CS184: Computer Graphics

Sping 2022

Assignment 2: Meshedit

Prince Wang

Overview

An overview of the project, your approach to and implementation for each of the parts, and what problems you encountered and how you solved them. Strive for clarity and succinctness. On each part, make sure to include the results described in the corresponding Deliverables section in addition to your explanation. If you failed to generate any results correctly, provide a brief explanation of why. The final (optional) part for the art competition is where you have the opportunity to be creative and individual, so be sure to provide a good description of what you were going for and how you implemented it. Clearly indicate any extra credit items you completed, and provide a thorough explanation and illustration for each of them.

Task 1: Bezier Curves with 1D de Casteljau Subdivision (10 pts)

Requirements

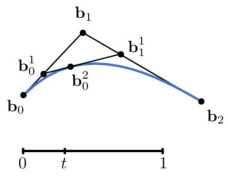
- Briefly explain de Casteljau's algorithm and how you implemented it in order to evaluate Bezier curves.
- Take a look at the provided .bzc files and create your own Bezier curve with 6 control points of your choosing. Use this Bezier curve for your screenshots below.
- Show screenshots of each step / level of the evaluation from the original control points down to the final evaluated point. Press E to step through. Toggle C to show the completed Bezier curve as well.
- Show a screenshot of a slightly different Bezier curve by moving the original control points around and modifying the parameter tt via mouse scrolling.

Explain de Casteljau's algorithm and how you implemented it in order to evaluate Bezier curves:

de Casteljau's algorithm is a recursive method to evaluate a Bezier curve. For an order n bezier curve, we first specify n+1 control points. Then, we insert a point t on every line that connects the two consecutive control points (algebraically this is equivalent to linearly interpolating on the two end points). We do this for all lines, resulting in n points. Using these n points we create n-1 points. We recursively apply this procedure until we have one point left. The trajectroy that this point went through will be our curve. Below is an example from lecture slide that represents this algorithm in algebraic format:

Bézier Curve – Algebraic Formula

Example: quadratic Bézier curve from three points



$$\mathbf{b}_0^1(t) = (1-t)\mathbf{b}_0 + t\mathbf{b}_1$$
$$\mathbf{b}_1^1(t) = (1-t)\mathbf{b}_1 + t\mathbf{b}_2$$

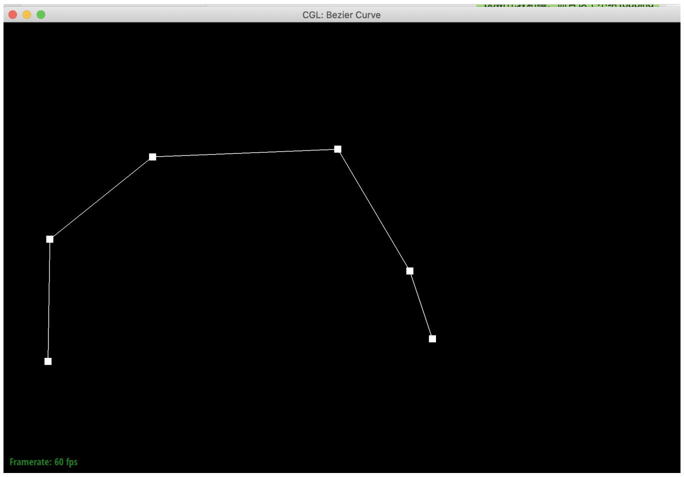
$$\mathbf{b}_0^2(t) = (1-t)\mathbf{b}_0^1 + t\mathbf{b}_1^1$$

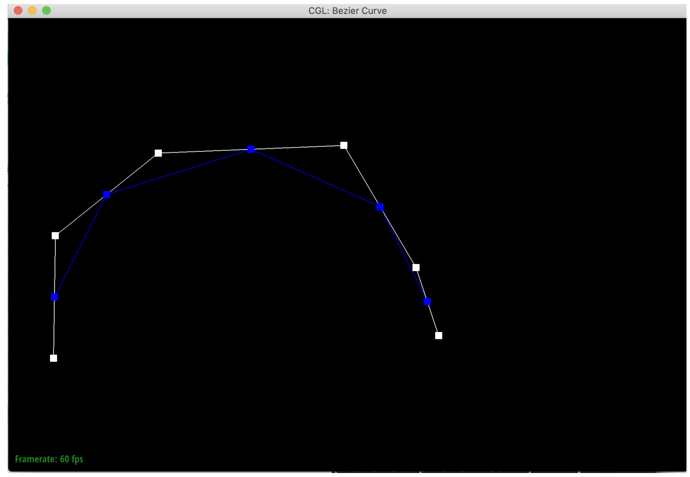
$$\mathbf{b}_0^2(t) = (1-t)^2\mathbf{b}_0 + 2t(1-t)\mathbf{b}_1 + t^2\mathbf{b}_2$$

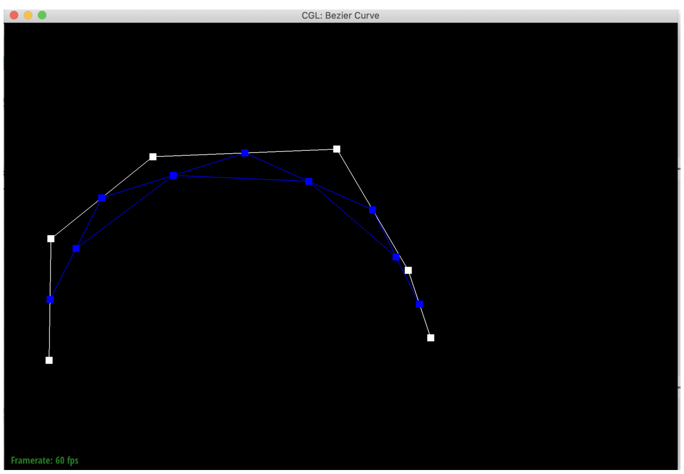
CS184/284A Ren Ng

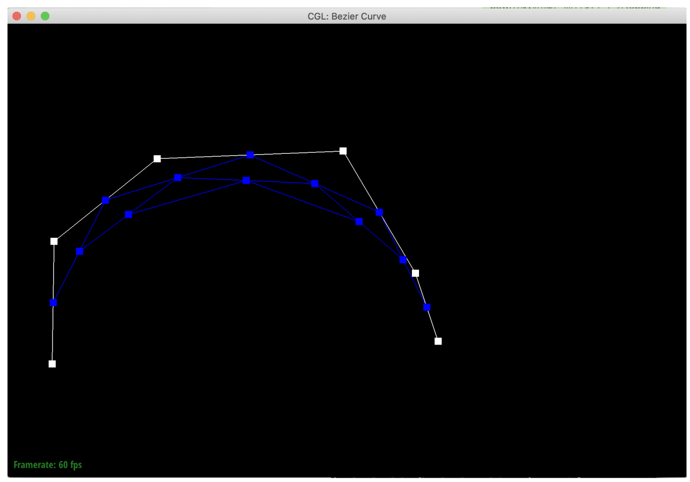
Take a look at the provided .bzc files and create your own Bezier curve with 6 control points of your choosing. Use this Bezier curve for your screenshots below.

