6.4420 Comp Fab Pset 3

Nomi Yu

April 2024

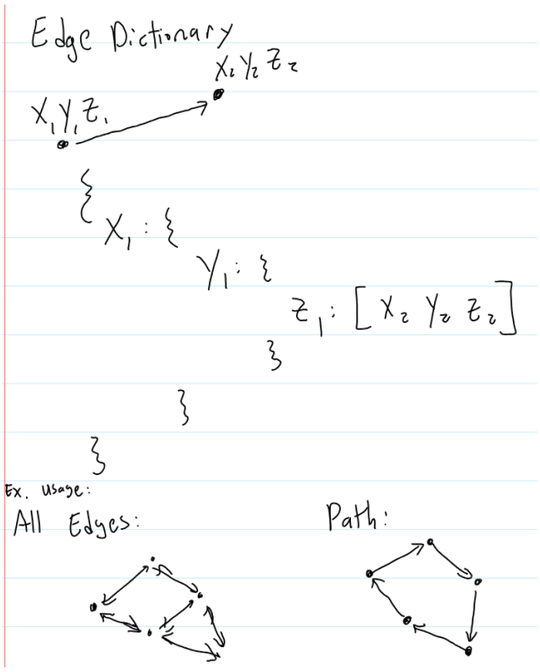
2. Print Slicer

2.1 Contours

Acceleration

The files for acceleration are located in slicing/VertexMap.py

The algorithm requires finding vertices connected to a given vertex. To avoid brute force searching through all vertices each time, I create a dictionary to quickly query connected edges as shown in the below graphic. To account for tolerancing, all entries are rounded to a tolerance (I chose 0.001 mm since it is unrealistic to physically achieve past that tolerance).



Algorithm and Edge Cases

I examined the following 4 edge cases:

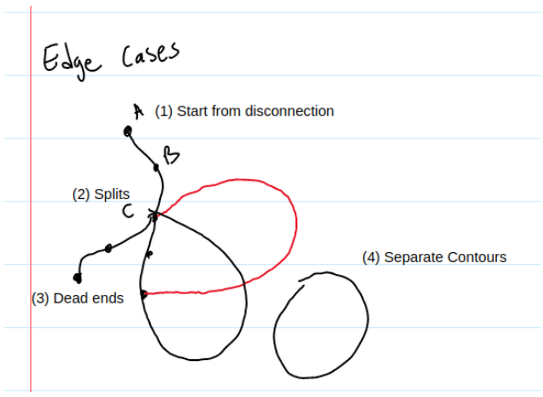
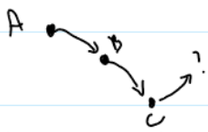


Figure 1: Edge Cases

The following summarizes the algorithm and how it handles edge cases:

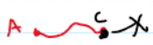
Algorithm:

- Begin from point A and travel down the path. At each point, query the connected vertices (excluding the previous vertex)

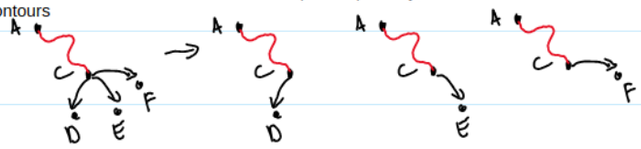


- next_vertices = get_next_vertices(vertex, exclude=last_vertex) **(note: accelerated by edge dictionary)**
- Cases:
  - 1 connected vertex: Add vertex to path and continue

  - 0 connected vertices: Dead end. This is not a path (**addresses edge case (3)**)

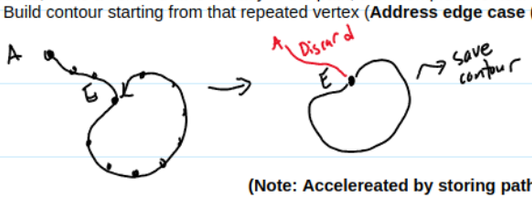


  - >1 connected vertices: Travel down each path separately, and obtain each of their contours



(**Addresses edge case (2)**)

- At all points, if hits a vertex that is already in the path, then complete contour has been found. Build contour starting from that repeated vertex (**Address edge case (1)**)



(Note: Accelereated by storing path in a dictionary)

- Once all contours starting from point A have been searched, remove all visited vertices from list of edges, and start a new search from remaining points **(This addresses edge case (4))**

