CS 184: Computer Graphics and Imaging, Spring 2017

Project 2: Mesh Editor

Sol Ah Yoo, CS184-act

Overview

This project was mostly about geometry (Bezier curves/surfaces, edge flipping/splitting, etc). I learned how to use the HalfEdgeMesh data structure and implement loop subdivision to make the mesh smoother. The last part of the project was dealing with shaders.

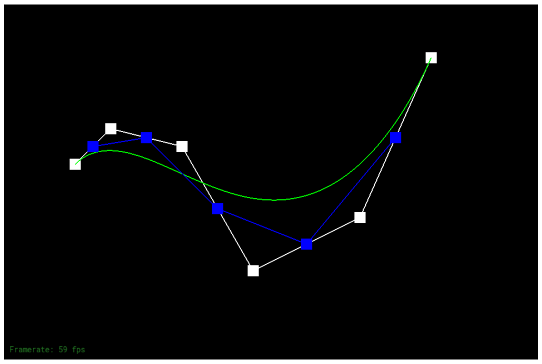
Section I: Bezier Curves and Surfaces

Part 1: Bezier curves with 1D de Casteljau subdivision

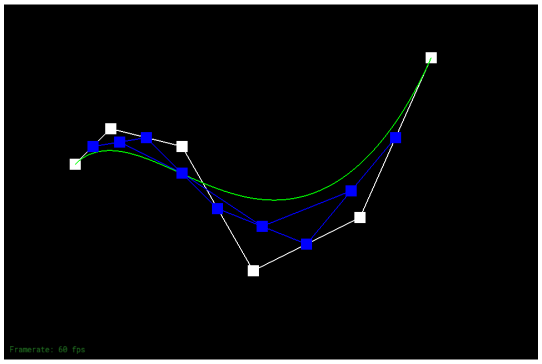
de Casteljau's algorithm is a way evaluate Bezier curves. Given k + 1 number of points, we can recursively apply this algorithm to create a new set of k points and so on until we are left with just one point, which is the point on the curve corresponding to time t.

Since Part 1 was only evaluating one step of the Bezier curse, I took the most recent level by taking the last vector of the evaluatedLevels vector and ran de Casteljau's algorithm once, where new points are created using this formula: p_i' = (1 - t) * p_i + t * p_{i+1}.

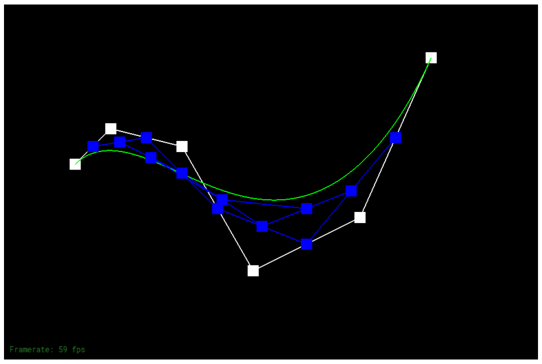
I had one for loop to iterate through all points and stored the new points in a vector called next, which I appended onto the evaluatedLevels vector.



