2023

Visualization Teaching Tool for Computational Geometry Algorithms

Seth Spire
sspire2@eagles.bridgewater.edu

Visualization Teaching Tool for Computational Geometry Algorithms

Seth Spire

App Link: https://computational-geo.onrender.com (as of 4/19/2023, email sspire2@eagles.bridgewater.edu for more details)
GitHub Link: https://github.com/sethspire/computational-geo

I. INTRODUCTION

Computational geometry is a branch of computer science dedicated to the study and development of algorithms which solve geometry problems (de Berg et al., 2008, p. 2). Algorithms are sets of instructions that are run on a given input to produce a desired output such as sorting a list of objects. The goal for computational geometry is to take a geometry problem (like finding the shortest path between points) and its corresponding geometric object (like a group of points or shapes on a plane) to formulate an algorithm which solves it in an efficient, elegant, and exact manner.

The general study of algorithms is certainly one of the harder courses in computer science that many do struggle with. Computational geometry is a graduate-level course often taken in a master's program, so these algorithms are often even harder to conceptualize and understand. Therefore, the goal of this project is to create a web app which will allow users to visualize and learn about computational geometry. Users can test different inputs and watch how the algorithm creates a solution alongside a description of what is happening. This web app will make learning computational geometry and the principles of some advanced algorithms easier. Altogether, this should end up being a great tool to be used in classrooms.

The project focuses on a few key problems and algorithms in computational geometry: convex hull, line segment interaction, triangulation, and Voronoi diagrams. These problems are foundational to the field of computational geometry and their respective algorithms that have been chosen to be implemented use many of the key principles used to solve the various problems.

II. PROCESS

Below is an outline of the general process I went through in order to develop the web app for this project. It covers choices made, when different parts of code were written, and what went into those tasks.

1. Select the graphics format and libraries to be used in creating visualizations:
    a. Scalable Vector Graphic (SVG) chosen for scalability, animation, and interactivity
    b. D3.js and svgdotjs used for ability to manipulate SVGs efficiently

2. Select type of web app and find hosting for it
    a. Node app hosted on Cyclic which is both fast and free
3. Create basic web app
    a. Design a clean, simple, user-friendly site to make it as accessible as possible
4. Create general capability to add to and edit an SVG canvas
    a. Add elements (points/lines/polygons) and/or delete upon user clicking screen
    b. Elements are given a unique ID based on its coordinates to make easily queryable
    c. Also, the graph area can be resized, made full screen, and cleared
5. Define format for storing visualization data of each step in the algorithm
    a. Chose a list with each index containing data for the state of its respective step
    b. Each entry contains a JSON object of what elements are changed, what attributes of each element change, what those attributes previously were (for going back), and what the pseudocode should display
    c. Storing only what changes rather than all data for every single element greatly saves on storage space, limits what has to updated, and vastly improves efficiency
6. Design functions to display the visualization as it changes
    a. User has control over visualization: play/pause to automatically advance steps, skip directly to end or beginning, and move forward or back 1 step
    b. All the data in the JSON object for a step is parsed to be sent to separate functions for updating points, lines, polygons, the progress bar, and pseudocode
7. Create functions to create visualizations of the various algorithms
    a. Research the algorithm to figure out how to implement it
    b. Write the algorithm such that it stores the visualization data at each step
        i. It repeatedly sends updated visualization data to various functions (for points, lines, polygons, etc.) which creates and appends a JSON object to the list of states
8. Write descriptions for problems and algorithms; provide tutorial on how to use the app
    a. Allows users to see code/descriptions of what the visualization is showing
    b. Users also would need a tutorial to explain how it to use it and what it can do to make it useable and accessible
9. Refactor and bug fix and repeat

III. ALGORITHMS AND PROBLEMS

A. CONVEX HULL

The convex hull problem is to take a list of points and output the smallest and simplest polygon that contains every point on the plane (Mount, 2021, p. 7). It is like if one were to wrap a rubber band around the points. See Figure 1 below for the visualization.

