CS 184: Computer Graphics and Imaging, Spring 2019

Project 2: Mesh Editor

Shenao Zhang

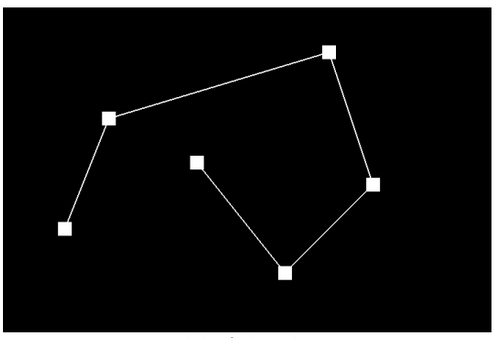
Overview

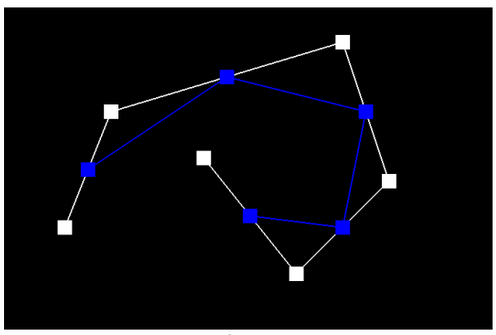
This project is about how to build our 3D model. It mainly contains two parts.
The first part is about how to build Bezier curves and surfaces, which will give us smooth curves and surfaces.
The second part is about looping subdivision of general triangle meshes. To implement that, we first implement a function that calculates the average normals at the vertices which can then be used for more realistic local shading compared to the default flat shading technique. Then we can implement loop subdivision for mesh upsampling by first spliting triangles into small triangles, then flipping the edges which connect an old vertex in the original mesh and new vertex we just created during the spliting step. We implement these two process using function "split" and "flip", which are exactly what we do in part 4 and 5. Spliting is the process that we calculate the middle point of an edge and connect it to the opposite vertices across two faces. Flipping is the process that we convert one edge to another new edge which connects another two vertices. Then we can finally implement loop subdivision.

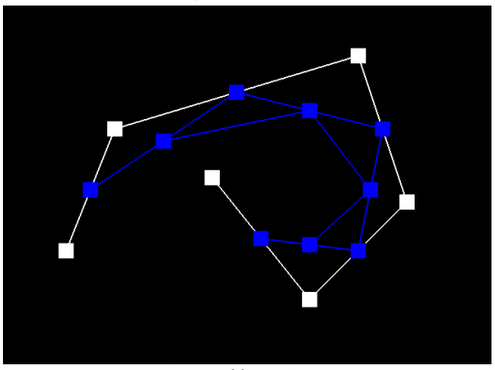
Section I: Bezier Curves and Surfaces

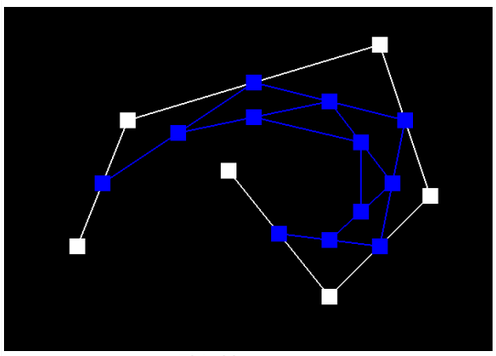
Part 1: Bezier curves with 1D de Casteljau subdivision

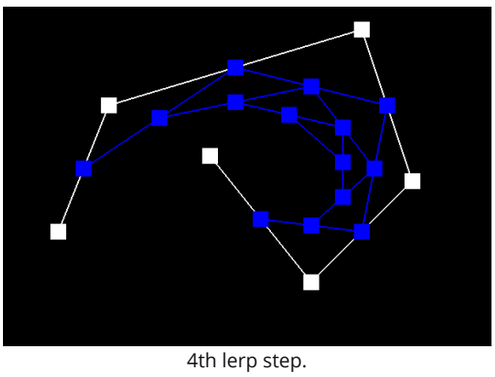
Description
We first evaluate Bezier curves by using de Casteljau's algorithm.De Casteljau's algorithm is a recursive method that we use to evaluate polynomials in Bézier curves. We can then get the smooth curves rather than broken lines.
Method
To implement it, I recursively calculate a new set of intermediate control points through linear interpolation (lerp). Then store the set of points in **newLevel**, pushing it to **evaluatedLevels** so we can pop the **newLevel** points newxt time to calculate another level points.
Example
screenshots of original points of curve3.bzc and lerp steps:

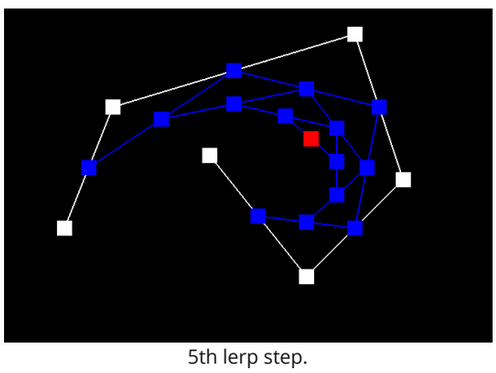

Original six points.

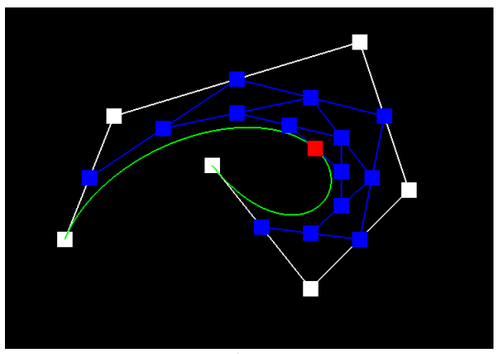

First lerp step.


Second lerp step.


Third lerp step.

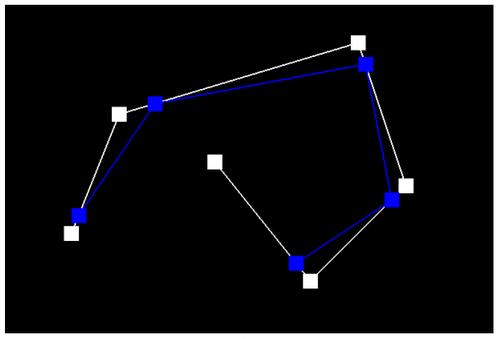

4th lerp step.


5th lerp step.

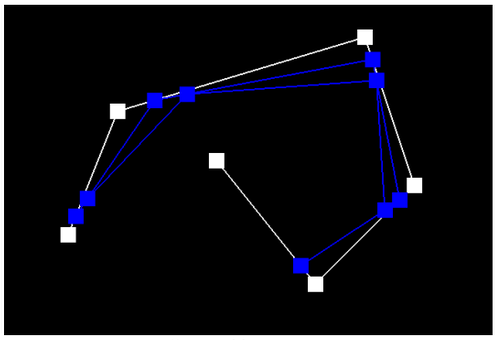


Final curve.

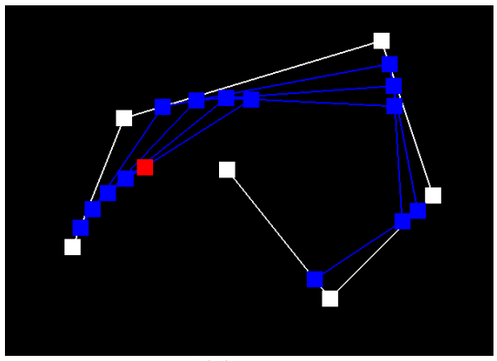
When scroll to make t smaller:



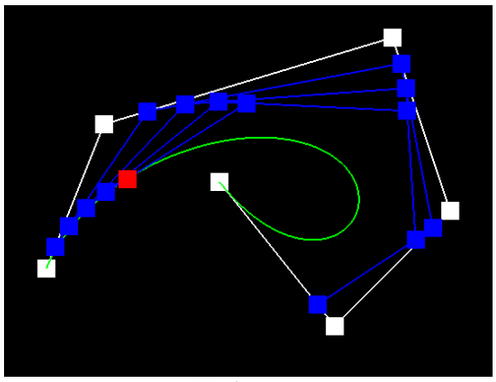
First Lerp when t is small



Second lerp step.

