[CSCI 2240] Assignment 2: Geometry Processing (mesh)

Released: 2/21/2020
Due: 3/6/2020 @ 11:59pm EST

In this assignment, you will gain experience with low-level manipulations of triangular meshes by implementing several geometry processing functions: subdivision, simplification, and one other method of your choice. Subdivision and simplification are, in a sense, opposites: one increases the resolution of a mesh while the other decreases it. The particular subdivision algorithm you will implement is called Loop subdivision; the particular simplification algorithm is called quadric error simplification.

Relevant Reading

- The lecture slides!
- Surface Simplification Using Quadric Error Metrics
- A Remeshing Approach to Multiresolution Modeling
- Bilateral Mesh Denoising

Requirements

This assignment is out of **100 points.**

To get full credit (i.e. a grade of A), you must implement the following:

- Loop subdivision
    - Correct subdivided triangle topology **(10 points)**
    - Correct subdivided vertex positions **(10 points)**
    - Runs in linear time in size of mesh **(5 points)**
- Simplification using quadric error metrics
    - Correct edge collapse topology **(10 points)**
    - Correct computation of optimal collapsed vertex position **(10 points)**
    - Full algorithm for simplification by iterative edge collapse **(10 points)**
    - Efficiency: constant-time edge collapse, priority queue for managing candidate edges **(5 points)**
        - Since you need to be able to update the costs of candidate edges in the priority queue, an std::set may be a better choice of data structure than std::priority_queue.
        - We will accept anything implementation that is better than (or equal to) O(n^2) in the worst case. However, it is highly recommended for you to have a O(nlogn) implementation.

    - NOTE: You only need to support contraction of vertex pairs connected by an edge (e.g. edge collapse), not arbitrary vertex pairs as is done in the paper.
- One other geometry processing function (**25 points** for correctness; **5 points** for efficiency)
    - Isotropic remeshing: make a mesh more regular by iteratively applying a set of local mesh operations. See Section 4 of this paper, as well as the lecture slides. More precisely, you should:
        - Compute the mean edge length L of the input.
        - Split all edges that are longer than 4L/3.
        - Collapse all edges that are shorter than 4L/5.
        - Flip all edges that decrease the total deviation from degree 6.
        - Compute the centroids for all the vertices.
        - Move each vertex in the tangent direction toward its centroid.
    - Bilateral mesh denoising: smooth noisy meshes while preserving important features. See Section 2 of this paper, as well as the lecture slides.
        - This method makes use of mesh vertex normals. As described in the paper, a common way to compute a vertex normal is to take the average of the normals of all faces adjacent to a vertex. The mesh data structure you've already built for the first two parts of this assignment should make this easy.
        - To test this method, you'll need to have some noisy input meshes. The easiest way to do this is to add synthetic noise to a clean mesh you already have. A simple technique is to add a random displacement to every vertex along its normal direction.
        - You don't need to deal with volume preservation, though you can if you want---Sections 3.1-3.2 of this paper describe a simple volume preservation technique.
    - Something else! Ask the instructor if you're uncertain whether what you have in mind is too complex / not complex enough.

We will run automated test scripts that generate results from test meshes, so it is important that you **follow the standard command-line interface**. Namely, each mesh operation should be called as

```
./Mesh-Stencil --input-obj-name --output-obj-name --method-name --args1
--args2 --args3 --args4
```

Possible method names are "subdivide | simplify | remesh | denoise", make sure that your program gracefully exits for methods which you do not implement. Please document extra options if you implement something else. Args1-args4 are respective parameters for the method. Details are documented in the stencil code. If you hardcode any of these (which will mostly likely happen for remesh and denoise), document them as well.

Though we will be running test scripts, you should also submit some results generated by your code **(5 points)**.
For each of the geometry processing functions you implement, submit at least one (before, after) image pair showing a mesh before and after being processed by that function.
You might want to use the 'after' mesh from simplification as the 'before' mesh for subdivision. Is the subdivided mesh the same as the original mesh before simplification?

