Assignment 3: Mesh Editing

CMSC 23700: University of Chicago
Due: Sat February 14th at 11:59PM

Introduction

In this assignment, you are given the skeleton of a halfedge data structure-based mesh processing library in the `meshing` folder. Your task will be to complete the implementation of the basic data structure, along with two basic mesh editing functions for the `meshing` module. We provide you with the following import, export, and viewer functions in the starting code: `PolygonSoup.from_obj()`, `Mesh.export_soup()`, `Mesh.export_obj()`, `Mesh.view_basic()`, and `Mesh.view_with_topology()`. See an example of how to call these functions below. We also provide you with more code examples for visualization/testing in the `starter.py` file.

```python
from meshing.io import PolygonSoup
from meshing.mesh import Mesh
soup = PolygonSoup.from_obj("bunny.obj")
mesh = Mesh(soup.vertices, soup.indices)
# View mesh in interactive GUI
mesh.view_basic()
# TODO: This will only work AFTER you complete P1 and P2
vertices, faces, edges = mesh.export_soup()
# TODO: This will only work AFTER you complete P1 and P2
mesh.export_obj("example.obj")
```

Note: You will not need to worry about checking boundaries for this assignment.

Required packages

- python >= 3.9
- numpy >= 1.21
- **polyscope >= 2.4.0**

Submission Instructions

Your submission should contain **only** the following files: `__init__.py`, `edit.py`, `io.py`, `mesh.py`, `primitive.py`, `topology.py`, `p7.obj`, and `p7_custom.obj`, and `documentation.pdf`. (These Python files are all from the `meshing` folder.) Do not submit additional Python files other than the ones given. Do not zip/tar your submission. Do not import external packages other than the ones listed above.

If you are planning to use your late bank but are checking your assignment output using the autograder - put the following text in your `documentation.pdf` during submission testing: *Please do not grade this submission yet, I am using my late bank.*

Documentation

You should thoroughly and thoughtfully document your work in this project, **both in code** and in a `documentation.pdf` file you submit alongside the code and meshes.

- In `documentation.pdf`, document your high-level progress through the assignment: your initial planned approaches, any issues you ran into during implementation, and how you solved them.
- In `documentation.pdf`, draw and include an annotated diagram or sketch you use to derive your solution. Graphics algorithms are especially suited for the use of diagrams to aid understanding and debugging. **Please create and include at least one such diagram in `documentation.pdf` as you go through the assignment. Drawing by hand is fine. Diagrams from conversations with a TA/instructor are fine, but must be your own work.** Here are some examples of things you may want to visualize/sketch out for this assignment:
  - How you implement the P2 functions for traversing the halfedge structure: the linkages between the primitives and how this gives rise to the adjacency neighborhoods and queries desired
  - How you implement edge collapses: the cases you consider, the neighborhood of primitives involved, which primitives to delete and which to repair references to/from

  There is no hard standard for how detailed such a diagram must be, but they should be sufficiently illuminating in order to have helped you implement the subproblem in question.
- In `documentation.pdf`, cite all resources, online articles, Q&A threads, etc. you use in your code. In line with the Generative AI policy on the course website, you must also cite any use of Generative AI involved:
  - You must include the questions you used and GenAI answers (e.g. a link to the saved conversation if applicable; a text paste of the conversation otherwise)

  - You may **not** use GenAI to describe your code or write any part of `documentation.pdf`
  - **You must only use GenAI for the permissible uses described in the Generative AI policy on the course website.**

- In **code comments**, thoroughly document nontrivial lines and leaps of logic. We encourage documenting the shapes of numpy arrays, even if they may be obvious in the moment. Comment thoughtfully so that you may recall what you did upon re-reading your code in six months; consider the thought process of a hypothetical reader who is in the course but has not done this assignment.

Visualization

In this assignment we make use of Polyscope to visualize and interact with our mesh elements. Please refer to the linked documentation for more things you can show using Polyscope. We encourage you to make use of this package to visualize your outputs while working through this assignment, as writing standard unit tests is somewhat difficult when working with objects in graphics. Some example tests and visualizations are provided for you in the `starter.py` main function.

The `Mesh.view_basic` function does not consider the halfedge data structure and is good for viewing the loaded mesh without any topology edits, but will not reflect any such edits. For that, use `Mesh.view_with_topology` which has the additional ability to highlight the topology `Primitive` objects you pass into it.

