A B-REP DATA STRUCTURE FOR POLYGONAL MESHES

Frutuoso G. M. Silva   Abel J. P. Gomes
IT – Networks and Multimedia Group
Department of Computer Science and Engineering
University of Beira Interior
Rua Marques d'Avila e Bolama, 6201-001 Covilhã, Portugal
fsilva@di.ubi.pt, agomes@di.ubi.pt

Abstract
This paper introduces a new b-rep (boundary representation) data structure, called AIF (Adjacency and Incidence Framework). It is concise and enables fast access to topological information. Its conciseness results from the fact that it is an orientable, but not an oriented, data structure, i.e. an orientation can be topologically induced as necessary in many applications. It is an optimal C_4^9 data structure for polygonal meshes, which means that a minimal number of direct and indirect accesses are required to retrieve adjacency and incidence information from it. Besides, the AIF data structure may accommodate general polygonal meshes, regardless of whether or not they are triangular and manifold.

Keywords
Geometric modeling, geometric data structures, boundary representations, polygonal meshes.

1. INTRODUCTION

We know that 3D object models are now more complex than before. Their complexity is in part due to more and more demanding interactive 3D design tools, data acquisition technologies, geometric processing, and graphics workstation capabilities. In order to cope with their complexity, 3D objects are approximated by polygonal meshes as usual in geometric systems and applications such as, for example, multiresolution, virtual reality and rendering of solids and surfaces.

Different applications have specific needs. Some applications need more storage space than speed, some need faster retrieval of adjacency and incidence information. Unfortunately, these retrieval operations are not as fast and efficient as necessary for current applications, in particular for meshes with a huge amount of cells (say, vertices, edges, and faces). This is partly due to the design of the data structures somehow.

This paper presents a new data structure, called AIF (Adjacency and Incidence Framework) together with a single query operator to retrieve topological information. It was designed to satisfy three major important requirements, namely: *responsiveness*, *conciseness* and *generality*.

This paper is organized as follows. Related work appears in Section 2. Section 3 describes the AIF representation. Section 4 presents its companion query operator, called mask operator, for fast retrieval of adjacency and incidence information. The representation of non-manifold meshes is discussed in Section 5. Section 6 presents the AIF data structure. Section 7 describes the orientation mechanism used in the AIF. The mask operator implementation appears in Section 8. A comparison of several data structures, including the AIF data structure, appears in Section 9. At last, Section 10 draws some conclusions and future work.

2. RELATED WORK

Many geometric data structures do not support fast and efficient traversal algorithms for large meshes. For example, finding a cell in the cell-tuple data structure [Brisson93] is a time-consuming operation, in particular for large meshes, because that requires processing all cell-tuples.

Some data structures represent a triangular mesh by a set of faces, each face consisting of a tuple of vertices. This makes it difficult to design and implement fast algorithms to retrieve adjacency and incidence data. For example, finding the set of faces incident at a given vertex is also a time-consuming operation, because that requires traversing all the faces stored in the data structure.

A way to speed up search algorithms is to use oriented boundary representation data structures (Winged-Edge [Baumgart72], Half-Edge [Mantyla88] or Radial Edge

[Weiler88]), but they need additional storage space for oriented cells. In a way, these oriented cells are redundant. For example, in the Radial Edge data structure, each face has two associated loops, one for each face side; hence, three faces incident along an edge require six oriented edges.

Lee and Lee [Lee01] proposed a new data structure, called Partial Entity structure that requires half the storage needed for the Radial Edge data structure.

Kallmann and Thalmann [Kallmann01] designed a concise data structure, called Star-Vertex, to store polygonal meshes. It is based on the incidence information around a vertex. Despite its conciseness, retrieving adjacency and incidence information is a slow task because no explicit information about edges and faces is kept.

Other data structures were designed only for representing *n*-dimensional simplicial meshes. For example, Directed-Edges [Campagna98] and Tri-Edges [Loop00] represent oriented 2-dimensional triangular meshes. More generally, PSC (Progressive Simplicial Complexes) [Popovic97] data structure represents orientable (i.e. not necessarily oriented) *n*-dimensional simplicial complexes. However, the absence of explicit oriented simplexes in PSCs or a geometric orientation mechanism for simplexes is troublesome in rendering. In fact, unlike progressive meshes proposed by Hoppe [Hoppe98], the PSC data structure avoids explicitly storing surface normals at vertices. Instead, it makes usage of smoothing group fields for different materials as used by Wavefront Technologies.

More recently, Floriani *et al.* [Floriani02] introduced a new data structure for *non-manifold* triangular meshes that explicitly encodes the vertices (V) and triangles (T). This data structure stores the following adjacency and incidence relations $V \to V$, $V \to T$, $T \to V$ and $T \to T$. It is concise but accessing to the adjacency and incidence information associated to edges is slow because edges are not explicitly represented.

