# Robust Geometry Kernel and UI for Handling Non-orientable 2-Mainfolds

*Yu Wang*

Acknowledgement

I would like to thank my advisor Professor Carlo H. Séquin for his great instruction and guidance during my M.S. study. I would also like to thank Professor Alexei A. Efros for his comments on this technical report. I would like to thank my former advisor Professor Iris D. Tommelein for her support and help when I decided to extend my research field into computer science. I would like to thank Beren Oguz for many helpful discussions and his interest to continue work in this project. I would also like to thank La Shana Porlaris for her support when I was working as a graduate student instructor. Last but not the least, I would like to thank my parents for their deep love and their support for my study.

# Robust Geometry Kernel and UI for Handling Non-orientable 2-Mainfolds

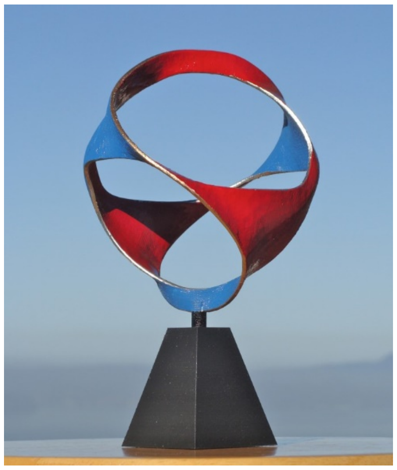
Yu Wang
EECS Computer Sciences, University of California, Berkeley
E-mail: andyatcal@cs.berkeley.edu

## Abstract

This report describes the realization of a geometry kernel and user interface for the purpose of constructing parameterized 2-manifold surfaces, smoothing them with Catmull-Clark subdivision, and offsetting them to generate models that are physically realizable on rapid-prototyping machines. The main focus is to make these operations working robustly also on single-sided, non-orientable 2-manifold such as Möbius bands and Klein bottles. NOME, a language inspired by SLIDE to describe initial meshes in a structural hierarchical form, has been designed. An interactive user interface has been developed to design topologically complex 2-manifolds inspired by the sculptures created by Charles Perry and Eva Hilds.

## 1. Introduction

For the last two decades, Berkeley SLIDE (Scene Language for Interactive Dynamic Environments) [22] has been the default geometrical modeling 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 or Macintosh computers, it gets more difficult to install SLIDE and the Tcl [24] components that provide the capability to interactively change the parameters 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 or Klein bottles, since the data structures used in those routines assume two-sided surfaces. Thus, while it is readily possible to create with SLIDE a smooth model of Charles Perry's "Tetra" sculpture [13] or a variation thereof that is also two-sided (Fig. 1a), it is not possible to


(a)

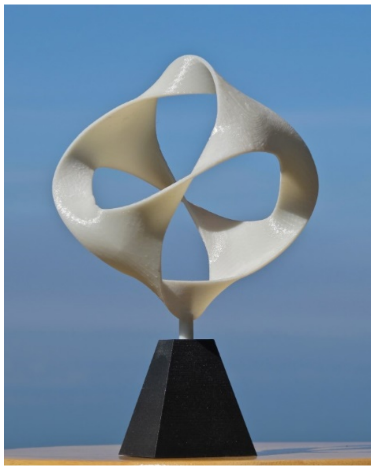

(b)

***Figure 1:*** *Two modifications of Perry's "Tetra" sculpture: (a) two-sided, (b) single-sided. (Photos: C.H. Séquin)*

create the model shown in Figure 1b, which is a single-sided, non-orientable variation of this sculpture. The singled-sided model shown in Figure 1b has been created with the geometry kernel described in this report.

The first part of 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 [19].To emulate these sculptures, this geometry kernel processes general 2-manifolds, i.e., thin surfaces with borders. The Surface Classification Theorem [6] states that all 2-manifolds can be uniquely classified by three topological characteristics: their 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). The new kernel was designed to handle all such 2-manifold, even if they are single-sided and/or self-intersecting. Along such self-intersection lines, one surface branch is oblivious to the existence of the other branch, and there is no connection or reference between the two branches. Thus, for each inner point in such a surface, the local neighborhood is that of a small disk; and for each border point, the neighborhood has the shape of a half-disk. Based on these assumptions, the new kernel can perform smoothing by Catmull-Clark subdivision, and physical thickening of the surface by creating two offset surfaces regardless of the topology characteristics of the mesh.

The geometry kernel accepts input in the form of a crude polygonal mesh in an .OBJ-like format [11] and merges individual polygonal facets into a coherent mesh, which overall may be single-sided or double-sided. It also accepts parameterized geometry input in SLIDE format, which is handled by a newly created .SLF file parser that accepts a relevant subset of SLIDE commands. Non-Orientable Manifold Editor (NOME), a language inspired by SLDIE, is designed to describe meshes in a structured hierarchical form and can handle a subset feature of SLIDE. It enables the user to change the shape of the geometry by moving sliders, with the same controls as in the original SLIDE program. A special .SIF [21] file parser has also been developed, so that static geometrical output shapes composed in the SLIDE environment can be easily transferred to the new geometry kernel. After any input file has been read into the program, the user is able to interactively view and eidt the geometry in the graphical user interface, to add polygons, to merge mesh borders, and to fine-tune a design.

This crude polygonal initial mesh can then be subjected to multiple levels of Catmull-Clark subdivision. The resulting smoothed mesh may be thickened by creating an offset surface on either side; these two offset meshes are connected and properly closed off with additional facets along all the border curves. If the original mesh was intersection-free and did not encroach onto itself closer than the thickness of the physical slab generated by the offsetting process, the resulting mesh will be a clean orientable 2-manifold that describes the surface of a physical part. Optionally this surface can be subjected to further subdivision steps to round off the sharp edges produced along the border curves.

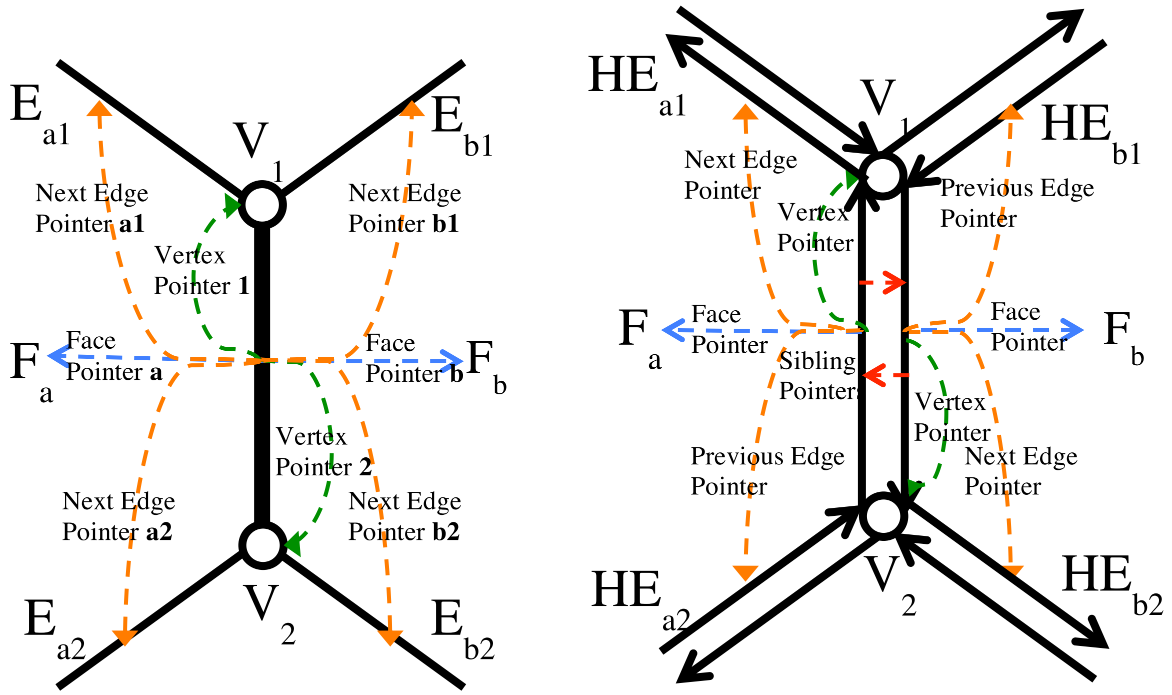
The final mesh can then be output as an .OBJ file for transferring it to other CAD environments for further processing, or it can be output as an .STL file for direct submission to some 3D-printer or other rapid-prototyping machine.

In this report, Section 2 describes the data structures used, and Section 3 to Section 5 show how these data structures are used in the hierarchical scene construction, in the subdivision, and in the offsetting processes. Section 6 discusses the parameterization of the hierarchical scene. Section 7 describes the workflow of the NOME program and the interactive user interface designed to eidt the hierarchical scene. Section 8 illustrates how the new geometry kernel and user interface haave been tested with emulations of sculptures by Charles Perry [13] and by Eva Hild [8], as well as with some abstract geometrical shape in the form of connected sums of Klein bottles.

## 2. Data Structures Used

The 2-manifolds that are the primary application domain of this new geometry kernel are modeled as polygon meshes. Such meshes comprise three basic geometry elements: vertices, edges, and faces. Some linking is needed between those elements to store the topological information (adjacency and connectivity) between those elements. Several adjacency structures have been developed in the past, including the winged-edge data structure [1], the half-edge data structure [5], the QuadEdge Data structure [7],and the FaceEdge Data Structure [4].

The winged-edge data structure and the half-edge data structure were considered as the two primary options in this project, because they associate predictable fixed storage size for the basic elements mentioned above. The storage size and construction time of a mesh are proportional to the number of edges and are independent of mesh topology. This prevents the explosion of storage space and keeps the gathering time for needed information for all mesh-processing operations within well-defined limits. Figure 2 shows the linking between the various elements for the winged-edge and half-edge data structure. The orange arrows are edge to edge pointers, green arrows are edge to vertex pointers, and blue arrows are edge to face pointers.



(a) winged-edge data structure (b) half-edge data structure

***Figure 2:** Linking in the winged-edge (a) and the half-edge (b) data structures [1][5]*

We then chose the winged-edge data structure over the half-edge data structure for the following reasons:

(1) The number of objects and pointers in the winged-edge data structure is smaller. This saves memory space and computation time. As shown in Figure 2a and 2b, primarily the sibling pointers can be eliminated. There are 8 pointers from an edge in the winged-edge data structure and 2 * 5 pointers in the half-edge data structure. When doing traversals around a vertex, we need to cross one more link, between the two half-edges, in the half-edge structure.

(2) In the half-edge data structure, edges and faces have orientations, so it requires complicated conditional statements to implement mesh operations that can handle both single-sided and double-sided

surfaces. In the winged-edge data structure, this information is handled by a couple of flags. With a proper order of pointer assignment (e.g., always assign pointer to $F_a$ before $F_b$), and a flag on the edge to keep track of Möbius condition, the winged-edge data structure can be made compatible both single-sided and double-sided surfaces.

(3) The half-edge structure does not perform well with mesh boundaries compared to the winged-edge structure. In the half-edge structure, we need to introduce boundary edge flags or dummy boundary half-edges for edges in mesh boundary. With the pointer assignment mentioned in (2), the test whether an edge lies on the boundary in the winged-edge structure just checks if the pointer $F_b$ is null or not.

(4) Overall, in a comparison of implementations of both data structures, the winged-edge showed better performance; it was about 10%~20% faster than the half-edge data structure when doing Catmull-Clark subdivision and offsetting.

## *2.1 Geometry Classes*

These are the elements that make up the winged-edge data structure for representing 2-manifold meshes.

### *Vertex*

A vertex is a point in 3D space. It contains three basic fields, (1) a tracking identifier, (2) a 3D vector for its position and, (3) a 3D vector for a vertex normal. Every vertex also contains a pointer to one of its connected edges. In this program, we allow several vertices coinciding at the same position if they belong to different manifold components of a mesh or if they belong to different meshes.

### *Edge*

An edge is a line segment that connects two vertices. In a 2-manifold, an edge is either shared by two adjacent faces or it lies on a mesh boundary. When shared by two adjacent faces, the orientation of these two adjacent faces can be the same or opposite. Therefore, all edges can be classified into three types: regular edges, Möbius edges, and boundary edges.

A regular edge is connected to two faces with same orientations, while a Möbius edges is connected to two faces with opposite orientations. We created a Möbius indicator for every edge to specify if it is a Möbius edge. In an orientable 2-manifold, all faces have the same orientations. However, in a non-orientable 2-manifold (e.g., a Möbius band), face orientations change at its Möbius connections.

