# CS184/284A Spring 2025 Homework 2 Write-Up

Names: Julia Isaac and Richard Yang

Link to webpage: https://ryang3881.github.io/cs184/hw2
Link to GitHub repository: https://github.com/cal-cs184-student/sp25-hw2-julia

## Overview

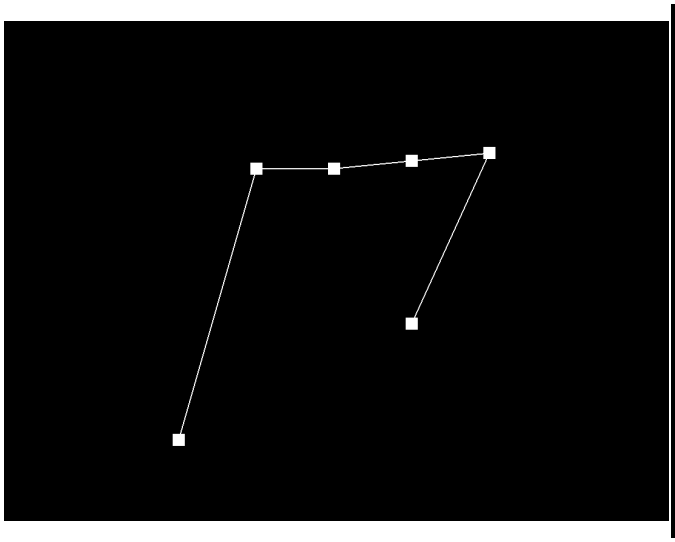
In Section I, we implemented de Casteljau's algorithm to compute points along a Bezier curve, and then extended this to surfaces beyond just curves. In Section II, we implemented area-weighted normal vectors at vertices, then local remeshing operations edge flips and edge splits to help support loop subdivision.

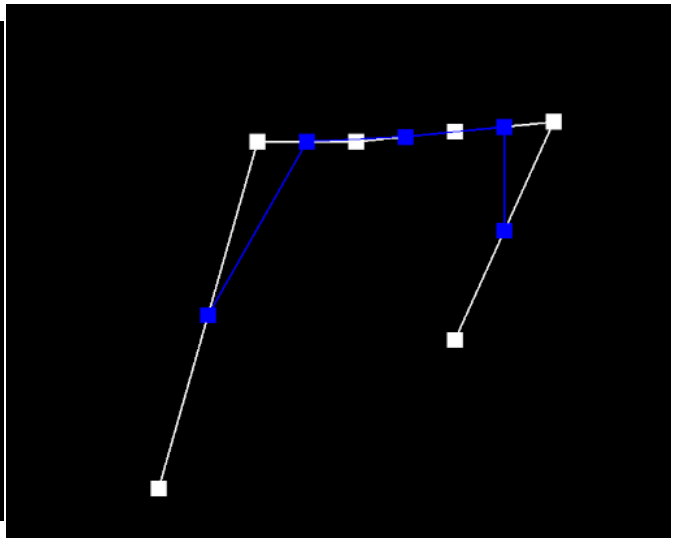
## Section I: Bezier Curves and Surfaces

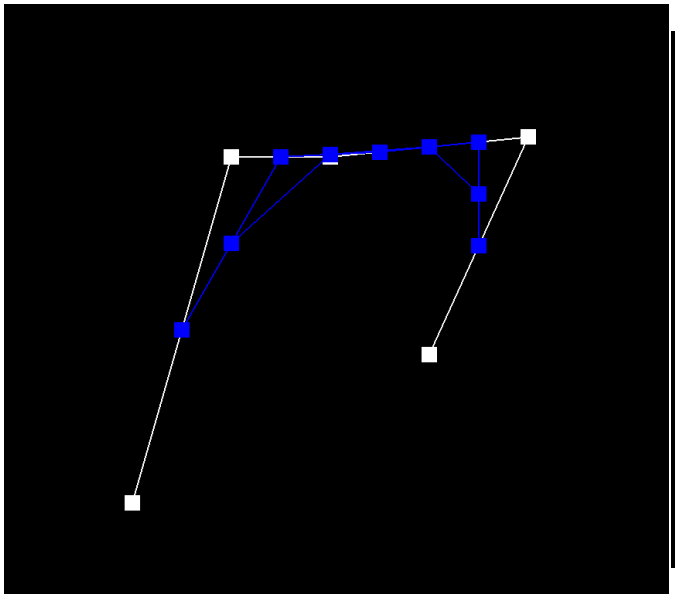
### Part 1: Bezier curves with 1D de Casteljau subdivision

