CSC4140 Assignment 5

Computer Graphics

April 5, 2022

Geometry

This assignment is 9% of the total mark.

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

Student ID: 119010344

Student Name: Xiao Nan

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

Signature: Xiao Nan

Contents

1 Overview

This project can be divided into two parts: Beizer curve and surface evaluation, as well as the mesh subdivision algorithms:

In the first part I implemented 1D evaluation of Bezier curve given control points and the parameter t using the De Casteljau's algorithm. I also implemented the evaluation of 2D Bezier surface by using separated 1D Bezier curve evaluation using De Casteljau's algorithm.

In the second part I implemented the calculation of vertex normal based on the area-weighted surface patch normals. Then, I also implemented basic local mesh operations including edge flip and edge split using the halfedge data structure on either the boundary or the inner plane. Based on that, I further implemented the loop subdivision algorithm for approximating a C^2 continuous smooth surface, as well as the butterfly subdivision algorithm for interpolating a C^1 smooth surface and preserving wrinkled details in the patch.

Overall, the two parts in the homework can be combined to form a simple pipeline for meshing. It first sets the Beizer using the given control points and sample on the surface to evaluate the triangular meshes based on the u, v parameters given. After getting enough triangular meshes, the program can then perform surface shading and finally render the mesh and the surface on the screen. Subdivision and vertex-normal-nased shading under this circumstance can be used to make the surface more smooth and real given the light source.

2 Bezier curves with 1D de Casteljau subdivision

2.1 Explanations and implementations of De Casteljau's algorithm

De Casteljau's algorithm is an algorithm used for evaluating a point on the Bezier curve, given the control points of that curve as well as the parameter t (for parametric equation of the Bezier curve). The recursive mathematical formula for the De Casteljau's algorithm is shown in 1:

$$\begin{cases} \beta_i^{(0)} := \beta_i, \quad i = 0, \ldots, n \\ \beta_i^{(j)} := \beta_i^{(j-1)}(1-t) + \beta_{i+1}^{(j-1)} t, \quad i = 0, \ldots, n-j, \quad j = 1, \ldots, n \end{cases} \tag{1}$$

Where n stands for the number of control points. β_i stands for the initial(input) control points. $\beta_i^{(j)}$ stands for the ith control point at iteration j. The algorithm starts from iteration 0 and in every iteration, the new set of control points are calculated based on the linear interpolation of two neighbouring previous control points (that is, $\beta_i^{(j)} = lerp(\beta_i^{(j-1)}, \beta_{i+1}^{(j-1)}, t)$ where $\beta_i^{(j-1)}, \beta_{i+1}$ are the two previous neighbouring control points). The iterations keep going until one final control point is computed (β_0^n), which is considered the evaluated point on the Bezier curve given parameter

t.

In the implementation, I defined a vector of $Vector2D$ points to store the resulting control points at each iteration, and for every element in the input control point vector ($std :: vector < Vector2D > constpoints$) I just compute the $lerp$ with respect to the input t and store the results in the vector. Finally the resulting vector is returned for the next-level control points.

2.2 Results and analysis

2.2.1 Stepwise evaluation

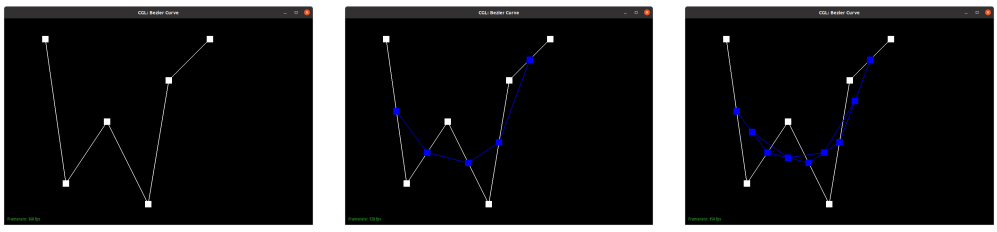
In this experiment I used a set of control points with size 6 to evaluate the Bezier curve in steps, as shown in figure 1.

runnable > HW5 > Assignment5 > bzc > crv.bzc

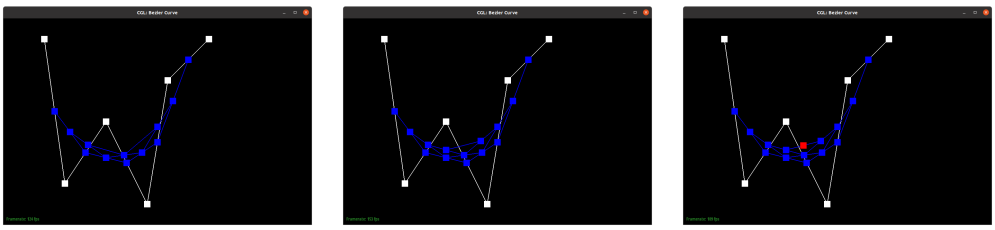
```
1   6
2   0.200 0.900   0.300 0.200   0.500 0.500   0.700 0.100   0.800 0.700   1.000 0.900
3
```

Figure 1: The 6 control points that I've chosen

The resulting intermediate control points after 0-5 steps of evaluations are shown in figure 2 (where t is selected near 0.5), where the linear interpolations at each step are shown explicitly with the demonstration of the intermediate control points.



