CSC4140 Assignment V

Computer Graphics

April 5, 2022

Geometry

This assignment is 9%(with 2 extra credit) of the total mark.

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

Student ID: 118010141

Student Name: Jingyu Li

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

Signature: JINGYU LI

Contents

1 Overview

This assignment aims to realize geometry operations on graphics and finish a series of tasks. Geometry is a branch of mathematics that studies the sizes, shapes, positions angles and dimensions of things. Geometry is vital in computer graphics field since it decides the object's orientation and structures. For example, triangle is the primitive element in modeling. A beautiful and complex model can be represented by a huge number of triangles, namely triangle meshes.

Starting from drawing a bezier curve with de Casteljau algorithm, we move on to extend 2-dimension case to 3-dimension and graph the bezier surfaces with similar idea. In the following part, we weight the vertex normals with face area, replacing flat shading to phong shading and making the meshes smooth. We then implement two fundamental edge operations: flip and split, with many tricky pointer assignment. Finally, we make use the flip and split function implemented to upsample the whole mesh using loop subdivision, increasing the resolution as well as the accuracy. This is the road-map of this assignment.

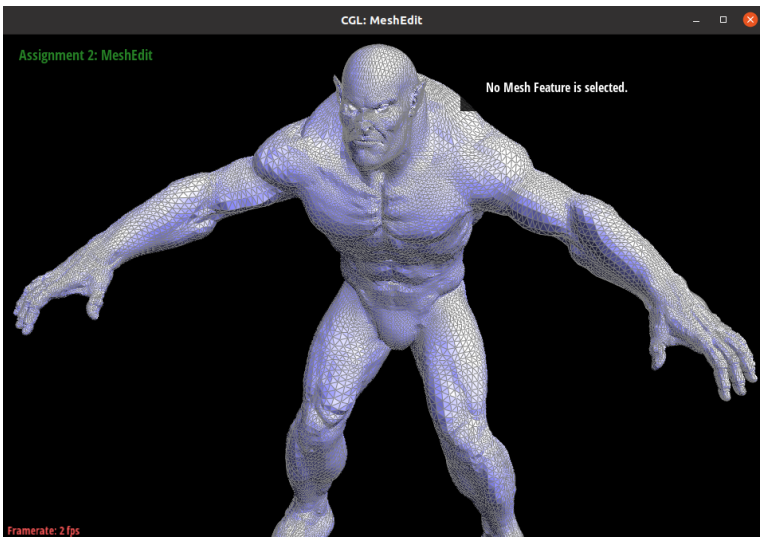


Figure 1: Beast in mesh

2 Task 1: Bezier curves with 1D de Casteljau subdivision

2.1 Design and Implementation

Task 1 requires us to implement **de Casteljau** subdivision algorithm to draw bezier curves. We would like to model smooth and infinitely scalable curves or surfaces in an easy way. De casteljau algorithm uses a parametric method to represent curve. To model curve with degree n, we require (n+1) control points and a parameter t. We recursively perform the following: compute the intermediate points of the current control points using linear interpolation (similar with **lerp** in linear method in pixel sampling). Reduce the point number by 1 each time until we have only a final single point, and that is the curve position at time t. Here t is the ratio in lerp and the range is [0, 1].

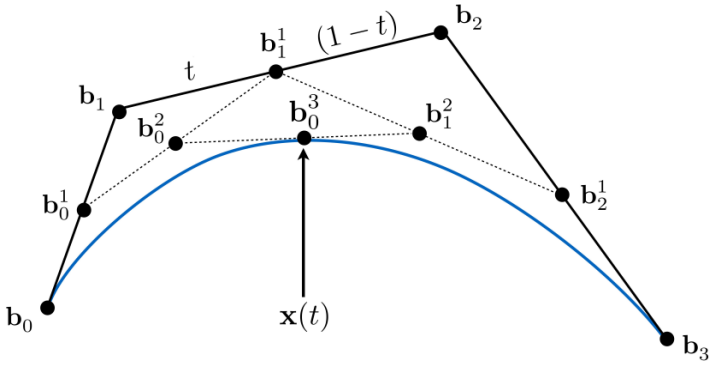


Figure 2: De Casteljau algorithm

$$p_i' = lerp(p_i, p_{i+1}, t) = (1-t)p_i + tp_{i+1}$$

Figure 3: Lerp

The code logic is simple: first encapsulate a lerp function which receives two Vector2D point and the parameter t and output the linear interpolation of the new point. In evaluate_step() function, we only perform one step/level, so given the control points, we just iterate through the vector and pick two points at a time, get the member variable t, use lerp(), and push the new point into another vector, return the new vector.

```
Vector2D lerp2D(Vector2D p1, Vector2D p2, float t) {
  return t * p1 + (1 - t) * p2;
}
```

Figure 4: lerp2D

```cpp
std::vector<Vector2D> BezierCurve::evaluateStep(std::vector<Vector2D> const &points)
{
  // TODO Task 1.
  std::vector<Vector2D> intermediate_points;
  for (int i = 0; i < points.size() - 1; i++) {
    intermediate_points.push_back(lerp2D(points[i], points[i+1], this->t));
  }

  return intermediate_points;
}
```

Figure 5: evaluateStep

2.2 Results and Analysis

Here shows the results of generated bezier curve with 6 control points. The file is in **../bzc/my-curve.bzc**, and the point coordinate are: (0.200, 0.350) (0.300, 0.600) (0.500, 0.750) (0.700, 0.450) (1.000, 0.900) (1.200, 0.200). By clicking **E** we can obtain each intermediate state. By clicking **C** we can obtain the complete curve. The following also shows the modified curve by dragging the 4^{th} original control point downwards and the curve with different parameter t by scrolling.

