

# Efficient Map Overlay for Safety-Critical Embedded Systems

Jens Brandt and Klaus Schneider  
Reactive Systems Group  
Department of Computer Science  
University of Kaiserslautern  
P.O. Box 3049, 67653 Kaiserslautern, Germany  
<http://rsg.informatik.uni-kl.de>

## Abstract

*Computational geometry algorithms are more and more frequently used in industrial embedded systems. These new applications require not only highly efficient implementations that run with limited computational resources, but also algorithms that are capable to robustly handle all geometric situations in a consistent way. In this paper, we present the adaptation and implementation of a map overlay algorithm, which originates from the area of geographical information systems. Besides the adaptation to an embedded platform, mainly two problems have to be tackled: First, safe results must be computed with limited precision arithmetic and second, map data must be adequately simplified to meet on the one hand real-time requirements, and on the other hand robustness criteria.*

## 1 Introduction

Modern industrial embedded systems in the automotive industry integrate a steadily growing functional range. In the past years, all these systems were restricted to only monitor components of the own car as done e.g. by fuel injection and anti-lock braking systems. Current and future embedded systems, however, are also aware of the environment of the car. Radar and optical sensors are already available in many cars to observe the objects in the environment of the car. These systems can now be used to analyse the situation, to assist the driver and even to actively take over some of his tasks, similar to the adaptive cruise control (ACC), which has now been well-established for a long time.

Whenever the spatial behaviour of vehicles has to be taken into account, physical objects and situations are naturally modelled as geometric objects and properties. In many cases, it is thereby sufficient to model

the environment as a two-dimensional plane, and the objects can be approximated by polygons on that plane. A lot of algorithms for dynamic motion planning or collision detection [14] follow this approach.

For the manipulation of these objects, well-known geometry algorithms [5] are used. In particular, algorithms for manipulating polygons, i.e. to triangulate a polygon, to determine its convex hull, and to compute the intersection, union or difference of a given set of polygons are fundamental algorithms to solve difficult motion planning and collision detection problems.

We aim at implementing a software library that implements this class of algorithms. In contrast to existing software libraries like CGAL<sup>1</sup>, our software library must meet the following additional requirements, which come from the embedded systems area:

- *Verification.* Since most applications are safety-critical, great care has to be taken when designing the systems. In particular, degenerate cases [6] impose many tricky and unforeseen problems that challenge the robust design of these systems. For this reason, formal verification is routinely performed as a complementary technique to testing and simulation in order to ensure the correctness of the system. As presented in [3, 2], we use three-valued logics [1, 12] to specify geometric primitives and to build a verified kernel library. To this end, we employ the interactive theorem prover HOL4 [9] for checking the formalised algorithms with respect to their specifications.
- *Conservative Computations.* In general, the limited precision of the arithmetic operations of microprocessors leads to a loss of information. The use of arbitrarily precise arithmetics or interval arithmetic [16, 17, 10] yields algorithms that are too

---

<sup>1</sup><http://www.cgal.org>

complex for small embedded devices. Relying on the usual rounding methods as provided by the IEEE 754 standard for floating point numbers, however, may lead to wrong results [4] in geometric computations. Hence, 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.

- *Limited Resources.* Due to economic constraints, only very limited resources are usually available on embedded systems. On the other hand, the response time of a system must often fulfil given real-time constraints. Hence, an application-specific memory management is required for embedded systems. In particular, this component includes mechanisms to limit the size of the current model and the time needed for computations.

In the following, we will focus on an important polygon processing algorithm, namely the map overlay algorithm [5]. It solves a fundamental geometric problem that 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 labelled maps.

Although there are well-known algorithms to solve the map overlay problem, we can not directly use these algorithms due to the particular needs for embedded systems that we mentioned above. In this paper, we therefore present modifications of well-known algorithms that have to be made to use the algorithm in an industrial embedded system.

The paper is structured as follows: In the following section, we define the general map overlay problem, and present its solution. In Section 3, we present our approach to limit the precision of the arithmetic computations, and in Section 4, we discuss the simplification of computed maps. Finally, we list some experimental results in Section 5 and draw some conclusions.

## 2 Maps and Map Overlay

### 2.1 Map

As the name suggests, the map overlay problem has its origin in the domain of geographic information systems. The process combines a set of separate spatial datasets (points, lines or polygons) together with

their labels, which describe geographic features and attributes, into a new single one. In order to explain the overlay, we first define maps and sketch their representation in our algorithms.

