Project 2 Write-Up

Yao Fu Zuo and Bohan Yu

Github Pages Link: https://cal-cs184-student.github.io/sp22-project-webpages-yfz3357/

Section 1 Part 1:

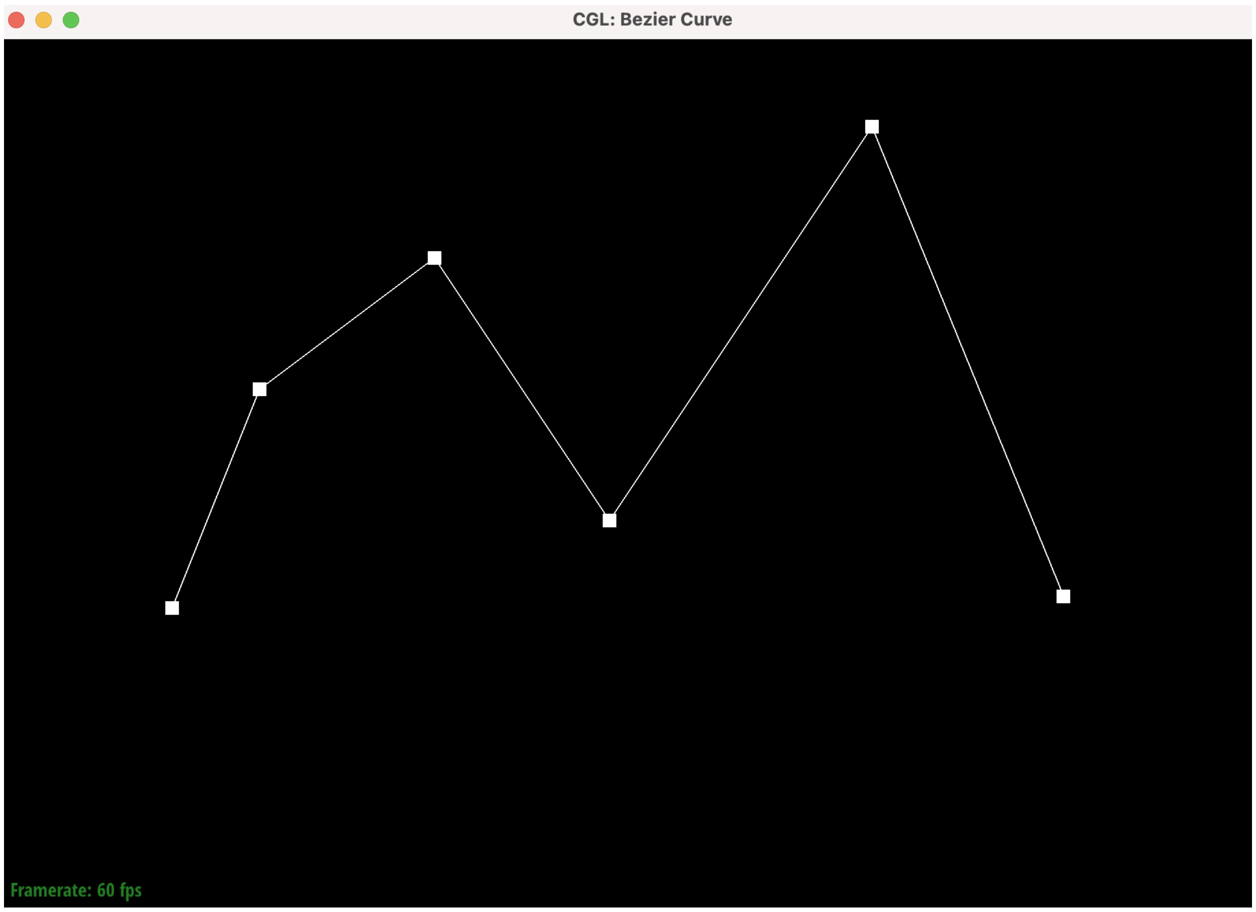
De Casteljau's algorithm recursively computes new control points (i.e. intermediate control points) using linear interpolation on each two consecutive control points, and will return a final point at the end. By adjusting the parameter “t” in the range of [0, 1], the final computed point will swipe a curve known as the Beizer curve.

Let’s say we have v_0, v_1, ..., v_n as our initial control points. Then at the first iteration of De Casteljau's, we will have n - 1 control points at the next level, and we do the same thing to these new intermediate control points until there’s only 1 control point at the end.

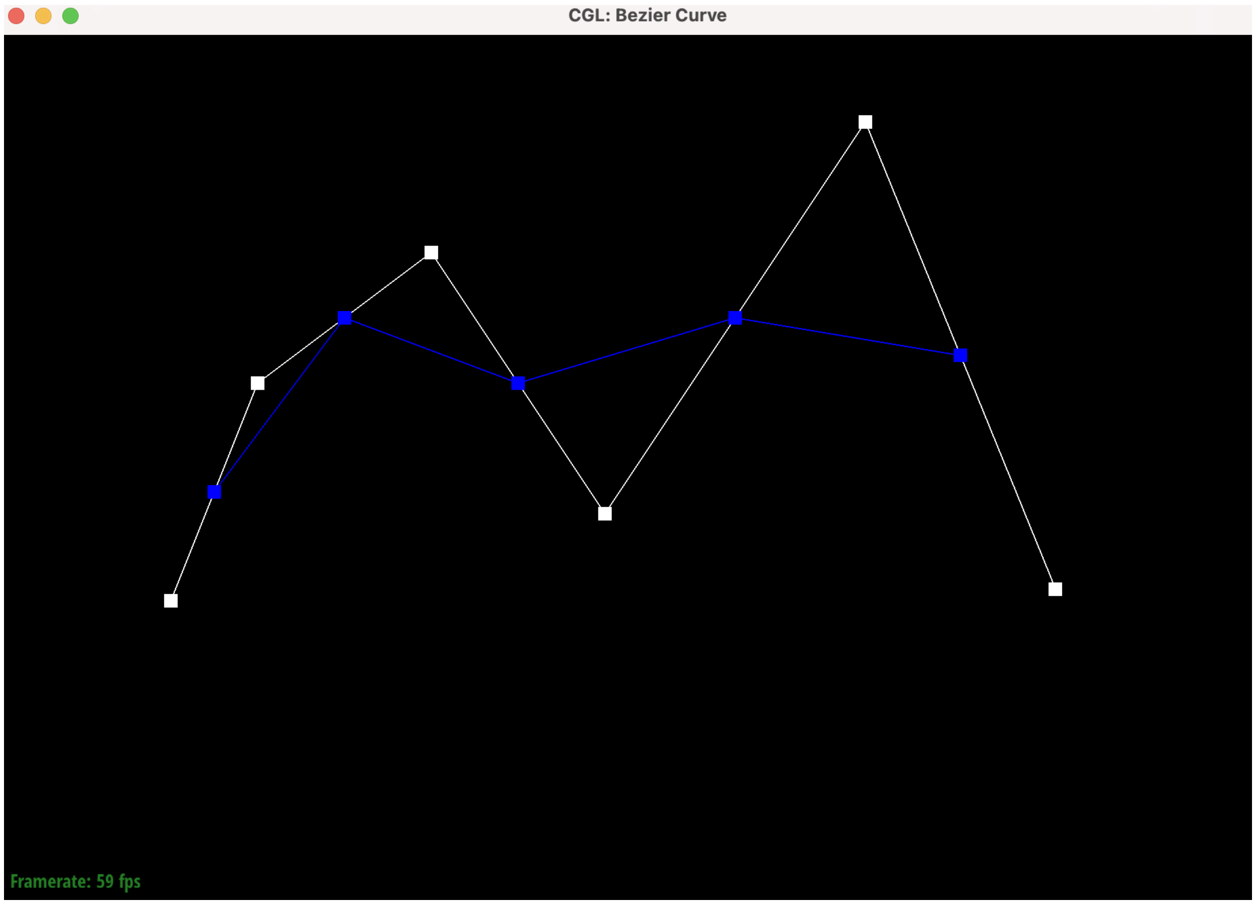
Our implementation computes each iteration of De Casteljau's by looping over all current control points and lerp on each two consecutive points. That is, we calculate $Lerp(v_0, v_1)$, $Lerp(v_1, v_2)$... $Lerp(v_n, v_{n-1})$, giving $n - 1$ new control points. We iterate until there is only one new control point. By adjusting the parameter t, the position of this final computed point will change accordingly, swiping a trajectory that is the final Beizer curve for the original control points.

