CS 184 HW Write Ups

link to webpage: https://cal-cs184-student.github.io/hw-webpages-sp24-EmmanuelCobian/hw2/index.html

Homework 2 - Mesh Edit

In this homework assignment, I explored topics on geometric modeling like building Bézier curves and surfaces using de Castel jau algorithm. I also learned to manipulate triangle meshes represented by the half-edge data structure. These manipulations include: `flipEdge`, `splitEdge`, and with the combination of these two elementary operations, `upsample` via loop subdivision. In particular, I found this final task of implementing `upsample` interesting because it was a nice way of building up from the basics up to something much more complex and applicable.

Section 1: Bezier Curves and Surfaces

Part 1: Bezier Curves with 1D de Casteljau Subdivision

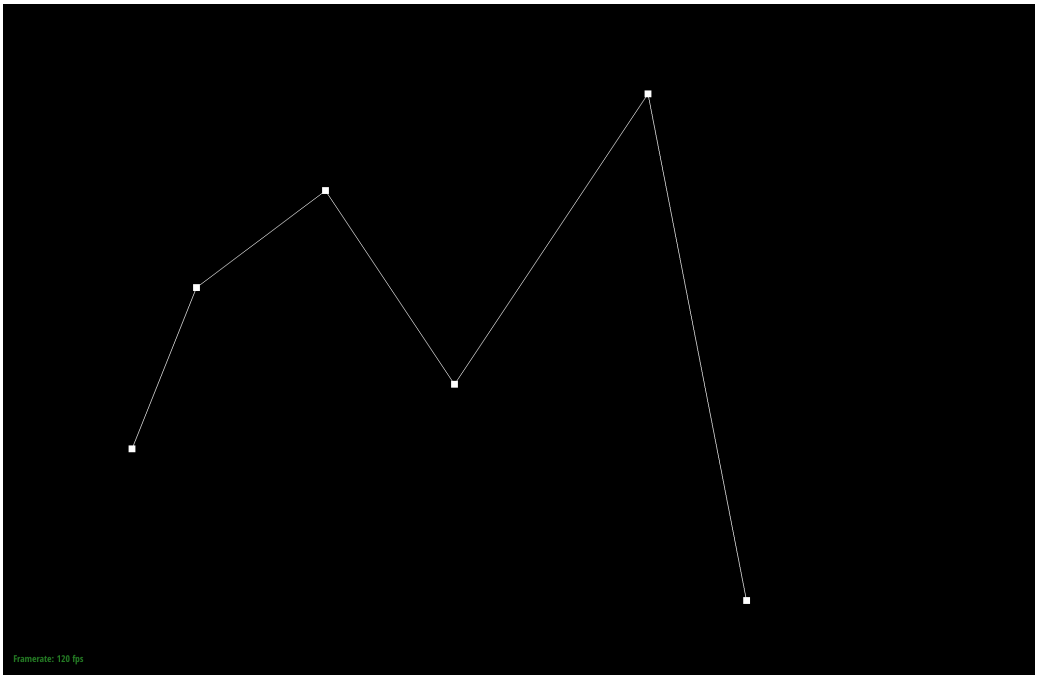
What is de Casteljau's algorithm?

De Casteljau's algorithm is a way to evaluate Bézier curves and surfaces. Through a recursive process, this algorithm splits a point between two weights and draws new line segments until there's none

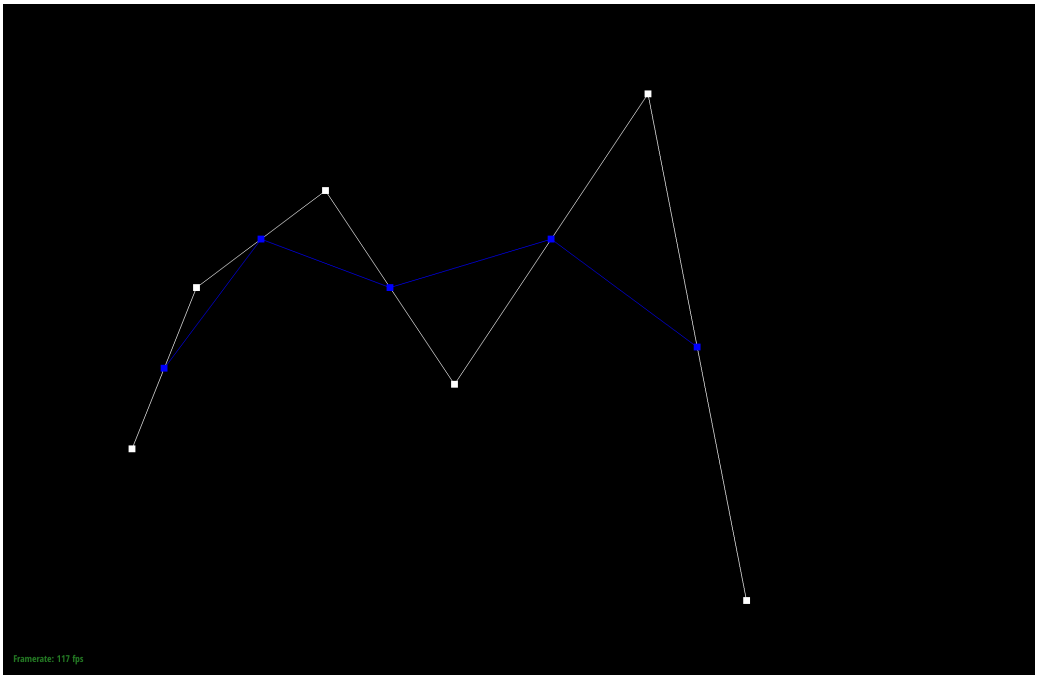
left to draw. This is the base case where the final point representes a point on the curve.

How was the de Casteljau algorithm implemented?

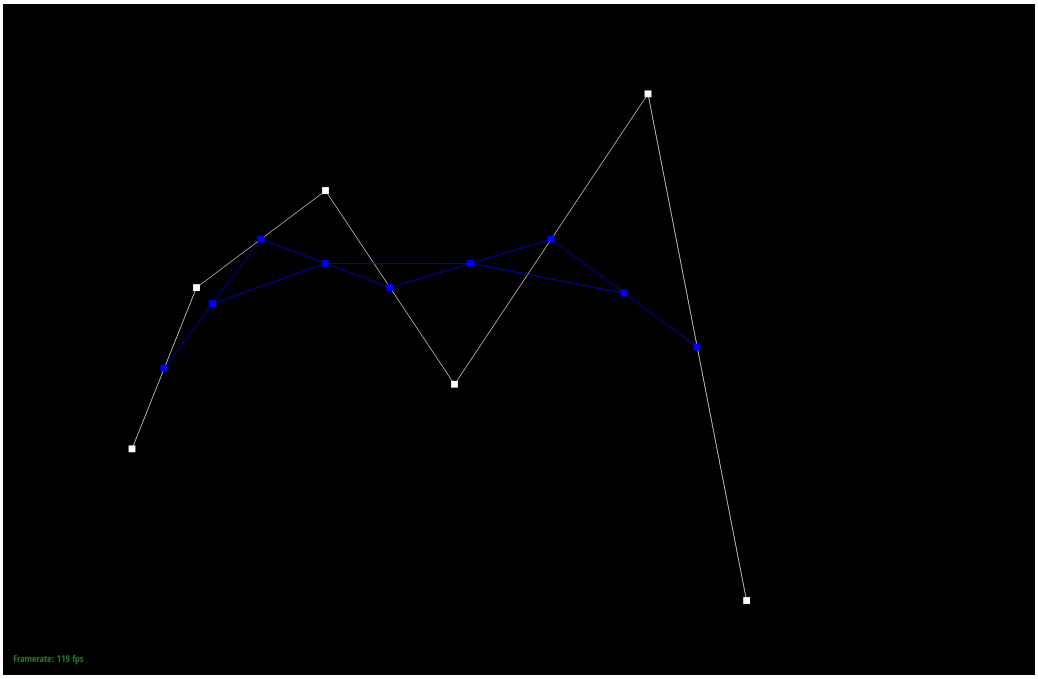
By interpolating between adjacent control points, we recursively divide the curve into smaller parts until we're left with a single point. This point represents a single point on the curve.



