Engineering with Computers (2024) 40:867–883
https://doi.org/10.1007/s00366-023-01826-7

ORIGINAL ARTICLE

Memory-efficient boundary-preserving tetrahedralization of large three-dimensional meshes

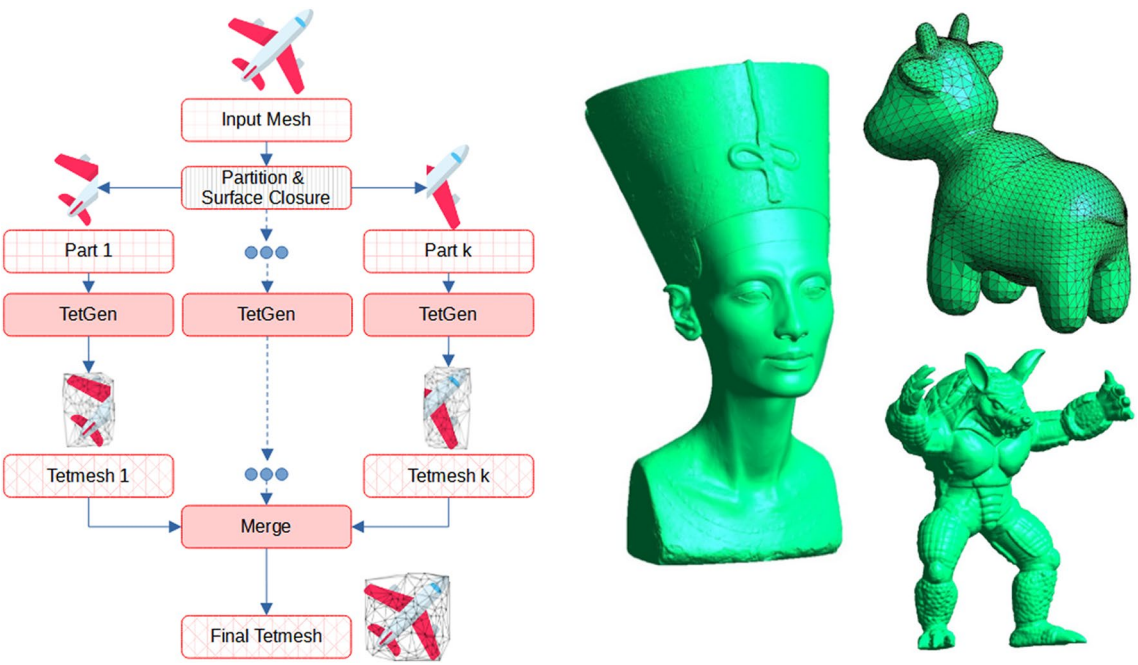
Ziya Erkoç[1] · Uğur Güdükbay[1] · Hang Si[2]

Received: 15 July 2022 / Accepted: 19 April 2023 / Published online: 9 May 2023
© The Author(s), under exclusive licence to Springer-Verlag London Ltd., part of Springer Nature 2023

Abstract

We propose a divide-and-conquer algorithm to tetrahedralize three-dimensional meshes in a boundary-preserving fashion. It consists of three stages: *Input Partitioning*, *Surface Closure*, and *Merge*. We first partition the input into several pieces to reduce the problem size. We apply 2D Triangulation to close the open boundaries to make new pieces watertight. Each piece is then sent to *TetGen*, a Delaunay-based tetrahedral mesh generator tool that forms the basis for our implementation. We finally merge each tetrahedral mesh to calculate the final solution. In addition, we apply post-processing to remove the vertices we introduced during the input partitioning stage to preserve the input triangles. The benefit of our approach is that it can reduce peak memory usage or increase the speed of the process. It can even tetrahedralize meshes that *TetGen* cannot do due to the peak memory requirement.

Graphical abstract



Keywords Boundary-preserving tetrahedralization · Parallelization · Three-dimensional mesh · Divide-and-conquer · Memory efficiency

✉ Uğur Güdükbay
gudukbay@cs.bilkent.edu.tr

Extended author information available on the last page of the article

1 Introduction

Tetrahedral meshes are widely used in many areas, including bioengineering [1], biomechanics [2, 3], computational fluid dynamics [4], computer graphics, and animation, especially the simulation of deformable bodies, including fracture and incision simulations [5–7], mechanical simulations such as turbomachinery flow [8], medical applications, such as medical device design [9, 10], medical image analysis [11], and soft tissue simulations [12]. Tetrahedral meshes are mainly used for the discretization of continuous materials and objects for finite element model (FEM) simulations [13] because they fit well for complex geometry [14].

Constrained tetrahedralization algorithms take a three-dimensional (3D) surface or triangular mesh and vertices as input and tetrahedralize the inside of the input mesh. The surface triangles that form the boundary of the input mesh are the constraint faces that describe the boundary of the generated tetrahedralization [15]. The constrainedness property makes these algorithms valuable for various applications such as FEM simulations [14] and acceleration structures for raytracing [16]. To speed up the ray-surface intersection calculations in raytracing, an algorithm for rendering 3D scenes, the 3D scene can be tetrahedralized constrainedly where input geometry components such as faces, line segments, and points are preserved. In the Finite Element Method (FEM), a 3D object can be tetrahedralized constrainedly to apply physics experiments such as a stress test for a suspension bridge [14]. Such applications require the discretization of 3D surface meshes by tetrahedralization algorithms in a constrained (boundary-preserving) fashion.

The 3D objects or surface meshes used in ray tracing and FEM can have millions of vertices; the constrained tetrahedralization algorithms must handle massive objects and scenes. However, current constrained tetrahedralization algorithms seem to fall short of tetrahedralizing large scenes in a constrained fashion when the memory requirement exceeds the available memory. In addition, the present methods might take a significant amount of time to execute.

We propose a divide-and-conquer boundary-preserving tetrahedral mesh generation algorithm to reduce execution time or decrease memory usage. The algorithm consists of *Partitioning*, *Surface Closure*, and *Merge* steps. We divide the object into k pieces at the *Partitioning* stage, where k is a parameter given by the user. As a result of that stage, we obtain k meshes with open boundaries, which we need to close at the *Surface Closure* stage. During this stage, we use the 2D Constrained Triangulation Algorithm as a sub-procedure to triangulate the open boundary, and then we refine the triangulation to increase the quality. Finally, we concatenate all tetrahedral mesh objects at the *Merge* step. At this stage, we also find missing neighbor relations between boundary tetrahedra at all pieces.

We implemented two modes for our algorithm, which are *Parallel Processing* and *Memory Requirement Reduction*. The first mode reduces execution time using multi-threading, and the second mode reduces memory requirement using a single thread by utilizing files to store partial results.

The outcome of the experiments implies that our algorithm can either consume less memory than *TetGen* or execute faster than it, depending on the chosen mode. Our algorithm might be applied as an alternative tetrahedral mesh generation tool when *TetGen* cannot process large objects due to their vast memory consumption or speed up the process. Although we may generate some non-Delaunay tetrahedra, we could generate meshes of better or similar quality compared to *TetGen*. In this regard, our divide-and-conquer algorithm improves the capabilities of *TetGen*. Our original input division procedures help us tetrahedralize massive objects that a sequential algorithm cannot do. However, our memory-efficient process may be slightly slower than *TetGen*. Figure 1 shows the proposed algorithmic framework and example tetrahedralized meshes generated using our implementation. Our reference implementation is available on the GitHub repository https://github.com/Rgtemze/MemoryEfficientTetMeshGen.

The rest of the paper is organized as follows. Section 2 discusses related work on triangulation and tetrahedralization algorithms. Section 3 describes our approach by explaining how we divide the input mesh, triangulate the open boundary of each part by a surface closure algorithm, and merge the sub-problems. Section 4 talks about the two modes of our algorithm in detail. Section 5 presents the experimental evaluation of the proposed approach in terms of execution time, memory usage, and mesh quality. Finally, Sect. 6 gives conclusions and future research directions.

2 Related works

We describe the related studies on sequential CDT, parallel Delaunay triangulation algorithms, and input partitioning methods.

