CSC4140 Assignment V

Computer Graphics

April 5, 2022

Geometry

This assignment is 9% of the total mark.

Strict Due Date: 11:59PM, April 5^{th}, 2022

Student ID: 118010202

Student Name: Zhixuan LIU

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

Signature: Zhixuan Liu

1 Task1: Bezier curves with 1D de Casteljau subdivision

1.1 Intro of Bezier curves

De Casteljau's algorithm is a recursive method that allows us to calculate a Bezier curve from a set of points. The process for doing so is to connect the points to create edges; we then evaluate each edge at a parameter in between 0 and 1.0, which represents a fraction of the distance from one point to the other point. We connect these points to get a connection of edges that is one less than the number of edges previously. The recursive linear interpolation steps finish with one point, allowing us to come up with a tangent point which the draw our curve to.

I implemented it using a recursive feeling methods. This function only do a Bezier curve step, and store the intermediate points into the buffer, and therefore can be shown on the screen.

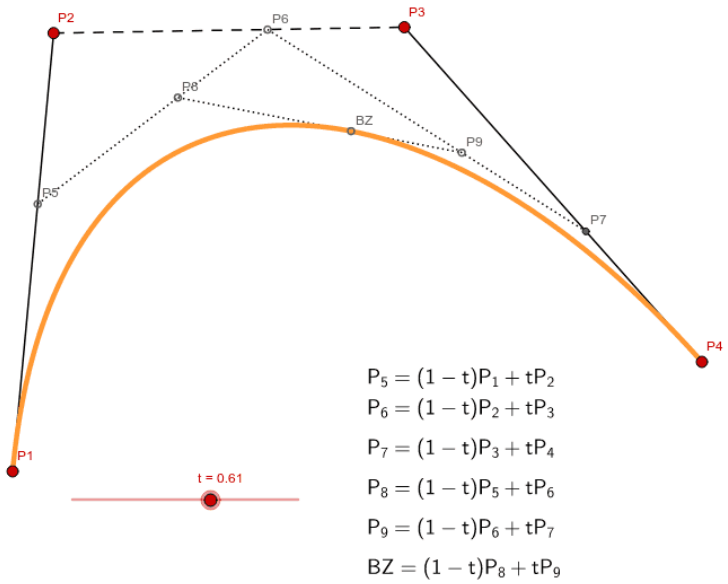


Figure 1: The rational of Bezier curves

```
1  std::vector<Vector2D> BezierCurve::evaluateStep(std::vector<Vector2D> const &points)
2  {
3    // TODO Task 1.
4    if (points.size() == 1) {
5      return points;
6    }
7
8    vector<Vector2D> nextControlPoints;
9    for (int i=0; i<points.size() - 1; i++){
10     Vector2D newPoint = (1-t)*points[i] + (t) * points[i+1];
11     nextControlPoints.push_back(newPoint);
12   }
```

```
13      return nextControlPoints;
14    }
```

1.2 Define my own 6 points Bezier curves

```
1  6
2  0.200 0.350  0.300 0.600  0.500 0.750  0.700 0.450 1.000 0.900 0.600 0.100
```

1.3 Screen Shots of Task 1

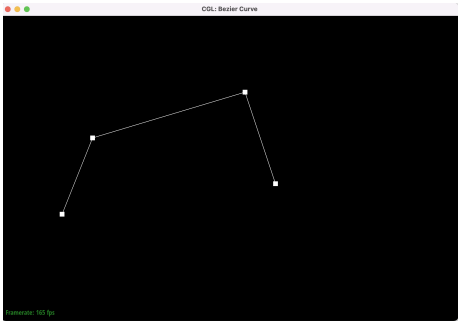


Figure 2: Bezier curves with 4 control points stage 1.

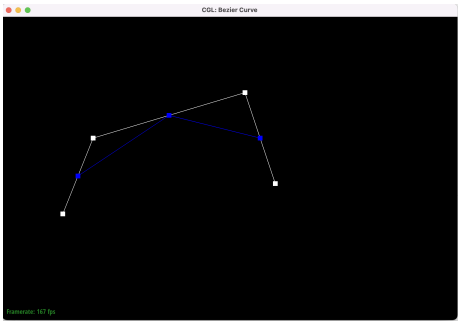


Figure 3: Bezier curves with 4 control points stage 2.

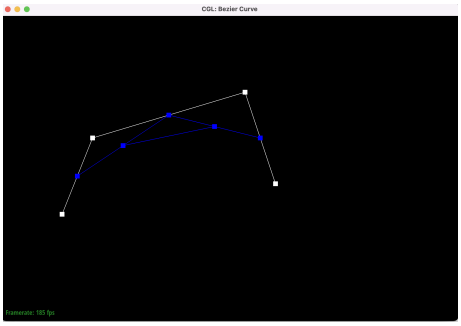


Figure 4: Bezier curves with 4 control points stage 3.

