Project 2

Isabel Li
https://ilyues.github.io/cs184-webpages/proj2/index.html

Overview

In this assignment, I explored various geometric modelling concepts to produce a mesh editor. I began by learning how to build Bezier curves using the De Casteljau algorithm, learning the basis of how the control points correspond with the produce curve and creating a curve editor within the program. I was able to reinforce my understanding of how the algorithm extends to building Bezier surfaces. Equipped with the ability to build surfaces, I then learnt and implemented different methods of manipulating the surface "mesh": applying Phong shading through area-weighted vertex normals, flipping and splitting edges through pointer reassignments, as well as mesh upsampling through loop subdivision. Overall, I was able to build a program which could load and manipulate three dimensional models.

Reflecting on my implementation and process, I appreciated how completing these tasks gave me a greater understanding of how to compute the values needed for these mesh manipulations. I encountered problems to do with float division, pointer reassignment, and incorrectly splitting/flipping edges for mesh upsampling. I was able to overcome some of these problems by using the debugger in Xcode and setting appropriate breakpoints, especially to identify where the program ran into infinite loops or discover errors in float division. Some of the other issues I was able to resolve through inspection of the output (such as the cube for Part 6) to recognise that edges were not being split properly, and focusing my debugging efforts on that part of the method.

Part 1: Bezier Curves with 1D de Casteljau Subdivision

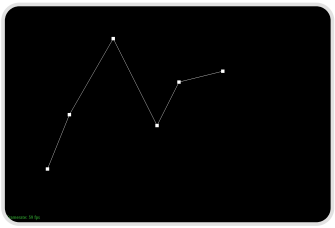
I implemented De Casteljau's algorithm to build a Bezier curve given an input of control points. De

Casteljau's algorithm is a recursive method used to evaluate Bezier curves. Through repeated linear interpolation, the algorithm allows us to find a point which lies on the Bezier curve at a given parameter t, where t is between 0 and 1.

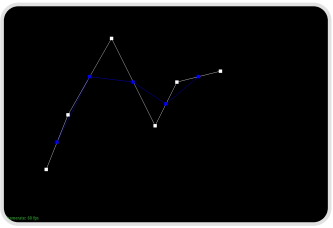
Given n control points p_1, ..., p_n and the parameter t, De Casteljau's algorithm uses linear interpolation to compute n - 1 intermediate control points at t in the next subdivision level. These are p'_1, ..., p'_{n-1}, where $p'_i = (1-t)\ p_i + tp_{i+1}$.

This is applied n - 1 times on the previous subdivision level until producing a final singular point which lies on the Bezier curve at t.

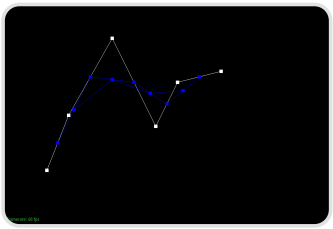
In this part, I implemented one level of the algorithm (one level of subdivision) in a function which took in an array of control points and returned an array of control points at the next subdivision level. I iterated through the argument array and applied the lerp function for p'_i described above, before pushing this to an array of the new control points.



