{ Draft for EECS Tech Report: 3/7/2016}

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. An interactive UI 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) [ref] 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 [ref] 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 [ref] or a variation thereof that is also two-sided (Fig.0a), it is not possible to create the model shown in Figure 0b, which is a single-sided, non-orientable variation of this sculpture. The singled-sided model shown in Figure 0b is created with the geometry kernel described in this report.

(a) (b)

**Figure 0:** Two modifications of Perry’s “Tetra” sculpture: (a) two-sided, (b) single-sided.

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 [Reference: 2-manifold sculpture]. To emulate these sculptures, this geometry kernel processes general 2-manifolds, i.e., thin surfaces with borders. The Surface Classification Theorem [ref] 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 [obj] 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 with the same control as in the original SLIDE file, that accepts a relevant subset of SLIDE commands. This enables the user to change the shape of the geometry by moving sliders. A special SIF [SIF] 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 is read into the program, the user is able to interactively view and modify the geometry in the GUI to add polygons and/or merge mesh borders for the initial mesh.

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 transfer to other CAD environments for further processing or can be output as a .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 Sections 3 and 4 show how these data structures are used in hierarchical scene construction, and in the subdivision and offsetting processes. Section 5 and 6 discusses the processing of the input and output files, parameterizations of the model and the graphical user interface to create new polygon mesh given the initial mesh. Section 7 illustrates how the new geometry kernel has been tested with emulations of sculptures by Charles Perry [ref] and by Eva Hild [ref], as well as with some unusual shape in the form of connected sums of Klein bottles.

1. 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 these elements. Several adjacency structures have developed in the past, including the winged-edge data structure (Baumgart, 1975), the half-edge data structure (Eastman, 1982), the QuadEdge Data structure (Guibas and Stolfi), and the FaceEdge Data Structure (Dobkin and Laszlo, 1987).

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 and keeps the gathering time for needed information for all mesh-processing operations within well-defined limits. Figure 1 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.

Fa

Fb

V1

V2

Ea1

Ea2

Eb1

Eb2

Next Edge Pointer

Next Edge Pointer

Next Edge Pointer

Next Edge Pointer

Face Pointer

Face Pointer

Vertex Pointer

Vertex Pointer

Fa

Fb

V1

V2

HEa1

HEa2

HEb1

HEb2

Vertex Pointer

Vertex Pointer

Face Pointer

Face Pointer

Next Edge Pointer

Next Edge Pointer

Previous Edge

Pointer

Previous Edge

Pointer

Sibling Pointers

(a) (b)

**Figure 1:** Linking in winged-edge (a) and half-edge (b) data structure

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 1 (a) and (b), primarily the sibling pointers can be eliminated. There are 8 pointers from an edge in winged-edge data structure and 10 in half-edge data structure. When doing traversals around a vertex, we need to cross one more link, between the two half-edges, in 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 Fa before Fb), and a flag on the edge to keep track of Möbius condition, winged-edge data structure can be made compatible to both single-sided and double-sided surfaces.

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

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

Section 2.1 describes the basic geometry classes built in this geometry kernel, including vertex, edge, face, and mesh; section 2.2 shows how to implement the geometry classes and construct an initial mesh; and section 2.3 introduces mesh face traversals and vertex traversals for future mesh operations like 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 Fa before Fb. It means for a boundary edge the pointer to Fb is always null. It allows a fast test to find boundary edges.

### Face

A face is defined as a polygon 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, and (2) a 3D vector for its face normal. It also has a pointer to one of its edges.

### Mesh

A mesh is a collection of polygon faces. The faces collectively form one or more 2-manifolds in the mesh. In this program, every mesh contains (1) a list of faces, (2) a list of vertices, (3) an auxiliary map from vertex to its conected edges in the mesh construction phase, and (4) the color of this surface.

The list of faces and vertices gives quick access to faces and vertices in the mesh. They also provide the basis for face traversals or vertices traversals for general mesh operations. In the construction phase, non-manifolds may exist in the partly constructed mesh. We leveraged an auxiliary map to help address this problem and to generate the final data structure (see section 2.2 for more details). Once all pointers are built for every edge, this auxiliary map can be removed.

## 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 a subset of the vertices above that forms 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 hash-table. 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. Mapping to the newly created edge from its end vertices are added to the auxiliary map for future references.

After all consecutive edges have been generated for one face, three types of pointers are created, i.e., (1) pointers from an edge to its adjacent edges in this face, (2) pointer from vertex to one of its connected edge, and (3) pointers from the 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.

At this point, all geometry instances associated with the current mesh have been created, and the links between instances are properly formed. The auxiliary map can be removed and mesh construction is complete. Now we have a good initial 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 mesh transformations, (e.g., an .SLF file). Through this mechanism, we are able to change the geometry interactively when running the program. Parameterization is addressed in section 6.1.

## 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 face normal. The traversal around vertices is used to build vertex points and compute a vertex normal. Examples of face traversals and vertex traversal are shown in Figure 2 (a) and (b).

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 if 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.

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 adjacent edge ending in same vertex, we can visit all adjacent edges and faces until 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 5.1. 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 of all vertices also visit every edge twice and it takes time linear to the total number of edges in the mesh.

F

E4

E1

E2

E3

Start!

V

E5

E1

E2

E3

Start!

