**Tech Report**

**A Geometry Kernel for Catmull-Clark Subdivision of Non-orientable 2-Mainfold**

Yu Wang

EECS Computer Sciences, University of California, Berkeley

E-mail: andyatcal@cs.berkeley.edu

# Abstract

This report describes a geometry kernel for the purpose of Catmull-Clark Subdivision of non-orientable 2-manifold.

1. Introduction

For the last two decades, Berkeley SLIDE (Scene Language for Interactive Dynamic Environments) [ref] has been the default geometrical modelling tool for professor Séquin and his students for making mathematical visualization models and abstract geometrical sculptures. SLIDE offers powerful sweep generators, a variety of subdivision techniques and offset surface functions, and convenient hierarchical scene composition tools. On the other hand, no serious maintenance has been performed on this poorly organized code, cobbled together by dozens of students in the period between the 1990’s and 2004. With every new issue of an operating system for Windows machines of Macintosh computers, it gets more difficult to install SLIDE and the Tcl [ref] components that are required for providing interactive change of the parameter that define a geometric shape. The existing SLIDE code also has some technical shortcomings. It cannot provide smooth subdivision and offset surface generation for non-orientable 2-manifolds such as Möbius bands.

This report describes the development of a new geometry kernel to resolve these problems. Many abstract geometrical sculptures have the shape of a (thickened) 2D surface embedded in 3D space [Reference: 2-manifold sculpture]. To emulate these sculptures, this geometry kernel works on the subdivision of 2-manifolds (i.e., thin surfaces with borders). According to Surface Classification Theorem [ref], All 2-manifolds can be uniquely classified by their topology characteristics, including orientability (double-sided or single-sided), the number b of their borders (loops of 1-dimensional rim lines), and their genus g (the number of independent closed loop cuts that can be made on such a surface, leaving all its pieces still connected to one another) [ref]. Catmull-Clark subdivision, regardless of the topology characteristics of the mesh, works well to produce smooth surfaces.

The project pipeline is shown in Figure 1. The subdivision machine takes input of initial polygon meshes and output the result of subdivision. In order to read data for the initial mesh, a SIF file parser was developed. An STL file writer is also export the subdivision mesh. Multiple initial meshes are tested on this geometry kernel. Two detailed test cases, Hild Sculpture and Tetra Sculpture, are described as examples of two-sided surface and one-sided surface in this report.

Input Geometry File

Initial Polygon Mesh

Subdivision Machine

Subdivided Polygon Mesh

Output Geometry File

***Figure 1:*** Catmull-Clark Subdivision Project Pipeline

1. Half-edge Data Structure

We can model a 3D object as polygon meshes. A polygon mesh comprises three types of basic elements: vertex, edge, and face. Adjacency data structure is needed to store the topological information (adjacency and connectivity) between these elements. Several adjacency structures were developed in the past, including simple data structure, winged edge data structure (Baumgart, 1975), half-edge data structure (Eastman, 1982), QuadEdge Data structure (Guibas and Stolfi), and FaceEdge Data Structure (Dobkin and Laszlo, 1987).

Winged edge data structure and half-edge data structure are considered as two options in this project because they have predictable storage size. The storage size and construction time of a mesh are proportional to the number of edges and are independent of mesh topology. It prevents the explosion of storage given a bad input and reduces searching time in subdivision.

We chose half-edge data structure against winged edge data structure for the following reasons.

(1) In winged edge data structure, in order to find the next or previous edge in the traversal, we need to do comparison operation of vertices. However, in half-edge data structure edges and faces have orientations. This reduces searching time for edges and faces in a mesh traversal.

(2) Half-edge data structure has more clear and simple implementation to deal with Mobius connection. A typical definition of edge in half-edge data structure is a pair of half-edges in opposite directions. We extend this definition with Mobius edge and boundary edge in this project to enable mesh traversals and subdivision for single-sided surfaces. In winged edge data structure, with no orientation defined, mesh traversal is determined by the comparison and checking of vertices and faces no matter it is a regular connection or Mobius connection. But we can reduce the running time for regular connections with half-edge data structure.