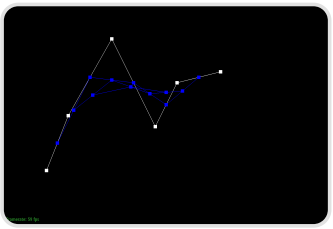
Unevaluated curve



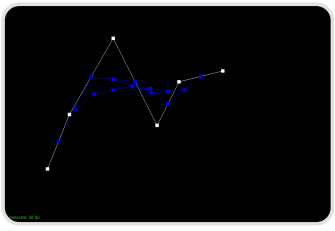
Step 1 of evaluation



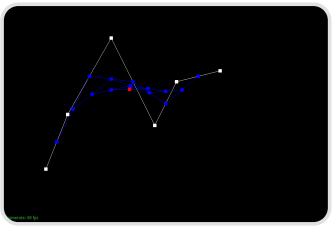
Step 2 of evaluation



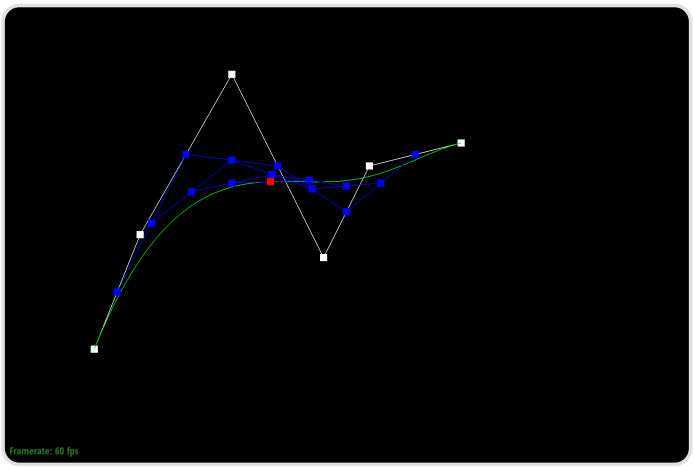
Step 3 of evaluation



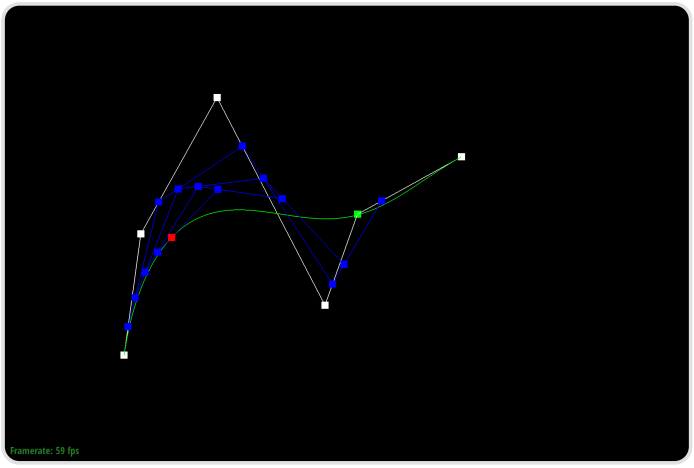
Step 4 of evaluation



(Final) step 5 of evaluation



Completed Bezier curve



Slightly modified Bezier curve with moved control points and modified t parameter.

Part 2: Bezier Surfaces with Separable 1D de Casteljau

I used the De Casteljau algorithm to build Bezier surfaces given a $n * n$ grid of input control points. The De Casteljau algorithm extends to Bezier surfaces through the separable 1D De Casteljau algorithm. The algorithm described above is applied to a $n * n$ grid of original control points, with parameters u and v.

Each row of n control points defines a Bezier curve parameterised by u; the respective final, single point P_i is evaluated recursively for each of n rows using the algorithm in part 1, with u as the argument t.

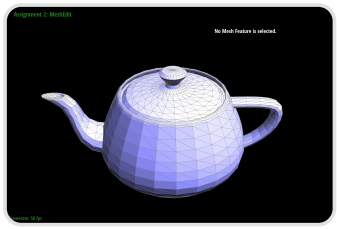
The n P_i's for each of the n rows then define a Bezier curve with parameter v. This is also then evaluated recursively with De Casteljau's to produce a final singular point which lies on the Bezier surface for given parameters u and v.

Screenshot of `bez/teapot.bez` evaluated by my implementation

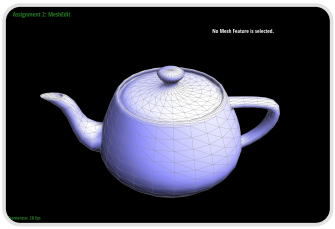
Part 3: Area-Weighted Vertex Normals

To produce Phong shading as opposed to flat shading, I calculated area-weighted normal vertices. I implemented area-weighted normal vectors at a vertex by iterating through neighbouring vertices, storing two vertex positions *A* and *B* at a time, in addition to the original vertex position *C*. Using these three vertices, I found vectors *C* - *A* and *C* - *B*, and used their cross product to calculate the normal to the face.

I then found the area-weighted normal by dividing this cross product by 0.5 * its magnitude, and pushed the area-weighted normal to an array of all weighted normals. I repeated this process for each face incident to the vertex. To find the final vector normal, I summed the weighted normals within the array, and divided this summed vector by its own magnitude.



