

```

/*
 * polyhedral_group.c
 *
 * This file provides the function
 *
 *     Boolean is_group_polyhedral(SymmetryGroup *the_group);
 *
 * which symmetry_group.c uses to recognize triangle groups.
 * is_group_polyhedral() fills in the the_group's is_polyhedral field,
 * and if it's TRUE it fills in the is_binary_group, p, q and r fields to
 * completely specify the triangle group.
 *
 * is_group_polyhedral() assumes that the_group's order, product[[[]],
 * order_of_element[] and inverse[] fields have been set. (Also the
 * is_dihedral field -- see below.)
 *
 * Note: is_group_polyhedral() assumes that the_group's is_dihedral
 * field has already been set. (It would be trivially easy to have
 * is_group_polyhedral() check for dihedral groups. To do so, just
 * call is_triangle_group(the_group, 2, 2, the_group->order/2, FALSE)
 * near the end of is_group_polyhedral(). I don't currently do this
 * because the special-purpose function is_group_dihedral() in
 * symmetry_group.c not only recognizes the group, but orders the elements
 * in a natural way as well.)
 *
 *
 * The remainder of this comment defines the "extended polyhedral
 * groups" and the "binary polyhedral groups", and explains why the
 * extended groups needn't be considered in the code. I've also included
 * the code (no longer used) for recognizing extended groups directly
 * from a presentation (at the time I wrote it I didn't realize the
 * extended groups aren't necessary.)
 *
 * Throughout the following discussion, "polyhedral" may refer to dihedral
 * as well as tetrahedral, octahedral and icosahedral groups. The polygon
 * corresponding to the dihedral group  $D_n$  is a regular n-sided pillow.
 * (It won't appear quite so degenerate if you think of polyhedra as
 * tilings of the 2-sphere, rather than the traditional things with
 * planar faces.)
 *
 * Definitions:
 *
 *     A plain polyhedral group consists of all orientation-preserving
 *     isometries of the polyhedron.
 *
 *     An extended polyhedral group consists of all isometries of the
 *     polyhedron, including the orientation-reversing ones.
 *
 *     An orientation-preserving isometry is an element of  $SO(3)$ , so a
 *     plain polyhedral group is naturally a subgroup of  $SO(3)$ .  $SO(3)$  is
 *     double covered by  $S^3$ , and the preimage in  $S^3$  of the plain
 *     polyhedral group is called the binary polyhedral group. Just as
 *     one may visualize a plain polyhedral group as the positions of
 *     a plain polyhedron, one may visualize a binary polyhedral group
 *     as the positions of a polyhedron with a "Dirac belt" attached.
 *     Whenever two distinct elements of the binary polyhedral group
 *     project down to the same element of the plain polyhedral group,
 *     they will correspond to polyhedra in identical positions, but
 *     with belts differing by an odd number of turns.
 *
 * Theorem A. An extended dihedral group is isomorphic to the
 * corresponding plain dihedral group cross  $\mathbb{Z}/2$ .
 *
 * Theorem B. The extended octahedral and icosahedral groups are
 * isomorphic to the plain octahedral and icosahedral group cross  $\mathbb{Z}/2$ .
 *
 * Theorem C. The extended tetrahedral group is isomorphic to the
 * plain octahedral group.
 *
 * Taken together, Theorems A, B and C imply that SnapPea's group
 * recognition algorithm need not bother with the extended group --
 * the plain and binary groups suffice.
 *
 * Proof of Theorem A. Ordinarily we visualize the plain dihedral group

```

* as acting on an n-sided pillow, but we may also visualize it as
 * acting on the pillow's equatorial cross section (which is an n-gon).
 * When acting on the pillow the plain dihedral group preserves orientation,
 * but when acting on the n-gon it does not. It's easy to see that the
 * extended dihedral group (acting on the pillow) is the direct product
 * of the plain dihedral group (acting on the n-gon) cross $Z/2$ (acting
 * on the pillows vertical axis). This follows from the fact that the
 * up-down reflections are completely independent of the group's action
 * on the equatorial n-gon. Algebraically, all elements in the
 * extended dihedral group have matrices of the form

$$\begin{pmatrix} * & * & 0 \\ * & * & 0 \\ 0 & 0 & \pm 1 \end{pmatrix}$$

* The two factors consist of matrices of the form

$$\begin{pmatrix} * & * & 0 \\ * & * & 0 \\ 0 & 0 & 1 \end{pmatrix} \quad \text{and} \quad \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -1 \end{pmatrix}$$

* Q.E.D.

