© Regeo. Geometric Reconstruction Group
www.regeo.uji.es
Technical Reports. Ref. 05/2009

Implementing the new algorithm for finding faces in wireframes

Peter A.C. Varley

Department of Mechanical Engineering and Construction, Universitat Jaume I, Spain

Abstract
This Technical Note describes an implementation of a new method of finding faces in wireframes. It also provides a basis for implementing the same algorithm to solve other least-cost graph problems where the cost of traversing an edge is context-dependent.

Index Terms: Wireframe, Face Identification, Line-Drawing Interpretation, Visual Perception

1. Introduction

In Varley and Company (2009), we introduce a new algorithm for finding faces in wireframes. This algorithm could be adapted for use in other least-cost graph problems where the cost of traversing an edge is not fixed but context-dependent.

This technical note offers guidance for implementing this algorithm. Section 2 describes the "pure" algorithm, and should be used as a guide when implementing the algorithm in other contexts. Section 3 describes the algorithm as implemented for Varley and Company (2009), which includes additional clauses to handle the awkward special case of K-vertices, described in Section 5.5 of Varley and Company (2009).

Our input data is partly *topological* (a *undirected* graph G(V,E)) and partly *geometric* ((x,y) coordinates for each vertex of the graph). The graph is subject to the restrictions (i) that no edge connects a vertex to itself and (ii) no two edges join the same two vertices.

Formally, the problem to be addressed is best expressed in terms of *half-edges*, where a half-edge is an *arrow* of the *directed* graph G'(V,A) which contains arrows {a,b} and {b,a} if and only if the original undirected graph G(V,E) contains the edge {a,b}.

A *cycle* is a closed path of half-edges.

The first objective is to obtain a set of cycles in which each half-edge is included in exactly one cycle. Additionally, a topology must be geometrically realisable: it must be possible to select vertex z-coordinates such that each cycle is planar (in practice, it is necessary to specify "planar within a tolerance" rather than "exactly planar").

Where only one valid set of cycles exists, the objective is simply to find it. Where more than one valid set of cycles exists, the objective is to find the one which matches the interpretation of the drawing which a human interpreter would consider to be the most plausible.

It is assumed that at least one valid set of cycles exists. The related but distinct problem of determining whether or not any valid set of cycles exists is not addressed here.

2. 2. GENERAL IMPLEMENTATION

2.1 2.1 DATA STRUCTURES

This section describes internal data structures and their associated operations.

One internal data structure is used. A <u>*string*</u> is a data structure which contains:

- Start and end vertices
- An ordered sequence of zero or more intermediate vertices
- A priority

The two operations which are performed on strings are <u>*concatenation*</u> and <u>*merger*</u>.

<u>*Concatenation*</u> combines two strings S and T to form a single string S'. It proceeds as follows:

- The start vertex of S' is the start vertex of S

- The end vertex of S’ is the end vertex of T
- The intermediate vertices of S’ are (in sequence) the intermediate vertices (if any) of S, the start vertex of T and the intermediate vertices (if any) of T
- The priority of S’ is application-dependent (I recommend that it should be lower than the priority of the higher-priority of the two strings S and T)
- The two strings S and T are deleted and the single string S’ is added in their place

Concatenation can only be performed if the end vertex of S is the same as the start vertex of T. It cannot be performed if any other vertex of S (start vertex or intermediate vertices) is the same as any other vertex of T (intermediate vertices or end vertex). It cannot be performed if the new vertex triple (centred on the common vertex) created by the operation already appears in any completed face or any other string.

<u>*Merger*</u> combines two strings S and T to form a single face F. It proceeds as follows:

- The vertices of F are (in sequence): the start vertex of S; the intermediate vertices (if any) of S; the start vertex of T; and the intermediate vertices (if any) of T.
- The two strings S and T are deleted and the face F is created and exported to the calling program.

