68 TWO COMPUTATIONAL GEOMETRY LIBRARIES: LEDA AND CGAL

Michael Hoffmann, Lutz Kettner, and Stefan Näher

INTRODUCTION

Over the past decades, two major software libraries that support a wide range of geometric computing have been developed: LEDA, the **L**ibrary of **E**fficient **D**ata Types and **A**lgorithms, and CGAL, the **C**omputational **G**eometry **A**lgorithms **L**ibrary. We start with an introduction of common aspects of both libraries and major differences. We continue with sections that describe each library in detail.

Both libraries are written in C++. LEDA is based on the object-oriented paradigm and CGAL is based on the generic programming paradigm. They provide a collection of flexible, efficient, and correct software components for computational geometry. Users should be able to easily include existing functionality into their programs. Additionally, both libraries have been designed to serve as platforms for the implementation of new algorithms.

Correctness is of crucial importance for a library, even more so in the case of geometric algorithms where correctness is harder to achieve than in other areas of software construction. Two well-known reasons are the *exact arithmetic assumption* and the *nondegeneracy assumption* that are often used in geometric algorithms. However, both assumptions usually do not hold: floating-point arithmetic is not exact and inputs are frequently degenerate. See Chapter 45 for details.

EXACT ARITHMETIC

There are basically two scientific approaches to the exact arithmetic problem. One can either design new algorithms that can cope with inexact arithmetic or one can use exact arithmetic. Instead of requiring the arithmetic itself to be exact, one can guarantee correct computations if the so-called *geometric primitives* are exact. So, for instance, the predicate for testing whether three points are collinear must always give the right answer. Such an exact primitive can be efficiently implemented using floating-point filters or lazy evaluation techniques.

This approach is known as the exact geometric computing paradigm and both libraries, LEDA and CGAL, adhere to this paradigm. However, they also offer straight floating-point implementations.

DEGENERACY HANDLING

An elegant (theoretical) approach to the degeneracy problem is ***symbolic perturbation***. But this method of forcing input data into general position can cause serious problems in practice. In many cases, it increases the complexity of (interme-

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

diate) results considerably; and furthermore, the final limit process turns out to be difficult in particular in the presence of combinatorial structures. For this reason, both libraries follow a different approach. They cope with degeneracies directly by treating the degenerate case as the "normal" case. This approach proved to be effective for many geometric problems.

However, symbolic perturbation is used in some places. For example, in CGAL the 3D Delaunay triangulation uses it to realize consistent point insert and removal functions in the degenerate case of more than four points on a sphere [DT11].

GEOMETRIC PROGRAMMING

Both CGAL and LEDA advocate *geometric programming*. This is a style of higher-level programming that deals with geometric objects and their corresponding primitives rather than working directly on coordinates or numerical representations. In this way, for instance, the machinery for the exact arithmetic can be encapsulated in the implementation of the geometric primitives.

COMMON ROOTS AND DIFFERENCES

LEDA is a general-purpose library of algorithms and data structures, whereas CGAL is focused on geometry. They have a different look and feel and different design principles, but they are compatible with each other and can be used together. A LEDA user can benefit from more geometry algorithms in CGAL, and a CGAL user can benefit from the exact number types and graph algorithms in LEDA, as will be detailed in the individual sections on LEDA and CGAL.

CGAL started six years after LEDA. CGAL learned from the successful decisions and know-how in LEDA (also supported by the fact that LEDA's founding institute is a partner in developing CGAL). The later start allowed CGAL to rely on better C++ language support, e.g., with templates and traits classes, which led the developers to adopt the *generic programming paradigm* and shift the design focus more toward flexibility.

Successful spin-off companies have been created around both LEDA[1] and CGAL[2]. After an initial free licensing for academic institutions, all LEDA licenses are now fee-based. In contrast, CGAL is freely available under the GPL/LGPL [GPL07] since release 4.0 (March 2012). Users who consider the open source license to be too restrictive can also obtain a commercial license.

GLOSSARY

Exact arithmetic: The foundation layer of the *exact computation paradigm* in computational geometry software that builds correct software layer by layer. Exact arithmetic can be as simple as a built-in integer type as long as its precision is not exceeded or can involve more complex number types representing expression DAGs, such as, `leda::real` from LEDA [BFMS00] or `Expr` from CORE [KLPY99].

[1] Algorithmic Solutions Software GmbH `<www.algorithmic-solutions.com>`.
[2] GeometryFactory Sarl `<www.geometryfactory.com>`.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

Floating-point filter: A technique that speeds up exact computations for common easy cases; a fast floating-point interval arithmetic is used unless the error intervals overlap, in which case the computation is repeated with exact arithmetic.

Coordinate representation: Cartesian and homogeneous coordinates are supported by both libraries. Homogeneous coordinates are used to optimize exact rational arithmetic with a common denominator, not for projective geometry.

Geometric object: The atomic part of a geometric kernel. Examples are points, segments, lines, and circles in 2D, planes, tetrahedra, and spheres in 3D, and hyperplanes in dD. The corresponding data types have value semantics; variants with and without reference-counted representations exist.

Predicate: Geometric primitive returning a value from a finite domain that expresses a geometric property of the arguments (geometric objects), for example, `CGAL::do_intersect(`p`,`q`)` returning a Boolean or `leda::orientation(`p`,`q`,`r`)` returning the sign of the area of the triangle (p, q, r). A ***filtered predicate*** uses a floating-point filter to speed up computations.

Construction: Geometric primitive constructing a new object, such as the point of intersection of two non-parallel straight lines.

Geometric kernel: The collection of geometric objects together with the related predicates and constructions. A ***filtered kernel*** uses filtered predicates.

Program checkers: Technique for writing programs that check the correctness of their output. A checker for a program computing a function f takes an instance x and an output y. It returns true if $y = f(x)$, and false otherwise.

68.1 LEDA

LEDA aims at being a comprehensive software platform for the entire area of combinatorial and geometric computing. It provides a sizable collection of data types and algorithms. This collection includes most of the data types and algorithms described in the textbooks of the area ([AHU74, CLR90, Kin90, Meh84, NH93, O'R94, Tar83, Sed91, Wyk88, Woo93]).

A large number of academic and industrial projects from almost every area of combinatorial and geometric computing have been enabled by LEDA. Examples are graph drawing, algorithm visualization, geographic information systems, location problems, visibility algorithms, DNA sequencing, dynamic graph algorithms, map labeling, covering problems, railway optimization, route planning, computational biology and many more. See `<www.algorithmic-solutions.com/leda/projects/index.html>` for a list of industrial projects based on LEDA.

The LEDA project was started in the fall of 1988 by Kurt Mehlhorn and Stefan Näher. The first six months were devoted to the specification of different data types and selecting the implementation language. At that time the item concept arose as an abstraction of the notion "pointer into a data structure." Items provide direct and efficient access to data and are similar to iterators in the standard template library. The item concept worked successfully for all test cases and is now used for most data types in LEDA. Concurrently with searching for the correct specifications, existing programming languages were investigated for their suitability as an

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

implementation platform. The language had to support abstract data types and type parameters (genericity) and should be widely available. Based on the experiences with different example programs, C++ was selected because of its flexibility, expressive power, and availability.

We next discuss some of the general aspects of the LEDA system.

EASE OF USE

The LEDA library is easy to use. In fact, only a small fraction of the users are algorithm experts and many users are not even computer scientists. For these users the broad scope of the library, its ease of use, and the correctness and efficiency of the algorithms in the library are crucial. The LEDA manual [MNSU] gives precise and readable specifications for the data types and algorithms mentioned above. The specifications are short (typically not more than a page), general (so as to allow several implementations) and abstract (so as to hide all details of the implementation).

EXTENSIBILITY

Combinatorial and geometric computing is a diverse area and hence it is impossible for a library to provide ready-made solutions for all application problems. For this reason it is important that LEDA is easily extensible and can be used as a platform for further software development. In many cases LEDA programs are very close to the typical textbook presentation of the underlying algorithms. The goal is the equation: *Algorithm + LEDA = Program.*

LEDA *extension packages* (LEPs) extend LEDA into particular application domains and areas of algorithmics not covered by the core system. LEDA extension packages satisfy requirements, which guarantee compatibility with the LEDA philosophy. LEPs have a LEDA-style documentation, they are implemented as platform independent as possible, and the installation process permits a close integration into the LEDA core library. Currently, the following LEPs are available: PQ-trees, dynamic graph algorithms, a homogeneous d-dimensional geometry kernel, and a library for graph drawing.

CORRECTNESS

Geometric algorithms are frequently formulated under two unrealistic assumptions: computers are assumed to use exact real arithmetic (in the sense of mathematics) and inputs are assumed to be in general position. The naive use of floating-point arithmetic as an approximation to exact real arithmetic very rarely leads to correct implementations. In a sequence of papers [BMS94b, See94, MN94b, BMS94a, FGK$^+$00], these degeneracy and precision issues were investigated and LEDA was extended based on this theoretical work. It now provides exact geometric kernels for 2D and higher-dimensional computational geometry [MMN$^+$98], and also correct implementations for basic geometric tasks, e.g., 2D convex hulls, Delaunay diagrams, Voronoi diagrams, point location, line segment intersection, and higher-dimensional convex hulls and Delaunay triangulations.

Programming is a notoriously error-prone task; this is even true when pro-

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

gramming is interpreted in a narrow sense: translating a (correct) algorithm into a program. The standard way to guard against coding errors is program testing. The program is exercised on inputs for which the output is known by other means, typically as the output of an alternative program for the same task. Program testing has severe limitations. It is usually only performed during the testing phase of a program. Also, it is difficult to determine the "correct" suite of test inputs. Even if appropriate test inputs are known it is usually difficult to determine the correct outputs for these inputs: alternative programs may have different input and output conventions or may be too inefficient to solve the test cases.

Given that program verification—i.e., formal proof of correctness of an implementation—will not be available on a practical scale for some years to come, *program checking* has been proposed as an extension to testing [BK95, BLR93]. The cited papers explored program checking in the area of algebraic, numerical, and combinatorial computing. In [MNS+99, MM96, HMN96] program checkers are presented for planarity testing and a variety of geometric tasks. LEDA uses program checkers for many of its implementations. A more general approach (*Certifying Algorithms*) was introduced in [MMNS11].

68.1.1 THE STRUCTURE OF LEDA

LEDA uses templates for the implementation of parameterized data types and for generic algorithms. However, it is not a pure template library and therefore is based on an object code library of precompiled code. Programs using LEDA data types or algorithms have to include the appropriate LEDA header files in their source code and must link to this library (*libleda*). See the LEDA user manual ([MNSU] or the LEDA book ([MN00]) for details.

