CS184/284A Summer 2025 Homework 2 Write-Up

Names: John Bragado, Kevin Hu

Link to webpage: john-bragado.dev/hw2/
Link to GitHub repository: github.com/cal-cs184/hw2-meshedit-triangles

Overview

For this assignment, we were tasked to implement bezier curves & surfaces, area-weighted vertex normal calculations, various edge operations with the halfedge data structure, and loop subdivision.

We tried to start as early as possible on this homework since we knew it would probably take a lot of time (especially dealing with halfedges and pointers), and we were right. Here's our thoughts on the implementation and debugging process:

Kevin: Personally, I really enjoy the part where we figure out how to traverse the half-edge data structure (though not so much the debugging). I used to be somewhat skeptical about whether it actually worked better than just storing pointers to vertices. But as we worked through the pointer assignments and learned how to flip and split edges, I became fascinated — and convinced — that it's a really clever and powerful structure.

John: This homework was super interesting to me and it was really satisfying seeing the bezier curves and loop subdivision actually work. Debugging was honestly a pain for the halfedge operations (tasks 4-6), but I learned a lot about using pointers and references in C++ (which was the cause of much of our pain debugging), but I am excited to learn more. I think being able to see the few small functions we wrote for bezier curves contribute to the rendering of such complex reminded me why graphics is so fascinating and makes me want to make my own graphics engine from scratch.

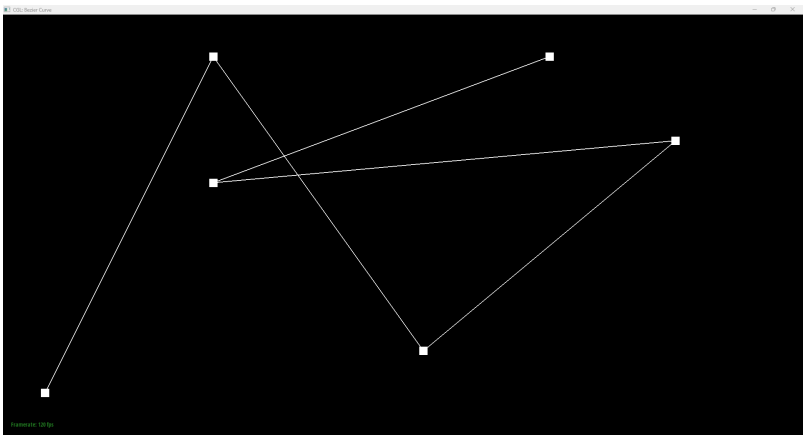
Section I: Bezier Curves and Surfaces

Part 1: Bezier curves with 1D de Casteljau subdivision

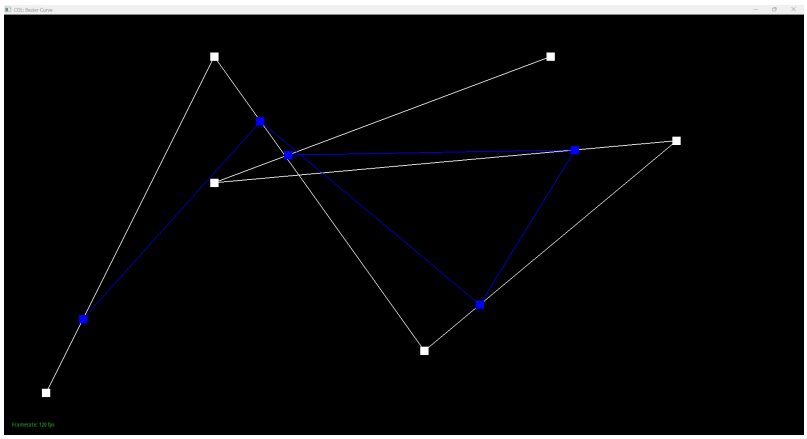
De Casteljau's algorithm is a recursive method for evaluating and creating Bezier curves. The way we implemented it was to first create a function `BezierCurve::evaluateStep(...)` that would take in a set of control points and a parameter `t`, and then output the "intermediate" points that would result from lerping* the points at `t`. By calling this function recursively, we were able to calculate the position of any intermediate control points as well as the position on the curve at any given value of `t`.

*We created separate helper functions to aid our calculations and simplify our code.

