Diss. ETH No. 13325

Software Design in Computational Geometry and Contour-Edge Based Polyhedron Visualization

Lutz Kettner

1999

Image Reference

Cover image: visible contour edges of the Matterhorn of Switzerland.
Image in running head: visible contour edges of different projections of Matterhorn.

Diss. ETH No. 13325

Software Design in Computational Geometry and Contour-Edge Based Polyhedron Visualization

A dissertation submitted to the
SWISS FEDERAL INSTITUTE OF TECHNOLOGY ZURICH

for the degree of
Doctor of Technical Sciences

presented by
Lutz Kettner
Diplom-Informatiker
University of Karlsruhe, Germany
born 2.2.1967
citizen of Germany

accepted on the recommendation of
Prof. Dr. Emo Welzl, examiner
Prof. Dr. Jean-Daniel Boissonnat, co-examiner

1999

Acknowledgments

I warmly thank my advisor Emo Welzl for the excellent working atmosphere, the freedom, the encouraging guidance and the strong support throughout these studies. I thank my co-advisor Jean-Daniel Boissonnat for his encouragement of my work and for his help in promoting software design in the computational geometry community.

I am grateful to Hervé Brönnimann, Michael Hoffmann, Ursel and Peter Homann, Nora Sleumer and Martin Will for reading and commenting on previous versions of this thesis. My special thanks goes to Andreas Fabri, Geert-Jan Giezeman, Stefan Schirra and Sven Schönherr for the intensive discussions on C++ design and software engineering since we started with the CGAL kernel almost five years ago. I would also like to thank Dietmar Kühl and Karsten Weihe for the advice and help with C++ and design questions for CGAL. The CGAL project has grown and I want to thank all the members of the project for the inspiring environment – keep on going! I wish to thank Bernd Gärtner, Michael Hoffmann, Andreas Rosenthal, Sven Schönherr, and Michael Schutte for the many fruitful discussions in the meetings in Berlin.

Last but not least, I thank all the people who provided us with numerous example objects for our analysis. In particular, the Federal Office of Topography in Switzerland that provided permission to use their data sets for Albis and Matterhorn that were used in this publication. Part of this research was carried out while I was at the Freie Universität Berlin with a scholarship from the *Graduiertenkolleg "Algorithmische Diskrete Mathematik"*, DFG We 1265/2-1. Support from the Swiss Federal Office for Education and Science (Projects ESPRIT IV LTR No. 21957 CGAL and No. 28155 GALIA) is also acknowledged.

Abstract

In computational geometry efficient algorithms and data structures are designed and analyzed for geometric problems. I address important software design questions to achieve flexible and efficient implementations. I present new design solutions particularly useful for geometric algorithms and data structures, among others a halfedge data structure for polyhedral surfaces. For three-dimensional polyhedral surfaces, visibility algorithms are an important class of algorithms. I present a new approach based on contour edges. The advantages are supported by my extensive experimental study. In particular, I describe a new object-space hidden-surface removal algorithm for three-dimensional polyhedral surfaces. The algorithms also serve as a case study for the software design solutions.

I have successfully applied the recent paradigm of *generic programming* to the domain of geometric algorithms and data structures. Examples are the implementation of my new visibility algorithms and my contributions in designing CGAL, the *Computational Geometry Algorithms Library*. CGAL, written in C++, is being developed by research groups in Europe and Israel. I give an introduction to generic programming and present our extensions, *circulators* and *geometric traits classes*, which solve specific problems in geometric algorithms.

In particular, I present a library design solution for combinatorial data structures such as polyhedral surfaces and planar maps. The design issues considered are flexibility, time and space efficiency, and ease-of-use. I focus on topological aspects of polyhedral surfaces and evaluate edge-based representations with respect to my design goals. A halfedge data structure has been realized. Connections to planar maps and face-based structures are clarified.

The new visibility algorithms for three-dimensional polyhedral surfaces are based on *contour edges*. Given a view point, an edge is a contour edge if it is incident to a front-facing facet and a back-facing facet. The algorithms exploit the fact that the number of contour edges is usually much smaller than the overall number of edges. I provide evidence for, and quantify, the claim that the number of contour edges is small in many situations. For example, an asymptotic analysis of polyhedral approximations of a sphere with Hausdorff distance ε shows that, while the required number of edges for such an approximation is $\Theta(1/\varepsilon)$, the number of contour edges in a random *orthogonal* projection is only $\Theta(1/\sqrt{\varepsilon}\,)$.

Indeed, the number of contour edges is small for many objects, especially for polyhedral approximations of curved surfaces. These findings are based on an experimental study of polyhedral surfaces from several application areas. I analyze, for random orthogonal projections, the expected number of contour edges and the expected number of their intersections in the projection. The number of intersections is a quantity relevant for the runtime of sweep-line algorithms, which I use in my visibility algorithms. The small number of intersections support fast expected running times for this approach.

Concluding from the experimental study, the extraction of the contour edges from the total number of edges, n, is likely to determine the runtime of this algorithms. Thus, a view-independent preprocessing is of interest. One can prepare a data structure of only nearly linear size so that one can answer a query for the contour edges for a particular viewing transformation in $O(\sqrt{n}\log n + n_\mathrm{c})$ for orthogonal projections or in $O(n^{2/3} \operatorname{polylog} n + n_\mathrm{c})$ for perspective projections, where n_c denotes the number of contour edges reported. These results are based on a transformation of the contour-edge reporting problem to a segment stabbing problem.

I present three new visibility algorithms based on contour edges: the computation of the *silhouette* of a three-dimensional polyhedral surface, *object-space hidden-surface removal* for general three-dimensional polyhedral surfaces and a faster and easier specialization of the object-space hidden-surface removal for *terrains*. Let n_c be number of contour edges, int_c the number of their intersections, and k the size of the output. The silhouette can be computed in $O((n_\mathrm{c} + \mathit{int}_\mathrm{c}) \log n_\mathrm{c})$ time. The *visibility map*, the output of the object-space hidden-surface removal algorithm, can be computed for terrains in $O((n_\mathrm{c}+\mathit{int}_\mathrm{c}) \log n_\mathrm{c}+k \log k)$ time. For general polyhedral surfaces, a geometric construction shows that the visibility map cannot be computed within a similar time bound by following this approach. This construction requires locating the visible facet within a hole. If we denote the total time needed for locating all these facets with T_loc then the visibility map for general polyhedral surfaces can be computed in $O((n_\mathrm{c} + \mathit{int}_\mathrm{c}) \log n_\mathrm{c} + n \log n + k \log k) + T_\mathrm{loc}$ time. The expected runtime of these algorithms for a particular object can be derived almost directly from the results of the experimental study.

The implementation of these visibility algorithms is based on the classical Bentley-Ottmann sweep-line algorithm for segment intersection. Degeneracies are successfully handled using symbolic perturbation of the viewing transformation. The geometric predicates in the algorithms are evaluated exactly by using exact and bounded integer arithmetic with efficient built-in number types. For each predicate, bounds on the required precision in the arithmetic are given. The new software design solutions proved to be very useful in these implementation.

Zusammenfassung

In der Algorithmischen Geometrie werden effiziente Algorithmen und Datenstrukturen für geometrische Probleme entworfen und analysiert. Ich bespreche wichtige Entwurfsfragen für flexible und effiziente Implementierungen und präsentiere neue Designlösungen, die besonders geeignet für geometrische Algorithmen und Datenstrukturen sind, unter anderem eine Datenstruktur mit Halbkanten für polyedrische Flächen. Sichtbarkeitsalgorithmen sind eine wichtige Algorithmenklasse für polyedrische Flächen. Ich führe einen neuen, auf Konturkanten basierten Ansatz ein. Seine Vorteile werden durch eine umfassende, experimentelle Studie belegt. Insbesondere beschreibe ich einen neuen Algorithmus, der die sichtbaren Flächen von dreidimensionalen polyedrischen Flächen im Objektraum berechnet. Die Algorithmen dienen auch als Fallstudie für die Designlösungen.

Ich habe das junge Paradigma des *Generischen Programmierens* erfolgreich auf geometrische Algorithmen und Datenstrukturen angewandt, zum Beispiel beim Implementieren der Sichtbarkeitsalgorithmen und bei meinem Beitrag zum Design von CGAL, der *Computational Geometry Algorithms Library*. CGAL wird von mehreren Forschungsgruppen in Europa und Israel in C++ entwickelt. Ich beschreibe Generisches Programmieren und unsere Neuerungen, *Circulators* und *Geometric Traits Classes*, die jeweils spezifische Probleme in geometrischen Algorithmen lösen.

Insbesondere präsentiere ich eine Lösung zum Entwurf von kombinatorischen Datenstrukturen in Bibliotheken, beispielsweise polyedrische Flächen oder planare Karten mit Blick auf Flexibilität, Zeit- und Platzeffizienz, sowie Benutzerfreundlichkeit. Mein Schwerpunkt liegt auf den topologischen Aspekten der Modellierung. Ich bewerte kantenbasierte Strukturen und realisiere eine Darstellung mit Halbkanten. Die Verbindungen zur planaren Karte werden aufgezeigt.

Die neuen Sichtbarkeitsalgorithmen für dreidimensionale polyedrische Flächen basieren auf *Konturkanten*. Eine Kante ist eine Konturkante, wenn sie zu einer dem Betrachter zugewandten Facette, und eine abgewandten Facette benachbart ist. Die Algorithmen nutzen die Tatsache aus, daß die Anzahl der Konturkanten häufig deutlich unter der Anzahl aller Kanten liegt. Zum Beispiel, eine asymptotische Analyse der polyedrischen Approximation von einer Kugel mit Haussdorff Abstand ε zeigt, daß, während die Anzahl benötigter Kanten mit $\Theta(1/\varepsilon)$ wächst, die erwartete Anzahl Konturkanten für eine zufällige, orthogonale Projektion nur mit $\Theta(1/\sqrt{\varepsilon}\,)$ wächst.

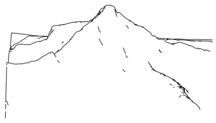
Tatsächlich ist die Anzahl Konturkanten für viele Objekte sehr gering, insbesondere für polyedrische Approximation von gekrümmten Flächen, wie eine experimentelle Studie von polyedrischen Flächen aus verschiedenen Anwendungsgebieten gezeigt hat. Ich analysiere die erwartete Anzahl von Konturkanten und deren Schnittpunkte in der Projektionsebene für zufällige, orthogonale Projektionen. Gerade die Anzahl Schnittpunkte ist für die Laufzeit von *Sweep-Line* Algorithmen interessant, wie ich sie auch für die Sichtbarkeitsalgorithmen einsetze. Die ebenfalls sehr geringe Anzahl zu erwartender Schnittpunkte läßt schnelle Laufzeiten für meinen Ansatz erwarten.

Der Studie folgend, kann die Laufzeit dieser Algorithmen maßgeblich durch das Finden der Konturkanten bestimmt werden. Daher ist eine vom Betrachterstandpunkt unabhängige Vorverarbeitung interessant. Man kann eine Datenstruktur von ungefähr linearer Größe vorbereiten, so daß man für Anfragen mit einem Betrachterstandpunkt die Konturkanten in $O(\sqrt{n}\log n + n_{\mathrm{c}})$ für orthogonale Projektionen und in $O(n^{2/3}\operatorname{polylog} n + n_{\mathrm{c}})$ für perspektivische Projektionen finden kann, wobei n_{c} die Anzahl Konturkanten bezeichnet. Diese Ergebnisse basieren auf der Trannsformation des Problems, Konturkanten zu finden, auf das Problem, Segmente mit einer Geraden aufzuspiessen.

Ich bespreche drei neue, auf Konturkanten basierende Sichtbarkeitsalgorithmen: die Berechnung der *Silhouette* einer dreidimensionalen polyedrischen Fläche, die *Berechnung sichtbarer Flächen im Objektraum* für allgemeine, dreidimensionale polyedrische Flächen und eine Spezialisierung auf Geländeflächen, die einfacher und schneller ist. Sei n_{c} die Anzahl Konturkanten, int_{c} die Anzahl ihrer Schnittpunkte in der Projektionsebene, und k die Größe der Ausgabe. Dann kann die Silhouette in $O((n_{\mathrm{c}} + int_{\mathrm{c}})\log n_{\mathrm{c}})$ Zeit berechnet werden. Die *Sichtbarkeitskarte*, das Ergebnis der Berechnung sichtbarer Flächen im Objektraum, kann für Geländeflächen in $O((n_{\mathrm{c}} + int_{\mathrm{c}})\log n_{\mathrm{c}} + k\log k)$ Zeit berechnet werden. Eine geometrische Konstruktion zeigt, daß, dem gleichen Ansatz folgend, die Sichtbarkeitskarte für allgemeine, polyedrische Flächen nicht in den gleichen Zeitschranken berechnet werden kann. Diese Konstruktion benötigt die Suche einer sichtbaren Facette hinter einem Loch in der Fläche. Wenn wir die Gesamtzeit für die Suche solcher Facetten als T_{loc} zusammenfassen, können die Sichtbarkeitskarte für allgemeine, polyedrische Flächen in $O((n_{\mathrm{c}} + int_{\mathrm{c}})\log n_{\mathrm{c}} + n\log n + k\log k) + T_{\mathrm{loc}}$ berechnet werden. Die erwartete Laufzeit dieser Algorithmen kann für ein bestimmtes Objekt fast unmittelbar aus den Ergebnissen der experimentellen Studie gefolgert werden.

Die Implementierung der Sichtbarkeitsalgorithmen basiert auf dem klassischen Bentley-Ottmann Sweep-Line Algorithmus zur Berechnung aller Schnitte von Segmenten in der Ebene. Degeneriertheiten werden durch das symbolische Perturbieren der Betrachterrichtung gelöst. Die geometrischen Prädikate werden mit Hilfe exakter, aber beschränkter Ganzzahlarithmetik der eingebauten Zahlentypen exakt berechnet. Ich gebe Schranken für die benötigte Genauigkeit aller Prädikate. Die neuen Designlösungen haben sich bei der Implementierung sehr bewährt.

Contents



Chapter 1

Introduction

Computational geometry is the sub-area of algorithm design that deals with the design and analysis of algorithms for geometric problems involving objects like points, lines, polygons, and polyhedra. Over the past two decades, the field has developed a rich body of solutions to a huge variety of geometric problems including intersection problems, visibility problems, and proximity problems. A number of fundamental techniques have been designed, and key problems and problem classes have emerged; see the textbooks [28, 55, 63, 113, 129, 144, 154, 159] and the handbooks [85, 164].

Geometric algorithms arise in various areas of computer science. Computer graphics and virtual reality, computer aided design and manufacturing, solid modeling, robotics, geographical information systems, computer vision, shape reconstruction, molecular modeling, and circuit design are well-known examples.

To a large extent the theory has been developed with asymptotic worst-case complexity analysis and under the assumption of the real RAM model. Computations with real numbers are assumed to be in constant time. For many (perhaps most) algorithms and problems this is a justified assumption that can be perfectly simulated with finite precision numbers if the input has limited precision.

In 1996 the Computational Geometry Impact Task Force published a report, *Application Challenges to Computational Geometry* [40]. Although crediting the remarkable success of the field in theory, the report now demanded to address also the applicability in practice. Recommendation number one on the list of four was "production and distribution of usable (and useful) geometric codes". Where are the difficulties in doing so?

There are four major reasons why implementing geometric algorithms in particular is seen to be more difficult than in other fields [40, 166]:

1. Algorithms in geometry are among the most advanced algorithms in algorithm design and they make frequent use of complicated data structures.

Furthermore, software engineering has ignored the difficulties in algorithm engineering for quite a while. For example, the main focus in object-oriented design is on data abstraction, data encapsulation, relationships among data, its reuse and the design of large-scale systems. Only recently have implementations of algorithms been rediscovered as an active topic in software engineering, for example with the generic programming paradigm [146, 147, 189].

2. The asymptotic worst-case complexity analysis does not match the practical need for two reasons. In practice, the constant hidden in the asymptotic analysis can easily outweigh asymptotic factors, such as log-factors, and the worst-case usually depends on a general input model that may be unrealistic.

   A good example is the space complexity of binary space-partition (BSP) trees for n triangles in three-dimensions. Their theoretical space complexity is $\Theta(n^2)$ [55]. Quadratic space would make them useless in computer graphics despite their applicability, such as for hidden-surface removal. But it is known that the space remains linear in practice and binary space-partition trees are one of the few success stories where a sophisticated data structure made it into practice, for instance, into the performance demanding gaming industry with the first-person shooters Doom (2d-BSP) and Quake (3d-BSP).

   More realistic input models, such as fatness, can help understanding the practicability of algorithms (for BSP see for example [3]). Some input models and their interrelations are discussed in [54].

3. Theoretical papers assume exact arithmetic with real numbers. The correctness proofs of the algorithms rely on exact computation, and replacing exact arithmetic by imprecise built-in floating-point arithmetic does not work in general. Geometric algorithms in particular are sensitive to rounding errors since numerical data and control flow decisions have usually a strong interrelation. The numerical problems may destroy the geometric consistency that an algorithm may rely on. As a result the program may crash, may run into an infinite loop, or – perhaps worst of all – may produce unpredictable erroneous output.

   The requirements on the arithmetic vary with the algorithm. Some algorithms require only sign computations of polynomial expressions of bounded degree in the input variables. Others require unbounded degree or algebraic roots. Various packages for exact arithmetic are available for different needs [15, 36, 75, 103, 111, 172, 196]. Another approach is to redesign the algorithm to cope with inexact arithmetic. Usually the output is only an approximation of the exact solution [96, 110, 137, 138]. As a common prerequisite for exact arithmetic the input is rounded, either to convert the input into the format required for the arithmetic (rounding floating point to integer), or to lessen the precision requirements on the arithmetic [86, 139, 163].

4. Often, theoretical papers exclude degenerate configurations in the input. Typically, these degeneracies are specific to the algorithm and the problem, and would involve the treatment of special cases in the algorithm. Simple examples

of configurations considered as degenerate are duplicate points in a point set or three lines intersecting in one point. For some problems, it is not difficult to handle the degeneracies, but for other problems the special case treatment distracts from the solution of the general problem and it can amount to a considerable fraction of the coding effort.

In theory, this approach of excluding degeneracies from consideration is justified by the argument that degenerate cases are very rare in the set of all possible input over the real numbers, i.e., they have zero probability if the input set is randomly chosen over the real numbers. Another argument is that it is first of all important to understand the general case before treating special cases.

In practice, however, degenerate input occurs frequently. For instance, the coordinates of the geometric objects may not be randomly chosen over the real numbers, but lie on a grid. They may be created by clicking in a window in a graphical user interface. In some applications, what are called degeneracies are even high-valued design criteria. In architecture features of buildings do align on purpose. As a consequence, practical implementations usually must address the handling of degeneracies.

General approaches in handling degeneracies are symbolic perturbation [64, 67, 140, 170, 197] or randomized perturbation with performance and correctness guarantee [92, 160].

The community has addressed these topics from time to time and with increasing intensity (several references are given above), but many useful geometric algorithms have not found their way into the application domains of computational geometry yet. This situation is also a severe hindrance for researchers if they wish to implement and evaluate their algorithms. Thus, the constants hidden in the analysis of the otherwise theoretically efficient algorithms often is not known.

1.1 Software Design in Computational Geometry

To remedy this situation the *Computational Geometry Algorithms Library*, CGAL[1], has been started five years ago in Europe in order to provide correct, efficient, and reusable implementations [70, 71, 156]. The library is being developed by several universities and research institutes in Europe and Israel and I am one of the core developers of CGAL since the project started 1995.

The major design goals for CGAL include correctness, robustness, flexibility, efficiency, and ease-of-use [71]. One aspect of flexibility is that CGAL algorithms can be easily adapted to work on data types in applications that already exist. The design goals, especially flexibility and efficient robust computation, led us to opt for

[1] `http://www.cs.uu.nl/CGAL/`

the generic programming paradigm using templates in C++ [38, 145, 178], and to reject the object-oriented paradigm in C++ (as well as in Java). In several appropriate places, however, we make use of object-oriented solutions and design patterns. Generic programming with templates in C++ also provides us with the help of strong type checking at compile time. Moreover, the C++ abstractions used by us do not cause any runtime overhead.

The key goal of generic programming is algorithmic abstraction [146, 147]. Thus, generic programming suggests itself for geometric algorithms because of geometric transformations that link several geometric structures together. For example, duality relates the problem of computing the intersection of halfplanes containing the origin to that of computing the convex hull of their dual points. The Voronoi diagram of a set of points is also dually related to its Delaunay triangulation, and this triangulation can be computed as a lower convex hull of the points lifted to a paraboloid in space [12, 28, 43, 55]. For this kind of problems, the relevance of generic programming is obvious. Much like the problem of finding a minimum element in a sequence, the use of a geometric algorithm, such as computing the convex hull, can have many applications via geometric transformations. In this setting, the algorithm does not operate on the original objects but on their transformed version, and the primitives used by the algorithm must also be applied through the same transformation.

Geometric computing also has been successful in abstracting several paradigms of its own that can be utilized with generic programming. For example, the sweep paradigm is used to compute arrangements of lines of segments, triangulations of polygons, Voronoi diagrams. Randomized incremental algorithms have been abstracted in a framework [28, 45] that uses few primitives. Geometric optimization problems have brought forth the generalized LP-type class of problems whose general solution has been so hugely successful in computing minimum enclosing circles and ellipses, distance between polytopes, linear programs, etc. [80, 191]. In all three cases, one can write the skeleton of an algorithm that will work, given the appropriate primitives.

An introduction to generic programming is given in Chapter 2 and an overview of CGAL is presented in Chapter 3.

Combinatorial structures such as planar maps are fundamental in computational geometry. In order to be useful in practice, a solid library for computational geometry must provide generic and flexible solutions as one of its fundamental cornerstones. Data structures for graphs have not been addressed in generic programming so far. Only first abstractions in accessing graphs for generic graph algorithms have been described [118, 119, 152].

In this thesis, I focus on edge-based representations of three-dimensional polyhedral surfaces and illustrate connections to planar maps and face-based structures which maintain polygons with holes for their facets. I concentrate on the topological aspects and derive solutions applicable to other data structures as well. My design criteria are flexibility, ease-of-use, time and space efficiency. In particular, I want to vary the internal storage organization and the kind of incidences that are actually stored. Additional user data can be integrated easily. A top-level interface ensures ease-of-use and combinatorial integrity. On the other hand, a protected access to the internal representation is granted. Chapter 4 explains the design.

1.2 Contour-Edge Based Polyhedron Visualization

In a classification of ten hidden-surface removal algorithms Sutherland introduced in 1974 the distinction between image-space and object-space algorithms [179]. Object-space methods are characterized to compute the exact answer to the hidden-surface removal problem, which link them to techniques used for visibility problems addressed in computational geometry. A recent survey by Dorward gives an extensive overview of object-space hidden-surface removal algorithms [60], and a book by de Berg also contains ample discussion [51]. The result of an object-space hidden-surface removal algorithm is an exact description of the visible parts of a three-dimensional polyhedral surface. A common representation is the *visibility map* containing the projections of all visible parts including their incidence relations.

Object-space algorithms have various advantages over image-space hidden-surface removal algorithms. They are device independent, especially device resolution independent, such that for high-resolution printers they may well be faster than image-space algorithms. Furthermore, they are useful for shadow computations [50, 74], form-factor computations for radiosity [46], and viewshed analysis in GIS applications. If the image is supposed to be stored in a file, e.g., in PostScript format, an object-space method can usually provide a much denser and shorter description of the image. PostScript output based on painter's algorithm, for example, stores also all invisible facets. Moreover, the output can keep useful symbolic information that can be used, for example, to determine the drawing style for different types of edges. This information can be manipulated later on without recomputing the hidden-surface solution.

Their disadvantages are best illustrated by the advantages of the popular image-space algorithm, the Z-buffer algorithm [39]. The Z-buffer algorithm is easy to implement, is robust, makes no assumptions on the input data, is easily made parallel, and is easily and effectively supported in hardware. All these properties are usually hard to achieve together, which explains the success of the Z-buffer algorithm, though it is worth mentioning that for example aliasing effects are hard to cope with [5]. Another disadvantage of the Z-buffer algorithm is the read-test-write cycle for the Z-buffer on the hardware level which cannot be parallelized and needs two memory cycles in practice, although it could be done in one cycle on specialized memory chips [115]. This memory bottleneck made various three-dimensional

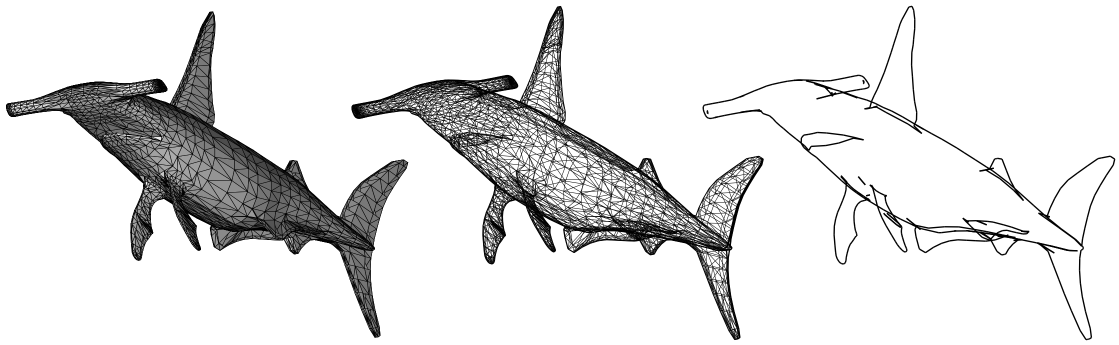


Figure 1.1: An example of contour edges: the hidden-surface rendering[3] of a hammerhead model, all projected edges, and the projected contour edges.

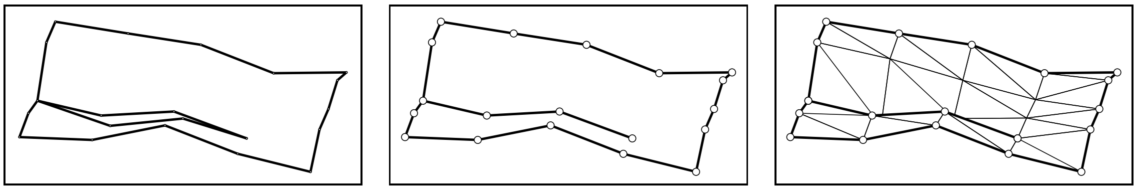
computer games opt for other solutions, for example, (approximate) depth sorting and painter's algorithm [98, 134].

The Z-buffer algorithm scan-converts each polygon before deciding on visibility. Conventional applications of the Z-buffer algorithm use therefore object-space preprocessing steps to reduce the number of polygons. Examples are back-face culling and view-frustum culling, or recent work on finding large occluders that can cull out large portions of the input (see [6] for an overview and a framework incorporating these techniques).

Algorithms on three-dimensional polyhedral surfaces, for example, hidden-surface removal algorithms, can exploit the fact that the number of *contour edges* is usually much smaller than the overall number of edges. An edge of a polyhedral surface is called *contour edge* with respect to a view point if the edge is incident to a front-facing facet and a back-facing facet (see Figure 1.1 for an example). Visibility algorithms in general can utilize contour edges, since the visibility in the viewing plane only changes at contour edges. This was first pointed out by Appel [10] in 1967 when describing an object-space hidden-line removal algorithm.

I provide evidence for and quantify the claim that the number of contour edges is small in many situations. Besides an asymptotic analysis of polyhedral approximations of a sphere, I have undertaken an experimental study based on a number of polyhedral surfaces from several application areas. I analyze the expected number of contour edges, $\overline{n_{\mathrm{c}}}$, and the expected number of intersections of contour edges in a projection, $\overline{int_{\mathrm{c}}}$, (a quantity relevant for sweep-line algorithms) when the viewing direction is picked uniformly at random. I compare these numbers with the number of edges, n, and the expected number of intersections of all edges in a projection, $\overline{int}$. For the 'hammerhead' example as depicted in Figure 1.1, these values rounded to the next integer are $n = 5120$, $\overline{n_{\mathrm{c}}} = 493$, $\overline{int} = 4868$, $\overline{int_{\mathrm{c}}} = 64$. Instead of sampling

[3] The three-dimensional shaded renderings are done with `Geomview`, a software written at the Geometry Center, University of Minnesota. The three-dimensional line drawings are computed with our own implementations.



(i) (ii) (iii)

Figure 1.2: Illustration of the three steps in our hidden-surface removal algorithm. The example is a small synthetic terrain with a valley on the left side: (i) In step 1 the contour edges are extracted. Contour edges are drawn bold in this example. (ii) In step 2 the planar map of the projections of the visible contour edges is formed. (iii) In step 3 the region of the planar map is filled with the visible facets. Note that some of them are clipped at the ridge.

from and averaging over a certain number of random directions, I describe how to compute these expected quantities directly.

From the examples I conclude that, indeed, the number of contour edges is small and the number of intersections of contour edges appears to be even more favorable. The latter is particularly interesting for object-space methods based on the sweep-line algorithm.

Contour edges have already been used for object-space hidden-surface removal algorithms [78, 171]. Both algorithms correspond to some sweep-line algorithm that computes the visibility map in a single pass. The information about contour edges is used only as a classification criterion at intersections of projected edges in order to determine visibility without depth comparisons. However, the contour edges have not been used to enhance the runtime complexity of the algorithm.

In my approach I extend the ideas found in [78, 171] to a three-pass algorithm (see Figure 1.2 for an example of the three steps): First, the contour edges are extracted and projected into the viewing plane. Second, the visible part of the projected contour edges is computed. The visible contour edges form a nice partition of the viewing plane. Third, each region in this partition is filled with the visible facets. Since the visibility can only change at contour edges, only one visible facet is needed to be known for each region to fill the region by traversing adjacent facets until the region is left. Facets need to be clipped at the boundary of the region to obtain the final subdivision of the viewing plane.

From the experiments follows that the number of edges might well be the dominating factor compared to the number of contour edges, n_{c}, or their intersections in the viewing plane, int_{c}. Therefore, a view-independent preprocessing is worth investigating.

A first application of contour edges that is useful for shadow casting is the silhouette of a polyhedral surface. I describe a practical solution for object-space hidden-surface removal based on contour edges and a sweep-line algorithm. The event handling in the sweep-line algorithm simplifies for the hidden-surface removal of terrains represented as triangulated irregular network (TIN) which I have implemented. I use exact integer arithmetic with fixed but sufficient precision based on built-in and efficient number types. As a preprocessing step for the integer arithmetic, the coordinates of the polyhedral surface and the coefficients of the viewing transformation are rounded. I use symbolic perturbation to handle degeneracies. Specifically, I exploit our strict definition of polyhedral surfaces and perturb only the viewing direction. I achieve therefore easily a valid and efficient perturbation scheme. In consequence, I can describe and implement the algorithms without treating degeneracies as special cases.

My work on the hidden-surface removal algorithms serves also as a test case for the software-design concepts introduced in the first part of this thesis. I make successfully use of all concepts including the polyhedral surface as input data structure and the halfedge data structure as foundation for the new planar map, the output structure.

1.3 Outline of the Remaining Chapters

Chapter 2. Generic Programming in Geometric Computing. I begin with a short comparison of generic programming and object-oriented programming. I continue with an introduction to generic programming in C++ and its application to geometric algorithms. New contributions are circulators and traits classes. These evolved during my work for CGAL. Circulators are a variation of the iterator concept for circular sequences. Traits classes are used to separate algorithms and data structures from the geometric primitives they use.

Chapter 3. CGAL, the Computational Geometry Algorithms Library. Related work on geometric software and precursors of CGAL are reviewed. An overview of the library structure is followed by a more detailed description of the design of the geometric kernel and the basic library.

Chapter 4. Designing a Data Structure for Polyhedral Surfaces. I give a strict combinatorial definition of polyhedral surfaces that suits my needs in the hidden-surface removal application well. I evaluate edge-based data structures with respect to a compact and efficient representation for the polyhedral surfaces. A review of related work on polyhedral surfaces in the context of software libraries follows. I refine our general design goals with respect to polyhedral surfaces and present the design, including several examples and a detailed discussion of its realization in C++.

Chapter 5. Contour-Edge Analysis for Polyhedron Projections. I provide evidence for and quantify the claim that the number of contour edges is small in many situations. Besides an asymptotic analysis of polyhedral approximations of a sphere, I have undertaken an experimental study based on a number of polyhedral objects from several application areas.

Chapter 6. Object-Space Hidden-Surface Removal Based on Contour Edges. I discuss related work and present a theoretically efficient preprocessing method for the contour edge extraction. I continue with a Bentley-Ottmann sweep-line algorithm to detect segment intersections in the plane. I extend this algorithm to compute the silhouette of a polyhedral surface and then to compute the visibility map. An easier and more efficient version is derived for terrains. I continue with the techniques used to implement the algorithms. In particular, these are exact integer arithmetic of fixed precision, rounding of vertex coordinates and viewing transformation, and symbolic perturbation of the viewing transformation to cope with degeneracies. I conclude with experiences made with my implementations and a discussion about the design choices made.

Chapter 7. Conclusion. I summarize our results, discuss them, and indicate future directions of this work.

Chapter 2

Generic Programming in Geometric Computing

Generic and flexible designs can be achieved following basically one of the two paradigms; *object-oriented programming* or *generic programming*. Both paradigms are supported in C++: Object-oriented programming, preferably using inheritance from base classes with virtual member functions, and generic programming, preferably using class templates and function templates. Both paradigms are also supported in other languages, but we adhere to the notion used in C++ as the choice made for our implementations and for CGAL.

Object-oriented programming focuses on the modeling of data and the relationships among data. Generic programming focuses on algorithmic design, which makes it more appropriate in our domain. We begin this chapter with a short comparison of both paradigms and continue with an introduction to generic programming in C++ and its application for geometric algorithms. New contributions are circulators and traits classes. These evolved during my work for CGAL as described in the introduction to CGAL in the next chapter. Circulators are a variation of the iterator concept for circular sequences. Traits classes are used to separate algorithms and data structures from the geometric primitives they use. This makes algorithms and data structures flexible and easily adaptable to different geometric primitives.

2.1 Generic and Object-Oriented Programming

The flexibility in the *object-oriented programming paradigm* is achieved with a base class which defines an interface, and with derived classes that implement this interface. Generic functionality can be programmed in terms of the base class and a user can select any of the derived classes wherever the base class is required. The actually used classes can be selected at runtime and the generic functionality can be implemented without knowing all derived classes beforehand. In C++ so-called virtual member functions and runtime type information support this paradigm. The base class is usually a pure virtual base class.

The advantages are the explicit definition of the interface with the base class and the runtime flexibility. There are four main disadvantages for the object-oriented programming paradigm:

1. It cannot provide strong type checking at compile time whenever dynamic casts are used. Dynamic casts are necessary in C++ to achieve the flexibility.
2. It enforces tight coupling through the inheritance relationship [120].
3. It requires additional memory for each object (in C++, the so-called *virtual function table pointer*).
4. It adds for each call to a virtual member function an indirection through the virtual function table [122].

The latter point is of particular interest when considering runtime performance since virtual member functions can usually not be made *inline* and therefore are not subject to code optimization within the calling function[1]. Modern microprocessor architectures can optimize at runtime, but, apart from that that runtime predictions are difficult, these optimization techniques are more likely to fail for virtual member functions. These effects are negligible for larger functions, but small functions will suffer a loss in runtime of one or two orders of magnitude. Significant examples are the access of point coordinates and arithmetic for low-dimensional geometric objects (see for example the case study [165]) and traversals of combinatorial structures. For example, vertices, edges and facets for polyhedra are anticipated to be small objects with simple member functions. The space and runtime overhead introduced through virtual member functions will not be negligible.

The *generic programming paradigm* features what is known in C++ as *class templates* and *function templates.* Templates are incompletely specified components in which a few types are left open and represented by formal placeholders, the *template arguments.* The compiler generates a separate translation of the component with actual types replacing the formal placeholders wherever this template is used. This process is called *template instantiation.* The actual types for a function template are implicitly given by the types of the function arguments at instantiation time. An example is a swap function that exchanges the value of two variables of arbitrary types. The actual types for a class template are explicitly provided by

[1] There are notable exceptions where the compiler can deduce for a virtual member function the actual member function that is called up, which allows the compiler to optimize this call. The keyword `final` has been introduced in Java to support this intention. However, so far these techniques are not realized in C++ compilers and they cannot succeed in all cases, even though it is arguable that typical uses in the domain of geometric algorithms can be optimized. However, distributing a software library in precompiled components will hinder their optimization which must be done at link time.

the programmer. An example is a generic list class for arbitrary item types. The definitions given below would enable us to use list<int>, with the actual type int given explicitly, for a list of integers and to swap two integer variables x and y with the expression swap(x,y), where the actual type int is given implicitly.

```
template <class T>  class list {
    // placeholder T represents the item type symbolically.
    void push_back( const T& t); // append t to list.
};
template <class T>  void swap( T& a, T& b) {
    T tmp = a; a = b; b = tmp;
}
```

The example of the swap function illustrates that a template usually requires certain properties of the template arguments, in this case that variables of type T are assignable. An actual type used in the template instantiation must comply with these assumptions in order to get a correct template instantiation. We can distinguish between *syntactic requirements* and *semantic requirements*. The syntactic requirement in our example would be the assignment operator for type T, and the semantic requirements would be that the operator should actually copy the value. If the syntactic requirements are not fulfilled, compilation simply fails. Semantic requirements are not checkable at compile time. However, it might be useful to connect a specific semantic requirement to an artificial, newly introduced syntactic requirement, e.g., a tag similar to iterator tags in [177]. This technique allows decisions at compile time based on the actual type of these tags.

The set of requirements that is needed to obtain a correct instantiation of a member function of a class template is usually only a fraction of all requirements for the template arguments of this class template. If only a subset of the member functions is used in an instantiation, it would be sufficient for the actual types to fulfill only the requirements needed for this subset of member functions. This is possible in C++ since as long as a C++ compiler is not explicitly forced, the compiler is not allowed to instantiate member functions that are not used; for this reason possible compilation errors due to missing functionality of the actual types cannot occur [38]. This enables us to design class templates with optional functionality. These class templates can be used if and only if the actual types used in the template instantiation fulfill the additional requirements. An example would be an optional normal vector for facets in polyhedral surfaces and a rendering algorithm that makes use of a precomputed normal vector for the lighting calculations. If the normal vector would not be present in the facets, another rendering algorithm would be needed that recomputes the normal vector for each facet.

A good and well known example illustrating generic programming is the Standard Template Library, STL [38, 145, 173]. A *concept* is a well defined set of requirements. Generality and flexibility have been achieved in STL with the careful selection and structuring of requirements to form powerful and abstract concepts. In our swap-function example, the appropriate concept is named 'assignable' and includes the requirement of an assignment operator [173]. If an actual type fulfills the requirements of a concept, it is a *model* for this concept. In our example, `int` is a model of the concept 'assignable'.

The advantages of the generic programming paradigm are strong type checking at compile time during the template instantiation, no need for extra storage nor additional indirections during function calls, and full support of inline member functions and code optimization at compile time [178]. One specific disadvantage of generic programming in C++ is the lack of a notation in C++ to declare the syntactical requirements for a template argument, i.e., an equivalent to the virtual base class in the object-oriented programming paradigm. The syntactical requirements are scattered throughout the implementation of the template. The concise collection of the requirements is left for the program documentation. In general, the flexibility is resolved at compile time which gives us the advantages mentioned above, but it can be seen as a disadvantage if runtime flexibility is needed. However, the generic data structures and algorithms can be parameterized with the base class used in the object-oriented programming to achieve runtime flexibility where needed.

2.2 Generic Algorithms Based on Iterators

Algorithmic abstraction is a key goal in generic programming [146, 147]. One aspect is to reduce the interface to the data types used in the algorithm to a set of simple and general concepts. One of them is the *iterator* concept in STL which is an abstraction of pointers. Iterators serve two purposes: They refer to an item and they traverse over the sequence of items that are stored in a data structure, also known as *container class* in STL. Five different categories are defined for iterators: input, output, forward, bidirectional and random-access iterators, according to the different possibilities of accessing items in a container class. The usual C-pointer referring to a C-array is a model for a random-access iterator.

Sequences of items are specified by a *range* `[first,beyond)` of two iterators. This notion of a half-open interval denotes the sequence of all iterators obtained by starting with the iterator `first` and advancing `first` until the iterator `beyond` is reached, but it does not include `beyond`.

