CS 184: Computer Graphics and Imaging, Spring 2022

Project 2: Mesh Editor

Larry Yan, CS184

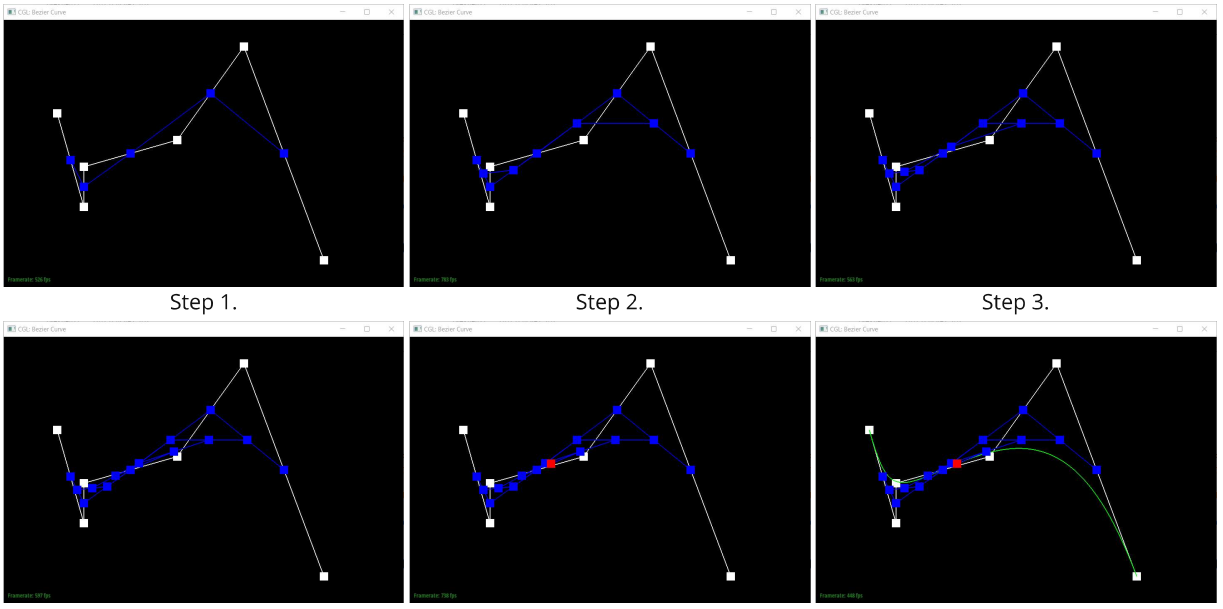
Overview

In this project, I implemented elements of rendering and manipulating meshes. First, I implemented the recursive logic for rendering bezier curves and surfaces, allowing for curved surfaces to be drawn. I then allowed for smoother shading with vertex normals, as well as mesh processing elements with edge flipping and splitting, before finally utilizing those in order to upsample meshes. Seeing the rendered meshes change based on the implemented functions, as well as the interface itself, was very impressive. Parts 4 and 5 did somewhat feel like an exercise in drawing a diagram and assigning pointers, but successfully doing so was nice.

