◊ I.6

Building Vertex Normals from an Unstructured Polygon List

Andrew Glassner
Xerox PARC
3333 Coyote Hill Road
Palo Alto, CA 94304
glassner@parc.xerox.com

◊ Abstract ◊

Many polygonal models are used as piecewise-flat approximations of curved models, and are thus "smooth-shaded" when displayed. To apply Gouraud or Phong shading to a model one needs to compute a surface normal at every vertex; often this simply involves averaging the surface normal of each polygon sharing that vertex.

This Gem provides a general-purpose procedure that computes vertex normals from any list of polygons. I assume that the polygons describe a simple manifold in 3D space, so that every local neighborhood is a flat sheet. I also assume that the structure is a *mesh*; that is, there are no "T" vertices, isolated vertices, or dangling edges. Except for the addition of normals at the vertices, the input model is unchanged.

I infer the topology of the model by building a data structure that allows quick access to all the polygons that have a vertex in the same region of space. To find the normal for a selected vertex, one needs only search the region surrounding the vertex and then average the normals for all polygons that share that vertex.

◊ Overview ◊

Polygons continue to be a popular primitive for approximating curved surfaces. To make polygonal models look smooth we can use Gouraud or Phong shading, often supported by hardware. Both of these techniques require a normal at each vertex of each polygon.

Typically a vertex normal is computed by combining the normals of all the polygons that share that vertex. A number of different strategies for this computation are presented in (Glassner 1990), but they all require that the polygons be identified first. When the polygons are generated in a mesh, it is easy to find all the polygons that share a vertex.

Some shape-generation programs are not so cooperative and instead generate polygons according to a less organized scheme. The result is a big list of polygons, each

IBM ISBN 0-12-336155-9
Macintosh ISBN 0-12-336156-7

identified by a list of explicit vertices. The trick then becomes finding which vertices are held in common by which polygons.

This Gem presents a piece of code that will run through all the polygons, identify shared vertices to infer the topology of the model, and average together the appropriate polygons to build vertex normals. Two vertices are labeled identical if they have very similar coordinates, to within a small tolerance.

I found it useful to provide a simple form of *edge preservation* for meshes, an idea originally introduced by Gouraud (Foley *et al.* 1990). If two adjacent polygons are sufficiently far from flat, then they are not averaged together at a common vertex. The use of edge preservation and its tolerance are selectable by the client (the calling routine).

The general idea is first to initialize the package, and then feed in a sequence of polygons in any order. When all the polygons are in, call a routine to compute all the normals. You then use the normals somehow (perhaps writing the resulting, augmented polygons to a file), and then free up the memory the package allocated.

◇ The Algorithm ◇

The algorithm used here is very simple. A few data structures guide the way.

First is the *Polygon*, which contains explicit storage for each vertex and its normal, and some bookkeeping information such as the polygon's own normal, its number of vertices, and so on.

The central organization comes from a hash table, which is made up of a linked list of *HashNode* structures. Each of these structures represents one vertex; it points to the polygon containing the vertex, identifies it by number, and provides its status, which is one of *waiting*, *working*, or *done* (explained ahead).

Polygons are entered one by one into the database. The client passes in a pointer to a list of vertices (an array of *Point3* structures), which is then copied, so the client can free or re-use that memory. When a polygon is entered, its normal is computed using Newell's method (Sutherland *et al.* 1974) (Tampieri 1992), and its vertices are inserted into the hash table. Each vertex is marked as *waiting*.

To compute normals, the system scans the list stored at each entry of the hash table and identifies all the polygons that have a particular vertex in common (to within the *fuzz* tolerance). The normals of all the participating polygons are averaged together and stored with the associated vertices.

The test for equality uses a parameter called *fuzz*, to accomodate floating-point errors when the model is made. If the Manhattan distance between two vertices doesn't exceed *fuzz*, then they are considered the same.

