Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Incorrect and correct triangulations

Bachelor Thesis

2022
Bohdan Jóža

Comenius University in Bratislava
Faculty of Mathematics, Physics and Informatics

Incorrect and correct triangulations

Bachelor Thesis

Study Programme:	Computer Science
Field of Study:	Computer Science
Department:	FMFI.KAG - Department of Algebra and Geometry
Supervisor:	doc. RNDr. Andrej Ferko, PhD.

Bratislava, 2022
Bohdan Jóža

Univerzita Komenského v Bratislave
Fakulta matematiky, fyziky a informatiky

ZADANIE ZÁVEREČNEJ PRÁCE

Meno a priezvisko študenta: Bohdan Jóža
Študijný program: informatika (Jednoodborové štúdium, bakalársky I. st., denná forma)
Študijný odbor: informatika
Typ záverečnej práce: bakalárska
Jazyk záverečnej práce: anglický
Sekundárny jazyk: slovenský

Názov: Incorrect and correct triangulations
Nesprávne a správne triangulácie

Anotácia: Viaceré publikované algoritmy triangulácie jednoduchého mnohouholníka v rovine boli chybné. Vysvetlíme algoritmy a napravíme chybné predpoklady, na výučbu i propagáciu výpočtovej geometrie.
Vytvorime systém na implementáciu a vizualizáciu algoritmov triangulácie mnohouholníka v rovine.

Cieľ: 1. Výskum daného problému a algoritmov.
2. Výklad podľa metodiky Forišek-Steinová (Prehľad, Metafora, Rozbor, Skúsenosti, Cvičenia).
3. Vizualizácia algoritmov.

Literatúra: FORIŠEK, M., STEINOVA, M. Explaining Algorithms Using Metaphors. Springer 2013.
SELLARÈS, J. A., TOUSSAINT, G. 2003. On the role of kinesthetic thinking in computational geometry. International Journal of Mathematical Education in Science and Technology. Roč. 34, č. 2, s. 219–237.
CHALMOVIANSKÝ, P. et al. Zložitosť geometrických algoritmov. UK 2001.

Kľúčové slová: Triangulácia jednoduchého polygónu, algoritmické stratégie

Vedúci: doc. RNDr. Andrej Ferko, PhD.
Katedra: FMFI.KAG - Katedra algebry a geometrie
Vedúci katedry: doc. RNDr. Pavel Chalmovianský, PhD.

Spôsob sprístupnenia elektronickej verzie práce:
bez obmedzenia

Dátum zadania: 20.10.2021

Dátum schválenia: 11.11.2021

doc. RNDr. Daniel Olejár, PhD.
garant študijného programu

.....
študent

.....
vedúci práce

Comenius University Bratislava
Faculty of Mathematics, Physics and Informatics

THESIS ASSIGNMENT

Name and Surname:	Bohdan Jóža
Study programme:	Computer Science (Single degree study, bachelor I. deg., full time form)
Field of Study:	Computer Science
Type of Thesis:	Bachelor´s thesis
Language of Thesis:	English
Secondary language:	Slovak

Title: Incorrect and correct triangulations

Annotation: Several published simple polygon triangulation algorithms have been incorrect. We will summarize algorithms and correct assumptions for teaching and promotion of computational geometry. We will create a system for the implementation and visualization of polygon triangulation algorithms in a plane.

Aim:
1. Research of given problem and algorithms.
2. Explanation according to the Forišek-Steinová methodology (Overview, Metaphor, Analysis, Experience, Exercises).
3. Algorithm visualization.

Literature:
FORIŠEK, M., STEINOVA, M. Explaining Algorithms Using Metaphors. Springer 2013.
SELLARÈS, J. A., TOUSSAINT, G. 2003. On the role of kinesthetic thinking in computational geometry. International Journal of Mathematical Education in Science and Technology. Roč. 34, č. 2, s. 219–237.
CHALMOVIANSKÝ, P. et al. Zložitosť geometrických algoritmov. UK 2001.

Keywords: Simple polygon triangulation, algorithmic strategies

Supervisor:	doc. RNDr. Andrej Ferko, PhD.
Department:	FMFI.KAG - Department of Algebra and Geometry
Head of department:	doc. RNDr. Pavel Chalmovianský, PhD.
Assigned:	20.10.2021

Approved: 11.11.2021 doc. RNDr. Daniel Olejár, PhD.
Guarantor of Study Programme

.....
Student

.....
Supervisor

Acknowledgments: I would like to thank doc. RNDr. Andrej Ferko, PhD. for a constant flow of inspiration and unique, valuable insight.

Abstrakt

Ako rozdeliť jednoduchý mnohouholník na trojuholníky? Ako nájsť diagonálu mnohouholníka? Prvé pokusy odpovedať na tieto otázky vedú k nesprávnym algoritmom. V tejto práci vysvetľujeme nesprávne a správne algoritmy triangulácie podľa metodológie Forišek-Steinová, aby sa dali ľahko pochopiť a ukazujeme ich prípadné nedostatky. Navyše implementujeme jednoduché prostredie, ktoré ukazuje tieto algoritmy v akcii. Dá sa použiť na vizualizáciu ďalších algoritmov výpočtovej geometrie na mnohouholníkoch v rovine alebo rozšíriť na vizualizáciu ďalších typov algoritmov.

Kľúčové slová: triangulácia jednoduchého polygónu, algoritmické stratégie, kinestetické myslenie, vizualizácia algoritmov

Abstract

How to divide a simple polygon into triangles? How to find a diagonal in a polygon? The first algorithms that come to mind are incorrect. In this thesis we offer explanations of incorrect and correct triangulation algorithms according to the Forišek-Steinová methodology, making them easier to understand and pointing out the flaws where necessary. Additionally we implement a simple environment that displays said algorithms in action. It can be used to visualize other computational geometry algorithms on polygons in a plane or extended to visualize additional types of algorithms.

Keywords: simple polygon triangulation, algorithmic strategies, kinesthetic thinking in computational geometry, algorithm visualization

Contents

List of Figures

Introduction

The computational geometry problems were always the hardest. Asking anyone at any level of competitive programming what the category of problems you fear the most is, the answer was always the same: computational geometry. To be able to solve them, you not only need intelligence and creativity, as in the case of the other problems but also remember how to correctly implement points, angles, distances, dot products, cross products... The task statements used to be dry, the solution was usually a combination of the already mentioned techniques and solving these problems just wasn't ejoyable. But later, at the university, I started thinking about this again. What makes a problem more attractive and fun to solve? Why are we devoting resources to solving already thoroughly explored problems?

Computer science advanced a great deal in the last decades. The landscape of the unknown where discoveries lurk was shrinking as the optimal solutions to most common and practical problems were being found. Development nowadays consists primarily of putting together existing building blocks and using highly optimized algorithms imported from libraries. Computational geometry is no exception and canonical algorithms like polygon triangulation and convex hull are also entering the import-when-needed territory.

However, once in a while, the need to understand one of the building blocks arises. Perhaps it doesn't behave in the desired way, or it may be more efficient to implement a modified version than to use the output straight away. Studying the source code of an algorithm can take a while, and at this point, a simple explanation can save a lot of time.

Many teachers and scientists believe that the perfect understanding of a topic is the ability to explain it effectively to others. So until a minimal explanation for a topic exists, we can't be sure we have explored it completely.

Additionally, a deeper understanding of a subject can lead to new discoveries elsewhere using the same principles or finding previously invisible similarities. Visual representation is especially helpful in this regard because visual similarities are the easiest to notice. The second half of this thesis is dedicated creating a visual representation of algorithms. The purpose is to offer the ability to *see* an algorithm with minimal adjustments in the code.

At the Formal Languages and Automata class the idea of minimalization first fascinated me. It can be implemented in many areas, ranging from minimal automata to practical explanations. This thesis aims to explain the first incorrect and correct triangulation algorithms in a simple way and summarize the polygon triangulation journey from the problem definition to an optimal solution.

Chapter 1

Definitions

Polygons are essential geometric structures with many practical applications, ranging from blueprints to shape recognition. They are an excellent digital representation of real-world objects. Most of the time, however, the real world is not simple and this needs to be reflected in its representation as well. Dealing with polygons in their entirety can be too complicated and most of the time it would be best to divide them into simpler structures. As triangles are the simplest polygons with a multitude of desirable properties, so arises the problem of polygon triangulation — th edecomposition of a polygon into triangles.

The problem of polygon triangulation is a thoroughly explored topic in computational geometry with many published correct and incorrect algorithms. In this chapter, we will define the problem and the terminology used to offer an overview and explanation of some algorithms.

