CONVEX HULL AND POLYGON INTERSECTION IMPLEMENTED IN PROLOG

Wm. Randolph Franklin & Peter Y.F. Wu

Electrical, Computer, and Systems Engineering Department
Rensselaer Polytechnic Institute
Troy, New York 12180
U.S.A.

Telephone: (518) 266-6077 & 266-8229

18 July 1985

ABSTRACT

Prolog offers several advantages as a tool for implementing geometry and graphics algorithms. The software engineering environment in Prolog allows coding in an intuitive manner, and fosters highly readable programs. Declarative statements used with pattern matching of geometric properties are very well suited for geometric algorithms. Operator overloading allows modular installation of new arithmetic domains, such as exact rational numbers, in existing programs. The efficiency of searches can be improved by new, filtering, rules. Programs on chain linking, polygon intersection, and convex hull are given as examples to illustrate some of these features.

This material is based upon work supported by the National Science Foundation under grant no. ECS-8351942, and by the Data Systems Division of the International Business Machine Corporation.
CONVEX HULL AND POLYGON INTERSECTION IMPLEMENTED IN PROLOG

Wm. Randolph Franklin & Peter Y.F. Wu

1. INTRODUCTION

Prolog represents a radically new approach to computer programming. The approach has been commonly known as "logic programming" [Kowa74]. Prolog is based on predicate logic. Unlike conventional programming languages, a Prolog program does not present a prescribed set of instructions to solve a particular problem, but it describes the objects and their relationships involved in the problem to be solved. Each line of the Prolog program declares a fact or defines a rule about the objects and how they are related to one another.

A Prolog program is invoked by a query: a conjecture to be determined whether it can be implied by the stated facts and propositions. Prolog then automatically goes through sequences of procedures to determine alternative answers to the query based on the given information.

Prolog is implemented by a mechanism of procedural interpretation, the "resolution principle" due to Robinson [Robi65], of clauses in logical propositions (known as Horn clause, named after logician Alfred Horn). To this end, Prolog is strictly sequential: the strategy is a form of linear resolution and depth first search. The procedural approach is adopted for implementation because of the restriction of the predominant architectures of digital computers.

Since its inception in the 70s, Prolog has been gaining wider acceptance in areas such as symbolic mathematics, natural language processing, expert systems and the fields of artificial intelligence. The Japanese Fifth Generation Computer Project has chosen Prolog to be the core programming language for computers of the future.

Clocksin and Mellish [Cloc81] should be consulted for programming in Prolog. A great variety of examples can be used as exercises in Coelho, Cotta and Perreira [Perr80]. This paper is intended to illustrate that Prolog is a viable computing tool for implementing geometry and graphics algorithms.

The examples in this paper were implemented using Waterloo Prolog on the IBM 4341 computer running VM/CMS in the Center for Interactive Computer Graphics, Rensselaer Polytechnic Institute. The source programs are included in the appendix.
2. COMPUTATIONAL GEOMETRY AND PROLOG

Geometry is an intuitive subject. Computational Geometry is concerned about dealing with geometry on the computer. (Forrest first referred to his discipline of dealing with curves, sculptured surfaces on the computer as "Computational Geometry" [Forr72]; Shamos referred to the study of the computational aspects of geometry within the framework of analysis of algorithms [Sham78].)

A fundamental problem in computational geometry is the primitive nature of conventional programming languages. Conceptually simple ideas often are unexpectedly difficult to implement. The descriptive nature of Prolog creates a software engineering environment for coding in a more intuitive manner, and encourages more readable programs. More specifically, the manner in which pattern matching can be done with properties of geometric entities is very well suited to the way geometric algorithms are specified.

Numerous special cases to geometry problems present another form of difficulty in implementation. (Note that failure in identifying special cases for consideration is not an implementation problem but rather a functional specification one, using the terms of software engineering.) Intersection of line segment chains is an illustrative example. The hope is that appropriate design decision in special cases for low level operations will allow the properly working out of algorithms in higher levels.