The screenshot for each iteration is shown below (on the next page):

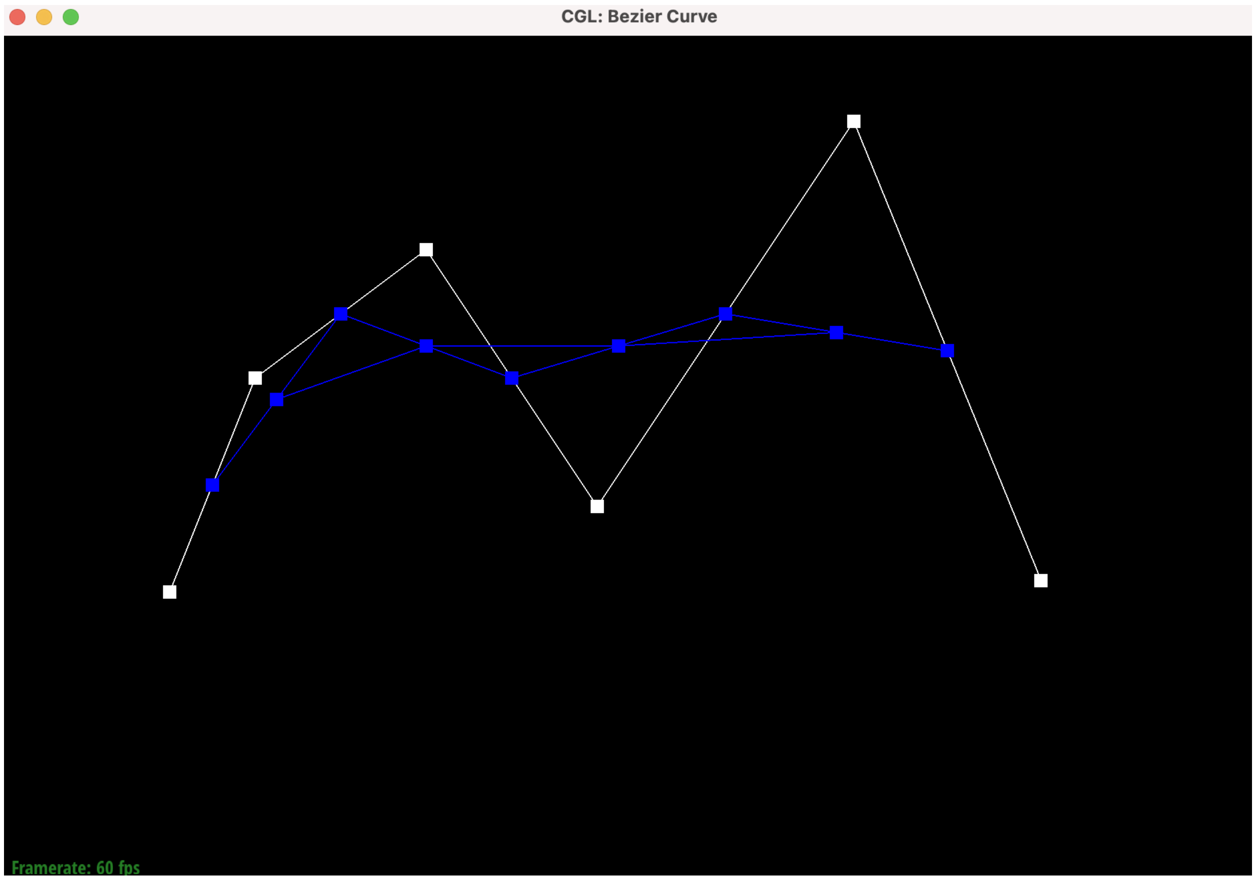
Before iteration:



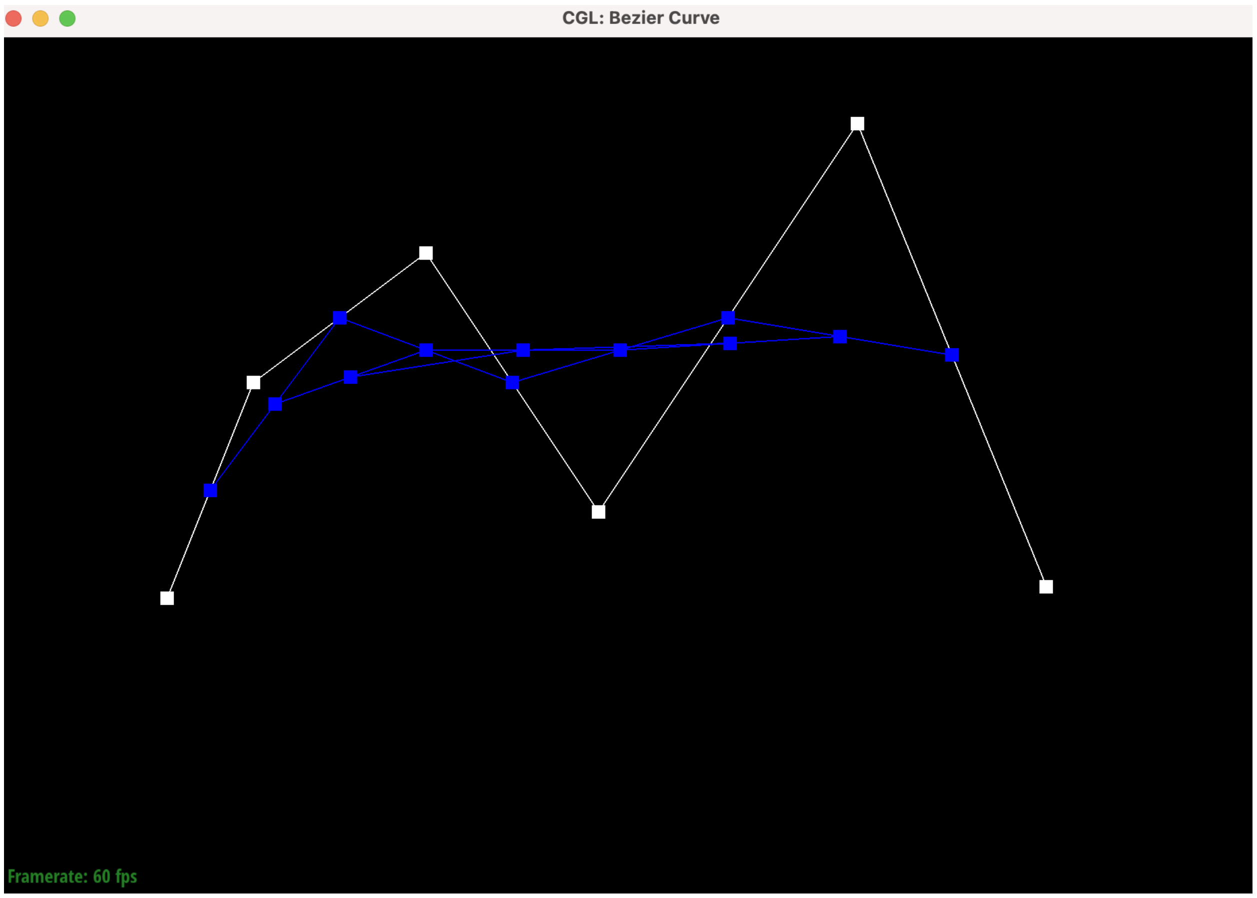
After 1 iteration:



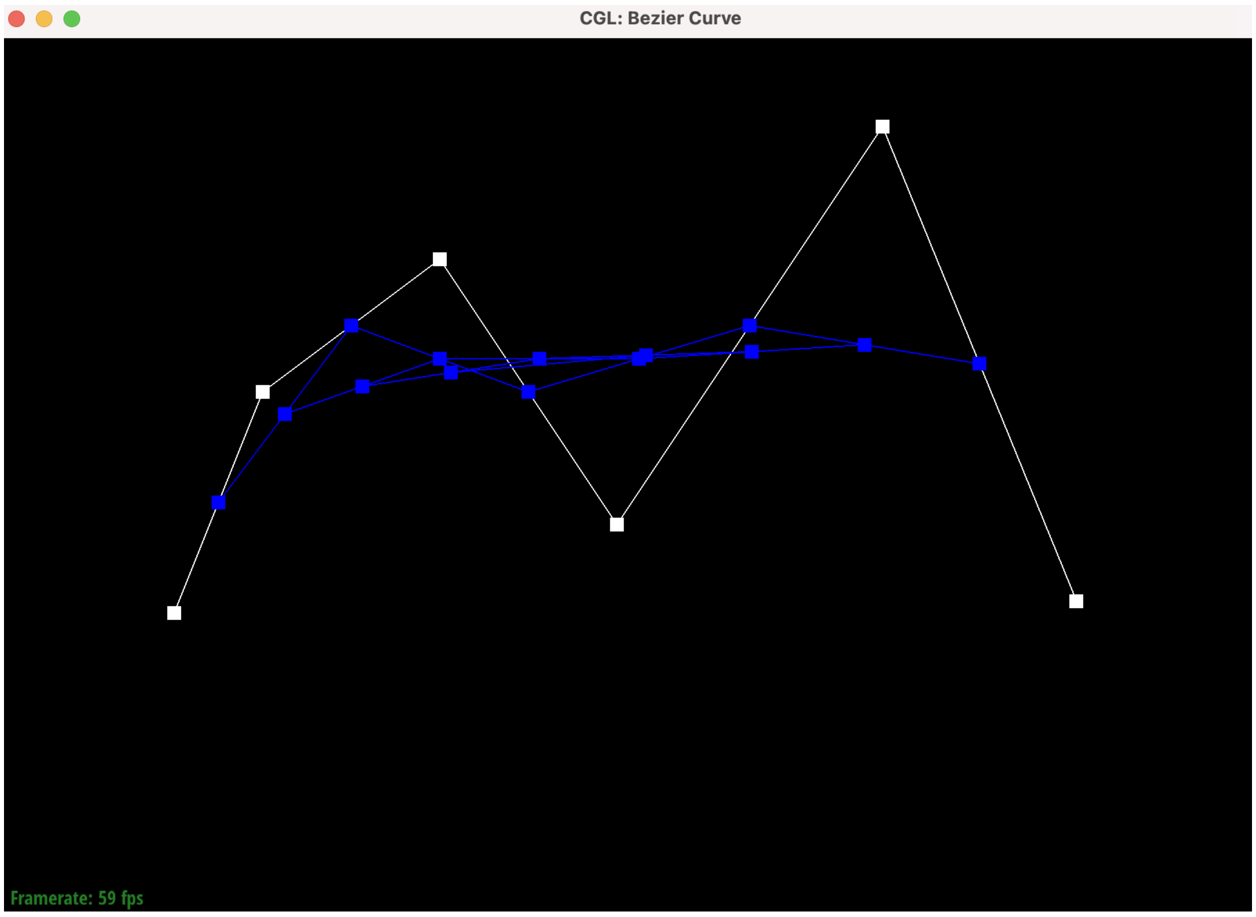
After 2 iterations:



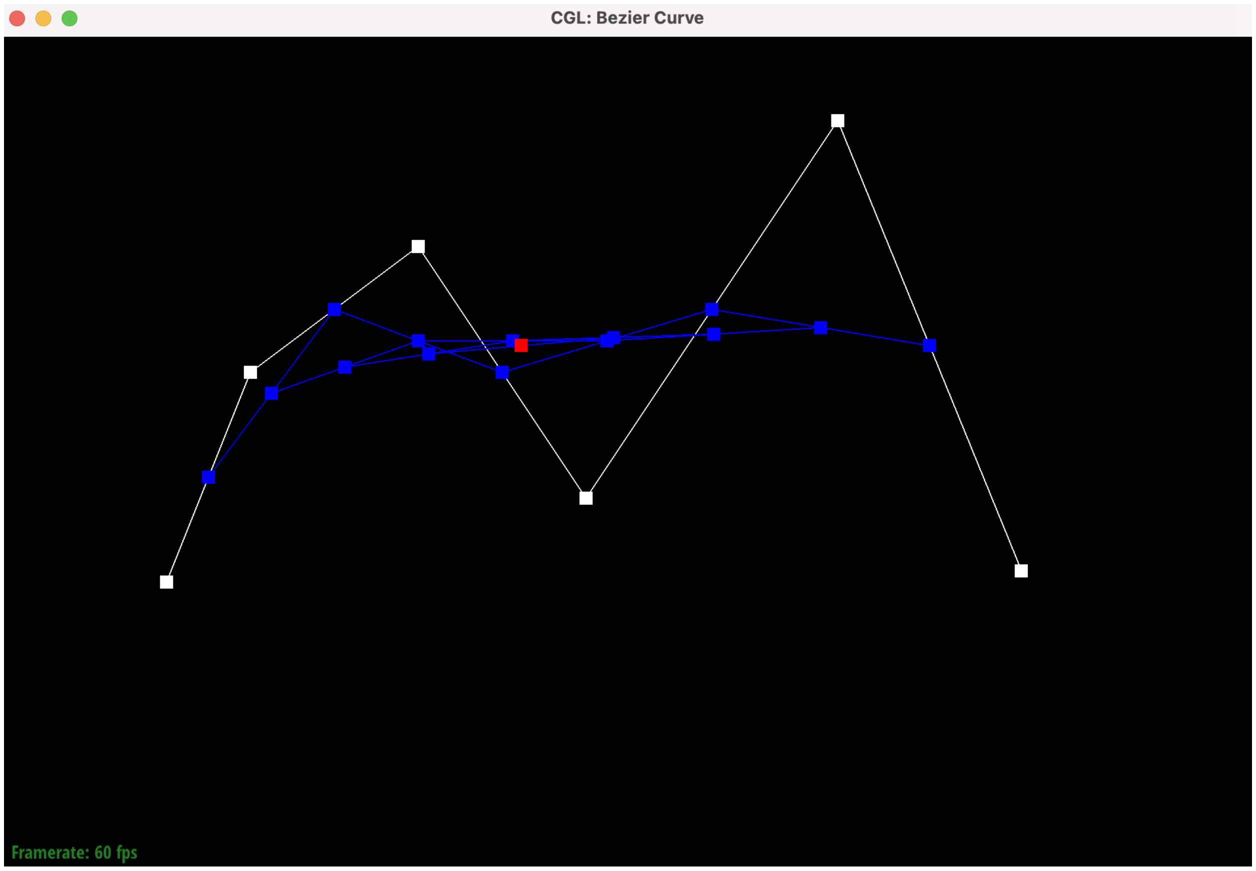
After 3 iterations:



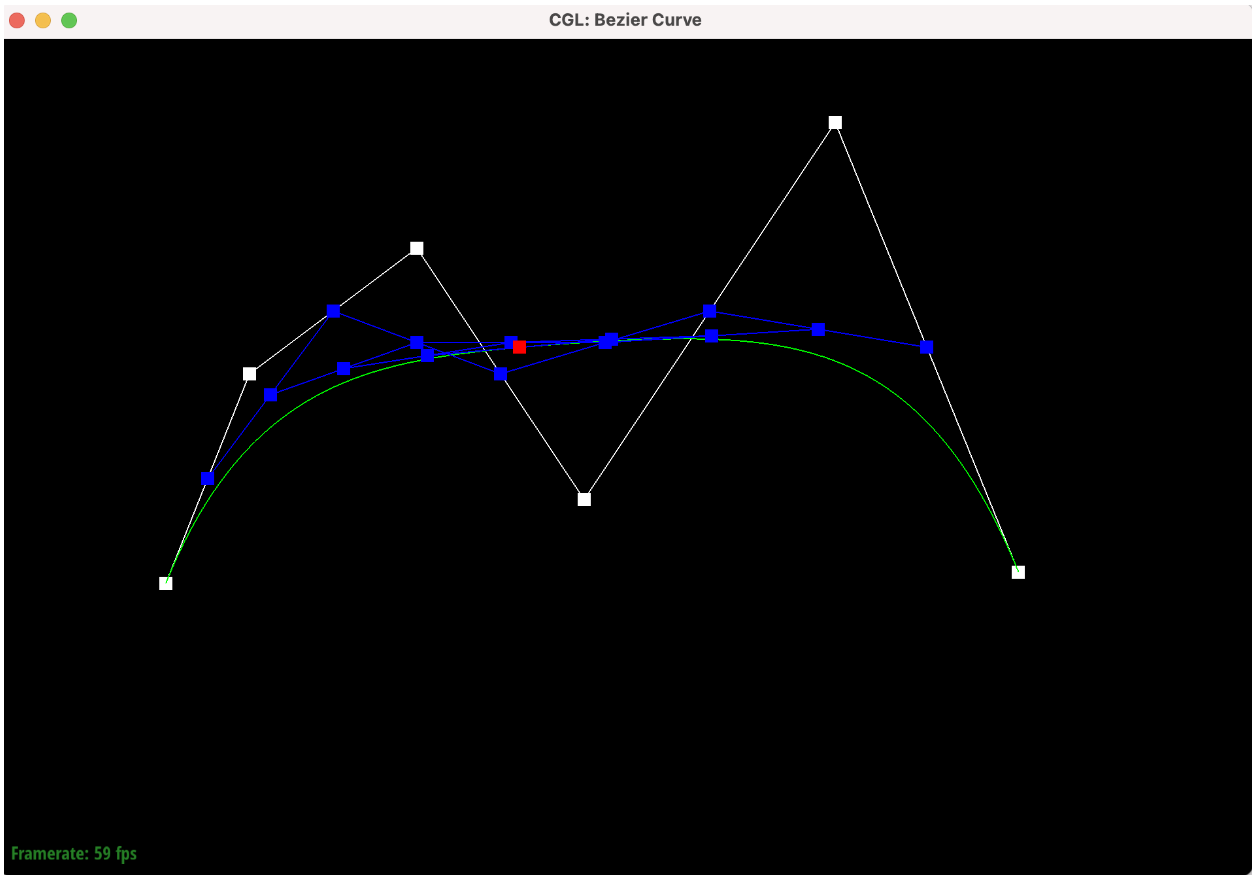
After 4 iterations:



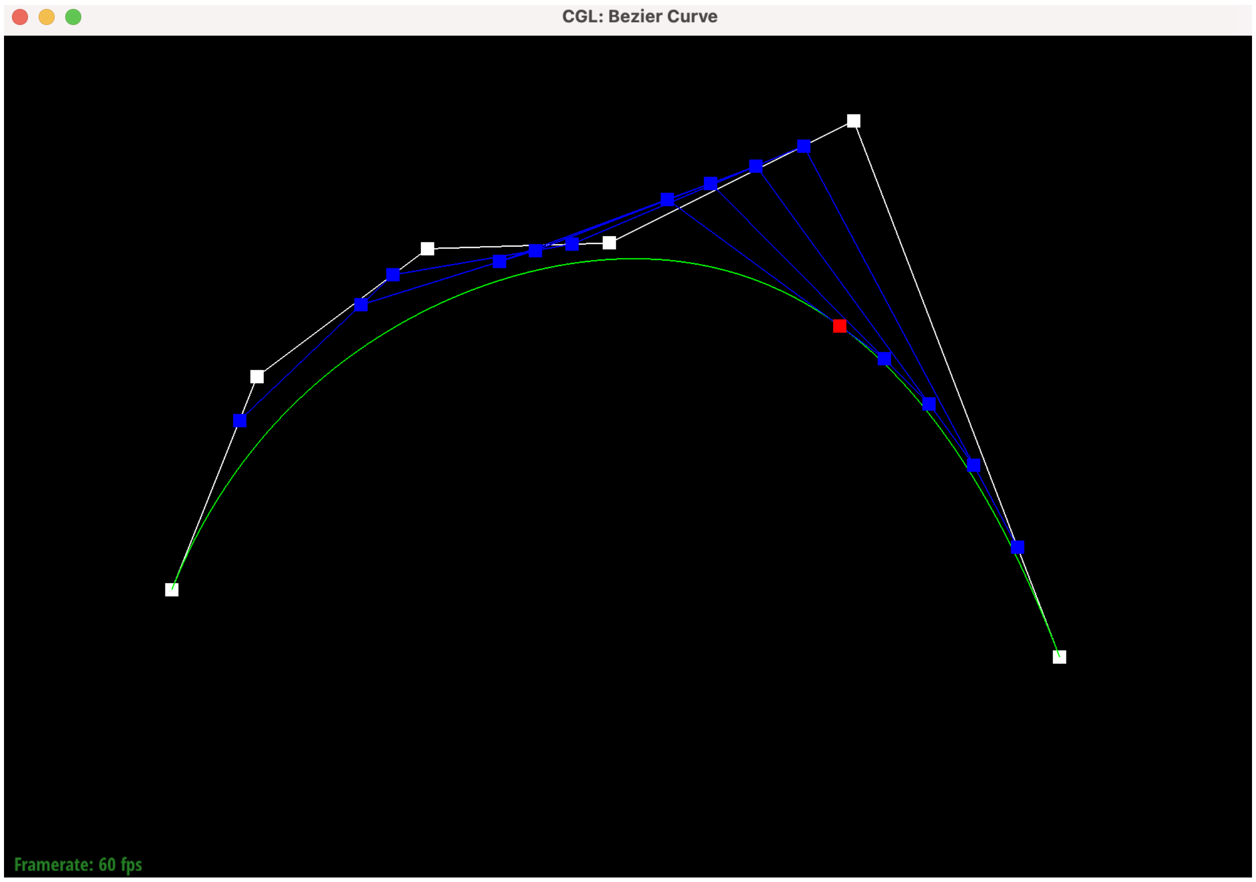
After 5 iterations:


CGL: Bezier Curve
Framerate: 60 fps

The final Beizer curve:



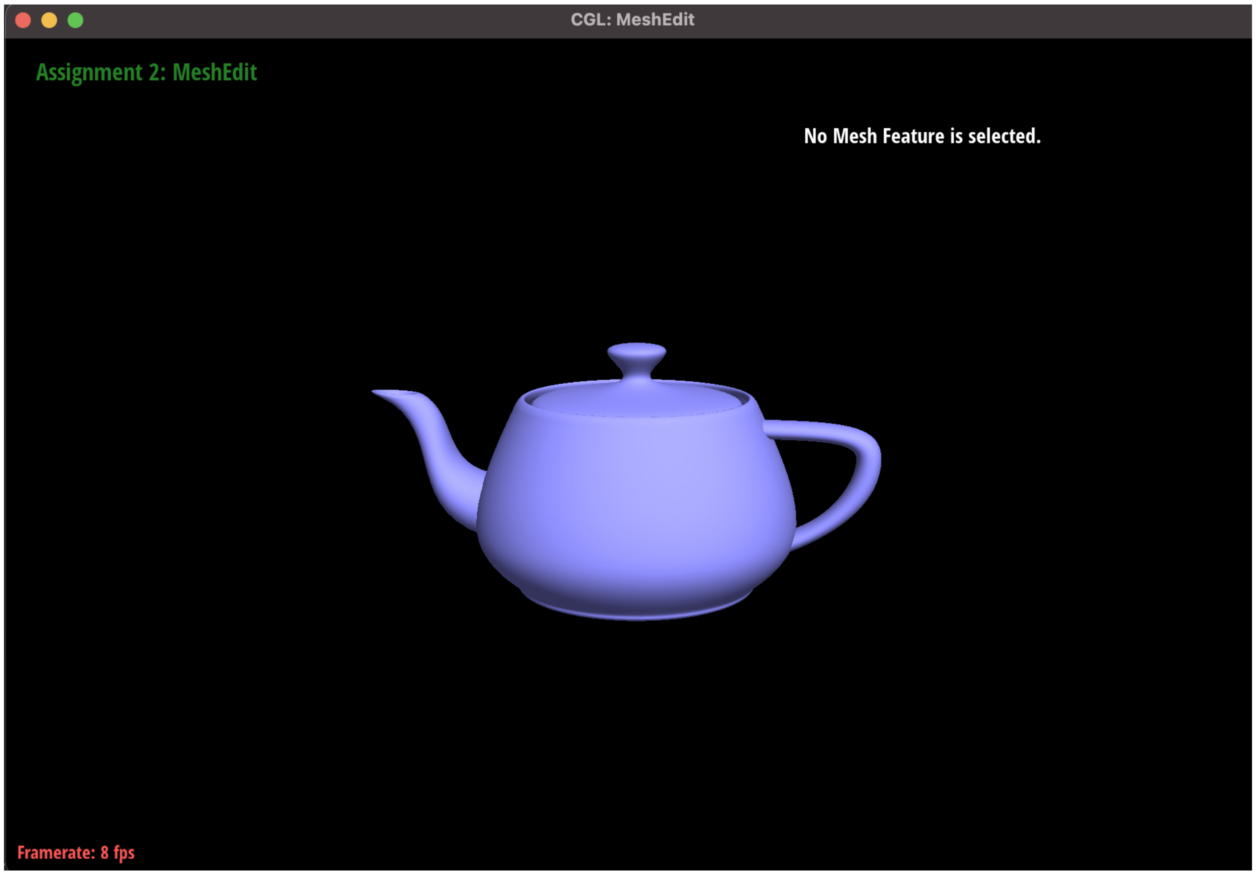
A slightly different Beizer Curve is shown below:



Section 1 Part 2:
In 2D De Casteljau's algorithm, we are given a vector of vector of control points as input. Let’s assume the length of the input is n (i.e. n vectors of control points). De Casteljau's algorithm first computes n Beizer curves (i.e. n final control points) using the parameter u. It then computes one final control point over these n points using the parameter v. By changing u and v, the trajectory of the final control point will form a surface known as the Beizer surface.

In our implementation, we first compute one final control point for each vector of control points according to the parameter u. This gives us a list of control points, and we compute one final control point over these points using the parameter v.