* Proof of Theorem B. Let G be either the extended octahedral group
 * or the extended icosahedral group. We'll define an isomorphism to
 * the plain octahedral or icosahedral group cross $Z/2$. Let R be inversion
 * through the origin. Note that both the octahedron and the icosahedron
 * are invariant under R (but the tetrahedron is not, which is why it's
 * not included here). Note too that R commutes with all elements of G
 * (proof: it's matrix is minus the identity).

* Define $f(g) = (g, 0)$ if g preserves orientation,
 * $= (gR, 1)$ if g reverses orientation.

* To prove that f is an isomorphism, we must prove that it is a
 * homomorphism, it's one-to-one, and it's onto.

* To prove that f is a homomorphism, we must show that for all g and h
 * in G , $f(gh) = f(g)f(h)$. We check the four possible cases:

* g preserves orientation and h preserves orientation.

$$\begin{aligned} f(gh) &= (gh, 0) \\ f(g)f(h) &= (g, 0)(h, 0) = (gh, 0) \end{aligned}$$

* g preserves orientation and h reverses orientation.

$$\begin{aligned} f(gh) &= (ghR, 1) \\ f(g)f(h) &= (g, 0)(hR, 1) = (ghR, 1) \end{aligned}$$

* g reverses orientation and h preserves orientation.

$$\begin{aligned} f(gh) &= (ghR, 0) \\ f(g)f(h) &= (gR, 1)(h, 0) = (gRh, 1) = (ghR, 1) \end{aligned}$$

* g reverses orientation and h reverses orientation.

$$\begin{aligned} f(gh) &= (gh, 0) \\ f(g)f(h) &= (gR, 1)(hR, 1) = (gRhR, 0) = (gh, 0) \end{aligned}$$

* To prove that f is one-to-one, assume $f(g) = f(h)$. If $f(g)$ and $f(h)$
 * have second component 0, then $(g, 0) = f(g) = f(h) = (h, 0) \Rightarrow g = h$.
 * Otherwise $(gR, 1) = f(g) = f(h) = (hR, 1) \Rightarrow gR = hR \Rightarrow g = h$.

* To prove that f is onto, note that each element $(g, 0)$ in the range
 * is $f(g)$, while each element $(g, 1)$ is $f(gR)$.

* Q.E.D.

* Proof of Theorem C. We will define an isomorphism from the extended
 * tetrahedral group to the plain octahedral group.

* The proof relies on the following image which you should draw
 * or -- better still -- build with polydrons. Make a regular octahedron
 * with blue and red faces alternating checkerboard-style. Now attach
 * a blue tetrahedron to each blue face of the octahedron; the easiest
 * way to do this is to remove each existing blue face and replace it
 * three blue triangles which have been snapped together to form the
 * three new faces of the tetrahedron you're attaching. When you're

```

* done you'll have a large tetrahedron (edge length two), each face
* of which consists of three blue triangles surrounding a red one.
* Throughout this proof you should imagine the original octahedron
* sitting inside the large tetrahedron.
*
* We will define a map f from the extended group of isometries of the
* large tetrahedron to the plain group of isometries of the original
* octahedron.
*
* R will be the central inversion, as in the proof of Theorem B.
* It commutes with all elements of SO(3).
*
* Let f(g)      = g      if g preserves orientation,
*                = gR     if g reverses orientation.
*
* To prove that f is an isomorphism, we must prove that it is a
* homomorphism, it's one-to-one, and it's onto.
*
* To prove that f is a homomorphism, we must show that for all g and h,
* f(gh) = f(g)f(h). We check the four possible cases:
*
*   g preserves orientation and h preserves orientation.
*       f(gh) = gh
*       f(g)f(h) = gh
*
*   g preserves orientation and h reverses orientation.
*       f(gh) = (gh)R
*       f(g)f(h) = g(hR)
*
*   g reverses orientation and h preserves orientation.
*       f(gh) = ghR
*       f(g)f(h) = (gR)h = ghR
*
*   g reverses orientation and h reverses orientation.
*       f(gh) = gh
*       f(g)f(h) = (gR)(hR) = gh
*
* To prove that f is one-to-one, assume f(g) = f(h). If f(g) and f(h)
* preserve the red-blue checkboard coloring of the octahedron, then
* g and h are orientation-preserving and g = f(g) = f(h) = h.
* If f(g) and f(h) reverse the coloring, then g and h are orientation
* reversing and gR = f(g) = f(h) = hR => g = h.
*
* To prove that f is onto, note that each element g which preserves the
* octahedron's coloring is f(g) for the orientation-preserving element g,
* while each element g which reverses the coloring is f(gR) for the
* orientation-reversing element gR.
*
* Q.E.D.
*
* Here's the code which used to check for extended polyhedral groups.
*/

#if 0 /* beginning of unused code */

static Boolean is_binary_triangle_group(
    SymmetryGroup *the_group,
    int p,
    int q,
    int r)
{
    /*
     * The binary (p,q,r) triangle group is like the plain one defined
     * in is_plain_triangle_group() above, except that all symmetries
     * of the labelled tiling are allowed, including the orientation-
     * reversing ones.
     *
     * Theorem. The binary (p,q,r) triangle group has presentation
     *
     *  = {a,b,c | (ab)^p = (bc)^q = (ca)^r = a^2 = b^2 = c^2 = 1}
     *
     * Proof. The proof is similar to the proof for the plain case
     * in is_plain_triangle_group() above, so here I will just point
    */
}