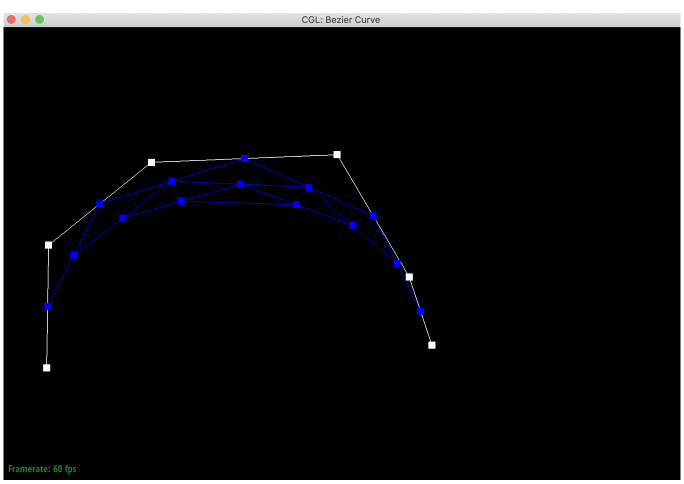
Show screenshots of each step / level of the evaluation from the original control points down to the final evaluated point. Press E to step through. Toggle C to show the completed Bezier curve as well.

