mesh

Notes

This project is implemented in Rust. I wasn't able to find a working rust install anywhere in the department file system (in theory, /course/cs1680/contrib/ should have one, but the permissions are wrong). A docker container which can run the project can be found here I have also included a binary in the `bin/` directory. This binary runs on the department machines and will probably run on any reasonably recent version of linux.

If extra arguments are supplied for a command, that will lead to an error. For instance, `./mesh inp.obj out.obj subdivide 1 0.38838 3 4` would error because only one argument is expected. `./mesh inp.obj out.obj subdivide 1` would work.

Data Structure

I implemented a half edge data structure. Rather than allocating each half edge/vertex/edge/face individually, I store them on vectors and mark each vertex/edge/face invalid when it is removed. This is $O(1)$ removal and is effectively the same as allocating using a slab allocator. It should be more efficient, because memory access can make better use of the cache. To add an element, I just append to the appropriate vector which is amortized $O(1)$. If a large number of collapses are run, and then further operations were conducted, it might improve efficiency and memory usage to purge invalid items. None of the operations here are probably substantially improved by this, so I didn't bother; however, this could be done by rebuilding the data structure (invalidating the meaning of any stored indexes). Periodically rebuilding the data structure still leads to amortized $O(1)$ insert and $O(1)$ removal because building is $O(n)$. Building uses a hashmap to keep track of which vertices and half edges have already been inserted, and hashmaps are amortized $O(1)$ insert and $O(1)$ look up.

Features

- subdivide
  - $O(n)$
  - I think my implementation is pretty fast because of data structure choices, but I don't really know
- simplify
  - $O(n \log n)$
  - Uses priority queue (binary heap) and a separate vector which stores the current "iteration" of a given edge. If an edge is popped with iteration less than the current iteration, the edge is discarded. Each time the cost for an edge is updated, the iteration is incremented and a new element is added to the heap which stores the current iteration

and edge index. Thus, "removing" from the heap is $O(1)$ (this doesn't included the cost to pop from the heap, which is still $O(\log n)$)

- denoise
  - $O(n)$
  - Subjectively pretty slow
- noise
  - $O(n)$
  - Adds noise to a mesh for testing denoise
  - Adds random value drawn from normal distribution with standard deviation given by command line argument.
  - For example: `./mesh meshes/bunny.obj noise_bunny.obj noise 0.01`
- remesh
  - Additional function/extra credit
  - $O(n)$
  - Some times edge collapse and edge flip can lead to worse results depending on number of iterations and the smoothing weight from what I have seen, so I added optional arguments to disable these. To disable edge collapse and/or flip, run with the optional arguments –no-collapse and/or –no-flip

Results

All results are in the `results/` directory. Each result has its own subdirectory. The name of the directory corresponds to the output name (but without obj). Images are also included of the input and output in each result directory. An image of the original mesh is also included when the operation is the second operation applied (for instance, simp_subdiv_cow also has a picture of the original cow).

input	output	command	args
cow.obj	subdiv_cow.obj	subdivide	2
subdiv_cow.obj	simp_subdiv_cow.obj	simplify	87060
cow.obj	simp_cow.obj	simplify	5000
bunny.obj	simp_bunny.obj	simplify	15000
bunny.obj	noise_bunny.obj	noise	0.01
noise_bunny.obj	denoise_bunny.obj	denoise	5 0.02 0.02 2
peter.obj	remesh_peter.obj	remesh	10 0.5