68.1.2 GEOMETRY KERNELS

LEDA offers kernels for 2D and 3D geometry, a kernel of arbitrary dimension is available as an extension package. In either case there exists a version of the kernel based on floating-point Cartesian coordinates (called *float-kernel*) as well as a kernel based on rational homogeneous coordinates (called *rat-kernel*). All kernels provide a complete collection of geometric objects (points, segments, rays, lines, circles, simplices, polygons, planes, etc.) together with a large set of geometric primitives and predicates (orientation of points, side-of-circle tests, side-of-hyperplane, intersection tests and computation, etc.). For a detailed discussion and the precise specification, see Chapter 9 of the LEDA book ([MN00]). Note that only for the rational kernel, which is based on exact arithmetic and floating-point filters, all operations and primitives are guaranteed to compute the correct result.

68.1.3 DATA STRUCTURES

In addition to the basic kernel data structures LEDA provides many advanced data types for computational geometry. Examples include:

- A general polygon type (`gen_polygon` or `rat_gen_polygon`) with a complete set of Boolean operations. Its implementation is based on efficient and robust

plane sweep algorithms for the construction of the arrangement of a set of straight line segments (see [MN94a] and [MN00, Ch. 10.7]).

- Two- and higher-dimensional geometric tree structures, such as range, segment, interval and priority search trees.
- Partially and fully persistent search trees.
- Different kinds of geometric graphs (triangulations, Voronoi diagrams, and arrangements).
- A dynamic `point_set` data type supporting update, search, closest point, and different types of range query operations on one single representation based on a dynamic Delaunay triangulation (see [MN00, Ch. 10.6]).

68.1.4 ALGORITHMS

The LEDA project never had the goal of providing a complete collection of the algorithms from computational geometry (nor for other areas of algorithms). Rather, it was designed and implemented to establish a *platform* for combinatorial and geometric computing enabling programmers to implement these algorithms themselves more easily and customized to their particular needs. But of course the library already contains a considerable number of basic geometric algorithms. Here we give a brief overview and refer the reader to the user manual for precise specifications and to Chapter 10 of the LEDA-book ([MN00]) for detailed descriptions and analyses of the corresponding implementations. The current version of LEDA offers different implementation of algorithms for the following 2D geometric problems:

- convex hull algorithms (also 3D)
- halfplane intersection
- (constraint) triangulations
- closest and farthest Delaunay and Voronoi diagrams
- Euclidean minimum spanning trees
- closest pairs
- Boolean operations on generalized polygons
- segment intersection and construction of line arrangements
- Minkowski sums and differences
- nearest neighbors and closest points
- minimum enclosing circles and annuli
- curve reconstruction

68.1.5 VISUALIZATION (GeoWin)

In computational geometry, visualization and animation of programs are important for the understanding, presentation, and debugging of algorithms. Furthermore, the

animation of geometric algorithms is cited as among the strategic research directions in this area. *GeoWin* [BN02] is a generic tool for the interactive visualization of geometric algorithms. *GeoWin* is implemented as a C++ data type. Its design and implementation was influenced by LEDA's graph editor *GraphWin* ([MN00, Ch. 12]). Both data types support a number of programming styles which have been shown to be very useful for the visualization and animation of algorithms. The animations use *smooth transitions* to show the result of geometric algorithms on dynamic user-manipulated input objects, e.g., the Voronoi diagram of a set of moving points or the result of a sweep algorithm that is controlled by dragging the sweep line with the mouse (see Figure 68.1.1).

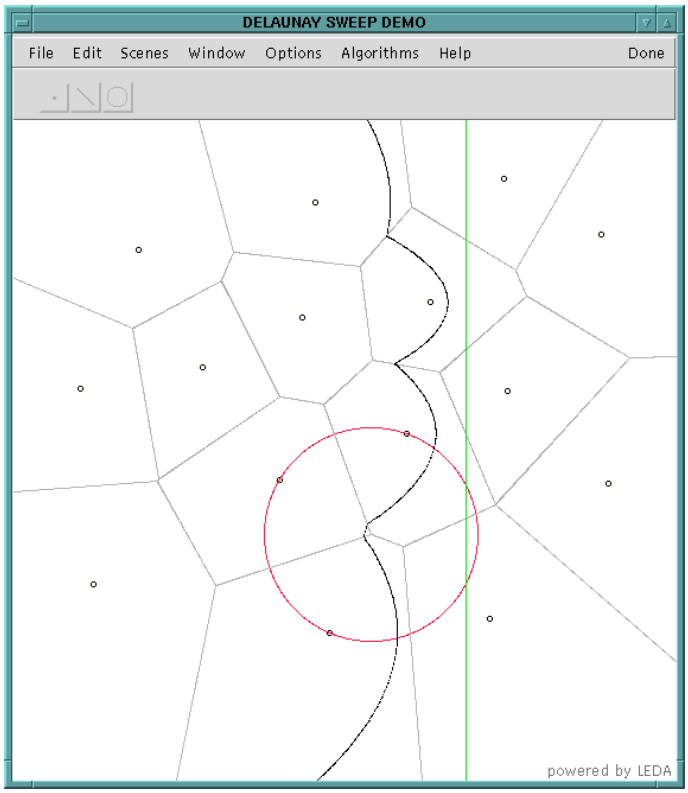


FIGURE 68.1.1
GeoWin animating Fortune's sweep algorithm.

A GeoWin maintains one or more geometric scenes. A geometric *scene* is a collection of geometric objects of the same type. A collection is simply either a standard C++ list (STL-list) or a LEDA-list of objects. GeoWin requires that the objects provide a certain functionality, such as stream input and output, basic geometric transformations, drawing and input in a LEDA window. A precise definition of the required operations can be found in the manual pages [MNSU]. GeoWin can be used for any collection of basic geometric objects (geometry kernel) fulfilling these requirements. Currently, it is used to visualize geometric objects and algorithms from both the CGAL and LEDA libraries.

The visualization of a scene is controlled by a number of attributes, such as color, line width, line style, etc. A scene can be subject to user interaction and it may be defined from other scenes by means of an algorithm (a C++ function). In the latter case the scene (also called the *result scene*) may be recomputed whenever one of the scenes on which it depends is modified. There are three main modes for recomputation: user-driven, continuous, and event-driven.

GeoWin has both an interactive and a programming interface. The interactive interface supports the interactive manipulation of input scenes, the change of geometric attributes, and the selection of scenes for visualization.

68.1.6 PROGRAM EXAMPLES

We now give two programming examples showing how LEDA can be used to implement basic geometric algorithms in an elegant and readable way. The first example is the computation of the *upper convex hull* of a point set in the plane. It uses points and the orientation predicate and lists from the basic library. The second example shows how the LEDA *graph* data type is used to represent triangulations in the implementation of a function that turns an arbitrary triangulation into a Delaunay triangulation by edge flips. It uses points, lists, graphs, and the side-of-circle predicate.

UPPER CONVEX HULL

In our first example we show how to use LEDA for computing the upper convex hull of a given set of points. We assume that we are in LEDA's namespace, otherwise all LEDA names would have to be used with the prefix `leda::`. Function `UPPER_HULL` takes a list L of rational points (type `rat_point`) as input and returns the list of points of the upper convex hull of L in clockwise ordering from left to right. The algorithm is a variant of Graham's Scan [Gra72].

First we sort L according to the lexicographic ordering of the Cartesian coordinates and remove multiple points. If the list contains not more than two points after this step we stop. Before starting the actual Graham Scan we first skip all initial points lying on or below the line connecting the two extreme points. Then we scan the remaining points from left to right and maintain the upper hull of all points seen so far in a list called *hull*. Note however that the last point of the hull is not stored in this list but in a separate variable p. This makes it easier to access the last two hull points as required by the algorithm. Note also that we use the rightmost point as a sentinel avoiding the special case that *hull* becomes empty.

```
list<rat_point> UPPER_HULL(list<rat_point> L) {
    L.sort();
    L.unique();

    if (L.length() <= 2) return L;

    rat_point p_min = L.front(); // leftmost point
    rat_point p_max = L.back();  // rightmost point

    list<rat_point> hull;        // result list
    hull.append(p_max);          // use rightmost point as sentinel
    hull.append(p_min);          // first hull point
```

```
    // goto first point p above (p_min,p_max)
    while (! L.empty() && ! left_turn(p_min, p_max, L.front())) L.pop();
    if (L.empty()) {              // upper hull consists of only 2 points
        hull.reverse();
        return hull;
    }

    rat_point p = L.pop();        // second (potential) hull point
    rat_point q;
    forall(q,L) {
        while (! right_turn(hull.back(), p, q)) p = hull.pop_back();
        hull.append(p);
        p = q;
    }
    hull.append(p);               // add last hull point
    hull.pop();                   // remove sentinel
    return hull;
}
```

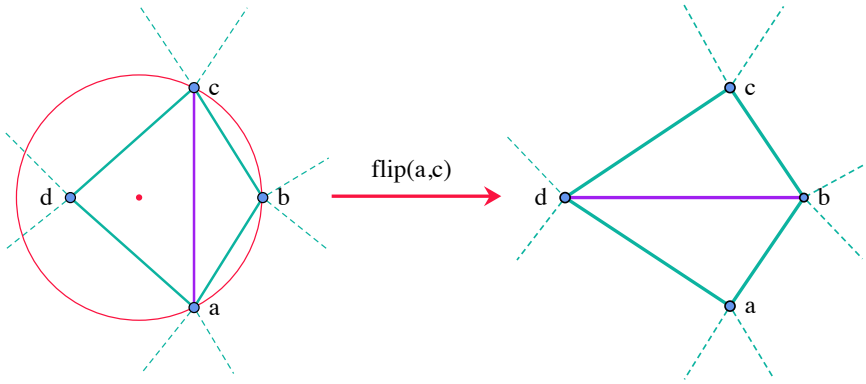
DELAUNAY FLIPPING

LEDA represents triangulations by bidirected plane graphs (from the graph library) whose nodes are labeled with points and whose edges may carry additional information, e.g., integer flags indicating the type of edge (hull edge, triangulation edge, etc.). All edges incident to a node v are ordered in counterclockwise ordering and every edge has a reversal edge. In this way the faces of the graph represent the triangles of the triangulation. The graph type offers methods for iterating over the nodes, edges, and adjacency lists of the graph. In the case of plane graphs there are also operations for retrieving the reverse edge and for iterating over the edges of a face. Furthermore, edges can be moved to new nodes. This graph operation is used in the following program to implement edge flips.

Function `DELAUNAY_FLIPPING` takes as input an arbitrary triangulation and turns it into a Delaunay triangulation by the well-known flipping algorithm. This algorithm performs a sequence of local transformations as shown in Figure 68.1.2 to establish the Delaunay property: for every triangle the circumscribing circle does not contain any vertex of the triangulation in its interior. The test whether an edge has to be flipped or not can be realized by a so-called *side_of_circle* test. This test takes four points a, b, c, d and decides on which side of the oriented circle through the first three points a,b, and c the last point d lies. The result is positive or negative if d lies on the left or on the right side of the circle, respectively, and the result is zero if all four points lie on one common circle. The algorithm uses a list of candidates which might have to be flipped (initially all edges). After a flip the four edges of the corresponding quadrilateral are pushed onto this candidate list. Note that `G[v]` returns the position of node v in the triangulation graph G. A detailed description of the algorithm and its implementation can be found in the LEDA book ([MN00]).

