

A Compact Face-Based Topological Data Structure for Triangle Mesh Representation

Yingzhong Zhang¹ (D), Xiaofang Luo² and Jia Jia³

¹Dalian University of Technology, <u>zhangyz@dlut.edu</u>.cn ²Dalian University of Technology, <u>lxf@dlut.edu</u>.cn ³Dalian University of Technology, <u>49978527@qq.com</u>

Corresponding author: Yingzhong Zhang, <u>zhangyz@dlut.edu</u>.cn

ABSTRACT

A comprehensive and compact topological representation is necessary for information processing of complex triangle meshes. In this paper, a novel objectoriented face-based topological data structure for triangle mesh representation is presented. In the presented model, the face, vertex, and topological relations between them are explicitly defined and the edges are implicitly represented in the sequence of storing vertices, which makes full use of the semantic relations among faces, edges, and vertices. Compared with other popular half-edge data structures, the memory consumption is reduced. In addition, by using object-oriented properties and virtual triangles, the presented model can uniformly represent and process manifold and non-manifold triangle meshes. The topological relations represented in this model are discussed and an algorithm for one-ring neighborhood queries is provided. A method to construct this topological model from STL files is presented, which shows this model can be constructed effectively and efficiently. Finally, an evaluation and discussion on the memory consumption, time performance, and comprehensive performance of this model are provided, which demonstrate that the presented model can represent and process triangle meshes with less memory footprint, high time efficiency, and high flexibility and extensibility.

Keywords: Triangle mesh, Topological data structure, Mesh representation, STL, Face-based data structure. **DOI:** https://doi.org/10.14733/cadaps.2019.539-557

1 INTRODUCTION

As triangle meshes have many unique characteristics, they are widely used to represent the surfaces of the geometric objects in computer graphics, CAD applications, 3D printing, and computational science [6]. In reverse engineering, the point cloud data obtained from 3D scanning is usually

converted into the triangular mesh data in STL format [24],[26]. In practical applications, a mesh model needs to carry out a lot of geometric information processing, such as Boolean operations, mesh simplification, mesh segmentation, and feature reconstruction, which require a lot of topological information among mesh entities [6],[12],[28]. A complete topological data structure for triangle meshes is the foundation to meet above application requirements. On the other hand, as a surface is usually represented with an assembly of tiny triangle facets, in a complex mesh model, the number of triangular facets may reach millions. As a result, the ever-increasing size and complexity of meshes impose stress on both memory usage and processing time. The storage space, querying and editing efficiency, and implementation complexity are important criteria for evaluating a mesh representation model. In addition, in many mesh applications, non-manifold shape may sometimes emerge [14]. For example, in the sampling mesh, due to the intensive sampling points, noise data, and floating point arithmetic errors, a triangle is prone to degenerate into a line or a point, and an edge may be shared by more than two triangles. The geometric and topological representation for non-manifold meshes has been the issues that need to be addressed.

The topological data structure has been researched intensively due to its importance. A wide variety of data structures for triangle meshes have been proposed. The half-edge data structure (HEDS) [22-23] is the most popular data structure and has been smartly designed and constantly improved [6-8],[23], which can succinctly describe the topological connection relation among geometric entities of triangle meshes and implement traversal search with high efficiency without any conditions. Typical implementations of HEDS include Open-Mesh, Surface_Mesh [6],[23], Directed Edge representation [8], Compact Array-Based data structure [3],[11], etc. Most HEDS employ array structures to store geometric and topological data in order to reduce memory footprints. However, many information operations, such as deleting a geometric object or adding a geometric object, are difficult in the geometry processing. In addition, using array structures many object references depend on the array index. Hence, on the one hand, memory consumption needs to further reduce. On the other hand, the computational performance, high flexibility and extensibility, as well as ease of use should be considered [11],[23]. How to achieve a balance between memory requirements, performances and efficiency, and uniformly to represent and process manifold meshes and non-manifold meshes have been the challenges to be confronted.

This paper aims at above challenges, focuses on the STL mesh applications in reverse engineering, and presents a novel compact face-based geometric and topological representation for triangle meshes. The presented model makes full use of the implicit semantic relations among vertices, edges, and triangle faces, which results in memory consumption decrease. In addition, the presented topological representation model does not only consider the compactness of the data structure but also considers the flexibility, scalability, and usability of the mesh model.

The remainder of the paper is organized as follows. Section 2 reviews related work and existing mesh data structures. Section 3 introduces some background knowledge and describes the presented topological representation model. Section 4 presents a method to reconstruct the topological model from STL files. Section 5 makes an evaluation and discussion for the presented model. Section 6 concludes this paper and proposes the further works to be researched.

2 RELATED WORK

As mentioned above, due to the importance of the topological data structure in the information processing of triangle meshes, a wide variety of data structures to represent geometric meshes have been proposed. Here, we limit our review of previous works to the data structure of triangle meshes as follows:

2.1 Face-based data structures

The traditional face-based data structure usually consists of a list of vertices and faces, where each face stores references to vertices defining its boundary. This kind of mesh representation is simple in data structure, and easy to read and display, but does not provide any topological information

among vertices, edges, and faces. As it does not represent the mesh connectivity, it is commonly referred to as the triangle soup or polygon soup. As a result, the geometric querying efficiency of this type of data structures is very low. In fact, the STL file format is a typical face-based data structure. If an efficient access to adjacency relations is needed, it needs to additionally store the neighboring faces of each face and the incident faces for each vertex [6]. CGAL's 2D triangulation data structure [10] and VCGLib [25] are two typical face-based mesh data structures with adjacency relations between faces and the incident relations between vertices and faces, which can retrieve topological relations among mesh entities efficiently.

2.2 Edge-based data structures

In contrast to the face-based representation, edge-based data structures store the topological information in edges or half-edges [6], [22-23]. In general, edge-based structures store references to incident vertices/faces as well as neighboring edges. Edges can represent the adjacency relationship between geometric entities better, since an edge does not only connect two vertices but also adjoins two faces. Currently, the edge-based data structure is the most popular form of triangle mesh data structures. The famous edge-based data structures are winged-edge structures [5] and half-edge structures. Winged-edge structure is an earliest data structure to represent topological information for polyhedral solid. As the traversal search requires conditions to implement choices, its search efficiency is low. Currently, the half-edge data structure (HEDS) is widely employed in the geometric representation of the triangle mesh. HEDS is smartly designed and constantly improved, which can succinctly describe the topological connection relation among geometric entities of the triangle mesh and implement traversal search with high efficiency without any conditions, as shown in Fig. 1. Sieger and Botsch [23] developed a Surface_mesh data structure. Also, Botsch et al. [7] introduced HEDS into OpenMesh. In addition, Campagna et al. [8] presented a directed-edge data structure which was specifically designed for triangle meshes. This data structure can enable the programmer to trade memory for access time, and represent non-manifold triangles by extension. However, the capability to handle non-manifold triangles is still very limited.



