An Implicit Surface Polygonizer

Jules Bloomenthal
The University of Calgary
Calgary, Alberta T2N 1N4 Canada

An algorithm for the polygonization of implicit surfaces is described and an implementation in C is provided. The discussion reviews implicit surface polygonization, and compares various methods.

Introduction

Some shapes are more readily defined by implicit, rather than parametric, techniques. For example, consider a sphere centered at C with radius r. It can be described parametrically as $\{P\}$, where:

$$(P_x, P_y, P_z) = (C_x, C_y, C_z)+(r\cos\beta\cos\alpha, r\cos\beta\sin\alpha, r\sin\beta),\ \alpha \in (0, 2\pi),\ \beta \in (-\pi/2, \pi/2).$$

The implicit definition for the same sphere is more compact:

$$(P_x-C_x)^2+(P_y-C_y)^2+(P_z-C_z)^2-r^2 = 0.$$

Because an implicit representation does not produce points by substitution, root-finding must be employed to render its surface. One such method is ray tracing, which can generate excellent images of implicit surfaces. Alternatively, an image of the function (not surface) can be created with volume rendering.

Polygonization is a method whereby a polygonal (*i.e.*, parametric) approximation to the implicit surface is created from the implicit surface function. This allows the surface to be rendered with conventional polygon renderers; it also permits non-imaging operations, such as positioning an object on the surface. Polygonization consists of two principal steps. First, space is partitioned into adjacent cells at whose corners the implicit surface function is evaluated; negative values are considered inside the surface, positive values outside. Second, within each cell, the intersections of cell edges with the implicit surface are connected to form one or more polygons.

In this gem we present software that performs spatial partitioning and polygonal approximation. We hope this software, which includes a simple test application, will encourage experimentation with implicit surface modeling.

The implementation is relatively simple (about 400 lines, ignoring comments, the test application, and the cubical polygonization option). Some of this simplicity derives from the use of the cube as the *partitioning cell*; its symmetries provide a simple means to compute and index corner locations. We do not employ automatic techniques (such as interval analysis) to set polygonization parameters (such as cell size); these are set by the user, who also must judge the success of the polygonization. This not only simplifies the implementation, but permits the implicit surface function to be treated as a 'black box.' The function, for example, can be procedural or, even, discontinuous (although discontinuous functions may produce undesirable results). The use of a fixed resolution (*i.e.*, unchanging cell size) also simplifies the implementation, which explains the popularity of fixed resolution over adaptive resolution methods.[1] This makes the choice of cell size important: too small a size produces too many polygons; too large a size obscures detail.

Before listing the code, we'll describe its operation and features. These include:

· fixed resolution partitioning by continuation,

[1] [Bloomenthal], [Moore] and [Snyder] provide descriptions of adaptive polygonization.

· automatic surface detection,
· root-finding by binary subdivision,
· unambiguous triangulated output in points-polygon format, and
· function evaluation caching

Overview

The spatial partitioning implemented here is based on the continuation scheme presented by [Wyvill *et al*], in which an initial cube is centered on a surface point (the 'starting point'). Continuation consists of generating new cubes across any face that contains corners of opposite polarity (of the implicit surface function); this process continues until the entire surface is contained by the collection of cubes. The surface within each cube is then approximated by one or more polygons. Unfortunately, some polarity combinations are ambiguous; the 'marching cubes' method produces errant holes in the surface because it treats these ambiguities inconsistently.

The implementation presented here treats all cube cases consistently, in one of two user-selectable ways. Either the cube is directly polygonized according to an algorithm given in [Bloomenthal], or it is decomposed into tetrahedra that are then polygonized, as suggested by [Payne and Toga]. Thus, either a cube or a tetrahedron serves as the *polygonizing cell*. Each edge of the polygonizing cell that connects corners of differing polarity is intersected with the surface; we call these surface/edge intersections 'surface vertices.' When connected together, they form polygons.

The continuation, decomposition, and polygonization steps are illustrated below.

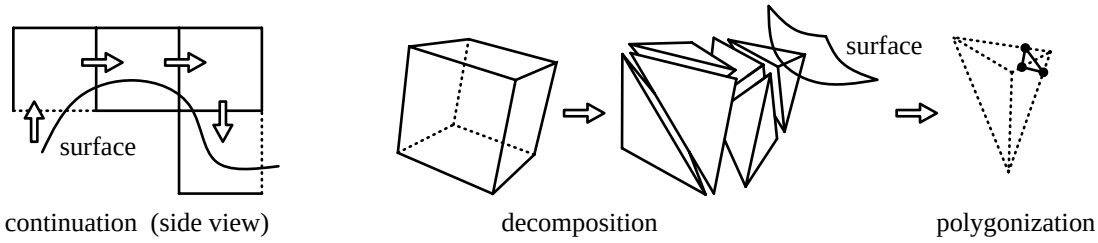

Figure 1: overview of the polygonizer.

Continuation versus Exhaustive Search

Continuation methods require $O(n^2)$ function evaluations, where n is some measure of the size of the object (thus, n^2 corresponds to the object's surface area). Methods that employ exhaustive search through a given volume require $O(n^3)$ function evaluations. We know of only one other published polygonization implementation, which is given in [Watt]; it employs a 'marching cubes' exhaustive search.

One benefit of exhaustive search is its detection of all pieces of a set of disjoint surfaces. This is not guaranteed with continuation methods, which require a starting point for each disjoint surface. With the implementation provided here, a surface is automatically detected by random search; thus, only a single object is detected and polygonized. If, however, the client provides a starting point for the partitioning, random search is not performed and disjoint objects may be polygonized by repeated calls to the polygonizer, each with a different starting point.

Root-Finding

Most exhaustive search methods were designed to process three-dimensional arrays (*i.e.*, discrete samples such as produced by *CAT* or *MRI* scans); the present polygonizer, however, was designed for objects defined by a continuous, real-valued function. Such functions allow the location of a surface

vertex to be computed accurately, rather than approximated by linear interpolation as is commonly employed with discrete arrays. The affects of interpolation can be seen in the following figure.

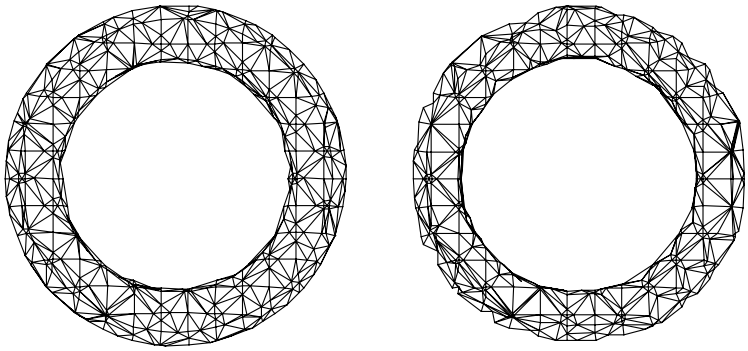

Figure 2: surface vertex computation: accurate (left) and interpolated (right).

Binary subdivision is a reliable and simple method to compute accurately the surface vertex location; given one point inside the surface and one outside, the subdivision converges to a point on the surface. Binary subdivision can, at times, be more efficient than other convergence methods, such as *regula falsi*. In our implementation, the number of iterations is fixed (at 10); the client can modify this constant or pass it to the program as a variable.