(a) Step 0 (b) Step 1 (c) Step 2

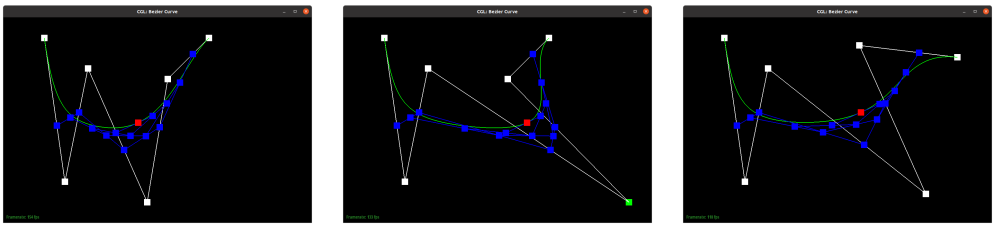


(d) Step 3 (e) Step 4 (f) Step 5

Figure 2: Resulting control points after 0-5 evaluation steps

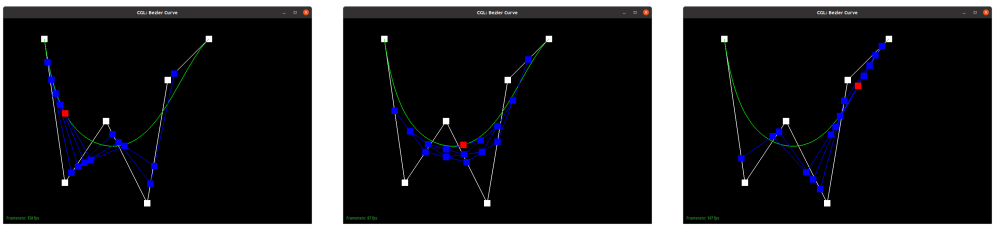
2.2.2 Modifying control points and parameter t

The screenshots for the Bezier curve with control points modified are shown in figure 3, where the shape of the curve is strongly affected due to the change of the control points. The results after changing parameter t are shown in figure 4 (t is set 0.15, 0.5, 0.85, respectively), where the points' location on the curve changes correspondingly, and as t increases the point gets closer to the rightmost end control point.



(a) (b) (c)

Figure 3: Moving control points



(a) t = 0.15 (b) t = 0.5 (c) t = 0.85

Figure 4: Changing t

3 Bezier surfaces with separable 1D de Casteljau

3.1 Bezier surface with de Casteljau and implementation

In addition to evaluating 1D point on a Bezier curve, the de Casteljau algorithm can also be applied to evaluate a point on a 2D Bezier surface (with given parameters u, v). Specifically, we can decompose the problem of sampling from the Bezier surface into two parts: First evaluating (using u) the control points for a given Bezier curve where the target point on the Bezier surface lies, and second evaluating the point's location on the curve (using v), which is equivalent to the location on the Bezier surface.

In my implementation, given the input 2D array of control points, I first iterate over each row and perform the 1D Bezier curve evaluation on the row of control points (corresponding to a given Bezier curve) to obtain the target control points (where the target point lies) using de

Casteljau's algorithm with $t = u$ and store them in another vector $std :: vector < Vector3D > target_approx_col$. Then I apply de Casteljau's algorithm again to evaluate with $t = v$ on the target Bezier curve so that I can obtain the target point's location and return it back.

3.2 Results and analysis

The result for evaluation on *bez/teapot.bez* is shown in figure 5, where the teapot's triangular mesh, evaluated from the pre-defined Bezier surface, is smooth and continuous enough for visualization.

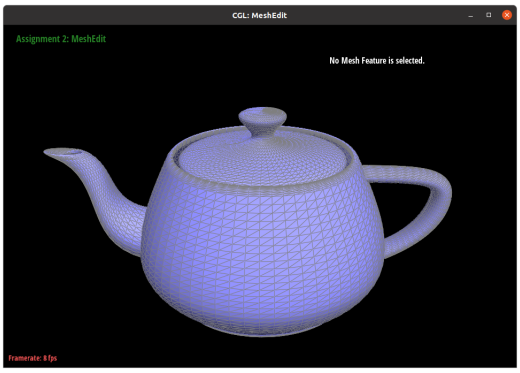


Figure 5: The 6 control points that I've chosen

4 Area-weighted vertex normals

4.1 Area-weighted vertex normal evaluation

The vertex normal is defined to be the area-weighted surface normals, then normalized by the norm of that vector, as shown in 2.