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

FIGURE 68.1.2
Flipping to establish the Delaunay property.




```
void DELAUNAY_FLIPPING(GRAPH<rat_point, int>& G) {
    list<edge> S = G.all_edges();
    while (! S.empty()) {
        edge e = S.pop();
        edge r = G.rev_edge(e);

        edge e1 = G.face_cycle_succ(r);    // e1,e2,e3,e4: edges of quadrilateral
        edge e2 = G.face_cycle_succ(e1);   // with diagonal e
        edge e3 = G.face_cycle_succ(e);
        edge e4 = G.face_cycle_succ(e3);

        rat_point a = G[G.source(e1)];     // a,b,c,d: corners of quadrilateral
        rat_point b = G[G.target(e1)];
        rat_point c = G[G.source(e3)];
        rat_point d = G[G.target(e3)];

        if (side_of_circle(a,b,c,d) > 0) {
            S.push(e1); S.push(e2); S.push(e3); S.push(e4);
            G.move_edge(e,e2,source(e4)); // flip diagonal
            G.move_edge(r,e4,source(e2));
         }
     }
}
```

68.2 CGAL

The development of CGAL, the Computational Geometry Algorithms Library, started in 1995 by a consortium of seven sites from Europe and Israel, funded by European research programs. The central goal of CGAL is to

> *make the large body of geometric algorithms developed in the field of computational geometry available for industrial application.*

The main design goals are correctness, flexibility, efficiency, and ease of use. The focus is on a broad foundation in computational geometry. Important related issues, for example visualization, are supported with standard formats and interfaces.

The first public release 0.9 appeared in June 1997. Since 2009 a biannual release cycle has been established and since 2015 the current development version is publicly available at `<github.com/CGAL>`. About 30 developers work on CGAL, though many of them part-time only. A release provides 4–5 new major features on average, such as new packages or significant updates to existing packages. With release 3.0 in 2003, CGAL became an open source project inviting everybody to join and has since successfully attracted more than 20 feature submissions from outside the founding institutes.

Nowadays CGAL is a standard tool in the area. More than 200 commercial customers work with CGAL and a list of more than 100 projects from diverse areas using CGAL can be found at `<www.cgal.org/projects.html>`. The presentation here is based on CGAL release 4.7 from October 2015, available from CGAL's home page `<www.cgal.org>`.

LIBRARY STRUCTURE

CGAL is structured in three layers: the layer of *algorithms and data structures*, which builds on the *geometric traits* layer with representations for geometric objects and primitive operations on these representations. The geometric traits layer in turn builds on the layer of *arithmetic and algebra* with concepts for algebraic structures and number types modeling these structures. Orthogonally, there is a *support library* layer with geometric object generators, file I/O, visualization, and other nongeometric functionality.

GENERIC PROGRAMMING IDIOMS

Concept: A formal hierarchy of polymorphic abstract requirements on data types.

Model for a concept: A type that fulfils all requirements of that concept and can therefore be used in places where the concept was requested.

Function object: An object that implements a function, e.g., as a C++ class with an `operator()`. It is more efficient and type-safe compared to a C function pointer or object-oriented class hierarchies.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

FLEXIBILITY

Following the generic programming paradigm, CGAL is highly *modular* and its different parts are independent of each other. The algorithms and data structures in CGAL are *adaptable* to already existing user code; see the geometric traits class example on page 1815. The library is *extendible*, as users can add implementations in the same style as CGAL. The library is *open* and supports important standards, such as the C++ standard with its standard library, and other established libraries, such as BOOST [Sch14], LEDA, EIGEN [GJ+10], and GMP [Gt14].

CORRECTNESS

CGAL addresses robustness problems by relying on exact arithmetic and explicit degeneracy handling. There is a well-established software process and communication channels set up for the distributed developer community, including a distributed system for revision management, bug and feature tracking. Every night an automatic test-suite is run on all supported platforms and compilers. An editorial board reviews new submissions and supervises design homogeneity.

EASE OF USE

Users with a base knowledge of C++ and generic programming experience a smooth learning curve with CGAL. Many concepts are familiar from the C++ standard library, and the powerful flexibility is often hidden behind sensible defaults. A novice reader should not be discouraged by some of the advanced examples illustrating CGAL's power. CGAL has a uniform design, aims for minimal interfaces, yet rich and complete functionality in computational geometry.

EFFICIENCY

CGAL uses C++ templates to realize most of its flexibility at compile time. That enables flexibility at places normally not considered because of runtime costs, e.g., on the number-type level. Furthermore, choices such as tradeoffs between space and time in some data structures, or between different number types of different power and speed can be made at the application level rather than in the library. This also encourages experimental research.

68.2.1 GEOMETRIC KERNELS

CGAL offers a wide variety of kernels. The *linear* kernels are mostly concerned with linearly bounded objects, such as hyperplanes and simplices. The geometric objects along with some of the available predicates and constructions—classified according to dimension—are summarized in Table 68.2.1.

For circles and spheres an algebraic description involves polynomials of degree two. Therefore, intersections involving circles and spheres do not admit an exact rational representation in general. The *circular 2D kernel* [EKP+04, DFMT02]

TABLE 68.2.1 Linear kernel objects with selected predicates and constructions.

DIM	GEOMETRIC OBJECTS	PREDICATES	CONSTRUCTIONS
all	Point, Vector, Direction, Line, Ray, Segment, Aff_transformation	compare_lexicographically, do_intersect, orientation	intersection, midpoint transform, squared_distance
2	Triangle, Iso_rectangle, Bbox, Circle	collinear, left_turn, side_of_oriented_circle	bbox, centroid, circumcenter, squared_radius, rational_-rotation_approximation
3	Plane, Tetrahedron, Triangle, Iso_cuboid, Bbox, Sphere	coplanar, left_turn, side_of_oriented_sphere	bbox, centroid, circumcenter, cross_product, squared_radius
d	Hyperplane, Sphere	side_of_oriented_sphere	center_of_sphere, lift_to_paraboloid

extends the linear kernel to allow for an exact representation of such intersection points, as well as for linear or circular arcs connecting such points. An analogous toolset in 3D is provided by the *spherical 3D kernel* [CCLT09].

PREDEFINED KERNELS

To ease the choice of an appropriate kernel, CGAL offers three predefined linear kernels for dimension two and three to cover the most common tradeoffs between speed and exactness requirements. All three support initial exact constructions from `double` values and guarantee exact predicate evaluation. They vary in their support for exact geometric constructions and exact roots on the number type level. As a rule of thumb, the less exact functionality a kernel provides, the faster it runs.

`CGAL::Exact_predicates_inexact_constructions_kernel` provides filtered exact predicates, but constructions are performed with `double` and, therefore, subject to possible roundoff errors.

`CGAL::Exact_predicates_exact_constructions_kernel` also provides filtered exact predicates. Constructions are performed exactly, but using a lazy computation scheme [PF11] as a geometric filter. In this way, costly exact computations are performed only as far as needed, if at all.

`CGAL::Exact_predicates_exact_constructions_kernel_with_sqrt` achieves exactness via an algebraic number type (currently, either `leda::real` [MS01] or `CORE::Expr` [KLPY99]).

GENERAL LINEAR KERNELS

The kernels available in CGAL can be classified along the following orthogonal concepts:

Dimension: The dimension of the affine space. The specializations for 2D and 3D offer functionality that is not available in the dD kernel.

Number type: The number type used to store coordinates and coefficients, and to compute the expressions in predicates and constructions.

CGAL distinguishes different concepts to describe the underlying algebraic structure of a number type, along with a concept *RealEmbeddable* to describe an order along the real axis. Most relevant here are the concepts *IntegralDomainWithoutDivision* and *Field*, whose names speak for themselves.

Coordinate representation: The Cartesian representation requires a *field type* (`FT`) as a number type (a model for *Field* and *RealEmbeddable*). The homogeneous representation requires a *ring type* (`RT`) as a number type (a model for *IntegralDomainWithoutDivision* and *RealEmbeddable*).

Reference counting: Reference counting is used to optimize copying and assignment of kernel objects. It is recommended for exact number types with larger memory size. The kernel objects have value-semantics and cannot be modified, which simplifies reference counting. The nonreference counted kernels are recommended for small and fast number types, such as the built-in `double`.

The corresponding linear kernels in CGAL are:

`CGAL::Cartesian<FT>`	Cartesian, reference counted, 2D and 3D
`CGAL::Simple_cartesian<FT>`	Cartesian, nonreference counted, 2D and 3D
`CGAL::Homogeneous<RT>`	homogeneous, reference counted, 2D and 3D
`CGAL::Simple_homogeneous<RT>`	homogeneous, nonreference counted, 2D and 3D
`CGAL::Cartesian_d<FT>`	Cartesian, reference counted, d-dimensional
`CGAL::Homogeneous_d<RT>`	homogeneous, reference counted, d-dimensional

FILTERED KERNELS

CGAL offers two general adaptors to create filtered kernels. The first adaptor operates on the kernel level, the second one on the number type level.

(1) `CGAL::Filtered_kernel<K>` is an adaptor to build a new kernel based on a given 2/3D kernel `K`. All predicates of the new kernel use `double` interval arithmetic as a filter [BBP01] and resort to the original predicates from `K` only if that filter fails. Selected predicates use a semi-static filter [MP07] in addition. Constructions of the new kernel are those provided by `K`.

(2) `CGAL::Lazy_exact_nt<NT>` is an adaptor to build a new number type from a given exact number type `NT`. The new number type uses filters based on interval arithmetic and expression DAGs for evaluation. Only if this filter fails, evaluation is done using `NT` instead. Due to the exactness of `NT`, also the new number type admits exact constructions. Specialized predicates avoid the expression DAG construction in some cases.

There is also a dD Version of the Exact_predicates_inexact_constructions_kernel called `CGAL::Epick_d<D>`: a Cartesian, reference counted kernel that supports exact filtered predicates. The parameter `D` specifies the dimension, which can either be fixed statically at compile-time or dynamically at runtime. Constructions are performed with `double` and, therefore, subject to possible roundoff errors.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

68.2.2 GEOMETRIC TRAITS

CGAL follows the generic programming paradigm in the style of the Standard Template Library [Aus98]; algorithms are parameterized with iterator ranges that decouple them from data structures. In addition, CGAL invented the *circulator* concept to accommodate circular structures efficiently, such as the ring of edges around a vertex in planar subdivisions [FGK+00]. Essential for CGAL's flexibility is the separation of algorithms and data structures from the underlying geometric kernel with a *geometric traits class*.

GLOSSARY

Iterator: A concept for a pointer into a linear sequence; exists in different flavors: input, output, forward, bidirectional, and random-access iterator.

Circulator: A concept similar to iterator but for circular sequences.

