CS 184: Project 2 Mesh Editor

Jieming Fan 3034504370

Overview

Part 1: Ray Generation and Scene Intersection

- In order to generate the rays, first I complete the function `Camera::generate_ray`. The ray formula is `ray = o + t * d`. Where *o* is a vector representing the origin, *t* is a scalar representing the travelling time along direction *d*, and *d* is the direction of the ray. In camera's coordinate system, the camera is positioned at the origin, and its bottom left and top right corners at:

  ```
  Vector3D(-tan(radians(hFov)*.5), -tan(radians(vFov)*.5),-1)
  ```

  ```
  Vector3D( tan(radians(hFov)*.5),  tan(radians(vFov)*.5),-1)
  ```

  Convert the input point to a point on this sensor so that (0,0) maps to the bottom left and (1,1) maps to the top right. The corresponding x, y, z can be calculated by:

  ```
  X = bottom_left.x + (top_right.x - bottom_left.x) * x
  ```

  ```
  Y = bottom_left.y + (top_right.y - bottom_left.y) * y
  ```

  ```
  Z = -1
  ```

  Because the camera looks along the -z axis, so set z be -1. Multuply this by *c2w* to convert it to world space. Then I could get the ray's *r* and *d*.

- I use the Moller Trumbore algorithm for triangle intersection.

- To implement Casteljau's algorithm, I use a `std::vector` type variable *evaluatedLevels* to store all the points I evaluated before. Use `evaluatedLevels.back()` to get the latest evaluated points to calculate new points.

- My Bezier curve with 6 control points

Step 0	Step 1	Step 2
Step 3	Step 4	Step 5
Completed curve	Move the points around	modify the value of t

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

- Casteljau's algorithm could use at least 4 control points to evaluate a Bezier curve. As for the Bezier surfaces, Casteljau's algorithm use at least 16 control points and two parameters *u* and *v*. For each row *i* of the control points and given parameter u, we could use Casteljau's algorithm to get the final point $p_i(u)$. Then there will be a series of points $p_1(u)$, $p_2(u)$, $p_3(u)$... Then we use the Casteljau's algorithm again to use these points and parameter v the get the final point $p(u, v)$. Changing the value of u and v from 0 to 1, the locus of the point $p(u, v)$ is the Bezier surface based the control points.

- To implement Casteljau's algorithm to evaluate Bezier surfaces, I first implement the function `evaluate1D`, which is really similar to what I have down in Part 1, to get the final point based on the given control points and parameter t. Then I use this function for each row i of the control points to get point $p_i(u)$, stored in variable p. At last, use the `evaluate1D` function to get the finial point $p(u, v)$ based on the points set p and parameter v.

- A screenshot of *bez/teapot.bez*. 

Section II: Loop Subdivision of General Triangle Meshes

Part 3: Average normals for half-edge meshes

- In this part, what I have done it to smooth out the teapot by using the average normal vector for a vertex. I implement the function `Vertex::normal` to get the area-weighted average normal vector at a vertex. I first take the cross product of two edges, which are computed by `h -> next() -> vertex() -> position - h -> vertex() -> position`, for each triangle that the vertex is connected to. Then add up all the cross products and compute the unit normal, which is the average normal vector for the vertex.

