

```

/*
 * winged_edge.h
 *
 * This file defines the usual "winged edge" data structure for
 * representing a convex polyhedron, along with some extra
 * fields describing the polyhedron's position in the projective
 * model of hyperbolic 3-space. (The "projective model" is
 * the Minkowski space model projected onto the hyperplane
 * x[0] == 1.)
 *
 * This file is intended solely for #inclusion in SnapPea.h.
 * It needs some of the typedefs which occur there.
 */

#ifndef _winged_edge_
#define _winged_edge_

#define INFINITE_RADIUS      1e20
#define INFINITE_DISTANCE    1e20
#define INFINITE_LENGTH      1e20

/*
 * The values of the following two enums are used to index the
 * static arrays in make_cube() in Dirichlet_construction.c,
 * so their values shouldn't be changed. (They are also toggled in
 * Dirichlet_extras.c Dirichlet_conversion.c with the "not" operator '!'.)
 */

typedef int WEEdgeEnd;
enum
{
    tail = 0,
    tip  = 1
};

typedef int WEEdgeSide;
enum
{
    left  = 0,
    right = 1
};

/*
 * The WEEdge structure keeps pointers to Tetrahedra for local use
 * in Dirichlet_conversion.c. The internal structure of a Tetrahedron
 * is private to the kernel, so we must include an "opaque typedef" here.
 * (Indeed even the existence of the Tetrahedron structure is private to
 * the kernel, so we are "cheating" a bit even by including the typedef.)
 */
typedef struct Tetrahedron      TetrahedronSneak;

/*
 * Forward declarations.
 */

typedef struct WEVertex          WEVertex;
typedef struct WEEdge            WEEdge;
typedef struct WEFace            WEFace;

typedef struct WEVertexClass      WEVertexClass;
typedef struct WEEdgeClass        WEEdgeClass;
typedef struct WEFaceClass        WEFaceClass;

typedef struct WEPolyhedron       WEPolyhedron;

struct WEVertex
{
    /*
     * The vector x gives the position of the WEVertex in the
     * projective model of hyperbolic 3-space. The projective
     * model is viewed as a subset of the Minkowski space model,
     * so x is a 4-element vector with x[0] == 1.0.
     */

```

```

    */
    O31Vector      x;

/*
 * The vector xx[] is an extra copy of x[] for local use in
 * the user interface. Even as the polyhedron spins, the UI
 * should not modify x[], but should write the coordinates
 * of the rotated vertices into xx[] instead. There are two
 * reasons for this:
 *
 * (1) This scheme avoids cumulative roundoff error in the
 *     coordinates, which could in principle deform the
 *     polyhedron. One can argue that on a 680x0 Mac, which
 *     has 8-byte mantissas, a polyhedron could spin for a
 *     million years before enough error would accumulate
 *     to make a 1-pixel difference on the screen. However,
 *     on an Iris, which uses floats instead of doubles, the
 *     roundoff error is a real issue. In any case, good
 *     programming style dictates that a routine for displaying
 *     a polyhedron should not alter the polyhedron it's
 *     displaying.
 *
 * (2) If the user changes the Dehn filling coefficients
 *     slightly, say from (7,1) to (8,1), the change in the
 *     polyhedron will be small, and we'd like the new polyhedron
 *     to appear in roughly the same position as the old one,
 *     so the continuity is visible. For this to occur, x[]
 *     must always contain the polyhedron's initial position,
 *     not its rotated position.
 */
    O31Vector      xx;

/*
 * The distance from the vertex to the origin. If the vertex is ideal,
 * dist is set to INFINITE_DISTANCE. Even though just the distance is
 * given here, the UI may want to display cosh(dist) as well.
 */
    double         dist;

/*
 * Is this an ideal vertex?
 */
    Boolean        ideal;

/*
 * The solid angle at this vertex of the Dirichlet domain.
 */
    double         solid_angle;

/*
 * The Dirichlet domain's face pairings group the vertices
 * into vertex classes.
 */
    WEVertexClass  *v_class;

/*
 * The visible field is used while displaying the WEPolyhedron to
 * keep track of whether the vertex is visible to the user.
 */
    Boolean        visible;

/*
 * The distance_to_plane field is used locally within
 * Dirichlet_construction.c to record the inner product
 * of the WEVertex's location x[] with an arbitrary but
 * fixed normal vector to the hyperplane currently under
 * consideration. Thus, distance_to_plane is proportional
 * to the Euclidean distance in the projective model from
 * the point (x[1], x[2], x[3]) to the intersection of the
 * hyperplane with the projective model (at x[0] == 1.0).
 *
 * The which_side_of_plane field is +1, 0 or -1 according
 * to whether, after accounting for possible roundoff error,
 * distance_to_plane is positive, zero or negative, respectively.

