A Simple and Robust Approach to Computation of Meshes Intersection

Věra Skorkovská[1], Ivana Kolingerová[1] and Bedrich Benes[2]
[1]*Department of Computer Science and Engineering, University of West Bohemia, Univerzitní 8, 306 14, Plzeň, Czech Republic*
[2]*Department of Computer Graphics Technology, Purdue University, 401. N. Grant Street, 47907-2021, West Lafayette, IN, U.S.A.*

Keywords: Triangular Mesh, Mesh Repair, Intersections, Neighbor Tracing.

Abstract: Triangular meshes are important in many fields in both basic and applied research that rely on their correctness and accuracy. Many operations with meshes can lead to undesirable situations and the resulting models can be damaged and further unusable. Self-intersection and mesh-to-mesh intersection are types of operations that are often present and can cause such problems. We propose an accurate geometry-based method for local repair of intersecting meshes. The state-of-the-art methods either solve the problem inaccurately, or use methods such as arbitrary precision arithmetic or virtual perturbation to deal with the troublesome boundary cases. Our method represents a robust way to repair intersecting meshes accurately without the need to manipulate with the input data or to employ arbitrary precision arithmetic. The correct solution is obtained through a careful classification of the cases that could result from a numerical imprecision of the floating point arithmetic.

1 INTRODUCTION

3D surface models are often used in various fields, ranging from animation and gaming industry to architectural design and CAD. Applications dealing with 3D models usually require correct meshes. Depending on the correctness definition, many types of defects can be found in polygon meshes that have to be repaired before any further use of the model. For a thorough survey on the defect types and their repair we refer to (Attene et al., 2013; Botsch et al., 2007).

This paper addresses the problem of mesh-to-mesh intersections and self-intersections. These intersections are unacceptable in many applications, for example when the mesh represents a boundary of a solid volume. The existing algorithms addressing this problem are usually designed to be either robust, efficient, or accurate. To the best of our knowledge, no method grants all the three desired properties. The accuracy of the mesh-repair methods is probably the most problematic and ignored topic, although it is the main interest in applications such as CAD modeling or exact physics simulations.

We propose a novel method for repair of intersecting meshes. Our algorithm uses a neighbor tracing algorithm (Lo and Wang, 2004; McLaurin et al., 2013), works with floating point arithmetic, and has a high accuracy achieved via a careful classification of all cases of intersections possibly caused by the numerical imprecision. The method deals with both self-intersections of a single mesh and intersections of two meshes. The method does not use virtual perturbation to avoid singular cases which makes it suitable for applications where alterations of the input model are undesirable. Our solution represents a robust way of handling mesh (self-)intersections in applications where accuracy of the solution is the main concern.

Our algorithm first detects the intersection boundary. In the second step the triangles containing the boundary are cut along the boundary, dividing the triangles into a valid part that is a part of the repaired mesh, and an invalid part that needs to be removed. The invalid parts are then discarded and the valid parts are triangulated. Finally, the valid parts of the mesh are stitched together. The correct identification of the intersection boundary in the first step is essential for the correct behavior of the rest of the algorithm and it is the main contribution of this paper.

Figure 1 shows an example of a result generated by our algorithm. The scene consists of six Stanford dragons and six Stanford bunnies (Levoy et al., 2005) placed at random positions with random rotation. 172 intersection boundaries were identified in the overlapped meshes by the first step of our algorithm. The detected intersections were repaired and the resulting output mesh does not contain any intersections.

Skorkovská, V., Kolingerová, I. and Benes, B.
A Simple and Robust Approach to Computation of Meshes Intersection.
DOI: 10.5220/0006538401750182
In *Proceedings of the 13th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2018) - Volume 1: GRAPP*, pages 175-182
ISBN: 978-989-758-287-5