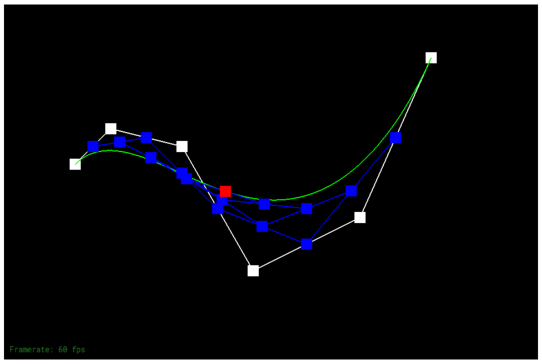
Step 1



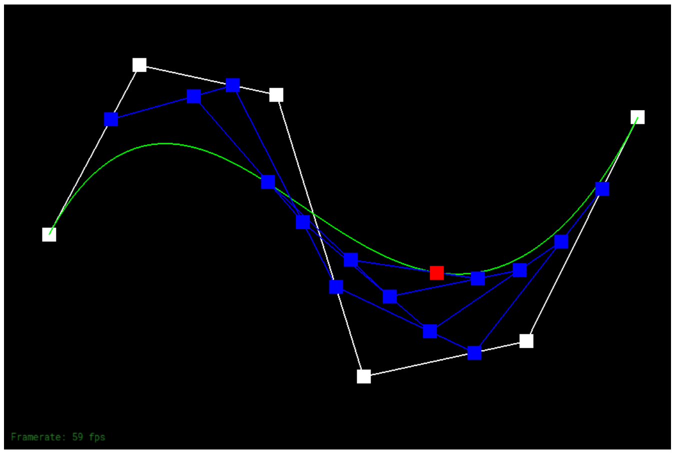
Step 2



Step 3



Last Step

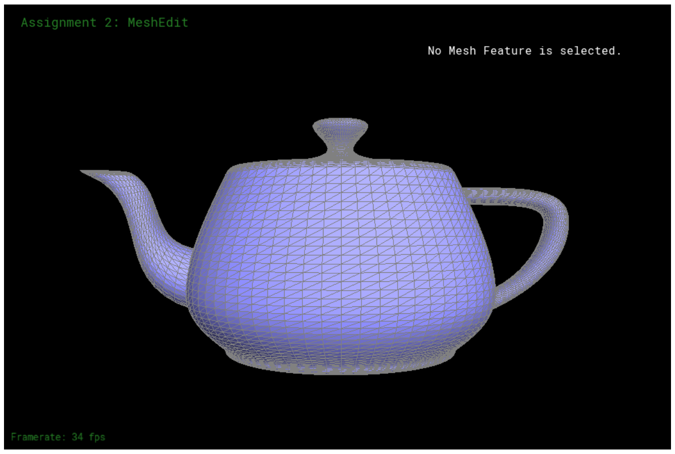


Different curve with modified t

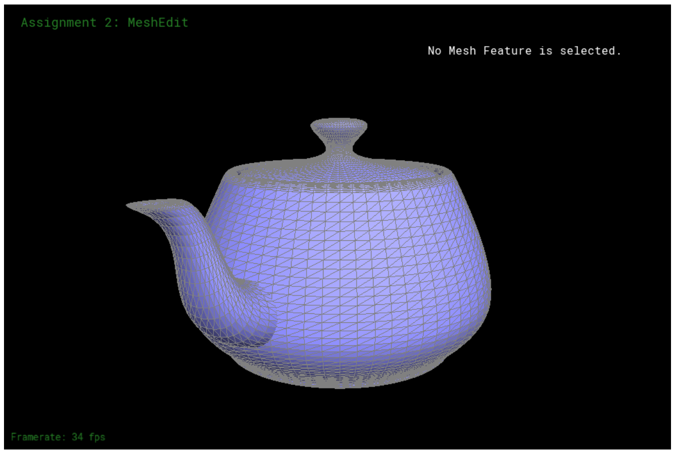
Part 2: Bezier surfaces with separable 1D de Casteljau subdivision

The de Casteljau's algorithm works a similar way for Bezier surfaces. Bezier surfaces can be interpreted as multiple Bezier curves in the 4 x 4 control area. So there is a Bezier curve for each 4x1 control point in u and the corresponding points on these 4 Bezier curves define the 4 control points for a moving curve in v that sweeps out the 2D surface.

To implement this, I first implemented a helper function similar to the function from part 1, except it evaluates all levels to return the final point. So I evaluated all curves in the i-th row with parameter u, then applied de Casteljau again for the points I got from the previous step.



Teapot - bez/teapot.bez



Teapot - bez/teapot.bez

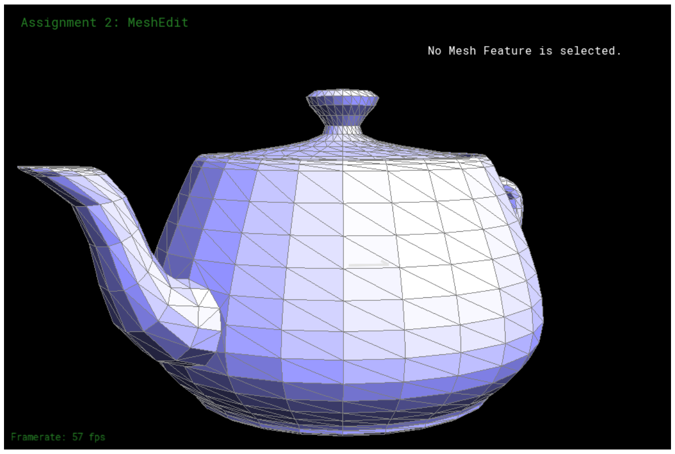
Section II: Sampling

Part 3: Average normals for half-edge meshes

To get the average normal vector, I first had to find the triangle edges to calculate the cross product. I got one edge by subtracting the current halfedge()->vertex()->position by halfedge()->twin()->vertex()->position and the other by subtracting halfedge()->twin()->vertex()->position by the halfedge()->next()->twin()->vertex()->position. Then, I advanced h to the halfedge of the next face by setting h = h->twin()->next().

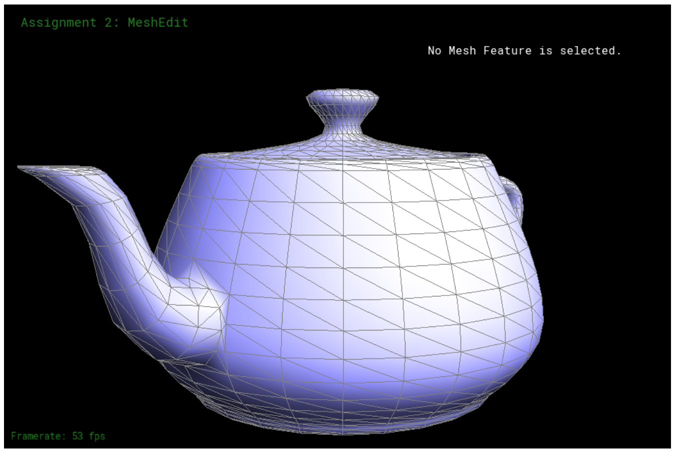
Once this algorithm ran through all the halfedges until it returned to the original one, I returned the normalized vector of the accumulated sum of the normal (return n.unit()).