This paper introduces the AIF data structure for polygonal meshes. These meshes need not be manifold and triangular. Thus, it may accommodate simplicial complexes and, more generally, cell complexes with or without dangling cells. Topologically speaking, it is an orientable, but not oriented, data structure, i.e. it does not store oriented cells. A minimal number of accesses are needed to retrieve adjacency and incidence information from it. The result is a more responsive, concise and general data structure.

3. ADJACENCY AND INCIDENCE FRAMEWORK

Let us review some notions about topological relations between cells of a cell complex. It is said that a cell x is adjacent to a cell y (symbolically, $x \prec y$) if x is in the frontier of y (or, equivalently, $fr(y) \cap x \neq \emptyset$) and the dimension of x is less than the dimension of y; equivalently, y is incident on x (symbolically, $y \succ x$). For example, a bounding vertex of an edge is said to be adjacent to such an edge, but there may be many edges incident at the same vertex. The adjacency (incidence) relation is transitive, i.e. if $v \prec e$ and $e \prec f$ then $v \prec f$.

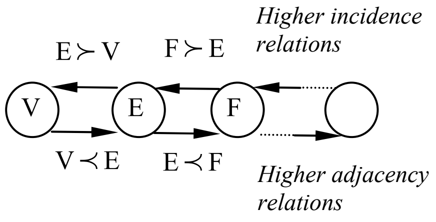

Figure 1: AIF diagram.

AIF data structure *explicitly* embodies a $C_{2n}^{(n+1)^2}$ representation, i.e. it represents $2n$ out of $(n+1)^2$ adjacency and incidence relations between cells of a n-dimensional cell complex (Figure 1). In particular, it represents 4 out of 9 relations between cells of a 2-dimensional cell complex, namely: two basic adjacency relations, $V \prec E$ and $E \prec F$, and their inverse relations or incidence relations, $E \succ V$ and $F \succ E$. These four basic relations can be combined to form the nine adjacency relations introduced by Weiler [Weiler88]. For example, the Weiler relations $\mathrm{V} \to \mathrm{F}$ and $\mathrm{E} \to \mathrm{E}$ can be obtained as follows:

$$\mathrm{V} \to \mathrm{F} = (\mathrm{V} \to \mathrm{E}) \circ (\mathrm{E} \to \mathrm{F}) = (\mathrm{E} \succ \mathrm{V}) \circ (\mathrm{F} \succ \mathrm{E})$$

$$\mathrm{E} \to \mathrm{E} = (\mathrm{V} \to \mathrm{E}) \circ (\mathrm{V} \to \mathrm{E}) = (\mathrm{V} \prec \mathrm{E}) \circ (\mathrm{E} \succ \mathrm{V})$$

According to Ni and Bloor [Ni94], these four basic relations form the best representation in the class C_4^9 in terms of information retrieval performance, i.e. it requires a minimal number of direct and indirect accesses to the data structure to retrieve those four explicit topological relations and the remaining five implicit topological relations, respectively. A direct access query involves a single call to the query operator, while an indirect access requires two or more calls to the query operator, i.e. a compound query.

The AIF has been designed to keep the essential four topological relations of the representation C_4^9 (and their higher-dimensional counterparts). Basically, a 2-dimensional mesh in the AIF data structure is defined by the triple $M=\{V,E,F\}$, where V is a finite set of vertices, E is a finite set of edges, and F is a finite set of simply connected faces.

A *vertex* $v \in V$ is represented by a non-ordered k-tuple of its incident edges, i.e. $v=\{e_1,e_2,e_3,\ldots,e_k\}$, where e_i ($i=1\ldots k$) is an incident edge at v; hence, $\mathrm{E} \succ \mathrm{V}$ relation is embedded in the AIF.

An *edge* $e \in E$ is a pair of tuples $e=\{\{v_1, v_2\},\{f_1,f_2,\ldots,f_k\}\}$, where the first tuple contains the vertices v_1 and v_2 bounding e, and the second tuple consists of the faces $f_1,f_2,\ldots,f_k$ incident on e. So, we have $\mathrm{V} \prec \mathrm{E}$ and $\mathrm{F} \succ \mathrm{E}$ relations.

A *face* $f \in F$ is represented by a k-tuple of its bounding (adjacent) edges, i.e. $f=\{e_1,e_2,e_3,\ldots,e_k\}$, where each e_i

(i=1…k) is an edge bounding f; hence, the E $\prec$ F relation.

