CS 184: Computer Graphics and Imaging, Spring 2022

Project 2: Mesh Editor

Alina Wang & David Wei, CS184-!!

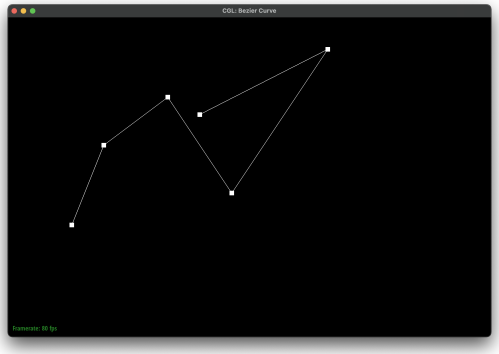
Overview

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.

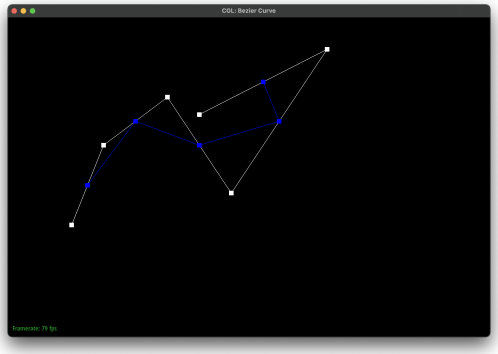
Section I: Bezier Curves and Surfaces

Part 1: Bezier curves with 1D de Casteljau subdivision

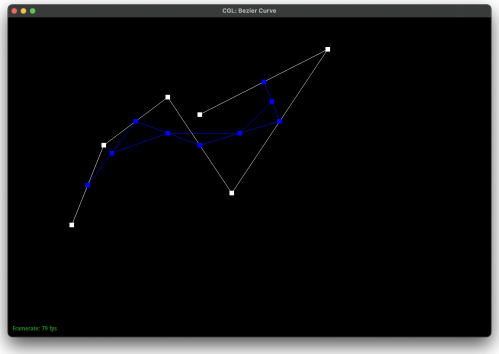
De Casteljau's algorithm locates a point on a line between two points using linear interpolation. Under recursion, de Casteljau's algorithm will reduce all points generated into a single point, which the Bezier curve will travel through. For all possible values of t, which defines the linear interpolation constant, you can "trace" along the original points/lines and generate all points for which the Bezier curve will travel through.