The hardest part of this was making sure I was iterating through the halfedges correctly. My first few attempts resulted in a teapot that weird angles on the handles and mouth.



```
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 57 fps
```

Teapot - dae/teapot.dae



```
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 53 fps
```

Teapot - smooth

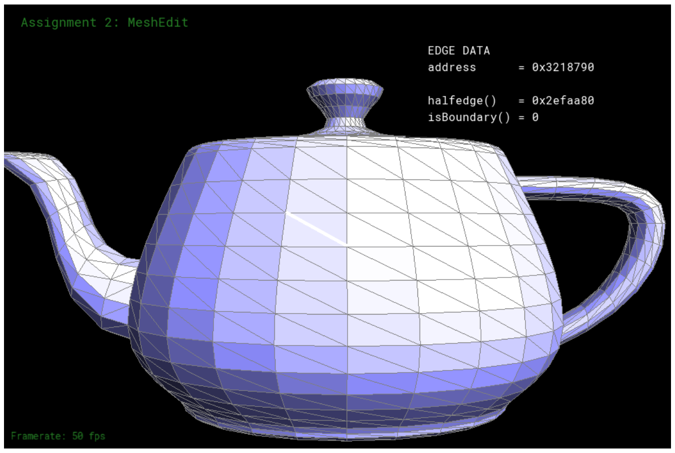
```
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 59 fps
```

Teapot - smooth

Part 4: Half-edge flip

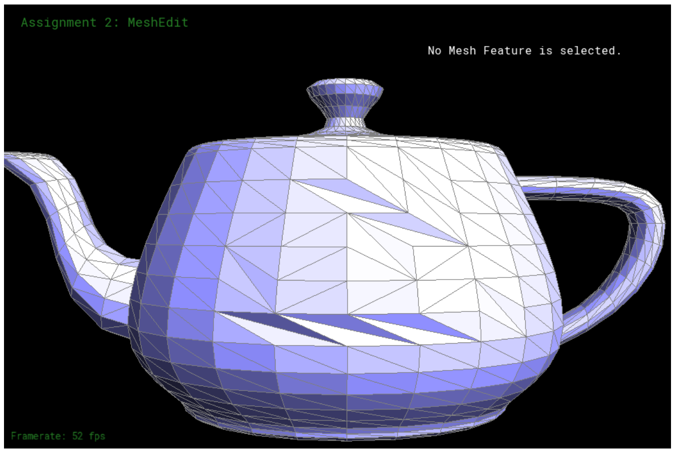
To reassign pointers to the correct edges and vertices, I first stored the halfedges in variables so I could access them easily. Then I reassigned all 4 vertices' halfedges to different halfedges and did the same for the 2 faces (Vertex a->halfedge() and Vertex d->halfedge() = flipped edge and so on). Next, I reassigned all halfedges' elements (next, twin, vertex, edge, face) using setNeighbors().

Some bugs I had were because I forgot to assign the correct halfedges to the faces, so I would get black empty spots when I tried to flip an edge. Some debugging strategies I used were to look at the addresses to figure out if halfedges or faces were pointing to the correct elements. I also found it very useful to draw diagrams to visualize which pointers should point to which elements.



```
Assignment 2: MeshEdit
EDGE DATA
address      = 0x3218790

halfedge()   = 0x2efaa80
isBoundary() = 0
Framerate: 50 fps
```

Before flipping edges



```
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 52 fps
```

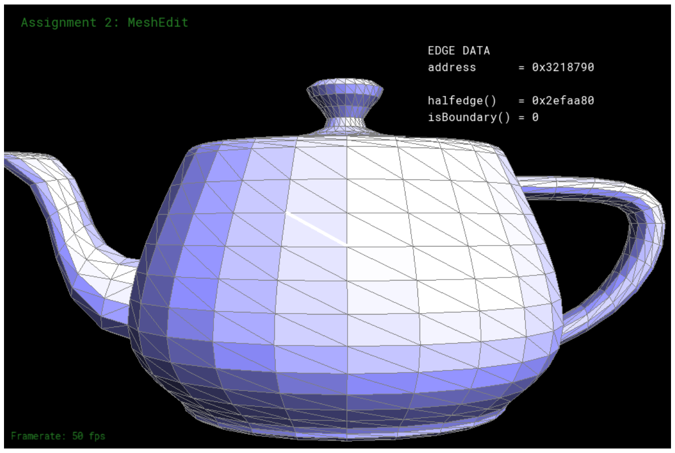
After flipping edges

Part 5: Half-edge split

The debugging process was similar to part 4's and diagrams were very useful to know which edges to create and which pointers should point where.

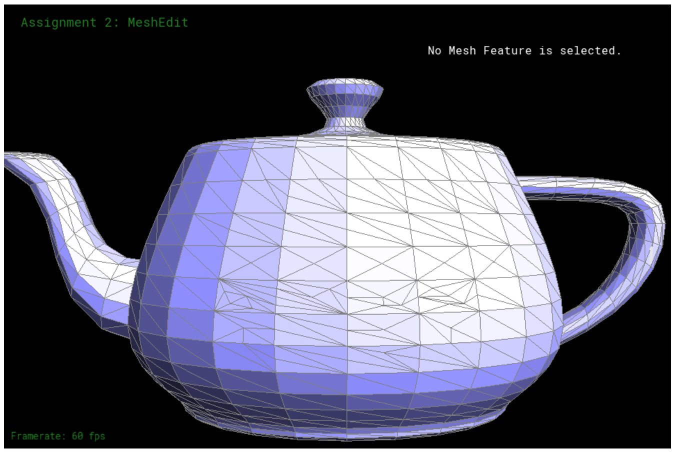
The first part in this section was to create 3 new edges, 3 faces, 6 halfedges, and a vertex and calculate the position of the new vertex (midpoint of the original edge).

