Homework 2: Bezier Curves and Triangle Meshes

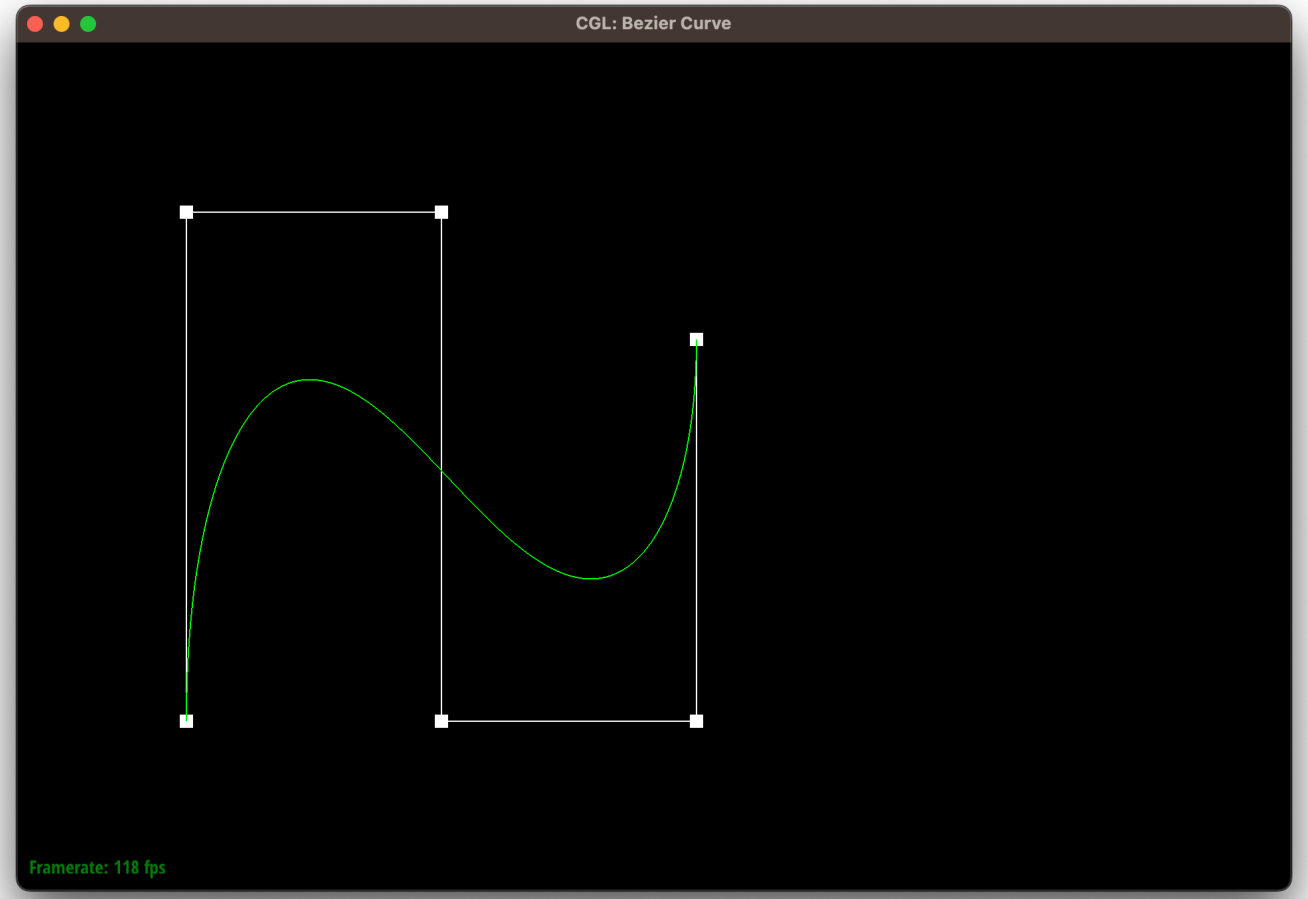
Overview

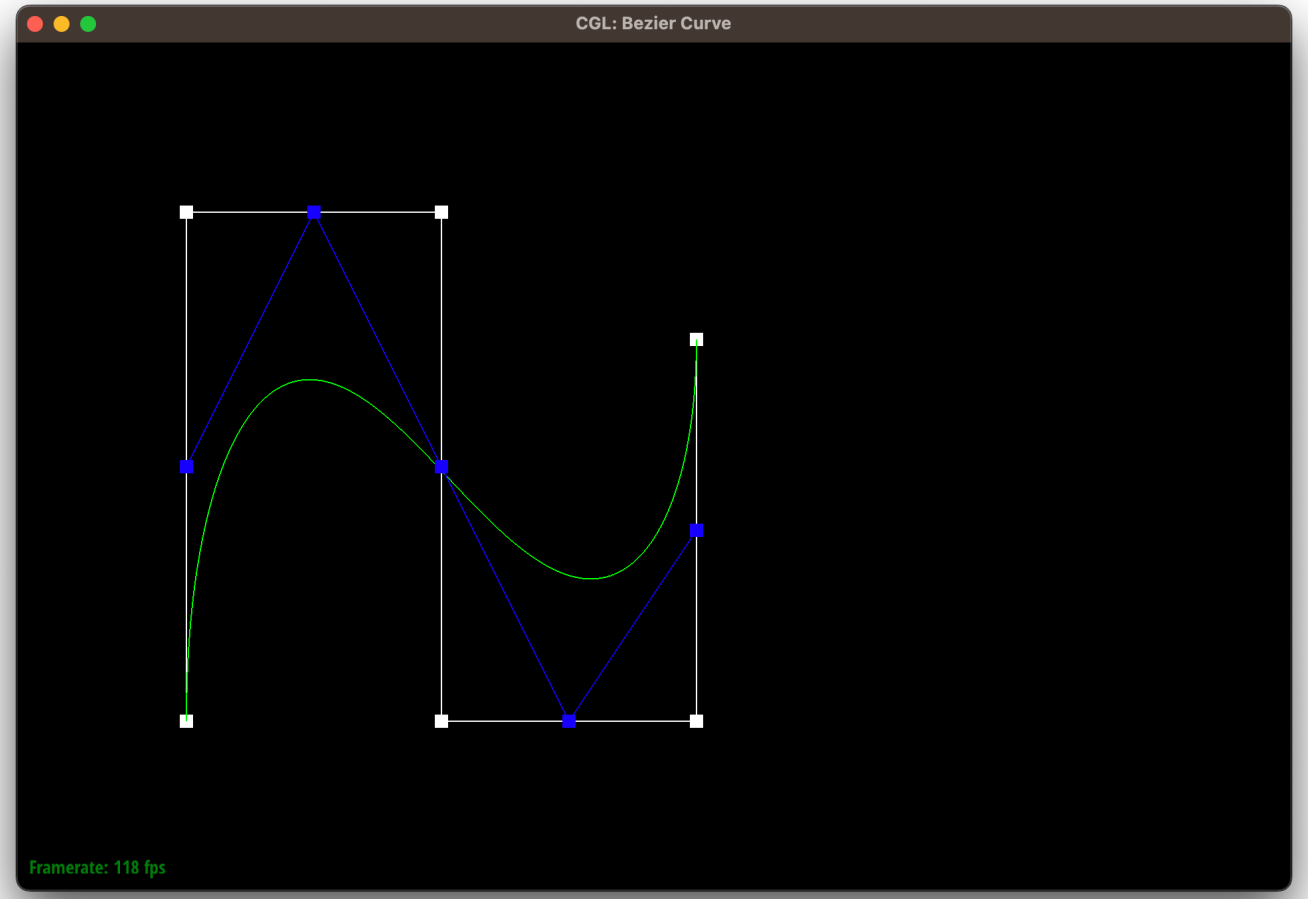
In this homework, I implemented algorithms to render bezier curves and triangle meshes. I used the de Casteljau algorithm to draw 1D bezier curves, and using multiple of these 1D bezier curves, I was able to create bezier surfaces. I implemented area-weighted surface normals to achieve phong shading, and I implemented triangle edge flips, edge splits, and loop subdivision for mesh upsampling. I thought it was interesting to learn more about how objects are rendered in 3D and how smoothing occurs with time and space constraints. Implementing bezier curves and triangle meshes gave me a better understanding and intuition of how they are used in the real world.

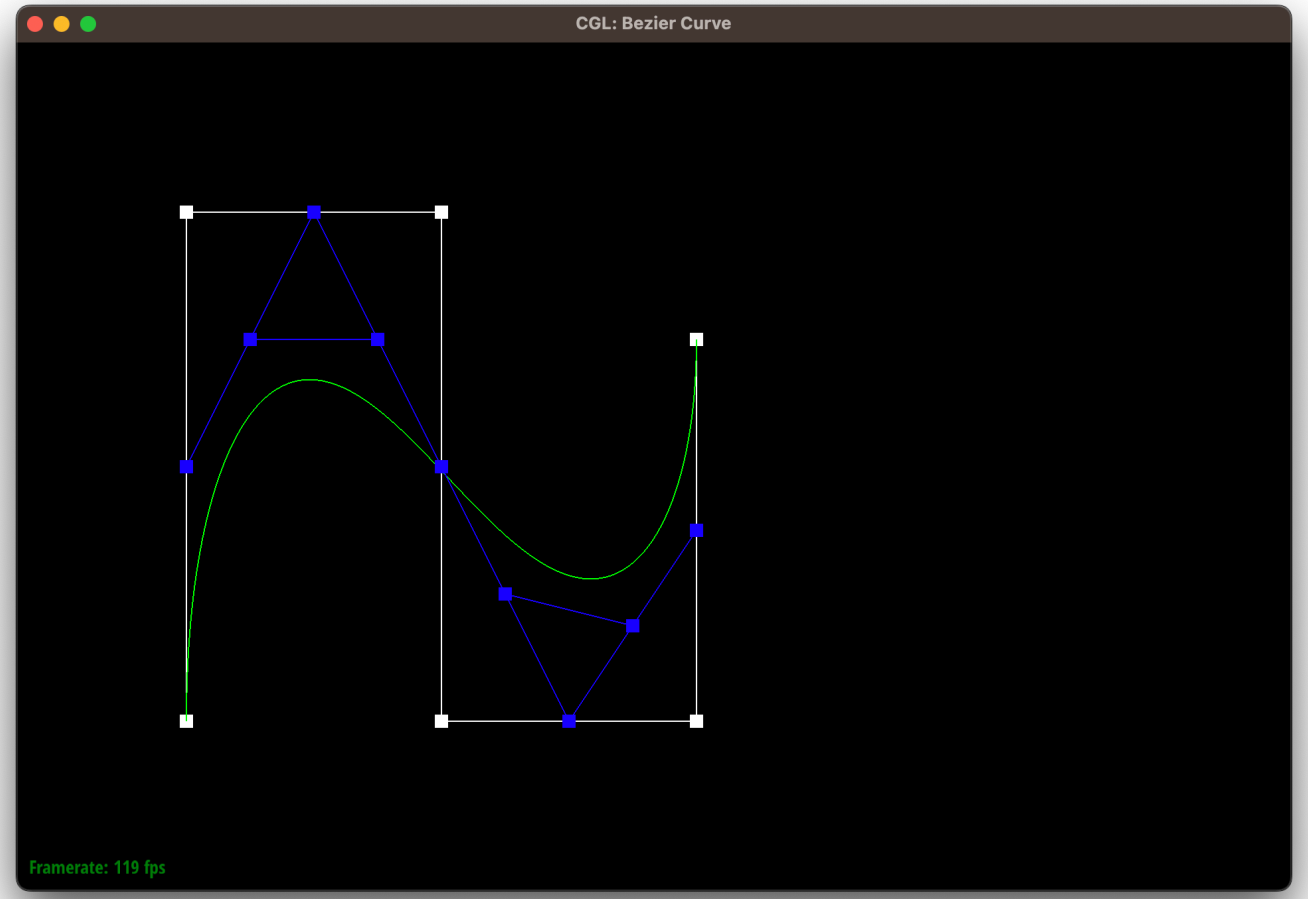
Part 1: Bezier Curves with 1D de Casteljau Subdivision