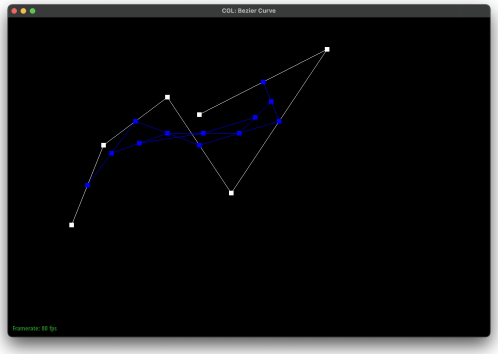
curve3-step0



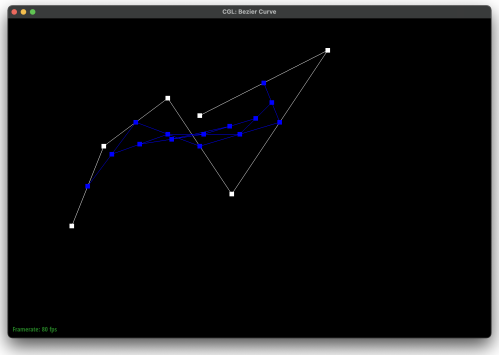
curve3-step1



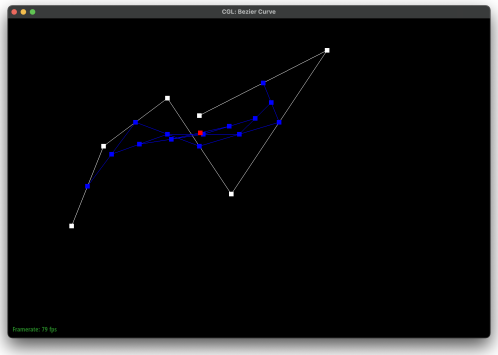
curve3-step2



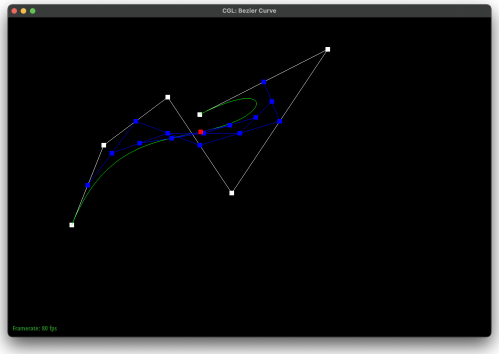
curve3-step3



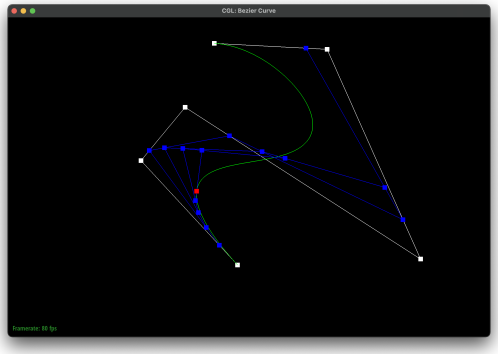
curve3-step4



curve3-step5



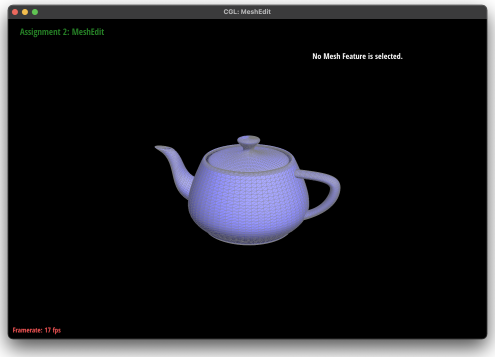
curve3-complete



curve3-variant

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

De Casteljau's algorithm can be extended to Bezier surfaces by applying its recursive properties two-fold. First we calculate an intermediate collection of Bezier points, which correspond to a given "row" in our case, and apply de Casteljau's algorithm on these points. This final point corresponds to the point on our final bezier surface, using the collection of control points and appropriate linear interpolation parameters u and v.

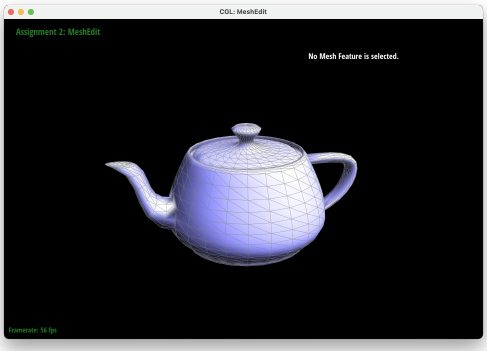


teapotg

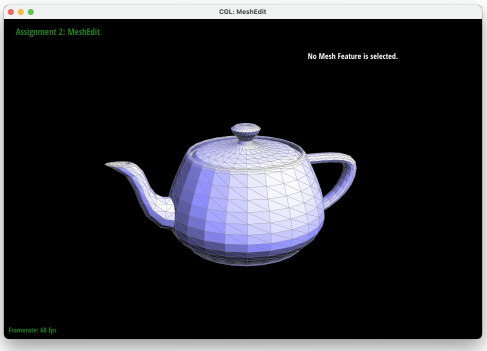
Section II: Sampling

Part 3: Average normals for half-edge meshes

To find the normal vector of a vertex, we calculated all the normal vectors of neighboring faces by taking the cross product of 2 vectors to find the normal of every face and summing together the normal vectors. Taking the cross product returns an area weighted normal vector so it is enough to use the results of the cross product when averaging between all neighboring normal vectors. Finally, we normalized the summed normal vectors in order to avg the final vertex normal vector.



