Parabola Volume 60, Issue 3 (2024)

She sails dual trails by the sea shore: approximating the area of a convex polygon through outside observations

Benjamin Hogan[1]

1 Introduction

You are a surveyor trying to measure the dimensions of a small, convex island in the ocean from their ship. Besides the trivial solution of directly measuring the island's dimensions by sailing the perimeter, how could one approximate the island's area with just angle measurements and a log of the ship's movement?

2 Related and original research

The surveyor could try applying trigonometry to angle measurements taken with respect to some landmarks. Two immediately come to mind: the leftmost and rightmost edges of the island in the ship's field of view.

Let's try tracking the leftmost and rightmost corners using rays extended from the ship, labelled S, to those left and rightmost vertices. These are the left and right "field of view rays" (abbreviated as "FOV rays"). They are labelled "l" and "r" in Figure 1.[2]

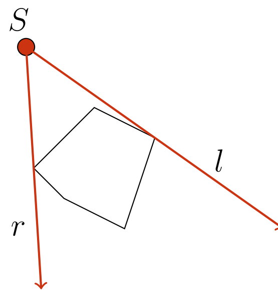

Figure 1: Example island

As the ship moves, different parts of the island reveal themselves. Notice in Figure 2 that the FOV rays will always intersect with two different vertices of the island such that the minor angle formed by them at S is greatest.

[1]Benjamin Hogan is a recent graduate of Herricks High School, and an incoming freshman at Stony Brook University.

[2]In order to improve accessibility, diagrams in this paper use both texturing and Paul Tol's "Vibrant" colour scheme, a colour-blind friendly palette [1].

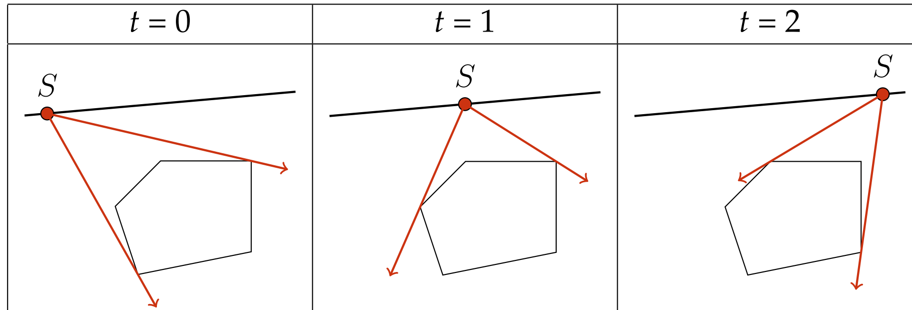

Figure 2: A ship moving

When a field of view ray intersects a vertex, that vertex is considered “visible”, meaning that the surveyor can measure the angle of the field of view ray intersecting it with respect to the ship’s orientation.

2.1 A coordinated approach

Because we know nothing about the island except the fact that it is a convex polygon, our method must not rely on geometry exclusive to any specific number of sides. Instead, we will use coordinates. All polygons are uniquely defined by a set of coordinates, so if the polygon’s coordinates are known, and we have a function to convert coordinates to area, then we’ll have solved a large chunk of the problem already.

2.1.1 The Shoelace Formula

The Shoelace Formula comes in many forms, but they all essentially do the same thing: convert an ordered set of coordinates that define a polygon into the area that polygon encloses.

One derivation of the Shoelace Formula comes from summing the signed areas of trapezoids below each edge of the polygon, which is shown in Figure 3.

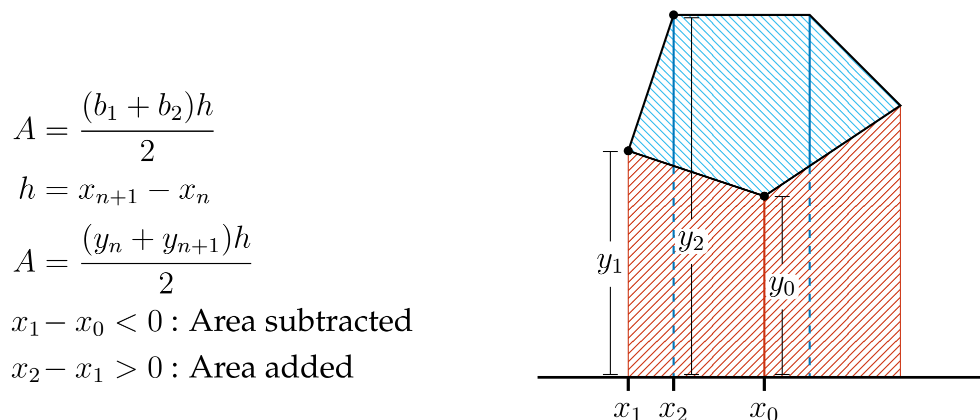

Figure 3: Leadup to the Shoelace Formula

Those familiar with the trapezoidal rule may recognize this equation for the area under a segment. The area of a trapezoid with height h, and bases b_1 and b_2 is [2]:

$$A = \frac{(b_1 + b_2)\,h}{2}.$$

If we have a list of coordinates ordered either clockwise or counter-clockwise, then we can replace h with $x_{n+1} - x_n$, and b_1 and b_2 with y_n and y_{n+1}:

$$A = \frac{(y_n + y_{n+1})\,h}{2}.$$

Notice that the sign of h changes depending on whether x_n is greater than or less than x_{n+1}. Summing all components and then taking its absolute value[3] will return the polygon's area. This can be rearranged and expressed like this [3, pp. 234–236]:

$$\left|\sum_{k=1}^{n-1}(x_{n+1} - x_n)\frac{y_n + y_{n+1}}{2}\right| = \frac{1}{2}\left|\sum_{k=1}^{n-1}(x_n y_{n+1} - x_{n+1} y_n)\right|.$$

If we're going to find a list of the island's coordinates to input into the Shoelace Formula, then we'll need to come up with a way to order the pairs. This is a form of the Convex Hull Problem [4], a problems in computational geometry where one tries to find the smallest convex polygon that "wraps" around all points in a point cloud.

Fortunately for us, the convex hull of a convex polygon is the polygon itself, so constructing our convex hull is simpler than it is for a general point cloud.