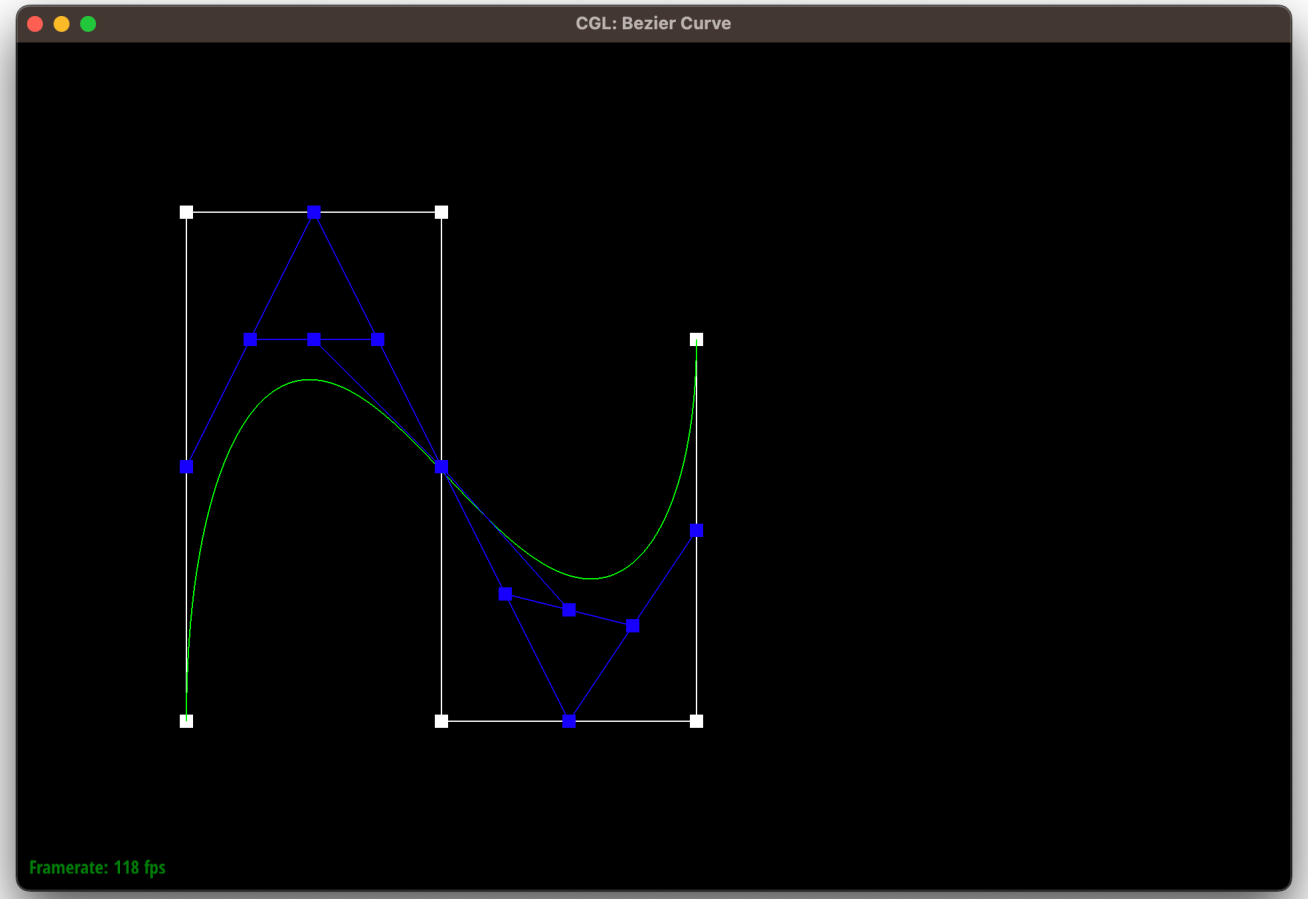
The de Casteljau algorithm is a recursive algorithm that takes the repeated linear interpolation between control points to result in a smooth curve between these points. In order to implement the algorithm, I looped through every control point and took the interpolation between each pair of them. For n points, this interpolation results in n-1 points.

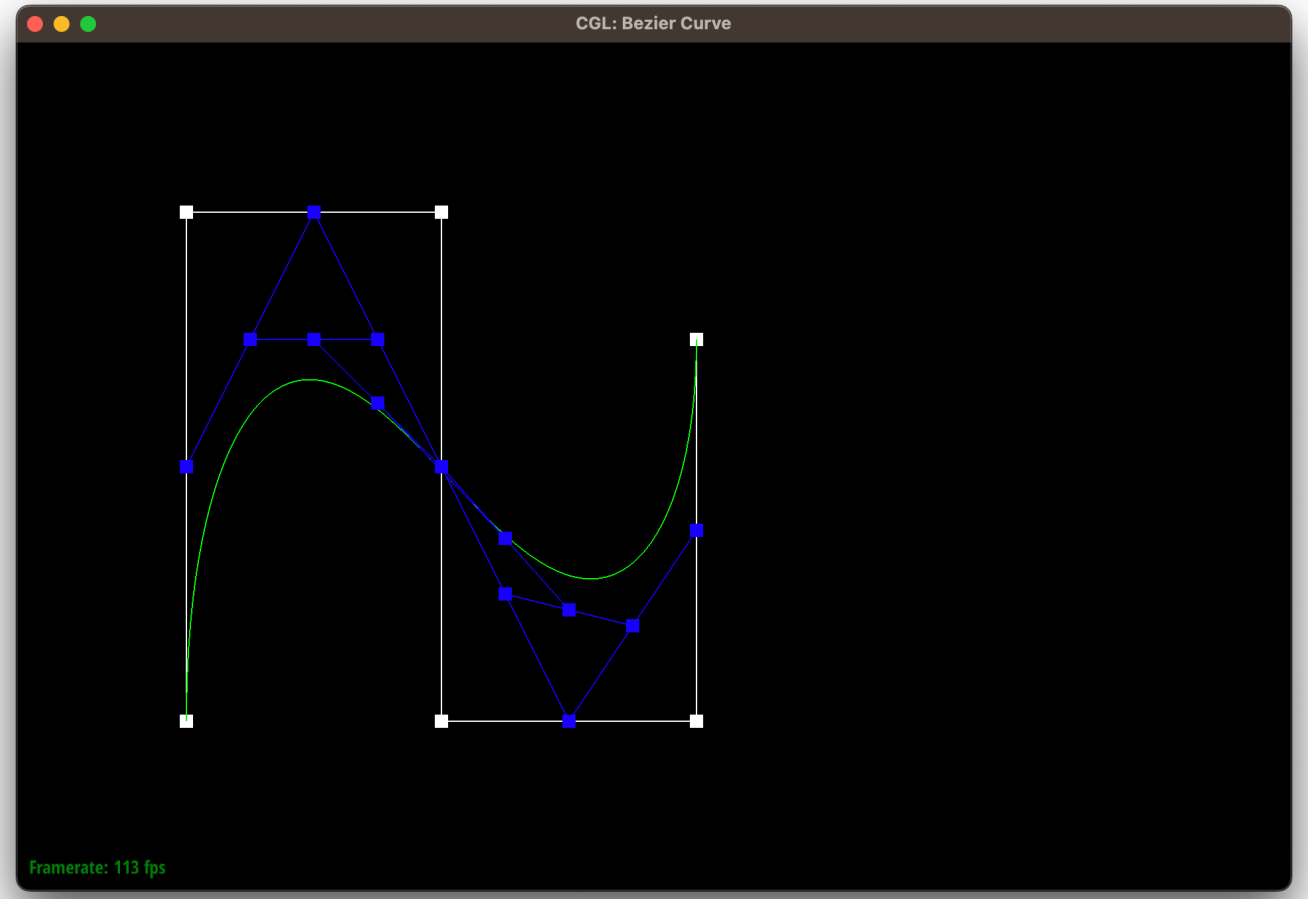
6 steps of Bezier Curve Evaluation with 6 Control Points

