Vlist and Ering:
Compact Data Structures for Simplicial 2-Complexes

A Thesis
Presented to
The Academic Faculty

by

Xueyun Zhu

In Partial Fulfillment
of the Requirements for the Degree
Master of Science in the
School of Electrical and Computer Engineering

Georgia Institute of Technology
December 2013

Vlist and Ering: Compact Data Structures for Simplicial 2-Complexes

Approved by:

Dr. Jarek Rossignac, Advisor
School of Interactive Computing
College of Computing
Georgia Institute of Technology

Dr. Monson H. Hayes
School of Electrical and Computer Engineering
Georgia Institute of Technology

Dr. Greg Turk
School of Interactive Computing
College of Computing
Georgia Institute of Technology

Dr. Ghassan AlRegib
School of Electrical and Computer Engineering
Georgia Institute of Technology

Date Approved: Nov 14, 2013

ACKNOWLEDGEMENTS

I would like to express my deepest appreciation and gratitude to my advisor, Dr. Jarek Rossignac, for his patient guidance and mentorship he provided all the way from when I first attended Georgia Tech as a MS student, through my completion of this degree. I would never have been able to finish it without his abundant help and support.

I would also like to thank my committee members, for the friendly guidance and thought-provoking suggestions.

Finally, I would like to thank my parents and my boyfriend for your moral support. You are always there for me encouraging me and cheering me up.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF SYMBOLS AND ABBREVIATIONS

V	number of vertices
T	number of triangles
S	number of sticks
C	number of edge-connected components
N	number of star-connecting vertices
A	number of triangles incident at non-manifold edges
M	number of loops incident to a vertex

ECT	Extended Corner Table
CAT	Constant Amortized Time
RAT	Random Access and Traversal
S2Cs	Simplicial 2-Complexes

SUMMARY

Various data structures have been proposed for representing the connectivity of manifold triangle meshes. For example, the Extended Corner Table (ECT) stores V+6T references, where V and T respectively denote the vertex and triangle counts. ECT supports Random Access and Traversal (RAT) operators at Constant Amortized Time (CAT) cost. We propose two novel variations of ECT that also support RAT operations at CAT cost, but can be used to represent and process Simplicial 2-Complexes (S2Cs), which may represent star-connecting, non-orientable, and non-manifold triangulations along with dangling edges, which we call sticks. Vlist stores V+3T+3S+3(C+S-N) references, where S denotes the stick count, C denotes the number of edge-connected components and N denotes the number of star-connecting vertices. Ering stores 6T+3S+3(C+S-N) references, but has two advantages over Vlist: the Ering implementation of the operators is faster and is purely topological (i.e., it does not perform geometric queries). Vlist and Ering representations have two principal advantages over previously proposed representations for simplicial complexes: (1) Lower storage cost, at least for meshes with significantly more triangles than sticks, and (2) explicit support of side-respecting traversal operators which each walks from a corner on the face of a triangle t across an edge or a vertex of t, to a corner on a faces of a triangle or to an end of a stick that share a vertex with t, and this without ever piercing through the surface of a triangle.

1. INTRODUCTION

3D models of human made materials, of physical or organic structures, or of human anatomies can comprise watertight shells, surface membranes, and attached chords, which we call sticks, that are defined as edges that do not bound a triangle. Those models can even be non-orientable. We propose two simple and compact data structures for representing the connectivity of a piecewise linear approximation of these models. We use the term simplicial 2-complex (abbreviated S2C) when referring to such models. An S2C is a connected simplicial complex with cells of dimension 0, 1, and 2. It may be used to represent non-orientable surfaces and combinations of surfaces and networks of one-dimensional (piecewise linear) curves in three dimensions. An example of an S2C is shown in Figure 1.

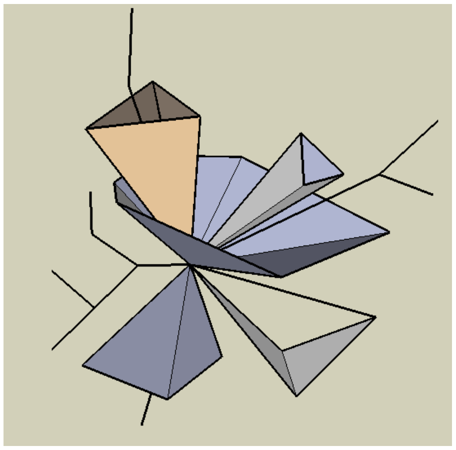


Figure 1: Example of a S2C with dangling edges (sticks) and a star connecting, i.e., several fans of incident triangles and several incident sticks that connect through v.

In this thesis, we provide two simple and compact data structures supporting time-efficient operators for accessing, traversing and editing S2C. One is called VList and stores V+3T+(3+k)S references, where V, T, and S respectively denote the count of vertices, triangles, and sticks and k denotes the number of sticks with both ends connected to triangles, as shown in Figure 5. The other one is called Ering and stores V+6T+ (3+k) S references. Both support Random Access and Traversal (RAT) operations with a space and time cost that is a function of the local valence, which is the number of cells incident upon a vertex or edge. Furthermore, both Vlist and Ering support editing (local connectivity changes) at constant cost, assuming that the maximum valence of a cell is negligible compared to the total number of cells.

Although Ering uses more storage then Vlist, it supports a faster implementation of the RAT operations than Vlist and guarantees a Constant Amortized Time (CAT) cost. Furthermore, Ering supports a purely topological implementation of these operators, while operators that work on Vlist use geometry to query the order of triangles around a non-manifold edge. Therefore we propose an extended set of side-respecting RAT operators that work on S2Cs and support walking on reachable triangle faces, piercing a triangle, and also jumping between triangles and sticks and from one stick to the next, without traversing across a triangle surface. Piercing a triangle can be visualized as going through the triangle surface from one side to the other.

Although we claim that, under for a certain class of simplicial complexes, Vlist and Ering are more compact than previously proposed representations, this is only true when we consider data structures that support RAT operators at CAT costs. Indeed, significantly more compact representations of connectivity may be achieved through compression. For example, Topological Surgery guarantees 3T bits, but typically achieves close to 2T bit connectivity storage [1]. EdgeBreaker [2] guarantees 2T bits, but in practice achieves about 1T bit for most meshes. Unfortunately, compressed formats cannot be used directly for mesh access and traversal. Formats that support partial decompression of desired mesh subsets offer a good compromise, but impose a performance overhead especially for decompression of each chunk.

Only a few prior publications deal with the transmission of simplicial complexes. For example, the Progressive Simplicial Complex (PSC) [3] and their variation [4] are extensions of a progressive formats for triangle meshes [5]. They start with a simplicial complex, coarsen it with edge collapses and vertex merges, and record the compressed form of the information necessary for performing the reverse of operations. A model with adaptive resolution can be reconstructed by starting from the coarse mesh and applying a subset of the vertex splits. In [3], storing the coarse mesh and the vertex splits information of a S2C with V vertices uses 3V*16+V(logV+7) bits. The 3V*16 bits are used for storing the vertex coordinates and Vlog(V+7) bits are used for storing the vertex split information. Still, as mentioned before, this compressed information only describes how to reconstruct the S2C. It doesn't support direct access to the cells or traversal operators directly. Hence, another data structure must be used to represent the decompressed model for processing.

Recently produced compact data structures for triangle meshes include SOT [6] which stores 3T references, SQuad [7], which stores about 2T references, LR [8], which stores about T references, and Zipper [9], which stores about .18T references with an average of 6 bits per triangle. These representations do not support local editing operations at constant time cost, but instead must be constructed for the entire mesh. The ESQ [10] variations of SQuad support local editing, including vertex insertion, mesh flip, and valence-3 vertex deletion, in constant time. Grouper [11] proposes a fixed file format/data structure for SQuad, interleaving geometry and connectivity information in a fixed format and supports streaming creation from a streamed input and both streamed and random access processing, even for high span meshes (meshes with large index differences between the triangles incident on the same vertex). Most of prior art on triangle mesh compression or on its compact representations deal with manifold or at least water tight meshes, for which there is no need to pierce through a surface, to jump to sticks, or to define side-respecting traversal operators.

A few prior publications describe data structures that deal with simplicial complexes. The prior art most related to our contribution offers a compact data structure for S2Cs [12]. In that data structure, each vertex v has two linked lists: (1) a linked list of references to every neighboring vertex w if edge (v, w) is a stick; and (2) a linked list of references to one triangle in each group of edge-connected component incident on v. Furthermore, for each triangle t, the indices to its three vertices v1, v2 and v3 are stored along with between 3 and 6 references to its adjacent triangles. These references define, for each non-manifold edge of this triangle, the next and previous triangle around the edge and, for each manifold edge, a reference to the only adjacent triangle. Figure 2 shows an instance of t1, t2 as the next and the previous neighbors of triangle t around the non-manifold edge (v,w).

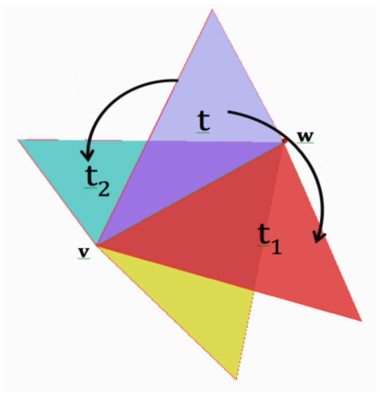


Figure 2: Non-manifold edge vw has 4 incident triangles. For triangle t, its adjacent triangle of edge (v, w) is t1 and t2.

Let V be the number of vertices, T be the number of triangles, S be the number of sticks, C be the number of star-connecting components that incident at non-manifold vertices and A be the number of triangles incident at non-manifold edges. The space used by the data structure of [2] is 4V+6T+6S+2C+A references plus 2V+3T bits. Furthermore, the traversal operators proposed in [2] do not distinguish between triangle piercing operators and side-respecting traversal. This distinction is important, in algorithms that identify the faces and sticks that bound a chamber, i.e., a connected 3D component of the complement of the S2C, such as a ventricle in an anatomical model of a heart, with 2D surfaces representing valves and 1D curves representing chords.

