Simulating 2D Voronoi Diagrams with C# and Unity

A-level Computer Science Project

James Richardson

Candidate Number: 8484
Center Number: 16437

November 18, 2022

Contents

Chapter 1

Introduction

"In the beginning, the universe was created. This has made a lot of people very angry and has been widely regarded as a bad move" [Adams, 1980]. Within this universe, and specifically this paper, I shall create a program that can simulate Voronoi tessellations using various algorithms with C# and the Unity framework with the aim of being user-friendly and allowing for some determination of the optimum method in terms of programming complexity, processing requirements and quality of simulation.

I am undertaking this investigation for a variety of reasons:

- The Voronoi diagram is seemingly ubiquitous throughout the world and appears in a vast array of fields of study including (but not limited to) biology, astrophysics, ecology, meteorology, engineering and bakery. Despite this I am yet to find anyone within my social circle who has ever heard of them and so through creating this project I hope to draw attention to this, in my opinion, underappreciated and fascinating field of mathematics.
- Upon my first introduction to the Voronoi diagram I soon found there are a variety of ways to compute it but nowhere I looked provided any form of explanation as to which method was best to use. This lapse in comparative usefulness is bound to only hinder further progression in the field and fields which use it. Thus I believe it to be very useful to determine the strengths and weaknesses of each computation method.
- There are not many user friendly programs for simulating Voronoi diagrams and so I would like to make one which is both powerful and very easy to use for someone with no programming knowledge.
- For a more personal reason I have always found Voronoi diagrams to be somewhat mesmerising ever since I first encountered them in a video by Sally Le Page [Page, 2019]. The ability to expand radially from points and to form straight line intersections has been stuck in my brain for about 2 years now and this project is a good way to scratch that itch.

Chapter 2

Voronoi Diagrams

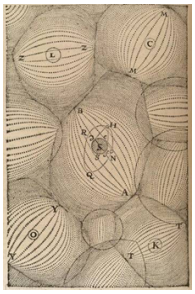
A Voronoi diagram (also known as: Dirichlet tessellations or Thiessen polygon maps) is a diagram that shows the regions for which a point in a given set of points is the closest (an example is seen right [Ertl, 2015]). It is named after Georgy Voronoy who is primarily credited for the introduction of the concept for n number of dimensions in 1908 [Voronoi, 1908]. As such they are a relatively new analytical tool and are yet widely used, as shall be shown, in a variety of academic disciplines with a variety of functions.

There are two types of voronoi diagram, Euclidean and Manhattan.
Euclidean distance refers to the straight line distance between two points, It is the square root of the sum of the squares of the relative x and y distances between two points, using Pythagoras.
Manhattan distance instead uses the sum of the absolute x and y distances between two points.
For example, imagine a square grid with two points A and B. Point B is 3 squares above and 4 squares to the right relative to A. Their Euclidean distance uses Pythagoras ($a^2 + b^2 = c^2$) to give a distance of 5 along the diagonal between them.
Manhattan distance instead gives a distance of 7, not allowing any diagonal movement.

For the purposes of our program we will only be considering Euclidean based voronoi diagrams although it is worth noting that both exist.

2.1 A Brief History of Voronoi Diagrams

The earliest generally accepted case of a Voronoi diagram is traced to the work of René Descartes in 1644 during his work on his vortices theory. This theory surrounded the description of the universe as matter rubbing against each other with the types of matter grouped into luminous, transparent and opaque [loc.gov]. Descartes's diagrams showing this theory are comparable to later Voronoi diagrams (although they were not defined the same way) as they showed regions surrounding a central point forming boundaries where they met.



A depiction of Rene Descartes's vortices [loc.gov]

The first major attempt at formalising the concept was undertaken by Peter Gustav Lejeune Dirichlet in 1850 when he was investigating positive quadratic forms [Weisstein]. He formalised the definition of these diagrams for 2 and 3 dimensions, however, it wasn't until Georgy Voronoy in 1907 that the concept was formalised to n number of dimensions.

Even before their formal definition, Voronoi or Voronoi-esque diagrams have been used in many contexts. One very famous example is the work of John Snow in his "Report on the Cholera Outbreak in the Parish of St. James, Westminster, During the Autumn of 1854" which included a map showing a continuous broken line called the "Boundary of equal distance between Broad Street Pump and other Pumps", which creates a voronoi diagram around the Broad Street Pump [Okabe et al., 2009, p. 9]. This diagram was used to trace the localised cholera outbreak to the Broad Street pump and eventually resulted in the handle of the pump being removed (although it was later replaced).

John Snow's map with his line made solid for clarity [Snow]

In the 1960s there was a rediscovery of the Voronoi diagram within ecology with reference to estimating the intensity of trees in a forest by describing each tree as a voronoi region for area potentially available. The next year another paper was written using the same concept for plants labelling the regions as "plant polygons" The usage is now commonplace throughout the field [Okabe et al., 2009, p. 9].

2.2 Applications

2.2.1 The Post Office Problem

"Given a finite set P of distinct points, find the nearest neighbour point among P from a given point p (p is not necessarily a point in P)." [Okabe et al., 2009, p. 61]

A famous example of the usage of Voronoi diagrams is what Donald Knuth described as the *post office problem*. It describes a problem in which, given a set of points P, you must find the closest point with reference to another point p. The post office analogy describes this by imagining a scenario in which the closest post office needs to be found with reference to a particular house or post box.

This problem can be solved through the usage of a Voronoi tessellation with the nodes representing the post offices and the regions representing the regions for which that post office is the closest.

This principle of using Voronoi diagrams to find the closest node in a set is a common theme throughout uses with similar examples including ambulance stations in relation to an injury and cell phone towers in relation to a phone.

2.2.2 Astronomy

In a paper by M.N. Drozdovskaya at Leiden University (nl), Voronoi tessellation has been used via Brown's algorithm within the field of Astronomy to aid in modelling the Cosmic web (I promise that's a real thing) [Drozdovskaya, 2010].

2.2.3 Baking

Voronoi diagrams are often seen in baking where borders are found between items baked on one tray. A common example of this is burger buns in a pack, the buns are connected together with straight lines because they were baked next to one another resulting in them expanding and fusing together.

2.2.4 Biology

It is becoming increasingly common to model the arrangement of cells with voronoi tessellations such as that investigated by Martin Bock et al in their paper [Bock et al., 2009]. A slightly less complicated use of a similar nature surrounds the growth of cell colonies in controlled environments. The cell colonies expand radially and when they meet (assuming they can safely coexist) they will form a voronoi tessellation with each other.

Voronoi tessellation has also been used to model bone architecture, in particular by Hui Li et al at the University of New York (usa) [Li et al., 2012].

2.2.5 Ecology

In 1997 F. Mercier from the Claude Bernard University and O. Baujard from the University of Geneva used a modified version of the Green and Sibson algorithm to generate a voronoi tessellation for the canopies of trees allowing them to investigate population dynamics of trees in a Guianan forest. Using the Green and Sibson algorithm allowed them to look at canopy openings and the formation of trees within these openings over time with trees being added and removed as required [Mercier and Baujard, 1997].

