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

Peter Hachenberger, Lutz Kettner, Kurt Mehlhorn

Max Planck Institut Informatik, Stuhlsatzenhausweg 85, 66123 Saarbrücken, Germany

Abstract

Nef polyhedra in d-dimensional space are the closure of half-spaces under boolean set operations. In consequence, 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. They were introduced by W. Nef in his seminal 1978 book on polyhedra. The generality of Nef complexes is essential for some applications.

In this paper, 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.

The implementation was released as Open Source in the Computational Geometry Algorithm Library (CGAL) release 3.1 in December 2004.

1 Introduction

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 [5, 14, 24, 30, 38]. We restrict ourselves to partitions of the three-dimensional space into cells induced by planes. A set of planes partitions space into cells of various dimensions. Each cell may carry a label. We call such a partition together with the labelling of its cells a *selective Nef complex (SNC)*. When the labels are boolean ($\{in, out\}$) the complex describes a set, a

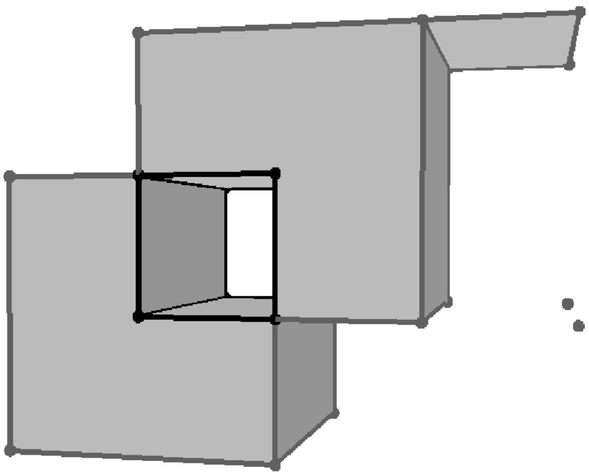


Fig. 1. A Nef polyhedron with non-manifold edges, a dangling facet, and two isolated vertices. The tunnel boundary does not belong to this point set.

so-called *Nef polyhedron* [36], and we call the labels *set-selection marks*. Nef polyhedra can be obtained from half-spaces by boolean operations union, intersection, and complement. Nef complexes slightly generalize Nef polyhedra through the use of a larger set of labels. Figure 1 shows a Nef polyhedron.

Nef polyhedra and complexes are quite general. They can model non-manifold solids, unbounded solids, open and closed sets, and objects comprising parts of different dimensionality. Is this generality needed?

(1) Nef polyhedra are the smallest family of solids containing the half-spaces and being closed under boolean operations. In particular, boolean operations may generate non-manifold solids, e.g., the symmetric difference of two cubes in Figure 1, and lower dimensional features. The latter can be avoided with regularized operations.

(2) Non-manifold edges arise frequently in polyhedra that model the interior of three-dimensional objects, e.g., in a geometric earth model or in a model of the interior of an integrated circuit. The properties of different cells—in an earth model these cells may represent soil layers, reservoirs, or faults—can be distinguished with labels. Many of these cells can share a common edge.

(3) In machine tooling, we may want to build a polyhedral object Q by a cutting tool M. 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, its *configuration space*, is the set $C = \{p;\ p+M \cap Q = \emptyset\}$; C may also contain lower dimensional features. Here, it is also convenient to have the distinction between open and closed sets available with Nef complexes as shown in Figure 2. This is one of the examples where Middleditch [33] argues that we need more than regularized boolean operations, i.e., we need to concurrently model objects of different dimensionality and with open and closed boundaries. In the context of robot motion planning this example is referred to as *tight passage*, see [20] for the case of planar configuration spaces.

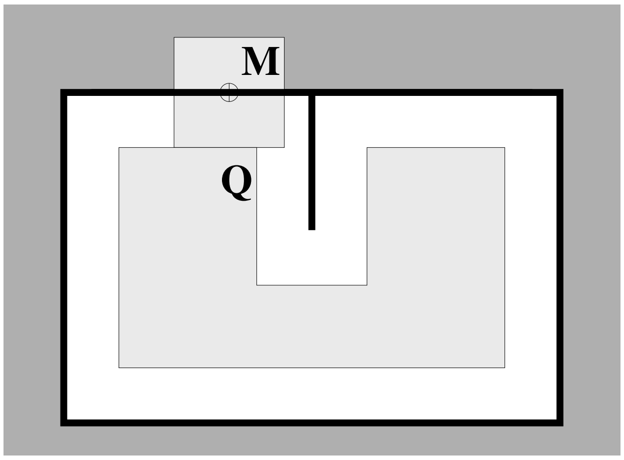


Fig. 2. The width of the cutter M is equal to the width of the cavity in Q. The boundary of the region of legal placements is shown in bold. It is an unbounded polygon with a dangling edge.

SNCs are closed under (generalized) set operations. If C_1 and C_2 are Nef complexes, we obtain a new SNC by intersecting every cell of C_1 with every cell of C_2 and labeling the resulting cell (if non-empty) with the pair of labels of its parent cells. Also, if f is a mapping from one label space to another, relabeling cells by f generates a new SNC.

Clearly, SNCs can be represented by the underlying plane arrangement plus the labeling of its cells. This representation is space-inefficient if adjacent cells frequently share the same label, and it is time-inefficient since navigation through the structure is difficult.

We give a more compact and unique representation of SNCs, algorithms realizing the (generalized) set operations based on this representation, *and an efficient implementation*. The uniqueness of the representation, going back to Nef's work [36], is worth emphasizing; two point sets are the same if and only if they have the same representation.

The current implementation supports the construction of Nef polyhedra from manifold solids, boolean operations (union, intersection, complement, difference, symmetric difference), topological operations (interior, closure, boundary), and rotations by rational rotation matrices (arbitrary rotation angles are approximated up to a specified tolerance [9]). Our implementation is exact. We follow the exact computation paradigm to guarantee correctness. All Nef polyhedra shown in the figures are created with our implementation, e.g., Figure 3 illustrates correctness.

Our representation and algorithms refine the results of Rossignac and O'Connor [37], Weiler [43], Gursoz, Choi, and Prinz [17], and Dobrindt, Mehlhorn, and Yvinec [12]; see Section 13 for a detailed comparison.

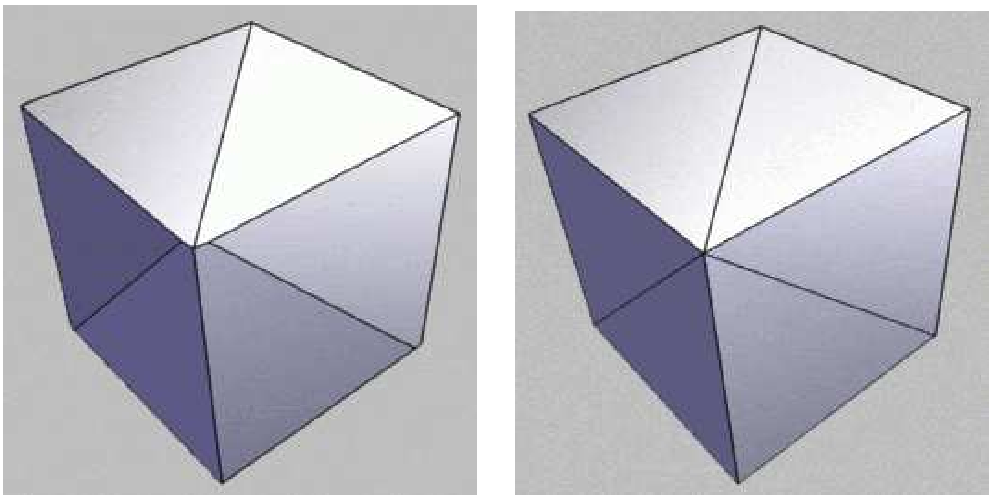


Fig. 3. The figure on the left shows the intersection of two cubes where one is rotated by five degrees around each coordinate axis. Observe that the front corner has three clearly distinguishable vertices. In the figure on the right we used 0.01 degree rotations. The three vertices in the front corner are not distinguished in the drawing, but the edges illustrate the correctness of the solution, e.g., none of the faces became coplanar.

The work presented here follows the PhD thesis of the first author [18], and parts of it appeared as two extended abstracts in [16, 19].

1.1 Our Results

We present a complete data structure for the representation of Nef polyhedra, together with correct and efficient algorithms for boolean and topological operations on Nef polyhedra. Our implementation was released as Open Source in the Computational Geometry Algorithm Library (CGAL) release 3.1 in December 2004. As a CGAL package, our implementation is documented, and tested by extensive regression tests in a test suite.

In a first implementation step, we emphasized completeness in the algorithm design. Then, we started optimizing the performance of our implementation with replacing some trivially implemented subroutines by sophisticated search data structures. In detail, we added a kd-tree [4, 21] for point location and ray shooting, and a fast box-intersection algorithm [45] for intersection finding. Both choices are known to be efficient in practice, but they are heuristics in the sense that they make assumptions about a nice distribution of the polyhedron faces. In the second series of optimizations we now evaluate the performance of our implementation, and add optimized implementations for important common cases. We evaluate the effectiveness of the optimizations individually and combined in a series of experiments.

Our current optimized implementation has become efficient enough to compare it with commercial systems. We selected ACIS R13, a common commercial CAD kernel used in many CAD systems [41]. A robustness experiment shows where the benefits of our exact and robust implementation lie. Furthermore, we run the same experiments that we have used for our implementation with ACIS, and we are pleased to see similar runtime results. We thereby demonstrate that our implementation based on such unpopular choices as exact arithmetic and a general full-fledged non-manifold data structure with sound theory can achieve industry-strength efficiency.

> *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

1.2 Structure

The paper is structured as follows: Nef polyhedra are reviewed in Section 2, our data structure is defined in Section 3, and the algorithms for generalized set operations are described in Section 4. We discuss their complexity in Section 5. In Section 6, we give details about the implementation, and about the design in Section 7. Then we perform several experiments. We use them to get an impression of the general runtime behavior of the binary operation and its subroutines in Section 8. In Section 9 we stress the most important subroutines with worst-case situations and thereby confirm our theoretical analysis of their complexity. We describe the optimizations for common cases and their experimental evaluation in Section 10, and compare our implementation with ACIS R13 in Section 11. In a final experiment, we analyze the impact of cascaded constructions on the bit complexity of the coordinate representation, and thus on the performance of our implementation described in Section 12. We relate our work to previous work in Section 13 and offer a short conclusion in Section 14.

