CS 184: Computer Graphics and Imaging, Spring 2023

Project 2: MeshEdit

Brian Santoso, CS184 Team: #1-lana-del-rey-fan-1

Overview

Give a high-level overview of what you have implemented in this assignment. Think about what you have built as a whole. Share your thoughts on what interesting things you have learned from completing this assignment.

Webpage:

https://briansantoso.github.io/project-webpages-sp23-BrianSantoso/proj2/index.html

In this assignment, I implemented key features of a mesh editor. I implemented 2D and 3D implementations of de Casteljau's algorithm, an algorithm to evaluate points on Bezier curves by repeatedly taking the midpoints of control points until there is only 1 point left, which is equal to the point of the Bezier curve at the given parameters. This allows us to render surfaces on a 3D model. I also implemented area-weighted vertex normals which allow for smooth interpolation across polygon faces. I learned to traverse the halfedge data structure, a data structure that supports constant time mesh operations such as edge flipping, splitting, and collapsing. I used the halfedge datastructure operations to implement loop subdivision, an algorithm to essentially up-sample a mesh by subdividing its polygons into several sub-polygons.

Task 1

Briefly explain de Casteljau's algorithm and how you implemented it in order to evaluate Bezier curves.

de Castelijau's algorithm repeatedly lerps between points with parameter t, starting with the control points, until arriving at a single point which represents a point on the Bezier curve.

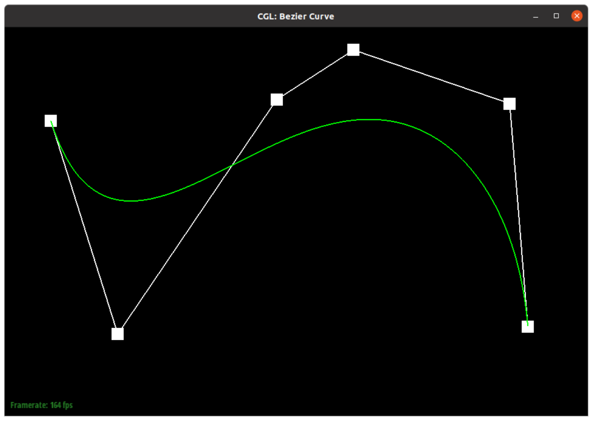
To implement de Castelijau's algorithm in order to evaluate Bezier curves, I implemented `evaluateStep` which executes a single step in de Castelijau's algorithm—Given n points and parameter t, I return a list of n-1 intermediate points.

I implement this by iterating through each points i, i+1 and linearly interpolating between them with parameter t to compute the new intermediate point corresponding to parameter t.

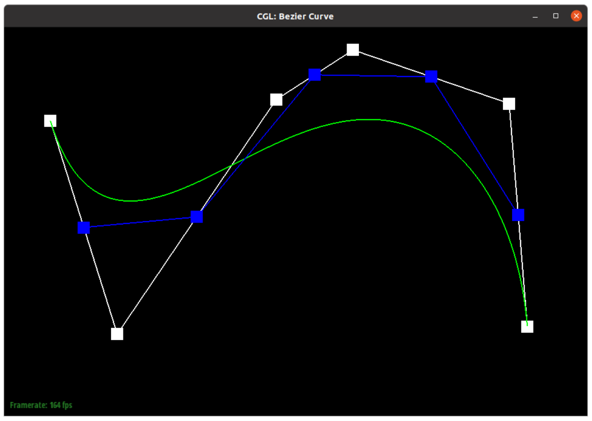
We can repeat this step until arriving at a single point, which represents the point on the Bezier curve corresponding to parameter t. Do this for all t in [0, 1] to trace the Bezier curve.

Take a look at the provided `.bzc` files and create your own Bezier curve with 6 control points of your choosing. Use this Bezier curve for your screenshots below.

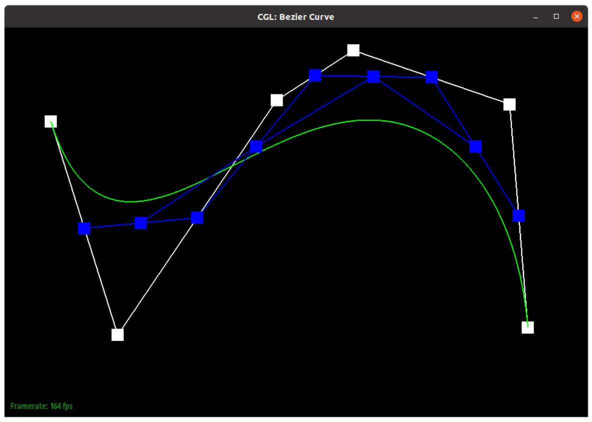
- **Show screenshots of each step / level of the evaluation from the original control points down to the final evaluated point.**



