

Technical University of Denmark



Cache-mesh, a Dynamics Data Structure for Performance Optimization

Nguyen Trung, Tuan; Dahl, Vedrana Andersen; Bærentzen, Jakob Andreas

Published in:
Procedia Engineering

Link to article, DOI:
[10.1016/j.proeng.2017.09.807](https://doi.org/10.1016/j.proeng.2017.09.807)

Publication date:
2017

Document Version
Publisher's PDF, also known as Version of record

[Link back to DTU Orbit](#)

Citation (APA):
Nguyen, T. T., Dahl, V. A., & Bærentzen, J. A. (2017). Cache-mesh, a Dynamics Data Structure for Performance Optimization. *Procedia Engineering*, 203, 193-205. DOI: 10.1016/j.proeng.2017.09.807

DTU Library

Technical Information Center of Denmark

General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

26th International Meshing Roundtable, IMR26, 18-21 September 2017, Barcelona, Spain

Cache-mesh, a Dynamics Data Structure for Performance Optimization

Tuan T. Nguyen^{a,*}, Vedrana A. Dahl^a, J. Andreas Bærentzen^a

^aTechnical University of Denmark, Anker Engelunds Vej 1, 2800 Kgs. Lyngby, Denmark

Abstract

This paper proposes the cache-mesh, a dynamic mesh data structure in 3D that allows modifications of stored topological relations effortlessly. The cache-mesh can adapt to arbitrary problems and provide fast retrieval to the most-referred-to topological relations. This adaptation requires trivial extra effort in implementation with the cache-mesh, whereas it may require tremendous effort using traditional meshes. The cache-mesh also gives a further boost to the performance with parallel mesh processing by caching the partition of the mesh into independent sets. This is an additional advantage of the cache-mesh, and the extra work for caching is also trivial. Though it appears that it takes effort for initial implementation, building the cache-mesh is comparable to a traditional mesh in terms of implementation.

© 2017 The Authors. Published by Elsevier Ltd.

Peer-review under responsibility of the scientific committee of the 26th International Meshing Roundtable.

Keywords: cache, geometry processing, dynamics structure, data structure, performance optimization

1. Introduction

3D meshes are an essential part of computational geometry processing, with applications in finite element methods, deformable bodies, fluid simulation, topology optimization, visualization, etc. For mesh processing, we are generally concerned about mesh quality, memory usage and performance. All of these factors are important, but for dynamic meshes where geometry and topology change frequently, performance is often a very high priority. This paper discusses performance optimizations for 3D meshes which do not compromise the quality and add just a small increase in memory usage.

A mesh consists of entities, e.g. vertices, edges, faces and tetrahedra in a tetrahedral mesh. It also consists of a topology that describes the relations between these elements. There are many different types of topological relations, although, typically, only a few such relations are stored to ensure memory efficiency and simplify the implementation. The other relations must be calculated, and the complexity of this operation depends on the number of indirections from one mesh entity to the set of entities that we need. It can be time-consuming, especially in high-dimension meshes. To be more specific, we define $R(k)$ as the mapping from an entity k to the set of entities R that we seek. An analysis of the performance of $R(k)$ in ten common mesh data structures can be found in [20]. Fig. 1 shows

* Corresponding author. Tel.: +45-4525-5984

E-mail address: tntr@dtu.dk

three examples from [20], namely F1, F3 and R1, and Tab. 1 shows the memory operation counts (storage, retrieval, assignment and comparison of topological relations) of topology retrieval for these three meshes. We can see that the memory operation counts are significantly different, even though we only consider a subset of relations. In general cases, topology relations are more complex and memory operations vary even more.

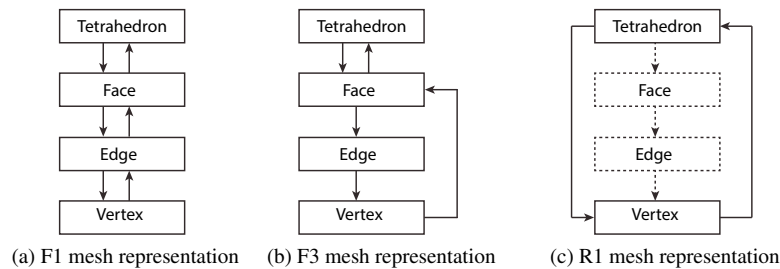


Fig. 1. Three data structures analysed in [20]. The arrow denotes stored adjacent relations. (a) Full one-level upward and downward adjacencies. (b) Full downward adjacencies and upward adjacencies from vertices to faces and faces to tetrahedra. (c) Reduced representation: Only vertices, tetrahedron and their adjacent relations in both ways are stored.

Table 1. Comparison of topology retrieval in memory operation count for three mesh data structures in Fig. 1, adopted from [20]. The notations V, E, F, T denote vertex, edge, face and tetrahedron, respectively.

Type	F(T)	E(T)	V(T)	T(F)	E(F)	V(F)	T(E)	F(E)	V(E)	T(V)	F(V)	E(V)
F1	4	36	30	2	3	13	50	5	2	619	399	14
F3	4	36	30	2	3	13	297	252	2	360	35	840
R1	72	58	4	302	24	3	214	721	2	23	3462	1969

We discuss the topology retrieval because it is one of the main factors that affect the performance. As mesh processing mainly deals with the topology, including inquiries, evaluation and modification, it leads to a large number of references to topological relations. Consequently, the data structure should provide fast retrieval of the most commonly used relations for a given problem. Unfortunately, this is not a straightforward process because the statistics of the topology retrievals differ greatly from one problem to another.

Generally, one avoids modifying the data structure of a mesh, because adding or removing one type of topological relation requires the modifications in all topological functions (functions that include topology changes). For this reason, the common approach for performance optimization is to select a suitable data structure, and this includes two steps: 1) profile the data references in the problem; and 2) then select a suitable data structure based on that statistic. There are limitations in this approach. First, because we are looking for something very specific, it may not exist, and often creating a bespoke data structure is not an option. Second, in order to profile the data reference, we need the problem to be implemented in advance. This means we solve and profile the problem with a pilot data structure, and then exchange it with an optimized data structure in the final phase. However, the extra effort for this replacement is also not negligible.

This paper proposes the cache-mesh; a dynamic mesh data structure that can be modified with trivial effort. The cache-mesh consists of a core mesh and a cache layer that stores extra topological relations. Our compelling advantage is the ability to change the types of stored topological relations at minor additional cost, which makes data structure optimization straightforward. Furthermore, the use of caching does not add much complexity if we decide to cache one more entity type.