1 Build Topology

Given the number of vertices `n_vertices` and an $|F| \times 3$ array of triangle faces with vertex indices `indices`, you will initialize the entirety of your halfedge data structure through the `build()` function in `topology.py`. Some check functions are called at the end of this function in `thorough_check()`. Your implementation should be able to pass all of these checks.

Note: we do not pass in the vertex array into `build()`. That is because the mesh **topology** (i.e. its structure) is completely independent of the coordinate positions of its vertices.

Each topology Primitive {`Halfedge, Vertex, Edge, Face`} will be indexed and stored in the `ElemCollection` class provided (you can basically consider these to be dictionaries). The `ElemCollection` class is equipped with an `allocate()` function, which you can call to initialize a new `Primitive` with a unique index. These `Primitive` classes are already declared for you in `primitive.py` with their relevant attributes listed. The `build()` should follow the rough outline:

```
pre-allocate n_vertices Vertex primitives
for face in indices:
    allocate Face f
    allocate 3 Halfedges (one for each face vertex/edge)
    (assuming face [i,j,k])
    for vertex index in [i,j,k]:
        he = one of the 3 allocated Halfedges
        he.next = another of the 3 allocated Halfedges
        v = self.vertices[vertex index]
        # set fields for he
        he.vertex = v
        he.face = f
        # set fields linking to he
        v.halfedge = he
        f.halfedge = he

        # NOTE: Iterating over the vertices is EQUIVALENT to iterating over the edges
        (assuming edge (i,j))
        if already visited edge (i,j):
            e = the Edge obj of edge (i,j)
            set he and e.halfedge as twins
            set edge of he to e
        else:
            allocate new Edge e
            link he and e to each other
```

VERY IMPORTANT: In order to pass our checks, you **MUST** allocate faces/halfedges in the order prescribed above. Namely, allocate faces row-by-row from `indices`, and if a face array contains indices [i,j,k], then allocate halfedges/edges in the order (i,j), (j,k), (k,i).

2 Primitive Construction

`primitive.py` contains the code for the parent `Primitive` class and declarations for the associate mesh elements {`Halfedge, Vertex, Edge, Face`}. You will now need to implement the remaining accessor functions, i.e. everything labeled with `TODO: P2`. (Remove the `raise NotImplementedError` line once you've begun completing each function.)

Tips:

- The vertex referenced by a halfedge is its root vertex, not its tip vertex.
- For debugging with `mesh.view_with_topology` to work, you may wish to implement `mesh.get_3d_pos` (from P4) at this point as well; it is *very* short and does not depend on anything from P3 or P4.

3 Check for Non-Manifoldness

Recall that a mesh vertex is manifold when all of its incident faces are adjacent to each other (i.e. form a fan). Recall that a mesh edge is manifold when it is incident to either 1 or 2 faces. See 1 for examples. Implement the `hasNonManifoldVertices()` and `hasNonManifoldEdges()` functions in `topology.py`. These should return `False` if there are no non-manifold elements and `True` otherwise. We give you two meshes to test your solution against. `nonemanif.obj` is non-edge manifold but is vertex manifold. `nonvmanif.obj` is non-vertex manifold but is edge manifold.

Note: You should comment out `thorough_check()` when loading these meshes. All we require is that your implementation can successfully build topology for non-manifold objs without breaking, and returns `True` when running the **the corresponding check it is non-manifold for**. You **DO NOT** need to check vertex manifoldness when the mesh is not edge manifold, and vice-versa. You will **NOT** be expected to deal with meshes which are both non-vertex and non-edge manifold. Your functions from P2 are **NOT** required to work for non-manifold meshes.