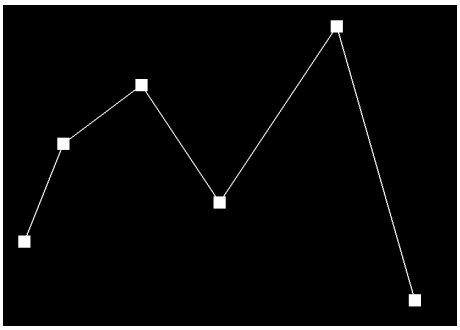


Figure 6: Curve1

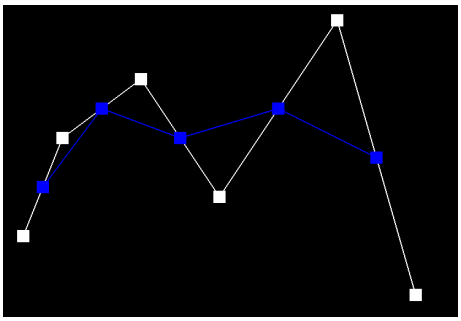


Figure 7: Curve2

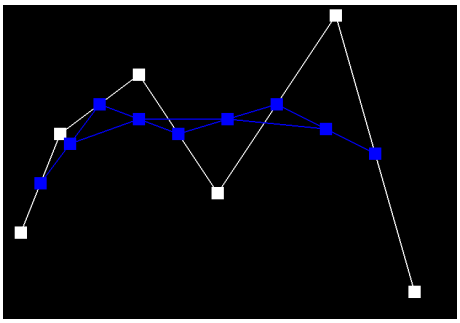


Figure 8: Curve3

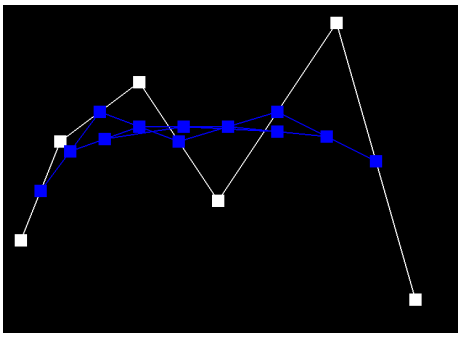


Figure 9: Curve4

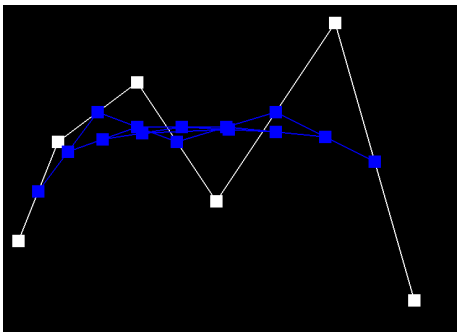


Figure 10: Curve5

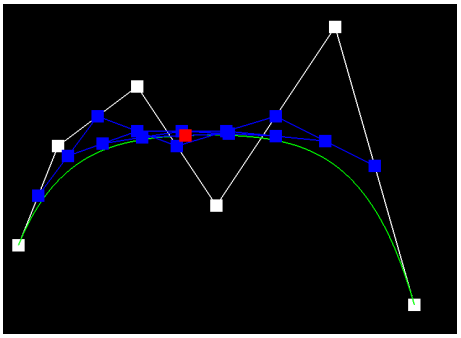


Figure 11: Curve_complete

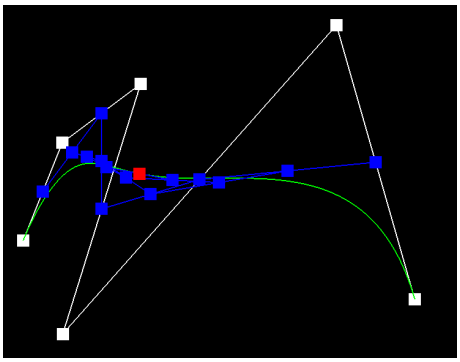


Figure 12: Curve_move

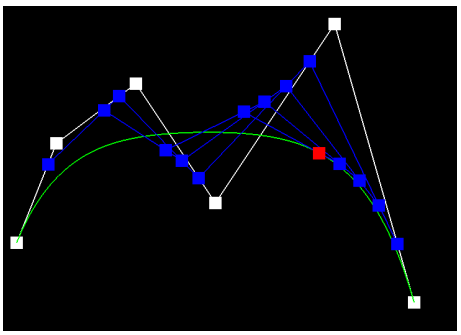


Figure 13: Curve_t

3 Task 2: Bezier surfaces with separable 1D de Casteljau

3.1 Design and Implementation

Task 2 requires us to implement de Casteljau algorithm on 2-dimension surface. This time the point coordinate is in 3D space. To generate a bezier surface with n * m degree, we require (n + 1) * (m + 1) control points, together with two parameters u, and v to determine the linear interpolation ratio, respectively. The algorithm extension is simple. We first consider one dimension, calculate the intermediate points row by row with parameter u and generate a series of final points. Then treat all these points as a row (column) and compute the single final point with parameter v, then this point is the position of the surface with (u, v). Here u, v locate in range [0, 1].

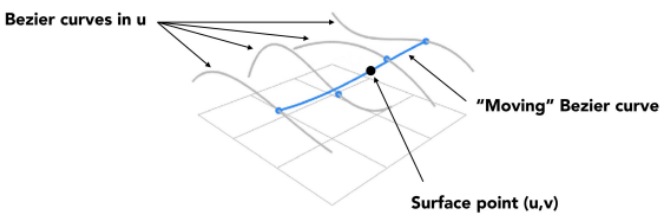


Figure 14: Bezier Surface

The code logic is as follows: First define a 3D lerp() function exactly the same as 2D case

but with Vector3D as the type. Also define evaluateStep() function similarly with task 1. In evaluate1D function, perform evaluateStep() on a row vector and return the final single point. The function evaluate() is the encapsulation of the previous two, given a 2-D points vector, perform evaluateStep() row by row and obtain a 1-D vector, then perform evaluateStep() on this vector to get the final point with parameters (u, v). Here shows the code screenshot of the last two functions.

```
Vector3D BezierPatch::evaluate1D(std::vector<Vector3D> const &points, double t) const
{
  // TODO Task 2.
  std::vector<Vector3D> intermediate_points = this->evaluateStep(points, t);
  while (intermediate_points.size() > 1) {
    intermediate_points = this->evaluateStep(intermediate_points, t);
  }
  return intermediate_points[0];
}
```

