www.regeo.uji.es
Technical Reports. Ref. 04/2009

An Implementation of Dijkstra's 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 a method of finding faces in wireframes based around Dijkstra's Algorithm. It includes minor improvements over similar methods used by Varley and Shesh and Chen, amongst others.

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

1. Introduction

This Technical Note describes a method of finding faces in wireframes based around Dijkstra's Algorithm. It includes minor improvements over similar methods used by Varley (2000 and 2003) and Shesh and Chen (2004), amongst others.

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. Improvements

The implementation described here incorporates the following two improvements over previously-published approaches.

2.1 Detection of Non-Planar Face Loops

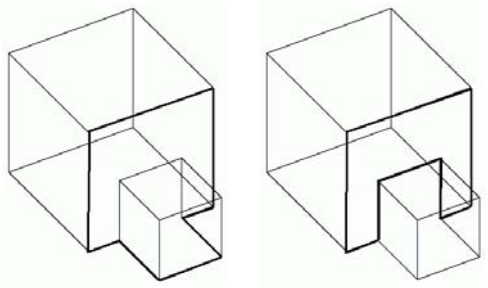

Figure 11: *Face Loops: Wrong and Right Choices*

The "wrong" choice can occur using the approaches of Varley (2000) and Shesh and Chen (2004), both of which set the "path length" (the cost of traversing a particular edge) to 1. One way of ensuring that the "right" choice is made is to use the path length as a measure of how badly a particular edge fits the rest of the loop of edges; this was suggested in Varley (2003) and is the approach adopted here. See Section 3.4

2.2 Detection of Non-Planar Face Loops

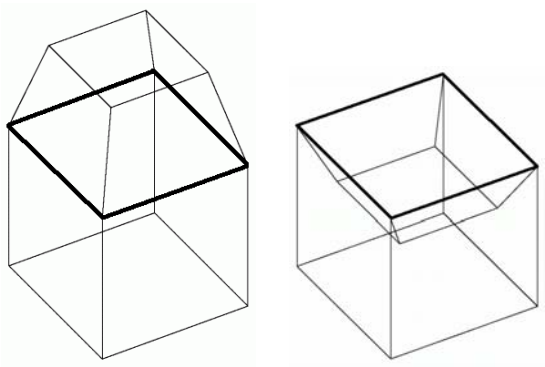

Figure 2: *Internal Faces*

Varley (2000 and 2003) does not contain any explicit method for avoiding internal faces, but generally does not suffer from this problem as visible face loops have already been identified by a previous process. Shesh and Chen (2004) does not contain any explicit method for avoiding internal faces either.

This technical note introduces a method of avoiding faces based on choosing an appropriate starting-point for face loop detection. This is described in Section 3.3.

3. Pseudocode

The problem to be solved can be stated as: find the set of loops of half-edges which corresponds to the set of external faces, while avoiding the pitfalls described in Section 2; each edge has two half-edges which are traversed in opposite directions, each half-edge must appear in exactly one loop, and the two half-edges of any edge must not appear in the same loop. The approach used here is similar to that presented in [36], with the difference that in that work (a) the algorithm used an inflated wireframe, not 2D data, and (b) the algorithm started with some face loops already known, whereas here it is used to identify *all* face loops.

Note that the overall control structure (Section 3.1) and Dijkstra's Algorithm (Section 3.2) are purely topological. When we wish to make use of geometric information, we must use the algorithm's externals, its *starting point* (Section 3.3) and the *cost function* (Section 3.4).

Note also that the algorithm does not know (or care!) whether it is determining clockwise or anticlockwise face loops, although it does ensure that the same choice is made for all faces in the same subgraph. Neither does it detect geometric problems such as self-intersection.

We assume that the drawings are in parallel projection. The algorithm presented here could be used for other projections by changing the cost function (Section 3.4).

3.1 Overall Algorithm

Essentially, this approach uses repeated application of Dijkstra's Algorithm (1959) for finding loops in an undirected graph. We should also take advantage of particular face types, as discussed next, and avoid two particular pitfalls, creation of internal faces (see 4.1.3) and choosing the wrong path (see 4.1.4).

Triangular loops of edges can be treated as a special case. Although, of necessity, all triangular loops of edges are planar, not all are true faces, since some could be internal faces. It is simple to prove that any triangular loop of edges containing at least one trihedral vertex must be a true (external) face. Triangular loops of edges containing no trihedral vertices may, but need not necessarily, also be external faces.