The web app currently implements the Graham Scan algorithm (Mount, 2021, p. 10-13). It works counter-clockwise from the leftmost point and iteratively adds points. If the angle of the last 3

points added is not convex (does not point outward), it removes middle of those 3 points to alleviate that issue.

A key part of developing this algorithm was implementing helper functions used in ordering points counter-clockwise and to check the angle of 3 points. The ordering part is done by sorting based on the polar angle between the left point and each point using the *arctan2* function: $atan2(y_2-y_1, x_2-x_1)$ where the left point is (x_1, y_1) and the other points are denoted (x_2, y_2). Finding if an angle (comprised of 3 points p_1, p_2, p_3) is convex works by getting the cross product of the vectors p_1p_2 and p_1p_3: $((x_2-x_1)*(y_3-y_1)) - ((y_2-y_1)*(x_3-x_1))$. If the cross product is positive, then the angle is convex (assuming the 3 points are in counterclockwise order.

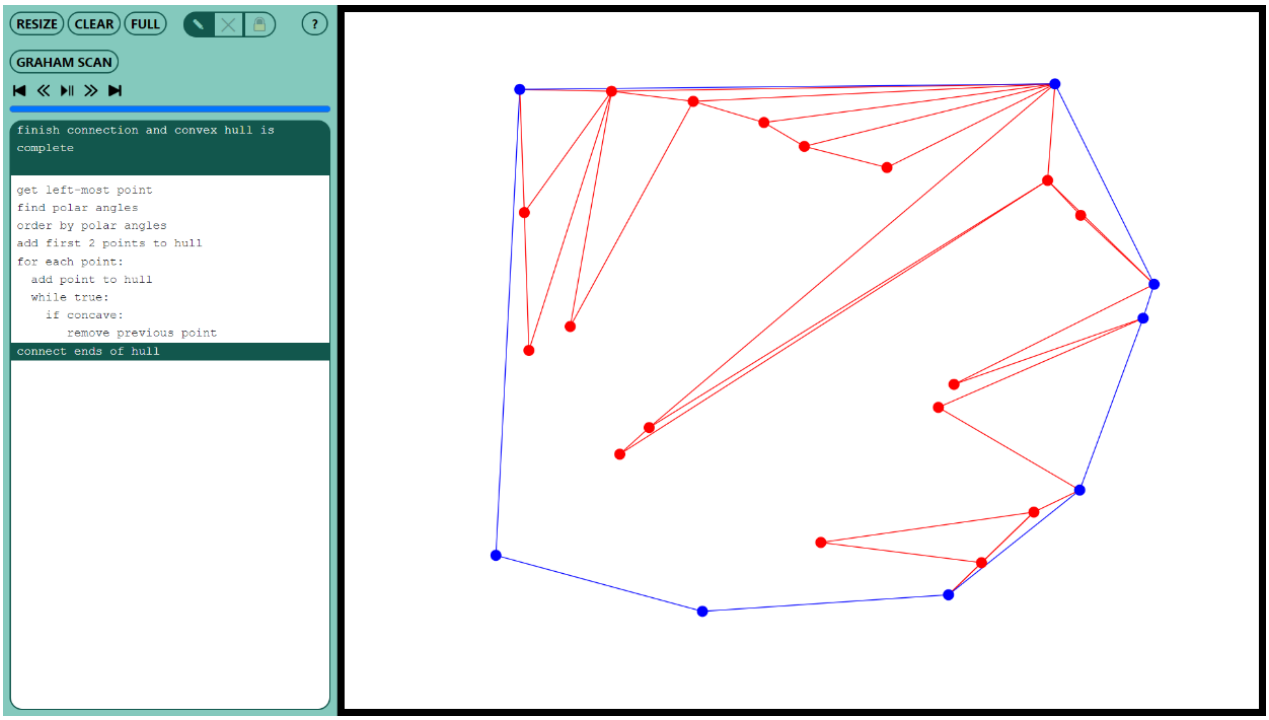

Figure 1. Screenshot of completed algorithm visualization for a Graham Scan getting the convex hull (blue points and lines).

