Contour of a Set of Rectangles

Oriana Maria Iancu, Cătălina Jemna
Students at West University of Timișoara
Email: catalina.jemna03@e-uvt.ro, oriana.iancu04@e-uvt.ro

January 24, 2025

Abstract

This paper addresses the problem of computing the contour of a union of rectangles, a common challenge in computational geometry. After encountering limitations with an interval-based approach, we developed a pixel-based method that represents the plane as a grid and identifies the contour by tracing boundary pixels. This approach is simple, robust, and handles complex scenarios like overlaps, gaps, and shared edges.

Contents

1 Introduction

1.1 Motivation

The problem of computing the contour of a set of rectangles is a fundamental challenge in computational geometry with numerous practical applications. This contour serves as a simplified and efficient representation of the combined spatial arrangement of these rectangles, by removing unnecessary internal details while preserving the essential geometry.

This problem comes up in many situations in the real world. In robotics, understanding the contours of obstacles enables robots to navigate environments safely and effectively. Another typical example is the visualization of a city's skyline which is used by various professionals and industries such as Urban Planners, Architects, and Game Developers. When viewed from a distance, the details of individual structures are hidden, and the visual perspective is dominated by the contour of the largest buildings. Likewise, in industrial or warehouse layouts, the shape of stacked or arranged products serves to maximize storage efficiency by giving information about the total amount of space utilized. It also serves in manufacturing, helping to optimize the cutting of materials like metal, glass, or fabric to reduce waste.

These practical applications illustrate the broad relevance of solving the contour computation problem.

1.2 Reading instructions

This paper delves into the problem of computing the contour of a set of rectangles, a fundamental challenge in computational geometry with significant practical applications.

First, we formally define the problem: outlining what a contour is, the challenges posed by overlapping rectangles, and the desired properties of an efficient solution. Following that, we suggest a solution that involves an algorithm designed to handle various scenarios, such as aligned edges, nested rectangles, disjoint rectangles, and partially overlapping regions. We delve into the implementation details, including the algorithm's logic and the used data structures. Then we will guide you on how to use our program, providing a step-by-step explanation of its inputs, outputs, and customizable features. Next, we run our algorithm on different data sets that represent diverse scenarios, analyzing its performance and accuracy. We conclude by summarizing our findings, talking about possible future study directions, and outlining possible research topics.

2 Formal Description of Problem and Solution

The problem we address is computing the contour of a set of rectangles in a 2D plane. The contour is defined as the outer boundary of the region covered by the rectangles, represented as a series of connected line segments. Given a set of rectangles, each defined by their bottom-left and top-right corners [x1,y1,x2,y2] the goal is to compute and visualize this boundary. This task becomes challenging when dealing with complex scenarios such as:

- Adjacent rectangles that share common edges.
- Overlapping rectangles that share portions of their surfaces.
- Nested rectangles where one rectangle is fully enclosed within another.
- Disjoint rectangles that do not interact but still contribute to the overall contour by having their own.
- Rectangles positioned to form discontinuous coverage, leaving gaps in their union.

Our objective is to develop an algorithm that handles all these scenarios seamlessly and outputs a clean, continuous contour.

We initially approached the problem using a sweep line algorithm combined with a segment tree to manage active intervals along the y-axis. As the sweep line moves across the x-coordinates of the rectangles, vertical edges are dynamically added or removed using the segment tree. The active intervals in the segment tree are used to construct the vertical components of the contour. However, this approach encountered significant issues. Handling edge cases like aligned edges and nested rectangles introduced errors. Also, the resulting contours were incomplete or incorrect for complex configurations such as gaps or overlapping rectangles.

That's why we decided to change our approach. Our final solution consists of a Pixel-Based Approach that represents the entire 2D space as a grid, treating each so-called pixel as a discrete unit, allowing for a simpler and more robust computation of the contour.

The first step involves mapping the given rectangles onto a 2D mask, where each rectangle is represented by marking the corresponding units (referred to as "pixels") within its boundaries as "filled" (e.g. set to 255 in a binary mask). For example, a rectangle defined by the coordinates

(x_1, y_1, x_2, y_2) would fill all pixels from (x_1, y_1) to $(x_2 - 1, y_2 - 1)$. In this way, we eliminate the need for complex interval management and provide a straightforward way to determine which areas are covered. Once the rectangles are represented on the grid, the next step is to identify boundary pixels, the pixels that form the outermost edges of the union of rectangles. A pixel is classified as a boundary if it is "filled" and any of its four neighbors (above, below, left, or right) is "unfilled". This check ensures that the contour captures all transitions between filled and unfilled regions. To efficiently manage the computed contour, we store it in an internal data structure, specifically a list of boundary points. As the contour is constructed, boundary points are added to the list in the order they are traced.

