Vis Comput (2016) 32:967–976
DOI 10.1007/s00371-016-1242-z

ORIGINAL ARTICLE

Parallel mesh simplification using embedded tree collapsing

Hyunho Lee[1] · **Min-Ho Kyung**[1]

Published online: 6 May 2016

Abstract We present a novel parallel algorithm for mesh simplification that can reduce an input triangle mesh with highly improved performance. To take full advantage of the GPU comprising many computing cores, we enable collapsing of connected edges to be processed at one time by breaking data dependency in the update of the mesh data structure. Our solution is a lazy update method, which temporarily stores edge update information in a table and then updates the mesh data with it in the next step. Thanks to the lazy update method, we can more freely choose a large number of edges in the form of small trees for collapsing. The constructed trees are split to satisfy an error constraint, prevent normal flipping, and preserve the mesh topology. In experiments performed on several test models of various scales, we found that our algorithm consistently outperformed the prior GPU algorithm of Papageorgiou and Platis (Vis Comput 31(2):235–244, 2015) by a factor of 10 or higher.

Keywords Mesh simplification · Edge collapsing · Parallel algorithm · GPU

1 Introduction

Mesh simplification has been intensively studied since the 1990s, and robust implementations of several efficient algorithms are available in free and commercial 3D software. Their typical performance can potentially reduce a mesh with 1 million triangles to a target size in a few seconds; this is fast enough for most graphics applications, which usually perform mesh reduction in a preprocessing step.

However, demand for a faster mesh simplification algorithm still exists in engineering applications in which substantial amounts of 3D scene data must be processed for visualization purposes; one example is the manufacturing industry, which is rapidly adopting smart factory environments. A typical 3D factory model consists of many parts and devices modeled with several million triangles. In such massive factory models, configurations change frequently; thus, engineers require 3D systems that can rapidly transition between tasks.

Graphics processing units (GPUs), which act as off-the-shelf parallel computers as well as 3D rendering devices, are now widely used to accelerate numerous large-scale problems at a low cost on an ordinary PC. However, to utilize the full power of a GPU, we must divide a problem into a large number of small independent and identical tasks, so that all GPU computing cores are saturated for most of the computation time. Mesh simplification is solved by a sequence of identical decimation operations, not all of which are independent. Thus, we must rearrange the sequence so that a large group of independent operations can be solved at each parallel step. Papageorgiou and Platis [15] developed an edge-collapsing algorithm based on independent vertex sets. A set of collapsing edges is constructed from an independent set, which is usually a sparse set covering only 8–9 % of the input mesh. Thus, their algorithm required many iterations to reach a target size; as a result, it ran only three to four times faster than an equivalent serial algorithm.

In this paper, we present a novel parallel algorithm for mesh simplification that can simultaneously collapse a large number of edges not restricted by a minimum distance requirement. Our technical contributions are twofold: a lazy update method for eliminating the data dependency of adja-

✉ Min-Ho Kyung
kyung@ajou.ac.kr

[1] Department of Digital Media, Ajou University, Suwon, Korea

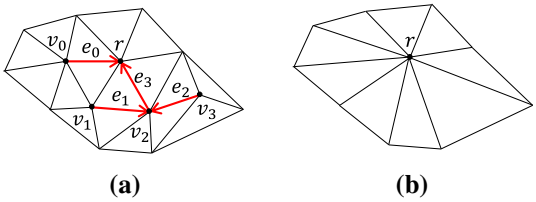


Fig. 1 Tree collapsing example. Edges e_0, e_1, e_2, and e_3 form a tree rooted at r. Collapsing all these edges, vertices v_0, v_1, v_2, v_3 are merged to r, while *eight triangles* are removed. **a** Collapsing tree. **b** After collapsing

cent edges being collapsed, and a new decimation scheme called *tree collapsing*. To break the data dependency among adjacent edges, we separate mesh updating from collapsing operations by storing necessary information for updating in a temporary table. Thanks to the lazy mesh updating, we have no restriction on distances between collapsing edges, allowing even connected edges. The only required condition for the edges is that they must be acyclic, because an articulation point could be made when a cycle of edges is collapsed. The acyclic condition naturally leads us to construct a forest of tiny trees for collapsing, which will be reduced to their root vertices as illustrated in Fig. 1. Although tree collapsing looks similar to clustering-based vertex decimation, it does not necessitate a complex triangulation step to fill a loop of edges surrounding a decimated cluster; moreover, it enables us to elaborately control the quality of reduced meshes, as will be explained later.

The paper is organized as follows. We summarize related work in Sect. 2 and discuss the data structure used in our algorithm in Sect. 3. Section 4 describes the details of the presented algorithm, and Sect. 5 discusses the test results for the four examples of different sizes. Finally, we conclude the paper with a summary and discussion of future work in Sect. 6.

2 Related work

Numerous researchers have investigated various methods to efficiently simplify 3D meshes since the early 1990s. Prior algorithms are classified into two general categories, according to basic operations used to reduce mesh complexity: vertex decimation and edge collapsing. Vertex decimation algorithms repeatedly remove vertices chosen on the basis of decimation criteria and incident triangles, and fill the holes by re-triangulating the loop of the bounding vertices [19]. To improve the mesh reduction rate, Rossignac and Borrel [16] proposed an algorithm that decimates a cluster of vertices based on geometric proximity. Lindstrom [13] simplified large polygonal model in an out-of-core manner using Rossignac and Borrel's vertex clustering. Schaefer and Warren [18] used octrees for adaptive vertex clustering on massive out-of-core meshes. Edge collapsing algorithms iteratively contract edges to a single vertex, eliminating the need for re-triangulation. Guéziec [8], Hoppe [10], and many other researchers proposed simplification algorithms based on edge-collapsing operations with various edge selection strategies. Garland and Heckbert [6] created a simple quadric error metric effectively measuring surface distortion incurred by a vertex change and proposed an efficient simplification algorithm based on the metric. For additional surveys of early prior works, we recommend the article written by Luebke [14]. Recently, Salina et. al. [17] proposed a structure-aware mesh decimation method preserving global structures in extreme simplification of massive surface meshes. Some researchers [1,21] attempted to simplify large mesh data in a distributed environment by partitioning them into sub-meshes. Their approaches are not appropriate for the GPU acceleration requiring fine-grained parallelism.