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

- $\mathcal{V}$  is the set of vertices. Each vertex  $v \in \mathcal{V}$  has a unique position that is described by its rational<sup>2</sup> Cartesian coordinates  $(x_v, y_v) \in \mathbb{Q}^2$ .
- $\mathcal{E}$  is the set of edges. An edge  $e$  connects two different vertices  $v_i, v_j \in \mathcal{V}$ . A point can be common to more than one edge only if it is the position of a vertex. Thus, since their edges do not intersect, a map is planar.
- The edges of a map imply a partitioning of the plane into a set of faces  $\mathcal{F}$ . 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  $\text{unbdFace}(M)$ , which has only inner components.

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

Figure 1 gives an example map and illustrates the data structure. We represent the maps as doubly connected edge lists: In this data structure, edges consist of pairs of half-edges (solid lines) representing both sides 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 a reference (dashed lines) to the two vertices  $\text{src}(e)$  and  $\text{dest}(e)$  that it connects. The face that lies on its left-hand side (as seen from source to destination) is its  $\text{leftFace}(e)$ , the opposite one is its  $\text{rightFace}(e)$ . A face  $f$  has references to its outer component  $\text{outerComp}(f)$  and the set of its inner components  $\text{innerComp}(f)$ . They point to an arbitrary edge of the component, from where the whole component can be traversed. Finally, each map  $M$  has a reference to its unbounded face  $\text{unbdFace}(M)$ .

### 2.2 Map Overlay

When information from different objects are combined, the map overlay comes into play. This operation merges a set of inputs maps (sometimes called layers) into a single new one. More formally:

<sup>2</sup>Since we restrict on straight edges and polygonal regions, the positions of the vertices are given by rational numbers.

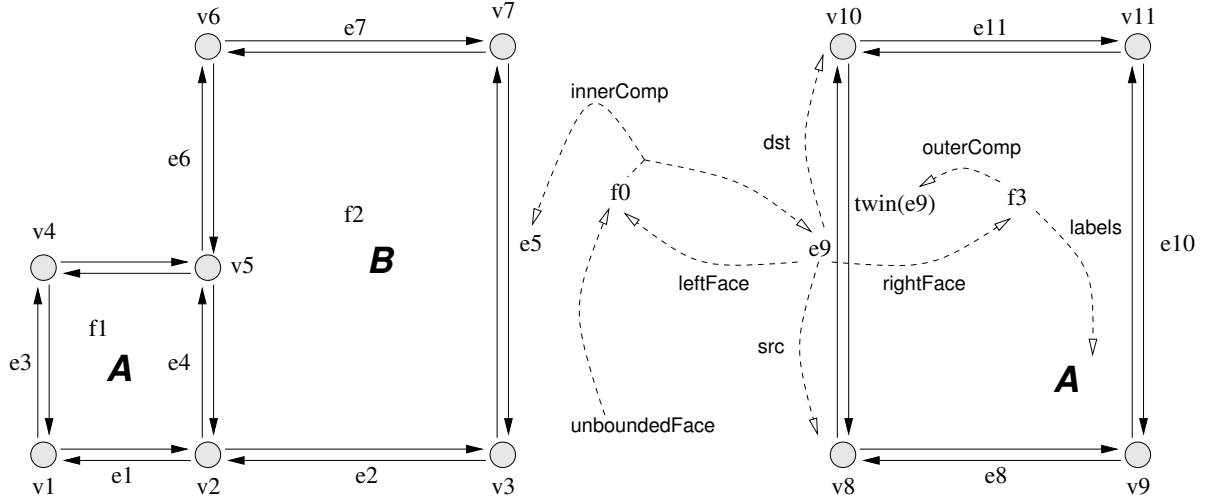


Figure 1. Example Map and Map Data Structure

```

function OverlayIntersection( $M, e_1, e_2$ )
   $v :=$  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 2. Planarisation of Maps

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

- $\mathcal{V}$  contains all input vertices  $\bigcup_{i=1 \dots n} \mathcal{V}_i$  as well as all their intersections.
- The set of edges  $\mathcal{E}$  is constructed by taking all input edges  $\bigcup_{i=1 \dots n} \mathcal{E}_i$  and then iteratively replacing all pairs of edges  $(e_1, e_2)$  that intersect with  $\text{OverlayIntersection}(e_1, e_2)$  and all pairs that have a common segment with  $\text{OverlayCommonSegment}(e_1, e_2)$ , respectively<sup>3</sup>.
- $\mathcal{F}$  is again induced by the set of edges  $\mathcal{E}$  of the new map.

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

<sup>3</sup>We use lexical ordering for map vertices:  
 $v_1 \prec v_2 \Leftrightarrow (x_{v_1} < x_{v_2} \vee x_{v_1} = x_{v_2} \wedge y_{v_1} < y_{v_2})$ .

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

As we are only interested in the labelled 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 [5], we developed an algorithm that can handle all maps, i.e. degenerate inputs do not pose a problem. In contrast to previous publications, we show our solution in great 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 [5] 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 labelling. Since we are not interested in redundant vertices and edges (with respect to the labelling), 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 (see Figure 3) is based on the plane sweep approach [5], i.e. the result is computed incrementally while moving an imaginary vertical line