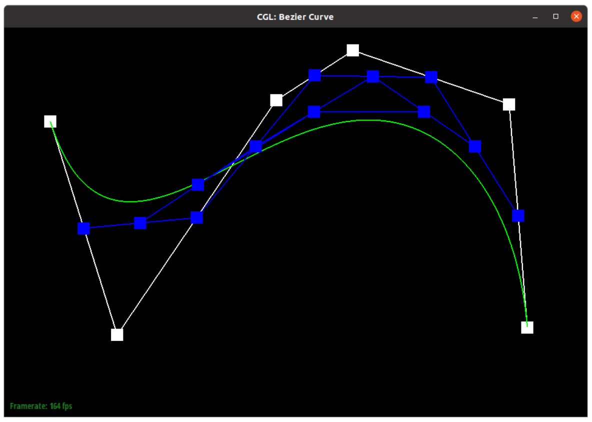
Step 1



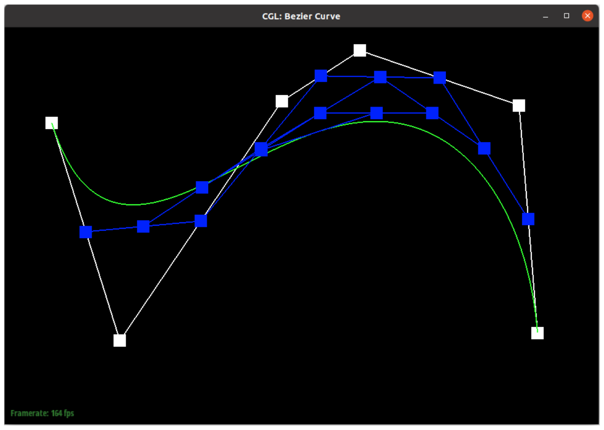
Step 2



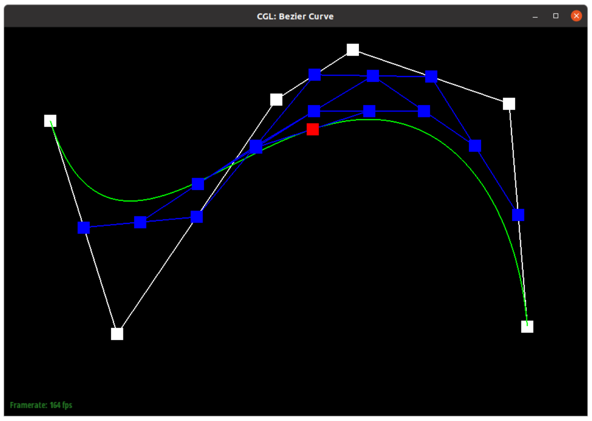
Step 3



Step 4

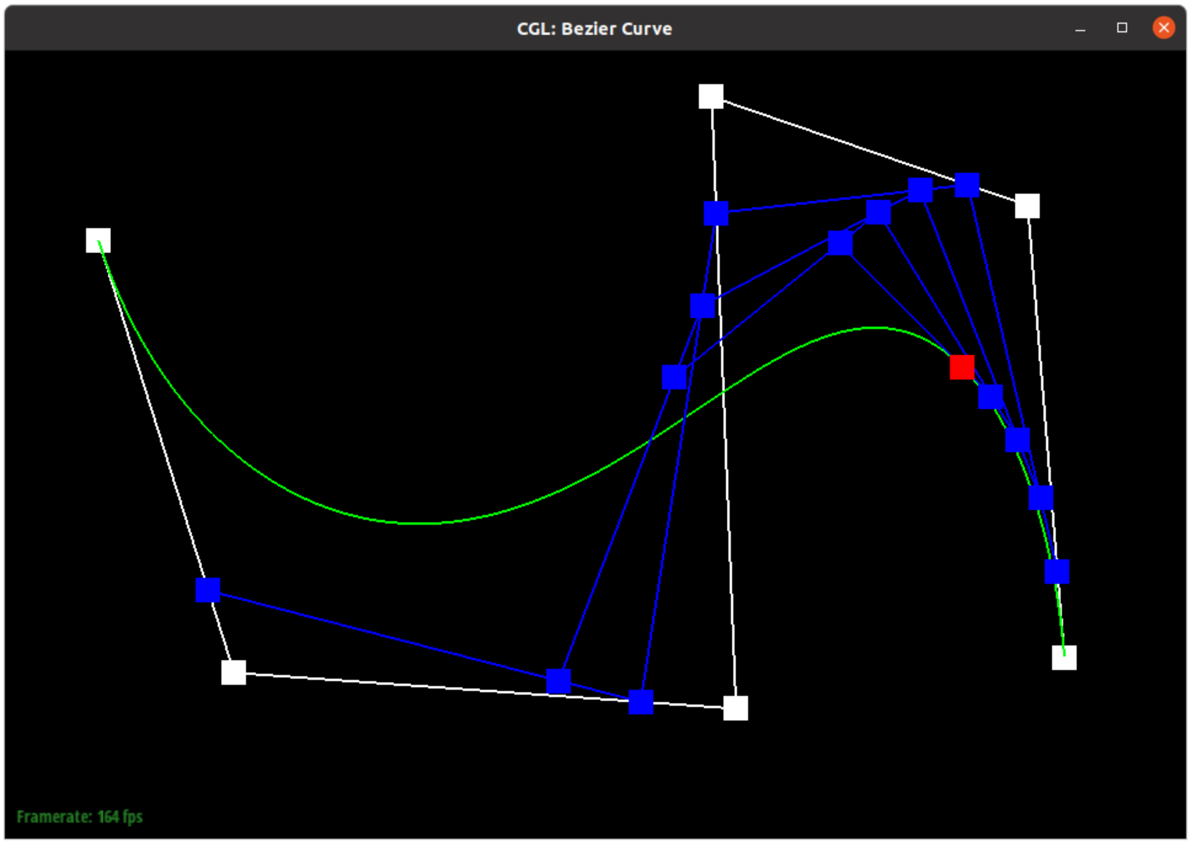


Step 5



Step 6 (Final surface point in red)

Show a screenshot of a slightly different Bezier curve by moving the original control points around and modifying the parameter *t* via mouse scrolling.

