# CS 184: Computer Graphics and Imaging

## Project 2: Meshedit

**Michael Lin, Rachel Lee**

## Overview

In this project, we implemented de Casteljau's algorithm for evaluating Bezier curves and experimented with different shading techniques to help see the differences in making an image more realistic. We also implemented local mesh operations such as mesh flipping and splitting to increase the granularity of the meshes with more defined edges. Finally, we implemented loop subdivision for mesh upsampling by subdividing the triangles into smaller sub-triangles for more accurate shading and dimension. It was interesting to see how the Bezier curves were calculated in an application setting where the estimated point can be found from an arbitrary set of control points. Being able to flip and split the edges also provided more insight into how these techniques enable the object to look more realistic with increasingly detailed dimension and shading.

## Task 1: Bezier Curves with 1D de Casteljau Subdivision

**Briefly explain de Casteljau's algorithm and how you implemented it in order to evaluate Bezier curves.**

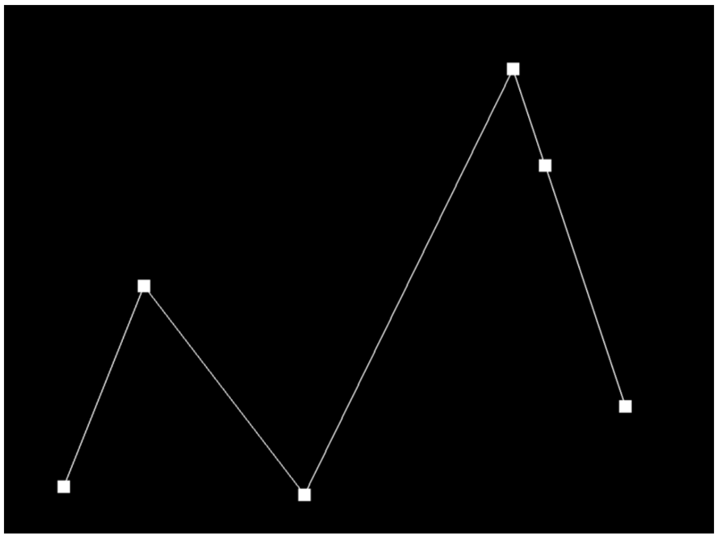
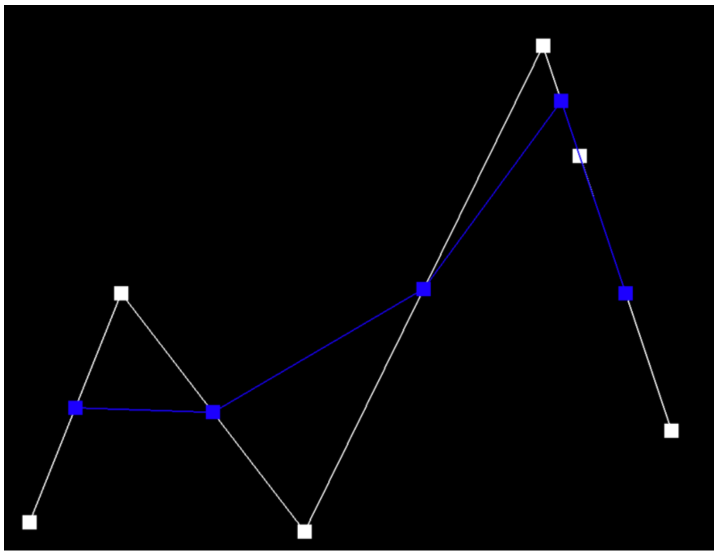
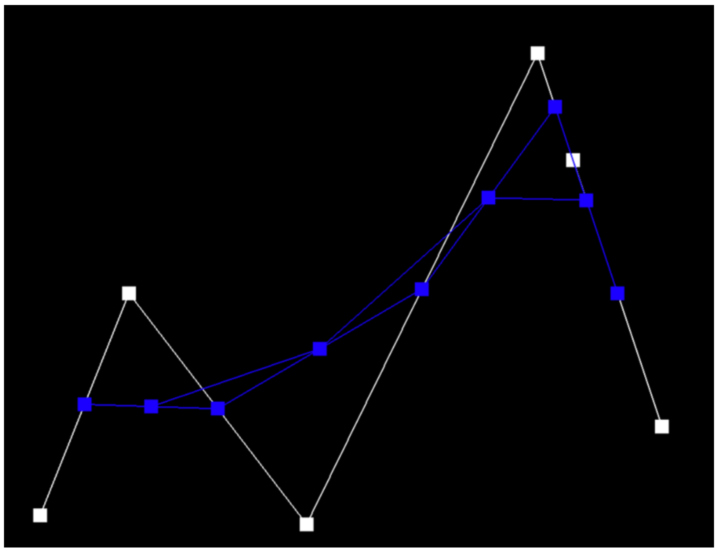
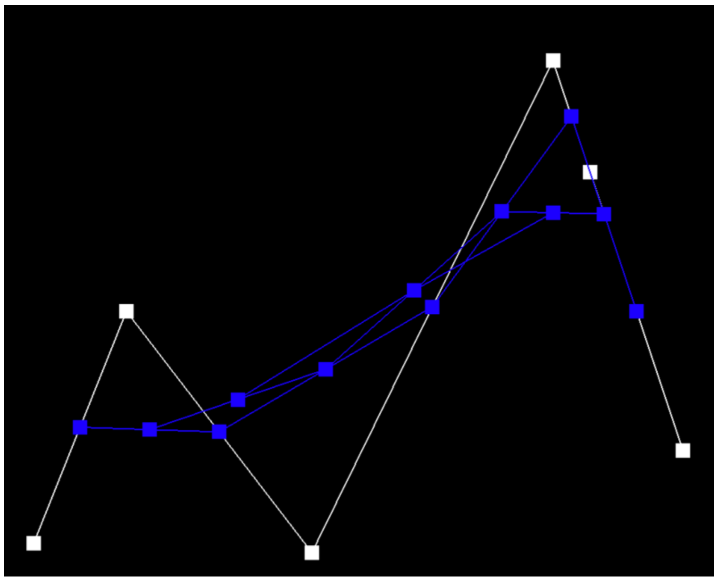
De Casteljau's algorithm is useful for evaluating Bezier curves provided a set of control points. The algorithm essentially uses linear interpolation to create subdivisions of the curve by adding a new point **along** each line segment edge and dividing it into new line segments. We calculate the new point at a given parameter t by using the formula:

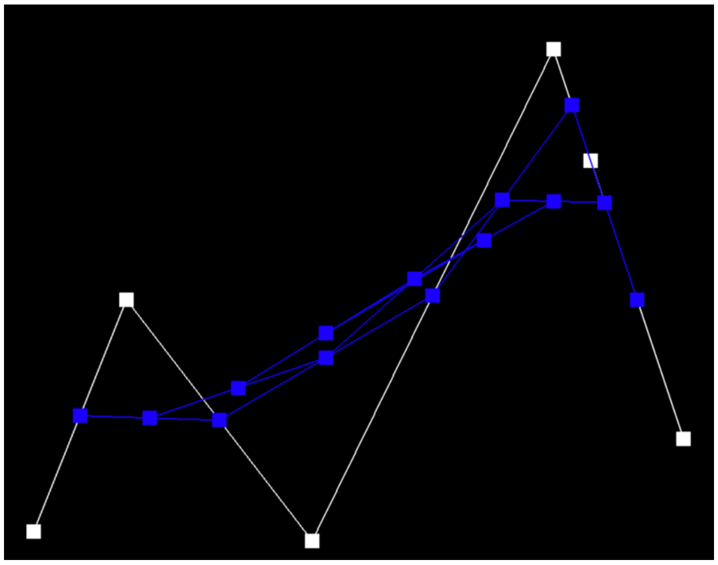
$$p_i' = lerp(p_i, p_{(i+1)}, t) = (1-t) * p_i + t * p_{(i+1)}$$

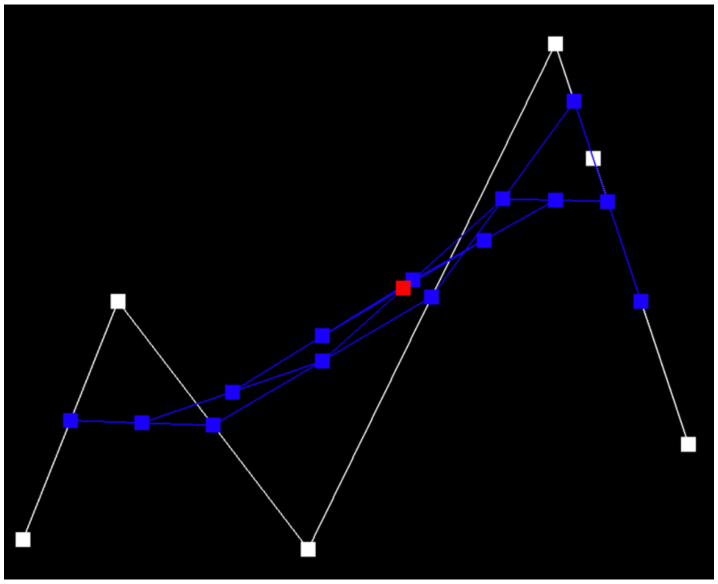
In `evaluateStep` , we iteratively calculate a new point from the set of control (or intermediate) points provided in the input using the above formula `points.size() - 1` times. Then, we add the new point to the 2D vector result and return the updated vector array containing the interpolated points.

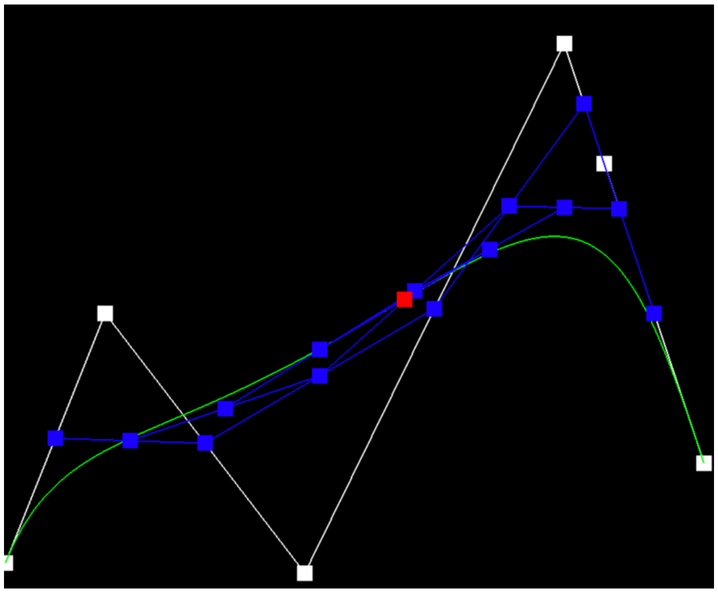
**Take a look at the provided .bzc files and create your own Bezier curve with 6 control points of your choosing. Show screenshots of each step / level of the evaluation from the original control points down to the final evaluated point. Press E to step through. Toggle C to show the completed Bezier curve as well.**

`task1.bzc` :