In this program, we always first fill in the pointer of $F_a$ before $F_b$. It means for a boundary edge the pointer to $F_b$ is always null. It allows a fast test to find boundary edges. However, when performing face deletion, we will first clear the pointer from the edge to the deleted face, either $F_a$ or $F_b$, and clean up later to maintain this fill in order and build the 2-manifold mesh.

### *Face*

A face is defined by its surronding edges in 3D space. It comprises a cyclic loop of consecutive edges. The edges in a face are not necessarily located in the same plane. A face contains three basic fields, (1) a tracking identifier, (2) a 3D vector for its face normal, and (3) a face color. It also has a pointer to one of its edges.

### *Mesh*

The lean and effcient manifold meshes used in subdivision and offsetting are collections of polygonal faces. The faces collectively form one or more 2-manifolds in the mesh. In this program, every manifold mesh contains (1) a list of faces, (2) a list of vertices, and (3) the color of this mesh. This is the minimal amount of information to get subdivision and offsetting done. The lists of faces and vertices give quick access to the faces and vertices in the mesh. They also provide the basis for face traversals or vertex traversals for general mesh operations.

During read-in and interactive editing of 2-manifold shapes, we use a somewhat richer and more flexiable version, which we call the “initial mesh”. In addition to the above minimal information, it also includes an auxiliary map from a vertex to all its conected edges. Non-manifold vertices may exist in the partly constructed mesh. (See Section 2.2 for more details). Once all the pointers have been built for every edge and the user has finished interactive editing, the user can create a merged copyof this mesh, and the initial mesh lies unused until the user decides to make further changes.

### *2.2 Initial Mesh Construction*

An initial mesh is constructed based on an input geometry file, such as .SIF or .OBJ files. We assume that the initial mesh is given as a set of vertices, with their x,y,z coordinates, and a set of faces, where each face references the vertices that form a cyclic boundary loop around the face.

All vertices, when first encountered, are entered into the vertex list; and this list index will serve as the vertex identifier. For the case where the vertices come with arbitrary string identifiers, a translation map has been introduced.

Faces are entered into the face list based on their order in the input files. Similarly, this list index serves as a face identifier. For every cyclically consecutive pair of vertices appearing in a face specification, we either create a new edge or reference an existing edge from the auxiliary map. When referencing an existing edge, its Möbius edge indicator will be set true if the orientation of new face is opposite to the face located on the other side of edge. Pointers between the newly created edge and its end vertices are added to the auxiliary map for future reference.

After all consecutive edges have been generated for one face, three types of pointers are created, i.e., (1) pointers from an edge to the four edges with which it shares a vertex, (2) a pointer from each vertex to one of its connected edges, and (3) a pointer from each face to one of its edges. After all faces have been created for a mesh, all boundary edges are found and linked together into closed loops by providing pointers from one boundary edge to the next one in this border. A list of border edges is then created for future references.

At this point, all geometry instances associated with the current mesh have been created, and the links between its elements are properly formed. If the given input was a complete design, the auxiliary map can be removed, and mesh construction is complete. Now we have a good 2-manifold mesh for further mesh operations like Catmull-Clark subdivision and offsetting. This construction process takes space and time linear in the number of edges in the mesh.

The input geometry file may also contain parameters that define the positions of vertices or geometrical transformations on instances of a mesh, as provided in an .SLF file. Through this mechanism, we are able to change the geometry interactively when running the program. Parameterization is addressed in Section 6.

### *2.3 Mesh Traversals*

Two types of mesh traversals are commonly used in general mesh operations: (1) looping over all vertices and edges around every face, and (2) looping over all edges and faces around every vertex. For example, in Catmull-Clark subdivision, traversal around faces is used to build face points, edge points, and to compute a face normal. The traversal around vertices is used to build new vertex points and to compute a vertex normal. Examples of face traversals and vertex traversal are shown in Figure 3a and 3b.

The traversal around a face visits all edges and vertices in the border of the face. In this traversal, we start by following the “face to edge” pointer. Then we follow the “next-edge” pointers sequentially to loop over vertices and edges until we hit the starting edge again. As one can tell, we have two possible orientations when traversing around the face. In this program, the orientation of face traversal follows the orientation of the face as it was presented in the input. Traversal of all faces in a mesh visit every edge twice (except boundary edges), so it takes time linear to the total number of edges in the mesh.

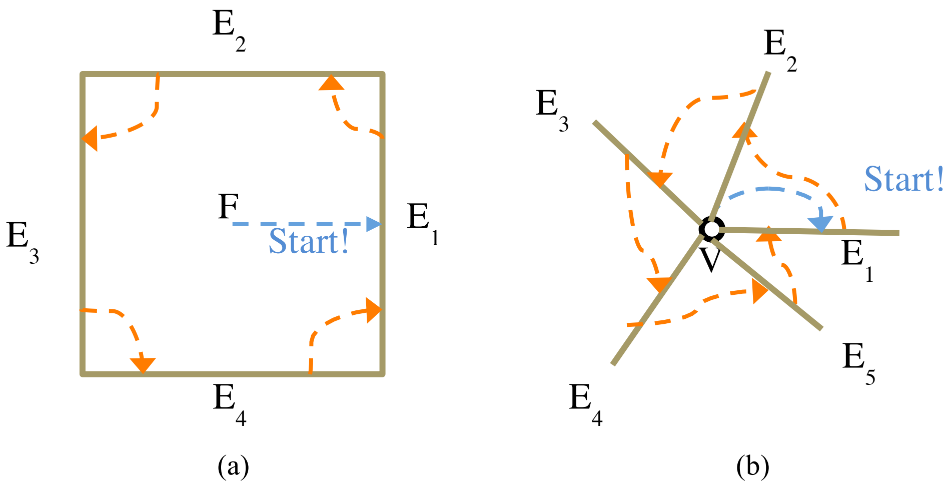


***Figure 3:*** *Example of mesh traversals (a) around a face (b) around a vertex*

The traversal around a vertex visits all edges ending in this vertex. We start following the "vertex to edge" pointer. Following "next-edge" pointers from the current edge to the adjacent edge ending in same vertex, we can visit all adjacent edges and faces until we hit the beginning edge again. Similarly, we have two choices of orienataion when traversing around a vertex. But it is more complicated because different adjacent faces may have different orientations. This orientation of this traversal is very important for normal computations. Special checks need to be made in order to have a consistent face normal and vertex normal. We will discuss this in Section 5. In our program, the vertex traversal follows the orientation of the face that contains the starting edge when it was created in the initial mesh. Traversals around all vertices also visit every edge twice and also take time linear to the total number of edges in the mesh.

## 3. Hierarchical Scene Construction

For efficiency, complex scenes are structured hierarchically where identical pieces of geometry are described only once and then are instantiated mutiple times whenever needed. For example, when building a car mesh, the four wheels are identical except for their positions. It is better to instantiate four copies of one wheel and translate them to the correct position than writing four separate descriptions in world coordinates. For this project, a language inspired by a subset of SLIDE [22] has been designed to describe meshes in a structured hierarchical form; it has been named the Non-Orientable Manifold Editor (NOME) language. A user can define a hierarchical scene by creating a .NOME input file with a text editor.

### *3.1 Structures for Hierarchical Scenes*

Using the NOME language, a directed graph structure is created to organize the relations between meshes at different levels. The general construct to build this grapch is the class "Group". It may contain a list of instances of meshes and of other groups, as shown in Figure 4. Different groups can reference the same meshes or groups by instantiation calls. Every leaf node in this graph is a mesh, with all its geometrical elements.  These leaf meshes can be as simple as a single triangle. Meshes can be transformed as specified in the instance calls in various groups, and those groups can be combined into higher level groups and finally into the whole scene.

Through this hierarchical scene construction, every group, mesh, face and vertex in this program can be assigned a unique hierarchical name by concatenating the names of its parents in the graph separated by "_". This permits global referencing of any vertex for the interactive construction of additional faces.

### *3.2 Mesh Instantiation*

Mesh instantiation makes copies of a previously defined mesh. It allows us to transform, reuse, and merge a mesh multiple times in the final scene. By creating new instances for every geometry element and copying the connections between these elements, we can generate independent copies of the original mesh. As new vertices are generated and entered into the global vertex list, their x, y, z coordinates are calculated based on the transformations specified in the instantiation command. These transformations are realized by matrix multiplication. According to the hierarchical description given in the scene graph, the transformation matrices are concatenated to obtain the final position of a piece of geometry.

This program supports general linear transformations given by $4\times4$ homogenous transformation matrices. The user-friendly commands in the NOME language provide transformations including non-uniform scaling, rotation around an arbitrary axis, translation, and mirroring at an arbitrary plane. In the NOME languange, a user states the transformation for a group or a mesh for going up one level in the the hierarchical scene graph. The user can also define the expressions to parameterize the transformations. So when he/she changes corresponding parameters at run time, all meshes or groups applying this transformation and will be updated for their new positions. (See Section 6 for parameterzation.)

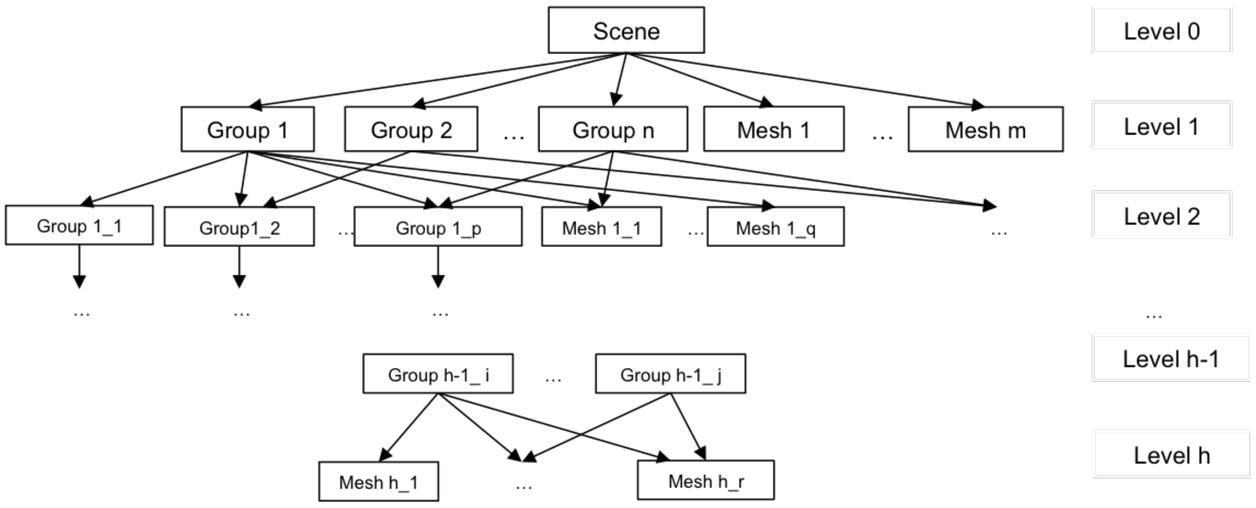


***Figure 4:*** *A directed graph structure for a hierarchical scene.*

### *3.3 Mesh Boundary Merges*

After mesh instantiation and transformation, vertices of two meshes that coincide at the same position are still viewed and treated as different vertices. Some mesh operations (e.g., Catmull Clark subdivision) requires the merger of all mesh instances by joining borders segments. Figure 5 shows the Catmull-Clark subdivision before and a after mesh merge. (More on Catmull-Clark subdivision in Section 4.)

Two types of mesh merging are implemented in this program: automatic merging and manual merging. In automatic merging, the user defines a small merging tolerance to specify a distance within which all vertices should be merged automatically, if this is possible by converting a pair of boundary edges from the two meshes into one inner regular edge or Möbius edge in the merged mesh. During the automatic merging process, some non-2-manifold structures may emerge before all mergers have been done.