Thus, the Vlist and Ering representations proposed here and the definition and implementation of the associated operators provide an important improvement over these prior contributions to the representation and processing of S2Cs:

- Definition and implementation of a simple set of RAT operators that support side-respecting traversal, and
- More compact representation for S2Cs with a relatively low stick/triangle ratio

Vlist may be trivially constructed in linear time and space from an indexed format representation of the S2C, which for example lists the vertex coordinates and then list the triangles and sticks, each defined by the integer references to its vertices.

The implementation of the RAT operators on Vlist and of the construction of Ering from Vlist requires testing side-respecting accessibility between faces and sticks. This test is performed to decide whether they are locally reachable from one to another by walking over an edge or vertex, but not piercing through any surface. An important contribution reported here is an elegant formulation and a simple implementation of this test, which is related to the problem of classifying a point with respect to a vertex neighborhood [13] and uses parity ideas similar to those used for cell-to-cell occlusion tests [14].

2. FOUNDATIONS

To provide a broad context to our work, we list here various topological classes of simplicial complexes. In the following chapters, we will start explain our data structure with simple complex and then the extension to the complex ones. Also, we defined different entities associated with triangles and sticks. Those entities are used for accessing and traversing S2Cs.

2.1 Type of complexes

We classified several different types of complexes as below. Staring from simple components, we add handles, boundaries, non-manifold edges, vertices and sticks one by one. How to deal with simple meshes for clarity and extend then to S2Cs will be discussed in the data structure section.

1. **Simple complex**: A zero-genus orientable manifold without boundary, i.e., a mesh homeomorphic to a triangulation of a sphere.
2. **Manifold complex**: An orientable manifold without boundary, but possibly with handles or higher genus.
3. **Manifold complex with boundary**: An orientable manifold with boundary.
4. **Connected-stars triangle complex**: A complex with no sticks that may have non-manifold edges, but does not have star-connecting vertices. A star-connecting vertex is a vertex with more than one connected component in their star, which comprises the incident triangles and sticks.

An example is shown in Figure 3, where a side-respecting traversal is illustrated: a bug on one of these faces (a) can walk across that non-manifold edge to the proper face of an adjacent triangle (b). Our side-respecting traversal operator called swing identifies the proper triangle and face. Results of iterating the swing operator are shown as the image sequence from (a) to (e) until the bug returns to its original face. The instantaneous position of the bug is identified by a corner, which is defined in the next section.

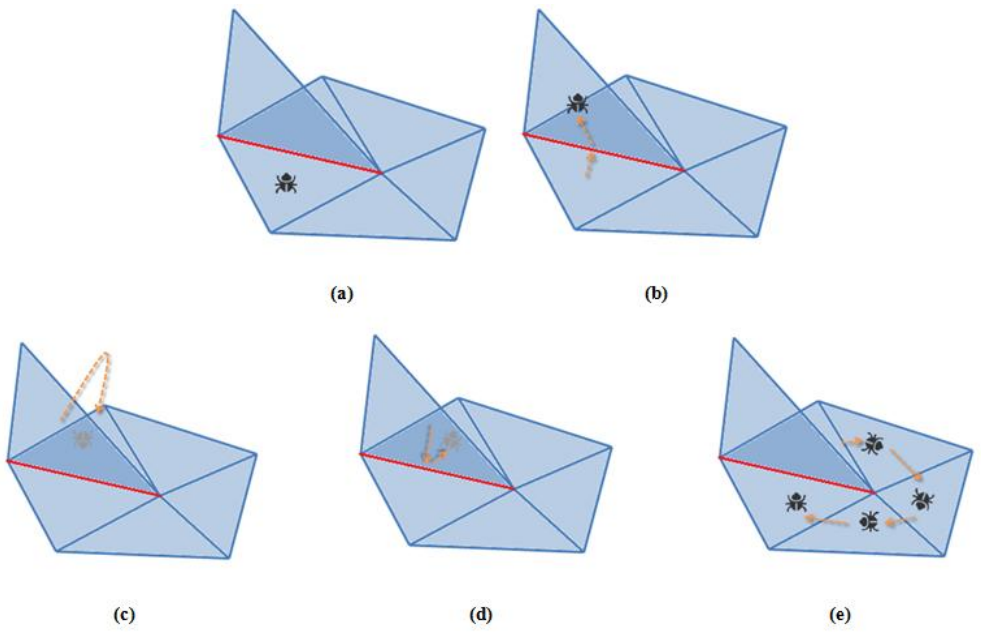


Figure 3: Example of side-respecting traversal.

5. **Triangle complex**: A simplicial complex with only triangles and no sticks that contains star-connecting vertices. An example is shown in Figure 4. A fan of incident triangles and a single incident triangle are connected only by v. A bug on one face of the triangle from the fan cannot reach a face of the isolated triangle by walking over an edge. It must jump from the current fan to the isolated triangle.

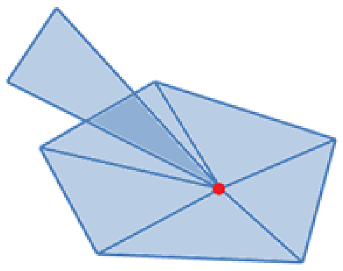


Figure 4: Example of a star-connecting vertex v marked in red.

6. **Simplicial 2-Complex (S2C)**: A connected simplicial complex of dimension 2 with dangling edges (sticks). Note that an S2C is connected, then it does not have isolated vertices. Figure 5 shows examples of several different S2Cs. The S2C in (a) has a single stick, and the one in (b) has a more complex (wireframe) structure of sticks.

The S2C in (c) is obtained by joining two manifold complexes with boundary by a stick. Hence, the k constant discussed above is 1 for this S2C.

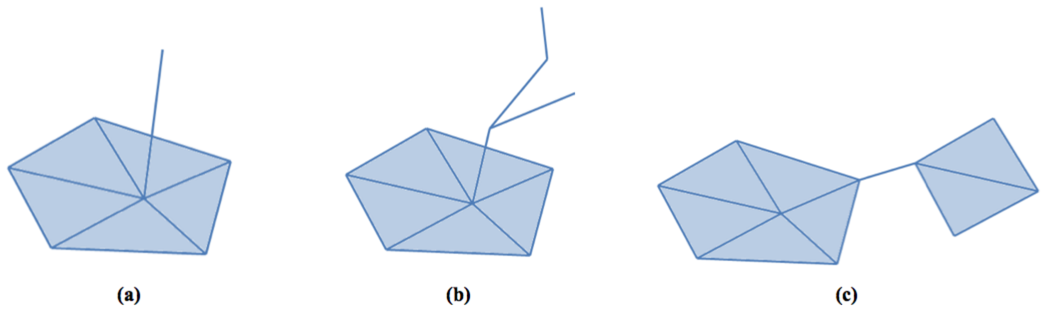


Figure 5: Examples of a Simplicial 2-Complex (S2C).

2.2 Entities used for access and traversal S2Cs

We discuss here entities that can be accessed by application algorithms or that are used by our implementation of the RAT operators or by our Vlist to Ering conversion algorithm. We discuss entities for the triangles first and then address those for sticks.

2.2.1 Triangle entities

From each triangle, we can access several entities shown in Figure 6. Every triangle has 3 vertices and 3 edges bounding it. Vertices are shown in magenta and edges are shown in gray.

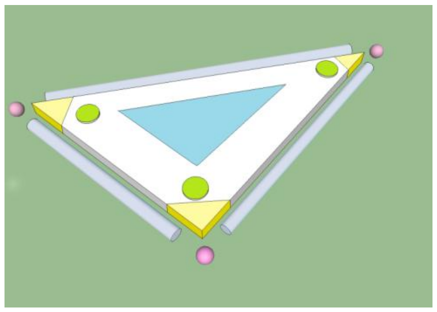


Figure 6: A triangle with its front face shown in cyan), its 3 wedges shown in yellow, and 3 of its 6 corners shown in green.

In addition to vertices and edges, we use wedge, corner and face. These are defined as follows. A triangle has three wedges shown in yellow in Figure 6. A wedge is the tuple associating the triangle with one of its vertices. In several prior publications mentioned above, this tuple was called a corner. To support our side-respecting traversal, we distinguish the two faces of a triangle and define 3 corners on each face. Hence, we use a different name, a wedge, for the triangle/vertex tuple when there is no need to distinguish one of the faces of the triangle.

As mentioned above, each triangle has two faces. Think of the triangle as a piece of cardboard. Each side or face of it can be painted in a different color. We call one of them the front face and the other one the back face. The order in which the references to the three vertices of the triangle are listed defines which face is front and which one is back. By convention, when a front face is visible, the vertices in the order in which they are referenced by the triangle appear clockwise in Figure 6, the center of the front face is painted cyan. We only paint the center part for clear representation.

With each triangle, we associate 6 corners, two per wedge as one on each face. The 3 corners of the front face are shown in Figure 6 as green disks, positioned close to the corresponding wedge.

Let V denote the vertex count, i.e., the total number of vertices in the S2C, and let T be the triangle count. Clearly, the wedge count W equals 3T. As in several recently proposed data structures, in order to reduce storage, we do not use heap records and pointers. Instead, we assign consecutive integers in [0,V-1] to the vertices, in [0, T-1] to the triangles, in [0, 3T-1] to the wedges, in [0, 2T-1] to the faces, and in [0, 6T-1] to the corners and we store integer references to these entities in tables that are indexed by these integers.

