CS 284 HW-2 Write-Up

MeshEdit

Yunzhe Liu - <3038581132> - liuyunzhe@berkeley.edu
Linzhe Tong - <3038584629> - billtong1998@berkeley.edu

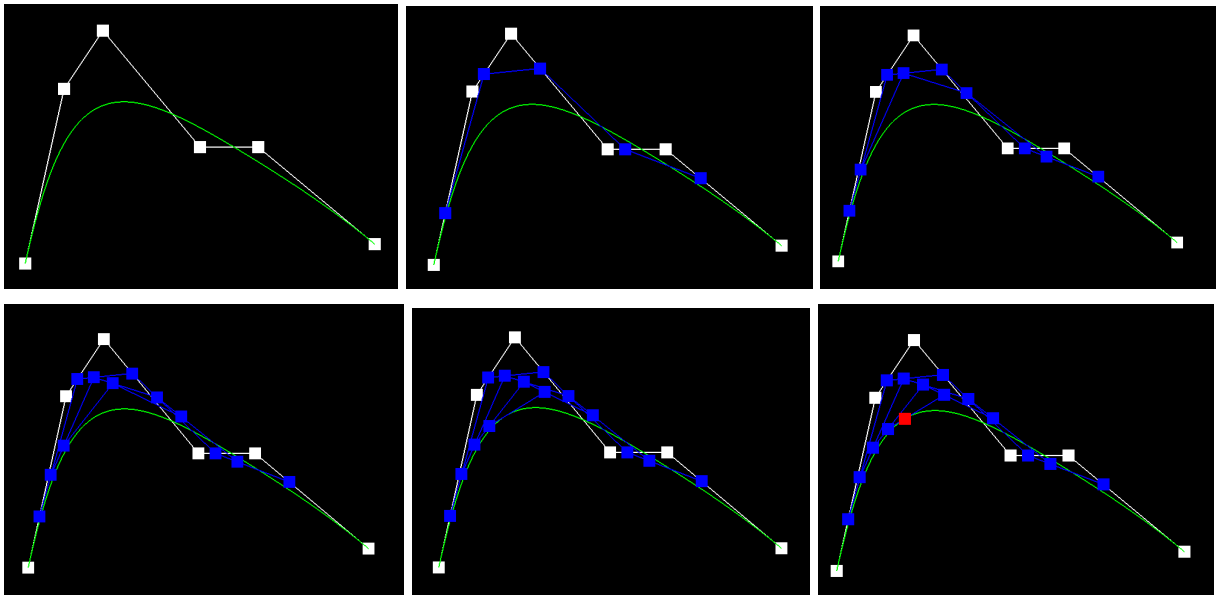
Webpage: https://hyperloop001.github.io/proj-webpage-template/proj2/index.html

Overview

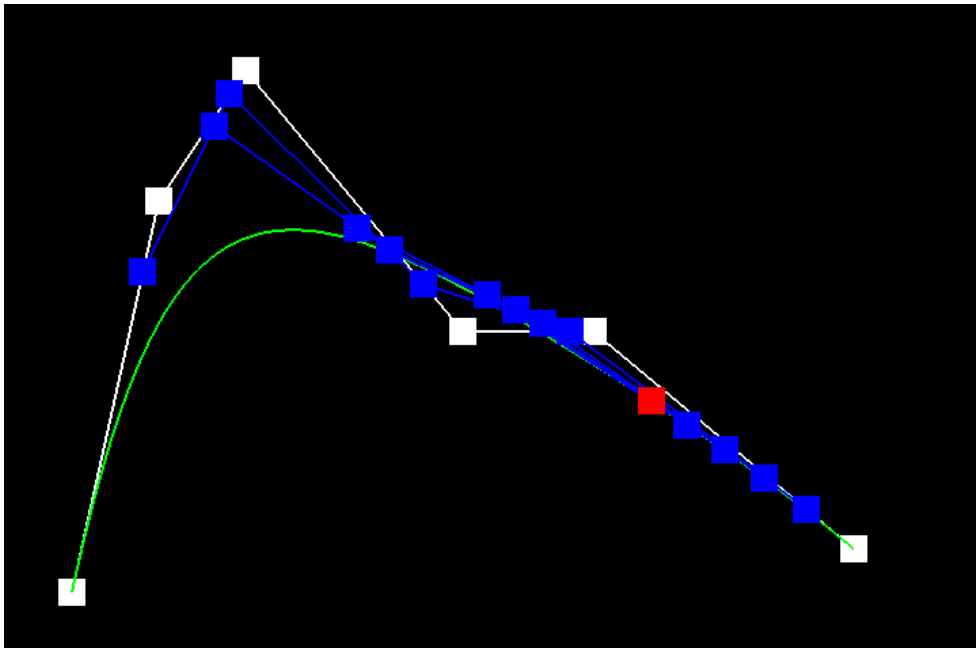
In this assignment, we implemented some basic curve rendering and mesh editing methods. In the first section of our assignment, we used de Casteljau's algorithm to create Bezier curve representations and further extended it to render 2-D curvature surfaces. For the second section, we built a complete editing algorithm for picking on and modifying the mesh and surfaces, which includes finding vertex normals, flipping/splitting edges, and upsampling edges. The most fascinating part of this assignment is learning and understanding how these simple meshes formed the foundation of modern 3-D computer graphics.