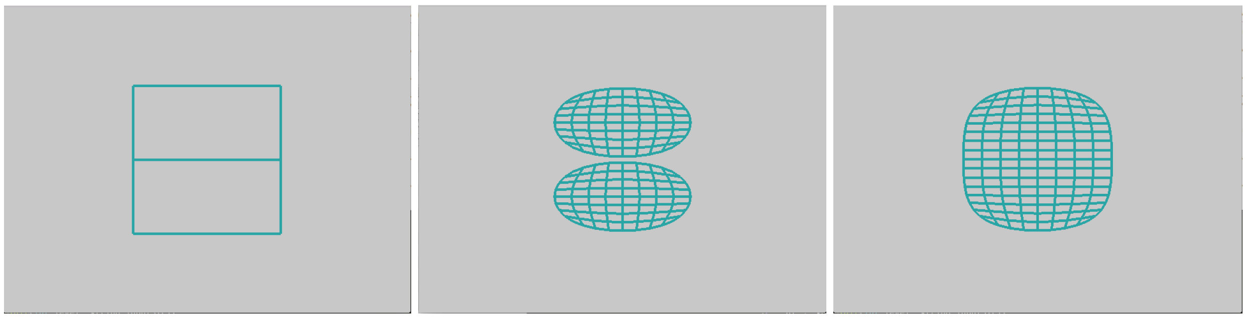


***Figure 5:*** *Example of Catmull-Clark subdivision before and after a mesh boundary merger: (a) Initial meshes, (b) subdivision before the merger, (c) subdivision after the merger.*

For example, two triangles share just one coinciding vertex without any overlapping edges. In this case, we will not yet merge the coinciding vertices. But a third triangle can be inserted between these two triangles to make the merged mesh manifold. Now we need to merge the three coinciding vertices into just one vertex for the merged mesh. To address this problem, we defined the following steps for mesh mergers.

(1) Match boundary edges from two meshes.

The word "match" means the end vertices of boundary edges from two meshes coincide (i.e., their Euclidian distance are smaller than a user-defined merging tolerance). Only boundary edges are being matched because matching inner edges would generate non-2-manifold structures. This boundary edge matching may create a Möbius edge in the merged mesh if its adjacent faces have different orientations.

(2) Merge vertices from matching boundary edges.

Every pair of matching boundary edges involves two pairs of vertices that can be merged. Suppose we are merging mesh 1 and mesh 2, we build a bipartite graph with vertices of mesh 1 on one side and vertices of mesh 2 on the other side. We also build edges in this bipartite graph that indicate two vertices from mesh 1 and mesh 2 can be merged. In this bipartite graph, all vertices in a connected component of the graph will be merged into a single vertex in the merged mesh.

To facilitate the creation of complex free-form 2-manifold structures such as some of Hild's or Perry's sculptures, we also implemented some forms of "manual" merging (zippering) of meshes whose boundaries are not in direct coincidence. The user indicates that two specified segments of two border curves are to be joined, even if the distances of corresponding vertex pairs are not within the merging tolerance. Additional zipper faces will be created using vertices from the boundaries of the two meshes to be connected. This manual merging is supported by an interactive graphical user interface (see Section 7).

## 4. Catmull-Clark Subdivision

A primary application of the new geometry kernel is the recreation of sculptural models inspired by the work of Charles Perry and Eva Hild. The final smooth shapes can be obtained through Catmull-Clark subdivision. This is a process to turn a coarse polyhedral model into a smooth 2-manifold. It is applied recursively on a polygon mesh to make the surface smoother by refining the tessellation. In each level of Catmull-Clark subdivision, every polygon face is subdivied into n quadrilateral faces (n is the number of vertices in this polygon). We also need to build connections of elements in these newly created subfaces, so they will be ready for the next level of subdivision. Hence, each level of subdivision is done in two major steps: (1) calculate position coordinates for a new set of vertices, and (2) construct a new mesh given these new vertices.

## *4.1 Compute New Vertex Positions*

In every level of Catmull-Clark subdivision, three types of new vertices are created: (1) face points, (2) edge points, and (3) vertex points. They are then added to the vertex list of the subdivided mesh and their list indices are used as their identifiers. The algorithms on how new vertices are generated are described below.

### *4.1.1 New Face Points*

Face points are related to the faces from the input mesh. For every face in the mesh, a new face point is defined as the centroid of all vertices of this face. If we denote the face point of face $f$ as $\boldsymbol{v}_f$, and vertices on this face as $\boldsymbol{v}_1, \boldsymbol{v}_2, \dots, \boldsymbol{v}_n$, the face point is

$$\boldsymbol{v}_\mathrm{f} = \frac{1}{n}(\boldsymbol{v}_1 + \boldsymbol{v}_2 + \cdots + \boldsymbol{v}_n) = \frac{1}{\mathrm{n}}\sum_{i=1}^{n} v_i$$

In order to generate all face points, we traverse around every face in the mesh (as described in Section 2.3), visiting all its vertices and calculating the average of their positions. Finding all face points takes time linear in the number of edges in the mesh.

### *4.1.2 New Edge Points*

Edge points are associated with edges from the input mesh. We follow the idea of DeRose et al. [3] and define sharpness of an edge as infinitely sharp or smooth. Following Hooper et al [10] and DeRose et al [3], all boundary edges are infinitely sharp.

Based on the sharpness of the edge, we have two equations to calculate the edge point. If it is infinitely sharp, the edge point is the centroid of its two end vertices; if it is smooth, the edge point is the centroid of the two end vertices and the two face points from the faces sharing this edge.

If we denote the edge point of edge $\boldsymbol{v}_i\boldsymbol{v}_j$ as $v_{ij}$, and the two face points of face adjacent as $\boldsymbol{v}_{f_1}$ and $\boldsymbol{v}_{f_2}$, the edge point for an infinitely sharp edge is

$$\boldsymbol{v}_\mathrm{ij} = \frac{1}{2}(\boldsymbol{v}_\mathrm{i} + \boldsymbol{v}_\mathrm{j})$$

and the edge point for a smooth edge is

$$\boldsymbol{v}_\mathrm{ij} = \frac{1}{4}(\boldsymbol{v}_\mathrm{i} + \boldsymbol{v}_\mathrm{j} + \boldsymbol{v}_{f_1} + \boldsymbol{v}_{f_2})$$

To find all edge points, we traverse around every face of the mesh in order to visit all edges. Every inner edge will be visited twice, but we calculate its edge point only in the first visit. This takes running time linear to the number of edges in the mesh.

### *4.1.3 New Vertex Points*

The new vertex points are derived from the vertices of the initial mesh. According to DeRose et al., the methods to find a new vertex point depends on the number of adjacent sharp edges $n_{sharp}$. If $n_{sharp} \geq 3$, the vertex is defined as a corner vertex, and the new vertex point keeps the same position. If $n_{sharp} = 2$, the vertex is defined as a border vertex (or crease vertex); its vertex point is the weighted average of the two vertices on the two adjacent sharp edges. If $n_{sharp} \leq 1$, the vertex is defined as a regular vertex (or inner crease). However, Catmull-Clark [2] and DeRose et al. [3] have different methods to calculate the new vertex points for a regular vertex. Catmull-Clark defines the new vertex point as the weighted average for the adjacent face points, the midpoints of all adjacent edges, and the original vertex. DeRose et al., use the new edge points of all adjacent edges instead of their midpoints.

If we denote the original vertex as $\boldsymbol{v}_0$, its vertex point as $\boldsymbol{v}$, the number of adjacent faces as $n$, the average for its adjacent face points as $\overline{\boldsymbol{v}}_{faces}$, the average for its adjacent edge points as $\overline{\boldsymbol{v}}_{edges}$, the average for the midpoints of all adjacent edges as $\overline{\boldsymbol{v}}_{mids}$, the average for the midpoints of all adjacent sharp edges as $\overline{\boldsymbol{v}}_{mids\_sharp}$, the equations to calculate the new vertex point are

If $n_{sharp} \geq 3$:

$$\boldsymbol{v} = \boldsymbol{v}_0$$

If $n_{sharp} = 2$:

$$\boldsymbol{v} = \frac{1}{2}(\overline{\boldsymbol{v}}_{mids\_sharp} + \boldsymbol{v}_0)$$

If $n_{sharp} \leq 1$, in the method by Catmull-Clark:

$$\boldsymbol{v} = \frac{(n-3)\,\boldsymbol{v}_0 + 2\,\overline{\boldsymbol{v}}_{mids} + \overline{\boldsymbol{v}}_{faces}}{n}$$

and in the method by DeRose et al.:

$$\boldsymbol{v} = \frac{(n-2)\,\boldsymbol{v}_0 + \overline{\boldsymbol{v}}_{edges} + \overline{\boldsymbol{v}}_{faces}}{n}$$

As the equations show, Catmull-Clark put more weights on edges and less weights on the original vertex. In order to better understand the difference between the two methods, we can calculate the above expressions with just the vertices from original mesh. For the regular point $\boldsymbol{v}_0$ in Figure 6, the Catmull-Clark method gives us a vertex point:

$$\boldsymbol{v} = \frac{72}{128}\boldsymbol{v}_0 + \frac{12}{128}(\boldsymbol{v}_2 + \boldsymbol{v}_4 + \boldsymbol{v}_6 + \boldsymbol{v}_8) + \frac{2}{128}(\boldsymbol{v}_1 + \boldsymbol{v}_3 + \boldsymbol{v}_5 + \boldsymbol{v}_7)$$

And DeRose et al. gives:

$$\boldsymbol{v} = \frac{92}{128}\boldsymbol{v}_0 + \frac{6}{128}(\boldsymbol{v}_2 + \boldsymbol{v}_4 + \boldsymbol{v}_6 + \boldsymbol{v}_8) + \frac{3}{128}(\boldsymbol{v}_1 + \boldsymbol{v}_3 + \boldsymbol{v}_5 + \boldsymbol{v}_7)$$

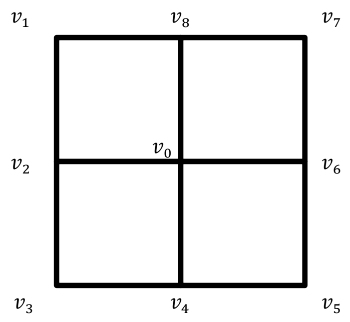


***Figure 6:** Comparison of Catmull-Clark and DeRose et al. methods in finding vertex point.*

Therefore, Catmull-Clark method puts more weights on the neighbor vertices $v_2$, $v_4$, $v_6$, and $v_8$, while DeRose puts more on the original vertex $v_0$. We implemented both methods in this project and will let the users choose their preferred scheme.

In this program, we traverse around every vertex in the mesh to calculate its vertex point. Then, we (1) count the number of adjacent sharp edges, (2) visit all adjacent edges to get their edge points or mid points, and (3) visit all adjacent faces to get their face points. The vertex points are then calculated as described above. This takes running time linear to the number edges in the mesh.

### *4.2 Constructing a Refined Mesh*

Similar to creating the initial mesh, we also need to add all necessary adjacencies to the geometry elements in the subdivided mesh. We traverse around every face from the original mesh, and divide it into *n* quadrilateral faces, where *n* is the number of edges in the face. This is done by connecting the face point with all edge points and connecting the new vertex points with their adjacent edge points. Figure 7 shows the sub-faces of a triangle, a quadrilateral, and a pentagon respectively.

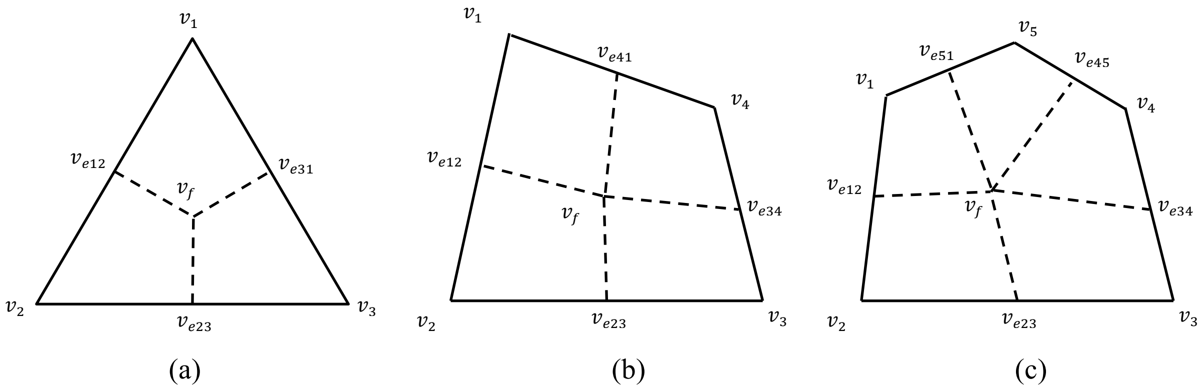


***Figure 7:*** *Example of sub-face construction for (a) a triangle. (b) a quadrilateral; (c) a pentagon.*

In the traversal around the face, for every edge $\boldsymbol{v}_i\boldsymbol{v}_j$, we will create four new edges. If we denote the edge point for $\boldsymbol{v}_i\boldsymbol{v}_j$ as $\boldsymbol{v}_e$ and the face points of its two adjacent faces as $\boldsymbol{v}_{f_1}$ and $\boldsymbol{v}_{f_2}$, the four new edges are $\boldsymbol{v}_i\boldsymbol{v}_e$, $\boldsymbol{v}_e\boldsymbol{v}_j$, $\boldsymbol{v}_{f_1}\boldsymbol{v}_e$, and $\boldsymbol{v}_{f_2}\boldsymbol{v}_e$. Edges $\boldsymbol{v}_i\boldsymbol{v}_e$ and $\boldsymbol{v}_e\boldsymbol{v}_j$ inherit the sharpness of $\boldsymbol{v}_i\boldsymbol{v}_j$. If $\boldsymbol{v}_i\boldsymbol{v}_j$ is a Möbius edge, we will also mark $\boldsymbol{v}_i\boldsymbol{v}_e$ and $\boldsymbol{v}_e\boldsymbol{v}_j$ as Möbius edges. Meanwhile, $\boldsymbol{v}_{f_1}\boldsymbol{v}_e$, and $\boldsymbol{v}_{f_2}\boldsymbol{v}_e$ are always smooth and not Möbius. The sub-faces will inherit the orientation of the original face.