Early attempts to accelerate mesh simplification with a GPU formatted the input and output of mesh data properly for the standard graphics pipeline, and performed simplification on shader units running small shader programs. DeCoro and Tatarchuk [4] clustered mesh vertices by subdividing them in a 3D grid or an octree for decimation on the GPU. Although their algorithm was impressively fast, the running time increased cubically according to the grid or octree resolution; this determined the granularity of the reduced meshes. Hjelmervik and Léon [9] encoded vertex position and edge connection data in OpenGL textures and accelerated half-edge collapses on the GPU.

Shontz and Nistor [20] proposed a hybrid method that split the mesh simplification workload into a CPU portion and a GPU portion. They used a soft-grained blocking method based on test-and-set to lock triangles affected by a collapsing edge. Papageorgiou and Platis [15] and Cellier et. al. [2] constructed a set of super-independent vertices to define independent areas, in each of which an edge can be collapsed without interfering with other areas. These methods showed only moderately higher performance than prior serial algorithms, because a super-independent vertex set, constructed coarsely, produces only a small number of collapsing edges; thus, the reduction rate per execution is hard to improve. Grund et. al. [7] presented a fast parallel mesh simplification algorithm using an array similar to our edge updating table. However, their mesh data structure seems to be hard to handle topology change and triangle normal flipping.

3 Data structure

We take an input mesh **M** in a vertex position array **V** and a triangle vertex index array **F** and construct a standard half-edge data structure. The connection data for a half-edge e is stored in four fields: *next*, *twin*, *inface*, and *head*. *next*

contains the next half-edge ID in counterclockwise order on the incident triangle, *twin* contains the mated half-edge ID in the opposite direction, *inface* contains the incident triangle ID, and *head* contains the start vertex ID. Half-edge data are stored in a 4D integer array indexed by edge IDs, denoted by **H**. Array **H**, allocated with size $3n$ for $n = |\mathbf{F}|$, is constructed on the GPU for the input arrays **V** and **F**. To set the *twin* fields, we use a radix sort library implemented on the GPU.

4 Parallel half-edge collapsing

In this section, we will discuss how to process edge-collapsing operations in parallel without incurring dependency problems. Because we must manage simultaneous inter-dependent collapsing operations, we cannot adjust a vertex position optimally for each operation; however, we can simply move a tail to a head vertex for each collapsed half-edge. This is often called *half-edge collapsing* to distinguish it from ordinary edge collapsing [12]. To minimize the quality loss of half-edge collapsing, we will add a step to refine vertex positions after completion of all operations. Let $\mathbf{E}_{\text{col}}$ be a set of half-edges selected for collapsing, and let $\mathbf{V}_{col}$ be an array indexed by a vertex ID whose entries have a collapsed half-edge leaving from a corresponding vertex.

To manifest the problem of inter-dependent collapsing operations, suppose that two connected half-edges are collapsed (e_0 and e_7 in Fig. 2a). If e_0 is first collapsed, the twin of $e_9(e_8)$ is changed from $e_1(e_2)$ to $e_8(e_9)$ as $e_1(e_2)$ is merged to $e_8(e_9)$, as shown in Fig.2b. Collapsing e_7 then changes the twin of $e_9(e_{12})$ from $e_8(e_6)$ to $e_{12}(e_9)$ (Fig. 2c). The other collapsing order also gives the same result. However, if these operations asynchronously update the mesh structure simultaneously, the twins of e_9 and e_{12} incorrectly point to the discarded e_8 and e_2, because each parallel task is performed without knowing the result of the other.

Atomic processing of collapsing operations would be a possible solution, but is difficult to implement efficiently on a GPU. Shontz et al. [20] attempted a test-and-set method to lock edges, which required many iterative launches of a GPU kernel to finish one task. A locking mechanism could be internally implemented by looping an atomic operation supported by the GPU. It is also wasteful, because computing cores held by waiting tasks are not allocated for other active tasks, but idling until all adjacent edges are unlocked.

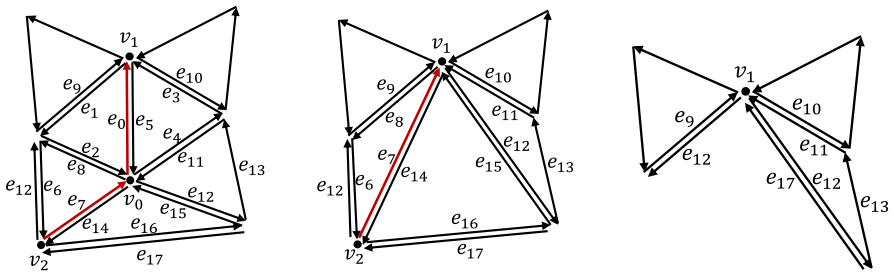
Instead of inefficient edge locking, we propose a simple and efficient solution: a lazy update method in which edge collapsing is decomposed into two sub-steps: (1) recording merged edges in a table and (2) updating the mesh structure using information from the table. Collapsing a half-edge $e \in \mathbf{E}_{\text{col}}$ drags its tail vertex t to the head vertex h so that the adjacent edges of t are discarded by merging to the adjacent edges of h. Here, the required update in the mesh data is simply to reconnect the remaining half-edges to new twins, which are actually those merged from the old discarded twins. Thus, for collapsed edges, we first record the merging destinations of discarded half-edges in a table (denoted by $\mathbf{M}_{\text{eg}}$) and then update the mesh data simply by looking it up.

$\mathbf{M}_{\text{eg}}$ is initially created with the entries set to their indices to represent unchanged half-edges. Each GPU thread for $e \in \mathbf{E}_{\text{col}}$ updates the entries corresponding to the merged half-edges as

$$\mathbf{M}_{\text{eg}}[e_1] \leftarrow \mathbf{H}[e_2].twin, \quad \mathbf{M}_{\text{eg}}[e_2] \leftarrow \mathbf{H}[e_1].twin,$$
$$\mathbf{M}_{\text{eg}}[e_3] \leftarrow \mathbf{H}[e_4].twin, \quad \mathbf{M}_{\text{eg}}[e_4] \leftarrow \mathbf{H}[e_3].twin,$$

where e_1, e_2, e_3, and e_4 are the adjacent half-edges of e:

$$e_1 \leftarrow \mathbf{H}[e].next, \quad e_3 \leftarrow \mathbf{H}[\mathbf{H}[\mathbf{H}[e].twin].next],$$
$$e_2 \leftarrow \mathbf{H}[\mathbf{H}[e].next], \quad e_4 \leftarrow \mathbf{H}[\mathbf{H}[e].twin].next.$$