The Graham Scan Algorithm [5] is an efficient algorithm that can be modified for our purposes. As a brief overview, the algorithm has three phases:

(a) Find one point with the lowest y-coordinate, and use the lowest x-coordinate as a tie-breaker. This point is guaranteed to be part of the convex hull.

(b) Sort the remaining points in order of increasing angles that the line connecting them to the point from step 1 makes with the x-axis.

(c) Connect the points in order, checking whether connecting them makes a clockwise or counter-clockwise turn. If you make a counter-clockwise turn, then keep going. If you make a clockwise turn, then that means the point you are on is within the convex hull, so delete the current point from your search, go back a step, and repeat.

Because every point of our polygon is part of the convex hull, deleting points in step three is unnecessary. We need only sort our inputs by the angle they make with the x-axis, resulting in a properly ordered list that we can plug into the Shoelace Formula.

[3]Taking the absolute value is necessary, as the summed area may end up being negative depending on the polygon being measured.

2.1.2 Where to find the coordinates

If we want to identify every coordinate of a polygon, then we must find where to look. Let's play around with the ship's position and map out where certain pairs of vertices are "visible", by colour-coding sections of the plane to match vertex pairs. Figure 4 shows one such map.

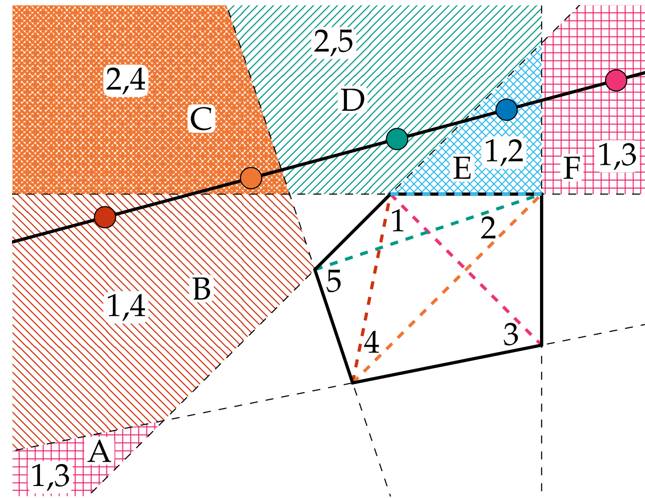

Figure 4: Regions formed by visibility lines

Notice that regions are clearly divided by the extended sides of the polygon. We will refer to these extended sides as "visibility lines".

Based on this diagram, we can make four observations about moving the ship around convex polygons:

(a) **As the ship crosses a visibility line, one vertex exits the field of view, obscured by the rest of the island, and another vertex enters the field of view.**

As the ship moves from region C to region D, vertex 4 exits the field of view, and vertex 5 enters it.

(b) **When crossing a visibility line, the vertex entering the field of view is *not* the same vertex exiting the field of view.**

This will be helpful for determining the shape of our path later.

(c) **If the ship does not cross a visibility line, then the vertices that are visible remain the same.**

All locations of the ship in section B, for example, have vertices 1 and 4 visible.

(d) **When the ship lies on a visibility line, the vertices line up in a way such that both intersect a field of view ray at the same time.**

For example, at the border of regions E and F, pairs 1,2 and 1,3 are both visible. They're indistinguishable along the visibility line, so you can treat FOV rays like they intersect both vertex 2 and vertex 3, whichever is more convenient.

2.1.3 Defining our path

There are two components of this problem: The ship's path must cross enough regions to make every vertex visible at least once, and along that path we need to calculate the coordinates of the vertices visible in that region.

Now that we know where the ship needs to be to isolate pairs of vertices, we must find a path that is guaranteed to see all vertices at least once.

A simple line, like the one from Figure 4, seems to work. Table 1 shows that neither regions E nor F provide any new information, so in some cases, a line actually visits more than enough regions.

The example in Figure 5 and Table 2, however, shows that this isn't always true. A line won't always visit more regions than necessary. Could there be cases where a linear path won't see all vertices?

Region	Vertex visible?					Unique vertices found
	1	2	3	4	5	
A	✓		✓			+2
B	✓			✓		+1
C		✓		✓		+1
D		✓			✓	+1
E	✓	✓				+0
F	✓		✓			+0

Table 1: Tracking the path in Figure 4

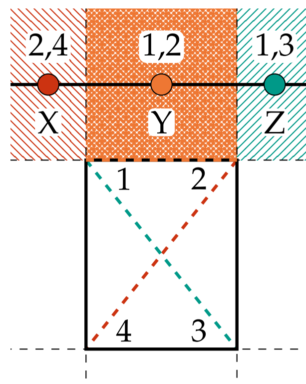

Figure 5: A linear path around a rectangle

Notice that the only visibility lines not crossed by a straight path are, by definition, those parallel to it. Because the island must be a convex polygon, at most two sides can be parallel to the ship's path, which is what we see in Figure 5.

This means that for an n-sided polygon, a straight line will cross $n-2$ visibility lines at minimum. From observation (a), each crossed line reveals one new vertex, and from (b), we know that that vertex is unique, so crossing those $n-2$ visibility lines reveals

Region	Vertex visible?				Unique vertices found
	1	2	3	4	
X		✓		✓	+2
Y	✓	✓			+1
Z	✓		✓		+1

Table 2: Tracking the path in Figure 5

$n-2$ unique vertices. Finally, no matter where in the ocean the ship starts, 2 unique vertices must be visible from the start. Putting all of this together, we find that any line not intersecting an n-sided polygon must visit all $(n-2)+2=n$ vertices.

This is probably not the most efficient path,[4] but it works for our purposes, so we will develop our algorithm with the assumption that the ship can only travel along a line.

2.2 Finding individual vertices within a region

The second half of this problem involves finding the coordinates of vertices within each region along our path.

2.2.1 Simplification

Let's simplify our view of the situation. As long as the ship remains within a specific sector bounded by visibility lines, we can reduce the problem to two vertices and ignore the rest of the polygon, as shown in Figure 6. These situations are indistinguishable as long as the ship never exits the region it's in.

In other words, as long as the ship never crosses a visibility line, we can temporarily treat the island as if it were a single line segment.