1.1 Notation

In this thesis we will be working with simple polygons on a two-dimensional plane.

Definition 1. *Vertex* is a point on a two-dimensional euclidean plane. Points are represented by cartesian coordinates $a = (x, y);\ x; y \in \mathbb{R}$. Vertices will be labelled as upper case letters, like A or X.

Definition 2. *Edge* is a segment connecting two adjacent vertices of a polygon. Edges will be labelled as pairs of uppercase letters or a lowercase letter, indicating that edge $e = AB$ connects vertices A and B.

Definition 3. *Polygon* is a sequence of vertices such that every two consecutive vertices (also the first and the last) are connected by edges. Polygons will be labelled as upper case letters, $P = ABC$ is a polygon with vertices A, B and C.

Definition 4. *Simple polygon* is a polygon with straight, pairwise non-intersecting edges and no holes.

The definitions of vertices, edges and polygons were more or less intuitive. Let's define the triangulation of a polygon.

Definition 5. *Triangulation* of a polygon P is any set of triangles (polygons with three vertices) such that

1. Each vertex of each triangle must be one of the vertices of P
2. No two triangle interiors share a common point
3. The union of these triangles is exactly P

Now we will prove that every polygon can be triangulated. First, we have to prove two other facts, then the existence of a triangulation will be proven easily.

Definition 6. *Convex vertex* is such a vertex of a polygon that it's two edges form a convex angle (with regard to the inside of the polygon).

Lemma 1. *Each polygon has at least one strictly convex vertex.*

Imagine a walker walking along the edges of a polygon P in counter-clockwise direction. The interior of the polygon is always to the left of this walker. A convex vertex is a left turn and a concave vertex is a right turn. Let x be the left-most vertex of P (with minimal x coordinate in P), if there are multiple, then the lowest one. x has to be a convex vertex because the walker has to turn left at the left-most point of the polygon, otherwise x wouldn't be the lowest.

Definition 7. *Diagonal* is a segment connecting two non-adjacent vertices of a polygon.

Definition 8. *Inner diagonal* is a diagonal that is entirely inside its polygon.

Lemma 2. *Every polygon with more than three vertices has an inner diagonal.*

Finding an inner diagonal is a crucial part of several triangulation algorithms. This will be proven later in the wrong algorithm analysis, let's assume it's true when proving the existence of a triangulation.

Theorem 1. *Each polygon P with n vertices can be triangulated (divided into non-overlapping triangles).*

Proof by induction. If $n = 3$, the polygon is a triangle and the theorem is true. Otherwise, let $d = XY$ be an inner diagonal of P, which exists because of 2. An inner diagonal divides a polygon into two smaller polygons P_1 and P_2, to which d is an edge.

Because these polygons have fewer vertices than P, if they can be triangulated, P can be triangulated as well, because the conjunction of the triangulations of P_1 and P_2 is a triangulation of P.

We can observe that every triangulation of an n-gon (polygon with n vertices) contains exactly $n-2$ triangles, similar proof by induction.

This proof is basically a constructive algorithm that constructs the triangulation by adding inner diagonals to the polygon.

1.2 Terminology

1.2.1 Kinesthetic thinking

Sellarés and Toussaint [1] distinguish between *logico-mathematical* thinking and *kinesthetic* thinking.

> The term *kinesthetic thinking* (...) signifies direct cognitive operations on tactile kinesthetic sense experiences. A kinesthetic heuristic is any heuristic in which cognition, undertstanding and learning takes place through perceptible results of dynamic manipulation [1].

Our knowledge and intelligence are originally acquired through visual, audial and other interactions with our environment, especially in the field of computational geometry. We live in a three-dimensional world that is often simplified into two dimensions in order to analyze it effectively. After we acquire information and form concepts though, we are able to perform logico-mathematical thinking without the influence of the 'real world'. Understandability of proofs and algorithms is crucial to facilitate progress. The faster we can teach existing scientific progress to new generations of researchers, the more time they will have to move the research forward. In addition, being able to explain a subject clearly is some indication of truly understanding the subject in great depth. There are various levels of understanding, but to be able to transfer knowledge to others effectively, one really has to have the deepest understanding of the topic himself.

Kinesthetic heuristics are easier to understand and easier to come up with. Sometimes even faster than their logico-mathematical counterparts. (In the case of the convex hull problem, for example.) Sellarés and Toussaint [1] call kinesthetic heuristics 'a double-edged sword'. As we will see in the case of polygon triangulation, some published algorithms in computational geometry turned out to be wrong. All of them were based on kinesthetic heuristics, and they were efficient. Because algorithms based on kinesthetic heuristics are usually faster, they are an excellent pedagogical strategy for the design of fast algorithms. On the other hand, correctness of these algorithms is harder to prove because of the gap between the logico-mathematical and kinesthetic.

Kinesthetic metaphors can be compelling, but the proof of correctness needs to be logico-mathematical, so some conversion is necessary.

We can illustrate the usefulness of kinesthetic thinking on the Knapsack problem. The formal definition as an NP optimalization problem is as follows:

relation R, target max and value function m such that

$$R = \{(x, y) \mid x = d(w_1)\#d(w_2)\# \dots \#d(w_n)\#d(c_1)\#d(c_2)\# \dots d(c_n)\#W$$

$$y = d(S),\ S \subseteq \{1, 2, \dots n\}, \sum_{i \in S} w_i \leq W$$

$$m(x, y) = \sum_{i \in S} c_i \leq W$$

(d(x) is a binary representation of x).

The problem is then for any given x to find y_x where $\forall y\ m(x, y_x) \geq m(x, y)$.

Did you understand the problem definition? How long did it take?

Now let's say we are going on a vacation and quickly need to pack. We have one bag with limited space W and the choice of N items; item i has a value c_i and will take up space w_i; $i \in \mathbb{N}$. How can we choose some items so that they fit together in the bag and have the largest total value?

The metaphor is generally easier to understand and even leads to the design of the solution right away. What if we consider the items one at a time? How does the decision of taking or not taking one item depend on the decisions regarding the previous items? And so on.

1.2.2 Metaphor & Analogy

As we will be talking about metaphors and analogies, it is important to define how we understand these terms. In the book Explaining Algorithms Using Metaphors, there are 2 relevant definitions based on Lakoff & Johnson [2].

> A *(conceptual) metaphor* is a cognitive process that occurs when a subject seeks understanding of one idea (the target domain) in terms of a different, already known idea (the source domain). The subject creates a conceptual mapping between the properties of the source and the target, thereby gaining new understanding about the target [2].

> An *analogy* is a cognitive process in which a subject transfers information from one particular object to another. The word analogy can also be used as a noun describing the similarity between the two particular objects [2].

As is usually the case, these terms mean different things in different contexts. For example, 'the homework was a breeze' is a metaphor in the context of literature, but we can hardly draw conclusions about the homework from thinking about a breeze. There is a difference between these terms. An analogy shows some properties of the target on the source, but a good metaphor allows the inference of other information about the target based on the connection with the source. On the other hand, a bad metaphor leads to wrong conclusions and confusion as a result.

1.3 Methodology

In this thesis, we will explain algorithms using the Forišek-Steinová methodology. In the main chapters of the book Explaining Algorithms Using Metaphors [3], topics from selected areas in computer science are presented using the same structure consisting of five parts.

Overview A brief overview of the topic is given, stating the problem definition and necessary background to follow the next sections.

Metaphor The core part where the metaphor is developed and it is shown how it can be used to teach the topic.

Analysis The problem is approached from a scientific point of view. The metaphor is extended to a complete algorithm solving the problem if necessary. A deeper analysis of the topic, mentioning the best solutions and research on the topic.

Experience Some didactic notes for teachers and possibly a recollection of the author's experience when teaching this topic.

Exercises Several exercises relevant to the topic are presented, providing an opportunity to apply the newly acquired knowledge to an unknown problem and measuring their understanding of the topic.

This thesis is focused on a deeper analysis of one problem instead of an overview across multiple topics. We apply a modified version of this structure to multiple algorithms and metaphors related to this topic. The sections Experience and Exercises are not included because they refer to the problem rather than the individual algorithms. On the other hand, the Metaphor and Analysis sections are essential in the following effective explanations of selected correct and incorrect triangulation algorithms.

Chapter 2

Algorithms

As mentioned in the previous chapter, the proof of existence of triangulation in any simple polygon is basically a constructive algorithm creating the triangulation. In fact, this was the first published solution to this problem, designed by Lennes in 1911 [4]. It is a Divide & Conquer algorithm, where for any polygon P:

1. If P is a triangle, we are done. If not, find an inner diagonal of P.
2. Using this diagonal divide P into two new polygons P_1 and P_2
3. Recursively find the triangulation of P_1 and P_2. their union is the triangulation of P

The problem of triangulation is thus reduced to the problem of finding an inner diagonal of a polygon.