Note that if we identify (i.e., merge) two vertices that are nearby to be the same then we might be tempted to use just one piece of storage to hold the vertex and its

normal, rather than replicate that vertex at each polygon. The reason I don't do this is to support *edge preservation*. This is a technique where two adjacent polygons are *not* averaged together if they diverge from coplanarity by more than a given criterion. For example, if a number of polygons all come together at the tip of a narrow cone, then we probably don't want to smooth the vertex at the tip, though we want to smooth the other vertices of the cone. So the vertex at the tip will have a different normal for each constituent polygon. Edge preservation is disabled by default; you can enable it (and supply its comparison threshold) with a function call.

Note that as with any hashing scheme, two spatially distinct vertices can hash to the same table entry, so we may need to pass through each hash-table list multiple times to pick up each vertex.

The general flow of the algorithm is a pair of small loops in a larger loop. The outer loop scans all the hash buckets. If the entry is `NULL` then it moves to the next; if it is non-`NULL` then it is processed. Processing starts up a loop that continues as long as there are any vertices in the list at this entry that are marked as *waiting* (I simply scan the list and look for any such vertex). The first vertex found that is *waiting* is set to *working*; this becomes the vertex for which we want to compute a normal. We initialize a new vertex normal with the normal of the polygon containing this vertex.

We now continue scanning the list. If we find another vertex that is acceptably close to this one spatially, then we examine it. If edge preservation is disabled, we mark that vertex also as *working* and add its polygon's normal in to the accumulating vertex normal. If edge preservation is enabled, we check the angle between the new polygon and the original one and mark it *working* and add in the new polygon's normal only if the angle is sufficiently large.

When we reach the end of the list, we normalize the vertex normal. We then pass through the list again, searching for nodes with the status *working*. For each such node, we copy the vertex normal into the normal pointer for that vertex for that polygon, and set the node to *done*.

When we reach the end of the list, we return to scan for any more vertices that are still *waiting*. If every node in the list is *done*, we move on to the next hash-table entry.

◇ Use ◇

You supply input to the package by filling up data structures of type *Polygon*, which the package augments with vertex normals.

A simple demonstration program is supplied with the code; it makes a mesh of quadrilaterals and triangles and writes them to the standard output with vertex normals. The `main` routine initializes the package, inserts polygons, enables edge preservation, builds normals, saves the polygons, and frees the package's memory. The mesh is a function $z = y|1-2x|$ over the unit square; so at $y = 0$ it's flat, and at $y = 1$ it's a sharp crease. In

the driver I turn on edge preservation with a minimum dot value of 0.0 (corresponding to 90°), so the crease is smooth up to $y = 0.5$, when it becomes crisp.

To use the package, call `initAllTables`, which returns a pointer to a data structure of type *Smooth*.

To insert polygons, fill in an array of *Point3* structures with the vertex locations, and pass this array, the vertex count, and the *smooth* pointer to `includePolygon`. This routine also allows you to attach a *user* pointer of arbitrary type to the polygon. I have found this useful for keeping color and texture information with the polygon.

To enable edge preservation, you can call `enableEdgePreservation` at any time after initialization and before computing normals. Pass it the *smooth* pointer and the value of the smallest dot product (i.e., the cosine of the largest angle) which you are willing to call "flat." If you later want to turn off this option, call `disableEdgePreservation`.

The fuzz tolerance for comparisons is set by a call to `setFuzzFraction`.

When all your polygons are in, call `makeVertexNormals`, passing in the *smooth* pointer. The result is that *Smooth* field *polygonTable* now points to a copy of your polygon list, only each polygon now contains its own surface normal (stored in *normal*) and an array of normals (stored in *normals* as an array of *Vector3* structures) corresponding to each vertex. The *next* field points to the next polygon so you can read them all back by following this link. The polygons are stored in this list in the same order in which you included them. When you're done with all the polygons, call `freeAll` with the pointer to *smooth* to release the storage used by the system.

You may want to play with the hashing function; I used a very simple one. First I round each vertex to three digits of precision (this is controlled by the `QSIZE` constant). I then scale the three coordinates by three different small primes. You can try any hash function you like, but it must always return a non-negative value. The size of the hash table is given by the `HASH_TABLE_SIZE` constant.