Then the hardest part was to figure out the correct elements each pointer should be pointing to.



```
EDGE DATA
address      = 0x3218790

halfedge()   = 0x2efaa80
isBoundary() = 0
```

Before splitting edges

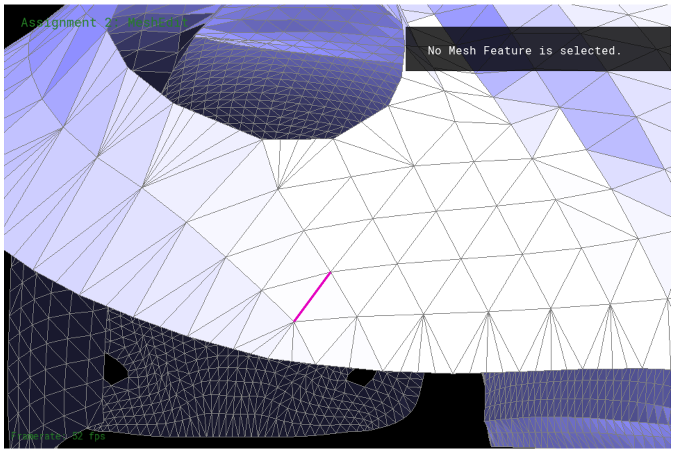


```
No Mesh Feature is selected.
```

After splitting edges

Splitting Boundary Edges

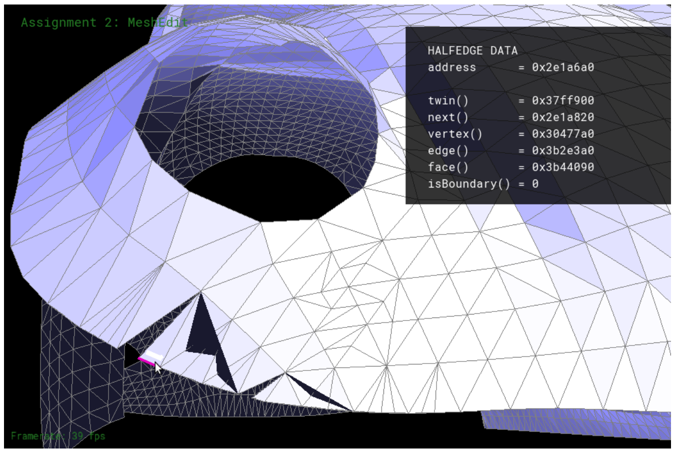
I implemented splitting boundary edges by using the "virtual face" implemented by the HalfEdgeMesh data structure. Because my original implementation creates extra edges and faces for splitting the virtual face into two, I deleted the extra edge that split the virtual face and also deleted the face created for the next half of the virtual face. I made sure that no existing structures were pointing to these and pointed the boundary halfedges to the right virtual face elements.



```
No Mesh Feature is selected.
```

After supporting boundary edges

Some bugs I got were similar bugs I got in the previous sections where faces would go black because the pointers were pointing at wrong elements. So I used gdb and check_for to find out which elements were incorrect. Here is an example of where things went wrong:



```
HALFEDGE DATA
address      = 0x2e1a6a0

twin()       = 0x37ff900
next()       = 0x2e1a820
vertex()     = 0x30477a0
edge()       = 0x3b2e3a0
face()       = 0x3b44090
isBoundary() = 0
```

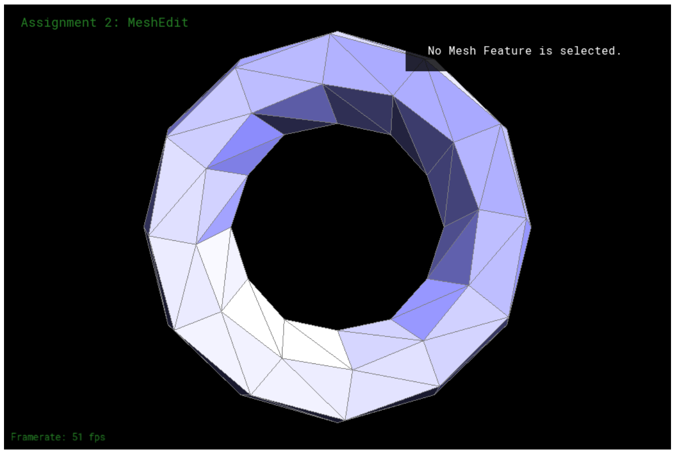
Boundary Edge bugs

Part 6: Loop subdivision for mesh upsampling

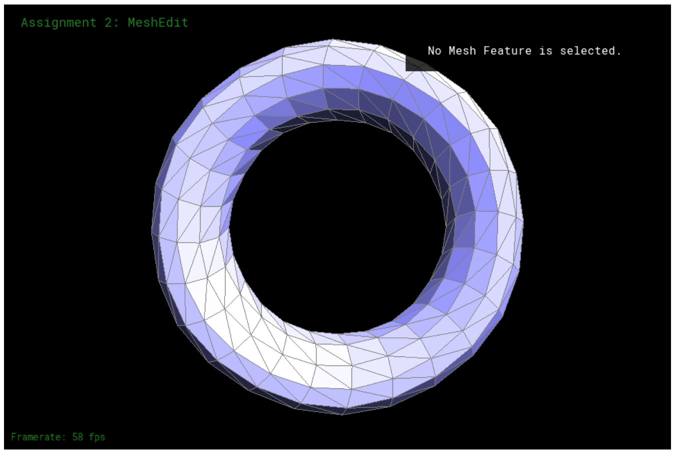
Implementing loop subdivision required a lot of different steps. First, for each vertex in the mesh, I found the sum of all neighboring vertices' positions and used that to calculate the vertex's new position, which I stored in newPosition for later use. Then I iterated through all edges to find the position for the new vertex that split that edge - and stored that vertex position in edge.newPosition. Next, I called splitEdge on all old edges. I had to change splitEdge to mark newly created edges as new so it wouldn't get stuck in an infinite loop, splitting again and again on the new edges. Since splitLoop returns a vertex v, I set v->isNew = true and v->newPosition = edge->newPosition. Then, in another loop, I iterated over all edges again and for all new edges that connected an old and new vertex, I called flipEdge on it. Finally, I reassigned all the vertices' positions to newPositions get the final mesh.

