

Boolean Operations on 3D Selective Nef Complexes: Optimized Implementation and Experiments

Peter Hachenberger*
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany

Lutz Kettner†
Max-Planck-Institut für Informatik
Stuhlsatzenhausweg 85
66123 Saarbrücken, Germany

Abstract

Nef polyhedra in d -dimensional space are the closure of half-spaces under boolean set operation. 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.

We presented in previous work a new data structure for the boundary representation of three-dimensional Nef polyhedra with efficient algorithms for boolean operations. These algorithms were designed for correctness and can handle all cases, in particular all *degeneracies*. To this extent we rely on exact arithmetic to avoid well known problems with floating-point arithmetic.

In this paper, we present important optimizations for the algorithms. We describe the chosen implementations for the point-location and the intersection-finding subroutines, a kd-tree and a fast box-intersection algorithm, respectively. We evaluate this optimized implementation with extensive experiments that supplement the runtime analysis from our previous paper and that illustrate the effectiveness of our optimizations. We compare our implementation with the ACIS CAD kernel and demonstrate the power and cost of the exact arithmetic in near-degenerate situations.

The implementation was released as Open Source in the CGAL release 3.1 in December 2004.

CR Categories: I.3.5 [Computer Graphics]: Computational Geometry and Object Modeling—Boundary representations; J.6 [Computer-Aided Engineering]: Computer-aided design (CAD);

Keywords: boundary evaluation, B-rep, CSG, Nef polyhedra, non-manifold, unbounded polyhedra, completeness, robustness, exactness, benchmark, experiments, data structures, algorithms

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 [Hoffmann 1989; Mäntylä 1988; Rossignac and Requicha ; Berberich et al. 2005; Fortune 1997]. We restrict ourselves to partitions of three 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 so-called *Nef*

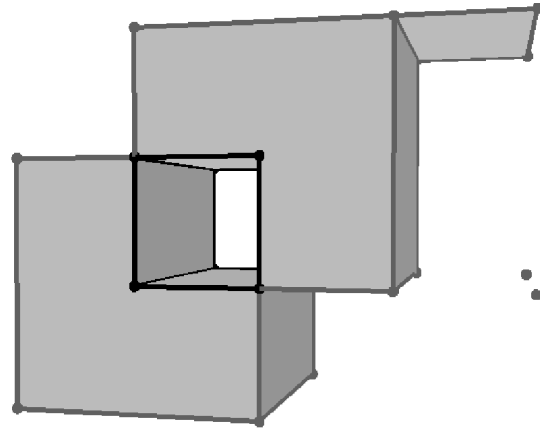


Figure 1: Nef polyhedron with non-manifold edges, a dangling facet, and two isolated vertices. The tunnel boundary does not belong to this point set.

polyhedron [Nef 1978], and we call the labels *set-selection marks*. Nef polyhedra can be obtained from halfspaces 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. In a three-dimensional earth model with different layers, reservoirs, faults, etc., one can use labels to distinguish between different soil types. Furthermore, we encounter here complex topology, for example, non-manifold edges.
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. This is one of the examples where Middleditch [Middleditch 1994] argues that we need more than regularized

*e-mail: hachenberger@mpi-sb.mpg.de

†e-mail: kettner@mpi-sb.mpg.de

©ACM, 2005. This is the authors' version of the work. It is posted here by permission of ACM for your personal use. Not for redistribution. The definitive version was published in the *Proceedings of the 2005 ACM Symposium on Solid and Physical Modeling (SPM 2005)*, <http://doi.acm.org/10.1145/nnnnn.nnnnn>

boolean operations. In the context of robot motion planning this example is referred to as *tight passage*, see [Halperin 2002] for the case of planar configuration spaces.

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 gave a more compact and unique representation of SNCs and algorithms realizing the (generalized) set operations based on this representation in [Granados et al. 2003]. The uniqueness of the representation, going back to Nef’s work [Nef 1978], 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 halfplanes and manifold solids, boolean operations (union, intersection, complement, difference, symmetric difference), topological operations (interior, closure, boundary), rotations by rational rotation matrices (rotation angles are approximated up to a given tolerance [Canny et al. 1992]). We follow the exact geometric computation paradigm [Yap 1997] to achieve robustness.

Our representation and algorithms refine the results of Rossignac and O’Connor [Rossignac and O’Connor 1989], Weiler [Weiler 1988], Gursoz, Choi, and Prinz [Gursoz et al. 1990], and Dobrindt, Mehlhorn, and Yvinec [Dobrindt et al. 1993]; see [Granados et al. 2003] for a detailed discussion of the relationship to our structure and other related data structures.

1.1 Our Results

At the time of our previous paper [Granados et al. 2003] we had the data structure and the algorithms fully implemented, but they were based on point-location and intersection-finding routines that were implemented as trivial all-pairs test. So we had a fully functional but slow implementation.

