CS 184: Computer Graphics and Imaging, Spring 2020

Project 2: Mesh Editor

Ethan Gnibus, CS184-gutter

Overview

Give a high-level overview of what you implemented in this project. Think about what you've built as a whole. Share your thoughts on what interesting things you've learned from completing the project.

In this project, I built some systems that are helpful to do geometric modelling in 2d and 3d. I implemented Bezier Curves, Bezier Surfaces which help turn control points into smooth lines. I also updated a 3d mesh codebase to support smooth shading, edge flips, edge splits, and loop subdivision to upsample the mesh. I really like Blender so I thought this project was really cool.

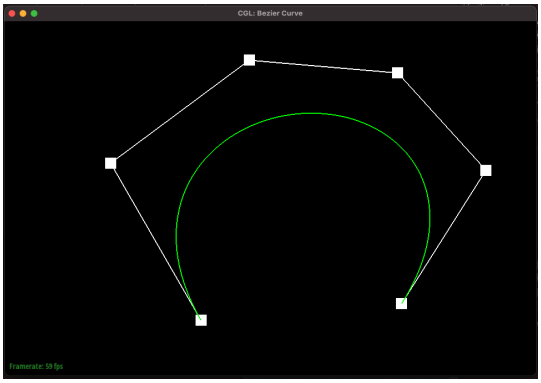
Section I: Bezier Curves and Surfaces

Part 1: Bezier curves with 1D de Casteljau subdivision

Briefly explain de Casteljau's algorithm and how you implemented it in order to evaluate Bezier curves.

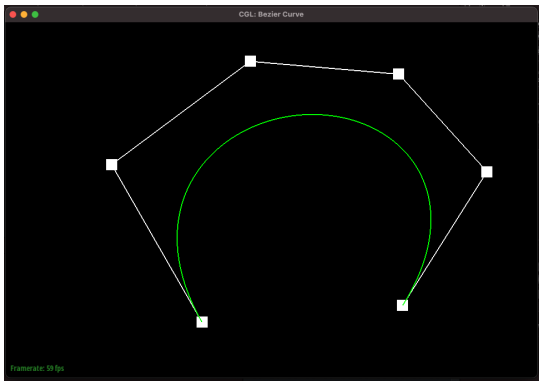
Casteljau's algorithm takes in a set of control points and outputs a smooth curve (Bezier curve) that can be manipulated by moving the control points. I implemented this algorithm by recursively finding midpoints between pairs of control points. This ratio was defined as t, where t is in the range [0, 1]. I start with N control points. between each pair of control points, I find a new point that is at the position t would be if the first old point is 0 and the second old point is 1 (the new point is somewhere on the line between the old point and the new point, governed by t). I then repeat this same process on these N-1 new points until I am left with a single point. This point represents the point "t" of the way through our final Bezier curve. I draw this point to the screen. I then repreat this process with a dense number of "t's" so that I am left with a curve!

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.

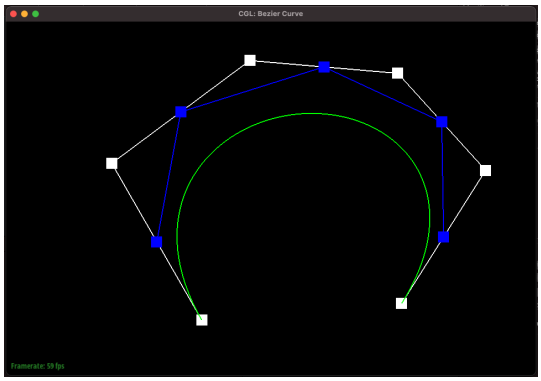


We will apply de Casteljau's to the points above to get the green line above.

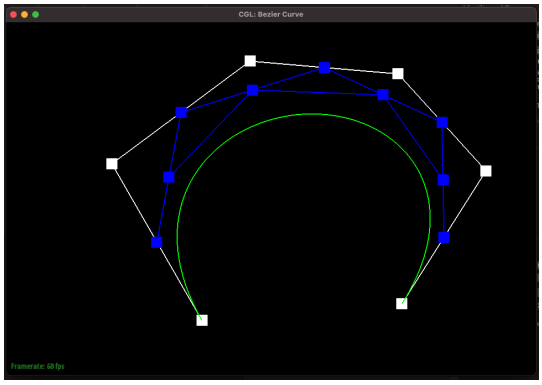
Show screenshots of each step / level of the evaluation from the original control points down to the final evaluated point. Press E to step through. Toggle C to show the completed Bezier curve as well.



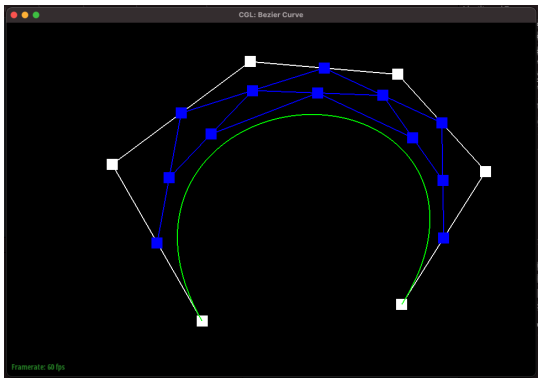
Zero iterations.



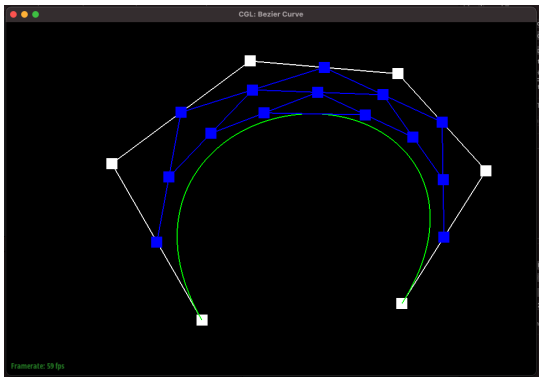
One iteration.