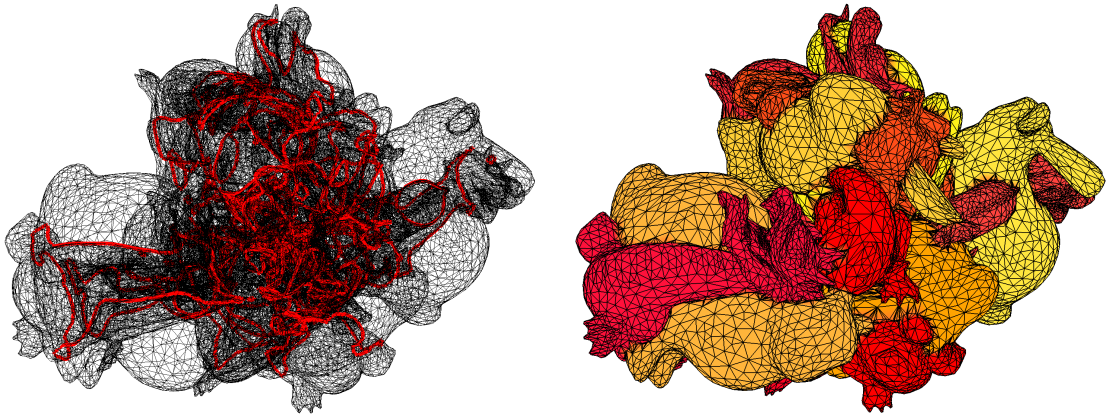

Figure 1: Six Stanford dragons and six Stanford bunnies (Levoy et al., 2005) intersect in 172 separate intersection boundaries (left). The final merged mesh (right) does not contain any intersections.

2 RELATED WORK

Many methods for repairing mesh intersections and self-intersections have been proposed. The approaches can be divided into two main categories: *global* (*volumetric*) and *local* (*surface oriented*) approaches (Attene et al., 2013).

Global approaches convert the polygonal mesh to an intermediate representation which is then used to generate a new valid mesh. Early methods utilize volumetric representation (Andújar et al., 2002; Nooruddin and Turk, 2003). While being effective at repairing the defects, this approach is sensitive to the resolution of the intermediate structure. Adaptive approaches exist that change the resolution of the intermediate structure according to the required detail (Ju, 2004; Bischoff et al., 2005). Binary space partition trees (Murali and Funkhouser, 1997), level set methods (Osher and Fedkiw, 2002; Enright et al., 2002) and deformable simplicial complexes (Misztal and Bærentzen, 2012; Christiansen et al., 2014) have been also used to solve this problem. The use of global approaches is appropriate if the mesh is highly inconsistent. It typically allows creating very robust methods at the cost of lower accuracy and efficiency.

Local approaches work directly with the input mesh and identify individual self-intersections which are then repaired locally, usually one at a time, and leave the rest of the mesh unchanged. These approaches change the input mesh as little as possible which is desirable in applications where accuracy of the solution is the main interest. Local approaches are more suitable when the intersections are located only at isolated parts of the mesh.

Some local approaches take a global method and apply it only in the local scale (Bischoff and Kobbelt, 2005; Attene, 2010; Wojtan et al., 2009). These methods depend highly on the grid resolution and can also lead to changes in the volume encompassed by the mesh. Other group of methods does not rely on the use of an intermediate data structure but works directly with the mesh. Zaharescu et al. (Zaharescu et al., 2011) search the mesh for partially valid faces and locally repair them. Others (Lo and Wang, 2004; McLaurin et al., 2013) use neighbor tracing to speed up the process of finding intersected faces and to achieve reliable results. A similar approach is used to simulate the interaction of dynamic meshes (Brochu and Bridson, 2009) and to simulate interactions between solids and liquids (Clausen et al., 2013). These geometric approaches often use exact arithmetic to achieve numerical stability of the intersection calculations, while boundary cases are avoided by using virtual perturbation method.

The above-mentioned methods usually concentrate on either being robust, accurate, or efficient. To the best of our knowledge, none of them satisfy all the three properties. We focus on the accuracy of the solution as it is very often overlooked.

The most accurate solutions can be found among local geometric solutions. They usually use vertex displacement or arbitrary precision arithmetic to improve the robustness. However, the vertex displacement changes the input mesh which can be unacceptable whereas the use of arbitrary precision arithmetic leads to lower efficiency of the method. If the performance of the algorithm is crucial, the use of floating point arithmetic may be preferred.

Our goal is to propose an accurate method for repairing (self-)intersections working with floating point arithmetic. The proposed solution ranks among local methods and extends the neighbor tracing algorithm (Lo and Wang, 2004; McLaurin et al., 2013) with the emphasis on the accuracy of the solution.