Range: A pair $[b, e)$ of iterators (or circulators) describing a sequence of items in a half-open interval notation, i.e., starting *with* b and ending *before* e.

Traits class: C++ programming technique [Mye95] to attach additional information to a type or function, e.g., dependent types, functions, and values.

Geometric traits: Traits classes used in CGAL to decouple the algorithms and data structures from the kernel of geometric representations and primitives. Every algorithm and data structure defines a geometric traits concept and the library provides various models. Often the geometric kernel is a valid model.

EXAMPLE OF UPPER CONVEX HULL ALGORITHM

We show two implementations of the upper convex hull algorithm following Andrew's variant of Graham's scan [Gra72, And79] in CGAL. The first implementation makes straightforward use of a sufficient CGAL default kernel and looks similar to textbook presentations or the LEDA example on page 1806.

```
typedef CGAL::Exact_predicates_inexact_constructions_kernel Kernel;
typedef Kernel::Point_2   Point_2;
Kernel kernel;  // our instantiated kernel object

std::list<Point_2> upper_hull( std::list<Point_2> L) {
    L.sort( kernel.less_xy_2_object());
    L.unique();
    if (L.size() <= 2)
        return L;
    Point_2 pmin = L.front(); // leftmost point
    Point_2 pmax = L.back();  // rightmost point
    std::list< Point_2> hull;
    hull.push_back(pmax);     // use rightmost point as sentinel
    hull.push_back(pmin);     // first hull point
    while (!L.empty() && !kernel.left_turn_2_object()(pmin,pmax,L.front()))
        L.pop_front();         // goto first point p above (pmin,pmax)
```

```
    if (L.empty()) {
        hull.reverse();         // fix orientation for this special case
        return hull;
    }
    Point_2 p = L.front();      // keep last point on current hull separately
    L.pop_front();
    for (std::list< Point_2>::iterator i = L.begin(); i != L.end(); ++i) {
        while (! kernel.left_turn_2_object()( hull.back(), *i, p)) {
            p = hull.back();    // remove non-extreme points from current hull
            hull.pop_back();
        }
        hull.push_back(p);      // add new extreme point to current hull
        p = *i;
    }
    hull.push_back(p);          // add last hull point
    hull.pop_front();           // remove sentinel
    return hull;
}
```

The second implementation is more flexible because it separates the algorithm from the geometry, similar to how generic algorithms are written in CGAL. As an algorithmic building block, we place the core of the control flow—the two nested loops at the end—into its own generic function with an interface of bidirectional iterators and a single three-parameter predicate. We eliminate the additional data structure for the `hull` by reusing the space that becomes available in the original sequence as our stack. The result is thus returned in our original sequence and the function just returns the past-the-end position. Instead of the sentinel, which does not give measurable performance benefits, we explicitly test the boundary case.

```
template <class Iterator, class F>  // bidirectional iterator, function object
Iterator backtrack_remove_if_triple( Iterator first, Iterator beyond, F pred) {
    if (first == beyond)
        return first;
    Iterator i = first, j = first;
    if (++j == beyond)          // i,j mark two elements on the top of the stack
        return j;
    Iterator k = j;             // k marks the next candidate value in the sequence
    while (++k != beyond) {
        while (pred( *i, *j, *k)) {
            j = i;              // remove one element from stack, part 1
            if (i == first)     // explicit test for stack underflow
                break;
            --i;                // remove one element from stack, part 2
        }
        i = j; ++j; *j = *k;    // push next candidate value from k on stack
    }
    return ++j;
}
```

With this generic function, we implement in two lines an algorithm to compute *all* points on the upper convex hull (rather than only the extreme points). All degeneracies are handled correctly. For the sorting, random access iterators are required. Note the geometric traits parameter and how predicates are extracted from the traits class. Any CGAL kernel is a valid model for this traits parameter.

```
template <class Iterator, class Traits>  // random access iterator
Iterator upper_hull( Iterator first, Iterator beyond, Traits traits) {
    std::sort( first, beyond, traits.less_xy_2_object());
    return backtrack_remove_if_triple( first, beyond,
                                       traits.left_turn_2_object());
}
```

EXAMPLE OF A USER TRAITS CLASS

Data structures and algorithms in CGAL can be easily adapted to work on user data types by defining a custom geometric traits class. Consider the following custom point class:

```
struct Point {  // our point type
    double x, y;
    Point( double xx = 0.0, double yy = 0.0) : x(xx), y(yy) {}
    // ... whatever else this class provides ...
};
```

To run the `CGAL::ch_graham_andrew` convex hull algorithm on points of this type, the CGAL Reference Manual lists as requirements on its traits class a type `Point_2`, and function objects `Equal_2`, `Less_xy_2`, and `Left_turn_2`. A possible geometric traits class could look like this:

```
struct Geometric_traits {  // traits class for our point type
    typedef double RT;        // ring number type, for random points generator
    typedef Point  Point_2;   // our point type
    struct Equal_2 {          // equality comparison
        bool operator()( const Point& p, const Point& q) {
            return (p.x == q.x) && (p.y == q.y);
        }
    };
    struct Less_xy_2 {        // lexicographic order
        bool operator()( const Point& p, const Point& q) {
            return (p.x < q.x) || ((p.x == q.x) && (p.y < q.y));
        }
    };
    struct Left_turn_2 {      // orientation test
        bool operator()( const Point& p, const Point& q, const Point& r) {
            return (q.x-p.x) * (r.y-p.y) > (q.y-p.y) * (r.x-p.x); // inexact!
        }
    };
    // member functions to access function objects, here by default construction
    Equal_2     equal_2_object()     const { return Equal_2(); }
    Less_xy_2   less_xy_2_object()   const { return Less_xy_2(); }
    Left_turn_2 left_turn_2_object() const { return Left_turn_2(); }
};
```

In order to let CGAL know that our traits class belongs to our point class, we specialize CGAL's kernel traits accordingly:

```
namespace CGAL {  // specialization that links our point type with our traits class
    template <> struct Kernel_traits< ::Point> {
        typedef ::Geometric_traits Kernel;
    };
}
```

Now the `CGAL::ch_graham_andrew` function can be used with the custom point class `Point`. The traits class also suffices for the random point generators in CGAL. Here is a program to compute the convex hull of 20 points sampled uniformly from a unit disk.

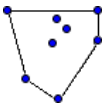
```
#include <CGAL/ch_graham_andrew.h>
#include <CGAL/point_generators_2.h>

int main() {
   std::vector<Point> points, hull;
   CGAL::Random_points_in_disc_2<Point> rnd_pts( 1.0);
   std::copy_n( rnd_pts, 20, std::back_inserter( points));
   CGAL::ch_graham_andrew( points.begin(), points.end(),
                           std::back_inserter( hull));
   return 0;
}
```

The separation of algorithms and data structures from the geometric kernel provides flexibility, the fingerprint of generic programming. Such flexibility is not only useful when interfacing with custom user data or other libraries but even within CGAL itself. An example is the geometric traits class `CGAL::Projection_traits_xy_3` that models the orthogonal projection of 3D points onto the xy-plane, hence treating them as 2D points. Such a model is useful, for instance, in the context of terrain triangulations in GIS (cf. Chapter 59).

68.2.3 LIBRARY CONTENTS

The library is structured into packages that correspond to different chapters of the reference manual. These packages in turn are grouped thematically.

CONVEX HULL

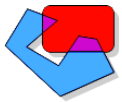


The *2D convex hull* algorithms return the counterclockwise sequence of extreme points. The *3D convex hull* algorithms return the convex polytope of extreme points or—in degenerate cases—a lower-dimensional simplex. The *d-dimensional* convex hull algorithm is part of the d-dimensional triangulation package. It constructs a simplicial complex from which the facets of the hull can be extracted. See Table 68.2.2 for an overview.

TABLE 68.2.2 Convex hull algorithms on a set of n points with h extreme points.

DIM	MODE	ALGORITHM
2	Static	Bykat, Eddy, and Jarvis march, all in $O(nh)$ time
	Static	Akl & Toussaint, and Graham-Andrew scan, both in $O(n \log n)$ time [Sch99]
	Polygon	Melkman for points of a simple polygon in $O(n)$ time
	Others	lower hull, upper hull, subsequences of the hull, extreme points, convexity test
3	Static	quickhull [BDH96]
	Incremental	randomized incremental construction [CMS93, BMS94b]
	Dynamic	by-product of the dynamic 3D Delaunay triangulation
	Test	convexity test as program checker [MNS+99]
d	Incremental	random. incr. constr. [CMS93, BDH09] in $O(n \log n + n^{\lceil d/2 \rceil})$ expected time

POLYGONS



A *polygon* is a closed chain of edges. CGAL provides a container class for polygons, but all functions are generic with iterators and work on arbitrary sequences of points. The functions available are polygon area, point location, tests for simplicity and convexity of the polygon, and generation of random instances.

CGAL also provides container classes to represent simple *polygons with holes* and collections of such polygons. Moreover, there are generalizations of these classes where the boundary is formed by general x-monotone curves rather than line segments specifically. Besides intersection and containment tests, all of these classes support *regularized Boolean operations*: intersection, union, complement, and difference. The implementation is based on arrangements. Regularization means that lower-dimensional features such as isolated points or antennas are omitted.

Where regularization is undesirable, Nef polygons [Nef78, Bie95] provide an alternative representation. A *Nef polygon* is a point set $P \subseteq \mathbb{R}^2$ generated from a finite number of open halfspaces by set complement and set intersection operations. It is therefore closed under Boolean set operations and topological operations, such as closure, interior, boundary, regularization, etc. It captures features of mixed dimension, e.g., antennas or isolated vertices, open and closed boundaries, and unbounded faces, lines, and rays. The potential unboundedness of Nef polygons is addressed with *infimaximal frames* and an extended kernel [MS03]. The representation is based on the *halfedge data structure* [Ket98] (see below), extended with face loops and isolated vertices [See01].

There are functions to compute the *Minkowski sum* of two simple polygons or an *offset polygon*, i.e., the Minkowski sum of a simple polygon with a disk. In both cases the result may not be simple and is, therefore, represented as a polygon with holes. For offset polygons these are generalized polygons, where edges are line segments or circular arcs. As an alternative to an exact representation of the offset, one can obtain an approximation with a guaranteed error bound, specified as an input. The implementation is based on the arrangement data structure and convex decomposition and convolution methods [Wei06, Wei07, BFH+15].

Polygons can also be *partitioned* into y-monotone polygons or convex polygons.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

The y-monotone partitioning is a sweep-line algorithm [BCKO08]. For convex partitioning one algorithm finds the minimum number of convex pieces, and two fast 4-approximation algorithms are given: one is a sweep-line algorithm [Gre83], the other is based on the constrained Delaunay triangulation [HM83].

The *straight skeleton* of a simple polygon [AAAG95] constructs a straight-line graph. The implementation is based on [FO98] with additional vertex events [EE99] and can also be used to compute a corresponding collection of offset polygons.

