Practical Point-in-Polygon Tests Using CSG Representations of Polygons

Robert J. Walker Jack Snoeyink

Department of Computer Science
University of British Columbia
201-2366 Main Mall
Vancouver, BC V6T 1Z4
Canada
{walker, snoeyink}@cs.ubc.ca

Technical Report TR-99-12

12 November 1999

Abstract

We investigate the use of a constructive solid geometry (CSG) representation of polygons in testing if points fall within them; this representation consists of a tree whose nodes are either Boolean operators or edges. By preprocessing the polygons, we seek (1) to construct a space-conserving data structure that supports point-in-polygon tests, (2) to prune as many edges as possible while maintaining the semantics of our tree, and (3) to obtain a tight inner loop to make testing the remaining edges as fast as possible. We utilize opportunities to optimize the pruning by permuting sibling nodes. We find that this process is less memory-intensive than the grid method and faster than existing one-shot methods.

1 Introduction

Point-in-polygon tests are common in computer graphics and are often used in raytracing where many points must be tested against a given polygon. The goal is to minimize the amortized cost per test in a practical situation. Unlike in standard computational geometry situations, we are concerned with the absolute performance of algorithms on polygons that will occur for such an application, and not so much on their asymptotic performance. Practical situations will tend not to contain worst-case polygons, and the number of edges dealt with really will tend to have an upper limit.

Practical point-in-polygon test algorithms can be classified into two major categories: those that utilize a vertex-list representation of the polygon as-is ("one-shot" methods), and those that perform preprocessing on the vertex-list representation, possibly transforming it into a different representation, before using the processed structure to perform multiple tests. The latter methods generally have a lower amortized cost when many tests will be performed on a particular polygon.

We discuss a new method utilizing preprocessing via a constructive solid geometry (CSG) tree representation of polygons. This method permutes sibling nodes within the tree and prunes unnecessary edge tests to make testing the remaining edges as fast as possible.

1.1 Existing Point-in-Polygon Algorithms

Haines [10] gives a thorough comparative treatment of existing point-in-polygon algorithms. Haines assumed that all test points would first be clipped against the polygons' bounding boxes; in his test implementation, he approximated bounding boxes by generating the polygon vertices within the unit square. His tests were conducted on one particular architecture.

Haines tested different types of polygons: regular polygons, and random, possibly self-intersecting, polygons. His results indicate that the fastest one-shot method is the crossings test of Shimrat [13] as corrected by Hacker [9]. For algorithms that require a little preprocessing and extra storage, Haines found that the half-plane algorithm of Green [8] and Spackman's algorithm [15] were fastest for random polygons with few sides, while the crossings test was faster for many sides; for regular polygons, the hybrid half-plane test was fastest for polygons with few sides, and for many sides the inclusion test [12] was fastest. The exception being when the ratio of bounding box area to polygon area is high, in which case the exterior edges algorithm is fastest. Haines' results indicate that the grid method [1] is the fastest algorithm when preprocessing and extra storage are available in abundance.

Details of these algorithms, test timings, and code are available in [10].

1.2 CSG Representation of Polygons

In a polygon, each bounding edge defines the boundary of two half-planes; orienting the edges counterclockwise about the polygon, we have that the half-plane to the left of the edge is considered to be inside the polygon, and the one to the right outside. The intersection and union of a set of such half-planes, performed in the correct order, defines the polygon and these half-planes are said to *support* the polygon. When testing a point against an edge, a Boolean value can be used to represent the truth of the statement: "The point lies inside"; the full CSG representation is then interpreted as a Boolean formula for the whole polygon.

Dobkin et al. [6] give an algorithm to construct a CSG representation of a non-self-intersecting polygon. The resulting representation is in the form of a directed binary tree whose nodes contain either Boolean operators or half-plane representations.[1] This tree may then be traversed to derive a monotone formula for the polygon—one in which every supporting half-plane appears exactly once, and no complementation is required. Thus, there are $n - 1$ logical operators appearing in the formula, and testing a point against such a structure will require $O(n)$ operations in the worst case: the hope is that the constants that occur in real cases will be small enough to make this method competitive in practical situations.

2 Point-in-Polygon Tests using CSG

2.1 n-ary Trees

Although a CSG tree can be constructed as a binary tree, an n-ary tree admits a more natural representation. Consider the polygon of Figure 1; its CSG tree representations are shown in Figure 2. The fact that edge u is an aunt of edge v rather than vice versa is arbitrary; the position of each is equivalent in terms of the logic represented by the tree, so we choose the n-ary representation instead. This structure will then allow us to reorder siblings in any manner we choose, which does not alter the logical structure of the tree, without having to consider rotations about any nodes.

[1] In our implementation, the tree contains oriented-edge representations, and the algorithm used to test against them computationally defines the corresponding half-planes. However, we will refer to an oriented edge of a polygon and the half-plane it defines interchangeably when no ambiguity results.

2.2 Evaluation by Tree Pruning

Conceptually, we wish to test a given point against each edge of the polygon. The location of the point can then be determined by evaluating the Boolean formula represented by the CSG tree. However, we take advantage of the alternating structure of the logical operators found in the CSG tree representation as seen in Figure 2b to eliminate unnecessary tests before computing their results. As soon as a subtree that is the child of an AND node is known to be FALSE, we may stop testing the children of that node, and assign the Boolean value FALSE to that AND node. Likewise, as soon as a subtree that is the child of an OR node is known to be TRUE, we may stop testing the children of that node, and assign the Boolean value TRUE to that OR node. The base case is at the leaves where we assign TRUE if the testing point is to the left of the edge, and FALSE if it is to the right. When a Boolean value is selected for the root of the tree, a value of TRUE indicates that the point is inside the polygon and FALSE indicates outside.

Since we are effectively pruning the CSG tree for some test points, we wish to permute the children of nodes within the tree to minimize the number of edges that need to be tested against, on average. In general, finding the permutation that yields optimal average performance requires us to analyze all such possible permutations (Section 3.3.1). We resort to heuristic methods to approximate the optimal permutation while performing less work. Such heuristics will be discussed in Section 3.3.

2.3 Obtaining a Tight Inner Loop

After each edge is tested, we ideally want only two alternatives as to which edge to test next. This allows us to eliminate a costly selection process from the inner loop of the testing algorithm. The selection of the next edge is determined only from the Boolean result of testing the current edge. These choices can be encoded in a new data structure, the *edge-sequence graph* (ESG) (see Figure 3).

An ESG is a data structure representing a single polygon; it contains two members: an array of edges defining the polygon, and the lookup array containing indices into the edge array. The lookup array is two dimensional: there are two integers in it for each member of the edge array. One integer indicates the index of the next edge to test if the current test yields the Boolean value FALSE, and the other is for TRUE. Arrays are assumed to be indexed from 0 to size $- 1$.

Given an ESG, testing a particular point against the polygon the ESG represents is straightforward. We begin by testing the point against the first edge in the edge array of the ESG; the next edge to test is determined by the appropriate index found in the lookup array. Such testing continues until the indicated next index is -1, in which case the current Boolean test result becomes the result for the full point-in-polygon test.

In the following algorithm, Points are represented by an x-y pair of Cartesian coordinates while Edges are represented by the oriented lines upon which they lie.

```
PointInPolygon( ESG esg, Point point ) returns Boolean

Integer index
Edge edge
Integer result

index := 0
do
   edge := esg.edges[index]
   result := Edge-PointLeftOf( edge, point )
   index := esg.lookup[index][result]
```

```
 9  until index = $-1$
10  return result
```

If edges are represented as lines in homogeneous coordinates, as they were in our implementation, then in 2-space the call to Edge-PointLeftOf() in line 6 would be equivalent to:

$$result := (edge.w + point.x \times edge.x + point.y \times edge.y \geq 0).$$

If strict containment within the polygon is required, the operator in this formula should be changed from "$\geq$" to "$>$". The PointInPolygon() procedure lends itself to optimization depending upon the implementation language and system architecture.

2.4 Experimental Results

The algorithm was tested in accordance with the testing methodology of Haines [10] with a few exceptions. Polygon vertices were randomly generated then scaled to fit the resulting polygon's bounding box to the unit square. Since the CSG representation can only cope with non-self-intersecting polygons, a simple edge-swapping algorithm was used to unfold the generated polygons prior to testing. All points to be tested against the polygons were generated pseudo-randomly, uniformly distributed across the polygon's bounding box.

