TP 3

Hashtables and Mesh Adjacency

The data structure `struct Mesh` which we have used in TP2 is kind of minimalistic and efficient in terms of memory footprint, but for some mesh operations it can reveal itself insufficient. In particular, there is no direct way to list the (at most[1]) three neighbouring triangles of a given triangle. Since this information is essential in many important algorithms acting on meshes, it is useful to build some adjacency tables in addition to the mesh structure itself. The goal of these exercise notes is to test and compare the benefit of hashtables in the construction of such tables.

Exercise 1. Implement a function

```
int edge_pos_in_tri(int v1, int v2, struct Triangle t);
```

which returns the position of the oriented edge $v1 \to v2$ in the triangle `t`. By convention, for a triangle whose vertices index list is `a`, `b`, `c`, the edge $a \to b$ has position 0, the edge $b \to c$ has position 1, and the edge $c \to a$ has position 2. If the edge $v1 \to v2$ is not part of `t`, then the function should return -1.

Exercise 2. Based on the previous, implement a function `tris_are_neighbors` which determines if two triangles are neighbours (i.e. if they share a common edge, with opposite orientations). The function shall return -1 if they are not, and otherwise the position (in the meaning of Exercise 1) of their common edge in the first triangle.

```
int tris_are_neighbors(int tri1, int tri2, const struct Mesh *m)
```

If a triangular mesh has $ntri$ triangles, then its adjacency table adj is an array of length $3 * ntri$ of integers which is formed in the following way: For each $0 \leq i \leq ntri - 1$, and for each $0 \leq j \leq 2$, $adj[3 * i + j)]$ contains the index of the triangle which is ajacent to the triangle of index i along its edge of index j. If such a triangle does not exist (i.e. if that edge is a boundary edge), then this is specified by letting $adj[3 * i + j)] = -1$.

Exercise 3. In this exercise you are asked to build a function `build_adjacency_table1` in the most naive way: Loop over all triangles in the mesh, and for each of them loop once more over all triangles of the mesh and check for adjancency using the function `tris_are_neighbors` of Exercise 2, and update the adjacency table accordingly. The function should initially allocate space for an array adj of `3 * m->ntri` integers, and fill it with -1. Upon completion the function should return the address of the adjacency array.

```
int *build_adjacency_table1(const struct Mesh *m)
```

The previous function has time complexity $O(ntri^2)$, because of the double loop. It turns out to be impractical for meshes typically used in applications (up to a few million triangles).

Two triangles being neighbours in the mesh if and only if they share a common edge, the construction of the adjacency table can be accelerated by first building a dictionnary of edges. A *key* of that dictionnary is a pair (v_1, v_2), representing an oriented edge, and the corresponding *value* is the index of the triangle which contains it.

Exercise 4. Following the lines above, and using the implementation already provided in `hash_tables.h` and `hash_tables.c` for generic hash tables in C, implement a function `build_edge_table1` which initialises and then fill a hash table whose keys are of type

[1]In the following we shall only consider meshes for which each edge can belong to at most two triangles, with opposite orientation in each of them.

```c
struct Edge {
    int v1;
    int v2;
};
```

and whose values are of type `int`. The function prototype is

```c
struct HashTable *build_edge_table1(const struct Mesh *m);
```

The function should first allocate memory for an hash table, and eventually return its address once its has been filled with the mesh data.

We are now ready for a second implementation of the construction of the adjacency table. For that purpose, once the edge hash table is built it suffices to loop over all triangles of the mesh, and then for each of its three edges look-up for the opposing edge (i.e. with vertex order reversed) in the hash table, and potentially update the adjacency table.

Exercise 5. Code a function `build_adjacency_table2`, which has the same inputs and outputs as the function `build_adjacency_table1` in Exercise 3, but based on the new strategy proposed above.

```c
int *build_adjacency_table2(const struct Mesh *m);
```

The function should create, use, and then dispose the edge hash table as part of its implementation.

Exercise 6. Here we shall implement a third (and optimal) strategy to build the adjacency table, using an ad-hoc (rather than generic) hash table based on chaining and taking advantage of the fact that the map $(v_1, v_2) \mapsto v_1$ is a good candidate hash function if not too much triangles share a common vertex (a situation which is common an considered a quality for meshes). For that purpose we define the structure

```c
struct EdgeTable {
    int *head;
    int *next;
};
```

The field `head` refers to an (allocated) array of *nvert* (the number of vertices of the mesh) integers, while `next` (of size $3*ntri$, the number of oriented edges in the mesh) will allow to perform the chaining. As previosly stated, the hash function associates to each oriented edge its first vertex index. When inserting an edge, say $v1 \to v_2$, one first look in the array $head[v_1]$: if it is empty (encoded by the value -1), then one encode there that edge (by convention the edge number j $(0 \leq j \leq 2)$ of triangle i $(0 \leq i \leq ntri - 1)$ is encoded by $3 * i + j$). If it is not empty, then one pushes the code of the edge as in a stack: the code in $head[v1]$ is first copied into a temporary variable (say tmp), then $head[v1]$ is updated with the value $3 * i + j$ and finally (this is the chaining) we encode $next[3 * i + j] = tmp$. You can then deduce (after a bit of thinking!) what is the process for a look-up in that edge table. That will bring you to implement the following functions:

```c
void edge_table_initialize(struct EdgeTable *et, int nvert, int ntri);

void edge_table_dispose(struct EdgeTable *et);

void edge_table_insert(int v1, int edge_code, struct EdgeTable *et);

int  edge_table_find(int v1, int v2, const struct EdgeTable *et,
                                          const struct Mesh *m);

struct EdgeTable *build_edge_table3(const struct Mesh *m);

int *build_adjacency_table3(const struct Mesh *m);
```

Exercise 7. Test the timings for the above three methods of construction of the adjacency table in a separate `tp3.c`. The corresponding exectuable should take the mesh input filename as a command line argument. Sample data is provided in the `data` directory in the TP archive, and routines to read them are provided in the `include` and `src` directories. Note that the first ($O(ntri^2)$) method will be too slow for the `bugatti.obj` mesh, you should exclude that method for that mesh.