In *polyline simplification* the goal is to reduce the number of vertices in a collection of polylines while preserving their topology, i.e., not creating new intersections. The input can be a single polygon or an arbitrary set of polylines, which may intersect and even self-intersect. Parameters of the algorithm are: a measure for the simplification cost of removing a single vertex and a stop criterion, such as a lower bound for the number or percentage of vertices remaining or an upper bound for the simplification cost. The algorithm [DDS09] is based on constrained triangulations.

2D visibility [BHH$^+$14] provides algorithms to compute the visibility region of a point in a polygonal domain. The input domain is represented as a bounded face in an arrangement and, in particular, may have holes.

CELL COMPLEXES AND POLYHEDRA

The *halfedge data structure* a.k.a. doubly connected edge list (DCEL) is a general purpose representation for planar structures. It is an edge-based representation with two oppositely directed halfedges per edge [Ket98]. A *polyhedral surface* is a mesh data structure based on the halfedge data structure. It embeds the halfedge data structure in 3D space. The polyhedral surface provides various basic integrity-preserving operations, the "Euler operations" [Ket98]. As an alternative, CGAL provides a *surface mesh* structure that is index rather than pointer based [SB11], which saves memory on 64-bit systems.

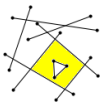
Most algorithms operate on models of *MutableFaceGraph*. This concept extends the *IncidenceGraph* concept of the Boost Graph Library (BGL) [SLL02] by a notion of halfedges, faces, and a cyclic order of halfedges around a face. Both polyhedral surface and surface mesh are models of MutableFaceGraph, but also third-party libraries such as OpenMesh [BSBK02] provide such models. Modeling a refinement of IncidenceGraph, these data structures can directly be used with corresponding BGL algorithms, e.g., for computing shortest paths.

A generalization to d-dimensional space is provided by the *combinatorial map* data structure [DT14]. It implements an edge-based representation for an abstract d-dimensional cell complex. The role of halfedges is taken by darts, which represent an edge together with its incident cells of any dimension. The *linear cell complex* adds a linear geometric embedding to the combinatorial structure, where every vertex of the combinatorial map is associated with a point.

3D Nef polyhedra are the natural generalization of Nef polygons to 3D. They provide a boundary representation of objects that can be obtained by Boolean operations on open halfspaces [HKM07]. In addition to Boolean operations, there are also algorithms [Hac09] to construct the *Minkowski sum* of two Nef polyhedra, and to decompose a given Nef polyhedron P into $O(r^2)$ convex pieces, where r is the number of reflex edges in P.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

ARRANGEMENTS



Planar *Arrangements* are represented using an extended halfedge data structure that supports inner and outer face boundaries with several connected components. The geometry of edges is specified via a traits class, for which many different models exist, such as line segments, polylines, conic arcs, rational functions [SHRH11], Bézier curves, or general algebraic curves. Edges may be unbounded but are assumed to be x-monotone. In order to support general curves, the corresponding traits classes specify a method to split a given curve into x-monotone pieces. The point location strategy can also be specified via the traits class. Available are, among others, a trapezoidal decomposition using randomized incremental construction [HKH12] and a landmark strategy that uses a set of landmark points to guide the search [HH09]. Other supported operations include vertical ray shooting queries and batched point location using a plane sweep.

Using a sweep in an arrangement, one can also compute all pairwise *intersections for* a set of n *curves* in $O((n+k)\log n)$ time, where k is the number of intersection points of two curves. Upper and lower *envelopes* of x-monotone curves in 2D or surfaces in 3D can be constructed using a divide-and-conquer approach. Finally, *Snap rounding* converts a given arrangement of line segments into a fixed precision representation, while providing some topological guarantees. The package also provides a direct algorithm to construct an iterated snap rounding [HP02]. The book by Fogel, Halperin and Wein [FHW12] gives many examples and applications using CGAL arrangements.

TRIANGULATIONS AND VORONOI DIAGRAMS

Triangulations use a triangle-based data structure in 2D, which is more memory efficient than an edge-based structure. Similarly, the representation in 3D is based on tetrahedra. Both data structures act as container classes with an iterator interface and topologically represent a sphere using a symbolic infinite vertex. The construction is randomized incremental and efficient vertex removal [BDP$^+$02, DT11] is supported. Where not mentioned otherwise explicitly, the following structures are available in both 2D and 3D.

Delaunay triangulations also implicitly represent the dual *Voronoi diagram*. But there is also an adaptor that simulates an explicit representation. Point location is the walk method by default. From 10,000 points on, it is recommended [DPT02] to use the Delaunay hierarchy [Dev02] instead. Batched insertion (including construction from a given range of points) spatially sorts the points in a preprocessing step to reduce the time spent for point location.

Regular triangulations are the dual of power diagrams, the Voronoi diagram of weighted points under the power-distance [ES96]. Both Delaunay and regular triangulations in 3D support parallel computation using a lock data structure.

For a *constrained triangulation* (cf. Chapter 29) one can define a set of constraining segments, which are required edges in the triangulation. Constrained (Delaunay) triangulations are available in 2D only. There is optional support for

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

intersecting constraints, which are handled by subdividing segments at intersection points.

For a *periodic triangulation* the underlying space is a torus rather than a sphere. The space is represented using a half-open cube as an original domain, which contains exactly one representative for each equivalence class of points. Points outside of the original domain are addressed using integral offset vectors. Not all point sets admit a triangulation on the torus, but a grid of a constant number of identical copies of the point set always does [CT09].

An *alpha shape* is a sub-complex of a Delaunay triangulation that contains only those simplices whose empty circumsphere has radius bounded by a given α. Alpha shapes are available both for unweighted and for weighted points under the power distance [EM94].

Apollonius graphs are the dual of *additively weighted* Voronoi diagrams (a.k.a. *Apollonius diagrams*). They are available in 2D only, and support dynamic vertex insertion, deletion, and fast point location [KY02].

A *segment Delaunay graph* is dual to a Voronoi diagram for a set of line segments. It is available in 2D only but for both the Euclidean [Kar04] and the L_∞ metric [CDGP14]. Intersecting segments are supported. Geometric primitives use a combination of geometric and arithmetic filtering.

A *d-dimensional triangulation* is combinatorially represented as an abstract pure simplicial complex without boundary. The data structure supports specification of the dimension at either compile time or runtime. Both general triangulations and Delaunay triangulations are available.

CGAL also provides a generic framework [RKG07] for *kinetic data structures* where points move along polynomial trajectories. Specifically it implements kinetic Delaunay triangulations in 2D and 3D and kinetic regular triangulations in 3D.

MESH GENERATION

In Delaunay refinement meshing, the goal is to obtain a Delaunay triangulation for a given domain that (1) respects certain features and (2) whose simplices satisfy certain shape criteria (e.g., avoid small angles). To this end, Steiner points are added to subdivide constraint features or to destroy bad simplices.

The *3D mesh generator* [JAYB15] computes an isotropic simplicial mesh represented as a subcomplex of a 3D Delaunay triangulation. The concept to describe the input domain is very general: The only requirement is an oracle that can answer certain queries about the domain. For instance, does a query point belong to the domain and—if so—to which part of the domain? Domain models include isosurfaces defined by implicit functions, polyhedral surfaces, and segmented 3D images (for instance, from CT scans). The algorithm can handle lower-dimensional features in the input domain using restricted Delaunay triangulations [BO05] and protecting balls [CDR10]. Several different optimization procedures can optionally be used to remove slivers from the resulting mesh: Lloyd smoothing [DW03], odt-smoothing [ACYD05], a perturber [TSA09] and/or an exuder [CDE$^+$00].

The *3D surface mesher* [RY07] handles an input domain that is a surface in 3D, represented as an oracle. The resulting mesh is represented as a two-dimensional subcomplex of a 3D Delaunay triangulation. Theoretical guarantees [BO05] regard-

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

ing the topology depend on the local feature size. But the algorithm can also be run to guarantee a manifold when allowed to relax the shape criteria.

A given 2D constrained triangulation can be transformed into a *conforming* Delaunay or conforming Gabriel triangulation using Shewchuk's algorithm [She00]. *3D skin surfaces* provides an algorithm [KV07] to construct the skin surface [Ede99] for a given set of weighted points (i.e., balls) and a given shrink factor.

GEOMETRY PROCESSING

CGAL provides three algorithms to reconstruct a surface from a set of sample points. Poisson reconstruction [KBH06] computes an implicit function from a given set of points with oriented normal vectors. A corresponding surface can then be obtained using the surface mesher. *Scale-space reconstruction* [DMSL11] computes a surface that interpolates the input points by filtering an alpha shape depending on a scale variable. The scale-space method tends to be more robust with respect to outliers and noise. *Advancing Front Surface Reconstruction* [CD04] greedily adds Delaunay triangles subject to topological constraints, so as to maintain an orientable 2-manifold, possibly with boundary.

3D surface subdivision implements four subdivision methods (cf. [WW02]) that operate on a polyhedron: Catmull-Clark, Loop, Doo-Sabin, and $\sqrt{3}$-subdivision. In the opposite direction, *surface mesh simplification* aims to reduce the size of a given mesh while preserving the shape's characteristics as much as possible. The implementation collapses at every step an edge of minimum cost. The cost function appears as a parameter, with the Lindstrom-Turk model [LT99] as a default.

Surface subdivision decomposes a given mesh based on a shape diameter function (SDF) [SSC08], which aims to estimate the local object diameter. The decomposition is computed using a graph cut algorithm [BVZ01] that minimizes an energy function based on the SDF values.

In *surface deformation* we are given a mesh and a subset of vertices that are to be moved to given target positions. The goal is to deform the mesh and maintain its shape subject to these movement constraints. The implementation operates on a polyhedron and uses the as-rigid-as-possible algorithm [SA07] with an alternative energy function [CPSS10].

Planar parameterization operates on a polyhedron and aims to find a one-to-one mapping between the surface of the polyhedron and a planar domain. There are different desiderata regarding this mapping such as a small angle or area distortion or that the planar domain is convex. Several different methods are provided, such as Tutte barycentric mapping [Tut63], discrete conformal map [EDD$^+$95], Floater mean value coordinates [Flo03], discrete authalic [DMA02], and least squares conformal maps [LPRM02], possibly in combination with boundary conditions.

Geodesic *shortest paths* on a *triangulated surface mesh* can be obtained using an algorithm by Xin and Wang [XW09]. *Triangulated surface mesh skeletonization* builds a 1D mean curvature skeleton [TAOZ12] for a given surface mesh. This skeleton is a curvilinear graph that aims to capture the topology of the mesh. Both algorithms work with a model of a generic *FaceListGraph* concept as an input.

Approximation of ridges and umbilics approximately determines regions of extremal curvature [CP05] on a given mesh, interpreted as a discretization of a smooth

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

surface. The algorithm [CP05] needs the *local differential properties* of the mesh to be provided, which can be obtained using a companion package [CP08].

