ISSN 2464-4617 (print)
ISSN 2464-4625 (online)
Computer Science Research Notes - CSRN
http://www.wscg.eu
WSCG 2025 Proceedings

A Robust Approach to Detect Intersections between Triangles with Different Numerical Representations

Luca Garau
University of Cagliari
Via Ospedale 72
09124, Cagliari, Italy
luca.garau@unica.it

Gianmarco Cherchi
University of Cagliari
Via Ospedale 72
09124, Cagliari, Italy
g.cherchi@unica.it

ABSTRACT

The detection and classification of intersections between triangles are crucial tasks in a wide range of applications within Computer Graphics and Geometry Processing, including mesh Arrangements, mesh Booleans, and generic mesh processing and fixing tasks. Existing methods are hard-coded and deeply integrated into specific algorithms, and significant efforts are usually required to integrate them into new pipelines or to extend them to different numerical representations. This paper presents a versatile and exhaustive algorithm to identify and classify intersections between triangles with either floating points, rational numbers, or implicit representations. The proposed tool is implemented as a C++ templated and header-only code that is generic and easy to integrate into further algorithms requiring the triangle-triangle intersection detection step. The developed tool has been tested and compared with a state-of-the-art approach, and it is shared with the Geometry Processing community with an Open Source license.

Keywords

Robust Geometry Processing, Triangle-triangle Intersections, Collision Detection, Mesh Arrangements, Exact Arithmetic, Implicit Representations.

1 INTRODUCTION

The detection and classification of intersections between triangles represent crucial tasks in several fields of Computer Graphics, Computational Geometry, and Geometry Processing, including mesh Boolean operations, mesh arrangements, mesh repairing, collision detection, 3D reconstruction, and other techniques involving triangle meshes.

The first complexity of this problem is generated by the significant number of different configurations in which two triangles can intersect each other. If two triangles can intersect into two different points, the complexity becomes particularly challenging in the case of coplanar triangles since we can have up to six different intersection points (each edge of a triangle crosses two edges of the other triangle). Enumerate all the possible configurations is tricky, and missing one of them when implementing a triangle-triangle intersection detection pipeline becomes quite easy.

A second crucial issue emerges when representing points generated by intersections detected in the previous step. Traditional numerical representations based on floating-point arithmetic often lack sufficient precision, causing rounding errors that can propagate throughout the entire algorithm and produce topological inconsistencies in the final result. Indeed, the floating-point space of the representable number leads to the necessity to approximate values, potentially leading to inaccuracies, especially in precision-sensitive computations like Boolean operations on meshes, where even minor errors can propagate into significant issues [Ber13].

Existing solutions for intersection detection and classification present significant limitations. First, they are usually deeply integrated with specific algorithms or custom data structures, complicating their reusability in different contexts. Second, most existing solutions are typically limited to a single numerical representation, such as floating-point arithmetic, rational numbers [The24], or implicit points [Att20], restricting the solution's code versatility. A brief overview of numerical representations and their importance in this class of problems will be discussed in Section 2.

To face this problem and facilitate the work of researchers and practitioners in these Computer Graphics areas, we propose an innovative templated tool capable of seamlessly operating across various numeri-

cal representations. Our solution, implemented as a header-only C++ library based on templates, works consistently with triangles represented using floating-point arithmetic, rational numbers, or implicit representations. This work represents a practical implementation of a problem discussed in [GSC25]. A central role in our tool is assumed by robust geometric predicates developed by Shewchuk in 1997 [RS97], specifically *Orient2D* and *Orient3D*, which efficiently and exactly determine the relative spatial relationships between a query point and, respectively, a line and a plane. Furthermore, the templated nature of our implementation allows for easy extension to additional numerical representations, allowing our code to be used with any numerical representation implementing the *Orient2D* and *Orient3D* geometric predicates.

The ease of integration offered by our tool facilitates incorporating the proposed method into new mesh processing tools as well as into existing mesh processing pipelines. To validate the robustness and performance of our solution, we conducted an extensive evaluation process detailed in Section 4. Results show that the performances of our tool are perfectly comparable to state-of-the-art algorithms, with additional advantages in numerical generality and ease of integrability and reusability.

Finally, we released our tool with Open-Source MIT license to encourage and facilitate further research developments in the Computer Graphics community (Repository **`LINK`**).

