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

Yu Wang

August 2015

(Table contents is for the review's purpose, will delete once report is done.)

Contents

1 Introduction

For the last two decades, Berkeley SLIDE (Scene Language for Interactive Dynamic Environments) [ref] has been the default geometrical modeling tool for professor Séquin and his students for making mathematical visualization models and abstract geometrical sculptures. On one hand, 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 this 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 subidvivion 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 input and output of this kernel are meshes of polygons. A SIF file parser and STL file writer are developed as an interface to inport the initial mesh input and export the subdivision mesh. Multiple initial meshes are tested on this geometry kernel. Two detailed test cases, Hild Sculpture and Tetra Sclupture, are described as examples of two-sided surface and one-sided surface in this report.

2 Half-edge Data Structure

We can model an 3D object as several meshes of polygons. A mesh comprises three types of elements: vertex, edge, and face. Adjacency data structure is need to store the topological information (adjacency and connectivity) between these elements.

Several adjacency structures have been fully developed, 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). Among all adjacency data structures, the author chooses half-edge data structure in this project to support Catmull-Clark subdivision, because of the following reasons.

1) The storage size of half-edge data structure is porpotional to the number of edges in the mesh, independent of mesh topology. The construction time of a mesh is also porpotional to the number of edges.

2) The edges and faces in half-edge data structure have orientations. This reduced the time of searching edges and faces in a mesh traversal.

The typical definition of edge in half-edge data structure is a pair of halfedges in opposite directions. The author extends this definition (as explained in section **??**) in this project to enable mesh traversals and subdivsion for single-sided surfaces.

The definition and assumption for vertex, edge, face, and mesh is shown in Table **??**.

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. Coninciding vertices in a mesh are allowed. However, they will never share the same ID.

2.2 Edge

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

	Definition	Assumption
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 segements connecting two vertices.	An edge connects exactly two non-coinciding vertices.
Face	A polygon containing a loop of and edges and corresponding vertices.	A face has at least three non-coinciding vertices. It is constructed with a complete loop of edges with on openings.
Mesh	A collection of polygon faces.	

Table 1: Definitions and assumptions of vertex, half-edge, and face

A regular edge is a pair of halfedges with opposite directions, as shown in Figure **??**. A regular edge is not on the boundary of the mesh and it is shared by two faces with same orientation. The two halfedges contain sibling pointers to find each other.

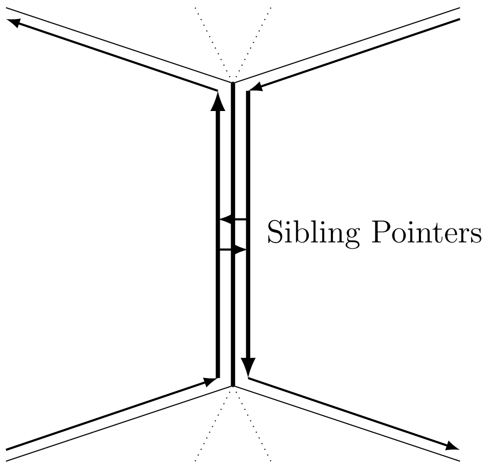

Figure 1: A Regular Edge

A mobius edge is a pair of halfedges with same directions, as shown in Figure **??**. It is not on the boundary of the mesh and it is shared by two faces with different orientations. The two halfedges contain mobius sibling pointers to find each other. Vertices on a mobius edge are also marked with a mobius flag.

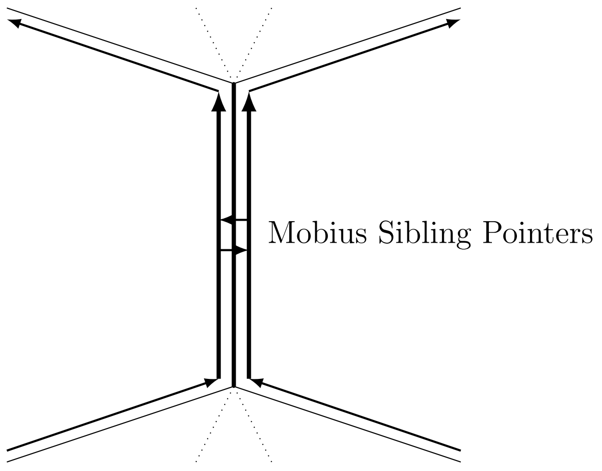

Figure 2: A Mobius Edge

A boundary edge is one half-edge on the boundary of a mesh, as shown in Figure **??**. It is only adjacent to one face in the mesh and adjacent to another two boundaries half-edges. In an orientable mesh, where all faces have the same orientation, the boundary half-edges flow in the same orientation, as shown by an example is shown in Figure **??**. However, in a non-orientable surface, the flow direction is reversed at the connection of mobius edge and boundary, as shown by an example in Figure **??**

2.3 Face

A face is defined as a polygon made by a loop consecutive of half-edges. The start of an half-edge is the end of its previous half-edge, and the end is the start of its next half-edge. Every half-edge contains two pointers, pointing to its previous and next half-edge respectively. An example of half-edge pointers in a face in shown in Figure **??** Every vertex in this face will also have a pointer to its outgoing half-edge.

As a summary, every element stores two types of information: self-information and adjacency information. As shown in Table **??**. The adjacency information includes adjacency in a face and adjacency between faces. The adjacency between faces include sibling links and boundary links. (And we will discussion more in the mesh section.)

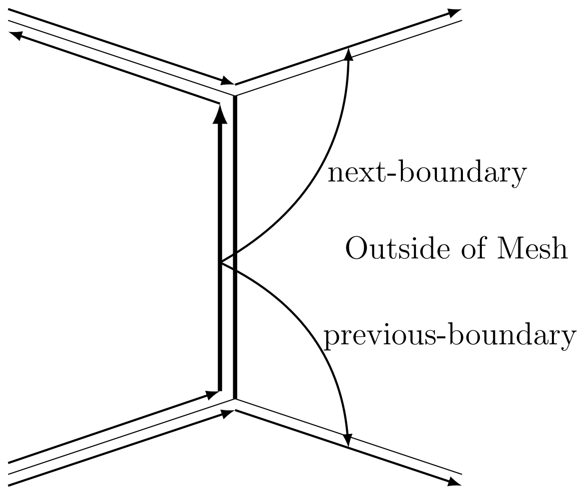

Figure 3: A Boundary Edge

2.4 Mesh

We define a mesh as a collection of basic elements (i.e. vertex, half-edge, and face). Hashtable is implemented to represent the collection in this project, because of its constant serach time for element. We construct three hashtables for all vertices, half-edges, and faces in the mesh respectively. The keys for these hashtable are element IDs and the contents are the element pointers.

The ID of a half-edge is related with the IDs of its start vertex and end vertex. In this project, we define the ID of a half-edge as start vertex ID * maximum number of vertices in a mesh + end vertex ID. This definition will guarantee a unique ID for every half-edge when they have a different start or different end vertex from other half-edges.

A mesh also includes the adjacency and connectivity for elements within. They can be classified into three groups: 1) half-edge flow in a single face, 2) face connections, and 3) boundary connections.

2.4.1 Face Connections and Sibling Links

There are two types of face connections, the normal connection and the mobius connection, as shown in Figure **??**. In a typical half-edge data structure, with the assumption of double-sided surface, a pair of half-edges between two faces is defined with opposite direction. We extends this idea to represent single-sided surface by adding another type of connection, named as mobius

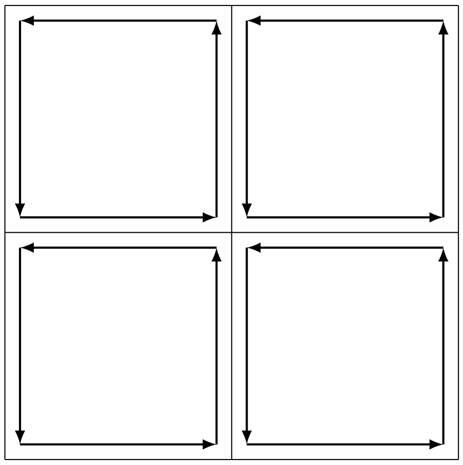
Figure 4: An example of boundary half-edges flow in orientable mesh

connection. In a mobius connection, a pair of half-edges are in same direction. The vertex on a mobius connection will also be marked for the purpose of vertex traversal in the future. In the example of Figure **??**, on the top, e_1 and e_1' are siblings to each other. On the bottom, e_1 and e_1' are mobius siblings to each other.

One thing to point out is that mobius sibling half-edges have the same element ID. Because they have the same start vertex and end vertex. However, we could check for the mobius sibling pointers in order to find all half-edges in the edge traversal of a mesh.