```

function MapOverlay( $M_1, \dots, M_n$ )
   $M := \text{newMap}()$ 
   $\text{curRaw}(\perp) := \sum_{i=1}^n \text{labels}(\text{unbdFace}(M_i))$ 
   $\text{curLabels}(\perp) := \text{computeLabels}(\text{curRaw}(\perp))$ 
   $\text{labels}(\text{unbdFace}(M)) := \text{curLabels}(\perp)$ 
   $\text{InitialiseEventQueue}(M_1, \dots, M_n)$ 
   $S := \langle \rangle$ 
  while ( $Q \neq \langle \rangle$ )
    ( $\text{lastVertex}, \mathcal{E}_{\text{begin}}, \mathcal{E}_{\text{end}} := \text{head}(Q)$ )
     $Q := \text{tail}(Q)$ 
     $\text{vertices}(M) := \text{vertices}(M) \cup \{\text{lastVertex}\}$ 
    ( $s_0, s_1, s_2, s_3 := \text{RegionEnd}(\text{lastVertex})$ )
     $f_{\text{above}} := \text{curFace}(\text{prev}(s_3))$ 
     $f_{\text{below}} := \text{curFace}(s_0)$ 
     $\text{leftEdges} := 0$ 
     $\text{rightEdges} := 0$ 
    if ( $s_1 \neq \perp$ )
       $\text{HandleEdgesEnd}()$ 
     $S := \text{StatusRemoveSet}(S, \mathcal{E}_{\text{end}})$ 
     $S := \text{StatusInsertSet}(S, \mathcal{E}_{\text{begin}})$ 
    ( $s_0, s_1, s_2, s_3 := \text{RegionBegin}(\text{lastVertex})$ )
     $\text{CheckIntersections}(Q, S)$ 
    if ( $s_1 \neq \perp$ )
       $\text{HandleEdgesBegin}()$ 
     $\text{HandleMerge}()$ 
    if ( $\text{leftEdges} = 0$  and  $\text{rightEdges} = 0$ )
       $\text{vertices}(M) := \text{vertices}(M) \setminus \{\text{lastVertex}\}$ 
  return  $M$ 

```

**Figure 3. Overlay Algorithm**

over the inputs from left to the 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, respectively.

For the map overlay algorithm, event points (see Figure 4) are the vertices of the result map. Each triple in the queue  $Q$  represents a vertex  $v$  together with the edges that start (lexically smaller endpoint is  $v$ ) and the edges that end (lexically greater endpoint is  $v$ ) there. Initially,  $\text{InitialiseEventQueue}$  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 neighbour edges intersect ( $\text{CheckIntersections}$ ). 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

```

function EventQueueInsert( $Q, v, \mathcal{E}_1, \mathcal{E}_2$ )
  ( $v', \mathcal{E}'_1, \mathcal{E}'_2 := \text{head}(Q)$ )
  switch
  case  $v' \prec v$  :
    return  $\text{head}(Q) :: \text{EventQueueInsert}(\text{tail}(Q), v, \mathcal{E}_1, \mathcal{E}_2)$ 
  case  $v' = v$  :
    return ( $v, \mathcal{E}_1 \cup \mathcal{E}'_1, \mathcal{E}_2 \cup \mathcal{E}'_2$ ) ::  $\text{tail}(Q)$ 
  case  $v' \succ v$  :
    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 ( $\text{src}(e) \prec \text{dest}(e)$ )
         $\text{EventQueueInsert}(\text{src}(e), \{e\}, \{\})$ 
         $\text{EventQueueInsert}(\text{dest}(e), \{\}, \{e\})$ 
      else
         $\text{EventQueueInsert}(\text{dest}(e), \{e\}, \{\})$ 
         $\text{EventQueueInsert}(\text{src}(e), \{\}, \{e\})$ 

function CheckIntersection( $Q, s_1, s_2$ )
   $v := \text{EdgeIntersection}(\text{head}(s_1), \text{head}(s_2))$ 
  if ( $v \neq \perp$ )
    forall ( $e \in s_1, s_2$ )
      if ( $\text{onEdge}(v, e)$ )
         $\text{EventQueueInsert}(Q, v, \{e\}, \{e\})$ 

function CheckIntersections( $Q, S$ )
  if ( $s_1 \neq \perp$ )
    if ( $s_0 \neq \perp$ )
       $\text{CheckIntersection}(Q, s_0, s_1)$ 
    if ( $s_3 \neq \perp$ )
       $\text{CheckIntersection}(Q, s_2, s_3)$ 
  else
    if ( $s_0 \neq \perp$  and  $s_3 \neq \perp$ )
       $\text{CheckIntersection}(Q, s_0, s_3)$ 

```

**Figure 4. Overlay Algorithm: Event Queue**

run through the vertex.

The status structure (see Figure 5)  $S$  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 end vertices. (This makes the detection of intersection points in  $\text{CheckIntersection}$  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  $\text{StatusCompare}$  implements all these rules.

