

```

/*
 * normal_surface_recognition.c
 *
 * The function
 *
 * void recognize_embedded_surface(
 *     Triangulation *manifold,
 *     Boolean *connected,
 *     Boolean *orientable,
 *     Boolean *two_sided,
 *     int *Euler_characteristic);
 *
 * reports the connectedness, orientability, two-sidedness and Euler
 * characteristic of the normal surface described in the parallel_edge,
 * num_squares and num_triangles fields of the manifold's Tetrahedra.
 * The present implementation assumes the manifold has no filled cusps.
 */

#include "kernel.h"
#include "normal_surfaces.h"

typedef struct
{
    /*
     * The "positive" normal vector to a square points in the direction of
     * tet->parallel_edge (which has EdgeIndex 0, 1 or 2) and away from
     * the opposite edge (which has EdgeIndex 5, 4 or 3, respectively).
     * The "positive" normal vector to a triangle points in the direction
     * of the associated ideal vertex.
     *
     * The algorithm for testing two-sidedness attempts to make a globally
     * consistent choice of normal vectors across the whole surface.
     */
    Boolean positive_normal;

    /*
     * A Tetrahedron's right_handed Orientation lets us extend the above
     * definition of a positive normal vector to a definition of a positive
     * orientation on each square and triangle. It doesn't matter whether
     * you imagine using a right-hand rule or a left-hand rule, just so
     * you're consistent.
     *
     * The algorithm for testing orientability attempts to make a globally
     * consistent choice of orientation across the whole surface.
     */
    Boolean positive_orientation;

    /*
     * Has the recursive algorithm visited this EmbeddedPolygon?
     */
    Boolean visited;
} EmbeddedPolygon;

/*
 * Each Tetrahedron will need one array of squares, and four arrays
 * of triangles, one for each cusp.
 */
typedef struct
{
    EmbeddedPolygon *squares,
                  *triangles[4];
} PolygonsInTetrahedron;

/*
 * The algorithm for determining connectedness, orientability and
 * two-sidedness keeps references to squares and triangles on a
 * NULL-terminated, singly linked list.
 */

typedef int EmbeddedPolygonType;
enum
{
    embedded_square,

```

```

    embedded_triangle
};

typedef struct ListNode
{
    /*
     * Which Tetrahedron is the polygon in?
     */
    Tetrahedron      *tet;

    /*
     * Is the polygon an embedded_square or an embedded_triangle?
     */
    EmbeddedPolygonType type;

    /*
     * If the polygon is an embedded_triangle, which ideal vertex is it at?
     */
    VertexIndex      v;

    /*
     * Parallel copies of a square or triangle are indexed in the
     * direction opposite the normal vector defined above. For example,
     * the triangle closest to the ideal vertex has index 0, the next
     * closest one has index 1, etc. Similarly, the square closest
     * tet->parallel_edge has index 0, the next closest one has index 1, etc.
     */
    int              index;

    /*
     * The next ListNode on the NULL-terminated, singly linked list.
     */
    struct ListNode   *next;
} ListNode;

static void connected_orientable_twosided(Triangulation *manifold, Boolean *connected,
    Boolean *orientable, Boolean *two_sided);
static int Euler_characteristic_of_embedded_surface(Triangulation *manifold);

void recognize_embedded_surface(
    Triangulation *manifold,
    Boolean *connected,
    Boolean *orientable,
    Boolean *two_sided,
    int *Euler_characteristic)
{
    /*
     * The present version of the software assumes all cusps are complete.
     */
    if (all_cusps_are_complete(manifold) == FALSE)
        uFatalError("recognize_embedded_surface", "normal_surface_recognition");

    /*
     * Compute the connectedness, orientability, two-sidedness and
     * Euler characteristic.
     */
    connected_orientable_twosided(manifold, connected, orientable, two_sided);
    *Euler_characteristic = Euler_characteristic_of_embedded_surface(manifold);

    /*
     * In an orientable 3-manifold, a surface is orientable iff it's 2-sided.
     */
    if (manifold->orientability == oriented_manifold
        && *orientable != *two_sided)
        uFatalError("recognize_embedded_surface", "normal_surface_recognition");

    /*
     * An embedded sphere must be 2-sided.
     */
    if (*connected == TRUE
        && *Euler_characteristic == 2

