Dependable Polygon-Processing Algorithms for Safety-Critical Embedded Systems

Jens Brandt and Klaus Schneider

University of Kaiserslautern
Reactive Systems Group, Department of Computer Science
P.O. Box 3049, 67653 Kaiserslautern, Germany
http://rsg.informatik.uni-kl.de

Abstract Algorithms that process geometric objects become more and more important for many safety-critical embedded systems, e.g. for motion planning or collision detection, where correctness is indispensable. The main challenge to demonstrating correctness is the consistent handling of degenerate cases like collinear line segments. In this paper, we therefore propose the use of an interactive theorem prover to develop dependable geometry algorithms for safety-critical embedded systems. Our solution is based on the use of a three-valued logic to make degenerate cases explicit. Using the theorem prover, we are not only able to prove the correctness of the obtained algorithms, but also to directly derive a software library of provably correct geometry algorithms for safety-critical applications.

1 Introduction

Applications like motion planning in robotics or collision detection of autonomous vehicles have to observe and to control the positions of certain objects in a particular environment. To this end, it is often sufficient to model the environment as a two-dimensional plane and the objects can be approximated as polygons on that plane. For this reason, fundamental algorithms of computational geometry [7] like polygon clipping or computing convex hulls can be used to develop embedded systems for these applications. As these embedded systems work in safety-critical domains, where even human live could be endangered, it is mandatory to guarantee their correctness. For this reason, it is necessary to built up specification languages and verification formalisms that allow the designers of such systems to reason about their correctness.

Geometric problems appear to be easy, since they can be visualised in a natural and intuitive way. However, at a second glance, even simple definitions turn out to be more complicated than expected: For example, what is the intersection point of two lines, if both lines are identical? For this reason, most algorithms work under certain preconditions like ‘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, we recommend the use of interactive theorem provers to reason about required definitions and algorithms. However, establishing consistent definitions for geometric primitives that can be universally used for all algorithms and even degenerate inputs turns out to be a great challenge [8]. For example, to define whether a point on the edge of a polygon belongs to the interior or not, yields essential problems when this definition is used in one or the other algorithm. These

problems impose inherent difficulties to develop a software library for safety-critical embedded systems to process geometric data.

In our opinion, the best solution is to make degenerate cases explicit, which requires the introduction of a third truth value for geometric primitives that normally are either true or false. We therefore extended the interactive theorem prover HOL [10] by a theory on two-dimensional analytic geometry [3] that is based on a three-valued logic. We formalised geometric objects, geometric primitives, and finally entire algorithms, so that we are able to reason about the correctness of geometric computations. As a first experiment, we successfully verified the Cohen-Sutherland clipping algorithm [9].

Though reasoning about geometric problems has a long tradition, our work deviates from existing work in several ways. In particular, Wu's work [21,6] on translating geometric propositions to algebraic forms, i.e. equations between polynomials is well-known. However, this approach is limited to the automated solution of particular instances of geometric problems, while we have to argue about the problems (and their algorithmic solutions) themselves. The work closest to ours is [15], where also an interactive theorem prover is used to formally reason about geometric problems. However, in contrast to our work, they used two-valued logic to formalise geometric primitives. To circumvent problems of degenerate cases, they modified definitions or used techniques like perturbation which, however, turned out to be problematic in practise [5].

In this paper, we therefore propose to use three-valued logic for the implementation of algorithms that process geometric data. Three-valued logic has proved to be an adequate tool in many areas of computer science [4,20,13,2,18,16,1,19]. We show that three-valued logic is well suited to define geometric primitives so that degenerate cases are consistently and concisely handled. To guarantee the correctness of the derived algorithms, we employ the interactive theorem prover HOL [10] to set up a library on two-dimensional, linear objects like lines, segments, and polygons [3]. Based on this library for the theorem prover, we derive a dependable software library that provides algorithms for geometric problems that have to be solved in upcoming embedded systems.

The paper is organised as follows: In Section 2, the specification and verification of polygon processing algorithms is described. Section 3 illustrates our formalisation approach by the verification of the Cohen-Sutherland algorithm for line clipping. Section 4 sketches the implementation of a software library for algorithms of computational geometry based on our three-valued formalisations. Finally, Section 5 draws some preliminary conclusions.

An extended version of this paper, the proof code, additional material, as well as detailed descriptions of all theories and examples described in this paper are available on our website.

2 Specification

In this section, we describe our three-valued formalisation of analytic geometry. We first consider degenerate cases and three-valued logic (Section 2.1) and then more complex geometric objects and primitives (Section 2.2).