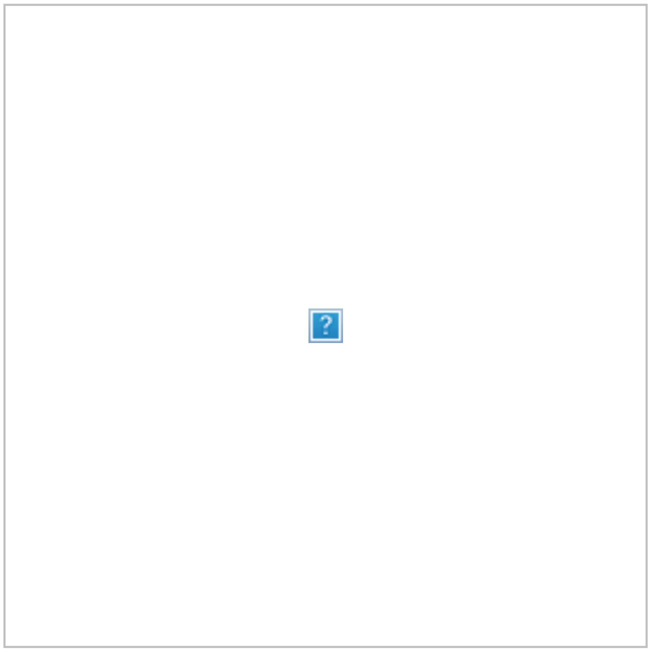
- Screenshots of *dae/teapot.dae*

With smoothed normals	Without smoothed normals

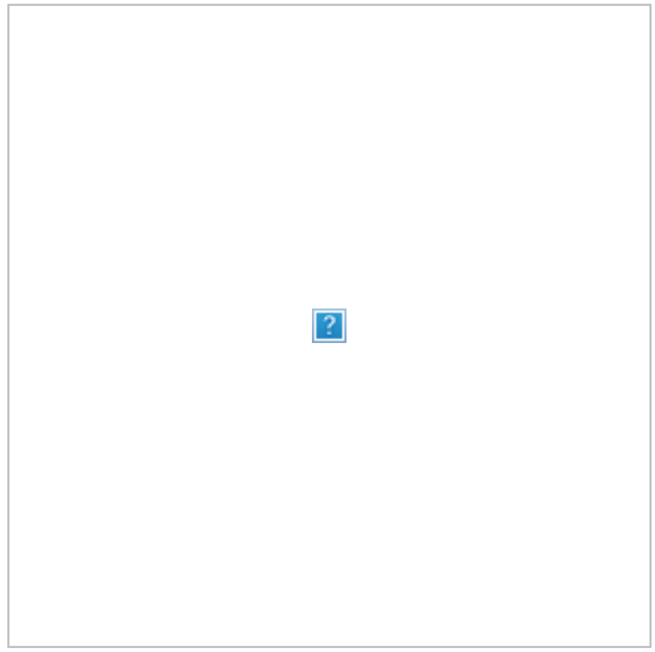
Part 4: Half-edge flip

- To implement the half-edge flip operation, I first draw two pictures about a mesh before flipping and after flipping, and assign all the elements appeared in the original mesh. Then I look through each of the halfedge in the filpped picture to check if it has some elements (vertices, edges, faces) different from it has in original picture. At last, I look at the flipped picture and reassign all the vertices, edges and faces to the halfedge near them. Since the half-edge flip operation won't create or remove any element, these reassignments above are enough.

- Screenshots of *dae/teapot.dae*



Original teapot

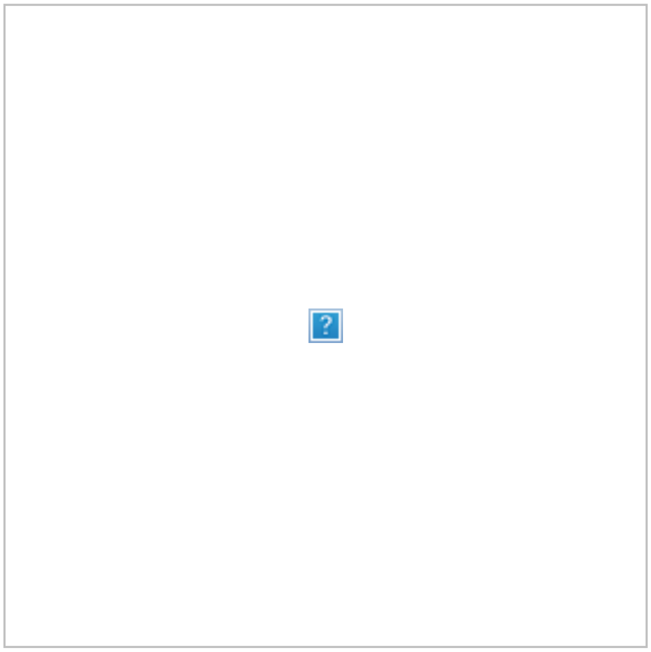
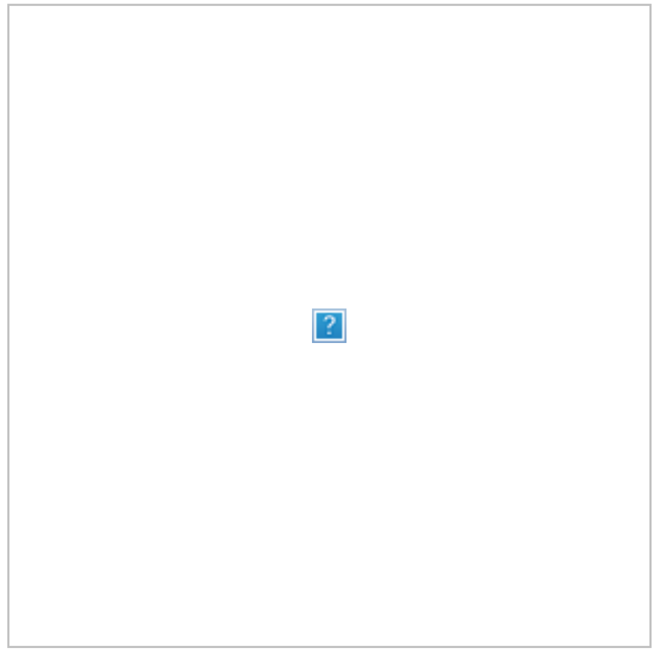