```

```

* out the differences.
* The first difference is that the generators a, b and c have
* a different interpretation. Here a, b and c are reflections in
* the three sides of the triangle.
* As before, we pick a base triangle. But unlike the plain case,
* there is no need to color the triangles blue and red. Because
* orientation reversing symmetries are allowed, every triangle is a
* potential image of the base triangle, so every triangle is colored
* blue and has a dot in the middle to represent the corresponding
* symmetry.
* There will now be two arcs connecting the centers of each pair
* of adjacent triangles. One arc points in each direction. The
* directions are chosen so that the arcs form a clockwise-directed
* bigon. That is, they pass each other like cars in Japan or
* Australia, not cars in Italy or the U.S.
* The arcs divide the surface (i.e. the sphere, Euclidean plane or
* hyperbolic plane, depending on the group) into four types of regions:
*
* (1) 2p-gons labelled "abab...", oriented counterclockwise
* (2) 2q-gons labelled "bcbc...", oriented counterclockwise
* (3) 2r-gons labelled "caca...", oriented counterclockwise
* (4) bigons labelled "aa", "bb" or "cc", oriented clockwise
*
* (Line (3) is neither offensive nor amusing to speakers of English.)
*
* The remainder of the proof is identical to that of the plain case.
* Q.E.D.
*
* Note: As in the plain case, we have shown that in the abstract
* group  $\{a,b,c \mid (ab)^p = (bc)^q = (ca)^r = a^2 = b^2 = c^2 = 1\}$ ,
* ab, ba and ca have order exactly p, q and r, respectively, and
* a, b and c each have order exactly 2.
*/

int a,
    b,
    c,
    ab,
    bc,
    ca,
    i,
    possible_generators[3];

/*
* Consider all possible images of the generator a in the_group.
*/
for (a = 0; a < the_group->order; a++)
{
    /*
    * If a does not have order 2, ignore it and move on.
    */
    if (the_group->order_of_element[a] != 2)
        continue;

    /*
    * Consider all possible images of the generator b in the_group.
    */
    for (b = 0; b < the_group->order; b++)
    {
        /*
        * If b does not have order 2, ignore it and move on.
        */
        if (the_group->order_of_element[b] != 2)
            continue;

        /*
        * If the product ab does not have order p, move on.
        */
        ab = the_group->product[a][b];
        if (the_group->order_of_element[ab] != p)
            continue;

        /*
        * Consider all possible images of the generator c in the_group.

