

```

/*
 * shingling.c
 *
 * This feature is being written for Bill Floyd and Jim Cannon.
 * Here is Bill's statement of the problem.
 *
 * Let D be a Dirichlet domain for the group action, and let S be
 * the generating set of non-identity group elements such that g(D)
 * intersects D. Given a nonnegative integer n, let B(n) be the
 * union of domains g(D) with word norm |g|_S <= n.
 *
 * Given a (probably small) nonnegative integer n, for output I
 * would like the circles at infinity corresponding to faces of
 * B(m) for m <= n. I would like to be able to do this for the
 * hyperbolic dodecahedral reflection group, and for other examples
 * (presumably found from SnapPea) where the dihedral angles are
 * submultiples of pi.
 */

#include "kernel.h"

#define SAME_MATRIX_EPSILON 1e-4
#define TOO_FAR_EPSILON 1e-3

/*
 * matrix_on_tree() searches all nodes whose key values are within
 * TREE_EPSILON of the given key value. If TREE_EPSILON is too large,
 * the algorithm will waste time sifting through large numbers of
 * irrelevant matrices, but if it's too small you might end up adding
 * the same matrix over and over.
 */
#define TREE_EPSILON 1e-5

/*
 * Isometries are sometimes stored locally on NULL-terminated
 * singly linked lists...
 */
typedef struct IsometryListNode
{
    O3lMatrix m;
    struct IsometryListNode *next;
} IsometryListNode;

/*
 * ...and sometimes on binary trees.
 */
typedef struct IsometryTreeNode
{
    O3lMatrix m; /* the isometry */
    struct IsometryTreeNode *left, /* the tree structure */
    *right;
    double key; /* the sort key value */
    struct IsometryTreeNode *next_on_stack, /* for tree traversals */
    *next_on_stack1;
} IsometryTreeNode;

/*
 * Shingles are stored locally as NULL-terminated
 * singly linked lists of ShingleNodes.
 */
typedef struct ShingleNode
{
    Shingle shingle;
    struct ShingleNode *next;
} ShingleNode;

static IsometryListNode *make_list_of_face_pairings(WEPolyhedron *polyhedron);
static IsometryListNode *make_list_of_other_isometries(WEPolyhedron *polyhedron,
    IsometryListNode *s0);
static void add_neighbors_to_s1(O3lMatrix m, WEPolyhedron *polyhedron,
    IsometryListNode *s0, IsometryListNode *s1_begin, IsometryListNode ***s1_end);

```

```

static Boolean      intersection_is_nontrivial(WEPolyhedron *polyhedron, O3lMatrix m);
static Boolean      vertex_in_polyhedron(O3lVector v, WEPolyhedron *polyhedron);
static void         add_isometry_list_node(IsometryListNode **isometry_list, O3lMatrix
m);
static void         add_isometry_tree_node(IsometryTreeNode **isometry_tree, O3lMatrix
m);
static Boolean      matrix_on_list(O3lMatrix m, IsometryListNode *isometry_list);
static Boolean      matrix_on_tree(O3lMatrix m, IsometryTreeNode *isometry_tree);
static double       key_value(O3lMatrix m);
static void         add_shingle_node(ShingleNode **shingle_list, O3lMatrix m0,
O3lMatrix m1, int index);
static void         free_isometry_node_list(IsometryListNode **isometry_list);
static void         free_isometry_node_tree(IsometryTreeNode **isometry_tree);
static Shingling    *convert_shingle_list_to_shingling(ShingleNode **shingle_list);