Each algorithm was tested to see how long it took to perform point-in-polygon tests on an identical set of 50 pseudo-randomly generated points against each of 50 pseudo-randomly generated polygons. Some tests were repeated and averaged to alleviate timing inaccuracies. Tests were performed on a Silicon Graphics Indy workstation with a 133 MHz IP22 CPU; all processes save system daemons were disabled during test execution to further minimize variability of measurements.

The results for the various methods are shown in Table 1. Only the half-plane method is faster (14%) for triangles than the CSG method using the simply-sorted heuristic described in Section 3.3.3, and only the 100×100 grid method is equally fast for quadrilaterals. Only the memory-intensive grid and trapezoid methods outperform the CSG method as polygons grow large; the nearest other rival (the crossings method) is 208% more time consuming for a 1000 vertex polygon.

Tests were also performed for sets containing only randomly-generated convex polygons; the results are given in Table 2. The CSG method is not competitive for this restricted class of polygons when test points are uniformly distributed and confined to each polygon's bounding box.

2.5 Memory Requirements

There exists a tradeoff between time to perform point-in-polygon tests and space to store the data structures to support the algorithms. The memory requirements described in this section refer to final storage of the data structures used for the actual point-in-polygon tests; preprocessing requirements were not examined, and our implementation of the CSG method is not optimized for preprocessing performance or memory footprint.

The CSG method described in this paper requires an array of edges (each of which consists of three reals), an integer recording the number of edges, a number-of-edges $\times$ 2 array of indices into the edge array, and a single bit to indicate orientation of the polygon. This totals $3Rn + (1 + 2n)I + B$ bytes for an n edge polygon; in our implementation for a Silicon Graphics Indy architecture, double-precision reals, short integers, and a full byte for the single bit are used. The total memory usage is therefore $28n + 3$ bytes.

Haines' implementation of the $p \times q$ grid method uses a constant three integers, eight double-precision reals, and three pointers, plus $p + q + 2$ doubles and pq grid cells. Each grid cell requires two integers and a pointer, plus one grid record for each edge passing through that cell. A grid record requires 10 double-precision reals. There is a minimum of n grid records in total if each edge is contained in exactly one grid cell; there is a maximum of npq grid records if each edge is contained in every cell. The average practical case tends to be much closer to the minimum. Haines' implementation works out to $3I + 10R + 3P + (p + q)R + pq(2I + P) + 10Rn$ bytes minimum and $3I + 8R + 3P + (p + q + 2)R + pq(2I + P) + 10Rnpq$ bytes maximum. This reduces to a range of $80n + 3618$ to $32000n + 3618$ bytes for the 20×20 grid method, or $80n + 11698$ to $80000n + 11698$ bytes for the 100×100 grid method.

Haines' implementation of the p-bin trapezoid method uses a constant one integer, six doubles, and one pointer, plus p trapezoids. Each trapezoid uses two doubles, one integer, and one pointer, plus one edge record for each edge falling within the bin. An edge record uses two integers and two doubles. There is a minimum of n edge records if each edge falls exactly into one bin, and a maximum of np if each edge falls within every bin (when the testing region is limited to the polygon bounding box, there is a minimum of 2 edge records per bin, and so the minimum is bounded below by $\max\{n, 2p\}$). The average distribution of edges would not be as skewed towards the minimum as for the grid method. This implementation works out to $6R + I + P + p(2R + I + P) + n(2I + 2R)$ bytes minimum and $6R + I + P + p(2R + I + P) + np(2I + 2R)$ bytes maximum. This reduces to a range of $20n + 494$ to $400n + 494$ bytes for the 20-bin trapezoid method, or $20n + 2254$ to $2000n + 2254$ bytes for the 100-bin trapezoid method.

The grid method tends to be more memory-intensive than the CSG method (2.5 to 2500 times as expensive), but significantly faster (15 to 40 times faster). The trapezoid method is also more memory-intensive (0.7 to 70 times as expensive) in general, but only somewhat faster (1.5 to 3 times faster). Application-dependent point-in-polygon algorithm selection is necessary for maximal performance.

3 Building the CSG Tree and Edge-Sequence Graph

3.1 Building the CSG Tree

Dobkin et al. [6] give a constructive proof that a CSG representation exists for every non-self-intersecting polygon, consisting of a Boolean formula in which each edge appears once and without complementation. We give a conceptual overview of the algorithm and refer the interested reader to the original for the details.

Each edge of a polygon can be described as being a segment of a line dividing the plane in half. By orienting the edges counterclockwise, we define the half-plane to the left of an edge to be the one of interest, and say that it supports the polygon. The interior of the polygon can then be described by a combination of Boolean AND and OR operators acting on the half-planes supporting the polygon. Whenever two edges meet at an acute angle, the interior of the polygon is described by an AND of the corresponding half-planes; an obtuse angle requires an OR of these half-planes. As a result, we can build up the sequence of operations that describe the interior, but the correct order in which these operations must be applied (the parenthesization of the formula) is still to be determined.

Parenthesization begins by dividing the polygon into two bi-infinite chains by splitting it at its left- and rightmost vertices and extending the edges incident upon these vertices to infinity as in Figure 4. The convex hull of a bi-infinite chain is determined and the extremal vertex in the direction opposite the vector sum of the two semi-infinite edges of the chain is found. The chain is then split at this vertex, and the incident edges extended to infinity, forming two smaller chains; the process proceeds recursively, ending on a chain when it contains only one edge. Whenever a split occurs, the half-planes corresponding to edges in a chain are grouped within parentheses in the Boolean formula for the polygon; this results in a properly-nested formula consisting of unambiguously ordered binary operations. The difficulty is in performing this construction efficiently; Dobkin et al.'s algorithm is a practical $O(n \log n)$ for an n-edge polygon.

Obtaining a binary tree representation of the resultant formula is a trivial operation; a CSG formula is either a half-plane, or a binary operation on two smaller formulae: a half-plane becomes a leaf node, while the binary operator becomes the root of a tree whose two children are subtrees formed by the subformulae.

The binary tree representation is consolidated into an n-ary form by merging adjacent Boolean operator nodes of the same type [11]. It might be possible to alter the tree construction algorithm to directly create the n-ary form; however, since we have not attempted to optimize the preprocessing stage of this algorithm, the existing algorithm of Dobkin et al. was not modified.

3.2 Building the Edge-Sequence Graph

For a given point p, we can evaluate every leaf of the CSG tree, and hence, every edge of the polygon it represents. The logical operators within the tree could then be evaluated to determine the location of p. However, we can take advantage of the alternating structure of the logical operators found in the CSG tree to prune unrequired edges from the tree during traversal. As soon as a node n evaluates to FALSE and n is the child of an AND node m, we may stop testing the children of m and assign the Boolean value FALSE to m. Likewise, as soon as a node n' evaluates to TRUE and n' is the child of an OR node m', we may stop testing the children of m' and assign TRUE to m'. At a leaf node e, we assign TRUE if p is to the left of the edge represented by e, and FALSE if it is to the right. When a Boolean value is determined for the root of the tree, the location of p is known: a value of TRUE indicates that p is inside the polygon and FALSE indicates that it is outside.

The CSG tree is evaluated via postorder traversal. After each edge is tested, there are exactly two choices as to which edge to test next; the selection of this edge is determined from the Boolean result of testing the current edge. These choices can be encoded in a new data structure, the edge-sequence graph (ESG), in such a way that most of the processing time required to perform the point-in-polygon test will be occupied in testing the point against edges of the polygon rather than determining which edges to test.

We amortize the cost of constructing an ESG across tests of multiple points for the same polygon. We assume that once a given embedding for the tree has been chosen, that embedding is used to perform all point-in-polygon tests henceforth. It would be costly to alter the embedding during the actual point test.