2 Theory of Nef Polyhedra

We repeat a few definitions and facts about Nef polyhedra [36] that we need for our data structure and algorithms. The definitions here are presented for arbitrary dimensions, but we restrict ourselves in the sequel to three dimensions.

Definition 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, difference and symmetric difference can be reduced to intersection and complement. 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.

Definition 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* $\mathrm{Pyr}_P(x)$.

Definition 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 $\mathrm{Pyr}_P(x) = \mathrm{Pyr}_P(y)$. The equivalence classes of $\sim$ are the* faces *of P. The dimension of a face s is the dimension of its affine hull,* $\dim s := \dim \mathrm{aff}\, s$.

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 $\mathrm{Pyr}_P(s)$. 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.

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 one whose local pyramid is the empty space. 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.

Example 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 4 illustrates the polyhedron with its partially closed and partially open boundary, i.e., vertices v_4, v_5, and v_6, and edges e_4 and e_5 are not part of P. The local pyramids for the faces are $\mathrm{Pyr}_P(f_1) = \emptyset$ and $\mathrm{Pyr}_P(f_2) = \mathbb{R}^2$. Examples for the local pyramids of edges are the closed half-space h_2 for the edge e_1, $\mathrm{Pyr}_P(e_1) = h_2$, and the open half-space that is the complement of h_4 for the edge e_5, $\mathrm{Pyr}_P(e_5) = \{(x,y) | x - y < 1\}$. The edge e_3 consists actually of two disconnected parts, both with the same local pyramid

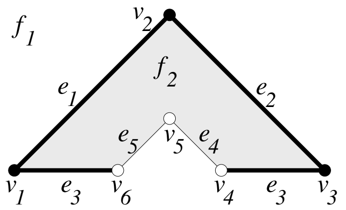


Fig. 4. Planar example of a Nef-polyhedron. The shaded region, bold edges and black nodes are part of the polyhedron, thin edges and white nodes are not.

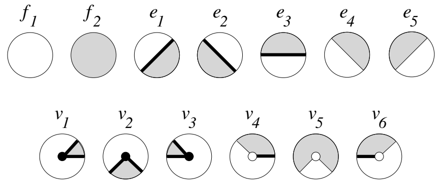


Fig. 5. Sketches of the local pyramids of the planar Nef polyhedron example. The local pyramids are indicated as shaded in the relative neighborhood in a small disc.

$\mathrm{Pyr}_P(e_3) = h_1$. *However, as explained later, in our data structure, we will represent the two connected components of the edge e_3 separately. Figure 5 illustrates all local pyramids for this example.*

Definition 5 (Incidence relation) *A face s is* incident *to a face t of a polyhedron P iff $s \subset closure(t)$. This defines a partial ordering $\prec$ such that $s \prec t$ iff s is incident to t.*

Bieri and Nef proposed several data structures for storing Nef polyhedra in arbitrary dimensions. In the *Würzburg structure* [8], 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*. For bounded polyhedra all minimal elements are vertices. 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 geometry. For example, they record the faces incident to an edge, but they do not record their cyclic ordering around the edge.

Bieri and Nef also proposed algorithms for computing Boolean and topological operations on d-dimensional Nef polyhedra [6, 8]. These operations recursively perform sweeps in dimensions $1, \ldots, d$. For planar Nef polyhedra their approach is similar to the one of Seel [39, 40]. The algorithm is given as pseudo-code, but details about the data structure, its transversal, and the complexity of the algorithm are missing. To our knowledge, the algorithm has never been realized.

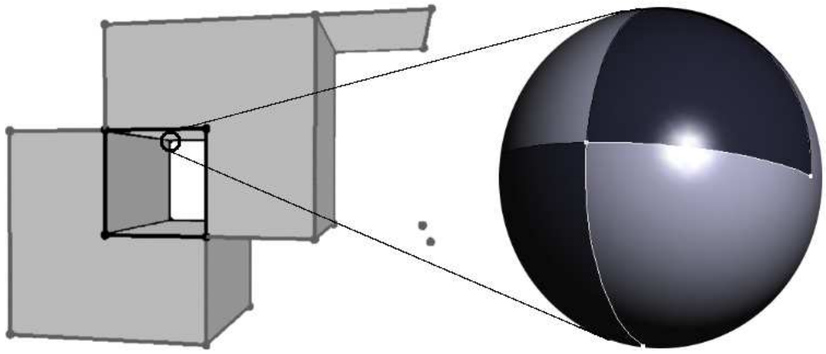


Fig. 6. An example of a sphere map. The different colors indicate selected and unselected faces.

3 Data Structures

In our representation for three-dimensions, we use two main structures: sphere Maps to represent the local pyramids of each vertex and the selective Nef complex representation to organize the local pyramids into a more easily accessible polyhedron boundary representation. It is convenient (conceptually and, in particular, in the implementation) to only deal with bounded polyhedra; a reduction from arbitrary polyehdra to bounded polyhedra is described in Section 3.3.

3.1 Sphere Map

The local pyramids of each vertex are represented by conceptually intersecting the local neighborhood with a small ε-sphere. This intersection forms a planar map on the sphere (Figure 6), which together with the set-selection mark for each item forms a two-dimensional Nef polyhedron embedded in the sphere. We add the set-selection mark for the vertex, i.e., the center of the sphere, and call the resulting structure the *sphere map* of the vertex. Sphere maps were introduced previously in [12].

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, the edge forms a great arc that is part of the great circle in which the supporting plane of the facet intersects the sphere. If a single facet intersects the sphere in a great circle, we get an *sloop* going around the sphere without any incident svertex. There is at most one *sloop* per sphere map because a second *sloop* would intersect the first. An *sface* corresponds to a volume. This representation extends the planar Nef polyhedron representation [39].

Having sphere maps for all vertices of a bounded polyhedron is a sufficient, but not easily accessible representation of the polyhedron. 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 below; see also Figure 7:

Edges: We store two oppositely oriented edges for each edge and have a pointer from one oriented edge to its opposite edge. Such an oriented edge 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.

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 identify an edge-use with an oriented *sedge* in the sphere map, or, in the special case also with an *sloop*. Without mentioning it explicitly in the remainder, all references to an *sedge* can also refer to an *sloop*.

Facets: We store oriented facets 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 facet, i.e., by using the opposite edge-use.

Shells: A 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 trace a shell from one entry element with a graph search. We offer this graph traversal in a visitor design pattern to the user.

Volumes: A volume is defined by a set of shells, one outer shell containing the volume and several (or maybe none) inner shells excluding voids from the volume. We store an arbitrary sface of each shell as an entry point for the graph traversal.

For each face we store a label, e.g., a set-selection mark, which indicates whether the face is part of the solid or if it is excluded. We call the resulting data structure *selective Nef complex*, *SNC* for short.

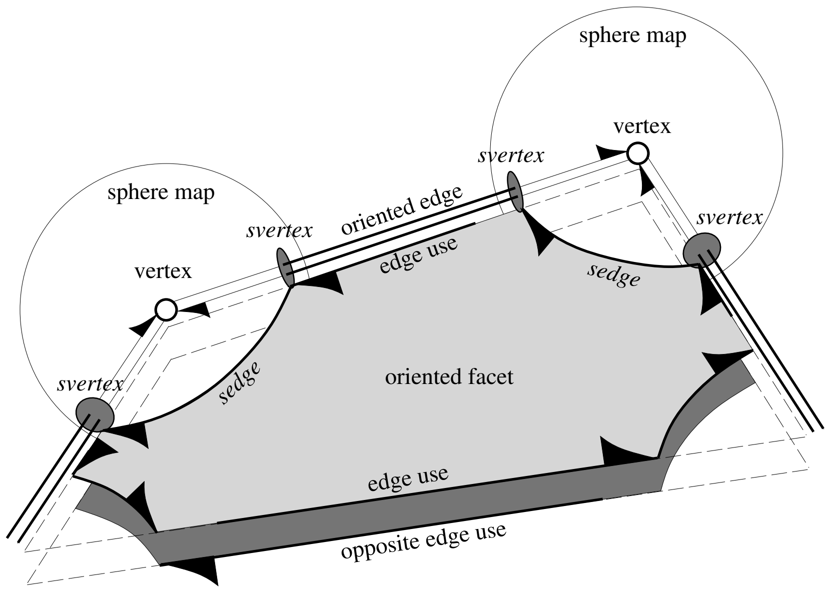


Fig. 7. An SNC. We show only one facet with four vertices, the sphere maps of two of the vertices, the connecting edges, and both oriented facets. Shells and volumes are omitted for this example.

3.3 Bounding Nef Polyhedra

In this section, we present a reduction to 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 unbounded Nef polyhedra, too.

We extend infimaximal frames [32] already used for planar Nef polygons [39, 40]. 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. In [32] it is argued that interpreting R as an infimaximal number instead of setting it to a large concrete number has several advantages, in particular increased efficiency and convenience.

Clipping lines and rays at this infimaximal box leads to points on the box that we call *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 one coordinate axis, and linear functions $f(R)$ for the other coordinates. We use linear polynomials over R as coordinate representation for standard points as well as for non-standard points, thus unifying the two kind of points in one representation, the *extended points*. In Lemma 6, we show that this representation is always sufficient, even in iterated constructions. For better readability of this section, we moved Lemma 6 into the appendix.

Analogous to extended points, we can define *extended segments* and *extended planes* as the unified representation of standard and non-standard segments or planes, respectively. Note that lines clipped at the infimaximal box become standard segments with non-standard endpoints; the supporting line can be still be represented by normal line equations. Non-standard segments arise from clipping half-spaces or planes at the infimaximal box. Extended planes only occur as the supporting planes of the sides of the infimaximal box.

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 non-vanishing coefficient determines their signs.

We 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; a proof is given in the appendix.

4 Algorithms

We describe the algorithms for constructing sphere maps for a polyhedron, the corresponding SNC, and the simple algorithm that follows from these data structures for performing boolean operations on polyhedra.

4.1 Construction of a Sphere Map

We extended the implementation of the planar Nef polyhedra in CGAL to the sphere map. We summarize the implementation of planar Nef polyhedra as described in [39, 40] and explain the changes needed here.

