CS 184: Computer Graphics and Imaging, Spring 2024

Homework 2: Mesh Edit

Satvik Muddana

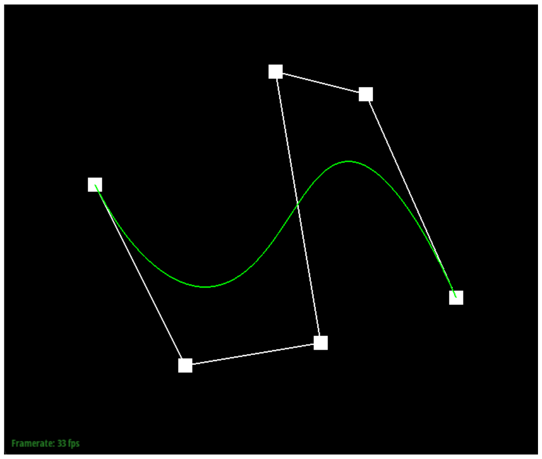
Overview

In this homework, I implemented various editing operations for smooth surfaces that built up towards full meshes. It started with Bezier curves and surfaces which are good at respresenting smooth surfaces and require less memory. However, they are more difficult to render directly. So, the rest of the homework, I worked on a triangle mesh editor and implemented things like edge flips and edge splits. With those two functions together, I could also implement loop subdivision. Overall, this homework was great at offering a hands-on demonstration of concepts like Bezier curves and halfedge components.

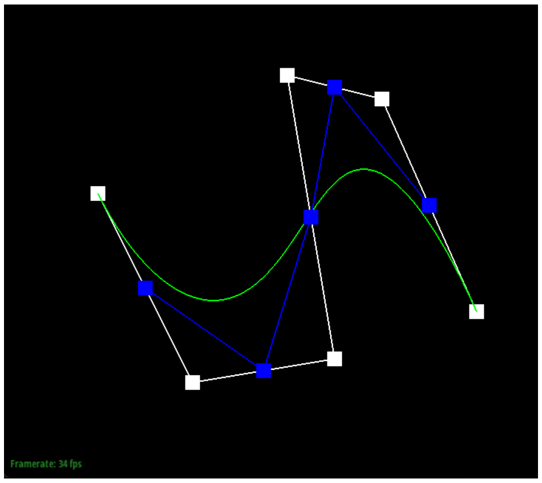
Section I: Bezier Curves and Surfaces

Part 1: Bezier Curves with 1D de Casteljau Subdivision

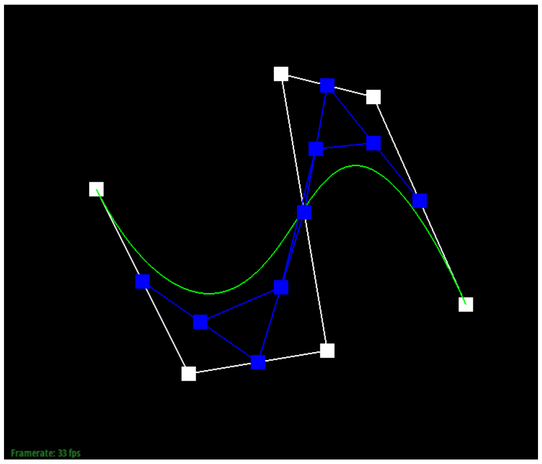
One can determine the position of a point on a Bezier Curve by a scalar t using the de Castelijau algorithm. It starts off with n control points p_1 to p_n and the scalar t which is a value between 0 and 1. Then there is recursive step which is what I implemented in the evaluateStep function. This recursive step outputs n-1 linearly-interpolated points. The formula to calculate each point is p'_i = (1-t)*p_i + t*p_(i+1). We can see the results of reach recursive step in the images below. The green line is the actual Bezier curve and the blue lines are the linearly-interpolated points. The red point is where the algorithm converged for a specific value of t. The last image just has a different t value and slightly different points.



