LAWRENCE
LIVERMORE
NATIONAL
LABORATORY

LLNL-TR-737748

A Walking Method for Non-Decomposition Intersection and Union of Arbitrary Polygons and Polyhedrons

M. Graham, J. Yao

August 30, 2017

Disclaimer

This work performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344.

A Walking Method for Non-Decomposition Intersection and Union of Arbitrary Polygons and Polyhedrons

Marissa Graham[a,b], Jin Yao[b,1]

[a]*Brigham Young University*
[b]*Lawrence Livermore National Laboratory*

Abstract

We present a method for computing the intersection and union of non-convex polyhedrons without decomposition in $O(n \log n)$ time, where n is the total number of faces of both polyhedrons. We include an accompanying Python package which addresses many of the practical issues associated with implementation and serves as a proof of concept.

The key to the method is that by considering the edges of the original objects and the intersections between faces as walking routes, we can efficiently find the boundary of the intersection of arbitrary objects using directional walks, thus handling the concave case in a natural manner. The method also easily extends to plane slicing and non-convex polyhedron unions, and both the polyhedron and its constituent faces may be non-convex.

Keywords:
non-convex, non-decomposition, intersection, union, directional walk

1. Introduction

While polyhedron intersection is a well-studied problem, traditional methods require one or both polyhedrons to be convex [1] [2], or involve decomposition of the original objects. We notice that the boundary of the intersection is composed of subsets of the original faces of the polyhedrons, as demonstrated in Figure 1; specifically, the portion which is interior to the other polyhedron. In the case of the union, the boundary is composed of the exterior portions.

[1]corresponding author

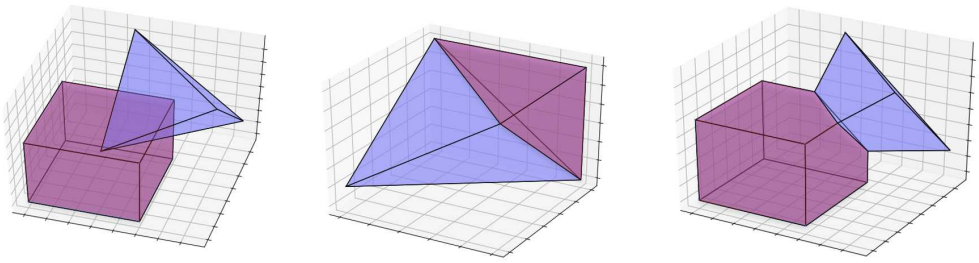

Figure 1: Example of intersection and union between two convex objects.

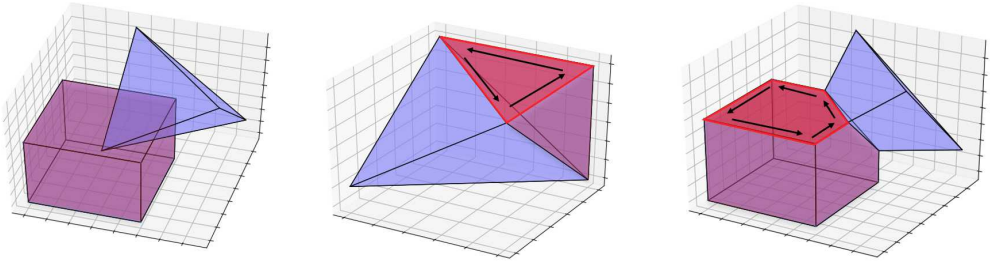

Figure 2: Example of the walks which form the faces of the intersection and union.

For two closed polyhedrons whose faces have outward pointing normal vectors, we may find these interior portions with a counterclockwise walk around the interior or exterior portion of the original faces, as demonstrated in Figure 2. The problem is then reduced to finding the interior or exterior portion of each face.

We observe that the edges of these new faces are created by a combination of subsets of the original edges and edges formed by intersection with faces of the other polyhedron. The intersection of polyhedrons P and Q then becomes a three step process, as shown in Figure 3.

- Intersect each face of P with the relevant faces of Q to find and save the segments of intersection.
- Check the edges of each face, if necessary, and clip away the exterior portions (or interior for the union) to get the rest of the edges of the intersection polyhedron.
- For each face of the original polyhedrons, walk along the saved edges

to obtain the new face(s) of the intersection polyhedron.