A container class is supposed to provide a local type called `iterator`, which is a model of an iterator, and two member functions: `begin()` returns the start iterator of the sequence and `end()` returns the iterator referring to the 'past-the-end'-position of the sequence. The `list` class template example from the previous section can be extended as follows, though we leave the actual implementation of the iterator open.

```
template <class T>  class list {
    void push_back( const T& t); // append t to list.
    typedef ... iterator;
```

```
    iterator begin();
    iterator end();
};
```

Generic algorithms are not written for a particular container class in STL, they use iterators instead. For example, a generic `contains` function can be written to work for any model of an input iterator. It returns `true` iff the `value` is contained in the values of the range `[first,beyond)`.

```
template <class InputIterator, class T>
bool contains( InputIterator first, InputIterator beyond, const T& value){
    while ((first != beyond) && (*first != value))
        ++first;
    return (first != beyond);
}
```

This generic `contains` function can be used with C-pointers referring to a C-array. Recall that C-pointers are a model for a random access iterator, which is more general than an input iterator. The following example declares an array of a hundred integers and searches for a 42.

```
int a[100];
// ... initialize elements of a.
bool found = contains( a, a+100, 42);
```

This generic `contains` function can also be used with our list class template as illustrated in the following example:

```
list<int> ls;
// ... insert some elements into ls.
bool found = contains( ls.begin(), ls.end(), 42);
```

Similarly, a convex hull algorithm can read its input points from an iterator range and write the resulting vertices of the convex hull to an output iterator. It returns the value of the output iterator after writing all points to it. Hence, the number of vertices of the convex hull is encoded in the difference of the return value and the output iterator given as function argument.

```
template <class InputIterator, class OutputIterator>
OutputIterator
convex_hull( InputIterator first, InputIterator beyond,
             OutputIterator result);
```

Given an appropriate type `Point_2` for two-dimensional points, the convex hull algorithm can be used with C-arrays as already illustrated for the generic `contains` function.

```
Point_2 points[100];
// ... initialize the array of points.
Point_2 hull[100];
Point_2* result = convex_hull( points, points+100, hull);
size_t h = result - hull; // number of vertices of the hull
```

However, the convex hull algorithm cannot be used directly to write new points to an empty list of points. The iterator obtained with the `begin` member function does not automatically create new nodes in the list if necessary. STL provides a so-called *adaptor* for this kind of problems. A `back_inserter` function accepts a container class as argument and returns an object that is a model of an output iterator. Each time an element gets assigned to this iterator, it calls the `push_back` member function of the container class. The following example illustrates its use for lists and the convex hull algorithm. The expression `list<Point_2>::iterator` denotes the local `iterator` type in the scope of the list class template.

```
list<Point_2> points;
// ... insert some new points.
list<Point_2> hull;
convex_hull( points.begin(), points.end(), back_inserter(hull));
size_t h = hull.size(); // number of vertices of the hull
```

How does the convex hull algorithm know the point type and the basic operations on points to compute the hull? The point type is known as the value type of the input iterator and can be extracted from the iterator type with *iterator traits* (see Section 2.5). The basic operations on points are either part of the fixed requirements the algorithm imposes, or another traits class can provide the flexibility to adapt the algorithm to different basic operations (see Section 2.6).

At the end of this section we would like to mention a useful feature inherent in the design of iterators in STL. Input iterators can represent infinite sequences, for example a random number generator or random points uniformly distributed in a certain region. The iterator contains the current item in an internal state and each call to the `operator++` generates the next item of the sequence.

2.3 Circulators

The concept of iterators in STL is tailored for linear sequences. In contrast, circular sequences occur naturally in many combinatorial and geometric structures. Examples are polyhedral surfaces and planar maps. The edges emanating from a vertex or the edges around a facet form a circular sequence.

Since circular sequences cannot provide efficient iterators, we have introduced the new concept of *circulators*. They share most of the requirements with iterators, with the main difference being the lack of a past-the-end position in the sequence. Appropriate adaptors are provided between iterators and circulators to integrate circulators smoothly into the framework of Stl. We give a short introduction to circulators and discuss advantages and disadvantages thereafter. We rewrite the example of the generic `contains` function from above using circulators. As usual for circular structures, a `do-while` loop is preferable, so that for a specific input, here it is for `c == d`, all elements in the sequence are enumerated.

```
template <class Circulator, class T>
bool contains( Circulator c, Circulator d, const T& value) {
    if (c != NULL) {
        do {
            if (*c == value)
                return true;
        } while (++c != d);
    }
    return false;
}
```

Three circulator categories are defined: forward, bidirectional and random-access circulators. Given a circulator `c`, the operation `*c` denotes the item to which the circulator refers. The operation `++c` advances the circulator by one item and `--c` steps a bidirectional circulator one item backwards. For random-access circulators `c+n` advances the circulator `n` steps. Two circulators can be compared for equality.

Circulators have a different notion of reachability and ranges than iterators. A circulator `d` is called *reachable* from a circulator `c` if `c` can be made equal to `d` with finitely many applications of the operator `++`. Due to the circularity of the sequence this is always true if both circulators refer to items of the same sequence. In particular, `c` is always reachable from `c`. Given two circulators `c` and `d`, the range `[c,d)` denotes all circulators obtained by starting with `c` and advancing `c` until `d` is reached, but does not include `d`, for `d` $\neq$ `c`. So far it is the same range definition as for iterators. The difference lies in the use of `[c,c)` to denote all items in the circular sequence, whereas for an iterator `i` the range `[i,i)` denotes the empty range. This resembles the difference between a `while` loop and a `do-while` loop. As long as `c != d` the range `[c,d)` behaves like an iterator range and can be used in Stl algorithms. For circulators, however, an additional test `c == NULL` is required that returns true if and only if the circular sequence is empty.

Supporting both, iterators and circulators, within the same generic algorithm is equally simple as supporting iterators only. This and the requirements for circulators are described in the Cgal Reference Manual [34].

Besides the conceptual cleanness, the main reason for inventing a new concept with a similar intent as iterators is efficiency. An iterator is supposed to be a light-weight object – merely a pointer and a single indirection to advance the iterator. Although iterators can be written for circular sequences, we do not know of an efficient solution. The missing past-the-end situation in circular sequences can be solved with an arbitrary sentinel in the cyclic order, but this would destroy the natural symmetry in the structure (which is in itself a bad idea), and additional bookkeeping in the items and checking in the iterator advance method reduces efficiency. Another solution may use more bookkeeping in the iterator, e.g., storing a start reference, a current reference, and a kind of winding-number that is zero for the `begin()`-iterator and one for the past-the-end situation[2]. In consequence, we have introduced the concept of circulators that allows light-weight implementations. We provide adaptor classes between iterators and circulators (with the corresponding penalty in efficiency) in order to integrate this new concept into the framework of STL.

A serious design problem is the slight change of the semantic for circulator ranges as compared to iterator ranges. Since this semantic is defined by the intuitive operators `++` and `==` which we would like to keep for circulators as well, circulator ranges can be used in STL algorithms. This could be a useful feature, if there would not be the definition of a full range $[c, c)$ that an STL algorithm will treat as an empty range. However, the likelihood of a mistake may be overestimated, since for a container `C` supporting circulators there is no `end()` member function, and an expression such as `sort( C.begin(), C.end())` will fail. It is easy to distinguish iterators and circulators at compile time, which allows for generic algorithms supporting both as arguments. It is also possible to protect algorithms against inappropriate arguments using the same technique, but an extension of STL algorithms is beyond our scope.

2.4 Function Objects

The report [177] on STL contains as its third example the following fully fledged C++ program, which copies all integer values from the input stream to the output stream that are not divisible by the first command-line argument.

```
main(int argc, char** argv) {
    if ( argc != 2) throw( "usage: remove_if_divides integer\n");
    remove_copy_if( istream_iterator<int>(cin), istream_iterator<int>(),
                    ostream_iterator<int>(cout, "\n"),
                    not1( bind2nd( modulus<int>(), atoi( argv[1]))));
    return 0;
}
```

This program makes use of the stream iterator adaptors `istream_iterator` and `ostream_iterator` to read and to write the integer values. But the interesting part here is the use of `modulus`, `bind2nd` and `not1`. The `modulus` function object computes the remainder of an integer division. It is a binary operation. The `bind2nd`

[2]This is currently implemented in CGAL as an adaptor class which provides a pair of iterators for a given circulator.

function takes a binary operation *op* and an argument value *b*. It returns itself a function object that is an unary operation which, when called with an argument *a*, returns the value of *op(a,b)*. The `not1` function returns a function object negating the return value of its argument which must be an unary function object. Function objects are first class citizens in the design of STL. For example, an object such as `binder2nd`, which is returned by `bind2nd`, and its counterpart `binder1st` realize concepts like *currying* known from functional programming languages [22].

A function object basically is an instance of a class with the `operator()` member function implemented, such that a call to this member function of the object looks like a function call[3]. Function objects are well suited as parameters for generic functions. A typical example would be the exchange of the equality comparison with a function object, which is currently hard coded as the `operator==` in the generic `contains` function from above. First, we define a function object `equals` that performs the same comparison.

```
template <class T> struct equals {
    bool operator()( const T& a, const T& b) { return a == b; }
};
```

We modify the iterator-based generic `contains` function from above. It needs an additional template parameter `Eq` and takes an additional function parameter `eq` for a bidirectional function object which is used for the comparison.

```
template <class InputIterator, class T, class Eq>
bool contains( InputIterator first, InputIterator beyond, const T& value,
               Eq eq ) {
    while ((first != beyond) && ( ! eq( first, value)))
        ++first;
    return (first != beyond);
}
```

The example using C-arrays with the `contains` function needs now an additional argument – the function object. The expression `equals<int>()` calls the default constructor for the template class `equals<int>` from above which is a function object comparing two integers for equality.

```
int a[100];
// ... initialize elements of a.
bool found = contains( a, a+100, 42, equals<int>());
```

The next section illustrates how the additional parameter of the contains function can be automatically selected if the value type of the iterator is known. C++ allows

[3] Furthermore, the argument types and the return type are declared as local types in the function object class, though we skip them here for the clarity of the presentation.

to use also simple function pointers as function objects. The advantage of objects is that they can have an internal state. We continue our example of the `contains` function and define a comparison object that is true when the absolute value of the difference of its two arguments is smaller than ε. The ε value is stored in the function object itself. At construction time of the function object the actual value for ε is initialized, in our example to one, so that the contains function will also return true if the values 41 or 43 do occur in the range.

```
template <class T> struct eps_equals {
    T epsilon;
    eps_equals( const T& eps) : epsilon(eps) {}
    bool operator()( const T& a, const T& b) {
        return (a-b < epsilon) && (b-a < epsilon);
    }
};
bool found = contains( a, a+100, 42, eps_equals<int>(1));
```

How about a function object that counts the number of comparisons needed as a side-effect? Here it is:

```
template <class T> struct count_equals {
    size_t& count;
    count_equals( size_t& c) : count(c) {}
    bool operator()( const T& a, const T& b) {
        ++count;
        return a == b;
    }
};
size_t counter = 0;
bool found = contains( a, a+100, 42, count_equals<int>(counter));
// counter contains number of comparisons needed.
```

2.5 Iterator Traits Class

Iterators refer to items of a particular value type. Algorithms parameterized with iterators might need the value type directly. Assuming that iterators are implemented as classes the value type can be defined as a local type of the iterator, as in the following example of an iterator referring to integer values. The value type can be referred to with the expression `iterator_over_ints::value_type`.

```
struct iterator_over_ints {
    typedef  int  value_type;
    // ...
};
```

Since a C-pointer is a valid iterator, this approach is not sufficient. The solution chosen for STL is the iterator traits class [148], which is a class template parameterized with an iterator:

```
template <class Iterator> struct iterator_traits {
    typedef  typename Iterator::value_type  value_type;
    // ...
};
```

The value type of the iterator example class above can now be expressed as `iterator_traits< iterator_over_ints >::value_type`. For C-pointers a specialized version of the iterator traits class exists.

```
template <class T> struct iterator_traits<T*> {
    typedef  T  value_type;
    // ...
};
```

Now the value type of a C-pointer, e.g., to `int`, can be expressed as `iterator_traits< int* >::value_type`. This technique of providing an additional, more specific definition for a class template is known as *partial specialization*. The iterator traits class contains also definitions about the `difference_type`, the `iterator_category`, the `pointer` type and the `reference` type of the iterator. The example of the generic `contains` function with the function object from above can be made more convenient for the default use with a default initializer as follows:

```
template <class InputIterator, class T, class Eq>
bool contains( InputIterator first, InputIterator beyond, const T& value,
   Eq eq = equals<typename iterator_traits<InputIterator>::value_type>());
```

STL makes use of traits classes in other places as well, for example `char_traits` to define the equality test and other operations for a character type. In addition, this character traits class is used as a template parameter for the `basic_string` class template, which allows the adaption of the string class to different character sets.

2.6 Parameterizing Geometric Algorithms with Traits Classes

Geometric algorithms are commonly separated into layers: The algorithm itself, a geometric kernel with geometric objects and primitive operations, and the number type used to represent the coordinates of the geometric objects (as shown in the left diagram in Figure 2.1). These layers coincide with the flexibility we want to achieve; number types and geometric kernels should be exchangeable.

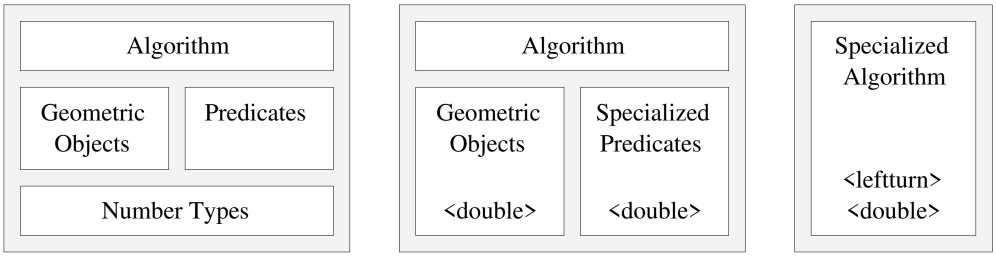


Figure 2.1: Different layers in geometric algorithms and specialization of predicates and algorithms (from left to right): Three layers, the algorithm, geometric objects with predicates, and number types. Two layers, the algorithm with geometric objects and predicates specialized on the built-in number type double. The algorithm itself specialized for the built-in number type double and a specific implementation of the predicates.

An algorithm is defined in terms of the geometric objects and operations on these objects as provided by the geometric kernel. Typical operations are decision predicates, such as lexicographic orders or sidedness tests. Other common operations are basic constructions, for example the midpoint of two points, geometric transformations, intersections, or the application of other algorithms.

The geometric kernel is based on a number type which is used to represent the coordinates of the geometric objects and to perform the calculations in the kernel operations. We identify here the number type with its arithmetic. Another arithmetic for a number type can be realized in C++ by encapsulating the number type in a new class, a wrapper, which will use the other arithmetic.

A fourth layer can be introduced between the geometric kernel and the number type which factors out different coordinate representations. For example in CGAL, we distinguish between a Cartesian and a homogeneous representation. The three-level hierarchy collapses to two levels if the geometric kernel is specialized for a particular number type, for example the built-in floating-point number type `double` or `float`. Specializations of the common orientation and in-circle predicates in particular are available [75, 172]. The hierarchy collapses to a single level if the geometric algorithm is specialized for a particular kernel. The algorithm appears as a monolithic block, no flexibility remains. The collapsed hierarchies are also illustrated in Figure 2.1.

Template specialization allows in C++ to provide an additional implementation for a class or a function template if a template parameter is fixed to a specific type. These template specializations allow the implementation of specialized kernels or algorithms with the same interface as the generic ones.

Each layer is a candidate for flexibility and might be a template parameter for the layer above. The use of a number type as a template parameter for a geometric kernel is obvious and needs no explanation. A geometric kernel is provided as a template parameter to a geometric algorithm or a geometric data structure with a traits class, similar to the parameterization of the `basic_string` class with `char_traits` in STL. We attach information to our own algorithms and data structures, but not to built-in types. Therefore, we can skip the indirection of partial specialization as for the `iterator_traits` or `char_traits` in STL. We define the geometric types as local types of our geometric traits class and the operations as local function objects with member access functions in the traits class. The traits class can be passed as a single template parameter to the algorithm or data structure. The single parameter makes it easy to apply already prepared implementations of traits classes. This technique is also known as the *'nested typedefs for name commonality'-idiom* [18, 117].

The following example of a convex hull algorithm illustrates the use of a geometric traits class. The algorithm used is Andrew's variant of Graham's scan [9, 55]. The actual implementation presented here stems from our framework for one sided error predicates [110]. The implementation has been modified to use the iterator based interface from above and the traits class.

Andrew's variant of Graham's scan needs only a point type and a `leftturn` predicate from the geometric kernel given that the input points are already sorted lexicographically. Thus, the geometric traits class is quite short in this example. The `leftturn` predicate for three points p, q, and r in the plane is true if the points in this order perform a left turn. For points represented in Cartesian coordinates, the predicate is equivalent to the sign of the following determinant (see also Section 6.9):

$$\texttt{leftturn}\ (p, q, r) \quad :\Longleftrightarrow \quad \begin{vmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{vmatrix} > 0\ .$$

In the following example, the geometric traits class for the convex hull algorithm is itself a class template parameterized with the number type `NT` for the point coordinates. `NT` is the same type as used for the Cartesian point type `Point_2`. The evaluation of the determinant is implemented as a function object assuming exact arithmetic and the `leftturn` member function gives access to the function object.

```
template <class NT> struct Point_2 {
    typedef NT Number_type;
    NT x;
    NT y;
};
template <class NT> struct Convex_hull_traits {
    typedef Point_2<NT> Point;
    struct Leftturn {
        bool operator()( const Point& p, const Point& q, const Point& r) {
            return (q.x-p.x) * (r.y-p.y) > (r.x-p.x) * (q.y-p.y);
        }
    };
    Leftturn leftturn() const { return Leftturn(); }
};
```

The algorithm follows the iterator based parameterization from Section 2.2. It requires that the sequence of input points from the range `[first,beyond)` of bidirectional iterators is lexicographically sorted and contains only pairwise disjoint points and at least two points. The algorithm computes the convex hull and copies all points on the boundary of the convex hull (not only the vertices) in counterclockwise order to the output iterator `result`. It runs in linear time and space and can produce up to $2n-2$ output points in the degenerate case of all points on a segment where n is the number of input points. The local vector can be omitted if the algorithm can use the output container as a stack. This is beyond the capabilities of the currently defined iterator categories and restricts the applicability of the algorithm. Furthermore this change would distract here from the purpose of this example, the illustration of the use of a geometric traits class in a geometric algorithm.

```
template <class BidirectionalIterator, class OutputIterator, class Traits>
OutputIterator
convex_hull( BidirectionalIterator first, BidirectionalIterator beyond,
             OutputIterator result, const Traits& traits) {
    typedef typename Traits::Point Point;
    vector<Point> hull;
    hull.push_back( *first); // sentinel
    hull.push_back( *first);

    // lower convex hull (left to right)
    BidirectionalIterator i = first;
    for ( ++i; i != beyond; ++i) {
        while ( traits.leftturn()( hull.end()[-2], *i, hull.back()))
            hull.pop_back();
        hull.push_back( *i);
    }
    // upper convex hull (right to left)
    i = beyond;
    for ( --i; i != first; ) {
        --i;
        while ( traits.leftturn()( hull.end()[-2], *i, hull.back()))
            hull.pop_back();
        hull.push_back( *i);
    }
    // clean up and copy hull to output iterator
    hull.pop_back();
    hull.front() = hull.back();
    hull.pop_back();
    return copy( hull.begin(), hull.end(), result);
}
```

Calling the algorithm with the traits class is straightforward. The default constructor of the traits class is used. Similar to the `contains` function at the end of Section 2.5 we can use `iterator_traits` to deduce the point type from the iterator type and from the point type the number type which can be used to provide our geometric traits class `Convex_hull_traits` as default argument to the convex hull algorithm. We use an overloaded definition of the convex hull algorithm with three parameters to do so.

```
template <class BidirectionalIterator, class OutputIterator>
OutputIterator
convex_hull( BidirectionalIterator first, BidirectionalIterator beyond,
             OutputIterator result) {
    typedef typename iterator_traits<BidirectionalIterator>::value_type P;
    typedef typename P::Number_type Number_type;
    typedef Convex_hull_traits<Number_type> Traits;
    return convex_hull( first, beyond, result, Traits());
}
```

One benefit of using function objects in the traits class instead of plain member functions is the possible association of a state with the function object. We extend this to a traits class with a state.

In our framework on one sided error predicates [110] we introduced the notion of a *conservative* implementation of a predicate. If a conservative implementation of the `leftturn` predicate returns true, the three points perform a left turn, but if it returns false, we do not know the orientation of the points. Thus, the decision errors due to rounding errors in inexact arithmetic are limited to one side of the two possible answers. Useful applications for such predicates will assume that these false answers occur rarely, but in principle an implementation saying always false is a legal implementation. Examples for convex hull algorithms and a triangulation of point sets are given in the paper that compute a well defined output if the predicate is not exact but a conservative implementation. Andrew's variant of Graham's scan as presented above computes a sequence of points of which the points on the convex hull are a subsequence if it is used with a conservative implementation of the predicate. The output can be easily postprocessed with the same algorithm but with an exact implementation of the predicate. For more properties of the computed output and other examples see [110].

For floating point arithmetic an error bound can be computed such that if the expression computing the determinant is larger than the error bound, the exact value of the determinant is greater than zero. The expression to compute the determinant and its error bound give us the conservative implementation

$$(\mathtt{q.x} - \mathtt{p.x}) * (\mathtt{r.y} - \mathtt{p.y}) - (\mathtt{q.y} - \mathtt{p.y}) * (\mathtt{r.x} - \mathtt{p.x}) > 8(3u + 6u^2 + 4u^3 + u^4)B^2 \ ,$$

where u is the unit roundoff of the floating-point number system and B is the absolute value of the maximal coordinate value of the points [172, 110]. We have $u = 2^{-53}$ for IEEE double precision, and $u = 2^{-24}$ for IEEE single precision floating-point numbers. In the following traits class we assume a built-in type `double` following the IEEE standard and we make the value B a state value of the traits class. In order to make the error bound representable as `double`, we round it to $(32^{-50} + 2^{-100})B^2$ and require B to be a power of two.

```
class Convex_hull_traits_2 {
    double B;
public:
    typedef Point_2<double> Point;
    Convex_hull_traits_2( double b) : B(b) {}
    struct Leftturn {
        double B;
        Leftturn( double b) : B(b) {}
        bool operator()( const Point& p, const Point& q, const Point& r) {
            const double C = 1.0 / 1024.0 / 1048576.0 / 1048576.0; //2^-50
            return   (q.x-p.x) * (r.y-p.y) - (r.x-p.x) * (q.y-p.y)
                   > (3.0 * C + C * C) * B * B;
        }
    };
    Leftturn leftturn() const { return Leftturn(B); }
};
```

Just to show a specialization of a class template, the following definition replaces the generic traits, which assumes exact arithmetic, with our specialized conservative predicate traits for the number type double. In this implementation, B is arbitrarily set to one.

```
template <>
struct Convex_hull_traits<double> : public Convex_hull_traits_2 {
    Convex_hull_traits() : Convex_hull_traits_2(1.0) {}
};
```

Since B is probably a constant parameter, it might be more appropriate to make it a template parameter in `Convex_hull_traits_2`. However, another example of a useful traits class with a state is the computation of the two-dimensional convex hull for a set of three-dimensional points projected onto a two-dimensional plane. Instead of projecting the points and computing the convex hull on the projected points, the convex hull can be computed with the original three-dimensional points and a modified `leftturn` predicate that takes into account the projection stored as a local state.

Another example of the flexibility of the geometric traits classes is the reconstruction of a terrain from a set of three-dimensional sample points. A common approach is to triangulate the sample points using a Delaunay triangulation in the xy-projection, just ignoring the elevation in the z-coordinate. Similar to the convex hull algorithm, a geometric class can be used to parameterize the two-dimensional triangulation algorithm to work on the three-dimensional data set.

2.7 Conclusion

We followed mainly the generic programming paradigm to achieve flexibility and efficiency in our implementations and in Cgal. The compliance with Stl is important in order to re-use its generic algorithms and container classes, and to unify the look-and-feel of the design with the C++ standard. As a consequence the design is easy to learn and easy to use for those who are familiar with Stl. In addition, synergisms are to be expected with other libraries following this design principles. Examples for the growing number of such libraries besides Cgal and Leda are Blitz++[4], a numerical library for scientific computing providing matrices and vector arithmetic [184, 183, 185], and Vigra[5], a computer vision library providing two-dimensional image processing [116].

The geometric traits classes in particular are an approach to separate the main control structure of an algorithm from the types and primitive operations it uses, just in the spirit of generic programming and its focus on algorithmic abstraction. We write an algorithm or data structure once and provide different geometric traits classes to apply it in different contexts.

The flexibility is also a key tool to achieve robust and efficient implementations. It allows the easy combination of number types, coordinate representation, and appropriate predicates with algorithms and data structure to compose a tailor made solution for the specific problem at hand (see [165] for a comparison of different number types, geometric kernels and convex hull algorithms in Cgal).

In a few circumstances we made also use of the object-oriented programming paradigm, for example the protected access to the internal representation of a polyhedral surface (see Section 4.8). This protected access is no time critical operation compared to the work that is supposed to be performed with the accessed internal representation.

[4] `http://oonumerics.org/blitz/`

[5] `http://www.egd.igd.fhg.de/~ulli/vigra/`

Chapter 3

CGAL, the Computational Geometry Algorithms Library

The birth of the CGAL-library dates back to a meeting in Utrecht in January 1995. Shortly afterwards, the five authors of [71] started developing the kernel. The CGAL-project has been funded officially since October 1996 and the team of developers has grown considerably among professionals in the field of computational geometry and related areas, in academia especially research assistants, PhD students and postdocs. The CGAL release 1.2 of January 1999 consists of approximately 110,000 lines of C++ source code for the library, plus 50,000 lines for accompanying sources, such as the test suite and example programs, not counting C++ comments or empty lines. In terms of the elder Constructive Cost Model (COCOMO) the line counts, the people involved, and the time schedule indicate a large project comparable to operating systems or database management systems [72]. The WWW home-page[1] of CGAL provides a list of publications about CGAL and related research: previous overviews [156, 186], the first design of the geometric kernel [70], recent overviews and descriptions of the current design [71, 33].

In addition to having influenced the overall design of the library, my particular contributions are the circulator design, the parameterization with traits classes in cooperation with Andreas Fabri, the polyhedral surface data structure and the tools used in writing the manuals. However, none of these contributions would have been what they are without the feedback of my colleagues in CGAL. The traits classes in particular are still subject to changes and enhancements and the latest design change for the polyhedral surfaces was influenced by suggestions of Hervé Brönnimann and Michael Hoffmann to redesign the current geometric kernel of CGAL.

In this chapter related work on geometric software and precursors of CGAL is reviewed. An overview of the library structure is followed by a more detailed description of the design of the geometric kernel and the basic library. The examples given here refer to the CGAL release 1.2 of January 1999.

[1] `http://www.cs.uu.nl/CGAL/`

3.1 Related Work

Three approaches of disseminating geometric software can be distinguished: collections of isolated implementations, integrated applications or workbenches, and software libraries. An overview on the state of the art of computational geometry software before CGAL including many references is given in [8].

The approach of collecting isolated implementations, also called the *Gems* approach according to the successful Graphics Gems series [82, 11, 112, 94], usually requires some adaption effort to make things work together. Compared to the graphics gems, computational geometry implementations usually use more involved data structures and more advanced algorithms. This makes adaption harder. A good collection provides the *Directory of Computational Geometry Software*[2].

Integrated applications and workbenches provide a homogeneous environment, for example with animation and interaction capabilities, and all parts work smoothly together. However, they tend to be monolithic, hard to extend, and hard to reuse in other projects. Examples date back to the end of the Eighties [69, 56], specifically XYZ GeoBench[3] developed at ETH Zurich, Switzerland, is one of the precursors of CGAL [151, 168].

Software libraries promise that the components work seamlessly together, that the library is extensible and that the components can be reused in other projects. Examples are the precursors of CGAL developed by members of the CGAL consortium. These precursors are PLAGEO [81] developed at Utrecht University, C++GAL [14] developed at INRIA Sophia-Antipolis, and the geometric part of LEDA[4] [132, 133, 131], a library for combinatorial and geometric computing, which has been developed at Max-Planck-Institut für Informatik, Saarbrücken. Another example is GEOMLIB [16], a computational geometry library implemented in Java at the Center for Geometric Computing, located at Brown University, Duke University, and John Hopkins University in the United States. They state their goal as *an effective technology transfer from Computational Geometry to relevant applied fields*.

3.2 Overview of the Library Structure

CGAL is structured into three layers and a support library, which stands apart. The three layers are the core library with basic non-geometric functionality, the geometric kernel with basic geometric objects and operations, and the basic library with algorithms and data structures.

[2] `http://www.geom.umn.edu/software/cglist/`
[3] `http://wwwjn.inf.ethz.ch/geobench/XYZGeoBench.html`
[4] `http://www.mpi-sb.mpg.de/LEDA/`

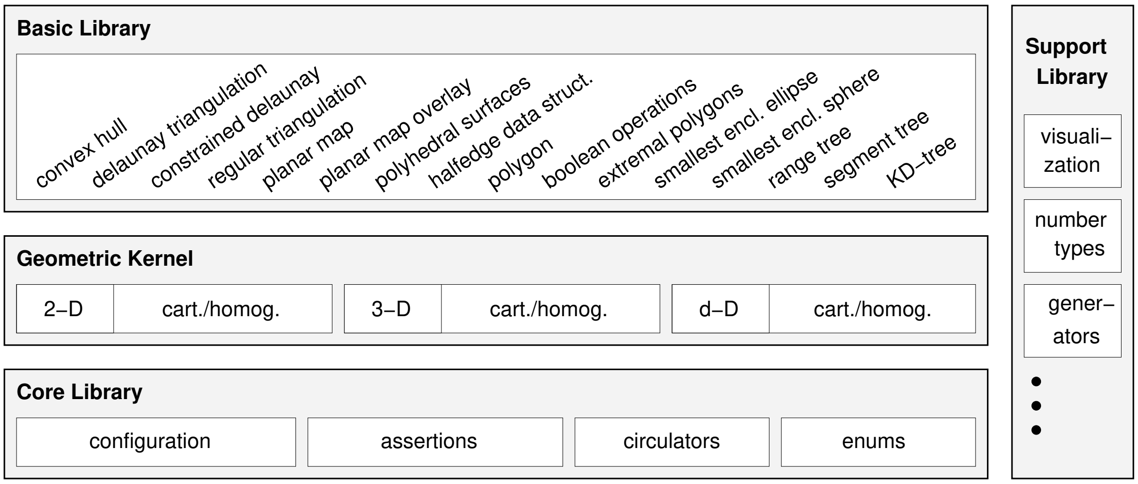


Figure 3.1: The structure of CGAL.

The library layers and the support library are further subdivided into smaller modular units (see Figure 3.1). The modular approach has several benefits: The library is easier to learn, the implementation work is more easily spread among the project partners, and the reduction of dependencies facilitates testing and maintenance [120].

The geometric kernel contains simple geometric objects of constant size such as points, lines, segments, triangles, tetrahedra, circle and more. It provides geometric predicates on those objects, operations to compute intersections of and distances between objects, and affine transformations. The kernel objects are closed under affine transformations, e.g., the existence of circles implies that there are also ellipses in the kernel.

The geometric kernel is split in three parts, one for two-dimensional objects, one for three-dimensional objects, and one for general-dimensional objects. Geometry in two and three dimensions is well studied and has lots of applications which explains their special status. For all dimensions there are Cartesian and homogeneous representations available for the coordinates.

To solve robustness problems, CGAL advocates the use of exact arithmetic instead of floating point arithmetic. An arithmetic is associated with a number type in CGAL and the classes in the geometric kernel are parameterized by number types. CGAL provides own number types [32, 31] and supports number types from other sources, e.g., from LEDA or the Gnu Multiple Precision library [87]. Since the arithmetic operations needed in CGAL are quite basic, every library supplying number types can be adapted easily to work with CGAL.

The basic library contains more complex geometric objects and data structures: polygons, triangulations, planar maps, polyhedra and so on. It also contains algorithms, such as for computing the convex hull of a set of points, the union of two polygons, smallest enclosing ellipse and so on. Figure 3.1 indicates the major parts in the basic library. These parts are mostly independent from each other and even independent from the kernel. This independence has been achieved with geometric traits classes as introduced in Section 2.6. Default implementations of the traits classes use the CGAL kernel for the types and primitive operations. Other implementations of the traits classes provided in CGAL use the LEDA geometric part. The traits class requirements are simple enough for a user to be able to write a traits class for own geometric data types and operations.

The core library offers basic non-geometric functionality that is needed in the geometric kernel or the basic library, for example support for coping with different C++ compilers which all have their own limitations. The core library contains the support for assertions, preconditions and postconditions. Circulators and random number generators belong here as well.

The support library also contains functionality with non-geometric aspects. In contrast to the core library, this functionality is not needed by the geometric kernel nor the basic library. The support library interfaces the geometric objects with external representations, like visualizations or external file formats. Among the list of supported formats are VRML and PostScript as well as the GeomView program and LEDA windows for 2D and 3D visualization. The support library also contains generators for synthetic test data sets, for example random points uniformly distributed in a certain domain. The adaptation of number types from other libraries is contained in the support library as well. The separation from the kernel and the basic library makes the functionality in the support library orthogonal and therefore open for future extensions.

3.3 Geometric Kernel

The geometric kernel contains types for objects of constant size, such as point, vector, direction, line, ray, segment, triangle, iso-oriented rectangle and tetrahedron. Each type provides a set of member functions, for example access to the defining objects, the bounding box of the object if existing, and affine transformation. Global functions are available for the detection and computation of intersections as well as for distance computations.

The current geometric kernel provides two families of geometric objects: One family is based on the representation of points using Cartesian coordinates. The other family is based on the representation of points using homogeneous coordinates. The homogeneous representation extends the representation with Cartesian coordinates by an additional coordinate which is used as a common denominator. More formally, in d dimensional space, a point with homogeneous coordinates $(x_0, x_1, \ldots, x_{d-1}, x_d)$, where $x_d \neq 0$, has Cartesian coordinates $(x_0/x_d, x_1/x_d, \ldots, x_{d-1}/x_d)$. This avoids divisions and reduces many computations in geometric algorithms to calculations over the integers. The homogeneous representation is used for affine geometry in CGAL, and not projective geometry, where the homogeneous representation is usually known from. Both families are parameterized by the number type used to represent the Cartesian or homogeneous coordinates. The type CGAL_Cartesian<double> specifies the Cartesian representation with coordinates of type double, and the type CGAL_Homogeneous<int> specifies the homogeneous representation with coordinates of type int. These representation types are used as template argument in all geometric kernel types, like a two-dimensional point declared as

```
template <class R> CGAL_Point_2;
```

with a template parameter R for the representation class. Typedefs can be used to introduce conveniently short names for the types. Here is an example given for the point type with the homogeneous representation and coordinates of type int:

```
typedef CGAL_Point_2< CGAL_Homogeneous<int> >   Point_2;
```

The class templates parameterized with CGAL_Cartesian or CGAL_Homogeneous provide the user with a common interface to the underlying representation. This common interface can be used in higher-level implementations independently of the actual coordinate representation. The list of requirements on the template parameter defines the concept of a *representation class* for the geometric kernel. The details for realizing this parameterization can be found in [70, 71].

CGAL provides clean mathematical concepts to the user without sacrificing efficiency. For example, CGAL strictly distinguishes points and (mathematical) vectors, i.e., it distinguishes affine geometry from the underlying linear algebra. Points and vectors are not the same as is discussed in [84] with regard to illicit computations resulting from identification of points and vectors in geometric computations. In particular, points and vectors behave differently under affine transformations [188]. We do not even provide automatic conversion between points and vectors but use the geometric concept of an origin instead. The symbolic constant CGAL_ORIGIN represents a point and can be used to compute the locus vector as the difference between a point and the origin. Function overloading is used to implement this operation internally as a simple conversion without any overhead. Note that we do not provide the geometrically invalid addition of two points, since this might lead to ambiguous expressions: Assuming three points p, q, and r and an affine transformation A, one can write in CGAL the perfectly legal expression $A(p+(q-r))$. The slightly different expression $A((p+q)-r)$ contains the illegal addition of two points. However, if we allow this addition, we would expect the same result coordinatewise as in the previous, legal expression. But this is not necessarily intended, since the expression within the affine transformation is meant to evaluate to a vector, and not to a point as in the previous expression. Vectors and points behave differently under affine transformations. To avoid these ambiguities, the automatic conversion between points and vectors is not provided as well.

Class hierarchies are used rarely in CGAL. One example are affine transformations, which maintain distinct internal representations specialized on restricted transformations. The internal representations differ considerably in their space requirements and the efficiency of their member functions. For all but the most general representation we gain performance in terms of space and time. And for the most general representation, the performance penalty caused by the virtual functions is negligible, because the member functions are computationally expensive for this general representation. Alternatively we could have used this general representation for affine transformations only. But the use of a hierarchy is justified, since the specialized representations, namely translation, rotation and scaling, arise frequently in geometric computing.

Another design decision was to make the (constant-size) geometric objects in the kernel non-modifiable. For example, there are no member functions to set the Cartesian coordinates of a point. Points are viewed as atomic units (see also [57]), and no assumption is made on how these objects are represented. In particular, there is no assumption that points are represented with Cartesian coordinates. They might use polar coordinates or homogeneous coordinates instead. Then, member functions to set the Cartesian coordinates are expensive. Nevertheless, in current CGAL the types based on the Cartesian representation as well as the types based on the homogeneous representation have both member functions returning Cartesian coordinates and member functions returning homogeneous coordinates. These access functions are provided to make implementing own predicates and operations more convenient.

Like other libraries [133, 29, 105] we use reference counting for the kernel objects. Objects point to a shared representation. Each representation counts the number of objects pointing to it. Copying objects increments the counter of the shared representation, deleting an object decrements the counter of its representation. If the counter reaches zero by the decrement, the representation itself is deleted (see [135, Item 29] for further information). The implementation of reference counting is simplified by the non-modifiability of the objects. However, the use of reference counting was not the reason for choosing non-modifiability. Using 'copy on write' (a new representation is created for an object whenever its value is changed by a modifying operation), reference counting with modifiable objects is possible and only slightly more involved. A comparison with a prototype of a geometric kernel without reference counting can be found in [165]. The test applications are two-dimensional convex hull algorithms. Reference counting costs about 15% to 30% runtime for the types `double` and `float`, but gains 2% to 11% runtime for the type `leda_real`.

Further details of the geometric kernel can be found in [71], for example the polymorphic behavior of the return type of the intersection functions.

3.4 Basic Library

The basic library contains more complex geometric objects and data structures, such as polygons, polyhedrons, triangulations (including Delaunay triangulations), planar maps, range trees, segment trees, and kd-trees. It also contains geometric algorithms, such as convex hull, smallest enclosing circle, ellipse, and sphere, boolean operations on polygons and map overlay.

Following the generic programming paradigm as introduced above, Cgal is made to comply with Stl. The interfaces of geometric objects and data structures in the basic library make extensive use of iterators, circulators and handles, so that algorithms and data structures can be easily combined with each other and with those provided by Stl and other libraries.

An example of a geometric algorithmic problem is the computation of the convex hull. The algorithm takes a set of points and outputs the sequence of extreme points on the boundary of the convex hull. The following program computes the convex hull of 100 random points uniformly distributed in the disc of radius one centered at the origin. The point generator gets as parameter a random source. This random source is initialized with the fixed seed 1 in this example. The result is drawn in a Leda window, first all points in black, then the hull as polygon in green and finally the vertices in red. The header file hides the usual typedefs and declares all types parameterized with the representation class `CGAL_Cartesian<double>`.

```
#include "cartesian_double.h"

int main () {
    Random                  rnd(1);
    Random_points_in_disc_2  rnd_pts( 1.0, rnd);
    list<Point_2>           pts;
    copy_n( rnd_pts, 100, back_inserter( pts));

    Polygon_2  ch;
    CGAL_convex_hull_points_2( pts.begin(), pts.end(), back_inserter(ch));

    Window* window = demo_window();
    Window_iterator_point_2  wout( *window);
    copy( pts.begin(), pts.end(), wout);
    *window << CGAL_GREEN << ch << CGAL_RED;
    copy( ch.vertices_begin(), ch.vertices_end(), wout);

    Point_2  p;
    *window >> p;     // wait for mouse click
    delete window;
    return 0;
}
```

This program also illustrates the use of the Cgal polygon as a container class. The `back_inserter` adaptor of Stl is applicable as expected, which illustrates the generic tool-box character of Stl and its concepts.

Triangulations are another example of a container-like data structure in the basic library. Triangulations in CGAL support the incremental construction. The following program is therefore even simpler than the previous one; without using any intermediate container to store all input points, 100 random points are copied into the triangulation data structure.

```
#include "cartesian_double.h"

int main () {
    Random                    rnd(1);
    Random_points_in_disc_2   rnd_pts( 1.0, rnd);

    Delaunay_triangulation_2  dt;
    copy_n( rnd_pts, 100, back_inserter( dt));

    Window* window = demo_window();
    *window << dt;
    Point_2  p;
    *window >> p;     // wait for mouse click
    delete window;
    return 0;
}
```

The major technological achievement in the design of the basic library was the concept of the geometric traits class, see Section 2.6 for the introduction, which allows the reuse of the triangulation data structure, for example to triangulate a set of three dimensional points with respect to their xy-projection (useful to reconstruct terrains). We assume in the following example that the representation class for the geometric kernel is named `REP` in the header file. The program reads three-dimensional points from `cin`, triangulates them, and writes the triangulation to `cout`. Note that most of these typedefs are equal to those hidden previously in the header file. The only change is the geometric traits class from `CGAL_Triangulation_euclidean_traits_2` to the one given here.

```
#include "cartesian_double.h"
#include <CGAL/Triangulation_euclidean_traits_xy_3.h>

typedef CGAL_Triangulation_euclidean_traits_xy_3<REP>             Traits;
typedef CGAL_Triangulation_vertex_base_2<Traits>                  Vb;
typedef CGAL_Triangulation_face_base_2<Traits>                    Fb;
typedef CGAL_Triangulation_default_data_structure_2<Traits,Vb,Fb> Tds;
typedef CGAL_Delaunay_triangulation_2<Traits,Tds>       Triangulation_xy;

int main () {
    Triangulation_xy  dt;
    copy( istream_iterator<Point_3>(cin), istream_iterator<Point_3>(),
          back_inserter( dt));
```

```
    cout << dt;
    return 0;
}
```

The triangle-based data structure and the halfedge data structure used for the planar map and the polyhedral surface in the basic library are based on the design of combinatorial data structures described in [107]. The revised design of the halfedge data structure for polyhedral surfaces is described in Chapter 4 and [108].