With the extension of mobius connection, there can be three different adjacent situations for a half-edge in a mesh: 1) it is on a normal connection and has a normal sibling, 2) it is on a mobius connection and has a mobius pointer, and 3) it lies on the boundary of the surface and does not have a sibling pointer nor a mobius pointer.

2.4.2 Boundary and Boundary Links

For half-edges on the boundary of a mesh, we connect them with boundary links. In a surface with no mobius connection, the boundary half-edges follow a continuous flow. We can traverse the boundary when starting at one vertex, following the natural flow on the boundary, and ending at the starting vertex. Previous boundary pointers and next boundary pointers are created to link these half-edges. When mobius connection occurs in a mesh, however, some

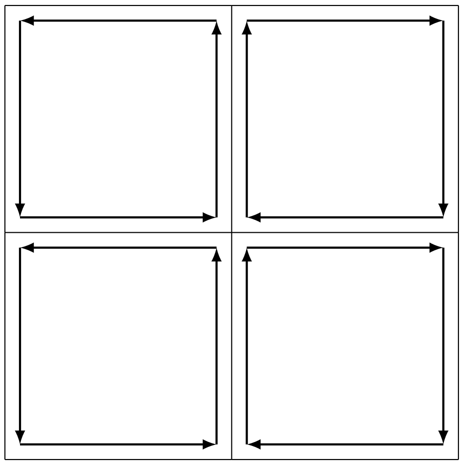

Figure 5: An example of boundary half-edges flow in non-orientable mesh

adjacent boundary half-edges will start or end at same vertex, which blocks the natural flow of boundary half-edges. If this happens, we build mobius boundary pointers to link these half-edges. Figure X shows an example of boundary half-edges with and without mobius connections.

¡Add normal boundary links figure and mobius boundary link figures here!¿

2.4.3 Build Mesh from Elements

As a conclusion of adjacency in a mesh, we need four steps in order to construct a mesh from basic elements.

Step 1: Create individual vertices. We create instances of vertices with their position and ID. The position of two vertices can be the same but their ID should always be unique.

This step takes O(V) time, where V is the number of vertices in the mesh.

Step 2: Construct individual faces. To build an individual face, we create consecutive instances of half-edges, with their start and end vertices. Meanwhile, every vertex will be assigned a pointer to its outgoing half-edge when we create half-edges. For each half-edge, We then add the previous and next pointers to its previous half-edge and next half-edge respectively.

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

Step 3: Build sibling links. For every half-edge, we need to find if there

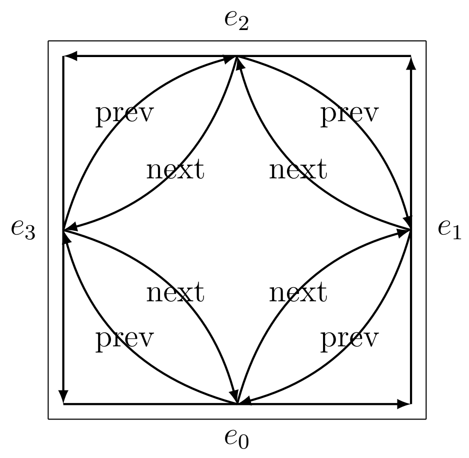

Figure 6: Pointers of half-edges in a single face

exist a sibling or mobius sibling from other faces in this mesh. If its start vertex is same with the end of another half-edge, and its end vertex is same with the start of another half-edge, we then find a sibling. If it has the same start and end vertex with another half-edge, we then find a mobius sibling.

In this project, with the implementation of hashtable, the ID of a half-edge is related with its start vertex ID and end vertex ID. The mobius sibling link is actually generated in step 2. When we create the half-edge, if its ID is equal to a half-edge that we created before, we know they are mobius siblings. The search of normal sibling is also in constant time because we can calculate the ID of its sibling half-edge knowing that start and end vertex are reversed.

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

Step 4: Build boundary links. After step 3, if a half-edge does not have a sibling or mobius sibling, it lies on the boundary of this mesh. We need to build the boundary links to these half-edges.

To do this, we need a counter to keep track of how many times we cross a mobius connection. We 1) set counter equal to zero and start from one half-edge on the boundary, 2) we go to its next half-edge if the counter is even, or its previous half-edge if the counter is odd, 3) we then go to its the sibling or moibus sibling, the counter increase by 1 if it is a mobius sibling, 4) check if the current half-edge is on boundary, if not, we repeat 2) and 3) until we reach to one, and 5) this boundary half-edge shares one vertex with

Element	Self-Information	Adjacency Information
Vertex	1. vertex ID 2. vertex position 3. vertex normal	1. one outgoing half-edge 2. mobius indicator
Halfedge	1. edge ID	1. start and end vertex 2. link to parent face 3. predecessor and successor in the parent face 4. sibling links to adjacent face 5. boundary links to adjacent face
Face	1. face ID 2. face normal	1. one side half-edge

Table 2: Definitions and assumptions of vertex, half-edge, and face

our last boundary half-edge, so we can build boundary links between them, and 6) we repeat 1) to 5) to build bounary links until we reach the starting boundary half-edge in 1). From 1) to 6), one boundary loop of this mesh is built. And we move on to build other loops by repeating 1) to 6), until every boundary half-edges have boundary links to its adjacent boundaries.

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

Now this mesh contains everything that we need to start a Catmull-Clark subdivision. In total, it takes O(E) time from step 1 to step 4.

2.5 Mesh Operations

Building from basic elements is not the only way to create the initial mesh for Catmull-Clark subdivison. A new mesh can also be made by the following mesh operations: 1) copy a mesh, 2) 3D transformation of a mesh, and 3) merging the boundaries for two meshes.

2.5.1 Copy a Mesh

In order to make a copy, we create one element instance for every element that belongs to the original mesh. The copy of a mesh will keep the adjacency for elements and positions of vertices. Therefore, new element instances

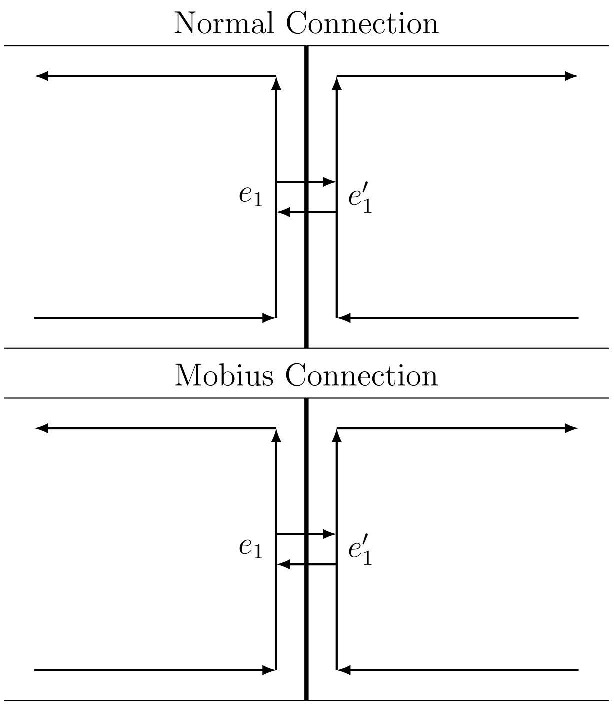

Figure 7: Normal connection and mobius connection between two faces

remain exact same adjacency information as the original mesh, except that the element ID is different and pointers are pointing to the new instances.

2.5.2 Mesh Transformation

In a 3D transformation, the positions for all vertices in the mesh will perform transformation. The adjacency of original mesh remains the same, so half-edges and faces will transform at the same time while vertices transform. Typically, for linear transformations, we multiply the transformation matrix to the position of every vertex from the original mesh.

2.5.3 Merging Meshes

Meshes can be merged into a new mesh if sibling links can be made between some of the boundary half-edges that belongs them. This merge of boundary half-edge will keep the new mesh as a 2-manifold. We also want the elements

ID from these meshes stays different, so no collision occurs after we merge them together.

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 any pair of vertices on the boundary of the two meshes has a distance smaller than the tolerance, we check if their boundary half-edges can be merged with sibling links. If they do, we merge them, continue to trace along the boundaries of these two meshes and merge boundary half-edges until we can't.

In manual merging, we force to merge boundaries from two meshes even if the distance of their vertices are larger than tolerance. 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. The boundary faces on Mesh 2 extend to the boundary of Mesh 1.

2) The opposite of 1).

3) Use the arithmetic mean position for vertex from Mesh 1 and Mesh 2 as the final vertex position after merging. The boundary faces from Mesh 1 and Mesh 2 both extend to these new vertex positions.

4) Build new faces between the two meshes and the vertex positions for Mesh 1 and Mesh 2 remain the same.