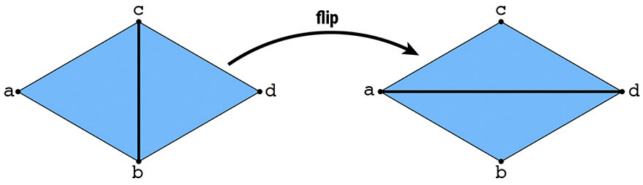
Screenshot of `dae/teapot.dae` without vertex normals (default flat shading)



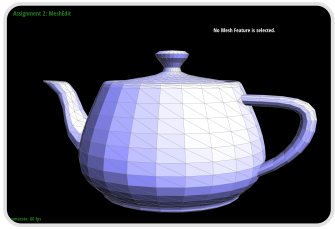
Screenshot of `dae/teapot.dae` with vertex normals (Phong shading)

Part 4: Edge Flip

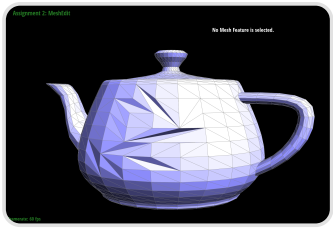
Edge flipping is one method of mesh manipulation which converts a pair of triangles into a new pair, with the edge between them rotated (or "flipped") in orientation. To implement edge flipping, I initiated pointers to every half-edge, face, and vertex. I then reassigned all the half-edges' next, twin, vertex, edge, and face pointers according to the flipped result -- I kept the vertices and halfedges on the sides of the external "diamond" in the same position, but re-assigned the pointer to edge (*b*, *c*) to the vertices *a* and *d*. The faces (*a*, *b*, *c*) and (*b*, *c*, *d*) then became (*a*, *c*, *d*) and (*a*, *b*, *d*) respectively, and I reassigned the corresponding halfedges based on any face changes. I implemented a check at the beginning of the function to return if the element is at a boundary.



Regarding debugging, my implementation was close to correct on my initial attempt. My "debugging" process mostly consisted of re-reading the assignments for each mesh element, and then catching and correcting some assignments which didn't make sense to me upon the second read.



Screenshot of `dae/teapot.dae` before edge flips



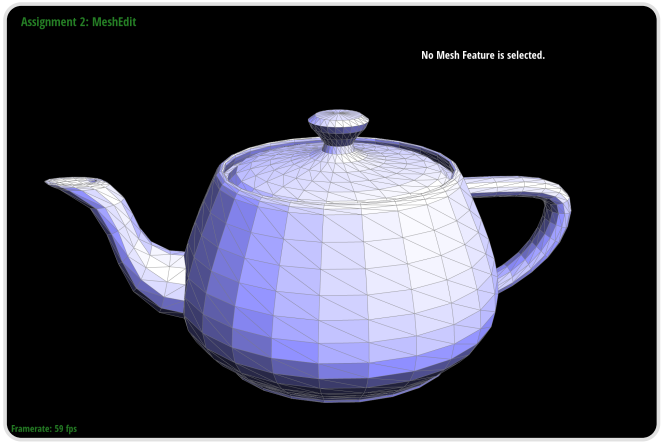
Screenshot of `dae/teapot.dae` after some edge flips

Part 5: Edge Split

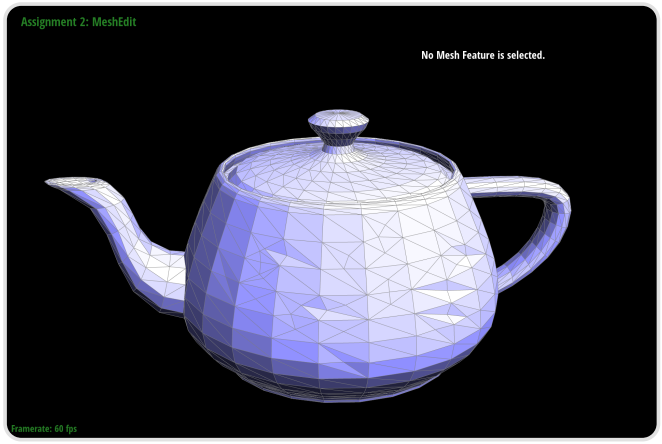
I implemented edge splitting, another local remeshing operation which takes a pair of triangles separated by an edge and inserts a new vertex at the centre of the edge, splitting each of the two triangles in half with two additional edges (resulting in four triangles). For this part, my implementation process was very similar to my implementation of edge flips. I initiated pointers to every relevant mesh element (vertices, faces, halfedges) and re-assigned all their pointers according to the "after split" diagram, including to newly created mesh elements. I created the appropriate new halfedges, edges, faces, and vertex,

and assigned all their pointers. Like Part 4, I "re-purposed" the original edge (*b*, *c*), this time to become (*b*, *m*), and faces (*a*, *b*, *c*) and (*b*, *c*, *d*) became (*a*, *b*, *m*) and (*b*, *d*, *m*) respectively. For the position of *m*, I added the positions of *a*, *b*, *c*, and *d*, and then divided this sum by 4.

