HW2: Meshedit

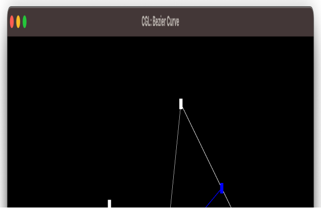
Overview

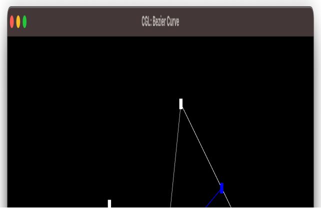
This project focuses on implementing techniques for geometric modeling to support a rudimentary mesh viewing and editing application, starting with Bezier curves and surfaces and eventually moving on to more complex mesh operations such as edge flipping and subdivision to improve mesh resolution. We work with algorithms such as the de Casteljau algorithm for Bezier curve interpolation as well as geometric algorithms for modifying edges and vertices in order to implement loop subdivision.

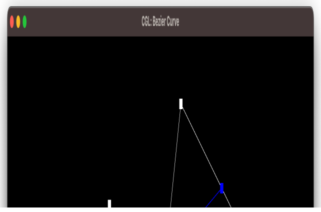
Task 1: de Castaeljau's Algorithm

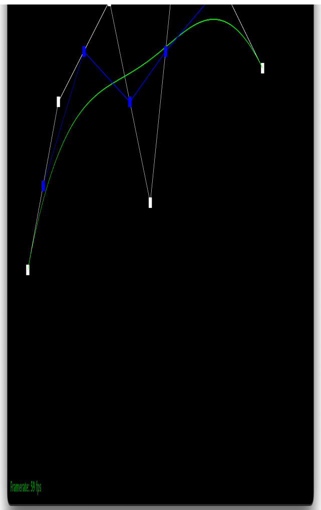
De Casteljau's algorithm is a simple interpolation algorithm that, given a set of control points and an interpolation parameter t, recursively interpolates between neighboring control points until we're left with a single control point, the actual point on the Bezier curve corresponding to t. In our implementation, we implemented it recursively using the vector data structure to store our intermediate interpolated points (calculated as *t * first_point + (1 - t) * second_point*) at each level, returning the single point in the vector when the algorithm is finished.

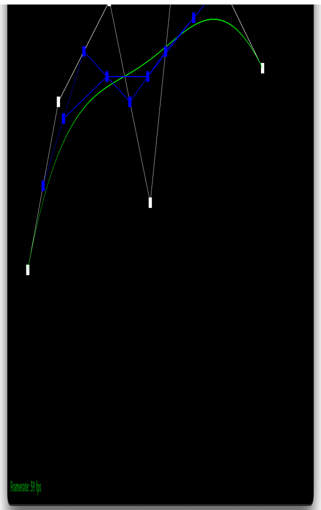
Below, we show the steps to find a point at t = 0.5 with a custom curve we defined as well as another point on a slightly modified curve at a smaller t-value.

