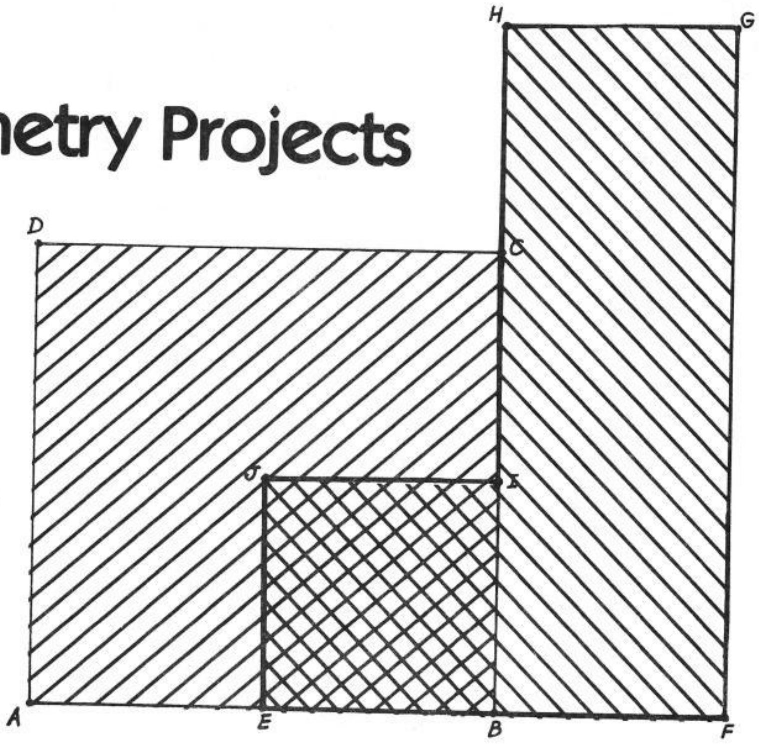
reprinted from IEEE Computer Graphics and applications

Prolog and Geometry Projects



Wm. Randolph Franklin, Peter Y.F. Wu,
Sumitro Samaddar,* and Margaret Nichols*

Rensselaer Polytechnic Institute

Prolog is a useful tool for geometry and graphics implementations because its primitives, such as unification, match the requirements of many geometric algorithms. During the last two years, we have implemented programs to solve several problems in Prolog, including a subset of the Graphical Kernel System, convex-hull calculation, planar graph traversal, recognition of groupings of objects, Boolean combinations of polygons using multiple precision rational numbers, and cartographic map overlay. Certain paradigms or standard forms of geometric programming in Prolog are becoming evident. They include applying a function to every element of a set, executing a procedure so long as a certain geometric pattern exists, and using unification to propagate a transitive function. This article describes the experiences, including paradigms of programming that seem useful, and finally lists what we see as the advantages and disadvantages of Prolog.

The fifth-generation logic programming language Prolog[1,2] appears appropriate for research in geometry and graphics. Examples of its use in architectural design,[3-5] in CAD,[6] and in constructing geometric objects from certain constraints[7] have been discussed.

Prolog has several properties important to geometric and graphics applications. It is a declarative instead of an imperative language, which means, at least in theory, that you specify the formula satisfied by the solution instead of the procedure to execute to find the solution. In practice, for nontrivial programs you must often, for efficiency, be somewhat imperative. Prolog, like pure Lisp, has no destructive assignments, except for modifications of the global database. The unification of logical formulas is a primitive operator. The only data structure is the list. Lists can contain free variables that are later instantiated or unified with other free or bound vari-

An earlier version of this article appeared in *Proc. Graphics Interface 86.*

*Sumitro Samaddar is now with Siemens Research and Technology Laboratories. Margaret Nichols is now with the North American Philips Lighting Co.

44-1

ables. Prolog is often combined with other languages, such as Fortran for numerical computations and graphics, or Lisp. Finally, Prolog is quite nonstandard, even in the basic syntax, and in the names of common procedures, such as "print."

Our area of interest is computational geometry, such as problems of spatial coincidences and intersections with a wide gap between conception and implementation. Problems such as polygon intersection, where an algorithm can be described to another person in about 100 words, can take weeks to design and code.

These problems include many special cases that can consume half the total amount of code, are often artifacts of the data structures used, and do not belong to the intrinsic problem. For example, one algorithm to solve the polygon intersection problem traces around the edges in order. In a special case many edges end at the same vertex. However, this case does not exist explicitly, and need not be considered if a data structure consisting only of a set of the edges is used. The correct handling of these special cases is important when a low-level routine is used to solve a higher level problem, since proper handling of the low-level special cases may automatically solve the higher level special cases, too.

