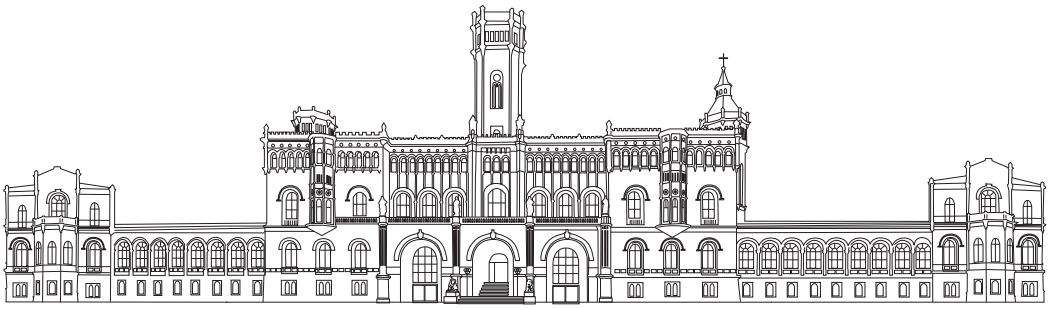


LEIBNIZ UNIVERSITÄT HANNOVER

FAKULTÄT FÜR ELEKTROTECHNIK UND INFORMATIK
INSTITUT FÜR THEORETISCHE INFORMATIK

Algorithmen zur Polygontriangulierung

Bachelorarbeit

eingereicht von

JAKOB LENNART WEGE

am 6. September 2021

Erstprüfer	:	Prof. Dr. Heribert Vollmer
Zweitprüfer	:	PD Dr. Arne Meier
Betreuerin	:	M.Sc. Sabrina Gaube

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

LISTINGS

1 EINLEITUNG

1.1 MOTIVATION

Das Zerlegen von komplexen Strukturen in einfachere Teilstrukturen ist ein gängiges Konzept in der Mathematik und Informatik. In der algorithmischen Geometrie sind Polygone eine der häufigsten komplexen Strukturen. Die einfachste zweidimensionale Fläche ist ein Dreieck. Daher liegt es nahe, Polygone in Dreiecke zu zerlegen, was als Triangulierung bezeichnet wird. Das Finden und Implementieren von Algorithmen zur Polygontriangulierung ist also von Interesse für die algorithmische Geometrie. In dieser Arbeit stelle ich einige Algorithmen zur Polygontriangulierung vor, vergleiche sie, und implementiere sie in Python.

1.2 AUFBAU DER ARBEIT

In Kapitel 2 definiere und veranschauliche ich einige Begriffe aus der Geometrie, die für die folgenden Kapitel relevant sind. In Kapitel 3 präsentiere ich Resultate und Datenstrukturen, die für die Implementation von Triangulierungsalgorithmen relevant sind. In Kapitel 4 stelle ich dann vier solche Algorithmen vor. Ich erkläre die Algorithmen und stelle sie als Pseudocode dar. Außerdem gehe ich auf Implementationsdetails ein und analysiere die Laufzeitkomplexität. In Kapitel 5 vergleiche ich die Algorithmen und ziehe ein Fazit. Im Anhang stelle ich meine Implementation der Algorithmen in Python dar.

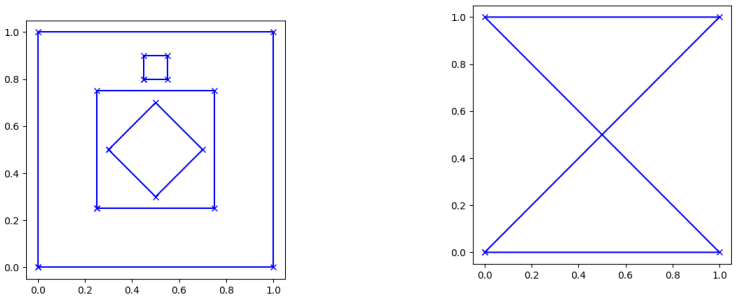
2 DEFINITIONEN

In diesem Kapitel werden grundlegende Definitionen zu Polygonen, Arten von Polygonen und Triangulierungen gegeben.

Nach [7] definiere ich ein Polygon formell folgendermaßen:

Definition 1 Ein *Polygon* P ist ein 2-regulärer planarer geradliniger Graph, welcher eine beschränkte Fläche ϕ (genannt das *Innere* von P) enthält, deren Seiten genau alle Kanten von P sind.

Dabei besteht ein planarer geradliniger Graph aus endlich vielen geschlossenen Liniensegmenten in der euklidischen Ebene, die bis auf Endpunkte disjunkt sind. Die Endpunkte bilden die Knoten und die Segmente die Kanten eines in die Ebene eingebetteten Graphen. 2-Regularität bedeutet, dass von jedem Knoten genau zwei Kanten ausgehen. Die Zusammenhangskomponenten werden als *Polygonketten* bezeichnet. Durch diese Definition werden Polygone mit überschlagenen Kanten ausgeschlossen. Polygone können jedoch Löcher haben.



(a) ein Polygon aus vier Polygonketten

(b) kein Polygon, da sich Kanten schneiden

Abbildung 2.1: gültiges und ungültiges Polygon

Ebenfalls nach [7] ist dann eine Triangulierung eines Polygons folgendermaßen definiert:

Definition 2 Eine *Triangulierung* eines Polygons P ist ein planarer geradliniger Graph, der aus den Kanten von P sowie zusätzlichen Segmenten innerhalb von P sowie genau den Ecken von P besteht und der das Innere von P in dreieckige Flächen zerlegt.

Eine solche Triangulierung existiert für jedes Polygon[7].
Bestimmte Klassen von Polygonen lassen sich einfacher triangulieren

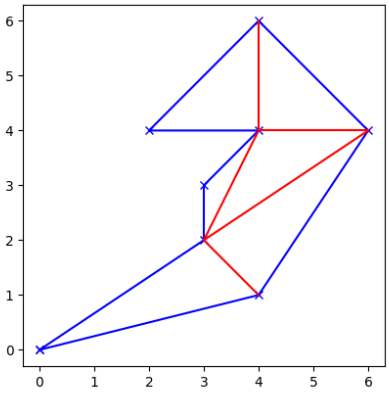
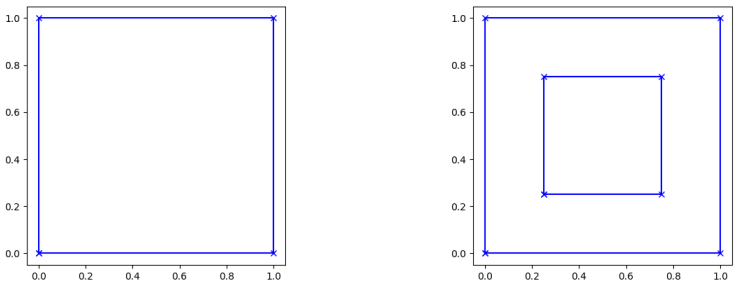


Abbildung 2.2: Ein Polygon (blau) mit einer möglichen Triangulierung (rot)

als andere. Im Folgenden werden die Klassen definiert, die in dieser Arbeit verwendet werden.

Definition 3 Ein *einfaches Polygon* ist ein Polygon ohne Löcher, d.h. ein Polygon, welches durch einen zusammenhängenden Graphen beschrieben wird.

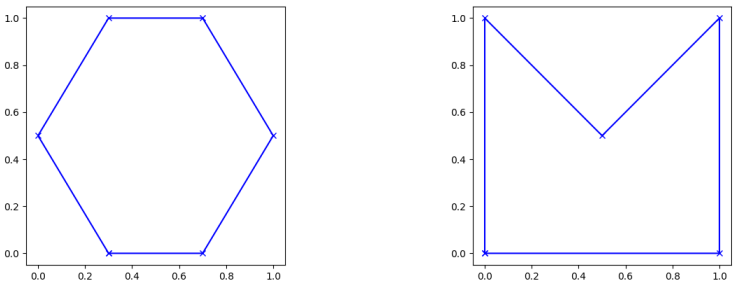


(a) ein einfaches Polygon (b) ein nicht einfaches Polygon

Abbildung 2.3: Unterschied zwischen einfachen und nicht einfachen Polygonen

Definition 4 Ein *konvexes Polygon* ist ein Polygon P, bei dem jedes Liniensegment, dessen Enden innerhalb von P liegen, vollständig innerhalb von P liegt. Dies ist äquivalent dazu, dass alle Innenwinkel kleiner oder gleich $180°$ sind. Nicht konvexe Polygone heißen *konkav*.

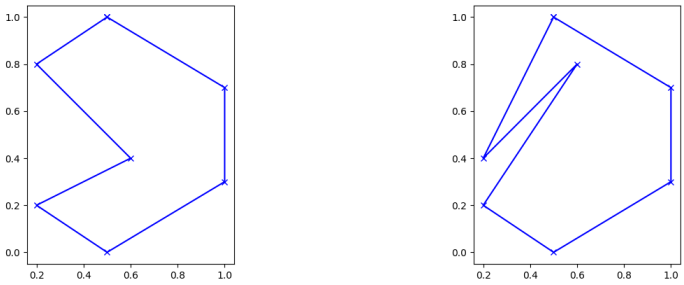
Definition 5 Ein *monotones Polygon* ist ein Polygon P, zu welchem eine Gerade g existiert, sodass jede zu g orthogonale Gerade höchstens 2 Kanten von P schneidet.



(a) ein konvexes Polygon (b) ein konkaves Polygon

Abbildung 2.4: Unterschied zwischen konvexen und konkaven Polygonen

Für die Gerade g wird in den Algorithmen typischerweise die y-Achse gewählt. Das hat zur Folge, dass sich das Polygon in zwei Ketten zerlegen lässt, in denen die y-Koordinaten der Eckpunkte entlang der Kette je nach Richtung monoton steigen bzw. fallen.

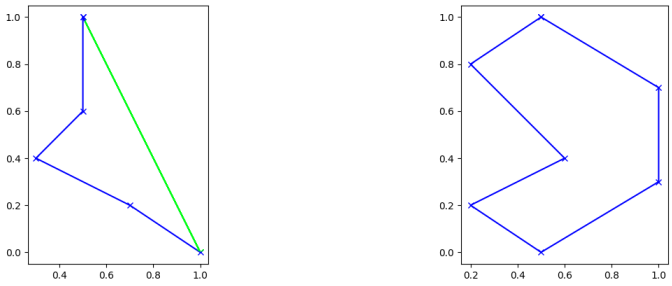


(a) ein monotones Polygon (b) ein nicht monotones Polygon, da von der obersten Ecke aus entlang der linken Kante die y-Koordinate erst fällt und dann steigt

Abbildung 2.5: Unterschied zwischen monotonen und nicht monotonen Polygonen

Definition 6 Ein *unimonotones Polygon* ist ein monotones Polygon, bei dem alle Eckpunkte entlang des Randes beginnend bei der Ecke mit minimaler y-Koordinate monoton steigende Koordinaten haben. [8]

Dies bedeutet, dass das Polygon aus einer Kette von Punkten mit zunehmender Höhe besteht, die auf einer Seite dann mit einer einzelnen Kante geschlossen wird. Die Kante, die die oberste und unterste Ecke verbindet, wird als *Hauptkante* bezeichnet.



(a) ein unimonotones Polygon mit grün markierter Hauptkante

(b) ein nicht unimonotones Polygon

Abbildung 2.6: Unterschied zwischen unimonotonen und nicht unimonotonen Polygonen

3 ALGORITHMISCHE GRUNDLAGEN

3.1 GRUNDLAGEN

Einige der beschriebenen Algorithmen wurden von mir in Python implementiert. Python wurde aufgrund der vorhandenen Bibliotheken für mathematische Zwecke, insbesondere zur graphischen Darstellung, gewählt. Außerdem erleichtern die dynamischen Typeneigenschaften von Python eine flexible Gestaltung der Datenstrukturen, da verschiedene Algorithmen unterschiedliche Annahmen voraussetzen, z.B. bestimmte Umlaufrichtungen von Polygonketten. Auf diese wird bei der Analyse der Algorithmen genauer eingegangen.

3.2 PUNKTE UND KANTEN

Jeder Eckpunkt des Polygons ist ein Objekt, welches die kartesischen Koordinaten als wichtigste Attribute enthält. Darüber hinaus kennt ein Punkt seine Nachbarn im Polygon sowie jegliche Punkte, mit denen er in einer Triangulierung verbunden ist. Die Kenntnis der Nachbarn eines Punktes erlaubt es, die Knoten in einer Polygonkette als doppelt verlinkte Liste zu behandeln und lokale Eigenschaften der Eckpunkte des Polygons zu betrachten. Es werden keine Punkte als Objekte betrachtet, die nicht Eckpunkte des Polygons sind.
Wenn die Kanten des Polygons als Objekte benötigt werden, werden sie als Tupel zweier Eckpunkte verarbeitet.

3.3 POLYGONE

Ein Polygon besteht aus einer oder mehreren Polygonketten. Jede dieser Ketten besteht aus einer geordneten Liste von Eckpunkten, die wie in Abschnitt 3.2 beschrieben aufgebaut sind. Beim Aufbau der Liste wird außerdem festgestellt und gespeichert, ob die Eckpunkte im oder gegen den Uhrzeigersinn angegeben sind.

3.4 KOMPLEXITÄTSBETRACHTUNGEN

Bei der Analyse der Algorithmen wird das Vorhandensein des Polygons im Speicher vorausgesetzt. Die Komplexität des Aufbaus der Strukturen wird daher ignoriert. Da jeder Eckpunkt lediglich zu einer Liste hinzugefügt und mit zwei anderen Eckpunkten verbunden werden muss und Python-Listen konstante Laufzeit für die Append- und Get-

Operationen bieten [17], kann jedoch von einer linearen Laufzeit ausgegangen werden.
Für Zugriffsoperationen auf Objektattribute wird konstante Laufzeit angenommen.

3.5 LAUFRICHTUNG

Einige Algorithmen setzen eine bestimmte Laufrichtung (im oder gegen den Uhrzeigersinn) des Polygons voraus. Dies ist zwar ein Detail der Datenstruktur, welches der Algorithmus voraussetzen kann. Es ist aber dennoch nützlich, die Laufrichtung bestimmen zu können, um für einfache Polygone dem Nutzer Spielraum bei der Eingabe zu bieten. Dazu lässt sich die Gaußsche Trapezformel verwenden. Dies ist eine Formel, die es erlaubt, die Fläche eines Polygons aus den Koordinaten der Eckpunkte zu berechnen [3]. Dazu werden Determinanten von 2x2-Matrizen aus den Koordinaten von aufeinanderfolgenden Eckpunkten wie folgt aufsummiert, wobei $(x_0, y_0), \dots, (x_{n-1}, y_{n-1})$ die Eckpunkte des Polygons in einer Umlaufreihenfolge sind:

$$\begin{aligned} A &= \frac{1}{2}\left(\begin{vmatrix} x_0 & x_1 \\ y_0 & y_1 \end{vmatrix} + \begin{vmatrix} x_1 & x_2 \\ y_1 & y_2 \end{vmatrix} + \dots + \begin{vmatrix} x_{n-1} & x_0 \\ y_{n-1} & y_0 \end{vmatrix}\right) \\ &= \frac{1}{2}\left(x_0y_1 - x_1y_0 + x_1y_2 - x_2y_1 + \dots + x_{n-1}y_0 - x_0y_{n-1}\right) \end{aligned}$$

In [3] wird gezeigt, dass das Ergebnis positiv ist, wenn die Eckpunkte gegen den Uhrzeigersinn angegeben werden, und negativ, wenn sie im Uhrzeigersinn angegeben werden. Das Vorzeichen von A kann also genutzt werden, um die Laufrichtung zu bestimmen. Der Faktor $\frac{1}{2}$ kann hierbei weggelassen werden, da er das Vorzeichen nicht verändert. Tatsächlich nutze ich eine alternative Form der Formel, die mehr Additionen und weniger Multiplikationen verwendet und deren Äquivalenz leicht zu zeigen ist:

$$\begin{aligned} 2A &= (x_0 - x_1)\cdot(y_0 + y_1) + (x_1 - x_2)\cdot(y_1 + y_2) \\ &\quad + \dots + (x_{n-1} - x_0)\cdot(y_{n-1} + y_0) \\ &= (x_0y_0 + x_0y_1 - x_1y_0 - x_1y_1) + (x_1y_1 + x_1y_2 - x_2y_1 - x_2y_2) \\ &\quad + \dots + (x_{n-1}y_{n-1} + x_{n-1}y_0 - x_0y_{n-1} - x_0y_0) \\ &= x_0y_1 - x_1y_0 + (x_1y_1 - x_1y_1) + x_1y_2 - x_2y_1 + (x_2y_2 - x_2y_2) \\ &\quad + \dots + x_{n-1}y_0 - x_0y_{n-1} + (x_0y_0 - x_0y_0) \\ &= x_0y_1 - x_1y_0 + x_1y_2 - x_2y_1 + \dots + x_{n-1}y_0 - x_0y_{n-1} \end{aligned}$$

Die Laufrichtung lässt sich also mit Algorithmus 1 bestimmen.