The boolean operations on the planar Nef polyhedra work in three steps—overlay, selection, and simplification—following [37]. The overlay computes the conventional planar-map overlay of the two input polyhedra with a sweep-line algorithm [31, section 10.7]. In the result, each face in the overlay is a subset of a face in each input polyhedron, which we call the support of that face. The selection step computes the mark of each face in the overlay by evaluating the boolean expression on the two marks of the corresponding two supports. This can be generalized to arbitrary functions on label sets. Finally, the simplification step has to clean up the data structure and remove redundant representations.

In particular, the simplification in the plane works as follows: (i) if an edge has the same mark as its two surrounding regions the edge is removed and the two regions are merged together; (ii) if an isolated vertex has the same mark as its surrounding region the vertex is removed; (iii) and if a vertex is incident to two collinear edges and all three marks are the same then the vertex is removed and the two edges are merged. The simplification is based on Nef's theory [7, 36] that provides a straightforward classification of point neighborhoods; the simplification just eliminates those neighborhoods that cannot occur in Nef polyhedra. The merge operation of regions in step (i) uses a union find data structure [11] to keep track of merged regions. Thus, an update of the pointers in the half-edge data structure associated with the regions is not needed after every single merge. It suffices to update the pointers once at the end of the simplification.

We extend the planar implementation to sphere maps in the following way: We (conceptually) cut the sphere into two hemispheres and rotate a great arc around each hemisphere instead of a sweep line in the plane. The running time of the sphere sweep is $O((n+m+s)\log(n+m))$ for sphere maps of size n and m respectively and an output sphere map of size s. Instead of actually representing the sphere map as geometry on the sphere, we use three-dimensional vectors for the *svertices*, and three-dimensional plane equations for the support of the *sedges*. Step (iii) in the simplification algorithm needs to be extended to recognize the special case where an *sloop* is obtained as a result.

4.2 Classification of Local Neighborhoods and Simplification in 3D

In order to understand the three-dimensional boolean operations and to extend the simplification algorithm from planar Nef polyhedra to three dimensions, it is useful to classify the topology of the *local neighborhood* of a point x (the sphere map that represents the intersection of the solid with the sphere plus the mark at the center of the sphere) with respect to the dimension of a Nef face that contains x. It follows from Nef's theory [7, 36] that:

- x is part of a volume iff its local sphere map is trivial (only one *sface* f^s with no boundary) and the mark of f^s corresponds to the mark of x.
- x is part of a facet f iff its local sphere map consists just of an *sloop* l^s and two incident *sfaces* f_1^s, f_2^s, the mark of l^s is the same as the mark of x, and at least one of f_1^s, f_2^s has a different mark.
- x is part of an edge e iff its local sphere map consists of two antipodal *svertices* v_1^s, v_2^s that are connected by a possible empty bundle of *sedges*. The *svertices* v_1^s, v_2^s and x have the same mark, which is different from the mark of at least one *sedge* or *sface* in between.
- x is a vertex v iff its local sphere map is none of the above.

Note that the classification of x is only valid if the sphere map was simplified; see the previous section.

Of course, a valid SNC will only contain sphere maps corresponding to vertices. But some of the algorithms that follow will modify the marks and potentially invalidate this condition. We extend the simplification algorithm from planar Nef polyhedra to work directly on the SNC structure. Based on the above classification and similar to the planar case, we identify redundant faces, edges, and vertices, we delete them, and we merge their neighbors.

For example, if a redundant facet has the same mark as its two neighboring volumes (that actually could be the same volume) then we remove the facet and merge the two volumes. Again, a union find data structure helps updating the pointers associated with shells, and another union find data structure helps with inner and outer boundary cycles of facets. We proceed similarly with redundant edges and vertices. This order of simplification steps simplifies the number of cases we need to detect.

4.3 Synthesizing the SNC from Sphere Maps

Given the sphere maps for a particular polyhedron, we can synthesize the corresponding SNC. The synthesis proceeds in the order of increasing dimension:

(1) We identify *svertices* that we want to link together as edges. We form an encoding for each *svertex* consisting of: (a) a normalized line representation for the supporting line, e.g., the normalized Plücker coordinates of the line [42], (b) the vertex coordinates, (c) a $+1$ or -1 indicating whether the normalization of the line equation reversed its orientation compared to the orientation from the vertex to the *svertex*. We sort all encodings lexicographically. Now, consecutive pairs in the sorted sequence form the desired edges.

    Normalizing the Plücker coordinates can be done by scaling them to have a leading coefficient of one. For an exact representation with integer arithmetic, an alternative is the division with the greatest common divisor of all six coefficients.

    To obtain normalized Plücker coordinates for an *svertex* of a frame point, we need two standard points on the supporting line of the *svertex*. Since the supporting line is the same for every R, we get the two standard points by instantiating the frame point with two arbitrary values for R. When comparing vertex coordinates in the sort, we use the notion of R as an infimaximal number.

(2) Edge-uses correspond to *sedges*. They form cycles around *svertices*. The cycles around two *svertices* that are linked by an edge have opposite orientations. Thus, corresponding *sedges* are easily matched up and we obtain all the boundary cycles of facets.

(3) We sort all boundary cycles by their normalized, oriented plane equation. We find the nesting relationship for the boundary cycles in one plane with a conventional two-dimensional sweep-line algorithm [31, section 10.7].

(4) Shells are found with a graph traversal. The nesting of shells is resolved with ray shooting from the lexicographically smallest vertex, whose sphere map also gives the set-selection mark for the corresponding volume by looking at the mark in the sphere map in the $-x$ direction. This concludes the assembly of volumes.

4.4 Boolean Operations

We represent Nef polyhedra as SNCs. We can trivially construct an SNC for a half-space. We can also construct it from a polyhedral surface [26] representing a closed orientable two-manifold by constructing sphere maps first and then synthesizing the SNC as explained in the previous section.

Based on the SNC data structure, we can implement the boolean set operations. For the set complement we reverse the set-selection mark for all vertices, edges, facets, and volumes. For the binary boolean set operations we find the sphere maps of all vertices of the resulting polyhedron with the following four steps and synthesize the SNC from there:

(1) Find possible candidate vertices. We take as candidates the original vertices of both input polyhedra and we create all intersection points of edge-edge and edge-face intersections.
(2) Given a candidate vertex, we find its local sphere map in each input polyhedron. If the candidate vertex is a vertex of one of the input polyhedra, its sphere map is already known. Otherwise a new sphere map is constructed on the fly. We use point location to determine where the vertex lies with respect to each polyhedron. If the candidate is in a volume or facet, its sphere map can be trivially constructed. If the candidate sits on an edge, the sphere map can be easily deduced from the sphere map neighborhood of one of the edge endpoints.
(3) Given the two sphere maps for a candidate vertex, we apply the boolean operation on sphere maps, see Section 4.1, to obtain the resulting sphere map.
(4) Based on the classification of local neighborhoods in Section 4.2, we check if the resulting sphere map represents a vertex, in which case we keep it for the final SNC synthesis step, and we discard it otherwise.

The topological operations *boundary*, *closure*, *interior*, *exterior*, and *regularization* are easy to implement. For example, for the boundary operation all volume marks are de-selected, all vertex, edge, and facet marks are selected, and the remaining SNC is simplified (Section 4.2).

The uniqueness of the representation implies that the test for the empty set is trivial. As a consequence, we can implement for two polyhedra P and Q the subset relation as $P \subset Q \equiv P - Q = \emptyset$, and the equality comparison with the symmetric difference.

5 Complexity

Amongst others, the complexity of most of our functions are essentially determined by the complexity of the point location, the vertical 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 expected complexity under a number of heuristic assumptions in Section 5.1 and 5.2. In Section 5.3 we include the complexities of the heuristic search data structures into the total complexity of the major functions provided by our package.

Let the total complexity of a Nef polyhedron be the number of vertices and sedges.

5.1 Kd-Tree

We chose to implement a kd-tree [21] for the ray-shooting and the point-location queries. 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 leaves, we store all vertices, edges, and facets that intersect the corresponding region. Consequently, long edges and large facets can be stored in up to $O(n)$ leaves, and there might exist leaves 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 cell in $O(\log n)$, but might be forced to check against $O(n)$ items in that cell.

Naturally 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 con-

Table 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.

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	$O(n^2)$	$O(n\log^3(n)+s)$

sequences: Each edge or facet is stored in a constant number of cells, and each cell 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 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 1. We study them experimentally in Section 8.

5.2 Box-Intersection Algorithm

We use the fast box-intersection algorithm described in [45] that is based upon streamed segment trees 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. The box-intersection algorithm is a heuristic that 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.

5.3 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 the construction of the kd-tree. The synthesis runs in expected $O(n\sqrt[3]{n}\cdot\log n)$ time, or even $O(n \log n)$ time, in particular if there are only $O(n^{\frac{2}{3}})$ many shells in the polyhedron. This is also the cost for constructing a polyhedron from an orientable two-manifold solid.

Table 2
Worst-case and expected complexity of unary operations, where n denotes the complexity of the polyhedron. See Table 3 for the binary operations. The expected runtime is given under the heuristic assumptions described in Section 5.1 and 5.2.

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	$O(n^2)$	$O(n \log n)$

Given a polyhedron of complexity n, the complement operation 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, which accounts for 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 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 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 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 polyhedron, unless the kd-tree deteriorates as explained above.

Table 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 5.1 and 5.2.

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	$O((n+m+k)\log(n+m)+ nm + k^2\sqrt[3]{k}\log k)$	$O((n+m)\log^3(n+m)+ k\log(n+m)+c\sqrt[3]{k}\log k)$

6 Implementation

Our implementation was released in December 2004 as part of CGAL release 3.1. CGAL, the *Computational Geometry Algorithms Library*, is an Open Source C++ software library [1] [13, 27]. Its design follows the generic programming paradigm. Its consequences for our implementation are a highly flexible and extendible data structure without compromises in the runtime efficiency, since the flexibility is realized with C++ templates and is resolved at compile-time rather than at runtime.

Unbounded polyhedra are supported with an extended geometric kernel that implements the infimaximal box. If we restrict us to bounded polyhedra, we can use the standard geometric kernels in CGAL without infimaximal box.

