

```

/*
 * canonize_part_2.c
 *
 * This file provides the function
 *
 *     void canonical_retriangulation(Triangulation *manifold);
 *
 * which accepts an arbitrary subdivision of a canonical cell decomposition
 * into Tetrahedra, and replaces it with a canonical retriangulation
 * (defined below).  If the canonical cell decomposition is itself a
 * triangulation (as is typically the case) then the canonical retriangulation
 * is just the canonical cell decomposition itself.  If the canonical cell
 * decomposition is not a triangulation, the canonical retriangulation will
 * introduce a finite vertex at the center of each 3-cell.
 * canonical_retriangulation() is intended to follow a call to proto_canonize()
 * (cf. the function canonize() in canonize.c); it assumes the tet->tilt[]
 * fields are correct.
 *
 * If *manifold has a hyperbolic structure and/or VertexCrossSections,
 * they are discarded.
 *
 * Definition of the canonical retriangulation.
 *
 * The canonical cell decomposition of a cusped hyperbolic 3-manifold
 * is defined in
 *
 *     J. Weeks, Convex hulls and isometries of cusped hyperbolic
 *       3-manifolds, Topology Appl. 52 (1993) 127-149.
 *
 * and
 *
 *     M. Sakuma and J. Weeks, The generalized tilt formula,
 *       Geometriae Dedicata 55 (1995) 115-123.
 *
 * If the canonical cell decomposition is a triangulation (as is
 * typically the case), then the canonical retriangulation is just
 * the canonical cell decomposition itself.  Otherwise, the canonical
 * retriangulation is defined by the following two step procedure.
 *
 * Step #1.    Subdivide each 3-cell by coning its boundary to a point
 *              in its interior.
 *
 * Step #2.    Each 2-cell in the canonical cell decomposition (just
 *              the original 2-cells, not the 2-cells introduced in Step #1)
 *              is the boundary between two 3-cells created at Step #1.
 *              The union of the two 3-cells is the suspension of the
 *              2-cell between two of the vertices introduced in Step #1.
 *              Triangulate this suspension in the "obvious" symmetrical way,
 *              as n tetrahedra surrounding a common edge, where n is the
 *              number of sides of the 2-cell, and the common edge runs
 *              from one finite vertex to the other.
 *
 * This two-step procedure defines a canonical retriangulation of
 * the canonical cell decomposition.  Note that it's not a subdivision.
 *
 * Computing the canonical retriangulation.
 *
 * The following is a mathematical explanation of the algorithm.
 * The details of the implementation are documented in the code itself.
 *
 * We begin with an arbitrary subdivision of the canonical cell
 * decomposition into Tetrahedra.
 *
 * Definition.  A 2-cell in the Triangulation is called "opaque" if
 * it lies in the 2-skeleton of the canonical cell decomposition,
 * and "transparent" if lies in the interior of a 3-cell of the
 * canonical decomposition.
 *
 * Step #1.
 * To cone a 3-cell to a point in its interior, first do a one-to-four
 * move to cone a single Tetrahedron to a point in its interior.
 * If that was the only Tetrahedron in the 3-cell, we're done.

```

```

* Otherwise, perform a two-to-three move across a transparent face of
* the coned tetrahedron. This yields a coned hexahedron. Continue
* in this fashion until all the Tetrahedra in the 3-cell have been
* absorbed into the coned polyhedron. This coned polyhedron may have
* some pair of faces on its boundary identified to yield the original
* 3-cell. Where this occurs, call cancel_tetrahedra() to simplify the
* coned polyhedron. (This operation is documented more thoroughly
* in the code itself.)
*
* Step #2.
* Do a two-to-three move across each opaque face, then cancel
* all pairs of Tetrahedra surrounding EdgeClasses of order 2.
* You may take it as an exercise for the reader to prove that this
* has the desired effect, or you may read the explanation provided
* in the function step_two() below.
*
*
* Programming note: I have coded this algorithm for simplicity
* rather than speed. But even though the code is "less efficient"
* because it does, e.g., some unnecessary rescanning of lists, I don't
* think this will make a measurable difference in practice, and
* in any case I think it's a small price to pay to keep the logic
* of the code clean and simple.
*/

#include "kernel.h"
#include "canonize.h"

static void      remove_vertex_cross_sections(Triangulation *manifold);
static void      attach_canonize_info(Triangulation *manifold);
static void      free_canonize_info(Triangulation *manifold);
static void      label_opaque_faces(Triangulation *manifold);
static void      step_one(Triangulation *manifold);
static void      initialize_tet_status(Triangulation *manifold);
static Boolean    cone_3_cell(Triangulation *manifold, int *num_finite_vertices);
static Boolean    find_unconed_tet(Triangulation *manifold, Tetrahedron **tet0);
static Boolean    expand_coned_region(Triangulation *manifold);
static Boolean    attempt_cancellation(Triangulation *manifold);
static Boolean    verify_coned_region(Triangulation *manifold);
static void      step_two(Triangulation *manifold);
static Boolean    eliminate_opaque_face(Triangulation *manifold);

void canonical_retriangulation(
    Triangulation *manifold)
{
    /*
     * Remove the hyperbolic structures and VertexCrossSections, if any.
     */

    remove_hyperbolic_structures(manifold);
    remove_vertex_cross_sections(manifold);

    /*
     * If the canonical cell decomposition is a triangulation, we're done.
     * (Note: Comment out this line if you want to invoke the more
     * elaborate canonical retriangulation scheme for all manifolds, not
     * just those whose canonical cell decompositions contain cells other
     * than tetrahedra.)
     */

    if (is_canonical_triangulation(manifold) == TRUE)
        return;

    /*
     * Add a CanonizeInfo field to each Tetrahedron to hold local variables.
     * These variables must be accessible to the low-level retriangulation
     * routines in simplify_triangulation.c, which is why we use the
     * special purpose pointer canonize_info in the Tetrahedron data
     * structure, rather than using the general purpose "extra" pointer.
     */

    attach_canonize_info(manifold);