3.6 GLEICHE Y-KOORDINATEN

Die Algorithmen in den Abschnitten 4.4 und 4.5 setzen als Annahme voraus, dass alle Eckpunkte verschiedene y-Koordinaten haben. Die-

Algorithmus 1 Laufrichtung

```
1: function WINDING(Liste ℓ der Eckpunkte (x_i, y_i) in einer Polygonkette)
2:     sum ← 0
3:     n ← Länge von ℓ
4:     for i = 0, ..., n − 1 do
5:         sum ← sum + (x_i − x_{i+1 mod n}) · (y_i + y_{i+1 mod n})
6:     if sum > 0 then
7:         return counterclockwise
8:     else
9:         return clockwise
```

se Annahme ließe sich zwar durch zusätzliche lexikographische Ordnung umgehen. Dies erschwert jedoch die Formulierung und Implementation der Algorithmen erheblich. Daher nutze ich die Tatsache, dass sich jedes Polygon so rotieren lässt, dass alle Eckpunkte verschiedene y-Koordinaten haben. Dies ist der Fall, da es nur endlich viele Geraden gibt, die durch Paare von Eckpunkten des Polygons verlaufen, und damit nur endlich viele Rotationen, bei denen mindestens eine dieser Geraden parallel zur x-Achse ist. Vor der Ausführung dieser Algorithmen wird das Polygon also vorverarbeitet, indem eine Rotation um den Ursprung mit einzigartigen y-Koordinaten der Ecken gefunden wird. Die ursprünglichen Koordinaten werden gespeichert, damit das ursprüngliche Polygon nach Beendigung des Algorithmus wiederhergestellt werden kann.

3.7 TRAPEZOIDISIERUNG

Mehrere Algorithmen ermitteln die Triangulierung über den Zwischenschritt der Trapezoidisierung. Das bedeutet, dass das zu triangulierende Polygon zunächst in Trapeze zerlegt wird und diese Trapezoidisierung dann im zweiten Schritt in eine Triangulierung umgewandelt wird. Die parallelen Kanten der Trapeze sind dabei auch alle parallel zueinander und zur x-Achse. Die Trapezoidisierung und ein Algorithmus zu ihrer Triangulierung wird in [8] beschrieben. Die Trapeze weichen leicht von der üblichen Definition ab:

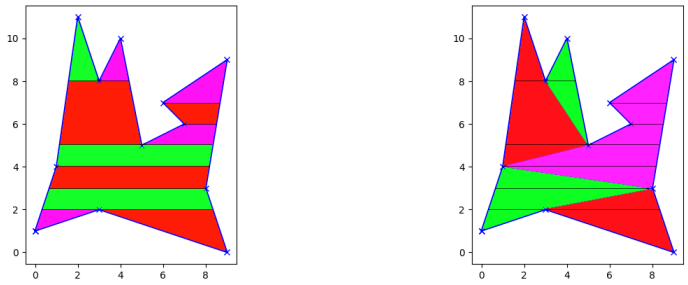
Definition 7 Ein *Trapez* ist eine Teilfläche der Ebene oder insbesondere eines Polygons, die oben und unten durch zur x-Achse parallelen Geraden und links und rechts durch Segmente begrenzt ist.

Ein Trapez kann somit vollständig durch die zwei y-Koordinaten der horizontalen Begrenzungen oder durch zwei Punkte, die jeweils auf diesen Begrenzungen liegen, und durch die zwei seitlich begrenzenden Strecken beschrieben werden.
Dabei sind einige Besonderheiten zu beachten. Die seitlich begrenzen-

den Segmente sind in den Algorithmen stets Kanten des zu triangulierenden Polygons. Falls diese Kanten einen Eckpunkt gemeinsam haben, verläuft eine der begrenzenden Geraden durch diesen. Das Trapez fällt in diesem Fall geometrisch zu einem Dreieck zusammen, kann aber immer noch in der oben genannten Art beschrieben werden.
Um das Polygon zu triangulieren, lässt sich nach [8] die Trapezoidisierung in eine Unimonotonisierung, also eine Zerlegung in unimonotone Polygone (siehe Definition 6) überführen. Dazu werden die Diagonalen der Trapeze verwendet. Damit ist hier folgendes gemeint:

Definition 8 Die *Diagonale* eines Trapezes T in einer Trapezoidisierung ist die Verbindung zwischen den beiden Punkten, deren y-Koordinaten die obere und untere Begrenzung des Trapezes definieren. Ist diese Verbindung bereits eine Kante des Polygons, heißt T ein *Trapez der Klasse A*. Ist sie keine Kante des Polygons, heißt T ein *Trapez der Klasse B*.

Die Diagonalen der Trapeze der Klasse B werden benutzt, um das Polygon in unimonotone Polygone zu zerlegen. Diese Diagonalen sind dabei bereits innere Kanten der endgültigen Triangulierung. Fournier und Montuno verwenden hierzu in [8] einen rekursiven Algorithmus. Für meine Implementation des Algorithmus aus [8] habe ich jedoch eine modifizierte Version des Algorithmus nach Adi in [1] verwendet, welche die unimonotonen Polygone iterativ während der Erstellung der Trapezoidisierung aufbaut.
Die unimonotonen Polygone lassen sich in Linearzeit triangulieren, was in 4.4.6 beschrieben wird.
Eine solche Trapezoidisierung sowie die entsprechende Unimonotonisierung sind in Abbildung 3.1 zu sehen.



(a) Ein trapezoidisiertes Polygon mit farblich markierten Trapezen

(b) Die Unimonotonisierung, die durch das Einzeichnen der Diagonalen entsteht, mit farblich markierten UMPs

Abbildung 3.1: Trapezoidisierung und Unimonotonisierung eines Polygons

4 ALGORITHMEN

4.1 OPTIMALE LAUFZEIT

Da jede Ecke eines Polygons mit n Ecken Teil mindestens eines Dreiecks in der Triangulierung ist, muss jeder Triangulierungsalgorithmus jede Ecke mindestens einmal besuchen. Damit kann kein Triangulierungsalgorithmus eine bessere Laufzeitkomplexität als $\mathcal{O}(n)$ haben.

4.2 FÄCHERTRIANGULIERUNG

Einer der einfachsten Triangulierungsalgorithmen ist die Fächertriangulierung (fan triangulation) [16]. Damit dieser Algorithmus anwendbar ist, muss es eine Ecke geben, sodass jede von dieser Ecke ausgehende Diagonale innerhalb des Polygons liegt. Dies ist insbesondere bei konvexen Polygonen nach Definition 4 offensichtlich für jede Ecke der Fall. Der Algorithmus 2 wählt dann eine beliebige Ecke und verbindet diese mit jeder anderen Ecke, wodurch das Polygon fächerartig aufgeteilt wird, wie in Abbildung 4.1 zu sehen ist.

Algorithmus 2 Fächertriangulierung

```
1: function TRIANGULATE(p)
2:     wähle eine Ecke v von p
3:     for all Ecke w ∈ p do
4:         if w ≠ v then
5:             füge Kante (v, w) zur Triangulierung hinzu
```

4.2.1 *Laufzeitanalyse*

Das Wählen einer Ecke ist in konstanter Zeit möglich. In meiner Implementation wird das erste Element der Eckenliste gewählt. Die Laufzeit des Algorithmus ist also von der for-Schleife dominiert, welche alle Ecken des Polygons einmal durchläuft und für jede höchstens eine Kante hinzufügt. Für ein Polygon mit n Ecken hat der Algorithmus damit eine Laufzeitkomplexität von $\mathcal{O}(n)$.
Die Einschränkung auf konvexe Polygone ist jedoch signifikant. Daher sollen im Folgenden weitere Algorithmen betrachtet werden, die auch für konkave Polygone funktionieren.

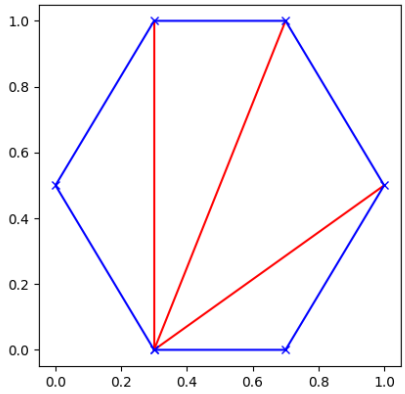


Abbildung 4.1: Ein Polygon, das mit dem Fächertriangulierungsalgorithmus trianguliert wurde. Die von einer Ecke zu allen anderen Ecken ausgehenden Kanten sind leicht zu erkennen.

4.3 TRIANGULIERUNG DURCH EAR CLIPPING

Die Fächertriangulierung ist zwar sehr simpel und laufzeitoptimal, funktioniert aber nur für sehr bestimmte Polygone. Ein weiterer recht einfacher Algorithmus, der aber für beliebige einfache Polygone funktioniert, ist die Ear-Clipping-Methode. Dazu muss zunächst ein neuer Begriff eingeführt werden:

Definition 9 Ein *Ohr* eines Polygons ist eine Ecke des Polygons, deren Nachbarn so liegen, dass die Verbindungsstrecke zwischen ihnen nur durch das Innere des Polygons verläuft.

Wie in [12] als *Zwei-Ohren-Theorem* gezeigt, hat jedes einfache Polygon mit mehr als drei Ecken mindestens zwei solche Ohren.
Ohren bieten sich unmittelbar als Triangulierungsmethode an. Die algorithmischen Details basieren hier auf [6]. Die Strecke, die die Nachbarn des Ohres verbindet, teilt das Polygon in ein Dreieck im Sinne der Triangulierung und ein Restpolygon, welches wieder einfach ist und daher bei ausreichender Eckenanzahl nach dem Zwei-Ohren-Theorem wieder Ohren besitzt. Ein einfaches Polygon lässt sich also triangulieren, indem schrittweise Ohren gefunden und abgeschnitten werden, bis das restliche Polygon selbst ein Dreieck ist. Dies wird in Algorithmus 3 beschrieben.

4.3.1 *Erkennen von Ohren*

Um zu erkennen, ob eine Ecke ein Ohr ist, muss man nur überprüfen, ob ein Eckpunkt des Polygons innerhalb des von der Ecke und ihren Nachbarn aufgespannten Dreiecks liegt. Ist dies nicht der Fall, kann die Verbindung der Nachbarn das Polygon nicht verlassen und es handelt

Algorithmus 3 Ear-Clipping-Methode

```
1: function TRIANGULATE(p)
2:     while p hat mehr als 3 Ecken do
3:         finde ein Ohr v von p
4:         w, x ← Nachbarn von v
5:         füge Kante w, x zur Triangulierung hinzu
6:         entferne v aus p
```

sich um ein Ohr. Nach [6] reicht es dabei aus, die Ecken mit Innenwinkel größer als 180° auf Lage innerhalb des Dreiecks zu überprüfen.

4.3.2 *Implementation und Laufzeitanalyse*

Nach [6] ist es möglich, diesen Algorithmus für ein Polygon mit n Ecken in $\mathcal{O}(n^2)$ durchzuführen. Dazu müssen die Ecken in einer doppelt verlinkten Liste gespeichert sein, sodass das Entfernen der Ohren in konstanter Zeit möglich ist. In meiner Implementation befinden sich die Ecken bereits in einer linearen Liste in der Polygonkettenstruktur und kennen ihre Nachbarn. Um die ursprüngliche Struktur des Polygons zu erhalten, werden zusätzliche Attribute für den Vorgänger und Nachfolger in der Liste für diesen Algorithmus eingeführt. Außerdem werden die Ohren einmal am Anfang des Algorithmus bestimmt, sodass beim Entfernen eines Ohres nur die angrenzenden Ecken darauf überprüft werden müssen, ob sie Ohren sind. Die noch vorhandenen Ohren werden ebenfalls in einer Liste gespeichert.
In der tatsächlichen Implementation werden zunächst alle Ohren bestimmt. Dafür müssen für jede der n Ecken die $\mathcal{O}(n)$ Ecken mit überstumpfen Innenwinkeln auf Lage im vom Ohr aufgespannten Dreieck geprüft werden. Dieser Schritt braucht also $\mathcal{O}(n^2)$ Zeit. Danach wird die while-Schleife durchlaufen. Da in jedem Schritt der while-Schleife eine Ecke entfernt wird, hat die Schleife $\mathcal{O}(n)$ Durchläufe. Das Finden eines Ohres und seiner Nachbarn ist aufgrund der oben beschriebenen Listen in konstanter Zeit möglich. Nach dem Entfernen des Ohres müssen die Nachbarn darauf geprüft werden, ob sie Ohren sind, was Vergleiche mit den $\mathcal{O}(n)$ Ecken mit überstumpfen Innenwinkeln benötigt und daher $\mathcal{O}(n)$ dauert. Damit läuft der gesamte Algorithmus in $\mathcal{O}(n^2)$.

4.4 ALGORITHMUS NACH FOURNIER UND MONTUNO

4.4.1 *Grundidee*

Dieser in [8] vorgestellte Algorithmus erzeugt eine Trapezoidisierung, wie in Abschnitt 3.7 beschrieben. Dazu werden die Eckpunkte nach y-Koordinate sortiert und von oben nach unten abgearbeitet. Je nach Typ

des Eckpunkts werden bestehende Trapeze erweitert oder abgeschlossen oder neue Trapeze erstellt. Sobald ein Trapez abgeschlossen ist, wird es zur Unimonotonisierung hinzugefügt. Der Algorithmus geht davon aus, dass die Eckpunkte im Uhrzeigersinn gegeben sind. Die Ecken werden in [8] in verschiedene Typen klassifiziert, welche von der Position der Kanten in Relation zu der horizontalen Linie durch die Ecke abhängen, wobei [1] die Typen II und III noch unterteilt:

Definition 10 Eine *Ecke vom Typ I* oder *reguläre Ecke* ist eine Ecke, deren angrenzende Kanten auf verschiedenen Seiten der horizontalen Linie durch die Ecke liegen.

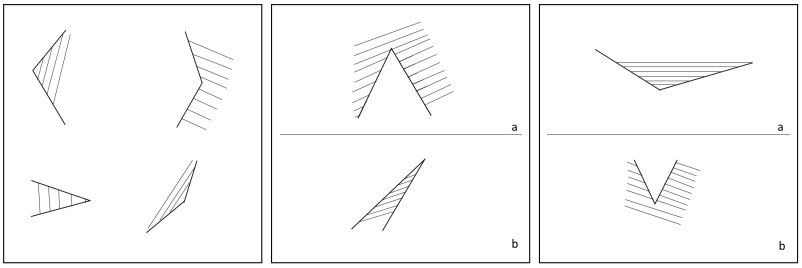
In der Trapezoidisierung liegen diese Ecken auf der Grenze zwischen zwei Trapezen.