Finally, yet importantly, we can store uncommon data, which rarely appears in a mesh data structure. We will demonstrate an example in which caching entity attributes helps in enabling parallel mesh processing for 3D meshes. Though the cache-mesh may sound complicated to implement, it can be achieved with little effort using an existing mesh framework as the core mesh.

2. Related work

2.1. Mesh data structures

Meshes differ mainly in the types of topological relations they store. Based on this difference, the authors in [15] categorize general mesh data structures into three groups: incidence-based, which stores incident relations, including boundary and co-boundary entities; adjacency-based, which stores adjacent relations, including close, adjoining or neighboring entities; and edge-based, which considers edges primary and store their relations with other entities. However, possible data structures are much more varied, with many proposals from previous research. The reader can refer to [15] for a data structure for simplicial complex, [20] for a data structure of finite element analysis (FEA) applications, and [1] for common index-based representation. In this section, we will discuss some typical data structures that store different amounts of topological relations.

The most fundamental mesh structure consists of vertices, and the highest order elements with their vertices, as in Fig. 1(c). For example, in a tetrahedral mesh we store the vertices, the tetrahedra and a topology that defines four vertices for each tetrahedron [37]. This data structure is compact, simple and suitable for finite element analysis which only queries the tetrahedra. Clearly, it is not optimal for problems that need topological relations, since these need to be inferred.

It is true that the higher the amount of stored topological relations, the better the performance [20]. However, this also raises the complexity in implementation as well as the memory usage. For this reason, mesh structures commonly stop short of storing all intermediate relations. Another example is the simplicial complex mesh [12], an incidence-based mesh that stores all boundaries and co-boundaries of the entities, as in Fig. 1(a). Such simplicial complex mesh can be considered the top-performance data structure in practice. Between this representation and the fundamental representation, there are several proposals that store different entities and topological relations: Primarily downward adjacencies [24], reduced incidence-based [14], only downward adjacency [9], etc.

Another approach is edge-centered representation that stores the edges and their relations to other entities. As vertices and faces are directly related to edges, this approach allows constant time adjacencies retrieval in 2D meshes. Several edge-centered data structures have been proposed: Winged-edge [3], half-edge [5], quad-edge [21], etc. For higher dimension, edge-based meshes are known for the advantages of oriented navigation, flexible modification, and the ability to generalize meshes in any dimension (e.g. Linear Cell Complex in CGAL [10,11]). Unfortunately, the performance becomes a huge drawback because there is no direct connection from the low-order entities to the high-order entities. Some researchers try to overcome this problem by combining edge-centered, face-centered structures with additional incidence relations [1,6,26]. Again, this raises the question of which relation sets should be stored.

In [4], the authors provide a memory comparison of different data structures. The comparison covers a wide range of different methods, although there is a lack of consideration for the problems that utilize the mesh. For this reason, they do not have criteria for measuring the overall performance and are not able to provide a performance comparison. In [20], the authors try to put the comparison into a real use case. They estimate the computation time using the statistic of topology references from one problem in the MEGA software [39]. One certain limitation is that their conclusion may not hold true for other problems.

One idea to make the data-structure optimization straightforward is a dynamic structure that can change its data effortlessly. This has appeared in the literature. In [27], the authors propose a multiple adjacency data set that allows users to choose between four different levels. The higher level uses more memory but improves the performance. The limitation of this approach is inflexibility, as users cannot choose the other data set they would like.

2.2. CPU cache and geometry processing

Caching is a technique that stores data for faster re-retrieval. The early idea of caching appeared in CPU architecture [13], when the gap between virtual memory access and register access increased. At software level, the CPU cache is utilized efficiently with compiler optimization [8] and cache-efficient algorithms [42].

Much research has considered CPU cache optimization for geometry processing with fixed meshes. The popular approach is to sort the data for optimal memory access. In [43–45], the authors propose a data layout for rendering and a boundary volume hierarchy for collision detection. In [7,22,29], CPU caches are mentioned for optimal rendering. In [35], the authors propose edge traverse for cache optimization, and also for rendering. Generally, a space-filling

curve [34,36] is utilized to store a cache-friendly layout. The limitation is that this can only be used for a fixed mesh. For a mesh that is dynamic, it is difficult to utilize such a low-level CPU cache.

For a dynamic mesh, caching appears at a higher level and serves as precomputed data. Examples are the precomputed polygon surface of NURB in [28] for collision detection; and the precomputed distance field in [41], also for collision detection. However, they do not include the ability to change the topology of the mesh. Not as much research contains cache for mesh with topological changes, as this raises the complexity in implementation. In summary, a CPU cache-efficient geometry-processing algorithm is only available for fixed meshes. For dynamic meshes, cache appears as precomputed data and is limited to its specific problem.

3. Some terminologies

This paper is concerned with 3D meshes, and our experiments utilize a tetrahedral mesh. As discussed above, a mesh \mathcal{M} contains entities linked by a topology. In geometry processing, we often refer to star and link in topology. For a single entity x , the star of x comprises all the entities that contain x . The closed star of x is the smallest subcomplex that contains the star of x . The link of x is the subtraction of the closed star and the star of x (Fig. 2).

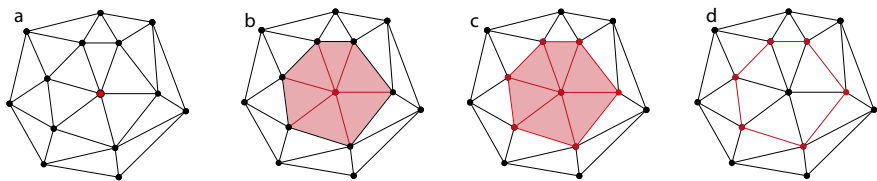


Fig. 2. A vertex (a), its star (b), its closed star (c), and its link (d)

Table 2. Complete list of first-order adjacencies (Ordered edges and ordered faces mean the entities are stored by a specific orientation)

Entity	First-order adjacencies
Vertex (V)	faces, edges, tetrahedra
Edge (E)	vertices, faces, tetrahedra
Face (F)	vertices, edges, ordered edges, tetrahedra
Tetrahedron (T)	vertices, edges, faces, ordered faces

Commonly used topological relations are incidence and adjacency relations. Incidences are boundary and co-boundary: boundary of an entity x is the set of one-level-lower entities that belong to x ; co-boundary of x is the set of one-level-higher entities whose boundaries contain x . Adjacencies are more general relations to entities which are close, adjoining or neighboring. For simplicity, meshes often store only first-order adjacencies (entities within the star of an entity, Tab. 2). In this paper, these commonly stored topological relations are called regular. In contrast, irregular relations are more complex and rarely appear in general mesh data structure.