The rest of this paper is structured as follows. In Section 2, we analyze some state-of-the-art works that address the problem of triangle-triangle intersection classification with different numerical systems and different goals. In Section 3, we describe the proposed tool, provide the motivation for the choices made, detail the implementation choices, and explain the use. In Section 4, we describe the testing and validation phase that we performed on our tool, comparing it with the same task performed by a portion of a state-of-the-art algorithm. Finally, in Section 5, we describe possible future developments in our work.

2 RELATED WORKS

Detecting triangle-triangle intersections and representing the generated intersecting points are crucial in several Computer Graphics subfields. Typical examples include mesh Arrangements [ZGZJ16, CLSA20, GF24], a fundamental step for mesh Boolean operations [ZGZJ16, CPAL22, JKSH13, Lé24], or collision detection [Eri04, DXLS24], mesh repairing [ACK13], 3D reconstruction [TNWK22], and several other tasks. Even minor inaccuracies in intersection detection or point representation can lead to significant topological problems, damaging the correctness and robustness of the entire algorithm pipeline.

Despite the variety of proposed algorithms in literature and reference implementations on the web, a recurring issue is the lack of a generic solution that works with different numerical representations and that can be easily integrated into newly developed algorithms. Indeed, existing methods cannot be easily adapted or reused across different scenarios without substantial modifications. Adaptation generally requires aligning the intersection detection strategy with different numerical representations or different data structures.

Several applications that detect intersections between triangles can (sometimes) sacrifice precision in favor of the execution time, as we often have in collision detection pipelines. However, there are pipelines, such as mesh Arrangements, in which precision and robustness are fundamental since even minor approximation errors can lead to completely wrong results.

Mesh Arrangements involve partitioning the space defined by a set of triangles into a valid simplicial complex, necessitating robust intersection detection and classification to avoid degeneracies or overlaps. In general, in a mesh Arrangements pipeline, we need two fundamental steps: (1) locate all the intersecting elements and (2) split them in order to create a new structure (via re-triangulation) which includes intersection points and segments [LCSA21]. Inaccurate handling of intersections can compromise the entire mesh structure, making the mesh unusable for subsequent applications. Both the intersection detection and the creation of the new topology require absolute precision, which can be achieved with different technologies. Zhou et al. [ZGZJ16] leverage adaptive predicates to detect the intersecting elements, then use rational numbers to ensure absolute precision in the representation of the coordinates of the new points. Cherchi et al. [CLSA20] improved upon this approach using geometric predicates [RS97] to detect the intersections, and then they rely on an implicit representation of the intersection points [Att20] gaining execution speed without losing robustness. Mesh Boolean operations similarly require exact intersection detection to accurately determine internal and external model regions. Errors in intersection detection may lead to ambiguous or incorrect outcomes. All the above works highlight the need to develop algorithms that are robust to numerical approximation errors by resorting to different types of exact representations.

The first issue in an intersection detection pipeline is ensuring absolute robustness while maintaining an efficient algorithm in terms of time and memory. Among the first fast algorithms for triangle-triangle intersection detection, Möller [Möl97] and Held [Hel97] proposed two algorithms, both based on checking intersec-

tions between a line and the planes containing the triangles. Although very fast, these algorithms are not particularly robust in precision, but they are a fundamental groundwork for future methods. Guigue and Devillers achieved significant advancement in robustness [GD03], who introduced an algorithm relying solely on the signs of geometric predicates. This method avoids constructing intersection lines or segments between planes, thus reducing potential sources of error. Ling-Yu Wei [Ly14] introduced further optimizations to reduce the average computational cost by improving the management of non-intersecting cases and using vectorized operations, making the method suitable for real-time applications in animation and virtual reality. Intersection detection algorithms remain a thoroughly explored topic in literature. We refer the reader to [Ska23] for a comprehensive overview of this class of algorithms.

Fundamental ingredients in this kind of algorithm are (1) the detection of the intersections and (2) the computation of the new points generated by the intersecting elements.