Since then we have written extensive regression tests that achieve code coverage and run automatically with CGAL’s test suite. We use them to validate correctness while we optimize our implementation.

The first obvious optimization was to replace the ray-shooting, point-location and intersection-finding routines by efficient implementations, namely a kd-tree [Havran 2000] and a fast box-intersection algorithm [Zomorodian and Edelsbrunner 2002] respectively. 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. We evaluate the effectiveness of these implementations in comparison to their expected runtime in practice and worst case runtime in theory and confirm their efficiency in practice.

So far, we emphasized completeness in the algorithm design. In the second series of optimizations we evaluate their performance and add optimized implementations for important common cases. We evaluate their effectiveness individually and combined in a series of experiments.

Our current optimized implementation has become efficient enough to compare it with other systems. We selected ACIS R13, a common commercial CAD kernel used in many CAD systems [Spa 2004]. We illustrate with a robustness experiment where the benefits of our exact and robust implementation are. Further on, we run the same experiments that we used above for our implementation with ACIS and we are pleased to see similar runtime results. It should be said that we are comparing apples with oranges here and that it is not clear which implementation should be the favorite, see the discussion in Section 8. Anyhow, we 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: We omit the underlying theory of Nef polyhedra and refer to [Nef 1978; Bieri 1995] and our previous paper [Granados et al. 2003]. We summarize our data structure in Section 2 and our algorithms for generalized set operations in Section 3. We discuss their complexity in Section 4, give details about the implementation in Section 5, and confirm these predictions with a series of experiments in Section 6. We describe the optimizations for common cases and their experimental evaluation in Section 7 and compare our implementation with ACIS R13 in Section 8.

2 Data Structures

Definition 2.1 (Nef polyhedron). A *Nef-polyhedron* in dimension d is a point set $P \subseteq \mathbb{R}^d$ generated from a finite number of open halfspaces by set complement and set intersection operations [Nef 1978].

Set union, difference and symmetric difference can be reduced to intersection and complement. Set complement changes between open and closed halfspaces, 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 and we restrict ourselves to three dimensions.

With the *Reduced Würzburg Structure* [Bieri 1996] a unique and complete data structure was introduced for Nef polyhedra. The key observation behind this data structure is that it suffices to represent the local neighborhood around each minimal element in the incidence relation among faces. If a Nef polyhedron is bounded, its minimal elements are the vertices. So it is convenient (conceptually and, in particular, in the implementation) to only deal with bounded polyhedra. For this reduction we use a so called *infiximal box* [Mehlhorn and Seel 2003] to symbolically bound an unbounded Nef polyhedron¹.

In our representation for three-dimensional Nef complexes, we use two main data structures: The *Sphere Maps* represent the local neighborhoods at each vertex, see Figure 2 for an example of one sphere map. The *Selective Nef Complex Representation* provides a more easily accessible polyhedron representation. It additionally stores the connections between the local sphere maps, see Figure 3 for an example. We describe both data structures in more detail.

2.1 Sphere Map

We conceptually intersect the local neighborhood of a vertex with a small ϵ -sphere. We obtain a planar map on the sphere (Figure 2), which together with the set-selection mark for each item forms a two-dimensional Nef polyhedron embedded on the sphere. We add the set-selection mark for the center vertex and call the resulting structure the *sphere map* of the vertex. Sphere maps were introduced in [Dobrindt et al. 1993].

We use the prefix *s* to distinguish the elements of the sphere map from the three-dimensional elements. An *svvertex* corresponds to

¹We will not use unbounded Nef polyhedra in our experiments and therefore skip all the details that can be found in [Granados et al. 2003]. Nonetheless, everything said about the data structures and algorithms extends to the case of unbounded polyhedra, but for the price of a runtime and memory increase.

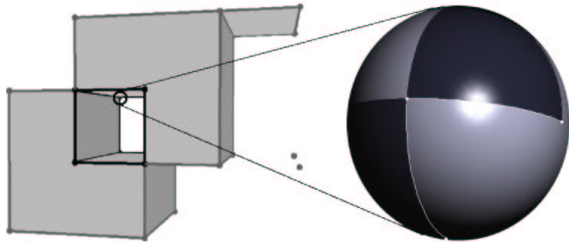


Figure 2: An example of a sphere map. The different colors indicate selected and unselected faces.

an edge intersecting the sphere. An *sedg*e 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 *sfac*e corresponds to a volume.

2.2 Selective Nef Complex Representation

Having sphere maps for all vertices of our 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 them². We discuss features in increasing order of dimension; see also Figure 3.

We store two oppositely oriented edges for each edge. Such an oriented edge can be identified with an *svertex* in a sphere map. 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. We can identify an edge-use with an oriented *sedg*e in the sphere map, or, in the special case also with an *sloop*. 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. The volume boundary decomposes into different connected components, the *shells*. 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. 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.