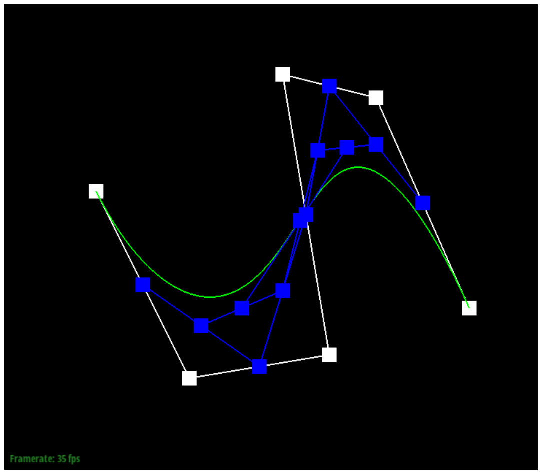
Level 0



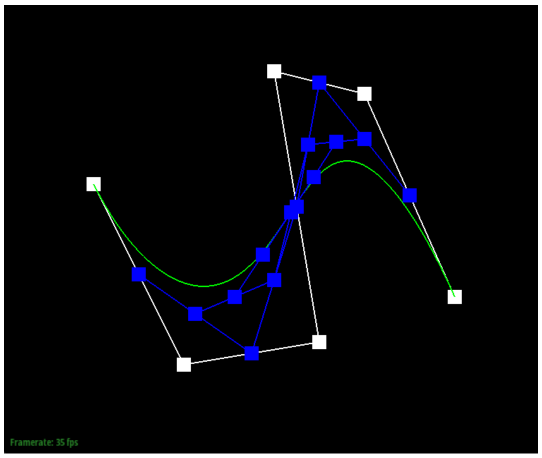
Level 1



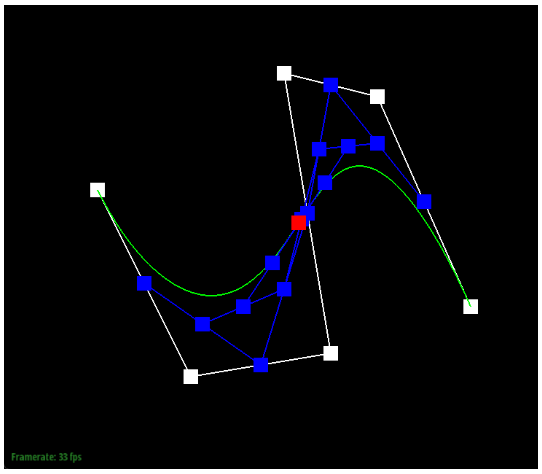
Level 2



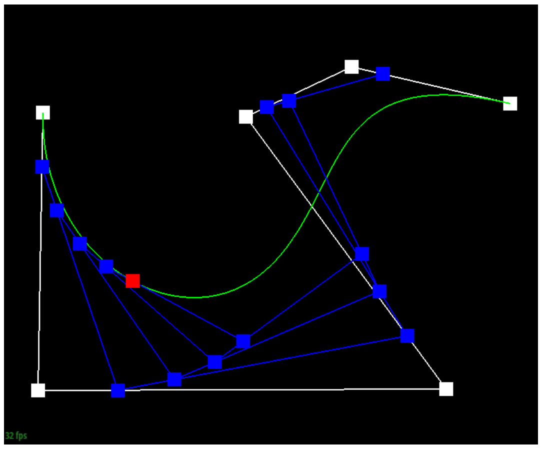
Level 3



Level 4



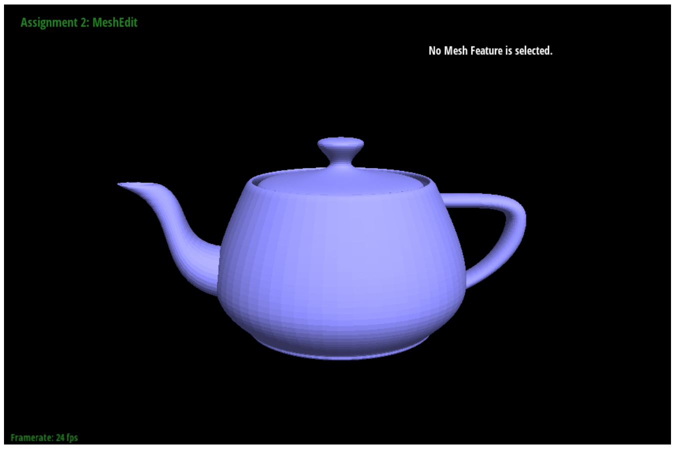
Level 5



Slightly modified Bezier curve with different t

Part 2: Bezier Surfaces with Separable 1D de Casteljau

The de Casteljau algorithm extends to Bezier surfaces because we can represent a Bezier surface as many 1D Bezier curves. If we are given a certain amount of Bezier curves that each have n control points and a common scalar u that parametrized them, we can take another parameter v and use each curve like a control point for a Bezier surface. Thus, we apply the 1D De Casteljau algorithm along the rows of the 2D vector, yielding a set of m Bezier curves evaluated at u. Subsequently, utilizing these m resulting points, we evaluate the 'sliding' Bezier curve at v using the De Casteljau Algorithm. This methodology facilitates the creation of intricate geometries, encompassing features like hard edges, beveled surfaces, and discontinuities, through a patchwork of Bezier surfaces, as illustrated in the accompanying image.

