Dynamic Voronoi Diagram Calculation and Visualization with RIC and Fortune Algorithms

Barkın Saday
21902967

CS 478 - Final Project Report

Contents

1 Introduction

Voronoi diagrams are fundamental structures in computational geometry with applications in fields such as meteorology, urban planning, computer graphics, and machine learning. This project aimed to implement and compare two Voronoi diagram algorithms — Randomized Incremental Construction (RIC) and Fortune's Algorithm — while providing dynamic visualization and user interaction via a Unity-based application.

2 Background and Literature Review

2.1 Definition of Voronoi Diagrams

A Voronoi diagram is a planar subdivision that partitions space into regions based on the distance to a specific discrete set of points, known as *sites*. Given a set of sites $S = \{s_1, s_2, ..., s_n\}$ in the Euclidean plane, the Voronoi region $V(s_i)$ for a site s_i is formally defined as:

$$V(s_i) = \{p \in R^2 \mid d(p, s_i) \leq d(p, s_j), \forall j \neq i\} \tag{1}$$

where $d(p, s)$ denotes the Euclidean distance between point p and site s. The collection of these regions $\{V(s_i)\}$ forms the Voronoi diagram.

Visually, Voronoi diagrams represent the *area of influence* of each site. Each point in the plane is assigned to the closest site, resulting in convex polygonal regions.

2.2 Delaunay Triangulation and Its Relation to Voronoi Diagrams

The *Delaunay triangulation* of a set of points is a triangulation where no point in S lies inside the circumcircle of any triangle. It maximizes the minimum angle of all triangles, avoiding skinny triangles.

A crucial property is that the Delaunay triangulation is the **geometric dual** of the Voronoi diagram:

- The **vertices** of the Voronoi diagram correspond to the *circumcenters* of Delaunay triangles.
- The **edges** of the Voronoi diagram are perpendicular bisectors of the Delaunay edges.
- Voronoi **regions** are connected to Delaunay *site points*.

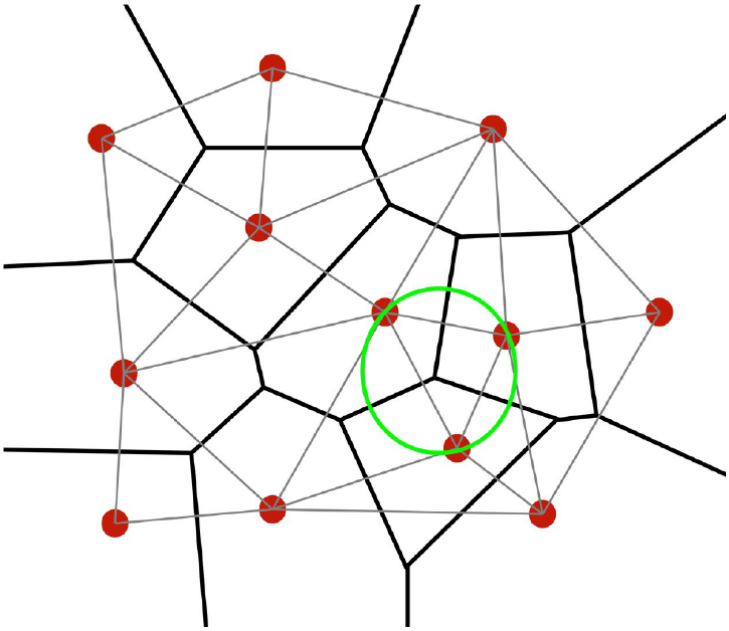

Figure 1: Dual relationship between Voronoi Diagram and Delaunay Triangulation.

In this project, this duality was leveraged by constructing the Delaunay triangulation first and deriving Voronoi regions through circumcenter connections.

