

```

/*
 * choose_generators.c
 *
 * This file contains the function
 *
 * void choose_generators( Triangulation *manifold,
 *                          Boolean compute_corners,
 *                          Boolean centroid_at_origin)
 *
 * which chooses a set of generators for the fundamental group
 * of the Triangulation *manifold. (The Dehn filling coefficients
 * are irrelevant.)
 *
 * A function which needs to use the generating set must first call
 * choose_generators(). [Note that this differs from the previous
 * SnapPea 2.0- convention, under which all functions which changed the
 * triangulation were responsible for calling choose_generators().
 * The old convention was more efficient at runtime, but the new one
 * makes programming easier.]
 *
 * The algorithm begins with an arbitrary Tetrahedron, and recursively
 * attaches neighboring Tetrahedra to create a fundamental domain for
 * the manifold which is topologically a ball. Whenever a face of a
 * Tetrahedron lies in the interior of this fundamental domain,
 * tet->generator_status[face] is set to not_a_generator. Faces on the
 * exterior of the fundamental domain correspond to active generators,
 * and will have status outbound_generator or inbound_generator, depending
 * on how a particular generator is oriented (one face of a matching pair
 * will have status outbound_generator, and the other inbound_generator).
 *
 * The algorithm simplifies the generating set in two ways:
 *
 * (1) When it finds an EdgeClass with only one incident 2-cell which
 * is dual to an active generator, it does a "handle cancellation"
 * to eliminate that generator, and also sets the EdgeClass's
 * active_relation field to FALSE. The algorithm continues doing
 * this type of simplification until it can make no further progress.
 *
 * (2) At this point the boundary of the fundamental domain is likely
 * to contain groups of faces which are essentially n-gons (n > 3)
 * arbitrarily divided into triangles. The generators for such
 * triangular faces are all equivalent, and get merged. The
 * active_relation fields of the interior EdgeClasses are set to
 * FALSE.
 *
 * active_relation fields which are not set to FALSE in (1) or (2)
 * are set to TRUE. Each EdgeClass's num_incident_generators field
 * says, not surprisingly, how many active generators it is incident to.
 * Note that num_incident_generators becomes 0 when a handle cancellation
 * occurs in (1) above, but num_incident_generators remains 2 when the
 * EdgeClass's active_relation field is set to FALSE in (2) (the rationale
 * is that there are still two active incident generators, even though
 * they happen to be the same (yeah, it sounds suspicious to me too, but
 * that's how it is)).
 *
 * Each generator is a 1-cell in the dual to the Triangulation.
 * The generator dual to a given face of a given Tetrahedron is
 * described by three variables:
 *
 * tet->generator_status[face] takes the value
 *
 * outbound_generator    if the generator is directed from
 *                        tet towards its neighbor,
 * inbound_generator     if the generator is directed from
 *                        the neighbor towards tet,
 * not_a_generator       if no generator corresponds to this
 *                        face (more on this in a minute), and
 * unassigned_generator  if the algorithm hasn't gotten around
 *                        to considering this face yet.
 *
 * tet->generator_index[face] tells the index of the generator.
 * The numbering runs from 0 to (number-of-generators - 1).
 * tet->generator_index[face] is defined iff tet->generator_status[face]
 * is outbound_generator or inbound_generator.

