# Assignment 3: Mesh Editing

CMSC 23700: University of Chicago
Due: May 3rd 11:59PM, 2025

## 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 >= 3.8
- 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.md`. (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 do not follow these instructions exactly, it will cause the autochecker to fail. This will result in a penalty to your grade.**

### Documentation

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

- You should aim to comment your code such that, hypothetically, another student just like you in the class who has not done the assignment reading your commented code can get a good idea of what to do (alternatively, so that yourself in about six months reading your own code again can remember what you did). Think about the thought process of a fresh reader as they read your code, and explain any nontrivial lines and logical leaps you make in the code.
- It is also highly encouraged to comment the numpy array shapes of important intermediate results, even if they may seem obvious in the moment. Array shapes are hard to visualize for a reader not sitting at an interpreter running the code, and writing the shapes out is very helpful in building a mental model of what happens, both for you (the programmer) and any reader.
- You're also encouraged to write type annotations in function signatures. This makes it easy to follow what functions expect and return.
- In **`documentation.md`**, 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.md`**, 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.md`
  - **You must only use GenAI for the permissible uses described in the Generative AI policy on the course website.**

### 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 in the mesh visualization.

# 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.)

## 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.

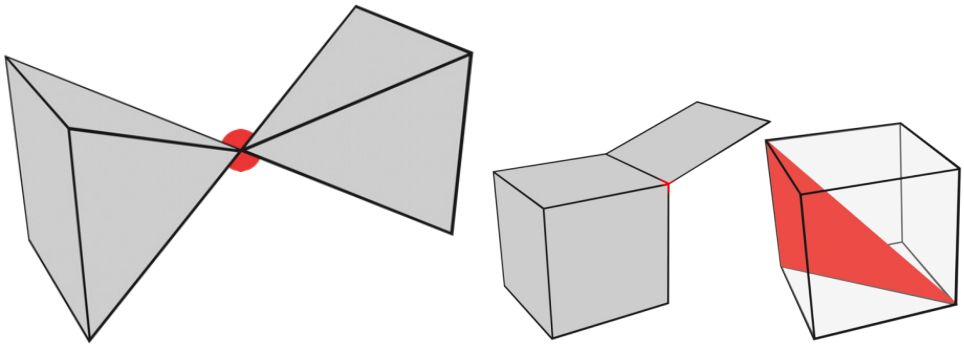


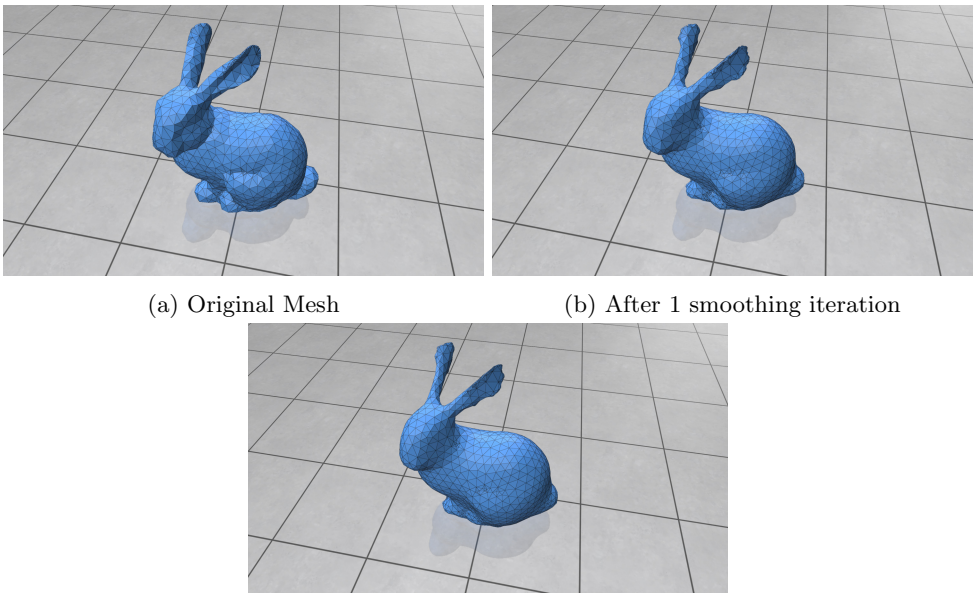
Figure 1: Non-manifold examples