```

```

    */
    double          distance_to_plane;
    int             which_side_of_plane;

    /*
     * The zero_order field is used locally in check_topology_of_cut()
     * to verify that precisely zero or two 0-edges are incident to
     * each 0-vertex.
     */
    int             zero_order;

    /*
     * The WEVertices are kept on a doubly-linked list.
     */
    WEVertex        *prev,
                    *next;
};

struct WEEdge
{
    /*
     * v[tail] and v[tip] are the vertices incident to the
     * tail and tip, respectively, of the directed WEEdge.
     */
    WEVertex        *v[2];

    /*
     * e[tail][left] is the WEEdge incident to both v[tail] and f[left].
     * e[tail][right] is the WEEdge incident to both v[tail] and f[right].
     * e[tip ][left] is the WEEdge incident to both v[tip ] and f[left].
     * e[tip ][right] is the WEEdge incident to both v[tip ] and f[right].
     */
    WEEdge          *e[2][2];

    /*
     * f[left] and f[right] are the faces incident to the
     * left and right sides, respectively, of the directed WEEdge.
     */
    WEFace          *f[2];

    /*
     * The dihedral angle between edge->f[left] and edge->f[right].
     */
    double          dihedral_angle;

    /*
     * dist_line_to_origin is the distance to the origin from the line
     * containing the edge.
     *
     * dist_edge_to_origin is the distance to the origin from the edge
     * itself. Usually this will be the same as dist_line_to_origin,
     * but occasionally the minimum distance from the line to the origin
     * will be realized at a point not on the edge itself. In the latter
     * case the minimum distance from the origin to the edge itself will
     * be realized at an endpoint.
     *
     * Note: The UI may want to display the squared hyperbolic cosines
     * of the above distances, as well as the distances themselves.
     *
     * closest_point_on_line and closest_point_on_edge record the points
     * at which the above minima are realized. (At present the Mac UI
     * ignores these, but eventually we may want to display them to the
     * user upon request.)
     */
    double          dist_line_to_origin,
                    dist_edge_to_origin;
    O31Vector       closest_point_on_line,
                    closest_point_on_edge;

    /*
     * How long is this edge?
     * If it's infinite, the length is set to INFINITE_LENGTH.
     */

```

```

    */
    double          length;

    /*
    *   The Dirichlet domain's face pairings group the edges
    *   into edge classes.
    */
    WEEdgeClass      *e_class;

    /*
    *   The visible field is used while displaying the WEPolyhedron to
    *   keep track of whether the edge is visible to the user.
    */
    Boolean          visible;

    /*
    *   The Dirichlet domain's face identifications determine which sets
    *   of edges are identified to single edges in the manifold itself.
    *   For each edge on the Dirichlet domain,
    *
    *       edge->neighbor[left] tells the WEEdge to which the given edge
    *       is mapped by edge->f[left]->group_element,
    *
    *       edge->preserves_sides[left] tell whether the left side of the
    *       given edge maps to the left side of the image,
    *
    *       edge->preserves_direction[left] tells whether the mapping
    *       preserves the direction of the edge, and
    *
    *       edge->preserves_orientation[left] tells whether the mapping
    *       preserves orientation,
    *
    *   and similarly for edge->neighbor[right], etc.
    *
    *   The preserves_sides, preserves_direction and preserves_orientation
    *   fields are intentionally redundant. Any two determine the third.
    *
    *   If the singular set is empty or consists of disjoint circles (as will
    *   always be the case for Dehn fillings on cusped manifolds), then the
    *   function Dirichlet_bells_and_whistles() will redirect WEEdges as
    *   necessary so that the preserves_direction[] fields are all TRUE.
    *   Even for orbifolds with more complicated singular sets, it will
    *   give consistent directions to the edges whenever possible.
    */
    WEEdge          *neighbor[2];
    Boolean          preserves_sides[2],
                    preserves_direction[2],
                    preserves_orientation[2];

    /*
    *   The tet[][] fields are used locally in Dirichlet_conversion.c
    *   to construct a Triangulation for the manifold represented by
    *   the Dirichlet domain. Otherwise they may be ignored.
    *   The four Tetrahedra incident to this WEEdge are tet[tail][left],
    *   tet[tail][right], tet[tip][left], and tet[tip][right].
    */
    TetrahedronSneak *tet[2][2];

    /*
    *   The WEEdges are kept on a doubly-linked list.
    */
    WEEdge          *prev,
                    *next;
};

struct WEFace
{
    /*
    *   some_edge is an arbitrary WEEdge incident to the given WEFace.
    */
    WEEdge          *some_edge;