Unlike the previous approach, this method inherently handles scenarios like overlapping edges, gaps, and nested rectangles without requiring additional logic.

After identifying the boundary pixels, the algorithm traces the contour by following these boundary pixels in a continuous sequence. The tracing ensures that the final contour forms a closed loop, accurately outlining the union of the rectangles.

To verify the correctness of the contour, the algorithm overlays the traced boundary onto the original grid representation of the rectangles. The contour is visualized as a series of line segments connecting the boundary pixels, providing a clear and precise depiction of the union of the rectangles.

3 Model and Implementation of Problem and Solution

3.1 Programming Language Choice

To model the problem of obtaining the contour of a set of rectangles, we use Python as our programming language. Python takes priority over all the existing programming languages, since its syntax is clean and readable, making writing, reading, and maintaining code much easier than many other languages. It also hosts powerful libraries, most noticeably NumPy for dealing with multidimensional arrays, and Matplotlib for creating clear and insightful visualizations. In regard to this task, NumPy facilitates the efficient manipulation of pixel grids (the rectangle masks and boundary computations), while Matplotlib helps us visualize both the original rectangles and their computed contours.

3.2 User Manual

The implementation of the solution is available on **t**his GitHub link. After downloading the file, open it in PyCharm (recommended) or any python IDE for smoother execution. Then follow these steps:

1. Install required packages
In your terminal, install the **numpy** and **matplotlib** packages by running:

```
pip install numpy matplotlib
```

2. Review the default rectangle data
The code contains a list of rectangles, each defined by four integers. Each rectangle is specified as $[x1, y1, x2, y2]$, where:

- $(x1, y1)$ is the bottom-left corner of the rectangle
- $(x2, y2)$ is the top-right corner of the rectangle

For example: $rectangles = [[1, 1, 8, 6], [6, 2, 10, 8], [3, 1, 7, 5]]$

3. Modify or add rectangles
You can customize the rectangle coordinates according to your needs or use the default data to test the solution.

4. Run the script
After making any necessary modifications, run the script. A white canvas will appear, split into two parts:

- On the left: The input rectangles (the ones you specified).
- On the right: The computed contour outlining the combined shape of all rectangles.

This visualization helps demonstrate how individual rectangles merge to form a single shape and how the resulting boundary is computed.

4 Case Studies and Experimental Evaluation

This section presents four case studies that demonstrate how the implemented solution behaves when presented with different rectangle configurations. We describe the experimental setup, the specific rectangle arrangements, and how to interpret the resulting contours. This allows interested readers to replicate the experiments from scratch, verify the outcomes, and compare with our results.

4.1 Case 1: Adjacent Rectangles that Share Common Edges.

Setup. Two or more rectangles partially overlap, sharing sections of their edges or corners. For example: $rectangles = [[1, 1, 5, 5], [5, 1, 7, 5], [1, 5, 4, 10]]$

Expected Behavior.

- Overlaps merge into a single filled region in the mask.
- Only the outer boundary of the combined shape should be detected and drawn in blue.
- The printed boundary coordinates correspond to the external perimeter; shared or internal edges do not appear as boundaries because they transition from "filled" to "filled".