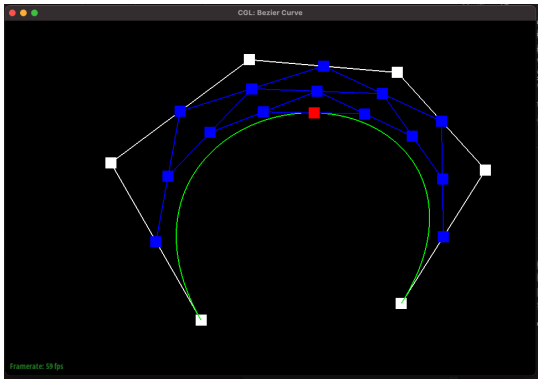
Two iterations.



Three iterations.



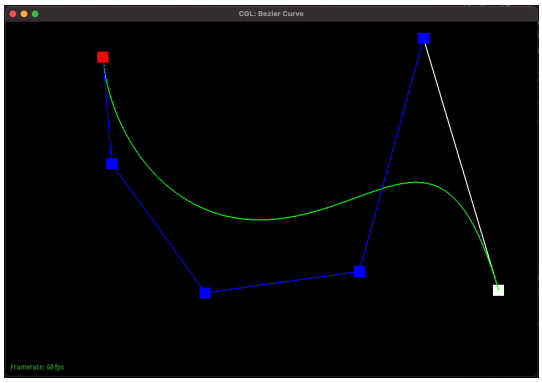
Four Iterations.



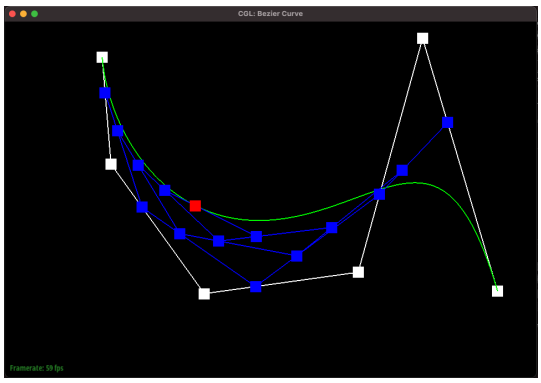
Five iterations.

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

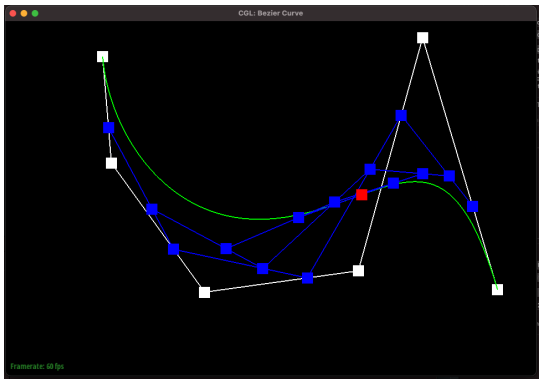
Below I will show a series of pictures of the same curve evaluated at different values of t.



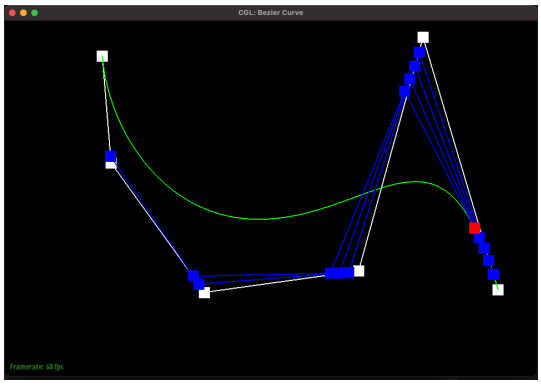
t at 0.



t around 0.3.



t around 0.7.



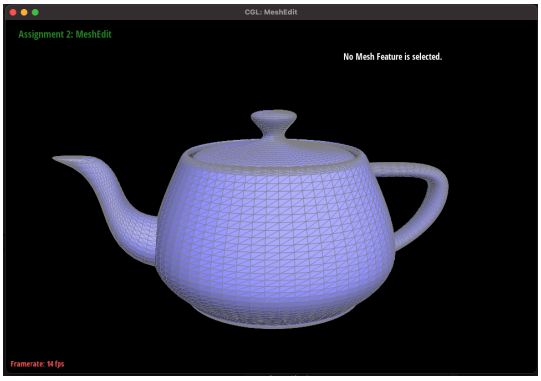
t close to 1.

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

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

Above we just showed why de Casteljau's is useful to represent 2d Bezier curves. We could represent 3d Bezier surfaces by applying de Casteljau's to "patches" of points. To do this we break up our patch of points into its rows, and apply de Casteljau to each individual row until we get the point specifiec at (u,v). To do this, we have the function BezierPatch::evaluate(...). In evalueate, we iterate through the rows of our patch and use BezierPatch::evaluate1D(...) specified at u to get back a list of control points at u. We then use BezierPatch::evaluate1D(...) specified at v to get the point at v that lies on the line of control points specified at u. BezierPatch::evaluate1D(...) repeatedly applies de Casteljau using BezierPatch::evaluateStep(...) until we get one point back. BezierPatch::evaluateStep(...) calculates one step of de Casteljau just like in Part 1.

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



The teapot above was construced by calculation Bezier Surfaces.

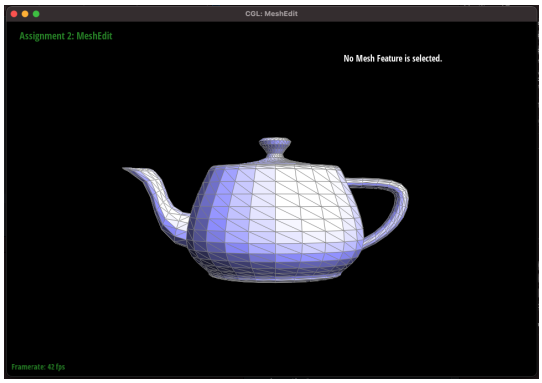
Section II: Sampling

Part 3: Average normals for half-edge meshes

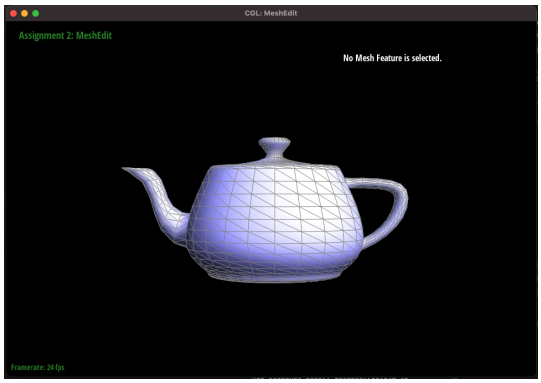
Briefly explain how you implemented the area-weighted vertex normals.