Section I: Bezier Curves and Surfaces

Part 1: Bezier curves with 1D de Casteljau subdivision

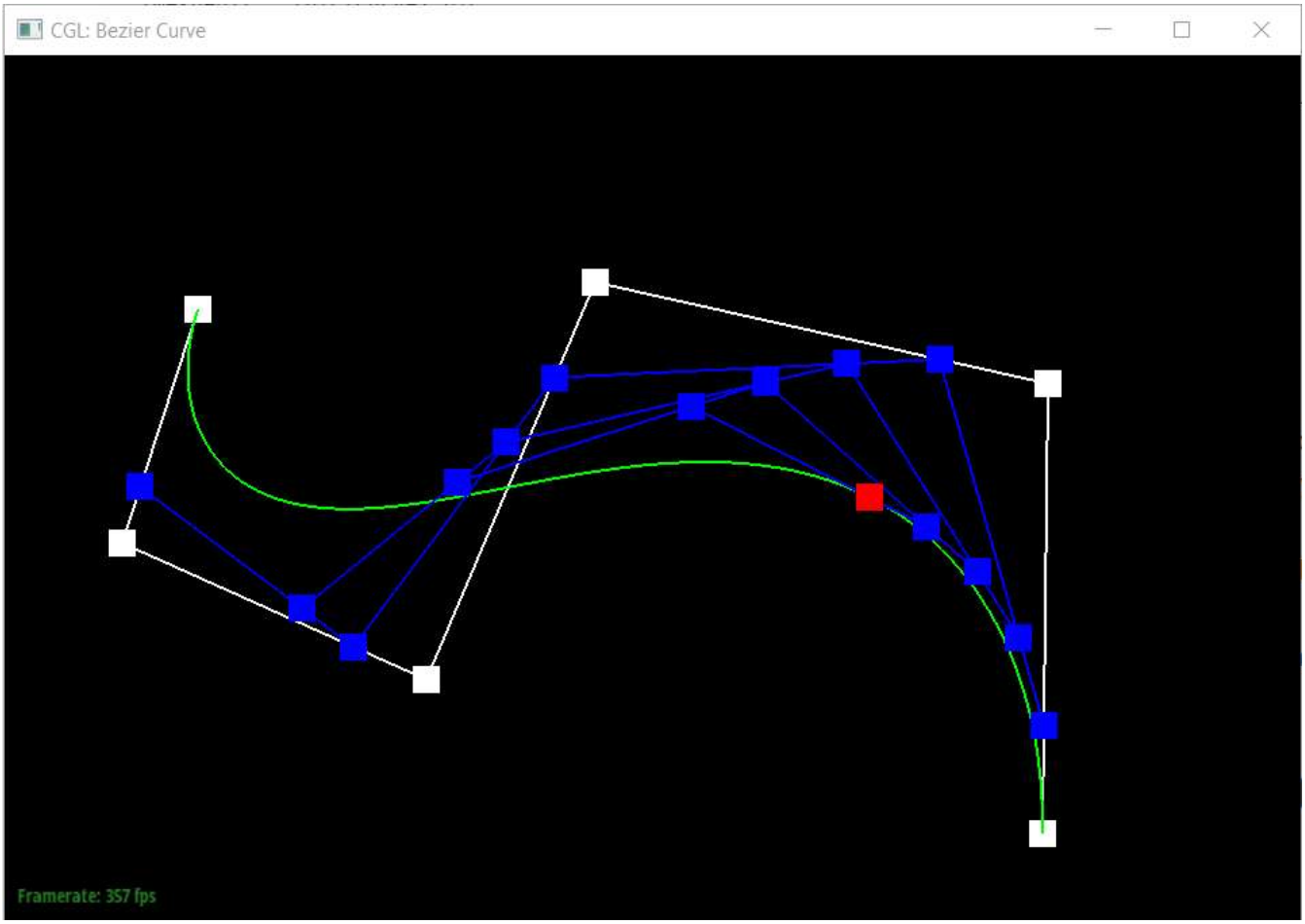
De Casteljau's algorithm is a recursive procedure for finding points on a bezier curve from its control points. From n control points, we find the next level of (n-1) intermediate points by interpolating each set of neighboring control points at some value t. This process is repeated with the intermediate points until only one point remains, which lies on the bezier curve. Repeating this process for differnt t can produce the full curve. For part 1, the recursive function takes a vector of Vector2D coordinates, which are then interpolated with float t to produce a new vector for the next step.



Step 1. Step 2. Step 3.

Step 4. Step 5. With curve.

