Out-of-core Constrained Delaunay Tetrahedralizations for Large Scenes

Ziya Erkoç, Aytek Aman, Uğur Güdükbay, and Hang Si

Abstract Tetrahedralization algorithms are used for many applications such as Ray Tracing and Finite Element Methods. For most of the applications, constrained tetrahedralization algorithms are chosen because they can preserve input triangles. The constrained tetrahedralization algorithms developed so far might suffer from a lack of memory. We propose an out-of-core near Delaunay constrained tetrahedralization algorithm using the divide-and-conquer paradigm to decrease memory usage. If the expected memory usage is below the user-defined memory limit, we tetrahedralize using TetGen. Otherwise, we subdivide the set of input points into two halves and recursively apply the same idea to the two halves. When compared with the TetGen, our algorithm tetrahedralizes the point clouds using less amount of memory but takes more time and generates tetrahedralizations that do not satisfy the Delaunay criterion at the boundaries of the merged regions. We quantify the error using the aspect-ratio metric. The difference between the tetrahedralizations that our approach produce and the Delaunay tetrahedralization are small and the results are acceptable for most applications.

1 Introduction

Tetrahedralization has many applications, ranging from finite element simulations to ray tracing accelerations. There are notable tetrahedralization algorithms in the literature. Yet, these algorithms are not appropriate for applications that require the use of very large meshes that do not fit into the memory. Besides, some of these applications require the faces of the input mesh to be preserved after

Z. Erkoç · A. Aman · U. Güdükbay (✉)
Department of Computer Engineering, Bilkent University, Ankara, Turkey
e-mail: ziya.erkoc@bilkent.edu.tr; aytek.aman@cs.bilkent.edu.tr; gudukbay@cs.bilkent.edu.tr

H. Si
Weierstrass Institute for Applied Analysis and Stochastics, Berlin, Germany
e-mail: si@wias-berlin.de

V. A. Garanzha et al. (eds.), *Numerical Geometry, Grid Generation and Scientific Computing*, Lecture Notes in Computational Science and Engineering 143,
https://doi.org/10.1007/978-3-030-76798-3_7

the tetrahedralization is complete. This kind of tetrahedralization algorithm is called *constrained tetrahedralization*. Constrained tetrahedralization algorithms guarantee that the input mesh is contained in the surface of the output tetrahedral mesh. This property is especially important for tetrahedralization-based ray tracing accelerators, not to disturb the surface geometry [4]. Although these algorithms are quite powerful and can complete the tetrahedralization process in a reasonable amount of time, their usage is limited by the available memory. These algorithms might fail when the tetrahedralization of an object requires a large amount of memory. For instance, a bridge model consisting of tens-of-millions of vertices can be analyzed using the Finite Element Method. Besides, a ray-traced scene may contain up to a few hundred million faces [8]. Those examples might require an excessive amount of memory.

We propose an out-of-core divide-and-conquer constrained tetrahedralization algorithm that will take the memory-constraint specified by the user into account and will not exceed it. Our algorithm divides the mesh into two pieces recursively as long as the expected memory usage is above the given constraint. When the given mesh is small enough to satisfy memory requirement it is tetrahedralized. Since our algorithm does not guarantee satisfying Delaunay property for every tetrahedron, it is a near Delaunay tetrahedralization algorithm. Yet, this approximation may be reasonable for applications such as tetrahedralizations as acceleration structures for Ray Tracing [4]. In their paper, Lagae and Dutré show that if tetrahedralization algorithms that are used in ray tracing relax the Delaunay criterion, the construction time decreases but the rendering time increases. Hence, our algorithm offers a trade-off between these two timings.

2 Related Works

There are many triangulation algorithms in the literature. However, these algorithms cannot tetrahedralize large models that do not fit into the memory in a constrained fashion.

