

```

/*
 * kernel_typedefs.h
 *
 * This file contains #defines, typedefs and enums common to
 * many parts of the SnapPea kernel (but hidden from the UI).
 * Typedefs for more complicated data structures are found in
 * separate files (e.g. triangulation.h).
 *
 * For C++ compatibility I've avoided definitions of the form
 *
 *     typedef enum
 *     {
 *         foo_up,
 *         foo_down
 *     } Foo;
 *
 * in favor of
 *
 *     typedef int Foo;
 *     enum
 *     {
 *         foo_up,
 *         foo_down
 *     };
 *
 * The problem with the former definition is that C++ insists on an
 * explicit typecast to assign an integer to a variable of type Foo,
 * so code like "for (foo = 0; foo < 2; foo++)" won't compile.
 */

#ifndef _kernel_typedefs_
#define _kernel_typedefs_

#include "SnapPea.h"

#define NEW_STRUCT(struct_type)      (struct_type *) my_malloc((size_t) sizeof(struct_type))
#define NEW_ARRAY(n, struct_type)   (struct_type *) my_malloc((size_t) (n) * sizeof(struct_type))

#define INSERT_BEFORE(new, old)      {
                                        (new)->next = (old); \
                                        (new)->prev = (old)->prev; \
                                        (new)->prev->next = (new); \
                                        (new)->next->prev = (new); \
                                    }

#define INSERT_AFTER(new, old)       {
                                        (new)->prev = (old); \
                                        (new)->next = (old)->next; \
                                        (new)->prev->next = (new); \
                                        (new)->next->prev = (new); \
                                    }

#define REMOVE_NODE(node)            {
                                        (node)->next->prev = (node)->prev; \
                                        (node)->prev->next = (node)->next; \
                                    }

#define ABS(x)    (((x) >= 0) ? (x) : -(x))

/*
 * Gluings of ideal tetrahedra.
 *
 * The neighbor and gluing fields of a Tetrahedron tell how the Tetrahedron
 * is glued to the other Tetrahedra in the Triangulation. The vertices of
 * each Tetrahedron in the Triangulation are implicitly indexed by the integers
 * {0,1,2,3}, and the faces are indexed according to the index of the opposite
 * vertex. The field tet->neighbor[i] contains a pointer to the Tetrahedron
 * which face i of Tetrahedron tet glues to. The field tet->gluing[i] describes
 * the gluing as a Permutation of the set {0,1,2,3}, using the following scheme
 * devised by Bill Thurston. If j {j = 0,1,2,3} does not equal i, then j is the
 * index of a vertex on face i of Tetrahedron tet, and the Permutation

```

```

* tet->gluing[i] applied to j gives the index of the vertex of Tetrahedron
* tet->neighbor[i] that vertex j glues to. If j equals i, then j is the index
* of the vertex of Tetrahedron tet opposite face i, and the Permutation
* tet->gluing[i] applied to j gives the index of the vertex of Tetrahedron
* tet->neighbor[i] opposite the image of face i; equivalently -- and more
* usefully -- j is the index of the given face on Tetrahedron tet, and the
* Permutation applied to j gives the index of its image face on
* tet->neighbor[i].
*
* Each gluing field contains a permutation on {0,1,2,3}, represented in a
* single byte as follows. The byte is conceptually divided into four two-bit
* fields. The two high-order bits contain the image of 3 under the
* permutation, the next two bits contain the image of 2, etc. So, for example,
* the permutation which takes 3210 to 2013 would be represented as 10000111.
* This representation is space-efficient, and also allows the definition of
* the following macro for evaluating permutations.
*/

typedef unsigned char    Permutation;

/*
* EVALUATE(p,n) is the image of n (n = 0,1,2,3) under the permutation p.
*/
#define EVALUATE(p,n)      ( ((p) >> 2*(n)) & 0x03 )

/*
* CREATE_PERMUTATION() is, roughly speaking, the inverse of EVALUATE().
* CREATE_PERMUTATION(a,pa,b,pb,c,pc,d,pd) creates the one-byte Permutation
* which takes a to pa, b to pb, c to pc and d to pd. For example,
* CREATE_PERMUTATION(1,1,3,2,0,3,2,0) defines the permutation 2013 (base 4),
* which equals 10000111 (binary) or 0x87 (hex). Symbolically this is
* the permutation {3->2, 2->0, 1->1, 0->3}.