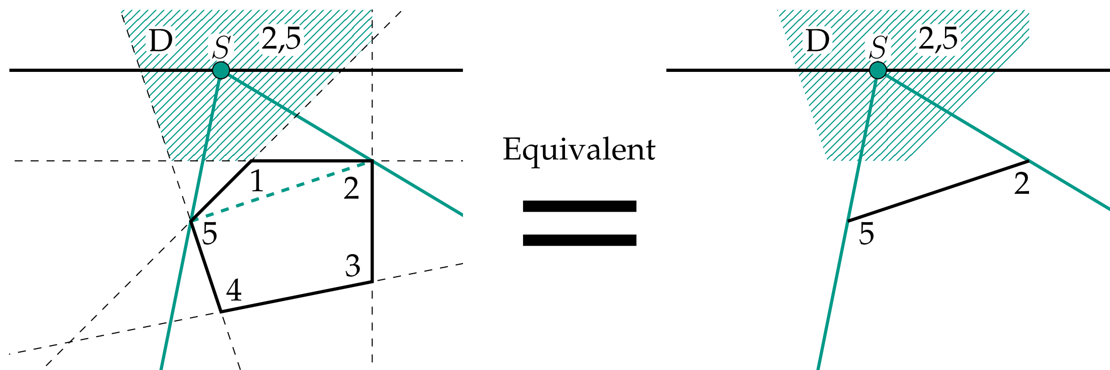

Figure 6: The left and the right are indistinguishable within region D

Let A and B represent the vertices of the simplified island. There isn't much we can say about A or B besides the angle of the field of view rays intersecting them.

[4] As discussed in Section 3.2

One set of these angles is not sufficient to uniquely identify $\overline{AB}$, as seen in Figure 7, there are infinite pairs of vertices that may "look" exactly the same from the ship's perspective—all of which form the same angle between the field of view rays.

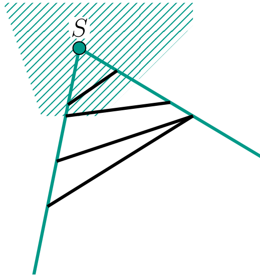

Figure 7: One perspective

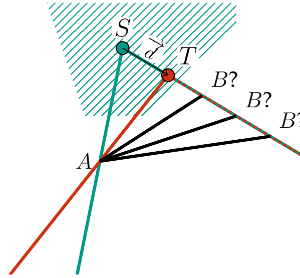

Figure 8:
A collinear displacement

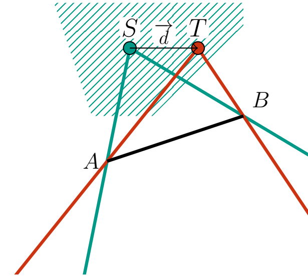

Figure 9: Two perspectives

What if S undergoes a displacement? As long as the displacement, $\vec{d}$, isn't collinear with either of the ship's field of view rays as in Figure 8, the new perspective narrows down the possibilities for $\overline{AB}$ to a single unique line segment defined by the intersections of field of view rays as in Figure 9.

A convenient choice for our displacement would be along the linear path that we found earlier. The benefits of this choice are twofold: It's simple and requires no deviation from the path we already found, and the ship's path is guaranteed to never intersect with the island,[5] so it could never be collinear with a field of view line.

2.3 How to find coordinates

Seeing as only two perspectives are needed to uniquely identify $\overline{AB}$ within a region, why not derive an equation that converts angle measurements to vertex coordinates? We want a black box that takes measurements from the ship and spits out the island's vertex coordinates.

First we'll use the law of sines to solve for the distance between the ship and A or B, then displace that by the ship's initial position, S, to get their true coordinates.

Figures 10 and 11 are derived from Figure 9 and only show the FOV rays necessary from each perspective to find A and B.

2.3.1 Finding the distance

First we'll solve for A using Figure 10. Begin by applying the law of sines [6]:

$$\frac{\sin\left(180 - \angle STA - \angle TSA\right)}{|\vec{d}\,|} = \frac{\sin\left(\angle STA\right)}{|\overrightarrow{SA}|},$$

[5] If it did, then you'd have bigger problems.

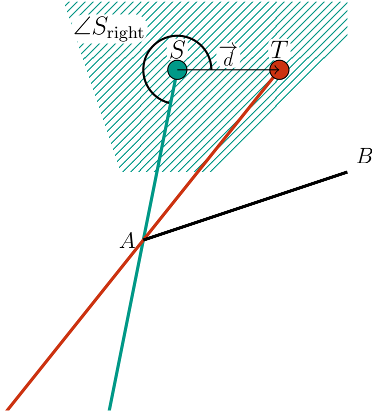

Figure 10: Right FOV rays

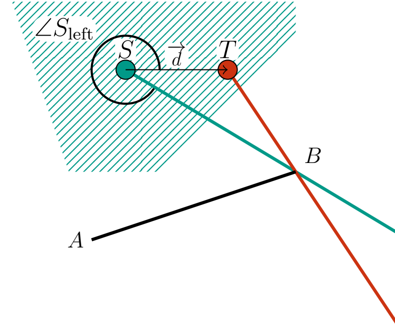

Figure 11: Left FOV rays

which can be rearranged for $|\overrightarrow{SA}|$:

$$|\overrightarrow{SA}| = \frac{|\vec{d}\,|\sin(\angle STA)}{\sin(180 - \angle STA - \angle TSA)}.$$

The process for finding $|\overrightarrow{SB}|$ is identical, the only difference being that $\angle STA$ and $\angle TSA$ are replaced by $\angle STB$ and $\angle TSB$ from Figure 11:

$$|\overrightarrow{SB}| = \frac{|\vec{d}\,|\sin(\angle STB)}{\sin(180 - \angle STB - \angle TSB)}.$$

2.3.2 Finding the coordinates

If one adds $\overrightarrow{SA}$ to S, then the result is the location of A:

$$A = \left(S_x + |\overrightarrow{SA}|\cos(\angle S_{\text{right}}), S_y + |\overrightarrow{SA}|\sin(\angle S_{\text{right}})\right).$$

Again, the process is identical for finding B, except for replacing $\overrightarrow{SA}$ by $\overrightarrow{SB}$, and $\angle S_{\text{right}}$ by $\angle S_{\text{left}}$:

$$B = \left(S_x + |\overrightarrow{SB}|\cos(\angle S_{\text{left}}), S_y + |\overrightarrow{SB}|\sin(\angle S_{\text{left}})\right).$$

2.3.3 A minor problem with this approach

It’s wrong.