B. LINE SEGMENT INTERSECTION

The line segment intersection problem is simply to take in a list of line segments and return a list of all the intersection points (de Berg et al., 2008, p. 21-22). This general idea of finding line segment intersections is at the heart of collision detection for video games and some forms of autopilot. See Figure 2 below for the visualization.

The algorithm currently implemented is known as a line sweep algorithm (de Berg et al., 2008, p. 21-29). It can be thought of as sliding a vertical line left to right where it tracks which lines segments cross it, ordered top to bottom. Then, only the segments on that list that are newly

made neighbors need to be checked for intersection. Unless two segments—at some point—cross the sweep line at the same time and are next to each other in the order list top to bottom, the two segments cannot intersect.

Checking if two segments intersect involves solving the system of the two linear equations for those segments and making sure the returned point falls in the range of both segments. Maintaining the list of points where the sweep line stops at required the development of a specialized priority queue. A priority queue was chosen over a regular sorted list because the intersection points are dynamically added to it as well when they are found by the algorithm. The data of which line segments are active on the sweep line is stored in a AVL binary search tree implemented for storing these segments. Since the value sorting the segments is the y-value where it crosses the sweep line, the comparator used in the AVL tree is dynamic: it is a multi-line function run which returns that value every time a comparison to said segment is needed.

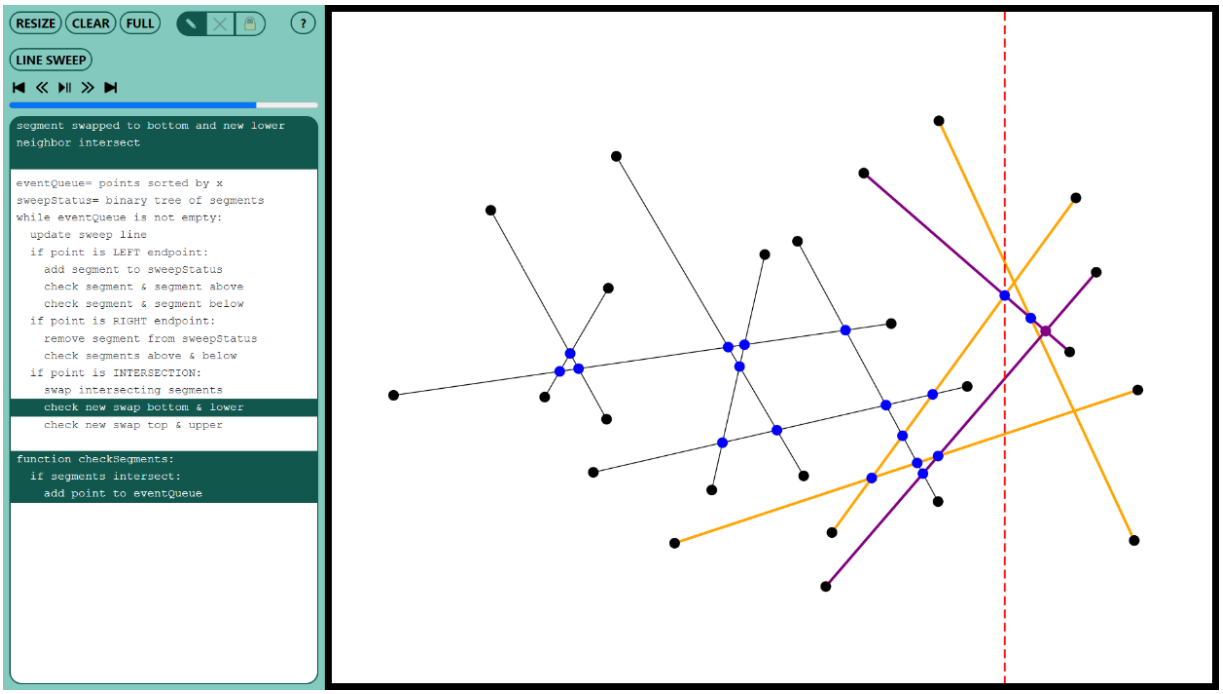

Figure 2. Screenshot of in-progress visualization of a line sweep algorithm to obtain line segment intersections. Blue points are found intersections. The red dotted line is the line sweep. The purple marks the segments being checked for an intersection. The orange segments are the other active segments (along with the purple ones) that are currently crossing the line sweep.