E4

(a) (b)

**Figure 2:** Example of mesh traversals (a) around a face (b) around a vertex

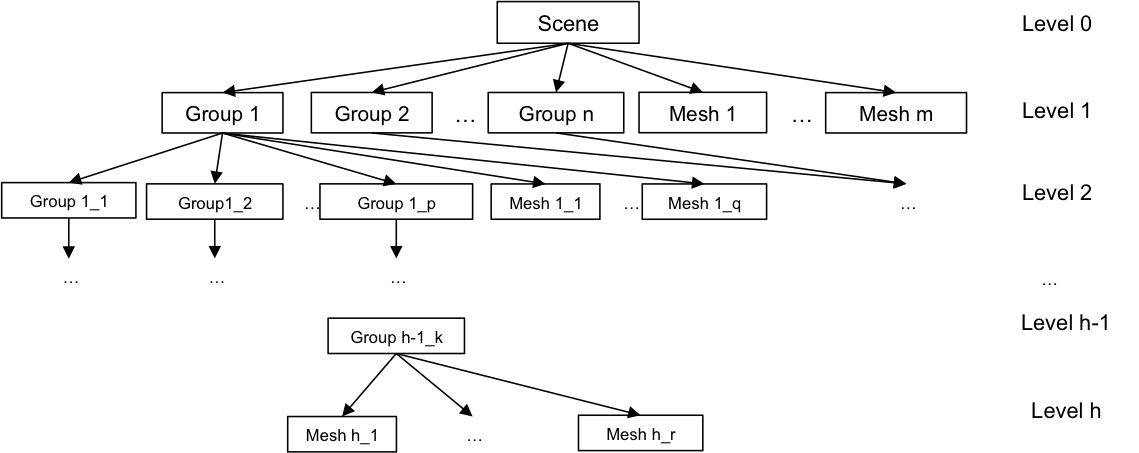
1. Hierarchical Scene Construction

For efficiency, complex scenes are structured hierarchically where identical piece 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 their positions. It is better to instantiate four copies of one wheel and translate them to the correct position than writing all four in the world coordinate separately.

We used a tree structure to represent the relationships of modular meshes in this program. Section 3.1 describe the implementation of this tree structure; section 3.2 - 3.4 shows the necessary mesh operations to realize hierarchical scene construction, including (1) mesh instantiation, (2) mesh 3D transformation, and (3) mesh boundaries merge.

## 3.1 Tree Structures for Hierarchical Scenes

In this program, a tree structure is created to organize the relations between meshes at different levels. The general construct to build this tree is the class “Group”. It may contain a list of meshes and of other groups, as shown in Figure 3. Every leaf in this tree is an instance of a mesh, which is described in an input geometry. The leave meshes are transformed as specified and are merged into higher level groups and finally into the whole scene.



