The Implementation of an Algorithm to Find the Convex Hull of a Set of Three-Dimensional Points

A. M. DAY
University of East Anglia

A detailed description of the implementation of a three-dimensional convex hull algorithm is given. The problems experienced in the production and testing of a correct and robust implementation of a geometric algorithm are discussed. Attention is paid to those issues that are often brushed over in the theoretical descriptions but cause errors in a real computation. These include degeneracies such as coplanar points, floating-point errors, and other special, but not necessarily degenerate, cases.

Categories and Subject Descriptors: I.3.5 [**Computer Graphics**]: Computational Geometry and Object Modelling—*curve, surface, solid and object representations; geometric algorithms, language and systems*

General Terms: Algorithms, Theory

Additional Key Words and Phrases: Convex hull, divide and conquer, edge structure, implementation, tetrahedron, triangulation

1. INTRODUCTION

This report investigates the implementation in Pascal of a geometric algorithm that constructs the convex hull of a set of points in three dimensions. A great deal of work in computational geometry (e.g., [9]) has been directed toward the design and analysis of algorithms, but fails to explore many of the implementation details. Our experience demonstrates that implementation is definitely not a trivial exercise and shows that without it the study of an algorithm is incomplete. Careful testing always reveals the inadequacies of algorithms that in theory appear both ingenious and optimal in terms of worst-case complexity. For example, experiment has revealed that this algorithm would appear to have a worst-case complexity of $O(n^2)$ and not $O(n \log n)$ as was previously thought.

The conversion into working programs raises several significant areas of difficulty. These include the creation and management of suitable data structures, the handling of special cases or degeneracies, and the inaccuracy and inconsistency of floating-point arithmetic. The algorithm chosen for implemention is

Author's address: School of Information Systems, University of East Anglia, Norwich NR4 7TJ, England.

based on the divide-and-conquer strategy for constructing the convex hull of points in three dimensions. It was chosen because

(1) after some preliminary studies it seemed that it was sufficiently complex to generate all the problems listed above; and
(2) a future objective is the investigation of parallel implementations of geometric algorithms, and this algorithm exhibits some potential for adaptation to a parallel environment.

It should be noted that the problems of dealing effectively with degeneracies such as points with coincident y-coordinates, and collinear and coplanar points have not been entirely solved at this stage. This is the subject of the next stage of the implementation and will probably use a perturbation technique (see, e.g., [6]).

2. ALGORITHM OUTLINE

The program is based on the Preparata–Hong algorithm: for the reader who is not familiar with this work, the following section gives a brief outline. A presentation can also be found in [6], [8], and [9]. The full program text can be found in [5].

Initially the points are sorted according to their y-coordinates, and the recursive divide-and-conquer technique can then be applied. The algorithm splits the ordered set of points into an upper and lower subset from which a pair of convex hulls are recursively generated. The "sub" convex hulls are then merged to form the complete hull using a "wrapping" technique that inspects edges around appropriate vertices on the upper and lower hulls in order to establish a set of connecting edges between them. The wrapping process is initiated with a connecting edge that is guaranteed to be part of the merged hull. To do this, we maintain, within the 3-D surface structure, the path describing the 2-D convex hull of the 3-D hull points when projected onto the x–y plane. In the sequel we refer to this path, which is in reality 3-D, as the "2-D convex hull." From a pair of these "2-D convex hulls," a supporting line is found and used as the start edge.

The recursive subdivision proceeds until a sufficiently small subset of points is reached for their convex hull to be constructed by a straightforward method. In this implementation we stop at four points and construct a tetrahedron.

An important point—to be discussed in more detail later—is that, due to the convexity of the merging polyhedrons, as an edge is selected on a vertex scan, those passed over can be permanently set aside, as they are guaranteed not to be on the final hull. Also as the "wrapping" triangulation is formed, the faces, edges, and points of the old "sub" hulls that become interior are removed from the surface data structure (see Figure 1).

The algorithm has been shown [8] to give the convex hull in optimal time $\Theta(N \log N)$, where N is the number of original points.

3. THE PROGRAM

3.1 Data Structures

The efficient implementation of this algorithm depends very much on the choice of suitable data structures. The literature relevant to this algorithm is unanimous

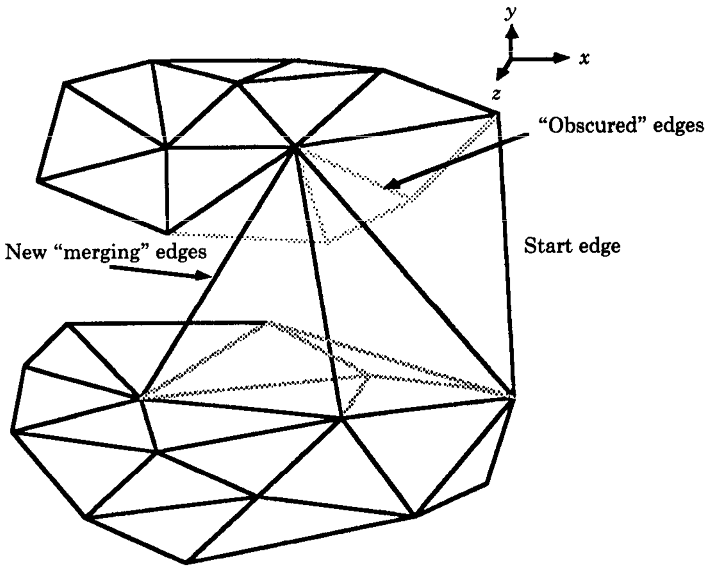


Fig. 1. "Sub" convex hulls and some connecting edges.

in taking the edge, rather than the point or the face [1], as the central component in the representation of hulls: Preparata and Shamos suggest a "doubly connected edge list" [9], and Edelsbrunner [6] uses a structure similar to the "quad-edge" structure of [7]. We have also used a variant of the winged edge structure [3, 4] (see Figure 2).

The main structures are defined in Pascal as follows:

```
  type
    pointptr = ^point3;
        edgeptr = ^edge;
        vertexptr = ^vertexinfo;
        point3 = record
            x, y, z : extended;
      end;     {-----3d point coords, x, y, z, -----}
    vec = point3;     {-----3d vectors-----}
    edge = record
        v1, v2 : vertexptr;
        fwd2d, bwd2d : edgeptr;
      end;
    vertexinfo = record
      v : pointptr;
          nedgelef, nedgerit : edgeptr;
      end;
```

The record type **edge** identifies the end vertices **v1** and **v2** and the corresponding edge links. The **nedgerit** and **nedgelef** links allow efficient scanning of edges around a vertex in either a clockwise or anticlockwise direction, respectively. This operation is fundamental to the efficiency of the wrapping stage since these pointers make it possible to visit all edges with a common origin in order, in either direction and in constant time per edge. The same links also

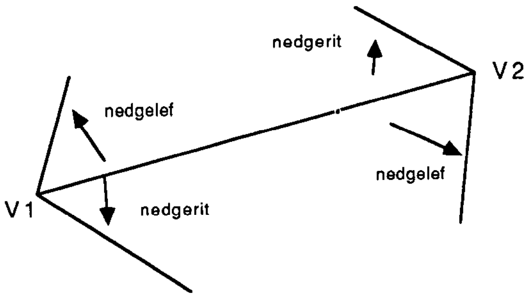


Fig. 2. Data associated with an edge.

make it possible to visit all edges of a facet in either direction and in constant time per edge. The edge record uses links (**vertexptr**) to **vertexinfo** rather than fields directly representing vertices and **edgeptrs**. This is because although the edges are essentially undirected, when considering the set of edges incident upon a given vertex, it is very convenient to consider all these edges as being temporarily directed away from the common vertex. To this end heavy use is made of the procedure **rev_edge**, which in effect interchanges the naming of the vertices. This operation can be performed much more cheaply if all the data associated with a vertex are accessed indirectly via a pointer. The edge record also contains fields that are available for use as pointers to the succeeding and preceding edges in the projected 2-D convex hull. These provide a doubly linked list for the edges on the 2-D hull that supports the efficient determination of supporting lines for a pair of hulls as mentioned in the previous section and as described in more detail in Section 3.4.

In order to improve the accuracy of numerical calculations, the Apple SANE numerics library (IEEE Standard Apple Numeric Environment; see, e.g., [2]) is incorporated in order to supply the TYPE **extended** that gives an IEEE standard 80-bit extended format number representation.