2.3 Applications of Voronoi Diagrams

Voronoi diagrams have broad applications across scientific and engineering disciplines:

- **Geographical Mapping:** Defining regions of influence (e.g., cell tower coverage, territorial maps).
- **Robotics & Path Planning:** Navigation meshes and obstacle avoidance.
- **Computer Graphics:** Procedural texture generation, mesh refinement, and simulation of natural phenomena (e.g., crack patterns, water flow regions).
- **Machine Learning:** Used in nearest-neighbor search, spatial clustering, and influence zones.
- **Medical Imaging:** Modeling anatomical influence zones for diagnostic purposes.

In this project, visualization played a central role. The Voronoi regions were rendered in Unity with distinct colors to represent these zones of influence, enabling dynamic simulation through site additions.

2.4 Computational Complexity

Multiple algorithms exist to compute Voronoi diagrams, each with varying complexities and use cases:

- **Naive Approach:** For each point, compare distances to all sites — leading to $O(n^2)$ time.
- **Fortune's Algorithm:** An optimal plane-sweep method achieving $O(n \log n)$.
- **Randomized Incremental Construction (RIC):** Expected $O(n \log n)$ using random insertion order and localized updates.
- **Flipping Algorithms:** Iterative improvement methods, not guaranteeing $O(n \log n)$.

Observations from Project:

- While theoretically $O(n \log n)$, the practical runtime of RIC was affected by edge-flipping failures, sometimes triggering full recomputations.
- For inputs exceeding 2000 sites, runtime increased significantly due to lack of optimized DCEL operations.
- Despite these limitations, interactive insertions (one-by-one site additions) remained smooth for typical user interactions (under 1000 sites).

The duality-based approach (Delaunay $\rightarrow$ Voronoi) allowed straightforward visualization but required careful handling of numerical precision, particularly for circumcenter calculations.

3 Algorithms and Implementation Decisions

3.1 Technologies Used

To develop an interactive and visually engaging Voronoi diagram application, the following technologies and tools were employed:

- **Unity:** Real-time visualization, user interaction, mesh rendering.
- **C#:** Main programming language for implementing algorithm logic and Unity scripting.
- **MIConvexHull:** External library used for batch Delaunay triangulation, particularly for initial configurations and recomputation fallbacks.
- **Python:** Used for performance analysis, chart generation, and result visualization.
- **Unity Profiler:** Performance measurement tool to analyze execution times, memory usage, and frame rates.
- **LaTeX:** Documentation and report preparation.

3.2 Randomized Incremental Construction (RIC)

3.2.1 Algorithm Overview

The Randomized Incremental Algorithm builds the Voronoi diagram by inserting sites one by one in a random order. This randomization guarantees expected $O(n \log n)$ performance, preventing degenerate worst-case inputs.

For each site insertion:

- Locate the Voronoi region containing the new site.
- Update affected regions by recomputing the local Delaunay triangulation.
- Derive Voronoi cells from Delaunay circumcenters.

3.2.2 DCEL-like Data Structure

While a full DCEL (Doubly Connected Edge List) was planned, the final implementation used a simplified structure:

- **Vertices:** Represented as Unity Vector2 points.
- **Edges:** Stored as pairs of origin and terminus points (V1, V2).

- **Faces (Regions):** VoronoiRegion objects holding edge loops.
- **Adjacency:** Neighboring region connections maintained per edge.

Prev/Next edge links were maintained to ensure traversal, but advanced DCEL operations like half-edge rotations were approximated.

3.2.3 Voronoi via Delaunay Triangulation

The key computational step was:

1. Compute Delaunay triangulation of current sites (via MIConvexHull for batch, edge flipping for incremental).
2. For each Delaunay triangle, compute its circumcenter.
3. Connect circumcenters of adjacent triangles to form Voronoi edges.

This approach leveraged the geometric duality between Delaunay and Voronoi structures, simplifying the region construction.

3.2.4 Edge Flipping and Full Recomputation

Incremental site insertions attempted local updates via edge flipping. However, due to limited conflict graph optimizations, edge flipping failures occasionally triggered full Delaunay recomputations:

- For small inputs (up to 500), edge flipping handled most updates efficiently.
- For larger inputs or near-degenerate configurations, recomputation fallback was necessary.