```

```

Shingling *make_shingling(
    WEPolyhedron *polyhedron,
    int num_layers)
{
    /*
     * Consider the tiling of  $H^3$  by translates  $g(D)$  of the Dirichlet
     * domain  $D$ . Decompose the tiling into layers, defined inductively
     * as follows.  $L[-1]$  is the empty set.  $L[0]$  is the set containing
     * only the identity element. For  $n > 0$ ,  $L[n]$  is the set obtained
     * by multiplying each element of  $L[n-1]$  by each element of  $S$ 
     * (defined above) and keeping only those elements which aren't
     * already in  $L[n-1]$  or  $L[n-2]$ .
     *
     * Abuse terminology slightly and think of  $L[n]$  sometimes as
     * a set of group elements  $g$  and sometimes as the union of the
     * corresponding translates  $g(D)$  of the Dirichlet domain.
     *
     * According to Bill Floyd's statement of the problem, the shingles
     * correspond to the faces of the boundary between each  $L[n-1]$  and
     * the next  $L[n]$ . It's easy to compute the shingles as we
     * compute the set  $L[n]$ : we get a shingle whenever an element
     * of  $L[n-1]$  shares a face (not just an edge or a vertex) with
     * an element of  $L[n]$ . This occurs when the element  $g$  of  $S$  which
     * takes an element of  $L[n-1]$  to an element of  $L[n]$  is a face
     * pairing isometry of the Dirichlet domain. So divide  $S$  into
     * two subsets
     *
     *  $S_0$  = elements  $g$  of  $S$  for which  $g(D)$  intersects  $D$  in a face
     *  $S_1$  = elements  $g$  of  $S$  for which  $g(D)$  intersects  $D$  in an edge or vertex
     */

    IsometryListNode *s0,
                    *s1,
                    *g;
    IsometryTreeNode *prev_layer,
                    *this_layer,
                    *next_layer,
                    *subtree_stack,
                    *subtree_root;
    ShingleNode      *shingle_list;
    O3lMatrix        new_matrix;
    Shingling        *shingling;
    int              i;

    /*
     * Shinglings of ideal polyhedra are infinite.
     * Don't try to compute them.
     *
     * Just to be robust, watch out for NULL polyhedra too.
     */
    if (polyhedron == NULL
        || polyhedron->outradius == INFINITE_RADIUS)
    {
        shingling = NEW_STRUCT(Shingling);
        shingling->num_shingles = 0;
        shingling->shingles = NULL;

        return shingling;
    }
}

```

```

}

/*
 * Set up S0 and S1 (defined above) as linked lists.
 */
s0 = make_list_of_face_pairings(polyhedron);
s1 = make_list_of_other_isometries(polyhedron, s0);

/*
 * Initialize prev_layer to L[-1] = {} and this_layer to L[0] = {id}.
 * The next_layer is initially empty. All three layers are stored
 * as binary trees.
 */
prev_layer = NULL;
this_layer = NULL;
next_layer = NULL;
add_isometry_tree_node(&this_layer, O31_identity);

/*
 * As we go along we'll accumulate Shingles onto a linked list.
 * At the end we'll copy them onto an array.
 */
shingle_list = NULL;

/*
 * The parameter num_layers specifies how many layers to compute.
 * Compute the shingles corresponding to the faces of L[n] for
 * n <= num_layers.
 */
for (i = 0; i <= num_layers; i++)
{
    /*
     * Multiply each tile in this_layer by each element of S0.
     *
     * Note: We read matrix products right-to-left, so
     * "premultiplication by an isometry" is the same as
     * "right multiplication by a matrix".
     */

    /*
     * To avoid possible stack-heap collisions, we want to traverse
     * the tree without resorting to recursive function calls.
     * Put this_layer's root (if any) onto the subtree_stack.
     */
    subtree_stack = this_layer;
    if (subtree_stack != NULL)
        subtree_stack->next_on_stack = NULL;

    /*
     * Pull a subtree off the subtree_stack, put its children
     * back onto the stack, and examine the matrix at the root.
     * Keep going until the subtree_stack is empty.
     */
    while (subtree_stack != NULL)
    {
        /*
         * Pull a subtree off the stack.
         */
        subtree_root = subtree_stack;
        subtree_stack = subtree_stack->next_on_stack;
        subtree_root->next_on_stack = NULL;

        /*
         * Put its left and right children back onto the subtree_stack.
         */
        if (subtree_root->left != NULL)
        {
            subtree_root->left->next_on_stack = subtree_stack;
            subtree_stack = subtree_root->left;
        }
        if (subtree_root->right != NULL)
        {
            subtree_root->right->next_on_stack = subtree_stack;
            subtree_stack = subtree_root->right;
        }
    }
}