2.1 Sequential CDT

Si put forth a constrained Delaunay tetrahedralization (CDT) algorithm, called *TetGen*, which generates high-quality tetrahedra [17]. Because *TetGen* does not apply problem partitioning, it may not scale well to large objects due to large memory requirements. Our algorithm is a divide-and-conquer extension of *TetGen*. In *TetGen*, CDT is defined as a

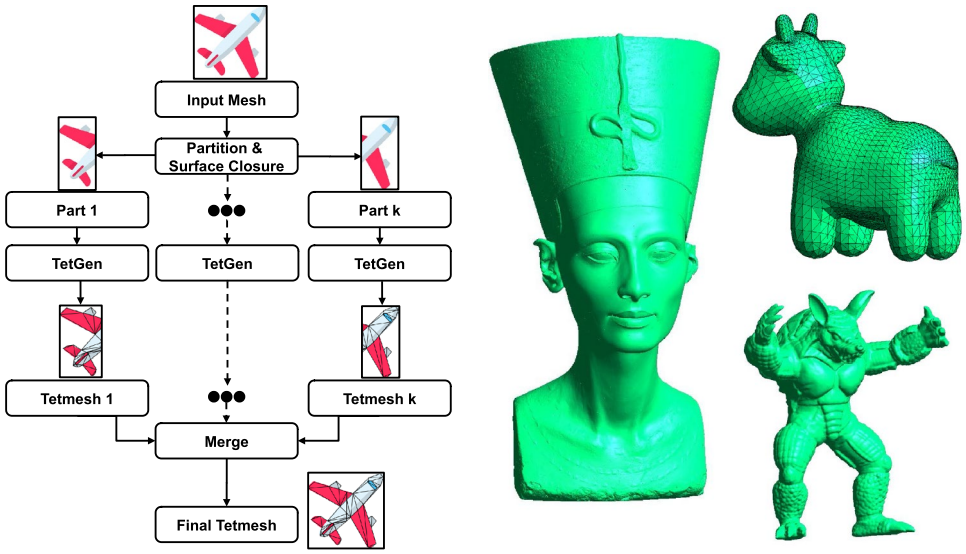


Fig. 1 The proposed boundary-preserving tetrahedralization framework and example tetrahedral meshes generated using our implementation

tetrahedralization T, where every face in T that is not part of input mesh is *locally Delaunay*. A face is *locally Delaunay* if it is part of a single tetrahedron or part of two tetrahedra t_1 and t_2, but its circumsphere does not contain any other point from t_1 and t_2. In addition, every triangle in the input must be a part of a tetrahedron in the output tetrahedralization. In our implementation, as a consequence of using a divide-and-conquer algorithm, the new triangles introduced around the cut region during the Surface Closure stage may not satisfy the *locally Delaunay* property. We apply refinement during the surface closure stage to improve the tetrahedral mesh quality around the cut regions. For this reason, we classify our algorithm as *boundary-preserving tetrahedralization* rather than CDT.

Hu et al. [18] developed a robust CDT mesh generator called *TetWild*. *TetWild* can tetrahedralize a wide range of objects. It does not make input assumptions and can even process non-manifold objects with self-intersections. Because our algorithm uses *TetGen* at the base, the input models we can handle are watertight non-self-intersecting meshes. *TetWild* is an implementation of an approximate constrained algorithm; it might not preserve the input perfectly. Still, it controls the input preservation level with a parameter. Moreover, Bridson and Doran developed *Quartet* to convert watertight meshes into a tetrahedral mesh that approximates the input mesh [19]. Both *TetWild* and *Quartet* are different from *TetGen* because *TetGen* can exactly constrain the input triangles instead of approximating them. *DelPSC* and *MMG3D* are also tetrahedral mesh generators [20, 21]. We selected *TetGen* as the basis of our implementation because it is robust and can preserve input features exactly as they are.

Chew proposes a two-dimensional sequential, high-quality constrained tetrahedralization algorithm [22]. The algorithm ensures that the internal angles of the triangles are between 30 and 120 degrees and edges are between h and $2\,h$ where h is a user-defined value. These properties are guaranteed to make sure triangulation contains near-equilateral triangles. The algorithm constantly computes triangulation and finds a Delaunay circle, a circumcircle of a Delaunay triangle, whose radius is greater than h. It then inserts the circle's center as the new point and recomputes the triangulation. *TetGen* also uses a similar approach by adding a circumcenter of the poor-quality tetrahedra to increase the mesh quality [17].

2.2 Parallel Delaunay triangulation

There are notable studies on parallel two-dimensional constrained Delaunay Triangulation (CDT) or parallel three-dimensional (3D) Delaunay triangulation (DT). However, we did not encounter parallel three-dimensional CDT or boundary-preserving tetrahedralization algorithms.

Chernikov and Chrisochoides's parallel 2D CDT algorithm uses a domain decomposition method called Medial Axis Domain Decomposition (MADD) [23]. After the decomposition, each subdomain is triangulated in parallel independently. The number of sub-domains created is much higher than the number of processors, and they use the Load Balancing library to assign sub-domains to processors in the most flexible way possible. In addition, they solve a graph partition problem to distribute sub-domains to processors so that each processor has a nearly equal amount of work. They use the message-passing model as their parallelization scheme instead of a shared-memory structure, and their implementation is based on Message-Passing-Interface (MPI) library. The domain decomposition algorithm presented here may work well in the 2D case, but in 3D, the existence of faces would make the problem more

complicated. Hence, we used mesh cutting to decompose the domain [24].

Coll and Guerrieri propose a 2D CDT algorithm that is parallelized using GPUs. Their algorithm consists of the Location, Insertion, Marking, and Flipping stages. The Location stage identifies triangles containing an uninserted point; the Insertion stage inserts points; the Marking stage marks the segments as valid or to be flipped (i.e., to eradicate non-Delaunayness or intersection); the Flipping stage flips the edges marked so. The algorithm is iterative and continues to run these four stages as long as edges and points are missing in the triangulation. These four stages are run one after another; they applied parallelization within each stage. In their implementation, the threads coordinate to avoid race conditions. For instance, when a thread is about to do a point insertion or edge-flip to a triangle, it informs the neighbor threads of that operation and who will be their new neighbor. It would be possible to extend this algorithm to a 3D algorithm. We adopted a more straightforward approach by creating independent parts and processing them individually. That way, we ensure no race condition, synchronization issue, or thread communication, helping to reduce parallelization overhead [25].

Blandford et al. [26] propose a parallel tetrahedralization algorithm that can be extended to an out-of-core algorithm. However, it is not a constrained tetrahedralization algorithm. They suggest that developing an out-of-core algorithm would allow large meshes to be tetrahedralized. Their parallel algorithm is based on the sequential incremental insertion algorithm. They use multi-threading and lock mechanisms to insert multiple vertices into the tetrahedral mesh simultaneously. Our algorithm differs from theirs because we use the divide-and-conquer paradigm to tetrahedralize. Their algorithm does not divide the input as in our case but works on a single mesh with multiple threads. Chernikov and Chrisochoides also generated quality tetrahedral meshes using the circumradius-to-shortest-edge ratio as the quality measure [27]. Their algorithm leverages multi-core processors through parallelization. Specifically, they focused on parallelizing the Delaunay refinement step to speed up the overall process.

Cignoni et al. [28] put forward a divide-and-conquer Delaunay triangulation algorithm, *DeWall*, that can triangulate point cloud data of any dimension. Although it is not implemented as a parallel algorithm, it is amenable to a parallel implementation. However, it is not a constrained tetrahedralization algorithm as it only operates on point cloud data. Like *TetGen*, our focus is on boundary-preserving triangulation, preserving input faces during tetrahedralization.

Our divide-and-conquer algorithm differs from *DeWall* in the non-recursive part. *DeWall* applies a merge step before the recursive step. This early-merge step uses a dividing plane and selects the vertices at either side of this plane to create an initial tetrahedralization. It chooses these vertices so that the generated tetrahedra have the smallest circumsphere radius to satisfy the Delaunay criterion. At this early merge step, the generated tetrahedra intersect the dividing plane. Then, it applies the same procedure recursively for the parts on either side of the dividing plane. We do not allocate buffer regions; we divide the mesh into parts and process them. Specifically, *DeWall* tetrahedralizes three pieces at each recursive step: the *DeWall* region around the dividing plane, left and right parts. We apply tetrahedralization to each part and do not spare a volume in the middle. The disadvantage of not reserving a middle region is that we cannot guarantee the Delaunay property for the tetrahedra around the cutting plane. Our rationale for not adopting that approach is not to slow down the process. Further, they could do this wall generation as part of a non-constrained triangulation algorithm. However, applying the same to a CDT algorithm might cause difficulties because a CDT algorithm must preserve the surface faces.

Chen et al. [29] proposed a parallel non-constrained near Delaunay triangulation algorithm. They divided the input into m blocks containing a nearly equal number of vertices. They triangulate each block using a divide-and-conquer algorithm. They call the area between these blocks as *interface*. These interfaces are built incrementally, applying a similar algorithm as used in *DeWall* to create the middle region. The middle-region creation is similar to *DeWall* and different from our algorithm because we do not spare a central part but divide the input mesh into several pieces directly. As discussed by Cignoni et al. [28], such input division cannot be readily applied to CDT algorithms because they need to preserve input faces.

Marot et al. [30] came up with a parallel 3D Delaunay Triangulation algorithm. Their Moore curve-based input partitioning allows different threads to work on different sets of vertices. They allocated a buffer zone between partitions to fix potential conflicts raised by multiple threads. In this approach, the boundary recovery stage where they preserve the input faces is not parallel. Only the non-constrained part of their process is parallel. The Delaunay Tetrahedralization part can be multi-processed, but they recover the boundary using the sequential pipeline of *TetGen*. However, in our approach, the boundary recovery is parallel, allowing us to reduce memory usage.

Hu et al. [31] later developed a faster version of *TetWild*, called *fTetWild*. It is as robust as the *TetWild* but at the same time significantly faster. They used parallelization structures to accelerate their algorithm. Similar to *TetWild*, input faces are not exactly preserved in the resulting tetrahedral mesh but are just approximated.