(3) In the construction of the initial mesh, we might encounter a state of non-manifold geometry even when it is an unfinished 2-manifold. Half-edge data structure stores these temporary edges as boundary edges. When we find their siblings later, we bind them and remove them from the temporary boundary edges. Winged edge data structure, however, will have unclear adjacent faces for these temporary edges.

The definitions and assumptions of geometry elements in this project are shown in Table 1.

*Table 1*: Definition and Assumptions for Geometry Elements

|  |  |  |
| --- | --- | --- |
|  | Definitions | Assumptions |
| Vertex | A 3-dimensional point | The position of a vertex is determined by its x, y, and z coordinate. No two vertices share same ID |
| Edge | A line segment connecting two vertices | An edge connects exactly two non-coinciding vertices |
| Face | A polygon made by a loop of consecutive edges | A face has at least three non-coinciding vertices. It is constructed with a complete loop of consecutive edges with no openings |
| Mesh | A collection of polygon faces | 2-manifold mesh |

**2.1 Vertex**

A vertex is a 3-dimensional point. The position of a vertex is defined by its x, y, and z coordinates. Every vertex has a vertex normal and a unique tracking ID. Coinciding vertices in a mesh are allowed. However, they will never share same ID. In this project, in order to make amortized constant time vertex searching, all vertices are stored in a hash-table, with IDs as their key. Every vertex also contains a pointer to its out-going half-edge.

**2.2 Edge**

An edge is a line segment that connects two vertices. An edge is shared by two faces or it lies on the boundary of the mesh. The orientation of the two adjacent faces can be the same or different. Therefore, three types of edges are defined in this project: regular edge, Mobius edge, and boundary edge.

A regular edge or a Mobius edge is made by a pair of half-edges, as shown in Figure 2(a) and 2(b). In a regular edge, the pair of half-edges has opposite directions and sibling pointers connect them. The two adjacent faces are in the same orientations. On the other hand, in a Mobius edge, the half-edge pair has same direction and Mobius sibling pointers connect them. The two adjacent faces are in different orientations. The vertex on a Mobius edge is marked with a Mobius flag for the purpose of vertex traversal.

Sibling Pointers

Mobius Sibling Pointers

Boundary Pointers

(a) (b) (c)

1. **Figure 2:**  (a) Regular Edge; (b) Mobius Edge; (c) Boundary Edge

A boundary edge has only half-edge. It lies on the boundary of a mesh and it is only adjacent to only one face in the mesh, as shown in Figure 2(c). However, it is also adjacent to another two boundaries half-edges in the mesh and connected with boundary pointers.

In an orientable mesh, where all faces have same orientations, the boundary half-edges flows in the same orientation. We create previous boundary pointer and next boundary pointer for these consecutive boundary half-edges. However, in a non-orientable surface, the flow direction is interrupted and reversed at Mobius connections. Mobius boundary pointers are created at these flow interruptions to specify the connection of boundary half-edges at Mobius connections. Figure 3(a) and 3(b) show examples of boundary edges in meshes of four quadrilateral faces, with and without Mobius connection respectively. Figure 4 shows examples of next boundary pointer, previous boundary pointer and Mobius boundary pointer that occur on four types of boundary connection in the mesh.