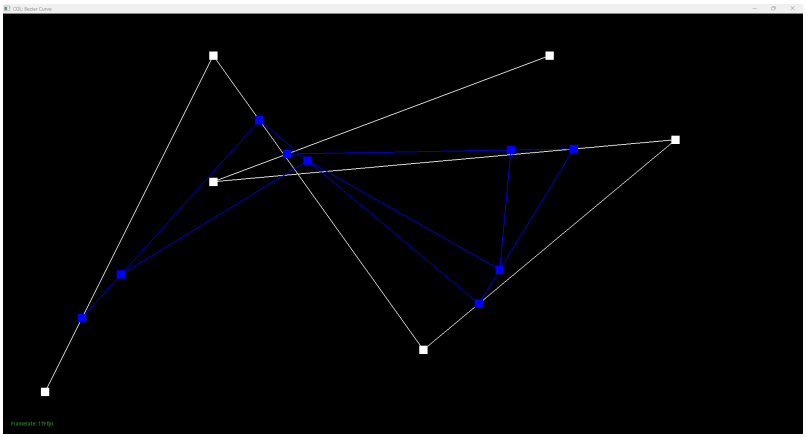
Below are screenshots taken at each recursive step of evaluating a Bezier curve with 6 points at an arbitrary value of `t`.



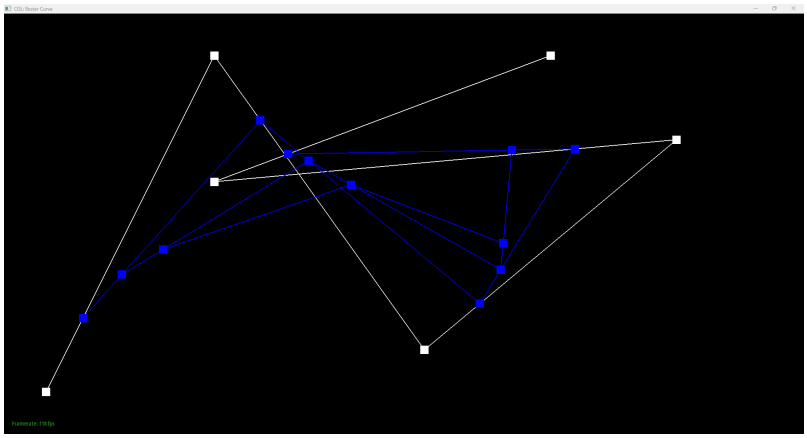
6 control points.



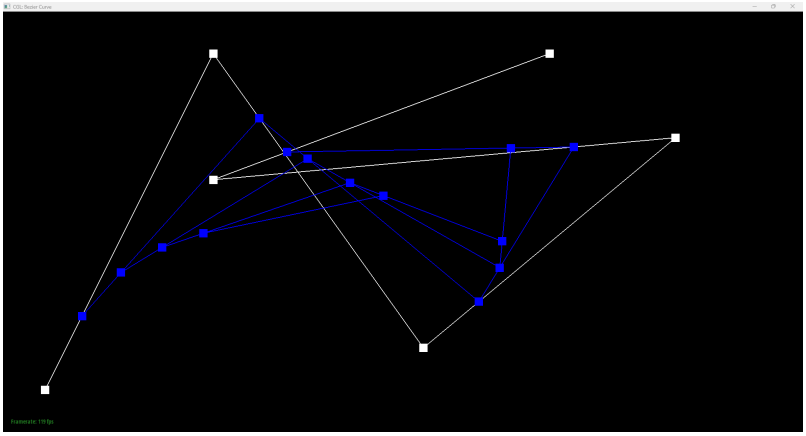
First Step: 5 intermediate points.