Access to both the 2- and 3-D hulls is via the record type **polyhedron**, which consists of pointers to both the leftmost and rightmost edges of the 2-D convex hull as shown in the following Pascal-type definition:

```
polyhedron = record
    lef2d, rit2d: edgeptr;
  end;
```

3.2 Preliminaries and Divide and Conquer

Initially the data are sorted using a Quicksort with Insertion sort for small subsets. The recursive subdivision and merge are then performed by the central procedure **chull**.

```
procedure chull (i: point_id;   {-----recursively splits, merges-----}
      n: natural;
      var h: polyptr);
    var
    n1, n2: natural;
    h1, h2: polyptr;
```

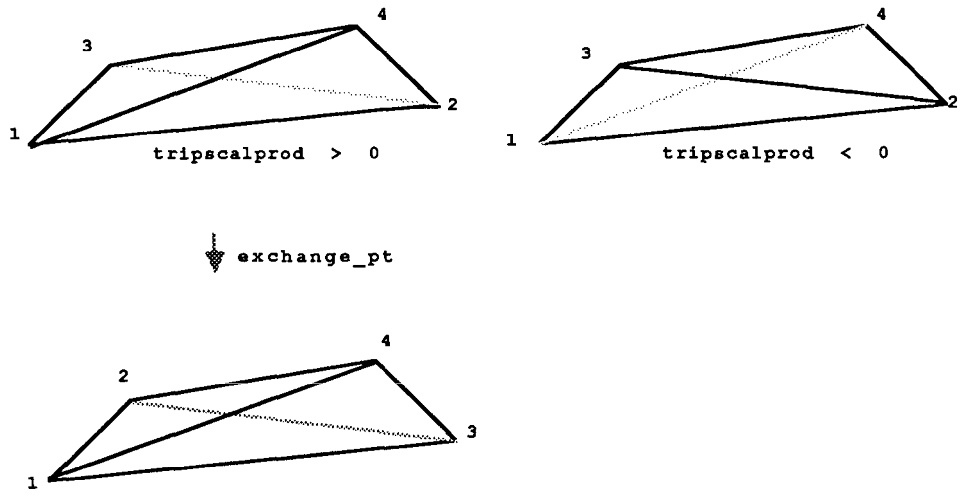


Fig. 3. Standardized naming of points in a tetrahedron.

```
begin
  new(h);
  if n = 4 then
    tetrahull(i, h)
  else    {-----n mod 4 = 0-----}
    begin
      nl := n div 2;
         nl := nl − nl mod 4;
         n2 := n − nl;
         chull(i, nl, hl);
         chull(i + nl, n2, h2);
         merge_3d(hl, h2, h);
    end;
end;    {-----of chull-----}
```

In order that the recursive subdivision performed by **chull** can be guaranteed to terminate with precisely the four points required for the construction of a tetrahedron, the size of the initial set of points is rounded up to be a multiple of four, if necessary, by the insertion of up to three dummy points into the original set. This is performed by the procedure **roundup_pcount**, which creates points that are guaranteed to be interior to the final hull. Given this precondition, **chull** can, as the code shows, always divide the set of points in such a way that both subsets also contain multiples of four points.

3.3 Construction of Tetrahedrons

The construction of a tetrahedron in procedure **tetrahull** starts with a call to **make_tetra_edges** takes four points (contiguous in y order) and uses the function **tripscalprod** to calculate the sign of the volume of the parallelepiped defined by the edges connecting three of the points to the fourth. This volume sign is used as a discriminant for the two possible relative orderings of the three edges around the point (see Figure 3). Procedure **exchange_pt** then swaps points **pt**[2] and **pt**[3], if necessary, so that we need only one set of linking operations for both orderings.

The local procedure **make_edge** builds an edge record by assigning the vertices and inserting the relevant edge links for the winged edge representation of the tetrahedron. The code of **tetrahull** follows:

```
procedure tetrahull (i:point_id;     {-----finds 3d tetrahedron
  hull-----} h:polyptr);
  var
    edges:tetra_edges;
    p:tetra_points;

  procedure make_tetra_edges (i:point_id;
                                                {-----construct linked edges-----}
      var p:tetra_points);
    var
      j:one_to_four;
      k:one_to_six;

  procedure make_edge (i:one_to_six;
      iv1, iv2:one_to_four;
      lef1, lef2, rit1, rit2:one_to_six);
      {-----code omitted-----}
  begin     {-----make_tetra_edges-----}
    for j := 1 to 4 do
      p[j] := points[i + j - 1];
    if tripscalprod(p) > 0 then
      exchange_pt(p[2], p[3]);
    for k := 1 to 6 do
      create(edges[k]);
    make_edge(1, 1, 2, 6, 5, 4, 2);
    make_edge(2, 2, 3, 1, 3, 5, 6);
    make_edge(3, 3, 4, 6, 5, 2, 4);
    make_edge(4, 4, 1, 3, 1, 5, 6);
    make_edge(5, 2, 4, 2, 4, 1, 3);
    make_edge(6, 1, 3, 4, 2, 1, 3);
  end;     {-----of make_tetra_edges-----}

begin     {-----tetra_hull-----}
  make_tetra_edges(i, p);
  tetra_hull_2d(edges, p, h);
end;     {-----of tetra_hull-----}
```

Having built the tetrahedron structure, procedure **tetra_hull_2d** then computes the links for its 2-D convex hull using a doubly linked list as shown by the following code:

```
procedure tetra_hull_2d (var edges:tetra_edges;
                                                {-----finds 2d hull of tetra-----}
      var p:tetra_points;
      hull:polyptr);
var
    nhull, rit, i:one_to_four;
    e1, e2:edgeptr;
begin
  order_tetra_points_for_hull(p, nhull, rit);
  e1 := edges[edge_for(1, 2)];
  hull^.lef2d := e1;
```

```
    for i := 2 to nhull do
      begin
        e2 := edges[edge_for(i, (i mod nhull) + 1)];
        e1 ^.fwd2d := e2;
        e2 ^.bwd2d := e1;
        if i = rit then
             hull^.rit2d := e2;
             e1 := e2;
      end;
    e1^.fwd2d := hull^.lef2d;     {-----complete circular linking-----}
        hull^.lef2d^.bwd2d := e1;
  end;    {-----of tetra_hull_2d-----}
```

Procedure **order_tetra_points_for_hull** performs the following:

(1) sorting the four points according to x-coordinate; and

(2) systematically testing, with the Boolean function **onleft**, the relative position of the points within successive triples, rearranging them, if necessary, so that their order defines a clockwise circular list. The function **onleft**, in effect, tests the sign of the area of the x–y projections of the triangle formed by the three points.

```
function onleft (a, b, c:pointptr):boolean;
{-----is b on left of line a to c-----}
  begin
    onleft := (b^.x - a^.x) * (c^ y - a^.y) < (b^.y - a^.y) * (c^.x - a^.x);
  end;
```

Figure 4 illustrates the various possibilities for point alignment and the corresponding transformations into the circular list. Note that in some cases we end up with only three points.

Procedure **tetra_hull_2d** then uses the function **edge_for** to identify the edge records for successive pairs of vertices on the clockwise list and inserts the 2-D links, **fwd2d** and **bwd2d**.

Finally, **tetra_hull_2d** records the initial leftmost and rightmost edges (**lef2d, rit2d**) in the polyhedron structure as start points for the procedures that find supporting lines in the 2-D merge process.

3.4 The 2-D Merge

It is a prerequisite for the main 3-D merging process on a pair of "sub" hulls that at least one of the connecting edges is already established. It is for this purpose that we maintain the two-way edge links that define the so-called 2-D hull, since this includes a connecting edge—in fact, it includes two, one on the left and one on the right. Thus, it is necessary for us to begin the process of constructing the 3-D hull for a pair of "sub" hulls by constructing the corresponding 2-D hull. This task is performed by the procedure **merge_2d**, which we describe in this section before going on to describe the 3-D merge proper in the following section.

Procedure **merge_2d**, whose code follows this subsection, is passed two **polyptrs** for the "sub" hulls (as well as one **polyptr** for the new merged hull). It returns pointers to the left and right supporting edges (**elef** and **erit** in Figure 5) and pointers to the edges at the ends of the right supporting line (**ear** and **ebr** in Figure 5). The latter are needed to start the 3-D edge testing.