Figure 1: A typical implementation for half-edge data structures [23].

The main issue of HEDS is that each edge needs to build two half-edges; at the same time, each half-edge stores a reference to the next and opposite half-edge, together with a reference to an incident face. Hence, a large number of memory resources have to be consumed, especially for the complex triangular mesh with more than one hundred thousand of triangles. In the half-edge data structure, memory consumption is a very important issue to be addressed. Therefore, a lot of research about the compact representation for triangle meshes has been carried out. Most published data structures try to reduce the number of references stored in the implementation. In general, in a pointer-based implementation of HEDS, a half-edge object stores pointers to its opposite, previous,

and next half-edges, as well as to its origin (or destination) vertex and to an incident face. Each face or vertex stores a pointer to an incident half-edge. Hence, such an implementation requires at least eight pointers per edge (four per half-edge) [23], and one pointer per vertex or face. Reducing the number of pointer reference is a direction to improve HEDS.

2.3 Compact topological data structures

In order to reduce the memory footprint, Alumbaugh and Jiao [3] presented an array-based mesh data structure, which is a typical compact representation for both surface and volume meshes. This data structure assigns each half-edge an ID composed of a pair of numbers < f, i >, where f is the ID of its containing face, and i is the index of the edge within the face. In order to reduce memory consumption further, it needs to encode the two-tuple of each half-edge into an unsigned long integer. The design and implementation of this data structure makes full use of the connectivity relations between geometric entities and benefits of the array data structure. As a result, it can provide efficient supports for querying incidence, adjacency, and boundary classification, but requires substantially less memory than traditional HEDS. On the basis of the array-based mesh data structure, many solutions for various applications have been developed. Dyedov et al. [11] proposed an array-based half-facet data structure for mixed-dimensional and non-manifold meshes. This data structure uses sibling half-facets as a core abstraction, coupled with other explicit and implicit representations of entities. Lage et al. [20] also proposed an index-based data structure for the representation of 2D and 3D hybrid meshes composed of elements of different types.

Feng et al. [13] developed a compact data structure for volumetric meshes by using compact combinatorial maps. Kraemer et al. [19] presented a practical implementation of n-dimensional meshes with combinatorial maps. Aleardi and Devillers [1] considered the problem of designing space efficient solutions for representing triangle meshes and proposed an explicit data structure for compactly representing planar triangulations. This data structure combines existing techniques from mesh encoding with a use of minimal Schnyder woods. In addition, Kallmann and Thalmann [18] presented a vertex-based data structure, star-vertex, to represent general planar meshes. Although the array-based mesh data structure can make use of implicit relation information and becomes more compact, this kind of data structures still lacks the flexibility, scalability, and capability to represent non-manifold meshes. In addition, it is difficult to allow storing any custom data in vertices, edges, and faces.

The corner-based mesh data structure is also another kind of compact connectivity data structures for triangle meshes. Aleardi et al. [2] presented a compact dynamic data structure called Editable SQuad (ESQ) for triangle meshes. Gurung et al. [16] proposed a compact representation of incidence and adjacency for manifold triangle meshes with fixed connectivity: Zipper. Zipper uses triangle corners to identify a triangle and one incident vertex. Zipper occupies small space and can be constructed in linear space and time, and supports all standard random-access and mesh traversal operators in constant time. Similarly, Luffel et al. [21] proposed Grouper. Grouper uses corners to represent the geometry and connectivity of a mesh, but it groups vertices and triangles into fixed-size records to ensure that vertices and triangles can be stored in a coherent order that enables memory-efficient sequential stream processing.

2.4 Topological data structures for non-manifold meshes

In recent years, a variety of applications, such as CAD, reverse engineering, and science computation, proposes many geometric issues on non-manifold entities. The topological data structures for non-manifold meshes are the foundation to solve these issues. The classical work on this is the Radial Edge data structure proposed by Weiler [27]. Subsequently, much effort has been made in improving the usability and cost effectiveness of data structures that represent shapes with non-manifold boundaries. Currently, from the published literature, the research works on non-manifold mesh data structures are relative fewer. Floriani et al. systemically carried out the research on the non-manifold mesh representation [9],[14-15]. In [14] they presented a multi-resolution topological representation for non-manifold meshes. Also, they proposed an adjacency-based

representation for non-manifold simplicial shapes in arbitrary dimensions, which is also called IA* data structure that explicitly encodes the relationships between vertices and the triangles of a mesh [9]. However, IA* data structure needs to define more rules or flags to cope with various cases. Its implementation and storage methods are similar to array-based data structures, and the usability and storage efficiency need to be improved.

In summary, from above literature review, we can see that even though many data structures have been successfully developed, they have their individual shortcomings and limitations for specific applications. The traditional half-edge data structure based on pointers consumes more memory resources and is unable to represent non-manifold meshes. The array-based half-edge data structure and the corner-based data structure can reduce memory footprint, but lacks the flexibility, extensibility, and capabilities to represent hybrid meshes and non-manifold meshes efficiently. More importantly, a mesh representation needs to efficiently sever the subsequent specific application. The good usability and extensibility, low memory footprint, and highly modeling and querying efficiencies are still the objective that a mesh representation model will be designed to pursuit.

3 A COMPACT FACE-BASED TOPOLOGICAL DATA STRUCTURE

3.1 Background notions

First, we review some concepts and terminologies, which will establish the foundation for discussing the following mesh representation model.

A mesh is a cell complex representing discretely a geometric object. An *n*-dimensional cell complex *K* is a finite collection of *i*-dimensional cells (i = 0, 1..., n). Here we call *k*-dimensional cells *k*-cells. Hence, vertices are 0-cells, edges are 1-cells, faces are 2-cells, and volume cells are 3-cells. Two cells of different dimensions are said to be incident if one is a subset of the other. For example, a vertex (0-cell) is one of vertices of an edge (1-cell), and the vertex is incident to the edge. Two *k*-cells of the same dimension are adjacent if they share a common (*k*-1)-cell. For example, in a two-dimensional mesh, if two 2-cell faces share a common edge (1-cell), the two faces are adjacent.