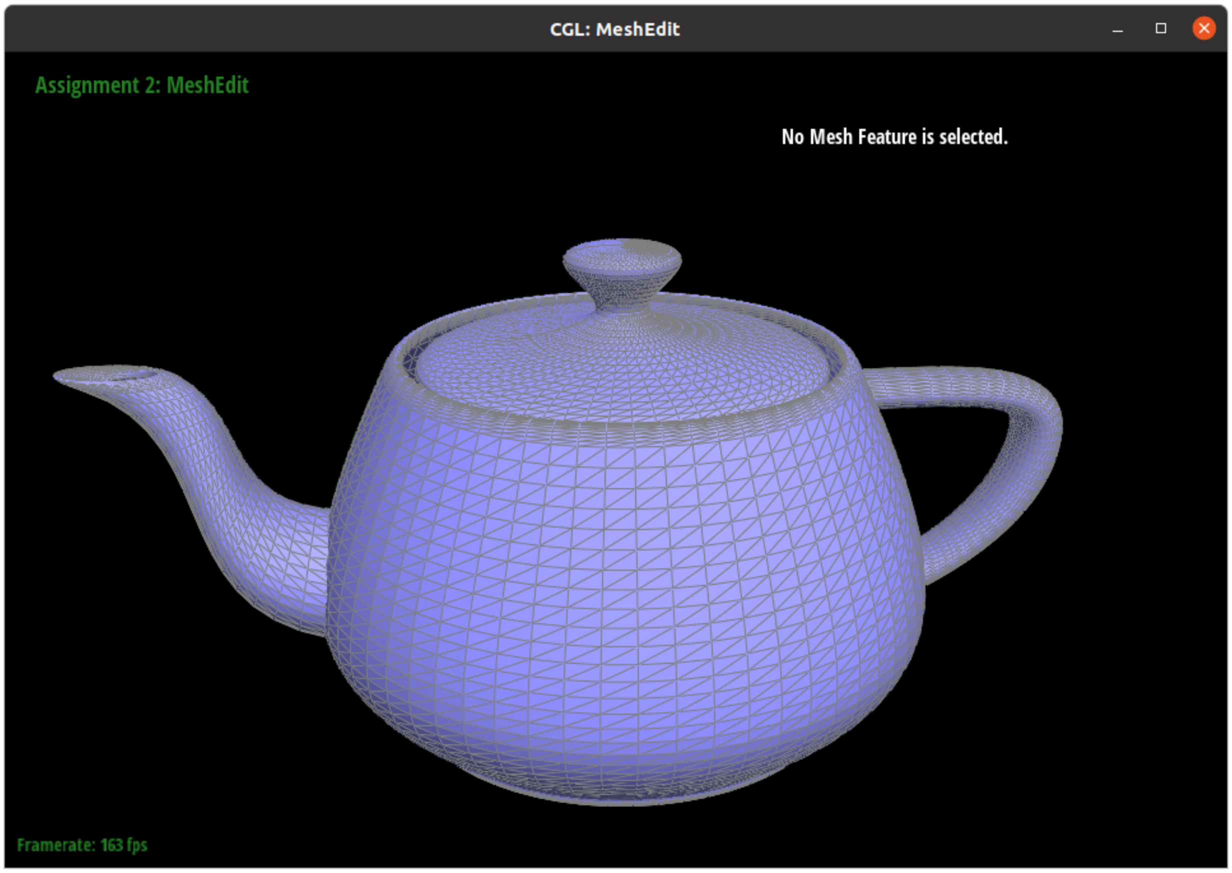


Task 2

Briefly explain how de Casteljau algorithm extends to Bezier surfaces and how you implemented it in order to evaluate Bezier surfaces.

The de Casteljau algorithm can extend to 3D Bezier surfaces by taking a grid of control points and parameters `u` and `v`. Along each row the algorithm evaluates the bezier curve point at parameter `u`, using the row of points as control points, just as in the 2D de Casteljau algorithm. For `r` rows, this leaves us with a column of `r` points, which we can use as control points for a “sweeping” bezier curve (whose position changes with parameter `u`) and we can again apply the 2D de Castelijau algorithm on these `r` points to evaluate the point at `v` which lies on the sweeping bezier curve. This final point is also the point on the bezier surface corresponding to parameters `(u, v)`.

Show a screenshot of `bez/teapot.bez` (not `.dae`) evaluated by your implementation.



Task 3

Briefly explain how you implemented the area-weighted vertex normals.

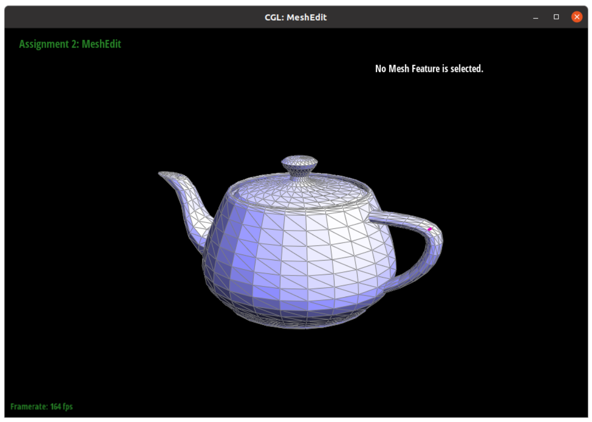
To implement the area-weighted vertex normals, I iterate through each of the faces incident to the vertex. However, I do not find the face object itself, just 2 of the edges of the face (represented as vectors) where one vector points into the vertex (we will call it `A`) and one out from the `vertex` (we will call it `B`),

To get `A` and `B` of each face, I start with the half edge exiting the vertex. To compute `A`, I subtract `h->twin()`'s vertex from the `vertex`. To compute `B`, I subtract the `vertex` from `h->twin()->next()->twin()`'s vertex.

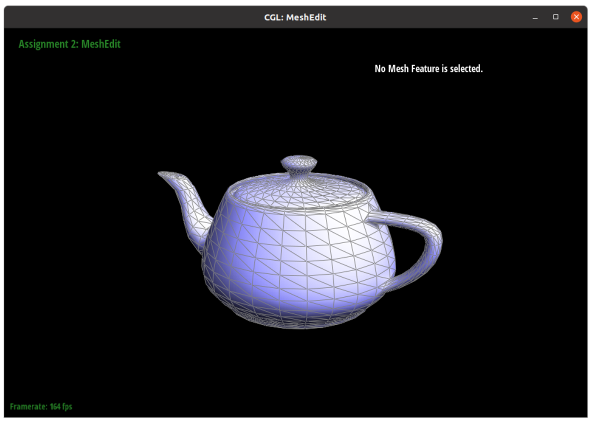
To compute the normal for the face, I first find compute the area of the triangle by taking half the magnitude of `A x B`. Then I normalize `A x B` and scale it by the area of the triangle.