```

```

*
*     tet->generator_parity[face] tells whether the generator is
*     orientation_preserving or orientation_reversing.
*
* The field tet->generator_path lets you reconstruct the complete path of
* a generator: it says by which face the given Tetrahedron was added to the
* fundamental domain (cf. the recursive algorithm described above). The
* central Tetrahedron used to begin the recursion has tet->generator_path = -1.
*
* If compute_corners is TRUE,
* choose_generators() also computes the location on the sphere at infinity
* of each ideal vertex of each Tetrahedron in the fundamental domain, and
* stores it in the field tet->corner[vertex]. That is, tet->corner[vertex]
* contains the complex number representing the location of the vertex in
* the boundary of the upper half space model. The (relative) locations of
* the corners are computed using the hyperbolic structure of the Dehn filled
* manifold. If centroid_at_origin is TRUE, the initial tetrahedron is
* positioned with its centroid at the origin; otherwise the initial tetrahedron
* is positioned with its vertices at {0, 1/sqrt(z), sqrt(z), infinity}.
*/

#include "kernel.h"

static void initialize_flags(Triangulation *manifold);
static void visit_tetrahedra(Triangulation *manifold, Boolean compute_corners, Boolean
centroid_at_origin);
static void initial_tetrahedron(Triangulation *manifold, Tetrahedron **tet, Boolean
compute_corners, Boolean centroid_at_origin);
static void count_incident_generators(Triangulation *manifold);
static void eliminate_trivial_generators(Triangulation *manifold);
static void kill_the_incident_generator(Triangulation *manifold, EdgeClass *edge);
static void merge_equivalent_generators(Triangulation *manifold);
static void merge_incident_generators(Triangulation *manifold, EdgeClass *edge);

void choose_generators(
    Triangulation *manifold,
    Boolean compute_corners,
    Boolean centroid_at_origin)
{
    /*
     * To compute the corners we need some sort of geometric structure.
     */
    if (compute_corners == TRUE
        && manifold->solution_type[filled] == not_attempted)
        uFatalError("choose_generators", "choose_generators.c");

    /*
     * For each Tetrahedron tet, set tet->flag to unknown_orientation
     * to indicate that the Tetrahedron has not yet been visited, and
     * set each tet->generator_status[i] to unassigned_generator to
     * indicate that no generator has yet been assigned to any face.
     */
    initialize_flags(manifold);

    /*
     * Start a recursion which visits each tetrahedron, assigns
     * generators to its faces, and recursively visits any unvisited
     * neighbors.
     */
    visit_tetrahedra(manifold, compute_corners, centroid_at_origin);

    /*
     * The number_of_generators should be one plus the number of tetrahedra.
     */
    if (manifold->num_generators != manifold->num_tetrahedra + 1)
        uFatalError("choose_generators", "choose_generators.c");

    /*
     * At this point we have a valid set of generators, but it's
     * not as simple as it might be. We'll perform two types of
     * simplifications. First we need to count how many of the
     * faces incident to each EdgeClass correspond to active generators.
     */

```

```

    * Initialize all the active_relation flags to TRUE while we're at it.
    */
count_incident_generators(manifold);

/* Now look for EdgeClasses in the Triangulation (2-cells in the
 * dual complex) which show that a single generator is homotopically
 * trivial, and eliminate the trivial generator. Topologically, this
 * corresponds to folding together two adjacent triangular faces
 * on the boundary of the fundamental domain (the close-the-book
 * move). Geometrically, this corresponds to realizing that two
 * faces of the (geometric) fundamental domain are in fact already
 * superimposed on each other. In Heegaard terms, it's a handle
 * cancellation.
 */
eliminate_trivial_generators(manifold);

/*
 * At this point the boundary of the fundamental domain is likely
 * to contain groups of faces which are essentially n-gons (n > 3)
 * arbitrarily divided into triangles. The generators for such
 * triangular faces are all equivalent, and can be merged. They
 * can be recognized by looking for EdgeClasses with exactly two
 * incident (and distinct) generators.
 */
merge_equivalent_generators(manifold);
}

static void initialize_flags(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    FaceIndex face;

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

        for (face = 0; face < 4; face++)
        {
            tet->generator_status[face] = unassigned_generator;
            tet->generator_index[face] = -2; /* garbage value */
        }
    }
}

static void visit_tetrahedra(
    Triangulation *manifold,
    Boolean compute_corners,
    Boolean centroid_at_origin)
{
    Tetrahedron **queue,
    *tet;
    int queue_first,
    queue_last;
    Tetrahedron *nbr_tet;
    Permutation gluing;
    FaceIndex face,
    nbr_face;
    int i;
    VertexIndex nbr_i;

    /*
     * choose_generators() has already called initialize_flags().
     */

    /*
     * Initialize num_generators to zero.
     */
    manifold->num_generators = 0;