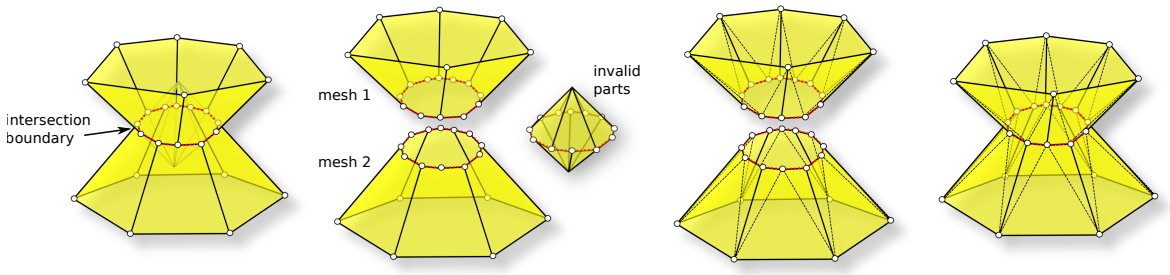

Figure 2: Left to right: intersection boundary detection, invalid parts elimination, polygon triangulation, and mesh stitching.

3 PROPOSED SOLUTION

Figure 2 shows an overview of our method. The input are two triangle meshes (or two nonadjacent parts of a single mesh) which are examined for intersection. In the first step we identify the intersection boundary, then cut both parts of the mesh along the boundary, triangulate the newly generated polygons, and then we stitch the mesh together.

Our method for finding the intersection boundary is based on the neighbor tracing algorithm (tracing the neighbors of intersecting triangles - TNOIT) by Lo and Wang (Lo and Wang, 2004) with further emphasis on the correct solution of singular cases of intersection.

The TNOIT algorithm (Lo and Wang, 2004) repairs mesh intersections only if the intersection segments of the boundary are calculated precisely enough to be correctly classified. The TNOIT algorithm fails when an intersection is not determined correctly. However, such cases are very common in real-world situations.

The problem can be alleviated by increasing the precision of the floating point arithmetic as shown in (Karasick et al., 1991). However, this decreases the algorithm performance and may limit time-critical applications such as real-time simulations. Moreover, it only shifts the problems to higher frequencies. Another way of solving the problem is to use a virtual perturbation method (e.g., Simulation of Simplicity (Edelsbrunner and Mücke, 1990)) to slightly displace the vertices of the mesh, avoiding the boundary cases altogether. However, the vertex displacement can be unacceptable in applications where a precise solution is needed.

We address this issue by a careful inspection of all the cases which can occur due to the imprecise calculation of the intersection, while using only floating point arithmetic. Our solution represents a robust way of handling mesh (self-)intersections in applications where the accuracy of the solution is the main concern.

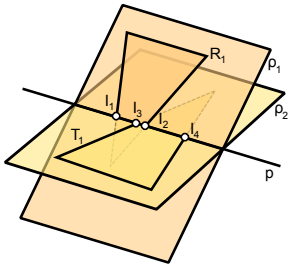

Figure 3: Nonparallel planes intersecting at the line p.

3.1 Numerical Imprecision Problem

Let us open the description by an analysis of a case where numerical imprecision could cause problems. Let ρ_1 and ρ_2 be two nonparallel planes intersecting at line p. Triangles T_1 and R_1 in these planes intersect only if they intersect the line p and if the corresponding intersection segments overlap. An example in Figure 3 shows this case; triangle T_1 intersects the line p in segment I_1I_2, triangle R_1 intersects p in segment I_3I_4. The line segments overlap, i.e., the line segment I_3I_2 is the intersection of T_1 and R_1. To correctly determine the intersection of the triangles, we need to identify the corresponding intersecting segments. Considering a triangle T located in the plane ρ and a line p in the same plane, there are five ways the triangle T can intersect the line p (see Figure 4):

1. Line p does not intersect the triangle T.
2. Line p intersects the triangle T at two of its edges.
3. Line p intersects the triangle T at a vertex.
4. Line p intersects the triangle T at a vertex and the opposite edge.
5. Line p intersects the triangle T at two of its vertices and the edge formed by these two vertices is in the line p.

The cases 1-2 can be handled even in the context of floating point arithmetic, but the situation is more

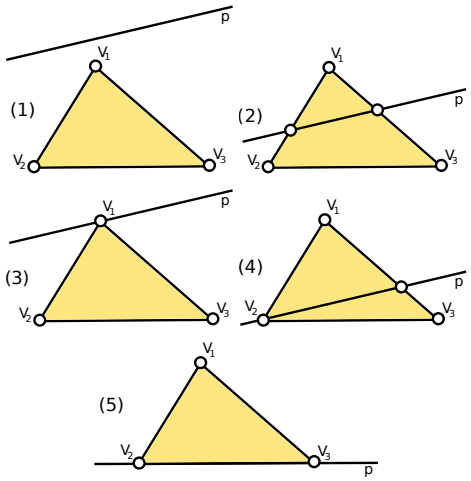

Figure 4: Intersections of a triangle and a line. Left to right, top to bottom: no intersection, intersection at two edges, at a vertex, at a vertex and an edge, and at two vertices.