2.6 Mesh Traversals

Catmull-Clark subdivision also requires two types of traversals in a mesh: 1) traversal around a face, and 2) traversal around a vertex. Traversal around a face is necessary to build face points and calculate face normal in the face. Traversal around a vertex is necessary to build vertex points and calculate vertex normal in the face.

2.6.1 Traversal Around Face

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

2.6.2 Traversal Around Vertex

The traversal around a vertex leads to all edges and faces that contain this vertex. The traversal of a vertex need to consider two issues: 1) is this vertex on a boundary, and 2) is this vertex on a mobius connection. This makes four different types of vertex traversals. See Figure **??** - **??** for examples.

In the vertex traversal without boundary and mobius issue, we start from one outgoing half-edge of this vertex. We continue to go to the next outgoing half-edge by going to the successor of its sibling until we hit the first outgoing half-edge. In the example of Figure **??**, if start and end at half-edge e_1, the sequence of traversal is: e_1 (sibling link to) e'_1 (successor link to) e_2 (sibling link to) e'_2 (successor link to) e_3 (sibling link to) e'_3 (successor link to) e_4 (sibling link to) e'_4 (successor link to) e_1.

In order to address the issue of a vertex on boundary, instead of using sibling links, we use boundary links. In the example of Figure **??**, we have a boundary e'_4 to e_2. If start and end at half-edge e_1, the sequence of traversal is: e_1 (sibling link to) e'_1 (successor link to) e_2 (boundary link to) e'_4 (successor link to) e_1.

To address the issue of vertex on a mobius connection, instead of using normal links, we use mobius links. At the same time, we switch between the successor and predecessor to the sibling every time we hit a mobius connection. In the example of Figure **??**, e_1 to e'_1 and e_3 to e'_3 have mobius siblings rather than normal sibling. If start and end at half-edge e_1, the sequence of traversal is: e_1 (mobius sibling link to) e'_1 (predecessor link to) e_2 (sibling link to) e'_2 (predecessor link to) e_3 (mobius sibling link to) e'_3 (successor link to) e_4 (sibling link to) e'_4 (successor link to) e_1.

If boundary and mobius connection both occur, we use a combination of the two methods above. In the example of Figure **??**, e_1 to e'_1 and e_3 to e'_3 have mobius siblings rather than normal sibling and we have a mobius boundary connection between e'_4 and e_2. If starting and ending at half-edge e_1, the sequence of traversal is: e_1 (mobius sibling link to) e'_1 (predecessor link to) e_2 (mobius boundary link to) e'_4 (successor link to) e_1.

As a summary of the four different situations above, in a vertex traversal, we 1) start the traversal from with one outgoing half-edge of the vertex, 2) go to sibling or boundary link half-edge, 3) go to the next or previous half-edge that contains the vertex as one end, and 4) repeat 2) and 3) until we reach to the starting outgoing half-edge. This vertex traversal runs in O(E) time, where E is the total number of half-edges in the mesh.

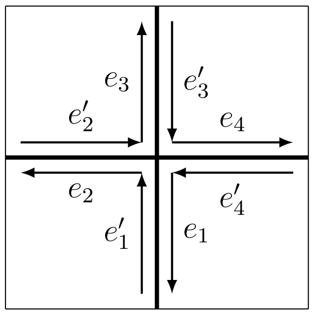

Figure 8: Vertex traversal without boundary and without moibus connection

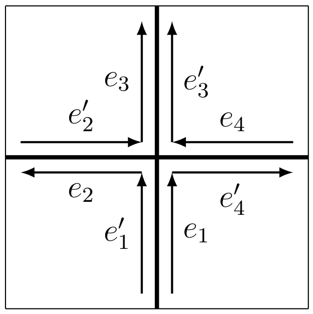

Figure 9: Vertex traversal without boundary and with moibus connection

3 Catumll-Clark Subdivision

Catmull-Clark subdivision is a recursive call on a mesh that we build in section X. Four each level of Catmull-Clark subdivision, we divide every polygon face in the mesh into N quadrilateral faces, where N is the number of half-edges of the polygon face. 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 new sub-faces so that they can be subdivided for the next level. Therefore, each level of subdivision is done in two steps: 1) compute the positions of new vertices, and 2) compile a new mesh given the new vertices.

3.1 Compute New Vertex Positions

There are three steps to compute the positions of vertices in the new mesh: 1) make new face points, 2) make new edge points, and 3) make new Vertex

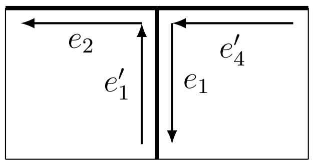

Figure 10: Vertex traversal withboundary and without moibus connection

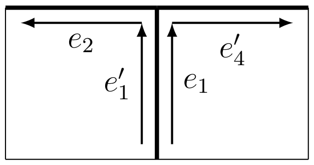

Figure 11: Vertex traversal with boundary and with moibus connection

Points. Apart from finding the positions for these new vertices, we also need to assign unique IDs for them so no collision will occur for the new mesh.

3.1.1 Face Points

Face points are related with faces from the mesh. For every face in the mesh, the position of its face point is defined as the average for positions of all vertices belong to this face. If we label the face point as f and vertices of this face as v_i, the equation to calculate face point is,

$$v_f = \frac{v_1 + v_2 + ... + v_n}{n}$$

Therefore, in order to get the face point position, we can do a traversal around a face, find all vertices and get the average of their positions.

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

3.1.2 Edge Points

Edge points are related with half-edges from the mesh. In this project, we used the idea from (Reference) and define the sharpness of a half-edge as either infinite sharp or smooth. Depends on the position and sharpness of the edge, we have two ways to calculate the edge point.

If a half-edge does not lie on the boundary, it has a sibling link or mobius link to anther half-edge. For the half-edges that are not marked as sharp, the position of edge point of a half-edge is the average of its start vertex position, end vertex position, the face point position of the face it belongs to, and the face point position of the face its sibling or mobius sibling belongs to. If we label the edge point as v_e, the start and end vertices as v_1 and v_2, the face point of its face is v_3 and the face point of its sibling face is v_4, the equation to calculate edge point is,

$$v_e = \frac{v_1 + v_2 + v_3 + v_4}{4}$$

If a half-edge lie on the boundary or it is marked as sharp, the position of the edge point is the average of its start vertex position and end vertex position. If we label the edge point as v_e, the start and end vertices as v_1 and v_2, the equation to calculate edge point is,

$$v_e = \frac{v_1 + v_2}{2}$$

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

3.1.3 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 **??**, 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.

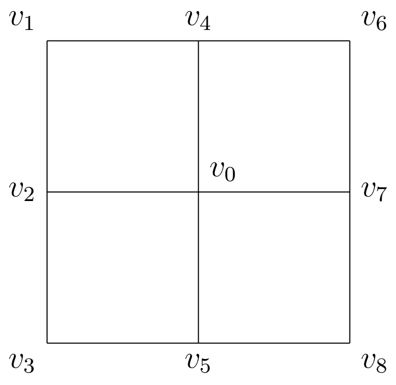

Figure 12: Example of difference of Catmull-Clark and DeRose Equation

3.2 Compile 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 **??**. 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 neigh-

bor.

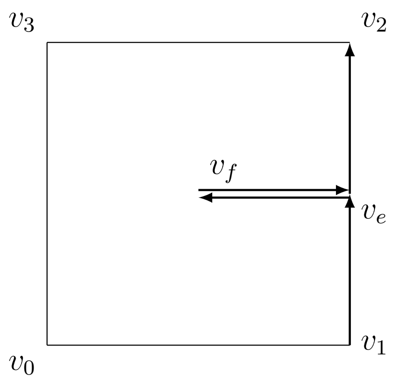

Figure 13: Create four sub-half-edges for one half-edge in subdivision.

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 Normal

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.

4.1.1 Newell's Method and Surface Normal

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 $n \geq 3$. The surface normal N_f is,

$$N_f = normalize(\sum_{i=1}^{n-2}(v_i - v_{i-1}) \times (v_{i+1} - v_i) + (v_{n-1} - v_{n-2}) \times (v_0 - v_{n-1}) + (v_0 - v_{n-1}) \times (v_1 - v_0))$$

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.

4.1.2 Vertex normal

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,

$$N_v = normalize(\sum_{i=1}^{m-1} N_{f_i})$$

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 Meshes

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 $v_i v_j$ 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 Offset vs. Subdivision

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.

5 Input and Output

The input and output of subdivision or offset are polygon meshes. Figure **??** gives a broader view of this subdivision/offset machine. We need to produce the initial mesh for subdivision and display its output mesh in certain ways.