*/
#define CREATE_PERMUTATION(a,pa,b,pb,c,pc,d,pd)      ((Permutation) (((pa) << 2*(a)) + ((pb) << 2*(b)) + ((pc) << 2*(c)) + ((pd) << 2*(d))))

/*
* For convenience, we #define the IDENTITY_PERMUTATION.
*/
#define IDENTITY_PERMUTATION      0xE4      /* = 11100100 = 3210 */

/*
* FLOW(,) is used in reconstructing simple closed curves from
* homological information. Specifically, let A be the number of times
* a curve intersects one side of a triangle, and B be the number of times
* it intersects a different side (of the same triangle). Then FLOW(A,B)
* is the number of strands of the curve passing from the first side to
* the second.
*/

#define FLOW(A,B) ( ((A<0)^(B<0)) ? \
                    (((A<0)^(A+B<0)) ? A : -B) : \
                    0 )

#define MAX(A,B)      ((A) > (B) ? (A) : (B))
#define MIN(A,B)      ((A) < (B) ? (A) : (B))

#define DET2(M)        ((M[0][0])*(M[1][1]) - (M[0][1])*(M[1][0]))

/* Some unix C libraries define PI in math.h, */
/* and complain about a second definition. */
#ifdef PI
#define PI              3.14159265358979323846
#endif
#define TWO_PI          6.28318530717958647693
#define FOUR_PI         12.56637061435917295385
#define PI_OVER_2       1.57079632679489661923
#define PI_OVER_3       1.04719755119659774615
#define THREE_PI_OVER_2 4.71238898038468985769
#define ROOT_3_OVER_2   0.86602540378443864676
#define ROOT_3          1.73205080756887729352

#define TRUE            1
#define FALSE           0

```

```

typedef signed char      VertexIndex,
                        EdgeIndex,
                        FaceIndex;

/*
 * The Orientation of a tetrahedron is determined by placing your hand
 * inside the tetrahedron with your wrist at face 0, your thumb at face 1,
 * your index finger at face 2 and your middle finger at face 3.  If this
 * is most comfortably accomplished with your right hand, then the
 * tetrahedron is right_handed.  Otherwise it's left_handed.
 *
 * Portions of the code assume that right_handed == 0 and
 * left_handed == 1, so please don't change them.
 */

typedef int Orientation;
enum
{
    right_handed = 0,
    left_handed = 1,
    unknown_orientation
};

typedef MatrixParity GluingParity;

/*
 * The constants complete and filled facilitate reference
 * to the shape of a Tetrahedron as part of the complete or
 * Dehn filled hyperbolic structure, respectively.
 */

typedef int FillingStatus;
enum
{
    complete,
    filled
};

/*
 * The constants initial and current are synonymous with complete
 * and filled, respectively.  They are used to refer to cusp shapes.
 * That is, cusp_shape[initial] is the cusp shape defined by the
 * complete hyperbolic structure, and cusp_shape[current] is the
 * cusp shape defined by the current Dehn filling.  In both cases
 * the cusp in question is complete (even though other cusps might
 * not be) which is why I decided to use the terms initial and current
 * instead of complete and filled.
 */

enum
{
    initial,
    current
};

/*
 * The constants ultimate and penultimate facilitate reference
 * to the approximate solutions at the ultimate and penultimate
 * iterations of Newton's method.
 */

typedef int Ultimateness;
enum
{
    ultimate,
    penultimate
};

/*
 * The ShapeInversion data structure records the event that
 * a Tetrahedron changes its shape, i.e. goes from having z.real >= 0

```

```

* to z.real < 0. This allows the Chern-Simons code (and potentially
* other, future additions to SnapPea) to work out the exact path
* of the Tetrahedron shape through the parameter space, up to isotopy.
*
* Each ShapeInversion records which of the three edge parameters
* (0, 1 or 2) passed through pi (mod 2 pi) as the Tetrahedron changed
* shape. The other two parameters will have passed through 0 (mod 2 pi).