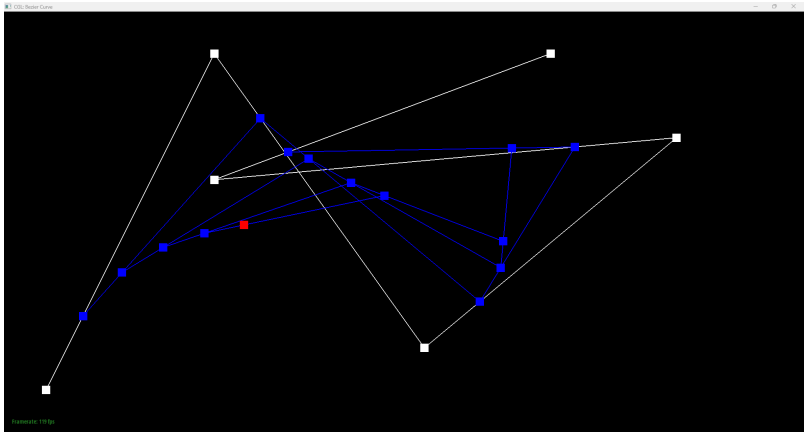
Second Step: 4 intermediate points



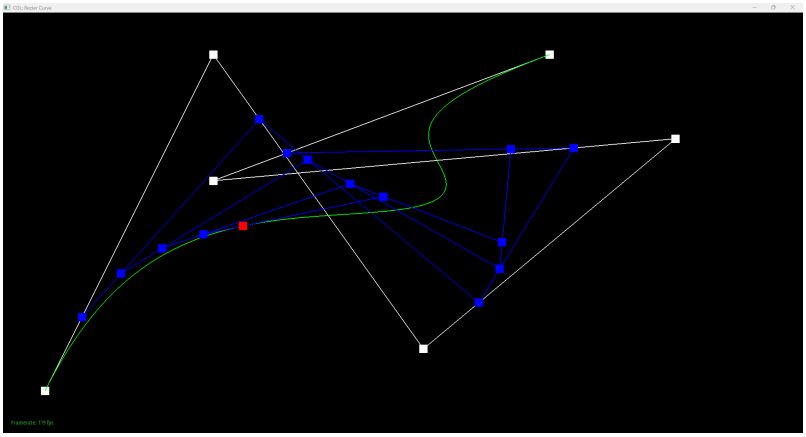
Third Step: 3 intermediate points.



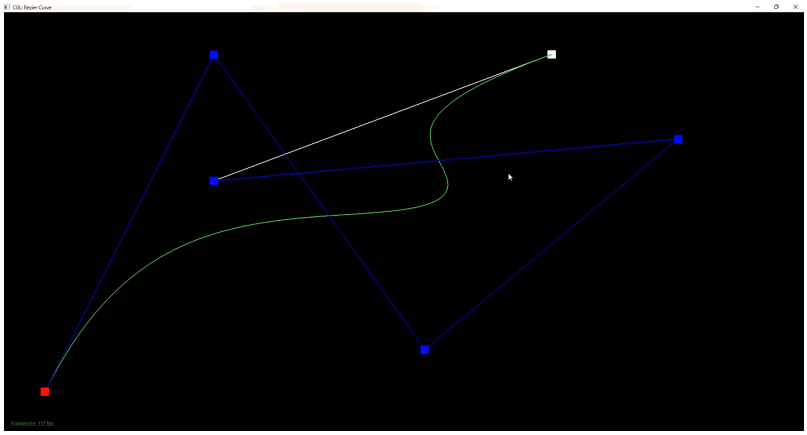
Fourth Step: 2 intermediate points.



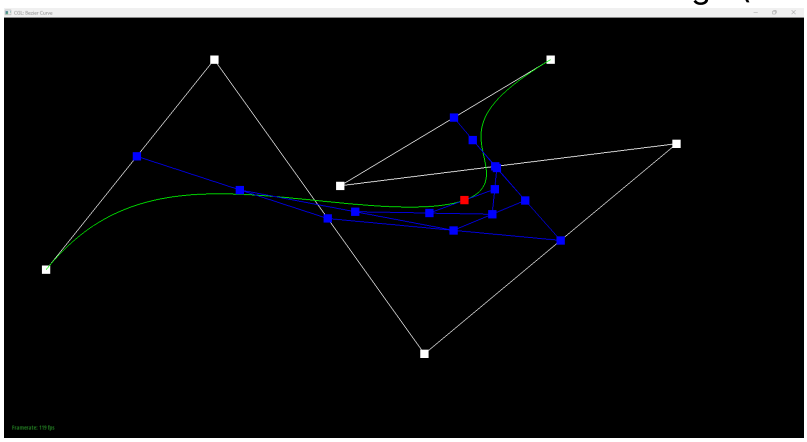
Fifth Step: 1 final point



The resulting Bezier Curve.



Increasing t (this is a looping GIF)



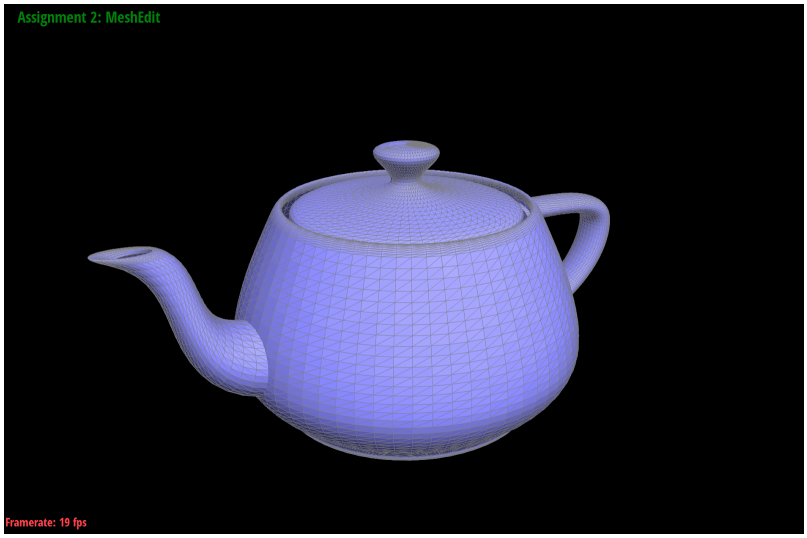
Another curve with points at slightly different points

Part 2: Bezier surfaces with separable 1D de Casteljau

Now that we can calculate vectors from one step of de Casteljau using `evaluateStep()`, we just need to run evaluateStep recursively until it only returns one `Vector3D`. We implement this in the function `evaluate1D()`.

