

```

/*
 * triangulation.h
 *
 * This file defines the basic data structure for an ideal
 * triangulation. SnapPea's various modules communicate with each
 * other primarily by passing pointers to Triangulations.
 *
 * The Triangulation data structure consists of some global information
 * about the manifold (number of tetrahedra, number of cusps, etc.)
 * following by doubly linked lists of Tetrahedra, EdgeClasses, and Cusps.
 * As the triangulation varies dynamically (for example, during the
 * triangulation simplification algorithm) Tetrahedra, EdgeClasses and
 * Cusps may be easily inserted and deleted using the INSERT_BEFORE()
 * and REMOVE_NODE() macros found in kernel_typedefs.h. The Triangulation
 * data structure contains header and tailer nodes for each doubly linked
 * list, to avoid having to consider special cases while inserting and
 * deleting nodes.
 *
 * To keep the global structure of this file as clear as possible,
 * most of the local documentation appears elsewhere. The comment
 * next to each field says what .c file (if any) contains the
 * documentation for that field.
 *
 * Most fields are maintained globally. That is, you may assume they
 * contain correct values at all times, and you should update their
 * values if you change the triangulation. Fields which are used locally
 * within a single file or module are so indicated. They do not contain
 * correct values outside that module, and you need not maintain them.
 *
 * Note that SnapPea.h (the only header file common to the user interface
 * and the computational kernel) contains the opaque typedef
 *
 *     typedef struct Triangulation    Triangulation;
 *
 * This opaque typedef allows the user interface to declare and pass
 * pointers to Triangulations, without being able to access a
 * Triangulation's fields directly. This file provides the kernel with
 * the actual definition.
 *
 * The inclusion of lower-level data structures within higher-level ones
 * forces the following typedefs to be organized in a bottom-up fashion,
 * beginning with the least significant data structure (ComplexWithLog)
 * and working towards the most significant one (Triangulation). I
 * therefore recommend that you start reading this file at the bottom
 * and work your way up.
 */

#ifdef _triangulation_
#define _triangulation_

#include "SnapPea.h"
#include "kernel_typedefs.h"

/*
 * Forward declarations.
 */

typedef struct ComplexWithLog    ComplexWithLog;
typedef struct TetShape          TetShape;
typedef struct Tetrahedron       Tetrahedron;
typedef struct EdgeClass         EdgeClass;
typedef struct Cusp              Cusp;

/*
 * ComplexWithLog stores a complex edge parameter in both rectangular
 * and logarithmic form. That is, the log field is always the complex
 * logarithm of the rect field. The imaginary part of the log varies
 * continuously during Dehn filling, and is not restricted to any
 * particular branch of the logarithm.
 *
 * The edge parameter is always expressed relative to the
 * right_handed orientation of the tetrahedron.
 */

```

```

struct ComplexWithLog
{
    Complex      rect;
    Complex      log;
};

/*
 * TetShape stores the complex edge parameters at edges 0,1 and 2 of
 * a given Tetrahedron. (See edge_classes.c for the edge indexing scheme.)
 * Edges 5, 4 and 3 are opposite 0, 1 and 2, respectively, and therefore
 * have equal edge parameters. The edge parameters are recorded at the
 * next-to-the-last as well as the last iteration of Newton's method in
 * the hyperbolic structures module, to allow the estimation of errors
 * in various computed quantities (volume, etc.). Warning: the true error
 * is usually greater than the error between the penultimate and ultimate
 * iterations of Newton's method. That is, if you switch to a different
 * triangulation of the same manifold, you'll find the volume, etc. differs
 * by more than it did between the last two iterations of Newton's method.
 * The edge parameters at the next-to-the-last iteration are stored as
 * cwl[penultimate][], while those at the last iteration are cwl[ultimate][].
 *
 *
 * Note that the Tetrahedron structure (immediately below) contains pointers
 * to TetShapes, rather than the TetShapes themselves. The disadvantage
 * of this scheme is that the TetShapes must be allocated and deallocated
 * explicitly. The advantages are
 *
 * (1) Some functions which temporarily require large numbers of Tetrahedra
 * can get by with less memory if they don't require the TetShapes.