Smolik and Skala put forth a divide-and-conquer tetrahedralization algorithm that works both in CPU and GPU [6]. They also develop an out-of-core version of their algorithm and observe a decrease in memory usage, thereby being able to tetrahedralize large objects with the same available memory. Yet, their algorithm is not a constrained tetrahedralization. They divide the input point cloud into a 3D grid and simultaneously tetrahedralize each grid cell and finally merge the cells. Our approach is different because we, before all, have developed a constrained tetrahedralization algorithm. Yet, dividing the object into grids may not be possible for constrained tetrahedralization because triangles should not extend to more than one grid cell, which is not possible with their algorithm. Hence, we divide the object into two at any time instead of dividing it into many small pieces.

Cignoni et al. propose a divide-and-conquer algorithm to triangulate meshes of any dimension [3]. However, they do not describe an out-of-core extension of

their algorithms. Besides, their algorithm is not a constrained tetrahedralization algorithm. Our divide-and-conquer algorithm differs from DeWall in the non-recursive part. Their algorithm applies a merge step before recurring. In this early-merge step, their algorithm uses a dividing plane, and by selecting the closest vertices at either side of this plane, it creates an initial tetrahedralization. Specifically, they choose these vertices so that the generated tetrahedra has the smallest circumsphere radius to satisfy the Delaunay criterion. Therefore, at this step, all the tetrahedra generated intersects that virtual plane. Then, it applies the same recursively for the other two sides. Our method is different from theirs in the sense that we do not select an area to be initially tetrahedralized; we just cut the mesh into two and tetrahedralize the parts. DeWall performs three tetrahedralization operations at each recursive step, one for around the plane, one for the left, and one for the right side. However, we only tetrahedralize left and right without allocation a middle region to be tetrahedralized. This approach comes with a cost for us because we do not have a middle-region like DeWall. Hence, we cannot guarantee that the Delaunay criterion is satisfied for the tetrahedra around the cutting plane.

Blelloch et al. [1] present a parallel Delaunay triangulation algorithm. Their algorithm uses the divide-and-conquer paradigm like ours. They utilize parallelism at the pre-recursive step to reduce the overall run-time cost. They experimented on various point distributions and observed significant improvements. Our algorithm is different because ours is a constrained triangulation algorithm that takes into account not only the input points but also the input triangles. Besides, while we aim to reduce the memory usage by compromising run-time, Blelloch et al. want to improve the run-time performance. Since they introduce parallelism, they need to include new data structures, which require an extra set of data stored in the main memory. Therefore, developing a parallel algorithm might defy our purpose of creating a memory-efficient algorithm.

Si developed a constrained Delaunay tetrahedralization algorithm [5]. This algorithm is both fast and robust. However, it requires a significant amount of memory for the tetrahedralization process because it is not an out-of-core algorithm. We compare our algorithm to TetGen because our algorithm depends, at its core, on it. We aim to create a memory-efficient version of TetGen to tetrahedralize large meshes that do not fit into the memory by applying an out-of-core approach on top of it.

3 Algorithm

3.1 Overview

Our algorithm is an out-of-core divide-and-conquer algorithm for constrained tetrahedralization. It, at its core, makes use of the TetGen software [5]. Our algorithm divides the input mesh into two as long as the memory is not enough

Algorithm 1 Our algorithm

```
1: procedure TETRA(vertices, faces)
2:     if CALCULATE_EXPECTED_MEMORY(vertex_count) ≤ memory_limit then
3:         TETGEN(vertices, faces)
4:     else
5:         left_vertices, left_triangles, right_vertices, right_triangles
6:             = CLIP(vertices, faces)
7:         left_mesh_file = TETRA(left_vertices, left_triangles)
8:         right_mesh_file = TETRA(right_vertices, right_triangles)
9:         output_mesh_file = MERGE(left_mesh_file, right_mesh_file)
10:        return output_mesh_file
11:    end if
12: end procedure
```

for it. We calculate the expected memory usage using linear regression. If the mesh fits into the memory, then we use TetGen with the mesh as input to generate the tetrahedral mesh. Otherwise, we divide the mesh into two pieces by a plane passing through the mean of the most variant axis. We then recursively apply the same procedure to the two parts. When the tetrahedralizations of the parts are complete, we merge these tetrahedral meshes into one tetrahedral mesh as the last step. Algorithm 1 provides thee pseudo-code of the algorithm.