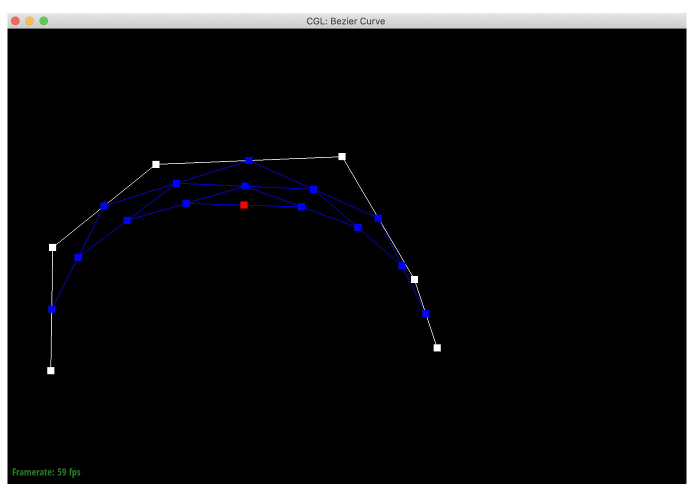


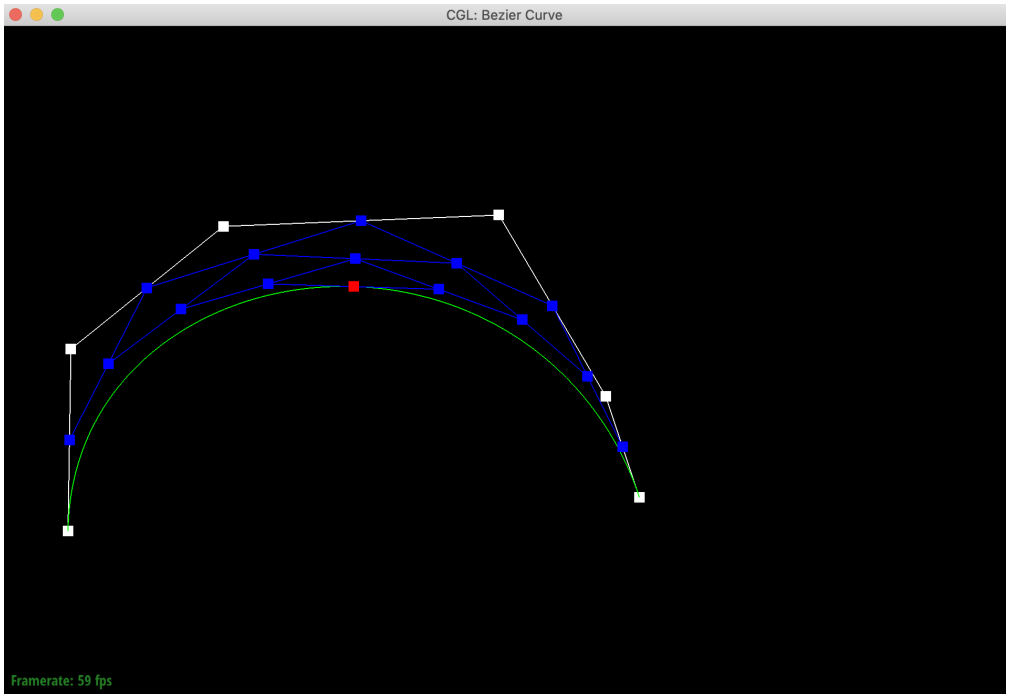




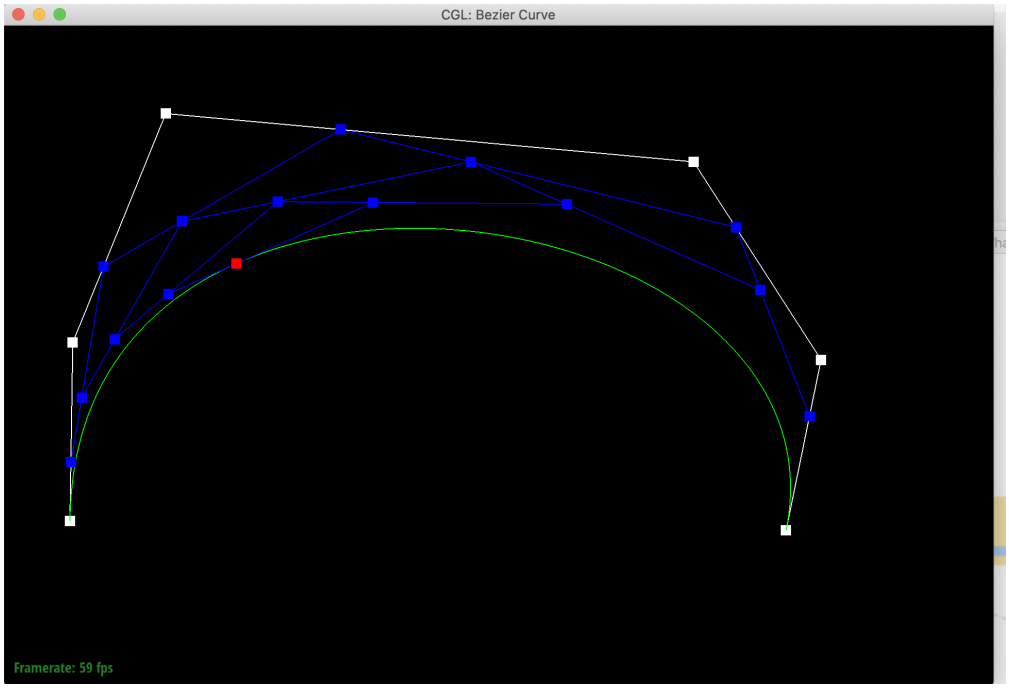








Show a screenshot of a slightly different Bezier curve by moving the original control points around and modifying the parameter tt via mouse scrolling.



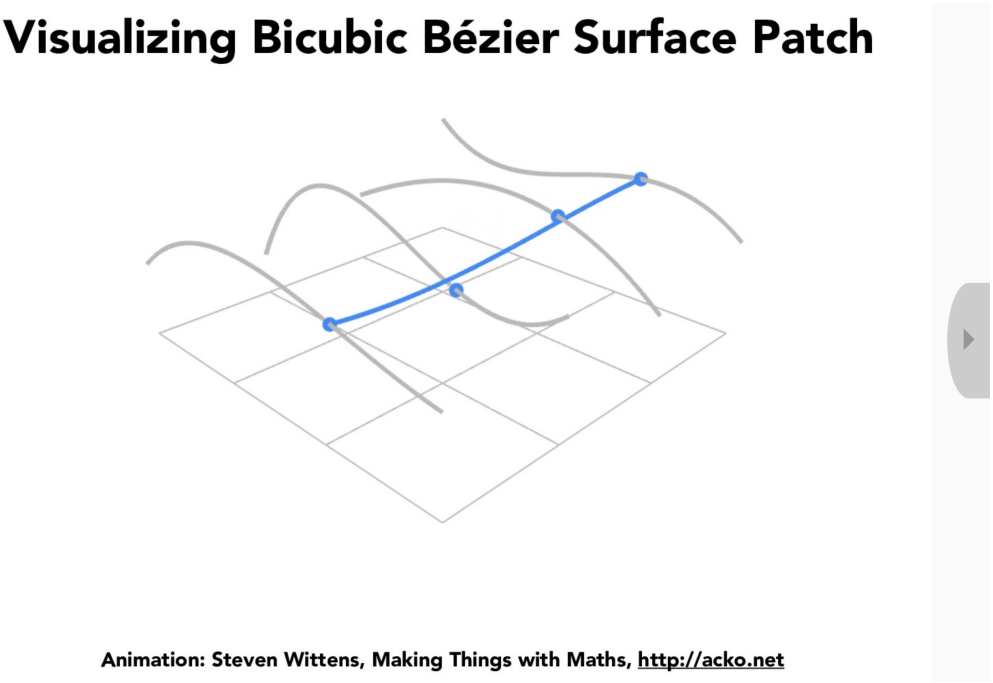
Task 2: Bezier Surfaces with Separable 1D de Casteljau (15 pts)

requirements

- Briefly explain how de Casteljau algorithm extends to Bezier surfaces and how you implemented it in order to evaluate Bezier surfaces.
- Show a screenshot of bez/teapot.bez (not .dae) evaluated by your implementation.

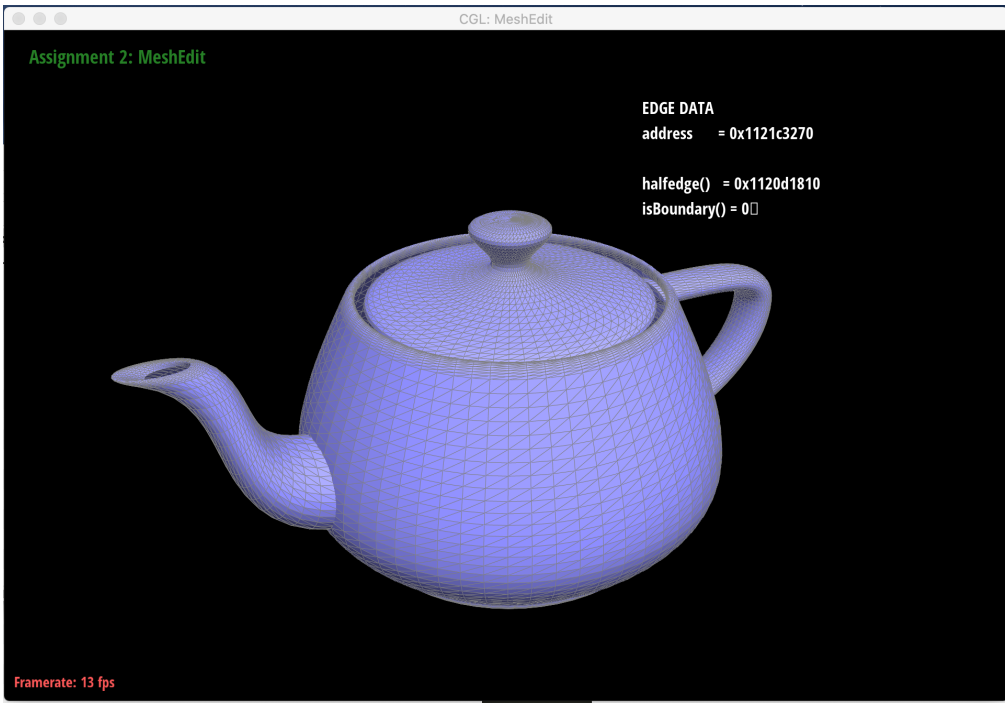
explain how de Casteljau algorithm extends to Bezier surfaces and how you implemented it in order to evaluate Bezier surfaces

The key to extending the algorithm onto Bezier surface is that a surface can be constructed using a series of curves. If we have a series of curves across an axis, then if we construct a curve by connecting the points along another axis, and move this curve along the other axis, we will create a surface.