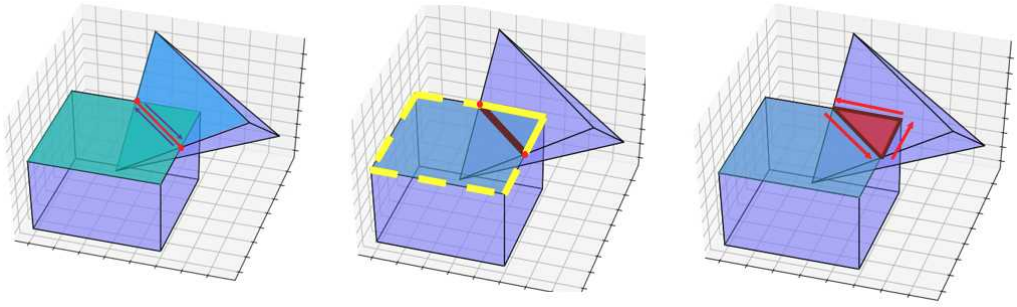

Figure 3: Overview of the steps of the intersection process.

1.1. Preprocessing

Throughout the method, we assume that the polyhedron is closed, and that vertices can uniquely hashed to a certain number of decimal places. We ensure that this is the case by only accepting polyhedrons on read-in that satisfy our requirements.

To ensure that our polyhedron is closed and the normal vectors point uniformly outwards or inwards, we require that each edge be present exactly twice and running in opposite directions. We attempt to minimize numerical instability by ensuring that all faces have area greater than a given ϵ which is intended to represent the digits of accuracy in the vertices. Similarly, all edges must have length greater than the same ϵ. We also check for non-planar, non-simple, and degenerate faces.

2. Inclusion Testing

In order to determine the portions of an edge which are interior or exterior, we need to be able to test whether the vertices of a polyhedron are inside, outside, or on the boundary of another. We also need to test the midpoints between vertices and intersections to determine whether line segments are interior or exterior. For our implementation, we use a ray casting method.

2.1. Bounding Boxes

For both inclusion testing and face intersection, it is important to quickly find the faces which intersect a certain bounding box or casting ray. In the absence of tree-based spatial structures, we use a simple set intersection to accelerate this process.

We find the minimum and maximum coordinate of each face along each axis and sort. Then finding the faces that satisfy a certain inequality condition ($x <= x_{\max}$, etc.) is reduced to the $O(\log n)$ operation of searching a sorted list, and we can use set intersection to find the faces that satisfy all the necessary inequality conditions. Excluding preprocessing, this method can approach $O(\log n)$ under good conditions. It is not a substitute for tree-based data structures, but it allows the large test cases to run in a reasonable amount of time, comparable to their read-in cost.

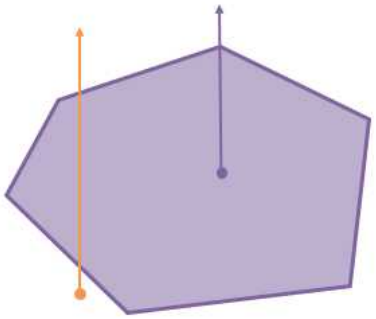

Figure 4: Even and odd numbers of boundary crossings indicating exterior and interior query points.

2.2. Method

Since our polyhedron is closed, we can count the number of boundary crossings to determine whether a certain query point is interior or exterior. An odd number of boundary crossings indicates an interior point and an even number indicates exterior, as in Figure 4.

To find these boundary crossings, we cast a ray from the query point in an axis-aligned direction. We find the faces whose bounding box intersects the ray, and determine whether each of them represents a boundary crossing, as in Figure 5. We project onto the plane perpendicular to the casting ray and test whether the projected query point is inside the projected face. If it is, we have crossed the boundary; if it is exterior, we have not.

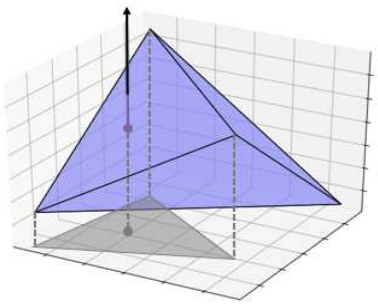

Figure 5: Projection of a query point and a face to test for a boundary crossing.

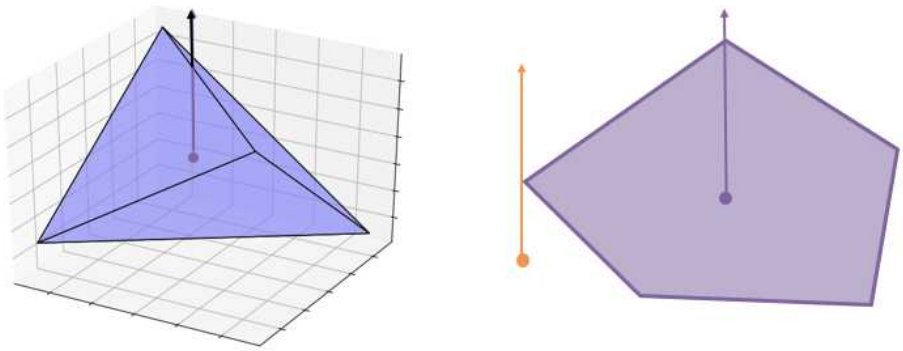

Figure 6: If the query point is directly below an edge or vertex, the result is ambiguous.