I continue iterating through each face (repeating the above process starting with `h_twin->next()`) and computing their normals. At the end I sum up all the area-weighted face normals and normalize the sum to get the area-weighted vertex normal.

Show screenshots of `dae/teapot.dae` (not `.bez`) comparing teapot shading with and without vertex normals.



Without Vertex Normals



With (Area-weighted) Vertex Normals

Task 4

Briefly explain how you implemented the edge flip operation and describe any interesting implementation / debugging tricks you have used.

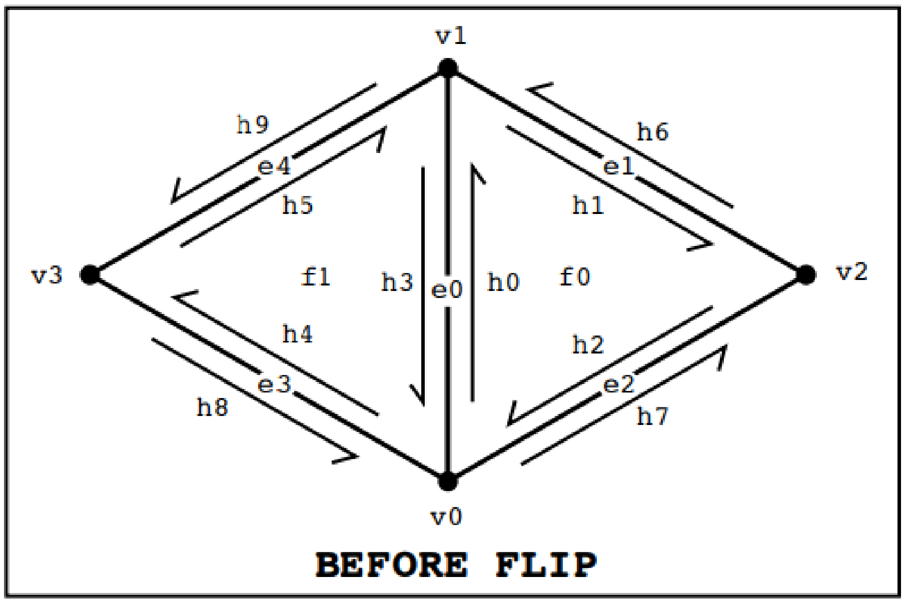
To implement the edge flip operation, I first collected references to every half edge, vertex, edge, and face incident to the 2 triangles containing the given edge.

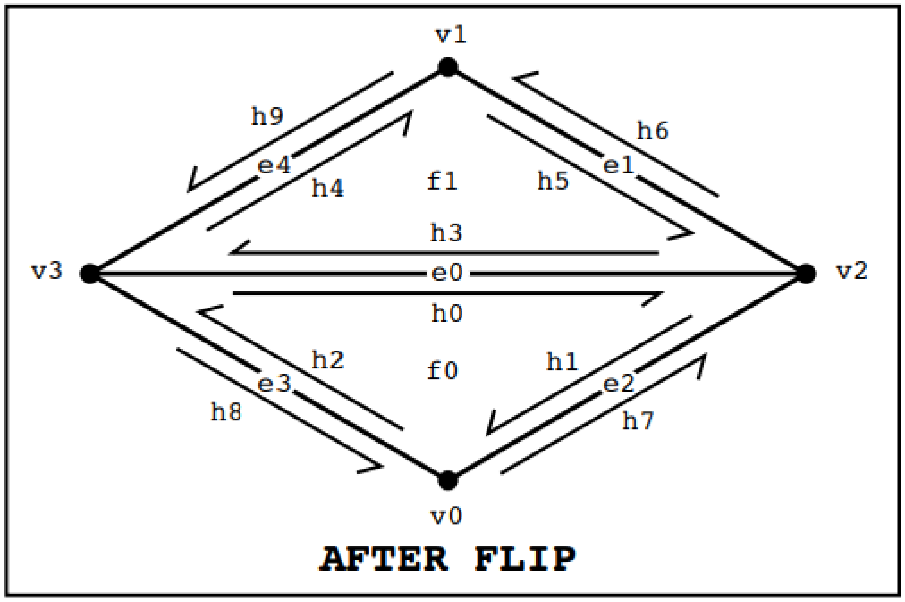
Since we do not want to flip any boundary edges, I then check if either of the two faces lie on a boundary, and if so, I return the given edge without any modification immediately.

Finally, I reassign the vertex, edge, and face halfedge pointers for all vertices, edges, and faces, along with all the pointers in each halfedge, even if they have not been modified. The pointers are reassigned corresponding the a halfedge flip operation, and this is easily done since I have already collected a list of all elements to reassign pointers to.

No elements are created or deleted in my implementation of edge flip. Only pointers are reassigned.

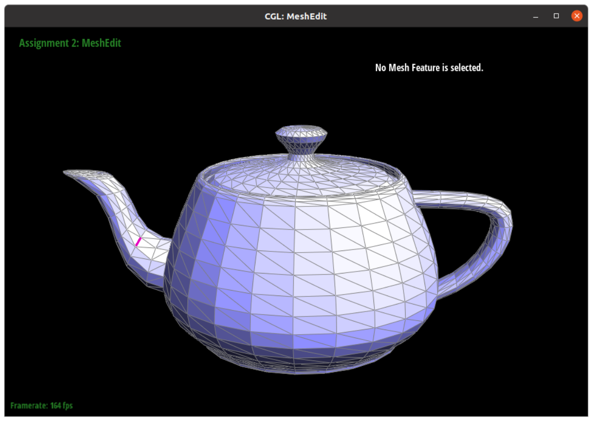
The following labeled diagram from (http://15462.courses.cs.cmu.edu/fall2015content/misc/HalfedgeEdgeOpImplementationGuide.pdf) was helpful for implementing my half edge flip operation and figuring out what element each pointer needs to be reassigned to.





