Bezier Surfaces and Triangle Meshes

by Jonah Bedouch and Brandon Wong

2025-03-07

This report is also available online at https://cs184-zen.vercel.app/projects/hw02.

The GitHub repository for this project can be found at
https://github.com/cal-cs184-student/sp25-hw2-okay/

Project Overview

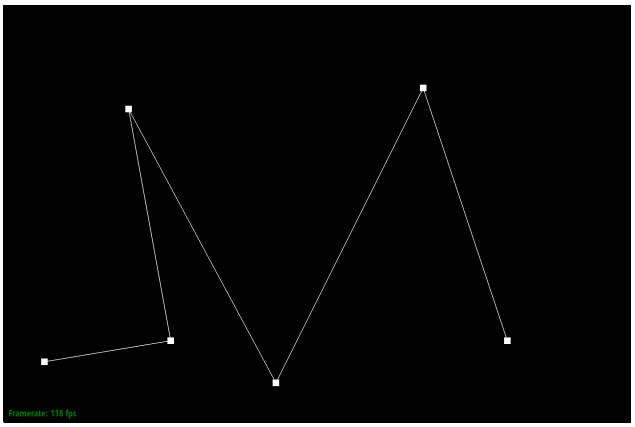
In this project, we worked with bezier curves and triangle meshes in order to learn different ways to render surfaces. We began by simply implementing the algorithm that allows us to render a bezier curve, then we extended this to cover bezier surfaces, a way of using bezier curves in order to render entire 3D models. From there, we pivoted to a more advanced meshing method, triangle meshes. Using a half-edge data structure that was provided to us, we implemented a function to find the area-weighted vertex normal of every vertex, allowing us to effectively do Phong shading. After this, we implemented a basic local mesh operation called edge flipping, which allowed us to flip the direction of any one edge in order to improve the feel of our triangle mesh locally. After this, we implemented the Edge Split operation, which adds a vertex at the midpoint of an edge, and then creates new triangles by bridging this new vertex to nearby existing vertices with new edges. Finally, we combine all of the work that we've done with triangle meshes to implement mesh subdivision, a global technique of improving the quality of a mesh by subdividing it into smaller triangles whose areas are weighted dependent on the specific shape of the mesh before subdivision. As it turns out, implementing this operation primarily consists of some clever math and the use of our local optimizations, which allowed us to take our local improvements and use them to provide a global improvement in mesh quality.

Section 1: Bezier Curves and Surfaces

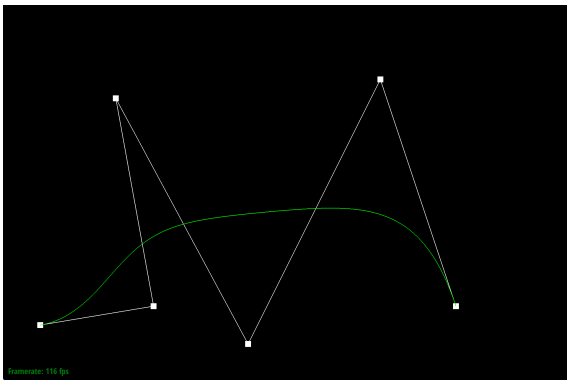
Part 1: Bezier Curves with 1D de Casteljau Subdivision

The way de Casteljau's algorithm works at each step is by looping through all the available points at the current level and finding the point between each pair of points in order. This means between the first and second points, second and third points, and so on until the second to last and last points. This point between each pair of points would be weighted by values 1 - t for the first one and t for the second one, with t depending on how far on the generated bezier curve the point you are searching for is at a value between 0 and 1, with 0 being the beginning of the curve and 1 being the end of the curve. Part 1 is the implementation of this step, which then gets run on loop at each level to find the corresponding point t on the curve. This repeated running

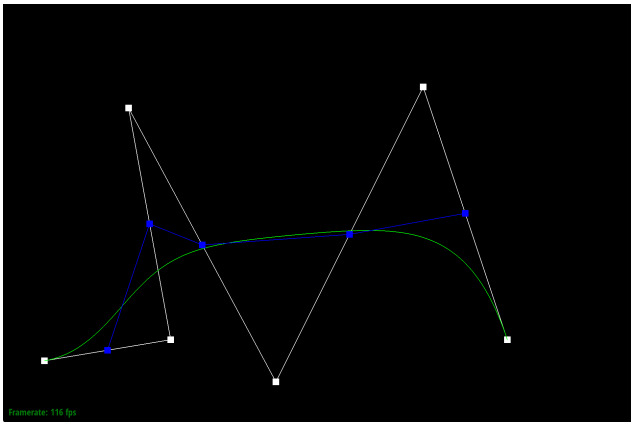
on each level is then run for each point in the curve to find the overall bezier curve. The results of a custom curve are shown below.



