RESEARCH ARTICLE

High-performance simplification of triangular surfaces using a GPU

Mohamed H. Mousa[1,2]*, **Mohamed K. Hussein**[1]

1 Department of Computer Science, Faculty of Computers and Informatics, Suez Canal University, Ismailia, Egypt, **2** Department of Information Technology, College of Computer Science at AlKamil, University of Jeddah, Jeddah, Saudi Arabia

* mohamed_mousa@ci.suez.edu.eg

OPEN ACCESS

Citation: Mousa MH, Hussein MK (2021) High-performance simplification of triangular surfaces using a GPU. PLoS ONE 16(8): e0255832. https://doi.org/10.1371/journal.pone.0255832

Editor: M. Usman Ashraf, University of Management and Technology, PAKISTAN

Received: February 26, 2021

Accepted: July 23, 2021

Published: August 5, 2021

Copyright: © 2021 Mousa, Hussein. This is an open access article distributed under the terms of the Creative Commons Attribution License, which permits unrestricted use, distribution, and reproduction in any medium, provided the original author and source are credited.

Data Availability Statement: The geometric models underlying the results presented in this paper are available directly from the following links: 1. Bunny: http://graphics.stanford.edu/pub/3Dscanrep/bunny.tar.gz 2. Dragon: http://graphics.stanford.edu/data/3Dscanrep/xyzrgb/xyzrgb_dragon.ply.gz 3. Lucy: http://graphics.stanford.edu/data/3Dscanrep/lucy.tar.gz 4. Gargoyle: http://visionair.ge.imati.cnr.it/ontologies/shapes/view.jsp?id=308-Gargoyle_1.7M Additional geometric models are available at: The Shape repository: http://visionair.ge.imati.cnr.it/ontologies/shapes/ The Stanford 3D Scanning Repository: http://graphics.stanford.edu/data/3Dscanrep/.

Abstract

Due to advances in high-performance computing technologies, computer graphics techniques—especially those related to mesh simplification—have been noticeably improved. These techniques, which have a strong impact on many applications, such as geometric modeling and visualization, have been well studied for more than two decades. Recent advances in GPUs have led to significant improvements in terms of speed and interactivity. In this paper, we present a mesh simplification algorithm that benefits from the parallel framework provided by recent GPUs. We customize the halfedge data structure for adaption with the dynamic memory restrictions of CUDA. The proposed algorithm is fully parallelized by employing a lock-free skip priority queue and a set of disjoint regions of the mesh. The proposed technique accelerates the simplification process while preserving the topological properties of the mesh. Some results and comparisons are provided to verify the efficiency of the proposed algorithm.

Introduction

With the recent improvements in high-performance computing (HPC) technologies, notable progress has been made in many fields, particularly in the computer graphics domain. Tasks in the computer graphics domain depend on piecewise linear surfaces, and with the increasingly strict requirements for quality, models with surfaces that contain hundreds of thousands or millions of polygons can be encountered. Triangles are commonly employed polygons for these surfaces, and the triangular mesh is one of the preferred boundary representations. In general, these meshes have a finite number of triangles and a certain surface resolution. Therefore, to overcome discretization issues, these meshes frequently contain a large number of triangles. Modern scanners and computer-aided designs can be utilized to obtain complex meshes. However, isosurface extraction techniques generate uniform and high-density meshes. These meshes are employed in many applications, such as surface rendering and visualization [1] and virtual reality [2].

Because of the restrictions of graphical devices, meshes are not easy to render and manipulate [3, 4]. Moreover, there is a tradeoff between the accuracy of a mesh and the processing

Funding: The author(s) received no specific funding for this work.

Competing interests: The authors have declared that no competing interests exist.

time. Therefore, in addition to improving the capabilities of hardware, efficient decimation algorithms are urgently needed [5, 6]. The principal objective of decimation techniques is to minimize the number of triangles in the graphical pipeline such that the visual difference is minimized; these techniques comprise a preprocessing phase. Mesh simplification has been intensively explored for many decades [7, 8]; however, recent improvements in the computational power of HPC graphics processing units (GPUs) has shifted research to new and promising areas [9–13].

Contribution

In this paper, we present a parallel algorithm for mesh decimation. The proposed algorithm relies on recent GPU advances and the efficient parallel computing platform offered by CUDA [14]. We have accelerated surface simplification by parallelizing elementary operations (i.e., edge contraction). Our proposed algorithm is summarized in (Fig 1), which shows the four basic steps of the algorithm.

1. A customized halfedge structure is developed to maintain the geometric and topological information in the GPU. Storing the data structure entirely in the device eliminates the memory overhead required to access the mesh during simplification.
2. A lock-free skip list is used as a customized priority queue to manipulate the list of candidate halfedges. These halfedges are ordered according to their contraction cost [15]. The

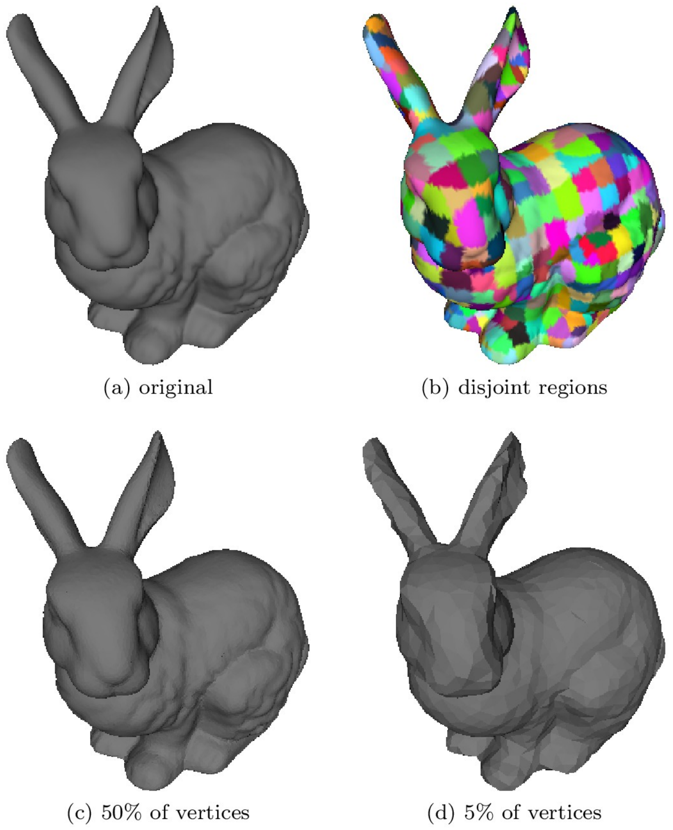

Fig 1. Overview of the proposed simplification process. (a) Stanford bunny, and (b) identification of disjoint parts; (c), (d) simplified versions of the mesh. The percentage here reflects the number of vertices with respect to the number of vertices of the original model.

https://doi.org/10.1371/journal.pone.0255832.g001

use of this kind of synchronized queue avoids problems related to the race condition of the concurrent threads manipulating the mesh.