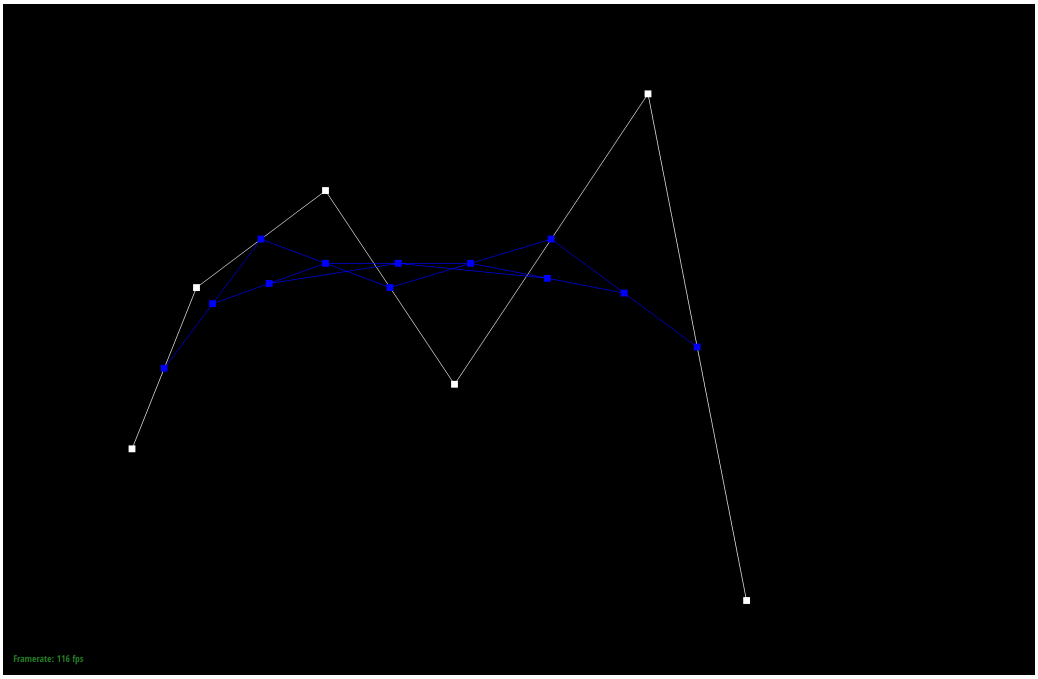
Original Points



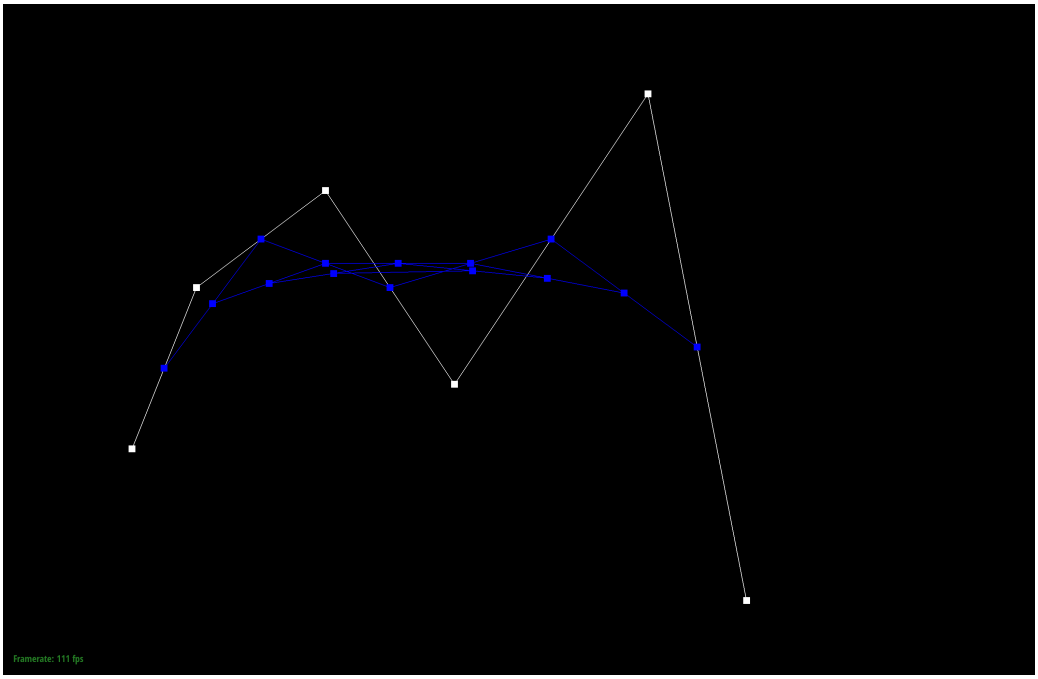
Step 1



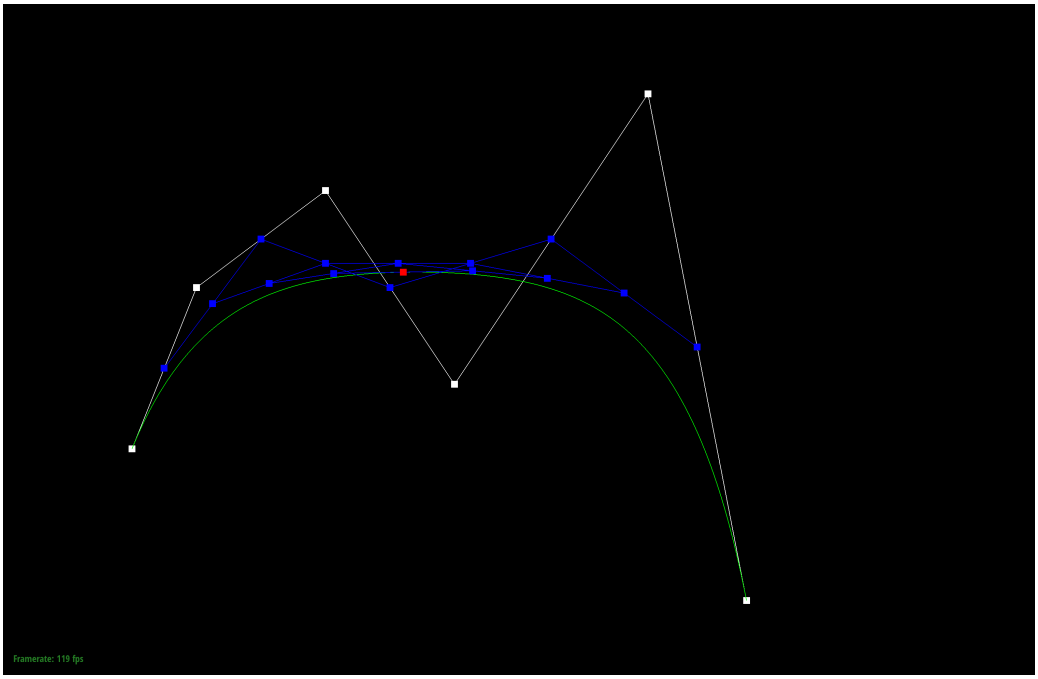
Step 2



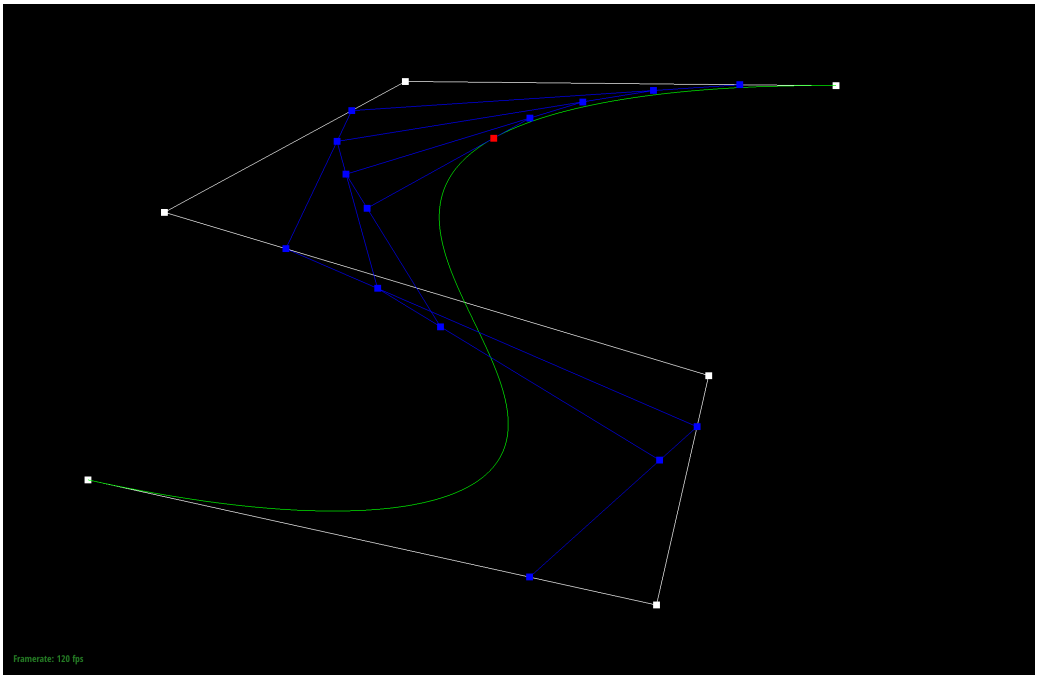
Step 3



Step 4



Step 5



Different Anchors and t-value

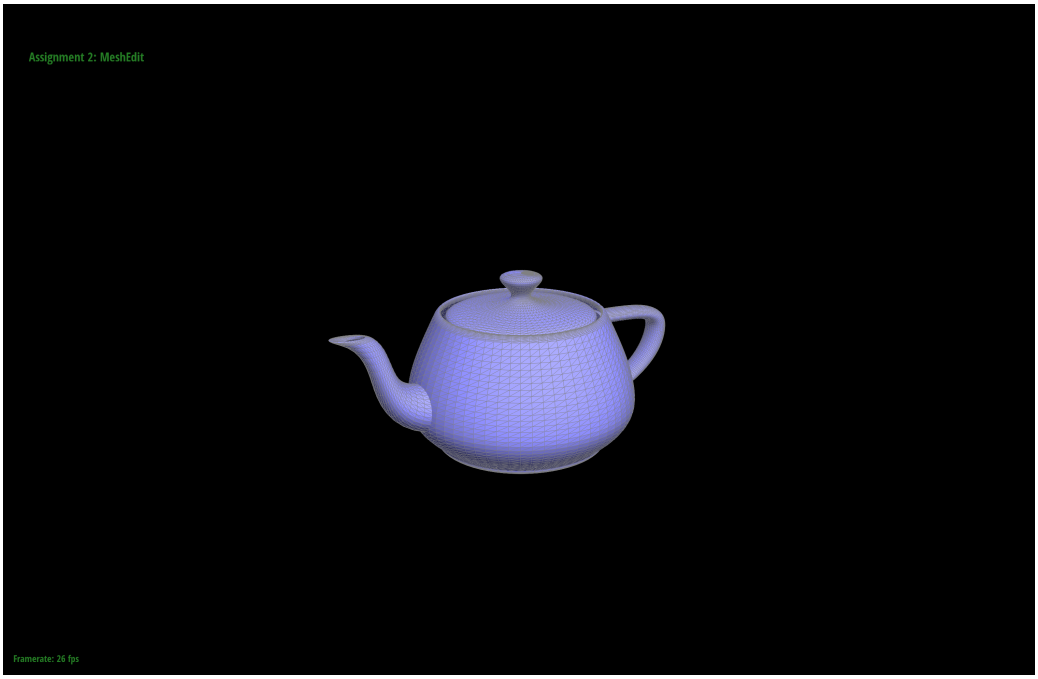
Part 2: Bezir Surfaces with Separable 1D de Casteljau

How does de Casteljau algorithm extend to Bezier surfaces?

De Casteljau's algorithm can be extended to Bezier surfaces by applying the same recursive subdivision to first the rows of the surface and then taking each of those evaluated points and creating another point from those.

How was this extension implemented to evaluate Bezier surfaces?

By expanding on the implementation for Bezier curves, I first implemented the logic for evaluating one step of the algorithm, then implementing