In terms of the algorithm and implementation, it is the following. Given a (u,v) coordinate, we first construct bezier curves along the u direction, creating a series of bezier curves using the original de Casteljau algorithm. Then, taking the points at u we received from creating the bezier curves, we use this points and construct a bezier curve along the v direction, getting the final point that we want.

Show a screenshot of bez/teapot.bez (not .dae) evaluated by your implementation.



Task 3: Area-Weighted Vertex Normals (10 pts)

requirements

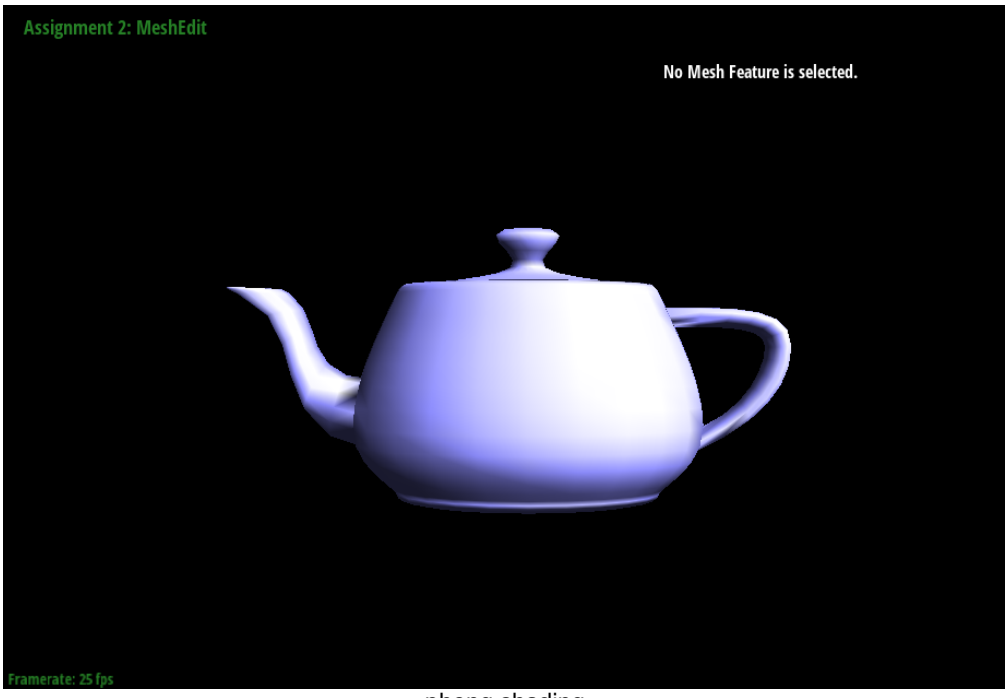
- Briefly explain how you implemented the area-weighted vertex normals.
- Show screenshots of dae/teapot.dae (not .bez) comparing teapot shading with and without vertex normals. Use Q to toggle default flat shading and Phong shading.

Briefly explain how you implemented the area-weighted vertex normals.

We utilizes the haldedgemesh data structure. To compute the area-weighted vertex normal, we traverse through all the faces that the vertex is attached to. Then, we aggregate the face normal vector of each faces, and weigh them by their area (which can be computed by taking the magnitude of the cross product of two edges divided by 2). Finally, we normalized our result vector to get the vertex normal.

screenshots of dae/teapot.dae (not .bez) comparing teapot shading with and without vertex normals

default flat shading



phong shading

Task 4: Edge Flip (15 pts)

requirements

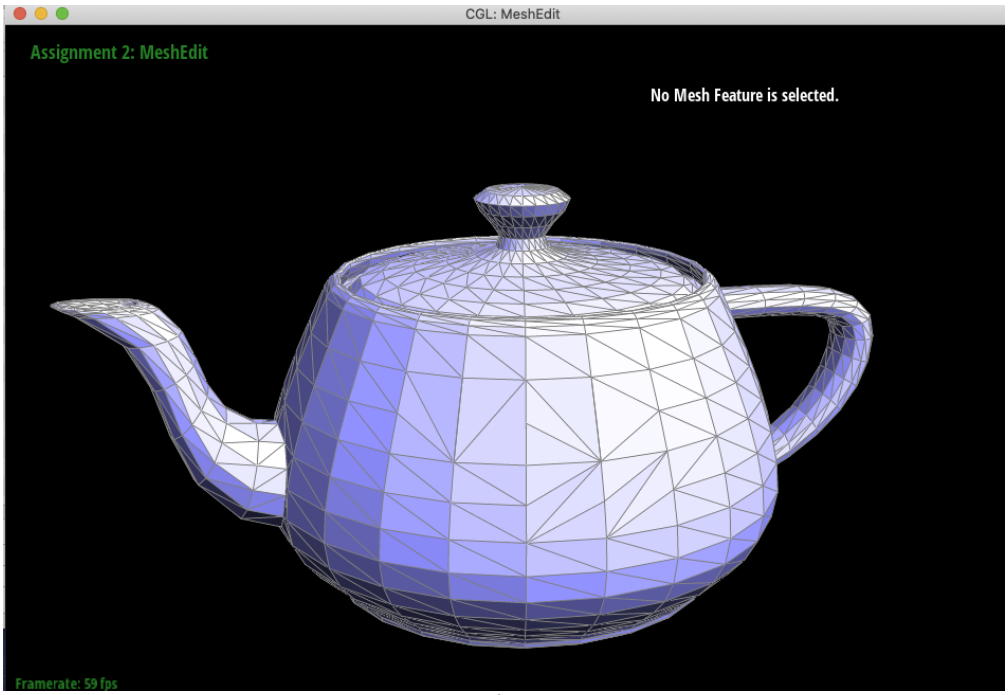
- Briefly explain how you implemented the edge flip operation and describe any interesting implementation / debugging tricks you have used.
- Show screenshots of the teapot before and after some edge flips.
- Write about your eventful debugging journey, if you have experienced one.

Briefly explain how you implemented the edge flip operation and describe any interesting implementation / debugging tricks you have used.