```

```

    }

    /*
     * Multiply subtree_root->m by each element g of S0.
     */
    for (g = s0; g != NULL; g = g->next)
    {
        /*
         * new_matrix = subtree_root->m * g
         */
        o3l_product(subtree_root->m, g->m, new_matrix);

        /*
         * If the new_matrix isn't an element of prev_layer
         * or this_layer, then the face between the old_tile
         * and the new tile defines a shingle.
         */
        if (matrix_on_tree(new_matrix, prev_layer) == FALSE
            && matrix_on_tree(new_matrix, this_layer) == FALSE)
        {
            add_shingle_node(&shingle_list, subtree_root->m, new_matrix, i);

            /*
             * Add the new_matrix to next_layer iff
             *
             * (1) it's not already there, and
             * (2) this isn't the last time through the loop
             * (in that case we won't need next_layer).
             */
            if (i != num_layers
                && matrix_on_tree(new_matrix, next_layer) == FALSE)
                add_isometry_tree_node(&next_layer, new_matrix);
        }
    }

    /*
     * If this isn't the last time through the loop,
     * multiply subtree_root->m by each element of S1
     * so we can be sure to compute all of next_layer.
     */
    if (i != num_layers)
        for (g = s1; g != NULL; g = g->next)
        {
            o3l_product(subtree_root->m, g->m, new_matrix);

            if (matrix_on_tree(new_matrix, prev_layer) == FALSE
                && matrix_on_tree(new_matrix, this_layer) == FALSE
                && matrix_on_tree(new_matrix, next_layer) == FALSE)

                add_isometry_tree_node(&next_layer, new_matrix);
        }
    }

    /*
     * Get ready to move on to the next level.
     */
    free_isometry_node_tree(&prev_layer);
    prev_layer = this_layer;
    this_layer = next_layer;
    next_layer = NULL;
}

/*
 * Release the lists of isometries.
 */
free_isometry_node_list(&s0);
free_isometry_node_list(&s1);
free_isometry_node_tree(&prev_layer);
free_isometry_node_tree(&this_layer);
free_isometry_node_tree(&next_layer);

/*
 * Repackage the results as a Shingling.
 * (This frees the ShingleNodes.)

```

```

    */
    shingling = convert_shingle_list_to_shingling(&shingle_list);

    return shingling;
}

void free_shingling(
    Shingling *shingling)
{
    if (shingling != NULL)
    {
        if (shingling->shingles != NULL)
            my_free(shingling->shingles);
        my_free(shingling);
    }
}

void compute_center_and_radials(
    Shingle *shingle,
    O31Matrix position,
    double scale)
{
    O31Vector n,
              nn,
              u;
    double factor,
            radius;
    O31Vector e1 = {0.0, 1.0, 0.0, 0.0},
              e2 = {0.0, 0.0, 1.0, 0.0};

    /*
     * Let n be the current position of the shingle's normal vector.
     */
    o31_matrix_times_vector(position, shingle->normal, n);

    /*
     * Let nn be a unit vector parallel to the projection of n onto
     * the xyz hyperplane. (shingle->normal has squared length 1.0,
     * so the argument of the sqrt() is at least 1.0.)
     */
    factor = 1.0 / sqrt(n[1]*n[1] + n[2]*n[2] + n[3]*n[3]);
    nn[0] = 0.0;
    nn[1] = factor * n[1];
    nn[2] = factor * n[2];
    nn[3] = factor * n[3];

    /*
     * The face plane is defined by the equation
     *
     * < (1, x, y, z), (n[0], n[1], n[2], n[3]) > == 0
     * or
     * n[1]*x + n[2]*y + n[3]*z = n[0]
     */

    /*
     * The center of the desired circle at infinity lies in the
     * direction of the vector nn (but with 0-th coordinate 1.0).
     * Scale it so that it satisfies the above equation.
     */

    shingle->center[0] = 1.0;
    shingle->center[1] = nn[1] * factor * n[0];
    shingle->center[2] = nn[2] * factor * n[0];
    shingle->center[3] = nn[3] * factor * n[0];

    /*
     * For the remainder of this function we work in the xyz slice
     * of O(3,1). That is, all 0-th coordinates are 0.0, and are
     * not mentioned in the comments.
     */
}