```

```

/*
 * Note which 2-cells are opaque and which are transparent.
 */

label_opaque_faces(manifold);

/*
 * Carry out the two step retriangulation algorithm described above.
 */

step_one(manifold);
step_two(manifold);

/*
 * Free the CanonizeInfo fields.
 */

free_canonize_info(manifold);

/*
 * We can't possibly compute the Chern-Simons invariant
 * for a manifold with finite vertices, so we set
 * CS_fudge_is_known to FALSE. However, we can leave
 * CS_value_is_known as TRUE if it is already TRUE, since
 * the manifold is still the same.
 */

manifold->CS_fudge_is_known = FALSE;
}

static void remove_vertex_cross_sections(
    Triangulation *manifold)
{
    Tetrahedron *tet;

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

        if (tet->cross_section != NULL)
        {
            my_free(tet->cross_section);
            tet->cross_section = NULL;
        }
}

static void attach_canonize_info(
    Triangulation *manifold)
{
    Tetrahedron *tet;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        /*
         * Just to be safe . . .
         */
        if (tet->canonize_info != NULL)
            uFatalError("attach_canonize_info", "canonize_part_2");

        /*
         * Attach the CanonizeInfo.
         */
        tet->canonize_info = NEW_STRUCT(CanonizeInfo);
    }
}

static void free_canonize_info(
    Triangulation *manifold)

```

```

{
    Tetrahedron *tet;

    for (tet = manifold->tet_list_begin.next;
         tet != &manifold->tet_list_end;
         tet = tet->next)
    {
        /*
         * Free the CanonizeInfo structure.
         */
        my_free(tet->canonize_info);

        /*
         * Set the canonize_info pointer to NULL just to be safe.
         */
        tet->canonize_info = NULL;
    }
}

static void label_opaque_faces(
    Triangulation *manifold)
{
    Tetrahedron *tet,
                *nbr_tet;
    FaceIndex f,
              nbr_f;
    double sum_of_tilts;

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

        for (f = 0; f < 4; f++)
        {
            nbr_tet = tet->neighbor[f];
            nbr_f = EVALUATE(tet->gluing[f], f);

            sum_of_tilts = tet->tilt[f] + nbr_tet->tilt[nbr_f];

            tet->canonize_info->face_status[f] =
                sum_of_tilts < - CONCAVITY_EPSILON ?
                opaque_face :
                transparent_face;
        }
}

static void step_one(
    Triangulation *manifold)
{
    int num_finite_vertices;

    /*
     * Initialize each part_of_coned_cell flag to FALSE.
     */

    initialize_tet_status(manifold);

    /*
     * Keep track of the number of finite vertices that
     * have been introduced, so they can be given consecutive
     * negative integers as indices.
     */

    num_finite_vertices = 0;

    /*
     * Cone each 3-cell of the canonical cell decomposition to a point
     * in its interior. Each call to cone_3_cell() cones a single 3-cell,
     * so we must keep calling cone_3_cell() until it returns FALSE.
     */

    while (cone_3_cell(manifold, &num_finite_vertices) == TRUE)