To calculate area-weighted vertex normals, I found a triangle's three vertices using the mesh datastructures. Then I used their positions to get two vectors that I could use to find the area of the triangle. I used sqrt(x^2 + y^2 + z^2) to find the area, then multiplied the area by the cross product of the the two vectors to get a perpendicular vector. I then took the norm of the perpendicular vector and did this to every triangle that the provided vertex is a part of. I then took the average of all these perpendicular vectors to get the average normal.

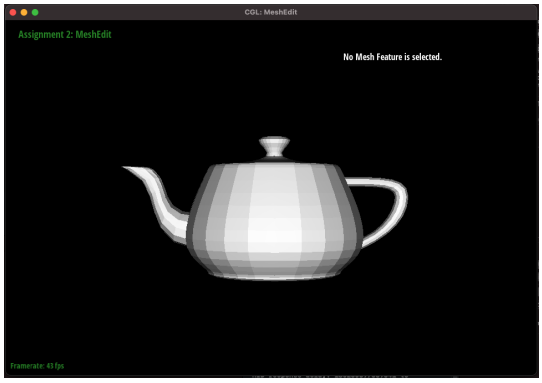
Show screenshots of dae/teapot.dae (not .bez) comparing teapot shading with and without vertex normals. Use Q to toggle default flat shading and Phong shading.



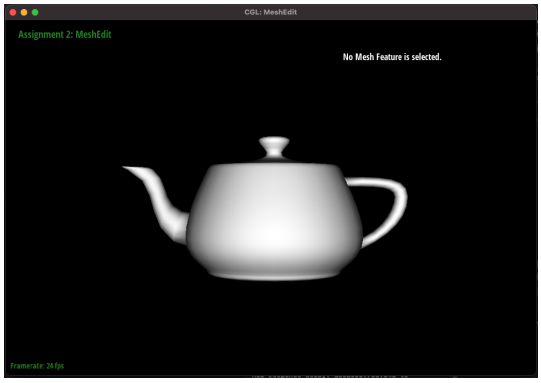
Flat shading with vertex normals.



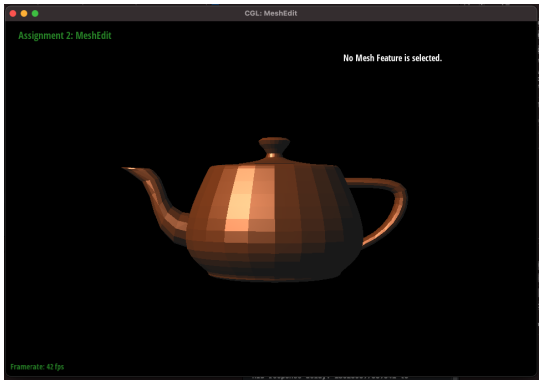
Flat shading with vertex normals.



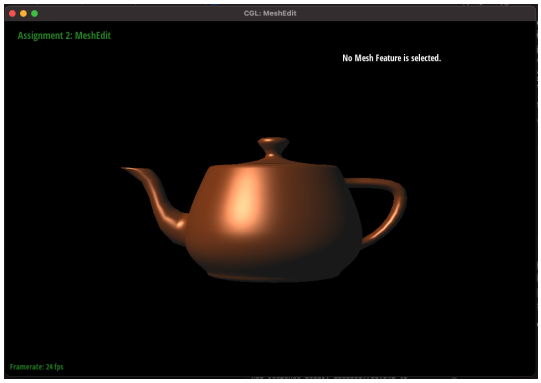
Flat shading with white shader.



Phong shading with white shader.



Flat shading with brown shader.

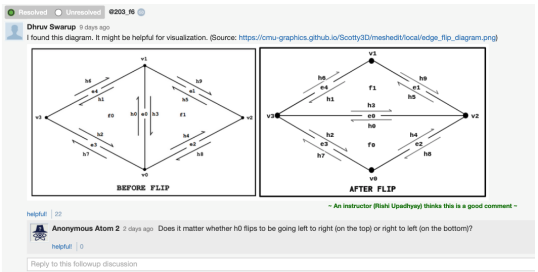


Phong shading with brown shader.

Part 4: Half-edge flip

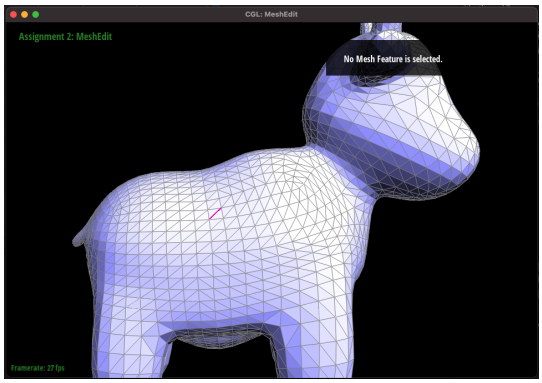
Briefly explain how you implemented the edge flip operation and describe any interesting implementation / debugging tricks you have used.

To implement edge flips, I loaded all vertices based on the left side of the image below using the mesh data structures. I then reassigned pointers based on the right side of the image. I found visually flipping edges and looking how the triangles line up on the mesh useful for debugging.