3. A set of independent regions of the given mesh is constructed to enable the edge contraction operations along these regions to run in parallel. In fact, our partition scheme creates a balanced set of partitions even if the input meshes vary in terms of density. Therefore, by updating the topological information of the local neighborhoods after a set of parallel half-edge contraction operations on independent areas, the topological consistency of the simplified mesh is maintained.
4. A set of edge contraction operations is performed in parallel until the global stopping condition (usually a target number of vertices) is reached.

The proposed algorithm has the following advantages:

- It can remove many edges in parallel at a reduced time cost compared to other serial and parallel algorithms.
- It selects global minimum candidates for simplification, thereby guaranteeing the best geometric quality.

The remainder of this article is structured as follows: "Related work" presents a brief review of related research. "The proposed algorithm" starts with an overview of the proposed simplification algorithm, followed by a description of the data structure and quadric error metric (QEM) calculation in "Data structure" and "Halfedge contraction cost", respectively. The extended search list is discussed in "Customized priority queue", while the spatial partitioning of the mesh is given in "Disjoint partitioning". We present our results in "Result and comparisons" and conclude the paper in "Conclusion".

Related work

Various mesh simplification techniques have been proposed [5, 7, 16–20], and the HPC architectures of recent GPUs have inspired parallel techniques [9, 12, 13] that can be applied to effectively simplify meshes. In the following two subsections, we summarize the key research on serial and parallel methods.

Serial approaches

Serial approaches use iterative methods to simplify input meshes. These iterative methods are classified into two main groups: vertex clustering and edge contraction. The key differences between the two groups are as follows:

1. the selection of the candidate element to delete from the mesh;
2. the repositioning of the new or remaining elements after deletion.

The vertex clustering technique, which was first introduced by [21], voxelizes the mesh into n^3 voxels, where n is the dimension of the grid. Each nonempty voxel is replaced with a candidate vertex that represents the contained vertices [22]. Vertex clustering techniques are very fast and have high decimation rates. However, these techniques produce low-quality simplified meshes [7], as voxelization does not preserve the salient features of the surface.

In contrast, edge contraction techniques are based on the iterative application of a combinatorial operation that is referred to as edge contraction. This operation merges the two endpoints of a given edge into a new vertex by deleting one vertex and two (or one) triangles at a time. [7] introduced a simplification algorithm named QSlim that associates a geometric error with each edge contraction operation. Low-cost edge contraction enables the production of high-quality simplified meshes [23]. Simplification techniques using the QEM are a compromise between two types of methods that are very fast but have poor quality and are very slow but have high quality. Thus, both speed and quality are considered [24]. [25] described the vertex clustering operation as a set of successive edge contraction operations of certain vertices. A similar approach was proposed by [26]; their decimation algorithm is based on triangle collapse operations. However, these operations require many instances to be manipulated during an application.

Parallel approaches

Due to recent GPU advances, several techniques have aimed to transform serial approaches into parallel frameworks [10, 12, 13, 27]. However, early approaches focused on concurrency rather than quality [9].

Concerning the vertex clustering techniques, [9] implemented classic vertex clustering on GPUs using geometry shaders. They developed a fast and parallel octree vertex clustering method. However, the proposed approach suffers from the same quality problems faced by all clustering techniques. The quality of the mesh is based mainly on the cell dimension utilized during voxelization. [10, 28] distributed the simplification process between a central processing unit (CPU) and a GPU. Swapping the execution between a CPU and a GPU causes high memory overhead since data must be frequently transferred between these components.

Regarding edge contraction techniques, [29] adopted a progressive mesh data structure [17] to be executed on a GPU, and [30] proposed an iterative parallel GPU-based edge contraction simplification technique. However, the identification of independent parts is not straightforward. In addition, postprocessing sorting and topological updates are applied after each group of steps, which increases the processing time. Moreover, [13] implemented a full-side iterative GPU simplification for triangular meshes that was faster than classic CPU iterative techniques. Similarly, [12] introduced an iterative algorithm that was faster than the method of [13]. However, the iterative nature of the two algorithms slows the simplification process. In addition, the update of the QEM at each vertex after removal is not applicable. [31] proposed a probabilistic selection scheme for edge contraction. They reduced the amount of required memory by choosing to not use a global cost list for the ordered edges. However, the selection of edges with the global minimum is not guaranteed, which affects the overall mesh quality.

The proposed algorithm

Our proposed algorithm mainly benefits from recent advances in CUDA [14]. The algorithm can be summarized as a set of parallel simplification CUDA kernels that work on independent areas of the mesh. The simplification process is summarized in the following four steps:

1. Adapt a halfedge data structure to maintain the input mesh entirely in the GPU memory.
2. Calculate the edge contraction cost, the QEM, for each halfedge.
3. Build a customized priority queue via a skip list of these costs to choose the candidate pair of vertices to be contracted.
4. Construct the set of disjoint parts of the mesh using k-d tree space partitioning.
5. Apply the decimation process.

In the following subsections, we describe each step and their implementation issues.