2.1 Degenerate Cases and Three-Valued Logic

Most algorithms of computational geometry are designed for the 'general case', thus excluding so-called degenerate cases: 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 [8,14]. Handling degenerate cases is a well-known problem in computational geometry.

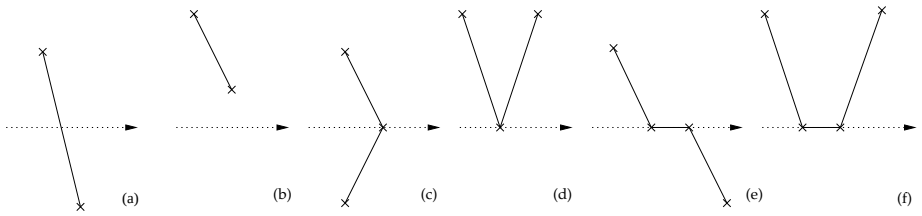

Figure 1. Winding number algorithm

As an example, consider the winding number algorithm: It calculates the winding number of a point p relative to a polygon P, i.e. how often the directed sequence of connected line segments of P turns around p (see [11] for a precise definition). To this end, it counts the intersections of an arbitrary ray starting in p with edges of the polygon P. If the intersected edge runs from the bottom to the top, it is counted positively - in the opposite direction negatively. Figure 1 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 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 the presented examples, cases (c) and (e) have to be defined as intersections, whereas cases (d) and (f) should not be intersections.

The explicit treatment of degeneracies requires substantial effort. In particular, to decide whether all cases have been considered is often unclear and nontrivial. Simple solutions like perturbation or other techniques have shown not to be as applicable as desired [5], so that degeneracies still form an impediment for the development of dependable algorithms: On the one hand, the enormous number of different degenerate cases does not allow an explict treatment, and on the other hand, consistent definitions of geometric primitives that implicitly handle all degenerate cases for all algorithms in an adequate way are not available, and probably do not exist: For example, consider the problem to determine whether a point is inside or outside a polygon. Assume that the points on the edge are considered to be outside the polygon (i.e. polygons are 'open' point sets). If we calculate the difference of two polygons as a set difference, the result is possibly a polygon that contains points on its edge.

For this reason, we propose the use of three-valued logic to handle degeneracies. Three-valued logic has already proven to be useful in many areas, e.g. for the analysis of asynchronous circuits [13,4,20], for program analysis [16,2,17,18], for the analysis of cache behaviour [1], and temporal properties of programs [19]. We show that three-valued logic is also a natural foundation to specify geometric problems. Morevover, we do not only use three-valued logic for the analysis of (two-valued) algorithms, but instead propose to implement all geometric primitives directly by three-valued logic. This allows us to describe algorithms in a concise way without having the need to enumerate many tedious degenerate cases. Clearly, these cases do not disappear, but three-valued logic allows us to implicitly treat them in a systematic way. An intuitive view on the third truth value is obtained by the analogy to exception handling in some

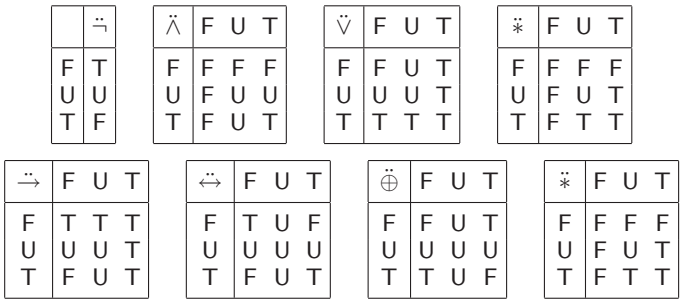

Figure 2. Truth tables of three-valued operators

programming languages: At the point of time where an error occurs, it is not clear how to handle it. Thus, an exception is thrown, which is finally caught by a function that has the necessary context knowledge to handle the problem. Analogously, the degenerate case is passed through all functions until the knowledge of the context is sufficient to resolve the problem.

