Assignment 3 - Mesh Simplification

In this assignment you will build a half edge data structures for triangle meshes loaded from `.obj` files and simplify these meshes using the quadratic error scheme described in class Garland and Heckbert 1997.

There is a framework of provided code to get you started. It loads a mesh with `trimesh`, just like the previous assignment, and sets up a viewing canvas and user interface. Notice that there are keyboard controls associated with the UI buttons and check boxes (e.g., N and T for walking to next and twin half edge). You should look through the four provided python files to familiarize yourself with the classes, members, methods, and the comments left in the code (see likewise the descriptions below).

The provided code also includes a few geometry shaders (not covered in this course) to help with certain aspects of the visualization. For your optional edification, notice that the glsl shaders to draw the mesh include a geometry shader that computes the face normals for flat shading. There is also glsl shaders to draw a half edge with "fancy lines" using ray tracing, and a module for rendering text (e.g., triangle and vertex indices to help you debug your code).

You are also provided in a data folder a number of sample triangle meshes that will be useful for testing your program. All meshes are triangular manifolds without boundary.

Getting Started

You can use the same python and environment that you set up in the previous assignment as this assignment has most of the same module requirements. New for this assignment, you must install the SortedContainers module:

```
python -m pip install sortedcontainers
```

Provided Code

Download the provided code from MyCourses. It is a working program that will load one of the meshes and will draw it. Beyond that, many of the UI controls will do nothing or cause the program to crash until you implement the associated objectives. The following classes and functions of interest are included in the provided `py` files:

HalfEdge

The half-edge class is simple, and provided for you. Half-edges can be drawn to help you verify that your program is working. For your convenience, it also contains a tail method to get the starting vertex for this half-edge.

Face

The face class contains one of its half edges, and its index within the Nx3 numpy array of face data (this is `faces` of the `SimplificationViewer` class, which is associated with the index buffer object `ibo`). The other

members of the class are used to cache information for the `draw_debug` method, which will draw the index of each face on the face.

Vertex

The vertex class is simply a container to hold the position of the vertex, and information for the quadric error metric. It holds a reference to one of the half-edges that has this vertex as head. It also has a method for drawing vertex indices to potentially help with debugging.

EdgeCollapseData

This class is a container for edge specific information common to the two half edges, and specifically the cost for collapsing the edge and its optimal vertex position. It has members for storing the quadric error metric for collapsing the edge, the optimal vertex position, and the error. It also points back to one of its half edges. The class implements the `less than` and `equality` operations so that you can use it in a sorted list.

build_heds()

The `heds.py` file contains a function for building the half edge data structure, and this is where you start (i.e., the first objective). This function build a list of half-edges and a list of `Faces` that will be returned to permit further processing within the viewer class.

SimplificationViewer

This a larger class because it handles viewer widget setup, mesh loading, shader loading, viewing and projection setup, UI methods (e.g., mouse interaction), and methods for the UI to call (e.g., next half edge, twin half edge, LOD control, etc.). You will need to do most of your work near the bottom of the file (i.e., the collapse method). That said, there are some important details to keep in mind so that your code works as intended with the viewing framework.

- The LOD slider slowly grows in range as you collapse edges, letting you view different levels of simplification once you've completed objective 3.
- Each collapse creates a new `Vertex` and adds it to the end of the `vert_objs` list, while also adding the vertex position into the vertex buffer object (for drawing). This permits easy rendering of different LODs where it is only the face indices that need to be updated to reference the old vertices or the new vertex.
- Each collapse removes 2 faces, and you must swap entries in the `faces` and `face_obj` members to put these faces *at the end* of the list of faces of the current LOD level. That is, your first collapse "moves" your removed faces to the end of the list, and your second collapse will move the next two removed faces to the next two spots (i.e., 4th last and 3rd last). This is so that for any given LOD you simply need to draw the number of faces of the mesh at that level (i.e., the original number of faces minus two times the `current_LOD`).
- Finally, note that changing the LOD slider will cycle through the list of changes, either undoing or redoing the collapse until arriving at the desired LOD. This is not changing your half edge data structure, which will remain in a state where it represents the coarsest mesh after a sequence of collapses. Instead it simply changes the vertex indicies for the set of faces affected by the collapse.