If, however, the projected query point is along the boundary of the projected face, as in Figure 6, we know nothing.

In this case, we randomly choose a new point between the original query point and the intersection point, and cast the ray in another direction, as in Figure 7.

Since the trouble is usually a result of integer-aligned vertices and query points, the randomly chosen point is highly unlikely to encounter edges or vertices and the process terminates with the next iteration.

We also encounter a problem if the query point is inside the bounding box of a face and it is therefore unclear which side of the face we are on, also as shown in Figure 7. In this case, we project the query point onto the face along the casting axis, and check whether the projected point is along the casting ray. This tells us whether the casting ray crosses the plane defined by the face.

3. Face Intersection

If two faces are not coplanar or parallel, there exists a line of intersection between the planes in which they lie. If the faces are not disjoint, the intersection between the two faces is then a subset of this line, consisting of line segments and points. The single points are ignored, but the line segments form edges of the polyhedron intersection or union.

For each face of each polyhedron, we collect the faces which intersect its bounding box, and for each of those we find these segments of intersection

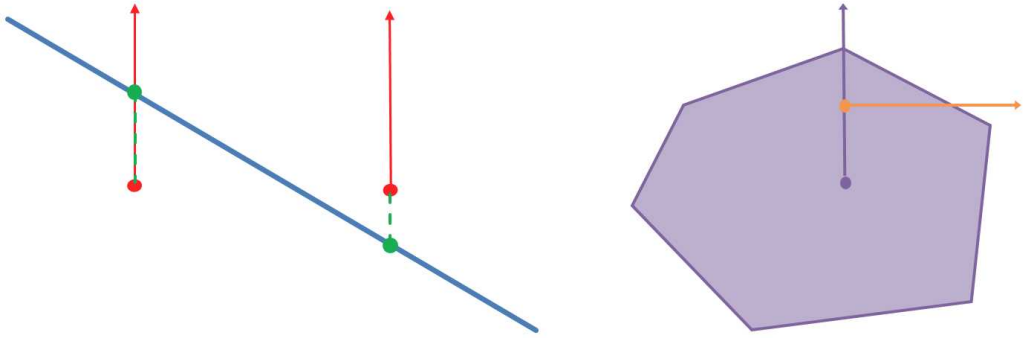

Figure 7: Left: Projecting query point onto the face to see if the projection is along the casting axis. On the left, the projection point is along the casting axis, so we have crossed the boundary. On the right, we have not. Right: Switching directions to avoid an ambiguous boundary crossing case.

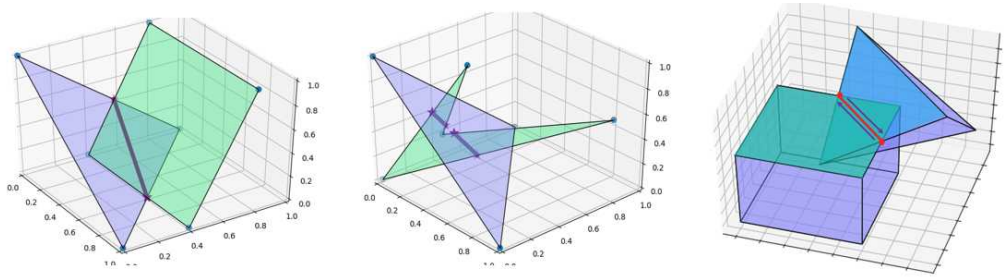

Figure 8: Left: A segment of intersection between two polygons. Center: Multiple segments of intersection between two polygons. Right: A segment of intersection between two faces, with corresponding edge directions.

and as illustrated in Figure 9.

- Get the points of intersection between the edges of each face and the plane in which the other face lies. This reduces to repeated line segment-plane intersection. In Figure 9, these are represented as stars the same color as their face of origin.
  - We note that this is the primary place in the code where new points are created, and thus the primary place in the code in which perturbations can be introduced. It is therefore important to have a robust implementation of line-plane intersection.
  - These points of intersection form the potential endpoints of the segments of intersection.