Kohout et al. [32] explored the parallelization of Delaunay triangulation algorithms on shared memory architectures. They investigated the effect of different

parallelization techniques on performance. However, they only focus on non-constrained Delaunay triangulation algorithms.

2.3 Input partitioning

We shall discuss algorithms that use input partitioning techniques to reduce the problem size. Joshi and Ourselind developed a constrained tetrahedralization algorithm that uses 3D convex decomposition and BSP trees [33]. They decompose the whole object into convex sub-polyhedra, tetrahedralize each piece and merge the meshes at the end. They accelerate the merge process using BSP trees. During the construction of the BSP tree, their algorithm introduces new vertices on the boundary. They experimented with non-convex polyhedra that are not very large. The boundary of the largest model they experimented with contains 26 vertices, and the number of produced tetrahedra for this model is 70. We did not consider such an approach because we cannot control the number of convex sub-polyhedra generated and might subdivide the problem redundantly. One problem with redundantly subdividing is that the overhead of merging at each step might significantly slow down the process. To this end, we divide the object into a user-defined number of pieces.

Smolik and Skala suggest a 3D triangulation algorithm that divides the input into a 3D Grid [34]. They extended their algorithm to be out-of-core so that the memory usage is reduced and large scenes can be tetrahedralized. However, their algorithm is not constrained. They accept a vertex cloud as input and embed each vertex to a cell in the 3D grid. After that, they triangulate each cell and merge them at the end cleverly to complete the algorithm. We could not apply that approach as we cannot divide the input surface mesh into a regular 3D grid. Triangles might be present in multiple grid cells, which makes it difficult tetrahedralize each cell. Therefore, we separate the object into several pieces instead of a grid structure.

Erkoc et al. [35] developed a divide-and-conquer constrained (boundary-preserving) tetrahedralization algorithm. They recursively divide the object into two at each step and call the *TetGen* as the base case. Unlike our algorithm, they do not introduce any parallelization structure. In addition, they do not propose any plane selection algorithm and introduce costly repairing and merging steps.

3 The proposed algorithm

The proposed algorithm is composed of three stages: *Input Partitioning*, *Surface Closure*, and *Merge* (cf. Algorithm 1).

Algorithm 1 Proposed Algorithm

```
1:  procedure TETRAHEDRALIZE(mesh, k, density_factor)
2:      meshes, planes = INPUT_PARTITIONING(mesh, k)
3:      for i = 0; i < meshes.size(); i++ do
4:          CLOSE_SURFACE(meshes[i], meshes[i + 1],
5:                  planes[i], density_factor)
6:      end for
7:      tetmeshes = []
8:      for i = 0; i < meshes.size(); i++ do
9:          tetmeshes[i] = TetGen(meshes[i])
10:     end for
11:     tetmesh = merge(tetmeshes)
12:     return tetmesh
13: end procedure
```

3.1 Input partitioning

The proposed algorithm begins by dividing the input mesh into several pieces (see Algorithm 2). We aim to divide the input surface mesh into as many evenly-sized pieces as possible. To this end, we need to find parallel planes that partition the mesh into equal-sized parts. We find such planes with the help of Principal Component Analysis (PCA). We apply PCA to the vertices of our input mesh to calculate the first principal component (PC_1). The PC_1 allows us to partition the input mesh into a maximum number of pieces. The PC_1 vector is the normal vector of these

parallel planes. We then find a different point in each of these planes to define their equations. To achieve that, we project the vertices of our mesh onto the PC_1. So, we end up with a one-dimensional *projections* array, and we sort it. At this stage, we need an input parameter, the number of parts, k. With that value in hand, we create a set of indices

$$I = \{(i/k) \times |V| \;\mid\; i \in \{0, 1, \ldots, k-1\}\},$$

where V is the vertex set of the input and $(k-1)$ corresponds to the number of planes we need to create k pieces. For instance, when $k = 2$, I will be $I = \{|V|/2\}$, which means we get the index of the median, and by letting the plane pass through the median, we can ensure that the division is balanced. We know how to find the projected elements and back-project them to world coordinates with the indices. The resulting points, along with the PC_1, will be used to construct the planes. Each part will contain an approximately equal number of vertices and hence an equal number of faces. Algorithm 3 shows the pseudo-code of the plane selection algorithm.

Algorithm 2 Input Partitioning

```
1: procedure INPUT_PARTITIONING(mesh, k)
2:     planes = FIND_PLANES(mesh, k)
3:     for each plane ∈ planes do
4:         INTERSECT_INSERT_POINTS(mesh, plane)
5:     end for
6:     meshes = redistribute_faces(mesh, planes)
7:                                  ▷ Distribute faces across different meshes
8:     return meshes, planes
9: end procedure
```

Algorithm 3 Plane Selection

```
1: procedure FIND_PLANES(mesh, k)
2:     principal_vectors = PCA(mesh)
3:     pc_1 = principal_vectors[0]
4:     planes = []
5:     projection = project(pc_1, mesh.vertices)
6:     for i = 0; i < k; i++ do
7:         index = (i/k) × |V|
8:         projected_point = projection[index]
9:         plane_point = back_project(principal_vectors, projected_point)
10:        new_plane = Plane(plane_point, pc_1) ▷ Create a new plane with a point
                                                ▷ on it and a normal vector
11:        planes.append(new_plane)
12:    end for
13:    return planes
14: end procedure
```

We need to insert new points into the mesh where the planes and mesh intersecting with these planes are known. Point insertion is essential because we want all points to be planar, simplifying the *Surface Closure* stage. The intersection and point insertion algorithm begins with iterating over all of the edges in the mesh. For each edge, we run a plane-segment intersection test. The intersection result can be a segment, a point, or nothing. We split the edge if the intersection is a point and the plane is not too close to the edge's endpoints. We control the closeness to an edge using a threshold parameter. We do not insert the point if the newly inserted point is close to either of the edges. For the intersection tests, we rely on exact predicates of the CGAL library. CGAL is reliable in detecting intersections. We have not come across an issue caused by intersection tests. Floating point errors occur when we insert

new points. Our method may fail if there are so many cut planes that intersect each other. When there are so many parallel planes, some may intersect due to floating point errors, although this never happened in our experiments. If we select an axis with very low variance, the extent of that axis is limited, so we cannot fit too many planes in there. However, if we find a very high-variance axis, we can have more room to store planes, thereby dividing the mesh into more pieces. Our aim in applying PCA is to find a high-variance axis.

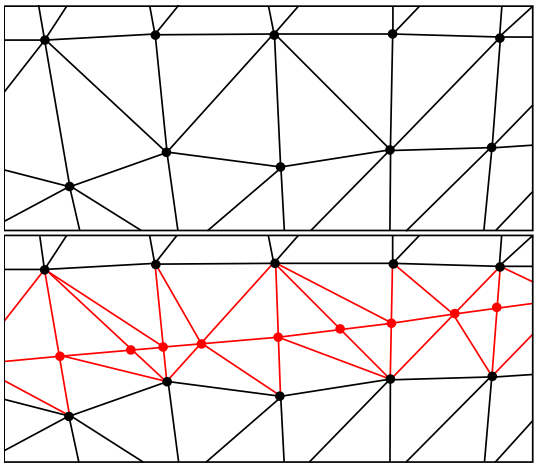


Fig. 2 The illustration of intersection and point insertion. The top image shows the input mesh before running the algorithm. The bottom image shows the result after point insertion. Newly inserted points are shown in red

Splitting the edge will introduce a new point between the two endpoints of the edge. We set the new point's location as the location of the intersection point. If the intersection is a line segment, we do not split it because the line segment is on the plane, eliminating the necessity for point insertion. We also skip an edge if the plane almost intersects one of its endpoints. Omitting this step would introduce nearly

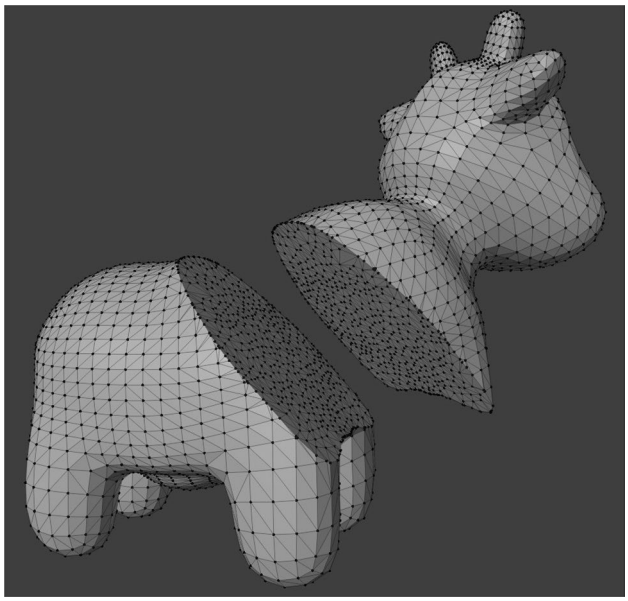


Fig. 4 The illustration of the parts after undergoing the surface closure stage. The hole is filled using triangulation. The resulting triangulation is inserted into both left and right meshes