Task 1

In this task, we implemented the *BezierCurve::evaluateStep* function using de Casteljau's algorithm to calculate one control point sampling step of the bezier curve. At each step of the algorithm, we combine two adjacent control points into a new control point using the lerp function with respect to an input scalar. For the algorithm to work properly, we continue this sampling step until there is only one finalized control point — the control point that draws all points of a Bezier curve.



Sampling process of a Beizer curve with 6 control points

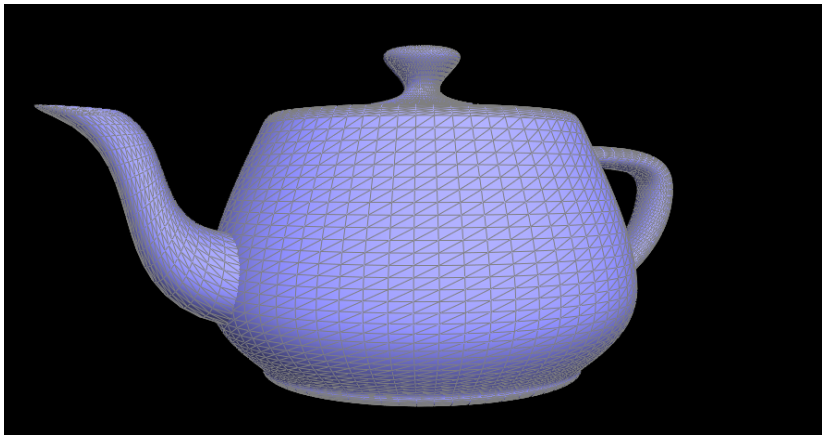


Beizer curve with toggled parameter t

Task 2

In this task, we extend the 1-D Beizer curve into 2-D Bezier surfaces. We have implemented three functions to achieve this:

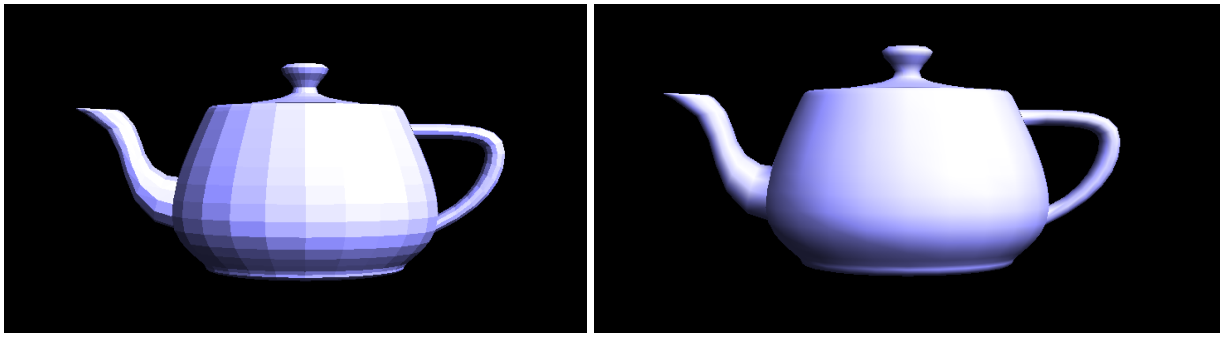
- *BezierPatch::evaluateStep*: Same as task 1, evaluating 1 step of a Beizer curve.
- *BezierPatch::evaluate1D*: Continuous evaluation of Beizer curve into a final control point.
- *BezierPatch::evaluate*: This is the core function to render a Beizer surface. It takes in a 2-D grid of original points, performs 1-D evaluation on each row to get a 1-D column of control points, then performs another 1-D evaluation on the resultant column points to render a Bezier surface.



Beizer surface rendering of bez/teapot.bez

Task 3

In order to calculate the area-weighted vertex normal, we started with a half-edge rooted at the current vertex. We then get the associated face of the half-edge and calculate the cross-product of two vectors formed by any of these three vertices of the triangular face, and add the resultant vector product into a summation. Note that the cross-product norm is naturally equal to the face area. We then hop onto the next half-edge rooted at the same vertex by going to the current half-edges' twin's next (i.e. h->twin()->next()) and repeat the process until we get back to the original half-edge. The last step is to normalize the summation vector to get the area-weighted vertex normal.