```

function StatusCompare( $v, e_1, e_2$ )
   $y_1 := \text{yValueAt}(e_1, x_v)$ 
   $y_2 := \text{yValueAt}(e_2, x_v)$ 
  if ( $y_1 = \perp$  and  $y_2 = \perp$ ) return 0
  if ( $y_1 = \perp$  and  $y_2 \neq \perp$ )
    if ( $y_2 < y_v$ ) return -1 else return +1
  if ( $y_1 \neq \perp$  and  $y_2 = \perp$ )
    if ( $y_2 < y_v$ ) return +1 else return -1
  if ( $y_1 \neq y_2$ )
    if ( $y_1 < y_2$ ) return -1 else return +1
   $s_1 := \text{Slope}(e_1)$ 
   $s_2 := \text{Slope}(e_2)$ 
  if ( $y_1 < y_v$ )
    if ( $s_2 < s_1$ ) return -1 else return +1
  else
    if ( $s_1 < s_2$ ) return -1 else return +1
  return 0

function StatusInsertCollinear( $s, e$ )
  if ( $\text{src}(e) \prec \text{head}(s)$  and  $\text{dest}(e) \prec \text{head}(s)$ )
    return  $\text{head}(S) :: \text{StatusInsertCollinear}(\text{tail}(S), e)$ 
  else
    return  $s :: \text{tail}(S)$ 

function StatusInsert( $S, e$ )
  switch ( $\text{StatusCompare}(\text{head}(S), e)$ )
  case -1 :
    return  $\text{head}(S) :: \text{StatusInsert}(\text{tail}(S), e)$ 
  case 0 :
    return  $\text{StatusInsertCollinear}(s, e) :: \text{tail}(S)$ 
  case +1 :
    return  $\langle e \rangle :: S$ 

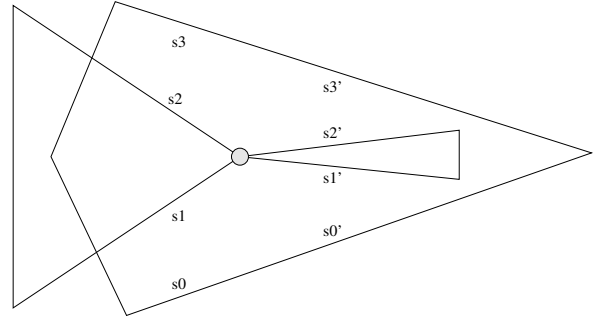
function StatusInsertSet( $S, \mathcal{E}$ )
  forall ( $e \in \mathcal{E}$ )
     $S := \text{StatusInsert}(S, e)$ 
  return  $S$ 

```

**Figure 5. Overlay Algorithm: Status Structure**

The functions `RegionEnd` (and `RegionBegin`, resp.) determine 4 elements of the status structure:  $s_0$  is the list of edges below the event point,  $s_3$  the list above it.  $s_1$  is the first list of edges that contains the event point on its left side (right side), and  $s_2$  is the last list. If no corresponding list can be found,  $\perp$  will be returned. In Figure 6, the value of these functions are shown for the marked event point: the ordinary ones show the result of `RegionEnd`, the primed ones the result of `RegionBegin`.

In our approach, the creation of the map and the labelling of faces is handled completely differently to [5] (see Figure 7): A vertex and the edges that connect it to the part of the result that has been already computed (on the left) are created (`HandleEdgesEnd`) at



**Figure 6. RegionEnd and RegionBegin functions**

each event point. `curFace(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 set the face references `leftFace` and `rightFace`.

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. `curRaw` stores the sum of all labels, whereas `curLabels` gives the actual labelling that is defined by the desired operation. E.g. if the union of maps should be calculated, at least one map must contribute a label to the corresponding `curRaw` entry. For the intersection, all maps must contribute to add this label to the actual label set `curLabels`.

One event must be handled separately (`HandleMerge`): At the current event point, there may be no edges ending there. At these merge points, two cases must be distinguished: The face above and the face below the current event point must 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  $f_{\text{above}}$  and all updating all references to it. This can be done very efficiently if indirect references are used for the face pointers. In the other case, the current event point marks a right end of an inner component, to which the face can be linked at this point.

### 3 Conservative Rounding

Limited precision arithmetic generally poses problems to computational geometry algorithms. In particular, the naive use of floating point arithmetic is dangerous and leads to unpredictable and bizarre results as already experienced by many researchers of this area[15]: E.g. because of rounding errors, the

