AN ALGORITHM ORIENTED MESH DATABASE (AOMD) APPLICATION: DECIMATION

B. Kaan Karamete

Coventor Inc., MA., U.S.A. kaan@coventor.com

ABSTRACT

This paper discusses the efficiency and ease of implementation of a mesh coarsening or decimation (simplification) algorithm by using Algorithm Oriented Mesh Database (AOMD). AOMD is first introduced by the author and his co-workers in recent study [1]. The manuscript is aimed to give the reader the novel idea of coupling algorithm and database (or data structure) design such that the database is evolved by the demands of the algorithm(s). AOMD design principles will be explained and the key factors that affect the performance of the decimation algorithm will be explored as a case study. The issues related to mesh inter-entity adjacencies, mesh-model associations, i.e., classifications and AOMD data structures will be discussed. The selected decimation algorithm coarsens the mesh (2D or 3D) to the desired level by maintaining the topological and geometrical integrity of the input mesh by means of successive edge collapses by keeping the mesh quality as good as possible. The details of the decimation algorithm with emphasize on how the algorithm drives AOMD will be studied explicitly within the context.

Keywords: mesh generation, mesh-model database, computational geometry, decimation, AOMD

1. INTRODUCTION

There has always been a difficulty of adapting a mesh-model data structure to a new algorithm whose best implementation requires changes in the pre-designed mesh database. This is exactly the aim of AOMD, i.e., the ability to provide means for the re-design of the data structures without really creating a new database. The idea of using flexible adjacency relations between mesh entities is the main design concept of AOMD. It also inherits the valuable concept of mesh-model associations, i.e., classifications to satisfy the mesh–geometry integrity from the works of Beall and Shephard [2]. This paper can be thought to be an extension to the earlier introductory study where we have introduced the main design concepts of AOMD [1]. The reader will find more “ready to grasp” ideas and a case algorithm implementation by using AOMD in this paper. In the first section, basic building blocks of AOMD will be studied by giving practical examples. To keep the pace of thinking process with the reader, the text of Section 2 will pose questions by reiterating the possible alternative ways in AOMD design. Section 2 is dedicated to the implementation of a decimation algorithm with the use of AOMD to show how the data structures can be changed by the desires of the algorithms. The secondary intent of the paper and Section 2 is to provide the algorithmic details of a decimation (mesh coarsening) process. In computer graphics and numerical simulation engineering, large meshes are prohibitive and should be avoided where possible. A decimation algorithm is needed to simplify large meshes that might have been generated from a range scanning device or pre-meshed by a less capable mesh generator. The aim is to reduce the number of mesh faces while maintaining the geometrical integrity and shape of the true geometry with quality meshes. A good survey of mesh simplification algorithms is done by Paul Heckbert and Michael Garland [3]. A number of mesh simplification or decimation algorithms are depicted in that work. I will specifically note the work of Hoppe, Turk, Garland, Schroeder and Frey [4-8] due to the fact that they all have utilized edge collapses, mesh optimizations after edge collapses and mentioned about geometrical validity of the operation(s). Hoppe tried to formulate the collapses by optimizing energy functional, which is a measure of initial vertex clustering and curvature. The order of vertex collapses is discussed by Turk. The deviation from the initial mesh and the quality of decimation is found by means of a quadratic error minimization constructed from the distances between the vertices and their adjacent average planes by Heckbert and Garland as well as by Scroeder. Frey has controlled the deviation of the simplified mesh by evaluating the distances between the vertices and their orthogonal projection to the ball of the vertex followed by mesh optimization by edge swaps and vertex smoothings. Some of these algorithms are quite complex and computational efficiency could be problematic. Although the algorithm in this paper discovers nothing new compared to the papers aforementioned above, it will generalize the rules of mesh simplification in a very simple conjecture and in an efficient manner. Geometrical integrity will be preserved as the direct result of maintaining mesh-model associations; classifications and by checking the angle differences between the mesh face normals of the initial and decimated configurations. Like Frey [8], the mesh will be optimized after the possible set of vertex collapses by means of a successive edge swapping procedure. The different levels of decimation are generated by using local edge length metrics that can be altered by any solution adaptation or a third party procedure. The details of the algorithm and how it is coupled with AOMD will be explained in Section 3. This algorithm can decimate both volume and surface meshes.

2. BASICS of AOMD

Mesh-Model entities of AOMD, namely vertex, edge, face and region has levels of hierarchy; vertex being the lowest(0) and region being the highest(3). This is because lower order mesh entities can be expressed in the closure of higher order entities, as is the case in graph theory. Each entity has lower and higher order entities with respect to its own level as depicted in Figure 1.

	Vertex(0)	Edge(1)	Face(2)	Region(3)
Vertex	NA	Upward(U) Higher(H)	U/H	U/H
Edge	Downw(D) Lower(L)	NA	U/H	U/H
Face	D/L	D/L	NA	U/H
Region	D/L	D/L	D/L	NA

Figure 1. Downward(D) or Lower(L) and Upward(U) or Higher(H) Order entities for each level (row to column); e.g., for edge, vertex is its lower order entity and face and region are higher to it.

	Vertex	Edge	Face	Region
Vertex	0	1	0	0
Edge	1	0	1	0
Face	1	0	0	1
Region	1	0	1	0

Figure 2. Adjacency Status Tensor T(I,J). If adjacency exists from (ith row) entity to (jth column) entity then its cell value is set to 1, otherwise 0. All edges around each vertex will be generated and kept since T(0,1) = 1.