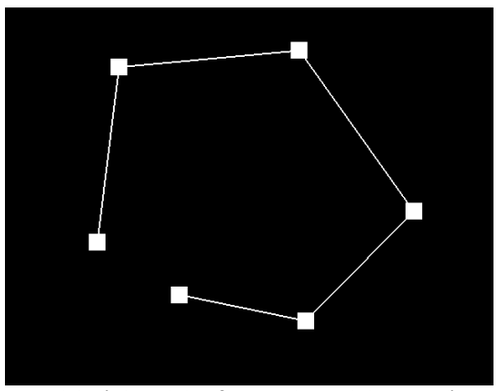


5th lerp step.

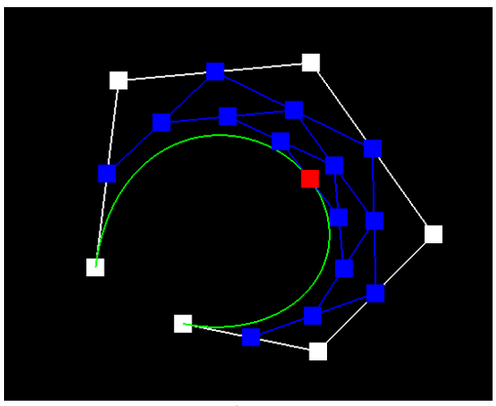


Final curve.

When move some points:



Original six points after moving points around.



Final curve.

Part 2: Bezier surfaces with separable 1D de Casteljau subdivision

Explanation

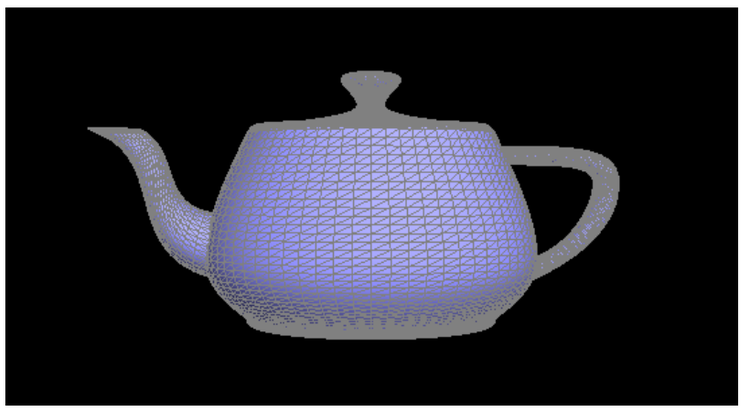
When it comes to surface, we can extend de Casteljau's algorithm to Bezier surfaces. The algorithm can be applied several times to find the corresponding point on a Bézier surface by first calculating intermediate control points of different set of points with the same t. Then we use de Casteljau's algorithm on the control points to get the final surface.

Method

More specifically, I implement function **evaluate1D** first. It can be seen as calling function **evaluateStep**using original points on only one row recursively until we get the point of the final curves. Secondly, I implement function **evaluate**,where **evaluate1D**is called recursively. Every time it is called, we get points of one final curve. So after it is called 4 times, we get a set of points which are on four curves. Then **evaluate1D** is called using this set of points we get, generating the surface.

Example

The rendering of bez/teapot.bez is as below.