One problem I had was that I wasn't getting the correct new vertex positions because I had used an int instead of a float. I found the problem while debugging and saw that the vertex positions weren't being updated correctly.

Loop subdivision slightly shrinks the mesh and smooths out sharp corners and edges.

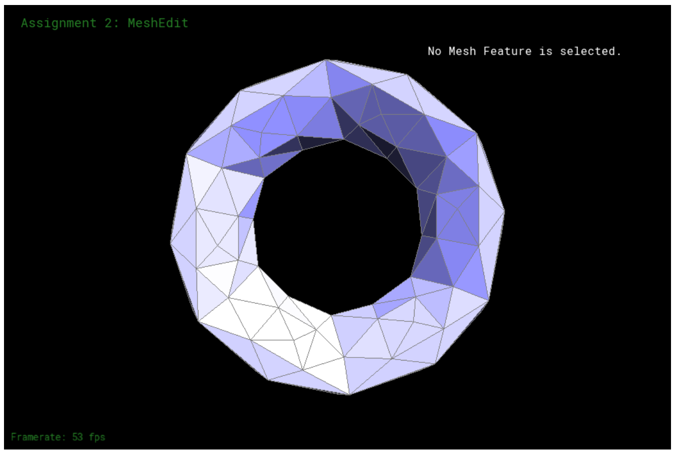


One step of loop subdivision

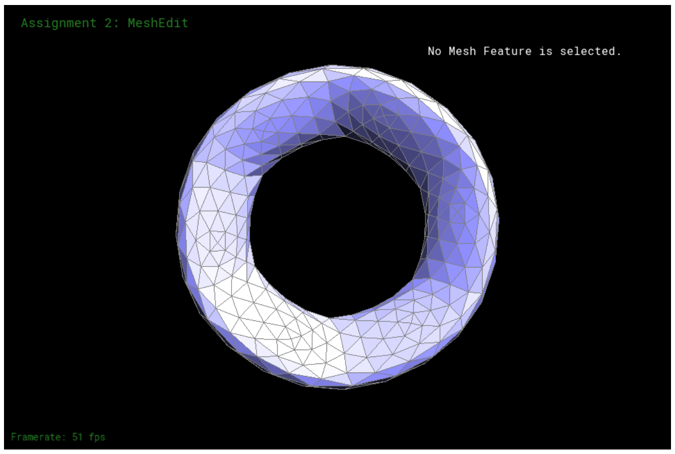


Two steps of loop subdivision

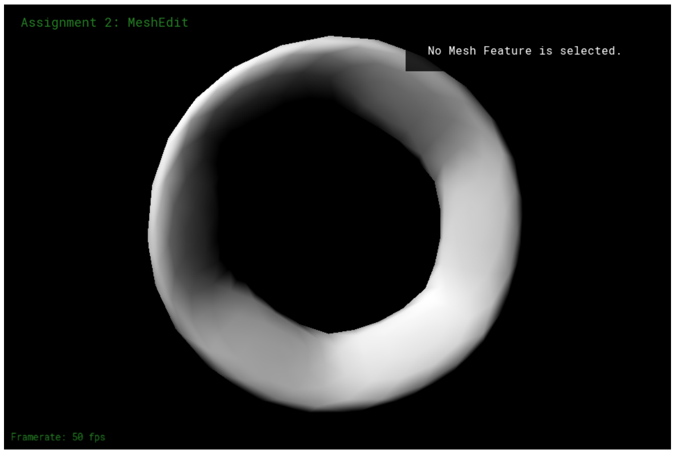
The original blocky mesh becomes smooth and round.



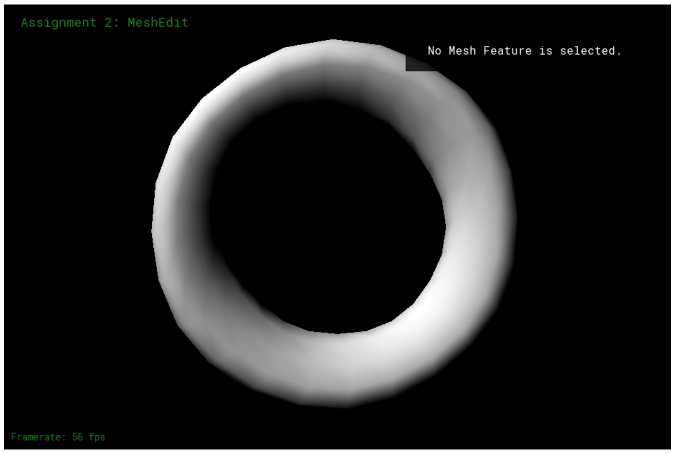
One step of loop subdivision (Pre-Split)



Two steps of loop subdivision (Pre-Split)