4. The cache-mesh

In the following, we focus on the application of caching to 3D tetrahedral meshes. While e.g. triangle meshes might also benefit from caching, the utility is larger for 3D simplicial complexes since the number of possible relations is larger making it more expensive to maintain the full set of possible relations.

A mesh basically provides two functionalities: querying and modifying. Normally, query of un-stored topological relations leads to the computations of that data in every query, see Fig. 3(a). The cache-mesh has the same purposes but provides a cache query, where the data is kept for later access. The caching data can be any topology-related data, i.e. it may change when a topological event occurs. The cache-mesh is, in fact, a normal mesh with a cache layer that consists of the caching data and two functions: invalidation and storage, see Fig. 3(b). Sec. 4.1 will describe how the data is stored and how the storage function works; Sec. 4.2 will describe the invalidation process.

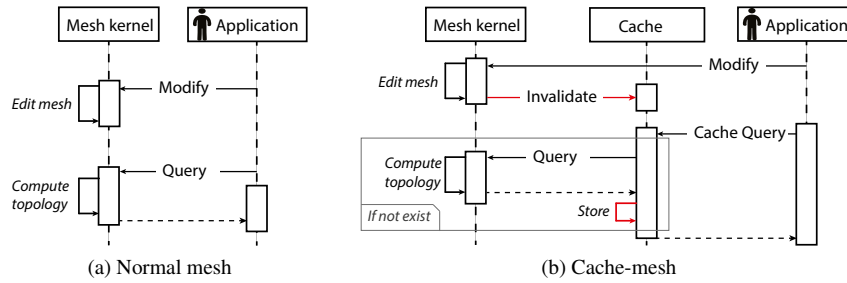


Fig. 3. Components of a normal mesh and the cache-mesh

4.1. The cache component

A mesh data structure often stores mesh entities in arrays for optimal memory and performance. Any mesh entity is accessed by its array index, and this index is used as the unique ID for the entity. The mesh kernel must be able to provide topology references of any type, and we call this topology-retrieval function: $get_data\langle data_type \rangle(ID)$. Here, $\langle data_type \rangle$ denotes a template which can change to any type of topological relations, and ID is the index of an entity. The $\langle data_type \rangle$ can also be a topology-related attribute that changes when the topology is modified.

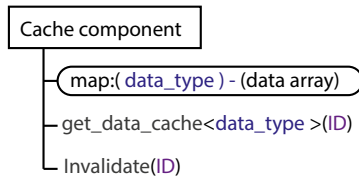


Fig. 4. Data structure of the cache component

Algorithm 1: get_data_cache function

Input: Mesh M , cache C , $data_type$, ID

```

1 if  $C.map[data\_type]$  does not exist then
2   | Allocate  $C.map[data\_type]$ 
3 if  $C.map[data\_type][ID]$  does not exist then
4   |  $C.map[data\_type][ID] =$ 
      |  $M.get\_data\langle data\_type \rangle(ID)$ 
Output:  $C.map[data\_type][ID]$ 

```

The cache component is an independent component, and it replaces some get_data functions. The structure of the cache component is shown in Fig. 4. We utilize a map to manage the caching relations, and the map is empty in the beginning. If a new type of topological relation is requested, we allocate a corresponding $data_type$ and data array in the map.

The cache component has two functions: $get_data_cache()$ that replaces the normal $get_data()$ function; and $invalidate()$ to clean the outdated cache. As mentioned above, the get_data_cache function is an upgrade from the normal get_data function as shown in Alg. 1. The reference locality (miss/hit rate of the second if-then instruction) determines the performance gain, and it will be analyzed in Sec. 5.3.

The effort to implement this function is trivial as only little code is added to give the priority to the cached relations and to call the get_data function if the data has not been cached. In fact, get_data_cache functions differ only in the data type of the caching relations. By utilizing template, we only need one template function, and other get_data functions can be upgraded to get_data_cache with one line of code. A sample implementation of the cache component in C++ can be found in Github [33].

4.2. Invalidate cache in common meshing procedures

Cache validation aims to maintain the accuracy of the caching topological relations. When a topology event occurs and changes the mesh, the cached relations may be different from the true data, hence the invalid cached data must be removed. The cache invalidation process is to find the affected entities in all topological functions. Though it may sound as if cache invalidation involves a lot of work, most geometry algorithms are the combinations of a few basic procedures, hence we only need to invalidate the cache in these basic functions.

To find the affected entities, we find the affected tetrahedra in the meshing procedure. The affected entities are all entities inside these tetrahedra (including vertices, edges, faces and the tetrahedra themselves). We limit the caching

topology relations to the closed star of the key entity. Note that knowing the deleted/added entities is required in these procedures in order to update the stored relations in the kernel mesh, we can utilize this information to find the affected tetrahedra.

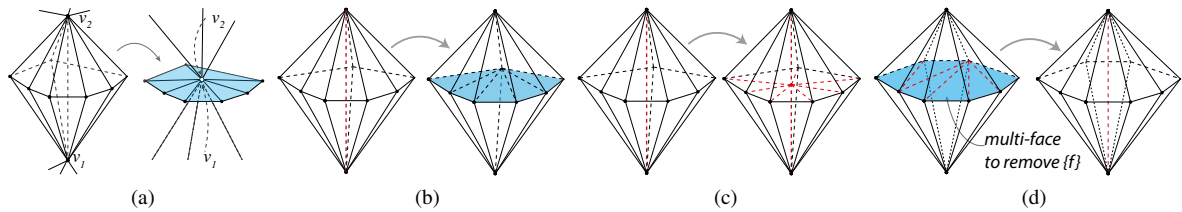


Fig. 5. Common topological algorithms. (a) Collapse edge by merging vertices. (b) Collapse edge with minimal topological change. (c) Edge split. (d) Multi-face removal.

Common and basic 3D meshing procedures are edge collapse, edge split, and face flip [18,25,32]. The 3D face flip algorithm is the generalization of the 2D edge flip, and it is called the multi-face removal algorithm [40]. Demonstrations of these algorithms are shown in Fig. 5. The affected tetrahedra in the four algorithms in Fig. 5 are described in Tab. 3.