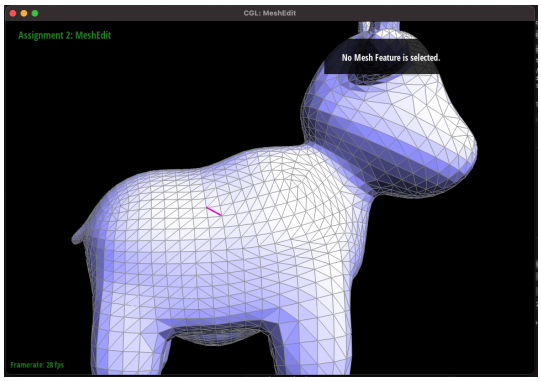


This piazza post was useful.

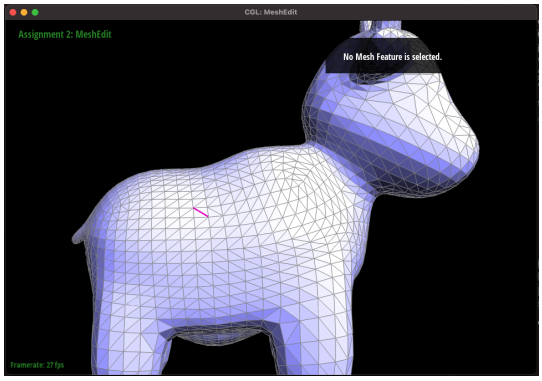
Show screenshots of a mesh before and after some edge flips.



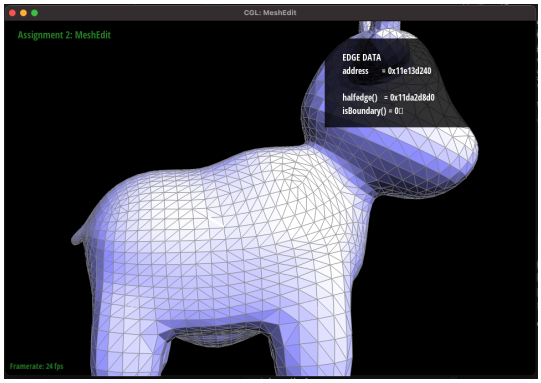
Before first diagonal flip.



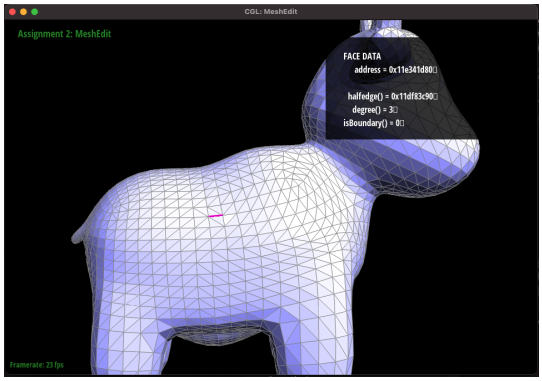
After first diagonal flip.



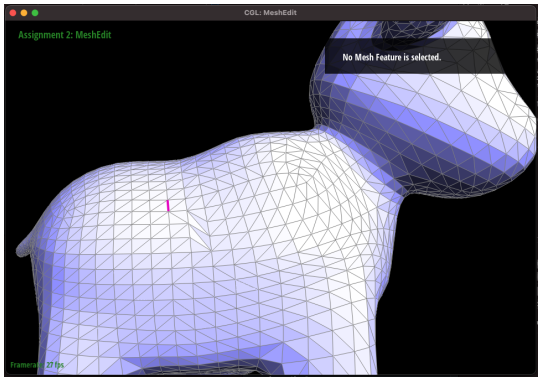
After second diagonal flip.



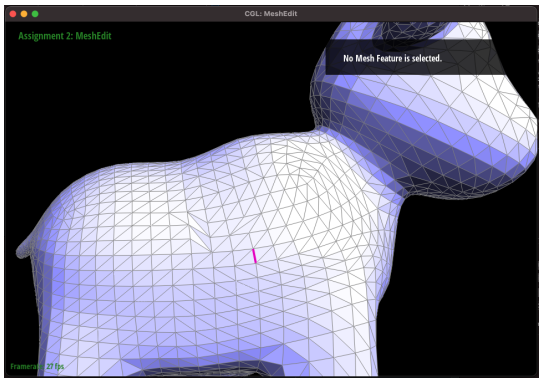
After third diagonal flip.



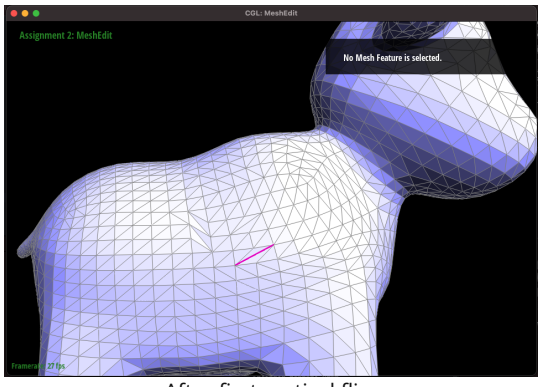
Before first horizontal flip.



After 2 consecutive horizontal flips.



Before first vertical flip.



After first vertical flip.

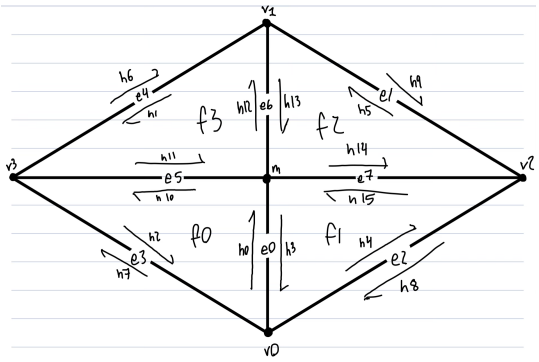
Write about your eventful debugging journey, if you have experienced one.

I had a really hard time figuring out why my program was stuck in an infinate loop. I spent hours on piazza and ended up finding out that I declared everything as some variation of _____Iter& varibleName = ...; instead of _____Iter& varibleName = ...;. This meant that I was resetting pointers wrong when I did things like v0->halfedge() = h4;. Removing all the &'s fixed my problem!

