CS184/284A Spring 2025 Homework 2 Write-Up

Names: Tamnhi Vu and Samuel German

Link to webpage:
https://cal-cs184-student.github.io/hw-webpages-einstein-write-ups/hw2/index.html

Link to GitHub repository:
https://github.com/cal-cs184-student/sp25-hw2-einstein2

I'm a little teapot, short and stout!

Overview

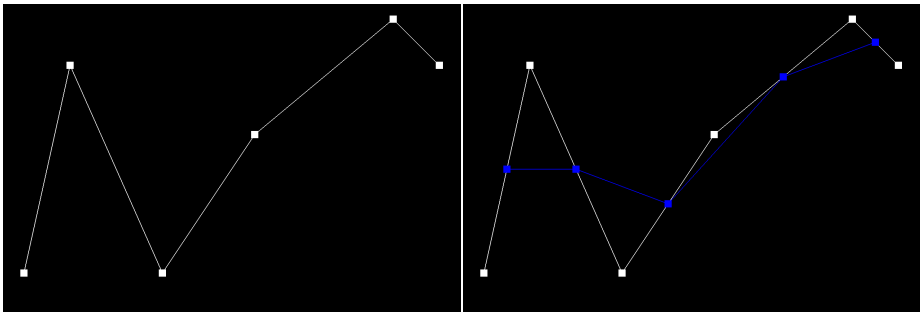
This project was really cool since it allowed to smooth out images using the Castelijau algorithm to the point(s) on Bezier curves and surfaces. We were also allowed to manipulate the half edge data structures which allowed us to create transition shading that would create smoother images. We did this by considering the area weighted normals, flipping edges, splitting edges, and upsampling. This was interesting to us since before this project, we were unaware of how these ideas could help us perceive an image and make it look smoother.

Section I: Bezier Curves and Surfaces

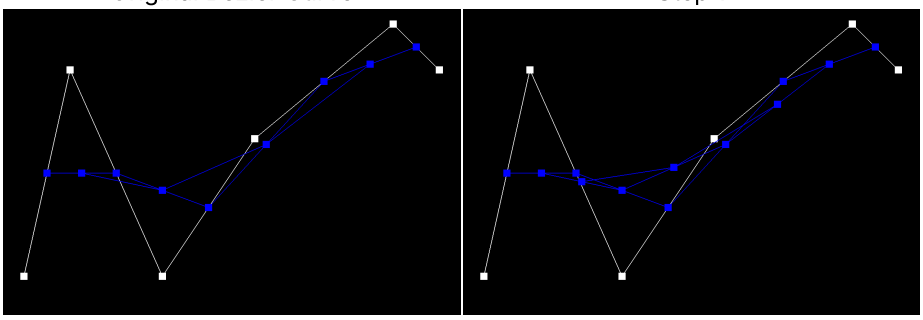
Part 1: Bezier curves with 1D de Casteljau subdivision

The Casteljau algorithm is a recursive method that allows me to implement the Bezier's curve. At each iteration of the Castelijau, I implemented it such that it would make a call to do one level/step of the Casteljau algorithm to give me the new control points by doing linear interpolation between the two current control points at each step. When we are given n control points, we can create a new vector of points due to linear interpolation. When our recursive process reaches our final remaining point, we can finally return the point on the completed Bezier curve.

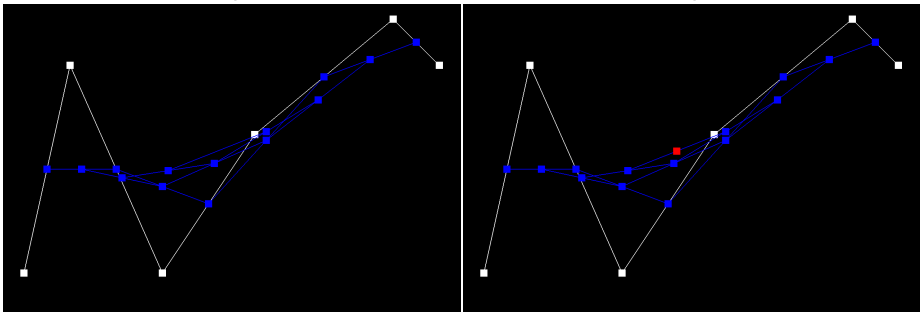
Here is an example with **6** control points:



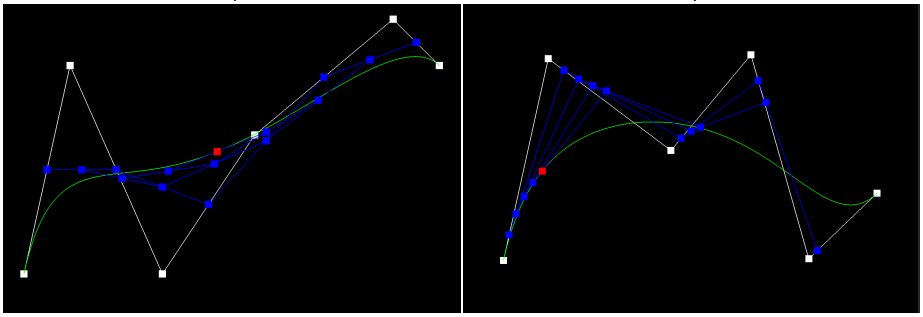
Original Bezier Curve | Step 1



Step 2 | Step 3



Step 4 | Step 5

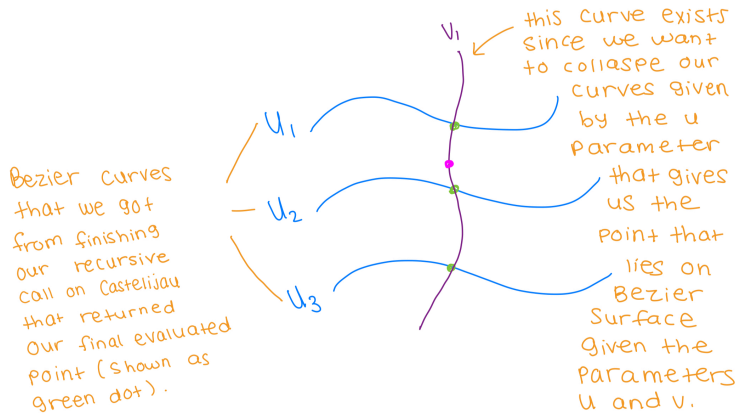


Completed Curve | Modified Bezier Curve

Part 2: Bezier surfaces with separable 1D de Casteljau

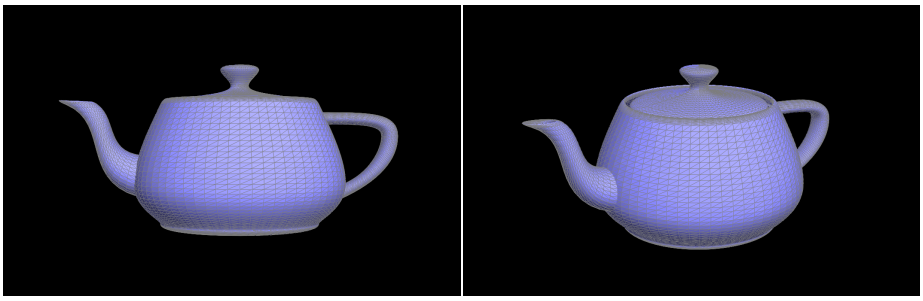
The Casteljau algorithm extends to Bezier surfaces by just applying the same recursive call, but on the parameters u and v. I implemented it in order to evaluate the Bezier surfaces by combining all the horizontal points along each of the rows along the parameter u which gives me a vector of intermediate points and finally interpolating one more time vertically along the v parameter to get the final individual point. By evaluating and interpolating the points until we only have one point remaining, this will give us the point on the Bezier curve given the parameter u and v.

Here is a visual explanation:



P.O.V. 1

Here is an example of a Bezier surface:



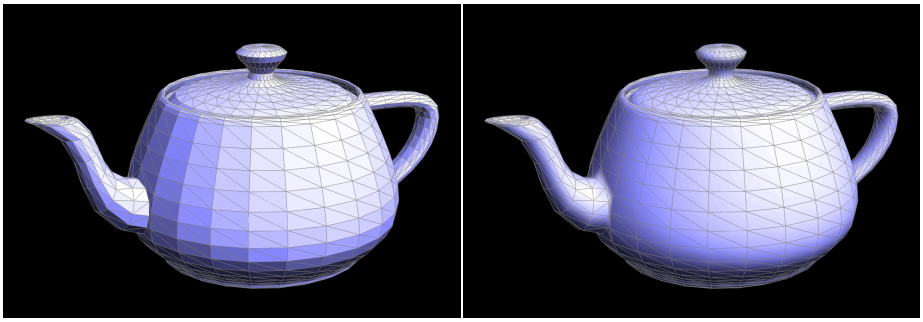
P.O.V. 1 P.O.V. 2

Section II: Triangle Meshes and Half-Edge Data Structure

Part 3: Area-weighted vertex normals

I was able to implement the area-weighted vertex normals by getting the half edges of a triangle by utilizing properties of a half edge and looping until I've grabbed all edges of a triangle. This allowed me to iterate through all of the faces of the object by repeatedly grabbing the neighbors positions . I was able to store the 3 vertex positions for every face and applied the cross product on the perpendicular edges to get the face normal where I then scaled this by the triangle's area. I was able to calculate the weighted normals sum and normalized the final result which allowed me to get the shading of the images to appear smooth.