A good case could also be made for treating quadrilateral loops of edges where at least one of the pair of opposite edges appears to be parallel as a special case. Unlike triangular loops, there is no criterion which means that these *must* be faces, but even so they quite often are. Determination of what is, and what is not, a pair of parallel edges is beyond the scope of this paper.

The resulting algorithm is:

Determine the subgraphs, allocating each edge to a subgraph

  For each subgraph

  (Preliminaries)

    List those triangular loops which must necessarily correspond to faces

    List those quadrilateral loops in which at least one pair of opposed edges appears to be parallel (see 3.4)

    Mark each half-edge (forward and reverse for each edge) as unallocated

    (Find the faces)

    While there are half-edges not allocated to a face

      If any listed triangular loop includes an edge which has only one unallocated half-edge

        Create the corresponding triangular face

        Mark the three half-edges as allocated

        Remove the triangular loop from the list

      Otherwise

        Choose a starting edge, as described in 3.3

        Create the face, using Dijkstra's Algorithm, as described in 3.2

3.2 Finding a Face

The algorithm for finding one face is:

Given a suitable half-edge (see 4.1.3)

  (Preliminaries)

The starting point is the end of the chosen half-edge

The target is the start of the chosen half-edge

Set *Cost Known* to *True* for the starting point and *False* for all other vertices

Set *Cost* to *Zero* for the starting point and *Unknown* for all other vertices

Set *Branched from Here* to *True* for the starting point

Set *Branched from Here* to *False* for all other vertices

Set *Route* for the starting point to the target

Set *Route* for all other vertices to *Unknown*

For each valid (see note below) vertex *V* which can be reached from the starting point via an unallocated half-edge

Set *Cost Known* for *V* to *True*, *Cost* for *V* to 1, and *Route* for *V* to the starting point

(Find the Face)

While *Cost Known* for the target is *False*

Choose the *Current Branching Point*: from the set of vertices for which *Cost Known* is *True* and *Branched from Here* is *False*, choose the one with the smallest *Cost*

For each valid vertex *V* which can be reached from the *Current Branching Point* via an unallocated half-edge:

Calculate the cost of traversing this half-edge (see 4.1.4)

Calculate the total cost for reaching *V* by this route as *Cost* of *Current Branching Point* plus the cost of traversing this half-edge

If *Cost Known* for *V* is *False* or if the total cost of reaching *V* by this route is less than the current *Cost* of *V*

Set *Cost Known* for *V* to *True*, set *Cost* for *V* to the total cost of reaching *V* by this route, and set *Route* for *V* to *Current Branching Point*

Set *Branched from Here* for *Current Branching Point* to *True*

(Postprocessing)

Find the loop of vertices by recursively retracing *Route*, starting at the target and working back

Mark the half-edges used to form this loop as allocated

Note: a vertex is *valid* (in the algorithm above) if including it in the current loop will not result in any three consecutive vertices being traversed in this loop which have already been traversed together in any preceding completed face loop. This ensures that we do not identify the two orientations (clockwise and anticlockwise) of a face loop as separate face loops.

3.3 Avoiding Internal Faces

Dijkstra's Algorithm allows us control of two variables: choice of starting point and choice of cost function. Since our most pressing problem is to avoid internal faces, we use our ability to choose the starting-point in such a manner as to ensure that the algorithm preferentially finds true, external faces.

Consider the left-hand drawing in Fig. 1. In which order do we want to determine face loops? It would be best to start with an edge which is demonstrably convex in 3D: all other edges lie on the same side of a plane through this edge (a line in 2D is demonstrably convex in 3D if all other lines lie on the same side of the (extended) line).

Perhaps we start with the top face, since this is all trihedral. We follow this with the four faces which border it, since we can ensure that these will be in the same orientation (clockwise or anticlockwise) as the initial face.

But which face should we follow next? We do not want to start at an edge joining the non-trihedral vertices, as that could lead us to follow an internal face. However, starting at an edge on the bottom face can also lead to a problem: since this is isolated from the faces we have already determined, there is no guarantee that we will follow it in the same orientation (clockwise or anticlockwise). What we want to do is start with one of the edges joining a partially-used non-trihedral vertex to an unused trihedral vertex.