Further, our algorithm can generate a bounding box for the input object so that the space around the object can be tetrahedralized, which is especially convenient for Ray Tracing accelerators.

3.2 Expected Memory Calculation

We observe the memory consumption of TetGen with several models and generated a linear regression model to predict the memory consumption of an input object. The first two columns of Table 1 show the vertex count of each object and the real

Table 1 Memory requirement observations for TetGen in Megabytes (MB). We provide the actual memory requirements and the estimated values for various models of different vertex counts using our linear regression model

Vertex count	Actual memory requirement	Expected memory requirement
1440	7.03	9.70
2880	13.97	16.28
34,560	167.67	161.04
112,220	514.80	515.91
172,971	792.97	793.51

memory usage when that object is tetrahedralized. We set up a linear regression model using this data and we ended up with the following equation:

$$y = 4.57 \times 10^{-3} X + 3.12,$$

where y is the expected memory requirement in Megabytes (MB), and X is the number of vertices of the input mesh. In the table, the last column corresponds to the expected memory requirement calculated using the above equation.

3.3 Subdivision Stage

To decrease the problem size, the input mesh must be divide into two pieces at each recursion level. We are dividing the input using the plane that passes through the mean of the most variant axis. Dividing the object requires a significant effort because after the division the cut surfaces of each object must match so that the resulting tetrahedral mesh of each side matches. Matching triangles will guarantee a match in the resulting tetrahedral mesh because we are applying constrained tetrahedralization.

To this end, we use the clip function of CGAL [7]. It takes input mesh and a plane as input, and slices the mesh using the plane and returns the positive side. It also allows us to triangulate the open-surface. We use it the following way: we first clip the object and get the surface triangulation. Then, for the second half, we again clip the object but not triangulate the surface. Instead, we just copy the triangulation of the first part and paste it to the second part. In that way, we guarantee that the triangulation at both sides will match. However, copy-pasting may lead to duplicate vertices and open borders. Therefore, we propose a repairing algorithm to eliminate these defects.

The way we clip the object prevents the tetrahedra near the clipping plane to breach the Delaunay criterion. This is because after each part is tetrahedralized we cannot guarantee that the circumsphere of the tetrahedra around the plane will contain points from the other side. Since while one half is tetrahedralized the other half is not considered, the Delaunay criterion might be violated. Yet, the magnitude of the violation depends on the number of tetrahedra around the plane and the number of times the object is divided. While choosing the plane, we make sure that the most variant axis is chosen, to also decrease the number of tetrahedra around the plane.

3.4 Repairing Stage

We introduce steps that we apply to eliminate defects in the meshes produced during the subdivision stage. Two defects that may arise during the clipping stage are

overlapping vertices and *overlapping edges*. Specifically, both problems occur because of copying the triangulation of the left side and pasting it to the right side. Faces cannot overlap because we copy the triangles generated for the left side to the open-boundary of the right side.

Overlapping vertices occur because when we copy the triangulation, the vertices of the left surface triangulation might coincide with the border vertices of the right mesh. We can repair overlapping vertices by iterating through all of the triangles around the dividing plane and checking if any of its vertices have any duplicates. If this is the case, we keep only one of the vertex and ignore the other one.

Edges may overlap after copying the triangulation from one side to the other. When a vertex from the left side does not have the corresponding vertex at the right side; then, this vertex would coincide with one of the edges of the right side. The clip function does not insert the same vertices to both sides, which causes this problem.

Figure 1a shows an example of overlapping edges. There are three triangles: ABC, BED, and DEC. Here ABC is an existing triangle of the mesh, but during the clipping operation, we can add the other two faces so that two halves match as described above. The problem here is the edge BC overlaps both the edges BD and DC. To fix this, we form triangles ABD and ADC and remove ABC (see Fig. 1b). We repair only the right side because we copy the triangles of the left side over the right side. Hence, only the topology of the right side is disturbed.