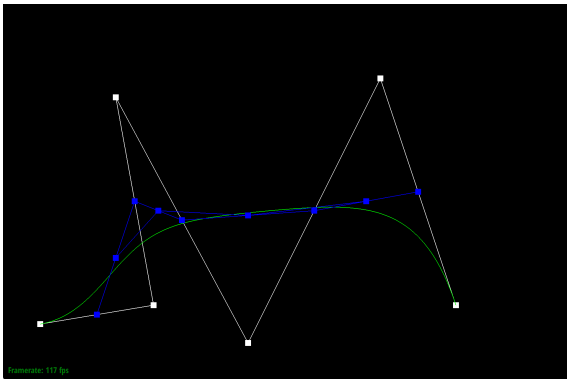
Custom Curve Initial Points



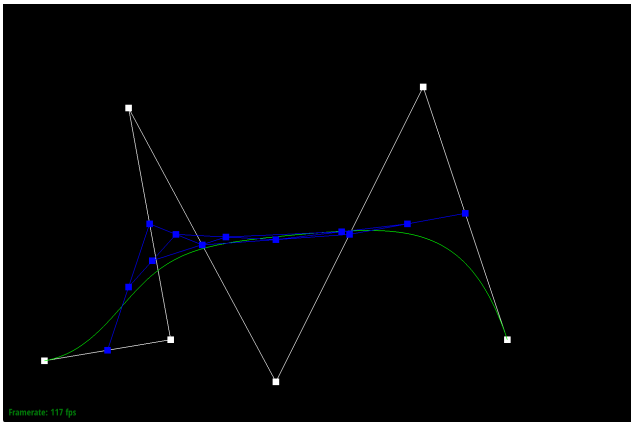
Custom Curve Step 1



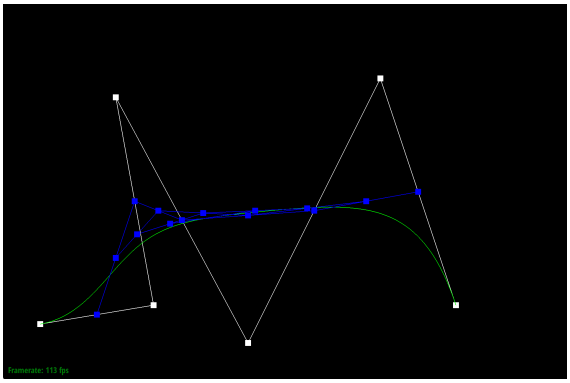
Custom Curve Step 2



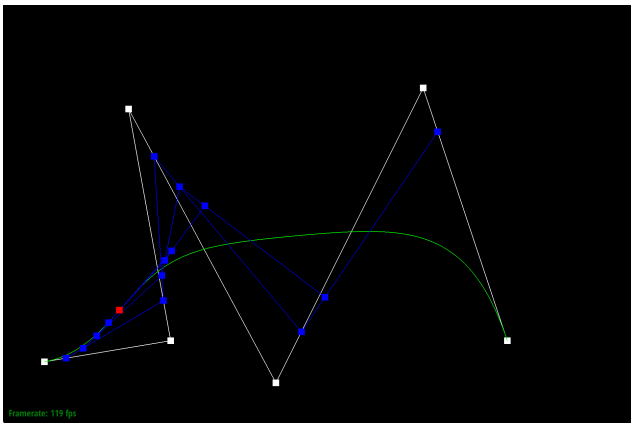
Custom Curve Step 3



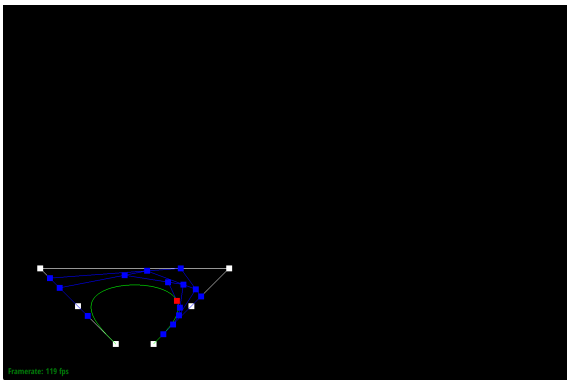
Custom Curve Step 4



Custom Curve Step 5



Custom Curve Step 7



Different Custom Curve

Part 2: Bezier Surfaces with Separable 1D de Casteljau

By classifying the points into rows and columns, de Casteljau's algorithm can be implemented on 3D points as well as 2D points. First, the bezier curve of each row can be found individually. Then by taking one point per row where each point is in the same column, the actual bezier curve can be found for each column, resulting in a 3D bezier surface. This was split into 3 parts in the actual implementation, one to find the resulting points in each step similar to the step function in the previous section, one to find the vector of 3D points that make up one bezier curve, and one to find the final point from a bezier curve along the curves in the other direction generated previously. This enables the generation of the 3D surface teapot shown below.

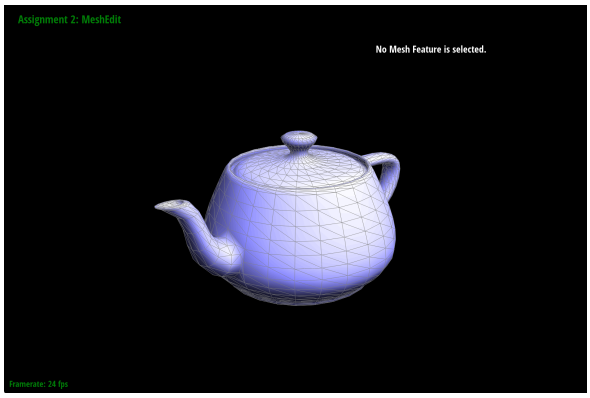
Bezier Teapot

Section 2: Triangle Meshes and Half-Edge Data Structure

Part 3: Area-Weighted Vertex Normals

To implement area-weighted vertex normals, we got the corresponding halfedge to the desired vertex, and then looped through until we reached the same vertex again. Each iteration, we grabbed the current face, got the face's normal, normlaized it, and then accumulated it into a variable called `face_normals`. After getting the face, we found the twin of the halfedge and then got the next value of that twin, since this would guarantee that we move on to the next face. At the very end, we divide the `face_normals` vector by the number of faces we ended up finding in our loop, and return this final number as our area-weighted vertex normal. We can see the result of this with Phong shading in the following images:

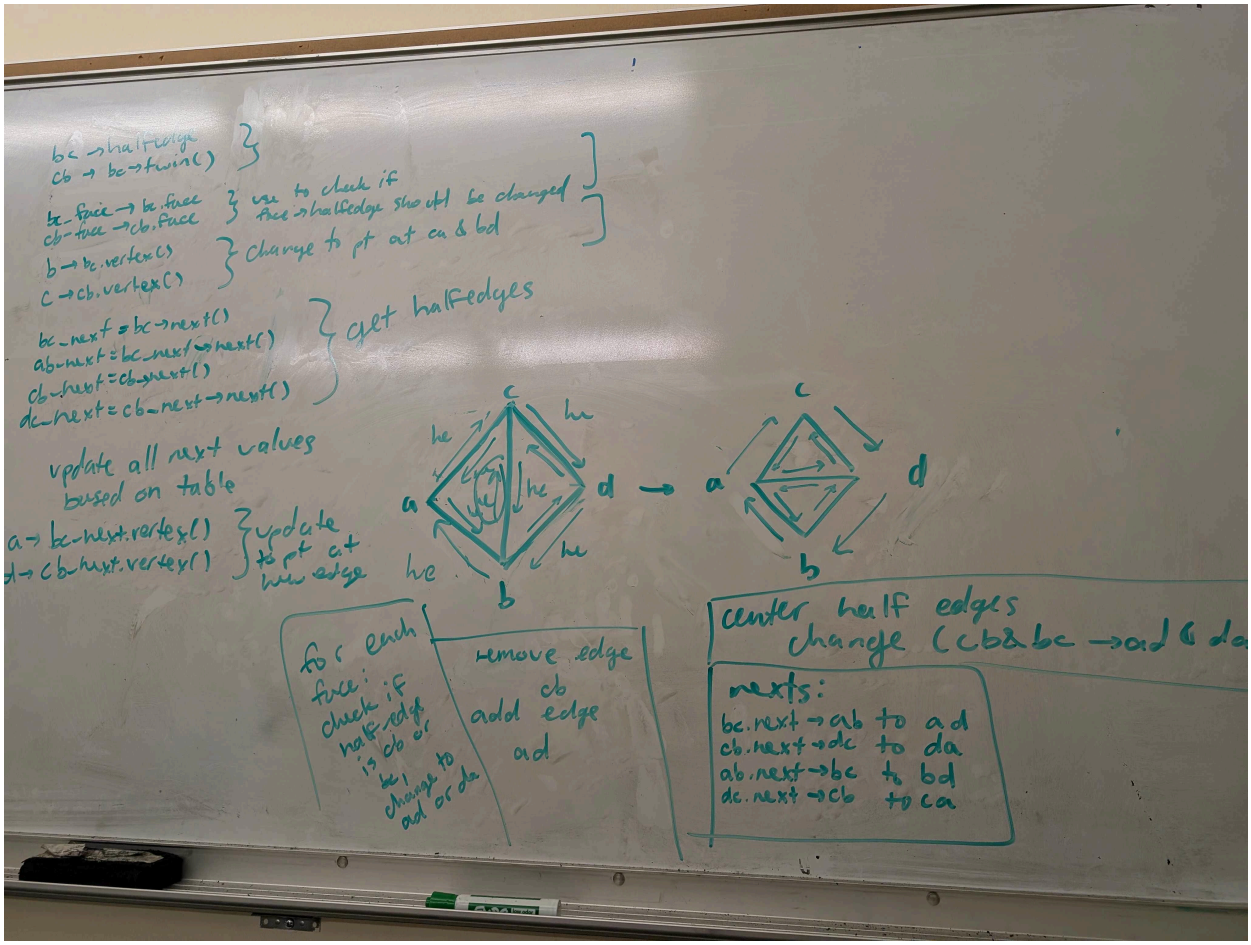
Teapot, flat shading



Teapot, Phong Shading

Part 4: Edge Flip

We implemented the edge flip operation by drawing out the necessary flips on a whiteboard, producing the following mess:



Whiteboarding out Part 4

Here, we drew out the initial and final set of vertices, edges, faces, and half edges. From there, we identified all of the changes that needed to happen to make a flip work out, and then pseudo-coded the flip on the whiteboard. Because of the level of scaffolding we did beforehand, the implementation was somewhat smooth, but a fun debugging story that we had is that we were incorrectly initializing our `EdgeIter` variables as `EdgeIter&`, which we spent rougly 3 hours debugging since all of our logic was otherwise correct, but the code would hang as a result instead of flipping, and we could not figure out why. We eventually asked someone on staff who noticed basically immediately.

Our implementation doesn't have a lot of interesting things, for the most part we just mapped out all of the changes and did them. The way that we made this easier to conceptualize is that we based all of our variables off of the `a`, `b`, `c`, `d` from our diagram, and made everything relative to the single half edge we circled (since no matter which half edge is selected, everything is the same relative to the selected edge).

The result is below:

Teapot without any flips

Teapot with a few flips

Teapot with many flips

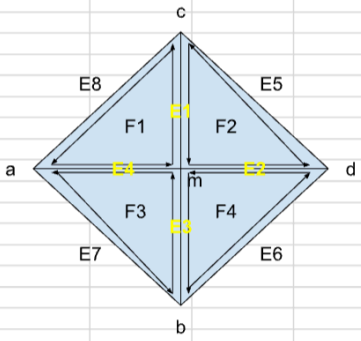
Part 5: Edge Split

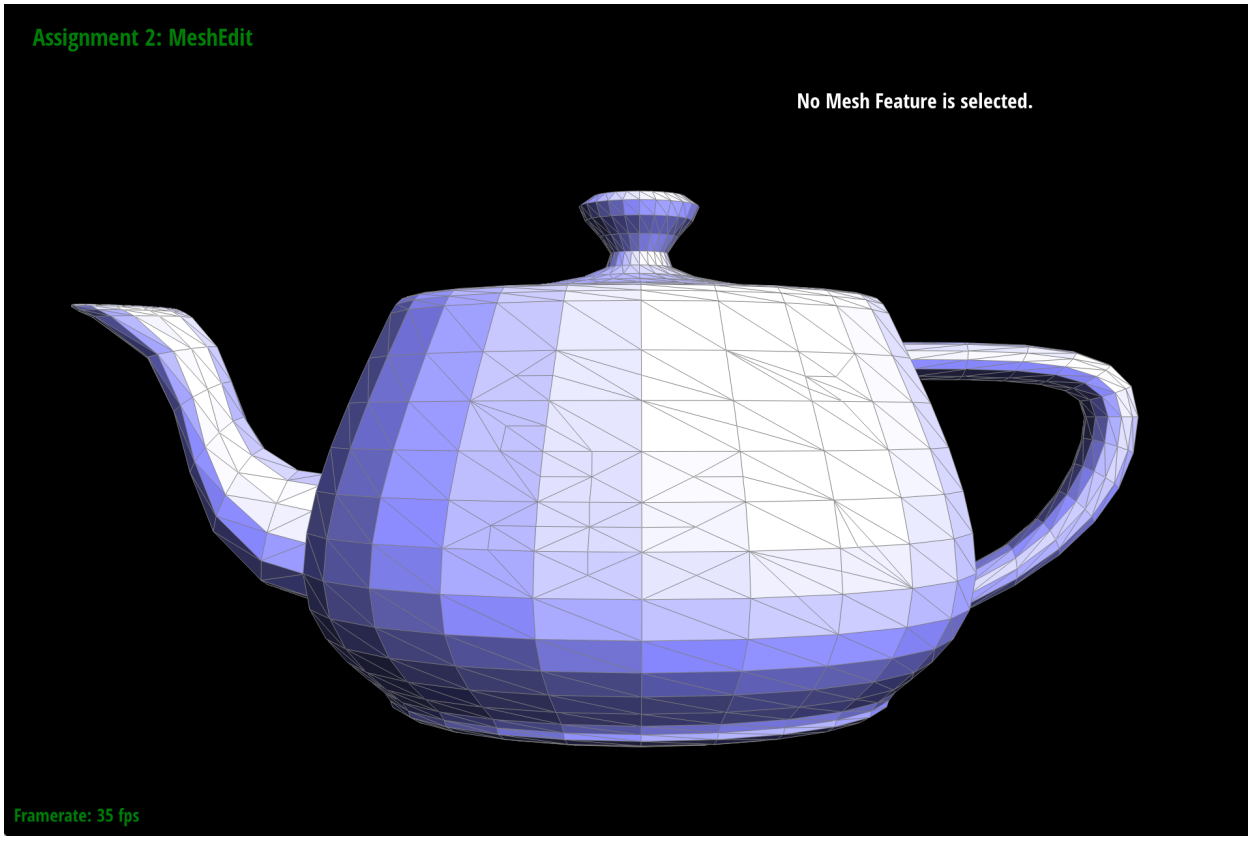
To do part 5, we created a spreadsheet containing each object we would be interacting with in the split, old and new. This means we put in all 12 of the inner halfedges, all 8 edges, all 5 vertices, and all 4 faces. For the half edges, we made a table showing what each half edge object contained both before and after the split. For all the other objects, we just put down what they would be pointing to after the split, as they are less complicated than the halfedges. This made implementation simple. First, we obtained all of the relevant old objects from the initial edge used as input. Then, We created all the new objects we would need. Finally, we assigned the new pointers to each object such that they pointed to what they would need to after the split. For the non half edge objects, we assigned them as they were obtained and generated once all the values that needed to be obtained from them were obtained as they are much simpler than the half edge objects. The table is shown below.