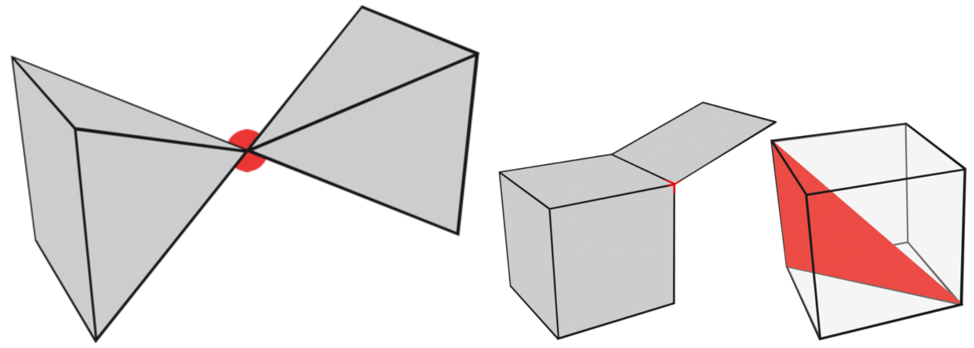


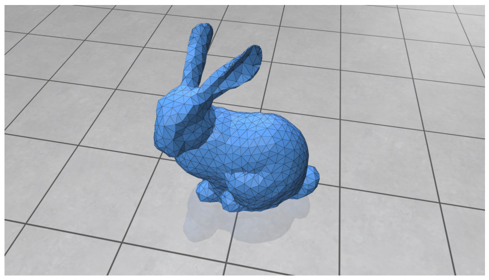
Figure 1: Non-manifold examples

4 Implement additional mesh functions

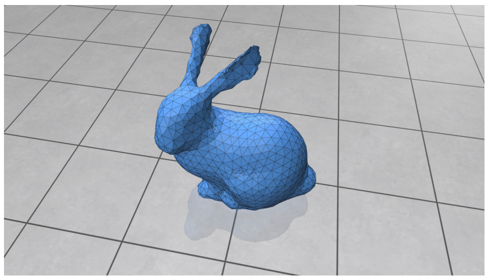
Implement the helper processing functions listed in `mesh.py`. Namely, `get_3d_pos()`, `vector()`, and `faceNormal()`.

5 Implement Laplacian Smoothing

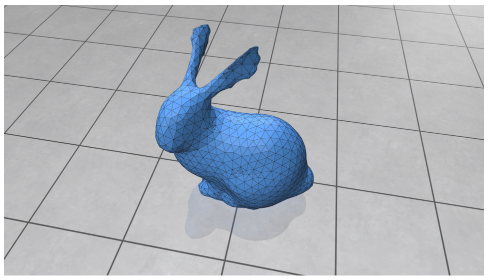
Your halfedge data structure is now complete! Implement `LaplacianSmoothing` in `edit.py`. Recall that this function smooths the mesh by setting each vertex position as the average of its neighbors' positions. Note that this function should generalize to any finite number of iterations of the Laplacian smoothing algorithm. Try it out with `example_smoothing()` in `starter.py`.



(a) Original Mesh



(b) After 1 smoothing iteration



(c) After 2 smoothing iterations

Figure 2: Example outputs from mesh smoothing on the Stanford bunny

6 Implement edge collapses

Implement `EdgeCollapse` in `edit.py`. To help keep organized, we've split the edge collapse implementation into 2 stages:

- Implement `prepare_collapse()` - Traverse the mesh's topology to **figure out which "operations" you will need to perform on your mesh to collapse the specified edge.** Operations include merging the two vertices of the collapsed edge, deleting primitives, and updating references. This function should not perform any operations, only list them out. You should store and return all operations in a `CollapsePrep` object. The `CollapsePrep` dataclass ("struct") already contains all the types of operations you may need, though note that you likely won't need all of these operations for your solution.
- Implement `do_collapse()` - **Perform all of the operations** specified in the inputted `CollapsePrep` dataclass. It is up to you to decide exactly how each operation works. We recommend implementing this first as it is more straightforward than `prepare_collapse()`.

This code will be called by the `EdgeCollapse` class, in turn called by the `collapse()` function in the `Mesh` class. To test this out, refer to `example_collapse_simple()` in `starter.py`.

An edge collapse will involve the **deletion** of

- the chosen edge,
- one of the edge's incident vertices,
- the two faces incident to the chosen edge,
- an additional edge from each of the indicent faces,
- and six total halfedges (two for each edge you delete, though they do not necessarily all have to belong to the deleted edges).

You should not create any new primitives, but should instead repurpose existing primitives by updating references using the operations in `CollapsePrep`. You will need to figure out how to reassign the primitive references accordingly (e.g. `next, twin, vertex, face, halfedge, etc..`) following the collapse to make sure the topology remains consistent. **Note that which vertex, edges, and halfedges to delete will be ambiguous. Your results should be the same with the given collapse sequence regardless of what you decide to delete.**

Hint: Depending on how you approach edge collapse, you may (or may not) need to consider the two cases `e.vertex.degree()` $= 3$ and `e.vertex.degree()` > 3 separately.