3.5 Conclusion

The generic programming paradigm and the concepts of STL worked well for geometric algorithms and data structures. CGAL is a highly modular, flexible, open and efficient library that interacts smoothly with other libraries following the same concepts or by adaption with the geometric traits classes, e.g., STL, LEDA, and others.

Generic programming with STL and geometric algorithms from CGAL are easy to teach and easy to use. However, implementing algorithms and data structures in this framework requires knowledge of the latest template issues in the C++ standard document. Default arguments allow us to hide most of the advanced techniques from the user. For example, iterator traits or geometric traits need not be known by the user in order to use CGAL effectively. A temporary situation is also the lack of compilers that comply to the standard and long compilation times.

Currently, each algorithm and each data structure provides its own default traits class. The next step would be to unify these traits classes to have a common default traits class. This default traits class would represent a whole geometric kernel. The current representation classes of the geometric kernel would work if we add the operations and basic constructions needed by the geometric traits classes. However, we need to redesign the geometric kernel in order to do so.

Chapter 4

Designing a Data Structure for Polyhedral Surfaces

Our work on hidden-surface removal motivated our interest in a proper definition and representation of polyhedral surfaces. A halfedge data structure was developed as underlying representation. The design evolved [107] and was released in CGAL 1.0. Since CGAL 1.2 the planar map is based on the same design and reuses the halfedge data structure [73]. The triangulation and the three-dimensional tetrahedrization are also based on a similar design. They use a triangle-based data structure and a tetrahedron-based data structure respectively [180]. A first application in CGAL uses the polyhedral surface data structure in the computation of swept volumes [160]. The design of the halfedge data structure evolved further [108], influenced by suggestions of Hervé Brönnimann and Michael Hoffmann to redesign the current geometric kernel of CGAL.

We give a combinatorial and rather strict definition of polyhedral surfaces that suits our needs in the hidden-surface removal application well. We evaluate edge-based data structures with respect to a compact and efficient representation for the polyhedral surfaces. A review of related work providing polyhedral surfaces in the context of software libraries follows. We refine our general design goals for polyhedral surface and present the design, including several examples and a detailed discussion of its realization in C++. The examples given in this chapter refer to an internal release in CGAL.

4.1 Polyhedral Surfaces

A boundary representation of a polyhedral surface consists of a set of vertices V, a set of edges E, a set of facets F, and an incidence relation on them. Introductions can be found in [95, 124]. For a living example see Figure 4.1.

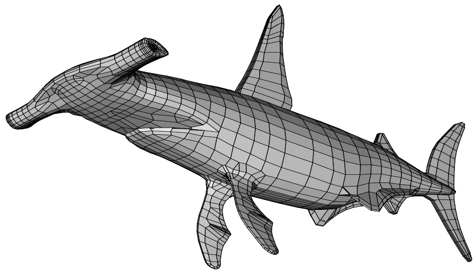


Figure 4.1: Hammerhead, an orientable 2-manifold of 2560 vertices. This one is homeomorphic to a sphere.

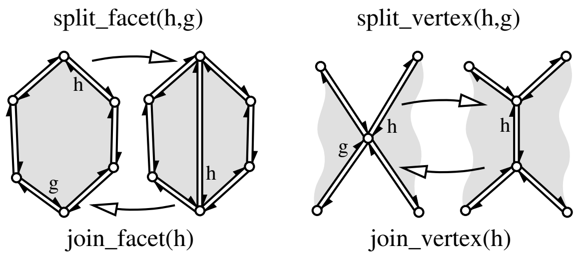


Figure 4.2: Euler operator examples for polyhedral surfaces.

The two types of boundary representations are 2-manifold and non-manifold surfaces. For each point on a 2-manifold surface there exists a neighborhood that is homeomorphic to the open disc. A non-manifold example would be two tetrahedra glued together at a single vertex or a common edge. The next distinction is between orientable and non-orientable 2-manifold surfaces. Without going into details, a surface is orientable if a consistent orientation can be assigned to each facet such that for each edge the two incident facets have opposite orientations at this edge. An example of a non-orientable 2-manifold is the Klein bottle. We consider only orientable 2-manifolds for our data structure.

The natural operations under which 2-manifolds are closed are Euler operations; four of them are shown in Figure 4.2. The principal characteristic of an Euler operation is the invariance of the Euler-Poincaré formula. A sufficiency proof for a specific set of Euler operations can be found in [124]. Note that 2-manifolds are not closed under boolean operations.

The class of representable surfaces is further restricted by the kind of geometry associated with vertices, edges and facets. Vertices map to points in $\mathbb{R}^3$. As far as polyhedra are concerned, the edges are typically the straight line segments between their two endpoints and the facets are simple, planar polygons. Other classes might allow curved surfaces as facets or might use vertices in projective space to model unbounded facets as well. However, they are currently not provided in CGAL.

We now present a definition for polyhedral surfaces following Steinitz [176]. The combinatorial nature of this definition makes reasoning about the data structure more convenient. For example, the same facet cannot appear on both sides of an edge. And it leads directly to the integrity definition and related test function of the polyhedral surface data structure. The definition captures the same class of representable surfaces informally introduced so far, but may require a different combinatorial representation. If a facet is incident to both sides of the same edge the facet would be split into two facets.

Definition 4.1.1. A *structural complex* is a union $C = V \cup E \cup F$ of three disjoint sets together with an incidence relation. We call V the vertices, E the edges and F the facets of the structural complex. The incidence relation on C must be symmetric. No two elements from the same set V, E or F are incident. If $v \in V$ is incident to $e \in E$ and e is incident to $f \in F$ then v is incident to f.

Definition 4.1.2. A *polyhedral complex* is a structural complex with four additional conditions.

(1) Every edge is incident to two vertices.

(2) Every edge is incident to two facets.

(3) For every incident pair v, f, there are exactly two edges incident to both.

(4) Every vertex and every facet is incident to at least one other element.

The *neighborhood* of a vertex are the edges and facets incident to the vertex. If we restrict the incidence relation to this neighborhood, then by condition (3) each facet is incident to exactly two edges and by condition (2) each edge is incident to exactly two facets. Thus, the neighborhood decomposes into disjoint cycles, each cycle being an alternating sequence of edges and facets. A polyhedral complex is a 2-manifold if and only if the neighborhood of each vertex decomposes into a single cycle. The definition of a polyhedral complex is symmetric for vertices and facets. A symmetrically defined neighborhood of a facet decomposes into cycles of incident edges and vertices. Assuming that the neighborhood of each facet is a single cycle (geometrically, the boundary of the facet is a single connected component, so the facet has no holes), we can define a polyhedral complex to be *oriented* if each cycle around a facet is oriented and if for each edge the two cycles of its two incident facets are oriented in opposite directions. A polyhedral complex is *orientable* if there exists such an orientation.

Definition 4.1.3. The *boundary representation of a polyhedron* is an orientable polyhedral complex, where the neighborhood of each vertex and each facet is a single cycle, together with a mapping $V \to \mathbb{R}^3$. This mapping extends for edges by mapping each edge to the open, straight line segment between its two endpoints. The following additional conditions must hold.

(5) The image of the neighborhood of each facet is the boundary of a simple, planar polygon. The mapping extends for facets to the open region of these polygons.

(6) For all elements in $C = V \cup E \cup F$, their images are pairwise disjoint.

Some useful properties of this boundary representation: The surface is an orientable 2-manifold. The neighborhoods of two vertices have at most one edge and two facets in common. The edge and vertex graphs are connected within each connected component of the surface. Each facet has at least three edges on its boundary. The smallest possible configuration is a tetrahedron.

The closed surfaces considered so far can be extended to surfaces with boundaries. We need to relax Condition (2) to allow edges that are incident to one facet; they are called *border edges*. This induces a modification of Definition 4.1.3: The neighborhood of a vertex decomposes into either a single cycle or a collection of open paths going from border edge to border edge. Although the surface is no longer closed, the orientation still defines a "solid" side of the surface. The minimal configuration for surfaces with boundaries is a triangle. The data structures we will describe can be used for polyhedra as well as for surfaces with boundaries. Border edges are simply marked by the missing facet at one side, i.e. by a null pointer.

A suitable data structure based on the Definition 4.1.3 for polyhedral surfaces has been successfully used in the project on contour-edge-based polyhedron visualization (see Chapter 6) where we take advantage of the strict properties imposed by the definitions. For example, the definition for contour-edges is based on the orientable 2-manifold property, and the lack of holes in facets simplifies our algorithms. An initial implementation of the data structure made it easy to compute the silhouette for a polyhedral surface [97]. The extension of this data structure design and their advantages are presented in the following sections.

4.2 Data Structures for Boundary Representations

The following survey of edge-based data structures addresses their sufficiency for modeling topology and the efficiency of their primitive operations and storage costs. The representative example chosen is the traversal around a vertex from one edge to the next edge in counterclockwise direction.

Winged-Edge Data Structure. The winged-edge data structure [19, 83] stores, for each oriented edge, eight references: two vertices (PVT, NVT), two faces (PFACE, NFACE) and four incident edges that share the same faces and vertices (PCW, PCCW, NCW and NCCW), the so-called *wings*, see Figure 4.3. An edge is oriented from the source vertex PVT to the target vertex NVT. The face PFACE is to the left of the oriented edge when the surface is seen from the outside.

This data structure is able to model orientable 2-manifolds. It is even sufficient for curved-surface environments where loops and multi-edges are allowed [190]. The basic operations include traversal around a vertex and around a facet. High-level

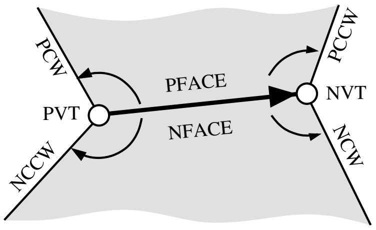


Figure 4.3: Winged-edge.

operations maintaining integrity are Euler operators. The next edge counterclockwise around a vertex `v` for an edge `e` is equal to `e->PCW` if `e->PVT == v` and `e->NCW` otherwise.

Variants are possible where vertex and facet pointers can even be omitted without losing the traversal capabilities if the previously visited edge is known. However, if loops or multi-edges are allowed, all four edge pointers must remain. Otherwise the traversal around a vertex or around a facet is no longer uniquely defined [190]. The winged-edge data structure with the wings PCCW and NCCW omitted has been called **Doubly Connected Edge List** (DCEL) by [141]. Now this name is more commonly used for the halfedge data structure [55].[1]

The two symmetric parts in the winged-edge correspond to the two possible orientations of the edge. The inefficient case distinction in the traversal computation results from the fact that a pointer to an edge does not encode the orientation it is currently used with. One extension of the winged-edge maintains an additional bit with each edge-pointer to code the orientation, but this can lead to cumbersome storage layouts and function interfaces.

Halfedge Data Structure. The orientation problem can be solved for the winged-edge data structure by splitting the edge into two symmetric records, called *halfedges*, and adding mutual links to each other [190]. There are two ways of splitting the edge: Either the edge is split along the facets, such that the oriented halfedges belong to the two facets incident to this edge (see Figure 4.4) or it is split into two halfedges belonging to the two vertices incident to this edge (see Figure 4.5). These two variants are dual to each other considering the usual notion of duality for graphs. Vertex and facet are dual to each other and the dual of an edge is an edge with the two incident facets as its endpoints.

In both splitting variants a halfedge contains a pointer to an incident vertex, an incident facet, and the opposite halfedge. It is a matter of convention whether the source or target vertex is the one chosen to be stored in a halfedge or whether the facet to the left or the right is stored. In [190] the source vertex and the facet to the right were chosen. The FE-structure in Figure 4.4 additionally stores a pointer to

[1]In order to avoid confusion we will not use the name DCEL since it turned out to be ambiguous. In fact, the name is misleading when denoting halfedges and the possible variants of single linking.

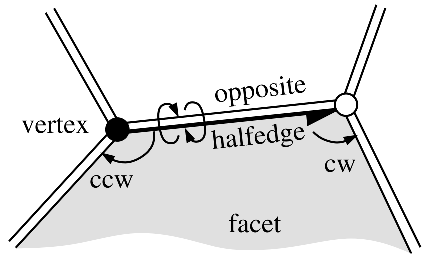


Figure 4.4: FE-structure.

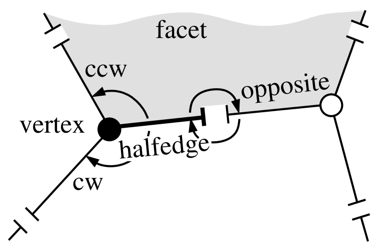


Figure 4.5: VE-structure.

the next clockwise halfedge and optionally a pointer to the previous counterclockwise halfedge around the facet. It is therefore biased towards traversals around the incident facet. The dual VE-structure is depicted in Figure 4.5. Its next and optional previous pointer refer to halfedges counterclockwise and clockwise around the incident vertex. The traversal operation that is not directly accessible with a single pointer access is available through the opposite halfedge. For example, the next halfedge around the incident source vertex for the FE-structure is `opposite()->next()`. The different conventions are not independent. If the convention defines the halfedge order around a facet to be clockwise, the halfedge order around the vertex will be counterclockwise, and vice versa.

The halfedge data structure is able to model orientable 2-manifolds. It is sufficient for modeling topology even in the presence of loops and multi-edges, which can occur in curved-surface environments [190]. High-level operations maintaining integrity are again Euler operators. The solid modeling system GWB is based on a halfedge-data structure, though it uses an additional edge record between two opposite halfedges, making this access less efficient [124]. Similarly, the ACIS kernel for CAD modelers distinguishes between co-edges, which carry the orientation information, and edges, which carry the incidence information. This representation is also used for non-manifolds in the ACIS Kernel [49]. The Minimal Rendering Tool MRT uses a halfedge data structure for polygonal surfaces [20].

Quad-Edge Data Structure. Similar to the idea for halfedges, each edge is split into four quad-edges to obtain the quad-edge data structure [91]. It provides a fully symmetric view on the primal and the dual graph as can be seen in Figure 4.6. Instead of using opposite pointers, a two bit counter r is used to address a slot in an edge record e of four quad-edges. This data structure is able to model non-orientable 2-manifolds with an additional bit f denoting the flipped status per edge.

A quad-edge data structure is formally defined as an edge algebra with three operations: `Onext()`, `Rot()` and `Flip()`. A quad-edge is represented as a triple (e, r, f) with e a record of four quad-edges $e[0]$ to $e[3]$ incident to the current edge, $r \in \{0, 1, 2, 3\}$ and $f \in \{0, 1\}$. The operations are implemented with a calculus modulus 4 for r and modulus 2 for f as follows:

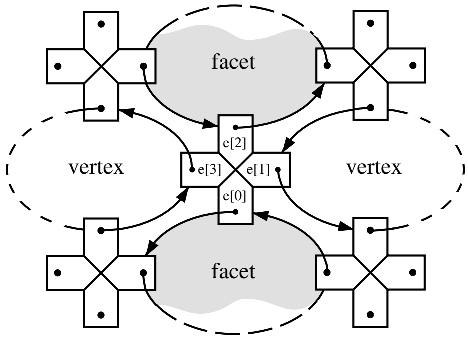


Figure 4.6: Quad-edge data structure.

$$\begin{aligned}
\texttt{Rot}(e,r,f) &= (e, r+1+2f, f),\\
\texttt{Flip}(e,r,f) &= (e, r, f+1),\\
\texttt{Onext}(e,r,f) &= \begin{cases} e[r] & \text{for } f=0 \\ \texttt{Flip}(\texttt{Rot}(e[r+1])) & \text{for } f=1\ . \end{cases}
\end{aligned}$$

Four different orientations of an edge are considered: two orientations from vertex to vertex and two orientations for the dual edge from facet to facet. The `Rot` operator rotates the edge by 90 degrees, oscillating between the primal and the dual view of the structure. For non-orientable 2-manifolds an edge additionally can be seen from above or below the surface, which is encoded in the f bit. The `Flip` operation changes the view from above to below or vice versa. The `Onext` operation gives the next quad-edge in counterclockwise order around the source vertex (origin), or the next quad-edge in clockwise order if f is equal to one. The values for `Onext` are simply stored in the record for each edge (i.e. four pointers and four times three bits for r and f). The operations simplify considerably for orientable 2-manifolds. They can be further simplified if the dual graph is not needed. The result would be the winged-edge data structure enriched with a bit to encode orientation.

The single high-level operation that modifies a quad-edge data structure is the `Splice()` operation. It is its own dual. The usual Euler operators can be implemented in terms of `Splice()`. The quad-edge data structure provides a unified view for the primal and dual graph. This implies that vertices and facets cannot be distinguished with strong type checking at compile time. The definition used for duality implies, furthermore, that the facets must have a single connected boundary. Holes in facets are not allowed. If strong type checking is desired, the `Splice()` operation is needed twice, once for the primal view and once for the dual view. `Splice()` can also be provided for the halfedge data structure.

Comparison of Edge-Based Representations. The main differences of these edge representations are captured in Table 4.1. The differences in the basic traversal

	Winged-Edge	Half-Edge	Quad-Edge
Modeling space	orientable 2-manifold		2-manifold
Operations	Euler operator		`Splice()`
Duality	adaptor at compile time		`Rot()` at runtime
Holes in facets	yes	yes	no
Basic traversal	case distinction	direct access	mod operation
Min size per edge	4 ptr	4 ptr	2 ptr + 2 bits
Max size per edge	8 ptr	10 ptr	8 ptr + 12 bits

Table 4.1: Comparison of the edge-based data structures.

capabilities are not negligible, especially when considering modern microprocessor architectures where conditional branching can be slower than computing by an order of magnitude. The storage size requirements are quite similar. Our design will focus on the flexibility of trading runtime against storage costs. We are interested in the minimal and maximal configurations for the halfedge data structure and the space efficiency of the quad-edge data structure. Another issue is the preference for strong type checking at compile time. Polyhedral surfaces have different information stored in the vertices and facets, namely points and plane equations. These can be treated as duals of each other, but in a strongly-typed geometry kernel (like the one CGAL provides) they are different types and might even be represented differently. Additional information, like color, will finally destroy the typeless symmetry of the duality assumed by the quad-edges. Holes in facets can be modeled with the winged-edge and the halfedge data structure, but the vertex-edge graph may be disconnected even for connected surfaces. Furthermore, if there are no holes in facets, the dual facet-edge graph is also a 2-manifold. We neglect non-orientable 2-manifolds, for example, the three-dimensional surface of a solid physical object is always orientable. With respect to a library design, we would refer to a more general non-manifold data structure to model non-orientable 2-manifolds.

We have chosen a halfedge data structure comparable to the FE-structure. The conventions used are depicted in Figure 4.7. We have `next()`, `opposite()` and `prev()` pointers for the halfedges. The incident vertex is the target vertex of the oriented halfedge. The incident facet is to the left of the halfedge which implies a counterclockwise ordering of the halfedges around the facet and a clockwise ordering around the vertex when seen from the outside. This complies with the right-hand rule for out-facing normals of plane equations for facets.

4.3 Related Work

The design presented here is a major revision of the design described in [107] as it is available in CGAL since Release 1.0 [34]. In the original design, the user provides the space for the incidence information in the vertices, halfedges and facets

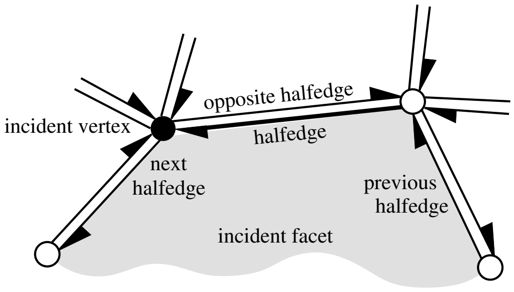


Figure 4.7: Halfedge data structure.

in terms of `void*` pointers. The halfedge data structure uses type-casts to provide correct type-safe pointers and the polyhedron converts the pointers to iterators and handles. The main drawback besides the tricky, nested implementation is that users cannot extend their vertices, halfedges and facets conveniently with further incidences. These new incidences are not captured in the type-casts provided with the halfedge data structure. In the revised design presented here, all layers of the design share the same type informations. Vertices, halfedges and facets can use the correct handle types to provide the space for the various incidences even though they are still decoupled from each other, e.g., the vertex type does not hard-code the actual type of its related halfedge type. Internally, this new design saves for us an implementation layer that was previously needed for the type-casting.

The Library of Efficient Datatypes and Algorithms (LEDA[2]) [132, 133, 131] provides a rich body of algorithms and data types. LEDA includes for example number types for exact arithmetic, dictionaries, priority queues, graph algorithms, and two-dimensional geometry. Its design is homogeneous and easy-to-use. LEDA has its own notion of iterators and it is generic and flexible within its own framework. This framework was incompatible with the approach taken by STL, but recent extensions make LEDA to comply with STL. In fact, to replace STL.

LEDA contains no data structure specifically tailored for three-dimensional polyhedra, but it provides a general data structure for s and one for s derived from the graph data structure. Additional information can be attached with node arrays and edge arrays or by using a parameterized graph.

A parameterized graph is a class template in LEDA with two template parameters: One for the auxiliary information in the vertices and one for the auxiliary information in the edges. The disadvantage is that if only one of the two types is needed, a dummy type must be given to the other template argument which wastes memory.

Node and edge arrays are associative arrays in LEDA based on hashing, which allow the easy addition of information even for temporary purposes. The disadvantages are the additional cost for the lookup operation and the additional memory consumption. A more subtle disadvantage is revealed when, for example, an iterator

[2] `http://www.mpi-sb.mpg.de/LEDA/leda.html`

over graph nodes is considered: A simple pointer to the node in general is not sufficient for the state of the iterator, since the iterator would not be able to give access to the associated attributes of the node. An additional reference to the associative arrays must be stored in the iterator (or alternatively in the node).

The current memory requirements for graphs in LEDA are equivalent to 13 pointers for a graph node and 11 pointers for a directed graph edge. There is no flexibility for obtaining smaller graph structures. LEDA does not reach the flexibility in tuning the runtime and space efficiency as achieved with the approach presented here.

The Minimal Rendering Tool MRT uses a halfedge data structure to represent polyhedral surfaces [20]. It is implemented as a C++ class hierarchy and provides Euler operations to maintain combinatorial integrity. The internal representation is accessible at construction time and protected thereafter. No other access is granted. It separates geometry and topology except for vertices where a point is incorporated just at the combinatorial level for efficiency reasons. Flexibility is achieved with virtual member functions for geometric properties. No flexibility is available at the topological level. Facets are responsible of storing the ring of halfedges of their boundary. In summary, this approach leads to larger nodes for vertices, halfedges and facets and slower functions for geometric properties than the solution we propose.

In visualization, a polyhedron is usually represented as a so-called indexed facet set which is a sequence of points followed by a sequence of facets. Each facet is a list of indices referring to the sequence of points. Shared edges must be derived implicitly from the points shared by facets. Examples are the internal representation as well as the file formats of VRML [93], Open Inventor [194], OpenGL [106, 149], Java 3D [175], or the Object File Format OFF used by GeomView [158]. These formats are not strict enough for our purposes since they can represent non-manifolds, non-orientable 2-manifolds, and may also violate condition (3) of a polyhedral complex. However, these formats cannot represent holes in facets.

The next generation library OpenGL Optimizer [174] and its successor [102] support the modeling of topology beyond the simple indexed facet set. They represent edges and their incidences explicitly with an extension of the winged-edge data structure for non-manifold surfaces. The interesting change in the perception of topology in industry is captured in the following quote from the OpenGL Optimizer white paper [88], and we believe that knowing the topology is good not only for back-face culling (see for example our work on hidden-surface removal in Chapter 6):

> Other model repair tasks can be performed once topology is known, such as ensuring that adjacent patches are properly oriented. Since orientation effects shading, repairing this can increase graphics performance by a factor of two on some systems. Thus, topology construction, specification, and maintenance is a key component for both high fidelity and high performance in CAD model visualization. [88]

4.4 Design Goals for Polyhedral Surfaces

A polyhedron can be viewed as a container class managing the vertices, halfedges, and facets of a polyhedral surface and maintaining their combinatorial structure. The following design issues were taken into account for our design:

1. The actual storage organization of the vertices, edges and facets influences the space and runtime efficiency. A doubly-connected list representation allows random insertion and removal while providing bidirectional iterators to enumerate all items. A more space efficient storage uses an STL vector, which allows only the efficient removal of items at the end of the vector, but provides random-access iterators. Other variants are possible, such as managing chunks of memory or simple allocation on the heap without any iterators.

2. The necessary incidence relations might depend on the application. In order to keep the halfedge data structure connected, we need at least a `next()` and an `opposite()` pointer per halfedge. The other incidences are optional. For example, the `prev()` pointer can be simulated with a search around the vertex or around the facet in the opposite direction. For triangulations this is a simple expression, i.e., `prev()` $\equiv$ `next()->next()`. Assuming a constant degree at vertices or a constant number of edges for facets, it is still a constant time operation. If no information needs to be attached to vertices or facets, no storage should be allocated for them, even the referencing pointers in the halfedge should be omitted. In the limit the data structure reduces to an undirected graph structure.

3. It should be easy for a user to add additional information to the different items, e.g., color to facets. Geometry will be attached using the same technique. Redefining one item should not hinder the re-use of the other items, for example, adding color to facets should not imply that a new vertex type must be declared.

4. The data structure should provide an easy-to-use high-level interface. This interface should protect the internal combinatorial integrity of the data structure as given in Definition 4.1.3. Advanced algorithms concerned with efficiency, e.g., a file scanner, should be allowed to access the internal structure in a controlled manner.

5. The management of connected components and containment relations, e.g., inner and outer boundaries for facets and nesting relationships of shells, is seen as an independent functionality that can be added separately, for example in its own layer and with its own flexibility.

6. The edge-based data structures discussed in Section 4.2 have a natural notion of the edges around a vertex or the edges around a facet. It would be costly to provide iterators for these kind of circular sequences, which is why we invented for this kind of structure the new concept of *circulators*.

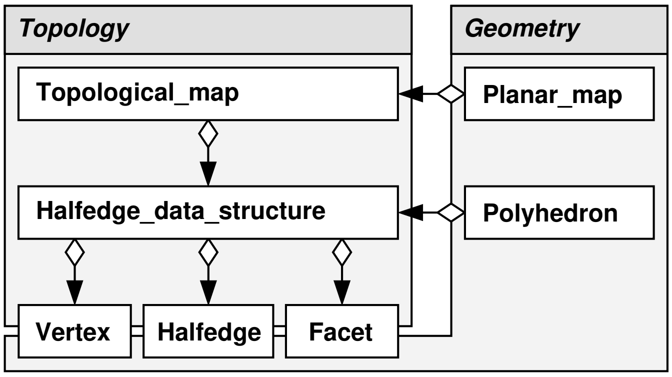


Figure 4.8: Overview of the design with the separation of topology and geometry.

7. STL containers base their interfaces on iterators. For polyhedral surfaces the order in which the items are stored in the polyhedron is not always well-defined from the perspective of the user, e.g., after Euler operations. Here we fall back on the concept of *handles*, which is the item-denoting part of iterators, and ignore the traversal capabilities. In particular, any model of an iterator or circulator is a model of a handle. A handle is also known as *trivial iterator*.

In the remainder of this chapter we concentrate on the combinatorial aspects of the polyhedral surface. Additional issues appear when considering geometry, for example flexibility in the point type and the geometric predicates. The technique explored for this are the geometric traits classes from Section 2.6.

4.5 Design Overview

Figure 4.8 illustrates the separation of topology and geometry in the design. Vertices, halfedges and facets store both. The container class `Halfedge_data_structure` manages these three items and their topological relations. The `Topological_map` adds to the halfedge data structure the management for holes in facets, which enumerates inner and outer boundaries for facets [73]. It is usually classified as a face-based representation. The `Polyhedron` adds geometric operations to the `Halfedge_data_structure`. It is based on the Definition 4.1.3 for polyhedral surfaces and guarantees a consistent representation (besides the costly self-intersection test and the polygon planarity test). The halfedge data structure is not restricted to these definitions, since it is also useful in implementing other data structures, but it will be used by the polyhedron to represent only surfaces following this restricted definition, for example, an edge has always two distinct endpoints. The `Planar_map` is based on the topological map, since it maintains holes in facets.

The `Halfedge_data_structure` and the items `Vertex`, `Halfedge` and `Facet` are concepts. Currently three different models are provided for the `Halfedge_data_`

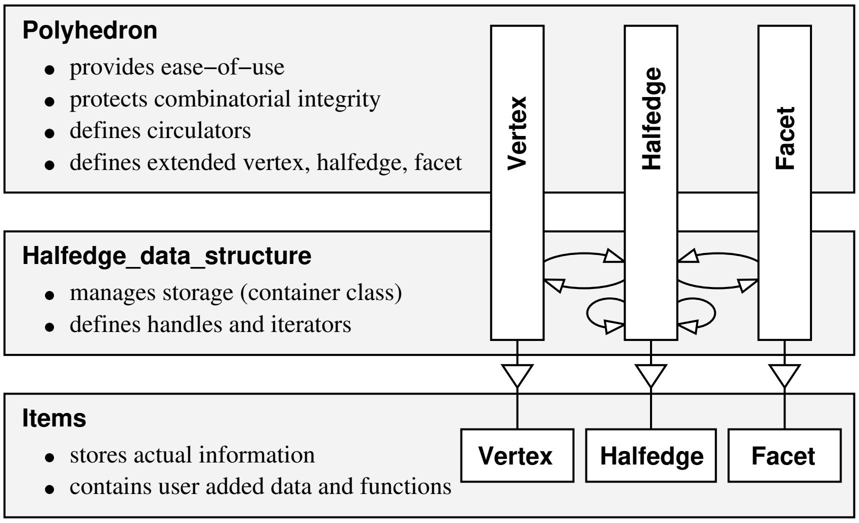


Figure 4.9: Responsibilities of the different layers in the design.

`structure`: One is based on the STL vector, the other two are based on list representations to manage the items internally. Various models are provided for the items and they can be easily extended by the user, for example with additional attributes.

Figure 4.9 illustrates the three layers of the polyhedral surface design: `Items`, `Halfedge_data_structure`, and `Polyhedron`. The items provide the space for the information that is actually stored, i.e., with member variables and access member functions in `Vertex`, `Halfedge`, and `Facet` respectively. Halfedges are required to provide a reference to the next halfedge and to the opposite halfedge. They may provide an optional reference to the previous halfedge, to the incident vertex and to the incident facet. Vertices and facets may be empty. They may provide an optional reference to the incident halfedge. These optional references are supported in the halfedge data structure and the polyhedron, for example, Euler operations update these optional references if they are present. Furthermore, the item classes can be extended with arbitrary attributes and member functions, which will be promoted by inheritance to the actual classes used for the polyhedral surfaces.

Implementations for vertices, halfedges and facets are provided that fulfill the mandatory part of the requirements. They can be used as base classes for extensions by the user. Richer implementations are also provided to serve as defaults; for polyhedra they provide all optional incidences, a three-dimensional point in the vertex type and a plane equation in the facet type.

Vertices, halfedges and facets are passed as local types of the `Items` class to the halfedge data structure and polyhedron. This is an implementation detail explained in more depth in Section 4.7.

The `Halfedge_data_structure` is responsible of the storage organization of the items. Currently, implementations using internally a bidirectional list or an STL vector are provided. The `Halfedge_data_structure` defines the handles and iterators belonging to the items. These types are promoted to the declaration of the items themselves and are used there to provide the references to the incident items. This promotion of types is done with a template parameter of the item types, see Section 4.7. The halfedge data structure provides member functions to insert, to delete, and to traverse items.

There already are three different models for the `Halfedge_data_structure` available. Therefore we have kept their interface small. Functionality common to all these models is separated into a helper class `Halfedge_data_structure_decorator`, which is not shown in Figure 4.9, but would be placed at the side of the `Halfedge_data_structure` since it broadens that interface but does not hide it. This helper class contains operations that are useful to implement the operations in the next layer, such as the polyhedron. For example, it adds the Euler operations and partial operations from which further Euler operations can be built, such as inserting an edge into the ring of edges at a vertex. Furthermore, the helper class contains adaptive functionality. For example, if the `prev()` member function is not provided for halfedges, the `find_prev()` member function of the helper class searches in the positive direction along the facet for the previous halfedge. But if the `prev()` member function is provided, the `find_prev()` member function simply calls it. This distinction can be resolved at compile time with a technique called *compile-time tags*, similar to iterator tags in [177], see Section 4.7.

The `Polyhedron` provides an easy-to-use interface of high-level functions and hides the flexibility provided underneath. The interface is designed to protect the integrity of the internal representation. Handles in particular are no longer mutable. The polyhedron adds the convenient and efficient circulators (see Section 2.3) for accessing the circular sequence of edges around a vertex or around a facet. To achieve this, the `Polyhedron` derives new vertices, halfedges and facets from those provided in `Items`. These new items are those actually used in the `Halfedge_data_structure` which gives us the coherent type structure in this design, especially if compared to our previous design [107]. The `Polyhedron` also adds the geometric interpretation. Therefore, the vertices used for polyhedron are required to store a geometric point.

4.6 Program Examples

Default template arguments allow us to hide the flexibility provided for the halfedge data structure. Therefore, in the first example program the polyhedron only needs to be instantiated with the geometric traits class representing the geometric kernel, for example, the Cartesian kernel using doubles:

```
typedef CGAL_Cartesian<double>                    Kernel;
typedef CGAL_Polyhedron_default_traits_3<Kernel> Traits;
typedef CGAL_Polyhedron_3<Traits>                 Polyhedron;
```

The following function applies an affine transformation to all points in the polyhedron. Note the use of `transform` which is a generic algorithm from STL. Our design encourages such re-use. Instead of providing an own function for affine transformations, the polyhedron provides the iterator for accessing all points, and the affine transformation of CGAL is an unary function object suitable for the `transform` function.

```
void apply( Polyhedron& P, const CGAL_Aff_transformation_3<Kernel>& A) {
    transform( P.points_begin(), P.points_end(), P.points_begin(), A);
}
```

The default polyhedron already provides a plane equation for facets. In the following example we replace the default facet type with our own facet type providing only a normal vector instead of a full plane equation. For that we define our own facet type and replace the facet in the default items type we have used so far. We make use of a predefined facet base class. The template argument `Refs` will be explained in Section 4.7. It is used to make the handle and iterator types of the halfedge data structure available in the items. The `Traits` argument provides a local type `Vector_3` that we can use for the normal vector type.

An items class contains three template member classes representing a vertex type, a halfedge type and a facet type respectively. Their names and template arguments are fixed, since they are used in the halfedge data structure to create objects of these types. We use inheritance from the default items class of the polyhedron to re-use the old vertex and halfedge types. Only the facet is replaced. The related template member class must provide a local type `Facet`, which is our redefined facet type. The new items class can be used as a second template argument to the polyhedron class, where it replaces the default items class. A third template argument can be used to replace the default halfedge data structure, but this is not illustrated here. The `Traits` argument is the same as above.

```
template <class Refs, class Traits>
struct My_facet : public CGAL_HalfedgeDS_facet_base<Refs> {
    typename Traits::Vector_3 normal;
};
struct My_items : public CGAL_Polyhedron_items_3 {
    template <class Refs, class Traits>
    struct Facet_wrapper {
        typedef My_facet<Refs, Traits> Facet;
    };
};
typedef CGAL_Polyhedron_3<Traits, My_items> Polyhedron;
```

Our next example computes the normal vectors of the facets using the modified facet type from our previous example. The generic `for_each` algorithm of STL can be used to apply a function object to each facet of the polyhedron. The function object assumes convex facets to compute the out-facing normal vector and it ignores numerical stability issues. We obtain three consecutive vertices on the boundary of the facet and compute the normal vector with a cross product. The function object uses a template member function which gives us a generic implementation that can be used with any facet type with a `normal` member variable and that keeps a reference to an incident halfedge. The facet knows the type for the vertex handle.

```
struct Normal_vector {
    template <class Facet>
    void operator()( Facet& f) {
        typedef typename Facet::Vertex_handle Vertex_handle;
        Vertex_handle p = f.halfedge()->vertex();
        Vertex_handle q = f.halfedge()->next()->vertex();
        Vertex_handle r = f.halfedge()->next()->next()->vertex();
        f.normal = CGAL_cross_product( q->point() - p->point(),
                                       r->point() - q->point());
    }
};
void compute_normals( Polyhedron& P) {
    for_each( P.facets_begin(), P.facets_end(), Normal_vector());
}
```

4.7 Implementation of the Design Using Templates in C++

We present the necessary details of an implementation based on templates in C++ to prove the feasibility of our design. We begin with a simplified version presenting the technique we use to decouple the item types and the halfedge data structure. Thereafter we present the realization of the polyhedron class. We omit details such as const-correctness and the CGAL prefix. Note that the following details of the design must be known only by the developer and not the user of the data structure.

The key is that a certain item type, for example a vertex, does not know the actual types of the related items like the halfedge type. Instead, it knows a formal placeholder for this type. We put all formal placeholders together in a single template argument `Refs` as local types. We indicate only the use of `Halfedge_handle` in the vertex type which provides the reference to the incident halfedge.

```
template <class Refs>
struct Vertex {
    typedef typename Refs::Halfedge_handle Halfedge_handle;
    Halfedge_handle halfedge()                         { return h; }
    void            set_halfedge( Halfedge_handle g) { h = g; }
private:
    Halfedge_handle h;
};
```

The other item types are implemented similarly. The halfedge data structure is parameterized with the item types. Since the item types are already class templates, we need templates as template arguments for the halfedge data structure. The order of type declarations in the halfedge data structure is the vertex and halfedge type, the container classes for these item types, and the handle types known from the container classes. The type dependencies contain a cycle; the item types need a template argument that tells them the handle types. The halfedge data structure knows the handle types and can be used as the actual type for the `Refs` argument of the item types, even though the handles have not been defined at this point. However, it is the difference between declaration and definition that allows us to use the declared type of the halfedge data structure for the item type instantiation. With respect to this instantiation our approach is similar to the template pattern described in [47], although we make no use of inheritance. The facet type is omitted in this example.

```
template < template <class> Vertex,  template <class> Halfedge>
struct HalfedgeDS {
    typedef HalfedgeDS< Vertex, Halfedge> Self;
    typedef Vertex<Self>                  V;
    typedef Halfedge<Self>                H;
    typedef list<V>                       Vlist
    typedef list<H>                       Hlist
    typedef typename Vlist::iterator      Vertex_handle;
    typedef typename Hlist::iterator      Halfedge_handle;
    // ...
};
```

This concludes the simplified version. What remains to be shown are the intermediate items class, the polyhedron class with its protected access to the items, the connection to the geometric kernel with the traits class, and the *compile-time tags* to adapt functions to the flexibility provided in the items. An item type usually has two template arguments, `Refs` as explained above and a `Traits` argument. The traits contains the geometric types and the basic operations available for these types [71]. The optional functionality of the item types is indicated with tags. In the example of a vertex type from above we add the local type `Supports_vertex_halfedge`. It can be either of the two predefined classes, `Tag_true` or `Tag_false`. This tag indicates whether the reference to the incident halfedge is provided or not.

```
template <class Refs, class Traits>
struct Vertex {
    typedef Tag_true Supports_vertex_halfedge;
    // ...
};
```

Other functionality can make use of this tag to adapt at compile time. For example, the Euler operations are implemented using this *compile-time tag*. The Euler operations update the reference to the incident halfedge automatically if this reference is available in the vertex. The following example uses function overloading in C++ to distinguish between two different implementations of a function `foo` depending on the tag. The single-argument function is called with a vertex as argument. The function call is forwarded to the corresponding two-parameter function that matches the actual type of the compile-time tag.

```
void foo( Vertex v) {
    foo( v, Vertex::Supports_vertex_halfedge());
}
void foo( Vertex v, Tag_true)  {
    // ... implementation making use of the v.halfedge() method.
}
void foo( Vertex v, Tag_false) {
    // ... implementation not making use of the v.halfedge() method.
}
```

The intermediate items class puts together the definition of the three item types: vertex, halfedge and facet. It uses member template classes, but itself is not a template class. It can be passed around without instantiating the item types. Furthermore, the halfedge data structure can be written without templates as template parameters. The names and template arguments within the items class are fixed: The member class templates are called wrapper and have two template arguments, `Refs` and `Traits`. These wrappers must provide a local type named after the corresponding item, `Vertex` for example, that refers to the actual class used. Besides technical reasons, it is convenient to pass the item types in a single parameter to the halfedge data structure. The single parameter is easier to use with items classes that are already defined in the library. On the other hand, a new items class must be derived if only a single item type is exchanged (see the example in the previous section). Another advantage of the items type is a possible separation of two kind of template parameters: User-specified parameters and those specified in the halfedge data structure, namely `Refs` and `Traits`. Further user-specified parameters cannot be passed as additional template arguments through the halfedge data structure, but the user can make the items class a template class for itself and can provide the actual types when instantiating the polyhedron. The items class for our example without additional user parameters looks as follows:

```
struct HalfedgeDS_items {
    template <class Refs, class Traits>
    struct Vertex_wrapper {
        typedef Vertex<Refs, Traits> Vertex;
    };
    // ... similar for halfedge and facet
};
```

The polyhedron derives new item types from the given item types to enhance functionality, e.g., circulators, and to protect the combinatorial structure. Member functions that allow changing the incidences of items are made private, while the remaining functionality and especially the user added functionality remains available to the user because of the public inheritance. Another solution would be to repeat explicitly the functionality that should remain public, but user added functionality could not be captured this way and would be lost. The solution chosen may sound weak with respect to protection, but since the user provides the bases, the user can always work around any protection mechanism. Our solution prohibits accidental misuse, the main purpose of protection.