Figure 15: evaluate1D

```
Vector3D BezierPatch::evaluate(double u, double v) const
{
  // TODO Task 2.
  std::vector<Vector3D> row_points;
  Vector3D final;
  for (int i = 0; i < this->controlPoints.size(); i++) {
    row_points.push_back(this->evaluate1D(this->controlPoints[i], u));
  }
  final = this->evaluate1D(row_points, v);
  return final;
}
```

Figure 16: Evaluate

3.2 Results and Analysis

Here shows the screenshot of the file **teapot.bez** evaluated by the bezier surface implementation.

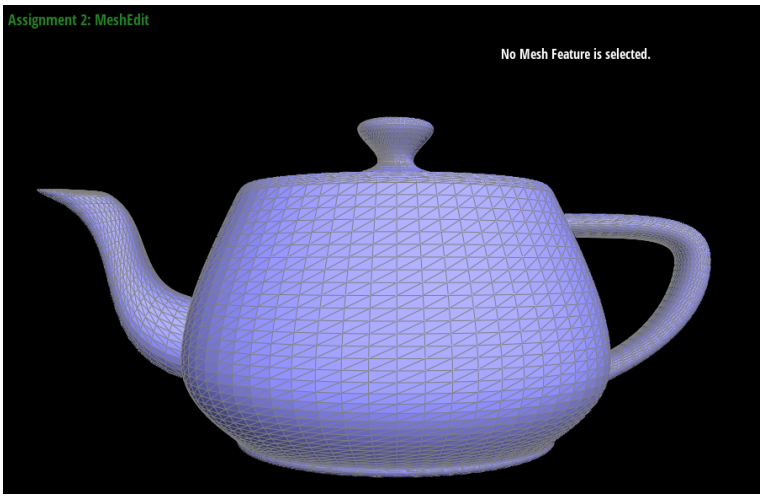


Figure 17: Teapot

4 Task 3: Area-weighted vertex normals

Start from this task, we try to make use of the half-edge data structure. This data structure store the connectivity information in a mesh instead of the absolute data value of the elements like vertex position or edge index. By operating on the iterator (pointers) provided by half-edge, from my understanding, a directly edge, we can access other related elements around. This data structure is artful, simple, flexible and provides great convenience when iterating through the complex triangle mesh.

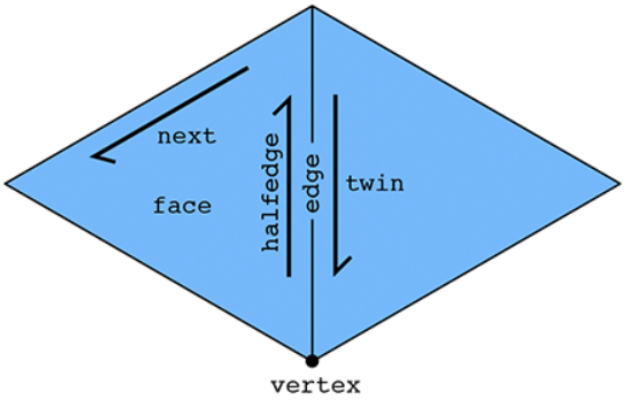


Figure 18: Halfedge

4.1 Design and Implementation

Task 3 requires us to obtain an approximate unit normal at the specific vertex, computed by taking the area-weighted average of the normals of neighboring triangles, then normalizing. The following shows the code implementation: Given the vertex, first obtain its corresponding half edge, define the weighted average Vector3D normal, find the three vertex position in this face (perform next() to get next half edge and get the vertex). Compute two vertices' difference to get two edge vectors and perform cross product, add the cross product directly to the normal as the sum. The reason why the computation is correct is that the direction of the cross product is perpendicular to the face, and the value is proportional to the area, so adding the cross product directly gives out the area weighted average normal direction and value. Iterate next face incident to the current vertex by calling halfedge's twin and next, until we come to the original halfedge. Normalize the normal sum in the end. Notice that here we use the constant iterator since we do not modify the member function. It provides data security.

```
Vector3D Vertex::normal( void ) const
{
  HalfedgeCIter h = halfedge();
  Vector3D N(0.0, 0.0, 0.0);

  do {
    Vector3D pA = h->vertex()->position;
    Vector3D pB = h->next()->vertex()->position;
    Vector3D pC = h->next()->next()->vertex()->position;
    Vector3D cross_product = cross(pB - pA, pC - pB);
    N += cross_product;

    h = h->twin()->next();
  } while (h != halfedge());

  return N.unit();
}
```

Figure 19: Normal

4.2 Results and Analysis

Here shows the **teapot.dae** and **cow.dae**. Use key **Q** to toggle between flat shading and phong shading and use **W** to toggle wireframe. As can be seen from the graph, before averaging the normal, one can clearly see the triangle/polygon corners in the mesh. But with phong shading, the surface becomes quite smooth and looks comfortable now.