| | |
|---|---|
|  task1-p1 |  task1-p2 |
|  task1-p3 |  task1-p4 |
| | |

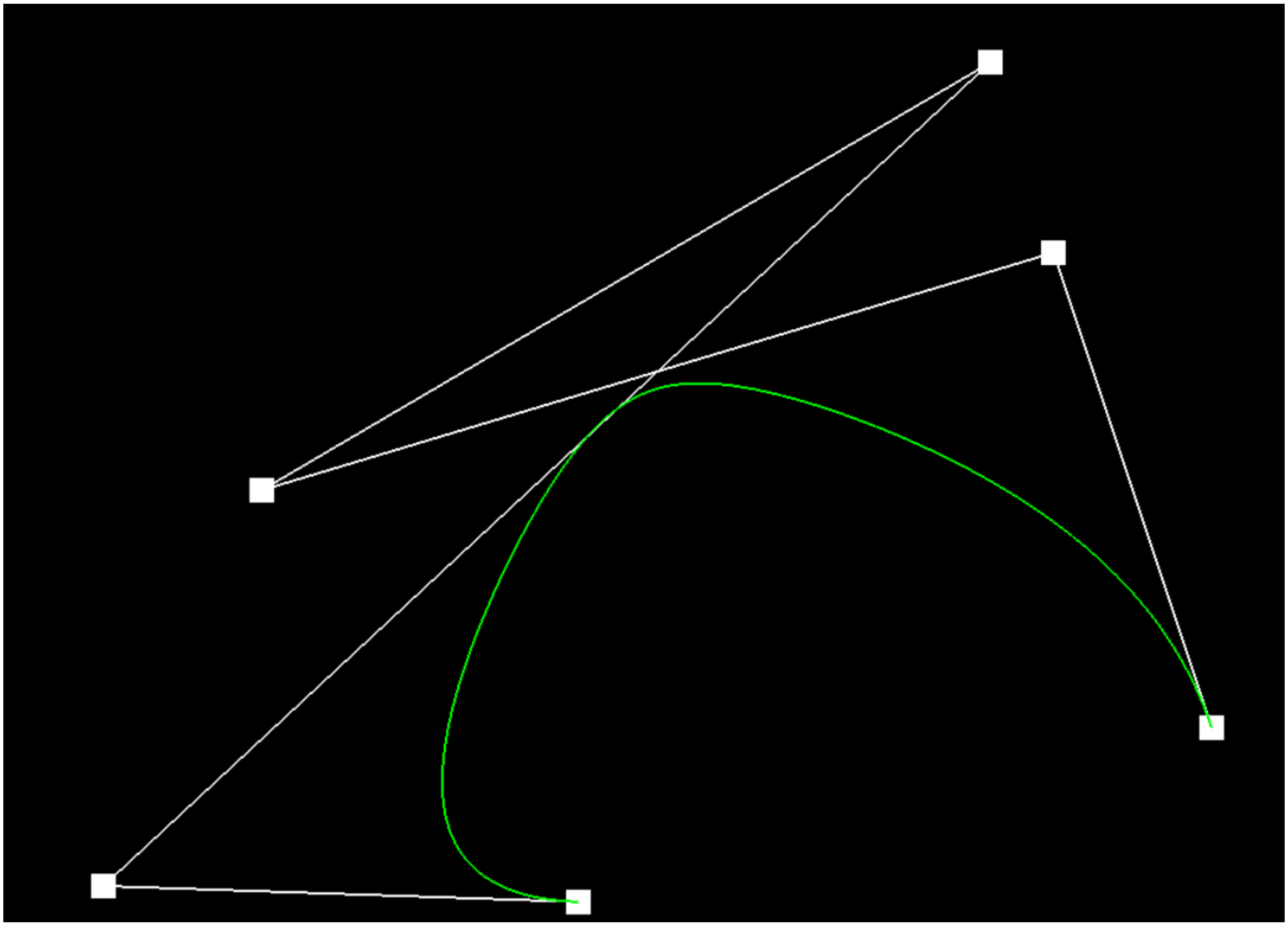

task1-p5


task1-p6


task1-p7

**Show a screenshot of a slightly different Bezier curve by moving the original control points around and modifying the parameter**

`task1-modified.bzc`:

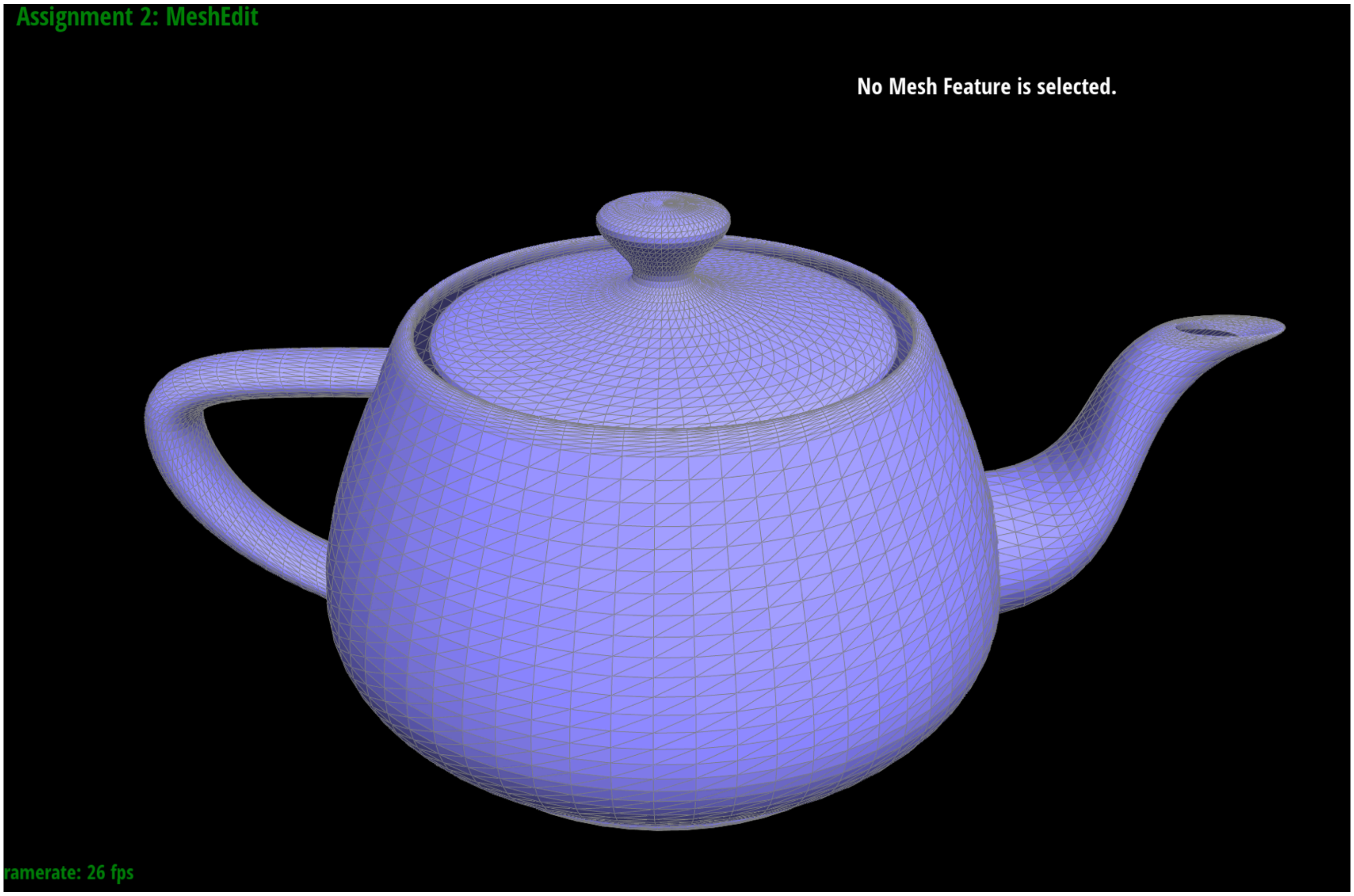


## Task 2: Bezier Curves with 1D de Casteljau Subdivision

**Briefly explain how de Casteljau algorithm extends to Bezier surfaces and how you implemented it in order to evaluate Bezier surfaces.**

We can extend de Casteljau's algorithm from evaluating Bezier curves to surfaces by taking a n x n grid of control points (rather than a single array) and calculate intermediate points along each curve of a row of points to evaluate the surface position corresponding to the parameters (u,v). Each row contains n control points $P_{i0}, \ldots, P_{i(n-1)}$ parameterized by u and we can recursively apply de Casteljau's algorithm to evaluate a point v on the "moving" Bezier curves in u. In our implementation, we do this by taking the n x n input vector points of control points and applying de Casteljau's algorithm to each row in the vector.