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 $\ddot{\neg}$, $\ddot{\wedge}$ and $\ddot{\vee}$ shown in Figure 2, which have already been used by Kleene [12]. We introduce further operators (see Figure 2) like implication $\ddot{\rightarrow}$, equivalence $\ddot{\leftrightarrow}$, exclusive-or $\ddot{\oplus}$ and a modified conjunction $\ddot{*}$ (the meaning of which will be explained in Section 2.2).
Moreover, we extend the theory by existential and universal quantification:

$$\texttt{exists3} \vdash_{\text{def}} \ddot{\exists} P = \mathbf{if}\ (\exists x.P(x) = \mathsf{T})\ \mathbf{then}\ \mathsf{T}\ \mathbf{else}\ (\mathbf{if}\ (\forall x.P(x) = \mathsf{F})\ \mathbf{then}\ \mathsf{F}\ \mathbf{else}\ \mathsf{U})$$

$$\texttt{foralll3} \vdash_{\text{def}} \ddot{\forall} P = \mathbf{if}\ (\forall x.P(x) = \mathsf{T})\ \mathbf{then}\ \mathsf{T}\ \mathbf{else}\ (\mathbf{if}\ (\exists x.P(x) = \mathsf{F})\ \mathbf{then}\ \mathsf{F}\ \mathbf{else}\ \mathsf{U})$$

We use a two-valued theorem prover HOL [10] to reason about our geometry primitives. Introducing three-valued formulas into such a two-valued environment, poses the problem to integrate both logics. The conversion of three-valued expressions to 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 $\neg(P = \mathsf{F})$, we introduce two new relations $\leq$ and $\geq$ to improve the readability. By their help, all relevant cases ($P = \mathsf{F}$, $P \leq \mathsf{U}$, $P \geq \mathsf{U}$), $P = \mathsf{T}$) can be described concisely (see Figure 3). The purpose of the operator $\twoheadrightarrow$ will be become clear in the following subsection.

2.2 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:

$$\texttt{les3} \vdash_{\text{def}} r_1 \prec r_2 = \mathbf{if}\ (r_1 < r_2)\ \mathbf{then}\ \mathsf{T}\ \mathbf{else}\ (\mathbf{if}\ (r_2 < r_1)\ \mathbf{then}\ \mathsf{F}\ \mathbf{else}\ \mathsf{U})$$

$\leq$	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

$\twoheadrightarrow$	F	U	T
F	T	T	T
U	T	T	F
T	T	F	T

Figure 3. Truth tables of $\leq$, $\geq$ and $\twoheadrightarrow$

Using this relation, we define geometric objects. 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 exchanged by their three-valued counterparts ($\mathsf{beg}(\ell_1)$ and $\mathsf{end}(\ell_1)$ denote the two points that define the line):

$$\texttt{on\_line\_def} \vdash_{\text{def}} \mathsf{onLine}(\ell, v) = \ddot{\exists}\lambda.\, v = \mathsf{beg}(\ell) + \lambda \cdot (\mathsf{end}(\ell) - \mathsf{beg}(\ell))$$

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.

$$\texttt{on\_seg\_def} \vdash_{\text{def}} \mathsf{onSeg}(\ell, v) = \ddot{\exists}\lambda.\, v = \mathsf{beg}(\ell) + \lambda \cdot (\mathsf{end}(\ell) - \mathsf{beg}(\ell)) \ddot{\wedge} (0 \prec \lambda) \ddot{\wedge} (\lambda \prec 1)$$

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

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

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

Degeneracies with respect to a such a primitive P are inputs $\boldsymbol{x}$ that cause the characteristic function to become zero $\chi_P(\boldsymbol{x}) = 0$. Following the approach presented in Section 2.1, the result U is returned in these cases. To define the primitives, we use the three-valued relation $\prec$ of the previous section. Since all primitives of the previous section compare their result with zero, we additionally introduce a relation pos :

$$\texttt{rat\_pos} \vdash_{\text{def}} \mathsf{pos}\, r = 0 \prec r$$

With its help, the primitive *left* and *below* can be defined as follows:

$$\texttt{left} \vdash_{\text{def}} \mathsf{left}(v, w) = \mathsf{pos}\ (\mathsf{x}_w - \mathsf{x}_v)$$
$$\texttt{below} \vdash_{\text{def}} \mathsf{below}(v, w) = \mathsf{pos}\ (\mathsf{y}_w - \mathsf{y}_v)$$

The primitives make use of the three-valued relation $\prec$, and thus, they have similar properties: The following theorems prove some sort of reflexivity, antisymmetry, and transitivity laws.