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.

3 Algorithms

Here 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.

3.1 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 described in [Seel 2001b; Seel 2001a] and explain the changes needed here.

²We actually depart slightly from the original definition of faces in a Nef polyhedron; we represent the maximally connected components of a face.

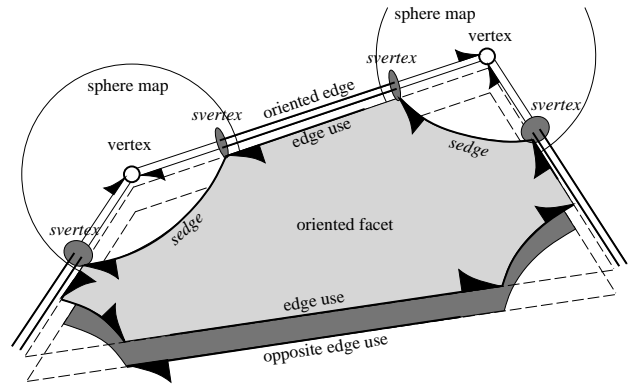


Figure 3: An SNC. We show only one facet with four vertices, the sphere maps of two of the vertices, the connecting edges, and both oriented facets. Shells and volumes are omitted for this example.

The boolean operations on the planar Nef polyhedra work in three steps—overlay, selection, and simplification—following [Rossignac and O’Connor 1989]. The overlay computes the conventional planar-map overlay of the two input polyhedra with a sweep-line algorithm [Mehlhorn and Näher 1999, 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 [Nef 1978; Bieri 1995] 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 [Cormen et al. 1990] to efficiently update the pointers in the half-edge data structure associated with the regions.

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 .

3.2 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 [Nef 1978; Bieri 1995] that:

- x is part of a volume iff its local sphere map is trivial (only one *sfac*e 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 and 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. This mark is different from at least one *sedge* or *sface* in between.
- x is a vertex v iff its local sphere map is none of the above.

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.

3.3 Synthesizing the SNC from Sphere Maps

Given the sphere maps for a particular polyhedron, we describe how we can synthesize the full SNC structure. We proceed in the order of increasing dimension:

1. We identify *svertices* that we want to link together as edges. We form an encoding for each *svortex* consisting of: (a) a normalized line representation for the supporting line, e.g., the normalized Plücker coordinates of the line [Stolfi 1991], (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 *svortex*. We sort all encodings lexicographically. Consecutive pairs in the sorted sequence form an edge.
Plücker coordinates are normalized to have a leading coefficient of one, or by dividing them by the greatest common divisor of all six coefficients in case of integers.
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.
4. Shells are found with a graph traversal. The nesting of shells is resolved with ray shooting from the lexicographically smallest vertex. Its sphere map also gives the set-selection mark for this volume by looking at the mark in the sphere map in $-x$ direction. This concludes the assembly of volumes.

3.4 Boolean Operations

We represent Nef polyhedra as SNCs. We can trivially construct an SNC for a halfspace. We can also construct it from a polyhedral surface [Kettner 1999] representing a closed 2-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 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.
3. Given the two sphere maps for a candidate vertex, we apply the boolean operation on sphere maps, see Section 3.1, to obtain the resulting sphere map.
4. Based on the classification of local neighborhoods in Section 3.2, we check if the resulting sphere map represents a vertex, in which case we keep it for the final SNC synthesis step, and otherwise we discard it.

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 3.2). 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.

4 Complexity

We analyzed the runtime complexity in our previous paper [Granas et al. 2003], but we did not give details on the runtime of two parts—the ray shooting needed for shells in the synthesis algorithm, whose runtime we denoted T_{\uparrow} , and the intersection algorithm, whose runtime we denoted T_I . The intersection algorithm finds all edge-edge and edge-face intersections, but also locates the position of a vertex of one polyhedron in the other polyhedron.

Let the total complexity n of a Nef polyhedron be the number of vertices, edges, and facets.

4.1 Kd-Tree

We chose to implement a kd-tree [Havran 2000] for the ray-shooting queries and we use it also for the point-location queries. We use the vertex set as a criterion for building the tree; we split the vertex set along alternating axes at its median vertex. We create a leaf when we have at most two vertices left, which guarantees logarithmic depth.

We store the vertices, but also all edges and facets in the leafs that they intersect, e.g., long edges and large facets can be stored in up to $O(n)$ leafs, and there may exist leafs with $O(n)$ items stored. The tight worst-case space bound is $\Theta(n^2)$ and construction takes $O(n^2 \log n)$.

The worst case result for vertical rays is better; they intersect at most $O(\sqrt[3]{n})$ cells, however, each could store $O(n)$ items, and we need logarithmic search time for locating a neighboring cell in our walk through the kd-tree. In total, we get a worst case runtime of $O(n\sqrt[3]{n} \log n)$.

