

```

/*
 * Dirichlet_rotate.c
 *
 * This file provides the following functions for the UI, to allow
 * convenient Dirichlet domain rotation.
 *
 * void set_identity_matrix(O3lMatrix position);
 * void set_poly_velocity(O3lMatrix velocity, double theta[3]);
 * void update_poly_position(O3lMatrix position, O3lMatrix velocity);
 * void update_poly_vertices(WEPolyhedron *polyhedron,
 *                           O3lMatrix position, double scale);
 * void update_poly_visibility(WEPolyhedron *polyhedron,
 *                             O3lMatrix position, O3lVector direction);
 *
 * set_identity_matrix() sets a position or velocity matrix to the identity.
 *
 * set_poly_velocity() takes as input the small rotation angles theta[]
 * about each of the coordinate axes, and computes the corresponding
 * rotation matrix. set_poly_velocity() may be used to update the
 * velocity when the user drags the polyhedron with the mouse.
 * For example, say the user interface is using a coordinate system
 * in which the x-axis points to the right, the y-axis points down,
 * and the z-axis points into the screen, and the user drags the
 * mouse a small distance (dx, dy). The dx corresponds to a small
 * clockwise rotation about the y-axis. The dy corresponds to a small
 * counterclockwise rotation about the x-axis. So we pass the vector
 * theta[3] = (dy, -dx, 0) to set_poly_velocity() and get back the
 * corresponding velocity matrix.
 *
 * update_poly_position() updates the position by left-multiplying by the
 * velocity. Call it to keep the polyhedron moving.
 *
 * update_poly_vertices() multiplies the standard vertex coordinates x[]
 * by the position matrix to obtain the rotated coordinates xx[].
 * It then multiplies the rotated coordinates by the constant "scale";
 * this lets the UI scale the rotated coordinates to match the window
 * coordinates (if you don't need this feature, pass scale = 1.0).
 *
 * update_poly_visibility() checks which vertices, edges and faces are
 * visible to the user with the polyhedron in its present position,
 * and sets their visibility fields accordingly. The direction vector
 * points from the center of the polyhedron to the user. Note that
 * the direction vector is an O3lVector; just set the 0-th component
 * to 0.0.
 */

#include "kernel.h"

static void set_face_visibility(WEPolyhedron *polyhedron, O3lMatrix position, O3lVector direction);
static void set_edge_visibility(WEPolyhedron *polyhedron);
static void set_vertex_visibility(WEPolyhedron *polyhedron);

void set_identity_matrix(
    O3lMatrix m)
{
    /*
     * This function is just a wrapper around a call to
     * o3l_copy(position, O3l_identity), to avoid giving
     * the UI access to the kernel's O(3,1) matrix library.
     */
    o3l_copy(m, O3l_identity);
}

void update_poly_position(
    O3lMatrix position,
    O3lMatrix velocity)
{
    /*
     * Multiply the position by the velocity to get the new position.
     */
}

```

```

    o3l_product(velocity, position, position);
}

void update_poly_vertices(
    WEPolyhedron    *polyhedron,
    O3lMatrix        position,
    double           scale)
{
    WEVertex        *vertex;

    for (vertex = polyhedron->vertex_list_begin.next;
         vertex != &polyhedron->vertex_list_end;
         vertex = vertex->next)
    {
        o3l_matrix_times_vector(position, vertex->x, vertex->xx);
        o3l_constant_times_vector(scale, vertex->xx, vertex->xx);
    }
}

void update_poly_visibility(
    WEPolyhedron    *polyhedron,
    O3lMatrix        position,
    O3lVector        direction)
{
    /*
     * Check which vertices, edges and faces are visible to the user with
     * the polyhedron in its present position, and set their visibility
     * fields accordingly. The direction vector points from the center
     * of the polyhedron to the user.
     */

    /*
     * direction[0] is probably zero already, but just in case it's not,
     * set it to zero explicitly. (We're using O3lVectors, but we care
     * only about the 3-dimensional part.)
     */
    direction[0] = 0.0;

    /*
     * Set the face visibility first.
     */
    set_face_visibility(polyhedron, position, direction);

    /*
     * An edge is visible iff it lies on a visible face.
     */
    set_edge_visibility(polyhedron);

    /*
     * A vertex is visible iff it lies on a visible edge.
     */
    set_vertex_visibility(polyhedron);
}

static void set_face_visibility(
    WEPolyhedron    *polyhedron,
    O3lMatrix        position,
    O3lVector        direction)
{
    WEFace          *face;
    O3lVector        old_normal,
                    new_normal;
    int              i;

    for (face = polyhedron->face_list_begin.next;
         face != &polyhedron->face_list_end;
         face = face->next)
    {
        /*
         * The first column of the group_element provides a normal

```

```

    /* vector relative to the polyhedron's original position.
    */
    for (i = 0; i < 4; i++)
        old_normal[i] = (*face->group_element)[i][0];

    /*
    * Apply the position matrix to find the normal vector relative
    * to the polyhedron's current position.
    */
    o3l_matrix_times_vector(position, old_normal, new_normal);

    /*
    * The face will be visible iff the new_normal has a positive
    * component in the given direction.
    */
    face->visible = (o3l_inner_product(direction, new_normal) > 0.0);
}
}

static void set_edge_visibility(
    WEPolyhedron *polyhedron)
{
    WEEdge *edge;

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

        edge->visible = (edge->f[left]->visible || edge->f[right]->visible);
}

static void set_vertex_visibility(
    WEPolyhedron *polyhedron)
{
    WEVertex *vertex;
    WEEdge *edge;

    /*
    * Initialize all vertex visibilities to FALSE.
    */

    for (vertex = polyhedron->vertex_list_begin.next;
         vertex != &polyhedron->vertex_list_end;
         vertex = vertex->next)

        vertex->visible = FALSE;

    /*
    * The endpoints of each visible edge will be visible vertices.
    */

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

        if (edge->visible)
        {
            edge->v[tail]->visible = TRUE;
            edge->v[tip]->visible = TRUE;
        }
}

```