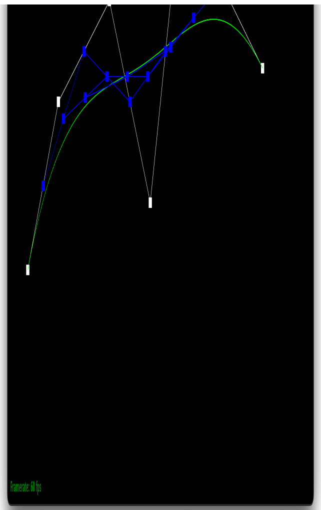


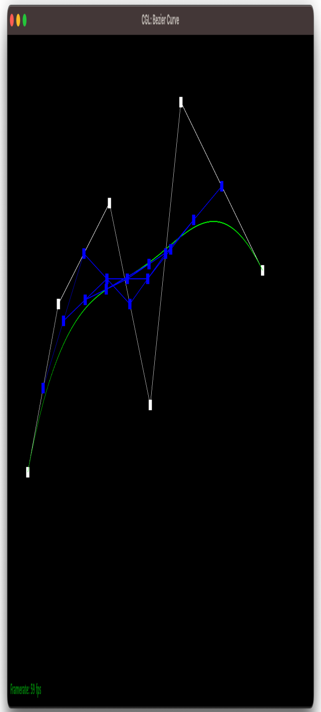


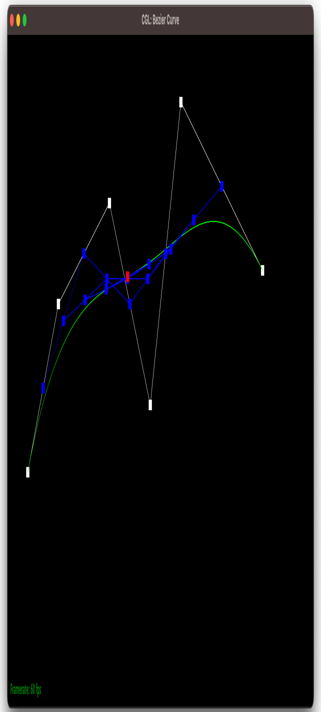


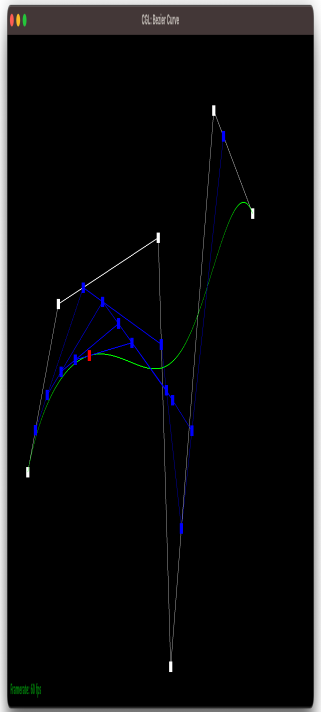

De Casteljau level 1


De Casteljau level 2


De Casteljau level 3


De Casteljau level 4

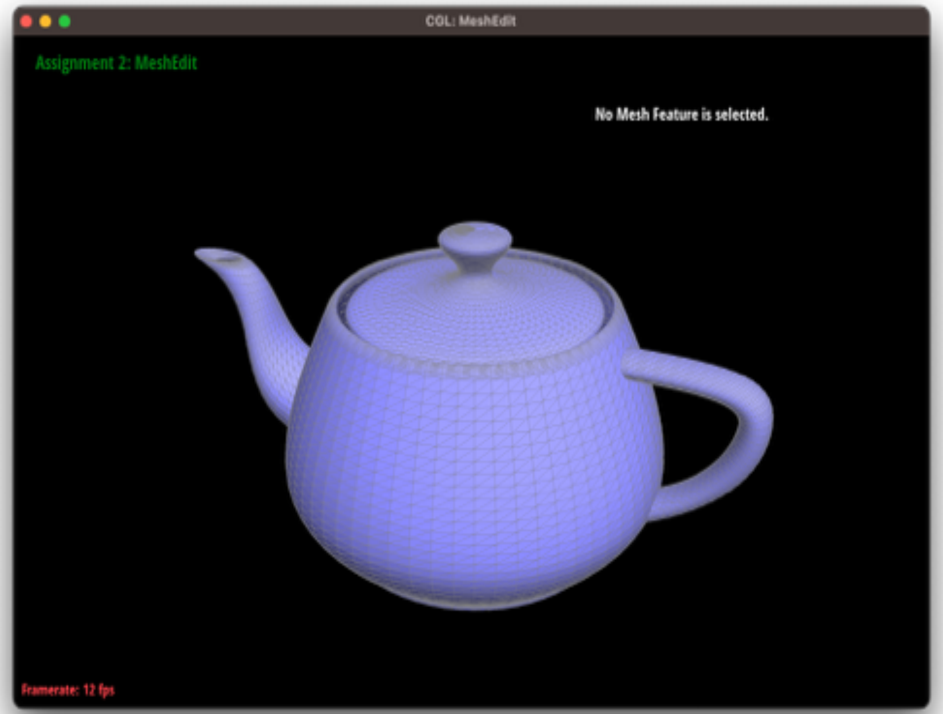

De Casteljau level 5


Other curve, de Casteljau level 5, different t-value

Task 2: Extending de Casteljau to Bezier Surfaces

De Casteljau's algorithm doesn't just apply to Bezier curves! Given an M x N set of control points and two interpolation parameters u and v, we can interpolate a point on the Bezier surface formed by the control points using the same 1-D de Casteljau algorithm with slight modifications. Here instead of putting all the control points in, we treat each row of control points as its own Bezier curve and use the 1-D algorithm to find the interpolated point at u on each curve. We then interpolate to find the point at v on the new curve formed by the u-interpolated points to find the actual point at (u, v) on the Bezier surface!