```

```

    ;
}

static void initialize_tet_status(
    Triangulation *manifold)
{
    Tetrahedron *tet;

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

        tet->canonize_info->part_of_coned_cell = FALSE;
}

static Boolean cone_3_cell(
    Triangulation *manifold,
    int *num_finite_vertices)
{
    Tetrahedron *tet0;

    /*
     * Is there a Tetrahedron which is not yet part of a coned cell?
     * If so, proceed.
     * If not, return FALSE.
     */

    if (find_unconed_tet(manifold, &tet0) == FALSE)
        return FALSE;

    /*
     * Cone tet0 to its center. This will introduce a finite vertex.
     * Each of the four new Tetrahedra will have part_of_coned_cell
     * set to TRUE. The face_status for the "exterior" faces will
     * be preserved, and the face_status for the "interior" faces
     * will be set to inside_cone_face.
     */

    one_to_four(tet0, &manifold->num_tetrahedra, -++*num_finite_vertices);

    /*
     * Expand the coned region. Whenever a Tetrahedron with
     * part_of_coned_cell == TRUE borders a Tetrahedron with
     * part_of_coned_cell == FALSE across a transparent face,
     * do a two-to-three move across the face to include the
     * (space occupied by the) latter Tetrahedron in the (growing)
     * coned region. Keep doing this as long as progress is
     * being made. The two-to-three move will set
     * part_of_coned_cell = TRUE for the three new Tetrahedra
     * it creates. It will preserve the face_status of the
     * "exterior" faces of the new Tetrahedra, and set the face_status
     * of the "interior" faces to inside_cone_face.
     *
     * (Yes, I know this isn't the optimally efficient way to do this,
     * but I don't think that's important. Please see the programming
     * note at the end of the documentation at the top of this file.)
     */

    while (expand_coned_region(manifold) == TRUE)
        ;

    /*
     * If the initial, arbitrary subdivision of the 3-cell contained
     * edges in its interior (as would be the case with an octahedron,
     * for example), then the coned 3-cell we just produced will have
     * some identifications on its boundary (for example, in the case
     * of the octahedron we'd have a coned decahedron with two adjacent
     * boundary faces identified to yield the octahedron). Assuming at
     * least one pair of identified faces are adjacent, we can call
     * cancel_tetrahedra() to simplify the structure of the coned region
     * from a coned n-hedron with k pairs of faces identified to a coned
     * (n-2)-hedron with (k-1) pairs of faces identified. As long as some

```

```

* pair of identified faces are adjacent (and identified in the obvious
* way) we can keep repeating the simplification to arrive at a coned
* (n-2k)-hedron with no faces identified, which is exactly what
* we want in Step #1 of this algorithm.
*
* It's theoretically possible to have faces of a coned polyhedron
* identified with no pair of identified faces adjacent to each other.
* For example, consider the complement of the house-with-two-rooms
* in the 3-sphere. Fortunately such examples are so rare and so
* complicated that I doubt any will ever show up as triangulations
* of 3-cells in canonical cell decompositions. If one did show up,
* verify_coned_region() would detect the condition and call uFatalError().
*
* Proposition. Identifying two adjacent faces (in the "obvious" way)
* creates an EdgeClass of order 2, and this is the only way
* EdgeClasses of order 2 arise.
*
* Proof. It's obvious that such an identification creates an
* EdgeClass of order 2. We must prove that this is the only way
* they may arise. Consider separately EdgeClasses from the
* original Triangulation (i.e. from the original arbitrary subdivision
* of the canonical cell decomposition), which connect one ideal
* vertex to another, and EdgeClasses which are added during the
* coning process, which connect an ideal vertex to a finite vertex.
*
* Case 1. Original EdgeClasses, which connect one ideal vertex
* to another. There are no EdgeClasses of order 2 in the initial
* Triangulation, because it is geometric. In a coned polyhedron
* (ignoring boundary identifications for the moment) each
* EdgeClass connecting one ideal vertex to another is incident
* to precisely two of the coned polyhedron's Tetrahedra. If the
* EdgeClass lies on the boundary of the 3-cell (of the canonical
* cell decomposition) then its true order will be greater than two.
* If it lies in the interior of the 3-cell, then the only way for
* the order to be exactly two is to have the two adjacent faces
* identified.
*
* Case 2. EdgeClasses added during the coning process, which
* connect an ideal vertex to a finite vertex. We assume such
* an EdgeClass has order 2, and will deduce a contradiction.
* Consider the coned polyhedron, ignoring boundary identifications
* for the moment. Consider the two faces on the boundary of
* incident to the ideal endpoint of the given EdgeClass. These
* faces are part of the original (geometric!) subdivision of the
* canonical cell decomposition. They have all three ideal vertices
* in common, therefore they must coincide. But this implies that
* the boundary component at the ideal vertex of the given EdgeClass
* is topologically a sphere.
*
* Q.E.D.
*/

```

```

while (attempt_cancellation(manifold) == TRUE)
    ;

```

```

/*
* As explained above, once we've cancelled all Tetrahedra incident
* to EdgeClasses of order 2, we will almost surely have the required
* coning of the 3-cell's boundary to a point in its interior, unless
* we encounter a situation like the complement of the
* house-with-two-rooms. verify_coned_region() checks that we
* do in fact have a coning of the original 3-cell; that is,
* no Tetrahedron with part_of_coned_cell == TRUE has a face with
* face_status transparent_face.
*/