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

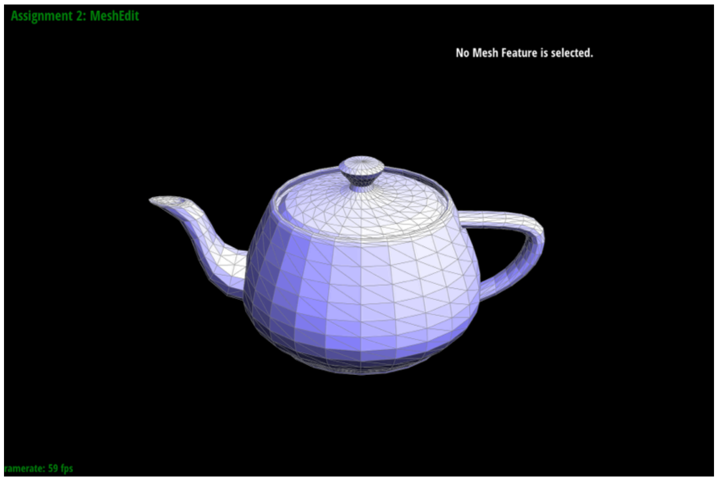
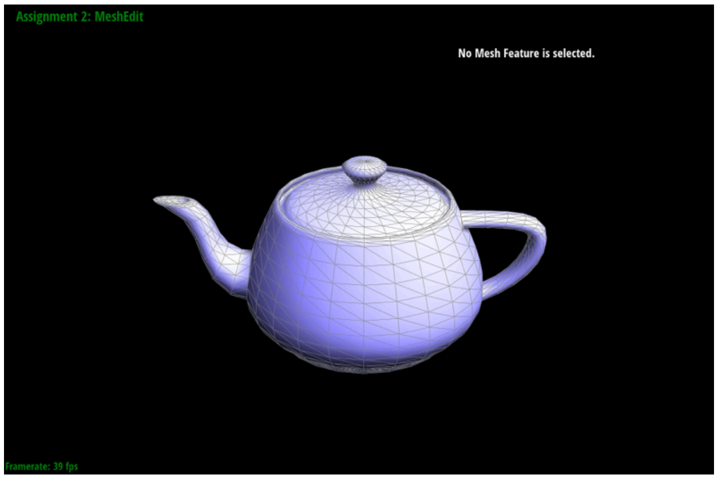


## Task 3: Area-Weighted Vertex Normals

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

To compute the area-weighted normal at a given vertex, we first iterated through the triangle faces neighboring the vertex using the `face()` and `next()` methods in the half edge data structure. We also called `vertex()->position` to get the neighboring vertices and their corresponding positions. Then, for each adjacent face we found the normal vector by calculating the cross product of the two edge vectors of the face and multiplied by its area. The result is then added to a 3D Vector result which keeps track of the current neighboring cross products calculated so far. Finally, we return the result's unit vector.

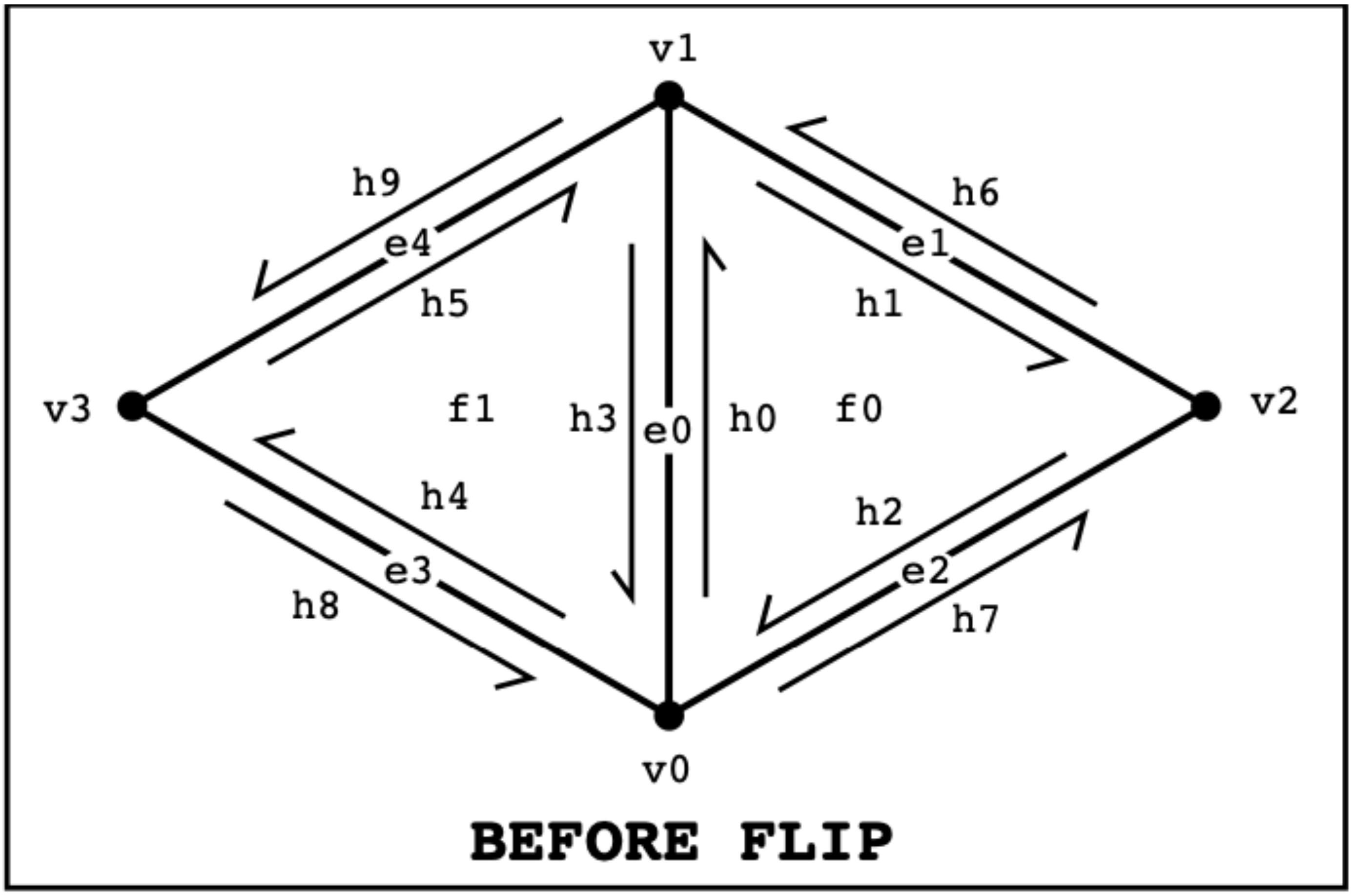
**Show screenshots of dae/teapot.dae (not .bez) comparing teapot shading with and without vertex normals. Use Q to toggle default flat shading and Phong shading.**