```

```

/*
 * Allocate space for a queue of pointers to the Tetrahedra.
 * Each Tetrahedron will appear on the queue exactly once,
 * so an array of length manifold->num_tetrahedra will be just right.
 */
queue = NEW_ARRAY(manifold->num_tetrahedra, Tetrahedron *);

/*
 * Initialize the queue.
 */
queue_first = 0;
queue_last = 0;

/*
 * Choose the initial Tetrahedron according to some criterion.
 * If compute_corners is TRUE, position its corners.
 * 2000/4/2 The choice of initial tetrahedron is independent
 * of compute_corners.
 */
initial_tetrahedron(manifold, &queue[0], compute_corners, centroid_at_origin);

/*
 * Mark the initial Tetrahedron as visited.
 */
queue[0]->generator_path = -1;
queue[0]->flag = right_handed;

/*
 * Start processing the queue.
 */
do
{
    /*
     * Pull a Tetrahedron off the front of the queue.
     */
    tet = queue[queue_first++];

    /*
     * Look at the four neighboring Tetrahedra.
     */
    for (face = 0; face < 4; face++)
    {
        /*
         * Note who the neighbor is, and which of
         * its faces we're glued to.
         */
        nbr_tet = tet->neighbor[face];
        gluing = tet->gluing[face];
        nbr_face = EVALUATE(gluing, face);

        /*
         * If nbr_tet hasn't been visited, set the appropriate
         * generator_statuses to not_a_generator, and then put
         * nbr_tet on the back of the queue.
         */
        if (nbr_tet->flag == unknown_orientation)
        {
            tet->generator_status[face] = not_a_generator;
            nbr_tet->generator_status[nbr_face] = not_a_generator;

            tet->generator_index[face] = -1; /* garbage value */
            nbr_tet->generator_index[nbr_face] = -1;

            nbr_tet->generator_path = nbr_face;

            nbr_tet->flag = (parity[gluing] == orientation_preserving) ?
                tet->flag :
                ! tet->flag;

            if (compute_corners)
            {
                for (i = 0; i < 4; i++)
                {
                    if (i == face)

```

```

        continue;
        nbr_i = EVALUATE(gluing, i);
        nbr_tet->corner[nbr_i] = tet->corner[i];
    }
    compute_fourth_corner(
        nbr_tet->corner, /* array of corner coordinates */
        nbr_face, /* the corner to be computed */
        nbr_tet->flag, /* nbr_tet's current orientation */
        nbr_tet->shape[filled]->cwl[ultimate]); /* shapes */
    }

    queue[++queue_last] = nbr_tet;
}
/*
 * If nbr_tet has been visited, check whether a generator
 * has been assigned to common face, and if not, assign one.
 */
else if (tet->generator_status[face] == unassigned_generator)
{
    tet->generator_status[face] = outbound_generator;
    nbr_tet->generator_status[nbr_face] = inbound_generator;

    tet->generator_index[face] = manifold->num_generators;
    nbr_tet->generator_index[nbr_face] = manifold->num_generators;

    tet->generator_parity[face] =
    nbr_tet->generator_parity[nbr_face] = ((parity[gluing] ==
orientation_preserving)
                                         == (tet->flag == nbr_tet->flag)) ?
                                         orientation_preserving :
                                         orientation_reversing;

    manifold->num_generators++;
}
}
}
while (queue_first <= queue_last);

/*
 * Free the memory used for the queue.
 */
my_free(queue);

/*
 * An "unnecessary" (but quick) error check.
 */
if (
    queue_first != manifold->num_tetrahedra
    || queue_last != manifold->num_tetrahedra - 1)
    uFatalError("visit_tetrahedra", "choose_generators");
}

static void initial_tetrahedron(
    Triangulation *manifold,
    Tetrahedron **initial_tet,
    Boolean compute_corners,
    Boolean centroid_at_origin)
{
    VertexIndex v[4];
    Complex z,
        sqrt_z,
        w[4];
    Tetrahedron *tet;
    EdgeIndex best_edge,
        edge;

    /*
     * Set a default choice of tetrahedron and edge.
     */
    *initial_tet = manifold->tet_list_begin.next;
    best_edge = 0;

    /*
     * 2000/02/11 JRW Can we choose the initial tetrahedron in such