Simple example

For a simple example of Prolog's use in geometry, consider the problem of linking isolated edges to form a chain (see Figure 1). Our only data structure is

```
chain(A, B, L)
```

which represents one chain. Here *A* is the name of the first vertex, *B* the last vertex, and *L* a list of the vertices in the chain from the second to the last. Initially each edge is separate, while eventually only one chain remains (if possible). Thus, in Figure 1, the initial database is

```
chain(a, b, [b]).
chain(d, e, [e]).
chain(c, d, [d]).
chain(b, c, [c]).
```

The random order emphasizes that this is a set of facts whose order does not matter. The final database is

```
chain(a, e, [b, c, d, e]).
```

The complete program to process this database is as follows:

```
1  join1 :-
2      chain(A, B, L1),
3      chain(B, C, L2),
4      retract(chain(A, B, L1)),
5      retract(chain(B, C, L2)),
6      append(L1, L2, L3),
7      assert(chain(A, C, L3) ).
8  join :- join1, fail.
9  join.
```

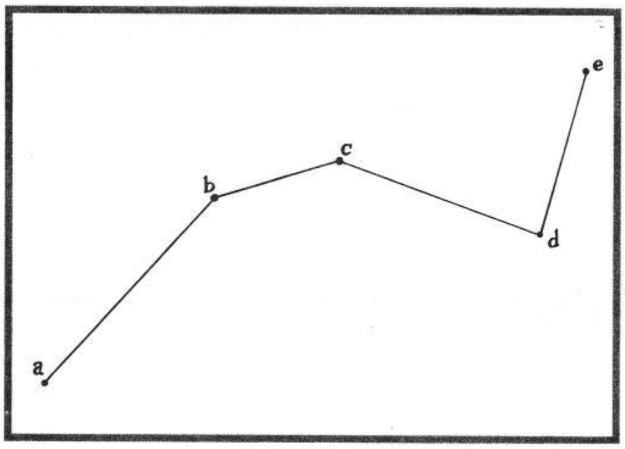


Figure 1. Four edges that will be linked.

The line numbers are not part of the program. Two procedures, *join1* and *join*, make up the program; *join1* searches for two chains that can be combined in lines 2 and 3, deletes them from the database in lines 4 and 5, forms the combined chain in line 6, and adds it to the database in line 7. In more detail, line 2 searches for a fact in the database that matches it or, in other words, is a chain fact. If it finds one, it assigns the three arguments of the fact to variables *A*, *B*, and *L1*, respectively. Then line 3 searches for a chain fact whose first argument matches *B*. If there is no such fact, line 3 fails and forces line 2 to redo, that is, to unbind the variables that it has set, find another fact, and bind the variables again. Superficially, the failure and redoing appear similar to a destructive assignment.

If line 3 does not succeed with any fact that line 2 finds, then the whole procedure *join1* fails and no more chains can be combined.

Procedure *join*, which is called by the user, simply calls *join1* until that fails. The fail in *join* here has the effect of a repeat loop because the *assert* and *retract* in *join1* are not redone when *join1* is.

This nine-line program is much more compact in Prolog than in many other languages. Another style of Prolog programming, instead of modifying the global database, recurses one level deeper each time two chains are joined and passes the modified database to the new call.

44-2

Table 1. Computers and operating systems used to implement Prolog.

Machine	Operating System	Prolog Version
IBM 3081	Michigan Terminal System	York (U.K.)
IBM 4341	CMS	Waterlog
IBM 4341	CMS	VM-Prolog
Prime 750	Primos	Salford
VAX 780	Unix bsd 4.3	UNSW
VAX 780	Unix bsd 4.3	C-Prolog

Implementations

Over the last two years we have implemented several graphics and geometric algorithms in Prolog—totaling a few thousand lines of code—using four different Prolog interpreters on four different computers (see Table 1).

Some of the versions were used for preliminary projects that will not be reported here. The implementations, which vary from warm-up exercises to large problems, include the following:

- Graphical Kernel System subset
- convex hull
- big rational numbers
- polygon intersection
- planar graph traversal
- cartographic map overlay
- photoreconnaissance inference

Graphical Kernel System subset

This graphics addition to Prolog was implemented on an IBM 4341 using Waterlog under the CMS operating system.[8,9] It allowed us to draw lines on the 3270 graphics VDT from a Prolog program. We implemented two classes of lines: permanent and backtrackable. If the Prolog procedure that drew a backtrackable line was backtracked over, then the line would be erased. The procedure used a feature of IBM's Graphics Subroutine Package (GSP).

We found some major problems. Waterlog, like most Prologs, lacks floating-point numbers and even 4-byte integers, a fact which was undocumented. It has, however, the powerful capability to be linked to programs in other languages such as Fortran. Thus we implemented a real number in Prolog as a data structure of the form *real*(*A*, *B*), where *A* and *B* were Prolog integers holding the upper and lower half-words, respectively, of the integer. The user never looked at *A* and *B*, but accessed the real numbers via procedures such as *addreal*(*X*, *Y*, *Z*) and *realtointeger*(*R*, *I*) that took real number structures in the stated form and did the obvious things. For example, our Prolog procedure *addreal* called a Fortran subroutine with six integer arguments that were the two halves of the two inputs and the output. The Fortran subroutine used equivalences to combine the 2-byte integers into 4-byte reals, add them, split the result, and pass it back.