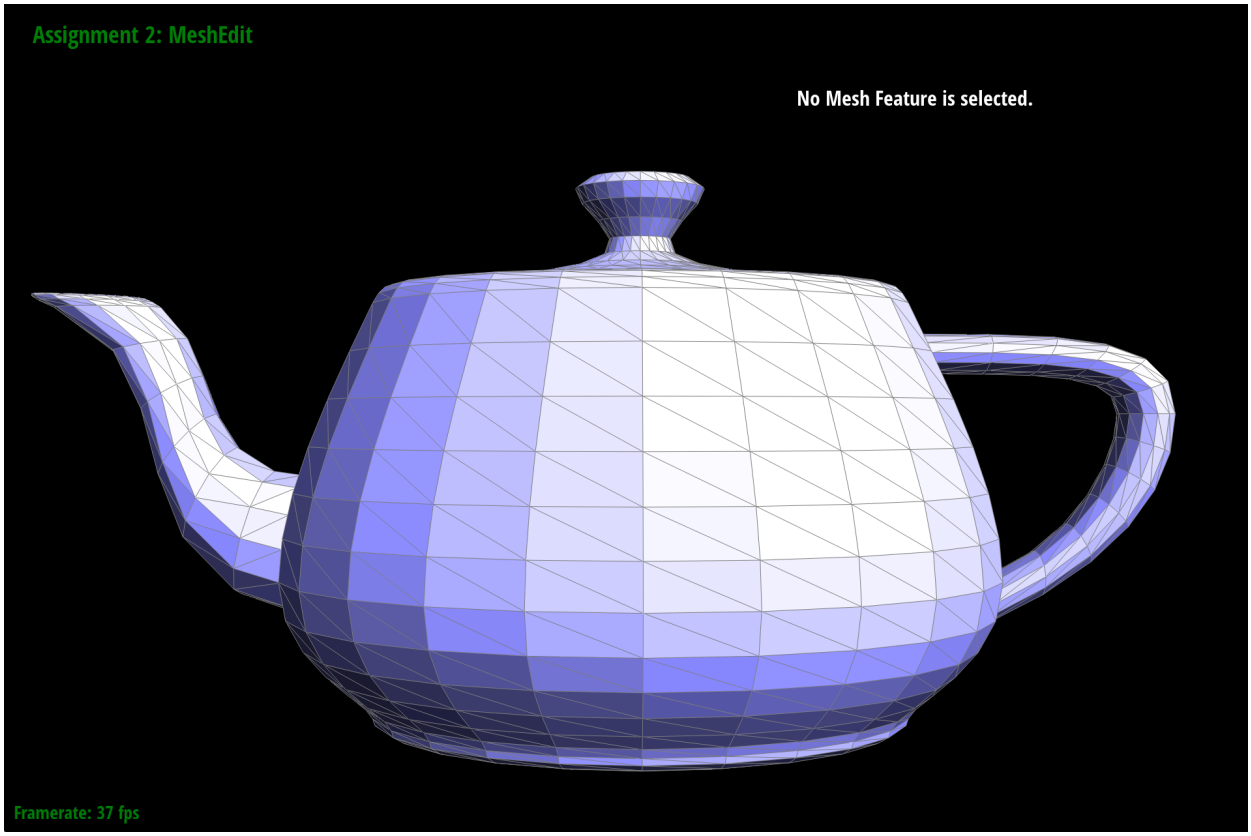
	Before					After					
Halfedge	**Next**	**Twin**	**Vertex**	**Edge**	**Face**	**Next**	**Twin**	**Vertex**	**Edge**	**Face**	**Halfedge**
am	DNE	DNE	DNE	DNE	DNE	mc	ma	a	E4	F1	am
ma	DNE	DNE	DNE	DNE	DNE	ab	am	m	E4	F3	ma
bm	DNE	DNE	DNE	DNE	DNE	ma	mb	b	E3	F3	bm
mb	DNE	DNE	DNE	DNE	DNE	bd	bm	m	E3	F4	mb
cm	DNE	DNE	DNE	DNE	DNE	md	mc	c	E1	F2	cm
mc	DNE	DNE	DNE	DNE	DNE	ca	cm	m	E1	F1	mc
dm	DNE	DNE	DNE	DNE	DNE	mb	md	d	E2	F4	dm
md	DNE	DNE	DNE	DNE	DNE	dc	dm	m	E2	F2	md
ab	bc	ba	a	E7	F1	bm	ba	a	E7	F3	ab
bd	dc	db	b	E6	F2	dm	db	b	E6	F4	bd
dc	cb	cd	c	E5	F2	cm	cd	c	E5	F2	dc
ca	ab	ac	d	E8	F1	am	ac	d	E8	F1	ca
bc	bd	cb	c	E0	F1	DNE	DNE	DNE	DNE	DNE	bc
cb	ca	bc	b	E0	F2	DNE	DNE	DNE	DNE	DNE	cb

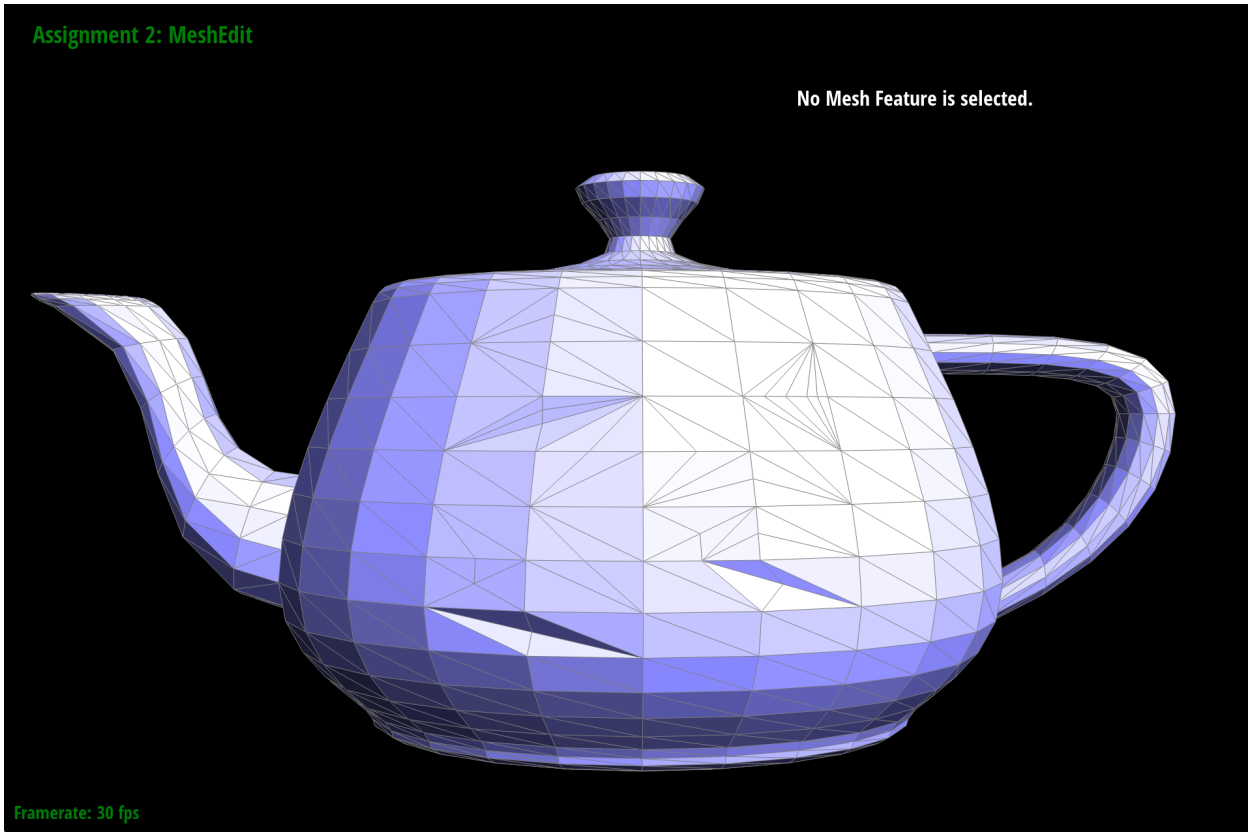
After:

Vertex	**Halfedge**	**Edge**	**Halfedge**	**Face**	**Halfedge**
a	ab	E1	mc	F1	ca
b	bd	E2	md	F2	dc
c	ca	E3	mb	F3	ab
d	dc	E4	ma	F4	bd
m	mb (along split edge)	E5	unchanged		
		E6	unchanged		
		E7	unchanged		
		E8	unchanged	e0 becomes e3	
				bc becomes bm	
If there is a boundary, there is no d vertex				cb becomes mb	

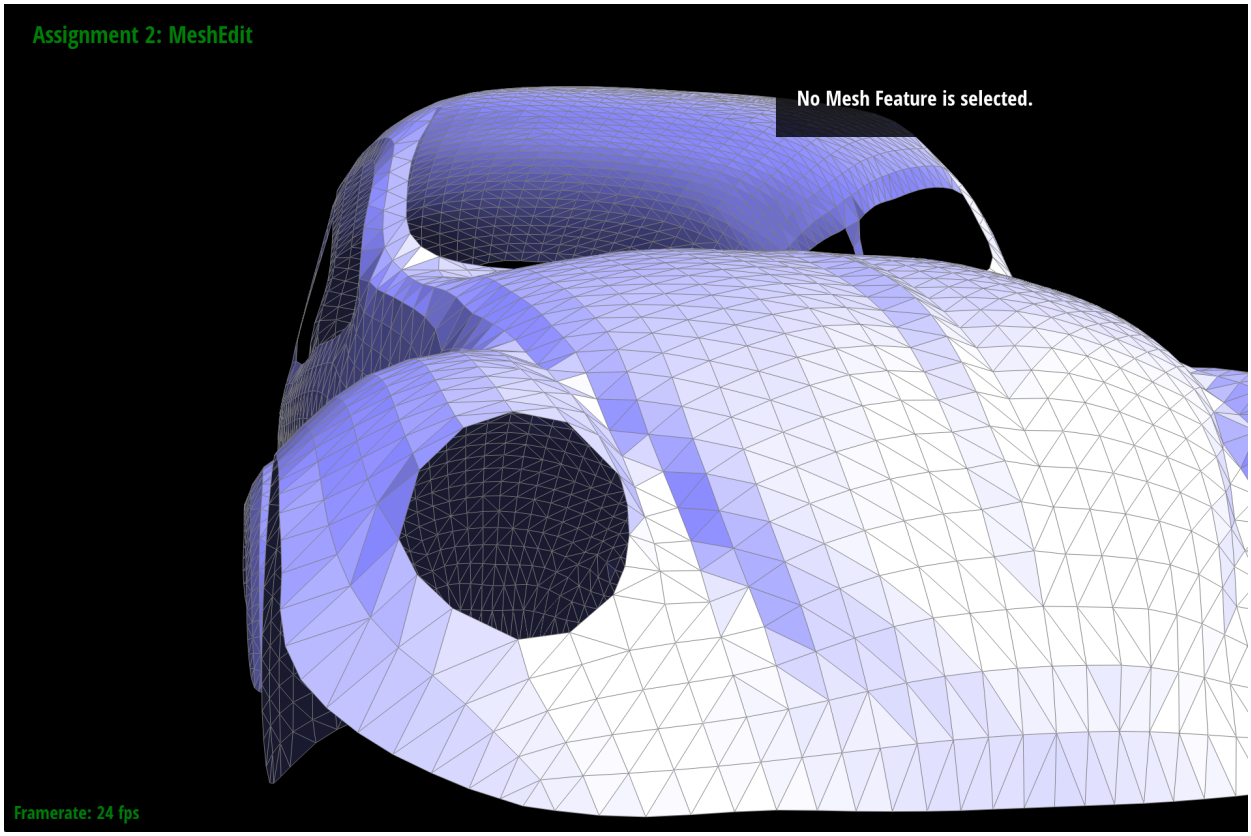


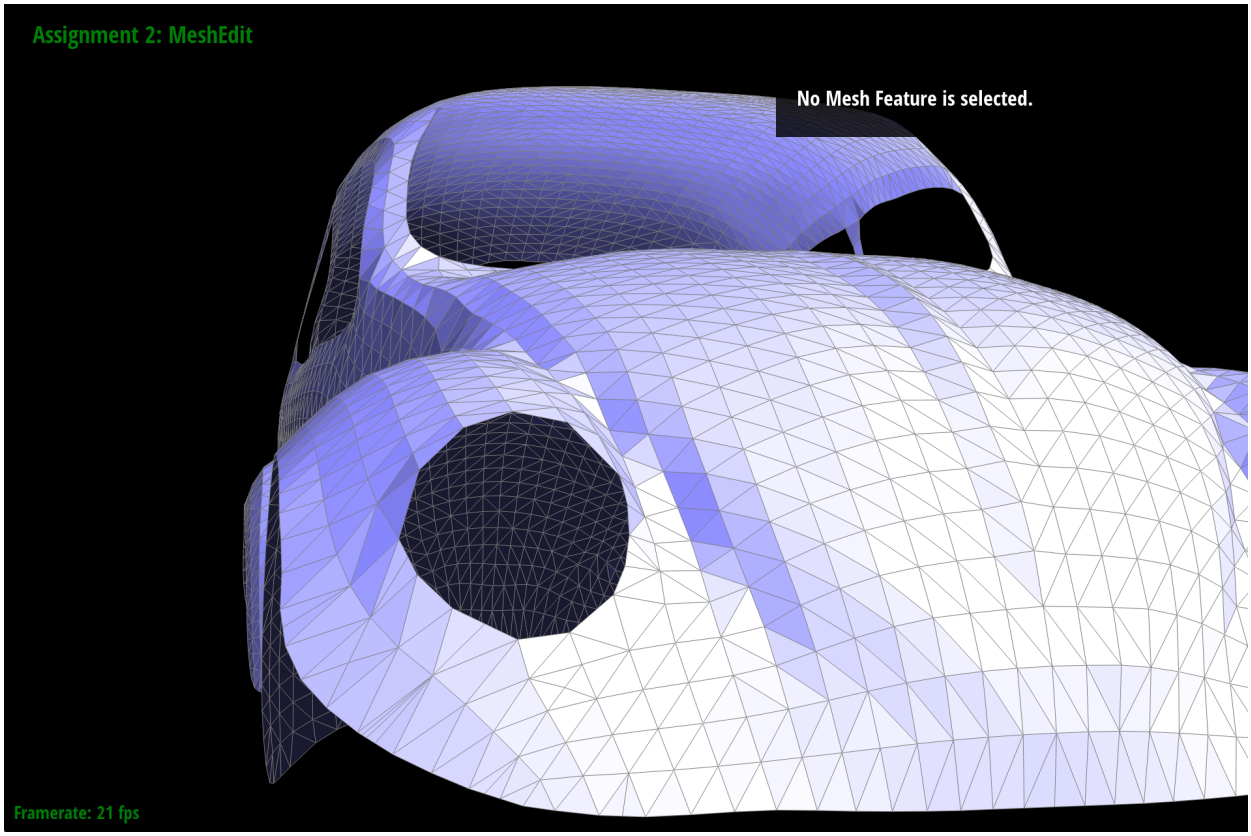

Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 35 fps


Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 37 fps


Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 30 fps

To handle boundary edges, we added a checker that would figure out which half edge was the boundary edge (if there was one) and would assign the non-boundary edge to the same variable each time, which meant we would not have to keep track of it later. Then, we set the objects that would be on the boundary side if the edge was a boundary edge to point to boundary things by default, and then set if statements that would have them point to the normal things if the edge was not a boundary edge. These results are shown below.


Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 24 fps

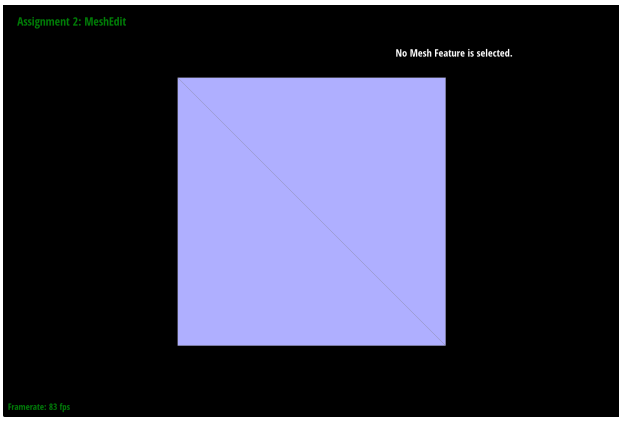

Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 21 fps

Debugging was both simpler and odder than in part 4. For this part, I kept getting a segfault after the function would finish running that would crash everything. I tried to use print statements throughout the code to check where the error was, which was how I detected the segfaults. In the process, I realized that I was assigning a couple of things wrong, but it turns out those were not causing the segfault. I also switched from deleting things to just reassigning, but again, that wasn't what was causing the issue (although it did help for the next part). Finally, I realized I was creating new objects wrong, as I had not realized there were dedicated new functions. Thankfully, there were no more issues after that (except a wrong face that would not be noticed until part 6 due to not causing any obvious issues).

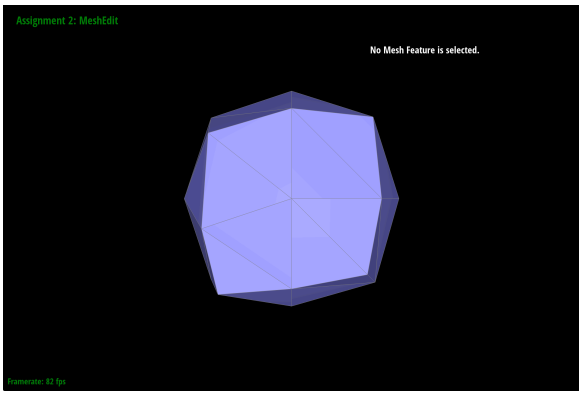
Part 6: Loop Subdivision for Mesh Upsampling

To implement loop subdividing, we rely on a few loops. First, we loop through the vertex set and the edge set in order to set all of the flags about new meshes to false. Then, we loop through the vertex set and calculate all of the new positions according to where all of the adjacent vertices are located (we follow the given formula from the spec). Then, we loop through the set of edges and base the position of the edge splits on the four surrounding vertices to the edge. We note both of these in the `newPosition` fields of the vertex and edge structs respectively. Next, we add all of our current edges to an array, which allows us to iterate through them without worrying about the addition of new verticies (we tried to do this in place, but we kept running into errors and eventually switched to this method to rule out any chance of error as a result of the loop). As we loop through all of the old edges, we split the edge, mark the new vertex as new, and update the `newPosition` of the vertex to match the `newPosition` that was on the edge. Then, we adjust the `isNew` value of the corresponding vertices that were created. We make sure only to mark the vertices which are not alonog the preexisting line as new, so that in our next iteration there is no chance that we attempt to flip them. Next, we loop again through our edges, flipping any edge that is marked as new and has at most one of its two vertices marked as new. Finally, we loop through all of the verticies and copy `v->newPosition` into `v->position`. Since we set the vertex position to the edge position earlier, we can just do this in a single pass.

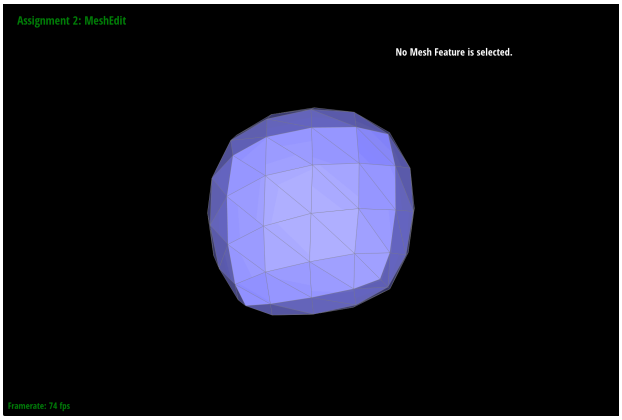
The result of loop subdivision is that vertices in the middle that are not along jagged edges smooth out very quickly, while jagged edges or corners take longer to resolve themselves. We can see this in the result of the cube, where the more iterations we go, the fewer vertices are actually along a jagged line, resulting in more uniformity, but in the first few iterations we see something that's quite spikey on the outside and more okay in the middle:



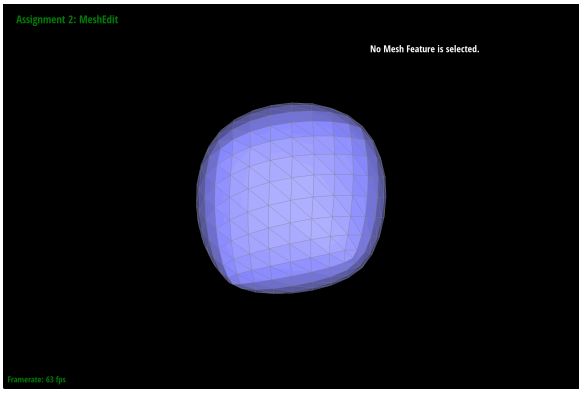
Box, no subdivisions



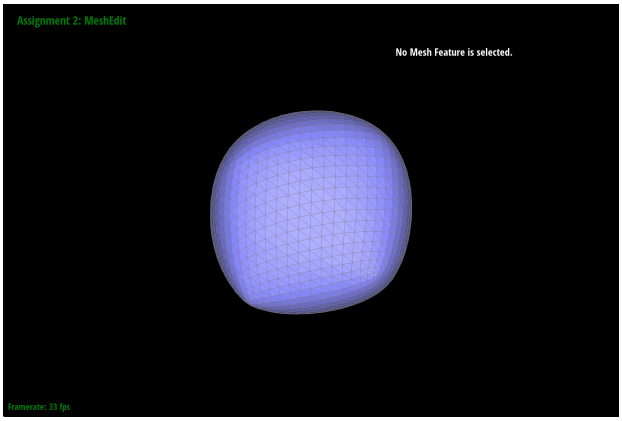
Box, 1 subdivision



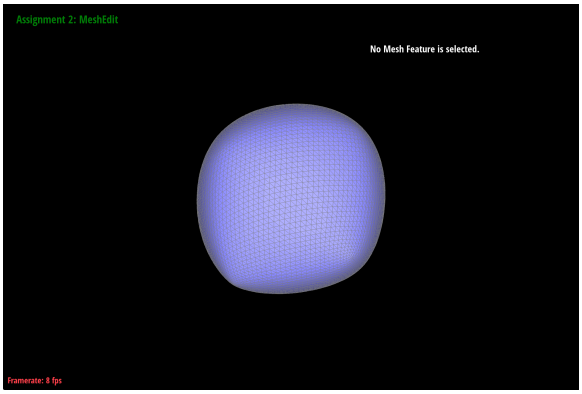
Box, 2 subdivisions



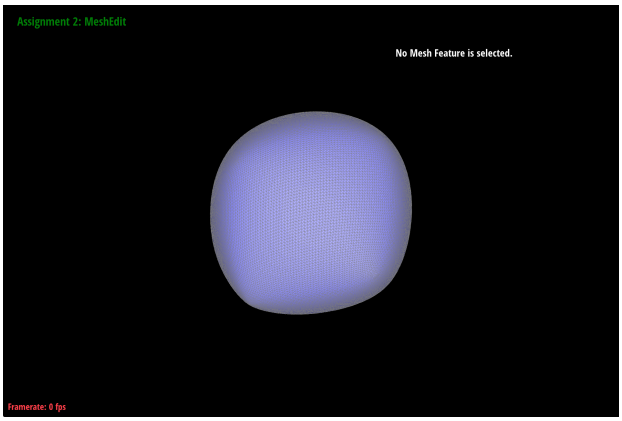
Box, 3 subdivision



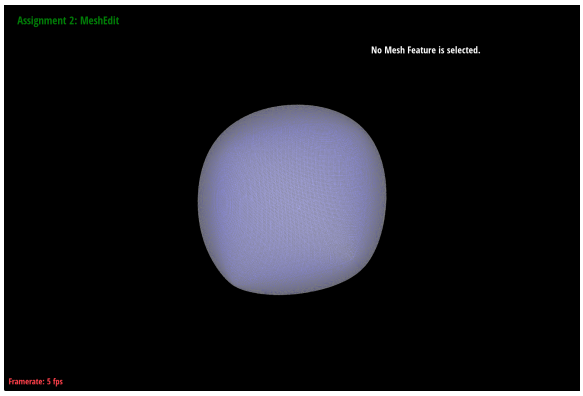
Box, 4 subdivisions



Box, 5 subdivision

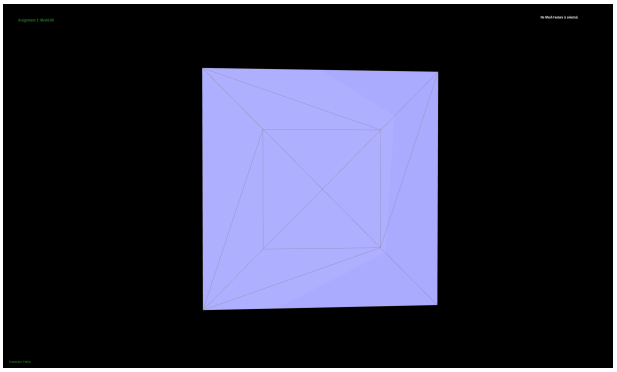


Box, 6 subdivisions

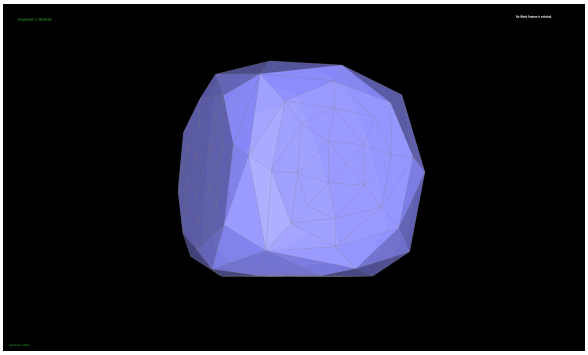


Box, 7 subdivision

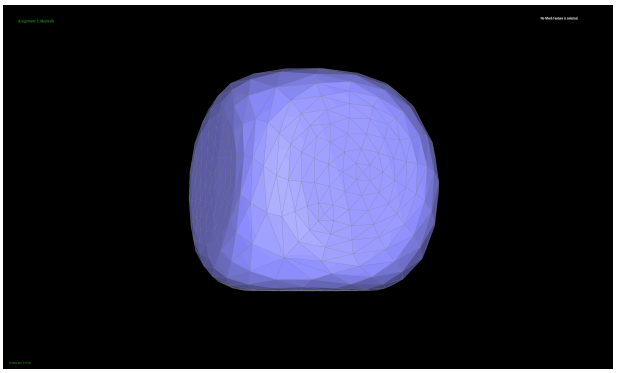
We notice that this effect can be minimized to an extent by adding more subdivisions near the edges themselevs. For instance, in the box case, we can add the following pattern and observe the result after 3 iterations:



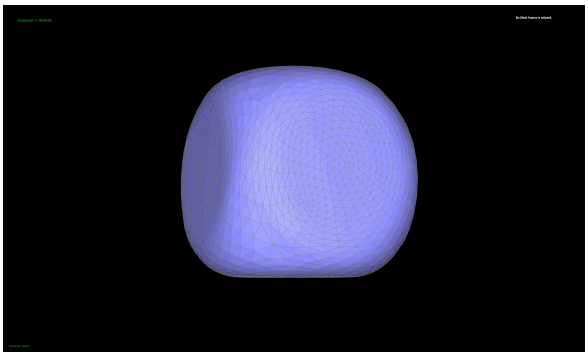
Box, split pattern



SP Box, 1 subdivision



SP Box, 2 subdivisions

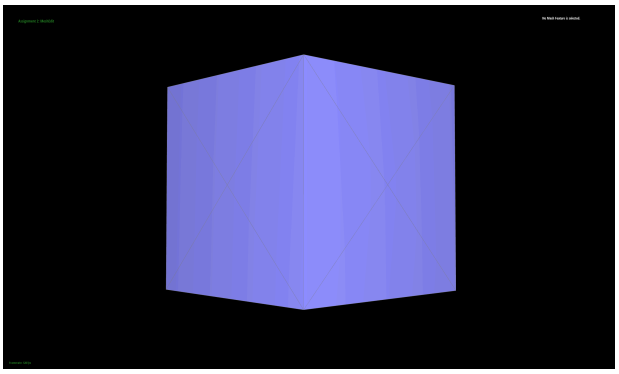


SP Box, 3 subdivision

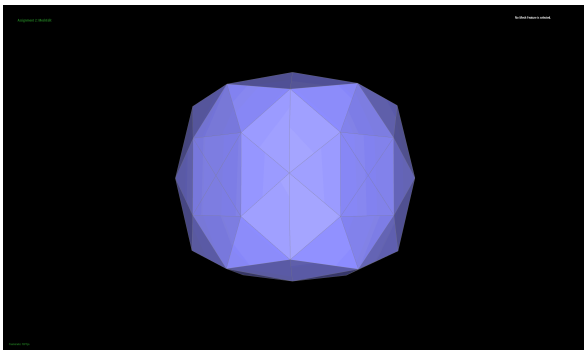
We see that the each iteration is much faster to conform to a rounded pattern, and by iteration 3 it is hard to notice any chunkinng (which we still could see visibly on the first one). This is because we have reduced the number of sharp edges, so when we subdivide, the vertices that we average to find the position of new vertices are no longer following a single line that skews the

data in one direction. Instead, there is some variance resulting in vertex placement that looks more natural and conforms to the subdivision faster.