And Reversed Here

And Reversed Here

Reversed Here

And Reversed Here

(a) (b)

1. **Figure 3:**  (a) Example for Boundary Edge Flow in an orientable mesh;
2. (b) Example for Boundary Edge Flow in a non-orientable mesh

Next

Boundary Pointer

Previous

Boundary Pointer

Regular Edge

Regular Edge

Next

Boundary Pointer

Mobius

Boundary Pointer

Regular Edge

Mobius Edge

Mobius

Boundary Pointer

Previous

Boundary Pointer

Mobius Edge

Regular Edge

Mobius

Boundary Pointer

Mobius

Boundary Pointer

Mobius Edge

Mobius Edge

1. **Figure 4:** Next, Previous, and Mobius Pointers in Four Types of Boundary Connections

**2.3 Face**

A face is defined as a polygon made by a loop of consecutive half-edges. Every half-edge is connected with its previous and next half-edge a previous pointer and a next pointer respectively. An example of half-edge pointers in a face in shown in Figure 4. Every face contains a pointer to one of its side half-edges.

Next

Next

Next

Next

Previous

Previous

Previous

Previous

1. **Figure 4:** Half-edge Flow and Pointers in a Quadrilateral Face

If two faces share a regular edge, they are adjacent to each other with a regular connection. Otherwise, if they share a Mobius edge, they are adjacent to each other with a Mobius connection.

**2.4 Mesh**

A mesh is a collection of polygon faces. In this project, every mesh contains a list of faces in order to visit every face. It also contains a hashtable of vertices to search vertices gloablly. An auxillary hashtable that map temporoary boundary halfedges to its starting vertex is built for mesh construction.

As a summary, every geometry element stores two types of information: self-information and adjacency information. As shown in Table 2. Adjacency information comprises pointers within a face and pointers between faces. The pointers between faces include sibling pointers and boundary pointers.

*Table 2*: Information Stored by Geometry Elements

|  |  |  |  |
| --- | --- | --- | --- |
| Elements | Self-information | Adjacency within face | Adjacecy between face |
| Vertex | Vertex ID  Vertex Position  Vertex Normal | Pointer to an out-going half-edge | Flag on Mobius Edge |
| Half-edge |  | Pointers to start and end vertex  Pointer to parent face  Pointers to next and previous half-edge in parent face. | Sibling pointers in a regular edge  Boundary pointers to adjacent boundary halfedges |
| Face | Face ID  Face Normal | Pointer to one half-edge in the face |  |
| Mesh | Hash-table of vertices  List of faces  Hash-table of temporary half-edges (in mesh construction) |  |  |

2.5 Mesh Construction

In order to construct a mesh by reading from geometry file or interactive graphics user interface, we need four steps. In order to show the runing time of each step, we denote V as the number of vertices in the mesh and E is the number of half-edges in the mesh.

Step 1: Create individual vertices. We create instances of vertices with their position and ID. This step takes O(V) time.

Step 2: Build individual faces. We create instances of consecutive half-edges, given their start and end vertices. For each half-edge, we add previous pointer and next pointer to its previous half-edge and next half-edge respectively. This step takes O(E) time. When building the half-edge, if its starting vertex does not have a pointer to an outgoing halfedge, we assigned the pointer to the current half-edge.

Step 3: Build regular edge and Mobius edge. We create a hash-table for temporary boundary edge in the construction phase of a mesh. When building the half-edge in Step 2, if the current half-edge shares two same vertices with a half-edge from temporary boundary edge hash-table, they are siblings or Mobius siblings to each other. We build sibling or Mobius sibling pointers for them and remove from the temporary boundary edge table. On the other hand, if we can not find its sibling or Mobius sibling, we add the current half-edge as a temporary bounary edge to the hash-table, with key of its starting vertex. This step takes O(E) time.

Step 4: Build boundary pointers. After we build all the half-edges, those that don’t have a sibling or mobius sibling are left in the temporary boundary hash-table. They are the boundary edges. We build the boundary pointers for these boundary edges with in the following way.

(1) We use a counter to count the number of time we cross mobius edges. We set counter to zero and start building from boundary half-edge,

(2) if the counter is even, we go to next half-edge; if the counter is odd, we go to previous half-edge,

(3) go to the sibling or moibus sibling, the counter increases by 1 if it is a mobius edge,

(4) check if we are now on an boundary edge, if not, we repeat (2) and (3) until we reach to one,

(5) the newly found boundary half-edge shares one vertex with our last boundary half-edge. We build boundary pointers for them,

(6) repeat (2) to (5) to build bounary pointers until we reach the starting boundary half-edge in (1). Now we have built boundary pointers for one boundary loop for the mesh.