Convex hull

This divide-and-conquer algorithm was implemented in Salford Prolog on the Prime and in Waterlog on the IBM 4341.[10,11] The Salford system allows both real numbers and dynamic linking to Fortran routines. We also tested York Prolog,[12] which is written in Pascal. The York system has the advantage of being portable to any machine that can compile a thousand-line Pascal program using 4-byte integers. Unfortunately, at that time these machines did not include the official Pascal compiler available from Prime. (We have not evaluated third-party Pascal compilers for Prime computers.) We also tested York Prolog on an IBM 3081 running the Michigan Terminal System, but found the other computers' operating systems more flexible and cheaper to use, since York Prolog is very slow.

The convex-hull algorithm proceeds as follows, using a divide-and-conquer paradigm.[13] Duplicate points are removed and then the set of points is split into a left and a right subset based on the points' *x*-coordinates. The convex hulls of these sets are found recursively. Merging the two convex hulls requires the top and bottom tangents or supporting lines. The first approximation to the top tangent is found by joining the top point of the left convex hull to the top point of the right one. Then, if necessary, these endpoints of the tangents are moved right and left until the tangent does not intersect the convex hulls (except at the endpoints).

This algorithm takes time $T = \theta(N \log(N))$. The Prolog code is about 200 lines including comments.

Boolean combinations of polygons

A program to perform operations such as intersection, union, and difference on two planar polygons was implemented on the Prime and IBM 4341.[10] First, a package to perform arithmetic using rational numbers in multiple precision was implemented.[14] Each number, abstractly a quotient of an integer numerator and denominator, is entered as a list of the numerator and denominator. Each numerator and denominator is a list of groups of the number's digits. For example, 123456789/987654321 is represented as [[56789, 1234], [54321, 9876]]. Rational numbers—as part of an ongoing investigation into their utility in geometry and the map overlay problem in cartography[15]—are used to avoid roundoff errors. They are used because they are closed under the operations of intersecting lines; that is, if the

44-3

endpoints of two lines are specified in rationals, then their intersection can also be exactly represented as a rational. The result is no roundoff error.

Although the rationals are not closed under general rotations, they are closed under rotations by *trig-rational* angles: angles with a rational sine and cosine. Trig-rational angles are dense in the reals, so that an arbitrarily accurate trig-rational approximation to any angle can always be found. Unfortunately, 45° is not trig-rational.

The big rational package was designed in several steps, following the ideas that have been available in Macsyma for some years. First, rational numbers are implemented. A rational number Q is stored as the expression N/D. This is upwardly compatible with integers, since *is*, which knows nothing of rationals, thinks it is just an integer expression. Also the rational number prints normally without a separate print procedure. We implement a new infix operator, *isr*, which operates on rationals just as *is* operates on integers. It also converts integers to rationals. Rational versions of all the integer arithmetic operators are also implemented.

Next, a big integer arithmetic package is implemented, along with a new infix operator *are* and big versions of all the operators. Each big integer is stored as a list of groups of digits. For 32-bit built-in integers, each group is four digits. Zero is stored as [ ], one as [1], 72 as [72], 10001 as [1, 1], 2180077 as [80077, 21], minus one as [-1], -123456 as [-56, -1234], and so on.

Then these are combined into one package with the operator *isx*. Now we can say things like

```
X isx ( [3456, 12] + 23) / [222, 3] )
```

The big rational package was tested by calculating π from the following formula, whose simplicity overrides its very slow convergence:

$$\pi = 2 \cdot \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{3} \cdot \frac{4}{5} \cdot \frac{6}{5} \cdot \frac{6}{7} \cdot \frac{8}{7} \cdot \frac{8}{9} \cdot \;\cdots$$

The UNSW Prolog code to execute this is

```
pi( [ ], [2]).   % preset value: pi = 2

step :-
  pi(R, P),
  R1 are R + [1],
  R2 are R1 mod [2],
  P1 isx P*( (R1 + R2)/(R1 - R2 + [1]) ),
  retract(pi(R, P) ),
  asserta(pi(R1, P1) ),
  pb(R1), print(': '), pxq(P1), nl,!.

go :- repeat,step,fail.
```

The output (using a stack size of 50000) is

```
1: 4
2: 8 / 3
3: 32 / 9
...
31: 288230376151711744 / 90324408810638025
```

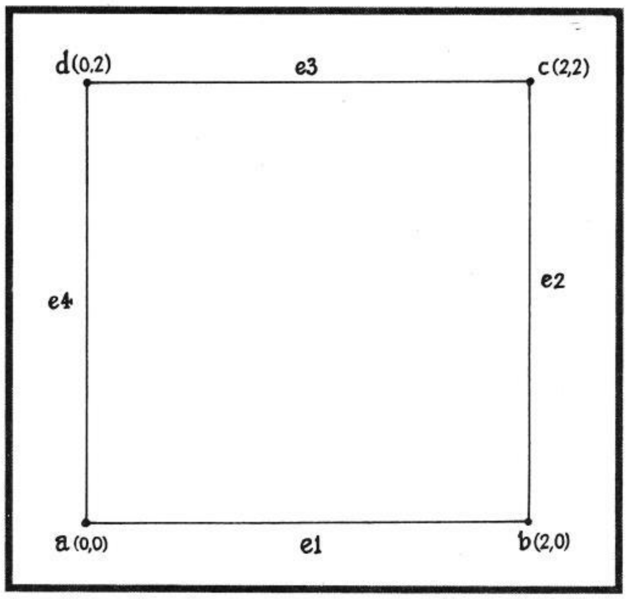


Figure 2. Data structure for a 2 × 2 square.