```

```

* a way that if we happen to have the canonical triangulation
* of a 2-bridge knot or link complement, the basepoint falls
* at a center of D2 symmetry? That is, can we find a Tetrahedron
* that looks like the "top of the tower" in the canonical
* triangulation of a 2-bridge knot or link complement?
*/
for (tet = manifold->tet_list_begin.next;
    tet != &manifold->tet_list_end;
    tet = tet->next)
    for (edge = 0; edge < 6; edge++)
        if (tet->neighbor[one_face_at_edge [edge]]
            == tet->neighbor[other_face_at_edge[edge]])
            {
                *initial_tet      = tet;
                best_edge          = edge;
            }

if (compute_corners)
{
    if (centroid_at_origin == TRUE)
    {
        /*
        * Proposition. For any value of w, positioning the corners at
        *
        *             corner[0] = w
        *             corner[1] = w^-1
        *             corner[2] = -w^-1
        *             corner[3] = -w
        *
        * defines a tetrahedron with its centroid at the "origin" and
        * the common perpendiculars between pairs of opposite edges
        * coincident with the "coordinate axes". [In the Klein model,
        * the tetrahedron is inscribed in a rectangular box whose faces
        * are parallel to the coordinate axes.]
        *
        * Proof: Use the observation that the line from a0 to a1 will
        * intersect the line from b0 to b1 iff the cross ratio
        *
        *             (b0 - a0) (b1 - a1)
        *             -----
        *             (b1 - a0) (b0 - a1)
        *
        * of the tetrahedron they span is real, and they will be
        * orthogonal iff the cross ratio is -1.
        *
        * [-w, w] is orthogonal to [0, infinity] because
        *
        *             (0 - -w) (infinity - w)
        *             ----- = -1
        *             (infinity - -w) (0 - w)
        *
        * and similarly for [-w^-1, w^-1] and [0, infinity].
        *
        * [w^-1, w] is orthogonal to [-1, 1] because
        *
        *             (-1 - w^-1) (1 - w)
        *             ----- = -1
        *             (1 - w^-1) (-1 - w)
        *
        * and similarly for [-w^-1, -w] and [-1, 1].
        *
        * [-w^-1, w] is orthogonal to [-i, i] because
        *
        *             (-i - -w^-1) (i - w)
        *             ----- = -1
        *             (i - -w^-1) (-i - w)
        *
        * and similarly for [w^-1, -w] and [-i, i].
        *
        * Q.E.D.
        *
        * The tetrahedron will have the correct cross ratio z iff
        */
    }
}

```

```

*      (w - -w^-1) (w^-1 - -w )      (w + w^-1)^2
*      z = ----- = -----
*      (w - -w ) (w^-1 - -w^-1)      4
*
* Solving for w in terms of z gives the four possibilities
*
*      w = +- (sqrt(z) +- sqrt(z - 1))
*
* Note that sqrt(z) + sqrt(z - 1) and sqrt(z) - sqrt(z - 1) are
* inverses of one another. We can choose any of the four solutions
* to be "w", and the other three will automatically become w^-1,
* -w, and -w^-1.
*
* Comment: This position for the initial corners brings out
* nice numerical properties in the O(3,1) matrices for manifolds
* composed of regular ideal tetrahedra (cf. the proofs in the
* directory "Tilings of H^3", which aren't part of SnapPea, but
* I could give you a copy).
*/

z = (*initial_tet)->shape[filled]->cwl[ultimate][0].rect;

w[0] = complex_plus(
    complex_sqrt(z),
    complex_sqrt(complex_minus(z, One))
);
w[1] = complex_div(One, w[0]);
w[2] = complex_negate(w[1]);
w[3] = complex_negate(w[0]);

(*initial_tet)->corner[0] = w[0];
(*initial_tet)->corner[1] = w[1];
(*initial_tet)->corner[2] = w[2];
(*initial_tet)->corner[3] = w[3];
}
else
{
    /*
    * Originally this code positioned the Tetrahedron's vertices
    * at {0, 1, z, infinity}. As of 2000/02/04 I modified it
    * to put the vertices at {0, 1/sqrt(z), sqrt(z), infinity} instead,
    * so that the basepoint (0,0,1) falls at the midpoint
    * of the edge extending from 0 to infinity, and the
    * tetrahedron's symmetry axis lies parallel to the x-axis.
    * To convince yourself that the tetrahedron's axis of
    * symmetry does indeed pass through that point, note
    * that a half turn around the axis of symmetry factors
    * as a reflection in the plane |z| = 1 followed by
    * a reflection in the vertical plane sitting over x-axis.
    */

    /*
    * Order the vertices so that the tetrahedron is positively
    * oriented, and the selected edge is between vertices
    * v[0] and v[1].
    */
    v[0] = one_vertex_at_edge[best_edge];
    v[1] = other_vertex_at_edge[best_edge];
    v[2] = remaining_face[v[1]][v[0]];
    v[3] = remaining_face[v[0]][v[1]];

    /*
    * Set the coordinates of the corners.
    */

    z = (*initial_tet)->shape[filled]->cwl[ultimate][edge3[best_edge]].rect;
    sqrt_z = complex_sqrt(z);

    (*initial_tet)->corner[v[0]] = Infinity;
    (*initial_tet)->corner[v[1]] = Zero;
    (*initial_tet)->corner[v[2]] = complex_div(One, sqrt_z);
    (*initial_tet)->corner[v[3]] = sqrt_z;
}
}