Table 3. Affected tetrahedra of common meshing algorithm in Fig. 5

Procedure	Merging vertices	Collapse edge	Edge split	Multi-face removal
Affected tetrahedra	Tetrahedra in the intersection of stars of v_1 and v_2	Tetrahedra in the star of the collapsing edge	Tetrahedra in the star of the splitting edge	Co-boundary tetrahedra of the faces that are being removed

4.3. Utilizing the cache-mesh and profiling the references of topological relations

The utilization of the cache-mesh depends on whether the user-implemented functions contain topological changes in the mesh. The two situations will be described in two examples in Sec. 5.2 and Sec. 5.3. Generally, we follow three steps: integrate the cache component to the kernel mesh (omit this step if the cache-mesh is already integrated); profile the topology references; and update the *get_data* functions of bottleneck topological relations to *get_data_cache*.

For profiling, our criteria are the number of calls and the computation time of the topology retrieval functions – one example is in Tab. 6. Of these two, the computation time has greater influence on deciding which topological relations are the bottleneck.

Another criterion for profiling is theoretical analysis of the algorithm, which is efficient for finding irregular topological relations references. These irregular relations should be within the closed star of the key entity since we limit the cache invalidation in the closed star.

5. Some practical examples

This section describes three examples of using the cache-mesh: quantitative information measurements (Sec. 5.2), where no topological event occurs; mesh processing (Sec. 5.3), where topological events occur; and parallel mesh processing (Sec. 5.4). This section will only shows methods and results, explanation of the results and discussion will be carried out in Sec. 6.

5.1. Experiment set-up

In all experiments, we utilize a tetrahedral mesh representing an armadillo. The specifications of the mesh are shown in Tab. 4. The kernel mesh is a simplicial complex mesh [12] that stores boundary and co-boundary of the entities as shown in Fig. 1(a). When needed, the deformation and topology changes of the test object are handled by

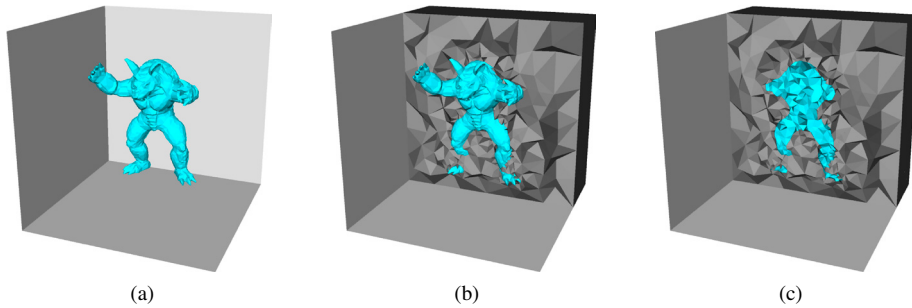


Fig. 6. The Deformable Simplicial Complex (DSC) labels the tetrahedra to represent different objects. (a) The DSC interface, which represents an armadillo, in its domain. (b) The interface and a cross-section of the DSC domain. (c) A cross-section of the whole mesh.

Deformable Simplicial Complex (DSC [31]), an explicit interface tracking method. See Fig. 6 for armadillo in DSC domain. In Sec. 5.3 we optimize performance of DSC, and there we describe the DSC algorithm in more detail. We use rotation and averaging motion for the experiments, see Fig. 7.

The specification of the mesh is shown in Tab. 4. The specification of the experimenting computer: Scientific Linux release 6.4 (Carbon) with 4 cores 2.5GHz CPU; 16GB of RAM; CPU cache: 32K L1d-cache, 32K L1i-cache, 256K L2-cache and 30M L3-cache. We use the gcc 7.1 compiler with `-Wall -O3` flags. We utilize “Oracle studio 12.5 performance analyzer” for all measurements. We perform each experiment three times, and we always observe consistent results.

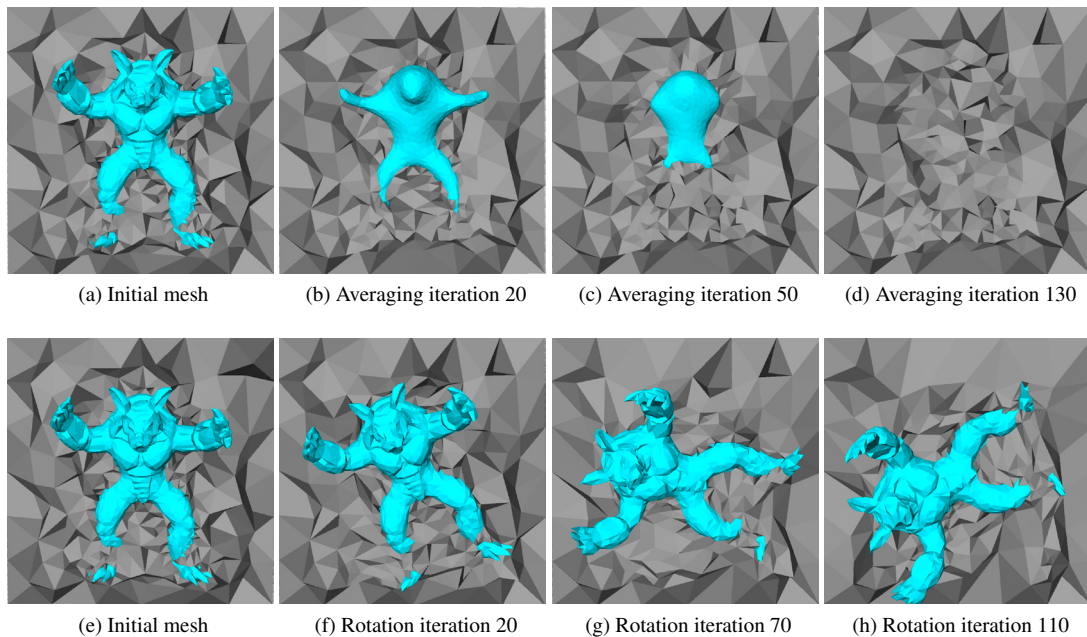


Fig. 7. Averaging motion (top row) and rotation motion (bottom row). The figures show a cross-section of the tetrahedral mesh that represent an armadillo.

5.2. Example 1: Extracting measurements with the cache-mesh

We first demonstrates optimizing the performance of operations which do not involve topological changes. In this example we measure quantitative information from the mesh: object volume; surface curvature; gradient of

Table 4. Mesh specification of the armadillo model. Interface vertices and interface faces are the entities on the armadillo surface.