To implement this in code, we re-used some of the code from the previous part and modified it to return a Vector3D instead of Vector2D. Again, we used a vector to store the intermediate 1-D u-interpolated points obtained from each vector in the vector of control point vectors and ran the 1-D routine again on those points to get the final point at (u, v).

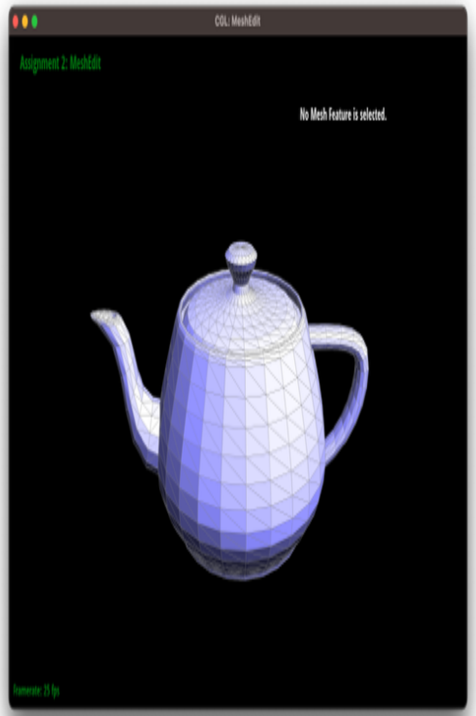


Meshedit ran on teapot.bez, now with Bezier surfaces!

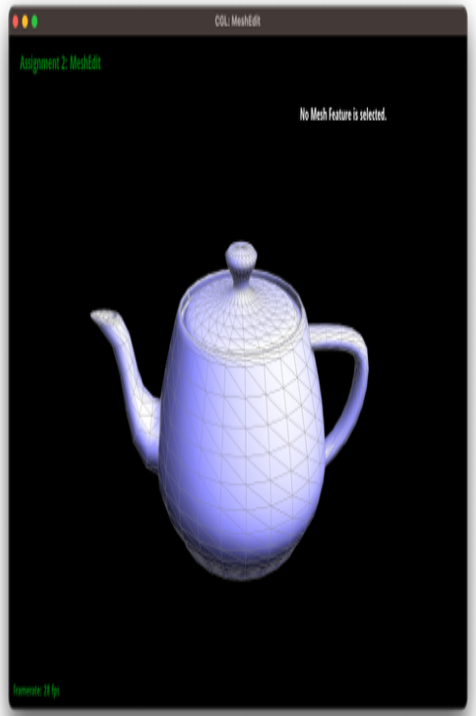
Task 3: Area-Weighted Vertex Normals

To implement the vertex normals, for each half edge bordering the vertex, we iterate through the neighboring half edges for that face to find each vertex making up the triangle. We then calculated the normal for that face by taking the cross product of the vectors from the original vertex to each of the other vertices. By repeatedly iterating through the half edge's twin's next half edge, we can repeat this procedure for each face bordering the vertex. The final area-weighted vertex normal is then obtained by iterating adding up each face normal vector times the vector's norm (face area) and then normalizing.

Below the differences between flat shading and Phong shading are shown.