Let's take a look at some kinesthetic heuristics for finding inner diagonals in polygons.

2.1 Inner diagonals

All algorithms follow the same pattern:

Find the left-most vertex of polygon P; call it B. If there are multiple such vertices, take the lowest one (with minimal y coordinate). Then based on Lemma 2 B is a convex vertex.

Let A, C be it's neighbours. If triangle $T = ABC$ doesn't contain any other vertices, AC is an inner diagonal.

Otherwise, choose a vertex inside T and say that BD is an inner diagonal.

The difference between these algorithms is the way they select vertex D. Each of them is some kind of kinesthetic heuristic, each of them was published and each of them is incorrect.

There are three fundamental transformations in computational geometry: scaling, translation and rotation. The following heuristics are intuitively inspired by these transformations.

2.1.1 Wrong heuristics

The easiest way to prove the incorrectness of an algorithm is to show an example input where the algorithm produces an illegal output. At copmetitive programming competitions, the algorithms are tested on random large inputs to reduce the probability of an incorrect algorithm passing these tests to the minimum or even impossibility. This is in attempt to guarantee that only the correct algorithms are accepted. But in the case of manually simulating an algorithm to understand why it is incorrect, large inputs are impractical. It is better to design tricky inputs that make as many algorithms fail as possible while being as small as possible. In this case, the example that disproves all of the following heuristics has to have at least five vertices: A, B, C that form the triangle $T = ABC$ and two more inside the triangle; a correct vertex D such that BD is an inner diagonal and another vertex E such that BE is not an inner diagonal. With four vertices, there would be only one candidate for the vertex D and thus all heuristics would find it (observe that at this stage, BD has to be an inner diagonal).

As it turns out, there is an example that fails all three and together with its twin (2.1) makes a solid sample test case. The second version shows incorrect algorithms approaching the problem from the other side; either rotating or sweeping from the other direction.

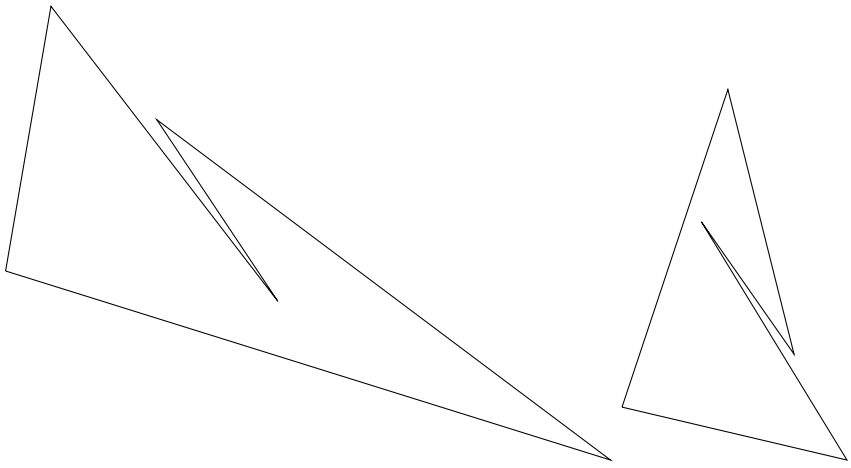

Figure 2.1: Minimal counterexample (left); with D and E inverted (right)

Scaling

Find the closest vertex to B inside triangle T [1]

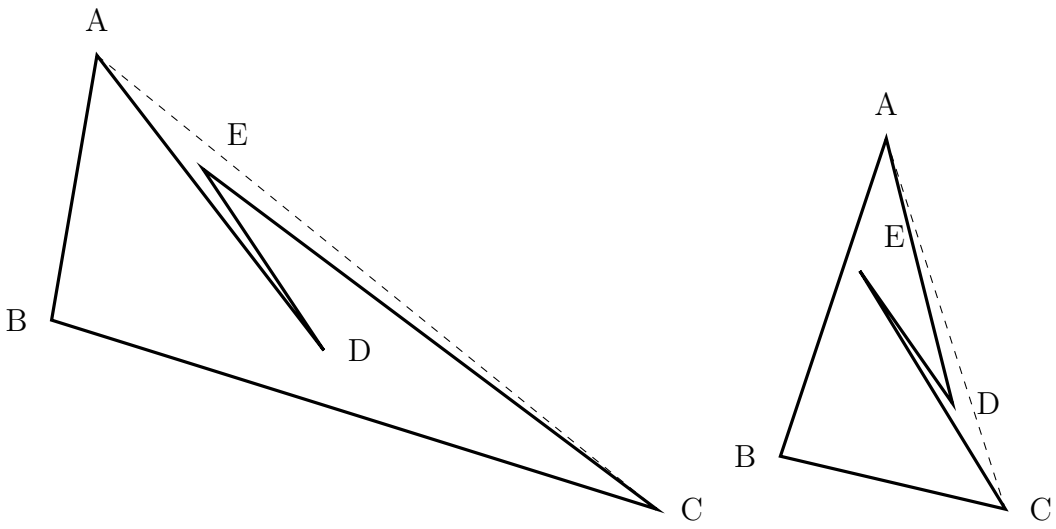

Figure 2.2: D and E are inside triangle ABC

The very first and simplest heuristic that comes to mind is simply taking the closest vertex to B. It is basically equivalent to growing a circle from B and whichever vertex inside T it hits first is chosen as the other end of the diagonal. Hence, the connection to scaling. The common analogy for this type of algoritm is spilled water. If B was a fountain and the other vertices were water outlets, the water would flow through D first.

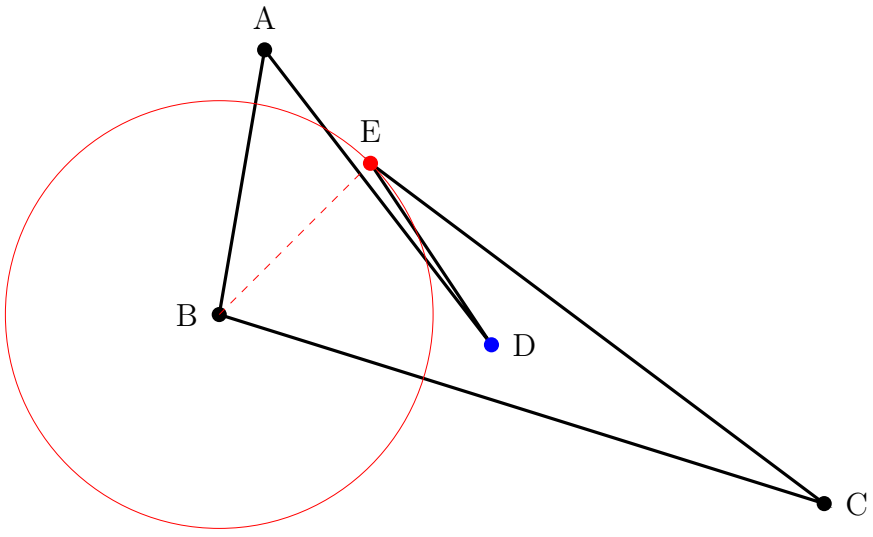

Figure 2.3: Scaling heuristic

This heuristic doesn't always produce an inner diagonal because it does not consider the boundaries of the polygon. The overall closest vertex to B does not have to form an inner diagonal with B, because there is nothing preventing the circle from crossing some edges of the polygon before choosing vertices.

Rotation

Find the vertex D inside T such that the angle CAB is the smallest. [1]

Imagine the polygon as a clock. It seems a little far-fetched at first, but we can set B as the middle and the other vertices inside T as hour marks. Different distances of the hour marks from the middle are a little unusual, but every clock tells the correct time at least twice a day. The heuristic is then very simple: set the time to A and wait a minute. The first point the minute arm crosses is the the inner diagonal's other end.