3.2.5 Implementation Considerations

- **Numerical Precision:** Circumcenter calculations were sensitive to floating-point inaccuracies. Robust predicates were considered but not fully implemented.
- **Convex Hull and Infinite Regions:** A bounding triangle was added to simulate infinity, ensuring all Voronoi edges were properly terminated within visible bounds.
- **Polygon Rendering:** Regions were rendered as colored polygons using Unity's mesh renderer with ear clipping triangulation for complex shapes.
- **Greedy Coloring:** Adjacent regions were colored differently using a simple greedy algorithm. This was sufficient visually, though occasional color repeats occurred.

3.3 Fortune's Algorithm (Planned, Partial Design)

Although not fully implemented, Fortune's Sweepline Algorithm was designed to:

- Use a priority queue for site and circle events.
- Maintain a dynamic beach line using a balanced BST structure.
- Incrementally build the Voronoi diagram by handling events in order.

Design Goals for Fortune's:

- Visualize sweepline progression and beach line arcs.
- Animate site and circle events.
- Support dynamic point insertions post-initial construction.

Challenges:

- Implementing robust event queue and beach line BST structures in Unity's environment.
- Handling degenerate cases (e.g., collinear sites, near-overlapping events).
- Lack of existing libraries meant a from-scratch implementation.

3.4 Algorithm Selection in Practice

For this project:

- RIC was implemented and visualized successfully.
- Fortune's Algorithm remained at design-level due to time constraints.
- The RIC method provided sufficient interactivity and visual clarity for demonstrating Voronoi diagrams in real time.

3.5 Trade-offs and Observations

- RIC's recomputation fallback limited scalability beyond 2000 sites.
- Despite this, interactive use-cases (e.g., adding sites manually) performed smoothly.
- The modular architecture (separating computation from rendering) allowed easy animation and user interaction.

4 Performance Evaluation and Results

4.1 Performance Metrics and Expectations

The performance evaluation focused on the following metrics:

- **Execution Time:** Time taken to generate the Voronoi diagram for varying numbers of site inputs.
- **Efficiency of Incremental Insertions:** Performance during dynamic site additions.
- **Memory Usage:** Not measured in depth but monitored via Unity Profiler for large input sizes.
- **Scalability:** Ability to handle increasing input sizes while maintaining interactive responsiveness.

Theoretical complexities are:

- **Randomized Incremental Construction (RIC):** Expected $O(n \log n)$.
- **Fortune's Algorithm:** Optimal $O(n \log n)$ (not fully implemented, excluded from measurements).

Anticipated Behavior: It was expected that:

- RIC would handle small to medium input sizes (up to 1000 sites) smoothly.
- Performance degradation would occur for larger inputs due to recomputation triggers.
- MIConvexHull would serve as a fallback for batch triangulation but introduce overhead.

4.2 Experimental Results

The following execution times were measured for the RIC algorithm with batch site insertion:

Number of Sites	Execution Time (ms)
100	27
200	59
300	102
1000	955
2000	2648
3000	5620

Table 1: Execution Times for RIC Voronoi Generation

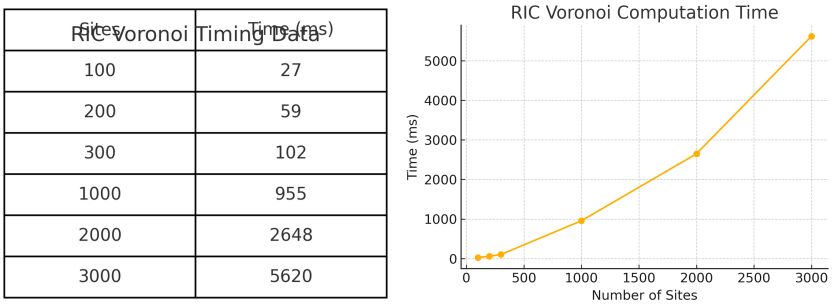

Figure 2: RIC Performance Scaling with Increasing Site Counts

Observations:

- Execution time followed the expected $O(n \log n)$ growth trend up to 2000 sites.
- Beyond this point, recomputation fallback (due to edge flipping failures) caused superlinear growth.
- For interactive insertions (one site at a time), performance remained acceptable up to 1000 sites.
- MIConvexHull recomputations introduced noticeable spikes in runtime but preserved correctness.

