CSC4140 Assignment 5

Computer Graphics

April 4, 2022

Geometry

This assignment is 9% of the total mark.

Strict Due Date: 11:59PM, April 05^{th}, 2022

Student ID: 119010265

Student Name: Shi Wenlan

This assignment represents my own work in accordance with University regulations.

Signature: Shi Wenlan

1 Overview

In this assignment, I learned to use de Casteljau algorithm to interpolate and implement Bézier Curves, and further implement Bézier surfaces. I also learned half-edge data strucutre in depth, and implement the calculation of area-weighted vertex normals, edge flip, edge split (including boundary cases), and mesh upsampling using the Loop subdivision algorithm based on this data structure.

1.1 Task 1 (10 points)

De Casteljau's algorithm: In simple terms, this algorithm is looking for a point on a Bézier curve. Represent a Bézier curve as $\mathbf{b}^n(t)$ where t is in the range of [0, 1] and n is the number of control points. Then we can compute the coordinate of $\mathbf{b}^n(t)$ for specific t using following formula:

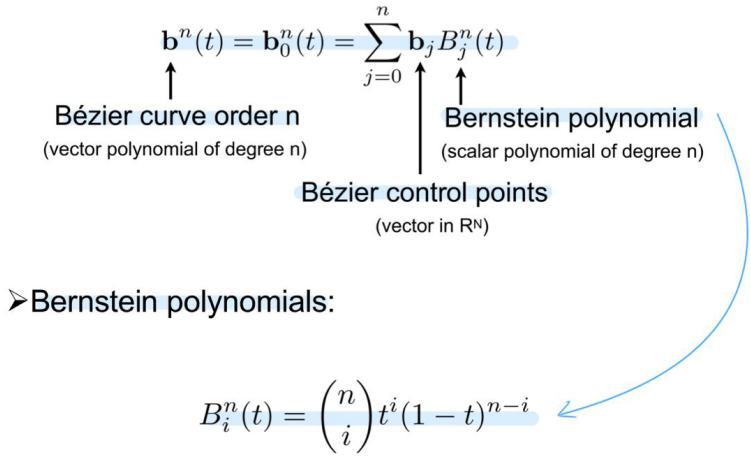


Figure 1: Bernstein form of a Bézier curve of order n

And the form that I'm going to use in this assignment is its recursive form:

$$\mathbf{b}_j^i(t) = (1-t)\mathbf{b}_j^{i-1} + t\mathbf{b}_{j+1}^{i-1}$$

Implementation: Read the intermediate point of the previous layer under a specific t, and calculate the point of the next layer using the recursive form formula. The points of different layers can be understood as shown in the following figure:

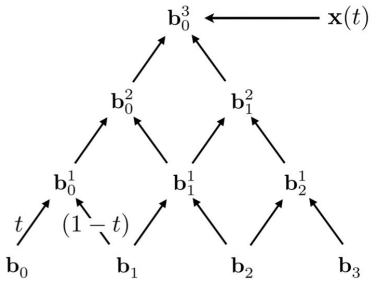


Figure 2: The points of different layers

Where every rightward arrow is multiplication by t and every leftward arrow by (1-t).

Screenshots: The screenshot of each step / level of the evaluation from the original control points down to the final evaluated point of my own Bézier curve is as following:

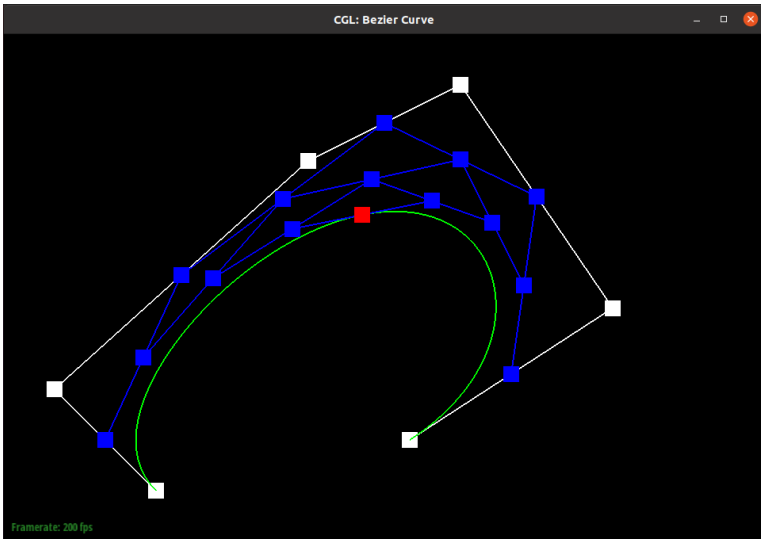


Figure 3: Original version

The screenshot of a slightly different Bézier curve by moving the original control points around and modifying the parameter t is as following:

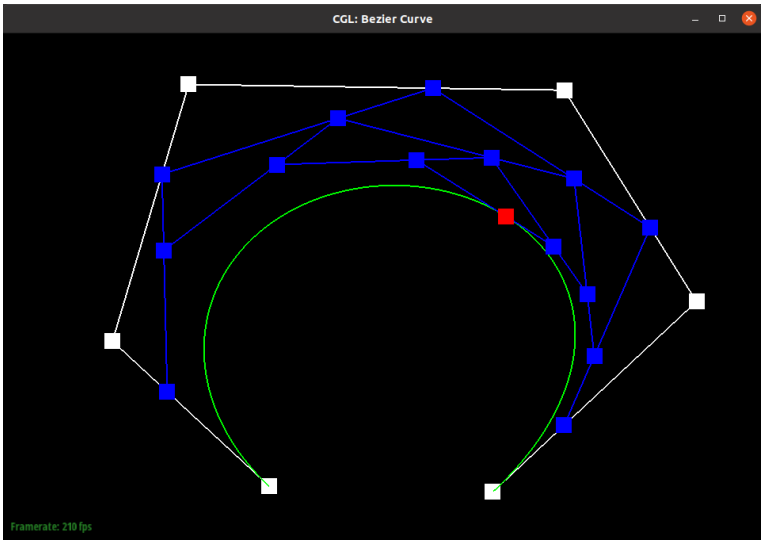


Figure 4: Another version

1.2 Task 2 (15 points)

Extend to Bézier surfaces: Take 4*4 control points as an example. Firstly, each 4*1 control points define a Bézier curve $\mathbf{b}^n(u)$, then we have 4 Bézier curves. Secondly, for specific u, we have 4 control points on these 4 Bézier curves (i.e. 1 control point per Bézier curve) which define a Bézier curves $\mathbf{b}^n(v)$. Then with different set of (u, v), we can sweep out a 2D surface.