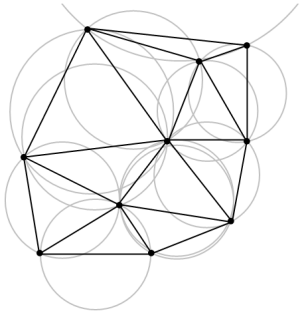
2.3 Methods of Generating Voronoi Diagrams

Below I shall be listing some of the methods of computing Voronoi diagrams that are commonly used. This is by no means an exhaustive list (not even close) and new algorithms are being developed frequently, the ones described here are those which I deem to be the ones what appear most frequently and are the ones I shall attempt to implement later on[1].

2.3.1 Via Delaunay Triangulation

2.3.1.1 What is a Delaunay triangulation

A Delaunay triangulation for a set of points is a triangulation such that none of the points are within the circumcircle of any triangles within the triangulation (A circumcircle being a circle for which all vertices on the polygon are also on the circumference of the circle). It is frequently used as "a technique for creating a method of contiguous, non-overlapping triangles from a dataset of points" [Duh]. The process favours short, wider triangles with long, thin ones being generally avoided. The process was invented by Boris Nikolaevich Delaunay in 1934, Georgy Voronoi was one of his doctoral advisors.



A Delaunay triangulation with circumcircles [es]

2.3.1.2 Using Delaunay for Voronoi

The Delaunay triangulation and the Voronoi diagram are mathematical duals or each other.
Duality in mathematics refers to a relationship in which two states can be switched between by applying the same function.
$f(x) = b \quad \& \quad f(b) = x \quad \& \quad ff(x) = x \quad \& \quad ff(b) = b$
The transformation that converts between Voronoi and Delaunay (v.v.) is the same for all dual graphs. It takes the centre-points of the regions of the graph and draws lines between each centre-point and the centre-points of all adjacent regions.[Weisstein]

2.3.2 The Green & Sibson Algorithm

The Green & Sibson algorithm is a recursive method that adds the seeds one at a time. I shall attempt to summarise the process below but for a more detailed (and probably more coherent) explanation I refer you to their original paper cited here [Green and Sibson, 1977][2].

The Green and Sibson algorithm treats a voronoi diagram as a set of points (seeds) and connections with adjacent regions, on the premise that once those are known all else can be extrapolated.
The diagram is generated by adding points one at a time in an anticlockwise fashion, "recursively modifying

[1] Some other algorithms not covered that you may be interested in:
Shamos and Hoey http://euro.ecom.cmu.edu/people/faculty/mshamos/1975ClosestPoint.pdf
Brown https://www.math.wustl.edu/ victor/classes/pmf/KQBrown.pdf
Yan et al. https://www.microsoft.com/en-us/research/wp-content/uploads/2016/12/Efficient-Computation-of-Clipped-Voronoi-Diagram-and-Applications.pdf

[2] As a quick aside, Green and Sibson win my award for best paper I've had to read. It's clear, concise, doesn't go too deep and actually stops after 3 dimensions whilst also covering problems like adding boundaries AND discussing implementations not simply leaving it in the realms of theory. And the only word I had to google was "contiguous". Utter heroes.

the contiguities as each point is added"[Green and Sibson, 1977]. When a new point is added the first point to check is its nearest, already added neighbour as it is guaranteed that there will be a connection with that point's region. The perpendicular bisector of the new point and its nearest neighbour is found and is followed clockwise to the edge of the nearest neighbour's tile. The edge at which it meets gives the next point for which there is a connection which is stored as a connection and the process is repeated, creating a perpendicular bisector with the next point and following it clockwise until the next tile is met, and so on until returning to the nearest neighbour.

The algorithm has a big O of $O(n^2)$ and (if the amount of times I've found it referenced is any indication) is considered to be a very effective algorithm.

2.3.3 Steven J. Fortune's Algorithm

Fortune's algorithm is a very common method of computing a Voronoi diagram in O(nlogn) time although the theory behind the algorithm is somewhat complicated to understand[3]. I have written an explanation but I would recommend looking at other sources such as [Heunis] which helped me understand the concept.

The diagram is conceptually generated by way of a sweep, from top to bottom although it is in practice event based. Before getting onto the details of the algorithm however, a small amount of mathematical background is required. There are multiple ways of describing a parabola, the most common is in the form $y = ax^2 + bx + c$ however there is another way. If there is a point, notated as p_1, and a line, notated as l, a set of all points that have the same distance to p_1 as they do to the closest point on l will form a parabola. If there is a second point, p_2, using the same line l there will be two parabolas that meet at a single point. As these parabolas are designed such that the distance to their respective point is the distance to l, the intersection of these parabolas will be perfectly equidistant between the points. This is the principle Fortune's algorithm is based on.

As the sweep line in our diagram progresses (the line l in our previous example), there are two events that can occur: site events and edge-intersection events.

A site event is where the sweep line encounters a new point to be added to the diagram. When this occurs the section of the beachline directly above the point is going to be divided into two with a new arc (the arc from the new point) appearing in the middle.

An edge-intersection event is where an arc on the beachline gets "squeezed" which is to say that the arcs either side of it expand and meet thus rendering the central arc non existent.
At this event you have three arcs meeting at a common point which thus means that point is equidistant to three points and thus will appear as a three way intersection on the finished diagram. That three way intersection will consist of the two lines that joined together and a new line expanding perpendicular to the arc that got squeezed.

The algorithm itself runs by queuing up all of the events, passing through them one by one and cleaning up the end of the beachline.

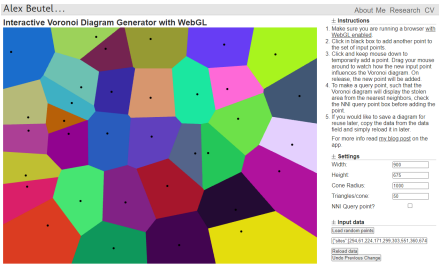
[3]As a quick aside, Steven J. Fortune wins my award for most pain-in-the-ass algorithm to understand. This took me a solid week to get my head around why are all the articles explaining it utter garbage what the actual hell???

Chapter 3

Current Solutions

3.1 Alex Beutel WebGL Program

On his site http://alexbeutel.com/webgl/voronoi.html , Alex Beutel has created a 2D voronoi generator in JavaScript with WebGL that is able to generate 2D voronoi diagrams from seeds added by clicking within the display box.
Alex's site has some features of interest, namely the ability to enter "sites" (seed locations) via a text input box in the UI, although it does have to be be entered with JavaScript Object Notation (JSON) formatting. The data in the text input box is also generated automatically when seeds are added by clicking thus allowing it to also be used as a save feature, with the user able to copy the data out of the box and save it locally.

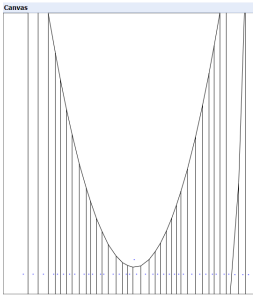


An example diagram created with Alex Beutel's site

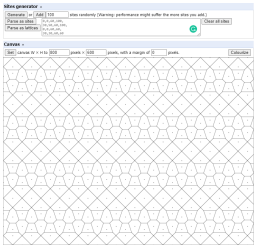
One thing which has bugged me while using his program is the inability to move seeds after they have been placed, especially given you do not know what the diagram will look like until the seed is placed. This is something I will hope to solve in my program.
Annoyingly, nowhere on his site does he state what algorithm he uses for his program, and the blog post he cites "For more info" no longer exists.