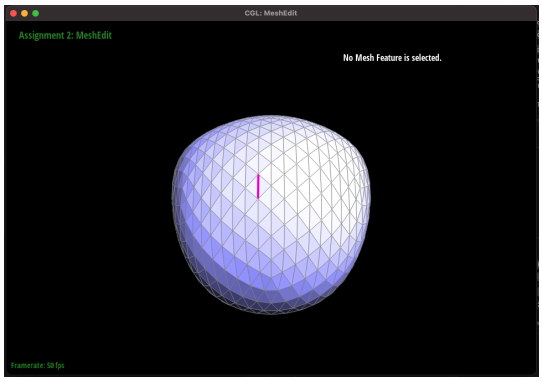
Part 5: Half-edge split

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

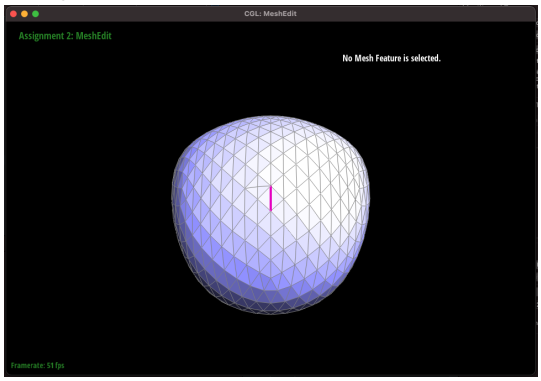
I used the same methodology as edge flip, but I used the diagram below as the right side of the image.



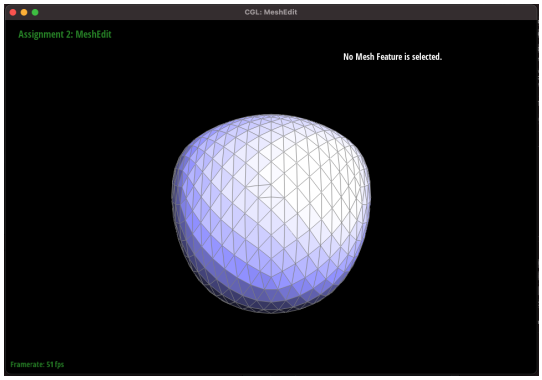
I drew this one myself!



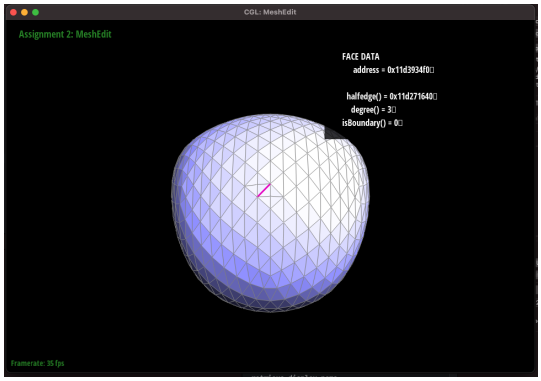
Before first split.



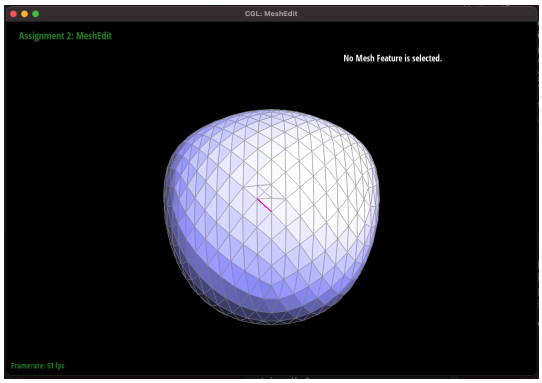
After first split.



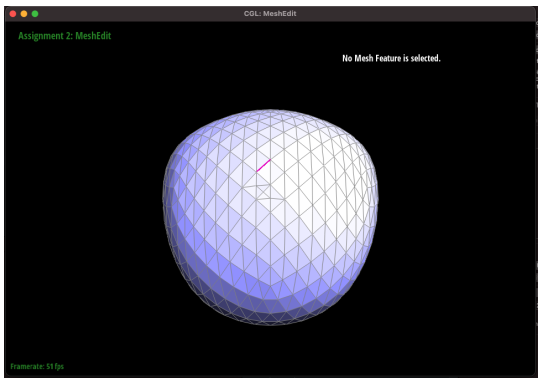
After second split.



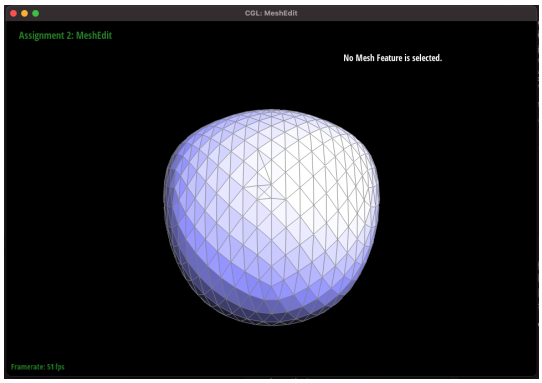
Before third split.



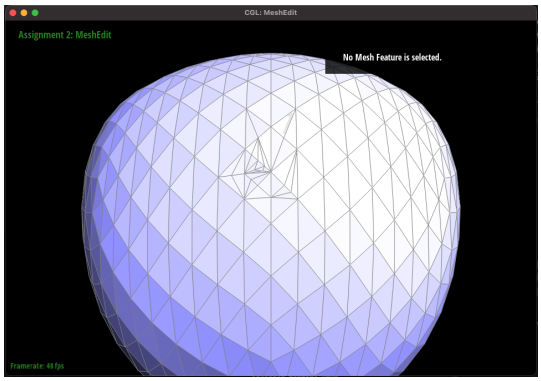
After third split.



Before horizontal split.