```

```

}

void compute_fourth_corner(
    Complex      corner[4],
    VertexIndex  missing_corner,
    Orientation   orientation,
    ComplexWithLog cwl[3])
{
    int          i;
    VertexIndex  v[4];
    Complex      z[4],
                cross_ratio,
                diff20,
                diff21,
                numerator,
                denominator;

    /*
     * Given the locations on the sphere at infinity in
     * the upper half space model of three of a Tetrahedron's
     * four ideal vertices, compute_fourth_corner() computes
     * the location of the remaining corner.
     *
     * corner[4]      is the array which contains the three known
     *                corners, and into which the fourth will be
     *                written.
     *
     * missing_corner is the index of the unknown corner.
     *
     * orientation    is the Orientation with which the Tetrahedron
     *                is currently being viewed.
     *
     * cwl[3]         describes the shape of the Tetrahedron.
     */

    /*
     * Set up an indexing scheme v[] for the vertices.
     *
     * If some vertex (!= missing_corner) is positioned at infinity, let its
     * index be v0. Otherwise choose v0 arbitrarily. Then choose
     * v2 and v3 so that the Tetrahedron looks right_handed relative
     * to the v[].
     */

    v[3] = missing_corner;

    v[0] = ! missing_corner;
    for (i = 0; i < 4; i++)
        if (i != missing_corner && complex_infinite(corner[i]))
            v[0] = i;

    if (orientation == right_handed)
    {
        v[1] = remaining_face[v[3]][v[0]];
        v[2] = remaining_face[v[0]][v[3]];
    }
    else
    {
        v[1] = remaining_face[v[0]][v[3]];
        v[2] = remaining_face[v[3]][v[0]];
    }

    /*
     * Let z[i] be the location of v[i].
     * The z[i] are known for i < 3, unknown for i == 3.
     */

    for (i = 0; i < 3; i++)
        z[i] = corner[v[i]];

    /*
     * Note the cross_ratio at the edge connecting v0 to v1.
     */

```



```

cross_ratio = cwl[edge3_between_faces[v[0]][v[1]]].rect;
if (orientation == left_handed)
    cross_ratio = complex_conjugate(complex_div(One, cross_ratio));

/*
 * The cross ratio is defined as
 *
 *      (z3 - z1) (z2 - z0)
 * cross_ratio = -----
 *      (z2 - z1) (z3 - z0)
 *
 * Solve for z3.
 *
 *      z1*(z2 - z0) - cross_ratio*z0*(z2 - z1)
 * z3 = -----
 *      (z2 - z0) - cross_ratio*(z2 - z1)
 *
 * If z0 is infinite, this reduces to
 *
 *      z3 = z1 + cross_ratio * (z2 - z1)
 *
 * which makes sense geometrically.
 */

if (complex_infinite(z[0]) == TRUE)
    z[3] = complex_plus(
        z[1],
        complex_mult(
            cross_ratio,
            complex_minus(z[2], z[1])
        )
    );
else
{
    diff20 = complex_minus(z[2], z[0]);
    diff21 = complex_minus(z[2], z[1]);

    numerator = complex_minus(
        complex_mult(z[1], diff20),
        complex_mult(
            cross_ratio,
            complex_mult(z[0], diff21)
        )
    );
    denominator = complex_minus(
        diff20,
        complex_mult(cross_ratio, diff21)
    );

    z[3] = complex_div(numerator, denominator); /* will handle division by Zero
correctly */
}

corner[missing_corner] = z[3];
}

static void count_incident_generators(
    Triangulation *manifold)
{
    EdgeClass    *edge;
    Tetrahedron *tet;
    FaceIndex    face,
                facel;

    /*
     * For each EdgeClass, initialize num_incident_generators to zero.
     * Initialize all the active_relation flags to TRUE while we're at it.
     */

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

