage of `visit_shell_objects` by an example function that calculates the lexicographically smallest vertex of a shell.

Similar to facet cycles, we distinguish between outer and inner shells. Each volume, except for the outermost, is enclosed by the so-called outer shell. In a volume, there can be an arbitrary number of inner shells bounding holes in the volume. To identify whether a shell S is an outer or inner shell, we locate the spherical point with direction $(-1,0,0)$ on the sphere map of the lexicographically smallest vertex v_s in S . If the query returns an svertex or and shalfedge, S is an inner shell, because the found svertex or shalfedge indicates the existence of a vertex smaller than v_s on the adjacent outer shell. If the query returns an sface, we check whether the sface belongs to S . If it belongs to S , S is an outer shell. Otherwise, it is an inner shell.

The resolving of the nesting structure is similar to the resolving of facet cycles in the planar overlay. The idea is to move from shell to shell until we find a shell S_e that encloses the shell S_s we started from. S_e also encloses all other shells we came across in our walk from S_s to S_e . To direct our walk, we determine the shell below the smallest vertex of the current shell by a ray shooting query in $-x$ direction. As a consequence, the smallest vertex of the successor shell is always lexicographically smaller than the smallest vertex of the current shell. Thus, the walk will eventually find an enclosing shell, or no more further shell. In the latter case, all shells found during the walk are not enclosed by any shell.

The set-selection mark of a volume can be obtained from an arbitrary sface of any shell bounding the volume. We take it from the smallest vertex of the enclosing shell. This concludes the assembly of the selective Nef complex.

4.7 Unary Operations

Because the result of a unary operation is always a simplification of the input, i.e., the latter one can be obtained from the first by uniting and deleting items, we want to copy and simplify the input rather than constructing a new polyhedron from its sphere maps. Thus, we realize a unary function by applying the selection function on the sphere map items and on the SNC items. With the new marks, the SNC already represents the result, but potentially with redundant sphere maps, edges, facets, and volumes. We identify and simplify the following situations in the listed order:

1. Identify redundant facet f that separates two volumes c_1 and c_2 , with f , c_1 , and c_2 having the same marks. c_1 and c_2 may be equal, if f is surrounded by the volume.
⇒ Delete f . Unite c_1 and c_2 if necessary.
2. Identify redundant edge e that separates two facets f_1 and f_2 , with e , f_1 and f_2 having the same marks. f_1 and f_2 can be equal, if e is surrounded by the facet.
⇒ Delete e . Unite f_1 and f_2 if necessary.
3. Identify redundant vertex v that separates two collinear edges e_1 and e_2 . v , e_1 and e_2 have the same marks.
⇒ Delete v . Unite e_1 and e_2 .
4. Identify redundant vertex v without incident edges that lies in a facet with the same mark.
⇒ Delete v .
5. Identify redundant vertex v without incident edges and facets that lies in a volume with the same mark.
⇒ Delete v .

Similar to the simplification on the sphere as described in Section 4.5, updating the SNC with every single simplification can lead to a quadratic runtime. When we delete a sphere map isolated on a facet f , we have to delete the corresponding facet cycle entry stored with f . When we delete a sphere map isolated in a volume c , we have to delete the corresponding shell entry stored with c . Again, we use a union-find data structure [CLR90] to efficiently update the pointers. Separate union-find structures are needed for the merge of volumes, facets, and sfaces.

Chapter 5

Search Data Structures

The main bottlenecks of our first implementation were point location, ray shooting and intersection finding. We implemented each task as a trivial brute force algorithm. With these algorithms, each point location, ray shooting and intersection query has complexity linear in the size of the queried polyhedron in the worst case. Summing up all queries during a binary operation, we spent $O(nm)$ time for point location queries, where n and m is the complexity of the input polyhedra, and $O(k^2)$ time for the ray shooting queries, where k is the complexity of the result polyhedron. To get rid of these bottlenecks, we implemented two search data structures: a kd-tree and fast box intersection. We describe the concepts in the following sections. A detailed discussion of the runtime and space complexity is given in Chapter 7. We confirm the efficiency of both structures by experiments in Section 9.3.

5.1 Kd-tree

As the last step of the synthesis, we resolve the nesting structure of the shells. We do this by shooting a ray from the lexicographically smallest vertex of each shell in negative x -direction and obtain the boundary item that is hit first by the ray. A brute force algorithm tests each boundary item for intersection with the ray and reports the item intersecting it first. The combined complexity of testing a ray for intersection with all vertices, edges and facets of a polyhedron is linear in the size of the polyhedron.

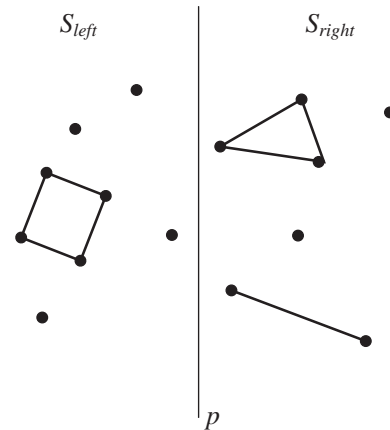
During the binary operation, we calculate the sphere maps of the result polyhedron. For this purpose, we deduce a set of candidate locations of vertices in the

result polyhedron, and then obtain a sphere map for each of these locations in both input polyhedron. We can easily determine the sphere map of a location l from the item that is located at l . We do not know the item at l , but we can identify it by a point location query. The brute force method solves point location by testing first whether l is on a vertex, is part of the relative interior of an edge or facet, or lies in a volume. In the latter case, we obtain the volume from the incidence relations of the item hit first by a ray shot from l in an arbitrary direction. The complexity is essentially the same as for a ray shooting query.

Both query types can be solved more efficiently with the help of a kd-tree. A kd-tree correlates with a decomposition of the space into full-dimensional regions. It stores the information which items can be found (at least partly) in each of the regions.

Having a kd-tree we need not consider all items in each ray shooting and each point location query. In a ray shooting query, it suffices to consider the items of the regions intersected by the ray. Furthermore, the kd-tree provides a rough order of the items along the ray. Since we search for the first hit, we need not check all items of the regions intersected by the ray. After the first intersection, it suffices to test the remaining items of the currently inspected region. In a point location query, we only consider items close to the queried location, i.e., only items of a region which contains the queried location. Miguel Granados implemented a kd-tree, which efficiently solves both query types on three-dimensional Nef polyhedra.

A *Binary Space Partitioning* (BSP) is a spatial subdivision of a k -dimensional space D into disjoint regions. Given a set of geometric objects S , D is iteratively split by hyper-planes into subregions until each subregion only contains a constant-sized subset of S . During the subdivision a *BSP tree* is created as follows: We start by splitting S at hyper-plane p into subsets S_{left} and S_{right} . Objects intersecting p are put into both subsets. The hyper-plane p is stored with a new node, which becomes the root node of the kd-tree. The children of the root are the BSP trees of S_{left} and S_{right} with respect to the subregions of D to the left and to the right of p . The recursion ends with constant-sized sets of objects, which are stored as the leaf nodes of the tree.



The structure of the BSP tree correlates to the subdivision process. In detail, each node n of the BSP tree correlates to some subregion r_n of D , such that each

5.1. KD-TREE

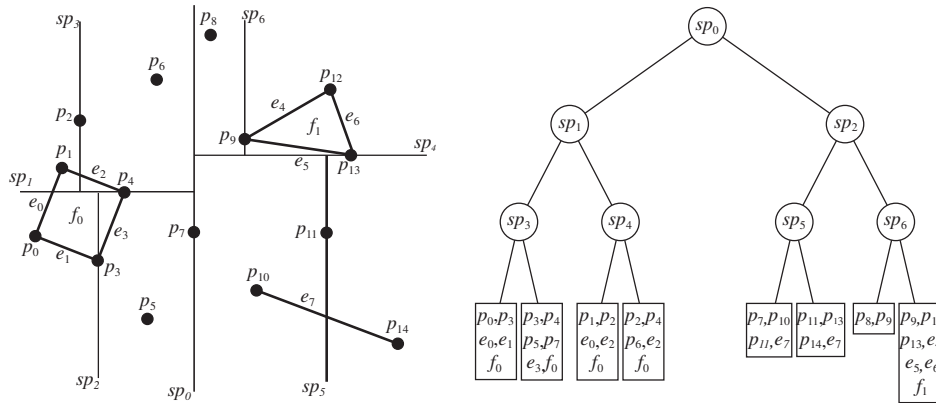


Figure 5.1: Two-dimensional kd-tree.

object that lies in or crosses r_n is stored in a leaf node of the subtree rooted at n . With the help of the splitting planes stored in the interior nodes, it is easy to obtain the object in the same region as some query point, or in the same regions that are crossed by a query edge, ray, or plane.

A *kd-tree* is an axis-aligned version of the BSP tree, i.e., the splitting planes are orthogonal to the coordinate axes in alternating order [Ben75, Sam90a, Sam90b]. As an example, Figure 5.1 shows a two-dimensional kd-tree. In the following discussion we concentrate on three-dimensional kd-trees.

To solve a point location query using a kd-tree, we only need to consider the objects in one leaf node, i.e., we consider the objects that at least partially lie in the same region r_l as the query location l . Restricted to a constant number of objects, we proceed similar to the brute force method. We check whether any of the objects has l in its relative interior. Otherwise l must be in the interior of a volume. To identify the volumes, we shoot a ray to the nearest vertex in the region—we will see later that there is at least one vertex in each region—and search for the first intersection of the ray with any object in the region. As the ray connects two points in a convex region, every object intersecting the ray between the two points must also lie in r_l .

For a ray shooting query, we have to consider the objects in every region crossed by the ray. We examine the regions in the order the ray traverses them. If we find an intersection, we only have to test the remaining objects of the current

region. Objects in upcoming regions are irrelevant, as they can only intersect the ray in some point that lies farer away from the source of the ray than the intersection already found.

During the construction of the kd-tree, there are different strategies for finding the splitting planes. For our purposes, we want to optimize the point-location queries rather than the ray-shooting queries. In a binary operation, one ray shooting query is posed per shell. In comparison, one point location query is posed per input vertex, per edge–edge, and per edge–facet intersection. As a result, ray shooting queries only consume a negligible amount of time compared to the time consumed by point location queries, which we will confirm in Section 9.3.

In our situation, a simple strategy for finding splitting planes applies well. To determine a splitting plane orthogonal to the x -axis for a set of objects S , we compute the median vertex v_m of the vertices in S with respect to the x -axis. Then, the splitting plane is the unique plane orthogonal to the x -axis with v_m in its interior. Finding splitting planes orthogonal to the y and z -axis works analogously. The division process terminates when a region contains at most two vertices. Consequently, the tree is limited to logarithmic depth, but we are not guaranteed to have only a constant number of edges and facets stored in the leafs.

The performance of both query types depends on the shape of the facets. A large facet with many vertices on the boundary may cross most or even all of the regions and thus are stored in most of the leafs. As a consequence, each point location query has to test for intersection with that facet. In another bad scenario many constant-sized facets each intersect most of the regions, such that the combined number of objects in the leafs of the kd-tree is quadratic compared to the complexity of the polyhedron. On the other side, we expect well-shaped facets most of the times, and as a result constant-sized leafs that allow efficient queries. We discuss the complexity of the kd-tree and its operations in the worst-case and under assumption of well-shaped polyhedra in Chapter 7. Also, we examine worst-case examples for the performance of the kd-tree in Section 9.3, and we discuss potential improvements of the kd-tree in order to handle bad scenarios as described above in Section 9.6.

5.2 Fast Box Intersection

To find all edge–edge and edge–facet intersections efficiently, we chose to implement fast box intersection as described in [ZE02]. This section summarizes their approach. Andreas Meyer conducted the implementation as a separate CGAL package.

5.2. FAST BOX INTERSECTION

Given two sets of n and m geometric objects, it is our goal to find all pairwise intersections between objects from different sets. The trivial method is the test of all pairs. The algorithm performs $n \cdot m$ intersection tests. The idea of box intersection is to put axis-aligned bounding boxes around each object and find the intersecting boxes. If two boxes intersect, we check whether the objects intersect, too. Replacing the intersection test of complicated objects by testing boxes can be much faster, especially since it suffices to use floating-point arithmetic for the coordinates of the bounding boxes. However, the biggest benefit is gained by using sophisticated algorithms for finding all pairwise intersecting boxes. In most domains of interest, the number of pairwise intersecting boxes k is in $O(n + m)$. Fast box intersection needs $O((n + m) \log^3(m + n) + k)$ time and $O(n + m)$ space to find all those boxes.

We use the following two properties of axis-aligned boxes to find intersecting box pairs efficiently:

Property 1 Boxes intersect if and only if they intersect in each dimension independently.

Property 2 Two intervals intersect if and only if one contains the low endpoint of the respective other.

If we test for the intersection of two axis-aligned boxes, Property 1 allows us to reduce this three-dimensional problem to three one-dimensional problems. The problem of deciding whether a given point lies in a given interval is known as the *stabbing problem*. With Property 2, it suffices to solve at most two stabbing problems for each of the three dimension. Instead of considering one box pair after the other, we rather apply *batched stabbing*, i.e., given points and intervals, we report for each point all intervals that contain it. Hence, we can find all intersecting boxes by applying batched stabbing six times. In each dimension, we test the lower endpoints of one set for intersection with the intervals of the respective other. Both sets are once considered as points and once as intervals.

In the following we describe several data structures and algorithms used for finding pairwise intersecting boxes by *batched stabbing* efficiently. At the end of the section we show how these techniques work together in a hybrid algorithm.

5.2.1 Segment Trees.

Figure 5.2 demonstrates the structure and the usage of segment trees. The segment tree is a balanced binary search tree. Each node represents a subinterval of the

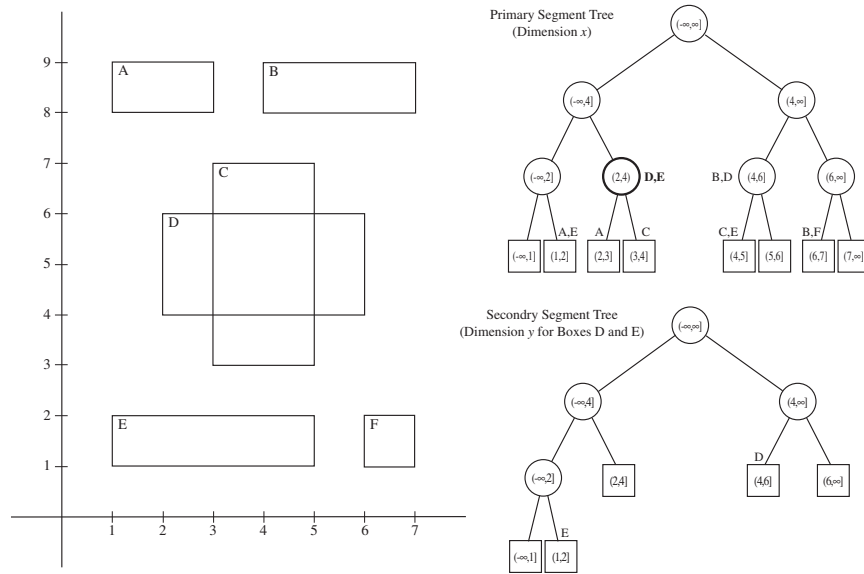


Figure 5.2: 2-level segment tree for a set of boxes in the plane.

number line, such that each level spans the whole number line and child nodes partition the interval of their common parent. In our example, the number line is partitioned by the endpoints of the given segments. The tree is then created as a balanced binary tree based on the intervals of the number line. The tree needs $O(n \log n)$ space and can be constructed in $O(n \log n)$ time.

The nodes of a segment tree store those segments that span the node's interval but not that of its parent. As a result, each segment is stored in at most $2 \log_2 n$ nodes. Posing a single stabbing query, we visit all nodes whose interval contains the query point and report the stored segments. Note that each reported segment contains the point and no segment is reported twice. The query takes $O(\log n + k)$ time, where k denotes the number of reported segments.

To use segment trees, we only consider the extent of a box in dimension d_i . This way, we reduce both sets of boxes to sets of segments in dimension d_i . Now, we find all pairwise intersections in dimension d_i of two sets of boxes as follows:

1. Build a segment tree for the first set of segments and query it with the left endpoints of the segments in the second set.
2. Build a segment tree for the second set of segments and query it with the left endpoints of the segments in the first set.

5.2. FAST BOX INTERSECTION

So far, we only have solved a one-dimensional problem. If we also want to resolve the other two dimensions, we can build secondary and tertiary trees. Secondary trees are built for each node of the primary tree. We reduce the boxes stored in a node to segments in the second dimension and build a segment tree from the resulting segments. From the boxes stored in the nodes of the secondary trees we then create tertiary trees. This approach uses $O(n \log_3 n)$ space and finds all intersecting boxes in $O((n+m) \log_3(n+m) + k)$ time.

5.2.2 Streaming.

The space requirements of multi-level segment trees are unsatisfactory. By applying the *streaming* technique [EO85] linear space requirements become sufficient. The solution is to perform all queries to the multi-level tree simultaneously. As a result, the tree is only traversed once in post-order and need not to be stored in memory. Each node is generated on demand and erased afterwards. Therefore only the $O(n+m)$ space for the boxes is needed. This technique is called streaming.

5.2.3 Scanning.

Segment trees are efficient but complex. Consequently, they have high hidden constants. Scanning is a much simpler method, which is faster for one-dimensional and small problems.

Given a set of n points P and a set of m segments S , we first sort both sets—the intervals are sorted by their low endpoints. Then we search for the first vertex p lying on the first segment s . From p onward we report all points until we find any that does not lie on s . We proceed the same way with the next segment s' , but begin our search for the first point on s' with p . The algorithm has running time $O(n \log n + m \log m + k')$. Neglecting the time for sorting, it runs in $O(n + m + k')$. For the one-dimensional case, the algorithm is very fast, but for the three-dimensional case, we have to check for all k' intersections found in the first dimension whether they also intersect in the second and third dimension. k' can be much bigger than the number of intersecting boxes k . The experiments in [ZE02] show that scanning is slightly faster for up to 200000 boxes. However, a hybrid algorithm combining streamed segment trees with scanning is essentially faster than both.

Algorithm 1 Hybrid algorithm for fast box intersection.

```
1: procedure HYBRID( $S, P, lo, hi, d$ )
2:   if  $S = \emptyset$  or  $P = \emptyset$  or  $hi < lo$  then
3:     return
4:   end if
5:   if  $d=1$  then one_way_scan( $S, P, 0$ )
6:   end if
7:   if  $|S| < c$  or  $|P| < c$  then modified_two_way_scan( $S, P, d$ )
8:   end if
9:    $S_m = \{i \in S \mid [lo, hi] \subseteq i\}$ 
10:  hybrid( $S_m, P, -\infty, +\infty, d-1$ )
11:  hybrid( $P, S_m, -\infty, +\infty, d-1$ )
12:   $mi = \text{approx\_median}(P)$ 
13:   $P_l = \{p \in P \mid p < mi\}$ 
14:   $S_l = \{i \in S - S_m \mid i \cap [lo, mi] \neq \emptyset\}$ 
15:  hybrid( $S_l, P_l, lo, mi, d$ )
16:   $P_r = \{p \in P \mid p \geq mi\}$ 
17:   $S_r = \{i \in S - S_m \mid i \cap [mi, hi] \neq \emptyset\}$ 
18:  hybrid( $S_r, P_r, mi, hi, d$ )
19: end procedure
```

5.2.4 The Hybrid Algorithm.

Algorithm 1 combines the methods and data structures described above. The function HYBRID computes all pairwise intersections of two d -dimensional boxes S and P , with the restriction, that in dimension d the boxes of S are only considered as segments, and the boxes of P are only considered as points. In the course of the execution of HYBRID, subsets of S and P are considered in both ways, but S and P themselves are not reinterpreted. For this reason, the function needs to be called twice. As an example, we use the following calls to compute all pairwise intersections of two sets of three-dimensional boxes B_1 and B_2 :

HYBRID($B_1, B_2, -\infty, +\infty, 3$)

HYBRID($B_2, B_1, -\infty, +\infty, 3$)

Neglecting the lines 2–8, the algorithm streams a multi-level segment tree. There are two types of recursions, which are used in such a way, that the each invocation of HYBRID uniquely correlates to a node of the segment tree. The recursive calls in lines 10 and 11 trigger the construction of the next tree level for those intervals S_m that completely cover the currently considered interval $[lo, hi)$, but do not cover the interval considered in the parent node. One of these two calls considers S_m as a set of segments and P as a set of points, while the other call considers them the other way. The recursive call in lines 15 and 18 create the two child nodes of the current node. For this purpose, the sets S and P , and the interval $[lo, hi)$ are split at mi into two halves, where mi is an approximation of median of the points in P .

There are three situations that stop the streaming of the segment tree and therefore terminate the recursion. Line 3 interrupts when there is nothing left to process, i.e., either of the sets S and P is empty, or the considered interval is empty. Lines 5 and 7 replace some part of the segment tree by scanning. In detail, line 5 triggers a one-way scan as described above as a replacement for the final level of the tree, and line 7 prunes the tree for small-sized problems and exerts a two-way scan instead. The two-way scan processes two one-way scans at once, i.e., it alternately considers the elements of both sets as points and segments. Each turn the smallest unprocessed segment s from both sets is processed. If s is in the first set, the second set is viewed as points, and vice versa. Both scanning routines can finally discover and report those boxes that intersect in all dimensions.

Chapter 6

Additional Functionality

The data structures and the Boolean and topological operations are the core of our implementation of Nef polyhedra in three-dimensional space. In this chapter we present the remaining functionality provided by our CGAL package.

6.1 Constructors, Input and Output

The class `Nef_polyhedron_3` provides three constructors. The first one creates a polyhedron that comprises a single volume. The constructor has a Boolean parameter that decides whether it creates the empty set or a polyhedron covering the whole three-space.

