

```

/*
 * finite_vertices.c
 *
 * Certain routines make temporary use of finite (as opposed to ideal)
 * vertices; e.g. they are used in triangulating a link complement,
 * in retriangulating a partially filled multicusp manifold, and in
 * splitting along a normal surface. SnapPea represents a finite vertex
 * as a cusp whose is_finite field is set to TRUE. Except for the
 * is_finite, index, prev and next fields, all other fields of the Cusp
 * data structure are ignored. The indices are negative integers.
 * Functions which do not use finite vertices may safely ignore the
 * is_finite field, and assume no finite vertices are present.
 * Finite vertices are never counted in the num_cusps, num_or_cusps,
 * or num_nonor_cusps fields of a Triangulation.
 *
 * This file contains the function
 *
 * void remove_finite_vertices(Triangulation *manifold);
 *
 * which is used within the kernel to retriangulate the manifold
 * to remove the finite vertices. If manifold has no real cusps,
 * a single real cusp is created (it may be either a torus or
 * Klein bottle cusp).
 *
 * Technical note: It's OK to pass a manifold with some or all
 * of the peripheral curves missing, the cusp topologies unknown,
 * and num_or_cusps and/or num_nonor_cusps not set. (For example,
 * normal_surface_splitting.c does exactly that.) Of course any
 * peripheral curves which are known will be preserved.
 */

/*
 *
 * The Algorithm
 *
 * Overview
 *
 * A finite vertex is represented by a "cusp" whose cross section is
 * topologically a sphere (in contrast to real cusps, whose cross sections
 * are always tori or Klein bottles). Throughout this documentation,
 * please imagine all tetrahedra to have truncated vertices, so that
 * a finite vertex appears as a spherical boundary component, and a
 * real cusp appears as a torus or Klein bottle boundary component.
 * To "remove a finite vertex", we'll modify the triangulation so as
 * to drill out a tube connecting a spherical boundary component to
 * a nearby torus or Klein bottle boundary component. We'll repeat
 * the procedure until no spherical boundary components remain.
 *
 * Details
 *
 * The manifold is assumed to be connected, so as long as spherical
 * boundary components remain we may find an edge E (in the triangulation
 * of the manifold) connecting a spherical boundary component to a
 * torus or Klein bottle boundary component. Let T be any triangle
 * (in the triangulation of the manifold) incident to E.
 *
 * If we cut along the triangle T, insert a triangular pillow, and reglue,
 * the topology of the manifold doesn't change. But instead of inserting
 * an ordinary triangular pillow, we'll insert a triangular pillow from
 * which a "tunnel" has been drilled out, so as to connect one of its
 * truncated vertices to another. (The tunnel is unknotted, but it
 * really doesn't matter.)
 *
 * A triangular-pillow-with-tunnel may be constructed from only two
 * ideal tetrahedra, according to the following gluings (the notation
 * is as in the file TriangulationFileFormat).
 *
 * tetrahedron 0
 *      1      free      free      1
 *      0213      ----      ----      1023
 *
 * tetrahedron 1
 *      0      1      1      0
 *      0213      0213      0213      1023
 */

```

```

* Note: In my drawing the two tetrahedra are glued together along
* face 0 to form a hexahedron. Vertex 0 of tetrahedron 0 appears
* at the "north pole", vertex 0 of tetrahedron 1 appears at the
* "south pole", and the remain three vertices appear along the "equator".
*
* Closed Manifolds
*
* If the triangulation has no real cusps, then an arbitrary spherical
* boundary component is selected, and all other spherical boundary
* components are connected to it as above. This yields a manifold
* with a single spherical boundary component. An additional
* triangular-pillow-with-tunnel is added to convert the spherical
* boundary component to a torus or Klein bottle boundary.
*
* Acknowledgements
*
* My path to this algorithm was indirect. I thank Sergei Matveev
* for suggesting I think about manifolds in terms of spines, and
* I thank Carlo Petronio for helpful discussions which led me in
* the direction of this construction.
