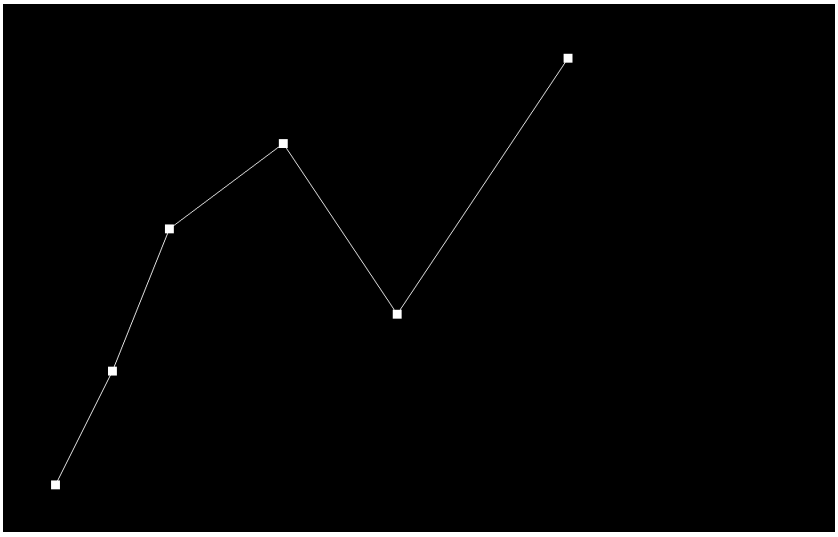
Overview

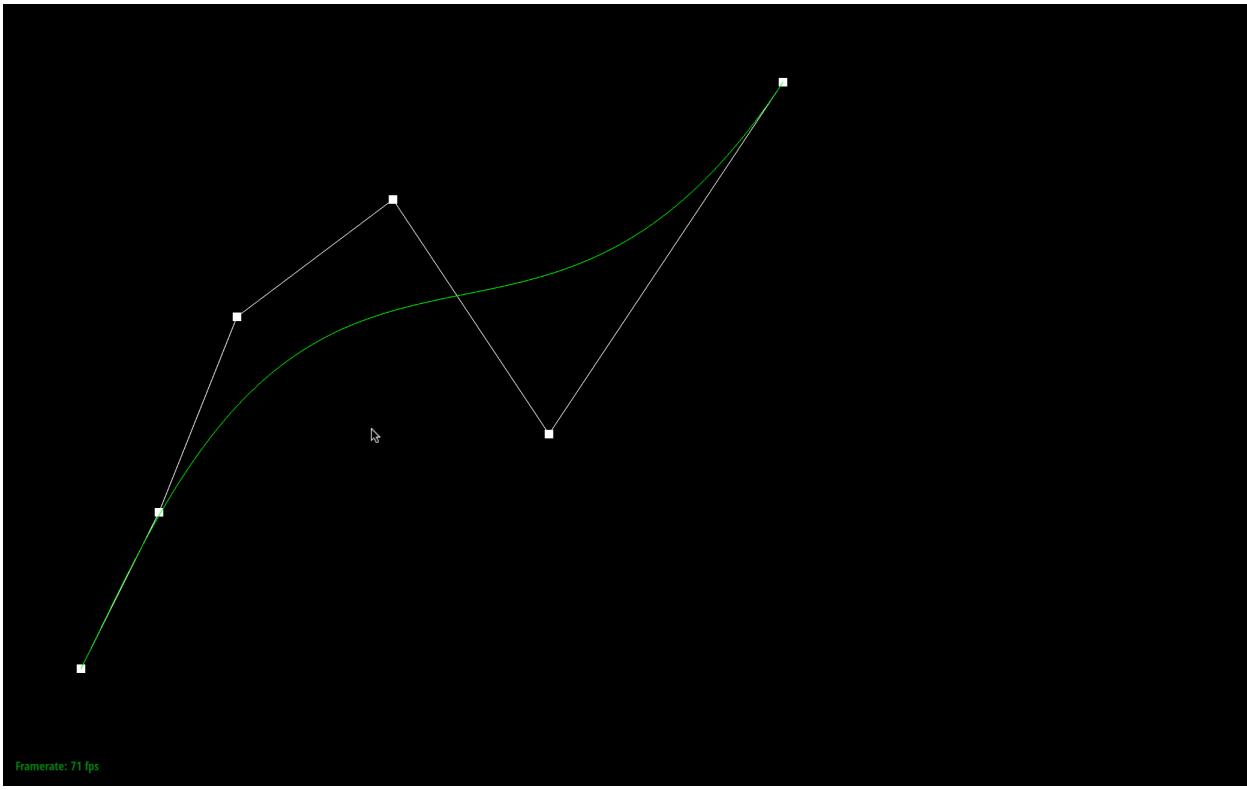
Give a high-level overview of what you have implemented in this assignment. Think about what you have built as a whole. Share your thoughts on what interesting things you have learned from completing this assignment.

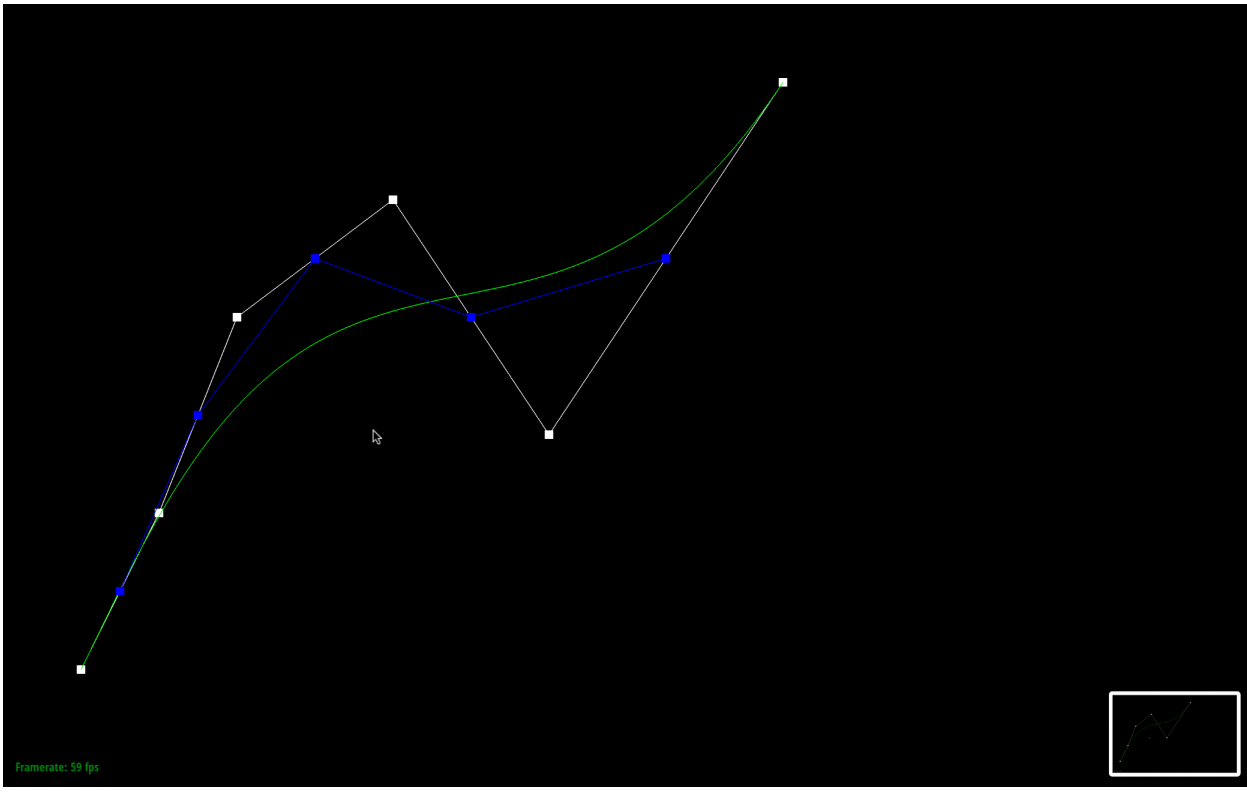
In the first part of the project we learned about Bezier curves and shading. It was a good exercise to see the Beizier curves and how it relates to t, u, and/or v and the initial as well as intermediate control points. In the second part of the project, the most interesting things we learned is how powerful splitting and flipping edges are. We were able to “sculpt” with just those two operations. The loop subdivision was simply a formulaic approach to “sculpting”.