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.

Of course we use the kd-tree since we expect it to perform much better in practice. The usual heuristic assumption is a well-shaped geometry with the following consequences: Each edge or facet is stored in a constant number of cells and each cell contains only a constant number of items. It suffices if these assumptions hold for an amortized analysis, 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. For certain well-shaped polyhedra, the ray-shooting query might even drop to $O(\log n)$. We study these runtimes experimentally in Section 6.

4.2 Box-Intersection Algorithm

We use the fast box-intersection algorithm described as streamed segment tree in [Zomorodian and Edelsbrunner 2002] 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 $O(n^2)$, but we expect it to be close to the true output complexity of the edge-edge and edge-facet intersection problem.

4.3 Expected 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 a manifold solid.

Given a polyhedron of complexity n , the complement runs in linear time. The topological operations *closure*, *boundary*, *interior*, *exterior*, and *regularization* require a simplification step and run in $O(n \cdot \alpha(n))$ worst-case time where $\alpha(n)$ denotes the inverse Ackermann function from the union-find structures in the simplification algorithm. However, we need to update the kd-tree afterwards, either the expected $O(n \log n)$ time or the $O(n^2 \log n)$ worst-case time.

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) $O((n+m) \log^3(n+m) + s)$ worst-case time to find all edge-facet and edge-edge intersections, where s is the number of pairwise intersecting bounding boxes of edges and facets, which we expect to be close to the true number of intersecting edges and facets and this in turn we expect³ to be k , (iii) $O((n+m+s) \log(n+m))$ worst-case time for the overlay of all $n+m+s$ sphere maps, and (iv) $O(k\sqrt[3]{k} \cdot \log k)$ expected time for the synthesis including the kd-tree construction.

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

³Except for arcane degenerate cases, all edge-edge and edge-facet intersections will contribute a vertex in the final boolean-operation result.

5 Implementation

Our implementation is part of the CGAL release 3.1, which was released in December 2004. CGAL, the *Computational Geometry Algorithms Library*, is an Open Source C++ software library⁴ [Fabri et al. 2000]. 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 [Kettner 1999], 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 [Canny et al. 1992]). We follow the exact geometric computation paradigm [Yap 1997] 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 geometric primitives, some data structures, and the generic sweep-line algorithm.

6 Experiments

In this section we experimentally evaluate the runtime behavior of our implementation, in particular the binary boolean operations. We have several experiments that support the expected runtime analyzed in Section 4, and we have designed experiments to stress our implementation with worst-case scenarios.

Besides the total runtime, we list also the runtime of important subroutines in the binary boolean operation to illustrate the distribution of resources, potential bottlenecks, and further places for optimizations. We summarize the important subroutines here in their order of usage, see Section 3.4 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.
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).

⁴<http://www.cgal.org>

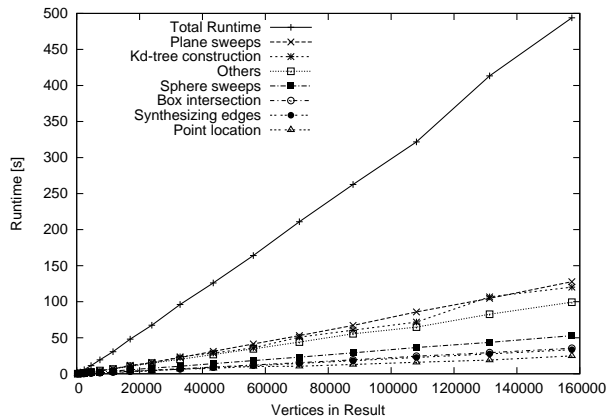


Figure 4: Total runtime and runtime distributed over the main sub-routines for our implementation in the TETGRID experiment.

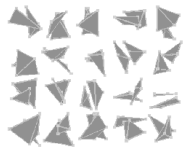
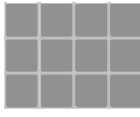
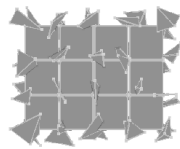
8. **Others:** all other parts not listed explicitly in the same graph, so parts which have no critical worst-case or no interesting practical runtime contributions.

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 which 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 [Hert et al. 2002] 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-sb.mpg.de/~kettner/proj/Nef/>.

6.1 Balanced Binary Operations

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

T *C* *TUC*

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 TUC .