2.4 How to find coordinates (but not the wrong ones)

Figure 12 shows an example application of our two-perspective method.

$$|\overrightarrow{SB}| = \frac{|\overrightarrow{d}|\sin(\angle STB)}{\sin(180 - \angle STB - \angle TSB)}$$

$$\angle S_{\text{left}} = 319.4°$$

$$B = \left(S_x + |\overrightarrow{SB}|\cos(\angle S_{\text{left}}), S_y + |\overrightarrow{SB}|\sin(\angle S_{\text{left}})\right)$$

Predicted B coordinates = (7.462... , -6.395...)

Actual B coordinates = (-7,-6) ?!

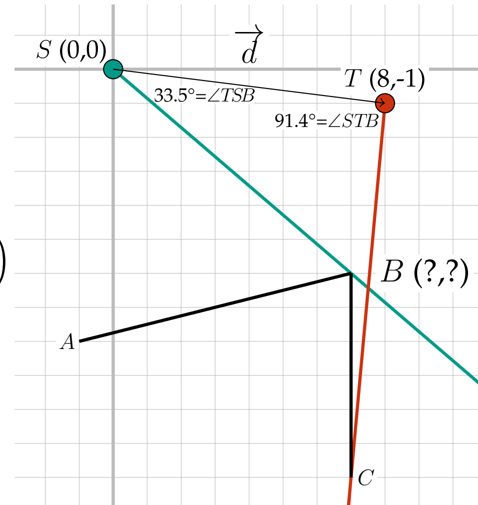

Figure 12: Two-perspective method results in a "phantom point"

Our predicted coordinates are completely off! What went wrong? To expand on the point outlined in Section 2.3.3, using two vertices to find coordinates will give you the wrong answer *whenever the line connecting the two perspectives crosses a visibility line.*

Remember when we reduced the polygon to a single line segment in Figure 6? Any equalities derived for that model only hold when both of the ship's perspectives lie in the same region.

When we apply the equation we derived in the previous section to Figure 12, the angle $\angle STB$ isn't what the boat observes at position T, it's actually $\angle STC$! Our method loses the geometric meaning we used to derive it, so instead of returning B, it returns the intersection of the FOV rays.[6]

We don't want to include these "phantom points" in our list of vertices because they are not part of the island, but how do we avoid them?

2.5 A third perspective

When we have only two perspectives, there's no way to tell when a displacement crosses a line. Any set of two nonparallel lines will intersect somewhere, but there is no guarantee that that "somewhere" is actually on a vertex. This isn't the case for three perspectives. A set of three field of view lines will only intersect on the same point if a visibility line isn't crossed. A third perspective lets us check our work: if at least one FOV ray doesn't intersect at the same point as the other two, then the ship must have crossed a visibility line when travelling to one of the perspectives.

[6] It's worth noting that the method *always* returns the coordinates of the point where the FOV rays intersect, it just so happens that they only intersect at one of the island's vertices when the displacement, $\vec{d}$, doesn't cross a visibility line.

2.5.1 Justification

Let there be two positions of the ship where observations are taken: M and N. If both positions are in the same region, then great! The equality we derived in Section 2.3 solves for that intersection's coordinates, which will lie on a vertex.

What if the positions aren't in the same region? The equality still solves for where the FOV rays intersect, but that intersection won't be on a vertex. There is no indication that the vertex we found is wrong, so we might be stuck with an inaccurate map if we add the result to our list of vertices!

When we add a third position, P, along the line formed by extending segment $\overline{MN}$, we can derive equalities that only hold when M, N, and P intersect at the same point, which is true if and only if the line connecting the perspectives doesn't cross a visibility line. We can prove this with a truth table. Let A represent the proposition "The path does *not* cross a visibility line", and B represent the proposition "The three FOV rays intersect at a single point". We want to prove that $A \iff B$.

A	B	$(\neg A \implies \neg B) \wedge (A \implies B)$	$A \iff B$
T	T	T	T
T	F	F	F
F	T	F	F
F	F	T	T

Table 3: Truth table utilizing observations (b) and (c)

Observation (b) states that $\neg A \implies \neg B$: when the path *does* cross a visibility line, at least one FOV ray sees a different vertex. This means that not all three FOV rays intersect at a single point.

Observation (c) states that $A \implies B$: When the path doesn't cross a visibility line, all perspectives see the same vertices, and therefore intersect at a single point.

Combining these conditions in Table 3 yields a truth table that shows that $A \iff B$ is logically equivalent to $(\neg A \implies \neg B) \wedge (A \implies B)$, as required for developing our formulas in the next section.

2.5.2 Crafting the equalities

If all three field of view rays intersect at the same point, then they must form triangles like $\triangle MAN$, $\triangle MAP$ and $\triangle NAP$ as shown in Figure 14. These specific triangles are formed by the right-side rays, and are useful for finding A, but we can easily mirror all calculations to find B through left-side rays.

If we take all three possible pairs of sides and use the law of sines to solve for the distance between the perspectives, then we will get two equations for each of $\overline{MA}$, $\overline{NA}$ and $\overline{PA}$:

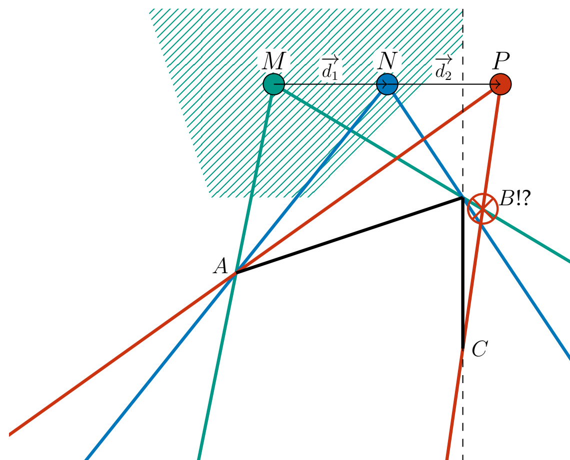

Figure 13: If all three don't agree, then a visibility line must have been crossed