The adjacency relations are stored in a 4x4 tensor. We will call this tensor as Adjacency Status Tensor and denote it by T(I,J). User will set certain adjacency relations by setting the cells of this tensor. In other words, by setting the cells of this tensor AOMD is told to create the adjacency links implicitly until a different user request is made. At any levels of execution of an algorithm, user may change the values of T. For example, at the beginning of a vertex smoothing algorithm, T(0,2) can be set to request from AOMD to create vertex to face adjacencies, i.e., the faces surrounding each vertex. This request is currently global to an entity level, i.e., all the vertices will have a list of faces around them. The good point is that user does not have to create this link explicitly. As a second example, if the user creates faces from vertices by a constructor call as below,

```
pFace_ face = F_create_(pMesh_ mesh,
                        pVertex_ v0, v1,v2,
                        int classification ,
                        int  tag);                    (1)
```

and have set T(1,2) a-priori, the edges will be created implicitly and all the edges will have a list of adjacent faces so that a question as below could be asked easily without any additional programming:

```
pEdge_ edge = E_exists_(v0,v1);
pList_ faces   = E_faces_(edge); // or              (2)
pFace_ face  = E_face_(edge,0); // if faces>0.
```

2.1 Classification

Classification is a way to associate mesh and model entities.. Each mesh entity has a classification field prescribing its underlying model entity. For instance, mesh vertices can be classified on model vertices, edges, faces or regions. In general, as a rule, a mesh entity can only be classified on model entities of equal or higher order than that of itself. We can formulate this relation as M^I [G^J iff J≥I, where G^J is the J^{th} order model entity and M^I as the I^{th} order mesh entity. Having classification information helps us achieve many operations possible more efficiently and more correctly such as mesh refinement, coarsening, optimization and e.g. boundary condition assignment to a cluster of mesh faces and etc. In mesh refinement, the new vertices can be snapped to the true geometry if the classification of the split entity is known. In mesh coarsening, the geometric integrity can be preserved by not allowing the edge collapse through model faces if the edge classification is known. Otherwise we would have to evaluate the validity of the collapse operation by computing the angles or distance differences between the new and the initial configurations usually by means of a threshold value which can never be as reliable as the simple classification check. It should be noted here that geometrical validity of the edge collapse operation is still required. This will be discussed in Section 3 in detail. Generally, almost all mesh generation techniques require some sort of classification information and the best practice is to keep this information even after meshing to provide the possibility of further mesh modifications or post-processing which can be a computational field simulation algorithm or selective visualization, etc.

However, there is an important problem of how to keep the classifications correctly while providing one of the most desirable features of creating mesh adjacencies implicitly by AOMD(*). For instance, user may want to request from AOMD to create edge to face adjacencies (T(1,2)) while constructing faces from vertices as shown in (1). In (1), the face is created from three vertices **v0,v1,v2** on the geometrical model entity of level **classification** and model entity id of **tag**. Mesh edges are then either created from pairwise vertex permutations (v0-v1,v1-v2,v2-v0) in the given order or could already be existing. An edge can be used by the face opposite to its vertex orders if the edge is existing a-priori. This usually happens between two mesh faces. For instance, edge M^1_5 is used by the face M^2_1 in reverse fashion since the vertex order (orientation) of M^2_1 is opposite to the edge's vertex direction as depicted in Figure 3.

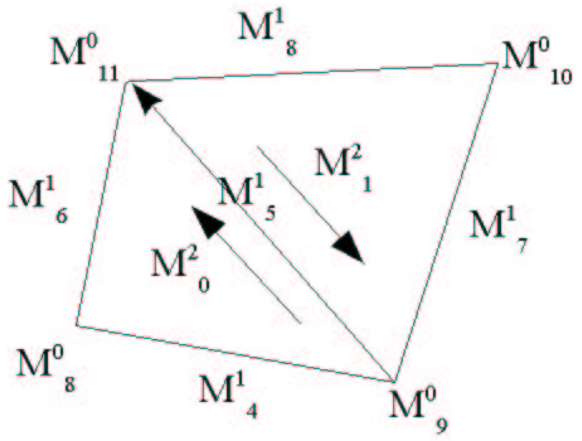

Figure 3. Face edge uses; the edge is defined from v0 to v1. f0 is using it in positive (dir=1) sense and f1 is using it in opposite (dir=0) sense.

2.1.1 Internal data manipulations

Face constructor stated in (1) not only creates edges or decides how to use existing ones, it also adds itself (face pointer) to the upward face adjacency list of all of its edges since T(1,2) is requested. The question of whether an edge exists as stated in (2) can be found in two different ways. If the vertex to edge adjacency (T(0,1)) exists then we can try to find the common edge by checking the edge lists of these two vertices. However, if T(0,1) is not set then AOMD creates an edge hash list by chaining. Basically, edges are stored in a list such that they can be searched and uniquely found from its vertices. This search operation can be achieved by a good key selection, which will result in least number of collisions i.e., the edges with the same key value. It is devised that a good key selection could be the sum of edge's vertex ids [1]. All mesh entity ids are unique since when they are stored in a mesh, an id generator keeps track of available entity ids and assigns them to the entity. When an entity is deleted, its id is stored on a stack. The id generator pops up an id from the head of the stack when a new entity is created or if there exists no available ids present then it increments the maximum id assigned so far.

