Project 1 Part II

1.1 Overview

You should design basic data structures and classes for computer graphics.

- Part I: matrices and vectors
- Part II: mesh data structures

1.2 Required Functionality: Part II

Here are the elements that your program must have.

- **Working halfedge data structure.** We provide a way to load a simple obj file and obj mesh files. Up to now, obj files can be inspected using the free **Meshlab** tool. Object files use a "shared vertex" representation. In this representation, vertex positions are stored directly, but faces use indices to reference their vertices. In mesh_t::load(), the vertices and faces are first stored in a std::vector. If the mesh has texture coordinates, they are loaded as well. Obj files can also be opened in a text editor, and it helps to make a drawing annotated with halfedges and their ids when stuck. Furthermore, when compiled in debug mode, mesh_t::dbgdump() will output the entire mesh

- **Task 1:** If you are unfamiliar with the C++ Standard Template Library (STL), have a look at the std::vector and std::list data types. Also, have a look at std::unordered_map, which uses a hash to store key, value pairs.

- **Task 2:** Familiarize yourself with the class definitions in mesh.h

- **Task 3:** In mesh.cpp, follow the TODO tags in the comments to implement the halfedge data structure. The idea is to iterate all facets and break them into "full" edges. Since an adjacent face uses the same edge, we use a hash (std::unordered_map) to retrieve relevant information that was previously processed.

    - **Implement the edge_t datastructure in mesh.cpp including the hash**. An edge is defined by vertex indices v1, v2. By convention, the smaller index is stored in v1 and the larger in v2 to make edge (v1,v2) equivalent to edge (v2,v1). As a hash, implement a simple mapping from the tuple (v1,v2) to a single value of type std::size_t.

- **Task 4:** Look at mesh_t::build_mesh(…). This is where the halfedge data structure is built.

  a) Write code to add all vertices to the mesh. Keep a list of pointers to the vertices. Add position and (if available) texture coordinate information to each vertex.

  b) Iterate all facets. For each facet, generate a new facet in the mesh. Break the facet into edges (and then into halfedges) and keep track of the first halfedge, the current halfedge, and the previous halfedge generated in this fashion.

  c) Iterate all edges of the face. For each edge, try to locate the edge in the edge_list. If the edge is not yet in the list, create two halfedges (an opposite pair) and update the following pointers of the halfedge:
  - opposite(), opposite()->opposite()
  - vertex(), opposite()->vertex
  - facet()

  Also, update the halfedge pointers in the vertex_list. Finally, store the opposite halfedge pointer in the edge_list for later use. Keep track of first, current, and previous halfedge of this facet.

  If the edge is already in the list, just keep track of the first, current, and previous halfedge of this facet.

  Then, update the missing pointers to the facet and use first, prev to update the next and prev pointers of the halfedge

  CONVENTIONS. A halfedge from v1 to v2 points to v2 (that is, halfedge->vertex() = pointer to v2). A vertex points to an outgoing halfedge, i.e., vertex->halfedge->opposite->vertex = same vertex.

- **Task 5:** Now you should have an almost working halfedge data structure. What is missing is proper treatment of boundaries. In particular, next and previous pointers of boundary halfedges are missing. Iterate all halfedges and fix this (the mesh "boundary object") is a simple mesh you can use for this purpose.

  a) Start by implementing next_around_vertex() and prev_around_vertex() in halfedge_t. Return nullptr if this does not exist.

  b) In mesh_t::build_mesh() add the boundary treatment.

- **Task 6: Implement the missing helper functions:**
  - vertex_t::degree() – computes the valence of the vertex
  - vertex_t::on_border() – true iff vertex on a border
  - halfedge_t::on_border() – true iff halfedge on a border
  - facet_t::degree() – computes the number of vertices on this facet
  - facet_t::on_border() – true iff facet on a border
  - facet_t::is_triangle() – true iff facet is a triangle

- **Task 7: Validate your code!** Your implementation **must** pass check_mesh() and the results of the tests in main.cpp must be reasonable. The provided obj files should all load.

- **Task 8: Compute per-vertex normals.** Add code to build_mesh(). First, compute area-weighted facet_t normal (each facet_t provides storage for a normal.) For this task, you may assume that every facet is a triangle, but you must output a warning if that is not the case! Do not normalize the facet normal, yet. In a second step, for each vertex, accumulate the normals of adjacent facets and normalize the vertex normal. Then, normalize the facets' normals as well.