`documentation.pdf` diagram: It is extremely helpful (maybe even *essential*) to draw a diagram of the neighborhood of the edge being collapsed: the primitives it is adjacent to, the primitives to be deleted, and the primitives whose references must be repaired.

The position of your new vertex after your edge collapse should be the mean of the two vertices that make up the collapsed edge. See illustration:

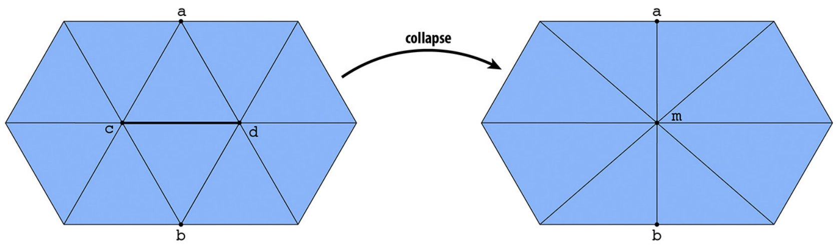


Figure 3: The result of an edge collapse.

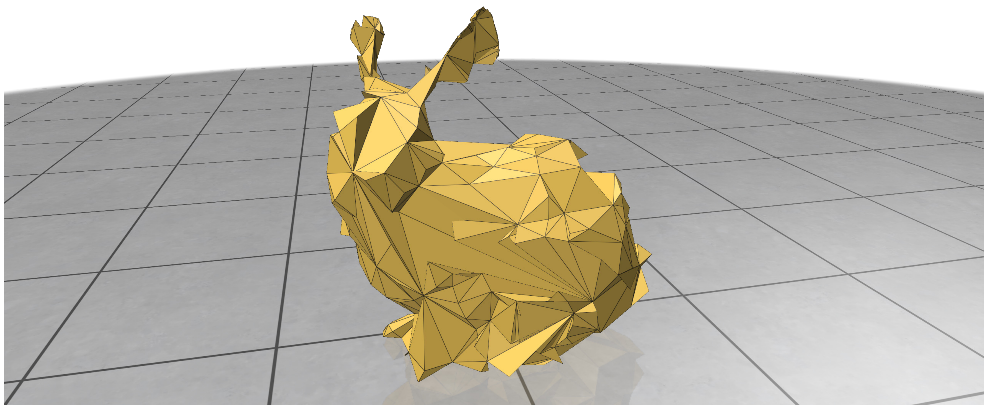
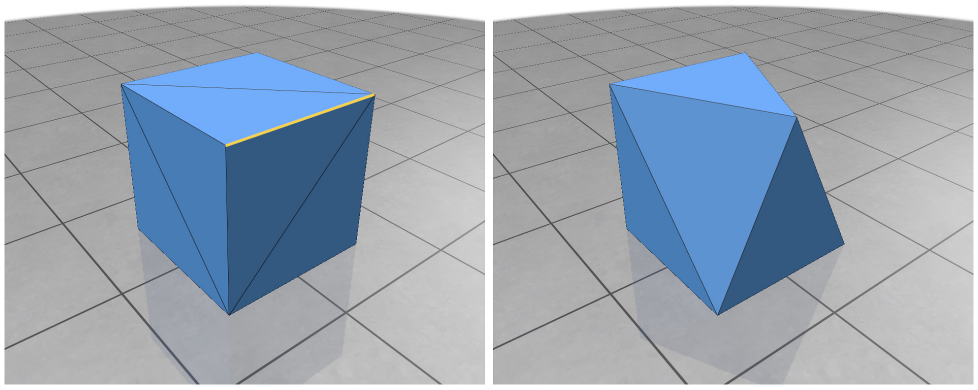


Figure 4: Expected output after collapsing edges in the the edge sequence specified in the file `bunny_collapses.npy`.

Debugging edge collapses

To help you debug your edge collapse implementation when starting this part, we've provided two meshes:

- `single_edge_collapse.obj`, which is a recreation of the diagram in figure 3 in a manifold mesh. The edge to collapse is edge index 0 and you should see a result like that in the diagram. Run this example with `example_collapse_simple()` in `starter.py`.
- `cube.obj`, which has a degree-3 vertex you can test on. The edge to collapse is edge index 0, and you should see the following before-and-after (figure 5). Run this example with `example_collapse_simple_cube()` in `starter.py`.