(a) **(b)** **(c)**

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
$\mathbf{M}_{eg}$	-	8	9	11	10	-	2	-	12	9	10	11	12	13	-	17	12	17

(d)

Fig. 2 Before and after collapsing half-edges e_0 and e_7, and the $\mathbf{M}_{\text{eg}}$ table for the collapsing process. $e_0, \ldots, e_5$ and the triangles incident to them are deleted. Deletion of e_1 and e_2 transforms e_8 and e_9 into twins, and deletion of e_3 and e_4 transforms e_{10} and e_{11} into twins. **a** Before collapsing. **b** After collapsing e_0. **c** Ater collapsing e_7. **d** Table $\mathbf{M}_{\text{eg}}$ constructed for collapsing e_0 and e_7

Figure 2d is a table constructed for the case shown in Fig. 2a. The entries with dashes represent the collapsed half-edges and their twins; these are ignored in the subsequent mesh update step.

There may be an entry asynchronously updated more than once, which occurs only when the half-edge corresponding to the entry is in a triangle with the other two half-edges collapsed. A race condition here is not critical, because the half-edge will be eventually deleted as its triangle is contracted to a single vertex.

Next, the *twin* field of half-edges is updated to realize mesh reduction. As mentioned above, if a twin is discarded by merging, its merging destination becomes a new twin replacing the old one. A merging destination is retrieved from $\mathbf{M}_{\text{eg}}$ by a single lookup, or multiple lookups if the half-edge from the first lookup is also a discarded one. We also need to update the *head* field, because a tail vertex of a collapsed half-edge is discarded after being dragged to a head vertex. For each half-edge, we find a new vertex for the *head* field by following the path defined by a connected sequence of collapsed half-edges starting from its old head vertex. We use the array $\mathbf{V}_{col}$ to follow the path and update the *head* field with the last vertex of the path. Both updating tasks are performed on the GPU in parallel, simply by assigning a GPU thread to process each half-edge update. They do not suffer from memory access conflicts, because reading and writing are performed on different arrays ($\mathbf{M}_{\text{eg}}$ and $\mathbf{V}_{col}$ for reading, and $\mathbf{H}$ for writing).

To reduce the frequency of multiple lookups of $\mathbf{M}_{\text{eg}}$, we update all the visited entries for each query to the final lookup result, similar to the path compression method of union-find algorithms. Merged vertices and half-edges are marked *deleted* and cleaned later after a target size is reached.

5 Tree-based collapsing algorithm

Although a constraint on edge dependency in $\mathbf{E}_{\text{col}}$ was resolved in the previous section, there is another constraint on half-edges in $\mathbf{E}_{\text{col}}$, that is, an acyclic constraint. If some collapsed half-edges make a cycle, the vertex to which the cycle is contracted will not be clearly determined in half-edge collapsing, and more seriously the contraction vertex will become an articulation or a non-manifold point. Thus, the set $\mathbf{E}_{\text{col}}$ should be acyclic or form a set of trees. In this section, we will discuss how to construct $\mathbf{E}_{\text{col}}$ as a set of trees embedded on the input mesh, controlled by a user-specified error threshold τ. τ acts as a quality control parameter by bounding the accumulated quadric error at a contraction vertex.

The procedure of the proposed simplification algorithm is summarized as:

Repeat 1–6 until a target size n'_t is reached.

1. Compute a quadric error matrix at every vertex.
2. Construct a set of trees with collapsible edges.
3. Split the trees to meet τ.
4. Cancel tree edges causing illegal mesh changes.
5. Collapse trees to their root vertices.
6. Refine root vertex positions.

Step 5 is actually a parallel collapsing step with $\mathbf{E}_{\text{col}}$ containing the tree edges. This was explained in the previous section. The details of the other steps will be discussed in the following sections.

5.1 Computation of quadric error matrix

The quadric error metric was first proposed by Garland and Heckbert [6] to measure the surface distortion caused by collapsing of an edge. Since then, it has been used as a common error metric in many mesh simplification algorithms, owing to its additive nature and effective measuring of visual quality loss. The quadric error that occurs when moving a vertex v to a position p in homogeneous coordinates is defined as $c_v(p) = p^T Q_v p$, where Q_v is a 4×4 symmetric matrix. Q_v is a coefficient matrix of a quadric equation that computes the sum of squared distances from the planes containing v. Because Q_v is symmetric and the last entry is always 1, only nine values must be stored.

For every vertex $v \in \mathbf{V}$, Q_v is precomputed and stored in array $\mathbf{Q}$ for frequent use in the following steps. Construction of $\mathbf{Q}$ on the GPU is straightforward. All the entries in $\mathbf{Q}$ are initially set to a zero vector. We launch a thread per triangle to build a plane equation containing a triangle and compute the nine entries of the quadric matrix with its coefficients. Then the thread accumulates the nine values to $\mathbf{Q}[v]$ for each triangle vertex v.

5.2 Construction of embedded trees

Tree edges are independently selected over vertices in parallel. From among the half-edges leaving from each vertex v, a tree edge $e = \overrightarrow{vh}$ must be selected. To ensure that e does not make a cycle with other tree edges, we select the half-edge with the smallest weight $w = \frac{1}{2}(c_v(h) + c_h(v))$, an average of two twins' collapsing errors. Half-edges selected in this manner have decreasing w on any paths formed by them, because an incoming half-edge to v in a path always has a larger w than the next half-edge leaving from v. Thus, they cannot make any cycles. Two twin edges may be exceptionally selected together by two adjacent vertices, when their weights (which are equal) are smallest at both end vertices. We add a verification step to identify them after tree con-