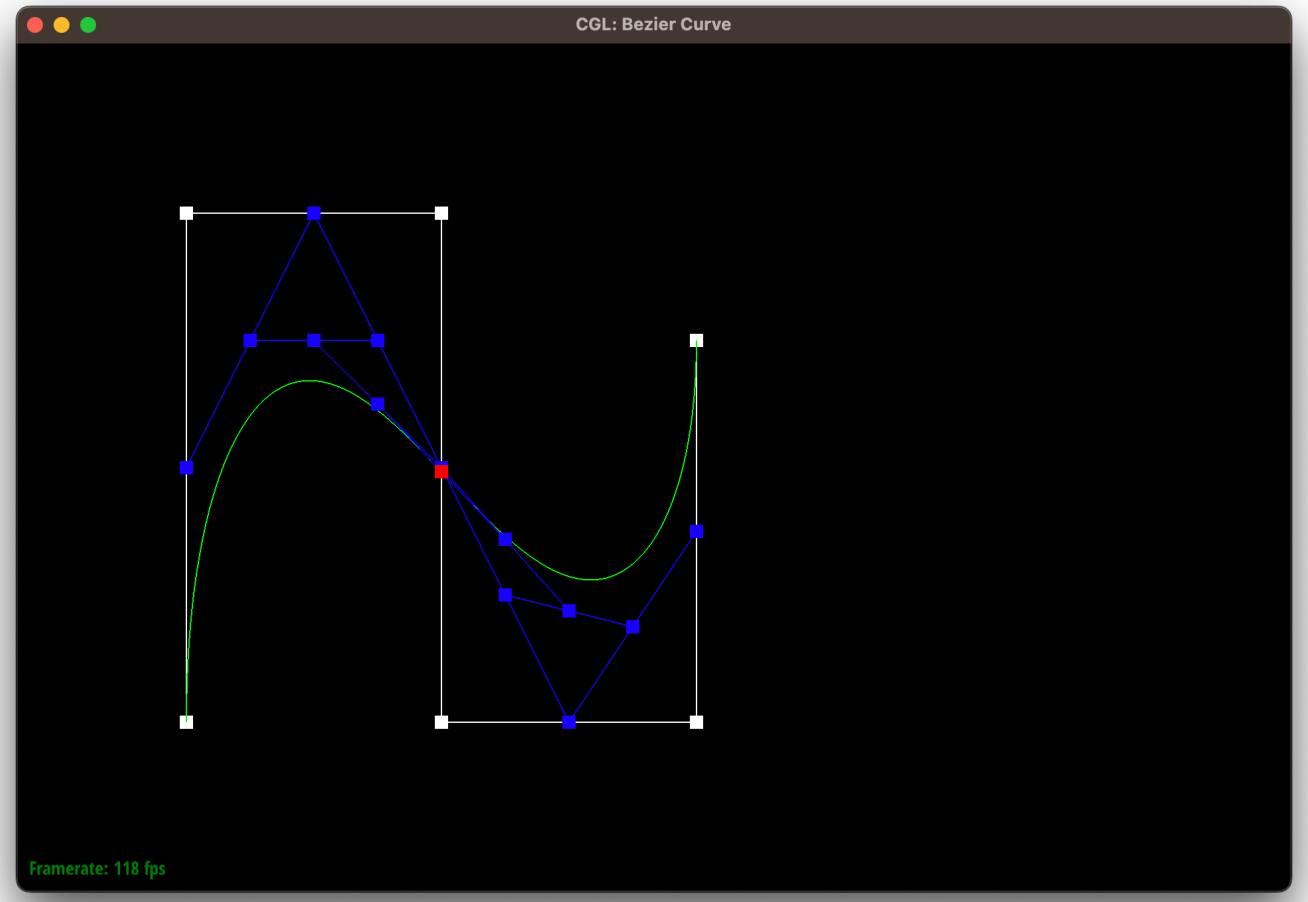

CGL: Bezier Curve
Framerate: 118 fps


CGL: Bezier Curve
Framerate: 118 fps

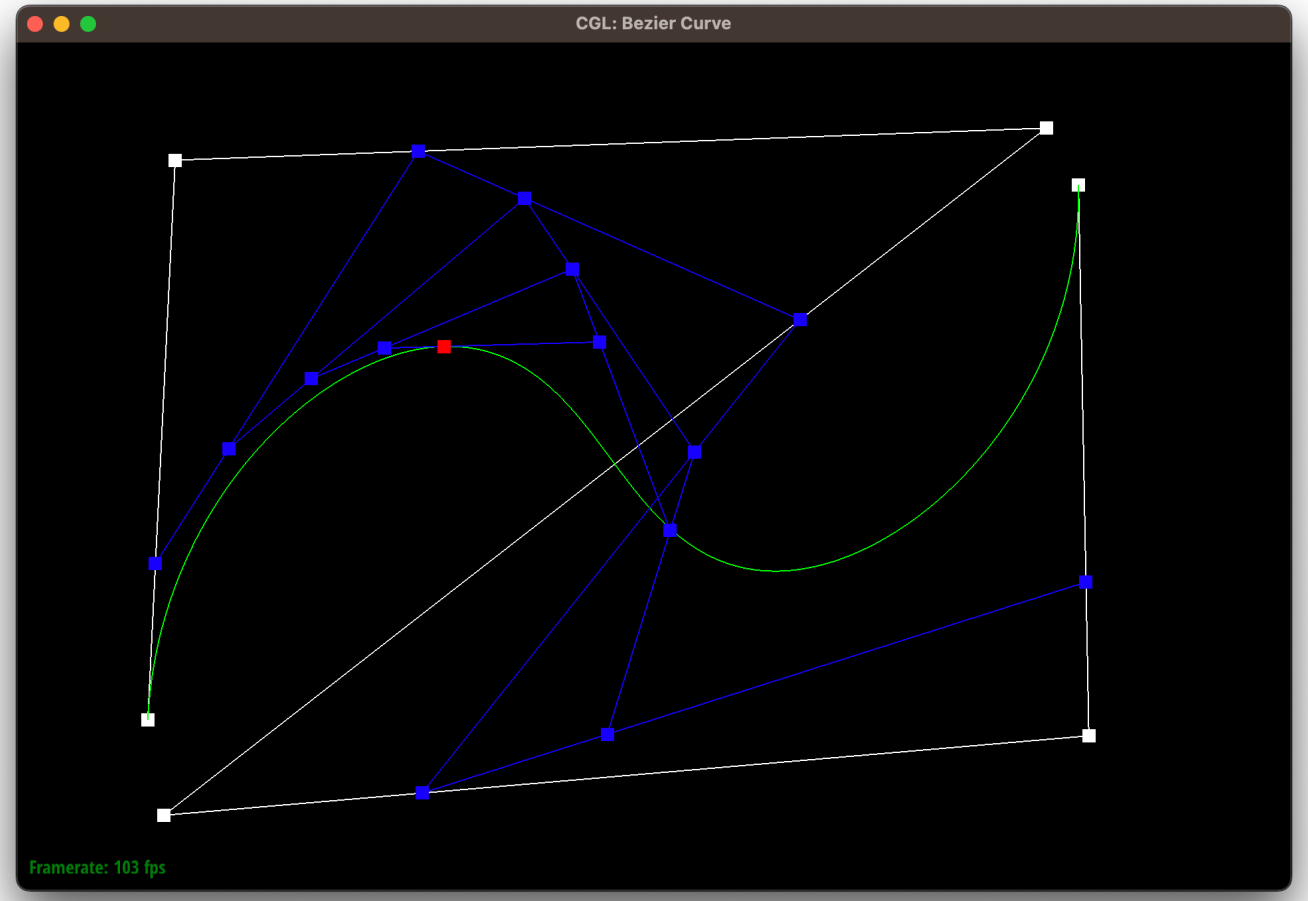

CGL: Bezier Curve
Framerate: 119 fps


CGL: Bezier Curve
Framerate: 118 fps


CGL: Bezier Curve
Framerate: 113 fps



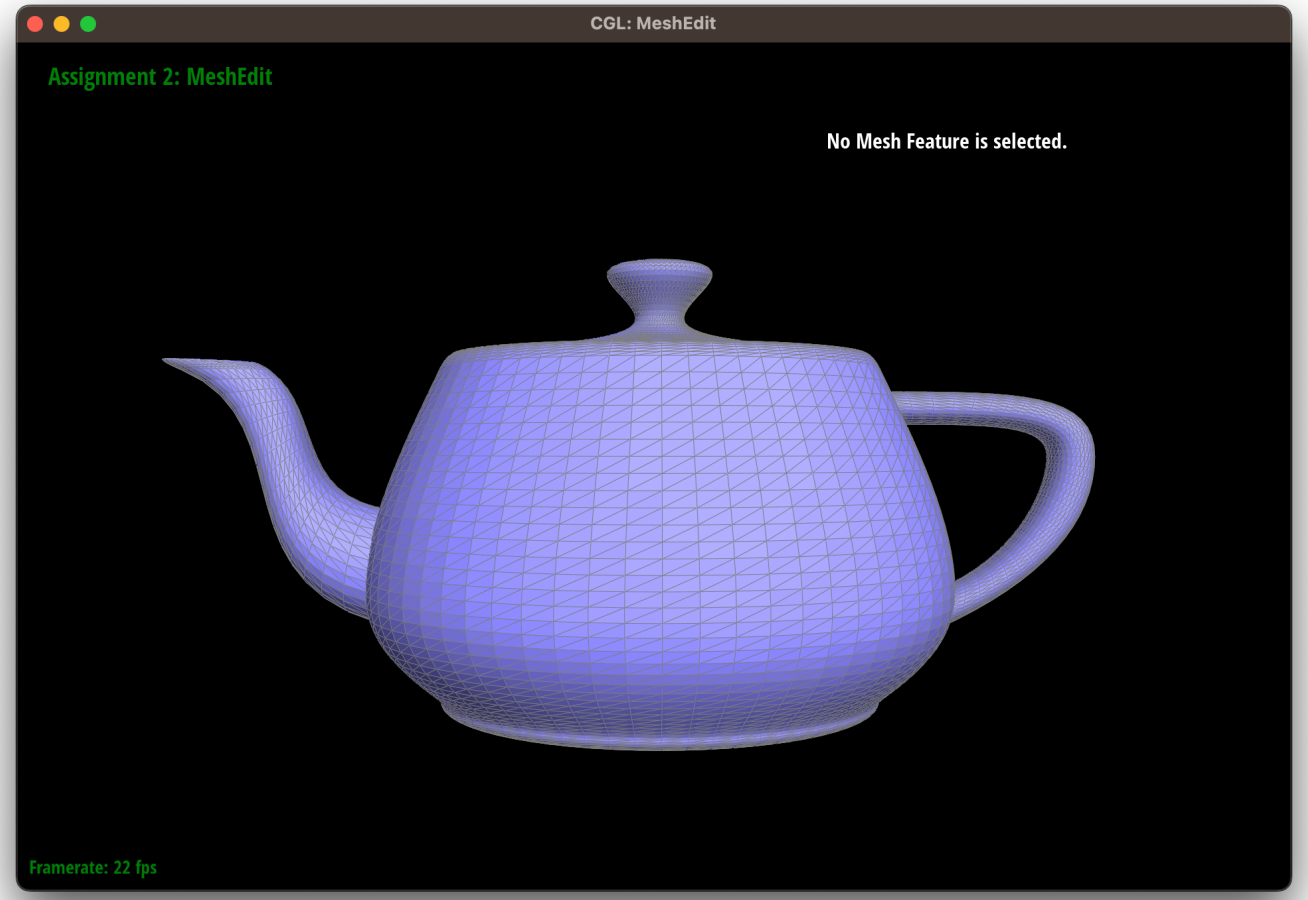
Bezier Curve Evaluation with Different Points and Different t Value



Part 2: Bezier Surfaces with Separable 1D de Casteljau

The de Casteljau algorithm extends to bezier surfaces by combining several bezier curves together in 3D space. To create a surface from some set of n bezier curves in space S, define a space U orthogonal to S. For some point v in U, evaluate each of the bezier curves at point v, resulting in a set of points p1, p2, ..., pn. Evaluate each of these points as a bezier curve in space U. By applying this technique to all points in space U, a 3D bezier surface is created. I implemented this algorithm by recursively evaluating each point v for each bezier curve in space S. I formed bezier curves from the resulting points, and after looping over all points in U, I formed a 3D bezier surface.