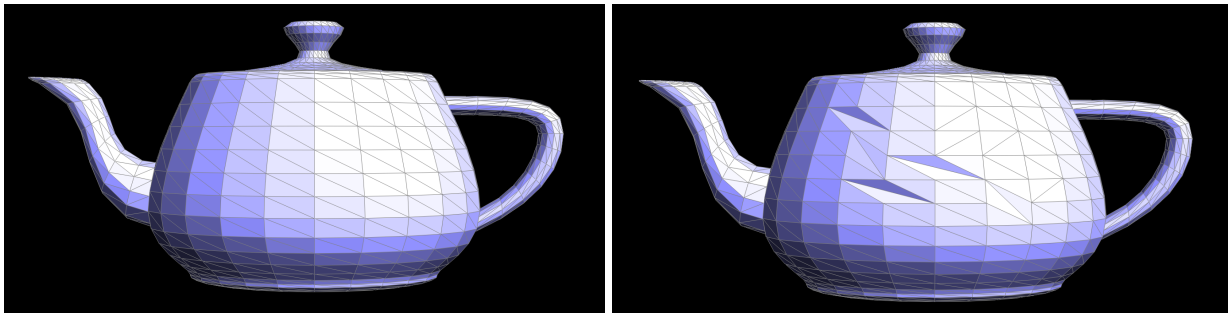


Shading with and without vertex normals on dae/teapot.dae

Task 4

In this part, we drew a mesh with a pair of triangles (a,b,c) and (c,b,d). Then we listed all the elements in the half-edge structure of this mesh. Note that there's no need to take any element outside these 2 faces into consideration because they keep the same while flipping this edge (b,c). Also, in this part, there's no need to add or remove any elements, all we need to do is to reassign the pointers. We modified 2 vertices, 2 faces, and 6 halfEdges in total. (A single halfEdge may need multiple pointer changes.) We omit the full modification list due to the complexity.

We hadn't experienced a debug stage because we carefully listed the elements and checked twice before we started to code.

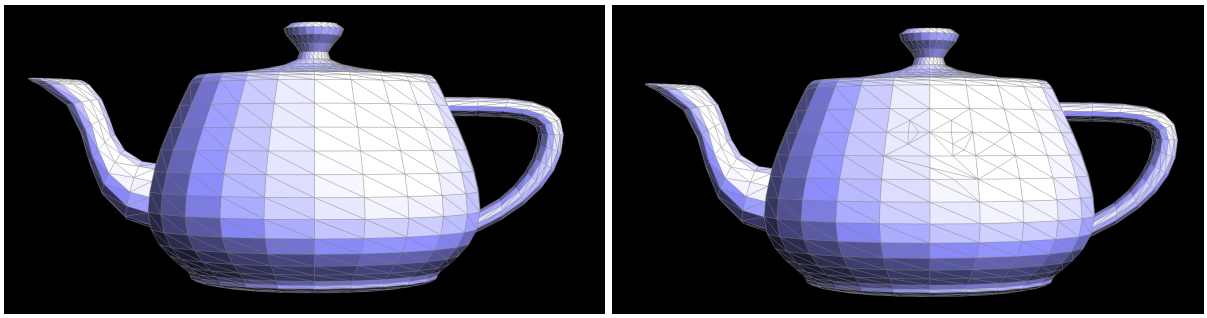


Teapot before and after flipping several edges

Task 5

In this part, we did similar things to the last one. We drew a draft mesh consisting of a pair of triangles and manually listed everything that needed to be changed. The difference here is that the split operation needs to add elements to the original mesh(1 midpoint vertex, 6 half-edges, 3 edges, and 2 faces). And it is much more complicated than the flip operation in the number of pointers needed to change. In our implementation, the split operation involves the modification of 2 vertices, 3 edges, 4 faces, and 7 half-edges. We omit the full modification list due to the complexity.

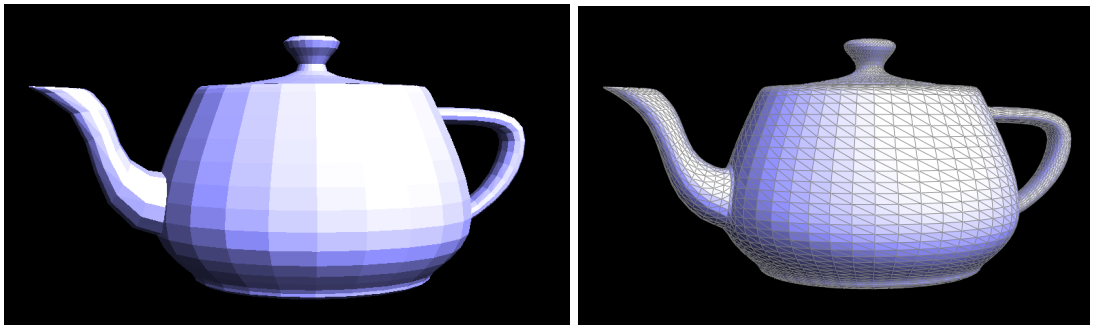
We hadn't experienced a debug stage because we carefully listed the elements and checked twice before we started to code. So coding is much easier than the preparation work.