```

```

    */
    for (c = 0; c < the_group->order; c++)
    {
        /*
         * If c does not have order 2, ignore it and move on.
         */
        if (the_group->order_of_element[c] != 2)
            continue;

        /*
         * If the product bc does not have order q, move on.
         */
        bc = the_group->product[b][c];
        if (the_group->order_of_element[bc] != q)
            continue;

        /*
         * If the product ca does not have order r, move on.
         */
        ca = the_group->product[c][a];
        if (the_group->order_of_element[ca] != r)
            continue;

        /*
         * At this point we know
         *  $(ab)^p = (bc)^q = (ca)^r = a^2 = b^2 = c^2 = 1$ .
         * We have a homomorphism from the binary (p,q,r) triangle
         * group to the_group. It will be an isomorphism iff
         * a, b and c generate the_group.
         */

        /*
         * Write a, b and c into an array . . .
         */
        possible_generators[0] = a;
        possible_generators[1] = b;
        possible_generators[2] = c;

        /*
         * . . . and pass the array to the function which checks
         * whether they generate the group.
         */
        if (elements_generate_group(the_group, 3, possible_generators) == TRUE)
            return TRUE;

        /*
         * If a, b and c failed to generate the_group, we continue
         * on with the hope that some other choice of a, b and c
         * will work.
         */
    }
}

return FALSE;
}

#endif /* end of unused code */

#include "kernel.h"

static Boolean is_triangle_group(SymmetryGroup *the_group, int p, int q, int r, Boolean
    check_binary_group);
static Boolean is_plain_triangle_group(SymmetryGroup *the_group, int p, int q, int r);
static Boolean is_binary_triangle_group(SymmetryGroup *the_group, int p, int q, int r);

Boolean is_group_polyhedral(
    SymmetryGroup *the_group)
{
    /*
     * By the time this function is called, we should have already checked
     * whether the group is dihedral. If it is, just fill in the
    */

```

```
    * required information and return TRUE.
    */

if (the_group->is_dihedral == TRUE)
{
    the_group->is_polyhedral = TRUE;

    the_group->is_binary_group = FALSE;

    the_group->p = 2;
    the_group->q = 2;
    the_group->r = the_group->order / 2;

    return TRUE;
}

/*
 * Is this a tetrahedral, octahedral or isocahedral group?
 */

switch (the_group->order)
{
    case 12:

        /*
         * Is it the plain tetrahedral group?
         */
        if (is_triangle_group(the_group, 2, 3, 3, FALSE))
            return TRUE;

        break;

    case 24:

        /*
         * Is it the binary tetrahedral group?
         */
        if (is_triangle_group(the_group, 2, 3, 3, TRUE))
            return TRUE;

        /*
         * Is it the plain octahedral group?
         */
        if (is_triangle_group(the_group, 2, 3, 4, FALSE))
            return TRUE;

        break;

    case 48:

        /*
         * Is it the binary octahedral group?
         */
        if (is_triangle_group(the_group, 2, 3, 4, TRUE))
            return TRUE;

        break;

    case 60:

        /*
         * Is it the plain icosahedral group?
         */
        if (is_triangle_group(the_group, 2, 3, 5, FALSE))
            return TRUE;

        break;

    case 120:

        /*
         * Is it the binary icosahedral group?
         */
        if (is_triangle_group(the_group, 2, 3, 5, TRUE))
```

```

        return TRUE;

        break;
    }

    /*
     * Is this a binary dihedral group?
     */

    if (the_group->order % 4 == 0)
        if (is_triangle_group(the_group, 2, 2, the_group->order/4, TRUE))
            return TRUE;

    /*
     * None of the above.
     */

    the_group->is_binary_group = FALSE;
    the_group->p               = 0;
    the_group->q               = 0;
    the_group->r               = 0;

    return FALSE;
}

static Boolean is_triangle_group(
    SymmetryGroup *the_group,
    int           p,
    int           q,
    int           r,
    Boolean       check_binary_group)
{
    the_group->is_polyhedral =
        (check_binary_group == TRUE) ?
        is_binary_triangle_group(the_group, p, q, r) :
        is_plain_triangle_group(the_group, p, q, r);

    if (the_group->is_polyhedral == TRUE)
    {
        the_group->is_binary_group = check_binary_group;
        the_group->p               = p;
        the_group->q               = q;
        the_group->r               = r;
    }

    return the_group->is_polyhedral;
}

static Boolean is_plain_triangle_group(
    SymmetryGroup *the_group,
    int           p,
    int           q,
    int           r)
{
    /*
     * A (p,q,r) triangle is a triangle with angles pi/p, pi/q and pi/r.
     * The triangle will be spherical, Euclidean or hyperbolic according
     * to whether the sum of the angles ( = pi(1/p + 1/q + 1/r) ) is
     * greater than, equal to, or less than pi.  Imagine tiling a sphere,
     * Euclidean plane, or hyperbolic plane (according to the geometry
     * of the triangle) with (p,q,r) triangles by starting with one such
     * triangle and recursively reflecting it across its own sides.
     * The (p,q,r) triangle group is the group of orientation-preserving
     * symmetries which preserve the tiling.  If the triangles are colored
     * red and blue checkerboard-fashion, the orientation-preserving
     * symmetries will preserve the coloring.  (N.B. The group preserves
     * labelled triangles.  If the triangles have "extra" symmetries --
     * such as in the (4,4,4) triangle group -- we exclude them from
     * consideration.)
     *
     * Theorem.  The (p,q,r) triangle group has presentation
     */
}