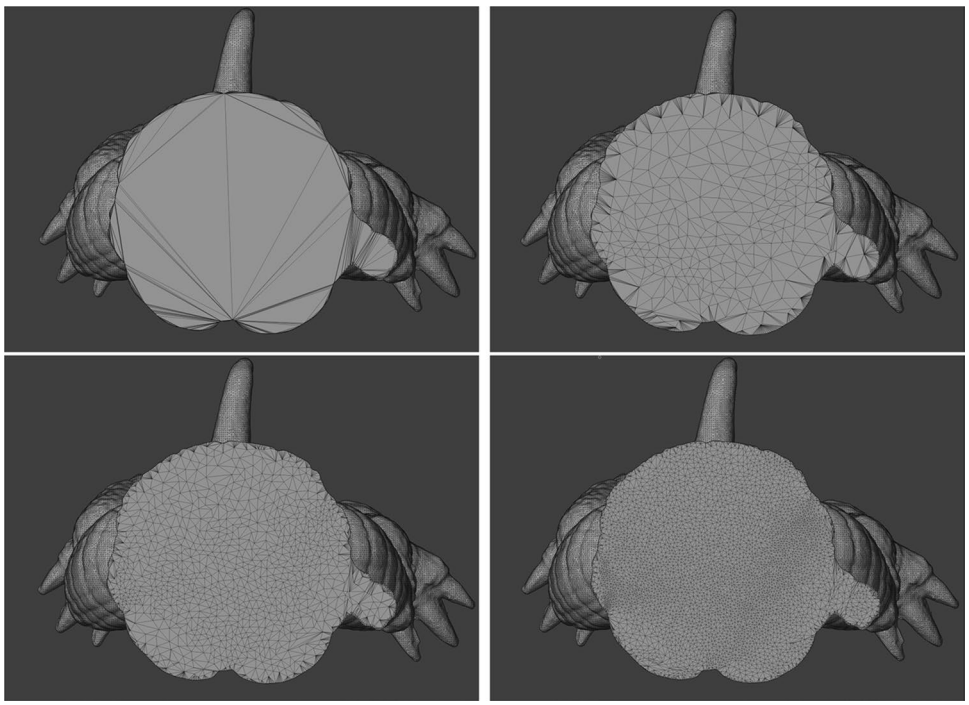


Fig. 3 The illustration of the refinement stage on the Armadillo model. The top-left image shows the mesh without refinement (only boundary-preserving tetrahedralization is applied). We use refinement to the others with the density factor 0.1, 0.2, and 0.4 for the top-right, bottom-left, and bottom-right images, respectively

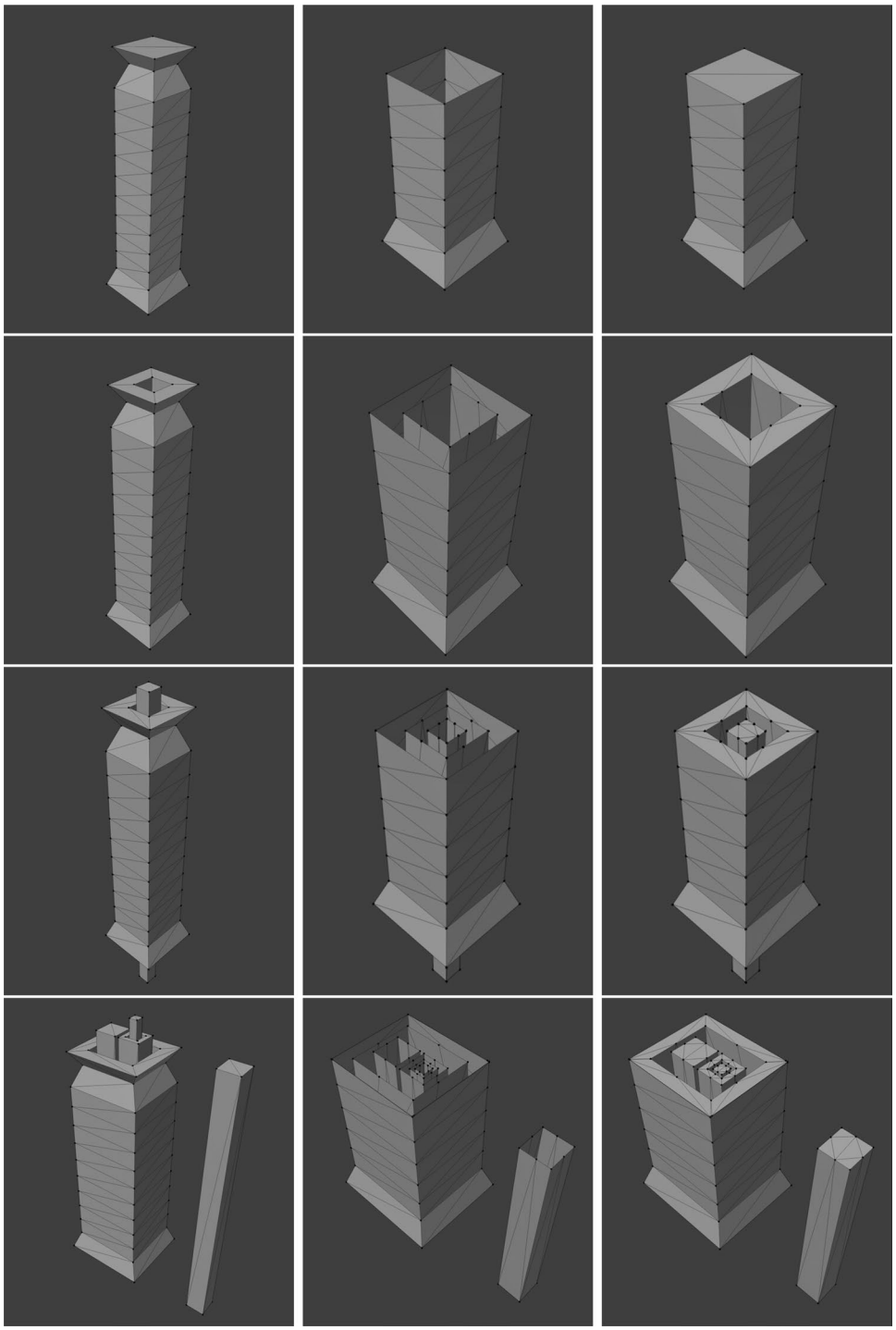


Fig. 5 The illustration of handling inputs with various topological structures during input partitioning. *First row:* the simplest case with a genus zero object. *Second row:* a genus one object. When we cut it in halves, we obtain two nested boundary cycles. We apply triangulation using the edges of both boundaries but only keep the triangles between them eventually. *Third row:* similar to the previous case, but we put another object inside the hole. After cut, we have three boundary cycles inter-bedded. Again, the triangulation is applied to all edges, but only necessary ones are kept. *Fourth row:* the combination of previous cases

duplicate points and tiny triangles that might be considered a self-intersection without sufficient floating-point precision. Figure 2 illustrates before and after the insertion of new points. After the intersection points are inserted into the mesh, we distribute the faces of the mesh into parts, ending up with k mesh objects. We remove those vertices during the post-processing stage to ensure all input faces are present in the output.

3.2 Surface closure

We apply a hole-filling operation on open boundaries. Although the state-of-the-art offers various methods to close a surface that nicely follows the curvature of the surface, they do not satisfy our needs [36, 37]. We aim to close the hole so it is guaranteed to be planar, and filling that boundary reduces to the 2D CDT problem.

Each part produced in the previous stage has open boundaries. We need to fill the holes because *TetGen* can only work on closed meshes. We use 2D Constrained Delaunay Triangulation to close the boundaries after finding them. The straightforward method of finding each boundary and triangulating the space inside might fail when the object has a genus of more than zero or a concavity in the input.

To handle all kinds of inputs, we do the following. First, we detect all boundaries and store them in an array. Each boundary will be a simple polygon, not intersecting one another. Then, we sort the array of boundary polygons in decreasing order of the polygonal area. We also create an array to keep track of potential parent polygons. We iterate over the sorted array and check if this polygon is the child of any polygon in the parent polygons array. If this is the case, we mark this polygon as the child of the parent polygon. Otherwise, we insert that polygon into the parent polygons array. We run a 2D CDT for each parent polygon where the constraint segments are the edges of all child polygons and the polygon itself. This process can fill the holes that should not be filled. To fix that, we run a breadth-first search on the triangulation to eliminate unnecessary triangles using the breadth-first search (BFS) implementation in CGAL [38]. This BFS implementation is similar to Shewchuck's "triangle-eating virus" algorithm [39]. Algorithm 4 gives the pseudo-code of the surface closure algorithm.

After we obtain the triangulation that closes the open boundaries, we apply further refinement to increase the quality of the triangles. The refinement stage subdivides the triangles using a density control factor parameter [38]. Increasing the value of this parameter leads to more uniform triangles, as illustrated in Fig. 3. However, this process also generates many new points and triangles, complicating the object to be tetrahedralized. We then insert this triangulation to meshes on both sides of the dividing plane, reversing the triangle vertex orders before adding them to the second mesh to achieve a consistent geometry. This way, we obtain two closed, watertight, and intersection-free meshes, just as TetGen requires (see Fig. 4).