We support the construction of Nef polyhedra from manifold solids [26], boolean operations (union, intersection, complement, difference, symmetric difference), topological operations (interior, closure, boundary, regularization), rotations by rational rotation matrices (rotation angles are approximated up to a tolerance [9]). We follow the exact geometric computation paradigm [44] to guarantee correctness.

The implementation of the sphere-map data structure and its algorithms has about 10000 lines of code, and the implementation of the SNC structure with its algorithms and the visualization code in OpenGL and Qt has about 20000 lines of code. Clearly, the implementation re-uses parts of CGAL; in particular the geo-

[1] <http://www.cgal.org>

metric primitives, some data structures, and the generic sweep-line algorithm are re-used.

A bound on the necessary arithmetic precision of the geometric predicates and constructions is of interest in geometric algorithms. Of course, Nef-polyhedra can be used in cascaded constructions that lead to unbounded coordinate growth. However, we can summarize here that the algebraic degree is less than ten in the vertex coordinates for all predicates and constructions. The computations of the highest degree are in the plane sweep algorithm on the local sphere map with predicates expressed in terms of the three-dimensional geometry. Other high degree predicates are the lexicographic comparison of constructed vertices in the planar sweep line algorithm for ordering face boundary cycles and the vertex ray shooting for classifying shells. Furthermore we have orientation tests, sorting of Plücker coordinates, gcd computations for Plücker coordinates and plane equations, the two-dimensional plane sweep algorithms embedded in a three-dimensional plane, intersection tests between lines, rays, and planes.

We support the construction of a Nef polyhedron from a manifold solid defined on vertices. Nef polyhedra are also naturally defined on plane equations, and combined with CGAL's flexibility, one can realize schemes where coordinate growth is handled favorably with planes as defining geometry [14].