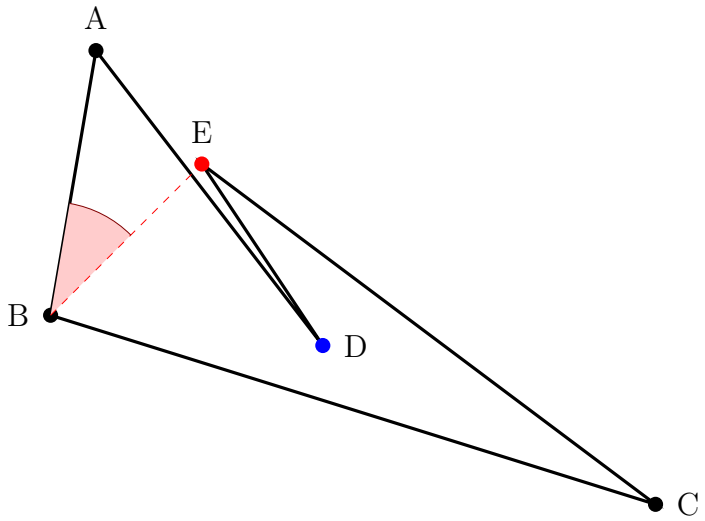

Figure 2.4: Rotation heuristic

Using a rotating line to find D doesn't always yield an inner diagonal. The reason for that is similar to the Scaling heuristic; the line doesn't take into account any edges inside T. Rotating the line in a counter-clockwise direction from C could also result in an incorrect vertex.

Translation

Find the left-most vertex inside triangle T. [1]

This would probably be the second most common heuristic. If the overall closest vertex doesn't work, what about the left-most vertex? Again, we draw a vertical line passing through B, but instead of rotating it we start moving it across the polygon. Whichever vertex it hits first we choose as D. We basically sweep the points until we hit one inside T.

As was the case in the previous heuristics, we again disregard the edges of the polygon. Nothing is preventing the line from leaving the inside of the polygon before hitting the vertex.

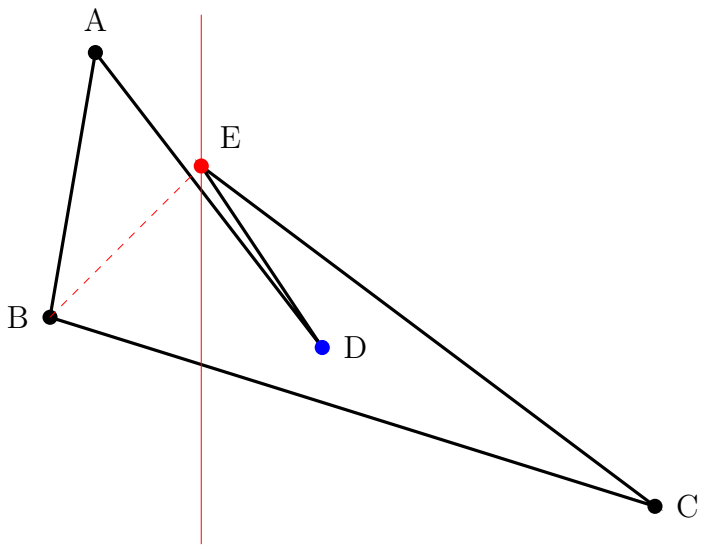

Figure 2.5: Translation heuristic

2.1.2 Correcting the heuristics

In this section, we will take a look at some corrections of the incorrect heuristics [1] and one other correct kinesthetic heuristic [3]. Any of these corrected heuristics is a constructive proof of existence of an inner diagonal (if the polygon has more than 3 vertices) mentioned in the previous chapter.

Correct Translation

Our algorithmic intuition should indicate that if we choose some constant when solving a more complex problem, there has to be a reason. For example, in this triangulation algorithm, we always select the left-most vertex as B. But that is because it is always convex. In the case of the Translation heuristic, there is no reason to choose vertical as the direction of the line straight away. The correction of this heuristic requires rotating the line to have the same direction as ca before sweeping. Then the first polygon found by this line always forms an inner diagonal with B. The essential difference is that at no point can any part of the line be outside the polygon AND inside T before hitting the first vertex.

Observe that when we sweep the vertices in T with any line starting at B, the line has to hit a vertex before any part of the line inside T leaves the polygon. This could be A or C or one of the candidates for D. In the case of a fixed horizontal line we were able to make the line hit A first, which can't be D, so we continued sweeping. But now a part of the line is outside of the polygon and exactly that part hits the incorrect vertex before the part inside T. By aligning the line with AC we force the line to hit A and C at the same time, at which point the line also leaves T. So the first vertex the line hits is guaranteed to be one of the candidate vertices and at that point no part of

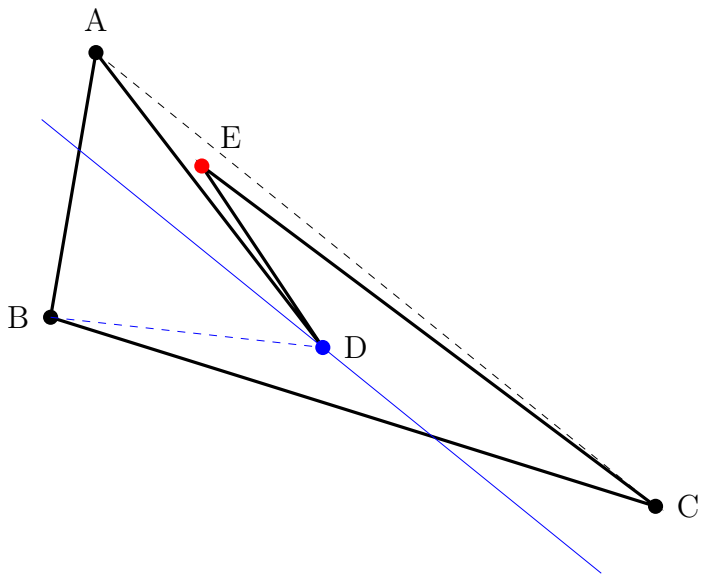

Figure 2.6: Correct translation heuristic

the line that is inside T has left the polygon.

Correct Scaling

If we were able to make the water respect the edges by turning them into some kind of boundaries, the water would choose the correct vertex because it is always inside P and T. This correction is hard to implement, maybe even increases the time complexity. Another correct variant requires looking at the problem in reverse. One approach to solving problems that often gets overlooked is to 'go backwards' while solving an opposite problem. In this case instead of searching for the vertex closest to B we search for the vertex that is farthest away from AC. Because of this connection, this always finds the same vertex as the correct Translation algorithm.

Correct Rotation

The correct version of the Rotation algorithm places the clock in C instead of B and sets the time to B.

Interestingly enough, similar observations can be made about this situation as in the case of correct Translation heuristic. When we sweep the vertices with a rotating the line this way, before any part of the line in T leaves the polygon it has to hit a vertex. And as it again can't be A, B or C so it has to be one of the candidate vertices. When we set the time to A and started rotating in the incorrect version, some part of the line inside T was outside the polygon and we were able to make that part hit a vertex before the correct part did.

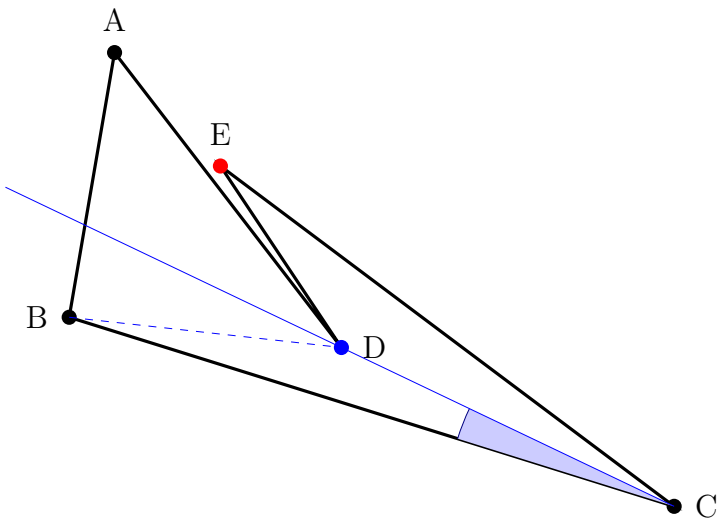

Figure 2.7: Correct rotation heuristic

Gravity

The metaphor described by Forišek and Steinová [3] uses a ball, a rubber band and gravity. First, find any *concave* angle of the polygon. Rotate the polygon so that both sides of this angle point upward and no edge is horizontal. Attach the rubber band to the ball and the vertex that is the tip of the concave angle, then drop the ball. It will fall until it hits a side. Let the ball slide down the side until it reaches a vertex. At this point if the rubber band is straight, then the vertex it ended in and the tip form an inner diagonal. If not, the band had to hit another vertex. The first vertex it hit and the tip form an inner diagonal.

This algorithm is also correct because the band is always inside the polygon, so it is guaranteed that it hits any vertex from the inside of the polygon. Consequently, the diagonal it creates is also completely inside.