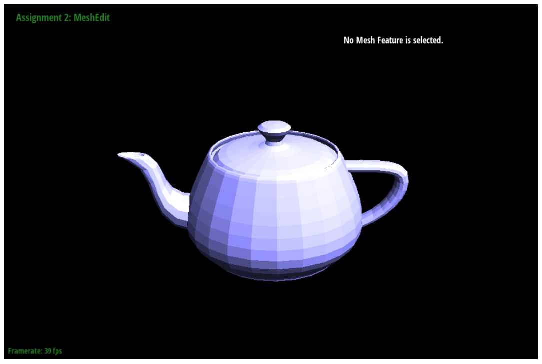


Bezier Surface Teapot

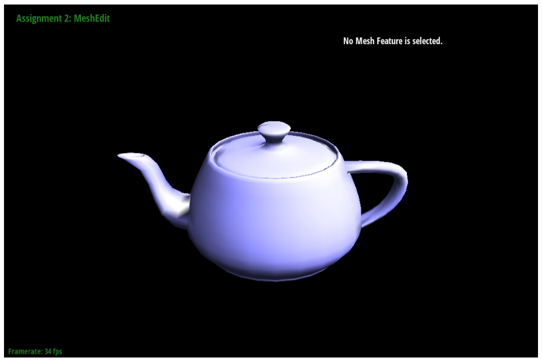
Section II: Triangle Meshes and Half-Edge Data Structure

Part 3: Area-Weighted Vertex Normals

To calculate the normal for a vertex in the mesh, I traversed the faces surrounding it, accessing their normal vectors and weighting them by the area of their respective faces. Traversing the mesh involved retrieving the halfedge associated with the target vertex and navigating through outgoing halfedges to reach adjacent faces. The normal vectors were readily available from the face elements, but determining the face area was complex more. I calculated the face area by forming temporary vectors from the face vertices and dividing their cross-product by two. Additionally, the cross-product inherently yields a vector normal to the plane formed by its arguments, eliminating the need to retrieve the normal from the face element. Finally, I computed the vertex normal as a weighted sum of the adjacent face normals, where weights corresponded to the fractional area of each face. This method significantly improved shading, as demonstrated by the images below where the difference between flat shading and phong shading is evident.