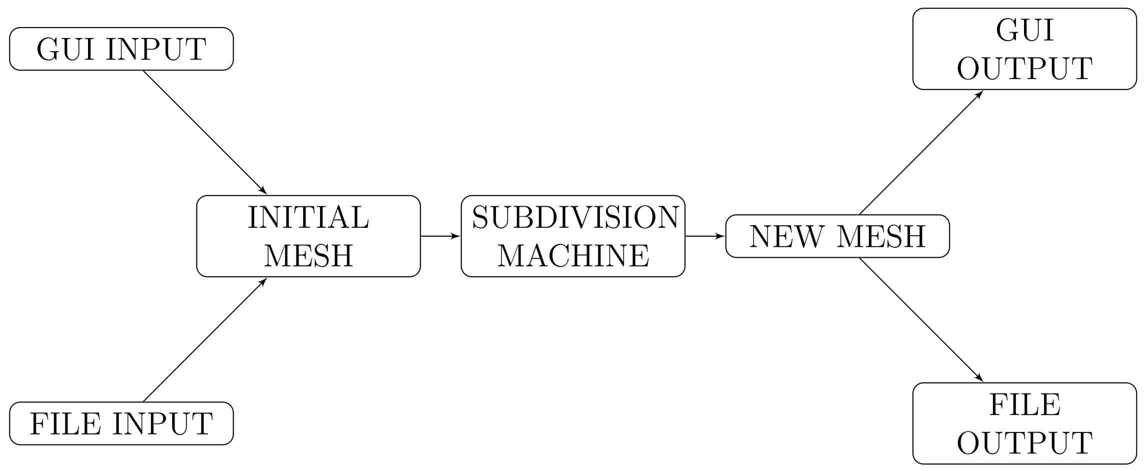

Figure 14: A broader view of the subdivision machine

5.1 Input

The basic information we need to build an initial mesh is: 1) the position for all vertices in the mesh, and 2) the polygon defined with these vertices. This information can be retrieved in two ways. It can come from the graphical user interface. Or it is read from a file contains vertex and polygon information.

5.1.1 Input File Parser

Currently, the input of initial polygon mesh comes from file input, which defines the position of vertices, and how polygons are built from these vertices. As long as the polygons are defined in a continuous flow of half-edges, or a loop of consecutive vertices in the right order, it is good to build the initial mesh of subdivision.

A question was raised on the read of vertices. As mentioned in section X, the assumption of vertex in the mesh is "no two vertices share same position in a mesh". The problem of vertex having same positions is that mesh at that position is seen as a boundary. The subdivision will shrink the mesh at this boundary and so it will be separated. Here, we define "same position" as the distance between two vertex is smaller than a tolerance value.

Reading from the input file, if the parser finds a new vertex having same position that we parsed before, we can merge these two vertices immediately. Or we can keep both of them as different vertices and create a temporary mesh. We then go through the boundary of the temporary mesh and check if any two half-edges can be merged. The problem with merging immediately is that the merge operation should be done on the unit of half-edge in order to maintain the property of 2-manifold.

If we force to merge vertex immediately, it may result in a non-2-manifold geometry. In the example of Figure **??**, v_0 and v_0' are very close to each other (probably smaller than the tolerance we defined). If we merge these two vertices directly, the result will be a non-2-manifold and the program can not figure out how to trace the boundary of this geometry.

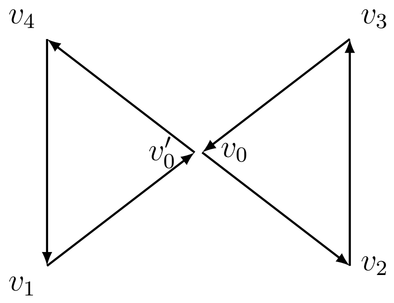

Figure 15: Problem of merging vertex immediately when parsing the file.

Another issue of parsing is associated with the extraordinary points in Catmull-Clark subdivision. Polygons that are not quadrilaterals will generate extraordinary points in the mesh. We preferably want as less extraordinary points as possible. Therefore, when parsing files with triangle definitions, it is a good practice to combine two triangles that share an edge into a quadrilateral.

5.1.2 Graphical User Interface

In a graphical user interface (GUI), a user can 1) draw the points in 3D space, 2) link these vertices into polygons, and 3) tell the program to merge the boundary of two meshes.

In order to do 1), the user can click and select point on screen, or use points on a bezier curve or bspline. To do 2), the user select points consecutively and ends up a loop. In the future, we want the user can do 1) and 2) simultaneously by creating a higher order geometry, like a funnel, tunnel, sweep lines, bezier curve or bsplines.

5.2 Output

5.2.1 Graphical User Interface

In this project, we display the output of the subdivision machine in OpenGL and GLUT (The OpenGL utility toolkit). Arcball feature is applied so the user can rotate the mesh with mouse control and view it from different directions. Keyboard controls are also applied so the user can 1) zoom in and out, 2) switch between flat shading and smooth shading, and 3) switch between the polygon mode and wireframe mode.

5.2.2 Output Files

In this project the output mesh can also be saved and written to a geometry definition file. For example, the STL file produced by this program can be displayed by other viewers, such as 3D Builder from Microsoft or the online viewer of github. It can also be passed to prototype software like QuickSlice and be 3D printed.

6 Test Cases

Two test cases were given to this subdivision machine to test the result of Catmull-Clark subdivision. The initial meshes for these test cases were the work of Professor Carlos Sequin. (The figures are now at the end of this report.)

6.1 Hild Sculpture

Hild Sculpture is a two-sided geometry. The initial mesh is shown in Figure **??**, level 1, level 2, and level 3 subdivisions are shown in Figure **??** to **??**. Offset after 2 level of subdivision is shown in Figure **??**.

6.2 Tetra Sculpture

Tetra is a single-sided geometry with 4 mobius connections. The initial mesh is shown in Figure **??**, level 1, level 2, and level 3 subdivisions are shown in Figure **??** to **??**. Offset after 2 level of subdivision is shown in Figure **??**.

7 Conclusion and Future Works

In this project, with the extended definition of half-edge data structure for single-sided surfaces, we build 1) a subdivision machine that can take in an initial polygon mesh and produce n level Catmull-Clark subdivision surface, 2) an offset machine that can take in a polygon mesh and generate offset mesh, and 3) input and output user interfaces to take in the data and display the result. It can be used to emulate sculptures and be used for the purpose of design and re-engineering.

More works would be done on the input and output end of this subdivision machine in the future. An integrated GUI will be designed to complete the following tasks:

1) Take input with higher order geometries (e.g., funnel, tunnel, bezier curve, bspline, sweep line), and generate initial mesh for Catmull-Clark subdivision.

2) Change geometries parameters with slide bars.

3) With a better interface to display the result.

We also want to use this tool to emulate several Eva or Hild's sculptures in the future.