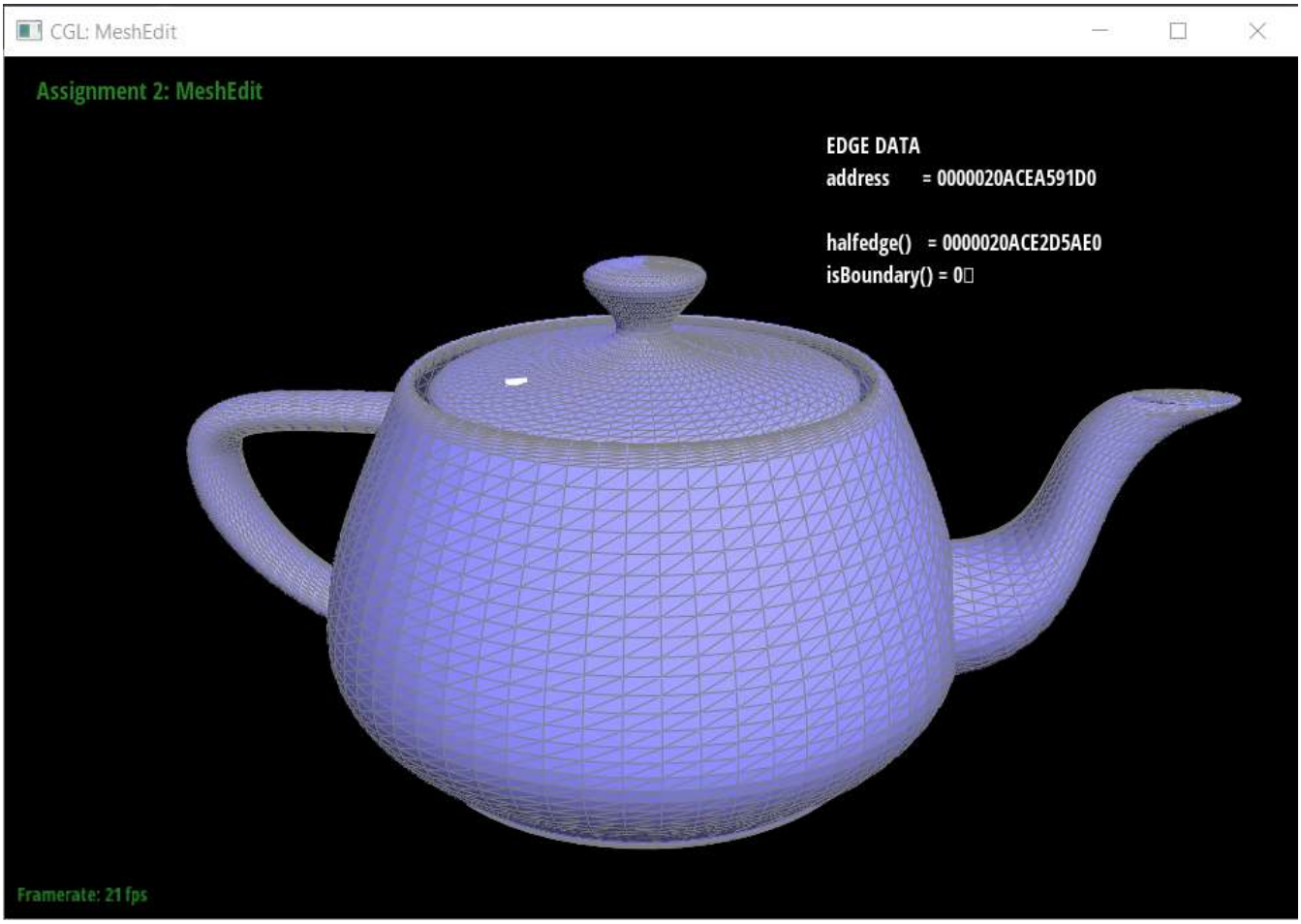
Steps to evaluating a 6 control point bezier curve.



The curve with shifted control points.

Part 2: Bezier surfaces with separable 1D de Casteljau subdivision

For bezier surfaces, instead of having n control points, we instead have n^2 points, where we first evaluate n bezier curves along one axis, producing points at some value u, then use those points as the control points for a second bezier curve along another axis for values v. By varying u and v, the full surface can be drawn. The overall recursive step is the same as in part 1 but with Vector3Ds, and additional functions for evaluating a 1D curve point (recursively call the step until only one point remains) and evaluating a curve (get the points for the n 1D curves, then use those for another 1D curve in the other direction) were used.

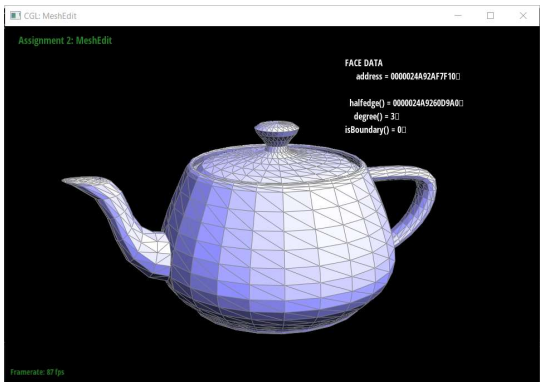


A teapot made of bezier surfaces.