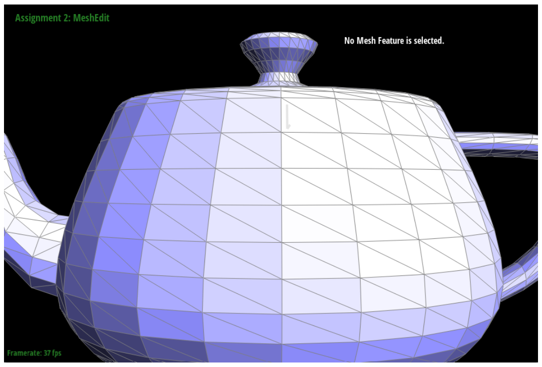
Flat Shading



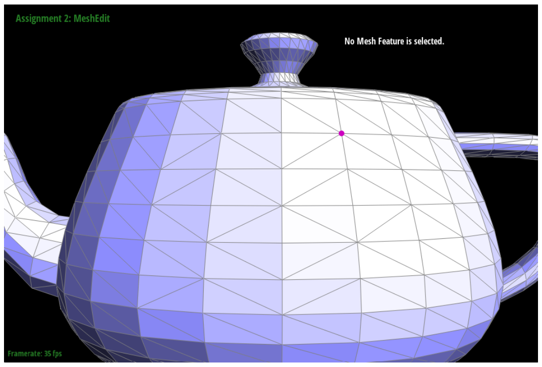
Phong Shading

Part 4: Edge Flip

Implementing the edge flip function was mostly changing pointers. I followed the recommended steps in the spec very carefully to the point I had very little issues with debugging. It was all about ensuring that I did not miss a step, but the steps were all laid out for me because I made a visual reference for myself. I started off by getting all the relevant halfedges and then using those to get all the relevant edges, faces, and vertices. From there it was just a matter of understand what changes in an edge flip and setting all the pointers of every variable to the right place. Looking at the implementation of the setNeighbors function helped a lot with this. Overall, I found that drawing it out helped tremendously in avoiding bugs. Below are before and after pictures of edge flips on a teapot mesh.



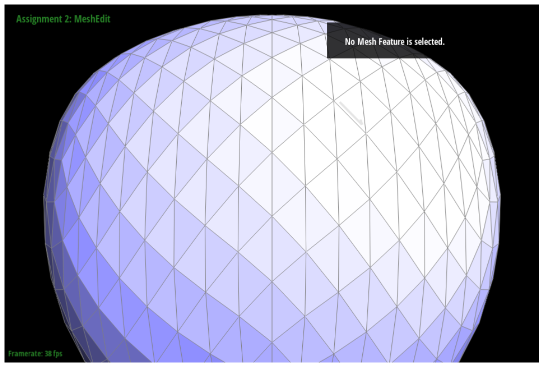
No edge flips



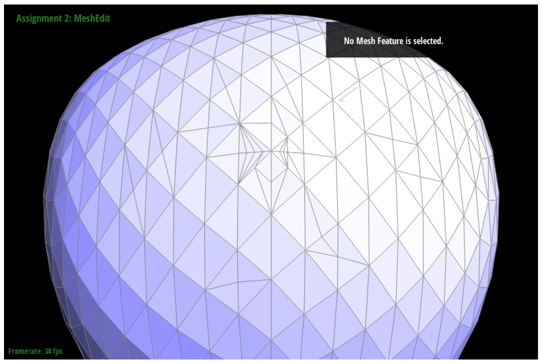
After some edge flips

Part 5: Edge Split

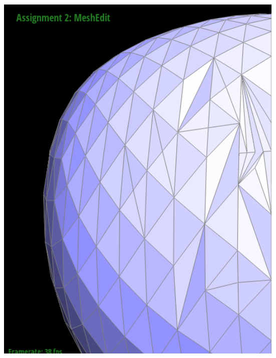
Implementing the edge split function took the same if not more caution that the edge flip function. Here, not only does one have to change pointers, but also create new elements. I specifically had to newly make 6 halfedges, 3 edges, 1 vertex (placed at midpoint of given edge), and 2 faces. Drawing it out still helped a lot in reducing the amount of debugging needed. Overall, the implementation was similar to the edge flip function with new elements. Since I was already creating a bunch of variables for each existing element, the new elements could be utilized similarly. The biggest mishap I had was not noticing that edges and vertices has an isNew field. This was not an issue in this question, but for part 6, it caused a lot of confusion. Below are before and after pictures of edge splits and flips on a bean mesh.



No edge splits



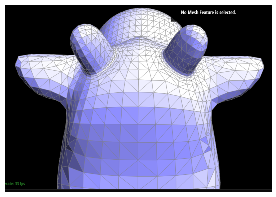
After some edge splits

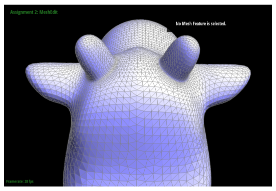


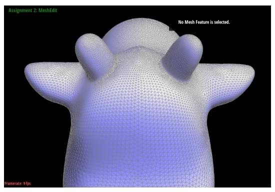
After some edge

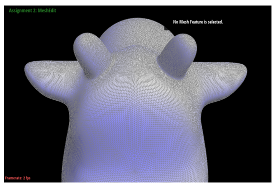
Part 6: Loop Subdivision for Mesh Upsampling