A *k*-dimensional cell complex $M \in \mathbb{R}^m$ is a *k*-manifold with boundary if the following conditions are satisfied [20]:

(1) A (k-1)-cell in *M* is incident to one or two *k*-cells of *M*.

(2) The open star of a vertex in *M* is homeomorphic to an open subset of either \mathbb{R}^k or \mathbb{R}^k_+ .

Informally, a manifold (with boundary) is a compact and connected subset of the Euclidean space for which the neighborhood of each of its points is homeomorphic to an open ball, or to an open half-ball. The (k-1)-cells in a k-manifold M incident to only one k-cell and their sub-cells are called boundary cells. A k-manifold is orientable when it is possible to choose a coherent orientation on its k-cells, i.e., two adjacent k-cells induce opposite orientations on their common (k-1)-cells. Objects that do not fulfill this property at one or more points are called non-manifold, while they are called non-regular if they also contain parts of different dimensions [20]. Three kinds of typical cases of non-manifold meshes [14] are follows and as shown in Fig. 2:

- Wire-edges (Dangling edges). An edge *e* of a mesh *M* is called a wire-edge if no triangle *t* ∈*M* exists such that *e* is an edge of *t*; otherwise, *e* is called an edge of a triangle face. When a vertex of a triangle shrinks to another vertex, the triangle will become a wire-edge. For example, edge *AB* shown in Fig. 2 is a wire-edge.
- Non-manifold edges. A triangle edge is a non-manifold edge if it has more than two incident triangle faces. As shown in Fig. 2, edge *BC* has three incident triangle faces so it is a non-manifold edge.
- Non-manifold vertices. In a *d*-dimensional mesh, d > 1, a vertex v is non-manifold if the set of elements incident on v can be partitioned into subsets so that no two entities from two different subsets share a facet. Vertex *D* as shown in Fig. 2 is a non-manifold vertex.

In addition, the vertex only incident to wire-edges is also called as the non-manifold vertex, such as vertex A as shown in Fig. 2.



Figure 2: An example of triangle meshes with non-manifold objects.

In this paper, we focus on the representation for triangle meshes, which are 2-cells and may be manifold or non-manifold. A triangle mesh is considered as 2-manifold, if it does neither contain non-manifold edges, non-manifold vertices, nor self-intersections.

3.2 Vertex neighborhoods in triangle meshes

A vertex neighborhood of a vertex ν is the set of vertices directly connected to ν via an edge and ν itself. In the mesh information processing, traversing the ring neighborhood of a vertex is an extremely important operation, which passes through the all incident faces attached to a given vertex and can obtain the set of the incident vertices connected to the vertex. The performing efficiency of this traversal operation is usually considered to be an important indicator to evaluate a topology data structure. Hence, at first we observe the topological dependence among the topological entities, i.e. vertices, edges and faces, by performing a 1-ring neighborhood traversal as shown in Fig. 3.



Figure 3: Ring neighborhoods of a vertex: (a) An open 1-ring neighborhood, (b) A closed 1-ring neighborhood, and (c) More than a 1-ring neighborhood.

It can be seen from Fig. 3 that we can set an incidence relation between a vertex object and a face object. Specifically, a vertex object contains a reference (or pointer) to a face object where it locates. As a result, from a vertex, its incident face can be visited. At the same time, in a face object, except its three vertices, three incidence relations to its mate faces are also set. A triangle face has three adjacent faces at most. The adjacent faces of a face are defined as its mate faces each other. For example, in Fig. 3(c), the mate faces of the face f_1 are f_2 , f_6 , and f_7 .

After setting the above incidence relations, we can easily implement a query of 1-ring neighborhood for a given vertex. As shown in Fig. 3(b), from the given vertex v_1 , f_1 can be obtained at first; and from f_1 , v_2 and v_7 can be obtained. From the order of vertex v_7 , a mate face of f_1 , namely f_6 , can be obtained. Performing the similar loop program, all vertices on 1-ring neighborhood of vertex v_1 can be obtained. The performing operation is in constant time and independent on the size of meshes.

3.3 An object-oriented face-based topological representation model

Based on the above analysis and geometric characteristics of triangle meshes, we can abstract following topological object classes according to object-oriented methods, and a novel face-based topological representation model (FBTM) for triangle meshes is presented as shown in Fig. 4.

• CVertex class

In triangle meshes, vertices are 0-cells of a mesh complex, which decide the position, shape, and size of a mesh. At the same time, a vertex is one of vertices of an edge, so it is incident to an edge. The vertex is a fundamental geomantic entity in a polygon mesh. Hence, we abstract a *CVertex* class to represent vertex objects.

Apart from the position coordinates of the vertex, we set a pointer to a triangle facet object where the vertex locates as an attribute of the *CVertex* object. Through the pointer, the incident relationship between a vertex and its incident face can be set up.



Figure 4: Class diagram of a face-based triangle mesh model.

• CFace class

A face is a fundamental topological entity in the boundary representation and OpenGL rendering. In triangle meshes, we employ the face entity to represent the 2-cells of a mesh complex. A triangle face is a convex facet surrounded by three edges which are sequentially connected by three vertices.

As the geometric characteristics of simplex, the boundary of a triangle face can be defined implicitly by its vertices. The counterclockwise sequential connection of vertices of a face forms its triangular boundary and its normal vector. Hence, the edges of a face entity don't need to be explicitly defined, which can reduce the memory footprint.

However, an edge is an important topological entity, which does not only connect two vertices but also adjoins two faces. Many topological queries need this kind of topological relations represented by edges. For example, query the next vertex of a given vertex along the boundary of the face. In order to meet this requirement, we define the following two kinds of virtual member functions:

(1) Obtaining an adjacent vertex of a given vertex

In a face, if a vertex is given, the array index of the vertex can be firstly obtained. If the index is *i*, the array index of the next adjacent vertex is (i+1)%3, and the index of the previous adjacent vertex is (i+2)%3. The symbol "%" is the division remainder.

(2) Obtaining an adjacent face

In a *CFace* object, its adjacent mate face object pointers are sequentially stored in its array *pMateFace* as same as its vertex storage. If a vertex is given, the array index of the vertex in its incident face can be firstly obtained. By the obtained array index, it is easy to obtain an adjacent face along the direction from the given vertex to its next adjacent vertex or to its previous vertex.