The second constructor creates an open or closed half-space. Its first parameter is a plane, which defines the boundary of the half-space. The orthogonal vector of the plane points to the outside. The second parameter decides whether the half-space is open or closed. Since infinitely bounded polyhedra can only be handled by extended kernels, this constructor is not provided in combination with a standard kernel.

Finally, we provide a constructor for manifold solids. The solid is passed as an instance of the CGAL class `Polyhedron_3`. This class comes with an input operator for the Object File Format (OFF), with file extension `.off`, which is also understood by GeomView [Phi96]. OFF files represent surfaces as a set of facets. Each facet is a list of indices pointing into a set of vertices. Vertices are represented as coordinate triples. `Polyhedron_3` restricts the format to orientable two-manifold solids with or without borders. Isolated edges and vertices are not allowed. Therefore, the smallest representable surface is a triangle; the smallest

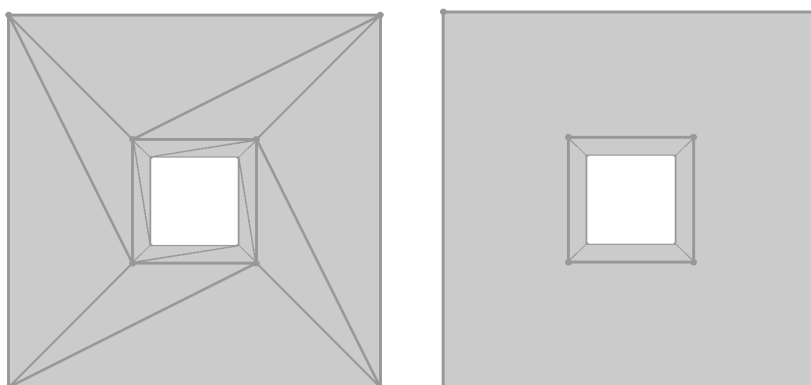


Figure 6.1: The polyhedron on the left side can be represented as a `Polyhedron_3` and as a `Nef_polyhedron_3`. In the latter case, the simplification routine unites coplanar facets. As a result, the front side becomes one facet with a hole. Hence, the polyhedron cannot be converted from `Nef_polyhedron_3` to `Polyhedron_3`.

representable surface without borders is a tetrahedron. We convert `Polyhedron_3` into `Nef_polyhedron_3` only if the surface can be turned into a manifold solid. Because of this, the constructor of `Nef_polyhedron_3` only accepts polyhedral surfaces without boundary. The surface then is converted into a closed solid by marking the surface and the volume enclosed by the surface.

A `Nef_polyhedron_3` can also be converted back into a `Polyhedron_3`, if each shell comprises a two-manifold surface whose facets do not have holes. The function `is_simple` is provided as a means to check whether the conversion is allowed. `Polyhedron_3` again provides output operators for writing the formats OFF, OpenInventor (.iv) [Wer94], VRML 1.0 and 2.0 (.wrl) [BPP95, VRM96, HW96], and Wavefront Advanced Visualizer object format (.obj).

Note, that it is not guaranteed that conversions are reversible. Obviously, the conversion from `Nef_polyhedron_3` to `Polyhedron_3` is more general than its reverse counterpart, as it allows the handling of multiple surfaces. Likewise not all conversions from `Polyhedron_3` to `Nef_polyhedron_3` are reversible. The reason lies in the unique representation of our data structures. As an example, Figure 6.1 shows a cube with a hole in the middle, represented by a triangulated mesh. The mesh is two-manifold and therefore can be converted to a `Nef polyhedron`. At the end of the conversion the SNC is simplified. As a part of the simplification process coplanar facets are united. In the example, all sides of the cube are coplanar. Thus, the triangles of each side are united and become a single facet. For

two sides this process results in a facet with a hole. Since `Polyhedron_3` does not allow facets with holes, we cannot support a conversion. To fix this problem, the facets with holes must either be decomposed into smaller facets without holes, or additional data for reconstructing the original facets must be stored. The latter procedure is in general too costly in relation to its benefit to be supported automatically. The former one is planned for future releases, but cannot guarantee a one-to-one reconstruction of the original surface.

`Nef_polyhedron_3` is also equipped with an input and an output operator for a proprietary file format. The file format includes the complete incidence structure, the geometric data, and the labels of each item. Because the output of the geometry and the labels is delegated to the respective output operators, the output depends upon the actual types of the geometric kernel and the labels. Thus, it is only possible to read a file, if the current geometric kernel and the current label type coincide with the types used during the creation of that file.

Looking at the output format of the geometric primitives, the output depends on the choice between homogeneous and Cartesian kernel, and on the used number type. Since extended kernels are realized by wrapping a polynomial class around the given number type, i.e., their real number type is not the given one but a polynomial with coefficients of that number type, this proposition also holds for the extended kernels. There is one exception to this behavior. The input and output operator bridge the difference between standard and extended kernels if a Nef polyhedron is finitely bounded. Using an extended kernel, the input operator can read a file based on standard geometry. Likewise, the output file of a finitely bounded Nef polyhedron is always formatted as though standard geometry was used, no matter if the used kernel is a standard or an extended kernel. If an extended kernel is used, the coordinates are converted from constants to constant polynomials and the items comprising the infimal box are added during the load operation of a file that contains a finitely bounded polyhedron. If a finitely bounded polyhedron is written, the coordinates are converted in the opposite direction and the infimal box is removed.

As a supplementary feature, the output operator can create a standardized output. Nef polyhedra imply the nice property that they can always be represented uniquely. As a consequence, it is possible to compare two Nef polyhedra for equality by a standardized output. Another way to perform the comparison is by a symmetric difference. The symmetric difference of two polyhedra equals the empty point set, if and only if the two polyhedra are equal. For our test suite, the first method is particularly useful, because using a standardized output is faster than performing a binary operation. Also, we can check the result of a binary operation without performing another.

6.2 Transformation

We support the following transformations on 3D Nef polyhedra: translations, scalings, and rotations by rational rotation matrices. Translations and scalings can be solved easily. It is sufficient to apply the transformation matrix to the points stored with the vertices, the planes stored with facets, and the splitting plane of the kd-tree. The geometry stored with the items in the sphere maps does not change, because the geometry of the items in a sphere map always relates to the coordinate system of the respective sphere.

Rotations are more complex for three reasons. First, the geometry of the sphere maps changes, too. Second, the kd-tree has to be recomputed, because the splitting planes would not be parallel to the coordinate planes after the rotation. Finally, a rotation changes the intersection of the polyhedron with the infimal box and must be recomputed. We start by computing the segments of intersection between the infimal box and the facets that intersect the infimal box. Then, we construct proper sphere maps for the endpoints of the segments. Since there is only one standard line that supports such an endpoint, the sphere maps have a certain structure. Still, they can be arbitrary large, because multiple facets may intersect the endpoint. When we have all sphere maps, we recompute the selective Nef complex, which also includes the recomputation of the kd-tree. Consequently, rotations are expensive operations, especially when an extended kernel is used.

α	runtime [s]	α	runtime [s]
10^{-1}	0.01	10^{-4}	4.47
10^{-2}	0.04	10^{-5}	44.89
10^{-3}	0.43	10^{-6}	450.56

Table 6.1: Runtime of the CGAL function `rational_rotation_approximation` to compute an exact rotation for the approximated angle α in degrees with the tolerance set to $\frac{\alpha}{10000}$.

Another problem with rotations is the computation of rotation matrices. Since $\sin \alpha$ and $\cos \alpha$ are irrational numbers in general, there currently is no practical possibility to perform a rotation of exactly α degrees. One solution is to approximate both values by rationals. But in addition to rotating by an approximation of the specified angle, the method often has another side effect. Approximating $\sin \alpha$ and $\cos \alpha$ separately usually introduces a scaling or a shearing of the rotated objects, if $\sin^2(\alpha) + \cos^2(\alpha) \neq 1$. CGAL provides a function `rational_rotation_approximation`, which returns exact values for $\sin(\alpha')$ and $\cos(\alpha')$ such that $\sin^2(\alpha') + \cos^2(\alpha') = 1$ and $|\alpha - \alpha'| < \varepsilon$ for a small speci-

6.3. VISUALIZATION

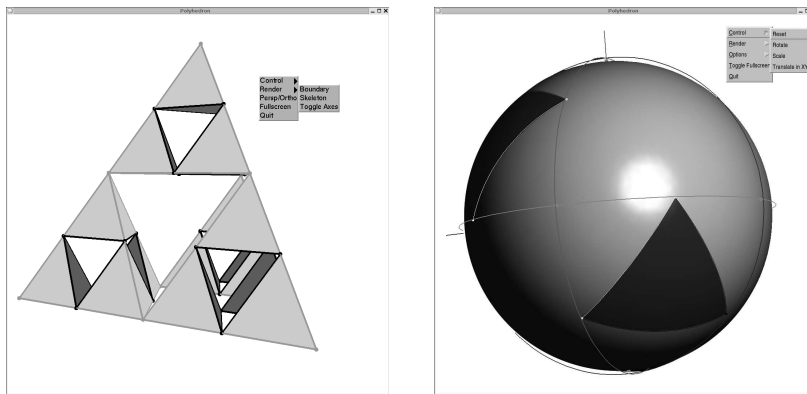


Figure 6.2: QT widgets for visualization of 3D Nef polyhedra and sphere maps.

fied $\varepsilon > 0$. The implementation offered by CGAL is based on Farey sequences as described in [CDR92]. It is division free but slower than the algorithm described in [CDR92]. The runtime for finding such an exact rotation matrix amounts to a non-negligible fraction for small ε , as can be seen in Table 6.1. The computations were performed on a computer with a 846 MHz Pentium III laptop with 256 MB RAM.

6.3 Visualization

We provide visualization via QT [Tro, BS04]—a cross-platform application development framework best known for its support of graphical user interfaces. In particular, we offer QT widgets for visualization of 3D Nef polyhedra and of 2D Nef polyhedra embedded on the sphere. The latter can also be used for sphere maps. Using the mouse and modifier keys, a visualized 3D Nef polyhedron can be translated in each direction, rotated and scaled. For spherical Nef polyhedra rotation and scaling is sufficient. Additional functionality is offered via context menus, such as different viewing modes or the displaying coordinate axes. Using inheritance, users can derive new classes from the widgets in order to include their own functionality. Figure 6.2 shows a snapshot of both widgets.

Chapter 7

Complexity

Amongst others, the complexity of most of our functions are essentially determined by the complexity of the point location, the ray shooting, and the intersection tests. Since we realized those subroutines with heuristic search data structures, the worst-case complexity deviates strongly from the expected runtime behavior. For this reason, we give two analyses of each of the two search structures. In addition to the worst-case analysis, we also give an analysis of the complexity expected under a number of heuristic assumptions in Section 7.1 and 7.2. In Section 7.3 we include the complexities of the heuristic search data structures into the total complexity of the major functions provided by our package.

Note that although we speak of expected runtime, we do not perform an average-case analysis. Usually in computer science, the terms expected runtime and average-case runtime refer to an analysis of the runtime that considers all inputs and the probability of their occurrence. Such an analysis seems misplaced for an algorithm that operates on complex geometric objects. It is unclear how we can argue about all possible Nef polyhedra and the likelihood of their occurrence. Instead, we argue that polyhedra used in practical applications are often well-shaped. As a result, we can exclude extreme situations and can therefore expect a better runtime than in the worst-case.

Let the total complexity of a Nef polyhedron be the number of vertices, edges, and sedges. Obviously, the number of all other items is not larger by more than a constant factor: Every sface is bounded by at least three shalfedges or one shalfloop, while there is one vertex for each sloop. Each facet is bounded by multiple shalfedges, and each volume (except for the outer volume) is bounded by at least one shell, which consists of multiple vertices, edges, and facets.

operation	worst-case runtime	expected runtime
kd-tree construction	$O(n^2)$	$O(n \log n)$
point location (single query)	$O(n)$	$O(\log n)$
ray shooting (single query)	$O(n \sqrt[3]{n} \log n)$	$O(\sqrt[3]{n} \log n)$
box-intersection	$O(n^2)$	$O(n \log^3(n) + s)$

Table 7.1: Worst-case and expected complexity of box-intersection and kd-tree based operations, where n denotes the input complexity, and s denotes the number of pairwise intersecting boxes found during box-intersection.

7.1 Kd-Tree

We chose to implement a kd-tree for the ray-shooting and the point-location queries [Ben75, Hav00]. During the construction of the kd-tree we use the vertex set as a criterion for finding proper splitting planes; we split the vertex set along alternating axes at its median vertex. The recursive subdivision ends when at most two vertices are left, which guarantees logarithmic depth. In the leafs, we store all vertices, edges and facets that intersect the corresponding region. Consequently, long edges and large facets can be stored in up to $O(n)$ leafs, and there might exist leafs with $O(n)$ items. The tight worst-case space bound is $\Theta(n^2)$.

Large facets may be cut by each splitting plane. In the worst-case, testing for the intersection of a facet with a splitting plane needs time linear in the size of the facet. Thus, constructing a kd-tree from an object with a linear sized facet that intersects all splitting planes implies linear time at each inner node of the kd-tree. The tight worst-case runtime of the kd-tree construction is $\Theta(n^2)$.

As explained earlier, we restrict ourselves to ray shooting in vertical direction. To be more precise, we only shoot rays parallel to the x-axis in negative direction. Hence, a ray intersects at most $O(\sqrt[3]{n})$ kd-tree regions. However, each region can store $O(n)$ items, and we need logarithmic search time for locating a neighboring region in our walk through the kd-tree. In total, we get a worst case runtime of $O(n \sqrt[3]{n} \log n)$ for vertical ray shooting.

For point-location queries, we find the containing region in $O(\log n)$, but might be forced to check against $O(n)$ items in that region.

Of course we use the kd-tree since we expect it to perform much better in practice. The usual heuristic assumption is a well-shaped geometry with the following consequences: Each edge or facet is stored in a constant number of regions and each region contains only a constant number of items. It suffices if these assumptions hold in an amortized sense, such that we get a linear storage size of the tree

7.2. BOX-INTERSECTION ALGORITHM

operation	worst-case runtime	expected runtime
construction of half-space		$O(1)$
construction from orientable two-manifold	$O(n^2)$	$O(n \log n)$
complement		$O(n)$
boundary, interior, closure, regularization	$O(n^2)$	$O(n \log n)$
translation, scaling		$O(n)$
rotation	$O(n^2)$	$O(n \log n)$

Table 7.2: Worst-case and expected complexity of unary operations, where n denotes the complexity of the polyhedron. See Table 7.3 for the binary operations. The expected runtime is given under the heuristic assumptions described in Section 7.1 and 7.2.

with $O(n \log n)$ construction time, an efficient ray-shooting query in $O(\sqrt[3]{n} \log n)$ time, and an efficient point-location query in $O(\log n)$ time. Both, the worst-case and the expected runtimes of all kd-tree related algorithms are summarized in Table 7.1. We study them experimentally in Section 9.2 and Section 9.3.

7.2 Box-Intersection Algorithm

We use the fast box-intersection algorithm described as streamed segment tree in [ZE02] to compute the edge-edge and edge-facet intersections. It runs in $O(n \log^3(n) + s)$ time, where n is the total number of boxes of both input sequences and s is the output complexity, i.e., the number of pairwise intersecting boxes. As a heuristic, the box-intersection algorithm assumes that bounding boxes approximate edges and facets well. If they do not, s can become as bad as $O(n^2)$, even though the edge-edge and edge-facet intersections might not reach that worst case themselves. However, we expect s to be close to the true output complexity of the edge-edge and edge-facet intersection problem.

7.3 Total Complexity

Given the sphere map representation for a polyhedron of complexity n , the synthesis of the SNC is dominated by sorting the Plücker coordinates, the plane sweep for the facet cycles and the shell classification. The latter task is solved by shooting a ray to identify the nesting relationship of the shells, so here we account also for

operation	worst-case runtime	expected runtime
point location (total)	$O(nm)$	$O(n \log m + m \log n)$
box intersection	$O(nm)$	$O((n+m) \log^3(n+m) + k)$
sphere sweeps	$O((n+m+k) \log(n+m))$	
synthesizing edges		$O(k \log k)$
plane sweeps		$O(k \log k)$
kd-tree construction	$O(k^2)$	$O(k \log k)$
ray shooting (total)	$O(k^2 \sqrt[3]{k} \log k)$	$O(c \sqrt[3]{k} \log k)$
binary operation	$O((n+m+k) \log(n+m) + nm + k^2 \sqrt[3]{k} \log k)$	$O((n+m) \log^3(n+m) + k \log(n+m) + c \sqrt[3]{k} \log k)$

Table 7.3: Worst-case and expected complexity of the major subroutines of the binary operation, where n and m denote the complexity of the input objects, k is the complexity of the result object, and c is the number of shells in the result object. The expected runtime is given under the heuristic assumptions described in Section 7.1 and 7.2.

the construction of the kd-tree. The synthesis runs in expected $O(c \sqrt[3]{n} \cdot \log n)$ time, where c is the number of shells in the result polyhedron. If there are most $O(n^{\frac{2}{3}})$ many shells in the result polyhedron, the expected runtime drops to $O(n \log n)$. This is also the cost for constructing a polyhedron from an orientable two-manifold solid.

Given a polyhedron of complexity n , the complement runs in linear time. The topological operations *closure*, *boundary*, *interior*, *exterior*, and *regularization* require a simplification step and run in $O(n \cdot \alpha(n))$ worst-case time, where $\alpha(n)$ denotes the inverse Ackermann function from the union-find structures in the simplification algorithm. However, we need to update the kd-tree afterwards, either the expected $O(n \log n)$ time or the $O(n^2)$ worst-case time. The affine transformations *translation*, *scaling*, and *rotation* also run in linear time. A kd-tree reconstruction is needed only after a rotation. Table 7.2 summarizes the complexities.

Given two polyhedra of complexity n and m , respectively, the Boolean set operation with a result of complexity k has a runtime that decomposes into four parts:

- (i) $O(n \log m + m \log n)$ expected time for the location of each vertex in the corresponding other input polyhedron.
- (ii) $O((n+m) \log^3(n+m) + k)$ expected time to find all edge-facet and edge-edge intersections. Here, we expect the number of intersections to be close to the number of pairwise intersecting boxes. Since each edge-edge and edge-facet

7.3. TOTAL COMPLEXITY

intersection corresponds to a vertex in the result polyhedron, the number of intersections is in $O(k)$.

- (iii) $O((n + m + k)\log(n + m))$ worst-case time for the overlay of all $n + m + k$ sphere maps.
- (iv) $O(c\sqrt[3]{k}\log k)$ expected time for the synthesis including the kd-tree construction. Table 7.3 gives an overview of the complexity of all major subroutines. It lists the total complexity of the binary operation.

The space complexity of our representation is clearly linear in the complexity of the Nef polyhedra, unless the kd-tree deteriorates as explained above.

Chapter 8

Software Design

We implemented two CGAL packages, *Nef_3* and *Nef_S2*, for 3D Nef polyhedra and Nef polyhedra embedded on the sphere, respectively. The design of the data structures extends design ideas presented by Kettner [Ket99], which were also used by Seel [See01b, See01a] for planar Nef polyhedra. The major goals of our software design are the following:

Flexibility: Nef polyhedra should work with various geometric kernels.

Extensibility: The functionality of Nef polyhedra shall be extensible via exchangeable items and labels.

Code reuse: To realize sphere maps, *Nef_3* shall reuse the code of *Nef_2* to a great extent. As a result, sphere maps shall be obtainable as a *Nef_polyhedron_S2*, such that functionality written for *Nef_S2* can be applied to sphere maps.

Each item type—vertex, halfedge, halffacet, volume, svertex, shalfedge, shalfloop, sface—is defined as a separate class. Those classes store the geometry and the combinatorial information of incidences (as CGAL handles), and they provide proper query and accessor functions for them. To be more precise, there are two implementations of the items shalfedge, shalfloop and sface, since 3D Nef polyhedra require additional incidences in comparison to planar Nef polyhedra embedded on the sphere. Similarly, a halfedge is the extended version of an svertex. Each class that realizes an item is parameterized with a template parameter that provides the const and non-const handle and iterator types of all items, the types of all necessary geometric primitives, like points and planes, and the label type. The

class definitions of the items of a 3D Nef polyhedron and a spherical Nef polyhedron are wrapped by outer classes `SNC_items` and `SM_items`, respectively. This way, the item types can be passed via a single template parameter.

```
class SNC_items {
    template <typename SNCTraits> class Vertex;
    template <typename SNCTraits> class Halfedge;
    template <typename SNCTraits> class Halffacet;
    template <typename SNCTraits> class Volume;
    template <typename SNCTraits> class SHalfedge;
    template <typename SNCTraits> class SHalfloop;
    template <typename SNCTraits> class SFace;
    ...
};

class SM_items {
    template <typename SMTraits> class SVertex;
    template <typename SMTraits> class SHalfedge;
    template <typename SMTraits> class SHalfloop;
    template <typename SMTraits> class SFace;
    ...
};
```

The class `SNC_structure` is a proper argument for the template parameter `SNCTraits`. Likewise, the class `Sphere_map` is a model for the `SMTraits`. Both classes include type definitions of all the handle and iterator types, the geometric objects, and the label type. Therefore, they themselves must be parameterized with the geometric kernel, the items, and the label type. In addition, both classes also constitute the representation layer of the respective data structure. They include a list for each of the item types. To be more specific, `Sphere_map` has three lists: one for all svertices, one for all shalfedges, and one for all sfaces. There is no need for a list of shalfloops, as there can be only two shalfloops or no shalfloop at all in a spherical Nef polyhedron. In a 3D Nef polyhedron, the items of the sphere maps are centrally stored in the `SNC_structure` for an easy iteration over all of them. Additionally, the items of a single sphere map are stored in consecutive order, such that the iteration over them also is simple and fast. The iterator ranges of the items of a sphere are stored with the center vertex of the sphere. Thus, `SNC_structure` maintains seven lists: for all vertices, halfedges (=svertices), halffacets, volumes, shalfedges, shalfloops, and sfaces. `Sphere_map` and `SNC_structure` provide interfaces for the proper creation and deletion of items.