◇ Discussion ◇

This algorithm can be sensitive to small variations in the input. For example, if two adjacent polygons share a vertex, but one stores the X coordinate of that vertex as 3.999999 and the other stores it as 4.0, then the two vertices might fall into different buckets. This could be fixed by multiple hashing: use two different, overlapping hashing functions, and run through both hash tables for each vertex (being careful not to duplicate included polygons).

The algorithm could be improved by making `QSIZE` also dependent on the overall bounding box of the model.

Some applications may find it useful to access the internal data structure which contains the inferred topology of the model before returning that memory to the operating system.

◇ C Code ◇

Header File

```
/* smooth.h */
/* header file for polygon smoothing */
/* Andrew S. Glassner / Xerox PARC */

#include <stdio.h>
#include <math.h>

#ifdef STANDALONE_TEST

#define NEWTYPE(x) (x *)malloc((unsigned)(sizeof(x)))

typedef struct Point3Struct {
   double x, y, z;
   } Point3;
typedef Point3 Vector3;
typedef int boolean;
#define TRUE  1
#define FALSE 0

Vector3 *V3Normalize(Vector3 *v);
Vector3 *V3Add(Vector3 *a, Vector3 *b, Vector3 *c);
double V3Dot(Vector3 *a, Vector3 *b);
#else
#include "GraphicsGems.h"
#endif

/********* MACROS and CONSTANTS *********/

/* new array creator */
#define NEWA(x, num) (x *)malloc((unsigned)((num) * sizeof(x)))

#define MARKWAITING    0
#define MARKWORKING    1
#define MARKDONE       2

/* fuzzy comparison macro */
#define FUZZEQ(x,y)  (fabs((x)-(y))<(smooth->fuzz))

/* hash table size; related to HASH */
#define HASH_TABLE_SIZE        1000

/* quantization increment */
#define QSIZE    1000.0

#define QUANT(x)      (((int)((x)*QSIZE))/QSIZE)
#define ABSQUANT(x)   (((int)((fabs(x))*QSIZE))/QSIZE)
#define HASH(pt)      ( \
                      (int)(((3*ABSQUANT(pt->x)) + \
```

```
                        (5*ABSQUANT(pt->y)) + \
                        (7*ABSQUANT(pt->z))) * \
                HASH_TABLE_SIZE)) % HASH_TABLE_SIZE

/********* STRUCTS AND TYPES *********/

typedef struct Polygonstruct {
    Point3      *vertices;  /* polygon vertices */
    Vector3     *normals;   /* normal at each vertex */
    Vector3     normal;     /* normal for polygon */
    int         numVerts;   /* number of vertices */
    void        *user;      /* user information */
    struct      Polygonstruct *next;
    } Polygon_def;
typedef Polygon_def *Polygon;

typedef struct HashNodestruct {
    Polygon   polygon;      /* polygon for this vertex */
    int       vertexNum;    /* which vertex this is */
    int       marked;       /* vertex status */
    struct    HashNodestruct *next;
    } HashNode_def;
typedef HashNode_def *HashNode;

typedef struct SmoothStruct {
    HashNode  hashTable[HASH_TABLE_SIZE];
    Polygon   polygonTable;
    Polygon   polyTail;
    double    fuzz;           /* distance for vertex equality */
    double    fuzzFraction;   /* fraction of model size for fuzz */
    boolean   edgeTest;       /* apply edging test using minDot */
    float     minDot;         /* if > this, make sharp edge; see above */
    } Smooth_def;
typedef Smooth_def *Smooth;

/********* public procs ************/
Smooth   initAllTables();
void     includePolygon(int numVerts, Point3 *verts, Smooth smooth, void *user);
void     makeVertexNormals(Smooth smooth);

/********* public option contorl procs ************/
void     setFuzzFraction(Smooth smooth, float fuzzFraction);
void     enableEdgePreservation(Smooth smooth, float minDot);
void     disableEdgePreservation(Smooth smooth);
```