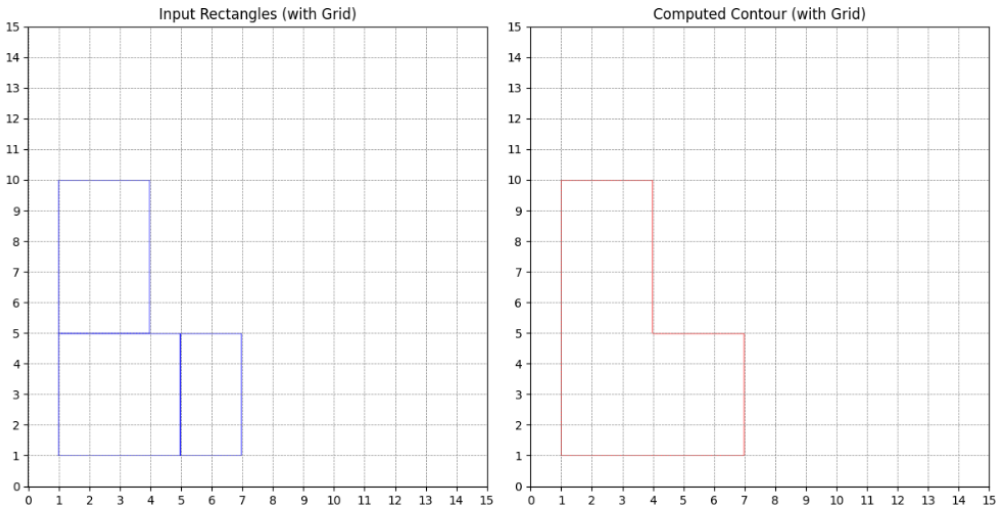

Figure 1: Adjacent Rectangles Sharing Edges

Observations. Running this configuration shows a single continuous contour encircling the union of the rectangles. This confirms that edges within the overlapping regions are treated as *internal* and thus excluded from the final boundary.

4.2 Case 2: Overlapping Rectangles that Share Portions of their Surfaces

Setup. One rectangle is entirely contained within another. For instance: $rectangles = [[1, 1, 8, 6], [3, 1, 7, 5], [4, 11, 11, 14], [2, 6, 5, 12], [8, 7, 12, 12]]$

Expected Behavior.

- The *outer perimeter* of any large grouping should still be clearly detected.
- Internal edges facing the unfilled space are similarly identified as valid contours.
- Any incomplete connection among the rectangles manifests as a break in the overall boundary, leading to visible open corners or edges.

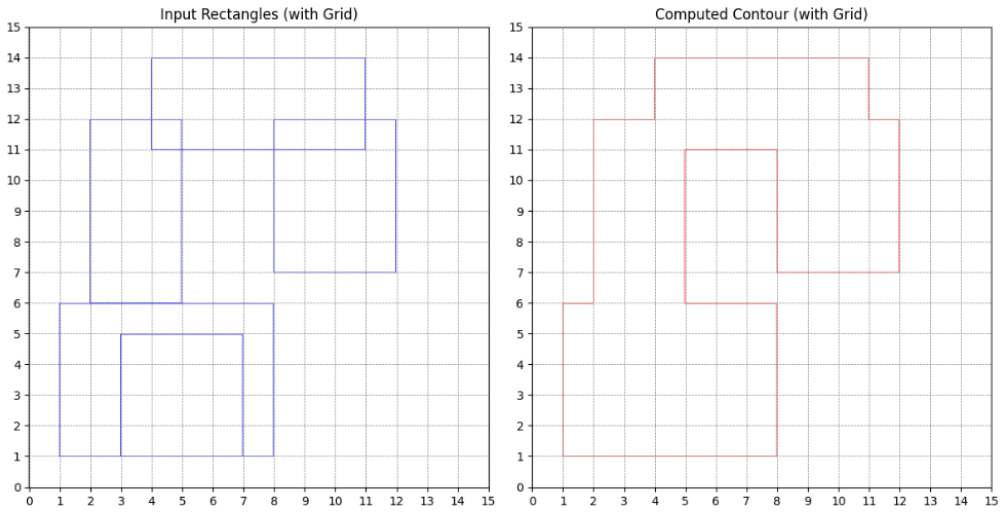

Figure 2: Overlapping Rectangles that Share Portions of their Surfaces

Observations. As shown on the left, the set of rectangles covers multiple regions but does not seal them off completely; certain interior sections remain unfilled. In the right panel, the boundary-detection algorithm capture. Because some rectangles do not connect seamlessly, the resulting contour has discontinuities where the combined coverage ends abruptly. This confirms that the method consistently recognizes every "filled-to-unfilled" transition, even when the rectangles leave partial gaps or open edges.

4.3 Case 3: Nested Rectangles (Fully Enclosed)

Setup. One rectangle is entirely contained within another. For instance: $rectangles = [[1, 1, 10, 10], [3, 3, 5, 8], [6, 3, 9, 4]]$

Expected Behavior.

- Both rectangles are "filled" in the mask.

- The outer perimeter of the large rectangle is identified as one contour.
- The inner rectangle's perimeter also qualifies as a boundary if there is a "filled-to-unfilled" transition around it (i.e., if the inner rectangle does not touch the outer one's boundary).