```

```

/*
 * mate is the WEFace which is identified to this face under
 * the action of the covering transformation group.
 */
WEFace          *mate;

/*
 * group_element is the O(3,1) matrix which takes this face's mate
 * to this face. In other words, this face lies in the plane which
 * passes orthogonally through the midpoint of the segment connecting
 * the origin to its (the origin's) image under the group_element.
 */
O3lMatrix       *group_element;

/*
 * The distance from the face plane to the origin. The point of
 * closest approach may or may not lie on the face itself.
 */
double          dist;
O3lVector       closest_point;

/*
 * The to_be_removed field is used locally in install_new_face() in
 * Dirichlet_construction.c to record which WEFaces are to be removed.
 */
Boolean         to_be_removed;

/*
 * The clean field is used locally in check_faces() in
 * Dirichlet_construction.c to record which WEFaces are known to be
 * subsets of their mates under the action of the group_element.
 */
Boolean         clean;

/*
 * The copied field is used locally in rewrite_gen_list() and
 * (independently) in poly_to_current_list() in Dirichlet_construction.c
 * to record which WEFaces have had their group_elements copied to the
 * MatrixPairList.
 */
Boolean         copied;

/*
 * The matched field show that the WEEdges incident to this face have
 * been matched with their neighbors incident to face->mate.
 */
Boolean         matched;

/*
 * The visible field is used while displaying the WEPolyhedron to
 * keep track of whether the face is visible to the user.
 */
Boolean         visible;

/*
 * How many sides does this face have?
 */
int             num_sides;

/*
 * The face and its mate are assigned to the same WEFaceClass.
 * In the case of an orbifold, a face may be its own mate, in which
 * case the WEFaceClass will have only one element.
 */
WEFaceClass     *f_class;

/*
 * The WEFaces are kept on a doubly-linked list.
 */
WEFace          *prev,
                *next;

```

```
};
```

```

/*
 * The Dirichlet domain's face pairings identify the WEVertices, WEEdges
 * and WEFaces into WEVertexClasses, WEEdgeClasses and WEFaceClasses.
 * Each equivalence class is assigned an index. (The indices are
 * assigned consecutively, beginning at zero.) The index is used
 * to define a "hue", which is the suggested hue for that cell.
 * The function index_to_hue() in index_to_hue.c assigns hues in such
 * a way that the low-numbered cells' hues are all easily distinguishable
 * from one another. This is especially important for the WEFaceClasses.
 * If there are a large number of faces we can't possibly hope that all
 * the hues will be easily distinguishable, but we do want the hues of
 * the largest faces (which are closest to the origin and have the lowest
 * indices) to be easily distinguishable.
 */

struct WEVertexClass
{
    /*
     * At present the WEVertexClasses are listed in arbitrary order,
     * but if necessary they could easily be sorted to provide
     * some control over their hues, as is done for WEFaceClasses.
     */
    int          index;
    double       hue;

    int          num_elements;

    /*
     * The total solid angle surrounding this vertex
     * (4pi for a manifold, 4pi/n for an orbifold).
     */
    double       solid_angle;

    /*
     * The "n" in the preceding 4pi/n is recorded as the singularity_order.
     * (For ideal vertices, n is set to zero.)
     */
    int          singularity_order;

    /*
     * Is this an ideal vertex class?
     */
    Boolean      ideal;

    /*
     * All the vertices in the vertex class should be the same distance from
     * the origin. Dirichlet_extras.c checks that the distances are indeed
     * approximately equal, and records their average here.
     */
    double       dist;

    /*
     * min_dist and max_dist are used locally in vertex_distances()
     * in Dirichlet_extras.c to check that the dist values of the
     * constituent vertices are consistent.
     */
    double       min_dist,
                max_dist;

    /*
     * belongs_to_region and is_3_ball are used locally in
     * compute_spine_radius() in Dirichlet_extras.c.
     * belongs_to_region keeps track of how various regions
     * have been united. is_3_ball records which such unified
     * regions are topologically 3-balls.
     */
    WEVertexClass *belongs_to_region;
    Boolean      is_3_ball;

    /*
     * The WEVertexClasses are kept on a doubly-linked list.
     */