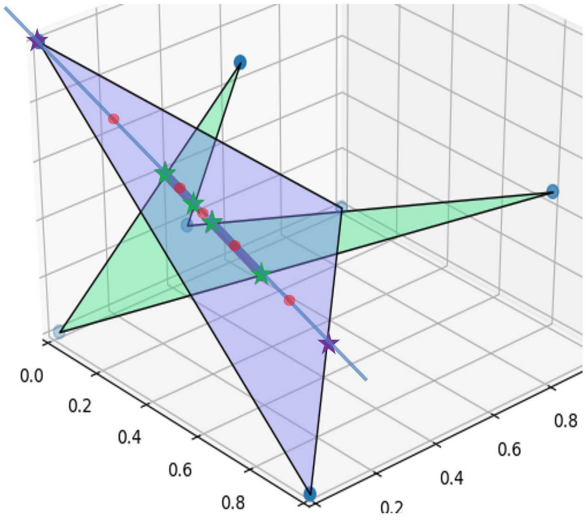

Figure 9: Collecting intersection points and testing midpoints for inclusion to determine the segments of intersection between two faces.

  - In addition to forming endpoints of the segments of intersection, these points indicate where edges may have crossed the boundary of the other polyhedron and therefore need to be stored for future use.

- Sort the points with respect to their distance along the line of intersection.

- Test the midpoints of each pair for inclusion in the spatial polygons being intersected. Since these are midpoints between the only intersection points, we are unlikely to lose the result to perturbations introduced in the 2d conversion process. In Figure 9, these are represented as red dots.

- If the midpoint between a sequential pair belongs to both polygons, it forms an edge of the intersection or union.

- Store the edge on both faces, ensuring the direction of the edge is counterclockwise with the face.

3.1. Coplanar Faces

Since the intersection between coplanar faces does not lie along a single line, they need to be handled differently. We convert the faces to 2d and get the intersection or union there, then add the resulting edges to each face. We also notice that in the case of the union, the rest of the edges found during the usual intersection process may be interior to the union of the coplanar faces, as shown in Figure 10. Therefore, while we still store the intersection points as usual, we do not store any additional edges.

In practice, this is implemented by clearing all edges when we discover that a face is coplanar and flagging it as such, and checking that faces have not been flagged as "coplanar" before intersecting them.

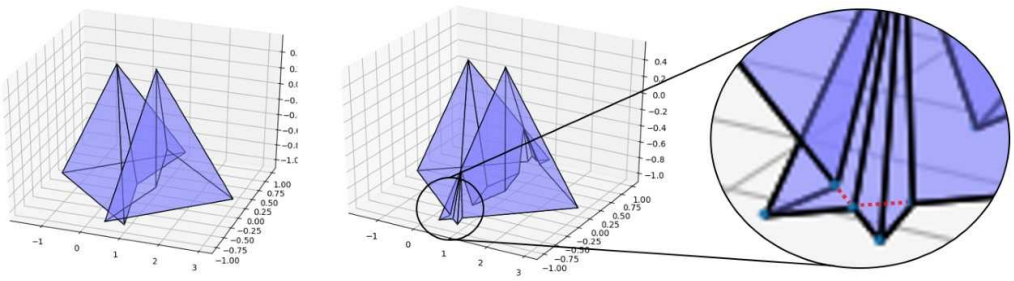

Figure 10: We take the union of the polyhedrons on the left. Notice that when we take the union, while the dotted red lines are on the boundary of their source faces, they are interior to the union of the two faces and should therefore not be included in the result.

3.2. Union

The edges found by face intersection are used in both the polyhedron intersection and the union. The difference is that the edges used for the intersection run the opposite direction as those used in the union, and are stored accordingly.

4. Edge Clipping

The rest of the edges used in the polyhedron intersection or union are found by clipping away the exterior or interior portion of the edges in the original faces. Since each edge of each face knows the intersections that occur along it, this is a fairly simple process.

- If the vertices of a face are uniformly exterior or interior to the other polyhedron and there are no edge intersections associated with the face, it can be safely added or ignored.
- Otherwise, we check each edge of the face individually.
  - If there are no intersections along an edge, it will be either completely interior or exterior. If the vertices are not both interior or exterior, we test the midpoint. This allows us to handle faces whose vertices are on the boundary of the other polyhedron.
  - If there is one intersection, store a new edge from the intersection point to the vertex or vice versa.
  - If there are multiple intersections, we test the midpoints between sequential pairs of vertices and intersections in a process similar to the face intersection.