• *CWireEdge* class

As aforementioned, a wire-edge is assumed the contraction result of a triangle face. Hence, we can define the *CWireEdge* class as the subclass of the *CFace* class, in which only two vertices need to be defined. When a wire-edge object needs to be represented, we can represent it as an instance of the *CWireEdge* class. As the *CWireEdge* class is the subclass of the *CFace* class, the object instance of the wire-edge can be stored in the face container and can be uniformly represented and processed.

In the *CWireEdge* class, as a wire-edge only needs two vertices and there are not any adjacent faces, it is necessary to redefine above four virtual member functions again. In addition, a virtual drawing function for OpenGL rendering is redefined, in which a line rather than a triangle is drawn.

• CMesh class

A mesh is a finite collection of 2-dimensional cells, or face entities. We employ the *CMesh* class to represent a whole mesh or a local sub-mesh of a three-dimensional geometric object. According to above analysis, only the face object and vertex object are needed to be explicitly defined in triangle meshes. As a result, we set two containers as follows:

(1) A container to store face objects

The face container includes two arrays. One is used to store practical triangle faces, and the other is used to store virtual triangle faces. Details on virtual triangle faces will be introduced in the latter.

(2) A container to store vertex objects

The vertex container only contains a hash array to store vertex objects. The vertex object is stored in the array ant its storing position is provided by a hash function so as to search a vertex quickly.

3.4 Representation for non-manifold meshes

As mentioned above, non-manifold meshes often appear in various mesh applications. It is best to use a unified topological data structure to represent manifold and non-manifold meshes. The topological data structure presented above can uniformly represent three cases of non-manifold meshes by defining and using the following rules and strategies:

(1) Wire-edges (Dangling edges)

Wire-edges are defined as instance objects of the *CWireEdge* class, in which only two vertices need to be employed, and there are no mate faces.

(2) Non-manifold edges

Although edge entities are not explicitly represented in the presented model, edges really exist in triangle meshes, and in some cases, non-manifold edges may appear. According to above definitions, a triangle edge is a non-manifold edge if it has more than two incident triangle faces, as shown in Fig. 5. Correspondingly, in the presented topological model, we can also implicitly represent non-manifold edges by defining two rules as follows:

- Setting the order of vertices of the triangles sharing the non-manifold edge to make their normal direction be consistent.
- The mate face selection of each triangle is in accordance with the cycle of triangles around the non-manifold edge in the counter-clockwise direction. For example, in Fig. 5, f_2 is selected as a mate face of f_1 sharing non-manifold edge $e = (v_3, v_1)$.

For example, as shown in Fig. 5, edge $e = (v_1, v_3)$ is a non-manifold edge, and a cycle of triangle faces around e is formatted. The vertex and mate face of the three triangle faces sharing edge e can be set as shown in Tab. 1.



Figure 5: Illustration for a non-manifold edge.

Face	Vertex	Mate face
f_1	V1, V2, V3	NULL, NULL, f2
f ₂	V1, V3, V5	NULL, NULL, <i>f</i> 3
fз	V1, V3, V4	<i>f</i> i, NULL, NULL



(3) Non-manifold vertices

As shown in Fig. 6, vertex v_1 is a typical non-manifold vertex, which divides the mesh into two parts. If there are non-manifold vertices like v_1 , it is difficult to traverse an entire mesh. Currently, as we know, most data structures are unable to deal with the non-manifold vertex case in a simple way. In this paper, we firstly propose a method to employ virtual faces. By constructing one or more virtual triangle faces, a non-manifold vertex can be converted as a manifold vertex. For example, as shown in Fig. 6(a), if a virtual triangle is constructed by linking vertex v_1 , v_3 , and v_4 , the vertex v_1 will be changed into a manifold vertex. The newly created face object is not stored in the face array: *FaceArray* of the mesh object, so we call it as a virtual triangle face. The virtual triangle face is not a practical entity of a mesh model, and only plays a role to connect two separated face sets. The constructing step is as follows:



Figure 6: Illustration for a non-manifold vertex v_1 and its virtual faces to be created: (a) One virtual face and (b) Two virtual faces.

Step1. For a non-manifold vertex v_1 , all faces that are incident to v_1 are collected.

Step2. In the collected face set, if there is a face that has not any mate face relations defined above with other all collected faces, the face is an isolated face, which is supposed as f_1 ; otherwise the construction step ends.

Step3. From *f*1, select an edge (two vertices) that is incident to v_1 , which is supposed as e_1 . Search other edges that are also incident to v_1 and are border edges from the collected face set. Compute the angle between e_1 and the searched edge. If the calculated minimum angle is less than 180 degrees, the edge is selected to create a virtual face, which is supposed as e_2 .

Step4. A virtual face is created by linking v_1 and two vertices of e_1 and e_2 . Go Step2.

3.5 Retrieving topological relations

In the triangle mesh geometry, the topological relations between geometric entities can mainly be categorized into the following two kinds of relations:

(1) Incidence relations

Incidence relations denote that in a *d*-dimensional mesh the relations between different dimensional entities, which can be classified to be either upward or downward. Specifically, Incidence relations map an entity to a higher- or lower-dimensional entity. For example, in 2-dimensional triangle meshes, upward incidence relations include the vertex-edge relation, the vertex-face relation, and the edge-face relation.

(2) Adjacency relations

Adjacency relations denote that in a *d*-dimensional mesh the relations between same dimensional entities. For example, in 2-dimensional triangle meshes, the adjacency relations mainly include the face-face relation, the edge-edge relation and the vertex-vertex relation.

A mesh representation model is required to represent above topological relationships completely, and at the same time it can be performed retrieving every incident and adjacent relations in a time independent of the mesh size. The following will elaborate how the presented model can retrieve above relations effectively and efficiently for manifold and non-manifold triangle meshes.

3.5.1 Retrieving incidence relations

According to the incidence relation definition mentioned above, two simplices are incident to each other if one of them is a part of the other. We can give the following incidence relations.

- (1) One-to-any upward incidence relations
- Vertex-edge relations

The vertex-edge relation refers the relation between a vertex and its all incident edges. In the presented model, edges are not explicitly defined and only vertices are explicitly defined in face objects. As a result, the vertex-edge relation can be retrieved indirectly by retrieving its adjacent vertices.