```

```

    WEVertexClass    *prev,
                    *next;
};

struct WEEdgeClass
{
    /*
     * At present the WEEdgeClasses are listed in arbitrary order,
     * but if necessary they could easily be sorted to provide
     * some control over their hues, as is done for WEFaceClasses.
     */
    int              index;
    double           hue;

    int              num_elements;

    /*
     * The total dihedral angle surrounding this edge
     * (2pi for a manifold, 2pi/n for an orbifold).
     */
    double           dihedral_angle;

    /*
     * The "n" in the preceding 2pi/n is recorded as the singularity_order.
     */
    int              singularity_order;

    /*
     * All the edges in the edge class should be the same distance from
     * the origin. Dirichlet_extras.c checks that the distances are indeed
     * approximately equal, and records their average here.
     */
    double           dist_line_to_origin,
                    dist_edge_to_origin;

    /*
     * How long is the identified edge?
     * If it's infinite, the length is set to INFINITE_LENGTH.
     */
    double           length;

    /*
     * Performing the face identifications on the Dirichlet domain gives
     * a manifold or orbifold. The link of the midpoint of an edge will
     * be a 2-orbifold.
     */
    Orbifold2        link;

    /*
     * min_line_dist and max_line_dist are used locally in edge_distances()
     * in Dirichlet_extras.c to check that the dist_line_to_origin values
     * of the constituent edges are consistent.
     */
    double           min_line_dist,
                    max_line_dist;

    /*
     * min_length and max_length are used locally in edge_lengths()
     * in Dirichlet_extras.c to check that the length values
     * of the constituent edges are consistent.
     */
    double           min_length,
                    max_length;

    /*
     * removed is used locally in compute_spine_radius() in Dirichlet_extras.c
     * to keep track of which 2-cells in the dual spine have been removed.
     */
    Boolean          removed;

    /*
     * The WEEdgeClasses are kept on a doubly-linked list.
     */
};

```

```

    WEEdgeClass      *prev,
                    *next;
};

struct WEFaceClass
{
    /*
     * Indices are assigned to face classes in order of increasing
     * distance from the origin.  The closest face class gets index 0,
     * the next closest gets index 1, etc.  (The distance is actually the
     * distance from the origin to the plane containing the face, whether
     * or not the face happens to include the point where the plane is
     * closest to the origin.)
     *
     * Lemma.  A face and its mate are the same distance from the origin.
     *
     * Proof.  The face plane is midway between the origin and the
     * origin's image under the group_element.   $d(g^{-1}(\text{origin}), \text{origin})$ 
     * =  $d(\text{origin}, g(\text{origin}))$ .  Q.E.D.
     *
     * The function index_to_hue() in index_to_hue.c insures that
     * the largest faces have easily distinguishable colors.  For example,
     * a (37,1) Dehn surgery on the figure eight knot yields a Dirichlet
     * domain with a few large faces and many tiny ones.  We wouldn't want
     * to color the large faces with, say, twelve different shades of blue.
     * The index-to-hue conversion scheme spreads their hues evenly through
     * the spectrum.
     */

    int              index;
    double           hue;

    /*
     * Typically a WEFaceClass will have two elements, but if a face is
     * glued to itself (in an orbifold), then the WEFaceClass will have
     * only one element.
     */
    int              num_elements;

    /*
     * The distance from the face plane to the origin.  The point of
     * closest approach may or may not lie on the face itself.
     */
    double           dist;

    /*
     * Is the gluing orientation_reversing or orientation_preserving?
     */
    MatrixParity     parity;

    /*
     * The WEFaceClasses are kept on a doubly-linked list.
     */
    WEFaceClass      *prev,
                    *next;
};

struct WEPolyhedron
{
    int              num_vertices,
                    num_edges,
                    num_faces;

    int              num_finite_vertices,
                    num_ideal_vertices;

    int              num_vertex_classes,
                    num_edge_classes,
                    num_face_classes;

    int              num_finite_vertex_classes,
                    num_ideal_vertex_classes;
};