Geometric Predicates

Fundamental tools of most of the algorithms that detect the intersection between triangles are the Shewchuk geometric predicates [RS97] introduced in 1997. Among them, two are particularly interesting for this class of problems. The first, called $Orient2D$, takes as input two points describing a straight line and a third one as a query point. Thanks to the evaluation of the sign of a determinant, this predicate is able to evaluate whether a point is located in one of the two half-planes ($Orient2D < 0$ or $Orient2D > 0$) or exactly on the line ($Orient2D = 0$). Analogously, the $Orient3D$ geometric predicate takes as input three points defining a plane and a fourth one as a query point, and it determines whether the point is located in one of the two half-spaces ($Orient3D < 0$ or $Orient3D > 0$) or precisely on the plane defined by the three points ($Orient3D = 0$). Combining these predicates makes it possible to quickly and robustly identify (without numerical precision problems) the intersections between the simplexes representing the two tested triangles. Shewchuk's predicates operate using an adaptive arithmetic technique. Initially, the expression is evaluated using standard floating-point operations. If the computed value exceeds a predefined safety threshold, the result is returned immediately, ensuring efficiency. In cases where the outcome falls near the uncertainty threshold, the method progressively increases the arithmetic precision, guaranteeing an exact result without resorting to full multiprecision arithmetic for every operation.

It is worth noting that these algorithms are not limited to intersection detection; they can also be used for classification. For instance, if an algorithm detects that triangles intersect, it can typically identify the edges involved in the intersection (calculating intersection points or segments). In the case of non-coplanar triangles, the intersection, if it exists, can only be a point or a linear segment; however, coplanar triangles can overlap in polygonal areas. Nevertheless, managing edge cases (e.g., triangles just touching at a vertex or lying on the same plane with overlapping edges) is delicate. Algorithms based on standard floating-point arithmetic may yield inconsistent results (e.g., a triangle "barely touches" another but, due to numerical errors, either appears entirely separate or with slight penetration). Algorithms based on geometric predicates provide an excellent foundation for robustness by reducing every decision to a determinant sign comparison.

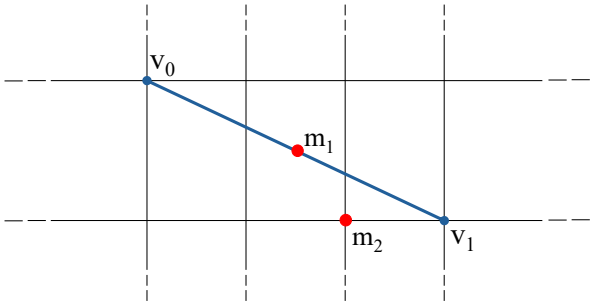

Figure 1: An example of approximation error with the floating-point notation. We can see the space of possible values representable by floating-point numbers as a grid. Even a simple operation like computing the midpoint of a segment can produce a point that cannot be precisely represented in the floating-point grid. This example shows that the exact midpoint m_1 of the segment $V_0 - V_1$ is approximated to the closest representable point m_2.

As we stated before, locating and classifying the intersection is not the only problematic step of this class of algorithms. After finding the intersecting elements, we need to represent the generated intersection points to use them in the remaining flow of the algorithm. The most common method employs floating-point arithmetic, approximating the coordinates of the new point into the closest representable number. Indeed, by construction, the space of numbers we can represent with floating-point notation can be seen as a non-uniform grid with an increased density in proximity to zero. Even a simple case, e.g., calculating the midpoint of a segment (see Figure 1), might yield a midpoint approximated to the nearest representable value, potentially causing errors propagating through subsequent pipeline steps. While floating-point numbers provide speed in computation, they cannot guarantee precision in numerical representation. For this reason, sometimes, we decide to renounce performance in favor of absolute precision.

Alternative Numeric Representation

A famous and spread alternative method to represent the coordinates of the new points is to use Rational numbers [ZGZJ16]. Rational numbers encode values as fractions with arbitrary-length numerators and denominators, providing absolute precision [The24]. However, rational arithmetic's computational complexity and performance overhead often limit their practical use in performance-critical applications. Consider that even a simple operation like a sum between two values (represented as a potentially huge fraction) can become highly resource-demanding.

In some cases, the explicit coordinates of the new points are not necessary to continue the algorithm flow. When we detect the intersection between the triangles simplexes, we can decide not to explicitly represent the intersection point's coordinates in order to avoid approximation errors. In this case, we represent intersection points explicitly by storing the coordinates of the original input elements involved in the intersection. For example, the intersection of a segment (edge of a triangle) and a triangle can be stored as "*LPI*" point, i.e. , a "Line-Plane Intersection". In this case, we store the coordinates of the two segments' endpoints and of the three triangles' corners rather than the coordinates of the points generated by their intersection. Representing intersection points in this indirect way also requires re-designing the geometric predicates, which has led to the development of "Indirect" predicates [Att20]. This new family of geometric predicates can compute the same information as the original Shewchuk's geometric predicates but works on implicitly represented points. This approach maintains robustness against rounding errors and offers a practical balance between computational efficiency and the stringent accuracy demanded by many advanced mesh processing tasks.