Teapot before and after splitting several edges

Task 6

We implemented the loop subdivision exactly as the given assignment suggested. We first compute the new positions for all vertices in the original mesh, then, we split the edge and flip those new edges that connect an old and new vertex. Last but not least, we copy the new vertex positions into the final mesh. We were lucky and did not run into any bugs for this part.

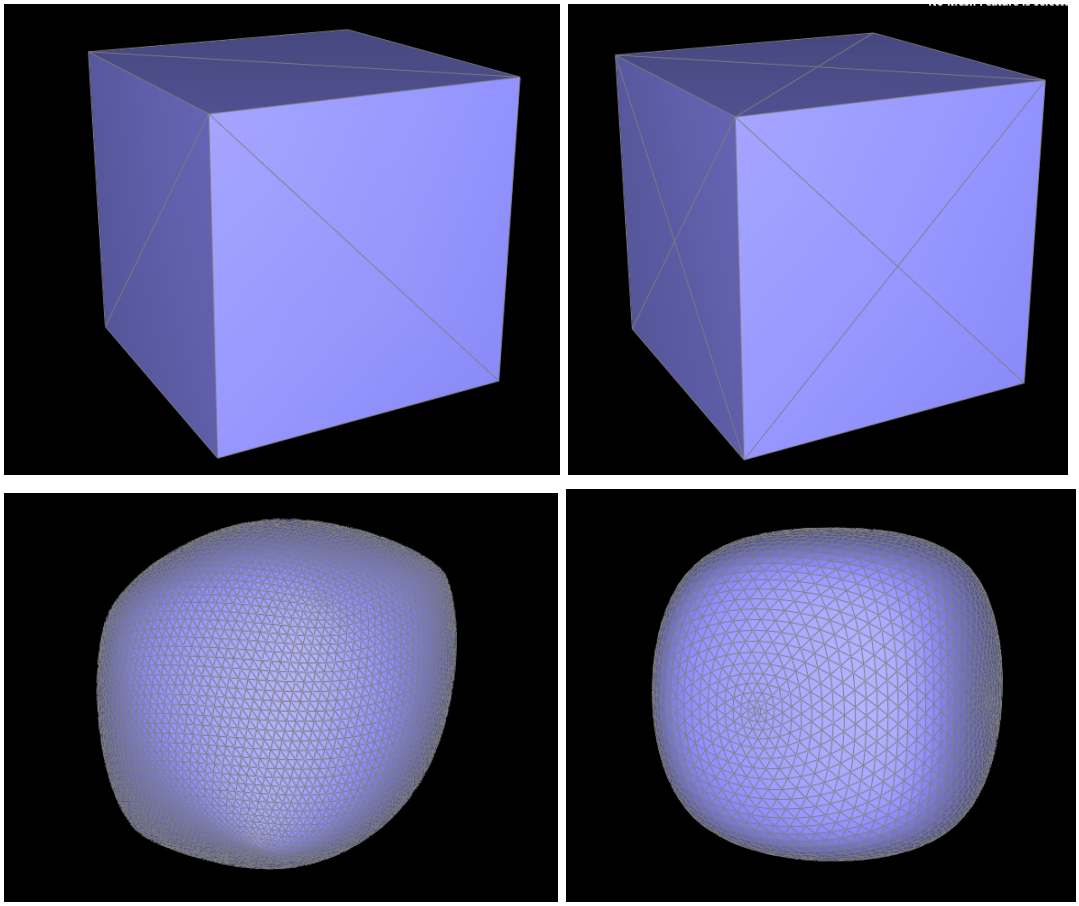


Teapot before and after loop subdivision

We noticed that after each loop subdivision, the teapot's surface gets smoother. However, the subdivision was asymmetric as shown in the figure above. We assumed that this is due to the

original mesh's allocation being asymmetric. We tried to flip/split some edges to make the surface symmetric and the upsampling become much better.

We then experimented with the dae/cube.dae. We noticed that the cube became asymmetric after upsampling. We believe this is exactly the same cause as we noticed on the teapot, so we preprocessed the mesh by creating a split on the asymmetric edges so the upsampling on the cube results in a symmetric mesh.



Comparison between original and pre-processed cubic mesh