Figure 20: Flat teapot

Figure 21: Phong teapot

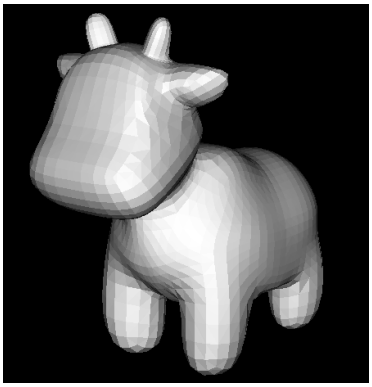


Figure 22: Flat cow

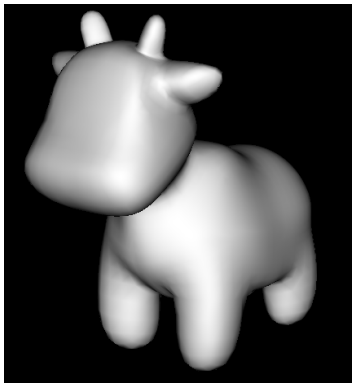


Figure 23: Phong cow

5 Task 4: Edge flip

5.1 Design and Implementation

Task 4 aims to implement edge flip. Edge flip is one of the local re meshing operations. See we have two triangles **abd** and **bcd**, by performing edge flip on edge **bc**, we flip it to the edge **ad** and the polygon now is divided into triangle **acd** and **abd**. Note that flip neither adds new elements nor deletes elements, we only need to transform the elements into others.

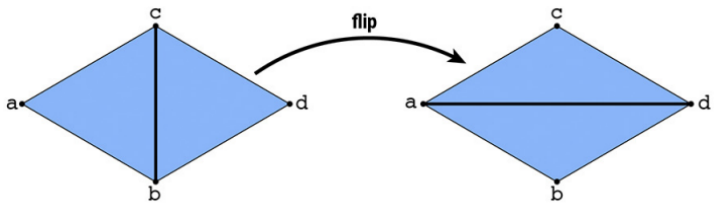


Figure 24: Edge flip

The code logic is shown as follows: To improve readability and succinctness, I define all the elements in the mesh element, polygon **abcd**: Halfedge, Vertex, Edge, and Face. Notice here we use mutable iterator to represent the element. Write down all the iterators according to the half edge data structure relationship. Pay attention to unite the orientation, here I set it as counter clockwise. Given the edge e0, we can first write:

$$HalfedgeIter \quad bc = e0 -> halfedge();$$

After defining halfedge bc, we can define other halfedge using the twin and next operator. For example:

$$HalfedgeIter cb = bc -> twin();$$

We have 10 halfedges in the original mesh in total, since one edge counts two halfedges. We also have two transformed halfedges **ad** and **da** in the new polygon, and we initialize their values as halfedge **bc** and **cb**.

For vertex, edge and face, they only have one private variable, the relevant halfedge. Because we can access these elements with one halfedge and the next() and twin() operator. As a result, we just use relevant halfedge to initialize these elements, notice here we should still use edge bc to initialize edge ad, and so do the faces. In the end the original edge bc will become ad and no modification on the mesh will happen. Here shows some example:

$$VertexIter \quad a = ab- > vertex();$$

$$EdgeIter \quad Eca = ca- > edge(); EdgeIter \quad Ead = Ebc;$$

$$FaceIter \quad Fadc = Fabc;$$

After defining all the elements, I set all the pointers even if they do not change, to avoid possible mistakes. For vertex, edge and face, I assign the iterator of halfedge to their corresponding halfedge (E.g. Vertex a -> Halfedge ad), and for element in the original mesh but not in the new mesh, use new elements to cover them (E.g. Edge bc -> Edge ad). Finally, the most important part, set all the halfedges, use the convenient setNeighbor() function, remember the counter-clockwise direction. The tricky part here is that for external halfedge (like ac), its next() and corresponding face does not lie in the polygon shown but in another polygon, so we should keep the value as its original value. Also, the halfedge bc and cd should be set at last since they transform to ad and da in the end. Finally return the flipped edge **ad**. Here shows the halfedge assignment. (And, the eventful debugging journey will appear in Section 8)

```
ca->setNeighbors(ad, ac, c, Eca, Fadc);
ac->setNeighbors(ac->next(), ca, a, Eca, ac->face());
ab->setNeighbors(bd, ba, a, Eab, Fabd);
ba->setNeighbors(ba->next(), ab, b, Eab, ba->face());
bd->setNeighbors(da, db, b, Ebd, Fabd);
db->setNeighbors(db->next(), bd, d, Ebd, db->face());
dc->setNeighbors(ca, cd, d, Edc, Fadc);
cd->setNeighbors(cd->next(), dc, c, Edc, cd->face());
ad->setNeighbors(dc, da, a, Ead, Fadc);
da->setNeighbors(ab, ad, d, Ead, Fabd);
bc = ad;
cb = da;
```

Figure 25: Flip code

5.2 Results and Analysis

Here shows the screenshot of edge flip. Select an edge and press **F** to flip. It looks like eliminate one diagonal and connect the other diagonal.