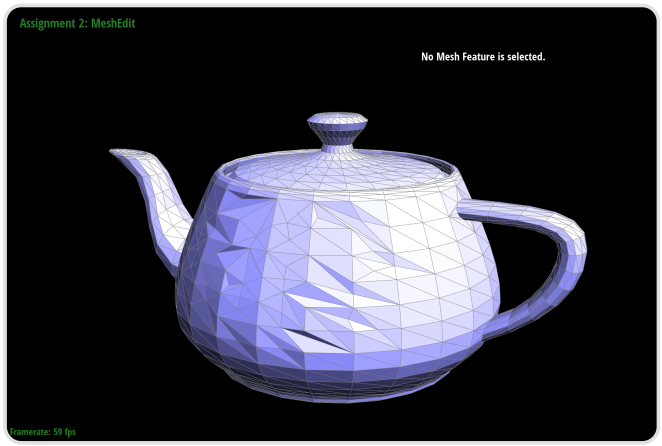
Errors I encountered here were also related to incorrect pointer assignments in my original attempt, usually to do with assigning to the correct new face and keeping in mind the anti-clockwise direction of the half-edges, since I didn't redraw the diagram. I debugged this again mostly through carefully re-reading my pointer assignments.



Screenshot of `dae/teapot.dae` before edge splits/flips



Screenshot of `dae/teapot.dae` with some edge splits



Screenshot of `dae/teapot.dae` with some edge splits and flips

Part 6: Loop Subdivision for Mesh Upsampling

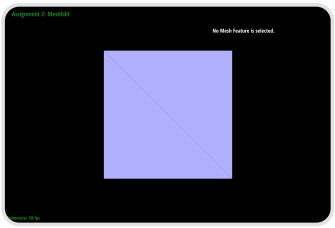
I used loop subdivision to process mesh upsampling. To implement loop subdivision, I followed the recommended solution of breaking up the method into computing new vertex positions for old vertices, computing new vertex positions associated with edges, splitting all edges, flipping new edges which connected an old and new vertex, and copying in the new vertex positions.

To begin, I computed new positions for all existing vertices in the input mesh using the equation `(1 - n * u) * original_position + u * original_neighbor_position_sum` where *n* is the number of edges incident to the vertex; and *u* is 3/16 if *n* = 3, or 3 / 8 * *n* otherwise; *original_position* is the original position of the old vertex, and *original_neighbor_position_sum* is the sum of all original positions of the neighboring vertices. I stored these new positions in each existing vertex's `newPosition` pointer, and also set the `isNew` property to false.

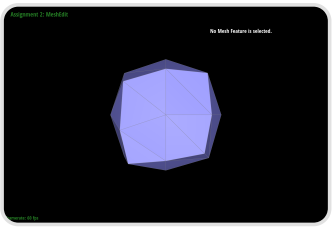
Similarly, I computed and new vertex positions and stored them in the `newPosition` property for the edges which would split to produce these new vertices. The position of a new vertex splitting the shared edge (*A*, *B*) between a pair of triangles (*A*, *C*, *B*) and (*A*, *B*, *D*) is `3/8 * (A + B) + 1/8 * (C + D)`. I also set these existing edges' `isNew` property to be false.

To split the appropriate edges, I split the edges which had no new vertices, and set the resultant vertex to be new along with assigning it its pre-computed new position. I also set the newly created edges to be true for `isNew`, to assist in flipping the correct edges.

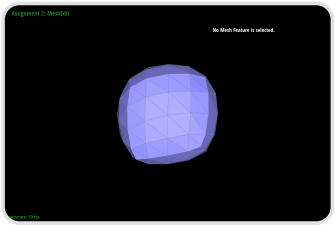
I then iterated again over all edges and flipped edges if they were new and had exactly one new vertex. Finally, I reassigned all old vertices to have their new position.