## 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`, which involves the following steps:

- Implement `prepare_collapse` and add your own fields to the `CollapsePrep` dataclass ("struct") to gather up and prepare the relevant `Primitive` objects for the collapse, but without making any changes. `prepare_collapse` should return a `CollapsePrep` object containing these objects and data that you need, for use with `do_collapse`.
- Implement `do_collapse` which takes the output of `prepare_collapse` and makes changes to the linkages of the halfedge structure, edits a vertex position, and deletes `Primitive` objects belonging to vertices/faces/edges that no longer exist after the 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 (including its halfedges),
- one of the edge's incident vertices,
- the two incident faces,
- as well as one edge (including its halfedges) from each of the incident faces.

You will need to figure out how to reassign the mesh elements accordingly (e.g. `next, twin, vertex, face, halfedge, etc..`) following the collapse to make sure the topology remains consistent. **Note that which edge in each incident face and which vertex to keep will be ambiguous. Your results should be the same with the given collapse sequence regardless of which edges you keep.**

**Hint:** Consider the two cases `e.vertex.degree()` $= 3$ and `e.vertex.degree()` $> 3$ separately.

A useful check for you to implement is to ensure that the number of vertices, edges, and faces match up with the Euler characteristic. In the case of genus zero shapes (which is true for the bunny and funky cone):

$$2 = V - E + F$$

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

**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:**

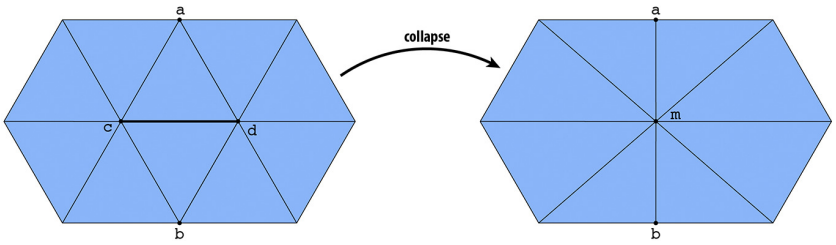



Figure 3: The result of an edge collapse.

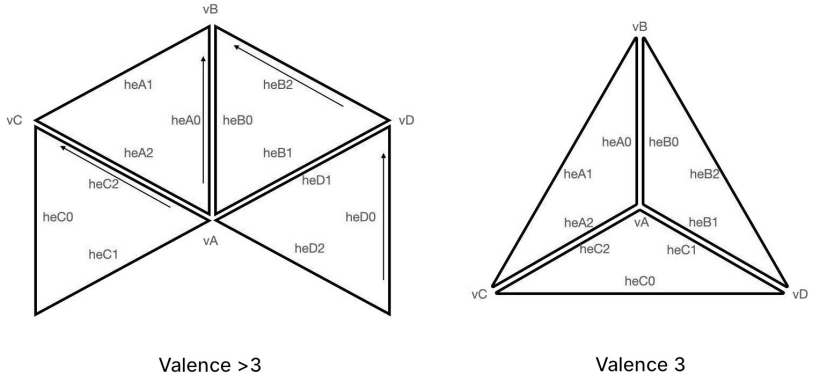



Figure 4: A naming convention for the halfedges, edges, and vertices in an edge collapse situation for the two main cases (degree $> 3$, and degree 3). The edge being collapsed is the one between `vA` and `vB`. *You don't have to follow this naming scheme, but it would be helpful so that students (and the course staff) have a common way to talk about and debug edge collapse implementations.*