We do not assume any particular order for the vertices or the triangles. This is an important advantage over some previously proposed data structures that reorder triangles, vertices, or both, because the original order may have been defined by algorithms that strive to reduce the span of the mesh, so as to enable streamed processing. We also do not assume any consistent orientation of the triangles. Indeed, a consistent orientation can be defined for water-tight non-manifold meshes without boundary, but there is no consistent orientation for a more general S2C, such as the one shown in Figure 3.

In the Vlist and Ering representations proposed here, the wedges of a triangle t are associated with the consecutive wedge IDs: 3t, 3t+1 and 3t+2. The front face of t is the one for which the vertices referenced by wedges 3t, 3t+1, and 3t+2 appear clockwise to those seeing it; similarly, the back face of t is the one corresponding to 3t, 3t+1 and 3t+2 appeared in counter-clockwise. We assign to the front face of triangle t the face ID 2t and, to the back face, the face ID 2t+1.

The corners on the front face of t are associated with corner IDs 6t, 6t+1 and 6t+2 and correspond to wedges 3t, 3t+1 and 3t+2 respectively. The corners on the back face of t are associated with corner IDs 6t+5, 6t+4 and 6t+3 and correspond to wedges 3t, 3t+1 and 3t+2. In Figure 7, triangle t is shown as an artificially thickened slab for clarity. On the left, t has its 3 wedges identified by color: 3t (red), 3t+1 (orange) and 3t+2 (yellow). Hence, its front face is the top one. Right: the IDs of the 6 corners are marked: 6t (red disc in solid), 6t+1 (orange disc in solid), 6t+2 (yellow disc in solid), 6t+3 (yellow disc in dash), 6t+4 (orange disc in dash) and 6t+5 (red disc in dash). This numbering ensures that, the corners of a visible face always appear in clockwise order.

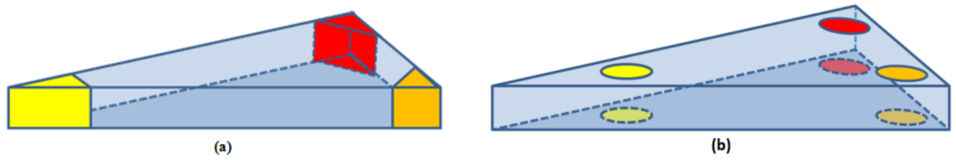


Figure 7 : Example of triangle face ID, wedge ID and corner ID.

2.2.2 Stick entities

We use the term stick when referring to the dangling edges that do not bound a triangle. Because the S2C is connected, a stick must share at least one of its vertices with at least on stick or with at least one triangle. From a stick we can access its two vertices, shown in red in Figure 8. We also identify its two ends shown in yellow. Each end is associated to the corresponding vertex. Note that an end of a stick plays the role of a wedge and of a corner, since the stick does not have faces. Hence, we will often use the term corner when referring to a stick end, so as to simplify explanations.

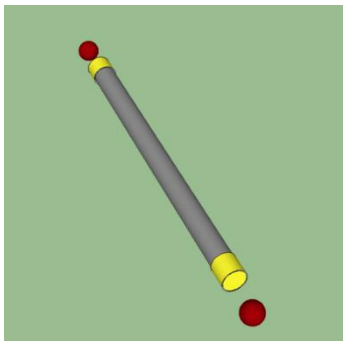


Figure 8: A stick with its 2 vertices shown in red and its two ends/corners shown in yellow.

3. RAT OPERATORS

In this chapter, we propose a set of Random Access and Traversal (RAT) operators for accessing and traversing S2Cs. We believe that the operators of practical value to application developers are those that apply to or produce corners. However, we also define other operators on wedges and on faces because these help us specify the corner operators and may be of value to some application developers. We use the object oriented syntax, where for example c.x.y means that we start with a corner c, apply the x operator, and then apply the y operator to the result. In practice, this cascading is implemented as y(x(c)), but we believe that the notation c.x.y is more intuitive and easier to follow.

3.1 Triangle and wedge operators

We use a single triangle associated operator t.w.

- t.w returns the ID of the first wedge of triangle t.

Triangle-to-corner and triangle-to-vertex access is achieved by combining this triangle operator with operators defined below.

We also propose three wedge operators as follows:

- w.t returns the triangle of wedge w.
- w.n returns the next wedge with respect to w around w.t. w.n is defined as the next wedge after w in clockwise order when looking at the front face of w.t.
- w.v returns the ID of the vertex of wedge w. Its computation depends on the data structure used and will be discussed in the next sections.

3.2 Corner operators

We propose several corner operators, where a corner c may refer to a triangle corner or the end of a stick.

- c.w returns the wedge of corner c, and c.f returns the face of corner c.

- c.n returns the next corner on face c.f. The term “next” is defined so that corners c, c.n, and c.n.n appear clockwise to a viewer that sees the face c.f. Note that, if c lies on the back face of the triangle, the wedge c.n.w of c.n is not the c.w.n. We made this semantic choice so that, from the application perspective, our corner operations on a shell correspond to the standard corner operators on a manifold triangle mesh [15].
- c.x returns the corner that is on the other side of the triangle c.t at the same wedge, i.e., the corner on the opposite face of the triangle. The mnemonic ‘x’ stands for “cross” since this operator crosses or pierces the surface of the triangle.
- c.s returns the swing corner obtained by swinging around c.v and crossing over one edge of c.t to access a face on the same side, i.e., not piercing through any triangle. For example, in Figure 9 corners c0, c1,… c5 are incident on the tip vertex of a closed loop fan of triangles. The tip vertex is marked in red. The loop may be traversed in order using swing: c0.s=c1, c1.s= c2, … c5.s= c0. Note that the loop of corners visited by swinging, when visible appears clockwise.

Note that the c.v operator that returns the ID of the vertex of corner c needs not be defined explicitly, since it may be achieved as the combination c.w.v. Similarly, c.t, which returns the triangle of corner c is achieved by c.w.t. The reverse of c.n is the previous corner operator c.p. The reverse of c.s is the unswing operator c.u achieved by performing c.n.s.n.

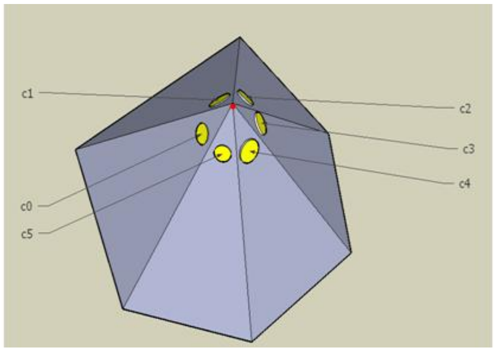
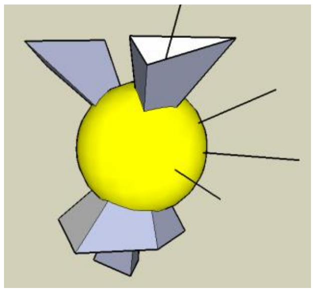
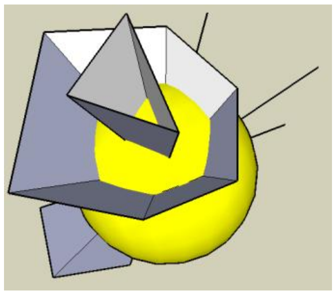


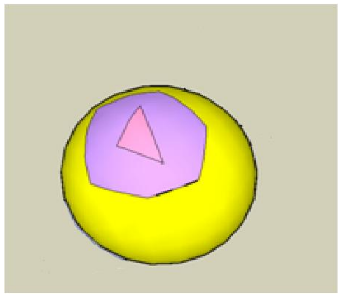
Figure 9: Swinging around the incident vertex shown in red: c0.s=c1, c1.s= c2, … c5.s= c0.

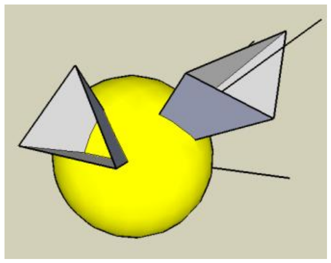
Consider a vertex v that has several incident triangles and sticks. Its star, v*, may have several connected components. We call them the incident components of v. Some of these components are individual sticks. Others are edge-connected sets of triangles incident upon v. We define the ball B(v) of v as a ball around v that does not contain any other vertex. The connected components of the intersection of the S2C with B(v)-v are called the cones of v. The boundary of each cone of v comprises some of the faces and some of the sticks of v. More specifically, the boundary of a cone of v may contain a triangle t. One or both faces of t may be exposed to the interior of that cone. For example, in Figure 4, there are 2 cones. One formed by the faces of the triangle fan that face downwards; another is sandwiched in between two boundaries: faces of the triangle fan that face upwards and faces of the isolated triangle. The front face and back face of the isolated triangle form the boundary. And both faces are exposed to that cone. To simplify the formulation, we say that all the sticks in the boundary and all the faces exposed to the interior of the cone are in the cone. We also say that a corner c is in a specific cone of c.v and is contained in a face or stick of that cone. We say that that cone is the cone of c.

Figure 10 shows the ball around v that has several incident triangles and sticks incident upon it. Figure 10(a)(b) and (d) are the same structure seen from different points. Figure 10(c) shows the intersections viewed from (b), and (e) shows intersection viewed from (d). The boundary of cones separates the ball to several parts that are colored differently.


(a)


(b)


(c)


(d)

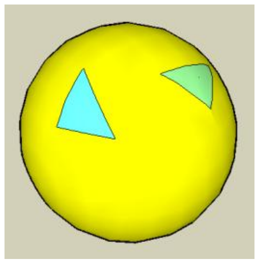

(e)

Figure 10: Examples of Ball B(v) that intersects with different components. Sticks map to an isolated point and triangles map to edges on B(v)'s surface. The intersection generates 5 cones colored in different colors.