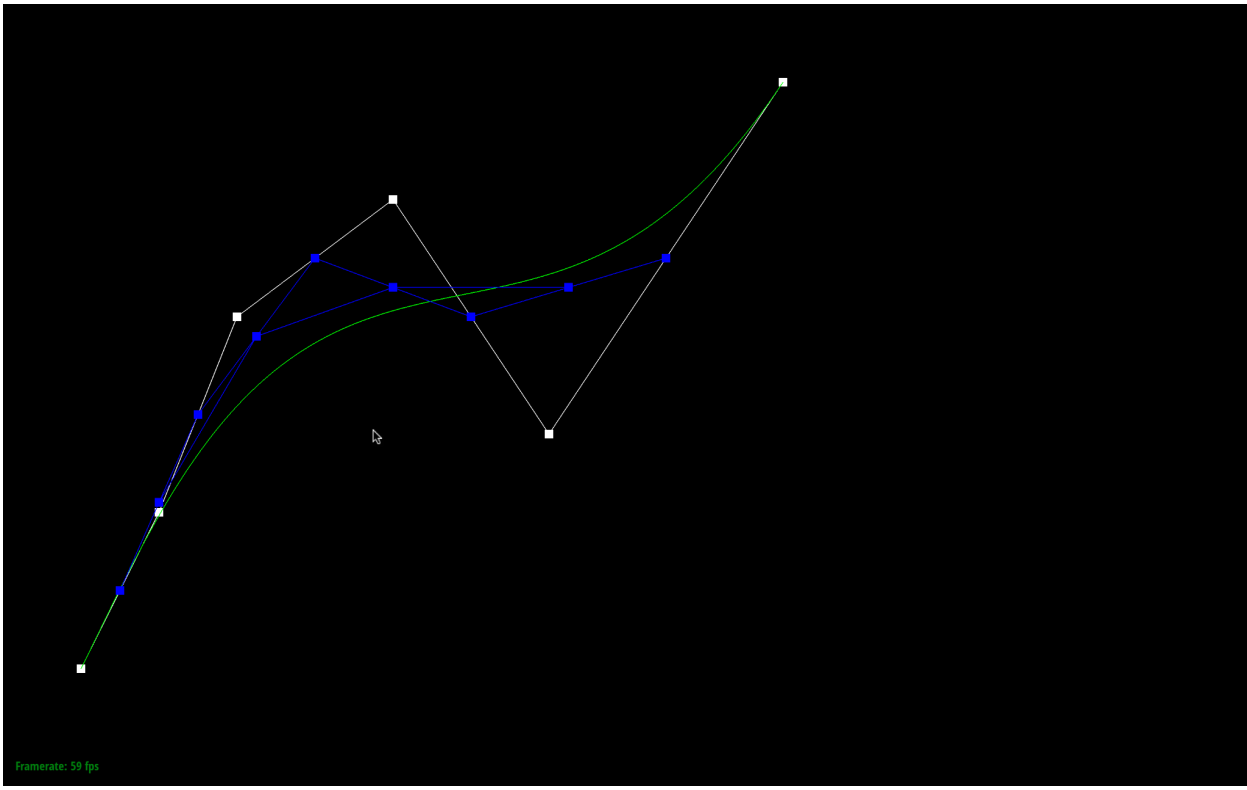
Part 1

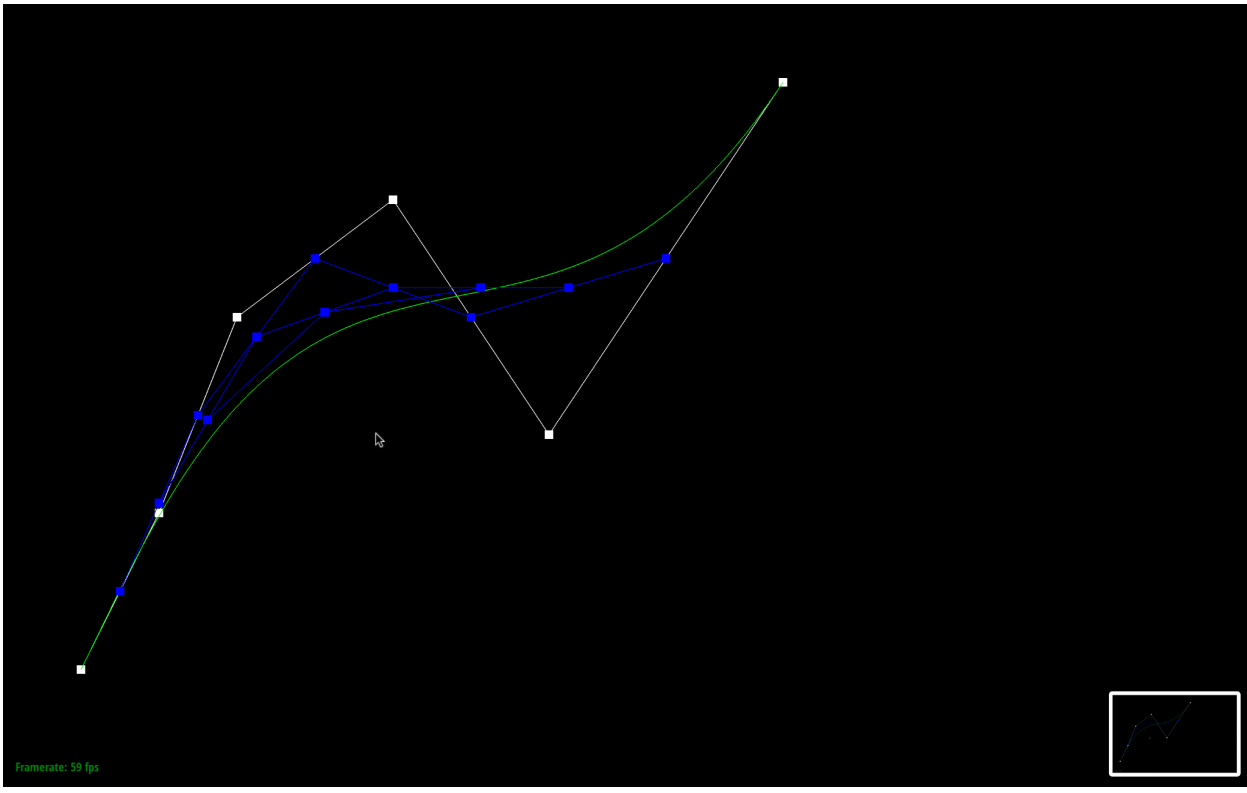
- Briefly explain de Casteljau's algorithm and how you implemented it in order to evaluate Bezier curves.
    - De casteljau’s algorithm helps us find a point on a bezier curve. We do so by first identifying n control points, then we perform linear interpolation and evaluate n - 1 intermediate control points at some set point (for all) along the line between the control points. For implementation, we defined a subroutine for the BezierCurve class called BezierCurve::EvaluateStep() which takes n control points and float t (for determining lerp location) and returns n - 1 intermediate points created through lerping. We will be using this simple function to do part 2 of the project.
- 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.

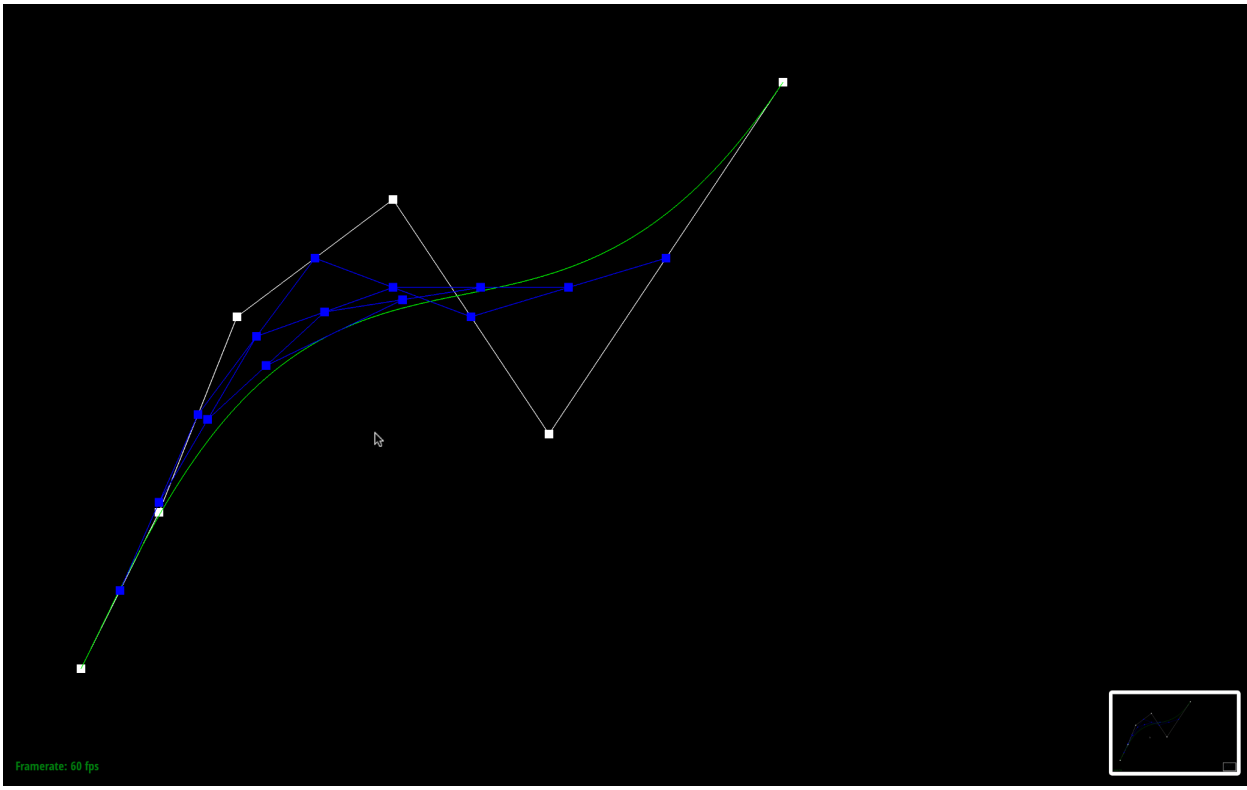
- 
- Show screenshots of each step / level of the evaluation from the original control points down to the final evaluated point. Press `E` to step through. Toggle `C` to show the completed Bezier curve as well.