Numerical inaccuracy due to discretization in finite precision computer arithmetic is another problem which often bedevils the implementation of geometry algorithms [Fran84]. An experimental programming environment is called for to study different arithmetic domains. Operator overloading in Prolog allows modular installation of arithmetic packages, such as exact rational numbers, in existing programs.

Rational number arithmetic will be used in the Prolog example in boolean combination of polygons to guarantee exact solutions to the line segment intersection problems.

3. A SIMPLE PROLOG EXAMPLE - LINKING CHAINS

A chain is a set of connected line segments. Suppose chains are represented by sets of directed edge segments. The problem of "linking chains" is that of determining the sequences of chains in a database of sets of edge segments.
We shall define a data structure "chain". A "chain" is an edge segment, or a connected linkage of segments. The existence of each geometric entity "chain" is described by the following fact:

\[
\text{chain}([V_1,V_2,\ldots,V_n],V_n)
\]

where \([V_1,V_2,\ldots,V_n]\) is a list of \(n\) vertices: \(n-1\) edge segments are connected at \(V_2,V_3,\ldots,V_{n-1}\) respectively and \(V_1,V_n\) the beginning and ending vertices. \((V_n\) is repeated for operational efficiency since it is easy to obtain the first element in a linked list but not the last.) Two chains \([a,b,c]\) and \([p,q,r,s]\) are represented by the Prolog database of the following facts at the outset:

\[
\begin{align*}
\text{chain}([a,b],b). \\
\text{chain}([b,c],c). \\
\text{chain}([p,q],q). \\
\text{chain}([q,r],r). \\
\text{chain}([r,s],s). \\
\end{align*}
\]

The complete Prolog program to link up the edge segments into connected chains consists of the following rules:

\[
\begin{align*}
\text{join}_1 & : - \text{chain}(L_1,X), \\
& \quad \text{chain}([X|L_2],Y), \\
& \quad \text{retract}((\text{chain}(L_1,X))), \\
& \quad \text{retract}((\text{chain}([X|L_2],Y))), \\
& \quad \text{append}(L_1,L_2,L_3), \\
& \quad \text{assert}((\text{chain}(L_3,Y))). \\
\text{join}_1 & : - \text{join}_1, \text{fail}. \\
\text{join}_1 & : - \text{join}_1. \\
\end{align*}
\]

"Join_1" searches for two chains which can be connected together (the ending vertex of one is the beginning vertex of the other). The two chains are then connected together by "retracting" the two chains from the database and "asserting" the connected chain with the appended list. The operator \([X|L]\) in Prolog allows \(X\) to be instantiated to the first element of the list and \(L\) the list of the rest. "Join_all" repeats "join_1" as long as two chains can be connected. "Join_all" therefore would turn the set of facts into:

\[
\begin{align*}
\text{chain}([a,b,c],c). \\
\text{chain}([p,q,r,s],s). \\
\end{align*}
\]

This program compares favorably with a Fortran or Pascal program to accomplish the same goal. It is intended to illustrate some simple features of Prolog in handling topological relationships between geometric entities.
4. ISSUES OF RANGE SEARCHING IN PROLOG

Prolog lacks random addressing for lists. However, direct indexing of an array can be simulated by searching the database for a fact identified by a unique key:

\[
\text{array(key,value)}.
\]

With properly designed hash coding function, random addressing in the above manner takes constant time. Furthermore, a certain predicate referring to a database of facts will also return an answer in constant time, regardless of the existence of a unique key. When there are alternative answers, each answer can be returned in constant time. Hence it is therefore conceivable that the following conjunction of predicates,

\[
\text{chain(L1,x), chain([x|L2],y).}
\]

may be performed in expected time proportional to the size of data set returned.

In geometric pairwise comparisons, the adaptive grid due to Franklin has shown promising linear expected time results [Fran83]. Linear time pre-processing of geometric objects can put them into the framework of adaptive grid described by the following structure:

\[
\text{grid(X,Y,geom_obj).}
\]

Hence, retrieving the set of all objects from a specific grid will take an expected time linear to the size of the set.