Unfortunately, this design neither allows the reuse of the `Nef_S2` code in `Nef_3`, nor a function that returns a sphere map of a `Nef_polyhedron_3` as a `const Nef_polyhedron_S2`. We introduce a new type `SNC_sphere_map`, which is supposed to behave like the type `Sphere_map`, except that it does not manage the items of the represented sphere map itself, but delegates this task to `SNC_structure`. With such a class, most of the `Nef_S2` code can be reused. Furthermore, we add another template parameter to `Nef_polyhedron_S2` that allows us to exchange `Sphere_map` by `SNC_sphere_map`. Now, `Nef_polyhedron_3` can construct a `Nef_polyhedron_S2` from a `SNC_sphere_map`.

The vertex type already fulfills most of the requirements of a class `SNC_sphere_map`. However, it cannot be used without adaptation. On the other hand, we do not want to adapt the vertex type itself, since it is exchangeable by the user, and thus should only comprise few functionality that is interesting for users. Instead, we realize `SNC_sphere_map` as a new class derived from the vertex type. As mentioned above, the new class must realize the whole functionality of the class `Sphere_map`. Most of the functionality is already given by the vertex type. The remaining functionality is related to the management of items, which is delegated to the `SNC_structure`. Because we regularly access the `Nef_S2` code within `Nef_3`, we replace the list of vertices stored in `SNC_structure` by a list of `SNC_sphere_maps`.

The final class signature of this implementation layer looks as follows:

```
template <typename Kernel, typename Items, typename Label>
class SNC_structure {
    typedef SNC_structure<Kernel, Items, Label>    Self;
    typedef SNC_sphere_map<Kernel, Items, Label>  Sphere_map;

    list<typename Items::Sphere_map<Self> >    vertices;
    list<typename Items::Halfedge<Self> >      halfedges;
    list<typename Items::Halfacet<Self> >      halffacets;
    list<typename Items::Volume<Self> >        volumes;
    list<typename Items::SHalfedge<Self> >     shalfedges;
    list<typename Items::SHalfloop<Self> >     shalfloops;
    list<typename Items::SFace<Self> >        sfaces;
    ...
};

template <typename Kernel, typename Items, typename Label>
class Sphere_map {
```

```

typedef Sphere_map<Kernel, Items, Label> Self;

list<typename Items::SVertex<Self> >   svertices;
list<typename Items::SHalfedge<Self> >  shalfedges;
list<typename Items::SHalfloop<Self> >  shalfloops;
list<typename Items::SFace<Self> >     sfaces;
...
};

template <typename Kernel, typename Items, typename Label>
class SNC_sphere_map {
    typedef SNC_sphere_map<Kernel, Items, Label> Self;

    list<typename Items::Halfedge<Self> >   svertices;
    list<typename Items::SHalfedge<Self> >  shalfedges;
    list<typename Items::SHalfloop<Self> >  shalfloops;
    list<typename Items::SFace<Self> >     sfaces;
    ...
};

```

Finally, the main classes of the two packages look as follows. The class `Nef_polyhedron_3` has three template parameters, one for the geometric kernel, one for the items, and one for the label. As default we use the class `SNC_items` for the items and assign `bool` as the label type. Furthermore, `Nef_polyhedron_3` has a protected member variable of the type `SNC_structure` as its representation layer, which is parameterized with the same types as the class `Nef_polyhedron_3`.

The class `Nef_polyhedron_S2` also has template parameters for the geometric kernel, the items (with the default type `SM_items`), and the labels (with default type `bool`). Additionally, it has a fourth parameter for the type of the sphere map, which by default is the class `Sphere_map` parameterized with the same geometric kernel, items, and label type as `Nef_polyhedron_S2`. The representation layer of `Nef_polyhedron_S2` is realized by a protected member of the given sphere map type.

```

template <typename Kernel,
         typename Items=SNC_items,
         typename Label=bool>
class Nef_polyhedron_3 {
    typedef SNC_sphere_map<Kernel, Items, Label>

```

```

    Sphere_map;
    typedef Nef_polyhedron_S2<Kernel, Items, Label, Sphere_map>
        Nef_polyhedron_S2;

protected:
    SNC_structure<Kernel, Items, Label> snc;
    ...
};

template <typename Kernel,
          typename Items=SM_items,
          typename Label=bool,
          typename Map=Sphere_map<Kernel, Items, Label> >
class Nef_polyhedron_S2 {

protected:
    Map sm;
    ...
};

```

While the items provide accessor functions for the incidence structure, the geometry, and the labels, the main classes `Nef_polyhedron_3` and `Nef_polyhedron_S2` offer constructors, the Boolean and topological operations, transformations, point location, and entries to the incidence structure. The latter are iterator ranges for all vertices, halfedges, edges, halffacets, facets, volumes, shalfedges, sedges, shalfloops, sloops, and sfaces, and a function that initiates a shell traversal. The user interface is completed by an input and an output operator. As usual, those operations are global functions.

Chapter 9

Algorithm Engineering

Our first implementation realized all data structures and the most important functionality, but used simple brute-force algorithms for ray shooting, point location and intersection finding. The idea behind this approach is to obtain a well structured, running solution in relative short time, and check the correctness of our concepts early. Afterwards we can concentrate on optimizing the efficiency of the code. With this approach we followed a famous saying by Hoare:

*We should forget about small efficiencies, say about 97% of the time:
premature optimization is the root of all evil.* C. A. R. Hoare 1980

With the first implementation, a binary operation on a manifold solid with more than 1000 vertices could only be performed in several hours. Now that we have added fast heuristic search data structures and applied additional optimizations, our current version computes the symmetric differences of a hammerhead shark (2560 vertices and 5116 facets) and a translated copy of the same shark (the result has 6864 vertices and 11090 facets) in 41 second on a 846MHz Pentium III laptop with 256 MB RAM.

In Section 9.1 we introduce optimizations and confirm their impact with experiments. In Section 9.2 and Section 9.3 we perform further experiments to examine the general runtime behavior of our implementation, and to stress the main sub-routines with complex situations. In Section 9.4 we compare ourselves to ACIS R13, the newest version of a common professional CAD kernel. The comparison to ACIS gives a first impression of the advantages and disadvantages of exact arithmetic in comparison to floating-point arithmetic. In Section 9.5, we deepen

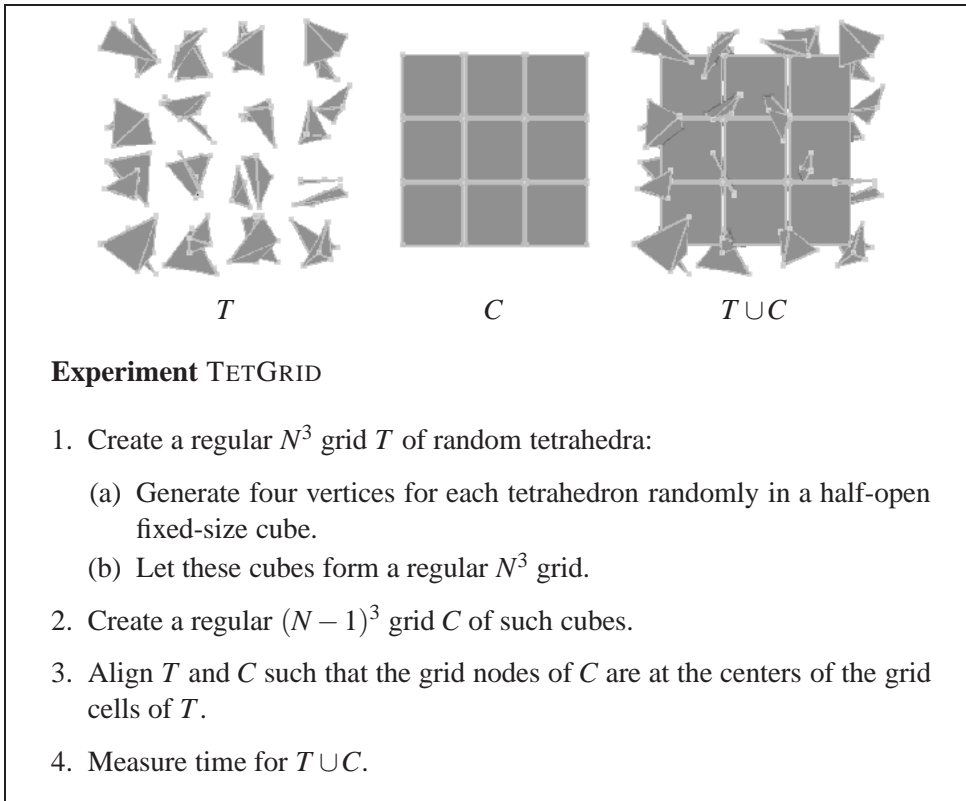
the comparison by examining the runtime behavior of our implementation in cascaded constructions. Finally, we summarize some results and discuss opportunities to remove weaknesses in Section 9.6.

The tests are performed on two different computers. Machine 1 has a 846 MHz Pentium III processor and 256MB RAM. It is used for tests that are later repeated with ACIS on the same computer. All other tests are measured on Machine 2, which has two 3 GHz Intel Xeon processors and 4GB RAM. When we performed the tests, we already applied some important corrections to our code as published in CGAL 3.1. We therefore performed the tests on CGAL 3.1., but exchanged the following packages with newer versions from internal release CGAL 3.2-I-???: planar Nef polyhedra, planar Nef polyhedra embedded on the sphere, 3D Nef polyhedra, and Box intersection. The test series were performed on a debian linux system and compiled with g++-3.3.4 and the options -O2 and -DNDEBUG. They were scheduled, run, and archived with the tool ExpLab [HPKS02]. The source code of the experiments, the test data, and the results are published for reference at <<http://www.mpi-inf.mpg.de/~hachenb/proj/Nef>> together with the proper versions of the exchanged packages.

9.1 Optimizations

In the following we introduce several optimizations of our implementation. We want to emphasize, that the code is not fully optimized. The presented improvements remove the most obvious bottlenecks, but there are still a couple of optimization opportunities left.

The benefit of the conducted improvements is confirmed by the TETGRID experiment, which unites two polyhedra of approximately equal size. We designed the scenario without aiming for special properties besides the size of the polyhedra. Also we wanted to allow all kinds of degeneracies without enforcing special ones. However, there are many collinear edges, coplanar facets, and the result has a high genus. As a reference, we perform a run of the TETGRID experiment ($N = 12$) with all optimizations activated. Input object T and C have 6912 and 10648 vertices, respectively. Their union has 43613 vertices and is computed in 56.8 seconds on machine 2. To examine the effect of some optimization o , we perform another run of the same scenario on the same machine but with o deactivated. We compare the runtimes of the union operation and of the interesting subroutines with the reference runtime.



9.1.1 Ray shooting and Point Location

The first optimization is the obvious step of replacing the trivial methods for point location, ray shooting and intersection finding with more sophisticated approach. The trivial implementations of these query types cause a quadratic runtime of binary operations. For point location and ray shooting queries, we add a kd-tree. As pointed out in Chapters 5 and 7, we expect clear improvement.

Table 9.1 shows the impact of the kd-tree on the ray shooting and point location queries. Its impact on the intersection finding is scrutinized in the next section together with the box intersection. With the kd-tree, the 17606 queries (17550 point location and 56 ray shooting queries) can be answered in less than 10 seconds instead of 8805 seconds. Adding the 6 seconds for the construction of the kd-tree, the time spent for operations on the kd-tree is only 0.18% of runtime needed for the same ray shooting and point location queries performed by the trivial algorithms. Repeating this comparison with $N = 1$, the ratio between the runtime of the two approaches becomes much smaller. In this test, the combined runtime of all kd-tree

	kd-tree	construction	point location	ray shooting	binary op.
$N = 12$	not used	0.00s	8704.68s	100.35s	8846.44s
	used	5.97s	9.16s	0.21s	56.8s
$N = 1$	not used	0.000s	0.033s	0.012s	0.067s
	used	0.004s	0.012s	0.001s	0.039s

Table 9.1: Experiment TETGRID: Comparison between the fully optimized binary operation and the binary operation without kd-tree support. Listed are the time spent for kd-tree construction, point location queries, ray shooting queries, and for the total runtime of the binary operation.

queries and the construction of the kd-tree is only 38% of the runtime spent by the trivial algorithms. Hence, for small instances the kd-tree is still significantly faster than the trivial method.

9.1.2 Intersection

As we already pointed out in Chapter 5, testing every single edge–edge and edge–facet pair for intersection is a very costly task. On the one hand, there can be a high number of these pairs, and on the other hand, intersection tests are very expensive, especially when large facets are involved. We have two heuristic search methods for fast intersection computation: a kd-tree and the box-intersection. Both return a set of candidate pairs, i.e., a set of edge–edge and edge–facet pairs that might possibly intersect. Of course, the heuristics must not overlook any intersecting pair, but are allowed to return too many. In consequence, it suffices to perform the intersection test on these candidate pairs. A good heuristic suggest only very few candidate pairs. We measure the quality of the heuristics by the time needed to

intersection method	candidate pairs	candidate pairs per intersection	runtime	
			search	total
trivial	358795008	11606	4579.56s	4631.87s
kd-tree	527113	17.05	13.66s	63.63s
box intersection	177177	5.73	4.49s	56.85s

Table 9.2: Experiment TETGRID with $N = 12$: Comparison between box intersection, intersection finding via kd-tree, and trivial intersection finding. Listed are the number of candidate pairs, the ratio between candidate pairs and the real intersections (30914), the time spent for finding all intersections, and the total runtime of the binary operation.

9.1. OPTIMIZATIONS

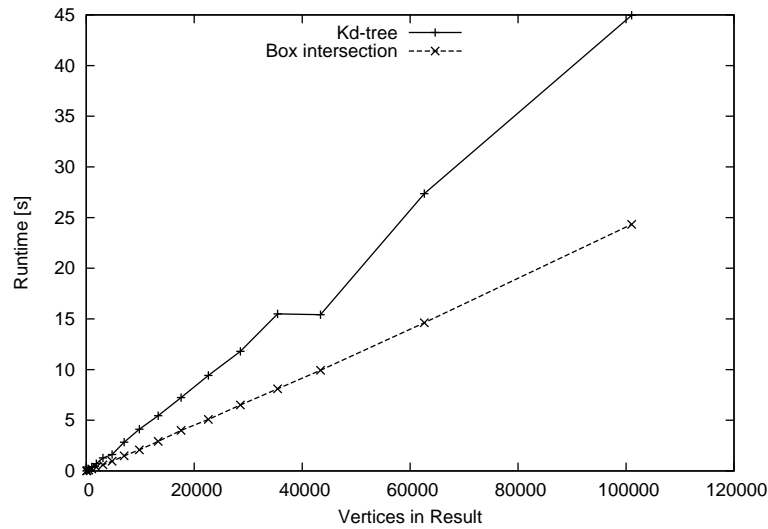


Figure 9.1: Experiment TETGRID with $N = 12$: Runtime comparison between the box-intersection algorithm and the kd-tree on the TETGRID experiment.

identify all intersections and the number of returned candidate pairs.

The box-intersection algorithm runs in $O(n \log^3(n) + s)$ time, where n is the number of boxes in the two input sequences and s is the number of pairwise intersections of boxes. Using the assumptions from Chapter 7 regarding the shape of the polyhedra, the expected complexity of obtaining all intersection candidates of some given edge from the kd-tree is $O(l \log n)$, where n is the total complexity of the polyhedron and l the number of cells crossed by the edge. Then we can express the complexity of all edge–facet and edge–edge intersections as $O(L \log n)$, where L is the sum of the l 's over all edges.

Considering the theoretic complexity, it is unclear which algorithm is more efficient, box intersection or the kd-tree. The kd-tree could win asymptotically for L is in $O(n \log^2(n))$. We expect that on average each edge traverses only a constant number of cells. On the other hand, the kd-tree tests the same candidate pairs several times, when an edge or facet is stored in several kd-tree cells. The complexity may not change, but the hidden constant factor might be higher than for the box intersection.

Table 9.2 shows that both heuristics are effective in comparison to the trivial method. Box-intersection leads in both quality measures; it runs faster and returns fewer candidates. But we want to take a closer look and perform a complete test series.

N	Input Complexity	candidate pairs / intersection	
		box intersection	kd-tree
2	172	6.64	13.33
4	1012	5.88	13.37
6	3100	5.78	14.28
8	7012	5.79	13.29
10	13324	5.76	13.74
12	22612	5.71	13.43
14	35452	5.79	14.07
17	62632	5.77	13.03
20	101044	5.79	13.61

Table 9.3: Experiment TETGRID with $N = 1, \dots, 20$: Number of candidate pairs tested per real intersection. The input complexity is the sum of the vertices in the two polyhedra.

Figure 9.1 shows the result of a test series with the TETGRID experiments for $N = 1, \dots, 20$. The box-intersection algorithm is clearly preferable. Table 9.3 supports this result. It lists the candidates per intersection ratio, which seems to stay constant. The kd-tree tests more (redundant) candidates. It always proposes about 13 candidates per intersection, while the box-intersection algorithm only suggests about 6 candidates per intersection.

9.1.3 Half-sphere Sweep

After we had exchanged the trivial methods for ray shooting, point location, and intersection finding with more sophisticated methods, we used the GNU profiler [gpr] to search for the main bottlenecks in our binary operation. We discovered that most of the runtime was spent in the sweep operations in all conducted experiments. The sweep-line algorithm is a powerful tool; we use it in the plane for resolving the nesting structure of the boundary cycles of facets, and we use it on half-spheres to compute the overlay of two sphere maps. Especially, the half-sphere sweep used more than 50% of the running time; it is called twice — once for each sphere map. However, the sweep is written to solve arbitrary complex overlays efficiently, while we use it for generic and therefore simple overlay situations most of the times.

As an example, we look at an edge–facet intersection. A sphere map sm_e of that models the local neighborhood of an edge e , has two oppositely oriented svertices and one halfloop for every facet incident to the edge from one svertex to the other. A sphere map sm_f that models the local neighborhood of a facet f only has an

9.1. OPTIMIZATIONS

sloop. The overlay of sm_e and sm_f combines those two simple structures, i.e., the sloop from sm_f is inserted into sm_e . Thereby it cuts each halfloop of sm_e into two parts. Likewise, the sloop is cut into several parts by the halfloops of sm_e . This construction has no degenerate situations, since e cannot lie in the supporting plane of f . Consequently, it is not necessary to apply an algorithm as powerful as the sweep-line algorithm to solve this overlay.

The following optimizations circumvent the execution of unnecessary halfsphere-sweeps, or replace them with simple solutions for specific situations.

- (i) Overlays for vertices located in a volume of the counterpart polyhedron and for edge–facet intersections are performed by hand, i.e., without the sweep-line algorithm.

In the first case, the vertex is cloned and the marks of the clone are deduced from the old marks, the mark of the volume and the Boolean function. Afterwards, the sphere map is simplified as usual. The simplification can be omitted if optimization (iii) is enabled.

In case of the edge–facet intersection, the resulting arrangement always has the same structure. Let e and f denote the edge and the facet participating in the intersection. Then the arrangement consists of several half-circles, i.e., one for each facet incident to e , which are all split by the plane supporting f . It is obvious how to compute the marks of the arrangement. The simplification on the sphere map is performed afterwards as usual.

As a result of this optimization, the sweep-line algorithm is only used in case of edge–edge intersections and when a vertex is located on a vertex, edge, or facet of the counterpart polyhedron. This means that all common situations are solved by specialized algorithms. The situations in which the sweep-line algorithm is still needed are only degenerate situations, which we argue that they occur in real world data sets as well, but then not very many of them.

- (ii) Our sphere sweep can process a half-sphere at once. Certain extra work has to be done to cut each sphere map in two halves and to paste the two resulting half-spheres back together. Since this includes cutting edges in two halves and introducing several equator edges, the two halves combined often have twice as many elements than the original sphere map. We therefore test, if all vertices and sedges of a sphere map either lie on the top, bottom, left, right, front, or back half-sphere. In such an instance, we need not cut the sphere into two halves. Instead it suffices to perform only one sweep on the relevant half of the sphere.

- (iii) For some vertices of the input polyhedra it is easy to determine that they will not appear in the resulting polyhedron. For instance, in a union operation every vertex of either polyhedron located in the inside of the other polyhedron is absorbed into this volume. Here, the selection routine assigns the same mark to each svertex, sedge and sface on the sphere map. This happens when the Boolean operation *bop* applied to the mark of the determined volume and any second mark always has the same result. Thus, if a vertex of the first polyhedron has been located in volume *c* of the second polyhedron, and $bop(true,mark(c))=bop(false,mark(c))$, then the vertex does not need to be considered.

Because two completely random polyhedra usually do not include degenerate situations, optimization (i) alone would reduce the number of sphere sweeps to zero in such a case. In the TETGRID experiment, the vertices are placed at locations with integer coordinates and the grid cells have dimensions $100 \times 100 \times 100$. Hence, there will probably be some remaining sphere sweeps in our test series.

Table 9.4 illustrates the benefit of the optimizations. The impact of optimization (i) is impressive. With this single optimization activated, 91072 of 96928 sphere sweeps are replaced by 30024 specialized edge–facet overlays and 15512 specialized vertex-in-volume overlays. Thereby, it reduces the runtime of the sweeps to 7% and the total runtime to 32%. Applying all three optimizations, we find further 811 sphere overlays that can be handled with sweeping only one half-sphere, and 568 vertex-in-volume overlays can be omitted. In total number of the executed sphere sweeps is reduced to 5%.

optimizations			number of sphere sweeps	runtime	
(i)	(ii)	(iii)		sphere sweeps	binary op.
-	-	-	96928	134.27s	227.23s
+	-	-	5856	10.88s	69.21s
-	+	-	80996	115.00s	205.27s
-	-	+	87792	124.42s	212.89s
+	+	-	5045	8.57s	55.21s
+	-	+	5856	10.06s	57.72s
-	+	+	76185	110.56s	195.54s
+	+	+	5045	8.74s	56.85s

Table 9.4: Experiment TETGRID with $N = 12$: Number of sphere sweeps performed, the runtime of all sweeps, and the complete binary operation are shown for runs with all combinations of enabled and disabled optimizations.

