
CHAPTER 7

A Verified Polygon-Processing Library for Safety-Critical Embedded Systems

Jens Brandt, Klaus Schneider

University of Kaiserslautern, Reactive Systems Group, Department of Computer Science, P.O. Box 3049, 67653 Kaiserslautern, Germany

CONTENTS

1. Introduction	1
2. Specification and Verification	2
2.1. The HOL System	3
2.2. Formalization of Basic Analytic Geometry	3
2.3. Degenerate Cases and Three-Valued Logic	4
2.4. Geometric Objects and Primitives	6
2.5. Proof Techniques	7
2.6. Example: Cohen-Sutherland Line Clipping	9
2.7. Example: Convex Hull Algorithm	11
3. Rounding and Simplification	12
3.1. Maps and Map Overlay	12
3.2. Conservative Rounding	16
3.3. Map Simplification	18
4. Implementation	19
4.1. Three-Valued Primitives	19
5. Conclusions	19
References	19

1. INTRODUCTION

Embedded systems are small computer systems that are included in other devices in order to optimize available functionalities or to add completely new functionalities to these devices. Important applications of embedded systems are found in the automotive industry. Embedded systems have been successfully used for several years to monitor and optimize different components of cars seen as fuel injection and anti-lock braking systems. The use of radar and optical sensors that are already available in many cars allows the embedded systems to also monitor objects in the

environment of the car. With the help of more complex software and hardware systems, future systems will analyze the current environmental situation on their own to assist the driver and actively take control over some of his tasks in order to avoid serious accidents or at least, reduce the expected damage.

In these new application areas, the spatial behavior of vehicles has to be taken into account. Physical objects and situations are thereby naturally modeled as geometric objects and their related properties, respectively. For these applications, it is sufficient to model the environment

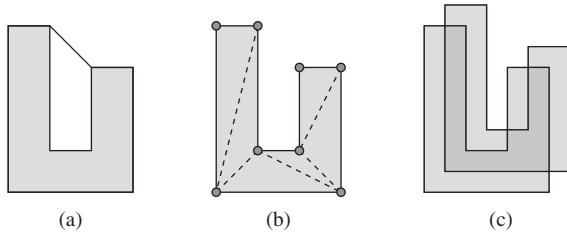


Figure 1. Polygon processing algorithms.

as a two-dimensional plane and approximate the objects by polygonal regions on that plane. Many algorithms for dynamic motion planning or collision detection [24] follow this approach.

For the manipulation of these objects, fundamental algorithms [14] are employed (Fig. 1), e.g., to determine its convex hull (a), to triangulate a polygon (b), and to compute set operations on a given set of polygons (c). By the help of these fundamental tools, more complex problems that occur in applications like collision detection and motion planning can be solved. Much work has been done over the years in developing algorithms to efficiently solve such fundamental problems. In the domain of computational geometry, seemingly every aspect of these problems has been studied in detail and great efforts have been made to transform the initial ideas into robust and efficient implementations as available in the LEDA system [26, 42] or in the CGAL library [38]. However, we cannot directly deploy these implementations in safety-critical embedded systems because of the reasons outlined below.

Verification. Since these systems are used in a safety-critical domain, design errors may lead to severe damages including the loss of human lives. For this reason, all kinds of design errors must be avoided. However, in case of computational geometry algorithms, degenerate cases [15] impose many tricky and unforeseen problems that challenge the design of robust embedded systems. For hardware circuits and embedded systems, formal verification is already routinely performed as a complementary technique to testing and simulation in order to ensure the correctness of the system. In order to apply formal verification to new applications that depend on algorithms from computational geometry, we formalize critical parts of these algorithms and employ the interactive theorem prover HOL [18] for checking relevant specifications [5]. This rigorous formal approach ensures that subtle cases will not be overlooked due to an underestimation of the complexity of the problems that often occurs because of their seemingly intuitive nature. As a result, a verified kernel library is obtained [4] by our approach.

Conservative Computations. In general, the limited precision of the arithmetic operations of microprocessors leads to a loss of information. For safety-critical systems, this loss of information may be fatal, since it may lead to inconsistent situations resulting in unwanted reactions of the system. Moreover, using the usual rounding methods as provided by the IEEE 754 standard for floating point numbers may even lead to completely wrong results [9] in geometric computations. For this reason, some researchers propose

the use of arbitrarily precise arithmetics or interval arithmetic [19, 29, 33] to circumvent these problems. However, since these solutions are too complex for small embedded devices, we conclude that there is a need for explicit rounding functions that approximate the results in a conservative manner according to the specific needs of the particular algorithms [6].

Limited Resources. Due to economic constraints, embedded systems have only very limited computing resources. Nevertheless, the response time must often fulfill given real-time constraints. Hence, application-specific memory management is required for embedded systems. In particular, this memory management includes mechanisms to limit the size of the current model, which limits not only the time needed for computations, but also the precision of the obtained results. In contrast to general approaches, all simplifications must be conservative [6] and may be even computed during the course of an algorithm.

We aim at implementing a dependable software library that meets these additional requirements. In this article, we present our preliminary work in this area to transfer computational geometry algorithms to the domain of safety-critical industrial embedded systems.

This article is structured as follows. In Section 2, we outline how we have specified and verified polygon processing algorithms. In Section 3, we present our approach to limiting the precision of the arithmetic computations by simplifying computed polygons. After some remarks about the implementation in Section 4, we finally draw some conclusions.

2. SPECIFICATION AND VERIFICATION

Geometric problems seem to be easy, since they can be visualized in a natural and intuitive way. As a consequence, the difficulty of geometric algorithms is often underestimated and the algorithms are often only loosely described by means of pictures. However, at a second glance, even simple definitions turn out to be much more complicated than expected. For example, what is the intersection point of two lines, if both lines are identical? For this reason, most algorithms sketched in literature work under certain preconditions such as “all points are pairwise distinct” or “no three points are collinear.”

In order to guarantee that all possible cases are consistently handled and that the used algorithms work as expected in all possible cases, we recommend the use of interactive theorem provers to reason about required definitions and algorithms. However, finding consistent definitions that also hold for degenerate input data is surprisingly difficult [15]. In fact, we found that for many predicates, no simple definition reflects the intuitive understanding in all cases. For example, to decide whether a point on the edge of a polygon belongs to the interior or not yields essential problems when used in one or the other algorithm. This makes it inherently difficult to develop a software library for safety-critical embedded systems that process geometric data.

On the other hand, formal reasoning about geometric problems has a long tradition in automated theorem proving. In particular, Wu’s work [37] on translating geometric propositions to an algebraic form, i.e., equations

between polynomials, became popular [10, 11]. However, this approach focuses on automated theorem proving of *geometric properties*, but it cannot be used to verify the correctness of geometric algorithms that make use of these properties. Moreover, the proofs in Wu’s method are made under the assumption that certain degenerate cases do not occur. In practice, however, degenerate cases do frequently occur, so we cannot neglect them in safety-critical applications.

Work similar to ours but restricted to a convex hull algorithm was done by Pichardie and Bertot [28], where an interactive theorem prover is used to formally reason about geometric properties. Moreover, in contrast to our work, they used two-valued logic to formalize geometric predicates. To circumvent problems of degenerate cases, they modified definitions or used techniques like perturbation, which proved problematic in practice [8].

In this section, we describe our formalization of analytic geometry and some exemplary algorithms of our library. First, we give a brief overview of the HOL system. Then, we focus on the handling of degenerate cases. After defining more complex geometric objects and primitives and some related proof techniques, we finally present two case studies: the verification of Cohen-Sutherland line clipping and a convex hull algorithm. We always restrict our consideration to the formalization of two-dimensional, linear objects such as lines, segments, and polygons, since these restrictions are sufficient for the applications we consider. However, the limitation to two dimensions is not severe, since our formalizations can be easily generalized to higher dimensions and also to other geometric objects.

2.1. The HOL System

The HOL system [18, 41] is an interactive theorem prover for higher order logic with polymorphic types. HOL has been developed by various researchers around the world for more than 20 years. The system provides many data types, mathematical theories, and proof tools that can be extended by users to adapt the system to personal needs. HOL has been used in many areas, including hardware design and verification, reasoning about security, semantics of programming languages, reasoning about real-time systems, and software verification.

The HOL logic is based on Church’s theory of simple types, i.e., a polymorphic higher-order logic [13]. HOL’s calculus consists of five axioms and eight primitive inference rules. This deductive system is shown to be sound, and all extensions of theories by definitions of types and constants are conservative, i.e., they preserve the consistency of the theory, since their existence has to be proved in advance. Thus, the consistency of all HOL theories is guaranteed by construction.

Theorems are of the form $t_1, \dots, t_n \vdash t$, where the assumptions t_1, \dots, t_n and the conclusion t are boolean terms. Such a theorem asserts that if its assumptions are true, then its conclusion is also true. Hence, such a theorem is equivalent to the validity of the formula $(t_1 \wedge \dots \wedge t_n) \rightarrow t$. Theorems with no assumptions are simply written as $\vdash t$. The only way to create theorems is by presenting the system

a proof, which can be achieved by forward and backward reasoning.

HOL is implemented in Moscow ML [43], a light-weight implementation of the strict functional programming language ML. HOL’s open structure allows the users to implement special proof rules and tactics in ML on top of HOL’s functions for their particular application. Types and terms of higher order logic are thereby implemented as data types in ML together with corresponding constructors. Theorems are represented by an ML data type `thm`. However, there is no primitive constructor for this data type. Instead, theorems can only be generated by applying rules, which are ML functions, to already available theorems.

HOL’s metalanguage can be used for programming and extending the theorem prover. New and more powerful inference rules can be obtained by combining simpler rules, and definitions and theorems can be aggregated to form new theories for later use. In this way, the metalanguage makes possible efficient proofs consisting of millions of derivation steps. An important role is played by proof tactics. These procedures determine a sequence of basic logical inference steps needed to formally prove a given goal.

The HOL system comes with dozens of theories providing hundreds of theorems, inference rules, tactics and other useful functions. As each theorem is derived from the core of the system, i.e., the axioms and the primitive inference rules, the HOL system ensures that only consistent theories are used.

2.2. Formalization of Basic Analytic Geometry

In order to reason about algorithms of computational geometry, analytic geometry must be formalized in advance. To this end, all objects (e.g., vectors, points, segments, rays, lines, curves, planes, circles, and spheres) are represented by an algebraic counterpart. Points v are represented by n -tuples of numbers, and a set of points is described by a predicate P such that $P(v)$ holds whenever v belongs to the represented set. Since we restrict our considerations to polygons in the two-dimensional plane, a vector is given by an ordered pair of rational numbers (`rat#rat`), encapsulated in a new HOL type `vec`.

For this type, we make the following definitions: $\vec{0}$ denotes the zero vector, and $\vec{1}_x$ and $\vec{1}_y$ denote the unit vectors. The components of a vector $v = (X_v; Y_v)$ can be accessed by X_v and Y_v , respectively. A vector can be mirrored, rotated, and multiplied by a scalar. A pair of vectors can be added and subtracted.

$$\text{vec_mir} \vdash_{\text{def}} -v = (-X_v; -Y_v)$$

$$\text{vec_orth} \vdash_{\text{def}} \text{orth}(v) = (Y_v; -X_v)$$

$$\text{vec_scale} \vdash_{\text{def}} r \cdot v = (r \cdot X_v; r \cdot Y_v)$$

$$\text{vec_add} \vdash_{\text{def}} v_1 + v_2 = (X_{v_1} + X_{v_2}; Y_{v_1} + Y_{v_2})$$

$$\text{vec_sub} \vdash_{\text{def}} v_1 - v_2 = (X_{v_1} - X_{v_2}; Y_{v_1} - Y_{v_2})$$

In addition to the dot product, the cross product of vectors is a frequently used operation. For the two-dimensional case, it does not exist per se, but a related product that is sometimes called the *perp dot product* exists. This is the dot product

where the first vector is replaced by the perpendicular vector. With its help, the linear dependency of two vectors is easily defined.

$$\text{sprod} \vdash_{\text{def}} v_1 \circ v_2 = x_{v1} \cdot x_{v2} + y_{v1} \cdot y_{v2}$$

$$\text{cprod} \vdash_{\text{def}} v_1 \times v_2 = x_{v1} \cdot y_{v2} - y_{v1} \cdot x_{v2}$$

$$\text{lindep} \vdash_{\text{def}} \text{lindep}(v_1, v_2) = (v_1 \times v_2 = 0)$$

The vectors vec form a vector space over the rational numbers rat . As consequence, various arithmetic laws can be derived. For example, the cross product has the following properties (inter alia):

$$\text{CPROD_RDISTRIB} \vdash (v_1 + v_2) \times v_3 = (v_1 \times v_3) + (v_2 \times v_3)$$

$$\text{CPROD_LSUM} \vdash (v_1 + v_2) \times v_2 = v_1 \times v_2$$

$$\text{CPROD_RSUM} \vdash v_1 \times (v_1 + v_2) = v_1 \times v_2$$

$$\text{CPROD_AINV} \vdash -(v_1 \times v_2) = v_2 \times v_1$$

Our library also includes important theorems of linear algebra like the two-dimensional case of Cramer's rule for the solution of a system of linear equations.

$$\text{CRAMERS_RULE} \vdash \neg(v_1 \times v_2 = 0) \rightarrow$$

$$\begin{aligned} & ((v_0 = r_1 \cdot v_1 + r_2 \cdot v_2) = \\ & (r_1 = (v_0 \times v_2) / (v_1 \times v_2)) \wedge \\ & (r_2 = (v_1 \times v_0) / (v_1 \times v_2))) \end{aligned}$$

We have also proved that the linear dependency relation is an equivalence relation and that it commutes with various vector operations.