The screenshot of the teapot is shown below:

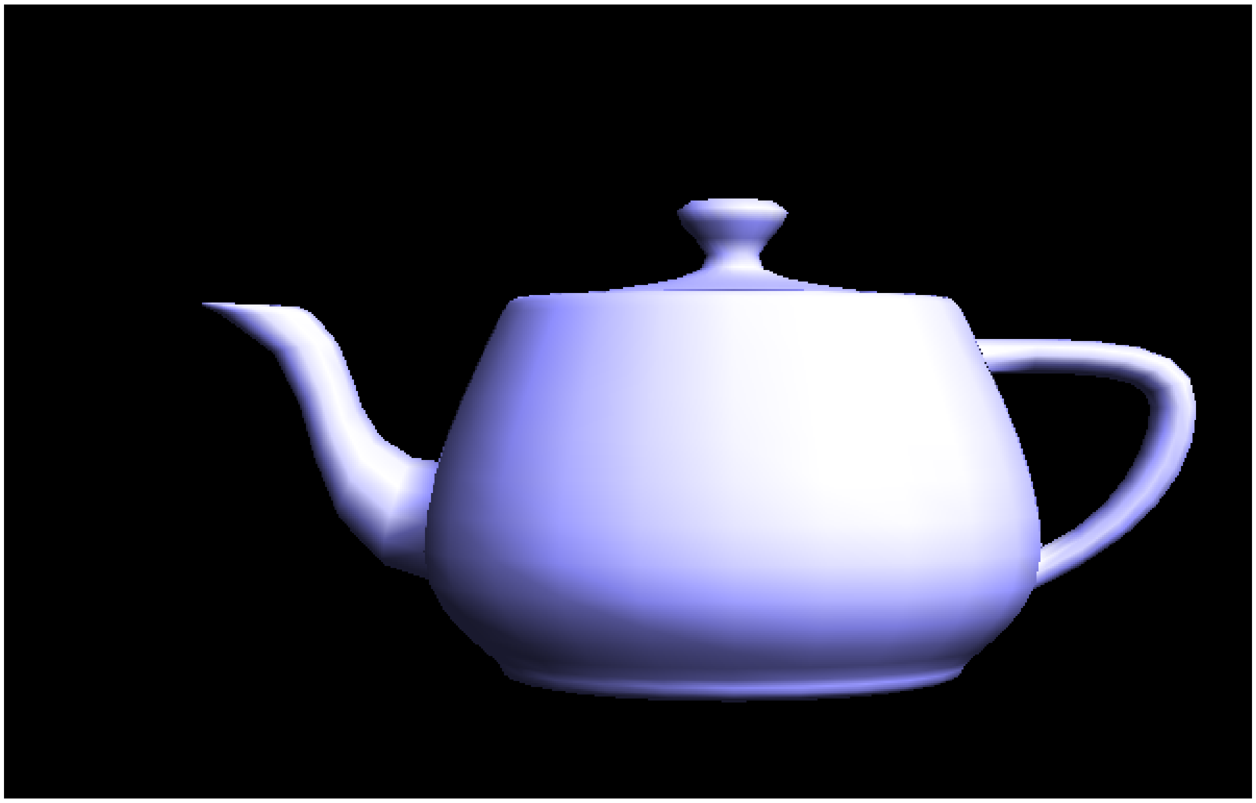


Section 2 Starts below:

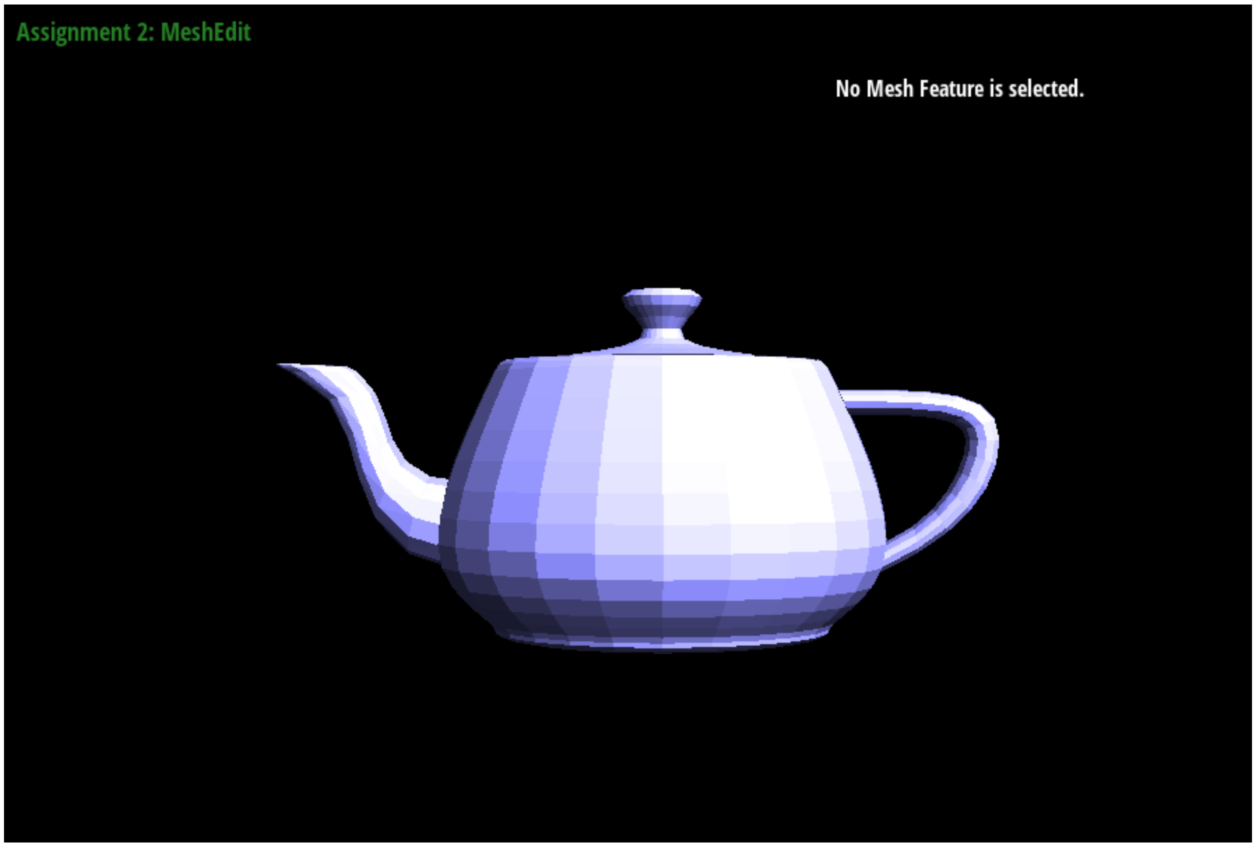
Part3:

In our implementation, we traversed through each face that this vertex is associated with using twins and their next. For each face, we first calculated its area using ½*normal vector's norm, where the normal vector is calculated by the cross product of the two edges(as vectors)from the face. During the face loop, we summed up each of the negative of the vector: (-1) *(area of the face) * (normal_vector of the face). Then we divided the total vector by the total area of all faces, normalized the length of that vector to 1, and returned it.

With vertex normals:

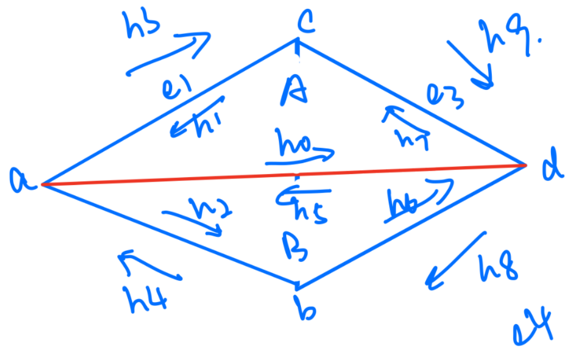


Without vertex normals:



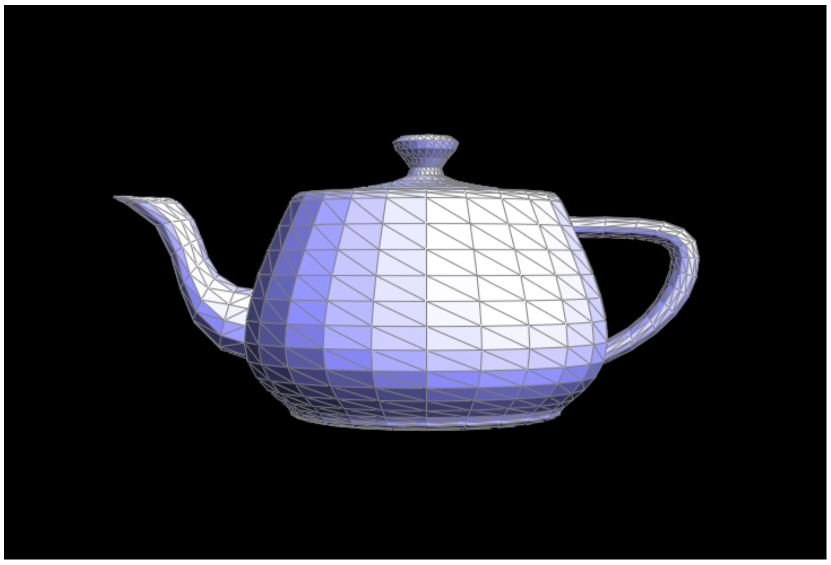
Part 4:
In our implementation, we first found all of the Halfedges in the two triangles that we need for the flip operation. We then exhaustively listed out each of those Halfedge's neighbors (twins, next, vertex, edge, face) in a table before the operation, and we figured out what those Halfedge's neighbours are **after** the flip operation. Also, for each vertex, edge, and face, we listed the Halfedge they are associated with before and after the flip operation. According to the list we created, we first updated Halfedge's neighbors using the function "setNeighbors". Then, we assign each of the vertex, edge, and face to its correct Halfedge.