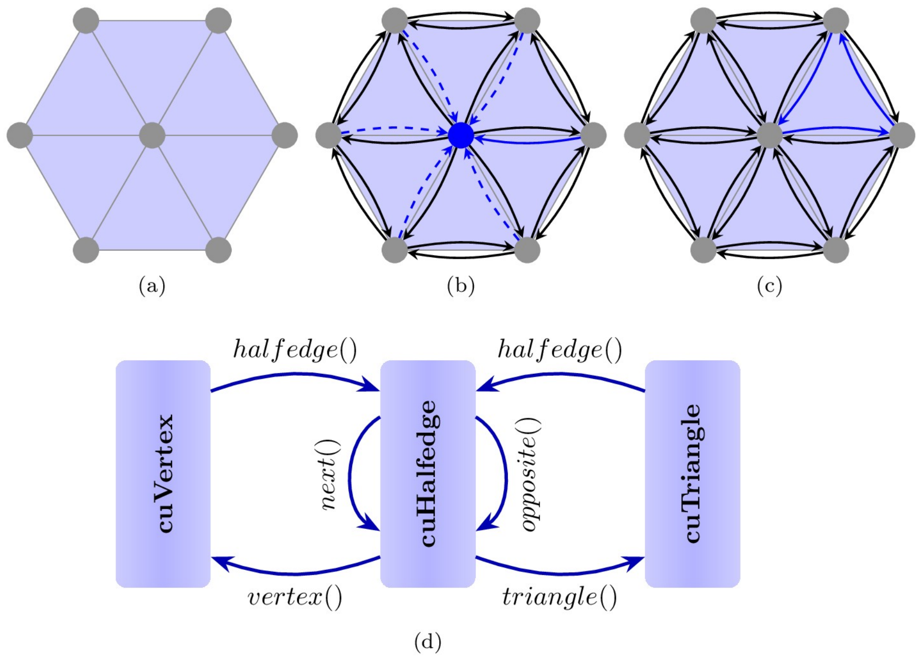

Fig 2. The halfedge data structure. (a) A triangular mesh, (b) the vertex is identified by one of its incident halfedges, (c) the triangle is identified by its three incident halfedges, and (d) the design of the data structure reflecting the relationship between the three components.

https://doi.org/10.1371/journal.pone.0255832.g002

Data structure

This subsection describes the proposed customization of the halfedge data structure. CUDA does not currently support dynamic memory management for memory spaces that are allocated in the global memory shared between the host and device sides. However, vertices, halfedges and triangles are dynamically created and deleted during simplification. To overcome this issue, we have customized the halfedge data structure [32] for adaption to CUDA memory management restrictions.

Our customized halfedge data structure is based on three main substructures: cuVertex, cuHalfedge and cuTriangle, as shown in (Fig 2). The input mesh is passed to the pipeline as two vectors; from the vertices and triangles, we build an array of halfedges. Each cuVertex provides the following information:

- the geometric coordinates of the vertex (12 bytes);
- a handle for one of its incident halfedges (4 bytes);
- a handle for the container partition (see the “Disjoint partitioning” section) (4 bytes);
- its QEM (see the “Halfedge contraction cost” section) (44 bytes).

Similarly, each cuHalfedge provides the following information:

- a handle for its incident cuVertex (4 bytes);
- a handle for its incident cuTriangle (4 bytes);
- a handle for its opposite cuHalfedge (if it exists) (4 bytes);

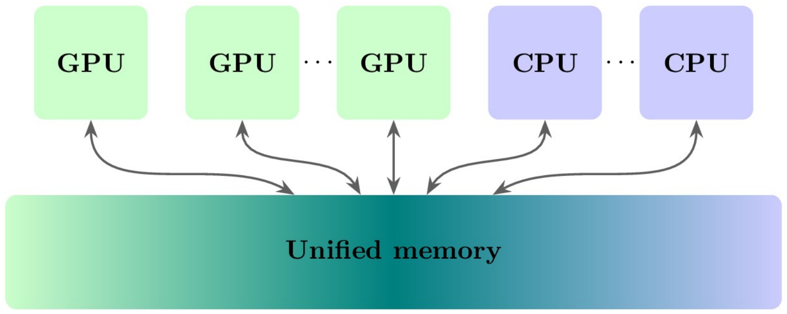

Fig 3. The unified memory.

https://doi.org/10.1371/journal.pone.0255832.g003

- a handle for the next cuHalfedge in the incident triangle (4 bytes);
- its contraction cost (4 bytes).

Each cuTriangle provides a handle for one of its three incident halfedges (4 bytes). The whole structure should reside in the GPU memory and be allocated from the CPU host side. Moreover, if we have more than one GPU, then the data structure should be accessible for all the GPUs' kernels in addition to creating the CPU. The only solution is the use of CUDA's page migration engine, i.e., "cudaMallocManaged()", which supports the unified memory principle, as shown in (Fig 3). The unified memory provides a single address space that is reachable from any CPU or GPU in the system. The unified memory may cause memory overhead when migrating page tables from one physical memory to another physical memory. To overcome this issue, we transfer all required data to the GPU memory before launching the processing kernels using the CUDA application programming interface (API), "cudaMemPrefetchAsync()". In our experiments, we use a single GPU; however, the application on multiple GPUs is straightforward.

Now, we will describe how to build the vertex, halfedge and triangle relationships. (Algorithm 1) describes how to build the cuHalfedge data structure from lists of vertices and triangles. The first loop fills in the triangle and vertex for every halfedge. The first loop also assigns the halfedge attribute for the associated triangles and vertices. The second and third loops build the opposite relationships between related halfedges.

For the contraction operation, a pair of vertices is tested before contraction to ensure that it does not violate the manifold properties of the mesh. Edge contraction is applied as depicted in (Fig 4). The vertex *h.opposite*().*vertex*() is deleted, and *h.vertex*() becomes the new vertex. Once the halfedge is contracted, the incident triangles and their halfedges are deleted, and the

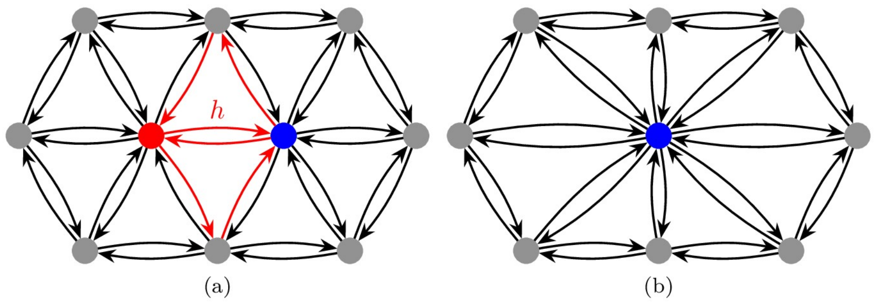

Fig 4. The edge contraction operation. (a) The two candidate vertices, and (b) the mesh after contracting the corresponding edge. The colored halfedges (red) and their incident triangles are marked as deleted.

https://doi.org/10.1371/journal.pone.0255832.g004

opposite relationships between the other halfedges around the new vertex are updated. On the other hand, since the data are allocated in the GPU global memory from the CPU host side, the memory occupied by the mentioned arrays cannot be dynamically freed or resized during simplification. To overcome this issue, we associate a flag with each vertex, halfedge and triangle to indicate whether the entry has been previously deleted from the data structure. To accelerate memory access, the deleted items are moved to the end of their container arrays as described in (Algorithm 2). The contraction steps are summarized in (Algorithm 3).