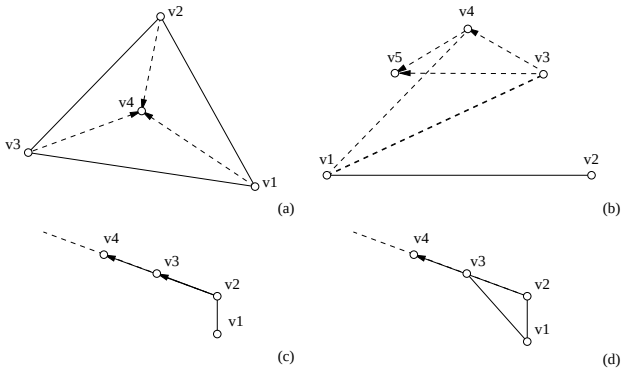

Figure 4. Properties of the left-turn primitive

$\texttt{LEFT\_REF} \vdash \mathsf{left}(v, v) = \mathsf{U}$
$\texttt{LEFT\_ASYM} \vdash \mathsf{left}(v, w) = \ddot{\neg}\mathsf{left}(w, v)$
$\texttt{LEFT\_TRANS} \vdash \mathsf{left}(u, v) \ddot{*} \mathsf{left}(v, w) \twoheadrightarrow \mathsf{left}(u, w)$

LEFT_TRANS makes use of the connectives $\ddot{*}$ and $\twoheadrightarrow$, which usually appear together in a proposition. They allow a succinct description of the following cases:

- If $\mathsf{left}(u, v) = \mathsf{T}$ and $\mathsf{left}(v, w) = \mathsf{T}$, then $\mathsf{left}(u, w) = \mathsf{T}$.
- If $\mathsf{left}(u, v) = \mathsf{T}$ and $\mathsf{left}(v, w) = \mathsf{U}$ or vice versa, then $\mathsf{left}(u, w) = \mathsf{T}$).
- If $\mathsf{left}(u, v) = \mathsf{U}$ and $\mathsf{left}(v, w) = \mathsf{U}$, then $\mathsf{left}(u, w) = \mathsf{U}$.
- If $\mathsf{left}(u, v) = \mathsf{F}$ or $\mathsf{left}(v, w) = \mathsf{F}$, then nothing is said about $\mathsf{left}(u, w)$.

The orientation primitives can be analogously defined:

$\texttt{lturn} \vdash_{\text{def}} \mathsf{lturn}(u, v, w) = \mathsf{pos}\ ((v - u) \times (w - v))$
$\texttt{rturn} \vdash_{\text{def}} \mathsf{rturn}(u, v, w) = \mathsf{lturn}(w, v, u)$

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

$\texttt{LTURN\_REF} \vdash \mathsf{lturn}(v_1, v_1, v_2) = \mathsf{U}$
$\texttt{LTURN\_SYM} \vdash \mathsf{lturn}(v_1, v_2, v_3) = \mathsf{lturn}(v_2, v_3, v_1)$
$\texttt{LTURN\_ASYM} \vdash \mathsf{lturn}(v_1, v_2, v_3) = \ddot{\neg}\mathsf{lturn}(v_2, v_1, v_3)$
$\texttt{LTURN\_TRIAN} \vdash$
$\quad \mathsf{lturn}(v_1, v_2, v_4) \ddot{*} \mathsf{lturn}(v_2, v_3, v_4) \ddot{*} \mathsf{lturn}(v_3, v_1, v_4) \twoheadrightarrow \mathsf{lturn}(v_1, v_2, v_3)$
$\texttt{LTURN\_TRANS} \vdash (\mathsf{lturn}(v_1, v_2, v_3) \ddot{\wedge} \mathsf{lturn}(v_1, v_2, v_4) \ddot{\wedge} \mathsf{lturn}(v_1, v_2, v_5) \geq \mathsf{U})$
$\Rightarrow \mathsf{lturn}(v_1, v_3, v_4) \ddot{*} \mathsf{lturn}(v_1, v_4, v_5) \twoheadrightarrow \mathsf{lturn}(v_1, v_3, v_5)$
$\texttt{LTURN\_MOD1} \vdash (\mathsf{onRay}(\overrightarrow{(v_2, v_3)}, v_4) = \mathsf{T}) \Rightarrow \mathsf{lturn}(v_1, v_2, v_3) = \mathsf{lturn}(v_1, v_2, v_4)$
$\texttt{LTURN\_MOD2} \vdash (\mathsf{onRay}(\overrightarrow{(v_4, v_3)}, v_2) = \mathsf{T}) \Rightarrow \mathsf{lturn}(v_1, v_2, v_4) = \mathsf{lturn}(v_1, v_3, v_4)$

These theorems are three-valued reformulations of the ones that can be found in [15]. 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 with negating the orientation of the sequence. LTURN_TRIAN describes the situation depicted in Figure 4 (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 left-turn primitive 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 (see Figure 4 (b)). The last two theorems (Figure 4 (c) and (d)) are used in [15] 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.