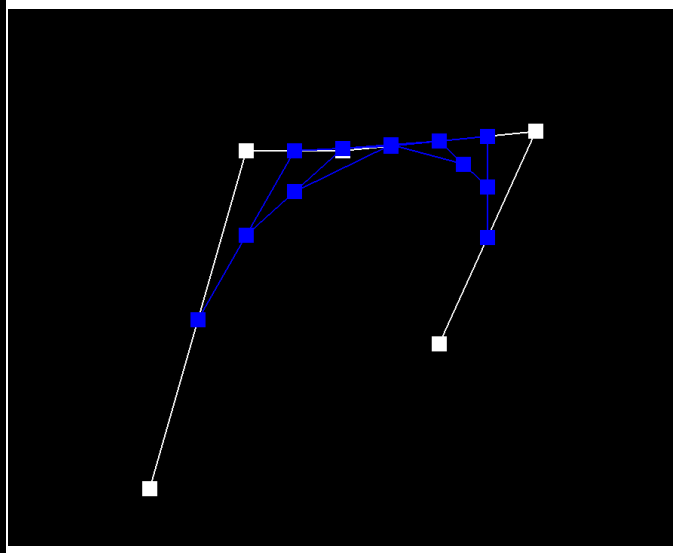
The de Casteljau algorithm is a recursive method to evaluate polynomials in a Bezier curve. Intuitively, on given two points, the de Casteljau algorithm linearly interpolates a point in between them given a t value. If we have three points given, we can repeat this for the two pairs to get two new linearly interpolated points. We can apply this recursively to interpolate between this new generated pair and now connect this final point when we make our Bezier curve. I implemented only one step of this by given n (possibly intermediate) control points and the parameter t, I loop through the control points and for each pair (p_i, p_i+1) I calculate ( 1 – t ) p_i + t p_i + 1 and return new calculated points into a vector.