In our implementation, the number of function evaluations is held to a minimum. The location of a surface vertex is computed only once; it is cached and subsequently indexed according to the coordinates of its edge endpoints, using a hashing technique reported in [Wyvill *et al*]. Function values at cube corners are cached similarly. The overhead in storing function values may exceed the cost of additional evaluations for very simple functions; for complex functions, however, the elimination of redundant evaluation is a significant advantage.

Although designed for continuous functions, the polygonizer can process three-dimensional arrays if the cube size is set to the sampling resolution of the array, and the real-valued coordinates are converted to indices into the array. In this case, binary subdivision isn't possible, and linear interpolation must be used to estimate the surface vertex locations. Alternatively, the implicit surface function can provide a tri-linear interpolation of the discrete samples in the neighborhood of the query point (*i.e.*, the argument to the function). This will work well for functions that are locally linear. Below is pseudo-code that produces an interpolated value given a three-dimensional array of evenly spaced values and a query point *P* within the range *Min*, *Max* of the array. The array resolution in three dimensions is given by *Res*.

```
ValueFromDiscreteArray (values, P, Min, Max, Res)
  if P not in (Min..Max)
    then error(OutOfRange)
    else begin
      x: real ← (P_x-Min_x)*(Res_x-1)/(Max_x-Min_x);
      y: real ← (P_x-Min_y)*(Res_y-1)/(Max_y-Min_y);
      z: real ← (P_z-Min_z)*(Res_z-1)/(Max_z-Min_z);
      v000: real ← values[⌊x⌋][⌊y⌋][⌊z⌋];          (first of eight corner values)
      v001: real ← values[⌊x⌋][⌊y⌋][⌊z⌋+1];
      v010: real ← values[⌊x⌋][⌊y⌋+1][⌊z⌋];
      v011: real ← values[⌊x⌋][⌊y⌋+1][⌊z⌋+1];
      v100: real ← values[⌊x⌋+1][⌊y⌋][⌊z⌋];
      v101: real ← values[⌊x⌋+1][⌊y⌋][⌊z⌋+1];
      v110: real ← values[⌊x⌋+1][⌊y⌋+1][⌊z⌋];
```

```
v111: real ← values[⌊x⌋+1][⌊y⌋+1][⌊z⌋+1];      (last of 8 values)
v00: real ← v000+frac(z)*(v001-v000);          (interpolate along x0y0 edge)
v01: real ← v010+frac(z)*(v011-v010);          (interpolate along x0y1 edge)
v10: real ← v100+frac(z)*(v101-v100);          (interpolate along x1y0 edge)
v11: real ← v110+frac(z)*(v111-v110);          (interpolate along x1y1 edge)
v0: real ← v00+frac(y)*(v01-v00);              (interpolate along x0 face)
v1: real ← v10+frac(y)*(v11-v10);              (interpolate along x1 face)
return[v0+frac(x)*(v1-v0)];                    (tri-linearly interpolated value)
end;
```

Polygonization

Polygonization of an individual cell is performed using a table that contains one entry for each of the possible configurations of the cell vertex polarities. For the tetrahedron, this is a 16 entry table; for the cube, it is a 256 entry table.

The tetrahedral configurations are shown below; each configuration produces either nothing, a triangle, or a quadrilateral (*i.e.*, two triangles). In the figure below, the elements of the set (denoted by braces) represent the edges of the tetrahedron that contain a surface vertex.

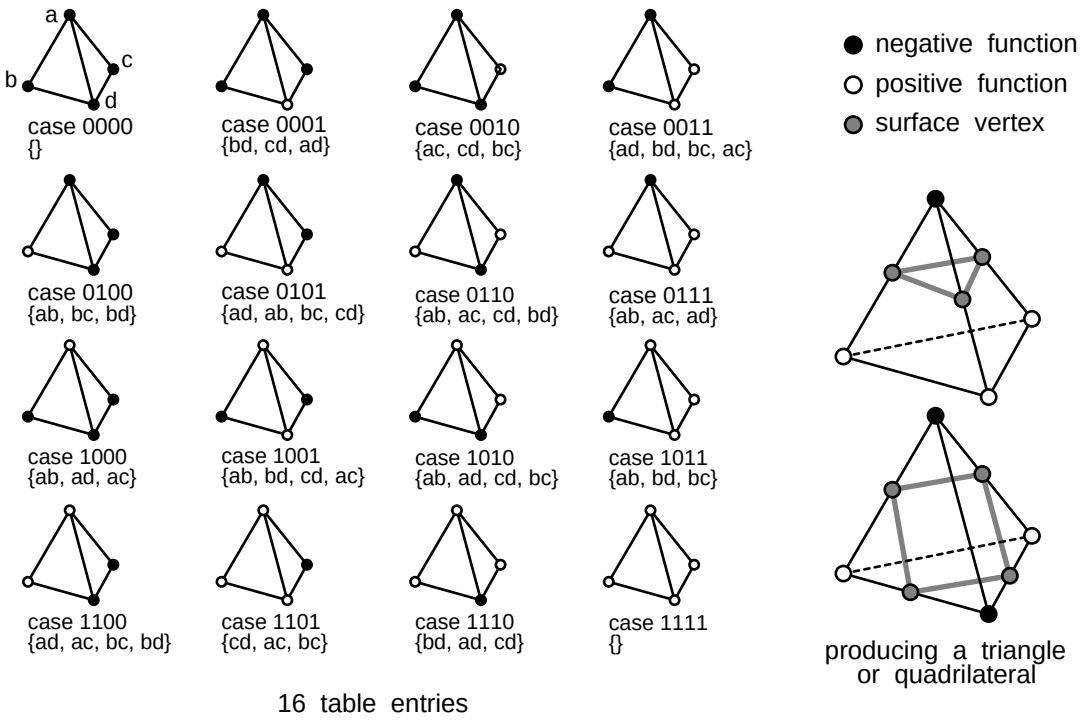

Figure 3: tetrahedral polygonization.

Although the tetrahedral table may be generated by inspection, the greater number of configurations for the cube motivates an algorithmic generation, which is illustrated below. For each of the possible 256 polarity configurations, a surface vertex is presumed to exist on those edges connecting differently signed vertices. The surface vertices are ordered by proceeding from one surface vertex to the next, around a face in clockwise order; upon arriving at a new vertex, the face across the vertex's edge from the current face becomes the new current face.

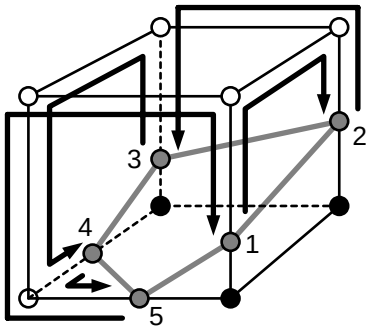

Figure 4: cubical polygonization.

Because the tetrahedral edges include the diagonals of the cube faces, the tetrahedral decomposition yields a greater number of surface vertices per surface area than does cubical polygonization, as shown below. Cubical polygonization requires less computation than does tetrahedral decomposition and polygonization, but requires a more complex implementation.