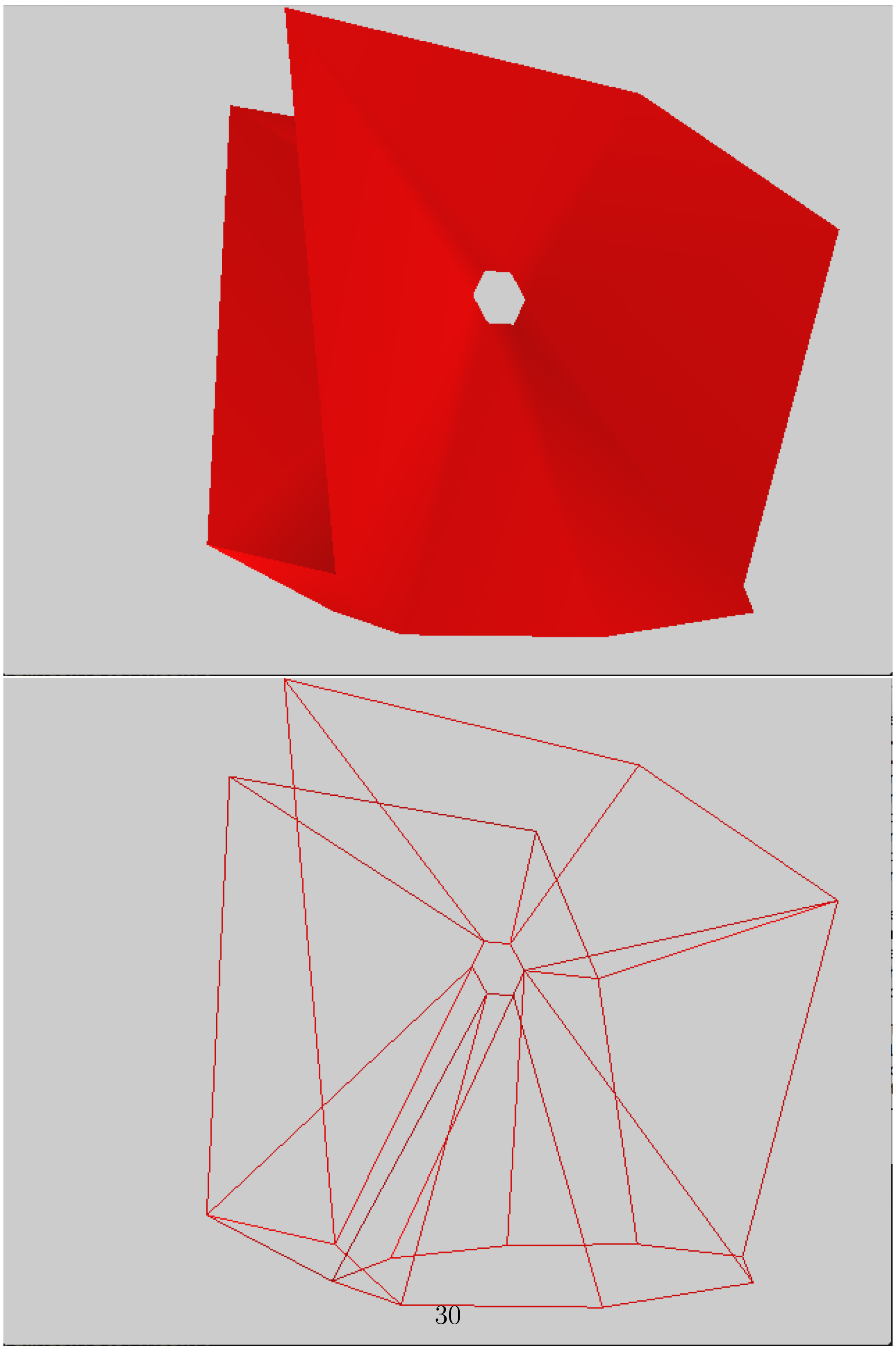

Figure 16: Initial Mesh of Hild Sculpture.

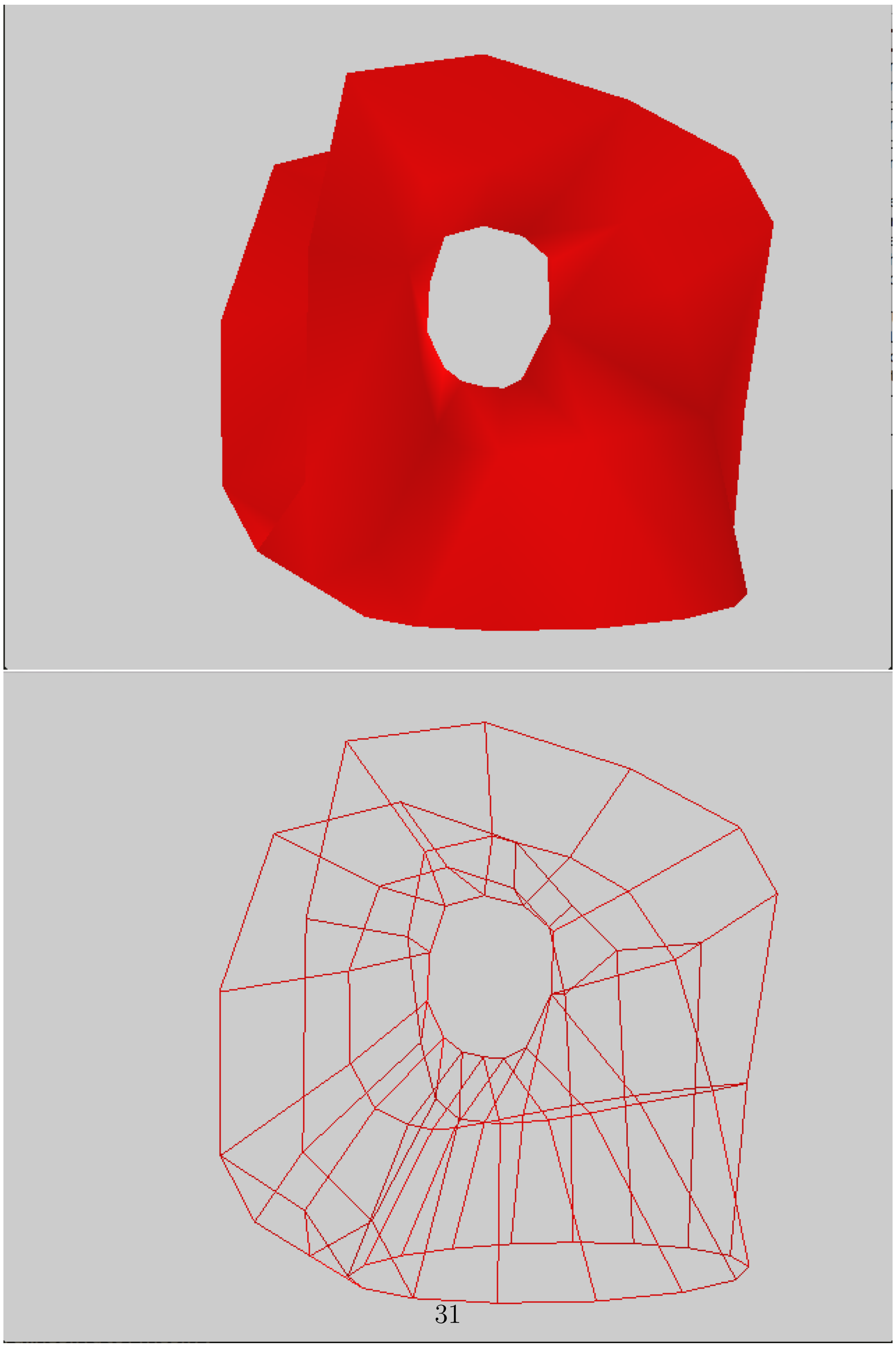

Figure 17: 1 Level Subdivision of Hild Sculpture.

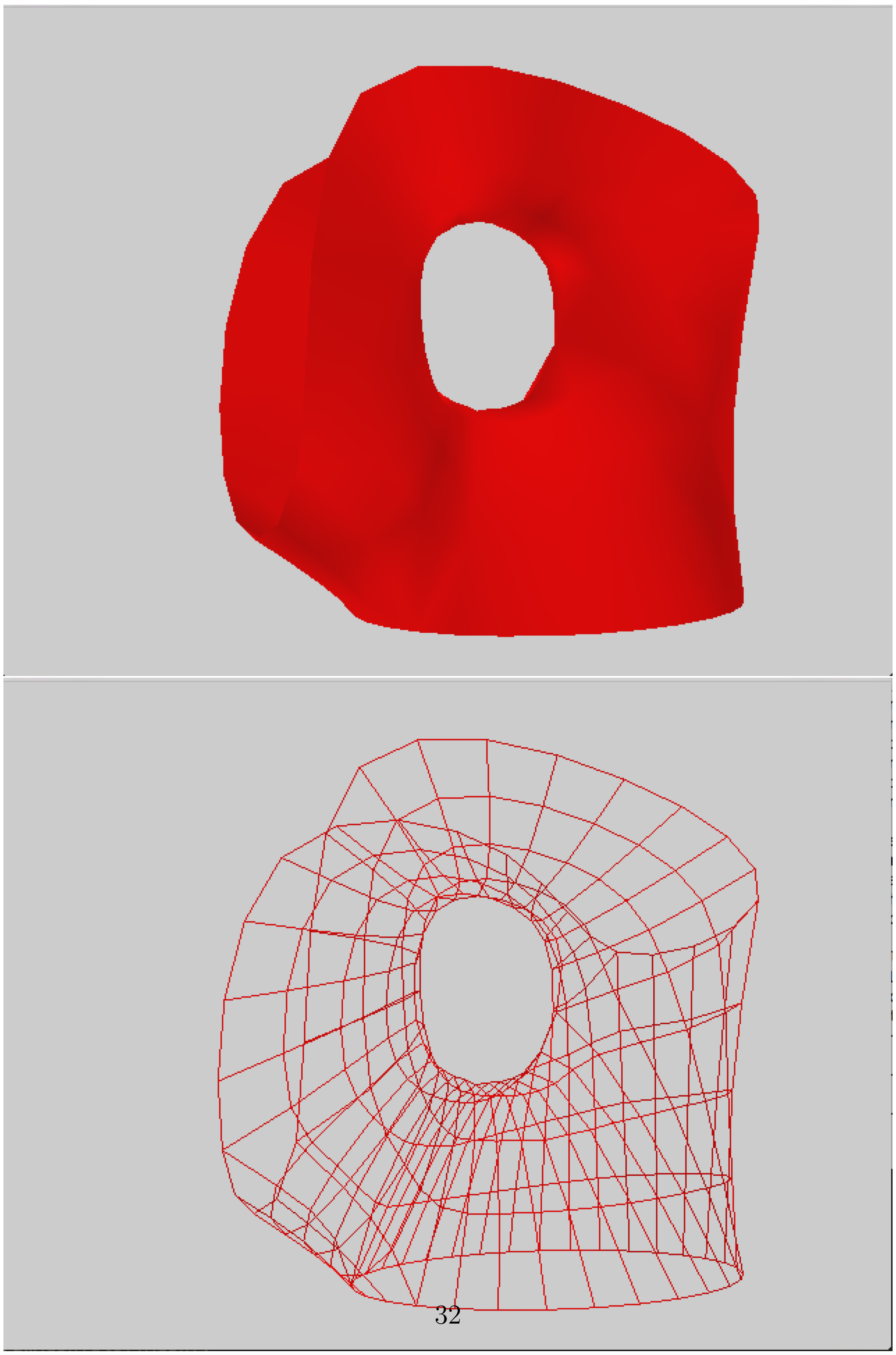

Figure 18: 2 Levels Subdivision of Hild Sculpture.