Completed Original Bezier Curve

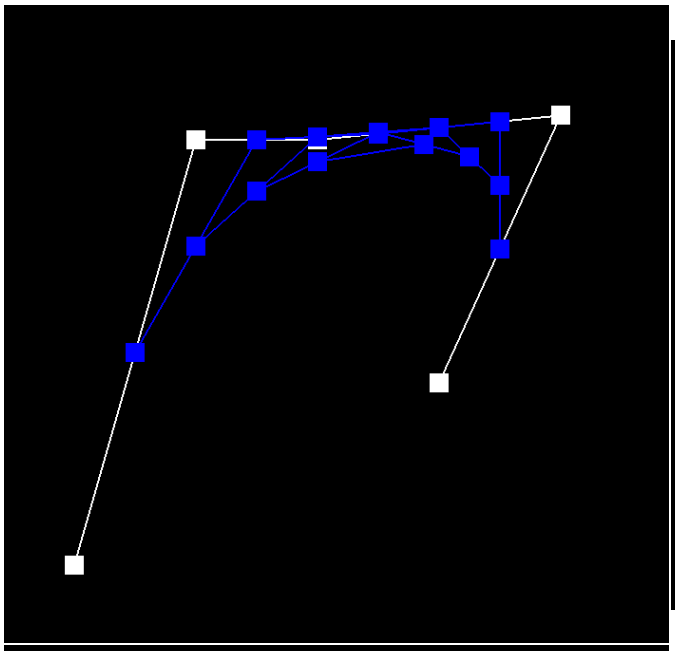


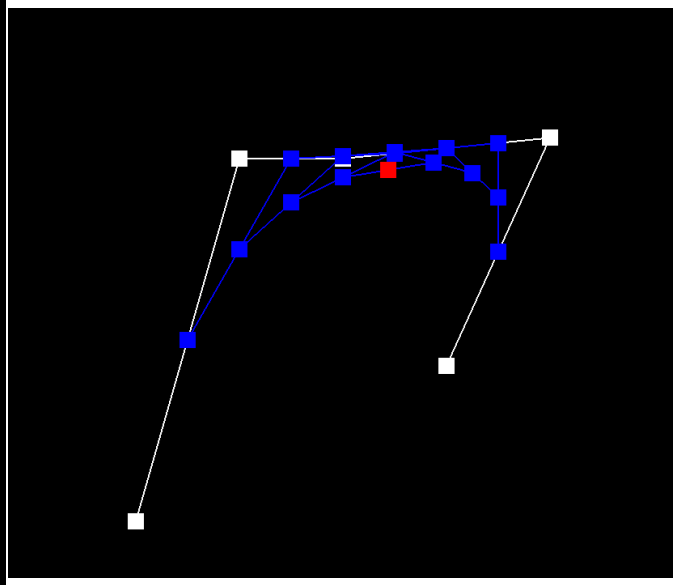


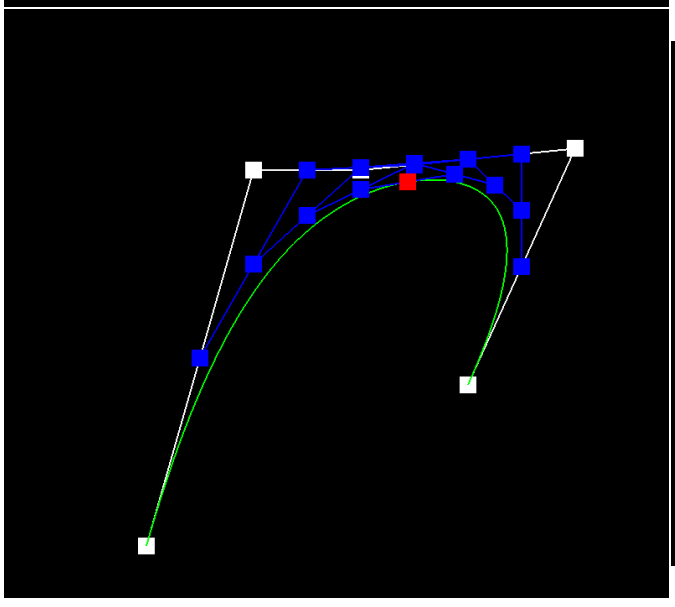


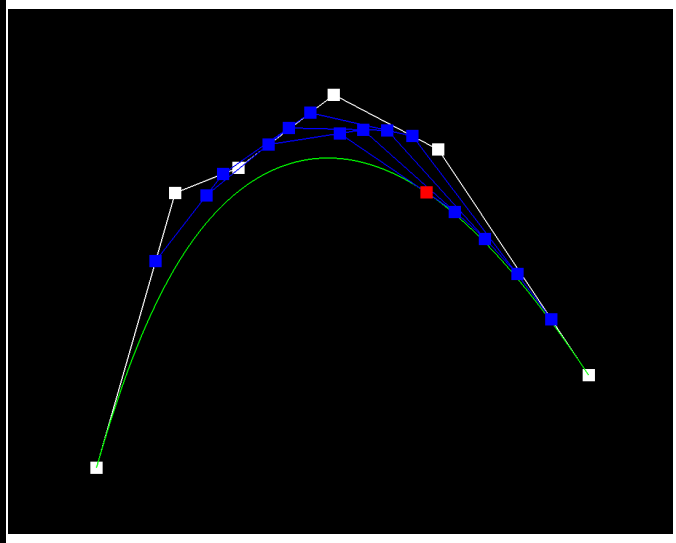


Slightly Different Bezier Curve with a Modified Parameter t.









### Part 2: Bezier surfaces with separable 1D de Casteljau

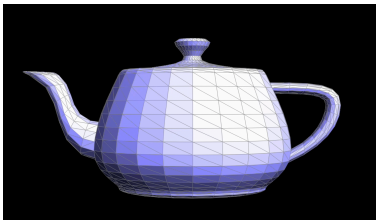
The de Casteljau algorithm extends to Bezier surfaces by first only considering each row the n x n control points, and then defining a Bezier curve parameterized by u. We then take these final outputted points on each Bezier curve at u, and then run the 1D algorithm again given a list of points, with one point for each row in the grid, we define a Bezier curve parameterized by v. I implemented this by first filling out the evaluate step function. I loop through the control points and for each pair (p_i, p_i+1) I calculate ( 1 – t ) p_i + t p_i + 1 and return new calculated points into a vector (same as part 1 but in 3d). I call this function recursively in evaluate1D until my output is length 1 and I return that element. In the final evaluate function call evaluate1D on the rows of the control points and then call evaluate1D once more on the row outputs.