```

```

    && *two_sided == FALSE)
        uFatalError("recognize_embedded_surface", "normal_surface_recognition");

/*
 * Orientable surfaces have even Euler characteristic.
 */
if (*orientable == TRUE
    && (*Euler_characteristic)%2 != 0)
    uFatalError("recognize_embedded_surface", "normal_surface_recognition");
}

static void connected_orientable_twosided(
    Triangulation *manifold,
    Boolean *connected,
    Boolean *orientable,
    Boolean *two_sided)
{
    PolygonsInTetrahedron *model;
    Tetrahedron *tet,
    *nbr;

    Permutation gluing;
    VertexIndex v,
    nbr_v;

    FaceIndex f,
    nbr_f;

    int i,
    index,
    nbr_index;

    ListNode *list,
    *node,
    *new_node;

    Boolean positive_normal,
    positive_orientation;

    EmbeddedPolygonType nbr_type;
    EmbeddedPolygon *data,
    *nbr_data;

/*
 * Make an explicit model of the surface using EmbeddedPolygon structures.
 */

    model = NEW_ARRAY(manifold->num_tetrahedra, PolygonsInTetrahedron);

    for (tet = manifold->tet_list_begin.next;
        tet != &manifold->tet_list_end;
        tet = tet->next)
    {
        /*
         * NEW_ARRAY uses my_malloc(), which gracefully handles requests
         * for zero bytes when num_squares or num_triangles is zero.
         */

        model[tet->index].squares = NEW_ARRAY(tet->num_squares, EmbeddedPolygon);
        for (i = 0; i < tet->num_squares; i++)
            model[tet->index].squares[i].visited = FALSE;

        for (v = 0; v < 4; v++)
        {
            model[tet->index].triangles[v] = NEW_ARRAY(tet->num_triangles[v],
EmbeddedPolygon);
            for (i = 0; i < tet->num_triangles[v]; i++)
                model[tet->index].triangles[v][i].visited = FALSE;
        }
    }

/*
 * Initialize the linked list to be empty.
 */
    list = NULL;

/*
 * Find an arbitrary embedded square. The calling routine won't
 * create empty or boundary-parallel surfaces, so a square must exist.

```

```

    * "Visit" the square and set its normal vector and orientation
    * to be positive. Put a reference to the square onto the linked list.
    * The linked list will hold squares and triangles which have been
    * visited, but whose neighbors have not yet been visited.
    */
for (tet = manifold->tet_list_begin.next;
     tet != &manifold->tet_list_end;
     tet = tet->next)
    if (tet->num_squares != 0)
    {
        model[tet->index].squares[0].positive_normal      = TRUE;
        model[tet->index].squares[0].positive_orientation = TRUE;
        model[tet->index].squares[0].visited              = TRUE;

        node = NEW_STRUCT(ListNode);
        node->tet      = tet;
        node->type     = embedded_square;
        node->v        = -1; /* unused for a square */
        node->index    = 0;
        node->next     = list;
        list          = node;
        break;
    }
if (list == NULL)
    uFatalError("connected_orientable_twosided", "normal_surface_recognition");

/*
 * Tentatively assume the surface is orientable and two-sided.
 * If the recursion below discovers that it cannot consistently
 * assign an orientation or normal vector, it will set the
 * corresponding variable to FALSE.
 */
*orientable = TRUE;
*two_sided  = TRUE;

/*
 * As stated above, the linked list holds squares and triangles which
 * have been visited, but whose neighbors have not yet been visited.
 */
while (list != NULL)
{
    /*
     * Pull the first node off the list.
     */
    node = list;
    list = list->next;

    /*
     * The node defines a square or triangle in the embedded surface.
     * Look at each of its neighbors.
     */
    tet = node->tet;
    for (f = 0; f < 4; f++)
    {
        /*
         * A square connects to all four of the tetrahedron's neighbors.
         * A triangle connects to only three of them.
         */
        if (node->type == embedded_triangle && node->v == f)
            continue;

        /*
         * What vertex (of the tetrahedron) are we at,
         * and what's the index of the sheet we're on?
         */
        switch (node->type)
        {
            case embedded_square:
                if (f == one_vertex_at_edge[tet->parallel_edge])
                {
                    v          = other_vertex_at_edge[tet->parallel_edge];
                    index      = node->index + tet->num_triangles[v];
                }
                if (f == other_vertex_at_edge[tet->parallel_edge])

```

```

        {
            v      = one_vertex_at_edge[tet->parallel_edge];
            index   = node->index + tet->num_triangles[v];
        }
        if (f == one_vertex_at_edge[5 - tet->parallel_edge])
        {
            v      = other_vertex_at_edge[5 - tet->parallel_edge];
            index   = ((tet->num_squares - 1) - node->index) + tet->
num_triangles[v];
        }
        if (f == other_vertex_at_edge[5 - tet->parallel_edge])
        {
            v      = one_vertex_at_edge[5 - tet->parallel_edge];
            index   = ((tet->num_squares - 1) - node->index) + tet->
num_triangles[v];
        }
        break;

    case embedded_triangle:
        v      = node->v;
        index   = node->index;
        break;

    default: uFatalError("connected_orientable_twosided",
"normal_surface_recognition");
}

/*
 * What normal vector and orientation are we passing
 * to the neighbor? Usually it will just be our own
 * normal vector and orientation, but in the case of
 * an "upside down" square we have to reverse them.
 */

if (node->type == embedded_square)
    data = &model[node->tet->index].squares[node->index];
else
    data = &model[node->tet->index].triangles[node->v][node->index];

positive_normal      = data->positive_normal;
positive_orientation = data->positive_orientation;
if (data->visited != TRUE)
    uFatalError("connected_orientable_twosided", "normal_surface_recognition");

if (node->type == embedded_square
    && edge_between_vertices[v][f] != tet->parallel_edge)
{
    positive_normal      = ! positive_normal;
    positive_orientation = ! positive_orientation;
}

/*
 * Find our neighbor.
 */
nbr      = tet->neighbor[f];
gluing   = tet->gluing[f];
nbr_f    = EVALUATE(gluing, f);
nbr_v    = EVALUATE(gluing, v);

/*
 * If the gluing is orientation_reversing, then
 * what the old tetrahedron saw as right-handed,
 * the neighbor will see as left-handed, and vice-versa.
 * They'll agree on normal vectors, though.
 */
if (parity[gluing] == orientation_reversing)
    positive_orientation = ! positive_orientation;

/*
 * Find the square or triangle we're connecting to.
 */
if (index < nbr->num_triangles[nbr_v])
{
    nbr_type      = embedded_triangle;