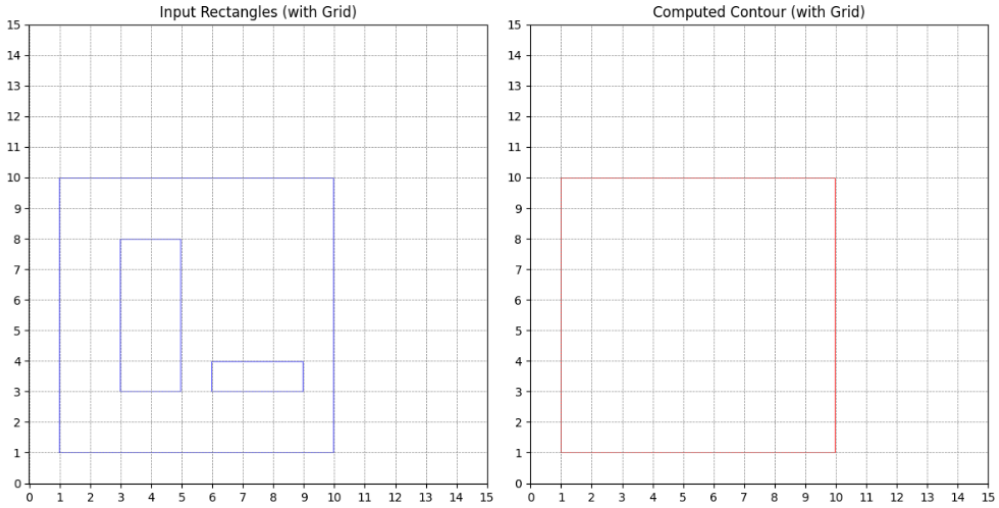

Figure 3: Nested Rectangles (Fully Enclosed)

Observations. This outcome confirms that any fully enclosed shape with a gap from the outer boundary will display an *inner* contour loop.

4.4 Case 4: Disjoint Rectangles

Setup. Rectangles are placed far apart so they do not touch or overlap, such as: $rectangles = [[1, 1, 4, 5], [5, 6, 10, 10], [6, 3, 9, 4]]$

Expected Behavior.

- Each rectangle is isolated, creating multiple disconnected regions in the mask.
- Each region should have its own closed contour, resulting in multiple disjoint boundary outlines in blue.

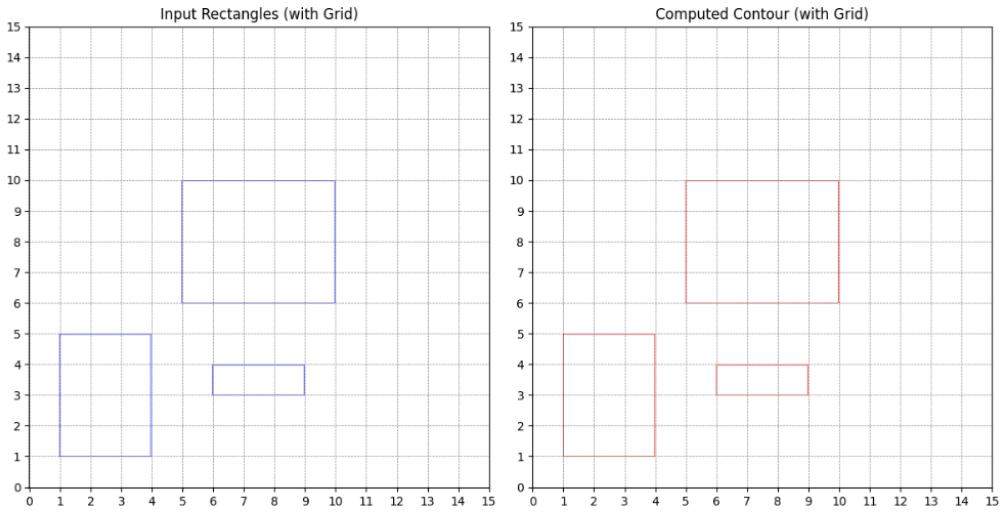

Figure 4: Disjoint Rectangles

Observations. The code produces four distinct rectangles on the left canvas, each with a red border. On the right, four separate blue contours appear, one per rectangle. The printed coordinates form four distinct groups, each representing a closed loop around an isolated region, verifying that disjoint shapes remain separately identifiable.