2.3. Degenerate Cases and Three-Valued Logic

Algorithms that can be found in textbooks about computational geometry are usually designed for the “general case.” Depending on the algorithm, several preconditions are assumed, e.g., that no points coincide, that given lines are not parallel, or that no three lines intersect in a common point. Even in research papers, authors tend to neglect some or all of these so-called *degenerate cases*, which pose a well-known problem to algorithms in computational geometry [15, 26]. However, experimental data that we obtained from an industrial partner shows that these cases appear frequently in practical applications. For this reason, we cannot simply ignore them in safety-critical embedded systems.

As an example, consider the *winding number algorithm*. It calculates the winding number of a point v relative to a polygon P , i.e., how often the directed sequence of connected line segments of P turns around v (see Ref. [21] for a precise definition). To this end, it counts the intersections of an arbitrary ray starting in v with edges of the polygon P . If the intersected edge runs from the bottom to the top, it is counted positively; if it runs in the opposite direction it is counted negatively. Figure 2 shows the possible cases, where the ray is drawn with a dotted line and some edges of polygon P are drawn with straight lines: (a) and (b) show simple

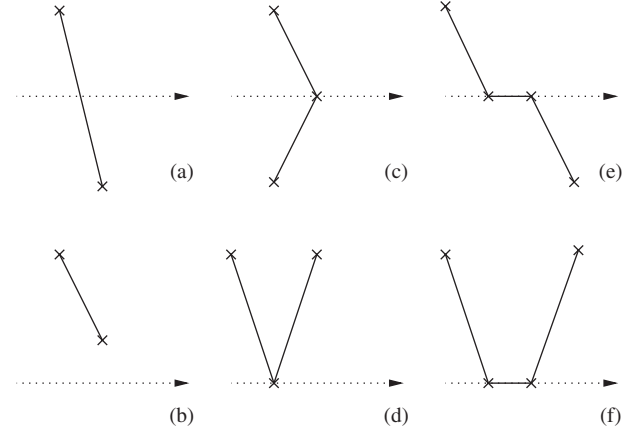


Figure 2. Winding number algorithm.

cases without problems, while (c) to (f) show degenerate cases, i.e., a vertex or an edge of the polygon is on the ray. Depending on the position of the adjoining edges of the polygon, the situation must be counted as an intersection or not. In Figure 2, cases (c) and (e) are counted as an intersection, whereas cases (d) and (f) are not counted as intersections.

Symbolic Perturbations. Degenerate cases like the examples above require a substantial amount of additional effort. Since the number of degenerate cases of an algorithm may grow exponentially due to the combination of several basic degenerate cases, it may not be desirable to explicitly refer to all degenerate cases of complex algorithms. A popular method for eliminating them is the symbolic perturbation of degenerate inputs [15], which resolves degeneracies by simply hiding them (black box method). Intuitively, each point is replaced with a symbolically perturbed point, given by a vector of polynomials of an infinitesimally small number ϵ . Substitution of the symbolically perturbed points in a predicate results in a polynomial in the variable ϵ with coefficients determined by the original points. The sign of the expression is given by the sign of the first nonzero coefficient, where coefficients are taken in increasing order of powers of ϵ . This resolves all degeneracies of the considered primitive. Programs that use this technique tend to be smaller and more robust because the tedious treatment of many special cases is replaced by a single consistent perturbation scheme.

While this method is certainly a useful tool for the implementation of geometric algorithms, existing perturbation schemes have not shown to be as applicable as desired [8]. First, symbolic perturbations give the programmer a rather unsatisfactory choice: either to find an approximation of the original problem, or to find a precise solution of an approximation of the original problem. In some applications, both choices might be inappropriate, and a post-processing step is then required to determine the exact solution of the original problem. Besides its negative impact on the runtime, the complexity of the solution can be significantly increased. Second, symbolic perturbations must be worked out in detail, a task that may be very complex. This has been done only for a few geometric primitives. Finally, objects

that are constructed by the algorithm (e.g., intersection points) are often forbidden in the computation, because their perturbation function depends on the construction of the object and is much more complicated than the one for a primitive object.

Explicit Treatment of Degeneracies. The explicit treatment of degeneracies suffers from the enormous number of cases, which may be hard to enumerate. As an example, consider the intersection of two line segments. In general position, two segments either do not intersect or intersect at a point interior to both segments. Two intersecting segments in special position may however overlap, share a common endpoint, or have one segment endpoint interior to the other segment—and each case exists in various slightly different variations (Fig. 3). Hence, it is obvious that an explicit treatment of all cases for complex algorithms is not an easy task.

As another example, consider the problem of determining whether a point is on the edge, inside, or outside of a polygon. Assume that the points on the edge are considered to be outside the polygon (i.e., polygons are “open” sets of points). However, if we calculate the difference of two polygons by a set difference, the result is possibly a polygon that contains points on its edge. There are two ways to solve the problem. The first one is to modify the definition of the difference. The second one is not to decide whether points on the edge are inside or outside, and therefore, use an undetermined third value for these predicates.

For this reason, we propose the use of three-valued logic to handle degeneracies. Three-valued logic has already proved to be useful in many areas, e.g., for the analysis of asynchronous circuits [7], in compiler analyses [30], for causality analysis [2, 25, 31, 32, 34], and for the analysis of cache behavior [1].

Hence, we do not decide whether a point on the edge is either inside or outside, rather we use in this case a third value to leave this case unspecified. This allows us to describe geometric properties and algorithms more precisely and more compactly without enumerating many tedious cases. Clearly, these cases do not disappear, but three-valued logic makes it possible to treat them in a systematic and concise way and to directly use the results for further computations. An intuitive view of the third truth value is obtained by the analogy to exception handling in some programming languages. When an error occurs, it is not clear how to handle it. Thus, an exception is thrown and is finally caught by a function that has the necessary knowledge to handle the error. Analogously, the degenerate case is passed through all functions until the knowledge of the context is sufficient to decide what to do.

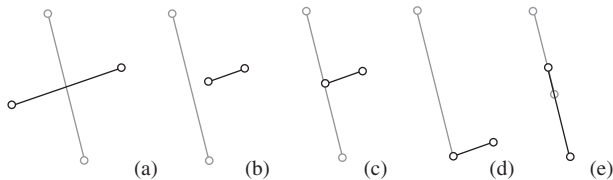


Figure 3. Cases for the intersection of two line segments.

$\bar{\neg}$	F	T
F	T	U
U	T	F

$\bar{\wedge}$	F	U	T
F	F	F	F
U	F	U	U
T	F	U	T

$\bar{\vee}$	F	U	T
F	F	U	T
U	U	U	T
T	T	T	T

$\bar{*}$	F	U	T
F	F	F	F
U	F	U	T
T	F	T	T

$\bar{\rightarrow}$	F	U	T
F	T	T	T
U	U	U	T
T	F	U	T

$\bar{\leftrightarrow}$	F	U	T
F	T	U	F
U	U	U	U
T	F	U	T

$\bar{\oplus}$	F	U	T
F	F	U	T
U	U	U	U
T	T	U	F

$\bar{*}$	F	U	T
F	F	F	F
U	F	U	T
T	F	T	T

Figure 4. Truth tables of three-valued operators.

Reconsider the *point-in-polygon problem*. The area of a polygon is described by a function that maps each point of the plane to one of the three truth values: *true* (T) is assigned to all points inside, *false* (F) to all points outside, and *degenerate* (U) is assigned to all points on the edge of the polygon. These considerations give rise to the definitions of the basic three-valued connectives $\bar{\neg}$, $\bar{\wedge}$, and $\bar{\vee}$ shown in Figure 4, which have already been used by Kleene [3, 22]. We introduce further operators (Fig. 4) like implication $\bar{\rightarrow}$, equivalence $\bar{\leftrightarrow}$, exclusive-or $\bar{\oplus}$ and a modified conjunction $\bar{*}$ (the meaning of which will be explained in Section 2.4). Moreover, we extend the theory by existential and universal quantification:

$$\begin{aligned} \text{exists3} \vdash_{\text{def}} \bar{\exists} P = & \\ & \text{if } (\exists x \cdot Px = T) \text{ then T else} \\ & \quad \text{(if } (\forall x \cdot Px = F) \text{ then F else U)} \\ \text{forall3} \vdash_{\text{def}} \bar{\forall} P = & \\ & \text{if } (\forall x \cdot Px = T) \text{ then T else} \\ & \quad \text{(if } (\exists x \cdot Px = F) \text{ then F else U)} \end{aligned}$$

We use the two-valued theorem prover HOL [18] to reason about our three-valued geometry predicates. Introducing three-valued formulas into such a two-valued environment poses the problem of integrating both logics. The conversion of three-valued expressions to the Boolean domain depends on the proposition: in some situations, T should be the only designated truth value; in other cases, it suffices that a proposition P is “at least U.” Although, this can be expressed by $\neg(P = F)$, we introduce two new relations \leq and \geq to improve the readability. With these relations, all relevant cases ($P = F$, $P \leq U$, $P \geq U$), $P = T$) can be described concisely (Fig. 5). The purpose of the operator $\bar{\rightarrow}$ will be become clear in the following subsection.

\leq	F	U	T
F	T	T	T
U	F	T	T
T	F	F	T

\geq	F	U	T
F	T	F	F
U	T	T	F
T	T	T	T

$\bar{\rightarrow}$	F	U	T
F	T	T	T
U	T	T	F
T	T	F	T

Figure 5. Truth tables of \leq , \geq and $\bar{\rightarrow}$.

2.4. Geometric Objects and Primitives

All geometric objects are formed by sets of points that are the solution of a proposition. To cope with endpoints and other extremal issues, we use three-valued inequations between rational numbers. For equal numbers, the validity of the inequation is U:

$$\begin{aligned} \text{les3} \vdash_{\text{def}} r1 < r2 = \\ \text{if } (r1 < r2) \text{ then T else} \\ \text{(if } (r2 < r1) \text{ then F else U)} \end{aligned}$$

Using this relation, we define geometric objects as sets of points in the following. We thereby focus on two-dimensional linear objects, i.e., lines, segments, and windows. Circles, curves, and objects of higher dimensions are not considered, since in embedded systems they are usually approximated by linear objects.

A line is usually defined by its parametric equation. To convert the classic definition of a line to a three-valued one, all two-valued operators are replaced by their three-valued counterparts ($\text{begin}(l)_1$ and $\text{end}(l)_1$ denote the two points that define the line):

$$\begin{aligned} \text{onLine} \vdash_{\text{def}} \text{onLine}(l_1, v) = \\ \exists \lambda \cdot (v = \text{begin}(l_1) + \lambda \cdot (\text{end}(l_1) - \text{begin}(l_1))) \end{aligned}$$

For a line l , there is no difference between the two-valued and three-valued case: l contains all points $(x; y)$ that are a solution of the traditional, two-valued equation. For a line segment, λ must be greater than 0 and less than 1. With these restrictions, the end points are degenerate points.

$$\begin{aligned} \text{onSeg} \vdash_{\text{def}} \text{onSegment}(l_1, v) = \\ \exists \lambda \cdot (v = \text{begin}(l_1) + \lambda \cdot (\text{end}(l_1) - \text{begin}(l_1))) \wedge \\ (0 < \lambda < 1) \end{aligned}$$

Most geometric algorithms rely on a small number of geometric primitives. Among them are predicates that take some input and classify it as one of a constant number of possible cases, e.g.,

- *Position of two points.* A point p is left from a point q iff $\chi_{\text{Left}}(p, q) := x_q - x_p < 0$. Analogously, point p is below q iff $\chi_{\text{Below}}(p, q) := y_q - y_p < 0$.
- *Orientation of three points.* The points p, q and r define a left turn iff

$$\chi_{\text{Lturn}}(p, q, r) := \begin{vmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_r & y_r & 1 \end{vmatrix} > 0 \quad (1)$$

Degeneracies with respect to a such a predicate P are inputs \vec{x} that cause the characteristic function to become zero $\chi_P(\vec{x}) = 0$. Following the approach presented in Section 2.3, the result U is returned in these cases. To define the predicates, we use the three-valued relation $<$ of the previous section. Since all predicates of the previous section compare

their result with zero, we additionally introduce a relation pos :

$$\text{rat}_{\text{pos}} \vdash_{\text{def}} \text{pos } r = r < 0$$

With this relation, the primitive $\text{left}(v_1, v_2)$ and $\text{below}(v_1, v_2)$ can be defined as follows:

$$\begin{aligned} \text{left} \vdash_{\text{def}} \text{left}(v_1, v_2) &= \text{pos}(X_{v_2} - X_{v_1}) \\ \text{right} \vdash_{\text{def}} \text{right}(v_1, v_2) &= \text{left}(v_1, v_2) \\ \text{below} \vdash_{\text{def}} \text{below}(v_1, v_2) &= \text{pos}(X_{v_2} - X_{v_1}) \\ \text{above} \vdash_{\text{def}} \text{above}(v_1, v_2) &= \text{below}(v_1, v_2) \end{aligned}$$

The primitive $\text{left}(\cdot)$ makes use of the three-valued relation $<$, and thus, it has similar properties. The following theorems prove some sort of reflexivity, antisymmetry, and transitivity laws.

$$\begin{aligned} \text{LEFT_REF} \vdash \text{left}(v_1, v_1) &= \text{U} \\ \text{LEFT_ASYM} \vdash \text{left}(v_1, v_2) &= \neg \text{left}(v_2, v_1) \\ \text{LEFT_TRANS} \vdash \text{left}(v_0, v_1) \ast \text{left}(v_1, v_2) &\rightarrow \text{left}(v_0, v_2) \end{aligned}$$

LEFT_TRANS makes use of the connectives \ast and \rightarrow , which often appear together in a proposition. They allow a succinct description of the following cases:

- If $\text{left}(v_0, v_1) = \text{T}$ and $\text{left}(v_1, v_2) = \text{T}$, then $\text{left}(v_0, v_2) = \text{T}$.
- If $\text{left}(v_0, v_1) = \text{T}$ and $\text{left}(v_1, v_2) = \text{U}$ or vice versa, then $\text{left}(v_0, v_2) = \text{T}$.
- If $\text{left}(v_0, v_1) = \text{U}$ and $\text{left}(v_1, v_2) = \text{U}$, then $\text{left}(v_0, v_2) = \text{U}$.
- If $\text{left}(v_0, v_1) = \text{F}$ or $\text{left}(v_1, v_2) = \text{F}$, then nothing is said about $\text{left}(v_0, v_2)$.