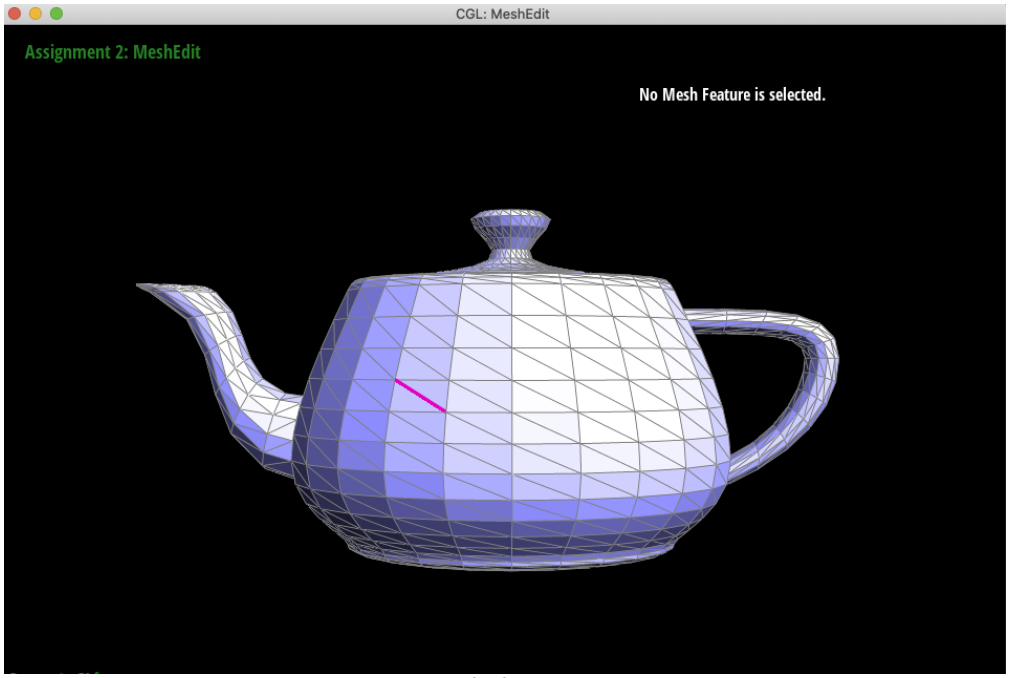
To implement edge flip, I retrieved all related variables such as vertices, faces, edges and half edges. Then, through a series of reassignment, I performed edge flip. I implemented and figured out the correct ordering of reassignment by drawing on my notebook a simple case of a diamond splited vertically in half, and examined how variables changed after the diamond became to be splitted horizontally. Here is a rough sequence of steps performed:

Step 1: reassign the fields of all the 6 half edges inside the diamond, such that each 3 of them formed a close loop. Step 2: reassign the four half edges outside the diamond. Step 3: Check all other vertices, faces and edges, if there is a need to reassign, then do so.

I did not have to debug much. In fact I was able to ran the code in my first try, so my coding journey is not very eventful.



after



before

Task 5: Edge Split (15 pts)

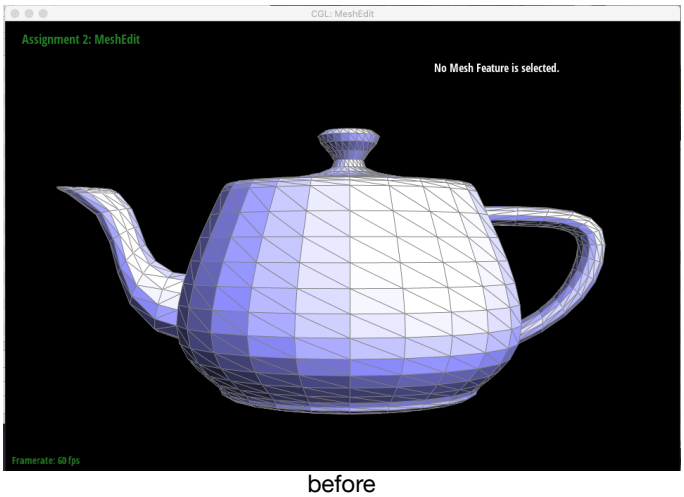
requirements

- Briefly explain how you implemented the edge split operation and describe any interesting implementation / debugging tricks you have used.
- Show screenshots of a mesh before and after some edge splits.
- Show screenshots of a mesh before and after a combination of both edge splits and edge flips.

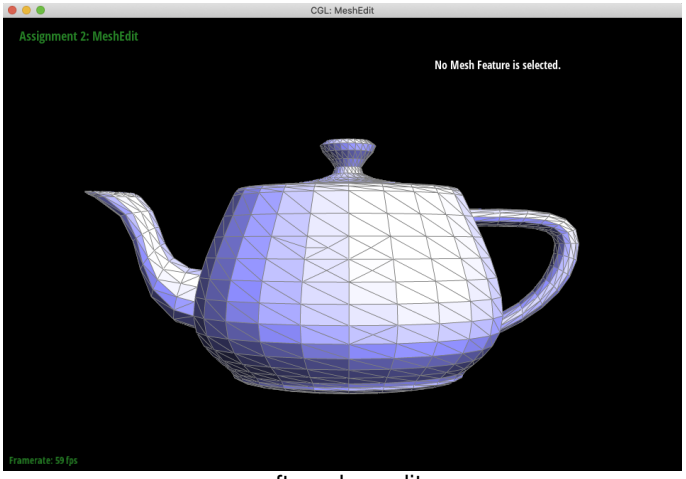
Briefly explain how you implemented the edge split operation and describe any interesting implementation / debugging tricks you have used.

The edge split is similar to the process of edge flip, but with a little more added complexity. We basically want to determine what relationship edges, halfedges, faces and vertices have with each other after splitting via reassigning. The challenge is that now we need to create new elements, but the rationale remain the same. I figured out an order of reassigning that ensured my function runs correctly through drawing on my notes, just like in task 4. A worthnoting point about the implementation is that we need to make sure we set isNew to true for our new elements.