2.1.3 Analysis

As already described, these heuristics lead to a Divide & Conquer algorithm. Each of them can be implemented in linear time and we have to repeat the algorithm $n - 3$ time (for each diagonal), resulting in $O(n^2)$ time complexity.

The implementation of these heuristics if mostly straight-forward. After sorting the vertices, finding A, B and C in any subpolygon is trivial, then we separate those inside triangle ABC and for each of them calculate the distance from AB or the angle BCD. The implementation of the Gravity algorithm is well described in Explaining Algorithms Using Metaphors [3].

2.2 Ears

2.2.1 Metaphor

Imagine the polygon as some exotic fruit. It isn't practical to eat it as it is, so we first want to slice it into triangles that we can munch on one by one. We rotate the fruit until we find a spot, where we can comfortably make a cut from vertex to vertex and slice a triangle off. We repeat this process until there is only a triangle left.

2.2.2 Analysis

This algorithm was first discovered by Meisters [5], who published a greedy ear-finding algorithm leading to an $O(n^3)$ time algorithm. He proved the two-ears theorem, which states that each simple polygon with more than 3 vertices has at least 2 non-overlapping ears. Vertex V_i is an ear If triangle $V_{i-1}V_iV_{i+1}$ contains no other vertices. The trivial algorithm for finding an ear checks this requirement for each vertex in $O(n^2)$ time. But later a linear-time algorithm was discovered and published by Toussaint [6]. Interestingly enough, it is similar to the triangulation algorithm itself in that it slices chunks off the polygon until only an ear remains. And it can be transformed into another, more efficient triangulation algorithm.

A *good subpolygon* of a polygon P is a subpolygon whose boundary differs from that of P in at most one edge. The edge not in P is also called a *cutting edge*. Before presenting the idea of the algorithm, we need to understand some facts.

An inner diagonal of a good subpolygon divides it to one good subpolygon and one that is not good. That is because one of the new subpolygons will also contain the cutting edge of the original one. (Assuming the original one has a cutting edge, otherwise both will be good subpolygons) The idea of the algorithm is following:

Given a strictly good subpolygon G of P and vertex V_i

1. Check if V_i is an ear. If so, we are done.

2. Find an inner diagonal $d = V_iV_j$. It is the cutting edge of a new good subpolygon G'. Rename it's vertices so that $V_i = U_0$ and $V_j = U_{k-1}$ (or $V_i = U_{k-1}$ and $V_j = U_0$)

3. Repeat with G' and $U_{\lfloor k/2 \rfloor}$

In each iteration we either find an ear or cut off approximately half of the polygon. We are guaranteed (from 2nd iteration onward) that $i = \lfloor k/2 \rfloor$ and that V_0V_{k-1} is the cutting edge. When we find V_j such that V_iV_j is an inner diagonal, the good subpolygon always has at most $\lfloor k/2 \rfloor + 1$ vertices. (If $j < i$ then $G' = V_j \dots V_i$, otherwise $G' = V_i \dots V_j$) The only obstacle in the correctness of the algorithm now is a proof that if V_i isn't an ear, it is one end of an inner diagonal. But if that is the case, wither V_i is a convex vertex such that there are some vertices in $V_{i-1}V_iVi + 1$ or it is

an concave vertex. In either case a constructive linear algorithm for an inner diagonal exists, desribed in the previous chapter.

The first and second steps can be performed in linear time so this algorithm's time complexity is $T(n) = cn + T(\lfloor n/2 \rfloor + 1)$. c is a constant so $T(n) \in O(n)$.

An interesting observation is that the ear finding algorithm is basically the same as the Divide & Conquer algorithm from the previous chapter. We perform the same steps: check if B is an ear and if not find an inner diagonal.

2.3 Faster algorithms

More complicated algorithms are naturally harder to explain, so here is merely an overview of some faster triangulation algorithms. Some of them are described in Toussaint's Efficient triangulation of simple polygons [7]. Explaining these using the Forišek-Steinová methodology is a good idea for some future work.

Following research involved designing algorithms for different classes of polygons. If there is some constraint on the input polygon, we can use a faster algorithm leveraging this constraint.

There are many classes of polygons: convex, monotone, edge-visible, star-shaped, spiral, L-convex, intersection-free, and many more. Specifically, the named classes have something in common: the existence of a triangulation algorithm in $O(n)$ time [7].

2.3.1 Garey (1978)

The first algorithm to break the $O(n^2)$ upper bound is a different take on the Divide & Conquer strategy. Instead of recursively solving the subpolygons we first divide the polygon into *monotone subpolygons* and then triangulate each of them separately. [8]

Different versions of the decomposition algorithm and the triangulation of monotone polygon were later published.

2.3.2 Tarjan and van Wyk (1988)

Even faster algorithm was designed by Tarjan and van Wyk in 1988. It uses complicated data structures to achieve $O(n \log \log n)$ time complexity. However, this was later matched by Kirkpatrick (1990) using simple data structures.

2.3.3 Chazelle (1990)

In May 1990 Chazelle finally showed that a polygon can be triangulated in linear time [9]. The algorithm, however, is very complicated and there are doubts regarding it's complete correctness.

2.4 Delaunay triangulation

The problem of triangulation we defined can be generalized. Instead of triangulating a polygon, the input is a set of vertices and the output is a set of non-intersecting edges between these vertices that divide the convex hull of these vertices into triangles. Additionally, if the circumference of these triangles can not contain another vertex, the problem is called *Delaunay Triangulation*. Our definition is then an instance of the *Constrained Delaunay Triangulation (CDT)*, where some edges have been pre-filled (i. e. the edges of the polygon) and the resulting triangulation has to respect these edges. There may be some redundant edges outside of the polygon, but those can be easily excluded. So any solution of the CDT problem can be used to triangulate a simple polygon.

Chapter 3

Visualization

Understanding the algorithm metaphor is one thing, but it is only the first step in solving a problem. One more step is necessary to test the correctness and actually use the algorithm: implementation. Translating an algorithm into a language a computer can understand is often not a straight-forward step-by-step copying of the idea. For example, implementing the gravity heuristic for finding inner diagonals in the previous chapter, we don't rotate the polygon, but merely use the axis of the concave angle as the direction of gravity and find the first intersection of the axis and an edge. To truly understand an algorithm, one also needs to know how it's implementation works.

The brain processes visual input acquired by the eyes all the time, so it developed an effective process for parsing and understanding visual data. This is important because visual stimuli are the most significant source of information about the world around us. Listening and reading about an algorithm may not be enough. Sometimes, to fully understand or to understand faster the algorithm needs to be *seen*. For this purpose, we will create a program with the potential to visualize every step of any computational geometry algorithm. There are some limitations and room for improvement, but it lays a solid foundation for a sandbox of sorts where any algorithm can be visualized.

3.1 Specifications

The aim is to create a tool that helps visualize any geometric algorithm with minimal adjustments in the code. There are various applications for such a tool, be it debugging, examination of an unknown algorithm or illustrating an algorithm in a classroom setting. For the implementation of this visualization tool, a couple of principles are followed.

universality – Keeping everything as general as possible so that the individual building blocks can be used as utlilities separately in other ways. This way the logging part of the project can be used to visualize any type of algorithm when paired with

appropriate visualizer. The class decorators may also have some other practical uses on their own.

simplicity – Keeping everything as simple as possible.

While trying to stay as universal as possible to pave the way for visualizations of other types of algorithms, here we focus on visualizing triangulations or other algorithms on simple two-dimensional polygons.

3.1.1 Previous work

There are many visualizations of algorithms online displaying specific algorithms. But the task of animating one concrete algorithm is easier than animating universally any algorithm or even one class of algorithms. It doesn't even necessarily require programming. We also found one project aiming for something similar [10]. It does a good job, but requires multiple lines of specific modifications in the code. So to visualize an algorithm one needs to understand it beforehand and then basically implement the visualization into the code. Therefore it doesn't quite fit my requirements.

3.1.2 Programming language

The main choices for the programming language were Python and C++. The primary target for this visualization is computational geometry algorithms and in C++, there is a comprehensive Computational Geometry Algorithms Library [11] implementing all essential computer geometry operations and even algorithms including convex hull and polygon triangulation. It is also the language primarily used to implement computer science algorithms, because it is fast and has a close connection to system calls and bitwise operations.

On the othe hand, in Python standard library, there is a default GUI module Tkinter which allows relatively simple visualization bypassing the need to include external modules. Python is also a simple and easy to learn programming language taught to students as an example or sandbox language nowadays. During research we discovered that the core of CGAL was recently migrated to Python as the `scikit-geometry` [12] module. Because of the superior Python experience in previous projects and all necessary computational geometry tools already implemented, Python ended up being the language of choice.