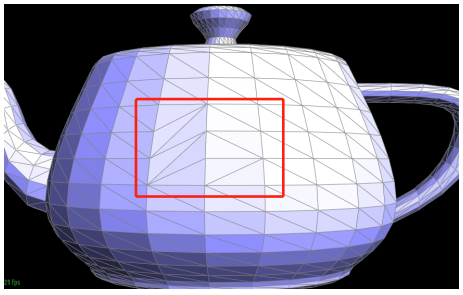


Figure 26: Teapot flip

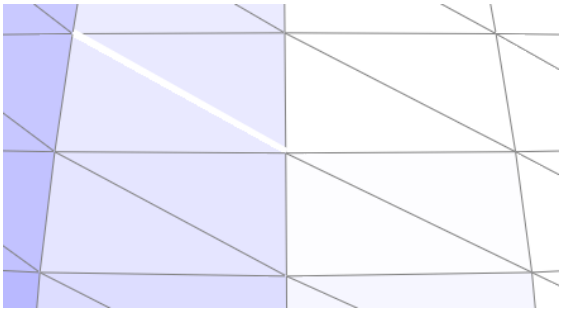


Figure 27: Before Flip

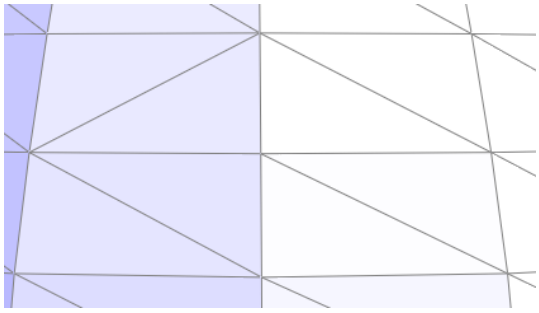


Figure 28: After Flip

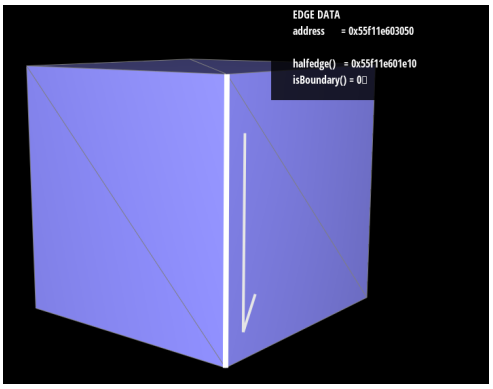


Figure 29: Cube Flip 1

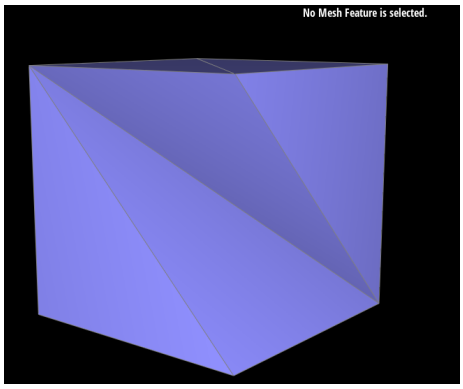


Figure 30: Cube Flip 2

6 Task 5: Edge split

6.1 Design and Implementation

Task 5 aims to implement edge split. Edge split is one of the local re meshing operations. See we have two triangles **abd** and **bcd**, by performing edge split on edge **bc**, we get four new edges and the polygon now is divided into four triangle **amc**, **mdc**, **abm** and **mbd**. Note that split is more complex than flip since we need to create new element and delete some old ones, which may cause bugs easily.

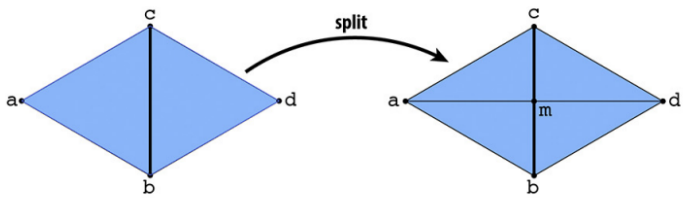


Figure 31: split

The code logic is similar with edge flip, but requires more consideration on construction and

destruction of elements. Define all elements in the polygon, this time we have 18 halfedges, 5 vertices, 9 edges, and 6 faces in total, counting both original and new situations. For original mesh elements, initialize as task 4, and for new mesh elements, use **newHalfedge()**, **newVertex()**, and so on to create a new empty element. The value assignment is also quite similar, but we need to set some member variable like **isNew** to mark whether the element is newly added. Also, we need to set vertex m's position as the midpoint of vertex b and vertex c.

```
a->halfedge() = am; // ab
b->halfedge() = bm;
c->halfedge() = cm;
d->halfedge() = dm; // dc
m->halfedge() = mc;
m->isNew = true;
m->position = (b->position + c->position) / 2;
m->newPosition = e0->newPosition;
```

Figure 32: Split code 1

```
// Ebc = Emc;
Eca->halfedge() = ca;
Eab->halfedge() = ab;
Ebd->halfedge() = bd;
Edc->halfedge() = dc;
Emc->halfedge() = mc;
Ebm->halfedge() = bm;
Eam->halfedge() = am;
Emd->halfedge() = md;
Eam->isNew = true;
Emd->isNew = true;
Ebm->isNew = true;
Emc->isNew = true;
```

Figure 33: Split code 2

Assign vertex, edge, face with their corresponding halfedge, and assign halfedge using setNeighbor(), and pay attention to the relationship. Moreover, we need to delete the extra elements in the end, like Edge bc, Face abc, Halfedge bc, and so on, or they will still be stored in the mesh and rendered out, causing unexpected errors. Finally, return vertex m.

The debugging trick and eventful debugging experience will be shown in section 8.

```
ca->setNeighbors(am, ac, c, Eca, Famc);
ac->setNeighbors(ac->next(), ca, a, Eca, ac->face());
ab->setNeighbors(bm, ba, a, Eab, Fabm);
ba->setNeighbors(ba->next(), ab, b, Eab, ba->face());
bd->setNeighbors(dm, db, b, Ebd, Fmbd);
db->setNeighbors(db->next(), bd, d, Ebd, db->face());
dc->setNeighbors(cm, cd, d, Edc, Fmdc);
cd->setNeighbors(cd->next(), dc, c, Edc, cd->face());
mc->setNeighbors(ca, cm, m, Emc, Famc);
cm->setNeighbors(md, mc, c, Emc, Fmdc);
bm->setNeighbors(ma, mb, b, Ebm, Fabm);
mb->setNeighbors(bd, bm, m, Ebm, Fmbd);
am->setNeighbors(mc, ma, a, Eam, Famc);
ma->setNeighbors(ab, am, m, Eam, Fabm);
md->setNeighbors(dc, dm, m, Emd, Fmdc);
dm->setNeighbors(mb, md, d, Emd, Fmbd);

deleteEdge(Ebc);
deleteFace(Fabc); deleteFace(Fcbd);
deleteHalfedge(bc); deleteHalfedge(cb);
```

Figure 34: Split code

6.2 Results and Analysis

Here shows the screenshot of edge split and some combinations of flip and split. Select an edge and press **S** to split. It looks like add another diagonal.