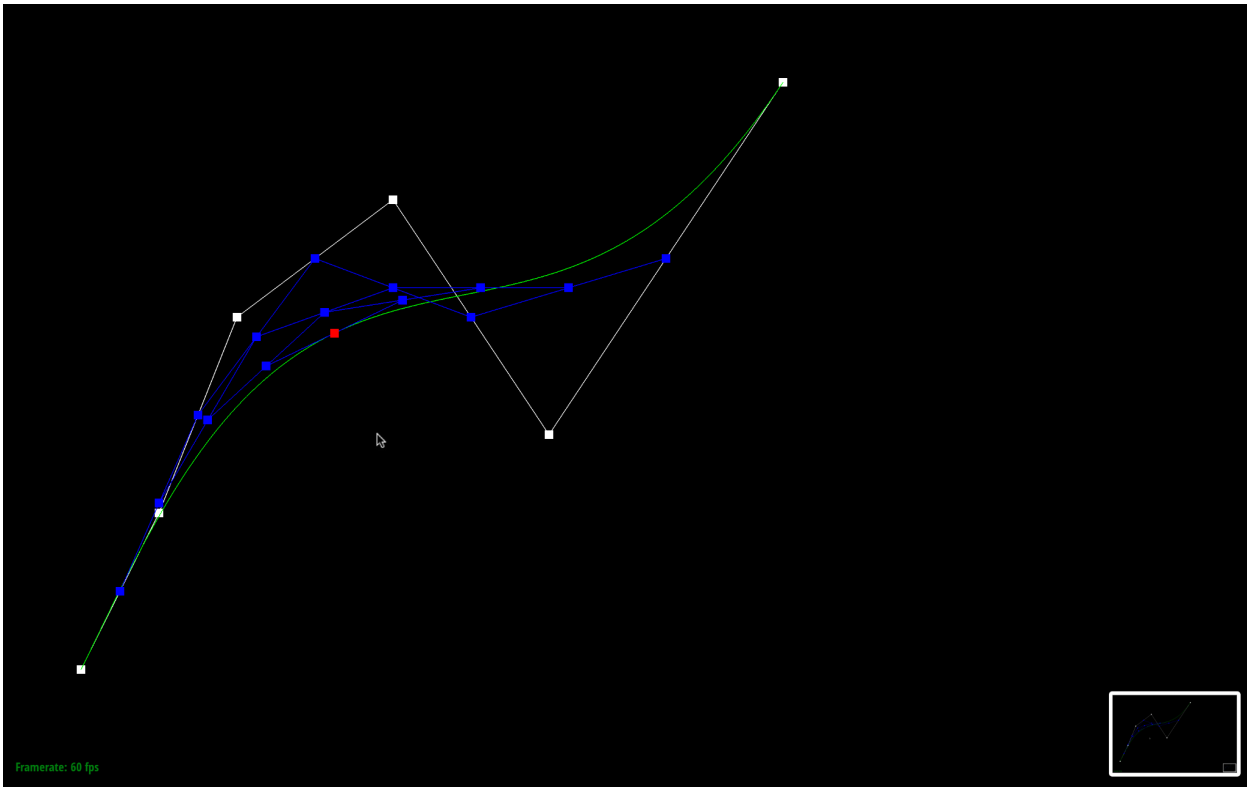

Framerate: 71 fps


Framerate: 59 fps

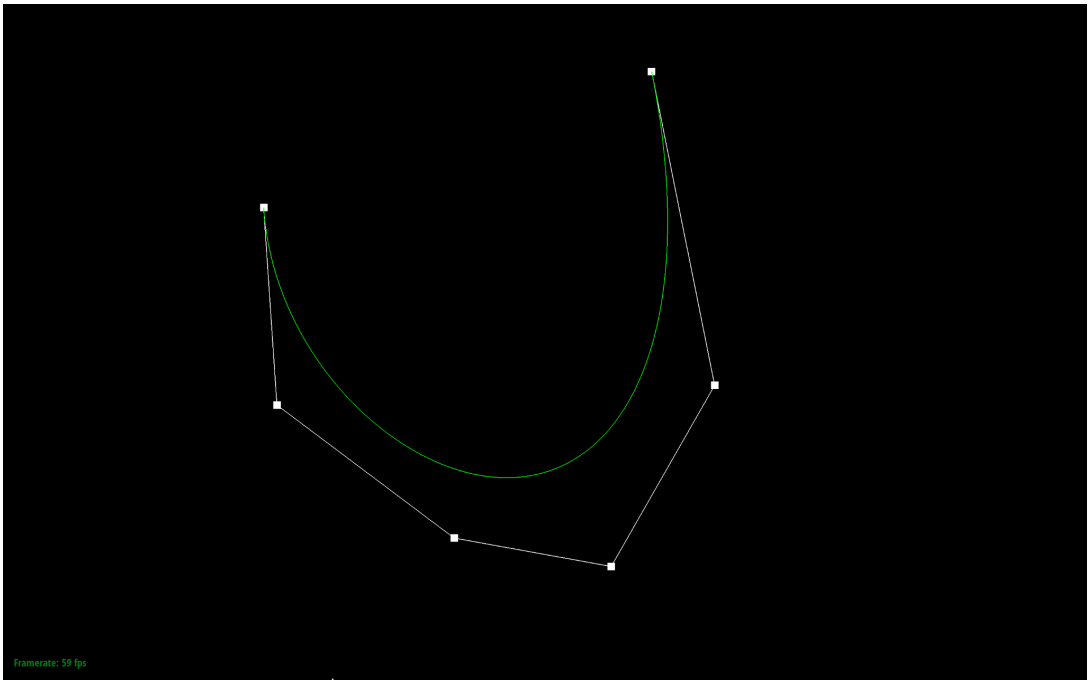

Framerate: 59 fps

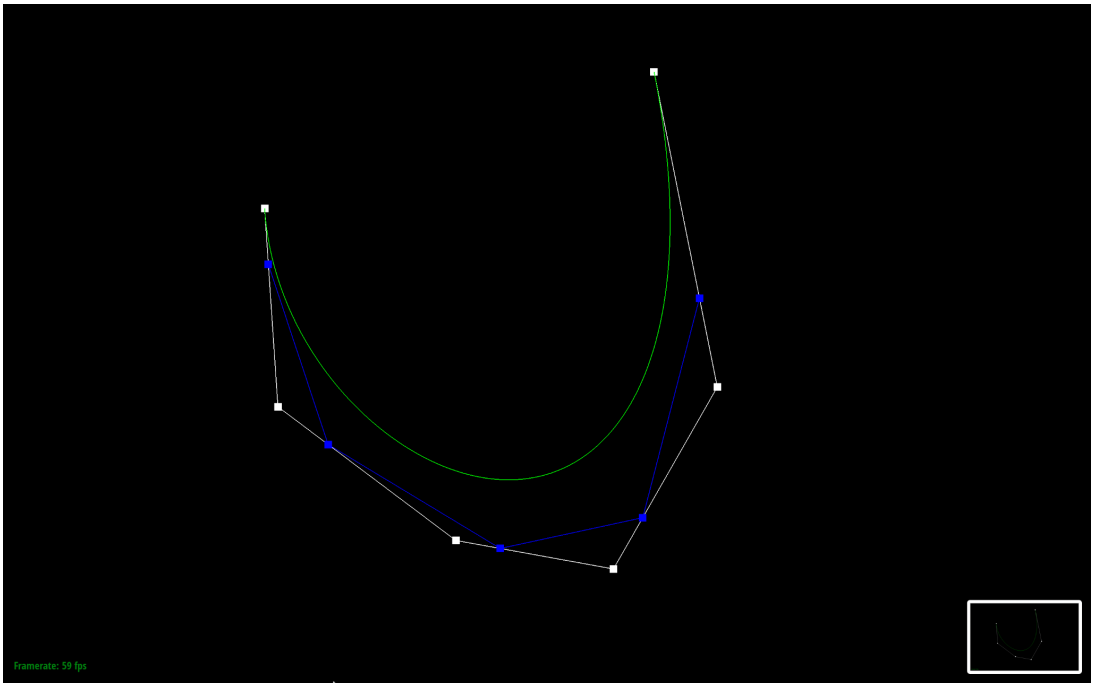

Framerate: 59 fps

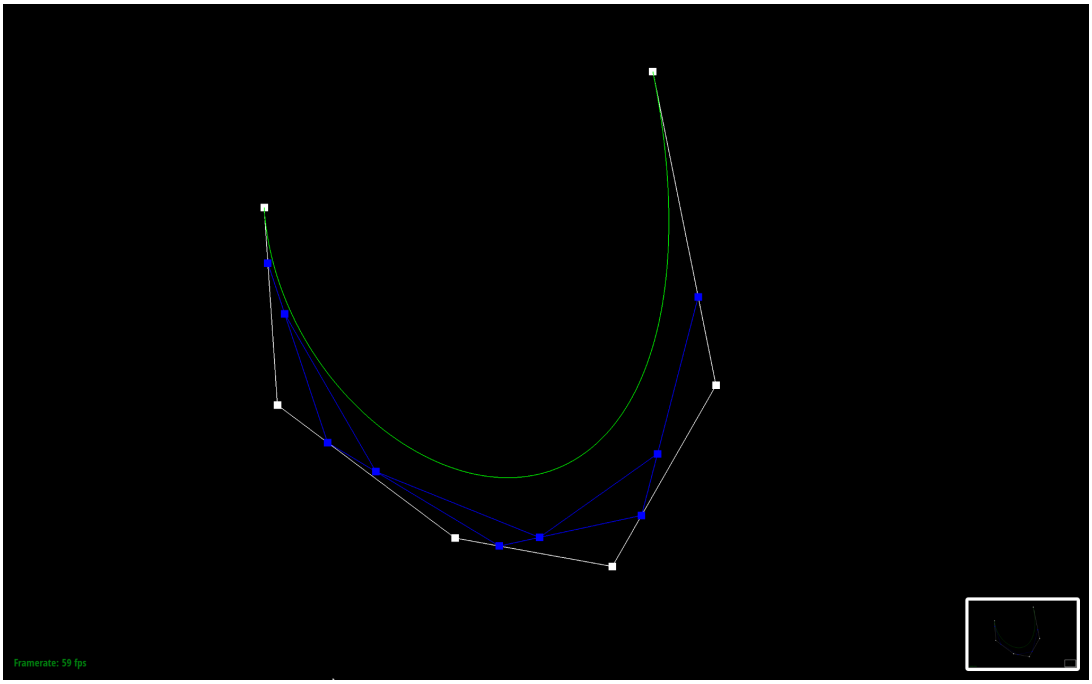


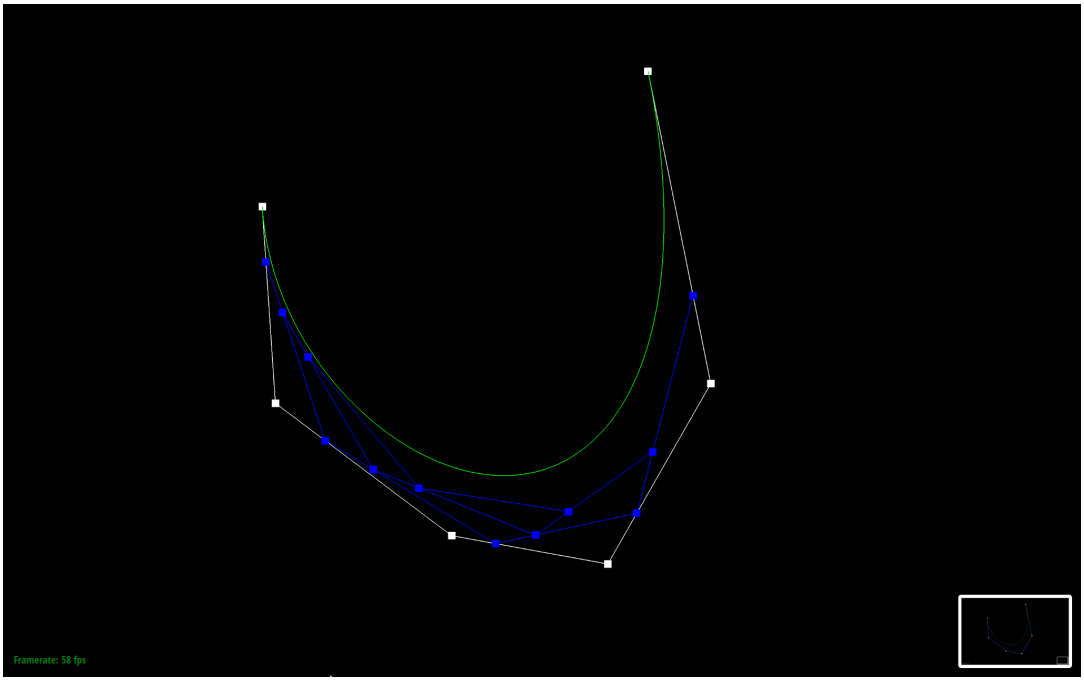


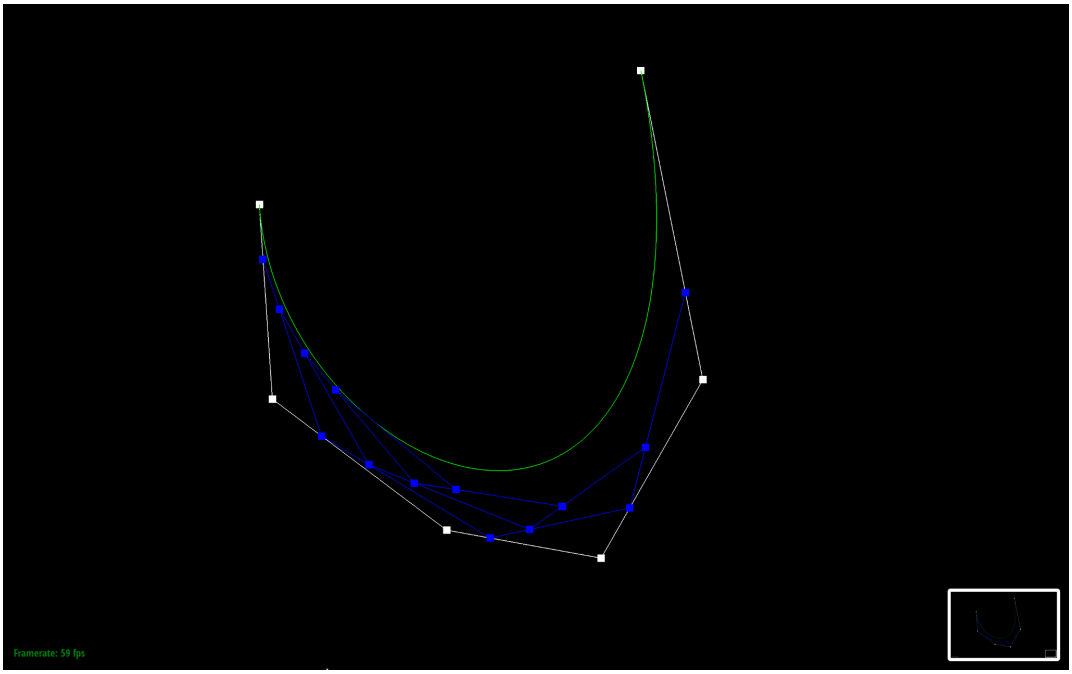
- Show a screenshot of a slightly different Bezier curve by moving the original control points around and modifying the parameter t via mouse scrolling.

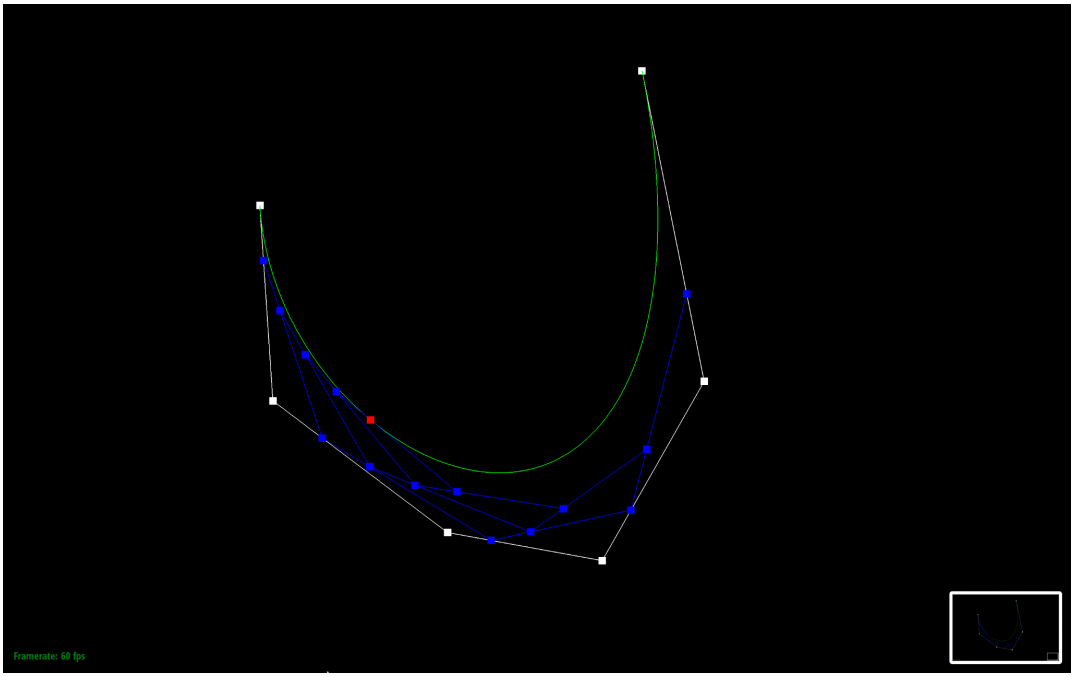
○ 
Framerate: 59 fps

○ 
Framerate: 59 fps

○ 
Framerate: 59 fps

○ 
Framerate: 58 fps

- 

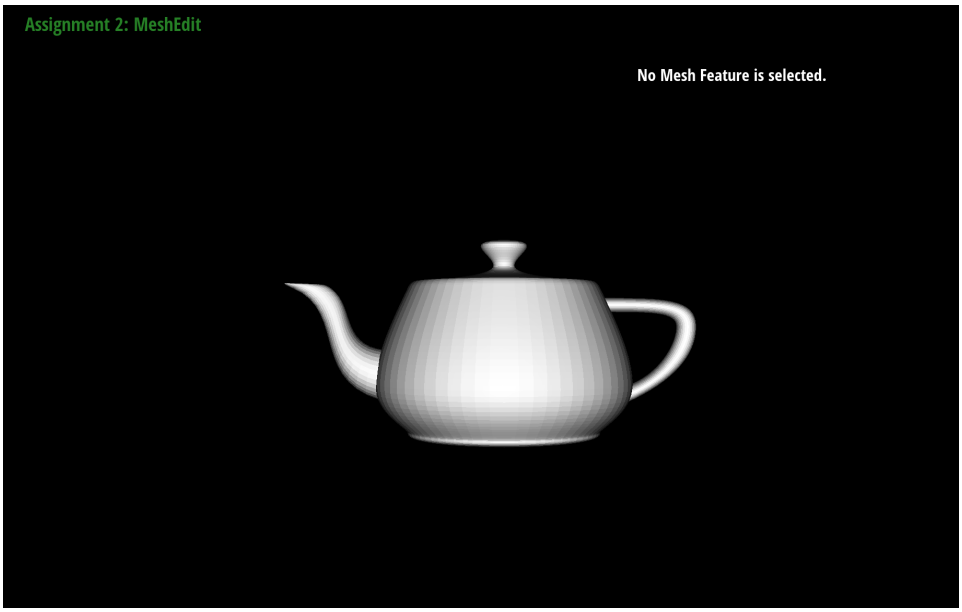
- 

Part 2

- Briefly explain how de Casteljau algorithm extends to Bezier surfaces and how you implemented it in order to evaluate Bezier surfaces.