```

```

*      (p,q,r) = {a,b,c | a^p = b^q = c^r = abc = 1}
*
* Proof. The generators a, b and c represent counterclockwise
* rotations of  $2\pi/p$ ,  $2\pi/q$  and  $2\pi/r$ , respectively, about the
* vertices A, B and C of a (p,q,r) triangle. The vertices A, B and C
* are arranged in counterclockwise order around the triangle.
* To understand this proof, it will be very useful to make
* yourself a picture, as follows. Draw a portion of a tiling by
* (p,q,r) triangles, colored blue and red, checkerboard fashion.
* We'll work mainly with the blue triangles, ignoring the red ones.
* Pick one blue triangle to be the "base triangle", and label its
* vertices A, B and C, going counterclockwise. The blue triangles
* are in one-to-one correspondence with the elements of the (p,q,r)
* triangle group: each symmetry is associated with the blue triangle
* to which it maps the base triangle. Put a dot in the middle of
* each blue triangle to represent the associated symmetry.
* Each vertex which is equivalent (under the symmetry group) to
* vertex A will be incident to exactly p blue triangles.
* Connect the dots at the centers of these p blue triangles with
* arcs, in a cyclic fashion; that is, an oriented arc runs from
* the center of each incident blue triangle to the center of the
* next incident blue triangle going counterclockwise. Label the
* arcs with the letter "a", and include arrows to show the direction.
* Do the same for all vertices equivalent vertices B and C.
* When you are done, you will have divided the surface (i.e. the
* sphere, Euclidean plane or hyperbolic plane, depending on the
* group) into four types of regions:
*
* (1) p-gons labelled "a", oriented counterclockwise
* (2) q-gons labelled "b", oriented counterclockwise
* (3) r-gons labelled "c", oriented counterclockwise
* (4) triangles labelled a-b-c, oriented clockwise
*
* (One a-b-c triangle sits over each red triangle in the tiling.)
* We'll soon see that these four types of regions give us the
* relations in the group.
*
* We are now prepared to define a map from the abstract group
*
*      {a,b,c | a^p = b^q = c^r = abc = 1}
*
* into the (p,q,r) triangle group.
*
* (0) Definition of map. The generators a, b and c map to the
* counterclockwise rotations of  $2\pi/p$ ,  $2\pi/r$  and  $2\pi/q$  about
* the vertices A, B and C of the base triangle. (And their
* inverses map to clockwise rotations, of course).
*
* (1) The map is a homomorphism from the free group on {a,b,c} to
* the (p,q,r) triangle group. This part of the proof relies on
* the convention that symmetries are performed right-to-left.
* For example, the word "abc" means "do symmetry c, then b,
* then a". It's easy to see that the image of the base triangle
* under the symmetry abc is the may be found by beginning at the
* base triangle and following the (forward pointing) arcs labelled
* "a", then "b", then "c". It then follows that the product of
* two symmetries, e.g.  $abc * baac = abcbaac$ , is obtained by
* concatenating the corresponding arc-paths. If inverses of
* generators were involved, we'd go against the direction of
* the arrows.
*
* (2) The kernel of the map from the free group on {a,b,c} to the
* (p,q,r) triangle group includes the relations
*  $a^p = b^q = c^r = abc = 1$ . This is trivial -- just trace out
* the words  $a^p$ ,  $b^q$ ,  $c^r$  and  $abc$  in your picture, and note that
* all four are closed loops. This tells us that the map defined
* on the free group on {a,b,c} projects to a map which is well-
* defined on the quotient group  $\{a,b,c \mid a^p = b^q = c^r = abc = 1\}$ .
* (It also explains why the four regions listed above give us the
* relations in the group presentation.)
*
* (3) The quotient map is onto. This follows immediately from the
* fact that the graph (of dots connected by "a", "b" and "c" arcs)
* is connected.