The orientation primitives can be analogously defined:

$$\begin{aligned} \text{lturn} \vdash_{\text{def}} \text{lturn}(v_1, v_2, v_3) &= \text{pos}((v_2 - v_1) \times (v_3 - v_2)) \\ \text{rturn} \vdash_{\text{def}} \text{rturn}(v_1, v_2, v_3) &= \text{lturn}(v_3, v_2, v_1) \end{aligned}$$

Again, various properties can be proved for the orientation primitive:

$$\begin{aligned} \text{LTURN_REF} \vdash \text{lturn}(v_1, v_1, v_2) &= \text{U} \\ \text{LTURN_SYM} \vdash \text{lturn}(v_1, v_2, v_3) &= \text{lturn}(v_2, v_3, v_1) \\ \text{LTURN_ASYM} \vdash \text{lturn}(v_1, v_2, v_3) &= \neg (\text{lturn}(v_2, v_1, v_3)) \\ \text{LTURN_TRIAN} \vdash \\ \text{lturn}(v_1, v_2, v_4) \ast \text{lturn}(v_2, v_3, v_4) \ast \text{lturn}(v_3, v_1, v_4) &\rightarrow \\ \text{lturn}(v_1, v_2, v_3) & \\ \text{LTURN_TRANS} \vdash \\ \text{lturn}(v_1, v_2, v_3) \wedge \text{lturn}(v_1, v_2, v_4) \wedge \\ \text{lturn}(v_1, v_2, v_5) \geq \text{U} \rightarrow \\ \text{lturn}(v_1, v_3, v_4) \ast \text{lturn}(v_1, v_4, v_5) &\rightarrow \text{lturn}(v_1, v_3, v_5) \end{aligned}$$

LTURN_MOD1 ⊢

onRay(mkLine(v_2, v_3), v_4) = T →

lturn(v_1, v_2, v_3) = lturn(v_1, v_2, v_4)

LTURN_MOD2 ⊢

onRay(mkLine(v_4, v_3), v_2) = T →

lturn(v_1, v_2, v_4) = lturn(v_1, v_3, v_4)

These theorems are three-valued reformulation of the ones found in Ref. [28]. The first three theorems (LTURN_REF, LTURN_SYM, and LTURN_ASYM) state that a sequence in which a point appears at least twice is a degenerate case. Moreover, a sequence can be rotated without changing the orientation, and two points can be interchanged without negating the orientation of the sequence. LTURN_TRIAN describes the situation depicted in Figure 6(a): If a point is on the positive side of three pairwise connected segments, they form a triangle with positive orientation. LTURN_TRANS proves the transitivity of the lturn(p,r)edicate under the condition that the three points v_3, v_4 , and v_5 lie on the positive side of a segment from v_1 to v_2 (Fig. 6(b)). The last two theorems (Figs. 6(c) and d)) are used in Ref. [28] to handle degenerate cases. Actually, they are not needed in our approach, since LTURN_TRIAN already covers these cases. This illustrates the advantages of our approach: We always address general and degenerate cases at the same time, which makes the description succinct and readable. The same holds for later implementations that are made with three-valued data types.

2.5. Proof Techniques

In the previous subsections, basic geometric objects and primitives have been defined. These are the basic building blocks of the formalization of the algorithms and their specifications. In this section, we consider related proof tools to automate the reasoning about these objects. Within an interactive theorem prover, these tools are provided in the form of theorems, conversions, and tactics.

For our proof obligations, special support is required in two areas. First, expressions involving three-valued logic should be handled efficiently. A collection of basic theorems

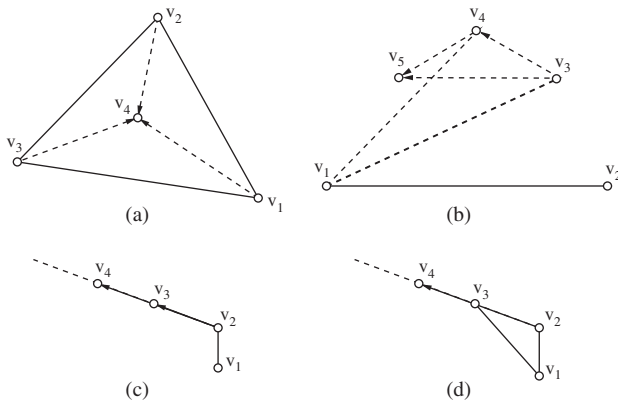


Figure 6. Properties of the lturn(p,r)edicate.

and tactics that are natural extensions of the two-valued cases as well as an automatic reduction to two-valued expressions form a convenient tool set to cope with three-valued proof obligations. Second, many subgoals involving geometric properties arise during the course of proofs. Many of these subgoals can be automatically proved by geometry decision procedures as developed by Wu [37] or Chou [11, 12].

a. Three-Valued Logic and Ternary Algebra. The system $\langle \mathbb{T}, \check{\vee}, \check{\wedge}, \check{\rightarrow}, F, T, U \rangle$ is a ternary algebra [7]. In addition to the laws of commutativity, associativity, distributivity, absorption, and de Morgan as known from a Boolean algebra, the following theorems can be used for the transformation of three-valued terms:

CONJ_TERNARY ⊢ $a \check{\wedge} \check{\rightarrow} a \check{\wedge} U = a \check{\wedge} \check{\rightarrow} a$

DISJ_TERNARY ⊢ $a \check{\vee} \check{\rightarrow} a \check{\vee} U = a \check{\vee} \check{\rightarrow} a$

However, the laws of Boolean algebras are not satisfied due to the violation of the complement laws:

CONJ_BOOLCOMP $\not\vdash a \check{\wedge} \check{\rightarrow} a = F$

DISJ_BOOLCOMP $\not\vdash a \check{\vee} \check{\rightarrow} a = T$

For interactive proofs, the theory offers several tactics that are adapted from the two-valued domain.

- LOG3_GEN_TAC strips the outermost universal quantifier from the conclusion of a goal. When applied to $A \vdash^? \check{\forall} x \cdot P$, it reduces the goal to $A \vdash^? P[x'/x]$ where x' is a variant of x chosen to avoid clashing with any variables free in the assumption list of the goal. This tactic reduces both $\check{\forall} x \cdot P(x) = T$ and $\check{\exists} x \cdot P(x) = F$, since both express universal goals.
- LOG3_EXISTS_TAC reduces an existentially quantified goal to one involving a specific witness. When applied to a term u and a goal $\check{\exists} x \cdot P$, LOG3_EXISTS_TAC reduces the goal to $P[u/x]$ (substituting u for all free instances of x in P , with variable renaming if necessary to avoid free variable capture).
- LOG3_DISCH_TAC moves the antecedent of a (three-valued) implicative goal into the assumptions.
- LOG3_CONJ_TAC reduces a conjunctive goal to two separate subgoals. When applied to a goal $A \vdash^? t_1 \check{\wedge} t_2$, the tactic reduces it to the two subgoals corresponding to each conjunct separately.
- LOG3_DISJ1_TAC and LOG3_DISJ2_TAC select one of the disjuncts of a disjunctive goal.
- LOG3_EQ_TAC reduces a goal of equivalence of three-valued terms to forward and backward implication. When applied to a goal $A \vdash^? t_1 \check{\leftrightarrow} t_2$, the tactic EQ_TAC returns the subgoals $A \vdash^? t_1 \check{\rightarrow} t_2$ and $A \vdash^? t_2 \check{\rightarrow} t_1$.
- Given a term u , LOG3_CASES_TAC applied to a goal produces three subgoals, one with $u = T$ as an assumption, one with $u = U$, and one with $u = F$. A simple and very effective tactic to automatically prove simple theorems about the three-valued logic is LOG3_EXPLORE_TAC. It performs a case distinction on all free variables of the type \mathbb{T} and then uses the simplifier of the theory.

A powerful tactic to prove goals specified in three-valued logic is the transformation to two-valued terms with a subsequent application of the traditional tactics for two-valued goals. For this purpose, a number of rewrite rules are provided that split up a three-valued proposition into positive atomic sub-proposition of the form $P = c$, $P \leq c$ or $P \geq c$ (where $c \in \{F, U, T\}$) connected by two-valued operators. The complete reduction step is implemented by the tactic `LOG3_CALC_TAC` and involves the following steps.

- Elimination of non-constant expressions on the right hand side of equations and inequations.

$$\begin{aligned} \text{LOG3_CASES_EQ} \vdash & (a = F) \wedge (b = F) \vee \\ & (a = U) \wedge (b = U) \vee \\ & (a = T) \wedge (b = T) \\ & = (a = b) \end{aligned}$$

$$\begin{aligned} \text{LOG3_CASES_LEQ} \vdash & (a = F) \wedge (b = F) \vee \\ & a \leq U \wedge (b = U) \vee \\ & (b = T) \\ & = a \leq b \end{aligned}$$

$$\begin{aligned} \text{LOG3_CASES_GEQ} \vdash & (b = F) \vee a \geq U \wedge (b = U) \vee \\ & (a = T) \wedge (b = T) \\ & = a \geq b \end{aligned}$$

In order to eliminate non-constant expressions on the right hand side, these rules must be applied from the right to the left. Of course, unconditional rewriting with these rules does not terminate.

- Elimination of propositions of the form $P = U$. As the following theorems only consider the cases $P = F$, $P = T$, $P \leq U$ and $P \geq U$, rewriting (from right to left) with the following theorem eliminates propositions of the form $P = U$.

$$\text{LOG3_LEQ_GEQ_UU} \vdash a \leq U \wedge a \geq U = (a = U)$$

- Elimination of macro connectives. By rewriting with the definitions of $\vec{\rightarrow}$, $\vec{\leftrightarrow}$, $\vec{\oplus}$, and $\vec{\exists}$, all terms only consist of basic connectives.
- Elimination of basic connectives. All basic three-valued connectives can be reduced to two-valued connectives by rewriting with theorems of the following form.

$$\begin{aligned} \text{LOG3_NOT_CALC} \vdash & ((\vec{\neg} t = F) = (t = T)) \wedge \\ & ((\vec{\neg} t = T) = (t = F)) \wedge \\ & (\vec{\neg} t \leq U = t \geq U) \wedge \\ & (\vec{\neg} t \geq U = t \leq U) \end{aligned}$$

$$\begin{aligned} \text{LOG3_AND_CALC} \vdash & ((a \vec{\wedge} b = F) = (a = F) \vee (b = F)) \wedge \\ & ((a \vec{\wedge} b = T) = (a = T) \wedge (b = T)) \wedge \\ & ((a \vec{\wedge} b) \leq U = a \leq U \vee b \leq U) \wedge \\ & ((a \vec{\wedge} b) \geq U = a \geq U \wedge b \geq U) \end{aligned}$$

$$\begin{aligned} \text{LOG3_OR_CALC} \vdash & ((a \vec{\vee} b = F) = (a = F) \wedge (b = F)) \wedge \\ & ((a \vec{\vee} b = T) = (a = T) \vee (b = T)) \wedge \\ & ((a \vec{\vee} b) \leq U = a \leq U \wedge b \leq U) \wedge \\ & ((a \vec{\vee} b) \geq U = a \geq U \vee b \geq U) \end{aligned}$$

$$\begin{aligned} \text{LOG3_EXT_CALC} \vdash & ((\vec{\Delta} a = F) = \neg a) \wedge \\ & ((\vec{\Delta} a = T) = a) \wedge \\ & (\vec{\Delta} a \leq U = \neg a) \wedge \\ & (\vec{\Delta} a \geq U = a) \end{aligned}$$

$$\begin{aligned} \text{LOG3_EXISTS_CALC} \vdash & ((\vec{\exists} x \cdot Px = F) = \forall b \cdot Pb = F) \wedge \\ & ((\vec{\exists} x \cdot Px = T) = \exists b \cdot Pb = T) \wedge \\ & ((\vec{\exists} x \cdot Px \leq U) = \forall b \cdot Pb \leq U) \wedge \\ & ((\vec{\exists} x \cdot Px \geq U) = \exists b \cdot Pb \geq U) \end{aligned}$$

$$\begin{aligned} \text{LOG3_FORALL_CALC} \vdash & ((\vec{\forall} x \cdot P(x) = F) = \exists b \cdot P(b) = F) \wedge \\ & ((\vec{\forall} x \cdot P(x) = T) = \forall b \cdot P(b) = T) \wedge \\ & ((\vec{\forall} x \cdot P(x) \leq U) = \exists b \cdot P(b) \leq U) \wedge \\ & ((\vec{\forall} x \cdot P(x) \geq U) = \forall b \cdot P(b) \geq U) \end{aligned}$$

- Elimination of negative terms. All two-valued negations in front of subterms can be eliminated, leaving better understandable expressions.

$$\begin{aligned} \text{LOG3_NOT2_CALC} \vdash & (\neg(a = F) = a \geq U) \wedge \\ & (\neg(a = T) = a \leq U) \wedge \\ & (\neg(a = U) = (a = F) \vee (a = T)) \wedge \\ & (\neg(a \leq U) = (a = T)) \wedge \\ & (\neg(a \geq U) = (a = F)) \end{aligned}$$

$$\text{LOG3_ABS_NOT} \vdash (\vec{\Delta} \neg a) = \vec{\neg}(\vec{\Delta} a)$$

b. Geometry Theorem Proving. Formal reasoning about geometric problems has a long tradition in automated theorem proving. In 1951, Tarski presented the first decision procedure [36], which was not yet very practical. The breakthrough came with the much more efficient algebraic method of Wu [37], which soon became very popular and allowed the proof of hundreds of geometry theorems [11]. However, the proofs were very low-level and thus not traceable. The method presented by Chou, Gao, and Zhang [12] finally produced short high-level proofs.