*/

#include "kernel.h"

static void initialize_matching_cusps(Triangulation *manifold, Cusp **special_fake_cusp);
static void merge_cusps(Triangulation *manifold);
static void drill_tube(Triangulation *manifold, Tetrahedron *tet, EdgeIndex e, Boolean
    creating_new_cusp);
static void set_real_cusps(Triangulation *manifold, Cusp *special_fake_cusp);

void remove_finite_vertices(
    Triangulation *manifold)
{
    Cusp *special_fake_cusp;

    /*
     * Simplify the triangulation before we begin.
     * basic_simplification() should work OK even with finite vertices.
     */
    basic_simplification(manifold);

    /*
     * The matching_cusp field of each fake cusp records the real cusp
     * to which the fake cusp has been connected. It's initialized to
     * NULL to indicate that the fake cusp has not yet been connected
     * to anything. For real cusps, it's convenient to have the
     * matching_cusp field always point to the cusp itself. If the
     * manifold has no real cusps, then choose a "special fake cusp"
     * to which all other fake cusps will be connected, and set its
     * matching_cusp field to point to itself.
     */
    initialize_matching_cusps(manifold, &special_fake_cusp);

    /*
     * Keep merging fake cusps with real cusps until no further progress
     * is possible.
     */
    merge_cusps(manifold);

    /*
     * Ideal vertices which used to be incident to fake cusps are
     * now all incident to real cusps. Update the tet->cusp[] fields,
     * and free the fake cusps (except for the special_fake_cusp, if any).
     */
    set_real_cusps(manifold, special_fake_cusp);

    /*
     * If the manifold is closed (no real cusps) it will, at this point,
     * have one spherical "cusp", namely the special_fake_cusp.
     * Drill out a tube connecting the special_fake_cusp to itself,
     * to convert it from a sphere to a torus or Klein bottle.
     */
    if (special_fake_cusp != NULL)

```

```

{
    /*
     * Simplify the triangulation before drilling,
     * to increase the chances that the drilled out tube will
     * follow a topologically nontrivial loop through the manifold.
     * (This is essential if we want to express the resulting
     * closed manifold as a hyperbolic Dehn filling.)
     */
    basic_simplification(manifold);
    drill_tube(manifold, manifold->tet_list_begin.next, 0, TRUE);
}

/*
 * The triangulation is now correct, but it is horribly inefficient.
 * Simplify it.
 *
 * Note: basic_simplification() calls tidy_peripheral_curves()
 * and compute_CS_fudge_from_value().
 */
basic_simplification(manifold);
}

static void initialize_matching_cusps(
    Triangulation *manifold,
    Cusp **special_fake_cusp)
{
    Boolean has_real_cusp;
    Cusp *cusp;

    has_real_cusp = FALSE;

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)

        if (cusp->is_finite)
            cusp->matching_cusp = NULL;
        else
        {
            cusp->matching_cusp = cusp;
            has_real_cusp = TRUE;
        }

    if (has_real_cusp == FALSE)
    {
        *special_fake_cusp = manifold->cusp_list_begin.next;
        (*special_fake_cusp)->matching_cusp = *special_fake_cusp;
    }
    else
        *special_fake_cusp = NULL;
}

static void merge_cusps(
    Triangulation *manifold)
{
    Boolean progress;
    EdgeClass *edge;
    Tetrahedron *tet;
    EdgeIndex e;
    Cusp *one_cusp,
        *other_cusp;

    do
    {
        progress = FALSE;

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

```

```

    one_cusp    = tet->cusp[one_vertex_at_edge[e]];
    other_cusp  = tet->cusp[other_vertex_at_edge[e]];

    if (one_cusp->matching_cusp == NULL
        && other_cusp->matching_cusp != NULL)
    {
        one_cusp->matching_cusp = other_cusp->matching_cusp;
        drill_tube(manifold, tet, e, FALSE);
        progress = TRUE;
    }

    if (other_cusp->matching_cusp == NULL
        && one_cusp->matching_cusp != NULL)
    {
        other_cusp->matching_cusp = one_cusp->matching_cusp;
        drill_tube(manifold, tet, e, FALSE);
        progress = TRUE;
    }
}
while (progress == TRUE);
}

static void drill_tube(
    Triangulation *manifold,
    Tetrahedron   *tet,
    EdgeIndex      e,
    Boolean        creating_new_cusp)
{
    /*
     * Insert a triangular-pillow-with-tunnel (as described at the top
     * of this file) so as to connect the boundary component at one
     * end of the given edge to the boundary component at the other end.