The new item types are collected again in an items class which is used in the halfedge data structure of the polyhedron. As a consequence, the halfedge data structure uses the correct items defined by the polyhedron. The original items provided by the user, therefore, are parameterized with the handles to the derived item types which are the actual types used. We have the coherent type system where all handles refer to the actual item types.

But the consistent use of the derived item types also prohibits the use of incidences by the polyhedron and the halfedge data structure which should be allowed. We provide the type of the base class in the derived item types. The halfedge data structure and the polyhedron now can change incidences by calling the member functions of the base class, similar to the implementation of the `set_halfedge` member function in the following example.

```
template <class Vertex_base>
struct Polyhedron_vertex : public Vertex_base {
    typedef Vertex_base Base;
private:
    void set_halfedge( typename Base::Halfedge_handle g) {
        Base::set_halfedge(g);
    }
};
template < class Items>
struct Polyhedron_items {
    template <class Refs, class Traits>
    struct Vertex_wrapper {
        typedef typename Items::Vertex_wrapper<Refs, Traits> Wrapper;
        typedef typename Wrapper::Vertex Vertex_base;
        typedef Polyhedron_vertex<Vertex_base> Vertex;
    };
    // ... similar for halfedge and facet
};
```

The polyhedron is a class template with three template parameters: `Traits`, `Items` and `HDS`, two of them have default arguments. It declares the derived item types and instantiates the halfedge data structure `HDS` to use it as internal representation. The `HDS` parameter is again a template as template parameter. If the halfedge data structure had used a template as template parameter for itself, we would have had a third level of templates, which is not allowed in C++. The items class avoids this, but other workarounds are possible as well.

```
template < class Traits,
           class Items = Items_default,
           template <class, class> class HDS = HalfedgeDS_default>
struct Polyhedron {
    typedef Polyhedron_items<Items> Derived_items;
    HDS <Traits, Derived_items> hds;
    // ...
};
```

The halfedge data structure from above basically remains unchanged. Instead of the separate item types we pass a traits parameter and an items class parameter. The item types are extracted from the wrappers in the items class.

Given a set of predefined implementations for these classes in a library, it is easy to combine and extend them. The flexibility we have demanded for our design in Section 4.4 is realized. The default list representation in the halfedge data structure can be replaced by other representations. Only those incidences are stored which we actually encode in the bases, and the tags are used to implement adaptive functionality corresponding to the incidences provided. For example, vertices are not even allocated in the halfedge data structure if they are not referenced from the halfedges. Adding additional information can be easily done by deriving own item types and replacing their definitions in the items class. All this is achieved without runtime nor storage overhead, nor any compromises for the ease-of-use at the top-level.

4.8 Granting Access to Internal Representation

Algorithms on polyhedra may have intermediate states that are invalid representations. To be efficient, they need access to the internal representation. We grant a protected access for all subclasses of `Modifier_base`. Our design was motivated by the strategy pattern [79] but has a different intent.

Figure 4.10 illustrates the class design for the example of a file scanner creating a polyhedron from a file. The approach is similar to a callback function, only encapsulated as a function object. The `Polyhedron` accepts a modifier object as an argument of its `delegate()` member function and calls the virtual `operator()` member function of this modifier object. Thereby, it passes the internal halfedge data structure as argument to the modifier object. The `Scanner` class from our example is derived from the `Modifier_base` and implements the `operator()` member function, where it can access the internal representation.

The achievement here is that the `delegate()` member function of the `Polyhedron` can verify the validity of the internal representation after the `operator()` member

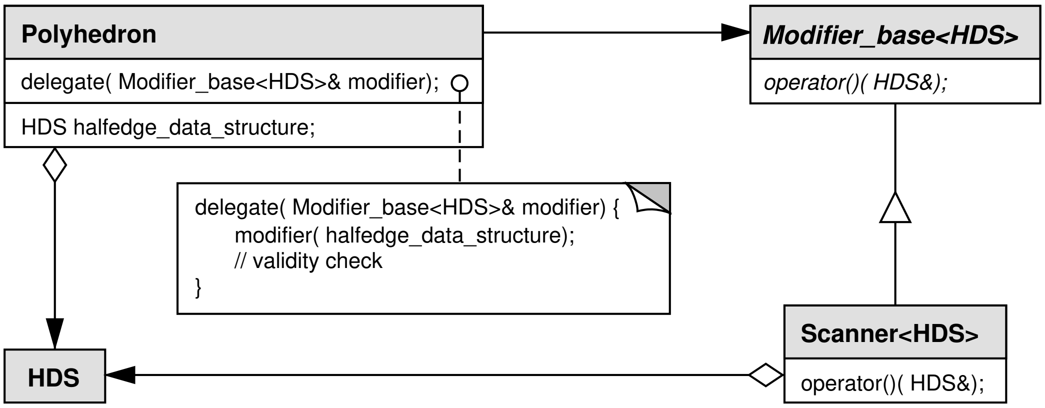


Figure 4.10: Class diagram illustrating the safe access to the internal representation of a polyhedron.

function returns from execution. The `Scanner` class is in charge of returning only with a valid representation, even in the case of a failure. From database systems this approach is also known as transactions; either the modifier succeeds or the modifier fails and is supposed to clean up. The special task accomplished by the `Scanner` (only creation of new items) enables us to implement the transaction scheme efficiently. A simple rollback function removes all items created so far in the case of a failure. In general, the rollback would be more costly.

4.9 Conclusion

We have presented a design framework for combinatorial data structures, such as polyhedral surfaces and planar-maps. It can be extended to model the topology of curved-surfaces and can be applied to other combinatorial data structures, such as triangle-based structures for triangulations.

A proper definition of the modeling space for polyhedral surfaces has been given the strictness of which has been proven useful in our applications. Various suitable edge-based data structures from the literature have been discussed and the halfedge data structure has been chosen for an implementation. The discussion has revealed several desirable options for the data structure among them the demand for flexibility in the design, especially the tradeoffs between space and time. An example in the CGAL Reference Manual [34] uses a similar idea like the encoding with indices from the quad-edge data structure; it uses a bit instead of a pointer to encode the opposite halfedge.

The generic programming paradigm has led to an easy-to-use and flexible design. We can explore many tradeoffs between time and storage, iterator categories, and modifiability. The solutions given for the design goals from Section 4.4 are in the same order:

1. The actual storage organization of the item types can be easily changed by selecting an appropriate halfedge data structure provided in the library.

2. The actually provided incidence relations are selected by the user with the kind of item types used. If the predefined item types from the library are not sufficient, new incidences are easily added, similar to the addition of other attributes. The optional incidences are supported by the halfedge data structure and the polyhedron. Compile-time tags are used to implement this support.

3. Further data and functions can be easily added to the item types by derivation of predefined item types and replacing them in the items class. The coherent type system with its use of templates decouples the item types, such that, for example, adding color to a facet does not imply that the vertex type or the halfedge type needs to be changed as well.

4. The data structure provides an easy-to-use high-level interface based on appropriate concepts, e.g., Euler operators, iterators, and circulators. These concepts support the optional incidences of the item types based on the compile-time tags. Furthermore, the integrity of the representation is maintained, but a protected access to the internal representation is granted for special function objects derived from the base class `Modifier_base`.

5. The management of connected components and containment relations is separated into its own layer, for example the topological map.

6. The concept of circulators supports circular sequences of edges around a vertex or around a facet. The concept is well integrated into the framework of STL.

7. The concept of handles has been introduced. Internally the handles are defined as the iterators of the container classes. Wherever handles are required as arguments, iterators or circulators can be used as well.

The design is still open to incorporate other techniques as well, such as runtime flexibility where appropriate or additional template parameters. We expect a continuation of this approach in CGAL.

Chapter 5

Contour-Edge Analysis for Polyhedron Projections

Given a polyhedron in 3-space and a view point, an edge of the polyhedron is called *contour edge*, if one of the two incident facets is directed towards the view point, and the other incident facet is directed away from the view point, see Figure 5.1 for an illustration. Algorithms on polyhedra (or polyhedral scenes) can exploit the fact that the number of contour edges is usually much smaller than the overall number of edges. This was first pointed out by Appel [10] when describing an object-space hidden-surface removal algorithm.

The main goal of this chapter is to provide evidence for and quantify the claim that the number of contour edges is small in many situations. First we make an asymptotic analysis of polyhedral approximations of a sphere with Hausdorff distance ε. While the required number of edges for such an approximation grows like $\Theta(1/\varepsilon)$, the number of contour edges in a random *orthogonal* projection is $\Theta(1/\sqrt{\varepsilon}\,)$; here and below *random* means that the projection vector is uniformly distributed on the unit sphere in 3-space. We believe that this 'square-root' behavior of the number of contour edges as compared to the overall number of edges appears in general for approximations (of increasing quality) of smooth (and not necessarily convex) bodies.

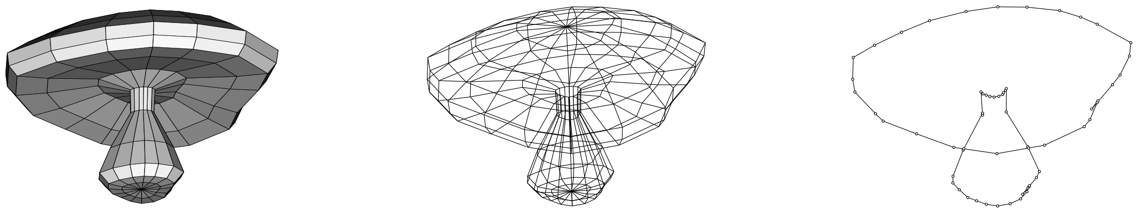


Figure 5.1: An example of contour edges: the hidden-surface rendering of a mushroom model, all projected edges, and the projected contour edges.

The second part of this chapter reports the results of an experimental study of a number of polyhedral objects from several application areas. We analyze the expected number of contour edges, $\overline{n_c}$, and the expected number of intersections of contour edges in a projection, $\overline{int_c}$, (a quantity relevant for line sweep algorithms). We compare these numbers with the number of edges, n, and the expected number of intersections of all edges in a projection, $\overline{int}$. For the 'mushroom' example as depicted in Figure 5.1, these values[1] are $n = 464$, $\overline{n_c} = 54$, $\overline{int} = 450$, $\overline{int_c} = 4$. Instead of sampling from and averaging over a certain number of random directions, we describe how to compute these expected quantities directly.

From the examples we conclude that, indeed, the number of contour edges is small and the number of intersections of contour edges appears to be even more favorable. The latter is particularly interesting for object-space methods based on the sweep-line algorithm, see Chapter 6.

This is joint work with Emo Welzl [109].

5.1 Contour Edges and Probabilistic Analysis

We consider polyhedral surfaces P given by their vertices V, edges E, and facets F, together with their incidences and geometry information $V \mapsto \mathbb{R}^3$. We follow Definition 4.1.3 with its extension for border edges, i.e., an edge is incident to one or two facets. For each facet f we define its normal vector $\mathfrak{n}_f$ such that the direction of $\mathfrak{n}_f$ is consistent with the orientation of f following the right-hand rule.

Definition 5.1.1 (contour edge). For a given viewing direction d we make the following definitions:

1. A facet f is a *front facet* if $\mathfrak{n}_f \cdot d < 0$. Otherwise it is a *back facet.*
2. An edge e is a *contour edge* if it is a border edge or if it is incident to both a front facet and a back facet.
3. An edge e is a *front edge* if it is a contour edge or if it is incident to two front facets.

The intuitive meanings of front facet and back facet assume that the polyhedral surface P encloses a solid, and the normal vectors associated with the facets point outside this solid. Although this intuition is not valid in our setting with a general polyhedral surface with boundaries, we want to distinguish between the inner (solid) side of a facet, and the outer side of a facet (where its normal vector points). The

[1] Throughout this chapter, expectations are rounded to the next integer.

definition of a contour edge is sensitive to the choice of the orientation of a surface only if a facet normal is orthogonal to the viewing direction.

We give an alternative definition of contour edges that is independent of surface normal vectors. The plane h spanned by the viewing direction d and the edge e defines two open halfspaces. The edge e is a contour edge if all incident facets extend in one halfspace and the other halfspace remains empty. This definition is applicable to non-orientable surfaces and to non-manifolds. The degenerate case of facets extending in h instead of one of the halfspaces still needs to be resolved. We continue with the definition for contour edges from above.

We want to analyze the expected number of contour edges of P with respect to a random viewing direction. Recall once more that a random viewing direction is a vector to a point uniformly distributed on the unit sphere in 3-space. A random projection is an orthogonal projection in such a random viewing direction. Because of linearity of expectation, it suffices to analyze for each edge e independently the probability that it is a contour edge. Clearly, for a border edge this probability is one.

Consider a sphere of infinitesimal radius centered at an interior point of an edge e with two incident facets f' and f''. Intersect the sphere with the wedge defined by the facets. (The facets define two complimentary wedges, and we refer to that one enclosing the smaller angle, a possible tie in case of coplanar facets broken arbitrarily). Let α_e, the normalized solid angle, be the ratio between the surface area of this intersection and the whole sphere itself. α_e can be computed as

$$\alpha_e = \frac{1}{2\pi} \arccos \frac{-\mathfrak{n}_{f'} \cdot \mathfrak{n}_{f''}}{|\mathfrak{n}_{f'}||\mathfrak{n}_{f''}|}.$$

We observe that α_e is the probability that both facets incident to e are front facets, and similarly, it is the probability that both facets are back facets. We conclude that

$$\Pr[e \text{ is contour edge}] = 1 - 2\alpha_e \ ,$$

and linearity of expectations yields

Lemma 5.1.2. *The expected number $\overline{n_c}$ of contour edges with respect to a random viewing direction is*

$$\overline{n_c} = \sum_{e \in E} 1 - 2\alpha_e \ .$$

We want to demonstrate in a concrete setting that $\overline{n_c}$ is much smaller than the overall number of edges. To this end we consider *convex* polyhedral approximations C of the unit sphere S. We call C an ε-approximation of S, if no point on S has distance larger than ε from C, and no point on C has distance larger than ε from S. This entails that the Hausdorff distance between C and S is upper bounded by ε.

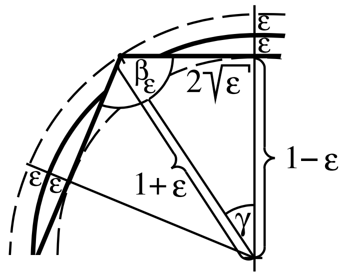


Figure 5.2: The extremal configuration for planes supporting adjacent facets of an ε-approximation.

Lemma 5.1.3. *If e is an edge of a convex ε-approximation C of the unit sphere with $\varepsilon < 1$ then $\alpha_e > \beta_\varepsilon/2\pi$, where β_ε is*

$$\beta_\varepsilon = \pi - 2\arctan\frac{2\sqrt{\varepsilon}}{1-\varepsilon}$$

in radians. Moreover, $1 - 2\alpha_e < \frac{4\sqrt{\varepsilon}}{\pi(1-\varepsilon)} = \frac{4}{\pi}\sqrt{\varepsilon} + O(\varepsilon^{3/2})$.

Proof. We exploit the facts that the supporting planes of the facets incident to an edge e must have distance at least $1-\varepsilon$ from the center of S (because of the approximation property and convexity of C), and that they must have a common point in the ball of radius $1+\varepsilon$ around the center of S (because their intersection carries an edge of C which must not have distance larger than ε from S). The extremal configuration, which allows the smallest possible angle β_ε between two such planes, is depicted in Figure 5.2. With the notation from that figure we have $\frac{\beta_\varepsilon}{2} = \frac{\pi}{2} - \gamma$ where $\tan\gamma = \frac{2\sqrt{\varepsilon}}{1-\varepsilon}$, β_ε and γ angles in radians. For the estimate of $1-2\alpha_e$ we use $\arctan x < x$ for $x > 0$. $\square$

Lemma 5.1.4. *Let n be the number of edges of a convex ε-approximation of the unit sphere, and let $\overline{n_c}$ be the expected number of contour edges of this approximation. Then $\overline{n_c} = O(\sqrt{\varepsilon}\, n)$.*

Proof. For the set E of edges of an ε-approximation, we have $\overline{n_c} = \sum_{e\in E} 1 - 2\alpha_e = O(\sqrt{\varepsilon}\,)|E|$ due to Lemmas 5.1.2 and 5.1.3. $\square$

The number of edges required for an approximation of the unit sphere with Hausdorff distance ε is known to be $\Theta(1/\varepsilon)$ (see [89]). We conclude that

Theorem 5.1.5. *Let n be the number of edges of an optimal convex ε-approximation of the unit sphere and $\overline{n_c}$ the expected number of contour edges. Then we have $\overline{n_c} = \Theta(\sqrt{n}\,)$.*

Proof. Since $n = \Theta(1/\varepsilon)$, we have $\overline{n_c} = O(1/\sqrt{\varepsilon}\,)$ by Lemma 5.1.4. Simple considerations analogous to the proof of Lemma 5.1.3 show that every orthogonal projection of an ε-approximation must have $\Omega(1/\sqrt{\varepsilon}\,)$ edges. $\square$

The asymptotic bounds obtained can be generalized to convex approximations of convex bodies of bounded curvature.

Here, and for the remainder of this chapter, we analyze orthogonal projections (from a view point at infinity), simply because here a natural distribution of projections exists. Note, however, that it may very well be argued that orthogonal projections tend to maximize the number of contour edges. (Consider, for example, again polyhedral approximations of the unit sphere.)

5.2 Computation of Expected Number of Intersections

In the previous section we have seen how to compute $\overline{n_c}$. The complexity of the arrangement of the contour edges in the projection plane is determined by the number of projected contour edges and by the number of their intersections. For the sake of comparison and as a building component we present first how to compute the expected number $\overline{int}$ of such intersections of all edges (not just contour edges) for random orthogonal projections. Then we condition on the event that the edges must be simultaneously contour edges while intersecting in the projection. This will be the expected number $\overline{int_c}$ of contour edge intersections. A known heuristic for hidden-surface removal deletes all edges between two back facets (back-face culling) before computing the visibility map since these edges can never be seen (at least, this is true for closed surfaces enclosing a solid). Only front edges remain for the visibility computation. For the sake of comparison with this approach, we compute the expected number $\overline{int_f}$ of intersections between front edges.

5.2.1 Intersection of Edges in the Projection

It suffices to consider all pairs of edges e' and e'' independently and to evaluate the probability that e' intersects e'' in a random projection. If e' and e'' intersect in space, then this probability is one. So let us assume that this is not the case, and when we refer to an intersection of two edges, we refer implicitly to the intersection of the projections of the edges in the projection plane.

Let T be the tetrahedron obtained as the convex hull of e' and e''. If e' and e'' are coplanar but disjoint, then they intersect with probability zero. So we may assume that T is a full dimensional simplex. We enumerate the edges of T as e_i, $i = 1, 2, \ldots 6$, so that $e_1 = e'$ and $e_2 = e''$. Vertices are enumerated as v_i, $i = 1, 2, 3, 4$. For the following analysis we refer to this tetrahedron when we talk about angles at edges, and about contour edges.

Observe that edges e_1 and e_2 intersect if and only if none of the two edges is a contour edge of T. That is, we can derive:

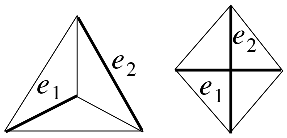


Figure 5.3: Projections of the tetrahedron T.

$$\Pr[e_1 \text{ intersects } e_2] = \Pr[\text{none of } e_1, e_2 \text{ is contour edge}] =$$

$$\frac{1}{2}\left(\Pr[e_1 \text{ is not contour edge}] + \Pr[e_2 \text{ is not contour edge}]\right. \tag{5.1}$$
$$\left. - \Pr[\text{exactly one of } e_1, e_2 \text{ is contour edge}]\right) .$$

The first two probabilities are $2\alpha_{e_1}$ and $2\alpha_{e_2}$, respectively. In order to calculate the third probability, note that exactly one of e_1, e_2 is a contour edge if and only if the tetrahedron projects to a triangle (see Figure 5.3). Equivalently, this means that one of the vertices does not appear on the boundary of the projection.

For the analysis of this event we define the normalized solid angle α_v at a vertex v similarly to the way we defined it for edges. That is, we center an infinitesimally small sphere at vertex v and take the ratio of the surface area in the tetrahedron compared to the full surface area. Consequently, $2\alpha_v$ is the probability that vertex v does not appear on the boundary of the projection. Since these events are disjoint for all vertices, $\sum_{i=1}^{4} 2\alpha_{v_i}$ is the probability that T projects to a triangle. Using the so-called Gram-Sommerville identity [90, 192], we can rewrite this sum as[2]

$$2\sum_{i=1}^{4} \alpha_{v_i} = 2\sum_{i=1}^{6} \alpha_{e_i} - 2 .$$

Plugging this into (5.1), we get

$$\Pr[e_1 \text{ intersects } e_2] = \frac{1}{2}\left(2\alpha_{e_1} + 2\alpha_{e_2} - \left(2\sum_{i=1}^{6} \alpha_{e_i} - 2\right)\right) .$$

Lemma 5.2.1. *Let e_1 and e_2 be two edges which do not lie in a common plane. Then e_1 and e_2 intersect in a random orthogonal projection with probability*

$$\Pr[e_1 \text{ intersects } e_2] = 1 - \sum_{i=3}^{6} \alpha_{e_i} ,$$

where $e_3, e_4, e_5,$ and e_6 denote the remaining edges of the tetrahedron obtained as the convex hull of e_1 and e_2.

[2] To see this directly, note that $\sum_{i=1}^{6} 1 - 2\alpha_{e_i}$ is the expected number of edges in the projection of T, $\sum_{i=1}^{4} 1 - 2\alpha_{v_i}$ is the expected number of vertices, and these two quantities have to be equal.

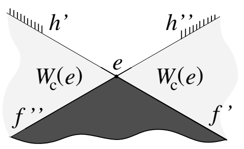


Figure 5.4: The scenario around an edge e is shown when intersected with a plane orthogonal to e: The incident facets f' and f'', the halfspaces h' and h'', and the double wedge $W_c(e)$.

The expected number $\overline{int}$ of intersections of the set of edges E in a random projection is

$$\overline{int} = \sum_{e',e'' \in E, e' \neq e''} \Pr[e' \text{ intersects } e''] \ ,$$

which we can compute easily.

5.2.2 Intersection of Contour Edges

We must take a closer look at the geometric configuration of the two edges and their incident facets in 3-space. Consider an edge e with two incident facets f' and f''. Let h' be the positive open halfspace bounded by the supporting plane of f', that is, the halfspace where the normal vector $\mathfrak{n}_{f'}$ points to. It can be viewed as the set of view points for which f' is a front facet. Similarly, define h'' for f''. Now let $W_c(e)$ be the symmetric difference of h' and h''. Intuitively speaking, $W_c(e)$ is the set of view points for which e is a contour edge (although we never formally defined a contour edge for a view point, we employ the obvious and natural extension for this illustration.) Observe that $W_c(e)$ is a double wedge (see Figure 5.4). For a border edge e, we define $W_c(e)$ to be the whole space.

If edges e' and e'' are contour edges and intersect in the projection plane, then the preimage of this intersection is a line in space intersecting e' and e'', and it is contained in $W_c(e')$ and $W_c(e'')$. It follows that only $e' \cap W_c(e'')$ can contribute to such an intersection and, in fact, e'' has to be a contour edge whenever the projection of $e' \cap W_c(e'')$ intersects the projection of e''. Because of symmetry, we conclude that e' and e'' intersect as contour edges if and only if $e' \cap W_c(e'')$ and $e'' \cap W_c(e')$ intersect.

Note that $e' \cap W_c(e'')$ need not be connected. It may be empty or consist of one or two line segments. Let S' denote the set of these line segments in $e' \cap W_c(e'')$, and let S'' be the segments in $e'' \cap W_c(e')$. Using this notation we get:

$$\Pr[e' \text{ and } e'' \text{ are contour edges and intersect}] = \sum_{g' \in S', g'' \in S''} \Pr[g' \text{ intersects } g'']$$

and it is now obvious how to calculate the expected number $\overline{int_c}$ of intersections of contour edges by summing these values for all pairs of edges in E.

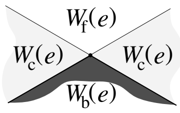


Figure 5.5: The wedges $W_c(e)$, $W_f(e)$, and $W_b(e)$ for an edge e with two incident facets.

5.2.3 Intersection of Front Edges

Some extra care has to be taken, because the definition of a front edge is sensitive to inverting the viewing direction. We denote the two remaining wedges apart from $W_c(e)$ at an edge e: The wedge $W_f(e)$ is the intersection of h' and h'', and $W_b(e)$ is the intersection of the complements of h' and h'' (see Figure 5.5). Intuitively speaking, $W_f(e)$ is the set of view points for which e is a *proper* front edge; that is, e is a front but not a contour edge. $W_b(e)$ is the set of view points where both facets incident to e are back facets. Both sets are defined to be empty for border edges.

Consider two edges e' and e''. The edge e' is subdivided in up to three segments by the regions $W_c(e''), W_f(e'')$, and $W_b(e'')$. Let S' denote the set of these segments and let S'' denote the set of the segments obtained by intersecting e'' with $W_c(e'), W_f(e')$, and $W_b(e')$. The probability that two edges are front edges and intersect is

$$\Pr[e' \text{ and } e'' \text{ are front edges and intersecting}] = \sum_{g' \in S', g'' \in S''} p(g', g'') ,$$

where

$$p(g', g'') = \begin{cases} \Pr[g' \text{ intersects } g''] & \text{for } g' \subseteq W_c(e'') \text{ and } g'' \subseteq W_c(e') \\ 0 & \text{for } g' \subseteq W_f(e'') \text{ and } g'' \subseteq W_f(e') \\ 0 & \text{for } g' \subseteq W_b(e'') \text{ and } g'' \subseteq W_b(e') \\ \frac{1}{2}\Pr[g' \text{ intersects } g''] & \text{otherwise .} \end{cases}$$

To show that this equation holds we look at the preimage of an intersection of the two edges in the projection plane. It is a line l in space intersecting both edges. Define the segment s as the segment on l between both edges. Figure 5.6 indicates all different cases (up to symmetry) of how s can lie relative to the wedges of the involved edges. The situation does only change if s is crossing the boundary of one of the wedges $W_c(e), W_f(e)$, or $W_b(e)$. Hence each pair of segments $g' \in S'$ and $g'' \in S''$ belongs to exactly one case.

Case (a) depicts two intersecting contour edges. For the segment s holds $s \subseteq W_c(e')$ and $s \subseteq W_c(e'')$. It follows that $g'' \subseteq W_c(e')$ and $g' \subseteq W_c(e'')$. Therefore, in this case the probability that e' and e'' intersect as front edges is equal to the probability that g' and g'' intersect.

Case (b) depicts the other possibilities of how two front edges can intersect. At least one of the edges has to be a proper front edge, since otherwise we are in case (a). That is either $g'' \subseteq W_f(e')$ or $g' \subseteq W_f(e'')$. Note that the intersection depends on the orientation of the viewing direction. For the reverse viewing direction we get that

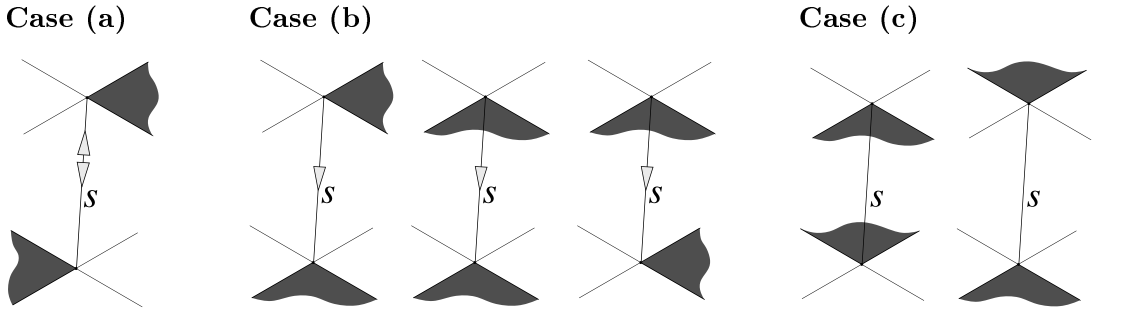


Figure 5.6: The possible cases for segment s between two edges: (a) Both edges are contour edges in both viewing directions. (b) At least one edge is a *proper* front edge for one viewing direction and the other one is a front edge. (c) No viewing direction achieves that both edges are simultaneously front edges.

either $g'' \subseteq W_{\mathrm{b}}(e')$ or $g' \subseteq W_{\mathrm{b}}(e'')$, respectively. Here e' and e'' will not intersect as front edges. Summing up, in this case the probability that e' and e'' intersect as front edges is half the probability that g' and g'' intersect.

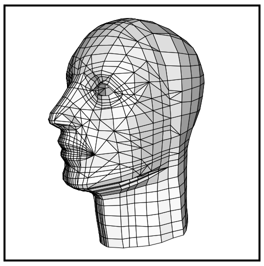
Case (c) depicts the situation where for any orientation of the viewing direction both edges cannot be front edges in the projection simultaneously. That is either $g'' \subseteq W_{\mathrm{b}}(e')$ or $g' \subseteq W_{\mathrm{b}}(e'')$. Thus, the probability of this case to contribute an intersection of front edges is zero.

5.3 Expectations Measured on Objects

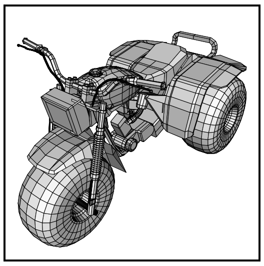
We have collected three-dimensional objects from different application areas to evaluate their properties. The application areas are animation, architecture, terrain data from geographical information systems, mesh generation from CAD models, surface reconstruction, medical imaging, molecular modeling, and mathematics. We count the number $n = |E|$ of edges and the number $|\partial E|$ of border edges. We compute the the expected number $\overline{n_{\mathrm{c}}}$ of contour edges where the border edges are included by definition. We compute the expected number $\overline{int}$ of intersections of all edges, $\overline{int_{\mathrm{c}}}$ of intersections of contour edges and $\overline{int_{\mathrm{f}}}$ of intersections of front edges.

In Figure 5.7 `head` is an example of an approximated curved surface in computer graphics. `honda` has the worst ratio of contour edges to edges under these examples due to the technical details. `beethoven` and `general` are handmade models for animation purposes. Details, such as the faces and medals, are modeled in great detail, other parts, such as the dress, are not. For example, `general` consists of 118 components (hardly visible in the small picture). The details contribute a lot to the contour edges.

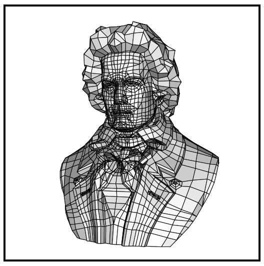
As for the intersections, observe that even $\overline{int_{\mathrm{f}}}$ is a factor 5 to 10 larger than $\overline{int_{\mathrm{c}}}$. The overall impression is that n seems to dominate all other quantities (apart from $\overline{int}$) in most examples.



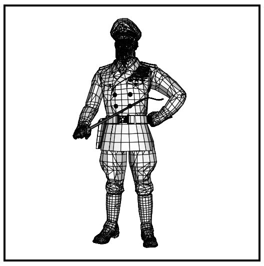
head



honda



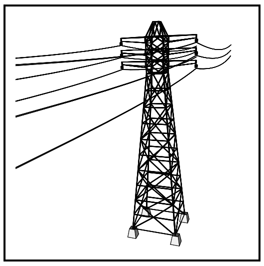
beethoven



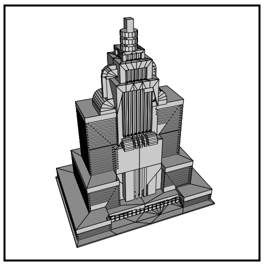
general

filename	n	$\lvert\partial\mathbb{E}\rvert$	$\overline{n_{\mathrm{c}}}$	$\%\frac{\overline{n_{\mathrm{c}}}}{n}$	$\overline{int}$	$\overline{int_{\mathrm{f}}}$	$\overline{int_{\mathrm{c}}}$	$\%\frac{\overline{int_{\mathrm{c}}}}{\overline{int}}$
`head`	3106	58	203	6	2950	456	64	2
`honda`	13875	574	2998	21	36970	11918	2138	6
`beethoven`	5461	268	1003	18	9415	2983	617	7
`general`	25858	6	4350	16	93285	27014	3094	3

Figure 5.7: Examples from computer graphics. `head` is from the `Geomview` distribution, `honda`, `beethoven`, and `general` are from Viewpoint [187].

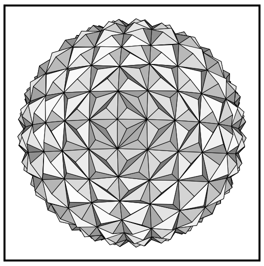


powerlns



skyscrpr

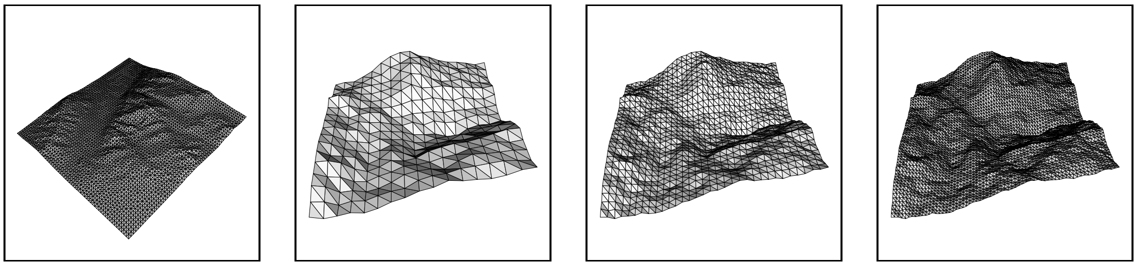
pagoda



epcot3

filename	n	$\lvert\partial\mathbb{E}\rvert$	$\overline{n_{\mathrm{c}}}$	$\%\frac{\overline{n_{\mathrm{c}}}}{n}$	$\overline{int}$	$\overline{int_{\mathrm{f}}}$	$\overline{int_{\mathrm{c}}}$	$\%\frac{\overline{int_{\mathrm{c}}}}{\overline{int}}$
`powerlns`	9214	0	2500	27	45162	17350	4530	10
`skyscrpr`	3711	0	1230	33	23462	10574	3796	16
`pagoda`	44137	4374	16971	38	274119	131333	50317	18
`epcot3`	2304	0	928	40	5459	2594	1585	29

Figure 5.8: Architecture examples. `powerlns` and `skyscrpr` are from Avalon [13], `pagoda` is from Viewpoint [187], `epcot3` is from the `Geomview` distributionarchitecture.



kground terrain1 terrain2 terrain3

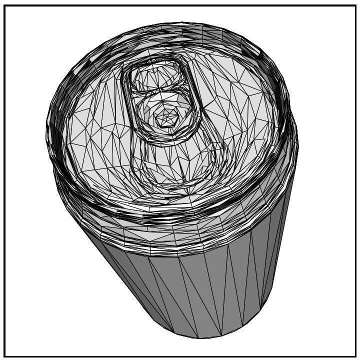
filename	n	$\lvert\partial\mathbb{E}\rvert$	$\overline{n_c}$	$\%\frac{\overline{n_c}}{n}$	$\overline{int}$	$\overline{int_f}$	$\overline{int_c}$	$\%\frac{\overline{int_c}}{\overline{int}}$
kground	10920	240	657	6	7604	2579	369	5
terrain1	800	64	130	16	623	230	49	8
terrain2	3136	128	371	11	3234	1124	201	6
terrain3	12416	256	1139	9	15807	5257	811	5

Figure 5.9: Terrain data examples for geographical information systems. `kground` is from Avalon [13], the terrain data sets are fractal terrains.

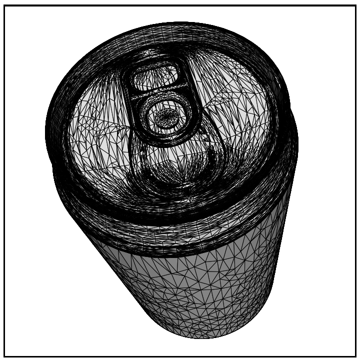
The bad contour edge ratios of the examples in Figure 5.8 reveal that architecture might not be a good candidate for the contour edge approach. It is perhaps counter-intuitive to see that `powerlns` perform better than `pagoda`. This is a consequence of the quality of representation of these models. `powerlns` consists of 20 components and has no border edges. `pagoda` had originally 1086 edges with more than two facets incident. In order to prepare the data set for our analysis we have chosen to trivially break these edges apart resulting in the 4374 border edges. They count as contour edges for all viewing directions. The number of components is 1573 for the `pagoda`. `epcot3` is our worst case example both with respect to contour edges and contour edge intersections.

The terrain data in Figure 5.9 perform quite well in the contour edge analysis. The terrains 1 to 3 are approximations of increasing density of the same terrain (defined by the limit of a fractal terrain generating process). We observe the decreasing ratio of contour edges with increasing approximation quality. Under the assumption that $\overline{n_c} = c_1\sqrt{n}$, we can compute the coefficient c_1 to be 4.5, 6.6, and 10.2 for `terrain` 1, 2, and 3, respectively. Further examples for terrains can be found in Section 6.10.

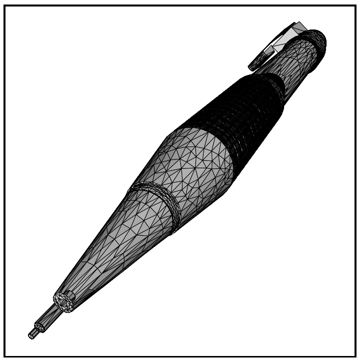
The first three examples in Figure 5.10 were generated by a mesh generation algorithm for objects with parametric facet representations [114]. The number in the filename denotes the tolerated error bound in the approximation. The coefficient in $c_1\sqrt{n}$ is here 21.7 and 30 for `can 0.05` and `can 0.005`, respectively. `fandisk1` is an optimized mesh from surface reconstruction [101]. From that, a piecewise smooth surface has been computed, and `fandisk3` is a piecewise linear approximation [100]. The coefficient c_1 is 3.5 and 3.0, respectively. `bunny` is a data set from a 3d-scanner. It has been used in [62, 99].