Definition 11 Eine *Ecke vom Typ II* oder *stalagmitische Ecke* ist eine Ecke, deren angrenzende Kanten beide unter der horizontalen Linie durch die Ecke liegen. Liegt das Innere des Polygons über der Ecke, heißt sie *vom Typ IIa*. Liegt das Innere des Polygons unter der Ecke, heißt sie *vom Typ IIb*.

In der Trapezoidisierung bilden Ecken vom Typ IIa die Grenze zwischen einem Trapez oberhalb und zwei Trapezen unterhalb. Ecken vom Typ IIb bilden die obere Grenze eines Trapezes.

Definition 12 Eine *Ecke vom Typ III* oder *stalagtitische Ecke* ist eine Ecke, deren angrenzende Kanten beider über der horizontalen Linie durch die Ecke liegen. Liegt das Innere des Polygons über der Ecke, heißt sie *vom Typ IIIa*. Liegt das Innere des Polygons unter der Ecke, heißt sie *vom Typ IIIb*.

In der Trapezoidisierung bilden Ecken vom Typ IIIb die Grenze zwischen zwei Trapezen oberhalb und einem Trapez unterhalb. Ecken vom Typ IIIa bilden die untere Grenze eines Trapezes.
Eine Übersicht über die Eckentypen ist in Abbildung 4.2 zu sehen.



(a) Typ I (b) Typ II (c) Typ III

Abbildung 4.2: Vergleich der Eckentypen. Das Innere des Polygons ist jeweils schraffiert.

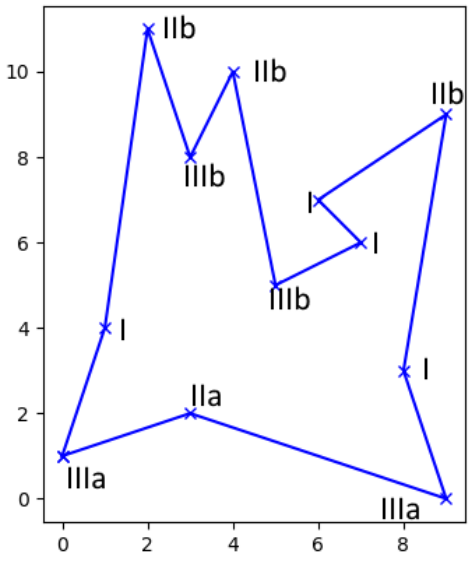


Abbildung 4.3: Ein einfaches Polygon mit Eckentypen.

In Abbildung 4.3 findet sich ein Polygon, in dem die Eckentypen beschriftet sind. Die Struktur dieses Polygons wurde aus [1] übernommen.

Der Algorithmus durchläuft die Ecken von oben nach unten und erstellt dabei Trapeze. Die unvollständigen Trapeze bestehen aus ihrer oben begrenzenden Ecke und den rechten und linken Kanten. Ein Trapez wird vervollständigt, indem auch die unten begrenzende Ecke hinzugefügt wird. Die unvollständigen Trapeze werden in einem 2-3-Baum gespeichert, aus dem sie bei Vervollständigung entfernt werden.

4.4.1.1 *Speichern der unvollständigen Trapeze*

Die Speicherung wird laut [8] in einem 2-3-Baum durchgeführt, da das Einfügen, Suchen und Entfernen von Elementen in 2-3-Bäumen in $\mathcal{O}(\log n)$ möglich ist. Diese Speicherung wird in [8] wie folgt beschrieben:

> In addition to the above, the algorithm uses a 2-3 tree holding the active trapezoids as leaves (a trapezoid is active if intersected by a horizontal line between the last vertex processed and the next one). The edges (right and left) are the keys used for comparison in searching. We use the fact that while a trapezoid is active, a vertex is inside the trapezoid if and only if it is inside its left edge and its right edge. This is true since we know the vertex being tested has to have a

> y coordinate within the closed interval defined by the coordinates of the two edge boundaries. This avoids having to compute the intersection of the edges with the current y horizontal line.

Dies ist in einiger Hinsicht unklar formuliert. Es ist nicht klar ersichtlich, ob die Datenelemente des Baumes einzelne Kanten oder Kantenpaare sind und wie sie mit den Trapezen in Verbindung stehen. Wie sich ein Eckpunkt „innerhalb“ einer Kante befinden kann, ist ebenfalls nicht klar. Nachdem ich provisorisch eine Liste als Datenstruktur verwendet habe, die aber eine Suchkomplexität von $\mathcal{O}(n)$ hat, bin ich zu folgender Interpretation gekommen:
In dem 2-3-Baum werden alle Kanten gespeichert, die momentan Begrenzung eines aktiven Trapezes sind. Jede Kante kennt das aktive Trapez, welches gerade an sie angrenzt. Dieses ist immer eindeutig bestimmt, da sich aktive Trapeze nie überschneiden und das Polygon von oben nach unten abgearbeitet wird. Die Trapeze befinden sich also nur indirekt in der Baumstruktur. Wie im obigen Abschnitt aus [8] beschrieben, muss die y-Koordinate der momentan verarbeiteten Ecke zwischen denen der Endpunkte jeder momentan gespeicherten Kante liegen, da die Kante ansonsten noch nicht zum Baum hinzugefügt oder bereits entfernt wurde. Daher ist für jede neu hinzuzufügende Kante, welche immer eine an die aktuelle Ecke angrenzende Kante ist, wohldefiniert, ob sie links oder rechts von einer Kante in der Datenstruktur liegt. Für die Zwecke der Operationen auf dem 2-3-Baum können die Kanten also als totalgeordnet betrachtet werden. Um die horizontale Lage zweier Kanten zueinander zu bestimmen, ist ein Endpunkt einer der beiden Kanten zu finden, dessen y-Koordinate zwischen denen der Endpunkte der anderen Kante liegt. Die x-Koordinate dieses Punktes kann dann mit der x-Koordinate des Schnittpunktes der Horizontalen durch den Punkt mit der anderen Kante verglichen werden. Da ein Vergleich dieser Art nicht zu vermeiden ist, gehe ich davon aus, dass der letzte Satz des obigen Zitats sich darauf bezieht, dass die Schnittpunkte nicht mit *allen* aktiven Ecken bestimmt werden müssen.
Der Grund für die Verwendung von 2-3-Bäumen sind die Laufzeiteigenschaften. Für meine Implementation habe ich daher eine besser auf Python abgestimmte Datenstruktur mit den gleichen Laufzeiteigenschaften genutzt. Dabei handelt es sich um `SortedList` aus dem Pythonmodul `sortedcontainers`, welches in [9] zu finden ist. Laut der dortigen Dokumentation haben Einfüge-, Entfern- und Suchoperationen dieses Typs ebenfalls eine Laufzeitkomplexität von $\mathcal{O}(\log n)$.
Die Quelle [1], welche für manche andere Implementationsdetails des Algorithmus nützlich war, behauptet, den Index der oberen Ecke des Trapezes in der Eckenliste des Polygons als Schlüssel im 2-3-Baum zu verwenden. Dies scheint aber keine verlässliche Methode zu sein, wie in Abbildung 4.4 zu sehen. In beiden Polygonen wird die Ecke mit dem Index 5 aufgrund ihrer y-Koordinate als dritte eingefügt und zu

diesem Zeitpunkt sind die beiden farblich markierten unvollständigen Trapeze aktiv. Im linken Polygon muss sie das linke (rote) Trapez beenden, im rechten Polygon das rechte (grüne). Diese Unterscheidung lässt sich nicht allein anhand der Indizes der Ecken treffen.

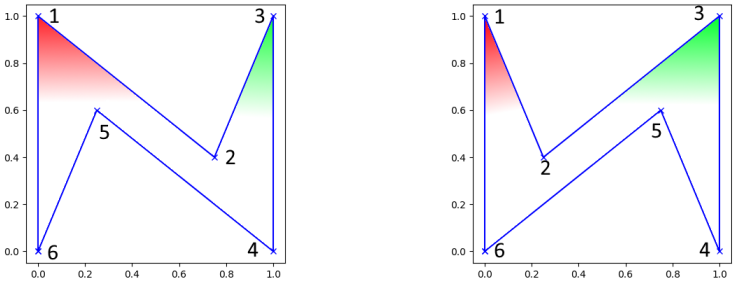


Abbildung 4.4: Polygone, in denen das enthaltende Trapez von Ecke 5 nicht allein anhand der Indizes festgestellt werden kann.

4.4.2 *Behandlung der Eckentypen*

Im Folgenden wird darauf eingegangen, wie der Algorithmus die Eckentypen konkret behandelt.

4.4.2.1 *Typ IIb*

Dies sind nach oben zeigende Ecken, die das Innere des Polygons unter sich haben. Darunter fällt insbesondere die oberste Ecke des Polygons, weshalb eine Ecke dieses Typs immer als erstes verarbeitet wird. Um eine Ecke dieses Typs zu verarbeiten, wird ein neues unvollständiges Trapez erstellt, welches oben von der Ecke und seitlich von den an die Ecke angrenzenden Kanten begrenzt wird. Bei diesen Trapezen wird es sich stets geometrisch um Dreiecke handeln, da die seitlichen Begrenzungen oben in der Ecke zusammenfallen. Ecken vom Typ IIb werden nie ein Trapez von unten begrenzen, da die Region über ihnen nicht im Inneren des Polygons liegt.

4.4.2.2 *Typ IIa*

Dies sind nach oben zeigende Ecken, die das Innere des Polygons über sich haben. Wird eine solche Ecke v verarbeitet, wird der 2-3-Baum nach dem Trapez durchsucht, in dessen Inneren v liegt. Dieses Trapez wird mit v als untere Ecke vervollständigt. Dann werden zwei neue Trapeze erstellt, welche v als obere Ecke haben. Eines dieser Trapeze liegt links von v und wird von der linken Kante des vervollständigten Trapezes und der linken an v angrenzenden Kante begrenzt. Das andere Trapez liegt rechts von v und wird von der rechten an v angrenzenden Kante und der rechten Kante des vervollständigten Trapezes begrenzt.

Die Region zwischen den an v angrenzenden Kanten liegt außerhalb des Polygons und ist daher nicht Teil eines Trapezes.

4.4.2.3 *Typ I*

Diese Ecken haben eine Kante nach oben und eine nach unten. Die nach oben führende Kante muss bereits die Begrenzungen eines Trapezes sein, da die Ecke am anderen Ende dieser Kante bereits verarbeitet wurde und selbst vom Typ I oder II ist. Dies gilt, da Ecken vom Typ III per Definition keine nach unten führenden Kanten aufweisen. Dieses Trapez wird mit der neuen Ecke als untere Begrenzung abgeschlossen und es wird ein neues Trapez erstellt, welches oben von der neuen Ecke und an den Seiten von der unten an die Ecke angrenzenden Kante und der im abgeschlossenen Trapez gegenüberliegenden Kante begrenzt wird.

4.4.2.4 *Typ IIIb*

Dies sind nach unten zeigende Ecken, die das Innere des Polygons unter sich haben. Beide Kanten, die an eine solche Ecke angrenzen, sind bereits Begrenzungen zweier unterschiedlicher Trapeze, da die angrenzenden Ecken bereits verarbeitet wurden und die Region zwischen den Kanten außerhalb des Polygons liegt. Diese beiden Trapeze werden mit der neuen Ecke als untere Begrenzung abgeschlossen und es wird ein neues Trapez erstellt, welches von der neuen Ecke und den jeweils äußeren Kanten der abgeschlossenen Polygone begrenzt wird, also denjenigen Kanten, die nicht die neue Ecke enthalten.

4.4.2.5 *Typ IIIa*

Dies sind nach unten zeigende Ecken, die das Innere des Polygons über sich haben. Darunter fällt insbesondere die unterste Ecke des Polygons, weshalb eine solche Ecke immer als letztes verarbeitet wird. Die angrenzenden Kanten einer solchen Ecke bilden bereits die Begrenzungen eines Trapezes, da die Region zwischen ihnen Teil des Polygons ist. Dieses Trapez wird mit der neuen Ecke als untere Begrenzung abgeschlossen.

4.4.3 *Eckenklassifizierung*

Um die Ecken klassifizieren zu können, ist es notwendig, festzustellen, ob der Winkel an einem Eckpunkt größer oder kleiner als 180° ist. Dazu kann die Voraussetzung, dass die Ecken im Uhrzeigersinn gegeben sind, genutzt werden. Dazu wird das Kreuzprodukt der Vektoren, welche von dem Eckpunkt zu seinen Nachbarn führen, genutzt. Seien (x, y) die Koordinaten des Eckpunktes, (x_0, y_0) die Koordinaten seines Vorgängers und (x_1, y_1) die Koordinaten seines Nachfolgers. $\vec{v_0} :=$

$\begin{pmatrix} x_0 - x \\ y_0 - y \end{pmatrix}$ ist dann der Vektor vom Eckpunkt zu seinem Vorgänger, $\vec{v_1} := \begin{pmatrix} x_1 - x \\ y_1 - y \end{pmatrix}$ der Vektor vom Eckpunkt zu seinem Nachfolger. Da die Eckpunkte im Uhrzeigersinn angegeben sind, ist der von $\vec{v_0}$ nach $\vec{v_1}$ im mathematischen Drehsinn, also gegen den Uhrzeigersinn, aufgespannte Winkel ein Innenwinkel des Polygons. Die z-Komponente des Kreuzprodukts

$$\begin{aligned} \vec{v_0}' \times \vec{v_1}' &= \begin{pmatrix} x_0 - x \\ y_0 - y \\ 0 \end{pmatrix} \times \begin{pmatrix} x_1 - x \\ y_1 - y \\ 0 \end{pmatrix} \\ &= \begin{pmatrix} 0 \\ 0 \\ (x_0 - x) \cdot (y_1 - y) - (y_0 - y) \cdot (x_1 - x) \end{pmatrix} \end{aligned}$$

der auf drei Dimensionen erweiterten Vektoren $\vec{v_0}'$ und $\vec{v_1}'$ hat dann im genutzten rechtshändigen Koordinatensystem positives Vorzeichen, wenn der aufgespannte Winkel kleiner als 180° ist, und negatives Vorzeichen, wenn er größer als 180° ist. Dies kann genutzt werden, um zwischen Ecken der Typen IIa und IIb bzw. IIIa und IIIb zu unterscheiden.
Die Unterscheidung zwischen den Typen I, II und III kann einfach durch Vergleich der y-Koordinate des Eckpunktes mit denen seiner Nachbarn getroffen werden. Bei einer Ecke vom Typ II liegen beide Nachbarn darunter, bei Typ III darüber und bei Typ I liegen sie auf verschiedenen Seiten. Somit kann der Typ einer Ecke mit Algorithmus 4 bestimmt werden:

4.4.4 *Verschränkte Unimonotonisierung*

Die unimonotonen Polygone werden in der verwendeten Variante des Algorithmus nach [1] bereits während der Trapezoidisierung erstellt. Immer wenn ein Trapez vervollständigt wird, wird es zur Unimonotonisierung hinzugefügt. Nach Definition 6 muss ein unimonotones Polygon (kurz UMP) eine Hauptkante haben, die die Eckpunkte mit kleinster und größter y-Koordinate verbindet. Die unvollständigen unimonotonen Polygone werden in einem 2-3-Baum *umps* gespeichert. Die Funktion `add_trapezoid` kennt *umps* und wird mit jedem vervollständigten Trapez aufgerufen, wobei die obere Ecke des Trapezes zu mindestens einem unimonotonen Polygon hinzugefügt wird. Trapeze werden je nach Typ der oberen Ecke unterschiedlich behandelt. Außerdem wird zwischen Trapezen, deren Diagonale extern, also eine Kante des Polygons, oder intern, also innerhalb des Polygons liegend, ist, unterschieden.