the logic to get the point in one dimension. These abstractions help make the next part easier by just evaluating the 1-D point for u first, then running the result of that again, but now for the t value.



Section 2: Triangle Meshes and Half-Edge Data Structure

Part 3: Area-Weighted Vertex Normals

How were area-weighted vertex normals implemented?

By iterating through the adjacent faces of a vertex and taking their area, I accumulated these values and then took the `unit()` of them to get the normalized version of the total area. When iterating through faces, I have a check for whether a face is a boundary because those shouldn't be taken into consideration for the shading.

Teapot shading without vertex normals

Teapot shading with vertex normals

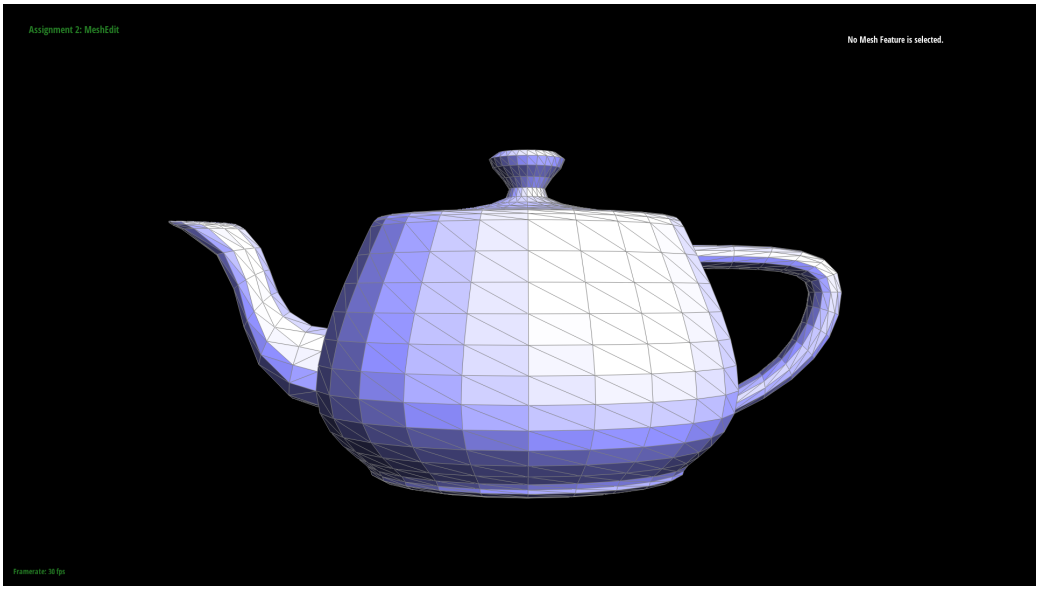
Part 4: Edge Flip

How was edge flip implemented?