**Figure 3:** A tree structure for hierarchical scene

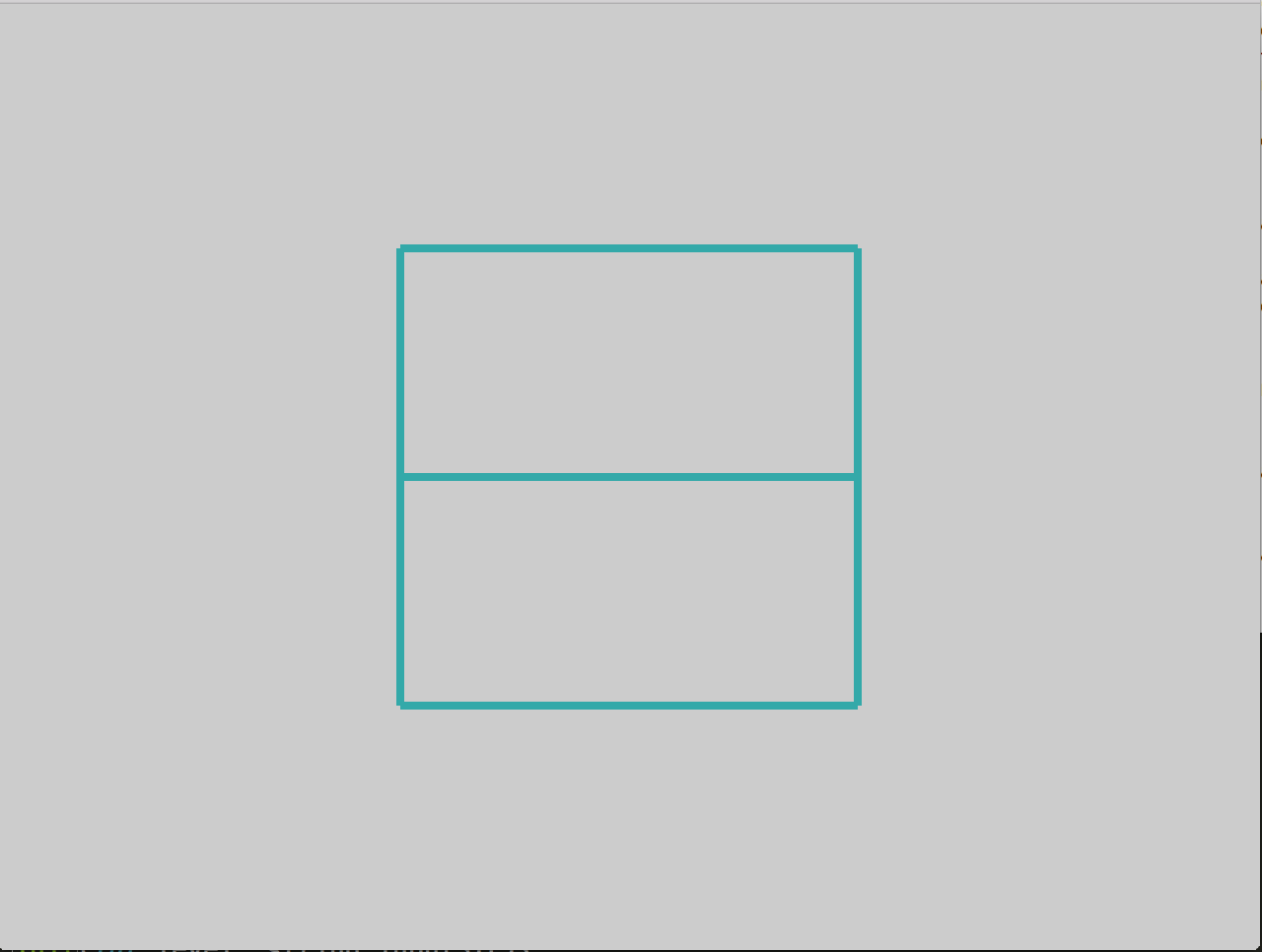
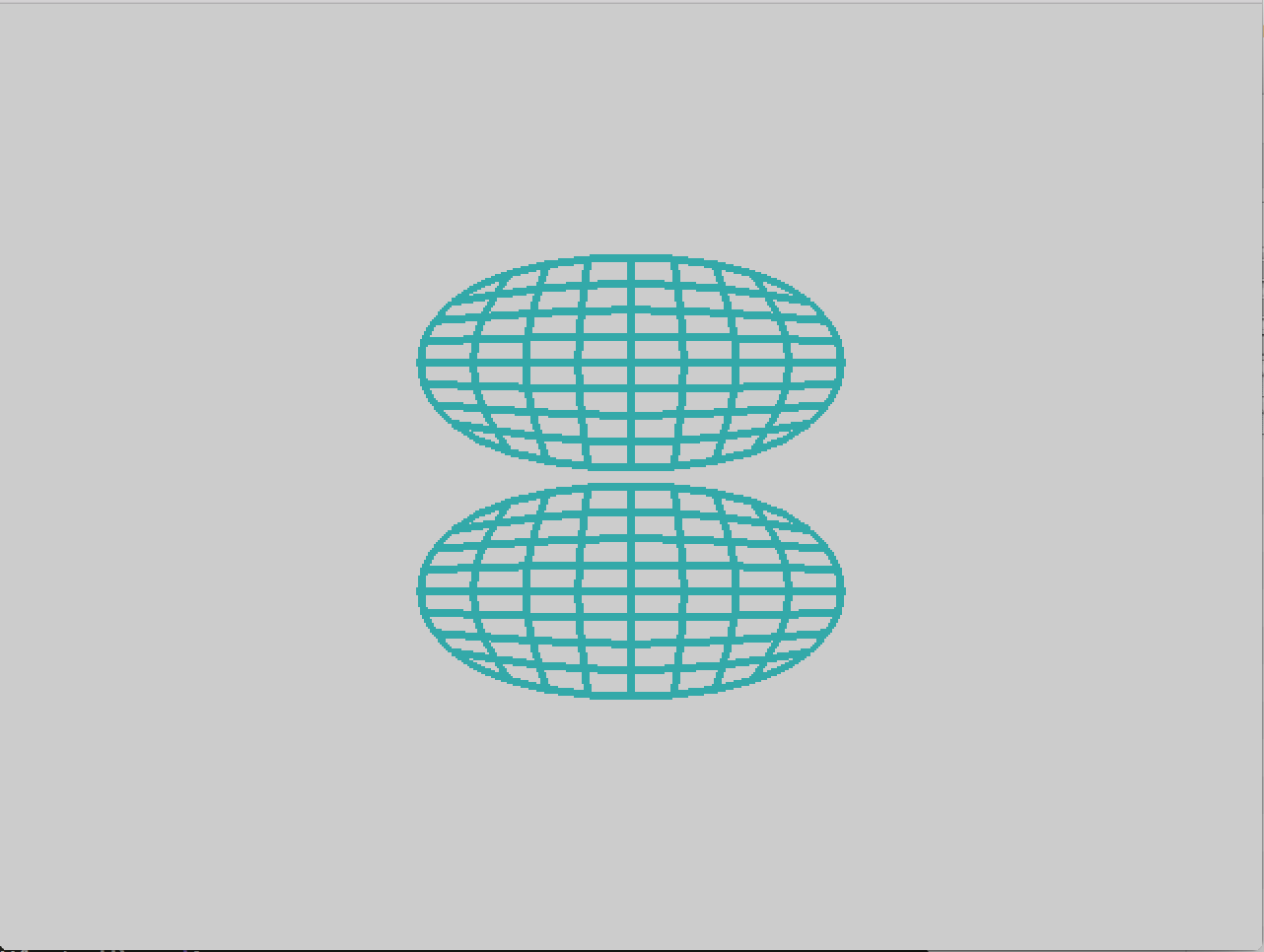
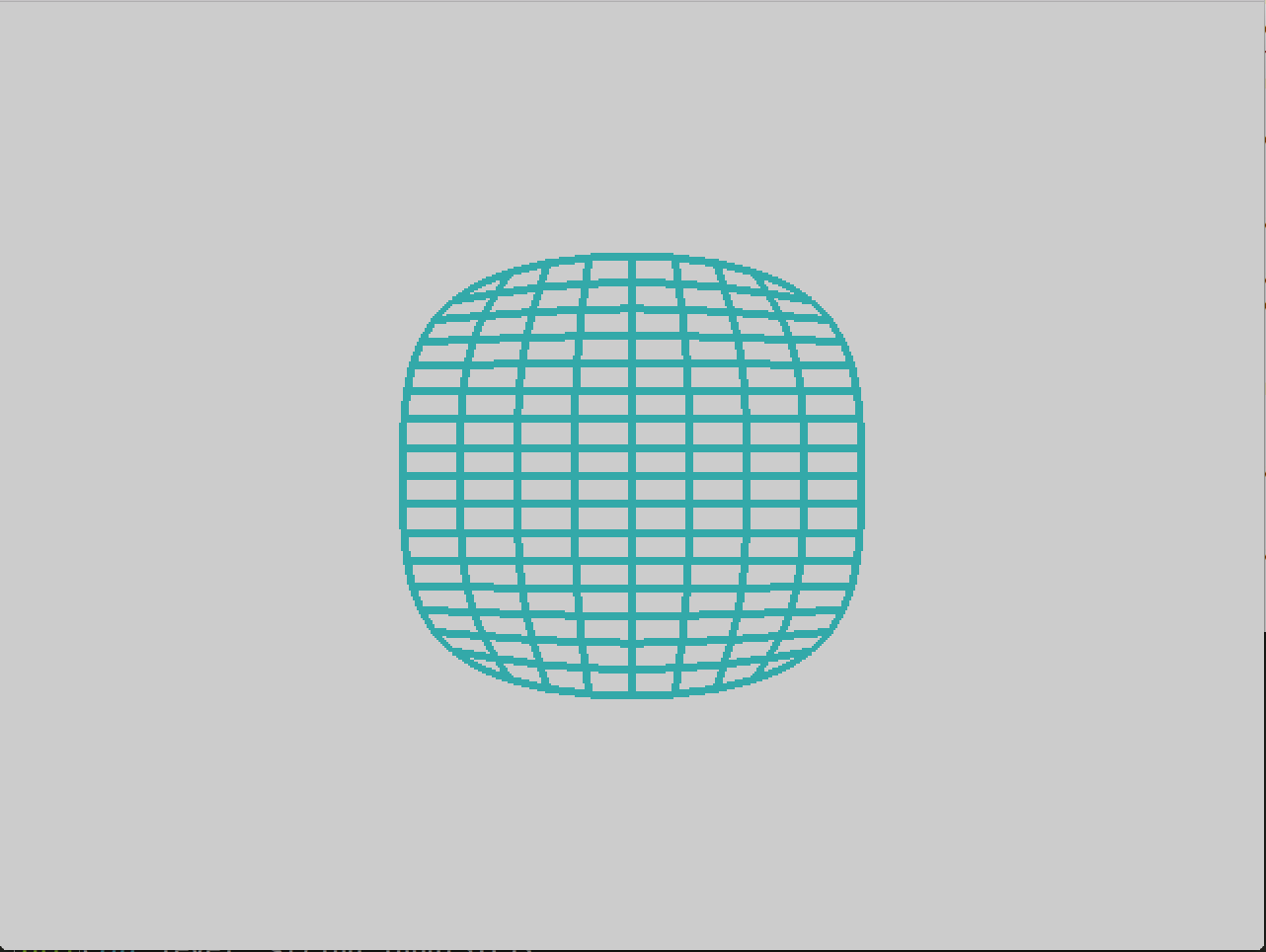
## 3.2 Mesh Instantiation