 * On a Mac, for example, the Tetrahedron structure itself requires
 * 242 bytes, while the TetShapes require an additional 576 bytes.
 * This difference can be significant. For example, the function which
 * computes an ideal triangulation for a partially filled multicusp
 * manifold will, when applied to a 100-Tetrahedron Triangulation,
 * temporarily create more than 3000 Tetrahedra. By omitting the
 * TetShapes, the memory requirement for these Tetrahedra drops
 * from 2.5 MB to 750 kB.
 *
 * (2) It's easy for the low-level retriangulation function (e.g.
 * the 2-3 and 3-2 moves) to determine whether the Tetrahedra
 * have shapes associated with them. If the pointers to TetShapes
 * are NULL, there are no shapes; otherwise there are.
 *
 * The TetShape corresponding to the complete (resp. Dehn filled) hyperbolic
 * structure is stored in the Tetrahedron data structure as tet->shape[complete]
 * (resp. tet->shape[filled]). By convention, TetShapes will be present iff
 * tet->solution_type[complete] and tet->solution_type[filled] are something
 * other than not_attempted.
 */

struct TetShape
{
    ComplexWithLog      cwl[2][3];
};

struct Tetrahedron
{
    Tetrahedron      *neighbor[4];          /* kernel_typedefs.h */
    Permutation      gluing[4];             /* kernel_typedefs.h */
    Cusp              *cusp[4];             /* the cusp containing each vertex */
    int               curve[2][2][4][4];    /* peripheral_curves.c */
    int               scratch_curve[2][2][2][4][4]; /* intersection_numbers.c (local) */
    EdgeClass         *edge_class[6];       /* edge_classes.c */
    Orientation       edge_orientation[6]; /* edge_classes.c */
    TetShape          *shape[2];            /* see TetShape and ComplexWithLog above */
    ShapeInversion     *shape_history[2];    /* kernel_typedefs.h */
    int               coordinate_system;     /* hyperbolic_structure.c (local) */
    int               index;                 /* hyperbolic_structure.c (local) */
    GeneratorStatus    generator_status[4]; /* choose_generators.c (local) */
    int               generator_index[4];    /* choose_generators.c (local) */
    GluingParity       generator_parity[4]; /* choose_generators.c (local) */
    Complex            corner[4];            /* choose_generators.c (local) */
};

```

```

    FaceIndex      generator_path;      /* choose_generators.c (local)      */
    VertexCrossSections *cross_section;  /* cusp_cross_section.c (local)    */
    double          tilt[4];             /* cusp_cross_section.c (local)    */
    CanonizeInfo    *canonize_info;      /* canonize_part_2.c (local)       */
    Tetrahedron     *image;              /* symmetry.h (local)              */
    Permutation     map;                 /* symmetry.h (local)              */
    Boolean         tet_on_curve;        /* dual_curves.c (local)           */
    Boolean         face_on_curve[4];    /* dual_curves.c (local)           */
    CuspNbhdPosition *cusp_nbhd_position; /* cusp_neighborhoods.c (local)    */
    EdgeIndex       parallel_edge;       /* normal_surfaces.h (local)       */
    int             num_squares,         /* normal_surfaces.h (local)       */
                  num_triangles[4];     /* normal_surfaces.h (local)       */

    Boolean         has_correct_orientation; /* normal_surface_splitting.c (local) */
    int             flag;                /* general purpose integer for local use as necessary */
    Extra           *extra;              /* general purpose pointer for local use as necessary */
                                   /* see Extra typedef in kernel_typedefs.h for details */

    Tetrahedron     *prev;              /* previous tetrahedron on doubly linked list */
    Tetrahedron     *next;              /* next tetrahedron on doubly linked list */
};

struct EdgeClass
{
    int             order;               /* number of incident edges of tetrahedra */
    Tetrahedron     *incident_tet;       /* one particular incident tetrahedron... */
    EdgeIndex       incident_edge_index; /* ...and the index of the incident edge */
    int             num_incident_generators; /* choose_generators.c (local) */
    Boolean         active_relation;      /* choose_generators.c (local) */
    Complex         *complex_edge_equation; /* gluing_equations.c (used locally) */
    double          *real_edge_equation_re, /* gluing_equations.c (used locally) */