```

```

    * To draw a circle we must extend nn to a basis for  $R^3$ .
    */

/*
 * Let u be an arbitrary unit vector not parallel to nn.
 */
o3l_copy_vector(u, (fabs(nn[1]) < fabs(nn[2])) ? e1 : e2);

/*
 * Let shingle->radialA = u x nn, normalized to have length one.
 */
shingle->radialA[0] = 0.0;
shingle->radialA[1] = u[2] * nn[3] - u[3] * nn[2];
shingle->radialA[2] = u[3] * nn[1] - u[1] * nn[3];
shingle->radialA[3] = u[1] * nn[2] - u[2] * nn[1];
o3l_constant_times_vector( 1.0 / sqrt(o3l_inner_product(shingle->radialA, shingle->
radialA)),
                        shingle->radialA,
                        shingle->radialA);

/*
 * Let shingle->radialB = shingle->radialA x nn.
 * Its length will automatically be one.
 */
shingle->radialB[0] = 0.0;
shingle->radialB[1] = shingle->radialA[2] * nn[3] - shingle->radialA[3] * nn[2];
shingle->radialB[2] = shingle->radialA[3] * nn[1] - shingle->radialA[1] * nn[3];
shingle->radialB[3] = shingle->radialA[1] * nn[2] - shingle->radialA[2] * nn[1];

/*
 * Rescale the radials so that shingle->center + shingle->radial
 * lies on the sphere at infinity. Recall (or deduce) from above
 * that the length of shingle->center's xyz component is
 * factor * n[0].
 */
radius = sqrt(1.0 - (factor * n[0]) * (factor * n[0]));
o3l_constant_times_vector(radius, shingle->radialA, shingle->radialA);
o3l_constant_times_vector(radius, shingle->radialB, shingle->radialB);

/*
 * Multiply our results by the pixel size of the window.
 */
o3l_constant_times_vector(scale, shingle->center, shingle->center);
o3l_constant_times_vector(scale, shingle->radialA, shingle->radialA);
o3l_constant_times_vector(scale, shingle->radialB, shingle->radialB);
}

static IsometryListNode *make_list_of_face_pairings(
    WEPolyhedron *polyhedron)
{
    IsometryListNode *s0;
    WEFace *face;

    s0 = NULL;

    for (face = polyhedron->face_list_begin.next;
         face != &polyhedron->face_list_end;
         face = face->next)

        add_isometry_list_node(&s0, *face->group_element);

    return s0;
}

static IsometryListNode *make_list_of_other_isometries(
    WEPolyhedron *polyhedron,
    IsometryListNode *s0)
{
    /*
     * The plan is to look at the neighbors of the elements of S0
     * (i.e. elements g such that g(D) and s(D) share a face, for
     * some s in S0), then at the neighbors of the neighbors, and so on.
     */

```

```

    * At each stage we keep only those elements g such that
    * g(D) intersects D in a vertex or edge (in other words we keep
    * elements of S which aren't already in S0). The connectedness
    * of S implies that we'll eventually get all of S1 this way.
    */

    IsometryListNode    *s1_begin,
                        **s1_end,
                        *s1_mark,
                        *g;

    /*
    * s1_begin points to the beginning of the NULL-terminated
    * singly linked list. s1_end points to the field containing
    * the terminal NULL (initially this is s1_begin itself, but
    * typically it's the last node's "next" field).
    */
    s1_begin    = NULL;
    s1_end      = &s1_begin;

    /*
    * Initialize s1 to contain all immediate neighbors of s0
    * which intersect D and aren't already in (s0 union {id}).
    */
    for (g = s0; g != NULL; g = g->next)
        add_neighbors_to_s1(g->m, polyhedron, s0, s1_begin, &s1_end);

    /*
    * Think of the linked list S1 as a queue. Go down the queue,
    * starting at the beginning, and compute the neighbors of
    * each element (i.e. those elements which share a face with
    * the given element). Those neighbors which aren't already
    * in S0 or S1 and which have a nonempty intersection with D
    * get appended to the end of the queue. The pointer s1_mark
    * points to the next element to be processed. When s1_mark
    * reaches the end, we will have computed the whole set S1,
    * and we'll be done.
    */
    for (s1_mark = s1_begin; s1_mark != NULL; s1_mark = s1_mark->next)
        add_neighbors_to_s1(s1_mark->m, polyhedron, s0, s1_begin, &s1_end);

    return s1_begin;
}

static void add_neighbors_to_s1(
    O3lMatrix          m,
    WEPolyhedron        *polyhedron,
    IsometryListNode    *s0,
    IsometryListNode    *s1_begin,
    IsometryListNode    ***s1_end)
{
    IsometryListNode    *h,
                        *new_node;
    O3lMatrix            mh;

    for (h = s0; h != NULL; h = h->next)
    {
        o3l_product(m, h->m, mh);

        if (o3l_equal(mh, O3l_identity, SAME_MATRIX_EPSILON) == FALSE
            && matrix_on_list(mh, s0) == FALSE
            && matrix_on_list(mh, s1_begin) == FALSE
            && intersection_is_nontrivial(polyhedron, mh) == TRUE)
        {
            new_node = NEW_STRUCT(IsometryListNode);
            o3l_copy(new_node->m, mh);
            new_node->next = NULL;
            **s1_end      = new_node;
            *s1_end       = &new_node->next;
        }
    }
}