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

This program supports general linear transformation given a homogenous transformation matrix. Meanwhile, it provides special transformations including non-uniform scaling, rotation around an arbitrary axis, translation, and mirroring to an arbitrary plane.

## 3.3 Mesh boundary merges

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

**Figure 4:** Example of Catmull-Clark subdivision before and after mesh boundary merges

(a) Initial meshes (b) subdivision before merge (c) subdivision after merge.

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 if vertices should be automatically merged if it is possible by converting a pair of boundary edges from two meshes into one inner regular edge or Möbius edge in the merged mesh.

During the automatic merging process, a non-2-manifold structure may occur before all merges are done. For example, two triangles share just one coinciding vertex without any overlapping edges. In this case, we shall not merge this coinciding vertex. But a third triangle can be merged 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 merges.

(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 user-defined merging tolerance). Only boundary edges can match because matching of inner edges would generate non-2-manifold. This boundary edges matching may create Möbius edges in the merged mesh if their adjacent faces are with different orientations.

(2) Merge vertices from the matching boundary edges.

The matching boundary edges offer an instruction on how to merge vertices from the two meshes. Every pair of matching boundary edges indicates two pairs of vertices that can be merged. Suppose we are merging mesh 1 and mesh 2, we can 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 to state that these 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 or Perry sculptures, we also implemented some forms of “manual” merging (zipping) of meshes whose boundaries are not in coincidence. The user indicates how specified segments of two border curves 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 done in an interactive graphical user interface (see Section 6).