Let us consider the triangle corners that are incident upon v. They form the swing loops of v that are traversed by iterating the swing operator, as explained above. For example, a component of v* that contains a single triangle has a swing loop of v that contains only two corners: a corner c and the corner c.x on the opposite side. As another example, consider the S2C of Figure 3. The central vertex has 2 swing loops: the one on the top crosses over the non-manifold edge twice; the one on the back crosses each edge of the fan spoke once. Each loop is contained in a single cone of v.

A good intuition of these two concepts may be obtained by considering the mapping of these entities onto their intersections with the sphere S that bounds B(v). The incident components of v each map to an isolated point on S for the sticks of v and to connected edge graphs for the incident triangles of v. These isolated points and the connected edge graphs decompose S into spherical faces. These spherical faces are the connected portions of the difference between S and the S2C, which is the complement of the isolated points and edge graphs with respect to S. Each cone of v maps to a different spherical face. We think of the edges as streets and of the swing loops as sidewalks along the streets. Each swing loop of v maps onto a closed-loop of sidewalks around some portion of a connected edge graph. Here, we define operators that jump from one closed-loop of sidewalks or isolated point to another closed-loop of sidewalks or isolated point that are accessible from the same spherical face.

In a subsequent section of this thesis, we explain how our representation adds auxiliary triangles incident upon v that map into golf cart paths on S through faces. We add a minimum set of golf cart paths to connect all the closed-loop of sidewalks and isolated points of a spherical face. Now consider the union of the golf cart paths and the streets. Each spherical face has a single closed-loop of sidewalks that visits them. Hence, all corners and all sticks of a cone can now be traversed by a single walk by using the swing operator.

We have already defined a swing operator and shown how to use it to visit the corners of a swing loop of v. Below, we define other operators for accessing the sticks and other swing loops in the boundary of the cone that contains a given corner c. To do so, we organize the corners of v, i.e., the corners c such that v=c.v, into ordered sets, which we call cycles. Each cycle is on the boundary of a unique cone of v. All the sticks in a cone form an ordered list or cycle. We will call it a stick cycle of vertex v. Each swing loop of v is a different entry in the swing loops cycle of a cone of v. Hence, each cone of v has a stick cycle and a swing loops cycle. Each of these may be empty. We define below a method for jumping from one element of a cycle to the next element in that cycle and a method for jumping to an element of the other cycle. We chose the following syntax.

- c.e jumps to an end of the stick cycle of the cone of c, when one exists. If c is a stick-end, the result of c.e is the next stick end in the same stick cycle. Otherwise, it is the first stick end in the stick cycle of the cone of c. c.e returns c when c is a triangle corner and the stick cycle of the cone of c is empty and when c is the only stick end in the stick cycle of that cone.
- c.k returns a triangle corner of the swing loops cycle of the cone of c, when one exists. If c is a triangle corner, then c.k returns a corner in the next swing loop cycle of c. Otherwise, when c is an end, c.e returns a triangle corner of the first group in that list. When the corner of c has no swing loops, c.k returns c.

Figure 11 demonstrates a start-connecting vertex having two cones incident to it. Each cone has a stick and 3 faces. Stick1 and stick2 are separated and belong to different cones.

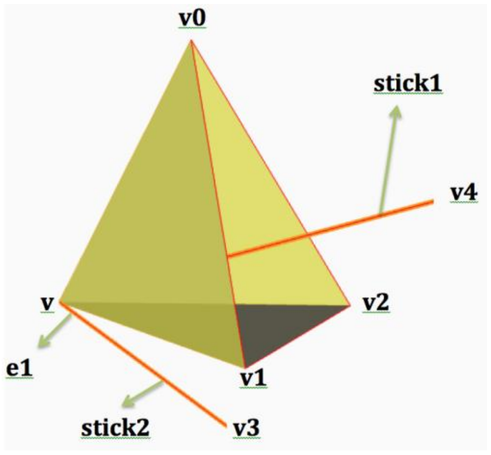


Figure 11: The vertex v in S2C has 3 incident components and 2 cones. Each cone has one stick and 3 faces.

Figure 12 shows a star-connecting vertex with 7 incident components. In Figure 12(a), starting from a triangle corner c of v, we can use c.s to reach the swing corner in the same swing loop, c.k to reach a corner on the next swing loop of the cone of c, and c.e to reach the first corner (stick end) of the cone of c. In Figure 12(b), starting from a stick end c, we can reach the next stick in the stick cycle of c.v using c.e and a triangle corner on the first swing loop of the swing cycle of the cone of c using c.k. These side respecting operators do not pierce any triangle surface, hence they cannot reach the partly visible cone of incident triangles that appears at the bottom of the figure. Figure 13 shows an example of traversing with those provided operators. Starting from corner c, we can get access other corners of the chamber of c.

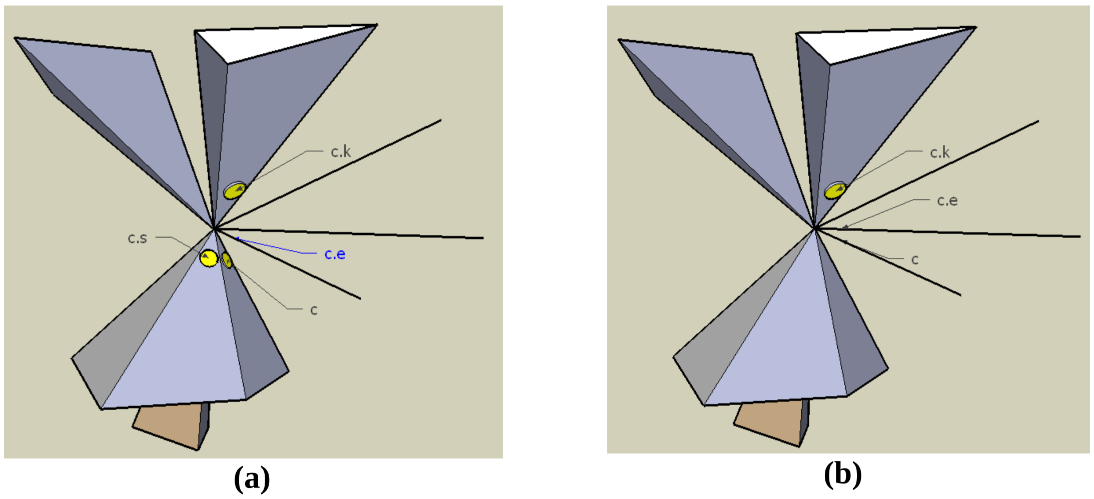


Figure 12: Illustration of operator .e, .k and .s from corner c. (a) c is a corner on the triangle and (b) c is a corner / end on a stick.

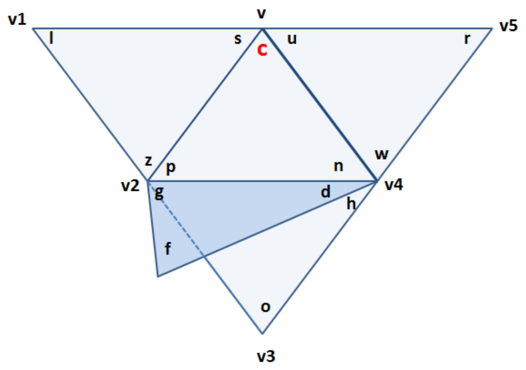


Figure 13: Starting from c, we can access n=c.n, p=c.p; s=c.s, l=s.p, z=s.n; u=c.u, w=u.p, l=u.n; d=n.u, f=d.n, g=p.s; h=d.x.u, o=h.n; v=c.v=u.v=s.v;v1=l.v; v2=z.v=p.v=g.v; v3=o.v; v4=w.v=n.v=d.v=h.v; v5=r.v;

Some RAT operators we discussed about do not depend on specific data structure. We list the implementation of some primitive RAT operators that are common for both our Vlist and Ering representations.

t.w is implemented as {return 3t;}

w.t is implemented as {return w/3;}

w.n is implemented as {return 3*(w/3)+(w+1) mod 3;}

c.w is implemented as {return 6*(c/6) + c mod 3;}

c.t is implemented as {return c/6;}

c.f is implemented as {return (c/3) mod 2;}

c.n is implemented as {return 3*(c/3)+(c+c.f+1) mod 3; }

c.x is implemented as {if(c.f==0) return c + 5– 2*(c mod 3); else return c -2*(c mod 3)-1;

}

4. VLIST FOR CONNECTED-STARS TRIANGLE COMPLEX

In this chapter, we explain the Vlist data structure for the special case of connected-stars triangle complexes. In these models, using our terminology on the sphere bounding B(v), each spherical face is bounded by a single closed-loop of sidewalks. Hence, all the corners of a cone of a vertex can be traversed by a single cycle of swings. Remember that such a S2C does not have sticks. The extension to more general S2Cs is discussed in chapter 6.

4.1 Representation and storage cost of connected-stars triangle complexes

Our Vlist data structure maintains three tables: a G table, an H table and an L table:

- The G table stores the location of 3D point G[v] of vertex with index v. Hence, it has V entries.
- The H table stores the head H[v] of the list of all wedges incident upon vertex v. Hence, it has V entries.
- The L table stores the ID L[w] of the next reference in the list of wedges of v. Hence, it has 3T entries, one per wedge.

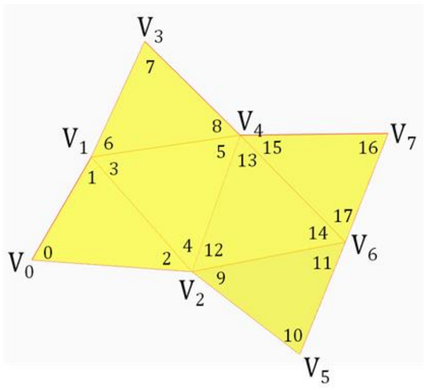
When the list is empty, H[v] stores the ID of vertex v. Otherwise, the last entry in the list stores the ID of v. Thus the list is circular and contains v as one of its elements. As explained below, we exploit the fact that w.v can be reached from w by following the list stored in L. To know whether L[w] stores the ID of a wedge or of the vertex w.v, we use the leading bit and set it to 1 when L[w] references a vertex.