    - De Casteljau's algorithm extends to Bezier surfaces because a Bezier surface can be thought of as a bunch of 1D bezier curves. The implementation is very similar to part 1 except we used a 3D vector. We

create controlPoints.size() many Bezier curves initially using u to determine lerp location. Next, we will execute 1D De Casteljau's algorithm on the created Bezier curves using v to determine lerp location.

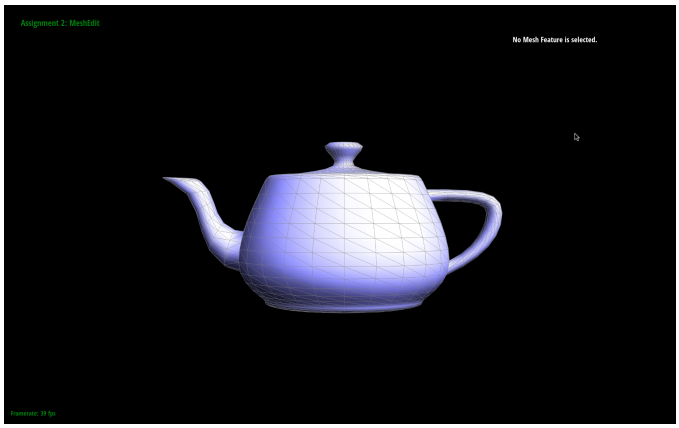
- Show a screenshot of `bez/teapot.bez` (**not** `.dae`) evaluated by your implementation.
  - 

  - 

Part 3

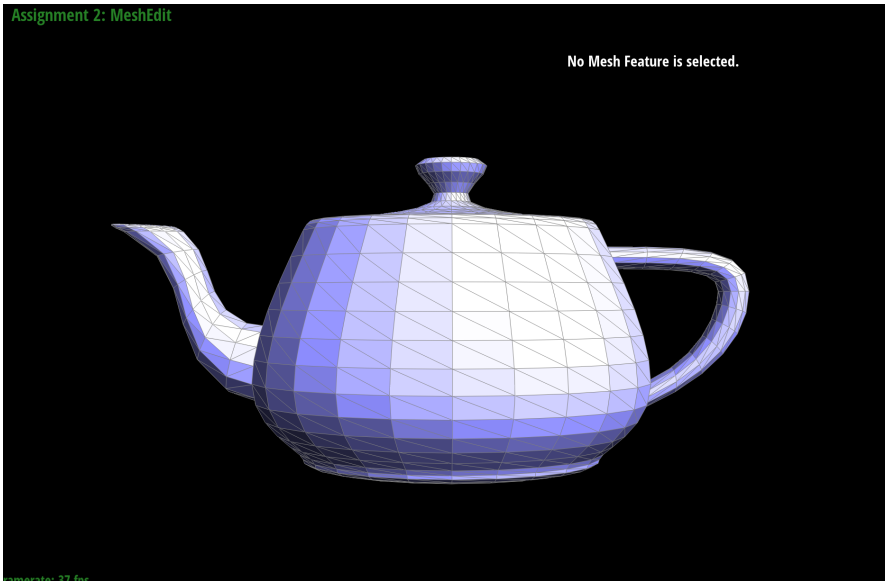
- Briefly explain how you implemented the area-weighted vertex normals.