Implementation: Firstly, $BezierPatch :: evaluateStep()$ is a Vector3D version of $BezierCurve :: evaluateStep()$, which can be used to compute Bézier curve in 3 steps in $BezierCurve :: evaluate1D()$. Secondly, $evaluate()$ use 4 final interpolated vectors $\mathbf{b}^n(u)$ for given u returned by 4 $evaluate1D()$ to compute the final interpolated vector $\mathbf{b}^n(v)$ for given v using $evaluate1D()$.

Screenshots: The screenshot of bez/teapot.bez evaluated by your implementation is as following:

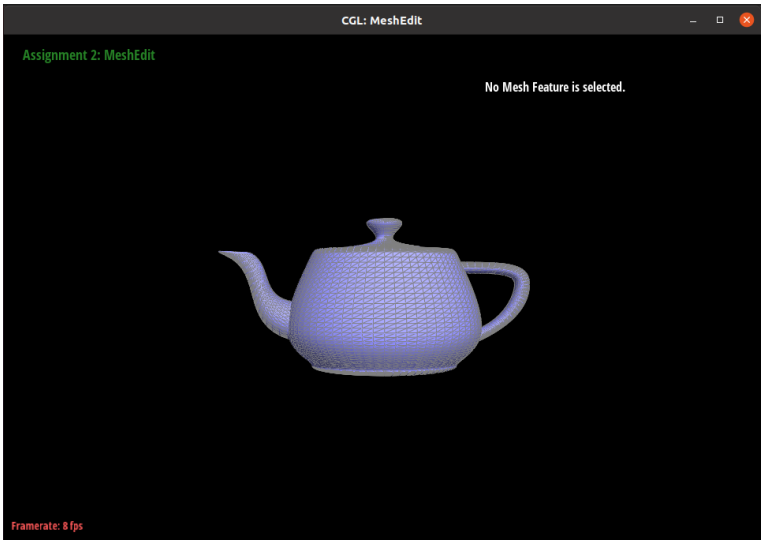


Figure 5: bez/teapot.bez

1.3 Task 3 (10 points)

Implementation: Given the coordinates **a**, **b**, **c** of 3 vertices of a triangle, we can compute the the normal of the triangle by normalizing $a \times b + b \times c + c \times a$, and compute the double of the area of the triangle by take the norm of $a \times b + b \times c + c \times a$. So, I sum up all $a \times b + b \times c + c \times a$ for each triangle around given vertex and normalize it.

Screenshots: The screenshots of dae/teapot.dae comparing teapot shading with and without vertex normals are as following:

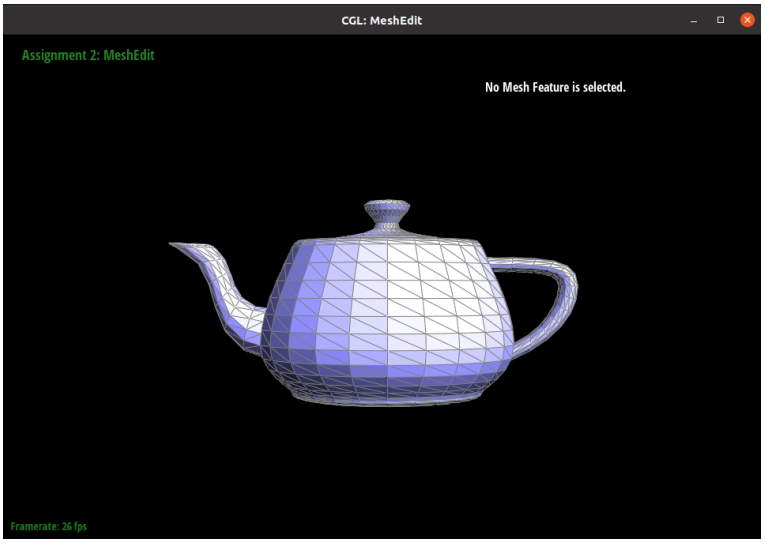


Figure 6: dae/teapot.dae shading without vertex normals

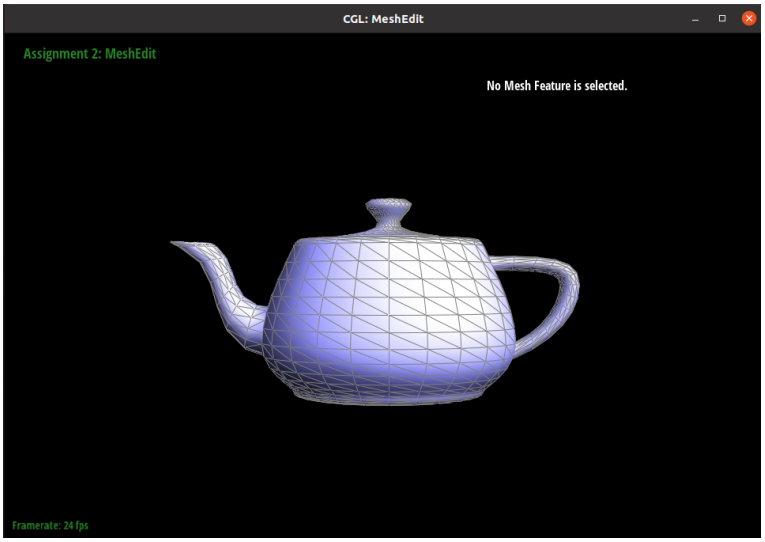


Figure 7: dae/teapot.dae shading with vertex normals

1.4 Task 4 (15 points)

Implementation: If the given edge is a boundary edge, simply return immediately. Otherwise, do the following steps. Firstly, collect elements including half-edges, edges, vertices and faces mentioned in the following figure:

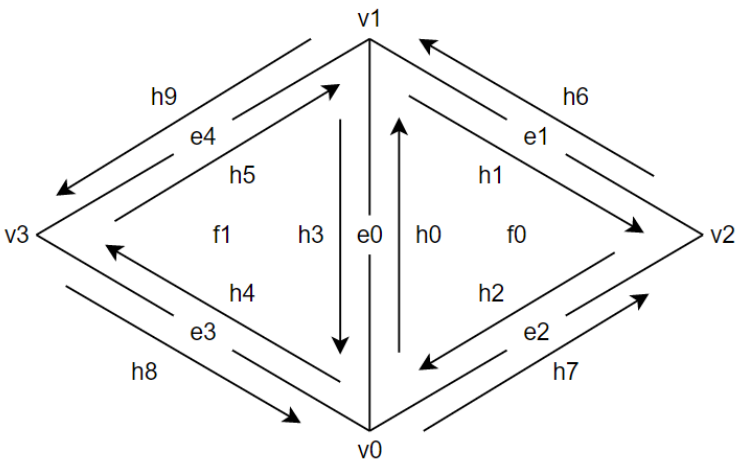


Figure 8: Before edge flip

Secondly, reassign elements including next(), twin(), vertex(), edge() and face() for half-edges, halfedge() for vertices, edges and faces as following figure:

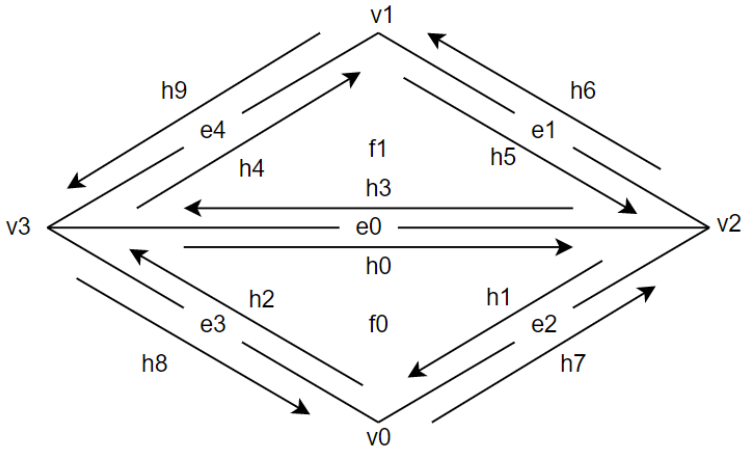


Figure 9: After edge flip

Even unchanged properties should be reset to avoid unexpected errors.

Screenshots: The screenshots of the teapot before and after some edge flips are as following:

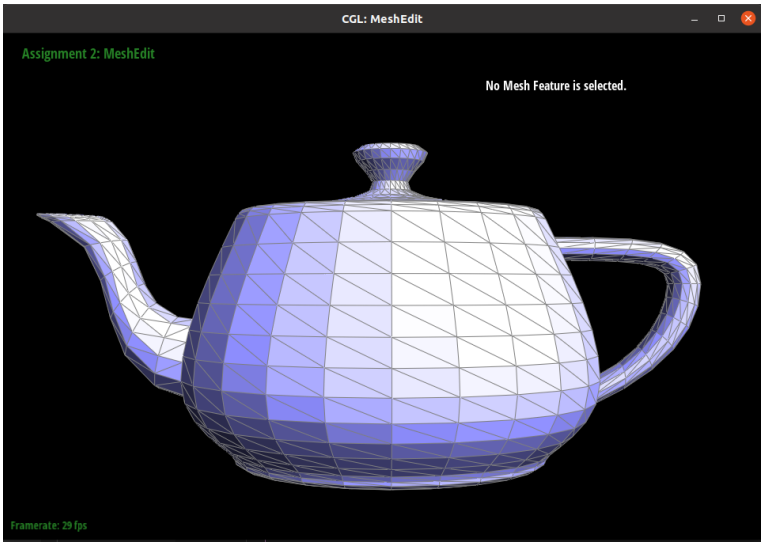


Figure 10: Before some edge flip

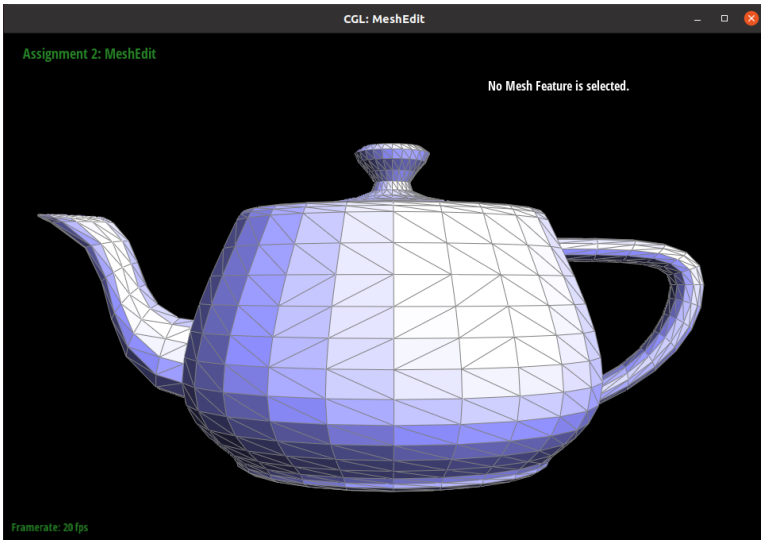


Figure 11: After some edge flip

1.5 Task 5 (15 points + 6 extra)

Implementation: Let me explain non-boundary edges first. Firstly, collect elements as task 4:

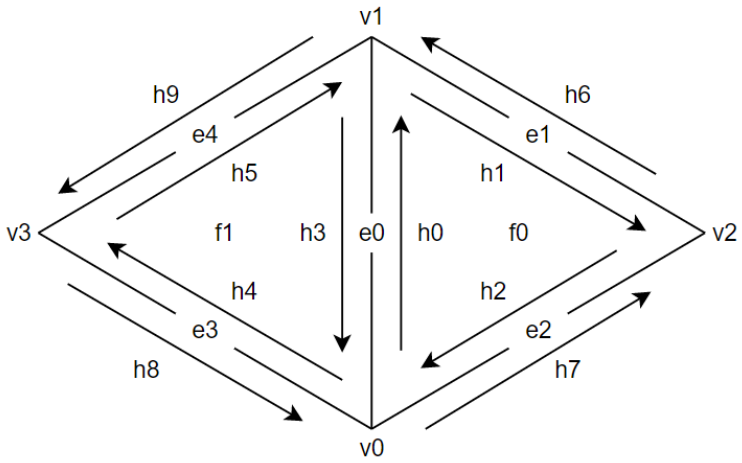


Figure 12: Before edge split (non-boundary)

Secondly, allocate new elements, such as v4, h10, h11 and so on. Thirdly, reassign all elements as following figure:

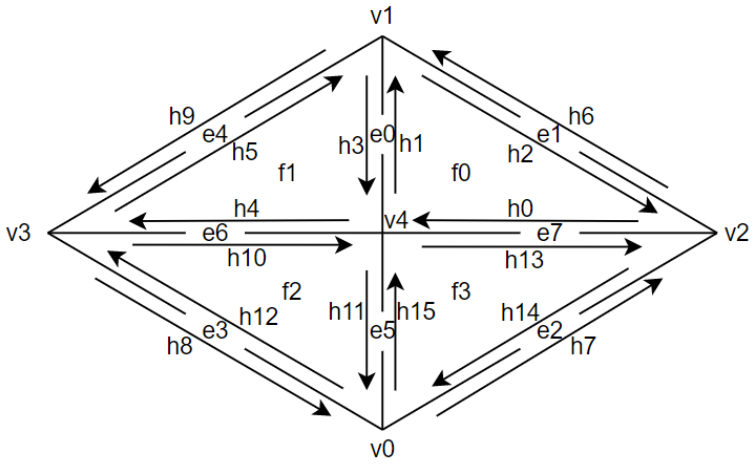


Figure 13: After edge split (non-boundary)

Remember to assign the *position* attribute of the new vertex. Still, even unchanged properties should be reset to avoid unexpected errors.