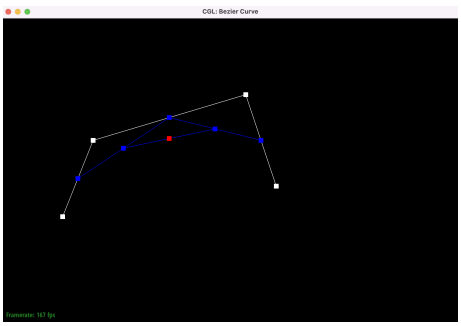


Figure 5: Bezier curves with 4 control points stage 4.

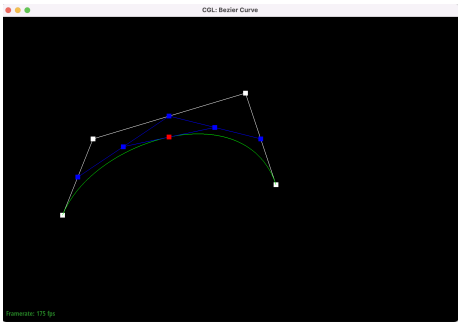


Figure 6: Bezier curves with 4 control points stage 5.

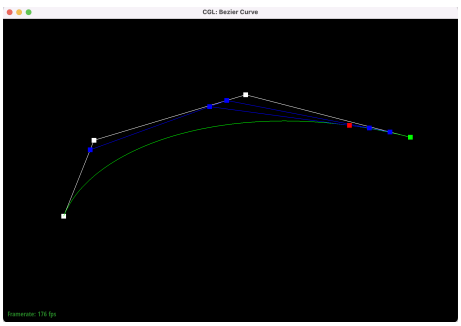


Figure 7: Bezier curves with 4 control points slightly modified.

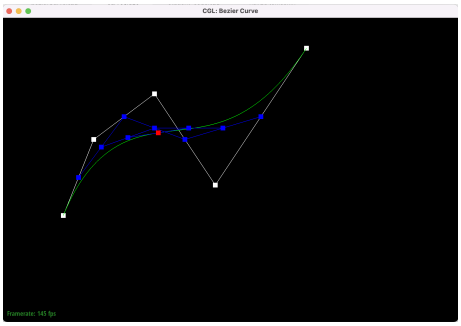


Figure 8: Bezier curves with 5 control points stage 1.

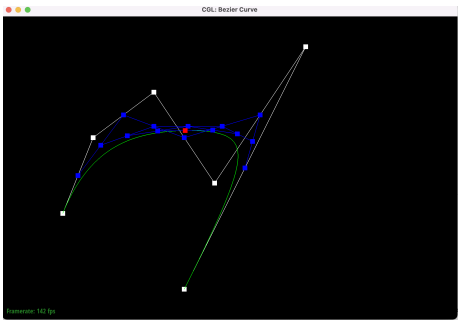


Figure 9: Bezier curves with 6 control points stage 2.

2 Task 2: Bezier Surfaces with Separable 1D de Casteljau

Bézier surfaces are a species of mathematical spline used in computer graphics, computer-aided design, and finite element modeling. As with Bézier curves, a Bézier surface is defined by a set of control points. Similar to interpolation in many respects, a key difference is that the surface does not, in general, pass through the central control points; rather, it is "stretched" toward them as though each were an attractive force. They are visually intuitive, and for many applications, mathematically convenient.

De Casteljau's algorithm extends to surfaces because I can apply what I did for the 1 dimensional curve, and just add another dimension to create a 2 dimensional surface. Applying multiple Bezier curves together would create a Bezier surface. I used a method called "Separable 1D de Casteljau", which does Casteljau in one dimension (the u dimension) first, and then uses those values to compute Casteljau in the second dimension (the v dimension). The visulization is in below:

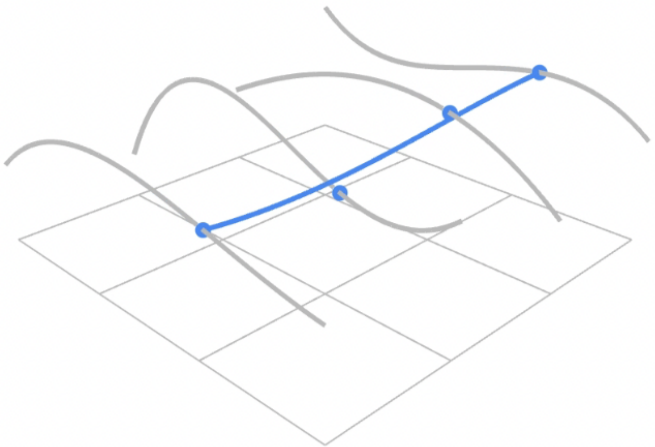


Figure 10: Bezier Surfaces with Separable 1D de Casteljau

And the code implementations are as follows:

2.1 Define my own 6 points Bezier curves

```
/**
 * Evaluates the Bezier patch at parameter (u, v)
 * @param u       Scalar interpolation parameter
 * @param v       Scalar interpolation parameter (along the other axis)
 * @return Final interpolated vector
 */
Vector3D BezierPatch::evaluate(double u, double v) const {
  // TODO Task 2.
  // go trough the rows first
  vector<Vector3D> verticalControlPoints;
  for (int i=0; i<controlPoints.size(); i++){
    verticalControlPoints.push_back(evaluate1D(controlPoints[i],u));
  }
  Vector3D results = evaluate1D(verticalControlPoints, v);
  return results;
}
```

2.2 Task 2 Results

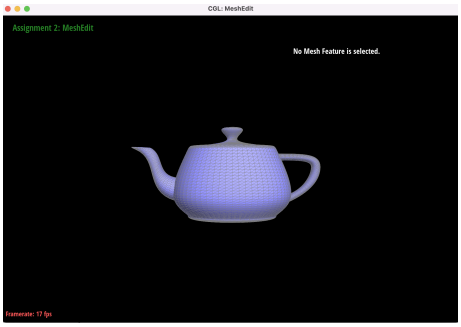


Figure 11: Bezier teapot in mesh form

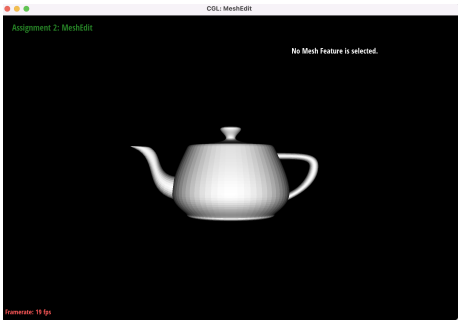


Figure 12: Bezier teapot in texture mapping form.

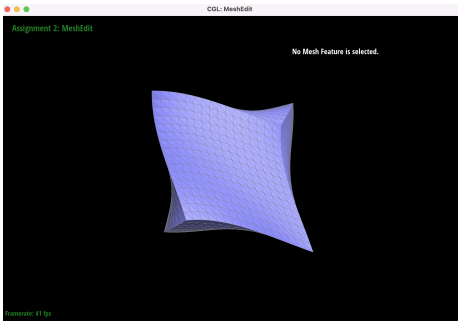


Figure 13: Bezier twavy cube in mesh form

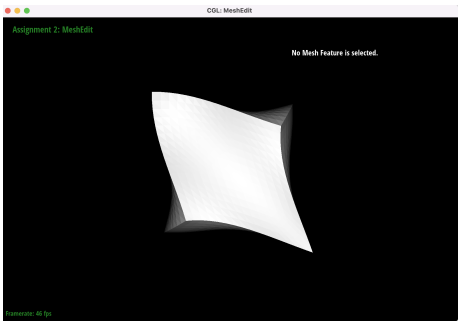


Figure 14: Bezier twavy cube in texture mapping

3 Task 3: Area-Weighted Vertex Normals

We want to implement the normal() function on a vertex. So the first step is to get the representation of the vertex.Here we use its halfedge iterator based on the object class.

After getting its position, we use the $halfedge- > twin()- > next()$ methods to go around the vertex until we get the original edge. We use the cross product to calculate the area and return the sum of area vector, and the do the norm.

```
1    while (this_edge != target){
2      this_twin = this_edge->twin();
3      next_edge = this_twin->next();
4      p1 = this_twin->vertex()->position;
5      p2 = next_edge->next()->vertex()->position;
6      edge1 = target_p - p1;
7      edge2 = p2 - target_p;
8      total_sum += cross(edge1, edge2)/2;
9      this_edge = next_edge;
10   }
```

This has a function of smoothing the edges and vertexes because the vertex is normalized by its surrounding areas.

3.1 Task 3 Results

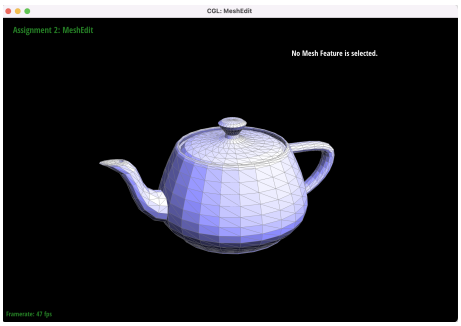


Figure 15: Task3 teapot without normal function

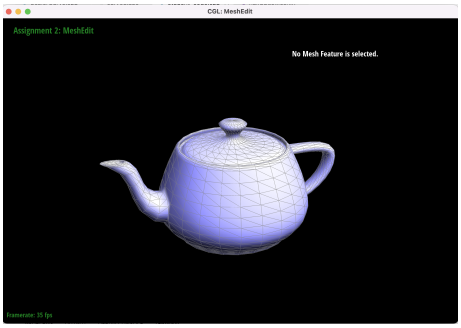


Figure 16: Task3 teapot with normal function

4 Task 4: Edge Flip

Given a pair of triangles (a,b,c) and (c,b,d), a flip operation on their shared edge (b,c) converts the original pair of triangles into a new pair (a,d,c) and (a,b,d), as shown below:

My method for flipping half-edges was to keep track of all of the half-edges inside both triangles, and having the faces, edges, and vertices point to one of the two middle half-edges. After that, I used the setNeighbors function to update the other half-edges to make two different triangles.