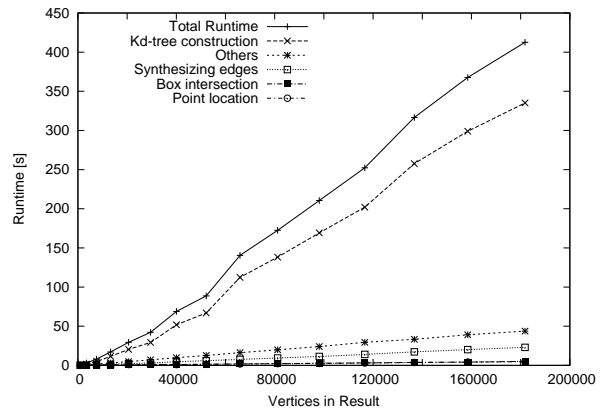
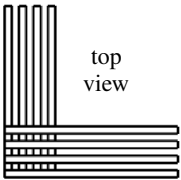
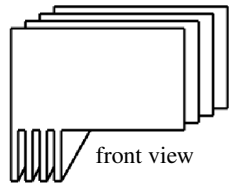


Figure 5: Total runtime and runtime distributed over the main sub-routines for our implementation in the QUADRATICWALLGRID experiment. Note that the plane sweep and the ray shooting are not executed in this experiment.

In Figure 4 we see the total runtime in the first graph 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 $O(n \log n)$, the total runtime must have a logarithmic factor, too.

6.2 Binary Operation with Quadratic Result

In the next test series we again start with input objects of equal size, but we achieve a worst-case output complexity, as described in the QuadraticWallGrid experiment. We run this experiment for $N = i * 10, i = 1, \dots, 15$ on machine 2. We see in Figure 5 that the construction of the kd-tree is responsible for the majority of the runtime. Recall that the kd-tree is constructed for the result. In the graph, the $O(k \log k)$ construction time becomes apparent.

top view front view

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 $W \cup W'$

6.3 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

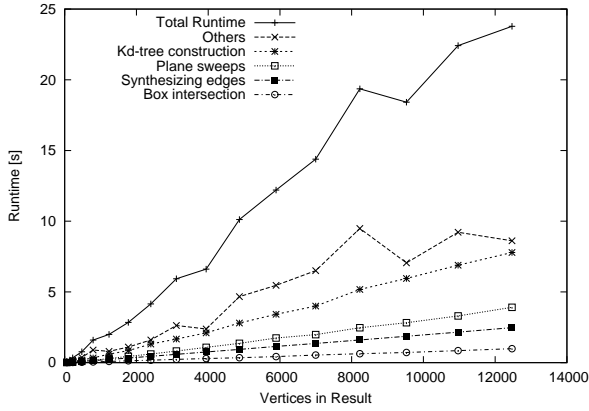


Figure 6: Total runtime and runtime distributed over the main sub-routines for our implementation in the COMPLEXMINUSSIMPLE experiment.

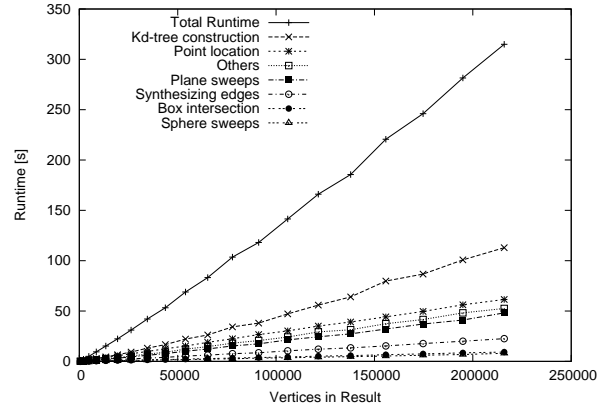
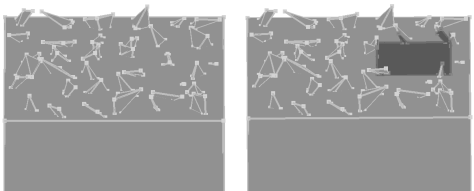


Figure 7: Total runtime and runtime distributed over the main sub-routines for our implementation in the COMPLEXFACET experiment.

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.



CUG $(CUG) \setminus c'$

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 $C' = CUG$.
4. Create a cube c of size 2^3 such that its vertices match centers of the grid cells.
5. COMPLEXMINUSSIMPLE: Measure time for $C' - c$

For the COMPLEXMINUSSIMPLE experiment, we measure the runtime for $N = i * 5, i = 1, \dots, 40$ on machine 1. There is no sub-routine, which is dominating the runtime. Still the kd-tree construction is most time consuming.

6.4 Complex Facet

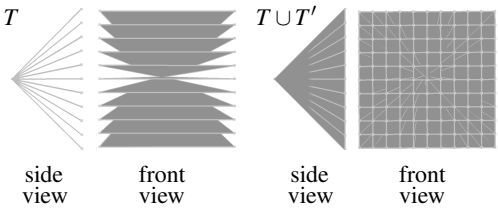
In large Nef polyhedra of complexity n , there is rarely a single supporting plane with complexity $O(n)$. On the other side, worst-

case examples do not seem as artificial as in the case of the sphere sweeps. We use the COMPLEXFACET experiment from Section 6.3 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 7 shows the result of the respective test series for $N = i * 20$ with $i = 1, \dots, 10$. The runtime looks close to linear but has actually 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.