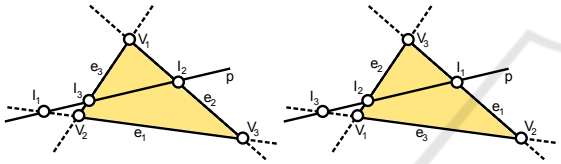

Figure 5: Intersections of line and triangle edges. Left: an example situation, right: vertices rotated counterclockwise.

difficult when we get closer to the boundary cases, because even a small error can cause an incorrect classification of the intersection cases.

We suggest a solution to this problem based on a careful classification of all the possible cases which can be caused by the incorrect calculation of the intersection. For each pair of triangles, we first identify the intersection line p (Figure 3). Then we calculate the intersection segment for each triangle. The intersection line p is guaranteed to lie in the same plane as the triangle, allowing us to compute the intersections of the line p and all edges of the triangle. An example of the situation is shown in Figure 5. Line p can intersect the line containing an edge in three ways: intersection lies outside the edge (I_1), at a vertex, or inside the edge (I_2, I_3).

Based on the position of the intersection points I_1, I_2, I_3, we classify the situation into one of the cases depicted in Figure 4. The problem gets more complicated when we take into consideration the fact that some (or all) of the intersection points might not have been determined correctly due to the numerical imprecision of the floating point arithmetic. We introduce the following substitute symbols for the relative position of the intersection points:

- 0 - the intersection point lies outside the edge,
- 1 - the intersection point lies at a vertex, and
- 2 - the intersection point lies inside the edge.

The order of the points is not relevant for the classification and so we can sort the symbolic representation in the ascending order. Using this notation, we can describe the situation from Figure 5 (left) as 022 - I_1 lies outside the edge e_1, I_2 and I_3 lie inside the edges e_2 and e_3 respectively. Let us consider a similar situation where the vertices of the triangle are rotated counterclockwise (Figure 5 (right)). The description of the situation using the substitute symbols would be 220. As mentioned above, we can sort the symbolic representation without the loss of generality, getting the same representation (022) for both cases. This simplification leaves us with 10 cases to address:

- 000 - all intersections outside the edges → case 1,
- 001 - impossible case,
- 002 - impossible case,
- 011 - one intersection outside, two intersections at vertices,
  - two intersections at the same vertex → case 3,
  - intersections at different vertices → case 5,
- 012 - impossible case,
- 022 - one intersection outside, two intersections inside the edges → case 2,
- 111 - three intersections at vertices → case 5,
- 112 - one intersection inside, two intersections at vertices,
  - two intersections at the same vertex, one intersection at the opposite edge → case 4,
  - intersections at two different vertices and on the edge between them → case 5,
- 122 - impossible case, and
- 222 - impossible case.

The *impossible cases* cannot occur as a result of a correct calculation. For these cases, no such straight line exists that all the intersection points would lie on it. An example of this situation is in Figure 6 where the line p intersects edges e_2 and e_3 close to vertex V_1. However, the incorrect calculation of the intersection puts the point I_3 outside e_3 whereas the point I_2 inside e_2, causing a contradiction with the original assumption that all the intersection points lie on the line p (points I_1, I_2 and I_3 are not collinear).

If the classification of the intersection ends up as one of the impossible cases, we know that the results must have been caused by numeric imprecision. In such a case we need to correct the results before the neighbor tracing algorithm can continue.

Even if the result of the classification is a valid option, some numeric error might have been introduced.

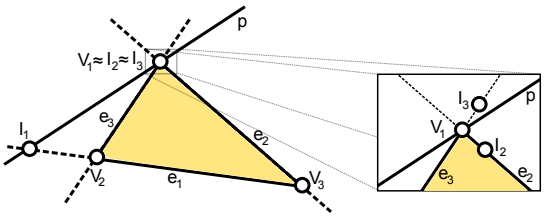

Figure 6: Impossible case of the intersection points as a result of numeric imprecision during the calculations.

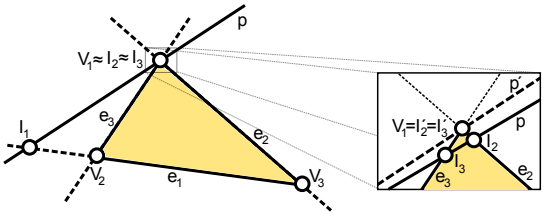

Figure 7: Incorrect intersection classification. The computed intersections I_2 and I_3 lie inside the edges while the correct intersections I_2' and I_3' lie at the vertex V_1.