Here is an example of area-weighted vetex normals to a teapot:



No shading Shading

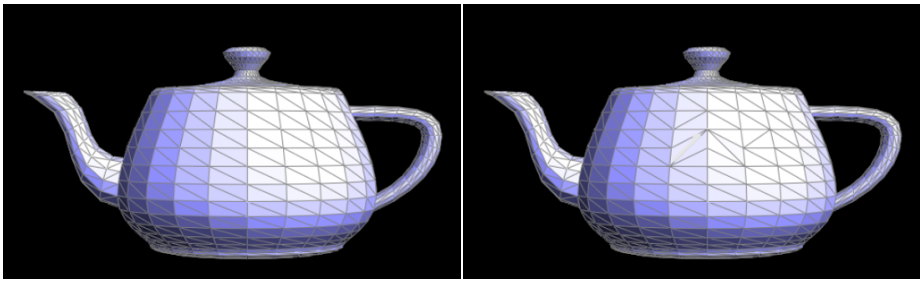
Part 4: Edge flip

I implemented edge flipping by, first of all, checking if an edge is a boundary edge, in which case I'd return instantly. Then, the next attributes

of the half edges are changed so that they cycle through the correct 2 triangles corresponding to the edge being flipped, rather than the 2 triangles that they are currently. Then, the vertices of each halfedge are reassigned to reflect the fact that from a single vertex, we now point in a completely different direction and thus correspond to a different half edge. The pointers to faces are also redefined for each halfedge, reflecting the fact that the faces are now different and thus involve different sets of half edges.

When I first implemented part 4, I didn't at all update the faces and the vertices, which had caused holes to form. I didn't see what was wrong at first, so I showed chat-GPT my code as well as the instructions for part 4 and asked what was wrong. Chat-GPT produced an updated snippet which did indeed perform updates to the faces and vertices. I learned that what the face and vertex attributes of a half edge are is very important, and that these notions are an integral part of the algorithms used to process halfedges in order to display them.

Here is an example of flipping a few edges near the center of a teapot:



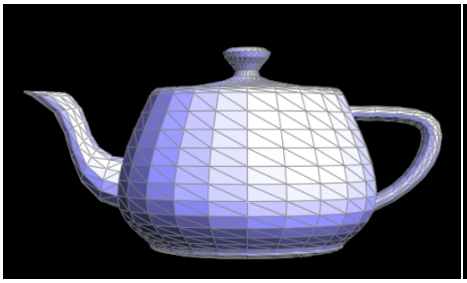
Before Flips | After Flips

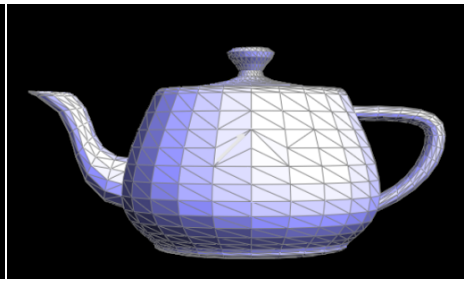
Part 5: Edge split

In the implementation, first of all we return instantly if either of the faces on either side of the edge are boundary. Then, we split the edge to be split into 2 half edges, and insert a new vertex at its midpoint, which is computed. We reuse the first half of the edge going from one endpoint to the center, and create half edges for the other 3 sets of edges, and set pointers appropriately so that these edges go from the midpoint to the other 3 vertices of the quadrilateral all thes edges were in. Then, we create two new faces, where the half edges of the original 2 faces have been changed to refer to the two other triangles out of the four in the desired split image; we then modify the half edges of the new faces to point to the appropriate cotners such that these 2 faces constitute the other 2 faces in the desired split edge figure, to make a total of four faces therein.