For the verification of polygon processing algorithms, automatic geometry theorem proving is a basic building block, since algorithms are often based on certain geometric properties that must hold for their correct execution. We followed the approach that Narboux [27] proposed for the Coq [39] proof assistant, but in contrast to him, we used our three-valued primitives for all lemmas and tactics.

The basic idea of the method presented by Chou, Gao, and Zhang is to express the proof goal in a constructive way. Based on some general points and lines, new points can be constructed as intersections of lines and new lines can be constructed as parallels to existing lines through a point, etc. With the help of some lemmas, the proof follows

these constructions and eliminates every occurrence of the constructed point in the goal. To this end, the goal must follow a given form, and only arithmetic expressions using three geometric quantities can be used: the signed area of a triangle, the Pythagoras difference and the ratio of two oriented distances. These primitives suffice to express that two lines are collinear, parallel, or orthogonal and that two points are equal [27].

Compared to Narboux [27], we use our three-valued primitives for the formulation of all elimination lemmas. We use general high-level tactics of the HOL system to identify the appropriate lemmas to apply. The resulting goal can be solved by deciding the equality between two rational expressions. To this end, we provide conversions and tactics that calculate normal forms of rational ring terms. With a simple syntactic equality test, the goal can be finally proved. As an alternative, we developed tactics that reduce expressions between rational numbers to expression between integers that can be handled by the integer decision procedures of the HOL system.

2.6. Example: Cohen-Sutherland Line Clipping

Formalization of the Algorithm in HOL. A frequent operation in many graphic applications is the line clipping to an upright rectangular window. To this end, the Cohen-Sutherland algorithm divides the plane into nine regions. Each of the edges of the clip window defines an infinite line that divides the plane into inside and outside half-spaces (Fig. 7(a)). The resulting regions can be described using a four bit *outcode*. The four bits of this code denote whether this region is situated above (*north*), below (*south*), left (*west*), and right (*east*) of the window.

With the help of the outcodes, necessary conditions can be formulated to check whether the line segment is inside or outside the window: The line segment is inside the window iff no bit of the outcodes of the endpoints is true. If a common bit is true in the outcodes of the endpoints, then both endpoints are outside the same side of the window, and thus, the entire line segment is outside the window. In most cases, a line segment will have been either accepted or rejected by these conditions.

In all other cases, the line segment is split into two pieces at an appropriate clipping edge. Assume that the considered end point $(x_1; y_1)$ is above the window w (Fig. 8(b)). Removing the portion of the line that is above the window results in a new line segment with the old endpoint $(x_2; y_2)$ and the new endpoint $(x'_1; y'_1)$. Since the new endpoint is

1001	1000	1010
0001	0000	0010
0101	0100	0110

Figure 7. Cohen-Sutherland: outcodes.

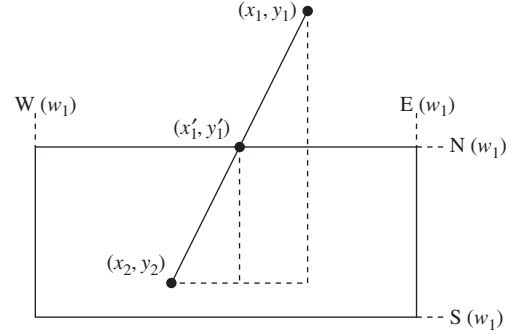


Figure 8. Cohen-Sutherland: computing the intersection.

on the top border of the window, we have $y'_1 = N(w)$. The other coordinate x'_1 can be computed as follows:

$$x'_1 = x_1 + (x_2 - x_1) \cdot \frac{N(w) - y_1}{y_2 - y_1}$$

Once the line segment is identified, the outcode of the new endpoint is computed. After this, the algorithm is restarted with the new values.

In order to formalize the algorithm in HOL, we start with the definition of the outcodes. To this end, we use $SW(w)$ (the lower left corner of w) and $NE(w)$ (the upper right corner of the window w):

$$\text{outcode} \vdash_{\text{def}} \text{Outcode}(w, v) = \begin{pmatrix} \text{below}(v, SW(w)) = T, \\ \text{above}(v, NE(w)) = T, \\ \text{left}(v, SW(w)) = T, \\ \text{right}(v, NE(w)) = T \end{pmatrix}$$

Based on this, we define:

$$\text{INSIDE} \vdash_{\text{def}} \text{Inside}(c_1) = (c_1 = (F, F, F, F))$$

$$\text{BOTTOM} \vdash_{\text{def}} \text{South}(c_1) = c_1[0]$$

$$\text{TOP} \vdash_{\text{def}} \text{North}(c_1) = c_1[1]$$

$$\text{LEFT} \vdash_{\text{def}} \text{West}(c_1) = c_1[2]$$

$$\text{RIGHT} \vdash_{\text{def}} \text{East}(c_1) = c_1[3]$$

$$\text{ACCEPT} \vdash_{\text{def}} \text{Accept}(c_1, c_2) = \text{Inside}(c_1) \wedge \text{Inside}(c_2)$$

$$\text{REJECT} \vdash_{\text{def}}$$

$$\text{Reject}(c_1, c_2) = \begin{pmatrix} c_1[0] \wedge c_2[0] \vee c_1[1] \wedge c_2[1] \vee \\ c_1[2] \wedge c_2[2] \vee c_1[3] \wedge c_2[3] \end{pmatrix}$$

The actual algorithm is taken from Ref. [16], where the loop is replaced by a recursive call. Moreover, as the algorithm does not return an edge in all cases, an *option type* is used for the result, returning *None* if the line is rejected, and *Some(e)* if the edge e is accepted. $W(w)$ and $E(w)$ denote

the maximal and minimal y -values of the window, and $N(w)$ and $S(w)$ denote the x -values.

```

csa_clip ⊢def CSAClip( $w, (v_b, v_e)$ ) =
  if Accept(Outcode( $w, v_b$ ), Outcode( $w, v_e$ )) then
    Some( $(v_b, v_e)$ )
  else if Reject(Outcode( $w, v_b$ ), Outcode( $w, v_e$ )) then
    None
  else if Inside(Outcode( $w, v_b$ )) then
    CSAClip( $w, (v_e, v_b)$ )
  else
    CSAClip( $w, ShortenedLine(w, (v_b, v_e))$ )

```

```

shortened_line ⊢def ShortenedLine( $w, (v_b, v_e)$ ) =

```

```

let

```

```

 $x_1 = X_{vb}$  and  $y_1 = Y_{vb}$  and

```

```

 $x_2 = X_{vE}$  and  $y_2 = Y_{vE}$  and

```

```

 $c_1 = \text{Outcode}(w, v_b)$ 

```

```

in

```

```

if North( $c_1$ ) then

```

```

   $\left( \left( x_1 + (x_2 - x_1) \cdot \frac{N(w) - y_1}{y_2 - y_1}; N(w) \right), v_e \right)$ 

```

```

else if South( $c_1$ ) then

```

```

   $\left( \left( x_1 + (x_2 - x_1) \cdot \frac{S(w) - y_1}{y_2 - y_1}; S(w) \right), v_e \right)$ 

```

```

else if West( $c_1$ ) then

```

```

   $\left( \left( W(w); y_1 + (y_2 - y_1) \cdot \frac{W(w) - x_1}{x_2 - x_1} \right), v_e \right)$ 

```

```

else

```

```

   $\left( \left( E(w); y_1 + (y_2 - y_1) \cdot \frac{E(w) - x_1}{x_2 - x_1} \right), v_e \right)$ 

```

Verification. As the first step of the verification, we have to set up a formal specification of the algorithm. Given a line segment (by its two endpoints v_b and v_e) and a rectangular window (by its lower left and its upper right corners), return the line segment that is inside the window, where all points on the edge are considered to be inside. To keep the specification concise, we use the following two predicates: $\text{CSAInWin}(w_1, v)$ holds if the point v is inside or on the edge of window w_1 , and $\text{CSAOnSeg}((v_b, v_e), v)$ holds if the point v is on the line segment whose endpoints are v_b and v_e (including the endpoints). Note that degenerate inputs and outputs are possible, since the input and the output segments may consist of a single point.

```

csa_in_win ⊢def CSAInWin( $w_1, v$ ) =

```

```

  (inWindow( $w, v$ ) ≥ U)

```

```

csa_on_seg ⊢def CSAOnSeg( $(v_b, v_e), v$ ) =

```

```

  if ( $v_b = v_e$ )

```

```

    then  $v_b = v$ 

```

```

    else onSegment(mkLine( $v_b, v_e$ ),  $v$ ) ≥ U

```

In any case, the result is a value of an option type. It is either None (if the line segment is outside the window) or it represents a pair of points defining the clipped line segment.

```

CSA_CORRECTNESS ⊢

```

```

  let  $r = \text{CSAClip}(w_1, (v_b, v_e))$  in

```

```

    if isNone( $r$ ) then

```

```

       $\neg(\exists v. \text{CSAOnSeg}((v_b, v_e), v) \wedge \text{CSAInWin}(w_1, v))$ 

```

```

    else

```

```

       $\forall v. \text{CSAOnSeg}(\text{The}(r), v) =$ 

```

```

        ( $\text{CSAOnSeg}((v_b, v_e), v) \wedge \text{CSAInWin}(w_1, v)$ )

```

The correctness of the above specification is proved by induction over the recursive calls of the clipping function $\text{CSAClip}(w_1, (v_b, v_e))$. There are two base cases. In the first case, the line segment is accepted, since both endpoints are inside the window. We have to prove that every point of the segment is then also in the window ($\text{ACCEPT_SEGMENT_INWINDOW}$).

```

ACCEPT_SEGMENT_INWINDOW ⊢

```

```

  Accept(Outcode( $w_1, v_b$ ), Outcode( $w_1, v_e$ )) →

```

```

  CSAOnSeg( $(v_b, v_e), v$ ) →

```

```

  CSAInWin( $w_1, v$ )

```

The second base case results from the rejection of the line segment in the first recursive call. For the proof, we use the bounding box check BoundingBox . If the Reject predicate holds, we prove that the window that is defined by the endpoints of the line segment and the clipping window have no intersection. Thus, the segment does not have an intersection with the clipping window, and it is correctly rejected.

```

bbox_check ⊢def BoundingBox( $v_1, v_2, v_3, v_4$ ) =

```

```

  (vecXint( $v_1, v_3$ )  $\checkmark$  vecXint( $v_1, v_4$ )  $\checkmark$ 

```

```

  vecXint( $v_3, v_1$ )  $\checkmark$  vecXint( $v_3, v_2$ ) ≥ U)  $\wedge$ 

```

```

  (vecYint( $v_1, v_3$ )  $\checkmark$  vecYint( $v_1, v_4$ )  $\checkmark$ 

```

```

  vecYint( $v_3, v_1$ )  $\checkmark$  vecYint( $v_3, v_2$ ) ≥ U)

```

```

REJECT_BBOX_CHECK ⊢

```

```

  Reject(Outcode( $w_1, v_b$ ), Outcode( $w_1, v_e$ )) →

```

```

   $\neg \text{BoundingBox}(\text{SW}(w_1), \text{NE}(w_1), v_b, v_e)$ 

```

```

CSA_BBOX_CHECK ⊢

```

```

   $\neg \text{BoundingBox}(v_1, v_2, v_3, v_4) \rightarrow$ 

```

```

  ( $\text{CSAInWin}(\text{mkWin}(v_3, v_4), v) \geq U) \rightarrow$ 

```

```

   $\neg \text{CSAInWin}(v_1, v_2)$ 

```

The first recursive call swaps the end points. Provided that the induction hypotheses holds, the algorithm is correct, because swapping the end points does not change the set of points of the segment (CSA_ONSEG_SYM).

$$\begin{aligned} \text{CSA_ONSEG_SYM} \vdash \\ \text{CSAOnSeg}((v_b, v_e), v) = \text{CSAOnSeg}((v_e, v_b), v) \end{aligned}$$

The second recursive call is the hardest part of the proof. The line segment is shortened, and two things must be proved. First, the shortened segment is a subset of the original one (CSA_CUT_CORRECT). Second, points on the segment that are inside the window are not cut off (CSA_CUT_COMPLETE).

$$\begin{aligned} \text{CSA_CUT_CORRECT} \vdash \\ \neg \text{Accept}(\text{Outcode}(w_1, v_b), \text{Outcode}(w_1, v_e)) \rightarrow \\ \neg \text{Reject}(\text{Outcode}(w_1, v_b), \text{Outcode}(w_1, v_e)) \rightarrow \\ \neg \text{Inside}(\text{Outcode}(w_1, v_b)) \rightarrow \\ \text{CSAInWin}(w_1, v) \rightarrow \\ \text{CSAOnSeg}(\text{ShortenedLine}(w_1, (v_b, v_e)), v) \rightarrow \\ \text{CSAOnSeg}((v_b, v_e), v) \\ \text{CSA_CUT_COMPLETE} \vdash \\ \neg \text{Accept}(\text{Outcode}(w_1, v_b), \text{Outcode}(w_1, v_e)) \rightarrow \\ \neg \text{Reject}(\text{Outcode}(w_1, v_b), \text{Outcode}(w_1, v_e)) \rightarrow \\ \neg \text{Inside}(\text{Outcode}(w_1, v_b)) \rightarrow \\ \text{CSAInWin}(w_1, v) \rightarrow \\ \text{CSAOnSeg}((v_b, v_e), v) \rightarrow \\ \text{CSAOnSeg}(\text{ShortenedLine}(w_1, (v_b, v_e)), v) \end{aligned}$$

$$\begin{aligned} \text{LINE_SPLIT} \vdash \\ (\text{onSegment}(l_1, v_m) = \text{T}) \rightarrow \\ \neg(v_m = \text{end}(l_1)) \rightarrow \\ ((\text{onSegment}(l, 1)v \geq \text{U}) = \\ (v = \text{begin}(l_1)) \vee \\ (\text{onSegment}(\text{mkLine}(\text{begin}(l_1), v_m), v) = \text{T}) \vee \\ (\text{onSegment}(\text{mkLine}(v_m, \text{end}(l_1)), v) \geq \text{U})) \end{aligned}$$

The key to prove the remaining part is to show that the line segment is split into two partitions (LINE_SPLIT). This concludes the correctness proof that holds for all cases (including that endpoints are on the edges of the window, or that both endpoints are the same). Hence, we have proven that the algorithm terminates and that it returns the specified result.