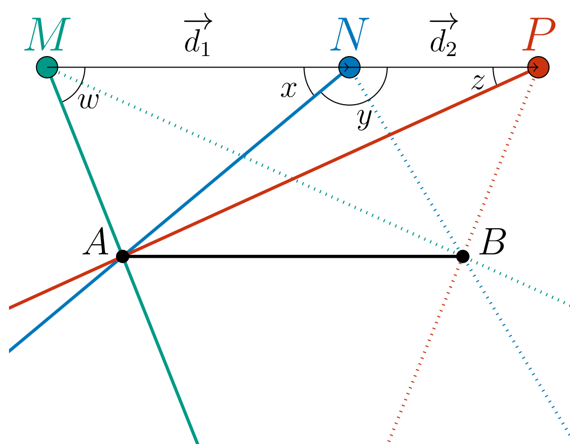

Figure 14: Three-perspective method with right-side FOV rays emphasized

$\triangle MAN$	
Equation 1: $MA = \dfrac{d_1 \sin x}{\sin(180 - w - x)}$	Equation 2: $NA = \dfrac{d_1 \sin w}{\sin(180 - w - x)}$
$\triangle MAP$	
Equation 3: $MA = \dfrac{(d_1 + d_2) \sin z}{\sin(180 - w - z)}$	Equation 4: $PA = \dfrac{(d_1 + d_2) \sin w}{\sin(180 - w - z)}$
$\triangle NAP$	
Equation 5: $NA = \dfrac{d_2 \sin z}{\sin(180 - y - z)}$	Equation 6: $PA = \dfrac{d_2 \sin y}{\sin(180 - y - z)}$

By pairing Equation 1 with 3, 2 with 5, and 4 with 6, we find three equalities that are true if—and *only* if—the three perspectives lie along a common line and don't cross a visibility line!

Table 4: Equalities

Equalities
Comparing the length of $\overline{MA}$ between triangles (equations 1 and 3)
$\frac{d_1 \sin x}{\sin(180 - w - x)} = \frac{(d_1 + d_2)\sin z}{\sin(180 - w - z)}$
Comparing the length of $\overline{NA}$ between triangles (equations 2 and 5)
$\frac{d_1 \sin w}{\sin(180 - w - x)} = \frac{d_2 \sin z}{\sin(180 - y - z)}$
Comparing the length of $\overline{PA}$ between triangles (equations 4 and 6)
$\frac{(d_1 + d_2)\sin w}{\sin(180 - w - z)} = \frac{d_2 \sin y}{\sin(180 - y - z)}$

Now we have a way to check if our three perspectives can be used to calculate a vertex's coordinates!

2.6 Putting it all together: our travel algorithm

In this section, we will develop our final algorithm. The flowchart in Figure 15 provides an overview of the process.

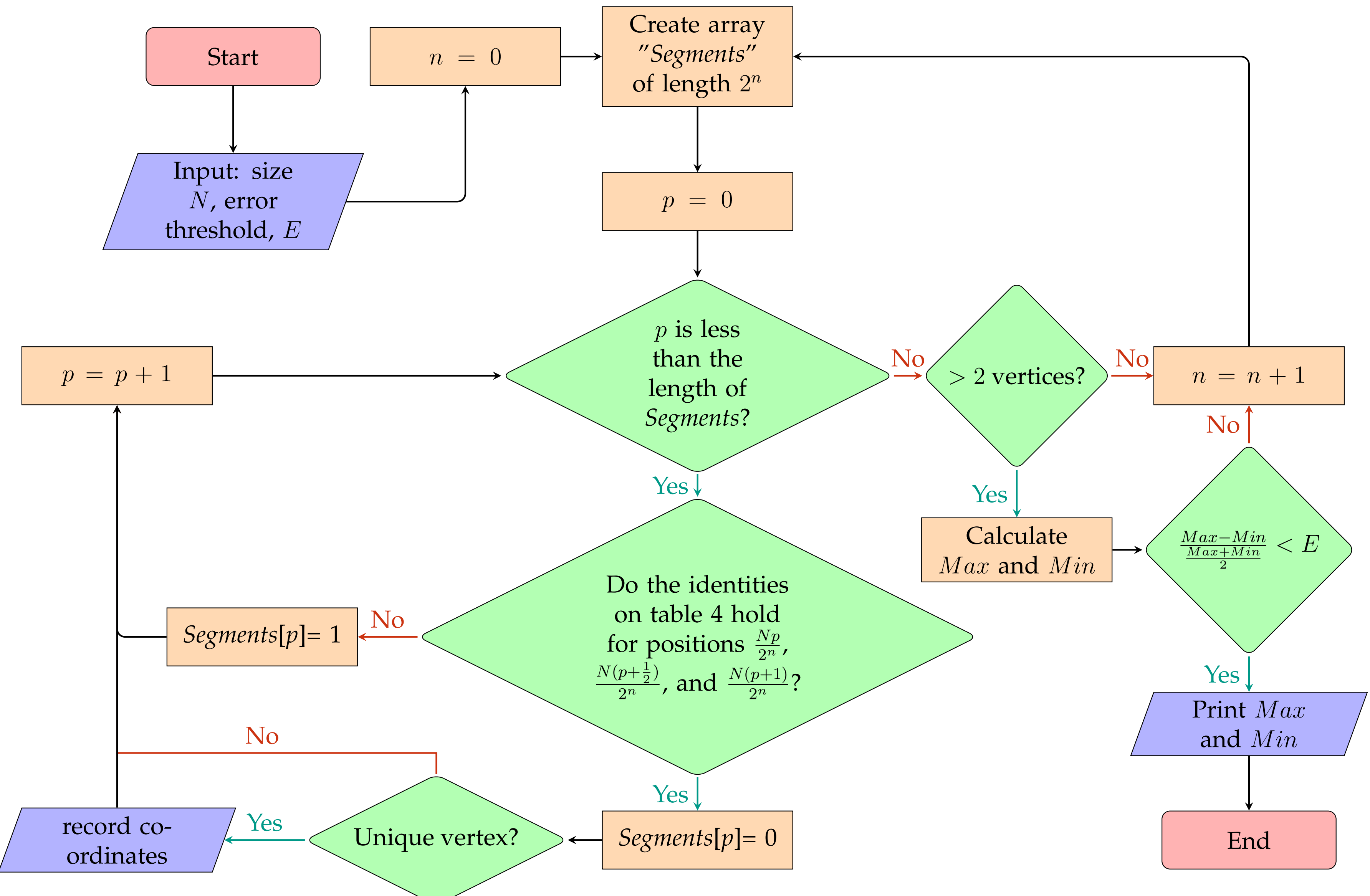

Figure 15: Solution process

2.6.1 Walkthrough

Define the line the ship travels on as an axis. Let the ship's path start at 0 and extend to some arbitrarily large number N. The line should be positioned such that $\frac{N}{2}$ is "close enough" to being the closest point on the ship's path to the island.[7]

Measure distances d_1, d_2 and the angles w, x, y and z at each position. When you plug them into the identities from Table 4, at least one of them shouldn't hold for the first sample, so the query should return "false". This is because we defined our initial range to be large enough to cross all non-parallel visibility lines.