Countless algorithms in the literature deal with finding the intersections between triangles. Considering that there are different ways to represent the intersection points and, consequently, different versions of the geometric predicates that operate with them, we feel that a tool like the one we propose could be extremely useful for the scientific community in these fields. In fact, our tool allows for easy integration into new algorithms under development and supports multiple numerical representations. Furthermore, by design, our tool allows for easy extension with additional numerical representations.

3 THE TOOL

We present an easy-to-use tool written in C++ as a header-only library. It takes as input the coordinates of two triangles and returns information about the identified intersections. The core of the tool is divided into two main tasks: (1) detect and classify the intersections, and (2) return the information to create the new intersection points with a chosen numerical representation. To address the first step, we split the problem into simplexes of lower dimensions, meaning that the intersections between two triangles are identified by checking the intersections between faces, segments, and points [Lé24]. An example of this process can be seen in Figure 2, when a first intersection is defined by two coincident vertices, a second one is defined by a point lying on a segment, and a third one is generated by a segment of the first triangle crossing a segment of the second one.

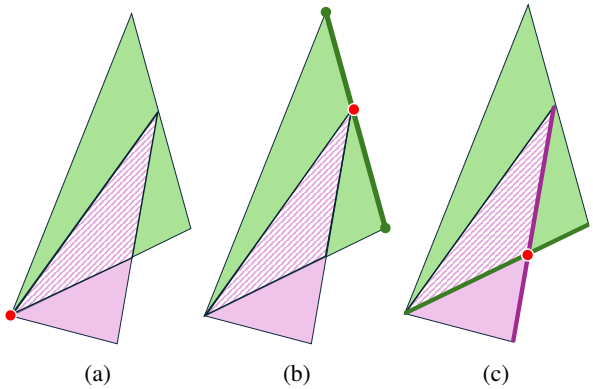

Figure 2: Decomposition of an intersection between two triangles into intersections between simplexes of smaller dimension: (a) coincident points, (b) point in segment, and (c) segment crossing segment.

To locate all possible intersections between the simplexes of the triangles in a robust and fast way, we use the *Orient2D* and *Orient3D* predicates [RS97] included in the [Liv19] library. Indeed, thanks to the combination of these two predicates, we can exactly understand if two simplexes touch or intersect each other. Since we aim to allow this identification process in triangles represented with different numerical representations, we developed our tool with C++ templates in order to support every kind of representation that is able to implement the *Orient2D* and *Orient3D* geometric predicates. In our current implementation, we already provide the support for the classical floating-point numbers with the original Shewchuck predicates [RS97], the rational numbers implemented in the CGAL library [The24], and the implicit points introduced in [Att20].

We classify all possible intersections into a set of intersecting simplexes, which we also return as output information. In particular, we can have (1) *coincident points*, (2) *point in segment*, (3) *point in triangle*, (4) *segment crossing segment* and (5) *segment crossing triangle* (see Figure 3 for a mosaic of all the five described cases) .

As a starting point of the detection pipeline, we catch, with six *Orient3D* calls, whether the input triangles are

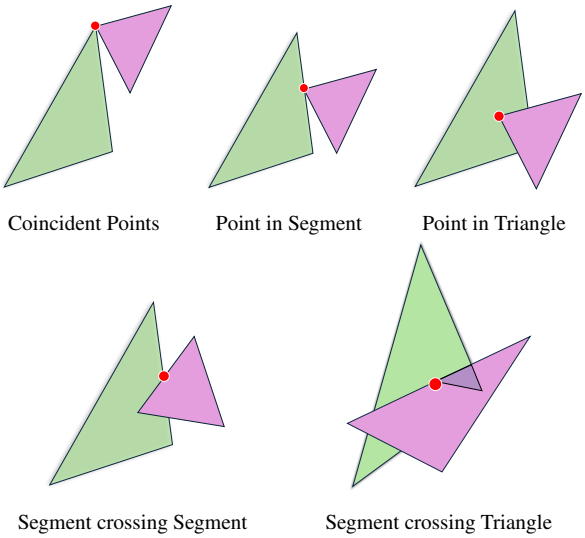

Figure 3: All possible cases of intersections between two triangle sub-simplexes.