3.2 RaymondHill.net JavaScript Program

In a similar vein to Alex Beutel's program, Raymond Hill has created a set of Voronoi generators on his site that allow users to create their own Voronoi diagrams by clicking on the display box. Raymond Hill uses Fortune's algorithm to generate the diagrams and introduces some interesting features such as the ability to "Colourize" the regions when a button is pressed, however the colour that is assigned to each region is random, uncontrollable and removed when an edit is made.
Another feature I am quite a fan of is the ability to easily add lattices with an input box which allows the user to describe a point pattern as a set of 4 numbers (startx, starty, deltax, deltay). The box can accept multiple patterns at once to create remarkably complex patterns and I think if possible some form of template or pattern loading feature could be useful for my program.



(a) A simple parabola I made with Raymond Hill's site



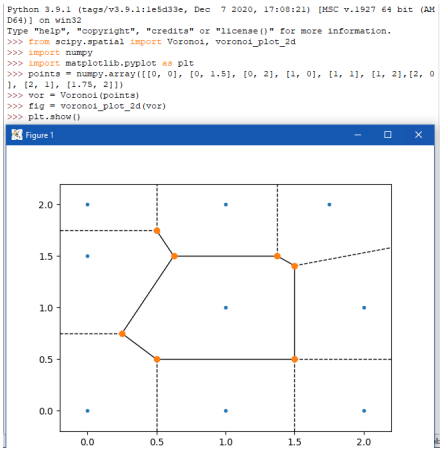
(b) A lattice made with Raymond Hill's generator

3.3 MATLAB

MATLAB offers support for Voronoi generation in two dimensions. However, as a MATLAB subscription costs £760/y and the funding for this project currently consists of 50 pence plus however much I can gain from begging and pleading on my knees in front of my headteacher (£0.00), a full investigation into the capabilities of this software is, for me, currently impossible. I shall attempt to summarise some information below that I have gleaned from examples I have found on the internet:
MATLAB uses a programming interface to allow users to create their own 2D voronoi diagrams by calling the voronoi() subroutine passing arrays of x and y values for seed locations [MATLAB]. As it is MATLAB, it is relatively easy to assign labels to the seed locations if required which is an idea I like and with the voronoin() function you can gain a matrix output of the diagram that allows for relatively straightforward colour-coding of regions [MathWorks]. It does this by using the QHull library (elaborated on below) to compute its outputs. MATLAB suffers from the same problem as Scipy will demonstrate below (in large part due to its dependence on QHull) which is its requirement that its users have proficiency in programming to use it.

3.4 Scipy

Scipy is a module for python which provides a broad array of tools for modelling various scientific problems including optimisation, linear algebra and, crucially for us, Voronoi diagram generation in 2D[Virtanen et al., 2020]. Scipy achieves this by using the Qhull library (elaborated on below) to take inputs in the form of a numpy array and produce a *scipy.spatial.qhull.Voronoi object* which contains the vertices, ridge-points, ridge-vertices, regions and the point-regions for the diagram in question. Although the Voronoi() function does work for n dimensions, its accompanying function voronoi_plot_2d() which plots the Voronoi diagrams with matplotlib only accepts 2D Voronoi diagrams and has no equivalent function for higher order diagrams[Scipy]. As such anyone wanting to use this to create a (N>2) diagram would need to write their own plotting algorithm.



A basic example of a voronoi tessellation with scipy

My takeaways from this are honestly relatively limited, it's ability to take n dimensional points is impressive but not a reflection on this specific program and not explicitly applicable to us as we are working in 2 dimensions. My main takeaway is predominantly that usability is a large priority and requiring the user write their own code to be able to interpret the output is undesirable.

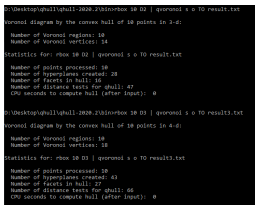
3.5 Qhull

Qhull is a set of packages that perform n-dimensional computations to generate convex hulls, halfspace intersections, Delaunay triangulations and Voronoi diagrams [Qhull, a]. Qhull uses the dual of the Delaunay triangulation to calculate its Voronoi diagrams [Qhull, c][1] and has no theoretical limit on the number of dimensions it can go to although the documentation does not recommend exceeding 7 as "Qhull does not work well with virtual memory" [Qhull, a].

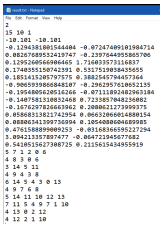
Installing the most basic form of Qhull is trivial, you download the zip file from their downloads page [Qhull, b] and extract it into a directory. From here running the *QHULL-GO* program opens a terminal from which you can operate Qhull.

[1] see addenda "Qhull's Method of Voronoi Computation" pg27.

On the right I have demonstrated the abilities of this form of the program.
I have entered two commands, the first of which is
"rbox 10 D2 | qvoronoi s o TO result.txt"
"rbox 10" generates 10 randomly spaced points
"D2" specifies 2 dimensions
"qvoronoi" specifies that I want it to generate a voronoi diagram
"s" requests a summary to be printed to the console
"o TO result.txt" formats and saves the result to a file called *result.txt* in the bin folder
The second command does the same but in 3 dimensions and saves the result in *result3.txt*.



(a) Inputs into the Qhull console



(b) result1.txt

The contents of the resulting files are... less than immediately useful, they have no intuitive graphical meaning and require a GUI to display them. Qhull recommends using an external program called Geomviewer[Qhull, a]. This continues our trend of programs not providing adequate facilities for actually viewing the output without technical skill. The Qhull program alone is not hugely friendly to use let alone having to install Geomviewer (which requires a Linux installation) to understand what it is producing. Qhull is an amazing piece of software but it falls far too short on usability.

Chapter 4

Potential User

4.1 Nathan Easow

The development of this project shall revolve around input from my primary user in the form of Nathan Easow.
Nathan is a 17 year old biology student and prospective medic who is interested in the applications of Voronoi patterns to simulate bacteria colony growth within boundaries. He is interested in analysing which method is most efficient for his purposes and would then wish to use the program for his own purposes after the analysis has occurred.