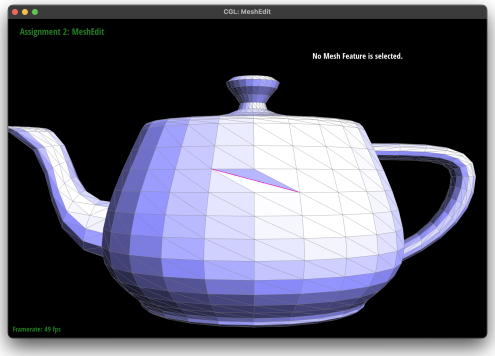
Teapot with Shading



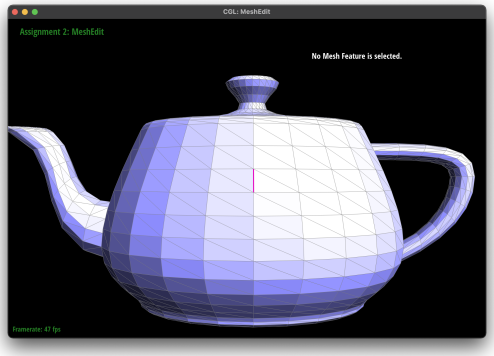
Teapot without Shading

Part 4: Half-edge flip

To flip the edge of our triangle, we referenced a handy illustration from CMU. Fundamentally, all we needed to do was rearrange some pointers, and reset some of the neighbors for our half-edges. In order to minimize chances for bugs/errors, we rearranged the pointers and neighbors of every vertex, face, edge, and halfedge, even if this meant reassigning a pointer back to where it was pointing originally.



teapot with original edges

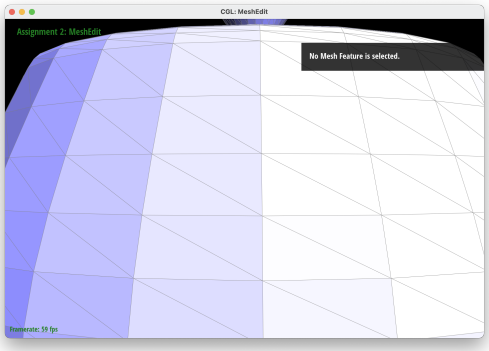


teapot with a flipped edge

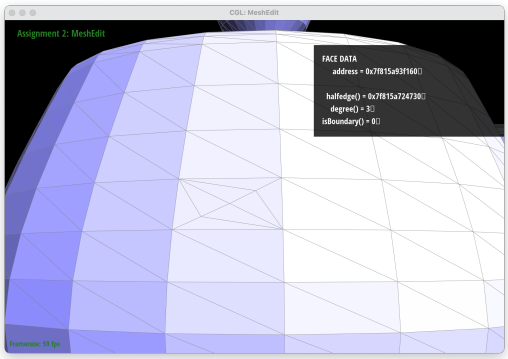
Part 5: Half-edge split

Similar to part 4, we drew out the process of splitting an edge - mapping every new pointer for faces, vertices, halfedges and twins. Having tracked where each new pointer goes, we created 15 new objects(faces, halfedges, vertices, etc) and reassigned existing and new object pointers respectively.