C. TRIANGULATION

The problem of triangulation is to take a polygon and subdivide it into a group of triangles (Mount, 2021, p. 32). This process is the basis of other algorithms that require a larger polygon to be broken down. For example, calculating the area of any polygon can be done by summing the areas of each triangle that makes it up. See Figure 3 below for the visualization.

The ear clipping algorithm is currently implemented (Garton). It finds which points are convex (angle points outward) and have no other points fall in the triangle formed by said point and its two neighbors. If both of those things are true, then it is an 'ear tip' that can be removed (clipped) successfully without issue. The algorithm iteratively clips off one ear at a time and updates the 2 neighbors to check if they are ear tips or not until the triangulation is complete.

This uses the same function as Graham Scan to check if a point is convex. Checking if a point (*P*) falls within the triangle formed by 3 different points (*A*, *B*, and *C*) is done by calculating the area of various triangles. First, the area of the triangle *ABC* is calculated. Then, the area of the triangles formed by *P* and 2 of the triangle points (*PBC*, *APC*, and *ABP*) are calculated. If the sum of areas of *PBC*, *APC*, and *ABP* equals the area of *ABC* then the point must be in the triangle.

To make the algorithm more efficient, when checking if a convex point is an ear tip, only the concave points need to be checked to see if they fall within the triangle since if at least 1 point is in the triangle, one of them must be a concave point.

