# CS 184: Computer Graphics and Imaging, Spring 2022

# Project 2: Mesh Editor

Jacob Hsiung, CS184-22

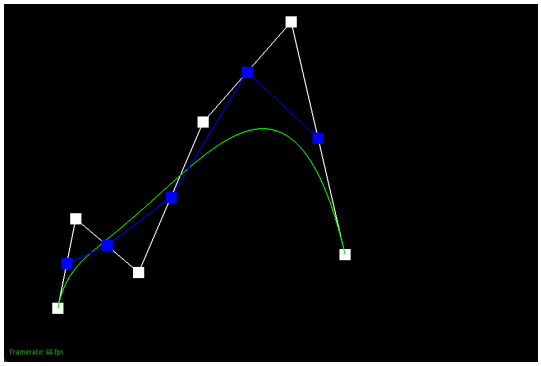
## Overview

The main topic of this project is meshes. Before directly dealing with meshes, we start with drawing 2D Bezier curves. Then, we used what we implemented in 2D and apply it to higher demensions to get Bezier surface. After that, we are ready to deal with meshes. Meshes are representd using HalfEdge class. To begin with, I implemented a way to calculate area-weighted vertex normals to achieve smoother shading. Secondly, we implement flipEdge and splitEdgte to perform edge-flipping and edge-flipping. With these 2 functions, we can then apply upsampling by loop subdivision.

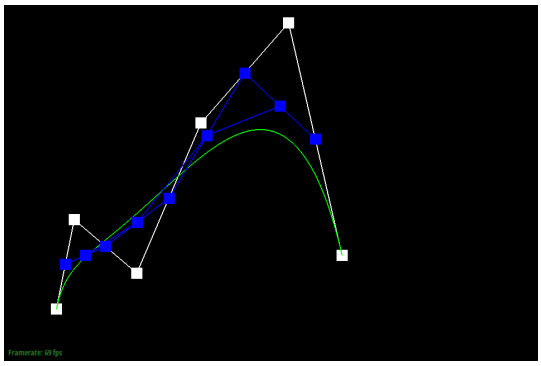
## Section I: Bezier Curves and Surfaces

### Part 1: Bezier curves with 1D de Casteljau subdivision

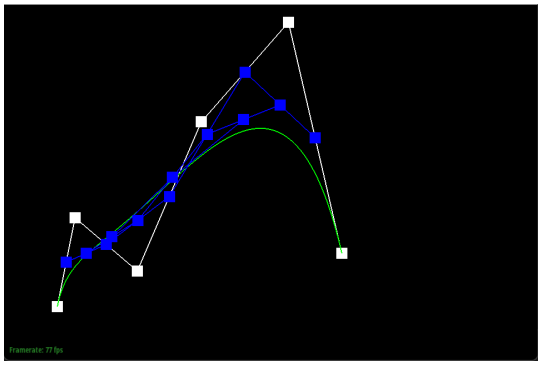
de Casteljau's algorithm describes the process of reducing a set of vertices using linear interpolation with a parameter t. In each iteration, we reduces the number of vertices by 1 until we are only left with 1 vertices. We then draw a cruves passing through the last point. Using this algorithm, I implemented evaluateStep, which calculate the result of the set of vertices after 1 step.