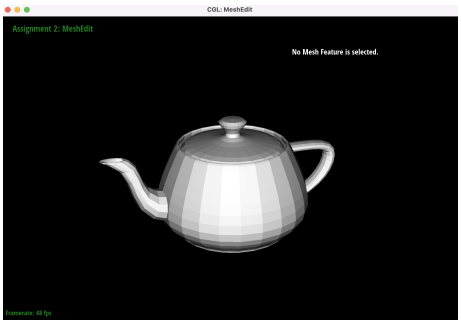


Figure 17: Task3 texture teapot without normal function

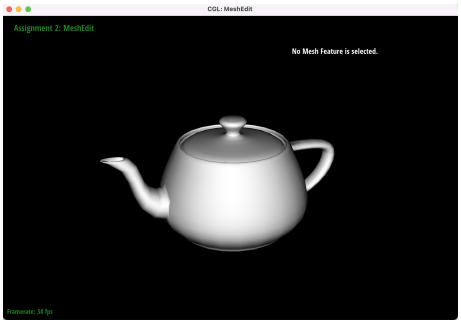


Figure 18: Task3 texture teapot with normal function.

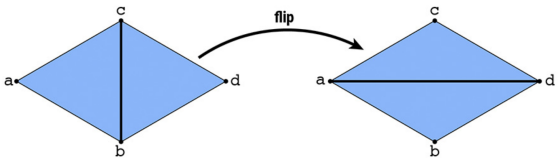


Figure 19: Flip description

I tried two ways to set the edges and flipping methods as follows, and the results show that the flipping directions or ways will not affect the final results.

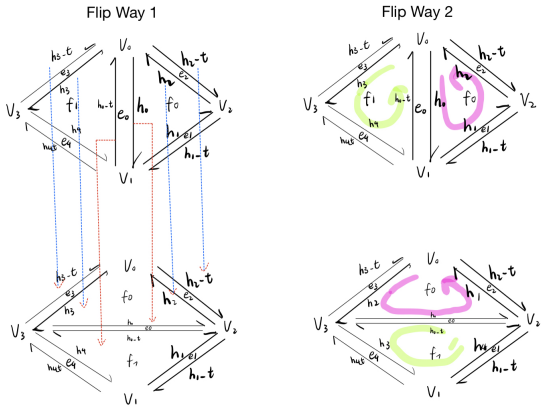


Figure 20: Two ways I tried to flip the edge

4.1 The problem I met

Because the edge reprensentation is too messy, so it is easy to forget one vertex or one edge. I had a hard time looking back again and again. The only way to deal with it is to be careful and careful.

Some errors were like this:

Figure 21: The bug that I met during the splitting

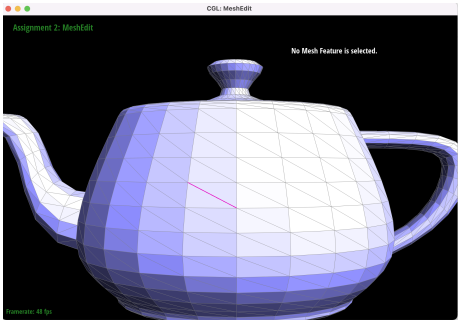


Figure 22: Teapot before flipping

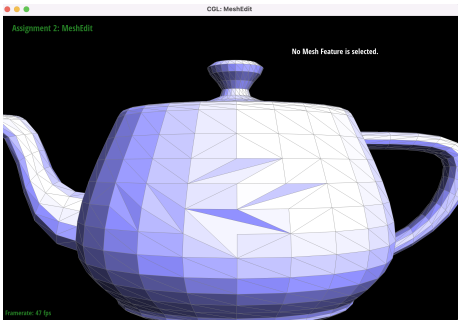


Figure 23: Teapot after flipping

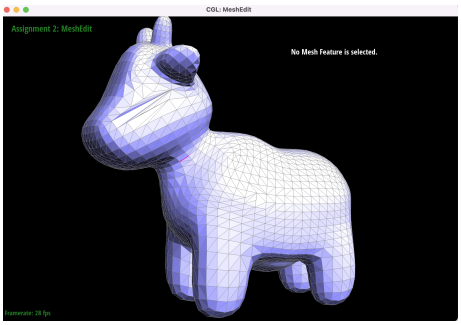


Figure 24: Cow after flipping

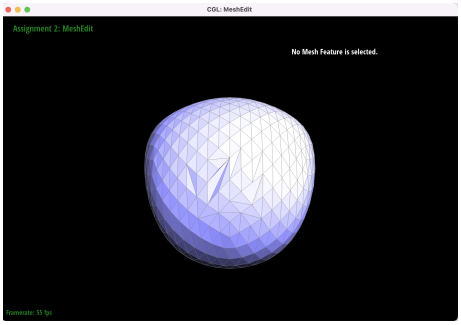


Figure 25: Bean after flipping.