6.5 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.



T $T \cup T'$

side view front view side view front view

Experiment COMPLEXSPHEREMAP

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

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 8 shows the result of a test series of this

scenario with $N = i * 50, i = 1, \dots, 16$. Again, the kd-tree dominates the runtime.

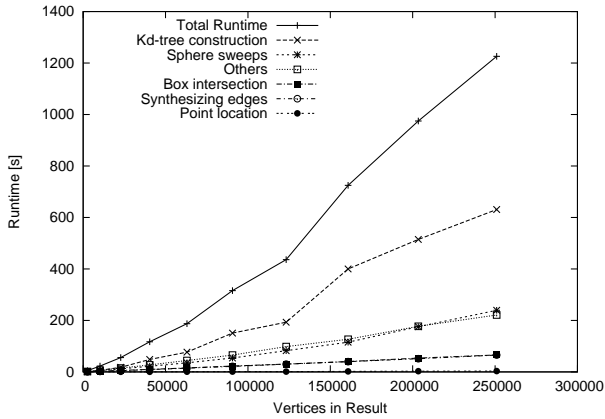


Figure 8: Total runtime and runtime distributed over the main subroutines for our implementation in the COMPLEXSPHEREMAP experiment for $N = i * 50, i = 1, \dots, 16$.

6.6 Point Location and Ray Shooting

We use a kd-tree for point location and ray shooting. For ray shooting we shoot a ray in the negative x -direction in order to find the first intersection. For point location, we locate the cell containing the query point and within the cell we find the closest object, from which we deduce the answer to the query. Since ray shooting never comprises a major part of the total runtime, we concentrate on the point location. The expected runtime is $O(\log n)$ per query, but can be as bad as $O(n \log n)$.

Figure 9 shows the runtime spent for 10000 random point location queries on the TETGRID experiment data for $N = 1, \dots, 25$, respectively. Using a logarithmic scale for the x -axis, the curve looks linear or even better. It confirms the expected logarithmic runtime.

7 Optimizations

We have seen in the previous section that certain subroutines of the algorithm are very dominant and others are less dominant, maybe to

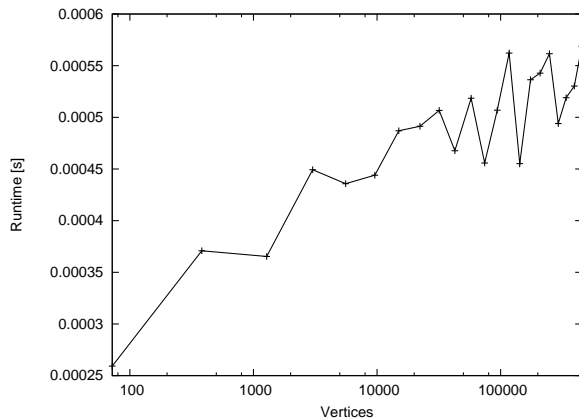


Figure 9: Runtime per point location on the TETGRID experiment data for $N = 1, \dots, 25$, respectively.

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.

7.1 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 1 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:

optimizations			number of sweeps	runtime	
(i)	(ii)	(iii)		sweeps	total
-	-	-	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

Table 1: 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 $N = 16$.

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

In the latter case, the vertex is cloned and the marks of the clone are evaluated from the old marks, the mark of the volume and the boolean function. Afterwards the sphere map is simplified as usual, which could 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. Again, the simplification is done as usual.

As a result of these optimizations, 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.

- (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 vertices 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 in the inside of the other polyhedron is absorbed into this volume. Here, the selection routine assigns the same mark to each svertex, sedge and sface on the sphere map. This happens when the boolean operation *bop* applied to the mark of the determined volume and any 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.

7.2 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 2 shows the effect of this optimization. We use the same test scenario as in Table 1 and perform two runs, one with a plane sweep for every supporting plane and one where we perform the necessary plane sweeps only.

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

Table 2: The number and runtime improvement for the plane-sweep optimization shown with a TETGRID experiment for $N = 16$.

7.3 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 side, 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.

Input Complexity	tests / intersections	
	box inters.	kd-tree
172	6.64s	13.33s
1012	5.88s	13.37s
3100	5.78s	14.28s
7012	5.79s	13.29s
13324	5.76s	13.74s
22612	5.71s	13.43s
35452	5.79s	14.07s
62632	5.77s	13.03s
101044	5.79s	13.61s

Table 3: Number of intersection candidates tested per existing intersections.