*
* A stack of ShapeInversions represents the history of a shape. The
* stack is implemented as a NULL-terminated linked list. Two consecutive
* ShapeInversions with the same edge parameter passing through pi
* will cancel.
*
* Each Tetrahedron has two ShapeInversion stacks, one for the complete
* hyperbolic structure and the other for the filled hyperbolic structure.
* Both are computed relative to the ultimate hyperbolic structure.
*/

typedef struct ShapeInversion
{
    /*
     * Which edge parameter passed through pi (mod 2 pi) ?
     */
    EdgeIndex          wide_angle;

    /*
     * The next field points to the next ShapeInversion on the
     * linked list, or is NULL if there are no more.
     */
    struct ShapeInversion *next;
} ShapeInversion;

/*
* The constants M and L provide indices for 2-element arrays
* and 2 x 2 matrices which refer to peripheral curves.
*
* For example, the Tetrahedron data structure records a meridian
* as curve[M] and a longitude as curve[L].
*
* The file i/o routines assume M == 0 and L == 1, so please
* don't change them.
*/

typedef int PeripheralCurve;
enum
{
    M = 0,
    L = 1
};

/*
* The TraceDirection typedef is used to specify whether a curve
* should be traced forwards or backwards.
*/

typedef int TraceDirection;
enum
{
    trace_forwards,
    trace_backwards
};

/*
* The GeneratorStatus typedef specifies the existence and direction
* of a generator of a manifold's fundamental group. See choose_generators.c
* for details.
*/

typedef int GeneratorStatus;
enum
{

```

```

    unassigned_generator, /* the algorithm has not yet considered the face */
    outbound_generator,   /* the generator is directed outwards */
    inbound_generator,    /* the generator is directed inwards */
    not_a_generator       /* the face does not correspond to a generator */
};

/*
 * The VertexCrossSections data structure represents a cross section
 * of each ideal vertex of the Tetrahedron which owns it.
 * The vertex cross section at vertex v of Tetrahedron tet is a
 * triangle. The length of its edge incident to face f of tet is
 * stored as tet->cross_section->edge_length[v][f]. (The edge_length
 * is undefined when v == f.) Please see cusp_cross_sections.c for
 * more details.
 *
 * By convention,
 *
 * when no cusp cross sections are in place, the cross_section field
 * of each Tetrahedron is set to NULL, and
 *
 * when cusp cross sections are created, the routine that creates
 * them must allocate the VertexCrossSections structures.
 *
 * Thus, routines which modify a triangulation (e.g. the two_to_three()
 * and three_to_two() moves) know that they must keep track of cusp cross
 * sections if and only if the cross_section fields of the Tetrahedra are
 * not NULL.
 */

typedef struct
{
    double edge_length[4][4];
    Boolean has_been_set[4];
} VertexCrossSections;

/*
 * cusp_neighborhoods.c needs to maintain a consistent coordinate system
 * on each cusp cross section, so that horoballs etc. appear at consistent
 * positions even as the canonical Triangulation changes.
 * Each Tetrahedron's cusp_nbhd_position field keeps a pointer to a
 * struct CuspNbhdPosition while cusp neighborhoods are being maintained;
 * at all other times that pointer is set to NULL. This offers the same
 * two advantages described in triangulation.h for the TetShape pointer,
 * namely,
 *
 * (1) we save memory when CuspNbhdPositions aren't needed, and
 *
 * (2) the low-level retriangulation routines can tell whether
 * they need to maintain CuspNbhdPositions or not.
 *
 * The cusp cross sections intersect each ideal tetrahedron in four small
 * triangles. The CuspNbhdPosition structure records the positions of
 * each of the four triangles' three vertices. Actually, we keep track
 * of the left-handed and right-handed sheets separately, so we can work
 * with the double covers of Klein bottle cusps; that way we're always
 * working with a torus. The indices of x[h][v][f] are as follows:
 *
 * h = left_handed or right_handed tells the sheet we're on,
 * v = 0,1,2,3 tells the vertex
 * f = 0,1,2,3 (for f != v) tells the face opposite the
 * ideal vertex of interest
 *
 * Note: The fields x[h][v][v], which are not needed for storing
 * the corner coordinates of triangles, are pressed into service in
 * get_cusp_neighborhood_Ford_domain() to store the locations of the
 * Ford domain vertices. Cf. the FORD_VERTEX(h,v) macro below.