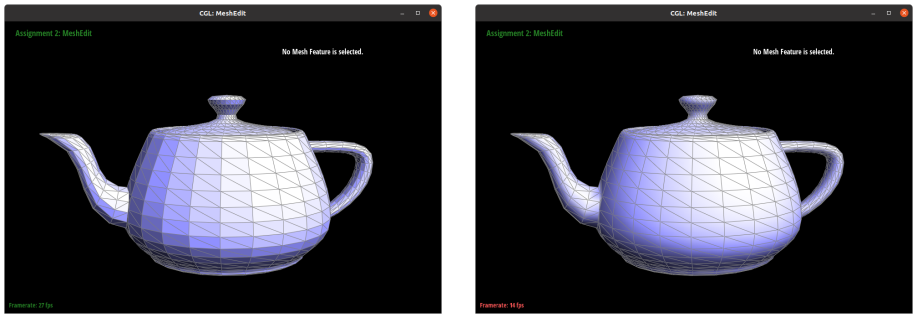
$$\begin{cases} n_V^{'} = \sum_i A_{S_i} n_{S_i} \\ n_V = \frac{n_V^{'}}{\|n_V^{'}\|} \end{cases} \tag{2}$$

Where V is the target vertex. S_i represents the face that has the target vertex as one of its component vertices. A_{S_i} represents the area of S_i. n_{S_i} represents the normalized normal vector of surface S_i. $\left\|n_V^{'}\right\|$ is the norm of $n_V^{'}$.

I implemented it by first computing the cross products of the edge vectors adjacent to the vertex (since it takes the area weights into account already), then adding the results together and normalizing that.

4.2 Results for vertex-normal-based shading

The screenshots for teapot shading with and without vertex normals are shown in figure 6, where the use of vertex normal makes the shading more smooth.



(a) (b)

Figure 6: Shading without and with vertex normal

5 Edge flip

5.1 Edge flip implementations

I implemented the edge flip operations by first determining what attributes in the mesh elements need changing (as shown in the upper part of figure 7, which is the draft when I was writing my program). After that, I build a model specifying what should or should not be changed by edge flipping strictly (that is, maintaining the relationships between edges, halfedges, vertices and faces before and after edge flipping with corresponding letters). Finally, I implement the changes on old vertices, edges, halfedges and faces in sequence in my program (carefully, since the modification of the halfedge data structure, similar to the linked list, can easily go wrong).

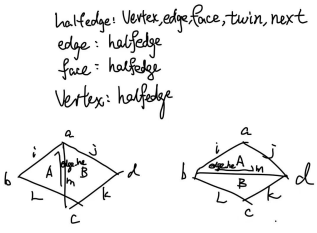


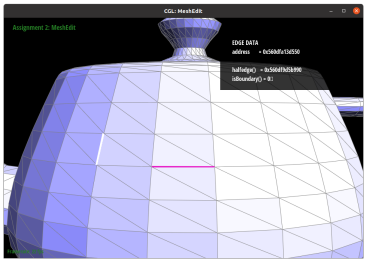
Figure 7: Flipping draft before implementation

Note that since the filpping of boundary edges are not allowed, the flipping function does nothing and returns the edge directly when the boundary edge is taken as input. For debugging, the $check_for()$ functions provided in the templete checking the mesh elements that are pointing to the input given mesh element are useful (for checking whether or not one forgets halfedge

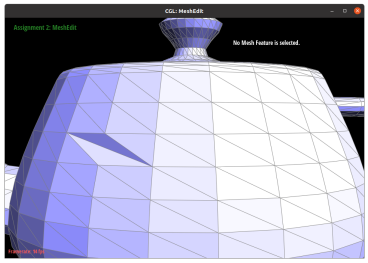
connections). On the other hand, *cout* is a powerful tool for printing out intermediate results for vertices' positions and halfedge data (and I used it a lot).

5.2 Edge flip results

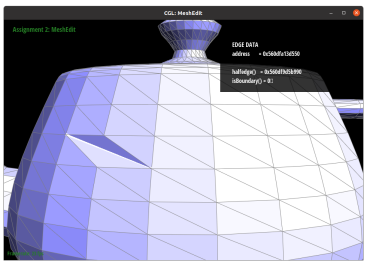
The results for the mesh before and after edge flippings are shown in figure 8, where figure 8(a), 8(b) show the mesh before and after flipping the selected (white) edge, respectively. Figure 8(c), 8(d) show the results for flipping the edge back. Figure 8(e), 8(f) demonstrate the results for flipping a boundary edge (which is not successful since boundary edge flipping is not allowed). We may observe that edge flipping changes the shading patterns nearby the flipped edge (given the light source) by increasing the contrast between the two faces adjacent to the flipped edge.