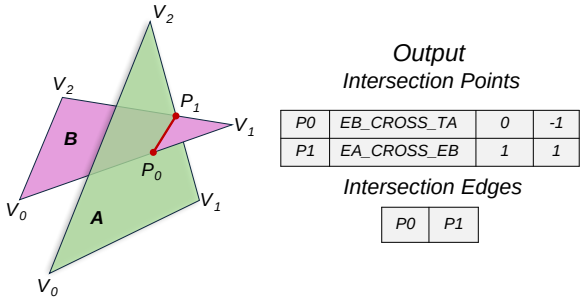

Figure 4: An example of an intersection between two triangles and the corresponding output representation. Edge $V_0 - V_1$ of triangle B crosses triangle A generating the P_0 intersection point. Point P_1 is given by the intersection between edge $V_1 - V_2$ of triangle A and edge $V_1 - V_2$ of B. Points P_0 and P_1 form an intersection edge. Note that, if less than two ids are required to express the intersection, we fill the useless field with -1.

coplanar or not, or if they do not intersect. In the case of non-coplanar triangles, we can find up to two intersections among the ones listed before, while in the case of coplanar triangles, we can find up to six intersections, excluding case (5).

Furthermore, intersections between two triangles can generate not only intersection points but also "*intersection segments*". Indeed, if you focus on Figure 4, the intersection between triangles A and B generated two intersection points linked together to form a new segment. For most of the algorithms requiring the classification of the intersections between triangles, intersection segments are also important information. For this reason, we also collect this information and return it as output together with the intersection points. Note that, finding the intersection segments can be a tricky problem. In fact, in the simplest case of non-coplanar triangles, we can have up to two intersection points and only a single intersection segment connecting them (Figure 5a). However, in the more complex case of coplanar triangles, we can combine up to six intersection points into intersection segments in several ways, leading to a planar polygon composed of up to six sides (Figure 5b).

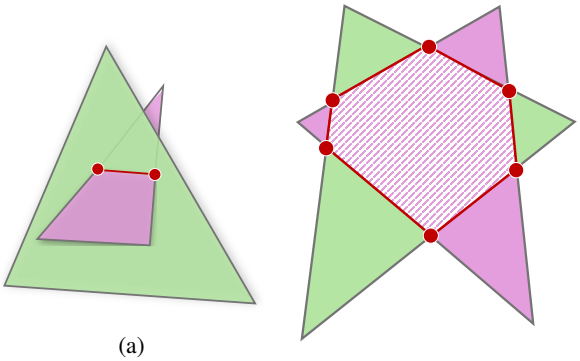

Figure 5: Examples of intersecting triangles. In (a), two non-coplanar triangles generate two intersection points connected by an intersection segment. In (b), two coplanar triangles produce six intersection points connected pairwise into segments, forming a planar polygon.

After finding the intersection points and segments, we are ready for the second step of the algorithm: returning all the information about the found intersections. Each triangle has been divided into lower simplexes, so we package the intersections as pairs of simplexes that generated it, together with a label providing information about the intersection type. In this way, new intersection points can be created with the numerical representation we select, whether explicit (e.g., floating-point or rational numbers) or implicit (implicit points).

The output structure of our tool is composed as follows:

- A list of intersection points, each one represented as a tuple in the form `(type, id0, id1)`, where `type` indicates the kind of intersection and the indices `id0` and `id1` to identify the simplexes involved in the intersection.
- A list of intersection segments, represented as tuples `(p0, p1)`, which refer to the positions of the intersection points inside the previously described list.
- Additional metadata providing supplementary information, such as whether the investigated triangles are coplanar, or other customizable information.

In Figure 4, we can see an example of our output structure, which describes the intersection of two non-coplanar triangles. In Algorithm 1, we sum up all the steps of the described tool in pseudo-code.

Algorithm 1 Triangle-Triangle Intersection. The algorithm identifies intersections by checking vertices, edges, and areas intersections. It takes as input the six vertices of two triangles T_0 and T_1 and returns a list L containing all the information about the detected intersections.