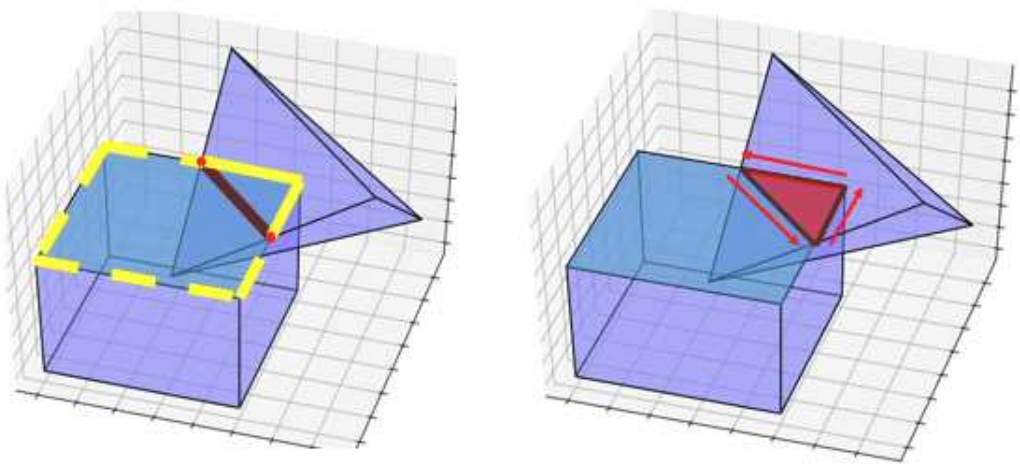

Figure 11: Left: The edge clipping process. Right: Edge walking.

In the case of an edge which lies along the boundary, it turns out that we need to keep the edge if we are intersecting and discard it if we are taking the union.

5. Edge Walking

We store our edges as relationships between numbered vertices; each edge is an entry in a Python dictionary with the tip as the key and the tail (or list of tails) as the value. This allows us to use the same walking method across dimension and for both intersection, union, and plane slicing. In each case, the process is simple, as described in Algorithm 1.

The main difficulty we encounter is the case in which vertices are shared by multiple edges on the same face. To solve this, we give the code the ability to handle both a single integer or a list of integers as the tail of an edge, both in insertion and in deletion. This will sometimes result in a non-simple polygon, but the presence of repeated vertices makes this easy to check for. Breaking the walk apart between the repeated vertices gives us the simple polygons we prefer.

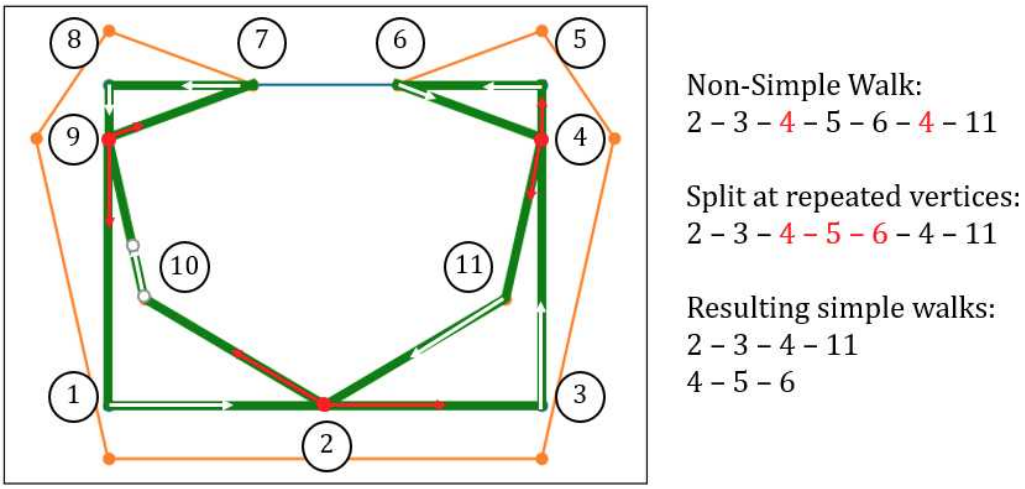

Figure 12: Handling non-simple polygons in the walking algorithm.

6. Special Considerations

6.1. 2d Intersection and Union

The same method applies to polygon intersection and union in two dimensions, but is even simpler, since the only edges involved are clipped from the original edges of the polygons. We collect the intersections between edges of each polygon, clip away the interior or exterior portion of the edges as usual, and then walk.

Algorithm 1 walk_edges

```
while edges are still available in the dictionary do
  Choose an edge
  if a tail of the chosen edge is the tip of another edge then
    while we have not hit a dead end or returned to the start do
      Get the next edge and add it to the walk
      Remove the edge from the dictionary
    end while
    if the resulting walk is non-simple then
      Split the walk up according to the repeated vertices
    end if
  else
    The tail is a dead end and the edge is deleted
  end if
end while
```

6.2. Plane Slicing

The method is also easily adapted to clip non-convex objects against an arbitrary plane. Instead of expensive inclusion testing, we only need to test whether vertices are left or right of a plane, which is a simple dot product with the normal vector.

