CS 184: Computer Graphics and Imaging, Spring 2023

Project 2: Meshedit

Jonathan Guo, Wentinn Liao, cs184-jgwl

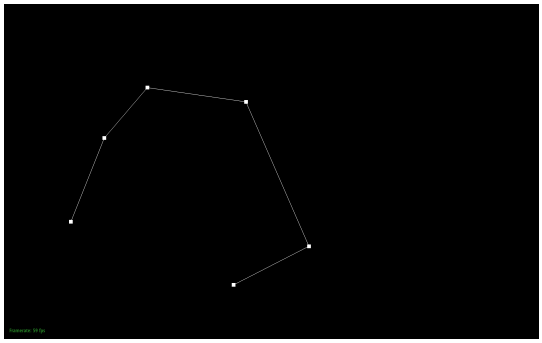
Overview

In this project, we implemented Casteljau's algorithm to compute Bezier interpolations of control points, which allows us to compute smooth curves and surfaces with little memory requirements. We also implemented area-weighted vertex normals, which allow us to shade polygons using barycentric interpolation, which allows for a smoother expression of the object. Finally, we implemented functions to split and flip half edges in the mesh data structure, which are tools for the loop subdivision algorithm. The subdivision algorithm creates finer meshes by interpolating and smoothing existing vertices, which allows us to smooth a coarser mesh into a finer, rounder mesh.

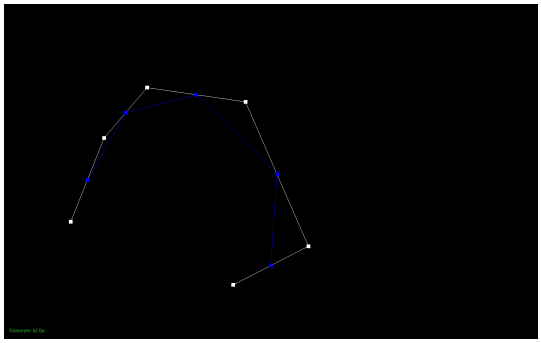
Section I: Bezier Curves and Surfaces

Part 1: Bezier Curves with 1D de Casteljau Subdivision

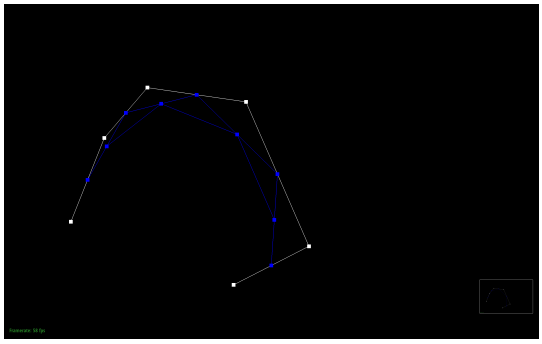
The Casteljau algorithm for Bezier curves recursively computes a linear interpolation of the control points. For each iteration, we iterated through the given control points and computed (1-t)p[i] + tp[i+1], which we then returned. If this function was called recursively until there was one point left, that point would be on the bezier curve. The figures below show how each iteration is computed from the previous iteration, using t = 0.5 (so it takes the midpoint of the previous control points)



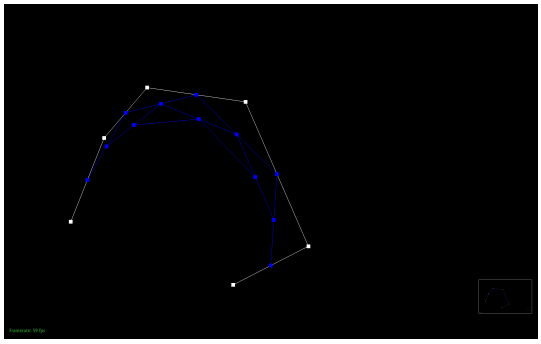
Iteration 0 *bzc/curve6.bzc*.



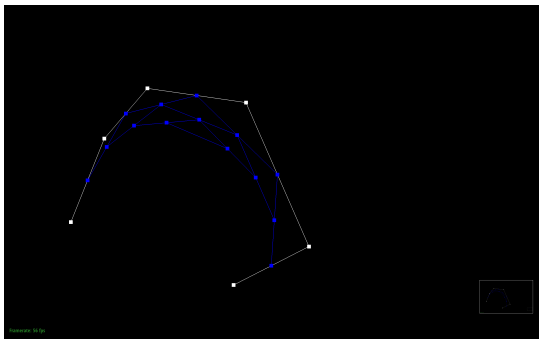
Iteration 1 *bzc/curve6.bzc*.