```

```

if (verify_coned_region(manifold) == FALSE)
    uFatalError("cone_3_cell", "canonize_part_2");

```

```

return TRUE;

```

```

}

```

```

static Boolean find_unconed_tet(

```

```

    Triangulation    *manifold,
    Tetrahedron      **tet0)
{
    Tetrahedron      *tet;

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

        if (tet->canonize_info->part_of_coned_cell == FALSE)
        {
            *tet0 = tet;
            return TRUE;
        }

    return FALSE;
}

static Boolean expand_coned_region(
    Triangulation    *manifold)
{
    Tetrahedron      *tet;
    FaceIndex        f;

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

        if (tet->canonize_info->part_of_coned_cell == TRUE)

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

                if (tet->canonize_info->face_status[f] == transparent_face)

                    if (tet->neighbor[f]->canonize_info->part_of_coned_cell == FALSE)
                    {
                        if (two_to_three(tet, f, &manifold->num_tetrahedra) == func_OK)

                            return TRUE;

                        else    /* this should never occur */

                            uFatalError("expand_coned_region", "canonize_part_2");
                    }

    return FALSE;
}

static Boolean attempt_cancellation(
    Triangulation    *manifold)
{
    EdgeClass        *edge,
                    *where_to_resume;

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

        if (edge->order == 2)
        {
            if (cancel_tetrahedra(edge, &where_to_resume, &manifold->num_tetrahedra) ==
func_OK)

                return TRUE;

            else
                /*
                 * I don't think failure is possible, but if it is
                 * I want to know about it.
                 */
                uFatalError("attempt_cancellation", "canonize_part_2");
        }
}

```

```

    return FALSE;
}

static Boolean verify_coned_region(
    Triangulation *manifold)
{
    /*
     * Because one_to_four(), two_to_three() and cancel_tetrahedra()
     * all maintain the coned polyhedron as some sort of coned
     * polyhedron, all we need to check is that it has no remaining
     * identifications on its boundary. That is, we need to check
     * that the faces of each Tetrahedron with part_of_coned_cell == TRUE
     * have face_status opaque_face or inside_cone_face, but never
     * transparent_face.
     */

    Tetrahedron *tet;
    FaceIndex f;

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

        if (tet->canonize_info->part_of_coned_cell == TRUE)

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

                if (tet->canonize_info->face_status[f] == transparent_face)

                    return FALSE;

    return TRUE;
}

static void step_two(
    Triangulation *manifold)
{
    /*
     * Step #1 of the algorithm has already been carried out, so
     * we know that every 3-cell in the canonical cell decomposition
     * has been subdivided by coning the boundary to a point in
     * the interior. There are three types of EdgeClasses:
     *
     * (A) Those which lie in the 1-skeleton of the canonical
     *     cell decomposition. They have order at least 6.
     *
     * (B) Those which lie in the 2-skeleton of the canonical
     *     cell decomposition, but not in the 1-skeleton. They
     *     serve to (artificially) subdivide the faces of the 3-cells
     *     into triangles. Each has order precisely 4.
     *
     * (C) Those in the interior of the 3-cells. They connect ideal
     *     vertices to finite vertices, and have order at least 3.
     *
     * Step #2 of the algorithm consists of two substeps:
     *
     * Substep A. Do a two-to-three move across every opaque face.
     *
     * Substep B. Perform all possible cancellations of Tetrahedra
     *             surrounding EdgeClasses of order 2.
     *
     * After Substep A is complete, we know that
     *
     *     EdgeClasses of type A will have order at least 3.
     *     EdgeClasses of type B will have order precisely 2.
     *     EdgeClasses of type C will have order at least 6.
     *     The new EdgeClasses introduced in Substep A will have
     *         order precisely 3.
     *
     * Therefore Substep B will eliminate precisely the EdgeClasses
     * of type B. It's easy to see that removing these EdgeClasses

```



```
    * creates the Triangulation specified in Step #2 of the definition
    * of the canonical retriangulation at the top of this file.
    */

    while (eliminate_opaque_face(manifold) == TRUE)
        ;

    while (attempt_cancellation(manifold) == TRUE)
        ;
}

static Boolean eliminate_opaque_face(
    Triangulation *manifold)
{
    Tetrahedron *tet;
    FaceIndex f;

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

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

            if (tet->canonize_info->face_status[f] == opaque_face)
            {
                if (two_to_three(tet, f, &manifold->num_tetrahedra) == func_OK)

                    return TRUE;

                else /* this should never occur */

                    uFatalError("expand_coned_region", "canonize_part_2");
            }

    return FALSE;
}
```