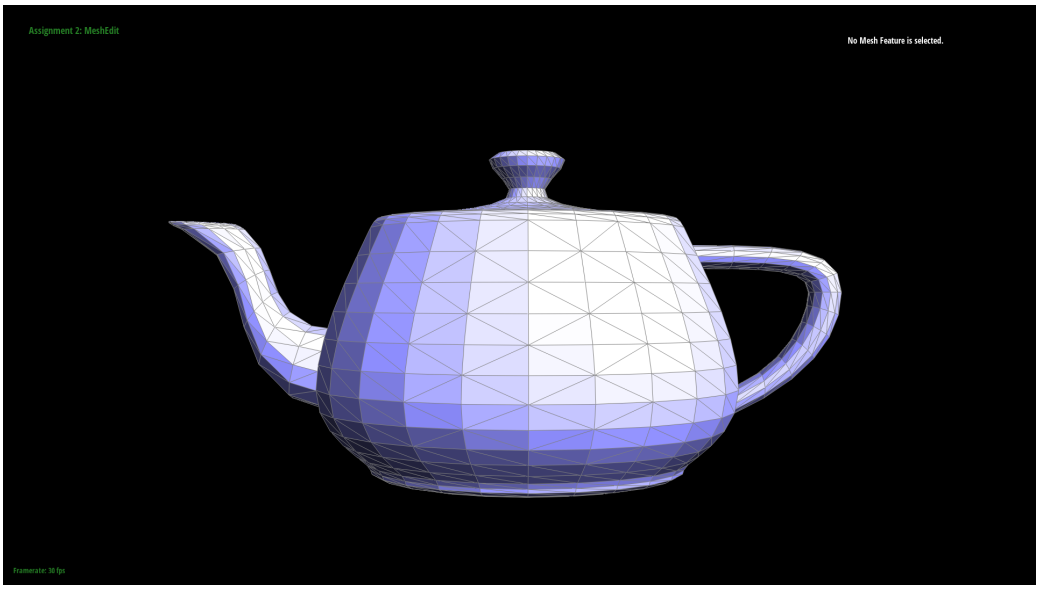
I first took inventory of all the components present in the pair of triangles `(a, b, c)` and `(c, b, d)`. This includes, halfedges, faces, verticies, and edges. Afterwards, I just update all the pointers of these components according to how they change after the flip. Even if some pointers didn't change, I still assigned those just in case I missed any.

Any interesting tricks for implementation/debugging?

I found that using a piece of paper and drawing out the components of the two triangles `(a, b, c)` and `(c, b, d)` was really helpful in keeping track where everything was. After drawing out the before and after, all that was left was to read out the locations/positions of the new components.



Before Edge Flips



After Edge Flips

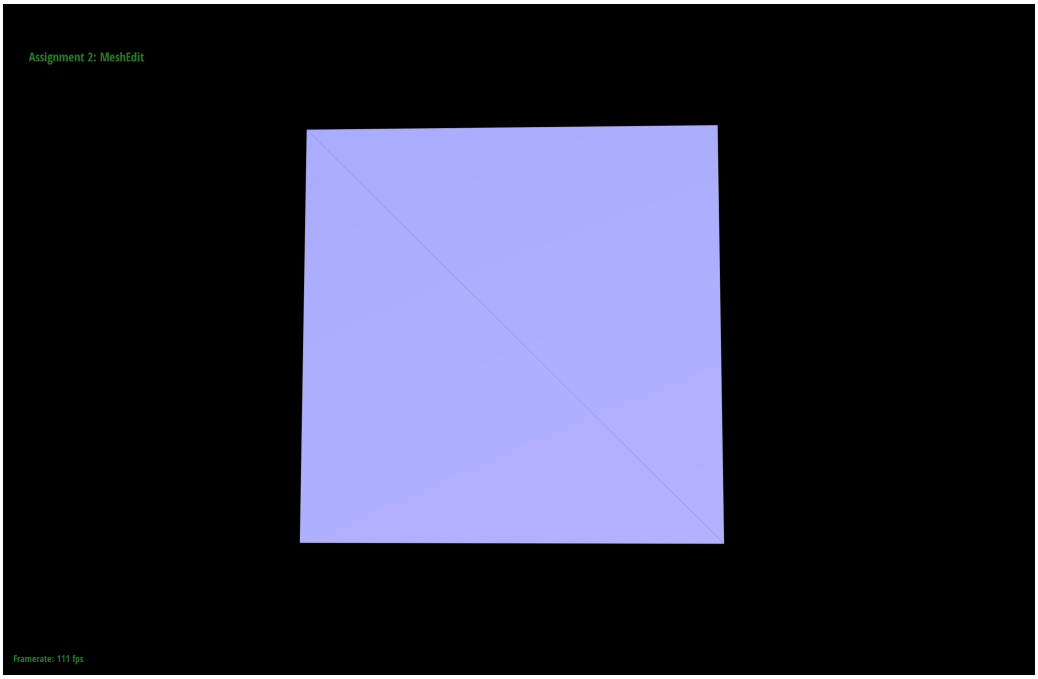
Part 5: Edge Split

How was edge split implemented? Any Interesting implementation tricks?

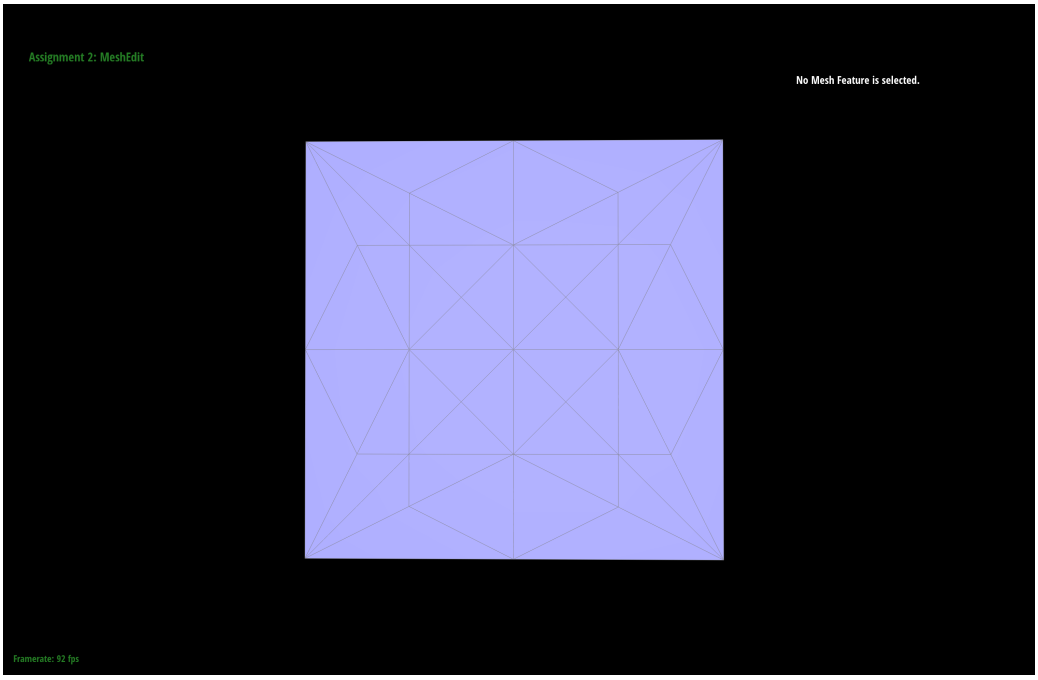
In a similar process to the previous part, I first drew out the before and after of what 2 neighboring triangles would look like during the edge split process. After that, the solution pretty much reads itself out. First by collecting all the components and then updating them as needed.