Algorithmus 4 Eckenklassifizierung

```
1:  function GET_VERTEX_TYPE(Ecke v)
2:      x, y ← Koordinaten von v
3:      x_0, y_0 ← Koordinaten des Vorgängers von v
4:      x_1, y_1 ← Koordinaten des Nachfolgers von v
5:      if y_0 < y < y_1 oder y_1 < y < y_0 then
6:          return Typ I
7:      else if y_0 < y und y_1 < y then                    ▷ Typ II
8:          if (x_0 − x) · (y_1 − y) − (y_0 − y) · (x_1 − x) > 0 then
9:              return Typ IIb
10:         else
11:             return Typ IIa
12:     else                                                ▷ Typ III
13:         if (x_0 − x) · (y_1 − y) − (y_0 − y) · (x_1 − x) > 0 then
14:             return Typ IIIb
15:         else
16:             return Typ IIIa
```

4.4.4.1 *Typ IIb mit externer Diagonalen*

In diesem Fall wird ein neues unimonotones Polygon begonnen, welches diejenige seitliche Begrenzung des Trapezes als Hauptkante hat, welche in y-Richtung länger ist.

4.4.4.2 *Typ IIb mit interner Diagonalen*

In diesem Fall werden zwei neue unimonotone Polygone erstellt, die jeweils eine der begrenzenden Kanten des Trapezes als Hauptkante haben.

4.4.4.3 *anderer Eckentyp mit externer Diagonalen*

In diesem Fall wird ein existierendes unimonotones Polygon gesucht, welches eine der begrenzenden Kanten des Trapezes als Hauptkante hat. Existiert dieses, wird der obere Eckpunkt des Trapezes zu diesem unimonotonen Polygon hinzugefügt. Ansonsten wird ein neues unimonotones Polygon erstellt, welches eine der begrenzenden Kanten als Hauptkante hat. Die Wahl der Kante ist hier beliebig. Ich verwende in meiner Implementation die linke Kante.

4.4.4.4 *anderer Eckentyp mit interner Diagonalen*

In diesem Fall wird ein existierendes unimonotones Polygon gesucht, welches die linke Kante des Trapezes als Hauptkante hat. Existiert dieses, wird der obere Eckpunkt des Trapezes zu diesem unimonotonen

Polygon hinzugefügt. Ansonsten wird ein neues unimonotones Polygon erstellt, welches die linke Kante als Hauptkante hat. Dieser Prozess wird dann für die rechte Kante wiederholt. Es werden also bis zu zwei neue unimonotone Polygone erstellt.
Die letzten beiden Punkte können zu einer Funktion zusammengefasst werden, die eine Menge von Kanten erhält, nach denen gesucht wird, wie in Algorithmus 5 zu sehen ist.

Algorithmus 5 Hinzufügen oder Erstellen von UMPs

```
1: function APPEND_TO_UMP(Ecke v, Kantenmenge edges)
2:     for all Kante e ∈ edges do
3:         p ← UMP aus umps mit e als Hauptkante
4:     if p existiert then
5:         füge v zu p hinzu
6:     else
7:         erstelle neues UMP mit einer Kante aus edges als Hauptkante
```

Hiermit kann nun die Funktion zum Hinzufügen eines Trapezes zur Unimonotonisierung erstellt werden, wie in Algorithmus 6 gezeigt.

Algorithmus 6 Hinzufügen eines Trapezes zur Unimonotonisierung

```
1: function ADD_TRAPEZOID(Trapez t)
2:     v ← obere Ecke von t
3:     w ← untere Ecke von t
4:     type ← get_vertex_type(v)
5:     if type = Typ IIb then
6:         if v, w ist eine der seitlich begrenzenden Kanten von t then
7:             erstelle neues UMP mit der Kante von t mit größerer y-Differenz als Hauptkante
8:         else                                          ▷ innere Diagonale
9:             erstelle neues UMP mit der linken Kante von t als Hauptkante
10:            erstelle neues UMP mit der rechten Kante von t als Hauptkante
11:    else
12:        if v, w ist eine der seitlich begrenzenden Kanten von t then
13:            append_to_UMP(v, {linke Kante von t, rechte Kante von t})
14:        else                                          ▷ innere Diagonale
15:            append_to_UMP(v, {linke Kante von t})
16:            append_to_UMP(v, {rechte Kante von t})
```

4.4.5 *Trapezoidisierung und Unimonotonisierung*

Mit diesen Bestandteilen kann jetzt der Trapezoidisierungsalgorithmus 7 angegeben werden.

Algorithmus 7 Trapezoidisierung und Unimonotonisierung nach Fournier und Montuno, modifiziert nach Adi

```
1: function TRAPEZOIDIZE_AND_UNIMONOTONIZE(Polygon p)
2:     vs ← Ecken von p nach absteigender y-Koordinate sortiert
3:     traps ← leerer 2-3-Baum
4:     umps ← leerer 2-3-Baum
5:     for all Ecke v ∈ vs do
6:         type ← get_vertex_type(v)
7:         if type = Typ I then
8:             suche Trapez t mit oberem Nachbarn von v in traps
9:             vervollständige t mit v als unterem Knoten
10:            entferne t aus traps
11:            add_trapezoid(t)
12:            füge neues Trapez zu traps hinzu mit v als oberem Knoten, begrenzt durch untere Kante von v und gegenüberliegende Kante von t
13:        else if type = Typ IIa then
14:            suche Trapez t in traps, das v beinhaltet
15:            vervollständige t mit v als unterem Knoten
16:            entferne t aus traps
17:            add_trapezoid(t)
18:            füge neues Trapez zu traps hinzu mit v als oberem Knoten, begrenzt durch linke Kante von v und rechte Kante von t
19:            füge neues Trapez zu traps hinzu mit v als oberem Knoten, begrenzt durch rechte Kante von v und linke Kante von t
20:        else if type = Typ IIb then
21:            füge neues Trapez zu traps hinzu mit v als oberem Knoten, begrenzt durch die Kanten von v
22:        else if type = IIIa then
23:            suche Trapez t mit linkem Nachbarn von v in traps
24:            vervollständige t mit v als unterem Knoten
25:            entferne t aus traps
26:            add_trapezoid(t)
27:        else                                              ▷ Typ IIIb
28:            suche Trapez t_1 mit linkem Nachbarn von v in traps
29:            vervollständige t_1 mit v als unterem Knoten
30:            entferne t_1 aus traps
31:            add_trapezoid(t_1)
32:            suche Trapez t_2 mit rechtem Nachbarn von v in traps
33:            vervollständige t_2 mit v als unterem Knoten
34:            entferne t_2 aus traps
35:            add_trapezoid(t_2)
36:            füge neues Trapez zu traps hinzu mit v als oberem Knoten, begrenzt durch linke Kante von t_1 und rechte Kante von t_2
```

4.4.6 *Triangulierung*

Abschließend müssen die unimonotonen Polygone noch trianguliert werden. Die Kanten der unimonotonen Polygone zusammen mit den Kanten, die bei ihrer Triangulierung entstehen, bilden dann die Kanten der Triangulierung des gesamten Polygons. Diese Triangulierung lässt sich nach [1] mit Algorithmus 8 durchführen. Dieser teilt nach und nach Ecken mit Innenwinkel kleiner 180° aus dem UMP als Dreiecke ab. Der Nachfolger einer Ecke ist hierbei die Ecke mit der nächstkleineren y-Koordinate im UMP.

Algorithmus 8 Triangulierung eines unimonotonen Polygons

```
1:  function TRIANGULATE_UNIMONOTONE(UMP p)
2:      current ← zweithöchste Ecke in p
3:      num ← Anzahl der Ecken von p
4:      stack ← leerer Stack
5:      while num ≥ 3 do
6:          if Winkel bei current ≤ 180° then
7:              füge Verbindung der Nachbarn von current zur Triangulierung hinzu
8:              num ← num − 1
9:              entferne current aus p
10:             if stack ist nicht leer then
11:                 current ← stack.pop()
12:             else
13:                 current ← Nachfolger von current
14:         else
15:             stack.push(current)
16:             current ← Nachfolger von current
```

4.4.7 *Polygone mit Löchern*

Der Algorithmus wurde oben für einfache Polygone, also Polygone ohne Löcher, beschrieben. Diese Tatsache wird im Algorithmus jedoch nie verwendet. Nach [8] lässt sich der Algorithmus also leicht auf Polygone mit Löchern erweitern, sogar, wenn die Löcher verschachtelt sind. Dabei muss nur auf die Laufrichtung der Polygonketten geachtet werden. Diese wird vom Algorithmus genutzt, um die Eckentypen festzustellen, da es nur unter Kenntnis der Laufrichtung möglich ist, lokal das Innere und Äußere des Polygons zu unterscheiden. Die äußere Kette wird weiterhin im Uhrzeigersinn angegeben. Ketten, die direkt innerhalb der äußersten liegen, werden gegen den Uhrzeigersinn angegeben. Ketten im Inneren solcher werden gegen den Uhrzeigersinn angegeben usw. Dies stellt sicher, dass der von drei in einer Kette auf-

einanderfolgenden Ecken A, B und C im mathematischen Drehsinn aufgespannte Winkel $\angle ABC$ stets ein Innenwinkel des Polygons ist.

4.4.8 *Laufzeitanalyse*

Der Trapezoidisierungsalgorithmus 7 wird von einer for-Schleife dominiert, welche alle Ecken des Polygons durchläuft und damit $\mathcal{O}(n)$ Durchläufe hat. Die einzigen Schritte innerhalb der Schleife, welche nicht in konstanter Zeit möglich sind, sind Suche, Hinzufügen und Entfernen von offenen Trapezen. Für diese wird durch die verwendete Datenstruktur sichergestellt, dass sie jeweils in $\mathcal{O}(\log n)$ möglich sind. In jedem Zweig der Fallunterscheidung wird eine konstante Anzahl von solchen Operationen durchgeführt. Das Hinzufügen des Trapezes zur Unimonotonisierung geschieht über Algorithmus 6, welcher ebenfalls durch Datenstrukturzugriffe in $\mathcal{O}(\log n)$ dominiert wird. Die Schleife im Trapezoidisierungsalgorithmus läuft also in $\mathcal{O}(n \log n)$. Die Sortierung der Eckenliste am Anfang kann mit einem gängigen Sortieralgorithmus in $\mathcal{O}(n \log n)$ durchgeführt werden. Die abschließende Triangulierung des unimonotonisierten Polygons durch Nutzung von Algorithmus 8 wird durch das Entfernen von $\mathcal{O}(n)$ Ecken dominiert und läuft in $\mathcal{O}(n)$. Damit hat der Algorithmus insgesamt eine Laufzeitkomplexität von $\mathcal{O}(n \log n)$.

4.5 LAS-VEGAS-ALGORITHMUS NACH SEIDEL

Der Algorithmus nach Fournier und Montuno ist zwar verhältnismäßig leicht zu verstehen, aber seine Laufzeit ist noch suboptimal. Seidel beschreibt in [14] einen Algorithmus von ähnlichem konzeptionellem Komplexitätsgrad mit besserer Laufzeit. Dieser Algorithmus trianguliert einfache Polygone.
Der Algorithmus verwendet Zufall. Es handelt sich um einen sogenannten Las-Vegas-Algorithmus. Dies ist ein Algorithmus, der Zufallselemente verwendet, aber immer ein korrektes Ergebnis liefert [10].

4.5.1 *Grundlagen*

Der Begriff der Trapezoidisierung wird hier von Polygonen auf beliebige Mengen von sich nicht außerhalb ihrer Eckpunkte schneidenden Strecken erweitert. Durch jeden Endpunkt einer solchen Strecke wird eine horizontale Linie gezogen, die sich in beide Richtungen jeweils bis zur nächsten Strecke fortsetzt. Diese Horizontalen zerlegen die Ebene zusammen mit den Strecken in Trapeze, die wie bekannt durch einen oberen und unteren Punkt sowie eine rechte und linke Strecke begrenzt sind. Im Gegensatz zu den Trapezen im Algorithmus von Fournier und Montuno können jedoch im allgemeinen Fall beliebig viele

dieser Begrenzungen fehlen, sodass sich ein Trapez ins Unendliche erstrecken kann. Außerdem speichert jedes Trapez die Trapeze, die oben und unten an es angrenzen. Der Algorithmus verwendet als Streckenmenge genau die Kanten des Polygons, da eine Trapezoidisierung wie bekannt in linearer Zeit in eine Unimonotonisierung und dann in eine Triangulierung überführt werden kann.

4.5.2 *Kernablauf des Algorithmus*

Der Algorithmus baut eine Trapezoidisierung Schritt für Schritt auf, in dem er die Kanten des Polygons in zufälliger Reihenfolge zur Ebene hinzufügt. Beim Hinzufügen jeder Kante wird die Trapezoidisierung der Ebene aktualisiert. Dazu wird das Trapez, das von der Horizontalen durch den oberen Eckpunkt geschnitten wird, horizontal in zwei neue Trapeze zerlegt. Das gleiche geschieht mit dem unteren Eckpunkt. Danach wird das Trapez, das von der Kante selbst geschnitten wird, entlang der Kante in zwei neue Trapeze zerlegt. Schließlich werden Trapeze, die ihre linke und rechte begrenzende Kante gemeinsam haben, miteinander verschmolzen.
Um diese Schritte durchzuführen, muss der Algorithmus in der Lage sein, die Position der Eckpunkte der Kante in der existierenden Trapezoidisierung zu finden. Dazu wird parallel zu der Trapezoidisierung eine Suchstruktur aufgebaut, die dies ermöglicht. Dabei handelt es sich um eine baumartige Struktur, genauer um einen azyklischer gerichteter Graphen mit einer einzelnen Quelle, mit drei Typen von Knoten. Die Blätter enthalten Trapeze, wobei jedes Trapez auch sein Blatt kennt. Die inneren Knoten dienen zum Finden der Trapeze. Es gibt zwei Typen von inneren Knoten: Y-Knoten enthalten eine y-Koordinate. Alle Trapeze im linken „Teilbaum“ des Y-Knotens liegen unterhalb dieser y-Koordinate, alle im rechten „Teilbaum“ oberhalb. X-Knoten enthalten eine Kante. Alle Trapeze im linken „Teilbaum“ liegen links von dieser Kante, alle im rechten „Teilbaum“ rechts von ihr. Es handelt sich um keinen echten Baum, da mehrere Pfade zum gleichen Blatt führen können. Dies geschieht durch den Schritt des Algorithmus, in dem Trapeze verschmolzen werden. Ein Beispiel einer einfachen Trapezoidisierung mit zugehöriger Suchstruktur ist in Abbildung 4.5 zu sehen. Die Kreise sind Blätter, die Rechtecke Y-Knoten und die abgerundeten Rechtecke X-Knoten.
Das Hinzufügen einer Kante zur Trapezoidisierung lässt sich dann nach [14] mit Algorithmus 9 erreichen.