3 Cohen-Sutherland Line Clipping

3.1 Formalisation 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 (see Figure 5 (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*, *below*, *left* and *right* of the window, respectively

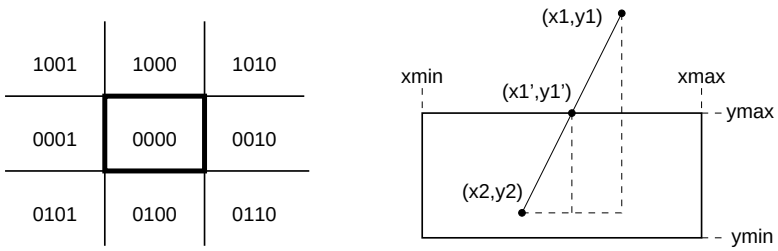

Figure 5. Cohen-Sutherland Algorithm: (a) *Outcodes*, (b) Computing the intersection

With the help of this outcode, 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 set. If a common bit is set in the outcodes of the endpoints, then both endpoints are outside the same side of the window. Thus, the entire line segment is outside the window. In most cases, a line segment will have been either accepted of 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 (see Figure 5 (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 on the top border of the window, we have $y_1' = y_{\max}$. The other coordinate x_1' can be computed as follows:

$$x_1' = x_1 + (x_2 - x_1) \cdot \frac{y_{\max} - y_1}{y_2 - y_1}.$$

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 formalise the algorithm in HOL, we start with the definition of the outcodes. To this end, we use the primitives $\mathsf{lole}(w)$ and $\mathsf{upri}(w)$, which denote the lower left and the upper right corner of the window w:

$$\texttt{outcode} \vdash_{\text{def}} \mathsf{code}(w, v) = (\mathsf{below}(v, \mathsf{lole}(w)) = \mathsf{T},\ \mathsf{above}(v, \mathsf{upri}(w)) = \mathsf{T},\ \mathsf{left}(v, \mathsf{lole}(w)) = \mathsf{T},\ \mathsf{right}(v, \mathsf{upri}(w)) = \mathsf{T})$$

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

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

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

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

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

$$\texttt{ACCEPT} \vdash_{\text{def}} \mathsf{ACCEPT}(c_1, c_2) = \neg(c_1[0] \vee c_2[0] \vee c_1[1] \vee c_2[1] \vee c_1[2] \vee c_2[2] \vee c_1[3] \vee c_2[3])$$

$$\texttt{REJECT} \vdash_{\text{def}} \mathsf{REJECT}(c_1, c_2) = c_1[0] \wedge c_2[0] \vee c_1[1] \wedge c_2[1] \vee c_1[2] \wedge c_2[2] \vee c_1[3] \wedge c_2[3]$$

The actual algorithm is taken from [9], 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 $\mathsf{SOME}(e)$ if the edge e is accepted.

$$\begin{array}{l}
\texttt{csa_clip} \vdash_{\text{def}} \mathsf{csaClip}(w, (v_\text{b}, v_\text{e})) = \\
\quad \textbf{if } \mathsf{ACCEPT}(\mathsf{code}(w, v_\text{b}), \mathsf{code}(w, v_\text{e})) \textbf{ then} \\
\qquad \mathsf{SOME}((v_\text{b}, v_\text{e})) \\
\quad \textbf{else if } \mathsf{REJECT}(\mathsf{code}(w, v_\text{b}), \mathsf{code}(w, v_\text{e})) \textbf{ then} \\
\qquad \mathsf{NONE} \\
\quad \textbf{else if } \mathsf{INSIDE}(\mathsf{code}(w, v_\text{b})) \textbf{ then} \\
\qquad \mathsf{csaClip}(w, (v_\text{e}, v_\text{b})) \\
\quad \textbf{else} \\
\qquad \mathsf{csaClip}(w, \mathsf{shortenedLine}(w, (v_\text{b}, v_\text{e})))
\end{array}$$