4.2 Task 4 Results

5 Part 5: Half-edge split

Half-edge splitting is a method that essentially creates four triangles from two adjacent triangles by inserting a new vertex in the midpoint of the shared edge, and connecting the opposite vertices of the two triangles through that vertex. Splitting half-edges is a challenge when needing to keep track of pointers and variables.

Just as Task 4, I started by keeping track of all of the possible elements (half-edges, vertices, edges, and faces) of the two triangles. Then, I allocated memory for 1 vertex, 3 edges, 2 faces, and 6 half-edges. The new vertex was just the middle distance between the two endpoints of edge e0 (which was the given parameter). I reassigned pointers for many of the elements. My splitting arrangement is as follows:

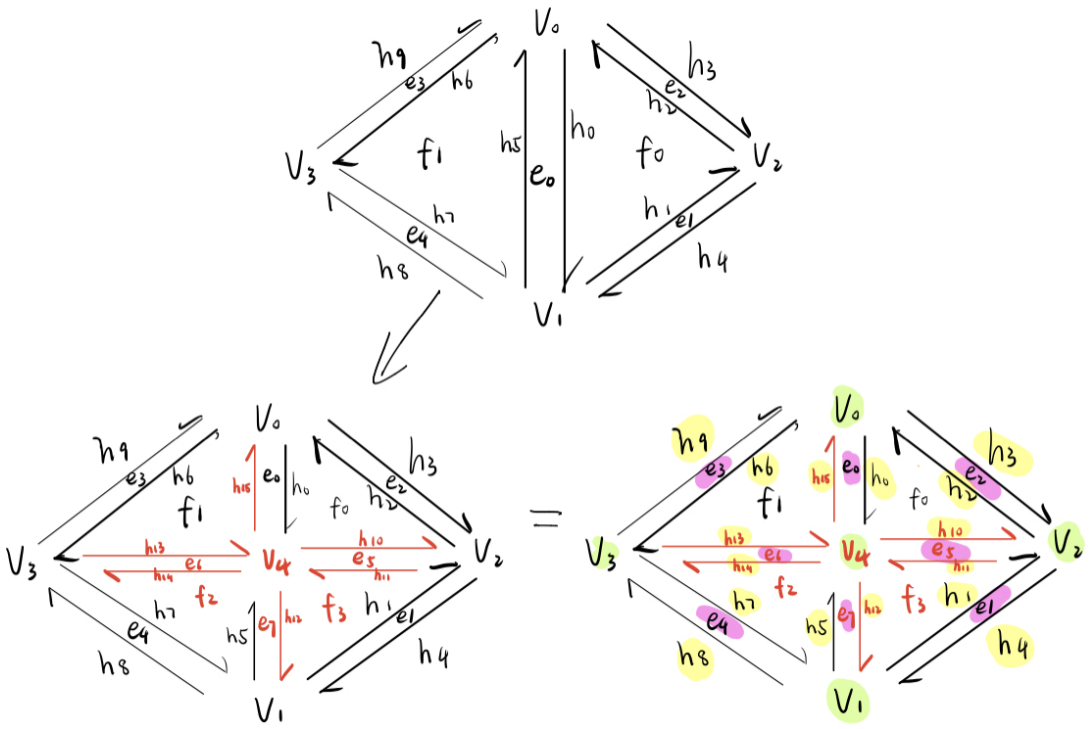


Figure 26: The element representation when I tried to split the edge

Luckily, I didn't get too much bugs this time because I learned the lesson from the previous task 4. So I draw the graph representation clearly and be very careful when I implement the coding.