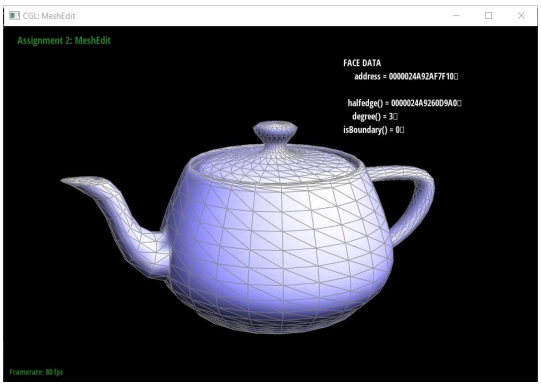
Section II: Sampling

Part 3: Average normals for half-edge meshes

To implement area-weighted vertex normals, I iterated through the faces adjacent to the vertex, iterating halfedges and getting their vertex positions, which I used to construct two vectors representing the sides of that triangle. I took a cross product of those vectors (proportional to the size of the triangles) and acumulated those, and after summing the products for all the faces, I normalized them. I also negated the sum in order for the vector to point outward (likely due to the order in which I multiplied the vectors for the cross product).



Default shading without vertex normals.



Phong shading with vertex normals.

Part 4: Half-edge flip

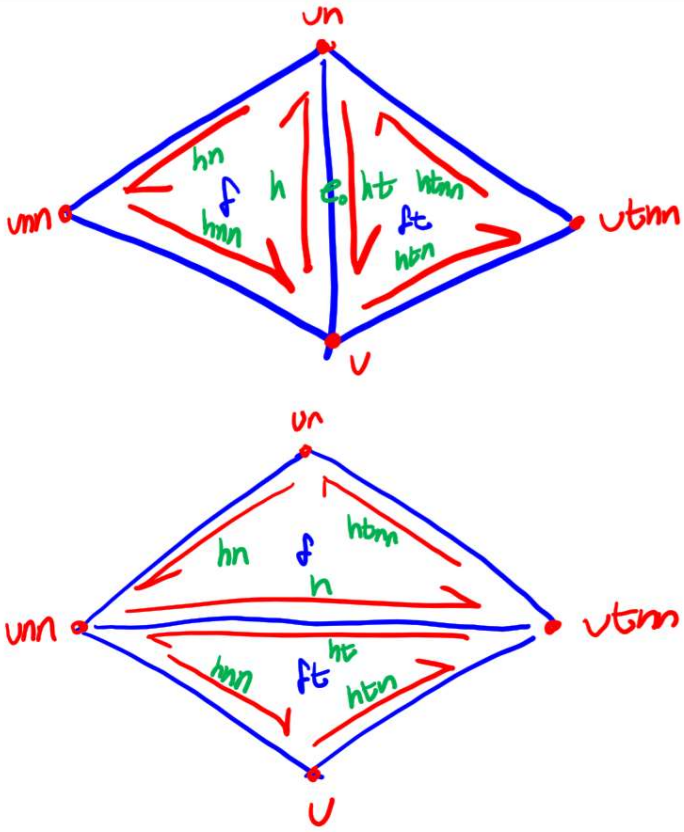
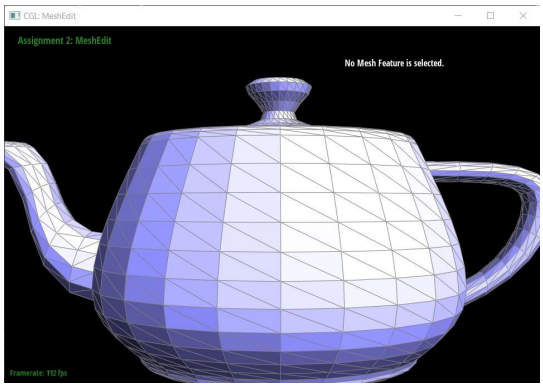
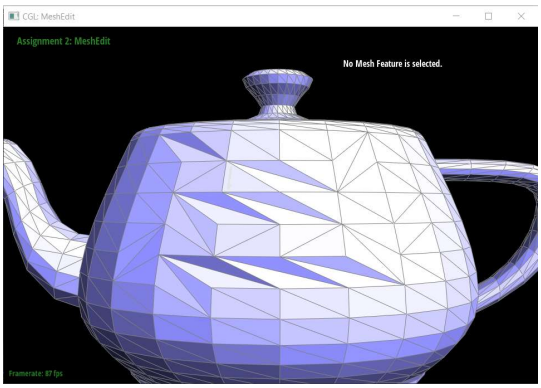


Diagram of changes to mesh elements used for this part.

To implement edge flipping, I drew out a diagram of the changes, then reassigned any pointers that needed to be changed. There was no significant debugging difficulties, with only two main issues; in the first, I forgot to assign values to the faces (set f = h->face()), and I was segfaulting on edge flipping because I was attempting to access null. In the second, I was seeing missing faces, but this was self-inflicted, as when debugging the first issue, I forgot to remove an early return statement before setting face pointers, resulting in detached faces.