(7) repeat (1) to (6) until we built boundary pointers for all boundary edges.

This step takes O(E’) time, where E’ is the number of boundary edges in the mesh.

Now this mesh is built and we have a good initial mesh Catmull-Clark subdivision. As a summary, it takes O(E) time to contruct a mesh from geometry elements.

2.6 Hierachical Scene Construction

Hierachical scene defines the relations between meshes. The final shape in a hierachinal scene may be a composition of several modular mesh pieces, reconstructed by merging or 3D transormation of existing meshes. We developed the following mesh operations to construct a hierachical scene: (1) mesh instantiation, (2) mesh transformation, and (3) merge meshes boundary.

Mesh instantiation make copys of an exsiting mesh by creating new instance for every element that belongs to the original mesh. The new instances of a mesh keep the same vertices positions and elements adjacency. However, vertex ID for the new instances should be different and unique, because we might want to merge several mesh instances later after mesh transformations.

In mesh transformation, we calculate new vertex coordinates by multiplication with the matrix describing the desired transformation. After transformation, the geometry element adjacency remains the same.

To obtain a proper 2-manifold, meshes should only be joined by their borders. We can merge two meshes into a new mesh by merging their boundary edges into regular edge or Mobius edge. Vertex IDs from the two meshes should be different, so no collision occurs after merge.

Two types of mesh merging are implemented in this project: automatic merging and manual merging. In automatic merging, we define a very small tolerance value. If vertices on two boundary edges from the two meshes respectively have distances smaller than the tolerance value, we merge them into a regular edge or a mobius edge. After this merging for every possible pair of boundary edges, two meshes become a single mesh.

In manual merging, we force to merge boundary edges from two meshes even if the distance of their vertices are larger than tolerance value. In practice, we have four ways to perform the manual merging. Assume we would like to force merge the boundary of Mesh 1 and Mesh 2, we can apply one of the following strategies:

(1) Vertex positions in mesh 1 remain the same after merging. Faces on boundary of Mesh 2 extend to the boundary of Mesh 1.

(2) The opposite of (1).

(3) Use the arithmetic mean coordinates for vertices from Mesh 1 and Mesh 2 as the vertex position after merging. Faces on boundary of Mesh 1 and Mesh 2 both extend to these newly created vertices.

(4) Build zip faces to connect two meshes with the vertices from boundary edges of Mesh 1 and Mesh 2.

2.7 Mesh Traversals

The implementation of Catmull-Clark subdivision requires two types of mesh traversals: (1) traversal around a face, and 2) traversal around a vertex. Traversal around a face is used to build face points, edge points, and calculate face normal. Traversal around a vertex is used to build vertex points and calculate vertex normal.

The traversal around a face visits all edges and vertices that belong to this face. It starts from one side half-edge of this face, follows the half-edge flow, and ends at the starting half-edge of the traversal. Traversals of all faces in a mesh takes O(E) time, where E is the number of edges in the mesh.

The traversal around a vertex visits all edges and faces adjacent to this vertex. In a vertex traversal, we need to consider if Mobius edge or boundary edge is adjacent to the vertex. In a regular vertex traversal (when we never visit a Mobius edge or boundary edge), we

(1) start from one outgoing half-edge of this vertex.

(2) process the current half-edge.

(3) we go to the previous half-edge for the sibling of current half-edge.

(4) repeat (2) and (3) and we stop the traversal when visiting the start outgoing halfedge in (1) again.

Figure 5(a) shows an example for a regular vertex traversal. The number on half-edges marks the sequence of visits.