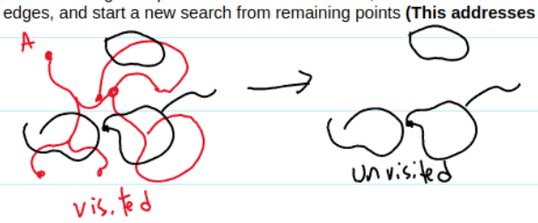
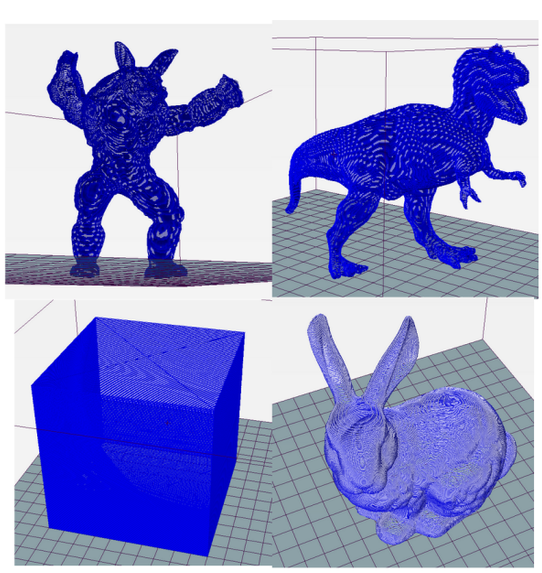


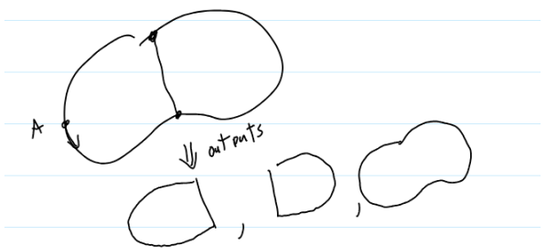
Figure 2: Algorithm

I obtain the following slices:



2.2 Robustness

I have attempted to address all edge cases / breaking inputs in my implementation. One issue, which is addressed by edge case (2) of splits but may still lead to ugly slicing, is if a given contour has an internal connected edge as shown below. This will produce repetitive contours as opposed to a single contour that envelopes everything.



3. Sheet Metal DSL

3.1 Design DSL

`generate_root_tab(`

- Width
- Length
- Angle

`) -> returns:`

- Index (a designator for future reference)

The initial generation has the same parameters as the child tab, but without positioning. The top left point is positioned at the origin.

`generate_child_tab(`

- Width
- Length
- Angle
- Offset (from parent)
- Parent Index (returned from generation)
- Parent Side (1 of 4 sides of parent

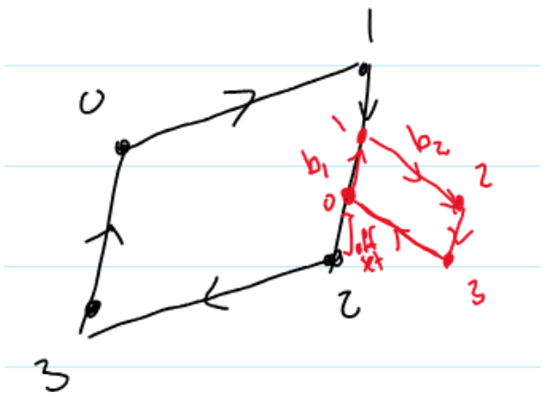
`) -> returns:`

- Index (a designator for future reference)

Same parameters described in the problem, but with an index reference to describe which parent to connect it to.

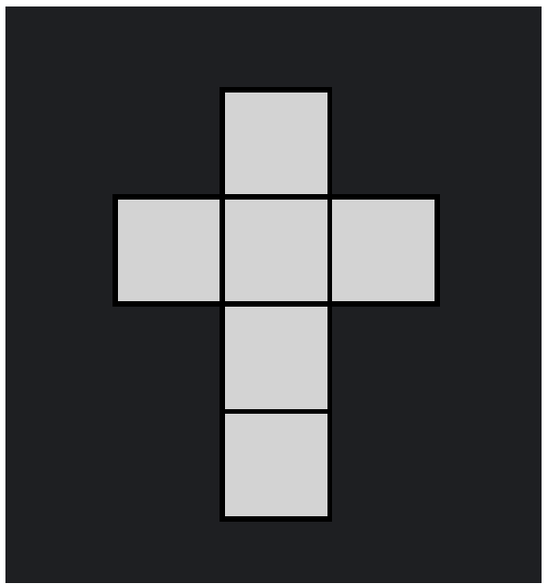
3.2 Implement

The implementation is straightforward. The only item of note is the convention of ordering points in CCW and computing vertex positions by defining a basis b_1, b_2 from the parent edge