Merger can only be performed if the end vertex of S is the same as the start vertex of T and the end vertex of T is the same as the start vertex of S. It cannot be performed if any of the intermediate vertices of S is the same as any of the intermediate vertices of T. It cannot be performed if either of the new vertex triples (centred on the common vertices) created by the operation already appears in any completed face or any other string. It cannot be performed if neither S nor T has intermediate vertices (a face must contain at least three vertices).

2.2 ALGORITHM

```
Stage 1: Initialisation
.       Perform any application-dependent initialisation
        which is required
.       Initialise the master list by creating the initial pool of
        strings (two per edge)
.       Merge two strings to break the initial symmetry

Stage 2: search for forced moves in the master list
.       If the master list is empty, exit
.       Search the master list for forced moves. If one is
        found:
.       .       Perform the forced move in the master list
.       .       If this move completes a cycle
.       .       .       Merge the two strings in the master
                        list to complete the cycle
.       .       .       Export the completed cycle
.       .       .       Remove the string from the master
                        list
.       .       Repeat from Stage 2

Stage 3: search for face completions in the master list
.       Search the master list for mergers which will complete cycles. If one is found:
.       .       Merge the two strings in the master list to
                complete the cycle
.       .       Export the completed cycle
.       .       Remove the string from the master list
.       .       Repeat from Stage 2

Stage 4: examine alternatives
.       Take a working copy of the master list

Stage 4a: continue examining alternatives
.       Find the highest priority string S in the working
        copy, and the string T which has the best mating
        value with S
.       Merge S and T in the working copy, reducing the
        priority of the resulting merged string

Stage 4b: examine consequences of most recent move (i):
forced moves
.       Search the working copy for forced moves. If one is
        found:
.       .       Perform the forced move in the master list
.       .       If this move completes a cycle
.       .       .       Merge the two strings in the working
                        copy to complete the cycle
.       .       .       Export the completed cycle
.       .       .       Remove from the master list all
                        strings which contribute to this cycle.
.       .       .       Repeat from Stage 2
.       .       Repeat from Stage 4b

Stage 4c: examine consequences of most recent move (ii):
face completions
```

```
.   Search the working copy for mergers which will
    complete cycles. If one is found:
.   .   Merge the two strings in the working copy
        to complete the cycle
.   .   Export the completed cycle
.   .   Remove from the master list all strings
        which contribute to this cycle
.   .   Repeat from Stage 2
.   Repeat from Stage 4a
```

3. IMPLEMENTATION FOR FINDING FACES IN WIREFRAMES

3.1 DATA STRUCTURES

This section describes additional internal data structures and their associated operations.

A potential *through-edge* is a data structure which contains:

- Start, middle and end vertices
- A flag indicating whether the through-edge is used in any cycle of edges

Through-edges modify the operations (concatenation and merger) available to strings. For any through-edge with start, middle and end vertices S, V and T, either S-V-T or T-V-S (but not both) must eventually be part of a face.

The flag is initially set to 3. When strings U-V-W are merged: the flag of any through S-V-T is decremented by 2; the flag of any other through S-V-* or T-V-* is decremented by 1; and any through edge with the flag now zero is deleted from the list of through-edges. This update should be implemented as part of the processes of *concatenation* and *merger* (Section 2.1).

If the through-edge flag S-V-T is exactly 2, strings S-V and V-T must be merged. This should be implemented as part of the process of checking for *forced moves* (Section 3.3.1).

If the through-edge flag S-V-T is exactly 3, strings S-V and V-T may be merged. This test should be implemented as part of the process of checking whether *concatenation* or *merger* is permitted (Section 2.1).

3.2 PREPROCESSING

This section describes processing which is best carried out once, before proceeding to the main algorithm.

3.2.1 Detection of Triangular Loops

Algorithm: for each vertex V: for each pair of edges E and F touching V, if an edge exists which joins the other end of E to the other end of F, a triangular loop has been found, in which case add it to the list.

2.2.2 Detection of Quadrilateral Loops

Algorithm: for each pair of (nearly) parallel edges E and F in the same subgraph, determine the vertices U_E, V_E, U_F and V_F; if all vertices are different, determine whether (i) edges U_E-U_F and V_E-V_F exist or (ii) edges U_E-V_F and U_F-V_E exist; in either case, if these two edges are (nearly) parallel, a quadrilateral loop has been found, in which case add it to the list.