     * The orientation on the triangular pillow will match the
     * orientation on tet, so that the orientation on the manifold
     * (if there is one) will be preserved.  Edge orientations are also
     * respected.
     */

    VertexIndex    v0,
                   v1,
                   v2,
                   vv0,
                   vv1,
                   vv2;
    FaceIndex      f,
                   ff;
    Tetrahedron    *nbr_tet,
                   *new_tet0,
                   *new_tet1;
    Permutation     gluing;
    EdgeClass       *edge0,
                   *edge1,
                   *edge2,
                   *new_edge;
    Orientation     edge_orientation0,
                   edge_orientation1,
                   edge_orientation2;
    PeripheralCurve c;
    Orientation     h;
    int             num_strands,
                   intersection_number[2],
                   the_gcd;
    Cusp            *unique_cusp;
    MatrixInt22     basis_change[1];

    /*
     * Relative to the orientation of tet, the vertices v0, v1 and v2
     * are arranged in counterclockwise order around the face f.
     */
    v0 = one_vertex_at_edge[e];
    v1 = other_vertex_at_edge[e];
    v2 = remaining_face[v1][v0];

```

```

f = remaining_face[v0][v1];

/*
 * Note the matching face and its vertices.
 */
nbr_tet = tet->neighbor[f];
gluing = tet->gluing[f];
ff      = EVALUATE(gluing, f);
vv0     = EVALUATE(gluing, v0);
vv1     = EVALUATE(gluing, v1);
vv2     = EVALUATE(gluing, v2);

/*
 * Note the incident EdgeClasses (which may or may not be distinct).
 */
edge0 = tet->edge_class[e];
edge1 = tet->edge_class[edge_between_vertices[v1][v2]];
edge2 = tet->edge_class[edge_between_vertices[v2][v0]];

/*
 * Construct the triangular-pillow-with-tunnel, as described
 * at the top of this file.
 */

new_tet0 = NEW_STRUCT(Tetrahedron);
new_tet1 = NEW_STRUCT(Tetrahedron);
initialize_tetrahedron(new_tet0);
initialize_tetrahedron(new_tet1);
INSERT_BEFORE(new_tet0, &manifold->tet_list_end);
INSERT_BEFORE(new_tet1, &manifold->tet_list_end);
manifold->num_tetrahedra += 2;

new_edge = NEW_STRUCT(EdgeClass);
initialize_edge_class(new_edge);
INSERT_BEFORE(new_edge, &manifold->edge_list_end);

new_tet0->neighbor[0] = new_tet1;
new_tet0->neighbor[1] = NULL; /* assigned below */
new_tet0->neighbor[2] = NULL; /* assigned below */
new_tet0->neighbor[3] = new_tet1;

new_tet1->neighbor[0] = new_tet0;
new_tet1->neighbor[1] = new_tet1;
new_tet1->neighbor[2] = new_tet1;
new_tet1->neighbor[3] = new_tet0;

new_tet0->gluing[0] = CREATE_PERMUTATION(0, 0, 1, 2, 2, 1, 3, 3);
new_tet0->gluing[1] = 0x00; /* assigned below */
new_tet0->gluing[2] = 0x00; /* assigned below */
new_tet0->gluing[3] = CREATE_PERMUTATION(0, 1, 1, 0, 2, 2, 3, 3);

new_tet1->gluing[0] = CREATE_PERMUTATION(0, 0, 1, 2, 2, 1, 3, 3);
new_tet1->gluing[1] = CREATE_PERMUTATION(0, 0, 1, 2, 2, 1, 3, 3);
new_tet1->gluing[2] = CREATE_PERMUTATION(0, 0, 1, 2, 2, 1, 3, 3);