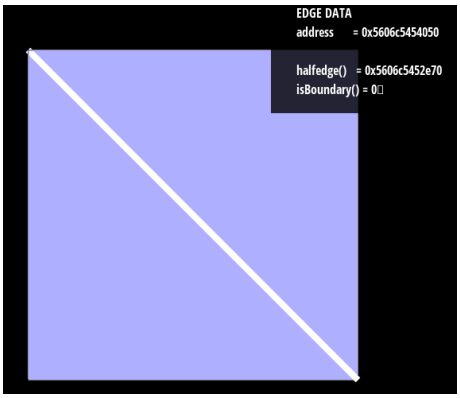


Figure 35: Cube Split 1

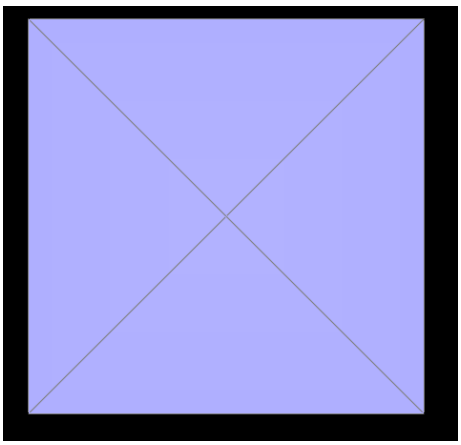


Figure 36: Cube split 2

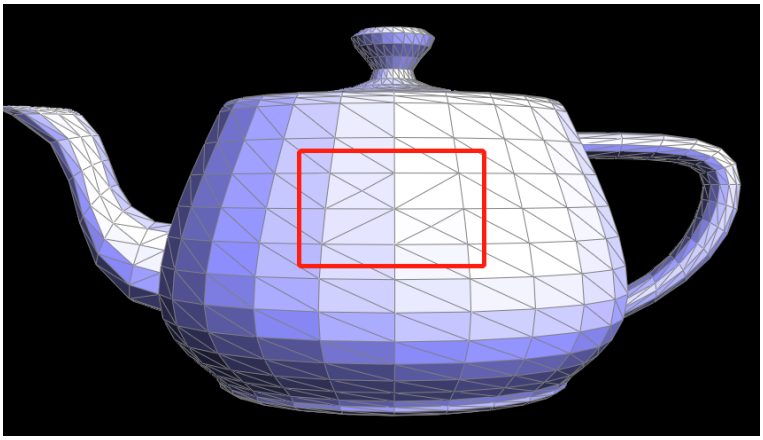


Figure 37: Teapot split 1

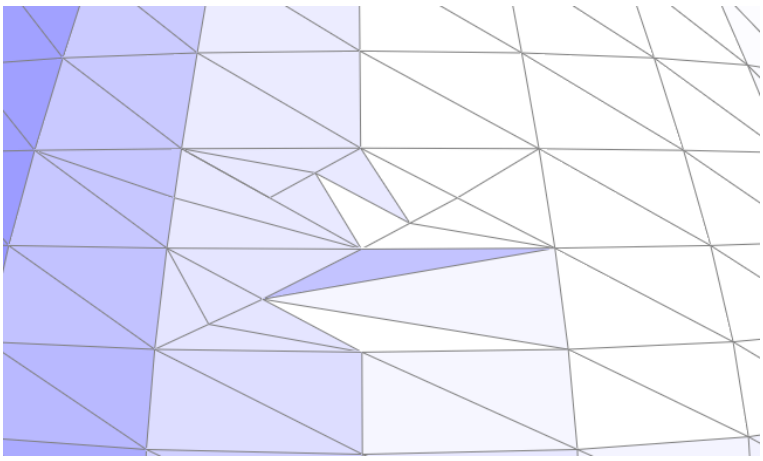


Figure 38: Teapot split 2

7 Task 6: Loop subdivision for mesh upsampling

7.1 Design and Implementation

Task 6 aims to implement loop subdivision algorithm to upsample the mesh, adding new smaller triangles. This operation converts a coarse polygon mesh into a higher resolution one for better display and more accurate simulation. Notice here we first update the position and then perform split flip. We should not flip the new edges along the boundary, we should flip the blue edges connecting old and new vertices but not the black one.

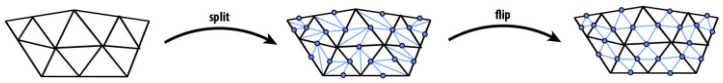
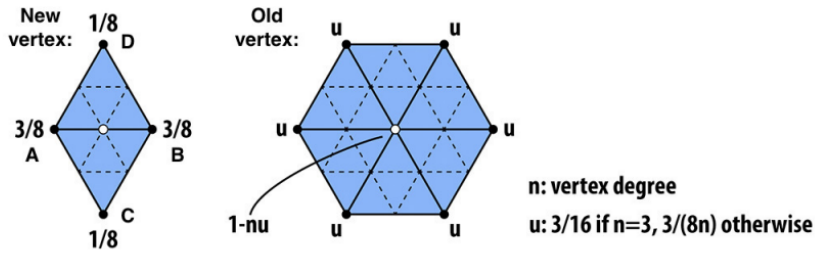


Figure 39: Upsample

The code logic is as follows, I follow the comments in the code:

(1) Compute new positions for all the vertices in the input mesh, using the Loop subdivision rule, shown as below:



(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 40: position

Store the new position in Vertex::newPosition. At this point, we also want to mark each vertex as being a vertex of the original mesh. Iterate all vertices, to get the incident vertices, use the halfedge->next()->twin() trick.

(2) Compute the updated vertex positions associated with edges, and store it in Edge::newPosition. Use the same rule as above and use halfedge to access four neighboring vertices.