3.1.3 scikit-geometry

The `scikit-geometry` [12] module provides essential computational geometry objects such as Point, Segment, Polygon, Ray, Vector etc. and essential operations on these

objects for example `Polygon.oriented_side(Point)` returns `Sign.POSITIVE` if the point is inside the polygon, otherwise `Sign.NEGATIVE` outside and `Sign.ZERO` on the boundary. This module will be used to implement some triangulation algorithms that will then be visualized.

Even though it is basically a Python copy of the CGAL C++ library, the documentation of `scikit-geometry` is non-exhaustive and lacks the reference to important classes, `Polygon` for instance. There is no reference to CGAL either. So most of the Polygon functionality had to be reverse-engineered.

The setup process was pretty complicated. It can be installed through `conda` environment or by manual build of the repository. The build didn't work, but the setup of the `conda` environment was successful which in turn allowed the installation of `scikit-geometry`.

3.2 Logging

To visualize an algorithm, each step of the algorithm has to be converted into one visual frame representing this step. There are two options for achieving this: transform the source code and a given input with into the visual representation or run the source code on a given input and track it's actions. The first option wouldn't require any modifications to the source code, but it would be needlessly complicated as it is basically creating another compiler, or in the case of Python interpreter. It is easier to use the existing interpreter and just track all objects, logging each action. To create the visualization data we then just run the program.

The most practical way to track an algorithm in Python is to decorate all relevant objects so that they log their actions at runtime. This requires some minimal adjustments in the code, the only requirement is transforming used classes and objects with this decorator.

3.2.1 Python objects and classes

In order to understand the implementation of the logger, some background is necessary about how Python treats objects. All data in a Python program is represented by objects or by relations between objects. Every object has an identity, a type and a value. The identity is basically its address in memory; it is returned by the `id()` function. An object type determines what properties it might have and possible values for that object. It is returned by `type()` function. Types are also objects, more on that later.

The identity and type of an object are unchangeable, but the value might change. There are also immutable objects whose value can't change after initialization. This is determined by object's type; types `int`, `float` and `tuple` are immutable while `list`

and `dict` are mutable for example. User-defined classes are basically custom types. The class (or a type) can be called to create an instance object (or a class object).

Let's take a closer look at class creation in Python. As previously mentioned, classes are objects. As such they also have a type and have to be an instances of something. Classes of classes are called *metaclasses*. The default metaclass is `type`. Call `type(obj)` returns the type of object `obj` and `type(name, bases, namespace)` returns a class object. The latter is basically the dynamic variant of the `class` statement. The process of class object creation involves determining the correct metaclass, preparing the class namespace and creating the class object itself. Every instance `ins` has two special attributes; `__dict__` is a mapping of the class namespace and `__class__` is a reference to its class. Special syntax `dir(instance)` or `dir(class)` returns the keys of the namespace of the given object.

A class instance are created by calling the `__new__()` method of a class object and initialized by calling the `__init__()` method before the object is returned to the caller. Instance attributes are usually fetched by calling `__getattribute__(attr_name)` method, which returns the attribute named `attr_name` or raises an `AttributeError` exception if it doesn't exist. This happens when invoking `instance.attr_name` in code. However, there are some exceptions; classes have special methods that skip the `__getattribute__` machinery. This allows significant speed optimizations. They are surrounded by two underscores, like `__len__()` or `__eq__()` and invoked by special syntax, like `len(instance)` or `instance == other`. If they are called explicitly, i. e. `instance.__len__()`, `__getattribute__()` is invoked with `__len__` as attribute name.

3.2.2 Decorating a class

For the purposes of logging, decorating instances of classes is impractical, as it requires the transformation call at every instance creation. It is simpler to decorate the class object and then make instances of the transformed class which will then automatically log their actions.

The task now is to decorate every method of the class so that it logs each call. There are two main solutions here.

Replacing the attributes

The first option is to iterate through the class namespace and decorate each callable attribute. Using `setattr` then replace the attributes with the decorated ones.

Listing 3.1: Decoration: first approach

```
1 def decorate_methods_parametrized(cls, get_decorator, exclusion=[]):
2     """
```

```
3     Decorates all callable attributes of a class,
4     replacing the original ones.
5     Internally excludes __class__, __repr__ and __new__.
6     """
7     exclusion_function = \
8         exclusion if callable(exclusion) else lambda x: x in exclusion
9     excl = ['__class__', '__repr__', '__new__']
10    for attr_name in dir(cls):
11        if attr_name not in excl and not exclusion_function(attr_name):
12            attr = getattr(cls, attr_name)
13            if callable(attr):
14                setattr(cls, attr_name, get_decorator(attr_name)(attr))
15    return cls
16
17def decorate_methods(cls, decorator, exclusion=[]):
18    return decorate_methods_parametrized(cls, lambda attr: decorator, exclusion)
```

This approach has some limitations and some possible problems. We can only decorate callable attributes this way because the class namespace rarely contains all the attributes from the instance namespace. All attributes defined inside the `__init__()` for example. Some care has to be taken not to override every callable attribute this way. The `__class__` attribute is writable, but it must be a class object. An attempt to overwrite it by a function, which is presumably what the decorator is, will result in an exception, even though the function internally calls the class and returns the instance.

When decorating special methods, there is the danger of infinite recursion. If we try to decorate the `__repr__()` attribute, which all classes have, with a function that prints its arguments before executing, we get infinite recursion. The first argument passed to a class method is always a reference to the instance. When the function tries to print its arguments, `__repr__()` is called implicitly to convert the instance to string representation, but then it tries to print it's arguments again before converting and then we are in infinite recursion. We need to be careful not to call other decorated methods from inside the decorator.

Modifying access

The second approach only overrides the `__getattribute__()` method of the decorated class, leaving all attributes as they are.

Listing 3.2: Decoration: second approach

```
1def decorate_attributes_parametrized(
2    cls,
3    get_callables_decorator=None,
4    get_non_callables_decorator=None,
5    exclusion=[],
6):
7    """
8    Decorates callable and non-callable attributes of a class.
9    The decorators are obtained by calling get_<decorator> with
10   arguments <instance>, <attribute name> respectively.
```

```
11      """
12      exclusion_function = exclusion if callable(exclusion) else \
13          lambda x: x in exclusion
14      def __modified_getattribute__(self, __name):
15          val = cls.__getattr__(self, __name)
16          if exclusion_function(__name):
17              return val
18          if callable(val) and get_callables_decorator is not None:
19              return get_callables_decorator(self, __name)(val)
20          if get_non_callables_decorator is not None:
21              return get_non_callables_decorator(self, __name)(val)
22          return val
23      setattr(cls, '__getattribute__', __modified_getattribute__)
24      return cls
25
26  def decorate_attributes(
27      cls,
28      callables_decorator=None,
29      non_callables_decorator=None,
30      exclusion=[]
31  ):
32      return decorate_attributes_parametrized(
33          cls,
34          (lambda instance, attr: callables_decorator) \
35              if callables_decorator is not None else None,
36          (lambda instance, attr: non_callables_decorator) \
37              if non_callables_decorator is not None else None,
38          exclusion,
39      )
```

This approach modifies the attributes accessed and returns their modified version. It is more convenient this way because there is no need to implement arbitrary exclusions. It is even possible to decorate the `__class__` attribute without raising an exception. There are some advantages compared to the first solution, but some problems nonetheless. It doesn't modify the attributes of the decorated class other than `__getattribute__`. There is an option to utilize the `__getattr__` method to obtain the attribute inside `__getattribute__`. `__getattribute__` is the default attribute access function and `__getattr__` gets called only when `__getattribute__` raises `AtrributeError`. Some classes don't even implement `__getattr__`, the built-in class `int`, for example. Incidentally, all the `scikit-geometry` classes used to implement the triangulation algorithms don't have `__getattr__` attribute either. Instead, the original `__getattribute__` gets stored in a new class method `__original_getattribute__` and then called from the modified version. Although there may be a namespace clash if the class already implements `__original_getattribute__`, this option is overall more reliable.

Since all non-callable attributes use the `__getattribute__` lookup, they can also be decorated this way. Although decorated methods are even more prone to infinite recursion, it opens some interesting possibilities. The `get_non_callable_decorator` function can't access any decorated non-callable attributes because that triggers another call of the decorator function. The non-callable attribute decorator then does not

return a modified function but the modified value of the attribute. Therefore it is more practical to implement the option to provide it separately from the callables decorator. Otherwise, the decorator would have to implement different behaviour based on the type of attribute it decorates.