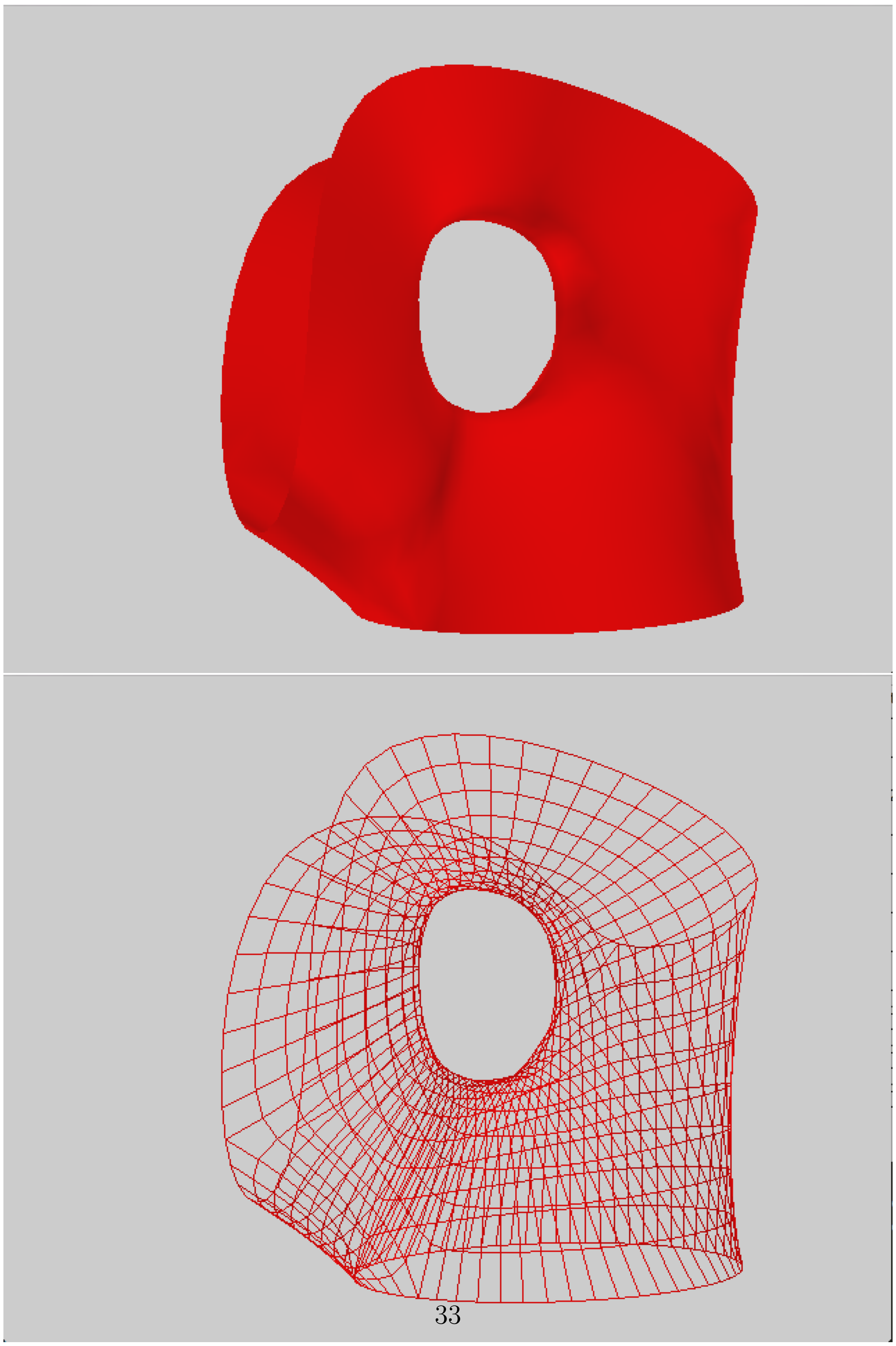

Figure 19: 3 Levels of Subdivision of Hild Sculpture.

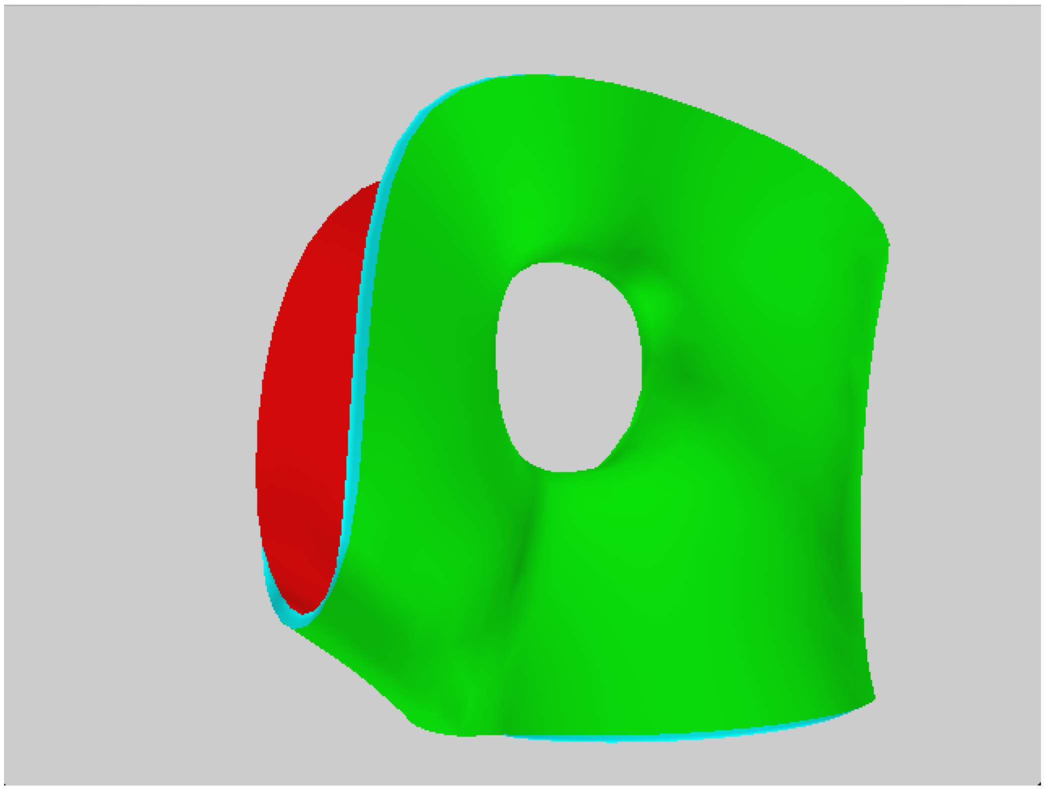

Figure 20: Offset after 3 Levels of Subdivision of Hild Sculpture.