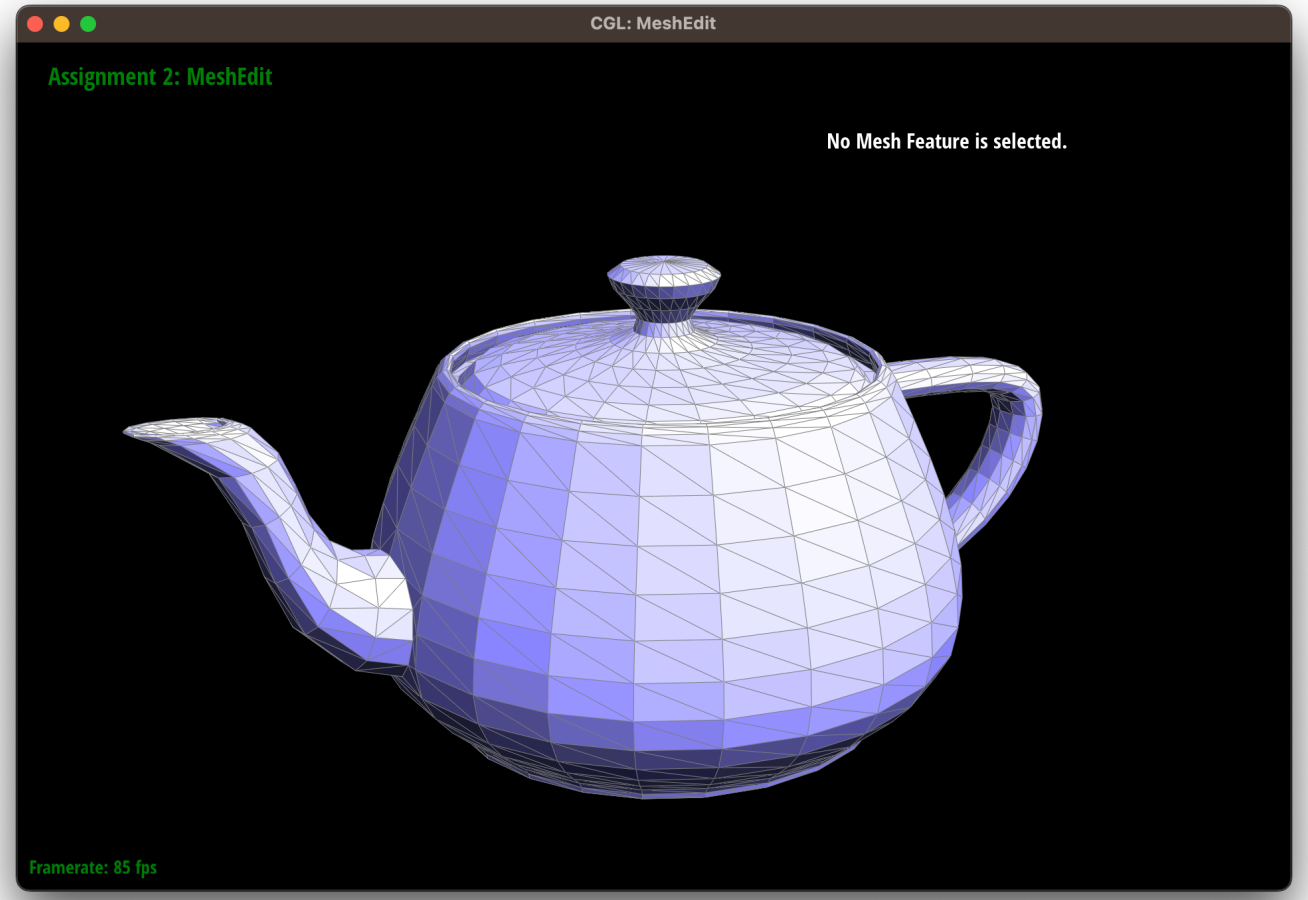
Teapot Rendered with Bezier Surfaces



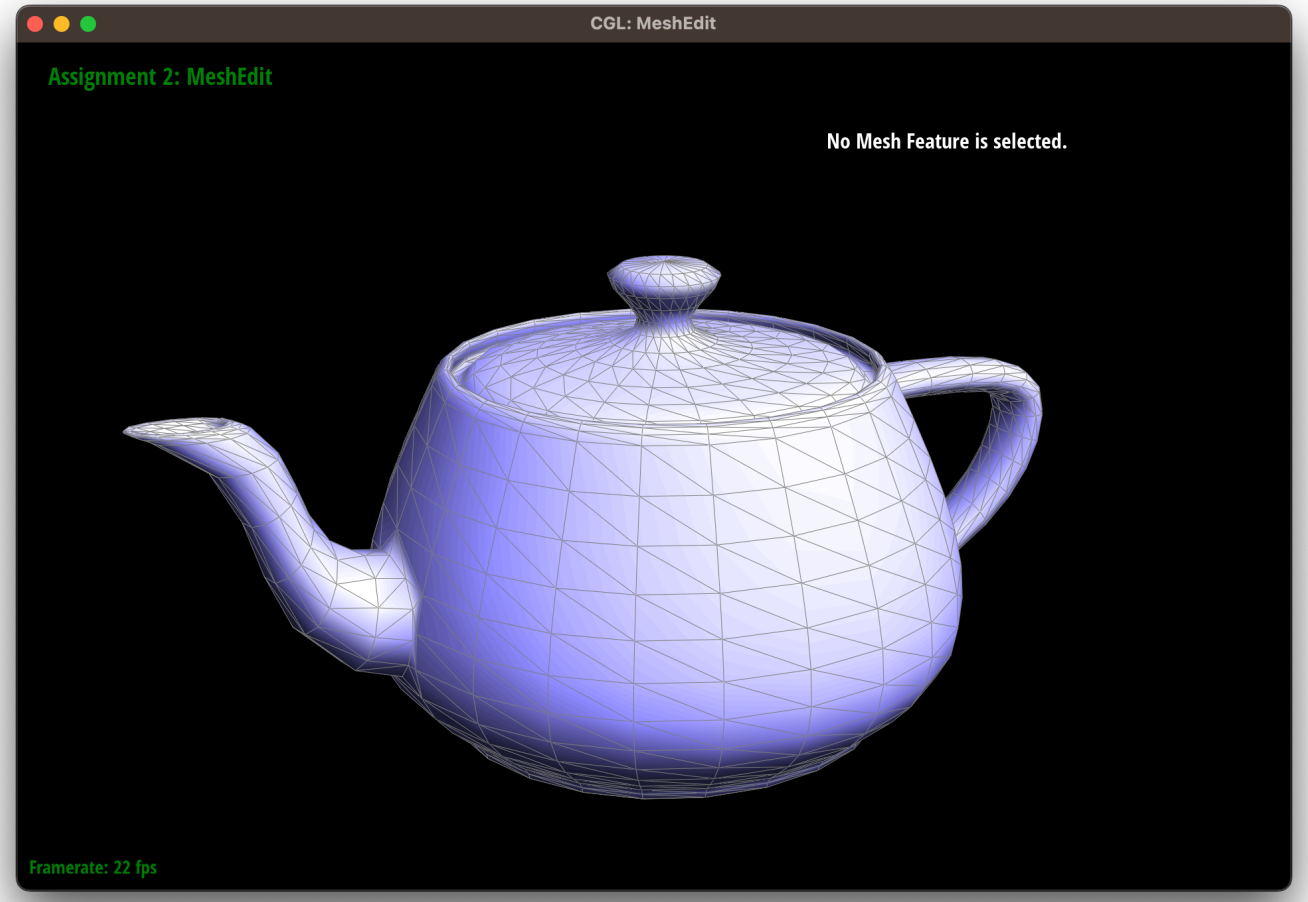
Part 3: Area-Weighted Vertex Normals

I implemented the area-weighted vertex normals by iterating through each of triangles incident to the vertex and added their normal vector scaled by their area to a running sum. After iterating through all triangles, I took the norm of that sum to yield the area-weighted vertex normal.

Teapot without Vertex Normals



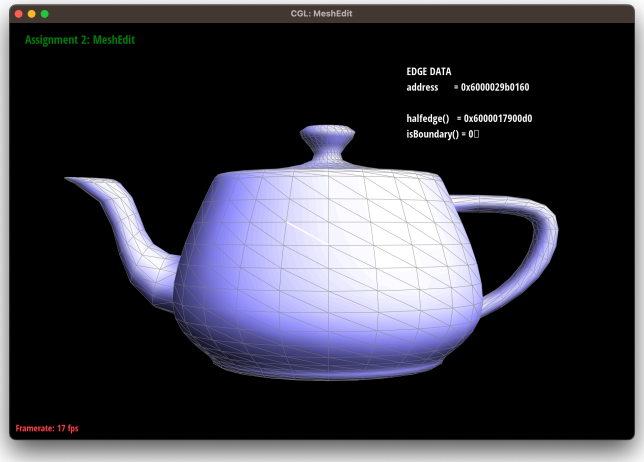
Teapot with Vertex Normals

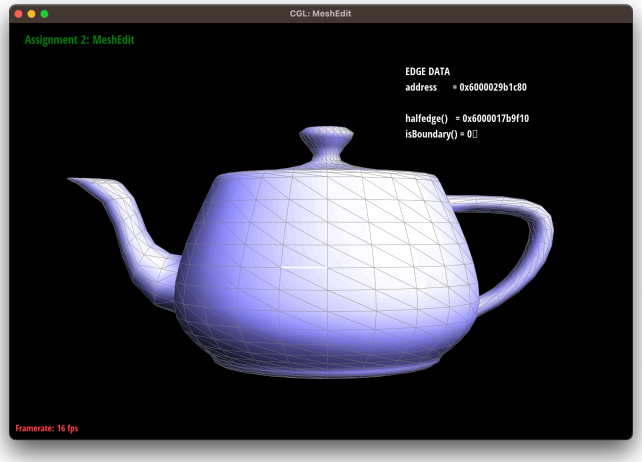


Part 4: Edge Flip

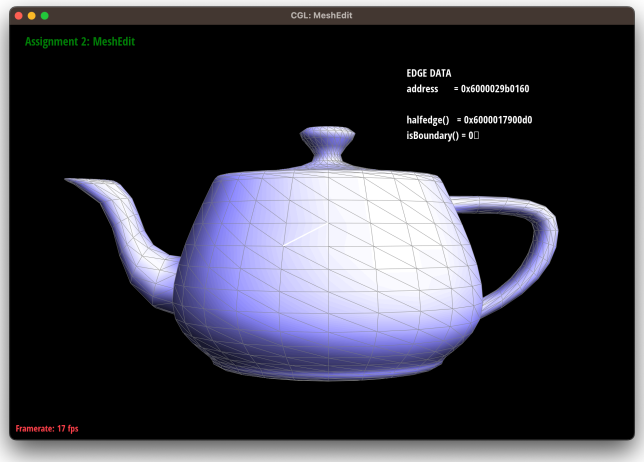
To implement the flip operation, I first drew a diagram of all the vertices, edges, faces, and halfedges of the triangles I start with and the triangles I hoped to end with. Next, I defined each of the vertices, edges, faces, and halfedges as new variables. Then, I redefined the next, twin, vertex, edge, and face components of each of the halfedges. After, I redefined the halfedge components of the verteices, edges, and faces. Lastly, I returned the original edge iterator that was passed in. I found following the guide linked on the debugging page very helpful in the process for this part, and after following that guide I had no debugging problems.

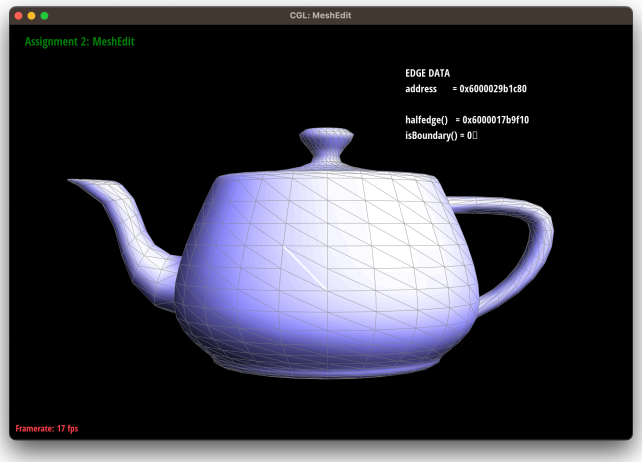
Teapot Before Edge Flips





Teapot After Edge Flips

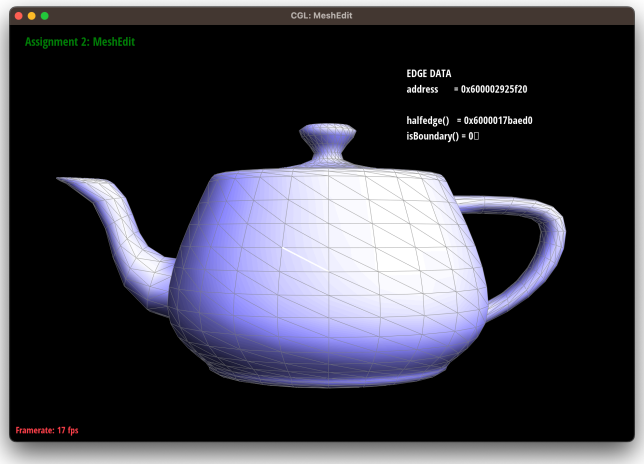


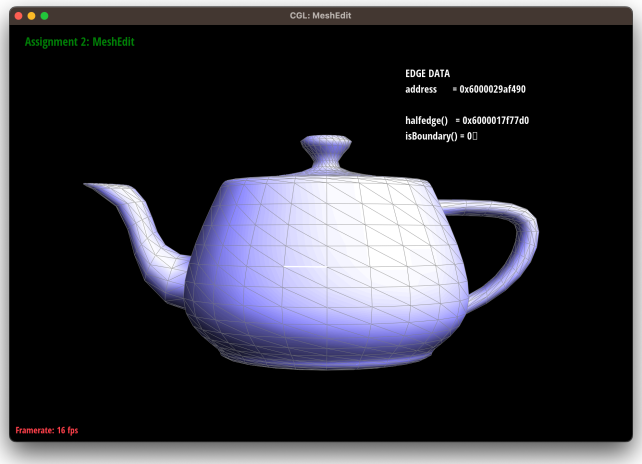


Part 5: Edge Split

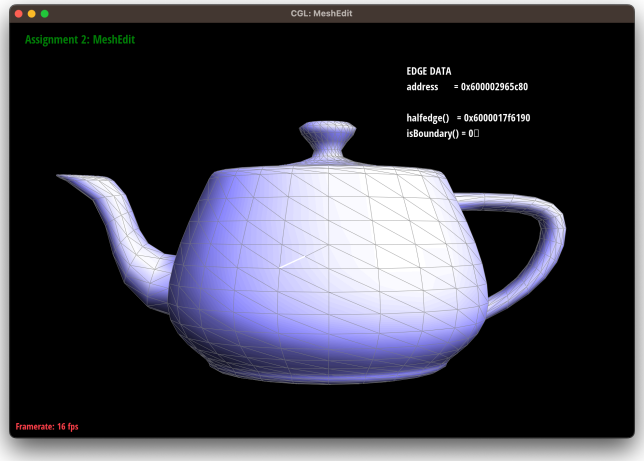
My implementation process for edge splits was similar to edge flips. I drew a diagram including all vertices, edge, faces, and halfedges, and I translated that diagram into the new diagram I drew. The difference this time, though, was that I needed to define one new vertex, two new edges, two new faces, and six new halfedges. I defined these after defining all other components of the first diagram, and I redefined their components similarly. I defined the isNew component of the old edge that I split as false and the isNew component of the new edges as true. Again, following the guide was very helpful in making sure my code did not have any bugs.