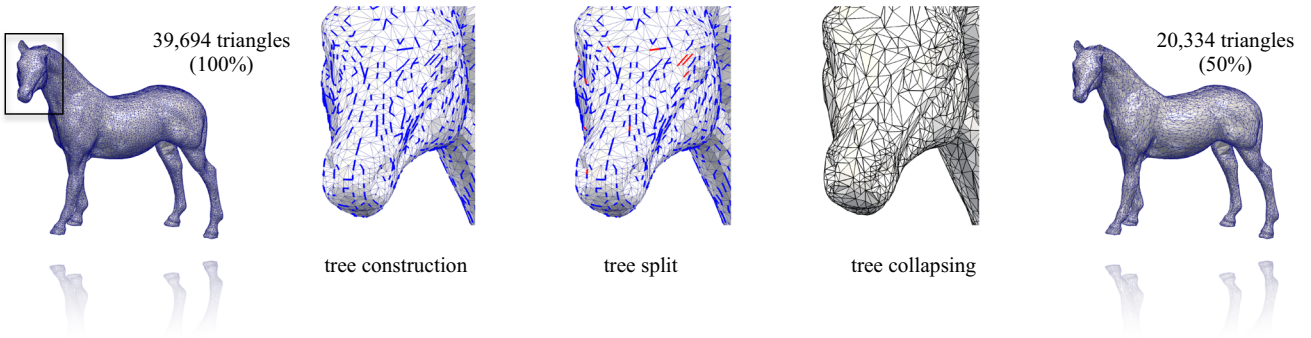


Fig. 3 Algorithm overview. The *horse head* (the *black box* in the *leftmost* figure) is magnified in the three middle figures to illustrate the steps of collapsing tree construction (*blue*), *deleted edges* (*red*) for splitting, and tree collapsing, respectively. The *rightmost* figure is the output after reduction

struction and take only the twin with the smaller collapsing error when they are found. Finally, we test if $c_v(h) \leq \tau$ for e and discard it if not.

In GPU implementation, the output of this step is stored in array $\mathbf{V}_{col}$, initialized with *null*. A thread assigned to each vertex v executes a kernel that scans outgoing half-edges around v to find the one with the smallest weight; it writes it to $\mathbf{V}_{col}[v]$ if its collapsing error is less than τ. Afterward, another kernel is launched to handle exceptions.

In Fig. 3, the second image shows a set of trees (red lines) constructed on the horse model. The constructed set is nearly maximal, as most of the vertices with any collapsible half-edges are included in the trees. However, it is not yet ready for collapsing, because of erroneous changes on the output mesh such as prohibitively large collapsing errors, normal flipping of triangles, and illegal topology changes. Because these occur when a mesh region is excessively contracted along a relatively large tree, they can be prevented by splitting such large trees before collapsing. We will discuss these issues in the following three sections.

5.3 Splitting trees

We want to keep all collapsing errors accumulated at vertices under τ, to control the quality of the reduced mesh. However, in the tree construction, only individual edge-collapsing errors were considered; thus, accumulated errors along a tree can be larger than τ. Here, we present a parallel algorithm to test and split trees in a bottom-up manner, so that no trees with an accumulated error larger than τ remain.

When a sub-tree T rooted at v is collapsed, all the tree vertices are contracted to v. Thus, the tree collapsing error $c(v)$ is simply computed with the accumulated quadric matrix Q'_v as

$$c(v) = \sum_{p \in T} c_p(v) = v^T \left(Q_v + \sum_u Q'_u \right) v = v^T Q'_v v,$$

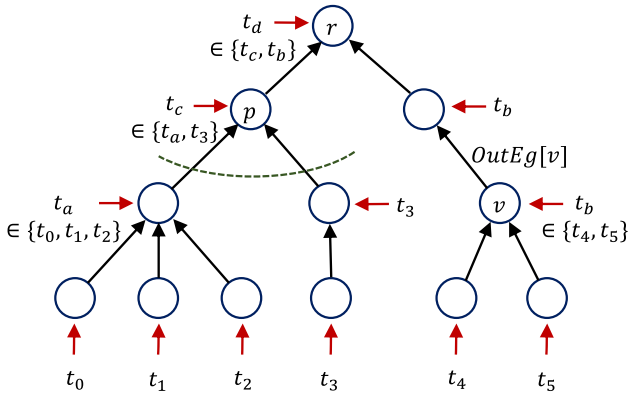


Fig. 4 A tree rooted at r is split at vertex p. $t_0, \ldots, t_5$ are GPU threads traversing the tree from the leaves, and some of them survive at each level to take care of parent vertices. The *green dashed line* shows that the subtrees under p are split by thread t_c

where us are children of v. For bottom-up computation of $c(v)$, we store Q'_v computed in the lower levels in an array $\mathbf{Q}'$ initially set with matrices in $\mathbf{Q}$. That is, as in the above equation, the matrix $\mathbf{Q}'[u]$ for a leaf vertex u is accumulated to its parent matrix $\mathbf{Q}'[v]$; further, when $\mathbf{Q}'[v]$ is complete, it is used to compute $c(v)$ and then is accumulated to its parent matrix, and so on.

When the test and split task is performed on the GPU, a thread is created for each leaf vertex; the task follows along the tree edges up to the root while repeating matrix accumulation and error testing. Because a tree has smaller vertices at higher levels, we do not need to keep all threads used in a level for the next iteration. Thus, we let the last thread that updated the parent matrix keep working on the parent vertex, while the other threads terminate as illustrated in Fig. 4. For example, thread t_a is selected among t_0, t_1, and t_2, which are threads assigned to the child vertices. For a thread to determine whether it survives or not, we use a *lot-drawing* method. A thread performs an atomic decrement-and-read operation on a storage initially set to the in-degree of a par-

ent p, immediately after updating $\mathbf{Q}'[p]$. If the read value of the atomic operation is non-zero, the thread decides to terminate. Because the in-degree value is equal to the number of children, only the last thread will read 0 and survive. We create an array D_{in} for storage of in-degree values of all the vertices before starting the split task.

The surviving thread, which sets $v \leftarrow p$, compares $c(v)$ with τ. If it is less than τ, the thread continues accumulating $\mathbf{Q}'[v]$ to the parent matrix. Otherwise, the sub-tree rooted at v must be split at v's children, as illustrated by the dashed line in Fig. 4. To split the child sub-trees, the thread sets all the entries of $\mathbf{V}_{col}$ indexed by v's children to *null*, and resets $Q'[v]$ to $Q[v]$ as v becomes a leaf vertex. The test and split task continues until a vertex p with $\mathbf{V}_{col}[p] = null$ is reached.

5.4 Prevention of triangle flipping

Although all tree collapsing errors are kept under τ, there may be a triangle whose normal is reversed, owing to an excessive position change of a triangle vertex caused by tree collapsing. Repairing such triangles after simplification would require a constrained nonlinear problem to be solved, because repairing one triangle could affect another triangle to be flipped. Thus, we find such triangles beforehand and further discard tree edges to prevent triangle vertices from moving too far.