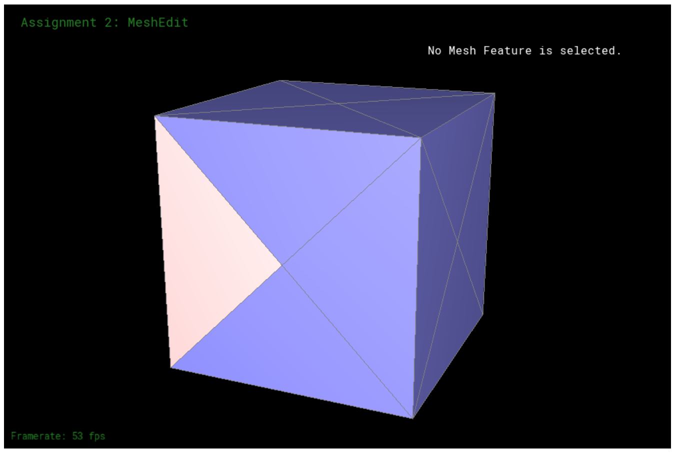
Average Normals (Pre-Split)



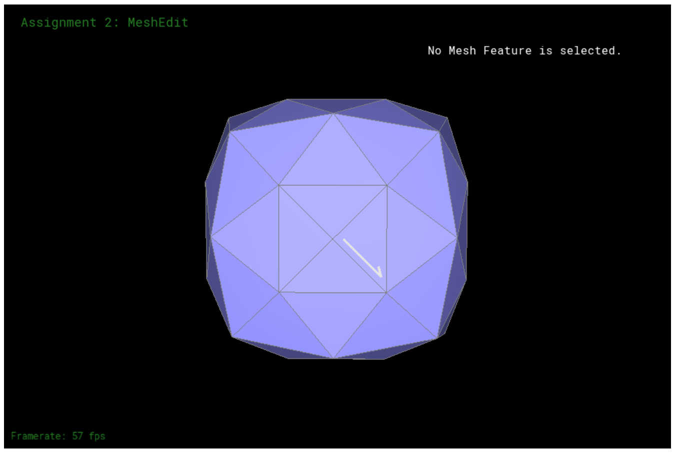
Average Normals (Not pre-split)

Pre-splitting doesn't lessen the effect but makes it even smoother and regular.

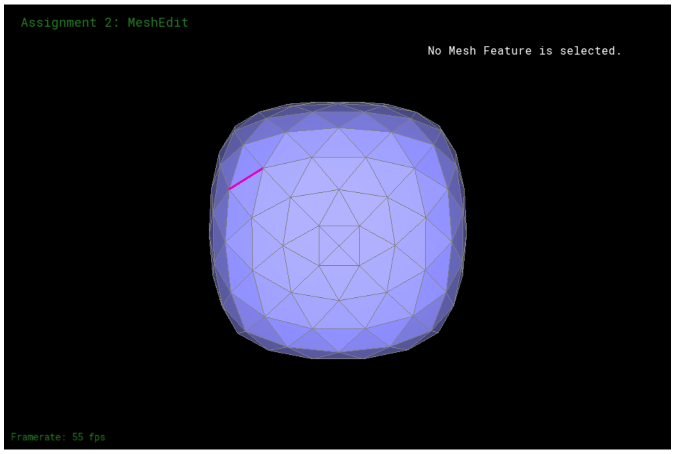
You can pre-process the cube by splitting all the diagonal edges on all 6 faces to make the cube more symmetrical when you apply loop subdivision.