The meshes we are working with exist in 3D space. So to evaluate a some n by m bezier patch (which is defined by a grid of control points that creates a 3D "surface"), we need to first call `evaluate1D()` on each of the n bezier curves going in one direction with some parameter u. Finally, we call `evaluate1D()` while passing in the n points from the previous step and another parameter v. All this will allow us calculate the surface's position at any parameters (u, v). This logic makes up our `evaluate()` function, which returns a single vector representing the value of the patch at (u, v)

Beneath the hood, the renderer may use the function to sample points from the curve and convert them to triangles to rasterize.



a teapot made up of bezier patches

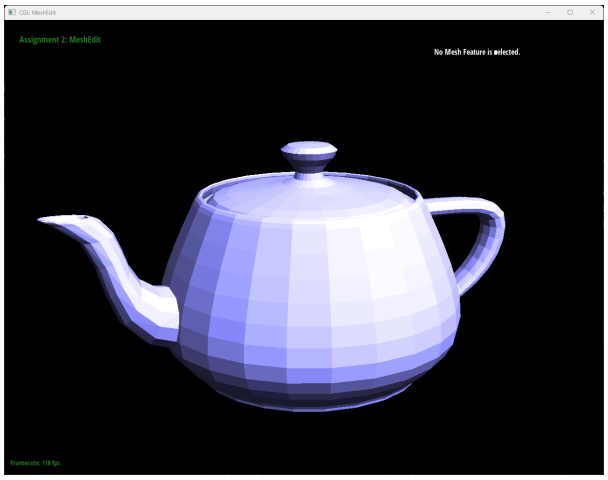
Section II: Triangle Meshes and Half-Edge Data Structure

Part 3: Area-weighted vertex normals

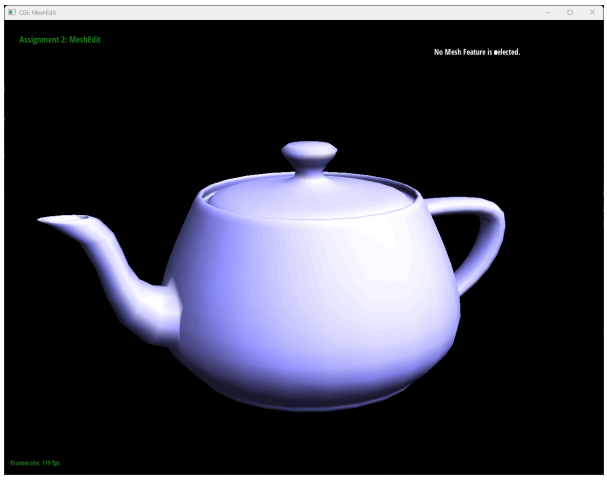
Our task was to create a function that would calculate the area-weighted normal of any given function. This requires finding the normals of all the faces that surround a vertex, then average each face by its normal to produce a vector normal for an approximation of the vertex.

To implement this, we created a while loop that would start with the halfedge pointed at by the vertex. Then we'd use next(), twin(), and vertex() pointers to find the positions of the vertices for each face. After summing up the cross products of their vectors (obtained via subtracting the positions of the vertices) we divided by the number of surrounding faces to average the normal vectors. We were able to avoid calculating the area of the faces since the cross product of vectors was already needed to calculate the normals.

Finally, we normalized the average weighted vector at the very end with calls to sqrt() and dot().