Algorithm 4 Surface Closure

```
1:  procedure CLOSE_SURFACE(mesh_left, mesh_right,
                    plane, density_factor)
2:      boundaries = extract_boundaries(mesh_left)
3:      sort(boundaries)
4:      parent_boundaries = []
5:      for i = 0; i < len(boundaries); i++ do
6:          boundary = boundaries[i]
7:          parent_index = -1
8:          for j = 0; j < parent_boundaries.size(); j++ do
9:              if boundary ∈ parent_boundaries[i] then
10:                 parent_index = j
11:                 break
12:             end if
13:         end for
14:         if parent_index == -1 then
15:             parent_boundaries.add(Pair(i, [i]))
16:         else
17:             parent_boundaries[parent_index].second.add(i)
18:         end if
19:     end for
20:     # CDT Begins
21:     cdt = CDT(plane.normal)
22:     for each parent_boundary ∈ parent_boundaries do
23:         for each boundaries ∈ parent_boundary.second do
24:             cdt.add_constraints(boundaries)
25:         end for
26:     end for
27:     triangles = cdt.get_triangles()
28:     triangles = refine(triangles, density_factor)
29:     triangles = BFS(triangles)
30:     mesh_left.insert_triangles(triangles)
31:     triangles = triangles.reverse_order()
32:     mesh_right.insert_triangles(triangles)
33: end procedure
```

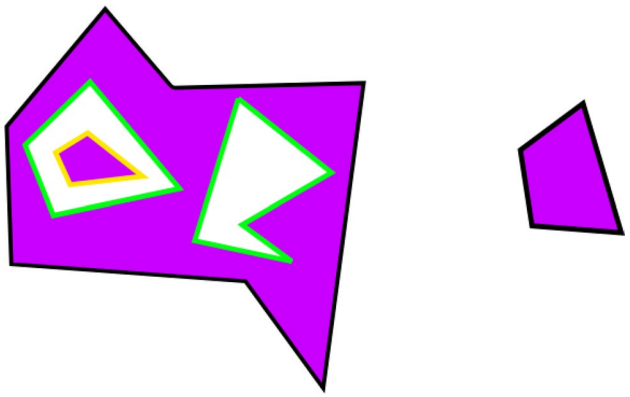


Fig. 6 The 2D view of example boundary polygons generated after surface partitioning

Figure 5 illustrates four cases with different topologies for the surface closure process. In each row, the leftmost image is the input mesh; the middle one is the bottom piece of the mesh when cut in half; the last one is after triangulating the boundaries.

Case 1: This is the simplest case with a genus zero object.

Case 2: This is a genus one object. When we cut it in halves, we obtain two nested boundary cycles. We triangulate using the edges of both boundaries but only keep the triangles between them eventually.

Case 3: It is similar to Case 2, but we put another object inside the hole. After cutting, we have three boundary cycles inter-bedded. Again, we triangulate using all edges, but only necessary ones are kept.

Case 4: This is the combination of previous cases.

We illustrate the surface closure process for the boundary polygons shown in Fig. 6. In this figure, the white regions in the image correspond to holes, while the purple areas correspond to our input domain. As a result of the parent finding algorithm, we can conclude that there are two-parent polygons here, shown in the black border. We then triangulate using all edges of the parent polygon and all the polygons inside it. There are three polygons inside the left parent polygon, whereas the parent polygon on the right is alone. We aim to form triangles in only purple regions. However, triangulation will also form triangles in the white areas. We run a BFS algorithm to eliminate them. Figure 7 illustrates the input partitioning and surface closure processes for Bunny and Armadillo objects.

3.3 Merge

We merge several tetrahedral meshes (tetmesh) in the merging stage and create one final tetmesh. One difficulty with the merge step is finding correspondence between tetrahedra around the cut region. Because each part is tetrahedralized independently, the neighboring tetrahedra at different parts will not be aware of one another. To find these missing neighbor relations, we store the neighborhood information during the *Surface Closure* stage, as we create triangles to close the boundary and use it in the merge stage. Figure 8 shows tetrahedral meshes generated from the merge stage. Since we know that TetGen will preserve the triangles, the triangles around the cut region will eventually perfectly fit after the tetrahedral mesh is created.

We adjust *TetGen* parameters so that it preserves the triangles. We disallow it to insert any new points into the

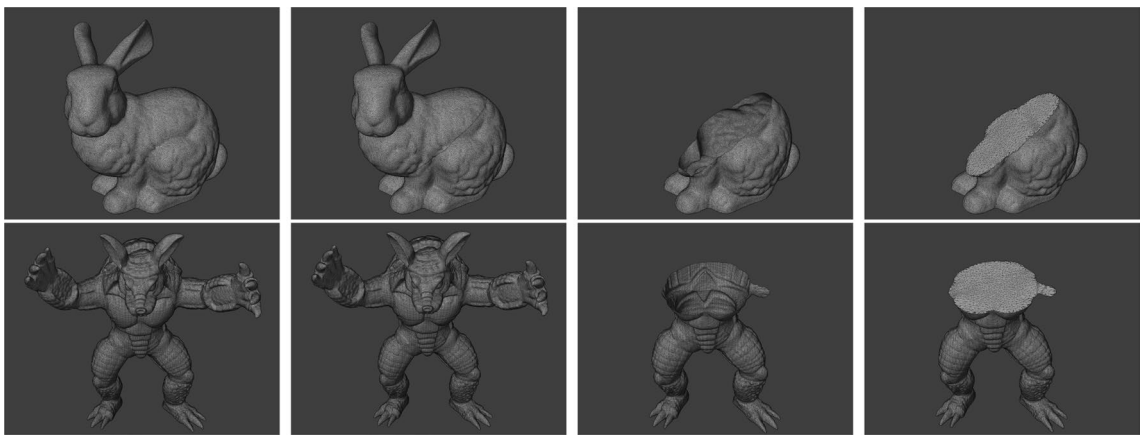


Fig. 7 The input partitioning and surface closure stages are illustrated on Bunny (top row) and Armadillo objects (bottom row). The first column is the input mesh; the second column is the mesh after point insertion; the third column is the object's bottom half; the last column is the bottom half after the surface closure algorithm

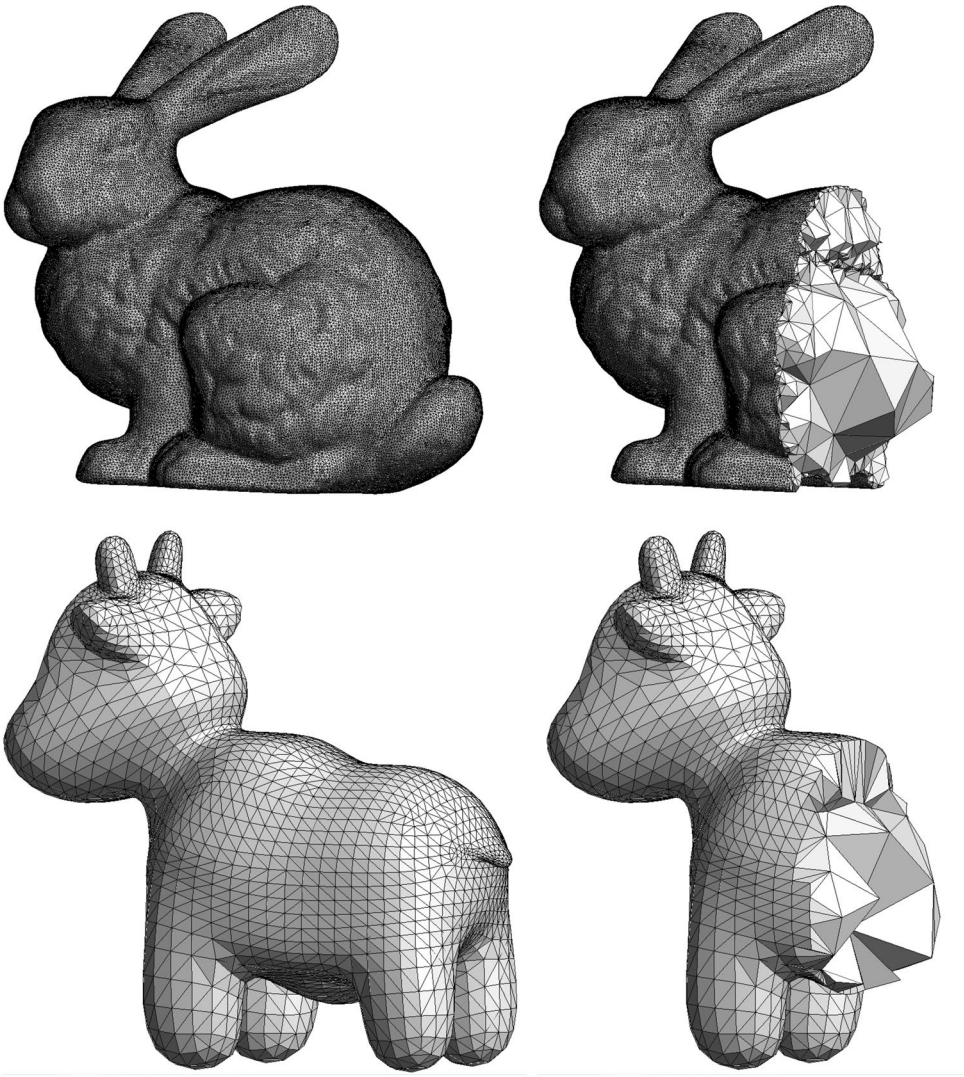


Fig. 8 Example tetrahedral meshes generated with our implementation. Tetrahedral mesh with edges and faces (left). The cut mesh shows tetrahedra inside the model (right)