9.1. OPTIMIZATIONS

The sphere sweep is not an optimal solution for the overlay. Andreas Meyer, a student at the Max-Planck-Institut implemented two algorithms for planar overlay as part of his Master thesis. The first algorithm is by Finke and Hinrichs [FH95] and the second is a randomized incremental approach by Mulmuley [Mul90]. Both always perform considerably better than Seel's sweep based overlay implementation. In overlay computations with many intersections, Mulmuley even outperforms the sweep by a factor of up to 13. Mulmuley's method proved to be the most memory and time efficient of the three methods.

We still need to adapt the interface of the alternative overlay methods, such that they also work for spherical geometry. It is very likely, that both outperform Seel's sweep on the sphere, also. Especially, because they probably can process a full sphere at once. Obviously, optimizations (i) and (iii) still work with these overlay algorithms, too. If an alternative overlay method can process the full sphere at once, optimization (ii) becomes superfluous.

9.1.4 Plane Sweep

For the plane sweep, we use the same generic sweep-line algorithm as for the sphere sweep. Again, we try to avoid as many sweeps as possible. In the most general case we perform a sweep for every plane supporting a facet, but we need the sweep only if there is a hole in a facet. As described in Section 4.6, we determine whether a boundary cycle is an inner or outer facet cycle by an orientation test at its smallest vertex. A left turn indicates an outer, and a right turn an inner cycle. Since we need to check the orientation at the smallest vertex of each facet cycle anyway in order to link the facet cycles correctly as inner or outer cycles, the optimization includes no significant overhead. Table 9.5 shows the effect of this optimization.

Like for the sphere overlay we can replace the sweep with a faster method. The method by Mulmuley is applicable, again; the method by Finke and Hinrichs is not applicable.

optimization	number of plane sweeps	runtime	
		plane sweeps	binary op.
off	6867	17.10s	60.21s
on	343	13.40s	56.85s

Table 9.5: Experiment TETGRID with $N = 12$: Listed are the number of performed plane sweeps, the runtime of the plane sweeps, and the runtime of the binary operation in runs with and without the plane-sweep optimization.

9.2 General Runtime Behavior

In this and in the following section, we experimentally evaluate the runtime behavior of our implementation, in particular the binary Boolean operations. We have several experiments that support the expected runtime analyzed in Chapter 7, and we have designed experiments to stress our implementation with worst-case scenarios.

Besides the total runtime, we list also the runtime of important subroutines in the binary Boolean operation to illustrate the distribution of resources, potential bottlenecks, and further places for optimizations. We summarize the important subroutines here in their order of usage (see Chapter 4 for further explanations):

Point location: queries the kd-tree of the input polyhedra to locate the vertices of the respective other polyhedron.

Box-intersection: intersection finding on the bounding boxes of facets and edges. Includes the cost of the intersection test on the actual edge and facet geometry.

Sphere sweeps: sum of all sphere sweep-line algorithms performed during Boolean operations on sphere maps.

Synthesizing edges: in the synthesis step, sorts the line representation based on Plücker coordinates.

Plane sweeps: in the synthesis step, sorts facet boundary cycles of the result polyhedron.

Kd-tree construction: in the synthesis step, initializes the kd-tree for the result polyhedron.

Ray shooting: in the synthesis step, used to resolve the nesting of shells of the result polyhedron.

Others: all other parts not listed explicitly in the same graph, so parts which have no critical worst-case or no interesting practical runtime contributions.

9.2.1 Balanced Binary Operations

In our first test series, we want to examine the generic runtime behavior when the two input polyhedra and the result all have similar size. We capture these

9.2. GENERAL RUNTIME BEHAVIOR

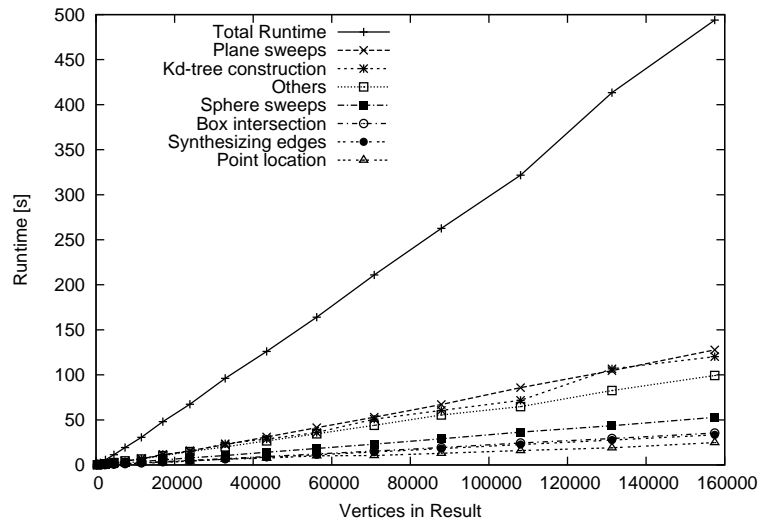


Figure 9.2: Total runtime and runtime distributed over the major subroutines for our implementation in the TETGRID experiment.

properties in the TETGRID experiment (see page 85). We measure its runtime for values $N = 3, \dots, 17$ on machine 1 for a later comparison with ACIS.

In Figure 9.2 we see the total runtime and the runtime distributed over the major subroutines. The plane sweeps and the construction of the kd-tree each comprise about a quarter of the total runtime. The total runtime looks linear in the size of the result. But since the construction of the kd-tree is $\Omega(k \log k)$, where k is the size of the result, the total runtime must have a logarithmic factor, too.

9.2.2 Binary Operation with Quadratic Result

In the next test series we again start with input objects of equal size, but we achieve a worst-case output complexity, as described in the QUADRATICWALLGRID experiment. We run this experiment for $N = i * 10, i = 1, \dots, 15$ on machine 2. We see in Figure 9.3 that the construction of the kd-tree is dominating the runtime.

In consideration of the results of the previous experiment, the results of the current experiment seems reasonable. The construction of the kd-tree already comprised a large part of the runtime in the TETGRID experiment. In the QUADRATICWALLGRID experiment it becomes even more dominating, because in contrast to other subroutines its runtime solely depends on the complexity of the result polyhedron. This argument also applies for the planar sweep. But since there is not a single facet with a hole in this scenario, the planar sweep is never executed.

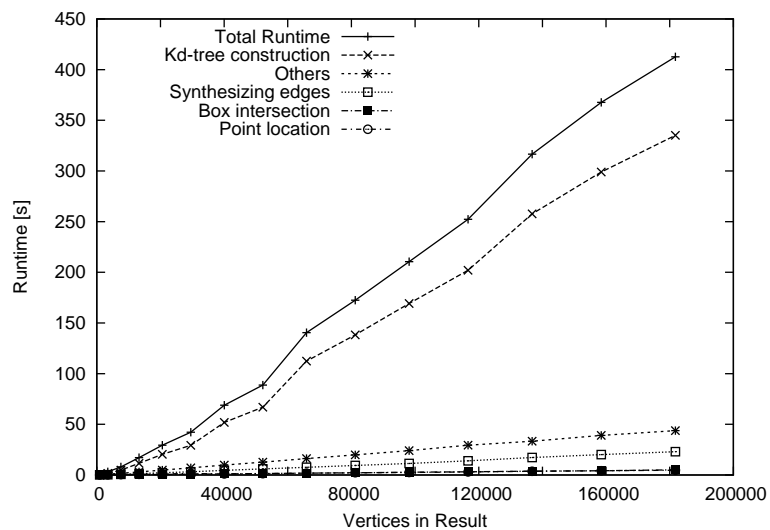
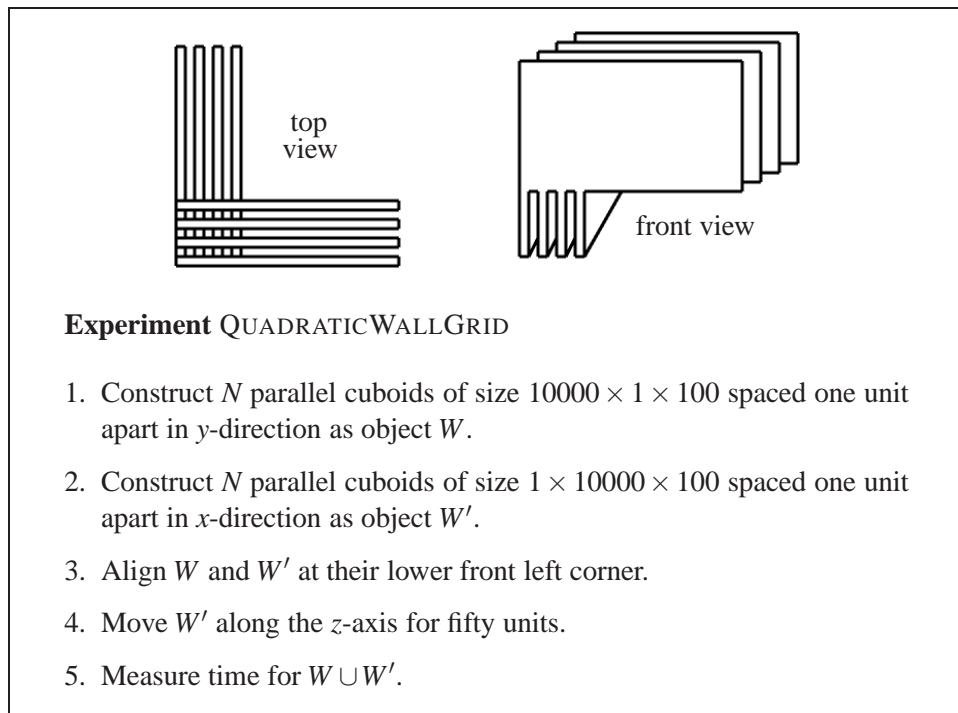
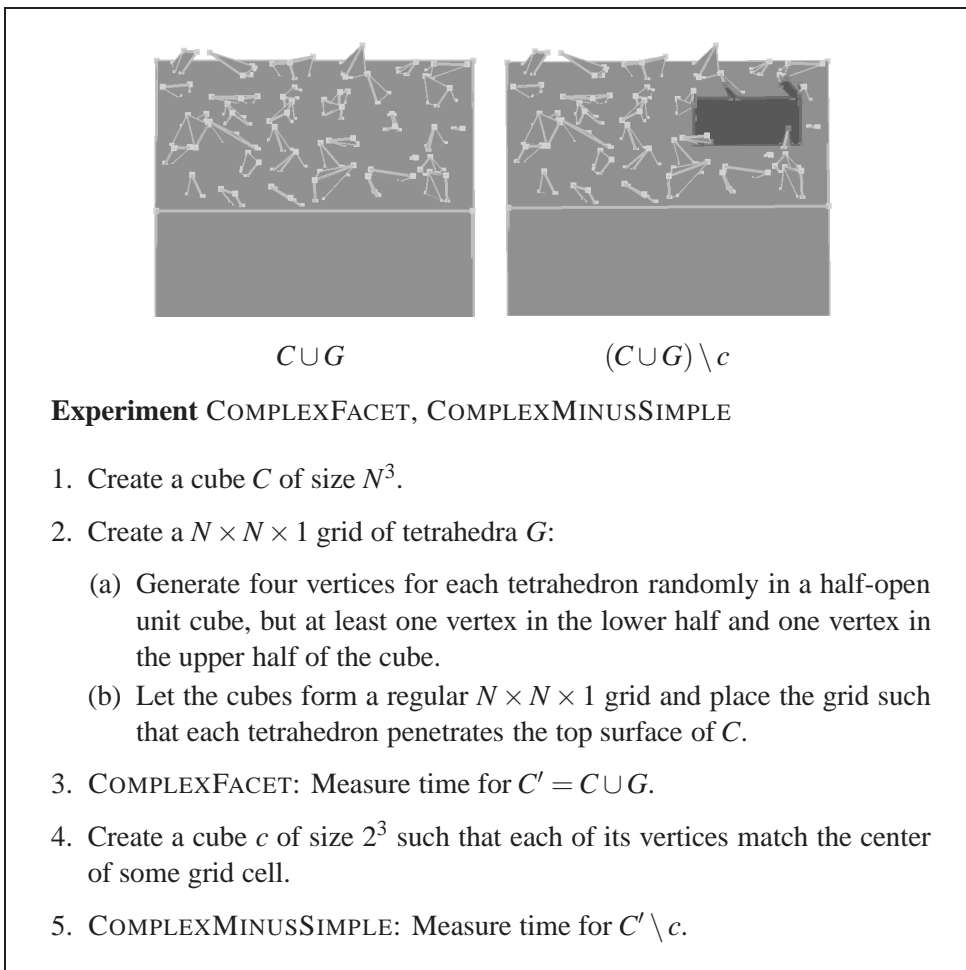


Figure 9.3: Total runtime and runtime distributed over the major subroutines for our implementation in the QUADRATICWALLGRID experiment. Note that the plane sweep and the ray shooting are not executed in this experiment.



9.2.3 A Complex Object Minus a Simple Object

We designed the COMPLEXMINUSSIMPLE experiment to reflect a common task in machine tooling where a small object is subtracted from a large complex object. Additionally, we use the first part — the construction of the complex object — as experiment COMPLEXFACET in order to stress the sweep-line algorithm sorting the facet boundary loops.

For the COMPLEXMINUSSIMPLE experiment, we perform a test series with $N = i * 5$ and $i = 1, \dots, 40$. Since we want to run this experiment with ACIS, too, we perform it on machine 1. Figure 9.4 shows the results of the experiment. There is no subroutine that is dominating the runtime. Still the kd-tree construction is

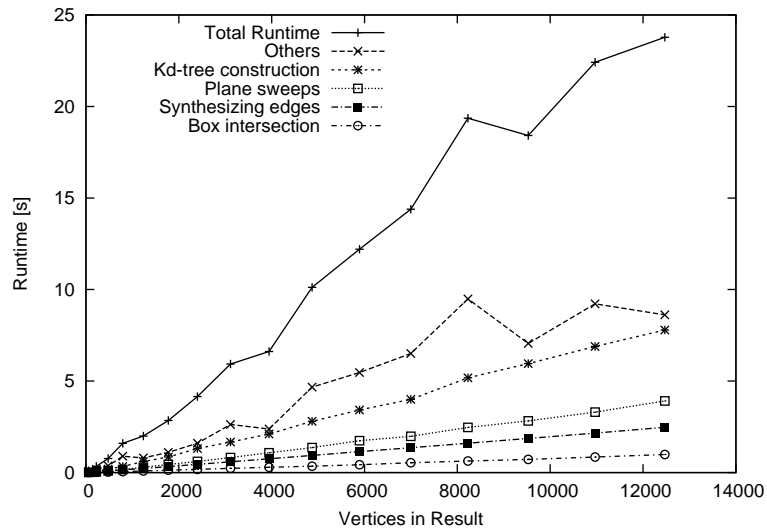


Figure 9.4: Total runtime and runtime distributed over the major subroutines of our binary operation in a test series of the COMPLEXMINUSSIMPLE experiment with $N = i * 5$ and $i = 1, \dots, 40$.

most time consuming. The other subroutines follow in the same order as in the TETGRID experiment.

As already discussed in our analysis from Chapter 7, the runtime of the binary operation depends on the complexities of both input and the output complexity. As a result, the runtime of the COMPLEXMINUSSIMPLE experiment is determined by the size of C' and the result polyhedron. Although only a constant-sized part of C' is changed by c , we perform a complete synthesis for the result polyhedron. Figure 9.4 confirms this property.

9.3 Runtime Behavior in Complex Situations

The following experiments are designed to stress single subroutines. Mostly, we are interested in those routines that proved to be most time consuming, and those that rely on a good average case performance. We want to identify those subroutines that can become the bottleneck in certain situations. Furthermore, we want to confirm the theoretical runtime analysis of Chapter 7.

9.3. RUNTIME BEHAVIOR IN COMPLEX SITUATIONS

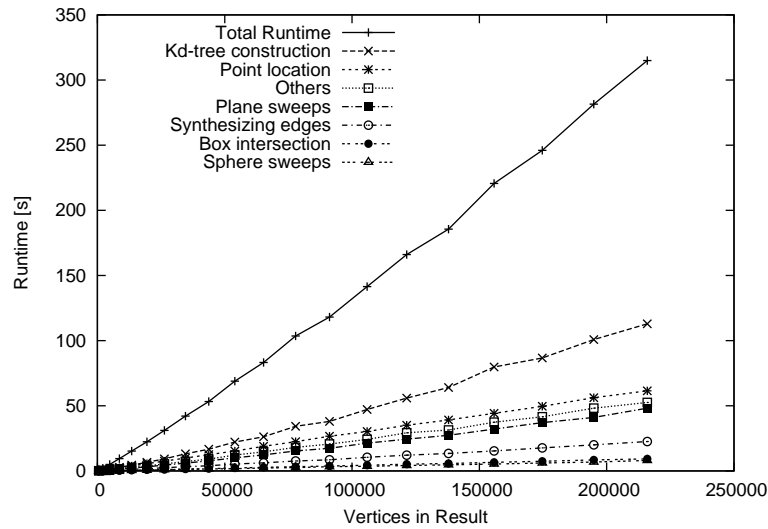


Figure 9.5: Total runtime and runtime distributed over the main subroutines for our implementation in the COMPLEXFACET experiment in a test series with $N = i * 20$, $i = 1, \dots, 10$.

9.3.1 Complex Facet

In large Nef polyhedra of complexity n , there rarely is a single supporting plane with complexity $O(n)$. On the other hand, worst-case examples do not seem very artificial. We use the COMPLEXFACET experiment (see page 95) as such worst-case example. The union of the grid of tetrahedra with the surface of the cube results in a facet with $O(n)$ holes.

Figure 9.5 shows the result of a test series with $N = i * 20$ and $i = 1, \dots, 10$ on machine 2. Although we tried to build a scenario that especially stresses the runtime of the planar sweep, and although both routines consumed about the same amount of runtime in the TETGRID experiment, the construction of the kd-tree consumes much more time in the COMPLEXFACET experiment. This effect is explicable, since the complex facet intersects most of the splitting planes. Each time an intersection test is performed, which on average consumes time linear in the size of the facet.

The runtime of the plane sweep looks close to linear, but actually has the (expected) $O(n \log n)$ behavior. If we divide the runtime by n , we still get an increasing curve. Dividing by $n \log n$ results in an oscillating, but neither increasing nor decreasing curve. Seel's experiments already confirmed this behavior [See01b].

Experiment COMPLEXSPHEREMAP

1. Create triangles $t_i, t'_i, i = 1, \dots, N + 1$ with the following properties:
 - (a) the first vertex of each triangle t_i/t'_i is located at the origin.
 - (b) the second vertex of each triangle t_i/t'_i has coordinates $(N, -N + 2 * i, N)/(N, N, -N + 2 * i)$
 - (c) the third vertex of each triangle t_i has coordinates $(N, -N + 2 * i, -N)/(N, -N, -N + 2 * i)$
2. Unite triangles t_i/t'_i as object T/T' .
3. Measure time for $T \cup T'$.

9.3.2 Complex Sphere Map

In the worst case, the overlay of all $n + m + s$ sphere maps runs in $O((n + m + s) \log(n + m))$ time. For this to happen, there must be a single sphere map with complexity $O(n + m + s)$. Usually, each sphere map is of constant size. Then the runtime of the overlay drops to $O(n + m + s)$.

In the COMPLEXSPHEREMAP experiment we can see a scenario where a sphere map of complexity $O(n + m + s)$ is created during a binary operation. Figure 9.6 shows the result of a test series of this scenario with $N = i * 50, i = 2, \dots, 17$ performed on machine 2. Again, the kd-tree construction dominates the runtime. The half-sphere sweep is the second biggest consumer, but only needs half the runtime of the kd-tree construction.

9.3. RUNTIME BEHAVIOR IN COMPLEX SITUATIONS

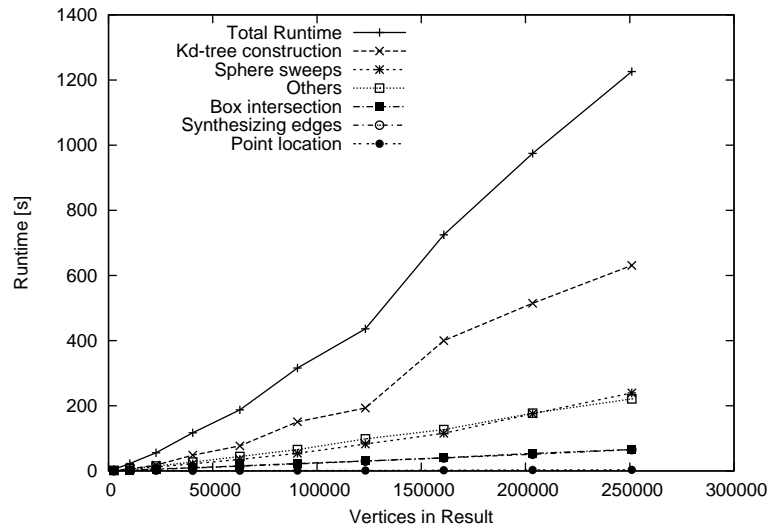


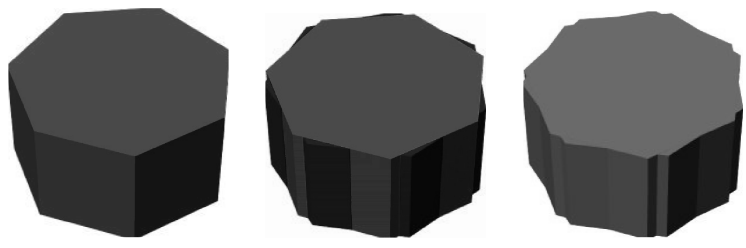
Figure 9.6: Total runtime and runtime distributed over the main subroutines for our implementation in a test series of the COMPLEXSPHEREMAP experiment with $N = i * 50, i = 2, \dots, 17$.