```

function HandleEdgesEnd()
   $s := s_1$ 
  if (curFace(prev(s)) = curFace(s))
    lastEdge := newEdge()
    edges(M) := edges(M)  $\cup$  {e}
    src(e) := curStart(s)
    dest(e) := v
    leftFace(e) := curFace(s)
    rightFace(e) := curFace(prev(s))
    leftEdges := leftEdges + 1
  for (s := next(s1) ... s2)
    if (curFace(prev(s)) = curFace(s))
      lastEdge := newEdge()
      edges(M) := edges(M)  $\cup$  {e}
      src(e) := curStart(s)
      dest(e) := v
      leftFace(e) := curFace(s)
      rightFace(e) := curFace(prev(s))
      leftEdges := leftEdges + 1
      f := faceBelow(s)
      outerComp(f) = lastEdge

function HandleEdgesBegin()
  curFace(s0) := fbelow
  curFace(s2) := fabove
   $s := s_1$ 
  slast := s1
  lastFace := fbelow
  curStart(s) := lastVertex
  curRaw(s) := curRaw(prev(s)) + LabelDiff(s)
  labels(s) := computeLabels(rawLabels(s))
  if (curLabels(s)  $\neq$  curLabels(prev(s)))
    rightEdges := rightEdges + 1
  for (s = next(s1) ... s2)
    curStart(s) := lastVertex
    curRaw(s) := curRaw(prev(s)) + LabelDiff(s)
    curLabels(s) := computeLabels(curRaw(s))
    if (curLabels(s)  $\neq$  curLabels(prev(s)))
      rightEdges := rightEdges + 1
      if (labels(flast)  $\neq$  curLabels(prev(s)))
        lastFace := newFace()
        faces(M) := faces(M)  $\cup$  {lastFace}
        labels(f) := curLabels(prev(s))
        for (t = slast ... prev(s))
          curFace(t) = lastFace
        slast = s
    for (t = slast ... s2)
      curFace(t) = lastFace

```

**Figure 7. Overlay Algorithm: Edge Handling**

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

```

function HandleMerge()
  if (leftEdges > 0 and rightEdges = 0)
    if (fbelow = fabove)
      innerComp(fbelow) :=
        innerComp(fbelow)  $\cup$  {lastEdge}
    else
      innerComp(fbelow) :=
        innerComp(fbelow)  $\cup$  innerComp(fabove)
      faces(M) := faces(M)  $\setminus$  {fabove}
      /* relink all fabove pointers to fbelow */

```

**Figure 8. Overlay Algorithm: Face Merging and Isolated Vertices**

(round towards zero, round to nearest, etc.) that are fatal for surrounding algorithm. In our application for example, the face labels mark impassable or dangerous areas for various vehicles. If a path were chosen that leads through the original but not through the rounded area, the rounding would cause a crash. So, e.g. traditional snap-rounding algorithms [11] relying on simple rounding methods are not applicable.

Since especially the correct calculation of 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 their value, so-called floating point filters have been proposed to solve this problem [8, 17]. However, these approaches are generally not able to handle geometric objects that have been created by the algorithm, e.g. intersection points. Although they can be described by a set of input objects, these approaches have no practical relevance as a result of the iterative use of the overlay algorithm as required for dynamic motion planning and collision detection systems.