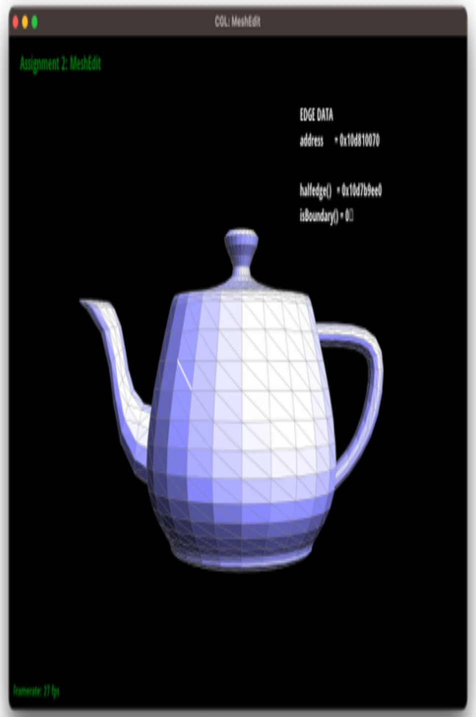
Flat shading



Phong shading with area-weighted vertex normals, much smoother!

Task 4: Edge Flipping

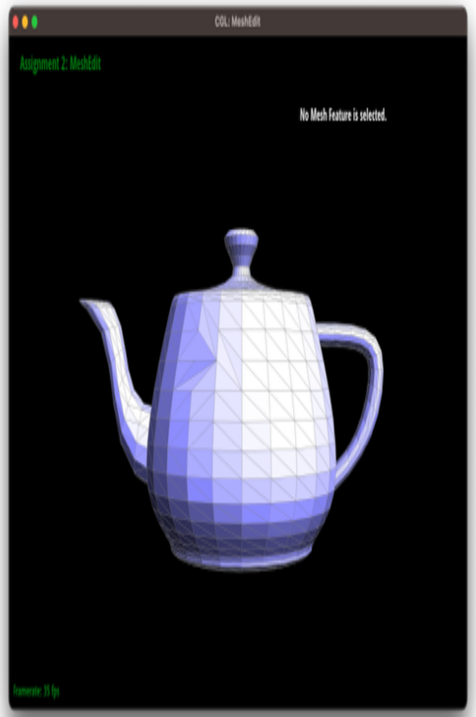
Edge flipping involves taking two bordering triangles and changing the orientation of the edge connecting them to connect the other two vertices instead. To implement this we manually grabbed references to every half edge, vertex, and face in each triangle by using the edge iterator passed in as well as any necessary instance variable and reassigning pointers accordingly. To ensure correctness, we used the diagram from lecture and drew out all mappings in the geometry from the old triangle to the new triangle. We debugged using inspection, but we had lots of confidence in our sketches!



```
EDGE DATA
address    = 0x10d810070

halfedge()  = 0x10d7b9ee0
isBoundary() = 0
```

teapot.dae before flipping edges



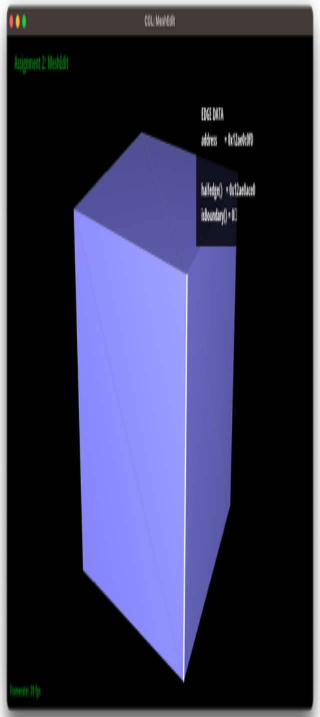
No Mesh Feature is selected.