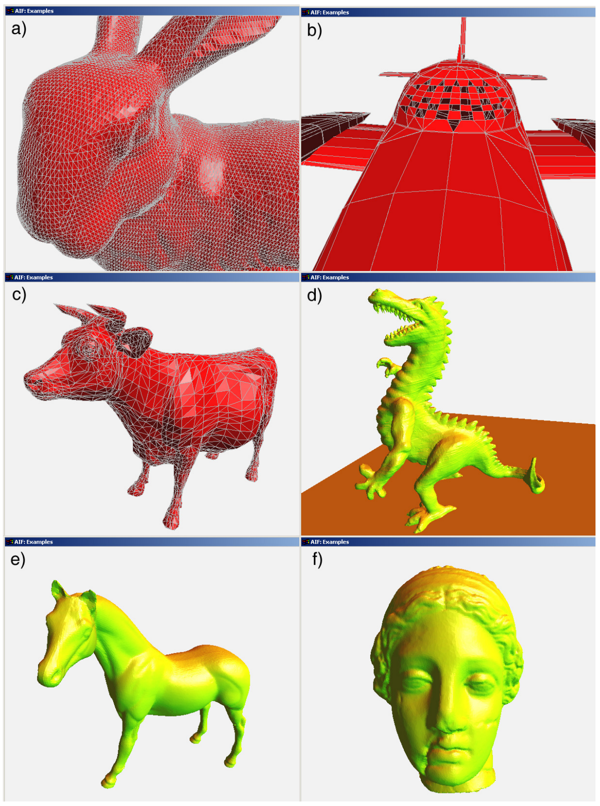

Figure 2: AIF polygonal meshes.

With the previous four basic adjacency and incidence relations embedded in the AIF data structure we can represent manifold (Figure 2 (a)(c)(d)(e)(f)) and non-manifold (Figure 2 (b)) meshes, regardless of whether they are triangular or not. In Figure 2 (b), we have a non-manifold mesh because some plane cockpit faces (say, dark regions) are missing. Note that a AIF mesh needs not be triangular; Figure 2 (a)(c)(d)(e)(f) depicts triangular meshes, but not Figure 2 (b).

4. TOPOLOGICAL QUERY OPERATOR

Run-time meshing applications need fast query and retrieval algorithms to efficiently identify cells adjacent to or incident at/on other cells. For example, in multiresolution area, this is essential to quickly refine and simplify meshes. In fact, both splitting and collapsing operations need topological information to refine and simplify a mesh, respectively.

Let us then pay attention to the AIF incidence scheme of a mesh. It can be described in terms of a set of cell-tuples T={v_i, e_j, f_k}, where f_k is a face incident on an edge e_j (symbolically, $f_k \prec e_j$) and e_j is incident at a vertex v_i ($e_j \prec v_i$); conversely, v_i is adjacent to e_j ($v_i \prec e_j$) and e_j is adjacent to f_k ($e_j \prec f_k$). For example, the tetrahedron depicted in Figure 3 has the following incidence scheme:

(v_1, e_1, f_2) (v_2, e_1, f_2) (v_3, e_2, f_1) (v_4, e_3, f_1)

(v_1, e_1, f_3) (v_2, e_1, f_3) (v_3, e_2, f_3) (v_4, e_3, f_2)

(v_1, e_2, f_3) (v_2, e_6, f_2) (v_3, e_4, f_3) (v_4, e_5, f_1)

(v_1, e_2, f_1) (v_2, e_6, f_4) (v_3, e_4, f_4) (v_4, e_5, f_4)

(v_1, e_3, f_2) (v_2, e_4, f_3) (v_3, e_5, f_1) (v_4, e_6, f_2)

(v_1, e_3, f_1) (v_2, e_4, f_4) (v_3, e_5, f_4) (v_4, e_6, f_4)

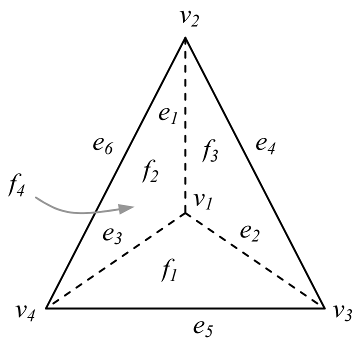

Figure 3: A *manifold* mesh.

The incidence scheme of a mesh can be considered as a data structure on its own. It is known as the cell-tuple data structure [Brisson93]. The AIF data structure has then the same adjacency and incidence descriptive power as the cell-tuple data structure, but it is more concise and less time-consuming in retrieving topological information. This is due to the fact that AIF consists of a set of cells (not a set of cell-tuples), with each cell representing one or two *explicit* topological relations:

- A *vertex* is defined in terms of its incident edges (i.e. the E $\succ$ V relation);
- An *edge* is defined in terms its adjacent vertices and incident faces (i.e. V $\prec$ E and F $\succ$ E relations);
- A *face* is defined in terms of its adjacent edges (i.e. the E $\prec$ F relation).