Clipping a polygon against a plane is similar to the polyhedron intersection process. The segments of intersection between the polygon and the plane are stored as edges and also indicate the intersection points along the edges of the face. We then clip the edges of the polygon at the intersection points to keep only the portion which is left of the plane, and walk to get the result, as in Figure 14.

To clip a polyhedron against a plane, we first collect all the faces which cross the plane. All those to the right are discarded, and all those to the left are added to the result polyhedron. To get the rest of the faces, we clip each face that crosses the plane against it and store those faces, and store the edges of intersection in the walking dictionary. Walking these edges gives us the faces along the slicing plane, and the clipped polyhedron is complete. Some examples are shown in Figure 15.

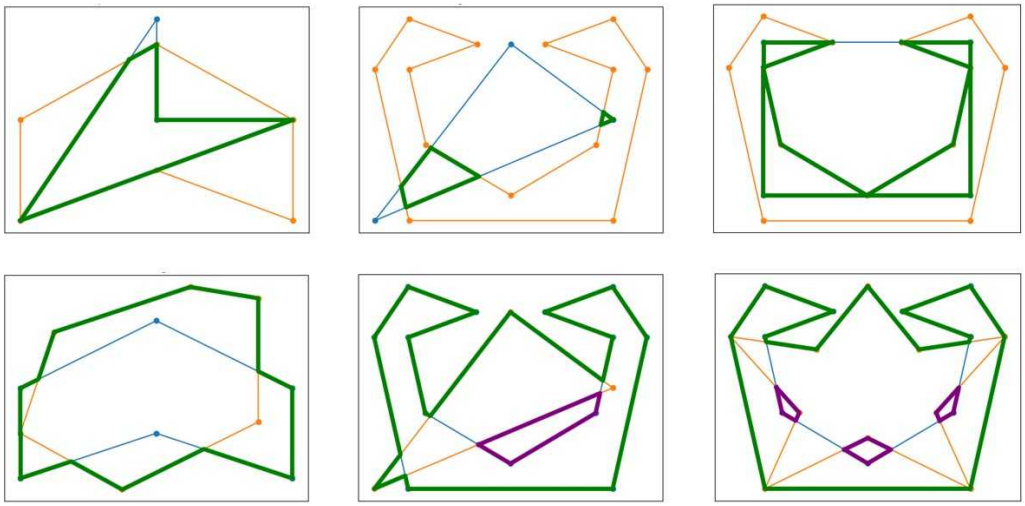

Figure 13: Examples of 2d intersection and union.

6.3. Holes

When we slice or take the union of non-convex objects, we frequently encounter a result in which some of the faces contain holes, as in Figure 16. The method does not require any special consideration to handle these, and represents them as clockwise faces which are contained in other faces of the object. In practice, however, this does cause a problem. Faces with holes are not supported by the .obj file format used in this project. Currently, we simply flag these cases. If any of the normal vectors point opposite the normal vector of the original face, we know we have a hole and store the indices of the relevant faces for postprocessing.

7. Conclusion

The overall temporal complexity of the algorithm is approximately $O((n+m)(f(n)+f(m)))$, where $f(n)$ is the cost associated with searching the spatial structure of the polyhedron and n and m are the number of faces of each polyhedron. With good tree-based spatial structures, $f(n)$ can be as low as $\log n$. Thus the overall complexity of the method can be as low as $O(n \log n)$.

We have observed successful intersection and union of small non-convex polyhedrons, including those with coplanar faces, vertices along the boundary, edges along the boundary, results including holes, results with faces with

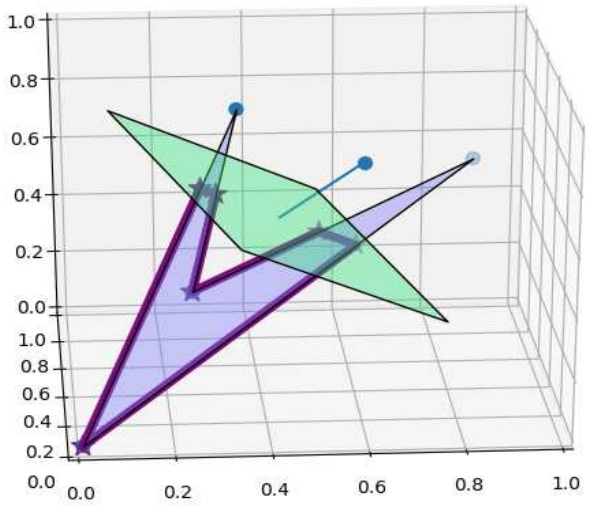

Figure 14: Slicing a polygon against a plane with normal vector in blue.

shared vertices (potential non-simple polygons in the result). We also note that the 2d inclusion method used is also independent of convexity, so both the polyhedron and its constituent faces may be non-convex.