An ESG is a data structure representing a single polygon; it contains an array of edges (*edges*), an array of indices into the edge array (*lookup*), an integer indicating the number of edges in the polygon (*edgec*), and a Boolean flag indicating whether the polygon is oriented counterclockwise (*ccw*). The latter is not strictly required, but is of practical value when comparing the algorithm against other algorithms that ignore orientation of edges in defining the true interior of a polygon. The *lookup* array contains two elements for each edge occurring in the polygon. An ESG is constructed from a CSG tree via the following algorithm.

```
ESG-Construct( CSGTree csgtree ) returns ESG

CSGTreeNode node
Integer counter
ESG esg

CSGTree-EnumerateLeaves( CSGTree csgtree )
esg.edge-count := csgtree.edge-count
node := CSGTree-FirstLeaf( csgtree.root )
counter := 0
while node $\neq$ NIL do
   esg.lookup[esg][$2 \times$ counter] := CSGTree-Branch( node, FALSE )
   esg.lookup[esg][$2 \times$ counter $+ 1$] := CSGTree-Branch( node, TRUE )
   esg.edges[counter] := node.edge
   counter := counter + 1
```

```
        node := CSGTree-NextLeaf( node )
    esg.ccw := ( csgtree.root.op = AND )
    return esg
```

This algorithm requires that the the calls to CSGTree-Branch() return the index within the edges array representing the next edge to test in each situation. The enumeration of the edges can be arbitrary; however, the node numbered zero will always be tested first in the point-in-polygon algorithm.

After enumeration, ESG-Construct() proceeds to walk through the leaves of the CSG tree. At each leaf, it determines which edge would next need to be tested given that the current edge tested either TRUE or FALSE against a particular point. The indices of these edges are then stored in the next two positions in the lookup array; at the same time, the current edge is also copied to the edge array. Finally, the Boolean flag indicating whether the polygon is oriented counterclockwise is set—the root node of the CSG tree is an AND node if and only if the polygon is oriented counterclockwise. The CSGTree-FirstLeaf() and CSGTree-NextLeaf() are simple tree-minimum and tree-successor functions.

The choice of next edge to test can be made according to structural patterns within the CSG tree: there are four cases dependent on whether the current edge evaluated to TRUE or FALSE and whether the parent of the current leaf is an AND node or and OR node. The case in which the current edge evaluated to FALSE and the parent of the current leaf is an AND node is illustrated in Figure 5; we refer to this as case FA. The choice is made during preprocessing and is encoded within the ESG.

In case FA, since the current leaf has evaluated to FALSE, the AND node parent is immediately known to be FALSE. Therefore, any siblings to the right of the current leaf need not be evaluated. We know that the value of the grandparent of the current node (an OR node) is not determinable yet, or we would not have tested the current leaf. Therefore, we must evaluate the siblings of the parent to its right. If the parent has no siblings to the right, the grandparent is then known to be FALSE, and testing passes to right-siblings of the great-grandparent. This pattern continues up the tree, skipping siblings of OR nodes. As soon as an extant subtree in one of the positions of interest is found, testing proceeds to the minimum leaf in that subtree. If no such subtree is found, testing can end as the CSG tree root has been assigned the value FALSE, and thus the point being tested would fall outside the polygon.

In case FO, subtree siblings of OR nodes are still skipped. Thus, an extant subtree is sought first in the right-sibling of the current leaf, any aunt subtrees are ignored, then a great-aunt subtree is sought, and so on. If the root of the entire tree is passed in this search, it evaluates to FALSE. The pattern for case TO is analogous to that for case FA, and the pattern for case TA is analogous to that for case FO. The root of the entire tree evaluates to TRUE for the latter two cases, if the root is passed during the search.

The algorithm for determining the next edge to test against is as follows; *node* is the current leaf and *result* is its Boolean value. It is assumed that the CSG tree has had its leaves enumerated in left-to-right order, starting with 0, prior to execution of this procedure; -1 is used to represent "no further tests".

```
CSGTree-Branch( CSGTreeNode node, Boolean result ) returns Integer

    CSGTreeNode ancestor

    if ( node.parent.op = AND and result = FALSE ) or
          ( node.parent.op = OR and result = TRUE ) then /* Cases FA and TO */
        ancestor := node.parent
    else /* Cases FO and TA */
        ancestor := node
    return CSGTree-FAES( ancestor )
```

The CSGTree-FAES() procedure returns the number of the minimum leaf in the subtree rooted at the right-sibling of the current ancestor node (it Finds the minimum node of Alternating Extant Subtrees). If none such exists, it resets the *ancestor* to be the grandparent of the current ancestor. If an ancestor is ever found not to exist, i.e., the root of the tree has been reached, -1 is returned to indicate that no more tests should be performed.

```
CSGTree-FAES( CSGTreeNode ancestor ) returns Integer

CSGTreeNode ancestor, next, right-sibling

while ancestor ≠ NIL do
   right-sibling := RightSibling( ancestor )
   if right-sibling ≠ NIL then
      next := CSGTree-FirstLeaf( right-sibling )
      return next.number
   ancestor := ancestor.parent
   if ancestor = NIL then
      return -1
   ancestor := ancestor.parent
return -1
```

3.3 Embedding the CSG Tree

Figure 2b shows a particular order for our n-ary tree CSG representation of the polygon in Figure 1. This ordering is not unique. Since the order of edges in the ESG data structure can greatly affect average performance of point-in-polygon tests, it is important that the CSG tree (the structure the ESG is derived from) be properly embedded in the plane by permuting sibling nodes.

The fastest permutation to perform in preprocessing is the default obtained from the construction of the CSG tree, i.e., none at all. A simple heuristic permutation is to sort sibling subtrees by increasing number of edges in them, breaking ties by testing longer edges first. More sophisticated heuristics are possible.

3.3.1 Defining an Analytic Cost Metric

To analytically determine the average performance of a given permutation of nodes within the tree, a metric is required. Consider the test points to be distributed uniformly across the bounding box of the polygon, and prescribe an edge-sequence graph for the polygon. Let us say that the initial edge of the polygon to be tested divides the bounding box in half. We know that half of the area of the bounding box is determined to be outside the polygon by this single test, and therefore, there is only a 50% probability that a second edge needs testing. Likewise, if the second edge to be tested eliminates half the remainder of the bounding box, there is only a 25% probability that a third edge need be tested. Expected value is linear; therefore, if there are three edges in this example polygon, the sum of their probabilities (1.75) is the expected number of edge tests for this polygon, edge order, and bounding region.

More generally, we can see that each edge lies on a line that divides the bounding region into two parts; if a given edge is tested, edges on one side or the other of this are successively tested with a probability equal to that of the region on that side of the given edge. We can see this as narrowing in on the particular region in which the tested point lies; the probability that a point falls in this region is proportional to the area of the region, for a uniform distribution (see Figure 6).

We now give a more formal description of the calculation of the expected number of edge tests. A finite set of lines L dissects the plane into a finite set of regions, called the *arrangement* $\mathcal{A}(L)$ of L. Let there be a *probability density function* ρ defined over some portion Π of the plane. We define the Π-*restricted arrangement* $\mathcal{A}_\Pi(L)$ of L to be the elementwise intersection of $\mathcal{A}(L)$ with Π removing null intersections, i.e.,

$$\mathcal{A}_\Pi(L) = \left(\bigcup_{R_i \in \mathcal{A}(L)} \{R_i\} \cap \Pi \right) - \{\emptyset\}. \tag{1}$$

We refer to the elements of $\mathcal{A}_\Pi(L)$ as the *restricted regions* of the arrangement $\mathcal{A}(L)$. Thus, each restricted region R_i^* has a probability

$$P(R_i^*) = \int_{R_i^*} \rho(p)\, dp \tag{2}$$

defined over it. For a uniform distribution, this probability is equal to the area of the restricted region divided by the area of Π.

To determine the expected number of edge tests required for a particular ESG, we proceed as follows. Define L to consist of the lines on which the edges of the polygon lie; orienting each line identically to its defining edge, each restricted region will lie either to the right or to the left of a given line. For each line, store two Boolean vectors: a *position* vector, each element of which indicates whether the corresponding restricted region lies to the left of the line, and a *restriction* vector with all elements initialized to FALSE, each of which corresponds to a restricted region.

We assume that the ESG is traversed in a topologically sorted manner. The first edge has its restriction vector initialized to all TRUEs. We set to TRUE all the elements of its left-child's restriction vector with indices corresponding to the TRUE elements of the current position vector. We set to TRUE all the elements of its right-child's restriction vector with indices corresponding to the FALSE elements of the current position vector. This further restricts the portion of the plane under consideration for the child edges to that part on either the right or the left of the current line.