You must also submit a plaintext README file **(5 points)**.
This file should describe how to run your code to reproduce the results shown in your submitted images. This must not require a modification to the source code itself--use command-line arguments (or a GUI, if you like) to change program behavior.
Your README should describe the data structures you used to achieve efficient asymptotic running times for your geometry processing functions.

Important note: You do ***not*** need to support meshes with boundaries for any part of this assignment! That is to say, you can always assume that every edge has two adjacent faces--so you don't need to worry about any special-casing for edges on the boundary. All of the meshes provided in the starter code (see below) should satisfy this property.

However, you can choose to support meshes with boundaries, as **extra credit (10 points).**
If you choose to do so, document that as well, and provide a few meshes / results that demonstrate this functionality.

Resources

The starter code for this assignment can be found here.
This starter project is quite barebones: it provides code to load .obj files into a simple mesh representation (list of vertices and list of faces), as well as code to save that representation back to an .obj file. You'll need to implement everything else: the mesh data structure you'll use to support efficient local edits, etc.

The starter project also contains some .obj files you can use to test your code. There are also tons of 3D model files available on the internet. A small amount of extra credit is available for finding and sharing a new (closed, manifold) .obj file with the class on Piazza **(2 points)**. To get extra credit, you'll also need post a before/after image pair of your code operating on this mesh (any operation is fine).

To inspect and interact with input/output meshes, it's worth getting familiar with some sort of 3D model viewing/editing software. One good option is MeshLab, which is free, lightweight, and provides a ton of useful functionality, including easy conversions between many different 3D file formats. If you're already familiar another 3D package such as Maya or Blender, those are perfectly fine, too.

Implementation & Debugging Tips

- This assignment requires that local mesh operations (e.g. edge splits, edge collapses) run in constant time. Before you start coding, spend some time planning out in detail what data structures you will use to make this possible.
    - You can augment the standard triangle adjacency list data structure (i.e. for each face, which vertices does it connect) with a vertex adjacency list (i.e. for each vertex, which faces is it incident to). **Keep in mind**: since this data structure must support constant-time addition and deletion of mesh elements, you won't be able to use linear indices to refer to vertices and faces (since those would need to be globally re-numbered every time an element is added/removed). Use some other form of unique ID to refer to mesh elements.
    - You can also try implementing a half-edge mesh data structure. This is more difficult to get right, but it can be more elegant than the augmented adjacency list and make some operations easier.
- Each local mesh operation requires careful updates to your mesh data structure. To make sure you get these right, it's helpful to plan by drawing diagrams of the operations being applied to simple 2D meshes, with each affected vertex/face clearly labeled.
- Not every edge operation can be applied to every edge--in some cases, edge operations can lead to a manifold mesh becoming non-manifold. You should spend some time thinking about which of the three local edge operations can have this effect, and under what circumstances. Diagramming some simple 2D meshes can be also be useful here. Your code should avoid performing operations that would make a manifold mesh become non-manifold!
- You'll want to start with simple meshes for testing/debugging your implementation. To help with this, the starter code repo contains an .obj file for an icosahedron, plus an .obj file for the result of applying each of the local mesh operations (edge flip, edge split, edge collapse) to the first edge in that mesh (i.e. the edge between the first two vertices in the .obj file).
- As always, implement features one by one and debug as you go. A good strategy: verify each of your local mesh operations (flip, split, collapse) is working correctly before moving on to implementing any global operations.
- **REMINDER**: Your code will run *much* faster if you compile in Release mode ;)

Submission Instructions

Submit your assignment by running `cs224_handin mesh` from a CS department terminal. You should run the handin script from a directory containing all the files you wish to submit. This directory must include a file named 'README' for the submission to be accepted.

Credits

Parts of this assignment are based on an assignment developed for CMU 15-462 by Keenan Crane and Kayvon Fatahalian.