teapot.dae with flat shading



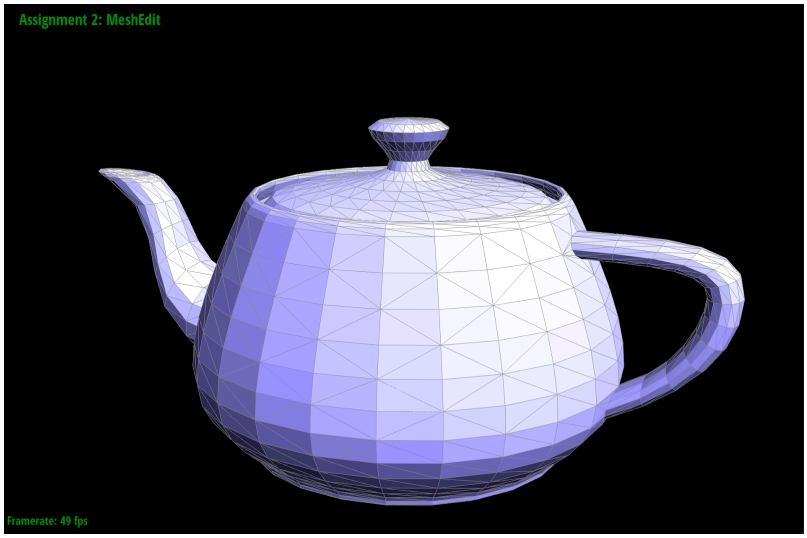
teapot.dae with Phong shading

Part 4: Edge flip

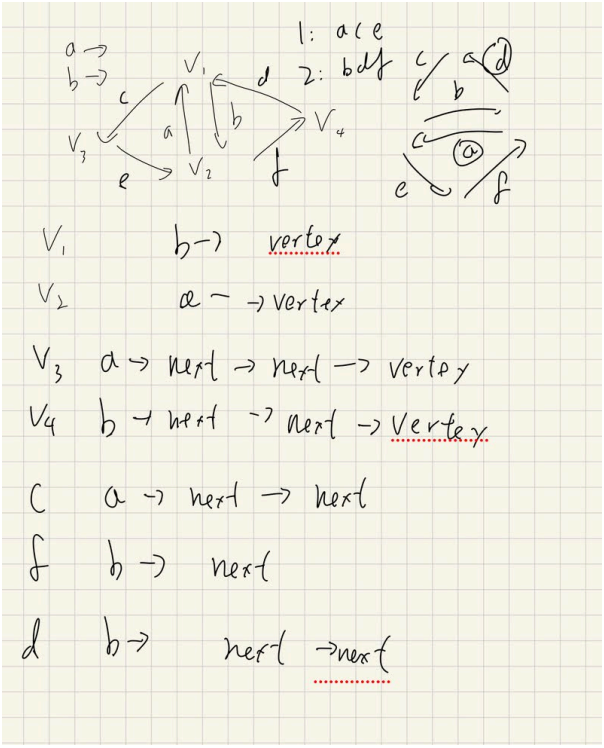
Below is our amazing, non-exhaustive list of pointer reassignments. We first created pointers for every item in two triangles (including vertices, halfedges, edges, and faces), and named them according to the convention in our diagram (using a as the initial halfedge). We then reallocated pointers, such as halfedge → next(), were pointing to accordingly.

As for debugging, our biggest hurdle was forgetting to reassign some pointers attributes like halfedge's vertex because neither of us can't really read that well (or didn't bother to read), and that puzzled us for a solid piece of time with spreading self doubt and distrust (to the computer and lecture slide not each other). But we figured it out.