Require: List of vertices V of two triangles T_0 and T_1
Ensure: List L containing information about the detected intersection points and segments

```
1:
2:  Compute orientations of T0 vertices w.r.t. T1 plane
3:  Compute orientations of T1 vertices w.r.t. T0 plane
4:  if all vertices of a triangle are on the same side of
    the other triangle plane then
5:      return L (empty, no intersections)
6:  end if
7:
8:  for all vertices v_i in V do
9:      check coincident vertices
10:     add coincident vertices in L
11:
12:     if v_i lies on an edge or area of the other triangle
    then
13:         add v_i in L
14:     end if
15: end for
16:
17: for all edges e_i of T0 and all edges e_j of T1 do
18:     if e_i crosses e_j then
19:         add info about intersection in L
20:     end if
21: end for
22:
23: if T0 and T1 are not coplanar then
24:     for all edge e_i of T0 or T1 do
25:         if e_i crosses the area of other triangle then
26:             add info about intersection in L
27:         end if
28:     end for
29:
30:     if L contains 2 intersection points then
31:         add intersection segment (p0, p1) in L
32:     end if
33: end if
34:
35: if T0 and T1 are coplanar then
36:     for all point combinations (p_i, p_j) in L do
37:         if p_i and p_j lies on the same input edge of
    T0 or T1 then
38:             add intersection segment (p_i, p_j) in L
39:         end if
40:     end for
41: end if
42:
43: return L
```

4 VALIDATION

We tested our tool with an empirical validation involving the intersection detection on thousands of triangle meshes from the famous Thingi10k dataset [ZJ16], which is widely recognized in the literature for its high complexity [LV25].

To evaluate the robustness and efficiency of the tool, we included our code in the public reference implementation of a state-of-the-art mesh Arrangement pipeline [CLSA20], replacing the original detection and classification module (i.e., the function `checkTriangleTriangleIntersections` inside the `classifyIntersections` function in the public code).

In the experimental setup, conducted workstation equipped with 12 cores intel I9 processor and 128GB of RAM, we compared the execution time of our detection algorithm with that of the original pipeline, imposing a maximum runtime of 15 minutes per model and collecting data about 9996 models.

We checked if our tool produces precisely the same intersection list as the state-of-the-art algorithm and if the final result is exactly the same. To do so, we listed all the obtained intersections (both points and segments) in a common format, and we compared the resulting arranged mesh in terms of the number of simplexes and Euler's characteristic.

Once we were sure that the result obtained was correct, we compared the execution time of the two algorithms. As can be seen from the charts in Figure 6 and Figure 7, the two algorithms are fully comparable in terms of execution times.

In Figure 6, we show the models taken from Thingi10K dataset [ZJ16] processed by our algorithm and the state-of-the-art comparison [CLSA20], split into classes based on the time required for the detection and classification steps. Each bar in the chart represents the number of models processed in that time interval. As is clearly visible, the number of models processed for each analyzed time interval is almost the same.

The same behavior is shown in the chart in Figure 7, in which we report a line chart showing a direct comparison of the execution times of the two algorithms. We used the logarithmic scale to emphasize the timing. The models are sorted by increasing time (of detection and classification). Also in this case, we can say that, except for almost imperceptible oscillations, the execution times of the two algorithms are almost identical.

In Table 1, we report some numbers of a subset of "interesting" models selected by [LV25], which varies from 38 to 1.8M detected intersections. For each model, we reported the model, the number of input vertices, the number of intersection points, the execution time of our tool and the one of [CLSA20].

Model ID	Input Vertices	Intersection Points	Ours' time	[CLSA20]' time
356074	14K	38	0.02s	0.02s
199665	506K	1.2K	0.46s	0.45s
71377	502K	4.3K	0.60s	0.63s
622327	125K	34K	1.73s	1.67s
113906	24K	67K	1.35s	1.28s
1368052	1.2M	93K	3.51s	4.02s
247516	90K	106K	1.10s	1.11s
498460	95K	319K	8.94s	10.31s
252786	355K	1.8M	61.91s	64.27s

Table 1: In this table, we report the statistics of 9 "interesting" models from Thingi10k [ZJ16] selected by [LV25]. For each model, we report the model's name, the number of input vertices (Input), the number of detected intersection points, the execution time of our detection and classification function (Ours' time), and the time of the reference function in [CLSA20] in seconds.

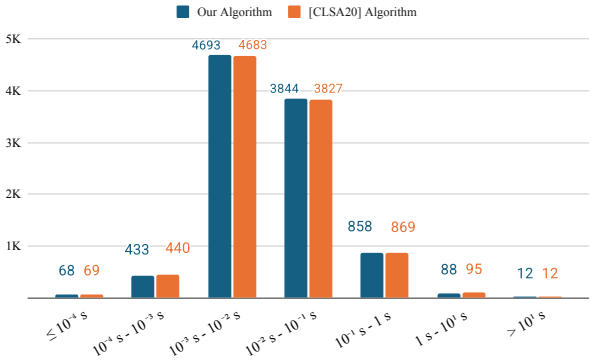

Figure 6: In this bar chart, we can see the models taken from Thingi10K dataset [ZJ16] processed by our algorithm and the one of [CLSA20], split into classes based on the time required for the detection and classification steps. On the horizontal axis we report the execution time intervals of the intersection detection and classification module, while on the vertical axis we report the number processed models. Blue bars (ours) and orange bars ([CLSA20])represent the number of models processed in that time interval with the two tested algorithms. As you can easily see, the number of models processed for each time interval is the same.