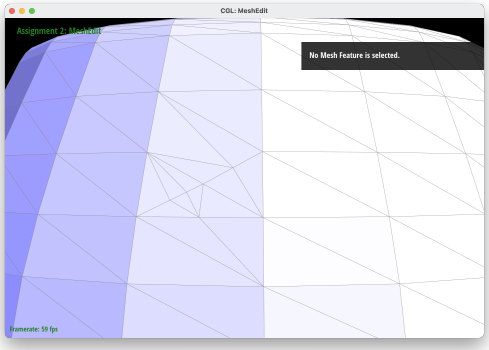
A common bug we ran into was the fact that our new vertex, in the middle of the split, was positioned at the origin. So every edge split resulted in a vertex shooting the middle of the teapot. From reading Piazza, we realized it was because we didn't initialize the position of the new vertex so it was being default set to (0, 0, 0). So we went back and initialized a position and that fixed everything.



Teapot with no flips or splits



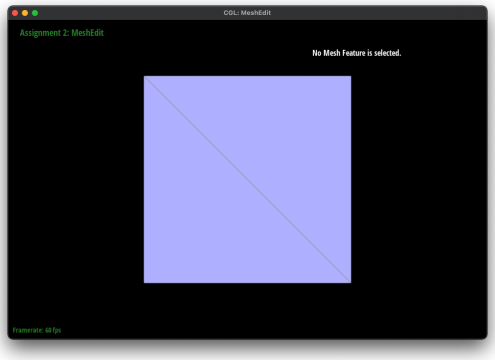
teapot with a splits



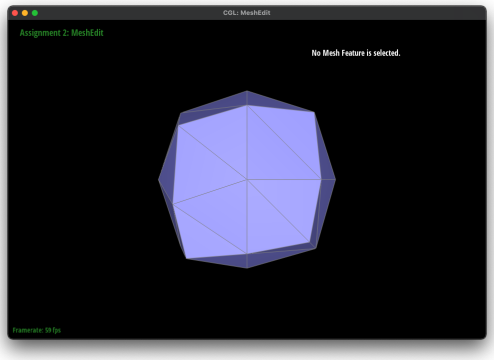
teapot with splits and flips

Part 6: Loop subdivision for mesh upsampling

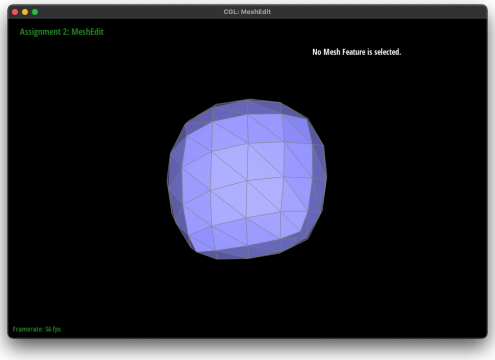
We implemented loop subdivision exactly according to the spec, we first calculate where all the new vertices should go and store them inside of newPosition, then we split and flip edges and set the new positions. In order to debug, we used print debugging to ensure that the positions of vertices are expected (for example, a position of 0,0,0 was typically a red flag).