We repair overlapping edges as follows. We iterate over all edges in the right mesh that are close to the dividing plane. For each face in this region, we iterate over all of the vertices of the mesh on the right. If the edge contains a vertex between its terminal points, it means that this vertex is leading to an overlapping edge. We make use of the point-line segment distance to find the overlapping between a vertex and edge. Afterward, the triangle containing this edge is fixed, as shown in Fig. 1.

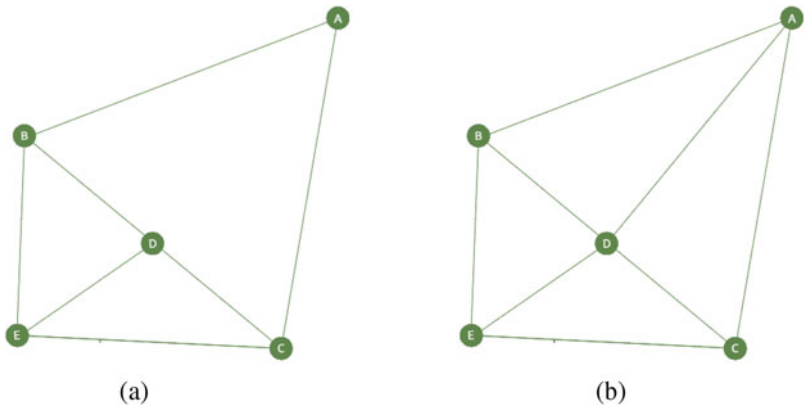

Fig. 1 Handling overlapping edges. (**a**) Overlapping edges. (**b**) After repairing overlapping edges

Algorithm 2 Merge procedure

```
procedure MERGE(left_mesh_file, right_mesh_file)
    # Concatenate vertices and tets
    CONCATENATE_VERTICES(left_mesh_file, right_mesh_file, output_mesh_file)
    CONCATENATE_TETS(left_mesh_file, right_mesh_file,output_mesh_file)
    # Find missing neighbors
    left_centroids = GET_CENTROIDS(left_mesh_file)
    left_centroids_grid = GENERATE_GRID(left_centroids)
    right_centroids = GET_CENTROIDS(right_mesh_file)
    for each right_centroid ∈ right_centroids do
        if right_centroid ∈ left_centroids_grid then
            ADD_NEIGHBOUR(right_tet, left_tet, output_mesh_file)
        end if
    end for
    return output_mesh_file
end procedure
```

3.5 Merging Stage

The merge step is another non-trivial step for our algorithm because it is the place where we finally produce the resulting tetrahedral mesh. In the merge step, we first merge two tetrahedral meshes into one mesh as depicted in Algorithm 2. That step simply involves concatenating two text files. Secondly, we also extract neighbor relations between tetrahedra. In the end, we generate a final *output_mesh_file* consisting of vertex locations, tetrahedra, and neighborhood information between tetrahedra. We put a non-trivial effort to find neighborhood information across the two pieces of the object after subdivision. After we cut the object into two pieces and tetrahedralize each piece, the neighbor relations around the cut faces are missing and we find those as well.

3.5.1 Spatial Hashing

We use three-dimensional *Spatial Hashing*. Specifically, the dimensions of the hash grid we have used are $50 \times 50 \times 50$ corresponding to 125,000 cells. We begin by putting the centroid of faces of the left piece that coincide with the cut plane, into the 3D grid. Then, we iterate over the faces of the right piece and find if their centroids match any of the centroids of the left piece by finding the corresponding cell and iterating through the centroids inside of it. If there is a match, it means that two tetrahedra share a common triangle and they must be neighbors. Then, we save this new neighborhood information.

3.5.2 Merging Time Complexity

Let V_L, V_R be the number of vertices, F_L, F_R be the number of faces and T_L, T_R be the number of tetrahedra of left and right pieces. Merging files will take $\theta(V_L + V_R + T_L + T_R)$ time. The time complexity of finding missing neighbour relations through *Spatial Hashing* will take $\theta(F_L + F_R)$ time on average. Specifically, we, first, iterate through all faces of left piece to putting the centroids in the grid taking $\theta(F_L)$ time. Then, we iterate through all faces in the right piece, $\theta(F_R)$ time, and for each face we search the centroid inside the grid which takes $\theta(1)$ time thanks to hash structure. Hence, overall, it takes $\theta(F_R) * \theta(1) = \theta(F_R)$ time. As a result, overall time complexity of, merge step is $\theta(V_L + V_R + T_L + T_R + F_L + F_R)$. The time complexity function can be further simplified using the fact that $V <= 3 * F$ and $V <= 4 * T$ to end up with $\theta(V_L + V_R)$.