1. 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. This can be realized by Catmull-Clark subdivision. It is a process to turn a coarse polyhedral model into a smooth 2-manifold with some connectivity. It is called recursively on a polygon mesh to make the surface finer and smoother. 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 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 identifiers set as its list index. 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, its face point is defined as the centroid of all vertices of this face. If we denote the face point of face as , and vertices on this face as , the face point is

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 to 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. [ref] and define sharpness of an edge as infinitely sharp or smooth. Following Hooper et al [ref] and DeRose et al [ref], 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 as , and the two face points of face adjacent as and , the edge point for an infinitely sharp edge is

and the edge point for a smooth edge is

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 vertex point depends on the number of adjacent sharp edges . If , the vertex is defined as a corner vertex. The new vertex point keeps the same position. If , the vertex defined as a border vertex (or crease vertex). Its vertex point is the weighted average of vertices on the two adjacent sharp edges. If , the vertex is defined as a regular vertex (or inner crease). However, Catmull-Clark [ref] and DeRose et al. [ref] 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. However DeRose et al., used the new edge points of all adjacent edges instead of their midpoints.

If we denote the original vertex as , its vertex point as , the number of adjacent faces as , the average for its adjacent faces points as , the average for its adjacent edge points as , the average for the midpoints of all adjacent edges as , the average for the midpoints of all adjacent sharp edges as , the equations to calculate the new vertex point are

If :

If :

If , in Catmull-Clark method:

and in DeRose et al. method:

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

And DeRose et al. gives

**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 , , , and , while DeRose puts more on the original vertex . We implemented both methods in this project and will let the user choose his 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 divided it into n quadrilateral faces, where n is the number of edges in the face. This is done by connecting the face points with all edge points and connecting the new vertex points with its adjacent edge points. Figure 7 shows the examples of sub-faces of triangle, quadrilateral, and pentagon respectively.

(a) (b) (c)

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

In the traversal around the face, for every edge , we will create four new edges. If we denote the edge point for as and the face points of its two adjacent faces asand , the four new edges are , , , and . Edges and inherit the and sharpness of . If is a Möbius edge, we will also mark and as Möbius edges. Meanwhile, , and are always smooth and not Möbius. The sub-faces will inherit the orientation of the face.

The level k + 1 mesh is then created after we traverse around every face in level k mesh. This running time is linear to number of edges in the level k + 1 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.

1. Mesh Offsetting

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 offset surfaces.

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) positive offset, where every vertex of M is translated by an offset value along the direction of its vertex normal; (2) negative offset, 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 won’t intersect with its negative offset. However, if M is a single-sided surface, the positive offset surface is connected with negative offset surface at the offset of Möbius edge from M. So special checks are requred for offseting on Möbius edges.

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

## 5.1 Compute 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 [ref] to calculate the surface normals for each polygon face,

Newell's method defines surface normal as the normalized average for cross products of all pairs of consecutive edges in the face. If a face has n edges, and we denote these edges as , ,…, , and , a vector normal for the face is:

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 normal by traversing around every vertex. If a vertex 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,

where is the jth adjacent face normal of and in the valence of the vertex.

However, when is adjacent to a Möbius edge, the orientations for some adjacent faces will change. 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 will be reverted. 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 faces of the vertex. Surfaces normal is added from faces that have the same orientation of the starting face, and are subtracted otherwise. The vertex normal calculated in this way is associated with the starting face in vertex traversal. It does not necessarily reflect the true “positive” direction of the vertex normal. For convenience we marked these vertices (i.e., those adjacent to at least one Möbius edge) with 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 can get the positive offset surface by instantiate the mesh and translate every vertex along its vertex normal with the chosen offset value. For negative offset surface, it’s the same except 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. The translation of vertices marked Möbius need special check because its vertex normal may not reflect the direction for its “positive” offset. Thus, we build the positive and negative offsets by traversal around every face. If a vertex is on Möbius edge, we check the dot product of its vertex normal and the face normal for 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 Mesh

The border mesh is the connection between positive and negative offset. We traverse around the boundary edges of the original mesh. For every boundary edge , we build a quadrilateral face with the positive offset and negative offsets for and . This creates the zip faces between positive offset mesh and negative offset surface. Similar to section 5.2, if a boundary vertex is marked Möbius, we need to 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 offset mesh after subdivision, the offset mesh itself is a two-sided polygonal mesh that can be subdivided. After its initial construction, the border surface 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 rim of the thickened slab.

1. Parameterization, User Interface and Workflow

Now with all classes and functions built for this geometry kernel, a user is ready to build 2-manifold mesh and hierarchical scene from an input geometry file and apply several mesh operations on it. A proper workflow and user interface is designed to help the user to create the initial model more conveniently in an interactive manner. This includes changing the positions of vertices, modifying mesh transformations in the hierarchical scene, adding polygon faces on top of the current vertices in the scene, choose a proper level of subdivision, or applying varied offset values for the offset mesh.

In this section, we are going to explain how these operations are implemented in our program and provide a general workflow on how to use this program. These operations are based on the concepts of parameterization (pre-defined modifications of user designed parameters) and interactive modifications in the GUI (changes on the fly). They are implemented with OpenGL, GLUT and Qt. Section 6.1 describes how a user can import parameters and its predefined range of values from a .SLF file [ref]; Section 6.2 shows several possible operations that are allowed in the program to view and change the mesh; Section 6.3 shows the output of the program; and Section 6.4 concludes with a general workflow for this program.

