Multimedia Systems (2006) 11(5): 480–494
DOI 10.1007/s00530-006-0017-1

REGULAR PAPER

Hung-Kuang Chen · Chin-Shyurng Fahn · Jeffrey J. P. Tsai · Rong-Ming Chen · Ming-Bo Lin

Generating high-quality discrete LOD meshes for 3D computer games in linear time

Published online: 24 March 2006

Abstract The real-time interactive 3D multimedia applications such as 3D computer games and virtual reality (VR) have become prominent multimedia applications in recent years. In these applications, both visual fidelity and degree of interactivity are usually crucial to the success or failure of employment. Although the visual fidelity can be increased using more polygons for representing an object, it takes a higher rendering cost and adversely affects the rendering efficiency. To balance between the visual quality and the rendering efficiency, a set of level-of-detail (LOD) meshes has to be generated in advance. In this paper, we propose a highly efficient polygonal mesh simplification algorithm that is capable of generating a set of high-quality discrete LOD meshes in linear run time. The new algorithm adopts memoryless vertex quadric computation, and suggests the use of constant size replacement selection min-heap, pipelined simplification, two-stage optimization, and a new hole-filling scheme, which enable it to generate very high-quality LOD meshes using relatively small amount of main memory space in linear runtime.

Keywords Interactive 3D multimedia · Computer game · Virtual reality · Mesh simplification · Level-of-detail mesh · Iterative full-edge collapse

H.-K. Chen (✉) · M.-B. Lin
Department of Electronic Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, R.O.C.
E-mail: hank@asia.edu.tw, mblin@et.ntust.edu.tw

H.-K. Chen · R.-M. Chen
Department of Information and Design, Asia University, Taichung, Taiwan, R.O.C.
E-mail: rmchen@asia.edu.tw

C.-S. Fahn
Department of Computer Science and Information Engineering, National Taiwan University of Science and Technology, Taipei, Taiwan, R.O.C.

J. J. P. Tsai
Department of Computer Science, University of Illinois at Chicago, U.S.A.
E-mail: tsai@cs.uic.edu

1 Introduction

The polygonal mesh has been the major representation scheme of the 3D objects in interactive 3D multimedia applications such as 3D computer games and virtual reality (VR) that has become one of the most popular multimedia applications in recent years [1]. Essentially, both the visual fidelity and degree of interactivity play important roles in such applications. Since the objects are represented by a polygonal mesh, the increase of visual fidelity usually raises the rendering cost by using more polygons, which adversely influences the rendering efficiency. To trade off between the visual fidelity and interactivity, the level-of-detail (LOD) technique is commonly employed to adaptively adjust the resolution of the objects by selecting from a set of pre-built LOD meshes according to their importance [2].

To fill such need, the 3D objects are usually pre-converted into a set of different resolution meshes by a polygonal mesh simplification algorithm before their use.

Figure 1 shows an example of the LOD meshes in which the resolution of the Dragon meshes are decreased in proportion to the distance from the viewer. In the figure, four LOD meshes comprising 27 K, 8.8 K, 3 K, and 1 K triangles are placed from near to far to the viewer where one can hardly tell the differences between these rendered images.

Besides the contributions to interactivity, the quality of the resulting LOD meshes has great importance to the visual fidelity. Significant shape features such as horns, claws, and tails may be lost if a low-quality simplification algorithm

Fig. 1 The LOD control of the Dragon mesh according to the distance from the viewer

is employed. Therefore, it is desirable to have a polygonal mesh simplification algorithm that is capable of producing high-quality LOD meshes.

In this paper, we propose a linear time polygonal mesh simplification algorithm to generate high-quality discrete LOD meshes with very low memory and runtime costs for interactive 3D multimedia applications. The new algorithm adopts memoryless vertex quadric computation and a constant size replacement selection min-heap (RS-heap) to save the main memory cost [3, 4]. By integrating the cost computations, RS-heap operations, and vertex ring simplifications into a single pipeline, a two-stage optimization scheme to remove unnecessary optimum placement calculations, and an effective hole-filling method to fill the unwanted defective holes, the new algorithm is capable of generating very high-quality LOD meshes in linear time.

In summary, the new algorithm has at least three benefits. First, it is runtime efficient and takes only $\Theta(n)$ run time. As a result, our algorithm is considerably faster than non-linear time methods. Opposed to the other works that claim to have linear run time [5–7], our algorithm accepts an ordinary index-mesh representation without presuming any special assumptions on the input sequence. Hence, it does not need any conversion or pre-sorting stage over the entire input sequence. Second, our algorithm is highly memory efficient. To simplify the David mesh (2 mm) that has over 7 million meshes in core, it takes only 307 MB main memory space. Third, the new algorithm produces very good quality LOD meshes.

The rest of this paper is organized as follows. Prior to the discussion of our work, we will first briefly review some of related works and terms in Sects. 2 and 3. Our new simplification algorithm will be discussed in Sect. 4 followed by an analytical analysis of its running time. Section 5 presents the experimental results to show the effectiveness of the new data structures used in the simplification algorithm. Section 6 gives the conclusions of this paper.

2 Review of related works

This paper addresses the issue of generating high-quality discrete LOD meshes. Some related simplification algorithms will be reviewed first followed by the overview of LOD schemes in the second subsection. Since there are a lot of excellent works proposed in these fields, the review here is quite limited. We suggest readers refer to the book written by Lubke et al. for a more complete survey [2].

2.1 The review of the mesh simplification algorithms

The polygonal mesh simplification algorithm can be employed for at least three purposes. First, it can be used to optimize the shape of an object subject to a pre-specified constraint. Second, it can be used to create a set of approximations in a variety range of resolutions of an object. Third, it can be used to perform either lossy or lossless data compression over the input mesh, which saves storage or transmission costs.

There are at least two types of polygonal mesh simplification algorithms that are commonly used: re-sampling methods [5, 8–11] and iterative contraction methods [3, 6, 7, 12–24]. The re-sampling methods usually run in linear time but yield relatively lower quality outputs. On the other hand, the iterative contraction methods usually produce very good quality outputs, but they run much slower than the spatial partitioning methods. In applications such as the 3D computer games, the discrete LOD is preferred for its low processing cost at running time, which gives more time to calculations such as the collision detection, scene update, and game AI, etc. [2]. Since we are interested in generating high-quality static LOD meshes, the review will be restricted to those using iterative edge collapses.

The mesh simplification is first treated as an optimization problem by Hoppe et al. [13]. In their work, three topological operators – the *edge collapse*, *vertex split*, and *edge swap* are applied iteratively to minimize a heuristic energy function for estimating the deviation between the current mesh and the original surface. The algorithm produces very good quality outputs; however, the runtime cost is unacceptably long due to its exponential complexity of running time. In a later work [14], they proposed a data structure called the *progressive mesh* (PM) generated by a greedy-based framework using iterative edge collapses. According to their paper, in addition to the contribution of the new data structure, the time complexity was also considerably improved to $O(n\log n)$ provided that the input mesh has n faces.

Similar to the plane-based error metrics proposed by Ronfard and Rossignac [15], Garland et al. proposed the quadric error metrics (QEM) [16] to calculate the sum of squared distances from a relocated vertex to the set of planes spanned by all faces in its neighborhood. In the light of their work, the QEM can be extended to handle vertex attributes such as vertex normal, vertex color, and texture coordinate [19–23]. A more recent work has even extended this metrics to any dimension [24]. In [16], the face quadrics as well as all the vertex quadrics are computed and saved before simplification. Lindstrom and Turk suggested the computation of vertex quadrics is performed on the fly with the simplification [3]. This scheme eliminates the need of storage space for saving the vertex quadrics and is called the memoryless simplification. According to their results, this scheme generates better quality outputs at the cost of poorer runtime efficiency.

Opposed to conventional greedy based approach, another steam of work using multiple-choice technique was proposed in [6]. This methods runs in linear time; however, it requires a random sampling process that may require $O(d^2)$ time to get d different samples. A later work extended this scheme to simplify large meshes [7]. The improved algorithm is successful, but the input stream is assumed to be a sequence of spatially ordered polygon soup. Once if the input mesh does not satisfy this assumption, an external sort

over the input stream that takes at least $O(n\log n)$ might be required; otherwise, the simplification might fail to proceed if the input mesh contains many holes.

2.2 The review of the leve-of-detail schemes

The basic principles of LOD techniques were proposed by Clark in 1976, in which he remarked that it is redundant to use many polygons for an object while it covers only a few pixels [25]. The hierarchical scene graph structure and concept of view-frustum culling were also proposed in the literature to eliminate such redundancy. To address such issue, tons of works are proposed thereafter. In accordance with [2], three basic frameworks for managing LOD are available: the discrete, continuous, and view-dependent LOD.

The *discrete LOD*, or known as the *static LOD*, is the traditional approach proposed by Clark [25]. This approach adopts a set of fixed size meshes in various resolutions generated offline by a mesh simplification algorithm. A runtime daemon is employed in 3D computer games to select an appropriate mesh from the LOD meshes according to certain criteria such as those proposed by [26]. The discrete LOD scheme suffers from the hopping side effect that leads to visual disturbance, which can be minimized by blending the rendered images of successive LOD meshes [27]. Despite the inefficiency of memory consumption and the visual disturbances caused by the hopping effect during LOD switching, the discrete LOD is still by far the most commonly used LOD scheme in the interactive 3D applications.