The numeric errors during the calculation can cause an incorrect classification. Figure 7 shows an example of such a situation. Line p intersects the edges e_2 and e_3 at the vertex V_1 but the computed intersection points I_2 and I_3 are located inside the corresponding edges. This error causes an incorrect classification as the case 022 instead of the correct case 011.

We cannot tell if the classification is valid if any of the intersection points lies close to a vertex. However, if the mesh does not contain any nearly degenerate triangles with edges shorter than the maximum error ε, we can continue the neighbor tracing. In such a case we can never stray away from the exact solution farther than to the direct neighbor. Figure 8 shows an example of such a situation where several incorrect classifications took place in a row. The green dashed line marks the exact intersection boundary while the red dotted line represents the computed intersection boundary affected by the computational error. The exact intersection is close to a vertex. The numerical imprecision causes an incorrect classification of the intersection with triangle T_6, leading the intersection boundary to T_5 instead of T_1. In an unlikely scenario it is possible for several incorrect classifications to chain, e. g., as in Figure 8, where the intersection is incorrectly classified for triangles T_6, T_5, T_4 and T_3. Triangles T_3, T_4 and T_5 should not be a part of the intersection at all, but they are included as a result of the numeric error of the calculation.

Subsequently, no intersection is found for triangle T_2. The failure to identify the intersection suggests that the previous segment or segments of the intersection boundary were affected by the numerical imprecision. Yet thanks to the condition we put on the input triangle mesh (no nearly degenerate triangles), we know the correct solution is going through a neighboring triangle. In such a case we have to test other possible classification solutions to find the right triangle where the intersection boundary continues. We start with the original case that we got as a result of the classification of the intersection of triangle T and look for the next closest option until we find the pair of triangles where the intersection boundary of the two meshes continues. The same approach is used if the original classification results in an impossible case.

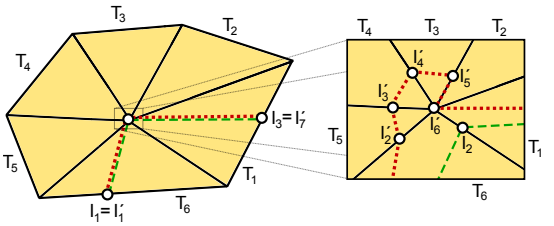

Figure 8: A chain of several incorrect classifications of the intersection. The green dashed line represents the exact intersection boundary, the red dotted line represents the computed intersection boundary.

3.2 Mesh Fixing

After the intersection boundary was found by the algorithm from Section 3.1, we can fix the mesh. We propose a novel method for repairing the mesh by using the detected intersection boundary. For each intersection segment found during the neighbor tracing, we store the pair of triangles which formed the intersection. This information is used during the repair step to fix the connectivity and topology of the mesh.

For each triangle participating in the intersection, a new valid polygon is created by cutting the inconsistent parts off (see Figure 9). To create the polygon, we first need to identify all the polylines that intersect it. We insert the polylines into an auxiliary data structure which helps to correctly trace the polygon as shown in Figure 10. We start with an empty list and insert the vertices of the triangle. We then insert the first point of each intersecting polyline into the list at the appropriate position corresponding to the line segment where the point is located in the original triangle. Finally, we connect the end points of the polylines with the vertices that come after them in the polygon boundary. Then we trace the boundary of the polygon: starting with any polyline, we trace the boundary following the pointers we set up in the auxiliary structure until we get back to the starting point. The points we visited form the boundary of the new polygon. If any polyline is left unvisited, the original triangle needs to be cut into more pieces. We repeat this process until all the polygons are found (Fig-

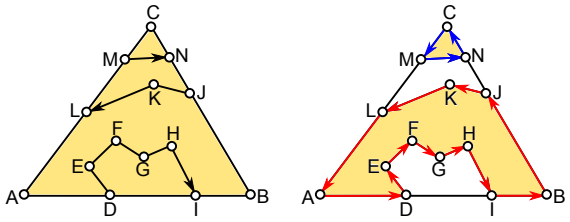

Figure 9: Polygon reconstruction. Intersecting polylines (left) and newly created polygons (right).