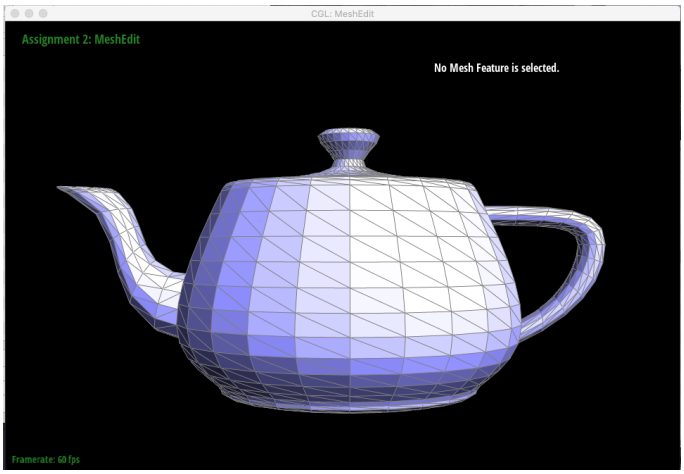
comparison of before and after



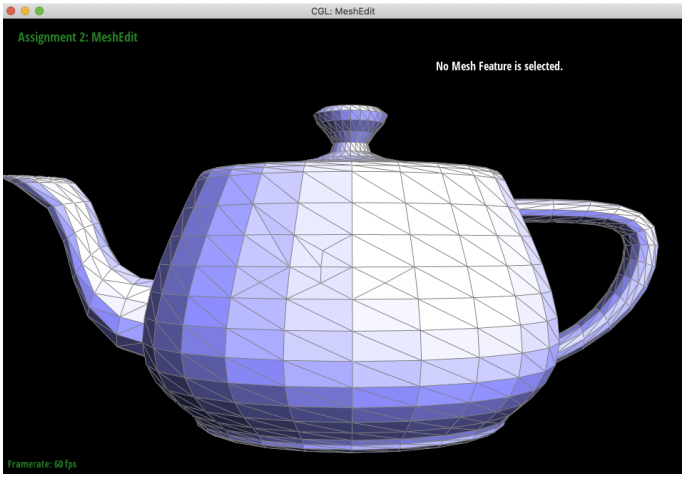
before



after edge split



before



after edge split and edge flip

Task 6

requirements

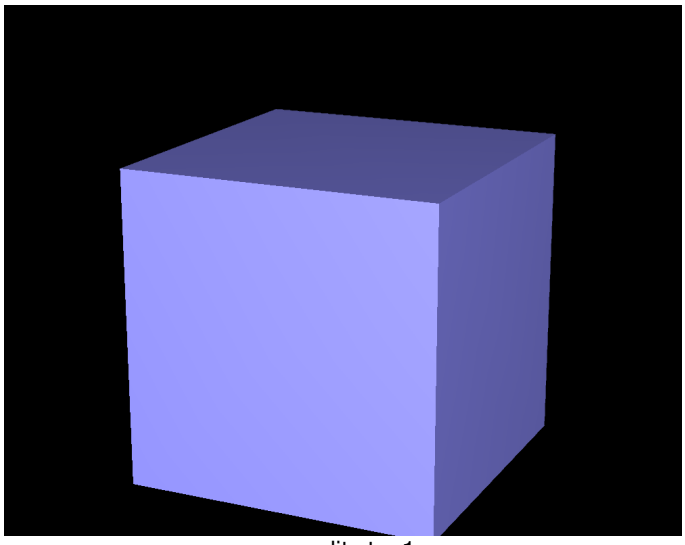
- Briefly explain how you implemented the loop subdivision and describe any interesting implementation / debugging tricks you have used.
- Take some notes, as well as some screenshots, of your observations on how meshes behave after loop subdivision. What happens to sharp corners and edges? Can you reduce this effect by pre-splitting some edges?
- Load dae/cube.dae. Perform several iterations of loop subdivision on the cube. Notice that the cube becomes slightly asymmetric after repeated subdivisions. Can you pre-process the cube with edge flips and splits so that the cube subdivides symmetrically? Document these effects and explain why they occur. Also explain how your pre-processing helps alleviate the effects.

Briefly explain how you implemented the loop subdivision and describe any interesting implementation / debugging tricks you have used.

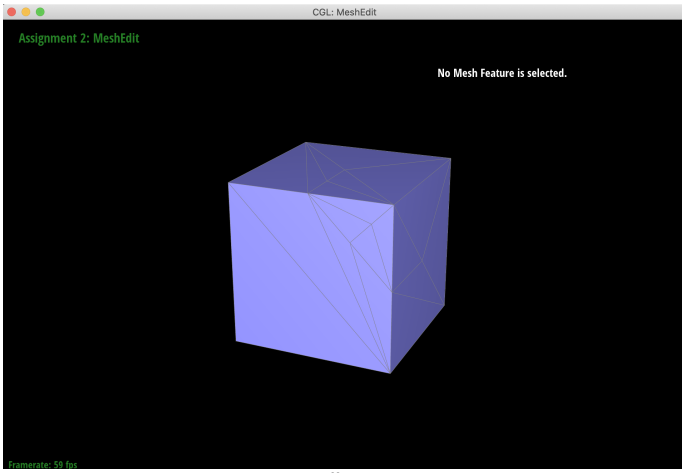
I used the method suggested by the project spec. Let me summarize it. We have three steps. First, we iterate through all the vertices and determine their new positions. We perform this computation before subdivision since this makes it much easier. Something worth noting is that we keep track of the old edges and vertices, since we will start splitting in the next step. Second, we begin the split, with differentiation between old and new edges to avoid a infinite loop/ We first go through existing edges and only split old edges. Then, we flip any edges that connects a old vertex with a new vertex. Lastly, we update the position of our vertices. I debug through print statements to make sure the amount of vertices and edges I am iterating through is correct.

Take some notes, as well as some screenshots, of your observations on how meshes behave after loop subdivision. What happens to sharp corners and edges? Can you reduce this effect by pre-splitting some edges?

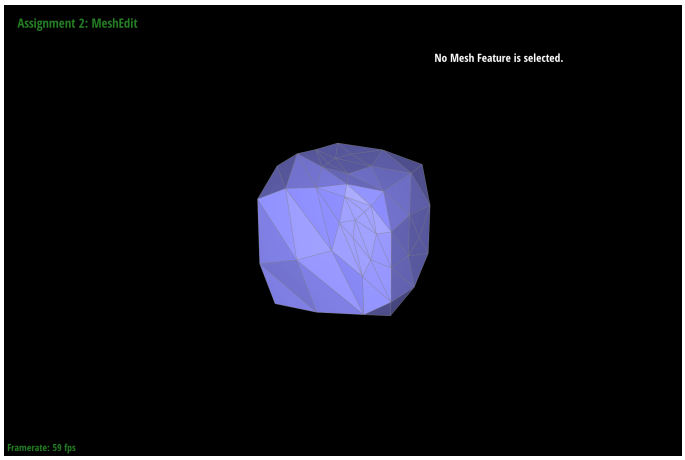
I noticed a few things. First of all, sharp corners and edges are smoothed out. This is probably because when we subdivide, the sharp corners are spread even into several edging, acting like a smoothing. I also noticed the mesh volume is getting smaller and smaller as we subdivide. I realized that pre-splitting helps maintain some corners and sharp edges as shown in the last two images below.