(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!). Notice that vertex iteration in this part is very tricky. Split an edge means delete the original edge, and we cannot perform edge++ after splitting, so we should access the next edge in advance. Also, we need to store the edges to be flipped in the next step in another vector to recognize different types' edges.

```
vector<EdgeIter> flip_edges;

EdgeIter e = mesh.edgesBegin();
EdgeIter next = e;
while (e != mesh.edgesEnd()) {
  next = e;
  next++;
  if (!e->isNew) {
    VertexIter m = mesh.splitEdge(e);
    flip_edges.push_back(m->halfedge()->next()->next()->edge());
    flip_edges.push_back(m->halfedge()->twin()->next()->edge());
  }
  e = next;
}
```

Figure 41: Loop split

(4) Flip any new edge that connects an old and new vertex. Make use of the vector we create in the previous step, judge the two vertices of the edge, whether they are one new and one old (can use xor operator).

```
for (EdgeIter e : flip_edges) {
  if (e->isNew) {
    HalfedgeIter h = e->halfedge();
    VertexIter start = h->vertex();
    VertexIter end = h->twin()->vertex();
    if (start->isNew ^ end->isNew) {
      mesh.flipEdge(e);
    }
  }
}
```

Figure 42: Loop flip

(5) Copy the new vertex positions into final Vertex::position. Update the isNew state to false for all vertices and edges.

7.2 Results and Analysis

Here shows the loop subdivision results. For the teapot, the loop subdivision performs pretty successfully. One triangle splits into four new triangles and arranges smoothly in the mesh. However, for the cube, the corners and edge seem sharp. There are hump at the corner side and cause the slight asymmetry. As for the upsample level increases, the surface also becomes more and more smooth. But we can still see the corner extrude faintly. This is interesting observation.