Smoothing Code

```
/* smooth.c - Compute vertex normals for polygons.
   Andrew S. Glassner / Xerox PARC
```

```
  The general idea is to 1) initialize the tables, 2) add polygons one by one,
  3) optionally enable edge preservation, 4) optionally set the fuzz factor,
  5) compute the normals, 6) do something with the new normals, then 7) free
  the new storage.  The calls to do this are:

  1) smooth = initAllTables();
  2) includePolygon(int numVerts, Point3 *verts, Smooth smooth);
  3) (optional) enableEdgePreservation(Smooth smooth, float minDot);
  4) (optional) setFuzzFraction(smooth Smooth, float fuzzFraction);
  5) makeVertexNormals(smooth);
  6) YOUR CODE
  7) freeSmooth(smooth);

  Edge preservation is used to retain sharp creases in the model.  If it is
  enabled, then the dot product of each pair of polygons sharing a vertex
  is computed.  If this value is below the value of 'minDot' (that is,
  the two polygons are a certain distance away from flatness), then the
  polygons are not averaged together for the vertex normal.

  If you want to re-compute the results without edge preservation, call
    disableEdgePreservation(smooth);

  The general flow of the algorithm is:
    1. currentHash = scan hashTable
    2. while (any unmarked) {
         3. firstVertex = first unmarked vertex.  set to MARKWORKING
         4. normal = firstVertex->polygon->normal
         5. scan currentHash.  If vertex = firstVertex
              6. normal += vertex->polygon->normal
              7. set vertex to MARKWORKING
              (end of scan)
         8. set normal to unit length
         9. scan currentHash.  If vertex set to MARKWORKING
              10. set vertex->normal = normal
              11. set to MARKDONE
              (end of scan)
         (end while)

  The HASH macro must always return a non-negative value, even for negative inputs.
  The size of the hash table can be adjusted to taste.
  The fuzz for comparison needs to be matched to the resolution of the model.
*/

#include "smooth.h"

void     addVertexToTable(Point3 *pt, Polygon polygon, int vNum, Smooth smooth);
void     makePolyNormal(Polygon polygon);
void     writeSmooth(FILE *fp, int numPolys);
HashNode getFirstWaitingNode(HashNode node);
void     processHashNode(HashNode headNode, HashNode firstNode, Smooth smooth);
int      hashPolys(boolean phase);
void     writeGeom(int numPolys);
void     freeSmooth(Smooth smooth);
```

```
boolean  compareVerts(Point3 *v0, Point3 *v1, Smooth smooth);
void     computeFuzz(Smooth smooth);

/********* ENTRY PROCS *********/

/* add this polygon to the tables */
void includePolygon(int numVerts, Point3 *verts, Smooth smooth, void *user) {
int i;
Point3 *vp, *ovp;
    Polygon polygon = NEWTYPE(Polygon_def);
    polygon->next = NULL;
    if (smooth->polyTail != NULL) {
      smooth->polyTail->next = polygon;
    } else {
        smooth->polygonTable = polygon;
    };
    smooth->polyTail = polygon;
    polygon->vertices = NEWA(struct Point3Struct, numVerts);
    polygon->normals = NEWA(struct Point3Struct, numVerts);
    polygon->user = user;
    vp = polygon->vertices;
    ovp = verts;
    polygon->numVerts = numVerts;