```

```

static Boolean intersection_is_nontrivial(
    WEPolyhedron    *polyhedron,
    O3lMatrix       m)
{
    WEVertex        *vertex;
    O3lVector        gv;
    double           temp,
                    max_length_squared;

    /*
     * If the matrix m translates the origin a distance greater than
     * twice the polyhedron's outradius, then the corresponding
     * image g(D) cannot possibly intersection the Dirichlet domain D.
     */
    if (m[0][0] > cosh(2.0 * polyhedron->outradius) + TOO_FAR_EPSILON)
        return FALSE;

    /*
     * Does the image g(v) of some vertex v lie within the original
     * Dirichlet domain D?
     */

    /*
     * We may safely ignore those g(v) whose lengths squared place
     * them beyond the polyhedron's circumradius. Please see the
     * documentation in compute_vertex_distance() in Dirichlet_extras.c
     * for an explanation of what's going on here.
     */
    temp = cosh(polyhedron->outradius) + TOO_FAR_EPSILON;
    max_length_squared = -1.0 / (temp * temp);

    for (vertex = polyhedron->vertex_list_begin.next;
         vertex != &polyhedron->vertex_list_end;
         vertex = vertex->next)
    {
        /*
         * Compute g(v).
         */
        o3l_matrix_times_vector(m, vertex->x, gv);

        /*
         * Normalize gv to have gv[0] == 1.0. As well as allowing
         * an easy comparison against max_length_squared, this will
         * make vertex_in_polyhedron()'s usage of vertex_epsilon
         * consistent with the usage in Dirichlet_construction.c.
         */
        o3l_constant_times_vector(1.0/gv[0], gv, gv);

        /*
         * Does gv lie outside the polyhedron's circumsphere?
         */
        if (o3l_inner_product(gv, gv) > max_length_squared)
            continue;

        /*
         * Test whether gv lies (approximately) on the polyhedron.
         */
        if (vertex_in_polyhedron(gv, polyhedron) == TRUE)
            return TRUE;
    }

    return FALSE;
}

static Boolean vertex_in_polyhedron(
    O3lVector        v,
    WEPolyhedron    *polyhedron)
{
    WEFace           *face;
    O3lVector        normal;
    int               i;

```



```

    for (face = polyhedron->face_list_begin.next;
         face != &polyhedron->face_list_end;
         face = face->next)
    {
        /*
         * Compute the face's normal vector, and normalize it to
         * have length one.
         */
        for (i = 0; i < 4; i++)
            normal[i] = (*face->group_element)[i][0] - (i ? 0.0 : 1.0);
        o3l_constant_times_vector( 1.0 / sqrt(o3l_inner_product(normal, normal)),
                                   normal,
                                   normal);

        /*
         * If the vertex v clearly lies beyond the face, return FALSE.
         */
        if (o3l_inner_product(v, normal) > polyhedron->vertex_epsilon)
            return FALSE;
    }

    return TRUE;
}

static void add_isometry_list_node(
    IsometryListNode **isometry_list,
    O3lMatrix m)
{
    IsometryListNode *new_node;

    new_node = NEW_STRUCT(IsometryListNode);

    o3l_copy(new_node->m, m);

    new_node->next = *isometry_list;
    *isometry_list = new_node;
}

static void add_isometry_tree_node(
    IsometryTreeNode **isometry_tree,
    O3lMatrix m)
{
    IsometryTreeNode *new_node,
                     **location;

    new_node = NEW_STRUCT(IsometryTreeNode);
    o3l_copy(new_node->m, m);
    new_node->left = NULL;
    new_node->right = NULL;
    new_node->key = key_value(m);
    new_node->next_on_stack = NULL;

    location = isometry_tree;
    while (*location != NULL)
    {
        if (new_node->key <= (*location)->key)
            location = &(*location)->left;
        else
            location = &(*location)->right;
    }
    *location = new_node;
}

static Boolean matrix_on_list(
    O3lMatrix m,
    IsometryListNode *isometry_list)
{
    IsometryListNode *node;

    for (node = isometry_list; node != NULL; node = node->next)
        if (o3l_equal(m, node->m, SAME_MATRIX_EPSILON) == TRUE)