original teapot.dae



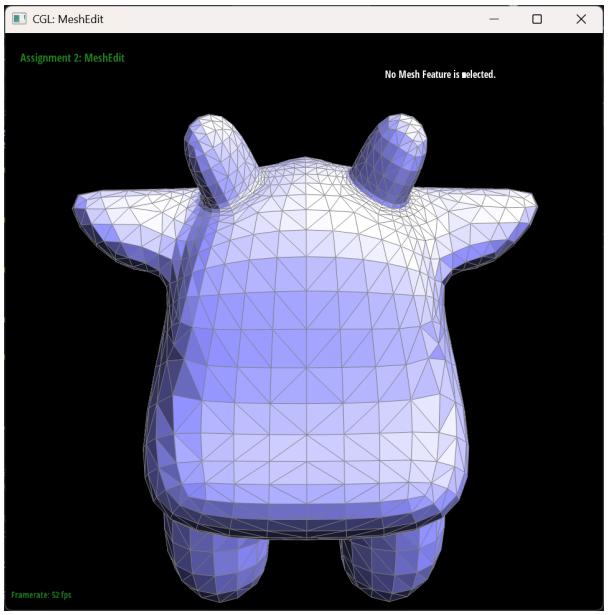
teapot.dae with some edge flips



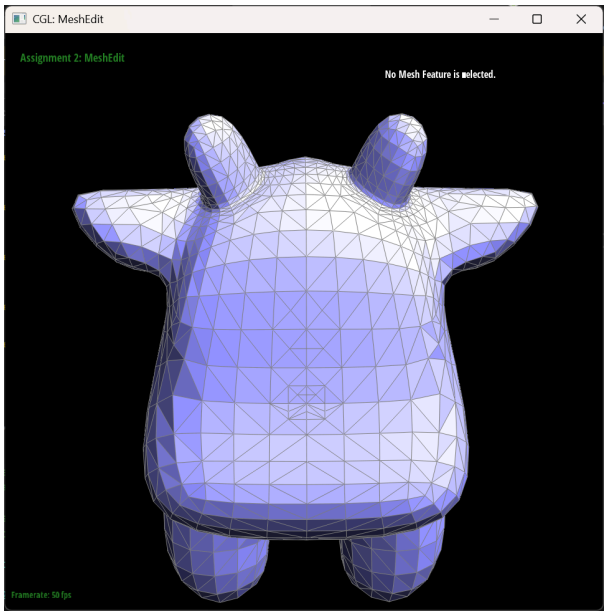
a diagram we drew to visualize flipping

Part 5: Edge split

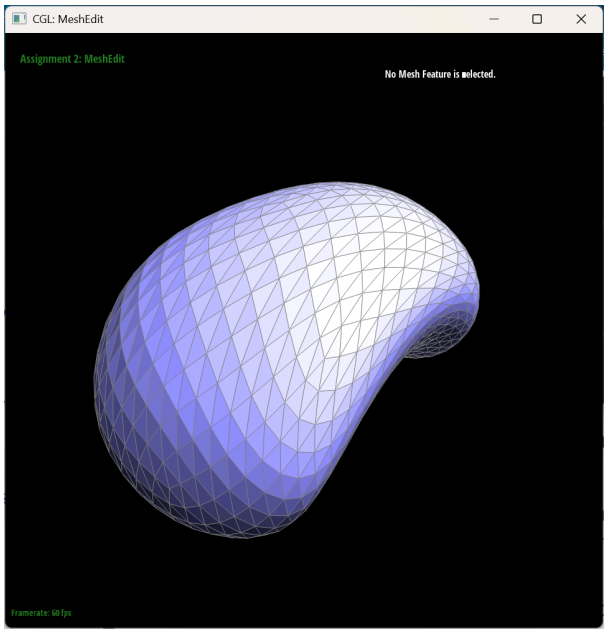
In task 5, we implemented the edge split operation. We first started by creating pointers for every halfedge, face, vertex, and edge for both initial triangles involved in the edge split. Then we calculated the position of the new "midpoint" (the vertex that bisects the original edge and which will connect all 4 faces after the edge split) by averaging the positions of both neighboring vertices. Then, we created new halfedge elements using the `newEdge()`, `newFace()`, and `newHalfedge()` constructors. Finally, we (after much contemplation and drawing diagrams) we assigned the pointers for all the new elements (most of which were the 6 new halfedges) and any elements that needed pointers reassigned.