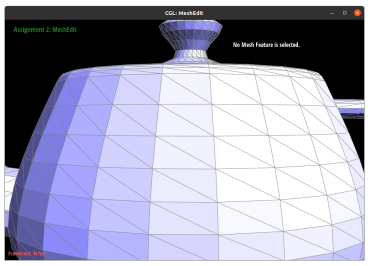
(a) Before the first flip



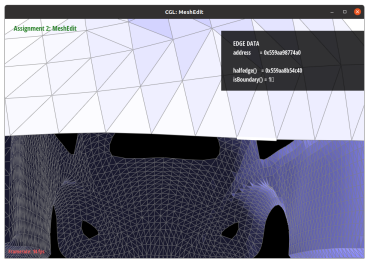
(b) After the first flip



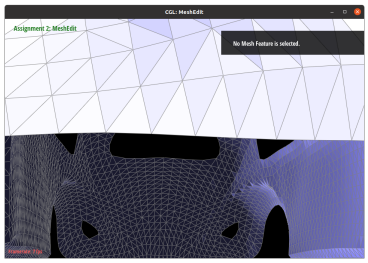
(c) Before the second flip



(d) After the second flip



(e) Before the edge flip



(f) After the edge flip

Figure 8: Edge flip results

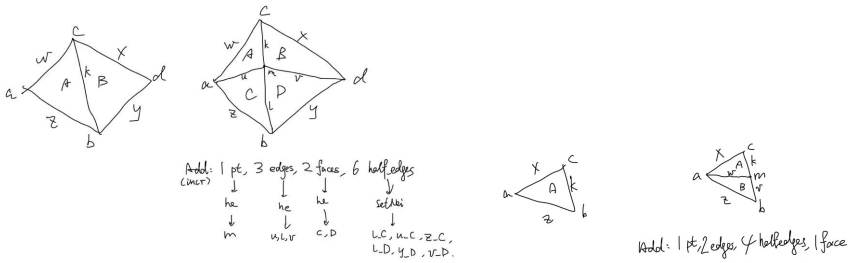
And it's really fortunate that after my careful planning of modifications and programming, I run through the edge flipping code once xD. But I suffered a lot more when implementing edge split operations and subdivision schemas.

6 Edge split (for both inner and boundary mesh)

6.1 Edge split implementation

Similar to edge flipping, I implemented the edge split operations by first drafting on the edges to spilt (including boundary case and non-boundary case, as shown in figure 9). After that, I judge if the target edge is the boundary edge: If it isn't the boundary, I create the new mesh elements (by invoking $mesh.newxxx()$ method of the mesh) and initialize them according to my draft. Then I iterate through and update the old mesh elements with the values specified in the draft. If the edge is the boundary, the program would similarly create another set of mesh elements and update the attributes of the old and new mesh elements.

Note that even though some edges (e.g. l in 9(a) and k in 9(b)) are newly created, I set their $isNew$ attribute to $false$ for the convenience of updates, as well as for the simplicity of loop subdivision later (which needs distinguishing the new inner edges).

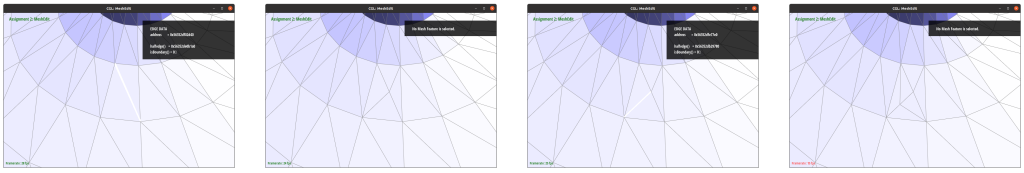


(a) Splitting a non-boundary edge (b) Splitting a boundary edge

Figure 9: Edge splitting drafts for non-boundary/boundary cases

6.2 Results and analysis for edge splitting, and flippings afterwards

The results for edge splitting and flipping operations (with non-boundary edges) are shown in figure 10, where 2 edge split operations and 2 edge flip operations are performed in sequence. Figures 10(a), 10(b), 10(c), 10(d) demonstrate the correctness of the normal edge splitting and figures 10(e), 10(f), 10(g), 10(h) demonstrate the correctness of edge flipping after splitting. I observe that edge splitting alone doesn't change the shading patterns nearby, but it does help preserve some sharp edges/corners by adding more edges and vertices as the subdivision goes on.



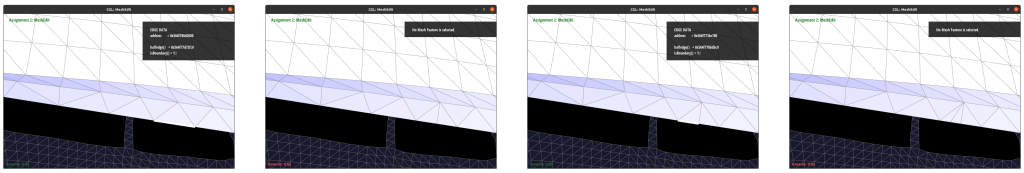
(a) Before the first split (b) After the first split (c) Before the second split (d) After the second split

(e) Before the first flip (after 2 splits) (f) After the first flip (after 2 splits) (g) Before the first flip back (after 2 splits and 1 flip) (h) After the first flip back (after 2 splits and 1 flip)

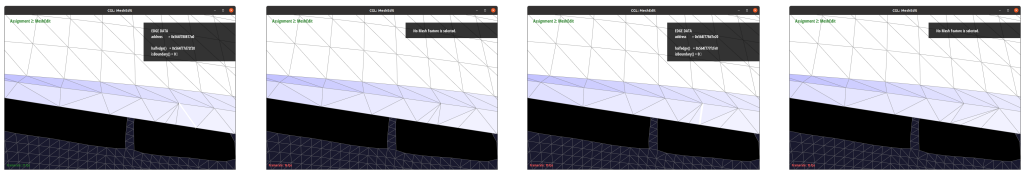
Figure 10: Edge splitting and flipping for non-boundary cases

6.3 Results and analysis for boundary edge splitting

The results for splitting boundary edges are shown in figure 11, where 11(a), 11(b), 11(c), 11(d) demonstrate the correctness of splitting and 11(e), 11(f), 11(g), 11(h) demonstrate that of flipping after splitting. I also observe that after splitting, the sharpness of the boundary is better kept after iterations of subdivisions, indicating that splitting on the boundary, similar to the non-boundary case, has the effect of preserving sharpness during subdivisions.