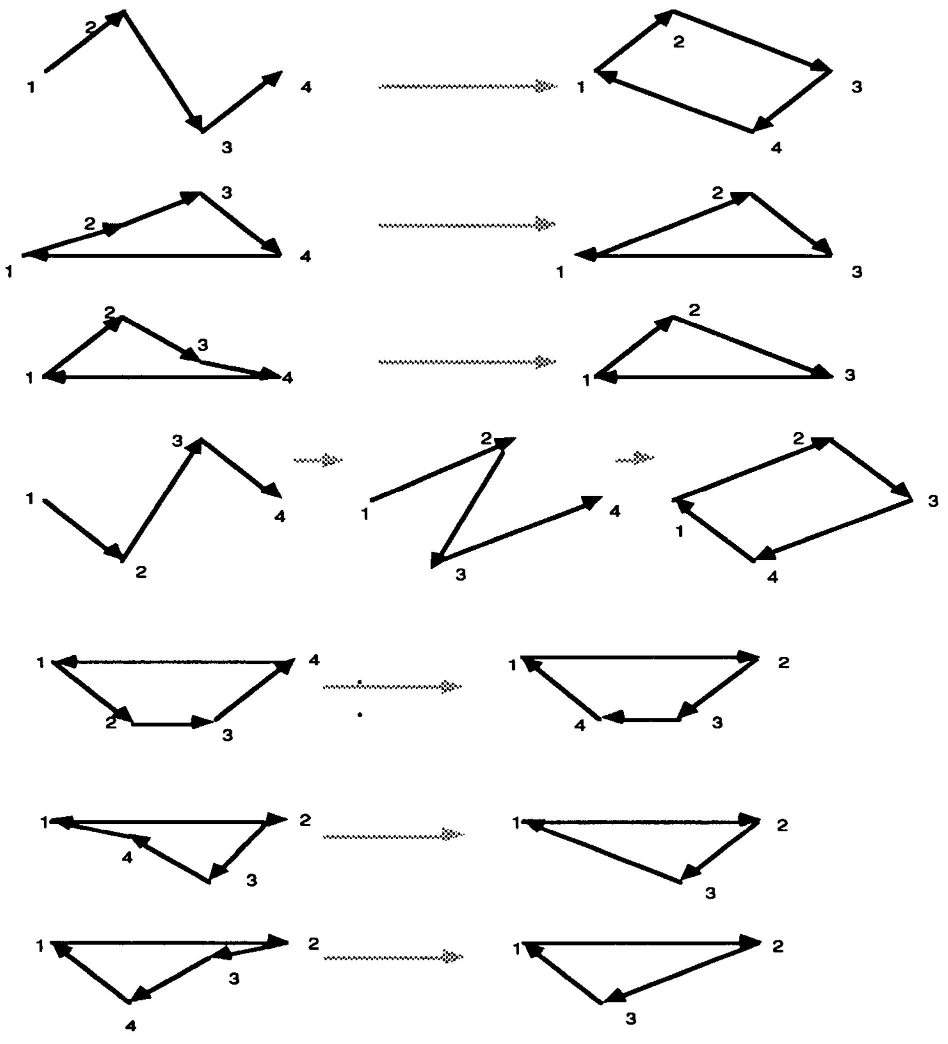


Fig. 4. Point transformations into circular list.

Thus, the central task of **merge_2d** is to find the left and right supporting edges. In each case this is achieved by calls to one of the procedures **sup_line_fwd** and **sup_line_bwd**. In the case of the right supporting line, **(erit) sup_line_fwd** is called when the rightmost point of the upper hull is to the right of the corresponding point in the lower hull, and conversely, **sup_line_bwd** is called when the rightmost point of the upper hull is to the left of the corresponding point in the lower hull. An analogous choice between the two routines is made for the left supporting edge (**elef**). Figure 6 shows two of the six possible cases. At this stage all edges in the 2-D hulls are aligned as shown in Figure 7.

Procedure **sup_line_fwd**, whose code is listed below, starts with two edges supplied by the caller, one on the "front" hull and the other on the "back" hull (**ef** and **eb**, respectively, in Figure 8). The line segment connecting the base

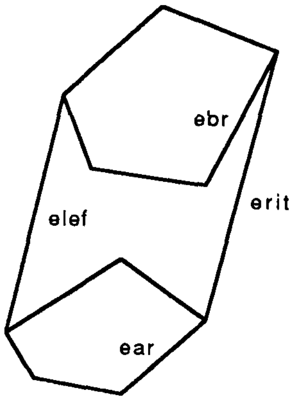


Fig. 5. The left and right supporting lines and the edges marking the end points of **erit**.

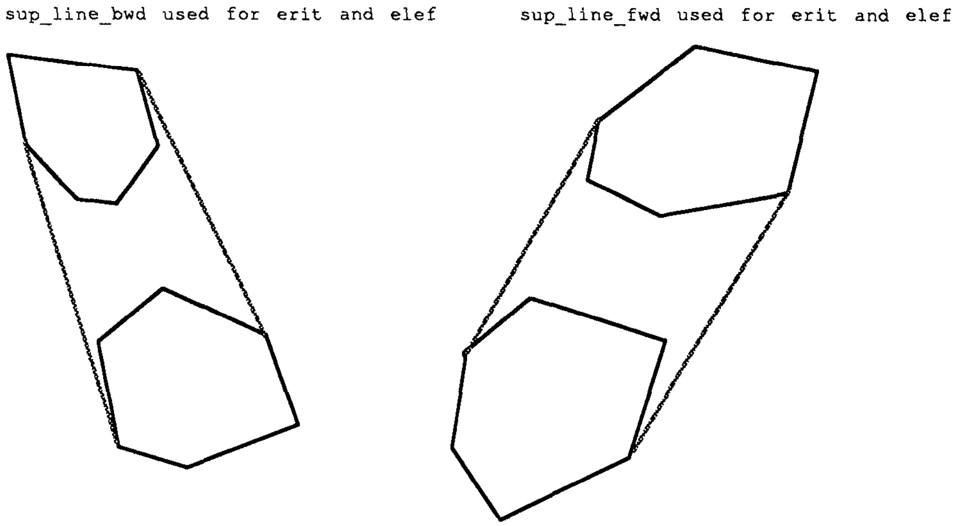
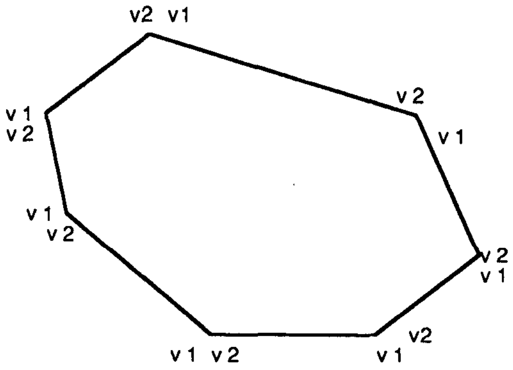

sup_line_bwd used for erit and elef
sup_line_fwd used for erit and elef

Fig. 6. Choice of routine for supporting edge.

Fig. 7. 2-D hull edge alignment.



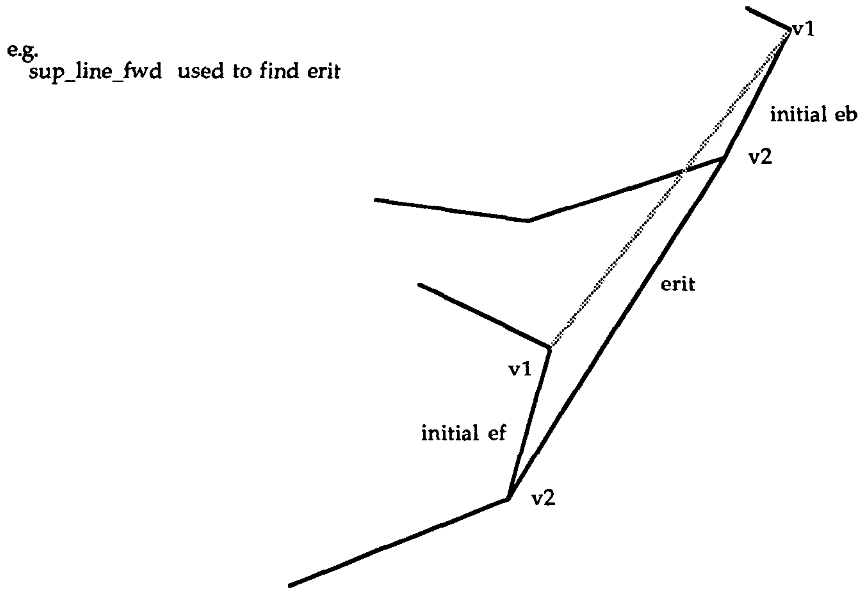


Fig. 8. Use of **sup_line_fwd** to find **erit**.

vertices of these two edges defines a prospective supporting edge. The procedure then attempts to improve upon this prospective edge by repeatedly advancing first the front edge (**ef**) around the front hull and then the back edge (**eb**) around the back hull, using the "hardworking" function **onleft** to test, for each potential advance, whether or not it does indeed give an improvement. A crucial point illustrated in the code is that, for the algorithm to function correctly in all cases, each sequence of **ef** moves must be followed by only a *single* **eb** move before the whole process is repeated.

```
procedure sup_line_fwd (var ef, eb:edgeptr; var backmoved:boolean);
                                        {-----finds supporting tangents lef or rit-----}
                                        {-----polytop lef.x>polybot lef.x-----}
                                        {-----polytop rit.x>polybot rit.x-----}
  var
    improvable: boolean;

begin
  improvable := true;
  backmoved := false;
  while improvable do
    begin
      while onleft(eb^.v1^.v, ef^.v2^.v, ef^.v1^.v) do
      ef := ef^.fwd2d;     {-----advance_the_front-----}
      if onleft(eb^.v1^.v, eb^.v2^.v, ef^.v1^.v) then
        begin
          eb := eb^.fwd2d;     {-----advance_the_back-----}
          backmoved := true;
        end
      else
        improvable := false;
    end;
end;    {-----of sup_line_fwd-----}
```