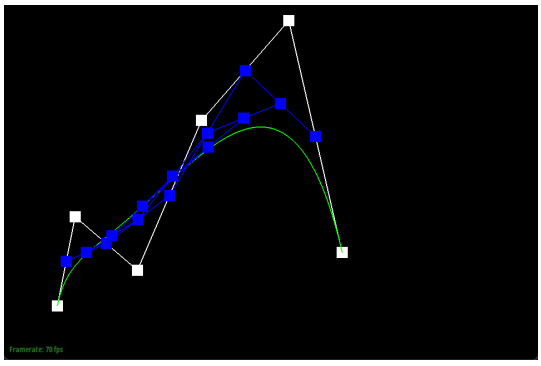
Step One



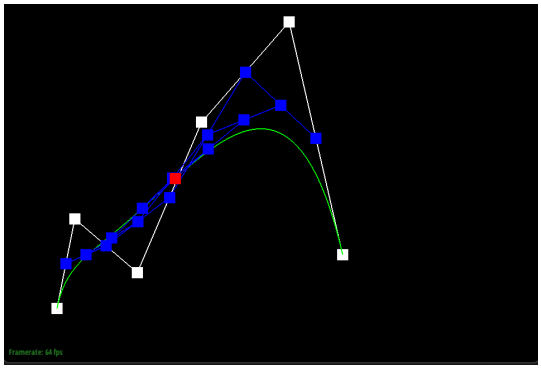
Step Two

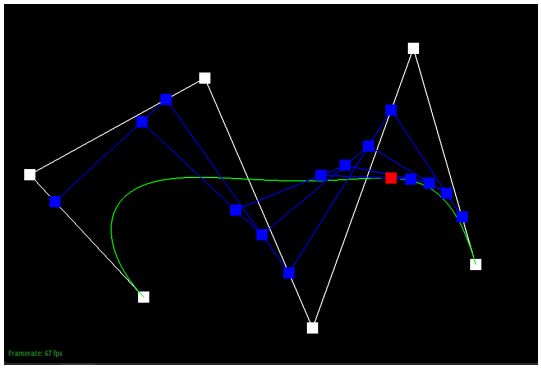


Step Three



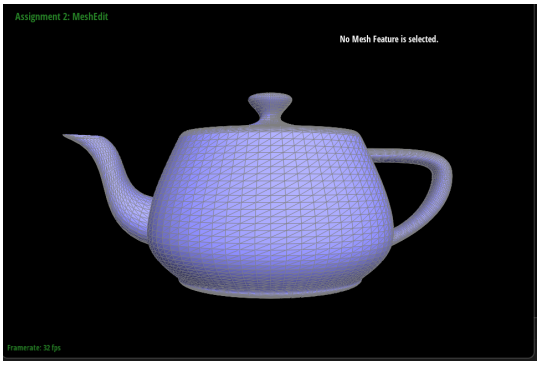
Step Four


Step Five


Different t and points

### Part 2: Bezier surfaces with separable 1D de Casteljau subdivision

The implementation of Bezier surfaces can be seperated into three functinons: evaluateStep, evaluate1D, and evaluate. The evaluateStep function is exactly the same as the case in lower dimension, but we need to change the class to Vector3D. Each call to the function performs one step of de Castejau algorithm. The evaluate1D function repeatedly calls evaluateStep until there is only 1 point left. We can call evaluate1D on every row of the surface, so the surface will be reduced to only several points, each point representing a row. Last but not least, the evaluate function calls evaluate1D and passes these set of points as argument to obtain a single point.

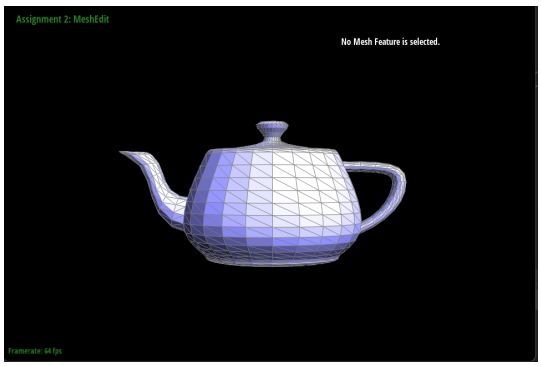


Teapot

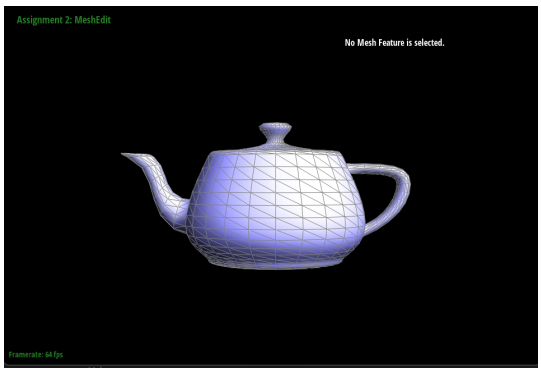
## Section II: Sampling

### Part 3: Average normals for half-edge meshes

Taking advantage of the HalfEdge mesh data structure, we can traverse through all adjacent faces easily. Since the magnitude of the cross product is proportional to the area of the face, we can just simply add up all the normal vectors of the adjacent faces. At the end, we only need to normalize the sum and that would be our average normals.