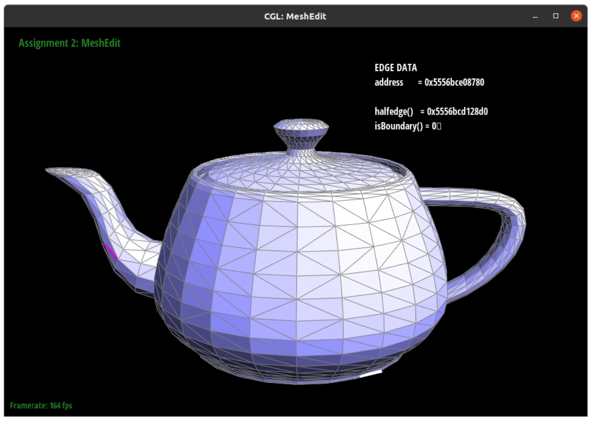
Labeled Diagram Showing Halfedge, Vertex, Edges, and Face elements Before and After an Edge Flip Operation

Show screenshots of the teapot before and after some edge flips.



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

Before Edge Flips



```
EDGE DATA
address      = 0x5556bce08780

halfedge()   = 0x5556bcd128d0
isBoundary() = 0
```

After Edge Flips

Write about your eventful debugging journey, if you have experienced one.

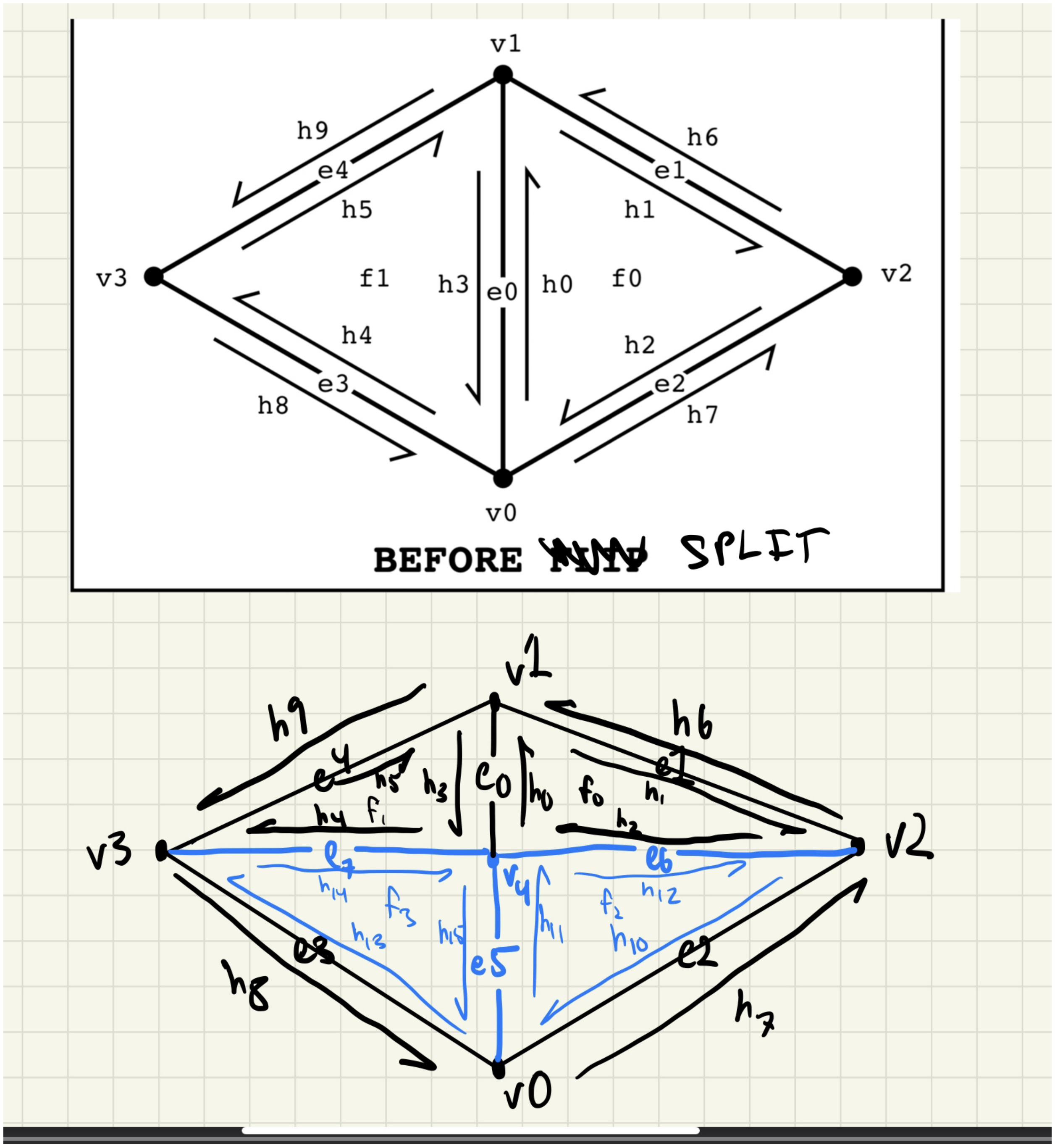
I did not experience any bugs with this task, thanks to good, cautious, and thorough coding practices while implementing it.

Task 5

Briefly explain how you implemented the edge split operation and describe any interesting implementation / debugging tricks you have used.