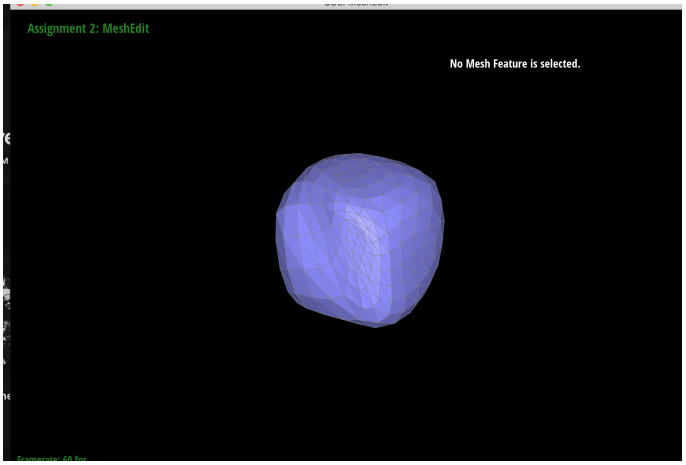
presplit:step1



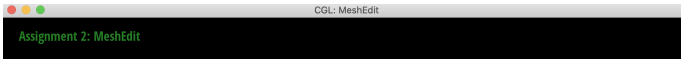
presplit:step2

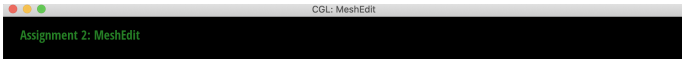


presplit:step3



presplit:step4



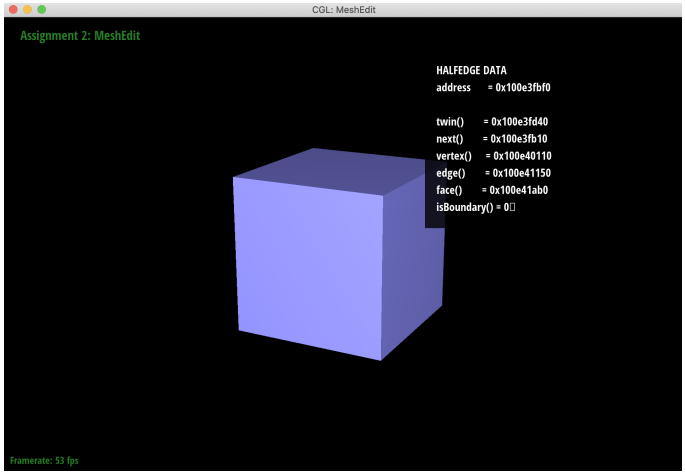


Load dae/cube.dae. Perform several iterations of loop subdivision on the cube. Notice that the cube becomes slightly asymmetric after repeated subdivisions. Can you pre-process the cube with edge flips and splits so that the cube subdivides symmetrically? Document these effects and explain why they occur. Also explain how your pre-processing helps alleviate the effects.

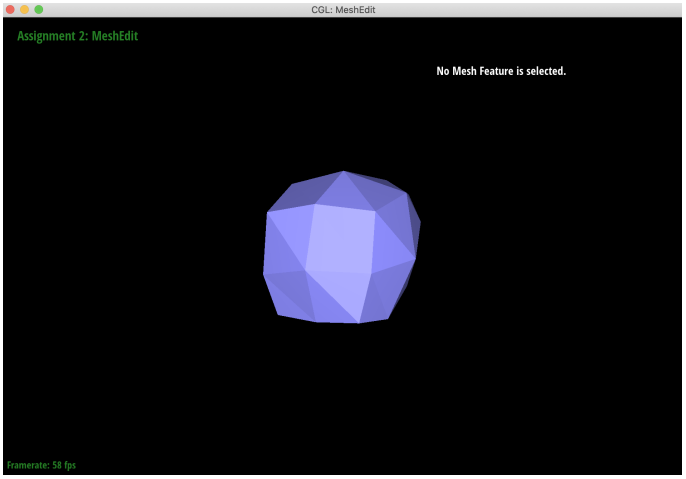
We can pre-process the cube by splitting all of the edges on each face other than the ones on the boundary of each face. In other words, we make sure we have 4 identical triangles on each face of the cube. This helps make the cube subdivision symmetrical.

The reason we see asymmetry before is that before we only have 2 triangles on each face, meaning that there will be specific corners that will draw multiple edges, making it overemphasized and pointy. Our preprocess helps making sure to eliminate these corners. Although as you can see it is still not perfectly spherical, we have made it symmetrical.