```

```

        edge = edge->next)
    {
        edge->num_incident_generators    = 0;
        edge->active_relation             = TRUE;
    }

    /*
     * For each face of a Tetrahedron dual to an outbound_generator,
     * increment the num_incident_generators count of the three
     * adjacent EdgeClasses. Ignore inbound_generators, to avoid
     * counting each generator twice.
     */

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

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

            if (tet->generator_status[face] == outbound_generator)

                for (face1 = 0; face1 < 4; face1++)

                    if (face1 != face)

                        tet->edge_class[edge_between_faces[face][face1]]->
num_incident_generators++;
}

static void eliminate_trivial_generators(
    Triangulation *manifold)
{
    Boolean    progress;
    EdgeClass  *edge;

    do
    {
        progress = FALSE;

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

            if (edge->num_incident_generators == 1)
            {
                kill_the_incident_generator(manifold, edge);
                progress = TRUE;
            }

    } while (progress == TRUE);
}

static void kill_the_incident_generator(
    Triangulation *manifold,
    EdgeClass     *edge)
{
    PositionedTet ptet,
                 ptet0;
    int          dead_index;
    Tetrahedron  *tet,
                 *nbr_tet;
    Permutation   gluing;
    FaceIndex     face,
                 nbr_face;

    /*
     * The EdgeClass edge is incident to a unique generator.
     * Find it.
     */

    set_left_edge(edge, &ptet0);

```

```

ptet = ptet0;

while (TRUE)
{
    /*
     * If we've found the active generator,
     * break out of the while loop. Otherwise . . .
     */

    if (ptet.tet->generator_status[ptet.near_face] != not_a_generator)
        break;

    /*
     * . . . move on to the next Tetrahedron incident to the EdgeClass.
     */

    veer_left(&ptet);

    /*
     * If we've come all the way around the EdgeClass without
     * finding a generator, something has gone terribly wrong.
     */

    if (same_positioned_tet(&ptet, &ptet0))
        uFatalError("kill_the_incident_generator", "choose_generators");
}

/*
 * Note the index of the about to be killed generator . . .
 */

dead_index = ptet.tet->generator_index[ptet.near_face];

/*
 * . . . then kill it.
 */

nbr_tet      = ptet.tet->neighbor[ptet.near_face];
gluing       = ptet.tet->gluing[ptet.near_face];
nbr_face     = EVALUATE(gluing, ptet.near_face);

ptet.tet->generator_status[ptet.near_face] = not_a_generator;
nbr_tet->generator_status[nbr_face]       = not_a_generator;

ptet.tet->generator_index[ptet.near_face]  = -1;    /* garbage value */
nbr_tet->generator_index[nbr_face]         = -1;

/*
 * The EdgeClass no longer represents an active relation.
 */

edge->active_relation = FALSE;

/*
 * Decrement the num_incident_generators count at each of
 * the incident EdgeClasses.
 */

ptet.tet->edge_class[edge_between_faces[ptet.near_face][ptet.left_face] ]->
num_incident_generators--;
ptet.tet->edge_class[edge_between_faces[ptet.near_face][ptet.right_face] ]->
num_incident_generators--;
ptet.tet->edge_class[edge_between_faces[ptet.near_face][ptet.bottom_face]]->
num_incident_generators--;

/*
 * Decrement *number_of_generators.
 */

manifold->num_generators--;

/*