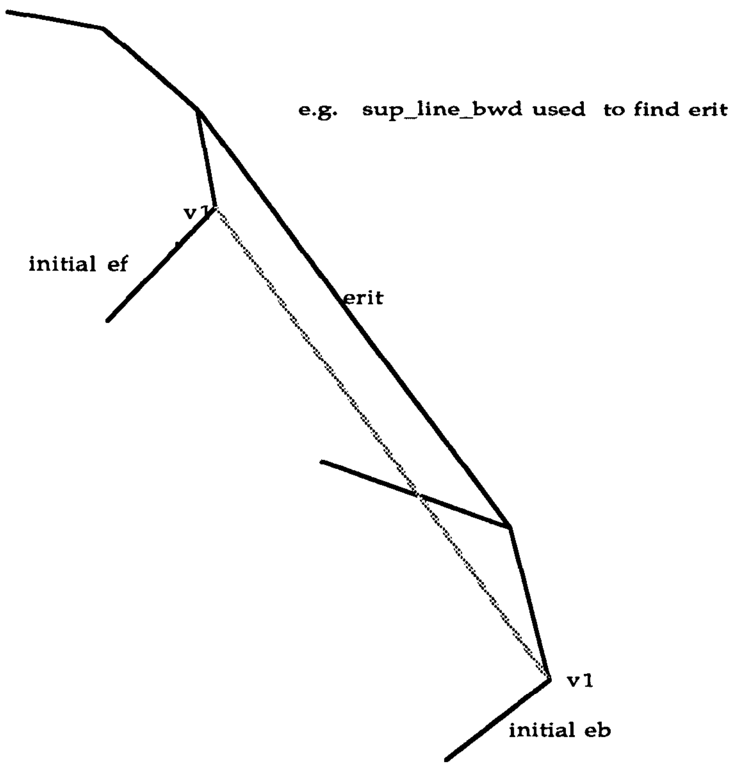


Fig. 9. Use of **sup_line_bwd** to find **elef**.

A note has to be made during this process, using the flag parameter, **back-moved**, of any movement of the back edge since the absence of such movement will mean that the new rightmost (or leftmost) edge will be distinct from the corresponding edge of the original back hull. The function **sel_edge** uses this information to select correctly the edge marking the rightmost or leftmost point.

The code for the procedure **sup_line_bwd** is omitted here, but the situation it deals with is depicted in Figure 9. Essentially the behavior of this procedure is analogous to that of **sup_line_fwd**. However, in this case the advance proceeds in the direction indicated by the **bwd** links. It is also worth noting that in this case it is unnecessary to record any reverse movement of edge **eb** on the back hull as was required in **sup_line_fwd**.

Once **merge_2d** has thus located the two new supporting edges, new edge records are created for them and linked into the existing edge structures to form the new 2-D hull. These edge records are then available for completion as they are reached in the wrapping process. As shown in the code, some care has to be exercised in the 2-D linking, since situations of the kind shown in Figure 10 can arise. In this case, the upper "sub" hull contributes no edges to the new 2-D hull, and so the new edges **elef** and **erit** must be directly interlinked.

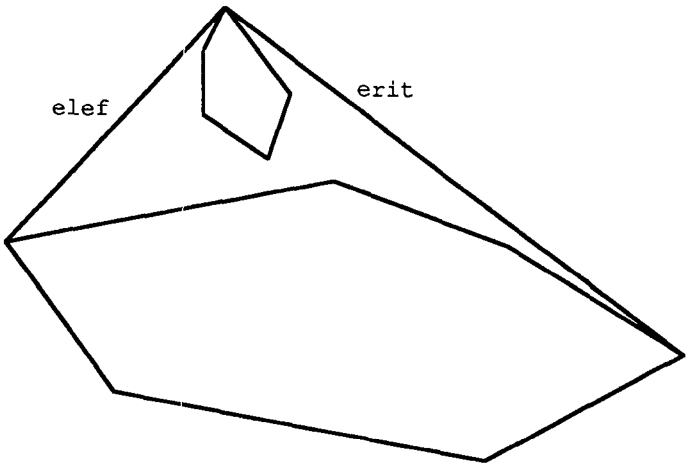


Fig. 10. Example of a special case in 2-D linking.

Code for **merge_2d** now follows:

```
procedure merge_2d (pa, pb : polyptr;
    var pnew : polyptr;
    var redge, ledge : edgeptr;
    var par, pbr : edgeptr);
                    {-----joins two 2d hulls using left, right tangents-----}
                        {-----returns pointers to right tangent ends-----}
  var
    pal, pbl : edgeptr;
    backmoved : boolean;

  function sel_edge (mov : boolean;
      newedge, oldedge : edgeptr) : edgeptr;
  begin
    if mov then
      sel_edge := oldedge
    else
      sel_edge := newedge;
  end;

begin
  create(redge);
  create(ledge);
                    {-----create the new right, left edges required-----}
                              {-----first we find right tangent-----}
  par := pa^.rit2d;
  pbr := pb^.rit2d;
  if (abs_eq(pa^.rit2d^.vl^.v^.x, pb^.rit2d^.vl^.v^.x)) then
    pnew^.rit2d := pbr
  else if (pa^.rit2d^.vl^.v^.x < pb^.rit2d^.vl^.v^.x) then
    begin
      sup_line_fwd(par, pbr, backmoved);
      pnew^.rit2d := sel_edge(backmoved, redge, pb^.rit2d);
    end
    else
      begin
        sup_line_bwd(pbr, par);
        pnew^.rit2d := pa^.rit2d;
      end;    {-----now merge right end of two hulls-----}
    link_edge_2d(redge, pbr^.bwd2d, par);
                                                    {-----then the left-----}
    pal := pa^.lef2d;
    pbl := pb^.lef2d;
    if (abs_eq(pa^.lef2d^.v1^.v^.x, pb^.lef2d^.v1^.v^.x)) then
      begin
        pnew^.lef2d := pbl;
      end
    else if (pa^.lef2d^.v1^.v^.x < pb^.lef2d^.v1^.v^.x) then
      begin
        sup_line_fwd(pbl, pal, backmoved);
            {----- uses bwd to get v2, s inposition}
        pnew^.lef2d := sel_edge(backmoved, ledge, pa^.lef2d);
      end
    else
      begin
        sup_line_bwd(pal, pbl);
        pnew^.lef2d := pb^.lef2d;
      end;
                              {----- now merge left end of two hulls-----}
    link_edge_2d(ledge, pal^.bwd2d, pbl);
                    {-----and test for coincidence of ledge, redge end points-----}
    if (redge^.v1^.v = ledge^.v2^.v) then
          re_link_2d(ledge, redge);
    if (redge^.v2^.v = ledge^.v1^.v) then
        re_link_2d(redge, ledge);
    if (pnew^.rit2d^.v1^.v = ledge^.v1^.v) then
        pnew^.rit2d := ledge;
    if (pnew^.lef2d^.v1^.v = redge^.v1^.v) then
          pnew^.lef2d := ledge;
  end;    {-----of merge_2d-----}
```

3.5 The 3-D Merge by Wrapping

Procedure **merge_3d**, which is responsible for the bulk of the computation in this program, implements the wrapping technique that finds a contiguous sequence of triangular facets connecting a pair of "sub" hulls. Each facet is formed by two connecting edges (**ecurr** and **enext** in Figure 12) and a single edge from either the upper or lower "sub" hull. The essence of the technique is to choose, from all the "sub" hull edges incident upon an end point of **ecurr**, the edge that together with **ecurr** defines a facet making the maximum possible angle with the previous facet.

Given the initial supporting edge **erit**, determined by the 2-D merge, we wish to construct an initial facet with edge **erit** in the role of **ecurr**. To initiate this process, we need a dummy facet against which the prospective edges determining the first real facet can be tested. This dummy facet is determined by edge **erit**, and a third point is chosen to ensure that the resulting facet is (1) outside the new merged hull and (2) makes an angle of less than 180° with any hull facet that has **erit** as one of its sides. The fact that **erit** is the rightmost connecting edge makes it a simple matter to satisfy these conditions. It is sufficient to choose any point such that the resulting dummy facet is parallel to the z-axis. Procedure **make_ref_pt** calculates this "ghost" point, which is somewhere immediately "behind" the connecting edge **erit**.

At this stage the Macintosh SANE library procedure **procentry** is called in order to permit use of the special floating-point value INF (infinity) so that we can handle floating-point overflow consistently (see below for more detail). Also, in this section of code, **rev_edge** is used extensively to ensure that, while testing angles between facets, the correct edge vertex is always used.

To determine the first real facet, we use procedure **first_edge**, which does a complete scan of all the edges around both endpoints (**v1**, **v2**) of **erit**. The parameter **way** controls the direction of movement around a vertex (anticlockwise, **left**, on the bottom hull, and clockwise, **right**, on the top), and procedure **fix_edge** is used to perform any necessary edge reversal.

The two calls to **first_edge** return two pairs of values each identifying the edge that provides the greatest convex angle, **ebest**, and its associated cotangent value, **cotbest**.