(a) Before the first boundary split (b) After the first boundary split (c) Before the second boundary split (d) After the second boundary split



(e) Before the first flip (after 2 splits) (f) After the first flip (after 2 splits) (g) Before the second flip back (after 2 splits and 1 flip) (h) After the second flip back (after 2 splits and 1 flip)

Figure 11: Edge splitting and flipping for boundary cases

7 Loop subdivision for mesh upsampling (with boundaries supported)

7.1 Implementations of loop subdivision

My implementation of loop subdivision consists of 3 steps:

In the first step I calculates the new positions of all the old vertices as well as the edge midpoints: Specifically, I first iterate through all vertice and for each vertex I update its $newPosition$ attribute by querying the neighbouring vertices using the vertex's halfedge and perform weighted sum, and after that I iterate through all edges to compute the new points' new positions and store them in the corresponding edge.

The way in which the approximated edge midpoints and vertices' coordinates are computed is shown in figure 12:

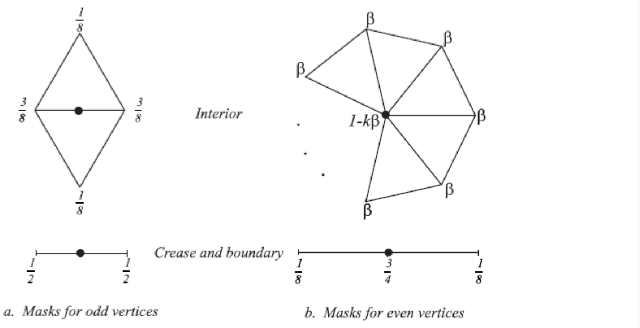


Figure 4.3: *Loop subdivision: in the picture above, β can be chosen to be either* $\frac{1}{n}(5/8-(\frac{3}{8}+\frac{1}{4}\cos\frac{2\pi}{n})^2)$ *(original choice of Loop [16]), or, for* $n>3$, $\beta=\frac{3}{8n}$ *as proposed by Warren [33]. For* $n=3$, $\beta=3/16$ *can be used.*

Figure 12: Loop subdivision algorithm

For the boundary edge points, I average the two adjacent vertices' positions as the new position for the edge midpoint. For the boundary vertices, I computed the weighted sum of the adjacent vertices' positions and the vertex's position (with weights specified in lower right in 12) as the new position. For the non-boundary edge points, I iterate through the two triangles glued by the edge on their vertices, and sum their positions with weights specified in upper left in 12. For the non-boundary vertices, I compute the weighted sum of the adjacent vertices' positions, as well as the position of the vertex itself, with weights determined by a cosine function of the vertex degree, as specified in upper right in 12.

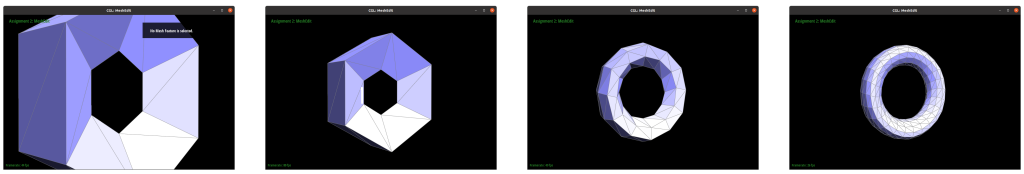
In the second step I split all the old edges and flip the new edges that connect an old vertex to a new vertex. Specifically for splitting, I back up the end iterator of the previous edge iterator vector (*edgend*) and use that to signal the iteration end, so that new edges won't be split and the loop will always end. For flipping, as mentioned in the previous chapter, I set the $isNew$ data field

of the new edge that is parallel to the old edge to be $false$ (that is, regarding the parallel edge as an old edge), so that the edge wouldn't be flipped wrongly.

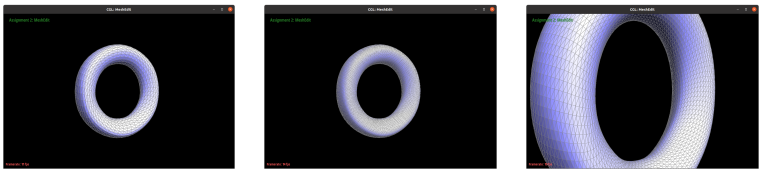
In the third step I update the new positions of all vertices (old and new) in correspondence to the positions that had been computed in the first step. For the old vertices, I directly update using the *newPosition* data field. For the new vertices, I update with the *newPosition* stored in its nearest old edge. Note that since both two edges, the real old edge and the new edge that is parallel to the original old edge (mentioned in the second step), may be the nearest old edge simultaneously, I set the *newPosition* data field of the "pseudo" old edge (the new edge parallel to the real old edge) during creation in the second step so as to avoid the case where the "pseudo" old edge is queried and the new position fails to be updated.