In a vertex traversal with visits of Mobius edges, we need to count the number of times we visit Mobius edges. We will have a different (1) and (3) for this trversal compared to a regular vertex traversal. We

(1) start from one outgoing half-edge of this vertex. Set the mobius edge counter to 0.

(2) process the current half-edge.

(3) if the counter is even, we go to the previous half-edge for the sibling or mobius sibling of current half-edge; if the counter is odd, we go to the next half-edge for the sibling or mobius sibling of current half-edge. When we use the Mobius sibling pointer, the mobius edge counter increase by 1.

(4) repeat (2) and (3) and we stop the traversal when visiting the start outgoing halfedge in (1) again.

Figure 5(b) shows an example for a regular vertex traversal, the number on half-edges marks the sequence of visits.

Start: 1

2

3

4

5

6

7

8

Vertex

Start: 1

2

3

4

5

6

7

8

Vertex

(a) (b)

***Figure 5:*** (a) A regular vertex traversal without visit of Mobius edge and boundary edge;

(b) A vertex traveral with visit of Mobius edge and without visit of boundary edge

If the vertex is on the boundary of the mesh, we will visit boundary edge in its traversal. Compared to a regular vertex traversal, boundary pointers are used instead of sibling pointers in (3). It is also possible that we not only visit Mobius edge but also boundary edge if a vertex lies on the boundary and on Mobius edge at the same time. In such cases, the counter for Mobius edge also increase by 1.

(1) start from one outgoing half-edge of this vertex. Set the mobius edge counter to 0.

(2) process the current half-edge.

(3) if the counter is even, we go to the previous half-edge for the (sibling / mobius sibling / next boundary half-edge / Mobius boundary half-edge) of current half-edge; if the counter is odd, we go to the previous half-edge for the (sibling / mobius sibling / previous boundary half-edge / Mobius boundary half-edge) of current half-edge. When we use the mobius sibling pointer or Mobius boundary half-edge pointer, the mobius edge counter increment by 1.

(4) repeat (2) and (3) and we stop the traversal when visiting the start outgoing halfedge in (1) again.

Figure 6(a) and 6(b) show examples for vertex traversal on a regular boundary connection and Mobius boundary connection respectively.

Start: 1

2

3

4

Vertex

Next Boundary Pointer

Start: 1

2

3

4

Vertex

Mobius Boundary Pointer

(a) (b)

Figure 6: (a) A vertex traversal with visit of boundary edge and without visit of Mobius edge;

(b) A vertex traversal with visit of Mobius edge and boundary edge.

In total, as a combination of situations mentioned above, traversals for all vertices take O(E) time, where E is the number of half-edges in the mesh.

1. Catmull-Clark Subdivision

Catmull-Clark subdivision is a recursive call on the initial polygon mesh that we build in section 2.5 or 2.6. For each level of Catmull-Clark subdivision, we divide every polygon face into N quadrilateral faces, where N is the number of vertices of the polygon. The new vertices for these sub-faces can be classified into three groups: 1) face points, 2) edge points, and 3) vertex points. We also need to add adjacency information to elements in these newly created sub-faces so they are ready for the next level subdvision. Therefore, each level of subdivision is done in two major steps: 1) calculate position coordinates for new vertices, and 2) construct a new mesh given these new vertices.

**3.1 Compute New Vertex Positions**

For each level of Catmull-Clark subdivision, three types of new vertices are created: (1) face points, (2) edge points, and (3) vertex points. We also assign unique IDs for new vertices to avoid collision in the new mesh.

3.1.1 New Face Points

Face points are related to faces of input mesh. For every face in the mesh, its face point is defined as the centroid of all vertices of this face. If we label the face point of polygon face i as , and vertices on the mesh as , the equation to calculate face point is,

In order to get a face point, we do a traversal around the face, find all vertices positions and get their average. Calculating all face points takes O(E) running time, where E is the number of half-edges in the kth level subdivision mesh.