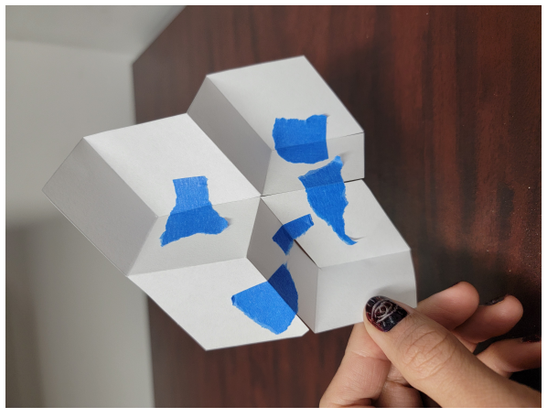
3.3 Cube

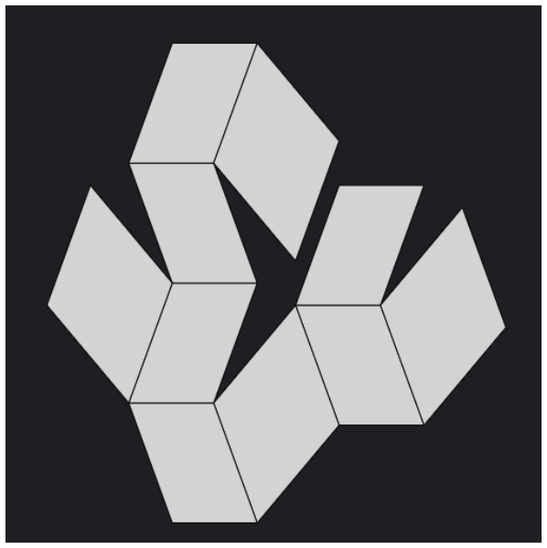
Files located in cube.py and cube.svg



3.4 Custom

Files located in complexj.py and complexj.svg





4. Discussion

4.1 DSL Application

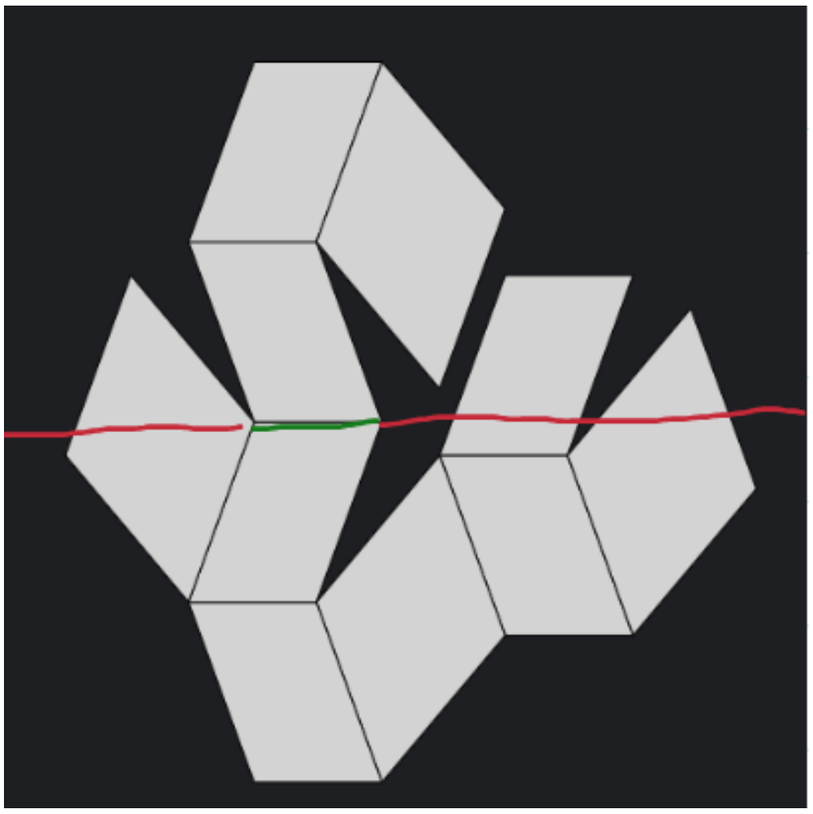
Two use cases for a DSL that ensures manufacturability are:

- Manufacturable design space. During a design search/optimization, the DSL can guarantee that any output is manufacturable

- High level abstraction. The DSL can make it easier for a non-expert to interact with carpentry by provinding a more interpretable interface. It is the same way that python acts as an interface to C++ or C++ as an interface to assembly.

4.2 Challenges and Tradeoffs

Available sheet metal presses are often wide and cannot bend isolated regions. Consider the prior complex sheet metal shape. In order to bend the edge in green, the intersected faces (red) will also need to be bent. A verifier could extend all edges and see if they intersect with another face



6. Extra Credit

6.2 Acceleration

A dictionary acceleration above allows for edge search in $\mathcal{O}(1)$ time for any vertex (3 dereferences), and it takes $\mathcal{O}(n)$ time to construct a dictionary for n edges