```

```

        return TRUE;

    return FALSE;
}

static Boolean matrix_on_tree(
    O31Matrix      m,
    IsometryTreeNode *isometry_tree)
{
    double          m_key,
                    delta;
    IsometryTreeNode *subtree_stack,
                    *subtree;
    Boolean          left_flag,
                    right_flag;

    /*
     * Compute a key value for the matrix m.
     */
    m_key = key_value(m);

    /*
     * Reliability is our first priority. Speed is second.
     * So if m_key is close to a node's key value, we want to search both
     * the left and right subtrees. Otherwise we search only one or the
     * other. We implement the recursion using our own stack, rather than
     * the system stack, to avoid the possibility of a stack/heap collision
     * during deep recursions.
     */

    /*
     * Initialize the stack to contain the whole tree.
     */
    subtree_stack = isometry_tree;
    if (isometry_tree != NULL)
        isometry_tree->next_on_stack1 = NULL;

    /*
     * Process the subtrees on the stack,
     * adding additional subtrees as needed.
     */
    while (subtree_stack != NULL)
    {
        /*
         * Pull a subtree off the stack.
         */
        subtree          = subtree_stack;
        subtree_stack    = subtree_stack->next_on_stack1;
        subtree->next_on_stack1 = NULL;

        /*
         * Compare the key values of the tile and the subtree's root.
         */
        delta = m_key - subtree->key;

        /*
         * Which side(s) should we search?
         */
        left_flag  = (delta < +TREE_EPSILON);
        right_flag = (delta > -TREE_EPSILON);

        /*
         * Put the subtrees we need to search onto the stack.
         */
        if (left_flag && subtree->left)
        {
            subtree->left->next_on_stack1 = subtree_stack;
            subtree_stack = subtree->left;
        }
        if (right_flag && subtree->right)
        {
            subtree->right->next_on_stack1 = subtree_stack;
            subtree_stack = subtree->right;
        }
    }
}

```

```

    }

    /*
     * Check this matrix if the key values match.
     * (We're leaving non-NULL values in some next_on_stack1 fields,
     * but that's OK.)
     */
    if (left_flag && right_flag)
        if (o3l_equal(m, subtree->m, SAME_MATRIX_EPSILON) == TRUE)
            return TRUE;
    }

    return FALSE;
}

static double key_value(
    O3lMatrix m)
{
    /*
     * Please see key_value() in length_spectrum.c
     * for a full explanation of what's going on here.
     */
    return( m[1][0] * 0.47865745183883625637
        + m[2][0] * 0.14087522034920476458
        + m[3][0] * 0.72230196622481940253);
}

static void add_shingle_node(
    ShingleNode **shingle_list,
    O3lMatrix m0,
    O3lMatrix m1,
    int index)
{
    /*
     * Let b = (1,0,0,0) be the Dirichlet domain's basepoint.
     * Let g0 and g1 be the isometries given by m0 and m1, respectively.
     * The caller has checked that g0(D) and g1(D) are adjacent
     * translates of the Dirichlet domain D. The points g0(b) and g1(b)
     * are the corresponding translates of the basepoint, and by
     * the definition of the Dirichlet domain the vector g1(b) - g0(b)
     * is orthogonal to g0(D) and g1(D)'s common face.
     */

    ShingleNode *new_node;
    int i;
    O3lVector zero_vector = {0.0, 0.0, 0.0, 0.0};

    /*
     * Allocate the new_node.
     */
    new_node = NEW_STRUCT(ShingleNode);

    /*
     * Compute the (spacelike) vector g1(b) - g0(b) and normalize
     * its length to 1.0.
     */
    for (i = 0; i < 4; i++)
        new_node->shingle.normal[i] = m1[i][0] - m0[i][0];
    o3l_constant_times_vector( 1.0 / sqrt(o3l_inner_product(new_node->shingle.normal,
new_node->shingle.normal)),
new_node->shingle.normal,
new_node->shingle.normal);

    /*
     * The UI will set the center and radii before displaying
     * the shingling.
     */
    o3l_copy_vector(new_node->shingle.center, zero_vector);
    o3l_copy_vector(new_node->shingle.radialA, zero_vector);
    o3l_copy_vector(new_node->shingle.radialB, zero_vector);

    /*