The polygon combination system uses an edge-based boundary representation. Each polygon is considered a set of edges. Here are the actual data structures:

```
vert(vertexname, x, y)
edge(edge_name, name_of_first_vertex,
     name_of_second_vertex)
edge_eqn(edge_name, a, b, c)
poly(polygon_name, edge_name, which_side)
```

The edge equation is of the form $ax + by + c = 0$. There is one *poly* fact for each edge of each polygon. Since a given edge may be used by more than one polygon, it is necessary to know which side of the edge is the inside of this particular polygon. Legal values are *left* and *right*. The facts for the polygon in Figure 2 are

```
vert(a, 0, 0).
vert(b, 2, 0).
vert(c, 2, 2).
vert(d, 0, 2).
edge(e1, a, b).
edge(e2, b, c).
edge(e3, c, d).
edge(e4, d, a).
edge_eqn(e1, 0, 1, 0).
edge_eqn(e2, 1, 0, -2).
edge_eqn(e3, 0, 1, -2).
```

44-4

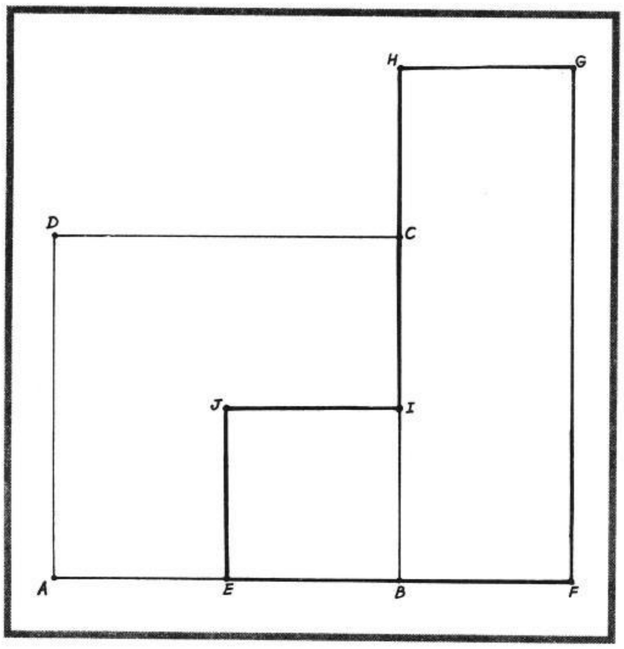


Figure 3. Combining polygons *ABCD* and *EFGHIJ*.

```
edge_eqn(e4, 1, 0, 0).
poly(a, e1, left).
poly(a, e2, left).
poly(a, e3, left).
poly(a, e4, left).
```

With this data structure, special cases involving multiple edges all ending at the same vertex are not a problem; in fact, the algorithm never knows that they exist. The data structure also does not store any global topology, such as the number of connected components. The information and the containment relationships could be calculated if needed, but are in fact never necessary.

The first stage of the algorithm is basically a forward-reasoning system. It searches for cases where two edges intersect. Whenever such a case is found, the two edges are deleted, and three or four new edges created. There will be three new edges if one edge's endpoint falls on another edge or if two edges are collinear. The process continues until no edges intersect, except possibly at both their endpoints.

This process is a little more complicated than it appears since we are modifying the list of *edge* facts as we iterate through it. Here different versions of Prolog behave differently. One solution is as follows:

1. Handle deletions not by actually retracting the edge, but by asserting a *deleted(edge_name)* fact to record the information.
2. Initially consider all edges to be of level 0.
3. Compare all the edges pair by pair. Whenever an intersection is found between two edges that do not have an associated deleted fact, assert a deleted fact about both of them and create three or four new edges by asserting level 1 edges.
4. Compare all the level 1 edges against each other and against all the level 0 edges without deleted facts. If any intersect, assert new level 2 edges and deleted facts about the intersecting edges.
5. Compare all the level 2 edges with each other and against all the level 0 and 1 edges.
6. Repeat this until no new intersections are found.
7. Clean up the database.

The procedure should be portable: it does not modify any particular fact because control is iterating through instances of that fact.

Next, in the Boolean combination, each edge is classified into one of six categories:

- an edge on polygon A that is inside polygon B
- an edge on polygon A outside polygon B
- an edge on polygon B inside polygon A
- an edge on polygon B outside polygon A
- an edge on both polygons A and B, and both polygons are on the same side of it
- an edge on both polygons, and they are on opposite sides of it

Finally, a subset of the edges is selected, depending on the particular result desired. For example, in a union, edges on either polygon that are outside the other polygon, plus edges on both polygons with both polygons on the same side, are needed. Since this selection takes almost no time, all the Boolean combinations are found at no extra cost. In Figure 3, polygon A is *ABCD* and polygon B is *EFGHIJ*. After intersecting edges are cut, edges *AB* and *EF* are cut into *AE*, *EB*, and *BF*; *HI* is cut into *HC* and *CI*. When the resulting edges are classified, edge *AB* is on polygon A outside of B. Edge *EJ* is on B inside A. Edge *EB* is on both polygons A and B, and they are on the same side. In contrast, edge *CI* is on both polygons, but they are on opposite sides.