$$\begin{array}{l}
\texttt{shortened_line} \vdash_{\text{def}} \mathsf{shortenedLine}(w, (v_\text{b}, v_\text{e})) = \\
\quad \textbf{let} \\
\qquad x_\text{min} = \mathsf{x}_{\mathsf{lole}(w)} \textbf{ and } y_\text{min} = \mathsf{y}_{\mathsf{lole}(w)} \textbf{ and} \\
\qquad x_\text{max} = \mathsf{x}_{\mathsf{upri}(w)} \textbf{ and } y_\text{max} = \mathsf{y}_{\mathsf{upri}(w)} \textbf{ and} \\
\qquad x_1 = \mathsf{x}_{v_\text{b}} \textbf{ and } y_1 = \mathsf{y}_{v_\text{b}} \textbf{ and } x_2 = \mathsf{x}_{vE} \textbf{ and } y_2 = \mathsf{y}_{v_\text{e}} \textbf{ and} \\
\qquad c_1 = \mathsf{code}(w, v_\text{b}) \\
\quad \textbf{in} \\
\qquad \textbf{if } \mathsf{TOP}(c_1) \textbf{ then} \\
\qquad\quad \left(\left(x_1 + (x_2 - x_1) \cdot \dfrac{y_\text{max} - y_1}{y_2 - y_1}\,;\, y_\text{max}\right), v_\text{e}\right) \\
\qquad \textbf{else if } \mathsf{BOTTOM}(c_1) \textbf{ then} \\
\qquad\quad \left(\left(x_1 + (x_2 - x_1) \cdot \dfrac{y_\text{min} - y_1}{y_2 - y_1}\,;\, y_\text{min}\right), v_\text{e}\right) \\
\qquad \textbf{else if } \mathsf{LEFT}(c_1) \textbf{ then} \\
\qquad\quad \left(\left(x_\text{min}\,;\, y_1 + (y_2 - y_1) \cdot \dfrac{x_\text{min} - x_1}{x_2 - x_1}\right), v_\text{e}\right) \\
\qquad \textbf{else} \\
\qquad\quad \left(\left(x_\text{max}\,;\, y_1 + (y_2 - y_1) \cdot \dfrac{x_\text{max} - x_1}{x_2 - x_1}\right), v_\text{e}\right)
\end{array}$$

3.2 Verification

As a 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 (given 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 primitives: $\mathsf{csaInWin}(w_1, v)$ holds if the point v is inside or on the edge of window w_1, and $\mathsf{csaOnSeg}((v_\mathrm{b}, v_\mathrm{e}), v)$ holds if the point v is on the line segment from v_b to v_e (including the endpoints). As before, $\mathsf{lole}(w_1)$ and $\mathsf{upri}(w_1)$ denote the lower left and upper right corner of the clipping window. Note that degenerate inputs and outputs are possible, since the input and the output segments may consist of a single point.

$$\texttt{csa\_in\_win} \vdash_{\mathrm{def}} \mathsf{csaInWin}(w_1, v) = \mathsf{inWin}(\mathsf{lole}(w_1), \mathsf{upri}(w_1), v) \geq \mathsf{U}$$

$$\texttt{csa\_on\_seg} \vdash_{\mathrm{def}} \mathsf{csaOnSeg}((v_\mathrm{b}, v_\mathrm{e}), v) =$$
$$\mathbf{if}\ (v_\mathrm{b} = v_\mathrm{e})\ \mathbf{then}\ (v_\mathrm{b} = v)\ \mathbf{else}\ \mathsf{onSeg}(\overrightarrow{(v_\mathrm{b}, v_\mathrm{e})}, v) \geq \mathsf{U})$$

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.

$$\texttt{CSA\_CORRECTNESS} \vdash$$
$$\mathbf{let}\ r = \mathsf{csaClip}(w_1, (v_\mathrm{b}, v_\mathrm{e}))\ \mathbf{in}$$
$$\mathbf{if}\ (\mathsf{IS\_NONE}(r))\ \mathbf{then}$$
$$\neg(\exists v.\, \mathsf{csaOnSeg}((v_\mathrm{b}, v_\mathrm{e}), v) \wedge \mathsf{csaInWin}(w_1, v))$$
$$\mathbf{else}$$
$$\forall v.\, \mathsf{csaOnSeg}(\mathsf{THE}(r), v) = (\mathsf{csaOnSeg}((v_\mathrm{b}, v_\mathrm{e}), v) \wedge \mathsf{csaInWin}(w_1, v))$$

The correctness of the above specification is proven by induction over the recursive calls of the clipping function $\mathsf{csaClip}(w_1, (v_\mathrm{b}, v_\mathrm{e}))$. There are two base cases: In the first case, the line segment is accepted, since both endpoints are inside the window. We must prove that every point of the segment is also in the window (ACCEPT_SEGMENT_INWINDOW).

$$\texttt{ACCEPT\_SEGMENT\_INWIN} \vdash \mathsf{ACCEPT}(\mathsf{code}(w_1, v_\mathrm{b}), \mathsf{code}(w_1, v_\mathrm{e})) \Rightarrow$$
$$\mathsf{csaOnSeg}((v_\mathrm{b}, v_\mathrm{e}), v) \Rightarrow \mathsf{csaInWin}(w_1, v)$$

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: If the reject primitive holds, we prove that the window that is defined by the endpoints of the line segment and the clipping window have no common area. Thus, the segment does not have an intersection with the clipping window, and it is correctly rejected.