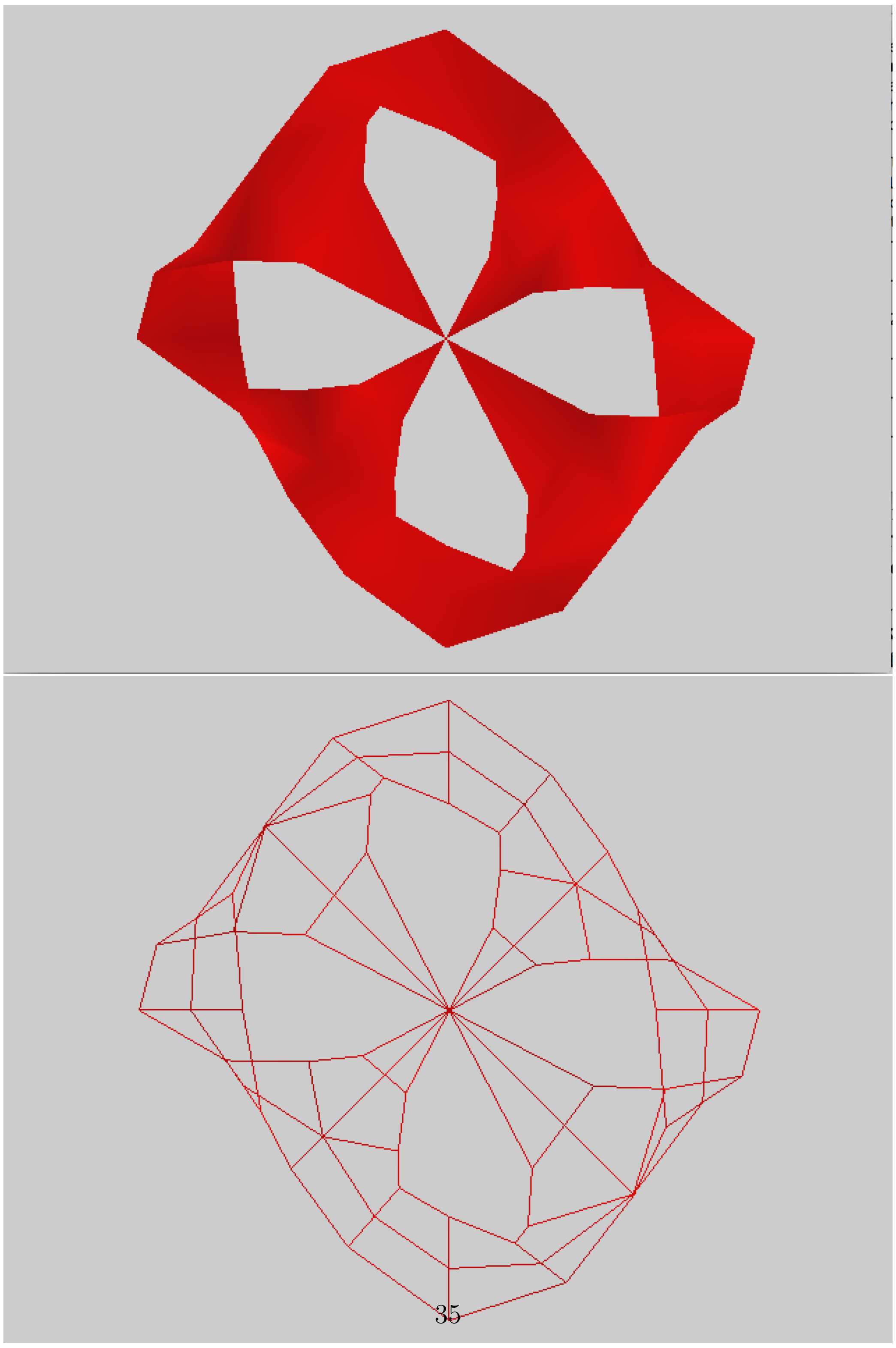

Figure 21: Initial Mesh of Tetra Sculpture.

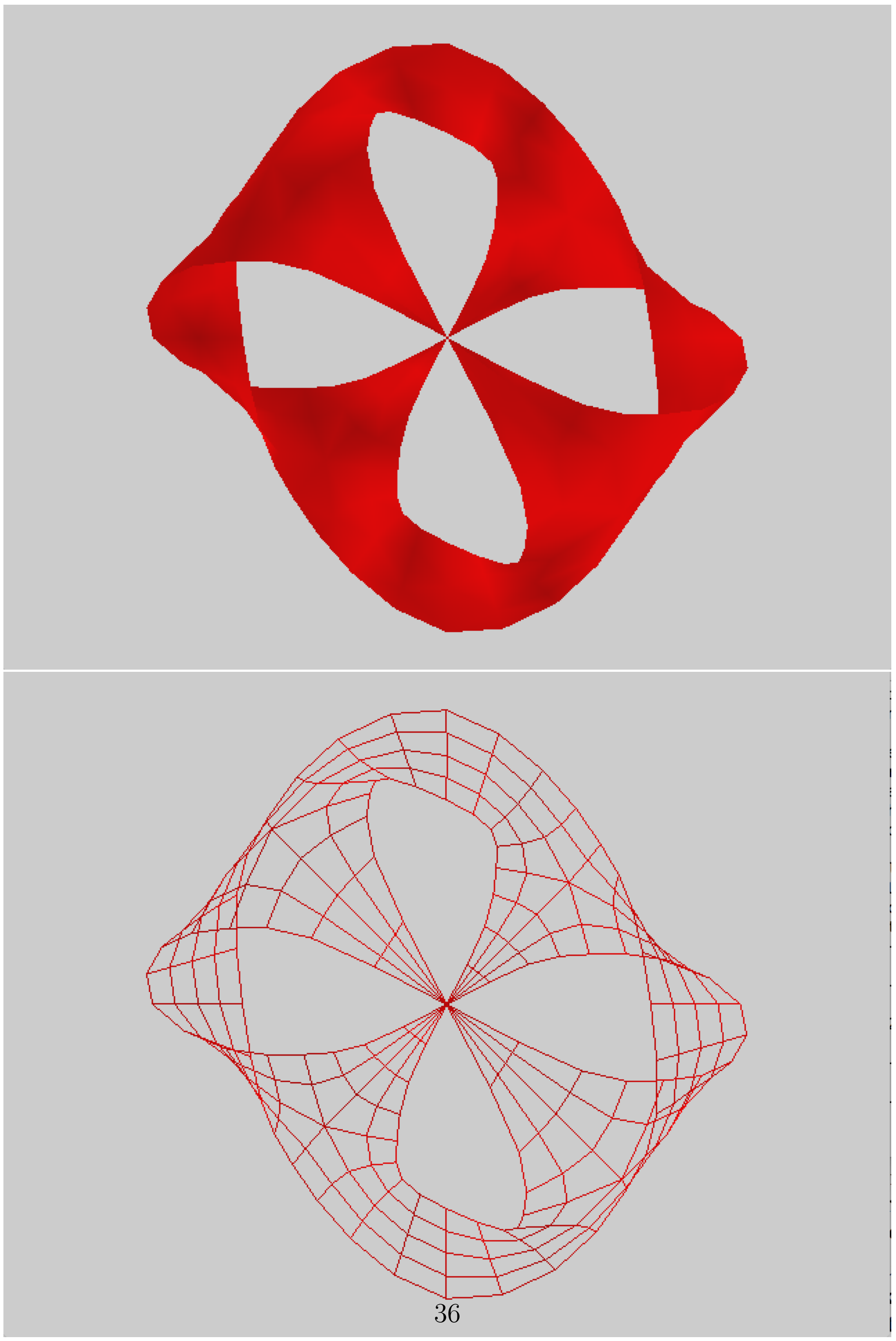

Figure 22: 1 Level Subdivision of Tetra Sculpture.