Point set processing provides tools to analyze and process 3D point sets, for instance, compute the average spacing, remove outliers, simplify, upsample, regularize or smooth the point set, or estimate the normals. The *shape* of a given *point set* with unoriented normals can be *detected* using a RANSAC (random sample consensus) algorithm [SWK07]. Supported shapes include plane, sphere, cylinder, cone and torus.

2D placement of streamlines generates streamlines (everywhere tangent to the field) for a given vector field, using a farthest point seeding strategy [MAD05].

OPTIMIZATION

Geometric optimization algorithms in CGAL fall into three categories: *Bounding Volumes*, *Inscribed Areas*, and *Optimal Distances*; see Table 68.2.3. In addition, *Principal Component Analysis* provides the computation of axis-aligned bounding boxes, centroids, and a linear least squares fitting in 2D and 3D.

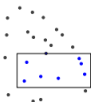
As for combinatorial algorithms, there is a *Linear and Quadratic Programming Solver* to compute exact solutions for linear programs and convex quadratic programs. The solver uses a simplex-like algorithm, which is efficient if either the number of variables or the number of constraints is small [GS00]. CGAL also provides generic implementations of *monotone* [AKM$^+$87] and *sorted* [FJ84] *matrix search*.

TABLE 68.2.3 Geometric optimization.

DIM	ALGORITHM
2,3,d	Smallest enclosing disk/sphere of a point set [Wel91, GS98b, Gär99]
2,3,d	Smallest enclosing sphere of a set of spheres [FG04]
2	Smallest enclosing ellipse of a point set [Wel91, GS98a, GS98c]
2	Smallest enclosing rectangle [Tou83], parallelogram [STV$^+$95], and strip [Tou83] of a point set
d	Smallest enclosing spherical annulus of a point set [GS00]
d	Smallest enclosing ellipsoid of a point set (approximation using Khachyian's algorithm [Kha96])
2	Rectangular p-center, $2 \leq p \leq 4$ [Hof05, SW96]
2	Maximum (area and perimeter) inscribed k-gon of a convex polygon [AKM$^+$87]
2	Maximum area axis-aligned empty rectangle [Orl90]
d	Distance between two convex polytopes (given as a convex hull of a point set) [GS00]
3	Width of a point set [GH01]
2	All furthest neighbors for the vertices of a convex polygon [AKM$^+$87]

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

SPATIAL SEARCHING AND SORTING



Generic *range trees* and *segment trees* [BCKO08] can be interchangeably nested to form higher-dimensional search trees. A related structure in 1D is the *interval skip list*, a fully dynamic data structure to find all intervals that overlap a given point [Han91].

Spatial searching provides various queries based on k-d-trees: k-nearest and k-furthest neighbor searching, incremental nearest and incremental furthest neighbor searching [HS95]. All queries are available as exact and approximate searches. Query items can be points and other spatial objects.

Spatial sorting organizes a sequence of points so as to increase spatial coherence. In combination with carefully dosed randomness this is useful to speed up the localization step of incremental algorithms [ACR03, Buc09].

2D Range and Neighbor Search provides an interface to the dynamic 2D Delaunay triangulation for nearest neighbor, k-nearest neighbors, and range searching in the plane [MN00].

Fast Intersection and Distance Computation in 3D can be done using a hierarchy of axis-aligned bounded boxes stored in an AABB tree. The data structure supports intersection tests, intersection counting, and intersection reporting for a single query object, and computing a closest point for a given query point.

Intersecting Sequences of dD Iso-oriented Boxes efficiently computes all intersecting pairs among a collection of axis-aligned bounded boxes [ZE02]. The algorithm is also useful as a preprocessing heuristic for computing intersections among more general objects.

68.3 SOURCES AND RELATED MATERIAL

FURTHER READING

The LEDA user manual [MNSU] and the LEDA book [MN00] discuss the architecture, the implementation, and the use of the LEDA system.

The most relevant and up-to-date resource about CGAL is its online reference manual at `<doc.cgal.org>`. The design of CGAL and the reasons to use the C++ language are thoroughly covered in [FGK+00]. Generic programming aspects are discussed in [BKSV00]. The design of the CGAL kernel is presented in [HHK+07, PF11], the d-dimensional kernel in [MMN+98], the circular kernel in [EKP+04, DFMT02], and the spherical kernel in [CCLT09]. Older descriptions of design and motivation are in [Ove96, FGK+96, Vel97]. In particular, precision and robustness aspects are discussed in [Sch96], and the influence of different kernels in [Sch99, BBP01]. The most recent survey about CGAL is [Ber14].

RELATED CHAPTERS

Chapter 26: Convex hull computations
Chapter 27: Voronoi diagrams and Delaunay triangulations

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

Chapter 28: Arrangements
Chapter 29: Triangulations and mesh generation
Chapter 30: Polygons
Chapter 33: Visibility
Chapter 35: Curve and surface reconstruction
Chapter 38: Point location
Chapter 40: Range searching
Chapter 45: Robust geometric computation
Chapter 49: Linear programming
Chapter 55: Graph drawing
Chapter 56: Splines and geometric modeling
Chapter 59: Geographic information systems
Chapter 67: Software

REFERENCES

[AAAG95] O. Aichholzer, D. Alberts, F. Aurenhammer, and B. Gärtner. A novel type of skeleton for polygons. *J. Universal Comput. Sci.*, 1:752–761, 1995.

[ACR03] N. Amenta, S. Choi, and G. Rote. Incremental constructions con BRIO. In *Proc. 19th Sympos. Comput. Geom.*, pages 211–219, ACM Press, 2003.

[ACYD05] P. Alliez, D. Cohen-Steiner, M. Yvinec, and M. Desbrun. Variational tetrahedral meshing. *ACM Trans. Graph.*, 24:617–625, 2005.

[AHU74] A.V. Aho, J.E. Hopcroft, and J.D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, Reading, 1974.

[AKM+87] A. Aggarwal, M.M. Klawe, S. Moran, P.W. Shor, and R. Wilber. Geometric applications of a matrix-searching algorithm. *Algorithmica*, 2:195–208, 1987.

[And79] A.M. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Inform. Process. Lett.*, 9:216–219, 1979.

[Aus98] M.H. Austern. *Generic Programming and the STL*. Addison-Wesley, Reading, 1998.

[BBP01] H. Brönnimann, C. Burnikel, and S. Pion. Interval arithmetic yields efficient dynamic filters for computational geometry. *Discrete Appl. Math.*, 109:25–47, 2001.

[BCKO08] M. de Berg, O. Cheong, M. van Kreveld, and M. Overmars. *Computational Geometry: Algorithms and Applications*, third edition. Springer, Berlin, 2008.

[BDH96] C.B. Barber, D.P. Dobkin, and H. Huhdanpaa. The Quickhull algorithm for convex hulls. *ACM Trans. Math. Software*, 22:469–483, 1996.

[BDH09] J.-D. Boissonnat, O. Devillers, and S. Hornus. Incremental construction of the Delaunay triangulation and the Delaunay graph in medium dimension. In *Proc. 25th Sympos. Comp. Geom.*, pages 208–216, ACM Press, 2009.

[BDP+02] J.-D. Boissonnat, O. Devillers, S. Pion, M. Teillaud, and M. Yvinec. Triangulations in CGAL. *Comput. Geom.*, 22:5–19, 2002.

[Ber14] E. Berberich. CGAL—Reliable geometric computing for academia and industry. In *Proc. 4th Internat. Congr. Math. Softw.*, vol. 8592 of *LNCS*, pages 191–197, Springer, Berlin, 2014.

[BFH+15] A. Baram, E. Fogel, D. Halperin, M. Hemmer, and S. Morr. Exact Minkowski sums of polygons with holes. In *Proc. 23rd Eur. Sympos. Algorithms*, vol. 9294 of *LNCS*, pages 71–82, Springer, Berlin, 2015.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

[BFMS00] C. Burnikel, R. Fleischer, K. Mehlhorn, and S. Schirra. A strong and easily computable separation bound for arithmetic expressions involving radicals. *Algorithmica*, 27:87–99, 2000.

[BHH+14] F. Bungiu, M. Hemmer, J. Hershberger, K. Huang, and A. Kröller. Efficient computation of visibility polygons. Preprint, arXiv:1403.3905, 2014.

[Bie95] H. Bieri. Nef polyhedra: A brief introduction. In *Geometric Modelling*, vol. 10 of *Computing Supplement*, pages 43–60, Springer, Berlin, 1995.

[BK95] M. Blum and S. Kannan. Designing programs that check their work. *J. ACM*, 42:269–291, 1995.

[BKSV00] H. Brönnimann, L. Kettner, S. Schirra, and R.C. Veltkamp. Applications of the generic programming paradigm in the design of CGAL. In *Generic Programming*, vol. 1766 of *LNCS*, pages 206–217, Springer, Berlin, 2000.

[BLR93] M. Blum, M. Luby, and R. Rubinfeld Self-testing/correcting with applications to numerical problems. *J. Comput. Syst. Sci.*, 47:549–595, 1993.

[BMS94a] C. Burnikel, K. Mehlhorn, and S. Schirra. How to compute the Voronoi diagram of line segments: Theoretical and experimental results. In *Proc. 2nd Eur. Sympos. Algorithms*, vol. 855 of *LNCS*, pages 227–239, Springer, Berlin, 1994.

[BMS94b] C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Proc. 5th ACM-SIAM Sympos. Discrete Algorithms*, pages 16–23, 1994.

[BN02] M. Bäsken and S. Näher. Geowin—A generic tool for interactive visualization of geometric algorithms. In *Software Visualization*, vol. 2269 of *LNCS*, pages 88–100, Springer, Berlin, 2002.

[BO05] J.-D. Boissonnat and S. Oudot. Provably good sampling and meshing of surfaces. *Graph. Models*, 67:405–451, 2005.

[BSBK02] M. Botsch, S. Steinberg, S. Bischoff, and L. Kobbelt. OpenMesh—A generic and efficient polygon mesh data structure. In *Proc. 1st OpenSG Symposium*, 2002.

[Buc09] K. Buchin. Constructing Delaunay triangulations along space-filling curves. In *Proc. 17th Eur. Sympos. Algorithms*, vol. 5757 of *LNCS*, pages 119–130, Springer, Berlin, 2009.

[BVZ01] Y. Boykov, O. Veksler, and R. Zabih. Fast approximate energy minimization via graph cuts. *IEEE Trans. Pattern Anal. Mach. Intell.*, 23:1222–1239, 2001.

[CCLT09] P.M.M. de Castro, F. Cazals, S. Loriot, and M. Teillaud. Design of the CGAL 3D spherical kernel and application to arrangements of circles on a sphere. *Comput. Geom.*, 42:536–550, 2009.

[CD04] D. Cohen-Steiner and F. Da. A greedy Delaunay-based surface reconstruction algorithm. *Visual Comput.*, 20:4–16, 2004.

[CDE+00] S.-W. Cheng, T.K. Dey, H. Edelsbrunner, M.A. Facello, and S.-H. Teng. Sliver exudation. *J. ACM*, 47:883–904, 2000.