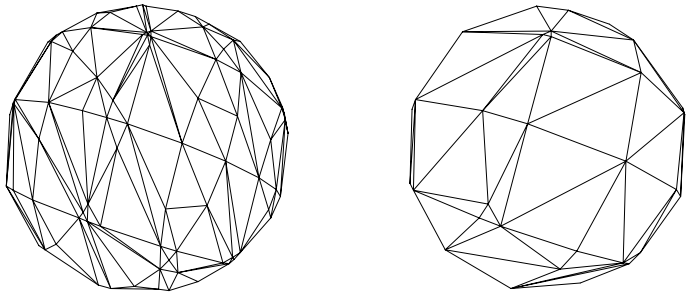

Figure 5: polygonization with tetrahedral decomposition (left) and without (right).

Some implementations produce surface vertices on a per polygon basis, duplicating a vertex whenever it appears in an adjoining polygon. A more efficient format, known as 'points/polygons,' is to list the vertices separately, and to define each polygon as a set of indices into the vertex list. This format often is more convenient for polygon renderers. Our implementation supports this format, producing polygons first, then vertices.

Client Use

The client of this software calls the procedure polygonize(), passing an arbitrary implicit surface function (in the test application provided, the function torus() is used); polygonize() will produce a set of triangles that approximates the implicit surface. In our test application, a 'triangleProc' updates a pointer to the (ever-growing) array of vertices, and copies the indices for the triangle onto an (ever-growing) array of triangles. The triangleProc can also be used to indicate a user-specified abort and to provide a graphical display of the polygonization progress. The polygonizer computes the normal at each surface vertex as an approximation to the function gradient. Presently, each vertex is a structure containing the position and normal of the vertex; the client may wish to add other fields, such as color.

The client may wish to experiment with reversing the normal direction and/or the triangle orientation, in order to suit a particular modeling or rendering environment. As presently implemented, triangles are produced (in a left-handed coordinate system) with vertices in counter-clockwise order when viewed from the out (positive) side of the object; surface normals will point outwards. Inverting the sign of the implicit surface function will reverse the normals and triangle orientation. If these are

incorrect with respect to the rendering system, then algorithms that rely on back-facing polygon tests will fail. For example, reversing the polygon orientation for the torus shown below, left, causes the 'inside' polygons to be displayed, resulting in the improbable looking torus on the right.

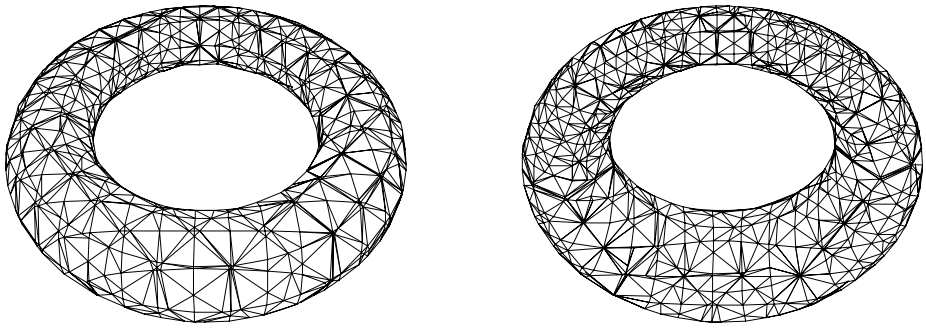

Figure 6: correctly and incorrectly oriented polygons.

Function Considerations

Many implicitly defined objects, such as the tori below, are bounded. The tori are both of the form $(x^2+y^2+z^2+R^2-r^2)^2-4R^2(x^2+y^2)$, where R and r are the major and minor radii (in this example, r = R/4). To achieve a rotation and offset, the lower torus is defined by $(x^2+(y+R)^2+z^2+R^2-r^2)^2-4R^2((y+R)^2+z^2)$. The right side of the figure depicts an equi-potential surface, *i.e.*, those points for which the torus functions are equal. This surface is not bounded and demonstrates the need to limit the polygonizer during propagation. In our implementation, the client sets the parameter *bounds*, which restricts the propagation in all six (left, right, above, below, near, and far) directions from the starting cube. For the equi-potential surface below, *bounds* is 7.

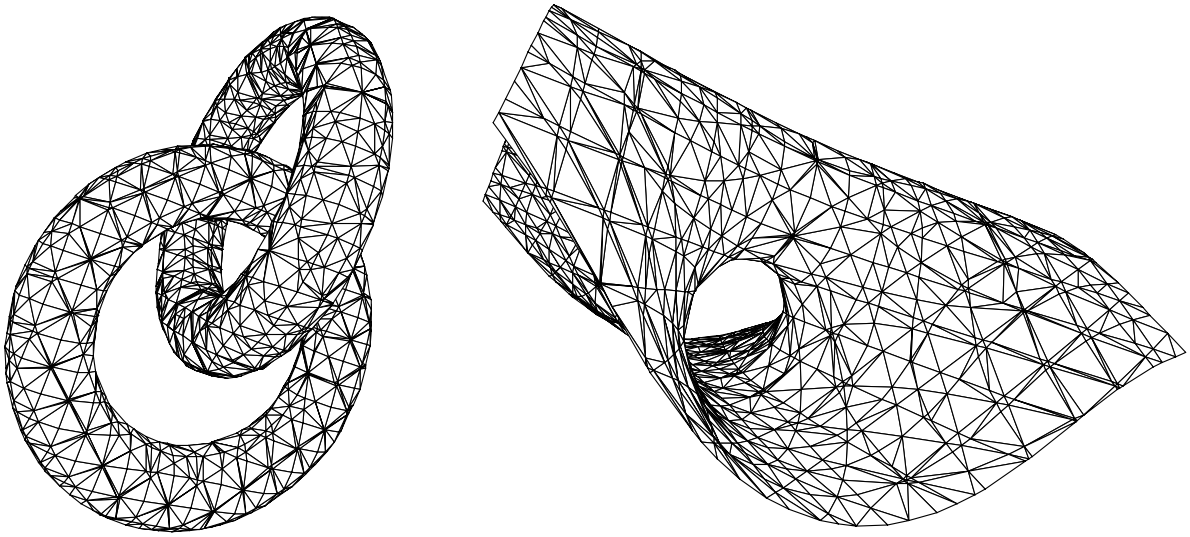

Figure 7: *torus r us*: two tori and their equi-potential surface.

There is no limit to the complexity of the implicit surface function. For example, the following object, which resembles a piece used in the game of jacks, is defined by:

$$(1/(x^2/9+4y^2+4z^2)^4+1/(y^2/9+4x^2+4z^2)^4+1/(z^2/9+4y^2+4x^2)^4+$$
$$1/((4x/3-4)^2+16y^2/9+16z^2/9)^4+1/((4x/3+4)^2+16y^2/9+16z^2/9)^4+$$

$$1/((4y/3-4)^2+16x^2/9+16z^2/9)^4+1/((4y/3+4)^2+16x^2/9+16z^2/9)^4)^{-1/4}-1$$

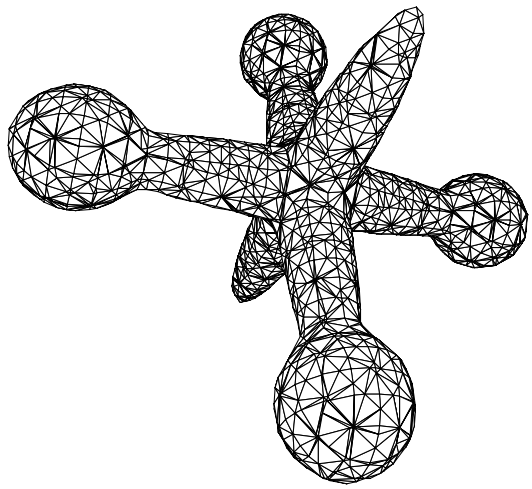

Figure 8: a jack.