2.7. Example: Convex Hull Algorithm

As a second example to illustrate our approach, we consider in this section the convex hull algorithm as presented in Ref. [14]. It divides the computation of the convex hull

into two parts: the upper part and the lower part of the hull (Fig. 9(a)). In this section, we focus on the construction of the lower part.

Formalization. The algorithm takes a list of points \mathcal{L} , which is sorted in lexicographic order (denoted as $\text{lexSorted}(\mathcal{L})$), i.e., points are first sorted by their x -coordinates, and if the x -coordinates should be the same, then the y -coordinates determine the ordering. The points are iteratively added to the lower part of the convex hull. After each addition, it is checked whether the last three points make a left turn. If this is not the case, the middle point is deleted. These steps are repeated until the last three points make a left turn, or there are only two points left (the leftmost point and the added point). Figure 9(b) illustrates this procedure. Formally, the construction of the lower hull can be described by the following functions, where $[]$ denotes the empty list, $[e_1; e_2]$ a list containing the two elements e_1 and e_2 , and $e :: \mathcal{L}$ denotes the concatenation of a new leftmost element e to an existing list \mathcal{L} :

$$\begin{aligned} \text{normalize_lower} \vdash_{\text{def}} \\ (\text{NormLow}([]) = []) \wedge \\ (\text{NormLow}([e_1]) = [e_1]) \wedge \\ (\text{NormLow}([e_1; e_2]) = [e_1; e_2]) \wedge \\ (\text{NormLow}((e_1 :: e_2 :: e_3 :: \mathcal{L})) = \\ \text{if } \text{Itturn}(e_1, e_2, e_3) = \text{T} \text{ then } e_1 :: e_2 :: e_3 :: \mathcal{L} \\ \text{else } \text{NormLow}((e_1 :: e_3 :: \mathcal{L}))) \\ \text{hull_lower} \vdash_{\text{def}} \\ (\text{LowerHull}([]) = []) \wedge \\ (\text{LowerHull}(e :: \mathcal{L}) = \text{NormLow}(e :: \text{LowerHull}(\mathcal{L}))) \end{aligned}$$

If \mathcal{L} has at least three elements, $\text{NormLow}(\mathcal{L})$ deletes the second element if the first three elements should not form a left turn, and hullLow applies this function to all sublists of a list \mathcal{L} .

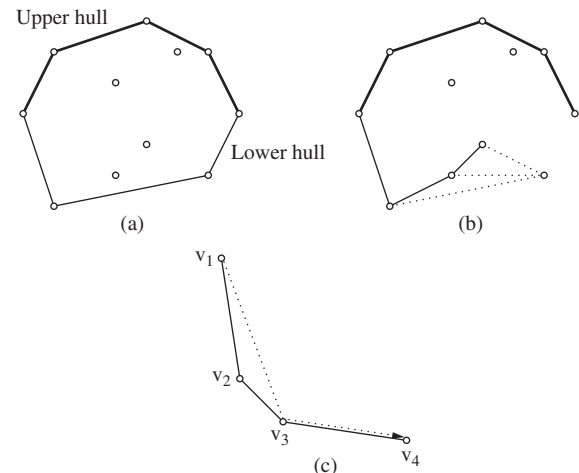


Figure 9. Computation of the convex hull.

Specification. A sequence of points is part of the convex hull if for two consecutive points, all other points lie on the left hand side of the line between these points. We define the corresponding predicate `lconvex` recursively: a sequence of no elements or one element is always convex. Each additional point that is added must lie on the left of all former edges of the constructed convex hull (`lpoint`), and all points must lie on the left side of the edge that is created by the insertion of the new point (`ledge`).

$$\begin{aligned}
\text{left_edge} &\vdash_{\text{def}} \\
&(\text{leftEdge}(e_1, e_2, [])) \wedge \\
&(\text{leftEdge}(e_1, e_2, e :: \mathcal{L}) = \\
&\quad (\text{lturn}(e, e_1, e_2) = \text{T}) \wedge \text{leftEdge}(e_1, e_2, \mathcal{L})) \\
\text{left_point} &\vdash_{\text{def}} \\
&(\text{leftPoint}(e, [])) \wedge \\
&(\text{leftPoint}(e, [e_1])) \wedge \\
&(\text{leftPoint}(e, e_1 :: e_2 :: t) = \\
&\quad (\text{lturn}(e, e_2, e_1) = \text{T}) \wedge \text{leftPoint}(e, e_2 :: \mathcal{L})) \\
\text{left_convex} &\vdash_{\text{def}} \\
&(\text{leftConvex}([])) \wedge \\
&(\text{leftConvex}([e_1])) \wedge \\
&(\text{leftConvex}(e_1 :: e_2 :: \mathcal{L}) = \\
&\quad \text{leftEdge}(e_1, e_2, \mathcal{L}) \wedge \text{leftPoint}(e_1, e_2 :: \mathcal{L}) \wedge \\
&\quad \text{leftConvex}(e_2 :: \mathcal{L}))
\end{aligned}$$

Verification. The verification is done in several steps. First, by applying the definitions, it is proved that every sublist of three points in the result make a left turn.

$$\begin{aligned}
\text{left_chain} &\vdash_{\text{def}} \\
&(\text{leftChain}([])) \wedge \\
&(\text{leftChain}([e_1])) \wedge \\
&(\text{leftChain}([e_1; e_2])) \wedge \\
&(\text{leftChain}(e_1 :: e_2 :: e_3 :: \mathcal{L}) = \\
&\quad (\text{lturn}(e_1, e_2, e_3) = \text{T}) \wedge \text{leftChain}(e_2 :: e_3 :: \mathcal{L})) \\
\text{LEFT_CHAIN_HULL_LOWER} &\vdash \\
&\text{leftChain}(\mathcal{L}_0) \Rightarrow \text{leftChain}(\text{LowerHull}(\mathcal{L}_0)\mathcal{L}_1)
\end{aligned}$$

Then, under the condition of a lexicographic ordering a kind of transitivity (Fig. 9(c)) is derived. To prove this, the lexicographic conditions are translated to left-turn conditions before the transitivity of the left-turn predicate `LTURN_TRANS` is used. With the help of this lemma, an induction proves the desired theorem `CVX_LOWER`.

$$\begin{aligned}
\text{CVX_TRANS_LOWER} &\vdash \\
&(\text{lturn}(v_1, v_2, v_3) = \text{T}) \wedge (\text{lturn}(v_2, v_3, v_4) = \text{T}) \wedge \\
&(v_1 < v_2 = \text{T}) \wedge (v_2 < v_3 = \text{T}) \wedge (v_3 < v_4 = \text{T})
\end{aligned}$$

$$\Rightarrow (\text{lturn}(v_1, v_3, v_4) = \text{T})$$

$$\text{CVX_LOWER} \vdash$$

$$\text{lexSorted}(\mathcal{L}) \wedge \text{leftChain}(\mathcal{L}) \Rightarrow \text{leftConvex}(\mathcal{L})$$

Note that in the proofs, we do not have to address the degenerate cases explicitly. Instead, we exploit that theorems like `LTURN_TRANS` subsume many cases. Thus, the correctness of the algorithm is guaranteed for all cases: in particular for the situation that two subsequent input points have the same y -coordinate or there are collinear points in the input set.

3. ROUNDING AND SIMPLIFICATION

While degenerate cases are a major pitfall for the design of computational geometry algorithms, even more problems arise for their implementation. Geometric primitives must be efficiently computed with limited-precision arithmetics, and limited memory may lead to the situation where the result of an algorithm cannot be exactly represented.

In this section, we present particular solutions to such problems that appear for a map overlay algorithm used in a safety-critical environment. First, in order to understand the problem and the given algorithms, map data structures and the map overlay problem are presented. Then, we explain our solution to conservative rounding and map simplification.

3.1. Maps and Map Overlay

As the name suggests, maps and the map overlay problem have their origin in the domain of geographic information systems. In our applications, we use maps to describe objects and polygonal regions, e.g., vehicles, obstacles, dangerous or safe areas. Basically, a map can be defined as follows.

DEFINITION 1 (Map) *A bounded map in the plane consists of a triple $M = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ where:*

- $\mathcal{V} = \text{vertices}(M) \subseteq \mathbb{Q}^2$ is a finite set of vertices.
- $\mathcal{E} = \text{edges}(M) \subseteq \mathcal{V} \times \mathcal{V}$ is the set of edges. An edge $e = (v_i, v_j)$ connects two different vertices $v_i, v_j \in \mathcal{V}$. A point v can belong to more than one line segment induced by an edge $e = (v_i, v_j)$ only if it is one of the vertices v_i or v_j . Since edges do therefore not intersect, a map is always a planar graph.
- The edges of a map induce a partition of the plane into a set of faces $\mathcal{F} = \text{faces}(M)$. Each face is a polygonal region bounded by one outer component (sequence of edges) and possibly several inner components, which are commonly referred to as holes. Moreover, a map has exactly one unbounded face $\text{unbdFace}(\text{()M})$, which has only inner components.

In our case, all faces are labeled. The multiset $\text{labels}(f)$ associated with the face $f \in \mathcal{F}$ of a map contains the attribute information, i.e., it describes the properties of f on application level.

Figure 10 gives an example map and illustrates the data structure. We represent maps as doubly connected edge lists.

In this data structure, edges consist of pairs of half-edges representing both directions of an edge. Two half-edges forming a pair are called twins, and each one has a reference to the adjacent face. For the sake of simplicity, we hide this detail in the following and use only edges in the descriptions of our algorithms.

Thus, each edge e has references (dashed lines) to the two vertices $src(e)$ and $dst(e)$ that it connects. The face that lies on its left-hand side (as seen from source to destination) is its $leftFace(e)$, the opposite one is its $rightFace(e)$. A face f has references to its outer component $outerComp(f)$ and the set of its inner components $innerComp(f)$. They point to an arbitrary edge of the component, from which the whole component can be traversed. Finally, each map M has a reference to its unbounded face $unbdFace(M)$.

The map overlay operation can be used to combine a set of input maps (sometimes called layers) into a single new one and covers a lot of other geometric problems by means of simple reductions. For example, the computation of Boolean combinations of polygons can be easily solved by a reduction to a map overlay problem, since polygons can be easily transformed to labeled maps. More formally, a map overlay is defined as follows.

DEFINITION 2 *The overlay of a set of maps $\mathcal{M} = \{M_1, \dots, M_n\}$ where $M_i = (\mathcal{V}_i, \mathcal{E}_i, \mathcal{F}_i)$ is a map $M = (\mathcal{V}, \mathcal{E}, \mathcal{F})$ where*

- \mathcal{V} contains all input vertices $\bigcup_{i=1..n} \mathcal{V}_i$ as well as all the intersections of the edges.
- The set of edges \mathcal{E} is constructed by taking all input edges $\bigcup_{i=1..n} \mathcal{E}_i$ and then iteratively replacing all pairs of intersecting edges (e_1, e_2) with $OverlayIntersection(e_1, e_2)$ and all pairs that have a common segment with $OverlayCommonSegment(e_1, e_2)$. We thereby use a lexical ordering for map vertices:
 $v_1 < v_2 \Leftrightarrow (X_{v_1} < X_{v_2} \vee X_{v_1} = X_{v_2} \wedge Y_{v_1} < Y_{v_2})$.
- \mathcal{F} is again induced by the set of edges \mathcal{E} of the new map.

The labels $labels(f)$ of a face $f \in \mathcal{F}$ of the overlay is defined as expected. Each face is labeled with the sum of labels of the faces that contribute to this face. To determine the set of contributing faces, choose an arbitrary point

inside the face and locate in each input layer the face that contains it.

As we are only interested in the labeled area of the faces, redundant vertices and edges can be removed. Edges are redundant if the face on both of its sides are the same. Vertices can be removed if no edge is connected to them.

The map overlay problem is a well-studied problem of computational geometry. Based on the fundamental ideas [14], we developed an algorithm that can handle all maps, i.e., where degenerate inputs do not pose a problem. In contrast to previous publications, we present our solution in full detail, since the handling of degeneracies is by far the most tricky part of the algorithm. While implementing the algorithm and reasoning about it with a theorem prover, we found a lot of imprecise explanations and forgotten cases. Although many authors claim to care about degenerate cases, important details are often left open, and possible cases sometimes remain unhandled: e.g., the authors of Ref. [14] do not handle overlapping edges in the map overlay algorithm. Moreover, our algorithm only constructs the parts of the overlay that are significant with respect to a given operation of the labeling. Since we are not interested in redundant vertices and edges (with respect to the labeling), isolated vertices are immediately removed, and new edges are only created if the labels of the faces on both sides are different. So, the memory usage of the overlay computation is kept as low as possible.

Our algorithm (Fig. 12) is based on the plane sweep approach [14], i.e., the result is computed incrementally while moving an imaginary vertical line over the inputs from left to right. Algorithms of this class generally maintain two data structures: an event queue Q , which contains all points where computations are performed, and S , a status structure that represents the situation at the current position of the sweep line. In the map overlay algorithm, this structure is updated at each event point. For the beginning of an edge e , e is inserted in the status structure, and at the endpoint of an edge, it is removed from the map.