To be consistent with our object-oriented notation, instead of writing G[v], H[v], and L[w], we will write v.g, v.h, and w.l. We use the Boolean function isVertex(w), which returns true when w references a vertex and flase when it is a wedge. Its implementation simply tests whether w<0.

Hence, the total storage costs for the connectivity of a connected-stars triangle mesh is V+3T: V entries for H, and 3T entries in L. Note that this cost is nearly half of the connectivity cost of the ECT representation mentioned above, and yet it provides the same functionality. Furthermore, it is close to the cost of the SOT representation, which is 3T. In a simple complex, T=2V–4, hence Vlist stores 3.5T references while SOT stores only 3T references.

For connected-stars triangle complexes, the advantage of the Vlist over SOT and other compact data structures discussed above is that Vlist does not require reordering the triangles and supports constant cost editing, assuming that the maximum number of triangles incident on a triangle is small compared to the triangle count.

A simple example of a connected-stars triangle complex and the details of its Vlist representation is shown in Figure 14. The wedge ID is labeled in the mesh shown on left. As we can see, we do not assume any consistent orientation of the triangles.



G

0	x_0	y_0	z_0
1	x_1	y_1	z_1
2	x_2	y_2	z_2
3	x_3	y_3	z_3
4	x_4	y_4	z_4
5	x_5	y_5	z_5
6	x_6	y_6	z_6
7	x_7	y_7	z_7

H

0	0
1	6
2	12
3	7
4	15
5	10
6	17
7	16

L

0	$-v_0$
1	$-v_1$
2	$-v_2$
3	1
4	2
5	$-v_4$
6	3
7	$-v_3$
8	5
9	4
10	$-v_5$
11	$-v_6$
12	9
13	8
14	11
15	13
16	$-v_7$
17	14

Figure 14: The wedge IDs (left) for a manifold complex with boundary, and the contents of the H, L and V tables stored in its Vlist representation (right).

4.2 Vlist construction algorithm

For a connected-stars triangle complex, we assume that the input to our Vlist construction is an indexed representation, which contains two parts:

- The geometry part lists the three Cartesian coordinates or any other chosen representation for each vertex. It is assumed that the vertices are associated with integer IDs, 0, 1, 2, … V-1 and that they match the order in which the vertices are listed.
- The connectivity part that lists the three integer IDs of the vertices associated with each triangle. It stores the incident information. Triangles are assigned integer IDs, 0, 1, … T-1, based on their order in this list.

In the example shown in Figure 14, the connectivity description is the list of triples {{0,1,2},{1,2,4},{1,3,4},{2,5,6},{2,4,6},{4,7,6}}. Triangle 0 has vertices {0,1,2}, triangle 1 has vertices {1,2,4} and so on. The Vlist construction algorithm proceeds as follows.

- Read the geometry part and fill the G table.
- Initialize the H table so that H[v]=v, but flip the first bit of each entry to produce a negative number.
- Read the connectivity data and, for each entry, update the H and L tables to insert the new wedge at the head of the wedge list associated with its vertex.

For example, when we read triplet {i,j,k} for triangle t, the following assignments are performed:

```
L[3t] = H[i]; H[i] = 3t;
L[3t+1] = H[j]; H[jj] = 3t+1;
L[3t+2] = H[k]; H[k] = 3t+2;
```

Table 1: Generating the L table and updating the H table.

4.3 Operator implementation

In this subsection, we provide the details of our implementation of operators for the Vlist representation

4.3.1 Vertex operator

For wedge w, the vertex operator w.v (implemented as v(w)) returns the ID of the vertex of w. Since Vlist does not store this information explicitly, it must be retrieved by traversing a portion of the wedge list that contains w, until we reach a negative entry, from which we extract the desired vertex ID. The complexity of this search is proportional to the number of wedges of w.v. Our implementation of the vertex operator, v(w), is shown in Table 2. v(w) returns the result of ANDing v with a mask that has ones, except for the leading bit.

```
int v(int w){
        int v=L[w];
        while(v>=0) v=L[v];
        return v&&'8FFF;
}
```

Table 2: Code to retrieve incident vertex ID of wedge w

For corner c, we implement c.v as c.w.v.

4.3.2 Swing operator

Here, we explain how to compute the swing c.s of a corner c. To reach c.s from c, we will cross over the oriented edge (v,u), where v=c.v. The wedges of v are ordered around edge (v,u) clockwise when looking from vertex v towards vertex u. Note that it is not necessary to compute the entire circular ordering. It is only necessary to compute the next wedge x after c.w. Once we have the ID of x, we can compute c.s. The result depends on whether we are walking onto the front or the back face of x.

Figure 15 shows a vertex v0 with one incident non-manifold edge marked in red joining it to vertex v3. For corner c of w0 on the visible face, three steps are taken to compute c.s. The first is to identify triangle {a, b, c} of corner c and edge {u, v} that c is about to cross. For corner c, the triangle is {v0, v1, v2} and the edge is {v0, v2}. The second step is to identify the wedge we we are walking onto. Candidate wedge w should be incident on c.v with its next or previous wedge incident onto c.p.v. From all those candidate wedges, we find the next wedge x after c.w when they are in clockwise order. The third stop is to identify the face of the triangle x.t and return the result. If we are looking at the front face of x.t, then c.s is the corner of x on the front face. If, on the other hand, we are looking at the back face of x.t, then c.s is the corner of x on the back face. The code for our implementation is given below.

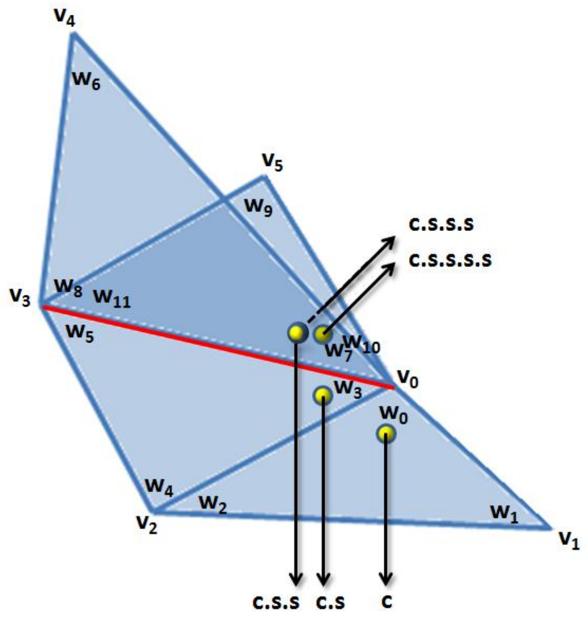


Figure 15: Illustration showing a corner c and the results of c.s, c.s.s, c.s.s.s, and c.s.s.s.s.

```
int getSwingCorner(int c) {
        ww=c.w; u=w.v; v=c.p.v; z=c.n.v;
        maxAngle = 2π;  x=ww;
        w=u.h;
        while(!isVertex(w)){
                if(w!=ww){
                        if(w.n.v==v || w.p.v==v ){ // find candidate wedge w satisfying condition w.t
                and c.t sharing edge (u, v) , assign p to the other vertex of w.t
                                if(w.n.v==v) p=w.p.v; else p=w.n.v;
                                planeAngle= angle(z.g, u.g, v.g, p.g);
                                if(planeAngle <maxAngle) {maxAngle = planeAngle; x=w;}
                        }
                }
```

```
                    w=w.l;
          }
          if(x==ww) return c.x; // there is no other triangle sharing edge (u,v) with c.t, return the corner on
the other side of c.w
          if(clockWise(E, x.v.g, x.n.v.g, x.p.v.g)) // function cw(E, A, B, C) check if vertex A, B, C appear
          clockwise from viewpoint E
                    return getCorner(x, true); //function getCorner(x, isFrontface) returns corner of x on the
          front face if isFrontface=true or on the back face if isFrontface=false
          else return getCorner(x, false);
}
int getCorner(int w, bool isFrontface){
          if(isFrontface) return (w/3*6+w%3);
          else return (w/3*6+5-2*(w%3));
}
float angle(point A, point B, point C, point D) {
          vector U= BA.crossProduct(BC).normalize; vector V=BD.crossProduct(BC).normalize;
//A.crossProduct(B) performs cross product A×B
          return acos(U.innerProduct(V)); //A.innerProduct(B) performs dot product A•B
}
bool clockWise (point A, point B, point C, point D) {
          vector U=EA, V=EB, W=EC;
          return U.innerProduct(V.crossProduct(W))>=0;
}
```

Table 3: From corner c, the code for getting the next corner by swinging in the clock wise order from view point

5. ERING FOR CONNECTED-STARS TRIANGLE COMPLEX

In this section, we present our Ering representation for a connected-stars triangle complex. We discuss the data structure and its storage cost, its construction, and the implementation of operators, where it differs from those discussed above. In chapter 6, we extend Ering to general S2Cs.

5.1 Representation and storage cost of connected-stars triangle complexes

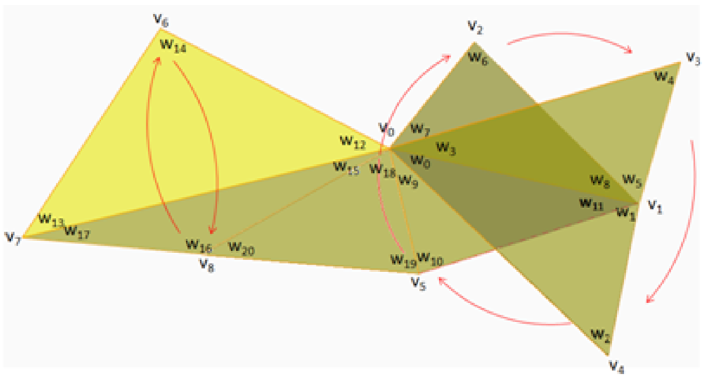
The Ering data structure that is proposed here supports RAT operations and CAT cost. In addition to the geometry table G, it maintains two connectivity tables: a V table and an R table.