new_tet1->gluing[3] = CREATE_PERMUTATION(0, 1, 1, 0, 2, 2, 3, 3);

new_tet0->edge_class[0] = edge1;
new_tet0->edge_class[1] = edge1;
new_tet0->edge_class[2] = edge0;
new_tet0->edge_class[3] = edge2;
new_tet0->edge_class[4] = edge0;
new_tet0->edge_class[5] = edge0;

new_tet1->edge_class[0] = edge1;
new_tet1->edge_class[1] = edge1;
new_tet1->edge_class[2] = edge0;
new_tet1->edge_class[3] = new_edge;
new_tet1->edge_class[4] = edge0;
new_tet1->edge_class[5] = edge0;

edge0->order += 6;
edge1->order += 4;
edge2->order += 1;

```

```

new_edge->order = 1;
new_edge->incident_tet = new_tet1;
new_edge->incident_edge_index = 3;

edge_orientation0 = tet->edge_orientation[e];
edge_orientation1 = tet->edge_orientation[edge_between_vertices[v1][v2]];
edge_orientation2 = tet->edge_orientation[edge_between_vertices[v2][v0]];

new_tet0->edge_orientation[0] = edge_orientation1;
new_tet0->edge_orientation[1] = edge_orientation1;
new_tet0->edge_orientation[2] = edge_orientation0;
new_tet0->edge_orientation[3] = edge_orientation2;
new_tet0->edge_orientation[4] = edge_orientation0;
new_tet0->edge_orientation[5] = edge_orientation0;

new_tet1->edge_orientation[0] = edge_orientation1;
new_tet1->edge_orientation[1] = edge_orientation1;
new_tet1->edge_orientation[2] = edge_orientation0;
new_tet1->edge_orientation[3] = right_handed;
new_tet1->edge_orientation[4] = edge_orientation0;
new_tet1->edge_orientation[5] = edge_orientation0;

new_tet0->cusp[0] = tet->cusp[v0];
new_tet0->cusp[1] = tet->cusp[v0];
new_tet0->cusp[2] = tet->cusp[v0];
new_tet0->cusp[3] = tet->cusp[v2];

new_tet1->cusp[0] = tet->cusp[v0];
new_tet1->cusp[1] = tet->cusp[v0];
new_tet1->cusp[2] = tet->cusp[v0];
new_tet1->cusp[3] = tet->cusp[v2];

/*
 * Install the triangular-pillow-with-tunnel.
 */

tet->neighbor[f] = new_tet0;
tet->gluing[f] = CREATE_PERMUTATION(f, 2, v0, 0, v1, 1, v2, 3);
new_tet0->neighbor[2] = tet;
new_tet0->gluing[2] = inverse_permutation[tet->gluing[f]];

nbr_tet->neighbor[ff] = new_tet0;
nbr_tet->gluing[ff] = CREATE_PERMUTATION(ff, 1, vv0, 0, vv1, 2, vv2, 3);
new_tet0->neighbor[1] = nbr_tet;
new_tet0->gluing[1] = inverse_permutation[nbr_tet->gluing[ff]];

/*
 * Typically creating_new_cusp is FALSE, meaning that we are
 * connecting a spherical boundary component to a torus or
 * Klein bottle boundary component, and we simply extend the
 * existing peripheral curves across the new tetrahedra.
 *
 * In the exceptional case that creating_new_cusp is TRUE,
 * meaning that the manifold has no real cusps and we are
 * connecting the "special fake cusp" to itself, we must
 * install a meridian and longitude, and set up the Dehn filling.
 */
if (creating_new_cusp == FALSE)
{
    /*
     * Extend the peripheral curves across the boundary of the
     * triangular-pillow-with-tunnel.
     *
     * Note: The orientations of new_tet0 and new_tet1 match that
     * of tet, so the right_handed and left_handed sheets match up
     * in the obvious way.