Retrieving topological information from the AIF data structure is done by using a single query operator, called mask operator. It is defined by $\rhd\!\!\blacktriangleleft_d$:V×E×F→C, being C=V∪E∪F the union of the set of vertices V, the set of edges E, and the set of faces F, such that $\rhd\!\!\blacktriangleleft_d(v_i,e_j,f_k)=\{c_l^d\}$, i.e. a set of d-dimensional cells.

The arguments of $\rhd\!\!\blacktriangleleft_d$ are cells in the set V×E×F. These arguments establish adjacency and incidence relations between them. Let x a cell argument of dimension n. If x=NULL, there is no restrictions on the cells of dimension n, i.e. all the n-cells are being considered. If x≠NULL, all n-cells are being considered, except x itself.

So, the $\rhd\!\!\blacktriangleleft_d$ operator returns all the d-cells incident on the (d-1)-cells and adjacent to the (d+1)-cells, after considering the restrictions imposed by non-NULL arguments.

Let us show how the query operator works in conjunction with the AIF data structure for the manifold mesh in Figure 3:

1. $\rhd\!\!\blacktriangleleft_1(v_1,\text{NULL},\text{NULL})=\{e_1,e_2,e_3\}$, directly returns all edges incident at v_1.
2. $\rhd\!\!\blacktriangleleft_2(v_1,\text{NULL},\text{NULL})=\{f_1,f_2,f_3\}$, indirectly returns all faces incident at v_1. This requires an intermediate call to $\rhd\!\!\blacktriangleleft_1(v_1,\text{NULL},\text{NULL})$ to return all edges e_1, e_2, e_3 incident at v_1. Then, the operator $\rhd\!\!\blacktriangleleft_2(\text{NULL},e_i,\text{NULL})$ is called for each edge e_i (i=1,2,3) in order to compute faces incident on e_i and v_1.
3. $\rhd\!\!\blacktriangleleft_0(\text{NULL},e_1,\text{NULL})=\{v_1,v_2\}$, directly returns bounding vertices of e_1.
4. $\rhd\!\!\blacktriangleleft_2(\text{NULL},e_1,\text{NULL})=\{f_2,f_3\}$, directly returns faces incident on e_1.
5. $\rhd\!\!\blacktriangleleft_0(\text{NULL},\text{NULL},f_1)=\{v_1,v_3,v_4\}$, indirectly returns all vertices bounding f_1. This requires an intermediate call to $\rhd\!\!\blacktriangleleft_1(\text{NULL},\text{NULL},f_1)$ to first determine all edges bounding f_1. Then, the operator $\rhd\!\!\blacktriangleleft_0(\text{NULL},e_i,\text{NULL})$ is called for each edge e_i in order to determine vertices bounding e_i and f_1.
6. $\rhd\!\!\blacktriangleleft_1(\text{NULL},\text{NULL},f_1)=\{e_2,e_3,e_5\}$, directly returns all edges bounding f_1.
7. $\rhd\!\!\blacktriangleleft_2(v_3,e_1,\text{NULL})=\{f_3\}$, directly returns faces incident on e_1 and at v_3.
8. $\rhd\!\!\blacktriangleleft_1(v_1,e_2,f_1)=\{e_3\}$, directly returns all edges bounding f_1 and incident at v_1, with the exception of e_2.
9. $\rhd\!\!\blacktriangleleft_0(v_1,\text{NULL},\text{NULL})=\{v_2,v_3,v_4\}$, returns all vertices neighboring v_1.

Note that the $\rhd\!\!\blacktriangleleft_d$ query operator does not handle all the data structure constituents at once. It handles a mesh locally by using the adjacency and incidence relations stored in the data structure. Thus, its time performance holds independently of the mesh size. This is very important for handling large meshes at interactive rates.

5. NON-MANIFOLD MESHES

AIF data structure can also represent non-manifold polygonal meshes. For example, we have three non-manifold objects in Figure 4. The object (a) has an edge e_1 without incident faces; so, its representation in the AIF data structure is as follows:

$$e_1=\{\{v_1,v_2\},\{\}\}$$

On the other hand, every edge of the object (b) has a single face incident on it; so, for example, the representation of e_1 of the object (b) is the following:

$$e_1=\{\{v_1,v_2\},\{f_1\}\}$$

The edge e_1 of the object (c) has three incident faces, f_1, f_2, f_3; so, its representation is given by:

$$e_1=\{\{v_1,v_2\},\{f_1,f_2,f_3\}\}$$

Thus, AIF data structure supports non-manifold meshes because we can distinguish between a manifold object and a non-manifold object by inquiring the AIF data structure through the query operator. In particular, we can identify the following situations:

- *Isolated vertex*. An isolated vertex has no incident edges.
- *Dangling edge*. A dangling edge has no incident faces.
- *Boundary edge*. A boundary edge has only a single incident face.
- *Cut-vertex*. A cut-vertex has at least 4+2*n* (*n*=0...k) incident boundary edges. A cut-vertex splits a mesh component into two or more mesh components; for example, the vertex v_1 in Figure 4 (b) is a cut-vertex.