4.5.3 *Vorläufige Laufzeitanalyse*

Wenn der Algorithmus 9 für jede Kante aufgerufen wird, handelt es sich um $\mathcal{O}(n)$ Aufrufe. Das Innere der Schleife wird dominiert durch das Finden des enthaltenden Trapezes, welches nach [14] eine erwar-

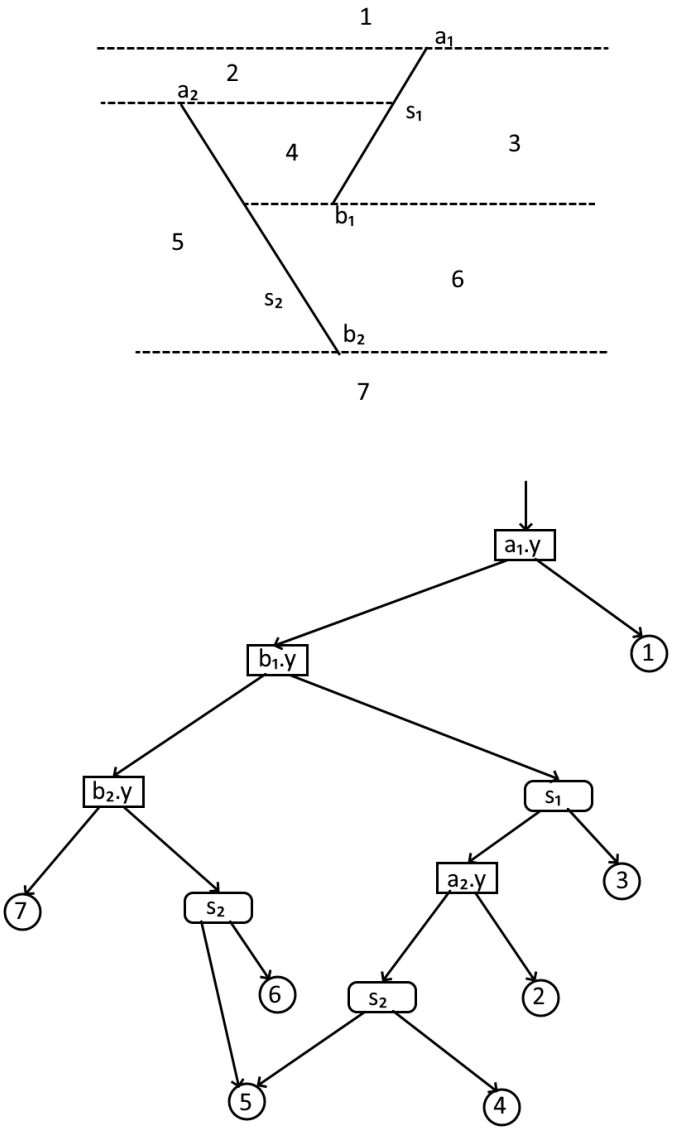


Abbildung 4.5: Eine einfache Trapezoidiserung und zugehörige Suchstruktur, übernommen aus [14]

Algorithmus 9 Hinzufügen einer Kante zur Trapezoidisierung

```
1: function ADD_SEGMENT(Kante k, Trapezoidisierung T, Suchstruktur S)
2:     for all Endpunkte p von K do
3:         if p ist kein Endpunkt einer bereits verarbeiteten Strecke then
4:             finde mit S das Trapez t, in dem p liegt
5:             entferne t aus T
6:             füge Trapez t_1 mit gleichen Begrenzungen wie t außer oberer Begrenzung p.y zu T hinzu
7:             füge Trapez t_2 mit gleichen Begrenzungen wie t außer unterer Begrenzung p.y zu T hinzu
8:             wandle zu t gehörendes Blatt in S in Y-Knoten mit Datenwert p.y, linkem Nachfolger t_1, rechtem Nachfolger t_2 um
9:     for all Trapeze t in T, die von k geschnitten werden do
10:        entferne t aus T
11:        füge Trapez t_1 mit gleichen Begrenzungen wie t außer rechter Begrenzung k zu T hinzu
12:        füge Trapez t_2 mit gleichen Begrenzungen wie t außer linker Begrenzung k zu T hinzu
13:        wandle zu t gehörendes Blatt in S in X-Knoten mit Datenwert k, linkem Nachfolger t_1, rechtem Nachfolger t_2 um
14:    for all Trapeze t, die links an k angrenzen, außer dem obersten do
15:        if oberer Nachbar t_1 von t hat gleiche linke Begrenzung then
16:            entferne t und t_1 aus T
17:            füge neues Trapez t_2 mit oberer Begrenzung von t_1, unterer Begrenzung von t, gleichen seitlichen Begrenzungen zu T hinzu
18:            erzeuge neuen Blattknoten für t_2
19:            ersetze alle Verbindungen in S, die zu den Knoten von t oder t_1 führen, durch Verbindungen zum Blatt von t_2
```

tete Laufzeit von $\mathcal{O}(\log n)$ hat. Das Ziel ist es jedoch, eine bessere Laufzeit als der Algorithmus von Fournier und Montuno zu erzielen. Dazu wird die Tatsache, dass alle Kanten zu einem Polygon gehören, genutzt.

4.5.4 *Verfolgen der Polygonkette durch die Trapezoidisierung*

Da der laufzeitintensive Teil des Algorithmus die Suche in der Suchstruktur ist, schlägt Seidel in [14] vor, die Segmente entlang der Polygonkette einzufügen, sodass die Lage eines Eckpunkts mithilfe der Lage des vorherigen Eckpunkts bestimmt werden kann, indem die verbindende Kante durch die bestehende Trapezoidisierung verfolgt wird. Dies ist möglich, da die Trapeze ihre oberen und unteren Nachbarn

speichern. Da sich keine Kanten überschneiden, muss die Kante nur über obere oder untere Nachbarn verfolgt werden. Da jetzt aber die zufällige Reihenfolge nicht mehr gegeben ist, wird die Trapezoidisierung oft komplexer und das Verfolgen einer Kante durch die Trapezoidisierung kann lineare Zeit brauchen. Die Lösung aus [14] ist es, die Kanten in zufälliger Reihenfolge einzufügen, aber in gewissen Abständen die gesamte Polygonkette durch die Trapezoidisierung zu verfolgen und sich die Lage der Eckpunkte zu merken.
Um den vollständigen Algorithmus anzugeben, muss noch etwas Notation eingeführt werden.

Definition 13 Für $i > 0$ heißt $\log^{(i)} n := \begin{cases} n & \text{falls } i = 0 \\ \log(\log^{(i-1)} n) & \text{sonst} \end{cases}$ der i-fach iterierte Logarithmus von n.

Definition 14 Für $n > 0$ sei $\log^* n := \max\left\{\ell \in \mathbb{N} \mid \log^{(\ell)} n \geq 1\right\}$.

Definition 15 Für festes $n > 0$ und $0 \leq h \leq \log^* n$ sei $N(h) := \left\lceil \frac{n}{\log^{(h)} n} \right\rceil$.

$\log^* n$ ist eine extrem langsam wachsende Funktion [5]. Beispielsweise ist $\log^* 10^{100000} = 4$. Für praktische Zwecke kann $\log^* n$ also fast als Konstante betrachtet werden.

[14] beschreibt dann Algorithmus 10 zur Trapezoidisierung basierend auf Algorithmus 9. Für alle Werte in tatsächlich auftretenden Grö-

Algorithmus 10 Trapezoidisierung nach Seidel

```
1:  function TRAPEZOIDIZE(einfaches Polygon p)
2:      n ← Anzahl der Ecken von p
3:      s_1, ..., s_n ← zufällige Anordnung der Kanten von p
4:      initialisiere Trapezoidisierung T und Suchstruktur S
5:      add_segment(s_1, T, S)
6:      for h = 1, ..., log* n do
7:          for N(h-1) < i ≤ N(h) do
8:              add_segment(s_i, T, S)
9:          verfolge die Kanten von p durch T, um für alle noch nicht
            eingefügten Punkte das enthaltende Trapez zu finden
10:     for N(log* n) < i ≤ n do
11:         add_segment(s_i, T, S)
```

ßenordnungen ist $N(\log^* n) > n$. Daher wird die letzte for-Schleife in der Praxis nicht ausgeführt.

4.5.5 *Laufzeitanalyse der Trapezoidisierung*

In [14] wird gezeigt, dass das Innere der ersten for-Schleife $\mathcal{O}(n)$ Zeit benötigt. Die Schleife läuft also insgesamt in $\mathcal{O}(n \log^* n)$. Die letzte for-Schleife läuft insgesamt in $\mathcal{O}(n)$. Damit läuft dieser Algorithmus in $\mathcal{O}(n \log^* n)$, also fast in Linearzeit.

4.5.6 *Triangulierung*

Wie aus Abschnitt 4.4.8 bekannt, ist die Triangulierung nach dem Zerlegen in unimonotone Polygone in $\mathcal{O}(n)$ möglich. Die Unimonotonisierung wird dort jedoch nach [1] im Verlauf der Trapezoidisierung aufgebaut. Dabei werden Annahmen über die Reihenfolge der Trapeze getroffen, die hier nicht zutreffen. Daher verwende ich zur Unimonotonisierung der vom Algorithmus nach Seidel trapezoidisierten Polygone den ürsprünglich in [8] beschriebenen rekursiven Algorithmus. Dieser läuft um den Rand des Polygons, beginnend bei einer übergebenen Ecke. Sobald eine Ecke erreicht wird, die Endpunkt einer inneren Diagonale eines Trapezes ist, wird sie vorübergehend mit dem gegenüberliegenden Endpunkt der Diagonale statt ihrem eigentlichen Nachfolger verbunden. Dann wird das Trapez beginnend bei dieser gegenüberliegenden Ecke rekursiv trianguliert, die vorübergehende Veränderung des Polygons rückgängig gemacht, und schließlich das restliche Polygon trianguliert. Wurde das ganze Polygon durchlaufen, hat der Algorithmus ein unimonotones Polygon erzeugt, welches dann mit Algorithmus 8 trianguliert wird. Dieser Prozess wird in Algorithmus 11 beschrieben. Für jede Ecke v sind `v.prev` und `v.next` dabei initial Vorgänger und Nachfolger der Ecke im Polygon, `v.done` ist ein mit „falsch“ initialisierter Wahrheitswert. Die Funktion `v.diagonal()` gibt das untere Ende einer Diagonalen im Sinne von Definition 8 zurück, deren oberes Ende v ist und deren enthaltendes Trapez im Inneren des Polygons liegt, oder `null`, falls keine solche Diagonale existiert.

Algorithmus 11 Triangulierung eines trapezoidisierten Polygons nach Fournier und Montuno

```
1:  function TRIANGULATE_TRAPEZOIDIZED(erster Eckpunkt first im Polygon p)
2:      current ← first
3:      while current.done ist falsch do
4:          current.done ← wahr
5:          bottom ← current.diagonal()
6:          if bottom ist nicht null then
7:              save_next ← current.next
8:              save_prev ← bottom.prev
9:              current.next ← bottom
10:             bottom.prev ← current
11:             triangulate_trapezoidized(bottom)        ▷ rekursiver Aufruf
12:             entferne das Trapez, in dem die Diagonale liegt
13:             current.done ← falsch
14:             bottom.done ← falsch
15:             current.next ← save_next
16:             bottom.prev ← save_prev
17:             current.prev ← bottom
18:             triangulate_trapezoidized(current)       ▷ abschließender rekursiver Aufruf
19:             return
20:         else
21:             current ← current.next
22:     erstelle ein UMP u durch Iteration über die Nachfolger von first
23:     triangulate_unimonotone(u)                       ▷ Algorithmus 8
```

5 FAZIT

5.1 VERGLEICH

In dieser Arbeit wurden vier Algorithmen zur Triangulierung von Polygonen beschrieben. Die Fächertriangulierung ist sehr einfach zu verstehen und zu implementieren und hat optimale Laufzeitkomplexität, funktioniert aber nur für konvexe Polygone. Die Ear-Clipping-Methode ist der implementationsmäßig einfachste Algorithmus, der mit allen einfachen Polygonen umgehen kann, läuft aber nur in $\mathcal{O}(n^2)$. Der Algorithmus von Fournier und Montuno ist konzeptionell komplizierter, aber immer noch recht einfach. Er kann außerdem mit beliebigen Polygonen, auch mit Löchern, umgehen und läuft in $\mathcal{O}(n \log n)$. Damit ist er gut für tatsächliche Anwendungen geeignet. Der Algorithmus von Seidel verbessert die Laufzeit weiter auf $\mathcal{O}(n \log^* n)$, also fast lineare Laufzeit. Er nutzt dafür aber Zufall, sodass es sich dabei nur um einen Erwartungswert handelt. Außerdem spielen einige Aspekte des Algorithmus, die zu der kleinen Laufzeitkomplexität führen, nur bei astronomisch großer Eingabe eine Rolle, was den Algorithmus für die Praxis unnötig kompliziert macht.
In der folgenden Tabelle gebe ich einen Überblick über die Algorithmen und ihren Nutzen, sortiert nach Laufzeitkomplexität:

Algorithmus	kann konkave Polygone triangulieren	kann Polygone mit Löchern triangulieren	(erwartete) Laufzeitkomplexität	verwendet Zufall
Ear Clipping	ja	nein	$\mathcal{O}(n^2)$	nein
Algorithmus nach Fournier und Montuno	ja	ja	$\mathcal{O}(n \log n)$	nein
Algorithmus nach Seidel	ja	nein	$\mathcal{O}(n \log^* n)$	ja
Fächertriangulierung	nein	nein	$\mathcal{O}(n)$	nein

Tabelle 5.1: Übersicht über die Algorithmen

5.2 AUSBLICK

Die in dieser Arbeit vorgestellten Algorithmen sind nicht die einzigen bekannten Algorithmen zur Triangulierung von Polygonen. Von besonderem Interesse ist hier der Algorithmus aus [4], welcher beliebige einfache Polygone in $\mathcal{O}(n)$ triangulieren kann, also in bestmöglicher Zeit. Dieser Algorithmus ist jedoch sehr kompliziert und aufgrund großer konstanter Terme nicht praxisgeeignet, sodass keine gängige Implementation existiert. Ein weniger komplizierter Las-Vegas-Algorithmus mit erwarteter linearer Laufzeit wird in [2] beschrieben.
In dieser Arbeit wurde nur das Finden einer einzelnen Triangulierung eines Polygons betrachtet. Es lässt sich jedoch auch die Frage untersuchen, wie viele Triangulierungen ein bestimmtes Polygon hat. Ein neues Resultat aus [7] zeigt, dass das Zählen der Triangulierungen eines allgemeinen Polygons #P-vollständig ist. Dort wird jedoch kein konkreter Algorithmus beschrieben, sondern nur eine Reduktion auf ein bekanntes Problem durchgeführt.

Teil I

APPENDIX

A

IMPLEMENTIERUNG

A.1 BESCHREIBUNG

Die Algorithmen habe ich in Python 3.9 [15] implementiert. Dabei habe ich folgende externe Bibliotheken genutzt:

- numpy [13] wird für Vektoroperationen bei geometrischen Berechnungen verwendet.
- matplotlib [11] wird für die graphische Darstellung der Polygone verwendet.
- sortedcontainers [9] wird für die Suchstruktur im Algorithmus in Abschnitt 4.4 verwendet.

Die Implementierung kann Polygone aus Textdateien einlesen. Eine solche Textdatei enthält eine oder mehrere durch Leerzeilen getrennte Polygonketten. Jede Kette besteht aus durch Zeilenumbrüchen getrennten Koordinaten von aufeinanderfolgenden Eckpunkten, wobei x- und y-Koordinate durch Komma voneinander getrennt sind. Um von den Triangulierungsalgorithmen korrekt verarbeitet zu werden, muss die äußere Kette im Uhrzeigersinn angegeben sein, darin direkt enthaltene Ketten gegen den Uhrzeigersinn etc. Es folgt ein Beispiel für dieses Format:

```
0,0
0,1
1,1
1,0

0.25,0.25
0.75,0.25
0.75,0.75
0.25,0.75

0.45,0.8
0.55,0.8
0.55,0.9
0.45,0.9

0.3,0.5
0.5,0.7
0.7,0.5
0.5,0.3
```

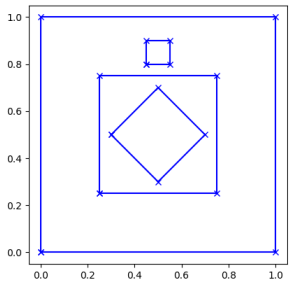


Abbildung A.1: Dieses Polygon wird vom Text links repräsentiert.