Although its complexity is unrestricted, the implicit surface function should be C^1 continuous; otherwise the object that results may be incomplete, have truncated edges, or contain small jutting pieces. This latter artifact is demonstrated by the 'wiffle cube,' a rounded cube with a sphere removed; it is defined by: $1-(a^2x^2+a^2y^2+a^2z^2)^{-6}-(b^8x^8+b^8y^8+b^8z^8)^6$, with a = 1/2.3 and b = 1/2. A sharply edged wedge occurs along each circular opening of the cuboid.

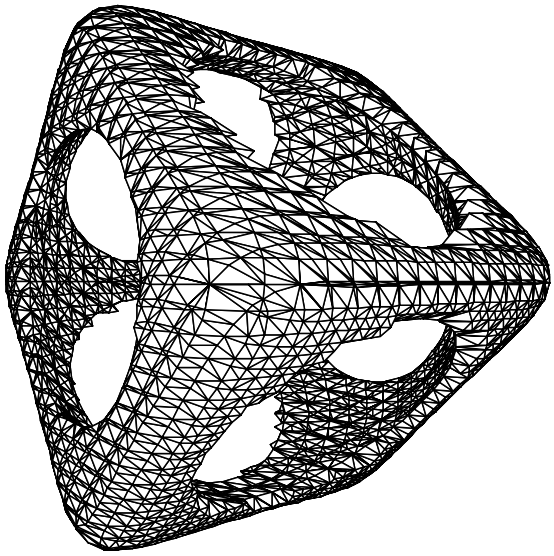

Figure 9: the 'wiffle cube.'

Here the disadvantage of a fixed grid resolution is evident; higher resolution is desirable along the sharp edges. The jutting pieces visible along the circular openings are due to the chance occurrence that a polygonizing cell edge intersects the wedge close to its center. A surface vertex calculated for this edge will lie very close to the center of the wedge; it will appear shifted with respect to neighboring vertices, which will lie along a wider part of the wedge.

Despite this difficulty (which is due to the inherently numerical nature of polygonizers), we believe the expanded range of modeling afforded by implicit surfaces holds great promise, and that polygonization is an important tool for implicit design. Happy modeling!

Acknowledgements

Many thanks to Paul Haeberli, Paul Heckbert, and Mark Hammel for assistance while at C; to Stewart Dickson for his comments; to Mark Ganter (ganter@u.washington.edu), who provided the jack, equi-potential tori, and wiffle cube functions; and, especially, to the editor, who carefully catalogued numerous facets in need of polishing.

References

Bloomenthal, J., Polygonization of Implicit Surfaces. Computer Aided Geometric Design, 5, 4 (November 1988).

Moore, D., Simplicial Mesh Generation with Applications, Computer Science Ph.D. dissertation, Cornell University (TR 92-1322), 1992.

Payne, B. and Toga, A., Surface Mapping Brain Function on 3D Models. IEEE Computer Graphics and Applications (September 1990).

Snyder, J., *Generative Modeling for Computer Graphics and CAD*, Academic Press, Boston, Mass, 1992.

Watt, A. and Watt, M., *Advanced Animation and Rendering Techniques*, Addison-Wesley, Reading, Mass, 1993.