Flat Shading



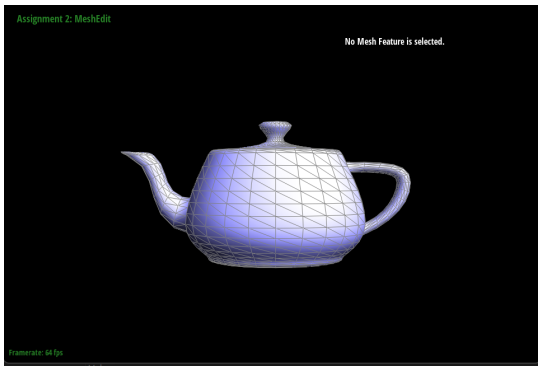
Phong Shading

We can clearly see the difference between the two shading method. One provide a smoother shading while another is a little bulky.

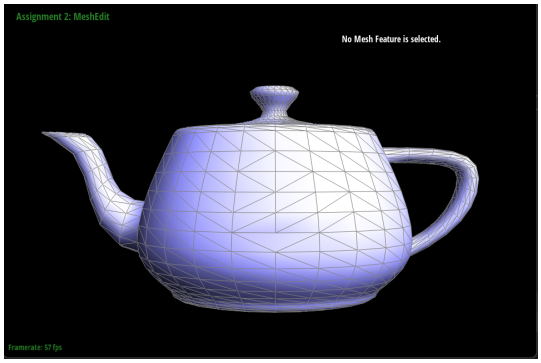
### Part 4: Half-edge flip

The implementation of splitEdge may seem a little intimidating at the first glance, but it is actually rather simple(if everything was set correctly from the beginnig). I use the recommended method on the project spec page. Firstly, I draw out two traingles sharing an

edge and clearly label the faces, halfedges, and vertices. Then, I draw out the resulting triangles after edge-flipping with everything labeled. Lastly, I go through every faces, vertices, edges, and halfedges, and set the correct paramteres using setNeighbors.



No Flipping

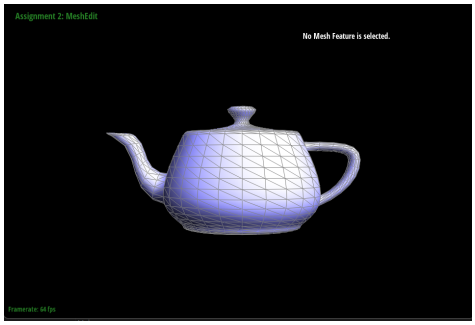


Flipped

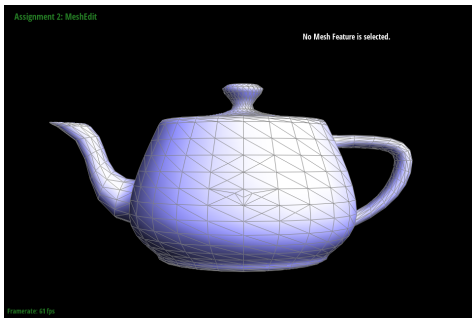
One interesting way to check the logic of your code is that, the everything should remain the same if you flip an edge twice. The everything here includes not just the appearance, but also the variables in the data structure. This logic is very useful to check if every pointer was set correctly.

## Part 5: Half-edge split

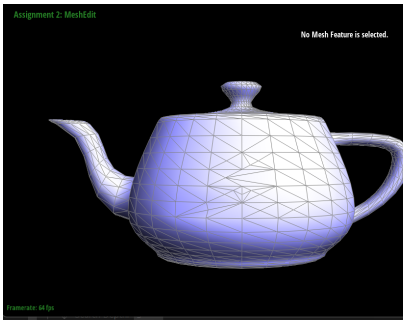
Splitting edges is slightly more complicated then flipping in that we are not only reassigning varaible, we are also adding new elements to the mesh. I start by creating all the elements that need to be added to the mesh. Then, I calculate the midpoint and assign its halfedge. Then, I use the setneighbors function to assign correct value to all the halfedges. Lastly, I updated the halfedges of all the vertices, faces, and edges.