The the steps to deal with boundary edge cases are similar, just follow the figure to collect elements, allocate new elements, and reassign all elements. And two boundary edge cases are symmetric so that they can share a set of code by naming trick when collecting elements. The figure before and after edge split for boundary edge cases are as following:

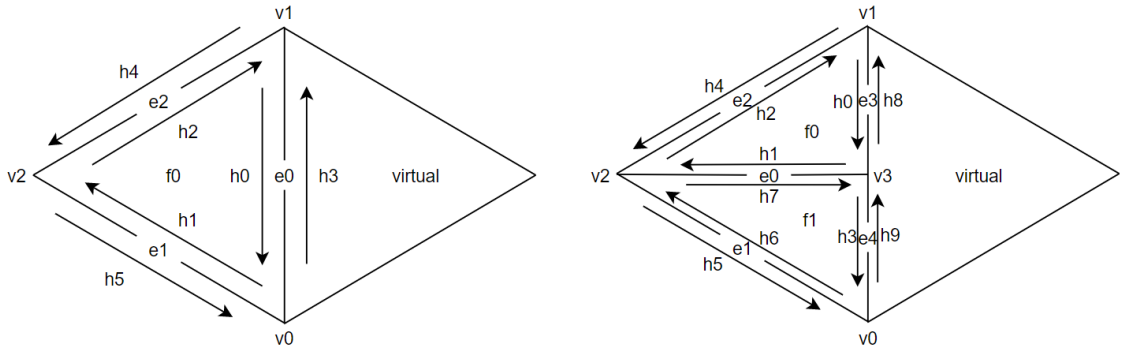


Figure 14: Before and after edge split (boundary case 1)

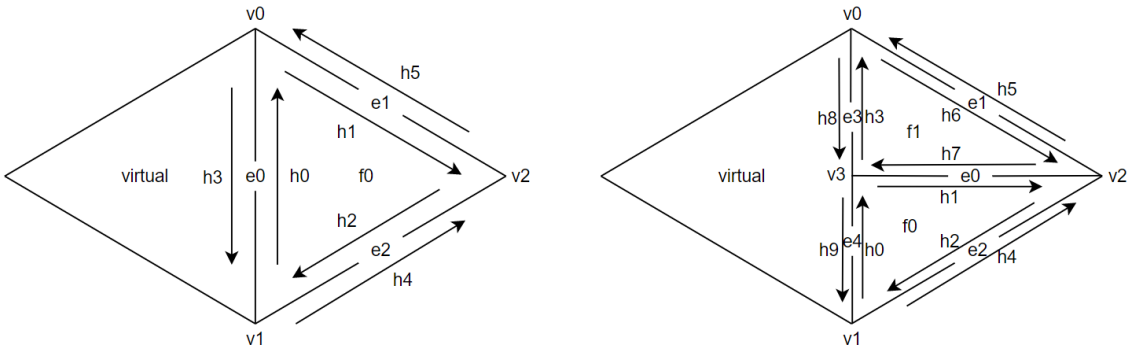


Figure 15: Before and after edge split (boundary case 2)

It should be noted that the original $halfedge()$ of the virtual face may be h3, so it also needs to be reassigned, and the $next()$ of boundary halfedge whose next() pointing to h3 also needs to be reassigned.

Screenshots: The screenshots of a mesh before and after some edge splits / a combination of both edge splits and edge flips of non-boundary edge case are as following:

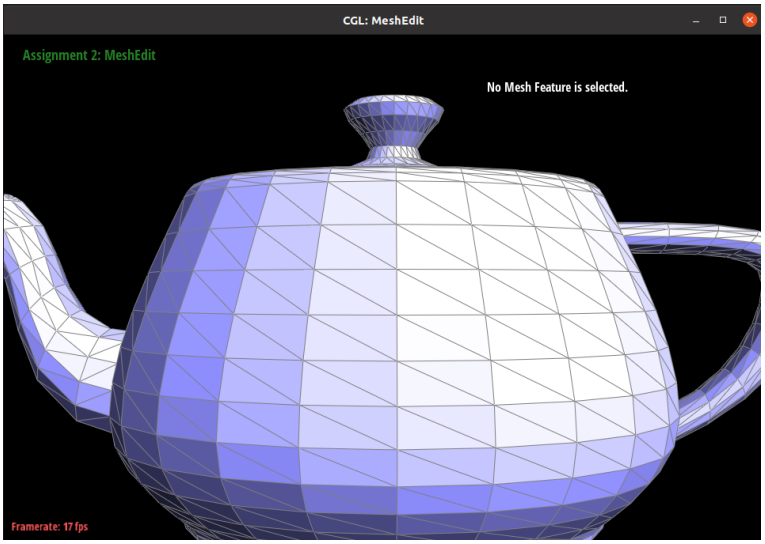


Figure 16: Original

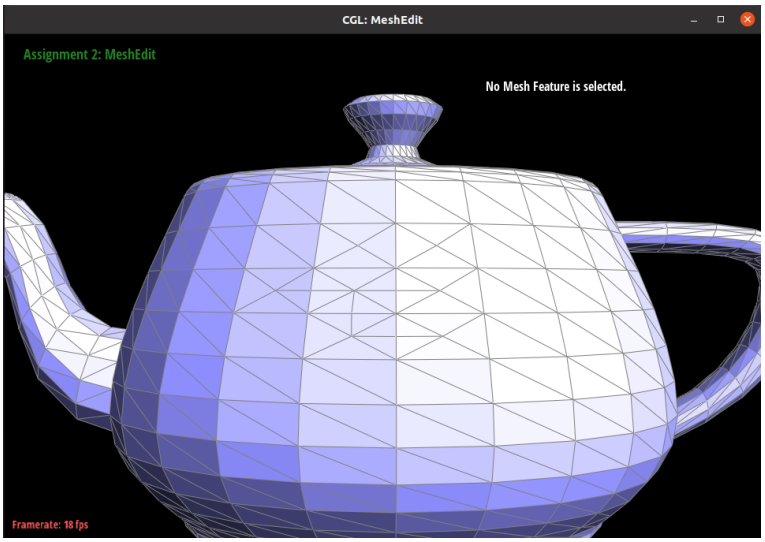


Figure 17: After some edge splits

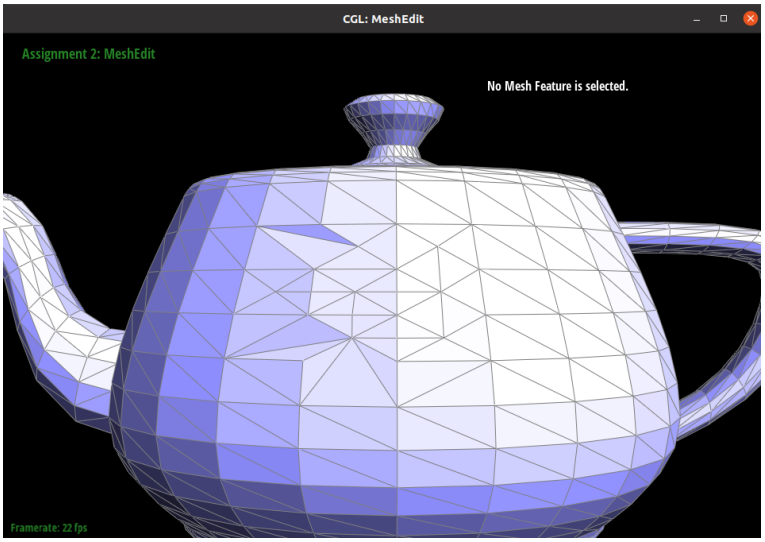


Figure 18: After a combination of both edge splits and edge flips

The screenshots of a mesh before and after some edge splits of boundary edge case are as following:

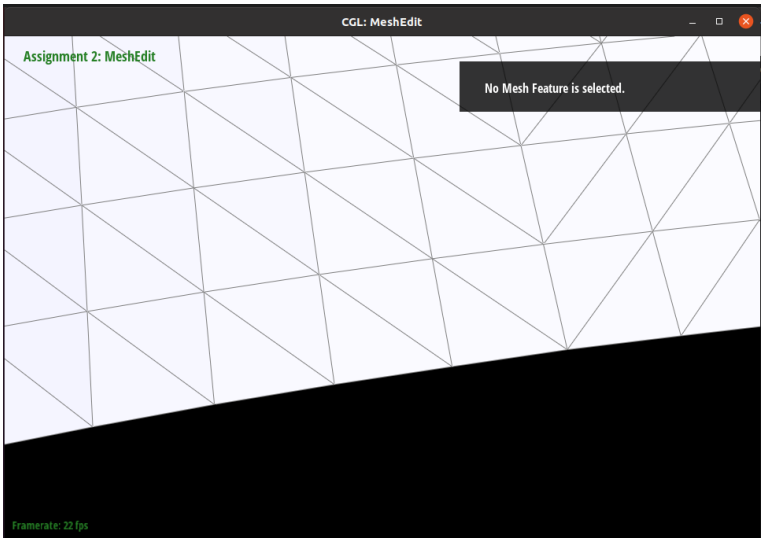


Figure 19: Before some edge splits (boundary case)

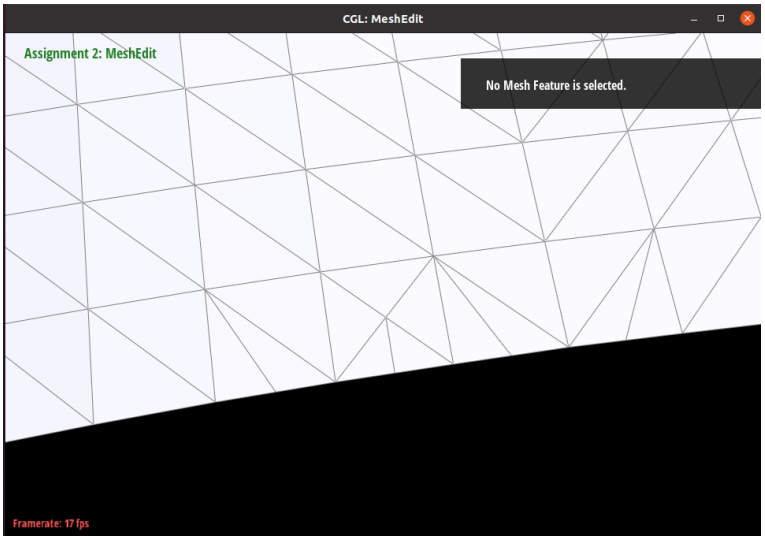
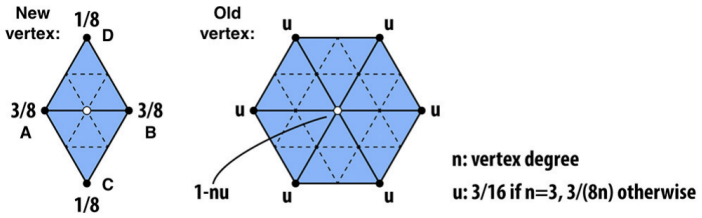


Figure 20: After some edge splits (boundary case)

1.6 Task 6 (25 points + 7 extra)

Loop subdivision: Loop subdivision is mainly divided into 2 steps: split each edge of the original mesh in any order, and then flip the newly generated edge whose one end is the new vertex and the other is the old vertex. Also, update the position of each vertex. The update rule for the position of vertices inside the mesh is shown in the following figure:



(1) The position of a new vertex splitting the shared edge (A, B) between a pair of triangle (A, C, B) and (A, B, D) is

```
3/8 * (A + B) + 1/8 * (C + D)
```

(2) The updated position of an old vertex is

```
(1 - n * u) * original_position
 + u * original_neighbor_position_sum
```

Figure 21: Update rule for non-boundary vertices

The update rule for the position of vertices on the boundary of the mesh that I found on the Internet is shown in the following figure:

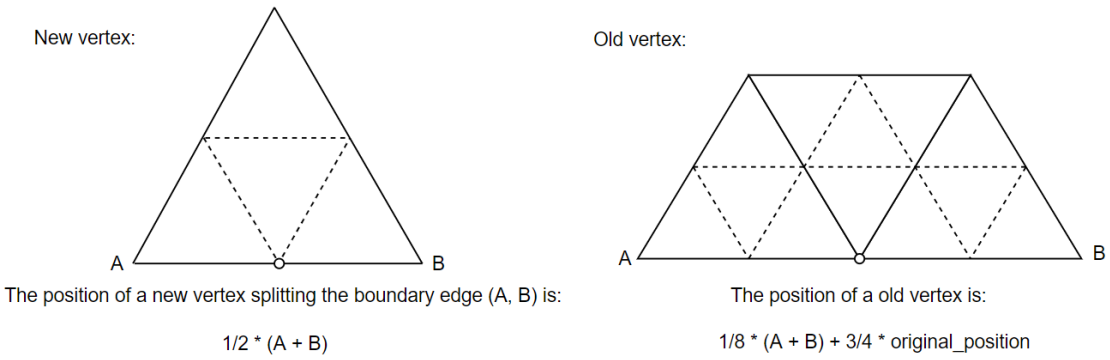


Figure 22: Update rule for boundary vertices

Implementation: The implementation mainly divided into 5 steps:

Firstly, compute new positions for all the vertices in the input mesh and store them in $Vertex$ :: $newPosition$. Take care to determine if the vertex is on the boundary, and use a different formula to calculate the new position if needed.