4.2 Interview

I did an interview with Nathan to assess priorities for the program, the transcript is below:

James: Okay interview time. Question one, what types of control do you want over the simulation setup?
Nathan: I guess being able to adjust the points after placement... because obviously that's a uh feature that a lot of creators don't allow at the moment.
James: Yeah.
Nathan: And also the ability to alter the shading of the different regions of the result.
James: Yep, that makes sense aight. Okay question 2, do you want any form of bounding box system so a way to restrict where it goes as such, put a limit on how far the Voronoi diagram goes?
Nathan: Yes that would definitely be useful yep.
James: Okay, cool. Question 3, do you need the program to be able to use both Euclidean and Manhattan distance or just Euclidean?
Nathan: Just Euclidean is fine.
James: Cool... 4, are you worried about watching the development of the diagram as an animation?
Nathan: It would be nice, if the method you use allows for it but it's not a key priority.
James: Right, right got ya. Suppose at the end of the day the end result is the important bit... so then yeah I've got what control do you want over the simulation progression but... suppose assuming their is an animation what control do you want over that?
Nathan: Yeah I mean I suppose speed control but that's not necessary that's obviously assuming the animation exists in the first place.
James: Cool. Question 6, Ed Balls?
Nathan: Ed Balls.
James: 7. Would you like to be able to save the diagram conditions?
Nathan: Yes that would definitely be useful yeah
James: and actually continuing on that would you, if you just had the output be able to save a screenshot of that? Do you want that?

Nathan: Yes that would also be great yeah rather than having to load up the program each time yeah.
James: Yeah. Easier to export. 8, how do you feel about having to run a few lines of code in order to run it?
Nathan: I'd prefer to avoid that and have an interface that I can interact with directly.
James: Yeah, not having to mess around with code is probably useful. Loading points from a file, would that be useful?
Nathan: Yeah that would be great rather than having to enter each one manually yeah.
James: Yeah, and finally question 10, what sort of information do you want regarding looking at the various algorithms, looking at how they compare?
Nathan: So I suppose processing time, memory efficiency and how well each one scales.
James: Yeah, yeah I suppose you don't want to be using one that works really badly with large amounts of data.
Nathan: Yeah.
James: Right ok that's all I've got, thank you very much.

4.3 Analysis

The interview with Nathan was able to confirm and clarify some assumptions I had. His emphasis on user experience and an intuitive UI "an interface I can interact with directly" was largely to be expected, as was the requests for features such as point location adjustment and saving. I was surprised by the lack of interest in an animation that shows the development of the diagram although with retrospect that might've been my own personal interest in those animations rather than their actual use. Loading data points from a file is a fairly obvious requirement and *should* be simple enough.

Chapter 5

Goals and Requirements

Based on the interview with Nathan and my analysis of other people's solutions, these are some general and specific goals for the end program.

5.1 General Requirements

- An intuitive user interface requiring no programming skill to interact with
- The ability to generate 2D Euclidean Voronoi diagrams via Delaunay transformations, the Green & Sibson algorithm and Steven J. Fortune's algorithm
- Performance data on the various algorithms

5.2 Specific Goals

CRITICAL
- The ability to load data points from a file
- The ability to alter seed positions after they have been placed by selecting them and either replacing them or altering their coordinates via input boxes
- Ed Balls
- The ability to adjust the RGBA values of the voronoi regions associated with each seed via an input box
- The ability to save the setup of the diagram (x,y positions and RGBA values)
- The ability to save the output of the program as a .png file.
- The UI must be fully graphical with no programming required
- The processing time and memory usage must be displayed when the program is run.

OPTIONAL
- The ability to restrict the maximum and minimum x and y values of the diagram via input boxes
- The ability to view an animation of the diagram developing if the algorithm in question allows it
- If above implemented, the ability to control the speed of the animation
- The ability to add a series of points conforming to a template or pattern by inputting information about the pattern into an input box.

Chapter 6

Testing Algorithms with Python

6.1 Delaunay

6.1.1 Problem Outline

Using the Delaunay triangulation to generate Voronoi diagrams has 2 stages, generation of the Delaunay triangulation, and construction of the Voronoi diagram using the dual of the former.

There is a well established algorithm for constructing Delaunay triangulations which is the Bower-Watson algorithm. This algorithm generates the Delaunay triangulation in $O(n^2)$ time and is rather simple to create once you have a formula for the circumcircle centres.

After the Delaunay triangulation is acquired, finding the dual is relatively trivial. For each triangle in the triangulation you draw a line between all of the circumcentres and then for the triangles on the diagram edge add a line parallel to the perpendicular bisector of the outer edge of the triangle heading out (or multiple if there are multiple outer edges).

6.1.2 Circumcircle Derivation

In order to perform the Bower-Watson algorithm, I am required to be able to generate the circumcircle for any given triangle. Mathematically speaking this is trivial, the centre of the circle is given by the intersection point of two of the perpendicular bisectors of the triangle's sides. The difficulty comes in the fact that this must all be done algebraically as I do not have any of the constants. My derivation of a formula for the (x, y) coordinates of the circumcentre is shown below:

$$let\ ABC = triangle\ where\ A = (x_1, y_1),\ B = (x_2, y_2),\ C = (x_3, y_3)$$

$$let\ (x_{ab}, y_{ab}) = midpoint\ of\ line\ AB$$

$$let\ (x_{ac}, y_{ac}) = midpoint\ of\ line\ AC$$

$$let\ l_1 = bisector\ of\ AB$$

$$let\ l_2 = bisector\ of\ AC$$

$$y - y_n = m(x - x_n)\ where\ m\ is\ gradient\ of\ the\ line$$

$$l_1 = -\frac{x_1 - x_2}{y_1 - y_2}(x - x_{ab}) + y_{ab}$$

$$l_2 = -\frac{x_1 - x_3}{y_1 - y_3}(x - x_{ac}) + y_{ac}$$

$$let\ l_1 = l_2$$

$$-\frac{x_1 - x_2}{y_1 - y_2}(x - x_{ab}) + y_{ab} = -\frac{x_1 - x_3}{y_1 - y_3}(x - x_{ac}) + y_{ac}$$

expand brackets

$$x\left(-\frac{x_1 - x_2}{y_1 - y_2}\right) - x_{ab}\left(-\frac{x_1 - x_2}{y_1 - y_2}\right) + y_{ab} = x\left(-\frac{x_1 - x_3}{y_1 - y_3}\right) - x_{ac}\left(-\frac{x_1 - x_3}{y_1 - y_3}\right) + y_{ac}$$

move x onto one side

$$x\left(-\frac{x_1 - x_2}{y_1 - y_2}\right) - x\left(-\frac{x_1 - x_3}{y_1 - y_3}\right) = x_{ab}\left(-\frac{x_1 - x_2}{y_1 - y_2}\right) - x_{ac}\left(-\frac{x_1 - x_3}{y_1 - y_3}\right) + y_{ac} - y_{ab}$$

factorise out x

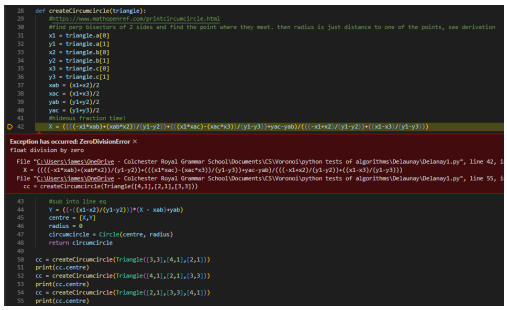
$$x\left(-\frac{x_1 - x_2}{y_1 - y_2} + \frac{x_1 - x_3}{y_1 - y_3}\right) = \frac{-x_1 x_{ab} + x_{ab}x_2}{y_1 - y_2} + \frac{x_1 x_{ac} - x_{ac}x_3}{y_1 - y_3} + y_{ac} - y_{ab}$$

divide to find x

$$x = \frac{\frac{-x_1 x_{ab} + x_{ab}x_2}{y_1 - y_2} + \frac{x_1 x_{ac} - x_{ac}x_3}{y_1 - y_3} + y_{ac} - y_{ab}}{\frac{-x_1 + x_2}{y_1 - y_2} + \frac{x_1 - x_3}{y_1 - y_3}}$$

Substitute value of x back into l_1 or l_2 to get y

Upon implementation the algorithm initially worked but encountered a problem; namely that if points A and B or points A and C had the same y coordinate then it would cause a $ZeroDivisionError$. So I got a refill of tea and a friend of mine pointed out that fortunately, in this scenario, the x coordinate will just be half way between the x coordinates of the two corresponding points. After which the y coordinate can be obtained by substituting this value of x into the equation that does not have both offending points.