Our naming reference is shown in the picture below: "h" represents halfedge, "e" represents edges, "a, b, c, d" are vertices, "A, B" are faces, the red edge is the original edge after flipping.

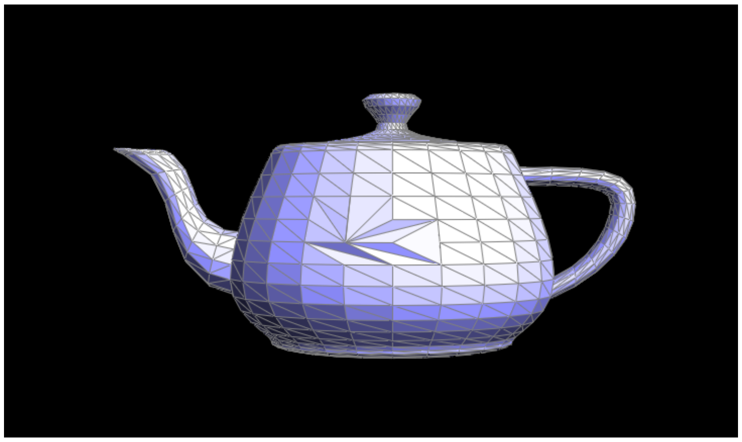


The effect of edge flipping is shown in the two pictures below.

The teapot **without edge flipping**:



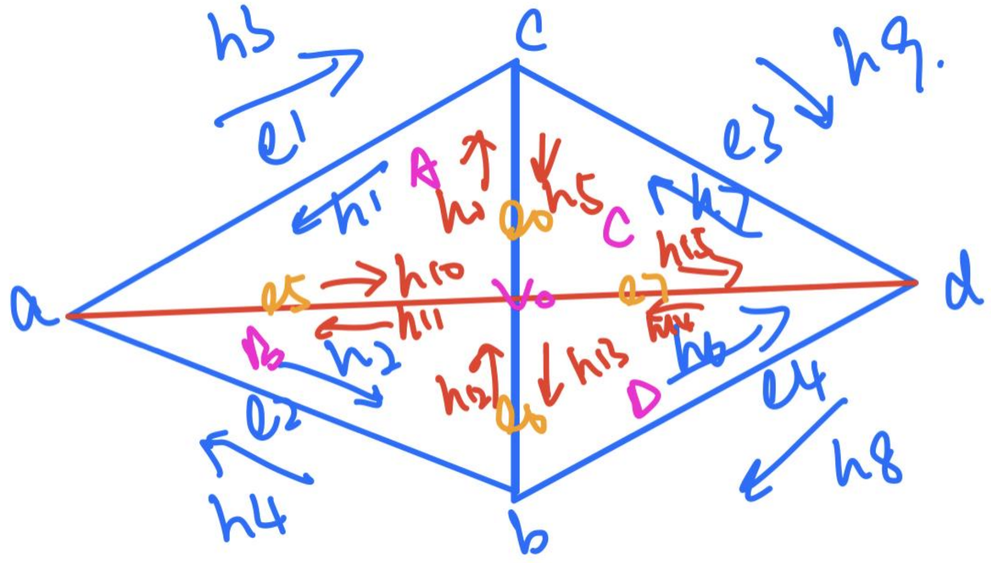
The teapot **after edge flipping**:



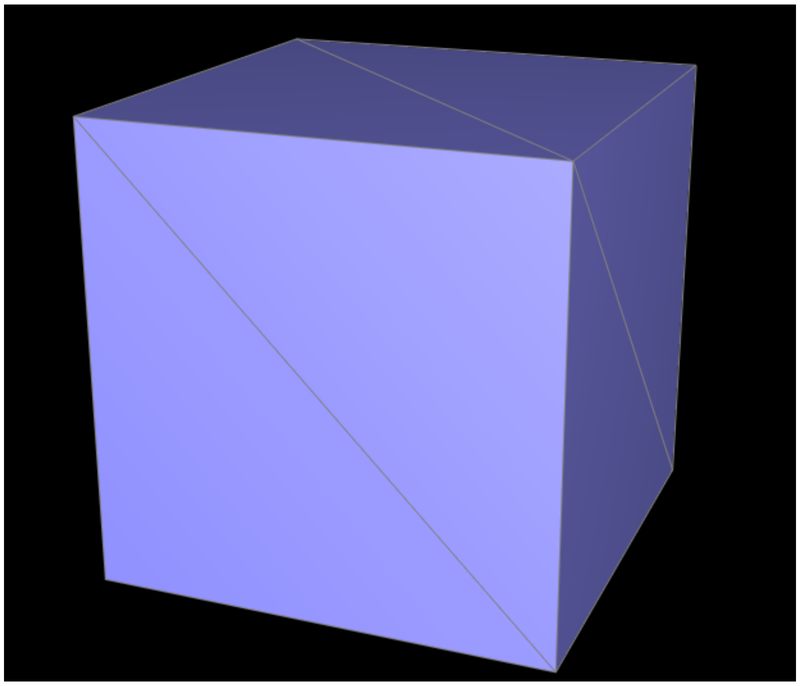
Since we exhaustively listed out each of the components and double-checked the correctness of our list before we coded, we didn't run into any significant bugs (hoo-ray!).

Part 5:

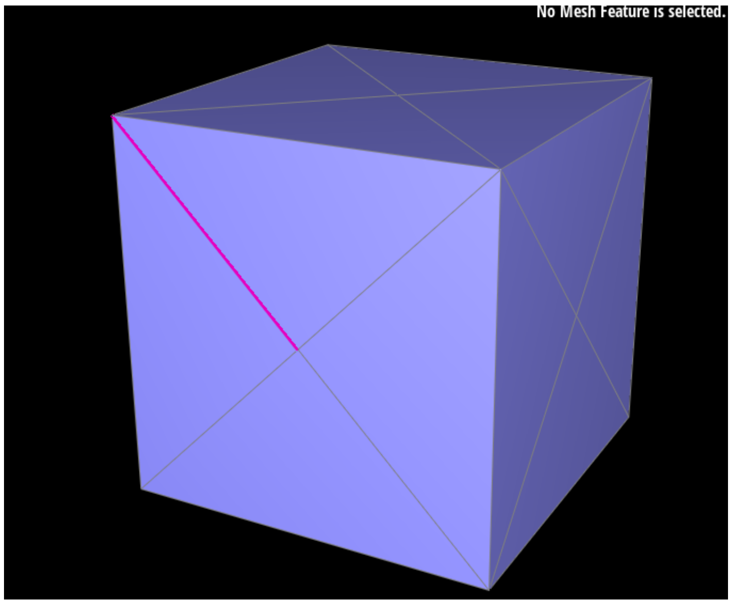
We followed the same strategy in part 4 to prevent bugs that are very hard to debug. We first assigned (and created new ones as needed) all halfedges, edges, vertices, and faces that are associated with the local mesh, accessed by the input EdgeIter e0. By drawing a picture representing this local mesh, we update each element's instance variables. That is, we used setneighbor to update Halfedges and we updated the halfedge that each vertex, edge and face is associated with. One important detail here is to set the position of the new vertex (not the newPosition!) to be the midpoint of the edge that is to be splitted. To support part 6, we also updated the "isNew" parameter for newly created edges and vertex. We followed the following screenshot to implement our edge split: "h" represents halfedge, "e" represents edges, "a, b, c, d" are vertices, "A, B, C, D” are faces, the red edge is the one that is newly generated.