(a) Edge 0 in `cube.obj` (b) Result of collapsing this edge

Figure 5: Expected result of collapsing an edge with a degree-3 vertex in `cube.obj`

After getting these two simple examples to work, you can move onto trying a sequence of more than one collapse, either on these meshes (the cube has room to do a few more easy-to-inspect collapses) or on the bunny mesh.

Warning: It is very easy to get solutions that look correct when visualized but that actually have some connectivity issues in the background. These types of mistakes will cause significant and hard-to-debug issues when performing a sequence of edge collapses!

One way to detect these connectivity problems is by verifying your mesh is consistent using `mesh.topology.consistency_check()`.

Another way to detect these connectivity problems is to ensure that the number of vertices, edges, and faces match up with the Euler characteristic. In the case of genus zero shapes (surfaces without topological holes, which is the case for all the meshes we are testing you on):

$$2 = V - E + F$$

(Note that the general formula for arbitrary genus is $2 - 2g = V - E + F$, where g is the genus of the shape.)

7 Custom Mesh

Create or download a custom mesh of your choosing, load it with your newly created meshing module, and apply a few iterations of edge collapses and Laplacian smoothing. For your submission for this question, send us the original obj named `p7.obj` and the edited obj named `p7_custom.obj`. You can handcraft models using open source software such as Blender. Free 3D models can be readily found online at sites such as Turbosquid, Sketchfab, Free3D, and CGTrader. Many 3D models online may not have all triangle faces. To fix this, you can use Blender to triangulate the mesh.

8 Extra credit: Link condition for edge collapse

Unfortunately, even when the mesh is manifold, there are certain topological cases in which applying an edge collapse will result in non-manifoldness. These topological cases are completely characterized by the link condition. The link condition can be summarized briefly as follows:

For an edge **ab**, the one-ring of **a** intersected with the one-ring of **b** should equal the one-ring of **ab**. The one-ring of a vertex **b** consists of all the other vertices that share an edge with **b**. The one-ring of an edge **ab** consists of all the vertices of the faces sharing the edge, minus the edge vertices (**a**,**b**).

An equivalent definition for a triangle mesh, assuming a manifold edge (adjacent to exactly two faces, meaning the edge one-ring contains only two vertices), is

For an edge **ab**, the intersection of the one-ring of **a** with the one-ring of **b** should be exactly two vertices.

Implement `link_condition()` in `edit.py`. This function should return `True` if the inputted mesh and edge satisfy the link condition and `False` otherwise. This function will be called by `EdgeCollapseWithLink` before it performs an edge collapse using the same logic as in `EdgeCollapse`. If this check fails, `EdgeCollapseWithLink` will block the calls to `prepare_collapse()` and `do_collapse()` and will instead print an error message.

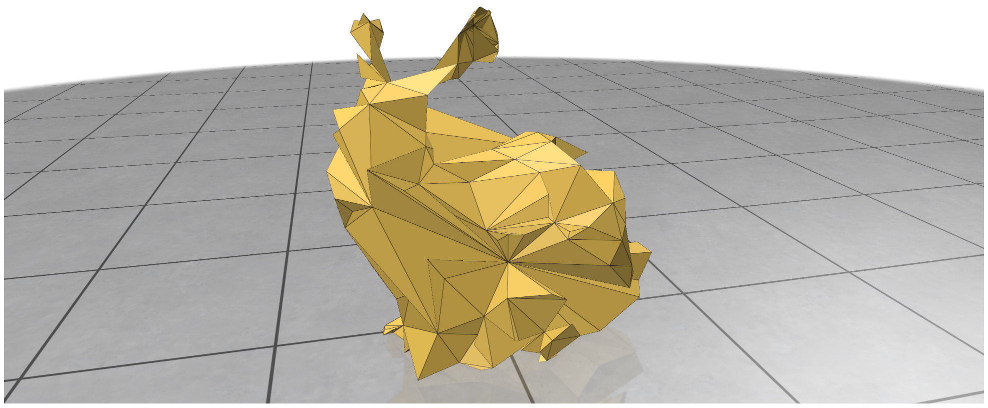


Figure 6: Expected output after collapsing edges in the edge sequence specified in `bunny_collapses_link.npy`, which contains edges that violate the link condition and need to be caught by your check.