For the map overlay algorithm, event points (Fig. 13) are the vertices of the result map. Each triple in the queue Q represents a vertex v together with the edges that

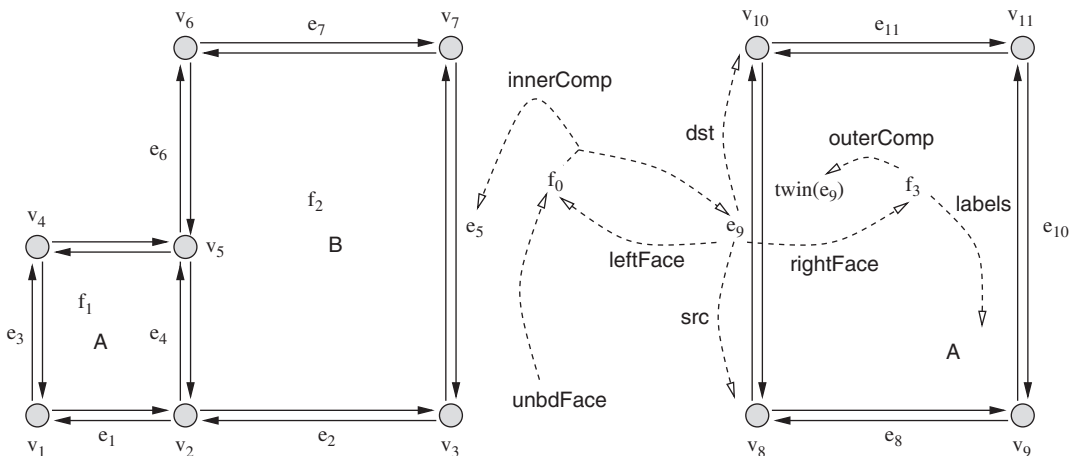


Figure 10. Example map and map data structure.

```

function OverlayIntersection( $M, e_1, e_2$ )
   $v := \text{EdgeIntersection}(e_1, e_2)$ 
  return  $\{(src(e_1), v), (src(e_2), v),$ 
     $(v, dest(e_1)), (v, dest(e_2))\}$ 
function OverlayCommonSegment( $M, e_1, e_2$ )
   $(v_1, v_2, v_3, v_4) :=$ 
    LexicalOrder( $src(e_1), dest(e_1), src(e_2), dest(e_2)$ )
   $S := \{(v_2, v_3)\}$ 
  if  $(v_1 \neq v_2)$   $S := S \cup \{(v_1, v_2)\}$ 
  if  $(v_3 \neq v_4)$   $S := S \cup \{(v_3, v_4)\}$ 
  return  $S$ 

```

Figure 11. Planarization of maps.

start (lexically smaller endpoint is v) and the edges that end (lexically greater endpoint is v) there. Initially, `InitialiseEventQueue` inserts all vertices of the input maps in this event queue, whereas intersections are added in the course of the plane sweep. Each time an edge is inserted or removed in the status structure S , it is checked whether the new neighbor edges intersect (`CheckIntersections`). If this is the case, a new intersection event is inserted in the event queue Q , encoded as a start and end event of all edges that run through the vertex.

The status structure (Fig. 14) \mathcal{S} contains the edges that are currently intersected by the sweep line. To handle degenerate cases, this is a list of lists of edges, where the sublists contain collinear edges ordered by the position of their right

```

function MapOverlay( $M_1, \dots, M_n$ )
   $M := \text{newMap}()$ 
   $curFaceAbove(\perp) := unbdFace(M)$ 
   $curRaw(\perp) := \sum_{i=1}^n labels(unbdFace(M_i))$ 
   $curLabels(\perp) := \text{computeLabels}(curRaw(\perp))$ 
   $labels(unbdFace(M)) := curLabels(\perp)$ 
  initialiseEventQueue( $M_1, \dots, M_n$ )
   $S := \langle \rangle$ 
  while  $(Q \neq \langle \rangle)$ 
     $(v_{last}, \mathcal{E}_{begin}, \mathcal{E}_{end}) := \text{head}(Q)$ 
     $vertices(M) := vertices(M) \cup \{v_{last}\}$ 
     $Q := \text{tail}(Q)$ 
     $f_{above} := curFaceAbove(Prev(SegAbove(S)))$ 
     $f_{below} := curFaceAbove(SegBelow(S))$ 
     $n_{end} := 0$ 
     $n_{begin} := 0$ 
    if  $(SegLowermost(S) \neq \perp)$ 
      HandleEdgesEnd()
     $S := \text{StatusRemoveSet}(S, \mathcal{E}_{end})$ 
     $S := \text{StatusInsertSet}(S, \mathcal{E}_{begin})$ 
    CheckIntersections( $Q, S$ )
    if  $(SegLowermost(S) \neq \perp)$ 
      HandleEdgesBegin()
    HandleMerge()
    if  $(n_{end} = 0 \text{ and } n_{begin} = 0)$ 
       $vertices(M) := vertices(M) \setminus \{v_{last}\}$ 
   $outerComp(unbdFace(M)) := \perp$ 
  return  $M$ 

```

Figure 12. Overlay algorithm.

```

function EventQueueInsert( $Q, v, \mathcal{E}_1, \mathcal{E}_2$ )
   $(v', \mathcal{E}'_1, \mathcal{E}'_2) := \text{HD}(Q)$ 
  switch
  case  $v' < v$ :
    return  $\text{HD}(Q) :: \text{EventQueueInsert}(\text{TL}(Q, v, \mathcal{E}_1, \mathcal{E}_2)$ 
  case  $v' = v$ :
    return  $(v, \mathcal{E}_1 \cup \mathcal{E}'_1, \mathcal{E}_2 \cup \mathcal{E}'_2) :: \text{TL}(Q)$ 
  case  $v' > v$ :
    return  $(v, \mathcal{E}_1, \mathcal{E}_2) :: Q$ 
function InitialiseEventQueue( $M_1, \dots, M_n$ )
   $Q := \langle \rangle$ 
  forall  $(i \in \{1 \dots n\})$ 
    forall  $(e \in \mathcal{E}_i)$ 
      if  $(src(e) < dest(e) = T)$ 
        EventQueueInsert( $src(e), \{e\}, \{\}$ )
        EventQueueInsert( $dest(e), \{\}, \{e\}$ )
      else
        EventQueueInsert( $dest(e), \{e\}, \{\}$ )
        EventQueueInsert( $src(e), \{\}, \{e\}$ )
function CheckIntersection( $Q, s_1, s_2$ )
   $v := \text{EdgeIntersection}(\text{HD}(s_1), \text{HD}(s_2))$ 
  if  $(v \neq \perp)$ 
    forall  $(e \in s_1, s_2)$ 
      if  $(\text{onEdge}(v, e) = T)$ 
        EventQueueInsert( $Q, v, \{e\}, \{e\}$ )
function CheckIntersections( $Q, S$ )
  if  $(\text{SegLowermost}(S) \neq \perp)$ 
    if  $(\text{SegBelow}(S) \neq \perp)$ 
      CheckIntersection( $Q, \text{SegBelow}(S), \text{SegLowermost}(S)$ )
    if  $(\text{SegAbove}(S) \neq \perp)$ 
      CheckIntersection( $Q, \text{SegUppermost}(S), \text{SegAbove}(S)$ )
  else
    if  $(\text{SegBelow}(S) \neq \perp \text{ and } \text{SegAbove}(S) \neq \perp)$ 
      CheckIntersection( $Q, \text{SegBelow}(S), \text{SegAbove}(S)$ )

```

Figure 13. Overlay algorithm: event queue.

end vertices (this makes the detection of intersection points in `CheckIntersection` more efficient). The individual lists are ordered by the y -value of their intersection point with the current sweep line; if ties remain, the list with the greater slope is considered to be greater. Vertical lines are always greater than any other edges, sorted by the y -value of the upper point. The function `StatusCompare` implements all these rules.

The border segment functions `SegBelow(S)`, `SegLowermost(S)`, `SegUppermost(S)`, and `SegAbove(S)` determine elements of the status structure. `SegBelow(S)` is the list of edges below the event point, and `SegAbove(S)` is the list above it. `SegLowermost(S)` is the first list of edges that contains the event point on its left side (right side), and `SegUppermost(S)` is the last list. If no corresponding list can be found, \perp will be returned. In Figure 15, the value of these functions are shown for the two situations: Just before the marked point, the following equations hold:

- $\text{SegBelow}(S) = s_0$
- $\text{SegLowermost}(S) = s_1$
- $\text{SegUppermost}(S) = s_2$
- $\text{SegAbove}(S) = s_3$,

```

function StatusCompare( $e_1, e_2$ )
   $y_1 := YValue\ At\ (e_1, X_{vlast})$ 
   $y_2 := YValue\ At\ (e_2, X_{vlast})$ 
  if ( $y_1 = \perp$  and  $y_2 = \perp$ ) return 0
  if ( $y_1 = \perp$  and  $y_2 \neq \perp$ )
    if ( $y_2 < Y_{vlast}$ ) return T else return F
  if ( $y_1 \neq \perp$  and  $y_2 = \perp$ )
    if ( $y_2 < Y_{vlast}$ ) return F else return T
  if ( $y_1 \neq y_2$ )
    if ( $y_1 < y_2$ ) return T else return F
   $s_1 := slope(e_1)$ 
   $s_2 := slope(e_2)$ 
  if ( $y_1 < Y_{vlast}$ )
    if ( $s_2 < s_1$ ) return T else return F
  else
    if ( $s_1 < s_2$ ) return T else return F
  return U

function StatusInsertCollinear( $s, e$ )
  if ( $src(e) < HD(s)$  and  $dest(e) < HD(s)$ )
    return  $HD(S) :: StatusInsertCollinear(TL(S), e)$ 
  else
    return  $s :: TL(S)$ 

function StatusInsert( $S, e$ )
  switch (StatusCompare( $HD(S), (e)$ ))
  case T:
    return  $HD(S) :: StatusInsert(TL(S), e)$ 
  case U:
    return StatusInsertCollinear( $s, e$ ) ::  $TL(S)$ 
  case F:
    return  $\langle e \rangle :: S$ 

function StatusInsertSet( $(, S), \mathcal{E}$ )
  forall ( $e \in \mathcal{E}$ )
     $S := StatusInsert((, S), e)$ 
  return  $S$ 

```

Figure 14. Overlay algorithm: status structure.

and after the event point:

- $SegBelow(S) = s'_0$
- $SegLowermost(S) = s'_1$
- $SegUppermost(S) = s'_2$
- $SegAbove(S) = s'_3$.

In our approach, the creation of the resulting map and the labeling of the faces is handled differently compared to

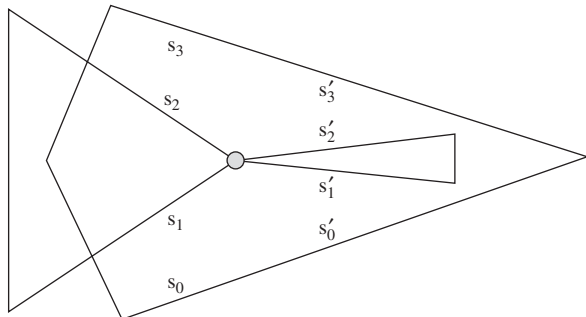


Figure 15. The border segments functions.

```

function HandleEdgesEnd()
   $s := SegLowermost(S)$ 
  if ( $curFaceAbove(prev(s)) = curFaceAbove(s)$ )
     $e_{last} := newEdge()$ 
     $edges(M) := edges(M) \cup \{e\}$ 
     $src(e) := curStart(s)$ 
     $dest(e) := v$ 
     $leftFace(e) := curFaceAbove(s)$ 
     $rightFace(e) := curFaceAbove(Prev(s))$ 
     $n_{end} := n_{end} + 1$ 
  for ( $s := next(SegLowermost(S)) \dots SegUppermost(S)$ )
    if ( $curFaceAbove(prev(s)) = curFaceAbove(s)$ )
       $e_{last} := newEdge()$ 
       $edges(M) := edges(M) \cup \{e\}$ 
       $src(e) := curStart(s)$ 
       $dest(e) := v$ 
       $leftFace(e) := curFaceAbove(s)$ 
       $rightFace(e) := curFaceAbove(prev(s))$ 
       $n_{end} := n_{end} + 1$ 
       $f := curFaceAbove(prev(s))$ 
       $outerComp(f) = e_{last}$ 

function HandleEdgesBegin()
   $s := SegLowermost(S)$ 
   $s_{last} := SegLowermost(S)$ 
   $f_{last} := f_{below}$ 
   $curStart(s) := v_{last}$ 
   $curRaw(s) := curRaw(prev(s)) + LabelDiff(s)$ 
   $labels(s) := computeLabels(curRaw(s))$ 
  if ( $curLabels(s) \neq curLabels(prev(s))$ )
     $n_{begin} := n_{begin} + 1$ 
  for ( $s = next(SegLowermost(S)) \dots SegUppermost(S)$ )
     $curStart(s) := v_{last}$ 
     $curRaw(s) := curRaw(prev(s)) + LabelDiff(s)$ 
     $curLabels(s) := computeLabels(curRaw(s))$ 
    if ( $curLabels(s) \neq curLabels(prev(s))$ )
       $n_{begin} := n_{begin} + 1$ 
      if ( $labels(f_{last}) \neq curLabels(prev(s))$ )
         $f_{last} := newFace()$ 
         $faces(M) := faces(M) \cup \{f_{last}\}$ 
         $labels(f) := curLabels(Prev(s))$ 
        for ( $t = s_{last} \dots prev(s)$ )
           $curFaceAbove(t) = f_{last}$ 
         $s_{last} = s$ 
      for ( $t = s_{last} \dots SegUppermost(S)$ )
         $curFaceAbove(t) = f_{above}$ 

```

Figure 16. Overlay algorithm: edge handling.

```

function HandleMerge()
  if ( $n_{end} > 0$  and  $n_{begin} = 0$ )
    if ( $f_{below} = f_{above}$ )
       $innerComp(f_{below}) :=$ 
         $innerComp(f_{below}) \cup \{e_{last}\}$ 
    else
       $innerComp(f_{below}) :=$ 
         $InnerComp(f_{below}) \cup innerComp(f_{above})$ 
       $faces(M) := faces(M) \setminus \{f_{above}\}$ 
      /* relink all  $f_{above}$  pointers to  $f_{below}$  */

```

Figure 17. Overlay algorithm: face merging.