3.1.2 New Edge Points

Edge points are related to edges of the input mesh. In this project, we use the idea from DeRose et al. [ref] and define the sharpness of an edge as infinite sharp or smooth. All boundary edges are defined as infinite sharp because they are adjacent to only one face.

Depending on the sharpness of the edge, we have two ways to calculate the edge point. If an edge is infinite sharp, the edge point is the centroid of two vertices of the edge. If it is smooth, the edge point is the centroid of two vertices of the edge and also the two face points of the two faces adjacent.

We denote the edge point of edge i as , the two vertices of the edge as and , and the two face points of face adjacent as and . The equation to calculate face point is

for infinite sharp edges and

for smooth edges.

Calculating all edge points takes O(E) running time, where E is the number of half-edges in the kth level subdivision mesh.

3.1.3 New Vertex Points

Vertex points are related with vertices from the mesh. In order to find the vertex point, we traverse around an original vertex in the mesh, find the information we need from adjacent faces and half-edges and count the number of sharp half-edges we cross in the traversal. The number of sharp half-edges linking with this vertex determines methods to calculate vertex points.

1) A vertex with three or more incident sharp edges is called a corner, the new vertex point has same position with the original vertex. If we label the vertex point as $v\_p$, the original vertex as $v\_1$, the equation to calculate vertex point is,

$$v\_p = v\_1$$

2) A vertex with two incident sharp edges is called a crease vertex. If we label the vertex point as $v\_p$, the original vertex as $v\_1$ and the two half-edges are labeled with $v\_1v\_2$ and $v\_1v\_3$. The equation to calculate vertex point is,

$$v\_p=\frac{v\_2 + 6v\_1 + v\_3}{8}$$

3) A vertex with less than two sharp edges is a normal vertex. There are two different approaches in calculating the vertex point for a normal vertex. We label the vertex point as $v\_p$, the original vertex point as $v\_1$, the average for all midpoints of edges that contains the original vertex is $v\_2$, the average for all edge points of edges that contains the original vertex is $v\_2'$, the average of the face points of all faces adjacent to the old vertex point as $v\_3$, and the number of faces adjacent to the vertex as $n$, and the number of edges adjacent as $n'$. Catmull-Clark (Reference) defined vertex point as

$$v\_p=\frac{(n - 3)v\_1 + 2v\_2 + v\_3}{n}$$

While DeRose (Reference) defined vertex point as

$$v\_p=\frac{(n' - 2)v\_1 + v\_2' + v\_3}{n'}$$

Catmull-Clark's equation used the average of all midpoints of incident edges and has one more weight on it and one less weight on the original vertex. DeRose's equation used the average of all edge points incident to the vertex. Meanwhile, in a mesh with boundary, $n$ and $n'$ would be different by 1. However, the actual difference between these two equations is very small.

In the example of Figure \ref{figure:differenceCandD}, if we do one more level of calculation and represent the new vertex point only with the original vertices, the results from Catmull-Clark and DeRose equations are,

$$v\_p=\frac{18}{32}v\_0 + \frac{6}{64}(v\_2 + v\_4 + v\_5 + v\_7) + \frac{2}{128}(v\_1 + v\_3 + v\_6 + v\_8)$$

and,

$$v\_p=\frac{27}{32}v\_0 + \frac{3}{64}(v\_2 + v\_4 + v\_5 + v\_7) + \frac{3}{128}(v\_1 + v\_3 + v\_6 + v\_8)$$

respectively. We can see that Catmull-Clark has more weights on the immediate neighbor vertices $v\_2$, $v\_4$, $v\_5$, and $v\_7$, while DeRose has much more on the original vertex. In this project, we used DeRose's equation to calculate vertex point for a normal vertex.

This step takes O(E) running time, where E is the number of half-edges in the kth level subdivision mesh.

3.2 Construct a New Mesh

The newly generated vertices also need to be linked with each other. We compile the new mesh with traversal around face for every face from the original mesh.