Algorithm 1: Build the data structure.

```
input: $T$, $V$
Output: H
for $i = 0 \cdots \#T$ do in parellel
  $H[3i + k] \cdot triangle = i;$                          // $(k = 0 \cdots 2)$
  $H[3i + k] \cdot next = H[3\,i + (k + 1)\%3];$           //$(k = 0 \cdots 2)$
  $H[3i + k] \cdot vertex = T[i][k];$                      //$(k = 0 \cdots 2)$
  $T[i][k] \cdot halfedge = H[3i + k];$                    //$(k = 0 \cdots 2)$
  $T[i] \cdot halfedge = H[3i];$
end
// Build the opposite relationships
for $i = 0 \cdots \#H$ do
  // Build a list of incident halfedges for each vertex
  $H_{tmp}[H[i] \cdot vertex] \cdot append(H[i]);$
end
for $i = 0 \cdots \#V$ do in parallel
  for $j = 0 \cdots H_{tmp}[i].size$ do
    for $k = 0 \cdots H_{tmp}[i].size$ and $k \neq j$ do
      $h_1 = H_{tmp}[i][j];$
      $h_2 = H_{tmp}[i][k];$
      if $h_2 \cdot next \cdot vertex \in h_1 \cdot triangle$ then
        $h_1 \cdot opposite = h_2 \cdot next;$
        $h_2 \cdot next \cdot opposite = h_1;$
      end
    end
  end
end
```

Algorithm 2: The functions deleteVertex, deleteHalfedge and deleteTriangle replace the corresponding item by the last vertex, halfedge and triangle in the list, respectively.

```
Function deleteVertex(cuVertex v)
  $tmp \leftarrow$ last vertex;
  $v \leftarrow tmp;$
  $h \leftarrow v \cdot halfedge;$
  repeat
    $h \cdot vertex \leftarrow v;$
    $h \leftarrow h \cdot next \cdot opposite;$
  until $h \neq v \cdot halfedge;$
  mark $tmp$ as deleted;
end
Function deleteHalfedge(cuHalfedge h)
  $tmp \leftarrow$ last halfedge;
  $h \leftarrow tmp;$
  $h \cdot next \cdot next \cdot next \leftarrow h;$
  $h \cdot opposite \cdot opposite \leftarrow h;$
  mark $tmp$ as deleted;
end
Function deleteTriangle(cuTriangle t)
  $tmp \leftarrow$ last triangle;
  $t \leftarrow tmp;$
  $t \cdot halfedge \cdot triangle \leftarrow t$;
  $t \cdot halfedge \cdot next \cdot triangle \leftarrow t$;
  $t \cdot halfedge \cdot next \cdot next \cdot triangle \leftarrow t$;
  mark $tmp$ as deleted;
end
```

Algorithm 3: The steps of the halfedge contraction.

```
Function contractHalfedge(cuHalfedge h)
  deleteVertex($h \cdot opposite \cdot vertex$);
  deleteTriangle($h \cdot face$);
  deleteTriangle($h \cdot opposite \cdot face$);
  /* update the incident property of halfedges around the remaining
vertex: $h \cdot vertex$ */
  $tmp \leftarrow h \cdot opposite$;
  repeat
    $tmp \cdot vertex \leftarrow h \cdot vertex$;
    $tmp \leftarrow tmp \cdot next \cdot opposite$;
  until $tmp \leftarrow h \cdot opposite$;
  /* update the opposite property between the remaining halfedges */
  $h1 \leftarrow h \cdot next \cdot opposite$;
  $h2 \leftarrow h \cdot next \cdot next \cdot opposite$;
  $h3 \leftarrow h \cdot opposite \cdot next \cdot opposite$;
  $h4 \leftarrow h \cdot opposite \cdot next \cdot next \cdot opposite$;
  $h1 \cdot opposite \leftarrow h2$;
  $h2 \cdot opposite \leftarrow h1$;
  $h3 \cdot opposite \leftarrow h4$;
  $h4 \cdot opposite \leftarrow h3$;
  deleteHalfedge($h$);
  deleteHalfedge($h \cdot next$);
  deleteHalfedge($h \cdot next \cdot next$);
  deleteHalfedge($h \cdot opposite$);
  deleteHalfedge($h \cdot opposite \cdot next$);
  deleteHalfedge($h \cdot opposite \cdot next \cdot next$);
end
```

Halfedge contraction cost

In this section, we describe how to evaluate the QEM associated with each vertex. A plane is identified by the equation $\boldsymbol{n} \cdot \boldsymbol{p} + d = 0$, where:

- $n = (n_x, n_y, n_z)$ is the unit normal, i.e., a direction perpendicular to the plane such that $n_x^2 + n_y^2 + n_z^2 = 1$, and
- d is an offset scalar representing the signed distance from the origin to the plane.

For any point that does not lie on the plane, the expression $\boldsymbol{n} \cdot \boldsymbol{p} + d$ represents the signed distance from the given point p to the plane. The squared distance can be evaluated by

$$D(\boldsymbol{p}) = (n_x x + n_y y + n_z z + d)^2 \tag{1}$$

This equation can be rewritten in matrix form as follows:

$$D(\boldsymbol{p}) = (\boldsymbol{p}^t(\boldsymbol{n}\boldsymbol{n}^t)\boldsymbol{p} + 2(d\boldsymbol{n})^t\boldsymbol{p} + d^2) \tag{2}$$

The quadric of the plane, $\boldsymbol{n} \cdot \boldsymbol{p} + d = 0$, is defined as $Q = (q_1, q_2, q_3)$, where:

- $q_1 = \boldsymbol{n}\boldsymbol{n}^t$ is a 3×3 symmetric matrix,
- $q_2 = d\boldsymbol{n}$ is a vector, and

- $q_3 = d^2$ is a scalar.

Therefore, given any quadric, the squared distance is evaluated by

$$Q(\boldsymbol{p}) = \boldsymbol{p}^t q_1 \boldsymbol{p} + 2q_2^t \boldsymbol{p} + q_3 \tag{3}$$

Each vertex of the mesh is associated with a set of planes that correspond to the incident triangles. The sum of the squared distances is given by the following equation:

$$Q(v) = \sum_i Q_i(v) \tag{4}$$

The linearity of the Q operator makes the sum of a set of quadrics equal to the sum of the corresponding components of the triples (q_1^i, q_2^i, q_3^i), i.e.,

$$Q(v) = \sum_i Q_i(v) = \left(\sum_i q_1^i, \sum_i q_2^i, \sum_i q_3^i \right)(v) \tag{5}$$

The triangle plane may be shared with at least three vertices. Therefore, to remove computational redundancy, a set of CUDA kernels is executed to calculate the plane equation and then the quadrics of the triangles of the mesh. Similarly, a set of CUDA kernels is executed to sum the corresponding quadric for each vertex. Hence, the contraction cost of a halfedge h is determined by the following steps:

1. Determine the vertex $v_i = h.vertex()$.
2. Determine the vertex $v_j = h.opposite().vertex()$.
3. Determine the position of the new vertex v.
4. Evaluate the new quadric $Q = Q_i + Q_j$, which is the quadric of the new vertex v.
5. Obtain the edge contraction cost $Q(v)$.