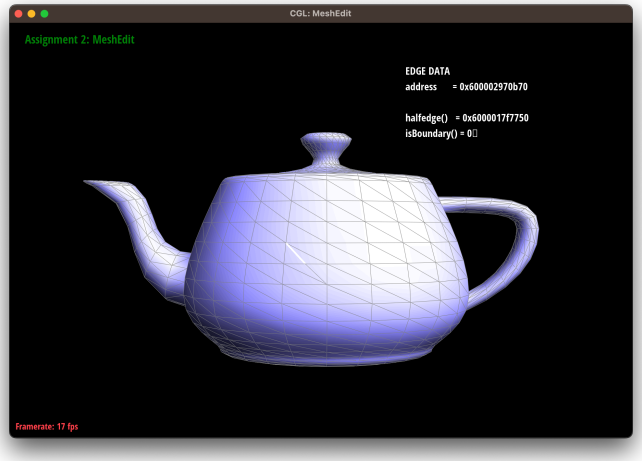
Teapot Before Edge Splits



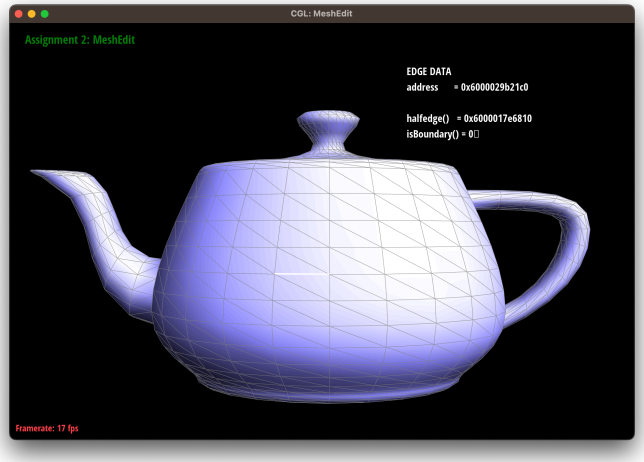


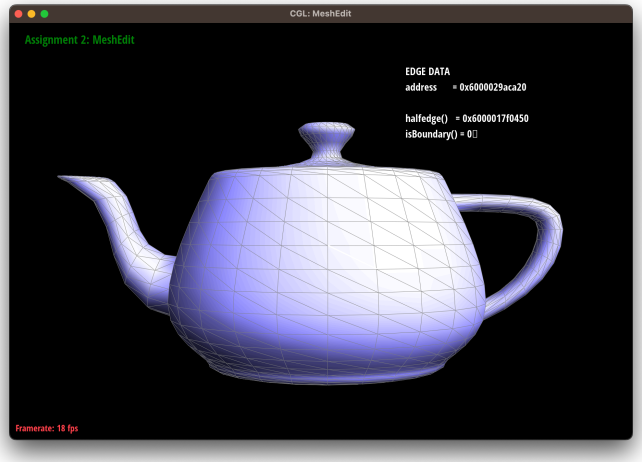
Teapot After Edge Splits



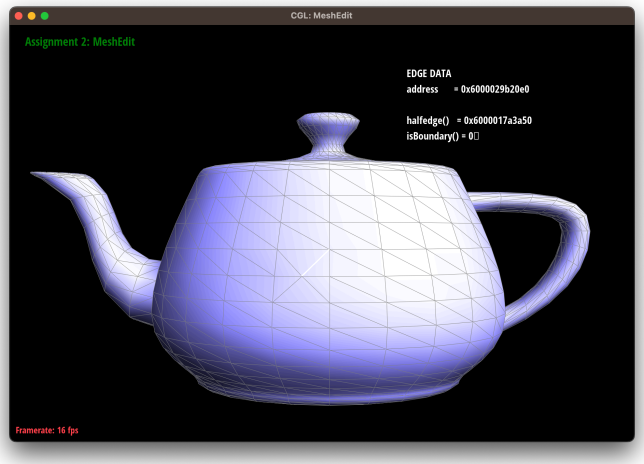


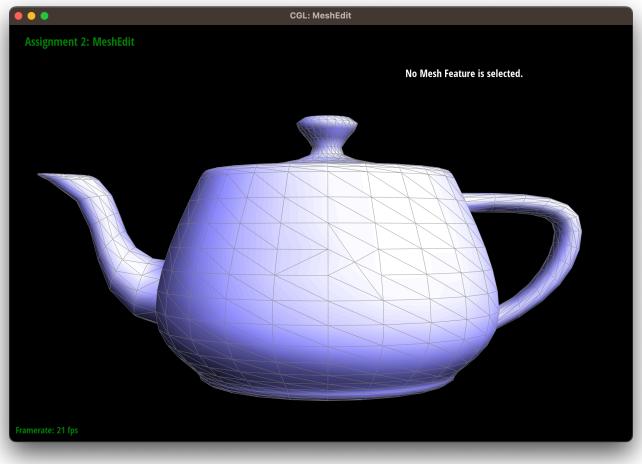
Teapot Before Edge Split and Flip





Teapot After Edge Split and Flip



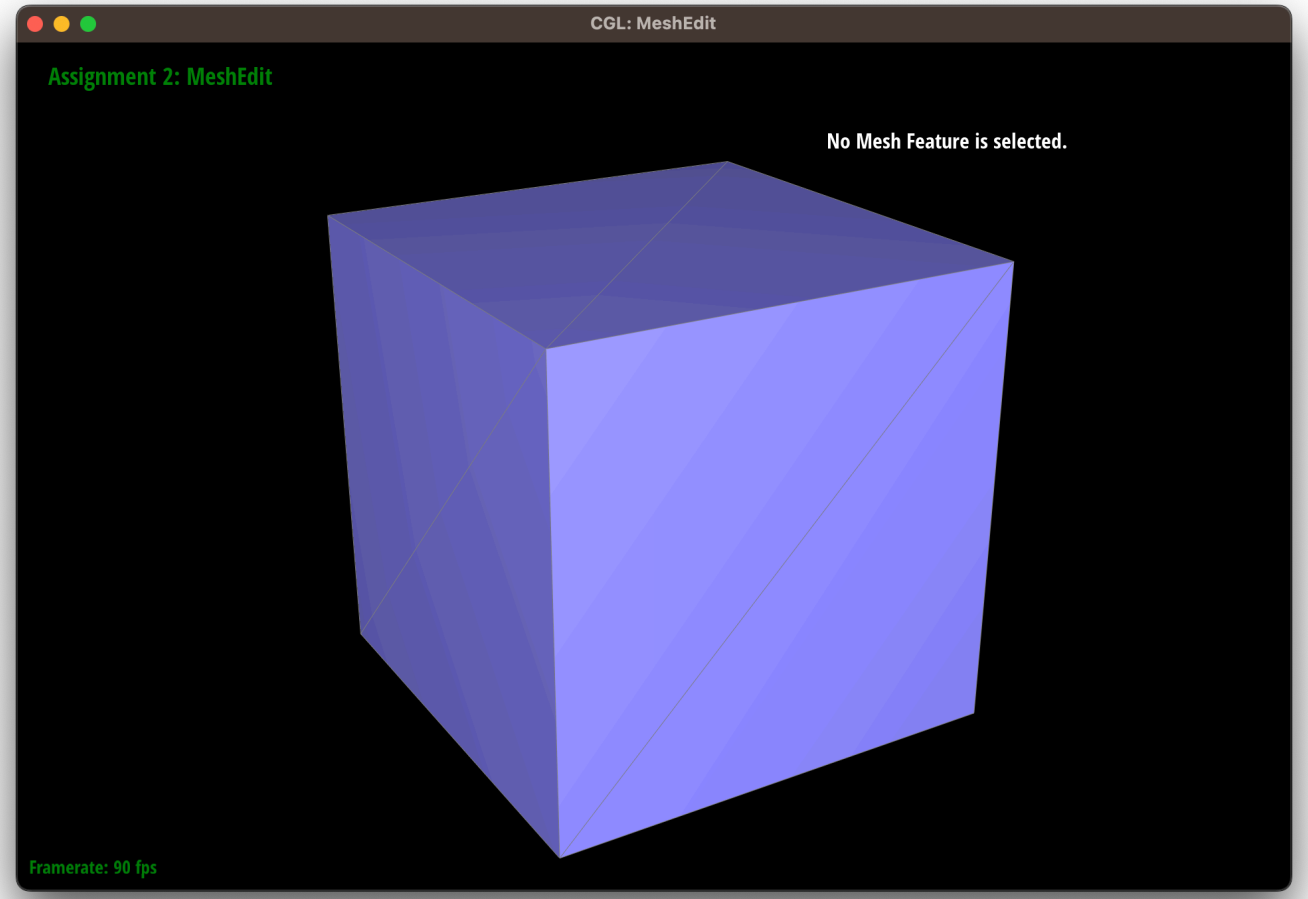


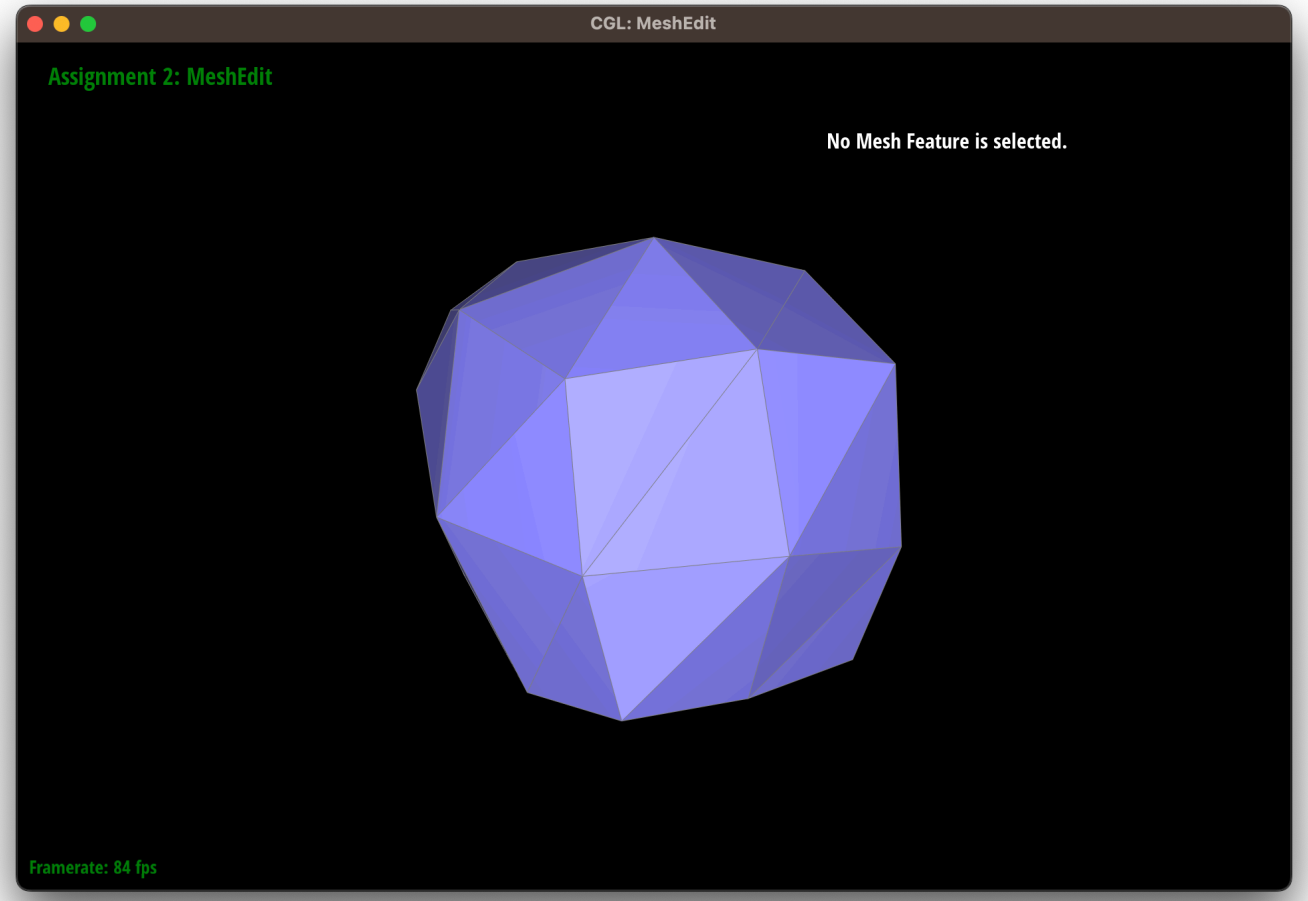
Part 6: Loop Subdivision for Mesh Upsampling

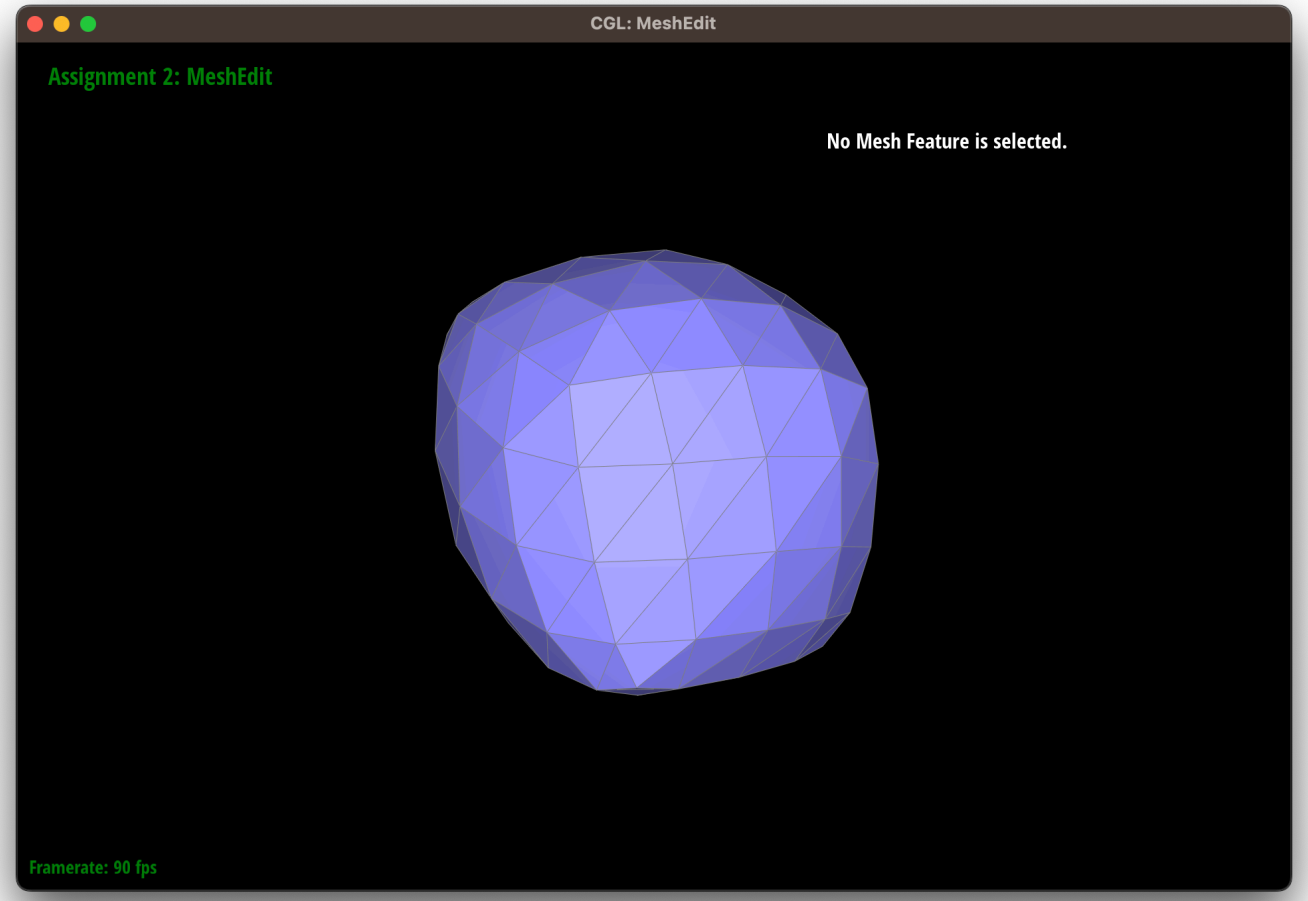
I implemented loop subdivision by splitting each triangle into four triangles. To do this, I needed to calculate the positions