• Vertex-face relations

The vertex-face relation can determine that a vertex is located on which face and a face includes which vertices. In the presented model, the vertex object explicitly defines a pointer, *pIncidentFace*, to the face object where it locates. By *pIncidentFace* pointer, the incident face of a vertex can be retrieved.

(2) One-to-any downward incidence relations

• Face-vertex relations

A face object includes three ordered vertices so the face-vertex relation has been explicitly defined in the face object.

• Face-edge relations

As edges are implicitly represented by the connection of vertices, the face-edge relation can be easily derived by the face-vertex relation.

3.5.2 Retrieving adjacency relations

Two kinds of adjacency relations can be retrieved as follows:

(1) Vertex adjacency relations

In the same triangle, the adjacency relation between a vertex and other two vertices is called as the vertex adjacency relation. According to their storage sequences, a vertex can derive its previous vertex and its next vertex, which is implemented in two virtual functions defined in the face object. If the queried face is a wire-edge object, due to only two valid vertices, the function to calculate the array index is different. Hence, in the *CWirEdge* class, we define two virtual functions again to get a previous vertex and a next vertex.

(2) Face adjacency relations

In the *CFace* object, three mate face pointers have been defined. By the mate face pointers of a triangle face object it is easy to retrieve its three adjacent faces.

3.5.3 One-ring neighborhood queries

The one-ring neighborhood query is an operation to traverse one-ring neighbor vertices of a given vertex, which has been widely used in the various information processing of mesh geometry. Direct access to the neighbor vertices through topological relations can improve the query efficiency greatly. The presented model can be used to the one-ring neighborhood query, including non-manifold cases. The querying procedure only performs few condition judgments and the querying time is independent on the size of meshes. The pseudo code for performing the one-ring neighborhood query of a vertex is as shown in the following Algorithm 1.

In Algorithm 1, the process of retrieving neighboring vertices is divided into two steps. Firstly, the triangle face f_i that the given vertex v_q refers is obtained. The next adjacent vertex v_n and previous adjacent vertex v_p of v_q in f_i are also obtained. Neighbor vertices are retrieved clockwise around v_q . If the newly retrieved vertex v_n is same as v_p , the retrieving process finishes. If the mate face obtained from vertex v_q and v_n is NULL, it indicates a border edge is met in the clockwise retrieving process, and it is required to continually retrieve neighbor vertices from v_p around v_q counter-clockwise. In Algorithm 1, *PrevVertex* and *NextVertex* are two virtual functions defined in the *CFace* object.

Algorithm 1: Retrieving one-ring neighborhood vertices of a given vertex

Input: vertex V_a **Output:** all one-ring neighbor vertices of vertex v_q 1: $f_i \leftarrow a$ triangle face that v_q refers; 2: Let *L* be a list of vertices neighboring v_q ; 3: $v_p \leftarrow \text{PrevVertex}(v_q)$ in f_i ; 4: $v_n \leftarrow \text{NextVertex}(v_q)$ in f_i ; Add v_h to L; 6: $f_m \leftarrow$ the mate face between vertex v_q and v_h in f_i ; 7: while $f_m \neq f_i$ and $f_m \neq$ NULL do 8: $\nu_n \leftarrow \text{NextVertex}(\nu_q) \text{ in } f_m;$ 9: Add v_n to L; 10: $f_m \leftarrow$ the mate face between vertex v_q and v_h in f_m ; 11: end do 12: if $v_n \neq v_p$ then 13: Add $v_{\rm p}$ to L; 14: $f_{\rm m} \leftarrow$ the mate face between vertex $v_{\rm q}$ and $v_{\rm p}$ in $f_{\rm i}$; 15: while $f_m \neq f_i$ and $f_m \neq \text{NULL do}$ 16: $v_{\rm p} \leftarrow \text{PrevVertex}(v_{\rm q}) \text{ in } f_{\rm m};$ 17: Add v_p to L; $f_{\rm m} \leftarrow$ the mate face between vertex $v_{\rm q}$ and $v_{\rm p}$ in $f_{\rm m}$; 18: 19: end do 20: end if 21: **return** *L*;

4 CONSTRUCTION OF TOPOLOGICAL MODELS FROM STL FILES

In reverse engineering, STL is an exchange file format for triangular mesh data. Currently STL has become a standard for data input of all types of rapid prototyping systems and has been widely used in many fields. However, A STL file just lists the geometry information of three vertices for each single triangle, resulting in a large amount of redundant data, and lacks topological information between the geometric entities. Hence, it is necessary to construct the topological representation of a STL mesh.

It is also very important for a topological representation model to be constructed effectively and efficiently. The following will describe how to construct a mesh model with the presented data structure model from STL files.

4.1 Constructing vertex objects

The construction of vertex objects needs to finish two tasks: creating a geometric vertex and setting an incidence relation with its incident face objects. The setting operation is usually performed after constructing the incident face object.

A geometric vertex can be easily created by its geometric coordinates provided in STL files. The key issue is to remove the vertex repetition. Hence, a search operation for vertices is needed. For the given geometry location data of a vertex, we need to search the whole mesh to determine if there is another vertex that coincides with the vertex to be created in the spatial location. If there is a coincidence vertex, the vertex object can be employed directly; otherwise a new vertex object will be created. The efficiency of searching vertices is very important for a complex mesh. Therefore, many methods to improve the efficiency of searching vertices were proposed in the past, such as the balanced binary tree [4], octree, and hash table methods [17]. In this paper, we employ two searching strategies as follows:

(1) Vertex search based on hash tables

The hash table method can provide a much higher efficiency for vertex search, but it needs to employ an appropriate hash function to establish connections between the keyword and the memory address (or a range of addresses). Specifically, the hash function is a function that transforms a value to an address where the value is stored. Here, we employ the hash function defined in [17] as follows:

$$Index = (DWORD)((\alpha X + \beta Y + \gamma Z)C + 0.5)\&T$$
,

where X, Y, Z are vertex coordinates, a, β , γ are coefficients of the hash function, C is a scaling coefficient set so that the full range of a 4-byte unsigned integer is used, i.e. range of the interval <0, $2^{32}-1$ >is fully used, T + 1 is the table size, & represents a modulo that is implemented as a logical operation and with the DWORD type, 0.5 is a constant that actually represents the rounding operation.