The following is the code for **fix_edge** and **first_edge**:

```
procedure fix_edge (var e:edgeptr;
                                              {-----aligns edge .v1 with point v-----}
    v:pointptr);
begin
  if v = e^.v2^.v then
  rev_edge(e);
end;    {-----of fix_edge-----}

procedure first_edge (way: direction;
                                  {-----finds initial best edge in a complete rev.-----}
      c:pointptr;
      ejoin:edgeptr;
        var ebest: edgeptr;
      var cotbest:extended);
  var
    cot:extended;
    edone, e:edgeptr;
    a, b, fix_pt:pointptr;
begin
  a := ejoin^.v1^.v;
  b := ejoin^.v2^.v;
  if way = lef then
        fix_pt := a
  else
    fix_pt := b;
  fix_edge(ebest, fix_pt);
    cotbest := cot_val(a, b, c, ebest^.v2^.v);
    edone := ebest;
    e := advan_edge(ebest, way);
    while e <> edone do
    begin
      fix_edge(e, fix_pt);
      cot := cot_val(a, b, c, e^.v2^.v);
```

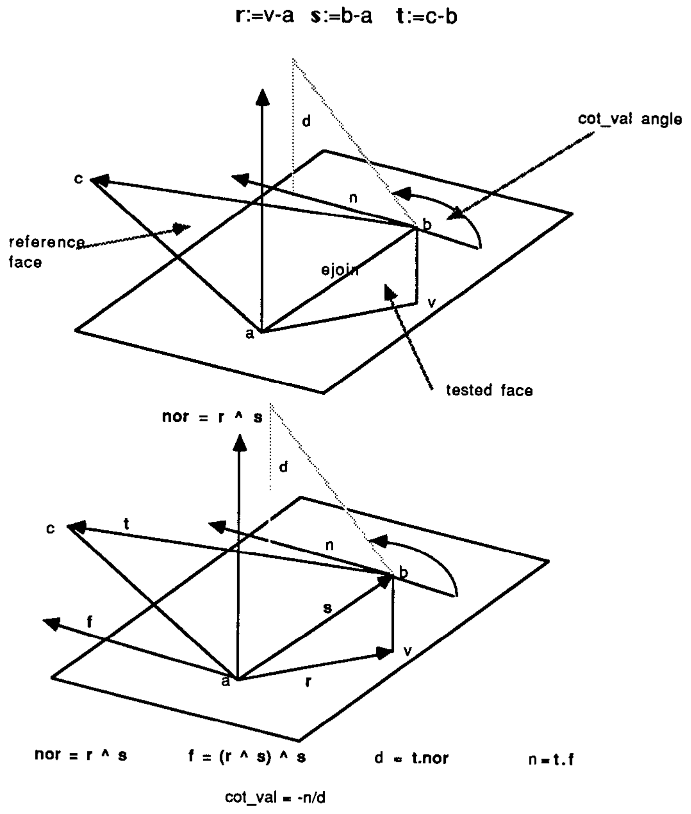


Fig. 11. Calculation of the cotangent of the angle between faces.

```
            if cot < cotbest then
                  begin
                    cotbest := cot;
                    ebest := e;
                  end;
                e := advan_edge(e, way);
        end;
    end;    {-----of first_edge-----}
```

The angles between faces (in fact, we use their cotangents) are calculated with procedure **cot_val** (see Figure 11). The relevant vectors connecting the points a, b, c, and v are constructed as shown below:

The **cot_val** is given by the expression −n/d which is calculated from the relevant vector and scalar products as shown in Figure 11. If the angle gets dangerously close to 0° or 180° (cot value +INF or −INF), where floating-point inaccuracies can give unreliable results we switch to the additional **sign of n** test (n is positive at the 180° end and negative at the 0° end). A consistent test (to the degree of accuracy provided by **eps2**) is thus provided for the two possible alignments of coplanar facets.

The following is the code for **cot_val** and **cotang**:

```
function cotang (a, b, nor : vec) : extended;
                        {-----computes cot for angle between faces comparison-----}
  var
    d, n, ct : extended;

begin
  d := (a.x * nor.x + a.y * nor.y + a.z * nor.z);
  n := (a.x * b.x + a.y * b.y + a.z * b.z);
  ct := -n/d;
  if (abs(ct) > eps2) then
                        {-----checks the inaccuracy at 0 and 180 degrees-----}
    if n < 0 then
      ct := inf
    else
      ct := -inf;
  cotang := ct;
end;    {-----of cotang-----}

function cot_val (a, b, c, v : pointptr) : extended;
{-----see diagram-----}
  var
    r, s, n, t, f : vec;
    cv : extended;
begin
  v_comps(a^, v^, r);
  v_comps(a^, b^, s);
  unit_norm(r, s, n);    {-----vector product-----}
  unit_norm(n, s, f);
  v_comps(b^, c^, t);
  cv := cotang (t, f, n);
      cot_val := cv;
end;    {-----of cotval-----}
```

Once these initial cotangent values have been determined for both hulls, the main repeat loop of **merge_3d** is entered, and subsequent edge scanning is performed in the procedure **best_edge** (of course, for the first time round this is a null scan since the optimum values have already been found using **first_edge**). The choice of start edge for each vertex scan and the convexity of the merging polyhedrons guarantee that the first edge to be reached with minimum **cot_value** will provide the facet with largest angle for its corresponding "sub" hull. In other words, it is unnecessary at each stage to scan all edges around a vertex, as was the case with procedure **first_edge**. Each time round the loop, **best_edge** is called once each for the upper and lower hulls and successively returns the values **cot1**, **1best** and **cot2**, **e2best**. The **cot1**, **cot2** values are compared, and the winning edge is chosen as the next new edge (**enext**).

The code for **best_edge** now follows:

```
procedure best_edge (way : direction;
                        {-----finds edge with best angle-----}
    c : pointptr;
    ejoin : edgeptr;
    var ebest : edgeptr;
    var cotbest : extended);
    var
      cotprev, cot: extended;
      e: edgeptr;
      a, b, fix_pt: pointptr;
      gotbest: boolean;
  begin
    a := ejoin^.v1^.v;
    b := ejoin^.v2^.v;
    if way = lef then
            fix_pt := a
    else
      fix_pt := b;
    fix_edge(ebest, fix_pt);
    cotbest := cot_val(a, b, c, ebest^.v2^.v);
    gotbest := false;
    while (not gotbest) do
            begin
            e := advan_edge(ebest, way);
                fix_edge(e, fix_pt);
                cot := cot_val(a, b, c, e^.v2^.v);
                if cot < cotbest then
                  begin
                    cotbest := cot;
                    ebest := e;
            end
          else
            gotbest := true;
        end;
  end;    {-----of best_edge-----}
```

The process of linking the newly determined edge (**enext**) into the existing data structures needs some care, particularly with regard to the order in which links in the existing "sub" hull structures are updated. We describe here the case where an edge has been chosen from the lower "sub" hull, that is, where **cot1 < cot2**, as shown in Figure 12 (the treatment of the case where an edge has been chosen from the upper hull is analogous). The donkey work is done by the local procedure **link_edge**, which first establishes the edge record (**enext** or **newedge** locally) by creating a new record or using those already established when edges **erit** and **elef** are encountered.

Once found, the **newedge** can be linked "backwards" to the edges **ecurr** and **e1best**, and we can provide the "forward" links from the previous **newedge**, now identified as **ecurr**. Also, having advanced **e1best** to provide the new start edge for the next **best_edge** search, we can insert the **olde1best** left and right links to **enext** and **ecurr**, respectively. As we have not chosen the next best edge from the top polyhedron, the next stage of searching in the top hull can recommence from the current **e2best** since it is not possible for one of the edges passed over in the last scan round this vertex to provide the best edge for the next face. At the final edge to be reached by the merge, **erit**, the procedure **link_edge** inserts the last "forward" links to complete the surface structure.

Finally, in **merge_3d** edge **erit** is reversed (it is the only edge on the 2-D hull whose direction is guaranteed to be known at this stage), and using this as a reference edge, all the 2-D edges are realigned as required for the next 2-D merge

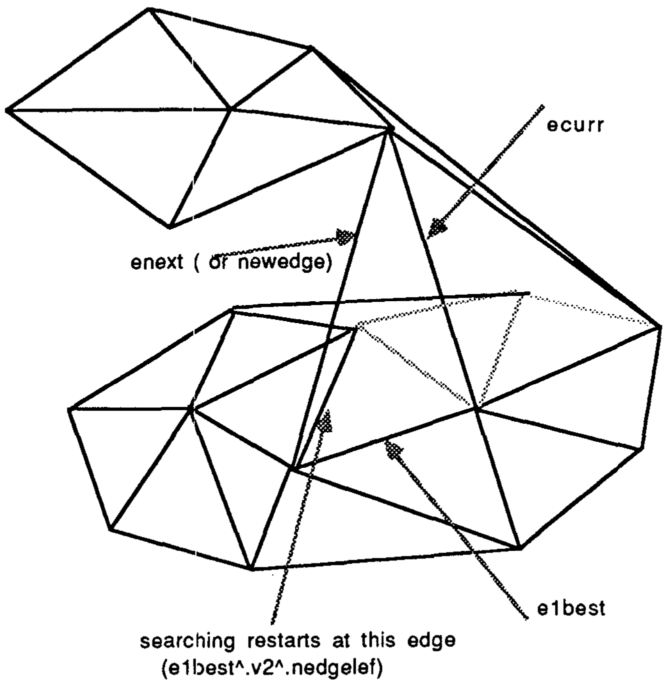


Fig. 12. Edges associated with a wrapping facet.

stage. Procedure **setenvironment** then returns to the ordinary "non-INF" environment.