The position of the new vertex v should be chosen such that $Q(v)$ is minimized. This minimization is expressed as $\nabla Q(v) = 2q_1\, v + 2q_2 = 0$, which yields $v' = -q_1^{-1} q_2$; its minimization error is expressed as follows:

$$\begin{aligned} Q(v') &= v'^t q_1 v' + 2q_2^t v' + q3 \\ &= (-q_1^{-1} q_2)^t q_1 (-q_1^{-1} q_2) + 2q_2^t(-q_1^{-1} q_2) + q_3 \\ &= -q_2^t q_1^{-1} q_2 + q_3 \end{aligned} \tag{6}$$

which is a special case of the standard positive definite quadratic minimization that can be solved by using the Cholesky factorization [33, 34]. During simplification, the halfedges with minimum costs are selected for contraction.

Customized priority queue

In this section, we describe how to maintain the costs of the set of halfedges in a customized priority queue. Concurrent access to the priority queue during simplification hinders the use of classic data structures for priority queues.

Classically, the implementation of queues using arrays rather than linked lists is recommended. However, lists are better than arrays for the insertion and removal operations. The main drawback of using a sorted linked list is the sequential access of its elements, $O(n)$, with $O(\lg n)$ for sorted arrays. To overcome this problem, we employ lock-free skip lists to implement

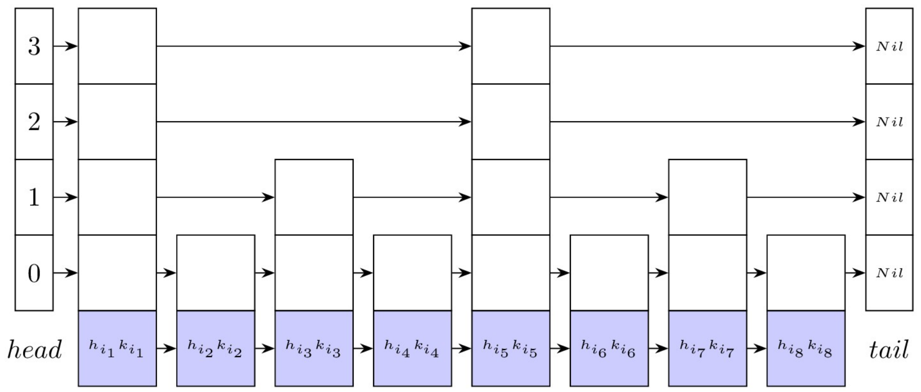

Fig 5. A skip list consisting of 4 linked lists. The original list of pairs (vertices, keys) is at level 0 and sorted in ascending order. Each level contains a sublist from the preceding level.

https://doi.org/10.1371/journal.pone.0255832.g005

our priority queue [35]. Additionally, this skip-list priority queue prevents the synchronization drawbacks that can occur due to race conditions. Moreover, the speed of the search, insertion and removal operations is increased to $O(\lg n)$ due to the multilayer structure.

A skip list is a nondeterministic framework based on linked lists, which consists of a multi-level linked list. The 0-level contains all elements of the list. The subsequent levels contain fewer elements. Each level is a subset of the preceding level and a superset of the next level. In (Fig 5), the original list is the set of halfedges, h_{i_j}, sorted by their quadric costs k_{i_j}. The insertion of any halfedge into the skip list starts at level 0, while element removal starts from the top level.

We will now describe how the concurrent insertion and removal of halfedges can be performed safely by using the CUDA atomic function *atomicCAS()* [14]. The *atomicCAS()* function has the following form: int *atomicCAS*(int **address*, int *compare*, int *val*). The function loads the value *old* located at *address*, evaluates the expression *(old == compare? val: old)* and saves the results at the location *address*. Thus, *atomicCAS()* ensures that the value at *address* is not changed using other threads. If the value is changed, then the function can easily detect this change.

The two main primitives that we employ are the insert operation and delete operation. To insert a pair (h_i, k_i)—the halfedge and its quadric cost—into the list, we navigate the list until we obtain the smallest entry (h_j, k_j) such that $k_j > k_i$. A new entry is created with a value of (h_i, k_i), and *address*$((h_i, k_i))$*.next* is set to *address*$((h_j, k_j))$. Next, *atomicCAS()* is applied to set *address*$((h_j, k_j))$*.previous.next* to *address*$((h_i, k_i))$. Furthermore, to remove the pair (h_i, k_i) from the list, *atomicCAS()* is applied to set *address*$((h_i, k_i))$*.previous.next* to *address*$((h_i, k_i))$*.next*. The node that contains (h_i, k_i) is not deleted due to CUDA restrictions but is instead added to a deleted list for further usage. During simplification, the halfedge nodes, for which the costs must be updated, are simply deleted, and the new values are inserted. (Algorithm 4) shows how the Find function determines the appropriate position of the halfedge with respect to its cost in the skip list for further insertion or deletion.

Algorithm 4: The Find function determines the appropriate position of *key* in the skip list.

```
Function Find(key)
  tmp ← top-first element in the skip list;
  // step down
  while tmp.below ≠ null do
    tmp ← tmp.below;
    // step forward
    while $key \geq tmp.next.cost$ do
      $tmp \leftarrow tmp.next$;
    end
  end
  return $tmp$;
end
```

Disjoint partitioning

Once the priority queue is constructed for the given triangular mesh, we must construct a set of independent areas on which we will parallelize the simplification process. In this section, we describe the major steps for constructing a set of disjoint partitions of the mesh using the k-d tree.

Algorithm 5: Space partitioning of $\mathcal{M}$ using the k-d tree.

```
$root \leftarrow AABB(\mathcal{M})$;
$SplitList \leftarrow root$;
repeat                                                    // parallel
  $tmp \leftarrow Pop(SplitList)$;
  if $C(tmp)$ then
    $(leftchild, rightchild) \leftarrow Split(tmp)$;
    $SplitList.append(leftchild)$;
    $SplitList.append(rightchild)$;
  end
until $SplitList$ is empty;
```

The partitioning procedure is summarized in (Algorithm 5). The algorithm starts by determining the axis-aligned bounding box of the mesh $\mathcal{M}$. The initial bounding box is the root of the k-d tree. Starting from this root, the following steps are applied for the candidate node:

1. Evaluate the splitting condition, $C(P_i)$, which in our case is a fixed number of contained points, i.e.,

$$C(P_i) = \begin{cases} true & |P_i| \leq \zeta \\ false & otherwise \end{cases} \tag{7}$$

   where ζ is the desired number of points in each cell P_i. This splitting condition generates a balanced number of partitions, even if the input mesh is not uniformly sampled.