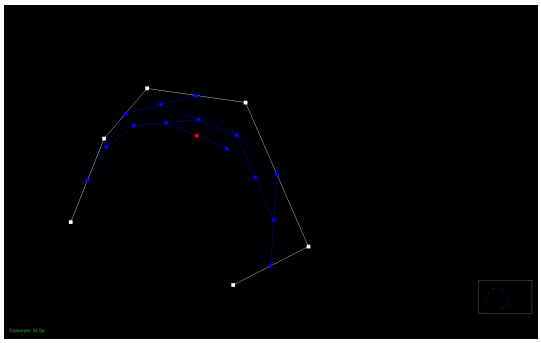
Iteration 3 *bzc/curve6.bzc*.



Iteration 3 *bzc/curve6.bzc*.

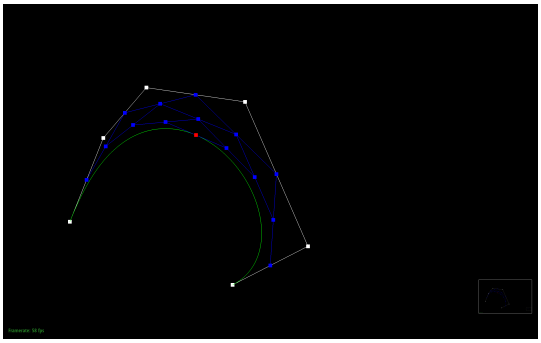


Iteration 4 *bzc/curve6.bzc*.



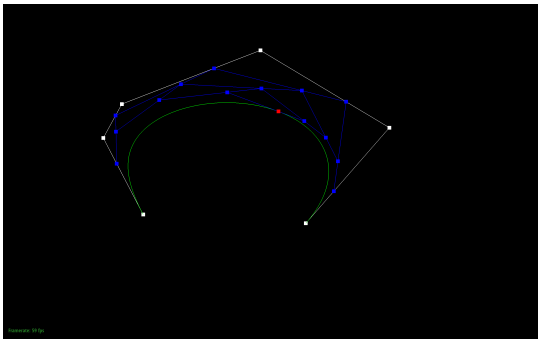
Iteration 5 *bzc/curve6.bzc*.

The image below shows the completed Bezier curve, computed by iterating over many t values between 0 and 1 using the above method.



bzc/curve6.bzc.

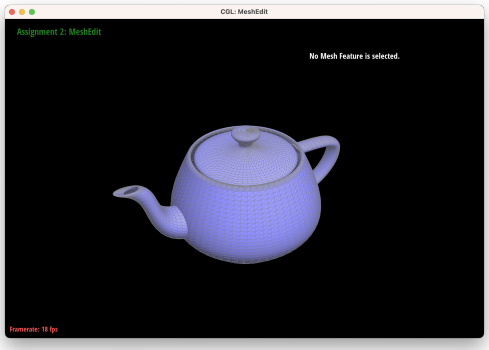
Finally, the image below shows the Bezier curve after some of the original control points were modified and with a different t value.



bzc/curve6.bzc.

Part 2: Bezier Surfaces with Separable 1D de Casteljau

The Casteljau algorithm for Bezier surfaces simply calls the 1D version multiple times. For the first dimension, 1D Casteljau interpolation is called on each row for parameter **u**. For an **n x n** grid of control points, this produces a vector of **n** intermediate control points. 1D Casteljau interpolation is called again on these intermediate control points with parameter **v** to obtain the final desired point.



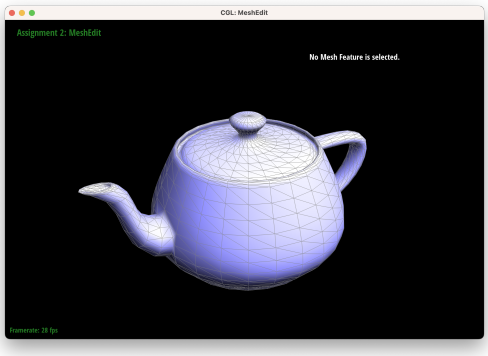
bez/teapot.bez.

Section II: Triangle Meshes and Half-Edge Data Structure

Part 3: Area-Weighted Vertex Normals

To reduce repeated code, we noted that the existing **normal()** function of **Face** already computes a normal vector whose norm is twice the area of the face itself, however, returns the normalized version. Thus, we wrote a function **areaWeightedNormal()** that returns the penultimate result of the code for **normal()**, and modify **normal()** to call that.

Then, to iterate over the faces, we select a starting halfedge then iterate by changing **h** to **h->twin()->next()**. Each iteration, we retrieve the face the halfedge is on, and accumulate the sum of the area weighted normals. This produces a weighted sum of the face normals adjacent to the vertex, and at the end we simply call **unit()** to normalize.