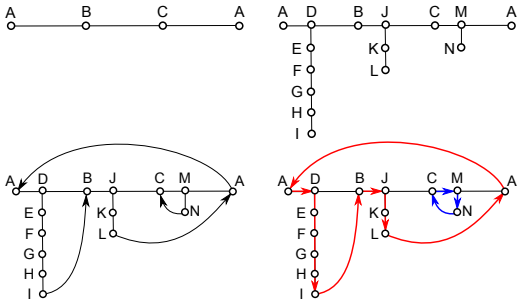

Figure 10: Left to right, top to bottom: list initiation; insertion of polylines; linking end points; polygon tracing.

ure 9). The order of the processing does not affect the outcome of the algorithm.

Then we use the ear cutting algorithm (Meisters, 1975) to triangulate the polygons created in the previous step. Finally, correct connectivity is restored through the shared intersection boundary.

4 RESULTS AND DISCUSSION

We tested our algorithm in various scenarios to show its robustness and correctness. Our implementation is a single-threaded C++ code and it was tested on Intel Core i7 clocked at 3.07 GHz with 8 GB RAM.

Our algorithm addresses the problem of numerical inaccuracy of floating point calculations which can affect the result of the neighbor tracing algorithm (Lo and Wang, 2004) when operating near the boundary cases. We created a simple scene to demonstrate the problem. A cuboid with edge ratio 2:2:2 intersects a cuboid with edge ratio 4:1:1 in Figure 11. The cuboids are aligned so that the intersection boundary is going exactly along the edges of the faces. The cuboids are rotated one degree along the x axis to magnify the inaccuracy problem. The position of some vertices cannot be represented exactly in floating point arithmetic after the rotation, amplifying the inaccuracy of the subsequent calculations.

The scene was set up in such a way so that we know the exact result of the intersection and can compare it with the results of our algorithm. The intersection boundary is aligned with the edges, so we can

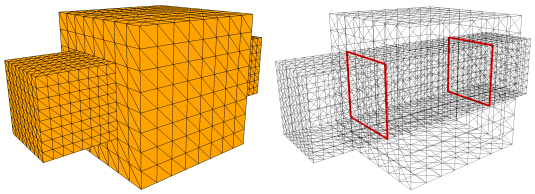

Figure 11: Two cuboids (left) and their intersection (right).

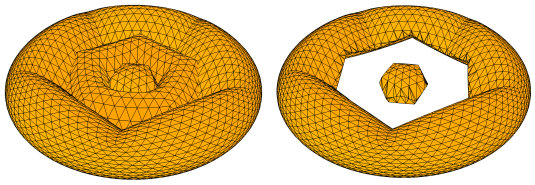

Figure 12: Input (left) and repaired model (right).

assume that every triangle participating in the intersection will have one of its vertices or edges lying on the intersection boundary. Using the notation introduced in Section 3.1, the result for each segment of the boundary should be 011, 111, or 112. However, the measured results differ from these expected values due to the numerical imprecision of the floating point arithmetic. Table 1 summarizes the results measured for the two cuboids with increasing resolution. For the example in Figure 11 consisting of 3,072 faces, the calculation results in impossible cases in 57 out of 128 instances. Furthermore, not even the valid results correspond to the expected outcome - none of the intersections was correctly identified as the cases 011, 111 or 112. Using the proposed method, we were able to repair the intersection despite the incorrect identification of the intersection cases. Existing methods (Lo and Wang, 2004; McLaurin et al., 2013) fail on these types of scenes unless exact arithmetic or virtual perturbation methods are used.

Our algorithm can also be used for other problems. It can detect and repair not only the intersection between separate objects, but also the self-intersections of a single model. Figure 12 captures such a self-intersecting object. The middle ring represents an inconsistent part of the mesh, where the normals are reversed as a results of the mesh overlap - the normals are pointing inwards instead of outwards. We can detect the two intersection boundaries using our algorithm and repair the mesh.

The algorithm can also easily be used to perform boolean operations on triangle meshes. Having two meshes A and B (Figure 13), we can perform boolean operations on them by setting the orientation of the normals of the mesh. As we cannot change the actual normals, we use *temporary* normals to obtain the desired behavior; these *temporary* normals are used during the repair of the mesh to determine, which part of

Table 1: The number of occurrences of possible intersection cases for the scene captured in Figure 11, with increasing resolution, using the notation introduced in Section 3.1.

faces	000	001	002	011	012	022	111	112	122	222	valid	impossible
336	11	8	0	0	5	8	0	0	0	0	19	13
3 072	33	1	55	0	1	38	0	0	0	0	71	57
49 152	116	0	201	0	5	188	0	0	0	0	304	206
196 608	224	2	364	0	0	426	0	0	0	0	650	366