of the old and new vertices using the loop subdivision rule before splitting, splitting all edges, flipping new edges that connected new and old vertices, and updating the positions of the new and old vertices. One bug I ran into was implementing the loop subdivision rule wrong, as I did not compute the sum of all neighboring vertices (also known as the centroid computation) correctly. I resolved this by using the computeCentroid function.

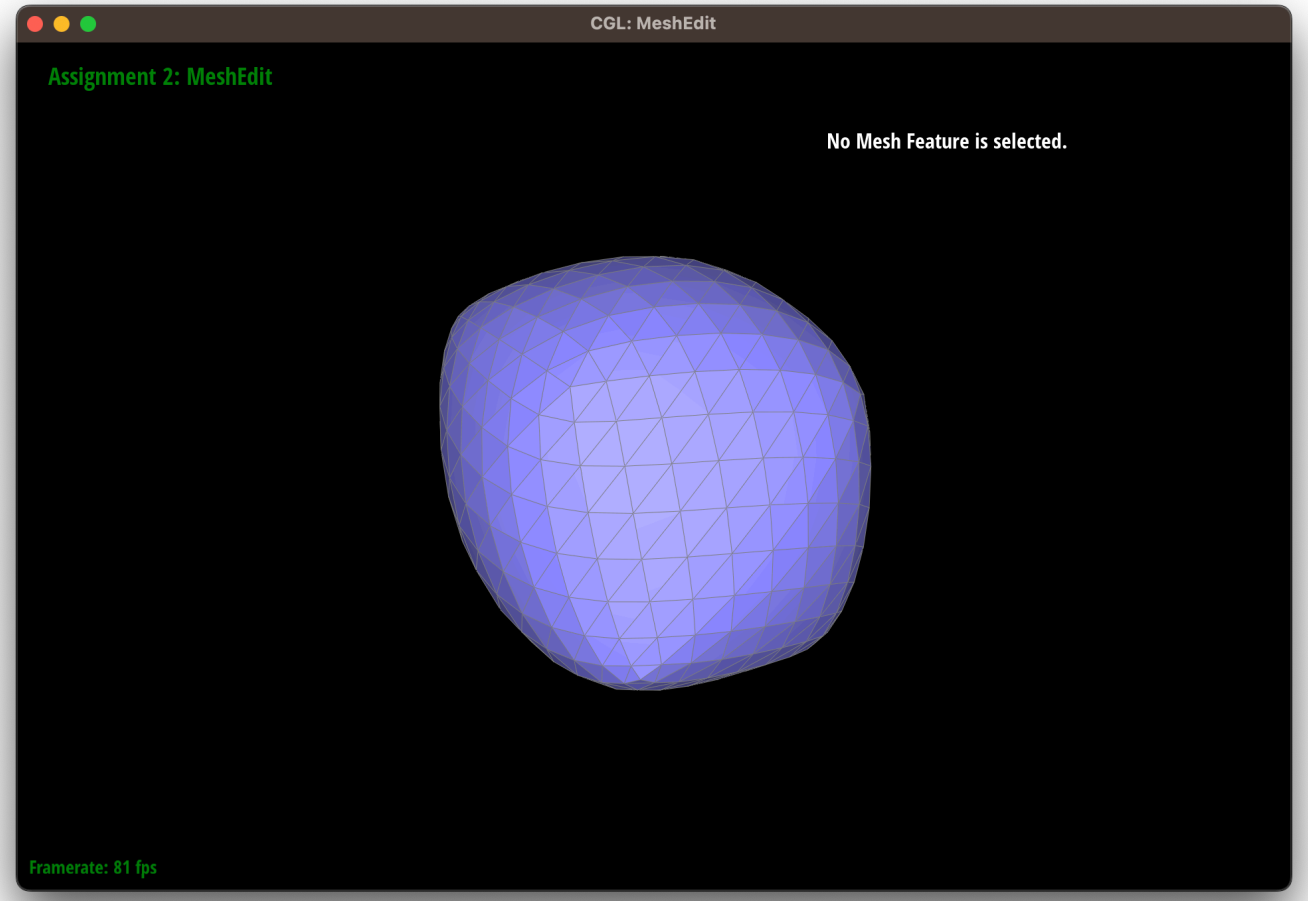
After many loop subdivisions, I noticed that the sharp corners and edges of the cube get smoothed out and the cube becomes asymmetrical as seen below.