5. BOOLEAN COMBINATION OF POLYGONS

In set theoretic definition, a polygon is a set of points in the 2D plane to the inside of its boundaries. Suppose we therefore represent a polygon by a set of edges and along with each edge the information indicating which side of the edge is inside to the polygon (such we shall call "boundary"). The intersection of two polygons A and B is given by:

\[
A \cap B = \{ p \mid p \in A \text{ and } p \in B \}.
\]

This is intuitive, but cannot be directly implemented. An algorithm, based on representing a polygon as a set of boundaries, will show that polygon intersection, as well as other boolean combination of polygons can be defined intuitively for implementation. The algorithm, due to Franklin [Fran82], is implemented in Prolog:
(a) Determine the intersection points between intersecting edges of polygon A and those of B. This includes the case of an end-point of an edge touching the interior of the other edge, but not the case of two end-points touching. Collinear edges overlapping with each other are reported intersecting if either edge contains an end-point of the other edge in its interior.

(b) The edges involved in intersection are split at their points of intersection unless the intersection point is an end-point of the edge.

(c) The edge segments are no longer involved in any further intersection. Each edge of a polygon, A, can be classified in relation to another polygon, B, in four different cases:

   (c.1) The edge is inside of B;
   (c.2) The edge is outside of B;
   (c.3) The edge is also an edge of B with A and B on opposite sides of the edge;
   (c.4) The edge is also an edge of B with A and B both on the same side of the edge; i.e., a common boundary of A and B.

(d) Use the following decision table to select the boundaries of the polygon resulted according to the boolean combination:

   A \bigcup B \quad \text{boundaries of A outside of B or boundaries of B outside of A or common boundaries of A and B;}

   A \bigcap B \quad \text{boundaries of A inside of B or boundaries of B inside of A or common boundaries of A and B;}

   A - B \quad \text{boundaries of A outside of B or opposite to those of B or boundaries of B inside of A but in the reverse sense of boundary;}

We have defined boolean operations on the polygons as operations on the sets of boundaries, since polygons are represented as sets of boundaries. A planar graph traversal may be called for to link up the boundaries in proper cyclic order upon acquisition for information on the global topology. Yet in view of that most basic operations on the polygon (point inclusion, area, further boolean operation, etc.) do not require global topological information, the representation by a set of
boundaries is preferred.

The Prolog program has about 400 lines of text, including comments. This compares very favorably with a typical Fortran program implementing the same algorithm, which would then commonly contain 1000 to 2000 lines of code.

The program demonstrates a clean structure which reads like detail specification of the algorithm. There are three sections: detection of intersection points between edges and the splitting of edges at the points of intersection; classification of all the edge segments; and the forming of boolean combinations according to the definition of the operations on the sets of boundaries. In this sense, Prolog is a good high level language in that it hides from the programmer irrelevant details.

Further, Prolog is good for implementing geometry algorithms because of the idea of pattern matching the intricate topological and geometric properties of objects and data items. For example, in determining an intersection point of two line segments, we do the following pattern matching of the geometric properties:

(1) the lines are not parallel;
(2) the end-points do not touch each other;
(3) two end-points not on the same side of other line.

Operator overloading allows the modular installation of a rational arithmetic package for a Prolog program. Numerical inaccuracy causes serious problems in implementation [Fran84]. In boolean combination of polygons, same problems are revealed in testing collinearity and identify the same point by its coordinates. These problems are resolved by using rational number arithmetic in the Prolog program.

In considering execution time, the approach of pattern matching properties of objects in the database for pairwise comparison would require $O(n)$ time complexity. Good implementation of Prolog can use hashing scheme so that it would take only constant time for each retrieval of a data entry. For example, given facts declared as:

polygon(p_name,edge).

Then the predicate "polygon(p3,E)" can return all the E's in time linear to the number "polygon" entries with p_name "p3".

