

```

/*
 * Dirichlet_conversion.c
 *
 * This file contains the function
 *
 *      Triangulation *Dirichlet_to_triangulation(WEPolyhedron *polyhedron);
 *
 * which converts a Dirichlet domain to a Triangulation, leaving the
 * Dirichlet domain unchanged. For closed manifolds, drills out
 * an arbitrary curve and expresses the manifold as a Dehn filling.
 * The polyhedron must be a manifold; returns NULL for orbifolds.
 */

/*
 *
 *      The Algorithm
 *
 * When subdividing a Dirichlet domain into tetrahedra, one faces a
 * tradeoff between the ease of programming and efficiency of the resulting
 * triangulation. Plan A is the cleanest to implement, but uses twice
 * as many tetrahedra as Plan B, and four times as many as plan C.
 *
 * Plan A
 *
 * The vertices of each tetrahedron are as follows.
 *
 *      index    location
 *      0        at a vertex of the Dirichlet domain
 *      1        at the midpoint of an edge incident to the aforementioned vertex
 *      2        at the center of a face incident to the aforementioned edge
 *      3        at the center of the Dirichlet domain
 *
 * As a mnemonic, note that vertex i lies at the center of a cell
 * of dimension i. Make yourself a sketch of the Dirichlet domain,
 * and it will be obvious that tetrahedra of the above form triangulate
 * the manifold. Moreover, all tet->gluing[]'s are the identity (!)
 * so the implementation need only be concerned with the tet->neighbor[]'s.
 *
 * Plan B
 *
 * Fuse together pairs of tetrahedra from Plan A across their faces
 * of FaceIndex 0. This halves the number of tetrahedra required,
 * but makes the programming more complicated.
 *
 * Plan C
 *
 * Fuse together pairs of tetrahedra from Plan B across their faces
 * of FaceIndex 3 (i.e. across the faces of the original Dirichlet domain).
 * This halves the number of tetrahedra required, but makes the programming
 * more complicated.
 *
 * The Choice
 *
 * For now I will go with Plan A because it's simplest.
 * If memory use turns out to be a problem (which I suspect it won't)
 * I can rewrite the code to use Plan B or C instead. In any case,
 * note that the large number of Tetrahedra are required only temporarily,
 * and don't have TetShapes attached. The only remaining danger with this
 * plan is that in the case of a closed manifold, the drilled curve might
 * not be isotopic to the geodesic in its isotopy class.
 */

#include "kernel.h"

#define DEFAULT_NAME    "no name"
#define MAX_TRIES       16

static Triangulation    *try_Dirichlet_to_triangulation(WEPolyhedron *polyhedron);
static Boolean          singular_set_is_empty(WEPolyhedron *polyhedron);

Triangulation *Dirichlet_to_triangulation(
    WEPolyhedron    *polyhedron)
{
    /*

```

```

    * When the polyhedron represents a closed manifold,
    * try_Dirichlet_to_triangulation() drills out an arbitrary curve
    * to express the manifold as a Dehn filling. Usually the arbitrary
    * curve turns out to be isotopic to a geodesic, but not always.
    * Here we try several repetitions of try_Dirichlet_to_triangulation(),
    * if necessary, to obtain a hyperbolic Dehn filling.
    * Fortunately in almost all cases (about 95% in my informal tests)
    * try_Dirichlet_to_triangulation() get a geodesic on its first try.
    */

    int count;
    Triangulation *triangulation;

    triangulation = try_Dirichlet_to_triangulation(polyhedron);

    count = MAX_TRIES;
    while ( --count >= 0
        && triangulation != NULL
        && triangulation->solution_type[filled] != geometric_solution
        && triangulation->solution_type[filled] != nongeometric_solution)
    {
        free_triangulation(triangulation);
        triangulation = try_Dirichlet_to_triangulation(polyhedron);
    }

    return triangulation;
}

static Triangulation *try_Dirichlet_to_triangulation(
    WEPolyhedron *polyhedron)
{
    /*
     * Implement Plan A as described above.
     */

    Triangulation *triangulation;
    WEEdge *edge,
            *nbr_edge,
            *mate_edge;

    WEEdgeEnd end;
    WEEdgeSide side;
    Tetrahedron *new_tet;
    FaceIndex f;

    /*
     * Don't attempt to triangulate an orbifold.
     */

    if (singular_set_is_empty(polyhedron) == FALSE)
        return NULL;

    /*
     * Set up the Triangulation.
     */

    triangulation = NEW_STRUCT(Triangulation);
    initialize_triangulation(triangulation);

    /*
     * Allocate and copy the name.
     */

    triangulation->name = NEW_ARRAY(strlen(DEFAULT_NAME) + 1, char);
    strcpy(triangulation->name, DEFAULT_NAME);

    /*
     * Allocate the Tetrahedra.
     */

    triangulation->num_tetrahedra = 4 * polyhedron->num_edges;

    for (edge = polyhedron->edge_list_begin.next;
        edge != &polyhedron->edge_list_end;