# vertices	# interface vertices	# edges	# faces	# interface faces	# tetrahedra
4,872	2,164	33,738	57,518	4,324	28,651

volume with respect to the displacement of the interface vertices; and energy change due to vertices displacements (the energy requires the information of the region around the vertex). These 4 measures are collected in between the (not-optimized) DSC-handled deformations of the object under an averaging motion and a rotation motion, resulting in 8 experiments.

To utilize the cache-mesh, we follow three steps in Alg. 2 (whereas there are four steps with traditional meshes in Alg. 3). In the Alg. 3, though the cache-mesh is not utilized, profiling is still necessary in order to select/build a data structure that fits the problem. In the current examples, the algorithms are known in advance, therefore we can analyze the set of the bottleneck topological relations theoretically. Tab. 5 shows the most-referred-to topological relations for each procedure. By storing these relations, we achieve up to 80% reduction in the computation time (Fig. 8).

To compare the effort in optimization between the cache-mesh and traditional meshes, we shall discuss the Alg. 2 and Alg. 3. Utilizing the cache-mesh requires mainly two steps, and they are similar to the first two steps of optimizing a traditional mesh. On the other hand, optimizing a traditional mesh requires two additional steps, and these steps are not trivial since implementing a mesh data structure includes a lot of work.

Table 5. Caching topological relations in the four measurements in the example 1

Quantitative information	Key entity (k)	Referent relations (R)
Compute volume	Tetrahedron	Vertices in the link
Compute surface curvature	Vertex	Interface vertices on the link
Compute volume gradient	Vertex	Interface edges on the link
Compute energy change	Vertex	Adjacent tetrahedra
	Vertex	Faces on the link

Algorithm 2: Optimize the performance for a problem with the cache-mesh

- 1 Solve the problem with the cache-mesh
- 2 Profile the topology references
- 3 Update *get_data* of bottleneck topological relations to *get_data_cache* /* trivial */

Algorithm 3: Select / build a traditional mesh that optimizes performance for a problem

- 1 Solve the problem with a pilot mesh
- 2 Profile the topology references
- 3 Select / build an optimal mesh
- 4 Replace the pilot mesh with the optimal mesh

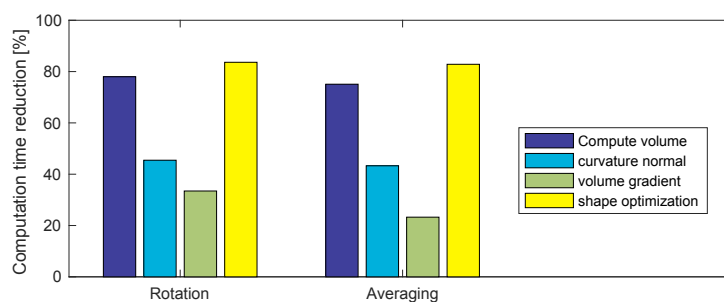


Fig. 8. Performance gain in quantitative information measurements by caching topological relations

As the statistic of the topology references is known in advance, one may question the necessity of the cache-mesh. In fact, the quantitative measurements vary, and it is not possible to store all topological relations that they

need. Furthermore, problems commonly do not utilize a single procedure but combine several procedures, and, in our experience, this fact leads to the differences in analytical profiling and practical profiling. To optimize the performance with a traditional mesh, we commonly need a pilot mesh (Alg. 3). On the other hand, the cache-mesh could adapt the cache to store different topological relation, and the effort for applying the cache is trivial. This case is useful for a public mesh framework that aims to serve many users with different requirements.

5.3. Example 2: Tuning mesh processing with the cache-mesh

This example demonstrates optimizing performance of meshing procedures that involve topological changes. Here we optimize the performance of the interface tracking method, Deformable Simplicial Complex (DSC). The DSC tracks the deformable interfaces by iteratively moving the interface vertices without making inverted tetrahedra. Between displacement phases, the DSC applies a mesh refinement procedure to maximize the mesh quality. The DSC algorithm is described in Alg. 4. The implementation of the DSC in C++ can be found in Github [2].

Algorithm 4: The DSC interface tracking

Input: Mesh M , displacements of interface vertices

```

1 begin
2   while Not all vertices are moved to their destination do
3     Move interface vertices as far as possible
4     Mesh refinement
5       Smooth
6       Topological edge removal
7       Multi-face removal
8       Short edge collapse
9       Long edge split

```

Because the DSC contains topological changes, the process to apply the cache-mesh is a bit different from Alg. 2 in the first step. To apply the cache-mesh for the DSC, we follow three steps:

1. *Update topological functions to invalidate the cache:* There are five functions in Alg. 4, fortunately they are the combinations of only three basic functions: edge split, edge collapse and multi-face removal. Finding affected tetrahedra in these three functions is described in Sec. 4.2.
2. *Profile the topology references:* We measure reference count and computation time (as described in Sec. 4.3) of first order adjacency retrieval functions (Tab. 6). We also analyze the DSC algorithm theoretically to find the irregular topology references (Tab. 7).
3. *Apply cache:* We upgrade the retrieval functions of the relations (*get_data*) from the above step to the cached version (*get_data.cache* in Alg. 1). The effort for this step is trivial.

Table 6. First-order adjacency references in the Deformable Simplicial Complex. The * denotes the stored topological relations in the kernel mesh.

Adjacent entities(key entities)	F(T)*	E(T)	V(T)	T(F)*	E(F)*	V(F)	T(E)	F(E)*	V(E)*	T(V)	F(V)	E(V)*
Queries count (millions)	37.8	0.25	16.7	62.5	80.4	75.7	0.35	24	165	0.95	0.81	5.6
Computation time (seconds)	14.6	0.8	78.6	20.7	28.5	136.6	0.1	8.4	57.2	39	6.7	2.1
Caching relations			✓			✓				✓		

Table 7. Irregular topology relations in DSC for caching. V, E, F denote vertex, edge and face.

Key entity (k)	V	V	E
Referent relations (R)	F in link	V in link	Sorted opposite E

The effort for invalidating cache is probably similar to adding one type of topological relations to the data structure, but we then add six types of topological relations with trivial extra effort. Fig. 9(a) show the results with performance

improvement and other factors including: the CPU cache miss in Fig. 9(b); memory usage overhead in Fig. 9(c); reuse statistic of the caching topological relations in Fig. 9(d); and the overhead of the computation time for the invalidating cache. The overhead includes the time for finding affected elements and the time for memory allocating, storing and retrieving the cache data; Fig. 9(e). Generally, we observe five times in performance gain with 50% memory overhead. We discuss these results in detail in Sec. 6.