For every half-edge $v\_iv\_j$ in the face, we create four new half-edges. If we denote its edge point of the current half-edge as $v\_e$ and the face point of its face as $v\_f$, the four new half-edges are $v\_iv\_e$, $v\_ev\_j$, $v\_ev\_f$, and $v\_fv\_e$. An example in shown in Figure \ref{figure:compileNewMesh}. The half-edge $v\_1v\_2$ generated $v\_1v\_e$, $v\_ev\_2$, $v\_ev\_f$, $v\_fv\_e$. We also add the following adjacency information while generating sub-half-edges:

1) Add one outgoing half-edge pointer to every new vertex. If the new half-edge is on mobius connection, mark the new vertex on mobius connection.

2) Add previous and next pointers to the sub-half-edges.

3) Add sibling pointers to every pair of $v\_ev\_f$ and $v\_fv\_e$.

4) $v\_iv\_e$ and $v\_ev\_j$ need to inherit the sharpness and boundary feature from their parent half-edge,

5) If $v\_iv\_j$ is not on boundary, we add sibling links between $v\_iv\_e$ and the corresponding sub-half-edge of its parent's sibling and add sibling links between $v\_ev\_j$ and the corresponding sub-half-edge of its parent's sibling.

6) If $v\_iv\_j$ is on the boundary and has boundary links with its boundary neighbor half-edges, we add boundary links for $v\_iv\_e$ and the corresponding sub-half-edge of its parent's boundary neighbor. We also add boundary links for $v\_iv\_e$ and the corresponding sub-half-edge of its parent's boundary neighbor.

When the traversal around face is done for every face in the level k subdivision mesh, the level k + 1 subdivision mesh would be created.

This step takes O(E') running time, where E' is the number of half-edges in the (k + 1)th level subdivision mesh.

As a summary, the Catmull-Clark at kth level takes O(E') running time, where E' is the number of half-edges in the (k + 1)th level subdivision mesh.

4 Offset Mesh

The offset with value offVal for a mesh M contains three parts: 1) Positive offset, where we translate every vertex of M by offVal along the direction of its vertex normal, 2) Negative offset, where we translate every vertex of M by offVal along the opposite direction of its vertex normal, and 3) the cover mesh between positive offset and negative offset, which is the extrusion for M's boundaries along positive and negative vertex normal direction. This offset mesh is a two-sided surface with no mobius connections.

In two-sided surfaces, the positive offset mesh is always separated from the negative mesh. However, in single-sided surfaces, the positive offset mesh will be connected with the negative mesh at the mobius connection of the original mesh.

In order to get the offset mesh, we need two steps: 1) calculate vertex normal for all vertices from the original mesh, 2) build positive and negative offset meshes, and 3) build cover mesh between positive and negative offsets.

4.1 Compute Vertex Normals

The vertex normal for a vertex is commonly defined as the average for the surface normal of all faces adjacent to the vertex. To get the surface normal for a face of N polygon, we use Newell's Method.

Newell's method defines the surface normal as the normalized average for cross products of all consecutive half-edges in the face. For a polygon face with n half-edges, we denote them as $v\_0v\_1$, $v\_1v\_2$, ...,$v\_{n-1}v\_0$, where $ n \ge 3$. The surface normal $N\_{f}$ is,

$$N\_{f} = normalize(\sum\limits\_{i=1}^{n-2} (v\_i - v\_{i - 1}) \times (v\_{i + 1} - v\_{i}) + (v\_{n - 1} - v\_{n - 2}) \times (v\_0 - v\_{n - 1}) + (v\_0 - v\_{n - 1}) \times (v\_1 - v\_0))$$

Therefore, we calculate surface normal for every face in the mesh by traversal around face. It takes O(E) running time, where E is the number of half-edges in the mesh.