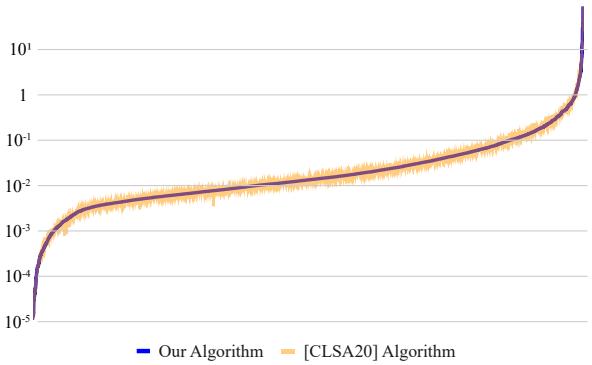

Figure 7: In this line chart, we directly compare the execution times of our detection and classification pipeline w.r.t. those of [CLSA20] on the models of Thingi10K dataset [ZJ16], in logarithmic scale. In the horizontal axis, we point out that the models are sorted by increasing time with respect to our algorithm, which is indicated in the vertical axis. Also in this case, the execution times of the two algorithms are almost identical.

As you can see, the execution times are also perfectly comparable for these models.

The results of this analysis clearly show that we are as robust and as fast as one of the most famous mesh Arrangements algorithms with a public implementation. Note that [CLSA20]'s algorithm is able to find and classify intersections in triangles represented exclusively with floating-point coordinates. Furthermore, their intersection detection module is fully integrated into their implementation and deeply dependent on their data structures. Therefore, the implementation of [CLSA20] cannot be smoothly extended to additional numerical representations or easily integrated into new algorithms. Instead, our tool already supports alternative numerical representations and can be easily extended with additional ones. Furthermore, our code is generic and can effortlessly be included in new pipelines. All of these features are guaranteed without sacrificing robustness or performance compared to the state-of-the-art algorithm.

5 CONCLUSION AND FUTURE WORKS

In this paper, we have presented a robust, fast, and easy-to-use tool for detecting and classifying triangle intersections. The strengths of this algorithm, implemented in C++ as a templated and header-only library, are more than one. First, it works with three different numerical representations (floating-point, rational numbers, and implicit points) and can be easily extended with additional ones. Second, it is implemented in a generic way and can be easily included in new algorithms under development. Furthermore, from a careful

testing and comparison phase with a state-of-the-art algorithm [CLSA20], the tool has been demonstrated to be equally fast and robust, but with the possibility of being easily integrated and extended, without dependencies on data structures or algorithms specific to implementations of specific projects.

We believe that this tool can be very useful for several sub-fields of the scientific community (and not only) in Computer Graphics areas. For this reason, our implementation is available under the MIT Open-Source license in the following repository: **`LINK`**.

As future developments, we are already working to support new numerical representation. Furthermore, we are already optimizing the code to make it more performant in terms of resources and execution times.

In a subsequent release of the code, we plan to optimize the pipeline in order to reduce the number of geometric predicates (Orient2D and Orient3D) invoked. In fact, the bottleneck of our code lies in the number of predicate calls, which, after careful analysis of the problem from a theoretical and combinatorial point of view, we hope to be able to significantly reduce the predicate calls, thus significantly reducing execution times.

Finally, since our goal is to help the community in their prototyping and development phases of new algorithms, we are open to help and code optimization from anyone who wants to contribute.

6 ACKNOWLEDGMENTS

The authors deeply thank Marco Livesu, for the precious technical and scientific support to this work. The authors gratefully acknowledge the support to their research by project "FIATLUCS - Favorire l'Inclusione e l'Accessibilità per i Trasferimenti Locali Urbani a Cagliari e Sobborghi" funded by the PNRR RAISE Liguria, Spoke 01, CUP: F23C24000240006.