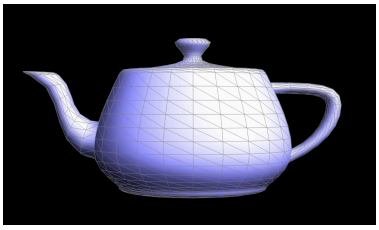
Here is the teapot mesh generated using the separable 1D algorithm.

## Section II: Triangle Meshes and Half-Edge Data Structure

### Part 3: Area-weighted vertex normals

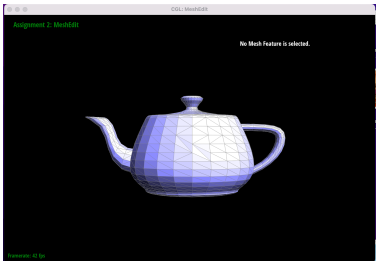
I iterate through faces by following a similar code to the primer where I get the face of a half edge each time changing to the twin's next pointer. I then loop through the 3 vertices in each face and save them into a list, calculating the normal per face by taking two sides connected by one vertex and taking the cross product. Since the area of a triangle is proportional to the magnitude of the normal at each face, I sum up the face normals which implicitly weights it by area. I then normalize the final normal vector to be unit length as the slides show.



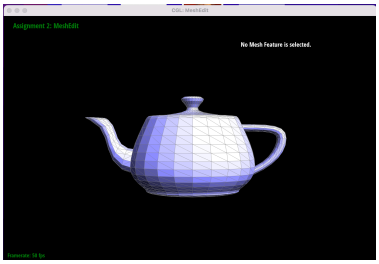


### Part 4: Edge flip

To implement the flip edge operation, first I check if the edge is a boundary (either its half edge or its twin is a boundary), and if so return immediately. I then I closely followed this guide to draw the 2 before and after (edge flip) triangles http://15462.courses.cs.cmu.edu/fall2015content/misc/HalfedgeEdgeOpImplementationGuide. I set all neighbors of the half faces appropriately (even if the pointers don't change just to be safe) and then I assign the half edges to vertices, edges, and faces accordingly. Debugging took a really long time because I didn't remember to include the halfedges "outside" the triangles, which led to faulty behavior since flipping an edge will affect their "twin" pointers. Once I realized this and fixed it, my code worked!



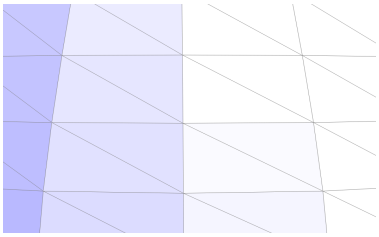

First I flipped some edges,



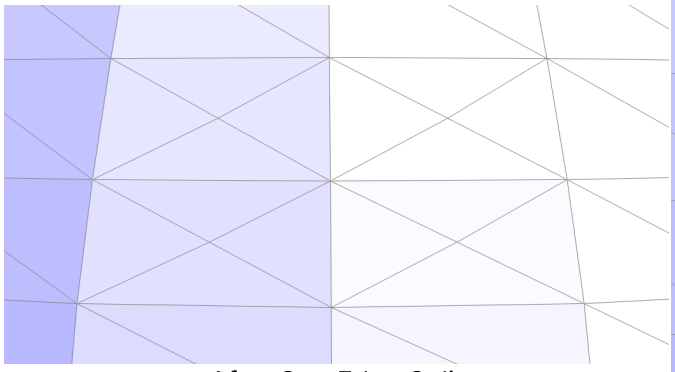

and then I Unflipped them (to make sure code works with multiple flips) to get the second picture!