Any eventful debugging journeys?

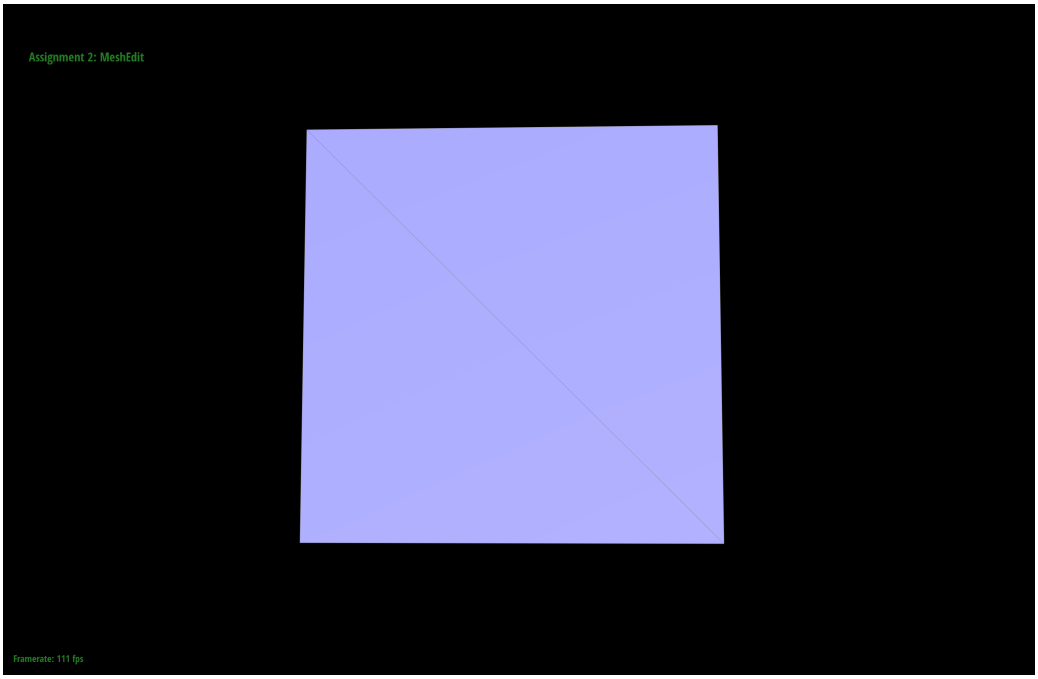
Although I thought this was going to be a quick implementation because of how easy it was to read out the updated components from the drawn diagram I had, I ended up debugging for a while because of an input error. One of the new halfedges was pointing to the wrong edge, leading to a bug when splitting the same edge multiple times. The solution was just to run through the diagram again and fix any misinputs.



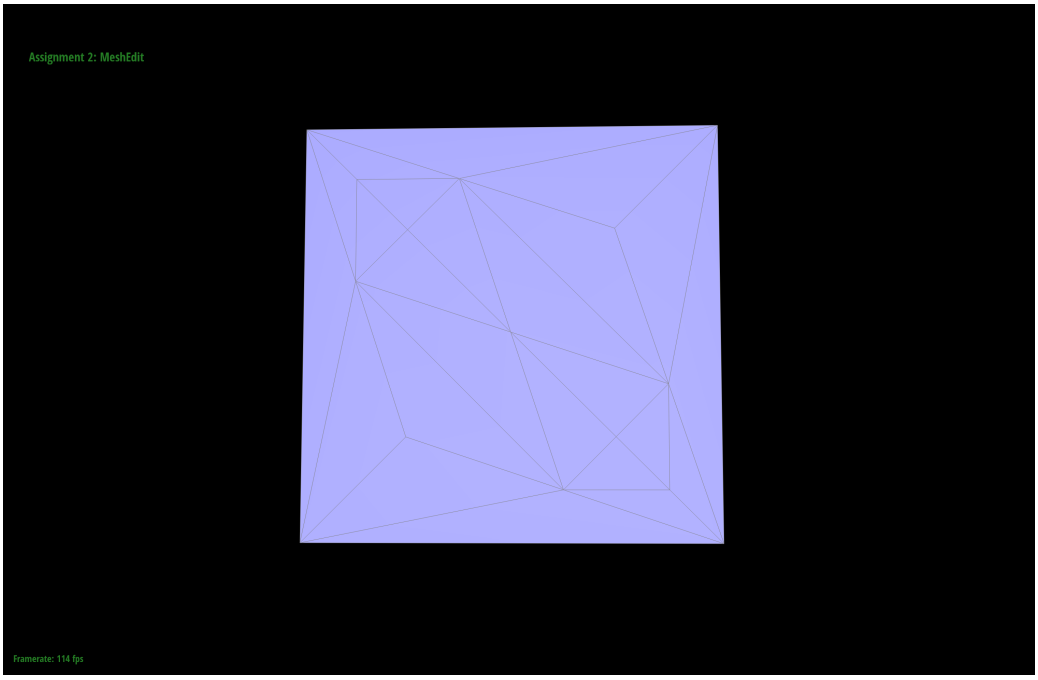
Before edge splits



After edge splits



Before edge splits



After combination of edge splits and flips

Part 6: Loop Subdivision for Mesh Upsampling

How was loop subdivision implemented?

First, I computed the positions of both new and old vertices using the original mesh. These computations came from the lecture slides where we saw the weights to use for placing verticies. Then, I subdivided every edge in the original mesh. Afterwards, I flipped all the new edges that connected an old and new vertex. Finally, I updated all vertex positions in the subdivided mesh.

Any interesting implementation tricks you used?

Similar to previous parts of this assignment, I found it really useful to look back at the drawn out diagrams for what a mesh should look like. Specifically, I found my diagram helpful to know which new edges to assign the `isNew` property because this was something I struggled with figuring out until I referenced back to the diagram.

Observations of how meshes behave after loop subdivision. What happens to sharp corners and edges?

From the screenshots below, I notice that sharp corners and edges smooth out, becoming more like curves. In general, I notice this is the case for all meshes I've looked at. If a mesh is already curved, then loop subdivision maintains the relative curveness of the mesh.