 *
 * The Boolean field in_use[h][v] records which x[h][v][v] fields
 * are actually in use. For a Klein bottle cusp, they all will be in use
 * (because we work with the double cover). For a torus cusp only half
 * will be in use (in an orientable manifold they'll be the ones on the
 * right_handed sheet, in a nonorientable manifold one sheet is chosen

```

```

*   arbitrarily).
*/

typedef struct
{
    Complex      x[2][4][4];
    Boolean      in_use[2][4];
} CuspNbhdPosition;

#define FORD_VERTEX(x,h,v)  x[h][v][v]

/*
*   The CanonizeInfo data structure records information used by
*   canonical_retriangulation() in canonize_part_2.c. It is almost local
*   to that file, but not quite. The low-level retriangulation functions
*   two_to_three() and one_to_four() in simplify_triangulation.c need to be
*   able to check whether CanonizeInfo is present, and if so transfer the
*   information correctly to the new Tetrahedra they create. Thus we make
*   the convention that the canonize_info field of each Tetrahedron is NULL
*   if no CanonizeInfo is present, and points to a CanonizeInfo structure if
*   such information is present.
*/

typedef int FaceStatus;
enum
{
    opaque_face,           /* The face lies in the 2-skeleton of the          */
                          /* canonical cell decomposition.                    */
    transparent_face,      /* The face lies in the interior of a 3-cell        */
                          /* in the canonical cell decomposition, but        */
                          /* has not yet been worked into the coned          */
                          /* polyhedron.                                       */
    inside_cone_face       /* The face lies in the interior of a 3-cell        */
                          /* in the canonical cell decomposition, and        */
                          /* also lies in the interior of the coned          */
                          /* polyhedron.                                       */
};

typedef struct
{
    FaceStatus  face_status[4];
    Boolean     part_of_coned_cell;
} CanonizeInfo;

/*
*   In the definition of a Tetrahedron, it's useful to include a
*   general purpose pointer which different kernel modules may use
*   to temporarily append different data structures to the Tetrahedron.
*   Even though different modules will append different temporary
*   data structures, we must have a single global declaration of the
*   pointer in triangulation.h. One solution would be to declare the
*   pointer as a pointer to a void, but this leads to a lot of messy
*   typecasting. Instead, we use the following opaque typedef of a
*   (globally nonexistent) struct extra. Each module may then define
*   struct extra as it sees fit. There is no need for the definition
*   of struct extra found in one source file to be consistent with
*   that of another source file, although obviously when a function in
*   one file calls a function in another the programmer must be absolutely
*   certain that the files do not make conflicting definitions of
*   struct extra.
*
*   As a guard against conflicts, we make the convention that the
*   Extra field in the Tetrahedron data structure is always set to NULL
*   when not in use. Any module that wants to use the fields first
*   checks whether it is NULL, and reports an error and exits if it isn't.
*/

typedef struct extra Extra;

/*
*   Normally one expects all the code to be compiled with the same set
*   of calling conventions. An exception arises when using C++

```

```
* in conjunction with ANSI library routines that require callback functions.
* Specifically, qsort() wants a comparison function declared using
* C calling conventions. Typically this requires that the C++ code
* declare the callback function as cdecl or _cdecl ("C declaration").
* If this doesn't work, try the following syntax (which will require
* something like "#define CDECL_END )" to provide the closing bracket).
*
*     extern "C"
*     {
*         int comp(const void *a, const void *b)
*         {
*             ...
*         }
*     }
*
* If all else fails, read the documentation for your C++ compiler,
* and contact Jeff Weeks (www.geometrygames.org/contact.html)
* if you have problems. If this gets to be a headache, I can
* replace the standard library's qsort() with a hard-coded qsort
* (such as the one on page 120 of K&R 2nd ed.).
*/

#ifdef __cplusplus

#define CDECL    _cdecl
// #define CDECL cdecl

#else

#define CDECL    /* We're not using C++, so CDECL may be empty. */

#endif

#endif
```