    for (i=0; i<numVerts; i++) {
        vp->x = ovp->x;
        vp->y = ovp->y;
        vp->z = ovp->z;
        addVertexToTable(vp, polygon, i, smooth);
        vp++;
        ovp++;
        };
    makePolyNormal(polygon);
    }

void enableEdgePreservation(Smooth smooth, float minDot) {
    smooth->edgeTest = TRUE;
    smooth->minDot = minDot;
    }

void disableEdgePreservation(Smooth smooth) {
    smooth->edgeTest = FALSE;
    }

void setFuzzFraction(Smooth smooth, float fuzzFraction) {
    smooth->fuzzFraction = fuzzFraction;
    }

/******** PROCEDURES ********/

/* set all the hash-table linked lists to NULL */
Smooth initAllTables() {
int i;
```

```
Smooth smooth = NEWTYPE(Smooth_def);
    for (i=0; i<HASH_TABLE_SIZE; i++) smooth->hashTable[i] = NULL;
    smooth->polygonTable = NULL;
    smooth->polyTail = NULL;
    smooth->edgeTest = FALSE;
    smooth->minDot = 0.2;
    smooth->fuzzFraction = 0.001;
    smooth->fuzz = 0.001;
    return(smooth);
    }

/* hash this vertex and add it into the linked list */
void addVertexToTable(Point3 *pt, Polygon polygon, int vNum, Smooth smooth) {
int hash = HASH(pt);
HashNode newNode = NEWTYPE(HashNode_def);
    newNode->next = smooth->hashTable[hash];
    smooth->hashTable[hash] = newNode;
    newNode->polygon = polygon;
    newNode->vertexNum = vNum;
    newNode->marked = MARKWAITING;
    }

/* compute the normal for this polygon using Newell's method */
/* (see Tampieri, Gems III, p. 517) */
void makePolyNormal(Polygon polygon) {
Point3 *vp, *p0, *p1;
int i;
   polygon->normal.x = 0.0; polygon->normal.y = 0.0; polygon->normal.z = 0.0;
   vp = polygon->vertices;
   for (i=0; i<polygon->numVerts; i++) {
      p0 = vp++;
      p1 = vp;
      if (i == polygon->numVerts-1) p1 = polygon->vertices;
      polygon->normal.x += (p1->y - p0->y) * (p1->z + p0->z);
      polygon->normal.y += (p1->z - p0->z) * (p1->x + p0->x);
      polygon->normal.z += (p1->x - p0->x) * (p1->y + p0->y);
      };
   (void) V3Normalize(&(polygon->normal));
   }

/* scan each list at each hash table entry until all nodes are marked */
void makeVertexNormals(Smooth smooth) {
HashNode currentHashNode;
HashNode firstNode;
int i;
    computeFuzz(smooth);
    for (i=0; i<HASH_TABLE_SIZE; i++) {
        currentHashNode = smooth->hashTable[i];
        do {
            firstNode = getFirstWaitingNode(currentHashNode);
            if (firstNode != NULL) {
                processHashNode(currentHashNode, firstNode, smooth);
                };
```

```
            } while (firstNode != NULL);
        };
    }

void computeFuzz(Smooth smooth) {
Point3 min, max;
double od, d;
Point3 *v;
int i;
Polygon poly = smooth->polygonTable;
  min.x = max.x = poly->vertices->x;
  min.y = max.y = poly->vertices->y;
  min.z = max.z = poly->vertices->z;
  while (poly != NULL) {
    v = poly->vertices;
    for (i=0; i<poly->numVerts; i++) {
      if (v->x < min.x) min.x = v->x;
      if (v->y < min.y) min.y = v->y;
      if (v->z < min.z) min.z = v->z;
      if (v->x > max.x) max.x = v->x;
      if (v->y > max.y) max.y = v->y;
      if (v->z > max.z) max.z = v->z;
      v++;
      };
    poly = poly->next;
    };
  d = fabs(max.x - min.x);