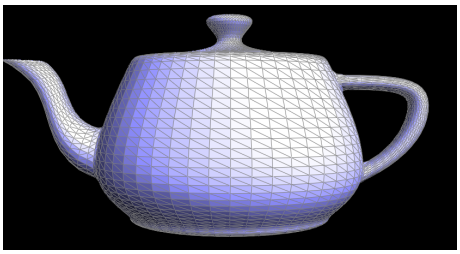


Figure 43: Teapot upsample 1

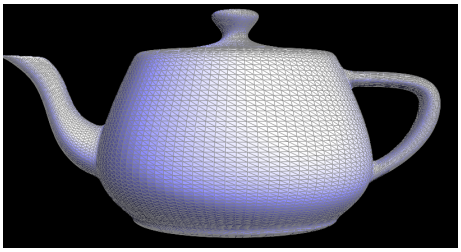


Figure 44: Teapot upsample 2

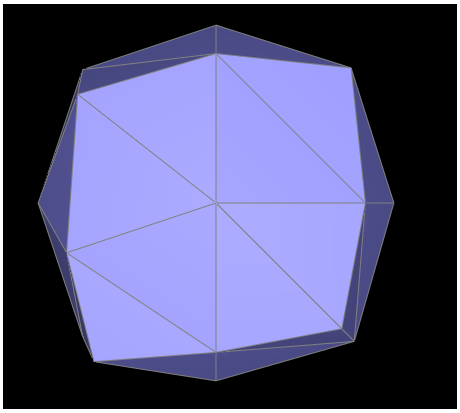


Figure 45: Cube upsample 1

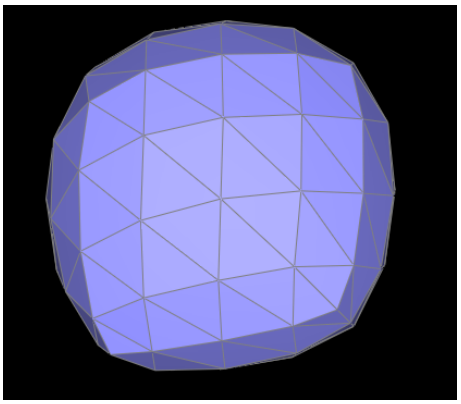


Figure 46: Cube upsample 2

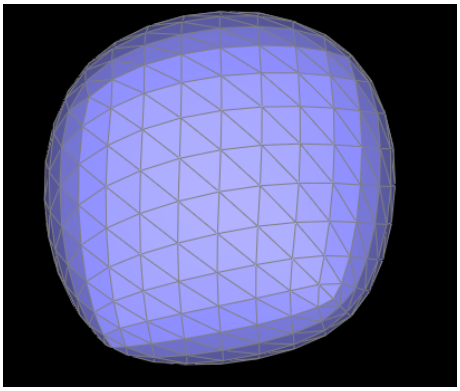


Figure 47: Cube upsample 3

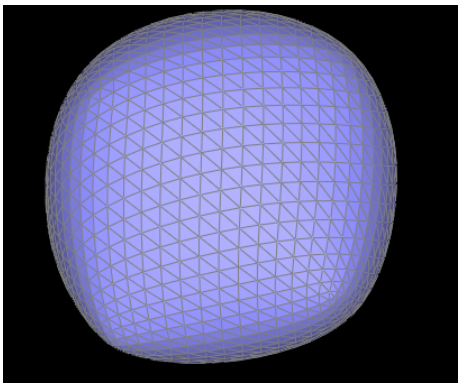


Figure 48: Cube upsample 4

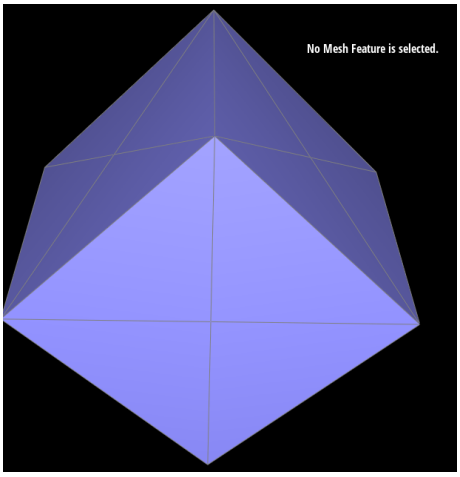


Figure 49: New Cube upsample 1

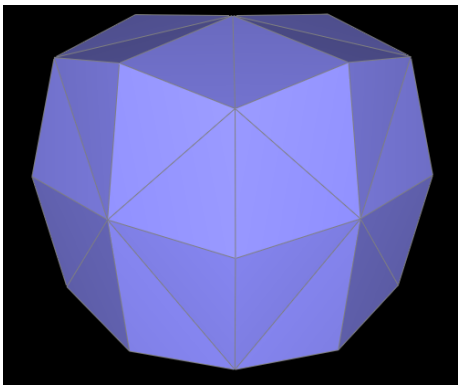


Figure 50: New Cube upsample 2

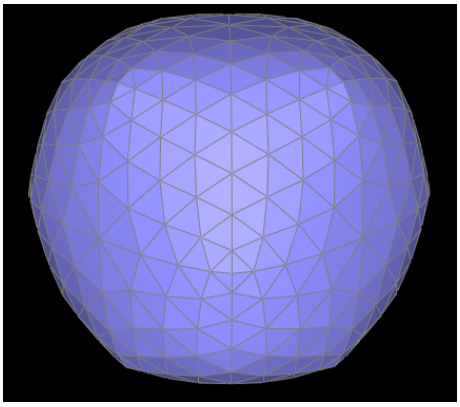


Figure 51: New Cube upsample 3

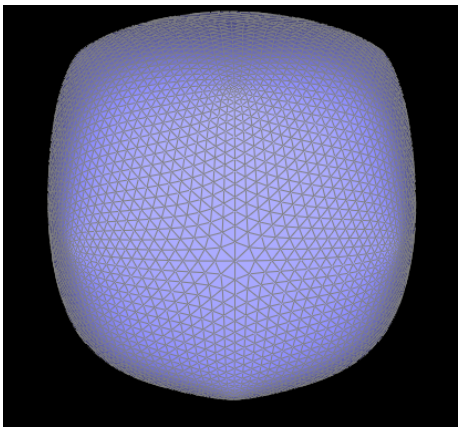


Figure 52: New Cube upsample 4

As can be seen from above, I reduce this effect by pre-splitting the other diagonal edges and make the original cube symmetric. At the beginning, there is only one diagonal connection in one face, and this will cause the asymmetry. After the processing the cube splits and flips symmetrically and the effect is alleviated.

8 Problems and Solutions

In this section, I will share some problems/bugs/eventful debugging experience when finishing this assignment.

(a) In task 2, I returned vector[0], which was a single double element, actually I should not add the operator [] but return the point vector.

(b) In task 4, at first I set the halfedge values in the form bc->next()->next() ... without defining all elements, causing the great mess, actually it'd better defining edges/vertices/faces clearly in advance.

(c) In task 4, at first I use reference to represent elements. E.g. Edge cb = bc->twin(), but this will cause chain effect and brings weird bugs.

(d) In task 5, I forgot to set m's new position as the edge's new position (updated position of the new vertex stored in edge momentarily).

(e) In task 6, at first I used for loop to iterate the edges and perform split, however, it's wrong since splitting will delete the edge and we cannot access the next edge once we delete. So we should use while loop and use another variable next to store next edge in advance and then split edge e.

(f) In task 6, I have written the code **next** = **e**++, and this will add the current edge e too!!! The correct way should be: **next** = **e; next**++

(g) In task 6, I forgot to judge the edge condition: one new vertex and one old vertex.

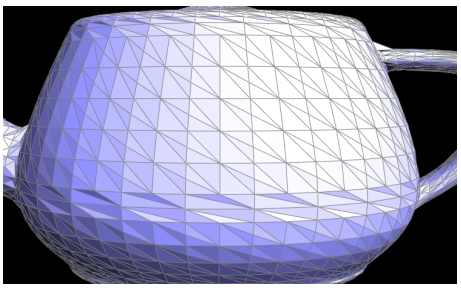


Figure 53: Wrong implementation 1

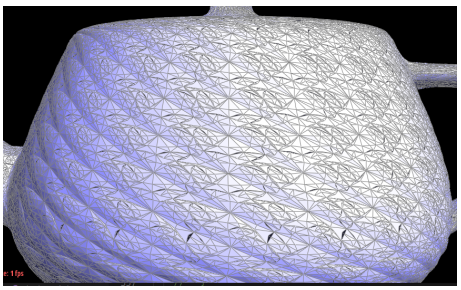


Figure 54: Wrong implementation 2

9 Execution

(a) Run **sh ./compile_run.sh** to compile and run

(b) After build, use the following command to render:

$$./meshedit \quad image\_path$$

(c) Supported image type: .bzc .bez .dae

(d) GUI usage on Bezier part: E->Perform one call to evaluate one step. Can perform repeatedly. C->Toggle full Bezier curve. Can click and drage control points and check the changes. Can scroll to adjust the value of t and see the walking path of the intermediate control points.

(e) GUI usage on mesh: Click mesh element to select. Drag the change the camera position. Scroll to zoom the image. Q->Toggle with vertex normals. F->Flip. S->Split. L->Upsample the mesh. N->next half-edge. T->twin half-edge. W->Toggle wireframe.

(f) Many results are not shown but the graph can be generated properly.

(g) Read the code for more details

10 Summary

Here I will summarize what I have learned when writing this assignment.

(a) Understanding of Geometry in computer graphics. For example, how to draw bezier curve/-surface, how to use half edge, how to perform edge flip split, and so on.

(b) Solve programming problems and have eventful debugging time.

(c) Object-Oriented programming paradigm, use class to encapsulate.

(d) Convenient task: integrate all the codes in one file: student_code.cpp

(e) Meaningful time on debugging and coding skill improvement, write-up report writing.

(f) The ability to use searching engine and meaningful discussion with peers.

(g) Time Scheduling. I should really start early due to Hofstadter's law

(h) A high-quality take-away write-up for future review.

(i) By the way, the document and code comments are detailed and clear, Kudos!

That's all.