The code for **merge_3d** now follows:

```
procedure merge_3d (p1, p2, px: polyptr);
  var
    v, c: pointptr;
    ecurr, erit, elef, ear, ebr, e1best, e2best, oldebest, enext: edgeptr;
    cot1, cot2: extended;

  procedure link_edge (var newedge: edgeptr;
      va, vb: pointptr;
      le1, re1, le2, re2: edgeptr);
  begin
    if (va = elef^.v1^.v) and (vb = elef^.v2^.v) then
      newedge := elef
    else if (va = erit^.v1^.v) and (vb = erit^.v2^.v)      then
      begin
        newedge := erit;
        le1 := newedge^.v1^.nedgelef;
        re2 := newedge^.v2^.nedgerit;
      end
    else
      create(newedge);
    with newedge^ do
      begin
        v1^.v := va;
        v2^.v := vb;
```

```
            v1^.nedgelef := le1;
            v1^.nedgerit := re1;
            v2^.nedgelef := le2;
            v2^.nedgerit := re2;
          end;
      end;    {-----of link_edge-----}

  begin
      procentry(env);    {-----to use SANE inf-----}
      merge_2d(p1, p2, px, erit, elef, ear, ebr);
      ecurr := erit;
      e1best := ear;
      e2best := ebr;
      make_ref_pt (c, ecurr);
      rev_edge (ecurr);    {-----point edge up-----}
      first_edge(rit, c, ecurr, e2best, cot2);
      first_edge(lef, c, ecurr, e1best, cot1);
      repeat
        best_edge(lef, c, ecurr, e1best, cot1);
        best_edge(rit, c, ecurr, e2best, cot2);
        if (cot1 < cot2) then
          begin
            link_edge(enext, e1best^.v2^.v, ecurr^.v2^.v, nil, e1best, ecurr, nil);
            ecurr^.v1^.nedgelef := e1best;    {-----more linking-----}
            ecurr^.v2^.nedgerit := enext;
            oldebest := e1best;
            e1best := e1best^.v2^.nedgelef;
            oldebest^.v2^.nedgelef := enext;
            oldebest^.v1^.nedgerit := ecurr;
            c := oldebest^.v1^.v;
          end
        else    {-----cot1 >= cot2)-----}
          begin
            link_edge(enext, ecurr^.v1^.v, e2best^.v2^.v, nil, ecurr, e2best, nil);
            ecurr^.v1^.nedgelef := enext;
            ecurr^.v2^.nedgerit := e2best;
            oldebest := e2best;
            e2best := e2best^.v2^.nedgerit;
            oldebest^.v1^.nedgelef := ecurr;
            oldebest^.v2^.nedgerit := enext;
            c := oldebest^.v1^.v;
          end;
        ecurr = enext;
      until ecurr = erit;
      rev_edge(erit);    {-----realign erit to start 2d merge-----}
      re_align_edges(erit);
  {-----check alignment of v1, v2 in 2d hull edges-----}
  setenvironment(env);
  end;
```

4. OUTPUT OF FACES

Having produced the complete convex hull, as described above, the final section of the program, **list_faces**, is used to output all the faces in the surface data structure. In brief, the algorithm, which is recursive, works by expanding a clockwise closed path that contains the facets already listed. Each increment of the expansion that tends to traverse the surface of the hull by a spiral "route," replaces an edge on the path by the other two edges of the facet based on that edge and on the left of the boundary, thus transferring this facet from the exterior to the interior of the boundary (see Figure 15). If the path already contains the vertex just added, we have in effect a pair of closed subpaths, as shown in Figure 13, which can both be expanded recursively in turn. In most cases, where the new point is not already on the path a single recursive expansion is performed. The recursion terminates when the path consists of just two connecting edges on the same pair of vertices but with opposite directions.

In order to simplify the traversal process, a new edge record, **scan_edge_rec**, is defined. In addition to the pointer to the corresponding edge record, it has links to the previously scanned edge and the next scanned edge on the closed path and also a field containing the index in the original array of points of the trailing vertex **v1**. The following code shows the data structures and the procedure **prep_expand_path** that initializes them:

```
type
  scan_edge_ptr = ^scan_edge_rec;
  scan_edge_rec = record
       polyedge: edgeptr;
       v: point_id;
       prev_se, next_se: scan_edge_ptr;
     end;
  point_edge_map = array[point_id] of scan_edge_ptr;
  face_pts = array[1 .. 3] of point_id;
var
  scan_edge_at: point_edge_map;
  init_path_head, init_path_tail: scan_edge_ptr;

procedure prep_expand_path (initpe: edgeptr;
     var init_head, init_tail: scan_edge_ptr;
     var scan_edge_map: point_edge_map);
  var
    i: point_id;
begin
     for i := 1 to pcount do
     scan_edge_map[i] := nil;
            new(init_head);
            new(init_tail);
            init_head^.polyedge := hull^.lef2d;
            init_tail^.polyedge := hull^.lef2d;
            with init_head^ do
     begin
       v := pt_index(polyedge^.v1^.v);
       scan_edge_at[v] := init_head;
     end;
   with init_tail^ do
     begin
       v := pt_index(polyedge^.v2^.v);
       scan_edge_at[v] := init_tail;
     end;
   join_scan_edges(init_head, init_tail);
   join_scan_edges(init_tail, init_head);
end;     {-----of prep_expandpath-----}
```

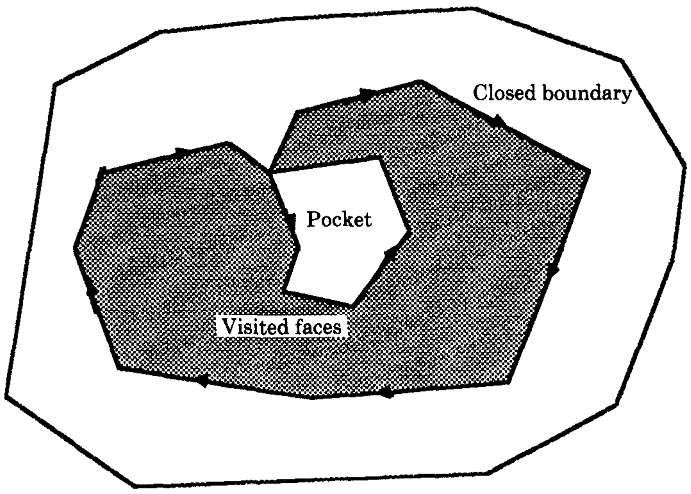
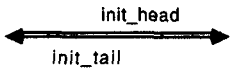


Fig. 13. Expansion of the closed path into itself.



expand_path_over_faces then proceeds to expand the boundary as follows:

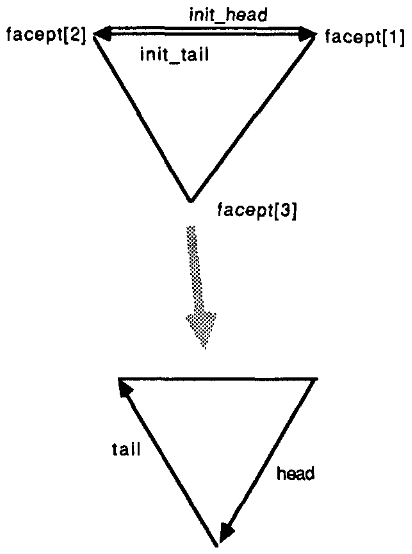


Fig. 14. The initial closed path and first expansion.

Procedure **prep_expand_path** takes as an initial edge the leftmost edge in the 2-D hull (**hull^.lef2d**), initializes all the **scan_edge_map** pointers (which define the edge on the path emanating from the points on the hull), and sets up the initial closed loop based on this single edge, as shown in Figure 14.