Rational arithmetic is only an attempted solution to numerical inaccuracies. The problem is that when two rationals are added or otherwise combined, the result usually has twice the number of digits of either operand. The high number is burdensome if the computation graph is deep. Another possible solution would be to develop robust algorithms that allow small errors while preserving critical aspects of the problem's topology. By analogy, numerical operations on large matrices are designed to pre-

44-5

serve important properties, such as relations among the eigenvalues, while allowing the actual coefficients to wander somewhat. However, such a solution would be difficult here, since the topology of a planar graph is much richer than that of a matrix.

Planar graph traversal

At some point in an object-space hidden-surface algorithm,[16] we have the set of the visible edges and must join them to find the visible polygons. Joining requires a planar graph traversal, sometimes called a tessellation. In Figure 4, we are given the vertices and edges in the form

```
vert(vert-name, x-coord, y-coord)
edge(edge-name, first-vertex, second-vertex, angle)
```

for example,

```
vert(v1, 0, 0).
edge(e1, v1, v2, 0).
```

The angle of the edge is supplied because of the difficulty of computing arctangents using only integers. The output is a set of facts of the form

```
polygon([v1, v2, v3, v4]).
```

This was implemented in UNSW Prolog[17] on a VAX.

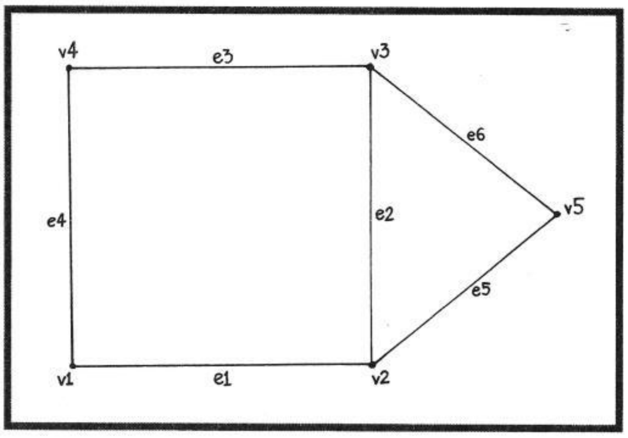


Figure 4. Finding the faces of a planar graph.

Cartographic map overlay

Cartographic map overlay is the process of superimposing two maps. The map overlay system was implemented on the Prime in Salford Prolog.[11,18]

A map is a 2D spatial data structure consisting of a set of chains, each of which is a polyline on the 2D plane. Each chain begins at a node and ends at a node (which may or may not be the same node). A chain does not intersect itself, nor any other chains in the same map. Hence the set of chains and nodes partition the 2D plane into regions, each enclosed by an alternating sequence of nodes and chains. Each region is called a polygon. The spatial structure of nodes, chains, and polygons forms a map. Since we require that each polygon does not contain any other polygon, we deal only with the maps having no separate components.

The algorithm is a combination and extension of the algorithms for polygon intersection and planar graph traversal. The major difference is that instead of edges, the algorithm deals with chains or polylines. Here is a brief overview of the algorithm:

1. Split intersecting chains. Search for intersecting chains and split the chains at the intersection points.
2. Compute node incidences. Identify the incident nodes for each chain and compute a measure of the incident angle.
3. Sort incidences and generate chain linkages. Sort the incident chains at each node into proper cyclic order. Associate each pair of adjacent chains (in reverse cyclic order) to form a linkage record. Hence each linkage record identifies a corner of a polygon.
4. Link up the polygon boundaries. Connect the linkage records into proper cyclic order, determining all the polygons.
5. Mark adjacent polygons at each chain. Identify for each chain the polygons on its left and right.

As we describe further details of the algorithm, we will define the data structures involved.

The input map is a set of chains. Each chain is represented by a Prolog fact, in the following format:

```
chain(chain_id,
        tail_node, head_node,
        [[X1, Y1], ..., [X2, Y2]],
        left_polygon, right_polygon)
```

Each chain has a unique *chain_id; tail_node* and *head_node* identify the nodes at the beginning and end of the chain. The list [[*X*, *Y*], ...] of coordinates describes the polyline. The polygons to the left and right of the chain are also identified.

In step 1, we search for intersecting chains and split the chains at the intersection point, which is a new node in the output map. The difficulty of simultaneous iteration and updating in the database arises in the same way as in polygon intersection, and we resolve it the same way.

In step 2, we examine each chain and declare the endpoints as nodes. Each node is represented by the following:

44-6

node(node_id, [*X*, *Y*])

For each chain-to-node incidence, we make an incidence record as a Prolog fact:

incidence(node_id, chain_incidence,
incident_angle)

The *chain_incidence* may be *h(chain_id)* or *t(chain_id)*, indicating the head or tail of the chain, respectively.