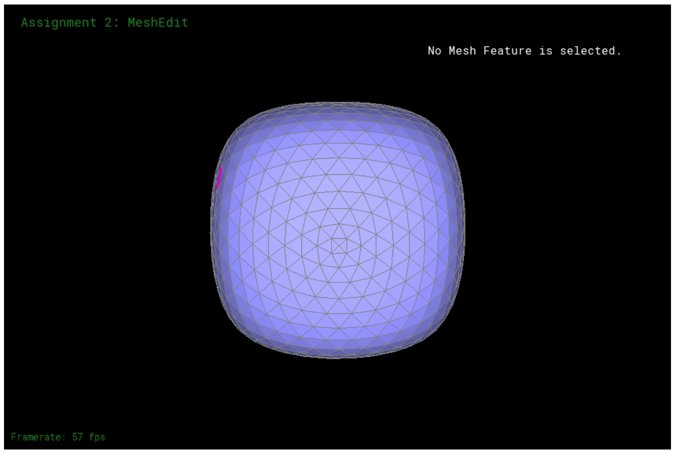
Split all edges before upsampling



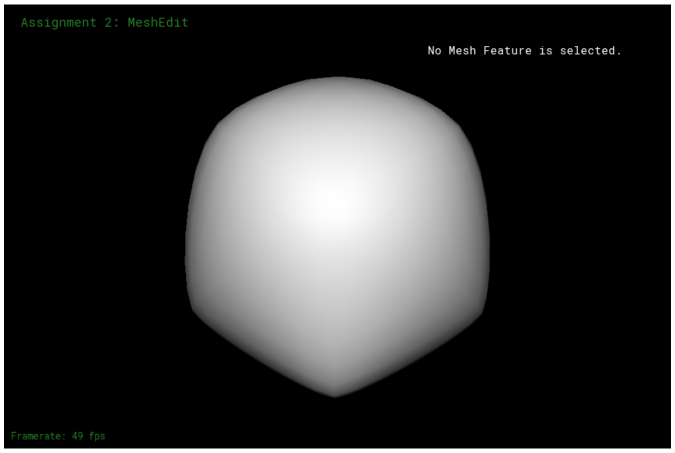
Loop subdivision step 1



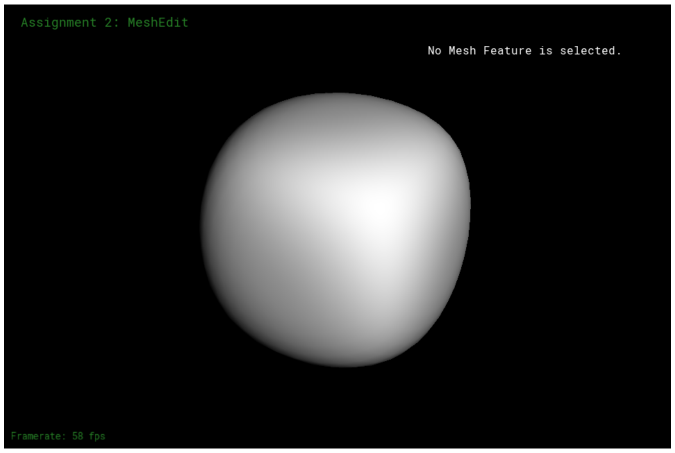
Loop subdivision step 2



Loop subdivision step 3



Average normal after 3 upsamplings (no pre-processing)

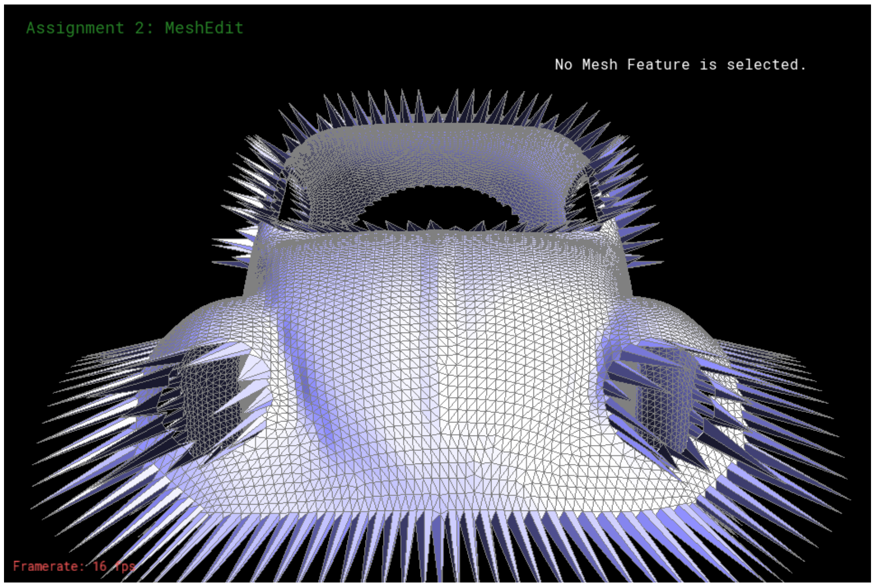


Average normal after 3 upsamplings (w/ pre-processing)

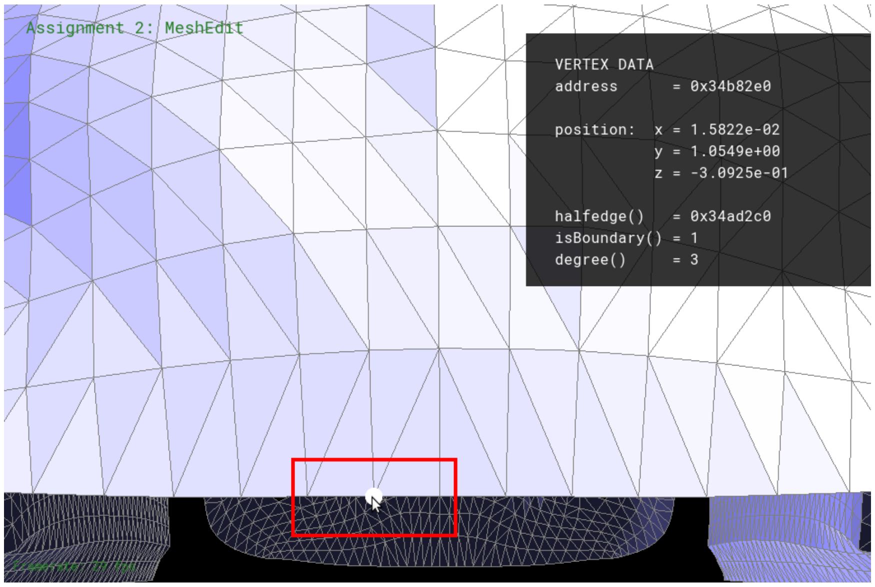
It's clear that splitting edges befor applying loop subdivision makes the resulting cube more symmetrical. Splitting the edges makes the original cube have symmetrical faces (diagonals on connecting all vertices of faces instead of just one diagonal connecting 2 vertices), hence the result.