## Part 5: Edge split

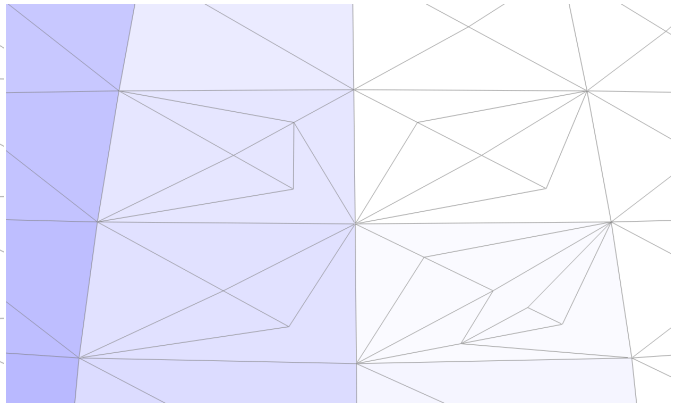
I implemented the edge split operation similarly to the edge flip operation by drawing out the two triangles (with vertices, edges, halfedges, and faces) before and after the split. I set all neighbors of the half faces using the setNeighbors function and then I assign the half edges to vertices, edges, and faces like last time. This time, however, I needed to actually create a new vertex, who's position was an average of the two old vertex positions of the edge being split. I also needed to create 3 new edges and 2 new faces. Luckily, I didn't need to debug much since I learned my major mistake from part 4 and I was really careful with drawing out the triangles. I did accidentally write one of the vertices wrong but found this by just double checking with my drawing thoroughly!