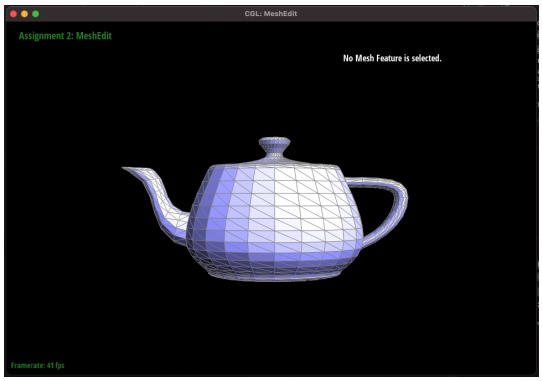


After horizontal split.

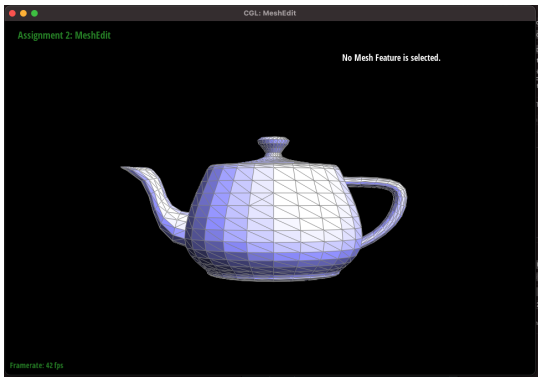


After lots of splits.

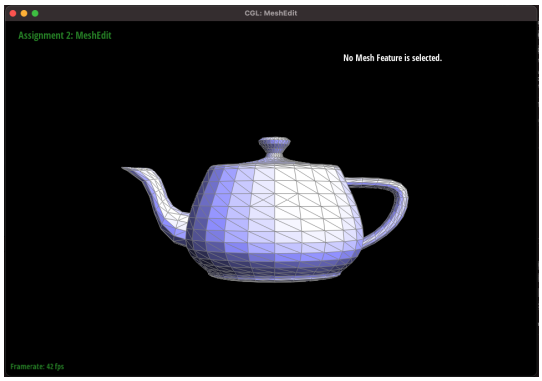
Show screenshots of a mesh before and after a combination of both edge splits and edge flips.



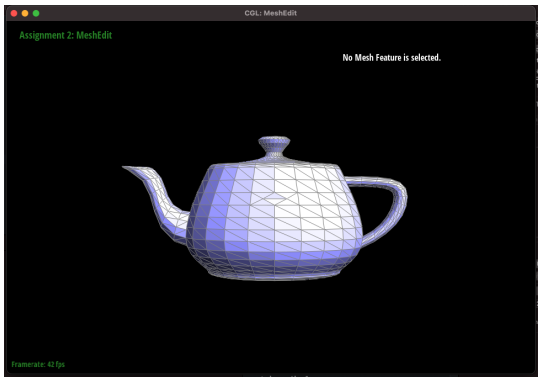
Before first split.



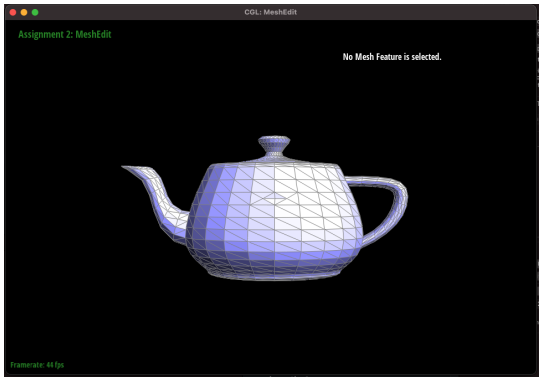
After first split.



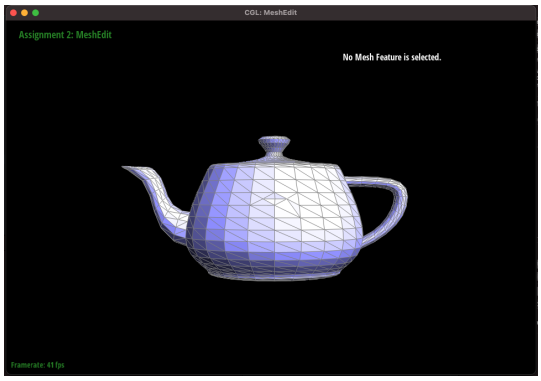
After second split.



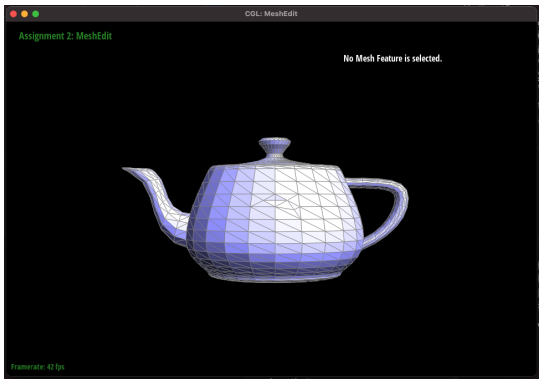
Before first flip.



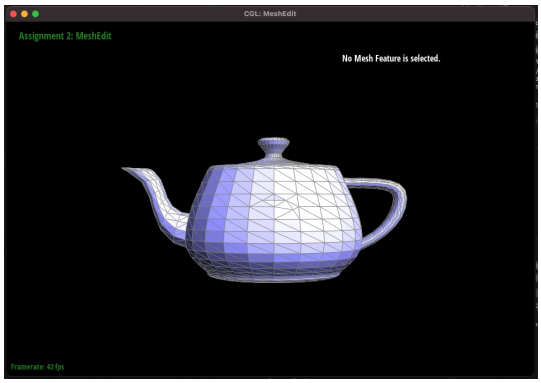
After third split.



After second flip.



After fourth split.



After third flip.

Write about your eventful debugging journey, if you have experienced one.

I actually ended up getting this problem first try. I think it's because I used the convention from the diagram I drew.

If you have implemented support for boundary edges, show screenshots of your implementation properly handling split operations on boundary edges.

I did not implement.