Above the Prolog implementation level, it has been suggested and implemented in many cases of geometric intersection problems the adaptive grid over pairwise comparison algorithms to improve the processing speed [Fran83]. The scheme can be adapted to our Prolog program
on boolean combination of polygons.

6. 2D CONVEX HULL IN OPTIMAL TIME

This program implements the 2D convex hull algorithm due to Preparata and Hong. [Prep77] The algorithm uses a divide-and-conquer paradigm: the input list of points are divided into left and right subsets partitioned along their X-coordinates. Duplicate points are removed. The convex hull returned is the list of hull points in positive cyclic order; no three consecutive hull points are collinear. Hence the case of a set with less than three points is trivial. The algorithm recursively sub-divides the input data set, terminating at the trivial cases. Then the left and right hulls are recursively knitted together by a merging procedure. Since the left and right hulls do not intersect with each other, they can be merged by finding the top and bottom tangents. To determine the tangents, the top/bottom extreme points are used as starting points and the left/right hull points are searched in the direction defined by the ordering of the extreme points. Since the merging procedure takes only linear time, the algorithm achieves \( O(n \log(n)) \) time complexity which is optimal.

The Prolog program has about 200 lines of text, including comments. Note in the merging procedure, operators P and D can be "+" or "-". P is "+" when searching for bottom tangent, and "-" for top tangent. D is "+" when searching in the counter-clockwise direction, and "-" if clockwise. This flexibility helps to minimize the number of separate segments of code to handle similar special cases. If an arithmetic domain can be defined so that division by zero yields a value "infinite" which can be properly processed at a higher level, the case of horizontal tangent does not need to be separately handled.

7. SUMMARY

Prolog is very briefly introduced. The declarative nature of Prolog allows coding in an intuitive manner and fosters readable code. Pattern matching the properties of geometric entities resembles the way geometry algorithms are specified. Operator overloading allows modular installation of different arithmetic fields, such as rational numbers, for experimentation. The complexity issues of range searching in the Prolog database are briefly discussed. Two examples, namely, Boolean Combination of Polygons and 2D Convex Hull in Optimal Time, are used to illustrate these features in Prolog.
8. REFERENCES


Convex Hull by Divide & Conquer.
Peter YF Wu 07/30/85 */

The Prolog program implements a 2D convex hull algorithm using the
divide and conquer paradigm: the input list of points are divided
into left and right subsets partitioned along their X-coordinates
(so that every point in the left subset is to the left of every
point in the right subset). Duplicate points are removed. The
convex hulls of the left and right subsets are recursively merged
by a merging procedure. Since the merging procedure takes only
linear time, the algorithm achieves \(O(n \log n)\) time complexity.
The algorithm implemented is due to Preparata and Hong, published
in Comm ACM 20,20 (Feb '77), pp. 87-93. */

hull(Input_List, Output_List).
Input_List - set of points on 2D plane: \(X, Y, X, Y, \ldots\).
Output_List - convex hull points in positive cyclic order. */

hull(L, H) :- sort(L, S), zap_duplicate(S, T), c_hull(T, H).

sort(L, S).
Sort list of \(X, Y\) coordinates L into S... */

sort([H|T], S) :-
split(H, T, A, B),
sort(A, A1),
sort(B, B1),
append(A1, [H|B1], S), !.
sort([T, T'], S).

split(H, [T1|T2], [A1|B1], B) :- order(T1, H), !, split(H, T1, A, B).
split(H, [T1|T2], A, [T1|B1]) :- order(H, T1), !, split(H, T1, A, B).
split(_, [T1, T2], [A1, B1]).

order([X1, Y1], [X2, Y2]) :- X1 < X2, !.
order([X, Y1], [X, Y2]) :- Y1 < Y2.

zap_duplicate(Sorted_Input, List_Output). */

zap_duplicate([A|A1|S2, T]):- !, zap_duplicate([A|S2], T).
zap_duplicate([A|S2, S3|A1|T3]) :- zap_duplicate(S, T).
zap_duplicate([S1, S2|T3]).