Accordingly the priority order for choosing starting edges is:

1. the remaining half-edge of any edge joining two trihedral vertices, both of which have already been used in one or more faces,
2. a half-edge of any edge joining two trihedral vertices, one of which has already been used in one or more faces,
3. a half-edge of any edge joining a trihedral vertex which has already been used in one or more faces to any vertex,
4. a half-edge of any edge joining a trihedral vertex to any vertex which has already been used in one or more faces,
5. a half-edge of any convex boundary edge joining two trihedral vertices,
6. a half-edge of any other edge joining two trihedral vertices.

Within each category, edges which are included in triangular loops which necessarily correspond to faces are to be preferred to those which are not, and edges which are included in quadrilateral loops which plausibly correspond to faces are to be preferred to those which are not.

3.4 Choosing the Right Path: Cost Function

Dijkstra's Algorithm assigns a cost, also termed the *path length*, for traversing each edge. Since geometry is continuous and *path length* is the only non-integer quantity in Dijkstra's algorithm, we use the cost to reflect the requirement that each face loop should be planar. For our purposes, we wish the cost to be small if the edge is close to being coplanar with the path taken to it, and large if the edge is far from coplanar with the path taken to it.

Varley (2003) showed that with an inflated wireframe, choosing the wrong path can be normally avoided by sensible choice of cost function. To test whether it is possible to do this for 2D wireframes too, we make use the idea that face loops should preferably comprise two groups of parallel lines.

It is worth noting that the *only* difference between the 2D and 3D versions of this algorithm is that they use different cost functions.

We calculate the cost of traversing an edge as $\boldsymbol{cost=\varepsilon+(1/max(\varepsilon,Merit))-1}$. Merit, in the range 0–1, varies depending on whether we are working in 2D, before inflation (see 4.1.4.1) or in 3D, after inflation (see 4.1.4.2). $\boldsymbol{\varepsilon}$ is a small value (we use $\boldsymbol{\varepsilon= 10^{-6}}$) included to ensure that the cost is always finite and positive.

3.4.1 2D Path Length Function

We calculate the path length function for working in 2D as follows:

Let D = vertex under consideration

Let C = known vertex prior to D

Let B = known vertex prior to C

Let A = known vertex prior to B

While AB and BC are collinear

  Let B = known vertex prior to C

  Let A = known vertex prior to B

If A is not a known vertex, *merit* is 0.5 (to make the cost of edge CD = 1)

Merit is the higher of $\boldsymbol{(\cos \alpha)^k}$ and $\boldsymbol{(\cos \beta)^k}$ where $\boldsymbol{\alpha}$ is the angle between AB and CD, $\boldsymbol{\beta}$ is the angle between BC and CD, and $\boldsymbol{k}$ is an arbitrary constant.

For historical reasons, we use $\boldsymbol{k=10}$.

3.4.2 3D Path Length Function

The function which we use to determine how close two lines are to being parallel is $\boldsymbol{(\hat{a}\cdot\hat{u})^k}$, where $\boldsymbol{\hat{a}}$ is a normalised vector in the directions of the edge under consideration, $\boldsymbol{\hat{u}}$ is the closest normalised vector to in the plane of the previous vertices on the path to this edge (calculated as $\boldsymbol{\hat{u}=(\hat{a}\times\hat{o})\times\hat{a}}$, where $\boldsymbol{\hat{o}}$ is a normal to the plane) and $\boldsymbol{k}$ is an arbitrary even integer constant. For historical reasons, we use $\boldsymbol{k=10}$.

References

[1] E.W. Dijkstra, 1959. *A Note on Two Problems in Connexion with Graphs*, Numerische Mathematik I, 269–271.

[2] A. Shesh and B. Chen, 2004. SMARTPAPER: *An Interactive and User Friendly Sketching System*, Computer Graphics Forum 12(3), 301–310.

[3] P.A.C. Varley and R.R. Martin, 2000. *Constructing Boundary Representation Solid Models from a Two-Dimensional Sketch: Topology of Hidden Parts*, Proc. First UK-Korea Workshop on Geometric Modeling and Computer Graphics, 129–144. Kyung Moon Publishers.

[4] P.A.C. Varley, 2003. *Automatic Creation of Boundary-Representation Models from Single Line Drawings*, PhD Thesis, University of Wales.