The level k + 1 mesh is created by traversing around every face in the level k mesh. Because this step dominates the total running time of Catmull-Clark subdivision, level k subdivision takes running time linear to the number of edges in the level k + 1 mesh.

## 5. Offset Meshes

After a smooth surface has been generated by Catmull-Clark subdivision, it is converted into a thickend slab that can be realized as a physical model. This is done by generating two offset surfaces and zippering them at the borders.

The result is a double-sided surface with no Möbius edges and no boundary edges. This offset mesh for a mesh *M* contains three parts: (1) a positive offset mesh, where every vertex of *M* is translated by an offset value along the direction of its vertex normal; (2) a negative offset mesh, where every vertex of *M* is translated by an offset value along the opposite direction of its vertex normal; and 3) the border mesh, where the boundary edges of the two meshes are joined with ribbons that have twice the width of the offset distance. If *M* is a double-sided surface, its positive offset is distinct from its negative offset. However, if *M* is a single-sided surface, the positive offset surface is connected with the negative offset surface where there are Möbius edges in *M*; this requires some special checks at such edges.

In order to construct the offset mesh, we will (1) calculate all vertex normals from the original mesh, (2) generate the positive and negative offset surfaces, and (3) construct the border meshes.

## *5.1 Computing a  Vertex Normal*

The vertex normal can be defined in serveral possible ways, e.g., as some average of the surface normals of all adjacent faces of the vertex. In this program, we use Newell's Method [23] to calculate the surface normals for each polygon face,

Newell's method calculates a surface normal as the normalized average for the cross products of all pairs of consecutive edges in the face. If a face has n edges, and we denote these edges as $\mathbf{v}_1\mathbf{v}_2$, $\mathbf{v}_2\mathbf{v}_3$,…, $\mathbf{v}_{n-1}\mathbf{v}_n$, and $\mathbf{v}_n\mathbf{v}_1$, a vector normal for the face $f$ is:

$$\mathrm{N_f} = \sum_{i=1}^{n-2}(\boldsymbol{v}_{i+1} - \boldsymbol{v}_i)\times(\boldsymbol{v}_{i+2} - \boldsymbol{v}_{i+1}) + (\boldsymbol{v}_n - \boldsymbol{v}_{n-1})\times(\boldsymbol{v}_1 - \boldsymbol{v}_n) + \ (\boldsymbol{v}_1 - \boldsymbol{v}_n)\times(\boldsymbol{v}_2 - \boldsymbol{v}_1)$$

We normalize the length of this vector after a complete traversal around the face. The direction of this surface normal is determined by the orientation of the polygon face, which follows the orientation given by the input file.

Now, we can calculate all vertex normals by traversing around every vertex. If a vertex $\boldsymbol{v}_i$ does not lie at the end of any Möbius edge, we calculate its vertex normal as a simple unweighted average of the surrounding face normals,

$$\boldsymbol{N}_{\boldsymbol{v}_i} = normalize(\sum_{j=1}^{m}\boldsymbol{N}_{f_{ij}})$$

where $\boldsymbol{N}_{f_{ij}}$ is the jth adjacent face normal of $\boldsymbol{v}_i$ and $m$ is the valence of the vertex.

However, when $\boldsymbol{v}_i$ is adjacent to a Möbius edge, not all adjacent faces have the same orientations. This results in surface normals pointing in opposite directions. A Möbius edge counter is used in this program to address this problem. Every time we cross a Möbius edge, the orientation of that face normal will be reversed. Therefore, this counter determines if the current face has the same orientation with the starting face in the traversal. In this vertex traversal, we visit every adjacent face of the vertex. Surface normals are added for faces that have the same orientation as the starting face, and are subtracted otherwise. The vertex normal calculated in this way is associated with the starting face in the vertex traversal. It does not necessarily reflect the true "positive" direction of the vertex normal. For convenience we mark these vertices (i.e., those adjacent to at least one Möbius edge) with a Möbius indicator.

## *5.2 Positive and Negative Offset Meshes*

For a mesh with no Möbius edge, the positive and negative offset surfaces are not connected. We get the positive offset surface by instantiating the mesh and translating every vertex along its vertex normal by the chosen offset value. For the negative offset surface, we translate every vertex along the opposite direction of its vertex normal.

For single-sided surfaces, the positive offset and the negative offset meshes will join whereever there is a Möbius edges. Offsetting a vertex marked "Möbius" needs a special check because its vertex normal may not reflect the direction for its "positive" offset. Thus, we build the positive and negative offsets by traversing around every face. If a vertex is on a Möbius edge, we check the dot product of its vertex normal and the face normal for the current face. If the dot product is positive, we translate along this vertex normal for its positive offset, and opposite direction for its negative offset.

## *5.3 Border Meshes*

A border mesh is a connection between the positive and negative offset surfaces. We traverse around the boundary edges of the original mesh. For every boundary edge $\boldsymbol{v}_i\boldsymbol{v}_j$, we build a quadrilateral face with the positive offset and negative offsets for $\boldsymbol{v}_i$ and $\boldsymbol{v}_j$. This creates the zipper faces between the positive offset mesh and the negative offset surface. Similar to Section 5.2, if a boundary vertex is marked Möbius, we make special checks to find its true positive offset direction.

In total, calculating offset meshes takes running time linear to the number of edges in the mesh.

### *5.4 Subdivision of the Offset Mesh*

Though we create the offset meshes after subdivision, the complete offset mesh itself is a two-sided polygonal mesh that can be subdivided further. After its initial construction,  the border surfaces joins the offset surface with a dihedral angle of about 90 degrees. Subdivision on this compound mesh can round off the sharp edges at the rims of the thickened slab.

## 6. Parameterization

We follow the key idea of parameterization from SLIDE [22] and allow the user to enhance the model description with a few adjustable parameters that remain effective from the creation of initial mesh all the way to the final output geometry.

### *6.1 Parameters and Sliders*

The interactive user interface displays a slider for every parameter. A slider includes information about the initial value and the admissible range of its corresponding parameter. At run time, users can change the value of parameters by moving the slider. The effect of the change will be shown to the user immediately on the screen, so that users can make decisions on their preferred values for the parameters.

The user-defined parameters are extracted from the input .NOME or .SLF file. Each includes the name of the parameter and information to create the slider for this parameter. We create a global map from parameter name to parameters, so that the name of a parameter gives quick access to this parameter. Every parameter is also associated with a list of instances that it controls. When the user moves a slider, the value of the corresponding parameter will be updated to its new value, and all instances controlled by this parameter, and only those instances, will be updated. The hierarchical scene is then reconstructed with updated instances, and subdivision and offsetting are applied to the new scene.

In the original SLIDE program, it only stores the parameterized expressions. So every expression need to be re-evaluated every time when the scene is re-rendered. To improve the performance, we also store the current value of every expression in the memory so it doesn't need to be re-evaluated when its corresponding parameters are not changed.

### *6.2 Parameterized Instances*

In theory, every numeric value that helps to define the scene can be parameterized, including (1) the position of a vertex, (2) the color of an instance, (3) the transformation of any instance in the scene. A change of vertex position may results in coinciding vertices in a face. This may lead to a problem in calculating face normals. When this situation happens, a warning message will pop up to alert the user.

The user defines expressions in the input file to specify how to calculate these numeric values. These expressions will be stored and evaluated during the creation of an instance. They will also be re-evaluated when the user moves a slider that controls this instance. Apart from this, we also defined three special generators for parameterized geometries in this program.

### *6.3 Generated Geometry*

One way to control the shape of a mesh is through changing the parameters of a pre-defined parameterized mesh. For example, the user can set two parameters, i.e., the number of edges $n$ and the radius $r$, to define a regular $n$-gon.  All vertex positions can be calcuated based on these two parameters instead of defining every vertex position independently. In this program, we provide three basic parameterized meshes: (1) *Funnel*, (2) *Tunnel*, and (3) *Rim Ribbon*. The user can create instances for these meshes in the input geometry file with expressions of their pre-defined parameters.

A "*Funnel*" is a truncated cone (*n*-gonal pyramid) standing on the x-y-plane, centered at the origin. The base circle of radius $r_o$ will become an outer border curve in the intended sculpture. The upper circle, formed by the truncation of the cone, is of radius $r_i$ and will become an inner poly-line of edges that connect with the rest of the sculpture mesh. Table 1 shows the parameters of a funnel.

**Table 1: Parameters of a Funnel**

| Parameters | Description |
|---|---|
| $r_o$ | Radius of base circle. |
| $dri = (r_i - r_o)/r_o$ | The fractional amount by which upper circle differs from the base circle. Its range should set from -0.5 to +0.5. |
| *hgt* | The height of high circle above base circle ($z = hgt$). |
| *nsg* | The number of segments in a regular polygon used to approximate the circles |

A "*Tunnel*" is made by two "funnels" sharing the same rim circle and going off in opposite directions. When it is symetrical on the x-y-plane, it has the same parameters as a funnel. Otherwise, it has two more parameters $dri_{low}$ and $hgt_{low}$, They specify the radius and height of the lower circle of the "negative" cone.

A "*Rim Ribbon*" is a "free-form" funnel. It is defined by a polyline and a ribbon made from this polyline. Its parameters include (1) the ribbon width, (2) the initial azimuth direction, and (3) the final twist of the ribbon. [In our current program, we just plot a rim line. We draw a cube around every vertex so the user has some finite-size geometry to select by clicking on these cubes. The size of the cube is propotional to the average lengths of edges connected to this vertex. We can change the shape of the rim line by moving the position of vetices in the rim line. An implementation of a full rim-ribbon is under-development].

When changing the parameters of these special generated meshes, it can result in adding or deleting vertices in the mesh. For example, when editing a funnel, some of its previously defined vertices are deleted if parameter *nsg* decreases, and new vertices are added if *nsg* increases. This change can propagate, and some of the temporary polygon added by the user in the interactive editing mode will be deleted or deformed. To prevent this from happening, after the user enters interactive editing mode, these parameters can no longer be changed.

## 7. Interactive User Interface

In addition to changing values of the user-defined parameters, we also empower the user to edit meshes on the fly in an interactive graphical user interface (GUI). The interactive user interface handles the workflow from building the initial hierarchical scene from an input geometry file all the way to exporting the completed model to a prototyping machine.

This interactive user interface is implemented in OpenGL [12] and Qt [16]. In this program, all user interface widgets are created by the Qt framework, including buttons, dialogs, windows, sliders, and the OpenGL canvas. Mouse and keyboard actions trigger by the user, for example moving a slider, will activate a signal and call the functions of the appropriate widgets in the program. This provides the basis of the interactive UI.

We designed and implemented five user interface blocks in this program: (1) the input file dialog and the input file parsers, (2) the OpenGL canvas to show and edit the scene, (3) the slider panels to change parameter values at run time, (4) the control panel for overall control and handling interactive editing, and (5) the output file dialog and output file writers.