Wyvill, G., McPheeters, C., and Wyvill, B. Data Structure for Soft Objects. The Visual Computer, 2, 4 (August 1986).

```
/* implicit.c
 *     an implicit surface polygonizer, translated from Mesa
 *     applications should call polygonize()
 *
 * to compile a test program for ASCII output:
 *     cc implicit.c -o implicit -lm
 *
 * to compile a test program for display on an SGI workstation:
 *     cc -DSGIGFX implicit.c -o implicit -lgl_s -lm
 *
 * Authored by Jules Bloomenthal, Xerox PARC.
 * Copyright (c) Xerox Corporation, 1991.  All rights reserved.
 * Permission is granted to reproduce, use and distribute this code for
  *  any  and  all  purposes,  provided  that  this  notice  appears  in  all
copies. */

#include <math.h>
#include <stdio.h>
#include <sys/types.h>

#define TET     0  /* use tetrahedral decomposition */
#define NOTET   1  /* no tetrahedral decomposition  */

#define RES     10 /* # converge iterations    */

#define L       0  /* left direction:   -x, -i */
#define R       1  /* right direction:  +x, +i */
#define B       2  /* bottom direction: -y, -j */
#define T       3  /* top direction:    +y, +j */
#define N       4  /* near direction:   -z, -k */
#define F       5  /* far direction:    +z, +k */
#define LBN     0  /* left bottom near corner  */
#define LBF     1  /* left bottom far corner   */
#define LTN     2  /* left top near corner     */
#define LTF     3  /* left top far corner      */
#define RBN     4  /* right bottom near corner */
#define RBF     5  /* right bottom far corner  */
#define RTN     6  /* right top near corner    */
#define RTF     7  /* right top far corner     */

/* the LBN corner of cube (i, j, k), corresponds with location
 * (start.x+(i-.5)*size, start.y+(j-.5)*size, start.z+(k-.5)*size) */

#define RAND()      ((rand()&32767)/32767.)  /* random number between 0
and 1 */
#define HASHBIT     (5)
#define HASHSIZE     (size_t)(1<<(3*HASHBIT)) /* hash table size (32768)
*/
#define MASK        ((1<<HASHBIT)-1)
#define    HASH(i,j,k)
```

```
((((((i)&MASK)<<HASHBIT)|((j)&MASK))<<HASHBIT)|((k)&MASK))
#define BIT(i, bit) (((i)>>(bit))&1)
#define FLIP(i,bit) ((i)^1<<(bit)) /* flip the given bit of i */

typedef struct point {            /* a three-dimensional point */
    double x, y, z;               /* its coordinates */
} POINT;

typedef struct test {                /* test the function for a signed
value */
    POINT p;                      /* location of test */
    double value;                 /* function value at p */
    int ok;                       /* if value is of correct sign */
} TEST;

typedef struct vertex {           /* surface vertex */
    POINT position, normal;       /* position and surface normal */
} VERTEX;

typedef  struct  vertices  {                  /*  list  of  vertices  in
polygonization */
    int count, max;               /* # vertices, max # allowed */
    VERTEX *ptr;                  /* dynamically allocated */
} VERTICES;

typedef struct corner {           /* corner of a cube */
    int i, j, k;                    /* (i, j, k) is index within lattice
*/
    double x, y, z, value;        /* location and function value */
} CORNER;

typedef struct cube {             /* partitioning cell (cube) */
    int i, j, k;                  /* lattice location of cube */
    CORNER *corners[8];           /* eight corners */
} CUBE;

typedef struct cubes {               /* linked list of cubes acting as
stack */
    CUBE cube;                    /* a single cube */
    struct cubes *next;           /* remaining elements */
} CUBES;

typedef struct centerlist {       /* list of cube locations */
    int i, j, k;                  /* cube location */
    struct centerlist *next;      /* remaining elements */
} CENTERLIST;

typedef struct cornerlist {       /* list of corners */
    int i, j, k;                  /* corner id */
    double value;                 /* corner value */
    struct cornerlist *next;      /* remaining elements */
```

```
} CORNERLIST;

typedef struct edgelist {          /* list of edges */
    int i1, j1, k1, i2, j2, k2;    /* edge corner ids */
    int vid;                       /* vertex id */
    struct edgelist *next;         /* remaining elements */
} EDGELIST;

typedef struct intlist {           /* list of integers */
    int i;                         /* an integer */
    struct intlist *next;          /* remaining elements */
} INTLIST;

typedef struct intlists {          /* list of list of integers */
    INTLIST *list;                 /* a list of integers */
    struct intlists *next;         /* remaining elements */
} INTLISTS;

typedef struct process {           /* parameters, function, storage */
    double (*function)();          /* implicit surface function */
    int (*triproc)();              /* triangle output function */
    double size, delta;            /* cube size, normal delta */
    int bounds;                    /* cube range within lattice */
    POINT start;                   /* start point on surface */
    CUBES *cubes;                  /* active cubes */
    VERTICES vertices;             /* surface vertices */
    CENTERLIST **centers;          /* cube center hash table */
    CORNERLIST **corners;          /* corner value hash table */
    EDGELIST **edges;              /* edge and vertex id hash table */
} PROCESS;

void *calloc();
char *mycalloc();


/**** A Test Program ****/


/* torus: a torus with major, minor radii = 0.5, 0.1, try size = .05 */

double torus (x, y, z)
double x, y, z;
{
    double x2 = x*x, y2 = y*y, z2 = z*z;
    double a = x2+y2+z2+(0.5*0.5)-(0.1*0.1);
    return a*a-4.0*(0.5*0.5)*(y2+z2);
}


/* sphere: an inverse square function (always positive) */
```

```
double sphere (x, y, z)
double x, y, z;
{
    double rsq = x*x+y*y+z*z;
    return 1.0/(rsq < 0.00001? 0.00001 : rsq);
}


/* blob: a three-pole blend function, try size = .1 */

double blob (x, y, z)
double x, y, z;
{
    return 4.0-sphere(x+1.0,y,z)-sphere(x,y+1.0,z)-sphere(x,y,z+1.0);
}


#ifdef    SGIGFX
/*************************************************************/

#include "gl.h"

/* triangle: called by polygonize() for each triangle; set SGI lines */

triangle (i1, i2, i3, vertices)
int i1, i2, i3;
VERTICES vertices;
{
    float v[3];
    int i, ids[3];
    ids[0] = i1;
    ids[1] = i2;
    ids[2] = i3;
    bgnclosedline();
    for (i = 0; i < 3; i++) {
        POINT *p = &vertices.ptr[ids[i]].position;
        v[0] = p->x; v[1] = p->y; v[2] = p->z;
        v3f(v);
    }
    endclosedline();
    return 1;
}


/* main: call polygonize() with torus function
 * display lines on SGI */

main ()
{
    char *err, *polygonize();
```

```
    keepaspect(1, 1);
    winopen("implicit");
    doublebuffer();
    gconfig();
    perspective(450, 1.0/1.0, 0.1, 10.0);
    color(7);
    clear();
    swapbuffers();
    makeobj(1);
     if ((err = polygonize(torus, .05, 20, 0.,0.,0., triangle, TET)) !=
NULL) {
        fprintf(stderr, "%s\n", err);
        exit(1);
    }
    closeobj();
    translate(0.0, 0.0, -2.0);
    pushmatrix();
    while(1) { /* spin the object */
        reshapeviewport();
        color(7);