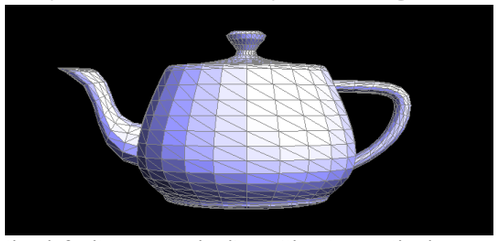


The rendering of bez/teapot.bez

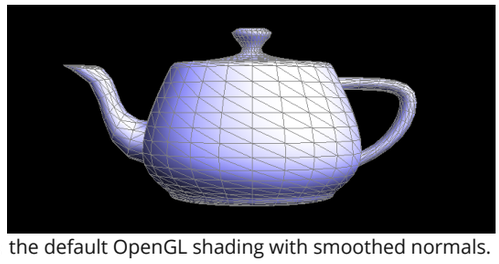
Section II: Sampling

Part 3: Average normals for half-edge meshes

Explanation
Since the default flat shading technique is not so realistic caused by color changing, we need to calculate the area-weighted average normal vector at a vertex, which can then be used for more realistic local shading compared to the default flat shading technique.
To do that, I implement a function **normal** which accumulate area-weighted normal of the faces at one vertex recursively. The main work is to find the corresponding faces recursively and calculate the cross product. Since we represent triangles using data structure "halfedge", we need to represent vertices of faces this way.
For example, **halfedge->next()->vertex()->position** can represent the position of one vertex of a triangle, and **halfedge->twin()->next()** is halfedge in another face.
Example
Below is the comparision of the default OpenGL shading with and without smoothed normals.



the default OpenGL shading without smoothed normals

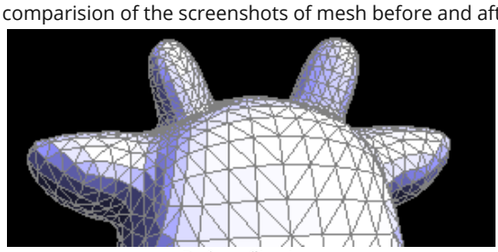


the default OpenGL shading with smoothed normals.

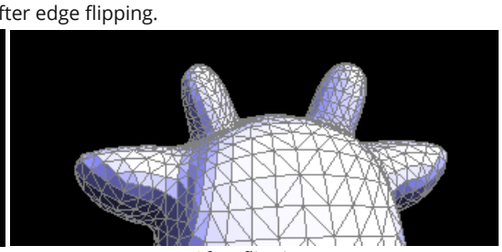
Conculsion
As we can see, because we average the normals of different faces at a given vertex, the default OpenGL shading with smoothed normals have smoother color change compared to the one without smoothed normals.

Part 4: Half-edge flip

Explanation
In this part, we need to flip the halfedge for futher subdivision of general triangle meshes.
Method
To implement **flipEdge** function, I first set the halfedges and vertices, faces of the original triangle according to the given edge. Then to flip the halfedge, we need to reset the relationship between data structures. For halfedges, I set their next halfedge, twin halfedge, vertex, edge, face using function **setNeighbors**. For vertices and faces, I set their halfedges.
Example
Below is the comparision of the screenshots of mesh before and after edge flipping.



Before flipping.

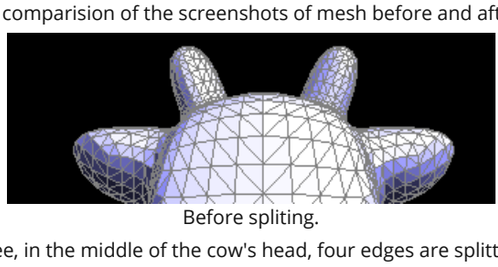


After flipping.

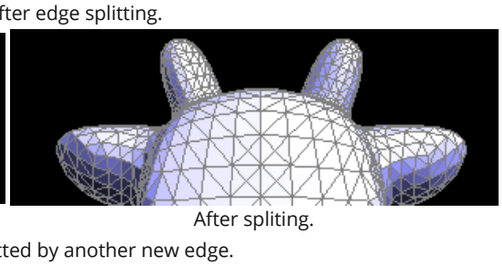
As we can see, in the middle of the cow's head, four edges are flipped.