     */

    for (c = 0; c < 2; c++) /* c = M, L */
        for (h = 0; h < 2; h++) /* h = right_handed, left_handed */
        {
            num_strands = tet->curve[c][h][v0][f];
            new_tet0->curve[c][h][0][2] = -num_strands;
            new_tet0->curve[c][h][0][1] = +num_strands;

```

```

        num_strands = tet->curve[c][h][v1][f];
        new_tet0->curve[c][h][1][2] = -num_strands;
        new_tet0->curve[c][h][1][0] = +num_strands;
        new_tet1->curve[c][h][2][0] = -num_strands;
        new_tet1->curve[c][h][2][1] = +num_strands;
        new_tet1->curve[c][h][1][2] = -num_strands;
        new_tet1->curve[c][h][1][0] = +num_strands;
        new_tet0->curve[c][h][2][0] = -num_strands;
        new_tet0->curve[c][h][2][1] = +num_strands;

        num_strands = tet->curve[c][h][v2][f];
        new_tet0->curve[c][h][3][2] = -num_strands;
        new_tet0->curve[c][h][3][1] = +num_strands;
    }
}
else /* creating_new_cusp == TRUE */
{
    /*
     * We have just installed a tube connecting the (unique)
     * spherical "cusp" to itself, to convert it to a torus or
     * Klein bottle.
     */
    unique_cusp = tet->cusp[v0]->matching_cusp;
    unique_cusp->is_complete = TRUE; /* to be filled below */
    unique_cusp->index = 0;
    unique_cusp->is_finite = FALSE;
    manifold->num_cusps = 1;

    /*
     * Install an arbitrary meridian and longitude.
     */
    peripheral_curves(manifold);
    count_cusps(manifold);

    /*
     * Two sides of the (truncated) vertex 0 of new_tet0
     * (namely the sides incident to faces 1 and 2 of new_tet0)
     * define the Dehn filling curve by which we can recover
     * the closed manifold. Count how many times the newly
     * installed meridian and longitude cross this Dehn filling curve.
     * To avoid messy questions about which sheet of the cusp's
     * double cover we're on, use two (parallel) copies of the
     * Dehn filling curve, one on each sheet of the cover.
     * Ultimately we're looking for a linear combination of the
     * meridian and longitude whose intersection number with
     * the Dehn filling curve is zero, so it won't matter if
     * we're off by a factor of two.
     */
    for (c = 0; c < 2; c++) /* c = M, L */
    {
        intersection_number[c] = 0;

        for (h = 0; h < 2; h++) /* h = right_handed, left_handed */
        {
            intersection_number[c] += new_tet0->curve[c][h][0][1];
            intersection_number[c] += new_tet0->curve[c][h][0][2];
        }
    }

    /*
     * Use the intersection numbers to deduce
     * the desired Dehn filling coefficients.
     */
    the_gcd = gcd(intersection_number[M], intersection_number[L]);
    unique_cusp->is_complete = FALSE;
    unique_cusp->m = -intersection_number[L] / the_gcd;
    unique_cusp->l = +intersection_number[M] / the_gcd;

    /*
     * Switch to a basis in which the Dehn filling curve is a meridian.
     */
    unique_cusp->cusp_shape[initial] = Zero; /* force current_curve_basis() to
    ignore the cusp shape */

```

```
        current_curve_basis(manifold, 0, basis_change[0]);
        if (change_peripheral_curves(manifold, basis_change) != func_OK)
            uFatalError("drill_tube", "finite_vertices");
    }
}

static void set_real_cusps(
    Triangulation *manifold,
    Cusp *special_fake_cusp)
{
    Tetrahedron *tet;
    int i;
    Cusp *cusp,
        *dead_cusp;

    /*
     * Update the cusp fields.
     */

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)

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

            tet->cusp[i] = tet->cusp[i]->matching_cusp;

    /*
     * Free the Cusp structures which had been used for finite vertices.
     */

    for (cusp = manifold->cusp_list_begin.next;
         cusp != &manifold->cusp_list_end;
         cusp = cusp->next)

        if (cusp->is_finite == TRUE
            && cusp != special_fake_cusp)
        {
            dead_cusp = cusp;
            cusp = cusp->prev; /* so the loop will proceed correctly */
            REMOVE_NODE(dead_cusp);
            my_free(dead_cusp);
        }
}
```