4.5 Case 5: Discontinuous Coverage and Gaps

Setup. Rectangles arranged to form a partial enclosure or ring-like structure, leaving internal gaps.
$rectangles = [[1, 1, 8, 6], [6, 2, 10, 8],$
$[3, 1, 7, 5], [4, 11, 11, 14], [2, 6, 5, 12], [8, 7, 12, 12]]$

Expected Behavior.

- The outer perimeter is detected as usual.
- Internal edges that face any unfilled region (the gap) also appear as a valid boundary.
- If the rectangles fail to connect completely, discontinuities in the final boundary will be visible (e.g., open corners or edges).

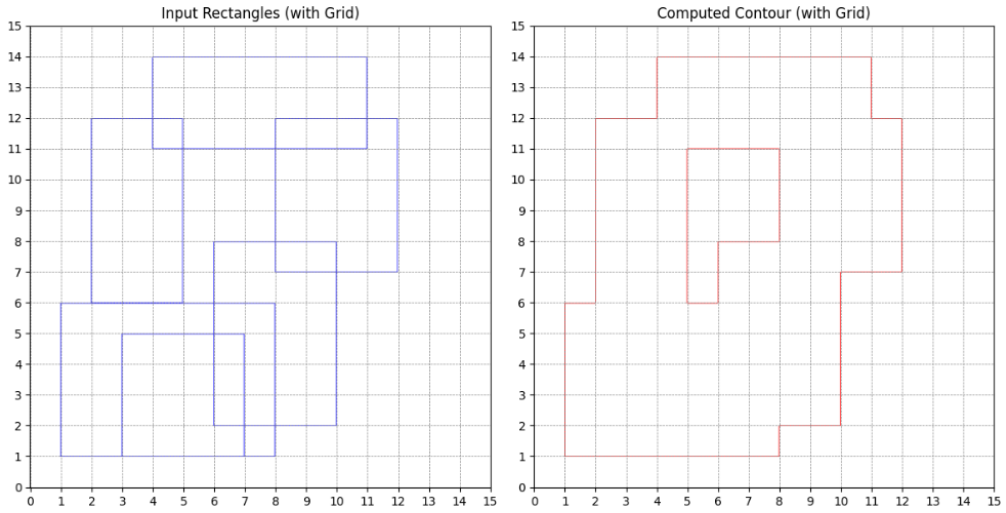

Figure 5: Discontinuous Coverage and Gaps

Observations. The right-hand image shows a large rectangular shape with a missing section (the gap). The boundary-detection algorithm traces both the outer edges and the edge around the gap (if it is enclosed on three sides and bordered by unfilled space), confirming that the approach accurately identifies *all* filled-to-unfilled transitions.

5 Conclusions and Future Work

Our pixel-based algorithm is able to compute contours for rectangular sets, showing its versatility in complex spatial scenarios. However, two key limitations exist:

1. Memory Constraints: Dense pixel grids can become very memory-intensive for large coordinate ranges, possibly limiting high-resolution applications.

2. Resolution Precision: The discrete pixel representation means boundary accuracy depends on the chosen grid size, which may not be sufficient for applications requiring exact continuous geometry.

Future research directions include developing adaptive grid strategies to dynamically refine high-detail areas, exploring hybrid approaches that combine pixel-based detection with advanced data structures like segment or interval trees, and implementing performance optimizations through parallelization and GPU-based frameworks. These strategic extensions aim to enhance the algorithm's scalability, precision, and computational efficiency

for large-scale and precision-critical use cases.

In conclusion,the method's strength lies in its ability to handle diverse scenarios—overlaps, enclosures, disjoint placements, and gaps—with intuitive simplicity.

6 References

- **Ruben Molano, Pablo G. Rodríguez, Andres Caro, Marisa Durán, Finding the Largest Area Rectangle of Arbitrary Orientation in a Closed Contour, June 2012.**

- **Paola Magillo, Lidija Čomić, Springer Journal of Mathematical Imaging and Vision, 23 September 2024.**

- **Coordinated Science Laboratory, Applied Computation Theory Group, University of Illions-Urbana,Illions, March 1980.**

- **JOURNAL OF ALQORITHMS 1, Finding the Contour of a Union of Iso-Oriented Rectangles, WITOLD LIPSKI, JR., AND FRANCO P. PICEPARATA Coorabated Science Laboratory, Uniwrsi@ of Illinois at Urbana-Chaqaign, Urbana, Illinois 61801 Received August 13,1979; revised January 3,198O.**