Part 5: Half-edge split

For this part, to split the halfedge, we need to add more these data structure.
I first draw a picture of the original mesh, set the two faces, four vertices, six halfedges in the original mesh, and then split the original edge. Observing that we need to add six halfedges(each short edge inside the triangle two), two faces, three edges and a mid point((v0->position+v1->position)/2). Then like part 4(flip), I reset all these data structure(both the original and the new) using **setNeighbors** and **halfedge** function.
Example
Below is the comparision of the screenshots of mesh before and after edge splitting.

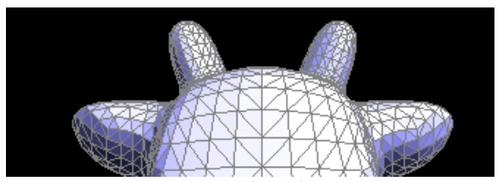


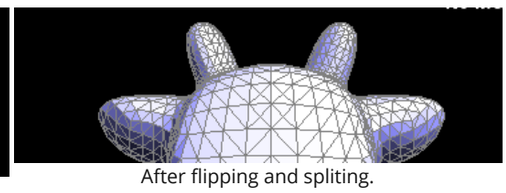
Before spliting.



After spliting.

As we can see, in the middle of the cow's head, four edges are splitted by another new edge.
Below is the comparision of the screenshots of original mesh and after edge splitting and flipping.


Original mesh.


After flipping and spliting.

Conculsion

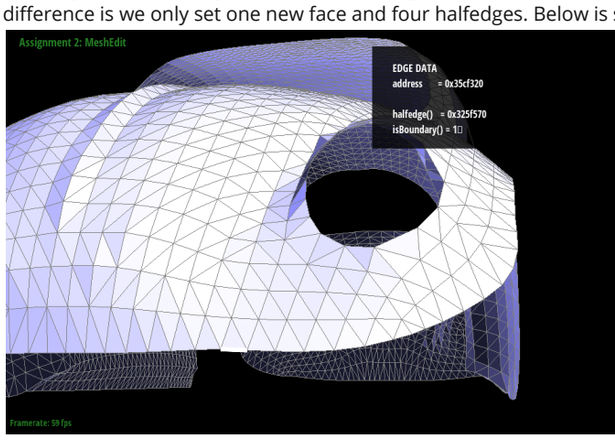
As we can see, in the middle of the cow's head, two edges are flipped, two edges are splitted by another new edge, two edges are first flipped then splitted(equals to splitted by another new edge).

Debugging Quest

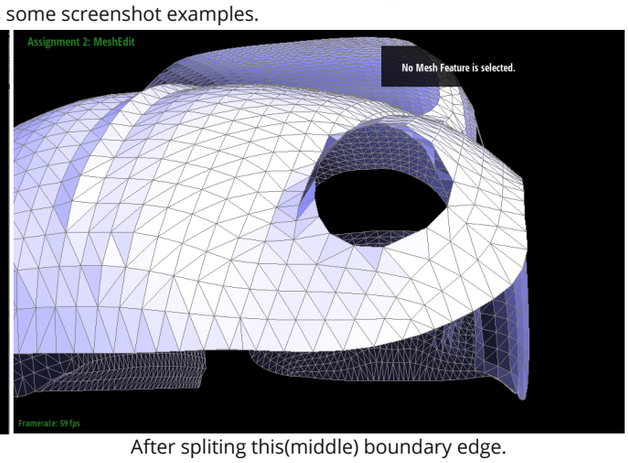
When all the above work is done, the flipping operation works. It seems everything is right. BUt when I went on to part 6, running the subdivision function, it implies that this spliting function lacks the process which set whether the newly defined edges are new or not compared with the original mesh. I will explain that more specifically in part 6.

Extra Credit

To support edge split for boundary edges, we need to split the edge in half but only split the face that is non-boundary into two. The medthod is like non-boundary edges method but excluding any variable which had to do with the second face from the normal split. Then, I got the boundary face and the halfedge of the selected edge that was on the boundary face and reassigned them. And the difference is we only set one new face and four halfedges. Below is some screenshot examples.