To eliminate large collisions, i.e., the edges having the same vertex id sums, a trick of calculating the sum of random numbers seeded by the vertex ids is applied. The sum is then mod to the size of the hash list to find the location of the edge in the hash list. The edge is appended to an expanding-shrinking array list at that bucket location of the hash list. Standard Template Library (STL) equivalent of this data container is a hash-set with the same less than key being the randomized vertex id sums. As a general rule, the search to find a higher order entity from a list of lower order entities can be done by means of hash sets where the hasher function is the sum of randomized entity ids. If there exists a hash set, its worst collision number and frequency is automatically monitored within AOMD. if the rate of 10 or more collisions occurs more than 10% then the set is rehashed by an order of magnitude till a maximum hash limit is reached. Therefore, AOMD internally keeps the possibility of creating, modifying and deleting hash sets (chains) for edges, faces and/or regions since all can be determined from their lower order entities. This enables to answer the questions like F_exists_ and R_exists_ for faces and regions given lower order entity lists usually vertices within allowable CPU [1].

In entity deletions, the adjacency status tensor is checked implicitly to determine to delete the entity from its downward entities. For instance, if a face is deleted from the mesh by a simple F_delete_(mesh,face) call then its existence from the upward face lists of its downward entities is checked and deleted. In addition, if the face is hashed then its signature from the faces' hash list is deleted as well to prevent memory leaks.

Let us reiterate the question posed at (*) in Section 2.1, that the edge classifications may not be assigned correctly if the face is created from vertices. This is a fact that lower order entity classifications can not be deduced either from higher or lower entity classifications unless the same-level classification of all entities are known, i.e., vertices on model vertices, edges on model edges, faces on model faces. In Figure 4, the edge classification between vertex 9 (M^0_9) and vertex 4 can be deduced neither from face 10's classification (on 0th model face G^2_0) nor from vertex classifications. However, if all the mesh edges classified on the model edges would have been known then it would be straightforward to define the classification of the edge between vertex 4 and 9.

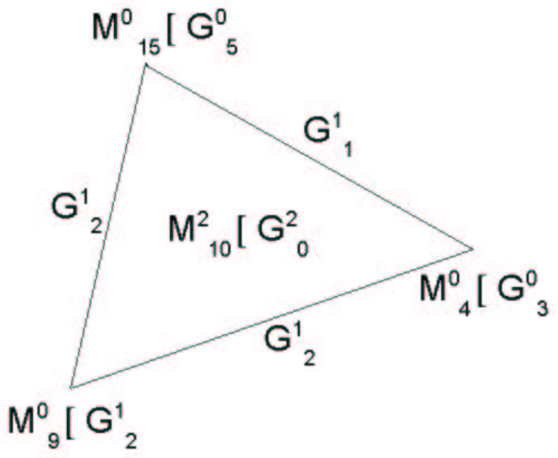

Figure 4. Edge classification between vertex 9 and 4 could only be decided if all the mesh edges on model edges are known.

Another approach could have been to device a walk strategy to reassign all the classifications by computing angle differences and adjacency relations. This way should only be practiced if the mesh classification information is not readily available or provided. Therefore, users should be extra cautious to request the adjacency links and use of constructs to eliminate classification problems. This is still the subject of further AOMD research.

2.2 Templates and Iterators

The mesh regions can be of type tetrahedron, hexahedron, pyramid, prism, etc. There is a template for each of these region types in AOMD to define region-face, face-vertex connectivities [1]. It is required for AOMD to figure out how to form the region if it is created from vertices or edges. These template functions can be overridden if additional region types would be added to AOMD. As an example, a region can be created from vertices as below:

```
pRegion_ region = R_create_(pMesh mesh,
                            pVertex_ v0,v1,v2,v3,
                            constant RTip_ TETRAHEDRON,
                            int classification, int tag);          (3)
```

If T(2,3) is requested then faces are created from the TETRAHEDRON template and region-face use directions are found. A region may use an existing face in opposite direction to its natural vertex order as stated in Section 2.1 for face-edge uses. The region itself is added to the upward region lists of all of its faces. If T(1,3) is asked then the region would be added to the region lists of all of its 6 edges that can be derived from its template structure. After the above constructor call, the following function calls could be done without any additional programming:

```
pFace_ face = R_face_(region,0);
pList_ regions = E_regions_(edge);
int usedir = R_dirFace_(region,face);                    (4)
```

The geometrical model is implemented on top of the solid modeling kernels as a wrapper. It consists of a set of generic function names which is linked with the chosen solid modeler at the compile time. This enables AOMD to operate on different solid modelers without the necessity of changing its model related function calls. For snapping a vertex on a true geometry, e.g., the following function call can be used irrespective of the solid modeler.

```
G_snapVertex_(pModel model, pVertex vertex);  (5)
```

The above function uses the linked solid modeler's specific functions to be able to snap the vertex to the closest point on the model. Mesh entities are stored on their classified model entities in separate lists. If the mesh faces classified on a particular model face are required then AOMD creates an iterator of the entity kind for the user to iterate through the classified mesh entities. The following iteration loop can be used for this purpose:

```
pGFace_ model_face = F_modeling_(face_ face);
iterator it;                                             (6)
while(face face=G_nextFace_(model_face,&it)){ …};
```

In the next section, the features of AOMD will be used by the decimation algorithm as a case study.