Secondly, compute the updated vertex positions associated with edges, and store it in $Edge$ :: $newPosition$.

Thirdly, split every edge in the mesh, in any order. Because after the split the edge that stores $newPosition$ can hardly be found, and after all splits are completed, the mapping between the new vertex and its $newPosition$ will also be lost, so, in this step, I sets a temporary variable $newPos$ to store the value of $newPosition$ before each single edge split, and assign the value of $newPos$ to the returned new vertex immediately after the single edge split. And at the same time, set the $isNew$ flag of new vertex to true to label them. Besides, in order to only iterate over edges of the original mesh, I create *splitlist* to copy the EdgeIter of the elements in *edges* of the given mesh. So that I can traverse only on *splitlist*. It should be noted that during the split process, do not move the relevant edges that have not been split. As shown in Figure 12, e1, e2, e3, and e4 are all left in place, and are not moved to fill in the blanks at e0, e5, e6, and e7. Only the split edge e0 is moved to fill in the blanks. Otherwise, even if we traverse on the copy of the edges of the original mesh, the same edge will still be split repeatedly.

Fourthly, flip any new edge that connects an old and new vertex. Filter edges that satisfy the conditions and store their $EdgeIter$ in $fliplist$, then traverse $fliplist$ and execute edge flip. And change the $isNew$ flag of new edges into false so as not to affect the next iteration.

Fifthly, copy the new vertex positions into final $Vertex$ :: $position$. And change the $isNew$ flag of new vertices into false so as not to affect the next iteration.

Debug experience: When I commented and debugged step by step, I found that the model became very confusing after step 4 (I did not take a screenshot at that time, and there was a risk

of reproducing the bug when the deadline was approaching. In short, it has a look that makes people worry that the computer will crash soon). After checking, it was found that the $isNew$ flag was not set and checked correctly, and the edge that should not have been flipped was flipped. So I made some modifications to the *splitEdge* and frame. I added a default value of false to the $isNew$ of both the $Vertex$ class and the $Edge$. Then I set the $isNew$ of the newly added edge to true in the *splitEdge* function, because after leaving the scope of this function, I will lose the information of "which edge is newly added". In addition, in order to avoid the new edge generated by manual split from affecting subsequent up-sampling, I added code to set $isNew$ of all edges to false in step 2.

Screenshots: The screenshots of several iterations of loop subdivision on the dae/cube.dae is as following:

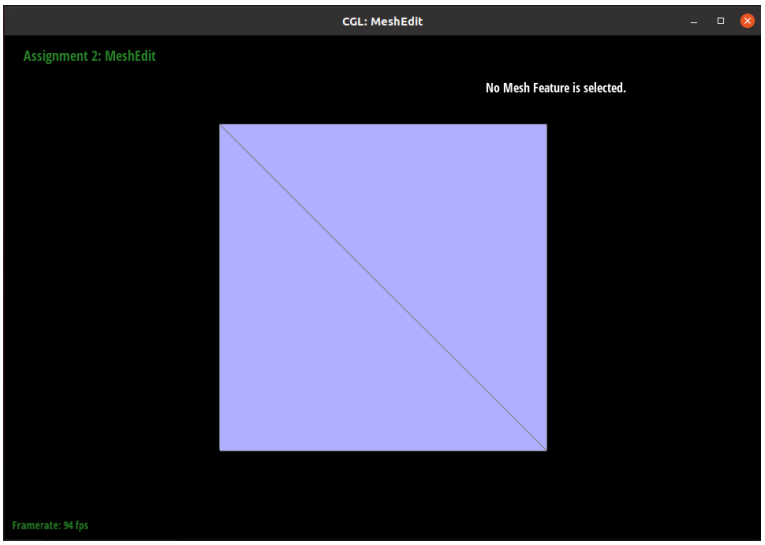


Figure 23: Iteration 0

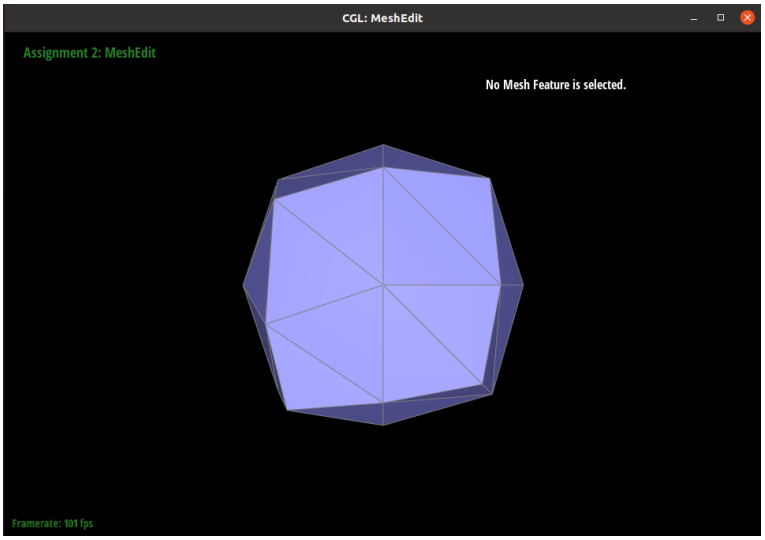


Figure 24: Iteration 1

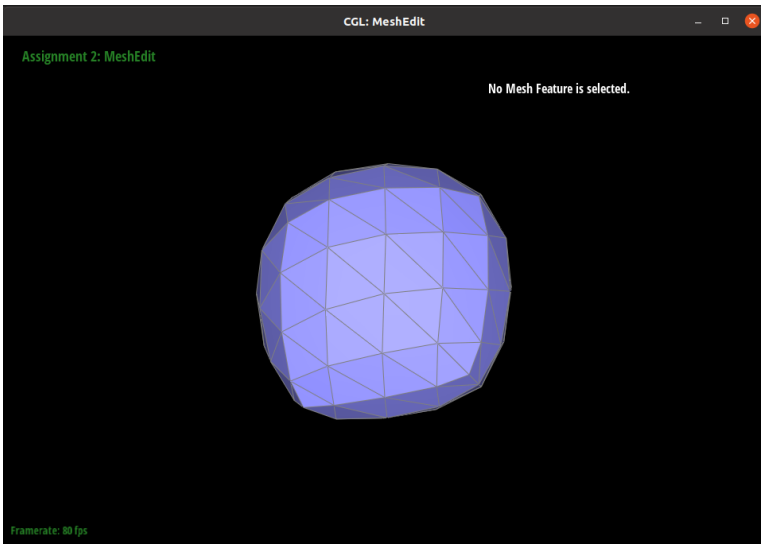


Figure 25: Iteration 2

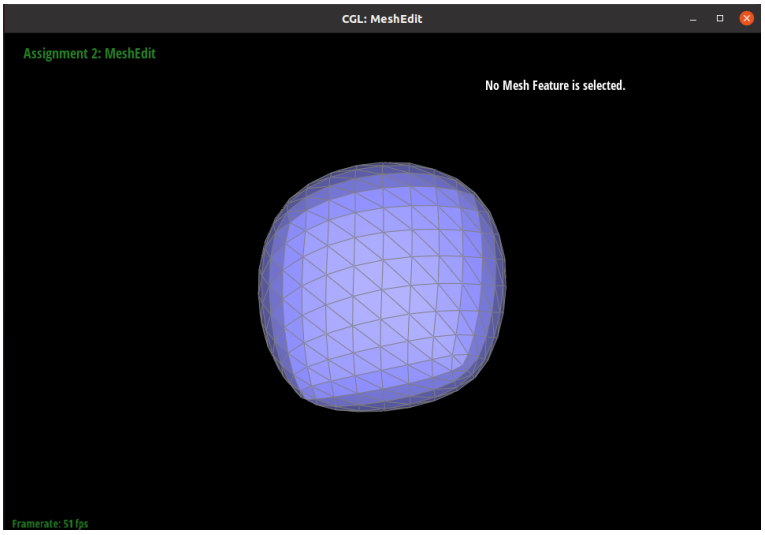


Figure 26: Iteration 3

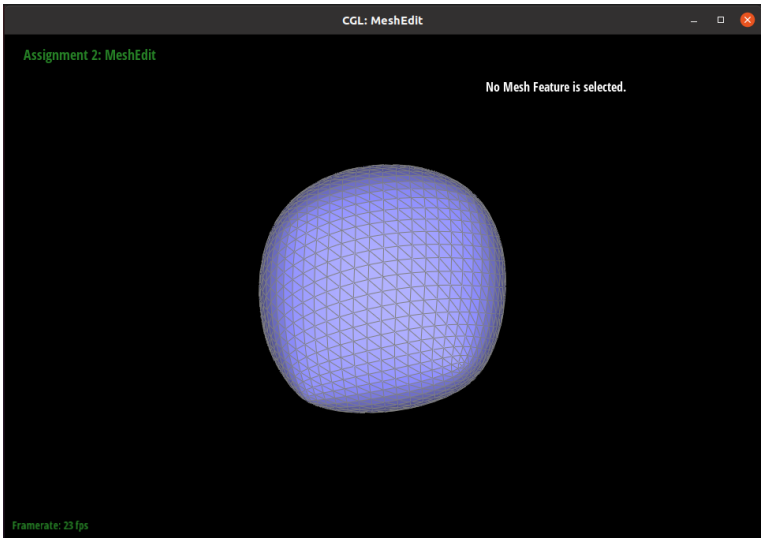


Figure 27: Iteration 4

We can see that sharp corners and edges become smooth. To reduce this effect, I pre-split the edges as following:

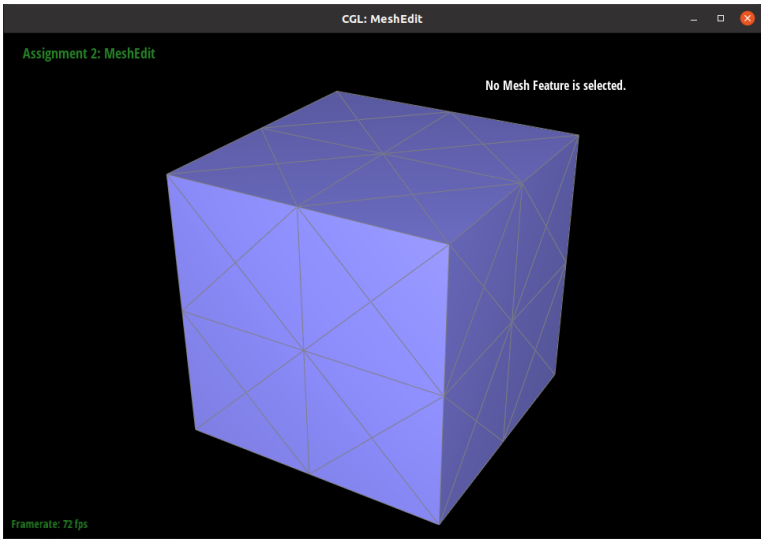


Figure 28: Before up-sample

And the result is as following:

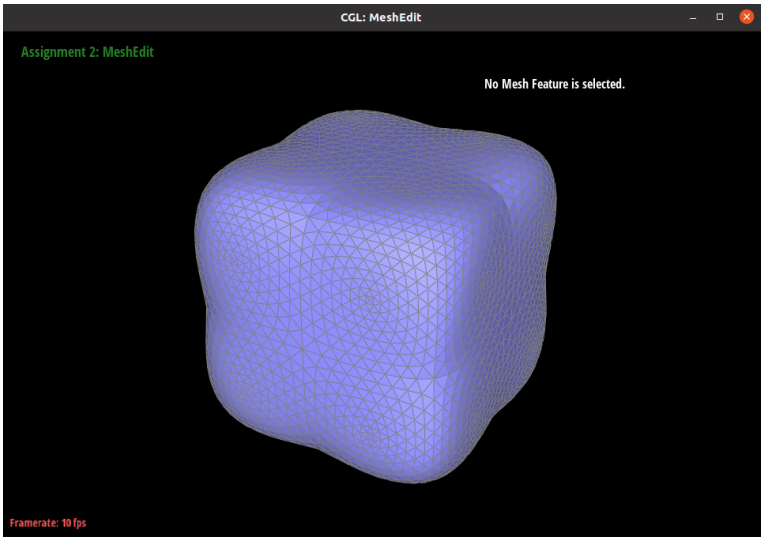


Figure 29: After up-sample

You can see the effect is relatively good. I think the basic idea to protect sharp corners and edges is to add the number of vertices at that area, so as to reduce the range of positions that the vertices can be updated to, and further reduce the effect of up-sampling.