Original mesh and select a boundary edge.



After spliting this(middle) boundary edge.

Part 6: Loop subdivision for mesh upsampling

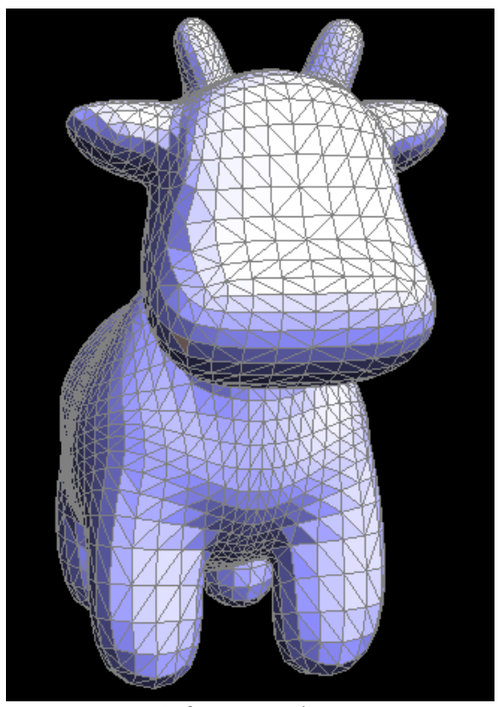
Explanation

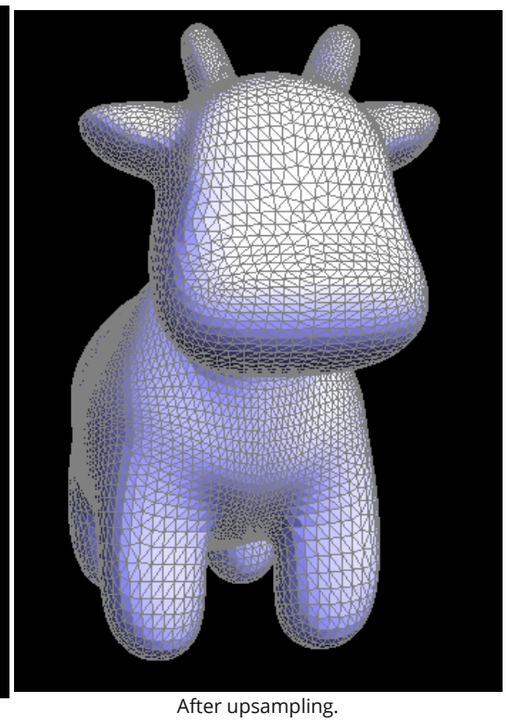
In part 6, we implement loop subdivision for mesh upsampling. To implement it, we need to use the two functions we just mentioned:**flip** and **split**. More specifically, the subdivision function mainly contains six parts.

Method

First, mark the vertices in the original mesh not new by set **isNew** false for futher only change the position of these points. Second, compute updated positions for all vertices in the original mesh using the vertex subdivision rule (1 - n*u) * original_position + u * neighbor_position_sum. Third, compute new positions associated with the vertices that will be inserted at edge midpoints(3/8 * (A + B) + 1/8 * (C + D)). The second and third step both store the position in the temporary storage in case that taking averages of values that have already been averaged. Then use **splitEdge** function which is implemented in part 5 to split every edge in the mesh. Then flip any new edge that connects an old and new vertex using **flipEdge** function which is implemented in part 4. Finally we can copy the new vertex positions into the usual vertex positions(marked in step 1). We get the subdivision for mesh upsampling.

Example


Before upsampling.


After upsampling.