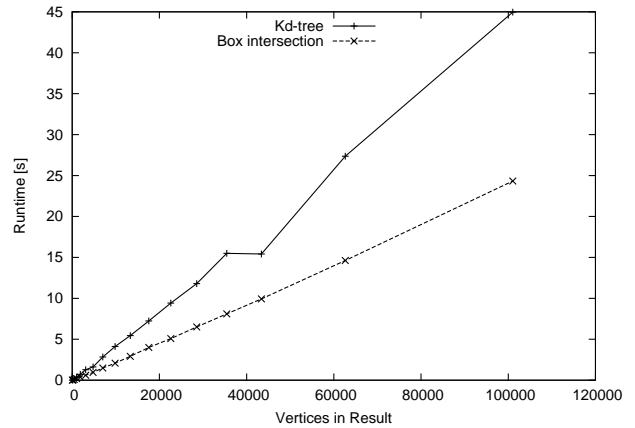


Figure 10: Runtime comparison between the box-intersection algorithm and the kd-tree on the TETGRID experiment.

Figure 10 shows the result of a test series with the TETGRID experiments for $N = 1, \dots, 20$. The box-intersection algorithm is clearly preferable. We also counted the number of intersection candidates that are tested in each algorithm, see Table 3 for the result. Indeed, the kd-tree tests more (redundant) candidates.

8 Comparison with ACIS R13

We compare our implementation with ACIS R13, a common commercial CAD kernel used in many CAD systems [Spa 2004]. 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. 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. 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.

8.1 Balanced Binary Operations

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

The results in Table 4 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.

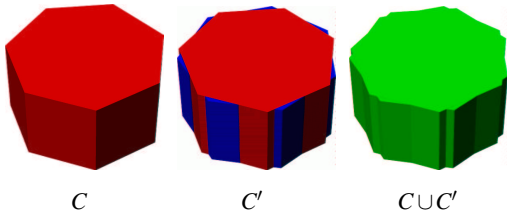
N	result vertices	runtime [s]	
		ACIS R13	Nef 3D
3	338	0.29	0.61
4	1135	0.63	2.53
5	2390	1.37	5.71
6	4548	2.79	11.37
7	7383	5.29	19.26
8	11555	10.13	30.61
9	16998	14.27	48.02
10	23883	22.81	67.31
11	32892	25.58	96.12
12	43418	35.58	126.01
13	56188	55.64	164.05
14	70827	swapping	211.02
15	87871	swapping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

Table 4: Comparison of ACIS R13 and our Nef polyhedron with the TETGRID experiment.

8.2 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, but we are not aware of any 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.

We designed the simple ROTCYLINDER experiment 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.



Experiment ROTCYLINDER

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

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.

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

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 [Canny et al. 1992]. The CGAL implementation is division free but slower than the algorithm described in [Canny et al. 1992]. The runtime for finding such an exact rotation matrix amounts to a non-negligible fraction for small ε , as can be seen in Table 6.

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

Table 6: 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 $\frac{\alpha}{10000}$.

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 5 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 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 7 show that both algorithms benefit from this change; our algorithm by a factor of

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

Table 7: 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.

about two, and ACIS by a factor of about three. Nonetheless, ACIS aborts the union computation for an angle below 10^{-4} .

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. We repeat this experiment with ACIS and compare it with the results of our implementation from Section 6.

Table 8 shows the results of a test series with $N = i * 3, i = 1, \dots, 13$. Here, the difference between ACIS and our algorithm is quite pronounced with ACIS being a factor of about six faster than our implementation. A notable difference might be in the software interface; ACIS modifies the first input object to become the result, while our implementation creates the result from scratch without modifying the two input polyhedra.

9 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.

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.

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

Table 8: Comparison of ACIS R13 and our Nef polyhedron with the COMPLEXMINUSSIMPLE experiment. ACIS is a factor of twenty faster.

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

Nef complexes are defined by planes. It follows from the work on the *Selective Geometric Complexes* (SGC) of Rossignac and O'Connor [Rossignac and O'Connor 1989] that our data structures extend immediately to complexes defined by curved surfaces. However, some of the algorithmic steps become difficult and need further work, 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.

The exact geometric computing paradigm [Yap 1997] allows us to rely in our implementation on correctness proofs from theory, a unique 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 [Fortune 1999].