To make the cube subdivides symmetrically, I pre-split the edges as following:

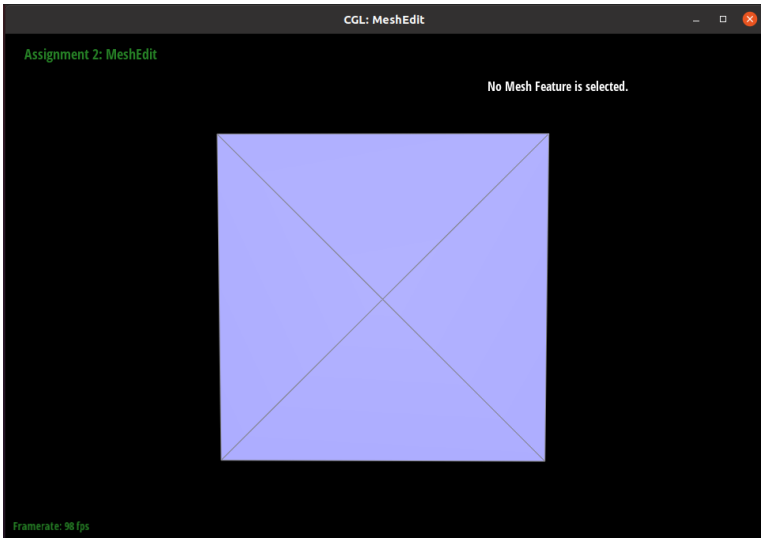


Figure 30: Before up-sampling

And the result is as following:

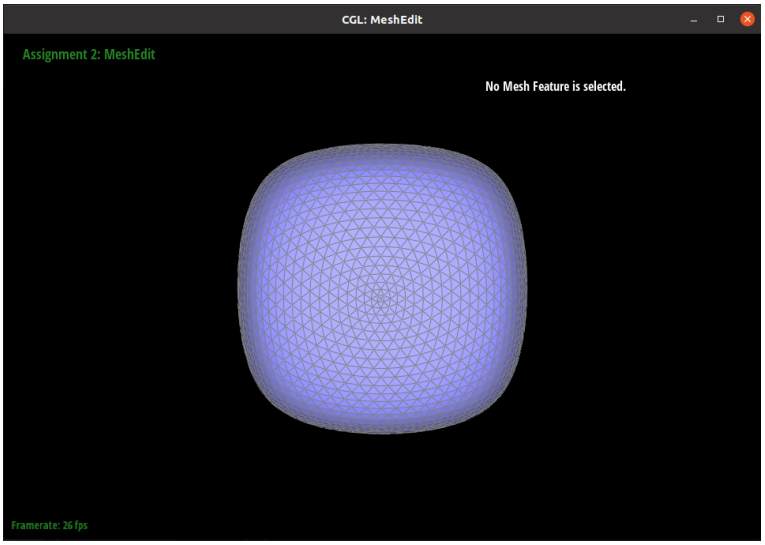


Figure 31: After up-sampling

It can be seen that the image is already symmetrical. The reason why the original cube is asymmetric after up-sampling is because of the asymmetry between original vertices. Some vertices have a degree of only 3 (the three vertices closest to the vertex on the cube), while some points have a degree of up to 5 (additionally including 2 vertices connected diagonally on the cube faces). The degree of the old vertex directly affects its new position, and the range of changes in the position of different points is therefore different, so as to produce the asymmetry. And I made the vertices symmetrical by the split the diagonal edges on the face.

Also I completed credit: support meshes with boundary. The result is as following:

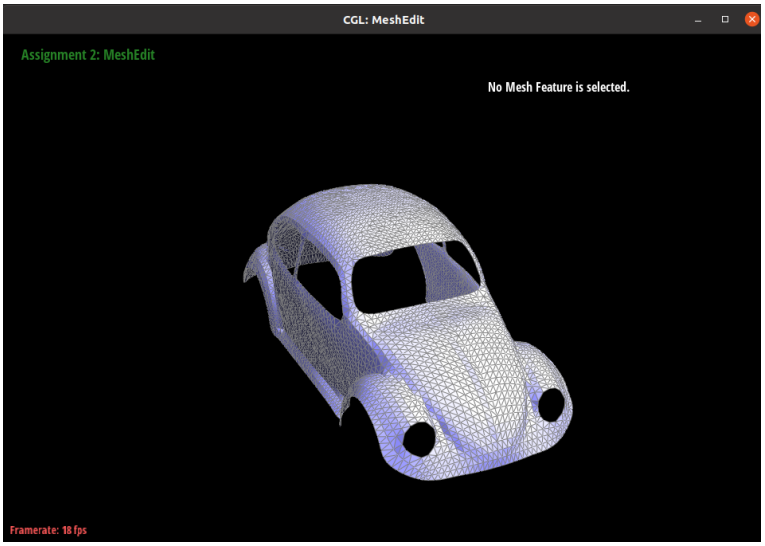


Figure 32: Before up-sample (boundary case)

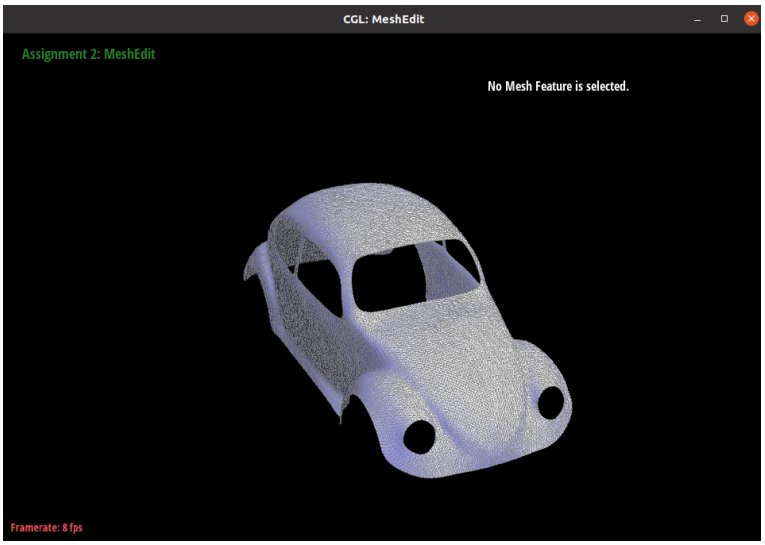


Figure 33: After up-sample (boundary case)