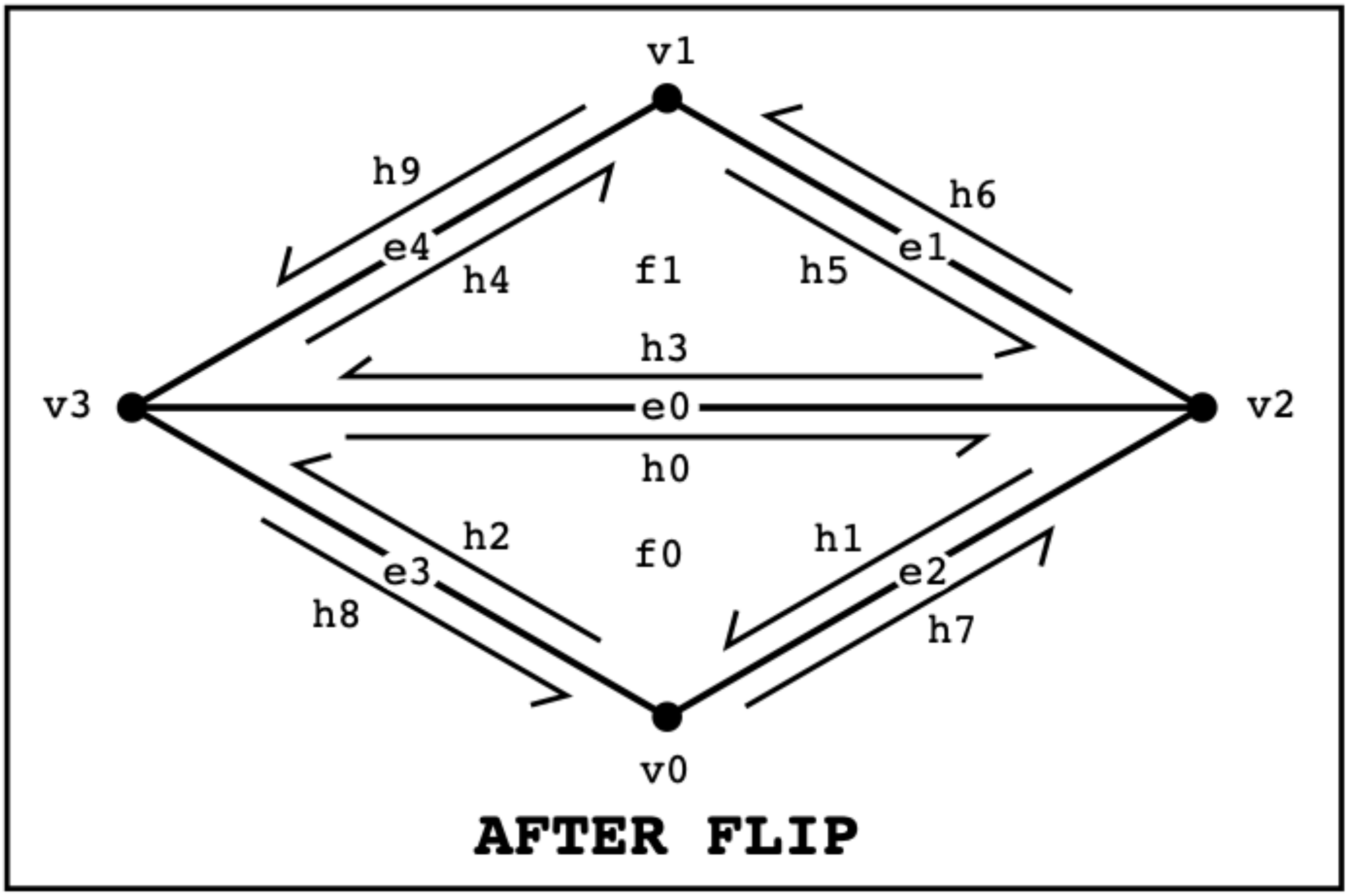
| | |
|---|---|
|  Assignment 2: MeshEdit<br>No Mesh Feature is selected.<br>ramerate: 59 fps default shading |  Assignment 2: MeshEdit<br>No Mesh Feature is selected.<br>Framerate: 39 fps Phong shading |

## Task 4: Edge Flip

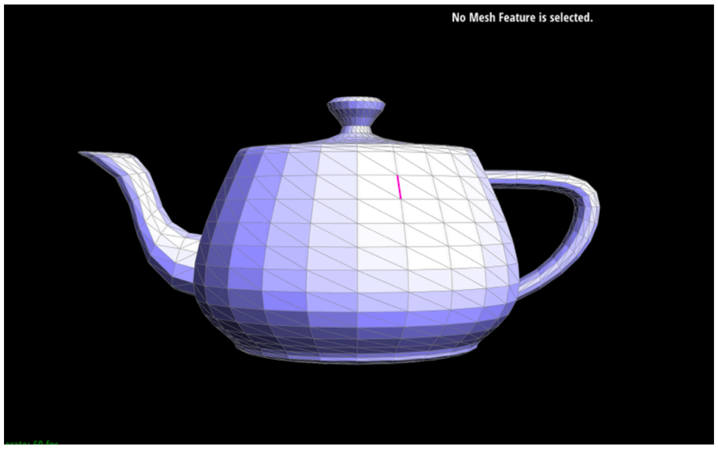
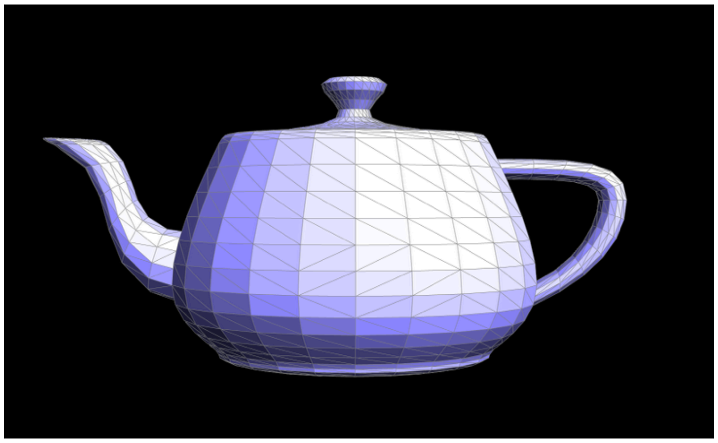
**Briefly explain how you implemented the edge flip operation and describe any interesting implementation / debugging tricks you have used.**