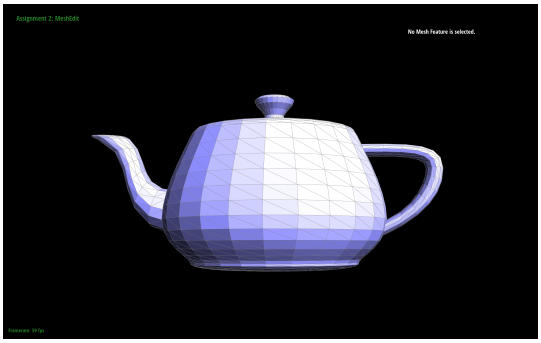


Area Weighted Vertex Normals: dae/teapot.dae.

Part 4: Edge Flip

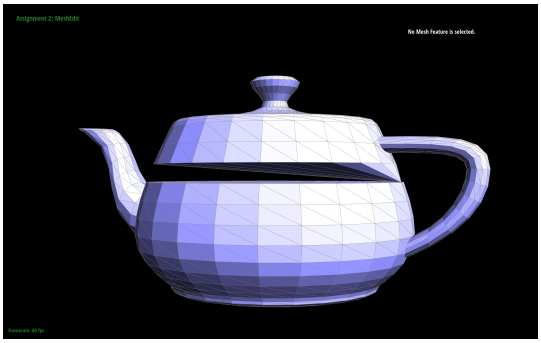
To implement Edge Flip, first we gathered all of the faces, vertices, and half edges necessary. Then, we set the neighbors of all 6 half edges, and then set the appropriate pointers of the faces and vertices. Note that we did not create a new half edge; we took the old one and just changed where it pointed from and to.

We ran into an error where we were unable to set the half edge of faces and vertices because it was a protected field, but by playing around with pointers and stuff we were able to get it to work.



Teapot before many edge flips

The image above shows the teapot before edge flips.



Teapot after many edge flips

The image above shows the teapot after some edge flips. We were able to carve a slab through the middle of the teapot.

Teapot after many edge flips

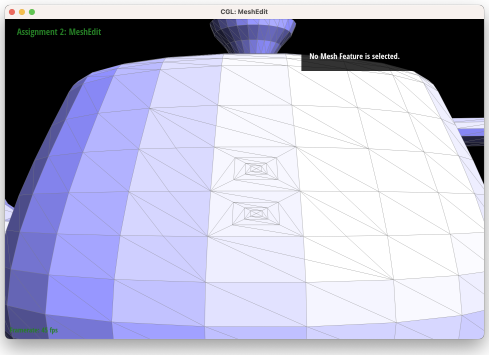
The image above shows the same teapot as before, but using the smooth shading instead.

Part 5: Edge Split

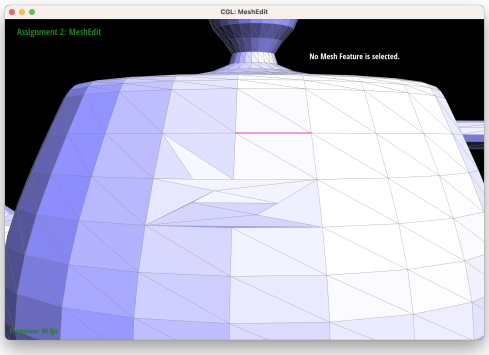
Carefully. The overall implementation was separated into 5 sections:

- Getting all vertex, halfedge, and face objects and naming in a way that is easy to understand
- Creating the midpoint, new halfedges, edges, and faces with the same naming convention as before
- Setting pointers for new objects
- Setting pointers for persistent old objects
- Deleting nonpersistent old objects

Small things that were almost missed were making sure to update the four original halfedges with new faces, and making sure to correctly delete the original faces. Keeping track of all data structures and variables that needed to be changed allowed us to implement it correctly first try.



Edge splits *dae/teapot.dae*.



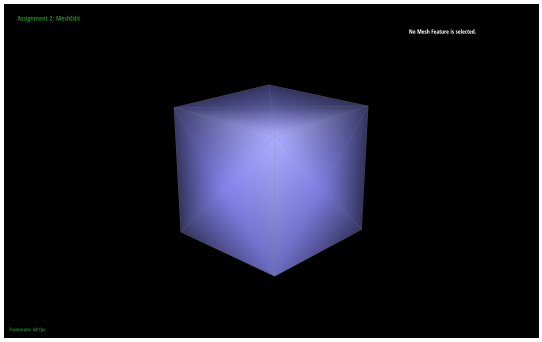
Combination of edge splits and flips *dae/teapot.dae*.