```

```

        nbr_index    = index;
        nbr_data     = &model[nbr->index].triangles[nbr_v][nbr_index];
    }
    else
    {
        if (edge3_between_vertices[nbr_f][nbr_v] != nbr->parallel_edge
            || index >= nbr->num_triangles[nbr_v] + nbr->num_squares)
            uFatalError("connected_orientable_twosided",
"normal_surface_recognition");

        nbr_type     = embedded_square;
        nbr_index     = index - nbr->num_triangles[nbr_v];

        /*
         * If the square is "upside down", adjust the index,
         * orientation and normal vector.
         */
        if (edge_between_vertices[nbr_f][nbr_v] != nbr->parallel_edge)
        {
            nbr_index = (nbr->num_squares - 1) - nbr_index;
            positive_normal = ! positive_normal;
            positive_orientation = ! positive_orientation;
        }

        nbr_data = &model[nbr->index].squares[nbr_index];
    }

    /*
     * Has the newly found square or triangle already been visited?
     * If it has, check whether its orientation and normal vector
     * agree with the ones we're passing.  If not, assign the
     * orientation and normal vectors, and add it to the linked list.
     */
    if (nbr_data->visited == TRUE)
    {
        if (nbr_data->positive_normal != positive_normal)
            *two_sided = FALSE;
        if (nbr_data->positive_orientation != positive_orientation)
            *orientable = FALSE;
    }
    else
    {
        nbr_data->positive_normal = positive_normal;
        nbr_data->positive_orientation = positive_orientation;
        nbr_data->visited = TRUE;

        new_node = NEW_STRUCT(ListNode);
        new_node->tet = nbr;
        new_node->type = nbr_type;
        new_node->v = (nbr_type == embedded_triangle) ? nbr_v : -1;
        new_node->index = nbr_index;
        new_node->next = list;
        list = new_node;
    }
}

/*
 * Free the node, and continue with the loop.
 */
my_free(node);
}

/*
 * The embedded surface is connected iff we visited all its polygons.
 */
*connected = TRUE;
for (tet = manifold->tet_list_begin.next;
     tet != &manifold->tet_list_end;
     tet = tet->next)
{
    for (i = 0; i < tet->num_squares; i++)
        if (model[tet->index].squares[i].visited == FALSE)
            *connected = FALSE;
    for (v = 0; v < 4; v++)

```

```

        for (i = 0; i < tet->num_triangles[v]; i++)
            if (model[tet->index].triangles[v][i].visited == FALSE)
                *connected = FALSE;
    }

    /*
     * Free the memory used to hold the EmbeddedPolygons.
     */
    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        my_free(model[tet->index].squares);
        for (v = 0; v < 4; v++)
            my_free(model[tet->index].triangles[v]);
    }
    my_free(model);
}

static int Euler_characteristic_of_embedded_surface(
    Triangulation *manifold)
{
    int          num_vertices,
                num_edges,
                num_faces;
    EdgeClass    *edge;
    Tetrahedron *tet;
    EdgeIndex    e;
    VertexIndex  v;
    int          total_squares,
                total_triangles;

    /*
     * Count the vertices.
     */

    num_vertices = 0;

    for (edge = manifold->edge_list_begin.next;
         edge != &manifold->edge_list_end;
         edge = edge->next)
    {
        tet = edge->incident_tet;
        e   = edge->incident_edge_index;

        if (edge3[e] != tet->parallel_edge)
            num_vertices += tet->num_squares;

        num_vertices += tet->num_triangles[one_vertex_at_edge[e]];
        num_vertices += tet->num_triangles[other_vertex_at_edge[e]];
    }

    /*
     * Count the edges and faces.
     */

    total_squares = 0;
    total_triangles = 0;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        total_squares += tet->num_squares;

        for (v = 0; v < 4; v++)
            total_triangles += tet->num_triangles[v];
    }

    num_edges = (4*total_squares + 3*total_triangles) / 2;
    num_faces = total_squares + total_triangles;

    /*

```

```
    *   Return the Euler characteristic.
    */
    return num_vertices - num_edges + num_faces;
}
```