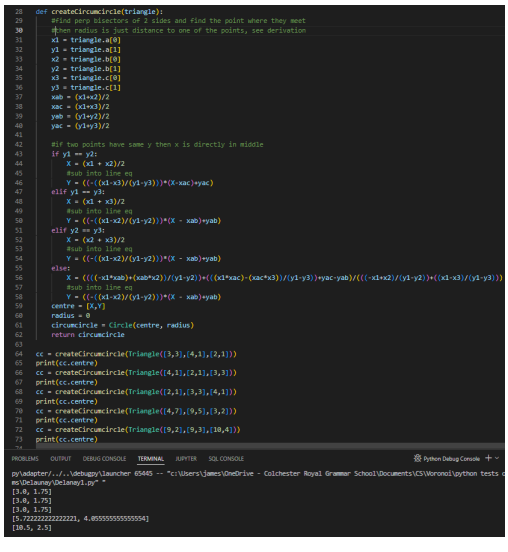


ZeroDivisionError

After this change was made the circumcentres were being generated reliably. All that remained was to calculate the radius which is equal to the distance between the circumcentre and any of the points.

$$r = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

This is using the distance formula that applies Pythagoras to the euclidean coordinates of the points.



working circumcircles code without radii

6.1.3 Creating the Dual

The other great hurdle of this algorithm comes with the transformation between the finished Delaunay triangulation and the Voronoi diagram. The first part is simple, connect the circumcentres of all adjacent triangles. The difficulty comes with the unbounded edges. These are the lines that stretch to theoretical infinity on the outside of the diagram. These lines have a gradient equal to that of the perpendicular bisector of their corresponding line in the Delaunay triangulation and connect to their triangle's circumcentre, the problem is deciding in which direction to extend the line. The obvious answer is "away from the diagram"; but how do you calculate that? You could try

comparing it to the average point location, that doesn't work.
You could try looking at the direction of the circumcentre relative to the original line, nope. Ultimately this became a far greater problem than I had anticipated and my solution to it is rather indicative of that.

As I saw it there were two solutions that I or anyone I asked could think of; the first was to use a form of ray casting algorithm to see if it collided with any other voronoi lines, this seemed overly complicated. Especially given the algorithm for that alone would've been $O(n^2)$.
The other was to treat the supertriangle used in the Bowyer-Watson algorithm as part of the diagram. The supertriangle is a triangle that encompasses all of the other points. By keeping this in the diagram instead of filtering it out at the end of the Delaunay code, it will create the unbounded voronoi lines as part of the regular Delaunay-Voronoi dual flip as described earlier. On the theory that the supertriangle is hard-coded to be so large that it's existence would only be noticeable when you are so zoomed out that you can't see the main section of the diagram. You can even then filter out any lines that go between two points that are far away from the main section to remove all evidence of its existence entirely and simply leave cropped lines that act as our unbounded ones.
This to me sounded rather... janky. And in the future I will try to find a better method of doing this. But for now it works, and works reliably. The "unbounded" lines could be cropped to fit any required reference frame by a program that recalculates their gradients and redraws them within required boundaries, or they could just be left hanging off-screen. Either way I will settle with it.

6.1.4 Full Algorithm

Below is my full python code for calculating the Delaunay triangulation and Voronoi diagram from a set of points, it outputs in Geogebra form for my ease of testing and I have not yet filtered out the outer Voronoi lines that relate to the supertriangle [1][2].

```
import math

class Point():
   def __init__(self, x, y):
      self.x = x
      self.y = y

class Edge():
   def __init__(self, a, b):
      self.a = a
      self.b = b

class Triangle():
   def __init__(self, a, b, c):
      self.a = a
      self.b = b
      self.c = c
      self.edges = [Edge(a,b), Edge(a,c), Edge(b,c)]
      self.circumcircle = self.genCircumcircle()

   def genCircumcircle(self):
      #find perp bisectors of 2 sides and find the point where they meet
      #then radius is just distance to one of the points, see derivation
      x1 = self.a.x
      y1 = self.a.y
      x2 = self.b.x
      y2 = self.b.y
      x3 = self.c.x
      y3 = self.c.y
```

[1] see 6.1.3
[2] Thanks go to Jameson Liu for his invaluable help in bug-fixing and actually making this work

```
        xab = (x1+x2)/2
        xac = (x1+x3)/2
        yab = (y1+y2)/2
        yac = (y1+y3)/2

        #if two points have same y then x is directly in middle
        if y1 == y2:
            X = (x1 + x2) / 2
            #sub into line eq
            Y = ((-((x1-x3)/(y1-y3)))*(X-xac)+yac)
        elif y1 == y3:
            X = (x1 + x3)/2
            #sub into line eq
            Y = ((-((x1-x2)/(y1-y2)))*(X - xab)+yab)
        elif y2 == y3:
            X = (x2 + x3)/2
            #sub into line eq
            Y = ((-((x1-x2)/(y1-y2)))*(X - xab)+yab)
        else:
            X = ((((-x1*xab)+(xab*x2))/(y1-y2))+(((x1*xac)-(xac*x3))/(y1-y3))+
            yac-yab)/(((-x1+x2)/(y1-y2))+((x1-x3)/(y1-y3)))
            #sub into line eq
            Y = ((-((x1-x2)/(y1-y2)))*(X - xab)+yab)
        #Assemble circle object
        centre = Point(X,Y)
        radius = math.sqrt((X-x2)**2 + (Y-y2)**2)
        circumcircle = Circle(centre, radius)
        return circumcircle

class Circle():
    def __init__(self, centre, radius):
        self.centre = centre
        self.radius = radius

def checkEdgeEquality(e1, e2):
    if (e1.a == e2.a and e1.b == e2.b) or (e1.a == e2.b and e1.b == e2.a):
        return True
    return False

def checkIfSharesEdges(edge, currentTri, triangleSet):
    for tri in triangleSet:
        if tri != currentTri:
            for e in tri.edges:
                if checkEdgeEquality(edge, e):
                    return True
    return False

def genDelaunay(points, supertriangle):
    triangulation = []
    triangulation.append(supertriangle)
    #add each point one by one
    for point in points:
        btriangles = []
        #find all triangles that are no longer valid
        for tri in triangulation:
            distToCircumcentre = math.sqrt(((tri.circumcircle.centre.x - point.x)**2) +
                ((tri.circumcircle.centre.y - point.y)**2))
            if tri.circumcircle.radius > distToCircumcentre:
                btriangles.append(tri)
```