Opposed to the discrete LOD, the *continuous LOD*, or the *dynamic LOD*, is essentially a continuous spectrum of LODs essentially consists of a coarse mesh and a series of refinement records [2]. Rather than selecting an appropriate LOD mesh, the LOD manager has to synthesize the desired mesh by applying necessarily refinements to the coarse mesh at runtime. Such a LOD scheme is also called the progressive mesh (PM) [14, 23]. This LOD scheme can be considered as a lossless compression method if the resulting mesh size is smaller than the original. However, it takes a great amount of time to generate the desired mesh at runtime; hence, it is usually not applied in interactive 3D multimedia applications, e.g., the 3D computer games and virtual reality.

The *view-dependent LOD* is intrinsically a hierarchical representation of the original mesh, which supports range queries to a certain region of the mesh [2]. Examples can be found in [28–32]. The major benefit of this scheme is a better distribution of the polygon resource, which ensures better fidelity. Such a scheme is usually employed in terrain rendering, scientific visualizations, and large model inspection or editing applications.

3 Terminologies

Our new algorithm adopts vertex-ring-based simplification, iterative full edge collapses, the quadric error metrics, and a constant size RS-heap. Thus, before introducing our work, a brief review on a number of related works and terms are given as follows.

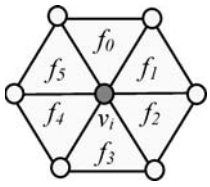

Fig. 2 The ring of a vertex v_i

3.1 Topological notations

In topology convention, an n-simplex is a topological entity defined by $n+1$ vertices. Hence, a 0-simplex is a vertex, a 1-simplex is an edge, and a 2-simplex is a triangle, and so forth. Given an n-simplex s, the $(n-1)$-simplices that bound it are called the faces of s. Therefore, the edges that constitute a triangle t are the faces of t, and the endpoints of an edge e are the faces of e. Let s be an n-simplex and S be a set of n-simplices, the following operators are used in this paper.

$\lceil s \rceil$: the set of $(n+1)$-simplices that s is a face of.
$\lfloor s \rfloor$: the set of faces of the n-simplex s.
∂S: the boundary faces of S.

3.2 The ring

Let r_i be the ring of a vertex v_i. With the notations from Sect. 3.1, it can be denoted as $\lceil\lceil v_i \rceil\rceil$, which is essentially the set of triangles adjacent to v_i, or $r_i = \{f_j | f_j \in \lceil\lceil v_i \rceil\rceil\}$. In the example shown in Fig. 2, the ring of the vertex v_i is $r_i = \{f_0, f_1, f_2, f_3, f_4, f_5\}$.

Therefore, the internal edges (*stars*) of $r_i = \lceil v_i \rceil$, the boundary edges (*crown*) of $r_i = \partial r_i = \lfloor r_i \rfloor - \lceil v_i \rceil$, and the boundary vertices of $r_i = \lfloor \lceil v_i \rceil \rfloor - \lceil v_i \rceil$.

3.3 The edge collapse

Given an undirected edge $\overline{v_s v_t}$ connecting vertices v_s and v_t, let the ring of v_s and v_t be r_s and r_t, respectively. The edge collapse operator that removes $\overline{v_s v_t}$ from the input mesh $M(V, F)$ generates a simplified mesh $M'(V', F')$ through the following procedure.

1. Find $I = r_s \cap r_t$ and let $S = r_s - I$.
2. Let $F' = F - I$ and $V' = V - v_s$.
3. $\forall f \in S$, replace the index of v_s with that of v_t.
4. Relocate v_t such that the error cost ε is minimized.

Note that Step 4 is usually computed in advance to facilitate global optimization with respect to the error costs. An example is presented in Fig. 3 where v_t is relocated to v_n.

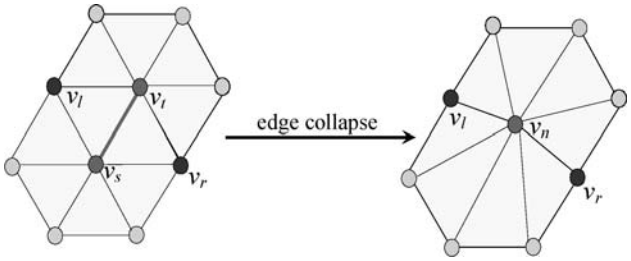

Fig. 3 An example of the edge collapse

3.4 The quadric error metrics

Unlike the plane-based error metrics proposed by Ronfard and Rossignac that measures the maximum distance from a set of planes to a given vertex, the quadric error metrics (QEM) efficiently calculates the sum of squared distances from a set of planes to the vertex by matrix techniques [15, 16]. Since we have adopted such metrics in our new algorithm, we will briefly review it as follows.

Given a plane $ax + by + cz + d = 0$ or $\mathbf{n}^{\mathbf{T}}\mathbf{v} + d = 0$ where $\mathbf{n} = [\,a\ b\ c\,]^{\mathbf{T}}$ is the unit face normal and d is a scalar constant, the squared distance from a vertex $\mathbf{v} = [\,x\ y\ z\,]^{\mathbf{T}}$ to the plane can be depicted below:

$$D(\mathbf{v}) = (\mathbf{n}^{\mathbf{T}}\mathbf{v} + d)^2 = \mathbf{v}^{\mathbf{T}}\mathbf{n}\mathbf{n}^{\mathbf{T}}\mathbf{v} + 2d\mathbf{n}^{\mathbf{T}}\mathbf{v} + d^2. \tag{1}$$

Then the *fundamental quadric* $\mathbf{Q}$ of this plane is defined as a 4×4 symmetric matrix as follows:

$$\mathbf{Q} = \begin{bmatrix} \mathbf{A} & \mathbf{b} \\ \mathbf{b}^{\mathbf{T}} & d^2 \end{bmatrix} = \begin{bmatrix} \mathbf{n}\mathbf{n}^{\mathbf{T}} & d\mathbf{n} \\ d\mathbf{n}^{\mathbf{T}} & d^2 \end{bmatrix} = \begin{bmatrix} a^2 & ab & ac & ad \\ ab & b^2 & bc & bd \\ ac & bc & c^2 & cd \\ ad & bd & cd & d^2 \end{bmatrix}. \tag{2}$$

Accordingly, the squared distance from a vertex v to the plane can be calculated by

$$Q(\mathbf{v}) = \bar{\mathbf{v}}^{\mathbf{T}}\mathbf{Q}\bar{\mathbf{v}}, \tag{3}$$

where $\bar{\mathbf{v}} = [\,\mathbf{v}^{\mathbf{T}}\ 1\,] = [\,x\ y\ z\ 1\,]^{\mathbf{T}}$ is the vertex v in a homogeneous coordinate system.

Given a vertex v, its vertex quadric is defined as the sum of the fundamental quadrics of the planes spanned by the faces incident to v. Let r_i be the set of faces incident to vertex v_i and $\mathbf{Q}_j$ be the face quadric of face f_j, where $f_j \in r_i$. Thus, the vertex quadric of v_i is

$$\mathbf{Q}_i^v = \sum_{\forall f_j \in r_i} \mathbf{Q}_j. \tag{4}$$

Considering an edge $\overline{v_s v_t}$, let the edge collapse perform the removal of $\overline{v_s v_t}$ and introduce a new vertex v_n. The distance from v_n to the original surface can be estimated by the distance from v_n to the set of planes spanned by the surfaces incident to v_s or v_t. Let r_s and r_t be the rings of v_s and v_t, respectively. The error distance from v_n to the original surface can be given by $Q_n^v(\mathbf{v}_n)$, where

$$\mathbf{Q}_n^v = \mathbf{Q}_s^v + \mathbf{Q}_t^v = \sum_{\forall f_i \in r_s} \mathbf{Q}_i + \sum_{\forall f_i \in r_t} \mathbf{Q}_i. \tag{5}$$

The error distance may be further minimized by solving an optimal placement of the new vertex v_n from $\nabla Q_n^v(\mathbf{v}_n) = 2\mathbf{A}\mathbf{v}_n + 2\mathbf{b} = \mathbf{0}$, or $\mathbf{A}\mathbf{v}_n = -\mathbf{b}$. Note that the linear system has a unique solution only when the matrix $\mathbf{A}$ is non-singular or invertible.

3.5 The replacement selection heap

The RS-heap is primarily used to generate sorted runs in the first phase of large file sorting [4]. It has at least two advantages when it is compared with other in-place sorting technique in sorted runs generation. First, with an array of n locations, it is capable of generating sorted runs of length equals $2n$ items in average for uniform random inputs. Moreover, when the file is approximately ordered, in specific, if for each item in the input stream, there are no more than n larger items before it, the input file can be sorted in a single pass.

The records inserted to the RS-heap may fall into one of two categories: current-run records or next-run records. In a minimum RS-heap, if the inserted record is less than the recently deleted record, it is tagged as a next-run record and inserted to the next-run part of the heap; otherwise, it is regarded as a current-run item and placed at the root of the heap followed by a sift-down operation to adjust the heap. We will assume the use of minimum RS-Heap throughout this context unless otherwise stated.