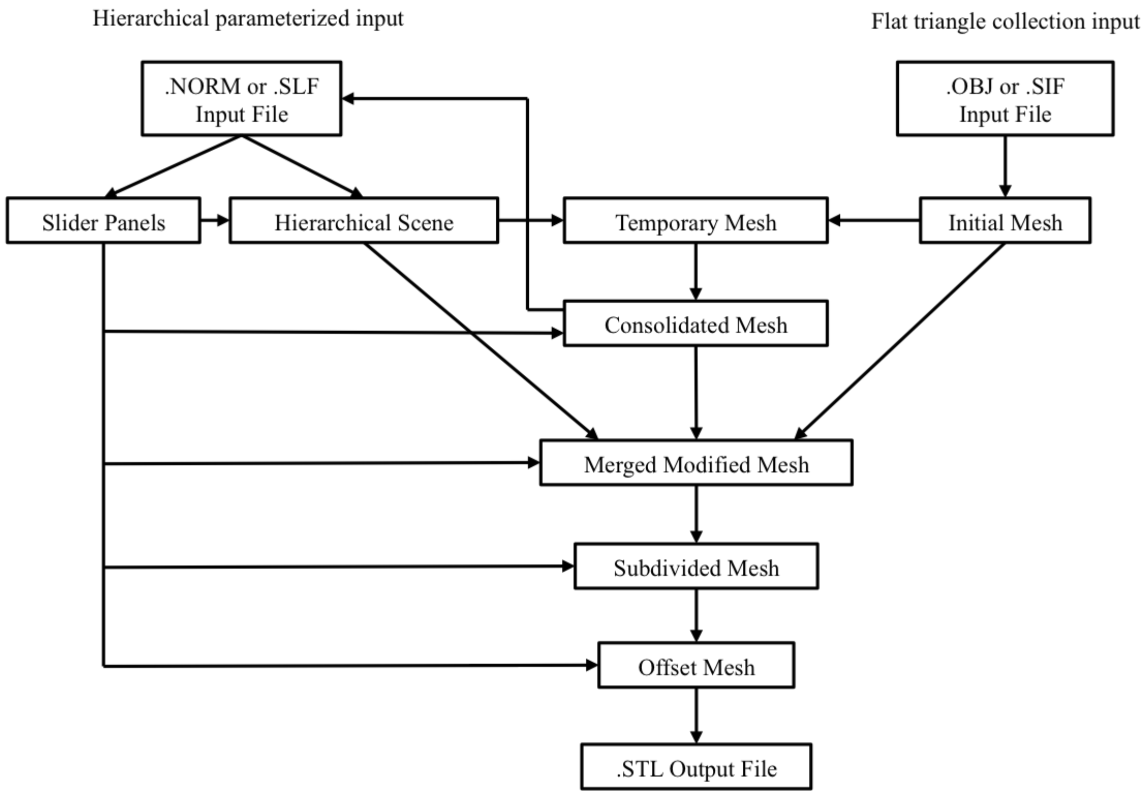


***Figure 8:*** *A general workflow in the program.*

The general workflow of this program is shown in Figure 8. Firstly, input may come in the form of a higher-level text file containing some specified parameterization, such as an .SLF or .NOME file. The input may also come from some flat geometry descriptions in .OBJ or .SIF format. The parsers are described in Section 7.1. Then, the user can interactively add/delete polygons by mouse and keyboard controls in the OpenGL canvas to create *temporay meshes*. The user can choose to accept this temporay change and add it to a *consolidated mesh*; or s/he can choose to forego the changes, and clear the *temporay mesh* and make a different set of modifications. The current design captured in the *consoidated mesh* as well as the current values of parameters can be saved as a new .NOME file, so that users can continue making design modifications from this saved state. Meshes from the hierachical scene and consolidated mesh can be merged into a mesh that contains only 2-manifolds. Subdivision and offsetting are then applied to this *merged mesh*. The final offset surface can then be saved as an .STL file.

## *7.1 Input Geometry File Parser*

An .NOME file or an .SLF file is used as the primary input of this program. The subset of SLIDE geometry that NOME can currently handle comprises point, polyline, face, object, mesh, instance and group. In an .NOME file, the user specifies the parameters for generating the sliders. The parameters are then packed into one or more parameter banks. Each parameter bank will be shown in a slider panel at run time. The user then specifies the definitions of vertices, meshes, and groups using the expressions of the above parameters. A hierarchical scene is constructed by instantiation and transformation of various pieces of geometries. If any instance is parameterized, we will store the expressions so that they can be evaluated at run time using the latest values of the parameters. For every parameter, we traverse down the hierarchical graph structure to find and store a list of the instances that it controls. Therefore, instead of re-evaluating every instance, we only re-evaluate the instances that are controlled by this parameter at run time. A parser for the complete .SLF language is under-development.

We also allow the user to start work from a flat geometry file produced by other programs (i.e., .OBJ and .SIF file). However, without definitions of parameters, descriptions of instance names, and how instances form the scene, these flat input files can produce only a non-parametric initial mesh. The parser first reads in all vertices and their positions. Then it reads in every polygon face, defined by a subset of these vertices. The vertices and faces are added to the mesh vertex list and face list respectively, with their list indices set as their identifiers. Since an .SIF input file contains only triangle faces, we can combine two adjacent triangles into a quadrilateral whenever possible to reduce the number of extraordinary points in the Catmull-Clark subdivision.

The program also handles exceptions caused by illegal constructs in the input files. Two major errors are concerned, (1) any syntax error in the input geometry file, and (2) any structural errors. Syntax errors include but are not limited to insufficient parameters in generating objects, unclosed parentheses, duplicated names of parameters or instances, numbers that can't be parsed, and tokens that can't be recognized. An example of a structual error is a non-manifold edge in the same mesh. This does not comply with winged-data structure and it cannot be subdivided at run time. When the user tries to create a non-manifold by making three faces share the same edge, the non-manifold exception is raised and an error message will be shown to the user. In the .SLIDE parser, a not yet implemented portion of the SLIDE language will raise an exception to tell the user that those constructs are not yet available.

## *7.2 Control Panel*

A control panel helps the user to choose from different options in the general workflow of this program. It comprises five subpanels: (1) operation selection panel, (2) viewing and editing panel, (3) subdivision panel, (4) offset panel, and (5) color panel.

The operation selection panel enables the users to select what they want to perform via a pull down menu. In our program, we have four main working operations: (1) interactive editing , (2) merging meshes, (3) subdivision, and (4) offsetting.

In the viewing and editing panels, the user can toggle between the vertex selection mode, border selection mode and polygon selection mode. A merge button serves to initiate the merger of all meshes in the hierarchical scene into a proper 2-manifold mesh. During this automatic merge, boundary edges from two different meshes with two coinciding end vertices will be merged. Otherwise, if the merge operation is skipped, the subdivided mesh will be done on the meshes individually (Fig. 5b).

The subdivision and offseting panels help the user select the level of subdivion and set the offset value, respectively. The color selection panel enables the user to choose (1) the default color for groups and meshes, (2) the color for the selected instances, (3) the color for temporary mesh, and (4) the color for consolidated mesh.

## *7.3 OpenGL Canvas and Interactive Editing*

An interactive graphical user interfaces enables the user to view and edit meshes in the current scene. The rendering task in our program is handled by OpenGL, which offers the functionality to draw 3D geometries and project them onto the camera plane.

### *7.3.1 Viewer: The Arcball Interface and Keyboard Control*

In order to let the user view the mesh from different directions, an *Arcball Interface* [20] has been implemented. The user can rotate the mesh by holding the left mouse button and moving the cursor on the screen. It calculates the axis and angle for an incremental rotation, and applies this rotation to the meshes on the screen. The camera distance is handled by zooming in and out by rolling the mouse wheel. Some other viewer modes are handled by keyboard controls, shown in Table 2.

**Table 2: Keyboard controls**

| Keys | Actions |
|---|---|
| I | Zoom in |
| O | Zoom out |
| W | Toggle between polygon mode and wire mode |
| S | Toggle between smooth and flat shading mode |
| X | Toggle between back face elimination mode |

*7.3.2 Face, Vertex and Border Selection*

Mouse selections in the graphical user interface are implemented to let the users interactively edit the mesh at run time. Our program provides three different selection modes: face selection, vertex selection, and border selection. The user can toggle between these selections modes in the control panel. We use the selection buffer from OpenGL to select a face or a vertex in that face in the graphical user interface. A selection indicator is generated for every vertex and face to show if it is currently selected.

In the face selection mode, when the user clicks with a mouse on the screen, a selection ray is generated and OpenGL will return a list of faces hit by this selection ray. The face with the nearest z-buffer value for the selection ray is then identified, marked as selected and visually highlighted. Clicking on this face again will cancel its selection.

The vertex selection mode is an extension of face selection. From the vertices of all faces hit by the selection ray, the vertex closest to the cursor is selected and its selection indicator is set to be true. Similary to face selection mode, clicking on a selected vertex again will cancel its selection.

The border selection mode allows users to select a group of border vertices at once. But in this mode, the user can only select vertices on the mesh borders. Every border in the scene is represented as a list of vertices in a loop. The border selection mode comprises two types of selections: whole border selection and partial border selection. In the whole border selection, when a border vertex is selected, all the other vertices on the same border will also be selected. The partial border selection requires three vertex selections: the first and second vertex selections to specify the two end vertices of the border, and a third vertex indicates which half part of the border loop should be selected. If the chosen part contains only two vertices, the third vertex selection is not needed. The user can cancel this selection by clicking on any vertex on the selected border segement.

*7.3.3 Interactive Editing*

*Adding Polygons*

When a list of vertices has been selected, the user can create a polygon formed by these vertices sequentially. This newly created polygon will be added to a *temporary mesh*. If the user is satisfied with the modifications in the *temporary mesh*, they can be added to a *consolidated mesh*, which can be saved in a new .NOME file. Otherwise, the user can clear the modifications in the *temporary mesh* and add different polygons. In this adding process, a user can add the zipper faces one by one between two or more meshes. Compared to automatic zippering, meshes are zippered based on the user's own judgment.

*Automatic Zippering of Mesh Borders*

If two sets of mesh border vertices are selected, they can be used as the input of a automatic zippering between the two borders. The automatic zipper program tends to minimize the lengths of edges between the two meshes and balance the skewness of vertices on the borders. When zippering partial borders, the program works greedily from the two ends of the two border segements to generate quadrilateral or triangle zipper faces. Otherwise, when zippering whole borders, it starts the work from the two matching points, which are vertices selected by the users in the whole border selection mode. Our program favors quadrilaterals over triangles, because triangles will generate extraordinary points in Catmull Clark subdivision. So a penalty is added to every triangle created. A larger triangle penalty tends to make fewer triangles than a small triangle penalty. Figure 9 and Figure 10 show two example for zippering two partial borders given different triangle penalties. Figure 9b and Figure 10a may look more preferable from the user's perspective

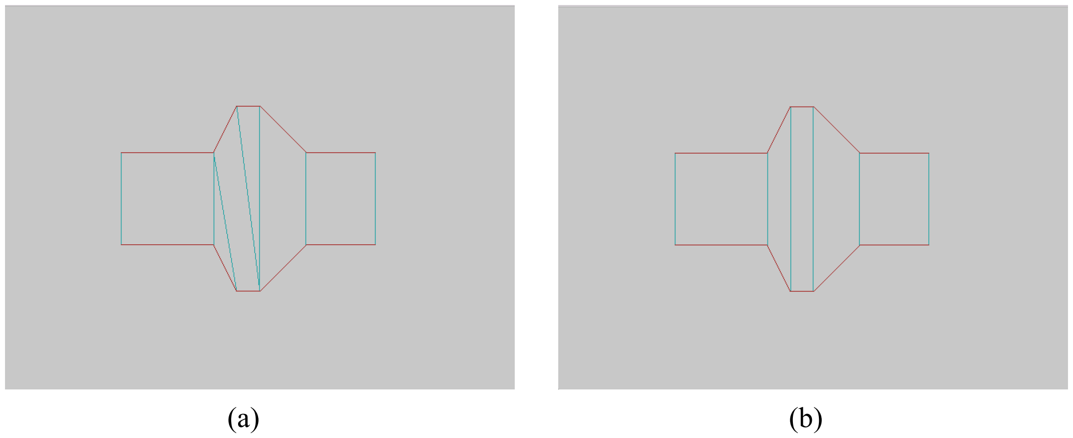


**Figure 9:** *Zippering two line segments with (a) triangle penalties = 1.3 (b) triangle penalties = 1.5.*

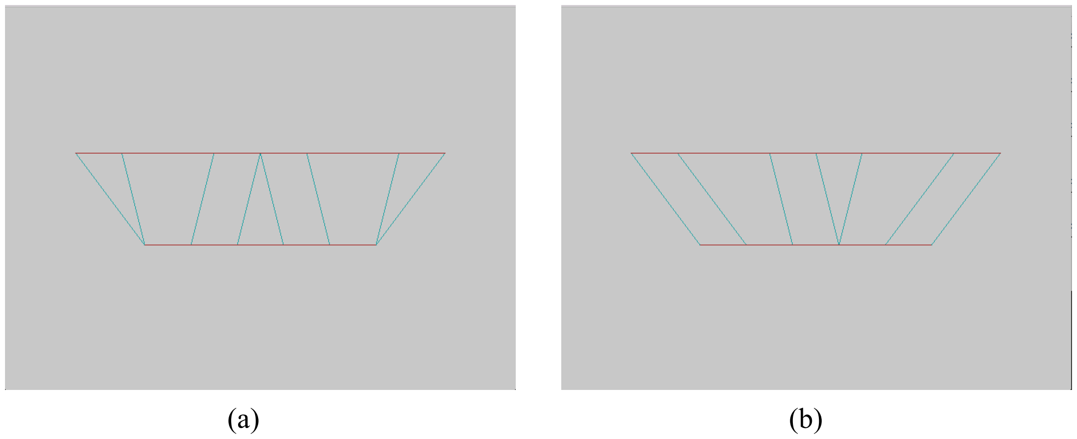


**Figure 10:** *Zippering two line segments with (a) triangle penalties = 1.0 (b) triangle penalties = 1.5.*

*Deleting Polygons*

When a polygon face is selected, we can delete it from the current scene. If two adjacent faces are both deleted, the shared edge will also be deleted. Similarly, when all edges around a vertex are deleted, the vertex will be removed from the mesh.