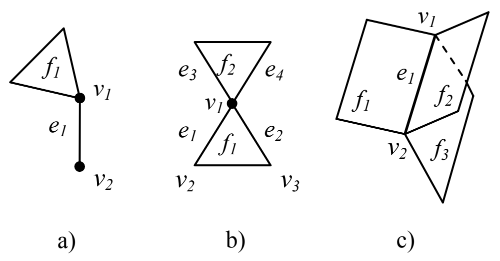

Figure 4: Non-manifold objects.

6. DATA STRUCTURE

The AIF data structure represents a mesh (or cell complex) consisting of a set of cells (vertices, edges and faces). A C++ class codes each cell type, but all cell types could be implemented as a single C++ class – as needed for *n*-dimensional meshes – because every *n*-dimensional cell is seen as a set of incident (*n+1*)-dimensional set and a set of adjacent (*n-1*)-dimensional cells. The AIF data structure is then as follows:

```
class Point {
  double x,y,z; // point coordinates
}
class Vertex {
  int id;        // vertex id
  evector li;    // incident edges
  Point *pt;     // geometry
}
class Edge {
  int id;         // edge id
  Vertex *v1,*v2;//adjacent vertices
  fvector li;     // incident faces
}
class Face {
  int id;         // face id
  evector la;     // adjacent edges
  Point *nf;      // face normal
}
class Mesh {
  int id;       // mesh id
  vvector vv;   // vector of vertices
  evector ev;   // vector of edges
  fvector fv;   // vector of faces
}
```

Note that AIF data structure is not topologically oriented provided that it does not include any oriented cells. It is geometrically oriented by the face normal `nf` in the class `Face`. That is, we use a geometric mechanism to induce an orientation on the mesh that is described in the next section.

7. ORIENTATION MECHANISM

Orientation is important for rendering geometric models (e.g. 2-dimensional meshes). It happens that AIF data structure is not an *oriented* topological data structure viewing that it does not include any oriented cells. Nevertheless, AIF is an *orientable* data structure. This means that an orientation can be topologically induced on the mesh. By inducing an orientation on a mesh we mean to induce a consistent orientation (either clockwise or counterclockwise) for all its faces.

Inducing an orientation on a face *f* requires traversing its frontier by alternately applying the operators $\rhd\!\blacktriangleleft_0$ and $\rhd\!\blacktriangleleft_1$ to the cells (edges and vertices) bounding it. The operator $\rhd\!\blacktriangleleft_0$ returns the next vertex in the frontier of *f*, and the operator $\rhd\!\blacktriangleleft_1$ returns the next edge bounding *f*. This means that we have to use as many times the query operator as the number of cells bounding *f*. After finishing the topological orientation process of *f*, a vector (e.g. geometric orientation) normal to it is calculated and stored in the data structure.

Then, we take the faces neighboring *f* to induce the same orientation on them. The faces neighboring *f* are retrieved by the query operator $\rhd\!\blacktriangleleft_2$. For example, in Figure 5, some faces are already oriented clockwise. Applying this orientation-inducing process to all faces of a mesh, we end up by having a clockwise-oriented mesh.

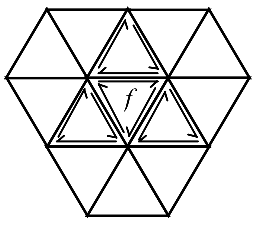

Figure 5: Orientation Mechanism.

We should be careful about non-manifold meshes because some faces risk to be left without orientation after such an orientation-inducing process. For example, in Figure 2 (b), the sub-mesh concerning the plane cockpit has some faces missing; so many of the remaining faces cannot be oriented properly. In this case, we have to re-apply the orientation process to them individually, though keeping a consistent rendering of the mesh.

8. MASK OPERATOR IMPLEMENTATION

We use the mask operator to retrieve adjacency and incidence information from the AIF data structure. Remember that the relations $V \prec E$, $E \prec F$, $E \succ V$ and $F \succ E$ are stored explicitly in AIF data structure; $V \succ E$ is represented by the vertices `*v1` and `*v2` in the class `Edge`, $E \prec F$ is represented by the dynamic vector `la` in the class `Face`, $E \succ V$ is represented by the dynamic vector `li` in the class `Vertex`, and $F \succ E$ is represented by the dynamic vector `li` in the class `Edge`.