9.3.3 Kd-tree Construction and Queries

We use the ROTCYLINDER experiment as a worst-case scenario for the construction of the kd-tree, as well as for the point location subroutine. In the construction of the kd-tree both large facets are intersected by most of the splitting planes. Consequently, most of the split operations need to test for intersection with at least one of these two facets, which have $O(n)$ size. We therefore expect quadratic construction time.

Figure 9.7 shows the result of a test series with $\alpha = 10^{-7}$ and $n = 100i$, $i = 1 \dots 30$ on machine 2. The curve of the kd-tree construction includes some irregularities. Most remarkable is an upward leap by more than 100% from $N = 1600$ to $N = 1700$. Looking more closely, there are further leaps in the curve. They can also be found at the same places in other experiments. For example, in Figure 9.3 includes a big jump between the two runs with result sizes of about 50000 and 60000 vertices, and a small jump between the runs with result sizes of about 120000 and 140000 vertices.

Scrutinizing the construction procedure, we can observe that the number of intersection tests against complicated facets grows steadily, but jumps upwards every time the number of vertices exceeds the next power of two. The reason is that we cut off the kd-tree construction at logarithmic depth. Exceeding a power of



C
 C'
 CUC'

Experiment ROTCYLINDER

1. Create a right cylinder C :
 - (a) the base of C is a regular polygon with N sides.
 - (b) the base is parallel to the xy -plane.
2. Create a copy C' of C .
3. Rotate C' around its vertical centerline by α degrees.
4. Measure time for CUC' .

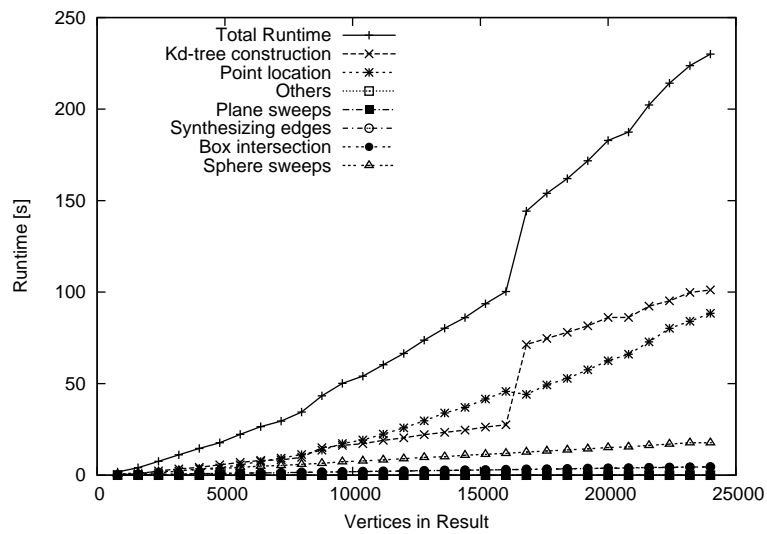
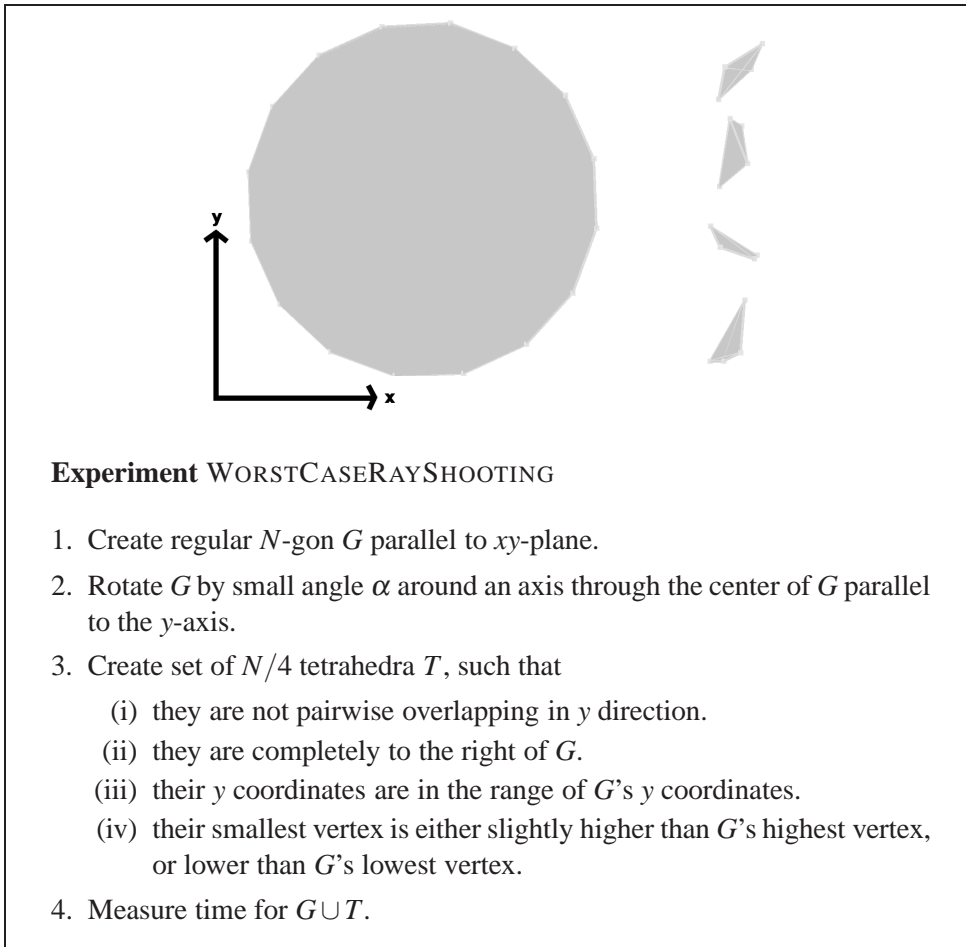


Figure 9.7: Experiment ROTCYLINDER with $\alpha = 10^{-7}$ and $n = 100i, i = 1 \dots 30$: Shown are the total runtime of the total runtime of the binary operation and the runtime of the main subroutines.



two, the tree is allowed to grow deeper by one further level. As a result, the number of inner nodes, and therefore the number of splitting planes intersecting complex facets increases abruptly. At the same time, the quality of the kd-tree as a heuristic search data structure improves, which is confirmed by the point location curve.

Because of the irregularities, it is not possible to analyze the curve properly. Another test series that only included runs where the result polyhedron is slightly larger than 2^i did not help to clarify the situation. We only can conclude that the runtime is worse than linear.

The ROTCYLINDER experiment works fine as a worst-case scenario for point location. Most of the leafs contain either of the large facets. In combination with $O(n)$ point location queries, the subroutine is expected to use quadratic runtime.

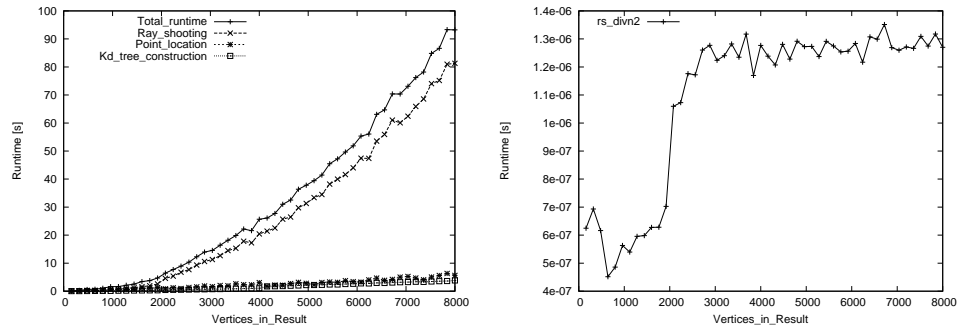


Figure 9.8: Experiment WORSTCASERAYSHOOTING with $N = 80i$, $i = 1, \dots, 50$: The left graph shows the runtime of the major subroutines and the total runtime of the binary operation. The graph on the right shows the runtime of the ray shooting subroutine divided by the squared output complexity.

The curve of the point location in Figure 9.7 supports our assumption, but a closer look at the data reveals a sub-quadratic behavior. Dividing the runtimes by the complexity of the result yields a curve, which seems to be linearly growing, but dividing the runtime by the squared complexity of the result gives a slightly falling curve.

It is not easy to construct a worst-case scenario for the ray shooting subroutine. As we have seen in previous experiments, usually ray shooting only accounts for a negligible amount of time. In a worst-case scenario, there must be $O(n)$ ray shooting queries, that visit $O(\sqrt[3]{n})$ kd-tree leaves. On average, the intersection tests performed at each kd-tree leaf must have linear complexity. We try to realize a worst-case scenario with the WORSTCASERAYSHOOTING experiment. The polyhedron that results from the final union operation has $O(n)$ shells. To resolve the nesting structure of the shells, a ray shooting query is performed from the lexicographically smallest vertex of each shell in the $-x$ direction. The $O(n)$ rays cast from the tetrahedra travel closely along G without hitting it. All those queries visit $O(\sqrt[3]{n})$ kd-tree leaves and many of the corresponding regions are intersected by the complex facet.

Figure 9.8 shows the result of a test series of the WORSTCASERAYSHOOTING experiment with $N = 80i$, $i = 1, \dots, 50$. Ray shooting accounts for most of the runtime in this experiment. Also we can see from the second graph, that the ray shooting probably has a quadratic behavior, maybe it even reaches the theoretic worst-case behavior. However, it is very unlikely to encounter a quadratic runtime of the ray shooting subroutine in a non-artificial scenario. The runtime already drops to sub-quadratic, if the complex facet in the WORSTCASERAYSHOOTING

experiment is not placed perfectly. For instance, the runtime drops when the facet becomes parallel to the xy -plane, or its normal vector essentially faces into the x -direction. Also, we can easily adjust the kd-tree such that ray shooting in the WORSTCASERAYSHOOTING experiment also drops to sub-quadratic by introducing bounding boxes around complex facets. Then, the intersection test of the ray with the box already reveals that they do not intersect.

9.4 Comparison with ACIS

We compare our implementation with ACIS R13 by Spatial Corp. [Spa04], one of the three common commercial CAD kernel along with Catia [Das] and Parasolid by UGS [UGS]. It is used in many CAD systems like for instance Auto CAD by Autodesk [Aut]. It should be said that we are comparing apples with oranges here. On the one hand, it is daunting for a research prototype to be compared with a long established and optimized industry implementation. On the other hand, ACIS is handling more general geometries and has some overhead in dispatching function calls to the specialized functions for linear geometry. However, our implementation handles Nef polyhedra in their full generality with all the potentially occurring degeneracies in the algorithms and it uses exact arithmetic to be reliable and robust. We use the SCHEME interface of ACIS that has some small overhead in translating function calls to the C++ library calls. ACIS also seems to store more information, because in our experiments it swaps earlier than our implementation. However, we store exact number types with their burden of memory usage. All this said, our comparison is still important to demonstrate where we are in the context of existing systems.

Comparisons with ACIS were measured on machine 1, a 846 MHz Pentium III processor with 256 MB RAM. Our implementation runs under Linux, while we used the Microsoft Windows XP version of ACIS. Our test programs for ACIS are written in SCHEME. According to Spatial Corp. SCHEME commands are mapped to their C++ counterparts. The input data for the SCHEME scripts is created by ACIS primitives or loaded from SAT-files. SAT is ACIS' open file format. For the generation of SAT files we implemented a function writing Nef polyhedra as SAT files.

9.4.1 Balanced Binary Operations

To get a general impression, we repeat the TETGRID experiment with ACIS. It contains no special difficulties. However, facets are likely to have holes and we do

N	result vertices	runtime [s]	
		ACIS R13	Nef 3D
3	338	0.29	0.61
4	1135	0.63	2.53
5	2390	1.37	5.71
6	4548	2.79	11.37
7	7383	5.29	19.26
8	11555	10.13	30.61
9	16998	14.27	48.02
10	23883	22.81	67.31
11	32892	25.58	96.12
12	43418	35.58	126.01
13	56188	55.64	164.05
14	70827	swapping	211.02
15	87871	swapping	262.62
16	108066	swapping	321.72
17	131304	swapping	413.32

Table 9.6: Comparison of ACIS R13 and our Nef polyhedron with the TETGRID experiment.

not exclude degeneracies explicitly, but they are highly unlikely. Naturally, both algorithms perform on the same data sets.

The results in Table 9.6 show that ACIS is faster by a factor of two to four. The factor fluctuates and no obvious trend is visible. ACIS swaps heavily for $N \geq 14$ on our test machine. We therefore excluded these timings for ACIS. As our implementation performs a few further runs without swapping, it seems that we need less memory for the representation of the same polyhedron.

9.4.2 Floating-Point versus Exact Arithmetic

One of the major differences between ACIS and our implementation is our use of exact arithmetic instead of floating-point arithmetic. Floating-point and interval arithmetic are the state-of-the-art in Computer Aided Design, and we are not aware of any commercial system that uses exact arithmetic to solve the remaining cases that floating-point and interval arithmetic cannot solve. An obvious reason is the runtime cost for exact arithmetic, but also the difficulties in realizing exact and efficient solutions for more general curves and surfaces may play a role.

9.4. COMPARISON WITH ACIS

n	α	time	runtime [s]
		ACIS R13	Nef 3D
100	10^{-1}	1.08s	3.47s
	10^{-2}	1.05s	3.50s
	10^{-3}	1.08s	3.59s
	10^{-4}	1.07s	3.64s
	10^{-5}	not executable	3.72s
	10^{-6}	not executable	3.77s
1000	10^{-1}	61s	67s
	10^{-2}	61s	68s
	10^{-3}	61s	69s
	10^{-4}	not executable	69s
	10^{-5}	not executable	71s
	10^{-6}	not executable	71s
2000	10^{-1}	252s	195s
	10^{-2}	253s	198s
	10^{-3}	255s	203s
	10^{-4}	not executable	205s
	10^{-5}	not executable	207s
	10^{-6}	not executable	210s
10000	10^{-7}	not executable	3219 s

Table 9.7: Comparison of ACIS R13 and our Nef polyhedron with the ROTCYLINDER experiment. Here, “not executable” means that ACIS could not compute the union without topological errors and therefore cancels the operation. As a result, ACIS keeps the first input object unmodified and deletes the second input object.

We designed the simple ROTCYLINDER experiment (see page 100) to demonstrate the effect of exact arithmetic; on one hand, we gain expressiveness in modeling, because we can compute results where other systems fail very soon, and on the other hand, we have to deal with the runtime costs of exact arithmetic. We already analyzed the aspect of coordinate growth in Section 9.5. Now, we investigate the runtime costs of exact arithmetic in comparison to ACIS.

In this ROTCYLINDER test scenario we have $4n$ edge–edge intersections. In one half of those intersections the endpoints of the intersecting edges are extremely close together. Without an adequate precision it is not possible to compute an intersection point that is on both edges and different from the endpoints.

We omit the expensive computation of the exact rotation in our test series (see Section 6.2) and focus on the binary Boolean operation. The result of the ROT-

n	α	runtime [s]	
		ACIS R13	Nef 3D
1000	10^{-1}	21.57	31.55
	10^{-2}	20.60	32.88
	10^{-3}	20.75	33.51
	10^{-4}	20.79	34.63
	10^{-5}	not executable	35.41
	10^{-6}	not executable	36.11

Table 9.8: Comparison of ACIS R13 and our Nef polyhedron with the ROTCYLINDER experiment with the modification that the second cylinder is translated along the z-axis before computing the union such that all edge–edge intersections change to edge–facet intersections.

CYLINDER experiment in Table 9.7 shows that ACIS’ floating-point operations are insufficient for α smaller than 10^{-3} . ACIS’ binary operations modify the first input object; it becomes the result during the operation. The second input object is deleted meanwhile. When the union operation fails, the first input object stays unchanged. The second input object is deleted.

On the other hand, ACIS is faster except for very large instances. For $n = 100$ the factor of our runtime and ACIS’ runtime is slightly below four; for $n = 2000$ we are faster up to a factor of 1.2. Additionally, we listed a run with $n = 10000$ and $\alpha = 10^{-7}$ for comparison. In Section 9.5 we performed further runs with angles as small as 10^{-40} . As claimed, robustness is not an issue.

This experiment is particularly complex because of the edge–edge intersections. We repeat parts of this experiment with the modification that the second cylinder is shifted along the z-axis before computing the union. As a result, we get edge–facet instead of edge–edge intersections, which can be computed by hand as discussed in Section 9.1.3. We expect that the modification results in a clear improvement of our runtime. The modification will also be beneficial for ACIS’ runtime, since vertices are not so close together any more. The results in Table 9.8 show that both algorithms benefit from this change; our algorithm by a factor of about two, and ACIS by a factor of about three. Nonetheless, ACIS aborts the union computation for an angle below 10^{-4} .

9.4.3 A Complex Object Minus a Simple Object

We designed the COMPLEXMINUSSIMPLE experiment (see page 95) to reflect a common task in machine tooling where a small object is subtracted from a large

9.5. GROWTH OF COORDINATE REPRESENTATION

N	result vertices	runtime [s]	
		ACIS R13	Nef 3D
3	61	0.044	0.10
6	218	0.078	0.34
9	460	0.156	0.77
12	801	0.233	1.59
15	1241	0.379	2.00
18	1759	0.556	2.84
21	2392	0.845	4.15
24	3117	1.056	5.93
27	3960	1.334	6.61
30	4870	2.069	10.12
33	5912	1.983	12.20
36	6999	2.814	14.38
39	8235	3.175	19.36

Table 9.9: Comparison of ACIS R13 and our Nef polyhedron with the COMPLEXMINUSSIMPLE experiment. ACIS is about six times faster than our implementation.

complex object. We repeat this experiment with ACIS and compare it with the results of our implementation from Section 9.3.

Table 9.9 shows the results of a test series with $N = i*3, i = 1, \dots, 13$. Here, the difference between ACIS and our algorithm is quite pronounced with ACIS being a factor of about six faster than our implementation. A notable difference might be in the software interface; ACIS modifies the first input object to become the result, while our implementation creates the result from scratch without modifying the two input polyhedra. Still, ACIS also does not seem to profit from the in principle constant-size problem complexity here.

9.5 Growth of Coordinate Representation

One major critic of exact computation is that it is slow in general, and gets even slower in cascaded constructions. Constructing geometric objects usually starts from geometric primitives, which are combined to complex objects via geometric constructions. Geometric constructions can be expressed by multiplications, additions and subtractions. Each addition or subtraction can increase the bit complexity by one, each multiplication adds up the bit complexities of the factors. In

cascaded constructions the output of one algorithm becomes the input of the next. This way, the bit complexity grows in every iteration. The cascaded construction of a geometric object from primitives can be illustrated as a construction tree, where the primitives are the leaves of the tree, and the final object is at the root of the tree. Milenkovic shows in [Mil00] that the bit complexity of a construction grows exponentially with the height of its construction tree in the worst case.

We have already compared floating-point arithmetic with exact arithmetic by one experiment in Section 9.4. But from this experiment we did not get a good impression of the impact of coordinate growth. We now examine two scenarios that should give us some insight. In cascaded constructions the output of an operation is taken as the input of the next. Consequently, the bit complexity is continuously growing with each step. In the first scenario, we are interested in the growth of the bit complexity when the result of an operation is combined with geometric primitive of constant bit complexity in the next run. This means, that in every operation we combine an object with constant bit complexity with an object with growing bit complexity. Here, the bit complexity grows linear with the height of the construction tree. In the second scenario, we want to examine the growth of the bit complexity when we have a real construction tree, i.e., on each level we only combine objects that have the same distance to the leafs. Consequently, only objects with roughly equal bit complexity are combined. The bit complexity at least doubles with each iteration.

Consecutive binary operations on Nef polyhedra do not increase the bit complexity, since binary operations do not introduce new plane coordinates. It is not possible to perform cascaded operations with the functionality provided by our package. Consequently, we rather simulate cascaded constructions.

Instead, we simulate the two scenarios with two variations of our ROTCYLINDER experiment (see page 100). For the first scenario, we perform a test series of the ROTCYLINDER experiment with a growing angle α . At the moment, there is no practical solution to perform a rotation of exactly α degrees, as pointed out in Section 9.4. Also, it is expensive to compute good rational approximations for sine and cosine. The CGAL function `rational_rotation_approximation` provides exact sine and cosine values for some α' , such that $|\alpha - \alpha'| < \varepsilon$ for a small specified $\varepsilon > 0$, but is very runtime intensive for small ε (see Table 6.1). Instead of approximating angles, we use angles from which we know the exact rational representation of sine and cosine. With $\sin(\alpha) = \frac{2 \cdot 10^i}{10^{2i} + 1}$ and $\cos(\alpha) = \frac{10^{2i} - 1}{10^{2i} + 1}$, we know that $\sin^2(\alpha) + \cos^2(\alpha) = 1$ and $\alpha \approx 1.993373 \cdot 10^{-i}$. We execute runs for $i = 1, \dots, 50$, and with $n = 1000, 2000$.

Figure 9.9 shows the result of the test series performed on machine 2. It depicts the relation of the runtime and the bit complexity of the coordinate representation.