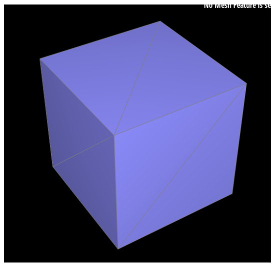
As recommended in the project spec, I began by preprocessing the weighted vertex locations for both the old and new vertices in the target mesh. For the old vertices, I stored the new locations in the newPosition field of the vertices themselves. However, storing the locations of the new vertices was a bit nuanced as they had not yet been created. Ultimately, I stored them in the newPosition field of the old mesh edges. In this way they would still be accessible after edge-splits and would be incident to the new vertices. Subsequently, I looped through all the old edges of the mesh and divided them using the edge-split operation. After toiling for some time with infinte loops caused by appending new edges to the mesh, I finally got the proper conditions set for splitting an edge (i.e. asserting that none of the vertices at the end of the edge were new). It then took me some time to convince myself that 4-1 triangle subdivision could indeed be achieved by carefully selecting which edges to flip after splitting every edge in the mesh. Once understanding the conditions for flipping an edge, I was able to update my edge-split subroutine to only set the isNew field to true for the new edges that bisected the original edge. Finally, I repositioned each vertex using the preprocessed locations identified at the beginning. Loop subdivision involves the relocation of vertices to achieve a more uniformly distributed mesh, causing once sharp edges to adopt a rounded appearance. These edges, akin to high-frequency 3D features within the mesh, lose their distinctiveness as the mesh's uniformity is enforced. Consequently, loop subdivision blunts sharp edges (as shown in the first row). In the case of the asymmetric cubes (as shown in the second row), they become symmetric if I pre-split some edges so that the mesh itself is all symmetrical (as shown in the third row). You can effectively call the original cube "undersampled," which is not uniform. Thus loop subdivision causes it to be asymmetrical.

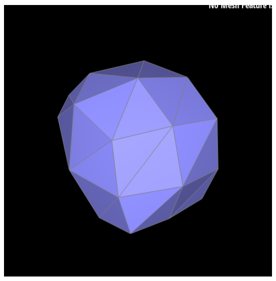

Cow: no upsample

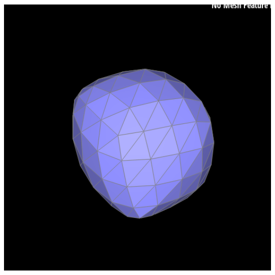

Cow: 1 upsample

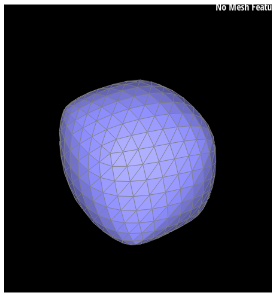

Cow: 2 upsamples

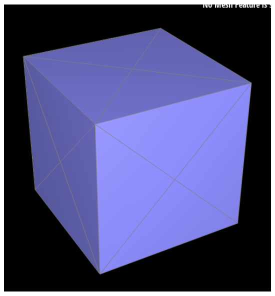

Cow: 3 upsamples

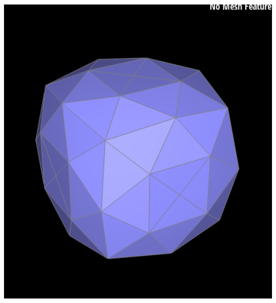

Asymmetric cube: no upsample

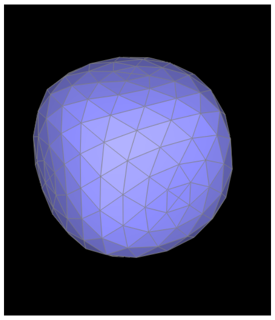

Asymmetric cube: 1 upsample

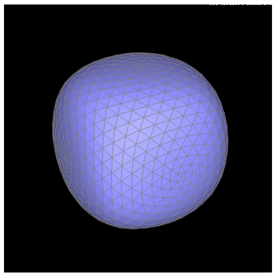

Asymmetric cube: 2 upsamples


Asymmetric cube: 3 upsamples


Symmetric cube: no upsample


Symmetric cube: 1 upsample


Symmetric cube: 2 upsamples


Symmetric cube: 3 upsamples