No Splitting



Split



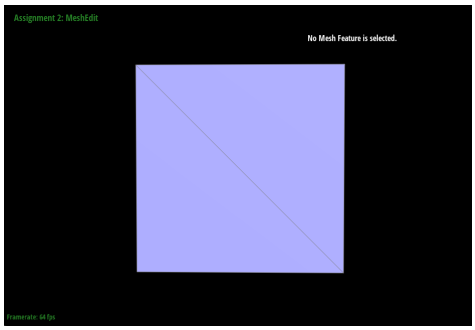
Flipped and Split

There is a really interesting bug. When I was assigning the the new position to the midpoints using midpoint->position = (v0->position + v1->position) / 2.0. This line of code was skipped when compiling. I was stuck on this for hours, but then when I deleted the entire block of code and retyped the exact same code, everything worked. I'm still not able to figure out what is the cause of the bug

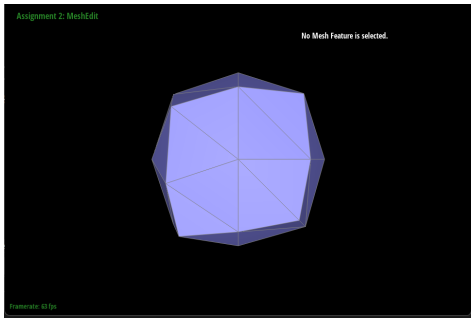
## Part 6: Loop subdivision for mesh upsampling

The implementation of Loop subdivision can be seperated into 5 loops. The first loop iterated through all the edges and calculate the newPosition using the given formula. The second for loop then iterates through all the vertices and calculate the newPosition of these old vertices. The thrid while loop iterates thorugh all the edges again and apply splitEdge to all the edges and assign the precalculate position to the vertex returned. The fourth for loop iterates through all the edges again and check if the edges is connecting a new vertex and a old vertex. If the condition is true, we flipp the edge. The last for loop iterates through all the vertices again, but this time, we assign the newPosition to position.

This part of the project was, indeed, a little tricky. The main issue I ran into was that my funcition went into infinite loop which crashes my computer. I had to restart my computer 4 times before figuring out the bug. I knew it had to do with the third loop, but I wasn't sure how I could solve it. I then realize I was adding edges while iterating over the edges. This means the boundary condition mesh.edgesEnd() will never return true. I had to add a second conditions which is edge_count is smaller than edge_num. One thing to notice here was that we cannot have edge_count smaller than mesh.nEdges() beacuse mesh.nEdges() will be increasing as we splitted the edges. We would still run into infinite loop.

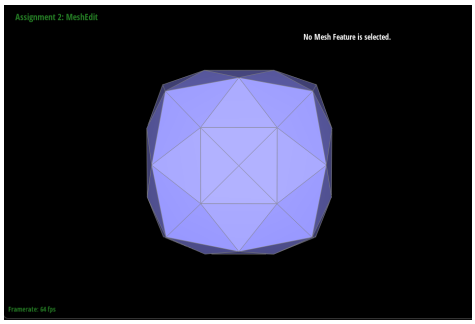



No Upsampling

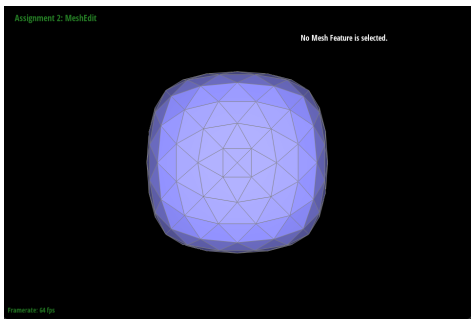



Upsampling Once

By comparison, the sharp corners are still sharp after upsampling. I don't think we can smoothen the sharpt corners by pre-splitting. However, if we perform upsampling multiple times, the sharp corners and edges would disappear.




Upsampling Once




Upsampling Twice

We can make the shape more symmetric by splitting all 6 diagonals of the cube. In my opinion, the reason why the the original cube becomes assymetric when upsampling is because each side of the cube is only consist of 2 triangles. When upsampling these two triangles, we are, in a way, streching the side in the two direction. But this can be alleviated if we split the diagonals. Now each side is consisted of 4 triangles, we will be streching in the four directions, and there will be no assymetric issue.