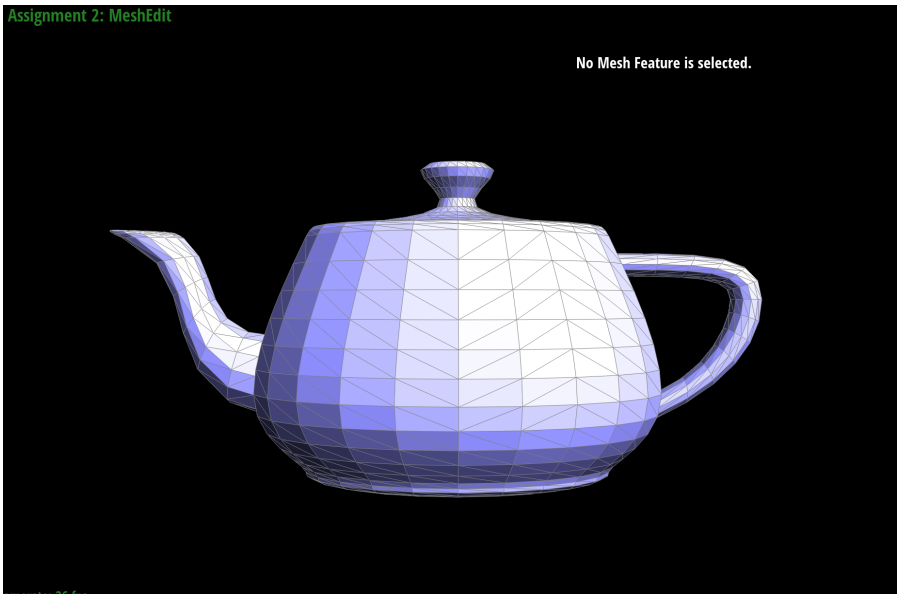
    - To implement the area-weighted vertex normals, we traversed all the faces with the vertex beginning with the half edge pointer of the vertex. We then computed the running-sum of the cross product between the two vectors of each face. Lastly, we normalized the summed vector to be a unit vector. To get to the next relevant halfedge, we called h -> next() -> twin() until we reached the original halfedge.
- 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.

With	Without
	
	

Part 4

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

    - First, we drew the reference image along with all of the elements that we needed to collect before the edge flip and set variables accordingly to the picture. After accounting for boundaries, we set all of the half edges to the proper ones in reference to the elements after the edge flip by using setNeighbors, which sets the next, twin, vertex, edge, and face of all the half edges. After setting the rest of the edges, vertexes, and faces, we return the flipped edge.
- Show screenshots of a mesh before and after some edge flips.
    -


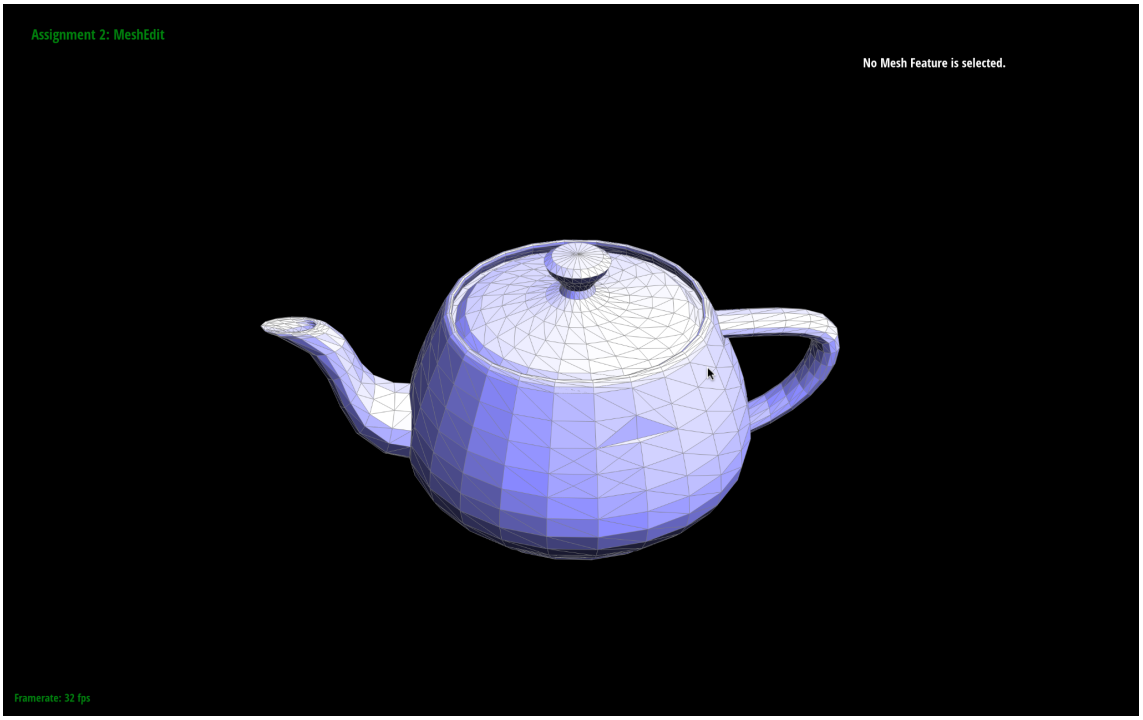
    -


Part 5

- Briefly explain how you implemented the edge split operation and describe any interesting implementation / debugging tricks you have used.
    - Just like edge flipping, first, we drew a reference image with all of the elements mapped out in order to collect them as variables. Then we created all of the new elements that were needed for the split, including a vertex, edges, half edges, and faces. we also set the position of the vertex to the midpoint between the two vertexes. Afterward, we followed the reference image we drew after the split and set all of the elements accordingly. One thing we had to debug was to make sure that we found the elements that had stayed the same after the split, where we had to pass a pointer to instead.
- 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.