4 Experimental Results

4.1 Runtime and Memory Results

In this section, we present the results of constrained tetrahedralization of several objects using both our algorithm and TetGen to provide the statistics of memory consumption and execution times. Our algorithm performs differently based on the intersection of the cutting plane with any object in the input mesh. If the plane touches an object, then our algorithm will run a repairing procedure. Otherwise, it skips the repairing procedure. We conduct the experiments on a high-end computer with an Intel Xeon E5-2620 2.10 GHz processor and 64 GB of RAM. In the experiments, for simplicity, we divide the input mesh into two parts but not further. For each experiment, we monitor the Windows Task Manager and record the peak memory usage. Hence, we report the physical memory usage.

Table 2 shows the statistics of computation time and memory consumption where each row contains the results of the experiment specified in the first column. We provide details about each experiment in the sequel.

Experiment 1: In that scene, because the plane that divides the scene into two halves intersects the armadillo, we apply the time-costly repairing step. This scene can be tetrahedralized with both methods. Figure 2a shows the resulting mesh.

Experiment 2: In this experiment, we encapsulate the armadillo with a bounding box so that the space around the armadillo can be tetrahedralized. This experiment also requires a repair step because the plane intersects the bounding box. Figure 2b shows the resulting mesh where the armadillo is at the center of the bounding box.

Experiments 3, 4, and 5: In these three experiments, the plane does not intersect any object, and hence the repairing stage is not applied. Consequently, both methods can tetrahedralize this scene. Figure 2c–e show the resulting mesh for Experiments 3, 4, and 5, respectively.

Table 2 Experimental results on the computer with Intel Xeon E5-2620 2.10 GHz processor and 64 GB of RAM. Execution time is in seconds (s) and the memory usage is in Megabytes (MB)

Experiment no.	No. vertices	No. faces	TetGen		Ours	
			Time	Memory	Time	Memory
1	172,969	345,938	37.26	850	429.44	483
2	172,969	345,938	66.54	947	456.67	564
3	345,938	691,876	43.96	1 700	138.20	922
4	1,346,688	2,693,376	294.93	6605	727.89	3584
5	1,704,146	3,408,292	380.74	8359	642.10	4537
6	17,682,248	35,364,496	91,101.46	53,160	7047.26	46,614
7	27,164,160	54,328,320	N/A	N/A	22,221.27	58,964