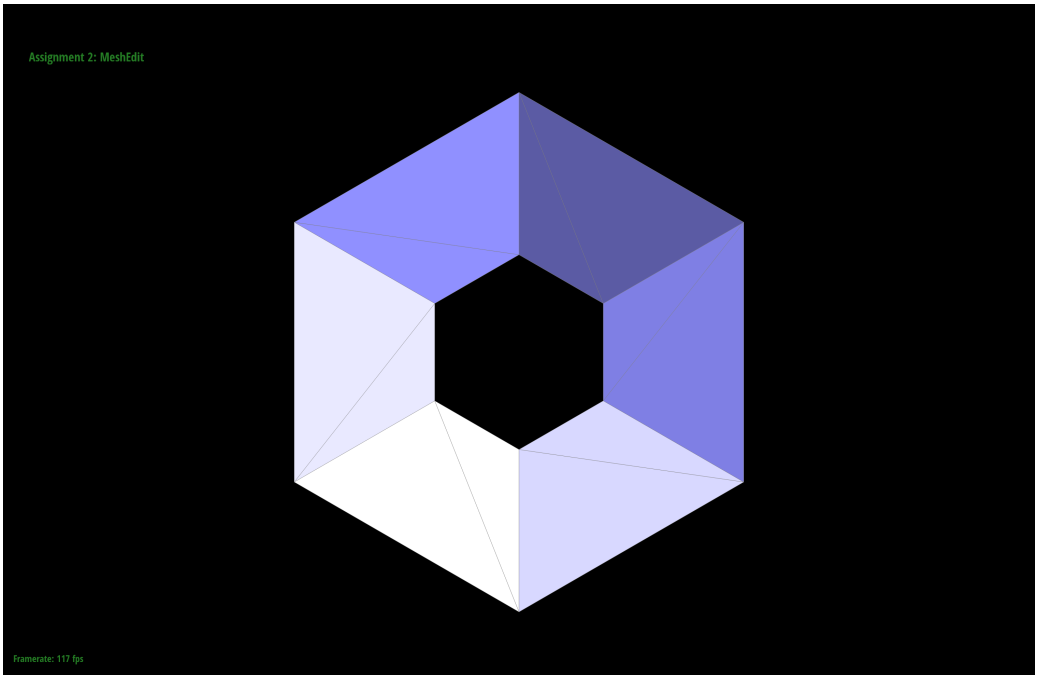


Figure before loop subdivision. Sharp edges and corners are present

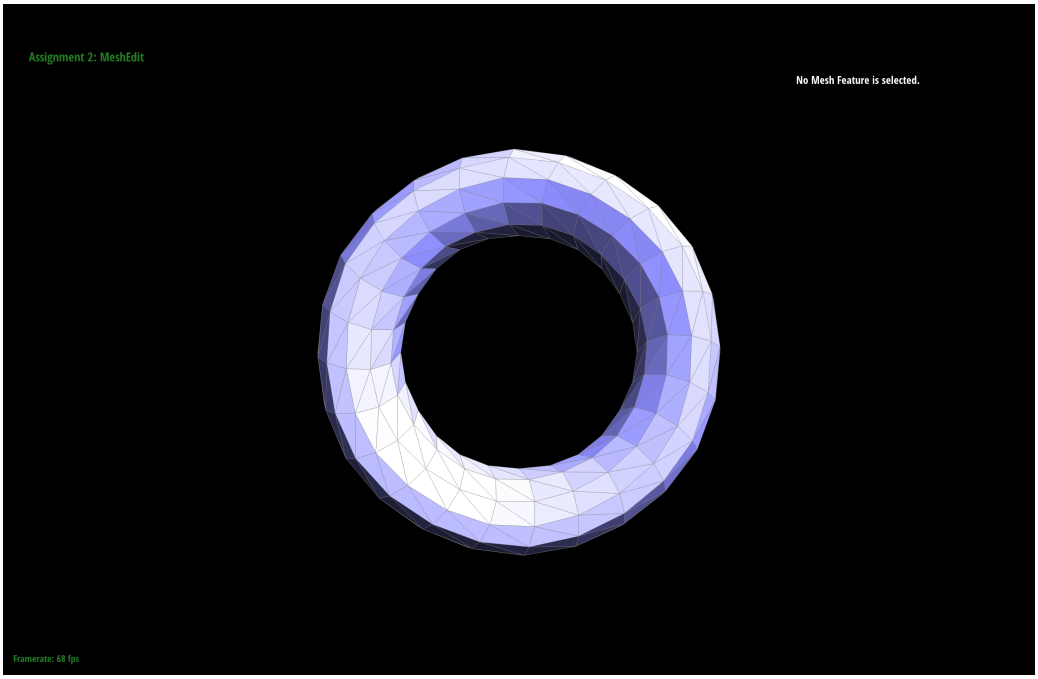


Figure after two iterations of loop subdivision. No sharp edges or corners are present

Can you reduce this effect by pre-splitting some edges?

I was able to reduce this effect by pre-splitting the sharp edges many times. I wouldn't say that there was a specific pattern or way that splitting these edges would better or worsen the mitigation of this effect. Below are screenshots showing the preprocessing splits before and after two iterations of loop sampling. The pre-processing was only done to the closest face, facing the camera.