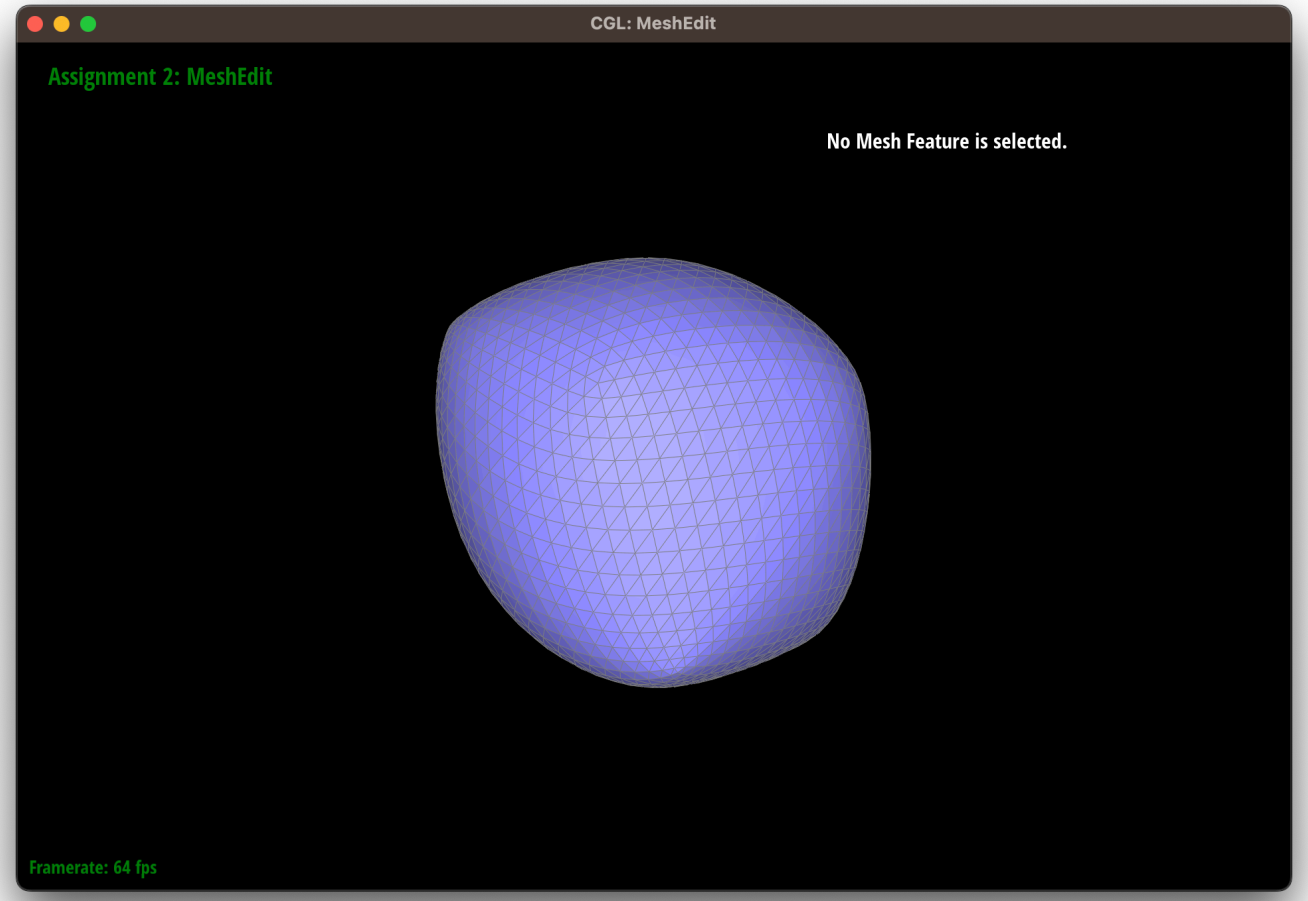
Steps of Loop Subdivision


CGL: MeshEdit
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 90 fps


CGL: MeshEdit
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 84 fps

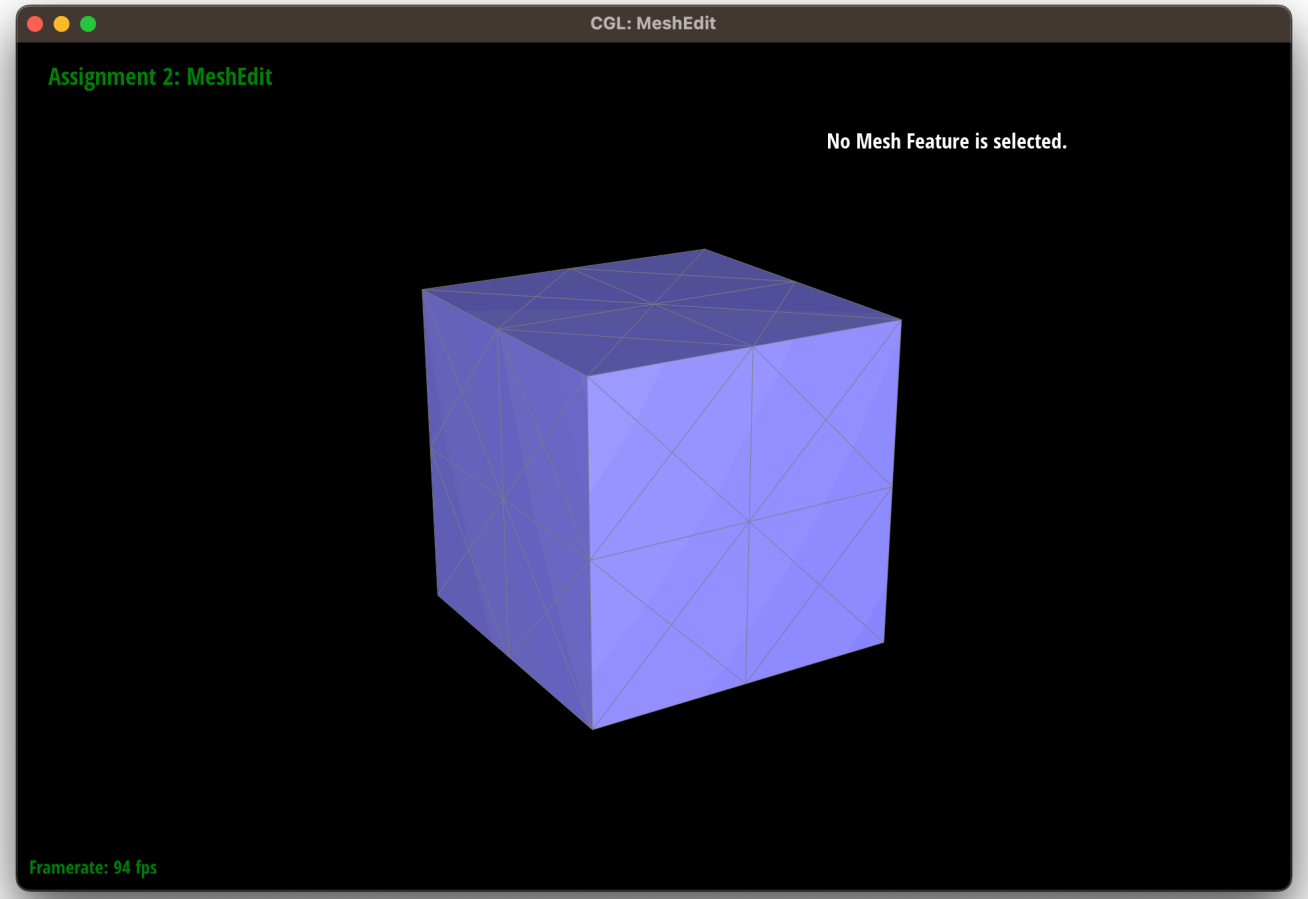

CGL: MeshEdit
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 90 fps

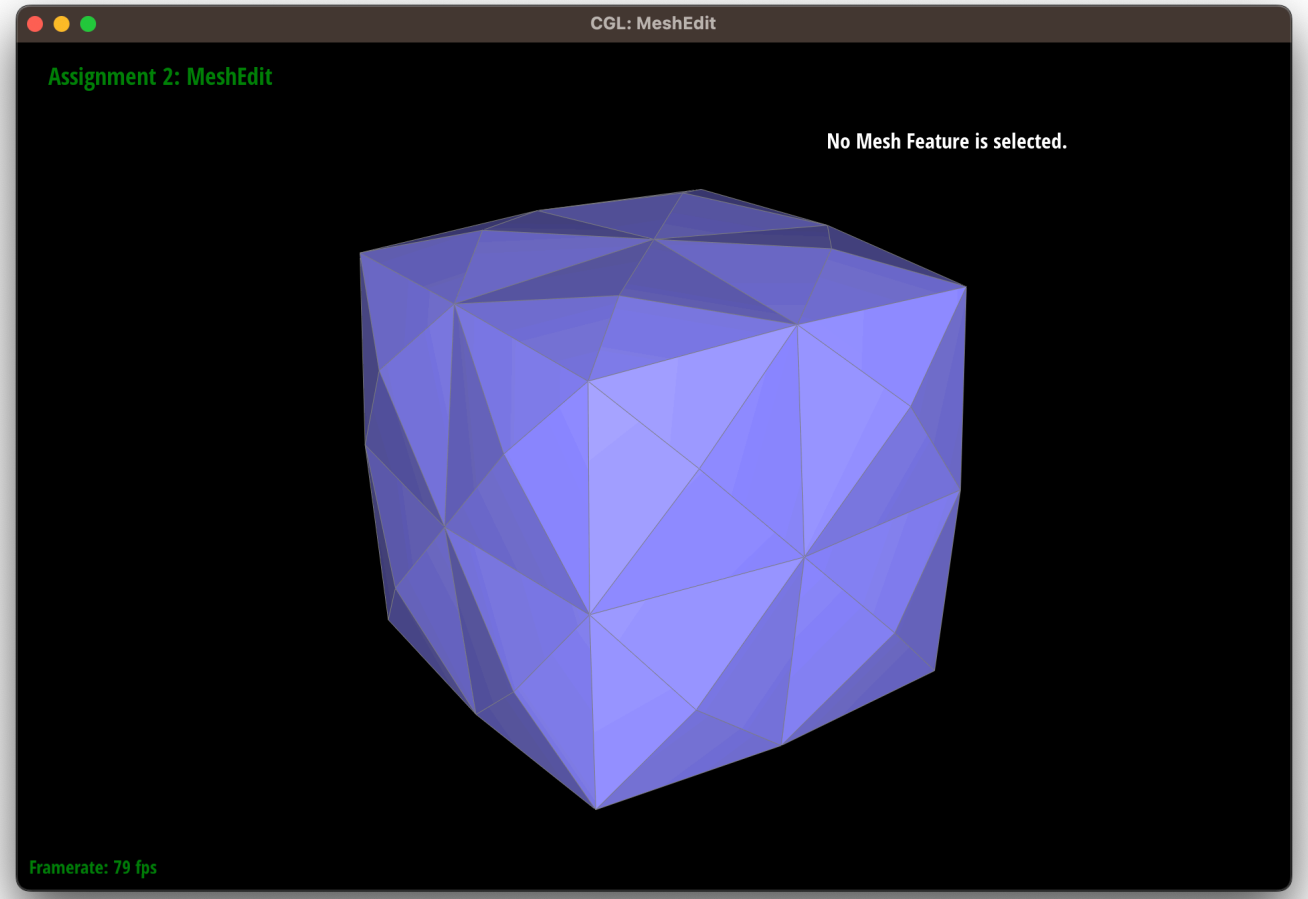

CGL: MeshEdit
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 81 fps

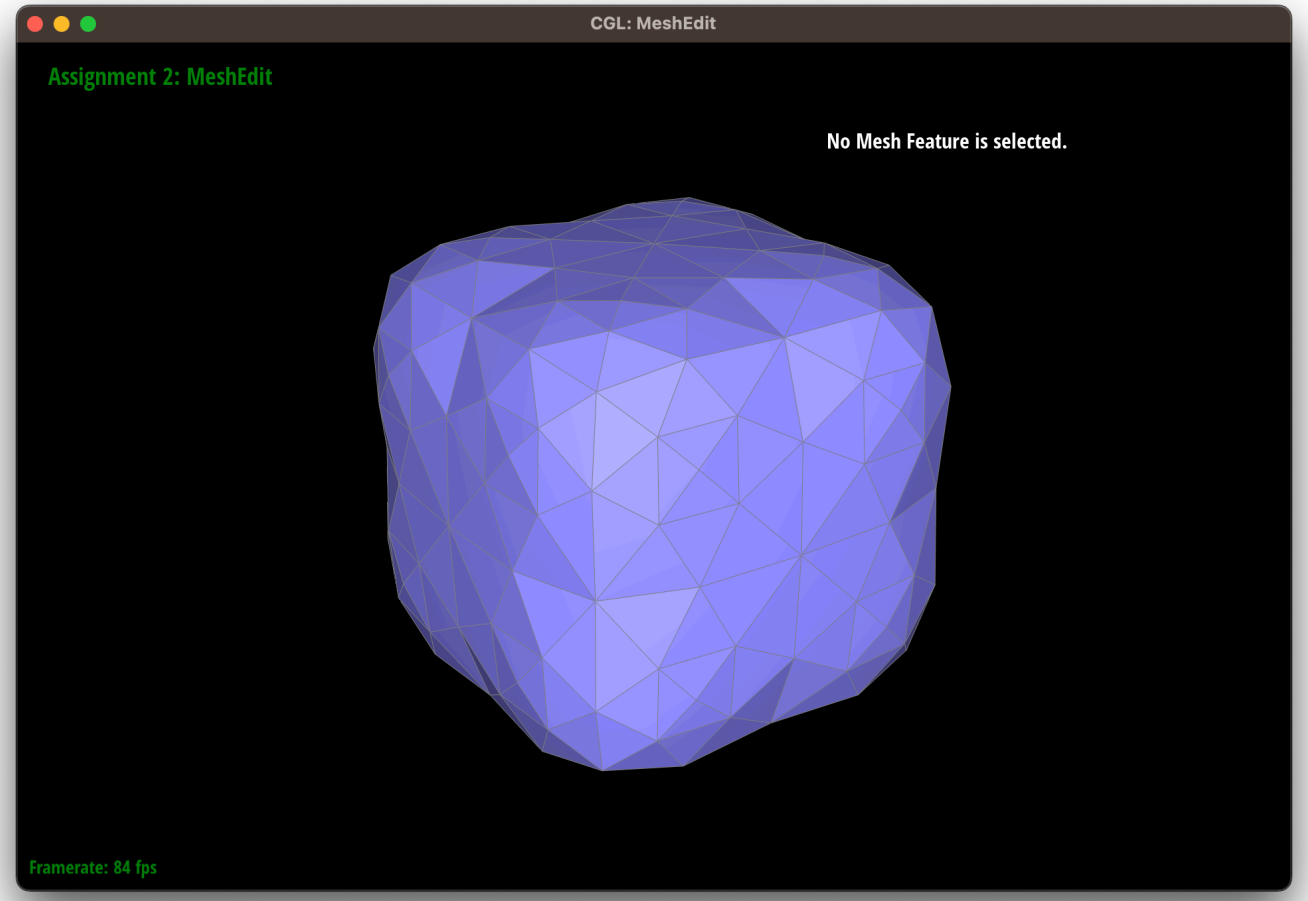


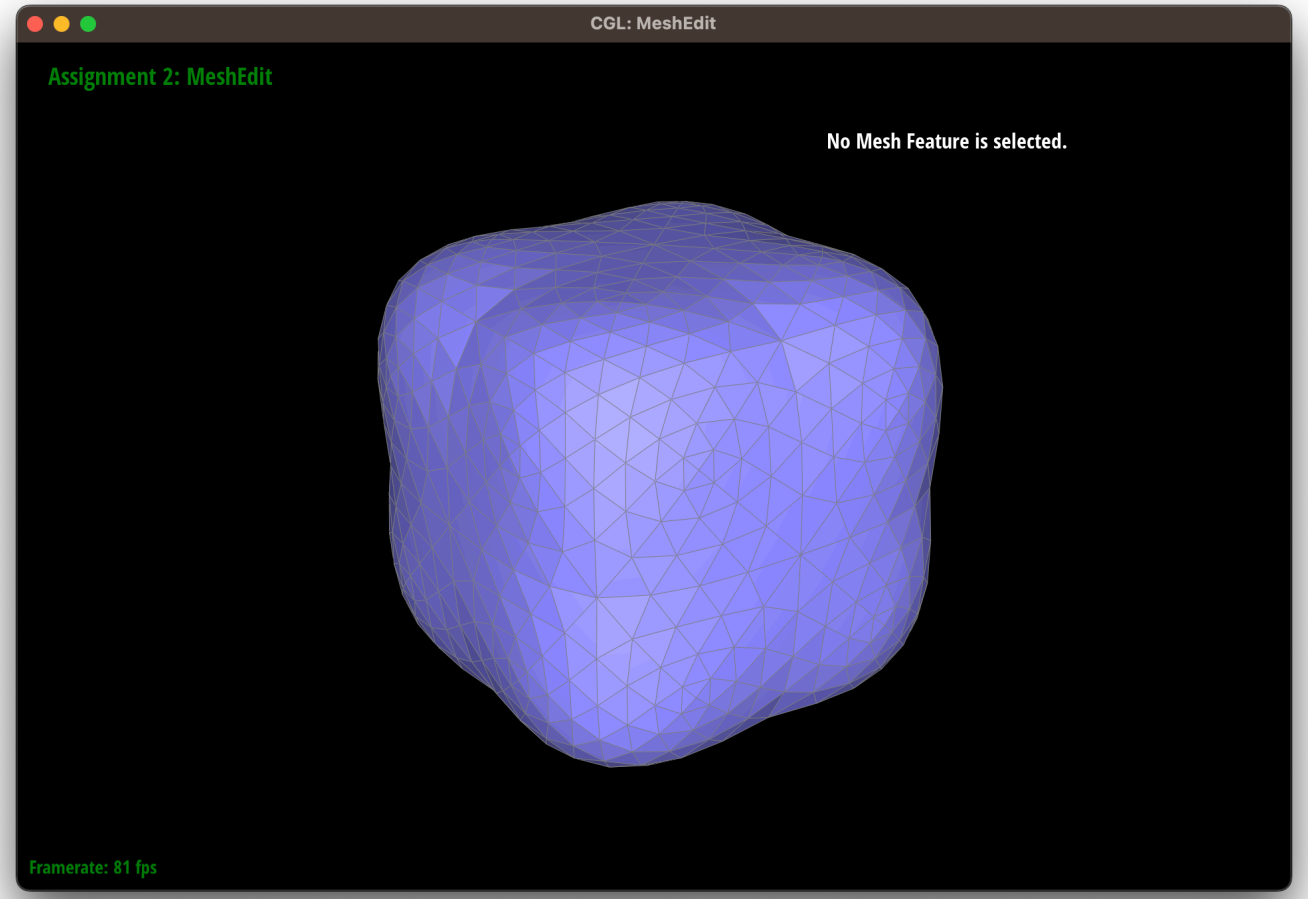
In order to mitigate these effects, we can preprocess some of the splitting and flipping of the cube so that the cube retains more of its symmetry and sharpness. Due to the weighting of the loop subdivision rule, the corners of the cube become normalized between the points that lie on the faces of the cube. Since the midpoint between these points is further inside the cube than the corner is, the corner point gets closer to the center of the cube and in turn gets rounded. The cube starts with one diagonal edge between two of its vertices, and since that edge is