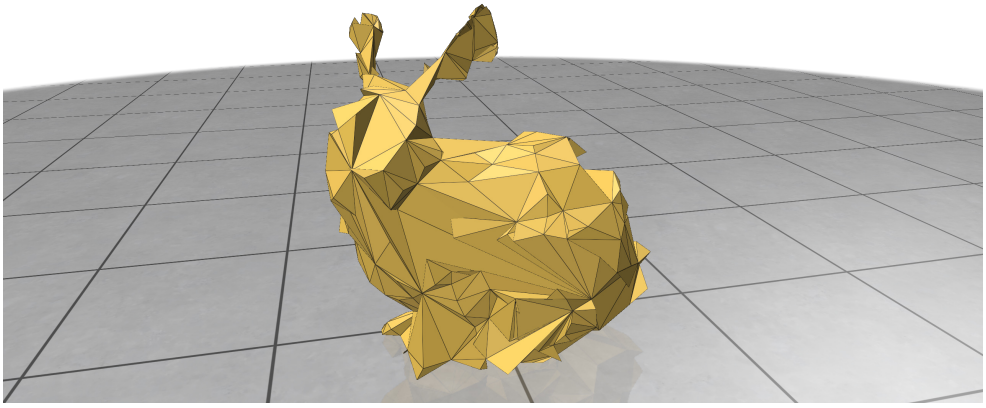
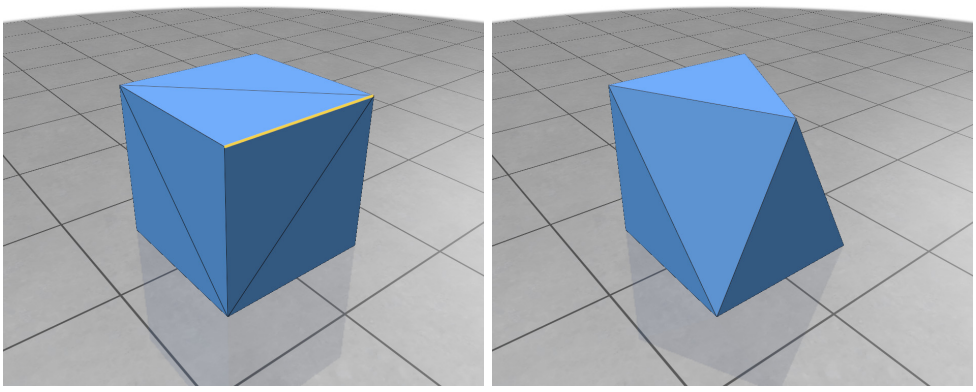


Figure 5: Expected output after collapses

## 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 6). Run this example with `example_collapse_simple_cube()` in `starter.py`.



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

Figure 6: 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.

# 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.

# 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, 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 `prepare_collapse_with_link_cond`. This should be mostly similar to your `prepare_collapse` from P6, except for the addition of the link condition check. If this check fails, you can immediately return **None** rather than continuing with gathering values for and returning a `CollapsePrep` object. The `apply()` function of `EdgeCollapseWithLink` will check if the return of `prepare_collapse_with_link_cond` is None and if so, not run the collapse. The rest of the collapse execution is identical to P6, so we may reuse the `do_collapse` you implemented (assuming you followed the suggested separation-of-concerns between *preparing* the collapse and *executing* the changes.)

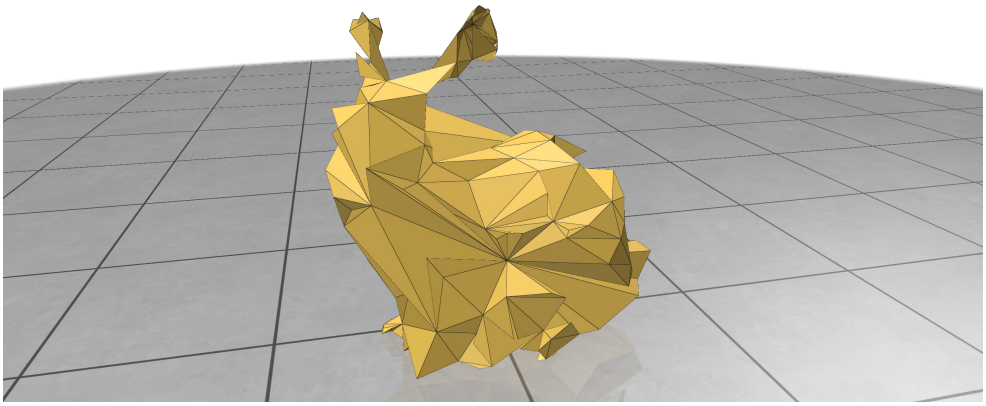


Figure 7: Expected output after collapses with link condition check