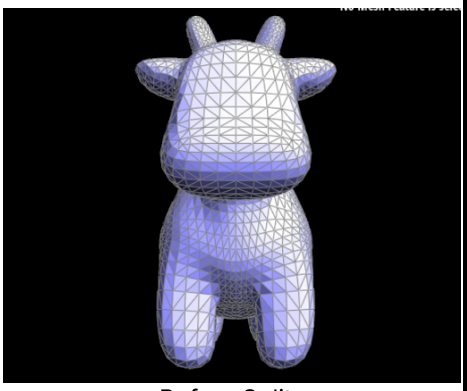
When I first implemented task 5, edges were being split, but I was getting a segfault after splitting a good number of them. I laboriously checked whether pointers were being set in a sound manner at first, by drawing out what happens when my implementation was ran on sample input; then I asked chat-GPT what could be wrong, as well as asked for how it would approach the problem. I then realized that I was doing this very inefficiently – I was creating 4 new faces from scratch, as well as four old faces from scratch. I TA with whom I brought up this issue said maybe that having the old half edges lying around could lead to problems, as I wasn't using them. In any case, Chat-GPT showed me an approach where we reuse the edges and faces as described above – we just modify the pointers of what's left over so that we can use them, as objects, in the image of a split edge. I learned that we can very often reuse old features of an image when changing it, and then failure to do so is inefficient and could lead to issues.

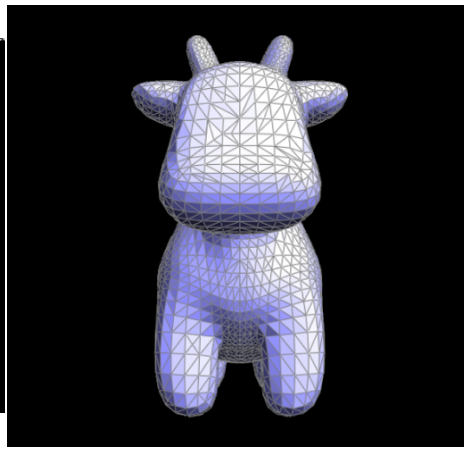
Here is an example of splitting a few edges near the center of a teapot:


Before Splits


After Splits

Here is an example of both splitting and flipping a few edges on a drawing of a cow:


Before Splits


After Splits

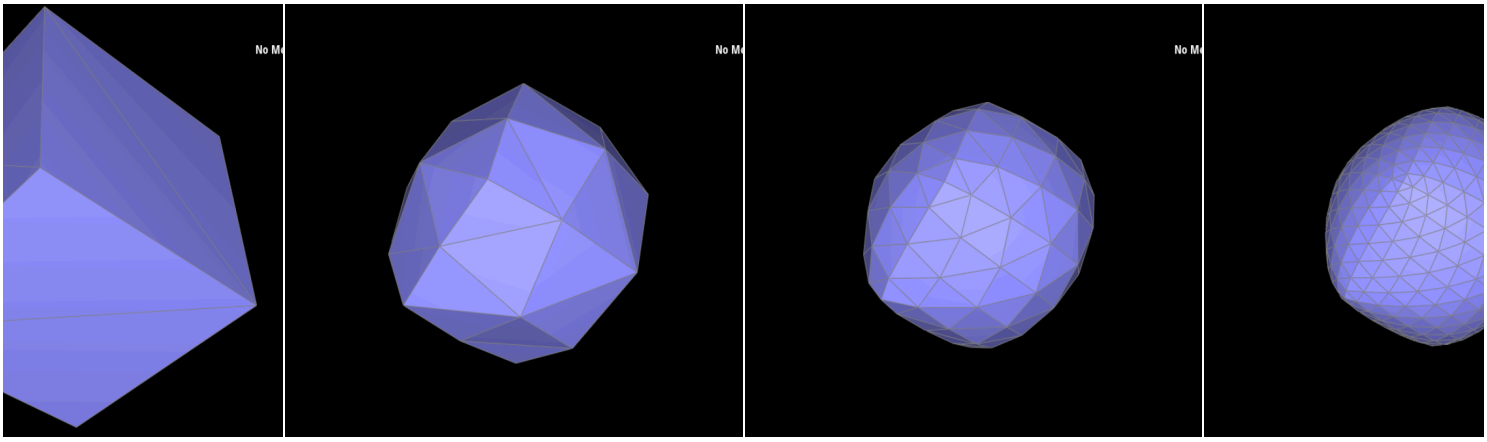
Part 6: Loop subdivision for mesh upsampling

The first thing the implementation does is compute new positions for the vertices given the formula described in the spec. Then, the connectivity between these vertices is refined into flipping the diagonals of quadrilaterals ensuing from splitting the diagonal of triangles by first of all discriminating between old and new vetrices (where old vertices are those present in the figure prior to the last subdivision and flipping, and new ones were the ones formed by these processes). Then, in order to perform one iteration, we split every edge in the mesh, and flip every edge connecting an old vertex to a new vertex, where the positions of new vertices formed hereby is determined by 3/8 * (A + B) + 1/8 * (C + D) . Finally, once the conectivity between elements of the mesh is decided, the coordinates of the new vertices are set to be the new positions we calculated for them earlier.