The next step for the project is to convert the package to C++. In this process, we will take advantage of existing robust computational geometry primitives and efficient tree-based spatial structures in order to bring the total complexity of the algorithm to $O(n \log n)$.

The conversion process will afford the opportunity to handle larger test cases and continue to make the algorithm more robust. We also intend to add or integrate further preprocessing features, including unit normalization and the ability to read and write to multiple file formats. In addition, the holes sometimes created in the union process need to be decomposed into simpler polygons in order to avoid problems with the .obj file format.

We will also add set differencing. Theoretically, this is accomplished by reversing the faces and normal vectors of the polyhedron to be removed and then intersecting. However, in practice, we need to rework the 2d inclusion methods to handle clockwise polygons more effectively before this feature can be added.

Currently, we simply discard polyhedrons with non-planar faces, but deformation processes can easily result in these and it is therefore useful to extend the method to handle these.

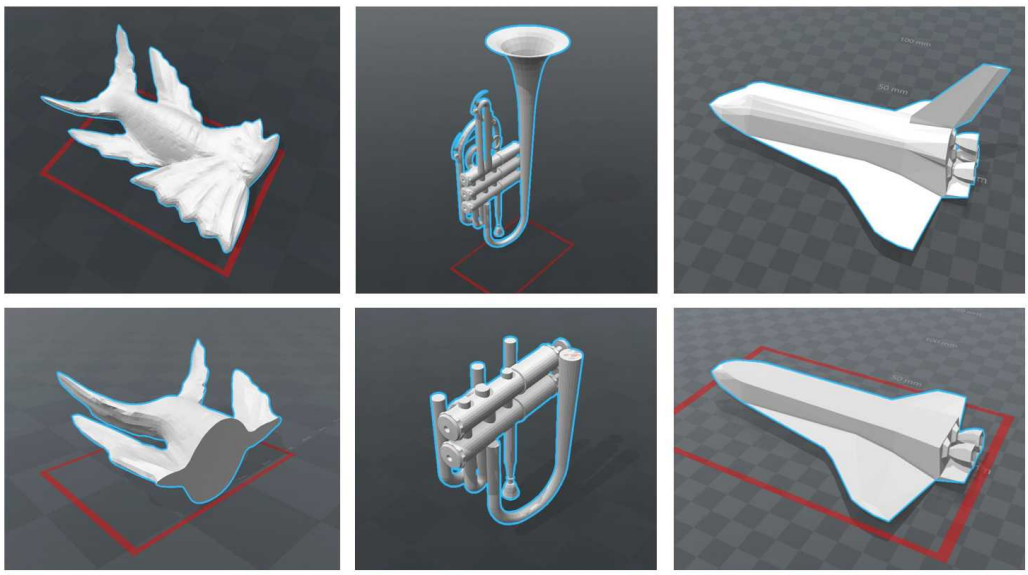

Figure 15: Some sliced polyhedrons generated by the code and viewed as .obj files.

Overall, this method presents a practical alternative to traditional methods of answering geometric queries for non-convex polyhedrons. Our Python code is intended to be a tool for simple cases and serve as a framework to address some of the practical issues associated with implementation of the method including numerical stability, poor inputs, data structures, and various edge cases.

Appendix A. Package Functionality

The package functionality can be roughly classified into 2d methods, methods of the Polyhedron class, and 3d functionality, as summarized in the the table below.

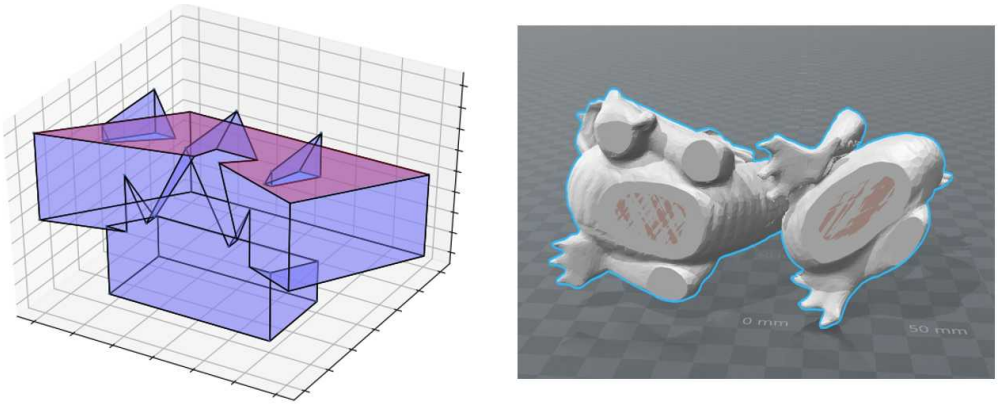

Figure 16: Left: A face with holes resulting from the union process. Right: Such holes are not properly represented by the .obj file format.