## 6.1 Parameterization and Text File Input

### 6.1.1 Parameterization

We follow the key idea of parameterization from SLIDE [ref] and allow the user to set a list of parameters in an input .SLF file. The .SLF file includes information about the initial values and possible ranges of these parameters. They are then specified on to calculate vertex positions, transformation matrices, and mesh operations.

At runtime, the program will create a corresponding slider for every parameter, to let the user control the parameter’s value. The change will be immediately reflected in the graphical user interface. So the user can make his design decisions with the best combination of parameter values.

To realize this parameterization, we used a map from the parameters to any instances that they influences. When any change of the parameter is made, we will recalculate the values of the impacted instances. The hierarchical scene is then reconstructed, and subdivision and offsetting are applied to the new scene again.

### 6.1.2 Text File Input

The major input of this program is a geometry file. It generates the initial mesh by defining vertices and polygons made by subsets of these vertices. Some geometry files, e.g. .SLF file, will also contain user defined parameters and the instructions for hierarchical scene construction. In our program, it includes a .SIF file parser and a .SLF file parser to handle both cases.

The output of the .SIF parser is a mesh. It first reads in all vertices and their positions. The vertices are then added to the mesh vertex list with identifiers set as its list index. Then it will read in every polygon face, defined by a subset of these vertices. During this initial mesh construction, we need to handle incorrect or illegal input files. Exceptions will be thrown when it happens. For example, it raises exception if three faces share the same edge. Because it creates a non-manifold edge that does not comply with winged-data structure and it can not be subdivided in the run time.

When two vertices from the input .SIF file coincide, we may want to merge them because they are actually the same vertex with different names. Otherwise, they will be treated as two different vertices on the boundary edges and this mesh will break apart in the subdivision. Meahwhile, the .SIF input files contains only triangle faces. To avoid creating extraordinary points in Catmull-Clark subdivision, we will combine two adjacent triangles into a quadrilateral whenever possible in the parser.

The output of .SLF parser is a hierarchical scene. In addition to reading in vertices and polygons, the parser will also read in user defined parameters, mesh transformation matrices and hierarchical scene structures from the file. It also contains tcl/tk blocks that defines group in the scene, which is handled sperately by the tcl/tk parser.

//More on how SLF parser works.

## 6.2 Interactive Graphical User Interface

### 6.2.1 Qt Widgets

In this program, all user interface widgets are created by Qt framework, including general buttons, dialogs, and windows, sliders, and OpenGL canvas. Any events trigger by the user, for example moving a slider, will activate a signal and call functions in the program. They together provide the basis for an interactive GUI, which contains both viewer and modifier for the initial scene constructed by the input file.

### 6.2.2 Viewer: Arcball interface and keyboard control

The rendering work in our program is handled by OpenGL, which offers functions to draw 3D polygons and project it to the camera plane. In order to let the user view the mesh from different angles, we implemented Arcball interface feature [ref]. The user can rotate the mesh by pressing the left button and moving cursor on the screen. The rotation axis and magnitude are calculated by the mouse movement.

Other viewing options, for example zooming in and out, are handled by keyboard controls. Table 1 shows actions that are handled by keyboard controls.

Table 1: Keyboard controls in the viewer

|  |  |
| --- | --- |
| **Keys** | **Action** |
| I | Zoom in |
| O | Zoom out |
| W | Switch between polygon mode and wire mode |
| S | Switch between smooth shading and Gouraud shading |
| Esc | Exit the program |