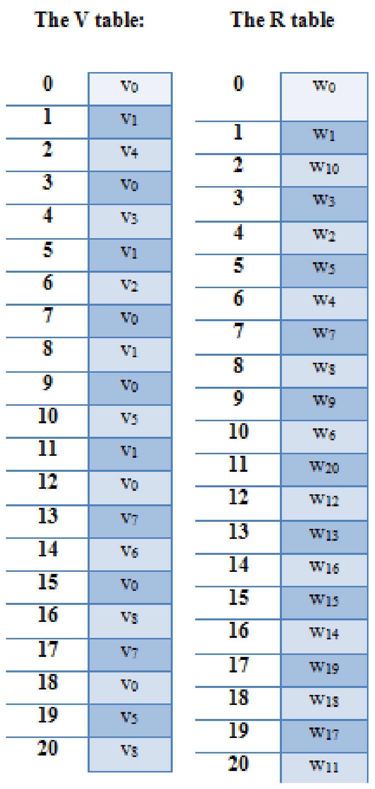
- The V table stores the index V[w] of the vertex of wedge w and it has 3T entries.
- The R table stores the index R[w] to the next wedge in the circularly ordered list of wedges that are facing facingEdge(w), which is the edge of w.t that is not bounded by w.v. If there is no wedge that is facing the facingEdge(w), then R[w] contains w. The table R has 3T entries.

An example of this representation is shown in Figure 16. Note that wedge w faces edge (u, v) if (w.n.v= =u && w.p.v= =v) || (w.p.v= =u and w.n.v= =v). When edge (u, v) is a border edge, it has only a single incident triangle, and R[w] contains only w. Similarly, when edge (u,v) it is a manifold edge, then it has exactly two incident triangles and R[w] contains only the other wedge that is facing (u,v). When the edge (u,v) is non-manifold, then we compute the clockwise cyclic order of all wedges that face (u,v). To define clockwise, we identify which vertex, v or u, has the smallest integer ID. If u<v, then the triangles are listed in a clockwise order around the oriented edge (u,v). Otherwise, a counterclockwise order is used.

Ering stores 2 references, V[w] and R[w], of each wedge w for connectivity. Therefore, the total storage cost is 6T references.

Figure 16 shows an example of a connected star triangle complex and the correspoinding Ering representation. We see that four wedges, w2, w4, w6 and w10 face the same edge v0v1. Assuming that v0<v1, these wedges are arranged clockwise as seen from v0 to v1. Thus, since the next wedge of w2 that facing edge (v0, v1) is w10, then R[w2]= w10. Similarly, we have R[w10] = w6, R[w6] = w4 and R[w4] = w2.





The V table:

0	v_0
1	v_1
2	v_4
3	v_0
4	v_3
5	v_1
6	v_2
7	v_0
8	v_1
9	v_0
10	v_5
11	v_1
12	v_0
13	v_7
14	v_6
15	v_0
16	v_8
17	v_7
18	v_0
19	v_5
20	v_8

The R table

0	w_0
1	w_1
2	w_{10}
3	w_3
4	w_2
5	w_5
6	w_4
7	w_7
8	w_8
9	w_9
10	w_6
11	w_{20}
12	w_{12}
13	w_{13}
14	w_{16}
15	w_{15}
16	w_{14}
17	w_{19}
18	w_{18}
19	w_{17}
20	w_{11}

Figure 16: A triangle mesh for Ering data structure demonstration and the tabular representation of Ering's encoding data.

5.2 Ering construction algorithm

Since the input to our Ering construction is a Vlist representation, described earlier, it is necessary to compute the V and R tables. To fill the Ering V table, for each wedge w, we set V[w]=w.v, using the Vlist implementation of w.v described above. Alternatively, table V may be filled directly from the indexed format as follows. When the triplet {i,j,k} of vertex indices is read for triangle t, we set V[3t]=v_i, V[3t+1]=v_j and V[3t+2]=v_k.

The algorithm of computing R[w] can be constructed by using the swing operator. Specially, for each corner c, we consider the triangles c.t and c.s.t, identify the wedges, u and w, of each that is not incident upon a vertex common to both triangles, and set R[u]=w or R[w]=u, depending on whether or not w.n.v<w.p.v. The algorithm is given in Table 4.

```
void computerR (int c){
        w=c.n.w; v=c.v; u=c.p.v; //trying to find R[w] for wedge w with facingEdge(w)=(v,u)
        if(v<u){ rw=c.s.p.w; } //seeking R[w] in clockwise order
        else {rw=c.x.u.n.w;} //seeking R[w] in counter-clockwise order
        R[w]=rw;
}
```

Table 4: Generating the R table

5.3 Operator implementations

In this section, we described how to implement these prime operators that depend on the data structure. The '.v' operator of wedge w can be simply referred to as V[w]. Using the explicit information stored in the R table, the swing operator can be implemented as follows.

We distinguish eight configurations shown in Figure 17 as follows. For corner c, let nw denote c.n.w, va denote c.v and vb denote c.p.v. By testing (1) whether va and vb are in ascending order, we get w2 from R table, and (2) whether we are viewing the front face of w2.t, we get the correct corner we walk on to. An implementation of this algorithm is given in Table 5.

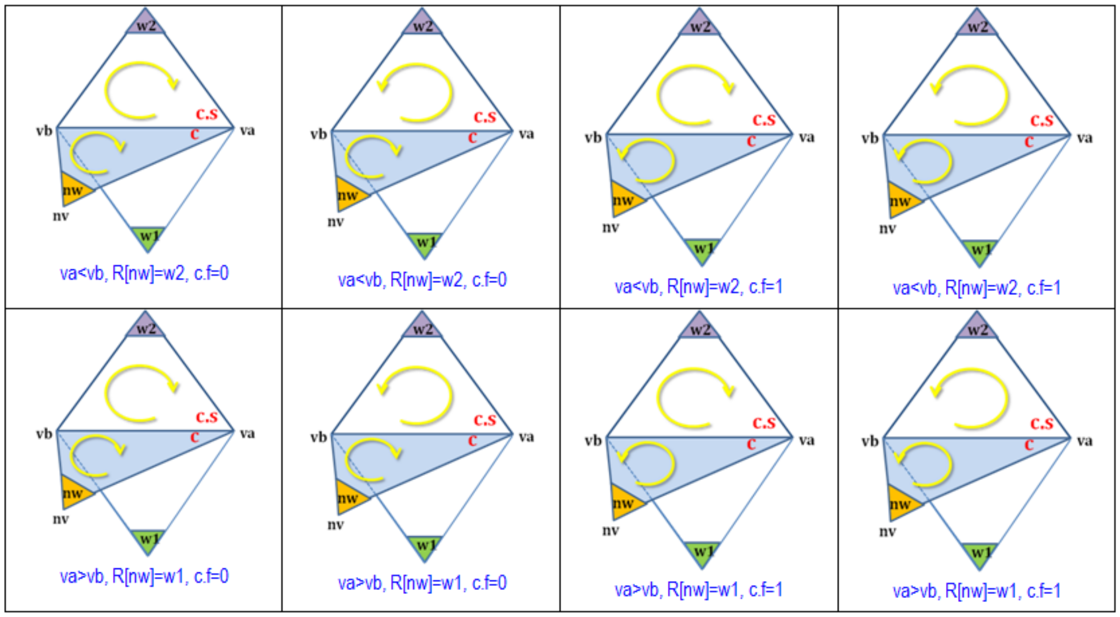


Figure 17: Configuration summaries for swing operator. Those configuration are different by (1) va and vb's value, (2) the face info of c and (3) the orientation consistency of c.t and w2.t.

```
int getSwingCorner(int c){
        va=c.n; vb=c.p.v; nw = c.n.w; nv=c.n.v;
        if(R[nw]==nw) return c.x;
        else{
                rw=R[nw]; //if R table is clockwise ordered viewed from va to vb, R[nw] is the next
        wedge with facingEdge(nw) and R[nw].t ==nw.s.t
                if(va>vb) while(R[rw]!=nw) rw=R[rw]; //if R table is counterclockwise ordered viewed
        from va to vb, iterating the R table to find the 'previous' wedge with facingEdge(nw)
                if(rw.n.v=va) return getCorner(rw.n, true); //function getCorner(x, isFrontface) returns
        corner of x on the front face if isFrontface=true or on the back face if isFrontface=false
                else return getCorner(rw.p, false);
        }
}
```

Table 5: Using Ering to implement the swing operator. From corner c, get the next corner by swinging in the clock wise order from the view point.

6. VLIST AND ERING EXTENSION TO S2C

A general S2C has vertices with disconnected stars and may contain sticks. We use the same Vlist and Ering data structures as those presented below for representing the S2C. The difference is that we introduce a set of stiffener triangles, as briefly mentioned above. Once these are in place, the new S2C does not contain any sticks or disconnected star vertices. Still, we need to alter the implementation of the operators so as to hide these stiffener triangles from the application, so that, for example, a swing operator does not return a corner on one of the stiffener triangles.

We first explain how to add stiffener triangles to convert a triangle complex to a connected star triangle complex. Then, we explain how to add stiffener triangles for the sticks. Finally, we explain how to modify the operators to distinguish stiffener triangles from the original ones and how to identify those used for sticks.