/* c_hull(Sorted_input_list, Output_Convex_hull) -
c_hull implements the divide and conquer paradigm;
solution is trivial when input list has less than 3 points. */

c_hull(S, S):-
length(S, N),
N<3, !.
c_hull(S, H):-
hull_partition(S, S1, S2),
c_hull(S1,H1),
c_hull(S2,H2),
merge_hulls(H1,H2,H).

/* hull_partition(Sorted_list, Left, Right). */
To sub-divide a list of points sorted in X-coordinates into two:
Left and Right partitions. */

hull_partition(S,S1,S2):-
    length(S,N),
    append(S1,S2,S),
    length(S1,N1),
    N=1+2*N1, !.

/* merge_hulls(Left_Hull, Right_Hull, Combined_Hull). */
To determine the combined convex hull of two given hulls. */

merge_hulls(H1,H2,H):-
    y_extremes(H1,H1B,H1T),    /*! find Bottom/Top values */
    y_extremes(H2,H2B,H2T),    /*! on left and right hull */
    set_scan_direction(H1B,H2B,B),  /*! ordering of Bottom pts */
    set_scan_direction(H1T,H2T,T),    /*! ordering of Top points */
    find_tangent(B,BT,BT),    /*! find BOTTOM tangent---*/
    find_tangent(T,TT,TT),    /*! find TOP tangent------*/
    assemble_hull(H1,BT,H2,TT,H),    /*! assemble the two hulls */
    !.

/* y_extremes(List,Bottom,Top).
   Find the extreme Y-values along the Y-direction:
   Bottom is minimal Y value and Top is maximal Y value. */

y_extremes(F1,F2,F3,Y,Y):-
    y_extremes(L1,L2,L3,Y,B,T),
    (Y<Y,L,L>B),    /*! B: minimal Y (bottom) */
    (Y>T,L,T>B),    /*! T: maximal Y (top) */
    !.

/* set_scan_direction(Left_Y_extreme, Right_Y_extreme, Direction). */
    set_direction "-" for counter-clockwise and "-" for clockwise,
    depending on the ordering of the Y extreme values. */

set_scan_direction(L,B,D):-(L>B,!,D=-; D=+).

/* form_search_arc(List,Y_marker,Direction,Search_Arc). */
    Form search arc as the list in the indicated direction starting
    at the point with Y-coordinate set to the Y_marker. */
form_search_arc(L, Y, D, A):-
    (D='+', L1=L; D='-', reverse(L, L1)),
    rotate(L1, [X1, X2, X3], A),
    !.

/* find_tangent(Pos, Dir, Arc1, Arc2, Tangent). */
/* Determines tangent X1,Y1, X2,Y2 by searching through the arcs; scan direction Dir=""/"""" indicates a max/min tangent gradient; Pos=""/"""" indicates bottom/top tangent. */

find_tangent(P, [X1, Y1], A, [X2, Y2], T):-
    tangent_hori(P, [X1, Y1], A1, [X2, Y2], A2, T),
    !.

find_tangent(P, [X1, Y1], A, [X2, Y2], A2, T):-
    G1 is X1-X2,
    G2 is Y1-Y2,
    tangent_grad(P, G1/G2, [X1, Y1], A1, [X2, Y2], A2, T),
    !.

/* tangent_hori(Pos, Arc1, Arc2, Tangent). */
/* When pending tangent is horizontal, stretch the convex hull for minimal number of vertices. Pos=""/"""" for top/bottom. */

tangent_hori(P, A1, A2, _) :-
    write('...HORI(Pos):'), write(P), nl,
    write('...ARC.1:'), write(A1), nl,
    write('...ARC.2:'), write(A2), nl, fail.

tangent_hori(P, [X1, Y1], [X2, Y2], A1, A2, T) :-
    oper(P, X1-X2, Q),
    (Q>0, !, X=X2; X=X1),
    tangent_hori(P, [X, Y1], A1, A2, T),
    !.

tangent_hori(P, A1, [X1, Y1], [X2, Y2], A2, T) :-
    oper(P, X1-X2, Q),
    (Q>0, !, X=X1; X=X2),
    tangent_hori(P, [X, Y2], A1, A2, T),
    !.

tangent_hori(_, [X1, Y1], [X2, Y2], Y1, Y2) :-
    !.