In the presented model, the container to store vertex objects is designed as a hash array and each newly created vertex object is stored in the array accorded to the index position provided by above hash function. As a result, if the coordinate of a vertex is known, it can be quickly found if there is a coincidence vertex.

(2) Making full use of adjacency relations between vertices

The presented model can provide capabilities for querying one-ring neighbor vertices, which is independent of the mesh size. Hence, when a vertex of a newly created triangle is located at an existed vertex of the mesh, we only need to search one to two levels of one-ring neighbor vertices of the located vertex. For example, as shown in Fig. 7., when a vertex of a triangle (dotted line) is located at v_1 , if other two vertices v_2 and v_3 need to search, the search only needs to compare with the one-ring neighbor vertices (v_0 and v_2) of v_1 , which reduces the search time greatly. The algorithm to search one-ring neighbor vertices has been introduced above.



Figure 7: Searching neighbor vertices of a located vertex.

4.2 Constructing mesh topological models from STL files

On the method of constructing vertex objects mentioned above, we can construct a mesh topological model from a STL file using Algorithm 2. The main operations are explained as follows:

• An initialization operation of a face object

When a face object is newly created, it needs to implement an initialization operation, in which all mate faces are initially set as NULL.

• A search operation to search duplicate vertices

Search a vertex object from the vertex hash array with a vertex coordinate. If there is a coincidence vertex, the vertex object can be employed directly; otherwise a new vertex object will be created. The searching method has introduced in above section.

• Operations creating vertex objects and setting up mate face relations

The operations from line 5 to line 15 in Algorithm 2 create a vertex object. The operations from line 15 to line 28 set the mate face relations between the newly created face object and existed face objects.

Algorithm 2: Constructing a mesh from a STL file
Input: A STL file handle F
Output: A triangle mesh object M
1: $N \leftarrow$ The number of triangle faces obtained from reading F;
2: for <i>i</i> ← 0 to <i>N</i> do
 pNewFace ← Create a new face object and initializing it;
4: $k \leftarrow 0$;
5: for <i>j</i> ← 0 to 2 do
6: $p \leftarrow a$ vertex coordinate from reading F ;
7: $v \leftarrow$ Find a vertex object with p ;
8: if $v = \text{NULL}$ then
9: $v \leftarrow$ Create a vertex object with p ;
10: <i>v.pIncidentFace</i> ← <i>pNewFace</i> ;
11: Insert <i>v</i> into the vertex hash array of <i>M</i> ;
12: end if
13: $pNewFace.Vertex[j] \leftarrow v$;
14: end do
15: for $k \leftarrow 0$ to 2 do
16: $n \leftarrow (k+1)\%3;$
17: $v_1 \leftarrow pNewFace.Vertex[k];$
18: $v_2 \leftarrow pNewFace.Vertex[n];$
19: if v_1 .pIncidentFace \neq pNewFace and v_2 .pIncidentFace \neq pNewFace then
20: $pNewFace.pMatFace[k] \leftarrow v_1.pIncidentFace;$
21: for $m \leftarrow 0$ to 2 do
22: $t \leftarrow (m+1)\%3;$
23: if v_1 .pIncidentFace.Vertex $[m] = v_2$ and v_1 .pIncidentFace.Vertex $[t] = v_1$ then
24: $v_1.pIncidentFace.pMatFace[m] \leftarrow pNewFace;$
25: end if
26: end do
27: end if
28: end do
29: Insert <i>pNewFace</i> into the face container of <i>M</i> ;
30: end do
_ 31: return <i>M</i> ;

4.3 Experiments to construct topological models

The presented data structure model has been implemented by programming on the Microsoft's Visual C++2010 and OpenGL platform, and has been applied to the reconstruction of design features for mechanical product mesh models. This section gives experiments to construct the topological model presented in this paper from a binary STL file. The experimental environment is on a computer with main frequency 3.6GHz, main memory 8.0GB, and Windows 10 operating system.

4.3.1 Experimental objects

In this paper, the experimental object is a box mechanical part that comes from the triangulation of laser scanning point clouds, as shown in Fig. 8. The triangle mesh model of the box part contains 970448 triangle facets and 2911344 vertices (including duplicate vertices), and is stored in a binary STL file. The size of the STL file is 47386 KB. Due to sampling limitation, sampling errors, and computation errors of triangulation, there are multiple mesh holes, and some discontinuous and non-manifold meshes appear in the mesh model.



Figure 8: A box part mesh model: (a) Rendering form and (b) Wireframe form.

4.3.2 Reconstruction time and memory consumption

After loading and reading the STL file to be tested, we begin to calculate the reconstruction time. The whole reconstructing process is divided into two parts. The first reconstructing process is to construct the topological model according to the steps shown in Algorithm 2, and this process takes 42.97 seconds. The second process is to check non-manifold vertices and non-manifold edges, and this process takes 3.25 seconds. If there are non-manifold vertices, the operation to construct virtual faces will be carried out. The second process can also be omitted if there is sure no non-manifold entities (e.g. from CAD meshes) in the processed meshes.

The memory resources occupied by the reconstructed model can be calculated from two memory statuses. Before constructing the topological model, a memory status function is first called, by which the current memory status can be obtained. Then, after the constructing process has finished, the memory status function is called again. The memory difference of the two statuses is the memory resources occupied by the reconstructed model. In this experiment, the memory consumption is 221.92 MB, which is close to the valuation provided below.

5 EVALUATION AND DISCUSSION

In this section, we present an evaluation for the presented model on the memory consumption, the performing time, and other comprehensive performances.

5.1 Memory consumption

Memory consumption is an important criterion to evaluate a data structure. Compared with other popular data structures, the presented model is much more compact and succinct. The following takes the famous half-edge data structure as a comparison benchmark.

In the presented model, each face object contains three pointers to its three boundary vertex objects, and three pointers to its three mate face objects. Each vertex object contains a pointer (*pIncidentFace*) to its incident face. As a result, to represent topological information, this data structure will occupy $4 \times (3+3) = 24$ bytes for a face, and $4 \times 1 = 4$ bytes for a vertex. In a manifold triangular mesh, according to Euler relations among vertices, edges and faces, the number of the triangular facet is approximately equal to two times the number of vertices, namely F = 2V; To represent topological relations of a mesh with *n* vertices, this data structure only requires $4n + 2 \times 24n = 52n$ bytes memory spaces, which demonstrates that this data structure is compact.