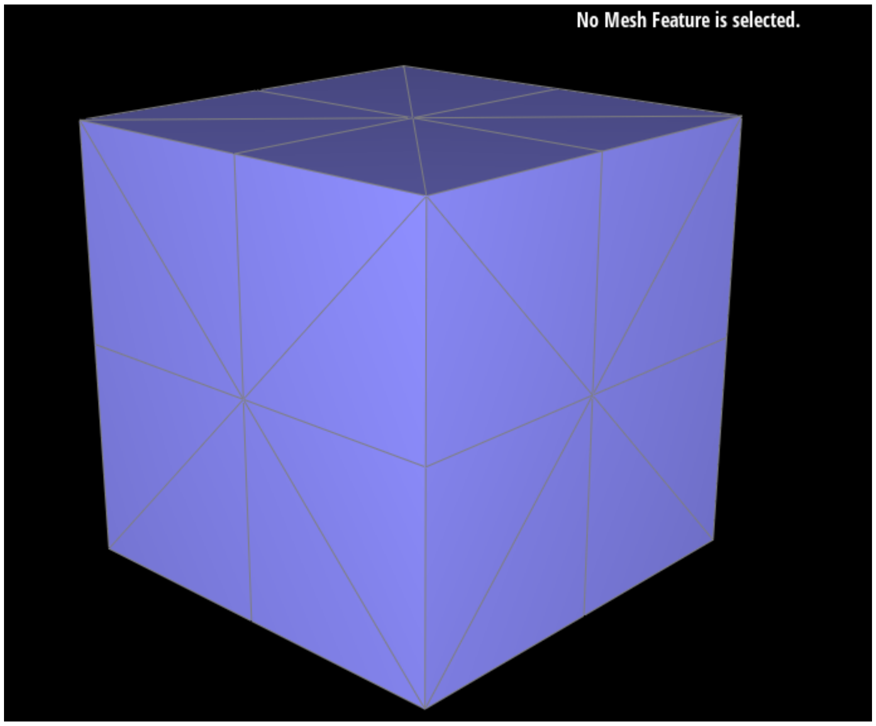
Before:



After some edge-splits:



After a Combination of edge splits and edge flips:



We did not experience any significantly interesting bugs, thanks to the comments by course staff!

Part 6:

Our algorithm:

1. For all the vertices which are currently in the input mesh, we computed their new position using the formula provided by the specs, and stored it in newPosition field of the vertices. During the loop, we also set the vertex's isNew to false(marking that it is an old vertex).
2. For all edges, we computed the position of the potential new vertices added this edge(from edgeSplit), using the formula provided by the specs, and stored it in newPosition field of this edge. During the loop, we also set the edge's isNew to false(marking that it is an old edge).
3. We apply edgeSplit to all the vertices of the old mesh, and set the created vertex's newPosition to its corresponding edge's new position we stored from Step 2.
4. For all edges, we flip this edge if the edge is new and it connects an old vertex to a new vertex.
5. For all vertices in the new mesh, we set its position to its new position, and set its isNew to false(just to be safe).

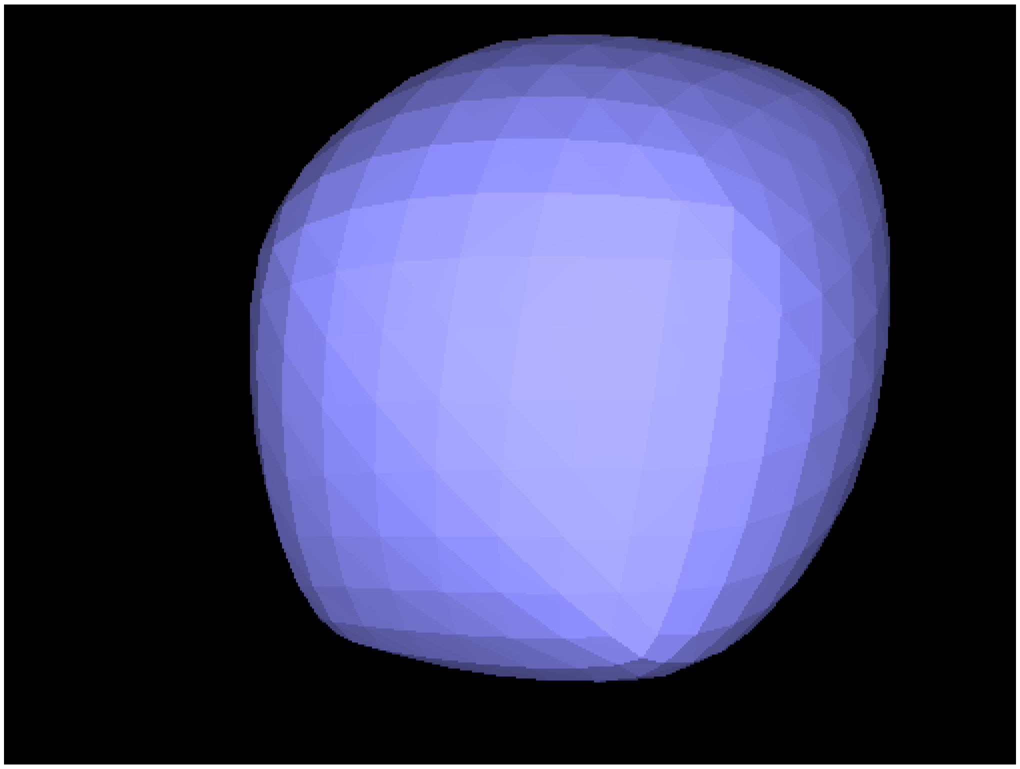
Interesting Bug:

One of the bugs we encountered is that in one of the vertices of the cube, its position didn't change after subdivisions, producing a sharp edge in a sphere. It turns out that we didn't correctly calculate the multiplier "u" due to integer division. When we wrote: "float u = 3/16", u is equal to 0. We fixed it by writing: "float u = 3.0/16.0".

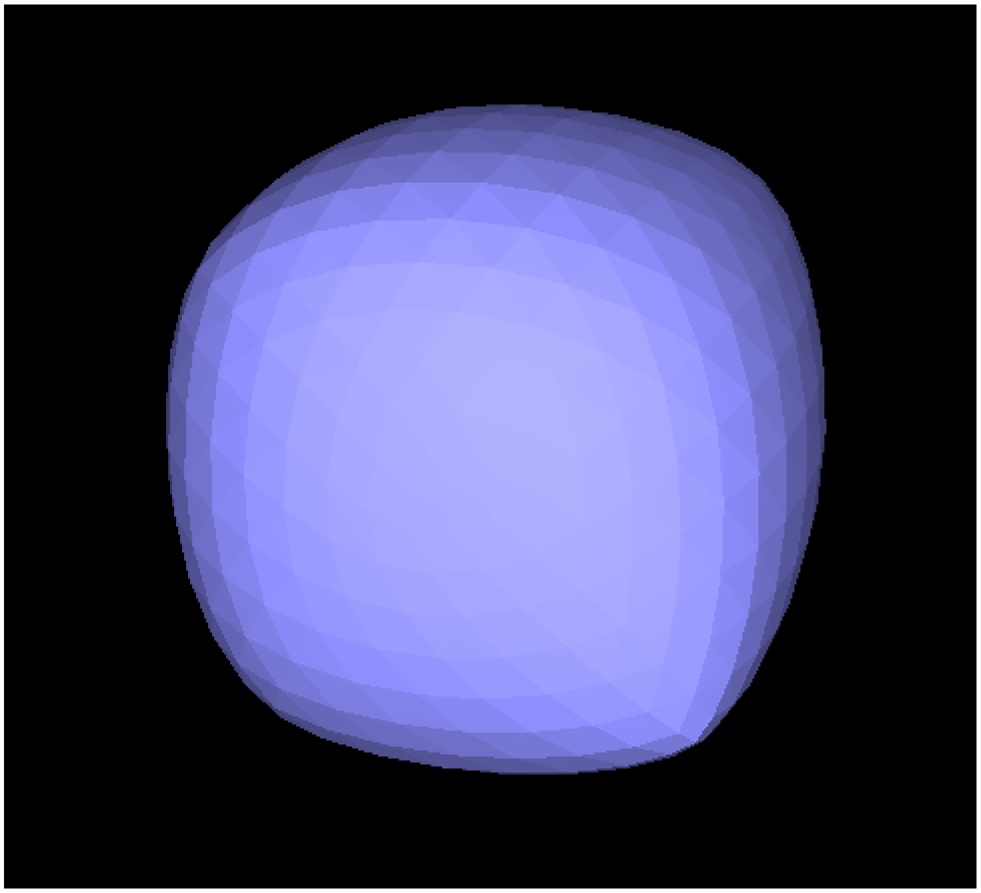
Presplitting before loop subdivision:

With pre-splitting, the sharp edges and corners seem to even out, to be replaced by multiple duller edges(of more triangles). Pre-splitting edges would for sure create more triangles locally, so it does reduce the sharpness effect by a bit.