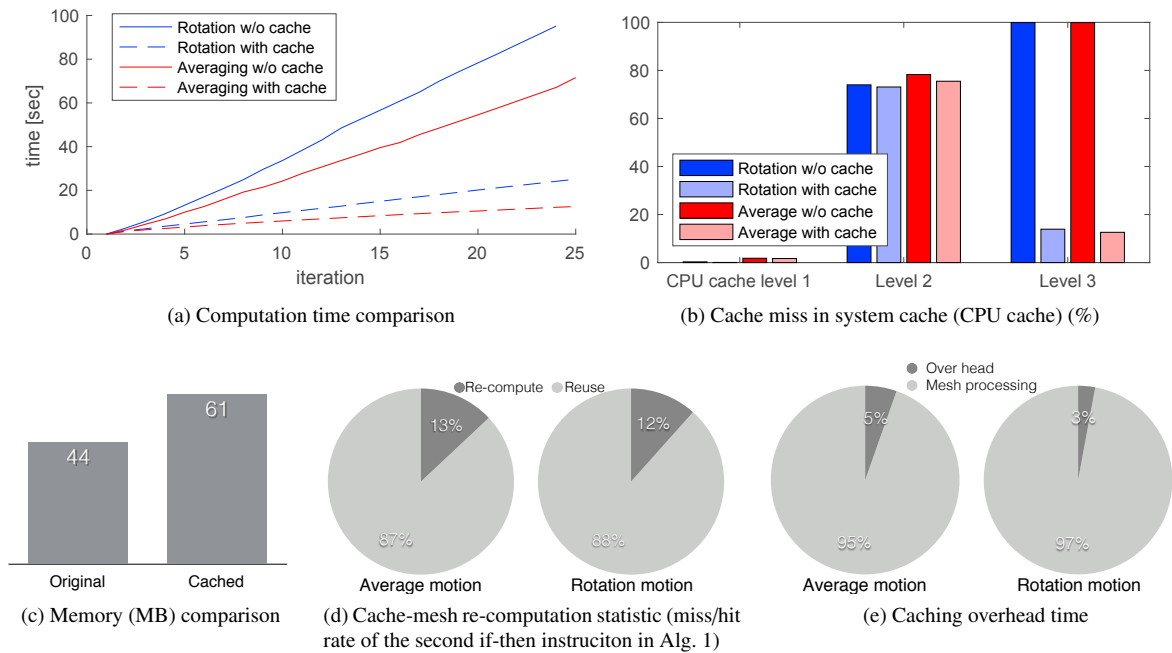


Fig. 9. Results for applying caching for the DSC. a) The computation time by iterations. b) The CPU cache statistic. c) Memory utilization. d) Re-computation and reuse of cached topological relations. e) The computation time for invalidating the cache.

5.4. Example 3: Parallel mesh processing with the cache-mesh

Parallel algorithms can be categorized into two groups: Distributed memory, in which a mesh is partitioned into parts that are independent in both memory and processing [38]; and shared memory, in which the partition only needs to be independent in processing. This section discusses a shared memory parallelism for mesh processing, and we employ the popular coloring method [17]. This method assigns different colors to independent entities (E.g. the independent colors of the vertices and the independent colors of the edges in Fig. 10(b) and 10(c)) so that entities with the same color can be processed in parallel. Though finding optimal colors is an NP-hard problem [19], independent sets can be defined by an efficient heuristic function [23,30], and it works well for 2D meshes. For general 3D dynamic meshes, it is still not possible to color the mesh iteratively as the time overhead for coloring is higher than the time for normal serial processing.

The cache-mesh enables coloring method for 3D dynamic meshes by caching the colors of the entities, and it reduces the overhead of coloring significantly. Define the color as an integer number, the algorithm of coloring method is shown in Alg. 5, and the algorithm for getting color of an entity is shown in Alg. 6. We apply caching for the colors for two meshing procedures: mesh smoothing [16] (the related entities are the neighbor vertices); and edge removal [40] (the related entities are the same as the affected edges for the cache invalidation in Fig. 5(b)).

The performance gain is shown in Fig. 11. Because of the limitation of the testing computer, we perform the experiment with up to four cores, and the computation time reduces up to 50%. About the scalability: the performance scales almost linear from one to two threads. However, the speed up reduces with more threads, and one reason is that full four cores could not be utilized by a single process. For a concrete number of parallel efficiency, we plan to experiment the cache-mesh with large data and better hardware in our future work.

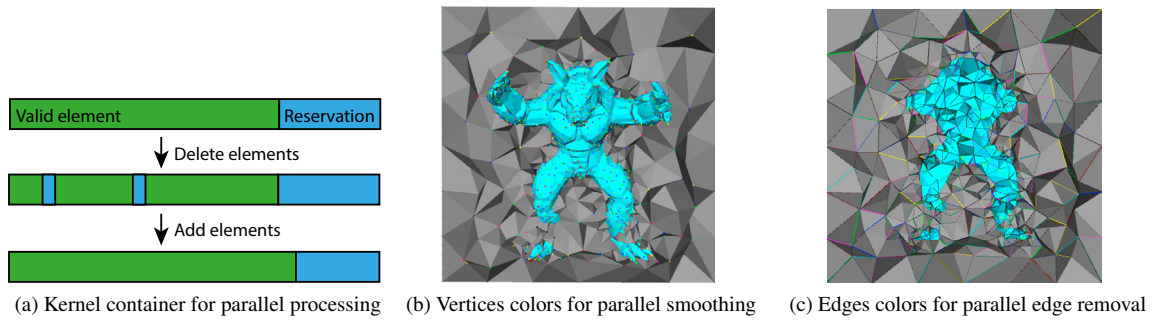
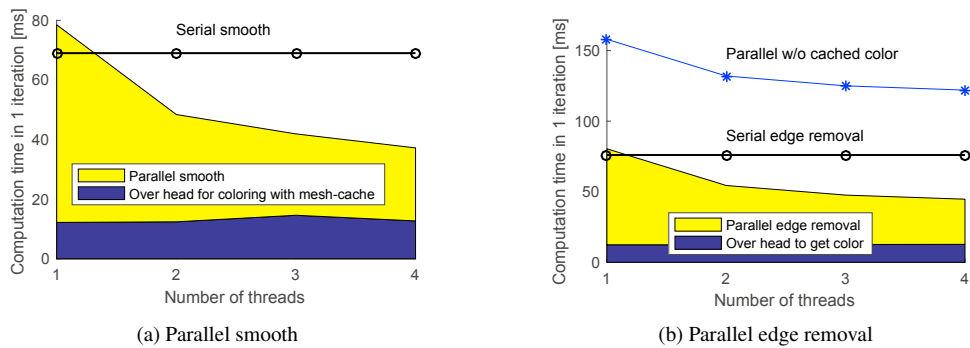