Part 6: Loop Subdivision for Mesh Upsampling

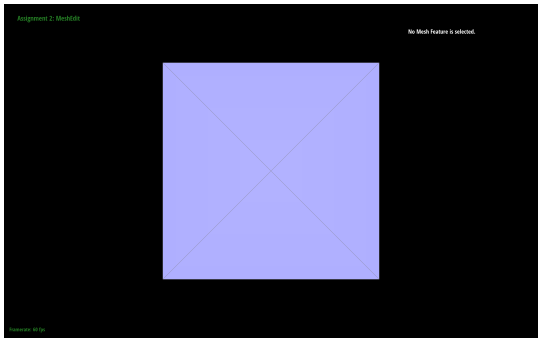
We implemented loop subdivision using the recommended method. First we calculated the new position of both the new vertices as well as the old vertices. Next, we split every edge. Here, because the list of edges is stored as a linked list, and new edges are appended to the end, what we did was we knew we only had to split a certain number of edges, so we blindly used a for loop to split that number of edges. This way we could use the isNew flag for the flipping. After splitting we did flipping and then set the vertex positions to the calculated ones.

Sharp edges and corners become less sharp (more rounded) after loop subdivision. This is because during loop subdivision, we move each vertex to a new position based on its neighbors. So, if a vertex or edge is very sharp, it will be moved closer to its neighbors and become less sharp.

We can preprocess the cube to avoid assymetry. The way to do this is to split every edge that goes along the face of the cube. This way, each face has an X instead of just a \ or a /, leading to more symmetry.

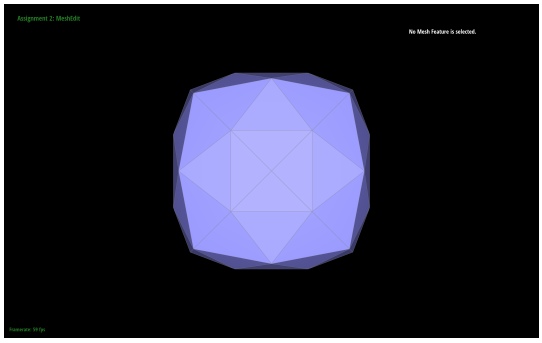


Cube after preprocessing*./dae/cube.dae*.

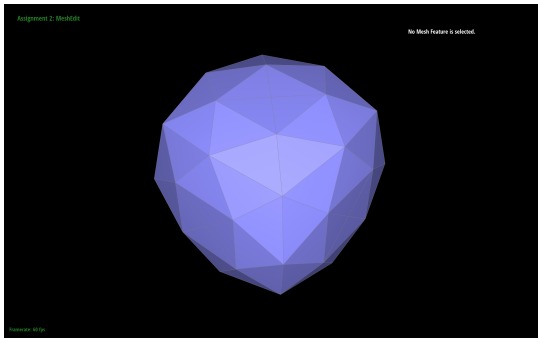


Cube after preprocessing*./dae/cube.dae*.

Next, once we upsample once, we see that the cube is symmetrical, because each face (before the upsampling) was symmetrical.

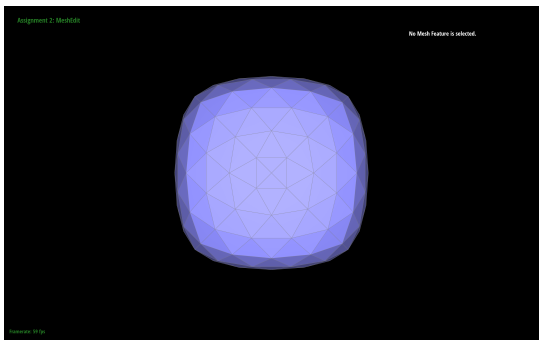


Cube after 1 loop subdivision*./dae/cube.dae*.

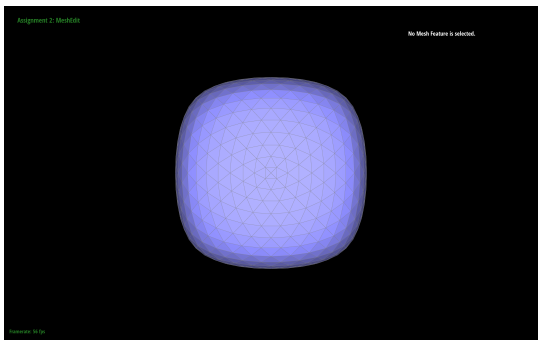


Cube after 1 loop subdivision*./dae/cube.dae*.

We can also do more loop subdivisions.



Cube after 2 loop subdivisions*./dae/cube.dae*.



Cube after 3 loop subdivisions*./dae/cube.dae*.