An RS-heap may have three states: full of current-run records, a mixture of current-run and next-run records, and full of next-run records. To keep track of the boundary of the current-run part and next-run part of the heap, a pointer called trailing-current-run (TCR) pointer, denoted as TCP, are used to point to the last current-run record of the heap. When a heap is full of next-run records, a state change is made by resetting the TCR pointer to the end of the heap, which turns all the next-run records into current-run records and start out a new sorted run.

The operation of an RS-heap comprises three stages, i.e., the initial, operation, and finale stages. Initially, the heap is full of null records. At the initial stage, the algorithm iteratively deletes null records at root and inserts valid records as next run items to the heap until the heap is full of valid records. At the operation stage, valid records from input stream are repeatedly inserted on the fly with the deletion of records from the root of the heap until the end of the stream. Following to the operation stage, the finale stage keep inserting null large key value records to the next-run part of the heap to ensure that all the remaining valid records are deleted from the heap.

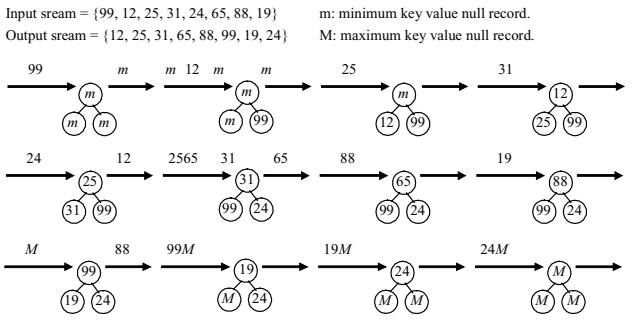

Fig. 4 An example of the process of sorted runs generation using the RS-heap

The root deletion and insertion operations are applied alternatively during the execution of the algorithm. Before the new record is inserted, it has to be compared with the recently deleted record: if the new record is smaller, it is a next-run record; otherwise, it is a current-run record. A next-run record is inserted to the next-run part of the heap according to the following steps:

(a) Move the TCR record to the root.
(b) Insert the new record to the location of TCR record.
(c) Adjust the TCR pointer.
(d) Apply a sift-down operation to the sub-tree rooted by the new record.

On the other hand, a current-run record is simply placed at the root followed by a sift-down operation from the root to the TCR record. Figure 4 shows an example illustrating such sorted runs generation process using the RS-heap.

4 The simplification algorithm

The input of our algorithm are an indexed face mesh $M(V, F)$ comprising a set of vertices V in a 3D Cartesian space and a set of triangular faces F represented by a triplet of indices to V together with a variable *goal* specifying the minimum number of faces of the output mesh. In our implementation, the vertex and face tables, respectively, store the vertices and faces of the input mesh.

There are seven data structures used by our algorithm, i.e., the vertex/face arrays, vertex rings, vertex/face/ contraction flags, and vertex indices. A concise summary of these data structures is listed in Table 1, which gives notations, data types, element sizes, and the number of elements.

The ring table stores the vertex rings providing local connectivity information of each vertex. The vertex and face flags respectively indicate the presence of corresponding vertices and faces, which are primarily used for exporting the simplified mesh at the output stage. The third flag, or the contraction flag, may be set to one of three possible values: *null*, *valid*, or *dirty*, reflecting the contraction state of a ring. All vertex rings are initialized to valid before the computation of contractions to the vertex rings. After the computation, if the procedure fails to find a valid contraction for a vertex ring then its contraction flag will be set to null. In addition, the contraction flag of a vertex ring is set to dirty if the vertex ring is a dependent ring of a previously performed contraction. The definition of the dependent ring and how it is related to the contraction will be given in Sect. 4.3. The last table, or the vertex index table, keeps the index of each remaining vertex in the new output mesh, which is used to export valid faces at the output stage.

4.1 Algorithm overview

In the main, the algorithm runs iteratively through three stages, i.e., the vertex ring construction, simplification, and output stages, until the termination condition is satisfied. A fragment of the pseudo-code of the new algorithm is given as follows.

Algorithm: HighQualityMeshSimplification()
Input:
IndexFaceMesh *M*;
Integer *goal*.
Output:
IndexFaceMesh *DLM*[MAX_NO_OF_LEVELS].
Procedure:
ContractionRecord **ComputeContractionCost**(VertexRing *r*);
void **HeapInsert** (Heap *H*, ContractionRecord *C*);
ContractionRecord **HeapDeleteRoot** (Heap *H*);
void **EdgeCollapse**(ContractionRecord *C*);
IndexFaceMesh **OutputLOD**(void);

Table 1 The seven data structures and their sizes

Data	Notation	Element type	Element size (bytes)	Number of elements
Vertex table	V	VertexT	12	$\|V\|$
Face table	F	FaceT	12	$\|F\|$
Ring table	R	VRingT	$4 \times 6^{a} + 5^{b}$	$\|V\|$
Vertex flags	F^{v}	Byte	1	$\|V\|$
Face flags	F^{f}	Byte	1	$\|F\|$
Contraction flags	F^{c}	Byte	1	$\|V\|$
Vertex index table	V^{i}	Unsigned integer	4	$\|V\|$

[a]On an average, the degree of vertex connectivity is 6.
[b]The basic storage cost of VRingT, which comprises an unsigned character used to keep the number of faces in the ring and a pointer to the array of face indices.

begin
int $l = 1$;
1. Initialize the flags and construct the set of vertex rings, $R = \{r_i | \forall v_i \in V\}$.
2. Simplify the input mesh:
   1) For $i = 1 \sim k$, where k is a small constant integer and $k \geq 1$:
      C_i = **ComputeContractionCost**(r_i);
      HeapInsert(H, C_i).
   2) For $i = k+1 \sim |V|$:
      If $f_i^c = 1$,
         C_i = **ComputeContractionCost**(r_i);
         C_s = **HeapDeleteRoot**(H);
         HeapInsert(H, C_i);
      If $f_t^c = 1$,
         perform **EdgeCollapse**(C_s);
         If $|F| < goal$, proceed to Step 3.
   3) Repeat the following operations until $|H| = 0$ or $|F| < goal$:
      C_s = **HeapDeleteRoot**(H);
      If $f_t^c = 1$,
         perform **EdgeCollapse**(C_s);
         If $|F| < goal$, proceed to Step 3.
3. $DLM[l]$ = **OutputLOD**(); $l = l+1$;If $|F| >$ goal, repeat Steps 1–3.

end

During each pass of the iteration, the algorithm starts with the construction of vertex rings. Following the vertex rings construction, the simplification is performed to simplify independent vertex rings until the termination condition is satisfied or no more vertex ring is valid to contract. At the finale stage, the remaining vertices and faces are output as a new level-of-detail approximation and as the input mesh to the next pass of iteration. Consequently, if the simplification takes k iterations, k level-of-detail approximations are generated.

4.2 The vertex ring construction stage

In addition to the initialization of the three flag variables, the vertex ring construction stage sets up the rings of every vertex by traversing the entire set of faces. In our implementation, the rings consist of a dynamic allocated array of face indices and a variable keeping the number of faces in the ring, which is declared as follows.

```
VRingT = {
    unsigned char degree;
    unsigned int * FaceIndexArray; }
```

The construction operations are summarized in the following:

For each face $f_i \in F$, $i = 1 \sim |F|$:
For each vertex v_j of f_i:
1. Insert f_i to r_j by:
   Allocate an array of $r_j.degree + 1$ integers.
   Copy the $r_j.degree$ items from the original array to the newly allocated array.
   Link the array pointer to the new array.
   Delete the old array.
2. Increase degree of r_j;

In spite of the overhead introduced by frequent allocation and deletion of small-sized arrays, such implementation has at least two benefits. First, the direct access to face indices of the vertex ring rather than sequential pointer list traversal is obviously more time efficient. In comparison with the pointer list approach, the new implementation uses much less space by using only one pointer. Contrast with our previous approach [33, 34], the new implementation sets no limit to the size of the ring; hence, it is more flexible and uses much less memory space.

4.3 The simplification stage

The simplification process has three sub-stages corresponding to the three phases of sorted runs generation using the RS-heap, namely, the initialization, operation, and finale in order [4]. A flow chart shown in Fig. 5 illustrates the simplification pipeline.

At the initial sub-stage, the algorithm simply performs the cost computation and heap insertion operations repeatedly until the RS-heap is full of valid contraction records. Next, at the operation sub-stage, the algorithm iteratively executes the cost computation, heap operations, and a full edge collapse according to the contraction record retrieved from the root of the heap if it passes the dependency check. At the finale sub-stage, since all the rings are processed, no cost computation is required. Instead of inserting new contraction records, null records with largest possible costs are inserted to the heap. The iteration stops when all the remaining valid records are deleted from the heap or the termination condition is satisfied The detailed discussion of each step will be given in the following subsections.

4.3.1 The dependency rules

In a strict way, an edge collapse is called the dependent edge collapse of another edge collapse if its cost may vary after the execution of that edge collapse. Most previous works simply tag the boundary vertices and decline all the edge collapses that involve these vertices [17]. However, this approach requires a lot of minor important updates and takes many levels of iteration. In our previous work [34], we have developed a set of rules for dependency control that provides various levels of relaxation to the definition of dependent edge collapses. A similar version of such dependency control strategies used in this paper is defined as follows.

We can define three classes of *dependent rings* according to the amount of modifications resulting from the execution of an edge collapse. A formal definition is given below.