        clear();
        color(0);
        callobj(1);
        rot(0.8, 'x');
        rot(0.3, 'y');
        rot(0.1, 'z');
        swapbuffers();

    }
}

#else
/*************************************************************************
*/

int gntris;          /* global needed by application */
VERTICES gvertices;  /* global needed by application */


/* triangle: called by polygonize() for each triangle; write to stdout
*/

triangle (i1, i2, i3, vertices)
int i1, i2, i3;
VERTICES vertices;
{
    gvertices = vertices;
    gntris++;
    fprintf(stdout, "%d %d %d\n", i1, i2, i3);
    return 1;
}
```

```
/* main: call polygonize() with torus function
 * write points-polygon formatted data to stdout */

main ()
    {
    int i;
    char *err, *polygonize();
    gntris = 0;
    fprintf(stdout, "triangles\n\n");
     if ((err = polygonize(torus, .05, 20, 0.,0.,0., triangle, TET)) !=
NULL) {
        fprintf(stdout, "%s\n", err);
        exit(1);
        }
          fprintf(stdout,   "\n%d   triangles,   %d   vertices\n",   gntris,
gvertices.count);
    fprintf(stdout, "\nvertices\n\n");
    for (i = 0; i < gvertices.count; i++) {
        VERTEX v;
        v = gvertices.ptr[i];
        fprintf(stdout, "%f  %f  %f\t%f  %f  %f\n",
            v.position.x, v.position.y,  v.position.z,
            v.normal.x,   v.normal.y,    v.normal.z);
    }
           fprintf(stderr,   "%d   triangles,   %d   vertices\n",   gntris,
gvertices.count);
    exit(0);
}

#endif
/*****************************************************************
/


/**** An Implicit Surface Polygonizer ****/


/* polygonize: polygonize the implicit surface function
 *   arguments are:
 *       double function (x, y, z)
 *               double x, y, z (an arbitrary 3D point)
 *           the implicit surface function
 *           return negative for inside, positive for outside
 *       double size
 *           width of the partitioning cube
 *       int bounds
 *              max. range of cubes (+/- on the three axes) from first
cube
 *       double x, y, z
```

```
 *           coordinates of a starting point on or near the surface
 *           may be defaulted to 0., 0., 0.
 *       int triproc (i1, i2, i3, vertices)
 *               int i1, i2, i3 (indices into the vertex array)
 *               VERTICES vertices (the vertex array, indexed from 0)
 *           called for each triangle
 *           the triangle coordinates are (for i = i1, i2, i3):
 *               vertices.ptr[i].position.x, .y, and .z
 *           vertices are ccw when viewed from the out (positive) side
 *               in a left-handed coordinate system
 *           vertex normals point outwards
 *           return 1 to continue, 0 to abort
 *       int mode
 *           TET: decompose cube and polygonize six tetrahedra
 *           NOTET: polygonize cube directly
 *   returns error or NULL
 */

char *polygonize (function, size, bounds, x, y, z, triproc, mode)
double (*function)(), size, x, y, z;
int bounds, (*triproc)(), mode;
{
    PROCESS p;
    int n, noabort;
    CORNER *setcorner();
    TEST in, out, find();

    p.function = function;
    p.triproc = triproc;
    p.size = size;
    p.bounds = bounds;
    p.delta = size/(double)(RES*RES);

    /* allocate hash tables and build cube polygon table: */
       p.centers  =  (CENTERLIST  **)  mycalloc(HASHSIZE,sizeof(CENTERLIST
*));
       p.corners  =  (CORNERLIST  **)  mycalloc(HASHSIZE,sizeof(CORNERLIST
*));
      p.edges  =    (EDGELIST    **)  mycalloc(2*HASHSIZE,sizeof(EDGELIST
*));
    makecubetable();

    /* find point on surface, beginning search at (x, y, z): */
    srand(1);
    in = find(1, &p, x, y, z);
    out = find(0, &p, x, y, z);
    if (!in.ok || !out.ok) return "can't find starting point";
    converge(&in.p, &out.p, in.value, p.function, &p.start);

    /* push initial cube on stack: */
    p.cubes = (CUBES *) mycalloc(1, sizeof(CUBES)); /* list of 1 */
```

```
    p.cubes->cube.i = p.cubes->cube.j = p.cubes->cube.k = 0;
    p.cubes->next = NULL;

    /* set corners of initial cube: */
    for (n = 0; n < 8; n++)
            p.cubes->cube.corners[n] = setcorner(&p, BIT(n,2), BIT(n,1),
BIT(n,0));

    p.vertices.count = p.vertices.max = 0; /* no vertices yet */
    p.vertices.ptr = NULL;

    setcenter(p.centers, 0, 0, 0);

    while (p.cubes != NULL) { /* process active cubes till none left */
        CUBE c;
        CUBES *temp = p.cubes;
        c = p.cubes->cube;

        noabort = mode == TET?
               /* either decompose into tetrahedra and polygonize: */
               dotet(&c, LBN, LTN, RBN, LBF, &p) &&
               dotet(&c, RTN, LTN, LBF, RBN, &p) &&
               dotet(&c, RTN, LTN, LTF, LBF, &p) &&
               dotet(&c, RTN, RBN, LBF, RBF, &p) &&
               dotet(&c, RTN, LBF, LTF, RBF, &p) &&
               dotet(&c, RTN, LTF, RTF, RBF, &p)
               :
               /* or polygonize the cube directly: */
               docube(&c, &p);
        if (! noabort) return "aborted";

        /* pop current cube from stack */
        p.cubes = p.cubes->next;
        free((char *) temp);
        /* test six face directions, maybe add to stack: */
        testface(c.i-1, c.j, c.k, &c, L, LBN, LBF, LTN, LTF, &p);
        testface(c.i+1, c.j, c.k, &c, R, RBN, RBF, RTN, RTF, &p);
        testface(c.i, c.j-1, c.k, &c, B, LBN, LBF, RBN, RBF, &p);
        testface(c.i, c.j+1, c.k, &c, T, LTN, LTF, RTN, RTF, &p);
        testface(c.i, c.j, c.k-1, &c, N, LBN, LTN, RBN, RTN, &p);
        testface(c.i, c.j, c.k+1, &c, F, LBF, LTF, RBF, RTF, &p);
    }
    free_all(&p);
    return NULL;
}


/* free_all: free all the memory we've allocated (except cubetable) */

free_all (p)
PROCESS *p;
```

```
/* courtesy Paul Heckbert */
{
    int i, index;
    CORNERLIST *l, *lnext;
    CENTERLIST *cl, *clnext;
    EDGELIST *edge, *edgenext;
    INTLISTS *lists, *listsnext;
    INTLIST *ints, *intsnext;

    for (index = 0; index < HASHSIZE; index++) {
  for (l = p->corners[index]; l; l = lnext) {
      lnext = l->next;
      free(l);                  /* free CORNERLIST */
  }
  for (cl = p->centers[index]; cl; cl = clnext) {
      clnext = cl->next;
      free(cl);                 /* free CENTERLIST */
  }
  for (edge = p->edges[index]; edge; edge = edgenext) {
      edgenext = edge->next;
      free(edge);               /* free EDGELIST */
  }
    }
     free(p->edges);                          /* free array of EDGELIST
pointers */
     free(p->corners);                       /* free array of CORNERLIST
pointers */
     free(p->centers);                       /* free array of CENTERLIST
pointers */
    free(p->vertices.ptr);             /* free VERTEX array */

    for (i = 0; i < 256; i++) {        /* free the cube table */
  for (lists = cubetable[i]; lists; lists = listsnext) {
      listsnext = lists->next;
      for (ints = lists->list; ints; ints = intsnext) {
   intsnext = ints->next;
   free(ints);    /* free INTLIST */
      }
      free(lists);    /* free INTLISTS */
  }
  cubetable[i] = 0;
    }
}

/* testface: given cube at lattice (i, j, k), and four corners of face,