9.5. GROWTH OF COORDINATE REPRESENTATION

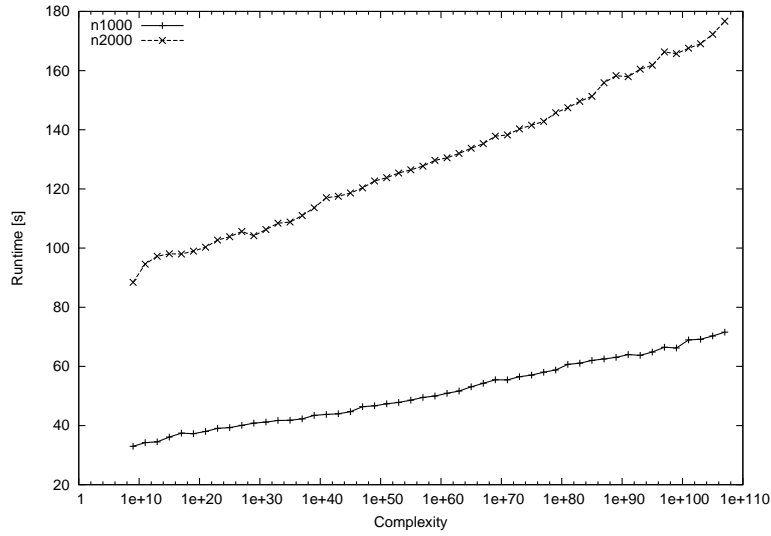


Figure 9.9: Experiment ROTCYLINDER with $\alpha \approx 1.993373 \cdot 10^{-i}$, $i = 1, \dots, 50$, and with $n = 1000, 2000$: The graph depicts the relation between the bit complexity of the coordinate representation and the runtime.

We measure the order of magnitude of the coordinate values by the value of the largest integer used to represent a coordinate, and compute the bit complexity of the integer. During the test series, the largest coordinate representation grows from 10^9 to 10^{107} . This relates to a bit complexity between 30 bit and 355 bit per vertex coordinate. In this experiment, the coordinates of C stay constant. Only the coordinates of C' grow. Consequently, the complexity of the arithmetic operations is only growing linearly. Figure 9.9 and a closer examination of the experiment data support this evaluation.

For the second scenario, we want to combine two objects with the same bit complexity. For this purpose, we adjust the ROTCYLINDER experiment as follows: We rotate C by $\beta \approx 1.993373 \cdot 10^{-i}$ degrees at the beginning of each run to obtain large coordinate representations. Then C is copied and the copy C' is rotated by $\alpha = 10^{-8}$ degrees. Finally, we unite C and C' . We perform test runs with $i = 1, \dots, 50$, and $n = 1000, 2000$. Since α is a relative large constant angle in comparison to β , the bit complexity of C and C' are about the same. Figure 9.10 depicts the result of this test series. An close examination of the data shows that the curves roughly fit the function $f(x) = ax^{1.4} + x_0$, where a and x_0 are constant values.

Now, we put our simulated scenarios in relation to a real scenario. Let us

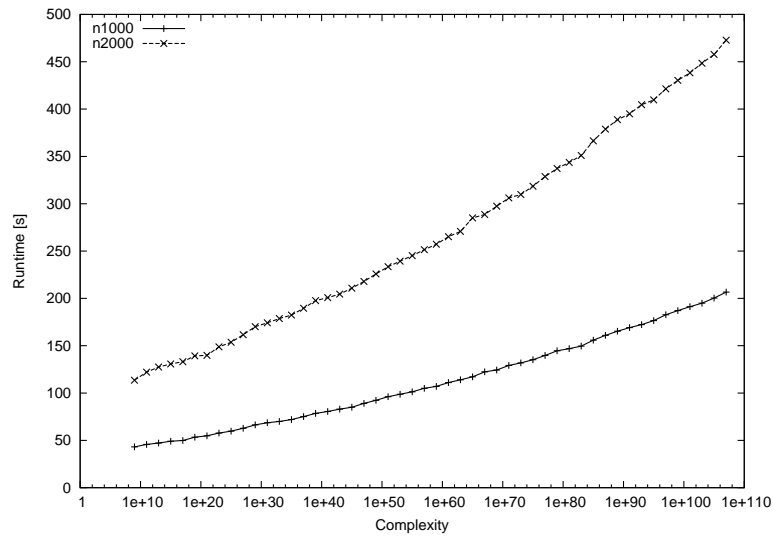


Figure 9.10: Experiment ROTCYLINDER with $\alpha = 10^{-8}$, and with $n = 1000, 2000$: In order to obtain two input polyhedra with equally large coordinate representations, C is initially rotated by $\beta \approx 1.993373 \cdot 10^{-i}$ degrees. The rotated C is then used as input object for the ROTCYLINDER experiment. The graph depicts test series with $i = 1, \dots, 50$

assume for the sake of this comparison that coordinates of 3D models in the CAD community are stored as signed numbers that consist of up to 9 decimal digits with a floating-point and no exponent. If we construct a Nef polyhedron from data using this representation, the floating-point numbers of the point coordinates are converted to rational numbers represented by integers with up to 30 bit.

For cascaded operations on Nef polyhedra, we must repeatedly apply two steps. First, we perform binary operations to obtain new points from the intersection of the involved polyhedra. Then, we construct new polyhedra from the intersection points. We are interested in the growth of the vertex coordinates caused by these two steps. New vertices are constructed from edge–edge and edge–facet intersections. With 30 bit vertex coordinates, the geometry of an edge can be represented by its two endpoints, and therefore only needs the existing 30 bit representations. The direction of an edge is computed as the difference of two points and needs 31 bit. The supporting planes of the facets are constructed from three points. This construction has degree 3 and constructs plane coordinates with at most 93 bit. An edge–plane intersection also has degree 3, where the factors of the monomials with the highest degree include one plane and two vector coordinates. Thus, the result

vertex can be represented with at most 158 bit. Vertices that result from edge–edge intersections are less complex.

From our experiments, we can conclude how the runtime changes from the first to the second operation in a row of cascaded operations. The first operation is performed on polyhedra with vertices that use 30 bit for each integer, while during the second operation 158 bit are needed. We compare runs of our experiment with $i = 1$ and $i = 21$, which resemble complexities of 30 and 162 bit. In the first scenario, the runtime grows by 41% for $n = 1000$ and by 39% for $n = 2000$. In the second scenario, the runtime grows by 114% for $n = 1000$ and by 99% for $n = 2000$.

9.6 Résumé

In this chapter, we confirmed the efficiency of our implementation, but we also discovered the following three weaknesses:

1. The kd-tree related operations perform bad on polyhedra with linear-sized facets.
2. The complexity of the binary operation is not sensitive to small areas of concern. For example, even if we subtract the empty space from some polyhedron, the operation still performs a complete synthesis and a complete kd-tree construction for the result.
3. Coordinate growth can become a big issue.

In the following, we discuss opportunities to deal with each weakness.

9.6.1 Further Improvement on Point Location and Ray Shooting

Although the kd-tree improves the performance of our binary operations considerably, it is still the bottleneck. Most notably are operations with a quadratic-sized result polyhedron, and operations on polyhedra with linear-sized facets.

In the latter case, we would like to have a method that allows us only to operate on the relevant part of a complex facet. In a point location query we want to test for intersection with that part of the facet that lies within the boundaries of the relevant kd-tree cell. Likewise, we want to split a facet into two halves each time it is intersected by some splitting plane during the construction of the kd-tree.

We performed experiments with storing triangulated facets in the kd-tree. Unfortunately, the triangulation often results in badly shaped triangles, i.e., the triangles are long and skinny, such that they intersect many kd-tree cells. As a result, the runtimes change for the worse. We expect better result from a triangulation method that returns *fat* triangles, i.e, triangles that are not long and skinny.

As pointed out above, the kd-tree construction is our major bottleneck. On the other hand, we construct the kd-tree at the end of the binary operation only to use it for a few ray shooting queries for most of the times. If the kd-tree is not used further for subsequent binary operations or for point location and ray shooting queries posed by the user, the effort seems wasted. Therefore, we want to find out, whether there is some efficient way to solve the ray-shooting queries without constructing the kd-tree, or at least without constructing it completely. We can either provide an additional ray shooting solution that is not based on the kd-tree, or we construct only those parts of the kd-tree that are needed to solve a given set of ray shooting queries. Then, we can offer exchangeable ray shooting strategies. The user can decide whether a kd-tree is constructed completely at the end of the synthesis for the ray shooting and future queries, or whether it is only constructed as much as needed for the ray shooting.

As a third approach to get rid of the kd-tree construction in the synthesis, we are interested in whether it is possible to perform the ray shooting queries needed for the synthesis on the kd-trees of the input polyhedra rather than on the kd-tree of the result. The following must be realized: First, we must deduce those input vertices that might become the smallest vertices of the shells in the result polyhedron. In a union operation these can only be the smallest vertices of the shells in both input polyhedra, and in an intersection operation these can only be the smallest intersection vertices of each shell with the respective other input polyhedron. In a symmetric difference and in a difference operation it is a combination of these two vertex types. Having the locations of these vertices, we perform ray shooting queries on the kd-trees of both input polyhedron. We shoot rays from the locations of the smallest vertex candidates in $-x$ direction. In order to remember the location l_h hit by a ray shot from location l_s , we associate l_h with a sphere map at l_s . If there is no sphere map at l_h , we create a redundant sphere map on the fly. This way we can obtain two sphere maps intersected by a ray shot in the result polyhedron from l in $-x$ direction. Since a ray shooting query reports the first intersected boundary element, we must determine which of the two given sphere maps represents the local pyramid of that boundary element. If both represent the local pyramid of some boundary element, the boundary element represented by the sphere map at the lexicographically larger position would be hit first. If only one of the sphere maps represents a boundary element, this boundary element would be hit. If both

sphere maps represent volumes, the ray shooting query would hit no boundary element. Hence, the considered shell is surrounded by the outer volume. Note that all artificially inserted sphere maps are removed by the simplification routine after the synthesis.

This method surely is only effective, if a single boolean operation is performed and afterwards no ray shooting or point location is needed by the user. But it can also become effective in consecutive binary operations, if we use it to replace the kd-tree completely with some streamed data structure similar to the fast box intersection. Since we perform ray shooting queries on the input polyhedra, a streamed data structure can perform all point location and ray shooting queries batched at the same time.

9.6.2 Modification Operations

The complexity of our binary operations always depends on the largest polyhedron among the two input polyhedra and the result polyhedron. Thus, subtracting a single point from a polyhedron P is just as complex as intersection P with a transformed copy of P , although the subtraction can be seen as a slight modification of P . Our data structure and algorithms are not sensitive to small changes or to a small area of concern. For this reason, we would also like to have modification operations in addition to our binary operations. Such operations yield the same results as the binary operations, but instead of creating a third polyhedron that holds the result, they return the modified first input polyhedron.

In order to realize modification operations we need update operations for the SNC and the kd-tree. An alternative to an update operation of the kd-tree is again to replace the kd-tree with some streamed data structure. This way we also can easily decide upfront which objects need to be considered. The decision is done by checking the bounding boxes of all objects against a box that encloses the area of concern.

9.6.3 Exact Geometric Computing

In general, the problem of growing coordinate representations in cascaded constructions cannot be avoided, but it is possible to reduce the performance downside of exactness in geometric algorithms. The exact geometric computing (EGC) paradigm [YD95] emphasizes that exactness must be in the geometry, not in the arithmetic. This means, we need not use exact arithmetic, but the outcome of every predicate evaluation must be correct for the given input. In CGAL many algorithms

and data structures are designed for the use of *floating-point filters*. The idea is to exploit the machine floating-point arithmetic, which is highly optimized on current hardware. For this purpose, the machine evaluation of predicates is certified. When a correct result cannot be guaranteed, the predicate is recomputed with the slower exact methods [LPY04].

Unfortunately, our current implementation does not benefit from floating-point filters. On the contrary, floating-point filters slow our implementation down. The reason is in our synthesis step. Floating-point filters are only effective as long as most of the predicates can be solved with floating-point arithmetic. In the synthesis step we regularly test for equality of geometric objects. In detail, we categorize halfedges by their supporting line, we identify facet cycles by the common supporting planes of shalfedges, and we perform plane sweeps for all shalfedges lying in a common plane. Equality tests on geometric objects can easily be handled with floating-point filtering, as long as the tested objects are clearly unequal. The closer to equal two objects are, the more precise must the arithmetic be in order to decide whether two objects are the same, or slightly different. In our case, we apply equality tests because we want to pair up equal geometric objects. Hence, the filters regularly have to fall back to exact arithmetic.

In order to allow the effective use of floating-point filters for our binary operations, we must re-design the synthesis step. The idea is to match up halfedges and shalfedges by indices instead of geometric properties. Indices can be set properly in the constructors and input operations. Then, they must be transferred and updated during the boolean and topological operations. The re-design is elaborate, but seems possible. As an example, we discuss the modifications that are necessary to pair up the halfedges by indices.

First, we have to specify how indices are assigned. The two halfedges of an halfedge pair must always have the same index. Multiple halfedges pairs with the same supporting line may share a common index, but do not have to. With this rule, the pairing is still easy and effective, and we do not include unnecessary restrictions.

In all our constructors, we know which halfedges comprise a pair before we compute their geometric properties. Thus, it is easy to assign a unique index to each halfedge pair. When we create the sphere map of an edge on the fly during a binary operation, both new svertices take on the index of the edge.

There are two steps, where the handling of indices is complicated. First, erasing a redundant sphere map on an edge, we may have to join two halfedge pairs with different indices, but have not even paired up the two pairs. And second, the overlay of two sphere maps may introduce new svertices. Such a new svertex must get the

9.6. RÉSUMÉ

same index as some other svertex in a different sphere map. Of the latter svertex, we do not know in which sphere map it is, and whether it has already been created. Both problems can be solved with additional associations.

With these modifications, the pairing is easy. Instead of categorizing halfedges by their common Plücker coordinates, we categorize them by their common index. Then we sort each list of halfedges as before. Note that we first decide whether two compared halfedges have the same source vertex before we compare their location lexicographically. This way we compare equal coordinates, only if compared points are unequal, but have the same x -coordinate, i.e., the supporting line is orthogonal to the x -axis.

Chapter 10

Applications for Nef polyhedra

In this chapter we present first approaches for realizing two applications of 3D Nef polyhedra—for the visual hull and for the Minkowski sum of two closed Nef polyhedra. These two implementations are complete and robust, but do not use the most sophisticated and efficient algorithms. Also, the Minkowski sum is limited to closed polyhedra, so far, We describe the algorithms, perform tests to get a first impression of the performance, and discuss their potential.

10.1 Visual Hull

The visual hull of a three-dimensional object is an approximation of the original, deduced from concurrent snapshots of several cameras facing the object [Lau94]. Each snapshot provides a two-dimensional silhouette. The idea is to create a cone for each camera, such that the shape of their cross-section equals the silhouette of the snapshot. This way, the cone closely covers each object that could have caused the silhouette. Each cone already is a rough approximation of the original object. Intersecting the cones of all cameras refines the approximation. Note, the original object is always a subset of the approximation. Figure 10.1 illustrates the method.

Having Nef polyhedra as provided in CGAL 3.1, we implemented a solution for the visual hull problem based on connected polygonal silhouettes with holes. Although cones are infinitely bounded polyhedra, we want to use a standard kernel in order to get a fast solution. We therefore clip each cone at a fixed box. This box is not computed by us, but defined by the user. The implementation was easy and took only a few days.

The main advantage of our implementation lies in its robustness. If many cameras are used, the boundary edges of the silhouettes of multiple cameras standing

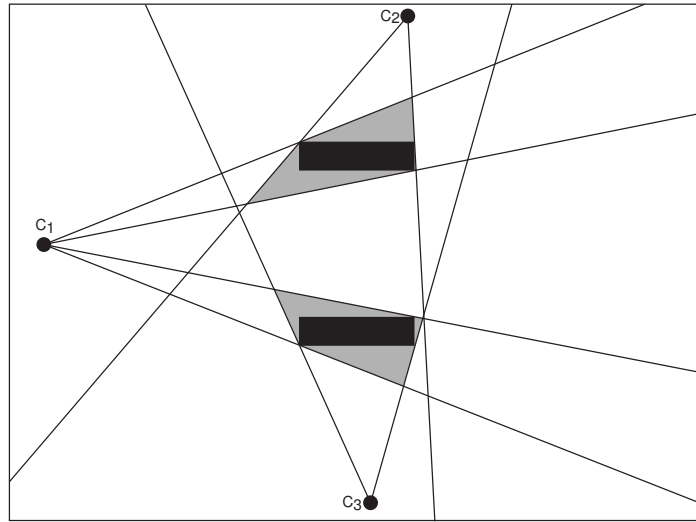


Figure 10.1: Two rectangular objects observed by three cameras c_1 , c_2 , and c_3 . The grey area illustrates the visual hull computed from the pictures seen by the cameras.

closely together might arise from the same edge of the object, i.e., several cones intersect in a common edge. Here, floating-point arithmetic can cause severe problems. The main disadvantage of our implementation is a lack of speed.

Without genuine snapshot data, we tested our implementation with artificially created data. Figure 10.2 shows such an artificial example. We computed the intersection of three cones generated from artificial silhouettes showing the letters M, P, and I. Using a computer with two 3GHz processors and 4GB RAM, our implementation creates a single cone from a silhouette with 10-20 vertices in about 0.005 seconds, and intersects two of these cones in about 0.1 seconds. The polyhedron shown in Figure 10.2 was computed in 0.217 seconds. Visual hulls are often used in real-time applications. For this purpose, we must speed up our implementation by a factor of about 100.

The intersection of the cones consumes most of the runtime. As we will see in the next section, the Minkowski sum has the same bottleneck. In this context, we consider different strategies to perform a sequence of union or intersection operations efficiently.



Figure 10.2: Two views on a visual hull example created from three two-dimensional silhouettes showing the capital letters M, P, and I.

10.2 Minkowski Sum of Two Nef polyhedra

The Minkowski sum of two point sets $S_1 \subset \mathbb{R}^d$ and $S_2 \subset \mathbb{R}^d$, denoted by $S_1 \oplus S_2$, is defined as

$$S_1 \oplus S_2 := \{p + q : p \in S_1, q \in S_2\},$$

where $p + q$ denotes the vector sum of the vectors from the origin to p and q , respectively. If $p = (p_1, \dots, p_d)$ and $q = (q_1, \dots, q_d)$ then we have

$$p + q := (p_1 + q_1, \dots, p_d + q_d).$$

Minkowski sums are often used in robot motion planning for non-trivially shaped robots with translational movement. For such an application, we want to compute the configuration space of a robot R with respect to a set of obstacles O , i.e., each placement of the robot without intersecting any obstacle. The placement of the robot is given with respect to some fixed reference point of R . By $R(x, y)$ we denote the placement of R at position (x, y) . The configuration space can be calculated as the negation of the Minkowski sum $O \oplus -R$, with $-R$ defined as $-R := \{-p : p \in R\}$, i.e., the point set $O \oplus -R$ includes all illegal placements of R . $-R$ can be obtained by reflection about the origin [dBvKOS97].

The computation of the Minkowski sum of two three-dimensional polyhedra M and N is a complex problem by nature. It yields another three-dimensional polyhedron, whose worst-case complexity is $O(m^3n^3)$, where m and n are the complexities of the respective input polyhedra. Because of this, there is no exact implementation

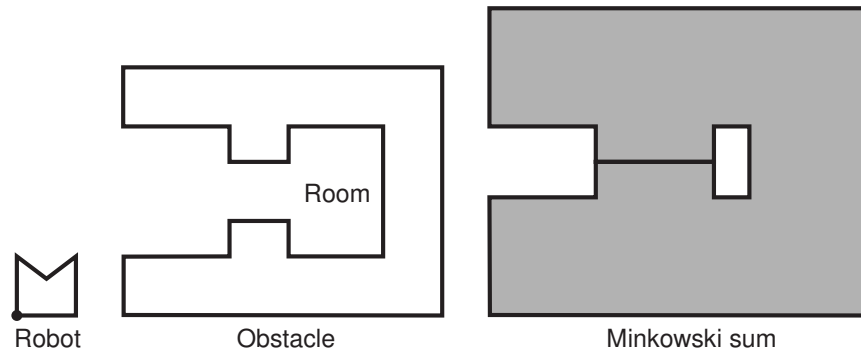


Figure 10.3: Tight passage: Can the robot move through the tight passage into the room by translational movement? The Minkowski sum provides the configuration space of the robot. Modeling the robot as an open polygon and the obstacles as a set of closed polygons, the Minkowski sum is an open point set that denotes the illegal placements. The robot can move along the boundary of the Minkowski sum into the room.

to this problem. There only exist approximative approaches like [VM04]. On the other hand, approximative approaches are obviously not sufficient to solve tight passage problems as shown in Figure 10.3. We present a first exact approach to the 3D Minkowski sum. Yet, our implementation is restricted to closed polyhedra, and not all subroutines are solved by the most efficient methods. Then again, with our binary operation as the basic building block for the most complex step, we could already achieve promising results.

Our implementation adopts a common approach for the Minkowski sum of non-convex polyhedra, which is based upon the solution for convex polyhedra. The idea is to decompose non-convex polyhedra into convex sub-polyhedra. Then, the Minkowski sum of two non-convex polyhedra is the union of all pairwise Minkowski sums of the two sets of sub-polyhedra.

In the following, we will first discuss solutions for the Minkowski sum of convex polyhedra and present our solution based upon Nef polyhedra embedded on the sphere. Then, we adapt a vertical decomposition method for the volume of a 3D Nef polyhedron. Afterwards, we test strategies for the union of multiple polyhedra by consecutive binary operations. Finally, we point out limitations and weaknesses of our first implementation, and discuss opportunities of improving it with respect to completeness and efficiency.

10.2.1 The Minkowski sum of convex polyhedra

The Minkowski sum of two convex polyhedra is a convex polyhedron, too. Furthermore, it is well known that each vertex $v_{P\oplus Q}$ of the Minkowski sum $P\oplus Q$ is the vector sum of vertices v_P in P and v_Q in Q [Lat91]. Hence, a trivial solution for the Minkowski sum of two convex polyhedra P and Q computes the convex hull of all vector sums of vertex pairs of P and Q . This algorithm performs a convex hull algorithm on pq vertices, where p and q are the number of vertices in P and Q . Thus, using the CGAL `convex_hull_3` function the trivial algorithm runs in $O(pq\log(pq))$ time.