7 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 [26], which were also used by Seel [39, 40] 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 classes SNC_structure and Sphere_map are models of the template parameters SNCTraits and SMTraits, respectively. 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 list: for all vertices, halfedges (=svertices), halffacets, volumes, shalfedges, shalfloops, and sfaces. Sphere_map and SNC_structure provide interfaces for 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 constant `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 constant `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 users, 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 deduced above, the new class realizes the hole functionality of the class `Sphere_map`. Most of the functionality is already given by the vertex. The remaining functionality is related to the management of items, which is delegated to the `SNC_structure`. For reusing the code of Nef_S2, we regularly need the type `SNC_sphere_map`. We therefore 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::Halffacet<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;
  ...
};
```

The main classes of the two packages look as follows. The class Nef_polyhedron_3 has three templates 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 (the default type is 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, transformation, point location, and entries to the incidence structure. The latter are functions that initiate a shell traversal, or provide iterator ranges of all vertices, halfedges, edges, halffacets, facets, volumes, shalfedges, sedges, shalfloops, sloops, and sfaces. The user interface is completed by an input and an output operator. As usual, those operations are global functions.

For a detailed discussion on how to use generic programming with CGAL see [13, 26] and the CGAL manual [10].

8 General Runtime Behavior

In this and in the following sections, 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 Section 5, and we have designed experiments to stress our implementation with worst-case scenarios.

Besides the total runtime, we also 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 Section 4.4 for further explanations.

(1) **Point location:** queries the kd-tree of the input polyhedra to locate the vertices of the other polyhedron.
(2) **Box intersection:** intersection finding on the bounding boxes only, excludes the cost of the intersection test on the actual edge and facet geometry.
(3) **Sphere sweeps:** sum of all sphere sweep-line algorithms called during boolean operations on sphere maps.
(4) **Synthesizing edges:** in the synthesis step, sorts the line representation based on Plücker coordinates.

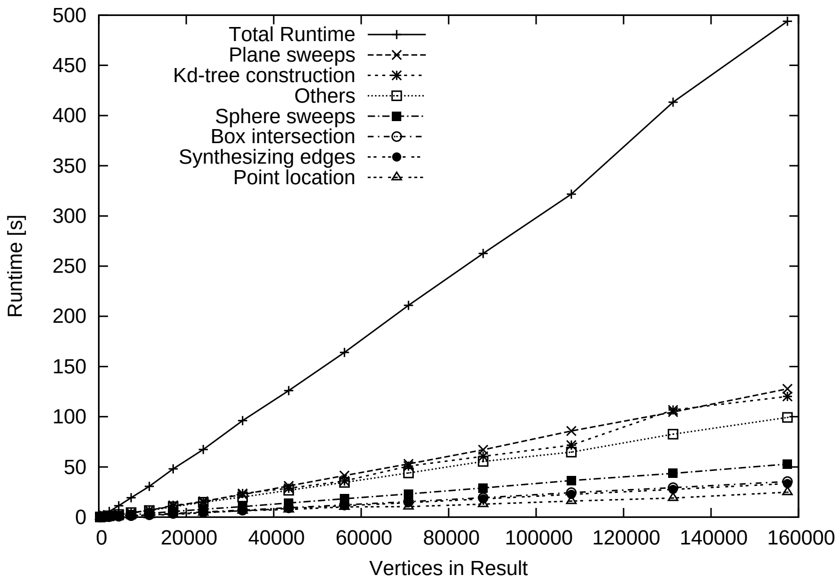


Fig. 8. Total runtime and runtime distributed over the main subroutines for our implementation in the TETGRID experiment.

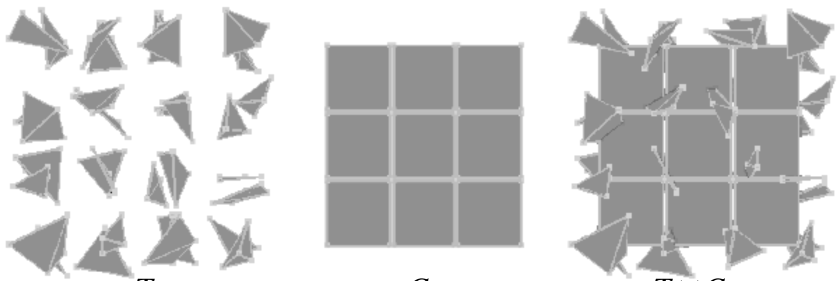
(5) **Plane sweeps:** in the synthesis step, sorts facet boundary cycles of the result polyhedron.
(6) **Kd-tree construction:** in the synthesis step, initializes the kd-tree for the result polyhedron.
(7) **Ray shooting:** in the synthesis step, used to resolve the nesting of shells of the result polyhedron (usually very small because of very few shells).
(8) **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.

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. We use g++-3.3.3 with the -O2 option.

We use the tool ExpLab [23] to schedule, run, and archive our test series. The source code, the test data, and the results are published for reference at `<http://www.mpi-inf.mpg.de/~kettner/proj/Nef>`.

8.1 Balanced Binary Operations

In our first test series, we want to examine the generic runtime behavior when the two input objects and the output object have all similar size. We capture these properties in the TETGRID experiment. We measure its runtime for values $N = 3, \ldots, 17$ on machine 1 for a later comparison with ACIS.



Experiment TETGRID
1. Create a regular N^3 grid T of random tetrahedra:
   (a) Generate four vertices for each tetrahedron randomly in a half-open fixed-size cube.
   (b) Let these cubes form a regular N^3 grid.
2. Create a regular $(N-1)^3$ grid C of such cubes.
3. Align T and C such that the grid nodes of C are at the centers of the grid cells of T.
4. Measure time for $T \cup C$.

In Figure 8 we see the total runtime, and the runtime distributed over the main subroutines in the second graph. The total runtime looks linear in the first graph. The plane sweeps and the construction of the kd-tree comprise a big part of the total runtime. 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.

8.2 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 $N = 10i$, $i = 1, \ldots, 15$ on machine 2. We see in Figure 9 that the construction of the kd-tree is responsible for the majority of 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, which already comprised a large part of the runtime in the previous experiment, becomes even more dominating because of its $\Omega(k \log k)$ runtime. Planar sweeps are not needed in this experiment, as there is no hole in any facet. Also, only very few sphere sweeps are needed. There are no edge–edge intersections and only very few vertices that are not located in a volume of the respective other polyhedron.

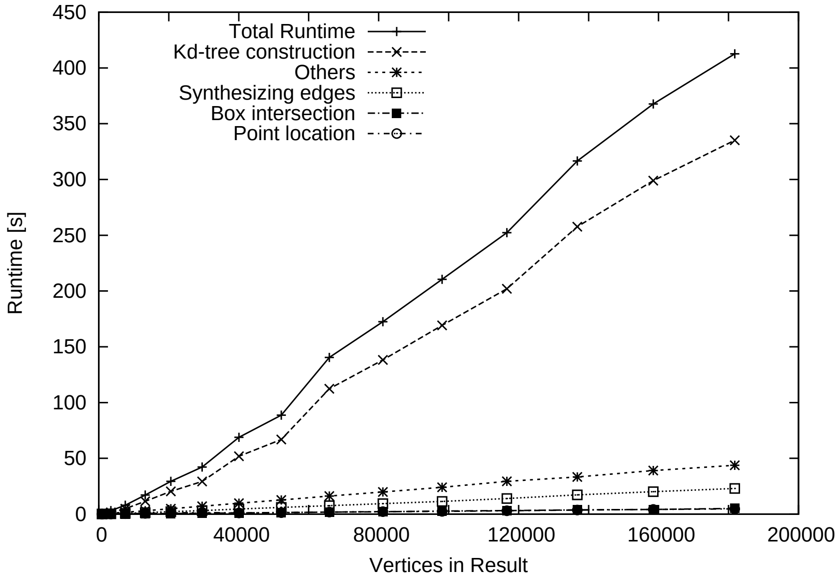
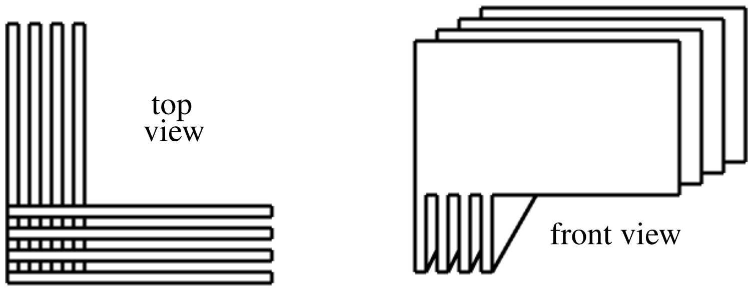


Fig. 9. Total runtime and runtime distributed over the main subroutines for our implementation in the QUADRATICWALLGRID experiment. Note that the plane sweep and the ray shooting are not executed in this experiment.

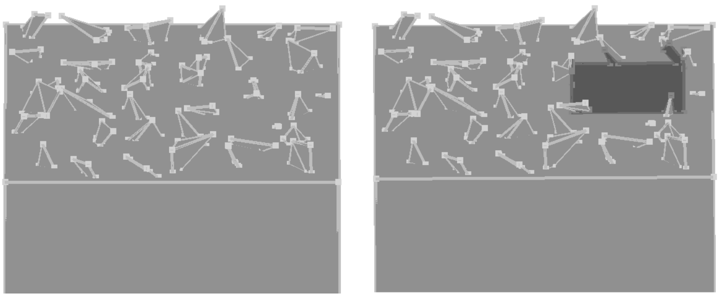


Experiment QUADRATICWALLGRID

1. Construct N parallel cuboids of size $10000 \times 1 \times 100$ spaced one unit apart in y-direction as object W.
2. Construct N parallel cuboids of size $1 \times 10000 \times 100$ spaced one unit apart in x-direction as object W'.
3. Align W and W' at their lower front left corner.
4. Move W' along the z-axis for fifty units.
5. Measure time for $W \cup W'$

8.3 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 the COMPLEXFACET experiment that stresses the sweep-line algorithm sorting the facet boundary loops.



$C \cup G$ $\quad\quad$ $(C \cup G) \setminus c$

Experiment COMPLEXFACET, COMPLEXMINUSSIMPLE

1. Create a cube C of size N^3.
2. Create a $N \times N \times 1$ grid of tetrahedra G:
   (a) Generate four vertices for each tetrahedron randomly in a half-open unit cube, but at least one vertex in the lower half and one vertex in the upper half of the cube.
   (b) Let the cubes form a regular $N \times N \times 1$ grid and place the grid such that each tetrahedron penetrates the top surface of C.
3. COMPLEXFACET: Measure time for $C' = C \cup G$.
4. Create a cube c of size 2^3 such that its vertices match centers of the grid cells.
5. COMPLEXMINUSSIMPLE: Measure time for $C' \setminus c$

For the COMPLEXMINUSSIMPLE experiment, we perform a test series with $N = 5i$ and $i = 1, \ldots, 40$. Since we want to run this experiment with ACIS, too, we perform it on machine 1. Figure 10 shows the results of the experiment. There is no subroutine, that is dominating the runtime. Still the kd-tree construction is most time consuming. The other subroutines follow in the same order as in the TETGRID experiment.

As already discussed in our analysis from Section 5, the runtime of the binary operation depends on the complexities of both input and the output complexity. As

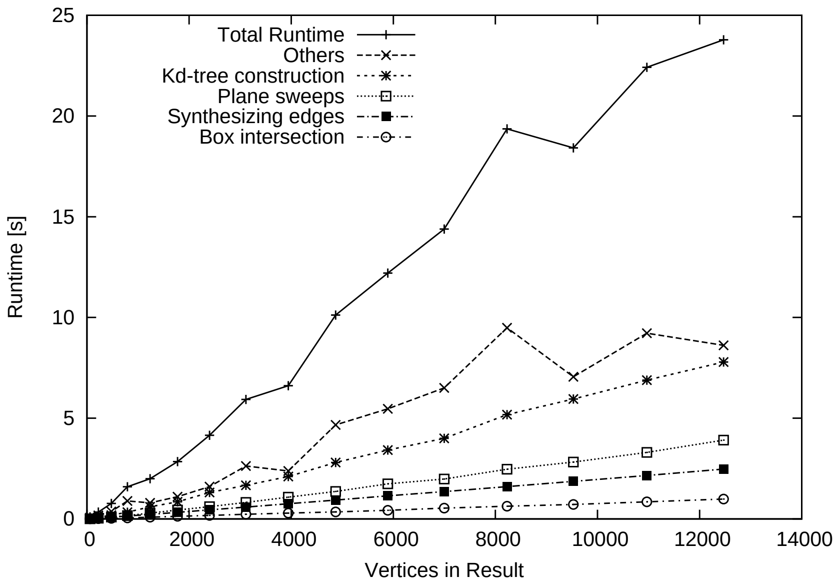


Fig. 10. Total runtime and runtime distributed over the main subroutines in the ComplexMinusSimple experiment.

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 10 confirms this property.

9 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 find out which subroutines can become bottlenecks. Furthermore, we want to confirm the theoretical runtime analysis of Chapter 5.

9.1 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 27) 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 11 shows the result of a test series with $N = 20i$ and $i = 1,\ldots,10$ on machine 2. Although we tried to build a scenario that especially stresses the runtime of the

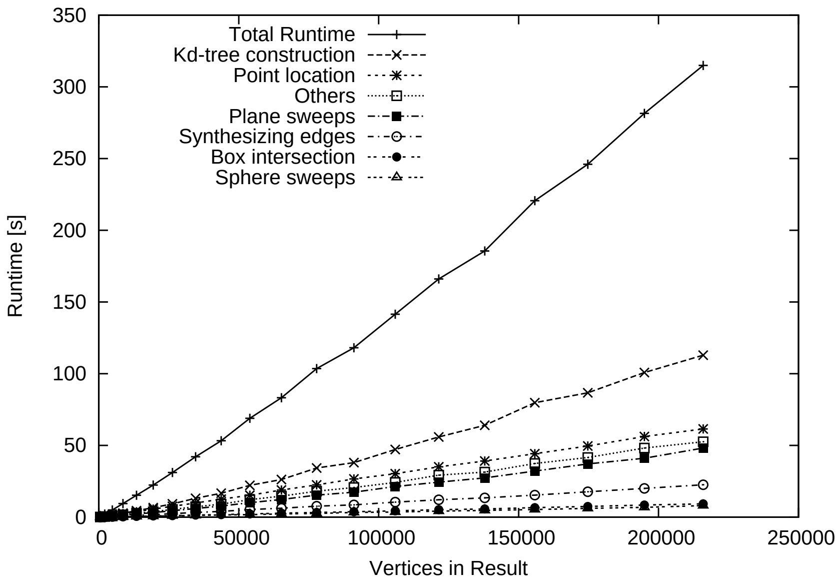


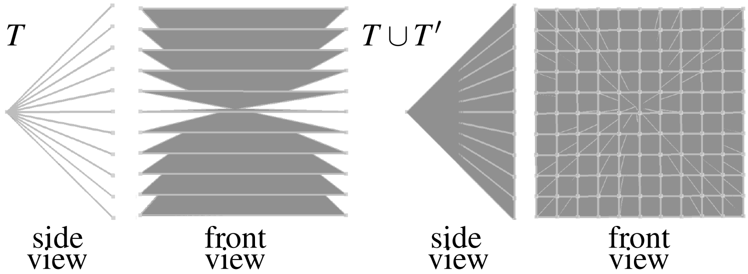
Fig. 11. Total runtime and runtime distributed over the main subroutines for our implementation in the COMPLEXFACET experiment.

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 [40].

9.2 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 sphere map with complexity $O(n+m+s)$. Normally, each sphere map is of constant size. Then the runtime of the overlay drops to $O(n+m+s)$ time.



Experiment COMPLEXSPHEREMAP

1. Create set of N triangles T such that
   (a) they all meet in a common vertex v.
   (b) the edges E opposite to v are parallel to each other with uniform distance between pairs of triangles.
   (c) the length of each edge e in E equals the largest distance between to pairs of edges in E.
2. Create a copy T' of T
3. Rotate T' around the axis through v and the center of the edges E by 90 degrees.
4. Measure time for $T \cup T'$