Teapot before flips.



Teapot after flips.

Part 5: Half-edge split

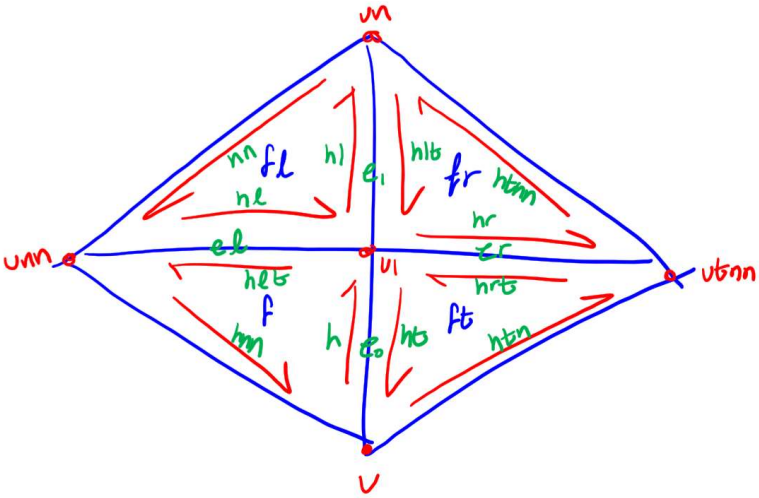
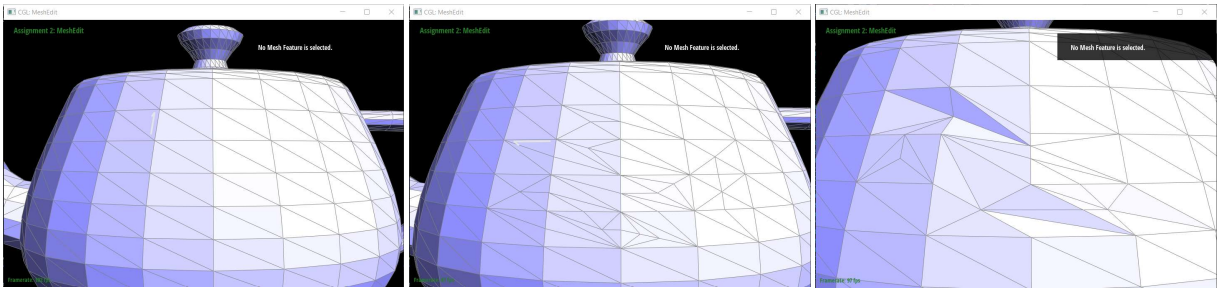


Diagram of changes to mesh elements used for this part.

To implement edge splits, I followed a similar procedure as in the previous part, using the above diagram to reassign pointers. Additionally, several new mesh elements were created, which were added via new[element]() functions and pointers set with setNeighbors(). For this part, there were no significant debugging issues, with only one failure because I forgot to initialize two new edges.

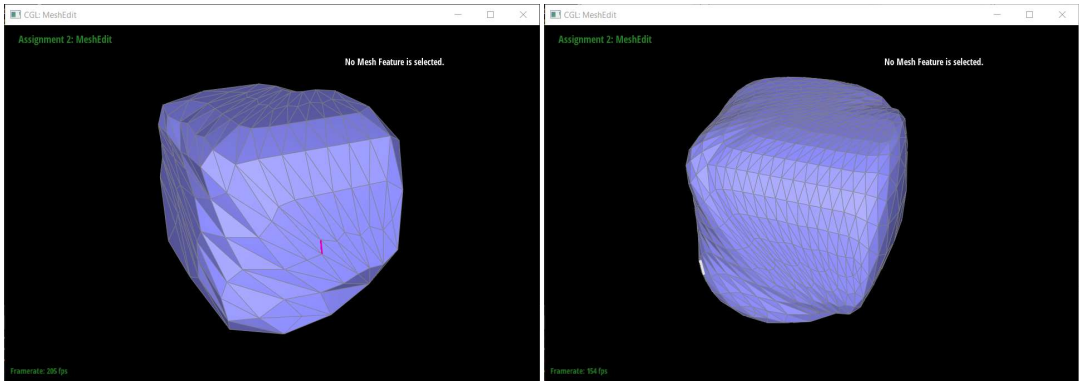


Teapot before splits. Teapot after splits. Teapot with both flips and splits.