### *7.4 Slider Panels and the Parametric Updates*

A slider is created for every parameter defined in the input .NOME file or .SLF file, and a slider panel is created for every bank of packed parameters. The value change in a slider will cause the update of all instances affected by the corresponding parameter. This change will propagate through all representations and show its effect on the current view.

Users should note that a change of a parameter might cause an automatically merged mesh to split. This change will then propagate to the subdivision and offset meshes. So the user will see splits in those meshes as well.

### *7.5 Output*

An intermediate result of this program, including the current values of the sliders and the consolidated can be saved to a new .NOME file. So users can re-open this saved file to continue their work from the saved state.

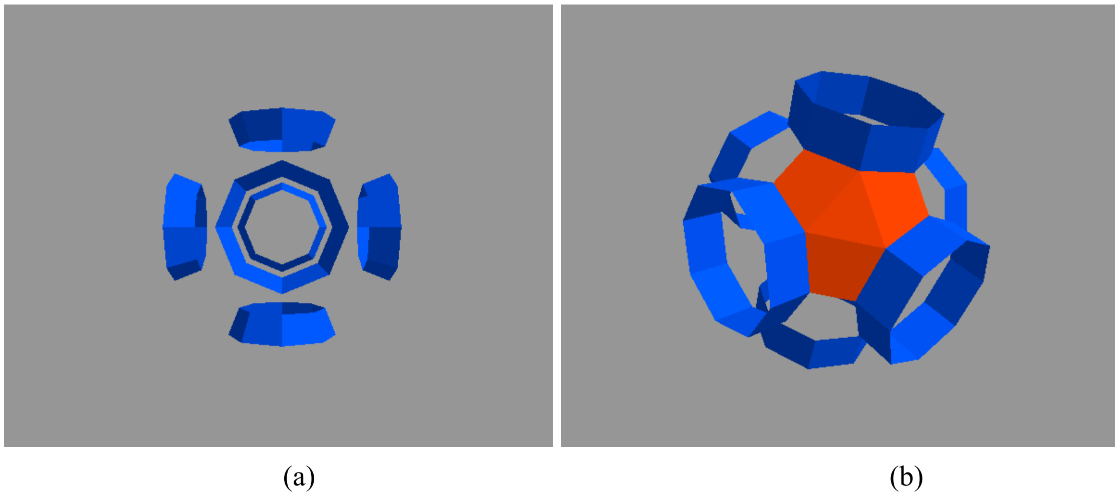
The final offset mesh can be saved as a geometry file in the .STL (STereoLithography) format. It can be displayed by other geometry file viewers, such as Microsoft 3D Builder and Preview program in Mac OS. It can also be passed to prototyping software like QuickSlice and can be used for 3D printing.

## 8. Test Cases

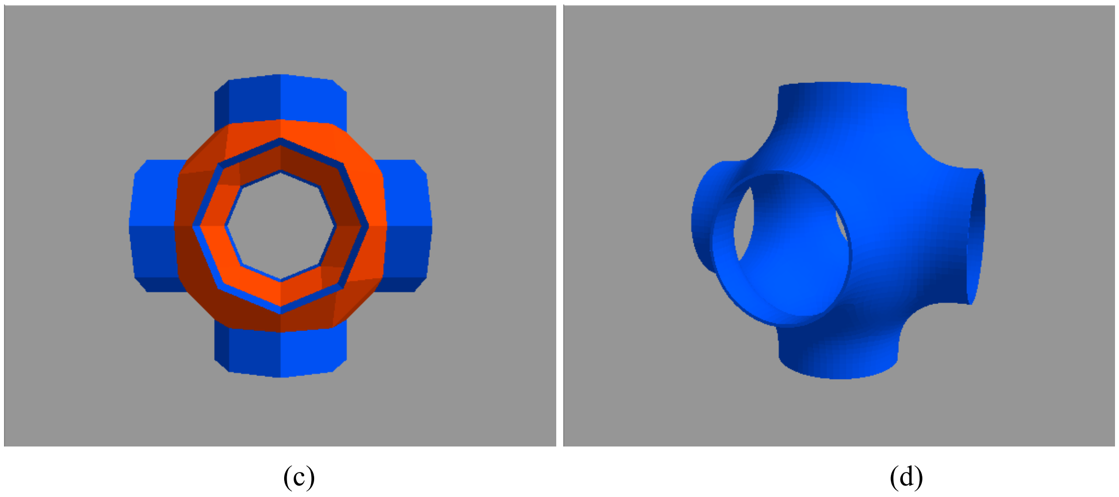
Several design examples were used to test the functionalities of the NOME system and to find out how the user interface can be improved. This section describes some of the preliminary tests made by Professor Séquin and by the author. Examples of some input geometry files are attached in Appendix A.

### *8.1 Funnel Cube*

The "Funnel Cube" is a cube structure made by connecting six funnels together to create a shape reminiscent of the periodic unit module in Schoen's surface [17]. We use this example to show the functionalities to (1) add polygons in the interactive editing mode, (2) save the consolidated mesh, and (3) instantiate and transform the saved consolidated mesh.



(a) (b)



(c) (d)

***Figure 11:*** *The process of generating the "Funnel Cube"*

In the input .NOME file, a funnel is specified by setting the four parameters defined in Section 6.3. It is then instantiated six times and transformed to the proper positions on the 6 faces of a cube by the user-specified transformations in six instances calls. The hierarchical scene constructed from the initial .NOME file is shown in Figure 11a. Using the sliders, we then adjust the parameters for the funnels and their transformation to bring the inner rims of the funnel close to one another so that they can be zippered together, but not too close so that sharp creases would result. Four polygons are then added below one corner of the bounding box cube, by selecting and clicking vertices from the top, front and left funnels. This is followed by a consolidation operation, as shown in Figure 11b. This consoliated mesh and the new parameter values are then saved to a new .NOME file. With a text editor, the instance call to the saved consolidated mesh is replicated eight times and the respective transformations are adjusted to place one instance at all 8 corners. Figure 11c shows the hierchical scene after reading in the modified .NOME file. Now we can merge all meshes in the scene and use this merged mesh for subdivision and offsetting. The final result is shown in Figure 11d.

### *8.2 Klein Bottle*

A Klein bottle is a single-sided surface without borders. It can be formed by passing one end of a tube through the side of the tube and joining it to the other end, like an inverted sock [18]. We use this example to show the functionalities of (1) the generated "tunnels", (2) zippering mesh borders in the interactive editing mode, (3) the subdivision and offsetting for a single-sided surface.

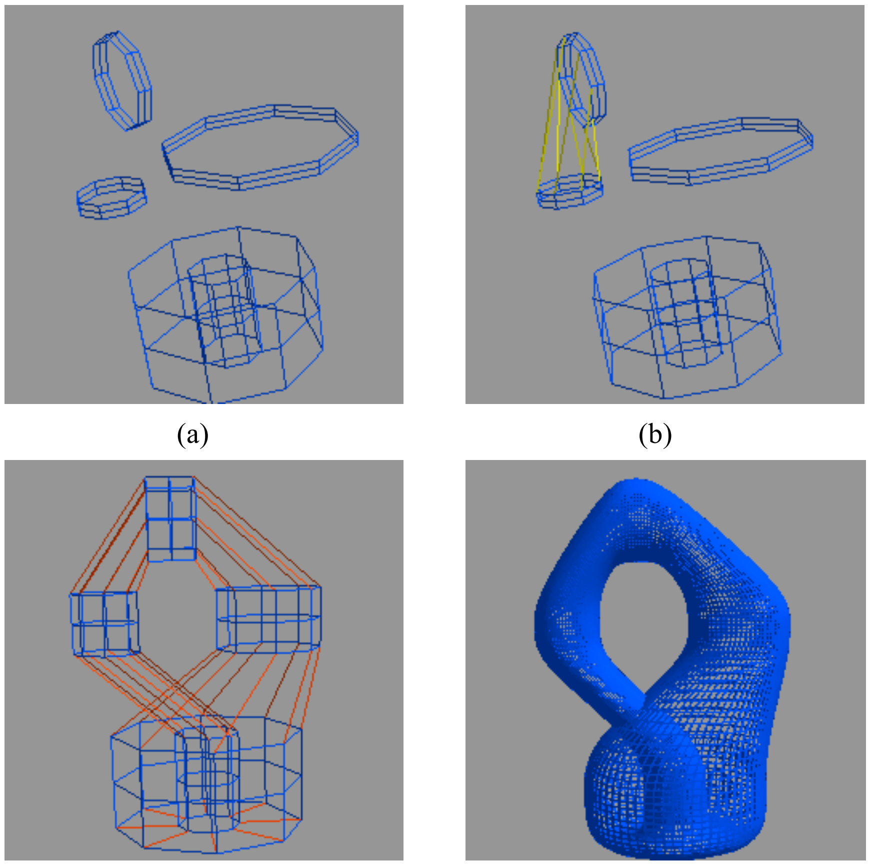

(a) (b) (c) (d)

***Figure 12:*** *The process of generating the "Klein Bottle"*

The input .NOME file contains five symmetrical tunnels roughly sized and positioned in the scene, as shown in Figure 12a. Tunnel radius, width and placement are fine-tuned by the user. Then a pair of adjacent rims on two neighboring tunnels is selected for zippering. The user can select the complete tunnel borders by clicking vertices on the borders. Every first selected vertex for a border will be used as the matching point for zippering. Figure 12b shows a first zippered connection between two tunnel rims. After zippering the borders of all five tunnels, we obtain a crude mesh of a Klein bottle (Fig. 12c). Subdivision and offsetting are then applied on the merged mesh. The final result is shown in Figure 12d.

### *8.3 Zero*

"Zero" is a sculpture by Charles Perry, as shown in Figure 13a [14]. It is a thickened single-sided surface standing in the x-z plane, with a "zero" shaped hole in a slightly slanted x-y plane. In this example, we will show how to reconstruct "Zero" by taking the input of two regular *n*-gons generated by SLIDE program and zippering the borders of the two ribbons.

The initial mesh generated after reading in the .SIF file is shown in Figure 13b. The outer ribbon is a ring in the x-z plane and has 24 segments. The O-shaped opening is a ribbon with 32 segments with a total twist of 720 degrees (Fig. 13b). Four new polygons are first added to connect the border segments with maximal ±y values of the inner ribbon. Then, a set of zippering polygons between the outside border of the remaining inner ribbon segments and the inside border of the outer ribbon are added. The mesh after consolidation is shown in Figure 13c. We then merge the initial mesh and the consolidated mesh into a merged mesh. The final result after subdivision and offsetting is shown in Figure 13d. An .STL file of this shape is sent to Fused Deposition Modeling (FDM) machine to produce a small maquette in PLA plastic (Fig. 14).