Boundary Edges

Without having checks for boundary vertices, I got this:



Then, I noticed that the boundary vertices' degrees were 1 less than what I thought they should have been:

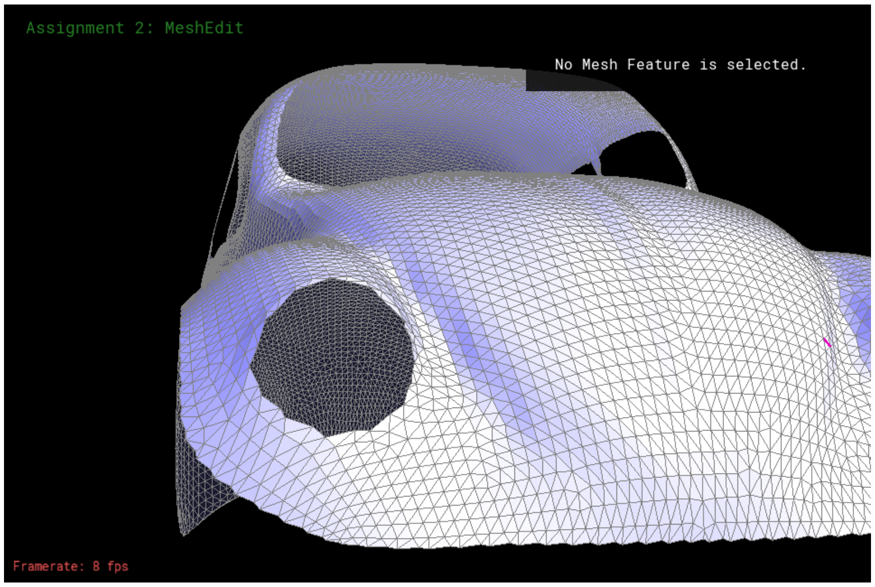


```
VERTEX DATA
address       = 0x34b82e0

position:  x = 1.5822e-02
           y = 1.0549e+00
           z = -3.0925e-01

halfedge()    = 0x34ad2c0
isBoundary()  = 1
degree()      = 3
```

So in my implementation, if a vertex was a boundary vertex, I added 1 to its degree when calculating its newPosition:

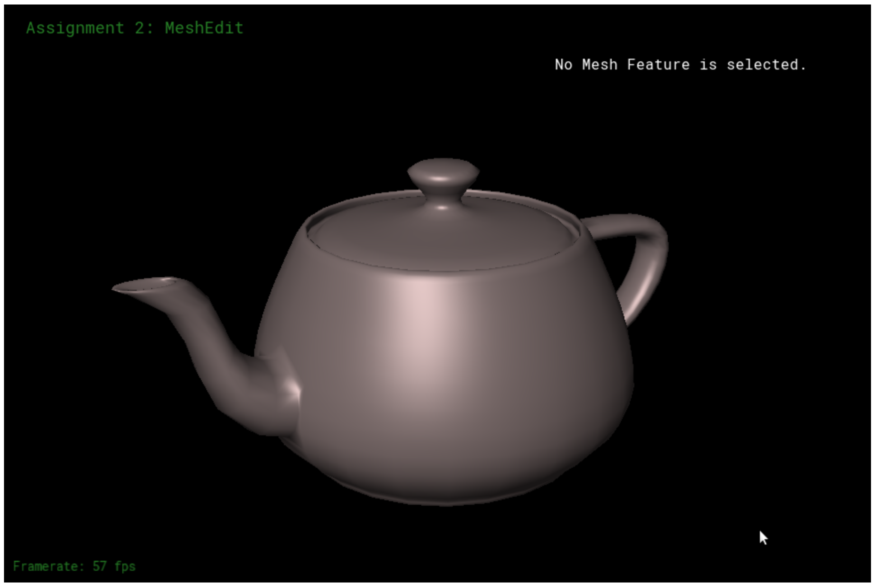
```
Size n = v->degree();
if (v->isBoundary()) n += 1;
float u = (n == 3) ? (3.0/16.0) : (3.0 / (8.0 * n));
v->newPosition = (1 - n * u) * v->position + u *(neighbor_pos_sum);
```

I'm not too sure if this was a right way but I couldn't find the part about boundary vertices in the link provided in the spec. But I think this turned out okay.

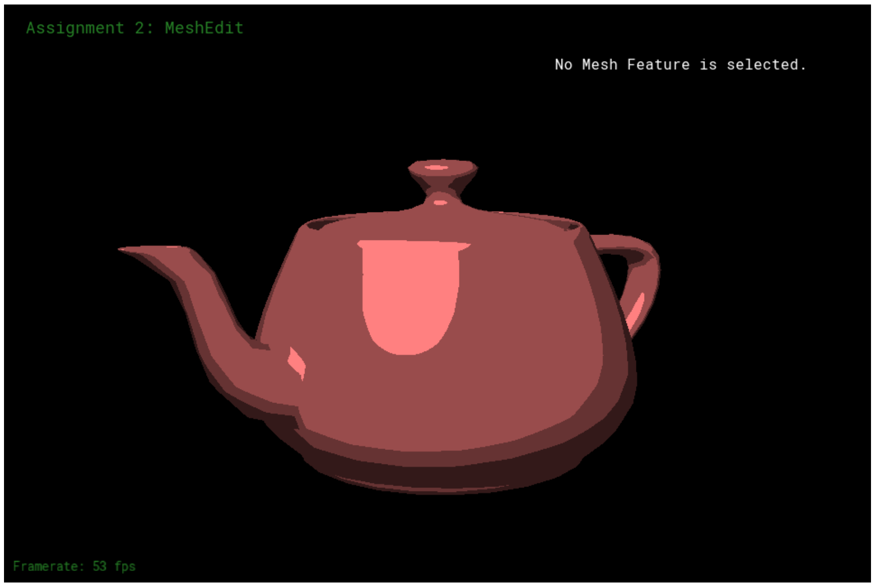


Section III: Shaders

Part 7: Fun with shaders



Phong Shading



Toon Shading

Section IV: Mesh Competition

If you are not participating in the optional mesh competition, don't worry about this section!

Part 8: Design your own mesh!