Whenever the query returns "False", we will check the current minimum and maximum areas from the points that have already been collected (discussed in the following sections). If we don't have enough vertices to calculate area, or if the upper and lower bounds aren't tight enough, we will bisect the range we query, turning it into two ranges of half the size and inserting a perspective in-between the old ones. This is shown in Figure 16.

It may take a few recursions, but eventually you will find sections where the equalities from Table 4 hold. In that case, you would mark that section as "solved", which can be done in multiple ways. The flowchart in Figure 15 uses an array to track this, but the impact of this tracking on algorithmic time complexity is not explored in this paper. This is discussed in Section 3.1.

If all three equalities return "True", then apply the method found in Section 2.3 to get the coordinates found in your current cell, then add them to a running list of coordinates if you haven't already recorded them previously.

2.6.2 Minimum size

Calculating the minimum size of the polygon from the information given is relatively simple. The lower bound is equal to the area of the polygon formed using the current confirmed vertices. This can be found by using the Shoelace Formula shown in Section 2.1.1.

2.6.3 Maximum size

The maximum size is a bit more complicated. To explain the upper bound, we will appropriate the term "visual hull" that Aldo Laurentini introduced in his 1994 paper "The Visual Hull Concept for Silhouette-Based Image Understanding" [7]. Although Laurentinfi's paper discusses visual hulls in the context of three-dimensional objects, the definition of a visual hull still applies for two-dimensional objects like the ones in our problem.

Laurentini defines a visual hull (for a shape S and set of perspectives, R) as "the maximal object silhouette-equivalent to S with respect to a viewing region, R". And

[7]Note that this wording is intentionally inexact. A previous draft of this paper stated that the ship would take its first measurements "at $-\infty$, 0, and ∞", but after discussing this section with people who know much more math than I do, I've learned that I am not yet qualified to mess with infinity. A "very large" N is an acceptable alternative.

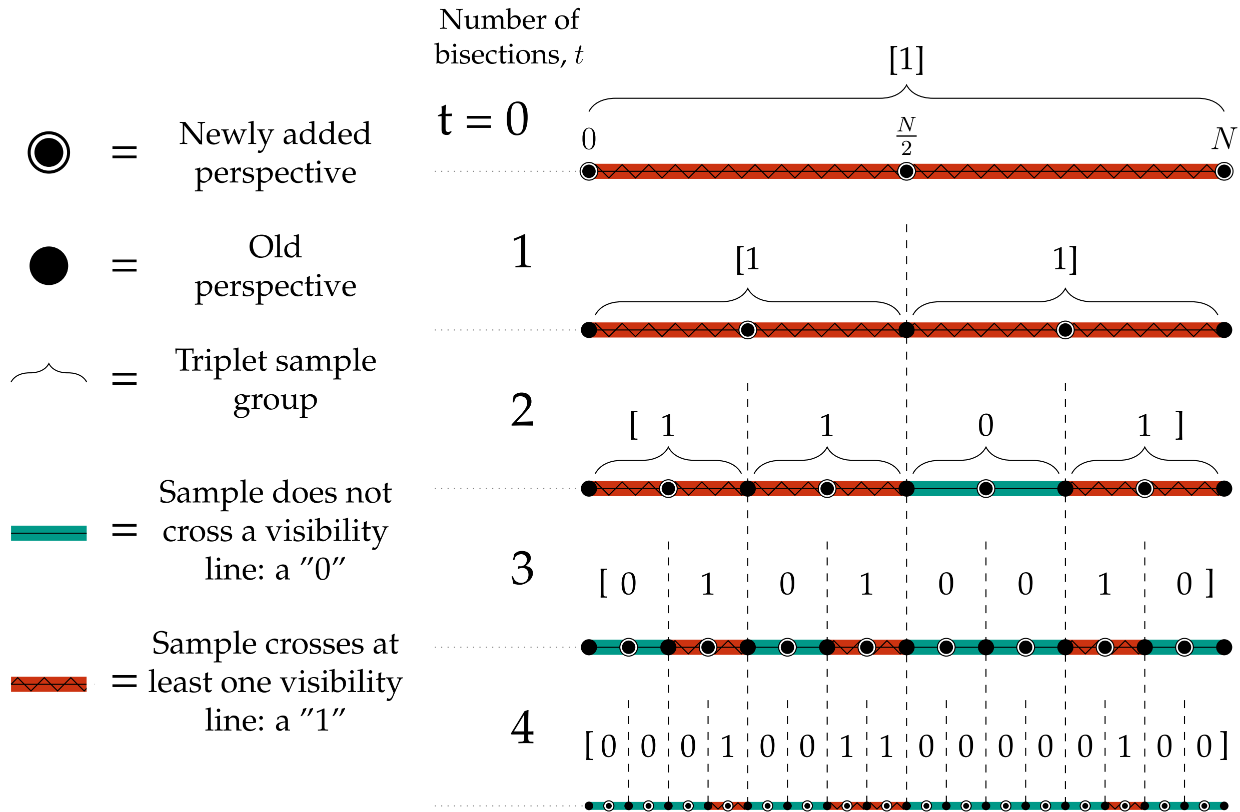

Figure 16: Bisecting the field of view to narrow in on visibility line locations

notes that it "is the closest approximation of S that can be obtained applying the volume intersection technique with viewpoints belonging to R".

In other words, every perspective that we take carves out an area that we know is definitely *not* part of the polygon. An example of the visual hull is shown in Figure 17 [8]:

After infinite iterations of the algorithm, the visual hull will perfectly represent the polygon. We don't want to take infinite measurements, though, so instead we'll settle for an approximation.