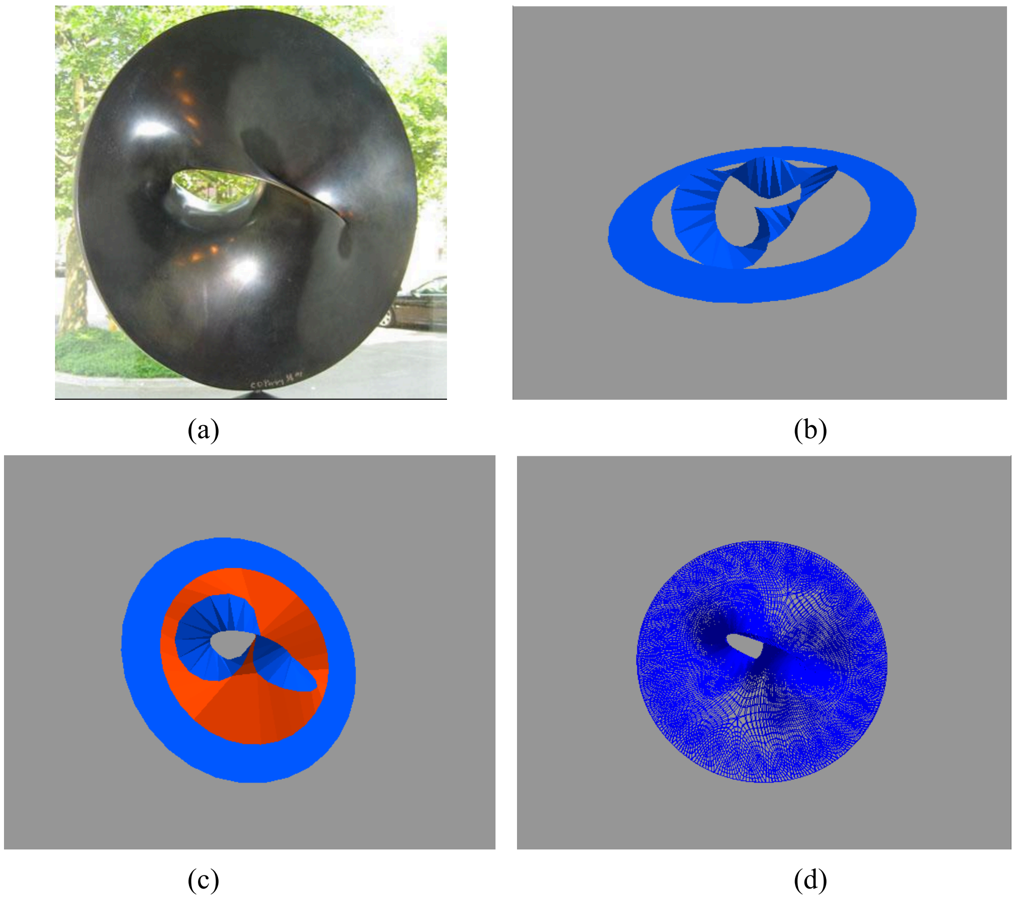


***Figure 13:*** *The process to reconstruct Charles Perry's sculpture "Zero"*

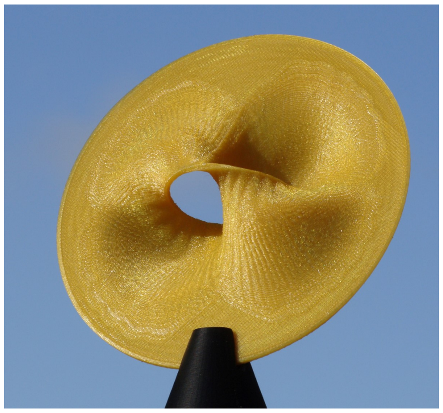


***Figure 14:*** *The re-constructed "Zero" (Photo: C. H. Séquin)*

### *8.4 Eva Hild's Snow Sculpture*

As a more challenging test for the new system (currently still in progress), we try to capture the complex sculpture that Eva Hild created at the International Snowsculpting Championships in 2011 [9]. This exercise will test all the functionalities of the NOME program, specifically, the 'funnel' and 'tunnel' generators, the adding and deleting of individual polygon faces, the assignment of user-defined colors to instances and groups of meshes, as well as the consolidation and merging of the interactively edited geometry. Eva Hild's snow sculpture is a double-sided thickened 2-mainfold with a single rim-line border (shown in Fig. 15).

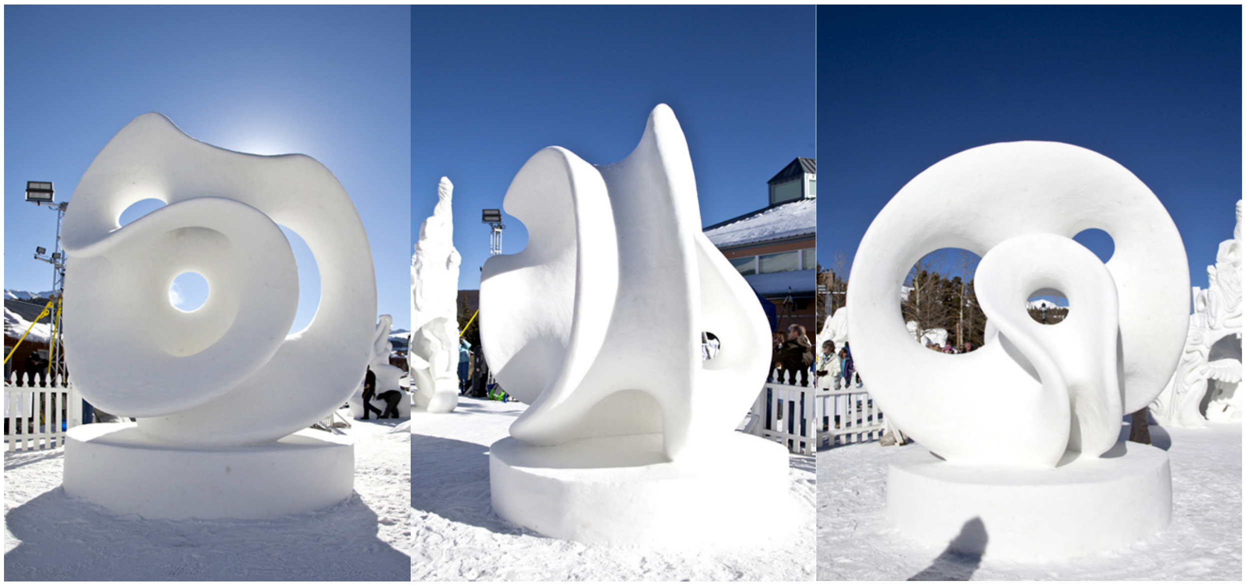


***Figure 15:*** *Eva Hild's snow sculpture from different angles [9].*

After a detailed observation, we decide to model every hole in the sculpture with one tunnel and model the border rim line with a combination of funnels. A proper placement of the input funnels and tunnels is shown in Figure 16a. It then requires a fairly large amount of work to merge the funnels and tunnels into the final sculpture. We will not only add polygons but also delete polygons in this process. For example, in order to connect the cyan tunnel, red tunnel and green funnel in Figure 16b, we need delete the top two polygons of the red tunnel to make it C-shaped. This can be done by selecting the two faces and deleting them from the scene. The final result after deletion is shown in Figure 16c. The current work in progress in shown in Figure 16d.

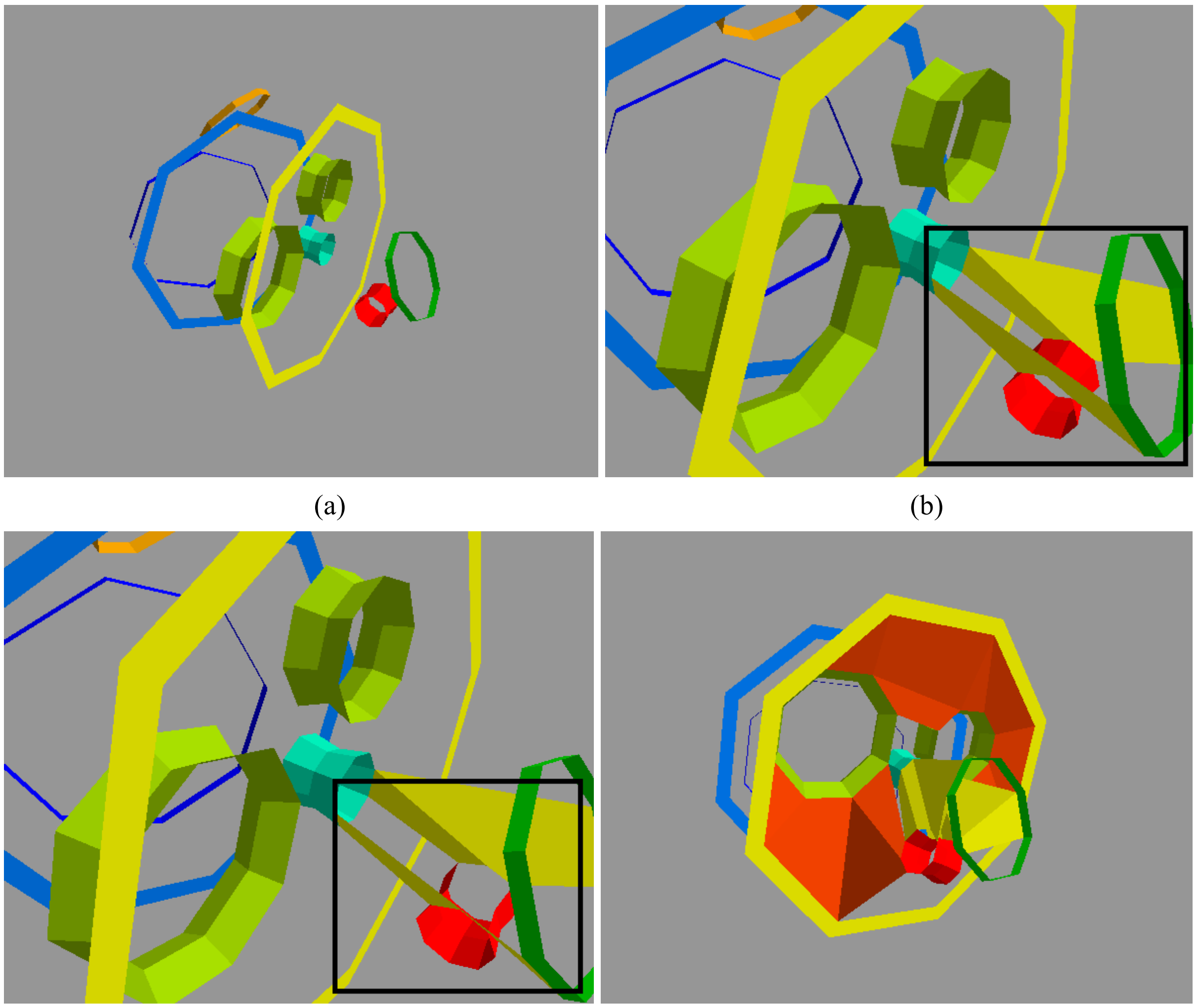

(a) (b) (c) (d)

***Figure 16:*** *Some initial stages to reconstruct Eva Hild's snow sculpture.*

## 9. Conclusions and Future Works

Inspired by the work outlined in a talk by Professor C. H. Séquin [19], and motivated by the functional limitations of SLIDE [22], an interactive program has been developed, that enables the reconstruction of some of the 2-manifold sculptures by Charles Perry or by Eva Hild. The two main contributions of this project are the implementations of mesh operations for non-orientable surface, and a graphical user interface that supports interactive editing and refinement of the generated shapes.

After the test on a more complicated example (i.e., the Eva Hild's snow sculpture), we find the NOME program functions quite well in general. Compared to the original SLIDE program, it saves the users time to locate and mark vertices or faces in the scene; helps to merge mesh boundaries by adding polygons or automatic zippering; helps to remove unwanted or redundant polygons; and enables the subdivision and offsetting for single-sided surfaces.

The following parts of the NOME program are currently under-development: (1) Introducing some additional constructs into the NOME language and its parser: individual faces and user-defined general meshes. (2) a complete .SLF file parser to construct a hierarchical scene with direct input of a SLIDE program (by Beren Oguz); and (3) a simple "rim ribbon" generator that allows to specify the azimuth angle and the twist of this ribbon (by Beren Oguz).