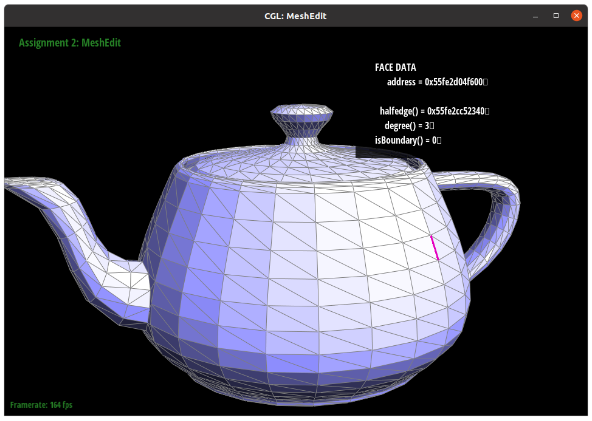
To implement the edge split operation, I first collected references to every half edge, vertex, edge, and face incident to the 2 triangles containing the given edge. I did not implement boundary edge splits so I simply return immediately if either neighboring face of the edge is on a boundary loop. Then I create 5 new halfedges, the new vertex, 3 new edges (not 4 because I utilize the current given edge), and 2 new faces. I reassign the vertex, edge, and face halfedge pointers for all of the existing and new vertices, edges, and faces, along with all the pointers in each halfedge, even if they have not been modified. Finally, I set the position of the newly created vertex to the midpoint of the 2 vertices incident to the given edge.

The pointers are reassigned according to the following diagram:



Labeled Diagram Showing Halfedge, Vertex, Edge, Face Elements Before and After an Edge Split Operation

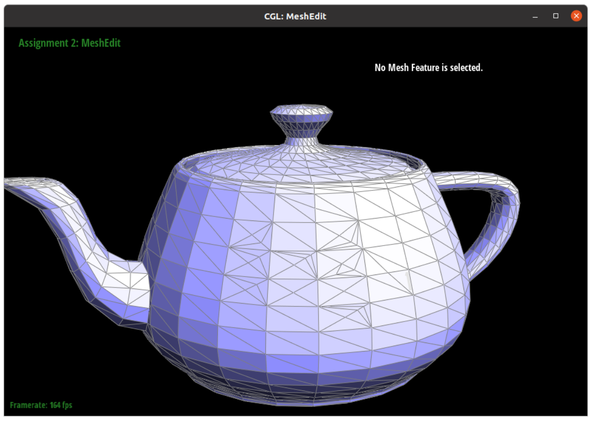
Show screenshots of a mesh before and after some edge splits.



```
FACE DATA
    address = 0x55fe2d04f600

  halfedge() = 0x55fe2cc52340
   degree() = 3
isBoundary() = 0
```

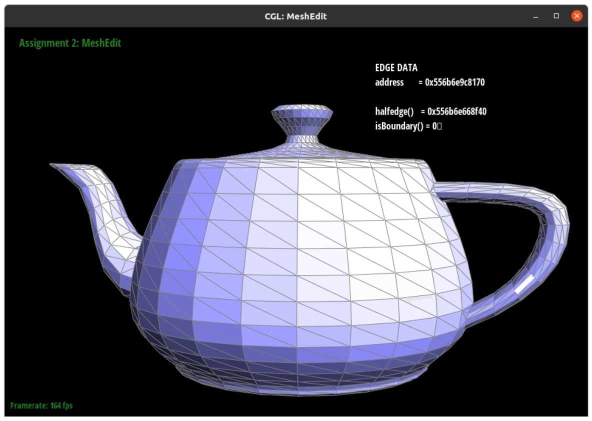
Before Edge Splits



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

After Edge Splits

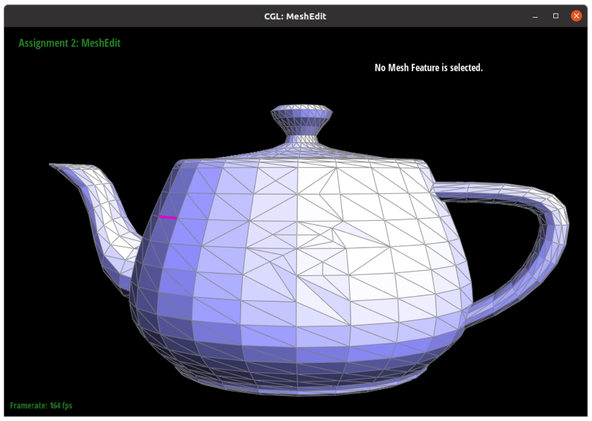
Show screenshots of a mesh before and after a combination of both edge splits and edge flips.



```
EDGE DATA
address      = 0x556b6e9c8170

halfedge()   = 0x556b6e668f40
isBoundary() = 0
```

Before Combination of Edge Splits and Flips



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

After Combination of Edge Splits and Flips

Write about your eventful debugging journey, if you have experienced one.

I did not experience any bugs with this task, thanks to good, cautious, and thorough coding practices while implementing it.

Task 6

Briefly explain how you implemented the loop subdivision and describe any interesting implementation / debugging tricks you have used.

To implement loop subdivision, I first iterate through all the vertices in the original input mesh, and calculate their updated position (without setting it). The new position of a vertex in the original mesh is equal to `newPosition = (1 - n * u) * original_position + u * original_neighbor_position_sum`, where `n` is the edge degree of the vertex and `u` is equal to 3/16 if the vertex has an edge degree of 3 and `3/(8n)` otherwise. I also flag all the vertices in the original mesh as an old vertex.

Then, since every edge will be split and a new vertex will be introduced near the midpoint of each edge, I compute the positions of the new vertices corresponding to each edge. I do this by iterating over all edges in the the original mesh, and compute the position of the new vertex corresponding to the edge. The position of the new vertex is equal to `3.0/8 * (A + B) + 1.0/8 * (C + D)` where `A`, `B` are the vertices incident to the edge, and `C`, `D` are the last 2 vertices of the 2 triangles incident to the edge split. I also flag all the edges in the original mesh as an old edge.

Note that I perform these calculations before executing any edge splits to keep the implementation simple and not have to worry about traversing a changing mesh.