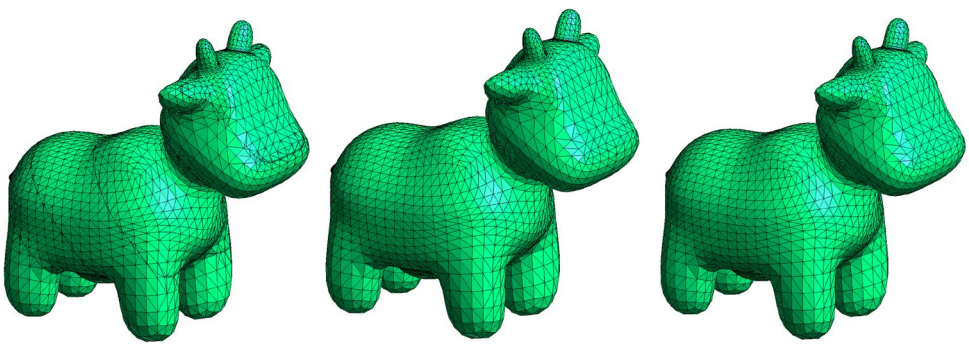


Fig. 9 Example tetrahedral meshes illustrating the result of the post-processing step. Post-processing is enabled in the middle image but not in the left image. The right image is the result of *TetGen*. All extra vertices on the boundary are removed, and the constrained faces are faithful to the input mesh. Our result with post-processing enabled is identical to the *TetGen*'s output, which is the right image

boundary. Otherwise, corresponding pieces would not match during merging. Rarely, *TetGen* refuses to remove the Steiner points it inserted into the boundary leading to a new point on the domain. We detect such cases and terminate the process immediately, resulting in an unsuccessful operation. This situation does not cause a significant problem because it

Table 1 Vertex and face counts of the objects used in the experiments

	# Vertices	# Faces
Spot	2930	5856
Bob	5344	10,688
Blub	7106	14,208
Bunny	72,027	144,046
Pitt Brdg	75,081	150,170
Armadillo	172,971	345,938
Nefertiti	1,009,118	2,018,232
Neptune	2,003,932	4,007,872
Nefertiti2	6,054,698	12,109,392

Nefertiti2 is the high-resolution version of the Nefertiti model

occurs rarely, and it will be improved in upcoming releases of *TetGen*. Si and Goerigk [40] describe how the Chazelle polyhedra, a family of non-convex polyhedra, can be tetrahedralized in a *boundary-preserving fashion* without modifying its exterior boundary, i.e., by only inserting Steiner points in the interior of it.

During the merge step, we apply post-processing to remove the vertices introduced while partitioning the input. We keep track of such vertices and remove them from the tetrahedral mesh, which would create a cavity in the tetrahedral mesh. Then, we tetrahedralize the cavity using *TetGen*. We ensure that the input mesh's original faces stay intact thanks to this operation. Figure 9 shows tetrahedral meshes with and without the postprocessing stage and the *TetGen* output. The postprocessing stage makes our output tetrahedral mesh identical to the *TetGen* output by removing extra vertices unless the tetrahedralization fails because of the failure of the merge step, as discussed in Sect. 3.3.

4 Modes of the algorithm

Our algorithm has two modes: *parallel processing* and *memory requirement reduction*. The parallel processing mode uses parallelization to speed up the process. The memory requirement reduction mode uses single-threaded programming and intermediate files to reduce memory usage.

4.1 Parallel processing

Because our framework allows input mesh to be divided into several pieces, we can apply multi-threaded processing at different places in our implementation. Each piece, after division, can be tetrahedralized entirely, independent from the other. If we process them simultaneously, no racing condition will occur. Hence, we parallelize the for-loop at the 8^{th} line of Algorithm 1. The mesh object is an instance

Table 2 Experiments on tetrahedral mesh quality with the average slim energy quality metric

Model	*TetGen*	Density control parameter				
		0.1	0.2	0.4	0.8	1.6
Spot	6.26	7.26	6.72	**6.15**	6.23	6.33
Bob	**5.99**	7.74	6.82	6.41	6.16	6.27
Blub	7.84	8.79	8.25	7.89	7.74	**7.46**
Pitt Brdg	7.27	7.72	7.36	7.26	7.317	**7.14**
Armadillo	6.65	6.82	6.76	6.71	6.681	**6.61**

The first column is the average slim energy for the standalone *TetGen* execution. Other columns show the average slim energy values for the corresponding density control parameter. The smallest value at each row is in bold

Table 3 Experiments on tetrahedral mesh quality with the maximum slim energy quality metric

Model	*TetGen*	Density control parameter				
		0.1	0.2	0.4	0.8	1.6
Spot	**75.90**	992.79	**75.90**	**75.90**	78.1	180
Bob	**60.96**	265.39	210.25	62.38	110.56	273.50
Blub	148.24	614.40	101.61	**81.42**	101.86	205.45
Pitt Brdg	91.43	231.55	118.19	147.33	**79.53**	123.12
Armadillo	**224721.83**	**224721.83**	**224721.83**	**224721.83**	**224721.83**	**224721.83**

The table shows the maximum slim energy for each shape across different control parameter values. The first column is the results for the standalone *TetGen* execution. Other columns show the values for the corresponding density control parameter. The smallest value at each row is in bold

Table 4 Experiments on the effects of post-processing (i.e., vertex removal) step on tetrahedral mesh quality

Model	*TetGen*	Without Postprocessing	With Postprocessing
Spot	6.26	9.59	7.26
Bob	5.99	9.91	7.74
Blub	7.84	10.84	8.79
Pitt Brdg	7.27	8.21	7.72
Armadillo	6.65	7.09	6.82

The quality metric is average slim energy; the lower it is, the better. The density control parameter is selected as 0.1

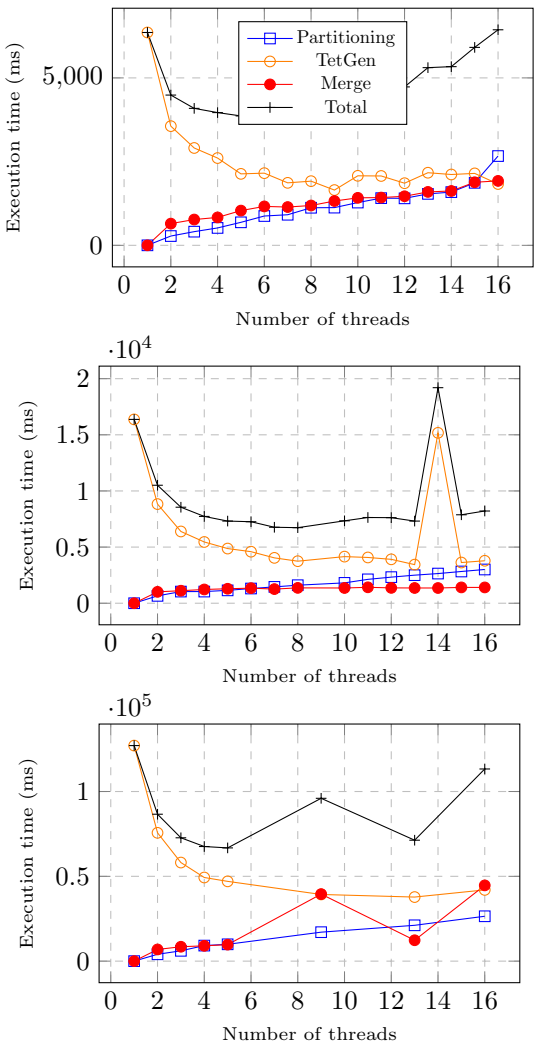


Fig. 10 Execution time dissection for Bunny, Armadillo, and Nefertiti objects (from top-to-bottom). Single-threaded execution corresponds to sequential *TetGen*

of the *Surface Mesh* class belonging to the CGAL library, which states that the object is vulnerable to race conditions [38]. This vulnerability of mesh objects prevented us from parallelizing some methods due to the high costs incurred by critical sections. Moreover, we decided not to apply a multi-threading scheme to the *Merge* stage because it is already fast and includes file I/O, which needs to be synchronized, diminishing the benefits of parallelism. Besides, we observe in Fig. 10 that the *Merge* step is not the bottleneck because its computational cost is a small percentage of the computational cost of the whole process. To parallelize code segments, we have used OpenMP 2.0 [41].

4.2 Memory requirement reduction

The benefit of reducing memory usage is two-fold. First, it allows the tetrahedralization of objects that would be impossible due to a memory shortage. Secondly, it will enable multiple objects that require high memory to be tetrahedralize simultaneously. For example, if we have a computer with 64 GB of RAM and two objects requiring 64 and 60 GB of memory, a sequential algorithm could only process either in that machine. However, our implementation can process both simultaneously by dividing each mesh at least once.