For every other edge encountered in the ESG, zero or more elements of its restriction vector will have been set to TRUE; when the probabilities of the corresponding restricted regions are summed, the value determined is the probability that this edge is tested. We continue through the ESG by masking the edge's restriction vector with its position vector, and setting its left-child's restriction vector to the result; masking the edge's restriction vector with the ones complement of its position vector, we set its right-child's restriction vector to the result. This process is summarized in the following algorithm.

```
ExpectedCost( ESG esg, Array portion of Polygon ) returns Real

Array arrangement of Polygon
Array prob of Real
Array( esg.edge-count ) position of Array of Boolean
Array( esg.edge-count ) restriction of Array of Boolean
Array( esg.edge-count ) topological-sort-order of Integer
Integer index, i, j
Real result

/* Preprocessing */
( arrangement, position ) := FormRestrictedArrangement( esg, portion )
```

```
10  Array-SetSize( prob, arrangement.size )
11  i := 0
12  while $i <$ arrangement.size do prob[i] := ComputeProbability( arrangement[i] )
13  i := 0
14  while $i <$ esg.edge-count do Array-SetSize( restriction[i], arrangement.size )
15
16  /* Initialization */
17  topological-sort-order := ESG-FindTopologicalSortOrder( esg )
18  i := 0
19  while $i <$ arrangement.size do restriction[0][i] := 1
20  i := 1
21  while $i <$ esg.edges do
22    j := 0
23    while $j <$ arrangement.size do restriction[i][j] := 0
24
25  /* Traversal */
26  result := 0.0
27  i := 0
28  while $i <$ topological-sort-order.size do
29    index := topological-sort-order[i]
30    right-child := esg.edges[index][0]
31    left-child := esg.edges[index][1]
32    j := 0
33    while $j <$ arrangement.size do
34      if restriction[index][j] then result := result + prob[j]
35      if position[index][j] then restriction[left-child][j] := restriction[index][j]
36      else restriction[right-child][j] := restriction[index][j]
37
38  return result
```

The ExpectedCost() algorithm is organized in this fashion to optimize for determining the expected cost of multiple permutations of a given polygon; the restricted arrangement of the edges in the polygon is identical for all permutations, and so lines 9 to 14 can be performed as a preprocessing step. Strictly speaking, the restriction vectors do not need to be dynamically resized as the algorithm suggests, since we have an upper bound of $O((n+m)^2)$ on the number of possible restricted regions; this bound is not necessarily tight, however, so static allocation could potentially waste much space.

The ESG can be constructed to have its edges in topologically sorted order by default—simply by numbering the edges with increasing indices in their left-to-right order in the CSG tree. Doing so can interfere with other constraints, such as maintaining adjacent edges with adjacent indices as is required for the Polygon-PointLocation() algorithm described in Section 2.3; this is why a level of indirection is used instead of simply sorting the ESG itself.

A Π-restricted arrangement of n lines, where Π is defined by m edges, consists of $O((n+m)^2)$ restricted regions. The FormRestrictedArrangement() procedure must define the polygons that are these restricted regions, and determine which restricted regions are to the right and to the left of each line. Edelsbrunner [7] gives an algorithm for constructing a two-dimensional arrangement in $O(n^2)$ operations; we can effectively form the restricted arrangement by incrementing the arrangement of n lines by the lines defined by the edges of Π. Any "restricted regions" outside Π are then assigned a probability of zero.

During the construction of the arrangement, edge nodes in the incidence graph can be marked with their defining line. The incidence graph can be partitioned at these nodes and a single face node on either side of the partition can be tested to determine if all the faces on that side are to the right or to the left of the line. The position vector for the line is then set by traversing the partitioned incidence graph in $O((n+m)^2)$ operations; determination of face position therefore takes $O(n(n+m)^2)$ operations for all the lines in L.

As long as any dynamic allocation of the probability array, restriction vectors, position vectors, or arrangement data structures and computing the probabilities of the restricted regions are all bounded by $O(n(n+m)^2)$, the preprocessing phase of the algorithm is bounded by $O(n(n+m)^2)$.

A topological sort on the ESG can be performed in $\Theta(n)$ operations; the initialization can thus be performed in $O(n(n+m)^2)$ operations. The ESG traversal will also require $O(n(n+m)^2)$ operations. The bound on the total number of operations for ExpectedCost() is therefore $O(n(n+m)^2)$, under the assumptions made in the previous paragraph.

3.3.2 Proof of ExpectedCost() Correctness

We prove here that the ExpectedCost() algorithm of Section 3.3.1 correctly computes the expected number of edge tests that will be performed on points in an arbitrary distribution.

It should be clear that edge-sequence graphs as we use them contain no cycles, and are therefore topologically sortable; thus, traversing an ESG in a topological-sort order, we will eventually arrive at every edge contained in the ESG. The probability of testing the first edge in the graph is unity, and since the sum of the probabilities of all the restricted regions of the arrangement is unity by definition, the initialization provided by the algorithm yields the correct probability for this first edge. It should also be clear that all and only those regions to the left of the first edge are added to the region set for the left-child, and that all and only those regions to the right of the first edge are added to the region set for the right-child. We arbitrarily select one of the children to be on the current path, without loss of generality. The probability that the child edge is tested on the current path is equal to the sum of the probabilities of its set of regions on the current path.

Assume that, for the kth edge on the current path, its parent on the current path correctly provided it the set of regions against which it will be testing on this path, and that the probability that it will be tested on this path is the sum of the probabilities of those regions. Also assume that this set of regions does not overlap with the sets provided by parents on other paths. If the point being tested lies to the left of this edge, all and only those regions in the set on the current path that are to the left of the edge are passed to the $(k+1)$th edge on the current path. If the point being tested lies to the right of this edge, all and only those regions in the set on the current path that are to the right of the edge are passed to the $(k+1)$th edge on the current path. Since there is no overlap between the sets of regions for each path as they impinge upon the current edge, regions not in the set on the current path are never passed to the child's set on the current path. Therefore, the probability of an edge being tested on a given path is correctly computed.

If an edge does not lie on any path from the first edge, its restriction vector will contain only FALSE elements; therefore, the probability that any edge is tested on a path rooted at any edge except the first is zero. And so, ExpectedCost() traverses all and only the possible testing paths through an ESG.

A path through an ESG for a given point is unique and can be selected deterministically; the probability that a given point is tested on more than one path is zero. Therefore, the probability of a given edge being tested is the sum of the probabilities of it being tested on each individual path. We therefore do not need to distinguish between the sets of regions for each path, and can keep the regions for all paths through a particular edge in a single set.

Through the linearity of expectation, the expected number of edges tested in an ESG is equal to the sum of the probability of each one being tested. Thus, ExpectedCost() correctly computes the expected number of edges to be tested.

3.3.3 A Simple Heuristic for Permutation Determination

Let us say that we have a subtree rooted with an AND node. Now let us say we are testing a point against this tree that happens to fall outside the region supported by this subtree—the value of the root of the subtree will be FALSE, so we want to find a child that is FALSE as fast as possible. The easiest way to do this is if there is a leaf node that is a child of the root of our subtree, for which the point lies to the right; if we are to appeal to an OR node that is a child of the root of the subtree, we are more likely to have to test many of its children since all must be false for the OR node to be false. Thus in this situation, we prefer to test leaf nodes before subtrees. The same result holds if our subtree is rooted with an OR node, and our point falls within the region supported by the subtree.

Now let us say that our point does lie within the region supported by this subtree; now the value of the root of the subtree will be TRUE, so all the children will be TRUE. However, we must prove this by actually computing the value of all the children, and since all the children must be computed anyway, their order does not matter. The same result holds if our subtree is rooted with an OR node, and our point falls outside the region supported by the subtree.

Now we have that leaves should be evaluated before subtrees. For the order within these groups, we use the heuristic applied by Worley and Haines in other tests [17], that it is more efficient to test longer edges first. And since we want to minimize the number of tests, we sort subtrees in order of increasing numbers of edges within them.

Table 1 illustrates that this heuristic (noted as the sorted variety) does improve average performance over the default permutation 0–20%, becoming greater for larger polygons. It is important to note that this is average performance, however; this heuristic does slow down performance for some instances of polygons.

Figure 7 illustrates the analytic performance of this simple heuristic for five random pentagons A–E. The range of possible costs for performing point-in-polygon tests against each of five random pentagons was calculated for all the possible permutations of each; points to be tested were assumed to be distributed uniformly across the bounding box of each.

This computation was repeated for the random polygons, with 10 or fewer edges, used in the point-in-polygon tests listed in Table 1. Figure 8 shows, from top to bottom the average maximum cost, average cost of the default permutation, average cost of the simply-sorted permutation, and average minimum cost, across the 50 sample polygons. The predicted cost of the simply-sorted permutation is consistently lower than that of the default; however, there is significant room for improvement, as the gap between the heuristic permutation and the true minimum is trending towards growth.