2. If $C(P_i)$ is satisfied, then P_i is appended to the split queue, *SplitList*.

3. If $C(P_i)$ is not satisfied, then P_i is a leaf node and is removed from the *SplitList* queue.

Simplification process

In the previous sections, we showed how to build the priority queue and identify the independent regions. Now, the simplification process is ready for execution in parallel. Since the contraction operator changes the topological properties of the mesh, we allow a single halfedge contraction for each independent region of the mesh at a time, which guarantees the consistency of the neighborhood properties among the vertices of the mesh.

Without loss of generality, our stopping criterion is that the mesh must be simplified to a certain number of vertices. Note that each contraction process reduces the total number of vertices by 1. The simplification process starts by creating the set of threads T. T picks the top halfedges from the customized priority queue, i.e., the halfedges with the minimum cost. For each independent region P_i, we create the thread T_i. T_i picks a halfedge h from the top of the priority queue. If $h \notin P_i$, then the halfedge is skipped, and T_i picks the next candidate halfedge. However, if only one of the two endpoints belongs to P_i, then the halfedge is selected, and the neighbor partition is blocked. Thus, the halfedge that crosses two adjacent partitions is contracted by the first partition thread, and the other partition thread waits until this halfedge is contracted, which avoids any overlap of the topological update of the conflicting areas.

However, a thread does not execute the contraction operation unless the halfedges of the smaller costs are picked from the queue for contraction or the total number of picked and skipped halfedges does not reach the target number of vertices. This condition gives a higher priority to contraction of halfedges with less cost to minimize the global geometric error. The simplification process is summarized in (Algorithm 6).

Algorithm 6: Major steps of the proposed algorithm.

```
Calculate the QEM for the set of halfedges // parallel
Build the customized priority queue // parallel
Construct the independent partitions {$P_i$} // parallel
$skipped \leftarrow 0$;                                          // shared
$removed \leftarrow 0$;                                          //shared
for each $P_i$ do                                                // parallel
  pick a halfedge $h$;
  while $h \notin P_i$ do
    if $h \cap P_i \neq \phi$ then
      find $P_j$ such that $h \cap P_j \neq \phi$;
      wait($P_j$, $h$ is contracted);
    end
    $skipped \leftarrow skipped + 1$;
    pick another $h$;
  end
  wait($P_i$, $removed + skipped < target$);
  contract $h$;
  if $removed = target$ then
    exit;
  end
end
```

As mentioned, the proposed simplification process guarantees decimation of the edges with the lowest costs. Therefore, the set of edges to be contracted will be identical to that of QSlim. The only difference is that these edges will be contracted in a different order, which will not affect the quality measure since the error metric is a linear operator.

Results and comparisons

We implement the algorithms presented in this paper by using NVIDIA's CUDA framework [14] on an Intel Core i7–8565U CPU @(1.80 GHz,1.99 GHz) with a GeForce MX130 4GB GPU on Windows 10. Recent architectures of CUDA, especially 5.0, enable direct memory access between the host and the kernel codes. In all algorithms described in this paper, statements that are said to be run in "parallel" are executed as parallel GPU device kernels. Otherwise, they are implemented in serial mode. Considering some CUDA driver limitations, we do not allow device threads to run for more than 8 s. CUDA assumes that device kernels have a short execution time. In cases in which a device kernel reaches its execution time limit, we stop the kernel and regenerate another set to complete the job.

The geometric models underlying the results presented in this paper can be found in [36], see "S1 Text". (Table 1) presents the numbers of points and triangular faces of the meshes utilized in our experiments. We simplified the input models to a target number of vertices.

Table 1. The set of meshes used in our experiments.

Model	#Points	#Faces
Bunny	37000	73996
Gargoyle	863210	1,726,416
Dragon	3609455	7,218,906
Lucy	14,027,872	28,055,742

https://doi.org/10.1371/journal.pone.0255832.t001

(Fig 6) shows the simplification of the Dragon into levels of detail of 1%, 5% and 50%. The simplification process creates a smooth surface for each simplified version.

To prove the effectiveness of the proposed approach, we compare our results with those of the QSlim serial algorithm [7] implemented by MeshLab v2016.12 [37] and the MCS GPU algorithm [31]. In addition, we use the geometric deviation, as shown in (Fig 7), as the quality measure [38]. The geometric deviation evaluates the geometric differences between two meshes, in our case, the input and simplified meshes. This is formally defined as follows [39]:

$$\max_{p \in M_1} \{d(p, M_2)\} \tag{8}$$

where M_1 is the input mesh, M_2 is the simplified mesh and d is the Euclidean distance. The timing and quality measures of [31] are extracted from the associated paper. (Table 2) shows the times and geometric qualities for the simplification process applied to some of the models of (Table 1) with respect to the mentioned vertex targets. (Table 2) shows that the rate of decimation of vertices per second decreases for large meshes; this phenomenon is attributed to the size of the customized priority queue. The construction and search of small priority queues are faster than those of dense priority queues.

Additionally, we compare our results with the timing and quality of the GPU-based algorithm [12]. As mentioned in "Related work", [12] applied an iterative approach. In each iteration, a set of parallel edge contraction operations is performed until a target number of vertices

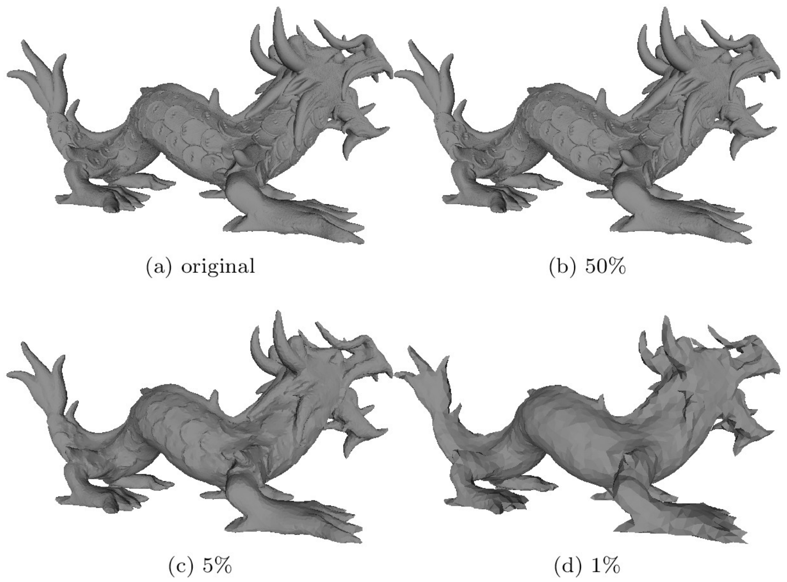

Fig 6. The simplification of the Dragon into different levels of detail. (a) The original dataset, (b) 50% of vertices, (c) 5% of vertices, and (d) 1% of vertices.