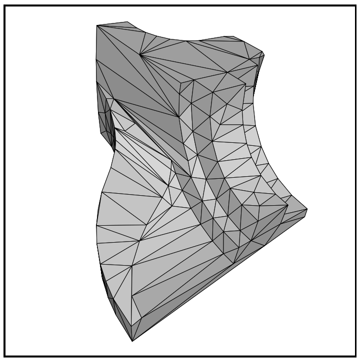
`can 0.05`



`can 0.005`

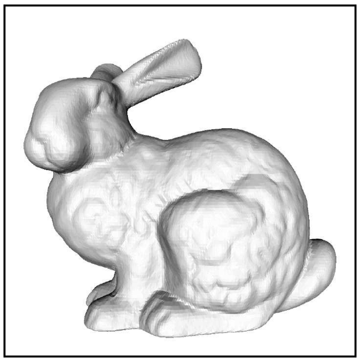


`pen 0.05`



`fandisk1`

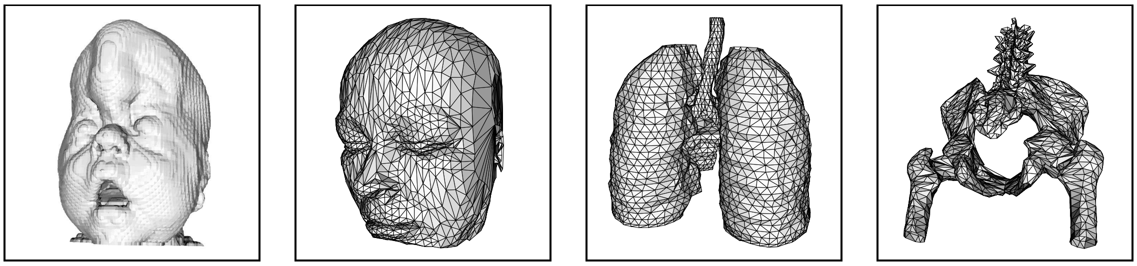
`fandisk2`



`bunny`

filename	n	$\lvert\partial\mathbb{E}\rvert$	$\overline{n_{\mathrm{c}}}$	$\%\frac{\overline{n_{\mathrm{c}}}}{n}$	$\overline{int}$	$\overline{int_{\mathrm{f}}}$	$\overline{int_{\mathrm{c}}}$	$\%\frac{\overline{int_{\mathrm{c}}}}{\overline{int}}$
`can 0.05`	8434	566	1994	23	42860	16526	5044	12
`can 0.005`	75218	1357	8216	10	390913	106196	16215	4
`pen 0.05`	55687	650	4356	7	619367	146135	8495	1
`fandisk1`	555	0	82	14	1226	198	11	1
`fandisk2`	19419	0	423	2	40836	3924	35	0
`bunny`	104310	201	3527	3	252948	36499	2122	1

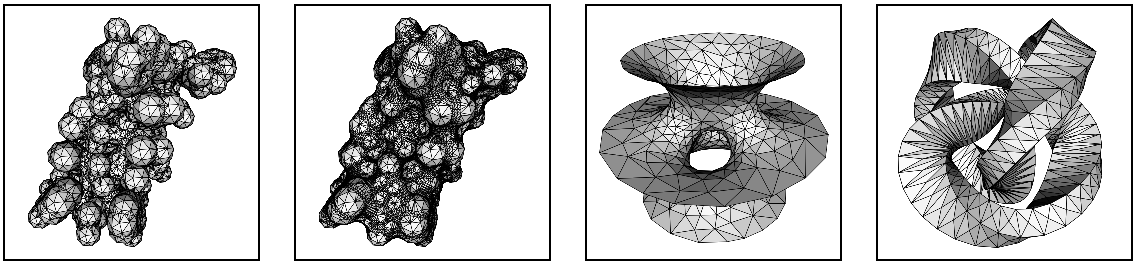
Figure 5.10: Examples from mesh generation from CAD models and surface reconstruction. `can` and `pen` are from Tobias Hüttner and Reinhard Klein, WSI/GRIS, Universität Tübingen, Germany. `fandisk` and `bunny` are meshes from the Computer Graphics Group, University of Washington.



baby head2 lung pelvis

filename	n	$\lvert\partial\mathbb{E}\rvert$	$\overline{n_c}$	$\%\frac{\overline{n_c}}{n}$	$\overline{int}$	$\overline{int_f}$	$\overline{int_c}$	$\%\frac{\overline{int_c}}{\overline{int}}$
baby	66616	332	5375	8	70654	14620	2227	3
head2	4481	238	678	15	11704	3051	798	7
lung	9200	172	1166	12	34527	8843	846	2
pelvis	6773	109	1065	15	31283	8991	1261	4

Figure 5.11: Examples from medical imaging. `baby` was supplied by Herve Delingette, Project Epidaure, INRIA-Sophia-Antipolis, France. `head2`, `lung`, and `pelvis` were supplied by Barbara Wolfers, FU Berlin, Germany (reconstructed from data supplied by INRIA, Sophia Antipolis and Konrad-Zuse-Zentrum für Rechentechnik, Berlin.)



mol1 mol2 hypersheet tre_twist

filename	n	$\lvert\partial\mathbb{E}\rvert$	$\overline{n_c}$	$\%\frac{\overline{n_c}}{n}$	$\overline{int}$	$\overline{int_f}$	$\overline{int_c}$	$\%\frac{\overline{int_c}}{\overline{int}}$
mol1	37290	0	4567	12	239187	58720	1832	1
mol2	74163	0	3772	5	306223	64585	5980	2
hypersheet	1407	63	154	10	2507	538	106	4
tre_twist	2400	0	375	15	13238	3313	490	4

Figure 5.12: Examples from molecular modeling and mathematics. `mol1` and `mol2` were supplied by Nataraj Akkiraju and Herbert Edelsbrunner, University of Illinois at Urbana-Champaign. `hypersheet` is from the Computer Graphics Group, University of Washington. `tre_twist` is from Avalon [13].

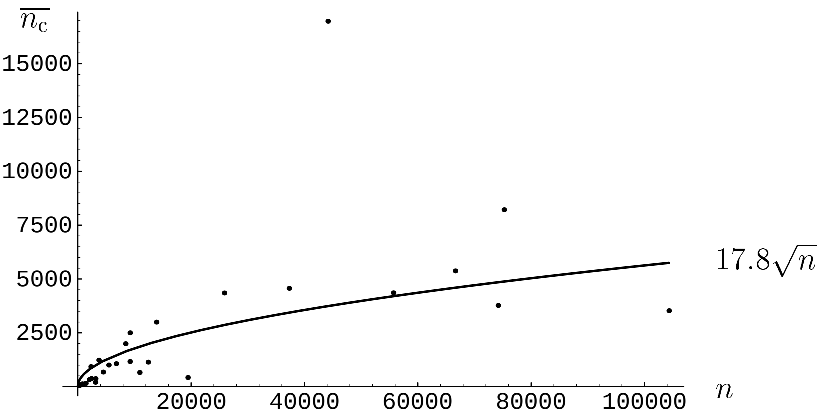


Figure 5.13: All values for expected contour edges $\overline{n_c}$ drawn in one plot.

`baby` in Figure 5.11 is generated with a marching cube algorithm. Although this algorithm has produced a remarkable band structure in the reconstruction, the contour edge ratio is quite good. The reconstruction examples `head2`, `lung`, and `pelvis` from [193] have a significantly lower resolution and, therefore, medium contour edge ratios.

`mol1` in Figure 5.12 is a relative coarse approximation of the van der Waals surface of Gramicidin A, a molecule of about 300 atoms used as an antibiotic since a long time ago. `mol2` is the molecular surface [162], also known as Conolly surface, of the same molecule. It is obtained from the van der Waals surface by introducing so-called reentrant surface patches. Cavities are removed or flattened as can be seen from the silhouette of `mol1` and `mol2`. The genus of `mol1`, which is 37, is lowered to 1 for `mol2`. Therefore, not only the contour edge ratio is better for `mol2`, but also the absolute number of contour edges is smaller. `hypersheet` is again a reconstruction taken from [99]. `tre_twist` realizes surprisingly high values $\overline{int}$ and $\overline{int_f}$ for such a small object.

Figure 5.13 plots all values for the expected number $\overline{n_c}$ of contour edges that are mentioned in the table above. The absolute maximum is 16971 and is achieved by the `pagoda` in figure 5.8. The overall average percentage of contour edges is 15%. Ignoring the obvious runaway `pagoda`, we computed a least-squares fitting curve $c_1\sqrt{n}$ with $c_1 = 17.8$ which is also depicted in Figure 5.13. If we choose $c_1\sqrt{n} + c_2 n$ as the fitting curve, the linear term turns out to be small with $c_2 = 0.0056$, while $c_1 = 16.5$.

5.4 Conclusion

The asymptotic analysis of polyhedral approximations of a sphere indicate that the expected number of contour edges is small for sufficiently dense polyhedral approximations of smooth surfaces. The experiments confirm this statement: Aside from the obvious outliers from architecture with orthogonal angles and from over-detailed objects, the examples have an average percentage of expected contour edges of 15%. As for the intersections, the trivial optimization of back-face culling would still result in five to ten times more expected intersections $\overline{int}_{\mathrm{f}}$ in the projection than $\overline{int}_{\mathrm{c}}$ for contour edges. The overall impression is that in most examples n seems to dominate all other quantities (apart from $\overline{int}$). This makes algorithms based in the sweep-line paradigm despite their worst case behavior quite promising. We report on a hidden-surface removal algorithm based on these ideas in the next chapter.

Chapter 6

Object-Space Hidden-Surface Removal Based on Contour Edges

In 1967 Appel described the use of contour edges in an object-space hidden-line removal algorithm [10]. We recall our Definition 5.1.1 of contour edges: An edge is a contour edge if it is a border edge or if it is incident to both a front facet and a back facet. Considering the projection of the edges and facets into the viewing plane, the visibility of an projected edge can only change at a crossing with a projected contour edge, i.e., exactly if a visible edge intersects a projected contour edge that is nearer to the viewer. The facet incident to the contour edge blocks the visibility of the other edge going behind the projected contour edge. Similarly, the visibility of projected facets also changes only at projected contour edges.

Contour edges had also been used for object-space hidden-surface removal algorithms [78, 171]. Both describe sweep-line algorithms that compute the visibility map in a single pass. The information about contour edges is only used as a classification criterion at intersections of projected edges in order to determine visibility without doing comparably expensive depth comparisons.

In our approach we extend the ideas found in [78, 171] to a three-pass algorithm (see Figure 6.1 for an example of the three steps): First, we extract the contour edges and project them into the viewing plane. Second, we compute the visible part of the projected contour edges. They form a nice partition of the viewing plane. Third, we fill each region in this partition with the visible facets. Since the visibility can only change at contour edges, we only need to know one visible facet for each region to fill the region by traversing adjacent facets. Facets need to be clipped at the boundary of the region to obtain the final subdivision of the viewing plane.

Algorithm HIDDENSURFACEREMOVAL(P, M)
Input. A polyhedral surface P and a viewing transformation M.
Output. The visibility map of P under the transformation M.
1. $C \leftarrow$ EXTRACTCONTOUREDGES(P, M)

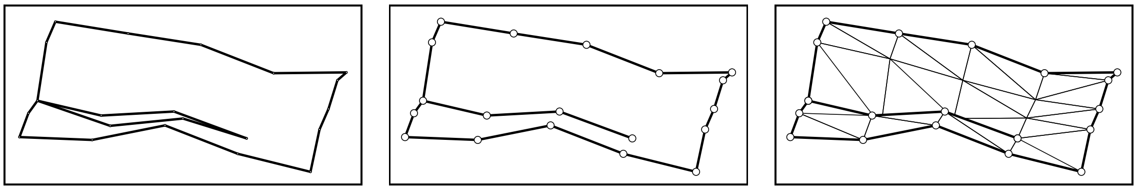


Figure 6.1: Illustration of the three steps in the algorithm HIDDENSURFACEREMOVAL using a small synthetic terrain as an example: (i) In step 1 the contour edges are extracted. Contour edges are drawn bold in this example. (ii) In step 2 the planar map of the projections of the visible contour edges is formed. (iii) In step 3 the region of the planar map is filled with the visible facets. Note that some of them are clipped at the ridge.

2. *planar_map* ← VISIBILITYMAPOFCONTOURS(C, P)
3. *planar_map* ← FILLREGIONSWITHFACETS(*planar_map*, P)
4. **return** *planar_map*.

Let n be the number of edges, n_c the number of contour edges, int_c the number of intersections between the projected contour edges, and k the complexity of the resulting visibility map, e.g., its number of vertices. The running time T of HIDDENSURFACEREMOVAL is the sum of three terms, $T = E(n, n_c) + V(n, n_c, int_c) + F(k)$, where $E(n, n_c)$ denotes the time for extracting the contour edges in step 1, $V(n, n_c, int_c)$ denotes the time for computing the planar map of the projections of the visible contour edges in step 2 which is the core part of the hidden-surface removal problem, and $F(k)$ denotes the time for filling the planar map with all visible facets to obtain the visibility map in step 3. Linearity of expectations will allow us to analyze the expected running time of this algorithm as the sum of the expected running times of its three parts.

The extraction of contour edges $E(n, n_c)$ is trivial in $O(n)$. From the experiments in the previous chapter we know that the number of edges might well be the dominating factor compared to the number of contour edges n_c or their intersections in the viewing plane int_c. Therefore, a view independent preprocessing is worth investigating. The preprocessing is assumed to be off-line and may be computationally expensive. Its result is a data structure DS that allows on-line the fast extraction of the contour edges for a viewing transformation M. Only step one of our algorithm changes.

Algorithm HIDDENSURFACEREMOVAL′(P, DS, M)
Input. A polyhedral surface P, a data structure DS, a viewing transformation M.
Output. The visibility map of P under the transformation M.
1. C ← EXTRACTCONTOUREDGES′(DS, M)
2. *planar_map* ← VISIBILITYMAPOFCONTOURS(C, P)
3. *planar_map* ← FILLREGIONSWITHFACETS(*planar_map*, P)
4. **return** *planar_map*.

The size of the planar map at the end of step 2 gives a lower bound for $V(n, n_c, int_c) = \Omega(n_c + int_c)$. Similarly, the size of the visibility map gives a lower bound for step 2 and step 3 together of $\Omega(k)$. Step 3 can be done with a sweep-line algorithm in $F(k) = O(k \log k)$. Actually, step 3 can be merged with the line sweep algorithms described for step 2 later on in this chapter. Alternatively, step 3 can be done in $F(k) = O(k)$ by projecting the visible contours back onto the polyhedral surface and traversing the incidence graph of the polyhedral surface in the visible regions to enumerate all visible facets. The region traversed during this step is a planar graph with a known embedding and the straightforward details are omitted here.

This work started with a project called CEBaP, for *Contour Edge Based Polyhedron Visualization*, at the Freie Universität Berlin, Germany, in 1995. Participating colleagues were Michael Hoffmann, Andreas Rosenthal, Sven Schönherr, Michael Schutte, and Emo Welzl. A first implementation computed the silhouette of a polyhedron. It is documented in Michael Hoffmann's diploma thesis [97].

In the remainder of this chapter we present related work and a theoretically efficient preprocessing method for the contour edge extraction. We continue with a Bentley-Ottmann sweep-line algorithm to detect segment intersections in the plane which we extend to compute the silhouette of a polyhedral surface and then to compute the visibility map. The event handling in the hidden-surface removal algorithm will be easier for certain classes of polyhedral surfaces, such as terrains represented as triangulated irregular networks (TIN). We continue with the techniques used to implement our algorithms. Among these techniques there is exact integer arithmetic with fixed but sufficient precision based on built-in and efficient number types. As a preprocessing step for the integer arithmetic, the coordinates of the polyhedral surface and the coefficients of the viewing transformation are rounded. We use symbolic perturbation to handle degeneracies. Specifically, we exploit our strict definition of polyhedral surfaces and perturb only the viewing direction. We achieve therefore easily a valid and efficient perturbation scheme. We conclude this chapter with experiences made with our implementations and a discussion about the design choices made.

6.1 Related Work

In a classification of ten hidden-surface removal algorithms Sutherland introduced in 1974 the distinction between image-space and object-space algorithms [179]. An introduction to standard methods for hidden-surface removal can be found in [74], including the well known image-space Z-buffer algorithm of Catmull [39], the painter's algorithm of Newell, Newell, and Sancha [150] which requires a depth order of the objects[1], and the related binary space-partitioning (BSP) tree methods of Fuchs,

[1] A cycle of overlapping polygons can prevent a depth order. The suggestion in textbooks, e.g., [74, pg.674], usually is to split conflicting polygons mutually at the supporting plane of the

Kedem, and Naylor [77, 76]. A more recent survey in 1994 by Dorward gives an extensive overview of object-space hidden-surface removal algorithms [60], and a book by de Berg in 1993 also contains ample discussion [51]. A survey of visualization methods for terrains represented as TIN is given in [52]. Methods to obtain TIN from other terrain representations can be found in [182].

The worst-case lower bound for the object-space hidden-surface removal is $\Omega(n^2)$, since the output structure can be this large. An example is a grid like arrangement of thin objects [167]. This lower bound holds also if the input is a terrain represented as a TIN, where the example consists of a set of spikes in front of a set of long horizontal triangles [52]. The object-space hidden-surface removal problem is solved optimally, i.e., it is in $\Theta(n^2)$ [128]. Similarly, the hidden-line removal problem is in the same class of $\Theta(n^2)$ [58]. Both algorithms have the disadvantage that they need quadratic time and quadratic space in all cases, not just in the worst case.

Depending on the input data, the algorithms for hidden-line removal in [155, 153] can be faster than the worst case. They are based on a sweep-line algorithm and have running times $O((n+k)\log^2 n)$ and $O((n+k)\log n)$ respectively, where k is the number of edge intersections of the projected edges. The running times are usually better than the worst-case bound, but for the worst-case examples they are not optimal. Furthermore, the space requirement of $O((n+k)\log n)$ for the algorithm in [153] is not practical.

Output-sensitive algorithms express their running times in terms of the input size *and* the output size. De Berg, Halperin, Overmars, Snoeyink, and van Kreveld have presented such an output-sensitive object-space algorithm that runs in $O(n^{1+\varepsilon}\sqrt{k})$ time and space, $\varepsilon > 0$ and k denoting output size [53]. The result is of theoretical interest but rather complicated and probably not practical to implement. Mulmuley published a randomized, quasi-output sensitive algorithm [142]. It is not strictly output sensitive, since the work spent for an average facet depends partially also on the number of facets obscuring it from the viewpoint. The randomized incremental algorithm by Boissonnat and Dobrindt is quasi-output sensitive as well [25].

A polyhedral terrain can be defined as a polyhedral surface that forms the graph of a function $z = F(x, y)$, i.e., each value in the xy-plane has a unique height z. This representation of terrains is also called *triangulated irregular network* (TIN) if the facets of the polyhedral surface are triangulated. For terrains, consisting of n triangles, it is particularly easy to compute a depth order in $O(n)$ [52]. This depth order leads to the easy and efficient image-space painter's algorithm. The depth order is also used by Reif and Sen [161] to obtain an object-space output-sensitive algorithm. The triangles are processed front to back and the boundary of the union of all triangles processed so far is maintained. The running time is $O((n+k)\log^2 n)$, which can be improved with fractional cascading to $O((n+k)\log n \log\log n)$ [42], k denoting the output size. Also based on the depth-order, Katz, Overmars, and

other polygon. This is incorrect and does not resolve the conflicting cycle, since the cycle can be large and the two polygons that have been identified might not intersect the supporting plane of the other polygon at all. A correct solution can be found in [150].

Sharir are able to solve the problem in $O(n\alpha(n)\log n + k\log n)$, $\alpha(n)$ denoting the inverse of Ackermanns function [104].

The idea of contour edges was introduced by Appel in 1967 [10]. Hamlin and Gear described 1977 two scan line conversion algorithms. One of these algorithms, the algorithm *cross*, corresponds to a sweep-line algorithm [78]. They made use of contour edges in analyzing intersections of projected edges. Their goal was to avoid expensive depth comparisons between edges. Séquin and Wensley extended this work to a broader class of input geometries including wires [171].

Contour edges have been used in several other application areas. Shadow casting or shadow-volume computation in computer graphics is based on contour edges [50]. Contour edges are important visual features of objects and are explicitly mentioned as quality criterion of adaptive surface simplifications methods [123, 195]. Contour edges can be used in collision detection [30] or image registration [121]. In medical imaging, an X-ray image shows discontinuities at the projections of contour edges.

6.2 Fast Extraction of Contour Edges

We recall our Definition 5.1.1 of contour edges: Facets are classified into front facets and back facets with respect to a viewing transformation M. An edge is a contour edge if it is a border edge or if it is incident to both a front facet and a back facet. The canonical implementation for extracting contour edges in linear time without preprocessing is a plain loop over all edges:

Algorithm ExtractContourEdges(P, M)
Input. A polyhedral surface P and a viewing transformation M.
Output. The set of contour edges of P under the transformation M.
1. **forall** e is edge of P: **if** e is contour edge with respect to M **then** report e.

The result we aim for in a view-independent preprocessing step is a data structure of nearly linear size that allows fast queries with a viewing transformation M. A query should return the set of contour edges with respect to M in time sub-linear in the number of the edges plus linear in the size of the output (the number of contour edges). Border edges are always contour edges and are therefore excluded from further considerations of preprocessing.

We distinguish between orthogonal and perspective projections. For orthogonal projections, the property of being a contour edge is independent of the actual position of this edge in space. It depends only on the normal-vector orientation of its two incident facets.

This problem of contour edge reporting under orthogonal projections can be reduced to the problem of stabbing segments in the plane with a stabbing line (see Figure 6.2). We transform the problem to an arrangement of arcs of great circles

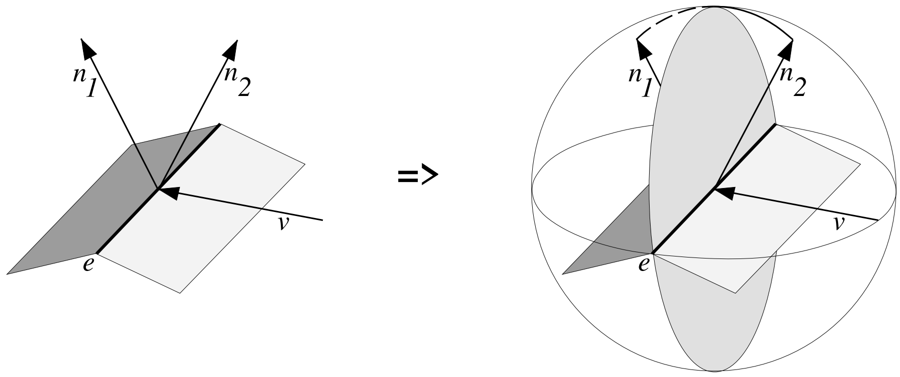


Figure 6.2: The configuration on the left side with a contour edge e, the normal vectors n_1 and n_2 of its incident facets, and a viewing direction v is equivalent to the configuration on the sphere on the right side.

on the unit sphere known as Gaussian diagram [24]: Each facet is represented by the tip of its normalized normal vector centered in the sphere. Each edge is represented by an arc of a great circle. The arc connects on the sphere the two tips of the normal vectors of the two incident facets. Of the two possible arcs on the great circle the shorter arc is taken, ties are broken arbitrarily. The viewing direction v of the orthogonal projection is represented by a great circle on the sphere. This great circle is the intersection between the sphere and the plane that is orthogonal to v and centered in the sphere. With this transformation onto the sphere, an edge e is a contour edge with respect to v if and only if the great circle representing v intersects the arc representing e. A transformation of the sphere in the plane reduces the problem of contour edge reporting to the problem of stabbing segments in the plane with a stabbing line. Edges correspond to segments and the viewing direction corresponds to the stabbing line. Technical details are the handling of degeneracies and that the mapping to the plane may introduce segments through infinity.

Segment stabbing has been addressed in [125] as an example of a multilevel simplex range-searching data structure. The first level allows simplex queries in time $O(n^{1-1/d} + k)$ in dimension d and output size k, which is optimally in dimension two and near optimal in higher dimensions. The first level requires linear space and $O(n \log n)$ preprocessing time [125]. The second level adds a polylog factor to the space and the query time. Preprocessing remains polynomial. We summarize the result in the following corollary.

Corollary 6.2.1 (contour edge reporting: orthogonal projection). *Given a polyhedral surface with n edges, the edges can be preprocessed in a view-independent data-structure of size $O(n \log^2 n)$ in polynomial time, such that for a query with an orthogonal projection all contour edges can be reported in $O(\sqrt{n} \log n + k)$ time, where k is the number of contour edges reported.*

Proof. We apply the transformation to segment stabbing from above and use the results for segment stabbing in [125] including the more precise statement for the polylog factor in dimension two. Segments through infinity can be split into two rays each. Instead of organizing segments and pairs of rays in one data structure, we can use a second, similar data structure for the segments through infinity. For the finite segments, the primary data structure stores all left endpoints of the segments and the secondary data structure stores the right endpoints of the respective canonical subsets. A halfspace query with the stabbing line reports the canonical sets containing all left endpoints in the halfspace left to the stabbing line. For each canonical set, a halfspace query in the secondary data structure reports all right endpoints that lie the right of the query line, thus reporting all regular segments crossing the stabbing line from left to right. Symmetrically, all regular segments crossing from right to left can be reported. For segments through infinity, the secondary data structure is queried for the right endpoint to be on the same side as the left endpoint. □

The solution in [125] is based on hierarchical cuttings and it is considered impractical [127]. Surveys on geometric range searching can be found in [1, 126]. More practical solutions are based on spanning paths with low crossing numbers [44] ($O(\sqrt{n}\log^2 n+k)$ query time for segment stabbing) or conjugation trees [66] ($O(n^{0.695}\log n+k)$ query time for segment stabbing). A practical alternative for the secondary data structure are halfspace range reporting data structures [4, 43]. In dimension two the data structure stores the decomposition of the point set into convex layers [43]. It is of linear size and allows halfspace range reporting in $O(\log n)$ query time using fractional cascading [41]. Altogether, for a solution based on conjugation trees and the convex layer decomposition as secondary structure, the only non-trivial step is the computation of the ham-sandwich cut [65] for the construction of the conjugation tree.

The problem of contour edge reporting under perspective projections can be reduced to the problem of stabbing segments in three-space with a stabbing plane [17]. This reduction is based on a duality between points and planes in dimension three. A plane equation $ax + by + cz + d = 0$ of a primal facet corresponds to a dual point $(a/d, b/d, c/d)$ and vice versa. A primal edge incident to two primal facets corresponds to a dual edge connecting the two dual points. As in the orthogonal projection, the dual edge might be a segment through infinity. The segment corresponds to the path of the dual point if its primal plane rotates around the primal edge until it matches the primal plane of the other incident facet. The segment goes trough infinity if the primal plane rotates through the origin. Using the range reporting structures discussed above [125], we can summarize the result in the following corollary.

Corollary 6.2.2 (contour edge reporting: perspective projections). *Given a polyhedral surface with n edges, the edges can be preprocessed in a view-independent data-structure of size $O(n \operatorname{polylog} n)$ in polynomial time, such that for a query with a perspective projection all contour edges can be reported in $O(n^{2/3} \operatorname{polylog} n + k)$ time, where k is the number of contour edges reported.*

As for practicability, in [17] a balanced aspect ratio tree (BAR-tree) [61] has been used. The authors report a significant performance improvement, but they omit bounds and experimental data. Furthermore, they have investigated the incremental update of the contour edges for a moving view point.

We have investigated the use of a hierarchical triangulation of the sphere as a bucketing structure for the case of orthogonal projections. However, the timing results were discouraging for object sizes below 13000 edges if compared to the version without preprocessing. Furthermore, the time spend for contour edge reporting is a small fraction of the total time of our hidden-surface removal algorithm. Thus, we have not further investigated practical preprocessing data structures.

6.3 Sweep-Line Algorithm to Compute All Segment Intersections

The central part of our algorithm is a sweep-line algorithm in the viewing plane that computes the arrangement with all intersections of the projected contour edges. We have opted for the simple Bentley-Ottmann sweep-line algorithm [21] without special treatment of degeneracies. Degeneracies are resolved at the level of the geometric predicates using symbolic perturbation (see Section 6.9). An alternative would be the explicit handling of all degeneracies (see for example the implementation in the LEDA library [130] or the description in [55]). Even if all degeneracies are handled, the basic sweep-line algorithm to compute all segment intersections remains fairly easy. But the extensions of the sweep-line algorithm for our hidden-surface removal algorithm are much easier if we can ignore degeneracies at this level. The careful design of the degeneracy handling in the sweep-line algorithm cannot be carried over to our extensions.

The sweep-line algorithm moves conceptually a vertical line from left to right in the viewing plane. The algorithm maintains the invariant that all intersections on the left side of the sweep line have been computed. To maintain this invariant, the algorithm only needs to act on discrete *event points*. These are the endpoints of segments and the intersection points between two segments. The events are maintained in a *priority queue* in the increasing order of their x-coordinates. Furthermore, the algorithm maintains a data structure, the *sweep-line status*, for all segments that intersect the current sweep line.

In the following, we describe the interfaces of the priority queue, the sweep-line status, and the geometric operations needed in a geometric traits class. The algorithm is not only supposed to compute the intersections, but with the hidden-surface removal algorithm in mind we augment it with the generation of a planar map representing the arrangement of segments and their intersections. We describe the specific interface of our planar map and conclude this section with the sweep-line algorithm based on these interfaces.

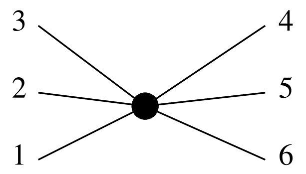


Figure 6.3: The order of segments with a common endpoint in the priority queue.

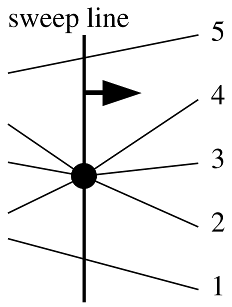


Figure 6.4: The order of segments with a common endpoint in the sweep-line status.

6.3.1 Event Scheduling with a Priority Queue

Event points are classified into three event types. Segments are ordered from left to right. Their left endpoints are called *start events* and their right endpoints are called *stop events*. The intersection point of two segments forms an *intersection event*. Event points store references to the incident segments.

The event points are processed in the order of increasing x-coordinates. The common setting for a line sweep is to require that no two event points have equal x-coordinate. This is also likely to be true for projections of a polyhedral surface (and will be achieved with symbolic perturbation later on), except for projected edges that meet in a common vertex. For this situation we extend the order predicate. If two events have equal x-coordinate, we define stop events to be smaller than start events. If two events have equal x-coordinate and equal event type, we define the event with the segment of steeper slope to be smaller than the other event (see Figure 6.3). Since overlapping segments are excluded as degenerate case, this order defines a total order for the event points. The event-point order is maintained with a priority queue. The interface of the priority queue (pqueue) is summarized in the following table.

signature	semantic
pqueue.push_back(x)	inserts a new item x into pqueue.
pqueue.top()	returns the smallest element of pqueue.
pqueue.pop()	removes the smallest element from pqueue.
pqueue.empty()	returns true if pqueue is empty.

6.3.2 Sweep-Line Status

The sweep-line status maintains all segments intersecting the current sweep line in a balanced search tree. The segments are ordered along increasing y-coordinates of their intersection with the current sweep line. Insertion, deletion and searching is in $O(\log n)$ given n the number of segments. Exchanging two segments is in $O(1)$. Each segment that intersects the current sweep line knows its position in the search tree. This simplifies many accesses to the tree and only the insertion of a new start event needs comparisons at all to locate its segment in the tree. Thus, the comparison object will be used only at coordinates of segment endpoints. Here, equal y-coordinates are excluded as degeneracies except for segments that belong to edges meeting in a common vertex. Considering this case, the sweep line is just moving through this event point and we are interested in the order of the segments to the right of this point. Thus, the steeper segment is defined to be larger than the other segment (see Figure 6.4). The interface of the sweep-line status (sls) is summarized in the following table. Positions in the sweep-line status are bidirectional iterators that allow the traversal of the leaves in amortized constant time.

signature	semantic
sls.insert(s)	inserts a new segment s into sls and returns its position.
sls.insert(i, s)	same, but uses i as a hint for the insert position.
sls.erase(i)	erases the segment at position i in sls.
sls.swap(i,j)	exchange the segment at position i with the segment at position j in sls.
sls.lower_bound(s)	returns the smallest position in sls such that the segment at this position is not smaller than s.

6.3.3 Planar Map as Output Data Structure

The planar map for the output extends the halfedge data structure with management for loops in faces. A face has one outer loop (the usual sequence of halfedges in counterclockwise order on its outer boundary) and a set of inner loops. Each inner loop is a clockwise sequence of halfedges describing a hole in the face. Faces and loops need to be merged quickly during the incremental construction of the planar map in the sweep-line algorithm. A union-find data structure with constant-time find operation and amortized logarithmic-time union operation is appropriate [48]. Therefore, each face and each loop keeps track of the number of incident halfedges.

A set of operations have been designed to construct the planar map during the sweep-line algorithm. The planar map to the left of the sweep line is finished and all edges crossing the sweep line lack a right vertex. These edges are therefore called *antennas*. Outer and inner loops are always represented by unique identifiers. During the sweep, loops are either closed or open. If they are open the loop begins and ends in an antenna. In an intermediate step loops may also contain antennas at inner vertices. Thus, a loop is traversed using the unique loop identifier.

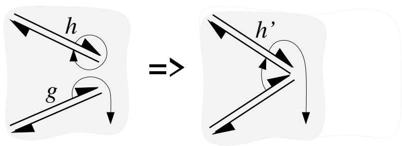
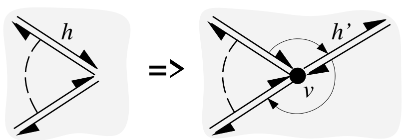
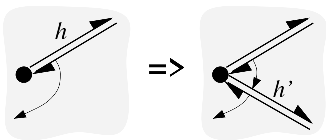
The operations are designed to process one vertex, corresponding to an event point, at a time. Let us assume the general case with edges to the left and edges to the right of the vertex. The edges to the left are antennas of the planar map. They are known with the help of the sweep-line status. These edges and their loops are joined together. Next, a new vertex is allocated for the planar map together with a new antenna for the first edge to the right of the vertex. The other edges to the right are added as antennas, each of them creating a new face.

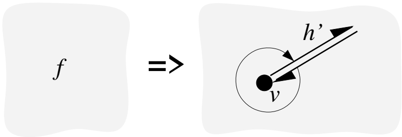
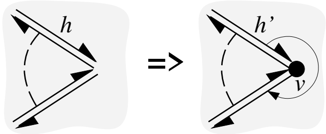
In the case that there are no edges to the right of the vertex, a new vertex is allocated and assigned to the joined antennas. The face above the topmost edge and the face below the bottommost edge are joined. If the faces are already the same, their loops are joined. If the loops are also the same, we have just closed a hole in the current face and no specific update of loops or faces has to done.

In the case that there are no edges to the left of the vertex, a new vertex with a new antenna is created for the first edge to the right of the vertex. The other edges are added as explained above.

The orientation of the edges in the operations is aligned with the order in which the edges appear in the sweep-line algorithm according to the order chosen for the sweep-line status and the priority queue.

The operations of the planar map (pm) used during the sweep-line algorithm are summarized in the following table. The initial configuration of the empty planar map contains one face, the unbounded *outer face*. Basic access operations, such as getting the loop of an edge, are ignored. The timing requirements for these operations are amortized logarithmic in the size of the planar map, except the constant-time access to the outer face. A constant number of these operations can be used per event point in the sweep-line algorithm without changing its asymptotic runtime complexity.

signature	semantic
$f' =$ pm.outer_face()	returns the unbounded outer face of pm.
$h' =$ pm.close_region(h,g) 	connects the antenna h with the antenna g. If the loops of h and g differ, only one of the loops can be an outer loop. If one of the loops is an outer loop, join the other loop with this outer loop, thus keeping the outer loop. Otherwise, if both loops are inner loops and differ, join the smaller loop with the larger loop.
$h' =$ pm.attach_antenna(h,v) 	attaches a new vertex copied from v and an antenna to the vertex denoted by the halfedge h. The loops on both sides of h' are updated.
$h' =$ pm.open_region(h) 	attaches a new antenna to the vertex denoted by the halfedge h. It creates thereby a new face.

signature	semantic
$h' =$ pm.create_antenna(v,f) 	creates a new antenna with a new vertex copied from v in face f. It creates a new inner loop with two edges in f.
$h' =$ pm.join_faces(h,v) 	creates a new vertex copied from v at the tip of h and join the faces h.face() with h.next().face(). If the faces are the same, the respective loops are joined similar to the close_region operation. If the faces differ, at least one edge belongs to an outer loop. If exactly one halfedge belongs to an outer loop, join its face with the other face, thus eliminating the outer loop. If both halfedges belong to outer loops, join the smaller face with the larger face.

6.3.4 Union-Find Operations for Loops and Faces in the Planar Map

In this set of operations for the planar map, we need one union-find data structure for the loops and one for the faces. A generic solution would be a union-find data structure based on trees and path compression with amortized constant cost for find and union [48]. But a simpler version with amortized logarithmic cost for the union is sufficient for our purposes. Its great benefit is that basically all information needed for the union-find is already encoded in the topology of the planar map, whereas the variant with tree and path compression would need additional space and the full topology of the planar map must be recovered later on.

Each halfedge has a pointer to its loop and to its face. The find operation is in constant time. The union of loops in the same face is needed in the close_region operation and in some cases of the join_faces operation. The operation needs to be done in time linear to the size of the smaller loop in order to obtain an amortized logarithmic-time union operation [48]. In the case that both loops are inner loops it is already mentioned in the description in the table. If only one loop is an inner loop it is joined with the outer loop and if the outer loop has smaller size than the inner loop this merging step needs further considerations. It has to change the loop pointers in the halfedges of the (smaller) outer loop to point to the (larger) inner loop. Thereafter, the loop record describing the inner loop can be changed to denote the outer loop while eliminating the record for the outer loop in constant time. One example of managing loop records would be a list, where the first entry denotes the outer loop. Manipulating the link pointers in the list allows to exchange the inner loop record with the outer loop record without changing their place in memory. An alternative would be an additional bit in the loop record to distinguish inner from outer loop records.

The union of faces is needed in the join_faces operation. Similar to the union of loops, the operation can always be done in time linear to the size of the smaller face.

6.3.5 Geometric Traits Class

The geometric traits class for the sweep-line algorithm isolates the geometric predicates from the control flow of the algorithm. This separation also makes the realization of symbolic perturbation easier. The geometric predicates needed for the sweep-line algorithm and the comparison predicates in the priority queue and the sweep-line status are summarized in the following table[2]. All points are either projected image points of the view transformation or intersection points of projected segments. We use the notation $\overline{pq}$ to denote a segment with the image points p and q as endpoints.

signature	semantic
traits.less_x_point_point(p,q)	returns $p_x < q_x$, both image points.
traits.less_x_int_point(p,q)	returns $p_x < q_x$, p an intersection point, q an image point.
traits.less_x_int_int(p,q)	returns $p_x < q_x$, both intersection points.
traits.less_slope(p,q,r,s)	returns slope($\overline{pq}$) < slope($\overline{rs}$), all image points.
traits.compare_y(p,q,r,s,v)	compares the y-coordinate of two segments at their intersection with the sweep line through the event point v. Returns 1 if $\overline{pq}$ is above $\overline{rs}$, 0 if $\overline{pq}$ is equal to $\overline{rs}$, and -1 if $\overline{pq}$ is below $\overline{rs}$.
traits.do_intersect(p,q,r,s,t)	returns true and the intersection point t if $\overline{pq}$ intersects $\overline{rs}$.

6.3.6 Sweep-Line Algorithm

At the beginning, the sweep-line algorithm initializes the priority queue with the start and stop events for all segments. The sweep-line status is initialized with a top and a bottom sentinel. These sentinels are horizontal segments that do not intersect any other segment and are never reported. They simplify the handling of the sweep-line status at both ends. The main loop of the algorithm processes the events from the priority queue. The order of the priority queue guarantees the processing of the events from left to right as long as no new event is inserted with an x-coordinate smaller than the current event. The event handling distinguishes between intersection events and other events.

[2]The call of the predicates in the pseudo code of this chapter is simplified compared to the real C++-code (compare with Section 2.6).

Algorithm SEGMENTINTERSECTIONS(S)
Input. A set S of segments in the plane.
Output. A planar map for the arrangement of segments incl. their intersections.
pm: planar map. *pqueue*: priority queue. *sls*: sweep-line status.
Insert all start and stop events from segments of S into *pqueue*.
Insert top and bottom sentinel into *sls*.
while not *pqueue*.empty() **do**
    event $\leftarrow$ *pqueue*.top(); *pqueue*.pop()
    if type of *event* = intersection event **then**
        HANDLEINTERSECTIONEVENT(*event, sls, pqueue, pm*)
    else
        HANDLESTARTSTOPEVENT(*event, sls, pqueue, pm*)
    endif
end
return *pm*.

6.3.7 Handling of Intersection Event Points

For intersection events at first the positions i and j of the two intersecting segments are located in the sweep-line status. The information stored in the segments is used to get their position without searching. The segment i is directly above segment j before the event point. They change position while passing the event point. Their entries in the sweep-line status are exchanged. Now, i still refers to the segment directly above segment j, but right after the event point. The exchanged segments are tested for intersections with their direct neighbors in the sweep-line status. The planar map is updated. Finally, the priority queue is checked for identical intersection events. We do not test for identical events upon insertion and can use a heap-based implementation for the priority queue. The runtime complexity of the algorithm remains unchanged, since we can charge each entry in the priority queue to the event point where it was inserted. And each event point can only insert a constant number of event points. However, the space complexity can be changed, as detailed later on.

Algorithm HANDLEINTERSECTIONEVENT(*event, sls, pqueue, pm*)
Input. An intersection *event* and the sweep-line data structures.
Output. Updated sweep-line data structures after processing *event*.
$i \leftarrow$ position in *sls* of the upper segment of *event*
$j \leftarrow$ position in *sls* of the lower segment of *event*
$g \leftarrow$ halfedge in *pm* corresponding to j.
$h \leftarrow$ halfedge in *pm* corresponding to i.
sls.swap(i, j)
CHECKINTERSECTION($i+1$, i, *event*, *pqueue*)
CHECKINTERSECTION(j, $j-1$, *event*, *pqueue*)
pm.close_region(h, g)

$h \leftarrow$ *pm*.attach_antenna(h, vertex for current event point)
pm.open_region(h)
Remove all event points identical to *event* from *pqueue*.

The procedure CHECKINTERSECTION tests two segments for an intersection with an x-coordinate to the right of the current event point. If the segments intersect the intersection point is inserted as a new event point into the priority queue. Note that segments with identical endpoints are not intersecting. The segments are oriented from left to right, g is above h at the current event point, left(h) denotes the left endpoint of h, and right(h) denotes the right endpoint.