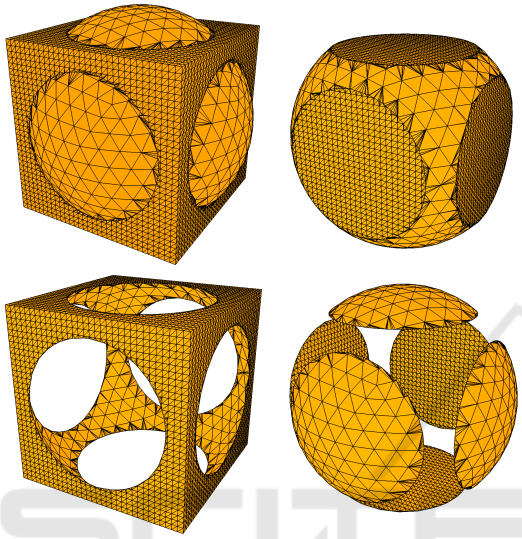

Figure 13: Boolean operation on cube A and sphere B. Left to right, top to bottom: $A \cup B$, $A \cap B$, $A \setminus B$, and $B \setminus A$.

Table 2: The resolution of the scenes.

	faces	vertices
Self-intersection (Fig. 12)	5 120	2 564
Boolean - union (Fig. 13)	13 568	6 790
Two cuboids (1) (Fig. 11)	49 152	24 582
Two cuboids (2) (Fig. 11)	196 608	98 310
Dragons & bunnies (Fig. 1)	76 716	38 372

the mesh should be discarded. To get the *union* $A \cup B$, the *temporary* normals are identical to the actual normals. On the contrary, the boolean operation *intersection* $A \cap B$ can be defined by setting all the *temporary* normals pointing inward the mesh. For the *difference* $A \setminus B$, the *temporary* normals of A are pointing outwards, while the normals of B are pointing inwards.

Table 2 contains the number of faces and vertices of the scenes presented in this paper. Table 3 shows the measured execution time. The measured data demonstrate that we usually find all the intersection boundaries long before we search the entire mesh, e.g., in the Two cuboids (2) example (Figure 11). Nevertheless, we cannot stop the calculation early, because we do not know if all the boundaries

Table 3: The execution time of steps of the algorithm (in ms). Description of the scenes can be found in Table 2.

	Find intersection boundaries	Search whole mesh	Fix mesh
Self-intersection	5	72	15
Boolean - union	24	221	43
Two cuboids (1)	63	394	33
Two cuboids (2)	24	1 454	76
Dragons & Bunnies	7 358	7 631	476

have been found until we finish searching the whole mesh. The Dragons & bunnies example (Figure 1) shows that for complex scenes with many separate intersection boundaries we have to go through most of the mesh to find all the intersections.

We do not focus on the speed of the method. We explore the capabilities of the method and its robustness. The examples presented in this section demonstrate the ability of the method to find the intersection boundary and repair the intersecting meshes even in the non-trivial cases, such as a scene containing many separate intersection loops (Figure 1) or a scene with many boundary cases (Figure 11).

5 CONCLUSION

We have proposed an approach for the repair of intersecting meshes based on the neighbor tracing algorithm (Lo and Wang, 2004; McLaurin et al., 2013) with the emphasis on the accuracy. Unlike the previous work, we do not use arbitrary precision arithmetic to achieve the accuracy of the solution, nor do we use virtual perturbation to avoid the undesirable boundary cases, as the change of input data can be unacceptable in some applications. Our method does not introduce any alteration of the input, works with floating point arithmetic and achieves accuracy through a careful classification of all sub-cases that could be caused by numerical imprecision. The neighbor tracing can be damaged by a numeric error near the boundary cases but for a mesh without nearly degenerate triangles, the intersection boundary can be correctly traced and re-

paired thanks to the classification of the intersection.

The method is designed for data in which we can expect a lot of boundary cases participating in the intersection boundary. We have tested the method on an example of two intersecting edge-aligned cuboids, where every segment of the intersection boundary represents a boundary case. We have also tested the method on a scene composed of several Stanford bunnies and Stanford dragons (Levoy et al., 2005) to demonstrate the correct behavior of the method in larger scene containing many intersection boundaries. Our experiments proved that the method can find the correct solution even in these non-trivial situations.

The correct behavior of the method can be guaranteed only if the maximum error ε of the calculation is smaller than the shortest edge of the mesh. For models that contain almost degenerate edges the algorithm may not be working correctly because the calculation error may corrupt the output of the method.

As the method works with single precision floating point arithmetic, it could be implemented on the GPU, where the higher precision operations can be very expensive. The transformation of the algorithm to be able to run it on the GPU is one of the possible avenues for this work.