3. DECIMATION ALGORITHM

The input to the decimation algorithm could be either a surface mesh or a volume mesh. Let us assume that the input is read from a finite element connectivity data and coordinates; with or without classification information. The decimation algorithm checks have two layers; first one is to always account for the classification information (topological checks); second one is to account for geometrical validity (geometrical checks). The main goal is to coarsen the mesh without sacrificing the main geometrical features and the mesh quality. Note that if the solid model is available then geometrical features are exactly known at every location. Otherwise, mesh implicitly defines the model features since mesh is an artifact or the discrete form of a solid model. The difficulty is to coarsen the mesh by respecting implicitly defined model features with quality meshes. The decimation algorithm uses the edge collapse operation to fulfill this task. The key issue in the successful implementation of the edge collapse algorithm is to perform geometrical and topological checks properly and reliably such that edge collapse always results in valid configurations. For efficiency reasons, topological (classification) checks should come first since they do not require any computation. In an edge collapse operation, the faces around the to-be collapsed edge and the edges around one of the end vertices of the edge are deleted. This vertex will be deleted and the edge will be collapsed onto the other end (retained) vertex of the edge. The faces around the to-be deleted vertex are stretched to the retained vertex of the edge by forming new faces. If the vertex to be deleted is classified on a model vertex, edge collapse should be avoided. If the to-be deleted vertex classification and tag is not equal to the edge classification, the operation should be avoided as well. For instance, if the vertex is classified on a model edge (M^0_1 in Figure 5a) and the edge to be collapsed is classified on another model edge or on a model face or region, the collapse would have changed the topology of the geometric model as depicted in Figure 5a. It should be noted here that if there is a face other than the faces of the edge whose area becomes flat, then the operation should be avoided so as not to create invalid faces as shown in Figure 5a for the mesh face M^2_3 for the collapse of M^0_2. The only valid collapse moves as depicted by I and II in Figure 5a, results in the configurations as shown in Figure 5b. In case II, the edge classified on the model edge G^1_3 is collapsed along the model edge. Note that the deleted vertex M^0_1 is classified on the same model edge. Geometrical validity check of the edge collapse operation is illustrated in Figure 6. The new faces after the collapse operation should have all positive and non-zero face areas with respect to the orientations. On surface meshes, this condition can be loosely stated by satisfying an angle threshold (usually a low value ten degrees) between the face normals of the initial and after collapse configurations. In fact, this check is sufficient for planar meshes since the case of inside out face creation (reverse orientation in face's vertex orders) results in an angle difference of 180 degrees. If the angle differences between N1's or N2's are greater than the angle threshold then the collapse is avoided as shown in Figure 6. Intersection checks are advised after the completion of collapse algorithm to account for the fact that angle threshold check may not be sufficient for detecting self-intersections especially for rough surfaces. Another possibility is to lower the angle threshold which might result in less decimation.

At the beginning of the decimation algorithm, a spacing value for each mesh vertex is assigned by computing the average edge lengths of the surrounding (adjacent) edges. Therefore, the input mesh (assumed to be the set of faces composed of integer vertex ids whose coordinates are also given) is read such that mesh edges are created and vertex to edge adjacencies are also made known. This is the required data structure to be able to assign average edge lengths to the vertices. AOMD satisfies T(0,1) implicitly in face creations from vertices while reading the input data.

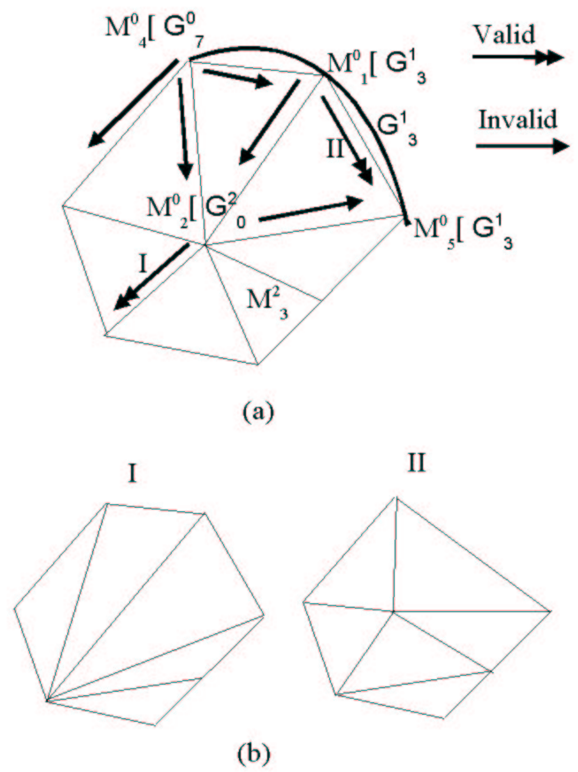

Figure 5. Definition and validity of edge collapses; (a) double arrows show valid moves and single arrows show invalid collapses. (b) After collapse configurations for cases I and II.