For convenience sake, we define $\gamma(v)$, a function that maps v to the final vertex to which it merged after collapsing, that is, the actual root of the tree v belongs to. Although it could be evaluated each time by following connected half-edges retrieved from $\mathbf{V}_{col}$, we instead create an array $\mathbf{R}$ to store the function value for every vertex. We use a path compression scheme for efficient construction of $\mathbf{R}$.

For each triangle $f = v_0v_1v_2$, we determine if the normal of f will be reversed after collapsing, by testing the sign of the dot product of its original and new normal vectors after simplification. The new normal vector is computed with $\gamma(v_0)$, $\gamma(v_1)$, and $\gamma(v_2)$. If at least two of them are the same vertex, f will be removed in the simplified mesh and is thus ignored. If a flipped triangle is found, we follow the triangle vertices step by step from their initial vertices along the collapsing paths of v_0, v_1, and v_2 to find the first moment of triangle flipping. The triangle f in Fig. 5 changes to $v_0v_3v_5$, which has a consistent normal, and then to $v_0v_4v_5$, which has an opposite normal. Then, we cancel the tree edge v_3v_4 by setting $\mathbf{V}_{col}[v_3]$ to *null*.

5.5 Preserving mesh topology

Edge collapsing may change the mesh topology; examples include two adjacent triangles making a fin shape by folding, a ring on a mesh contracting to a point, and so on. Edges causing topological changes after collapsing can be identified

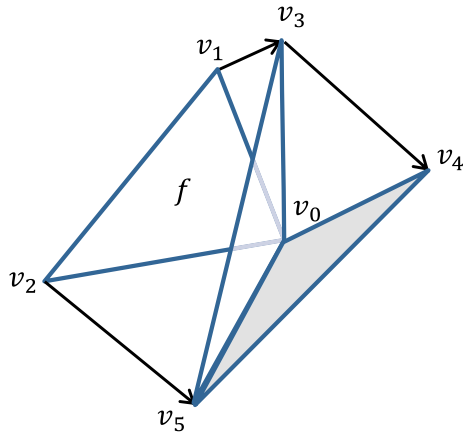


Fig. 5 The normal of a *triangle* f is flipped by vertex merging $v_1 \rightarrow v_3 \rightarrow v_4$ and $v_2 \rightarrow v_5$

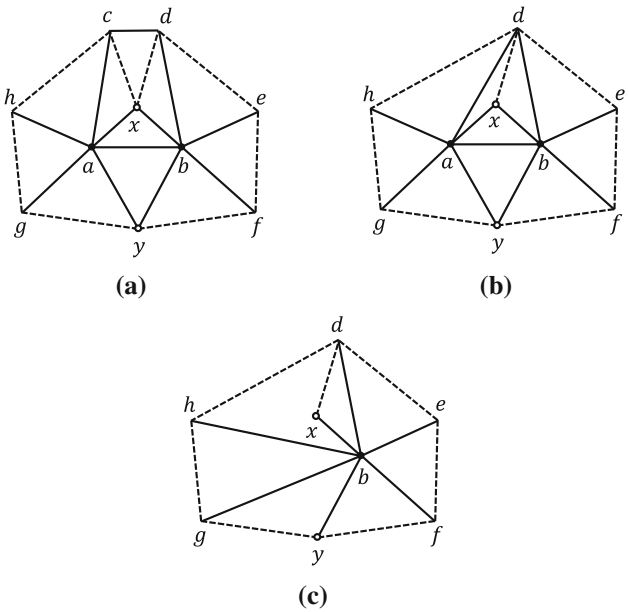


Fig. 6 Topological change by simultaneous collapsing of two edges: **a** Half-edges $\overrightarrow{ab}$ and $\overrightarrow{cd}$ satisfy the link condition, **b** $\overrightarrow{ab}$ not satisfying the link condition after collapsing $\overrightarrow{cd}$, and **c** a *triangle* fin Δxbd dangling from $\overline{bd}$ made by collapsing $\overrightarrow{ab}$

by testing the *link condition*: if $\mathrm{Lk}\, a \cap \mathrm{Lk}\, b = \mathrm{Lk}\, \overline{ab}$, the edge $e = \overline{ab}$ is safely collapsed without any topological change. This was proven by Dey et al. [5]. Here, $\mathrm{Lk}\, i$ represents a link of geometric entity i. If i is a vertex, $\mathrm{Lk}\, i$ is the boundary of an area formed by the incident triangles of i. $\mathrm{Lk}\, a$ in Fig. 6a is a region bounded by the vertices $c, h, g, y, b,$ and x. If i is an edge, $\mathrm{Lk}\, i$ is a set of two vertices opposite to i in its incident triangles. For example, $\mathrm{Lk}\, \overline{ab}$ in the figure is $\{x, y\}$.

However, the link condition is not directly applicable for parallel edge collapsing, because a test on one edge could be invalidated by another edge close or adjacent to it. In Fig. 6a, we have two tree edges $\overrightarrow{ab}$ and $\overrightarrow{cd}$, both satisfying the link

condition in the input mesh. However, collapsing both edges results in a topology-changed mesh, as in Fig. 6c, because collapsing $\overrightarrow{cd}$ prevents the other tree edge from satisfying the condition any longer (Fig. 6b). Thus, to ensure topology preservation in parallel edge collapsing, we reformulate the link condition on corresponding output edges and vertices in the simplified mesh.

In a 2-manifold triangle mesh, if the link condition is not satisfied, there must be a common vertex v shared by the two vertex links and not contained in the edge link. If v, possibly reduced to the other vertex, is still found separately from the contraction vertex of e in the simplified mesh, the edges connecting them are made into a non-manifold edge. This is because the edge has four incident triangles. Our new condition is developed based on this observation. Let L_a be a link boundary in a circular list containing link vertices of Lk a in counterclockwise order, and define a reduced link boundary

$$L'_a = \text{Compact}\ (\gamma(v_0)\gamma((v_1)\ldots\gamma(v_{m-1})),$$

for $L_a = v_0 v_1 \cdots v_{m-1}$, where *compact* is an operation that removes repeated elements in a given list. In Fig. 6a, we obtain $L'_a = y\underline{b}xdhg$ for $L_a = ybxchg$ and $L'_b = dx\underline{b}yfe$ for $L_b = dxayfe$. Here, $\underline{b}$ is a vertex to which edge $\overline{ab}$ contracts. Then, the new condition is defined as:

Post-link condition For a half-edge $\overrightarrow{ab}$, L'_a and L'_b have no common vertices except those next to $\underline{b}$ in L'_a and L'_b.