teapot.dae after multiple edge flips

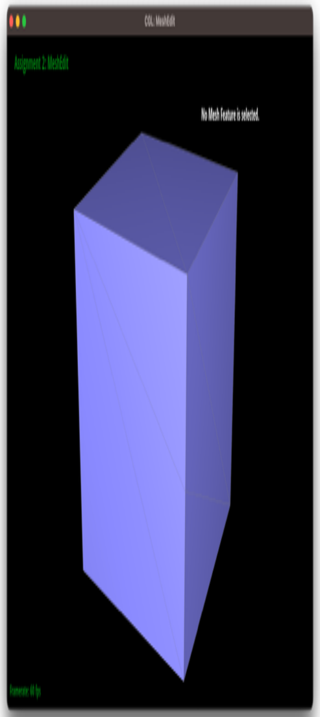
Task 5: Edge Splitting

Edge splitting is similar to edge flipping, but instead of flipping the orientation, we add in the new edge on top the old edge, creating a new vertex in the middle. This involved significantly more pointer reassignments than the previous part since instead of just reassigning pointers, we need to do that on top of creating a new vertex, 2 new faces, 3 new edges, and all associated half edges.

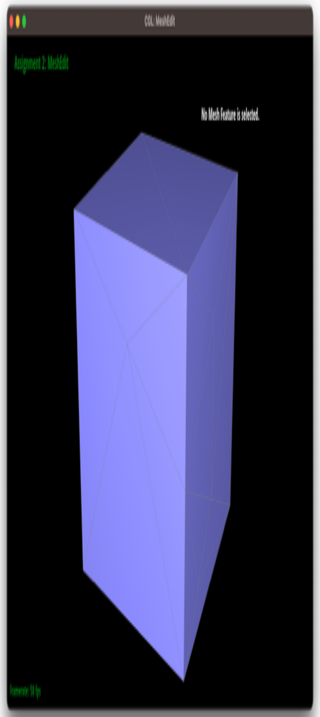
Thankfully, our approach of implementing it essentially the same way as edge flipping (drawing out all mappings from old triangle to new triangle(s)) let us finish this task without much need for debugging. What made sense for us when implementing the algorithm was also defining all the geometry that needed to be created at once, and then performing half edge operations one face at a time.



cube.dae



cube.dae, the edge selected in the first image has been split



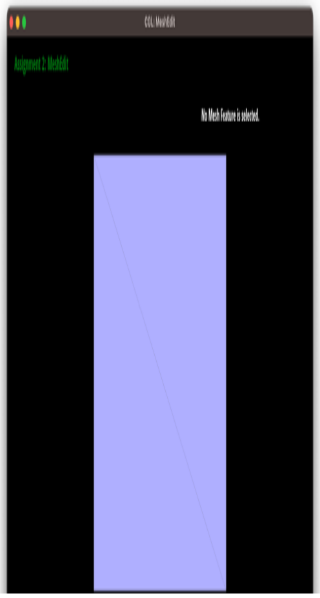
cube.dae after multiple splits and flips

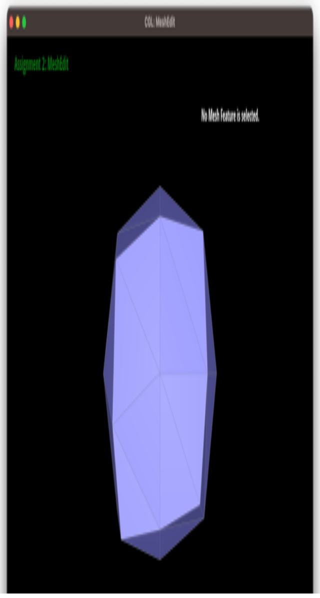
Task 6: Loop Subdivision for Mesh Upsampling

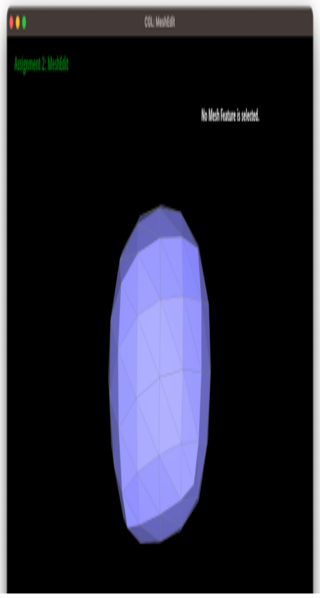
In order to upsample the meshes, we implemented the Loop subdivision algorithm. We followed the given procedure without any modifications. First we calculated the new positions of each original vertex using the Loop subdivision rules, then calculated new positions for all "edge" vertices to be inserted during subdivision. To perform the subdivisions, we then did splits on all edges and flips on only new edges created when splitting by taking a look at the isNew instance variable for all geometry, set during the first part of the subdivision or during an edge split. Finally we adjust the vertex positions to what we initially calculated, completing the process.