 * if surface crosses face, compute other four corners of adjacent cube
 * and add new cube to cube stack */

testface (i, j, k, old, face, c1, c2, c3, c4, p)
CUBE *old;
PROCESS *p;
```

```
int i, j, k, face, c1, c2, c3, c4;
{
    CUBE new;
    CUBES *oldcubes = p->cubes;
    CORNER *setcorner();
    static int facebit[6] = {2, 2, 1, 1, 0, 0};
       int  n,  pos  =  old->corners[c1]->value  >  0.0  ?  1  :  0,  bit  =
facebit[face];

      /*  test  if  no  surface  crossing,  cube  out  of  bounds,  or  already
visited: */
    if ((old->corners[c2]->value > 0) == pos &&
        (old->corners[c3]->value > 0) == pos &&
        (old->corners[c4]->value > 0) == pos) return;
     if (abs(i) > p->bounds || abs(j) > p->bounds || abs(k) > p->bounds)
return;
    if (setcenter(p->centers, i, j, k)) return;

    /* create new cube: */
    new.i = i;
    new.j = j;
    new.k = k;
    for (n = 0; n < 8; n++) new.corners[n] = NULL;
    new.corners[FLIP(c1, bit)] = old->corners[c1];
    new.corners[FLIP(c2, bit)] = old->corners[c2];
    new.corners[FLIP(c3, bit)] = old->corners[c3];
    new.corners[FLIP(c4, bit)] = old->corners[c4];
    for (n = 0; n < 8; n++)
        if (new.corners[n] == NULL)
                  new.corners[n]  =  setcorner(p,  i+BIT(n,2),  j+BIT(n,1),
k+BIT(n,0));

    /*add cube to top of stack: */
    p->cubes = (CUBES *) mycalloc(1, sizeof(CUBES));
    p->cubes->cube = new;
    p->cubes->next = oldcubes;
}


/* setcorner: return corner with the given lattice location
   set (and cache) its function value */

CORNER *setcorner (p, i, j, k)
int i, j, k;
PROCESS *p;
{
    /* for speed, do corner value caching here */
    CORNER *c = (CORNER *) mycalloc(1, sizeof(CORNER));
    int index = HASH(i, j, k);
    CORNERLIST *l = p->corners[index];
    c->i = i; c->x = p->start.x+((double)i-.5)*p->size;
```

```
    c->j = j; c->y = p->start.y+((double)j-.5)*p->size;
    c->k = k; c->z = p->start.z+((double)k-.5)*p->size;
    for (; l != NULL; l = l->next)
        if (l->i == i && l->j == j && l->k == k) {
            c->value = l->value;
            return c;
            }
    l = (CORNERLIST *) mycalloc(1, sizeof(CORNERLIST));
    l->i = i; l->j = j; l->k = k;
    l->value = c->value = p->function(c->x, c->y, c->z);
    l->next = p->corners[index];
    p->corners[index] = l;
    return c;
}


/* find: search for point with value of given sign (0: neg, 1: pos) */

TEST find (sign, p, x, y, z)
int sign;
PROCESS *p;
double x, y, z;
{
    int i;
    TEST test;
    double drand48();
    double range = p->size;
    test.ok = 1;
    for (i = 0; i < 10000; i++) {
        test.p.x = x+range*(RAND()-0.5);
        test.p.y = y+range*(RAND()-0.5);
        test.p.z = z+range*(RAND()-0.5);
        test.value = p->function(test.p.x, test.p.y, test.p.z);
        if (sign == (test.value > 0.0)) return test;
        range = range*1.0005; /* slowly expand search outwards */
    }
    test.ok = 0;
    return test;
}


/**** Tetrahedral Polygonization ****/


/* dotet: triangulate the tetrahedron
 * b, c, d should appear clockwise when viewed from a
 * return 0 if client aborts, 1 otherwise */

int dotet (cube, c1, c2, c3, c4, p)
CUBE *cube;
int c1, c2, c3, c4;
```

```
PROCESS *p;
{
    CORNER *a = cube->corners[c1];
    CORNER *b = cube->corners[c2];
    CORNER *c = cube->corners[c3];
    CORNER *d = cube->corners[c4];
    int index = 0, apos, bpos, cpos, dpos, e1, e2, e3, e4, e5, e6;
    if (apos = (a->value > 0.0)) index += 8;
    if (bpos = (b->value > 0.0)) index += 4;
    if (cpos = (c->value > 0.0)) index += 2;
    if (dpos = (d->value > 0.0)) index += 1;
     /* index is now 4-bit number representing one of the 16 possible
cases */
    if (apos != bpos) e1 = vertid(a, b, p);
    if (apos != cpos) e2 = vertid(a, c, p);
    if (apos != dpos) e3 = vertid(a, d, p);
    if (bpos != cpos) e4 = vertid(b, c, p);
    if (bpos != dpos) e5 = vertid(b, d, p);
    if (cpos != dpos) e6 = vertid(c, d, p);
      /* 14 productive tetrahedral cases (0000 and 1111 do not yield
polygons */
    switch (index) {
        case 1:  return p->triproc(e5, e6, e3, p->vertices);
        case 2:  return p->triproc(e2, e6, e4, p->vertices);
        case 3:  return p->triproc(e3, e5, e4, p->vertices) &&
                        p->triproc(e3, e4, e2, p->vertices);
        case 4:  return p->triproc(e1, e4, e5, p->vertices);
        case 5:  return p->triproc(e3, e1, e4, p->vertices) &&
                        p->triproc(e3, e4, e6, p->vertices);
        case 6:  return p->triproc(e1, e2, e6, p->vertices) &&
                        p->triproc(e1, e6, e5, p->vertices);
        case 7:  return p->triproc(e1, e2, e3, p->vertices);
        case 8:  return p->triproc(e1, e3, e2, p->vertices);
        case 9:  return p->triproc(e1, e5, e6, p->vertices) &&
                        p->triproc(e1, e6, e2, p->vertices);
        case 10: return p->triproc(e1, e3, e6, p->vertices) &&
                        p->triproc(e1, e6, e4, p->vertices);
        case 11: return p->triproc(e1, e5, e4, p->vertices);
        case 12: return p->triproc(e3, e2, e4, p->vertices) &&
                        p->triproc(e3, e4, e5, p->vertices);
        case 13: return p->triproc(e6, e2, e4, p->vertices);
        case 14: return p->triproc(e5, e3, e6, p->vertices);
    }
    return 1;
}


/**** Cubical Polygonization (optional) ****/


#define LB      0  /* left bottom edge  */
```

```
#define LT      1  /* left top edge     */
#define LN      2  /* left near edge    */
#define LF      3  /* left far edge     */
#define RB      4  /* right bottom edge */
#define RT      5  /* right top edge    */
#define RN      6  /* right near edge   */
#define RF      7  /* right far edge    */
#define BN      8  /* bottom near edge  */
#define BF      9  /* bottom far edge   */
#define TN      10 /* top near edge     */
#define TF      11 /* top far edge      */

static INTLISTS *cubetable[256];

/*                          edge: LB, LT, LN, LF, RB, RT, RN, RF, BN, BF,
TN, TF */
static    int    corner1[12]                    =
{LBN,LTN,LBN,LBF,RBN,RTN,RBN,RBF,LBN,LBF,LTN,LTF};
static    int    corner2[12]                    =
{LBF,LTF,LTN,LTF,RBF,RTF,RTN,RTF,RBN,RBF,RTN,RTF};
static int leftface[12]     = {B,   L,   L,   F,   R,   T,   N,   R,   N,   B,
T,  F};
                                  /* face on left when going corner1 to
corner2 */
static int rightface[12]    = {L,   T,   N,   L,   B,   R,   R,   F,   B,   F,
N,  T};
                                 /* face on right when going corner1 to
corner2 */


/* docube: triangulate the cube directly, without decomposition */

int docube (cube, p)
CUBE *cube;
PROCESS *p;
{
    INTLISTS *polys;
    int i, index = 0;
     for (i = 0; i < 8; i++) if (cube->corners[i]->value > 0.0) index +=
(1<<i);
    for (polys = cubetable[index]; polys; polys = polys->next) {
        INTLIST *edges;
        int a = -1, b = -1, count = 0;
        for (edges = polys->list; edges; edges = edges->next) {
            CORNER *c1 = cube->corners[corner1[edges->i]];
            CORNER *c2 = cube->corners[corner2[edges->i]];
            int c = vertid(c1, c2, p);