We first check if the given edge input is a boundary edge using the isBoundary() function, as boundary edges cannot be edge flipped and are returned. Then, we use the `twin()` and `next()` pointers to move around the mesh and store the proper mesh elements such as the half edges, vertices, neighboring edges, and faces from these half edges. Finally, we set the pointers for every element in the modified mesh to its correct element using the `setNeighbors` function. We used the following diagram for reference in visualizing the edge flip operation and how the mesh elements are connected:


v1
h9
h6
e4
e1
h5
h1
v3
f1
h3
e0
h0
f0
v2
h4
h2
e3
e2
h8
h7
v0
BEFORE FLIP


v1
h9
h6
e4
e1
h4
f1
h5
h3
v3
e0
v2
h0
h2
f0
h1
e3
e2
h8
h7
v0
AFTER FLIP

**Show screenshots of a mesh before and after some edge flips.**

| | |
|---|---|
|  before flip |  after flip |

**Write about your eventful debugging journey, if you have experienced one.**

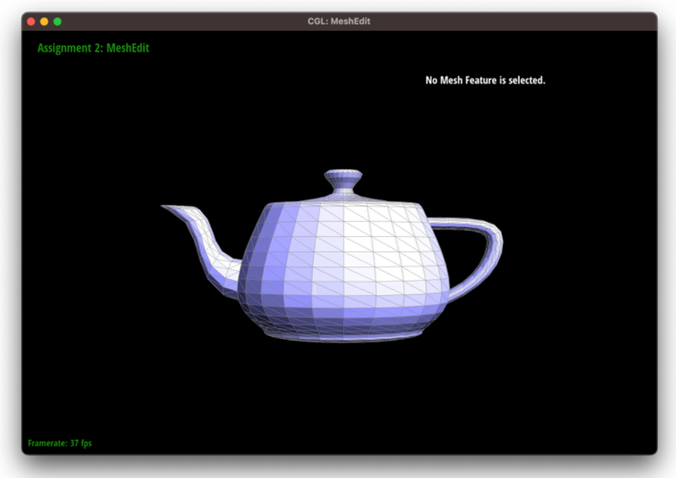
We encountered some bugs while trying to find the right pointers for the next half edges and twin edges. By referring to the diagram, we were able to trace through the direction of each pointer to the next mesh element such as a corresponding edge or vertex, and backtrace when needed when encountering segfaults when trying to access an invalid mesh element. We also frequently printed the current vertex, edges, and faces we were traversing through and stored them in separate lists to organize them.

## Part 5: Edge Split

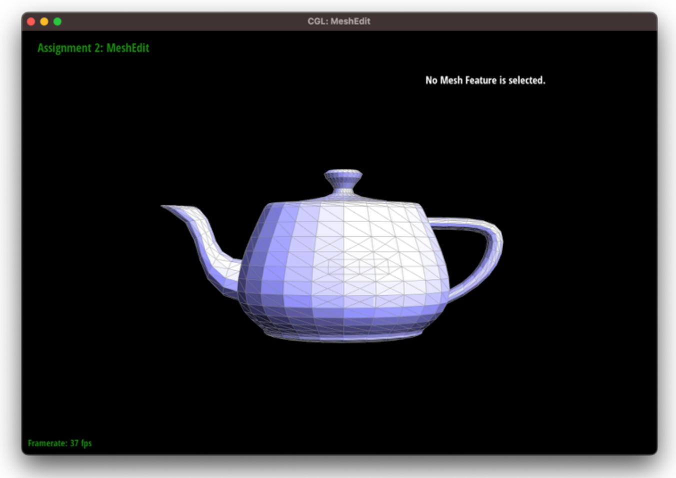
**Briefly explain how you implemented the edge split operation and describe any interesting implementation / debugging tricks you have used.**

Unlike in part 4 where we just set the new pointers with new neighboring mesh elements, edge splitting requires adding a new vertex that acts as a midpoint for opposing vertices along an edge. We first check if the edge is again a boundary edge, in which case we just return the edge from the function. Then, we use the same method as in Part 4 to traverse through the mesh elements using the `next()` and `twin()` functions, while also storing 6 new half-edges, 1 new vertices, 3 new edges, and 2 new faces. Finally, we update the pointers for the newly added vertex and neighboring vertices as well as the pointers for the new edges using `setNeighbors` .