In step 3, we sort the incidence records at each node by the *incident_angle*. Hence we have the incident chains at each node in proper cyclic order. Every pair of adjacent chains in the cyclic list therefore determines a corner of a polygon. For every corner of a polygon, we form a linkage record as a Prolog fact, identifying every pair of adjacent chains around each polygon:

linkage([boundary_1, boundary_2])

Each *boundary* in the pair of adjacent chains can be either *th(chain_id)* or *ht(chain_id)* to indicate, respectively, that the boundary chain goes in the tail-to-head or head-to-tail direction.

Prolog has the same high-level advantages as Lisp, for example, the equivalence of code and data, and dynamic data allocation.

In step 4, linkage records are connected to trace out polygons. The process is similar to the one in "Simple example"—linking isolated edges to form a chain. A linkage record that begins and ends with the same boundary chain (in the same direction) identifies a polygon to be declared as such in the output map:

polygon(polygon_id, [boundary_1,
boundary_2, ...])

In step 5, we examine the boundary chains of each polygon, and form the following left and right facts:

```
left(chain_id, polygon_id)
  % if boundary has th(chain_id)
right(chain_id, polygon_id)
  % if boundary has ht(chain_id)
```

We can then update the chain records with the polygon identifiers on the left and right. We have the chains in the desired output format:

```
chain(chain_id,
      tail_node, head_node,
      [[X1, Y1], ..., [X2, Y2]],
      left_polygon, right_polygon).
```

Prolog provides a relational approach in data structuring. Geometric entities are defined as Prolog facts, and Prolog rules implement geometry algorithms for data processing. Each step is a sequence of set-based operations. Explicit directives, namely *assert* and *retract*, add and delete entries to and from the database. Calculations are done using big rational arithmetic. Incident angles are strictly trig-rational, guaranteeing numerical accuracy and thus topological consistency in the computation.

Photoreconnaissance inference

Now we wish to infer which units of an army organization are present after seeing, via photoreconnaissance, an incomplete picture of their equipment.[19] The army organization, parts of which may be present in the photo, is described with Prolog facts such as the following:

child(Father, Son, Number)

This says that unit *Father* ideally contains *Number* of the subunit *Son*. For example, parts of a Soviet motorized rifle division might be defined thus:

```
child(motorized_rifle_division, btr_regiment, 2)
child(motorized_rifle_division, bmp_regiment, 1)
child(motorized_rifle_division, tank_regiment, 1)
child(motorized_rifle_division,
  artillery_regiment, 1)
child(bmp_regiment, bmp_battalion, 3)
child(bmp_battalion, bmp_company, 3)
child(bmp_company, bmp_platoon, 3)
```

Each unit's equipment is described by the following form of fact:

eqpmnt_overall(Unit, Ename, Number)

Unit is the name of the unit that has the equipment, such as *art_reg* for an artillery regiment. *Ename* is the name of the equipment, such as *sa_6* for an SA-6 antiaircraft missile. *Number* is the maximum number of pieces of equipment that the unit can have. The fact for a particular unit includes only equipment that the unit possesses directly, not a subunit's equipment. Some sample facts are

```
eqpmnt_overall(art_reg, sa_6, 20)
eqpmnt_overall(mr_div, amphi_brdm, 48)
```

Then facts defining what equipment has been recognized are stated as follows:

equipment(Name, Number)

For example,

```
equipment(sa_7, 7)
equipment(rpg_7, 23)
```

44-7

Given this information, the inference engine reports that

> Based on that, my first guess about the unit present, and the remaining equipment associated with it, is
>
> ```
> Remaining = [[arm_per_car_btr, 38],
>              [mortar_120mm_1943, 6]]
> Unit = mot_rif_btln_btr
> ```

This inference engine was designed to be part of a larger blackboard format system where a low-level image-interpretation and geometry engine makes a first guess about the objects present and passes the information up to this unit. The output of this unit can be used to bias the prior probabilities of the geometry system as it continues to look.

The system is robust: it automatically handles cases when the unit on the ground is understrength and its image-interpretation system is unable to find everything.

Strengths and weaknesses of Prolog

Advantages and disadvantages of Prolog for graphics and geometric applications have become evident from the above implementations.

Advantages of Prolog

Prolog has the same high-level advantages as Lisp, for example, the equivalence of code and data, and dynamic data allocation. Moreover, Prolog has specific advantages. Unification makes determining graph connectivity a primitive operation and is, in general, useful for propagating transitive properties that occur frequently, such as graph connectivity. This is a counterexample to the proposition that "Unification is what you do when you don't know what you are doing."

The pattern matching fits with the form of expression of many algorithms. For example, our polygon combination algorithm proceeds as follows. Whenever the pattern of two edges intersecting or one edge ending on the interior of another edge occurs, we retract those edges and assert new smaller edges. When this pattern no longer exists, we have a superset of the edges in the output polygon.

Although many of the above features could be implemented in any language that is Turing equivalent, Prolog is somewhat standard. Therefore, different researchers can understand and use each other's extensions.

Disadvantages of Prolog

There are, however, some problems with using Prolog for geometry.

Because Prolog lacks nesting in the program and databases, software engineering problems arise when using it for a large project. Many geometry algorithms are more natural to a forward-reasoning system than a backward-reasoning system; that is, we are more likely to want the output from some given input rather than the reverse.

The natural way of expressing pattern-matching algorithms requires us to modify a database as we are searching through it. Thus, in polygon overlay, whenever we find the pattern of two edges crossing, we retract them and assert four new edges. Backtracking and redoing a database that we are modifying does not work on all Prologs. Furthermore, Prolog does not support coroutines, which are a natural way to express many algorithms.

In general, Prolog is completely unstandardized around the fringes, as some tests of the cut procedure show.[20]

Paradigms of programming

Certain techniques have proven to be generally useful in our implementations, and may be useful to others also. They include the following paradigms.

Set-based algorithms

Many algorithms such as polydedron intersection and hidden-surface algorithms[16,21] are the alternation of two steps:

- Applying function to every element of a set
- Combining all the elements having a common key

Clearly this is easy in Prolog.

Pattern matching

The second paradigm uses pattern matching to propagate certain properties. For example, in the planar graph traversal algorithm, the edges around each vertex are found and sorted by the angle at which they leave the vertex. Then the edges around each vertex are paired to form corners. These corners can be considered fragments of the output polygons. Whenever two fragments exist such that the last edge of one is the same as the first edge of another, these two fragments are retracted and a single, longer fragment asserted. When such a pattern no longer exists, we have the output polygons.

Unification

Frequently we wish to determine the closure of some transitive property, as when we are given a set of graph edges *edge*(u, v) and wish to determine the connected components. We have implemented the following short algorithm that uses unification and the set-processing paradigm:

44-8

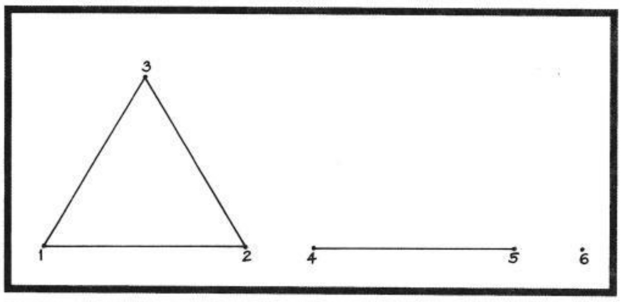


Figure 5. Determining graph connectivity.

- Create a property list (*plist*) with one record per vertex, and the property of each vertex a free variable. For example, in Figure 5 we would have [[1,_], [2,_], [3,_], [4,_], [5,_], [6,_]].
- Process the set of edges and for each edge unify the free variable properties of the endpoints. After this we will have [[1,_1], [2,_1], [3,_1], [4,_2], [5,_2], [6,_3]] with one unique free variable per graph component.
- Bind a name identifying each component to the free variables in the list to give something like [[1, first], [2, first], [3, first], [4, second], [5, second], [6, third]].

A longer example of a simple hidden-surface algorithm would go as follows:

- Wherever the pattern of the projections of two edges intersecting occurs, split the edges into four smaller edge segments.
- For each edge segment find the set of faces hiding its midpoint. If it is empty, the edge segment is visible. Draw the edge.
- Use a planar graph traversal algorithm such as described above to link the visible edges into polygons.
- For each polygon, find a point inside it and then find the set of faces whose projections contain the projection of that point. Find the closest such face—the polygon came from it. Color the polygon accordingly.

This example illustrates all of the paradigms operating together.

Summary

Although not perfect, Prolog is a powerful tool for expressing graphics and geometry algorithms in a concise and natural format. This expressibility allows larger problems to be solved with a given amount of the designer's time, and raises the size of the largest problem that can be solved before the details become overwhelming. ■

Acknowledgments

Different parts of this work were supported by the National Science Foundation under PYI grant no. ECS-8351942, the Data Systems Division of IBM, and the Rome Air Development Center under the postdoctoral development program via Syracuse University.

Franklin's access to Prolog while at Berkeley was sponsored by the Defense Advance Research Projects Agency (DoD) Arpa order no. 4871 monitored by Naval Electronic Systems Command under contract no. N00039-84-C-0089. The VM-Prolog at RPI was made available by the Center for Interactive Computer Graphics.