If $\overrightarrow{ab}$ meets the post-link condition, collapsing $\overrightarrow{ab}$ does not produce a non-manifold edge or vertex. If there is a common vertex $v = \gamma(c)$ that is not adjacent to $\underline{b}$, it implies that two input edges ($\overline{ca}$ and $\overline{cb}$) have both been transformed to the same edge $\overline{v\underline{b}}$, hence non-manifold. In Fig 6c, d, a common vertex in L'_a and L'_b, is incident to a non-manifold edge $\overline{db}$, that is, a merging of $\overline{ac}$ and $\overline{bd}$. A more rigorous proof for a single collapsing case can be found in [11].

The post-link condition is tested on all the tree edges to filter out topologically unsafe ones before collapsing. Each tree edge is assigned to a GPU thread, which constructs L'_a and L'_b by looking up $\mathbf{R}$, which was constructed in the previous step. Because L_a and L_b have various sizes, we first count the vertices by tracing the link boundaries and then dynamically allocate a local GPU array fit to $|L_a| + |L_b|$. In addition, we store in the array the vertex IDs of L'_a and L'_b obtained in the second tracing, skipping repeated vertices. Further, every pair of vertices in the two lists are compared to find common vertices. For a tree edge $\overrightarrow{ab}$, if a common vertex not adjacent to $\underline{b}$ is found, $\mathbf{V}_{col}[a]$ is set to *null*.

After all topologically unsafe edges have been canceled, we construct $\mathbf{E}_{\text{col}}$ by applying a parallel scan algorithm on $\mathbf{V}_{col}$. If $|\mathbf{E}_{\text{col}}| > 2d$, where d is the target number of deleted triangles, we remove from $\mathbf{E}_{\text{col}}$ the edges having the first d largest errors by parallel radix sorting to meet the target mesh size. Subsequently, we carry out the parallel edge collapsing on edges in $\mathbf{E}_{\text{col}}$, as explained in Sect. 4.

5.6 Vertex refinement

To improve mesh quality, we refine the positions of root vertices after the collapsing step by minimizing their accumulated quadric errors. As Garland and Heckbert [6] suggested, we solve a linear system for $\overline{v}$

$$Q^3 \cdot \overline{v} = -l^3, \tag{1}$$

where Q^3 and l^3 are the upper left 3×3 sub-matrix and the 3D vector of the fourth column of Q_v, respectively. Because Q^3 is a symmetric positive definite matrix, we use the conjugate-gradient method to solve Eq. 1. If $|v - \overline{v}| > \delta$ for a given δ, solution $\overline{v}$ is suspected to be overshot, owing to numerical errors; thus, it is discarded.

6 Results

We tested our simplification algorithm on four examples having different levels of complexity. The test was performed on a PC with a 3.5GHz Intel i7 CPU and an Nvidia GTX Titan with 6GB onboard memory. We implemented the algorithm on the CUDA platform. The result statistics of the four examples are summarized in Table 1. We simplified the four models three times for 50, 10, and 1 % target sizes. To reduce an input model down to each target size, we manually select an optimal τ to minimize quality loss. Because each input model has a different optimal τ, we needed to identify it for each case by conducting several trials. For real-time applications, we may tolerate a moderate loss of quality; thus, we may set τ to a fairly large number such as 0.1 (as in the horse example). If mesh reduction is performed off-line, trying several different values of τ with our algorithm would not be difficult. A trend in the results shows that the optimal τ decreases as the input size increases, presumably affected by the average triangle size as well as other shape characteristics. Thus, we may be able to estimate an optimal value of τ from sample data measured for different input and target sizes.

We tested the simplification process with $\tau = \infty$, to obtain an empirical upper bound of reduction rates regardless of mesh quality. The results summarized in Table 2 show almost identical reduction rates, in which the average is 70.5 %. This is far better than the results reported in [15], which showed an 8–9 % reduction rate at most. The higher reduction rate of our algorithm requires fewer iterations, which reduces the overhead of GPU kernel launches. Note that the true upper bound is not actually 70 %, but can be even higher. For exam-

Table 1 Summary of the results

	Triangles	Vertices	50 %			10 %			1 %		
			τ	Iter.	Time	τ	Iter.	time	τ	Iter.	Time
Horse	39,694	19,847	10^{-5}	1	4	0.001	2	7	0.1	4	11
Gargoyle	1,035,698	517,852	10^{-6}	1	28	10^{-4}	2	45	0.01	4	65
Asian dragon	7,218,906	3,609,455	10^{-8}	1	155	10^{-6}	2	250	10^{-4}	4	330
Thai statue	10,000,000	4,999,996	10^{-8}	1	250	10^{-6}	2	380	10^{-4}	4	495

The table columns from left to right show the triangle and vertex counts of input meshes, τ, the number of iterations, and running times for 50, 10, and 1 % target sizes, respectively. Times are represented in ms

Table 2 Simplification results with $\tau = \infty$

Model	Simplified	Reduction rate (%)
Horse	11,770	70.3
Gargoyle	310,350	70.0
Asian dragon	2,029,762	71.8
Thai statue	2,971,836	70.2

The four models are similarly reduced by approximately 70 % with the infinite τ, which implies that 70 % is the practical upper bound of simplification we can achieve in one execution of our algorithm

Table 3 Comparison with the algorithm of Papageorgiou

Model	Papageorgiou		Ours		Speedup
	Iter.	Time	Iter.	Time	
Horse	40	66	2	7	9.4
Gargoyle	35	605	2	45	13.4
Asian dragon	35	3690	2	250	14.8
Thai statue	35	5290	2	380	13.9

The running times to reduce the four models by 90 % are measured in ms with both algorithms, summarized in the third and fifth columns. Our algorithm completes the task approximately 13 times faster

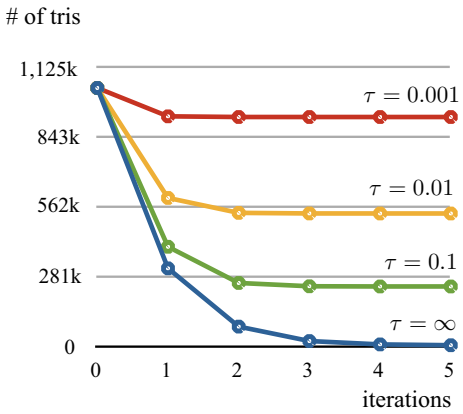


Fig. 7 Convergence of mesh reduction. The graph shows the triangle count of the gargoyle model simplified with different values of τ. After 2–3 iterations, the count reaches a convergence point regardless of τ