  od = fabs(max.y - min.y);  if (od > d) d = od;
  od = fabs(max.z - min.z);  if (od > d) d = od;
  smooth->fuzz = od * smooth->fuzzFraction;
  }

/* get first node in this list that isn't marked as done */
HashNode getFirstWaitingNode(HashNode node) {
    while (node != NULL) {
        if (node->marked != MARKDONE) return(node);
        node = node->next;
        };
    return(NULL);
    }

/* are these two vertices the same to with the tolerance? */
boolean compareVerts(Point3 *v0, Point3 *v1, Smooth smooth) {
    float q0, q1;
    q0 = QUANT(v0->x); q1 = QUANT(v1->x); if (!FUZZEQ(q0, q1)) return(FALSE);
    q0 = QUANT(v0->y); q1 = QUANT(v1->y); if (!FUZZEQ(q0, q1)) return(FALSE);
    q0 = QUANT(v0->z); q1 = QUANT(v1->z); if (!FUZZEQ(q0, q1)) return(FALSE);
    return(TRUE);
    }

/* compute the normal for an unmarked vertex */
void processHashNode(HashNode headNode, HashNode firstNode, Smooth smooth) {
    HashNode scanNode = firstNode->next;
```

```
    Point3 *firstVert = &(firstNode->polygon->vertices[firstNode->vertexNum]);
    Point3 *headNorm = &(firstNode->polygon->normal);
    Point3 *testVert, *testNorm;
    Point3 normal;
    float ndot;

    firstNode->marked = MARKWORKING;
    normal.x = firstNode->polygon->normal.x;
    normal.y = firstNode->polygon->normal.y;
    normal.z = firstNode->polygon->normal.z;

    while (scanNode != NULL) {
        testVert = &(scanNode->polygon->vertices[scanNode->vertexNum]);
        if (compareVerts(testVert, firstVert, smooth)) {
            testNorm = &(scanNode->polygon->normal);
            ndot = V3Dot(testNorm, headNorm);

                if ((!(smooth->edgeTest)) || (ndot > smooth->minDot)) {
                V3Add(&normal, testNorm, &normal);
                scanNode->marked = MARKWORKING;
                };
            };
        scanNode = scanNode->next;
        };

    V3Normalize(&normal);

    scanNode = firstNode;
    while (scanNode != NULL) {
        if (scanNode->marked == MARKWORKING) {
            testNorm = &(scanNode->polygon->normals[scanNode->vertexNum]);
            testNorm->x = normal.x;
            testNorm->y = normal.y;
            testNorm->z = normal.z;
            scanNode->marked = MARKDONE;
            testVert = &(scanNode->polygon->vertices[scanNode->vertexNum]);
            };
        scanNode = scanNode->next;
        };
    }

/* free up all the memory */
void freeSmooth(Smooth smooth) {
HashNode headNode;
HashNode nextNode;
Polygon poly;
Polygon nextPoly;
int i;
    for (i=0; i<HASH_TABLE_SIZE; i++) {
        headNode = smooth->hashTable[i];
        while (headNode != NULL) {
            nextNode = headNode->next;
            free(headNode);
```

```
            headNode = nextNode;
            };
        };
    poly = smooth->polygonTable;
    while (poly != NULL) {
        nextPoly = poly->next;
        freePoly(poly);
        poly = nextPoly;
        };
    smooth->polygonTable = NULL;
    free(smooth);
    }

freePoly(polygon) Polygon polygon; {
    if (polygon->vertices != NULL) free(polygon->vertices);
    if (polygon->normals != NULL) free(polygon->normals);
    polygon->next = NULL;
    free(polygon);
    }
```