Remember that the visual hull is the area that hasn't been carved out yet. Look at the example in Figure 18. Notice that the smallest upper bound occurs when we add the perspectives closest to the visibility line they cross. Moving away from the red band only loosens our upper bound by making the maximum area larger. Fortunately for us, this means we only have to account for the perspectives nearest to the visibility line crossings!

It is also worth noting that we need to consider the area hidden behind the polygon, so when calculating the upper bound, we must also take a measurement of the "phantom points" formed by the pair of perspectives at 0 and N.

We already have a way of calculating the added area from this "uncarved" section.

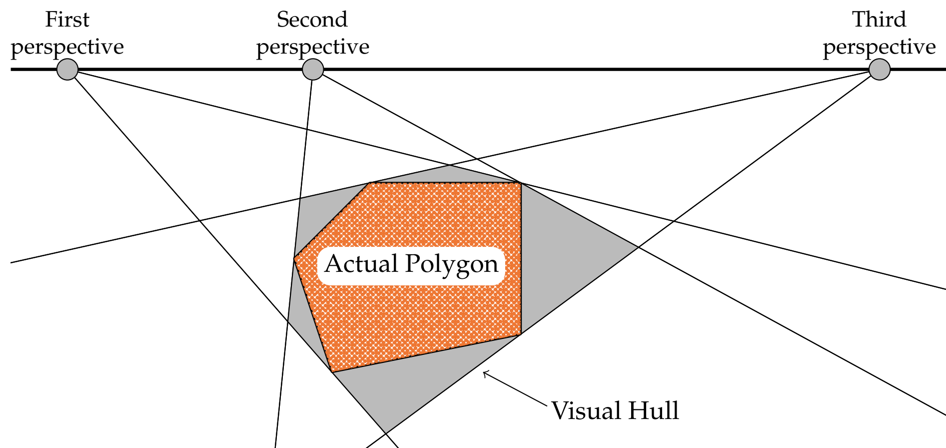

Figure 17: The Volume Intersection Technique

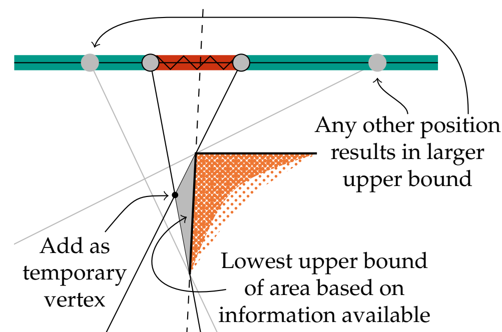

Figure 18: The smallest upper bound forms the visual hull

Remember the "phantom points" from Figure 12? After each step in Figure 16, we apply the formula from Section 2.3 to the two perspectives surrounding every single red band *and* to the perspectives at 0 and N, and we then temporarily add the resultant vertices to the coordinate list. This is our visual hull, and its area is our upper bound.

2.6.4 When to stop

This is a "divide-and-conquer" algorithm. We split the problem into smaller problems (bisecting our ranges to query) and then patch those together to figure out where approximately the ship's path crosses visibility lines.

Think of it like a binary search across an infinite array where every position on the line is an index: even as you repeatedly split the search range in half, you will never reach a point where you can't bisect the range one more time.

Our search along the ship's path is a numerical problem because it deals with continuous values. It is not possible to get an exact answer from this algorithm with finite steps, so we need to settle for finding an upper and lower bound [9]. Our algorithm currently lacks a stopping point. Recursive functions require a base case where they're told to stop or else they become an infinite loop, so we need to find a base case [10].

We will define our base case as the point where the gap between our upper and lower bounds are sufficiently tight, meaning we'll stop once the upper and lower bounds are "close enough" to being equal. Let's call that "close enough" E, our Error threshold.

We want our threshold calculation to scale with size, meaning that it shouldn't be easier to meet our threshold if the island is huge or tiny. If we simply check for whether $Max - Min$ is less than or equal to 3, then the approximation is quite accurate for a 1000-unit island, but relatively inaccurate for a 2-unit island. We can account for this by scaling the range by the average of the upper and lower bounds. If E is the threshold we select, our base case would occur when

$$\frac{Max - Min}{\frac{Max+Min}{2}} < E\,.$$

This should provide a more uniform runtime distribution for a given threshold E, as the range of valid stopping points scales with the polygon's area. This is elaborated on in Section 3.3.

3 Recommendations for further research

I had many ideas while making this project that I did not have enough time to flesh out. Here are some starting points in case you wanted to continue my research:

3.1 Divide and conquer

Look at the red (zigzagged) and green (straight) bands from Figure 16. Notice that there is no need to recheck green bands. We've already shown they contain no visibility lines. This algorithm would run faster if it skipped sections already proven empty.

Keeping track of which parts of our path cross a visibility line using arrays like *Segments[]* from the flowchart in Figure 15 would help significantly reduce runtime, but a specific implementation is out of the scope of this paper.

3.2 Alternate paths

This paper uses an infinite linear path, unfortunately, that is slightly longer than most oceans and inconvenient for real boats. I didn't mention this in Section 2.1.3, but any path that crosses all visibility lines is a possible path for the ship. That could be a square surrounding the island, a circle, arc segment or even an L-shaped path (as pictured below in Figure 19.)

If I were to continue this paper, then I would start by defining an L-shaped path with dimensions as a function of the island's length and width (note that length and width *can* be observed from the outside). This would make the method more practical, but would probably need some modifications to deal with the corners and the change in direction. I haven't looked into paths like arcs or squares, but they should also work.

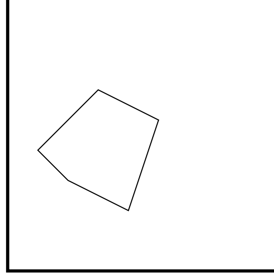

Figure 19: L-shaped path

3.3 Geometric interpretation of the error threshold

While writing this, I realized that there could be a lot more thought put into how the definition of an error threshold can give an "advantage" to bigger or smaller polygons. I found out that if one selects a threshold value of E, then they can check the fraction of the area under the curve of all possible errors that is sufficiently low to stop the search. This curve depends on how we define "error". When we define error as

$$E \leq \frac{Max - Min}{\frac{Max+Min}{2}},$$

the percent of valid areas remains constant for a given threshold value:

$$\%Area = \frac{\int_0^E (Max - \frac{2Max - Max \cdot Min}{Min+2})\, dMin}{E \cdot Max} = \frac{2(E - 2\ln(E+2) + \ln(4))}{E}.$$

I'm sure this has some significance. Intuitively, it seems to show that polygon size gives no time complexity "advantage", but I'll have to put more thought into what constitutes an "advantage", and what the area under the curve represents.