4 Extensions

We have considered a number of extensions to and refinements of the algorithms discussed in this paper, and began preliminary investigation of these in some cases.

4.1 Raytracing in 3D

Generalization of the algorithms described in this paper to higher dimensions is desirable; unfortunately, Dobkin et al. have proven that not all polyhedra of more than two dimensions can be described by monotone Boolean formulae—the keystone to our algorithm for the construction of the ESG data structure. This does not mean that the ESG data structure cannot be used for performing point-in-polytope tests for higher dimensions, only that constructing them will require a different, potentially expensive algorithm.

Ray/polygon intersection is the basis of raytracing algorithms in computer graphics rendering. The algorithms described in this paper generalize to detect ray/polygon intersections in 3-space in a straightforward way, once one is familiar with *Plücker coordinates* [16]. They also use the minimum arithmetic precision possible for the problem, which allows them to be implemented exactly. For small polygons, the number of arithmetic operations is less than that for Badouel's algorithm [2] for example, but this is an unfair comparison. Badouel uses a clever reduction to 2D point location, but then employs one of the poorer 2D point location schemes. Using his reduction with a better 2D scheme gives fewer arithmetic operations than detection directly in 3-space, although the reduction still increases the arithmetic complexity.

To generalize our algorithms, we use Plücker coordinates to represent edges and lines in 3-space. Using homogeneous coordinates, an oriented edge will start at point $\mathbf{p}$ and end at point $\mathbf{q}$. Such an edge can be represented as a Plücker point through a 6-tuple of determinants:

$$\widehat{\mathbf{pq}} = \left(\begin{vmatrix} p_w & p_x \\ q_w & q_x \end{vmatrix}, \begin{vmatrix} p_w & p_y \\ q_w & q_y \end{vmatrix}, \begin{vmatrix} p_w & p_z \\ q_w & q_z \end{vmatrix}, \begin{vmatrix} p_x & p_y \\ q_x & q_y \end{vmatrix}, \begin{vmatrix} p_x & p_z \\ q_x & q_z \end{vmatrix}, \begin{vmatrix} p_y & p_z \\ q_y & q_z \end{vmatrix} \right). \tag{3}$$

Similarly, an oriented line passing from the point $\mathbf{r}$ to the point $\mathbf{s}$ can be represented as a Plücker hyperplane through a different 6-tuple of determinants:

$$\widetilde{\mathbf{rs}} = \left(\begin{vmatrix} r_y & r_z \\ s_y & s_z \end{vmatrix}, - \begin{vmatrix} r_x & r_z \\ s_x & s_z \end{vmatrix}, \begin{vmatrix} r_x & r_y \\ s_x & s_y \end{vmatrix}, \begin{vmatrix} r_w & r_x \\ s_w & s_x \end{vmatrix}, - \begin{vmatrix} r_w & r_y \\ s_w & s_y \end{vmatrix}, \begin{vmatrix} r_w & r_z \\ s_w & s_z \end{vmatrix} \right). \tag{4}$$

The dot product of these two vectors happens to be identical to the determinant:

$$\delta = \begin{vmatrix} p_w & p_x & p_y & p_z \\ q_w & q_x & q_y & q_z \\ r_w & r_x & r_y & r_z \\ s_w & s_x & s_y & s_z \end{vmatrix}; \tag{5}$$

this is reasonable, since the expansion of δ into 2×2 minors gives

$$\delta = \sum_{i=1}^{6} \widehat{pq}_i \, \widetilde{rs}_i \, . \tag{6}$$

This determinant will be zero if all four points are coplanar; the right-hand rule explains the sign of a non-zero δ. Point the thumb of your right-hand in the direction of the edge $\mathbf{pq}$ and curl your fingers around this line. If the line $\mathbf{rs}$ passes $\mathbf{pq}$ in this direction, δ will be positive; if it passes in the opposite direction, δ will be negative. Since we only care about the sign of δ, we can scale it or the two vectors arbitrarily; this means that we can normalize each 6-tuple by dividing it by one of its elements, effectively reducing it to a 5-tuple.

We can use this for performing raytracing by altering the PointInPolygon() procedure slightly. Edge-sequence graphs will now contain the Plücker point coordinates, and the oriented line (i.e., the ray) to be tested is passed to the new procedure instead of a single point. We simply replace the Edge-PointLeftOf() procedure with a test of the sign of the determinant d between the edge being tested and our ray:

result := (DotProduct(edge, line) $\leq$ 0);

notice that the operator sign has been flipped.

There is one additional concern: we need to make sure the polygon is oriented correctly. Depending on whether we are looking at its backface or its front-face, the interior and exterior of the polygon will be switched. To determine the side of the polygon on which a point lies, we can use three vertices $\mathbf{p}$, $\mathbf{q}$ and $\mathbf{r}$ of the polygon to expand Equation 5 into:

$$\pi(\mathbf{s}) = s_w \begin{vmatrix} p_x & p_y & p_z \\ q_x & q_y & q_z \\ r_x & r_y & r_z \end{vmatrix} - s_x \begin{vmatrix} p_w & p_y & p_z \\ q_w & q_y & q_z \\ r_w & r_y & r_z \end{vmatrix} + s_y \begin{vmatrix} p_w & p_x & p_z \\ q_w & q_x & q_z \\ r_w & r_x & r_z \end{vmatrix} - s_z \begin{vmatrix} p_w & p_x & p_y \\ q_w & q_x & q_y \\ r_w & r_x & r_y \end{vmatrix}. \tag{7}$$

For a variable point $\mathbf{s}$, this is the equation for the plane π that contains the polygon. The four minors in 7 can be computed in the preprocessing step and stored with the polygon. If $\pi(\mathbf{s})$ is greater than zero, the point $\mathbf{s}$ is on the side of the polygon where the edges are oriented counterclockwise, i.e., the front-face; if it is less than zero, $\mathbf{s}$ is on the other side. For a ray with origin $\mathbf{u}$ passing through point $\mathbf{v}$, if $|\pi(\mathbf{u})| \leq |\pi(\mathbf{v})|$, the ray does not intersect the plane of the polygon. Otherwise, if $\pi(\mathbf{u}) > \pi(\mathbf{v})$, the ray is pointing towards the front-face of the polygon; if $\pi(\mathbf{u}) < \pi(\mathbf{v})$, the ray is pointing towards the backface.

The disadvantages of this alteration to the CSG approach are in time and storage. In 3-space, each edge must store its Plücker coordinates in either 5 rationals or 6 integers. PointInPolygon() performs, for each edge tested, 4 rational multiplications if the Plücker coordinates have been normalized (using different coordinates to normalize Plücker points and Plücker hyperplanes) or 6 integer multiplications.

The main advantage is the low arithmetic precision required. Suppose that the input points have w-coordinates of unity, that other coordinates are b-bit integers, and that for endpoints of an edge or for the ray points $\mathbf{r}$ and $\mathbf{s}$ the maximum difference in any coordinate is an integer of $d \leq b$ bits. Then the coordinates of the Plücker points and hyperplanes are integers of d or $b + d + 1$ bits, and $b + 2d + 2$ bits are sufficient to contain the result of any Plücker dot product. Thus, we could, in 53-bit native arithmetic, exactly evaluate Plücker tests on coordinates with $b = 24$ bits when the difference between endpoints can be expressed in $d = 12$ bits.

Badouel [2] gives a two-step process to reduce ray/polygon intersection into a point-in-polygon test of a projection on one of the coordinate planes. Let P be the polygon in 3-space and let π denote the plane containing P. In the first step, after checking that the ray $\mathbf{rs}$ does intersect π, this intersection point $\rho = \mathbf{rs} \cap \pi$ is calculated. In the second step, the intersection is projected onto one of the coordinate planes—which plane is determined by the two longest dimensions of the axis-aligned bounding box for P in 3-space [14]. The ray intersects P if the projection of ρ is inside the projection of P.