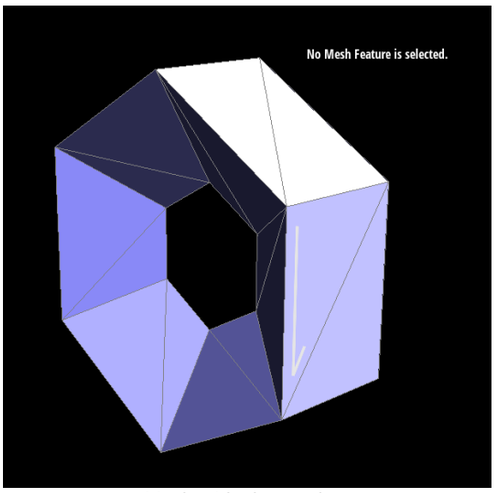
We can see that the original triangles are splitted into small triangles regularly. So the surface is smoother than before. But due to the large computation, the renderization is slower.

Debugging Quest

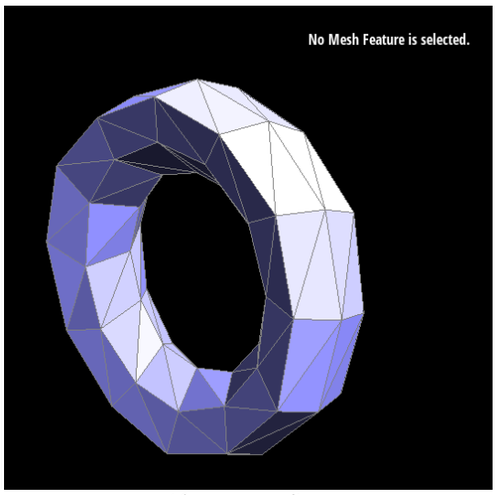
When I first implement the function **upsampling** and subdivision **dae/cow.dae**, it can't stop looping. (And every time it is in endless loop, my Ubuntu crashed. So I spent a lot of time dealing with this bug.) It turns out that when iterate **splitEdges** function, I use while loop, and forget tracing only the edges in the original mesh. So it keeps splitting edges that I just created. After fixing it, it is still not correct. As I mentioned in part 5, I forget to mark the edges **isNew** in **splitEdge** function. It results in wrong subdivision due to the wrong spliting and flipping. After fixxing these two bugs, it works. **Analysis**

When there are sharp corners and edges, as shown below, after upsampling the shape corners/edges get smoother because we use more triangles instead of a single triangle.

Example



Mesh with sharp edge.

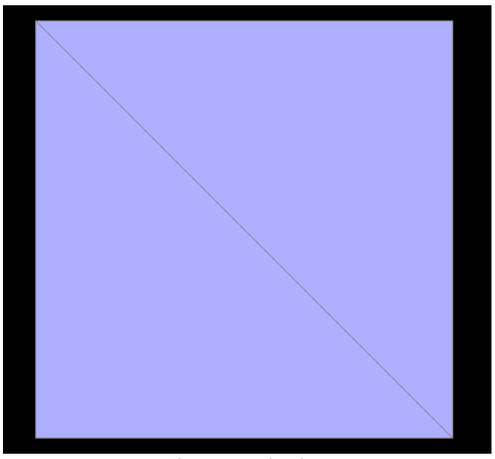


After upsampling.

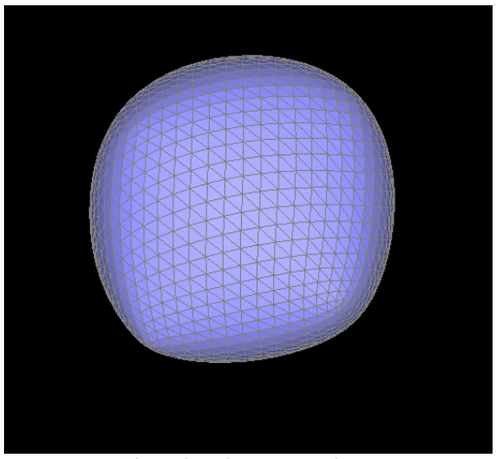
When loading up dae/cube.dae, we get a cube as below. But after upsampling, it becomes asymmetric. As far as I am concerned, it is slightly asymmetric because of uneven distribution of edges in the original mesh.

To deal with it, we can use **splitEdge** to make the distribution of edges even. Then it subdivides symmetrically.