5.1 Task 5 Results

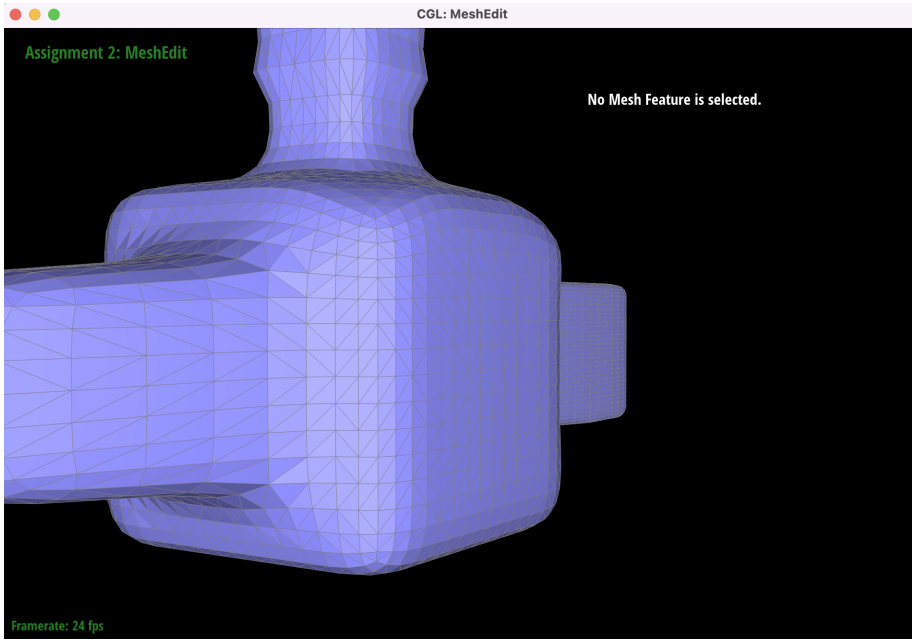


Figure 27: Weird before splitting

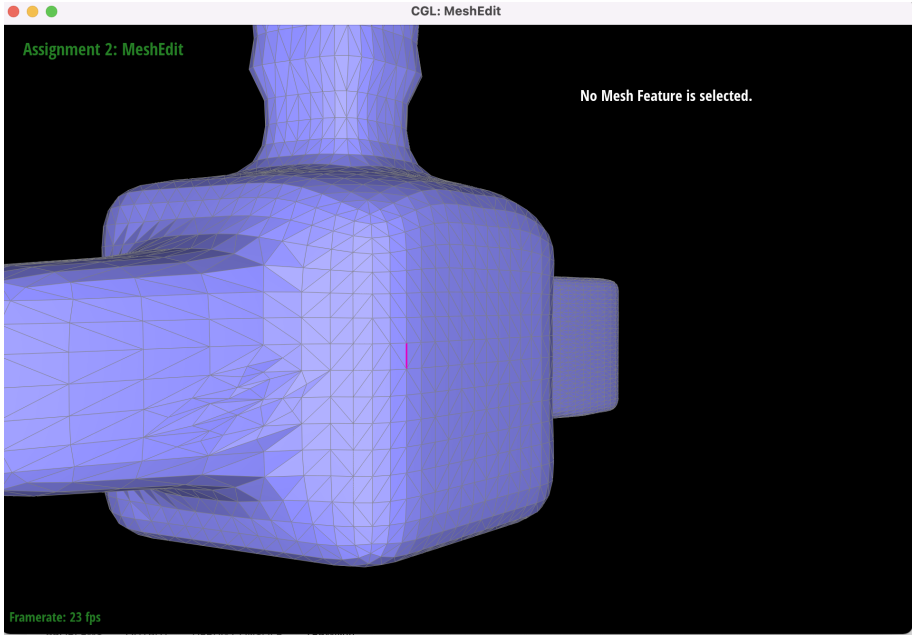


Figure 28: Weird After spliting

6 Task 6: Loop Subdivision for Mesh Upsampling

This routine should increase the number of triangles in the mesh using Loop subdivision.

1. Compute new positions for all the vertices in the input mesh, using the Loop subdivision rule, and store them in Vertex::newPosition. At this point, we also want to mark each vertex as