Objectives

1. **(2 marks) Half Edge Data Structure.**
    - Write code that builds the half edge data structure. You will probably want to keep track of the half edges you create as you make them so that you can easily connect each half edge with its twin (refer to the discussion from class, and consider using a dictionary of 2-tuples).
    - Test that your structure is correct by using the UI to walk around the data structure (next, twin, next, etc.).
2. **(2 marks) Edge Collapse.**
    - Implement code to collapse the current half edge when `C` or the corresponding button is pressed. Given you've not yet implemented the quadric error metric yet, you can test by collapsing to the edge midpoint.
    - See the the provided code creates the new vertex, using `he.next.twin` as its associated half edge, which is returned at the end of the method.
    - Recall (from above) that you must do swaps to edit the `faces` and `face_objs` members to ensure the removed faces are at the *end* of the current LOD's faces.
    - Test that you can collapse edges and still correctly walk around the mesh.
3. **(2 marks) Undo / Redo**
    - Create the correct `CollapseRecord` to permit the face indices of affected faces to be done or undone to change between levels of detail.
    - Note that `CollapseRecord` contains `Face` objects rather than integer face indices. This is because the face indices will change as you go through the simplification process (i.e., always moving the collapsed faces to the end of the current LOD's faces).
    - Note that when collapse is called it sets the LOD to be the current `max_LOD`. This ensures the state of the `face` and `face_objs` members are representing the coarsest level so as to match the current state of the half edge data structure.
4. **(2 marks) Avoid Topological Problems.**
    - Write code to check if a collapse will cause topological problems using the following heuristics. That is, check if the 1-rings of the edge vertices have more than 2 vertices in common.
    - Note that a mesh will not simplify beyond a tetrahedron because the methods that call Collapse check that there are more than 6 edges in the mesh before calling collapse.
    - If you do not maintain the half edge data structure in a valid state, then you will quickly encounter problems on subsequent edge collapse operations!
5. **(3 marks) Quadric Error Minimum.**
    - Write code to compute the quadric error matrix for faces, vertices, and complete the constructor for `EdgeCollapseData` which solves the quadratic equation for the given edge. Use your preferred technique for handling rank deficient matrices.

6. **(2 marks) Mesh Simplification.**

    - Once you've implemented the constructor for `EdgeCollapseData`, the `compute_edge_collapse_costs()` method will insert the edges into the `sorted_edge_list`. You will be able to jump to the best half edge with the `J` key or associated button, and likewise collapse this best edge (without jumping) with `B`.

    - You will need to make sure your code in the `collapse` function updates the `sorted_edge_list` correctly at each collapse. EdgeCollapseData of edges adjacent to the collapsing edge must be removed from the sorted list, and those edges adjacent to the new vertex will need to have new EdgeCollapseData positions and costs computed and re-added to the `sorted_edge_list`.

    - Note that the quadric error of the new vertex is simply the sum of the Q for the adjacent vertices. That is, it does not contain any regularization, and is not reconstructed based on the planes of the new faces surrounding the vertex. That is, the Q at this vertex reflects the desired position given the shape of the original mesh as opposed to the shape at the current level of detail.

    - With this objective complete, you should be able to collapse all to simplify meshes to a tetrahedron. You can view all levels of detail with the LOD slider, and can likewise couple a heuristic scaling to show the mesh at smaller sizes for smaller meshes. You might look at the code and notice the scaling heuristic used to link scale with the LOD.

Finished?

Great! Submit your python implementation and log file, and optionally a readme file as a `zip` archive to MyCourses (only use `zip` format). Use the comments at the top of the files you submit to list your name and student number, and likewise add your name and student number to the title of the main application window. Recall the readme can be used to request a late penalty waiver following the mechanism described in the first lecture slides.

Note that you are encouraged to discuss assignments with your classmates, but not to the point of sharing code and answers. All code and written answers must be your own. Please see the course outline and the fine print in the slides of the first lecture.