```



```

*
* (4) The quotient map is one-to-one. This follows from the fact
* that the underlying space (the sphere, Euclidean plane or
* hyperbolic plane) is simply connected. A word, e.g.
* "a(b^-1)cacbc(c^-1)", maps to the identity iff the corresponding
* path in your picture is a closed loop. First consider the
* special case that it's a simple closed loop. The closed loop
* bounds some number of regions (cf. types (1)-(4) above).
* The corresponding relations may be conjugated and multiplied
* to show that the given word lies in the normal subgroup generated
* by  $a^p = b^q = c^r = abc = 1$ . If the loop is not simple, we
* use the preceding technique to remove subloops which are simple,
* until the whole word has been shown to be trivial. (Yes, I
* realize this explanation is light on details, but I hope the
* idea is clear.)
*
* (5) The map is an isomorphism. This follows from (3) and (4).
*
* Q.E.D.
*
* Note: The above proof shows not only that the abstract group
*  $\{a,b,c \mid a^p = b^q = c^r = abc = 1\}$  is isomorphic to the
*  $(p,q,r)$  triangle group. It also shows that the generators
*  $a, b$  and  $c$  have order exactly  $p, q$  and  $r$ . (This is not obvious
* from the presentation alone). We'll use this fact in the code
* below.
*
* Getting back to the computational problem at hand . . .
*
* We seek an isomorphism from  $\{a,b,c \mid a^p = b^q = c^r = abc = 1\}$  to
* the_group. An isomorphism is given by specifying the images of the
* generators  $a, b$  and  $c$ . Naively one would try all possibilities
* for the images of  $a, b$  and  $c$ , and in each case check whether
* the relations are satisfied. This would be a cubic time algorithm,
* as a function of the size of the group. We can reduce this to a
* quadratic time algorithm by considering all possible images of
*  $a$  and  $b$ , and in each case letting  $c = (ab)^{-1}$ . The algorithm is
* further streamlined by mapping  $a$  and  $b$  only to elements of order
*  $p$  and  $q$ , respectively.
*
* Once we find images for  $a, b$  and  $c$  satisfying the relations
*  $a^p = b^q = c^r = abc = 1$  we know we have a homomorphism from
*  $\{a,b,c \mid a^p = b^q = c^r = abc = 1\}$  to the_group. Because the
* function is_group_polyhedral() insures that the_group has the same
* order as the  $(p,q,r)$  triangle group, to check whether the
* homomorphism is an isomorphism it suffices to check that it is onto.
*
* In the following code, the variables "a", "b" and "c" represent
* the images of  $a, b$  and  $c$  in the_group.
*/

int a,
    b,
    c,
    possible_generators[3];

/*
* Consider all possible images of the generator a in the_group.
*/
for (a = 0; a < the_group->order; a++)
{
    /*
    * If a does not have order p, ignore it and move on.
    */
    if (the_group->order_of_element[a] != p)
        continue;

    /*
    * Consider all possible images of the generator b in the_group.
    */
    for (b = 0; b < the_group->order; b++)
    {
        /*