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 12 shows the result of a test series of this scenario with $N = 50i$, $i = 2, \ldots, 17$ performed on machine 2. Again, the kd-tree construction dominates the runtime. The half-sphere sweeps are the second biggest consumer, but only need half the runtime of the kd-tree construction.

9.3 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.

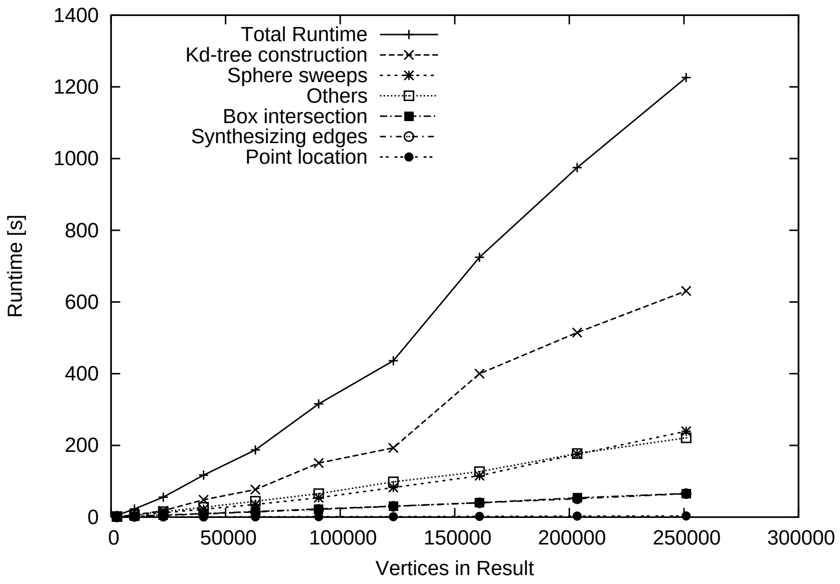
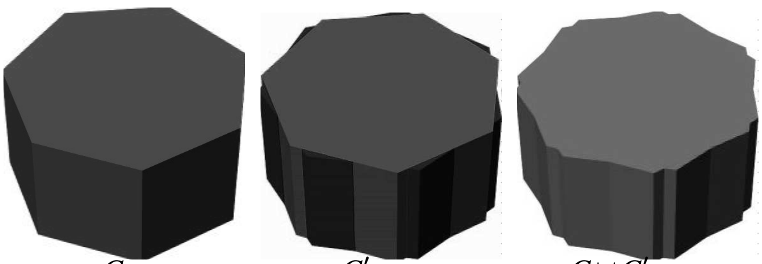


Fig. 12. Total runtime and runtime distributed over the main subroutines for our implementation in the COMPLEXSPHEREMAP experiment for $N = 50i$, $i = 2, \ldots, 17$.



C C' $C \cup C'$

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 $C \cup C'$.

Figure 13 shows the result of a test series with $\alpha = 10^{-7}$ and $n = 100i$, $i = 1 \ldots 30$ on machine 2. The curve of the kd-tree construction includes some irregularities. Most remarkable is a 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 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.

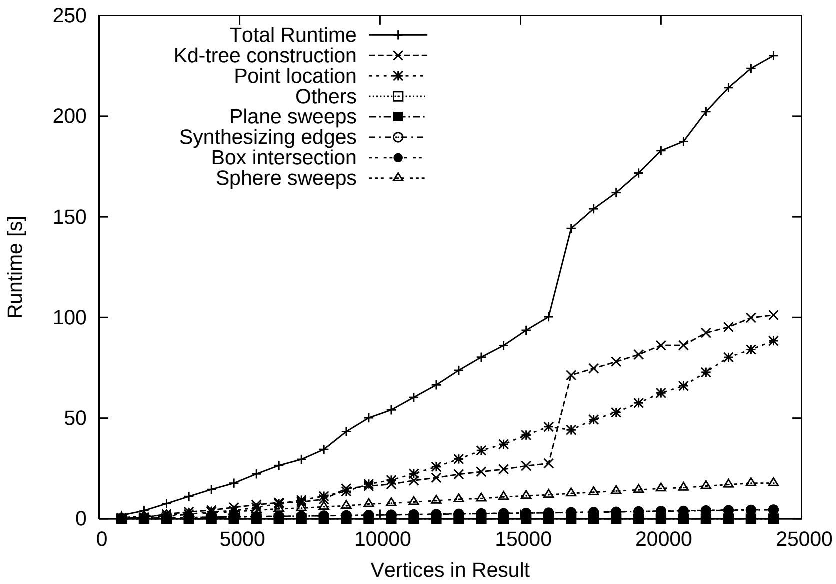


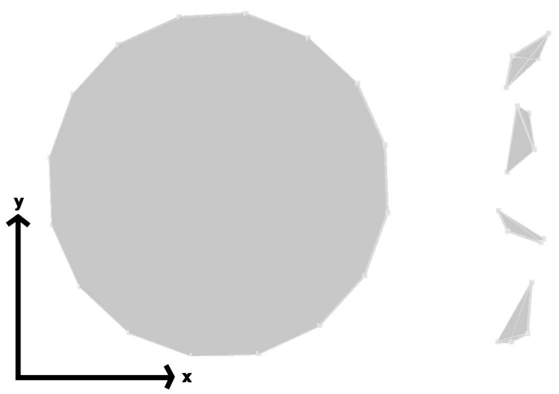
Fig. 13. Experiment ROTCYLINDER with $\alpha = 10^{-7}$ and $n = 100i$, $i = 1 \ldots 30$: Shown are the total runtime of the total runtime of the binary operation and the runtime of the main subroutines.

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 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^x 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 leaves contain either of the large facets. In combination with $O(n)$ point location queries, the subroutine is expected to use quadratic runtime. The curve of the point location in Figure 13 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.



Experiment WORSTCASERAYSHOOTING
1. Create regular N-gon G parallel to xy-plane.
2. Rotate G by small angle α around an axis through the center of G parallel to the y-axis.
3. Create a set of $N/4$ tetrahedra T, such that
   (a) they do not pairwisely overlap in y direction.
   (b) they are completely to the right of G.
   (c) their y coordinates are in the range of G's y coordinates.
   (d) their smallest vertex is either slightly higher than G's highest vertex, or lower than G's lowest. vertex.
4. Measure time for $G \cup T$.

Figure 14 shows the result of a test series of the WORSTCASERAYSHOOTING experiment with $N = 80i$, $i = 1, \ldots, 50$. 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 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 inter-

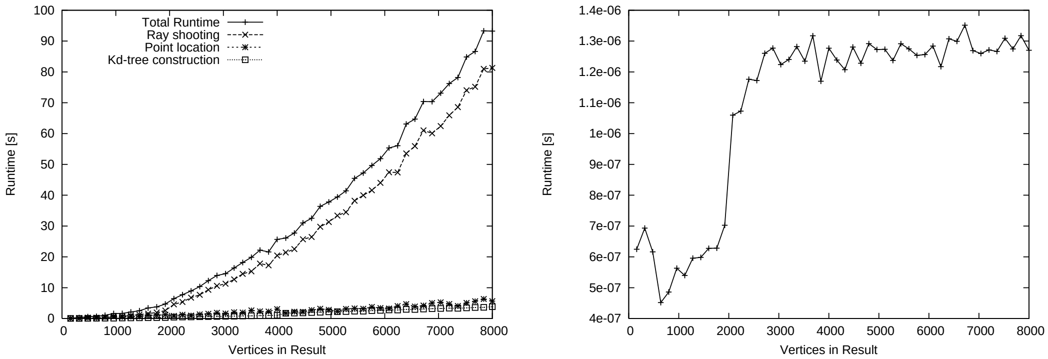


Fig. 14. Experiment WORSTCASERAYSHOOTING with $N = 80i$, $i = 1,\ldots,50$: The left graph shows the runtime of the major subroutines and the total runtime of the binary operation. The graph on the right shows the runtime of the ray shooting subroutine divided by the squared output complexity.

section test of the ray with the box already reveals that they do not intersect.

9.4 Résumé

Although the kd-tree improves the performance of our binary operations considerably, it is still the bottleneck. Most notably are operations with a quadratic result 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 only 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 narrow, such that they intersect many kd-tree cells. As a result, the runtimes change for the worse.

Further approaches store the relevant intervals of facet cycles, or create two new facet objects for each split facet cycle. The latter approach involves large additional memory resources for the new facet cycles. Furthermore, it seems too costly to split facets into two halves, here. We have already experimented with the first approach. Our solution is based upon a specialized point–in–polygon test that can handle the facet cycle intervals. First test runs were promising, but the method is not completely implemented and tested, yet. As the new method requires a bounding box for each original facet and the box must be updated during each affine transformation, we trade a faster kd-tree construction for a slower transformation. It is unclear whether the tradeoff is profitable.

Another approach to handle complex facets stores the facets in interior nodes of the kd-tree. This way, we considerably speed up the construction, but risk to increase the number of intersection tests performed by point-location queries. As a compromise, we can exchange the intersection test between splitting planes and complex facets by the intersection of boxes. Consequently, we omit the costly intersection tests and lower the risk of inefficient point-location queries.

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 and only use it for a few ray shooting queries afterwards. If it is not adopted for point location by the user or in another binary operation, the effort seems wasted. Therefore, we want to find out, if there is some efficient way to solve the ray-shooting queries without constructing the whole kd-tree, and postpone its final construction to the beginning of the next binary operation, when it is needed for the point location queries. If this approach works efficiently, we can offer the user to decide about the construction time of the kd-tree, i.e., he decides whether a kd-tree is needed or not.

10 Optimizations

We have seen in the previous section that certain subroutines of the algorithm are very dominant and others are less dominant, maybe to the surprise of the reader. One main reason is that we implemented several optimizations that prevent the execution of some complex subroutines for many common cases. We will discuss the optimizations in detail and justify their effectiveness with experiments.

10.1 Sphere Sweep

The sweep-line algorithm is a powerful tool. We use it in the plane for facet boundary cycles and we use it on half-spheres for the sphere maps. However, it is a comparatively costly step, although its asymptotic complexity is close to optimal.

We evaluate the contribution of the sphere sweeps to the total runtime of a binary boolean operation with a TETGRID experiment for $N = 16$ on machine 2. Table 4 lists the number of sphere sweeps performed during this operation, together with the running time of all sphere sweeps and the total running time. The values in the first row refer to a test run without any optimizations. The other rows refer to test runs with one or more of the following optimizations activated:

(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.

Table 4
Number of sphere sweeps performed and the runtime of all sweeps and the complete binary operation are shown for runs with no optimization, one optimization, and all three optimizations enabled during a TETGRID experiment with $N = 16$.

optimizations			number of	runtime	
(i)	(ii)	(iii)	sweeps	sphere sweeps	binary op.
-	-	-	240716	238.70s	460.62s
+	-	-	14932	17.06s	147.19s
-	+	-	201470	224.95s	440.68s
-	-	+	217744	226.26s	446.27s
+	+	+	12940	14.74s	145.51s

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. They are unlikely to occur in non-artificial scenarios.

(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. Often this results in twice as many elements that need to be swept. We therefore test, if all svertices and sedges of a sphere map either lie on the top, bottom, left, right, front, or back half-sphere. In such an instance, only one sweep is performed instead of two.

(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 inside 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 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.

10.2 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. Table 5 shows the effect of this optimization. We use the same test scenario as in Table 4 and perform two runs, one with a plane sweep for every supporting plane and one where we perform the necessary plane sweeps only.

Table 5
The number and runtime improvement for the plane-sweep optimization shown with a TETGRID experiment for $N = 16$.

optimization	# sweeps	time sweeps	total time
off	16207	43.71s	145.51s
on	757	36.47s	135.78s

10.3 Intersection

We have two instruments for fast intersection computation: a kd-tree and an box-intersection algorithm.

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. The expected complexity of finding an edge-facet or an edge-edge intersection with 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.

It is unclear which algorithm is more efficient, box intersection or the kd-tree. The kd-tree could win asymptotically for L 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 may actually test the same candidates several times if they happen to be stored together in several kd-tree cells. The complexity may not change, but the hidden factor might be higher than for the box intersection.

Figure 15 shows the result of a test series with the TETGRID experiments for $N = 1,\ldots,20$. The box-intersection algorithm is clearly preferable. We also counted the

Table 6
TETGRID experiment: Number of intersection candidates tested per existing intersections.

N	Input Complexity	tests / intersections	
		box inters.	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

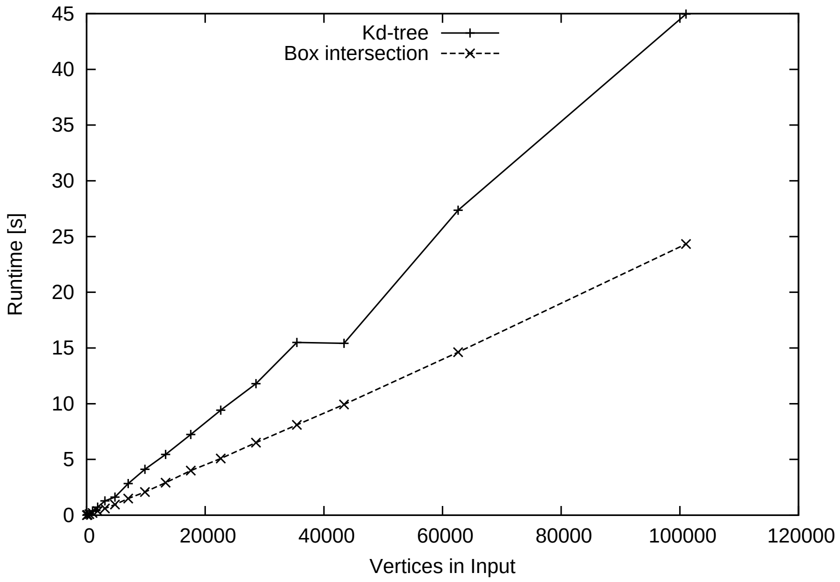


Fig. 15. Runtime comparison between the box-intersection algorithm and the kd-tree on the TETGRID experiment.

number of intersection candidates that are tested in each algorithm, see Table 6 for the result. Indeed, the kd-tree tests more (redundant) candidates.

11 Comparison with ACIS R13