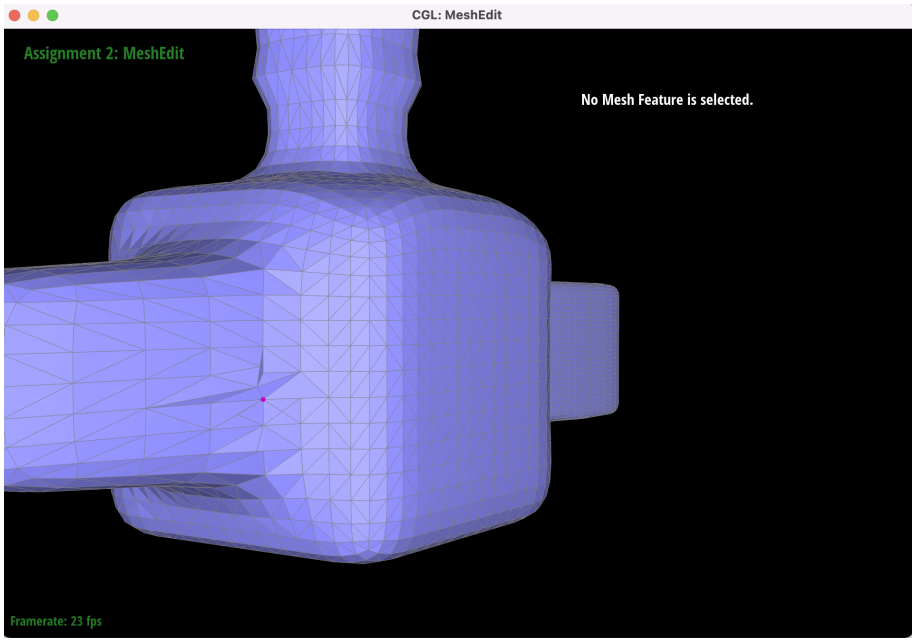


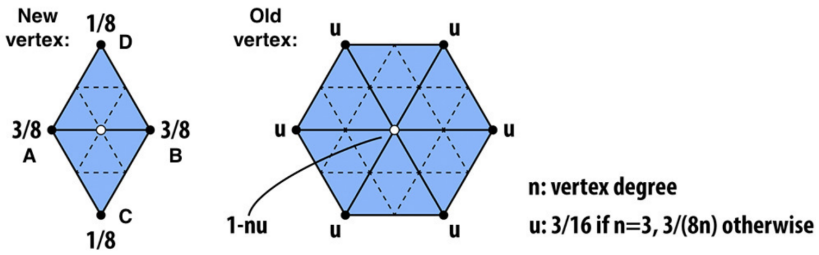
Figure 29: Weird After spliting and flipping

being a vertex of the original mesh.

```
1  // ITERATE AROUND ALL THE VERTEX AND FIND ITS NEW POSITION
2  for( VertexIter vertx = mesh.verticesBegin(); vertx != mesh.verticesEnd(); vertx++ )
3  {
4    // DO SOMETHING INTERESTING WITH V
5    vertx->isNew = false;
6    int degree_count = 0;
7    Vector3D original_neighbor_position_sum = Vector3D(0,0,0);
8    HalfedgeIter h_start = vertx->halfedge();
9    HalfedgeIter h = vertx->halfedge();
10   do{
11     HalfedgeIter h_twin = h->twin();
12     VertexIter vertex_u = h_twin->vertex();
13     original_neighbor_position_sum += vertex_u->position;
14     degree_count++;
15     h = h_twin->next();
16   }while(h != h_start);
17   float u;
18   if (vertx->degree() == 3){
19     u = 3/16;
20   }else{
21     u = 3/(8*vertx->degree());
22   }
23   vertx->newPosition = (1 - vertx->degree() * u) * vertx->position + u *
         original_neighbor_position_sum;
24   vertx->isNew = false;
```

```
25 }
```

2. Compute the updated vertex positions associated with edges, and store it in Edge::newPosition.
3. Split every edge in the mesh, in any order. For future reference, we're also going to store some

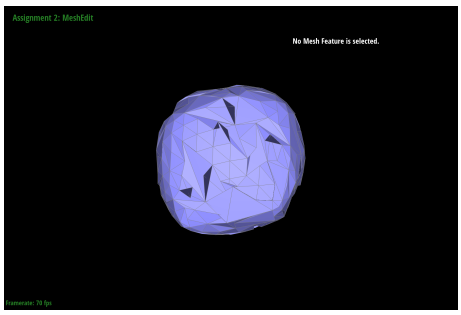


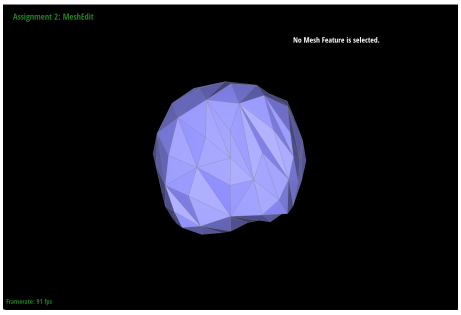
information about which subdivide edges come from splitting an edge in the original mesh, and which edges are new, by setting the flat Edge::isNew. Note that in this loop, we only want to iterate over edges of the original mesh—otherwise, we'll end up splitting edges that we just split (and the loop will never end!)

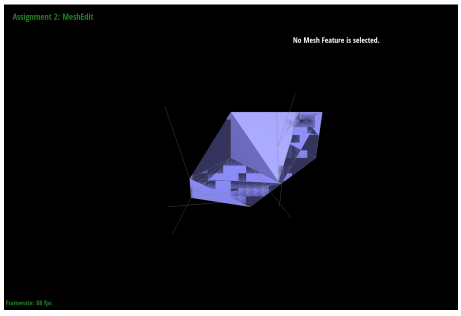
4. Flip any new edge that connects an old and new vertex. 5. Copy the new vertex positions into final Vertex::position.

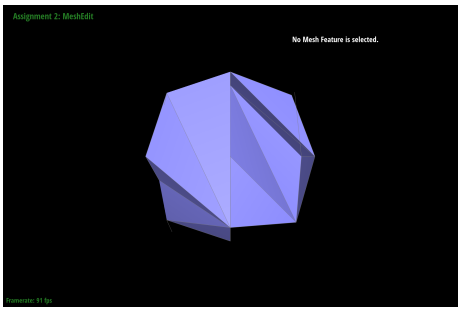
6.1 Problem that I met

If we did not update the edge "isNew" information, or if we failed to store the newposition to the buffer ahead of time, we will have the following failures when doing upsampling.