So, without modifications to the algorithm, arbitrary-precision arithmetic would be needed for all computations. But this is far too complex for non-trivial examples: Consider the sequence of overlay operations  $O_1 \dots O_n$ , where the results 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.

This means that the complexity of the results grows exponentially (about  $n \cdot 4^i$  are needed in step  $i$ ). Even for ordinary desktop systems, the exponential blowup

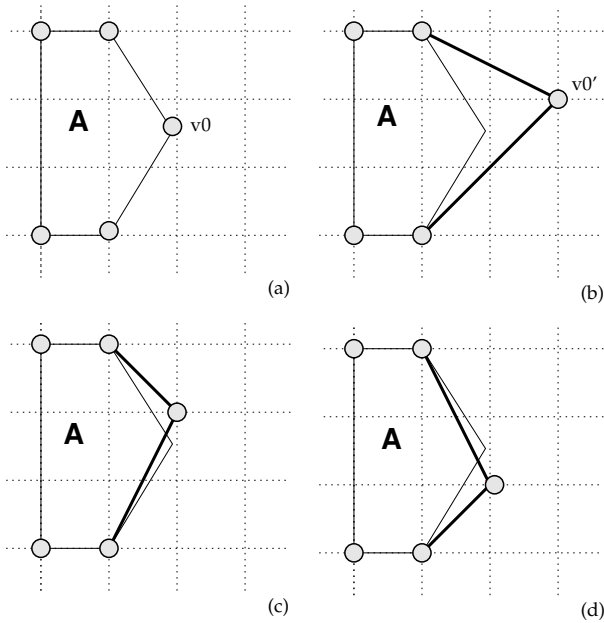


Figure 9. Rounding a vertex

slows down the computation significantly and fills the memory after some steps as we experienced with an implementation based on the GNU Multi Precision library<sup>4</sup>.

The apparent 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 the all geometric predicates of a step can be computed correctly for this precision, a solution for the problem is found. This process can be visualised 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 labelled areas of the rounded map cover at least the original area, the algorithm cannot simply choose the closest grid point. It even may be the case that none of the surrounding grid points can be chosen (see Figure 9), since for all of them a part of the original area is cut off. There, some vertex further to the right must be chosen (b).

**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.

<sup>4</sup><http://www.swox.com/gmp/>

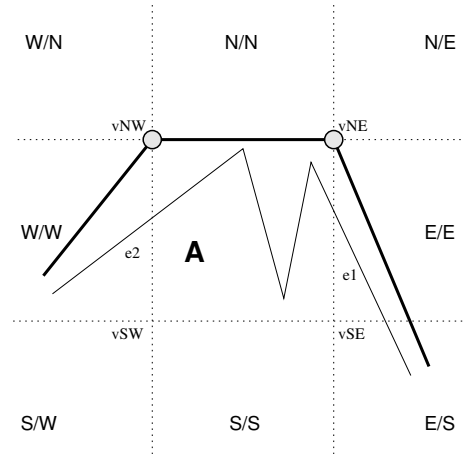


Figure 10. Entry/Exit Codes and Example

- For each label  $\ell$ , all faces of  $\mathcal{F}'$  that are labelled with  $\ell$  cover at least the area that has been labelled by the faces of  $\mathcal{F}$ .

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

To solve this problem, we propose to approximate a vertex that is not on the grid basically by several vertices that lie on the grid points around the original vertex. Despite this simple idea, the algorithm is a bit more complicated, since all inputs must be handled.<sup>5</sup>

The algorithm (see Figure 11) 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 [7, 18] is computed. Based on this outcode, we determine an entry code and an exit code for both of the two adjoining vertices. Depending on them, 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 counter-clockwise direction. Figure 10 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 that must be treated

<sup>5</sup>In the following, we will restrict to maps that have only one label. Multi-coloured maps can be handled as well by rounding the vertices for each colour separately.