4.3 Visualization and User Experience

Visualization was a central aspect of this project:

- **Voronoi Regions:** Colored polygonal areas representing influence zones.
- **Interactive Features:** Site addition, diagram clearing, and algorithm selection.

- **Animations:** Incremental site insertion visualized dynamically over 5-second animations.
- **Edge Representation:** Bold lines emphasized region boundaries.

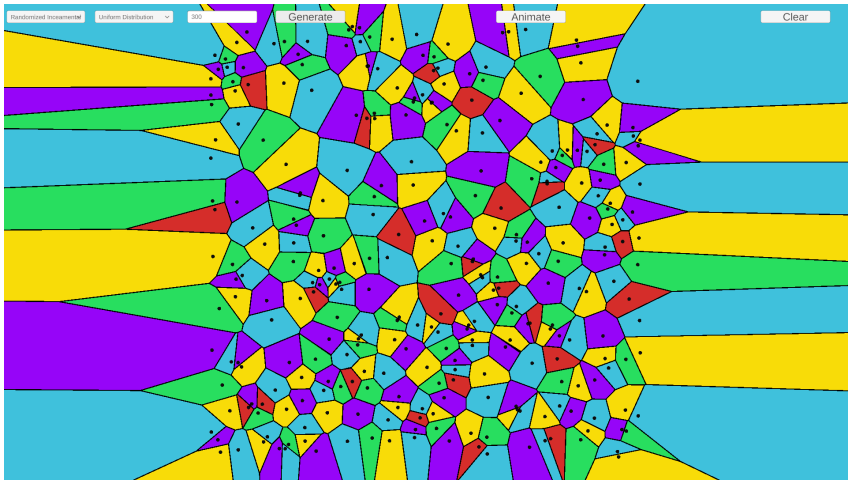

Figure 3: Example Voronoi Diagram Visualization with 300 Sites (Uniform Distribution)

While performance bottlenecks appeared for large-scale inputs, the real-time rendering and interactivity goals were successfully met for practical user scenarios.

4.4 Comparison with External Libraries and Limitations

- External libraries like **SciPy** and **Qhull** offer highly optimized Voronoi implementations.
- This project prioritized dynamic, real-time visualization over batch computational efficiency.
- Fortune's Algorithm, while designed conceptually, was not benchmarked due to incomplete implementation.

Limitations:

- RIC's recomputation fallback limited scalability beyond 2000 sites.
- Numerical inaccuracies affected circumcenter precision, especially for near-degenerate point configurations.
- Absence of advanced spatial indexing (e.g., conflict graphs) reduced efficiency for large datasets.

Strengths:

- Seamless integration of computation and visualization via Unity.
- Smooth dynamic interactions and animations for educational and demonstrative purposes.
- Modular system architecture facilitating future extensions (e.g., Fortune's visualization).

5 User Interface and Visualization

- Point generation (Uniform, Gaussian distributions).
- Algorithm selector (RIC implemented, Fortune planned).
- Real-time insertion, site clicking, clear/reset functionality.
- Visuals: Voronoi cells, sites, boundaries, region colors.