In the commonly used half-edge data structure, such as the Surface_mesh [23] data structure, a half-edge needs 20 bytes; a vertex needs a half-edge; an edge needs two half-edge; and a face needs a half-edge [6],[23]. As a result, for a triangle mesh with *n* vertices, the total memory consumption is: $20n + 20 \times 2 \times 3n + 20 \times 2n = 180n$ bytes. If the previous half-edge is not explicitly stored, the memory footprint can be reduced to 144 bytes/vertex. The array-based compact data structure [3],[11] is a memory-efficient variant of the half-edge data structure, and reduce the memory footprint of a half-edge to 8 bytes; but it also consumes memory at least 80 bytes/vertex. If a half-edge is encoded to an unsigned long integer, the memory footprint will be reduced to 64 bytes/vertex; but it needs to implement operations of encoding and decoding.

It is noted that this model uses pointers instead of indices of arrays. On 32-bit architectures pointers consume same amount of memory as indices, but can access entity data much more rapidly. However, if the 64-bit architecture is employed, pointers consume twice as much memory as 32-bit indices. In that case, we can replace pointers with 32-bit indices by using a new 64-bit version.

5.2 Time performance evaluation

Time performance evaluation should be taken into account from the following two aspects:

(1) Time of accessing mesh entities

As introduced in Section 3.5, the presented model can completely represent incidence relations and adjacent relations among mesh entities. All topological queries are independent of the mesh size. Of course, as edge information is implicitly represented, in some cases, an intermediate variable is needed to access entity data. For example, given a vertex, to access its next adjacent vertex on the same face, the array indices of storing this vertex should be obtained first, which may take a little time. As only at most three pointers are involved in each comparison operation, it takes very little time and is almost negligible. The algorithm of querying 1-ring neighborhoods provided above can demonstrate the efficiency of assessing mesh entities. In addition, due to using pointers, entity data can be accessed much more rapidly than the data structure using index.

Furthermore, as all vertex objects are stored in a hash array, the geometric queries for a vertex can be speeded up.

(2) The efficiency of reconstructing topological information

It takes a certain amount of time to construct a topological model from a set of discrete and scattered triangular facets. The efficiency of topological reconstruction is one of the criteria for evaluating the time performance. Theoretically, the number of variables in the model can determine the reconstructing efficiency largely. It is evident that the less the variable in the model, the higher the efficiency of reconstruction. In the presented model, there are only four variables (three mate faces in a face object and one incident face in a vertex object) for topological relations. If the traditional Half-edge data structure is employed, only a half-edge will needs four variables. Although the variable number of the compact array-based data structure is same as the presented model, the compact array-based data structure is and decode half-edges, which will take a little time. In addition, there are other factors that may affect the efficiency of reconstruction, for example, the efficiency of searching for a vertex, etc.

Name	Mesh	Vertex/ Face	Surface_mes h[23]	AMDS [3]	CGAL [10]	FBTM
Impelle r		170280/ 56760	3.26	3.08	2.81	2.85

Gear	112053/ 37351	2.69	2.45	2.02	2.16
Nozzle	36114/ 12038	1.08	1.02	0.82	0.86
Pump Cover	1503897/5 01299	30.86	30.12	24.20	26.45

 Table 2: Experimental running times of reconstructing topological models (in seconds).

Tab. 2 lists the experimental time of reconstructing four different mesh topological models from STL files using four different data structures. In Tab.2, AMDS refers the compact array-based mesh data structure presented by literature [3], FBTM refers the topological data structure presented in this paper. In the four experimental mesh models, the gear model and pump cover model come from the triangulation of laser point clouds so that the number of triangle faces and vertices is very large. CGAL data structure uses same number of topological variables as this model, but it does not involve non-manifold operations so that the running time is slightly less than FBTM model. In addition, it is noted that the reconstructing experiment of other three data structures is implemented by uniform programing according to the data structure provided by their public literature.

5.3 Comprehensive performances

A mesh data structure is comprehensive if it can perform every incident, adjacent, or classification query in a time independent of mesh size [3]. It can be seen that from the analysis provided in Section 3.5, the presented data model is comprehensive. In addition, the presented model has the following advantages:

Much more flexibilities and extensibilities

Object-oriented representation can provide much more flexibilities, extensibilities, and usability for various mesh applications. In the presented model, each mesh face or vertex is described as an object instance of the respective geometric entity class. According to application requirements, users can simply derive their subclasses to create the user-defined mesh object, which makes it easy to implement hybrid meshes consisting of different types of mesh faces. As a result, various attributes, such as texture or color attributes, can be easily appended to the face object. In addition, the face object instance and the vertex object instance can be flexibly accessed.

• Uniformly representing and processing manifold meshes and non-manifold meshes

Currently, most of existing mesh data structures have been less than satisfactory in representing and processing non-manifold meshes. By defining the *CWireEdge* class and creating virtual face objects, the presented model can uniformly represent and process mesh boundary and non-manifold meshes. By contrast, in order to cope with boundary edges of meshes many other data structures, such as Compact array-based data structure, an additional array is needed [3], which adds complexity of models.

6 CONCLUSIONS

A comprehensive and compact topological representation is necessary for information processing of complex triangle meshes. Meanwhile, it is required to uniformly represent manifold meshes and nonmanifold meshes with more flexibilities and extensibilities. In this paper, a novel compact face-based topological representation for triangle meshes is presented. The presented model makes full use of the semantic relations among faces, edges, and vertices. As a result, the memory footprint to represent topological information is reduced. Compared with the general edge-based data structure, the presented model has much more flexibility and extensibility. In addition, the presented model can uniformly represent and process hybrid meshes, manifold meshes, mesh boundaries, and non-manifold meshes. The queries of mesh entities are independent on the size of meshes, which ensures the time performance of the presented model. Furthermore, a reconstruction method and experiment for STL format meshes has been carried out, which verifies that the presented model is feasible and effective.

In the future work, we will further improve the topological model to meet various requirements in reverse engineering applications, which includes improving the capability to implement parallel computations based on GPUs, the interoperability with other models, and the friendliness with the application interface.

ACKNOWLEDGEMENTS

This work is supported by the National Science Foundation of China (Grant No. 51775081 and 51375069). The authors thank the anonymous reviewers for their helpful suggestions on this study.