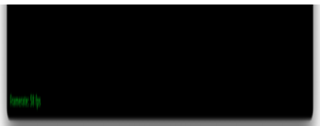
During the implementation/debugging process we didn't run into much trouble since we followed the given procedure, but some interesting errors we ran into involved vertex positions not being calculated correctly due to a typo in the math as well as a typo resulting in using the wrong vertices to calculate new positions of "edge" vertices inserted.

Below are some images of cube.dae after several rounds of subdivision.

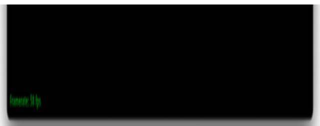




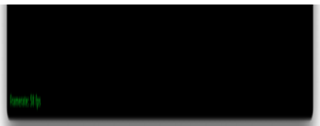




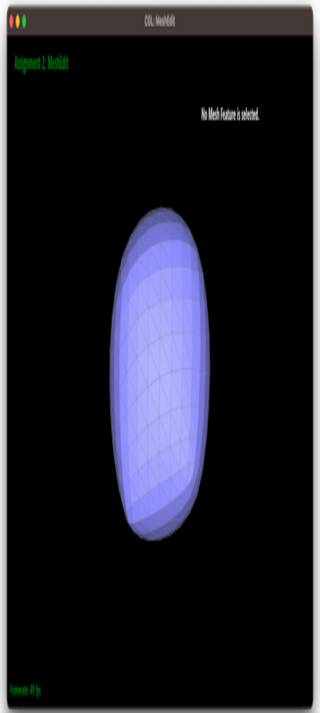
cube.dae, no subdivision



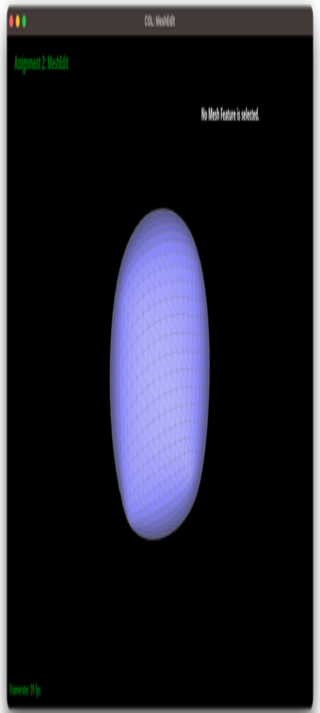
cube.dae, 1 round of subdivision



cube.dae, 2 rounds of subdivision



cube.dae, 3 rounds of subdivision



cube.dae, 4 rounds of subdivision

Some interesting behaviors we noticed in the cube and other meshes in the dae folder were that the mesh often doesn't subdivide symmetrically, as vertices with larger degrees tend to change position less than vertices with smaller degrees due to the Loop subdivision vertex rules heavily reducing the weight of vertices' neighbors in the original vertices' new positions as degree increases due to the degree being in the quotient. The orientation of edges affects degree and is thus related to how the mesh deforms after subdivision as well.

We attempted to completely reduce these effects in cube.dae by splitting edges to make faces symmetric, but since spliting marks the new edges created as new, the new edges resulted in weird deformations as the subdivision algorithm attempted to flip them during the 4-1 subdivision process. Flipping edges also didn't work since no amount of flipping would result in every vertex on the cube having the same degree, so even though the resulting geometry changed in a more symmetric way depending on which faces we flipped edges on to look the same, it was never fully symmetric.