We compare our implementation with ACIS R13, a common commercial CAD kernel used in many CAD systems [41]. It should be said that we are comparing apples with oranges here. On one side, it is daunting for a research prototype to be compared with a long established and optimized industry implementation. On the other side, 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. Note that we use no floating point filter, yet. Unfortunately, our synthesis step is improper for effective filtering. A redesign of this step is complicated and was not part of this work. On the other hand, our latest work shows that a redesign is possible and permits effective filtering.

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 ran under Linux, while ACIS ran under Microsoft Windows XP.

11.1 Balanced Binary Operations

We repeat the TETGRID experiment with ACIS, to get a general impression. It contains no special difficulties. However, facets are likely to have holes and we do not exclude degeneracies explicitly, but they are highly unlikely. Naturally, both algorithms perform on the same data sets.

The results in Table 7 show that ACIS is faster by a factor of 2 to 4. 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.

11.2 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.

Table 7
Comparison of ACIS R13 and our Nef polyhedron with the TETGRID experiment.

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

We designed the simple ROTCYLINDER experiment (see page 31) 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 show the runtime cost for exact arithmetic, because the input coordinates grow in this series of experiments.

In this 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.

Up front, we want to explain a problem of exact arithmetic in this scenario, where we need exact rotation. We compute exact values for $\sin(\alpha')$ and $\cos(\alpha')$ such that $\sin^2(\alpha')+\cos^2(\alpha')=1$ and $|\alpha-\alpha'|<\varepsilon$ for a small specified $\varepsilon>0$ that we fix in our experiment to $\frac{\alpha}{10000}$ with α given in degrees. We can use an implementation in CGAL that is based on Farey sequences as described in [9]. The CGAL implementation is division free but slower than the algorithm described in [9]. The runtime for finding such an exact rotation matrix amounts to a non-negligible fraction for small ε, as can be seen in Table 9.

Table 8
Comparison of ACIS R13 and our Nef polyhedron with the ROTCYLINDER experiment. Here, “not executable” means that ACIS could not compute the union correctly and therefore cancels the operation. As a result, ACIS keeps the first input object unmodified and deletes the second input object.

n	α	time ACIS R13	runtime [s] 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
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 $\frac{\alpha}{10000}$.

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

We omit the computation of the exact rotation in our test series and focus on the binary boolean operation. The result of the ROTCYLINDER experiment in Table 8 shows that ACIS’ floating-point operations are insufficient for α smaller than 10^{-3}. On the other side, ACIS is faster except for very large instances. For $n = 100$ the

Table 10
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.

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

factor of our runtime and ACIS' runtime is slightly below 4; for $n = 2000$ we are faster up to a factor of 1.2. Additionally, we performed a run with $n = 10000$ and $\alpha = 10^{-7}$ to highlight the robustness of our arithmetic operations.

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. The results in Table 10 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}.

11.3 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. We repeat this experiment with ACIS and compare it with the results of our implementation from Section 9.

Table 11 shows the results of a test series with $N = 3i$, $i = 1,\ldots,13$. 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.

12 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 [35] 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 Chapter 11. 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

Table 11
Comparison of ACIS R13 and our Nef polyhedron with the COMPLEXMINUSSIMPLE experiment. ACIS is a factor of twenty faster.

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

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 leaves. 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 31). 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 Chapter 11. 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 9). 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 $i = 1,\ldots,50$, and with $n = 1000$, 2000.

Figure 16 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. 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 16 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 $\alpha = 10^{-8}$

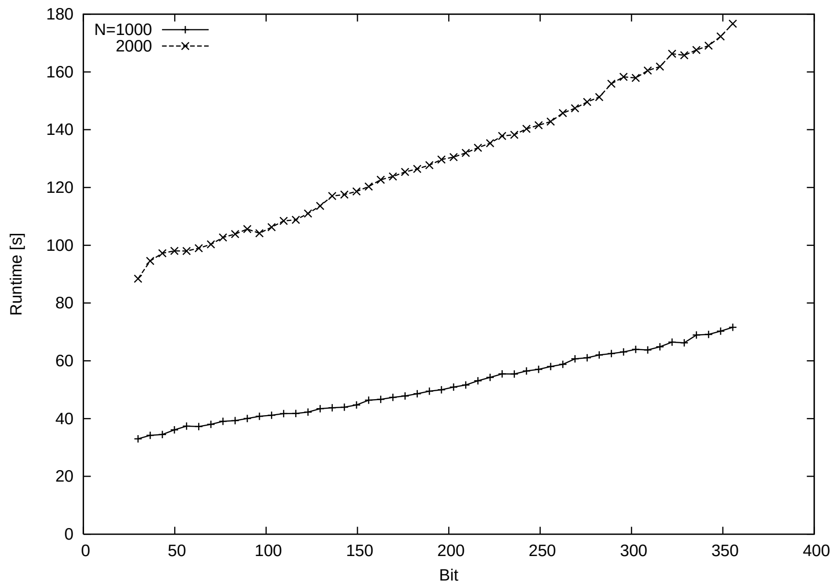


Fig. 16. Experiment ROTCYLINDER with $\alpha \approx 1.993373 \cdot 10^{-i}$, $i = 1, \ldots, 50$, and with $n = 1000, 2000$: The graph depicts the relation between the bit complexity of the coordinate representation and the runtime.

degrees. Finally, we unite C and C'. We perform test runs with $i = 1, \ldots, 50$, 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 17 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.

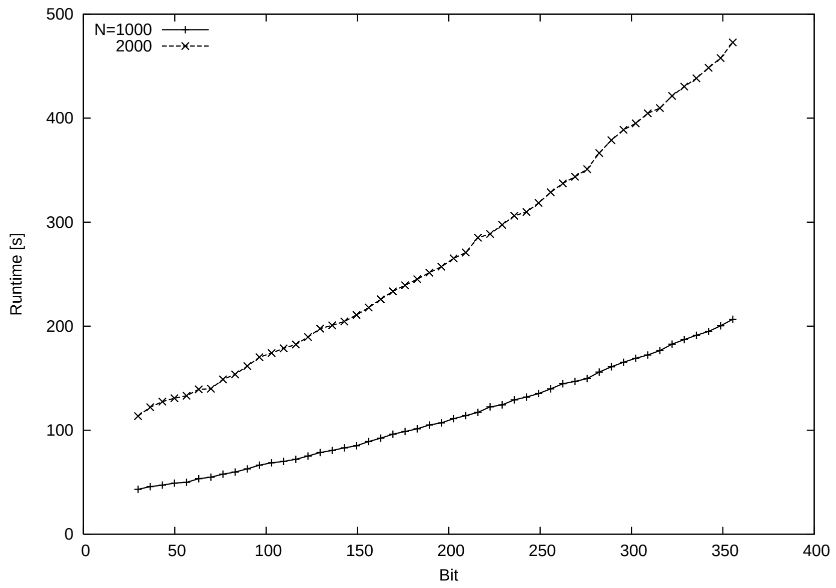


Fig. 17. Experiment ROTCYLINDER with $\alpha = 10^{-8}$, and with $n = 1000$, 2000: In order to obtain two input polyhedra with equally large coordinate representations, C is initially rotated by $\beta \approx 1.993373 \cdot 10^{-i}$ degrees. The rotated C is then used as input object for the ROTCYLINDER experiment. The graph depicts test series with $i = 1, \ldots, 50$

Now, we put our simulated scenarios in relation to a real scenario. Let us 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 $n = 2000$.

13 Comparison to Extant 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 [14, 22, 24, 30, 38]. In their seminal work, Nef and, later, Bieri and Nef [8, 36] developed the theory of Nef polyhedra. Dobrindt, Mehlhorn, and Yvinec [12] 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 [39, 40] gives a detailed study of planar Nef polyhedra; his implementation is available in CGAL. [2]

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* [37]. 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 a subset of cells of the complex that build 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 [43] and Karasick's star-edge boundary representation [25] 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 are, however, not completely covering all incidences [17]. 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 [17] 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 [14] centers around plane equations and uses symbolic perturbation of the planes' distances to the origin, which allows the handling of non-manifold situations and lower-dimensional faces, while 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 was 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

[2] <http://www.cgal.org>

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

That we can quite naturally handle all degeneracies, including non-manifold structures, as well as unbounded objects and produce always the correct mathematical result differentiates us from other approaches. Previous approaches using exact arithmetic [1, 2, 3, 14, 28] work in a less general modeling space, some unable to handle non-manifold objects and none able to handle unbounded objects.

14 Conclusion

We achieved our goal of a complete, exact, correct, and efficient implementation of boolean operations on a very general class of polyhedra in space. Still there is more room for optimization, e.g., our current bottleneck is the construction of the kd-tree as discussed in Section 9.4. Furthermore, our data structure is verbose; it provides all adjacency relations by storing them explicitly. Other approaches, like the Partial Entity Structure by Lee and Lee [29], optimize the memory efficiency of a representation by omitting to store some relations that can be obtained from others on demand with very little overhead.

Useful extensions with applications in exact motion planning are Minkowski sums and the subdivision of the solid into simpler shapes, e.g., a trapezoidal or convex decomposition in space.

For ease of exposition, we restricted the discussion to boolean flags. Larger label sets can be treated analogously.

While Nef complexes are restricted to sub-algebraic sets defined by linear polynomials, they can serve as a starting point for a representation of curved surfaces. Some of the algorithms become difficult and need further investigation, such as the synthesis algorithm, where we need unique representations of intersection curves, and where we need to sort points on intersection curves, which is possible with curves in parametric representation but difficult for curves in implicit representation. See [5] for an approach for quadric surfaces.