/* tangent_grad(Pos, Dir, Grad, Arc1, Arc2, Tangent). */
/* Examine gradient of proposed tangent according to direction specified; if tangent gradient is not satisfactory, advance the respective arc to search the next point for the tangent. Pos=""/"""" indicates bottom/top tangent. */

tangent_grad(P, D, G, A1, A2, _) :-
    write('...GRAD(Pos, Dir, Grad):'), write(P, D, G), nl,
    write('...ARC.1:'), write(A1), nl,
    write('...ARC.2:'), write(A2), nl, fail.

tangent_grad(P, D, [X1, Y1], [X2, Y2], A1, A2, T) :-
    oper(D, X1-X2, Q),
    oper(P, Q, R),
    oper(P, Q, R),
    !.

/* Complete code to determine tangents for a hull */
(R>0; !, X=X1; X=X2),
  find_tangent(P, D, [X1, X1, Y1], A1), A2, T),
  !.

tangent_grad(P, D, G1/G2, [X1, Y1], A1, A2, T) :-
  H1 is (X1-X2)*G2,
  H2 is (Y1-Y2)*G1,
  oper(D, (H1-H2)*(Y1-Y2)*G2, R),
  (R>0; H1=H2, D=**)!,
  find_tangent(P, D, [X2, Y2], A1, A2, T),
  !.

tangent_grad(P, D, G1/G2, [X1, Y1], A1, A2, T) :-
  oper(D, X1-X2, Q),
  oper(P, Q, R),
  (R>0; !, X=X1; X=X2),
  find_tangent(P, D, [X2, Y2], A1, A2, T),
  !.

tangent_grad(P, D, G1/G2, A1, [X1, Y1], A2, T) :-
  H1 is (X1-X2)*G2,
  H2 is (Y1-Y2)*G1,
  oper(D, (H1-H2)*(Y1-Y2)*G2, R),
  (R>0; H1=H2, D=**)!,
  find_tangent(P, D, [X2, Y2], A1, A2, T),
  !.

tangent_grad(P, D, G1/G2, A1, [X1, Y1], A2, T) :-
  oper(S, E, R)
  F =.. [S, E],
  R is F.
  /
* assemble_hull(L_hull, bottom_Tangent, R_hull, top_Tangent, Hull).
  To stitch the left and right hulls together along the bottom and top tangents to assemble the combined hull. */

assemble_hull(H1, [B1, B2, T1], H2, [T2, T1], H) :-
  rotate(H1, T1, L1),
  append(L, [B1], L1),
  rotate(H2, B2, L2),
  append(R, [T2], L2),
  append([B1, B2, T2, L2], L),
  append(L, [T1, H], H).
/* UTIL PROLOG 07/03/85
utilities on list manipulation.
Peter Wu 07/30/85 */

/* Length...
Compute length of a list (in time linear to length) */

length([], N) :- length([], N1), N is N1 + 1.
length([_|L], N) :- length(L, N1), N is N1 + 1.

/* Append...
Appending a list to the end of another.
Notice that any two of the arguments of "append" can be instantiated,
and "append" will instantiate the third argument to the appropriate
result. This property is known as "reversible programming". */

append([], L, L).
append([H|T], L, [H|S]) :- append(T, L, S).

/* Rotate...
Rotate a list until the first occurrence of a marker becomes the 1st
entry in the list. */

rotate(L1, X, L2) :- append(K1, [X|K2], L1), append([X|K2], K1, L2).

/* Reverse...
Reversing a list. */

reverse(L1, L2) :- reverse(L1, [L2|L], L2).

reverse([H|L1], L, L2) :- reverse(L1, [H|L2], L2).
reverse([_|L1], L, L2) :- reverse(L1, L2, L2).