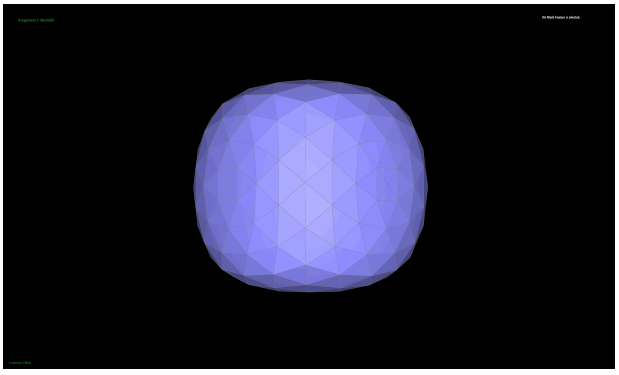
Now, in terms of allowing the cube to subdivide symetrically, we observe in the above set of images that the cube is skewed along one axis. If we pay attention, we see the result is a more oblong cuboid on the vertices where there is not directly an edge bisecting them. This makes sense, because the vertex that results from the split and flip operation will be weighted along the closer vertices in the corners where there is actually an existing edge, and will not on the other edges, resulting in an oblong appearance. We can easily resolve this by making the mesh edges symmetrical. That is, by splitting the one edge that runs through the cube into a cross pattern, we can force the new vertices to be evenly spaced along all four corners, resulting in a symmetric cube. This effect can be seen below:



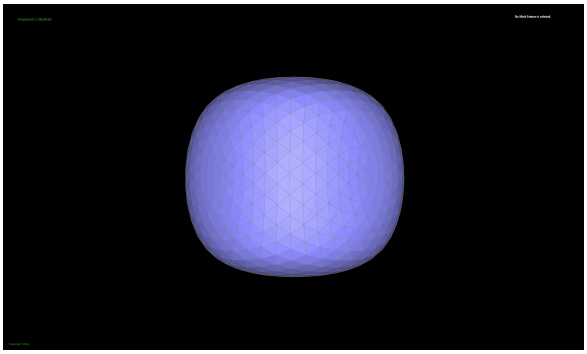
Symmetric Box, no subdivisions



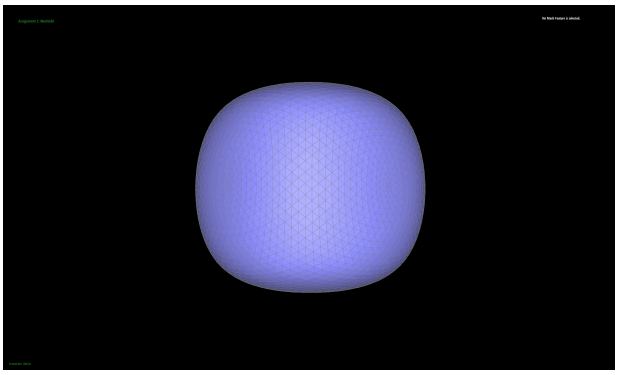
Symmetric Box, 1 subdivision



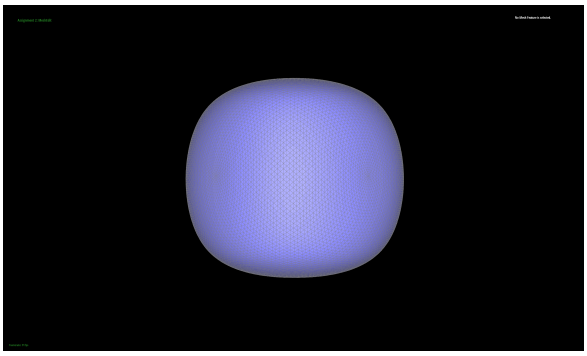
Symmetric Box, 2 subdivisions



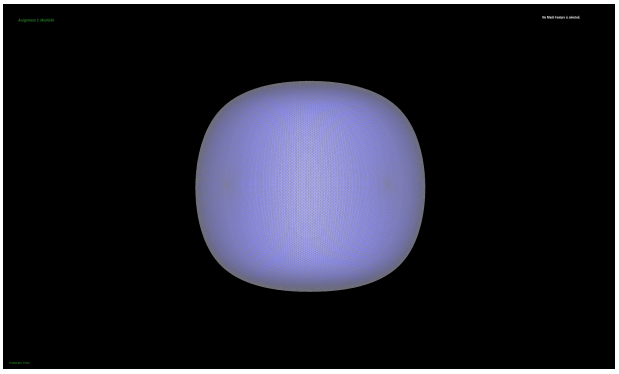
Symmetric Box, 3 subdivision



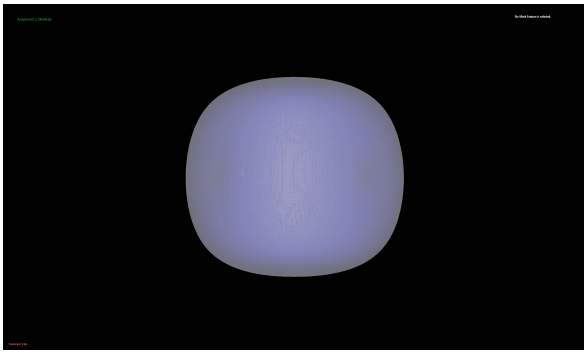
Symmetric Box, 4 subdivisions



Symmetric Box, 5 subdivision



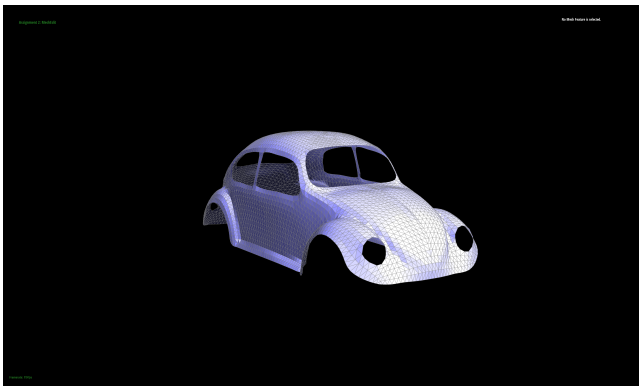
Symmetric Box, 6 subdivisions



Symmetric Box, 7 subdivision

Extra Credit

Since we implemented boundary splitting in Part 5, we figured that we might as well implement boundary splitting in Q6 as well. Very little had to be changed from our Q5 and 6 implementations on their own, except for a single calcuation. When computing the position of our new edges, we check to see if either halfedge is on a boundary. If so, instead of attempting to compute the interpolated point based on a weighted average of our two directions, we simply place the point halfway between the two already existing boundary verticies. In doing this, we avoid the strange waving effect that appears by default when subdividing boundary edges with the previous ratios. This can be seen working on the beetle mesh below:



Beetle, no subdivisions

Beetle, subdivided