We have taken several precautions to minimize memory footprint when it comes to implementation. More specifically, we aimed to reduce the peak memory usage of our implementation. If the computer does not have enough memory to accommodate the peak memory needed for execution, it will terminate without processing the input. For such cases, we disable multi-threading and opt for single-threaded execution. Simultaneously processing multiple pieces requires memory to hold the data for all parts. Eventually, the memory footprint will be no less than *TetGen*. Instead of keeping the meshes in memory, we store the file handles. When a part is needed, we read it from the file, and when we update it, we write the changes to the corresponding file.

5 Experiments

We conducted experiments on mesh quality, parallel processing performance, and memory requirements. The statistics about the meshes used in the experiments are given in Table 1.

5.1 Mesh quality

We performed experiments on the quality of the resulting tetrahedral meshes. We used slim energy as the quality measure, as in *TetWild* [18]. The smaller the energy is, the higher the quality is. The final quality value for a tetrahedral mesh is the average slim energy across all tetrahedra.

Table 2 depicts the effect of the density control parameter on mesh quality using the average slim energy quality metric. When we merge the partial tetrahedral meshes, the triangles we newly created for *Surface Closure* stage will be the faces of internal tetrahedra. We expect to observe an increase in quality with increasing density control parameter values. Thanks to the large density parameter, we could get higher-quality meshes than *TetGen* in some cases. Table 3 shows the effect of the density control parameter on mesh quality using the maximum slim energy quality metric, which corresponds to the worst quality of the produced tetrahedra. In some cases, our algorithm produces tetrahedra with maximum slim energy and better quality than *TetGen*. Hence, the non-Delaunay triangles we introduce do not create many problems.

We also investigated the effect of the post-processing/vertex removal step on the tetrahedral mesh quality. As shown in Table 4, removing extra vertices and tetrahedralizing the cavity increases the tetrahedral mesh quality. The quality difference between our meshes and *TetGen*'s become similar with post-processing enabled. Hence, although our algorithm may create some non-Delaunay triangles, the quality difference appears slim compared to the gain achieved by reducing the memory footprint and computation time.

Table 5 The effect of density control parameter on the execution time

	TetGen	Density control parameter				
		0.1	0.2	0.4	0.8	1.6
Spot	120	139	115	**108**	123	174
Bob	283	475	214	225	**212**	354
Blub	413	307	**289**	318	338	467
Pitt Brdg	4502	3642	3557	**3309**	3533	3822
Armadillo	14793	**9203**	9542	9484	11131	22492

The execution time is in milliseconds. The best values are shown in bold. We divided the object into two and used two threads for these experiments

Table 6 Memory usage (in MB) and processing times (in seconds) for various models with post-processing enabled

Model	Part counts									
	1		2		4		8		16	
	Mem	Time	Mem	Time	Mem	Time	Mem	Time	Mem	Time
Armadillo	923	12	542	22	336	34	268	60	360	109
Nefertiti	5200	85	3700	166	2500	248	2200	475	3000	726
Neptune	11800	250	6700	344	4400	500	–	–	–	–
Nefertiti2	x	x	23500	1552	–	–	–	–	–	–

When the part count is one, *TetGen* is directly used. Our implementation failed for some input-part count pairs due to the floating-point errors; these are shown with "-". The inputs that *TetGen* could not process are marked with an "x". Nefertiti2 is the high-resolution version of the Nefertiti model

Table 7 The effect of density control parameter on memory usage

	TetGen	Density control parameter				
		0.1	0.2	0.4	0.8	1.6
Spot	21	16	16	14	16	20
Bob	33	25	23	24	25	30
Blub	43	30	28	28	36	39
Pitt Brdg	388	227	228	256	237	262
Armadillo	930	547	582	565	623	944

The memory usage is in Megabytes

5.2 Parallel processing

We conducted experiments to see how our parallelization scheme performs. The PC used for the experiments has two eight-core processors, equivalent to 16-core processing power.

We first selected a few objects and tetrahedralized them using up to 16 threads. We calculated how much computation time the program spends on the Partitioning, *TetGen*, and Merge steps. In our experiments, we ignored reading and writing times. The execution time of the *TetGen* stage is calculated as the time the longest thread has taken to tetrahedralizing the piece it is responsible for. Figure 10 plots the execution time dissection for various models. The division and merge steps do not take a significant amount of time, but the *TetGen* stage dominates execution. Importantly, we observed a steady speed-up improvement up to around eight cores. When we increase the thread count above eight, the speed-up starts decreasing.

The graphs show that *TetGen* execution times increase as the number of threads increases. The reason may be the imbalanced data partition. If the processing of the whole mesh by *TetGen* on a single thread takes T time, the threads processing each piece should ideally complete the execution in T/X time, where X is the number of threads. However, this ideal case does not always occur. Some threads run faster, and some are slower due to imbalanced input partitioning. We ensure that each part has an equal number of vertices and faces, but the execution time of each thread might be different due to topological differences. In other words, even though the parts have a similar number of vertices, *TetGen* may need to spend different computational times processing each part depending on the shape of the part. For instance, the amount of Steiner points may vary depending on the topology/shape of the input, which affects the time *TetGen* spends to add/remove them. In addition, we are inserting new faces during the *Surface Closure* stage, and each part might get a different number of vertices and faces appended to it depending on the boundary polygons. All these factors lead to data imbalance. As we increase the number of pieces, this issue becomes crucial, slowing down the process.

The choice of the density control parameter used in the refinement stage affects the execution time of our algorithm. Incrementing that value increases the number of triangles used to fill the holes and the quality of the triangles (cf. Subsection 5.1). Higher-quality triangles require *TetGen* to spend less time optimizing the mesh. Table 5 shows that the value of 0.4 seems reasonable for this parameter considering the trade-off between the computational cost and mesh quality.

Some cases failed because of the precision issues that occurred while inserting new points into the mesh. We used an inexact construction kernel of the CGAL [38]; this is why such failures might occur. In addition, *TetGen* rarely inserts Steiner points on the input triangles, preventing the merge process from running correctly. The merge fails if the triangles at both sides do not match due to new Steiner points. We count this as a failure case too. These problems mainly emerge because we use an inexact construction kernel of CGAL during mesh division. Switching to exact construction would lead to a more robust approach, which increases the computation cost. Such edge subdivision operations have been used in applications like self-intersection fixing, but it costs a lot of time. So, that design choice creates a trade-off between robustness and speed. If such failure cases occur, the user can update the density control or threshold parameter that defines the closeness of two points.

5.3 Memory requirements

We tested our algorithm with several objects and observed the peak memory usage and execution times by enabling the memory reduction mode of our algorithm. We limited the available memory to 36 GB (16 GB Physical RAM + 20 GB Virtual Memory). Table 6 shows the results for various input-part count pairs. We excluded reading and writing times from the execution time.

As we increase the number of pieces, we see a significant decrease in peak memory usage despite increasing execution time. Moreover, *TetGen* could not process the Nefertiti2 model due to high memory usage, whereas our method could tetrahedralize it. The execution time of the memory-efficient version of our algorithm appears to be reasonable compared to *TetGen*.

We also investigated the relationship between memory usage and the density control parameter. Table 7 shows that increasing the density parameter increases memory usage. Although it is often less than the standalone *TetGen* execution, the memory usage can even exceed that in some cases if the density control parameter is too high. We expect this result because the density parameter controls the number of triangles, and its increase leads to more triangles. Creating more triangles means tetrahedralizing objects with more vertex and triangles counts, which increases memory usage.

6 Conclusion

We propose a divide-and-conquer algorithm that can be used to reduce memory usage or speed up the constrained tetrahedral meshing process. Although our algorithm may introduce some non-Delaunay triangles, it can increase the quality of the tetrahedral mesh. Despite non-Delaunay triangles,

the increase in quality makes our method useful. We can even successfully tetrahedralize meshes that *TetGen* cannot do due to lack of memory. Although our input partitioning stage introduces new vertices, we remove them during the merge step to conserve the input triangles.

We could extend our work in various ways. Firstly, we used PCA and parallel planes during input partitioning to reduce the overhead. To set a trade-off between speed and more balanced decomposition, we could experiment with other approaches, such as convex decomposition and recursive PCA. As convex decomposition is relatively slow, and partitioning with non-parallel planes -as with recursive PCA- requires a complicated merge step (i.e., BSP trees to keep track of neighboring pieces efficiently), the overall process may be slower, but decomposition could be more balanced. Secondly, we applied our framework to only *TetGen*. However, it can be used with other meshing tools such as *TetWild*. Since *TetWild* may also suffer from out-of-memory errors, it would benefit from such an approach.

Acknowledgements This research is supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant No. 117E881. The Bunny and Armadillo are obtained from The Stanford 3D Scanning Repository. The Neptune model is courtesy of Laurent Saboret by the AIM@SHAPE-VISIONAIR Shape Repository. The Spot, Blub, Bob, Pittsburg Bridge, and Nefertiti models are courtesy of Keenan Crane's 3D Model Repository.

Data Availability Data used to test our implementation is from publicly available resources listed in the acknowledgments.