Algorithm CHECKINTERSECTION(g, h, *event*, *pqueue*)
Input. Segment g above segment h, the current *event*, and *pqueue*.
Output. Updated *pqueue* if the segments intersects on the right side of *event*.
if g and h have no endpoints in common **then**
  if *traits*.do_intersect(left(g), right(g), left(h), right(h), p) **then**
    pqueue.push(intersection event between g and h at point p)
  endif
endif

6.3.8 Handling of Start/Stop Event Points

The procedure HANDLESTARTSTOPEVENT handles at once all start and stop event points that belong to the same shared endpoint. The events are processed in the order defined for the priority queue. That is, stop events, if any, come first and start events, if any, follow. Stop events and start events are sorted with respect to the slope of their segments. Steeper segments come first.

In the first part of this procedure, all stop event points are processed including the first start event point. The position for inserting new starting segments in the sweep-line status is kept in the variable *below* for use in the second part. This variable denotes the position in the sweep-line status below the current event point. It is read from the stop events or, if there is no stop event, it is searched in the sweep-line status. This is the only place where the order predicate of the sweep-line status is actually called. Another variable, *new_h*, stores the edge of the planar map that might be created for the first start event in the first part, again for use in the second part. In the second part of this procedure, the remaining start events are processed.

Algorithm HANDLESTARTSTOPEVENT(*event, sls, pqueue, pm*)
Input. A start or stop *event* and the sweep-line data structures.
Output. Updated sweep-line data structures after processing *event*.
below: position variable in *sls*. *new_h*: halfedge in *pm*.
if type of *event* = stop event **then**
  below $\leftarrow$ (position in *sls* of the segment of *event*) - 1
  $h \leftarrow$ halfedge in *pm* corresponding to the segment of *event*.

```
        sls.erase(below +1)
        while more stop events with same endpoint do
            event ← pqueue.top(); pqueue.pop()
            sls.erase(position in sls of the segment of event)
            g ← halfedge in pm corresponding to the segment of event
            h ← pm.close_region(g, h)
        end
        if type of pqueue.top() = start event and with same endpoint then
            event ← pqueue.top(); pqueue.pop()
            new_h ← pm.attach_antenna(h, vertex for current event point)
        else
            pm.unify_faces(h, vertex for current event point)
        endif
    else (* no stop events, locate in sls *)
        below ← (sls.lower_bound(the segment of event)) −1
        new_h ← pm.create_antenna(vertex for current event point,
                                  face in pm above below)
    endif
    if type of event = start event then
        above ← below +1
        forever
            sls.insert( below, the segment of event)
            if no more start events of same endpoint then break
            event ← pqueue.top(); pqueue.pop()
            new_h ← pm.open_region( new_h)
        end
        CheckIntersection(above, above −1, event, pqueue)
    endif
    CheckIntersection(below +1, below, event, pqueue)
```

6.3.9 Runtime and Space Requirements

This concludes our description of the sweep-line algorithm. The algorithm SegmentIntersections computes a planar map representing the arrangement of a set of n segments in the plane including their intersections in time $O((n+k)\log n)$ where k is the output size. The output size k is proportional to n plus the number of pairwise segment intersections. This number of pairwise segment intersections is equal to the number of intersection points if we excluded degenerate input configurations, such as n segments intersecting in a point. But in our approach using symbolic perturbation, these n segments intersecting in a single point will be perturbed to form $\binom{n}{2}$ pairwise intersections.

The algorithm needs $O(n+k)$ space because of the size of the planar map and the size of the priority queue. The sweep-line status is linear in n. The priority queue

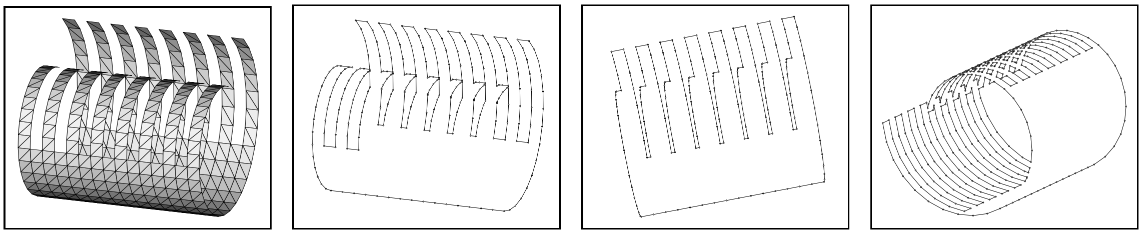


Figure 6.5: Three silhouettes created from the `bracelet` example shown on the left.

would be linear in n if intersection events are only kept in the priority queue as long as the two intersecting segments are direct neighbors in the sweep-line status [35, 55]. The algorithm would need to erase events from the priority queue whenever intersecting neighbors are separated by some event. Since we need the space for the planar map anyway, we have preferred to stay with a simpler and faster heap-based implementation of the priority queue instead of more complicated and slower implementations, for example, a balanced search tree. However, a variant of this algorithm mentioned later on requires that intersection events are stored only once in the priority queue. This variant will allow us to lower the algebraic needs of the geometric predicates [26].

6.4 Silhouette Computation

The *silhouette* of a polyhedron is the boundary of its projection. The silhouette consists of those edges in the projection that separate the visible facets from the background. This definition includes boundaries of holes through which the background can be seen (see Figure 6.5 for an example). The silhouette is always a subset of the contour edges and it can be computed considering only the contour edges.

The silhouette can be used to cast a shadow from an object onto other objects [50]. We compute the silhouette with respect to the direction of the light, either using orthogonal projection for parallel light sources (sunlight) or perspective projection for point light sources. We project the silhouette along the lighting direction onto other objects to obtain the shadow boundary. The exact solution provided by such an object-space algorithm avoids aliasing effects common to raster algorithms.

We compute the silhouette with a modified version of the sweep-line algorithm SegmentIntersections from the previous section applied to the projected contour edges of a polyhedral surface. The processing of the segments, here the projected contour edges, remains the same. Changes in the algorithm are the annotation of the sweep-line status with the *quantitative invisibility*, defined in the next paragraph; moreover, the planar map is only created for those parts of the input that form the silhouette.

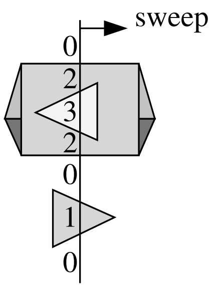


Figure 6.6: An example of the quantitative invisibility ν along the sweep line.

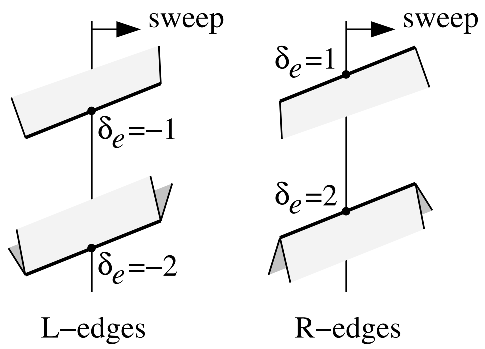


Figure 6.7: The classification into L-edges, R-edges, and their change δ_e in quantitative invisibility crossing from top to bottom.

The *quantitative invisibility* of a feature has been defined by Appel [10] as the number of facets hiding this feature from the viewer. We annotate the sweep-line status with the quantitative invisibility ν of the background (see Figure 6.6 for an example). This is simply the total number of faces stacked upon each other in this region. If the quantitative invisibility is zero on one side of an edge and greater than zero on the other side, this edge is part of the silhouette.

The quantitative invisibility ν is always greater than or equal to zero. It changes at contour edges and it changes only at contour edges. Thus a sufficient condition for a projected edge in the sweep line to be part of the silhouette is that the quantitative invisibility is zero for one of its two incident faces. Specifically, ν changes by one at border edges and by two at contour edges that are not border edges.

We classify contour edges into L-edges and R-edges according to the side where their incident face(s) point to. L stands for left and R for right if seen along the progress direction of the sweep line (see Figure 6.7). We store the quantitative invisibility of a face in the edge below the face. The quantitative invisibility of the next face below is increased if the edge is an R-edge, and it is decreased otherwise. We define δ_e to be the amount of change at an edge e, $\delta_e \in \{-2, -1, 1, 2\}$ (see Figure 6.7).

Using δ_e we can easily restore the quantitative invisibility ν for all new edges at each event in the sweep-line algorithm. We start with the quantitative invisibility ν of the face above the event point and add δ_e for each new edge ordered around the vertex, which matches the insertion order of segments in the algorithm. If ν or $\nu + \delta_e$ is zero the edge is part of the silhouette and is inserted in the planar map.

Both procedures for handling event points need considerable changes in the control flow since not all edges are necessarily part of the output in the planar map. The main control flow remains to process all projected edges. But an additional control flow, encoded in status variables, is needed to keep track of the planar map generation. One status variable *st* keeps track in the first part if any planar map edge arrives from the left at the current event point. If any new edge is inserted in the planar map at this event point, the *pm*.attach_antenna operation is used in the case that *st* is true, or the *pm*.create_antenna operation is used otherwise. If no new edge is inserted, the *pm*.join_faces operation is used in the case that *st* is true, or otherwise nothing has to be done. Another status variable distinguishes between the first insertion of a new edge and all other insertions. We omit the pseudo code here. This method and an alternative stripe-sweep algorithm have been implemented and described in [97].

Altogether, no additional effort is necessary for the silhouette computation compared to the sweep-line algorithm. We summarize the result in the following theorem.

Theorem 6.4.1 (silhouette computation). *Given a set of n_c contour edges that intersect int_c times in the projection, the silhouette of a polyhedral surface can be computed with our modified* SegmentIntersections *algorithm in time $O((n_c + int_c) \log n_c)$.*

Proof. The runtime follows from the sweep-line algorithm in Section 6.3. ☐

The expected runtime of this algorithm (and the hidden-surface removal algorithm following below) for random orthogonal viewing directions follows immediately from the linearity of expectations and $\log n_c < \log n$ for n the number of all edges. The results of the analysis of real-world objects in Chapter 5 fit in here to get the expected runtime.

6.5 Hidden-Surface Removal

We have structured our algorithm HiddenSurfaceRemoval into three steps. The second step, the VisibilityMapOfContours(C,P) algorithm, computes the planar map of all visible parts of the contour edges from the set of projected contour edges C and the polyhedral surface P. Furthermore, each region in the planar map is annotated with a visible facet from the polyhedral surface. This facet serves as a seed for the third step filling in all visible facets. The second step is the core part of the hidden-surface removal problem. The third step is standard and not further described here.

We compute the planar map of the visible contour edges with a modified version of the sweep-line algorithm SegmentIntersections from Section 6.3 applied to the projected contour edges of a polyhedral surface. The processing of the segments, here the projected contour edges, remains the same. Changes in the algorithm are the annotation of the sweep-line status with the *visibility* of the contour edges, and creation of the planar map only for those parts of the input that are visible.

	g		h		
1	R	v	R	v	$\beta(h) \leftarrow \beta(g)$
2a	R	v	R	i	if h is in front of $\beta(g)$: $\beta(h) \leftarrow \beta(g)$, $\beta(g) \leftarrow \mathrm{fc}(h)$
2b					if h is behind $\beta(g)$: nothing changes
3	R	i	R	v	nothing changes
4	R	i	R	i	nothing changes
5	R	v	L	v	impossible[3]
6a	R	v	L	i	if h is in front of $\beta(g)$: for $\beta(h)$, $\beta(g)$ see main text
6b					if h is behind $\beta(g)$: nothing changes
7	R	i	L	v	symmetric to 6
8	R	i	L	i	nothing changes
9a	L	v	R	v	if h is in front of g: $\beta(h) \leftarrow \mathrm{fc}(g)$
9b					if h is behind g: $\beta(g) \leftarrow \mathrm{fc}(h)$
10	L	v	R	i	nothing changes
11	L	i	R	v	symmetric to 10
12	L	i	R	i	nothing changes
13–16	L	*	L	*	symmetric to 1–4

Table 6.1: Case distinction for intersection event points; v denotes visible, and i denotes invisible projected contour edge.

Each contour edge e knows its incident front facing facet, which we call $\mathrm{fc}(e)$ for now. In addition, each contour edge e has a pointer $\beta(e)$ to the next facet behind its projection. It is only maintained for visible contour edges and denotes the facet visible on the side of e that is opposite to $\mathrm{fc}(e)$. Both pointers provide the required labeling of the regions in the planar map with visible facets.

For an intersection event we use the classification into L- and R-edges and the visibility of the two projected edges, g and h, to distinguish 16 cases (see Table 6.1). Among these 16 cases six are mirror symmetric to other cases where g and h, as well as the L and R classifications, are exchanged. Among the ten remaining cases three cases require a depth comparison. Only case 6a needs further discussion, all other cases update the visibility information and $\beta(e)$ as shown in the table.

Case 6a in Table 6.1 represents the situation where a hole opens behind two visible contour edges. The next visible facet in this hole needs to be found. But there is an example showing that there is not enough information in the arrangement of projected contour edges. But before discussing this example, we introduce one aspect of the start/stop event handling that is used in the arguments for the example.

[3] This case is impossible for intersections since both visible contours must be incident to the same visible region and they would intersect in three-space at the intersection point, a contradiction. If this case happens it would be an indication of bad input data, such as non-planar facets. However, this case turns up for start/stop events.

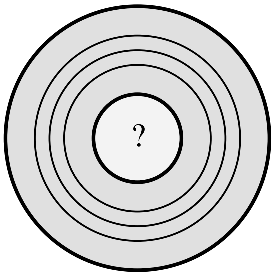


Figure 6.8: Counter example for using contour edges and output sensitivity only.

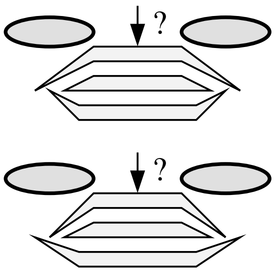


Figure 6.9: Cross sections of two possible interpretations of the counter example in Figure 6.8 on the left.

When processing a start/stop event that consists only of outgoing edges, its visibility cannot be determined from incoming edges. In some cases the visibility can be determined from the facets incident to the outgoing edges, but in other cases it cannot. One example are two outgoing edges; the upper edge an R-edge and the lower edge an L-edge. The visibility has to be tested against the facet that is currently visible at this event point. This can be done in an output sensitive manner. Step two and step three of our algorithm have to be merged together, such that we know the complete visibility map to the left of the current sweep line. Thus the facet to test against is known in the planar map and can be accessed through the sweep-line status structure. Only if the event is visible, the $\beta(e)$ pointer must be set for the outgoing edges.

Back to our example, it shows that even in this output-sensitive setting there is not enough information to solve case 6a in Table 6.1. This example consists of a torus and we look through its hole. Several objects are placed behind the hole. Each object has a simple contour, a single loop. Each contour-edge loop is hidden behind the torus body and no two loops intersect in the projection. Figure 6.8 shows the projected contour edges of an example. The issue arising in case 6a is as follows: Which object is visible in the hole of the torus? Even assuming that we know all visible facets in the output of the torus and its surrounding, we cannot determine any visible facet in the hole. To illustrate this fact, we give two possible cross-sections in Figure 6.9 that would produce exactly the same projection of contour edges shown in Figure 6.8. Note that in the example all contour-edge loops are at the same depth. Furthermore, note that this counter example is not restricted to sweep-line algorithms. It is valid for all algorithms that restrict themselves to contour edges and filled-in regions that are already known to be visible.

One approach solving case 6a might maintain the $\beta(e)$ pointers for all projected contour edges, not only the visible ones. Case 6a could simply assign $\beta(g) \leftarrow \beta(h)$. This works fine for all intersection event points, but fails for start/stop event points, since now we would have to find the $\beta(e)$ pointer also for non-visible start/stop event points. In our counter example, this approach would require to determine the depth of all contour-edge loops as soon as we encounter them.

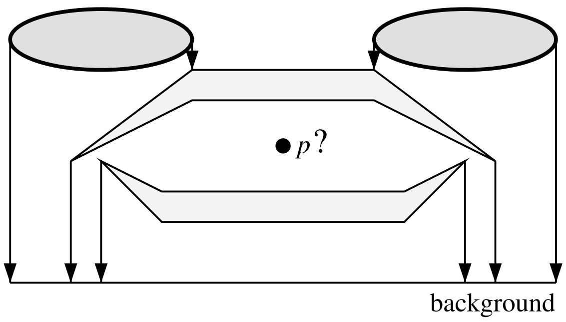


Figure 6.10: Locating a new point p in the cross section.

Tracking all $\beta(e)$ pointers for all edges e can be visualized in the cross section at the position of the sweep line (see Figure 6.10 for an example). Each contour edge points to the facet underneath it (drawn as arrows in Figure 6.10). We are processing event point p and ask for the next facet underneath p, which is the next edge below p in the cross section. Again, the information on the contour edges alone is not sufficient to locate p, and furthermore, in order to find the actual facet we have to traverse non-visible facets starting at some contour edge. The configuration in the cross-section can be extended to a dynamic trapezoidal decomposition with a point location structure with logarithmic query time and polylogarithmic update time [143], if the surface between contour edges would allow an efficient sidedness test in the cross section. For example, in a collection of convex polyhedra the sidedness test can be done using the segment connecting the two points in the cross section that represent the contour of a convex body.

Since we cannot solve the facet-location problem in case 6a and its symmetric counterpart in case 7 satisfactorily, we define a parameter f_{loc} counting the number of facet locations needed. Their frequency is dependent on the polyhedral surface and we expect that the class of objects suitable for the contour edge approach, namely polyhedral approximations of smooth surfaces, have also a low number f_{loc} of facet locations. One argument for that is that better polygonal approximations of the same smooth surface do not change the number of facet-location problems. But the worst-case complexity of f_{loc} is of the order $n_{\mathrm{c}} + int_{\mathrm{c}}$, the total number of events.

To solve the facet-location problem, a general solution would be ray-shooting in three-dimensions amidst triangles. The known query time $T_{\mathrm{ray}}(n)$ based on linear space data structures preprocessed for n triangles is $O(n^{3/4} \operatorname{polylog} n)$ [1, 2], which is not satisfactory in a worst case analysis, since it is needed $O(n_{\mathrm{c}} + int_{\mathrm{c}})$ times in the worst case. However, one can argue that practical implementations for ray shooting work efficiently [74]. Another approach similar to the algorithm in [78] uses the current sweep-line status information and searches for the facet. For each R-edge in the sweep-line status that is above the current event point we search along the surface downwards for the facet underneath the current event point. For a restricted yet useful class of polyhedral surfaces, namely terrains represented as triangulated irregular network (TIN), a better result is shown in the next section.

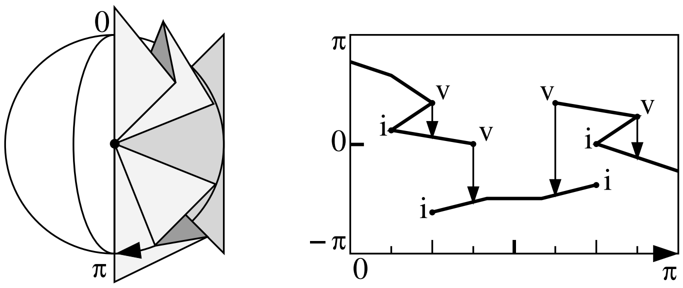


Figure 6.11: Example of the rotational sweep in the parameter space (on the right) of a small ε-sphere around a start/stop event point (on the left), v denotes visible contour edges and i denotes invisible ones.

We continue with the discussion of the start/stop event-point handling. The processing of the projected contour edges in the HandleStartStopEvent procedure remains the same. In addition, during the first phase of processing incoming edges, the visibility of the event point and the visible facet above the event point is determined. In the second phase, if the event point is visible, a rotational sweep starts with the visible facet from vertically above the event point and rotates over all outgoing edges. For each outgoing edge, visibility is tested and the visible facet below the outgoing edge is determined.

In the first phase, three cases can be distinguished. The last (i.e. topmost) incoming visible edge e is an L-edge, it is an R-edge, or there is no incoming visible edge. In the first two cases, the event point is visible and the facet visible above the event point is $\mathrm{fc}(e)$ or $\beta(e)$ respectively. In the third case, the event point has to be tested against the currently visible facet (which we can maintain in an output sensitive manner as explained above). If the event point is visible, the currently visible facet is also the facet visible above the event point. Note that an event point with incoming edges, all of them invisible, is not necessarily invisible as well.

In the second phase, a rotational sweep computes the visibility and the facets behind the outgoing edges. To illustrate the rotational sweep, we intersect the neighborhood of the event point with a small ε-sphere (see Figure 6.11). The rotational sweep is equivalent to a line sweep in the parameter space of the sphere. Since we allow border edges, the arrangement in the parameter space can be an arbitrary arrangement of segments. The sweep is straightforward, except that we might have holes in the surface with the same problems and solutions as for the intersection event points (e.g., parameter f_{loc} and ray shooting). Ignoring this additional costs for holes, the sweep takes $O(d \log d)$ time if d is the out-degree at this event point. The out-degree is counted including non-contour edges. We may argue that the vertex degree is constant on the average and therefore does not change our runtime analysis, but in the worst-case we can process all edges of the polyhedral surface in such a sweep even though only a constant number are contour edges. Still, a good worst case bound can be obtained considering all vertices v of the polyhedral surface. Let d_v be the degree of the vertex v and let n be the overall number of edges. The total runtime for the rotational sweep excluding the runtime for searching in holes is bounded by $O(n \log n)$, since $\sum_{v \in V} d_v \log d_v \leq 2n \log n$.

If we have no border edges in the polyhedral surface, the handling of start/stop event points simplifies. The arrangement in the parameter space of the sphere is a single connected closed path, thus the upper hull is also a singly connected component. Since we consider only half of the sphere, the path may leave and enter this half several times and the upper hull may have two branches in this half. The upper hull of this path can be computed in linear time following this path and maintaining the upper hull known so far on a stack. Whenever the path loops back in front of the current path, the stack is popped. Whenever the path loops back behind the current path, we wait until the path reappears at the current end of the upper hull. One additional case happens if the path loops back on the visible side of the current path and enters a pocket there. A pocket is a region where the path previously looped back on the invisible side of the hull. By Jordan's curve theorem the path will leave the pocket between the entrance point and the upper hull and will continue the upper hull from thereon.

We summarize the runtime of our method in the following theorem.

Theorem 6.5.1 (visible contour edges). *Given a polyhedral surface of n edges and a set of n_c contour edges that intersect int_c times in the projection for a given viewing transformation, the arrangement of visible contour edges can be computed with our modified* SEGMENTINTERSECTIONS *algorithm in time $O((n_c + int_c) \log n_c + n \log n + f_{loc} T_{ray}(n))$, where f_{loc} is the number of face locations needed and $T_{ray}(n)$ refers to the query time of a ray-shooting query amidst n triangles in dimension three. If the polyhedral surface is closed (no border edges) the $n \log n$ term simplifies to n.*

Proof. The runtime follows from the sweep-line algorithm in Section 6.3 plus the event handling of start/stop events and the facet-location problem behind holes. ☐

6.6 Specialization on Terrains

The computation of the visible contour edges simplifies considerably for a polyhedral terrain represented as a triangulated irregular network (TIN). At the beginning we ignore boundary cases and consider an infinite terrain with the viewpoint above the surface. We discuss orthogonal projections first and perspective projections thereafter. For orthogonal projections we choose a specific alignment of the vertical sweep line with the object coordinate system to obtain the following result. We recall that we use a right-handed coordinate system in image space as well as in object space. The sweep line is parallel to the y-axis in image space. The terrain is a function in z over the xy-plane in object space.

Lemma 6.6.1. *Given an orthogonal projection whose image y-axis is parallel to the projected z-axis of the object coordinate system and that points in the same direction, then among all contour edges of a terrain only R-edges can be visible if the (infinitely far away) viewpoint lies above the terrain surface.*

Proof. The plane spanned by the pre-image of the sweep-line (parallel to the image y-axis) and the viewing direction is orthogonal to the xy-plane in object space. Intersecting this orthogonal plane with the terrain gives a two-dimensional function graph for some local coordinate system in this orthogonal plane. Another local coordinate system is defined by the viewing direction. If we define maxima and minima for the function graph with respect to the direction orthogonal to the viewing direction and positive inner product with the z-axis of the object coordinate system, then only maxima can be seen from a viewpoint that is above the function graph. Maxima correspond to intersections of R-edges with the orthogonal plane. □

Among the 16 cases for intersection event points in Table 6.1 only the cases one to four remain. Specifically the problem of holes has disappeared in general, since a hole appears only between an L-edge and an R-edge.

The handling of start/stop events in the general case happens in two phases: locating the event point and the rotational sweep. The first phase simplifies slightly for terrains. If the event point is visible, we know that it lies in the currently visible face of the planar map and does not create a new region nearer to the viewer. The second phase, the rotational sweep, simplifies again considerably. We start with the visible facet above the event point and trace the visible surface until we encounter a contour edge (an R-edge) that is closer to the viewer and continue with the visible surface beginning at this contour. Note that the priority queue gives us the contour edges already in the correct sorted order and that all the additional non-contour edges traced in this sweep are visible edges. Thus the rotational sweep requires constant time per contour edge plus constant time per visible edge.

We generalize now to perspective projections of terrains. We would like to carry over the result of Lemma 6.6.1. The central construction in its proof is that the plane spanned by the sweep line and the viewing direction is orthogonal to the xy-plane in object space. For perspective projections, the planes through the viewpoint that are orthogonal to the xy-plane in object space form a pencil $\mathcal{P}$ of planes. The intersection of $\mathcal{P}$ with the viewing plane forms a pencil $\mathcal{L}$ of lines in the viewing plane with a common intersection point, a *vanishing point*. This vanishing point will be required to lie outside of the viewing frame. This is equivalent to the condition that the z-direction is not part of the perspective viewing cone (we are not looking straight downwards or upwards).

Since the intersection of the pencil $\mathcal{P}$ with the viewing plane is in general not a pencil of parallel lines, we cannot align easily the sweep line to this direction as we have done for the orthogonal case. Instead, we align the sweep line horizontally and sweep from top to bottom. Important is, that the vanishing point of the pencil stays

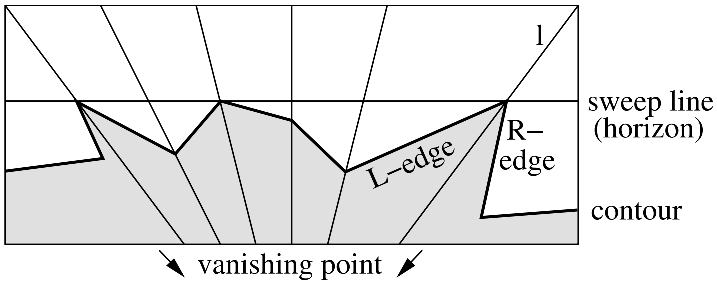


Figure 6.12: Example of a perspective projection of a terrain contour.

always on the same side of the sweep line (see Figure 6.12). The following lemma excludes again the problematic case 6a of Table 6.1.

Lemma 6.6.2. *Given a perspective projection with a view point above the terrain surface and an y-axis in image space that has a positive inner product with the z-axis in object space. Let $\mathcal{P}$ be the pencil of planes orthogonal to the xy-plane of the object coordinate system going through the viewpoint. Let v be the vanishing point of the intersection of $\mathcal{P}$ with the viewing plane. If v is in image space strictly above or strictly below the visible area of the viewing plane, then for a sweep line sweeping from top to bottom there is for any event point no visible L-edge to the right of any visible R-edge.*

Before proving this lemma, it should be pointed out that the definition of R- and L-edges is with respect to the rotated sweep-line orientation.

Proof. The proof of Lemma 6.6.1 carries over to the pencil $\mathcal{L}$ of intersection lines from the pencil $\mathcal{P}$ with the viewing plane. Rephrased for the rotated definition of R- and L-edges, it states the following: Through each event point exist a unique line $l \in \mathcal{L}$. Visible edges to the left of this line must be L-edges, and visible edges to the right of this line must be R-edges. $\square$

Case 6a would require a visible L-edge to the right of a visible R-edge and is thus impossible. All other events are processed as in the general case. The simplification of the rotational sweep for the terrains under orthogonal projections works here as well, except that the rotational sweep has to start from both sides of the sweep line and has to stop once the rotating ray reaches the vanishing point. Instead of introducing the vanishing point formally, the rotational sweep can also proceed until it finds a visible contour edge of the wrong kind of L/R-classification.

There are two alternatives for the perspective case. One alternative requires a perspective projection with a viewing plane orthogonal to the xy-plane in the object coordinate system. Then, the pencil of lines is again parallel and the algorithm can proceed as in the orthogonal case. A postprocessing step can project the result to the final viewing plane. The second alternative stays with the converging pencil of lines, but changes the x-order of the events correspondingly, such that the sweep line moves along the pencil of lines. Both alternatives change the geometric predicates and the analysis of the demands on the arithmetic.

For an infinite terrain the initialization of our sweep-line algorithm must be modified. We need to compute the sweep-line status of the leftmost or topmost sweep line. This implies detecting and inserting (sorting) all contour edges that intersect the sweep line, which can be done in $O(n_c \log n_c)$. Furthermore, all neighbors in the sweep-line status must be checked for intersections.

Boundary edges can be handled easily in the sweep-line algorithm that sweeps from top to bottom if we change the modeling of the problem slightly. We extend each boundary edge of the terrain with an infinite curtain hanging from this edge parallel to the z-axis to $z = -\infty$. Whenever we encounter a visible boundary edge, we keep the depth information of the curtain until some event occurs that is closer to the viewer than the curtain depth. With this conceptual change of the terrain model, the surface of the terrain will not be visible from the underside even if there are holes in the surface.

We summarize the result for terrains in the following theorem.

Theorem 6.6.3 (visible contour edges of terrains). *Given a terrain with a set of n_c contour edges, possibly including border edges, that intersect int_c times in the projection for a given viewing transformation, the arrangement of visible contour edges can be computed with our modified* SEGMENTINTERSECTIONS *algorithm in time $O((n_c + int_c) \log n_c)$.*

Proof. The runtime follows from the sweep-line algorithm in Section 6.3 and Lemma 6.6.2. If the conditions in Lemma 6.6.2 are violated, orientations can be turned around or the image plane can be subdivided into two planes by a line through the vanishing point. The simplified rotational sweep also runs within this bound. ◻

Boundary edges can be handled more easily for orthogonal projections if the boundary, projected onto the xy-plane, forms a single convex polygon. The boundary of a terrain is usually a convex polygon in practice, for example, if the terrain has been computed with a Delaunay triangulation from a point sample.

For the sweep-line algorithm for terrains under orthogonal projections we sweep from left to right. Between the first event point and the last event point, the sweep line intersects exactly two boundary edges. One of these two boundary edges is in front of all edges currently intersecting the sweep-line status and the other boundary edge is behind all intersected edges. The reason can be seen in the xy-plane in object space. Since the plane spanned by the sweep line and the viewing direction is orthogonal to the xy-plane and since it intersects the convex boundary in the projection to the xy-plane exactly twice or not at all, it intersects also the boundary in three space twice or not at all (degeneracies resolved using symbolic perturbation).

The path of border edges in front of the terrain is always visible. Despite of the fact that parts of the surface might be visible below this chain we declare them to be invisible since they would be seen from below the surface. All parts below this front chain are invisible and the algorithm keeps track of the current position of this front chain. The rotational sweep used in a start/stop event stops as soon as it encounters this front chain.

The algorithm simplifies further, since it only has to consider R-contour edges. There may be visible boundary L-edges in the result, but they are not needed to compute the arrangement of visible contour edges. Their incident regions will be filled using the R-edges on the top side of the region. The last change needed for this variant is a special first start-event.

6.7 Exact Integer Arithmetic and Rounding

The combinatorial part of a sweep-line algorithm is known to be sensitive to wrong decisions in the geometric predicates due to rounding errors. The algorithm then tends to compute garbled results or may crash. We use exact integer arithmetic to avoid rounding errors. We use built-in types of limited bit size to avoid the performance drawback of long integer packages of arbitrary precision. In the next section we analyze the required bit size to avoid underflow/overflow problems in the geometric predicates. In this section we describe how to round the input data to a required bit size.

Our algorithm has two input sources that require rounding, the polyhedral surface and the viewing transformation. The polyhedral surface follows Definition 4.1.3. Facets are triangulated to avoid problems with skew polygons. The polyhedral surface is transformed to be inside the sphere of radius R and centered at the origin. Thus for all vertices p: $|p| \leq R$. This transformation can be easily computed optimally using the minimal enclosing ball of the polyhedral surface [191]. We call such a polyhedral surface *normalized with radius* R. The radius R is a parameter for the rounding step.

The quality measure for the rounding steps are the errors in image space. Examples of image space are the screen or the printer output. For the ease of the analysis we assume a squared image space with centered origin of size $[-C, +C]^2$ of unit pixels. We call it an image space of *radius* C.

The viewing transformation maps points p of the polyhedral surface to image points p'. The transformation is *well formed* if for all p with $|p| \leq R$ holds that $|p'| \leq C$. Clipping and zooming happens later on in image space and is not discussed here.

6.7.1 Rounding Point Coordinates of Polyhedral Surfaces

We round a polyhedral surface by rounding the coordinates of its points to the nearest integer value. The radius R determines the number of bits needed for the coordinates. The quality of the rounding is measured with the absolute error in pixels in the image space. The relation between R and the image-space error is established in the following lemma.

Lemma 6.7.1 (rounding point coordinates). *Let P be a normalized polyhedral surface of radius R and let C be the radius of the image space. Let $\overline{p}$ be the point p with coordinates rounded to the next integer value. Then $\forall p \in P : |p' - \overline{p}'| \leq \varepsilon$ for a well-formed orthogonal projection and*

$$\varepsilon = \frac{\sqrt{3}}{2}\frac{C}{R},$$

or for a well-formed perspective projection with aperture angle α and

$$\varepsilon = \frac{\sqrt{3}}{2}\frac{C}{R}\left(\frac{\cos\frac{\alpha}{2}}{1-\sin\frac{\alpha}{2}}\right).$$

Proof. Ignoring translations, the orthogonal projection factors into a rotation and a scaling operation. The scaling factor s must be smaller or equal to C/R for a well-formed projection. Thus, the error bound translates to object space: $\varepsilon/s \geq \varepsilon R/C$. The maximal error while rounding object coordinates is half the diagonal of a unit cube, i.e. $\sqrt{3}/2$, which must be smaller than or equal to $\varepsilon R/C$ to fulfill the claim.

For a perspective projection the point closest to the view point causes the maximal error in image space during rounding. To determine the closest distance for a well-formed perspective projection (i.e., the complete object is visible in image space) we construct the *tangent cone* defined by the apex in the view point (with the aperture angle α in the apex of the cone) and the tangent surface with the sphere $|q| = R$. The view point cannot get any closer as with this tangent cone. The tangent plane at the point on the sphere closest to the viewpoint intersects the tangent cone in a circle of radius $r = R \cdot ((1 - \sin\alpha/2)/(\cos\alpha/2))$. Similar to the argument above, the maximal object space error is now $\varepsilon r/C$.

The proof for point coordinates of vertices extends to all points of the polyhedral surface (i.e. on edges, facets, and tetrahedra in the volume) since they can be represented as a positive convex combination of the points at their vertices. $\square$

We give example values of ε for a screen space of radius $C \in [127, 255, 511]$ (a graphics window of side length 256, 512, or 1024 pixels) and a printer with an image region of radius $C = 100$ mm (200 mm are roughly the width of A4 or letter size paper). We list the values for orthogonal projections and a range of R going from 7 bits to 12 bits to represent coordinates of the polyhedral surface. For a perspective projection with an aperture angle of 40 degree the values for ε would be about 40% larger. A conclusion of this table is that 10 bits are quite sufficient.

		graphics window			printer
bits	R	$C = 127$	255	511	100 mm
7	63	1.75	3.51	7.02	1.37 mm
8	127	0.87	1.74	3.48	0.68 mm
9	255	0.43	0.87	1.74	0.34 mm
10	511	0.22	0.43	0.87	0.17 mm
11	1023	0.11	0.22	0.43	0.085 mm
12	2047	0.054	0.11	0.22	0.042 mm

6.7.2 Rounding of a Viewing Transformation

Here, we consider only orthogonal projections so far. We start with the viewing direction d given as a vector. We round d by scaling it to the length D and then rounding its coordinates to the next integer value. The constant parameter D is used to control the number of bits needed to represent d.

Lemma 6.7.2 (rounding a view direction). *Let C be the radius of the image space and let d be the viewing direction of a well-formed orthogonal projection. Then the image space error ε introduced by scaling d to the length of D and rounding its coordinates to the next integer values is at most*

$$\varepsilon \geq \frac{\sqrt{3}}{2}\frac{C}{D}.$$

Proof. We argue as in Lemma 6.7.1. The error bound in object space is greater than or equal to $\varepsilon R/C$. Assume that d is scaled properly, i.e., $|d| = D$. The error made when rounding the coordinates of d is equivalent to the amount of movement made by the vector d in the object space. This amount is largest at distance R. Thus the object space error bound scales with D/R to an viewing direction error bound. The error introduced by rounding the viewing direction is at most the diagonal of the unit cube: $\sqrt{3}/2 \leq D/R \cdot \varepsilon R/C$. ❑

Ignoring translations, the orthogonal projection can be represented as an orthogonal 3×3 matrix $M = [m_{ij}] \in \mathbb{R}^{3\times3}$. The matrix M factors into a rotation and a scaling operation. Let s be the scale factor. Then the inverse transformation of M is $M^{-1} = \frac{1}{s^2}M^T$. The viewing direction in image space is equal to $(0, 0, 1)$ and in object space $M^{-1} \cdot (0, 0, 1)$. The scale factor does not change the direction, thus the view direction d is the bottom row of M, $d = (m_{31}, m_{32}, m_{33})$. Similarly the other rows represent the pre-images of the remaining base vectors $(1, 0, 0)$ and $(0, 1, 0)$. All three pre-images have a length of $1/s$.

To round an orthogonal projection matrix M, each row is rounded like the view direction in Lemma 6.7.2, i.e., the matrix M is scaled by D/s to get vectors of length D and the matrix entries are rounded to the nearest integer value. We need to compensate for the factor D/s at the end of the algorithm. Since the vectors are not rounded in synchronization to each other, the matrix obtained with this method will also contain shear operations and non-uniform scaling operations. For the general method this causes no problems. But for terrains, the rounding might perturb the up-pointing vector $(0, 1, 0)$ far enough that the alignment of the sweep line with the orientation of the terrain gets violated. Furthermore, for the finite terrain with the convex boundary in the xy-projection the boundary conditions may fail. But for the general algorithm for terrains, sweeping from top to bottom, rounding works well. The alignment criterion requires that the horizon stays horizontal, i.e., $M^{-1} \cdot (1, 0, 0) = \frac{1}{s^2}(m_{11}, m_{12}, m_{13})$ has a zero z-coordinate, m_{13}. Since m_{13} is zero before rounding, it is so after rounding.

6.7.3 Interplay of the Rounding Steps

If we project the rounded polyhedral surface of radius R using the rounded orthogonal projection of magnitude D, the resulting coordinates in image space are in the range $[-3RD, 3RD]$.[4]

Three rounding steps can be identified: Rounding the point coordinates of the input geometry, rounding the viewing transformation, and the final rounding to the output device. The rounding towards the integer grid of the output device introduces an error of at most $\sqrt{3}/2$ pixels. Using the error bound control described above, the absolute error in image space can be controlled individually for the other two rounding steps. One solution might enforce that all three rounding steps must not change the resulting projection by more than one pixel, i.e., the sum of the two error bounds must be smaller than $\frac{1}{2}$ pixels. Another scenario might argue that the three rounding steps are independent in their influence on the users perception. For example, the rounded view direction is not the one provided from the view control, but it is a legal view direction and successive views from a rotation of the polyhedral surface will result in a smooth movement of the object.

6.8 Bit Precision Required for Exact Geometric Predicates

With respect to the algebraic degree of the geometric operations the problem of finding segment intersections in the plane has several variants. We compute the arrangement of intersecting line segments in the plane, which includes sorting two segment intersections on a third segment. This problem is known to have algebraic degree four [26]. The Bentley-Ottmann algorithm as described above requires degree five, since it compares the x-coordinate also of unrelated intersection points in

[4] Using a *skew projection*, this range can be tightened to $[-2RD, 2RD]$, which may save a bit but introduces a singularity.

the priority queue. The algorithm is easy but suboptimal. In the following we determine the exact requirements on the arithmetic. Thereafter, we discuss the small modifications necessary to make the same algorithm run with degree four predicates only, following the results in [26].

The geometric predicates use exact integer arithmetic with a sufficient number of bits to guarantee the exact result. The geometric predicates work on two basic types of input: image points in Cartesian representation and intersection points in homogeneous representation. An image point p in Cartesian representation is the image of a point of the polyhedral surface under a viewing transformation. The viewing transformation gives also a correct depth value for the depth comparisons in image space. Thus the image point p consists of the three coordinates (p_x, p_y, p_z). An intersection point in homogeneous representation consists of three entries, (p_{hx}, p_{hy}, p_{hw}), and its Cartesian coordinates are $(p_x, p_y) = (p_{hx}/p_{hw}, p_{hy}/p_{hw})$.

We denote a segment from the image point p to the image point q with $\overline{pq}$. We denote a triangle defined by three image points, p, q, and r, with Δpqr.

The Cartesian coordinates of the image points are (rounded) integer values within the *value range* $\mathbb{Z}^B := [-B, +B]$.[5] We give bounds on the value range of intermediate results for all the geometric predicates and the intersection point representation. If the exact integer number type used for the computation in a predicate does not overflow nor underflow in the value range the result of the predicate is exact.