2d Functionality	Polyhedron Methods	3d Functionality
Polygon area	Read from .obj files	Polygon area
Left-right testing	Write to .obj files	Find normal vectors
Inclusion testiong	Print relevant features	Polygon-polygon intersection
Line intersection	Volume	Polygon-plane intersection
Polygon intersection	Naive join	Left-right testing
Polygon union	Equality testing	Line-plane intersection
	Translation	Polyhedron inclusion testing
	Scaling	Polyhedron plane slicing
	Rotation by a matrix	Polyhedron intersection
	Display using matplotlib	Polyhedron union

References

[1] D.E.Muller and F.P.Preparata, "Finding the intersection of two convex polyhedra", Theoretical Computer Science, vol.7 issue 1, pp 217-236, 1978.

[2] Kurt MehlhornKlaus Simon, "Intersecting two polyhedra one of which is convex", Fundamentals of Computation Theory, Lecture Notes in Computer Theory, volume 199, 1985.

[3] A. Bemporad, K. Fukuda, and F. D. Torrisi, "Convexity recognition of the union of polyhedra", Computational Geometry 18, 141154, 2001

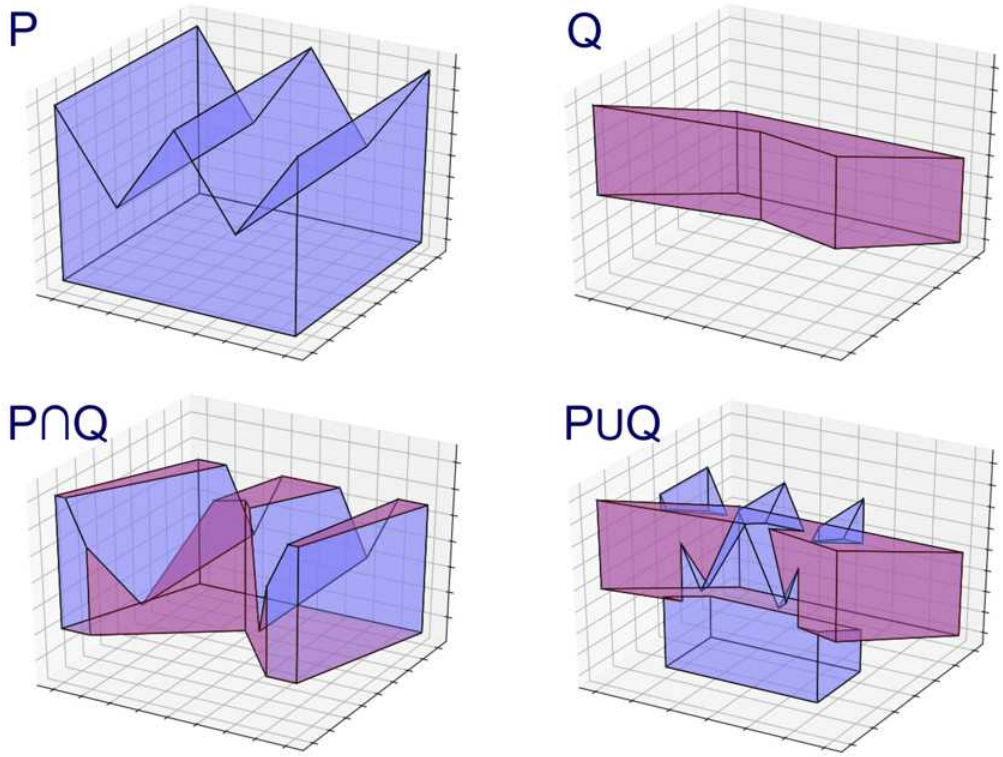

Figure 17: A successful intersection and union of two non-convex objects, colored to show the source of each face in the result.

[4] B. Aronov, M. Sharir, B. Tagansky, "The union of convex polyhedra in three dimensions" SIAM J. Comput. 26, pp. 1670-1688, 1997.

[5] F.R. Feito and J.C. Torres, "Inclusion test for general polyhedra", Computers & Graphics, vol. 21, pp. 23-30, 1997

[6] J. Lane, B. Magedson, and M. Rarick, "An efficient point in polyhedron algorithm" Computer Vision, Graphics, and Image Processing. vol. 26. pp. 118-125, 1984.

[7] J. O'Rourke, "Computational Geometry in C", Combridge University Press, 2005.