Table 4 Comparison of root mean-square(RMS) errors and Hausdorff distances (HD) for the gargoyle model between MeshLab and our algorithm

Target size (%)	MeshLab ($\times 10^{-5}$)		Our Algo. ($\times 10^{-5}$)	
	RMS	HD	RMS	HD
50	2.3	4.1	2.3	3.8
25	5.2	8	5.8	8.5
10	13.3	19.7	15.2	21.6
5	23.8	34.6	28.3	40.1

To reduce the input model to the target sizes with our algorithm, we used $\tau = 10^{-6}$, 10^{-5}, 10^{-4} and 10^{-3}, respectively. All errors were measured in MeshLab

ple, if a collapsing tree is a Hamiltonian path, it reduces a mesh of any size to a point.

We compared the performance of our algorithm with the Papageorgiou algorithm [15], the most recently published GPU algorithm for mesh simplification. For both algorithms, we measured the numbers of iterations and running times when reducing the test models by up to 10 % of their original sizes and summarized the results in Table 3. The Papageorgiou algorithm required 35–40 iterations and ran approximately 11 times longer than ours. Our algorithm gets to the target sizes in two iterations for all the test models.

Figure 7 is a graph showing how the gargoyle model is differently reduced for a fixed τ. We iteratively simplified the gargoyle model and measured the triangle count at each iteration for different values of τ. Although the number of iterations to reach a convergence point seems to be proportional to the order of τ, it is bounded by five, which is the case of $\tau = \infty$. It shows that our GPU algorithm has a very high level of parallelism not achieved by any other previous algorithms based on edge collapsing.

To compare the quality of the simplified meshes produced by our algorithm, we measured the distance from the original mesh to a simplified mesh at each vertex, and calculated the root mean-square (RMS) errors and Hausdorff distances. Table 4 summarizes the errors of output meshes produced by MeshLab [3] and our algorithm for various target sizes. Although our results have error rates 20–30 % higher than those simplified by MeshLab, the order of measured errors

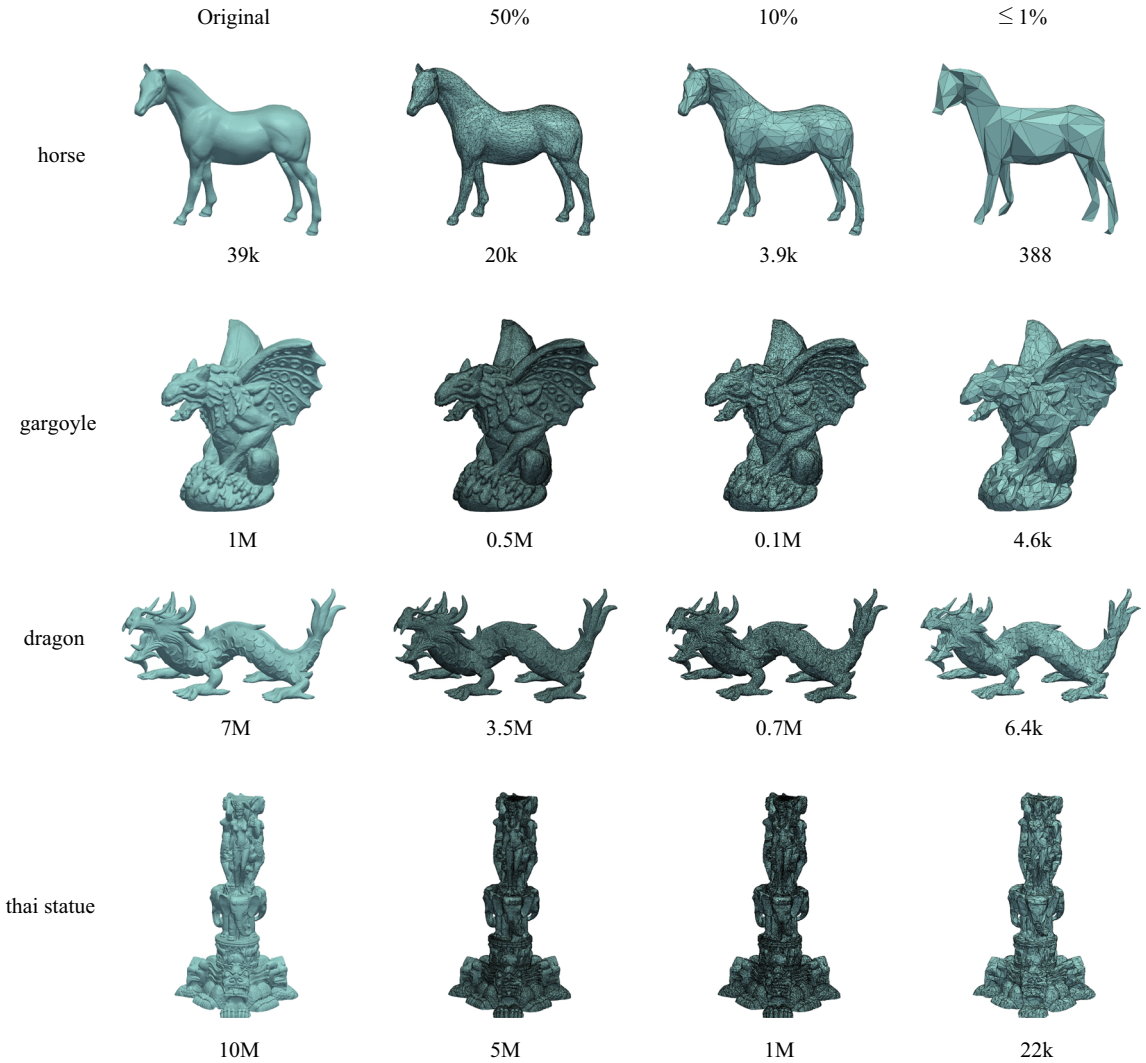


Fig. 8 Reduced mesh images for the four test models

is very low; the difference does not appear to be noticeable by visual inspection (Fig. 8).