A signed integer number type of b bits can hold a value range of $\mathbb{Z}^{2^{b-1}-1}$. Given a value range $\mathbb{Z}^E$ required for an expression, the evaluation of the expression needs at least $2 + \lfloor \log_2 E \rfloor$ many bits including the sign bit.

Lemma 6.8.1. *Given two signed integer values u in $\mathbb{Z}^{B_u}$ and v in $\mathbb{Z}^{B_v}$,*

i) *$sign(u)$, $even(u)$, and $odd(u)$ are in $\mathbb{Z}^1$.*

ii) *$|u|$ and $-u$ are in $\mathbb{Z}^{B_u}$.*

iii) *$u < v$, $u \leq v$, $u = v$ and $u \neq v$ are in $\mathbb{Z}^{\max(B_u, B_v)}$.*

iv) *$u + v$ and $u - v$ are in $\mathbb{Z}^{B_u + B_v}$ and*

v) *$u * v$ is in $\mathbb{Z}^{B_u * B_v}$.*

Proof. Calculations with the minimal/maximal values of the ranges. □

[5] We ignore the potential asymmetry of the bounds if negative numbers are represented as two-complements.

Lemma 6.8.2 (triangle area). *Given three points, p, q and r, the determinant*

$$\det(p,q,r) := \det \begin{pmatrix} p_x & p_y & 1 \\ q_x & q_y & 1 \\ r_x & r_y & 1 \end{pmatrix}$$

computes twice the signed two-dimensional area of the triangle Δpqr. The sign is positive if the points pqr are given in counterclockwise order. This expression can be evaluated using

$$\det(p,q,r) = (q_x - p_x)(r_y - p_y) - (q_y - p_y)(r_x - p_x)$$

in the value range $\mathbb{Z}^{4B^2}$ provided p, q, r have coordinates in $\mathbb{Z}^B$, which is tight.

Proof. For the relation to the signed area, we refer to standard text books [23]. Subtracting the first line from the other lines in the determinant gives the expression for evaluation. This can also be seen geometrically as a coordinate transformation with a vector $(-p_x, -p_y)$ which does not change the area of the triangle. Calculating the value ranges straightforward yields the bound for all intermediate results except the final result that would be in $\mathbb{Z}^{8B^2}$. We observe that the largest triangle we can build within the allowed coordinate values has its vertices on the boundary of the square $[-B, +B] \times [-B, +B]$. The maximal area achievable is $(2B)^2/2$. Thus, the maximum for the determinant is $4B^2$. The bound will be achieved if we place the three points in three different corners of the mentioned square. ☐

Lemma 6.8.3 (tetrahedron volume). *Given four points, p, q, r, and s, the determinant*

$$\det(p,q,r,s) := \det \begin{pmatrix} p_x & p_y & p_z & 1 \\ q_x & q_y & q_z & 1 \\ r_x & r_y & r_z & 1 \\ s_x & s_y & s_z & 1 \end{pmatrix}$$

computes six times the signed volume of the tetrahedron spanned by $pqrs$. The sign is positive if s is on the positive side of the oriented triangle Δpqr. This expression can be evaluated using

$$\begin{aligned} \det(p,q,r,s) &= (r_y - p_y) * ((q_x - p_x) * (s_z - p_z) - (s_x - p_x) * (q_z - p_z)) \\ &+ (r_z - p_z) * ((q_y - p_y) * (s_x - p_x) - (s_y - p_y) * (q_x - p_x)) \\ &+ (r_x - p_x) * ((q_z - p_z) * (s_y - p_y) - (s_z - p_z) * (q_y - p_y)) \end{aligned}$$

in the value range $\mathbb{Z}^{16B^3}$ provided p, q, r, s have coordinates in $\mathbb{Z}^B$. The final result is in $\mathbb{Z}^{8B^3}$, which is tight.

Proof. Again, a standard transformation yields the second expression used for evaluation. Calculating the value ranges straightforward and Lemma 6.8.2 applied to the suitable subexpressions yields the bound for all intermediate results and the same volume argument yields the bound for the final result. The bound will be achieved if we place the four points in four affine independent corners of the cube. ☐

The table in Section 6.3.5 lists all geometric operations needed in the sweep-line algorithm to compute the arrangement of a set of segments. For the silhouette computation and the hidden-surface removal five additional operations are needed. All points in this table are image points.

signature	semantic
traits.leftturn(p,q,r)	returns true if p, q, and r form a left turn.
traits.edge_infrontof_edge(p,q,r,s)	returns true if $\overline{pq}$ is in front of $\overline{rs}$.
traits.edge_infrontof_facet(p,q,r,s,t)	returns true if $\overline{pq}$ is in front of Δrst.
traits.edge_infrontof_facet2(p,q,r,s)	returns true if $\overline{pq}$ is in front of Δprs.
traits.point_infrontof_facet(p,q,r,s)	returns true if q is in front of Δqrs.

The following table lists all geometric predicates and their value ranges. Coordinates of image points are in $\mathbb{Z}^B$. B is a parameter for the results. The easy bounds are explained in the table. The other bounds follow in the remainder of this section.

signature	$\mathbb{Z}^\bullet$	comment
traits.do_intersect(p,q,r,s,t)	$4B^2$	if the segments intersect t_{hx}, t_{hy} are in $\mathbb{Z}^{4B^3}$, t_{hw} is in $\mathbb{Z}^{8B^2}$ (see below).
traits.less_x_point_point(p,q)	B	basic.
traits.less_x_int_point(p,q)	$8B^3$	Eval $p_{hx} < q_x p_{hw}$, use bounds on p.
traits.less_x_int_int(p,q)	$32B^5$	Eval $p_{hx} q_{hw} < q_{hx} p_{hw}$, use bounds on p and q.
traits.less_slope(p,q,r,s)	$8B^2$	Eval $(q_y - p_y)(s_x - r_x) < (q_x - p_x)(s_y - r_y)$.
traits.compare_y(p,q,r,s,v)	$8B^3$	(see below)
traits.leftturn(p,q,r)	$4B^2$	Eval $\det(p, q, r) > 0$.
traits.edge_infrontof_edge(p,q,r,s)	$16B^3$	(see below)
traits.edge_infrontof_facet(p,q,r,s,t)	$16B^3$	(see below)
traits.edge_infrontof_facet2(p,q,r,s)	$16B^3$	(see below)
traits.point_infrontof_facet(p,q,r,s)	$16B^3$	(see below)

When intersecting two segments in the sweep-line algorithm with traits.do_intersect(p,q,r,s,t) their relative order on the sweep line is known and only an intersection to the right of the sweep line is of interest. This specialized situation is shown in Figure 6.13. Furthermore, we are only interested in proper intersections in the interior of the segments. Common endpoints are handled as start/stop events in our algorithm and an endpoint in the interior of the other segment will be excluded as degenerate (and will be resolved by the symbolic perturbation in the next section).

Lemma 6.8.4 (segment intersection). *Given a segment $\overline{pq}$ and a segment $\overline{rs}$, oriented from left to right, and $\overline{rs}$ is below $\overline{pq}$ at the current position of the sweep line. Let t_{hw} be the slope difference*

$$t_{hw} = (q_x - p_x)(s_y - r_y) - (q_y - p_y)(s_x - r_x).$$

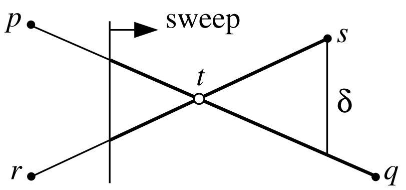


Figure 6.13: Configuration of two intersecting segments, $\overline{pq}$ and $\overline{rs}$, with intersection point t in the traits.do_intersect(p,q,r,s,t) operation.

If the segments intersect to the right of the sweep line, the slopes converge and $t_{hw} > 0$. *Given that the coordinates of* p, q, r, s *are in* $\mathbb{Z}^B$, *the intersection test can be decided in* $\mathbb{Z}^{4B^2}$. *If the segments intersect the hx and hy-coordinates of* t *are in* $\mathbb{Z}^{4B^3}$ *and* t_{hw} *is its common denominator and is in* $\mathbb{Z}^{8B^2}$.

Case 1: $s_x \leq q_x$. *Let* $\delta' = \det(p, q, s)$. *Then,* $\delta' > 0$ *if and only if the segments intersect to the right of the sweep line and the intersection point* t *has the value*

$$t = \begin{pmatrix} (t_{hw} - \delta')s_x + \delta' r_x \\ (t_{hw} - \delta')s_y + \delta' r_y \\ t_{hw} \end{pmatrix} .$$

Case 2: $s_x > q_x$. *The symmetric result is* $\delta' = \det(r, q, s)$, *and*

$$t = \begin{pmatrix} (t_{hw} - \delta')q_x + \delta' p_x \\ (t_{hw} - \delta')q_y + \delta' p_y \\ t_{hw} \end{pmatrix} .$$

Proof. Case 1: The efficient computation of t reusing known temporary results is taken from [97, Lemma 2.80], where it is stated as

$$\begin{pmatrix} t_{hx} \\ t_{hy} \end{pmatrix} = \begin{pmatrix} t_{hw}s_x - (s_x - r_x)\delta' \\ t_{hw}s_y - (s_y - r_y)\delta' \end{pmatrix} .$$

The auxiliary value δ' is twice the area of the respective triangle. It is also equal to $\delta(q_x - p_x)$, where δ denotes the vertical distance between s and $\overline{pq}$ (see Figure 6.13).

This equation has a value range for intermediate results that exceeds the value range for the final results. A change in the evaluation order and the observation that $t_{hw} - \delta' = \det(p, r, q)$ yield the better bound.

The value of t_{hw} is obviously in $\mathbb{Z}^{8B^2}$. The values of t_{hx} and t_{hy} are obviously in $\mathbb{Z}^{8B^3}$, which is the correct answer for the intersection point of two arbitrary lines. But the intersection point must lie on the two segments, which implies

$$-B \leq \frac{t_{hx}}{t_{hw}}, \frac{t_{hy}}{t_{hw}} \leq B .$$

If t_{hw} is only in $\mathbb{Z}^{4B^2}$ the better value range follows immediately from the above inequalities. Lets assume that t_{hw} is in $\mathbb{Z}^{8B^2}$ but not in $\mathbb{Z}^{4B^2}$. W.l.o.g. $q_x \geq p_x$ (otherwise exchange p and q), $q_y \geq p_y$ (otherwise mirror the image at the x-axis), $p_x q_y - p_y q_x \leq 0$ (otherwise exchange the x- and the y-coordinate axis of the image, which will change the sign of this 2×2 determinant), and $s_y \geq r_y$ (otherwise exchange s and r). Then, we observe in the expression $t_{hw} = (p_y - q_y)(s_x - r_x) - (q_x - p_x)(r_y - s_y)$ that the first three parentheses are positive expressions. To force this expression not to be in $\mathbb{Z}^{4B^2}$ the expression in the last parentheses must be negative, thus $s_x < r_x$. Applying these inequalities to the factors in the general intersection solution $t_{hx} = (q_x - p_x)(r_x s_y - r_y s_x) - (p_x q_y - p_y q_x)(s_x - r_x)$ the two products have equal signs (or might be zero). Because the two products are subtracted, they cannot add up to values in $\mathbb{Z}^{8B^3}$. The values stay in $\mathbb{Z}^{4B^3}$. $\square$

An example of an almost tight lower bound is $p = (-B, B), q = (B, B-1), r = (B-1, -B)$, and $s = (B, B)$.

The order comparison traits.compare_y(p,q,r,s,v) along the sweep line is only needed at start/stop event points v. Let $x = v_x$. Then the y-value of the intersection of the sweep line with the segment $\overline{pq}$ is $y = ((q_x - x)p_y + (x - p_x)q_y)/(q_x - p_x)$. The comparison of two such y-coordinates is in $\mathbb{Z}^{8B^3}$.

The depth comparisons traits.edge_infrontof_edge(p,q,r,s), traits.edge_infrontof_facet2(p,q,r,s), and traits.point_infrontof_facet(p,q,r,s) reduce directly to a sign test of $\det(p, q, r, s)$.

The depth comparison traits.edge_infrontof_facet(p,q,r,s,t) returns true if the segment $\overline{pq}$ is front of the triangle Δrst at the current event point. The segment crosses the triangle. If the segment is in front of the triangle at the event point, it is in front of triangle everywhere in the projection of the triangle. The segment may have a common endpoint with the triangle. We test p against r, s, and t. In the case of a common endpoint we can answer the predicate using the corresponding traits.edge_infrontof_facet2 predicate. Otherwise, using traits.leftturn we test if p is in the interior of the projected triangle and answer the predicate with a call to the traits.point_infrontof_facet predicate. If p is not in the interior of the projected triangle the traits.leftturn tests tell us an edge where the segment leaves the triangle. Thus, we can use the traits.edge_infrontof_edge predicate to give the answer. This last depth comparison also boils down to a sign test of $\det(p, q, r, s)$.

The Bentley-Ottmann algorithm as described so far needs degree five predicates. Boissonnat and Preparata prove in [26] that this algorithm computes correctly all segment intersections (but not the arrangement) under the predicate arithmetic model of degree three. Therefore, the algorithm needs to exclude multiple entries of intersection events in the priority queue and the arithmetic must be based on an IEEE floating point type.

Extending the arguments from [26], we show that the same algorithm computes the arrangement of intersecting segments under the predicate arithmetic model of degree four. The result for the degree three model guarantees that the algorithm finds all intersection points. For the arrangement, we need to show that the algorithm processes multiple intersections on the same segment in the right order. The order comparison is done using the degree five predicate traits.less_x_int_int(p,q) with the two segment intersection points p and q. The expression behind this predicate is $p_{hx}q_{hw} < q_{hx}p_{hw}$. Numerator and denominator are degree three and degree two respectively and therefore exact. The comparison of the x-coordinate of two intersection points on the same segment is known to be degree four [26]. Thus, in this case the expression $p_{hx}q_{hw} < q_{hx}p_{hw}$ evaluates exactly in the degree four model. Unrelated intersection points might cause rounding errors in this expression, but the monotonicity of IEEE floating point arithmetic guarantees that these comparisons cannot interfere with the correct order of the degree four comparisons of related intersection points. All other predicates are of degree four or less and therefore exact.

Symbolic perturbation has to be aligned with this changed model of predicate evaluation. In particular, the predicate traits.less_x_int_int(p,q) needs perturbation if p and q have a segment in common, but it must not be perturbed if p and q have no segment in common.

6.9 Solving Degeneracies with Symbolic Perturbation

Degenerate input is handled in our algorithms using symbolic perturbation. Symbolic perturbation moves the effort of dealing with degeneracies from the control flow of the algorithm to the evaluation of the geometric predicates. An overview of symbolic perturbation can be found in [68], various schemes are described in [7, 59, 64, 136, 140, 170, 197]. Specifically, our scheme is influenced by the so-called simulation of simplicity [64, 140] and the work in [7].

Our definition 4.1.3 of polyhedral surfaces excludes many typical degeneracies in three-space. For example, the 2-manifold property forbids the intersection of segments in three-space except for common endpoints. The viewing transformation and the vertical sweep line introduce all the degeneracies we have to address. Thus, we only perturb the viewing direction symbolically instead of applying a general scheme such as simulation of simplicity to the input data.

Degeneracies can be distinguished into problem-dependent and algorithm-dependent degeneracies [169, 197]. The vertical sweep line causes algorithm-dependent degeneracies that are easily resolved with symbolic perturbation. The degeneracies introduced by the viewing transformation are problem dependent. Since we perturb the viewing transformation it is obvious that the perturbation will be valid and that the result of the modified algorithm will be a solution of a nearby problem instance. In case of degeneracies, the output might contain empty facets or zero-length edges. Postprocessing is easy and will not exceed the cost of hidden-surface removal. Sometimes, postprocessing is not needed, for example, if the output is only visualized.

The following list enumerates all the degeneracies in the projection that we have to address with our scheme and how we have solved them. We use a right-handed coordinate system in image space. The image plane corresponds to the xy-plane. The sweep line moves from left to right along the x-axis. The z-axis denotes the depth value. The positive z-axis points towards the viewer.

1. *Two start/stop events have the same x-coordinate.* The usual lexicographic order translates to our symbolic perturbation notion; points of equal x-coordinate but different y-coordinates are separated with an infinitesimal shear transformation in the xy-plane along the x-axis:
$$S_{xy}(\varepsilon_1) = \begin{pmatrix} 1 & \varepsilon_1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$
Points of equal x- and y-coordinate but different z-coordinates are separated with an infinitesimal shear transformation in the xz-plane parallel to the x-axis:
$$S_{xz}(\varepsilon_2) = \begin{pmatrix} 1 & 0 & \varepsilon_2 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}.$$
Note that all three coordinates are equal if the points belong to the same vertex. Otherwise the input format definition is violated.

2. *Vertical segments, segments of zero length.* Vertical segments are tilted slightly by S_{xy}. Segments of zero length must have some depth since there are no segments of length zero in the three-dimensional surface. Thus, S_{xz} tilts the segment slightly. Note that horizontal segments are not degenerate in our setting. Since we shear and do not rotate horizontal segments stay horizontal so far.

3. *Three segments intersect in a point.* The stabbing line through three segments in space defines a ruled surface [181]. Whenever the ruled surface contains the view point or the viewing direction, the three segments intersect in a single point in the projection. Note that S_{xy} does not change this situation at all. And the single parameter ε_2 of S_{xz} is not sufficient in any case to move the view point or viewing direction away from the ruled surface. Another parameter is needed to form a two-dimensional parameter space. We use an infinitesimal shear transformation in the yz-plane parallel to the y-axis:
$$S_{yz}(\varepsilon_3) = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & \varepsilon_3 \\ 0 & 0 & 1 \end{pmatrix}.$$

4. *Two overlapping segments.* Since no two segments overlap or intersect in their interior in three-space, two overlapping segments in the projection span a plane in three space. The perturbation with S_{xz} and S_{yz} moves the view point or view direction out of this plane. Thus, overlapping segments are considered never as intersecting in the projection.

5. *The intersection point of two segments has the same x-coordinate as a segment endpoint.* If the y-coordinates differ the transformation S_{xy} establishes the lexicographic order. If the y-coordinates are equal there is no natural z-coordinate for the intersection point to continue easily with the lexicographic order. Lets assume a line parallel to the y-axis that goes through the segment endpoint in question. Now, this degeneracy is equivalent to the intersection of three segments and already solved, except that the auxiliary line may intersect one of the segments (never both). But this specific case is perturbed by S_{xz}.

6. *Two intersection points have the same x-coordinate.* If the y-coordinates differ the transformation S_{xy} establishes the lexicographic order. Otherwise, we have four segments intersecting in a point which we have already addressed above with three intersecting segments.

7. *An endpoint intersects the interior of a segment.* Since the segment and the endpoint are disjoint in three-space, they span a plane and S_{xz} and S_{yz} perturbs this case.

8. *No two segments cross the sweep line – positioned at a start/stop event – at the same y-coordinate except they have a common endpoint equal to the start/stop event.* Proof by contradiction: Assume the two segments intersect in their interior and with the sweep line in a single point (or the case is trivial). This intersection has the same x-coordinate than the start-stop event where the sweep line is positioned. Equal x-coordinates have already been excluded.

Our perturbation scheme has therefore the three parameters ε_1, ε_2, $\varepsilon_3 > 0$. The shear transformations can be composed to the single transformation

$$T(\varepsilon_1, \varepsilon_2, \varepsilon_3) = S_{yz}(\varepsilon_3) \cdot S_{xz}(\varepsilon_2) \cdot S_{xy}(\varepsilon_1) = \begin{pmatrix} 1 & \varepsilon_1 & \varepsilon_2 \\ 0 & 1 & \varepsilon_3 \\ 0 & 0 & 1 \end{pmatrix}.$$

We perturb all projected points using the transformation T. The expressions in the geometric predicates become polynomials in $\varepsilon_1, \varepsilon_2, \varepsilon_3$. The result of a geometric predicate is computed as a limit process for $\varepsilon_1, \varepsilon_2, \varepsilon_3$ going to zero. More precisely, for the predicates under consideration the three epsilon parameters can be replaced with a single epsilon parameter ε and appropriate exponents: $\varepsilon_1 \mapsto \varepsilon$, $\varepsilon_2 \mapsto \varepsilon^3$, and $\varepsilon_3 \mapsto \varepsilon^{10}$. The monomials in the epsilon polynomials of the geometric predicates can be sorted according to the exponent of their epsilon coefficient. The term without epsilon coefficient is equal to the unperturbed computation. Only if this expression fails to compute the result, the other parts of the polynomial need to be evaluated according to their order. Thus symbolic perturbation is not only elegant but also efficient in a way that the extra computation is only needed in the case of degeneracies.

The actual computation of these polynomials is a mechanical task and we illustrate only a few examples here. Given an image point p, the perturbed point

$$p' = \begin{pmatrix} p'_x \\ p'_y \\ p'_z \end{pmatrix} = T \cdot p = \begin{pmatrix} p_x + \varepsilon p_y + \varepsilon^3 p_z \\ p_y + \varepsilon^{10} p_z \\ p_z \end{pmatrix}.$$

Given two image points p and q, we compare their perturbed x-coordinates and obtain the lexicographic order (as expected),

$$\begin{aligned} p'_x < q'_x &\Leftrightarrow p_x + \varepsilon p_y + \varepsilon^3 p_z < q_x + \varepsilon q_y + \varepsilon^3 q_z \\ &\Leftrightarrow (p_x < q_x) \vee (p_x = q_x \wedge p_y < q_y) \vee (p_x = q_x \wedge p_y = q_y \wedge p_z < q_x). \end{aligned}$$

The next example computes the leftturn predicate, i.e., $\det(p,q,r) > 0$. Instead of expanding and factoring polynomials, we can directly factor out determinants. We will obtain similar determinants as $\det(p,q,r)$ but using different coordinate axes. Therefore, we define the following shorthand notation; for all i,j in $\{x,y,z\}$:

$$\det{}^{<ij>}(p,q,r) := \det \begin{pmatrix} p_i & p_j & 1 \\ q_i & q_j & 1 \\ r_i & r_j & 1 \end{pmatrix}.$$

Consequently, $\det^{<xy>}(p,q,r)$ is equivalent to $\det(p,q,r)$, and $\det^{<ij>}(p,q,r) = -\det^{<ji>}(p,q,r)$, since it transposes two adjacent columns. We derive the result for $\det(p',q',r')$ using this notation:

$$\begin{aligned} \det(p',q',r') &= \det \begin{pmatrix} p'_x & p'_y & 1 \\ q'_x & q'_y & 1 \\ r'_x & r'_y & 1 \end{pmatrix} \\ &= \det \begin{pmatrix} p_x + \varepsilon p_y + \varepsilon^3 p_z & p_y + \varepsilon^{10} p_z & 1 \\ q_x + \varepsilon q_y + \varepsilon^3 q_z & q_y + \varepsilon^{10} q_z & 1 \\ r_x + \varepsilon r_y + \varepsilon^3 r_z & r_y + \varepsilon^{10} r_z & 1 \end{pmatrix} \\ &= \det(p,q,r) - (\varepsilon^3 - \varepsilon^{11}) \det{}^{<yz>}(p,q,r) + \varepsilon^{10} \det{}^{<xz>}(p,q,r). \end{aligned}$$

This expression is still simple enough to see that the area of a triangle in the perturbed projection is zero if and only if the triangle has zero area in all three projections, which implies that the triangle has zero area in three-space.

A disadvantage of this symbolic perturbation scheme is that the expressions of the more complicated predicates, for example the comparison of the x-coordinate of two intersection points, are beyond this kind of plausibility check.

For more complicated predicates some monomials in the epsilon polynomial are combinations of several terms. Adding these terms increase the value range needed in the arithmetic. The worst what happens for our predicates is $\mathbb{Z}^{256B^5}$ for the comparison of the x-coordinate of two intersection points compared to the unperturbed value range of $\mathbb{Z}^{32B^5}$. This is an increase of three bits in the bit size needed. But this increase is only needed in the case of degeneracies.

6.10 Conclusion

We have implemented two prototypes. The first implementation uses a sweep-line algorithm similar to the implementation in Leda [130]. It realizes hidden-line removal for terrains and draws the result right away without creating a planar map.

The explicit handling of the degeneracies is elegant and easy at the level of the sweep-line algorithm itself, but it turned out to be a combinatorial explosion of special cases when continuing. Procedures, such as tracing the projection of an edge along the surface behind the edge (to maintain the β(e) pointer) got cluttered with special cases and it was hard to convince oneself that all cases had been covered. Another example is the rotational sweep, where events of different depths had to be considered in a unified manner. Finally, this implementation is hard to document.

At this point, we have started the second implementation using symbolic perturbation and the Bentley-Ottmann sweep-line algorithm as described above. This implementation computes the arrangement of the projected contour edges in a planar map. The result is drawn from the planar map. Based on this second implementation, the silhouette computation for general polyhedral surfaces has been realized.

In this second implementation, degeneracy handling is factored out of the main algorithmic control flow. The degeneracy handling is separated in its own layer which implements the geometric predicates for the perturbed input. These perturbed geometric predicates can use internally the non-perturbed geometric predicates if the non-perturbed predicates can be parameterized appropriately for the different coordinate axes (see for example the determinant evaluation in the section above).

We have deliberately adapted the general symbolic perturbation scheme to our specific setting with our strict definition for polyhedral surfaces as part of the input. This adaptation made symbolic perturbation successful and easy to analyze in our setting. Whether general schemes "from the shelf" would have been applicable that easily is doubtful [37] and this poses a challenge for the usefulness of symbolic perturbation in a library such as Cgal.

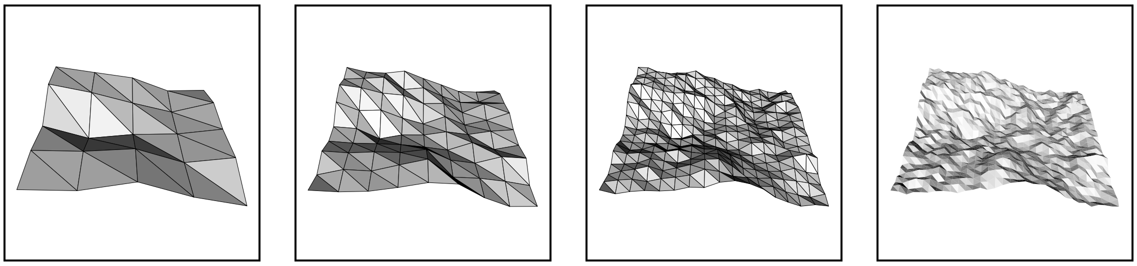
We had also the impression during coding that the Bentley-Ottmann sweep-line algorithm would be faster in practice for non-degenerate input, since it is "specialized" on the general case. One example are intersection events. Here, the Bentley-Ottmann sweep-line algorithm knows that only two segments intersect. The other algorithm explicitly tests for more segments through the same event point. The degeneracy testing in the Bentley-Ottmann sweep-line algorithm is hidden in the predicates and costs virtually nothing for non-degenerate cases. Only in the case of degeneracies, the Bentley-Ottmann algorithm is disadvantaged by the increasing number of events.

Since both implementations differ in the output handling – the first implementation draws directly onto screen while the second one produces a planar map as an intermediate structure – a comparison is difficult, but our expectation is confirmed. The plain segment-intersection algorithm in both implementations is approximately equally fast in drawing the image on screen. Considering for the second algorithm only the time to create the planar map, thus not counting the drawing nor the deletion of the planar map, it is about a factor of two faster.

The opposite claim with respect to the efficiency of degeneracy handling approaches in sweep-line algorithms is made in [37]. We discuss why the reasons given for this claim in [37] do not hold in our comparison and why this claim may not be a contradiction to our finding. Reason one is the actual difference in the runtime complexity of both algorithms. However, our examples frequently contain degeneracies but, compared to the number of events, degeneracies are clearly a minority. Reason two addresses the complexity of postprocessing. Postprocessing is not necessary in our setting. Even if postprocessing is necessary its complexity is fairly low compared to the hidden-surface removal. Reason three makes the claim that the algorithm handling degeneracies directly is only moderately more complex. This is true for the segment-intersection algorithm, but not for our hidden-surface removal algorithm as our experience with the first implementation showed.

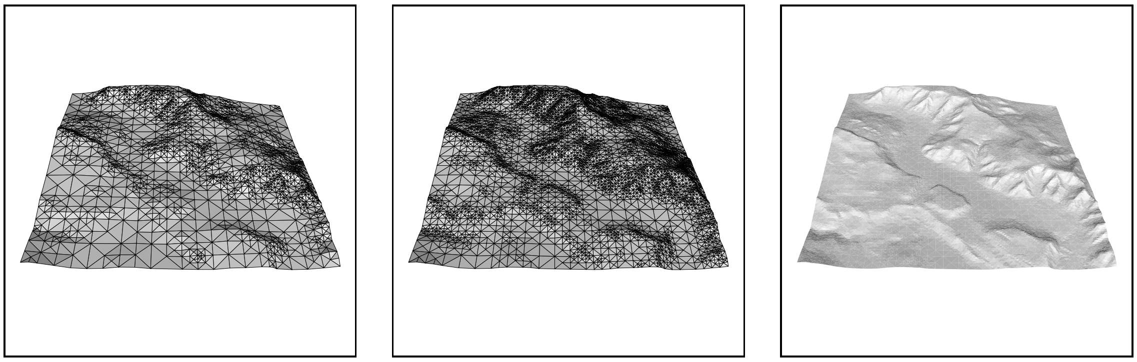
The second important decision is the use of exact arithmetic. A correct implementation must use its input in a consistent way. The polyhedral surface and the viewing transformation need to be rounded before the preprocessing. Our current implementations make no use of preprocessing. We round the polyhedral surface after projecting it but before extracting contour edges. Rounding before projecting would reduce the available value range for the polyhedral surface, respectively increase the value range needed for the geometric predicates.

We parameterize the geometric predicates in our implementation to work as long as possible with 32 bit signed integers and we switch to 64 bit signed integers for all other geometric predicates. Ignoring symbolic perturbation the largest value range in our algorithm is $\mathbb{Z}^{32B^5}$. Using 64 bit signed integers we can set B to 3104. For the larger value range needed for symbolic perturbation we use long double with 107 bit mantissa. If we propagate B further to the parameters R and D for rounding the polyhedral surface and the viewing direction in object space we would obtain $R = D = 55$, which is obviously not enough. Using long double would give $B = 1383604$ and $R = D = 1176$, which is quite reasonable for imaging without the possibility of zooming. The result that arithmetic of degree four is sufficient has not been used in the implementation yet.



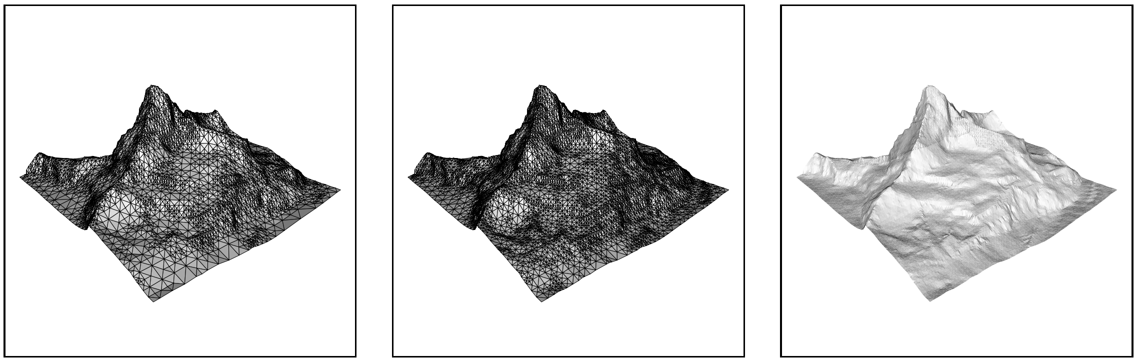
rnd3 rnd4 rnd5 rnd6

Figure 6.14: A synthetic terrain in different resolutions.



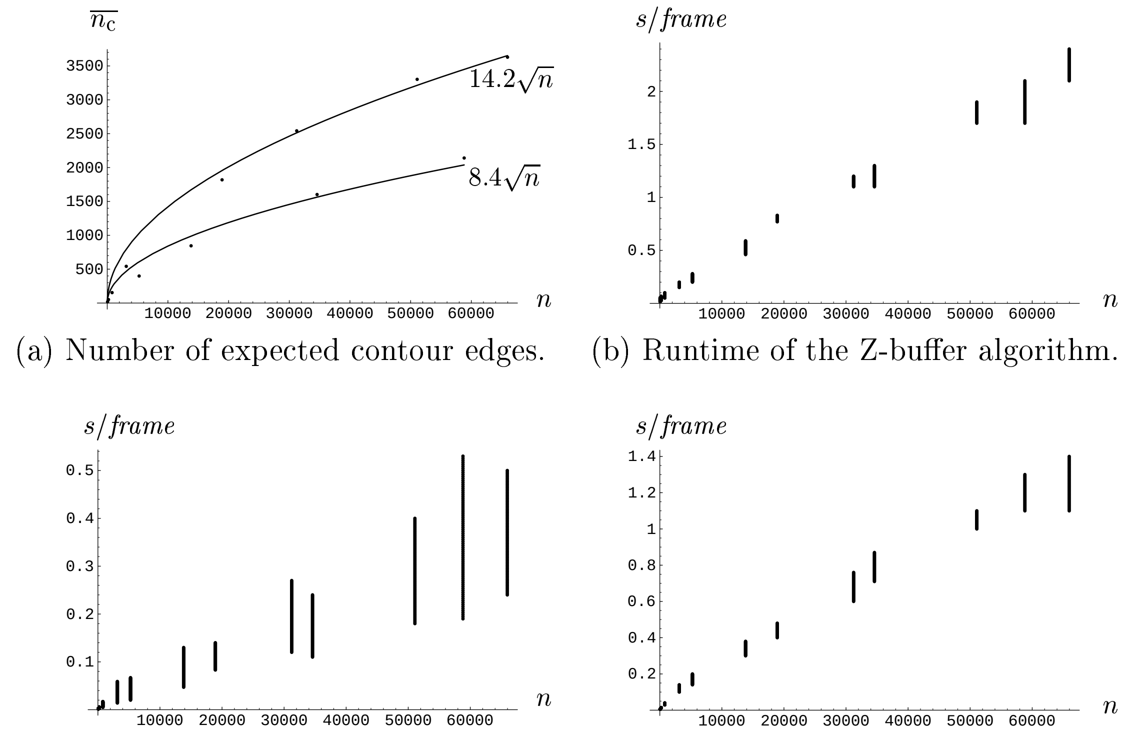
Albis 10 Albis 5 Albis 2

Figure 6.15: Albis and Türlersee in Switzerland in different resolutions. © Federal Office of Topography, Switzerland, 1999, Source: Digital Height Model DHM25 (BA4475,JD2982).



Matter 10 Matter 5 Matter 2

Figure 6.16: The Matterhorn mountain in Switzerland in different resolutions. © Federal Office of Topography, Switzerland, 1999, Source: Digital Height Model DHM25 (BA4475,JD2982).



(a) Number of expected contour edges. (b) Runtime of the Z-buffer algorithm.

(c) Runtime for visible contour edges. (d) Runtime for hidden-line removal.

Figure 6.17: The number of expected contour edges for the terrain examples in the runtime experiments compared to the runtime results of the three algorithms. Figure (a) shows also the fitting curves for the expected contour edges of Albis, $8.4\sqrt{n}$, and Matterhorn, $14.2\sqrt{n}$.

We give a few runtime results to indicate the order of magnitude for our algorithm. We give the running time in seconds per frame for the computation of the visible contour edges and the computation of the hidden-line image for terrains based on our first implementation. We also give the runtime for a Z-buffer algorithm based on an OpenGL implementation as a reference. The OpenGL implementation uses two light sources, two-sided lighting, and it is not optimized to use triangle strips. It is thus not competitive and only given here as a rough reference. The test data consists of a synthetic terrain in different resolutions, `trn3`, `trn4`, `trn5`, and `trn6` (see Figure 6.14), a data set from Albis and Türlersee in Switzerland, `Albis`, and a data set from the Matterhorn mountain in Switzerland, `Matter`. Both data sets from Switzerland are extracted from the ViRGIS system [157] using different tolerance values from 10 meters to 1 meter. The original base data is © Federal Office of Topography, Switzerland, 1999, Source: Digital Height Model DHM25 (BA4475,JD2982). The test machine is a Linux PC with a Pentium II 300 MHz processor. We use Mesa as OpenGL implementation. The following table lists for each data set the number of edges, the expected number of contour edges and their intersections in the projection, and the runtime range observed in seconds per frame (see also the charts in Figure 6.17).

data set	n	$\overline{n_c}$	$\overline{int_c}$	runtime in seconds per frame Z-buffer	visible contour	hidden line
`trn3`	56	19	2	0.014–0.053	0.0015–0.0025	0.0050–0.0056
`trn4`	208	51	14	0.025–0.067	0.0025–0.0056	0.013–0.014
`trn5`	800	154	73	0.050–0.10	0.0050–0.017	0.029–0.040
`trn6`	3136	542	409	0.15–0.20	0.014–0.059	0.10–0.14
`Albis 10`	5256	399	158	0.20–0.28	0.020–0.067	0.14–0.20
`Albis 5`	13819	845	393	0.46–0.59	0.047–0.13	0.30–0.38
`Albis 2`	34582	1602	829	1.1–1.3	0.11–0.24	0.71–0.87
`Albis 1`	58821	2141	1159	1.7–2.1	0.19–0.53	1.1–1.3
`Matter 10`	18924	1819	1323	0.77–0.83	0.083–0.14	0.40–0.48
`Matter 5`	31227	2542	1874	1.1–1.2	0.12–0.27	0.60–0.76
`Matter 2`	51076	3302	2397	1.7–1.9	0.18–0.40	1.0–1.1
`Matter 1`	65965	3629	2770	2.1–2.4	0.24–0.50	1.1–1.4

Another test machine is a Silicon Graphics Indy with 180 MHz IP22 Mips processor and a plain Indy 8-bit graphics board. We obtain similar results, except that the machine is about a factor of two slower. Noteworthy, the OpenGL implementation of Silicon Graphics is that much faster than the Mesa implementation on Linux that the Z-buffer rendering is about 30% faster on the otherwise slower Indy.

Note that the algorithms cannot be compared, also because their output is too different. But the results indicate that our method can achieve interactive response times.

Chapter 7

Conclusion

The work presented here centers around designing and implementing geometric algorithms and data structures. As a testbed application for the design as well as for its own right, I have studied a new hidden-surface removal algorithm based on contour edges. My interest in the contour-edge approach is motivated by a promising analysis on the expected number of contour edges and the intersections of their projections for a random orthogonal projection.

I have summarized generic programming, a new paradigm well suited to algorithm-centered software design, and have illustrated its use in the domain of geometric algorithms. Generic programming has been used successfully in my work and in designing CGAL. A major contribution is the design of the halfedge data structure underlying the polyhedral surface. The strict definition of the polyhedral surface also has been proven useful in the hidden-surface removal algorithm.

7.1 Results

My results can be seen in three categories: software design, algorithm design, and experimental results. In software design I have successfully applied generic programming to the domain of geometric algorithms and data structures. In particular:

- Generic programming leads to a coherent design of the CGAL kernel and basic library.
- Points and vectors are different types. A symbolic origin converts between them.
- Algorithms access data structures using the iterator concept or the new circulator concept.

- The combinatorial structure of algorithms and data structures is separated from the geometric representation and geometric computations with the new concept of geometric traits classes. Geometric traits classes provide modularization and flexibility while retaining efficiency.
- A design framework for combinatorial data structures. A halfedge data structure has been realized in this framework and successfully used for the polyhedral surface and the planar map in the applications as well as in CGAL.
- A design pattern for granting a safe access to an otherwise protected internal representation.

In algorithm design I have developed a family of algorithms around the idea of contour edges. Several of them follow the sweep-line paradigm to compute the arrangement of projected contour edges in the viewing plane. Let n be the number of edges of a polyhedral surface and let n_c be the number of contour edges for a given viewing transformation. The results are in particular:

- An algorithm to compute the silhouette of a polyhedral surface.
- A hidden-surface removal algorithm for polyhedral surfaces.
- A hidden-surface removal algorithm specialized for terrains.
- A polyhedral surface can be preprocessed independently from the viewing transformation using near linear space such that the contour edges can be reported for a particular viewing transformation in $O(\sqrt{n}\log n + n_\mathrm{c})$ for orthogonal projections or $O(n^{2/3}\,\mathrm{polylog}\, n + n_\mathrm{c})$ for perspective projections. Asymptotically slower, but probably more practical alternatives are also suggested.
- Algorithms to compute exactly the expected number $\overline{n_\mathrm{c}}$ of contour edges, the expected number $\overline{int_\mathrm{c}}$ of intersections of projected contour edges in the viewing plane, and other expectations under random viewing directions for a polyhedral surface.
- A successful application of symbolic perturbation for hidden-surface removal.
- Value-range bounds for the arithmetic of all geometric predicates needed for the hidden-surface removal. Most bounds are tight. The bound for the intersection point of two segments in particular has been shown to be a factor of two smaller than the general bound for intersecting lines.