Sample Driver

```
/* test.c - sample driver for polygon smoother */
/* makes a mesh height field of quadrilaterals and triangles */
/* Andrew S. Glassner / Xerox PARC */

#include "smooth.h"

#ifdef STANDALONE_TEST
/* from Graphics Gems library ; for standalone compile */

/* normalizes the input vector and returns it */
Vector3 *V3Normalize(Vector3 *v) {
  double len = sqrt(V3Dot(v, v));
  if (len != 0.0) { v->x /= len;  v->y /= len; v->z /= len; }
  return(v);
  }

/* return vector sum c = a+b */
Vector3 *V3Add(Vector3 *a, Vector3 *b, Vector3 *c) {
  c->x = a->x+b->x;  c->y = a->y+b->y;  c->z = a->z+b->z;
  return(c);
  }

/* return the dot product of vectors a and b */
double V3Dot(Vector3 *a, Vector3 *b) {
  return((a->x*b->x)+(a->y*b->y)+(a->z*b->z));
  }
#endif
```

```
/* make a square height field of quadrilaterals and triangles */
main(int ac, char *av[]) {
int xres, yres;
Smooth smooth;
  if (ac < 3) { printf("use: test x y\n"); exit(-1); };  /* abrupt, I know */
  xres = atoi(*++av);
  yres = atoi(*++av);
  smooth = initAllTables();              /* initialize */
  buildMesh(smooth, xres, yres);         /* build the mesh (calls includePolygon) */
  enableEdgePreservation(smooth, 0.0);   /* 90 degree folds or more stay crisp */
  makeVertexNormals(smooth);             /* build the normals */
  savePolys(smooth);                     /* save the result in a file */
  freeSmooth(smooth);                    /* take only normals, leave only footprints */
  }

/* z=f(x,y) */
double fofxy(double x, double y) {
   double h;
   h = 2.0 * (0.5 - x); if (h < 0) h = -h; h = h * y;
   return(h);
   }

buildMesh(Smooth smooth, int xres, int yres) {
int x, y;
Point3 *vlist;
double dx, dy, lx, ly, hx, hy;
  vlist = NEWA(struct Point3Struct, 4);
  dx = 1.0/((double)(xres));
  dy = 1.0/((double)(yres));
  for (y=0; y<yres; y++) {
    ly = y * dy;
    hy = (y+1) * dy;
    for (x=0; x<xres; x++) {
       lx = x * dx;
       hx = (x+1) * dx;
       if ((x+y)%2 == 0) addTriangles(lx, ly, hx, hy, vlist, smooth);
                    else addQuadrilateral(lx, ly, hx, hy, vlist, smooth);
      };
    };
  free(vlist);
  }

addTriangles(double lx, double ly, double hx, double hy,
             Point3 *vlist, Smooth smooth) {
Point3 *p = vlist;
   /* make the first triangle */
   p->x = lx;  p->y = ly;  p->z = fofxy(p->x, p->y); p++;
   p->x = hx;  p->y = ly;  p->z = fofxy(p->x, p->y); p++;
   p->x = lx;  p->y = hy;  p->z = fofxy(p->x, p->y); p++;
   includePolygon(3, vlist, smooth, NULL);  /* add the polygon */
   /* make the other triangle */
   p = vlist;
   p->x = hx;  p->y = ly;  p->z = fofxy(p->x, p->y); p++;
```

```
   p->x = hx;  p->y = hy;  p->z = fofxy(p->x, p->y); p++;
   p->x = lx;  p->y = hy;  p->z = fofxy(p->x, p->y); p++;
   includePolygon(3, vlist, smooth, NULL);  /* add the polygon */
   }

addQuadrilateral(double lx, double ly, double hx, double hy,
                 Point3 *vlist, Smooth smooth) {
Point3 *p = vlist;
   p->x = lx;  p->y = ly;  p->z = fofxy(p->x, p->y); p++;
   p->x = hx;  p->y = ly;  p->z = fofxy(p->x, p->y); p++;
   p->x = hx;  p->y = hy;  p->z = fofxy(p->x, p->y); p++;
   p->x = lx;  p->y = hy;  p->z = fofxy(p->x, p->y); p++;
   includePolygon(4, vlist, smooth, NULL);  /* add the polygon */
   }

savePolys(Smooth smooth) {
Polygon poly = smooth->polygonTable;
int i, k;
Point3 *v, *n;
   printf("NQUAD\n");   /* header for point/normal format */
   while (poly != NULL) {
      for (i=0; i<4; i++) {
         k = i;   /* we always write 4 points so double 3rd triangle vertex */
         if (i >= poly->numVerts) k = poly->numVerts-1;
         v = &(poly->vertices[k]);
         n = &(poly->normals[k]);
         printf("%f %f %f %f %f %f\n", v->x, v->y, v->z, n->x, n->y, n->z);
         };
      printf("\n");
      poly = poly->next;
      };
   }
```

◇ Bibliography ◇

(Foley *et al.* 1990) James Foley, Andries van Dam, Steven Feiner, and John Hughes. *Computer Graphics Principles and Practice*, second edition. Addison-Wesley, Reading, 1990.

(Glassner 1990) Andrew S. Glassner. Computing surface normals for 3d models. In Andrew S. Glassner, editor, *Graphics Gems*, chapter 10.12. Academic Press, Boston, 1990.

(Sutherland *et al.* 1974) I. E. Sutherland, R. F. Sproull, and R. A. Schumacker. A characterization of ten hidden-surface algorithms. *AMC Computing Surveys*, 6(1):1–55, 1974.

(Tampieri 1992) Filippo Tampieri. Newell's method for computing the plane equation of a polygon. In David Kirk, editor, *Graphics Gems III*, chapter V.5. Academic Press, Boston, 1992.