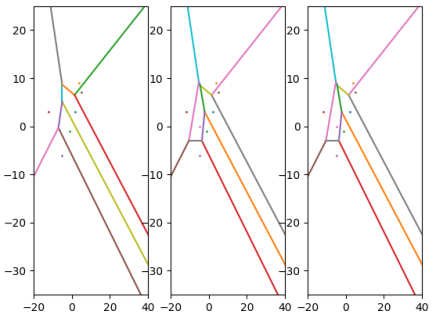
```
      #find boundary of polygonal hole
      polygon = []
      for tri in btriangles:
         for edge in tri.edges:
            if not checkIfSharesEdges(edge, tri, btriangles):
               polygon.append(edge)
      #remove bad triangles from triangulation
      for tri in btriangles:
         triangulation.remove(tri)
      #create new triangles
      for edge in polygon:
         triangulation.append(Triangle(edge.a, edge.b, point))

   return triangulation

supertriangle = Triangle(Point(-100000,-1000000), Point(0,1000000), Point(1000000,-1000000)) #very
    large supertriangle
points = [Point(2,3), Point(4,9), Point(-1,-1), Point(-12, 3), Point(-5,-6), Point(5,7)] #Example
    points
delaunay = genDelaunay(points, supertriangle)


#Convert to Voronoi

voronoiLines = []
for triangle in delaunay:
   for tri in delaunay:
      if triangle != tri:
         if checkEdgeEquality(triangle.edges[0], tri.edges[0]) or
             checkEdgeEquality(triangle.edges[0], tri.edges[1]) or
             checkEdgeEquality(triangle.edges[0], tri.edges[2]) or
             checkEdgeEquality(triangle.edges[1], tri.edges[0]) or
             checkEdgeEquality(triangle.edges[1], tri.edges[1]) or
             checkEdgeEquality(triangle.edges[1], tri.edges[2]) or
             checkEdgeEquality(triangle.edges[2], tri.edges[0]) or
             checkEdgeEquality(triangle.edges[2], tri.edges[1]) or
             checkEdgeEquality(triangle.edges[2], tri.edges[2]):
            voronoiLines.append(Edge(triangle.circumcircle.centre, tri.circumcircle.centre))

#remove duplicates
for line in voronoiLines[:]:
   for l in voronoiLines[:]:
      if line != l:
         if (line.a == l.a and line.b == l.b) or (line.a == l.b and line.b == l.a):
            voronoiLines.remove(line)

print("VORONOI LINES")
for i in voronoiLines:
   print("Segment((", i.a.x, ",", i.a.y, "),(", i.b.x, ",", i.b.y, "))")
```

6.1.5 Incrementability

The Bowyer-Watson algorithm is already built on a theory of incremental construction, i.e. it creates the diagram by adding the points one by one and correcting after each. So to edit the code to edit the Delaunay triangulation would be trivial. By taking the contents of the *for point in points* : loop and putting it in an *addPoint*(*point*) function you could easily allow for the Delaunay triangulation to be updated with new points.

The Voronoi diagram however is more difficult. The obvious response is to simply call the code that calculates the dual of the graph again after you've added all your new points. This would work but is wildly inefficient. Even if you added all the new points to the Delaunay first it is still less than ideal, especially given in the context of my finished program it will only be adding one point at a time anyway.
In order to optimise this incremental Voronoi flip one would need to keep track of all the edges what are changed in the Delaunay triangulation and only adjust those ones. As such I have now remade my python code with this in mind.

Outside of general structure changes (reorganising my classes to be... better), the first major difference I have is the sectioning off of the $addPoint()$ and $addToVoronoi()$ functions so that they can be called whenever required without running the entire generation. After this a small but very important change is the new Edge property called $dualLine$ which upon initialisation is set to $None$. Whenever a Voronoi line is generated from an edge, that edge's $dualLine$ is set to equal the new Voronoi edge. This is so that when updating the Voronoi diagram later it is easy to find and remove the deleted edges from the list of edited triangles.



The matplotlib output of my incremental code. Left is original, middle is updated with full redo, right is updated with partial redo

In the interest of brevity I shall not include the entirety of my new program here but I will include snippets showing the $findChangedTriangles()$ function which gets lists of the triangles that have been added and removed to the new Delaunay triangulation. And the $updateVoronoi()$ function which is rather self explanatory. I have also set up a rudimentary timing system to see if there are any improvements in computation time.
For 6 points incremented to 7 the time taken by redoing the entire Voronoi diagram was 0.00196695s, the time taken by using my partial redo method was 0.00101352s, giving a difference of 0.00095343s.
With 50 points incremented to 55 the time taken by redoing the entire Voronoi diagram was 0.09634828s, the time taken by using my partial redo method was 0.02517295s, giving a difference of 0.07117533s
With 300 points incremented to 320 the time taken by redoing the entire Voronoi diagram was 3.46091080s, the time taken by using my partial redo method was 0.80913591s giving a difference of 2.651774889s.
So my method is faster although practically speaking it does appear to be somewhat minimal outside of extreme cases. A situational success, if you will.

```
def findChangedTriangles(delaunay1, delaunay2):
   deletedTriangles = []
   newTriangles = delaunay2.copy()
   for triangle in delaunay1:
      try:
         newTriangles.remove(triangle)
      except:
         deletedTriangles.append(triangle)

   return newTriangles, deletedTriangles

def updateVoronoi(voronoiLines, delaunay, newTriangles, deletedTriangles):
   #add the new stuff
   for triangle in newTriangles:
      voronoiLines = addToVoronoi(triangle, delaunay, voronoiLines)
   #delete the old stuff
   for triangle in deletedTriangles:
      if triangle.edges[0].dualLine in voronoiLines:
         voronoiLines.remove(triangle.edges[0].dualLine)
      if triangle.edges[1].dualLine in voronoiLines:
         voronoiLines.remove(triangle.edges[1].dualLine)
```

```
      if triangle.edges[2].dualLine in voronoiLines:
         voronoiLines.remove(triangle.edges[2].dualLine)
   #remove duplicate lines
   for line in voronoiLines[:]:
      for l in voronoiLines[:]:
         if line != l:
            if (line.a == l.a and line.b == l.b) or (line.a == l.b and line.b == l.a):
               voronoiLines.remove(line)
   return voronoiLines
```

6.1.6 Potential for animation

The problem with animating the Voronoi generation via the Delaunay triangulation is that the algorithm being used is one which does the entire diagram at once. If I wanted some form of expanding circles algorithm I would need to only calculate the triangulations of points that have grown to the extent that they make a connection and then I would have to shorten the Voronoi lines that come out of it to only extend to the width of the circle at that section. And I would then need to draw the circle with the sections missing around the point. Long story short it's not really feasible with this algorithm.