Consider the arithmetic complexity in more detail. The two-step process parameterizes the ray $\mathbf{rs}$ and solves for the parameter t that determines when $\rho = r + t(s - r)$ lies on the plane π. This takes 1 division of the dot products of r and s with the plane; since these dot products are also used by the Plücker algorithm, we will not count them, but will count the division. Now, t is a rational number with numerator and denominator each having $b + 2d$ bits. To obtain the rational coordinates of ρ in the projection plane takes 2 more multiplications. Then a 2D point location such as the unaltered CSG method can be used in the projection, with 2 multiplications per edge tested . With integers, the division is avoided, 4 multiplications are needed to find the projection of ρ in homogeneous form with $2b + 2d$ bits, then each edge test takes 3 multiplications and uses $2b + 3d$ bits, plus two or three bits for rounding. Thus, the two-step process performs fewer multiplications/divisions if more than 2 edges are tested. Native 53-bit arithmetic supports exact computation on mere $b = 14$ bit coordinates when the difference between endpoints can be expressed in $d = 7$ bits.

The projection onto a coordinate plane can also reduce the performance of the fastest 2D point location methods, since the projection will tend to produce more long, skinny triangles. Although the grid method performs well on long, skinny triangles under normal circumstances, the projection will also alter the distribution of test points in a similar manner. As a result, the heaviest concentration of test points will occur in those grid cells in the vicinity of the polygon—where the grid method performs least well.

The final arbiter of performance will be empirical measurement of implementations. As with the 2D form of the CSG algorithm, the space-, time-, and precision-tradeoffs here need to be considered in an application-specific context.

4.2 Reporting More Precise Boundary Information

A minor tweak to the PointInPolygon() algorithm allows it to report whether the point is strictly inside, strictly outside, or on the boundary of the polygon. First, the lookup array is expanded to include a 2-bit flag for each member of the edge array. Second, the order of the members of the lookup array must be identical to the order of their corresponding edges in the polygon; this is easily arranged during construction of the ESG. If an edge test indicates that the point falls on the line being tested, the line immediately before and the line immediately after it are also tested. The position of the point relative to these two other lines allow the point to be bounded to determine if it falls within the interval defining the edge of the polygon; each bit of the flag is used to indicate if the point needs to be to the left or to the right of the corresponding line to fall on the edge.

```
Polygon-PointLocation( ESG esg, Point point ) returns ( Position, Integer )

 0   Integer index
 1   Edge edge, prev, next
 2   Position result
 3
 4   index := 0
 5   do
 6      edge := esg.edges[index]
 7      result := Edge-PointLocation( edge, point )
 8      if result = ON-BOUNDARY then
 9         prev := esg.edges[index - 1]
10         next := esg.edges[index + 1]
11         if Edge-PointInBounds( prev, next, flag1, flag2 ) then
12            return ( result, index )
13      index := esg.lookup[index][result]
14   until index = $-1$
15   return ( result, $-1$ )
```

The Position type takes on three values: INSIDE, OUTSIDE, and ON-BOUNDARY. The call at line 11 to Edge-PointInBounds() is equivalent to:

(Edge-PointLeftOf(*prev*, *point*) XOR *flag1*) NOR (Edge-PointLeftOf(*next*, *point*) XOR *flag2*).

4.3 Other Extensions

The preprocessing for the point-in-polygon algorithm we have discussed is currently inefficient both in implementation and design; we were concerned chiefly with the performance of the ESG data structure after preprocessing. However, even the amortized cost of preprocessing is not negligible if a polygon is not to be tested against arbitrarily many times; this is why it was necessary to attempt a heuristic solution to find the lowest-cost permutation of the CSG tree prior to its conversion to an ESG. Some applications might test against a polygon enough times to benefit from a point-in-polygon method using preprocessing, but not so many times that the cost of preprocessing can be ignored. Optimization of preprocessing must proceed under the constraint that its impact on point-in-polygon tests should be either nil, or unavoidable as in the case of the heuristic embedding.

Most of the experimental and analytical work in this paper on determining cost of the algorithms has concentrated on point-in-polygon tests in which the points to be tested are uniformly distributed. In particular application domains, such as geographical information systems, test points could be heavily concentrated in particular regions, e.g., queries about average annual rainfall at a city on a map. Specialized heuristics are required for non-uniform distributions, and the performance of all the methods tested here are likely to change in this situation.

The simply-sorted heuristic for embedding the CSG tree does a better than average job of speeding up the performance of the point-in-polygon algorithm; however, there is significant room for improvement, especially for polygons with many-sides, as evidenced by Figure 8.

We have begun an investigation of other possible heuristics by analyzing the cost of all the permutations of randomly-generated 4- and 5-sided polygons, to see why the fast ones are fast and the slow ones are slow. Selection of the initial edge to test against is often a major determinating factor in cost. We generally want to test against an edge whose corresponding line creates a region large in probability that is completely outside (or completely inside) the polygon. Such an edge is often not available, and it remains unclear if there exists one or two simple determining factors of cost in such an instance. Randomization of child nodes is another possible heuristic for increasing average performance. Given that the simply-sorted heuristic gives better than median or mean performance, on average, for polygons with a few sides, randomization will not improve performance here on average. For polygons with more sides this might not be the case.

It has come to our attention that the binary decision diagram (BDD) data structure from the field of formal verification is closely related to CSG trees and that ordered BDDs (OBDD) [4] are closely related to ESGs. A survey of OBDDs can be found in [5]. Reordering of variables in OBDDs is analogous to reordering edge tests in ESGs; it is recognized as a difficult problem and finding an optimal solution for variable ordering in an OBDD has been proven to be NP-hard [3]. With OBDDs, the chief concern is to reduce their size; with ESGs, it is to reduce their probability-weighted average path length. It is unclear whether these two concerns are compatible.

5 Conclusion

We have presented an algorithm for performing point-in-polygon tests using a constructive solid geometry (CSG) representation of polygons. This algorithm uses a new data structure, the edge sequence graph (ESG), that is constructed from a CSG representation of a given polygon during a preprocessing phase. The speed of the CSG point-in-polygon algorithm derives from a tight inner loop and the pruning of unnecessary edge tests, made possible by ESGs. Algorithms for the construction of an ESG and the calculation of the expected cost of using a given ESG to test a random point were also presented. Because the ordering of edges affects the cost of using an ESG, heuristics were presented for selecting a good ordering—deterministic selection of the fastest permutation is NP-hard. Three-dimensional raytracing and other extensions were briefly discussed.

The CSG point-in-polygon algorithm provides a tradeoff of slower execution for reduced storage space requirements over faster, existing methods. We have demonstrated empirically that this new point-in-polygon algorithm using a simple heuristic is faster on most non-convex polygons than all the one-shot methods presented by Haines. Only the half-plane method is faster for triangles (by 13%) and only the 100×100 grid method is equally fast for quadrilaterals. Only the grid and trapezoid methods outperform this algorithm as polygons grow large. We have shown that the average storage space required for this algorithm is significantly less than for the grid and trapezoid methods.

References

[1] Franklin Antonio. Faster line segment intersection. In David Kirk, editor, *Graphics Gems III*, chapter 1.4, pages 199–202. Academic Press, Boston, MA, USA, 1992.

[2] Didier Badouel. An efficient ray-polygon intersection. In Andrew S. Glassner, editor, *Graphics Gems*, pages 390–393. Academic Press, Boston, MA, 1990.

[3] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, 45(9):993–1002, 1996.

[4] Randal E. Bryant. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers*, C-35(6):677–691, August 1986.

[5] Randal E. Bryant. Symbolic Boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3):293–318, September 1992.

[6] David Dobkin, Leonidas Guibas, John Hershberger, and Jack Snoeyink. An efficient algorithm for finding the CSG representation of a simple polygon. In John Dill, editor, *Computer Graphics (SIGGRAPH '88 Proceedings)*, volume 22, pages 31–40, August 1988.

[7] Herbert Edelsbrunner. *Constructing Arrangements*, volume 10 of *EATCS Monographs on Theoretical Computer Science*, chapter 7. Springer-Verlag, Berlin, Germany, 1987.

[8] Chris Green. Simple, fast triangle intersection. *Ray Tracing News*, 6(1), 1993. `ftp://ftp.princeton.edu/pub/Graphics/RTNews`.

[9] R. Hacker. Certification of algorithm 112: position of point relative to polygon. *Communications of the ACM*, 5:606, 1962.

[10] Eric Haines. Point in polygon strategies. In Paul S. Heckbert, editor, *Graphics Gems IV*, chapter 1.4, pages 24–46. Academic Press, Boston, MA, USA, 1994.

[11] Donald E. Knuth. *Fundamental Algorithms*, volume 1 of *The Art of Computer Programming*. Addison-Wesley, second edition, 1973.

[12] Franco P. Preparata and Micheal Ian Shamos. *Computational Geometry: An Introduction*. Springer-Verlag, Berlin, Germany, 1985.