The code of the main procedure **expand_path_over_faces** is as follows:

```
procedure expand_path_over_faces (path_head, path_tail: scan_edge_ptr);
  var
    tailpe, newpe1, newpe2: edgeptr;
    edge_pc_pt: scan_edge_ptr;
    facept: face_pts;

  procedure new_head (var head: scan_edge_ptr;
      newpe: edgeptr;
      newpt: point_id);
    var
      newse: scan_edge_ptr;
  begin
    new(newse);
    with newse^ do
      begin
        polyedge := newpe;
        v := newpt;
      end;
    join_scan_edges(head, newse);
    scan_edge_at[newpt] := newse;
    head := newse;
  end;    {-----of new_head-----}

  procedure adjust_tail (tail: scan_edge_ptr;
      newpe: edgeptr;
      newpt: point_id);
  begin
    with tail^ do
      begin
        polyedge := newpe;
        v := newpt;
      end;
    scan_edge_at[newpt] := tail;
  end;    {-----of adjust_tail-----}

    begin    {-----expand_path_over_faces-----}
      facept[1] := path_tail^.v;
      facept[2] := path_tail^.next_se^.v;
      tailpe := path_tail^.polyedge;
      fix_edge(tailpe, points[facept[1]]);
      newpe1 := tailpe^.v1^.nedgelef;
      newpe2 := tailpe^.v2^.nedgerit;
      fix_edge(newpe1, points[facept[1]]);
      facept[3] := pt_index (newpe1^.v2^.v);
      out_face(facept);
      new_head(path_head, newpe1, facept[1]);
      if (scan_edge_at[facept[3]]) < > nil then
        begin
          edge_pc_pt := scan_edge_at[facept[3]]^.prev_se;
          expand_left_of_pt(path_head, facept[3]);
          path_head := edge_pc_pt;
        end;
  adjust_tail(path_tail, newpe2, facept[3]);
  join_scan_edges(path_head, path_tail);
  elim_null_cct(path_head, path_tail);
  if path_head < > nil then
        expand_path_over_faces(path_head, path_tail);
end;    {-----of expand_path_over_faces-----}
```

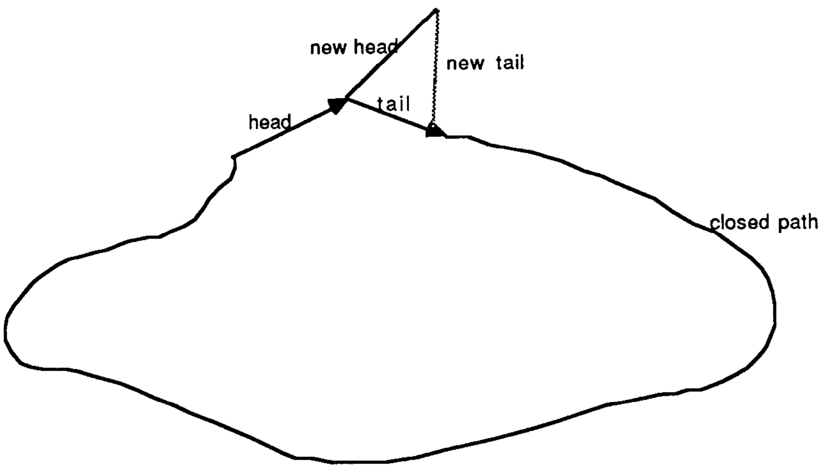


Fig. 15. Boundary expansion.

Procedures **new_head** and **adjust_tail** establish new **scan_edge_recs** and expand the boundary as shown in Figure 15.

In order to deal effectively with pockets, the code

```
if (scan_edge_at[facept[3]]) < > nil
```

tests for cases such as that shown in Figure 16.

In Figure 16, the condition **scan_edge_at[facept [3]] < > nil** holds since the point **facept[3]** is already on the boundary. The previous edge to this contact point, **edge_pc_pt**, is saved, and procedure **expand_left_of_pt**, shown below, is used to expand the left-hand closed subpath.

```
procedure expand_left_of_pt(mainhead: scan_edge_ptr;
      newpt: point_id);
    var
      lefhead, leftail: scan_edge_ptr;
  begin
    join_scan_edges(mainhead, scan_edge_at[newpt]);
    lefhead := mainhead^.prev_se;
    leftail := mainhead;
    elim_null_cct(lefhead, leftail);
    if lefhead < > nil then
          expand_path_over_faces(lefhead, leftail);
  end;    {-----of expand_left_of_pt-----}
```

This procedure once again uses **expand_path_over_faces**, having established the edges, **lefhead** and **leftail**, using the old head and the edge behind it. Once this is complete, **edge_pc_pt** becomes **pathhead**, and the tail is adjusted as shown above so that the pocket can be traversed using **expand_path_over_faces** again.

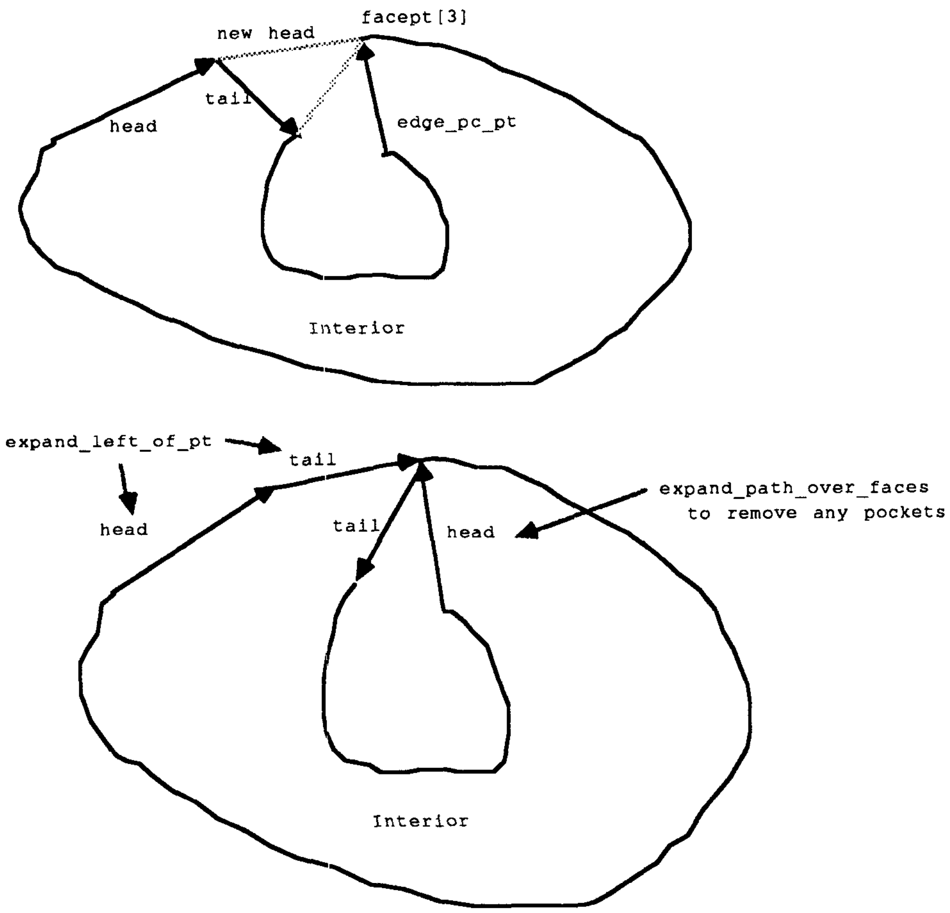


Fig. 16. Dealing with a pocket in a closed path.

The procedure **elim_null_cct**, whose code is shown below, checks for the case of coincident head and tail edges:

```
procedure elim_null_cct (var head, tail: scan_edge_ptr);
  begin
    if head^.polyedge = tail^.polyedge then
      begin
        if head^.prev_se = tail then
        begin
                head := nil;
                tail := nil;
        end;
      end;
end;    {-----of elim_null_cct-----}
```

In this case, the fact that a pair of closed subpaths is always generated, immediately edge self-contact is detected, guarantees that the closed path under consideration consists of these two coincident edges above. The exterior of the path is therefore null and recursion terminates.

5. TEST PROCEDURES

Testing of the complete version of this program was carried out in three stages:

Stage 1. The program was run on small sets of points whose convex hulls had been previously calculated by hand. The computed hull was displayed, in order to immediately detect any obvious errors, and face vertices were output using the "spiral" path method.

Stage 2. Potentially difficult sets of points were devised and tested as above. Such sets included cases where the merging of subhulls would leave a single vertex or edge from one subhull in the merged hull. Figure 17 illustrates a typical example.

Stage 3. This was carried out with procedures that took each face in turn and systematically checked that all other original points were on the same side of the plane defined by the face. To do this, a new origin was created interior to the convex hull, and all points were transformed to make this their new origin. It was then a relatively simple but computationally heavy task to test every point using the sign of the function $F(x)$ where

$$F(\underline{x}) = \underline{x}.\,(\text{p1} \times \text{p2} + \underline{\text{p2}} \times \underline{\text{p3}} + \underline{\text{p3}} \times \underline{\text{p1}}) - [\underline{\text{p1}}, \underline{\text{p2}}, \underline{\text{p3}}]$$

and $\underline{x}$ is the point tested, $\underline{\text{p1}}$, $\underline{\text{p2}}$, $\underline{\text{p3}}$ are the face vertices, $\times$ is the vector product, and $[-, -, -]$ is the triple vector product.

The output procedures access the face vertices on a consistent order, which means that for correct points the sign of $F(x)$ does not vary from face to face.