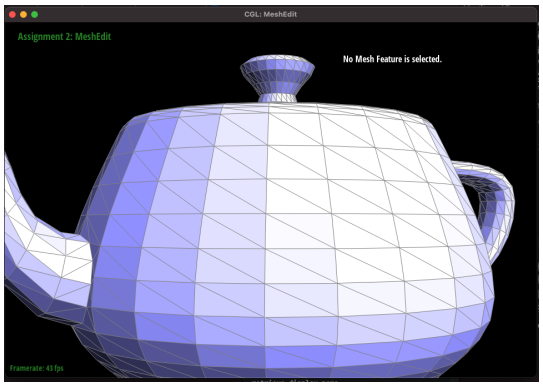
Part 6: Loop subdivision for mesh upsampling

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

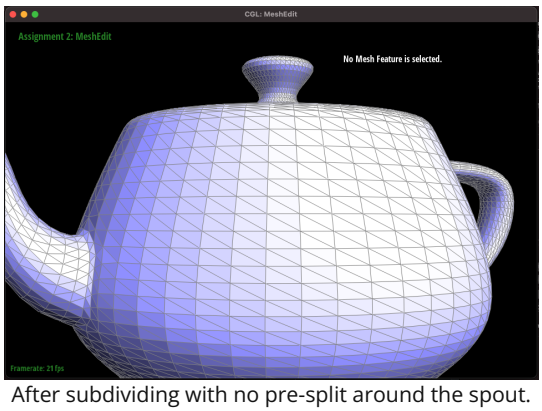
To implement loop subdivision, I iterated through all edges and set all edges and vertices's ->isNew field to false, calculated new vertex locations based on proximal vertices, calculated updated old vertex locations using a different proximal equation, split edges and assingned the midpoint's new location based on what I calculated, flipped new edges that aren't on a preexisting line, then updated the old vertex locations.

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?

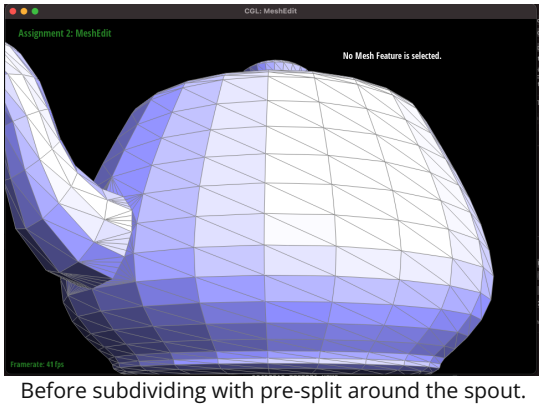
I observed that after subdivision with no pre-split, sharp corners and edges are rounded off. Differences in meshes subdivided with and without a pre-split can be seen below.



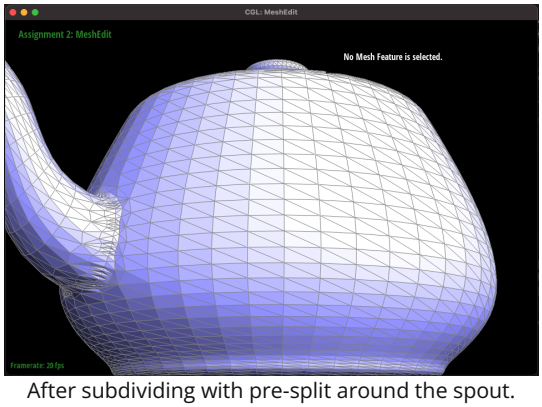
Before subdividing with no pre-split around the spout.



After subdividing with no pre-split around the spout.



Before subdividing with pre-split around the spout.

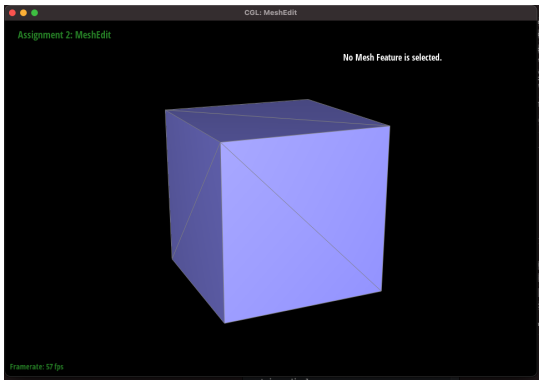


After subdividing with pre-split around the spout.

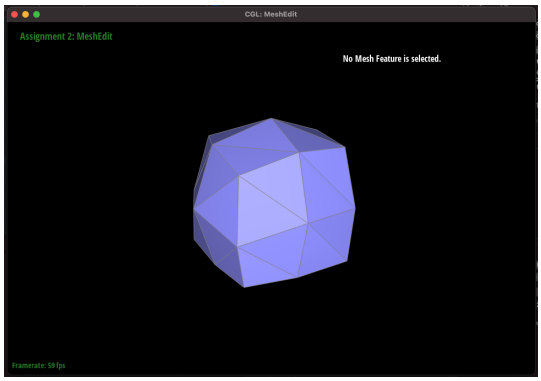
As we can see, in areas I pre-split, some more of the high frequency change is preserved. I CAN reduce this effect by presplitting. The subdivided mesh with pre-splitting looks far closer to the original mesh than the subdivided mesh with no pre-splitting. This seemed to happend with both sharp corners and edges in my research.

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.

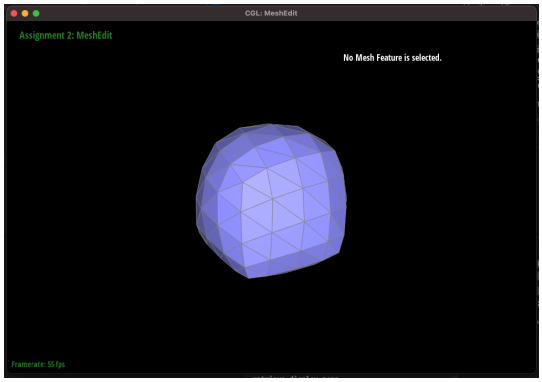
With our algorithm for subdivision, we find new vertex positions based on old vertex positions and move old vertex positions based on the positions of adjacent vertices. A mesh with low numbers of vertices will lose lots of information about where vertices originally are as the update and will dramatically change as seen below.