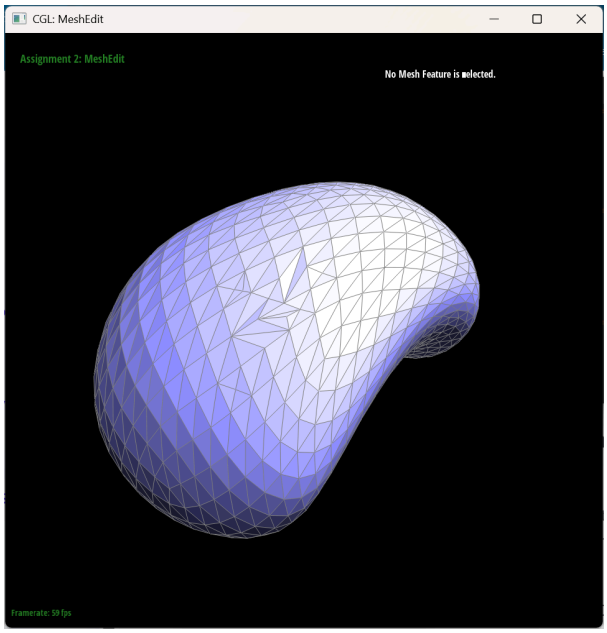
cow.dae



cow.dae with edge splits

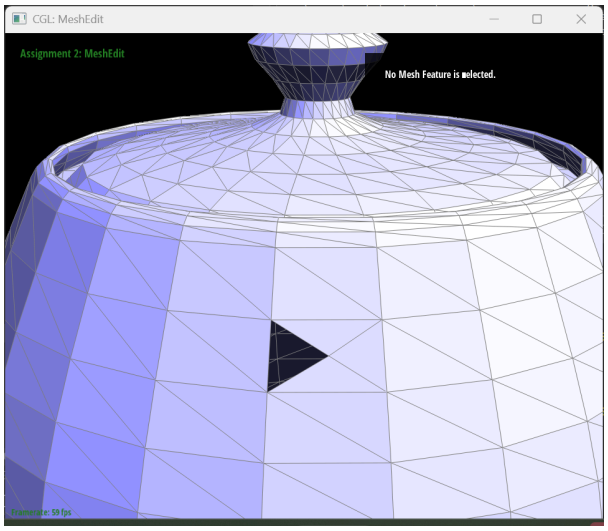


bean.dae

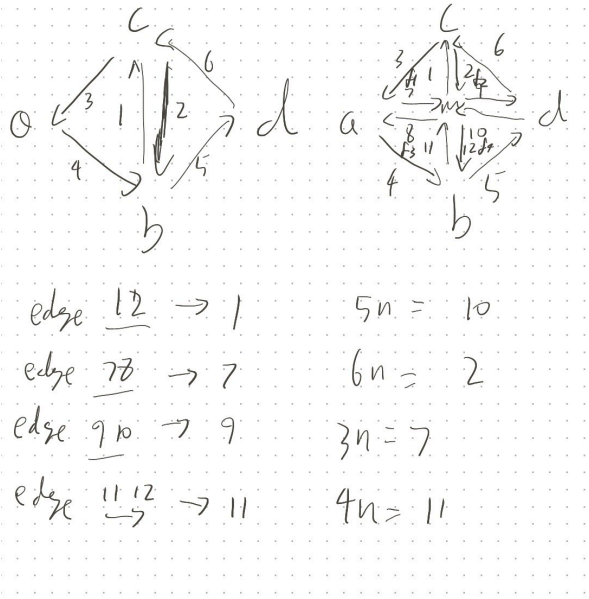


bean.dae with edge splits and flips

Out of all the tasks on this assignment, task 5 was the one we definitely spent the most time on due to the tedious debugging journey we went through. We breezed through the initial code pretty quickly at first. But when we tested it out on a mesh, for some reason 1 or 2 faces would just vanish into thin air (see screenshot below). We reviewed every line to make sure the pointer assignments were right, we used the `check_for()` function and printed things to the console to keep track of which elements were pointing to what, and we even wrote down a page full of memory addresses from the debugging info screen just to see which were right.



missing face after edge splitting



scribblings of a madman

```
205   VertexIter c = he2->vertex();
206   VertexIter a = he1->next()->next()->vertex();
207   VertexIter d = he2->next()->next()->vertex();
208
209   FaceIter f1 = he3->face();
210   FaceIter f2 = he6->face();
211   f1->halfedge() = he3; // adding in this line of code,
212   f2->halfedge() = he6; // (and this one) ended up fixing our bug!
213
214   // NEW FACES
215   FaceIter f3 = newFace();
216   f3->halfedge() = he4;
217   he4->face() = f3;
218   FaceIter f4 = newFace();
```

the two lines of code that fixed our bug

Part 6: Loop subdivision for mesh upsampling

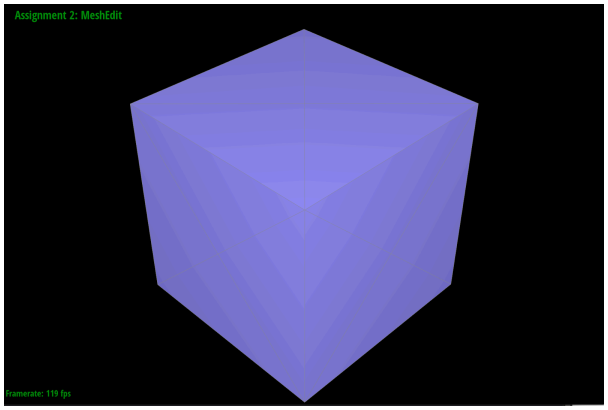
Our implementation of Loop Subdivision mostly follows the spec from HW2. We started by iterating through all the vertices and computing their updated positions using the `u` and `1 - n * u` ratios described in the spec. To find the neighboring vertices, we used a variant of the solution to Discussion Worksheet 5, Problem 1 but instead we accessed vertices instead of edges. Then we stored these new positions in each vertex's `newPosition`.

Next, we traversed all the edges and calculated their new midpoint positions using weights of `3/8` for the neighboring vertices and `1/8` for the opposite vertices, and stored the result in the edge's `newPosition`. We also store these edges in a `vector<EdgeIter>` to keep track of which edges existed before subdivision.

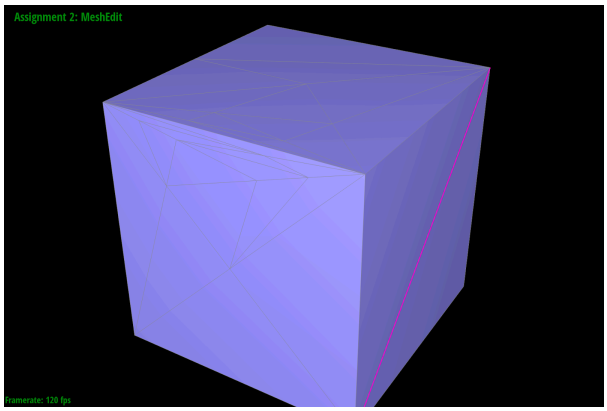
We then split all the original edges (from the stored vector) so that we wouldn't accidentally split any new ones, and also marked the new midpoints as new vertices.