```

```

    edge = edge->next)

    for (end = 0; end < 2; end++)    /* = tail, tip */

        for (side = 0; side < 2; side++)    /* = left, right */
        {
            new_tet = NEW_STRUCT(Tetrahedron);
            initialize_tetrahedron(new_tet);
            INSERT_BEFORE(new_tet, &triangulation->tet_list_end);
            edge->tet[end][side] = new_tet;
        }

/*
 * Initialize neighbors.
 */

for (edge = polyhedron->edge_list_begin.next;
    edge != &polyhedron->edge_list_end;
    edge = edge->next)

    for (end = 0; end < 2; end++)    /* = tail, tip */

        for (side = 0; side < 2; side++)    /* = left, right */
        {
            /*
             * Neighbor[0] is associated to this same WEEdge.
             * It lies on the same side (left or right), but
             * at the opposite end (tail or tip).
             */
            edge->tet[end][side]->neighbor[0] = edge->tet[!end][side];

            /*
             * Neighbor[1] lies on the same face of the Dirichlet
             * domain, but at the "next side" of that face.
             */
            nbr_edge = edge->e[end][side];
            if (nbr_edge->v[!end] == edge->v[end])
                /* edge and nbr_edge point in the same direction */
                edge->tet[end][side]->neighbor[1] = nbr_edge->tet[!end][side];
            else if (nbr_edge->v[end] == edge->v[end])
                /* edge and nbr_edge point in opposite directions */
                edge->tet[end][side]->neighbor[1] = nbr_edge->tet[end][!side];
            else
                uFatalError("Dirichlet_to_triangulation", "Dirichlet_conversion");

            /*
             * Neighbor[2] is associated to this same WEEdge.
             * It lies at the same end (tail or tip), but
             * on the opposite side (left or right).
             */
            edge->tet[end][side]->neighbor[2] = edge->tet[end][!side];

            /*
             * Neighbor[3] lies on this face's "mate" elsewhere
             * on the Dirichlet domain.
             */
            mate_edge = edge->neighbor[side];
            edge->tet[end][side]->neighbor[3] = mate_edge->tet
                [edge->preserves_direction[side] ? end : !end]
                [edge->preserves_sides[side] ? side : !side];
        }

/*
 * Initialize all gluings to the identity.
 */

for (edge = polyhedron->edge_list_begin.next;
    edge != &polyhedron->edge_list_end;
    edge = edge->next)

    for (end = 0; end < 2; end++)    /* = tail, tip */

        for (side = 0; side < 2; side++)    /* = left, right */

```

```

        for (f = 0; f < 4; f++)

            edge->tet[end][side]->gluing[f] = IDENTITY_PERMUTATION;

/*
 * Set up the EdgeClasses.
 */
create_edge_classes(triangulation);
orient_edge_classes(triangulation);

/*
 * Attempt to orient the manifold.
 */
orient(triangulation);

/*
 * Set up the Cusps, including "fake cusps" for the finite vertices.
 * Then locate and remove the fake cusps. If the manifold is closed,
 * drill out an arbitrary curve to express it as a Dehn filling.
 * Finally, determine the topology of each cusp (torus or Klein bottle)
 * and count them.
 */
create_cusps(triangulation);
mark_fake_cusps(triangulation);
peripheral_curves(triangulation);
remove_finite_vertices(triangulation);
count_cusps(triangulation);

/*
 * Try to compute a hyperbolic structure, first for the unfilled
 * manifold, and then for the closed manifold if appropriate.
 */
find_complete_hyperbolic_structure(triangulation);
do_Dehn_filling(triangulation);

/*
 * If the manifold is hyperbolic, install a shortest basis on each cusp.
 */
if (    triangulation->solution_type[complete] == geometric_solution
    || triangulation->solution_type[complete] == nongeometric_solution)
    install_shortest_bases(triangulation);

/*
 * All done!
 */
return triangulation;
}

static Boolean singular_set_is_empty(
    WEPolyhedron    *polyhedron)
{
    /*
     * Check whether the singular set of this orbifold is empty.
     */

    WEVertexClass    *vertex_class;
    WEEdgeClass       *edge_class;
    WEFaceClass       *face_class;

    for (vertex_class = polyhedron->vertex_class_begin.next;
         vertex_class != &polyhedron->vertex_class_end;
         vertex_class = vertex_class->next)

        if (vertex_class->singularity_order >= 2)

            return FALSE;

    /*
     * Dirichlet_construction.c subdivides Dirichlet domains for
     * orbifolds so that the k-skeleton of the singular set lies
     * in the k-skeleton of the Dirichlet domain (k = 0,1,2).
     * Thus if there are no singular VertexClasses, there can't be
     * any singular EdgeClasses or FaceClasses either.
     */

```

```
    */

    for (edge_class = polyhedron->edge_class_begin.next;
         edge_class != &polyhedron->edge_class_end;
         edge_class = edge_class->next)

        if (edge_class->singularity_order >= 2)

            uFatalError("singular_set_is_empty", "Dirichlet_conversion");

    for (face_class = polyhedron->face_class_begin.next;
         face_class != &polyhedron->face_class_end;
         face_class = face_class->next)

        if (face_class->num_elements != 2)

            uFatalError("singular_set_is_empty", "Dirichlet_conversion");

    /*
     * No singularities are present.
     */

    return TRUE;
}
```