Ref. [14] (Fig. 16): A vertex and the edges that connect it to the part of the result that has already been computed (on the left) are created at each event point (HandleEdgesEnd). $curFaceAbove(s)$ always points to the face that is above a list s of collinear edges at the current sweep line position. This information is used to determine the face references $leftFace(f)$ and $rightFace(f)$.

If at least two edges start at the current event point (HandleEdgesBegin), a new face is created in between. Its labels are computed by taking the labels of the adjacent face and adding the differences caused by the edge in between. The multiset $curRaw(s)$ stores the sum of all labels, whereas $curLabels(s)$ gives the actual labeling that is defined by the desired operation. For example, if the union of maps should be calculated, at least one map must contribute a label to the corresponding $curRaw(s)$ entry. For the intersection, all maps must contribute to add this label to the actual label set $curLabels(s)$.

One event (HandleMerge) must be handled separately. It may be the case that there may be no edges ending at the current event point. At these merge points, two cases must be distinguished. The face above and the face below the current event point must then be the same since there is no edge dividing them on the right hand side of the event point. Nevertheless, the faces may be independently created by the algorithm. In this case, the two branches are merged by deleting the face f_{above} and all references to it. This can be done very efficiently if indirect references are used for the face pointers. In the remaining case, the current event point marks a right end of an inner component, to which the face can be linked at this point.

3.2. Conservative Rounding

In the previous section, we have presented our version of the map overlay algorithm in full detail, so that also degenerate cases have been handled. In this section, we address the problem of implementing this algorithm on processors with only a limited precision of arithmetic operations.

Limited precision arithmetic generally poses problems to computational geometry algorithms. In particular, the naive use of floating point arithmetic is dangerous and leads to wrong results [26]. For example, due to rounding errors, the convex hull of a set of collinear points may consist of more than two points, or non-collinear lines may have more than one intersection point. The basic problem is that the rounding of numbers follows general rules (round towards zero, round to nearest, etc.) that are fatal for some algorithms.

In our application of the map overlay algorithm, the face labels mark inaccessible areas for some objects. If a path were chosen that leads through the original, but not through the rounded area, the rounding would therefore cause a crash. Hence, traditional snap-rounding algorithms [20] relying on simple rounding methods are not applicable.

Since the correct calculation of certain geometric primitives is a crucial point of geometric algorithms, various techniques have been proposed to solve this special problem. As evaluating the sign of geometric primitives can be done without exactly computing its value, so-called floating point filters have been proposed [17, 33]. However, these

approaches are generally not able to compute geometric objects like intersection points.

Hence, without further modifications to the map overlay algorithm, arbitrary-precision arithmetic would be required for its implementation. However, this is too complex for non-trivial examples. Consider the sequence of overlay operations $O_1 \dots O_n$, where the result of O_i is a map M_i that is taken to build the inputs of the following step O_{i+1} . Assume that the positions of the vertices \mathcal{V}_i in step i are represented by an integer of n bits. Then, the vertices \mathcal{V}_{i+1} have either been in one of the sets of vertices before or a new vertex is created at the position of an intersection of two edges, which requires about $4n$ bits ($2n$ for both the numerator and the denominator) for its representation. As a consequence, the size of the results grows exponentially (about $n \cdot 4^i$ bits are needed in step i). This exponential blowup fills the available main memory after a few steps as we experienced with an implementation based on the GNU Multi Precision library [40].

A better approach is to limit the precision of the numbers after each step in a way that complies with the application requirements. If the position of all vertices are rounded to values of the initial precision and all geometric predicates of a step can be computed correctly for this precision, a solution for the problem is found. This process can be visualized by some kind of grid snapping, where the cross-over points represent all positions that are representable in the initial precision.

As all positions must be rounded in a way that the labeled areas of the rounded map cover at least the original area, the algorithm cannot simply choose one of the neighbor grid points as shown in Figure 18 (since for all neighbors a part of the original area is cut off). Instead, some vertex at a further distance must be chosen (case (b) of Fig. 18). We therefore formulate the following problem.

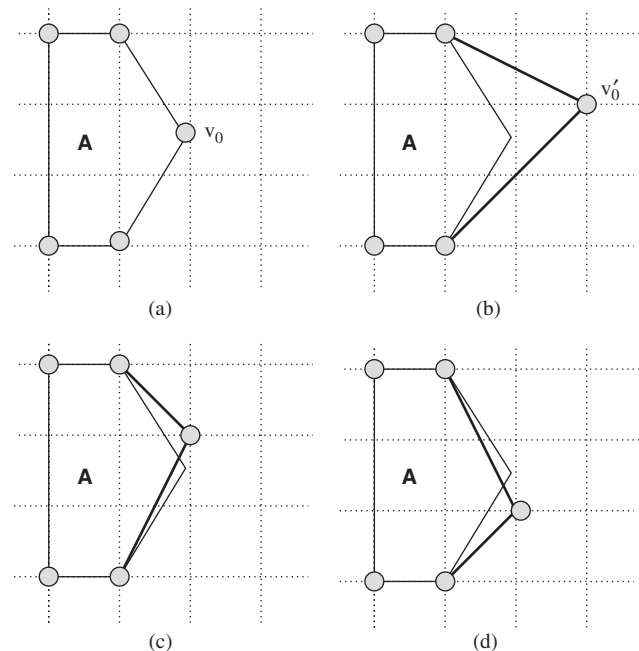


Figure 18. Rounding a vertex.

DEFINITION 3 (Conservative Grid Snapping). For a given map $M = (\mathcal{V}, \mathcal{E}, \mathcal{F})$, the conservative grid snapping problem consists of finding an approximation $M' = (\mathcal{V}', \mathcal{E}', \mathcal{F}')$ with

- \mathcal{V}' only contains vertices on the grid.
- For each label ℓ , all faces of \mathcal{F}' that are labelled with ℓ cover at least the area that has been labelled by the ℓ faces of \mathcal{F} .

Needless to say, grid snapping involves some error that should be kept as small as possible by the rounding algorithm. Moreover, it should be clear that finding the optimal solution is very complex. Hence, we are interested in a fast and simple heuristic algorithm, which can be used in the embedded domain.

To solve this problem, we approximate a vertex that is not on the grid by several vertices that lie on the grid around the original vertex. Despite this simple idea, the algorithm is a bit more complicated, since all inputs must be considered. In the following, we will restrict our consideration to maps that have only one label. Multi-colored maps can be handled as well by rounding the vertices for each color separately.

The algorithm given in Figure 19 does not round all vertices separately. For each rounding step, a chain of vertices that are within the same grid box are processed simultaneously. The algorithm first identifies the vertices that precede and follow the chain that is rounded (with the face to be conservatively rounded to the left). For them, an outcode similar to the one of the Cohen-Sutherland-algorithm [16, 35] is computed. Based on this outcode, we determine an entry code and an exit code for both of the two adjoining vertices. Depending on these codes, a sequence of the corner points of the grid box is taken to approximate the map, where the basic idea is to walk around the grid box in

```

function RoundVertices( $w, e_1, e_2$ )
   $v_1 := src(e_1)$ 
   $v_2 := dest(e_1)$ 
   $v_3 := src(e_2)$ 
   $v_4 := dest(e_2)$ 
   $(c_0, c_1) := \text{EntryExit}(w, v_1, v_4)$ 
  if  $(c_0 = c_1 \text{ and } \text{return}(v_1, v_2, v_3) \vee$ 
     $\text{collinear}(v_1, v_2, v_3) \wedge \text{return}(v_1, v_3, v_4))$ 
    return  $\langle v_1, v_4 \rangle$ 
  else
    switch  $(c_0)$ 
    case NORTH:
      if  $(c_1 = \text{WEST})$ 
        return  $\langle v_1, \text{NW}(w), v_4 \rangle$ 
      else if  $(c_1 = \text{SOUTH})$ 
        return  $\langle v_1, \text{NW}(w), \text{SW}(w), v_4 \rangle$ 
      else if  $(c_1 = \text{EAST})$ 
        return  $\langle v_1, \text{NW}(w), \text{SW}(w), \text{SE}(w), v_4 \rangle$ 
      else
        return  $\langle v_1, \text{NW}(w), \text{SW}(w), \text{SE}(w), \text{NE}(w), v_4 \rangle$ 
    case EAST: ...
    case SOUTH: ... /* other cases analogously */
    case WEST: ...

```

Figure 19. Vertex rounding algorithm.

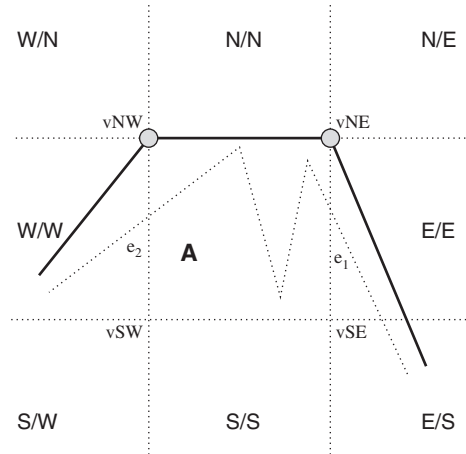


Figure 20. Entry/exit codes and example.

counter-clockwise direction. Figure 20 shows the mapping between the corner points and the entry and exit codes.

However, there are two special cases that cannot be handled by this approach and therefore require a separate consideration. The first case occurs when there are no preceding and following vertices, i.e., the contour is completely inside a grid box. In this case, this grid box is the approximation. In the second case, the preceding and following vertices have the same code, and the part describes a concave part of the face (Fig. 21). Then, no local approximation is possible. A very simple solution is to just remove all intermediate vertices and to insert an edge that directly connects the preceding vertex to the following vertex. Although tests have shown that this leads to good results, the error caused by this rounding might be unacceptable. Therefore, we integrated the following alternative. Assume that the edges both cross the right border of the grid box (w.l.o.g.), then we choose the rounding point as follows: First, we determine the slopes for the incoming and outgoing edges s_{in} and s_{out} and calculate the x -value where both edges have at least a distance of the height of the grid box $x_{NE} - x_{SW}$. Assuming the worst case, i.e., the edges almost touch at the right

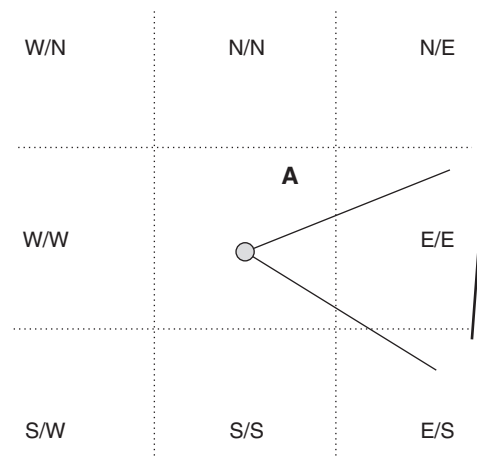


Figure 21. Entry/exit codes and example.

border of the grid box, the vertex v can be approximated by $v = (x', y')$ with

$$\begin{aligned} x' &= \left\lceil x_{NE} + \frac{x_{NE} - x_{SW}}{s_{in} - s_{out}} \right\rceil \\ y' &= \lceil yValueAt(e_{in}, x') \rceil \end{aligned} \quad (2)$$

With the exception of the last optimization, the algorithm does not rely on a specific grid. In particular, the grid does not need to be equidistant. Hence, it can be used for integers, and also for rational and floating point numbers with limited precision. In particular, we can use floating points with a quarter of the available precision for the input vertices positions and round the results. Another possibility is to use a double or quarter-precision arithmetic as already described in detail by Knuth [23].

Unfortunately, vertices cannot be simply moved or replaced in a map as done in the algorithm above, since the planarity property of the map may be lost by overlapping faces. Therefore, after the rounding step, another overlay of the rounded map must be computed. That may create vertices that are not on the grid, which requires the rounding step to be run again. Our experiments have shown that in more than 96%, two iterations are sufficient.

3.3. Map Simplification

Embedded systems generally possess very limited resources with respect to computation power and memory. As they often have to meet real-time requirements, the simplification of maps is an integral part of the design of an overlay algorithm for embedded devices. The aim of the simplification operation is the elimination of vertices, edges, or faces to reduce the overall computation complexity. Similar to the rounding of vertices, the simplification of the maps must be conservative, i.e., labelled areas must not get smaller. Clearly, the simplified map should resemble the old map while using less vertices and edges.

Generally, any bounded labeled area of a map can be approximated with three vertices forming a triangle that covers the colored area. A very simple approximation requiring four vertices is the bounding box, a rectangle aligned to the coordinate system that totally covers the labelled area. Moreover, if the map has a lot of jagged areas, the convex hull of them is good simplification. All these approximations can be efficiently calculated, however, they are too imprecise for practical usage. Therefore, we use again a local approach to simplify maps iteratively. It is based on the following two operations (Fig. 22):

- (a) Remove a vertex at a concave position of a labeled area (including redundant vertices that connect two collinear edges). The error of this operation is

$$\varepsilon = \frac{1}{2} \vec{e}_1 \times \vec{e}_2 \quad (3)$$

where $\vec{a} \times \vec{b} = \vec{a} \cdot \vec{b}^\perp = x_a y_b - x_b y_a$.