[CDGP14] P. Cheilaris, S.K. Dey, M. Gabrani, and E. Papadopoulou. Implementing the L∞ segment Voronoi diagram in CGAL and applying in VLSI pattern analysis. In *Proc. 4th Internat. Congr. Math. Softw.*, vol. 8592 of *LNCS*, pages 198–205, Springer, Berlin, 2014.

[CDR10] S.-W. Cheng, T.K. Dey, and E.A. Ramos. Delaunay refinement for piecewise smooth complexes. *Discrete Comput. Geom.*, 43(1):121–166, 2010.

[CLR90] T.H. Cormen, C.E. Leiserson, R.L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, Cambridge, 1990; third edition 2009.

[CMS93] K.L. Clarkson, K. Mehlhorn, and R. Seidel. Four results on randomized incremental constructions. *Comput. Geom.*, 3:185–212, 1993.

[CP05] F. Cazals and M. Pouget. Topology driven algorithms for ridge extraction on meshes. Research Report 5526, INRIA Sophia-Antipolis, 2005.

[CP08] F. Cazals and M. Pouget. Algorithm 889: Jet_fitting_3—A generic C++ package for estimating the differential properties on sampled surfaces via polynomial fitting. *ACM Trans. Math. Software*, 35:24, 2008.

[CPSS10] I. Chao, U. Pinkall, P. Sanan, and P. Schröder. A simple geometric model for elastic deformations. *ACM Trans. Graph.*, 29:38, 2010.

[CT09] M. Caroli and M. Teillaud. Computing 3D periodic triangulations. In *Proc. 17th Eur. Sympos. Algorithms*, vol. 5757 of *LNCS*, pages 37–48, Springer, Berlin, 2009.

[DDS09] C. Dyken, M. Dæhlen, and T. Sevaldrud. Simultaneous curve simplification. *J. Geogr. Syst.*, 11:273–289, 2009.

[Dev02] O. Devillers. The Delaunay hierarchy. *Internat. J. Found. Comput. Sci.*, 13:163–180, 2002.

[DFMT02] O. Devillers, A. Fronville, B. Mourrain, and M. Teillaud. Algebraic methods and arithmetic filtering for exact predicates on circle arcs. *Comput. Geom.*, 22:119–142, 2002.

[DMA02] M. Desbrun, M. Meyer, and P. Alliez. Intrinsic parameterizations of surface meshes. *Comput. Graph. Forum*, 21:209–218, 2002.

[DMSL11] J. Digne, J.-M. Morel, C.-M. Souzani, and C. Lartigue. Scale space meshing of raw data point sets. *Comput. Graph. Forum*, 30:1630–1642, 2011.

[DPT02] O. Devillers, S. Pion, and M. Teillaud. Walking in a triangulation. *Internat. J. Found. Comput. Sci.*, 13:181–199, 2002.

[DT11] O. Devillers and M. Teillaud. Perturbations for Delaunay and weighted Delaunay 3D triangulations. *Comput. Geom.*, 44:160–168, 2011.

[DT14] G. Damiand and M. Teillaud. A generic implementation of dD combinatorial maps in CGAL. *Procedia Engineering*, 82:46–58, 2014.

[DW03] Q. Du and D. Wang. Tetrahedral mesh generation and optimization based on centroidal Voronoi tessellations. *Internat. J. Numer. Methods Eng.*, 56:1355–1373, 2003.

[EDD+95] M. Eck, T. DeRose, T. Duchamp, H. Hoppe, M. Lounsbery, and W. Stuetzle. Multiresolution analysis of arbitrary meshes. In *Proc. 22nd Conf. Comput. Graph. Interactive Tech*, pages 173–180, ACM Press, 1995.

[Ede99] H. Edelsbrunner. Deformable smooth surface design. *Discrete Comput. Geom.*, 21:87–115, 1999.

[EE99] D. Eppstein and J. Erickson. Raising roofs, crashing cycles, and playing pool: Applications of a data structure for finding pairwise interactions. *Discrete Comput. Geom.*, 22:569–592, 1999.

[EKP+04] I.Z. Emiris, A. Kakargias, S. Pion, M. Teillaud, and E.P. Tsigaridas. Towards and open curved kernel. In *Proc. 20th Sympos. Comput. Geom.*, pages 438–446, ACM Press, 2004.

[EM94] H. Edelsbrunner and E.P. Mücke. Three-dimensional alpha shapes. *ACM Trans. Graph.*, 13:43–72, 1994.

[ES96] H. Edelsbrunner and N.R. Shah. Incremental topological flipping works for regular triangulations. *Algorithmica*, 15:223–241, 1996.

[FG04] K. Fischer and B. Gärtner. The smallest enclosing ball of balls: Combinatorial structure and algorithms. *Internat. J. Comput. Geom. Appl.*, 14:341–378, 2004.

[FGK+96] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. The CGAL kernel: A basis for geometric computation. In *Proc. 1st ACM Workshop Appl. Comput. Geom.*, vol. 1148 of *LNCS*, pages 191–202, Springer, Berlin, 1996.

[FGK+00] A. Fabri, G.-J. Giezeman, L. Kettner, S. Schirra, and S. Schönherr. On the design of CGAL a computational geometry algorithms library. *Softw. Pract. Exp.*, 30:1167–1202, 2000.

[FHW12] E. Fogel, D. Halperin, and R. Wein. *CGAL Arrangements and Their Applications—A Step-by-Step Guide*. Vol. 7 of *Geometry and Computing*, Springer, Berlin, 2012.

[FJ84] G.N. Frederickson and D.B. Johnson. Generalized selection and ranking: sorted matrices. *SIAM J. Comput.*, 13:14–30, 1984.

[Flo03] M.S. Floater. Mean value coordinates. *Comput. Aided Geom. Des.*, 20:19–27, 2003.

[FO98] P. Felkel and S. Obdržálek. Straight skeleton implementation. In *Proc. 14th Spring Conf. Comput. Graph.*, pages 210–218, 1998.

[Gär99] B. Gärtner. Fast and robust smallest enclosing balls. In *Proc. 7th Eur. Sympos. Algorithms*, vol. 1643 of *LNCS*, pages 325–338, Springer, Berlin, 1999.

[GH01] B. Gärtner and T. Herrmann. Computing the width of a point set in 3-space. In *Proc. 13th Canad. Conf. Comput. Geom.*, pages 101–103, 2001.

[GJ+10] G. Guennebaud, B. Jacob, et al. Eigen v3. `http://eigen.tuxfamily.org`, 2010.

[GPL07] GNU general public license V3+. URL: `http://www.gnu.org/licenses/gpl.html`, June 29, 2007.

[Gra72] R.L. Graham. An efficient algorithm for determining the convex hull of a finite planar set. *Inform. Process. Lett.*, 1:132–133, 1972.

[Gre83] D.H. Greene. The decomposition of polygons into convex parts. In *Computational Geomety*, vol. 1 of *Adv. Comput. Res.*, pages 235–259, JAI Press, Greenwich, 1983.

[GS98a] B. Gärtner and S. Schönherr. Exact primitives for smallest enclosing ellipses. *Inform. Process. Lett.*, 68:33–38, 1998.

[GS98b] B. Gärtner and S. Schönherr. Smallest enclosing circles—An exact and generic implementation in C++. Technical Report B 98–04, Informatik, Freie Universität Berlin, 1998.

[GS98c] B. Gärtner and S. Schönherr. Smallest enclosing ellipses—An exact and generic implementation in C++. Technical Report B 98–05, Informatik, Freie Universität Berlin, 1998.

[GS00] B. Gärtner and S. Schönherr. An efficient, exact, and generic quadratic programming solver for geometric optimization. In *Proc. 16th Sympos. Comput. Geom.*, pages 110–118, ACM Press, 2000.

[Gt14] T. Granlund and the GMP development team. *The GNU Multiple Precision Arithmetic Library*, 6th edition. Manual, `https://gmplib.org/`, 2014.

[Hac09] P. Hachenberger. Exact Minkowksi sums of polyhedra and exact and efficient decomposition of polyhedra into convex pieces. *Algorithmica*, 55:329–345, 2009.

[Han91] E.N. Hanson. The interval skip list: A data structure for finding all intervals that overlap a point. In *Proc. 2nd Workshop Algorithms Data Struct.*, vol. 519 of *LNCS*, pages 153–164, Springer, Berlin, 1991.

[HH09] I. Haran and D. Halperin. An experimental study of point location in planar arrangements in CGAL. *ACM J. Exp. Algorithms*, 13:3, 2009.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

[HHK+07] S. Hert, M. Hoffmann, L. Kettner, S. Pion, and M. Seel. An adaptable and extensible geometry kernel. *Comput. Geom.*, 38:16–36, 2007.

[HKH12] M. Hemmer, M. Kleinbort, and D. Halperin. Improved implementation of point location in general two-dimensional subdivisions. In *Proc. 20th Eur. Sympos. Algorithms*, vol. 7501 of *LNCS*, pages 611–623, Springer, Berlin, 2012.

[HKM07] P. Hachenberger, L. Kettner, and K. Mehlhorn. Boolean operations on 3D selective Nef complexes: Data structure, algorithms, optimized implementation and experiments. *Comput. Geom.*, 38:64–99, 2007.

[HM83] S. Hertel and K. Mehlhorn. Fast triangulation of simple polygons. In *Proc. 4th Internat. Conf. Found. Comput. Theory*, vol. 158 of *LNCS*, pages 207–218, Springer, Berlin, 1983.

[HMN96] C. Hundack, K. Mehlhorn, and S. Näher. A simple linear time algorithm for identifying Kuratowski subgraphs of non-planar graphs. Unpublished, 1996.

[Hof05] M. Hoffmann. A simple linear algorithm for computing rectilinear three-centers. *Comput. Geom.*, 31:150–165, 2005.

[HP02] D. Halperin and E. Packer. Iterated snap rounding. *Comput. Geom.*, 23:209–225, 2002.

[HS95] G.R. Hjaltason and H. Samet. Ranking in spatial databases. In *Proc. 4th Sympos. Advances Spatial Databases*, vol. 951 of *LNCS*, pages 83–95, Springer, Berlin, 1995.

[JAYB15] C. Jamin, P. Alliez, M. Yvinec, and J.-D. Boissonnat. CGALmesh: A generic framework for Delaunay mesh generation. *ACM Trans. Math. Softw.*, 41:23, 2015.

[Kar04] M.I. Karavelas. A robust and efficient implementation for the segment Voronoi diagram. In *Proc. 1st Internat. Sympos. Voronoi Diagrams*, pages 51–62, 2004.

[KBH06] M.M. Kazhdan, M. Bolitho, and H. Hoppe. Poisson surface reconstruction. In *Proc. 4th Eurographics Sympos. Geom. Process.*, pages 61–70, 2006.

[Ket98] L. Kettner. Designing a data structure for polyhedral surfaces. In *Proc. 14th Sympos. Comput. Geom.*, pages 146–154, ACM Press, 1998.