2.2.3 IdentifyThroughVertices

Algorithm: for each pair of (nearly) parallel edges E and F meeting at a vertex V where (i) four or five edges meet at V or (ii) six or more edges meet at V, the remaining four being on the same side of the line through E and F: this constitutes a through edge, so add it to the list.

3.2.4 Others

Note: in practice, inter-edge angles are better calculated as required, not by a pre-processor.

3.3 3.3 ALGORITHM

```
Stage 1: Initialisation
.   Detect triangular loops, quadrilateral loops and
    through edges, as described above
.   Initialise the master list by creating the initial pool of
    strings (two per edge)
.   Break the initial symmetry (Section 3.3.2)

Stage 2: search for forced moves in the master list
.   If the master list is empty, exit
.   Search the master list for forced moves (Section
    3.3.1). If one is found:
.   .   Perform the forced move (Section 3.3.1) in
        the master list
.   .   If this move completes a cycle
.   .   .   Merge the two strings in the master
            list to complete the cycle
.   .   .   Export the completed cycle
.   .   .   Remove the string from the master
            list
.   .   Repeat from Stage 2

Stage 3: search for face completions in the master list
.   Search the master list for mergers which will com-
    plete cycles. If one is found:
.   .   Merge the two strings in the master list to
        complete the cycle
.   .   Export the completed cycle
```

```
        Remove the string from the master list
        Repeat from Stage 2

Stage 4: examine alternatives
    Take a working copy of the master list

Stage 4a: continue examining alternatives
    Find the highest priority string S in the working
    copy, and the string T which has the best mating
    value with S
    Merge S and T in the working copy, reducing the
    priority of the resulting merged string

Stage 4b: examine consequences of most recent move (i):
forced moves
    Search the working copy for forced moves (Section
    3.3.1). If one is found:
        Perform the forced move (Section 3.3.1) in
        the master list
        If this move completes a cycle
            Merge the two strings in the work-
            ing copy to complete the cycle
            Export the completed cycle
            Remove from the master list all
            strings which contribute to this cy-
            cle.
            Repeat from Stage 2
        Repeat from Stage 4b

Stage 4c: examine consequences of most recent move (ii):
face completions
    Search the working copy for mergers which will
    complete cycles. If one is found:
        Merge the two strings in the working copy
        to complete the cycle
        Export the completed cycle
        Remove from the master list all strings
        which contribute to this cycle
        Repeat from Stage 2
    Repeat from Stage 4a
```

3.3.1 Forced Move

```
Input parameter: list (master list or working list)

Stage 1: detection of forced moves

Initialise a count to zero
For each pair of strings S and T in the given list
    If the end vertex of T is the start vertex of S
        If the through edges (penultimate vertex of
        T, end vertex of T) and (end vertex of T,
        start vertex of S) require that the two
        strings must be concatenated or merged
        (Section 3.1), set the count to one and exit
        the loop as this merger must be performed
        regardless of what else is possible
        Otherwise, if strings S and T can be
        merged (Sections 2.1 and 3.1), increment
        the count
If the resulting count is exactly one, there is a forced move
to be performed

Stage 2: Performing the forced move

If the end vertex of S is the start vertex of T, merge the two
strings to create a face (Sections 2.1 and 3.1)
Otherwise, concatenate the two strings (Sections 2.1 and
3.1)
```

3.3.2 Make First Move

```
Choose the highest-priority trihedral vertex as the seed
vertex V

If an appropriate seed vertex V is found
    Choose any string S starting with V and a string T
    which ends at V and is not the other half-edge of S
    and merge them
Else
    Choose any triangular loop of edges and create a
    face from them, deleting the three edges from the
    master list
```

References

[1] P.A.C. Varley and P.P. Company, 2009. *A New Algorithm For Finding Faces In Wireframes*, accepted for publication in Computer-Aided Design.