https://doi.org/10.1371/journal.pone.0255832.g006

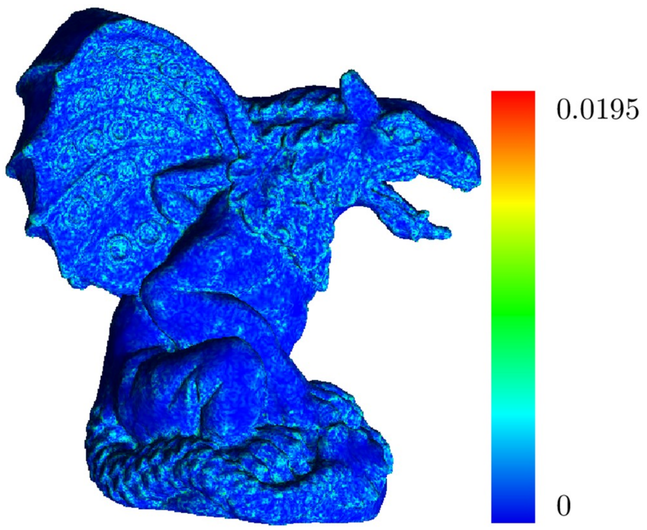

Fig 7. The geometric deviation of a simplified version of the Gargoyle model at 10% of its original number of vertices.

https://doi.org/10.1371/journal.pone.0255832.g007

Table 2. The time expended in seconds and the quality measures of the simplification of the input meshes in comparison with those of QSlim [7] and MCS [31].

Models	Target	QSlim [7]		MCS [31]		ours	
		time	quality	time	quality	time	quality
Gargoyle	589,860	22.86	0.01966	1.30	0.03542	1.05	0.01967
Dragon	2,466,461	109.02	0.0815	6.39	0.0836	4.29	0.0816
Lucy	9,585,712	469.089	0.0186	28.03	0.032	22.05	0.0187

https://doi.org/10.1371/journal.pone.0255832.t002

Table 3. Comparison of the timing in seconds and the quality of the simplification of the Gargoyle model using [12] and our proposed algorithm.

Target	[12]		Ours	
	time	quality	time	quality
25%	3.9	0.01524	0.73	0.00253
10%	5.1	0.03357	0.95	0.01366
5%	5.6	0.05961	1.05	0.01967

https://doi.org/10.1371/journal.pone.0255832.t003

is reached. The higher the number of vertices decimated in each iteration is, the lower the quality of the mesh. To improve the mesh quality, the number of decimated vertices should be decreased in each iteration, which will cause a dramatic increase in time. (Table 3) shows the timing and mesh quality of our algorithm and those of [12] at targets of 25%, 10% and 5% using 18 iterations, 29 iterations and 37 iterations, respectively. The timing and quality evaluations are extracted from the related paper.

Conclusion

In this paper, we introduce an approach for triangular surface simplification using recent GPU advances. Our approach follows the edge contraction framework, which presents a high-quality surface simplification. We build a customized priority queue based on a skip list for the set of candidate halfedges. This skip list enables simultaneous update of the customized priority queue. The proposed approach identifies a set of independent regions on the surface, on which we apply parallelism. The applied parallelism significantly reduces the time required for the overall process. Compared to competing serial and parallel algorithms, the proposed approach is both simple and efficient.

Supporting information

S1 Text.
(PDF)

Author Contributions

Conceptualization: Mohamed H. Mousa, Mohamed K. Hussein.

Formal analysis: Mohamed K. Hussein.

Methodology: Mohamed H. Mousa.

Software: Mohamed H. Mousa, Mohamed K. Hussein.

Supervision: Mohamed K. Hussein.

Validation: Mohamed H. Mousa, Mohamed K. Hussein.

Writing – review & editing: Mohamed H. Mousa, Mohamed K. Hussein.

References

1. Pütz S, Wiemann T, Hertzberg J. Tools for Visualizing, Annotating and Storing Triangle Meshes in ROS and RViz. In: ECMR. IEEE; 2019. p. 1–6.
2. Han P, Zhao G. A review of edge-based 3D tracking of rigid objects. Virtual Reality & Intelligent Hardware. 2019; 1(6):580–596. https://doi.org/10.1016/j.vrih.2019.10.001
3. Yuan Y, Wang R, Huang J, Jia Y, Bao H. Simplified and tessellated mesh for realtime high quality rendering. Comput Graph. 2016; 54:135–144. https://doi.org/10.1016/j.cag.2015.07.011
4. Liu X, Lin L, Wu J, Wang W, Wang CCL. Generating sparse self-supporting wireframe models for 3D printing using mesh simplification. Graphical Models. 2018; 98:14–23. https://doi.org/10.1016/j.gmod.2018.05.001
5. Choi HK, Kim HS, Lee KH. An improved mesh simplification method using additional attributes with optimal positioning. The International Journal of Advanced Manufacturing Technology. 2010; 50(1):235–252.
6. Kwon S, Mun D, Kim BC, Han S. Feature shape complexity: a new criterion for the simplification of feature-based 3D CAD models. International journal of advanced manufacturing technology. 2017; 88(5):1831–1843.
7. Garland M, Heckbert PS. Surface Simplification Using Quadric Error Metrics. In: Proceedings of the 24th Annual Conference on Computer Graphics and Interactive Techniques. SIGGRAPH 97. USA: ACM Press/Addison-Wesley Publishing Co.; 1997. p. 209–216. Available from: https://doi.org/10.1145/258734.258849.
8. Kobbelt L, Campagna S, Seidel HP. A General Framework for Mesh Decimation. In: Davis WA, Booth KS, Fournier A, editors. Graphics Interface. Canadian Human-Computer Communications Society; 1998. p. 43–50.
9. 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. New York, NY, USA: Association for Computing Machinery; 2007. p. 161–166.
10. Hjelmervik J, Leon J. GPU-Accelerated Shape Simplification for Mechanical-Based Applications. In: IEEE International Conference on Shape Modeling and Applications 2007 (SMI'07); 2007. p. 91–102.