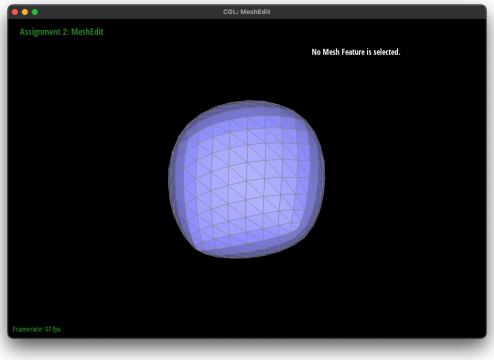
cube with no loop subdivision



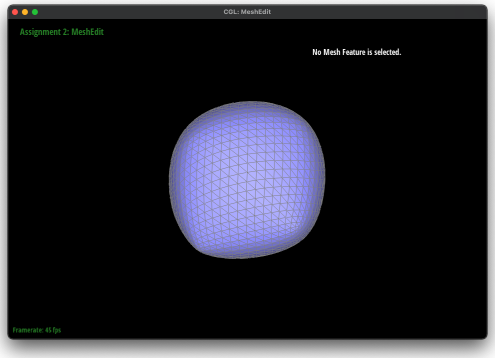
cube with 1 loop subdivisions



cube with 2 loop subdivisions

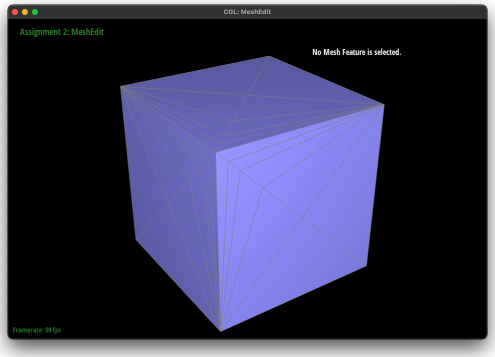


cube with 3 loop subdivisions

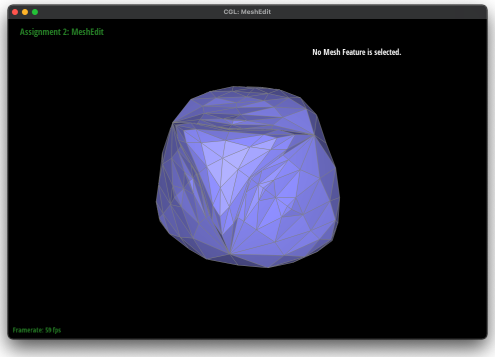


cube with 4 loop subdivisions

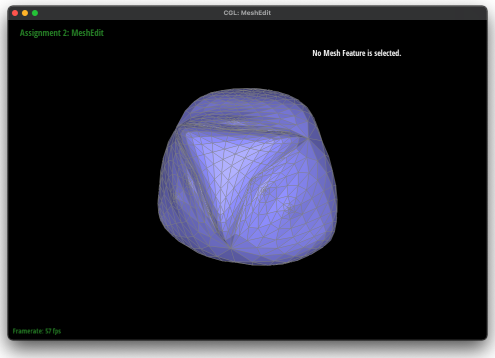
While performing loop subdivisions, we noticed that sharp corners and edges tend to get squashed down / smoothed out, causing the mesh as a whole to get smaller. In order to minimize this, we pre-split corners; we know that the triangles touching a corner will be squashed down, so minimizing the size of the corner triangles will reduce the squashing. We support this idea with the following images. If you focus on the top left of the square, after loop subdivision you will notice that it maintains its shape much better than without preprocessing.



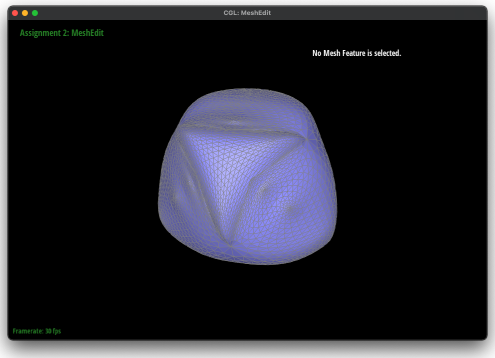
cube with no loop subdivision, preprocessed



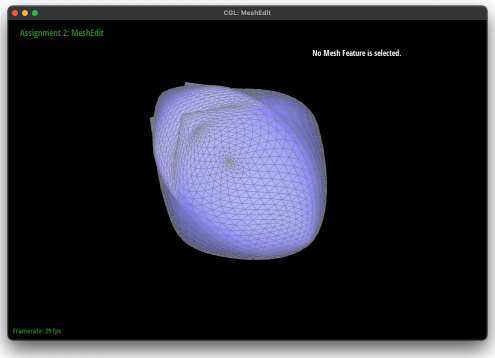
cube with 1 loop subdivisions, preprocessed



cube with 2 loop subdivisions, preprocessed



cube with 3 loop subdivisions, preprocessed



cube with 3 loop subdivisions, preprocessed, original view