The mask operator retrieves the remaining five topological relations. It retrieves the relations $F \succ V$ and $V \prec F$ in the following way:

```
MaskOperator(2,v,NULL,NULL) //V->F
//Input: index, vertex, edge, face
Begin
  For each e incident at v do
     For each f incident on e do
        Add f to result
   return set of faces
End
MaskOperator(0,NULL,NULL,f) //F->V
//Input: index, vertex, edge, face
Begin
  For each e adjacent to f do
     For each v adjacent to e do
        Add v to result
   return set of vertices
End
```

The relations $V \to V$, $E \to E$ and $F \to F$ are inferred as follows:

```
MaskOperator(0,v,NULL,NULL) //V->V
//Input: index, vertex, edge, face
Begin
  For each e incident at v do
     For each $v_i$ adjacent to e do
        If ($v_i$ <> v)
           Add $v_i$ to result
   return set of vertices
End
MaskOperator(1,NULL,e,NULL) //E->E
//Input: index, vertex, edge, face
Begin
  For each v adjacent to e do
     For each $e_i$ incident at v do
        If ($e_i$ <> e)
           Add $e_i$ to result
   return set of edges
End
MaskOperator(2,NULL,NULL,f) //F->F
//Input: index, vertex, edge, face
Begin
  For each e adjacent to f do
     For each $f_i$ incident e do
        If ($f_i$ <> f)
           Add $f_i$ to result
   return set of faces
End
```

9. STORAGE AND ACCESSING EFFICIENCY COMPARISONS

9.1. Storage Cost

Table 1 compares several data structures in terms of storage cost for a triangular mesh with *n* vertices. The first column, "**Mesh type**", refers to the type of a mesh, which can be either triangular (Δ) or polygonal (Any). The "**Non-manifold**" column indicates whether a data structure supports non-manifold meshes or not. The "**Bytes/Δ**" column shows the number of bytes needed for each triangle in memory. The variable k stands for the vertex degree, i.e. the number of edges incident at a given vertex.

Data structures	Mesh type	Non-manifold	Bytes/Δ
Triangle List	Δ	Yes	18
Star-Vertex	Any	Yes	$10+4k$
Progressive Meshes	Δ	No	33
Tri-Edges	Δ	No	35
AIF	Any	Yes	$29+2k$
PSC	Δ	Yes	$37+2k$
Directed-Edges	Δ	Yes	44
Half-Edge	Any	No	46
FastMesh	Δ	No	53
Radial Edge	Any	Yes	56
Winged-Edge	Any	No	60

Table 1: Storage comparisons.

The Triangle List data structure is a concise data structure because it only stores vertices and faces. Basically, it is only used for visualization purposes provided that graphical cards are optimized for triangles.

The Star-Vertex is another concise data structure that stores $10+4k$ bytes per triangle. It only stores vertices and their neighboring vertices. Every neighboring vertex comes with an index that facilitates traversal of the data structure, e.g. traversing the vertices bounding a face counterclockwise. It also supports general meshes, but the adjacency information retrieval is time-consuming. This is explained by the fact that neither edges nor faces are explicitly represented in the data structure. Consequently, six adjacency/incidence relations are "**Not defined**" (Table 2).

Progressive Meshes [Hoppe98] and Tri-Edge [Loop00] data structures need 33 and 35 bytes per triangle, respectively. Thus, the first four data structures in Table 1 are all more concise than the AIF data structure. But, with the exception of the Star-Vertex data structure, they were designed only for manifold triangular meshes.

As far as the AIF data structure is concerned, a mesh approximately requires 35, 37, 39, 41 bytes per triangle in memory for k=3,4,5 and 6, respectively. This complies with the Euler formula for 2-dimensional triangular meshes, according which a mesh with n vertices has about m faces ($m=2n$) and e edges ($2m=3e$). Assuming that floats and pointers are 4 bytes each, the corresponding runtime space cost is $(3\times4+4k)n=(12+4k)n=(6+2k)m$ bytes for vertex coordinates, $(2\times4+2\times4)e=16e\approx11m$ bytes for edges (references to vertices and faces), and $(3\times4)m=12m$ bytes for faces (references to edges).

Besides, AIF data structure does not possess oriented cells (e.g. half-edges or directed edges). It is then more concise than traditional b-rep data structures (e.g. Winged-Edge [Baumgart72]) and their oriented variants (e.g. Half-Edge [Mantyla88], Directed-Edges [Campagna98], FastMesh [Pajarola01], and Radial Edge [Weiler88]). The remaining data structures (e.g. PSC [Popovic97]) in Table 1 need more memory space than the AIF data structure, and some of them cannot represent polygonal or non-manifold meshes.

