Computational Geometry

Animating Convex Hull Construction

Team Name
Optical Scrollers

Team Members

Kavish Vora	2024101100
Shivansh Santoki	2024101119
Meet Parekh	2024101122
Sahaj Chokshi	2024101031
Divy Padariya	2024113010

October 13, 2025

Contents

1. Project Introduction

The convex hull of a set of points is the smallest convex polygon that encloses all of them. Conceptually, if the points were nails on a board, the convex hull would be the shape formed by a rubber band stretched around the outermost nails. This fundamental structure in computational geometry has wide-ranging applications in fields such as computer graphics, robotics, geographic information systems, and data analysis.

This project aims to implement, animate, and rigorously analyze several classical algorithms for constructing the 2D convex hull. The primary goal is to move beyond static implementations and create smooth, elegant animations that provide intuitive visual insight into the operational logic of each algorithm. In addition to the visual component, the project includes the implementation of utility functions to compute the area and perimeter of the resulting hull and a test for point inclusion. The final deliverable will be a comprehensive comparative analysis of the implemented algorithms based on both theoretical and empirical data.

2. Core Algorithms for Analysis

The following algorithms form the foundational requirement of this project. They represent distinct and historically significant approaches to solving the convex hull problem.

2.1 Graham's Scan

This algorithm first identifies an anchor point (typically the point with the lowest y-coordinate). It then sorts all other points based on the polar angle they form with this anchor. Finally, it iterates through the sorted points, using a stack to maintain a chain of vertices. If adding a new point results in a non-left turn (i.e., a right turn or a straight line), points are popped from the stack until the left-turn property is restored. The algorithm's performance is dominated by the initial sorting step.

- **Time Complexity:** $O(n \log n)$

2.2 Jarvis March (Gift Wrapping Algorithm)

This is an output-sensitive algorithm that simulates wrapping a gift. It begins with an extreme point guaranteed to be on the hull (e.g., the leftmost point). It then iteratively finds the next vertex on the hull by selecting the point that has the smallest polar angle with respect to the current hull vertex. This process continues, "wrapping" the set of points until it returns to the starting vertex.

- **Time Complexity:** $O(nh)$, where n is the total number of points and h is the number of vertices on the convex hull.

2.3 QuickHull

This algorithm employs a divide-and-conquer strategy, analogous to the Quicksort algorithm. It begins by identifying two extreme points (e.g., maximum and minimum x-coordinates) which form a line that partitions the set. For each partition, it finds the

point furthest from the line. This point, along with the two original points, forms a triangle. Points inside this triangle are discarded, and the algorithm is recursively called on the points outside the two new edges of the triangle.

- **Time Complexity:** Average Case $O(n \log n)$, Worst Case $O(n^2)$.

2.4 Chan's Algorithm

A sophisticated and asymptotically optimal algorithm that cleverly combines the strengths of Graham's Scan and Jarvis March. It partitions the points into smaller groups, computes the convex hull of each group, and then runs a Jarvis-like march on the hulls themselves. Its performance gracefully adapts to the number of vertices on the final hull.

- **Time Complexity:** $O(n \log h)$

3. Additional Algorithms

To demonstrate a deeper understanding of the subject, the following additional algorithms can be implemented if required.

3.1 Monotone Chain (Andrew's Algorithm)

An elegant and efficient algorithm that improves upon Graham's Scan. It avoids angle calculations and potential floating-point issues by relying solely on cross-products. The algorithm sorts points lexicographically (by x, then y) and then constructs the upper and lower hulls of the point set in two separate passes.

- **Time Complexity:** $O(n \log n)$

3.2 Merge-based Divide and Conquer

A classic divide-and-conquer algorithm that guarantees $O(n \log n)$ worst-case performance. It works by splitting the points into two halves, recursively computing the hull for each half, and then merging the two resulting hulls by finding their common upper and lower tangents. The merge step is a non-trivial geometric operation that makes this algorithm a rigorous challenge.

- **Time Complexity:** $O(n \log n)$

4. Comparative Analysis Methodology

Our comparison of the algorithms will be multifaceted, combining theoretical analysis with robust empirical testing to provide a complete performance profile.

4.1 Theoretical Analysis

We will begin with a formal discussion of the time and space complexity for each algorithm, covering their best, average, and worst-case scenarios. We will explain the underlying reasons for these complexities (e.g., sorting bottlenecks, dependence on hull size).

4.2 Empirical Performance Analysis

This forms the core of our comparison. We will measure the actual wall-clock execution time for the hull construction phase under various conditions.

1. **Varying Input Size (n):** We will run the algorithms on randomly generated point sets of exponentially increasing size (e.g., $n = 10^2, 10^3, 10^4, 10^5, 10^6$).
2. **Varying Input Distribution:** We will test performance on datasets designed to trigger best-case or worst-case scenarios:
    - **Uniform Random:** Points scattered within a square.
    - **Circular:** Points arranged on the circumference of a circle.
    - **Clustered:** Points clustered in the center with a few outliers.
    - **Geometric Shapes:** Points arranged to form shapes like stars, lines, or grids to test edge cases.
3. **Data Visualization:** The results will be plotted on graphs (Time vs. n, Time vs. h) to visually represent the performance differences and confirm the theoretical complexities.

4.3 Functional Analysis

We will verify the correctness of our hull constructions by testing the area, perimeter, and point-inclusion functions.