Yingzhong Zhang, http://orcid.org/0000-0003-3584-7239

REFERENCES

- [1] Aleardi L. C.; Devillers O.: Explicit array-based compact data structures for triangulations, Algorithms and Computation, ISAAC 2011. Lecture Notes in Computer Science, Springer, Berlin, Heidelberg, 2011. <u>https://doi.org/10.1007/978-3-642-25591-5_33</u>
- [2] Aleardi L. C.; Devillers O.; Rossignac J.: ESQ: Editable SQuad representation for triangle meshes, 25th SIBGRAPI Conference on Graphics, Patterns and Images, IEEE, Ouro Preto, Brazil, 2012.
- [3] Alumbaugh, T. J.; Jiao, X.: Compact array-based mesh data structures, Proceedings of the 14th International Meshing Roundtable, Springer, Berlin, Heidelberg, 2005.
- [4] Andersson A.: General Balanced Trees, Journal of Algorithms, 30(1), 1999, 1-18. https://doi.org/10.1006/jagm.1998.0967
- [5] Baumgart B. G.: Winged-edge polyhedron representation, Stanford University, Computer Science Department, Stanford, CA, 1972.
- [6] Botsch, M.; Kobbelt, L.; Pauly, M.; Alliez, P.; Lévy, B.: Polygon mesh processing, A K Peters, Natick, Massachusetts, 2010.
- [7] Botsch M.; Steinberg S.; Bischoff S.; Kobbelt L.: OpenMesh A generic and efficient polygon mesh data structure, Proceedings of OpenSG Symposium, 2002.
- [8] Campagna, S.; Kobbelt, L.; Seidel, H.-P.: Directed edges—A scalable representation for triangle meshes, Journal of Graphics Tools, 3(4), 1998, 1-12. <u>https://doi.org/10.1080/10867651.1998.104</u> 87494

- [9] Canino D.; Floriani L. D.; Weiss K.: IA*: An adjacency-based representation for non-manifold simplicial shapes in arbitrary dimensions, Computers & Graphics, 35 (3), 2011, 747–753. <u>https://doi:10.1016/j.cag.2011.03.009</u>
- [10] CGAL, Computational Geometry Algorithms Library, <u>http://www.cgal.org</u>, The CGAL Project.
- [11] Dyedov, V.; Ray. N.; Einstein, D.; Jiao, X.; Tautges, T. J.: AHF: Array-based half-facet data structure for mixed-dimensional and non-manifold, Engineering with Computers, 31(3), 2015, 389-404. <u>https://doi.org/ 10.1007/s00366-014-0378-6</u>
- [12] Feito, F. R.; Ogayar, C. J.; Segura, R. J.; Rivero, M. L.: Fast and accurate evaluation of regularized Boolean operations on triangulated solids, Computer-Aided Design, 45(3), 2013, 705-716. <u>https://doi.org/10.1016/j.cad.2012.11.004</u>
- [13] Feng X.; Wang Y.; Weng Y.; Tong Y.: Compact combinatorial maps: A volume mesh data structure, Graph Models, 75(3), 2013, 149-156. <u>http://dx.doi.org/10.1016/j.gmod.2012.10.001</u>
- [14] Floriani L. D.; Magillo P.; Puppo E.; Sobrero D.: A multi-resolution topological representation for non-manifold meshes, Computer-Aided Design, 36 (2), 2004, 141–159. <u>doi:10.1016/S0010-4485(03)00058-7</u>
- [15] Floriani L. D.; Greeneldboyce D.; Hui A.: A data structure for non-manifold simplicial dcomplexes, Eurographics Symposium on Geometry Processing, Nice, France, 2004.
- [16] Gurung T.; Luffel M.; Lindstrom P.; Rossignac J.: Zipper: A compact connectivity data structure for triangle meshes, Computer Aided Design, 45(2), 2013, 262-269. <u>https://doi:10.1016/j.cad.2012.10.009</u>
- [17] Hrádek, J.; Kuchar, M.; Skala, V.: Hash functions and triangular mesh reconstruction, Computers & Geosciences, 29(6), 2003, 741-751. <u>https://doi.org/10.1016/S0098-3004(03)00037-2</u>
- [18] Kallmann M.; Thalmann D.: Star-Vertices: A compact representation for planar meshes with adjacency information, Journal Graph Tools, 6(1), 2001, 7-18. <u>https://doi.org/10.1080/10867651.2001.10487533</u>
- [19] Kraemer P.; Untereiner L.; Jund T., Thery S.; Cazier D.: CGoGN: n-dimensional meshes with combinatorial maps, Proceedings of the 22nd International Meshing Roundtable, Springer, Berlin, Heidelberg, 2014.
- [20] Lage M.; Martha L. F.; de Almeida J. P. M.; Lopes H.: IBHM: index-based data structures for 2D and 3D hybrid meshes, Engineering with Computers, 27(4), 2017, 727-744. <u>https://doi.org/10.1007/s00366-015-0395-0.</u>
- [21] Luffel M.; Gurung T.; Lindstrom P.: Grouper: A compact, streamable triangle mesh data structure, IEEE Transactions on Visualization and Computer Graphics, 20(1), 2014, 84-98. <u>https://doi.org/10.1109/TVCG.2013.81</u>
- [22] Mäntylä, M.: An introduction to solid modeling, Computer Science Press, 1988.
- [23] Sieger, D.; Botsch, M.: Design, implementation, and evaluation of the surface mesh data structure, Proceedings of the 20th International Meshing Roundtable, Springer, Berlin Heidelberg, 2011.
- [24] Szilvasi-Nagy M.; Mátyási G.: Analysis of STL Files, Mathematical and Computer Modelling, 38(7-9), 2003, 945-960. <u>https://doi.org/10.1016/S0895-7177(03)90079-3</u>
- [25] The VCG Library, <u>http://vcg.sourceforge.net</u>, Visual Computing Lab of the Italian National Research Council.
- [26] Várady, T.; Martin R. R.; Cox J.: Reverse engineering of geometric models-an introduction, Computer-Aided Design, 29(4), 1997, 255-68. <u>https://doi.org/10.1016/S0010-4485(96)00054-</u>1
- [27] Weiler K.: The radial edge data structure: a topological representation for non-manifold geometric boundary modeling, Geometric modeling for CAD applications, Amsterdam, North-Holland, 1988.
- [28] Werghi N.: Assessing the Regularity of 3D Triangular Mesh Tessellation Using a Topological Structured Pattern, Computer-Aided Design and Applications, 8(5), 2011, 633-648. <u>https://doi.org/10.3722/cadaps.2011.633-648</u>