3.4 Integration with other fields

I initially went into this project with an "I need to derive everything from scratch" mindset, but in hindsight, working solo was counterproductive. I wish I had done more background research and worked under the guidance of a math professor, so I could better integrate my original research with other topics.

I recently found out that this problem is linked to the field of computer vision, and this specific problem belongs to a subfield called "shape reconstruction". Knowing this, I wish I looked more into the literature surrounding shape reconstruction, as that would have allowed me to build on the work of others. I don't fully understand these terms, but if you would like to expand this paper, then I'd recommend you look into epipolar geometry, computer stereo vision and photogrammetry, especially if you want to expand this method into three dimensions.

The way I used visual hulls in this paper to find the polygon's upper bound area is like a 2D version of the "Shape-From-Silhouette" technique, in which you take multiple images and determine which parts are part of the foreground and background.

Reading other papers about this helped me discover that, if developed further, this topic is very relevant to the computer vision research of the early 2000s, especially for developing software that is used for motion tracking [11].

Related to Shape-From-Silhouette is the topic of tomography. Usually, tomography involves the casting of beams through the target object and then checking how the beam's properties change, allowing for more information to be gleaned than just whether the beam passes an object or not, but I did find one paper on "Silhouette Tomography" [12] which uses analysis similar to the one seen in this paper. Instead of measuring analog properties of the beam passing through an object, they simplify the information from the beam to a 1 or a 0. I would like to note that their paper takes a different approach to reconstruction. While in this paper we use geometric properties to reconstruct a 2D object, their paper uses a neural network to reconstruct a 3D object. If I refine this algorithm and make it work in three dimensions, then it may be possible to learn more about this problem by comparing my results with theirs.

Inspiration

I started this project in June 2022, over two years ago. My friend showed me a rendering engine he had coded with the help of an online tutorial. The engine reminded me of the 1993 DOS video game *Doom*. I believe we spent the rest of the visit talking about a video I watched that explained how *Doom*'s engine was two-dimensional, but used a technique called "ray casting" to render a convincing three-dimensional environment.

As soon as I got home, I began sketching diagrams to see if I could recreate the engine from scratch. This was entirely self-motivated. I wanted to prove to myself that I could make something independently, so I fumbled around in `Python` until I could successfully render a wall.

Once the school year began, I shifted focus from a rendering engine into a paper exploring a question I had while making the engine: *In Doom, if you walk around a pillar and measure the pillar's size in your field of view every step of the way, could you use that information to calculate the pillar's area?*

Because most people haven't heard of *Doom*, I decided to change this prompt into the analogy seen in this paper's introductory paragraph.

Although I used this paper in my math research class, this project was almost entirely self-motivated and completed independently, which is one thing I somewhat regret. If anyone reading this paper is in a similar situation, then I recommend they look at the surrounding literature instead of trying to figure everything out on your own. There is no shame in asking for help, and I believe I would have learned a lot more about the field of computational geometry if I had asked for help earlier.

Acknowledgements

I was surprised to find that the challenges I encountered writing this paper were more often psychological than technical. For that reason, I'm confident that I would never have finished this project without the encouragement of my teachers at Herricks High School and others.

In particular, I would like to thank Dr. Karim Gangji, who let me squat in his classroom near the end of my junior year, so I could both use his whiteboard and chat about life with him. His advice on creating a balanced work schedule helped me avoid burnout, and his guidance gave me the confidence to take a risk and fully commit to finishing this project.

This paper started more than two years ago as a smaller scale project under my math research teachers, Mr. Salvatore DiLorenzo and Mrs. Kari Tumminello. They encouraged me to continue the project over the summer of 2023 to generalise my result from finding the area of triangles and rectangles to finding the area of any convex polygon.

Finally, I would like to thank Dr. Thomas Britz for both his help in revising and refining my article, as well as relieving me of that lingering fear that all of my work would be for nothing with his positive response to my first draft.

References

[1] N. Frerebeau, Paul Tol's Color Schemes, *https://cran.r-project.org/*, last accessed on 2024-12-25.

[2] A. Kaw, Trapezoidal Rule of Integration, *LibreTexts Mathematics*, last accessed on 2024-12-25.

[3] A. Laaksonen, *Guide to Competitive Programming*, 2nd ed., Springer, Cham, 2020.

[4] H. Shangguan, Z. Miao and I. Jia, Convex Hull Algorithms in 2D, *Medium*, last accessed on 2024-12-25.

[5] R.L. Graham, An efficient algorithm for determining the convex hull of a finite planar set, *Information Processing Letters* **2 (4)** (1972), 132–133.

[6] M. Corral, The Law of Sines, *LibreTexts Mathematics*, last accessed on 2024-12-25.

[7] A. Laurentini, The visual hull concept for silhouette-based image understanding, *IEEE Transactions on Pattern Analysis and Machine Intelligence* **16** (1994), 161–162.

[8] S. Savarese, H. Rushmeier, F. Bernardini and P. Perona, Implementation of a shadow carving system for shape capture, *1st International Symposium on 3D Data Processing Visualization and Transmission* (2002), 12–23.

[9] A. Levitin, *Introduction to the Design and Analysis of Algorithms*, 3rd ed., Addison-Wesley, Boston, 2012.

[10] K. McMahon, Recursion, *CSCI 241 Intermediate Programming in C++*, last accessed on 2024-12-25.

[11] K. Grauman, G. Shakhnarovich and T. Darrell, Virtual Visual Hulls: Example-Based 3D Shape Estimation from a Single Silhouette, *AI Memo 2004-003*, DSpace@MIT, 2004.

[12] E. Bell, M.T. McCann and M. Klasky, Supervised reconstruction for silhouette tomography, *arXiv:2402.07298*, 2024.