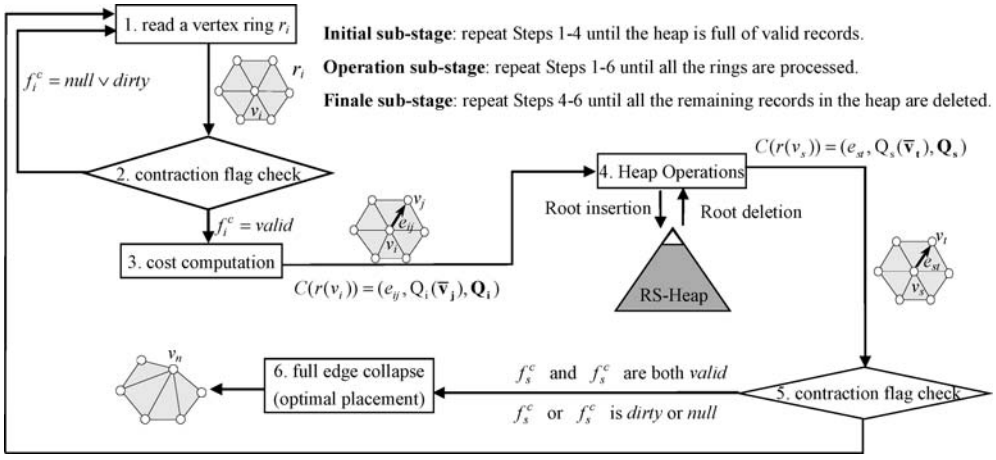

Fig. 5 A flow chart of the simplification stage

Definition 1 The Three Classes of Dependent Rings

Considering the full edge collapse of an edge $e_{st} = \overline{v_s v_t}$, the *dependent rings* of the full edge collapse are defined as follows.

1. Class A dependent rings of e_{st}, $R_A = \{r_s, r_t\}$;
2. Class B dependent rings of e_{st}, $R_B = \{r_j | e_{st} \in \partial r_j\}$;
3. Class C dependent rings of e_{st}, $R_C = \{r_j | v_s \in \lfloor \lceil v_j \rceil \rfloor - v_j \vee v_t \in \lfloor \lceil v_j \rceil \rfloor - v_j\}$.

An example illustrating the three classes of dependent rings is given in Fig. 6, where the shaded regions represent the dependent rings. According to Fig. 6, we can easily find that in the Class A rings, an internal edge is deleted; in the Class B rings, a boundary edge is deleted; whereas in the Class C rings, none of the edges of the ring is deleted.

On the basis of the definition given above, we can derive three dependent control strategies for various extent of dependency control as follows.

Definition The Dependent Control Strategies

Given three classes of dependent rings: R_A, R_B, and R_C, of an edge collapse, the three dependency control strategies, Strategy I, II, and III, respectively, decline the collapse of the edges in

$$E_I = \{e \mid e \in R_A - \partial R_A\},$$
$$E_{II} = \{e \mid e \in R_A - \partial R_A \cup R_B - \partial R_B\},$$
$$E_{III} = \{e \mid e \in R_A - \partial R_A \cup R_B - \partial R_B \cup R_C - \partial R_C\}.$$

In our implementation, a sentinel called the contraction flag is used to keep track of the contraction state of a vertex ring. If the contraction flag of a vertex ring is dirty, it is not contractable. Furthermore, if the contraction flag of one of the endpoints of an edge is dirty, the collapse of this edge is not valid. Since we are using a small constant heap, the algorithm will avoid ring contractions by collapsing such edges.

It is important to know the portion of contracted rings, denoted as P_c, in a pass of simplification, which determines not only the running time efficiency of the algorithm but also the number of LOD meshes generated. Provide that the input mesh is manifold mesh, the portion of contracted rings is roughly equal to the ratio of contracted rings over its dependent rings. Let $P_{I,c}$, $P_{II,c}$, and $P_{III,c}$, respectively denote the portion of contracted rings using Strategy I–III. Thus,

$$P_{\mathrm{I},c} = \frac{|\{r_s\}|}{|R_A|} = \frac{|\{r_s\}|}{|\{r_s, r_t\}|} = \frac{1}{2} = 50\%, \tag{6}$$

$$P_{\mathrm{II},c} = \frac{|\{r_s\}|}{|R_A \cup R_B|} \cong \frac{|\{r_s\}|}{|\{r_s, r_t, r_l, r_r\}|} = \frac{1}{4} = 25\%, \tag{7}$$

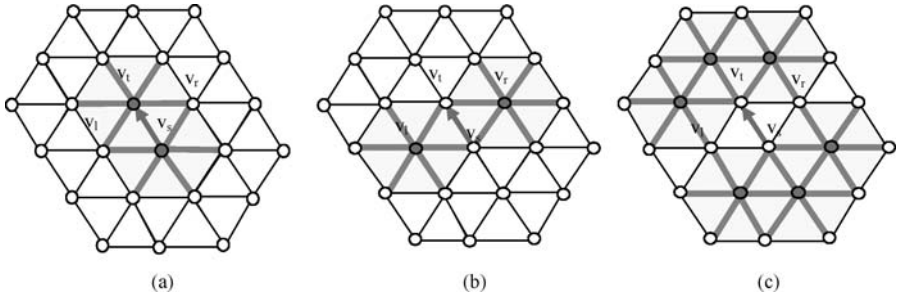

Fig. 6 **a** Class A, **b** Class B, **c** Class C dependent rings of the edge collapse of e_{st}

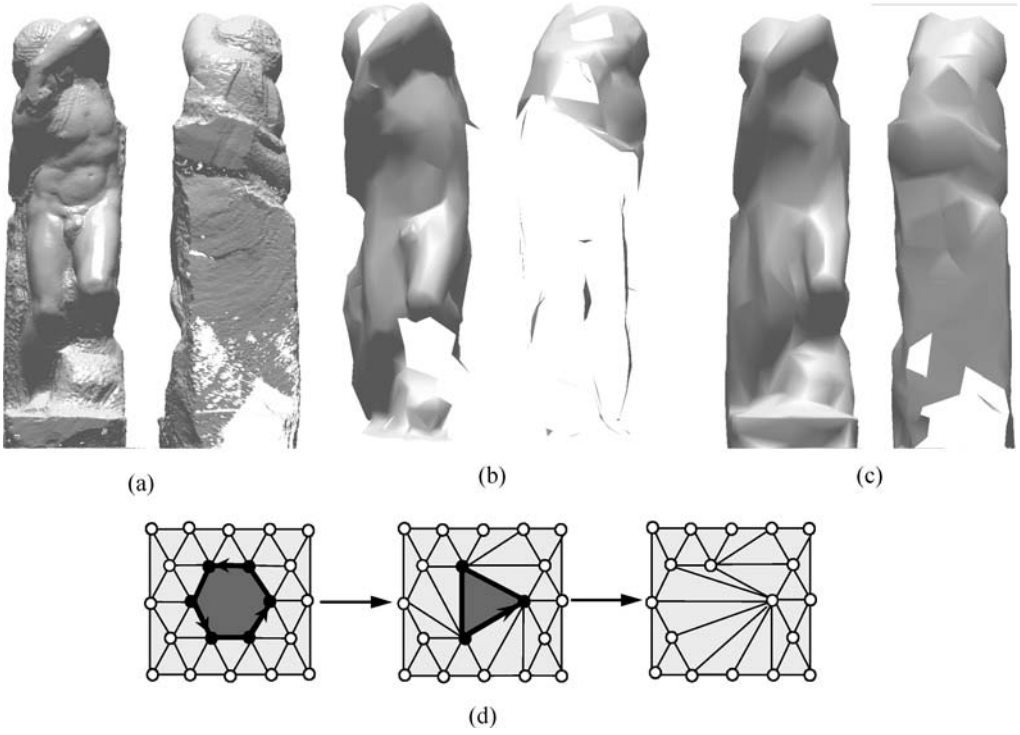

Fig. 7 An example of holes in a mesh and the hole-filling process, **a** the original Youthful mesh: both its front and back sides contain numerous holes, **b** the simplified Youthful mesh (1,000) triangles without hole filling, **c** the simplified Youthful mesh (1,000 triangles) with hole filling, **d** an example of the hole-filling process

and for the average case

$$P_{\mathrm{III},c} = \frac{|\{r_s\}|}{|R_A \cup R_B \cup R_C|} \cong \frac{1}{10} = 10\%. \tag{8}$$

However, it is possible that all the internal edges of a ring are invalidated by previous contractions, which makes the ring turns into *null* state. Fortunately, this situation is rare in practice. According to the evaluations presented in [34], Strategy I is the best policy; hence, we will adopt it in this paper.

4.3.2 The cost computation and hole filling

From the experimental outcomes, we have found some visual defects resulting from the enlargement of the tiny holes of the original mesh as shown in Fig. 7a. Such tiny holes looked insignificant when the mesh is at high resolution. As the simplification proceeds, if these holes are not properly treated, they will become large cracks indicated in Fig. 7b.