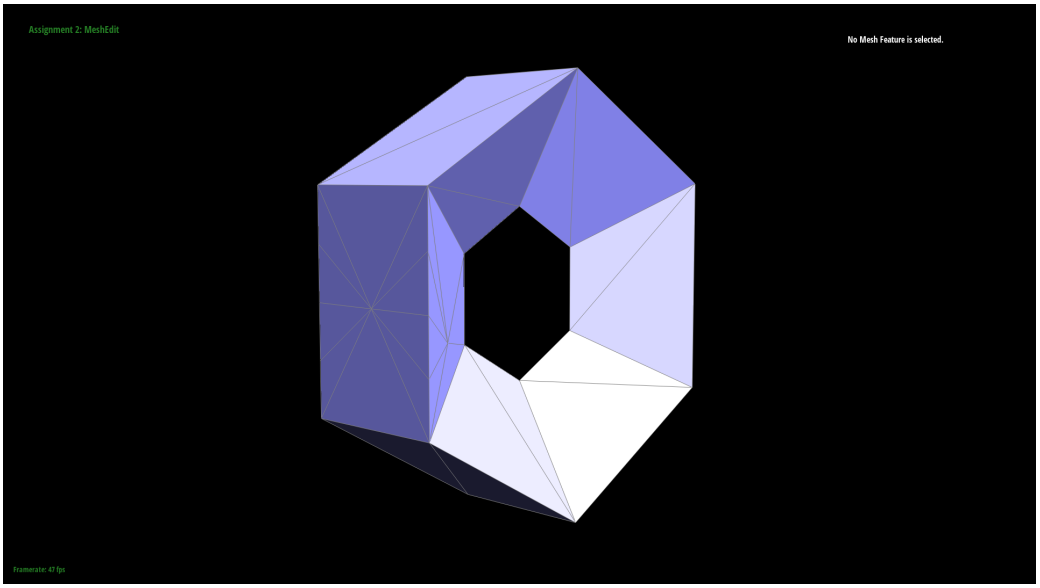


Figure preprocessed before loop subdivision

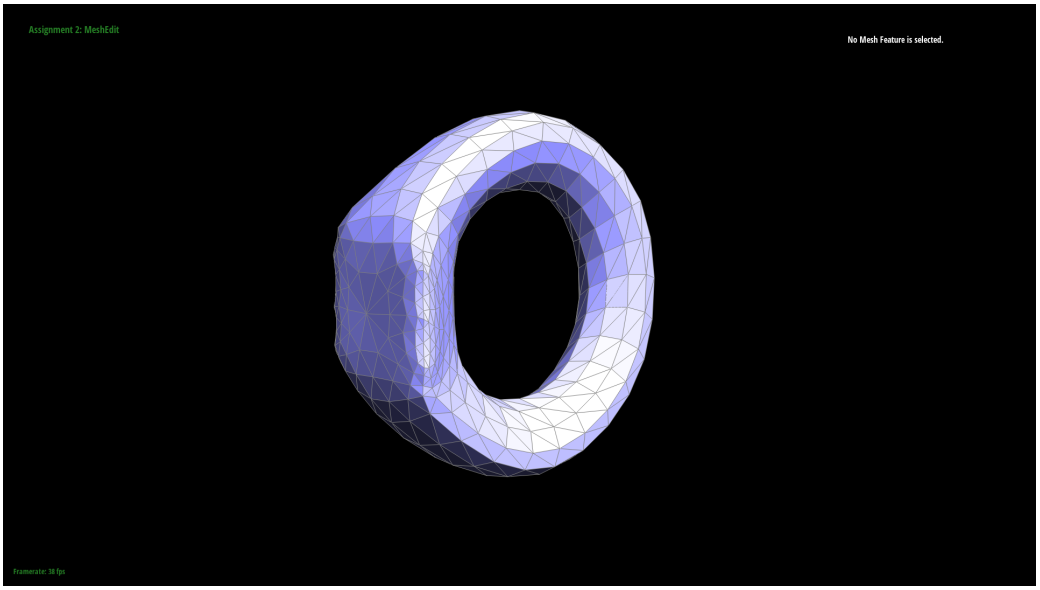
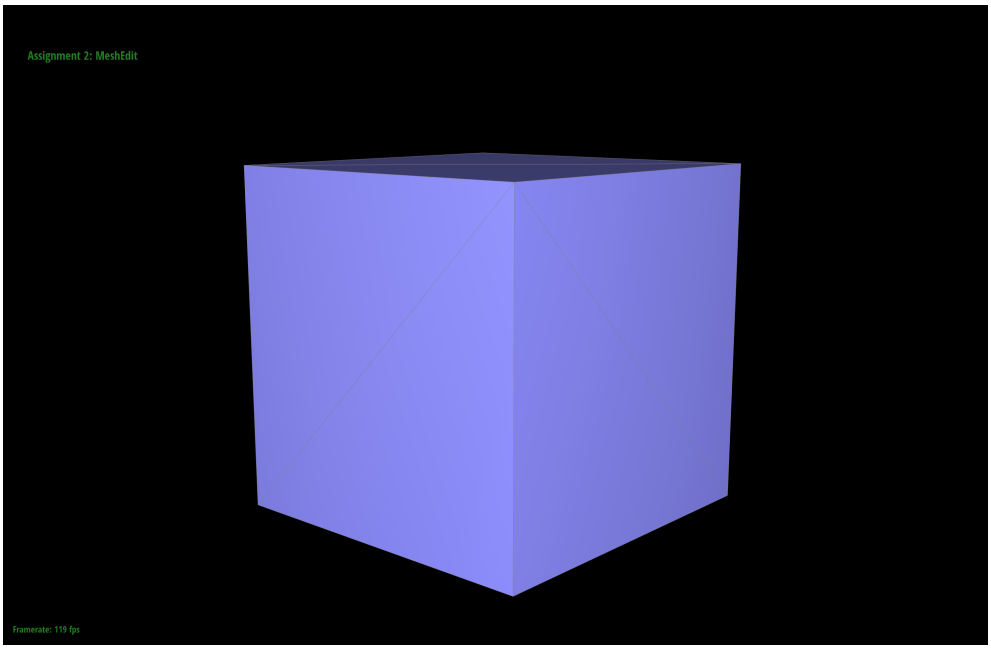


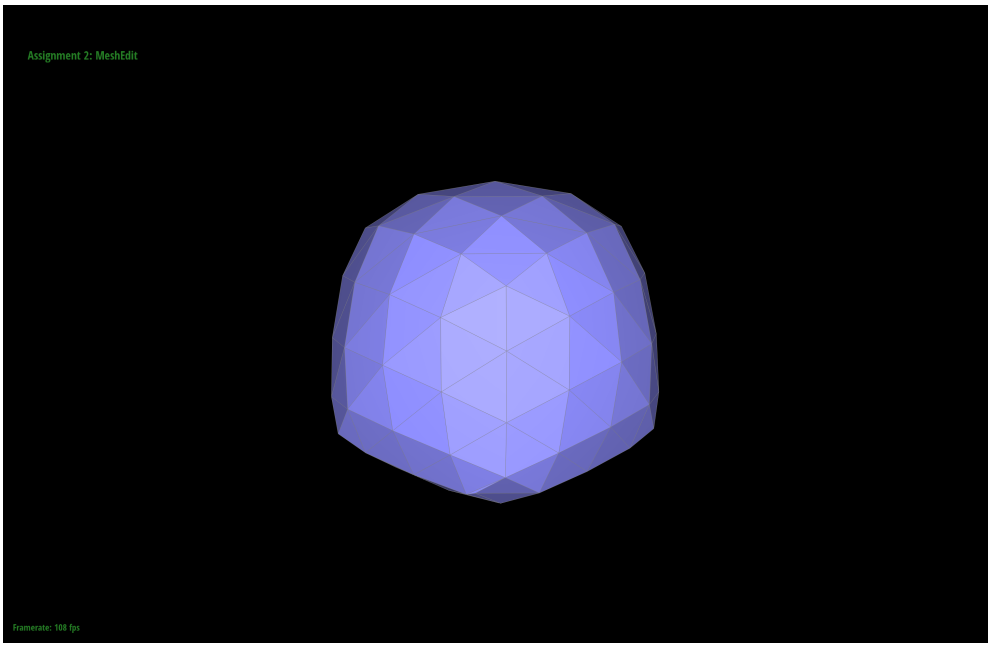
Figure preprocessed after 2 iterations of loop subdivision. Edges still remain relatively sharp.

Perform several iterations of loop subdivision on the cube. Document how the cube becomes asymmetrical after several iterations.

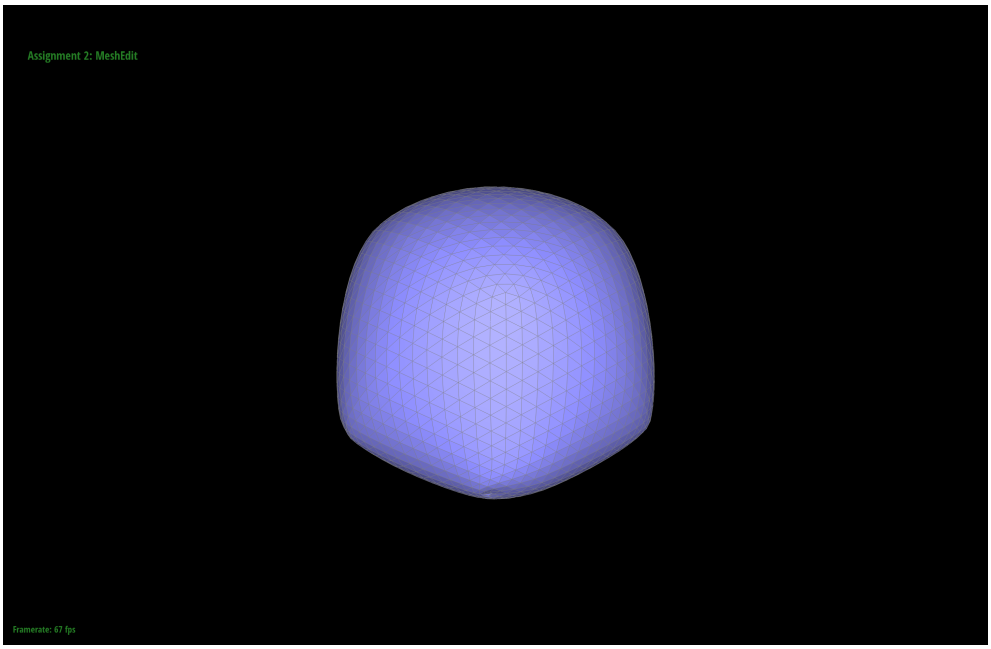
By the screenshots below, I notice that the cube does in fact become asymmetrical after only two iterations of the loop subdivision algorithm. When increased by two more iterations. this effect only becomes more apparent.



cube before loop subdivision.