If there is no classification information available the classification field of edges may not be accurate as explained in the preceding Section 2.1. This may be solved by walking the faces through edges. Edges along model edges (implicitly defined by the mesh) may be assigned if the angle difference between the adjacent mesh faces of the edge exceeds an angle threshold (e.g., 20 degrees). However, this step is not absolutely necessary since our edge collapse mechanism is backed up by the geometrical validity checks which avoid collapses if the angle difference between the initial and after collapse configurations is greater than an angle threshold as well. Aside from the advantages of creating classification information explained in Section 2.1, having classifications may also be favorable in computational perspective since the number of geometrical checks may be less in checking collapses locally for every candidate edge.

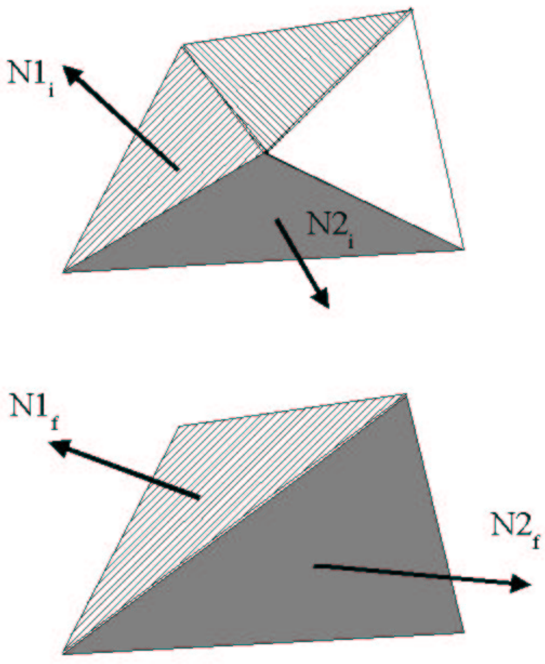

Figure 6. Geometrical check of edge collapse: collapse is avoided if the angle difference between N1s or N2s is greater than an angle threshold.

The level of decimation is defined as a factor that lowers the spacing field of the vertices. Therefore, vertex spacings (average edge lengths) are updated by dividing the values by a user prescribed factor (2,3,4, etc..).

For the successful implementation of the decimation algorithm, faces around vertices and faces adjacent to edges are needed. Therefore, T(0,2) and T(1,2) are requested from AOMD. In practice, faces can be created either from vertices or from edges. However, if the faces are created from edges, T(0,1) is not required although AOMD can still find if an edge exists from the two vertices by use of hash lists as explained in Section 2.1.1. It is recommended to use T(2,1) to overcome the additional data handling in manipulating hash-lists or additional memory requirement for T(0,1).

The rest of the decimation algorithm is relatively simple since we have already set the algorithmic design details by choosing the best data structure for its successful implementation. The edge collapse operation is tried for all the mesh edges successively. There might be situations for which the collapse operation could not be performed due to geometrical reasons (large angle differences, invalid face creations) at the end of one edge collapse cycle for all the mesh edges. Edge collapses degrade mesh quality and may create configurations for further collapses to become impossible. The overall quality of the mesh needs to be improved to perform more collapses and better triangulation. One possible way of improving mesh quality globally is to apply edge swaps to maximize minimum face angles (the converse is also true) to the edges around each vertex locally. The details of this algorithm can be found in [9] and [10]. The difference in face normals between initial and after swap configuration is checked for the geometrical validity of the edge swap operation similar to the check done for the edge collapse case. The vertex locations are smoothed such that the location of surrounding edges will be averaged to find the new vertex locations. The new location should not affect the difference between the face normals more than the angle threshold so that the initial geometrical features are preserved. Having increased the quality of the triangulation, the edge collapses are tried once again. This cycle is repeated until no more edge collapses are possible. In practice, for efficiency purposes, the mesh quality improvement loop is visited if the number of edge collapse sweeps is less than a couple of iterations. The C/C++ program fragment for the decimation algorithm is depicted below for reader's convenience. It iterates through all the mesh edges and tries to collapse each edge if the vertex spacing of any one of the edge vertices (*v[0]* or *v[1]*) is greater than the edge length *elength*. After all the edge collapses are tried, the mesh quality is improved by swapping the edges around each vertex where we need to request T(0,1) from AOMD. The examples of the decimation algorithm are given in Section 4.

```
do{
  done=0;tmp=0;
   while(edge = M_nextEdge_(mesh,&tmp)){
      v[0]=E_vertex_(edge,0);v[1]=E_vertex_(edge,1);
      elength =E_length_(edge);
      s[0]=L_dataD_(v[0]->list,"spacing");
      s[1]=L_dataD_(v[1]->list,"spacing");
      for(i=0;i<2;++i)
          if(s[i]>elength)
            if(E_2Dcolapse_(mesh,edge,v[i])){
                  done = 1;
                   break;
             }
   }
   tmp=0;
    while(vertex=M_nextVertex_(mesh,&tmp)){
       V_swapOpEdges_(mesh,vertex);
       V_smooth_(mesh,vertex);
     }
}while(done);
```

Figure 7. Decimation Algorithm.