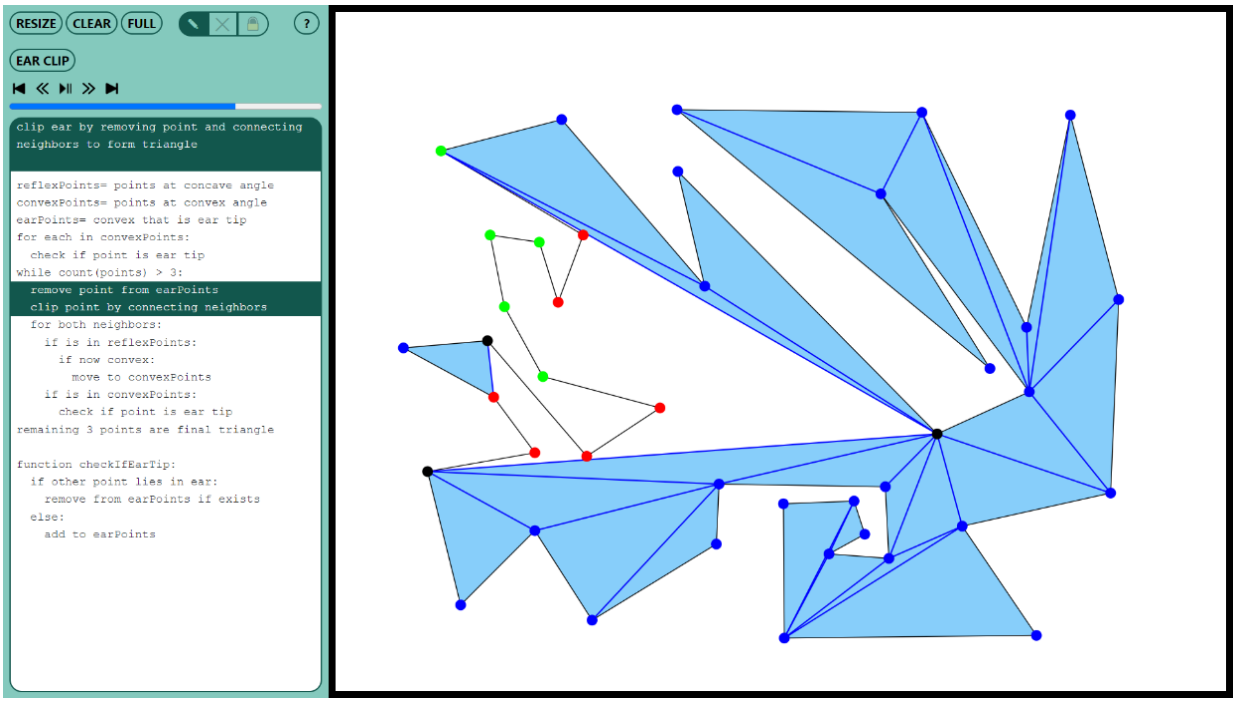

Figure 3. Screenshot of an in-progress visualization of the ear clipping algorithm for finding a polygon triangulation. Blue triangles have already been found. Green points mark where future triangles can be located currently. Red points are concave points that cannot be removed yet.

D. VORONOI DIAGRAM

The Voronoi diagram problem is to take a list of points and subdivide the entire plane into sections (cells) for each point where the entire area of that section is closer to its corresponding point than any other (Mount, 2021, p. 66). This has been used in city planning to check which area should make up each school zone.

The incremental algorithm that has been implemented involves iterating over the points and adding them to the diagram one at a time (Sacristán). When a point is added to the diagram, it is compared to all other current points. The perpendicular bisector (line that runs in the middle of 2 points at a right angle to the line segment made by those 2 points) of the new point and every other point already added to the diagram is calculated. The perpendicular bisectors divide area into which point it is closest to. When a line segment associated with the cell being checked is intersected by the perpendicular bisector then that segment can be cut to the side matching the cell being checked. The points of intersection can be connected to form the barrier segment between the new cell and cell being checked. On the other hand, if a segment is not intersected but falls entirely on the new cell side of the perpendicular bisector, then it can be removed (unless it is part of the border in which case it joins the new cell). See Figure 4 below for the visualization.

Checking if two segments intersect is the same as with line segment intersection. Checking what side of the perpendicular bisector a segment falls on is done by finding the turn from the endpoint of the perpendicular bisector (P_1 and P_2) to both the segment endpoints and the new point added to the diagram (all which are denoted P_3) using the following formula (which was also used to test if a point is convex in other algorithms): $((x_2-x_1)*(y_3-y_1)) - ((y_2-y_1)*(x_3-x_1))$. If the signs of all the turns (more specifically, cross products) calculated match, then the new point and the segment are on the same side of the perpendicular bisector.