Part 6: Loop subdivision for mesh upsampling

For loop subdivision, I generally followed the five step process described in the comments. First, I iterated through all of the preexiting verticies and calculated their new positions with the weighted sum formula from the sum of the positions of all adjacent vertices and the degree of the vertex. I also marked these vertices as not new. For the second part, for all non-boundary edges, I calculated the new positions of the vertices that would be on them, and also marked these edges as not new. For the third part, I iterated through all edges and split those that were not boundaries and not new. I also set the vertex::newPosition to edge:newPosition for each new vertex. When testing, I found that I was infinite-looping during this part: I discovered that isNew for edges was not immediately set for all new edges, but was intended only those inside an original triangle. Instead, I decided to manually set isNew to be true for the newly created edges in splitEdge, which allowed this part to split the original edges properly. For the fourth part, I iterated through all vertices, and for those that were not new (from split edges), I took the edge associated with their half-edge and marked it as not new. Based on the way I implemented splitEdge, this would always be the new half of the original split edge. Then, I could iterate through all edges and flip all the new edges between a new and old vertex. For part 5, I just iterated through all vertices again, and set their positions to newPosition, as all of them had already had that set properly.

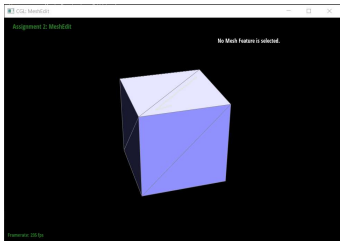
Generally, loop subdivision removes sharp edges and rounds them out. Splitting the edges ahead of time reduces this effect, as the vertices in a plane do not cause movement, while those that are on the edges have vertices closer to them, reducing the rounding effect.



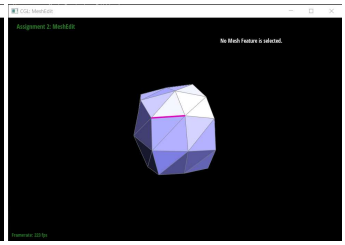
Various flips produces slightly less rounding

This is more apparent with more subdivision.

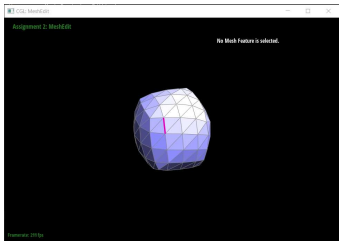
Without any changes, the basic cube becomes slightly asymmetrical due to the two triangles on each face affecting the four vertices of that face unequally: vertices with higher degree become closer to the center, while those with lesser degree remain more pointed. This can be avoided with a single edge split on the diagonal edge on each face, resulting in a symmetric X on each face. This results in equal degree on all vertices and allows for symmetric division.



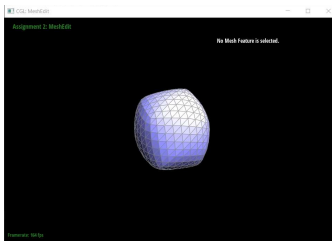
No changes, Step 1.



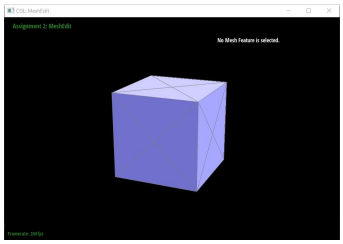
No changes, Step 2.



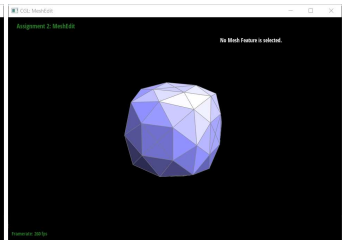
No changes, Step 3.



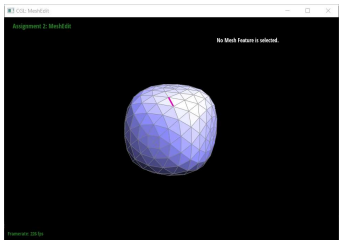
No changes, Step 4.



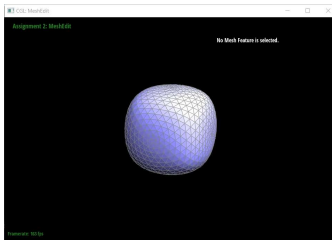
With one split, Step 1.



With one split, Step 2.



With one split, Step 3.



With one split, Step 4.

Section III: Optional Extra Credit

If you are not participating in the optional mesh competition, don't worry about this section!

Part 7: Design your own mesh!

Link: https://cal-cs184-student.github.io/sp22-project-webpages-yanlarry/