To treat such a problem, we enforce the contraction to the ring of boundary vertices to the boundary edges of the mesh. An example of such process is illustrated by Fig. 7d. To do this, we first calculate the multiplicity of each internal edge during the construction of the set of boundary vertices of the ring. If the ring has some internal edges whose multiplicity equals one, we will enforce the contraction of the ring by collapsing one of these edges. Figure 7c shows the simplification results of our algorithm on the Youthful (2 mm) mesh from the Project of Michelangelo [2], which proves that our algorithm can effectively fill such holes rather than enlarge it. What follows is the pseudo-code for such cost computation and hole-filling procedure computing the best contraction C_i for a vertex ring r_i.

Procedure CostComputation(VRingT r_i)
Input: VRingT r_i.
Output: ContractionRecord C_i.
begin
  Let *cost* be a large number.
  Compute the vertex quadric $\mathbf{Q}_i^v$ of v_i, $\forall v_i \in r_i$.
  Find $b_i = \lfloor \lceil v_i \rceil \rfloor - v_i$.
  Compute the multiplicity of e_{ij}, $m(e_{ij})$, $\forall$ internal edges $e_{ij} = (v_i, v_j) \in r_i$, where $v_j \in b_i$.
  For all v_j of b_i:
  If there exists an e_{ij} in r_i whose multiplicity $m(e_{ij}) = 1$,
    If $m(e_{ij}) = 1$ and $cost \geq Q_i^v(v_j)$,
      $cost = Q_i^v(\mathbf{v}_j)$;
      $C_i = (e_{ij}, Q_i^v(\mathbf{v}_j), \mathbf{Q}_i^v)$;
  else
    If $cost \geq Q_i^v(\mathbf{v}_j)$,
    $cost = Q_i^v(\mathbf{v}_j)$;
    $C_i = (e_{ij}, Q_i^v(\mathbf{v}_j), \mathbf{Q}_i^v)$.
end

Note that the procedure chooses from valid internal outward half-edges the one with smallest error cost computed by the

quadric error metric. Actual full edge collapse with optimum placement is performed later in the edge collapse procedure.

The full edge collapse

The full edge collapse of our algorithm is similar to the one in Sect. 3.3 except that the optimal placement is not computed in advance. Instead, we find the optimal position of the new vertex only after the edge collapse is actually performed. Consequently, all the unnecessary optimum placement computations are avoided. Furthermore, to avoid wrong solutions caused by numerical errors, the placement of the new vertex is computed according to the following formula:

$$\mathbf{v}_n = \begin{cases} \mathbf{v}_t & \text{if } |\overline{\mathbf{v}_s\mathbf{v}_o}|^2 + |\overline{\mathbf{v}_o\mathbf{v}_t}|^2 \geq |\overline{\mathbf{v}_s\mathbf{v}_t}|^2 \\ \mathbf{v}_o & \text{otherwise,} \end{cases} \tag{9}$$

where $\mathbf{v}_o$ is the calculated optimal placement of the new vertex v_n.

The pseudo-code of the edge collapse procedure is given as follows.

Procedure EdgeCollapse(ContractionRecord C)
Input: ContractionRecord $C = (e_{st}, \mathrm{Q}_s(v_t), \mathbf{Q}_s)$;
begin
Delete v_s from V;
Find $I = r_s \cap r_t$ and let $S = r_s - I$;
Delete all f of I from F;
Replace the index of v_s with that of v_t, $\forall f \in S$;
Set both f_s^c and f_t^c to be *dirty*;
Calculate $\mathbf{Q}_t^v$ and let $\mathbf{Q}_t^v = \mathbf{Q}_s^v + \mathbf{Q}_t^v$;
Find the optimal placement v_o via solving

$$\nabla Q_t^v(\mathbf{v}_o) = 2\mathbf{A}\mathbf{v}_o + 2\mathbf{b} = \mathbf{0};$$

Relocate v_t to v_n in accordance with

$$\mathbf{v}_n = \begin{cases} \mathbf{v}_t & \text{if } |\overline{\mathbf{v}_s\mathbf{v}_o}|^2 + |\overline{\mathbf{v}_o\mathbf{v}_t}|^2 \geq |\overline{\mathbf{v}_s\mathbf{v}_t}|^2 \\ \mathbf{v}_o & \text{otherwise.} \end{cases}$$

end

4.4 The output stage

In this paper, we only concentrate on the real-time interactive 3D multimedia applications such as 3D computer games and virtual reality. In consequence, we apply our linear time algorithm to generate the discrete LOD only. The output stage exports the remaining vertices and faces. Since the algorithm is iteratively performed to simplify independent vertex rings of the input mesh, the output stage is capable of generating a new LOD mesh in the end of each pass of iteration. If the algorithm takes k passes to simplify the mesh to the coarsest resolution, k LOD meshes can be generated. These LOD meshes constitute a set of discrete LOD representations of the input mesh in which the original mesh is the level-0 LOD mesh, or M^0, and the output mesh of the i-th pass of iteration is the level-i LOD mesh, or M^i.

4.5 Time complexity analysis

Let $M_1(V_1, F_1)$, $M_2(V_2, F_2), \ldots,$ and $M_i(V_i, F_i)$ be the outputs from the 1st, 2nd, ... , and i-th pass of simplification, respectively. By applying Strategy I of dependency control, about 50% rings are contracted for each pass of iteration according to Eq. (6). Thus,

$$\begin{aligned} |V_1| &\approx n - \frac{n}{2}, \\ |V_2| &\approx |V_1| - \frac{|V_1|}{2} = \frac{|V_1|}{2} \approx n \cdot \left(\frac{1}{2}\right)^2, \\ &\vdots \\ |V_i| &\approx |V_{i-1}| - \frac{|V_{i-1}|}{2} = \frac{|V_{i-1}|}{2} \approx n \cdot \left(\frac{1}{2}\right)^i. \end{aligned}$$

Let n_0 be the number of vertices of the lowest resolution mesh specified by the termination condition variable and the simplification successfully stops during the m-th pass of simplification. Since the number of remaining vertices after the m-th pass of simplification is roughly equal to $n \cdot (\frac{1}{2})^m$,

$$n_0 \geq n \cdot \left(\frac{1}{2}\right)^m, \tag{10}$$

By taking logarithms on the both sides, we have

$$\begin{aligned} &\log_2 n_0 \geq \log_2 n - m \\ &\Rightarrow m \leq \log_2 n - \log_2 n_0 \\ &\Rightarrow m = \mathrm{O}(\log n) \end{aligned} \tag{11}$$

Let $T_{1,i}$, $T_{2,i}$, and $T_{3,i}$, respectively, be the running times of the vertex construction, simplification, and output stages of the i-th pass of simplification. The time complexity of the algorithm is

$$\sum_{i=1}^{m} (T_{1,i} + T_{2,i} + T_{3,i}). \tag{12}$$

Since the preprocessing stage needs only a single pass over the entire face table, the running time of this stage in the i-th pass of simplification is

$$T_{1,i} = \Theta(|F_i|) = \Theta(|V_i|). \tag{13}$$

The simplification stage comprises three sub-stages, i.e., the initialization, operation, and finale sub-stages, which takes $|V_i| + h$ steps for the i-th pass of simplification if the heap size is h. Since the running time of each step is the sum of the running times of a cost computation, two heap operations, and a full edge collapse, if the heap size h is a small constant value, the total cost of each step remains constant, or O(1), with respect to the input size n. Therefore, the running time for the i-th pass of simplification of the simplification stage is

$$T_{2,i} = \sum_{i=1}^{|V_i|+h} \mathrm{O}(1) = \Theta(|V_i|). \tag{14}$$

Table 2 Test models and their statistics

Model	Number of vertices	Number of faces
Rabbit	67,038	134,074
Dragon	437,645	871,414
Happy Buddha	543,652	1,087,716
Blade	882,954	1,765,388
David (2 mm)	3,614,098	7,227,031
Dawn (1 mm)	3,432,236	6,594,103
Youthful (2 mm)	1,728,305	3,411,563
Awakening (2 mm)	2,057,930	4,060,497

The output stage consists of two loops exporting remaining vertices/faces from the vertex/face table. Thus, the running time for the i-th pass of simplification of this stage is

$$T_{3,i} = \Theta(|V_i|) + \Theta(|F_i|) = \Theta(|V_i|). \quad (15)$$

From Eqs. (12)–(15), the overall running time of our algorithm is

$$T = \sum_{i=1}^{m} (T_{1,i} + T_{2,i} + T_{3,i}) = \Theta(n). \quad (16)$$

5 Experimental results

In this paper, all the experiments were executed on a personal computer equipped with an Intel Pentium IV 2.2 GHz processor, 1 GB DDR DRAM, and an disk RAID of two IDE 7,200 RPM hard disks running Microsoft Windows 2000 SP4. All the program codes are written in C++ and compiled by Microsoft VC++ 6.0. To provide enough evidence, eight meshes are used to verify the effectiveness of the new algorithm. The Rabbit, Dragon, Happy Buddha, and Blade meshes are retrieved from the Stanford 3D Scanning Repository (http:// www-graphics.stanford.edu/data/3Dscanrep/), whereas the David (2 mm), Dawn (1 mm), Youthful (2 mm), and Awakening (2 mm) meshes are downloaded from the Digital Michelangelo Project Archive of 3D Models (http://www-graphics.stanford.edu/dmich-archive/) [35]. The statistics of these meshes are listed in Table 2.