After some edge flips

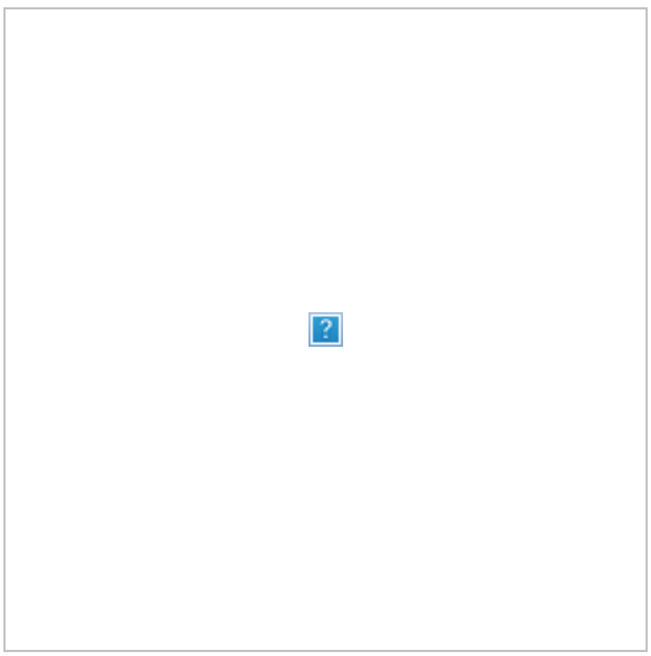
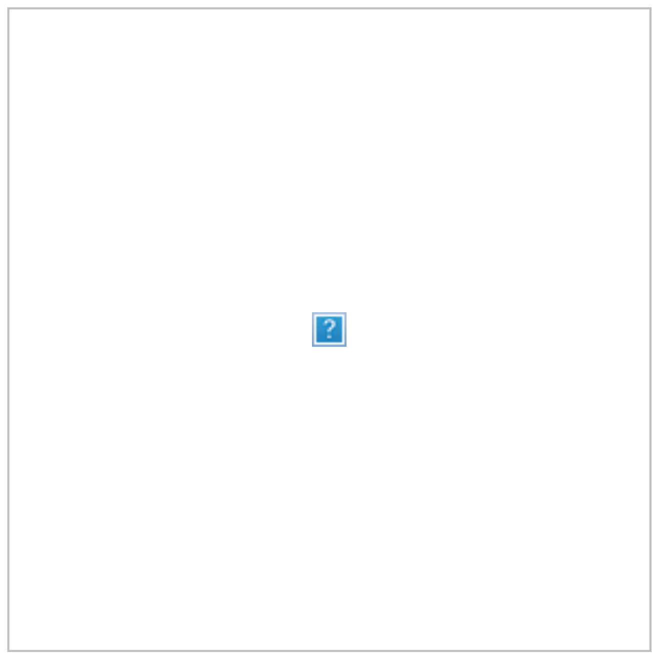
Part 5: Half-edge split

- To implement the half-edge split operation, I also draw two pictures about a mesh before splitting and after splitting, and assign all the elements. The difference in split operation to flip operation is that the split operation will create 3 new edges, 2 new faces, 1 new vertex, and 6 new halfedges. I allocate them, and calculate the midpoint's position by averaging the positions of two vertices along the edge that is being splited. At last, like flip operation, I reassign all the elements.

- Screenshots of *dae/cube.dae* before and after some edge splits.

 Original cube	 After some edge splits

- Screenshots of *dae/cube.dae* before and after a combination both edge splits and edge flips.

 Original cube	 After some edge splits and flips

- To implement the half-edge split operation on boundary edges, I draw two pictures about a mesh with boundary edge before splitting and after splitting. Then, as before, I assign all the elements before splitting, allocate new elements and reassign them.

- Screenshots of *dae/beetle.dae* before and after split operations on boundary edges.

Original beetle	After some boundary edge splits

Part 6: Loop subdivision for mesh upsampling

- To implement the loop subdivision for mesh upsampling, I have completed the following things.