7.2 Results and analysis for sharp corners and edges

The results for loop subdivision for sharp corners are shown in figure 13, where obviously the sharp characteristics are smoothed and hence lost as the number of iterations increase.



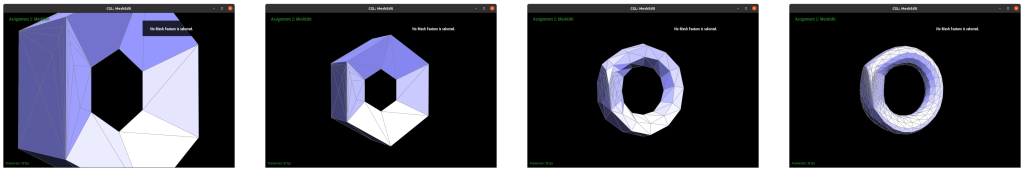
(a) Zoom in for the original sharp corner (b) Before loop subdivision (c) After the 1st first loop subdivision (d) After the 2nd first loop subdivision



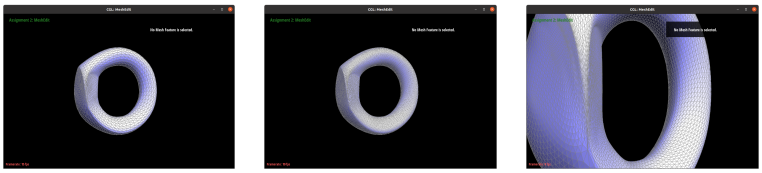
(e) After the 3rd first loop subdivision (f) After the 4th first loop subdivision (g) Zoom in, after 4 iterations

Figure 13: Loop subdivision for sharp corner (without pre-splitting)

I resolved the smoothing problem by pre-splitting the edges that are adjacent to the sharp edge, hence making the edges near the sharp cornor more dense and strengthening the surface by making it more difficult to be smoothed and subdivided. The results after improvement are shown in figure 14, where the surface around the cornor preserves sharpness after 4 iterations of loop subdivision.



(a) The original sharp corner after pre-splitting (b) Before loop subdivision (c) After the 1st first loop subdivision (d) After the 2nd first loop subdivision

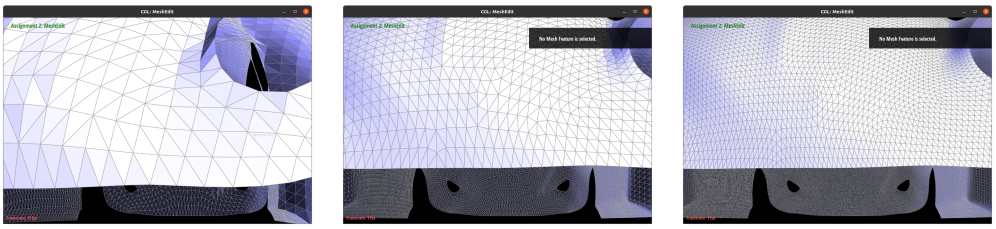


(e) After the 3rd first loop subdivision (f) After the 4th first loop subdivision (g) Zoom in, after 4 iterations and pre-splitting

Figure 14: Loop subdivision for sharp corner (with pre-splitting)

7.3 Results on the boundary

The results for the subdivision on the boundary are shown in figure 15, demonstrating the success on the boundary.

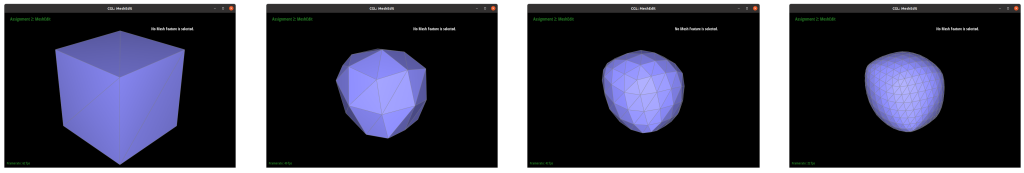


(a) Before subdivision (b) After the 1st subdivision (c) After the 2nd subdivision

Figure 15: Loop subdivision at the boundary

7.4 Results and analysis for the cube

The results for subdividing the cube for 3 iterations and observing the asymmetry are shown in figure 16, where the originally symmetric cube (16(a)) becomes asymmetric (16(d)) when observed from the diagonal view.

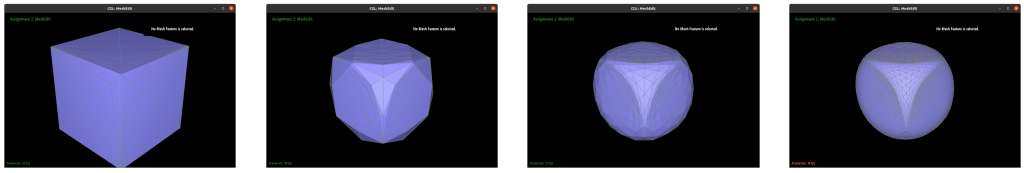


(a) Before subdivision (b) After the 1st subdivision (c) After the 2nd subdivision (d) After the 3rd subdivision, the cube seems asymmetric

Figure 16: Loop subdivision for the cube (without pre-splitting)

This is because, as shown in figure 16(a), even though the resulting surface is symmetric, the mesh elements composing the cube surface are asymmetric (in the positions of the vertices, edges and halfedges), making the subdivisions take place in an asymmetric way. As the number of subdivision iterations increase, such asymmetric subdivision error accumulates, which makes the subdivided surface seemingly more and more asymmetric.