Declarations

Conflict of interest The authors declare that they have no conflict of interest/competing interests.

References

1. Susersic TI, Filipovic N (2020) Computational modeling of dry-powder inhalers for pulmonary drug delivery. In: Filipovic N (ed) Computational modeling in bioengineering and bioinformatics. Academic Press, Cambridge, pp 257–288
2. Anderson P, Fels S, Harandi NM, Ho A, Moisik S, Sánchez CA, Stavness I, Tang K (2017) FRANK: a hybrid 3D biomechanical model of the head and neck. In: Payan Y, Ohayon J (eds) Biomechanics of living organs, vol 1. Translational epigenetics. Academic Press, Oxford, pp 413–447
3. Payan Y, Ohayon J (eds) (2017) Biomechanics of living organs. Translational epigenetics, vol 1. Academic Press, Oxford
4. Tu J, Yeoh G-H, Liu C (2018) CFD mesh generation: a practical guideline. In: Tu J, Yeoh G-H, Liu C (eds) Computational fluid dynamics, 3rd edn. Butterworth-Heinemann, Oxford, pp 125–154
5. Molino N, Bridson R, Teran J, Fedkiw R (2003) A crystalline, red green strategy for meshing highly deformable objects with tetrahedra. In: Shepherd J (ed) Proceedings of the 12th International Meshing Roundtable, IMR 2003, pp 103–114
6. Teran J, Sifakis E, Irving G, Fedkiw R (2005) Robust quasistatic finite elements and flesh simulation. In: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation. SCA '05. Association for Computing Machinery, New York, NY, USA, pp 181–190
7. Sifakis E, Der KG, Fedkiw R (2007) Arbitrary cutting of deformable tetrahedralized objects. In: Proceedings of the ACM SIGGRAPH/Eurographics Symposium on Computer Animation. SCA '07. Eurographics Association, Goslar, DEU, pp 73–80
8. Montazerin N, Akbari G, Mahmoodi M (2015) Developments in turbomachinery flow: forward curved centrifugal fans, vol 1. Woodhead Publishing, Sawston
9. Driscoll M (2019) The impact of the finite element method on medical device design. J Med Biol Eng 39:171–172
10. Mollica F, Ambrosio L (2009) The finite element method for the design of biomedical devices. In: Merolli A, Joyce TJ (eds) Biomater Hand Surg. Springer, Milano, pp 31–45
11. Wittek A, Miller K (2020) Computational biomechanics for medical image analysis. In: Zhou SK, Rueckert D, Fichtinger G (eds) Handbook of medical image computing and computer assisted intervention. The Elsevier and MICCAI society book series. Academic Press, London, pp 953–977
12. Freutel M, Schmidt H, Dürselen L, Ignatius A, Galbusera F (2014) Finite element modeling of soft tissues: material models, tissue interaction and challenges. Clin Biomech 29(4):363–372
13. Galbusera F, Niemeyer F (2018) Mathematical and finite element modeling. In: Galbusera F, Wilke H-J (eds) Biomechanics of the spine. Academic Press, Oxford, pp 239–255
14. Schneider T, Hu Y, Gao X, Dumas J, Zorin D, Panozzo D (2019) A large scale comparison of tetrahedral and hexahedral elements for finite element analysis. arXiv preprint arXiv:1903.09332
15. Shewchuk JR (2008) General-dimensional constrained Delaunay and constrained regular triangulations, I: combinatorial properties. Discret Comput Geom 39(1):580–637
16. Lagae A, Dutré P (2008) Accelerating ray tracing using constrained tetrahedralizations. Comput Graph Forum 27(4):1303–1312
17. Si H (2015) TetGen: a Delaunay-based quality tetrahedral mesh generator. ACM Trans Math Softw 41(2):1–36
18. Hu Y, Zhou Q, Gao X, Jacobson A, Zorin D, Panozzo D (2018) Tetrahedral meshing in the wild. ACM Trans Graph 37(4):60–16014
19. Bridson R, Doran C (2022) Quartet: a tetrahedral mesh generator based on Jonathan Shewchuk's isosurface stuffing algorithm. Available at: https://github.com/crawforddoran/quartet. Accessed 10 Oct 2022
20. Dey TK, Levine JA (2009) Delaunay meshing of piecewise smooth complexes without expensive predicates. Algorithms 2(4):1327–1349
21. Dobrzynski C (2012) MMG3D: tetrahedral fully automatic remesher. User Guide. Technical Report RT-0422, HAL Id: hal-00681813, The National Institute for Research in Digital Science and Technology (INRIA). Available at: https://hal.inria.fr/hal-00681813/document. Accessed 10 Oct 2022
22. Chew LP (1993) Guaranteed-quality mesh generation for curved surfaces. In: Proceedings of the Ninth Annual Symposium on Computational Geometry. SCG '93. Association for Computing Machinery, New York, NY, USA, pp 274–280
23. Chernikov AN, Chrisochoides NP (2008) Algorithm 872: parallel 2D constrained Delaunay mesh generation. ACM Trans Math Softw 34(1):1–20
24. Linardakis L, Chrisochoides N (2008) Algorithm 870: A static geometric medial axis domain decomposition in 2D Euclidean space. ACM Trans Math Softw 34(1):1–28
25. Coll N, Guerrieri M (2017) Parallel constrained Delaunay triangulation on the GPU. Int J Geogr Inf Sci 31(7):1467–1484

26. Blandford DK, Blelloch GE, Kadow C (2006) Engineering a compact parallel Delaunay algorithm in 3D. In: Proceedings of the Twenty-second Annual Symposium on Computational Geometry. SoCG '06, pp 292–300
27. Chernikov AN, Chrisochoides NP (2008) Three-dimensional Delaunay refinement for multi-core processors. In: Proceedings of the 22nd Annual International Conference on Supercomputing. ICS '08. Association for Computing Machinery, New York, NY, USA, pp 214–224
28. Cignoni P, Montani C, Scopigno R (1998) DeWall: a fast divide and conquer Delaunay triangulation algorithm in e^d. Comput Aided Des 30(5):333–341
29. Chen M-B, Chuang T-R, Wu J-J (2004) Efficient parallel implementations of near Delaunay triangulation with high performance Fortran. Concurr Comput 16(12):1143–1159
30. Marot C, Pellerin J, Remacle J-F (2019) One machine, one minute, three billion tetrahedra. Int J Numer Method Eng 117(9):967–990
31. Hu Y, Schneider T, Wang B, Zorin D, Panozzo D (2020) Fast tetrahedral meshing in the wild. ACM Trans Graph 39(4):117–111718
32. Kohout J, Kolingerová I, Žára J (2005) Parallel Delaunay triangulation in E^2 and E^3 for computers with shared memory. Parallel Comput 31(5):491–522
33. Joshi BJ, Ourselin S (2003) BSP-assisted constrained tetrahedralization. In: Proceedings of the 12th International Meshing Roundtable. IMR '03, pp. 251–260
34. Smolik M, Skala V (2014) Fast parallel triangulation algorithm of large data sets in E^2 and E^3 for in-core and out-core memory processing. Proceedings of the international conference on computational science and its applications ICCSA 14. Springer, Cham, pp 301–314
35. Erkoç Z, Aman A, Güdükbay U, Si H (2021) Out-of-core constrained Delaunay tetrahedralizations for large scenes. In: Garanzha VA, Kamenski L, Si H (eds) Numerical geometry, grid generation and scientific computing. Springer, Cham, pp 113–124
36. Zhao W, Gao S, Lin H (2007) A robust hole-filling algorithm for triangular mesh. Vis Comput 23(12):987–997
37. Tekumalla LS, Cohen E (2004) A hole-filling algorithm for triangular meshes. Technical Report UUCS-04-019, School of Computing, University of Utah, UT, USA
38. The CGAL Project (2020) CGAL user and reference manual. CGAL Editorial Board. Available at: https://doc.cgal.org/5.0.2/Manual/packages.html, Accessed 10 Oct 2022
39. Shewchuk JR (1996) Triangle: engineering a 2D quality mesh generator and Delaunay triangulator. Proceedings of workshop on applied computational geometry. Springer, Cham, pp 203–222
40. Si H, Goerigk N (2018) On tetrahedralisations of generalised Chazelle polyhedra with interior Steiner points. Comput Aided Des 103:61–72. https://doi.org/10.1016/j.cad.2017.11.005
41. Dagum L, Menon R (1998) OpenMP: an industry standard API for shared-memory programming. IEEE Comput Sci Eng 5(1):46–55

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Springer Nature or its licensor (e.g. a society or other partner) holds exclusive rights to this article under a publishing agreement with the author(s) or other rightsholder(s); author self-archiving of the accepted manuscript version of this article is solely governed by the terms of such publishing agreement and applicable law.

Authors and Affiliations

Ziya Erkoç[1] · Uğur Güdükbay[1] · Hang Si[2]

Ziya Erkoç
ziya.erkoc@bilkent.edu.tr

Hang Si
si@wias-berlin.de

[1] Department of Computer Engineering, Bilkent University, 06800 Ankara, Turkey

[2] Weierstrass Institute for Applied Analysis and Stochastics, Mohrenstraße 39, Berlin, Germany