  1. Compute and store the new positions for all vertices in the original mesh, by using this formula `(1 - n*u) * original_position + u * neighbor_position_sum` . And set the value of `Vertex::isNew` to be false.
  2. Compute and store the positions for new vertices that will be inserted at edge midpoints, by using this formula `3/8 * (A + B) + 1/8 * (C + D)` . And set the value of `Edge::isNew` to be false.
  3. Modify function `splitEdge` implemented in Part 5, add some statements to set the value of `Vertex::isNew` and `Edge::isNew` to be true for newly added elements. Split all old edges by using this function and update the new vertices' positions as well.
  4. Iterate over all edges in the mesh to flip the new edges which connect an old and new vertex by using function `flipEdge` implemented in Part 4.
  5. Update the positions for all vertices.

- Screenshots of *dae/torus/input.dae* before and after loop subdivision.

Original torus	Level 1
Level 2	Level 3

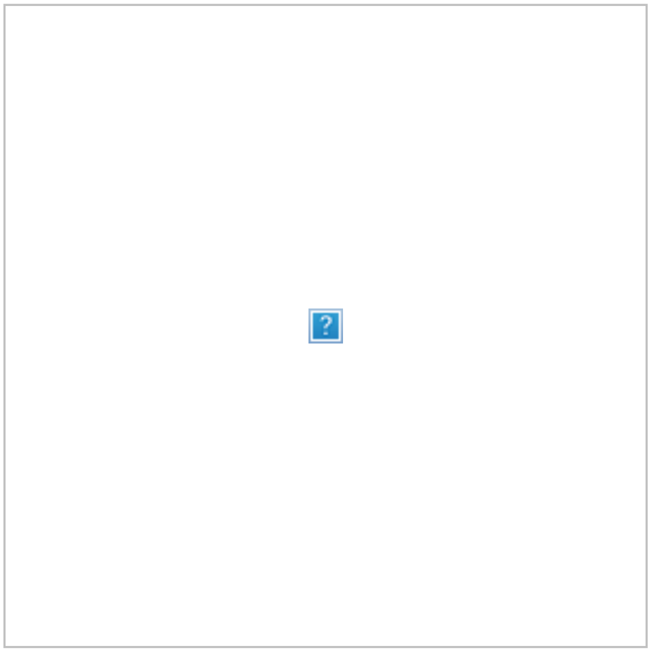
- The sharp corners and edges become much smoother than before after loop subdivision. Fortunately, pre-splitting some edges could be helpful to lessen this effect.

- Screenshots of *dae/cube.dae* before and after loop subdivision without pre-processing.

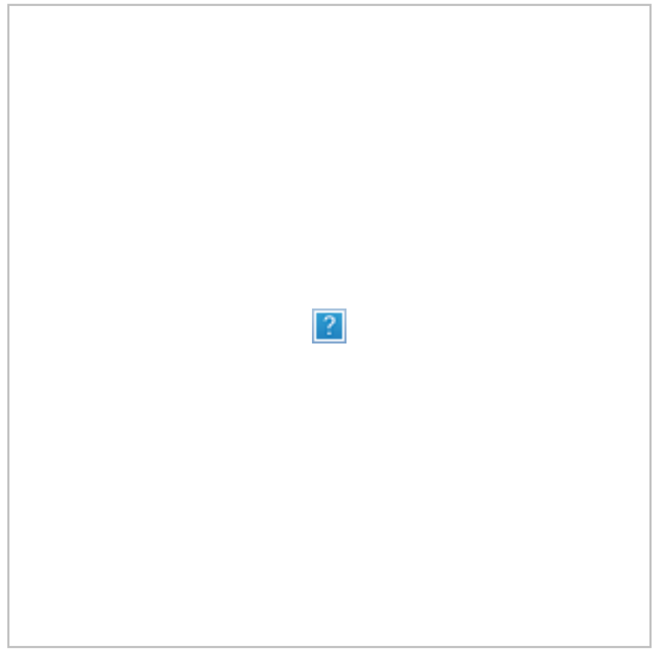
Original cube	Level 1
Level 2	Level 3

- Screenshots of *dae/cube.dae* before and after loop subdivision with pre-processing (pre-splitting all the edges).

Original cube	After pre-processing
Level 1	Level 2



Level 3



Level 4

- The symmetric cube might become asymeetric after severl iterations of loop subdivision. The reason why the cube becomes asymmetrical is because the edges on the cube is irregular. After spliting all the edges in the original mesh to make the edges regular, the effects are alleviated a lot.