**Show screenshots of a mesh before and after some edge splits.**

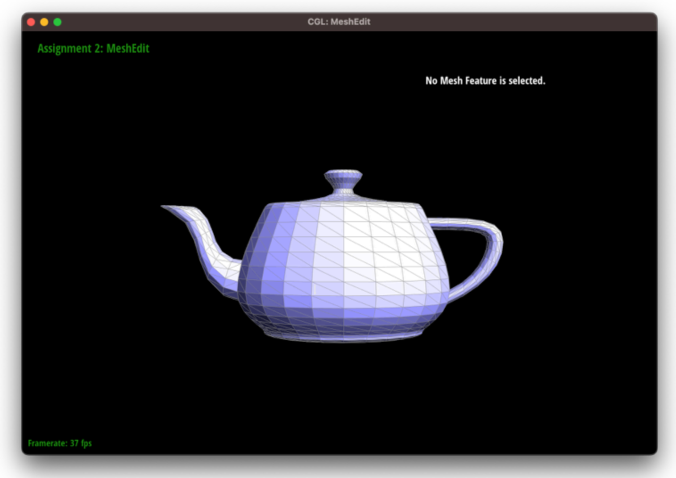


before split

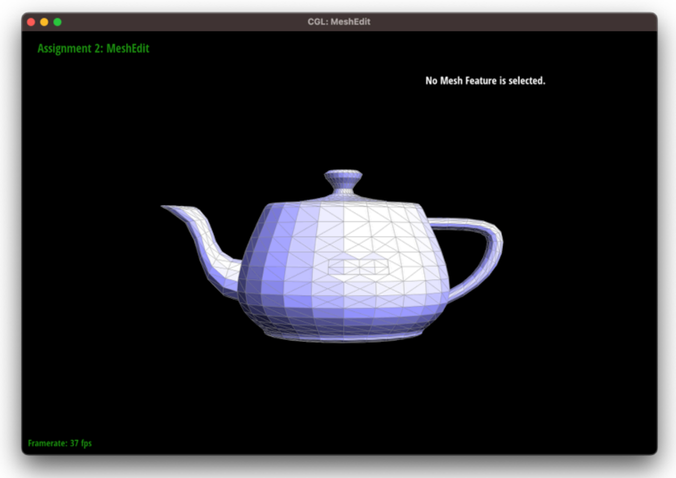


after split

**Show screenshots of a mesh before and after a combination of both edge splits and edge flips.**



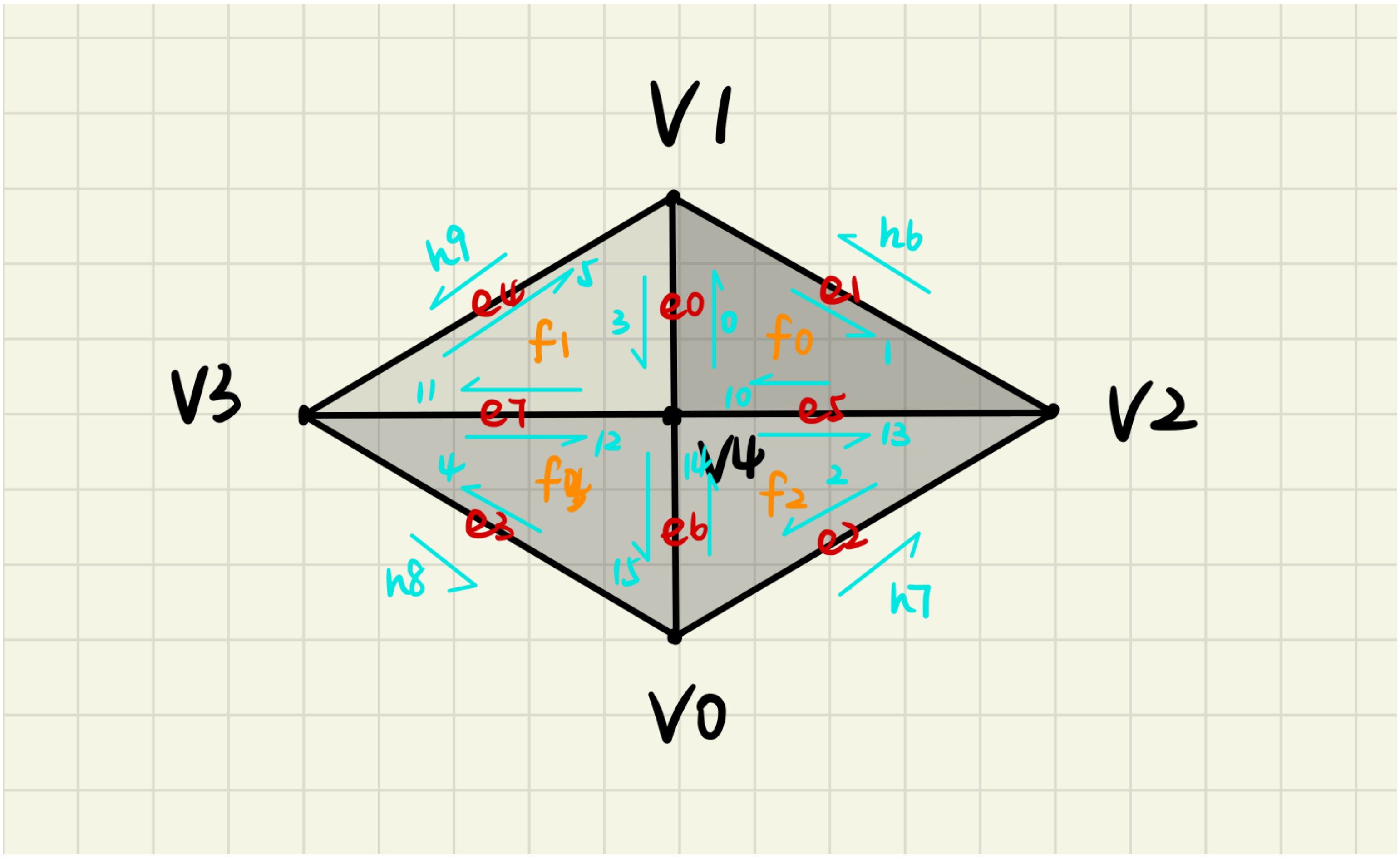
before split flip



after split flip

**Write about your eventful debugging journey, if you have experienced one.**

We used the following diagram to visualize the edge-splitting process and mesh traversal



Since a lot of logic of the mesh traversal was reused from part 4, there were fewer issues when debugging the proper half-edges and vertex pointers. To keep our mesh elements organized, we stored the added vertices, edges, and faces in their corresponding lists and set the updated half-edge pointers for the modified mesh at the end one by one to ensure that each pointer was properly set with its updated neighboring elements one at a time.

## Part 6: Loop Subdivision for Mesh Upsampling

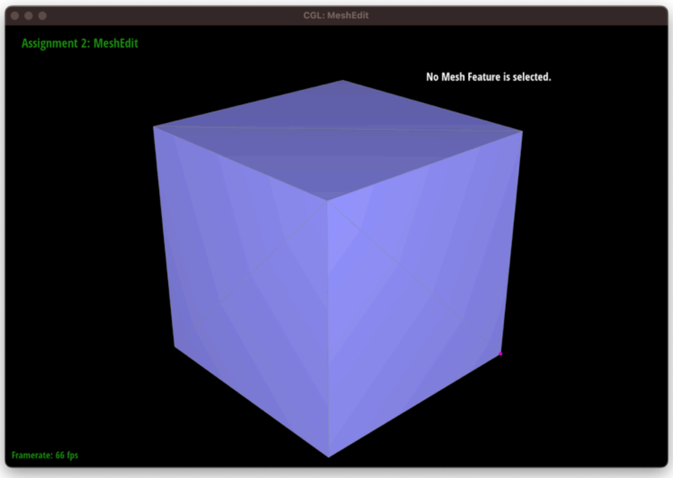
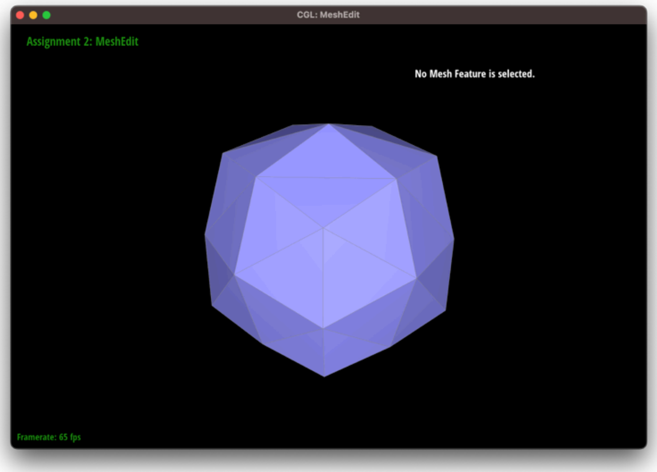
**Briefly explain how you implemented the loop subdivision and describe any interesting implementation / debugging tricks you have used.**

I modify the functions in part 4/5 so that these functions will set the `isNew` flags on newly created elements. This simplifies the implementation for part 6 quite significantly. For debugging, I simply uses Xcode breakpoints and the built-in llvm print tool.