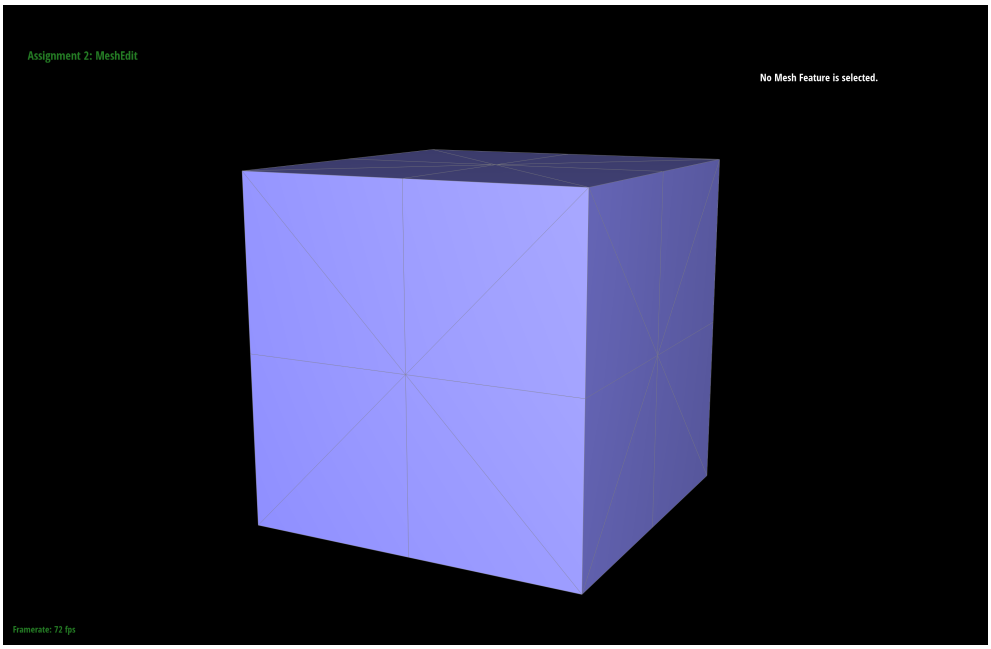
cube after two iterations of loop subdivision.



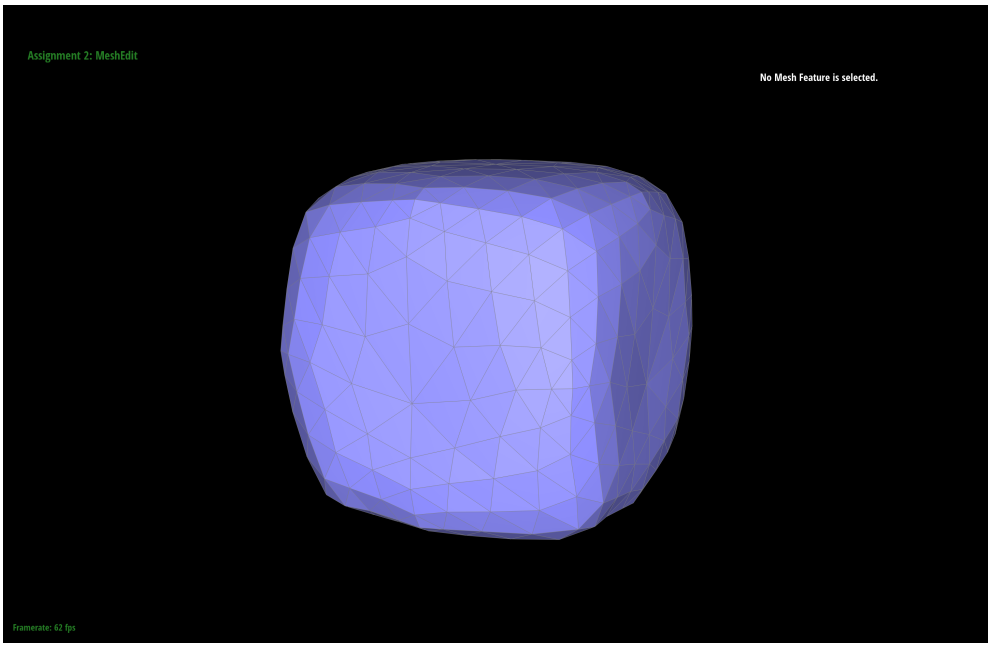
cube after four iterations of loop subdivision.

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.

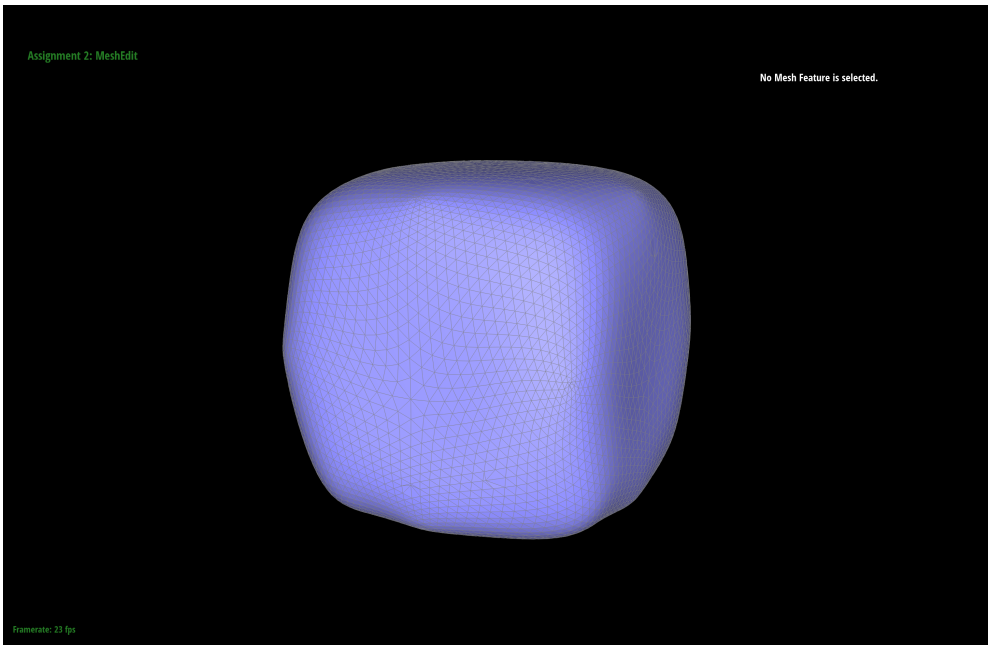
By splitting the edges in an order such that distances between verticies become more equal, I was able to make it so that the cube subdivides more equally. In the screenshots below, I notice how much more symmetrical the cube is after just two iterations when compared to the case with no pre-processing. Intuitively, I believe this happens because by subdividing the cube into smaller, equal, parts before performing loop-subdivision, we dictate more of the resulting shape we desire after the algorithm finishes.



pre-processed cube before loop subdivision



pre-processed cube after two iterations of loop subdivision



pre-processed cube after four iterations of loop subdivision

Emmanuel Cobian Duarte - CS 184 - Spring 2024