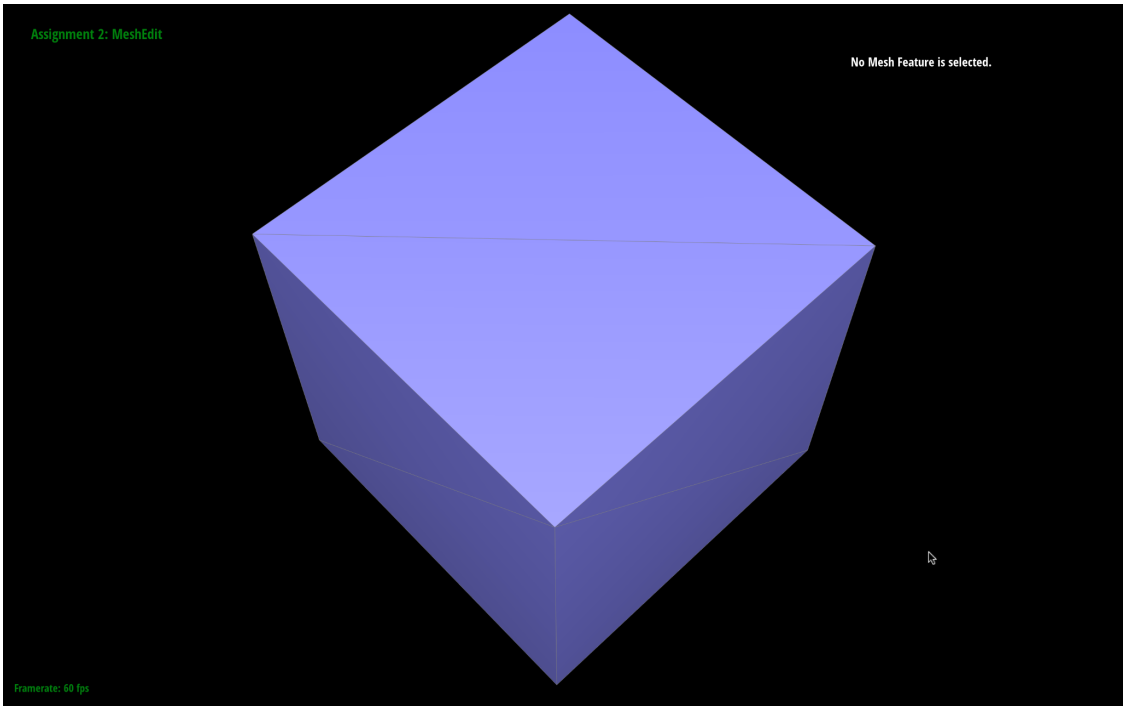
- 

Part 6

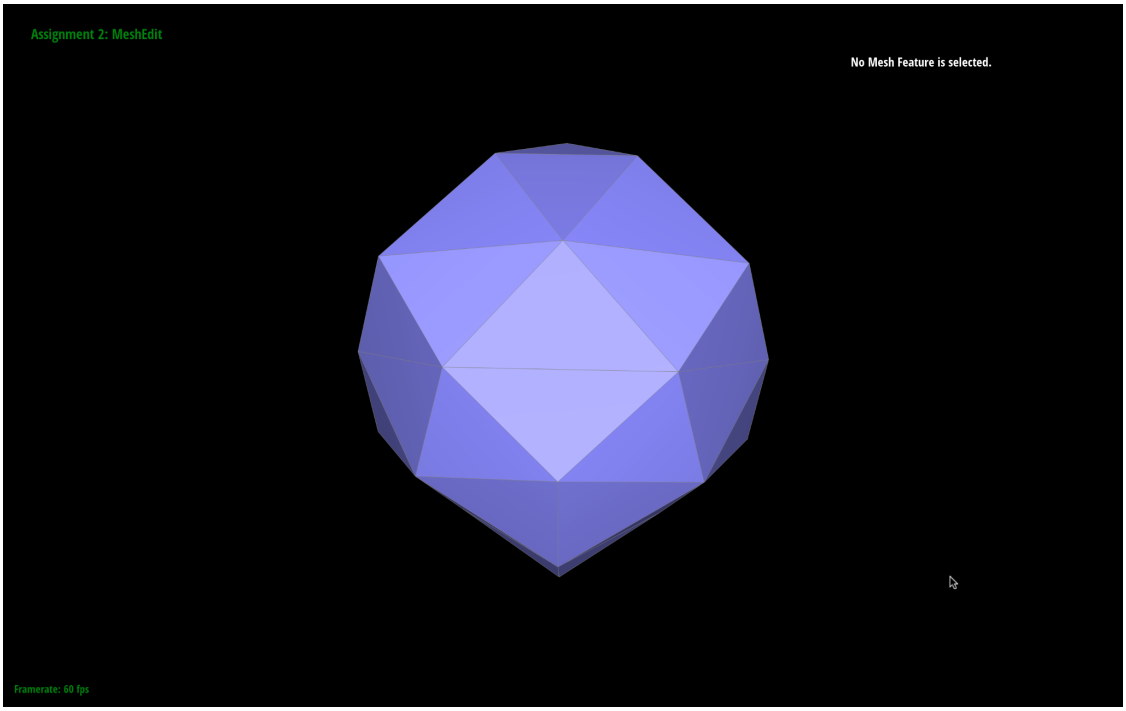
- Briefly explain how you implemented the loop subdivision and describe any interesting implementation / debugging tricks you have used.

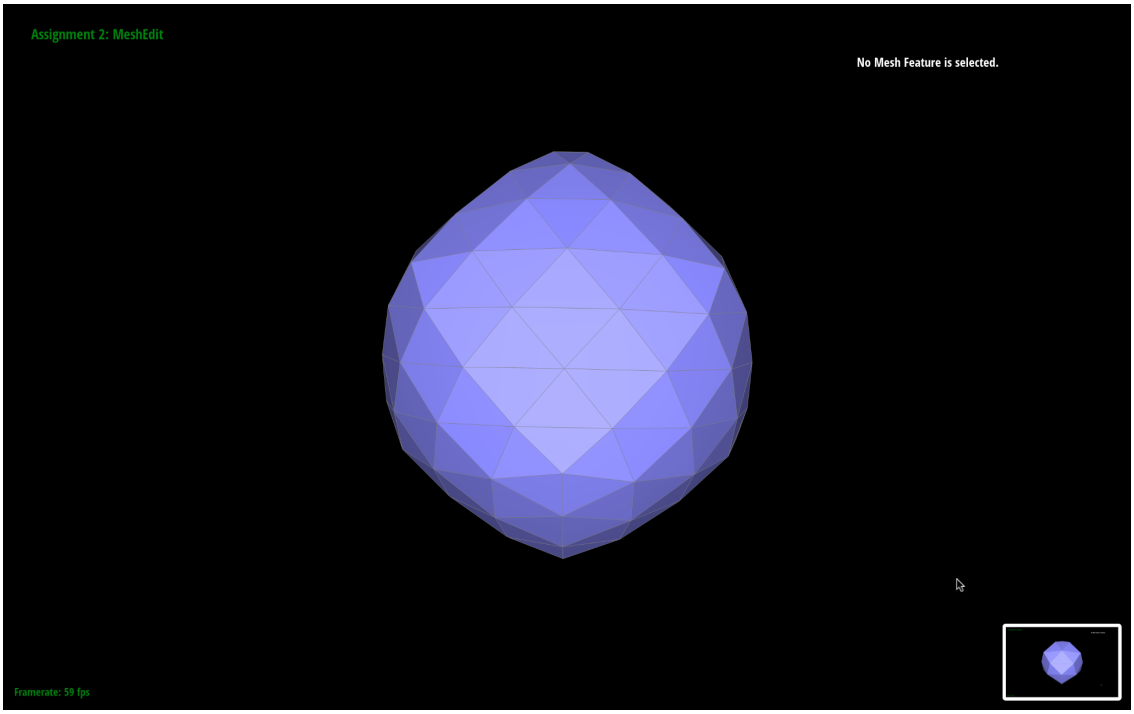
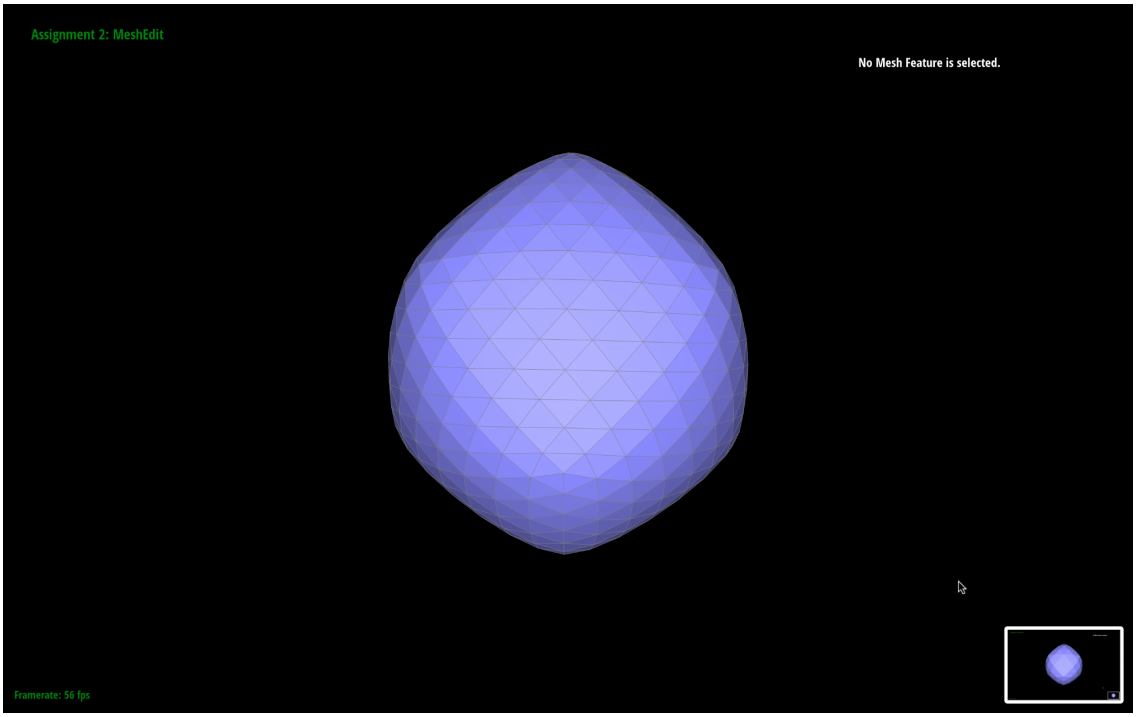
    - To implement loop subdivision, we divided the task into 5 parts. In part 1, we computed computed new positions for all the vertices through using the equation (1 - n * u) * original_position + u * original_neighbor_position_sum. In part 2, we computed the updated vertex positions associated with edges by iterating through all old edges and computed the position that would be created after splitting the edges (midpoint). We used the formula (3.0/8)*(A+B)+(1.0/8)*(C+D) to compute the new position. In part 3, we iterate through all of the edges and split them using part 5 while updating the isNew variable for the edges and vertex. In part 4, we look at all of the new edges that were created through the splits, find those that connect between an old vertex and and new vertex, and flip those edges. Lastly, we set all the vertex's position to the new position. One debugging implementation we had to go through was to make sure to only iterate through the edges within the first mesh and not include those that were created from the splits in order to prevent it from infinite looping.
- 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?
    - The cube becomes more rounded and moves towards a spherical shape as you continue to subdivide it. The sharp corners and edges gets smoothed out as you continue to loop subdivide, but the corners with the edge going towards it will sort of bulge out as the subdivision takes place. Yes, You can reduce this effect by presplitting some edges. An example is talked about in the next portion.