ACKNOWLEDGEMENTS

This work has been supported by the project SGS-2016-013 - Advanced Graphical and Computing Systems.

REFERENCES

Andújar, C., Brunet, P., and Ayala, D. (2002). Topology-reducing surface simplification using a discrete solid representation. *ACM Trans. Graph.*, 21(2):88–105.

Attene, M. (2010). A lightweight approach to repairing digitized polygon meshes. *The Visual Computer*, 26(11):1393–1406.

Attene, M., Campen, M., and Kobbelt, L. (2013). Polygon mesh repairing: An application perspective. *ACM Comput. Surv.*, 45(2):15:1–15:33.

Bischoff, S. and Kobbelt, L. (2005). Structure preserving cad model repair. *Computer Graphics Forum*, 24(3):527–536.

Bischoff, S., Pavic, D., and Kobbelt, L. (2005). Automatic restoration of polygon models. *ACM Trans. Graph.*, 24(4):1332–1352.

Botsch, M., Pauly, M., Kobbelt, L., Alliez, P., Lévy, B., Bischoff, S., and Röossl, C. (2007). Geometric Modeling Based on Polygonal Meshes. This document is the support of a SIGGRAPH 2007 course.

Brochu, T. and Bridson, R. (2009). Robust topological operations for dynamic explicit surfaces. *SIAM Journal on Scientific Computing*, 31(4):2472–2493.

Christiansen, A. N., Nobel-Jørgensen, M., Aage, N., Sigmund, O., and Bærentzen, J. A. (2014). Topology optimization using an explicit interface representation. *Struct. Multidiscip. Optim.*, 49(3):387–399.

Clausen, P., Wicke, M., Shewchuk, J. R., and O'brien, J. F. (2013). Simulating liquids and solid-liquid interactions with lagrangian meshes. *ACM Trans. Graph.*, 32(2):17:1–17:15.

Edelsbrunner, H. and Mücke, E. P. (1990). Simulation of simplicity: A technique to cope with degenerate cases in geometric algorithms. *ACM Trans. Graph.*, 9(1):66–104.

Enright, D., Fedkiw, R., Ferziger, J., and Mitchell, I. (2002). A hybrid particle level set method for improved interface capturing. *Journal of Computational Physics*, 183(1):83–116.

Ju, T. (2004). Robust repair of polygonal models. *ACM Trans. Graph.*, 23(3):888–895.

Karasick, M., Lieber, D., and Nackman, L. R. (1991). Efficient delaunay triangulation using rational arithmetic. *ACM Trans. Graph.*, 10(1):71–91.

Levoy, M., Gerth, J., Curless, B., and Pull, K. (2005). The Stanford 3D Scanning Repository. `http://graphics.stanford.edu/data/3Dscanrep/`.

Lo, S. and Wang, W. (2004). A fast robust algorithm for the intersection of triangulated surfaces. *Engineering with Computers*, 20(1):11–21.

McLaurin, D., Marcum, D., Remotigue, M., and Blades, E. (2013). Repairing unstructured triangular mesh intersections. *International Journal for Numerical Methods in Engineering*, 93(3):266–275.

Meisters, G. H. (1975). Polygons Have Ears. *The American Mathematical Monthly*, 82(6):648–651.

Misztal, M. K. and Bærentzen, J. A. (2012). Topology-adaptive interface tracking using the deformable simplicial complex. *ACM Trans. Graph.*, 31(3):24:1–24:12.

Murali, T. M. and Funkhouser, T. A. (1997). Consistent solid and boundary representations from arbitrary polygonal data. In *Proceedings of the 1997 Symposium on Interactive 3D Graphics*, I3D '97, pages 155–ff., New York, NY, USA. ACM.

Nooruddin, F. S. and Turk, G. (2003). Simplification and repair of polygonal models using volumetric techniques. *IEEE Transactions on Visualization and Computer Graphics*, 9:191–205.

Osher, S. and Fedkiw, R. (2002). *Level Set Methods and Dynamic Implicit Surfaces (Applied Mathematical Sciences)*. Springer, 2003 edition.

Wojtan, C., Thürey, N., Gross, M., and Turk, G. (2009). Deforming meshes that split and merge. *ACM Trans. Graph.*, 28(3):76:1–76:10.

Zaharescu, A., Boyer, E., and Horaud, R. (2011). Topology-adaptive mesh deformation for surface evolution, morphing, and multiview reconstruction. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 33(4):823–837.