Let int_c be the number of intersections of projected contour edges in the viewing plane and let k be the output size of the visibility map. The running time T for all three algorithms is the sum of three terms, $T = E(n, n_\mathrm{c}) + V(n, n_\mathrm{c}, int_\mathrm{c}) + F(k)$, where $E(n, n_\mathrm{c})$ denotes the time for extracting the contour edges, $V(n, n_\mathrm{c}, int_\mathrm{c})$ denotes the time for computing the planar map of the projections of the visible contour edges, and $F(k)$ denotes the time for filling the planar map with the visible facets. The term $E(n, n_\mathrm{c})$ is either one of the bounds from above if preprocessing is used, or otherwise it is n. The term $F(k)$ is in $O(k \log k)$ if merged with the sweep-line algorithm which is required for the output sensitive solution of one of the visibility tests. The term $F(k)$ is not needed in the silhouette computation. The running times are summarized in the following table given that the contour edges are already extracted. Let f_loc be the number of face locations needed and $T_\mathrm{ray}(n)$ the query time of a ray-shooting query amidst n triangles in dimension three.

algorithm	time, given the contour edges
silhouette	$O((n_\mathrm{c} + int_\mathrm{c}) \log n_\mathrm{c})$
hidden-surface removal	$O((n_\mathrm{c} + int_\mathrm{c}) \log n_\mathrm{c} + n \log n + f_\mathrm{loc} T_\mathrm{ray}(n) + k \log k)$
— for closed polyhedra	$O((n_\mathrm{c} + int_\mathrm{c}) \log n_\mathrm{c} + n + f_\mathrm{loc} T_\mathrm{ray}(n) + k \log k)$
— for terrains	$O((n_\mathrm{c} + int_\mathrm{c}) \log n_\mathrm{c} + k \log k)$

Experimental results gave evidence that the contour-edge approach is promising. I have computed the expected values $\overline{n_\mathrm{c}}$, $\overline{int_\mathrm{c}}$, and others for a collection of real-world examples of polyhedral surfaces. First runtime results of my implementations indicate also that the contour-edge approach is competitive.

- The expected number $\overline{n_\mathrm{c}}$ of contour edges for polygonal approximations of curved surfaces, here the sphere, behaves like $\Theta(\sqrt{n})$. This finding is confirmed with my measurements on real-world examples. A least-squares fit based on the curve $c_1\sqrt{n} + c_2 n$ resulted in $16.5\sqrt{n} + 0.0056n$.
- The expected number $\overline{int_\mathrm{c}}$ of intersections of projected contour edges is very promising and usually in the order of n, the number of edges.
- I also compare the contour-edge approach with the common heuristic of back-face culling that uses front edges, i.e., all edges except back edges. The number $\overline{int_\mathrm{c}}$ of intersections of contour edges is usually five to ten times better than the number $\overline{int_\mathrm{f}}$ of intersections of front edges.
- The implementations prove that the contour-edge approach can achieve interactive response times.
- A Bentley-Ottmann sweep-line algorithm for computing segment intersections using symbolic perturbation is considerably faster than a sweep-line algorithm treating all degeneracies explicitly in the main loop.

7.2 Discussion

The generic programming paradigm gave me flexibility in my implementations and in CGAL without sacrificing efficiency. All C++ abstraction techniques used here can be resolved at compile time, and, dependent on the quality of the compiler, they are resolved at compile time, which I have verified in the assembler output.

The compliance with the Standard Template Library, STL, is important in order to re-use its generic algorithms and container classes, and to unify the look-and-feel of the design with the C++ standard. As a consequence the design is easy to learn and easy to use for those who are familiar with STL. In addition, synergisms are to be expected with other libraries following this design principle.

The generic programming paradigm and the concepts of STL worked well for the geometric algorithms and data structures in CGAL. CGAL is highly modular, flexible, open and efficient. CGAL interacts smoothly with other libraries following the same concepts or by adaptation with the geometric traits classes, e.g., STL and LEDA.

Generic programming with STL and geometric algorithms from CGAL are easy to teach and easy to use. However, implementing one's own algorithms and data structures in this framework requires knowledge of some of the latest changes in the C++ standard document. Default arguments allow us to hide most of the advanced techniques from the user. For example, iterator traits or geometric traits need not be known by the user in order to use CGAL effectively. A temporary situation is also the lack of compilers that comply to the standard and long compilation times. A more serious problem is the bad support of long identifiers resulting from template names in error messages and in source-level debugging.

The geometric traits classes in particular are a new approach to separate the main control structure of an algorithm from the types and primitive operations used, just in the spirit of generic programming and its focus on algorithmic abstraction. We write an algorithm or a data structure once and use them in different contexts by means of different geometric traits classes. Geometric traits classes serve also as a convenient modularization technique, but in distinction to conventional modularization techniques they support flexibility and optimization at compile time.

Flexibility is also a key tool to achieve robust and efficient implementations. It allows the easy combination of number types, coordinate representation, and appropriate predicates with algorithms and data structures to compose a tailor-made solution for the specific problem at hand. For example, the hidden-surface removal algorithm is compiled for a specific output device resolution to take advantage of the known value ranges, but it can be compiled with a more general (and slower) number type as well to obtain a more general algorithm.

I have presented a design framework for combinatorial data structures such as polyhedral surfaces and planar maps. It can be extended to model the topology of curved-surfaces and can be applied to other combinatorial data structures such as triangle-based structures for triangulations. Such graph-like data structures have not been addressed in generic programming and STL until now.

A proper definition of the modeling space for polyhedral surfaces has been given the strictness of which has been proven useful in our contour-edge based applications.

Various suitable edge-based data structures from the literature have been discussed and the halfedge data structure has been chosen for my implementation. The discussion has revealed several desirable options for the data structure, among them the demand for flexibility in the design, especially the tradeoffs between space and time.

The generic programming paradigm has led to an easy-to-use and flexible high-level interface for polyhedral surfaces featuring handles, iterators, the new concept circulators, and Euler operators. The internal representation can be chosen from a wide range of different halfedge data structures exploiting many tradeoffs between time and storage efficiency, iterator categories, and modifiability. Additional attributes are easy to add. The design is still open to incorporate other techniques as well, such as dynamic type checking at runtime or additional template parameters. The integrity of the internal representation is protected and a mechanism is available that grants safe access to the internal representation.

Based on the sound foundation of the halfedge data structure the planar-map data structure was written in three days. The main effort was to design the set of operations for the incremental construction during the sweep line and the union-find structure needed therein.

The chosen target application is interesting in its own right. The analysis of contour edges identifies interesting application domains for the contour-edge approach such as medical imaging, smooth objects in CAD or modeling, such as surfaces usually obtained from reconstruction methods, or terrain visualization. Even though the worst-case bound in the general case is far from the optimum, I expect for many objects in these application domains a running time close to $O((n_c + int_c) \log n_c + n + k)$, which is the worst-case bound for the hidden-surface removal of terrains with my algorithm (without using preprocessing for the contour edge extraction).

A larger application might well have several hidden-surface removal algorithms at hand and might select the appropriate algorithm depending on some characteristic values of the object. I provide an algorithm that is good for polygonal approximations of smooth objects. Several good characteristic values have been computed in my experimental study. Their computation is too expensive for online decisions, but useful for off-line preprocessing. Other characteristic values, for example estimations of the same values but for a small random sample, may work for online decisions.

Also the other algorithms based on contour edges are useful on their own. First of all, contour edges are a prominent visual feature of geometric objects. Consequently their fast extraction is of interest. The silhouette can be used to cast shadows on other objects. The arrangement of visible contour edges can be used to cast shadows on other objects and to cast intra-object shadows. The set of visible contour edges is one quality criterion used in adaptive surface simplifications methods. Projected contour edges denote discontinuities in X-ray images.

Generic programming has established modularization guidelines, for example iterators and geometric traits classes. These guidelines have kept the complexity of my implementations low while retaining flexibility. My implementations are flexible prototypes that support experimentation. Another example of the successful realization of experimental implementations is our work on conservative predicates [110].

7.3 Future Work

Possible directions for future work are further evaluations of our algorithms and extensions towards more practicability. Among further evaluations are:

- Evaluation of the algorithms with our collection of real-world examples.
- Comparison with other object-space algorithms.
- How often does the facet-location-problem occur in our examples?
- Experimenting with various implementations, for example plain search trees, randomized search trees, or rb-trees, for the sweep line status. Currently an rb-tree is used for the sweep-line status. Similarly for the priority queue.
- Experimental comparison of the data structures proposed for preprocessing the extraction of contour edges.

Among the extensions for more practicability are:

- Zooming and clipping. What is their influence on the initialization of the data-structures and can we benefit from the smaller viewing region with respect to the demand on the value range of the predicates?
- Transparent surfaces.
- Fast estimators to predict online the running time of our algorithms.
- Alternatives for the sweep-line algorithm for computing the arrangement of the visible contour edges. For example, contour edges can be partitioned into a set of closed loops. What is the complexity of a randomized incremental algorithm inserting these loops in a random order?
- Incremental update of the set of contour edges under continuous motion of the view point or of the view direction.
- Symbolic perturbation in CGAL.

Bibliography

[1] P. K. Agarwal. Range searching. In J. E. Goodman and J. O'Rourke, editors, *Handbook of Discrete and Computational Geometry*, chapter 31, pages 575–598. CRC Press LLC, Boca Raton, FL, 1997.

[2] P. K. Agarwal and J. Matoušek. On range searching with semialgebraic sets. *Disc. Comput. Geom.*, 11:393–418, 1994.

[3] P. K. Agarwal, T. Murali, and J. Vitter. Practical techniques for constructing binary space partitions for orthogonal rectangles. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 382–384, 1997.

[4] A. Aggarwal, M. Hansen, and T. Leighton. Solving query-retrieval problems by compacting Voronoi diagrams. In *Proc. 22nd Annu. ACM Sympos. Theory Comput.*, pages 331–340, 1990.

[5] K. Akeley. RealityEngine graphics. In *Computer Graphics (Proc. SIGGRAPH '93)*, volume 27, pages 109–116, 1993.

[6] D. Aliaga, J. Cohen, A. Wilson, H. Zhang, C. Erikson, K. Hoff, T. Hudson, W. Stuerzlinger E. Baker, R. Bastos, M. Whitton, F. Brooks, and D. Manocha. A framework for the real-time walkthrough of massive models. Technical report UNC TR# 98-013, Computer Science Department, University of North Carolina at Chapel Hill, 1998.

[7] P. Alliez, O. Devillers, and J. Snoeyink. Removing degeneracies by perturbing the problem or the world. Technical report 3316, INRIA, Sophia-Antipolis, France, 1997.

[8] N. Amenta. Computational geometry software. In *Handbook of Discrete and Computational Geometry*, pages 951–960. CRC Press, 1997.

[9] A. M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inform. Process. Lett.*, 9(5):216–219, 1979.

[10] A. Appel. The notion of quantitive invisibility and the machine rendering of solids. In *Proc. ACM National Conf.*, pages 387–393, 1967.

[11] J. Arvo, editor. *Graphics Gems II*. Academic Press, 1991.

[12] F. Aurenhammer. A new duality result concerning Voronoi diagrams. *Disc. Comput. Geom.*, 5(3):243–254, 1990.

[13] Avalon. A server of public domain 3d objects: `http://www.viewpoint.com/`

[14] F. Avnaim. *C++GAL: A C++ Library for Geometric Algorithms.* INRIA Sophia-Antipolis, 1994. Manual.

[15] F. Avnaim, J.-D. Boissonnat, O. Devillers, F. P. Preparata, and M. Yvinec. Evaluating signs of determinants using single-precision arithmetic. *Algorithmica*, 17:111–132, 1997.

[16] J. E. Baker, R. Tamassia, and L. Vismara. GeomLib: Algorithm engineering for a geometric computing library, 1997. (Preliminary report).

[17] G. Barequet, C. A. Duncan, M. T. Goodrich, S. Kumar, and M. Pop. Efficient perspective-accurate silhouette computation. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 417–418, 1999. 8th Annual Video Review of Computational Geometry.

[18] J. J. Barton and L. R. Nackman. *Scientific and Engineering C++*. Addison-Wesley, Reading, MA, 1997.

[19] B. G. Baumgart. A polyhedron representation for computer vision. In *National Computer Conference*, pages 589–596, Anaheim, CA, 1975. AFIPS.

[20] H. Bendels, D. W. Fellner, and S. Havemann. Modellierung der Grundlagen: Erweiterbare Datenstrukturen zur Modellierung und Visualisierung polygonaler Welten. In D. W. Fellner, editor, *Modeling – Virtual Worlds – Distributed Graphics*, pages 149–157, Bad Honnef / Bonn, 27.–28. November 1995. `http://www.graphics.uni-bonn.de/CompGraph.ResearchProjects.MRT`

[21] J. L. Bentley and T. A. Ottmann. Algorithms for reporting and counting geometric intersections. *IEEE Trans. Comput.*, C-28(9):643–647, September 1979.

[22] R. Bird and P. Wadler. *Introduction to Functional Programming.* Prentice Hall, 1988.

[23] W. Boehm and H. Prautzsch. *Geometric Concepts for Geometric Design.* A. K. Peters, Wellesley, MA, 1994.

[24] J.-D. Boissonnat, E. de Lange, and M. Teillaud. Slicing Minkowski sums for satellite antenna layout. *Comput. Aided Design*, 30(4):255–265, 1998. Special Issue on Computational Geometry and Computer-Aided Design and Manufacturing.

[25] J.-D. Boissonnat and K. T. G. Dobrindt. On-line construction of the upper envelope of triangles and surface patches in three dimensions. *Computational Geometry: Theory and Applications*, 5:303–320, 1996.

[26] J.-D. Boissonnat and F. P. Preparata. Robust plane sweep for intersecting segments. Report TR 3270, INRIA, Sophia Antipolis, September 1997.

[27] J.-D. Boissonnat and M. Yvinec. *Géométrie Algorithmique*. Ediscience international, Paris, 1995.

[28] J.-D. Boissonnat and M. Yvinec. *Algorithmic Geometry*. Cambridge University Press, UK, 1998. Translated from [27] by H. Brönnimann.

[29] G. Booch and M. Vilot. Simplifying the Booch components. In S. Lippman, editor, *C++ Gems*, pages 59–89. SIGS publications, 1996.

[30] R. Bouma and G. Vaněček Jr. Velocity-based collision detection. In A. Paeth, editor, *Graphics Gems V*. Academic Press, 1995.

[31] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 165–174, 1998.

[32] H. Brönnimann, I. Emiris, V. Pan, and S. Pion. Computing exact geometric predicates using modular arithmetic with single precision. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 174–182, 1997.

[33] H. Brönnimann, L. Kettner, S. Schirra, and R. Veltkamp. Applications of the generic programming paradigm in the design of CGAL. In *Dagstuhl Workshop on Generic Programming*, LNCS. Springer Verlag, 1999. submitted for publication.

[34] H. Brönnimann, S. Schirra, R. Veltkamp, and M. Yvinec, editors. *CGAL Reference Manual*. January 1999. CGAL R1.2. `http://www.cs.uu.nl/CGAL`

[35] K. Q. Brown. Comments on "Algorithms for reporting and counting geometric intersections". *IEEE Trans. Comput.*, C-30:147–148, 1981.

[36] C. Burnikel, J. Könnemann, K. Mehlhorn, S. Näher, S. Schirra, and C. Uhrig. Exact geometric computation in LEDA. In *Proc. 11th Annu. ACM Sympos. Comput. Geom.*, pages C18–C19, 1995.

[37] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 16–23, 1994.

[38] International standard ISO/IEC 14882: Programming languages – C++. American National Standards Institute, 11 West 42nd Street, New York 10036, 1998.

[39] E. Catmull. A subdivision algorithm for computer display of curved surfaces. Technical Report UTEC-CSC-74-133, Dept. Comput. Sci., Univ. Utah, Salt Lake City, 1974.

[40] CG Impact Task Force Report. Application challenges to computational geometry. Technical report TR-521-96, Princeton University, April 1996. `http://www.cs.princeton.edu/~chazelle/taskforce/CGreport.ps.Z`

[41] B. Chazelle and L. J. Guibas. Fractional cascading. *Algorithmica*, 1(3):133–191, 1986.

[42] B. Chazelle and L. J. Guibas. Visibility and intersection problems in plane geometry. *Disc. Comput. Geom.*, 4:551–581, 1989.

[43] B. Chazelle, L. J. Guibas, and D. T. Lee. The power of geometric duality. *BIT*, 25:76–90, 1985.

[44] B. Chazelle and E. Welzl. Quasi-optimal range searching in spaces of finite VC-dimension. *Disc. Comput. Geom.*, 4:467–489, 1989.

[45] K. L. Clarkson and P. W. Shor. Applications of random sampling in computational geometry, II. *Disc. Comput. Geom.*, 4:387–421, 1989.

[46] M. F. Cohen and J. R. Wallace. *Radiosity and Realistic Image Synthesis*. Academic Press Professional, 1993.

[47] J. O. Coplien. Curiously recurring template patterns. *C++ Report*, pages 24–27, February 1995.

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

[49] J. Corney. *3D Modeling with the ACIS Kernel and Toolkit*. John Wiley & Sons, New York, 1997.

[50] F. C. Crow. Shadow algorithms for computer graphics. *Comput. Graph.*, 11(2):242–248, 1977.

[51] M. de Berg. *Ray Shooting, Depth Orders and Hidden Surface Removal*. Lecture Notes in Computer Science. Springer Verlag, 1993.

[52] M. de Berg. Visualization of TINs. Technical report UU-CS-97-22, Department of Computer Science, Utrecht University, The Netherlands, 1997.

[53] M. de Berg, D. Halperin, M. Overmars, J. Snoeyink, and M. van Kreveld. Efficient ray shooting and hidden surface removal. In *Proc. 7th Annu. ACM Sympos. Comput. Geom.*, pages 21–30, 1991.

[54] M. de Berg, M. J. Katz, A. F. van der Stappen, and J. Vleugels. Realistic input models for geometric algorithms. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 294–303, 1997.

[55] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry: Algorithms and Applications*. Springer Verlag, 1997.

[56] P. de Rezende and W. Jacometti. Geolab: An environment for development of algorithms in computational geometry. In *Proc. 5th Canad. Conf. Comput. Geom.*, pages 175–180, 1993.

[57] T. D. DeRose. Geometric programming: A coordinate-free approach. In *Theory and Practice of Geometric Modeling*, pages 291–303, Blaubeuren, FRG (Oct 1988), 1989. Springer-Verlag.

[58] F. Dévai. Quadratic bounds for hidden line elimination. In *Proc. 2nd Annu. ACM Sympos. Comput. Geom.*, pages 269–275, 1986.

[59] 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 Algorithms Data Struct.*, volume 709 of *Lecture Notes Comput. Sci.*, pages 314–324, 1993.

[60] S. E. Dorward. A survey of object-space hidden surface removal. *Int. J. of Computational Geometry & Applications*, 4(3):325–362, 1994.

[61] C. A. Duncan, M. T. Goodrich, and S. Kobourov. Balanced aspect ratio trees: Combining the advantages of k-d trees and octrees. In *10th ACM-SIAM Symposium on Discrete Algorithms (SODA)*, 1999.

[62] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Computer Graphics (Proc. SIGGRAPH '95)*, volume 29, pages 173–182, 1995.

[63] H. Edelsbrunner. *Algorithms in Combinatorial Geometry*, volume 10 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, Germany, 1987.

[64] H. Edelsbrunner and E. P. Mücke. Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104, 1990.

[65] H. Edelsbrunner and R. Waupotitsch. Computing a ham-sandwich cut in two dimensions. *J. Symbolic Comput.*, 2:171–178, 1986.

[66] H. Edelsbrunner and E. Welzl. Halfplanar range search in linear space and $O(n^{0.695})$ query time. *Inform. Process. Lett.*, 23:289–293, 1986.

[67] I. Emiris and J. Canny. A general approach to removing degeneracies. *SIAM J. Comput.*, 24:650–664, 1995.

[68] I. Z. Emiris, J. F. Canny, and R. Seidel. Efficient perturbations for handling geometric degeneracies. *Algorithmica*, 19(1–2):219–242, September 1997.

[69] P. Epstein, J. Kavanagh, A. Knight, J. May, T. Nguyen, and J.-R. Sack. A workbench for computational geometry. *Algorithmica*, 11(4):404–428, April 1994.

[70] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. In M. C. Lin and D. Manocha, editors, *ACM Workshop on Applied Computational Geometry*, pages 191–202, Philadelphia, PA, May, 27–28 1996. Lecture Notes in Computer Science 1148.

[71] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL, the computational geometry algorithms library. *Software – Practice and Experience*, 1999. To appear.

[72] R. Fairley. *Software Engineering Concepts.* McGraw-Hill Series in Software Engineering and Technology. McGraw-Hill, 1985.

[73] E. Flato, D. Halperin, I. Hanniel, and O. Nechushtan. The design and implementation of planar maps in CGAL. In *Proc. 15th European Workshop on Computational Geometry*, Antibes, France, 1999.

[74] J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes. *Computer Graphics — Principles and Practice (2nd Ed.).* Addison-Wesley, 1990.

[75] S. Fortune and C. J. Van Wyk. Static analysis yields efficient exact integer arithmetic for computational geometry. *ACM Transactions on Graphics*, 15(3):223–248, July 1996.

[76] H. Fuchs, G. D. Abrams, and E. D. Grant. Near real-time shaded display of rigid objects. *Comput. Graph.*, 17(3):65–72, 1983. Proc. SIGGRAPH '83.

[77] H. Fuchs, Z. M. Kedem, and B. Naylor. On visible surface generation by a priori tree structures. *Comput. Graph.*, 14(3):124–133, 1980. Proc. SIGGRAPH '80.

[78] Jr. G. Hamlin and C. W. Gear. Raster-scan hidden surface algorithm techniques. In *Computer Graphics (Proc. SIGGRAPH '77)*, volume 11, pages 206–213, 1977.

[79] E. Gamma, R. Helm, R. Johnson, and J. Vlissidis. *Design Patterns – Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[80] B. Gärtner and E. Welzl. Linear programming – randomization and abstract frameworks. In *Proc. 13th Sympos. Theoret. Aspects Comput. Sci.*, volume 1046 of *Lecture Notes Comput. Sci.*, pages 669–687. Springer-Verlag, 1996.

[81] G.-J. Giezeman. *PlaGeo, a library for planar geometry, and SpaGeo, a library for spatial geometry.* Utrecht University, 1994.

[82] A. S. Glassner, editor. *Graphics Gems.* The graphics gems series. Academic Press, 1990.

[83] A. S. Glassner. Maintaining winged-edge models. In J. Arvo, editor, *Graphics Gems II*, pages 191–201. Academic Press, 1991.

[84] R. N. Goldman. Illicit expressions in vector algebra. *ACM Transaction on Graphics*, 4(3):223–243, July 1985.

[85] J. E. Goodman and J. O'Rourke, editors. *Handbook of Discrete and Computational Geometry.* CRC Press LLC, Boca Raton, FL, 1997.

[86] M. Goodrich, L. J. Guibas, J. Hershberger, and P. Tanenbaum. Snap rounding line segments efficiently in two and three dimensions. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 284–293, 1997.

[87] T. Granlund. *GNU MP, The GNU Multiple Precision Arithmetic Library, 2.0.2*, June 1996. manual.

[88] Silicon Graphics. OpenGL optimizer. White paper, Silicon Graphics, Inc., 1997.

[89] P. M. Gruber. Approximation of convex bodies. In P. M. Gruber and J. M. Wills, editors, *Convexity and Its Applications*, pages 131–162. Birkhäuser, 1983.

[90] B. Grünbaum. *Convex Polytopes.* John Wiley & Sons, New York, 1967.

[91] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Transaction on Graphics*, 4(3):74–123, July 1985.

[92] D. Halperin and C. Shelton. A perturbation scheme for spherical arrangements with application to molecular modeling. In *Proc. 13th Annu. ACM Sympos. Comput. Geom.*, pages 183–192, 1997.

[93] J. Hartman and J. Wernecke. *The VRML 2.0 Handbook: Building Moving Worlds on the Web.* Addison-Wesley, 1996.

[94] P. S. Heckbert, editor. *Graphics Gems IV.* The graphics gems series. Academic Press Professional, 1994.

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

[96] C. M. Hoffmann, J. E. Hopcroft, and M. T. Karasick. Robust set operations on polyhedral solids. *IEEE Comput. Graph. Appl.*, 9(6):50–59, November 1989.

[97] M. Hoffmann. Line-Sweep auf einem Gitter, 1996. Diplomarbeit. Freie Univ. Berlin, Germany.

[98] B. Hook. *Building a 3D Game Engine in C++.* John Wiley & Sons, New York, 1995.

[99] H. Hoppe. *Surface reconstruction from unorganized points.* PhD thesis, University of Washington, 1994.

[100] H. Hoppe, T. DeRose, T. Duchamp, M. Halstaed, H. Jin, J. McDonald, J. Schweitzer, and W. Stuetzle. Piecewise smooth surface reconstruction. In *Computer Graphics (Proc. SIGGRAPH '94)*, volume 28, pages 295–302, 1994.

[101] H. Hoppe, T. DeRose, T. Duchamp, J. McDonald, and W. Stuetzle. Mesh optimization. In *Computer Graphics (Proc. SIGGRAPH '93)*, volume 27, pages 19–26, 1993.

[102] C. Insinger. Fahrenheit scene graph: An introduction to scene graph technology. White paper, Silicon Graphics, Inc., 1998.

[103] V. Karamcheti, C. Li, I. Pechtchanski, and C. Yap. A core library for robust numeric and geometric computation. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 351–359, 1999.

[104] M. J. Katz, M. H. Overmars, and M. Sharir. Efficient hidden surface removal for objects with small union size. *Comput. Geom. Theory Appl.*, 2:223–234, 1992.

[105] T. Keffer. The design and architecture of Tools.h++. In S. Lippman, editor, *C++ Gems*, pages 43–57. SIGS publications, 1996.

[106] R. Kempf and C. Frazier, editors. *OpenGL Reference Manual: The Official Reference Document to OpenGL, Version 1.1.* Addison-Wesley, second edition, 1996.

[107] L. Kettner. Designing a data structure for polyhedral surfaces. In *Proc. 14th Annu. ACM Sympos. Comput. Geom.*, pages 146–154, 1998.

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

[109] L. Kettner and E. Welzl. Contour edge analysis for polyhedron projections. In W. Strasser, R. Klein, and R. Rau, editors, *Geometric Modeling: Theory and Practice*, Focus in Computer Graphics, pages 379–394. Springer Verlag, 1997.

[110] L. Kettner and E. Welzl. One sided error predicates in geometric computing. In K. Mehlhorn, editor, *Proc. 15th IFIP World Computer Congress, Fundamentals - Foundations of Computer Science*, pages 13–26, 1998.

[111] J. Keyser, T. Culver, D. Manocha, and S. Krishnan. MAPC: A library for efficient and exact manipulation of algebraic points and curves. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 360–369, 1999.

[112] D. Kirk, editor. *Graphics Gems III.* The graphics gems series. Academic Press, 1992.

[113] R. Klein. *Algorithmische Geometrie.* Addison-Wesley, Bonn, 1997.

[114] R. Klein and W. Straßer. Mesh generation from boundary models with parametric face representations. In C. Hoffmann and J. Rossignac, editors, *Third Symposium on Solid Modeling and Applications*, pages 431–440. ACM Press, May 1995.

[115] G. Knittel and A. Schilling. Eliminating the Z-buffer bottleneck. In *The European Design and Test Conference ED&TC 95*, pages 12–16, Paris, France, 1995. IEEE.

[116] U. Köthe. Reusable software in computer vision. In B. Jähne, H. Haußecker, and P. Geißler, editors, *Handbook on Computer Vision and Applications*. Academic Press, 1999. Volume 3.

[117] K. Kreft and A. Langer. Iterators in the standard C++ library. *C++ Report*, 8(10):27–32, Nov.-Dec. 1996.

[118] D. Kühl and K. Weihe. Iterators and handles for nodes and edges in graphs. Konstanzer Schriften in Mathematik und Informatik Nr. 15, Universität Konstanz, Germany, September 1996. `http://www.informatik.uni-konstanz.de/Schriften`

[119] D. Kühl and K. Weihe. Using design patterns for reusable, efficient implementations of graph algorithms. Konstanzer Schriften in Mathematik und Informatik Nr. 1, Universität Konstanz, Germany, January 1996. `http://www.informatik.uni-konstanz.de/Schriften`

[120] J. Lakos. *Large Scale C++ Software Design*. Addison-Wesley, 1996.

[121] S. Lavallée, R. Szeliski, and L. Brunie. Anatomy-based registration of three-dimensional medical images, range images, x-ray projections, and three-dimensional models using octree-splines. In R. H. Taylor, S. Lavallée, G. C. Burdea, and R. Mösges, editors, *Computer-Integrated Surgery*, chapter 7, pages 115–143. MIT Press, 1995.

[122] S. B. Lippman. *Inside the C++ Object Model*. Addison-Wesley, 1996.

[123] D. Luebke and C. Erikson. View-dependent simplification of arbitrary polygonal environments. In *Proc. SIGGRAPH '97*, pages 199–208, 1997.

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

[125] J. Matoušek. Range searching with efficient hierarchical cuttings. *Disc. Comput. Geom.*, 10(2):157–182, 1993.

[126] J. Matoušek. Geometric range searching. *ACM Comput. Surv.*, 26:421–461, 1994.

[127] J. Matoušek, 1996. Personal communication.

[128] M. McKenna. Worst-case optimal hidden-surface removal. *ACM Transaction on Graphics*, 6(1):19–28, January 1987.

[129] K. Mehlhorn. *Data Structures and Algorithms 3: Multi-dimensional Searching and Computational Geometry*, volume 3 of *EATCS Monographs on Theoretical Computer Science*. Springer-Verlag, Heidelberg, Germany, 1984.

[130] K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the straight & line segment intersection problem. Research Report MPI-I-94-160, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, October 1994.

[131] K. Mehlhorn and S. Näher. LEDA: A platform for combinatorial and geometric computing. *Communications of the ACM*, 38(1):96–102, January 1995.

[132] K. Mehlhorn and S. Näher. *The LEDA Platform for Combinatorial and Geometric Computing.* Cambridge University Press, 1999. In press.

[133] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. *The LEDA User Manual, Version 3.7.* LEDA Software GmbH, 66123 Saarbrücken, Germany, 1998.

[134] J. Meyer-Spradow. Laß Puppen Tanzen: Entwickeln mit der Programmierer-PlayStation 'Net Yaroze'. *c't, Germany*, 18:94–96, 1998.

[135] S. Meyers. *More Effective C++*. Addison-Wesley, 1996.

[136] D. Michelucci. An ϵ arithmetic for removing degeneracies. In *Proc. IEEE Symp. on Computer Arithmetic*, pages 230–237, Bath, July 1995.

[137] V. Milenkovic. Verifiable implementations of geometric algorithms using finite precision arithmetic. In D. Kapur and J. L. Mundy, editors, *Geometric Reasoning*. North-Holland, Amsterdam, The Netherlands, 1988.

[138] V. Milenkovic. Double precision geometry: A general technique for calculating line and segment intersections using rounded arithmetic. In *Proc. 30th Annu. IEEE Sympos. Found. Comput. Sci.*, pages 500–505, 1989.

[139] V. Milenkovic and L. R. Nackman. Finding compact coordinate representations for polygons and polyhedra. In *Proc. 6th Annu. ACM Sympos. Comput. Geom.*, pages 244–252, 1990.

[140] E. P. Mücke. *Shapes and Implementations in Three-Dimensional Geometry.* PhD thesis, Department of Computer Science, University of Illinois, Urbana, IL, 1993. Technical report UIUCDCS-R-93-1836.

[141] D. E. Muller and F. P. Preparata. Finding the intersection of two convex polyhedra. *Theoretical Computer Science*, 7:217–236, 1978.

[142] K. Mulmuley. An efficient algorithm for hidden surface removal. In *Computer Graphics (Proc. SIGGRAPH '89)*, volume 23, pages 379–388, 1989.

[143] K. Mulmuley. A fast planar partition algorithm, I. *J. Symbolic Comput.*, 10(3-4):253–280, 1990.

[144] K. Mulmuley. *Computational Geometry: An Introduction Through Randomized Algorithms.* Prentice Hall, Englewood Cliffs, NJ, 1994.

[145] D. R. Musser and A. Saini. *STL Tutorial and Reference Guide: C++ Programming with the Standard Template Library.* Addison-Wesley, 1996.

[146] D. R. Musser and A. A. Stepanov. Generic programming. In *1st Intl. Joint Conf. of ISSAC-88 and AAEC-6*, pages 13–25. Springer LNCS 358, 1989.

[147] D. R. Musser and A. A. Stepanov. Algorithm-oriented generic libraries. *Software – Practice and Experience*, 24(7):623–642, July 1994.

[148] N. C. Myers. Traits: a new and useful template technique. *C++ Report*, June 1995.

[149] J. Neider, T. Davis, and M. Woo. *OpenGL Programming Guide: The Official Guide to Learning OpenGL, Release 1.* Addison-Wesley, 1993.

[150] M. Newell, R. Newell, and T. Sancha. A new solution to the hidden surface problem. In *Proc. ACM Annu. Conf.*, pages 443–448, 1972.

[151] J. Nievergelt, P. Schorn, M. de Lorenzi, C. Ammann, and A. Brüngger. XYZ: A project in experimental geometric computation. In *Proc. Computational Geometry: Methods, Algorithms and Applications*, volume 553 of *Lecture Notes Comput. Sci.*, pages 171–186. Springer-Verlag, 1991.

[152] M. Nissen and K. Weihe. Combining LEDA with customizable implementations of graph algorithms. Konstanzer Schriften in Mathematik und Informatik Nr. 17, Universität Konstanz, Germany, October 1996. `http://www.informatik.uni-konstanz.de/Schriften`

[153] O. Nurmi. A fast line-sweep algorithm for hidden line elimination. *BIT*, 25:466–472, 1985.

[154] J. O'Rourke. *Computational Geometry in C.* Cambridge University Press, 1994.

[155] T. Ottmann, P. Widmayer, and D. Wood. A worst-case efficient algorithm for hidden line elimination. *Internat. J. Comput. Math.*, 18:93–119, 1985.

[156] M. H. Overmars. Designing the computational geometry algorithms library CGAL. In M. C. Lin and D. Manocha, editors, *ACM Workshop on Applied Computational Geometry*, Philadelphia, PA, May, 27–28 1996. Lecture Notes in Computer Science 1148.

[157] R. B. Pajarola. *Access to large scale Terrain and Image Databases in Geoinformation Systems.* PhD thesis, ETH Zurich, Switzerland, 1998. DISS. ETH No. 12729.

[158] M. Phillips. *Geomview Manual: Geomview Version 1.5 for Silicon Graphics Workstations*. The Geometry Center, University of Minnesota, October 1994.

[159] F. P. Preparata and M. I. Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, New York, NY, 1985.

[160] S. Raab. Controlled perturbation for arrangements of polyhedral surfaces with application to swept volumes. In *Proc. 15th Annu. ACM Sympos. Comput. Geom.*, pages 163–172, 1999.

[161] J. H. Reif and S. Sen. An efficient output-sensitive hidden-surface removal algorithms and its parallelization. In *Proc. 4th Annu. ACM Sympos. Comput. Geom.*, pages 193–200, 1988.

[162] F. M. Richards. Areas, volumes, packing and protein structure. *Annu. Rev. Biophys. Bioeng.*, 6:151–176, 1977.

[163] J. Rossignac and P. Borrel. Multi-resolution 3D approximations for rendering complex scenes. In B. Falcidieno and T. L. Kunii, editors, *Modeling in Computer Graphics*, pages 455–465. Springer Verlag, 1993. IFIP TC5 WG 5.10 conference held in Genova, Italy.

[164] J.-R. Sack and J. Urrutia, editors. *Handbook of Computational Geometry*. North-Holland, 1999. To appear.

[165] S. Schirra. A case study on the cost of geometric computing. In *Proc. of ALENEX'99*, 1999.

[166] S. Schirra. Precision and robustness issues in geometric computation. In *Handbook on Computational Geometry*. Elsevier Science Publishers, Amsterdam, The Netherlands, 1999.

[167] A. Schmitt. Time and space bounds for hidden line and hidden surface algorithms. In *Proc. Eurographics 81*, pages 43–56, Amsterdam, The Netherlands, 1981. North-Holland.

[168] P. Schorn. Implementing the XYZ GeoBench: A programming environment for geometric algorithms. In *Proc. Computational Geometry: Methods, Algorithms and Applications*, volume 553 of *Lecture Notes Comput. Sci.*, pages 187–202. Springer-Verlag, 1991.

[169] P. Schorn. Degeneracy in geometric computation and the perturbation approach. *The Computer Journal*, 37(1):35–42, 1994.

[170] R. Seidel. The meaning and nature of perturbations in geometric computing. In *Proc. 11th Symp. on Theoretical Aspects of Computer Science (STACS 94)*, pages 2–17, 1994.

[171] C. H. Séquin and P. R. Wensley. Visible feature return at object resolution. *IEEE Computer Graphics and Application*, 5:37–50, May 1985.

[172] J. R. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Disc. Comput. Geom.*, 18(3):305–363, October 1997.

[173] Silicon Graphics Computer Systems, Inc. Standard template library programmer's guide. `http://www.sgi.com/Technology/STL/`, 1997.

[174] Silicon Graphics, Inc. *OpenGL Optimizer Programmer's Guide: An Open API for Large-Model Visualization*, 1998. Manual.

[175] H. Sowizral, Rushforth K, and M. Deering. *The Java 3D API Specification.* The Java Series. Addison-Wesley, 1998.

[176] E. Steinitz and H. Rademacher. *Vorlesung über die Theorie der Polyeder (unter Einschluß der Elemente der Topologie).* Springer, Berlin, 1934.

[177] A. A. Stepanov and M. Lee. The standard template library. `http://www.cs.rpi.edu/~musser/doc.ps`, October 1995.

[178] B. Stroustrup. *The C++ Programming Language.* Addison-Wesley, 3rd edition, 1997.

[179] I. E. Sutherland, R. F. Sproull, and R. Schumaker. A characterization of ten hidden-surface algorithms. *ACM Computing Surveys*, 6(1):1–55, 1974.

[180] M. Teillaud. 3d triangulations in CGAL. In *Proc. 15th European Workshop Computational Geometry*, Antibes, France, 1999.

[181] S. Teller. Computing the antipenumbra of an area light source. In *Computer Graphics (Proc. SIGGRAPH '92)*, volume 26, pages 139–147, 1992.

[182] M. van Kreveld. Algorithms for triangulated terrains. In *Proc. XXIV-th SOFSEM*, number 1338 in Lecture Notes Comput. Sci., pages 19–36. Springer-Verlag, 1997.

[183] T. Veldhuizen. Expressions templates. *C++ Report*, pages 26–31, June 1995.

[184] T. Veldhuizen. Template metaprograms. *C++ Report*, pages 36–43, May 1995.

[185] T. Veldhuizen and K. Ponnambalam. Linear algebra with C++ template metaprograms. *Dr. Dobb's Journal*, August 1996.

[186] R. C. Veltkamp. Generic programming in CGAL, the computational geometry algorithms library. In *Proceedings of the 6th Eurographics Workshop on Programming Paradigms in Graphics*, 1997.

[187] Viewpoint DataLabs, Orem UT, USA. Commercial 3d objects and the Avalon server of public domain 3d objects: `http://www.viewpoint.com/`

[188] B. Wallis. Forms, vectors, and transforms. In A. S. Glassner, editor, *Graphics Gems*, pages 533–538. Academic Press, 1990.

[189] K. Weihe. Reuse of algorithms: Still a challenge to object-oriented programming. In *Proceedings of the 12th Annual ACM Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA'97)*, pages 34–48, 1997.

[190] K. Weiler. Edge-based data structures for solid modeling in curved-surface environments. *IEEE Computer Graphics and Application*, 5(1):21–40, January 1985.

[191] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In H. Maurer, editor, *New Results and New Trends in Computer Science*, volume 555 of *Lecture Notes in Computer Science*, pages 359–370. Springer-Verlag, 1991.

[192] E. Welzl. Gram's equation – a probabilistic proof. In *Results and Trends in Theoretical Computer Science*, pages 422–424. Springer Verlag, 1994. *Lecture Notes in Computer Science,* 812.

[193] E. Welzl and B. Wolfers. Surface reconstruction between simple polygons via angle criteria. *J. Symbolic Computation*, 17:351–369, 1994.

[194] J. Wernicke. *The Inventor Mentor: Programming Object-Oriented 3D Graphics with Open Inventor, Release 2.* Addison-Wesley, 1994.

[195] J. Xia and A. Varshney. Dynamic view-dependent simplification for polygonal models. In *IEEE Visualization '96 Proceedings*, pages 327–334, San Francisco, CA, October 1996. ACM/SIGGRAPH Press.

[196] C. Yap. Towards exact geometric computation. *Comput. Geom. Theory Appl.*, 7(1):3–23, 1997.

[197] C.-K. Yap. Symbolic treatment of geometric degeneracies. *J. Symbolic Comput.*, 10:349–370, 1990.

Index

Curriculum Vitae

Lutz Kettner

born on February 2, 1967 in Kiel, Germany

1977–1986	Ernst-Barlach Gymnasium in Kiel, Germany
1986–1988	Civil service in Kiel, Germany
1988–1994	Student in Computer Science University of Karlsruhe, Germany Diplom-Informatiker (Dipl.-Inform.)
1994–1996	PhD student in Computer Science Freie Universität Berlin, Germany
1996–1999	PhD student in Computer Science ETH Zurich, Switzerland