7 Conclusion

We have proposed an efficient parallel edge-collapsing algorithm for triangular mesh simplification, which solves the problem of illegal mesh updates often arising from asynchronous edge collapsing by delaying mesh updates through an intermediate merging table. We first construct a set of collapsing trees embedded on an input triangle mesh, some of which are split into smaller trees if their collapsing costs are higher than a given threshold. Tree edges that reverse triangle normals and affect the topology type of the input mesh are removed from collapsing trees. We then collapse the tree edges by storing the merged edges information in a temporary table. Finally, the mesh data are updated by replacing the deleted edge IDs with the merging edges retrieved by table lookups. Our algorithm reduced a mesh of 10,000,000 triangles to 10 % of its original size in 620 ms with acceptable visual quality, which outperforms previous GPU algorithms by a factor of 10.

We control the reduction rate of our algorithm using only the error threshold τ, which constrains the sum of quadric errors of internal vertices merged to a root. Because it does not tell a reliable estimation on a reduced size, we need several trials to obtain its optimal value, to reach the target size in the smallest number of iterations without seriously reducing the quality. We may use a table of τ constructed with predefined triangle sizes and target sizes to estimate a suitable value of τ for a given case.

The size of an input mesh for simplification is bounded by the GPU memory. Because the current implementation

of our algorithm used almost 3GB GPU memory to simplify the Thai statue, we can handle up to 20 million triangles on our GPU, even though the performance is sufficient to simplify a mesh with 100 million triangles in less than 10 s. We note that large mesh models are not rare in real-world applications such as smart factory simulators. Our next plan is to extend the tree-collapsing algorithm to an out-of-core algorithm by partitioning an input mesh to sub-meshes of appropriate sizes.

Acknowledgments This research was supported by the Basic Science Research Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Education (NRF-2013R1A1A2011733).

References

1. Cabiddu, D., Attene, M.: Large mesh simplification for distributed environments. Comput. Graph **51**(C), 81–89 (2015)
2. Cellier, F., Gandoin, P.M., Chaine, R., Barbier-Accary, A., Akkouche, S.: Simplification and streaming of gis terrain for web clients. In: Proc. of the 17th International Conference on 3D Web Technology, Web3D '12, pp. 73–81. New York (2012)
3. Cignoni, P., Callieri, M., Corsini, M., Dellepiane, M., Ganovelli, F., Ranzuglia, G.: Meshlab: an open-source mesh processing tool. In: Scarano, V., Chiara, R.D., Erra U. (eds.) Proceedings of Eurographics Italian Chapter Conference, pp. 129–136 (2008)
4. DeCoro, C., Tatarchuk, N.: Real-time mesh simplification using the GPU. In: Proceedings of the 2007 Symposium on Interactive 3D Graphics and Games, I3D '07, pp. 161–166. ACM, New York (2007)
5. Dey, T.K., Edelsbrunner, H., Guha, S., Nekhayev, D.V.: Topology preserving edge contraction. Publ. Inst. Math. **66**, 23–45 (1998)
6. Garland, M., Heckbert, P.S.: Surface simplification using quadric error metrics. In: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '97, pp. 209–216. ACM Press/Addison-Wesley Publishing Co. (1997)
7. Grund, N., Derzapf, E., Guthe, M.: Instant level-of-detail. In: Proceedings of Vision, Modeling, and Visualization, pp. 293–299 (2011)
8. Guéziec, A.: Surface simplification with variable tolerance. In: Second Annual Intl. Symp. on Medical Robotics and Computer Assisted Surgery, pp. 132–139 (1995)
9. Hjelmervik, J., Léon, J.C.: GPU-accelerated shape simplification for mechanical-based applications. In: Proceedings of the IEEE International Conference on Shape Modeling and Applications 2007. SMI '07, pp. 91–102. IEEE Computer Society, Washington, DC (2007)
10. Hoppe, H.: Progressive meshes. In: Proc. of the 23rd Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '96, pp. 99–108. ACM, New York (1996)
11. Hoppe, H., DeRose, T., Duchamp, T., McDonald, J., Stuetzle, W.: Mesh optimization. Tech. Rep. TR 93-01-01. Dept. of Computer Science, University of Washington (1993)
12. Kobbelt, L., Campagna, S., peter Seidel, H.: A general framework for mesh decimation. In: Proceedings of Graphics Interface, pp. 43–50 (1998)
13. Lindstrom, P.: Out-of-core simplification of large polygonal models. In: Proceedings of the 27th Annual Conference on Computer Graphics and Interactive Techniques, SIGGRAPH '00, pp. 259–262 (2000)
14. Luebke, D.P.: A developer's survey of polygonal simplification algorithms. IEEE Comput. Graph. Appl. **21**(3), 24–35 (2001)
15. Papageorgiou, A., Platis, N.: Triangular mesh simplification on the GPU. Vis. Comput. **31**(2), 235–244 (2015)
16. Rossignac, J., Borrel, P.: Multi-resolution 3d approximation for rendering complex scenes. In: Modeling in Computer Graphics, pp. 455–465. Springer, Berlin, Heidelberg (1993)
17. Salinas, D., Lafarge, F., Alliez, P.: Structure-aware mesh decimation. Comput. Graph Forum **34**(6), 211–227 (2015)
18. Schaefer, S., Warren, J.: Adaptive vertex clustering using octrees. In: Proceedings of SIAM Geometric Design and Computing (2003)
19. Schroeder, W.J., Zarge, J.A., Lorensen, W.E.: Decimation of triangle meshes. SIGGRAPH Comput. Gr. **26**(2), 65–70 (1992)
20. Shontz, S.M., Nistor, D.M.: CPU-GPU algorithms for triangular surface mesh simplification. In: Proceedings of the 21st International Meshing Roundtable, pp. 475–492 (2012)
21. Xiong, H., Jiang, X., Zhang, Y., Shi, J.: Parallel simplification of large meshes on pc clusters. In: Proc. of the 8th Eurographics Conference on Parallel Graphics and Visualization, EGPGV '08, pp. 33–40 (2008)

Hyunho Lee received the BS degrees in information and computer engineering from Ajou University in 2010. Currently, he is a PhD candidate with the Graduate School of Life Media, Ajou University. His major research topic is a parallel processing on the GPU for 3D modeling.

Min-Ho Kyung received his BS and MS in Computer Science from POSTECH, Korea, in 1993 and 1995, respectively, and PhD in Computer Science from Purdue University in 2001. He joined Ajou University in 2002, and now is a professor of Department of Digital Media. His research interests include geometric modeling, computer-aided design, and robotics