The solution is to split the asymmetric edges near the corner in order to make the mesh elements (edges, faces, halfedges, vertices) symmetric (in positions) with respect to the camera. In this way the subdivision operates in a symmetric way (by subdividing symmetric edges and adding symmetric points, faces and halfedges), resulting in symmetric subdivided surfaces, as shown in figure 17(d).



(a) Before subdivision (b) After the 1st subdivision (c) After the 2nd subdivision (d) After the 3rd subdivision, the cube becomes symmetric

Figure 17: Loop subdivision for the cude (with pre-splitting)

8 Modified butterfly subdivision for mesh upsampling (with boundaries supported)

8.1 Implementations of modified butterfly subdivision

Similarly, my implementation of modified butterfly subdivision can be divided into 3 steps:

In the first step I compute the new positions for the new points (the interpolated points) and store them in the corresponding edges.

The way in which the interpolated edge points' coordinates are computed is shown in figure 18:

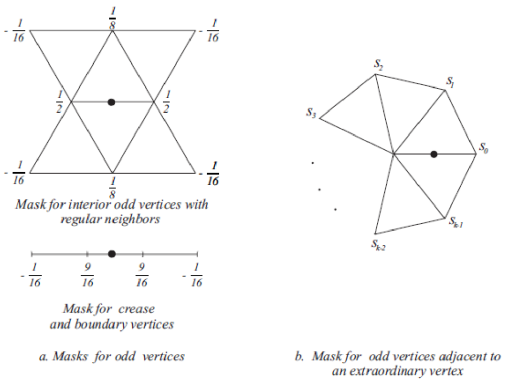


Figure 4.5: *Modified Butterfly subdivision. The coefficients* s_i *are* $\frac{1}{k}\left(\frac{1}{4}+\cos\frac{2i\pi}{k}+\frac{1}{2}\cos\frac{4i\pi}{k}\right)$ *for* $k>5$. *For* $k=3$, $s_0=\frac{5}{12}$, $s_{1,2}=-\frac{1}{12}$; *for* $k=4$, $s_0=\frac{3}{8}$, $s_2=-\frac{1}{8}$, $s_{1,3}=0$.

Figure 18: Modified butterfly algorithm

For the boundary edge points, I compute the coordinates using the sum of the positions of the two adjacent vertices (with weight $\frac{9}{16}$) as well as the two boundry vertices adjacent to the two adjacent vertices, correspondingly (with weights $\frac{-1}{16}$, indicated in the lower left in 18).

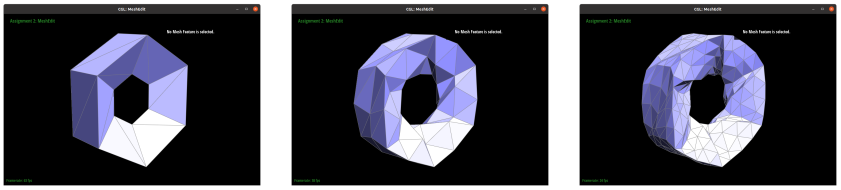
For the non-boundary edge points, if the two vertices corresponding to the edge are regular (with degree=6), then the new position would be computed using the positions of the vertices adjacent to the two vertices, respectively, with weights specified in upper left in 18. And if one of the vertices is irregular (with degree not equal to 6), the position would be computed using the weighted sum of the irregular vertex's position together with its adjacent vertices (with weights indicated in the right in 18). Furthermore, if two vertices are irregular, the new position would be computed based on the average of the new positions computed from the two irregular vertices.

In the second step I split the old edges and perform edge flips on the new edges that connect an old vertex to a new vertex, which is similar to the second step of the loop subdivition.

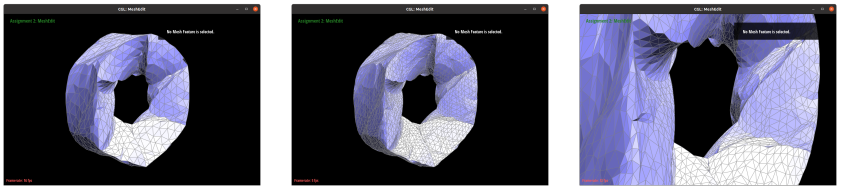
In the third step I simply update the positions of the new vertices (since the butterfly subdivision is an interpolation-based algorithm, which only interpolates new vertices instead of modifying old vertices) with the corresponding positions stored in the old edges.

8.2 Results and analysis for sharp corners and edges

The results for the modified butterfly subdivision on sharp edges are shown in figure 19, where we observe that the sharp edges (high-frequency features) on the corners of the toru are preserved (and somehow, the degree of sharpness is even enhanced as the number of subdivisions increase), and more detailed wrinkles around the sharp corners also arise, making the sharp corners more complex in shapes.