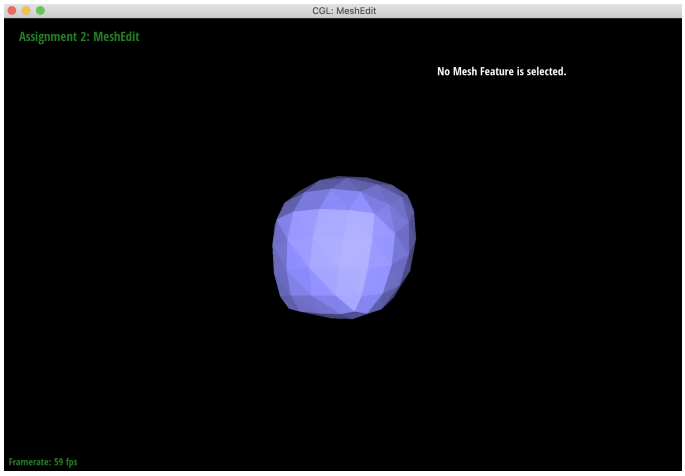
Before preprocessing



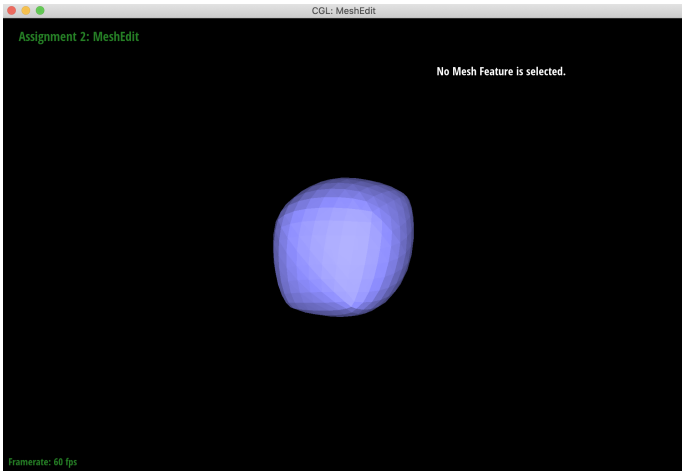
asym:step1



asym:step2

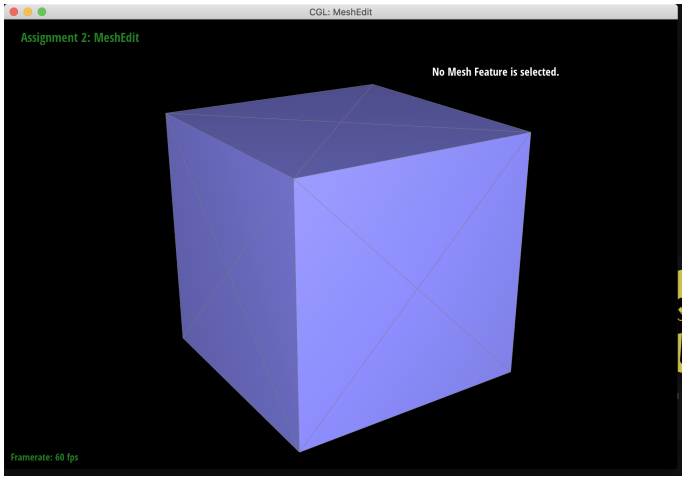


asym:step3

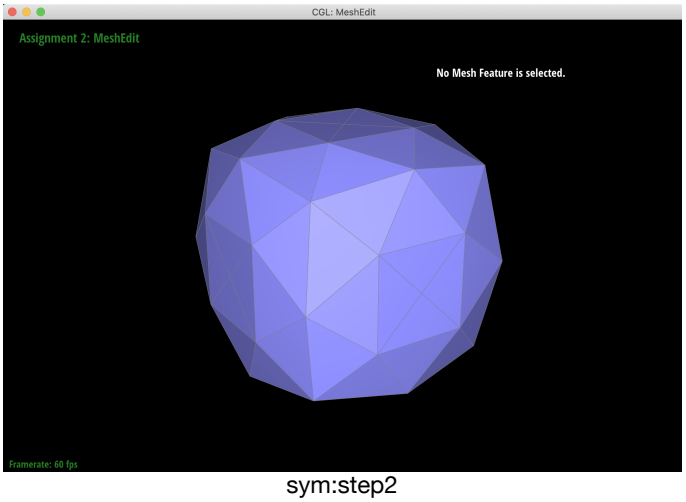


asym:step4

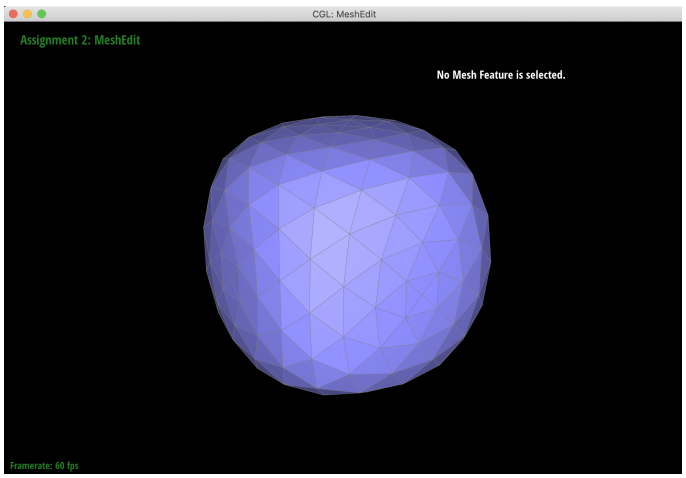
After preprocessing



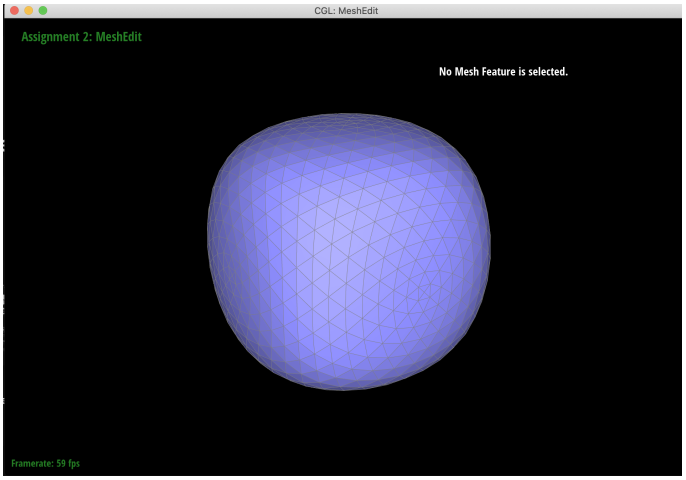
sym:step1



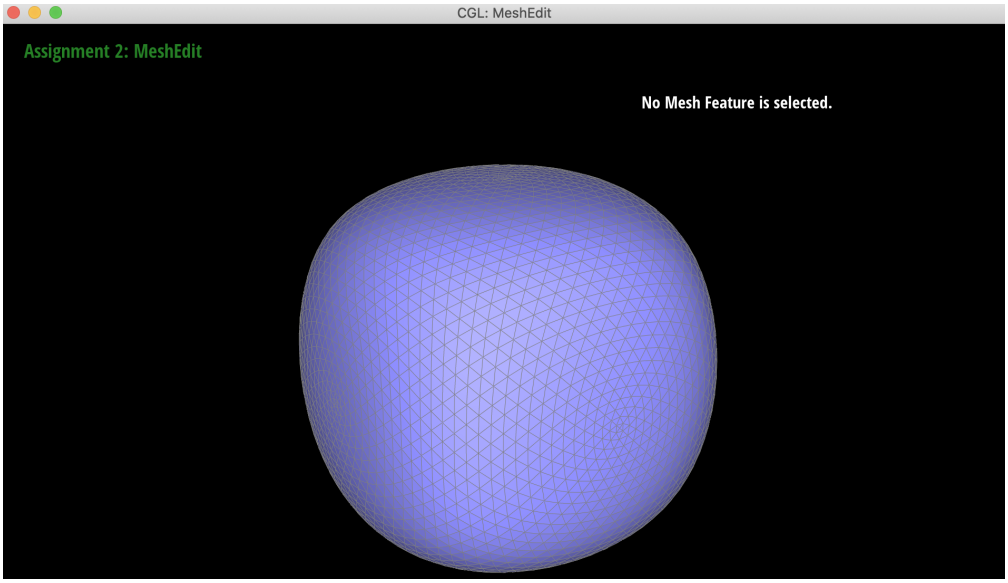
sym:step2



sym:step3



sym:step4



In [ ]: