EXPERIENCES WITH USING PROLOG FOR GEOMETRY

Wm. Randolph Franklin
Computer Science Division, 543 Evans
University of California, Berkeley, CA, 94720, USA.
(Arpanet: wrf@Berkeley.EDU)
and
Rensselaer Polytechnic Institute
Troy, NY, 12180, USA
(wrf%RPI-MTS.Mailnet@MIT-Multics.ARPA)
Margaret Nichols
North American Philips Lighting Co., Bloomfield NJ, USA
Sumitro Samaddar
Peter Wu
Rensselaer Polytechnic Institute
Troy, NY, 12180, USA
(Peter_Wu%RPI-MTS.Mailnet@MIT-Multics.ARPA)

ABSTRACT

The Prolog language is a useful tool for geometric and
graphics implementations because its primitives, such
as unification, match the requirements of many
geometric algorithms. We have implemented several
problems in Prolog including a subset of the Graphics
Kernel Standard, convex hull finding, planar graph
traversal, recognizing groupings of objects, and boolean
combinations of polygons using multiple precision
rational numbers. Certain paradigms, or standard
forms, of geometric programming in Prolog are becom­
ing 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. Certain strengths
and weaknesses of Prolog for these applications are
now apparent.

KEYWORDS: Prolog, Geometry, Graphics Kernel
Standard

INTRODUCTION

The fifth generation logic programming language
Prolog(Clocksin81a, Coelho80a), appears appropriate
for research in geometry and graphics. Some examples
of its use in architectural design are given in
[Swinson82a, Swinson83a, Swinson83b]. Its use in
CAD has been evaluated in [Gonzalez84a]. Construct­
ing geometric objects from certain constraints is
described in [Brüderlin85a]. Over the past two years,
the authors of this present paper have implemented
several geometric and graphic problems in Prolog using
assorted machines. This paper describes the experi­
ences, including some paradigms of programming that
have appeared useful, and finally listing the advantages
and disadvantages of Prolog that we have experienced.

Over the last two years we have implemented
several graphics and geometric algorithms in Prolog,
totally a few thousand lines of code, using four
different Prolog interpreters on four different comput­
ers. The systems include:

<table>
<thead>
<tr>
<th>Machine</th>
<th>Operating System</th>
<th>Prolog Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>IBM 3081</td>
<td>Michigan Terminal System</td>
<td>York (U.K.)</td>
</tr>
<tr>
<td>IBM 4341</td>
<td>CMS</td>
<td>Waterlog</td>
</tr>
<tr>
<td>Prime 750</td>
<td>Primos</td>
<td>Salford</td>
</tr>
<tr>
<td>VAX 780</td>
<td>Unix bsd 4.3</td>
<td>UNSW</td>
</tr>
</tbody>
</table>

This work was supported by the National Science
Foundation under grant no. ECS-8351942, the Data
Systems Division of the International Business
Machines Corporation, and by the Rome Air Develop­
ment Center under the postdoctoral development pro­
gram via Syracuse University.
The implementations include:

- Graphics Kernel Standard subset
- Convex Hull
- Planar Graph Traversal
- Big Rational Numbers
- Polygon Intersection
- Organization Inference

They will now be described in detail.

IMPLEMENTATIONS

Graphics Kernel Standard Subset

This graphics addition to Prolog was implemented by Nichols [Nichols85a] on an IBM 4341 using Waterlog [Roberts84a], under the CMS operating system. This allowed us to draw lines and so on on the 3270 graphics CRT 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. This used a feature of the graphics package GSP.

The major problems were as follows. Waterlog, like most Prologs, lacks floating point numbers, and even four byte integers. (The latter was undocumented; large integers just didn't work.) However it has 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 are Prolog integers holding the upper and lower halfword, respectively, of the integer. The user never looks at A and B, but accesses the real numbers via procedures such as addreal(X, Y, Z) and realtointeger(R, I) that took real numbers in the stated form and did the obvious things.

Convex Hull

This Graham Scan algorithm was implemented by Wu [Franklin85a] on both the IBM 4341, and on the Prime in Salford Prolog [Salford84a]. The Salford system allows both real numbers and dynamic linking to machine that can compile a thousand line Pascal program. We also tested York Prolog on an IBM 3081 running the Michigan Compiler, but found the other computers more flexible and cheaper to use.

The terminal system, but found the other computers

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 by Franklin and Wu [Franklin85a] on the Prime and IBM 4341, and by Franklin and Wu [Franklin85a] on the Prime and IBM 4341. The algorithm was by Franklin [Franklin85a]. Wu first implemented a package to perform arithmetic using rational numbers in multiple precision. Each number, in life a quotient of an integer numerator and denominator, is implemented as a list of the numerator and denominator. Each of them is a list of groups of the digits of the number. For example, 123456789/987654321 is represented as [[56789, 1234], [54321, 9876]]. Rational numbers are used to avoid roundoff errors, as part of an ongoing investigation into their utility in geometry and the map overlay problem in cartography [Franklin84a].

The big rational package was designed in several steps as follows. First, rational numbers were implemented. A rational number Q is stored as the expression N/D. This is upward compatible with integers since is, which knows nothing of rationals, thinks it is just an integer expression. This also means that the rational number prints normally without a separate print procedure. We implemented 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 were also implemented.

Next, a big integer arithmetic package was 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 4 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 were combined into one package with the operator isx. Now we can say things like

\[ X \text{ isx } [[3456,12] + 23] / [222,3]. \]

The big rational package was tested by calculating \( \pi \) from the following formula, whose simplicity overrides its very slow convergence.

\[
\pi = 2 \cdot \frac{2}{1} \cdot \frac{2}{3} \cdot \frac{4}{5} \cdot \frac{4}{7} \cdot \frac{6}{8} \cdot \frac{6}{9} \cdot \frac{8}{11} \cdot \frac{8}{13} \cdots
\]

The UNSW Prolog code to execute this is:

\[ \text{pi([], 2]). } \]

% preset value: Pi = 2

Graphics Interface '86

Vision Interface '86
step :-
  pi(R,P),
  R1 are R+{1},
  R2 are R1 mod {2},
  P1 is x P*((R1+R2)/(R1-R2+{1})),
  retract(pi(R,P)),
  asserta(pi(R1,P1)),
  pb(R1), prin(\), pxq(P1), nl!,

  go :- repeat, step, fail.

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(vertex_name, 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.

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 of their existence. This data structure also does not store any global topology, such as the number of connected components, and which are inside which other. This information, which could be calculated if needed, is 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 this is found, those two edges are deleted, and three or four new edges are created. There will be three new edges if one edge's endpoint falls on another edge. This includes the case where the two edges are collinear. This process continues until no edges intersect, except possibly at both their endpoints.

This process is a little more complicated than appears since we are modifying the list of edge facts as we are iterating through it. This is one of the areas where 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, then
   a) assert a deleted fact about both of them, and
   b) create three or four new edges by asserting level 1 edges.
4. Then 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. Then compare all the level 2 edges against each other and against all the level 0 and 1 edges.
6. Repeat this until no new intersections are found.
7. Finally clean up the database.

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

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

- an edge of polygon A that is inside polygon B,
- on A outside B,
- on B inside A,
- on B outside A,
- an edge that is on both polygons A and B, and both polygons are on the same side of it, and
- 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 on the same side, are needed. Since this selection takes almost no time, all the boolean combinations are found at no extra cost. For example, see figure 1, where polygon A is \(ABCD\) and polygon B is \(EFGHIJ\). After intersecting edges are cut, edges \(AB\) and \(EF\) are cut into \(AB, EB,\) 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.

![Figure 1: Combining Polygons A B C D and E F G H I J](image-url)
Planar Graph Traversal

At some point during an object space hidden surface algorithm [FranklinSOa], we have the set of the visible edges and must join them to find the visible polygons. This requires a planar graph traversal, sometimes called a tessellation. For example, in figure 2,

![Diagram of a planar graph]

we are given the vertices and edges in the form

- \text{vert}(\text{vert-name, x-coord, y-coord})
- \text{edge}(\text{edge-name, first vertex, second vertex, angle})

for example

- \text{vert}(v1, 0, 0)
- \text{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

- \text{polygon}([v1, v2, v3, v4])

This was implemented in UNSW Prolog [SammutS3a] on a Vax.

Organization Inference

In this work, described in more detail in [SamaddarS5a], we wish to infer which units of an army organization are present after seeing, via photoreconnaissance, an incomplete picture of the equipment they possess. The army organization, parts of which may be present in the photo, is described with Prolog facts such as the following.

- \text{child}(\text{Father, Son, Number})

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

- \text{child(motorized_rifle_division,btr_regiment,2)}
- \text{child(motorized_rifle_division,bmp_regiment,1)}
- \text{child(motorized_rifle_division,tank_regiment,1)}
- \text{child(motorized_rifle_division,artillery_regiment,1)}
- \text{child(bmp_regiment,bmp_battalion,3)}
- \text{child(bmp_battalion,bmp_company,3)}
- \text{child(bmp_company,bmp_platoon,3)}

The equipment that each unit possesses is described by the following form of fact:

- \text{eqpmnt_overall(Unit, Ename, Number)}

Unit is the name of the unit that owns the equipment, such as \text{art_reg} for an artillery regiment. Ename is the name of the equipment, such as \text{sa-6} for an SA-6 anti-aircraft missile. Number is the maximum number of pieces of equipment that that unit can own. The fact for a particular unit includes only equipment that the unit owns directly, and not equipment owned by a subunit. Some sample facts are:

- \text{eqpmnt_overall(art_reg, sa_6, 20)}
- \text{eqpmnt_overall(mr_div, amphi_brdm, 48)}

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

- \text{equipment(Name, Number)}

For example,

- \text{equipment(sa_7, 7)}
- \text{equipment(rpg_7, 23)}

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:

- \text{Remaining = \{[arm_per_car_btr, 38], [mortar_120mm_1943, 6]\}}

- \text{Unit = mot_rif_btl_btr}

This inference engine is 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.

This system is robust since it automatically handles the cases of the unit on the ground being under strength, and the image interpretation system not finding everything.

STRENGTHS AND WEAKNESSES OF PROLOG

Certain advantages and disadvantages of Prolog for graphics and geometric applications are becoming evident from these implementations.

Advantages Of Prolog

- Prolog has same high level advantages of Lisp, as the equivalence of code and data and dynamic data allocation.
- There are the specific advantages of Prolog. Unification makes determining graph connectivity a primitive operation and in general is useful for propagating transitive properties such as graph connectivity which occur frequently. 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 ends on the interior of another edge, occurs, then retract those edges and assert new smaller edges. When this pattern no longer exists, then 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 so that different researchers can understand and use each others' extensions.

Disadvantages Of Prolog

However, there are some problems with using Prolog for geometry.

• There are software engineering problems with using Prolog for a large project because of its lack of nesting in the program and databases.

• 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 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.

• 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 cuts in [Moss85a] show.

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 polyhedron intersection and hidden surface algorithms, Franklin [Franklin82a, Franklin80a], are the alternation of two types of steps:

• Applying function to every element of a set, and

• Combining all the elements having a common key.

This is clearly 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 it. Then the edges around each vertex are paired to form corners. These corners can be considered to be 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, then these two fragments are retracted and a single longer fragment asserted. When such a pattern no longer exists, then we have the output polygons.

Unification

Frequently we wish to determine the closure of some transitive property, such 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.

\[
\begin{align*}
&\text{Create a property list } (\text{plist}) \text{ with one record per vertex, and the property of each vertex a free variable. For example in figure 3i we would have } [1], [2], [3], [4], [5], [6].

&\text{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] \text{ with one unique free variable per graph component.}

&\text{Bind a name identifying each component to the free variables in the list to give something like } [1, \text{first}], [2, \text{first}], [3, \text{first}], [4, \text{second}], [5, \text{second}], [6, \text{third}].

\end{align*}
\]

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

• Wherever the pattern of two edges' projections 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 then the edge segment is visible. Draw them.

• 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 illustrates all of the paradigms operating together.
SUMMARY

Although not perfect, Prolog is a powerful tool for expressing graphical and geometry algorithms in a concise and natural format. This allows larger problems to be solved in a given time, and raises the size of the largest problem that it is feasible to solve.

REFERENCES

Brüderlin85a.
Beat Brüderlin, "Using Prolog for Constructing Geometric Objects Defined by Constraints," Eurocal 85, Conference Proceedings, Linz, Austria, 1985. Institut für Informatik, ETH Zürich, CH-8092, Zürich, Switzerland

Clocksin81a.

Coelho80a.

Franklin80a.

Franklin82a.

Franklin84a.

Franklin85a.

Gonzalez84a.

Moss85a.