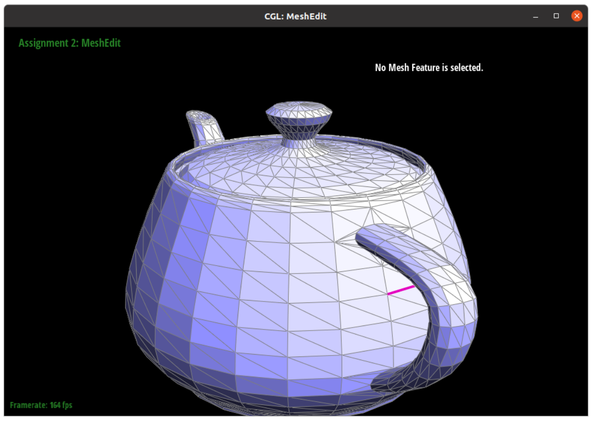
I then split every edge in the mesh by iterating through all the edges in the original mesh.

I then flip any new edge that connects and old a new vertex. To check if a new edge connects an old and new vertex, I take the XOR of the `isNew` flag of the incident vertices.

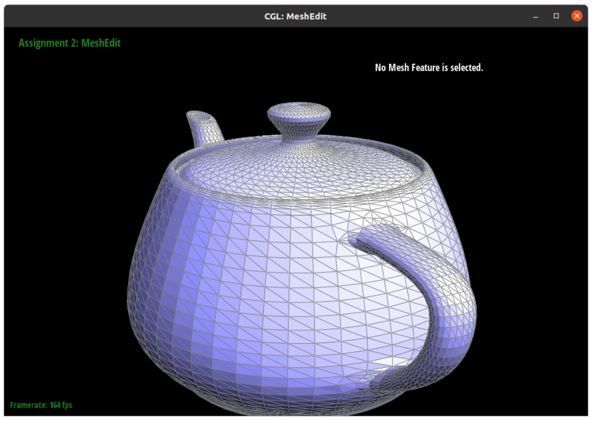
Finally, I update all vertices to their the new vertex positions by iterating through all the vertices in the new mesh.

Take some notes, as well as some screenshots, of your observations on how meshes behave after loop subdivision. What happens to sharp corners and edges? Can you reduce this effect by pre-splitting some edges?

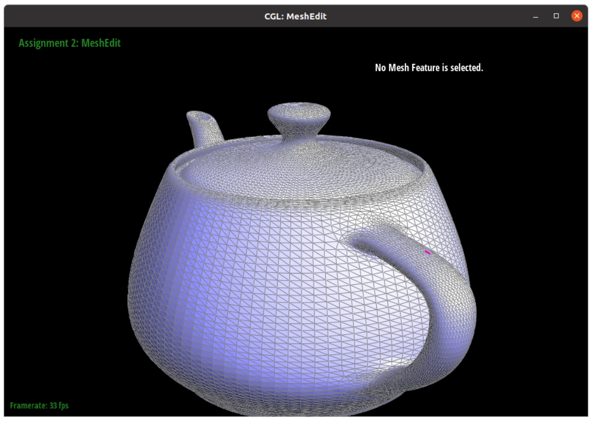
At sharp corners and edges, such as the approximately 90 degree concave angle at the meeting of the teapot and it's handle, the mesh extrudes into the space outside the bounds of the teapot. This is because loop subdivision attempts to interpolate between vertices, and interpolating at a concave 90 degree angle without a sufficient number of polygons at the edge produces a vertex that is in the space out of the boundary of the original teapot. Alternatively, a sharp convex angle would collapse the mesh too far within its boundary without a sufficient number of polygons at the edge.



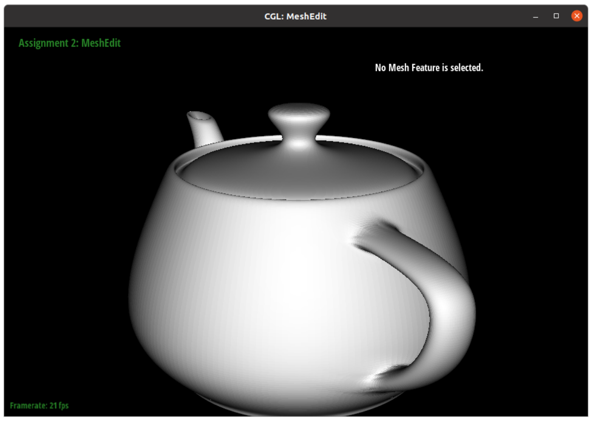
Teapot before loop subdivisions



Teapot after 1 loop subdivision



Teapot after 2 loop subdivisions



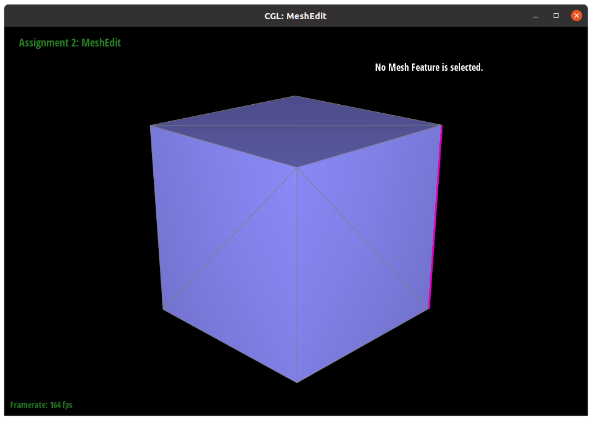
Teapot after 3 loop subdivisions (wireframe didsabled). Observe that at the sharp interaction between the teapot and it’s handle, the mesh becomes extruded into space.

Load `dae/cube.dae`. Perform several iterations of loop subdivision on the cube. Notice that the cube becomes slightly asymmetric after repeated subdivisions. Can you pre-process the cube with edge flips and splits so that the cube subdivides symmetrically? Document these effects and explain why they occur. Also explain how your pre-processing helps alleviate the effects.