- (b) Replace two vertices by a single one at a convex position of a colored area. The position of the new vertex is the intersection of the lines through the two adjoining edges e_1 and e_3 , which can be computed as

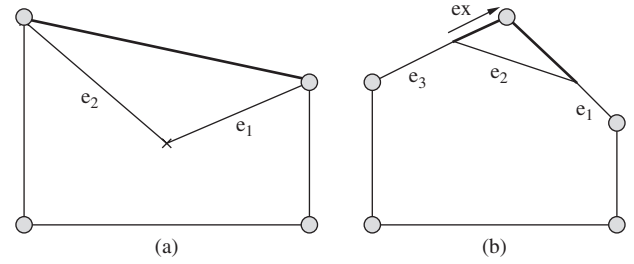


Figure 22. Basic simplification operations.

follows: Since $\lambda \vec{e}_1 - \mu \vec{e}_3 = \vec{e}_2$ (for $\lambda, \mu \in \mathbb{Q}$ with $\lambda > 0, \mu < 0$), Cramer's rule can be applied to compute λ, μ and e_x :

$$\lambda = \frac{\vec{e}_2 \times \vec{e}_3}{\vec{e}_1 \times \vec{e}_3} \quad \mu = -\frac{\vec{e}_1 \times \vec{e}_2}{\vec{e}_1 \times \vec{e}_3} \quad e_x = \frac{\vec{e}_1 \times \vec{e}_2}{\vec{e}_1 \times \vec{e}_3} \cdot \vec{e}_3 \quad (4)$$

Hence, the error of the simplification step is

$$\varepsilon = \frac{1}{2} \left(\frac{\vec{e}_1 \times \vec{e}_2}{\vec{e}_1 \times \vec{e}_3} \cdot \vec{e}_3 \times \vec{e}_2 \right) \quad (5)$$

The simplification of a map can be either performed with respect to an upper bound of the error or w.r.t. to a projected number of vertices. Whereas the first strategy is good for a general speed-up of the computation, for our application, we have chosen the second approach. Since our application must hold tight deadlines, the worst case execution time of the map overlay, which is primarily affected by the input complexity, must be restricted. Moreover, an unlimited number of vertices may cause the embedded system to run out of memory. Needless to say, the results may become less useful after the simplification. However, for hard real-time systems like our application, violating the deadline is worse than any approximated result. In order that other components of the system can judge how trustworthy the results of a map overlay are, the simplification procedure additionally returns the computed error.

However, even if we only consider the above operations for the simplification, finding the optimal map with a minimal number of vertices that covers a given map is still a complex problem. Since the removal of a vertex changes the errors that are made by the removal of other vertices, the problem apparently falls into the class NP. Hence, we implemented a heuristic following a greedy approach: among all vertices, the one with the least error is selected. This is repeated until the desired number of vertices is reached. With this approximation process, the best solution for a given system and a situation is calculated.

Unfortunately, the simplification suffers from the same problem as the rounding. After each approximation step, the map may no longer be planar: we must compute the overlay again after the simplification. Even worse, new vertices may not be on the grid, so that the rounding must be done as well. This leads to the natural sequence of both operations: first, simplify the polygons and then align them to the grid.

4. IMPLEMENTATION

For the implementation, we use the definitions of the previous sections. To use the same techniques for the specification and the implementation has various obvious advantages. First, the specification and the actual implementation are as close as possible, since all primitives are implemented in software in the same way as they were defined in the HOL theory. Second, algorithms are more compact, because several cases of the two-valued formalization can be merged in the three-valued setting. We implemented a software library in C based on three-valued logics and rational numbers. With the help of the GMP library, we perform all calculations with arbitrary precision. The following paragraphs sketch the implementation.

4.1. Three-Valued Primitives

The truth values of the three valued logic are represented by a signed integer, and the logical functions are implemented by their truth tables.

```
typedef signed char log3;
/* F=-1, U=0, T=1 */

const log3 not3_table[] = {T,U,F};
log3 not3( log3 a ) { return not3_table[a+1] }

const log3 and3_table[] = {{F,F,F},{F,U,U},{F,U,T}};
log3 and3( log3 a, log3b ) { return and3_table[a+1][b+1] }
```

The existence of an intersection point of two line segments can be computed as follows:

```
log3 do_seg_intersect ( line l1, line l2 )
{
  return and3(
    equ3(
      lturn( l1.beg,l1.end,l2.beg ),
      rturn( l1.beg,l1.end,l2.end )
    ),(
      lturn( l2.beg,l2.end,l1.beg ),
      rturn( l2.beg,l2.end,l1.end )
    )
  )
}
```

If both segments truly intersect, T is returned. If they only touch or have a common line segment, U is returned, since this is a degenerated case. If there are no common points, F is returned. This corresponds to the function $\exists x \cdot \text{onSegment}(l_1, x) \wedge \text{onSegment}(l_2, x)$.

As another example, reconsider the *winding number* algorithm, which is described in Section 2.3. A classic implementation of this algorithm that directly deals with degenerate cases cannot avoid the case distinctions shown in Figure 2. The position of the previous and following points must be taken into account, which leads to even more subcases. With the help of a three-valued intersection, the algorithm can be formulated much more easily. Three-valued primitives eliminate all degeneracies. This clearly demonstrates the benefits of our approach.

Provided that $e[i]$ denotes the i -th of n edges of a polygon and $\text{xRay}(v)$ is the ray from v to the right, the following C fragment calculates the winding number w of a point v .

```
w = 0;
for( i = 0 ; i < n ; i++ )
  if( doIntersect( xRay(v),e[i] ) >= U )
  {
    w += below( v, e[i].begin );
    w += above( v, e[i].end );
  }
w = w / 2;
```

5. CONCLUSIONS

In this paper, we dealt with two problems. The first one is the verification of polygon processing algorithms. We propose the use of three-valued logic to develop dependable algorithms for geometric applications to be used in embedded systems. The use of a third truth value allows us to make degenerate cases explicit so that they can be appropriately handled by different algorithms. Starting from applications like motion planning and collision detection, we dealt with basic geometric objects and primitives used in analytical geometry. The obtained software library has been formalized in the interactive theorem prover HOL. Moreover, we used HOL to formally reason about the geometric predicates in order to assure their consistent use. In particular, we are able to verify entire algorithms that are used in embedded systems in order to guarantee that required safety properties like avoidance of collisions are met. Furthermore, since degenerate cases are succinctly described by three-valued predicates, the derived algorithms are both robust and compact, and their results are to a large extent independent of the numeric precision of the underlying microprocessors.

The second problem area that we considered is implementation problems that are caused by the limited precision of arithmetics and limited memory. Based on a map overlay algorithm that had been particularly designed for safety-critical embedded systems (including the treatment of degenerate cases), we solved two significant problems that are posed by our application domain. First, conservative rounding strategies take care that limited precision arithmetics comply with the safety requirements of the system. Second, map simplification limits the time and space complexity of the repeated use of the map overlay algorithm to guarantee the execution with predefined execution time and memory limits.

REFERENCES

1. M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm, “Cache Behavior Prediction by Abstract Interpretation,” In Static Analysis Symposium (SAS), Springer, Aachen, Germany, 1996, Vol. 1145, pp. 52–66.
2. G. Berry, The Constructive Semantics of Pure Esterel. <http://www.sop.inria.fr/esterel.org>, July 1999.
3. L. Bolc and P. Borowik, Many-Valued Logics, Springer, 1992.
4. J. Brandt and K. Schneider, “Dependable Polygon-Processing Algorithms for Safety-Critical Embedded Systems” (L. T. Yang, M. Amamiya, Z. Liu, M. Guo, and F. J. Rammig, Eds.), International Conference on Embedded and Ubiquitous Computing (EUC), Springer, Nagasaki, Japan, 2005, Vol. 3824, pp. 405–417.
5. J. Brandt and K. Schneider, Using Three-Valued Logic to Specify and Verify Algorithms of Computational Geometry (K.-K. Lau and

- R. Banach, Eds.), International Conference on Formal Engineering Methods (ICFEM), Springer, Manchester, UK, 2005, Vol. 3785, pp. 405–420.
6. J. Brandt and K. Schneider, “Efficient Map Overlay for Safety-Critical Embedded Systems,” In IEEE Symposium on Industrial Embedded Systems, Antibes, France, 2006.
 7. J. A. Brzozowski and C.-J. H. Seger, *Asynchronous Circuits*, Springer, 1995.
 8. C. Burnikel, K. Mehlhorn, and S. Schirra, “On Degeneracy in Geometric Computations,” In Symposium on Discrete Algorithms (SODA), ACM, Arlington, Virginia, USA, 1994, pp. 16–23.
 9. K. Bühler, E. Dyllong, and W. Luther, “Reliable Distance and Intersection Computation Using Finite Precision Geometry” (R. Alt, A. Frommer, R. Baker Kearfott, and W. Luther, Eds.), Numerical Software with Result Verification, Dagstuhl Castle, Springer, Germany, 2004, Vol. 2991, pp. 160–190.
 10. B. Chazelle and H. Edelsbrunner, *Journal of the ACM* 39, 1 (1992).
 11. S.-C. Chou, “Mechanical Geometry Theorem Proving, Mathematics and Its Applications.” D. Reidel Publishing Company, Dordrecht, Boston, Lancaster, Tokyo, 1988.
 12. S. C. Chou, X. S. Gao, and J. Z. Zhang, “Machine Proofs in Geometry.” World Scientific, Singapore, 1994.
 13. A. Church, *Journal of Symbolic Logic* 5, 56 (1940).
 14. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf, *Computational Geometry*, Springer, 2000.
 15. H. Edelsbrunner and E. P. Mücke, *ACM Transactions on Graphics* 9, 66 (1990).
 16. J. D. Foley, A. van Dam, S. K. Feiner, and J. F. Hughes, *Computer Graphics: Principles and Practice*, Addison Wesley, 2000.
 17. S. Fortune and C. J. Van Wyk, “Efficient Exact Arithmetic for Computational Geometry,” In Symposium on Computational Geometry, 1993, pp. 163–172.
 18. M. J. C. Gordon and T. F. Melham, “Introduction to HOL: A Theorem Proving Environment for Higher Order Logic.” Cambridge University Press, 1993.
 19. M. Grimmer, K. Petras, and N. Revol, “Multiple Precision Interval Packages: Comparing Different Approaches” (R. Alt, A. Frommer, R. Baker Kearfott, and W. Luther, Eds.), Numerical Software with Result Verification, Springer, Dagstuhl Castle, Germany, 2004, Vol. 2991, pp. 64–90.
 20. J. Hobby, *Computational Geometry* 13, 199 (1993).
 21. K. Hormann and A. Agathos, *Computational Geometry: Theory and Applications* 20, 131 (2001).
 22. S. C. Kleene, *Introduction to Metamathematics*, North Holland, 1952.
 23. D. E. Knuth, “The Art of Computer Programming.” Addison Wesley, 1998, Vol. 2.
 24. M. Lin and S. Gottschalk, “Collision Detection Between Geometric Models: A Survey,” In Proceedings of IMA Conference on Mathematics of Surfaces, 1998, pp. 37–56.
 25. S. Malik, “Analysis of Cyclic Combinational Circuits,” In Conference on Computer Aided Design (ICCAD), IEEE Computer Society, Santa Clara, California, November 1993, pp. 618–625.
 26. K. Mehlhorn and S. Näher, *The LEDA Platform of Combinatorial and Geometric Computing*, Cambridge University Press, 1999.
 27. J. Narboux, “A Decision Procedure for Geometry in Coq” (K. Slind, A. Bunker, and G. Gopalakrishnan, Eds.), International Conference on Theorem Proving in Higher Order Logics (TPHOL), Springer, Park City, Utah, USA, 2004, Vol. 3223, pp. 225–240.
 28. D. Pichardie and Y. Bertot, “Formalizing Convex Hull Algorithms” (R. J. Boulton and P. B. Jackson, Eds.), Higher Order Logic Theorem Proving and Its Applications (TPHOL), Springer, Edinburgh, Scotland, UK, 2001, Vol. 2152, pp. 346–361.
 29. D. M. Priest, “Algorithms for Arbitrary Precision Floating Point Arithmetic” (P. Kornerup and D. W. Matula, Eds.), Symposium on Computer Arithmetic, IEEE Computer Society, 1991, pp. 132–144.
 30. T. W. Reps, M. Sagiv, and R. Wilhelm, “Static Program Analysis via 3-Valued Logic” (R. Alur and D. A. Peled, Eds.), Conference on Computer Aided Verification (CAV), Springer, Boston, MA, USA, 2004, Vol. 3114, pp. 15–30.
 31. K. Schneider, J. Brandt, T. Schuele, and T. Tuerk, “Improving Constructiveness in Code Generators,” In Synchronous Languages, Applications, and Programming (SLAP), Edinburgh, Scotland, UK, 2005.
 32. K. Schneider, J. Brandt, T. Schuele, and T. Tuerk, “Maximal Causality Analysis,” In Conference on Application of Concurrency to System Design (ACSD), IEEE Computer Society, St. Malo, France, June 2005, pp. 106–115.
 33. J. R. Shewchuk, *Discrete and Computational Geometry* 18, 305 (1996).
 34. T. R. Shiple, *Formal Analysis of Synchronous Circuits*, Ph.D. Thesis, University of California at Berkeley, 1996.
 35. I. E. Sutherland and G. W. Hodgeman, *Communications of the ACM* 17, 32 (1974).
 36. A. Tarski, “A Decision Method for Elementary Algebra and Geometry.” University of California, 1951.
 37. W.-T. Wu, *Scientia Sinica* 21, 157 (1978).
 38. CGAL: Computational Geometry Algorithms Library, August 2006. Available at <http://www.cgal.org/>.
 39. Coq Proof Assistant, August 2006. Available at <http://coq.inria.fr/>.
 40. GNU Multiprecision Library: GNU MP, August 2006. Available at <http://www.swox.com/gmp/>.
 41. HOL Kananaskis, August 2006. Available at <http://hol.sourceforge.net>.
 42. The LEDA Platform of Combinatorial and Geometric Computing, August 2006. Available at <http://www.mpi-sb.mpg.de/~mehlhorn/LEDAbook.html>.
 43. Moscow ML, August 2006. Available at <http://www.dina.kvl.dk/~sestoft/mosml.html>.