11. Mousa MH, Hussein MK. Surface approximation using GPU-based localized fourier transform. Journal of King Saud University—Computer and Information Sciences. 2020;. https://doi.org/10.1016/j.jksuci.2020.04.010

12. Odaker T, Kranzlmueller D, Volkert J. GPU-Accelerated Real-Time Mesh Simplification Using Parallel Half Edge Collapses. In: Kofroň J, Vojnar T, editors. Mathematical and Engineering Methods in Computer Science. Cham: Springer International Publishing; 2016. p. 107–118.

13. Papageorgiou A, Platis N. Triangular Mesh Simplification on the GPU. Vis Comput. 2015; 31(2):235–244. https://doi.org/10.1007/s00371-014-1039-x

14. NVIDIA. CUDA Toolkit Documentation 10.2; 2019. Available from: docs.nvidia.com/cuda/index.html.

15. Garland M, Heckbert P. Quadric-Based Polygonal Surface Simplification. USA: Carnegie Mellon University; 1999.

16. Chattopadhyay A, Carr HA, Duke DJ, Geng Z, Saeki O. Multivariate topology simplification. Comput Geom. 2016; 58:1–24.

17. Hoppe H. Progressive meshes. In: Fujii J, editor. Proceedings of the 23rd annual conference on Computer graphics and interactive techniques—SIGGRAPH'96. New York, New York, USA: ACM Press; 1996. p. 99–108. Available from: http://portal.acm.org/citation.cfm?doid=237170.237216.

18. Lee H, Kyung MH. Parallel mesh simplification using embedded tree collapsing. The Visual Computer. 2016; 32(6-8):967–976. https://doi.org/10.1007/s00371-016-1242-z

19. Schroeder WJ, Zarge JA, Lorensen WE. Decimation of triangle meshes. In: Proceedings of the 19th annual conference on Computer graphics and interactive techniques—SIGGRAPH'92. SIGGRAPH 92. New York, New York, USA: ACM Press; 1992. p. 65–70. Available from: https://doi.org/10.1145/133994.134010 http://portal.acm.org/citation.cfm?doid=133994.134010.

20. Wang Y, Zheng J, Wang H. Fast Mesh Simplification Method for Three-Dimensional Geometric Models with Feature-Preserving Efficiency. Scientific Programming. 2019; 2019:1–12.

21. Rossignac J, Borrel P. Multi-resolution 3D approximations for rendering complex scenes. In: Falcidieno B, Kunii TL, editors. Modeling in Computer Graphics. IFIP Series on Computer Graphics. Springer; 1993. p. 455–465.

22. Luebke DP, Erikson C. View-dependent simplification of arbitrary polygonal environments. In: Owen GS, Whitted T, Mones-Hattal B, editors. SIGGRAPH. ACM; 1997. p. 199–208.

23. Heckbert PS, Garland M. Optimal triangulation and quadric-based surface simplification. Comput Geom. 1999; 14(1-3):49–65. https://doi.org/10.1016/S0925-7721(99)00030-9

24. Boubekeur T, Alexa M. Mesh simplification by stochastic sampling and topological clustering. Comput Graph. 2009; 33(3):241–249. https://doi.org/10.1016/j.cag.2009.03.025

25. 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. USA: ACM Press/Addison-Wesley Publishing Co.; 2000. p. 259–262. Available from: https://doi.org/10.1145/344779.344912.

26. Pan Z, Zhou K, Shi J. A new mesh simplification algorithm based on triangle collapses. Journal of computer science and technology. 2001; 16(1):57–63. https://doi.org/10.1007/BF02948853

27. Hu L, Sander PV, Hoppe H. Parallel view-dependent refinement of progressive meshes. Proceedings of I3D 2009: The 2009 ACM SIGGRAPH Symposium on Interactive 3D Graphics and Games. 2009; p. 169–176. https://doi.org/10.1145/1507149.1507177

28. Shontz SM, Nistor DM. CPU-GPU Algorithms for Triangular Surface Mesh Simplification. In: Jiao X, Weill JC, editors. Proceedings of the 21st International Meshing Roundtable. Springer; 2012. p. 475–492.

29. Liang Hu, Sander PV, Hoppe H. Parallel View-Dependent Level-of-Detail Control. IEEE Transactions on Visualization and Computer Graphics. 2010; 16(5):718–728. https://doi.org/10.1109/TVCG.2009.101 PMID: 20616388

30. Grund N, Derzapf E, Guthe M. Instant Level-of-Detail. In: Vision, Modeling, and Visualization. The Eurographics Association; 2011. p. 293–299.

31. Koh N, Zhang W, Zheng J, Cai Y. GPU-based Multiple-Choice Scheme for Mesh Simplification. In: Magnenat-Thalmann N, Kim J, Rushmeier HE, Lévy B, Zhang HR, Thalmann D, editors. Computer Graphics International. ACM; 2018. p. 195–200.

32. Baumgart BG. A Polyhedron Representation for Computer Vision. In: Proceedings of the May 19-22, 1975, National Computer Conference and Exposition. AFIPS 75. New York, NY, USA: Association for Computing Machinery; 1975. p. 589–596. Available from: https://doi.org/10.1145/1499949.1500071.

33. Flannery BP, Press WH, Teukolsky SA, Vetterling W. Numerical recipes in C: The Art of Scientific Computing. Press Syndicate of the University of Cambridge, New York. 1992;.

34. Strang G. Linear algebra and its applications. 3rd ed. Philadelphia, PA: Saunders; 1988.

35. Misra P, Chaudhuri M. Performance Evaluation of Concurrent Lock-free Data Structures on GPUs. In: IEEE International Conference on Parallel and Distributed Systems. IEEE Computer Society; 2012. p. 53–60.

36. AIM@SHAPE. The Shape Repository;. Available from: http://visionair.ge.imati.cnr.it/ontologies/shapes/.

37. Cignoni P, Callieri M, Corsini M, Dellepiane M, Ganovelli F, Ranzuglia G. MeshLab: an Open-Source Mesh Processing Tool. In: Scarano V, Chiara RD, Erra U, editors. Eurographics Italian Chapter Conference. Eurographics; 2008. p. 129–136.

38. Silva S, Madeira J, Santos BS. PolyMeCo—An integrated environment for polygonal mesh analysis and comparison. Computers & Graphics. 2009; 33(2):181–191. https://doi.org/10.1016/j.cag.2008.09.014

39. Roy M, Foufou S, Truchetet F. Mesh comparison using attribute deviation metric. International Journal of Image and Graphics. 2004; 04(01):127–140. https://doi.org/10.1142/S0219467804001324