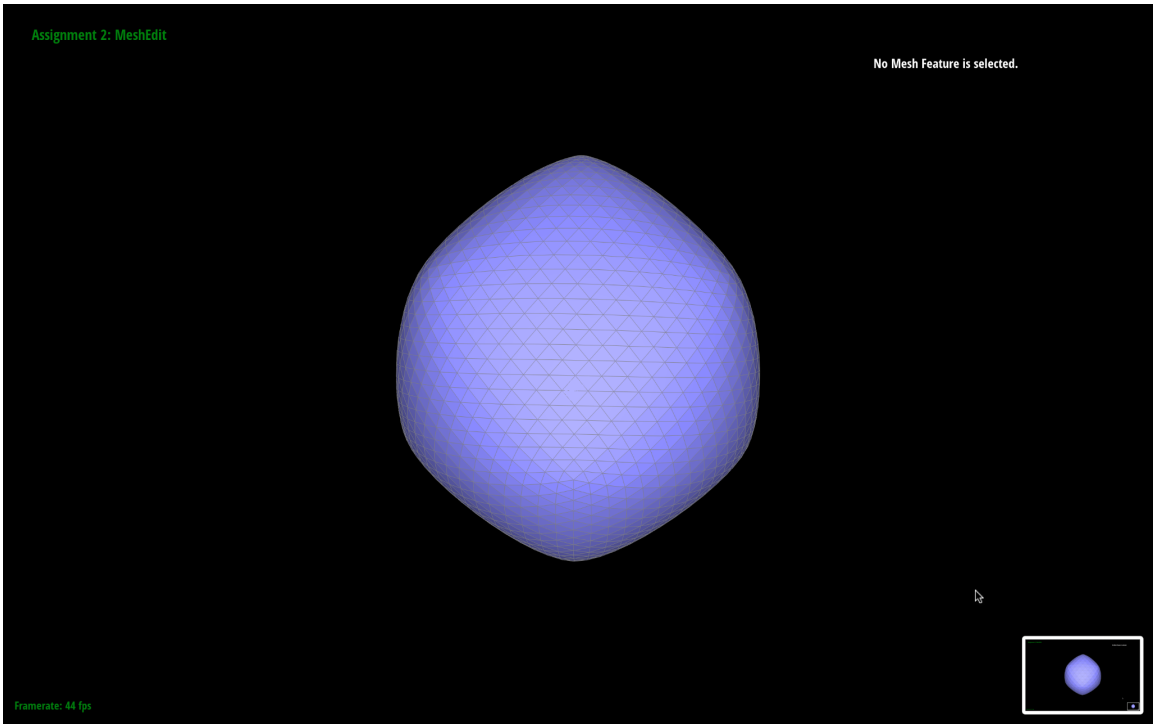
○

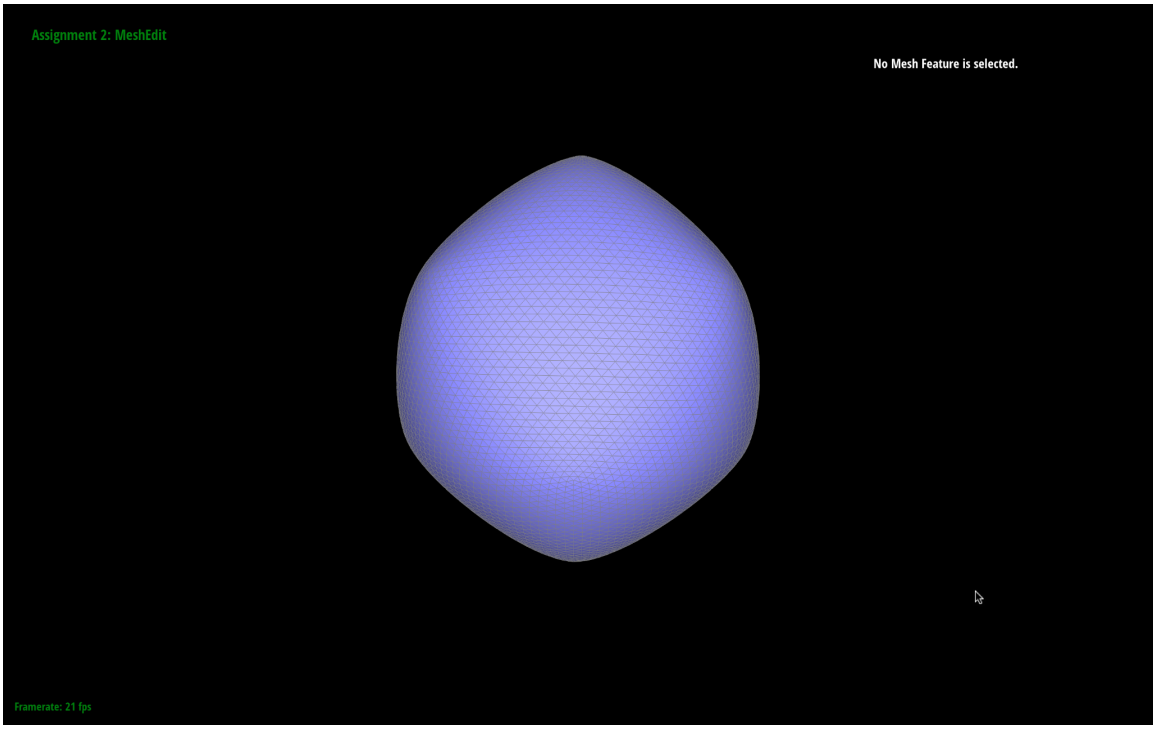

Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 60 fps

○

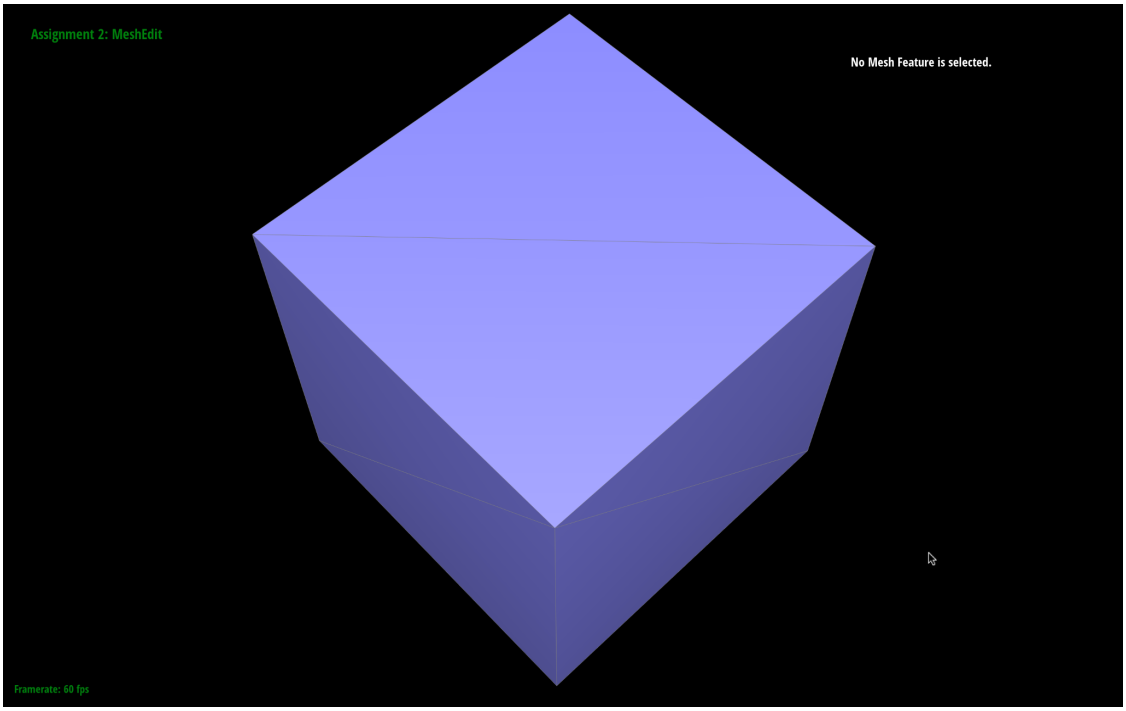

Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 60 fps

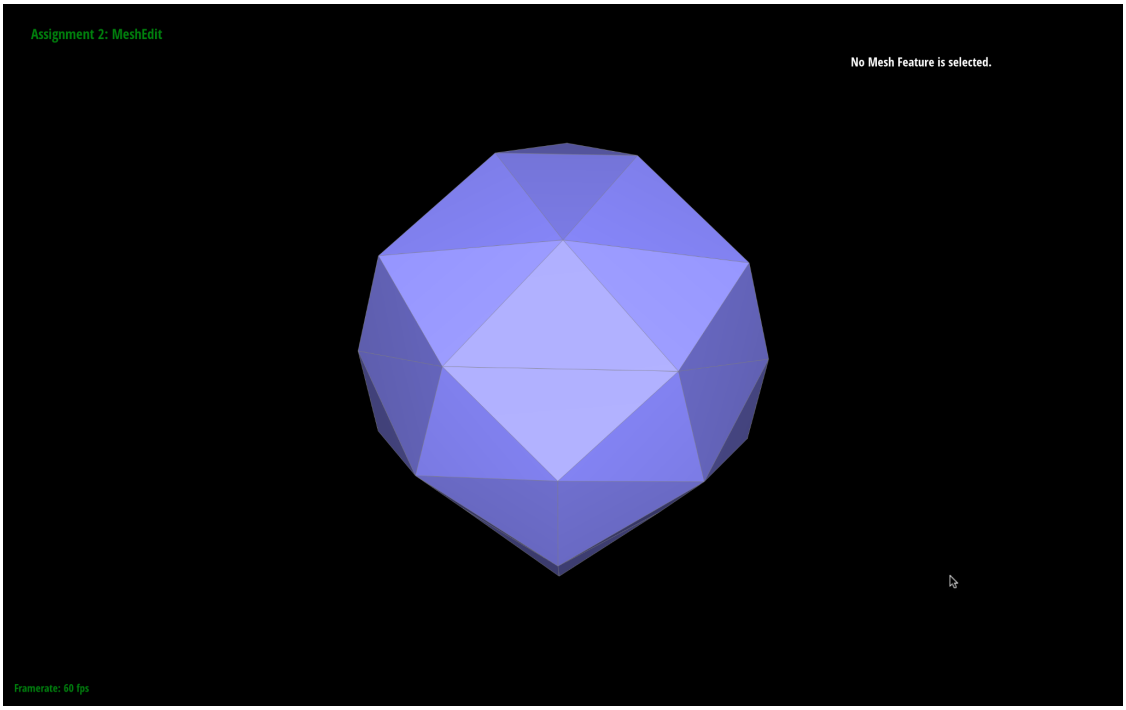
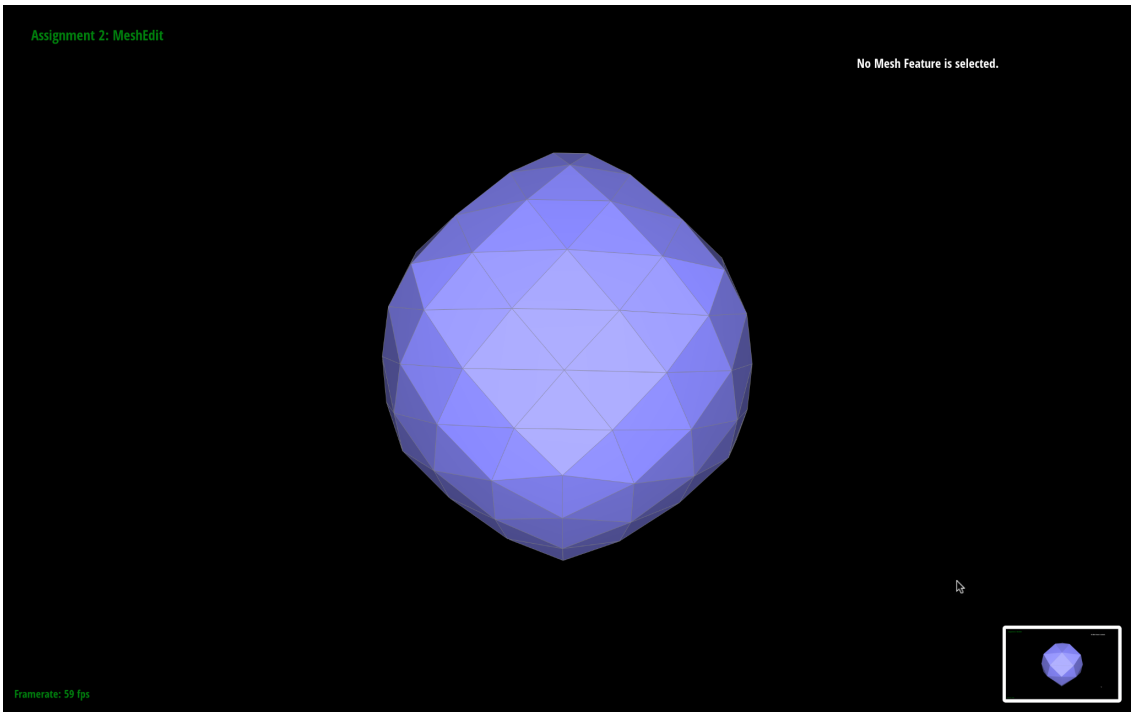
- 
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 59 fps
- 
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 56 fps