A.2 AUSFÜHRUNG

Es wird eine Installation von Python 3.9 [15] benötigt. Die externen Bibliotheken lassen sich dann mit `pip install numpy`, `pip install matplotlib` und `pip install sorted` installieren. Ein Polygon im oben beschriebenen Textformat lässt sich mit der Datei `main.py` triangulieren und als Bild ausgeben, indem diese im Format `python main.py <polygon> <algorithmus> <ausgabe>` aufgerufen wird, z.B. `python main.py nested_square.txt fournier output.png`. Die Optionen für den Algorithmus sind `fan`, `earclipping`, `fournier` und `seidel`.

A.3 `polygons.py`

Diese Datei enthält grundlegende Klassen und Funktionen, die von den Algorithmen genutzt werden. Zunächst wird wie in Abschnitt 3.2 die Klasse für die Ecken definiert.

Listing A.1: `polygons.py`

```
from typing import TextIO, Union, List, Tuple, Optional
import pathlib
import operator
import math
from enum import Enum
import functools
import matplotlib.pyplot as plt
import matplotlib.axes
import numpy as np


Path = Union[str, pathlib.Path]


@functools.total_ordering
class Vertex:
    """Klasse für die Ecken eines Polygons."""
    def __init__(self, x: float, y: float):
        self.x = float(x)
        self.y = float(y)
        self.orig_x = self.x  # für die Rotation
        self.orig_y = self.y
        self.neighbors: List[Vertex] = []  # Nachbarn im Polygon
        self.internal_neighbors: List[Vertex] = []  # Kanten der
            ↪ Triangulierung
        self.prev: Optional[Vertex] = None  # erlauben Verwendung
            ↪  als doppelt verlinkte Liste,
        self.next: Optional[Vertex] = None  # ohne das
            ↪ ursprüngliche Polygon zu verändern
        self.vtype = None
        self.done = False
```

```
        self.sinks_above = []
        self.sinks_below = []

    def add_neighbor(self, neighbor: 'Vertex'):
        """Füge eine im Polygon benachbarte Ecke hinzu."""
        self.neighbors.append(neighbor)

    def add_internal_edge_to(self, neighbor):
        """Füge eine Ecke hinzu, die in der Triangulierung durch
            ↪ eine Kante verbunden ist."""
        self.internal_neighbors.append(neighbor)

    def coords(self):
        return self.x, self.y

    def rotate(self, angle):
        """Rotiert um einen gegebenen Winkel um den Ursprung."""
        # dtype=object, weil Indizieren mit np.bool_ deprecated
        self.x, self.y = (np.array([[math.cos(angle), -math.sin(
            ↪ angle)],
                                    [math.sin(angle), math.cos(
                                        ↪ angle)]], dtype=object)
                          @ np.array(self.coords(), dtype=object)
                              ↪ )

    def unrotate(self):
        """Stellt die ursprünglichen Koordinaten wieder her"""
        self.x = self.orig_x
        self.y = self.orig_y

    def __gt__(self, other):
        return self.y > other.y or self.y == other.y and self.x >
            ↪  other.x

    def __eq__(self, other):
        return self.x == other.x and self.y == other.y

    def __str__(self):
        return str((self.x, self.y))

    def __hash__(self):
        return hash((self.x, self.y))
```

Darauf folgen diverse geometrische Hilfsfunktionen, die die Lage von geometrischen Objekten zueinander bestimmen.

```
def segment_to_linear_fun(start: Vertex, end: Vertex) -> Tuple[
    ↪ float, float]:
    """Gibt Steigung und y-Achsenabschnitt der linearen Funktion
        ↪ durch die zwei Punkte zurück."""
    v1, v2 = (start, end) if start.x > end.x else (end, start)
    m = (v2.y - v1.y) / (v2.x - v1.x)
    b = v1.y - m*v1.x
```

```
    return m, b


def x_at_y(start: Vertex, end: Vertex, y: float) -> float:
    """Gibt den x-Wert der Geraden durch die zwei Punkte an der
        ↪ gegebenen y-Koordinate zurück."""
    if start.x == end.x:
        return start.x
    if y == start.y:
        return start.x
    if y == end.y:
        return end.x
    m, b = segment_to_linear_fun(start, end)
    return (y - b) / m


def get_intersection(start1: Vertex, end1: Vertex, start2: Vertex
    ↪ , end2: Vertex) -> Optional[Tuple[float, float]]:
    """Gibt den Schnittpunkt zweier Strecken zurück, falls dieser
        ↪  existiert."""
    if start1.x == end1.x or start2.x == end2.x:  # eine der
        ↪ Strecken ist vertikal
        if start1.x == end1.x and start2.x == end2.x:
            return None  # beide vertikal -> parallel
        if start2.x == end2.x:
            start1, start2 = start2, start1
            end1, end2 = end2, end1
        x = start1.x
        m, b = segment_to_linear_fun(start2, end2)
        y = m * x + b
        seg1 = sorted((start1, end1), key=operator.attrgetter("y"
            ↪ ))
        seg2 = sorted((start2, end2), key=operator.attrgetter("x"
            ↪ ))
        return (x, y) if seg1[0].y <= y <= seg1[1].y and seg2[0].
            ↪ x <= x <= seg2[1].x else None
    m1, b1 = segment_to_linear_fun(start1, end1)
    m2, b2 = segment_to_linear_fun(start2, end2)
    if m1 == m2:
        return None
    x = (b2 - b1) / (m1 - m2)
    y = m1 * x + b1
    seg1 = sorted((start1, end1), key=operator.attrgetter("x"))
    seg2 = sorted((start2, end2), key=operator.attrgetter("x"))
    return (x, y) if seg1[0].x <= x <= seg1[1].x and seg2[0].x <=
        ↪  x <= seg2[1].x else None


def zcross(v1, v2):
    """Gibt die z-Komponente des Kreuzprodukts der auf 3
        ↪ Dimensionen erweiterten Vektoren zurück."""
    return v1[0] * v2[1] - v2[0] * v1[1]
```

```
50
def same_side(start: Vertex, end: Vertex, p1: Vertex, p2: Vertex)
    ↪ :
    """Prüft, ob zwei Punkte auf derselben Seite einer Gerade
        ↪ liegen."""
    edge = np.subtract(end.coords(), start.coords())
    return (math.copysign(1, zcross(edge, np.subtract(p1.coords()
        ↪ , start.coords())))
55          == math.copysign(1, zcross(edge, np.subtract(p2.
                ↪ coords(), start.coords()))))


def point_in_triangle(pt: Vertex, v1: Vertex, v2: Vertex, v3:
    ↪ Vertex):
    """Prüft, ob ein Punkt in einem Dreieck liegt."""
60  return not pt in (v1, v2, v3) and same_side(v1, v2, v3, pt)
        ↪ and same_side(v2, v3, v1, pt) and same_side(v3, v1, v2,
        ↪  pt)


def left_of(start: Vertex, end: Vertex, p: Vertex):
    """Prüft, ob p zur Linken einer Geraden liegt."""
65  if start.x == end.x:
        return p.x < start.x
    m, b = segment_to_linear_fun(start, end)
    return p.x < (p.y - b) / m


70
class Winding(Enum):
    CLOCKWISE = -1
    COUNTERCLOCKWISE = 1


75
def is_reflex_angle(ray1: Vertex, vertex: Vertex, ray2: Vertex,
    ↪ winding=Winding.CLOCKWISE):
    """Prüft, ob ein Winkel überstumpf (>180°) ist."""
    # nutzt Vorzeichen der z-Komponente des Kreuzproduktes
    v1 = np.subtract(ray1.coords(), vertex.coords(), dtype=object
        ↪ )
80  v2 = np.subtract(ray2.coords(), vertex.coords(), dtype=object
        ↪ )
    return zcross(v1, v2) * winding.value > 0


class PolygonStateError(Exception):
85  pass
```

Die Klassen für Polygonketten und Polygone sind wie in 3.3 beschrieben definiert.

```
class PolygonalChain:
    """Klasse für eine einzelne Polygonkette."""
```

```
    def __init__(self):
        self.vertices: List[Vertex] = []
        self.closed = False
        self.winding = None
        self.edges = []

    def add_to_end(self, v: Vertex):
        if self.closed:
            raise PolygonStateError("polygonal chain is closed")
        if self.vertices:
            pred = self.vertices[-1]
            pred.add_neighbor(v)
            v.add_neighbor(pred)
            self.edges.append((pred, v))
        self.vertices.append(v)

    def is_closeable(self):
        return len(self) >= 3

    def close(self):
        if not self.is_closeable():
            raise PolygonStateError("chain not long enough to
                ↪ close")
        self.vertices[0].neighbors.insert(0, self.vertices[-1])
        self.vertices[-1].add_neighbor(self.vertices[0])
        self.edges.append((self.vertices[-1], self.vertices[0]))
        self.closed = True
        # bestimme Umlaufrichtung mit Gaußscher Trapezformel
        self.winding = (Winding.CLOCKWISE
                        if (sum((v2.x - v1.x) * (v2.y + v1.y)
                                for v1, v2 in zip(self.vertices,
                                    ↪ self.vertices[1:] + [self.
                                    ↪ vertices[0]]))
                            > 0)
                        else Winding.COUNTERCLOCKWISE)

    def swap_winding(self):
        """Kehrt die Umlaufrichtung des Polygons um."""
        self.vertices = self.vertices[::-1]
        for v in self.vertices:
            v.neighbors[0], v.neighbors[1] = v.neighbors[1], v.
                ↪ neighbors[0]
        self.winding = Winding.COUNTERCLOCKWISE if self.winding
            ↪ is Winding.CLOCKWISE else Winding.CLOCKWISE

    def force_winding(self, winding: Winding):
        """Erzwingt eine bestimmte Umlaufrichtung."""
        if winding is not self.winding:
            self.swap_winding()

    def get_first_edge(self) -> Tuple[Vertex, Vertex]:
        return self.vertices[0], self.vertices[1]
```

```
    def __len__(self):
        return len(self.vertices)

    def __str__(self):
        return " - ".join(map(str, self.vertices)) + " -" if self
            ↪ .closed else ""

    def __getitem__(self, item):
        return self.vertices[item]


class Polygon:
    """Klasse für ein allgemeines Polygon."""
    def __init__(self):
        self.chains: List[PolygonalChain] = []
        self.vertices: List[Vertex] = []

    def add_chain(self, chain: PolygonalChain):
        self.chains.append(chain)
        self.vertices.extend(chain.vertices)

    def get_edges(self):
        return sum((chain.edges for chain in self.chains), start
            ↪ =[])

    def rotate(self, angle):
        for v in self.vertices:
            v.rotate(angle)

    def unrotate(self):
        """Stellt die urspüngliche Rotation wieder her."""
        for v in self.vertices:
            v.unrotate()

    def eliminate_horizontals(self):
        """Rotiert das Polygon, bis keine Ecken die gleiche y-
            ↪ Koordinate haben."""
        divisions = 10
        rots = 0
        while len(set(v.y for v in self.vertices)) < len(self.
            ↪ vertices):
            rots += 1
            if rots == divisions:
                rots = 0
                divisions *= 10
                self.unrotate()
            self.rotate(1 / divisions * 2 * math.pi)

    def __str__(self):
        return "\n".join(map(str, self.chains))
```

Es folgen einige Hilfsfunktionen, welche die Mitgliedschaft eines Polygons in den in Kapitel 2 beschriebenen Typen prüfen.

```
def is_nonintersecting(p: Polygon):
    """Prüft, ob sich irgendwelche Kanten des Polygons
        ↪ überschneiden."""
    edges = p.get_edges()
    for i, e in enumerate(edges):
        for j, f in enumerate(edges):
            if e is not f and e[0] not in f and e[1] not in f and
                ↪  get_intersection(*e, *f) is not None:
                return False
    return True


def is_convex(p: Polygon):
    if len(p.chains) > 1:
        return False
    chain = p.chains[0]
    count = len(chain)
    for i in range(count):
        if is_reflex_angle(chain[(i - 1) % count], chain[i],
            ↪ chain[(i + 1) % count], chain.winding):
            return False
    return True


def is_simple(p: Polygon):
    return len(p.chains) == 1


def is_monotone(p: Polygon):
    if not is_simple(p):
        return False

    max_v = max(p.chains[0].vertices, key=operator.attrgetter("y"
        ↪ ))
    min_v = min(p.chains[0].vertices, key=operator.attrgetter("y"
        ↪ ))

    cur = min_v
    nex = min_v.neighbors[1]
    while cur != max_v:
        if nex.y < cur.y:
            return False
        cur, nex = nex, nex.neighbors[1]
    while cur != min_v:
        if nex.y > cur.y:
            return False
        cur, nex = nex, nex.neighbors[1]


def is_unimonotone(p: Polygon):
```

```
    return (is_monotone(p)
            and max(p.chains[0].vertices, key=operator.attrgetter
                ↪ ("y"))
            in min(p.chains[0].vertices, key=operator.attrgetter(
                ↪ "y")).neighbors)
```

Schließlich enthält die Datei noch Funktionen zum Einlesen von Polygonen aus Textdateien im in A.1 beschriebenen Format.

```
def read_polygon_from_path(path: Path, force_legal=True) ->
    ↪ Polygon:
    with open(path) as f:
        polygon = read_polygon(f, force_legal)
    return polygon


def draw_polygon(polygon: Polygon, ax: matplotlib.axes.Axes =
    ↪ None):
    if ax is None:
        fig, ax = plt.subplots()
        ax.set_aspect("equal")
    for vertex in sum((chain.vertices for chain in polygon.chains
        ↪ ), start=[]):
        for other in vertex.internal_neighbors:
            ax.plot((vertex.x, other.x), (vertex.y, other.y),
                ↪ color="red")
    for chain in polygon.chains:
        ax.plot(*zip(*((v.x, v.y) for v in chain.vertices + [
            ↪ chain.vertices[0]])), color="blue", marker="x")
    return ax
```

A.4 fan.py

Diese Datei enthält die in 4.2 beschriebene Fächertriangulierung, also die Implementierung von Algorithmus 2.

Listing A.2: fan.py

```
import polygons
from polygons import Polygon


def fan_triangulation(polygon: Polygon):
    if not polygons.is_convex(polygon):
        raise ValueError("polygon is not convex")
    v = polygon.chains[0].vertices[0]
    for w in polygon.chains[0].vertices:
        if w != v and w not in v.neighbors:
            v.add_internal_edge_to(w)
```

A.5 earclipping.py

Diese Datei enthält die in Abschnitt 4.3 beschriebene Ear-Clipping-Methode, also die Implementierung von 3.