```

function RoundVertices( $v_{SW}, v_{SE}, v_{NE}, v_{NW}, e_1, e_2$ )
   $v_1 := \text{src}(e_1)$ 
   $v_2 := \text{dest}(e_1)$ 
   $v_3 := \text{src}(e_2)$ 
   $v_4 := \text{dest}(e_2)$ 
   $(c_0, c_1) := \text{EntryExitCode}(v_{SW}, v_{SE}, v_{NE}, v_{NW}, v_1, v_4)$ 
  if ( $c_0 = c_1$  and  $\text{RightTurn}(v_1, v_2, v_3) \vee$ 
     $\text{Collinear}(v_1, v_2, v_3) \wedge \text{RightTurn}(v_1, v_3, v_4)$ )
    return  $\langle v_1, v_4 \rangle$ 
  else
    switch ( $c_0$ )
    case N :
      if ( $c_1 = W$ )
        return  $\langle v_1, v_{NW}, v_4 \rangle$ 
      else if ( $c_1 = S$ )
        return  $\langle v_1, v_{NW}, v_{SW}, v_4 \rangle$ 
      else if ( $c_1 = E$ )
        return  $\langle v_1, v_{NW}, v_{SW}, v_{SE}, v_4 \rangle$ 
      else
        return  $\langle v_1, v_{NW}, v_{SW}, v_{SE}, v_{NE}, v_4 \rangle$ 
    case E : ...
    case S : ... /* other cases analogously */
    case W : ...

```

Figure 11. Vertex Rounding Algorithm

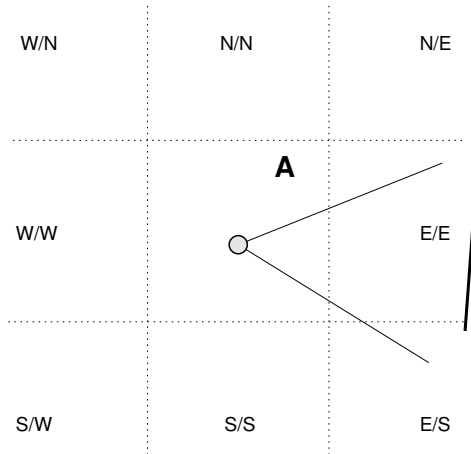


Figure 12. Entry/Exit Codes and Example

separately. The first one occurs when there are no preceding and following vertices, i.e. the contour is completely inside a grid box. Then, this grid box is the approximation. In the second one, the preceding and following vertices have the same code, and the part describes a concave part of the face (see Figure 12). Then, no local approximation is possible. A very simple solution is to just remove all intermediate vertices and to insert an edge that connects the preceding to the following vertex directly. 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 chose 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 border of the grid box, the vertex  $v$  can be approximated by  $v = (x', y')$  with

$$\begin{aligned} x' &= \lceil x_{NE} + \frac{x_{NE} - x_{SW}}{s_{in} - s_{out}} \rceil \\ y' &= \lceil \text{yValueAt}(e_{in}, x') \rceil \end{aligned} \quad (1)$$

With the exception of the last optimisation, the algorithm does not rely on a specific grid. In particular, the grid does not need to be equidistant. So, as a basic type not only integers are possible but rationals and floating point numbers as well. So, we can use e.g. 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 great detail by Knuth [13].

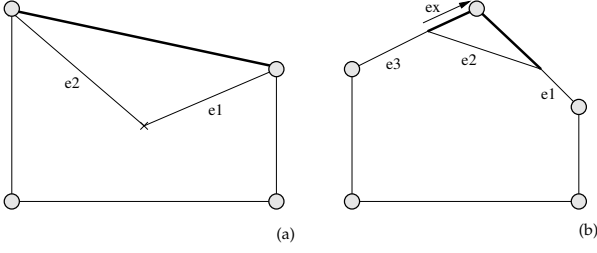
Unfortunately, vertices cannot be simply moved or replaced in a map as done in the algorithm above, since the planarity property of the map may be lost by overlapping faces. Therefore, after the rounding step, an overlay of the rounded map must be computed. That may create vertices that are not on the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

## 4 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 simplification operation aims at the elimination of vertices, edges or faces to reduce the overall computation complexity. Similar to the rounding of vertices, the abstraction of the maps must be conservative, i.e. labelled areas must not get smaller. And again, the abstracted map should resemble the old map while using less vertices and edges.

Generally, any bounded labelled area of a map can be approximated with three vertices forming a triangle that covers the coloured area. A very simple ap-



**Figure 13. Basic abstraction operations**

proximation requiring four vertices is the bounding box, a rectangle aligned to the coordinate system that totally covers the labelled area. And if the map has a lot of jagged areas, the convex hull of them is good abstraction. All these approximations can be on one side very efficiently calculated but on the other side, they are too imprecise for practical usage.

Therefore, we use again a local approach to abstract maps iteratively. It is based on the following two operations (see Figure 13):

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

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

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 coloured 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 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  $\lambda, \mu$  and  $e_x$ :

$$\lambda = \frac{\vec{e}_2 \times \vec{e}_3}{\vec{e}_1 \times \vec{e}_3} \quad \mu = -\frac{\vec{e}_1 \times \vec{e}_2}{\vec{e}_1 \times \vec{e}_3} \quad e_x = \frac{\vec{e}_1 \times \vec{e}_2}{\vec{e}_1 \times \vec{e}_3} \cdot \vec{e}_3 \quad (3)$$

So, the error of the abstraction step is

$$\varepsilon = \frac{1}{2} \left( \frac{\vec{e}_1 \times \vec{e}_2}{\vec{e}_1 \times \vec{e}_3} \cdot \vec{e}_3 \times \vec{e}_2 \right) \quad (4)$$

The simplification of a map can be either performed with respect to an upper bound of error or to a given number of vertices. Whereas the first strategy is good for a general speed-up of the computation, for our application, we chose 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 that the embedded system runs out of memory, leading to a crash or in best case, a step without data. Needless to say, the results may become less useful after the abstraction, but for hard real-time systems like our application, not holding the deadline is worse than any approximate result. In order that other components of the system can judge how trustworthy the results of a map overlay are, the abstraction procedure additionally returns the error.