$$\texttt{bbox\_check} \vdash_{\mathrm{def}} \mathsf{bboxCheck}(v_1, v_2, v_3, v_4) =$$
$$(\mathsf{vecXint}(v_1, v_3, v_2)\ \ddot{\vee}\ \mathsf{vecXint}(v_1, v_4, v_2)\ \ddot{\vee}$$
$$\mathsf{vecXint}(v_3, v_1, v_4)\ \ddot{\vee}\ \mathsf{vecXint}(v_3, v_2, v_4) \geq \mathsf{U}) \wedge$$
$$(\mathsf{vecYint}(v_1, v_3, v_2)\ \ddot{\vee}\ \mathsf{vecYint}(v_1, v_4, v_2)\ \ddot{\vee}$$
$$\mathsf{vecYint}(v_3, v_1, v_4)\ \ddot{\vee}\ \mathsf{vecYint}(v_3, v_2, v_4) \geq \mathsf{U})$$

$$\texttt{REJECT\_BBOX\_CHECK} \vdash$$
$$\mathsf{REJECT}(\mathsf{code}(w_1, v_\mathrm{b}), \mathsf{code}(w_1, v_\mathrm{e})) \Rightarrow \neg\mathsf{bboxCheck}(\mathsf{lole}(w_1), \mathsf{upri}(w_1), v_\mathrm{b}, v_\mathrm{e})$$

$$\texttt{CSA\_BBOX\_CHECK} \vdash$$
$$\neg(\mathsf{bboxCheck}(v_1, v_2, v_3, v_4) \wedge (\mathsf{inWin}(v_3, v_4, v) \geq \mathsf{U}) \Rightarrow \neg(\mathsf{inWin}(v_1, v_2, v) \geq \mathsf{U})$$

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).

$$\texttt{CSA\_ONSEG\_SYM} \vdash \mathsf{csaOnSeg}((v_\mathrm{b}, v_\mathrm{e}), v) = \mathsf{csaOnSeg}((v_\mathrm{e}, v_\mathrm{b}), v)$$

The second recursive call is the hardest part of the proof. The line segment is shortened. Two things must be proven: 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).

$$\begin{aligned}
&\texttt{CSA_CUT_CORRECT} \vdash \\
&\quad \neg\mathsf{ACCEPT}(\mathsf{code}(w_1, v_\mathrm{b}), \mathsf{code}(w_1, v_\mathrm{e})) \wedge \\
&\quad \neg\mathsf{REJECT}(\mathsf{code}(w_1, v_\mathrm{b}), \mathsf{code}(w_1, v_\mathrm{e})) \wedge \\
&\quad \neg\mathsf{INSIDE}(\mathsf{code}(w_1, v_\mathrm{b})) \wedge \\
&\quad \mathsf{csaInWin}(w_1, v) \Rightarrow \\
&\quad \mathsf{csaOnSeg}((\mathsf{shortenedLine}(w_1, (v_\mathrm{b}, v_\mathrm{e}))), v) \Rightarrow \mathsf{csaOnSeg}((v_\mathrm{b}, v_\mathrm{e}), v)
\end{aligned}$$

$$\begin{aligned}
&\texttt{CSA_CUT_COMPLETE} \vdash \\
&\quad \neg\mathsf{ACCEPT}(\mathsf{code}(w_1, v_\mathrm{b}), \mathsf{code}(w_1, v_\mathrm{e})) \wedge \\
&\quad \neg\mathsf{REJECT}(\mathsf{code}(w_1, v_\mathrm{b}), \mathsf{code}(w_1, v_\mathrm{e})) \wedge \\
&\quad \neg\mathsf{INSIDE}(\mathsf{code}(w_1, v_\mathrm{b})) \wedge \\
&\quad \mathsf{csaInWin}(w_1, v) \Rightarrow \\
&\quad \mathsf{csaOnSeg}((v_\mathrm{b}, v_\mathrm{e}), v) \Rightarrow \mathsf{csaOnSeg}(\mathsf{shortenedLine}(w_1, (v_\mathrm{b}, v_\mathrm{e}), v)
\end{aligned}$$

$$\begin{aligned}
&\texttt{LINE_SPLIT} \vdash (\mathsf{onSeg}(\ell_1, v_m) = \mathsf{T}) \Rightarrow \\
&\quad ((\mathsf{onSeg}(\ell_1, v) \geq \mathsf{U}) = (v = \mathsf{beg}(\ell_1)) \vee (\mathsf{onSeg}(\overrightarrow{(\mathsf{beg}(\ell_1), v_m)}, v) = \mathsf{T}) \vee \\
&\qquad\qquad\qquad\qquad\qquad\qquad (\mathsf{onSeg}(\overrightarrow{(v_m, \mathsf{end}(\ell_1))}, v) \geq \mathsf{U}))
\end{aligned}$$

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.