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 Miguel Granados for the 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

- BERBERICH, E., HEMMER, M., KETTNER, L., SCHÖMER, E., AND WOLPERT, N. 2005. An exact, complete and efficient implementation for computing planar maps of quadric intersection curves. To appear in Proc. 21st Symposium on Computational Geometry.
- BIERI, H. 1995. Nef polyhedra: A brief introduction. *Computational Suppl. Springer Verlag* 10, 43–60.
- BIERI, H. 1996. Two basic operations for Nef polyhedra. In *CSG 96: Set-theoretic Solid Modelling: Techniques and Applications*, Information Geometers, 337–356.
- CANNY, J., DONALD, B. R., AND RESSLER, E. K. 1992. A rational rotation method for robust geometric algorithms. In *Proc. 8th Annu. ACM Symposium on Computational Geometry*, 251–260.
- CORMEN, T. H., LEISERSON, C. E., AND RIVEST, R. L. 1990. *Introduction to Algorithms*. MIT Press.
- DOBRINDT, K., MEHLHORN, K., AND YVINEC, M. 1993. 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, 314–324.
- FABRI, A., GIEZEMAN, G.-J., KETTNER, L., SCHIRRA, S., AND SCHÖNHERR, S. 2000. On the design of CGAL a computational

- geometry algorithms library. *Softw. – Pract. Exp.* 30, 11, 1167–1202.
- FORTUNE, S. 1997. Polyhedral modelling with multiprecision integer arithmetic. *Computer-Aided Design* 29, 123–133.
- FORTUNE, S. 1999. Vertex-rounding a three-dimensional polyhedral subdivision. *Discrete and Computational Geometry* 22, 593–618.
- GRANADOS, M., HACHENBERGER, P., HERT, S., KETTNER, L., MEHLHORN, K., AND SEEL, M. 2003. Boolean operations on 3D selective Nef complexes: Data structure, algorithms, and implementation. In *Proc. 11th Annu. European Sympos. Algorithms (ESA'03)*, Springer, Budapest, Hungary, LNCS 2832, 654–666.
- GURSOZ, E. L., CHOI, Y., AND PRINZ, F. B. 1990. Vertex-based representation of non-manifold boundaries. *Geometric Modeling for Product Engineering* 23, 1, 107–130.
- HALPERIN, D. 2002. Robust geometric computing in motion. *International Journal of Robotics Research* 21, 3, 219–232.
- HAVRAN, V. 2000. *Heuristic Ray Shooting Algorithms*. Ph.D. thesis, Faculty of Electrical Engineering, Czech Technical University, Prague, Czech Republic.
- HERT, S., POLZIN, T., KETTNER, L., AND SCHÄFER, G. 2002. Explab - a tool set for computational experiments. Research Report MPI-I-2002-1-004, MPI für Informatik, Saarbrücken, Germany.
- HOFFMANN, C. M. 1989. *Geometric and Solid Modeling – An Introduction*. Morgan Kaufmann.
- KETTNER, L. 1999. Using generic programming for designing a data structure for polyhedral surfaces. *Computational Geometry: Theory and Applications* 13, 65–90.
- MÄNTYLÄ, M. 1988. *An Introduction to Solid Modeling*. Computer Science Press, Rockville, Maryland.
- MEHLHORN, K., AND NÄHER, S. 1999. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press.
- MEHLHORN, K., AND SEEL, M. 2003. Infimaximal frames: A technique for making lines look like segments. *International Journal of Computational Geometry and Application* 13, 3, 241–255.
- MIDDLEDITCH, A. E. 1994. "The bug" and beyond: A history of point-set regularization. In *CSG 94 Set-theoretic Solid Modelling: Techn. and Appl.*, Inform. Geom. Ltd., 1–16.
- NEF, W. 1978. *Beiträge zur Theorie der Polyeder*. Herbert Lang, Bern.
- ROSSIGNAC, J. R., AND O'CONNOR, M. A. 1989. SGC: A dimension-independent model for pointsets with internal structures and incomplete boundaries. In *Geom. Model. for Product Engin.*, M. Wozny, J. Turner, and K. Preiss, Eds. North-Holland.
- ROSSIGNAC, J. R., AND REQUICHA, A. G. Solid modeling. <http://citeseer.nj.nec.com/209266.html>.
- SEEL, M. 2001. Implementation of planar Nef polyhedra. Research Report MPI-I-2001-1-003, MPI für Informatik, Saarbrücken, Germany, Aug.
- SEEL, M. 2001. *Planar Nef Polyhedra and Generic Higher-dimensional Geometry*. PhD thesis, Universität des Saarlandes, Saarbr., Germany.
- SPATIAL CORP., A DASSAULT SYSTÈMES COMPANY. 2004. *ACIS R13 Online Help*.
- STOLFI, J. 1991. *Oriented Projective Geometry: A Framework for Geometric Computations*. Academic Press.
- WEILER, K. 1988. The radial edge structure: A topological representation for non-manifold geometric boundary modeling. In *Geometric Modeling for CAD Applications*, M. J. Wozny, H. W. McLaughlin, and J. L. Encarnação, Eds., IFIP, 3–36.
- YAP, C. 1997. Towards exact geometric computation. *Computational Geometry: Theory and Applications* 7, 1, 3–23.
- ZOMORODIAN, A., AND EDELSBRUNNER, H. 2002. Fast software for box intersection. *Int. J. Computational Geometry and Applications* 12, 143–172.