```

```

    * Set the index.
    */
    new_node->shingle.index = index;

    /*
    * Put the new_node on the shingle_list.
    */
    new_node->next = *shingle_list;
    *shingle_list = new_node;
}

static void free_isometry_node_list(
    IsometryListNode    **isometry_list)
{
    IsometryListNode    *dead_isometry_node;

    while (*isometry_list != NULL)
    {
        dead_isometry_node = *isometry_list;
        *isometry_list     = (*isometry_list)->next;
        my_free(dead_isometry_node);
    }
}

static void free_isometry_node_tree(
    IsometryTreeNode    **isometry_tree)
{
    IsometryTreeNode    *subtree_stack,
                        *subtree;

    /*
    * Implement the recursive freeing algorithm using our own stack
    * rather than the system stack, to avoid the possibility of a
    * stack/heap collision.
    */

    /*
    * Initialize the stack to contain the whole tree.
    */
    subtree_stack = *isometry_tree;
    if (subtree_stack != NULL)
        subtree_stack->next_on_stack1 = NULL;

    /*
    * Process the subtrees on the stack one at a time.
    */
    while (subtree_stack != NULL)
    {
        /*
        * Pull a subtree off the stack.
        */
        subtree          = subtree_stack;
        subtree_stack     = subtree_stack->next_on_stack1;
        subtree->next_on_stack1 = NULL;

        /*
        * If the subtree's root has nonempty left and/or right subtrees,
        * add them to the stack.
        */
        if (subtree->left != NULL)
        {
            subtree->left->next_on_stack1 = subtree_stack;
            subtree_stack = subtree->left;
        }
        if (subtree->right != NULL)
        {
            subtree->right->next_on_stack1 = subtree_stack;
            subtree_stack = subtree->right;
        }

        /*
        * Free the subtree's root node.
        */
    }
}

```

```

        */
        my_free(subtree);
    }

    /*
     * Clear the caller's reference to the defunct tree.
     */
    *isometry_tree = NULL;
}

static Shingling *convert_shingle_list_to_shingling(
    ShingleNode **shingle_list)
{
    Shingling    *shingling;
    ShingleNode  *shingle,
                 *dead_shingle_node;
    int          i;

    /*
     * Allocate the Shingling.
     */
    shingling = NEW_STRUCT(Shingling);

    /*
     * Count the Shingles.
     */
    shingling->num_shingles = 0;
    for (    shingle = *shingle_list;
          shingle != NULL;
          shingle = shingle->next)
        shingling->num_shingles++;

    /*
     * Allocate the array.
     */
    shingling->shingles = NEW_ARRAY(shingling->num_shingles, Shingle);

    /*
     * Copy the Shingles into the array, freeing the ShingleNodes as we go.
     */
    for (i = 0; i < shingling->num_shingles; i++)
    {
        if (*shingle_list == NULL)
            uFatalError("convert_shingle_list_to_shingling", "shingling");

        shingling->shingles[i] = (*shingle_list)->shingle;

        dead_shingle_node    = *shingle_list;
        *shingle_list        = (*shingle_list)->next;
        my_free(dead_shingle_node);
    }
    if (*shingle_list != NULL)
        uFatalError("convert_shingle_list_to_shingling", "shingling");

    return shingling;
}

```