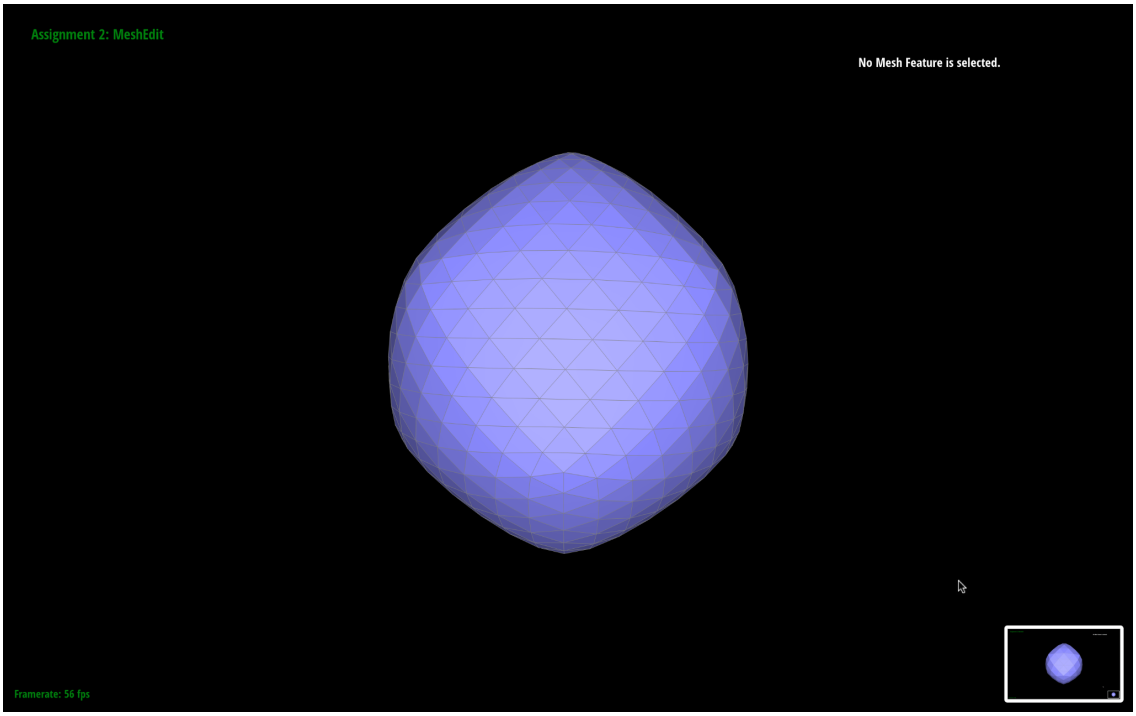
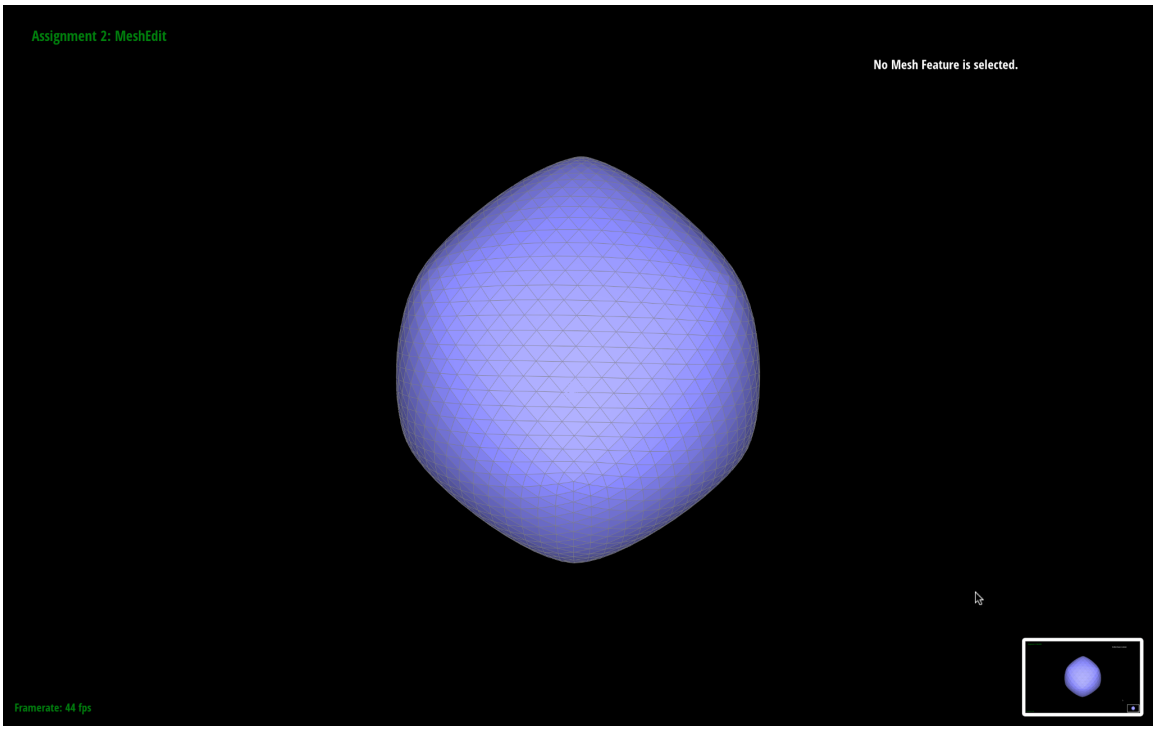
- 

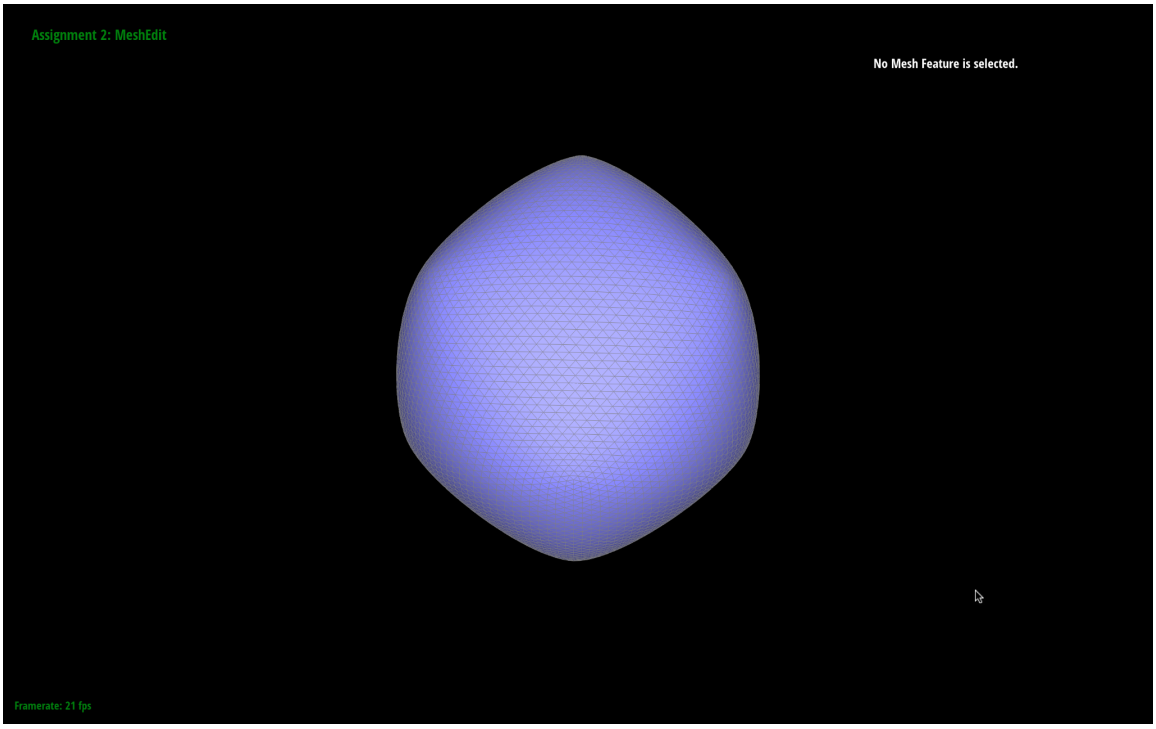
- 

- 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.

- Yes, one thing that we ca do is to split each edge on the 6 faces so that there is an X across all of them. By preprocessing it this way, as you subdivide, the cube shape gets retained, just a more rounded cube because it started as a symmetrical cube. Without doing this, this effect of being slightly asymmetrical happens since the subdivisions would be larger on one side than the other, and this effect would continue forwards as you subdivide. So after the pre-processing, the input of the subdivision is symmetrical, so the triangles that it subdivides into will be of the same size on the corners, and thus symmetrical.
- 

- 
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 60 fps
- 
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 59 fps

- 
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 56 fps
- 
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 44 fps

- 
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 21 fps