6.2 Green and Sibson

6.2.1 Problem Outline

6.2.2 Putting points in clockwise order

For the Green and Sibson algorithm to work intentionally, the points list needs to be put in clockwise order; I have achieved this with polar coordinates.
I start by finding the centre of the points by taking the average of the x and y coordinates.

```
sumX = 0
sumY = 0
for point in points:
    sumX += point.x
    sumY += point.y
centre = Point((sumX / len(points)), (sumY / len(points)))
```

From there it's a matter of finding the angle in radians relative to north. I have done this with a series of if elifs to cover all 4 regions around the centre that then use the arcsine of the absolute relative y distance divided by the distance to find the angle.

```
for point in points:
r = math.sqrt(((centre.x - point.x)**2) + ((centre.y - point.y)**2))
relx = point.x - centre.x
rely = point.y - centre.y
if relx > 0 and rely > 0:
   angle = (math.pi / 2) - math.asin(rely/r)
elif relx < 0 and rely < 0:
   angle = ((math.pi * 3) / 2) - math.asin(abs(rely)/r)
elif relx < 0 and rely > 0:
   angle = ((math.pi * 3) / 2) + math.asin(rely/r)
elif relx > 0 and rely < 0:
   angle = (math.pi / 2) + math.asin(abs(rely)/r)
elif relx == 0:
   if rely > 0:
      angle = 0
```

```
   else:
      angle = math.pi
#storing polar coords as [rad, angle]
point.polarCoords = [r, angle]
```

Finally the points are ordered by their angle with a simple bubble sort algorithm.

```
n = len(points)
swapped = True
while swapped:
   swapped = False
   for i in range(n - 1):
      if points[i].polarCoords[1] > points[i + 1].polarCoords[1]:
         swapped = True
         points[i], points[i + 1] = points[i + 1], points[i]
   n -= 1
```

6.2.3 A little maths

There are two small bits of maths required to program the Green and Sibson algorithm. They are much simpler than the circumcentre equation for the Delaunay triangulation but it is worth noting them.

First is finding an equation for the perpendicular bisector of two points:
The gradient for this bisector is equal to the negative reciprocal of the gradient of the line that goes between the two points.

$$A = (x_1, y_1)$$

$$B = (x_2, y_2)$$

$$gradient\ AB = \frac{y_1 - y_2}{x_1 - x_2}$$

$$gradientBisector = -\frac{1}{gradient\ AB}$$

$$= \frac{-x_1 + x_2}{y_1 - y_2}$$

This line will pass through a point with coordinates equal to the average of the coordinates of A and B

$$Centrepoint = C = (x_3, y_3)$$

$$x_3 = \frac{x_1 + x_2}{2}$$

$$y_3 = \frac{y_1 + y_2}{2}$$

Substitute into line equation to get equation of bisector

$$y - y_3 = m(x - x_3)$$

$$y = mx - mx_3 + y_3$$

$$where\ m = \frac{-x_1 + x_2}{y_1 - y_2}$$

The other is finding the coordinates of a point that is mirrored across a line $y = mx + c$.

$$given\ equation\ y = m_1 x + c$$

$$let\ A = original\ point\ location = (x_1, x_2)$$

$$gradient\ of\ perpendicular\ line = m_2 = \frac{-1}{m_1}$$

find c value for perpendicular line by rearranging line formula

$$y - y_1 = m_2(x - x_1)$$

$$y = m_2 x - m_2 x_1 + y_1$$

$$-m_2 x_1 + y_1 = y - m_2 x$$

$$thus\ c = -m_2 x_1 + y_1$$

The midpoint between this point and the reflected one will be the intersection between the original line and the perpendicular line derived above. We shall refer to this point as (x_2, y_2) This midpoint will be half way between the two points and is thus their average, reversing this average will give the x,y coordinates of the reflected point (x_3, y_3)

$$x_2 = \frac{x_1 + x_3}{2}$$

$$x_3 = 2x_2 - x_1$$

$$y_2 = \frac{y_1 + y_3}{2}$$

$$y_3 = 2y_2 - y_1$$

6.2.4 Handling Edge Cases

The useful part of the Green and Sibson algorithm is that it is designed with edge cases in mind. That is, the paper which describes how the algorithm works [Green and Sibson, 1977] has a large explanation into "windows" and how the algorithm interacts with boundary lines.

Essentially, it handles them by pretending they are like any other voronoi line with a corresponding seed. When the algorithm interacts with them it gives them a "virtual point" such that the bisector of the virtual point and the currently being added seed is the boundary line. Despite the fact that I am evidently terrible at explaining it, this solves all problems surrounding boundaries with the exception of scenarios where there aren't any. In which case I will just assign some arbitrarily large ones.

6.2.5 Full Algorithm

6.2.6 Incrementability

6.2.7 Potential for animation

6.3 Fortune's Algorithm

6.3.1 Problem Outline

6.3.2 Full Algorithm

6.3.3 A stupid amount of maths

Fortune's algorithm requires some rather irritating bits of maths, the first of which being the requirement to find the centre of a circle which passes through two points and is tangent to a line $y = c$. I shall outline my method below which was acquired from animation uploaded to GeoGebra by Irina Boyadzhiev[Boyadzhiev, 2016]. It uses a multi-step derivation that I shall outline below, the specifics of why the construction works are immaterial and thus shall not be elaborated upon here, it merely matters that the steps are completed correctly.

6.3.4 Incrementability

6.3.5 Potential for animation

Chapter 7

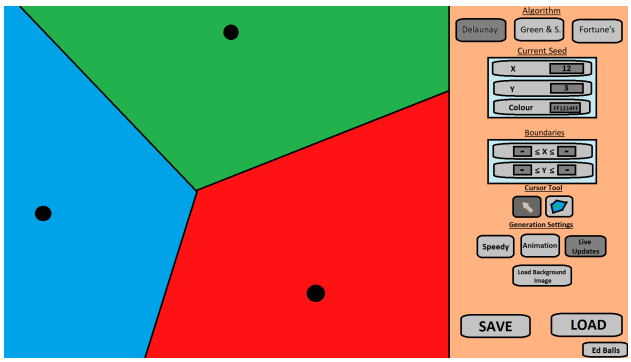
Designing an Interface

7.1 Interface requirements

The interface requirements in large part draw from the specific objectives in section 5.2. From which I can glean that my interface needs to be able to support:
- Loading data points
- Altering seed positions
- Saving conditions and images
- Performance displayed
- Bounding boxes
- Animations
- Animation speed control
- Lattices/pattern generation/templates

7.2 Version 1

7.2.1 Overview



On the right is my first interface design that I've put together in digital art station software.
It consists of a main voronoi viewing section with a right hand side panel that contains the controls.

From top to bottom:
The Algorithm buttons (Delaunay, Green & S., Fortune's) refer to what algorithm is going to be used to generate the diagrams. These are mutually exclusive and will disable others when picked.
The current seed input box contains input boxes for x, y coordinates and colour for whatever seed is selected (currently that is the bottom right red seed).
The boundaries input boxes allow the user to input maximum and minimum values for which the diagram is displayed with regards to the x and y axis. The boxes currently include a hyphen to indicate no limits.
The cursor tool buttons are another set of mutually exclusive buttons that choose between the cursor clicks selecting which seed is selected and adding a new seed.
The generation settings buttons are mutually exclusive and dictate how the diagram will be generated. Whether it will be generated as fast as possible, with an accompanying animation, or in a form where you

can add to the diagram after generation.
Load Background Image will open a window that will let the user select an image to put behind the voronoi diagram being displayed.