```



```

/*
 * Because matrices in  $O(3,1)$  tend to accumulate roundoff error, it's
 * hard to get a good bound on the accuracy of the computed volume.
 * Nevertheless, the kernel computes the best value it can, with the
 * hope that it will aid the user in recognizing manifolds defined
 * by a set of generators. (The volume of a manifold defined by
 * Dehn filling a Triangulation can be computed directly to great
 * accuracy, using the kernel's volume() function.)
 */
double approximate_volume;

/*
 * The inradius is the radius of the largest sphere (centered at the
 * basepoint) which can be inscribed in the Dirichlet domain.
 * The outradius is the radius of the smallest sphere (centered at the
 * basepoint) which can be circumscribed about the Dirichlet domain.
 * The outradius will be infinite for cusped manifolds, in which
 * case it's set to INFINITE_RADIUS.
 */
double inradius,
       outradius;

/*
 * spine_radius is the infimum of the radii (measured from the origin)
 * of all spines dual to the Dirichlet domain. compute_spine_radius()
 * in Dirichlet_extras.c shows that spine_radius equals the maximum
 * of dist_edge_to_origin over all edge classes. The spine_radius
 * plays an essential role in the length spectrum routines.
 *
 * Note: In practice compute_spine_radius() removes selected 2-cells
 * from the spine to reduce its radius, and thereby reduce the time
 * required to compute length spectra. Please see compute_spine_radius()
 * in Dirichlet_extras.c for details.
 */
double spine_radius;

/*
 * Each face pairing isometry is an element of  $SO(3,1)$ , so the inner
 * products of its i-th column with its j-th column should be -1 (if
 *  $i = j = 0$ ), +1 (if  $i = j \neq 0$ ) or 0 (if  $i \neq j$ ). The greatest
 * deviation from these values (over all faces) is recorded in the
 * deviation field.
 */
double deviation;

/*
 * The geometric Euler characteristic of the quotient orbifold (i.e. the
 * orbifold obtained by doing the face identifications) is computed as
 *
 * 
$$c[0] - c[1] + c[2] - c[3]$$

 *
 * where
 *
 *  $c[0]$  = the sum of the solid angles at the vertices divided by  $4\pi$ ,
 *
 *  $c[1]$  = the sum of the dihedral angles at the edges divided by  $2\pi$ ,
 *
 *  $c[2]$  = half the number of faces of the Dirichlet domain,
 *
 *  $c[3]$  = the number of 3-cells, which is always one.
 *
 * This corresponds to the definition of the Euler characteristic of an
 * orbifold, as explained in Chapter 5 of the 1991 version of Thurston's
 * notes. It should, in theory, always come out to zero. But since
 * we compute it using floating point approximations to the solid and
 * dihedral angles, it provides a measure of the numerical inaccuracies
 * in the computation.
 */
double geometric_Euler_characteristic;

/*
 * vertex_epsilon is used in the construction of the Dirichlet domain.
 * If the squared distance from a vertex to a hyperplane is within

```

```
* vertex_epsilon of zero, the vertex is assumed to lie on the hyperplane.
* If vertex_epsilon is too large, we won't be able to resolve the small
* faces which occur in high order Dehn fillings. If vertex_epsilon is
* too small, we'll get spurious Dirichlet plane intersections for
* manifolds with simple, symmetrical, non-general-position covering
* transformation groups.
*/
double          vertex_epsilon;

/*
* The following dummy nodes serve as the beginnings and ends of the
* doubly-linked lists of vertices, edges and faces.
*/
WEVertex        vertex_list_begin,
                vertex_list_end;
WEEdge          edge_list_begin,
                edge_list_end;
WEFace          face_list_begin,
                face_list_end;

/*
* The following dummy nodes serve as the beginnings and ends of the
* doubly-linked lists of vertex classes, edge classes and face classes.
*/
WEVertexClass   vertex_class_begin,
                vertex_class_end;
WEEdgeClass     edge_class_begin,
                edge_class_end;
WEFaceClass     face_class_begin,
                face_class_end;

};

#endif
```