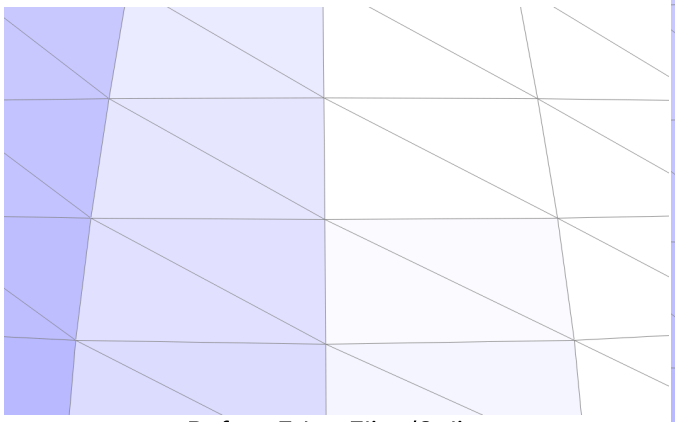
Before Edge Splits



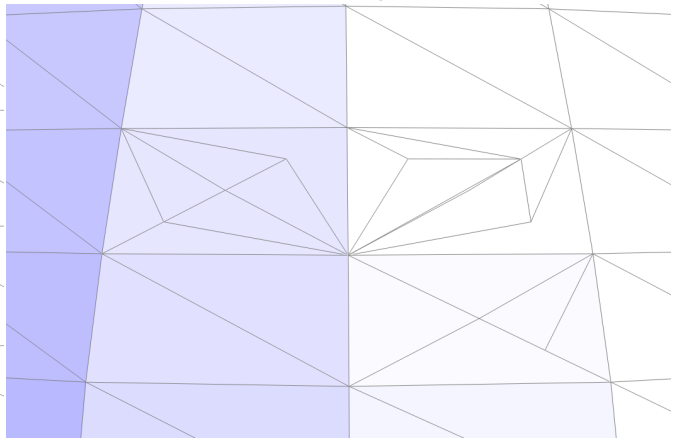
After One Edge Split



After Many Edge Splits



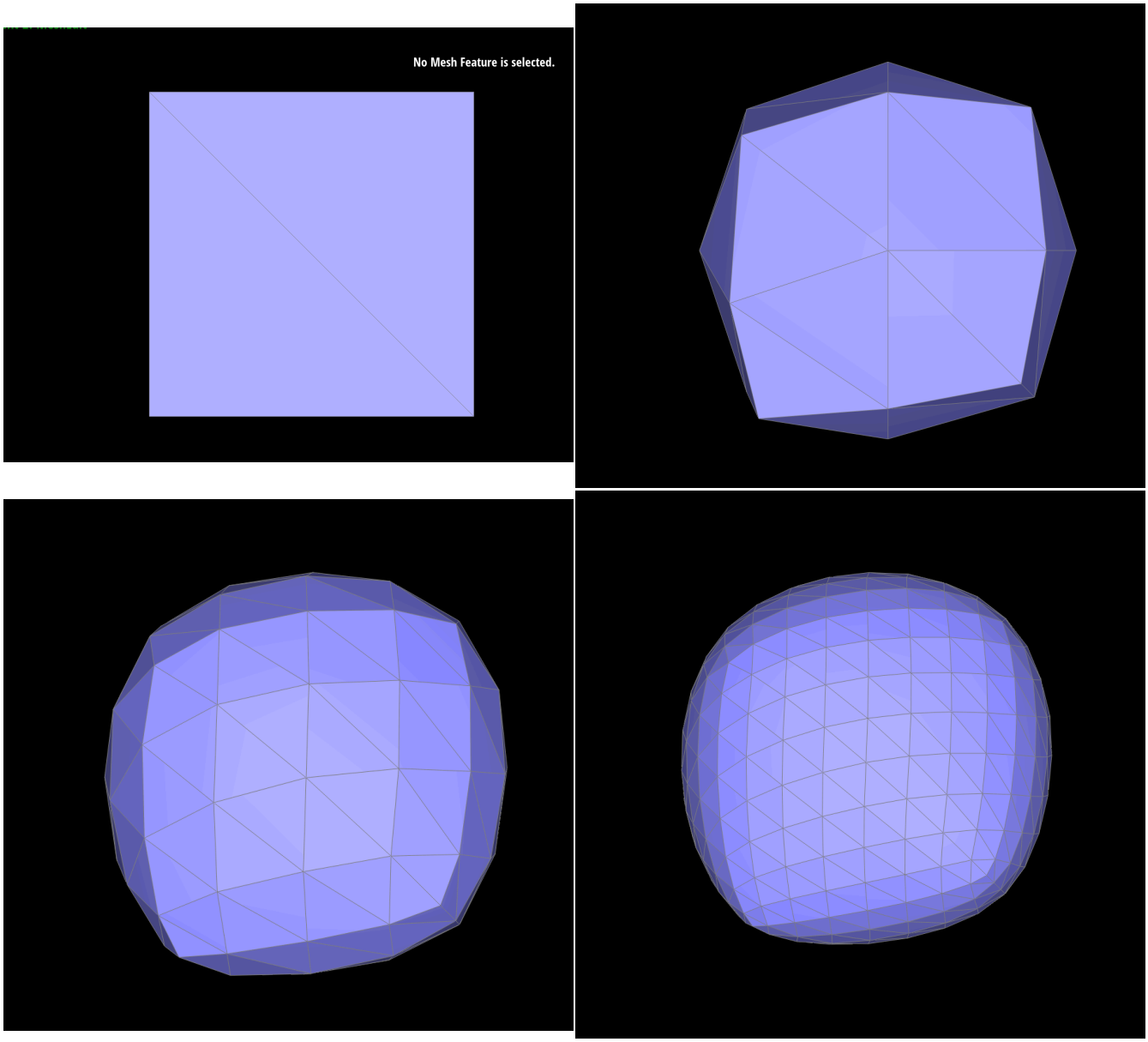
Before Edge Flips/Splits



After Many Edge Flips/Splits

## Part 6: Loop subdivision for mesh upsampling

I implemented loop subdivision by first computing the new positions for all the vertices in the input mesh, using the Loop subdivision rule v→newPosition = (1 - n * u) * v→position + u * original_neighbor_position_sum. Where I calculate the original_neighbor_position_sum by looping over all neighbouring vertices of a given vertex and summing up their positions. I store these calculated positions in Vertex::newPosition. I also mark each vertex as being a vertex of the original mesh. In the second step, I compute the updated vertex positions associated with edges by referring to the diagram featured in the spec. In the third step, I split every edge in the mesh and also set the vertex new position to the edge new position. This part was hard to debug because I ran into an infinite loop (which was due to splitting edges that we just split so I had to end up deciding if an edge was old if both of their vertices were not new. Then I flip any new edge that connects an old and new vertex. Finally, I copy the new vertex positions into the final Vertex::position.




Here I perform several iterations of loop subdivision on the cube. But notice that the cube becomes slightly asymmetric after repeated subdivisions. Since loopsub division changes the position of vertices based on the topology of their neighbors, we can alleviate this effect by pre-processing the edges before loop subdivision. We originally had one triangle cutting across the face of the cube, which is not symmetric. I achieve a symmetric looking cube by splitting every edge that goes across the face of the cube such that we create an “x” on each face. This allows symmetry since the X shape is symmetric about the reflection of every planar cut through midpoints. We get balanced vertex valence (where not with only one diagonal) and no directional bias to maintain a symmetric shape across multiple subdivision steps. See images below for the symmetrical cube!