References

1. W.F. Clocksin and C.S. Mellish, *Programming in Prolog*, Springer-Verlag, New York, 1981.
2. H. Coelho, J.C. Cotta, and L.M. Pereira, *How to Solve It with Prolog*, Ministerio da Habitacao e Obras Publicas, Labatorio Nacional de Engenharia Civil, Lisboa, 1980.
3. P.S.G. Swinson, "Logic Programming: A Computing Tool for the Architect of the Future," *Computer Aided Design*, Vol. 14, No. 2, Mar. 1982, pp.97-104.
4. P.S.G. Swinson, F.C.N. Pereira, and A. Bijl, "A Fact Dependency System for the Logic Programmer," *Computer Aided Design*, Vol. 15, No. 4, pp.235-243.
5. P.S.G. Swinson, "Prolog: A Prelude to a New Generation of CAAD," *Computer Aided Design*, Vol. 15, No. 6, Nov. 1983, pp.335-343.
6. J.C. Gonzalez, M.H. Williams, and I.E. Aitchison, "Evaluation of the Effectiveness of Prolog for a CAD Application," *IEEE CG&A*, Vol. 4, No. 3, Mar. 1984, pp.67-75.
7. Beat Bruderlin, "Using Prolog for Constructing Geometric Objects Defined by Constraints," *Proc. Eurocal 85*, Linz, Austria, 1985.
8. Margaret Nichols, "The Graphical Kernel System in Prolog," master's thesis, Rensselaer Polytechnic Institute, Troy, N.Y., 1985.
9. Grant Roberts, *Waterloo Core Prolog User's Manual (Version 1.5)*, Intralogic Inc., Waterloo, Ont., Canada, 1984.
10. Wm. R. Franklin and P.Y.F. Wu, "Convex Hull and Polygon Intersection Implemented in Prolog," SIAM Conf. Geometric Modeling and Robotics, Albany, N.Y., July 1985.
11. Univ. of Salford, *LISP/PROLOG Reference Manual*, Mar. 1984.

44-9

12. J.M. Spivey, *Univ. of York Portable Prolog System (Release 1) User's Guide*, York, U.K., 1983.

13. F.P. Preparata and S.J. Hong, "Convex Hulls of Finite Sets of Points in Two and Three Dimensions," *Comm. ACM*, Vol. 2, No. 20, Feb. 1977, pp.87-93.

14. P.Y.F. Wu, "Two Arithmetic Packages in Prolog: Infinite Precision Fixed Point and Exact Rational Numbers," Tech. Report TR-082, Rensselaer Polytechnic Institute, Troy, N.Y., 1986.

15. Wm. R. Franklin, "Cartographic Errors Symptomatic of Underlying Algebra Problems," *Proc. Int'l Symp. on Spatial Data Handling*, Vol. 1, Zurich, Switzerland, Aug. 1984, pp. 190-208.

16. Wm. R. Franklin, "A Linear Time Exact Hidden Surface Algorithm," *Computer Graphics* (Proc. SIGGRAPH 80), Vol. 14, No. 3, July 1980, pp.117-123.

17. Claude Sammut, *UNSW Prolog User Manual*, Univ. of New South Wales, Australia, 1983.

18. Wm. R. Franklin, "A Simplified Map Overlay Algorithm," Harvard Computer Graphics Conf., Cambridge, Mass., July 31-Aug. 4, 1983.

19. Sumitro Samaddar, "An Expert System for Photo Interpretation," master's thesis, Rensselaer Polytechnic Institute, Troy, N.Y., 1985.

20. Chris Moss and Earl Fogel, "Tests to Distinguish Various Implementations of Cut in Prolog," Imperial College and Logicware Inc., reported on Usenet June 1985.

21. Wm. R. Franklin, "Efficient Polyhedron Intersection and Union," *Proc. Graphics Interface 82*, Toronto, May 1982, pp.73-80.

Rensselaer Polytechnic Institute Troy, New York 12180-3590

Wm. Randolph Franklin is an associate professor with the Electrical, Systems, and Computer Engineering Dept. at Rensselaer Polytechnic Institute. During 1985-86, when some of this work was performed, he was on sabbatical at the University of California at Berkeley. His research interests include graphics, geometry algorithms, and artificial intelligence. Franklin received the BSc in computer science from the University of Toronto in 1973, and the AM and PhD in applied mathematics (computer science) from Harvard in 1975 and 1978. He is a member of IEEE, ACM, and SIAM.

Franklin's electronic address is FRANKLIN@CSV.RPI.EDU.

Sumitro Samaddar is a research scientist with the robotics group in the Siemens Research and Technology Laboratories. He received his BTech in electrical engineering from the Indian Institute of Technology and MS in computer and systems engineering from Rensselaer Polytechnic Institute. His interests include computer graphics and knowledge-based systems. Now he is involved with the development of graphics tools and interfaces in an integrated robot programming environment for the design, simulation, and off-line programming of automated factory cells. Samaddar is a member of the IEEE Computer Society.

Peter Y.F. Wu is a PhD candidate in the Electrical, Systems, and Computer Engineering Dept. at Rensselaer Polytechnic Institute. He received his BS in electrical engineering from the University of Rochester in 1979, and MS in computer and systems engineering from Rensselaer Polytechnic Institute in 1983. His research interests include geometry, computer graphics, and the software technologies involved. He is a member of IEEE and ACM.

Margaret Nichols is a programmer with North American Philips Lighting Co. She obtained her BS from State University of New York at Binghamton and MS in computer and systems engineering from Rensselaer Polytechnic Institute. Her master's thesis was about the development of GKS in Prolog. Her interests include artificial intelligence and computer graphics. She is a member of ACM.

Franklin and Wu can be contacted at the ECSE Dept., Rensselaer Polytechnic Institute, Troy, NY 12180.

44-10