The author suggests the following two future enhancements of the NOME program. (1) Introduce the B-spline curves and the complete progressive sweep constructs from the original SLIDE program. (2) Implement a different instance selection function, because the selection buffer from OpenGL will be omitted in the future. This can potentially help to speed up the selection with binary space partition (BSP). It can also enable the user to select points that do not belong to any face in the scene.

## Acknowledgement

I would like to thank my advisor Professor Carlo H. Séquin for his great instruction and guidance during my M.S. study. I would also like to thank Professor Alexei A. Efros for his comments on this technical report. I would like to thank my former advisor Professor Iris D. Tommelein for her support and help when I decided to extend my research field into computer science. I would like to thank Beren Oguz for many helpful discussions and his interest to continue work in this project. I would also like to thank La Shana Porlaris for her support when I was working as a graduate student instructor. Last but not the least, I would like to thank my parents for their deep love and their support for my study.

## References

[1] B. Baumgart, A polyhedron representation for computer vision, AFIPS Proc., pp 589-596, 1975.
[2] E. Catmull and J. Clark, *Recursively generated B-spline surfaces on arbitrary topological meshes,* Computer-Aided Design, Vol. 10, pp. 350-355, 1978.
[3] T. DeRose et al., *Subdivision surfaces in character animation*, SIGGRAPH '98 Proc., pp 85-94, 1998.
[4] D. P. Dobkin and M. J. Laszlo, Primitives for the manipulation of three-dimensional subdivisions, SCG '87 Proc., pp 86-99, 1987.
[5] Eastman. *HalfEdge Data Structure*, 1982. – http://groups.csail.mit.edu/graphics/classes/6.838/S98/meetings/m4/IV.HalfEdge.html
[6] G. K. Francis and J. R. Weeks, *Conway's zip proof*, Amer. Math. Monthly 106 (1999) pp 393–399.
[7] Guibas and Stolfi, *Primitives for the Manipulation of General Subdivisions and the Computation of Voronoi*, ACM Transactions on Graphics (TOG), Vol 4 Iss. 2, pp 74-123, April 1985.
[8] E. Hild, Homepage. – http://evahild.com/#
[9] E. Hild, Snowsculpting in Breckenridge. - http://evahild.com/?page_id=466
[10] H. Hoppe et al., *Piecewise smooth surface reconstruction*, Computer Graphics, 28(3), pp295–302, July 1994.
[11] OBJ format, *Wavefront .obj file,* – http://www.martinreddy.net/gfx/3d/OBJ.spec
[12] OpenGL, OpenGL homepage, – https://www.opengl.org/
[13] C. Perry, *Topological sculpture* – http://www.charlesperry.com/sculpture/style/topological/
[14] C. Perry, *Zero* – http://www.charlesperry.com/sculpture/zero
[15] U. Pinkall, Regular homotopy classes of immersed surfaces, Topology 24 (1985) pp 421–434.
[16] Qt, Qt homepage, – http://www.qt.io/
[17] A. H. Schoen, Triply-periodic minimal surfaces, – http://schoengeometry.com/e-tpms.html
[18] C. H. Séquin, *From Moebius Bands to Klein-Knottles*. Bridges Conf. Proc., pp 93-102, Towson, July 2012.
[19] C. H. Séquin, *2-Manifolds*. Bridges Conf. Proc., pp 17-26, Baltimore, July 29-August 2, 2015.
[20] K. Shoemake, *ARCBALL: A User Interface for Specifying Three-Dimensional Orientation Using a Mouse*, – https://www.talisman.org/~erlkonig/misc/shoemake92-arcball.pdf
[21] SIF format, – https://www.cs.berkeley.edu/~ug/slide/docs/slide/spec/spec_frame_packages.shtml#parselib
[22] J. Smith, *SLIDE design environment*. (2003). – http://www.cs.berkeley.edu/~ug/slide/
[23] F. Tampieri, a Newell's method for computing the plane equation of a polygon, Graphics Gems III, pp 231-232, 1992.
[24] Tcl, *Tcl Developer Site*, – https://www.tcl.tk/

## Appendix A: The Input Geometry Files

### 1. Input .NOME for Section 8.1 (`funnel_cube.nome`)

```
# All parameters:
bank fp
    set     n          8       3       10       1
    set     ro       0.4     0.1      1.0     0.1
    set     ratio   -0.2    -0.5      0.5     0.1
    set     h        0.2     0.1      1.5     0.1
    set     z       -0.8    -1.5     -0.1     0.1
endbank

# Parameterized generator:
funnel fun ({expr $fp_n} {expr $fp_ro} {expr $fp_ratio} {expr $fp_h}) endfunnel

# 6 placments of a funnel:
group cubeMadeByFunnel
    instance fun1 fun translate (0 0 {expr $fp_z}) endinstance
    instance fun2 fun translate (0 0 {expr $fp_z}) rotate (1 0 0) (180) endinstance
    instance fun3 fun translate (0 0 {expr $fp_z}) rotate (1 0 0) (90) endinstance
    instance fun4 fun translate (0 0 {expr $fp_z}) rotate (1 0 0) (-90) endinstance
    instance fun5 fun translate (0 0 {expr $fp_z}) rotate (0 1 0) (90) endinstance
    instance fun6 fun translate (0 0 {expr $fp_z}) rotate (0 1 0) (-90) endinstance
endgroup

# Root of the scene hierarchy:
instance cf cubeMadeByFunnel endinstance
```

### 2. Modification of the saved .NOME file for Section 8.1 (`funnel_cube_consolidated.nome`)

```
# Adjusted parameters:
bank fp
    set     n       8       3       10      1
    set     ro      0.5     0.1     1.0     0.1
    set     ratio   0       -0.5    0.5     0.1
    set     h       0.3     0.1     1.5     0.1
    set     z       -1      -1.5    -0.1    0.1
endbank

# Parameterized generator:
funnel fun ({expr $fp_n} {expr $fp_ro} {expr $fp_ratio} {expr $fp_h}) endfunnel

# 6 placments of a funnel:
group cubeMadeByFunnel
    instance fun1 fun translate (0 0 {expr $fp_z}) endinstance
    instance fun2 fun translate (0 0 {expr $fp_z}) rotate (1 0 0) (180) endinstance
    instance fun3 fun translate (0 0 {expr $fp_z}) rotate (1 0 0) (90) endinstance
    instance fun4 fun translate (0 0 {expr $fp_z}) rotate (1 0 0) (-90) endinstance
    instance fun5 fun translate (0 0 {expr $fp_z}) rotate (0 1 0) (90) endinstance
    instance fun6 fun translate (0 0 {expr $fp_z}) rotate (0 1 0) (-90) endinstance
endgroup

# Root of the scene hierarchy:
instance cf cubeMadeByFunnel endinstance

# Saved consolidated mesh.
mesh m1
    face f1 (fun3_hc3 fun2_hc5 fun2_hc6 fun3_hc2) endface
    face f2 (fun3_hc3 fun5_hc3 fun2_hc5) endface
    face f3 (fun3_hc4 fun5_hc2 fun5_hc3 fun3_hc3) endface
    face f4 (fun5_hc3 fun5_hc4 fun2_hc4 fun2_hc5) endface
endmesh

# The followings are the instantiations and transformations for the consolidate mesh
# added in text for by the designer.
instance tp0 m1 endinstance
instance tp1 m1 rotate (0 0 1) (90) endinstance
instance tp2 m1 rotate (0 0 1) (180) endinstance
instance tp3 m1 rotate (0 0 1) (270) endinstance
instance tp4 m1 rotate (0 1 0) (-90) endinstance
instance tp5 m1 rotate (0 1 0) (-180) endinstance
instance tp6 m1 rotate (0 0 1) (90) rotate (0 1 0) (-90) endinstance
instance tp7 m1 rotate (0 0 1) (90) rotate (0 1 0) (-180) endinstance
# All top level instances (shown in bold) will be rendered.
```

## 3. Input .NOME file for Section 8.2 (`Klein_bottle.nome`)

```
#The parameters for five tunnels.
bank tp1
    set     n          8        3      10       1
    set     ro         1      0.1     1.0     0.1
    set     ratio      0     -0.5     0.5     0.1
    set     h        1.5      0.1     1.5     0.1
endbank

bank tp2
    set     n          8       3      10       1
    set     ro         1     0.1     1.0     0.1
    set     ratio      0    -0.5     0.5     0.1
    set     h        0.2     0.1     1.5     0.1
endbank
```

```
bank tp3
    set     n          8       3      10       1
    set     ro         2     0.1     2.0     0.1
    set     ratio      0    -0.5     0.5     0.1
    set     h        0.2     0.1     1.5     0.1
    set     x         -2      -4       2     0.1
    set     z       8      5   10    0.5
endbank

bank tp4
    set     n          8       3      10       1
    set     ro         3     0.1     3.0     0.1
    set     ratio      0    -0.5     0.5     0.1
    set     h        0.2     0.1     1.5     0.1
endbank

bank tp5
    set     n          8       3      10       1
    set     ro         3     0.1     3.0     0.1
    set     ratio      0    -0.5     0.5     0.1
    set     h        1.5     0.1     1.5     0.1
endbank

# Parameterized generators:
tunnel tun1 ({expr $tp1_n} {expr $tp1_ro} {expr $tp1_ratio} {expr $tp1_h}) endfunnel
tunnel tun2 ({expr $tp2_n} {expr $tp2_ro} {expr $tp2_ratio} {expr $tp2_h}) endfunnel
tunnel tun3 ({expr $tp3_n} {expr $tp3_ro} {expr $tp3_ratio} {expr $tp3_h}) endfunnel
tunnel tun4 ({expr $tp4_n} {expr $tp4_ro} {expr $tp4_ratio} {expr $tp4_h}) endfunnel
tunnel tun5 ({expr $tp5_n} {expr $tp5_ro} {expr $tp5_ratio} {expr $tp5_h}) endfunnel

# Placements of 5 tunnels:
group kleinbottle
    instance t1 tun1  endinstance
    instance t2 tun2  translate (-3 0 5) endinstance
    instance t3 tun3   rotate (0 1 0) (90) translate ({expr $tp3_x} 0 {expr $tp3_z})
endinstance
    instance t4 tun4  translate (2 0 5) endinstance
    instance t5 tun5  endinstance
endgroup

# Root of the scene hierarchy:
instance kb kleinbottle endinstance
```

## 4. Input .SIF file for section 8.3. (Example of `zero.sif` generated by SLIDE program)

```
(SIF_SFF 2 0
  (units inches)
  (solid
    (shell
      (*
        ERROR: Unclosed shell: unmatched edges = 64
        (loop 33 32 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42
41 40 39 38 37 36 35 34)
        (loop 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7
6 5 4 3 2 1 0)
      *)
      (vertices 64
        (v -0.010542 0.053071 0.000000)
        (v -0.020695 0.049963 0.000000)
        (v -0.030072 0.044981 0.000000)
        ...//More vertices here.
        (v -0.021791 0.032474 0.000000)
        (v -0.014955 0.036105 0.000000)
        (v -0.007554 0.038371 0.000000)
      )
      (triangles 64
        (t 0 32 31)
        (t 0 63 32)
        (t 1 63 0)
        ...//More triangles here.
        (t 30 33 34)
        (t 31 33 30)
        (t 32 33 31)
      )
    )
    (shell
      (*
        ERROR: Unclosed shell: unmatched edges = 64
        (loop 33 32 63 62 61 60 59 58 57 56 55 54 53 52 51 50 49 48 47 46 45 44 43 42
41 40 39 38 37 36 35 34)
        (loop 31 30 29 28 27 26 25 24 23 22 21 20 19 18 17 16 15 14 13 12 11 10 9 8 7
6 5 4 3 2 1 0)
      *)
      (vertices 64
        (v 0.017689 0.000000 -0.003513)
        (v 0.016653 0.000000 -0.006897)
        (v 0.014992 0.000000 -0.010023)
        ...//More vertices here.
        (v 0.019778 0.013858 -0.013192)
        (v 0.026452 0.010607 -0.010957)
        (v 0.031270 0.005740 -0.006275)
      )
      (triangles 64
        (t 0 32 31)
        (t 0 63 32)
        (t 1 63 0)
        ...//More triangles here.
        (t 30 33 34)
        (t 31 33 30)
        (t 32 33 31)
      )
    )
  )
)
```