[Kha96] L.G. Khachiyan. Rounding of polytopes in the real number model of computation. *Math. Oper. Res.*, 21:307–320, 1996.

[Kin90] J.H. Kingston. *Algorithms and Data Structures.* Addison-Wesley, Reading, 1990.

[KLPY99] V. Karamcheti, C. Li, I. Pechtchanski, and C.K. Yap. A core library for robust numeric and geometric computation. In *Proc. 15th Sympos. Comput. Geom.*, pages 351–359, ACM Press, 1999.

[KV07] N. Kruithof and G. Vegter. Meshing skin surfaces with certified topology. *Comput. Geom.*, 36:166–182, 2007.

[KY02] M. Karavelas and M. Yvinec. Dynamic additively weighted Voronoi diagrams in 2D. In *Proc. 10th Eur. Sympos. Algorithms*, vol. 2461 of *LNCS*, pages 586–598, Springer, Berlin, 2002.

[LPRM02] B. Lévy, S. Petitjean, N. Ray, and J. Maillot. Least squares conformal maps for automatic texture atlas generation. *ACM Trans. Graph.*, 21:362–371, 2002.

[LT99] P. Lindstrom and G. Turk. Evaluation of memoryless simplification. *IEEE Trans. Vis. Comput. Graph.*, 5:98–115, 1999.

[MAD05] A. Mebarki, P. Alliez, and O. Devillers. Farthest point seeding for efficient placement of streamlines. In *Proc. 16th IEEE Visualization*, pages 479–486, 2005.

[Meh84] K. Mehlhorn. *Data Structures and Algorithms 1, 2, and 3.* Springer, Berlin, 1984.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

[MM96] K. Mehlhorn and P. Mutzel. On the embedding phase of the Hopcroft and Tarjan planarity testing algorithm. *Algorithmica*, 16:233–242, 1996.

[MMN+98] K. Mehlhorn, M. Müller, S. Näher, S. Schirra, M. Seel, C. Uhrig, and J. Ziegler. A computational basis for higher-dimensional computational geometry and applications. *Comput. Geom.*, 10:289–303, 1998.

[MMNS11] R.M. McConnel, K. Mehlhorn, S. Näher, and P. Schweitzer. Certifying algorithms. *Computer Science Review*, 5:119–161, 2011.

[MN94a] K. Mehlhorn and S. Näher. Implementation of a sweep line algorithm for the straight line segment intersection problem. Technical Report MPI-I-94-160, Max-Planck-Institut für Informatik, 1994.

[MN94b] K. Mehlhorn and S. Näher. The implementation of geometric algorithms. In *Proc. 13th IFIP World Computer Congress*, vol. 1, pages 223–231, Elsevier, Amsterdam, 1994.

[MN00] K. Mehlhorn and S. Näher. *LEDA: A Platform for Combinatorial and Geometric Computing*. Cambridge University Press, 2000.

[MNS+99] K. Mehlhorn, S. Näher, M. Seel, R. Seidel, T. Schilz, S. Schirra, and C. Uhrig. Checking geometric programs or verification of geometric structures. *Comput. Geom.*, 12:85–103, 1999.

[MNSU] K. Mehlhorn, S. Näher, M. Seel, and C. Uhrig. The LEDA User Manual. Technical report, Max-Planck-Institut für Informatik. `http://www.mpi-sb.mpg.de/LEDA/leda.html`.

[MP07] G. Melquiond and S. Pion. Formally certified floating-point filters for homogeneous geometric predicates. *ITA*, 41:57–69, 2007.

[MS01] K. Mehlhorn and S. Schirra. Exact computation with leda_real — Theory and geometric applications. In *Symbolic Algebraic Methods and Verification Methods*, pages 163–172, Springer, Vienna, 2001.

[MS03] K. Mehlhorn and M. Seel. Infimaximal frames: A technique for making lines look like segments. *Internat. J. Comput. Geom. Appl.*, 13:241–255, 2003.

[Mye95] N.C. Myers. Traits: A new and useful template technique. *C++ Report*, 1995. `http://www.cantrip.org/traits.html`.

[Nef78] W. Nef. *Beiträge zur Theorie der Polyeder*. Herbert Lang, Bern, 1978.

[NH93] J. Nievergelt and K.H. Hinrichs. *Algorithms and Data Structures*. Prentice Hall, Upper Saddle River, 1993.

[O'R94] J. O'Rourke. *Computational Geometry in C*. Cambridge University Press, 1994.

[Orl90] M. Orlowski. A new algorithm for the largest empty rectangle problem. *Algorithmica*, 5:65–73, 1990.

[Ove96] M.H. Overmars. Designing the Computational Geometry Algorithms Library CGAL. In *Proc. 1st ACM Workshop Appl. Comput. Geom.*, vol. 1148 of *LNCS*, pages 53–58, Springer, Berlin, 1996.

[PF11] S. Pion and A. Fabri. A generic lazy evaluation scheme for exact geometric computations. *Sci. Comput. Program.*, 76(4):307–323, 2011.

[RKG07] D. Russel, M.I. Karavelas, and L.J. Guibas. A package for exact kinetic data structures and sweepline algorithms. *Comput. Geom.*, 38:111–127, 2007.

[RY07] L. Rineau and M. Yvinec. A generic software design for Delaunay refinement meshing. *Comput. Geom.*, 38:100–110, 2007.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

[SA07] O. Sorkine and M. Alexa. As-rigid-as-possible surface modeling. In *Proc. 5th Eurographics Symp. Geom. Process.*, pages 109–116, 2007.

[SB11] D. Sieger and M. Botsch. Design, implementation, and evaluation of the surface_mesh data structure. In *Proc. 20th Int. Meshing Roundtable*, pages 533–550, Springer, Berlin, 2011.

[Sch96] S. Schirra. *Designing a Computational Geometry Algorithms Library.* Lecture Notes for Advanced School on Algorithmic Foundations of Geographic Information Systems, CISM, Udine, 1996.

[Sch99] S. Schirra. A case study on the cost of geometric computing. In *Proc. 1st Workshop Algorithm Eng. Exper.*, vol. 1619 of *LNCS*, pages 156–176, Springer, Berlin, 1999.

[Sch14] B. Schäling. *The Boost C++ Libraries*, 2nd edition. XML Press, Laguna Hills, 2014.

[Sed91] R. Sedgewick. *Algorithms.* Addison-Wesley, Reading, 1991.

[See94] M. Seel. Eine Implementierung abstrakter Voronoidiagramme. Master's thesis, Fachbereich Informatik, Universität des Saarlandes, Saarbrücken, 1994.

[See01] M. Seel. Implementation of planar Nef polyhedra. Research Report MPI-I-2001-1-003, Max-Planck-Institut für Informatik, 2001.

[She00] J.R. Shewchuk. Mesh generation for domains with small angles. In *Proc. 16th Sympos. Comput. Geom.*, pages 1–10, ACM Press, 2000.

[SHRH11] O. Salzman, M. Hemmer, B. Raveh, and D. Halperin. Motion planning via manifold samples. In *Proc. 19th Eur. Sympos. Algorithms*, vol. 6942 of *LNCS*, pages 493–505, Springer, Berlin, 2011.

[SLL02] J.G. Siek, L.-Q. Lee, and A. Lumsdaine. *The Boost Graph Library—User Guide and Reference Manual.* C++ in-depth series. Prentice Hall, Upper Saddle River, 2002.

[SSC08] L. Shapira, A. Shamir, and D. Cohen-Or. Consistent mesh partitioning and skeletonisation using the shape diameter function. *Visual Comput.*, 24:249–259, 2008.

[STV+95] C. Schwarz, J. Teich, A. Vainshtein, E. Welzl, and B.L. Evans. Minimal enclosing parallelogram with application. In *Proc. 11th Sympos. Comput. Geom.*, pages C34–C35, ACM Press, 1995.

[SW96] M. Sharir and E. Welzl. Rectilinear and polygonal p-piercing and p-center problems. In *Proc. 12th Sympos. Comput. Geom.*, pages 122–132, ACM Press, 1996.

[SWK07] R. Schnabel, R. Wahl, and R. Klein. Efficient RANSAC for point-cloud shape detection. *Comput. Graph. Forum*, 26:214–226, 2007.

[TAOZ12] A. Tagliasacchi, I. Alhashim, M. Olson, and H. Zhang. Mean curvature skeletons. *Comput. Graph. Forum*, 31:1735–1744, 2012.

[Tar83] R.E. Tarjan. Data structures and network algorithms. In *CBMS-NSF Regional Conference Series in Applied Mathematics*, vol. 44, 1983.

[Tou83] G.T. Toussaint. Solving geometric problems with the rotating calipers. In *Proc. IEEE MELECON '83*, pages A10.02/1–4, 1983.

[TSA09] J. Tournois, R. Srinivasan, and P. Alliez. Perturbing slivers in 3D Delaunay meshes. In *Proc. 18th Int. Meshing Roundtable*, pages 157–173, Springer, Berlin, 2009.

[Tut63] W.T. Tutte. How to draw a graph. *Proc. London Math. Soc.*, 13:743–768, 1963.

[Vel97] R.C. Veltkamp. Generic programming in CGAL, the computational geometry algorithms library. In *Proc. 6th Eurographics Workshop on Programming Paradigms in Graphics*, 1997.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.

[Wei06] R. Wein. Exact and efficient construction of planar Minkowski sums using the convolution method. In *Proc. 14th Eur. Sympos. Algorithms*, vol. 4168 of *LNCS*, pages 829–840, Springer, Berlin, 2006.

[Wei07] R. Wein. Exact and approximate construction of offset polygons. *Comput. Aided Des.*, 39:518–527, 2007.

[Wel91] E. Welzl. Smallest enclosing disks (balls and ellipsoids). In *New Results and New Trends in Computer Science*, vol. 555 of *LNCS*, pages 359–370, Springer, Berlin, 1991.

[Woo93] D. Wood. *Data Structures, Algorithms, and Performance*. Addison-Wesley, Reading, 1993.

[WW02] J. Warren and H. Weimer. *Subdivision Methods for Geometric Design—A Constructive Approach*. Morgan-Kaufmann, San Francisco, 2002.

[Wyk88] C.J. van Wyk. *Data Structures and C Programs*. Addison-Wesley, Reading, 1988.

[XW09] S.-Q. Xin and G.-J. Wang. Improving Chen and Han's algorithm on the discrete geodesic problem. *ACM Trans. Graph.*, 28:104:1–104:8, 2009.

[ZE02] A. Zomorodian and H. Edelsbrunner. Fast software for box intersections. *Internat. J. Comput. Geom. Appl.*, 12:143–172, 2002.

Preliminary version (August 10, 2017). To appear in the Handbook of Discrete and Computational Geometry, J.E. Goodman, J. O'Rourke, and C. D. Tóth (editors), 3rd edition, CRC Press, Boca Raton, FL, 2017.