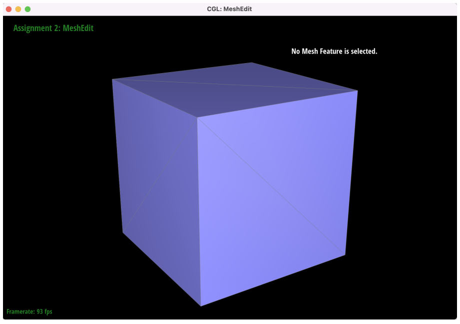


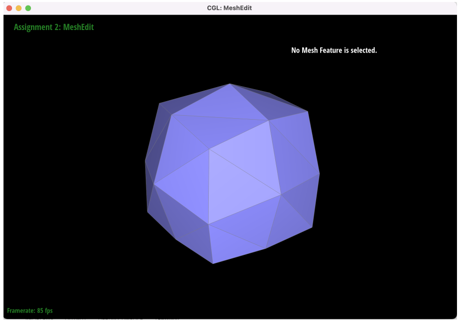


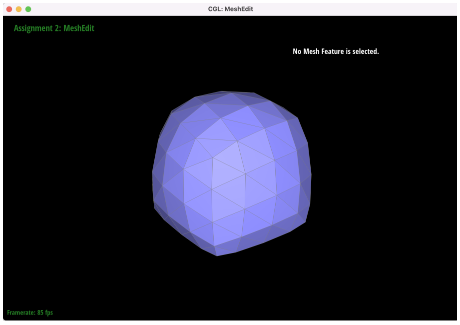


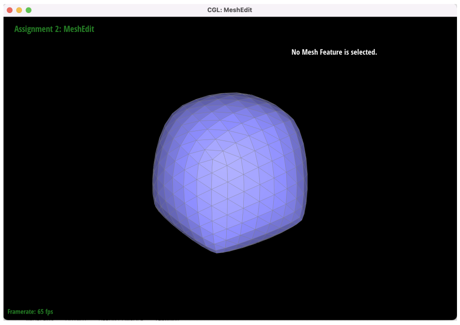
7 Results

1. The results of upsampling of cubes:





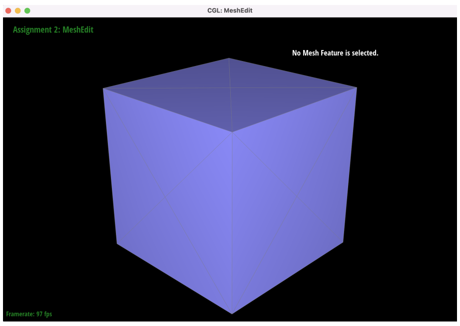


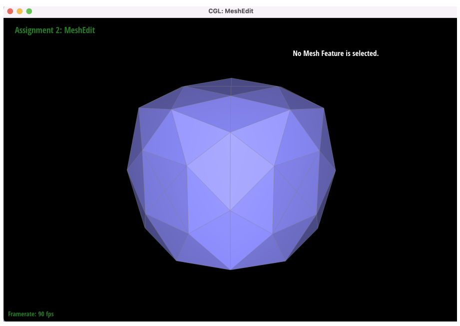


As we can see that sharp edges and corners get smoothed out. and provide a more continuous mesh.

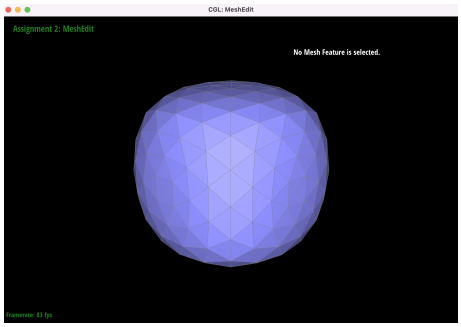
If we want to pre-process some edges, then we can reduce the amount being smoothed. This makes sharp edges less smooth, or in other words, they are sharper.

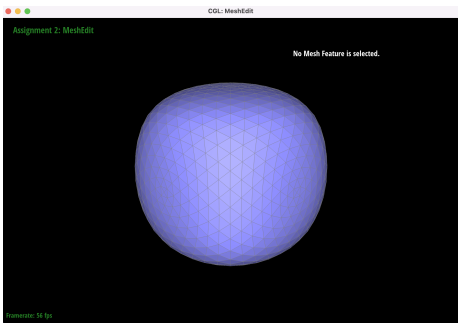
2. The cube becomes asymmetric after repeated subdivision steps because the original mesh had asymmetric edges. One way to change it is to set each conors with even number's degree, by pre processing as follows:





Therefore, if we can keep the vertexes with even numbers, then the split could be symmetric for symmetric objects.


CGL: MeshEdit
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 83 fps


CGL: MeshEdit
Assignment 2: MeshEdit
No Mesh Feature is selected.
Framerate: 56 fps