Fig. 10. Parallel mesh processing by coloring method

Fig. 11. Computation time of parallel edge removal and parallel smooth [ms]. In figure (a), we do not plot the computation time of parallel smooth w/o cached color as it is significantly larger ($\approx 451ms$) than that of serial smooth ($\approx 70ms$).**Algorithm 5: Coloring method**

```

1 Get colors of all processing entities
2  $S = \{s_i\}$ ,  $s_i$  is the set of entities with same color
3 for each  $s_i \in S$  do
4   | Start a thread to process  $s_i$ 
5 Wait for all threads to stop

```

Algorithm 6: Get color of entity

```

Input: Mesh  $\mathcal{M}$ , entity  $x$ 
1 array  $S = \text{Get color of related entities of } x$  /* The
   related entities depend on the actual
   procedures */
2 if  $S$  is empty then
3   | color  $\leftarrow 1$ 
4 else
5   | color  $\leftarrow$  min number which is not contained in  $S$ 
Output: color

```

Note that in order to make it possible for parallel processing, the mesh kernel must allow simultaneous modifications. For index-based mesh, the container is often an array, and it may not be able to add or remove elements at the same time. We slightly modify the array so that the container includes a reservation buffer, as described in Fig. 10(a). When we remove elements, the memory is not deleted but changes the state to *buffer*. When we add elements, the memory will be allocated in the reservation area. This modification is necessary for parallel mesh processing even with or without the cache-mesh, and, in fact, it is not rare in implementation of dynamic meshes.

6. Discussion and conclusion

Our experiments show that if we store the topological relations, which are the most time-consuming to compute, we can reduce the computation time by up to 80%. This fact affirms that topology retrieving consumes the major computation resource in our testing cases. However, the references of topological relations differ greatly from one

problem to another. Furthermore, they are not always regular for storing in a data structure. These two reasons make it difficult to optimize performance by fitting the data structure to the problem.

The cache-mesh provides a dynamic data structure that can adapt to arbitrary problems effortlessly. The advantages of the cache-mesh include:

- *Straightforward performance optimization:* Building an optimal mesh includes three steps with the cache-mesh, and the last step is trivial; whereas it takes four steps with traditional meshes, and none of them is trivial (Sec. 5.2).
- *Storing additional topological relations without increasing the complexity in implementation:* Since adding data is effortless with the cache-mesh, we can store additional topological relations for a better performance.
- *Storing irregular relations:* Some topological relations, e.g. opposite edges or some user-defined relations, are not stored in traditional meshes, otherwise the mesh data structures would be highly specific.
- *Parallel mesh processing:* Sec. 5.4 demonstrates the example where storing entity colors helps enable parallel mesh processing with minor extra effort.

The other facts of the cache-mesh, which can be considered advantages, are memory overhead and effort to implement from scratch that are comparable to a traditional mesh. The correlation of the cache-mesh and the CPU cache is also an interesting fact that we would like to discuss. In Fig. 9(b), the L3 cache seems to reflect the performance gain of the cache-mesh, which means it is not the instruction counter, but the latency in RAM has stronger influence on the computation time of the meshing procedures.

Regarding the performance gain, this depends on the portion of the affected entities during topological events. In our experiments of an deforming mesh, this portion is less than 15% (Fig. 9(d)), and caching reduces the computation time by around 80%. We also notice that the cache-mesh may not be as fast as traditional meshes that store the same topological relations because the caching data is not utilized inside the mesh kernel. Fortunately, this difference only manifests itself when the topological relations are recomputed (the 15% in our experiments). For dynamic meshes, the number of topological events should be controlled not only to reduce the number of entities affected by topology changes but also to increase the quality and accuracy of the mesh.

In conclusion, the cache-mesh provides a dynamic mesh data structure for convenient and effortless modification, which makes the performance optimization process straightforward. The cache-mesh boosts the performance in three ways: faster topological relations retrieval, as the relations are cached; possibility to store more relations, as adding topological relations does not raise the complexity; and parallel processing, which is difficult for traditional meshes.

For further study, we plan to experiment the cache-mesh with more types of kernel mesh as well as providing a stand-alone cache-mesh that has an optimal kernel mesh. We also intended to analyze the parallel efficiency with large data and better hardware.

References

- [1] T. J. Alumbaugh and X. Jiao. Compact Array-Based Mesh Data Structures. In *Proceedings of the 14th International Meshing Roundtable*, pages 485–503. Springer Berlin Heidelberg, 2005.
- [2] J. A. Bærentzen. Deformable Simplicial Complex. <https://github.com/janbaq/2D-DSC>.
- [3] B. G. Baumgart. Winged edge polyhedron representation. Technical report, DTIC Document, 1972.
- [4] M. W. Beall and M. S. Shephard. A general topology-based mesh data structure. *International Journal for Numerical Methods in Engineering*, 40(9):1573–1596, 1997.
- [5] S. Campagna, L. Kobbelt, and H.-P. Seidel. Directed edges-A scalable representation for triangle meshes. *Journal of Graphics tools*, 3(4):1–11, 1998.
- [6] W. Celes, G. H. Paulino, and R. Espinha. A compact adjacency-based topological data structure for finite element mesh representation. *International Journal for Numerical Methods in Engineering*, 64(11):1529–1556, 2005.
- [7] J. Chhugani and S. Kumar. Geometry engine optimization: cache friendly compressed representation of geometry. In *Proceedings of the 2007 symposium on Interactive 3D graphics and games*, pages 9–16. ACM, 2007.
- [8] S. Coleman and K. S. McKinley. Tile size selection using cache organization and data layout. *ACM SIGPLAN Notices*, 30(6):279–290, 1995.
- [9] S. D. Connell and D. G. Holmes. Three-dimensional unstructured adaptive multigrid scheme for the Euler equations. *AIAA Journal*, 32(8):1626–1632, 1994.
- [10] G. Damiand. Combinatorial Maps. In *{CGAL} User and Reference Manual*. CGAL Editorial Board, 4.9 edition, 2016.
- [11] G. Damiand. Linear Cell Complex. In *{CGAL} User and Reference Manual*. CGAL Editorial Board, 4.9 edition, 2016.