9.2. Retrieval of Adjacency and Incidence Information

Table 2 compares several data structures in terms of efficiency in retrieving adjacency and incidence information from a triangular mesh. This has to do with the number of relations that are explicitly and implicitly stored in a mesh data structure, i.e. how many accesses are necessary to retrieve *all* adjacency and incidence information from it.

Data structures	Not defined	Class
AIF	0	C_4^9
PSC	0	C_4^9
Winged-Edge	0	C_2^9
Radial Edge	0	C_2^9
Half-Edge	0	C_2^9
FastMesh	3	C_2^9
Directed-Edges	3	C_2^9
Triangle List	6	C_2^9
Star-Vertex	6	C_2^9
Progressive Meshes	6	C_2^9
Tri-Edge	6	C_2^9

Table 2: Accessing comparisons.

In Table 2, the "**Class**" column gives us the number of adjacency and incidence relations explicitly represented in data structure. For example, a data structure C_4^9 has four out of nine adjacency and incidence relations explicitly represented. If an adjacency/incidence relation is explicitly represented in a data structure, we can directly access to its associated set of cells without additional processing cost. The remaining five adjacency and incidence relations of a data structure C_4^9 are represented implicitly or are not defined at all. By an implicitly represented relation we mean a relation that can inferred from those explicitly represented in a 2-dimensional mesh data structure having cells of dimension from 0 up to 2. On the other hand, a not-defined relation has to be also inferred, but first we have to compute missing cells of specific dimensions. For example, the Star-Vertex data structure for

2-dimensional meshes has no edges and faces. Consequently, six adjacency and incidence relations are missing (see column "**Not defined**" in Table 2).

Looking again at Table 2, we conclude that the AIF and PSC are the best data structures in terms of accessing efficiency. In fact, both AIF and PSC data structures belong to the class C_4^9 (4 direct accesses and 9-4=5 indirect accesses to retrieve adjacency and incidence information).

The remaining data structures in Table 2 belong to the class C_2^9 (2 direct accesses and 9-2=7 indirect accesses to retrieve adjacency and incidence information). For example, the FastMesh data structure has no vertices in the data structure. Thus, three adjacency and incidence relations involving vertices are missing, but they can be inferred with substantial computational cost anyway. The remaining six adjacency and incidence relations involving edges and faces can be retrieved more easily.

9.3. Time Performance

The runtime performance tests were performed on a PC equipped with a 1.6GHz Intel Pentium 4, 768MB RAM, a GeForce 4 graphics card with 64MB, and running Windows 2000 OS.

Table 3 shows some results for different AIF models (Figure 2).

Models	#F	Loading	Orientation	GL_P	GL_L
Cow	5804	0.180	0.060	0.01	0.01
CessnaNM	3946	0.180	0.040	0.01	0.01
CessnaT	13546	0.491	0.120	0.03	0.01
Dragon	50761	1.622	0.631	0.11	0.02
Bunny	69473	2.674	1.001	0.17	0.03
Horse	96966	3.374	1.322	0.22	0.04
Venus	100000	3.445	1.402	0.25	0.04

Table 3: Rendering times in seconds.

Note that before rendering a model, we need first to load it into memory.

Loading an AIF model from a text file includes two steps:

- Loading a mesh from disk (Table 3 column "**Loading**");
- Inducing an orientation on the mesh using the orientation mechanism (Table 3 column "**Orientation**").

Rendering an AIF model can be done by:

- Creating a GL primitive and its display (Table 3 column "**GL_P**"), or, alternatively
- Creating a GL List with the primitive and its display (Table 3 column "**GL_L**").

Table 3 shows that rendering large manifold meshes (e.g. the venus, the horse, and the bunny depicted in Figure 2), as well as non-triangular and non-manifold meshes (e.g. the CessnaNM pictured in Figure 7) is fast, even when we do not use GL lists (column "**GL_P**" in Table 3). GL lists substantially reduce the rendering time (column "GL_L" in Table 3), but with an additional time cost in creating GL list.

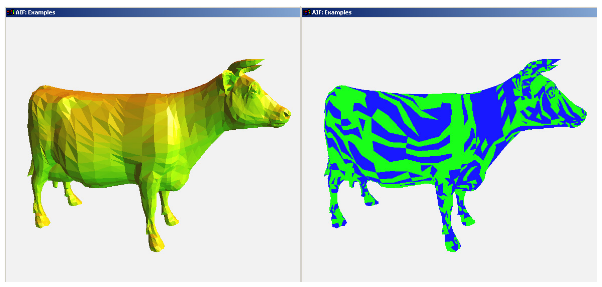

Figure 6: Cow mesh using GL_TRIANGLES and GL_TRIANGLES_STRIP.