4. RESULTS AND DISCUSSION

The results of the above decimation algorithm can be seen on the example cases in Figure 8 through Figure 16. The surface mesh of the Stanford bunny model, which is originally created by a range-scanning device, is shown in Figure 8. The bunny is decimated by lowering the edge length scale 4 and 8 times in Figure 9 and 10 respectively. Figure 11 represents the coarsest possible configuration with the angle threshold of 10 degrees. It should be noted here that the decimated meshes of Figure 9, 10 and 11 preserve the main features of the bunny with quality meshes by reducing the number of mesh entities considerably. The effect of angle threshold on the decimation algorithm can be seen in Figure 12. The angle threshold value for edge collapses is taken as 60 degrees in Figure 12. It may be difficult to identify the mesh of Figure 12 as the bunny of the initial original mesh of Figure 8. The geometrical features of the bunny are deteriorating at this angle value and if the value is further reduced, we may not be able to identify the bunny at all. The same set of comparisons is done on the dragon model as depicted in Figures 13-17. The amount of decimation for bunny and dragon models corresponding to the Figures 8-11 and Figures 13-16 are given in Table 1 and 2, respectively. Maximum compression ratio of the models is between 1/10 and 1/20 in terms of the number of mesh faces as depicted in Table 1 and Table 2.

Table 1. The amount of decimation for the bunny.

Figure No	Vertices	Edges	Faces
8	35947	0	69451
9	4567	10294	6837
10	2961	5496	3645
11	2667	4626	3069

Table 2. The amount of decimation for the dragon.

Figure No	Vertices	Edges	Faces
13	100250	0	202520
14	22166	66555	44464
15	11364	34099	22811
16	9115	27345	18306

It is worthwhile to discuss the differences of adjacency requests between consecutive algorithms. In fact, as usual, we have to make a decision between memory and CPU time since we can delete the adjacencies, namely T(0,2) for the completed edge collapse loop and create T(0,1) for vertex smoothing and edge swap loop. In practice, additional memory requirement for surface meshes even for large (~100K mesh faces) cases is not drastic (<16Mbyte), and sacrificed in favor of the gain in the computational time. The decimation algorithm runs within a couple of seconds for the bunny and less than 10 seconds for the dragon model on a 700MHz. PC.

In general, AOMD is designed to provide a flexible basis for mesh-model inter-entity adjacencies. All the adjacency creations and deletions are buried inside AOMD by keeping the usage as simple as possible while performing the most complicated adjacency relations correctly and efficiently. The power of an efficient database can be measured by the ease of use and the amount of work that can be achieved internally. We believe that AOMD is the right database design for the success of CAD and simulation engineering algorithms simply because it lets the algorithms to customize itself. It is hoped that AOMD design concepts will have positive impact on meshing and related communities.

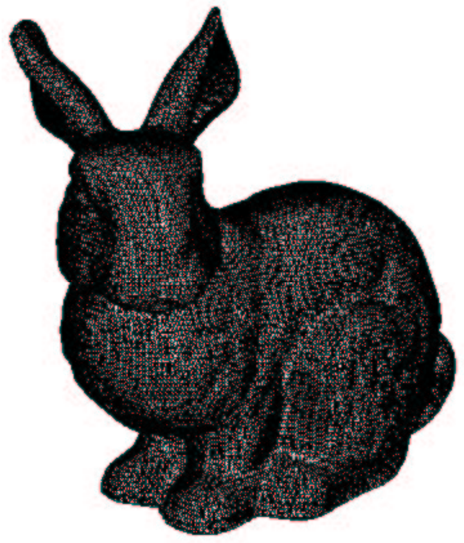

Figure 8. Original surface mesh of the Stanford bunny. (http:\\www.stanford.edu\data\3Dscanrep\bunny.tar.gz). The mesh consists of ~36K mesh vertices and ~70K mesh faces.

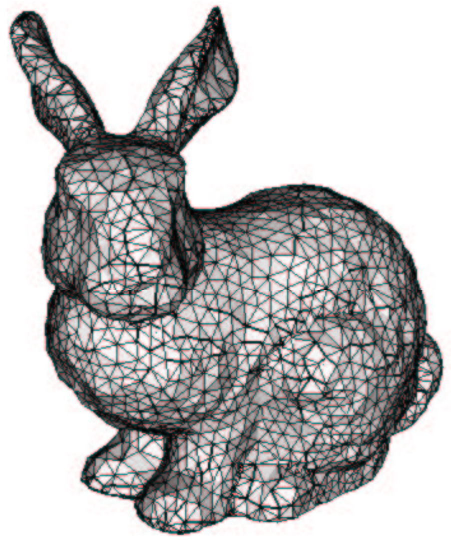

Figure 9. Stanford bunny is decimated such that its original edge length scale is reduced by 4 times. The mesh consists of ~4.5K vertices, 10K edges and ~6.8K faces.

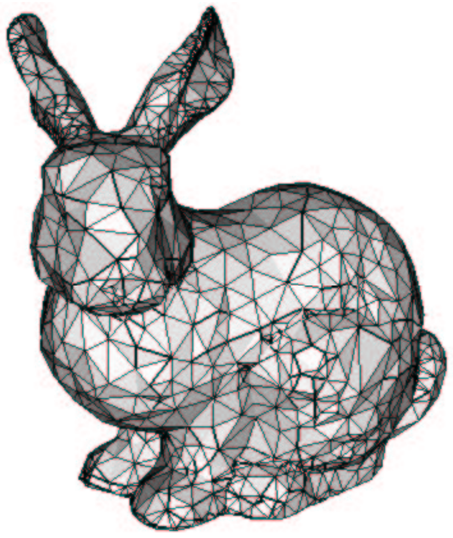

Figure 10. Stanford bunny is decimated such that its original edge length scale is reduced by 8 times. The mesh consists of ~3K vertices, 5.4K edges and ~3.6K faces.