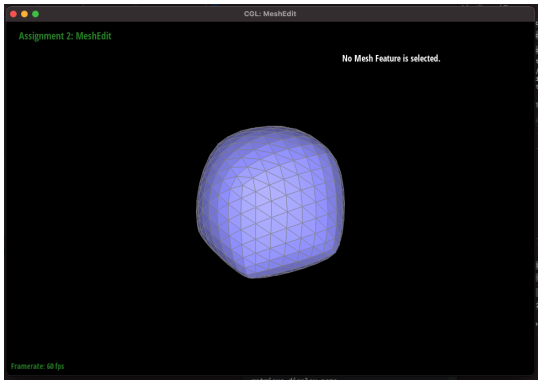
No Subdivision.



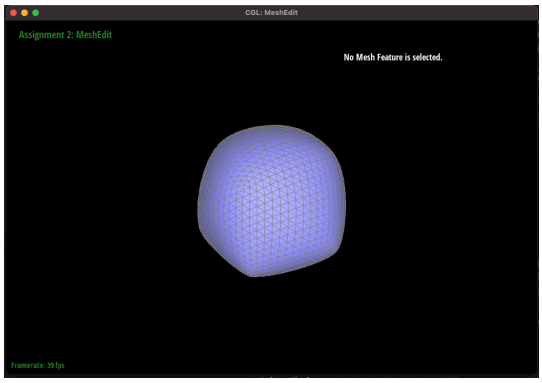
Subdivision 1.



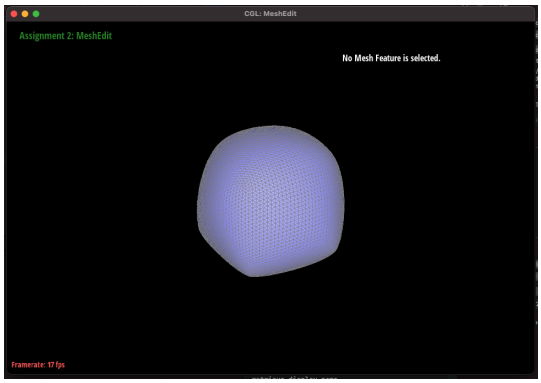
Subdivision 2.



Subdivision 3.

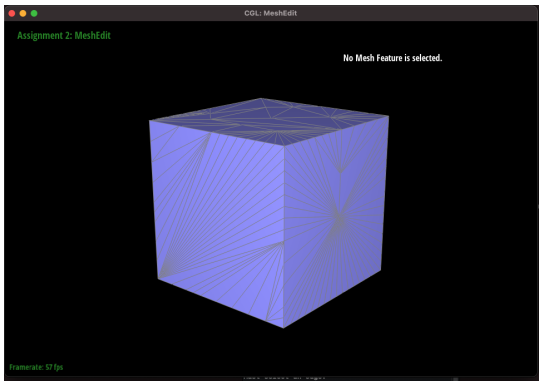


Subdivision 4.

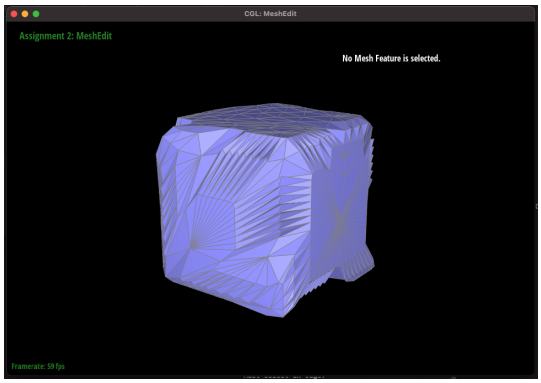


Subdivision 5.

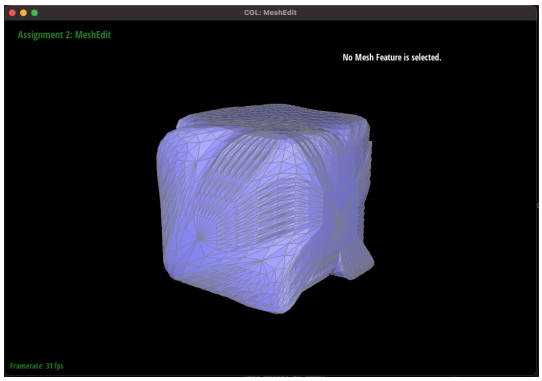
To mitigate this, we will repetitively split so neighboring vertices update on adjacent vertices that are closer together. This will mitigate a good amount of the subdivision rounding as seen below.



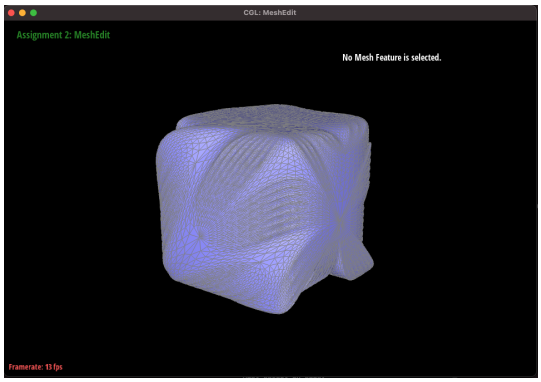
No Subdivision.



Subdivision 1.



Subdivision 2.



Subdivision 3.

As we could see, the cube's shape is retained much better. It would look even better if I split edges in a uniform way!

If you have implemented any extra credit extensions, explain what you did and document how they work with screenshots.

I did not.

Section III: Optional Extra Credit

Part 7: Design your own mesh!

Save your best polygon mesh as partsevenmodel.dae in your docs folder and show us a screenshot of the mesh in your write-up.

Not doing.

Include a series of screenshots showing your original mesh and your mesh after one and two rounds of subdivision. If you have used custom shaders, include screenshots of your mesh with those shaders applied as well.

Not doing.

Describe what you have done to enhance your mesh beyond the simple humanoid mesh described in the tutorial.

Not doing.

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