[13] M. Shimrat. Algorithm 112: position of point relative to polygon. *Communications of the ACM*, 5:434, 1962.

[14] John M. Snyder and Alan H. Barr. Ray tracing complex models containing surface tessellations. *Computer Graphics (SIGGRAPH '87 Proceedings)*, 21(4):119–128, July 1987.

[15] John Spackman. Simple, fast triangle intersection, part II. *Ray Tracing News*, 6(2), 1993. `ftp://ftp.princeton.edu/pub/Graphics/RTNews`.

[16] Jorge Stolfi. *Oriented Projective Geometry: A Framework for Geometric Computations*. Academic Press, 1991.

[17] Steve Worley and Eric Haines. Bounding areas for ray/polygon intersection. *Ray Tracing News*, 6(1), 1993. `ftp://ftp.princeton.edu/pub/Graphics/RTNews`.

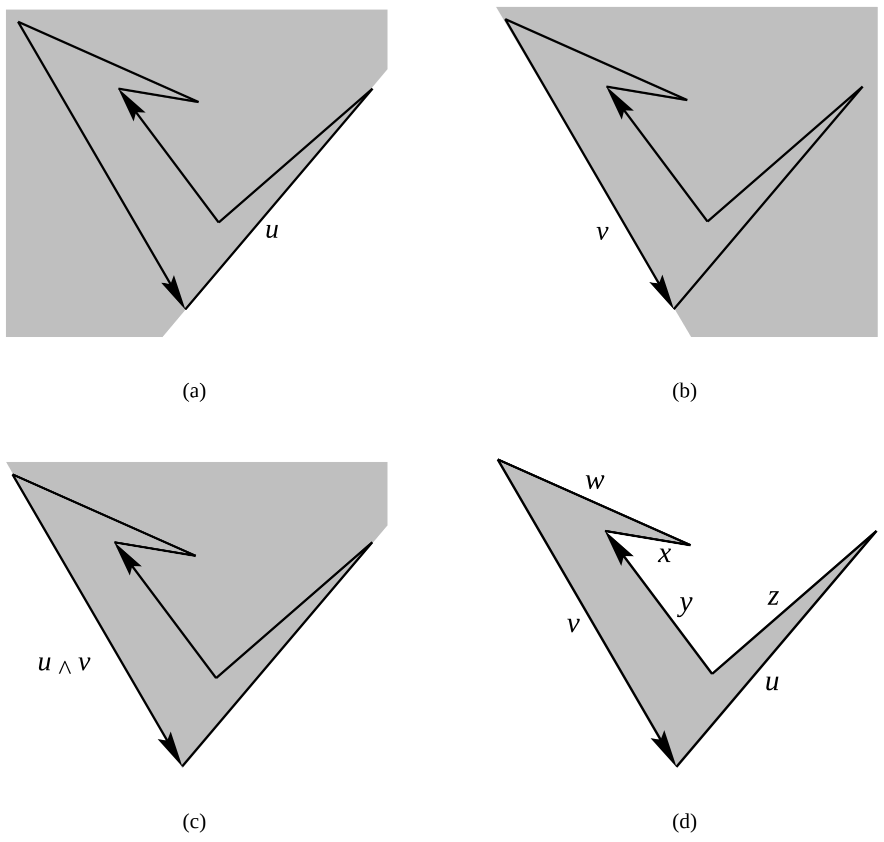
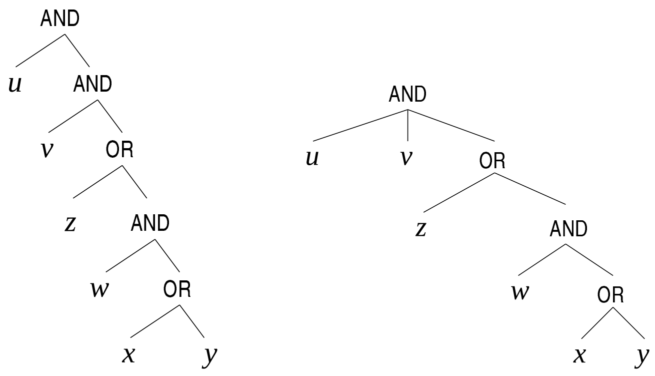


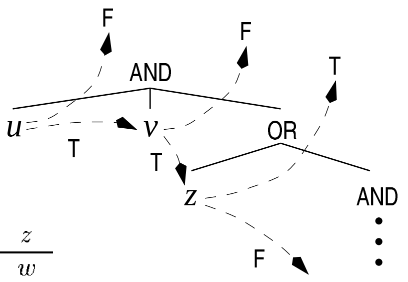
Figure 1: *A polygon yielding the Boolean formula* $u \wedge v \wedge (w \wedge (x \vee y) \vee z)$*: (a) the half-plane defined by the edge* u*; (b) the half-plane defined by the edge* v*; (c) the region of the plane formed by the intersections of these two half-planes, beginning to define the interior of the polygon; and (d) the complete interior of the polygon is defined.*



(a) Binary tree

(b) n-ary tree

Figure 2: *Two CSG representations for the polygon of Figure 1.*



	u	v	w	x	y	z
F	-	-	-	y	-	w
T	v	z	x	-	-	-

Figure 3: *Test paths through the n-ary tree of Figure 2 are represented by an* edge-sequence graph*, shown here as a table. The entire tree does not need to be tested; instead, "short-circuit" paths may be taken when certainty of the location of a point is achieved.*

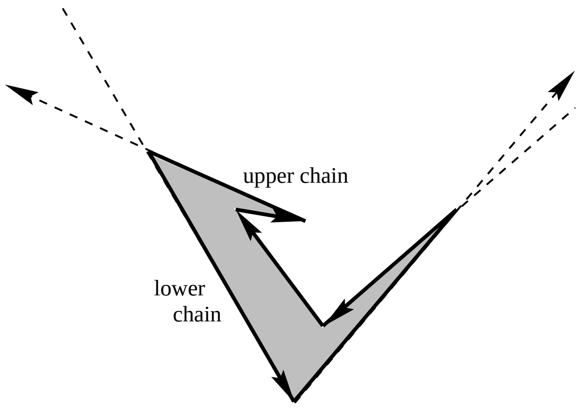


Figure 4: *A polygon split at its left- and rightmost vertices, and the edges adjacent to those vertices extended to infinity.*

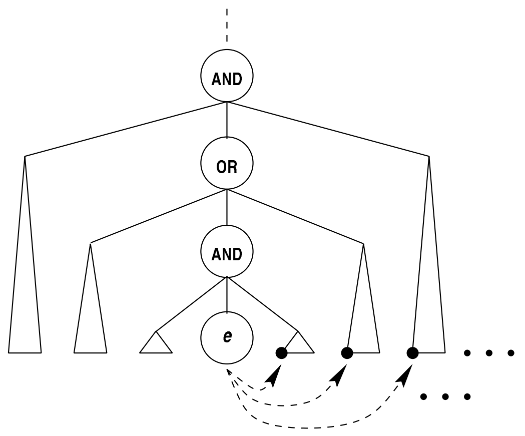


Figure 5: *One of four cases (case FA) in finding the next leaf to test. The first of the leaves (from left to right) indicated by the dashed arrows that is present in the tree is the next to be tested.*

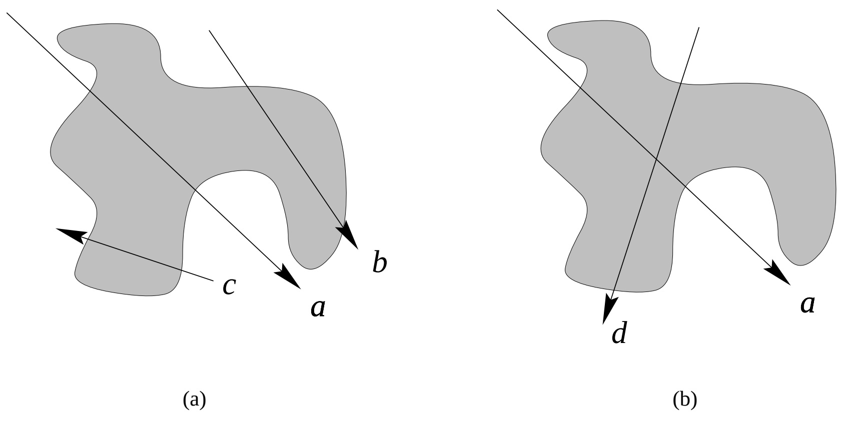


Figure 6: *In case (a), the probability that edge b will be tested immediately after a is the probability that the tested point lies in the region to the left of a; edge c is tested immediately after a with the same probability that the tested point lies in the region to the right of a. In case (b), edge d is immediately tested after a regardless of the outcome of a's test. Due to the linearity of expectation, this is equivalent to the probability that the tested point lies in the region to the left or the region to the right of a.*