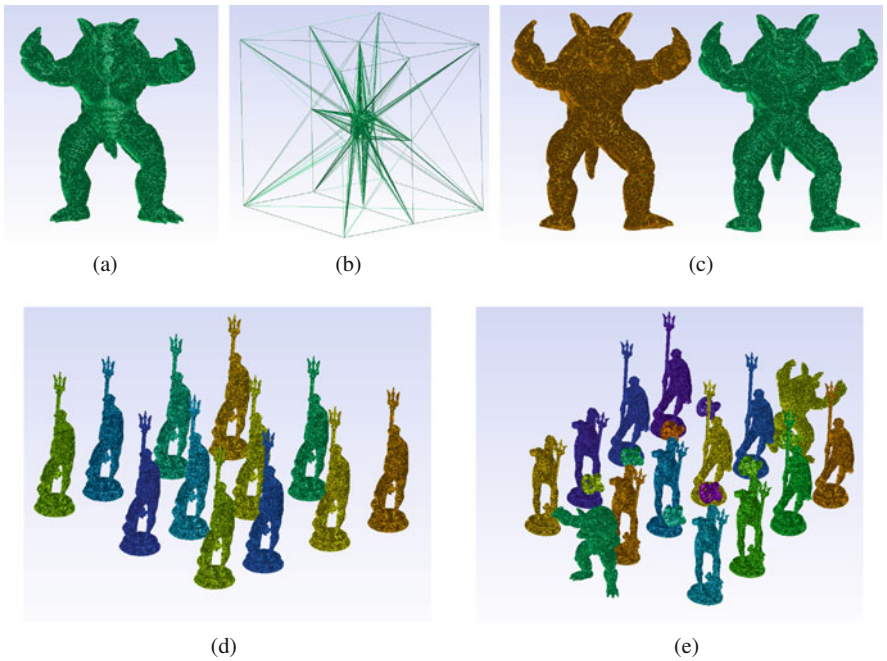

Fig. 2 Generated tetrahedral meshes. (**a**) Experiment 1. (**b**) Experiment 2. (**c**) Experiment 3. (**d**) Experiment 4. (**e**) Experiment 5

Experiment 6: This experiment takes around 117 min and consumes 47 GB of RAM with our algorithm. On the other hand, TetGen tetrahedralizes in 25 h using around 53 GB of memory. TetGen frequently make use of virtual memory to complete its task. Memory footprint shows that its virtual memory usage goes up to 90 GB (out of 130 GB). Yet, this requires an abundance of disk accesses that slows down the process. We observe that TetGen made around 200 page-faults per second, which is the main reason for the slowdown.

Experiment 7: This experiment takes around 6 h and consumes around 59 GB of RAM with our algorithm. Yet, TetGen cannot successfully tetrahedralize after 4 days of execution and force the computer to restart. TetGen uses almost all of the physical memory and consumes all of the 130 GB of virtual memory, which still is not sufficient for its execution. Hence, it cannot achieve to complete the task.

In our experiments, we observe that our method can tetrahedralize using less memory than TetGen. In some experiments, TetGen either takes more time to complete than ours or cannot complete its execution at all. TetGen continuously allocates memory as much as it needs without considering the availability. Hence, the operating system consistently provides it with memory as long as it fits into physical and virtual memory. There are two memory thresholds for TetGen. These are available physical and virtual memories. When these memories are sufficient, TetGen runs fast, as expected. If the physical memory is exhausted, the operating system allocates from virtual memory. In this case, TetGen starts making page faults because its working set cannot fit into physical memory. It directs a large portion of memory accesses to disk, which slows down TetGen. Finally, if the memory requirement of TetGen exceeds even the virtual memory, then the operating system can no longer provide memory to the TetGen, and the computer freezes and restarts itself. This phenomenon happens because TetGen lacks a mechanism to track the expected memory usage and readjust itself to stay below the memory threshold, whereas our algorithm has this mechanism.

4.2 Quality Results

In this section, we present results on the quality of the tetrahedra generated by our algorithm by comparing it with the results of TetGen. We use the aspect ratio metric used to measure the quality of tetrahedron in TetGen [5]. We calculate this metric by dividing the longest edge by the smallest height. A low aspect ratio implies high-quality tetrahedralization. As seen in Table 3, the average tetrahedron quality of TetGen is higher than our method in all cases. In torus knot, the result of TetGen is around three times better than ours, while in Armadillo, it is 1.76 times, and in Neptune, 1.48 times better on the average. Although our algorithm could generate better quality tetrahedra for Armadillo and Neptune according to the

Table 3 Experimental results on the quality of the tetrahedral mesh based on the aspect ratio metric

Model name	No. vertices	No. faces	TetGen			Ours		
			Min.	Max.	Ave.	Min.	Max.	Ave.
Torus knot	1440	2880	1.90	125.74	7.52	2.01	10,365.34	22.20
Neptune	112,224	224,448	1.30	536.49	8.72	1.28	250,088.04	12.92
Armadillo	172,969	345,938	1.30	262,232.06	7.31	1.27	260,203.69	12.84

minimum aspect ratios, overall, our tetrahedral meshes seem to be of worse quality. We expect this quality degradation because of the increase in the surface area of the object and the number of tetrahedra on the surface as we divide the point cloud into two parts. The circumspheres of these tetrahedra might extend outside the object boundary because it will not contain any point, thereby satisfying the Delaunay criterion. Hence, the quality of these tetrahedra might be reduced without breaching the constrained Delaunay criterion (see [2] for constrained Delaunay criterion).