When I was first trying to implement part 6, I was stuck, so I asked chat-GPT for help by showing it the instructions, but what it produced had a massive error: pressing L once resulted in a highly asymmetrical figure, and pressing L again did nothing to the figure's appearance. To debug at this point, I commented out the part that assigns the new vertices to their positions, so that I could see the connectivity part in isolation. I saw that the connectivity part was highly asymmetrical and not at all what it should be. Finally, using the suggestion from a TA, I changed part 5 to manually change whether half edges it adds are new; I then deleted massive amounts of chat-GPT's code, most importantly the part that tried to do some sort of complex computations to account for whether a vertex was old or new; and then the code worked! I also used chat-GPT to ask it conceptual questions about this task, such as "how do the changes done to vertices, half edges when performing flips and splits compare to the changes when computing coordinates for them, inasmuch as which of their features are changed?" The answer chat-GPT gave to me on this question in particular helped me understand that the flips and splits were about the connectivity of the vertices, rather than their actual positions, like the coordinates are.

The below are some photos with what is originally a corner front and center. Sharp corners and edges turn into relatively asymmetrical structures: e,g, the midpoint of this quadrilateral was a corner, and its structures, and htose in its viccinity, are asymmetrical.

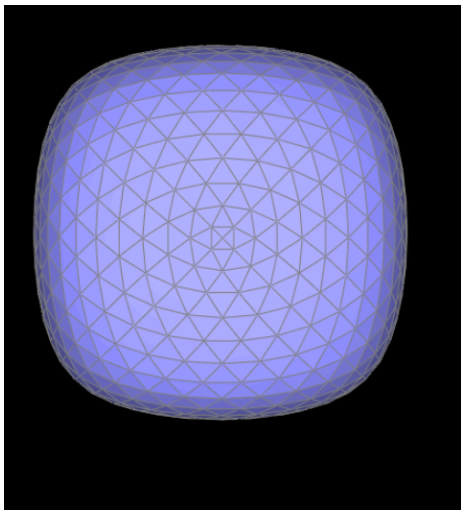
Progression of Loop Subdivision Algorithm



Start Iteration 1 Iteration 2 Iteration 3

By splitting, for every face, the edge which constitutes its diagonal, this assymetry can be handled. This error is occuring because a corner to this cube starts out with bordering one of its sides, and then having neigbors in a more concentrated area of it (namely, its front); not having its neighbors be uniformly distriuted as such, the sum of its neighbor's values included in the formula to compute a vertex's new position will be skewed in one direction. Now after we've split more edges and created more vertices, the specific parts where we've created vertices will be highly symmetrical, as the vertex lies on a smooth part of the cube to start out with, so its neighbors are more uniformly distributed around it. Then, as a result of more parts of the cube being smooth, the neighbors of the corner are more uniformly sampled about it than they would be had we not split edges, thus making the part of the cube corresponsding to where the corner formerly was, after the loop division algorithm has been applied many times, more symmetric due to its neighbors being more uniformly sampled around it. Here is an example after I've split edges as described.

Outcome after Prepocessing by Splitting Every Diagonal of a Side



several iterations in after preporocessing splits

I used chat-GPT for conceptual help on understanding why the part of the cube the corner was on at first becomes more assymetrical -- its answer wasn't fully satisfying to me, but gave me a clue of the real reason. It said that because the corner is on a boundary, its neighbors aren't uniformly distributed. This was a vague answer in my view, as saying the corner "is

on a boundary" isn't well defined; it isn't a boundary edge, and all the points on the surface are biased in one direction due to having the other points be only on one side of it. However, Chat-GPT's comment allowed me to see that the neihbors of the corner are nevertheless in an extremely concentrated area of the part of the mesh they are on, which makes the sum of the neighbors even more biased than if the corner was just any other point -- that is, a far smaller portion of the radius around the corner describes where all of its surrounding parts are than another non-corner point on the mesh.