- [12] L. De Floriani, A. Hui, D. Panozzo, and D. Canino. A dimension-independent data structure for simplicial complexes. In *Proceedings of the 19th International Meshing Roundtable*, pages 403–420. Springer, 2010.
- [13] P. J. Denning. Virtual Memory. *ACM Computing Surveys (CSUR)*, 2(3):153–189, 1970.
- [14] M. Field, M. Field, R. Biswas, and R. C. Strawn. A new procedure for dynamic adaption of three-dimensional unstructured grids. *Applied Numerical Mathematics*, 13(6):437–452, 1994.
- [15] L. D. Floriani and A. Hui. Data Structures for Simplicial Complexes: An Analysis And A Comparison. *Symposium on Geometry Processing*, 2005.
- [16] L. Freitag, M. Jones, and P. Plassmann. An Efficient Parallel Algorithm for Mesh Smoothing. *INTERNATIONAL MESHING ROUNDTABLE*, pages 1–14, 1995.
- [17] L. Freitag, M. Jones, and P. Plassmann. A Parallel Algorithm for Mesh Smoothing. *SIAM Journal on Scientific Computing*, 20(6):2023–2040, jan 1999.
- [18] L. Freitag and C. Ollivier-Gooch. Tetrahedral Mesh improvement using swaping and smoothing. *International Journal for Numerical ...*, 40(November 1996):3979–4002, 1997.
- [19] M. Garey and D. Johnson. *Computer and intractability*. W. H. Freeman, New York, 1979.
- [20] R. V. Garimella. Mesh data structure selection for mesh generation and FEA applications. *International Journal for Numerical Methods in Engineering*, 55(4):451–478, 2002.
- [21] L. Guibas and J. Stolfi. Primitives for the manipulation of general subdivisions and the computation of Voronoi. *ACM transactions on graphics (TOG)*, 4(2):74–123, 1985.
- [22] M. Jan and L. Kobbelt. OpenFlipper : An Open Source Geometry Processing and Rendering Framework. In *International Conference on Curves and Surfaces*, pages 488–500, 2012.
- [23] M. T. Jones and P. E. Plassmann. A Parallel Graph Coloring Heuristic. *SIAM Journal on Scientific Computing*, 14(3):654–669, may 1993.
- [24] Y. KALLINDERIS and P. VIJAYAN. Adaptive refinement-coarsening scheme for three-dimensional unstructured meshes. *AIAA Journal*, 31(8):1440–1447, 1993.
- [25] B. M. Klingner and J. R. Shewchuk. Aggressive tetrahedral mesh improvement. *Proceedings of the 16th International Meshing Roundtable, IMR 2007*, pages 3–23, 2008.
- [26] M. Kremer, D. Bommers, and L. Kobbelt. Open VolumeMesh—A Versatile Index-Based Data Structure for 3D Polytopal Complexes. *Proceedings of the 21st International Meshing Roundtable*, pages 531–548, 2013.
- [27] M. Lage, T. Lewiner, H. Lopes, and L. Velho. CHF: a scalable topological data structure for tetrahedral meshes. *Proceedings of the XVIII Brazilian Symposium on Computer Graphics and Image Processing (SIBGRAPI'05)*, pages 1–6, 2005.
- [28] R. W. Lau, R. W. Lau, O. Chan, M. Luk, M. Luk, F. W. Li, and F. W. Li. A collision detection framework for deformable objects. *Proceedings of the ACM symposium on Virtual reality software and technology - VRST '02*, page 113, 2002.
- [29] J. Levenberg. Fast view-dependent level-of-detail rendering using cached geometry. *VIS02 IEEE Visualization 2002*, pages 259–265, 2002.
- [30] M. Luby. A simple parallel algorithm for the maximal independent set problem. *Annual ACM Symposium on Theory of Computing*, page 1, 1985.
- [31] M. K. Misztal and J. A. Bærentzen. Topology-adaptive interface tracking using the deformable simplicial complex. *ACM Transactions on Graphics*, 31(3):1–12, may 2012.
- [32] M. K. Misztal, J. A. Bærentzen, F. Anton, and K. Erleben. Tetrahedral mesh improvement using multi-face retriangulation. *Proceedings of the 18th International Meshing Roundtable, IMR 2009*, pages 539–555, 2009.
- [33] T. Nguyen, V. A. Dahl, and J. A. Bærentzen. Template code for mesh cache: github.com/tuannt8/cache-template, 2016.
- [34] H. Sagan. *Space-filling curves*. Springer Science & Business Media, 2012.
- [35] P. V. Sander, D. Nehab, E. Chlamtac, and H. Hoppe. Efficient traversal of mesh edges using adjacency primitives. *ACM Transactions on Graphics*, 27(5):1, 2008.
- [36] S. P. Sastry, E. Kultursay, S. M. Shontz, and M. T. Kandemir. Improved cache utilization and preconditioner efficiency through use of a space-filling curve mesh element- and vertex-reordering technique. *Engineering with Computers*, 30(4):535–547, 2014.
- [37] W. Schroeder, K. Martin, and B. Lorensen. *The Visualization Toolkit*. Kitware, 4 edition, 2006.
- [38] E. S. Seol and M. S. Shephard. Efficient distributed mesh data structure for parallel automated adaptive analysis. *Engineering with Computers*, 22(3-4):197–213, 2006.
- [39] M. S. Shephard. Meshing Environment for geometry based analysis. *International Journal of Numerical Methods in Engineering*, 47(1-3):169–190, 2000.
- [40] J. Shewchuk. Two discrete optimization algorithms for the topological improvement of tetrahedral meshes. *Unpublished manuscript*, pages 1–11, 2002.
- [41] M. Teschner, S. Kimmerle, B. Heidelberger, G. Zachmann, L. Raghupathi, A. Fuhrmann, M. P. Cani, F. Faure, N. Magnenat-Thalmann, W. Strasser, and P. Volino. Collision detection for deformable objects. *Computer Graphics Forum*, 24(1):61–81, 2005.
- [42] J. S. Vitter. External Memory Algorithms and Data Structures. *ACM Computing Surveys (CSUR)*, 33(2):209–271, 2001.
- [43] S. E. Yoon and P. Lindstrom. Mesh layouts for block-based caches. *IEEE Transactions on Visualization and Computer Graphics*, 12(5):1213–1220, 2006.
- [44] S.-E. Yoon, P. Lindstrom, V. Pascucci, and D. Manocha. Cache-oblivious mesh layouts. *ACM Transactions on Graphics*, 24(3):886, 2005.
- [45] S. E. Yoon and D. Manocha. Cache-efficient layouts of bounding volume hierarchies. *Computer Graphics Forum*, 25(3):507–516, 2006.