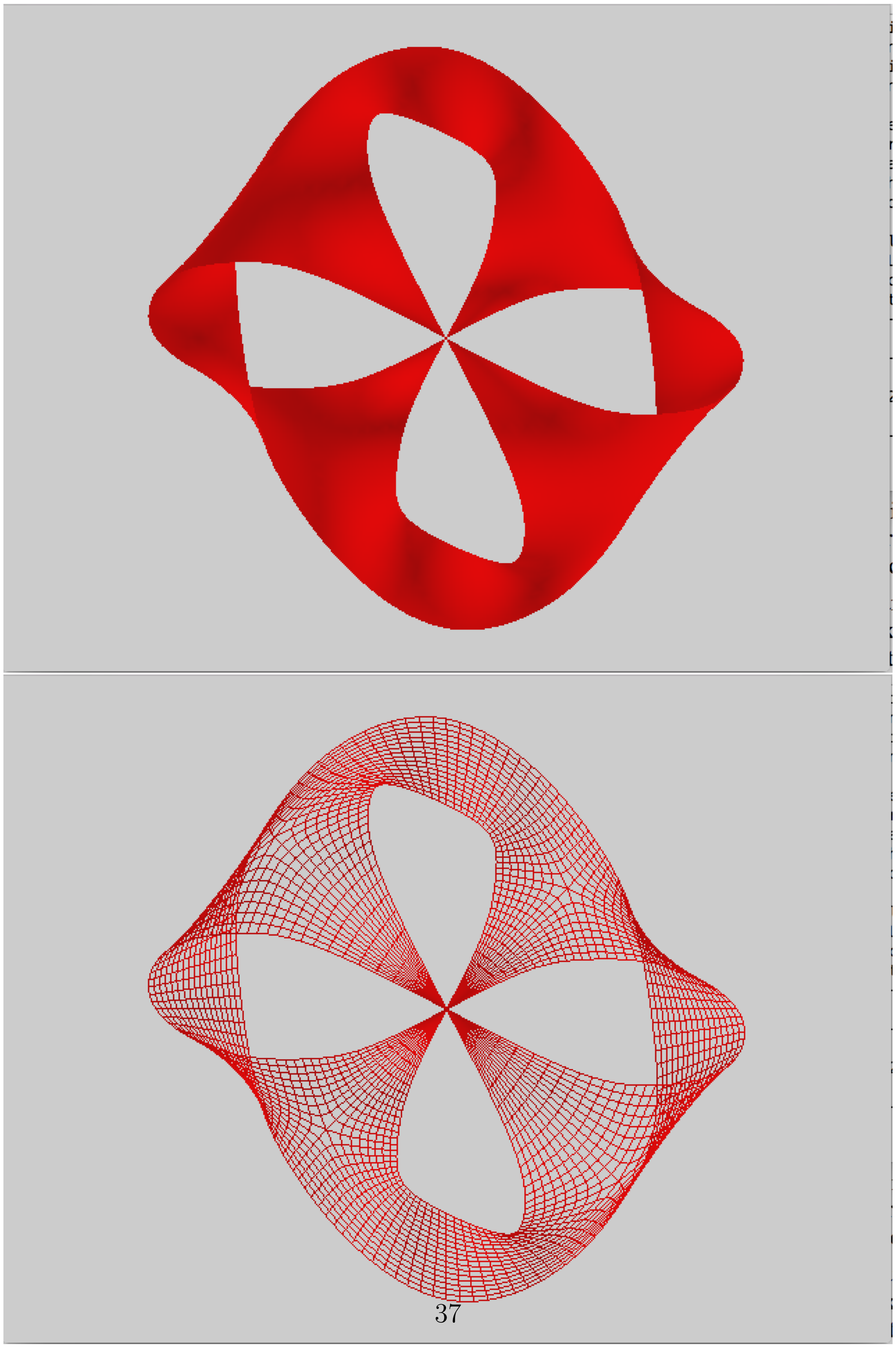

Figure 23: 2 Levels Subdivision of Tetra Sculpture.

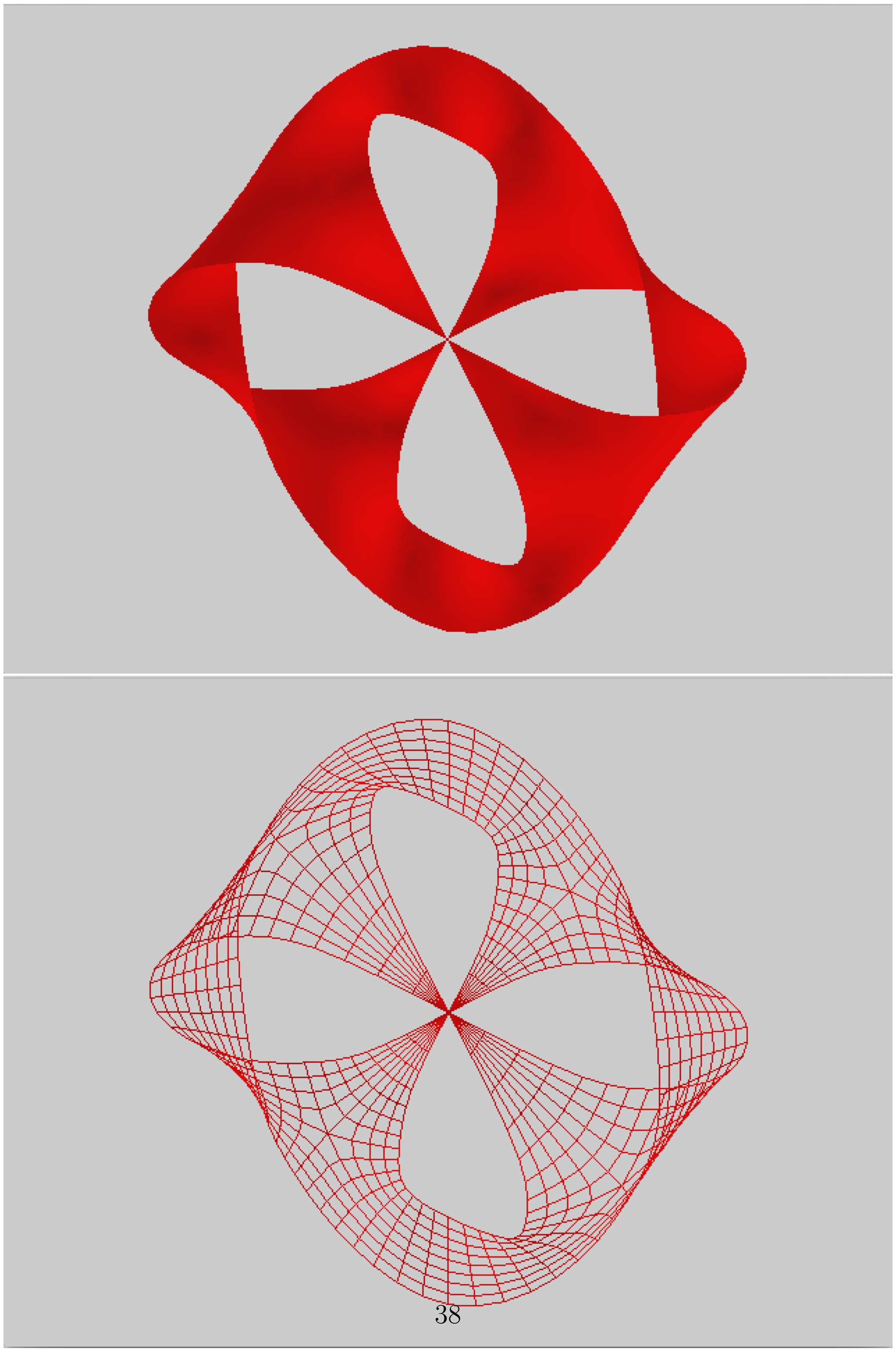

Figure 24: 3 Levels of Subdivision of Tetra Sculpture.

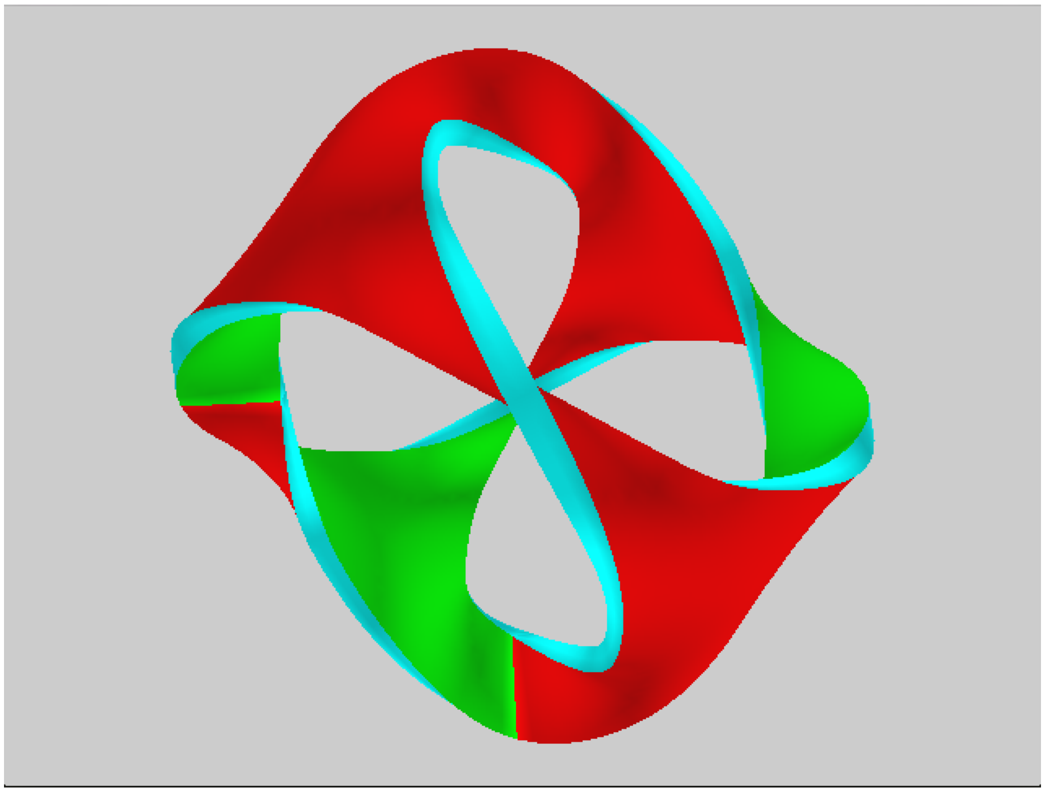

Figure 25: Offset after 3 Levels of Subdivision of Tetra Sculpture.