```

```

    * If dead_index was not the highest numbered generator, then removing
    * it will have left a gap in the numbering scheme. Renummer the highest
    * numbered generator to keep the numbering contiguous.
    */

    if (dead_index != manifold->num_generators)
    {
        for (tet = manifold->tet_list_begin.next;
             tet != &manifold->tet_list_end;
             tet = tet->next)

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

                if (tet->generator_index[face] == manifold->num_generators)
                {
                    if (tet->generator_status[face] == not_a_generator)
                        uFatalError("kill_the_incident_generator", "choose_generators");

                    nbr_tet      = tet->neighbor[face];
                    gluing       = tet->gluing[face];
                    nbr_face     = EVALUATE(gluing, face);

                    tet      ->generator_index[face]      = dead_index;
                    nbr_tet->generator_index[nbr_face]    = dead_index;

                    /*
                     * Rather than worrying about breaking out of a
                     * double loop, let's just return from here.
                     */
                    return;
                }

        /*
         * The program should return from within the above double loop.
         */

        uFatalError("kill_the_incident_generator", "choose_generators");
    }

    else /* dead_index == manifold->num_generators, so nothing else to do */
        return;
}

static void merge_equivalent_generators(
    Triangulation *manifold)
{
    EdgeClass *edge;

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

        if (edge->num_incident_generators == 2)
            merge_incident_generators(manifold, edge);
}

static void merge_incident_generators(
    Triangulation *manifold,
    EdgeClass *edge)
{
    PositionedTet ptet,
    ptet0;
    Tetrahedron *tetA,
    *tetB,
    *tet;
    FaceIndex faceA,
    faceB,
    face;
    int indexA,
    indexB;
    Boolean generator_A_has_been_found,
    directions_agree;

```

```

/*
 * Find the two incident generators by letting ptet
 * rotate around the EdgeClass. The first time we
 * encounter a nontrivial generator, call it
 * faceA of tetA; the second time, faceB of tetB.
 */

set_left_edge(edge, &ptet0);

ptet = ptet0;

generator_A_has_been_found = FALSE;

while (TRUE)
{
    /*
     * If we've found an active generator, record it.
     * If this is the second one we've found, break out of the loop.
     */

    if (ptet.tet->generator_status[ptet.near_face] != not_a_generator)
    {
        if (generator_A_has_been_found == FALSE)
        {
            tetA = ptet.tet;
            faceA = ptet.near_face;
            generator_A_has_been_found = TRUE;
        }
        else
        {
            tetB = ptet.tet;
            faceB = ptet.near_face;
            break;
        }
    }

    /*
     * Move on to the next Tetrahedron incident to the EdgeClass.
     */

    veer_left(&ptet);

    /*
     * If we've come all the way around the EdgeClass without
     * finding both generators, something has gone terribly wrong.
     */

    if (same_positioned_tet(&ptet, &ptet0))
        uFatalError("kill_the_incident_generator", "choose_generators");
}

/*
 * If the two generators are the same, then either their product is
 * aA (in which case there is no further work to be done) or aa (in
 * which case they cannot be merged). Either way, we simply return.
 * [JRW 95/1/19. Actually, I don't think the first case (aA) is
 * likely to occur. The n-gons which are subdivided into triangles
 * have no interior vertices, so under normal circumstances the
 * generators we're merging should be distinct. If they're not,
 * it means we have a "face" which is topologically a cylinder,
 * or something weird like that. At any rate, we should return
 * without taking any action.]
 */

indexA = tetA->generator_index[faceA];
indexB = tetB->generator_index[faceB];
if (indexA == indexB)
    return;

/*
 * Do the directions of the generators agree or disagree?
 * Note that the generator will point in the same direction

```

```

    * relative to the boundary of the fundamental domain iff
    * one is an outbound_generator and the other is an inbound_generator
    * relative to the preceding cyclic traversal around the EdgeClass.
    */

directions_agree = (tetA->generator_status[faceA] != tetB->generator_status[faceB]);

/*
 * If directions_agree is FALSE, reverse the direction of generator A.
 * Then let generator A inherit the index of generator B.
 *
 * Let the highest numbered generator inherit the former index
 * of generator A, and decrement the number_of_generators count.
 *
 * Even in the special cases where indexA or indexB is the highest
 * index, generators A and B get merged, and the previously highest
 * index will no longer occur. This keeps the indices contiguous.
 */

manifold->num_generators--;

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

    for (face = 0; face < 4; face++)
    {
        if (tet->generator_index[face] == indexA)
        {
            if (directions_agree == FALSE)
            {
                if (tet->generator_status[face] == outbound_generator)
                    tet->generator_status[face] = inbound_generator;
                else if (tet->generator_status[face] == inbound_generator)
                    tet->generator_status[face] = outbound_generator;
                else
                    uFatalError("merge_incident_generators", "choose_generators");
            }
            tet->generator_index[face] = indexB;
        }

        if (tet->generator_index[face] == manifold->num_generators)
            tet->generator_index[face] = indexA;
    }

/*
 * The EdgeClass no longer represents an active relation.
 */

edge->active_relation = FALSE;
}

```