Trapezoidal Decomposition

A Python Implementation for Trapezoid Decomposition

Terrance Williams

November 22, 2023

1 Logic

First, the user creates the representation of the obstacles using the classes defined in *shapes.py*. The *shapes.Polygon* class can be used to create general polygonal shapes, obviating the need to combine the more specific shapes like **Quadrilateral** or **Triangle**.

The main loop iterates over every obstacle. Each obstacle's vertices are iterated over. For a given obstacle vertex, two **Edge** objects are created: the first is a vertical segment from the vertex to the top border, the second from vertex to bottom border. My thinking was that it is more efficient to iterate over the known vertices since each desired segment passes through said points. The alternative was brute-force stepping through the map and checking every point along the way.

Take the top segment. Let the current obstacle be called *obst*. We now iterate over every edge in *obst*. The sole purpose of this iteration is to determine if the vertical segment results in a self-intersection within *obst*. We calculate the intersection between *seg* (the vertical segment) and a given edge using the method defined in the **Edge** class. If 'None' is returned, the two edges are parallel, so no intersection. If the intersection is the vertex itself, ignore it (this means the given edge is one of the adjacent edges to the current vertex). Either of those conditions results in the loop proceeding to the next edge in the shape.

If those two conditions are *not* present, then we proceed to perform more checks on the intersection point. Because the segments are split into top and bottom, we are only concerned if there is a self-intersection above or below the vertex, respectively. So, given the top segment, for example, if the intersection point is on the edge and its y value is greater than the vertex y value, there is a self-intersection.

Self-intersection results in a 'continue' command; we end the checks for the current vertical segment. Given no self-intersections, we remain with the same vertical segment, and the program proceeds to check the other obstacles for intersections.

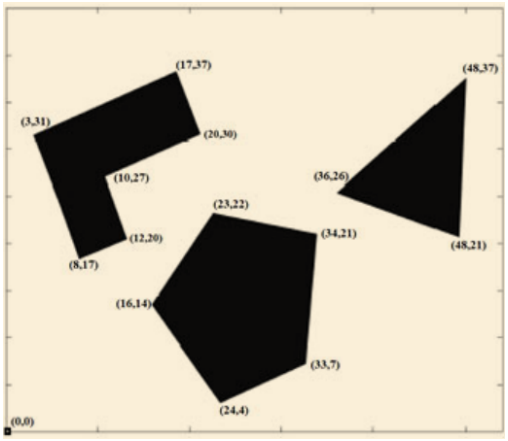
Just to recap, we are currently on a given obstacle, *obst*, using a vertex from *obst* and working with one of two vertical segments created from said vertex.

We enter another iteration where the other obstacles (non-*obst* obstacles) are each looped over. Each edge in said obstacle is checked for an intersection. The following check is then performed: if the intersection does not exist (edges parallel), the intersection point does not lie on the edge[1], the intersection is identical to the vertex, the intersection has the aforementioned invalidating y-value, or the intersection is contained within the other shape, the intersection is invalid, and the program proceeds to the next edge. Otherwise, the y-value is stored in a list of potential values.

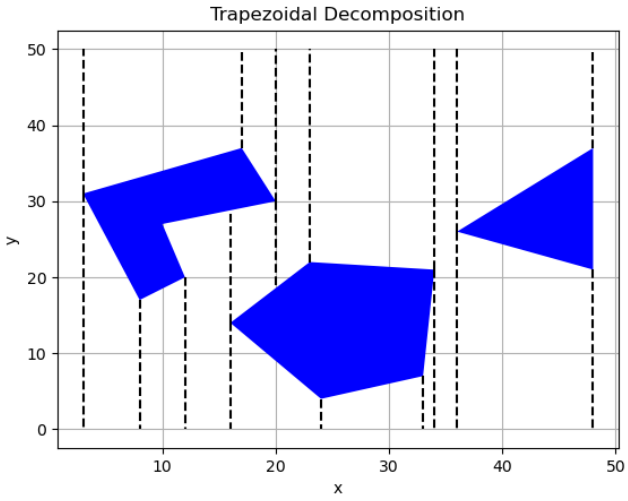
Once this check is done for all edges in the shape for all non-*obst* shapes, the minimum y value is used for the intersection point. This is done to handle the case in which multiple obstacles would intersect with the vertical segment at valid points. We want to choose the minimum value if we are currently testing the top vertical segment and the maximum if using the bottom segment. Finally, if no intersections were found, the relevant vertical segment itself is added to the list of segments.

The above operation is then repeated for every vertex in the obstacle for all obstacles. The result is the following:

[1] It indeed lies on the lie defined by the edge but outside of the shape itself.



(a) Original Map



(b) Applied Trapezoid Decomposition

Figure 1: Trapezoidal Decomposition Comparison