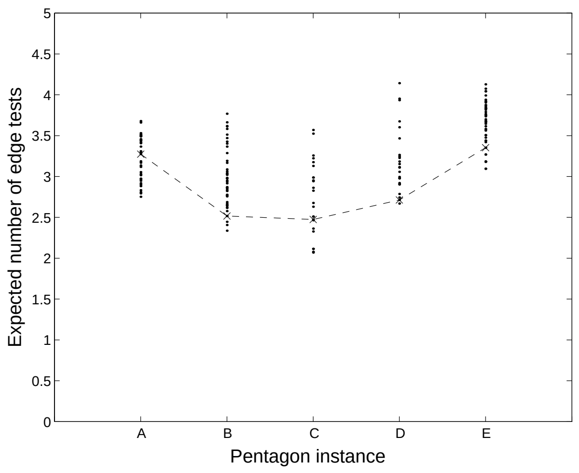


Figure 7: *Analytic performance of the simple heuristic, pentagon instance vs. expected number of edge tests; the cost of the permutations selected by the simple heuristic are connected with a dotted line.*

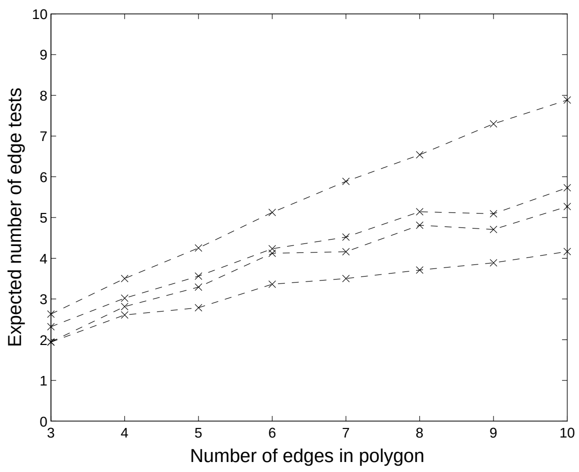


Figure 8: *Analytic performance comparison, polygon size vs. cost metric. The four lines represent (from top to bottom) average maximum cost permutation, average default permutation, average simply-sorted heuristic permutation, and average minimum cost permutation. Data computed on 50 sample polygons used in Table 1 for each of three through ten edges per polygon.*

Method	Variety	3	4	5	6	7	8	9	10	20	50	100	1000
angle sum	-	15.9	20.4	24.9	29.6	34.0	38.6	43.1	47.7	94.5	232.7	461.8	4656.8
barycentric	-	2.0	3.5	4.9	6.2	7.6	8.9	10.3	11.6	24.5	62.8	125.5	1252.1
crossings	-	1.9	1.8	1.9	1.9	2.1	2.2	2.4	2.5	3.9	7.9	14.5	127.5
crossings	multiply	1.3	1.5	1.6	1.8	1.9	2.1	2.3	2.4	4.0	8.3	15.2	132.5
crossings	winding	2.1	2.1	2.2	2.2	2.5	2.7	2.8	2.9	4.8	9.9	18.4	170.2
CSG	-	0.9	1.2	1.3	1.6	1.7	1.8	2.1	2.1	3.6	7.0	11.3	71.7
CSG	sorted	0.8	1.0	1.2	1.5	1.7	1.8	1.9	2.1	3.1	5.7	9.7	41.4
grid	100	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.1	1.3
grid	20	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.2	1.3	1.5	3.1
half-plane	-	0.7	1.2	1.6	2.1	2.5	2.9	3.3	3.8	7.9	20.5	41.8	630.8
half-plane	sorted	0.7	1.1	1.4	1.8	2.2	2.5	3.0	3.3	7.0	17.3	35.0	600.6
Spackman	-	0.9	1.3	1.7	2.1	2.5	2.9	3.3	3.7	7.4	18.5	37.0	676.8
Spackman	sorted	0.8	1.2	1.6	2.0	2.4	2.8	3.2	3.6	7.2	18.1	35.9	540.8
trapezoid	100	1.0	1.1	1.1	1.1	1.1	1.2	1.3	1.3	1.7	2.5	3.4	14.7
trapezoid	20	1.1	1.2	1.3	1.4	1.4	1.5	1.6	1.7	2.3	4.1	6.3	39.9
Weiler	-	2.3	2.4	2.7	2.8	3.1	3.5	3.8	4.0	7.0	16.0	31.1	307.2
Weiler	winding	2.3	2.4	2.7	2.8	3.1	3.4	3.7	4.0	7.1	16.1	30.9	307.4

Table 1: *Average times for point-in-polygon tests (in microseconds), number of edges per polygon vs. method for random polygons.*

Method	Variety	3	4	5	6	7	8	9	10	20	50	100	1000
angle sum	-	15.5	20.2	25.1	29.7	34.5	39.5	44.0	48.5	95.3	235.0	471.0	4669.6
barycentric	-	2.2	3.7	5.0	6.4	7.6	8.8	10.1	11.3	23.5	59.6	120.3	1227.1
crossings	-	1.7	1.6	1.7	1.7	1.8	1.9	2.0	2.1	3.1	6.6	12.7	122.6
crossings	convex	1.7	1.6	1.6	1.6	1.6	1.7	1.8	1.8	2.5	5.0	9.2	86.3
crossings	multiply	1.3	1.4	1.6	1.7	1.8	1.9	2.0	2.1	3.4	7.0	12.8	123.5
crossings	winding	1.9	1.8	1.9	2.1	2.1	2.2	2.4	2.6	4.0	8.6	16.4	161.7
CSG	-	1.0	1.3	1.6	1.9	2.2	2.5	2.8	3.1	6.1	15.1	30.2	380.0
CSG	sorted	0.9	1.3	1.6	1.9	2.2	2.5	2.8	3.0	5.8	14.4	27.5	345.4
exterior	-	0.7	0.9	1.1	1.4	1.6	1.7	1.9	2.2	4.1	9.8	19.7	309.1
exterior	randomized	0.7	1.0	1.2	1.3	1.6	1.7	1.9	2.1	4.0	9.4	18.1	282.2
grid	100	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0	1.0
grid	20	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.1	1.5
half-plane	-	0.8	1.3	1.7	2.2	2.6	3.1	3.5	4.0	8.4	21.7	43.9	721.0
half-plane	convex, hybrid	0.7	1.1	1.4	1.8	2.1	2.5	2.8	3.2	6.5	16.2	33.1	519.0
half-plane	convex, sorted	0.7	0.9	1.1	1.3	1.5	1.6	1.8	2.0	3.7	8.6	17.0	311.1
half-plane	convex, sorted, hybrid	0.7	0.9	1.1	1.2	1.4	1.6	1.7	1.9	3.3	8.1	15.0	294.6
half-plane	sorted	0.7	1.1	1.4	1.6	1.9	2.2	2.5	2.8	5.4	13.5	26.9	550.0
inclusion	-	3.1	3.2	3.3	3.3	3.3	3.4	3.4	3.5	3.7	4.2	4.2	5.3
Spackman	-	0.9	1.4	1.8	2.2	2.5	2.9	3.2	3.6	7.0	17.2	34.3	574.2
Spackman	sorted	0.9	1.3	1.7	2.0	2.4	2.7	3.0	3.3	6.5	16.4	32.4	508.8
trapezoid	100	1.1	1.2	1.2	1.3	1.3	1.3	1.3	1.3	1.4	1.6	1.8	5.0
trapezoid	20	1.2	1.4	1.4	1.5	1.5	1.6	1.6	1.6	1.9	2.7	3.6	19.9
Weiler	-	2.1	2.2	2.4	2.6	2.9	3.1	3.4	3.7	6.5	15.2	29.9	308.6
Weiler	winding	2.1	2.1	2.4	2.6	2.8	3.2	3.4	3.6	6.5	15.4	30.4	305.4

Table 2: *Average times for point-in-polygon tests (in microseconds), number of edges per polygon vs. method for convex polygons only.*