Two famous iterative edge collapse-based methods, the QSlim V2.0 [36], M. Garland's implementation of [16], and the multiple choice algorithm (MCA) [6], are used to provide a fair basis of comparisons. The QSlim V2.0 is a conventional greedy-based approach that has $O(|V| \log |V|)$ time complexity while the MCA is a multiple random choice-based algorithm running in linear time. Since we do not have the author's implementation of the MCA, we will use our own implementation of it in the experiments. Hence, the results may be slightly different from those presented in [6]. In order to measure the geometric errors, another public domain available package, Metro V3.1 [37], is employed to analyze the resulting meshes. We will compare the three different approaches in terms of the main memory costs, runtimes efficiency, and output quality through Sects. 5.1–3. Note that, in all experiments, we let $d = 8$ and heap size = 512 records for the MCA and HQMS methods, respectively.

5.1 Memory costs

According to Table 1, our new algorithm (HQMS) requires $47 \times |V| + 13 \times |F| \cong 73 \times |V|$ bytes for buffering both the input and intermediate data. In comparison with the QSlim and MCA methods that cost about $268 \times |V|$ and $105 \times |V|$ bytes,[1] respectively [6, 16], the main memory cost of the HQMS is significantly lower. Since the main memory constraint is 1GB, QSlim V2.0 cannot be employed to simplify the David (2 mm) and Dawn (2 mm) meshes. Table 3 presents the peak main memory costs of the QSlim, MCA, and HQMS methods during the simplification of the eight test meshes.

According to the results shown in Table 3, it is obvious that the HQMS uses much less main memory than the other two methods, which implies that, with the same amount of main memory resource, HQMS is capable of simplifying larger meshes than the other two methods.

5.2 Runtime efficiency

To analyze the runtime performance of the HQMS, the eight test meshes are simplified down into 1,000 triangles using the three methods. The runtime performance in terms of the overall execution times and triangle reduction rates of the three approaches are presented in Table 4. For the same reason as we have stated in previous subsection, QSlim is not able to simplify the David and Dawn meshes with 1 GB main memory; thus, the results on these two meshes are not available.

The results in Table 4 shows that the new linear time algorithm HQMS is not only faster than the $O(|V| \log |V|)$ greedy-based QSlim V2.0 but also significantly faster than the linear time MCA method. The triangle reduction rate of the HQMS is around five times higher than that of the QSlim V2.0 and three times higher than that of the MCA, which are higher than 130 K triangles (1 K = 1,000) per second. Obviously, we can conclude that our new linear time approach is significantly better than the other two methods in terms of runtime efficiency.

5.3 Output quality

The quality measurement is difficult and depends on the applications in use. In this paper, the evaluation is given by means of the geometric errors and the rendered images of the outputs. However, owing to the limitation imposed by the Metro V3.1, such measurements cannot be performed to meshes whose sizes are larger than 2 millions triangles [37]. Therefore, the comparisons on geometric errors are restricted to the LOD meshes of the Rabbit, Dragon, Happy Buddha, and Blade meshes.

[1] $40 \times |V|$ bytes for the vertex quadrics, $29 \times |V|$ bytes in average for the vertex rings, and $36 \times |V|$ bytes for the base mesh.

Table 3 The main memory costs (KB)

	Rabbit	Dragon	Happy Buddha	Blade	David	Dawn	Youthful	Awakening
QSlim V2.0	25,048	158,136	197,152	314,444	N/A	N/A	613,848	735,304
MCA	12,936	81,160	100,760	161,488	657,952	616,188	312,872	372,164
HQMS	6,608	40,052	49,888	78,808	314,840	293,884	148,672	178,388

Table 4 The running times and reduction rates

	Rabbit	Dragon	Happy	Blade	David	Dawn	Youthful	Awakening
Running time (s)								
QSlim	4	26	32	50	N/A	N/A	110	136
MCA	2	17	21	34	162	143	70	83
HQMS	1	6	7	12	55	50	25	30
Reduction rate (KTriangles/s, 1 K = 1,000)								
QSlim	33	33	34	35	N/A	N/A	31	30
MCA	55	51	51	52	45	46	49	49
HQMS	146	149	147	149	130	132	137	135

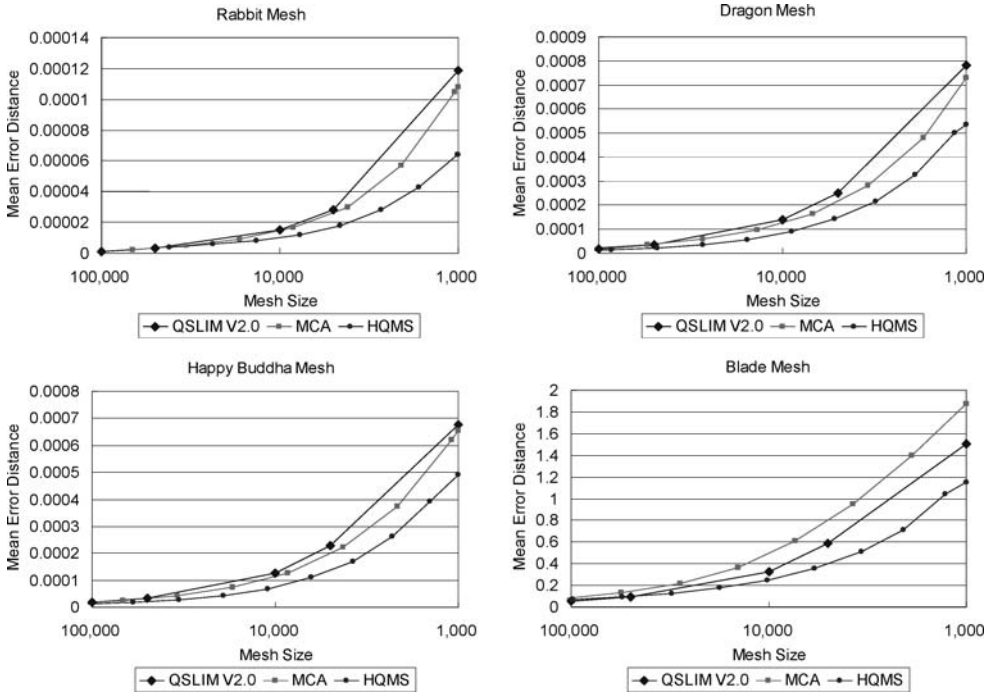

Fig. 8 The mean geometric errors of the LOD meshes of the Rabbit, Dragon, Happy Buddha, and Blade meshes generated by the QSlim V2.0 (default configuration), MCA ($d = 8$) and HQMS (heap size = 1,023)

Figure 8 shows the comparisons on geometric errors of the LOD meshes generated by QSlim V2.0, MCA, and HQMS methods, which includes the geometric errors of the LOD meshes of the Rabbit, Dragon, Happy, and Blade meshes measured by Metro. It is pretty obvious that our new method (HQMS) outperforms the other two methods by yielding lower error outputs.

On the other hand, the comparisons on visual quality via rendered images of the LOD meshes generated by the three methods are presented in Figs. 9 and 10 where all the illustrative rendered images are derived from smooth shading via the OpenGL V1.2. For the same reason as we have mentioned earlier, QSlim V2.0 are not able to generate LOD meshes of the David and Dawn meshes using only 1 GB main memory; hence, the images of these two meshes are generated by using 2 GB main memory in the same platform. In addition, a portion of the LOD meshes of the David, Dawn, Youthful, and Awakening meshes created by HQMS are demonstrated in Figs. 11–13.