We calculate vertex normal by traversal around vertex. If a vertex $v$ is not on a mobius connction, we denote the m faces adjacent to vertex $v$ are $f\_0, f\_1, ..., f\_m$, the vertex normal $N\_v$ is,

$$N\_{v} = normalize(\sum\limits\_{i=1}^{m-1} N\_{f\_i})$$

However, when $v$ is on mobius connection, the half-edge flows for some adjacent faces are in opposite directions (clockwise vs. counter clockwise). It leads to surface normal in opposite directions. We need to fix this issue by introducing a mobius counter. In the vertex traversal, we start by adding the surfaces normal when we find a face. When we cross a mobius connection, we add the negative surface normal for the upcoming faces until we cross another mobius connection.

Another way to address this mobius vertex issue is to check the dot product of every surface normal with the sum that we have got. If the dot product is positive, we add this surface normal, if not, we add its negative surface normal.

Calculating all vertex normal takes O(E) running time, where E is the number of half-edges in the mesh.

4.2 Positive and Negative Offset Mesh

For a mesh with no mobius connection, the positive and negative offset meshes will have no connection with each other. We get the positive offset by copy the mesh, and translate every vertex along its normal with an offset value.

For negative offset, we still want to copy the mesh but the half-edge flow in the faces is in opposite directions. We then translate every vertex along the opposite direction of its normal with an offset value.

When there is a mobius connection, positive offset mesh and negative offset will join each other at the mobius connection. Normal of vertices on mobius connection may not reflect the direction for its positive translation.

In this project, we make the translations of vertex positions with traversal around face. If a vertex is on mobius connection, we can check the dot product of its normal and the face normal for the face it belongs to. If the dot product is positive, we translate along this vertex normal for its positive offset and the opposite for its negative offset. If the dot product is negative, we translate along this vertex normal for its negative offset and the opposite for its positive offset.

This step takes O(E) running time, where E is the number of half-edges in the mesh.

4.3 Cover Mesh

We also need to build the cover mesh to connect positive offset with negative offset. For a mesh with no mobius connection, we traverse along its boundary. For every half-edge $v\_iv\_j$ on the boundary, we build a quadrilateral face $v\_{posJ}v\_{posI}v\_{negI}v\_{negJ}$, where $v\_{posI}$ and $v\_{posJ}$ are the positive offset for $v\_i$ and $v\_j$, and $v\_{negI}$ and $v\_{negJ}$ are the negative offset for $v\_i$ and $v\_j$.

As we mentioned before, normal of vertices on mobius connection may not reflect the direction for its positive translation. For a vertex $v\_i$ on mobius connection, we need to check the dot product of its normal with the surface normal that it belongs to. If the dot product is positive, we use $v\_i$'s vertex normal to calculate $v\_{posI}$ and negative vertex normal to calculate $v\_{negI}$. If the dot product is negative, we use $v\_i$'s vertex normal to calculate $v\_{negI}$ and negative vertex normal to calculate $v\_{posI}$. After this check for $v\_i$ and $v\_j$, we build the quadrilateral face $v\_{posJ}v\_{posI}v\_{negI}v\_{negJ}$.

This step takes O(E) running time, where E is the number of boundary half-edges in the mesh.

The offset mesh will be the combination of the positive offset, negative offset and cover mesh that we build above. In total, it takes O(E) running time to build the offset mesh.

4.4 Sharp or Curved?

A question was raised on whether we do offset first or subdivision first. The answer depends on our expectation for the sharpness between the cover mesh and positive/negative offset mesh of the final product. If we want a smooth curve, then offset should be done first. If we want it as a sharp edge, then subdivision should be done first. A combination of pre-subdivision, post-subdivision and offset (e.g. 2 levels of subdivision, offset, another 2 levels of subdivision) can lead to a smooth curve between the two above.

If we do offset first, the initial mesh for subdivision will be a two-sided surface with no mobius connection. The down side of doing offset first, however, is that we have more than twice of amount of elements in the offset mesh. It will take double running time to get the final result.