4 Implementation of a Dependable Software Library

The previous sections illustrated how rational numbers and a three-valued logic can be used to specify and verify polygon processing algorithms. We use the same techniques to implement a software library. The advantages are obvious: 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 and more important, algorithms are more compact, because several cases of the two-valued formalisation 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.

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 $\ddot{\exists}x.\,\mathsf{onSeg}(\ell_1, x) \ddot{\wedge} \mathsf{onSeg}(\ell_2, x)$.

As another example, reconsider the *winding number* algorithm, which is described in Section 2.1. A classic implementation of this algorithm that directly deals with degenerate cases cannot avoid the case distinctions shown in Figure 1. 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 easier. Three-valued primitives eliminate all extrinsic degeneracies. This clearly demonstrates the benefits of our approach.

Provided that `e[i]` denotes the i-th of n edges of a polygon and `xRay(v)` is the ray from v to the right, the following C fragment calculates the winding number w of 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 propose the use of three-valued logic to develop dependable algorithms for geometric applications to be used in safety-critical 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.

The obtained software library has been formalised in the interactive theorem prover HOL. Moreover, we used HOL to formally reason about the geometric primitives 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 primitives, 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.

References

1. M. Alt, C. Ferdinand, F. Martin, and R. Wilhelm. Cache behavior prediction by abstract interpretation. In *Static Analysis Symposium (SAS)*, volume 1145 of *LNCS*, pages 52–66, Aachen, Germany, 1996. Springer.
2. G. Berry. The constructive semantics of pure Esterel. http://www-sop.inria.fr/esterel.org, July 1999.
3. J. Brandt and K. Schneider. Using three-valued logic to specify and verify algorithms of computational geometry. In *International Conference on Formal Engineering Methods (ICFEM)*, LNCS, Manchester, UK, 2005. Springer.
4. J.A. Brzozowski and C.-J.H. Seger. *Asynchronous Circuits*. Springer, 1995.
5. C. Burnikel, K. Mehlhorn, and S. Schirra. On degeneracy in geometric computations. In *Symposium on Discrete Algorithms (SODA)*, pages 16–23, Arlington, Virginia, USA, 1994. ACM.
6. S.C. Chou, X.S. Gao, and J.Z. Zhang. *Machine Proofs in Geometry*. World Scientific, Singapore, 1994.
7. M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer, 2000.
8. H. Edelsbrunner and E.P. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.
9. J.D. Foley, A. van Dam, S.K. Feiner, and J.F. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, 2000.
10. M.J.C. Gordon and T.F. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.
11. K. Hormann and A. Agathos. The point in polygon problem for arbitrary polygons. *Computational Geometry*, 20(3):131–144, 2001.
12. S.C. Kleene. *Introduction to Metamathematics*. North Holland, 1952.
13. S. Malik. Analysis of cyclic combinational circuits. In *Conference on Computer Aided Design (ICCAD)*, pages 618–625, Santa Clara, California, November 1993. IEEE Computer Society.
14. K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.
15. D. Pichardie and Y. Bertot. Formalizing convex hull algorithms. In R.J. Boulton and P.B. Jackson, editors, *Higher Order Logic Theorem Proving and its Applications (TPHOL)*, volume 2152 of *LNCS*, pages 346–361, Edinburgh, Scotland, UK, 2001. Springer.
16. T.W. Reps, M. Sagiv, and R. Wilhelm. Static program analysis via 3-valued logic. In R. Alur and D.A. Peled, editors, *Conference on Computer Aided Verification (CAV)*, volume 3114 of *LNCS*, pages 15–30, Boston, MA, USA, 2004. Springer.
17. 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.
18. K. Schneider, J. Brandt, T. Schuele, and T. Tuerk. Maximal causality analysis. In *Conference on Application of Concurrency to System Design (ACSD)*, pages 106–115, St. Malo, France, June 2005. IEEE Computer Society.
19. T. Schuele and K. Schneider. Three-valued logic in bounded model checking. In *Formal Methods and Models for Codesign (MEMOCODE)*, Verona, Italy, 2005. IEEE Computer Society.
20. T.R. Shiple. *Formal Analysis of Synchronous Circuits*. PhD thesis, University of California at Berkeley, 1996.
21. W.-T. Wu. On the decision problem and the mechanization of theorem proving in elementary geometry. *Scientia Sinica*, 21:157–179, 1978.