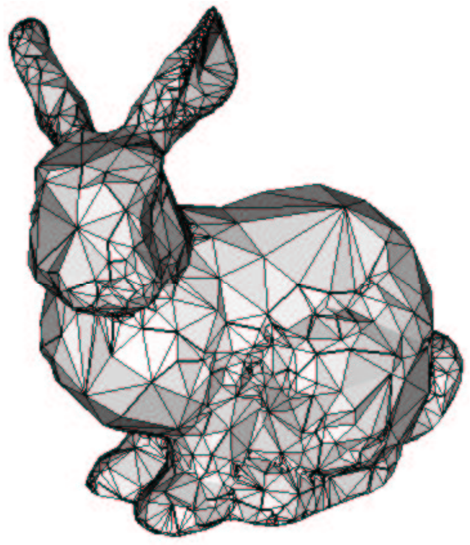

Figure 11. Stanford bunny is decimated as coarse as possible with an angle threshold value of 10 degrees for edge collapses. The mesh consists of ~2.6K vertices, ~4.6K edges and ~3K faces.

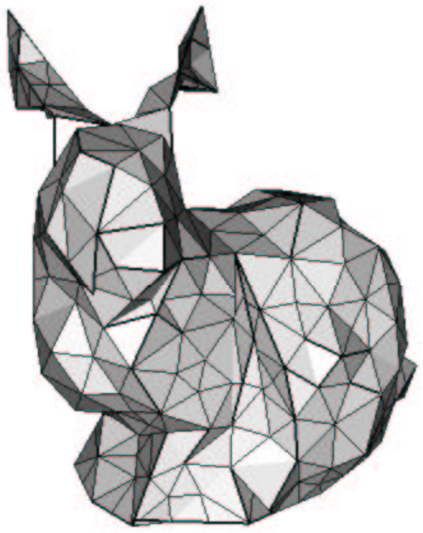

Figure 12. Stanford bunny is decimated as coarse as possible with an angle threshold value of 60 degrees for edge collapses. The bunny's features are hardly identifiable.

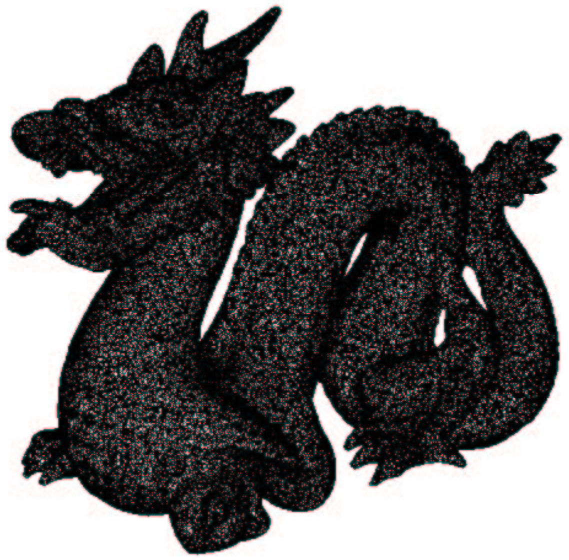

Figure 13. Original surface mesh of the dragon model.(http:\\www.stanford.edu\data\3Dscanrep\dragon). Mesh consists of ~100K mesh vertices and ~200K mesh faces.

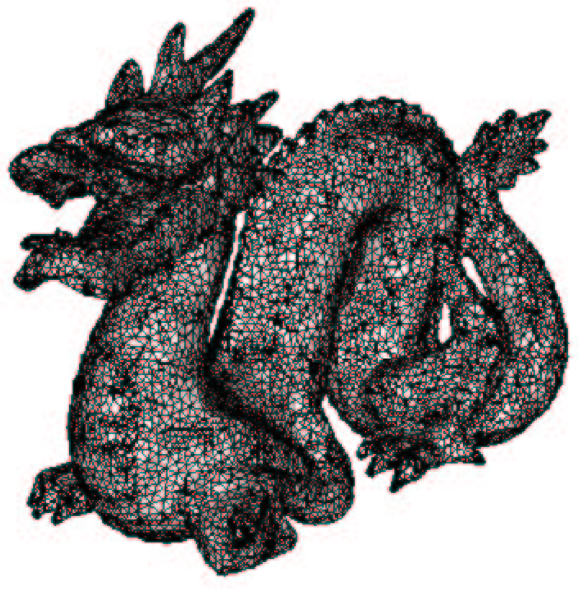

Figure 14. Dragon is decimated such that its original edge length scale is reduced by 4 times. The mesh consists of ~22K vertices, 66K edges and ~44K faces.

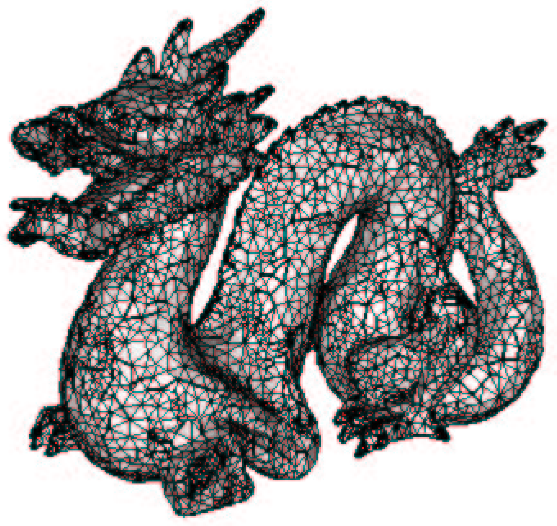

Figure 15. Dragon is decimated such that its original edge length scale is reduced by 8 times. The mesh consists of ~11K vertices, 34K edges and ~22K faces.