### 6.2.3 Modifier: Instance selection, adding, and zipping

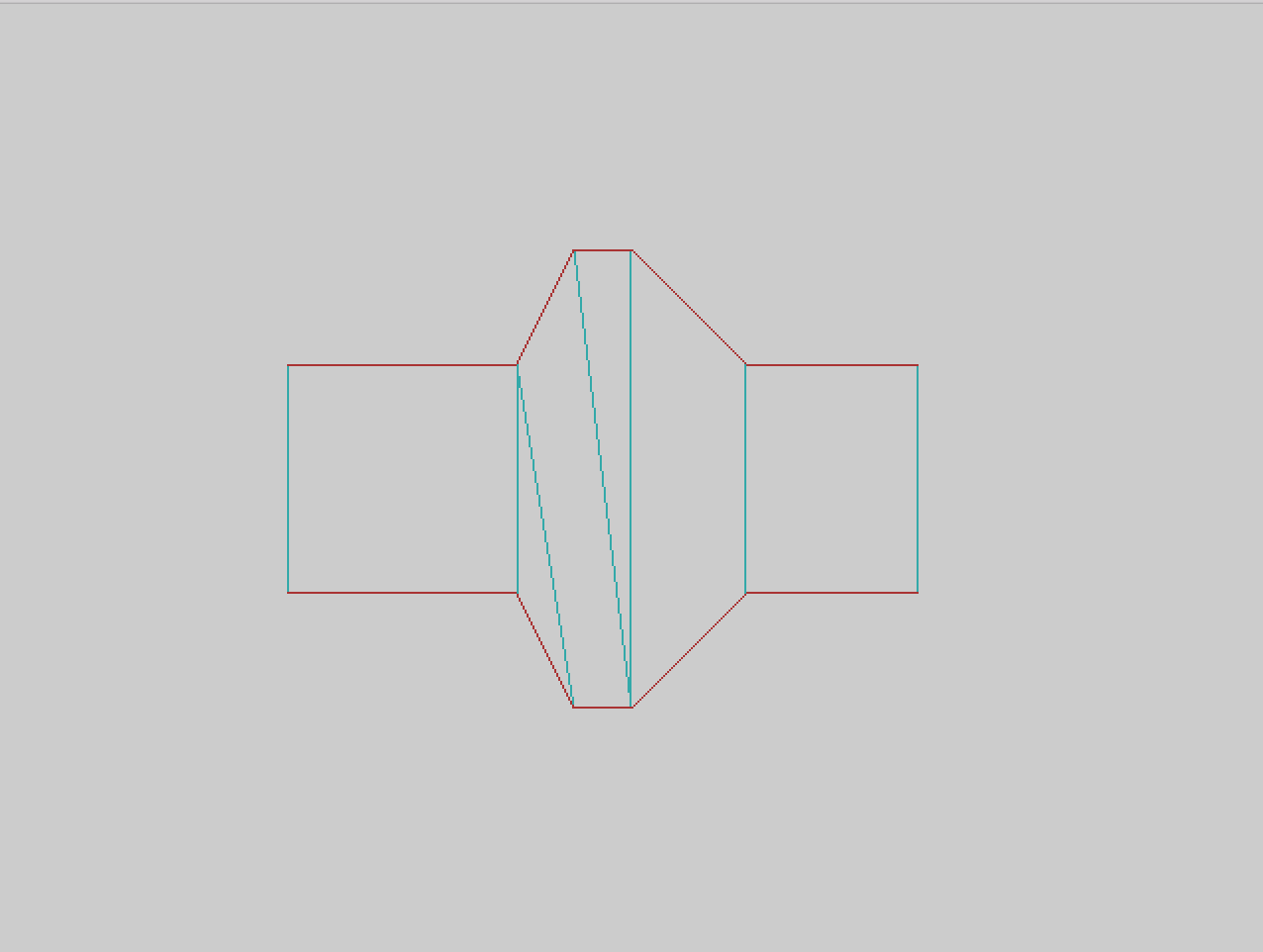
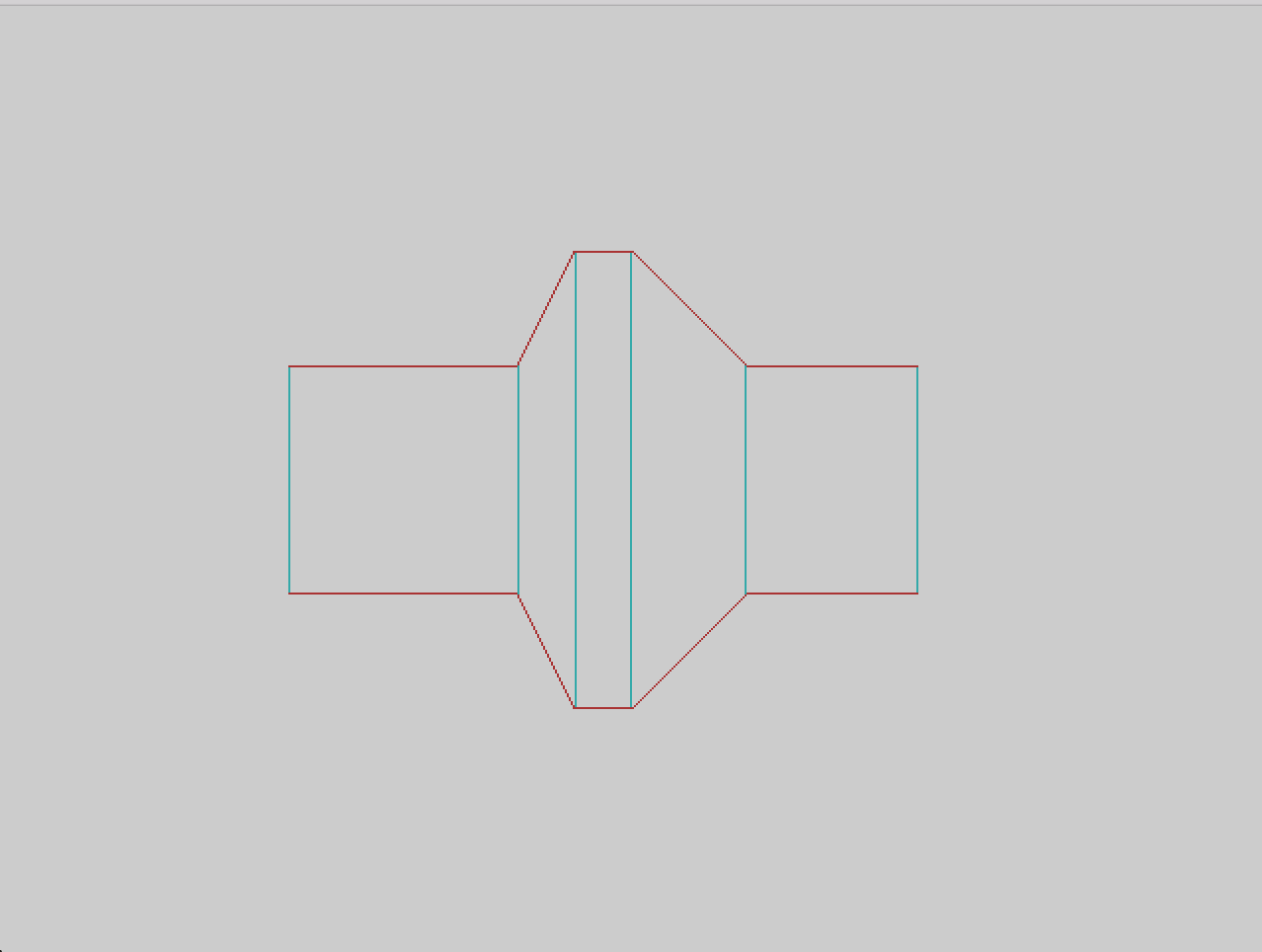
The user can add more polygon faces at the run time interactively in the GUI by the vertices defined from the input file. This requires instance selection in the GUI. Our program contains three different selection modes: vertex selection, border selection (line segment) and border selection (line loop).