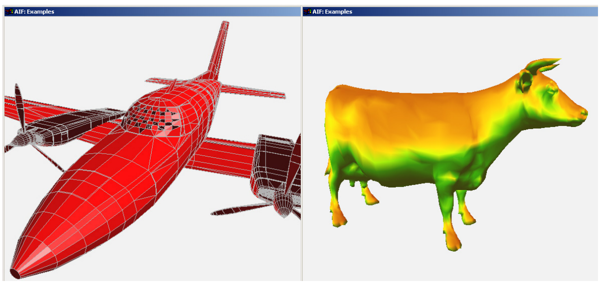

Figure 7: Cessna mesh with GL_POLYGON and Cow mesh using GL_TRIANGLE_FAN.

Besides, AIF easily supports different graphical primitives for rendering: triangular primitive and stripe primitive (Figure 6), polygonal primitive and triangle fan primitive (Figure 7). The cow mesh depicted in Figure 7 looks smooth because we have used vertex normal, instead of face normal as in Figure 6.

10. CONCLUSIONS AND FUTURE WORK

This paper has described a new data structure, called AIF, together with a query operator in order to satisfy three major requirements, namely:

- *Responsiveness.* AIF was designed to guarantee fast queries by using a single indexed mask operator. It belongs to class C_4^9, what means that a minimal number of direct and indirect accesses are required to retrieve adjacency and incidence information from it.
- *Conciseness.* Unlike other b-rep data structures, AIF is an orientable – but not oriented – data structure for 2-dimensional meshes, i.e. it does not contain oriented cells. As a consequence, it is more concise than other b-reps.
- *Generality.* AIF supports polygonal meshes. Besides, it may store manifold and non-manifold polygonal meshes, even in higher dimensions.

In the near future, we hope that AIF data structure can address important application requirements in multiresolution and animation such as, for example, interactive editing of multiresolution meshes.

ACKNOWLEDGEMENTS

The models are courtesy of: Cyberware, Stanford University, Viewpoint Digital Inc., Avalon archive, and 3D Cafe.

REFERENCES

[Baumgart72] B. G. Baumgart. Winged-edge polyhedron representation. Technical report, STAN-CS-320, Stanford University, 1972.

[Brisson93] E. Brisson. Representing geometric structures in *d* dimension: topology and order. *Descrete & Computational Geometry*, Vol.9, no.4, pp.387-426, 1993.

[Campagna98] S. Campagna, L. Kobblet, and Hans-Peter Seidel. Directed Edges - A Scalable Representation for Triangular Meshes. *Journal of Graphics Tools: JGT*, Vol. 3. no.4, pp.1-12, 1998.

[Floriani02] L. Floriani, P. Magillo, E. Puppo, and D. Sobrero. A multi-resolution topological representation for non-manifold meshes, Proceedings of the seventh ACM symposium on Solid modeling and applications, pp.159-170. 2002.

[Hoppe98] H. Hoppe. View-Dependent Refinement of Progressive Meshes. *Technical Report MSR-TR-98-02*, Microsoft Research, 1998.

[Kallmann01] M. Kallmann and D. Thalmann. Star-vertices: A Compact Representation for Planar Meshes with Adjacency Information. *JGT*, Vol.6, no.1, 2001.

[Lee01] S. H. Lee and K. Lee. Partial Entity Structure: A Fast and Compact Non-Manifold Boundary Representation Based on Partial Topological Entities. *In Proceedings of the Sixth ACM Symposium on Solid Modeling and Applications*, pp.159-170, 2001.

[Loop00] C. Loop. Managing Adjacency in Triangular Meshes. *Tech. Report No. MSR-TR-2000-24*, Microsoft Research, January 2000.

[Mantyla88] M. Mäntylä. *An Introduction to Solid Modeling*, Computer Science Press, 1988.

[Ni94] X. Ni, and M. Susan Bloor. Performance Evaluation of Boundary Data Structures. *IEEE Computer Graphics and Applications,* Vol.14, no.6, pp.66-77, 1994.

[Pajarola01] R. Pajarola. FastMesh: Efficient View-dependent Meshing. *In Proceedings of the Pacific Graphics 2001*, pp.22-30, 2001.

[Popovic97] J. Popovic, H. Hoppe. Progressive Simplicial Complexes. *Computer Graphics*, Vol.31, pp.217-224, 1997.

[Weiler88] K. Weiler. The radial edge structure: a topological representation for non-manifold geometric boundary modeling. *In Geometric modelling for CAD applications.* M. Wozny, H. McLaughlin, and J. Encarnacao, (eds.), North-Holland, pp. 3-36, 1988.