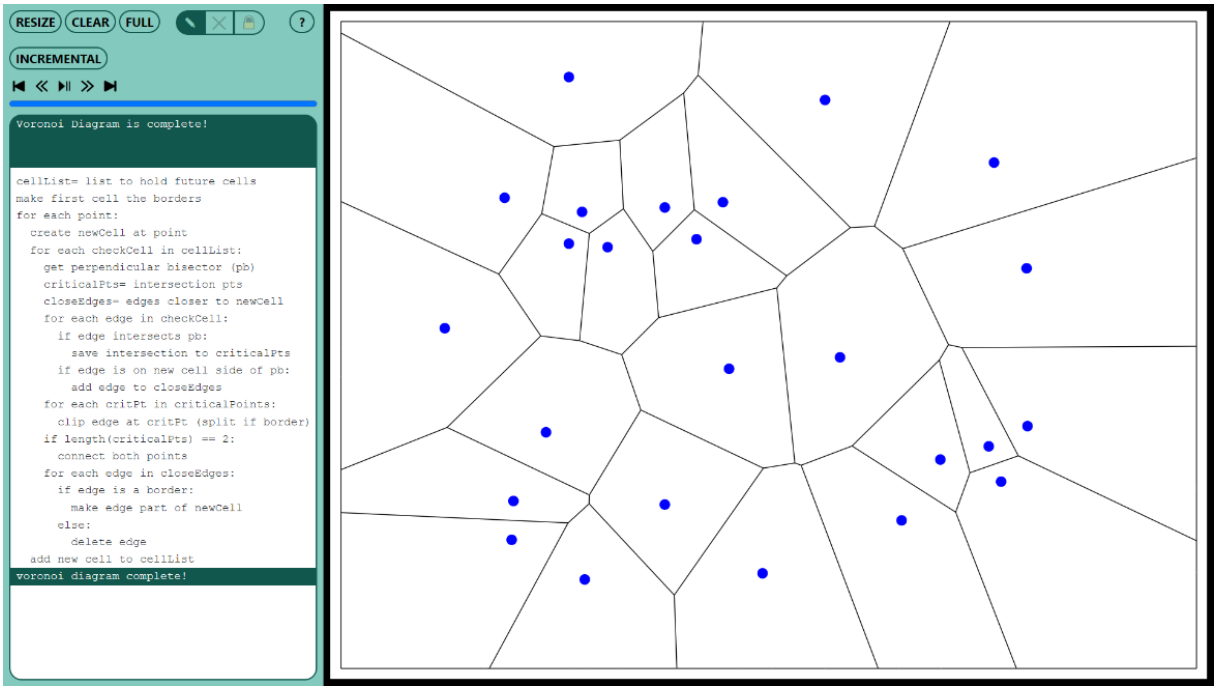

Figure 4: Screenshot of a completed visualization of a Voronoi Diagram.

IV. CONCLUSIONS

This project is the largest overall project I have personally created, both in hours given and in lines of code I personally wrote (in the range of 5500 lines). Learning to manage a large project was extremely beneficial and sometimes overwhelming. I had to make a substantial change to the structure of the code halfway through to make the data and functions for managing the state of the visualization useable for different types of visualizations and various different files. However, if I had planned better, I would have seen the issues beforehand. I definitely learned from experience. Also, I now have a great knowledge of the core concepts of computational geometry. I got to learn a bunch of new algorithms, concepts, and technologies which I will continue to use far into the future.

In the end, I think the web app is a fairly user-friendly, instructive way for people to learn about these complex algorithms. Users can see the points and lines moving on the screen alongside the code to see exactly what each step of the algorithm is doing. Those learning about computational geometry can see dynamic renderings rather than simple, static images in a textbook. This is ideal for learning; this web app seems perfectly set up to be a learning tool for future instructors explaining these algorithms.

However, for the future of this app, all of the algorithms, visualizations, and writeups need continued refining to make them as clear and accessible as possible. I need to put a much greater focus on ensuring that the documentation, pseudocode, and descriptions given are

comprehensible and are actually what people need in order to learn these various algorithms. Also, the animations are not fully working as I want them to be, so that code needs updating. They move step by step perfectly, but there is no transition period where, for example, a segment is added and it extends to its full length. I plan to maintain this web app and add more algorithms to it over time. Then it could be used to compare how different algorithms solve the same problem, often in very different ways. Overall, this has been a very beneficial project for my own knowledge and I hope that this can one day be used by other to learn many of the same concepts and algorithms I have.

V. REFERENCES

de Berg, M., Cheong, O., van Kreveld, M., & Overmars, M. (2008). *Computational geometry: Algorithms and applications* (3rd ed.). Springer.

Garton, Ian. “Ear Cutting for Simple Polygons.” *Computational Geometry - Ear Cutting for Simple Polygons*, McGill University, http://www-cgrl.cs.mcgill.ca/~godfried/teaching/cg-projects/97/Ian/cutting_ears.html.

Mount, D. M. (2021). *CMSC 754: Computational geometry* [PDF]. University of Maryland CMSC 754: https://www.cs.umd.edu/class/fall2021/cmsc754/Lects/cmsc754-fall-2021-lects.pdf.

Sacristán, Vera. *Algorithms for Constructing Voronoi Algorithms* [PDF]. Facultat d’Informàtica de Barcelona, Universitat Politècnica de Catalunya: https://www.ic.unicamp.br/~rezende/ensino/mo619/Sacristan,%20Voronoi%20Diagrams.pdf.