The picture below shows the mesh after 3 subdivisions, **without pre-splitting**. We can clearly see sharp edges and corners near the corners of this shape.

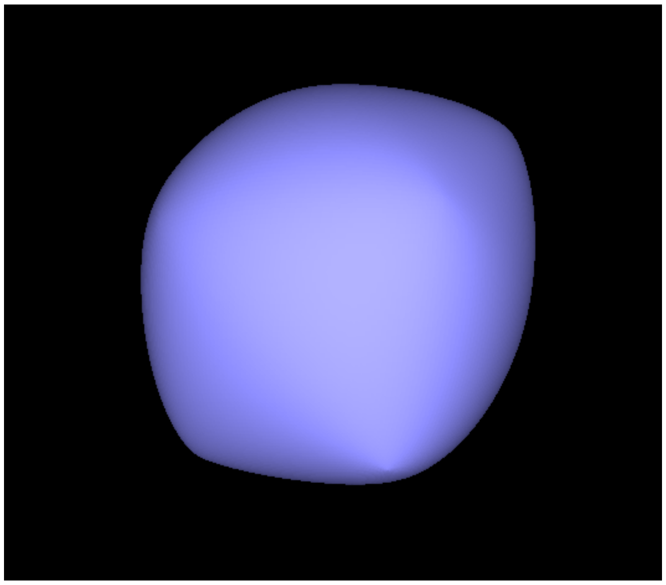


The picture below shows the mesh after 3 subdivision, **with pre-splitting**. Now the edges and the corners are less sharp compared to those without pre-splitting.

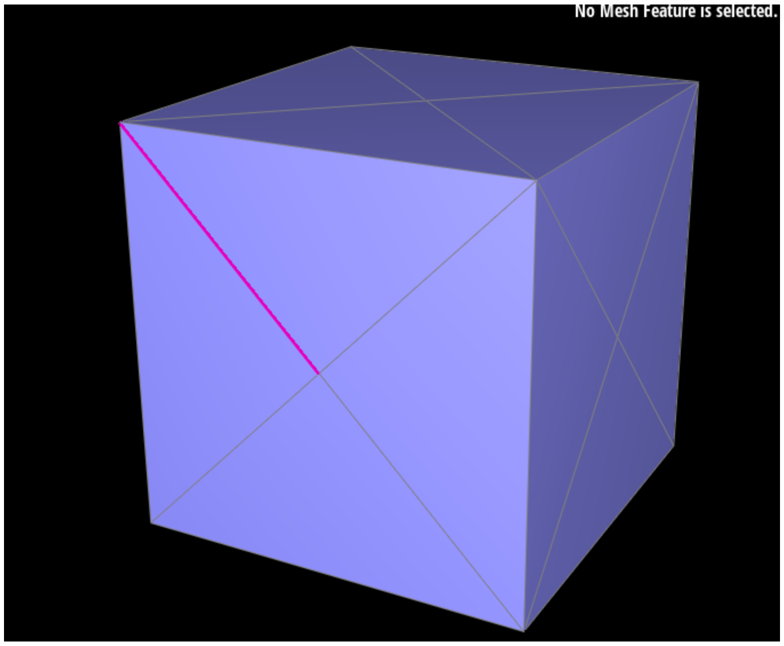


The effect of pre-processing on cube subdivision:

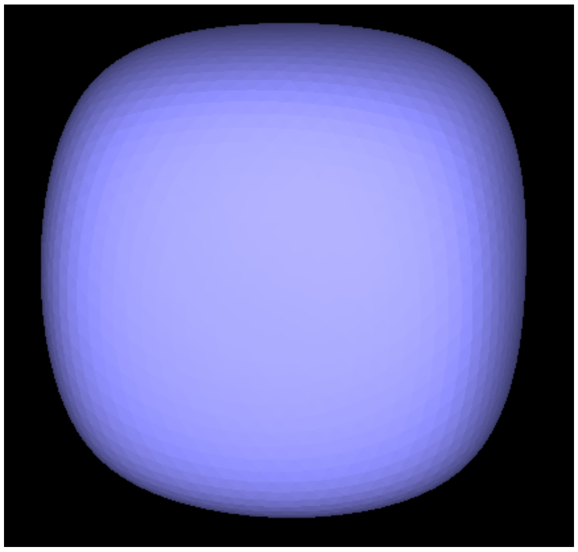
Pre-processing also helps alleviate the distortion effect of subdivision. As shown in the picture below, without pre-processing, the cube looks no longer like a cube: the upper-right corner seemed to be stretched out more than the lower-right corner.

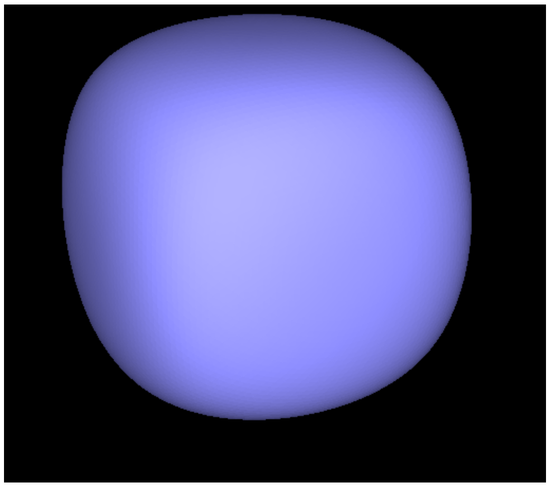


To alleviate this problem, we first split and flip the edges to make the degree of each vertex to be the same (i.e. n = 6), as shown in the figure below.



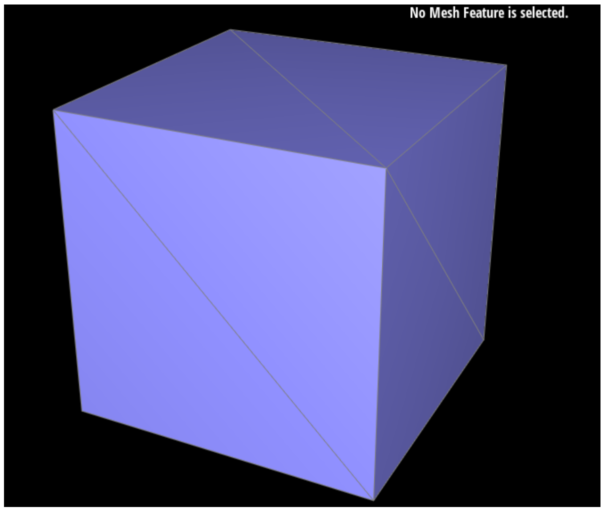
Then, we perform loop subdivision, and the resulting cube (as shown in the two pictures below, each from a different angle) looks less distorted than before, as each face is being stretched more symmetrically.





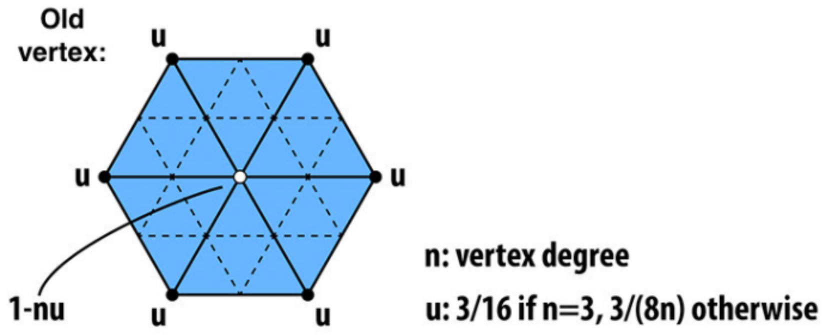
Explanation:

In the default cube, each of the 6 faces are splitted into two triangles as shown in the picture below:



Some corners of the cube have a degree of 5, while some have a degree of 4. Thus, when we subdivide along the edges, updated positions for vertices with degree 5 would be calculated differently than those with degree 4, using lecture's formula as shown below, as u and n would be different. This causes asymmetric update of positions, stretching each corner differently and resulting a sharper edge / corner on vertices with degree 4.

Position updating formula for old vertices, cited from Lecture 8:



To alleviate this issue, we split the diagonal edges to create new edges so that each corner of the cube has the same degree of 6, resulting in a symmetric update on each vertex's position, since each vertex has same $n \: and \: u$ (i.e. all vertices have the same "n", and all vertices have the same "u"). This reduces the distortions and thus makes the cube more cube-like.