(a) Before subdivision (b) After the 1st subdivision (c) After the 2nd subdivision

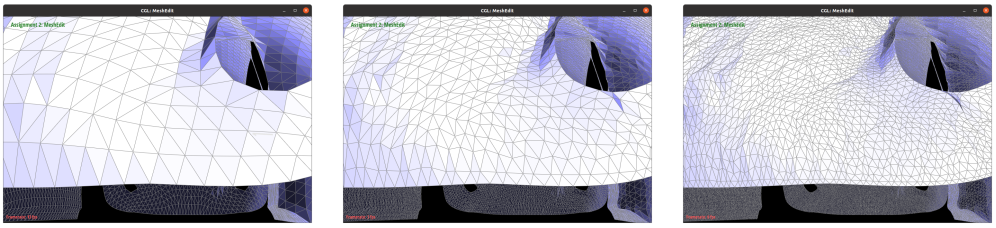


(d) After the 3rd subdivision (e) After the 4th subdivision (f) Zoom in to see that high-frequency features are preserved

Figure 19: Modified butterfly subdivision for the toru

8.3 Results on the boundary

The results for the subdivision on the boundary are shown in figure 20, demonstrating the success on the boundary.

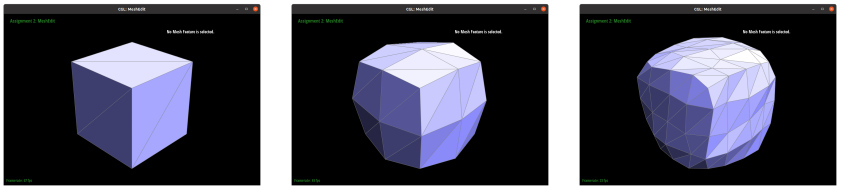


(a) Before subdivision (b) After the 1st subdivision (c) After the 2nd subdivision

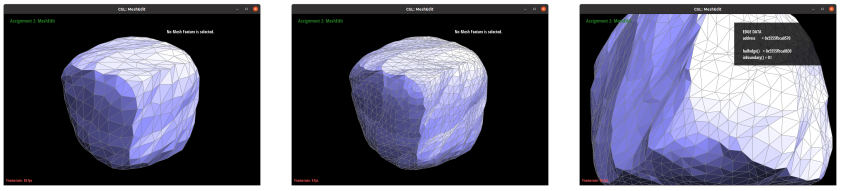
Figure 20: Modified butterfly subdivision at the boundary

8.4 Results and analysis for the cube

The results for applying butterfly subdivision on the cube are shown in figure 21, where the resulting "cube"s are still asymmetric (due to the asymmetric mesh elements' configurations). In addition, I notice that the size of the cube doesn't vary a lot, which potentially is because of the fact that the interpolating modified butterfly algorithm doesn't change the positions of the old vertices, making the general shape and size of the object more stable.



(a) Before subdivision (b) After the 1st subdivision (c) After the 2nd subdivision



(d) After the 3rd subdivision (e) After the 4th subdivision (f) Zoom in to see that the size is preserved

Figure 21: Modified butterfly subdivision for the toru

9 Comparisons between the modified butterfly subdivision and the loop subdivision

Combining the results obtained from the last two sections, the butterfly subdivision may have the following advantages in subdividing meshes in reality:

First, it preserves the high-frequency information from the original signal, hence preventing the loss on high-frequency details of the original signal during subdivisions. Comparing the results from 19 and 13, we may see that the loop subdivision acts as a strong low-pass filter filtering almost all high-frequency sharp edges and corners, making the toru almost entirely smooth after merely 3 iterations, but the modified butterfly subdivision preserves those sharp edge details well by just making those edges a bit more continuous with adjacent edges instead of sacrificing their sharpness for stronger continuity.

Then, the butterfly subdivision preserves the size of the original object. Comparing the results from 21 and 16, we may see that after several iterations of the butterfly subdivision, the cube's size doesn't vary much, whereas for the loop subdivision after just one iteraiton, 16(b) already has a size evidently smaller than 16(a). This can be attributed to the fact that the loop subdivision, an approximating subdivision algorithm, modifies all vertices' positions with the weighted sum of the neighbouring vertices, hence making the object size more fragile to changes as the subdivision goes. On the other hand, as mentioned in this section, modified butterfly algorithm preserves the old vertices' positions, which increases the stability of size.

However, the modified butterfly subdivision may be unsatisfactory in the following two aspects,

in comparison to the loop subdivision:

The modified butterfly subdivision preserves a lower level of continuity than that of the loop subdivision. Specifically, the modified butterfly algorithm preserves C^1 continuity for triangular meshes (which already improves compared to the un-modified version having only C^0 in irregular points). But the loop subdivision generates C^2 continuous surfaces except at extraordinary vertices, where it guarantees C1 continuity. Comparisons between 19(d) and 16(d) also demonstrate evidently the differences on smoothness between the two algorithms. As a result, the loop subdivision gives a smoother result, which beats the modified butterfly algorithm especially under the circumstances where circular (or other smooth) objects need rendering.

In a word, the problem of the choice between the two algorithms is task-dependent: One may choose the modified butterfly subdivision when an object with complex shapes and corners (like a human) needs rendering (subdividing). One may also choose the loop subdivision when rendering a smooth object (like a raindrop, or a ball).