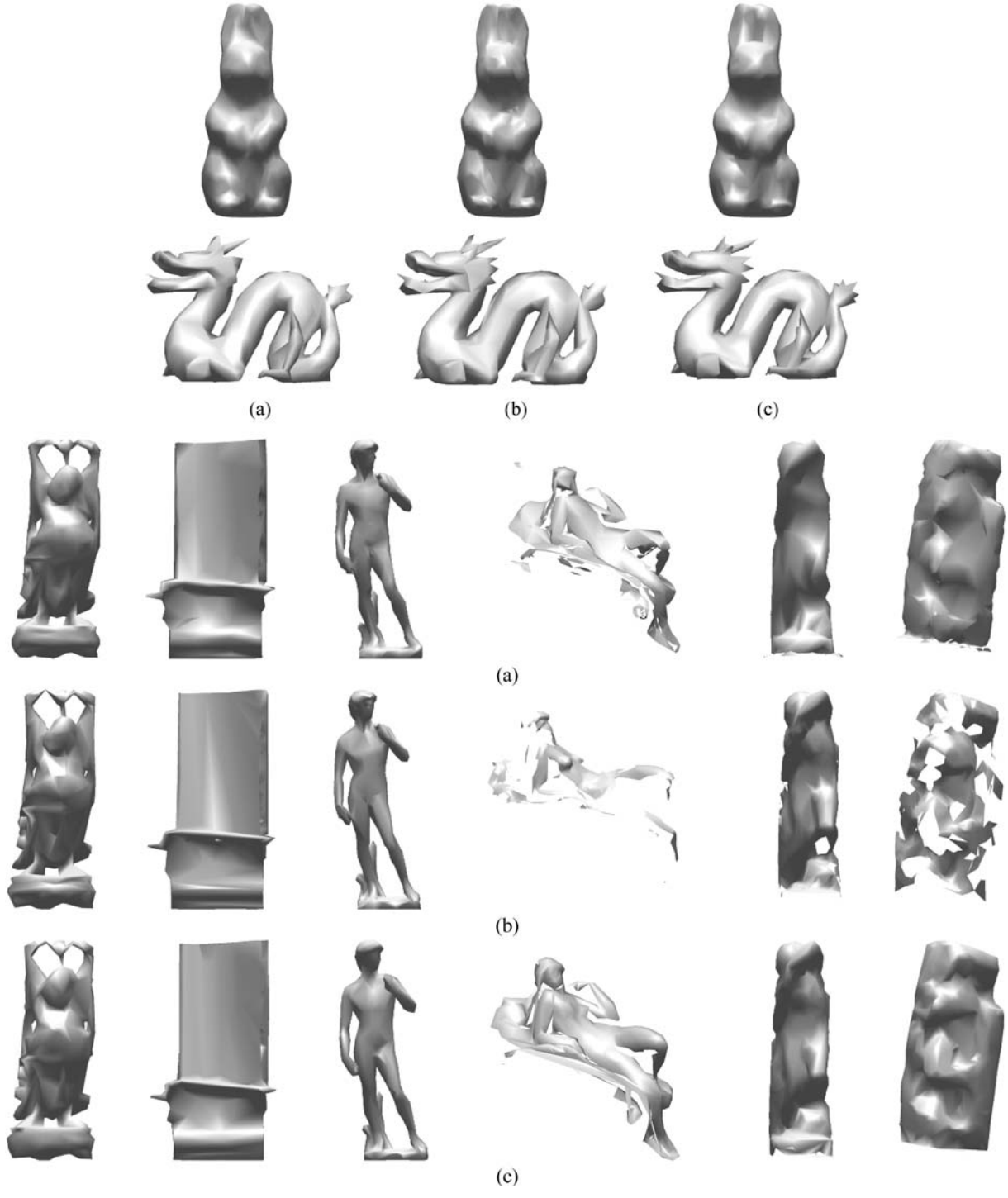

Fig. 9 The 1,000 triangles LOD meshes generated by the **a** QSlim, **b** MCA, **c** HQMS methods

Through examining the images shown in Fig. 9, one could easily found that our new method featured with two-stage optimization and enhanced hole-filling capability obviously outdoes the other two by generating far better quality outputs especially for meshes with numerous holes, e.g., the Dawn, Youthful, and Awakening meshes. Thus, the new hole-filling scheme is evidently successful.

Through the comparisons we have provided thus far, it is sufficient to conclude that the HQMS is a very efficient and effective approach to producing high-quality discrete LOD meshes for interactive multimedia applications such as 3D games and virtual reality (VR).

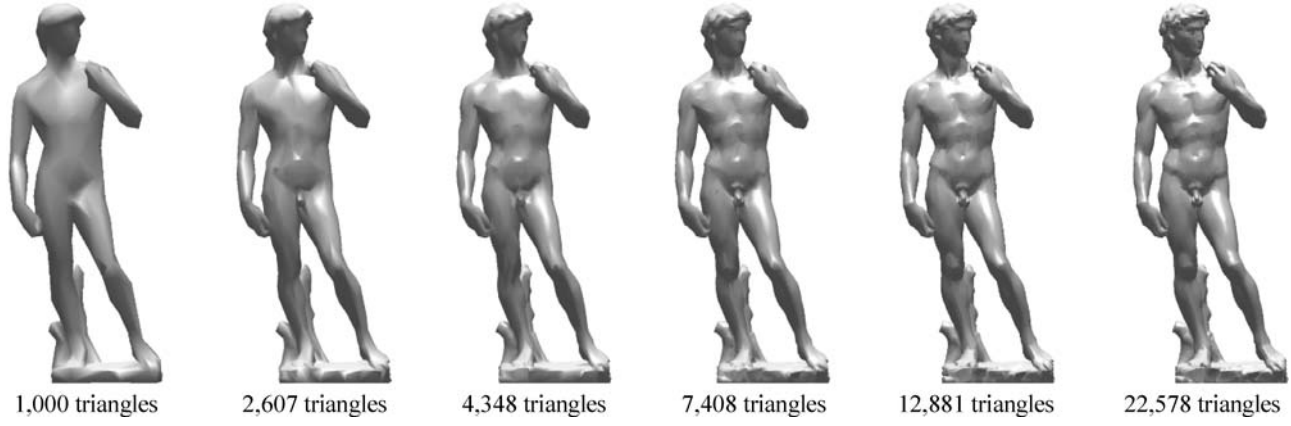

Fig. 10 The LOD meshes of the David (2 mm) mesh generated by the HQMS

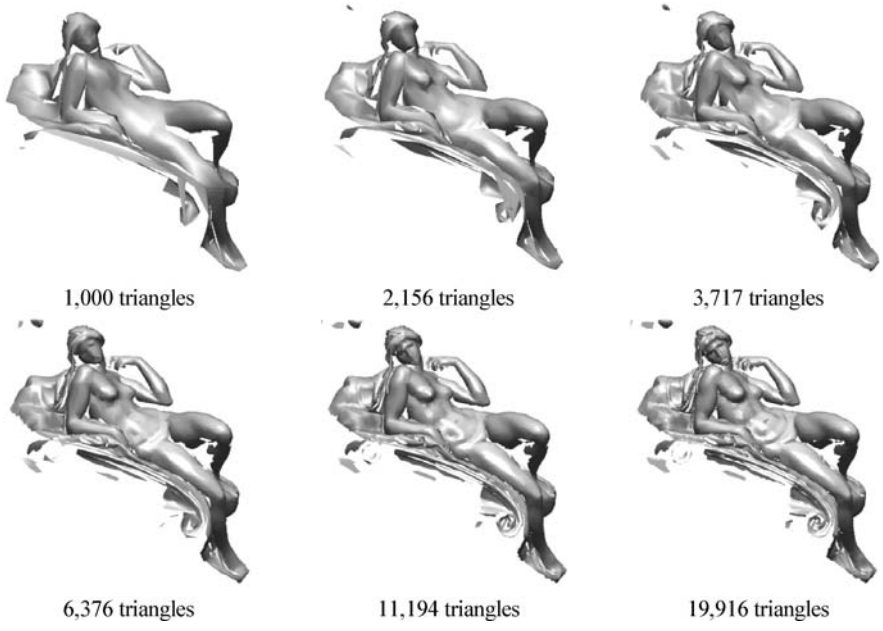

Fig. 11 The LOD meshes of the Dawn (1 mm) mesh generated by the HQMS

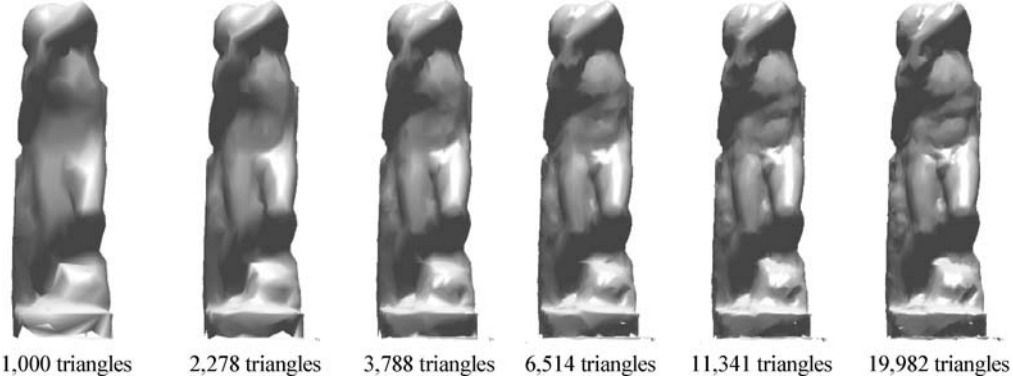

Fig. 12 The LOD meshes of the Youthful (2 mm) mesh generated by the HQMS

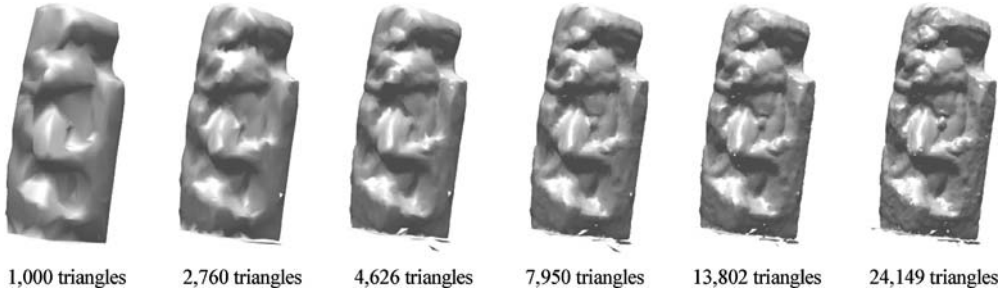

Fig. 13 The LOD meshes of the Awakening (2 mm) mesh generated by the HQMS

6 Conclusions

In this paper, we have addressed the issue of generating very high-quality discrete LOD meshes for interactive multimedia applications such as the 3D computer games and virtual reality and proposed a new runtime and memory efficient polygonal mesh simplification algorithm capable of generating very good quality discrete LOD meshes within linear running time.