7.2.2 Initial Thoughts

As a first draft I think this layout has some merits; the organisation is relatively neat with all the buttons and inputs being tucked away to the side and grouped by category. However, it does feel slightly cluttered.
I also have a problem with my colour inputs. The value represented in the image is $FF1214FF$, this uses an 8-digit RGBA colour code to represent the colour and transparency of the tile. Personally I think having a separate transparency slider would likely be better although I shall confer with my user on this.
I also think after looking at this it would likely be beneficial to have some form of help button if my documentation is insufficient (and it is also likely true that most users would prefer a simple help menu to a clunky documentation file).
Finally, I am not currently fulfilling the requirement that I display performance data, I have no template loading system and for the non-incremental running modes there is currently no "Run" button. This will obviously require remedying.

7.2.3 User's Thoughts

7.2.4 Conclusions

7.3 Version 2

7.3.1 Overview

7.3.2 Initial Thoughts

7.3.3 User's Thoughts

7.3.4 Conclusions

Acknowledgements

Miss Baker	For teaching me my proficiency in C#
Nathan Easow	Agreeing to be my user
My friends who are forced to listen to me	For putting up with my excited rambling and proof reading some of my grammar
Lydia Gee	For suggesting a risk assessment
The unofficial school cat	Mental health support
The Flying Spaghetti Monster	All are welcome into the loving embrace of His Noodly Appendage
Newton's first law	Great support
Entropy	Providing an unassailable deadline
Connor Fox Cunningham	Pointing out that my acknowledgements are just another form of procrastination
The "free science lessons" guy	Acting as a true role model
Sam the snake plant	Making the air in my room slightly less toxic

Addenda

Clarifications

Qhull's Method of Voronoi Computation

I stated in 3.5 that Qhull computes Voronoi diagrams via the dual of the Delaunay triangulation, and this is what is stated on their documentation page for the function. However, when the function is run for a 2D diagram it displays the line "Voronoi diagram by the convex hull of 10 points in 3-d). This would refer to some variation on Brown's algorithm which I did not cover here. I did not comment on this discrepancy in the main section as I did not consider it relevant however I would like to discuss it here.

Qhull is, at its core, not a Voronoi program. It is designed to generate what are known as convex hulls which are the smallest shape for a set of points which contains the entire linear line between any two points within it. As such I believe the creators are simply generating Voronoi diagrams based on what they already have. In their paper it occurs to me that their main priorities are in generating convex hulls and Delaunay triangulations from said hulls. It does not specify how it makes the Voronoi diagrams, and thus it could be that Qhull uses the convex hulls directly (such as with Brown's algorithm) or what I believe is more likely is that Qhull is using convex hulls to generate Delaunay triangulations and is then using the dual of those to generate the Voronoi diagrams. That theory agrees with both statements that it is generated from convex hulls and from Delaunay. I only wish that if this is the case then Qhull would just say so.

Bibliography

Douglas Adams. *The Restaurant at the End of the Universe*. 1980.

Martin Bock, Amit Kumar Tyagi, Wolfgang Alt, and Jan-Ulrich Kreft. Generalized voronoi tessellation as a model of two-dimensional cell tissue dynamics. 2009. URL https://arxiv.org/pdf/0901.4469.pdf.

Irina Boyadzhiev. Construction of a circle through two points and tangent to a given line, 2016.

M.N. Drozdovskaya. Voronoi diagrams and their application in the dtfe reconstructions of the cosmic web, 2010. URL https://www.universiteitleiden.nl/binaries/content/assets/science/mi/scripties/drozdovskayabach.pdf.

Jiunn-Der (Geoffrey) Duh. W6_delaunaytriang.pdf. URL http://web.pdx.edu/~jduh/courses/geog493f09/Students/W6_DelaunayTriang.pdf.

Balu Ertl. 20 points and their voronoi cells, 2015. URL https://commons.wikimedia.org/wiki/File:Euclidean_Voronoi_diagram.svg.

Nü es. A delaunay triangulation with circumcircles, based on delaunay_triangulation_small.png. URL https://commons.wikimedia.org/wiki/File:Delaunay_circumcircles.png.

P. J. Green and R. Sibson. Computing dirichlet tessellations in the plane. 1977. URL https://academic.oup.com/comjnl/article-pdf/21/2/168/1291172/21-2-168.pdf.

Jacques Heunis. Fortunes algorithm: An intuitive explanation. URL https://jacquesheunis.com/post/fortunes-algorithm/.

Hui Li, Kang Li, Taehyong Kim, Aidong Zhang, and Murali Ramanathan. Spatial modeling of bone microarchitecture. 2012. URL https://cse.buffalo.edu/DBGROUP/bioinformatics/papers/Hui_SPIE2012.pdf.

loc.gov. Physical astronomy for the mechanistic universe. URL https://www.loc.gov/collections/finding-our-place-in-the-cosmos-with-carl-sagan/articles-and-essays/modeling-the-cosmos/physical-astronomy-for-the-mechanistic-universe#:~:text=To%20explain%20motion%20Descartes%20introduced,the%20stars%20were%20made%20of.

MathWorks. Matlab function reference voronoi. URL http://matlab.izmiran.ru/help/techdoc/ref/voronoi.html.

MATLAB. voronoi. URL https://uk.mathworks.com/help/matlab/ref/voronoi.html.

F. Mercier and O. Baujard. Voronoi diagrams to model forest dynamics in french guiana. 1997. URL http://www.geocomputation.org/1997/papers/mercier.pdf.

Atsuyuki Okabe, Barry Boots, Kokichi Sugihara, and Sung Nok Chiu. *Spatial Tessellations Concepts and Applications of Voronoi Diagrams*. 2 edition, 2009.

Sally Le Page. Science stand-up comedy that's completely bananas, 2019. URL https://www.youtube.com/watch?v=HQL3MkgvOug.

Qhull. Qhull manual, a. URL http://www.qhull.org/html/index.htm.

Qhull. Qhull downloads, b. URL http://www.qhull.org/download/.

Qhull. qvoronoi – voronoi diagram, c. URL http://www.qhull.org/html/qvoronoi.htm.

Scipy. scipy.spatial.voronoi. URL https://docs.scipy.org/doc/scipy/reference/generated/scipy.spatial.Voronoi.html#scipy.spatial.Voronoi.

John Snow. John snow's map. URL https://johnsnow.matrix.msu.edu/book_images12.php.

P. Virtanen, R. Gommers, and T.E. Oliphant et al. Scipy 1.0: fundamental algorithms for scientific computing in python, 2020.

Georgy Voronoi. Nouvelles applications des paramètres continus à la théorie des formes quadratiques. 1908. URL https://gdz.sub.uni-goettingen.de/download/pdf/PPN243919689_0133/LOG_0010.pdf.

Eric W. Weisstein. Voronoi diagram. URL https://mathworld.wolfram.com/VoronoiDiagram.html#:~:text=Voronoi%20diagrams%20were%20considered%20as,Voronoi%20diagrams%20to%20higher%20dimensions.