5 Discussion and Future Work

We propose an out-of-core constrained tetrahedralization algorithm for tetrahedralizing large three-dimensional scenes. We have shown that our algorithm uses the memory more efficiently than TetGen and can tetrahedralize meshes that TetGen is unable to do because of insufficient memory. In essence, TetGen does not aim to use memory efficiently. Its main goal is computational efficiency. Therefore, TetGen tetrahedralizes meshes faster than our method if the main memory is sufficient. Our algorithm uses a divide-and-conquer approach and TetGen. In this way, we could create a memory-optimized version of TetGen by compromising the execution time.

Our algorithm divides the scene into two halves at each step, tetrahedralizes them, and finally merges them into a single tetrahedral mesh. Yet, our algorithm does not guarantee that the tetrahedra around the cut region satisfy the Delaunay criterion. In other words, the circumspheres of some tetrahedra around the cut region might contain vertices of other tetrahedra. In fact, we observed that the overall quality of the tetrahedral meshes generated by our algorithm is lower than TetGen. Hence, We are looking forward to adding a refinement process to satisfy the Delaunay criterion around the division region. When we add this step, tetrahedral mesh construction time will be longer, but the quality of the tetrahedra around the cut region will increase.

Although we use TetGen to apply constrained tetrahedralization, the clip procedure may introduce new vertices and faces on the objects in the scene where they intersect the plane. Nevertheless, it only divides a face into smaller parts, and hence the faces on the final tetrahedral mesh cover the input triangle soup.

Our algorithm works faster if the dividing plane does not intersect with any of the objects. If the dividing plane intersects the objects in the input mesh, the tetrahedralization requires costly repairing operation. If we choose the dividing plane carefully, we can tetrahedralize the mesh faster. Hence, a better dividing plane finding algorithm would be employed to avoid the intersection of the dividing plane with the objects in the scene. The dividing plane does not need to be planar; it could be an arbitrary polynomial surface or a curved surface.

Currently, our algorithm does not take the mesh density into account. Its performance in the case of a mesh with high varying density is dependent on the behavior of TetGen. A possible extension to tetrahedralize meshes of highly varying

density is to take the mesh density function as input and balance the partitions during the subdivision stage in terms of mesh density.

Acknowledgments This research is supported by The Scientific and Technological Research Council of Turkey (TÜBİTAK) under Grant No. 117E881.

References

1. Blelloch, G.E., Miller, G.L., Talmor, D.: Developing a practical projection-based parallel Delaunay algorithm. In: Proceedings of the 12th Annual Symposium on Computational Geometry, SCG '96, pp. 186–195. ACM, New York (1996)
2. Chew, L.P.: Constrained Delaunay triangulations. Algorithmica **4**(1–4), 97–108 (1989)
3. Cignoni, P., Montani, C., Scopigno, R.: DeWall: a fast divide and conquer Delaunay triangulation algorithm in E^d. Comput.-Aided Des. **30**(5), 333–341 (1998)
4. Lagae, A., Dutré, P.: Accelerating ray tracing using constrained tetrahedralizations. Comput. Graph. Forum **27**(4), 1303–1312 (2008)
5. Si, H.: TetGen, a Delaunay-based quality tetrahedral mesh generator. ACM Trans. Math. Softw. **41**(2), 1–36 (2015)
6. Smolik, M., Skala, V.: Fast parallel triangulation algorithm of large data sets in E^2 and E^3 for in-core and out-core memory processing. In: Proceedings of the International Conference on Computational Science and Its Applications, ICCSA '14, pp. 301–314. Springer, Berlin (2014)
7. The CGAL Project: CGAL User and Reference Manual, 5.0.2 edn. (2020). https://doc.cgal.org/5.0.2/Manual/packages.html
8. Woop, S., Schmittler, J., Slusallek, P.: RPU: a programmable ray processing unit for realtime ray tracing. ACM Trans. Graph. **24**(3), 434–444 (2005)