Infinite recursion

There may be a way to prevent infinite recursion by calling the decorator only once. We could achieve this by manipulating the arguments of the call, passing a new private keyword argument indicating that the call is made from inside the decorator. It can then adjust it's behaviour to avoid recursion. For example if the previously mentioned decorated printing `__repr__` method knew when it's being called from inside another `__repr__` call, the printing would be omitted.

There may be some problems, however. There are possibilities of namespace clashes again and there may be errors caused by a different number of expected arguments. I didn't explore this possibility further as it is outside this project's scope.

Immutability

All of the above class decorators mutate the given class, so the undecorated version won't exist after `decorate_methods` or `decorate_attributes` is called. However, sometimes it is desirable to keep the undecorated version as well. This functionality can be implemented in the following way.

Listing 3.3: Immutable decoration

```
def decorate_attributes_parametrized_immutable(
    cls,
    get_callables_decorator=None,
    get_non_callables_decorator=None,
    exclusion=[]
):
    """
    Decorates callable and non-callable attributes of a class.
    The decorators are obtained by calling get_<decorator> with
    arguments <instance>, <attribute name> respectively.
    """
    exclusion_function = exclusion \
        if callable(exclusion) else lambda x: x in exclusion

    class DecoratedClass(cls):
        def __getattribute__(self, __name: str):
            val = cls.__getattribute__(self, __name)
            if exclusion_function(__name):
                return val
            if callable(val) and get_callables_decorator is not None:
                return get_callables_decorator(self, __name)(val)
            if get_non_callables_decorator is not None:
                return get_non_callables_decorator(self, __name)(val)
            return val
```

```
25
26     return DecoratedClass
27
28 def decorate_attributes_immutable(
29     cls,
30     callables_decorator=None,
31     non_callables_decorator=None,
32     exclusion=[]
33 ):
34     # same as before, transform decorators, call the
35     # parametrized version
```

In this version, we subclass the given class in a completely new class and override the new class' `__getattribute__` method. There is no need to create a new reference to the old `__getattribute__`, because the original class object is available, providing the original. Moreover, if the original would be later overwritten, we would want to use the new version, which is the case in this approach.

Class decorators

Python supports class decorators in the same syntax as function decorators; if a function takes exactly one argument, a class object, following code passes the class object to the decorator before returning it to the caller.

```
1 @class_decorator
2 class MyClass:
3     pass
```

The decorators described in this section can be viewed more as transformation functions. They are able to transform already created class objects. But they can easily be converted into class decorators in the following way.

Listing 3.4: Second approach as a class decorator

```
1 def class_decorator(
2     get_callable_dec=None,
3     get_non_callable_dec=None,
4     exclusion=[],
5 ):
6     def decorator(cls):
7         return decorate_attributes_parametrized(
8             ...
9         )
10     return decorator
11
12 ...
13
14 @class_decorator(callable_dec_generator, ...)
15 class MyClass:
16     pass
```

With this syntax we can't the attribute decorators directly to decorate all attributes of a class, that's why the class decorators and transformators are necessary.

3.2.3 Logging

The objective here is to create a log describing each step of a given algorithm. Every method call in the algorithm has to add an entry to the log when the execution is started and when it is finished. Before explaining the implementation, let's look at the log structure.

The Log

The log structure is basically an array containing log entries. Aside from this array, a dictionary `state` is maintained, with all *active* objects mapped to unique keys. Each such object must have a `__log_repr__` method that returns an immutable representation of the object in its current state. The visualizer then parses these representations and visualizes them in a meaningful way. Following function `_log_call()` creates one log entry for a method call.

Listing 3.5: Creating a log entry

```
def _log_call(origin, call, args=[], kwargs={}, result=None, call_type='return'):
    key = origin if isinstance(origin, str) else origin._log_key
    special_keys =  ['___call']
    if key not in state and key not in special_keys: return
    entry = dict(map(lambda x: (x, state[x].__log_repr__()), state))
    affected_objects = list(
        filter(
            lambda x: x is not None and x in state,
            arg_to_affected_objects(args) + arg_to_affected_objects(kwargs)
        )
    )
    cargs = args_to_keys(args, include=True, chain=False)
    ckwargs = args_to_keys(kwargs, include=True, chain=False)
    cresult = args_to_keys(result, include=True, chain=False)
    log.append((key, call, cargs, ckwargs, cresult, affected_objects, entry,
        call_type))
```

We will first describe the arguments, then the log structure.

```
def _log_call(origin, call, args=[], kwargs={}, result=None, call_type='return'):
```

origin – The instance of a logger class on the given method. Each such instance has to have `_log_key` and `__log_repr__` attributes. If the origin is a logged function, `origin = '___call'`.

call – Name of the called method.

args – Arguments given to the called method, if any.

kwargs – Keyword arguments given to the called method, if any.

result – Value returned by the called method, if any.

call_type – `"call"` when the execution is started, `"return"` when the execution is finished.

args_to_keys – Function that replaces objects with their keys if they have a `_log_key` attribute. The function recursively replaces elements of arrays and dictionaries. If the `include` switch is set, it leaves the non-logger objects as they are, otherwise it replaces them with `None`. If the `chain` switch is set, only a list of related logger-object keys is returned.

The log entry structure is apparent based on line 15 of the previous listing:

```
15    log.append((key, call, args, kwargs, result, affected_objects, entry, call_type))
```

key – `_log_key` of the `origin` or `___call` in case of a function call.

call – Name of the called method.

args – Arguments given to the called method if any.

kwargs – Keyword arguments given to the called method if any.

result – Value returned by the call if any.

affected_objects – This is an array containing keys of all objects relevant to the call. The premise is that all objects the method worked with are in `args` and `kwargs`. So these are searched in-depth to extract the relevant keys.

entry – A snapshot of the `state` at the moment of the call. All active objects are converted using their `__log_repr__` method.

call_type – `call` when the execution is started, `return` when the execution is finished.

The premise is that the arguments and result are references to actual objects where possible so that the log can reflect their relevance in each call. Argument objects and the result object can then be highlighted. Since the objects get replaced by their keys in the log entries, there may be a clash when the keys are generic. If a logger function returns `"A"` and an object has log key `"A"`, it gets highlighted. It is desirable to make the keys unique to avoid such scenarios.

Logging the calls

The function decorator gets generated by the following function.

Listing 3.6: Decorator generator

```
def get_callable_attr_decorator(instance, attr_name):
    def decorator(f):
        def wrapper(*args, **kwargs):
            _log_call(instance, attr_name, args, kwargs, call_type='call')
            result = f(*args, **kwargs)
            _log_call(instance, attr_name, args, kwargs, result, call_type='return')
            return result
        return wrapper
    return decorator
```

A call entry is created, then the function is executed, a return entry is created and the value is returned. Note that we can pass this generator directly as the `get_callable_decorator` argument in the class decorator described in the previous section. The function log decorator can then be implemented by calling the generator with `___call` as instance and the function name as the attribute:

Listing 3.7: Transform function into logger

```
def logged_function(f):
    """Decorator for a function to log every call"""
    return get_callable_attr_decorator('___call', f.__name__)(f)
```

`with` statement

The `with` statement in Python is useful when working with resources that need to be opened and closed. For example

```
with open('input.txt', 'r') as datafile:
    data = datafile.read()
    # process data
# file is closed, program continues
...
```

We can work with the resources inside the `with` statement code block and when they are not needed anymore, they are automatically closed. Internally, the `with` statement initializes the resources and then executes their `__enter__()` method. Their `__exit__()` methods are executed at the end of the code block. Any class implementing these two methods can act as such a resource.

This provides a handy object deletion. When visualizing, there may be temporary objects whose visual representation is needed only a few steps and then they are not relevant anymore. Python has a garbage collector and doesn't explicitly delete unreferenced objects. Therefore, the `__exit__` function provides a convenient way to remove temporary objects from the visualization.

Logger class

Finally, we describe the function that turns a class object into a logger.