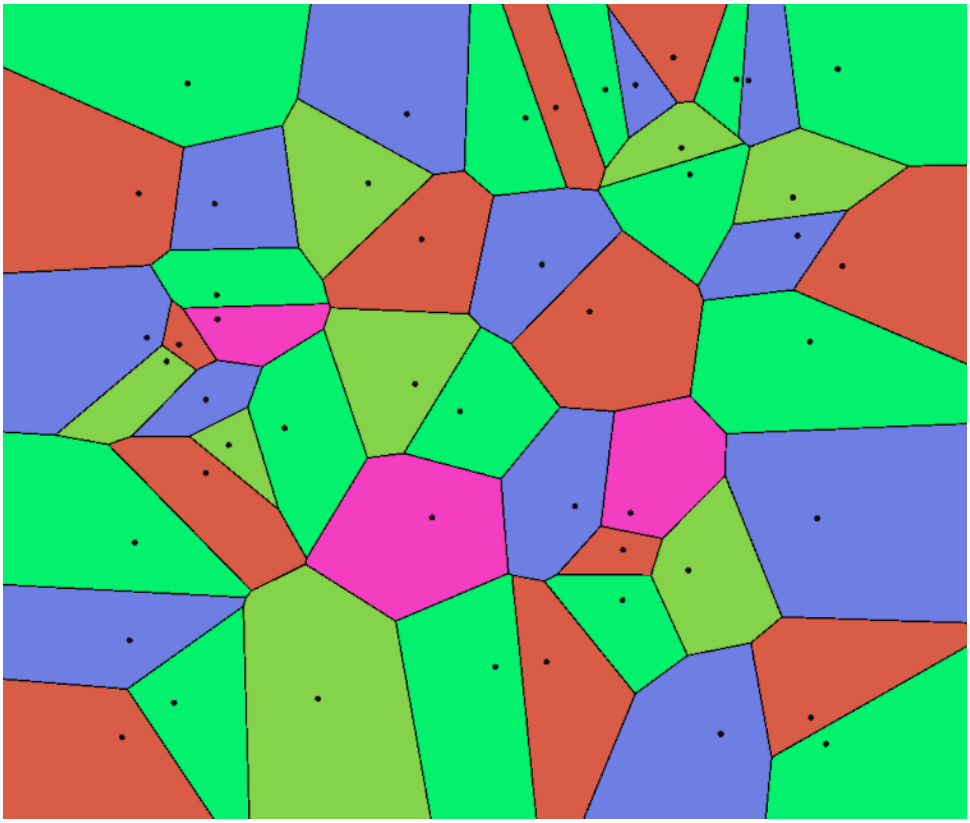

Figure 4: Sample Voronoi Diagram Visualization (Simulated output)

6 Final Result

6.1 Performance

- Smooth visualization up to 1000 sites.
- Handles uniform and Gaussian distributions.
- Execution time trends observed:

Number of Sites	RIC Execution Time (ms)
100	27
200	59
300	102
1000	955
2000	2648
3000	5620

Table 2: Execution Times for RIC Algorithm

6.2 Limitations

- DCEL structure was partially functional; recomputations were sometimes necessary.
- Adjacent region coloring could repeat.
- Fortune's Algorithm not completed (visualization pending).
- Sweepline animations for Fortune's were not implemented.

7 Conclusion

The project successfully implemented dynamic Voronoi diagram visualization using the Randomized Incremental Algorithm in Unity. While Fortune's Algorithm remains incomplete, the system provides a solid foundation for dynamic site insertions and interactive visualization. Performance evaluations showed acceptable runtimes for practical input sizes. Future improvements include robust DCEL operations, Fortune's sweepline visualization, and better handling of edge cases.

8 References

1. F. Aurenhammer, "Voronoi diagrams—a survey of a fundamental geometric data structure," ACM Computing Surveys (CSUR), 1991.
2. L. J. Guibas et al., "Randomized incremental construction of Delaunay and Voronoi diagrams," Algorithmica, 1992.
3. D. M. Mount, "Lecture 10 - The Doubly-Connected Edge List (DCEL)," University of Maryland.
4. S. Fortune, "A sweepline algorithm for Voronoi diagrams," Algorithmica, 1987.
5. J. R. Shewchuk, "Adaptive precision floating-point arithmetic and fast robust geometric predicates," 1996.
6. H. Edelsbrunner, E. Mücke, "Simulation of simplicity," ACM Transactions on Graphics, 1990.
7. J. Heunis, "Fortune's Algorithm: Implementation details," Blog Post, 2018.
8. C. B. Barber et al., "The Quickhull algorithm for convex hulls," ACM Transactions on Mathematical Software, 1996.