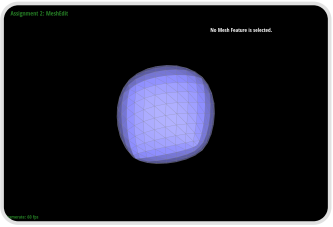
Screenshot of `dae/cube.dae` before upsampling



Screenshot of `dae/cube.dae` after upsampling once



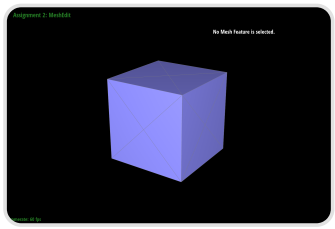
Screenshot of `dae/cube.dae` after upsampling twice



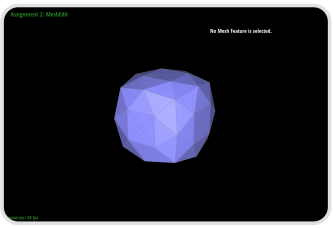
Screenshot of `dae/cube.dae` after upsampling three times

I noticed that after loop subdivision, the sharp corners and edges on meshes become more rounded, "curving" even if they were original straight lines. This is because they are pulled closer in towards the neighbouring vertices in position when upsampling -- for example, for a corner of the cube which directly protrudes towards a certain direction, every neighbouring vertex (other corners of the cube) is further away from that direction. Therefore, by weighting neighbouring vertices in the computation of this vertex's new position, it moves slightly away from the direction it pointed in and "rounds" out.

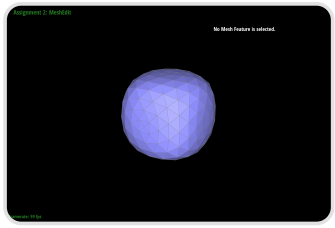
I also saw that after a few iterations of subdivision on the cube model, the mesh becomes slightly asymmetrical after repeated subdivisions. This is because the original mesh is asymmetrical, despite the symmetrical cube surface produced. Since each face of the cube is split diagonally but asymmetrically, some cube vertices have five neighbouring vertices, while some have four. This difference factors into the calculation of new vertex positions, leading to asymmetry in the new vertices and an overall asymmetry to the shape.



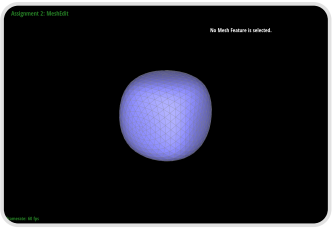
Screenshot of `dae/cube.dae` before upsampling, with edges split symmetrically on each face



Screenshot of `dae/cube.dae` after upsampling once, with edges split symmetrically on each face

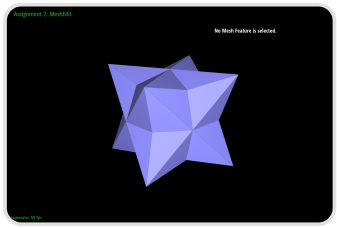


Screenshot of `dae/cube.dae` after upsampling twice, with edges split symmetrically on each face

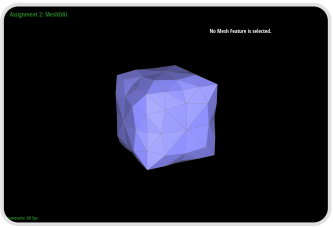


Screenshot of `dae/cube.dae` after upsampling three times, with edges split symmetrically on each face

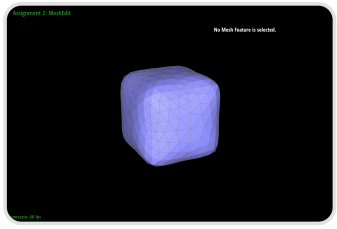
To subdivide the cube symmetrically, I pre-processed by splitting the edge of the mesh which cuts across each face of the cube so that the mesh for the cube was also symmetrical. This meant all the vertices had the same number of neighbouring vertices, and so the new vertex position calculations were relatively the same, producing a "perfect" rounded cube.



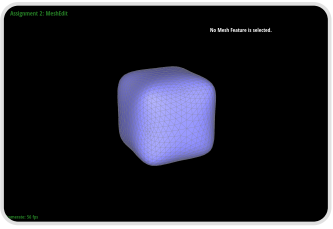
Screenshot of `dae/cube.dae` before upsampling, with "pre-processing" for sharper corners



Screenshot of `dae/cube.dae` after upsampling once, with "pre-processing" for sharper corners



Screenshot of `dae/cube.dae` after upsampling twice, with "pre-



Screenshot of `dae/cube.dae` after upsampling three times, with

processing" for sharper corners

"pre-processing" for sharper corners

I reduced the effect of the rounded corners by splitting the edges of the cube (first across each face to be symmetrical, then at each edge of the cube) so that the vertices original protrude more, mitigating the rounding effect when the vertex is "pulled back" by its neighbouring vertices for its new position.

https://ilyues.github.io/cs184-webpages/proj2/index.html