                if (++count > 2 && ! p->triproc(a, b, c, p->vertices))
return 0;
            if (count < 3) a = b;
            b = c;
```

```
        }
    }
    return 1;
}


/* nextcwedge: return next clockwise edge from given edge around given
face */

int nextcwedge (edge, face)
int edge, face;
{
    switch (edge) {
        case LB: return (face == L)? LF : BN;
        case LT: return (face == L)? LN : TF;
        case LN: return (face == L)? LB : TN;
        case LF: return (face == L)? LT : BF;
        case RB: return (face == R)? RN : BF;
        case RT: return (face == R)? RF : TN;
        case RN: return (face == R)? RT : BN;
        case RF: return (face == R)? RB : TF;
        case BN: return (face == B)? RB : LN;
        case BF: return (face == B)? LB : RF;
        case TN: return (face == T)? LT : RN;
        case TF: return (face == T)? RT : LF;
    }
}


/* otherface: return face adjoining edge that is not the given face */

int otherface (edge, face)
int edge, face;
{
    int other = leftface[edge];
    return face == other? rightface[edge] : other;
}


/* makecubetable: create the 256 entry table for cubical polygonization
*/

makecubetable ()
{
    int i, e, c, done[12], pos[8];
    for (i = 0; i < 256; i++) {
        for (e = 0; e < 12; e++) done[e] = 0;
        for (c = 0; c < 8; c++) pos[c] = BIT(i, c);
        for (e = 0; e < 12; e++)
            if (!done[e] && (pos[corner1[e]] != pos[corner2[e]])) {
                INTLIST *ints = 0;
```

```
                        INTLISTS *lists = (INTLISTS *) mycalloc(1,
sizeof(INTLISTS));
                int start = e, edge = e;
                   /* get face that is to right of edge from pos to neg
corner: */
                int face = pos[corner1[e]]? rightface[e] : leftface[e];
                while (1) {
                    edge = nextcwedge(edge, face);
                    done[edge] = 1;
                    if (pos[corner1[edge]] != pos[corner2[edge]]) {
                        INTLIST *tmp = ints;
                                        ints = (INTLIST *) mycalloc(1,
sizeof(INTLIST));
                        ints->i = edge;
                           ints->next = tmp; /* add edge to head of list
*/
                        if (edge == start) break;
                        face = otherface(edge, face);
                    }
                }
                  lists->list = ints; /* add ints to head of table entry
*/
                lists->next = cubetable[i];
                cubetable[i] = lists;
            }
    }
}


/**** Storage ****/


/* mycalloc: return successful calloc or exit program */

char *mycalloc (nitems, nbytes)
int nitems, nbytes;
{
   char *ptr = calloc(nitems, nbytes);
   if (ptr != NULL) return ptr;
   fprintf(stderr, "can't calloc %d bytes\n", nitems*nbytes);
   exit(1);
}


/* setcenter: set (i,j,k) entry of table[]
 * return 1 if already set; otherwise, set and return 0 */

int setcenter(table, i, j, k)
CENTERLIST *table[];
int i, j, k;
{
```

```
    int index = HASH(i, j, k);
    CENTERLIST *new, *l, *q = table[index];
    for (l = q; l != NULL; l = l->next)
        if (l->i == i && l->j == j && l->k == k) return 1;
    new = (CENTERLIST *) mycalloc(1, sizeof(CENTERLIST));
    new->i = i; new->j = j; new->k = k; new->next = q;
    table[index] = new;
    return 0;
}


/* setedge: set vertex id for edge */

setedge (table, i1, j1, k1, i2, j2, k2, vid)
EDGELIST *table[];
int i1, j1, k1, i2, j2, k2, vid;
{
    unsigned int index;
    EDGELIST *new;
    if (i1>i2 || (i1==i2 && (j1>j2 || (j1==j2 && k1>k2)))) {
        int t=i1; i1=i2; i2=t; t=j1; j1=j2; j2=t; t=k1; k1=k2; k2=t;
    }
    index = HASH(i1, j1, k1) + HASH(i2, j2, k2);
    new = (EDGELIST *) mycalloc(1, sizeof(EDGELIST));
    new->i1 = i1; new->j1 = j1; new->k1 = k1;
    new->i2 = i2; new->j2 = j2; new->k2 = k2;
    new->vid = vid;
    new->next = table[index];
    table[index] = new;
}


/* getedge: return vertex id for edge; return -1 if not set */

int getedge (table, i1, j1, k1, i2, j2, k2)
EDGELIST *table[];
int i1, j1, k1, i2, j2, k2;
{
    EDGELIST *q;
    if (i1>i2 || (i1==i2 && (j1>j2 || (j1==j2 && k1>k2)))) {
        int t=i1; i1=i2; i2=t; t=j1; j1=j2; j2=t; t=k1; k1=k2; k2=t;
    };
    q = table[HASH(i1, j1, k1)+HASH(i2, j2, k2)];
    for (; q != NULL; q = q->next)
        if (q->i1 == i1 && q->j1 == j1 && q->k1 == k1 &&
            q->i2 == i2 && q->j2 == j2 && q->k2 == k2)
            return q->vid;
    return -1;
}
```

```
/**** Vertices ****/


/* vertid: return index for vertex on edge:
 * c1->value and c2->value are presumed of different sign
 * return saved index if any; else compute vertex and save */

int vertid (c1, c2, p)
CORNER *c1, *c2;
PROCESS *p;
{
    VERTEX v;
    POINT a, b;
      int vid = getedge(p->edges, c1->i, c1->j, c1->k, c2->i, c2->j,
c2->k);
     if (vid != -1) return vid;                          /* previously
computed */
    a.x = c1->x; a.y = c1->y; a.z = c1->z;
    b.x = c2->x; b.y = c2->y; b.z = c2->z;
     converge(&a, &b, c1->value, p->function, &v.position); /* position
*/
    vnormal(&v.position, p, &v.normal);                  /* normal */
     addtovertices(&p->vertices, v);                          /* save
vertex */
    vid = p->vertices.count-1;
    setedge(p->edges, c1->i, c1->j, c1->k, c2->i, c2->j, c2->k, vid);
    return vid;
}


/* addtovertices: add v to sequence of vertices */

addtovertices (vertices, v)
VERTICES *vertices;
VERTEX v;
{
    if (vertices->count == vertices->max) {
        int i;
        VERTEX *new;
        vertices->max = vertices->count == 0 ? 10 : 2*vertices->count;
                new = (VERTEX *) mycalloc((unsigned) vertices->max,
sizeof(VERTEX));
                 for (i = 0; i < vertices->count; i++) new[i] =
vertices->ptr[i];
        if (vertices->ptr != NULL) free((char *) vertices->ptr);
        vertices->ptr = new;
    }
    vertices->ptr[vertices->count++] = v;
}
```

```
/* vnormal: compute unit length surface normal at point */

vnormal (point, p, v)
POINT *point, *v;
PROCESS *p;
{
    double f = p->function(point->x, point->y, point->z);
    v->x = p->function(point->x+p->delta, point->y, point->z)-f;
    v->y = p->function(point->x, point->y+p->delta, point->z)-f;
    v->z = p->function(point->x, point->y, point->z+p->delta)-f;
    f = sqrt(v->x*v->x + v->y*v->y + v->z*v->z);
    if (f != 0.0) {v->x /= f; v->y /= f; v->z /= f;}
}


/* converge: from two points of differing sign, converge to zero
crossing */

converge (p1, p2, v, function, p)
double v;
double (*function)();
POINT *p1, *p2, *p;
{
    int i = 0;
    POINT pos, neg;
    if (v < 0) {
        pos.x = p2->x; pos.y = p2->y; pos.z = p2->z;
        neg.x = p1->x; neg.y = p1->y; neg.z = p1->z;
    }
    else {
        pos.x = p1->x; pos.y = p1->y; pos.z = p1->z;
        neg.x = p2->x; neg.y = p2->y; neg.z = p2->z;
    }
    while (1) {
        p->x = 0.5*(pos.x + neg.x);
        p->y = 0.5*(pos.y + neg.y);
        p->z = 0.5*(pos.z + neg.z);
        if (i++ == RES) return;
        if ((function(p->x, p->y, p->z)) > 0.0)
             {pos.x = p->x; pos.y = p->y; pos.z = p->z;}
        else {neg.x = p->x; neg.y = p->y; neg.z = p->z;}
    }
}
```