7 REFERENCES

[ACK13] Marco Attene, Marcel Campen, and Leif Kobbelt. Polygon mesh repairing: An application perspective. *ACM Comput. Surv.*, 45(2), March 2013.

[Att20] Marco Attene. Indirect predicates for geometric constructions. *Computer-Aided Design*, 126:102856, 2020.

[Ber13] Gilbert Bernstein. Cork boolean library, 2013.

[CLSA20] Gianmarco Cherchi, Marco Livesu, Riccardo Scateni, and Marco Attene. Fast and robust mesh arrangements using floating-point arithmetic. *ACM Trans. Graph.*, 39(6):1–16, 2020.

[CPAL22] Gianmarco Cherchi, Fabio Pellacini, Marco Attene, and Marco Livesu. Interactive and robust mesh booleans. *ACM Trans. Graph.*, 41(6), 2022.

[DXLS24] Lei Dong, Yikai Xiao, Yanfeng Li, and Ruimin Shi. A collision detection algorithm based on sphere and ebb mixed hierarchical bounding boxes. *IEEE Access*, 12:62719–62729, 2024.

[Eri04] Christer Ericson. *Real-time collision detection*. Crc Press, 2004.

[GD03] Philippe Guigue and Olivier Devillers. Fast and robust triangle-triangle overlap test using orientation predicates. *Journal of Graphics Tools*, 8(1):25–32, 2003.

[GF24] Jia-Peng Guo and Xiao-Ming Fu. Exact and efficient intersection resolution for mesh arrangements. *ACM Trans. Graph.*, 43(6), November 2024.

[GSC25] Luca Garau, Riccardo Scateni, and Gianmarco Cherchi. Triangle-triangle intersections with different numerical representations. In *2025 IEEE Conference on Virtual Reality and 3D User Interfaces Abstracts and Workshops (VRW)*, pages 401–404, 2025.

[Hel97] Martin Held. Erit—a collection of efficient and reliable intersection tests. *Journal of Graphics Tools*, 2(4):25–44, 1997.

[JKSH13] Alec Jacobson, Ladislav Kavan, and Olga Sorkine-Hornung. Robust inside-outside segmentation using generalized winding numbers. *ACM Trans. Graph.*, 32:1 – 12, 2013.

[LCSA21] M. Livesu, G. Cherchi, R. Scateni, and M. Attene. Deterministic Linear Time Constrained Triangulation using Simplified Earcutfig. *IEEE Transactions on Visualization and Computer Graphics*, 28(12):5172–5177, 2021.

[Liv19] Marco Livesu. cinolib: a generic programming header only c++ library for processing polygonal and polyhedral meshes. *Transactions on Computational Science XXXIV*, 2019. https://github.com/mlivesu/cinolib/.

[LV25] Sylvain Lazard and Leo Valque. Removing self-intersections in 3D meshes while preserving floating-point coordinates. working paper or preprint, January 2025.

[Ly14] Wei Ling-yu. A faster triangle-to-triangle intersection test algorithm. *Comput. Animat. Virtual Worlds*, 25(5–6):553–559,

September 2014.

[Lé24] Bruno Lévy. Exact predicates, exact constructions and combinatorics for mesh csg. *arXiv*, 2024.

[Möl97] Tomas Möller. A fast triangle-triangle intersection test. *Journal of Graphics Tools*, 2(2):25–30, 1997.

[RS97] Jonathan Richard Shewchuk. Adaptive precision floating-point arithmetic and fast robust geometric predicates. *Discrete & Computational Geometry*, 18:305–363, 1997.

[Ska23] Vaclav Skala. A brief survey of clipping and intersection algorithms with a list of references (including triangle-triangle intersections). *Informatica*, 34(1):169–198, 2023.

[The24] The CGAL Project. *CGAL User and Reference Manual*. CGAL Editorial Board, 6.0.1 edition, 2024.

[TNWK22] Philip Trettner, Julius Nehring-Wirxel, and Leif Kobbelt. Ember: exact mesh booleans via efficient & robust local arrangements. *ACM Trans. Graph.*, 41(4), July 2022.

[ZGZJ16] Qingnan Zhou, Eitan Grinspun, Denis Zorin, and Alec Jacobson. Mesh arrangements for solid geometry. *ACM Trans. Graph.*, 35(4):1–15, 2016.

[ZJ16] Qingnan Zhou and Alec Jacobson. Thingi10k: A dataset of 10,000 3d-printing models. *arXiv preprint arXiv:1605.04797*, 2016.