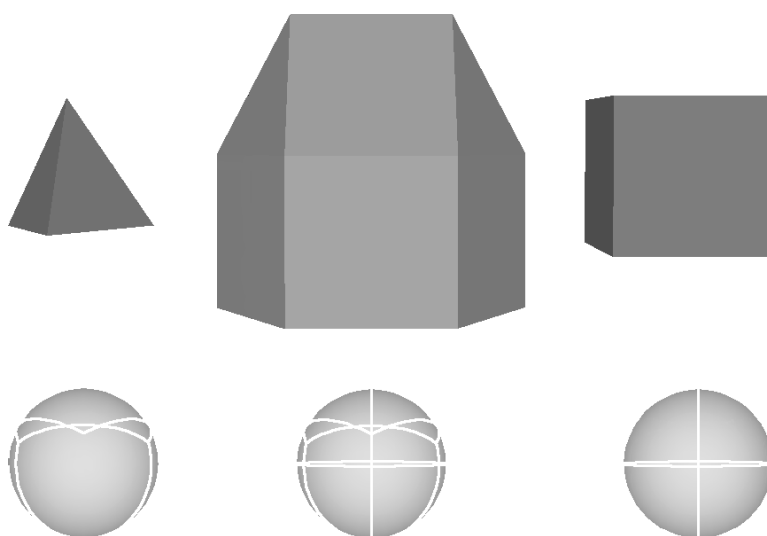


Figure 10.4: The upper row shows a tetrahedron and a cube together with their Minkowski sum. The lower row shows the normal diagrams of the three objects.

A more efficient solution can be obtained by using normal diagrams. Each convex polyhedron P has a unique dual representation N_P called the *Gaussian diagram* or *normal diagram*. It is a subdivision of the sphere into vertices, edges and faces, such that the outward-directed normal directions of all planes supporting some item of P constitute an item of N_P . A plane supports an item i of P , if the intersection of the plane and P is i . For a facet of P there is exactly one plane supporting it. Thus, its dual item is the single point on the sphere with the same normal direction as the supporting plane. The normal directions of the planes supporting an edge e_P of P form a great arc on the sphere. The endpoints of the great arc are dual items of the facets incident to e_P . A face f_n on N_P is the dual item of a vertex v_P of P . f_n is bound by a convex cycle of edges and vertices, which are the dual items of

the edges and facets incident to v_p . The order of the edges and vertices around f_n coincides with the order of dual items around v_p .

The faces of $N_{P \oplus Q}$ are intersections of faces of N_P and N_Q . What is more, the dual face of $v_{P \oplus Q}$ is the intersection of the dual faces of v_P and v_Q with $v_P + v_Q = v_{P \oplus Q}$. As a consequence, the overlay of N_P and N_Q is the normal diagram of the Minkowski sum $P \oplus Q$. Using the overlay of the normal diagrams improves on the trivial algorithm in two points. First, we can obtain the set of vertices of $P \oplus Q$ easily from the overlay by computation of the vector sum of the supports of each face in the overlay. As a result, the construction of $P \oplus Q$ operates on a set of vertices that might be far smaller than pq . However, in the worst case, $P \oplus Q$ still has $O(pq)$ vertices. And second, The incidence structure of $N_{P \oplus Q}$ allows us to construct $P \oplus Q$ from it in time linear to $P \oplus Q$.

With Nef polyhedra embedded on the sphere, we can realize normal diagrams easily. Also we can reuse their overlay algorithm for the Minkowski sum. For this purpose, we store the coordinates of each primal vertex as the label of its dual sface. During the overlay, the selection function performs the vector sum on these coordinates. The sfaces of $N_{P \oplus Q}$ are labeled with the coordinates of their primal vertex.

In addition, we also store the usual set-selection marks in the labels of the normal diagrams, such that we can perform Minkowski sums on convex polyhedra with selected and unselected vertices, edges and facets. Without set-selection marks, the input polyhedra must either be considered as open or closed. With either input polyhedron being open, the Minkowski sum must also be open. Adding marks, the boundary of the Minkowski can become more complex. For instance, a vertex $v_{P \oplus Q}$ of the Minkowski sum, which is the vector sum of vertices v_P and v_Q , must be selected iff both v_P and v_Q are selected. In order to obtain the correct marks for all items of the Minkowski sum, we store the marks of P and Q as labels of their dual items and apply the and-operation in the selection step of the spherical overlay.

Summing up, we combine point coordinates and boolean set-selection marks to a new type `PointBool`, and store it with each item in the normal diagram. The new class defines the `operator&&`, which performs vector addition upon the points, and the and-operation upon the bools. For svertices and sedges, the point coordinates are meaningless; a default value is assigned.

Note that each side of the polyhedron is restricted to a single simple facet. We do not know how to handle a polyhedron side that consists of multiple facets with different selection marks, since each side of the polyhedron corresponds to a single vertex in the normal diagram. It is unclear, how the structure and the set-selection

marks of complex polygon sides can be encoded as attributes of a vertex in the normal diagram, or how to combine those attributes during a Boolean operation of two normal diagrams.

Having Nef polyhedra embedded on the sphere, this approach is quite simple and more efficient than the trivial algorithm. It computes the Minkowski sum in $O((p+q)\log(p+q)+r)$ time, where r is the complexity of the Minkowski sum. In addition to the trivial method, it can also handle selected and unselected boundaries. On the other hand, Fogel and Halperin showed that there are much faster methods [FH06]. They conducted experiments to compare the efficiency of their own implementation with the trivial solution, with Weibel's implementation of Fukuda's method [Fuk04], and with our implementation. There is no other known exact and robust implementation at the moment.

Fogel and Halperin implemented a *Cubical Gaussian map* [FH06], which projects the normal diagram onto the unit cube. They perform separate overlays for each side of the cube. The method runs in $O(r\log(p+q))$ time. Fukuda's method is based on linear programming. Its complexity is $O(\delta LP(3, \delta)V)$, where δ is the sum of the maximal degrees of vertices in the two input polytopes, V is the number of vertices of the resulting Minkowski sum, and $LP(d, m)$ is the time required to solve a linear programming in d variables and m inequalities. Note, the implementation of Fogel and Halperin is the only implementation specifically optimized for the computation of the Minkowski sum of convex three-dimensional polyhedra. Fukuda's algorithm is more general, as it can be used to compute the Minkowski sum of polytopes in an arbitrary dimension. Our binary operations on sphere maps can handle more complex overlays than those of normal diagrams, which are always convex arrangements; they never include nested faces or lower dimensional features.

The Cubical Gaussian Map proved to be much faster than the other implementations. In the conducted experiments, it was between 36 and 60 times faster than our implementation, and between 4 and 31 times faster than Weibel's implementation. For small instances, our implementation was even slower than the trivial method. But since the trivial method is not output-sensitive, our implementation is much faster in complex experiments.

10.2.2 The Vertical Decomposition of a 3D Nef polyhedron.

The problem of partitioning a polyhedron into convex pieces is more complex than its two-dimensional counterpart. In general it is not possible to decompose a polyhedron into simplices, i.e., into tetrahedra, without introducing Steiner

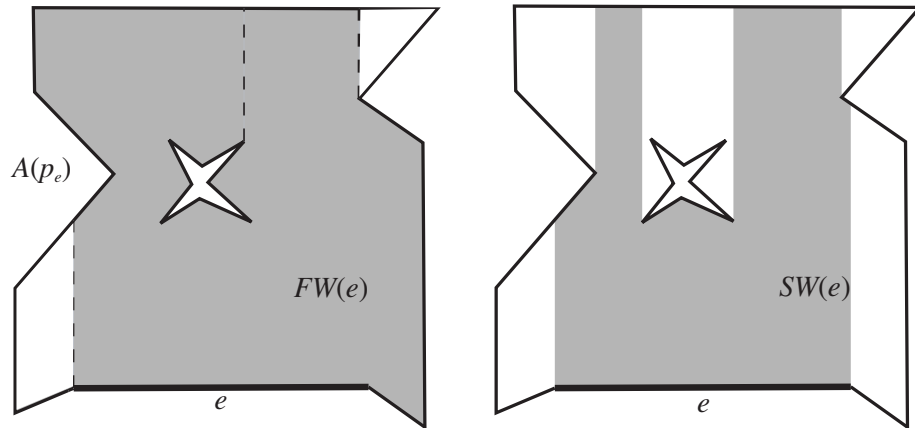


Figure 10.5: The vertical wall for the edge e according to the two definitions. The wall flood wall $FW(e)$ completely fills the intersection of the vertical plane p_e through e and the cell above e . The cell above e is formed by the original polyhedron and previously inserted walls (dashed lines). The sign wall $SW(e)$ covers all points that can be connected to e by a vertical edge without intersections.

points [O’R87]. The decomposition of a polyhedron into a minimum number of convex pieces is known to be NP-hard [O’R87]. Chazelle showed that a polyhedron with input complexity n and r reflex angles, i.e., angles larger than 180 degrees, can be decomposed into $O(r^2)$ convex pieces in $O(nr^3)$ time and $O(nr^2)$ space. He also provided an example for which the bound of $O(r^2)$ convex sub-polyhedra is tight [Cha84].

We choose to perform a vertical decomposition, which seems to be an intuitive and easy to implement decomposition method. We follow the common approach of adding vertical facets usually denoted as walls. This approach was introduced for the vertical decomposition of the three-dimensional space with respect to a set of triangles [AS88, dBGH94]. We adapt it for the decomposition of 3D Nef polyhedra.

A vertical wall $W(e)$ of some non-vertical edge e is a connected subset of the vertical plane p_e that supports e . There are two different definitions of vertical walls. In the following, we present both definitions together with the vertical decompositions based upon them. Then we introduce an easy method for creating vertical walls and discuss its applicability for the two decompositions.

Vertical walls were first defined by Aronov and Sharir [AS88]. Adapting their definition to our problem, vertical walls are defined as follows: Let $A(p_e)$ be the planar arrangement of the intersection of the polyhedron, including previously

erected walls, with p_e . Then, the vertical wall of e consists of all faces of $A(p_e)$ that are incident to e and inside the polyhedron. The left graphic of Figure 10.5 illustrates the planar arrangement $A(p_e)$ and the vertical wall of e . In order to distinguish this definition from the other definition, which will be given later, we denote such a vertical wall as a *flood wall* $FW(e)$.

With the definition of flood walls, a decomposition into convex pieces is rather simple. Erecting the flood wall of an *reflex edge* e , i.e., of an edge whose adjacent facets form a reflex angle, the wall divides the reflex angle into two or three non-reflex angles. Also, the wall does not introduce new reflex edges [AS88]. This way, a convex decomposition can be achieved by erecting flood walls for all reflex edges. In the degenerate case of a vertical reflex edge e_v , it suffices to consider the edge as slightly perturbed to determine a plane p_{e_v} together with the corresponding arrangement $A(p_{e_v})$. Either plane that supports e_v is appropriate.

De Berg, Guibas, and Halperin defined a vertical wall as the set of all points that can be connected to e via a vertical segment that does not intersect a face, edge, or vertex [dBGH94]. Adapting their definition for our purposes, we only consider those parts of the wall that lie within the polyhedron. Further on, we denote such a wall as the *sight wall* $SW(e)$. The right graphic of Figure 10.5 illustrates the definition.

Sight walls also divide reflex angles into non-reflex angles, but their vertical boundary edges may become new reflex edges. Some of them may be resolved by the other sight walls of the original reflex edges, but some may not be resolved.

In the original scenario, the decomposition of the three-dimensional space into convex cells with respect to a set of triangles, the decomposition works in two steps. In the first step, vertical walls are erected for all non-vertical edges. As a consequence, the three-dimensional space becomes subdivided into cylindrical cells, i.e., each cell is bounded by several vertical, convex facets, and by two equally shaped, non-vertical, not necessarily convex facets—one at the top and one at the bottom. Note that there are degenerate cases, where the top and bottom facet meet in a common vertex or edge. In the second step, further vertical walls are added. They are chosen in such a way, that they decompose the top and bottom facets of each cylindrical cell into convex sub-facets, and thereby they also decompose the cells into convex sub-cells. For this purpose, any common polygon decomposition method can be applied on the top and the bottom facet. For every edge inserted by the polygon decomposition, another vertical wall is erected. Figure 10.6 illustrates the two steps of the vertical decomposition by sight walls. Note that every vertical wall, that is created in the second step, is convex. As a result, we need not distinguish between flood walls and sight walls. They are the same.

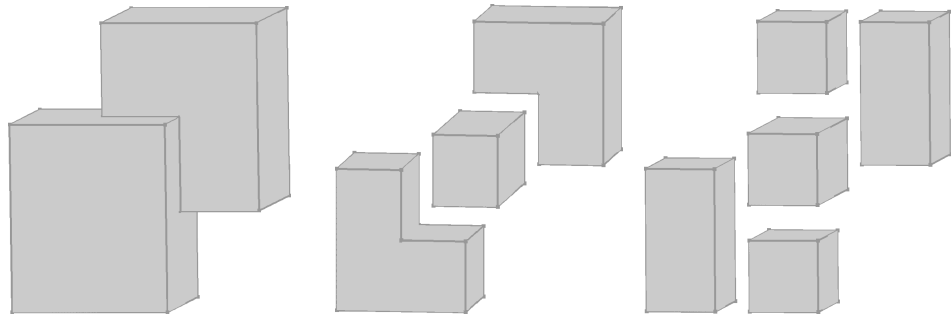


Figure 10.6: Vertical decomposition based on the insertion of sight walls (viewed from the top). In the first step, the polyhedron is decomposed into xy -monotone sub-cells. Then, further vertical walls are inserted to subdivide the cells into convex sub-cells.

In our scenario, where we decompose the selected volumes of a Nef polyhedron, it is more efficient to erect walls only for those edges, whose adjacent facets form a reflex angle. As a consequence, the cells that result from the first step are not cylindrical, but still xy -monotone, i.e., the intersection of a vertical line with a cell is either empty or connected. The cells are still bounded by multiple vertical facets, but the top and the bottom may consist of multiple facets, which form convex surface patches. Again, we can find proper vertical walls for the second step by decomposing a polygon. Projecting the top and bottom patch of a cell into the xy -plane, both projections form the same polygon. This polygon needs not to be convex. Vertical walls, whose projection decompose the polygon into convex sub-polygons, decompose the cell into convex sub-cells.

The decomposition based on sight walls implies two major advantages: the result of the decomposition does not depend on the order of the wall erection, and the decomposition yields fewer sub-polyhedra [dBGH94].

So far, we described how to partition the selected volumes of a Nef polyhedron. For holes in a volume we need no special treatment. If there are selected boundary parts in an unselected volume that do not enclose a selected volume, we handle each such facet, isolated edge, or isolated vertex as a separate sub-polyhedron.

But how do we create walls? Since we decompose volumes of a polyhedron enclosed by a shell, often it is fairly easy to insert a flood wall of some edge e in the outer shell of a volume. Starting from e , we walk along the intersection of the shell with the vertical plane p_e supporting e . During the walk, we adapt the sphere maps of the encountered vertices and create new vertices when the walk crosses an edge. The twin relation between halfedges can easily be updated. The walk

terminates when it returns to e . If this newly created facet cycle is an outer cycle of the wall, and there are no inner cycles, we only need to recompute the SNC in order to finalize the creation of the wall.

For the walk along the shell, we need ray shooting to detect the next intersection with the 1-skeleton of the shell. We use our kd-tree for this purpose, which we update every time a new vertex or a new edge is created. For this purpose, we add those objects into the proper leaf nodes of the kd-tree. Because the walk only needs the 1-skeleton and the kd-tree to be up-to-date, it is not necessary to recompute the SNC in a series of wall creations. It suffices to recompute it once at the end.

The walk is an easy solution to create the outer boundary of a wall, but it is not sufficient to create all walls properly. In the first wall creation phase, a wall may be intersected in its interior and therefore may contain holes. Since after the first wall creation phase, there are no inner cycles left, it seems like those boundary cycles are not necessary. But they become part of an outer cycle later in the decomposition process.

Another problem occurs, when a walk started from an edge e does not create outer cycles, but an inner cycles. This happens, when multiple shells intersect p_e , or one shell intersects p_e multiple times. In both cases the walk would create an inner facet cycle we are probably not interested in, because only in degenerate cases there is another edge e' that lies in the same plane $p_e = p_{e'}$, and that triggers the construction of the outer cycle of $W(e) = W(e')$.

We want to adjust the walk in such a way, that it creates sight walls instead of flood walls. Because of their definition, a sight wall is xy -monotone. As a result, there can be no holes in a sight wall; they only have an outer face cycle. The outer boundary of a sight wall $SW(e)$ is composed of three types of segments: intersections with the shells, intersections with other walls, and vertical segments from an endpoint of e to the first intersection with some boundary element. We denote segments of the third type as the *lateral delimiters* of $SW(e)$. The walk can be applied for the creation of $SW(e)$, if we can guarantee that the walls that are part of $SW(e)$'s boundary, further on denoted as the *prerequisite walls* of $SW(e)$, have been erected before. Then, we start by creating the lateral delimiters and perform two separate walks afterwards; one creates the lower, and the other the upper part of $SW(e)$. With lower and upper parts, we refer to the set of points of a sight wall $SW(e)$ that are vertically below or above e .

In general, the reflex edges of a selected volume cannot be sorted in such a way, that all prerequisite walls of some wall $SW(e)$ are erected before $SW(e)$ itself. There can be mutual and cyclic dependencies. The mutual dependencies can be resolved by first creating the lower parts of all sight walls, and then all upper parts,

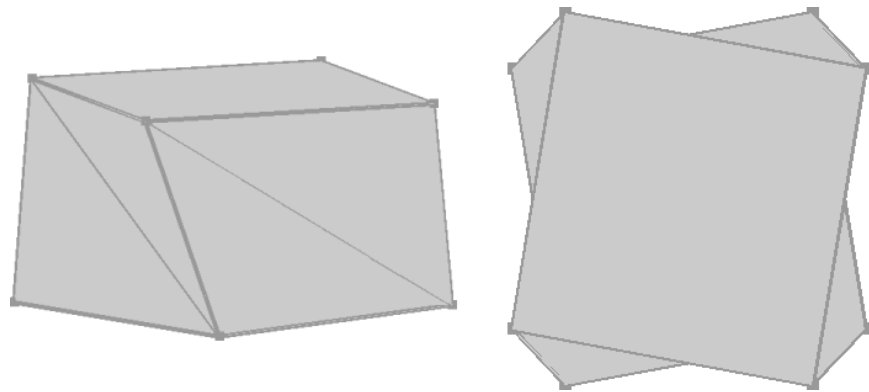


Figure 10.7: Variation of Schönhardt polyhedron with a quadratic base viewed from the side and from the top. The diagonals of the sides are reflex edges, which are circular dependent on one another.

or the other way around. But amongst all lower, and amongst all upper parts, there can still be cyclic dependencies. Figure 10.7 shows an example for cyclic dependencies.

To create the lower parts of the sight walls of all reflex edges, we sort the reflex edges by their smaller endpoint in ascending lexicographic order, resolve dependency problems, create the lateral delimiters for the lower parts, and finally apply the walk in the determined order. After sorting the reflex edges by their lower endpoint, we must resolve the situation, where e_1 is sorted in front of e_2 , because its lower endpoint is lexicographically smaller, but e_2 is a prerequisite wall of e_1 . The edges e_1 and e_2 have a common vertical intersection line l_v , which intersects e_1 in i_1 and e_2 in i_2 , where i_1 lies above i_2 . We split $e_2 = (s_2, t_2)$ at i_2 into $e'_2 = (s_2, i_2)$ and $e''_2 = (i_2, t_2)$, and insert both parts at the proper position of the sorted sequence of reflex edges. In consequence, e_1 is still created before both parts of e_2 , but does not depend on their prior erection. Instead of e_2 , the common vertical delimiter of e'_2 and e''_2 have become part of $SW(e_1)$'s boundary. The complete procedure for decomposing the selected volume of a Nef polyhedron into xy -monotone cells, is summarized in Algorithm 2.

For the second step of the decomposition—the decomposition of xy -monotone cells into convex sub-cells—we still need to specify how we find a proper set of vertical walls that divides the remaining reflex edges. Those remaining reflex edges either are vertical edges not handled in the first phase, or vertical boundary edges of the sight walls created in the first phase. We choose the following easy to adapt y -vertical polygon decomposition. For every *reflex vertex* v , i.e., for every vertex

Algorithm 2 Decomposition of a 3D Nef polyhedron into xy -monotone pieces.

```

1: procedure XY_MONOTONE_DECOMPOSITION( $N$ )
2:    $R = \emptyset$ 
3:   for all edges  $e$  of  $N$  do
4:     if !is_vertical( $e$ ) AND volume_below_is_selected( $e$ ) then
5:        $R = R \cup e$ 
6:     end if
7:   end for
8:   Sort edges in  $R$  by their smaller endpoints in ascending order
9:   Resolve dependencies in  $R$ 
10:  Insert lateral delimiters in  $-z$  direction
11:  for all edges  $e \in R$  do create_lower_part_of_sight_wall( $e$ )
12:  end for
13:   $R = \emptyset$ 
14:  for all edges  $e$  of  $N$  do
15:    if !is_vertical( $e$ ) AND volume_above_is_selected( $e$ ) then
16:       $R = R \cup e$ 
17:    end if
18:  end for
19:  Sort edges in  $R$  by their larger endpoints in descending order
20:  Resolve dependencies in  $R$ 
21:  Insert lateral delimiters in  $z$  direction
22:  for all edges  $e \in R$  do create_upper_part_of_sight_wall( $e$ )
23:  end for
24: end procedure

```

whose interior angle is bigger than 180 degrees, we insert either y -vertical edge that starts at v and crosses the polygon's interior. The reflex vertices of the 2D version correlate to the remaining reflex edges of the xy -monotone cells, and the y -vertical edges correlate to walls parallel to the yz -plane, where each wall divides the cell in two separate parts. We can easily create such a wall by starting our walk from a reflex edge in the proper direction. Summing up, we decompose an xy -monotone cell into convex pieces by creating either wall parallel to the yz -plane that divides a reflex edge.