Other approaches go far beyond modeling sub-algebraic sets defined by linear or quadratic polynomials. The *Selective Geometric Complexes* (SGC) by Rossignac and O'Connor [37] proposes boolean operations on arbitrary semi-algebraic sets and the Djinn API by the Djinn project [34] is even designed for semi-analytic sets. However, to our knowledge there exists no running implementation that realizes semi-algebraic or semi-analytic sets together with proper set or "set-like" operations.

The exact geometric computing paradigm [44] allows us to rely in our implementation on correctness proofs from theory, a fundamental strength of this approach, but the underlying representation with exact number types cannot be easily interfaced with common programming APIs and file formats. The available IEEE `double` approximation usually suffices for rendering purposes, but the approximation error can in principle destroy all laboriously computed properties of the polyhedron. A possible solution to this *geometric rounding* problem is given in [15].

Another effect of the exactness is that constructions might contain small features, such as slivers, that are perceived as undesirable and that would have been merged in conventional approaches based on an ε tolerance value. Again, in our view this problem needs to be addressed separately, either on the modeling side by avoiding these small features altogether or as a simplification step afterwards.

Acknowledgements

We thank Michael Seel and Susan Hert for their work on planar Nef polyhedra and their initial parts for the 3D implementation. Furthermore, we thank Miguel Granados for a preliminary kd-tree implementation, and Andreas Meyer for the box-intersection implementation.

Work on this paper has been partially supported by the IST Programme of the EU as a Shared-cost RTD (FET Open) Project under Contract No IST-2000-26473 (ECG - Effective Computational Geometry for Curves and Surfaces).

References

[1] A. Agrawal and A. G. Requicha. A paradigm for the robust design of algorithms for geometric modeling. *Computer Graphics Forum*, 13(3):33–44, 1994.

[2] R. Banerjee and J. Rossignac. Topologically exact evaluation of polyhedra defined in CSG with loose primitives. *Computer Graphics Forum*, 15(4):205–217, 1996.

[3] 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.

[4] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Commun. ACM*, 18(9):509–517, 1975.

[5] E. Berberich, M. Hemmer, L. Kettner, E. Schömer, and N. Wolpert. An exact, complete and efficient implementation for computing planar maps of quadric intersection curves. In *Proc. 21st Symposium on Computational Geometry (SOCG'05)*, pages 99–106, New York, NY, USA, 2005. ACM Press.

[6] H. Bieri. Boolean and topological operations for nef polyhedra. In *CSG 94: Set-theoretic Solid Modelling: Techniques and Applications*, pages 35–53. Information Geometers, April 1994.

[7] H. Bieri. Nef polyhedra: A brief introduction. *Computational Suppl. Springer Verlag*, 10:43–60, 1995.

[8] H. Bieri and W. Nef. Elementary set operations with d-dimensional polyhedra. In *Comput. Geom. and its Appl.*, LNCS 333, pages 97–112. Springer Verlag, 1988.

[9] J. Canny, B. R. Donald, and E. K. Ressler. A rational rotation method for robust geometric algorithms. In *Proc. 8th Annu. ACM Symposium on Computational Geometry*, pages 251–260, 1992.

[10] *The CGAL Manual*. `http://www.cgal.org/Manual/`.

[11] T. H. Cormen, C. E. Leiserson, and R. L. Rivest. *Introduction to Algorithms*. MIT Press, 1990.

[12] K. Dobrindt, K. Mehlhorn, and M. Yvinec. A complete and efficient algorithm for the intersection of a general and a convex polyhedron. In *Proc. 3rd Workshop on Algorithms and Data Structures*, LNCS 709, pages 314–324, 1993.

[13] 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.

[14] S.J. Fortune. Polyhedral modelling with multiprecision integer arithmetic. *Computer-Aided Design*, 29:123–133, 1997.

[15] S.J. Fortune. Vertex-rounding a three-dimensional polyhedral subdivision. *Discrete and Computational Geometry*, 22:593–618, 1999.

[16] M. Granados, P. Hachenberger, S. Hert, L. Kettner, K. Mehlhorn, and M. Seel. Boolean operations on 3D selective Nef complexes: Data structure, algorithms, and implementation. In *Proc. 11th Annu. European Sympos. Algorithms (ESA'03)*, LNCS 2832, pages 654–666, Budapest, Hungary, 2003. Springer.

[17] E. Levent Gursoz, Young Choi, and Friedrich B. Prinz. Vertex-based representation of non-manifold boundaries. *Geometric Modeling for Product Engineering*, 23(1):107–130, 1990.

[18] Peter Hachenberger. *Boolean OPerations on 3D Selective Nef Complexes: Optimized Implementation and Experiments*. PhD thesis, Universität des Saarlandes, Saarbrücken, Germany, November 2006.

[19] Peter Hachenberger and Lutz Kettner. Boolean operations on 3d selective Nef complexes: Optimized implementation and experiments. In Leif Kobbelt and Vadim Shapiro, editors, *ACM Symposium on Solid and Physical Modeling (SPM 2005)*, pages 163–174, Cambridge, MA, USA, June 2005. Association for Computing Machinery (ACM), ACM.

[20] D. Halperin. Robust geometric computing in motion. *Internatioal Journal of Robotics Research*, 21(3):219–232, 2002.

[21] V. Havran. *Heuristic Ray Shooting Algorithms*. Ph.D. thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic,

2000.

[22] 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.

[23] 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.

[24] C. M. Hoffmann. *Geometric and Solid Modeling – An Introduction*. Morgan Kaufmann, 1989.

[25] M. Karasick. *On the Representation and Manipulation of Rigid Solids*. Ph.D. thesis, Dept. Comput. Sci., McGill Univ., Montreal, PQ, 1989.

[26] L. Kettner. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry: Theory and Applications*, 13:65–90, 1999.

[27] L. Kettner and S. Näher. Two computational geometry libraries: Leda and cgal. In Jacob E. Goodman and Joseph O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 64, pages 1435–1463. CRC Press LLC, Boca Raton, FL, second edition, 2004.

[28] 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.

[29] S. H. Lee and K. Lee. Partial entity structure: a compact non-manifold boundary representation based on partial topological entities. In *SMA '01: Proceedings of the sixth ACM symposium on Solid modeling and applications*, pages 159–170, New York, NY, USA, 2001. ACM Press.

[30] M. Mäntylä. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, Maryland, 1988.

[31] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[32] 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.

[33] 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.

[34] A. E. Middleditch, C. Reade, and A. J. P. Gomes. Set-combinations of the mixed-dimension cellular objects of the djinn api. *Computer-Aided Design*, 31(11):683–694, 1999.

[35] V. Milenkovic. Shortest path geometric rounding. *Algorithmica*, 27(1):57–86, 2000.

[36] W. Nef. *Beiträge zur Theorie der Polyeder*. Herbert Lang, Bern, 1978.

[37] 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, *Geom. Model. for Product Engin.* North-Holland, 1989.

[38] J. R. Rossignac and A. G. Requicha. Solid modeling. `http://citeseer.ist.psu.edu/209266.html`.
[39] M. Seel. Implementation of planar Nef polyhedra. Research Report MPI-I-2001-1-003, MPI für Informatik, Saarbrücken,Germany, August 2001.
[40] M. Seel. *Planar Nef Polyhedra and Generic Higher-dimensional Geometry*. PhD thesis, Universität des Saarlandes, Saarbr., Germany, 5. December 2001.
[41] Spatial Corp., A Dassault Systèmes company. *ACIS R13 Online Help*, 2004.
[42] J. Stolfi. *Oriented Projective Geometry: A Framework for Geometric Computations*. Academic Press, 1991.
[43] 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, *Geometric Modeling for CAD Applications*, pages 3–36. IFIP, May 12–16 1988.
[44] C. Yap. Towards exact geometric computation. *Computational Geometry: Theory and Applications*, 7(1):3–23, 1997.
[45] A. Zomorodian and H. Edelsbrunner. Fast software for box intersection. *Int. J. Computational Geometry and Applications*, 12:143–172, 2002.

Appendix

Lemma 6 *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

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

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 $\frac{y_1}{x_1} = 1$
  - $x_0 - y_0 > 0$
    $p_1 = (R, R - x_0 + y_0, \frac{R-x_0}{x_1} z_1 + z_0)$, $p_2 = (-R + x_0 - y_0, -R, \frac{-R-y_0}{y_1} z_1 + z_0)$
  - $x_0 - y_0 < 0$
    $p_1 = (-R, -R - x_0 + y_0, \frac{-R-x_0}{x_1} z_1 + z_0)$, $p_2 = (R + x_0 - y_0, R, \frac{R-y_0}{y_1} z_1 + z_0)$
  - $x_0 = y_0$
    $p_1 = (R, R, \frac{-R-x_0}{x_1} z_1 + z_0)$, $p_2 = (-R, -R, \frac{R-x_0}{x_1} z_1 + z_0)$
- $|x_1| = |y_1| = |z_1|$

  Dependent upon the sign of x_1, y_1, and z_1 we consider one of four diagonals through the cube. Dependent upon the values of x_0, y_0, and z_0 each diagonal may intersect two corner vertices of the infimaximal box, a combination of an edge and a plane, or two planes. In total we distinguish 28 sub-cases. The discussion of these sub-cases is straight forward and therefore omitted.

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

$$|(\pm R - x_0)\frac{y_1}{x_1} + y_0| \leq R \tag{1}$$
$$|(\pm R - x_0)\frac{z_1}{x_1} + z_0| \leq R \tag{2}$$

With $C = -\frac{y_1}{x_1}x_0 + y_0$, inequality 1 can be rewritten as follows:

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

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 2 follows analogously. Consequently, the given points are correct endpoints of l. $\square$