Listing 3.8: Transform class into logger

```
def transform_into_logger(cls, repr_fn=None, key_fn=None, excluded=[]):
    """Transforms a class into a logger that logs
    every attribute call inside a with statement"""
    log_key_fn = key_fn if key_fn is not None else repr
    decorated_class = decorate_attributes_parametrized_immutable(
        cls,
        get_callable_attr_decorator,
        exclusion=[
            '__init__',
            '__enter__',
            '__exit__',
            '__log_repr__',
            '_log_key'
        ] + excluded
    )

    class LoggedClass(decorated_class):
        def __init__(self, *args, **kwargs):
            super().__init__(*args, **kwargs)
            enter_on_init = kwargs.pop('_enter_on_init', False)
            custom_key = kwargs.pop('_log_key', None)
            self._log_key = custom_key if custom_key is not None else log_key_fn(self)
            if enter_on_init:
                self.__enter__()

        def __log_repr__(self):
            return (repr_fn if repr_fn is not None else repr)(self)

        def __enter__(self, *args, **kwargs):
            _log_enter(self)
            return self

        def __exit__(self, *args, **kwargs):
            _log_exit(self)
            return None

    return LoggedClass
```

The arguments required for this process are the target class, `repr_fn` and `key_fn` functions and exclusions if any.

key_fn – Function that transforms the logger class instance into a unique key. Defaults to `repr`

repr_fn – Function that transforms the logger class instance into an immutable representation. Defaults to `repr`

The transform function doesn't mutate the given class, only returns its Logger copy. First, the class methods are decorated turning them into logger functions excluding the methods necessary for the logging process. Then a `LoggerClass` class object is created, inheriting all methods from given class and implementing mentioned necessary methods.

`_log_enter` and `_log_exit` only call `_log_call` with a special `___enter` and `___exit` call name respectively. When initializing the new class, there is an option to make it enter the log state straight away. There is also an option to provide a custom log key and skip the `key_fn` call.

3.2.4 Usage

Using these tools it is possible to create logging copies of any class object.

Listing 3.9: Logger examples

```
from logger import transform_into_logger, logged_function
import skgeom as sg

Point2V = transform_into_logger(
    sg.Point2,
    repr_fn=lambda p: ('point', [p.x(), p.y()]),
    excluded=['x', 'y']
)

Line2V = transform_into_logger(sg.Line2)

Segment2V = transform_into_logger(
    sg.Segment2,
    repr_fn=lambda s: ('segment', [
        [s.source().x(), s.source().y()],
        [s.target().x(), s.target().y()]
    ]),
    excluded=['source', 'target'],
)

Polygon2V = transform_into_logger(
    sg.Polygon,
    repr_fn=lambda p: ('polygon', [[x, y] for x,y in p.coords])
)

@logged_function
def compare_lengths(l, r):
    if l.squared_length() > r.squared_length():
        return r
    return l
```

Some other examples will follow in the next sections describing the visualization.

3.3 Wizer

In this section we will describe the visualizer that shows each step of an algorithm in a graphical user interface. For each log entry, relevant information about the call is displayed and objects active at the time of the call are drawn. Additionally, different colours are used based on the object's relevance to the call.

3.3.1 Drawing

The visualizer is implemented using the built-in `tkinter` module. By default, `tkinter` doesn't support transparent shapes, but in combination with the `PIL` module, it is possible to create a transparent image with the desired shape and then draw it as an image in `tkinter`. This is important when displaying the interiors of objects and overlapping objects.

The usage example in the previous section shows examples of representation functions. For the purpose of visualization the only information needed is the shape and coordinates of any given object. The objects are drawn based on this representation. Every object representation is a tuple consisting of the shape type and an array of vertices. The supported shape types include `Point`, `Line`, `Segment` and `Polygon`. The vertex coordinates define each of them and the colour can be inferred from the additional information in the entry.

Relevant objects are extracted from the call arguments. In the logging phase, `args` and `kwargs` are searched in-depth, lists and dictionaries recursively. Each object with a `_log_key` attribute is replaced by its log key and all such keys are recorded separately as well. On every `call` type entry the affected objects are orange to represent that it is unknown what will happen to them or which one is returned. On every `return` type entry the arguments connected to call's return value are green and other affected objects are red. Irrelevant active objects are black.

The white canvas represents the plane. The vertex coordinates are adjusted based on the zoom level and the position of the 'camera', representing the coordinates of the center of the screen. They are also modified to be consistent with the `tkinter` coordinate system. Point `(0,0)` is in the upper-left corner and the y coordinates increase downwards. But in the algorithms, y coordinate increases upwards and it is more intuitive when the `(0,0)` point is in the lower-left corner.

Lastly, the shapes are drawn in separate images and displayed on the tkinter canvas.

3.3.2 Call information

The call information, such as the called attribute's name, arguments and result is always printed in separate lines in the upper-left corner. First, the representation of the origin object is printed, then the call name. Arguments, keyword arguments and the result are printed optionally in following lines if they exist.

3.3.3 Controls

Some basic intuitive form of control over the canvas is implemented. Left and right arrows switch the log entry number, up and down arrows zoom in and out. Dragging

across the screen changes the coordinates of the center of the screen to create the illusion that the mouse moves the canvas. The images are re-drawn only on zoom and log entry switch as it is possible to change the coordinates at which the images are displayed.

3.4 Usage

At this time, Wizer is a minimal working prototype. The latest version of the code and updates can be found at https://github.com/bohdanator/wizer. This tool has the potential to be able to visualize any computational geometry algorithm on a two-dimensional plane. It works best if every object and every function is a logger and the log keys are unique. The actual log with the algorithm data is stored in the `log` variable inside the `logger` module.

3.5 Future work

There is a lot of space for improvement in the visualization. The logging base can be used to display virtually any type of algorithm with the correct visualizer. Since the way the objects are represented in the log can be customized, it can be used in combination with any visualizer that can parse and display the log accordingly.

The logging also offers opportunities for extension. Currently, every action gets logged to the same array. It may be possible to enhance the log entry structure, but timestamps may be an issue.

Lastly, the class utils implemented for logging purposes can be utilized in other projects as well. Both the logging and the class decorator subroutines could be optimized and exported as separate packages.

Conclusion

Dividing polygons into triangles is one of the fundamental tasks in computational geometry. It has numerous applications ranging from designing security camera layouts to pattern recognition. It is a building block for multiple algorithms, for example tiling algorithms.

We explained four triangulation algorithms using the Forišek-Steinová methodology for the purpose of teaching and popularization of computational geometry. We provided an overview of the research in the problem of polygon triangulation throughout history and described some of the faster solutions.

Additionally, we implemented a visualization tool providing the means to visually analyze geometric algorithms. It consists of two parts. The first part logs every action of the algorithm at runtime. The second part, Wizer, displays each step of a computational geometry algorithm on a plane. It can be used to present how algorithms work or why they don't work, to learn new algorithms or to find mistakes and bugs. It provides decorators to easily turn objects into loggers that create a record of their every method call during the execution of an algorithm. This log can be consequently parsed and visualized. The visualizer and the form of the log are customizable to make vizualization of other types of algorithms possible.

Our hope is that this thesis will be a meaninful contribution to the academic field of computational geometry.

Bibliography

[1] J. A. Sellares and G. Toussaint. On the role of kinesthetic thinking in computational geometry. *International Journal of Mathematical Education in Science and Technology.*, 34(2):219–237, 2003.

[2] Lakoff G. and Johnson M. *Metaphors We Live By.* University of Chicago press, 2003.

[3] M. Forišek and M. Steinová. *Explaining algorithms using metaphors.* Springer, 2003.

[4] Lennes N. J. Theorems on the simple finite polygon and polyhedron. *American Journal of Mathematics*, 33:37–62, 1911.

[5] Meisters G. H. Polygons have ears. *Amer. Math. Monthly*, June/July:648–651, 1975.

[6] ElGindy H., Tarjan R. E, and Everett H. Slicing an ear using prune-and-search. *Pattern Recognition Letters*, 14(9):719–722, 1993.

[7] Toussaint G. Efficient triangulation of simple polygons. *Visual Computer*, 7, 1991.

[8] Garey M. R. et. al. Triangulating a simple polygon. *Information Processing Letters*, 7(4), 1978.

[9] Chazelle B. Triangulating a simple polygon in linear time. *Proc. 31st Annual Symposium on Foundations of Computer Science*, 1990.

[10] Open Source. Algorithm visualizer. https://github.com/algorithm-visualizer/algorithm-visualizer, retrieved May 2, 2022.

[11] The CGAL Project. Computational geometry algorithm library. https://www.cgal.org/. retrieved December 4, 2021.

[12] Wolf Vollprecht. scikit-geometry. https://scikit-geometry.github.io/scikit-geometry/index.html, 2019. retrieved Feb 2, 2022.

Appendix A

Wizer source code

The attachment contains the source code of the minimal working prototype of the Wizer environment. An example triangulation algorithm is included and visualized. To successfully run the example, `scikit-geometry` Python package needs to be installed, but screenshots of the results are also included.

The latest version of the environment and updates are published at https://github.com/bohdanator/wizer.