10.2.3 Uniting a Set of 3D Polyhedra

The implementation of the main routine of the Minkowski sum is straight forward except for one point. The union of the Minkowski sums of the convex sub-polyhedra is resolved by multiple binary union operations. Here, it is essential not to perform the binary operations in arbitrary order. The complexity of our binary operation depends on the complexities of both input and the result polyhedron in equal shares. As a consequence, it is favorable uniting small polyhedra first. Since we cannot foresee the optimal order, we test three different strategies.

The trivial method maintains one Nef polyhedron holding the current intermediate result. It starts with an empty polyhedron and adds the polyhedra one by one. This method is expected to perform very badly, since most of the union operations involve at least one big polyhedron, namely the intermediate result.

The second method aims for more balanced operations. We initialize a queue with all polyhedra in arbitrary order. The method continuously takes the first two polyhedra from the queue, unites them, and appends the result. As the decomposition creates constant-sized sub-polyhedra, we can assume that we always unite polyhedra of similar size. The method finishes with the result left as the sole remaining item in the queue.

The third method refines the second one. Instead of a normal queue, we maintain a priority queue. The priority of a polyhedron is its size measured by the number of its vertices. As a result, each union operation is performed on the two smallest polyhedra in the priority queue. Again, the result of each union is inserted into the queue, and the method terminates with the result left as the final remaining element in the queue.

We compute the Minkowski sum of a two-manifold, triangulated mesh, which depicts a mushroom, with a cube. Our implementation decomposes the mushroom, which has 226 vertices, 672 facets, and 213 reflex edges, into 304 sub-polyhedra. We do not decompose the cube, which surely is convex. Figure 10.8 shows the

10.2. MINKOWSKI SUM OF TWO NEF POLYHEDRA

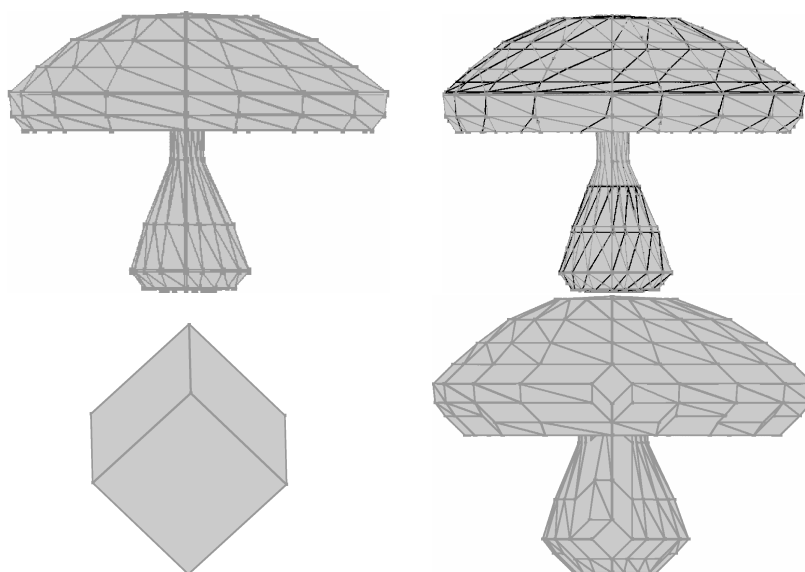


Figure 10.8: The bottom right picture shows the Minkowski sum of a mushroom (top left) and a cube (bottom left). The top right picture shows the mushroom vertically decomposed by vertical walls.

mushroom, the decomposed mushroom, the cube, and the Minkowski sum of the mushroom and the cube. The total runtime of the Minkowski sum and the runtimes of its major parts are listed in Table 10.1.

The fastest method computes the Minkowski sum in 106 seconds. As expected, the union of the intermediate results takes most of the time—about 75% with the fastest union method. Surprisingly, the queue outperformed the priority queue. We assume, that the order of the intermediate results in the queue was not arbitrary after all. It is likely, that the volumes are ordered in such a way, that the Minkowski

	trivial	queue	priority queue
decomposition	9s	10s	10s
convex sum	14s	17s	15s
union	269s	79s	104s
total	292s	106s	129s

Table 10.1: Runtime of the Minkowski sum of a mushroom and a cube. For the union of the Minkowski sums of the cube with each the sub-polyhedra of the mushroom, three methods are compared.

sums of neighbor cells in the decomposition reappear close to each other in the queue. The union of two such polyhedra has a lower complexity than the union of the same two polyhedra far apart from one another. Using a priority queue, the neighboring structure gets totally lost. Hence, the first unions unite polyhedra far apart. We conclude, that there is much to gain from an elaborate union strategy.

10.2.4 Limitations and Future Work

As already mentioned at the beginning of this section, our current implementation of the Minkowski sum is restricted to closed polyhedra. The reason is, that the sides of the convex sub-polyhedra returned by our decomposition may be complex, i.e., a side may consist of several selected and unselected facets. On the other hand, we do not know how to compute the Minkowski sum of convex sub-polyhedra with complex sides. Neither of the presented methods applies.

One approach to overcome this problem is a finer decomposition, such that each side of the convex sub-polyhedra is simple. This approach seems not effective, because a finer decomposition means that more convex sums, which also have larger combined complexity, have to be united at the end of the Minkowski sum computation. Hence, the union step, which already is the most time consuming step, would become slower. A more promising approach computes the Minkowski sum of the complex sides as a separate step. Here, well-examined solutions for the Minkowski sum of non-convex polygons can be applied [AFH02].

The test run of our Minkowski sum implementation showed that the secluded union step is the major bottleneck. We already examined three strategies for uniting a set of polyhedra by consecutive binary unions, and want to continue with more sophisticated strategies. Another way to reduce the time spent for the union is to implement an n -ary Boolean operation on 3D Nef polyhedra. It is not clear how efficient an n -ary operation can be. But the potential seems large in consideration of the many intermediate kd-trees and SNCs that can be spared. Essential for an efficient n -ary operation is a proper search data structure that identifies the locations of the candidate vertices. Also it should provide the set of input polyhedra whose boundary intersects the location of a candidate vertex. An n -ary operation surely is interesting for the visual hull computation, too.

Chapter 11

Conclusion

In this thesis we have presented data structures that realize a boundary representation of Nef polyhedra in three-dimensional space, together with algorithms for Boolean and topological operations on them. Our implementation has two features that improve on the polyhedron modelers currently on the market. First, it is exact, i.e., it always computes the correct result, is robust, and can handle all degeneracies. Second, its modeling space contains half-spaces and is closed under Boolean and topological operations. Consequently, we can represent non-manifold situations, open and closed boundaries, and mixed-dimensional features.

In December 2004, our implementation was released as Open Source software in the Computation Geometry Algorithm Library (CGAL) release 3.1. It supports the construction of Nef polyhedra from half-spaces and manifold solids, Boolean and topological operations, rotation by rational rotation matrices, translation and scaling. Furthermore, we provide visualization via QT.

11.1 Results

By performing and analyzing several experiments, we examined our binary operation routine and its major subroutines. We were able to confirm their worst-case runtime, as well as the runtime expected under the assumption of well-shaped geometry. The experiments showed that the runtime of the binary operations clearly diverges from the runtime expected under the assumption of well-shaped geometry if a polyhedron contains a facet of linear size. Thus, point-location and ray-shooting queries posed to our kd-tree can only be answered in time quadratic to the polyhedron's complexity.

Another set of experiments showed that our implementation can compete with the commercial CAD kernel ACIS R13. Usually ACIS is faster by a factor of about four, but in some situations we perform even better than ACIS, despite that exact arithmetic is used instead of floating-point arithmetic. Because of the exact arithmetic, our algorithms are robust even in scenarios where ACIS fails. We lose ground on ACIS in situations where a complex polyhedron is slightly altered by a small and simple polyhedron. In this case, we are about seven times slower than ACIS.

We realized two example applications: the visual hull of a three-dimensional polyhedron and the Minkowski sum of two closed Nef polyhedra. Both applications are not optimized; however, they demonstrate the potential of our polyhedron modeler. What is more, our implementation of the Minkowski sum is the first exact solution for three-dimensional non-convex polyhedra. At the moment we can only handle closed polyhedra, and therefore cannot handle tight passages.

11.2 Future work

The most important improvement to our implementation will be the adaptation for the effective use of floating point filters. We expect it to provide a major speed up. In addition to the faster arithmetic operations, categorizing items by indices will clearly be faster than categorizing them by geometric properties.

The kd-tree is an important tool for our application. It efficiently solves point-location and ray-shooting queries. On the other side, it often becomes the bottleneck of our implementation, especially if a polyhedron with a linear-sized facet is involved in a binary operation. In Section 9.6 we discussed several opportunities to improve on the point location and the ray shooting. One idea is to solve all point-location and ray-shooting queries needed in a binary operation batched with the help of a special streamed data structure as the first step of the binary operation. Also, it might be interesting to offer several ray-shooting and point-location strategies, which are exchangeable by the user between binary operations. This way, the user can choose the best strategy depending on the subsequent need for point location and ray shooting.

In addition, we want to improve our two applications. For both applications it will be interesting to develop efficient n -ary operations. With n -ary intersection operations and the efficient use of floating point filters, we want to make the visual hull competitive to its inexact counterparts. Also, we want to extend our implementation of the Minkowski sum to arbitrary Nef polyhedra in order to exactly compute configuration spaces with tight passages in three-dimensional space.

Bibliography

- [AFH02] P. K. Agarwal, E. Flato, and D. Halperin. Polygon decomposition for efficient construction of minkowski sums. *Computational Geometry: Theory and Application*, 21:39–61, 2002.
- [AR94] A. Agarwal and A. G. Requicha. A paradigm for the robust design of algorithms for geometric modeling. *Computer Graphics Forum*, 13(3):33–44, 1994.
- [AS88] B. Aronov and M. Sharir. Triangles in space or building (and analyzing) castles in the air. In *SCG '88: Proceedings of the fourth annual symposium on Computational geometry*, pages 381–391, New York, NY, USA, 1988. ACM Press.
- [Aut] Autodesk. *The Autodesk homepage*. <http://www.autodesk.com>.
- [Ben75] J. L. Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, September 1975.
- [BFS98] C. Burnikel, S. Funke, and M. Seel. Exact arithmetic using cascaded computation. In *Proceedings of the 14th Annual Symposium on Computational Geometry (SCG'98)*, pages 175–183, 1998.
- [Bie95] H. Bieri. Nef polyhedra: A brief introduction. *Computing Supplement*, 10:43–60, 1995.
- [Bie96] H. Bieri. Two basic operations for Nef polyhedra. In *CSG 96: Set-theoretic Solid Modelling: Techniques and Applications*, pages 337–356. Information Geometers, April 1996.
- [BMP94] M. Benouamer, D. Michelucci, and B. Peroche. Error-free boundary evaluation based on a lazy rational arithmetic: a detailed implementation. *Computer-Aided Design*, 26(6), 1994.

- [BN88] H. Bieri and W. Nef. Elementary set operations with d -dimensional polyhedra. In *Proceedings on International Workshop on Computational Geometry on Computational Geometry and its Applications*, LNCS 333, pages 97–112. Springer Verlag, 1988.
- [BO79] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, September 1979.
- [BPP95] G. Bell, A. Parisi, and M. Pesce. Vrm1 the virtual reality modeling language: Version 1.0 specification. <http://www.web3d.org/>, May 26 1995. Third Draft.
- [BR96] R. Banerjee and J. Rossignac. Topologically exact evaluation of polyhedra defined in CSG with loose primitives. *Computer Graphics Forum*, 15(4):205–217, 1996.
- [BS04] Jasmin Blanchette and Mark Summerfield. *C++ GUI Programming with Qt 3*. PH, 2004.
- [CDR92] J. Canny, B. R. Donald, and E. K. Ressler. A rational rotation method for robust geometric algorithms. In *Proceedings of the 8th annual Symposium on Computational Geometry (SCG '92)*, pages 251–260, 1992.
- [CGA] *The CGAL Homepage*. <http://www.cgal.org/>.
- [Cha84] B. Chazelle. Convex partitions of polyhedra: a lower bound and worst-case optimal algorithm. *SIAM Journal on Computing*, 13(3):488–507, 1984.
- [CLR90] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.
- [Das] Dassault Systèmes. *The Dassault Systèmes homepage*. <http://www.3ds.com/>.
- [DBGH94] M. de Berg, L.J. Guibas, and D. Halperin. Vertical decompositions for triangles in 3-space. In *SCG '94: Proceedings of the tenth annual symposium on Computational geometry*, pages 1–10, New York, NY, USA, 1994. ACM Press.
- [dBvKOS97] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Verlag, 1997.

BIBLIOGRAPHY

- [DMY93] K. Dobrindt, K. Mehlhorn, and M. Yvinec. A complete and efficient algorithm for the intersection of a general and a convex polyhedron. In *Proceedings of the 3rd Workshop on Algorithms and Data Structures*, LNCS 709, pages 314–324, 1993.
- [EO85] H. Edelsbrunner and M. H. Overmars. Batched dynamic solutions to decomposable searching problems. *Journal of Algorithms*, 6:515–542, 1985.
- [FGK⁺00] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. – Pract. Exp.*, 30(11):1167–1202, 2000.
- [FH95] U. Finke and K.H. Hinrichs. Overlaying simply connected planar subdivisions in linear time. In *SCG '95: Proceedings of the eleventh annual symposium on Computational geometry*, pages 119–126, New York, NY, USA, 1995. ACM Press.
- [FH06] E. Fogel and D. Halperin. Exact and efficient construction of minkowski sums of convex polyhedra with applications. In *7th Workshop on Algorithm Engineering and Experiments (ALENEX 06)*, 2006. to appear.
- [For97] S.J. Fortune. Polyhedral modelling with multiprecision integer arithmetic. *Computer-Aided Design*, 29:123–133, 1997.
- [Fuk04] K. Fukuda. From the zonotope construction to the minkowski addition of convex polytopes. *Journal of Symbolic Computation*, 38(4):1261–1272, 2004.
- [FW93] S.J. Fortune and C.J. Van Wyk. Efficient exact arithmetic for computational geometry. In *SCG '93: Proceedings of the ninth annual symposium on Computational geometry*, pages 163–172, New York, NY, USA, 1993. ACM Press.
- [FW96] S.J. Fortune and C.J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, July 1996.
- [GCP90] E.L. Gursoz, Y. Choi, and F.B. Prinz. Vertex-based representation of non-manifold boundaries. *Geometric Modeling for Product Engineering*, 23(1):107–130, 1990.

- [GHJV95] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
- [gpr] The GNU profiler homepage. <http://www.gnu.org/software/binutils/manual/gprof-2.9.1/>.
- [Hal02] D. Halperin. Robust geometric computing in motion. *Int. J. of Robotics Research*, 21(3):219–232, 2002.
- [Hav00] Vlastimil Havran. *Heuristic Ray Shooting Algorithms*. Ph.d. thesis, Department of Computer Science and Engineering, Faculty of Electrical Engineering, Czech Technical University in Prague, November 2000.
- [Hof89] Christoph M. Hoffmann. *Geometric and Solid Modeling – An Introduction*. Morgan Kaufmann, 1989.
- [HPKS02] S. Hert, T. Polzin, L. Kettner, and G. Schäfer. Explab - a tool set for computational experiments. Research Report MPI-I-2002-1-004, MPI für Informatik, Saarbrücken, Germany, 2002.
- [HSW01] M. Hemmer, E. Schömer, and N. Wolpert. Computing a 3-dimensional cell in an arrangement of quadrics: Exactly and actually! In *ACM Symp. on Comp. Geom.*, pages 264–273, 2001.
- [HW96] J. Hartman and J. Wernecke. *The VRML 2.0 Handbook: Building Moving Worlds on the Web*. Addison-Wesley, 1996.
- [JLM00] M. Jazayeri, R. Loos, and D.R. Musser, editors. *Generic Programming, International Seminar on Generic Programming, Dagstuhl Castle, Germany, April 27 - May 1, 1998, Selected Papers*, volume 1766 of *Lecture Notes in Computer Science*. Springer, 2000.
- [Kar89] M. Karasick. *On the Representation and Manipulation of Rigid Solids*. Ph.D. thesis, Dept. Comput. Sci., McGill Univ., Montreal, PQ, 1989.
- [Ket99] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry: Theory and Applications*, 13:65–90, 1999.
- [KKM97] J. Keyser, S. Krishnan, and D. Manocha. Efficient and accurate B-rep generation of low degree sculptured solids using exact arithmetic. In *Proc. ACM Solid Modeling*, 1997.

BIBLIOGRAPHY

- [KLN91] M. Karasick, D. Lieber, and L. R. Nackman. Efficient delaunay triangulation using rational arithmetic. *ACM Trans. Graph.*, 10(1):71–91, 1991.
- [KM76] K. Kuratowski and A. Mostowski. *Set Theory*. North-Holland Publishing Co., 1976.
- [KMP⁺04] Lutz Kettner, Kurt Mehlhorn, Sylvain Pion, Stefan Schirra, and Chee Yap. Classroom examples of robustness problems in geometric computations. In *ESA*, pages 702–713, 2004.
- [Lat91] J.-C. Latombe. *Robot Motion Planning*. Kluwer Academic Publishers, Norwell, MA, USA, 1991.
- [Lau94] A. Laurentini. The visual hull concept for silhouette-based image understanding. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(2):150–162, 1994.
- [LPY04] Chen Li, Sylvain Pion, and Chee Yap. Recent progress in exact geometric computation. *J. of Logic and Algebraic Programming*, 64(1):85–111, 2004. Special issue on “Practical Development of Exact Real Number Computation”.
- [Män88] M. Mäntylä. *An Introd. to Solid Modeling*. Comp. Science Press, Rockville, Maryland, 1988.
- [Mid94] A. E. Middleditch. “The bug” and beyond: A history of point-set regularization. In *CSG 94 Set-theoretic Solid Modelling: Techn. and Appl.*, pages 1–16. Inform. Geom. Ltd., 1994.
- [Mil00] Victor Milenkovic. Shortest path geometric rounding. *Algorithmica*, 27(1):57–86, 2000.
- [MN94] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *Proceedings of the 13th IFIP World Computer Congress*, volume 1, pages 223–231. Elsevier Science B.V. North-Holland, Amsterdam, 1994.
- [MN99] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
- [MS03] K. Mehlhorn and M. Seel. Infimaximal frames: A technique for making lines look like segments. *International Journal of Computational Geometry and Application*, 13(3):241–255, 2003.

- [Mul90] K. Mulmuley. A fast planar partition algorithm, i. *J. Symb. Comput.*, 10(3-4):253–280, 1990.
- [Mye95] Nathan C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.
- [Nef78] W. Nef. *Beiträge zur Theorie der Polyeder*. Herbert Lang, Bern, 1978.
- [O’R87] J. O’Rourke. *Art Gallery Theorems and Algorithms*. Oxford University Press, Inc., New York, NY, USA, 1987.
- [Phi96] M. Phillips. *Geomview Manual, Version 1.6.1 for Unix Workstations*. The Geometry Center, University of Minnesota, 1996. <http://www.geom.umn.edu/software/download/geomview.html>.
- [Req80] Aristides G. Requicha. Representations for rigid solids: Theory, methods, and systems. *ACM Computing Surveys*, 12(4):437–464, 1980.
- [RO89] J. R. Rossignac and M. A. O’Connor. SGC: A dimension-independent model for pointsets with internal structures and incomplete boundaries. In M. Wozny, J. Turner, and K. Preiss, editors, *Geometric Modeling for Product Engineering*. North-Holland, 1989.
- [RR] J. R. Rossignac and A. G. Requicha. Solid modeling. <http://citeseer.nj.nec.com/209266.html>.
- [Sam90a] H. Samet. *Applications of Spatial Data Structures: Computer Graphics, Image Processing, and GIS*. Addison-Wesley, Reading, MA, 1990.
- [Sam90b] H. Samet. *The Design and Analysis of Spatial Data Structures*. Addison-Wesley, 1990.
- [See01a] M. Seel. Implementation of planar Nef polyhedra. Research Report MPI-I-2001-1-003, MPI für Informatik, Saarbrücken, Germany, August 2001.
- [See01b] M. Seel. *Planar Nef Polyhedra and Generic Higher-dimensional Geometry*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, 5. December 2001.
- [Spa04] Spatial Corp., A Dassault Systèmes company. *ACIS R13 Online Help*, 2004.

BIBLIOGRAPHY

- [Sto91] J. Stolfi. *Oriented Projective Geometry: A Framework for Geometric Computations*. Academic Press, New York, NY, 1991.
- [Tro] Trolltech. *Trolltech—Cross-platform C++ Gui Development, and Embedded Linux Solutions*. <http://www.trolltech.com/>.
- [UGS] UGS. *UGS: Product Lifecycle Management (PLM) Solutions*. <http://www.ugs.com/>.
- [VM04] G. Varadhan and D. Manocha. Accurate minkowski sum approximation of polyhedral models. In *PG '04: Proceedings of the Computer Graphics and Applications, 12th Pacific Conference on (PG'04)*, pages 392–401, Washington, DC, USA, 2004. IEEE Computer Society.
- [VRM96] The virtual reality modeling language specification: Version 2.0, ISO/IEC CD 14772. <http://www.web3d.org/>, August 4 1996.
- [Wei88] K. Weiler. The radial edge structure: A topological representation for non-manifold geometric boundary modeling. In M. J. Wozny, H. W. McLaughlin, and J. L. Encarnaçao, editors, *Geom. Model. for CAD Appl.*, pages 3–36. IFIP, May 12–16 1988.
- [Wer94] J. Wernicke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2*. Addison-Wesley, 1994.
- [Yap97] C. Yap. Towards exact geometric computation. *Computational Geometry: Theory and Applications*, 7(1):3–23, 1997.
- [YD95] C. K. Yap and T. Dubé. The exact computation paradigm. In D.-Z. Du and F. K. Hwang, editors, *Computing in Euclidean Geometry*, volume 4 of *Lecture Notes Series on Computing*, pages 452–492. World Scientific, Singapore, 2nd edition, 1995.
- [ZE02] A. Zomorodian and H. Edelsbrunner. Fast software for box intersection. *Int. J. Computational Geometry and Applications*, 12:143–172, 2002.