6.1 Adding ghost triangles to a triangle complex

Assume that we are given a triangle complex that has star-connected vertices. For instance, Figure 18 shows a stiffener triangles in green added to connect two triangle components that have swing loops in the same cone. We insert a stiffener triangle shown in green to join two swing loops in the same cone. After insertion, the cone has a single swing loop, which may be traversed by swinging. However, from the application's perspective, the official swing operator should not reach a corner on the green triangle.

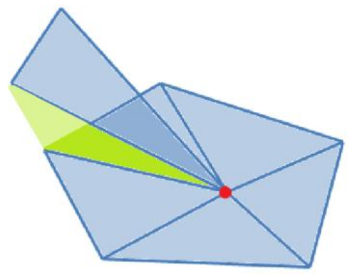


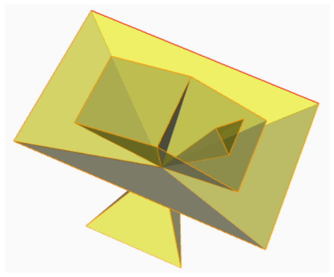
Figure 18: Example of adding ghost triangle to connect loops inside a cone.

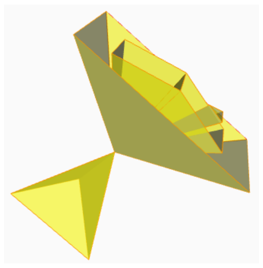
Hence, for each star-connecting vertex v of k components, we need to add k-1 stiffener triangles. These triangles must link all of the corner cycles of each cone of v.

We proceed as follows. We identify the corner loops of v by tracing them one by one. Using the Vlist, we visit all corners incident upon v. For each corner c in that list, if c is not marked, we mark it with the ID of the next corner cycle and perform a walk using a series of swings until we return to c. During that walk, we mark all corners with the new cycle ID. Then, for each pair of swing cycle, we pick an edge on each and consider the triangle t that joins these two edges which share v as a vertex as a candidate for a support triangle. Now, we will explain how we test whether t is a support triangle.

Let the link be the border ring of the triangles in a swing loop. It is shown in red in Figure 19. Let e be the edge of t that is not incident upon v. For each swing loop, we compute the parity p of the number of intersections between e and the triangles of the swing loop and between t and the link of the swing loop. If that parity is even, then t is a support triangle and we insert it. If we are using the Ering representation, the insertion requires modifying the R table of the two edges where the triangle is attached.

In Figure 19, we take an example of 4 triangle groups and show how those ghost triangles are added. Note: we cannot add ghost triangles between two triangle groups if they don't share any face of the same chamber. An instance is that a ghost triangle cannot be added between the innermost 3-face cone and the outermost 3-face cone. As the figures shown above, the ghost triangles are painted in green and connecting every triangle group which shares some faces of the same chamber. Thus a bug can crawl all over the mesh through those ghost triangles.





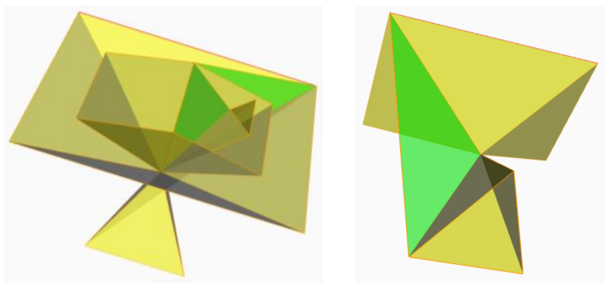


Figure 19: Top left and top right: 4 different triangle groups incident to vertex v. Bottom left and bottom right: 3 ghost triangles shown in green are added to connect triangle groups that have faces from the same chamber.

Here are the details of the algorithm. For every unvisited startCorner that satisfying startCorner.v=v, let c=startCorner, mark c as visited and keep updating c as c=c.s. In this way, c will swing around v and end up back at startCorner again. The faces c traversed form a loop. Different triangle group topology has different loop numbers. For examples, a triangle group as an open fan has only one loop; a cone structure has 2 loops; while a group with non-manifold edges can have even more. In the structure above, loop1 and loop2 belong to one triangle group thus they are glued together. Similarly, loop3, loop4 and loop5 are glued together; loop 6 and loop7 are glued together; loop 8 and loop 9 are glued together. Figure 20 decomposes the structure in Figure 19 with 4 triangle groups and 9 loops in total.

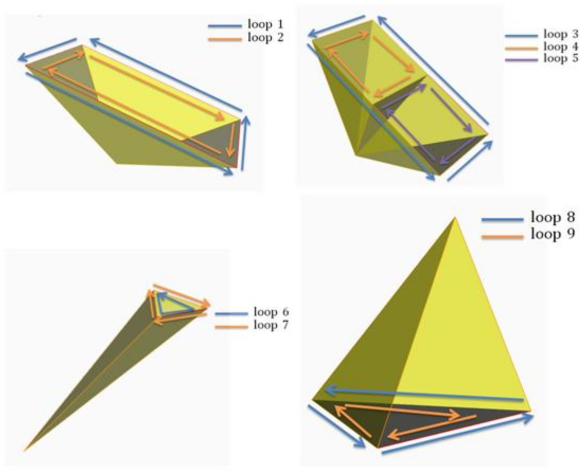


Figure 20: Decomposition of the structure shown in Figure 19 with 4 triangle groups

Imagine there is a ball centered at v. All loops are projected on the ball's surface. Every loop separates the ball's surface to two parts: area inside the loop and the area outside the loop. Denote M as the number of loops and l_0, l_1…, l_{m-1} as those loops. If group A and group B are reachable, then loop l_i belonging to group A and loop l_j belonging to group B will have the same status, i.e., inside or outside, with regard to other loops. In other word, for any vertex v_i on l_i and any vertex v_j on l_j, if v_i is outside of l_k then v_j must also be outside of l_k. Draw a line to connect v_i and v_j. For every loop l_k which is not l_i and l_j and those glued to l_i or l_j, every time the line intersects with l_k, the status flips either from inside to outside or from outside to inside. In order to keep v_i and v_j in the same status, the times of intersection must be kept in even number of times. This rule needs to be applied to every loops other than l_i, l_j and those glued to them. If there is one loop l_k with odd number of intersections, v_i and v_j can be diagnosed as separated by loop l_k, thus group A and group B are not reachable.

The algorithm can be expressed as below:

```
for every loop i {
  for every loop j {
    for every loop k {
      if(i<j and i!=k and j!=k and !isGlued(i,j) and !isGlued(i,k) and !isGlued(j,k)) {
          pick $v_i$ from loop i, pick $v_j$ from loop j;
          status=true;
          for every edge ($v_k$, $v_k$+1) from loop k{
            if(triangle(v, $v_i$, $v_j$) intersects edge($v_k$, $v_k$+1) or triangle(v, $v_k$, $v_k$+1) intersects edge($v_i$, $v_j$))
              status=!status;
          }
        }
        if(!status) break;
    }
    if(k==componentNumber) addGhostTriangle(v, $v_i$, $v_j$);
    }
  }
}
```

Table 6: Adding ghost triangles within every cone without sticks

We add those ghost triangles to the end of the list and keep track of their counts.

6.2 Sticks

The input information that defines the sticks may be defined in the indexed format as a list of vertex ID pairs, one pair per stick. Each pair is represented by 2 vertex IDs and associated with the integer stick ID defined by the order of appearance. We unify the representation of sticks and triangles to make it simpler for development. The sticks information is stored the same way as triangles with the third vertex ID the same as the first one. To be specific, when we read (vi, vj) as the vertex IDs of stick s, the stick end IDs will be 3s, 3s+1 and 3s+2 with respect to vi, vj and vi. The third element serves as a flag indicating this is a stick. For example, given an corner/end ID number i, isStick(i) will return true if i is on a stick or false otherwise by performing i.t.w.v==(i.t.w.p.v).

When the stick is attached to another swing loop and reachable, we add a support triangle. The support triangle is made of the stick and of an edge of that swing loop. The process of testing which swing loop to attach it to is similar to the case between triangle complexes. We identify the corner loops of v the same way as described above. But in this case, we treat every stick incident to v as a single loop. As a triangle loop has the border ring consisting of several vertices, the border ring of stick loop only has one vertex. Then we take no difference between triangle loop and stick loop. The same parity test is performed for testing the proper candidate.

For each vertex v with a star-connecting vertex of k components (including triangle complexes and sticks), we need to add k-1 stiffener triangles. Let V be the number of vertices, T be the number of triangles, S be the number of sticks, C be the number of edge-connected components that incident upon non-manifold vertices and N be the number of disconnected star vertices. The Vlist stores V+3T+3S+3(C+S-N) references and Ering stores 6T+3S+3(C+S-N) references.

The modified algorithm with stick extension can be expressed as below:

```
for every loop i{
        for every loop j{
                for every loop k{
                        if(i<j and i!=k and j!=k and !isStick(k) and !isGlued(i,j) and !isGlued(i,k) and !isGlued(j,k)) { //if loop k is a stick loop, there is no need for parity test.
                                pick $v_i$ from loop i, pick $v_j$ from loop j;
                                status=true;
                                for every edge ($v_k$, $v_k$+1) from loop k{
                                        if(triangle(v, $v_i$, $v_j$) intersects edge($v_k$, $v_k$+1) or triangle(v, $v_k$, $v_k$+1) intersects edge($v_i$, $v_j$)) status=!status;
                                }
                                If(!status) break;
                        }
                }
                If(k==componentNumber) addGhostTriangle(v, $v_i$, $v_j$);
        }
}
```
Table 7:Adding ghost triangles within every cone with consideration of sticks

6.3 Implement '.k' and '.e' operator

As ghost triangles are added, faces and sticks that are in the same chamber are connected together. So a bug on corner c', can crawl to all triangle faces and sticks of the chamber of c'. For instance, in Figure 21, the non-manifold vertex v marked in red has v* of two triangle groups: an isolated triangle and a triangle fan. The ghost triangle shown in green is added to connect these two groups. Then a bug on the corner c' of the isolated triangle with c'.v=v can reach the corner of the fan triangle on the visible face: c'.s will swing to the ghost triangle and c'.k=c'.s.s will land on the triangle fan. The ghost triangle serves as an 'invisible' bridge for swinging around to different triangle faces and sticks that are in the same cone but not connected.