For the test data, points were randomly generated within various ranges and run using single, double, and extended representation and arithmetic. The SANE implementation for the Macintosh showed no decrease in speed when using extended; in fact, it always ran faster, but of course, this behavior depends on the particular SANE implementation. The hull produced was consistently found to be correct using extended with an **eps2** value (used in the **cot_val** calculations) of 1E16 and using random data from a uniform distribution in a variety of ranges. However, when using singles, and to a lesser extent doubles, it was necessary to reduce the value of **eps2**, and hence, the accuracy of angle comparisons as faces approached the coplanar case. Failure to do this caused errors that exhibited themselves in a variety of forms, such as pointer dereferencing zero, incorrect hulls, and infinite looping. In general, the value of **eps2** would be governed by the range, magnitude, and number of significant digits used in the test data and also by the machine characteristics.

In addition to the tests described above, separate units of the code, such as the tetrahull section, the 2-D merge section, and the **cot_val** section, were individually tested with random points and potentially difficult cases. In particular, the 2-D merge required careful design of test data to ensure that cases such as "one vertex left from upper or lower hull" or "deletion of edges previously labeled as

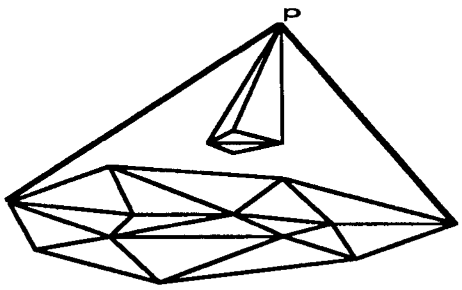


Fig. 17. A single vertex left in a merging "subhull."

the leftmost or rightmost in the hull" were handled correctly. The behavior of the program was also monitored for point sets that were known to contain exact or nearly exact coplanar faces, in order to ascertain the limits of accuracy for a reliable hull.

6. CONCLUSIONS

As expected, the implementation of this algorithm has not been a trivial exercise, but it has raised several interesting questions that are not apparent in the original papers. As is often the case with geometric algorithms, the original publication tends to give a rather concise strategic outline that emphasizes the complexity analysis but glosses over implementation issues.

The problems of numerical accuracy and the inconsistency of floating-point arithmetic have been encountered and partially solved using multiple-precision floating-point arithmetic and "alternative" computation techniques, as shown in the **cot_val** computation.

Some difficulty was experienced in producing a reliable 2-D merge. Initially, a method, similar to that shown in the original paper, was used to compute and compare the slopes of edges from the upper hull and lower hull with the slope of the connecting edges. It was found, however, that this method did not cope efficiently with all possible cases and needed extra "messy" code to deal with some special cases. The **onleft** method finally chosen is not only straightforward and efficient, but also handles all "nasty" cases.

Another interesting situation, not mentioned in the original algorithm but clearly described by Edelsbrunner [6], is the case where a vertex appears more than once on the path followed by the upper (or lower) boundary edges of the wrapping faces (see Figure 18). It would appear that such a case would require $O(n^2)$ merge time.

The implementation deals effectively with all such cases and illustrates why it is crucial to carry out certain operations, such as the linking and scanning of edges in **merge_3d**, in the correct order. Another example is the case where just a single vertex from the top (or bottom) hull is left in the merged hull (see Figure 17). The investigation of such potentially awkward situations requires careful design of the input data and careful testing of the complete final hull. It was found that this consumed a high proportion of development time.

One disadvantage of the edge-based representation is the necessity to test the direction of and, if necessary, realign every edge as it is scanned. The method of providing pointers to **vertexinfo** provides a means of reducing the cost of this

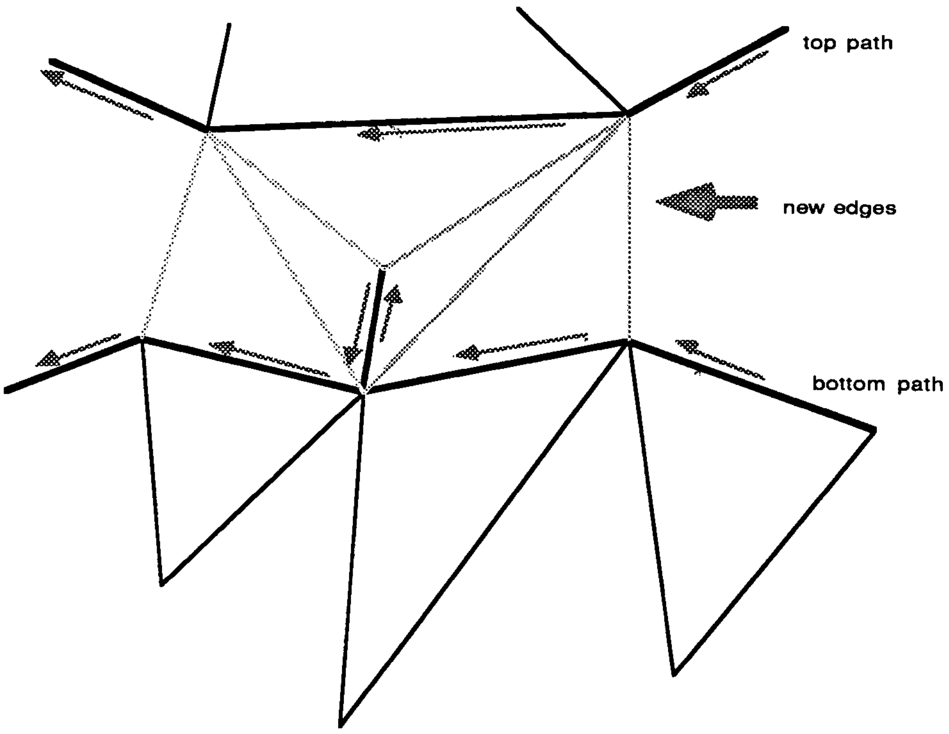


Fig. 18. A vertex appearing more than once on the lower path.

operation. Also, in its present form the program does not discard records for edges that become hidden. This could, of course, cause a problem if memory is limited and a very large number of points are used; however, the removal of discarded records is not as trival an operation as would at first appear, and the current version therefore relies on an abundance of memory.

Another significant feature of this algorithm, not covered in detail in previous literature, is the construction of a 3-D convex hull at the lowest level of recursion. In order to produce concise and efficient code for this section, some care was taken in its development. We have chosen to stop at four points and use a tetrahedron to start the merge process.

The program generates triangular faces even when the eventual face is non-triangular (coplanar triangular faces). This is not a problem, however, since the resulting redundant edges could easily be detected on construction and, if necessary, removed later.

The extensive test procedures carried out on the code subsections and on the whole program have shown that, within the bounds previously discussed, the implementation is robust and correct.

Future work on this program will incorporate a perturbation technique (see, e.g., Edelsbrunner [6]) and will hopefully solve the degeneracy problem without a significant decrease in efficiency. The floating-point arithmetic problem is a different matter since any method that guarantees "true" results is bound to give a significant increase in the execution time of the program. Rational, Interval, and Level Index arithmetics are some of the possibilities that could be used. The other objective is a fast parallel implementation using a transputer system, and the algorithm's potential for adaptation to that environment is to be investigated.

ACKNOWLEDGMENTS

The author would like to thank Robin Forrest and Ralph Elliot for their encouragement and advice on this paper.

REFERENCES

1. ANSALDI, S., DE FLORIANI, L., AND FALCIDIENO, B. Geometric modelling of solid objects by using a face adjacency graph representation. *Comput. Graph. 19*, 3 (1985), 131–139.
2. APPLE COMPUTER, INC. *Macintosh Pascal Technical Appendix*, Apple Computer, Inc., 1984, D 1–42.
3. BAUMGART, B. G. Geometric modelling for computer vision. Stanford Artif. Intell. Lab. Rep. STAN-CS-74-463, Computer Science Dept., Stanford Univ., Stanford, Calif., 1974.
4. BRAID, I. Notes on a geometric modeller. C.A.D. Group Doc. 101, Univ. of Cambridge, U.K., June 1979.
5. DAY, A. M. The implementation of an algorithm to find the convex hull of a three dimensional set of points. Memo CGP 87/5, Dept. of Computing Science, Univ. of East Anglia, Norwich, U.K., 1987.
6. EDELSBRUNNER, H. *Algorithms in Combinatorial Geometry*. Springer-Verlag, New York, 1987.
7. GUIBAS, L. J., AND STOLFI, J. Primitives for the manipulation of general subdivisions and the computation of Voronoi diagrams. *ACM Trans. Graph. 4*, 2 (Apr. 1985), 74–123.
8. PREPARATA, F. P., AND HONG, S. J. Convex hull of a finite set of points in two and three dimensions. *Commun. ACM 20*, 2 (Feb. 1977), 87–93.
9. PREPARATA, F. P., AND SHAMOS, M. I. *Introduction to Computational Geometry*. Springer-Verlag, New York, 1985.

Received March 1988; revised November 1988; accepted November 1988

Editor: Martti Mäntylä