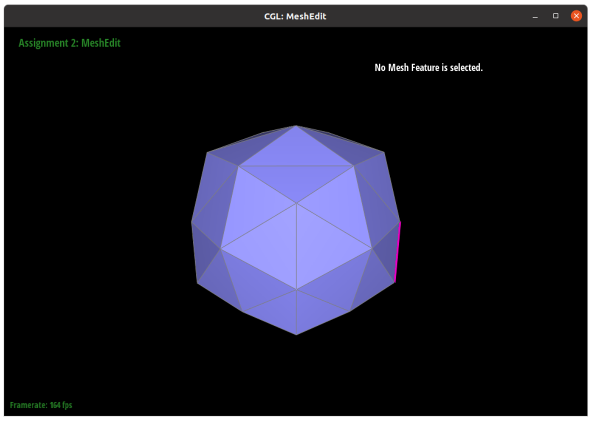
Loop subdivision up-samples a mesh, making it smoother in the process. However, at sharp edges without a sufficient number of polygons, loop subdivision will prominently collapse/extrude a mesh inside/outside the original bounds.

For example, after repeated subdivisions, the cube becomes asymmetric and approaches a round object with repeated subdivisions. This occurs because the original cube mesh itself is asymmetric

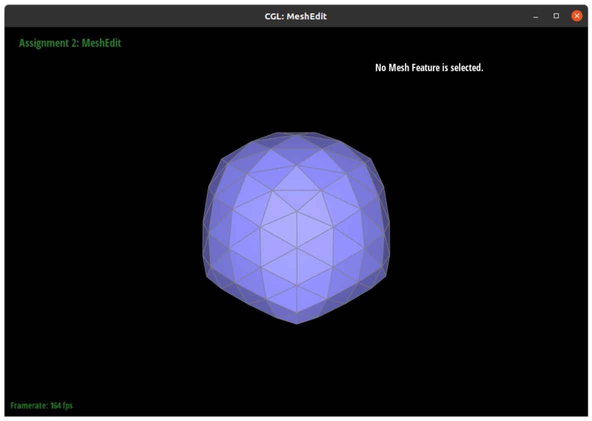
- note that the faces of a the cube are formed by 2 triangles which are not radially symmetric! A better mesh to use would be to split those edges to create 4 radially symmetric triangles.



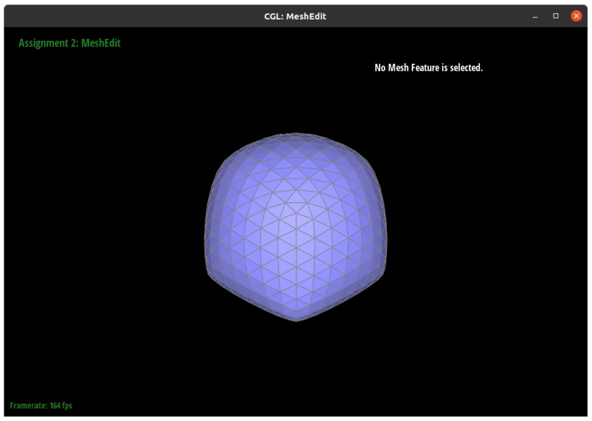
Cube before loop subdivision



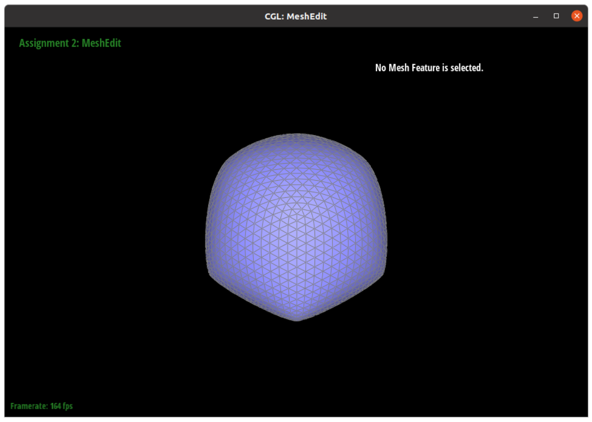
Cube after 1 loop subdivision



Cube after 2 loop subdivisions

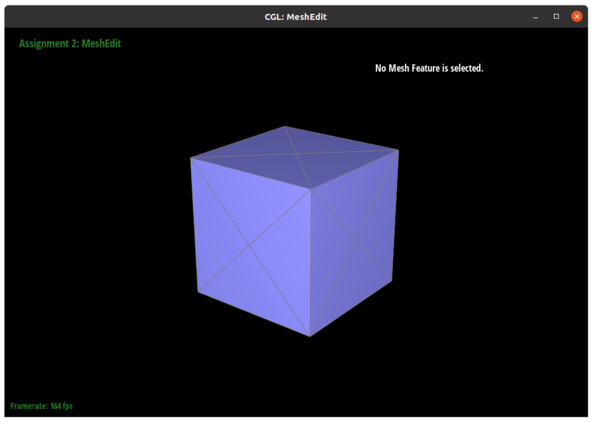


Cube after 3 loop subdivisions

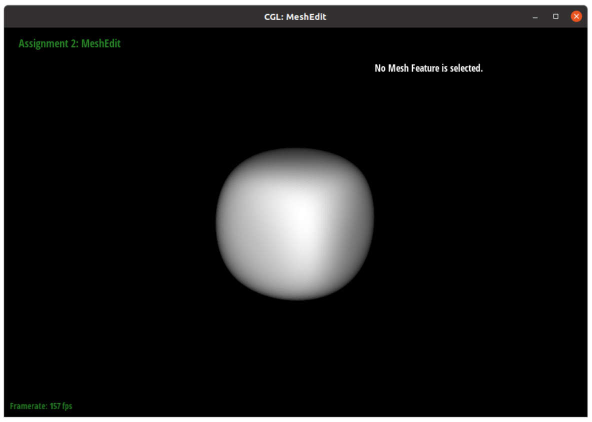


Cube after 4 loop subdivisions

To fix the asymmetric subdivision of the cube, I preprocess the cube by splitting the diagonal edges on the inside of the faces of the cube. Now the faces of the cube are radially symmetric.



Preprocessed cube



Cube after 5 loop subdivisions, it looks much more like a cube and is symmetric!