Listing A.3: earclipping.py

```
import polygons
from polygons import Polygon


def ear_clipping(p: Polygon):
    if not polygons.is_simple(p):
        raise ValueError("polygon is not simple")
    chain = p.chains[0]
    count = len(chain)
    for v in chain:
        v.prev = v.neighbors[0]
        v.next = v.neighbors[1]
    reflex_vertices = []
    for i in range(count):
        if polygons.is_reflex_angle(chain[(i - 1) % count], chain
            ↪ [i], chain[(i + 1) % count], chain.winding):
            reflex_vertices.append(chain[i])
    ears = set(filter(lambda v: v not in reflex_vertices and not
        ↪ any(polygons.point_in_triangle(pt, v, *v.neighbors) for
        ↪  pt in reflex_vertices),
                      chain.vertices))
    while ears and count >= 4:
        ear = ears.pop()
        count -= 1
        ear.prev.add_internal_edge_to(ear.next)
        ear.prev.next = ear.next
        ear.next.prev = ear.prev
        ear.done = True
        for n in (ear.prev, ear.next):
            if not polygons.is_reflex_angle(n.prev, n, n.next,
                ↪ chain.winding):
                if n in reflex_vertices:
                    reflex_vertices.remove(n)
        for n in (ear.prev, ear.next):
            if not polygons.is_reflex_angle(n.prev, n, n.next,
                ↪ chain.winding):
                is_ear = not any(polygons.point_in_triangle(pt, n
                    ↪ , n.prev, n.next) for pt in reflex_vertices
                    ↪ )
                if is_ear and not n.done:
                    ears.add(n)
                elif n in ears:
                    ears.remove(n)
            else:
                reflex_vertices.append(n)
```

```
39

if __name__ == "__main__":
    p = polygons.read_polygon_from_path("data/adi_1988_simple.txt
        ↪ ")
    ear_clipping(p)
44  polygons.draw_polygon(p).get_figure().show()
```

A.6 fournier.py

Diese Datei enthält den in Abschnitt 4.4 beschriebenen Algorithmus. Zunächst wird der Algorithmus 4 zur Bestimmung des Eckentyps implementiert.

Listing A.4: fournier.py

```
1  """Triangulierung nach Fournier und Montuno in O(n log n)."""
   import functools
   import operator
   from dataclasses import dataclass, field
   from enum import Enum
6  from typing import Callable, Tuple, List, Optional,
       ↪ MutableMapping, MutableSequence

   from sortedcontainers import SortedList

   import polygons
11 from polygons import Polygon, Vertex


   class VertexType(Enum):
       REGULAR = (1, "")
16     STALAGMITE_ABOVE = (2, "a")
       STALAGMITE_BELOW = (2, "b")
       STALACTITE_ABOVE = (3, "a")
       STALACTITE_BELOW = (3, "b")


21
   def get_vertex_type(v: Vertex, winding: polygons.Winding):
       y = v.y
       pred_y = v.neighbors[0].y
       succ_y = v.neighbors[1].y
26     if pred_y < y < succ_y or succ_y < y < pred_y:
           return VertexType.REGULAR
       if pred_y < y and succ_y < y:  # Typ II / Stalagmit
           return (VertexType.STALAGMITE_ABOVE
                   if polygons.is_reflex_angle(v.neighbors[0], v, v.
                       ↪ neighbors[1], winding)
31                 else VertexType.STALAGMITE_BELOW)
       return (VertexType.STALACTITE_BELOW
```

```
            if polygons.is_reflex_angle(v.neighbors[0], v, v.
                ↪ neighbors[1], winding)
            else VertexType.STALACTITE_ABOVE)
```

Dann werden die notwendigen geometrischen Objekte definiert. Dabei wird hier eine Kantenstruktur benötigt, die das momentan an die Kante angrenzende aktive Trapez kennt, da diese Kanten in der Suchstruktur (in [8] im 2-3-Baum) gespeichert werden und zum Suchen der Trapeze genutzt werden.

```
@dataclass
class Trapezoid:
    top_vertex: Vertex
    left_edge: "Edge"
    right_edge: "Edge"
    bottom_vertex: Vertex = None

    def __str__(self):
        return f"{str(self.top_vertex)} - {str(self.bottom_vertex
            ↪ )}"


class UnimonotonePolygon:

    def __init__(self, main_edge: Tuple[Vertex, Vertex], v:
        ↪ Vertex):
        self.main_edge = main_edge
        self.vertices = [v]


@functools.total_ordering
@dataclass
class Edge:
    vertices: Tuple[Vertex, Vertex]
    # Kante hat immer höchstens ein aktives angrenzendes Trapez:
    trapezoid: Optional[Trapezoid] = field(default=None)

    def __eq__(self, other):
        return (self.vertices == other.vertices
                or self.vertices == (other.vertices[1], other.
                    ↪ vertices[0]))

    def __hash__(self):
        return hash((min(self.vertices), max(self.vertices)))

    def __lt__(self, other):
        """Prüft, ob die Kante links von einer anderen liegt."""
        if max(self.vertices, key=operator.attrgetter("y")) ==
            ↪ max(other.vertices, key=operator.attrgetter("y")):
            return min(self.vertices, key=operator.attrgetter("y"
                ↪ )).x < min(other.vertices, key=operator.
                ↪ attrgetter("y")).x
```

```
            if min(self.vertices, key=operator.attrgetter("y")) ==
                ↪ min(other.vertices, key=operator.attrgetter("y")):
                return max(self.vertices, key=operator.attrgetter("y"
                    ↪ )).x < max(other.vertices, key=operator.
                    ↪ attrgetter("y")).x
            if min(other.vertices[0].y, other.vertices[1].y) < self.
                ↪ vertices[0].y < max(other.vertices[0].y, other.
                ↪ vertices[1].y):
                return polygons.left_of(*other.vertices, self.
                    ↪ vertices[0])
            if min(other.vertices[0].y, other.vertices[1].y) < self.
                ↪ vertices[1].y < max(other.vertices[0].y, other.
                ↪ vertices[1].y):
                return polygons.left_of(*other.vertices, self.
                    ↪ vertices[1])
            if min(self.vertices[0].y, self.vertices[1].y) < other.
                ↪ vertices[1].y < max(self.vertices[0].y, self.
                ↪ vertices[1].y):
                return not polygons.left_of(*self.vertices, other.
                    ↪ vertices[1])
            if min(self.vertices[0].y, self.vertices[1].y) < other.
                ↪ vertices[1].y < max(self.vertices[0].y, self.
                ↪ vertices[1].y):
                return not polygons.left_of(*self.vertices, other.
                    ↪ vertices[1])
            else:
                raise ValueError("edges are not next to each other")
                    ↪  # keine Totalordnung

        def __contains__(self, item):
            return item in self.vertices

        def __str__(self):
            return " - ".join(map(str, self.vertices))
```

Die Klasse `UnimonotoneBuilder` wird zum Aufbau der Unimonotonisierung verwendet. Die Methode `add_trapezoid` implementiert Algorithmus 6. Die Methode `_append_to_UMP` implementiert Algorithmus 5.

```
class UnimonotoneBuilder:
    """Hilfsklasse für den Aufbau der Unimonotonisierung nach Adi
        ↪ """
    def __init__(self):
        self.finished: List[UnimonotonePolygon] = []
        self.umps: MutableMapping[Edge, UnimonotonePolygon] = {}

    def add_trapezoid(self, t: Trapezoid):
        v = t.top_vertex
        vtype = v.vtype
        if vtype is None:
            v.vtype = (vtype := get_vertex_type(v, polygons.
                ↪ Winding.CLOCKWISE))
```

```
            if vtype is VertexType.STALAGMITE_BELOW:
                if (t.bottom_vertex in t.left_edge and t.top_vertex
                    ↪ in t.left_edge
                        or t.bottom_vertex in t.right_edge and t.
                            ↪ top_vertex in t.right_edge):  # externe
                            ↪  Diagonale
                    if v.y - v.neighbors[0].y > v.y - v.neighbors[1].
                        ↪ y:
                        self.umps[t.left_edge] = UnimonotonePolygon(t
                            ↪ .left_edge.vertices, v)
                    else:
                        self.umps[t.right_edge] = UnimonotonePolygon(
                            ↪ t.right_edge.vertices, v)
                else:
                    self.umps[t.left_edge] = UnimonotonePolygon(t.
                        ↪ left_edge.vertices, v)
                    self.umps[t.right_edge] = UnimonotonePolygon(t.
                        ↪ right_edge.vertices, v)
            else:
                if (t.bottom_vertex in t.left_edge and t.top_vertex
                    ↪ in t.left_edge
                        or t.bottom_vertex in t.right_edge and t.
                            ↪ top_vertex in t.right_edge):  # externe
                            ↪  Diagonale
                    self._append_to_UMP(v, (t.left_edge, t.right_edge
                        ↪ ))
                else:
                    self._append_to_UMP(v, (t.left_edge,))
                    self._append_to_UMP(v, (t.right_edge,))

        def _append_to_UMP(self, v: Vertex, edges):
            found = False
            for edge in edges:
                if edge in self.umps:
                    self.umps[edge].vertices.append(v)
                    found = True
            for n in v.neighbors:
                if tuple(sorted((n, v), key=operator.attrgetter("y"),
                    ↪  reverse=True)) in self.umps:
                    ump = self.umps[Edge((n, v))]
                    ump.vertices.append(v)
                    self.finished.append(ump)
                    del self.umps[Edge((n, v))]
            if not found:
                self.umps[edges[0]] = UnimonotonePolygon(edges[0].
                    ↪ vertices, v)

        def close(self):
            for p in self.umps.values():
                p.vertices.append(min(p.main_edge, key=operator.
                    ↪ attrgetter("y")))
                self.finished.append(p)
```

Die folgende Funktion implementiert den Hauptteil der Triangulierung nach Fournier und Montuno, den Trapezoidisierungsalgorithmus 7.

```
def trapezoidize_fournier(p: Polygon, umps: UnimonotoneBuilder =
    ↪ None):
    ordered_vertices = sorted(p.vertices, key=operator.attrgetter
        ↪ ("y"), reverse=True)
    edges: SortedList[Edge] = SortedList()  # diese Struktur
        ↪ ersetzt den 2-3-Baum im ursprünglichen Algorithmus
    finished_trapezoids: MutableSequence[Trapezoid] = []
    for v in ordered_vertices:
        vertex_type = get_vertex_type(v, polygons.Winding.
            ↪ CLOCKWISE)
        v.vtype = vertex_type
        if vertex_type is VertexType.REGULAR:  # Typ I
            # finde das zu beendende Trapez
            t_to_close = edges[edges.index(Edge((v, max(v.
                ↪ neighbors, key=operator.attrgetter("y"))))].
                ↪ trapezoid
            t_to_close.bottom_vertex = v
            finished_trapezoids.append(t_to_close)
            if umps:
                umps.add_trapezoid(t_to_close)  # fertiges Trapez
                    ↪  zur Unimonotonisierung hinzufügen
            # neues Trapez erstellen:
            if v in t_to_close.left_edge:
                new_left = Edge((v.neighbors[0], v))
                new_t = Trapezoid(v, new_left, t_to_close.
                    ↪ right_edge)
                edges.remove(t_to_close.left_edge)
                edges.add(new_left)
                new_t.left_edge.trapezoid = new_t
                new_t.right_edge.trapezoid = new_t
            elif v in t_to_close.right_edge:
                new_right = Edge((v, v.neighbors[1]))
                new_t = Trapezoid(v, t_to_close.left_edge,
                    ↪ new_right)
                edges.remove(t_to_close.right_edge)
                edges.add(new_right)
                new_t.left_edge.trapezoid = new_t
                new_t.right_edge.trapezoid = new_t
            else:
                assert False, "regular vertex not bottom of any
                    ↪ trapezoid?"

        elif vertex_type is VertexType.STALAGMITE_ABOVE:  # Typ
            ↪ IIa
            # finde Einfügeort in der Kantenstruktur
            v_left_edge = Edge((v, min(v.neighbors, key=operator.
                ↪ attrgetter("x"))))
            v_right_edge = Edge((v, max(v.neighbors, key=operator
                ↪ .attrgetter("x"))))
```

```
            split_point = edges.bisect_left(v_left_edge)  #
                ↪ bestimmt Position der linken neuen Kante in der
                ↪  Struktur
            t_to_close = edges[split_point].trapezoid  # zu
                ↪ vervollständigendes Trapez gefunden
            t_to_close.bottom_vertex = v
            finished_trapezoids.append(t_to_close)
            if umps:
                umps.add_trapezoid(t_to_close)  # fertiges Trapez
                    ↪  zur Unimonotonisierung hinzufügen
            # zwei neue Trapeze entstehen:
            new_t_left = Trapezoid(v, t_to_close.left_edge,
                ↪ v_left_edge)
            new_t_right = Trapezoid(v, v_right_edge, t_to_close.
                ↪ right_edge)
            edges.add(v_left_edge)
            edges.add(v_right_edge)
            new_t_left.left_edge.trapezoid = new_t_left
            new_t_left.right_edge.trapezoid = new_t_left
            new_t_right.left_edge.trapezoid = new_t_right
            new_t_right.right_edge.trapezoid = new_t_right

        elif vertex_type is VertexType.STALAGMITE_BELOW:  # Typ
            ↪ IIb
            left_edge = Edge((v.neighbors[0], v))
            right_edge = Edge((v, v.neighbors[1]))
            t_new = Trapezoid(v, left_edge, right_edge)
            left_edge.trapezoid = t_new
            right_edge.trapezoid = t_new
            edges.add(left_edge)
            edges.add(right_edge)

        elif vertex_type is VertexType.STALACTITE_ABOVE:  # Typ
            ↪ IIIa
            # Trapez zwischen v's angrenzenden Kanten wird
                ↪ geschlossen
            t_to_close = edges[edges.index(Edge((v, v.neighbors
                ↪ [0])))].trapezoid  # reicht, eine Kante zu
                ↪ finden
            t_to_close.bottom_vertex = v
            finished_trapezoids.append(t_to_close)
            if umps:
                umps.add_trapezoid(t_to_close)
            edges.remove(t_to_close.left_edge)
            edges.remove(t_to_close.right_edge)

        else:
            assert vertex_type is VertexType.STALACTITE_BELOW  #
                ↪ Typ IIIb
            # finde beide zu schließende Trapeze
            t_to_close_1 = edges[edges.index(Edge((v, v.neighbors
                ↪ [0])))].trapezoid
```

```
            t_to_close_2 = edges[edges.index(Edge((v, v.neighbors
                ↪ [1])))].trapezoid
            if v in t_to_close_1.right_edge:
                assert v in t_to_close_2.left_edge
                t_to_close_left, t_to_close_right = t_to_close_1,
                    ↪  t_to_close_2
            else:
                assert v in t_to_close_1.left_edge
                assert v in t_to_close_2.right_edge
                t_to_close_left, t_to_close_right = t_to_close_2,
                    ↪  t_to_close_1
            t_to_close_left.bottom_vertex = v
            t_to_close_right.bottom_vertex = v
            finished_trapezoids.append(t_to_close_left)
            finished_trapezoids.append(t_to_close_right)
            if umps:
                umps.add_trapezoid(t_to_close_left)
                umps.add_trapezoid(t_to_close_right)
            edges.remove(t_to_close_left.right_edge)
            edges.remove(t_to_close_right.left_edge)
            # neues Trapez wird erstellt:
            new_t = Trapezoid(v, t_to_close_left.left_edge,
                ↪ t_to_close_right.right_edge)
            new_t.left_edge.trapezoid = new_t
            new_t.right_edge.trapezoid = new_t
    if umps:
        umps.close()
    return finished_trapezoids
```

Der letzte Schritt, die Triangulierung der unimonotonen Polygone mit Algorithmus 8, wird durch die folgende Funktion implementiert.

```
def triangulate_unimonotone(ump: UnimonotonePolygon):
    """Trianguliert ein UMP nach Adi."""
    for i, v in enumerate(ump.vertices[:-1]):
        if ump.vertices[i + 1] not in v.neighbors:
            v.add_internal_edge_to(ump.vertices[i + 1])
    if ump.vertices[0] not in ump.vertices[-1].neighbors:
        ump.vertices[-1].add_internal_edge_to(ump.vertices[0])
    num_vertices = len(ump.vertices)
    cur = 1

    def prev(i):
        return (i - 1) % num_vertices

    def nex(i):
        return (i + 1) % num_vertices

    stack = []
    while num_vertices >= 3:
        if polygons.is_reflex_angle(ump.vertices[prev(cur)], ump.
            ↪ vertices[cur], ump.vertices[nex(cur)],
```

```
                                        # Laufrichtung hängt von Lage
                                            ↪  der Hauptkante des
                                            ↪ UMPs ab
                                        (polygons.Winding.CLOCKWISE,
                                            ↪ polygons.Winding.
                                            ↪ COUNTERCLOCKWISE)[
                                            polygons.left_of(*ump.
                                                ↪ main_edge, ump.
                                                ↪ vertices[1])]):
                    stack.append(ump.vertices[cur])
                    cur = nex(cur)
                else:
                    if ump.vertices[prev(cur)] not in ump.vertices[nex(
                        ↪ cur)].neighbors:
                        ump.vertices[prev(cur)].add_internal_edge_to(ump.
                            ↪ vertices[nex(cur)])
                    ump.vertices.pop(cur)
                    if stack:
                        cur = ump.vertices.index(stack.pop())
                    num_vertices -= 1
```

Die letzte Funktion kombiniert die Schritte.

```
def triangulate_fournier(p: Polygon):
    p.eliminate_horizontals()  # stellt Abwesenheit von gleichen
        ↪ y-Koordinaten sicher
    umps = UnimonotoneBuilder()
    trapezoidize_fournier(p, umps)
    for ump in umps.finished:
        triangulate_unimonotone(ump)
    p.unrotate()
```

A.7 seidel.py

Diese Datei enthält den in Abschnitt 4.5 beschriebenen Algorithmus. Zunächst wird auch hier eine Trapezstruktur definiert, da die Trapeze für diesen Algorithmus im Gegensatz zum Algorithmus nach Fournier und Montuno ihre Nachbarn kennen müssen und jede ihrer Begrenzungen im Unendlichen liegen kann.