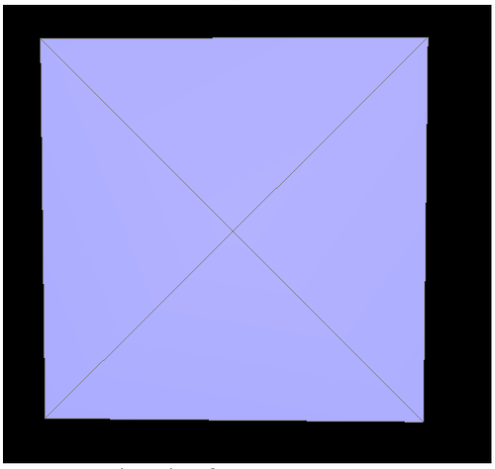
Screenshots



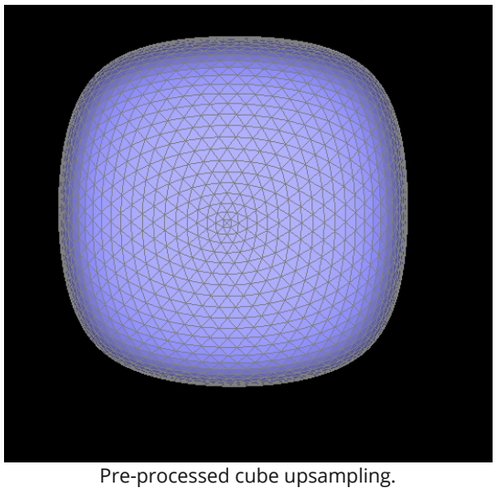
The original cube.



The cube after upsampling.



The cube after pre-processing.



Pre-processed cube upsampling.

Explanation

I split the six edges on six faces of the original cube. Then I get a cube with two edges one face. So after upsampling, we can get regularly distributed triangles. Finally, after I upsample several times, it subdivides symmetrically.

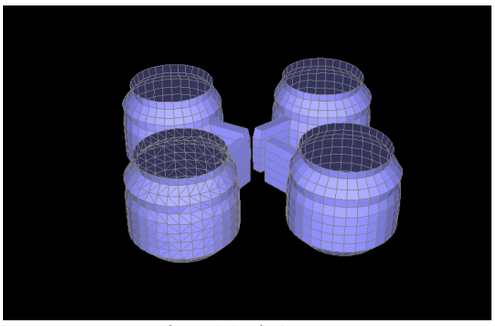
Section III: Mesh Competition

If you are not participating in the optional mesh competition, don't worry about this section!

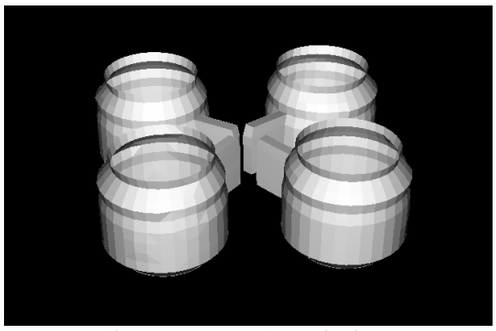
Part 7: Design your own mesh!

I use Blender to create my own model and I find it really hard because it needs so many shortcuts. So I made a simple ejector. And I only apply triangulate modify on one part because when I apply it to the whole model, the render can not show anything. And I don't knoe why.

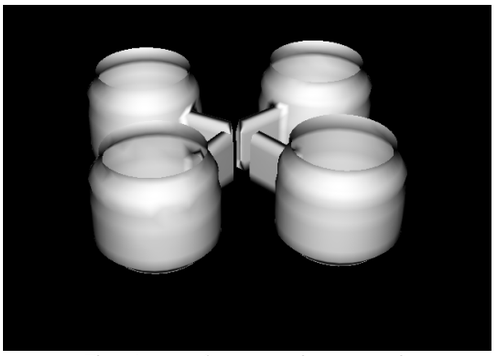
Screenshots



The original ejector.



The ejector using GLSL shaders.



The ejector after normals averaged.