We used the selection buffer from OpenGL to select instances in the GUI. For vertex selection, we generate a selection indicator for every vertex. When the user click a point with mouse on the screen, OpenGL will return a list of polygons hit by this ray. Then, we can calculate the nearest vertex from the selected point and set its selection indicator to be true. Clicking on a vertex already selected will cancel its selection.

Border selection extends vertex selection, but in this mode, the user can only select vertices on borders. Every border in the mesh is made of a loop of sequential vertices. In line loop border selection, when a boarder vertex is selected, all other vertices on the same border will also be selected. This selected point serves as the matching point in line loop border zipping. Line segment selection takes in selection of three vertices: first and second vertex selections to specify the two end vertices of the line segment and a third vertex to indicate which half part of the loop the user wants. If the line segment contains only two vertices, the third vertex selection can be ignored. The user can cancel this selection by clicking on any vertices on the selected border.

Instance selections provide two options for the user to interactively create faces and merge boundaries. The first choice is based on vertex selection. By clicking and selecting the boundary vertices from the two meshes, the user can create a polygon zip face. He/she can feely explore and add the zip face one by one until all selected boundary edges have been merged. This offers flexibilities and considerations of the user’s own judgments on how the meshes should be merged.

The second choice is based on border selection. The program will zip the two borders automatically for the partial borders or entire border selected by the user from one or two meshes. The automatic zipper program tends to minimize the lengths of edges between the two meshes and balance the skewness of vertices positions on the borders. If the two borders are partial borders, it will work greedily from the two ends of the lines to generate either quadrilaterals or triangles as zipper faces. If they are line loops, it will work from the two sides of a pair of matching points selected by the users. Our program is biased to favor quadrilaterals over triangles by adding penalties to every triangles created, because triangles will generate extraordinary points in Catmull Clark subdivision. A larger triangle penalty tends to make fewer triangles than a small triangle penalty. Figure 5 shows an example for zipping two line segments given different triangle penalties. Figure 5 (b) may look more preferable from the user’s point of view.

(a) (b)

**Figure 5:** Zipping two line segments with (a) triangle penalties = 1.3 (b) triangle penalties = 1.5

The newly created polygons, either added or zipped, will form a new temporary mesh. If the user is not satisfied with current modification, he can undo the change. The temporary mesh will be added to the scene when the user decides to finalize the change and click a button. This newly made mesh can be used for further modifications.

## 6.3 Output

The final output of our program can be saved as a geometry file with .STL (STereoLithography) format. It can be displayed by other geometry file viewers, such as Microsoft 3D Builder. It can also be passed to prototyping software like QuickSlice and 3D printing.

## 6.4 Workflow

Table 2 summaries all widgets currently designed in this program.

Table 2: UI Summary

| **Widgets** | **Functions** |
| --- | --- |
| Input dialog | Read in and parse .SLF and .SIF files |
| Parameter panels | Change user defined parameter values; one panel for each group of parameters stated in the .SLF file. |
| Control panel | Control the type and level of subdivision, generate subdivision; Control the offset value and generate offset mesh; Control the modification mode |
| Output dialog | Save .STL files |

The following is a general workflow of this program.

1. Import an .SLF file or .SIF file.

2. Adjust the user-defined parameters with sliders; view feedback in the GUI with mouse and keyboard control; further modifications with mouse control

3. Subdivision

4. Offsetting

5. Save the work to an .STL file.

1. Applications and Test Cases

Show some test cases here. Eva Hild & Charles Perry Sclupture.

// More

1. Conclusion and Future Works

// More…

# References

1. (Baumgart, 1975),
2. Catmull-Clark
3. DeRose et al.
4. (Dobkin and Laszlo, 1987).
5. (Eastman, 1982),
6. (Guibas and Stolfi)
7. E. Hild, Homepage. – [http://evahild.com/#](http://evahild.com/)
8. OBJ format
9. Perry: Tetra
10. U. Pinkall, *Regular homotopy classes of immersed surfaces*, Topology 24 (1985) pp 421–434.
11. C. H. Séquin, *From Moebius Bands to Klein-Knottles*. Bridges Conf. Proc., pp 93-102, Towson, July 2012.
12. C. H. Séquin, *2-Manifolds*. Bridges Conf. Proc., pp 17-26, Baltimore, July 29-August 2, 2015.
13. SIF format
14. J. Smith, SLIDE design environment. (2003). – <http://www.cs.berkeley.edu/~ug/slide/>
15. Tcl