However, even if we restrict to the above operations for the abstraction, finding the optimal map with a maximal number of vertices that covers a given map is 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 NP. So, 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 abstraction suffers from the same problem as the rounding. After each approximation step, the map may not be planar any more: We must compute the overlay again after the abstraction. And even worse, new vertices may be not on the grid, so that the rounding must be done as well. This leads to the natural of both operation: First, abstract the polygons and then align them to the grid.

## 5 Experimental Results

We implemented the map overlay with the rounding and abstraction procedures described in the previous sections. Altogether, approximately 11000 lines of C code form a library that contains a three-valued geometry kernel based on rational numbers, map data structures with dedicated memory management and the overlay functions. We integrated the library into a prototype of a motion planning system developed for the use in the automotive industry. So, all the following scenarios are not artificial but reflect real-world input data. We measured the runtimes and the map size of different versions of the algorithm. The first one is the basic map overlay algorithm with arbitrary-precision arithmetic, the second one includes conservative rounding, and the third version also abstracts map data.

The table in Figure 14 shows the average values for randomly selected parameters (number of objects, di-

step	basic		+round		+abstract	
	time	size	time	size	time	size
20	31s	34	4.1 s	39	5.6 s	36
40	< 1h		10.7 s	46	11.3 s	49
60			21.5 s	89	17.3 s	53
80			39.2 s	162	25.7 s	65
100			44.8 s	45	31.9 s	42

**Figure 14. Benchmark Results**

rection and speed of the objects) of the scenarios: As it can be seen in, the algorithm using arbitrary-precision arithmetic has to give up very soon because of the exponential blowup of the numbers. With the rounding activated, the end of the computation can be reached. However, in the middle of the sequence, the algorithm tends to be rather slow, as many vertices are required for the computations. If abstractions are enabled, the size of the map and the runtimes of the steps remains stable. However, this has a price: The labelled area of the computed results is about 12% larger than the rounded solution.

## 6 Conclusions

In this paper, we presented a map overlay algorithm that had been designed for safety-critical embedded systems. Besides the careful overall design, in particular, with respect to degenerate cases, we tackled two significant problems that are posed by the new 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 map overlay to guarantee the execution with predefined execution time and memory limits.

## References

[1] L. Bolc and P. Borowik. *Many-Valued logics*. Springer, 1992.

[2] J. Brandt and K. Schneider. Dependable polygon-processing algorithms for safety-critical embedded systems. In L.T.Yang, M. Amamiya, Z. Liu, M. Guo, and F. Rammig, editors, *International Conference on Embedded and Ubiquitous Computing (EUC)*, volume 3824 of LNCS, pages 405–417, Nagasaki, Japan, 2005. Springer.

[3] J. Brandt and K. Schneider. Using three-valued logic to specify and verify algorithms of computational geometry. In K.-K. Lau and R. Banach, editors, *International*

*Conference on Formal Engineering Methods (ICFEM)*, volume 3785 of LNCS, pages 405–420, Manchester, UK, 2005. Springer.

[4] K. Bühler, E. Dyllong, and W. Luther. Reliable distance and intersection computation using finite precision geometry. In R. Alt, A. Frommer, R. Baker Kearfott, and W. Luther, editors, *Numerical Software with Result Verification*, volume 2991 of LNCS, pages 160–190, Dagstuhl Castle, Germany, 2004. Springer.

[5] M. de Berg, M. van Kreveld, M. Overmars, and O. Schwarzkopf. *Computational Geometry*. Springer, 2000.

[6] H. Edelsbrunner and E. Mücke. Simulation of simplicity: a technique to cope with degenerate cases in geometric algorithms. *ACM Transactions on Graphics*, 9(1):66–104, 1990.

[7] J. Foley, A. van Dam, S. Feiner, and J. Hughes. *Computer Graphics: Principles and Practice*. Addison Wesley, 2000.

[8] S. Fortune and C. Van Wyk. Efficient exact arithmetic for computational geometry. In *Symposium on Computational Geometry*, pages 163–172, 1993.

[9] M. Gordon and T. Melham. *Introduction to HOL: A Theorem Proving Environment for Higher Order Logic*. Cambridge University Press, 1993.

[10] M. Grimmer, K. Petras, and N. Revol. Multiple precision interval packages: Comparing different approaches. In R. Alt, A. Frommer, R. Baker Kearfott, and W. Luther, editors, *Numerical Software with Result Verification*, volume 2991 of LNCS, pages 64–90, Dagstuhl Castle, Germany, 2004. Springer.

[11] J. Hobby. Practical segment intersection with finite precision output. Technical report, Bell Laboratories (Lucent Technologies), 1993.

[12] S. Kleene. *Introduction to Metamathematics*. North Holland, 1952.

[13] D. Knuth. *The Art of Computer Programming*, volume 2. Addison Wesley, 1998.

[14] M. Lin. *Efficient Collision Detection for Animation and Robotics*. PhD thesis, Department of Electrical Engineering, University of California at Berkeley, 1993.

[15] K. Mehlhorn and S. Näher. *The LEDA Platform of Combinatorial and Geometric Computing*. Cambridge University Press, 1999.

[16] D. Priest. Algorithms for arbitrary precision floating point arithmetic. In P. Kornerup and D. Matula, editors, *Symposium on Computer Arithmetic*, pages 132–144. IEEE Computer Society, 1991.

[17] J. Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete and Computational Geometry*, 18(3):305–363, October 1996.

[18] I. Sutherland and G. Hodgeman. Reentrant polygon clipping. *Communications of the ACM*, 17(1):32–42, 1974.