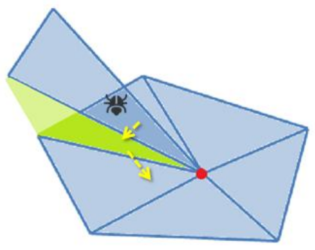


Figure 21: example of 'jump' from one triangle group to another by performing swing.

The algorithm can be expressed as:

```
int k(int c){
        int startC=c;
        c=c.s;
        ghostFlag=false;
        while(c!=startC) { if(ghostFlag && !isStick(c)) break; if(isGhost(c.t)) ghostFlag=true; c=c.s;}
        return c;
}
```

Table 8: Implementation of .k operator

Similarly, the '.e' operator can be implemented as:

```
int e(int c){
        int startC=c;
        c=c.s;
        ghostFlag=false;
        while(c!=startC) { if(ghostFlag && isStick(c)) break; if(isGhost(c.t)) ghostFlag=true; c=c.s;}
        return c;
}
```

Table 9: Implementation of .e operator

6.4 Supporting operator

We provide the implementation of point justification as supporting operator. It does not require additional reference vertex and is at the time cost linear to the vertex valence. It is useful for implementing prime operators and may be of value to some application developers.

For a closing fan which separates the space into two halves, the two sides of the fan will face each half-space respectively. Given an arbitrary point in space, the algorithm classifies which half-space this point is in without taking a reference point. Figure 22 shows a ball B(v) cut by a fan shown in yellow centered at v. The ball is colored with green in one half-space and red in the other. The coloring is achieved by sampling the ball surface and repeat performing the classfyX function described below.

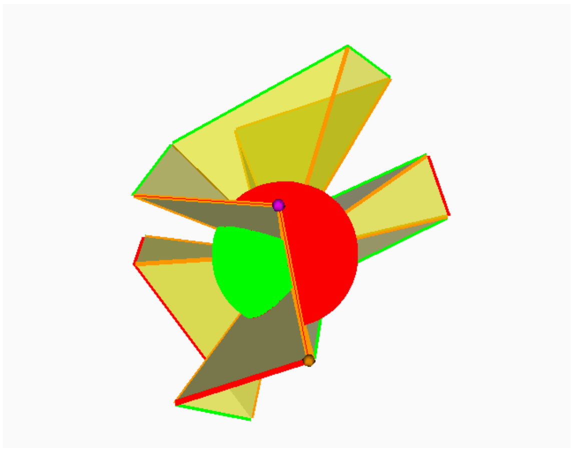


Figure 22: Example of point classification by shown points on the surface of a ball in different colors.

Let v be the center of the closing fan, and v_0, v_1, … v_n are vertices that form the rim of the fan. For point p, the idea of classifying p with regard to the closing fan can be expresses as:

1. Initial the point p status by checking if p is in the cone (v, v_n, v_0, v_1) or not. Cone (va, vb, vc, vd) is a structure formed by vertex va, vb, vc and vd resembling tetrahedron without the surface formed by vb, vc and vd.
2. For every v_i from v_0 to v_{n-1}. check if edge pv_0 intersects with triangle vv_iv_{i+1} or edge v_iv_{i+1} intersects with triangle vv_0p. One intersection means the point crosses the fan's wall once. Flip the value of the initial status for each intersection.
3. After looping round the rim, point p is classified by status.

The algorithm can be expressed as:

```
bool classifyX(point P){
        bool status=isInCone(v, vn, v0, v1, P); // isInCone(V, A, B, C, P) will return true if P is inside
cone(V, A, B, C) emanating from V
        for(i=0;i<n;i++)
                if(edgeIntersectsTriangle(v0, P, v, vi, vi+1) || edgeIntersectsTriangle(vi, vi+1, v, v0,P)) //
edgeIntersectsTriangle (E1, E2, A, B, C) will return true if edge E1E2 intersects triangle ABC
                status=!status;
        return status;
}
bool isInCone(point V, point A, point B, point C, point P){
        bool a = clockWise(V, A, B, C); // clockWise (point A, point B, point C, point D) returns true if
point B, point C, point D appear as clockwise seen from point A
```

```
        bool b = clockWise (V, P, B, C);
        bool d = clockWise (V, A, B, P);
        return a && b && d || !a && (b || d);
}
bool edgeIntersectsTriangle(pt P, pt Q, pt A, pt B, pt C)  {
        vector PA=V(P,A), PQ=V(P,Q), PB=V(P,B), PC=V(P,C), QA=V(Q,A), QB=V(Q,B), QC=V(Q,C);
        bool p= clockWise (PA,PB,PC), q= clockWise (QA,QB,QC), a= clockWise (PQ,PB,PC), b= clockWise (PA,PQ,PC), c= clockWise (PA,PB,PQ);
        return (p!=q) && (p==a) && (p==b) && (p==c);
  }
```

Table 10: Implementation of point classification

7. CONCLUSION

A simplicial 2-complex (S2C) can represent combinations of manifold / non-manifold, orientable/non-orientable meshes with wire-frame (networks of sticks) attached to them. We propose two data structures for S2C: Vlist and Ering. They are both simple and compact. Vlist stores V+3T+3S+3(C+S-N) references and Ering stores 6T+3S+3(C+S-N) references, where V, T, S, C and N respectively define the vertex, triangle, stick, edge-connected component and disconnected star vertex counts. Vlist can be trivially constructed from an indexed format and supports mesh access and traversal at a time cost proportionally to the vertex valence. The Ering data structure, although requiring more storage than Vlist, support faster access and traverse operators. Furthermore, their implementation is purely topological, ie., not requiring geometric tests.

Both data structures support Random Access and Traversal (RAT) operators at Constant Amortized Time cost (CAT). They support side-respecting traversal operators that do not pierce the triangle surface. For models like human heart, where many vessels are attached to the inside of a ventricle and some other vessels are grown out from the heart’s outer skin, side-respecting operators can tell which vessels share the same space and which are separated by the heart skin.

REFERENCES

[1] G. Taubin and J. Rossignac, “Geometric compression through topological surgery,” *ACM Trans Graph*, vol. 17, no. 2, pp. 84–115, Apr. 1998.

[2] J. Rossignac, “Edgebreaker: connectivity compression for triangle meshes,” *IEEE Trans. Vis. Comput. Graph.*, vol. 5, no. 1, pp. 47–61, 1999.

[3] J. Popović and H. Hoppe, “Progressive simplicial complexes,” in *Proceedings of the 24th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 1997, pp. 217–224.

[4] P.-M. Gandoin and O. Devillers, “Progressive lossless compression of arbitrary simplicial complexes,” in *Proceedings of the 29th annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 2002, pp. 372–379.

[5] H. Hoppe, “Progressive meshes,” in *Proceedings of the 23rd annual conference on Computer graphics and interactive techniques*, New York, NY, USA, 1996, pp. 99–108.

[6] T. Gurung and J. Rossignac, “*SOT*: compact representation for tetrahedral meshes,” in *2009 SIAM/ACM Joint Conference on Geometric and Physical Modeling*, New York, NY, USA, 2009, pp. 79–88.

[7] T. Gurung, D. Laney, P. Lindstrom, and J. Rossignac, “SQuad: Compact Representation for Triangle Meshes,” *Comput. Graph. Forum*, vol. 30, no. 2, pp. 355–364, 2011.

[8] T. Gurung, M. Luffel, P. Lindstrom, and J. Rossignac, “LR: compact connectivity representation for triangle meshes,” in *ACM SIGGRAPH 2011 papers*, New York, NY, USA, 2011, pp. 67:1–67:8.

[9] T. Gurung, M. Luffel, P. Lindstrom, and J. Rossignac, “Zipper: A compact connectivity data structure for triangle meshes,” *Comput.-Aided Des.*, vol. 45, no. 2, pp. 262–269, Feb. 2013.

[10] L. C. Aleardi, O. Devillers, and J. Rossignac, “ESQ: Editable SQuad Representation for Triangle Meshes,” in *2012 25th SIBGRAPI Conference on Graphics, Patterns and Images (SIBGRAPI)*, 2012, pp. 110–117.

[11] M. Luffel, T. Gurung, P. Lindstrom, and J. Rossignac, “Grouper: A Compact, Streamable Triangle Mesh Data Structure,” *IEEE Trans. Vis. Comput. Graph.*, vol. Early Access Online, 2013.

[12] L. De Floriani, P. Magillo, E. Puppo, and D. Sobrero, “A multi-resolution topological representation for non-manifold meshes,” *Comput.-Aided Des.*, vol. 36, no. 2, pp. 141–159, Feb. 2004.

[13] R. Juan-Arinyo, àLvar Vinacua, and P. Brunet, “CLASSIFICATION OF A POINT WITH RESPECT TO A POLYHEDRON VERTEX,” *Int. J. Comput. Geom. Appl.*, vol. 06, no. 02, pp. 157–167, Jun. 1996.

[14] I. Navazo, J. Rossignac, J. Jou, and R. Shariff, “ShieldTester: Cell-to-Cell Visibility Test for Surface Occluders,” *Comput. Graph. Forum*, vol. 22, no. 3, pp. 291–302, 2003.

[15] J. Rossignac, A. Safonova, and A. Szymczak, “Edgebreaker on a Corner Table: A Simple Technique for Representing and Compressing Triangulated Surfaces,” in *Hierarchical and Geometrical Methods in Scientific Visualization*, G. Farin, B. Hamann, and H. Hagen, Eds. Springer Berlin Heidelberg, 2003, pp. 41–50.