**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?**

Sharp edges becomes smooth and rounded out after upsampling. For example, in `cube.dae` , the cube lost its shape and becomes more rounded after just upsampling once.

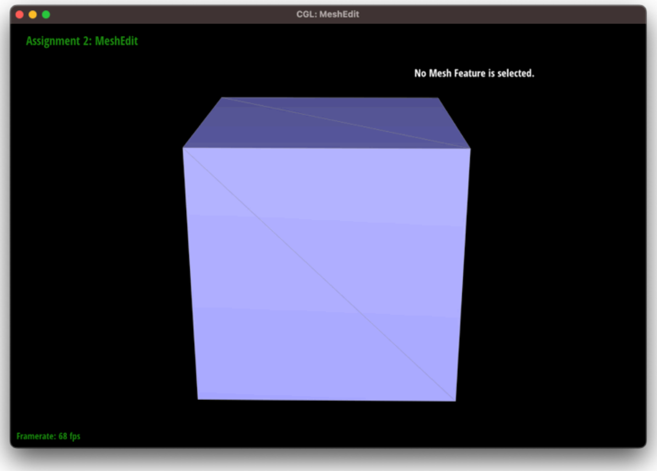
| | |
|---|---|
|   CGL: MeshEdit<br>Assignment 2: MeshEdit<br>No Mesh Feature is selected.<br>Framerate: 66 fps Cube.dae, no upsampling |   CGL: MeshEdit<br>Assignment 2: MeshEdit<br>No Mesh Feature is selected.<br>Framerate: 65 fps Cube.dae, upsampled once |

This is an unwanted effect if the intent is to have shape edges and corners. We can mitigate this by pre-splitting those edges on the corner because that makes the area denser in vertices, and this will help maintain the sharpness in that area after splitting.

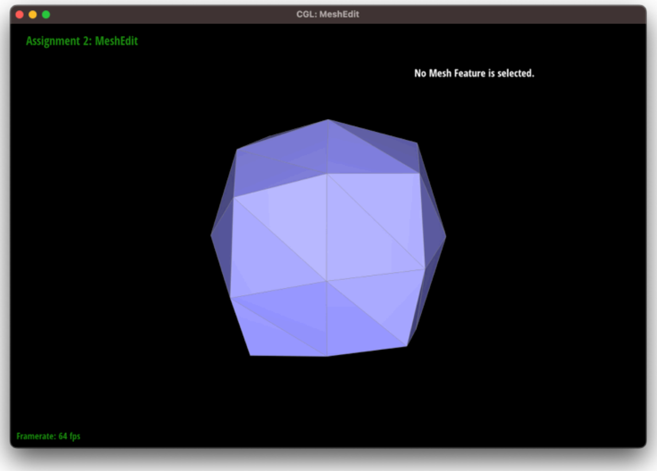
**Load dae/cube.dae. Perform several iterations of loop subdivision on the cube. Notice that the cube becomes slightly asymmetric after repeated subdivisions. Can you pre-process the cube with edge flips and splits so that the cube subdivides symmetrically? Document these effects and explain why they occur. Also explain how your pre-processing helps alleviate the effects.**

From ths camera angle, we can clearly see that the cube become asymmetrical after one upsampling.

| | |
|---|---|
| | |

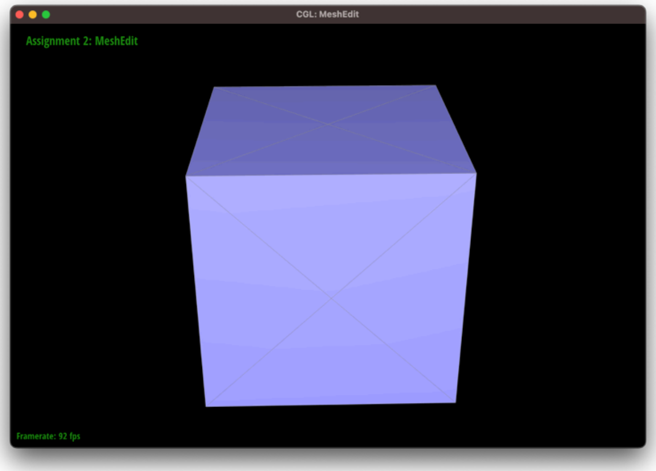


Cube.dae, no upsampling

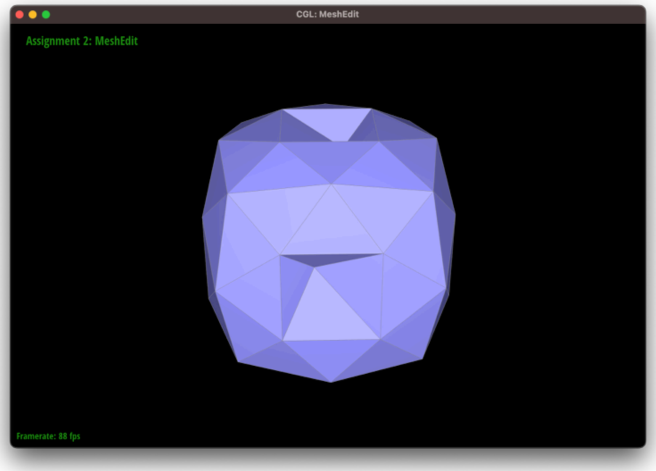


Cube.dae, upsampled once

This is because, as we can see in the picture, the two triangles that comprises the face of the cube is not symmetrical. Therefore, we can fix it by pre-splitting the diagonal edge on each face once, as shown in the pictures below:



Cube.dae, no upsampling



Cube.dae, upsampled once

**Project Webpage:** https://cal-cs184-student.github.io/p1-rasterizer-sp23-d-2/