```

```

    * If b does not have order q, ignore it and move on.
    */
    if (the_group->order_of_element[b] != q)
        continue;

    /*
    * The relation  $abc = 1$  will be satisfied iff  $c = (ab)^{-1}$ .
    */
    c = the_group->inverse[the_group->product[a][b]];

    /*
    * c should have order r.
    */
    if (the_group->order_of_element[c] != r)
        continue;

    /*
    * At this point we know  $a^p = b^q = c^r = abc = 1$ .
    * We have a homomorphism from the  $(p,q,r)$  triangle group
    * to the_group. It will be an isomorphism iff a, b and c
    * generate the_group.
    */

    /*
    * Write a, b and c into an array . . .
    */
    possible_generators[0] = a;
    possible_generators[1] = b;
    possible_generators[2] = c;

    /*
    * . . . and pass the array to the function which checks
    * whether they generate the group.
    */
    if (elements_generate_group(the_group, 3, possible_generators) == TRUE)
        return TRUE;

    /*
    * If a, b and c failed to generate the_group, we continue on
    * with the hope that some other choice of a and b will work.
    */
}

return FALSE;
}

static Boolean is_binary_triangle_group(
    SymmetryGroup *the_group,
    int p,
    int q,
    int r)
{
    /*
    * I don't yet know how to prove that the following presentation
    * for the binary triangle group is correct, but I want to get
    * the code up and running now anyhow. (Pat Callahan found the
    * presentation in a book and e-mailed it to me, so I'm pretty
    * sure it's correct.)
    *
    * Theorem (relayed by Pat). The binary  $\langle p,q,r \rangle$  triangle group has
    * presentation
    *
    *  $\langle p,q,r \rangle = \{a,b,c \mid a^p = b^q = c^r = abc\}$ 
    *
    * Theorem (modified by me). The binary  $\langle p,q,r \rangle$  triangle group has
    * presentation
    *
    *  $\langle p,q,r \rangle = \{a,b,c,d \mid a^p = b^q = c^r = abc = d, d^2 = 1\}$ 
    *
    * The element d is the identity map from the polyhedron to itself,
    * but with a single full turn in "the belt". Thus d has order
    * exactly 2, while a, b and c have orders exactly 2p, 2r and 2q,
    */
}

```

```

* respectively.
*
* Proof: ???
* [In the spherical case one ought to be able to prove this by
* mapping a, b and c to the appropriate quaternions. JRW 96/2/6]
*
* Also, I'm not sure to what extent this presentation -- and my
* added commentary -- is meaningful in the Euclidean and hyperbolic
* cases. Probably it works there too, but I don't really know.
*
* I need to find a copy of Coxeter & Moser's "Generators and Relations
* for Discrete Groups".
*
* In the following code, the variables "a", "b" and "c" represent
* the images of a, b and c in the_group.
*/

int a,
    b,
    c,
    ap,
    bq,
    cr,
    count,
    possible_generators[3];

/*
 * Consider all possible images of the generator a in the_group.
 */
for (a = 0; a < the_group->order; a++)
{
    /*
     * If a does not have order 2p, ignore it and move on.
     */
    if (the_group->order_of_element[a] != 2*p)
        continue;

    /*
     * Compute a^p.
     */
    ap = 0;
    for (count = 0; count < p; count++)
        ap = the_group->product[ap][a];

    /*
     * Consider all possible images of the generator b in the_group.
     */
    for (b = 0; b < the_group->order; b++)
    {
        /*
         * If b does not have order 2q, ignore it and move on.
         */
        if (the_group->order_of_element[b] != 2*q)
            continue;

        /*
         * Compute b^q.
         */
        bq = 0;
        for (count = 0; count < q; count++)
            bq = the_group->product[bq][b];

        /*
         * If a^p != b^q, move on.
         */
        if (ap != bq)
            continue;

        /*
         * The relation abc = a^p will be satisfied
         * iff c = (ab)^-1 a^p.
         */
        c = the_group->product
            [the_group->inverse[the_group->product[a][b]]]

```

```

        [ap];

    /*
     * c should have order 2r.
     */
    if (the_group->order_of_element[c] != 2*r)
        continue;

    /*
     * Compute c^r.
     */
    cr = 0;
    for (count = 0; count < r; count++)
        cr = the_group->product[cr][c];

    /*
     * If a^p != c^r, move on.
     */
    if (ap != cr)
        continue;

    /*
     * At this point we know a^p = b^q = c^r = abc.
     * We have a homomorphism from the (p,q,r) triangle group
     * to the_group. It will be an isomorphism iff a, b and c
     * generate the_group.
     */

    /*
     * Write a, b and c into an array . . .
     */
    possible_generators[0] = a;
    possible_generators[1] = b;
    possible_generators[2] = c;

    /*
     * . . . and pass the array to the function which checks
     * whether they generate the group.
     */
    if (elements_generate_group(the_group, 3, possible_generators) == TRUE)
        return TRUE;

    /*
     * If a, b and c failed to generate the_group, we continue on
     * with the hope that some other choice of a and b will work.
     */
    }
}

return FALSE;
}

```