Listing A.5: `seidel.py`

```
"""Triangulierung nach Seidel in erwarteter Zeit O(n log* n)."""
from enum import Enum, auto
from typing import Union, Tuple, MutableMapping, List, Optional,
    ↪ Set
import operator
import itertools
import random
import math
from polygons import Polygon, Vertex
import polygons
```

```
from fournier import UnimonotonePolygon
import fournier


class Trapezoid:

    def __init__(self):
        self.left: Optional[Tuple[Vertex, Vertex]] = None
        self.right: Optional[Tuple[Vertex, Vertex]] = None
        self.top_neighbors: List[Trapezoid] = []
        self.bottom_neighbors: List[Trapezoid] = []
        self.top: Vertex = Vertex(0, float("inf"))
        self.bottom: Vertex = Vertex(0, float("-inf"))
        self.sink: Optional["Node"] = None
        """zugehöriger Blattknoten in Suchstruktur"""

    def is_fully_defined(self):
        return (self.left is not None and self.right is not None
                and math.isfinite(self.top.y) is not None and
                    ↪ math.isfinite(self.bottom.y))

    def is_inside_polygon(self):
        return (self.is_fully_defined()
                and max(self.left).neighbors[0] == min(self.left)
                    ↪   # linke Kante läuft von unten nach oben
                and min(self.right).neighbors[0] == max(self.
                    ↪ right))  # rechte Kante läuft von oben nach
                    ↪  unten

    def diagonal(self):
        """Gibt untere Ecke des Trapezes zurück, falls es eine
            ↪ innere Diagonale hat und im Polygon liegt."""
        return (self.bottom if self.is_inside_polygon()
                                and not ((self.top in self.left
                                    ↪ and self.bottom in self.left
                                    ↪ )
                                        or (self.top in self.
                                            ↪ right and self.
                                            ↪ bottom in self.
                                            ↪ right))
                else None)

    def __str__(self):
        return f"left: {self.left if self.left is None else ', '.
            ↪ join(str(v) for v in self.left)}, right: {self.
            ↪ right if self.right is None else ', '.join(str(v)
            ↪ for v in self.right)}, top: {self.top}, bottom: {
            ↪ self.bottom}"
```

Dann werden Klassen für die in Abschnitt 4.5.2 definierte Suchstruktur definiert. Die verschiedenen Knotentypen sind dabei Instanzen der

gleichen Klasse und speichern ihren momentanen Typ, was die Umwandlung in verschiedene Typen leicht ermöglicht.

```
class NodeType(Enum):
    X = auto()
    Y = auto()
    SINK = auto()


class QueryError(Exception):
    pass


class Node:
    """Klasse für die Knoten in der Suchstruktur."""
    def __init__(self, type_: NodeType, key: Union[Tuple[Vertex,
        ↪ Vertex], Vertex, Trapezoid]):
        self.type = type_
        self.key = key
        self.child1 = None  # x node: links, y node: unter
        self.child2 = None  # x node: rechts, y node: über

    def get_child(self, v: Vertex) -> "Node":
        if self.type is NodeType.X:
            assert isinstance(self.key, tuple), "malformed x node
                ↪ "
            return self.child1 if polygons.left_of(*self.key, v)
                ↪ else self.child2
        if self.type is NodeType.Y:
            assert isinstance(self.key, Vertex), "malformed y
                ↪ node"
            return self.child2 if v > self.key else self.child1
        if self.type is NodeType.SINK:
            raise QueryError("sink cannot have children")
        assert False, "nonexistent node type"

    def query(self, v: Vertex) -> "Node":
        """Finde Ecke in Suchstruktur."""
        if self.type is NodeType.SINK:
            assert isinstance(self.key, Trapezoid), "malformed
                ↪ sink node"
            return self
        return self.get_child(v).query(v)
```

Die Klasse `TrapezoidationWithQueryStructure` verwaltet die Suchstruktur. Ihre Methode `insert_segment` implementiert den Algorithmus 9.

```
class NodeType(Enum):
    X = auto()
    Y = auto()
    SINK = auto()


class QueryError(Exception):
```

```
    pass


class Node:
    """Klasse für die Knoten in der Suchstruktur."""
    def __init__(self, type_: NodeType, key: Union[Tuple[Vertex,
        ↪ Vertex], Vertex, Trapezoid]):
        self.type = type_
        self.key = key
        self.child1 = None  # x node: links, y node: unter
        self.child2 = None  # x node: rechts, y node: über

    def get_child(self, v: Vertex) -> "Node":
        if self.type is NodeType.X:
            assert isinstance(self.key, tuple), "malformed x node
                ↪ "
            return self.child1 if polygons.left_of(*self.key, v)
                ↪ else self.child2
        if self.type is NodeType.Y:
            assert isinstance(self.key, Vertex), "malformed y
                ↪ node"
            return self.child2 if v > self.key else self.child1
        if self.type is NodeType.SINK:
            raise QueryError("sink cannot have children")
        assert False, "nonexistent node type"

    def query(self, v: Vertex) -> "Node":
        """Finde Ecke in Suchstruktur."""
        if self.type is NodeType.SINK:
            assert isinstance(self.key, Trapezoid), "malformed
                ↪ sink node"
            return self
        return self.get_child(v).query(v)
```

Die folgende Funktion führt die Trapezoidisierung nach Algorithmus 10 durch.

```
def trapezoidize(p: Polygon):
    if not polygons.is_simple(p):
        raise ValueError("polygon is not simple")
    chain = p.chains[0]
    for v in chain:
        v.prev = v.neighbors[0]
        v.next = v.neighbors[1]
    segments = [(chain[i], chain[i+1]) for i in range(len(chain)
        ↪ - 1)] + [(chain[-1], chain[0])]
    random.shuffle(segments)
    traps = TrapezoidationWithQueryStructure()
    for seg in segments:
        traps.insert_segment(seg)
    return traps
```

Die nächsten beiden Funktionen implementieren die Triangulierung des trapezoidisierten Polygons nach Algorithmus 11. Die erste Funk-

tion davon ist ein Wrapper um die Funktion zur Triangulierung unimonotoner Polygone aus `fournier.py`, da die unimonotonen Polygone hier zunächst nicht als explizite Datenstrukturen erzeugt werden.

```
def triangulate_unimonotone(first: Vertex):
    vertices = [first]
    v = first.next
    while v != first:
        vertices.append(v)
        v = v.next
    top = max(vertices, key=operator.attrgetter("y"))
    i = vertices.index(top)
    if top.prev < top.next:
        ump = UnimonotonePolygon((top, top.prev), top)
        ump.vertices.extend(vertices[i + 1:] + vertices[:i])
    else:
        ump = UnimonotonePolygon((top, top.next), top)
        ump.vertices.extend(vertices[i - 1::-1] + vertices[:i
            ↪ :-1])
    fournier.triangulate_unimonotone(ump)


def triangulate_trapezoidized(first: Vertex):
    curr = first
    while not curr.done:
        curr.done = True
        bottom: Vertex = curr.trapezoid.diagonal() if curr.
            ↪ trapezoid is not None else None
        if bottom is not None:
            save_next = curr.next
            save_prev = bottom.prev
            curr.next = bottom
            bottom.prev = curr
            triangulate_trapezoidized(bottom)
            curr.trapezoid = None
            curr.done = False
            bottom.done = False
            curr.next = save_next
            bottom.prev = save_prev
            bottom.next = curr
            curr.prev = bottom
            triangulate_trapezoidized(curr)
            return
        else:
            curr = curr.next
    triangulate_unimonotone(first)
```

Schließlich kombiniert die Funktion zur Triangulierung die obigen Schritte. Auf die Implementierung des Verfolgens der Polygonkette durch die Trapezoidisierung wurde in [14] kaum eingegangen, lediglich die Möglichkeit, dies zu tun, wurde erwähnt. Daher wurde dies bei der Implementierung hier vernachlässigt.

```
def seidel_triangulate(p: Polygon):
    p.eliminate_horizontals()
    traps = trapezoidize(p)
    for t in traps.trapezoids:
        if t.is_inside_polygon():
            t.top.trapezoid = t
        else:
            t.top.trapezoid = None
    triangulate_trapezoidized(p.vertices[0])
    p.unrotate()
```

A.8 main.py

Diese Datei bietet ein Kommandozeileninterface zur Ausführung der Algorithmen.

Listing A.6: main.py

```
import argparse
import sys
import polygons
import fan
import earclipping
import fournier
import seidel

if __name__ == "__main__":
    parser = argparse.ArgumentParser(description="Trianguliere
        ↪ ein gegebenes Polygon.")
    parser.add_argument("polygon", help="Textdatei, die ein
        ↪ Polygon beschreibt", type=argparse.FileType("r"))
    parser.add_argument("algorithmus", help="
        ↪ Triangulierungsalgorithmus",
                        choices=["fan", "earclipping", "fournier"
                            ↪ , "seidel"])
    parser.add_argument("output", help="Bilddatei, in die das
        ↪ triangulierte Polygon geschrieben wird",
                        type=argparse.FileType("wb"))

    if len(sys.argv) == 1:
        parser.print_help()
        sys.exit(1)
    args = parser.parse_args()
    with args.polygon as f:
        p = polygons.read_polygon(f)
    if args.algorithmus == "fan":
        fan.fan_triangulation(p)
    elif args.algorithmus == "earclipping":
        earclipping.ear_clipping(p)
    elif args.algorithmus == "fournier":
        fournier.triangulate_fournier(p)
```

```
        else:
30          seidel.seidel_triangulate(p)

        polygons.draw_polygon(p).get_figure().savefig(args.output)
```

LITERATUR

[1] Soehadi Adi. „Modification of Fournier and Montuno's triangulation algorithm for simple polygons“. In: *Department of Computer Science, Oklahoma State University* (1991).

[2] N. M. Amato, M. T. Goodrich und E. A. Ramos. „A Randomized Algorithm for Triangulating a Simple Polygon in Linear Time“. In: *Discrete & Computational Geometry* 26.2 (2001), S. 245–265. ISSN: 1432-0444. DOI: 10.1007/s00454-001-0027-x. URL: https://doi.org/10.1007/s00454-001-0027-x.

[3] Bart Braden. „The Surveyor's Area Formula“. In: *The College Mathematics Journal* 17.4 (1986), S. 326–337. DOI: 10.1080/07468342.1986.11972974. URL: https://doi.org/10.1080/07468342.1986.11972974.

[4] Bernard Chazelle. „Triangulating a simple polygon in linear time“. In: *Discrete & Computational Geometry* 6.3 (1991), S. 485–524. DOI: 10.1007/BF02574703. URL: https://doi.org/10.1007/BF02574703.

[5] T.H. Cormen, T.H. Cormen, C.E. Leiserson, Inc Books24x7, Massachusetts Institute of Technology, MIT Press, R.L. Rivest, C. Stein und McGraw-Hill Publishing Company. *Introduction To Algorithms*. Introduction to Algorithms. MIT Press, 2001. ISBN: 9780262032933. URL: https://books.google.de/books?id=NLngYyWFl_YC.

[6] David Eberly. *Triangulation by Ear Clipping*. 2015. URL: https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf (besucht am 29. 08. 2021).

[7] David Eppstein. „Counting Polygon Triangulations is Hard“. In: *Discrete & Computational Geometry* 64.4 (2020), S. 1210–1234. DOI: 10.1007/s00454-020-00251-7.

[8] Alain Fournier und Delfin Montuno. „Triangulating Simple Polygons and Equivalent Problems“. In: *ACM Transactions on Graphics* 3 (Apr. 1984), S. 153–174. DOI: 10.1145/357337.357341.

[9] Grant Jenks. *Python Sorted Containers*. URL: http://www.grantjenks.com/docs/sortedcontainers/ (besucht am 26. 08. 2021).

[10] Michael Luby, Alistair Sinclair und David Zuckerman. „Optimal speedup of Las Vegas algorithms“. In: *Information Processing Letters* 47.4 (1993), S. 173–180. ISSN: 0020-0190. DOI: https://doi.org/10.1016/0020-0190(93)90029-9. URL: https://www.sciencedirect.com/science/article/pii/0020019093900299.

[11] *Matplotlib: Python plotting - Matplotlib 3.4.3 documentation*. URL: https://matplotlib.org/ (besucht am 05. 09. 2021).

[12] G. H. Meisters. „Polygons Have Ears“. In: *The American Mathematical Monthly* 82.6 (1975), S. 648–651. ISSN: 00029890, 19300972. URL: http://www.jstor.org/stable/2319703.

[13] *NumPy*. URL: https://numpy.org/ (besucht am 05. 09. 2021).

[14] Raimund Seidel. „A simple and fast incremental randomized algorithm for computing trapezoidal decompositions and for triangulating polygons“. In: *Computational Geometry* 1.1 (1991), S. 51–64. ISSN: 0925-7721. DOI: https://doi.org/10.1016/0925-7721(91)90012-4. URL: https://www.sciencedirect.com/science/article/pii/0925772191900124.

[15] *Welcome to Python.org*. URL: https://www.python.org/ (besucht am 05. 09. 2021).

[16] K. Wendland und A. Werner. *Facettenreiche Mathematik: Einblicke in die moderne mathematische Forschung für alle, die mehr von Mathematik verstehen wollen*. Mathematik Populär. Vieweg+Teubner Verlag, 2011. ISBN: 9783834881731. URL: https://books.google.de/books?id=6zYpBAAAQBAJ.

[17] Python Wiki. *TimeComplexity*. URL: https://wiki.python.org/moin/TimeComplexity (besucht am 20. 08. 2021).

COLOPHON

This document was typeset using the typographical look-and-feel `classicthesis` developed by André Miede and Ivo Pletikosić. The style was inspired by Robert Bringhurst's seminal book on typography "*The Elements of Typographic Style*". `classicthesis` is available for both LaTeX and LyX:

https://bitbucket.org/amiede/classicthesis/

Happy users of `classicthesis` usually send a real postcard to the author, a collection of postcards received so far is featured here:

http://postcards.miede.de/

Thank you very much for your feedback and contribution.

Final Version as of 23. Oktober 2023 (classicthesis v4.6).