After that, we traversed through all edges and flipped any edges that connected a new vertex to an old one before traversing through all vertices and edges one last time to mark all vertices and edges as `isNew = false`.

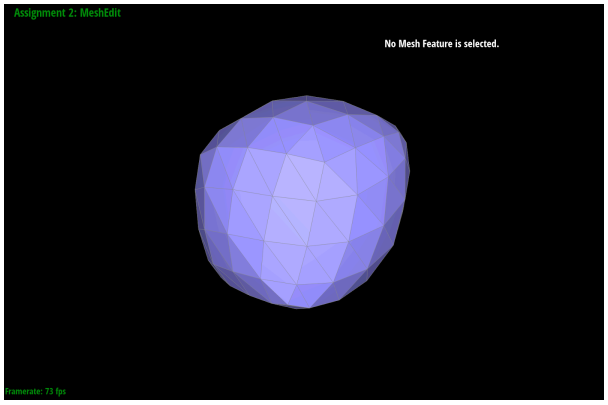
After Loop Subdivision, meshes become noticeably smoother (especially around sharp corners and edges). This smoothing effect can be reduced by pre-splitting certain edges. By doing this, you create areas with many short edges that visually maintain sharpness. When subdivision is applied, it averages over these short adjacent edges, which helps preserve the sharpness in those regions. As a result, the subdivision doesn't overly smooth out those edges.



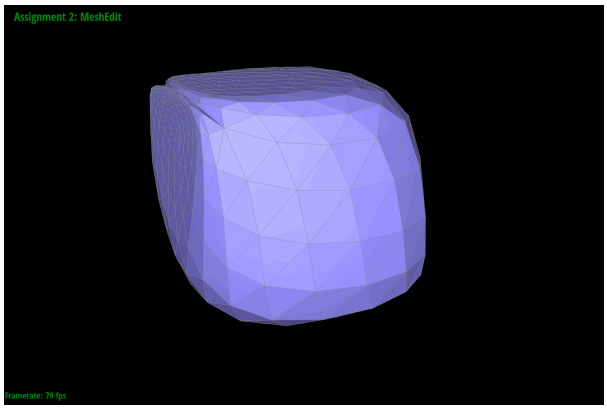
cube.dae



cube.dae after pre-splitting many edges near a corner

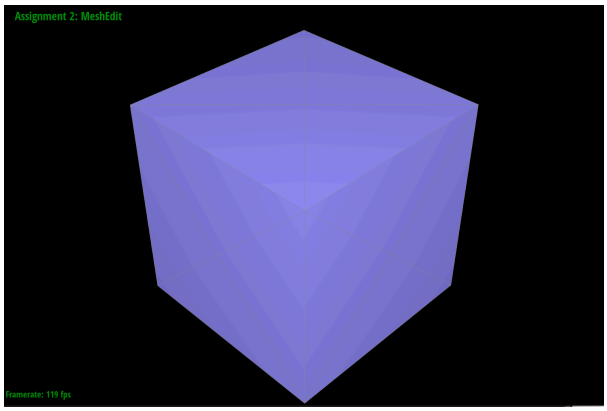


cube.dae after loop subdivision

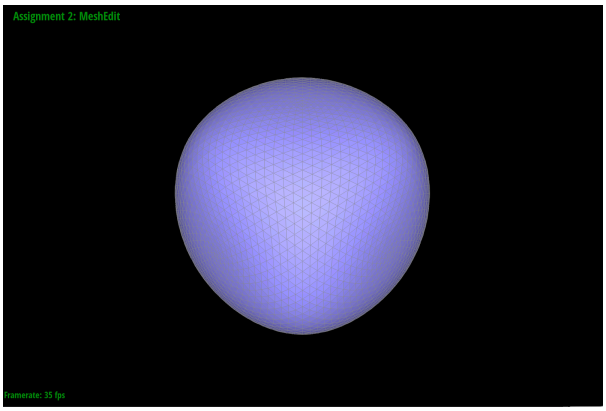


cube.dae (with pre-splitting) after loop subdivision

Some vertices seem to be smoothed less than others, it seems that the less degree the vertex has, it has less smoothing effect. I think this happens because even though the cube is geometrically symmetric, they are not really topologically symmetric. As is visible below, we just subdivides all the edges that lie in the diagonal of a face so that the cube is topologically symmetric.



cube.dae with edge splits on diagonals



a symmetrical, loop subdivided cube.dae