labeled as an old edge, it retains more of its sharp corner shape than the other vertices of the face due to how the loop subdivision rule handles vertex positions. Since those corners are sharper than the non-connected corners, the cube becomes asymmetrical. By splitting this edge at the beginning so every corner is connected by an edge before running our loop subdivision algorithm, our cube will stay symmetric, and by dividing the cube into smaller triangles before running our algorithm, the corners will retain more of their sharp shape and reduce rounding. Below is the steps of the loop subdivision with preprocessed edges.

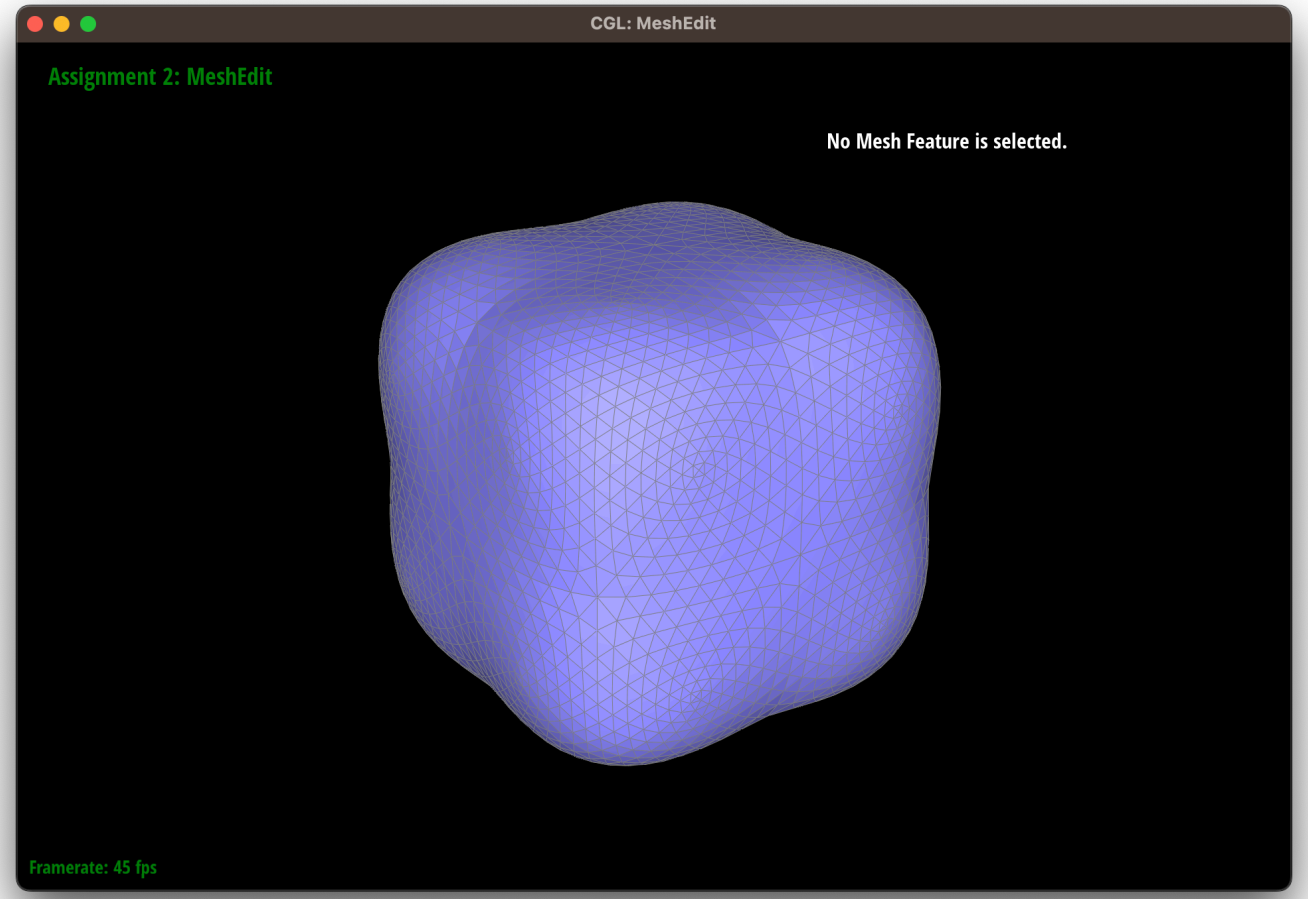
Steps of Loop Subdivision with Preprocessed Edges


CGL: MeshEdit
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 94 fps


CGL: MeshEdit
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 79 fps


CGL: MeshEdit
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 84 fps


CGL: MeshEdit
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 81 fps


CGL: MeshEdit
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 45 fps