The new algorithm integrates the vertex quadrics computation, costs evaluation, optimization, and vertex ring contractions into a single pipeline. This new design eliminates not only the storage cost of the vertex quadrics by using the memoryless quadrics computation suggested by [3] but also the cost of the contraction queue by replacing the large greedy queue with a small constant size RS-heap. In addition, we have suggest a two-stage optimization scheme to save computation overhead resulting from unnecessary calculations for optimum placements and a simple and effective approach to fill the undesirable holes of the meshes.

According to the empirical evidences we have provided in Sect. 5, the proposed algorithm is very successful. It not only outperforms the QSlim V2.0 that runs in $O(|V|\log|V|)$ but also the linear time MCA method in runtime efficiency. Besides this, the memory consumption of our algorithm is also significantly lower than those of the QSlim, MCA, and most other similar algorithms [4, 6, 12–19]. In practice, we have successfully simplified the Lucy mesh that comprises more than 28 million triangles using 1.2 GB main memory space in 328 s on a 2.2 GHz Intel Pentium IV PC, which implies this method also has the ability to deal with an extent of large meshes. Furthermore, according to the comparisons we have made in Figs. 8 and 9, the new algorithm (HQMS) is proved to have the capability of producing very high-quality LOD meshes that are even superior to those generated by the greedy based (QSLIM V2.0) and multiple choice based (MCA) methods. Concluding from previous statements, we believe that the new method is very suitable for discrete LOD mesh generations in most interactive 3D multimedia applications.

In spite of the advantages we have mentioned above, the outputs of the new algorithm share the same disadvantages with the discrete LOD, i.e., the large disk I/O latency and bandwidth for loading the LOD meshes and the visual disturbance caused by the hopping effect while switching LODs. To deal with such problem, we will improve the new algorithm by introducing the option of producing continuous and view-dependent LODs in addition to discrete LOD for progressive viewing or transmission and visualization applications, respectively. Moreover, we do not address the issue of color and texture preservation in this paper since the quadric error metrics itself can be extended to process such attributes naturally. One who is interested in the preservation of theses additional attributes may refer [16, 19–24]. Another interesting topic involves animation, a number of works on this issue have been proposed to animate/morph the LOD meshes [38, 39].

Acknowledgements We would like to thank the 3D Scanning Repository and the Digital Michelangelo Project Archive of 3D Models for providing us all the test models. We also give our gratitude to Michael Garland for making his QSlim V2.0 available, Paolo Cignoni et al. for providing the Metro V3.1, and the anonymous reviewers for their precious comments and suggestions.

References

1. Watt, A.: 3D Computer Graphics, 3rd ed. Addison-Wesley, Reading, MA (2000)
2. Luebke, D. et al.: Level-of-Detail for 3D Graphics. Morgan Kaufman, San Francisco, CA (2003)
3. Lindstrom, P., Turk, G.: Evaluation of memoryless simplification. IEEE Trans. Vis. Comput. Graph. **5**(2), 98–115 (1999)
4. Knuth, D.E.: The Art of Computer Programming, vol. 3. Addison-Wesley, Reading, MA (1973)
5. Lindstrom, P.: Out-of-core simplification of large polygonal models. In: Proceedings of the SIGGRAPH '00, vol. 34, pp. 259–270 (2000)
6. Wu, J., Kobbelt, L.: Fast mesh decimation by multiple-choice techniques. In: Proceedings of 7th International Fall Workshop on Vision, Modeling, and Visualization, pp. 241–248 (2002)
7. Wu, J., Kobbelt, L.: A stream algorithm for the decimation of massive meshes. In: Proceedings of the Graphics Interface '03, pp. 185–192 (2003)
8. Rossignac, J., Borrel, P.: Multi-resolution 3D approximation for rendering complex scenes. In: Proceedings of the Geometric Modeling in Computer Graphics '93, pp. 455–465 (1993)
9. Low, K.L., Tan, T.S.: Model simplification using vertex clustering. In: Proceedings of the Symposium on Interactive 3D Graphics '97, pp. 75–82 (1997)

10. Lindstrom, P., Silva, C.T.: A memory insensitive technique for large model simplification. In: Proceedings of the IEEE Visualization ’01, vol. 35, pp. 121–126 (2001)
11. Shaffer, E., Garland, M.: Efficient adaptive simplification of massive meshes. In: Proceedings of the IEEE Visualization ’01, pp. 127–134 (2001)
12. Schroeder, W.J., Zarge, J.A., Lorensen, W.E.: Decimation of triangle meshes. In: Proceedings of the SIGGRAPH ’92, vol. 26, pp. 65–70 (1992)
13. Hoppe, H. et al.: Mesh optimization. In: Proceedings of the SIGGRAPH ’93, vol. 27, pp. 19–26 (1993)
14. Hoppe, H.: Progressive meshes. In: Proceedings of the SIGGRAPH ’96, vol. 30, pp. 99–108 (1996)
15. Ronfard, R., Rossignac, J.: Full-range approximation of triangulated polyhedra. In: Proceedings of the Eurographics ’96, vol. 15, no. 3, pp. 67–76 (1996)
16. Garland, M., Heckbert, P.S.: Surface simplification using quadric error metrics. In: Proceedings of the SIGGRAPH ’96, vol. 30, pp. 209–216 (1997)
17. Kobbelt, L., Campagna, S., Seidel, H.P.: A general framework for mesh decimation. In: Proceedings of the Graphics Interface ’98, pp. 43–50 (1998)
18. Campagna, S., Kobbelt, L., Seidel, H.P.: Efficient decimation of complex meshes. Technical Report, Computer Graphics Group, University of Erlangen-Nürnberg, Germany (1998)
19. Hoppe, H.: New quadric metric for simplifying meshes with appearance attributes. In: Proceedings of IEEE Visualization ’99, pp. 59–66 (1999)
20. Cignoni, P., Montani, C., Rocchini, C., Scopigno, R.: A general method for preserving attribute values on simplified meshes. In: Proceedings of IEEE Visualization ’98, pp. 59–66 (1998)
21. Cohen, J., Manocha, D., Olano, M.: Simplifying polygonal models using successive mappings. In: Proceedings of IEEE Visualization ’97, pp. 395–402 (1997)
22. Cohen, J., Olano, M., Manocha, D., Appearance preserving simplification. In: Proceedings of SIGGRAPH ’98, pp. 115–122 (1998)
23. Sander, P., Snyder, J., Gortler, S., Hoppe, H.: Texture mapping progressive meshes. In: Proceedings of SIGGRAPH ’01, pp. 409–416 (2001)
24. Garland, M., Zhou, Y.: Quadric-based simplification any dimension. ACM Trans. Graph. **24**(2) (2005)
25. Clark, J.H.: Hierarchical geometric models for visible surface algorithms. Commun. ACM **19**(10), 547–554 (1976)
26. Funkhouser, T.A.: Database and display algorithms for interactive visualization of architectural models. Ph.D. Dissertation, The University of California at Berkeley (1993)
27. Giegl, M., Wimmer, M.: Unpopping: solving the image-space blend problem. available at www.cg.tuwien.ac.at/research/vr/ unpopping/unpopping.pdf, submitted to Journal of Graphics Tools, Special Issue on Hardware-Accelerated Rendering Techniques (2005)
28. Xia, J.C., El-Sana, J., Varshney, A.: Adaptive real-time level-of-detail-based rendering for polygonal models. IEEE Trans. Vis. Comput. Graph. **3**(2), 171–183 (1997)
29. El-Sana, J., Chiang, Y.J.: External memory view-dependent simplification. Comput. Graph. Forum **19**(3), 139–150 (2000)
30. Luebke, D., Erikson, C.: View-dependent simplification of arbitrary polygonal environments. In: Proceedings of the SIGGRAPH ’97, vol. 31, pp. 199–208 (1997)
31. Cignoni, P. et al.: External memory management and simplification of huge meshes. IEEE Trans. Vis. Comput. Graph. **9**(4), 525–537 (2003)
32. Shafer, E., Garland, M.: A multiresolution representation for massive meshes. IEEE Trans. Vis. Comput. Graph. **11**(2), 139–148 (2005)
33. Chen, H.K. et al.: A linear time algorithm for high-quality mesh simplification. In: Proceedings of the IEEE 6th International Symposium on Multimedia Software Engineering, pp. 169–176 (2004)
34. Chen, H.K. et al.: A novel cache-based approach to large mesh simplification. J. Inf. Sci. Eng. (to appear)
35. Levoy, M. et al.: The Digital Michelangelo Project: 3D scanning of large statues. In: Proceedings of the SIGGRAPH ’00, pp. 131–144 (2000)
36. Garland, M., Heckbert, P.S.: QSlim v.2.0 Simplification Software, Dept. of Computer Science, University of Illinois, http://graphics.cs.uiuc.edu/∼garland/software/QSlim.html (1999)
37. Cignoni, P., Rocchini, C., Scopigno, R.: Metro: measuring error on simplified surfaces. In: Proceedings of the Eurographics ’98, vol. 17, no. 2, pp. 167–174 (1998)
38. Houle, J., Poulin, P.: Simplification and real-time smooth transitions of articulated meshes. In: Proceedings of the Graphics Interface ’01, pp. 55–60 (2001)
39. Lin, C.H., Lee, T.Y.: Metamorphosis of 3D polyhedral models using progressive connectivity transformations. IEEE Trans. Vis. Comput. Graph. **11**(1), 2–12 (2005)