                  *real_edge_equation_im; /* gluing_equations.c (used locally) */
    Complex         edge_angle_sum; /* used locally in hyperbolic structures module */
    int             index;               /* used locally for saving Triangulations to disk */
    double          intercusp_distance; /* cusp_neighborhoods.c (used locally) */
    EdgeClass       *prev;               /* previous EdgeClass on doubly linked list */
    EdgeClass       *next;               /* next EdgeClass on doubly linked list */
};

struct Cusp
{
    CuspTopology    topology;            /* torus_cusp or Klein_cusp */
    Boolean         is_complete;          /* is the cusp currently unfilled? */
    double          m,                   /* Dehn filling coefficient */
                  l;                     /* Dehn filling coefficient */
    Complex         holonomy[2][2];      /* holonomy.c */
    Complex         *complex_cusp_equation; /* gluing_equations.c (used locally) */
    double          *real_cusp_equation_re, /* gluing_equations.c (used locally) */
                  *real_cusp_equation_im; /* gluing_equations.c (used locally) */
    Complex         cusp_shape[2];       /* cusp_shapes.c */
    int             shape_precision[2];  /* cusp_shapes.c */
    int             index;               /* cusp number, as perceived by user */
                                   /* (numbering starts at zero) */
    double          displacement,         /* cusp_neighborhoods.c (used globally) */
                  displacement_exp,      /* cusp_neighborhoods.c (used globally) */
                  reach,                 /* cusp_neighborhoods.c (local) */
                  stopping_displacement; /* cusp_neighborhoods.c (local) */
    Cusp           *stopper_cusp;        /* cusp_neighborhoods.c (local) */
    Boolean         is_tied;              /* cusp_neighborhoods.c (local) */
    Complex         translation[2],       /* cusp_neighborhoods.c (local) */
                  scratch;               /* cusp_neighborhoods.c (local) */
    double          exp_min_d;            /* cusp_neighborhoods.c (local) */
    Tetrahedron     *basepoint_tet;      /* fundamental_group.c (semi-local) */
}

```

```

VertexIndex      basepoint_vertex;      /* fundamental_group.c (semi-local) */
Orientation      basepoint_orientation; /* fundamental_group.c (semi-local) */
int              intersection_number[2][2]; /* intersection_numbers.c (local) */
Boolean          is_finite;              /* finite_vertices.c (used locally) */
Cusp              *matching_cusp;         /* subdivide.c, finite_vertices.c,
                                         /* cover.c, normal_surface_splitting.c
                                         /* (used locally)
int              euler_characteristic;    /* cusps.c (local)
Cusp              *prev;                  /* previous Cusp on doubly linked list
Cusp              *next;                  /* next Cusp on doubly linked list
};

struct Triangulation
{
    char          *name;                  /* name of manifold
    int           num_tetrahedra;          /* number of tetrahedra
    SolutionType  solution_type[2];        /* complete and filled
    Orientability  orientability;          /* Orientability of manifold
    int           num_cusps,               /* total number of cusps
    num_or_cusps, /* number of orientable cusps
    num_nonor_cusps; /* number of nonorientable cusps
    int           num_generators;          /* choose_generators.c (local)
    Boolean       CS_value_is_known,       /* Chern_Simons.c
    CS_fudge_is_known; /* Chern_Simons.c
    double        CS_value[2],             /* Chern_Simons.c
    CS_fudge[2]; /* Chern_Simons.c
    double        max_reach,               /* cusp_neighborhoods.c (local)
    tie_group_reach, /* cusp_neighborhoods.c (local)
    volume;      /* cusp_neighborhoods.c (local)
    Tetrahedron  tet_list_begin, /* header node for doubly linked list of Tetrahedra
    /*
    tet_list_end; /* tailer node for doubly linked list of Tetrahedra
    /*
    EdgeClass    edge_list_begin, /* header node for doubly linked list of Edges
    /*
    edge_list_end; /* tailer node for doubly linked list of Edges
    /*
    Cusp          cusp_list_begin, /* header node for doubly linked list of Cusps
    /*
    cusp_list_end; /* tailer node for doubly linked list of Cusps
    /*
};

#endif

```