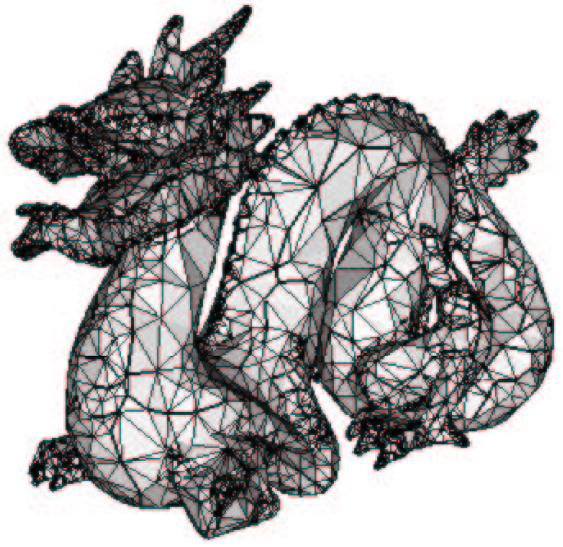

Figure 16. Dragon is decimated as coarse as possible with an angle threshold value of 10 degrees for edge collapses. The mesh consists of ~9.1K vertices, ~27K edges and ~18K faces.

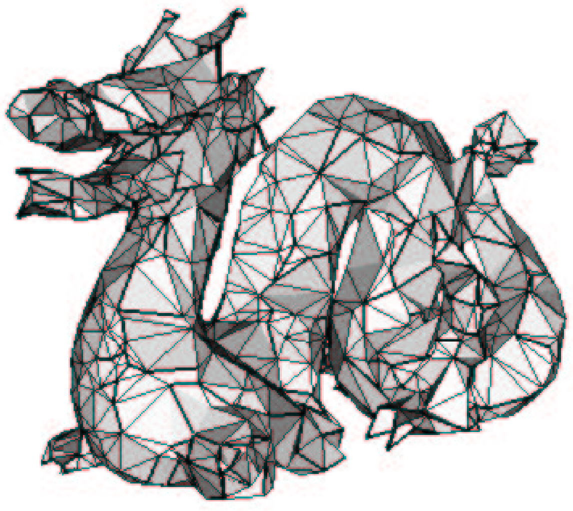

Figure 17. Dragon is decimated as coarse as possible with an angle threshold value of 60 degrees for edge collapses. (~2.6K vertices, 4.3K edges, 2.9K faces) Dragon's features degrade.

REFERENCES

[1] Jean Francois Remacle, B. Kaan Karamete and MS Shephard, “Algorithm Oriented Mesh Database (AOMD)”, *9th International Meshing Roundtable, Sandia National Laboratories*, pp 349-359 October 2-5 2000, New Orleans, Louisiana USA.

[2] MW Beall and MS Shephard, “A general topology-based mesh data structure”, *Int. J. Num. Meth. Eng.*, Vol 40 pp.727-758 (1997).

[3] Paul S. Heckbert and Michael Garland, “Survey of Polygonal Surface Simplification Algorithms”, *Carnegie Mellon University Tech. Report, and Multiresolution Surface Modeling Course SIGGRAPH’97*. http://www.cs.cmy.edu/~ph/mcourse97.html.

[4] Hugues Hoppe, Tony DeRose, Tom Duchamp, John McDonald and Werner Stuetzle. “Mesh Optimization”, *SIGGRAPH’ 93 Proc*. pp. 19-26, Aug. 1993 http:// www.research.microsoft.com/research/graphics/hoppe

[5] William Schroeder, Jonathan A. Zarge and William E. Lorensen. “Decimation of triangle meshes”, *Computer Graphics, (SIGGRAPH ‘ 92 Proc.)*, Vol: 26(2) pp. 65-70, July 1992.

[6] Michael Garland and Paul Heckbert, “Surface Simplification Using Quadric Error Metrics”, *Proceedings SIGGRAPH 97*. http://sulfuric.graphics.cs.cmu.edu/~garland/quadrics.ps.gz

[7] G. Turk, “Re-tiling polygonal surface”, *Computer Graphics*, Vol: 26(2) pp. 55-64 (1992).

[8] Pascal Frey, "About Surface Remeshing", *Proceedings, 9th International Meshing Roundtable*, Sandia National Laboratories, pp.123-136, October 2000

[9] B. Kaan Karamete, Rao V. Garimella, Mark S. Shephard, “Recovery of an Arbitrary Edge on an Existing Surface Mesh Using Local Mesh Modifications”, *Int. J. Num Meth. Eng.*, Vol 50-6 pp. 1389-1409 (2001).

[10] B. Kaan Karamete, Mark W. Beall and Mark S. Shephard, “Triangulation of Arbitrary Polyhedra”, *Int. J. Num. Eng.*, Vol 49-2, pp. 167-191 (2000).