Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

GUI Graphical User Interface

ECA Ear-Clipping-Algorithmus

DT Delaunay Triangulation

CDT Constrained Delaunay Triangulation

TM Turingmaschine

CPU Central Processing Unit

FIST fast industrial-strength triangulation framework

SP Steiner Punkte

1 Einleitung

Die größte technische Wende nach der industriellen Revolution war die digitale Revolution gegen Ende des 20. Jahrhunderts.[1] Eingeleitet durch die Entwicklung des Mikrochips und der damit verbunden Verbreitung des Computers in allen Lebensbereichen führte sie zu einer dramatischen Veränderung. Nicht nur in der Industire und Produktion fanden diese tiefgreifenden Umbrüche statt, welche sich in flexibler Automatisierung äußerten, sondern auch in anderen Bereichen. So wurde die Entwicklung des Internets durch vernetzte Rechner möglich. Als dann Computer nicht mehr nur in der Forschung und für die automatische Produktion genutzt wurden, sondern auch für den täglichen Gebrauch im Büro und daheim etablierten, benötigte man grafische Benutzeroberflächen und Betriebssysteme. Doch dort machte die Entwicklung nicht halt. Auch die Unterhaltungsbranche erfuhr mit Videospielen eine Revolution, welche ebenso auf Computergrafik angewiesen ist wie ein einfaches Graphical User Interface (GUI).

Zu Beginn beschränkte sich die Darstellung auf sogenannte ASCII-Art, bei der übliche Zeichen aus dem ASCII-Alphabet benutzt wurden, um komplexe Bilder zu erzeugen. Da dies jedoch nicht genügte, um Flächen und Objekte lückenlos darzustellen, bedurfte es einer Innovation. Obwohl es für Menschen einfach ist, Flächen als Ganzes zu betrachten und Polygone in unterschiedlichster Komplexität zeichnerisch darzustellen, ist es für Computer nicht so einfach, diese zu speichern, geschweige denn darzustellen. Flächen und dreidimensionale Objekte kann man, so die Idee, über ihre Eckpunkte (Vertices) und die dazwischen liegenden Kanten (Edges) zu repräsentieren. Man erzeugt also ein Polygonnetz, welches den Körper abbildet. Dabei ist die Wahl des Polygons zunächst irrelevant. So könnte man beispielsweise einen Würfel, abhängig von der Definition der Kanten, aus Quadraten oder Dreiecken aufbauen.[2] In der Praxis sind Polytope und Polygone jedoch meistens unregelmäßig, ergeben sie sich doch zum Beispiel aus Umgebenungsscans mit einem Laser-Scanner oder der Oberfläche einer Videospielfigur. Es bietet sich in solchen Fällen nicht an, regelmäßige Polygone, wie Quadrate oder Rechtecke, als Grundlage für das Polygonnetz zu nutzen.

Eine geeignete Methode, um diese komplexen Polygone für Computer effizient darzustellen, ist die Nutzung von Dreiecken als primitive Form für die Zerlegung. Dieses Vorgehen bezeichnet man als Triangulation. Diese ist formal die Zerlegung eines topologischen Raumes, hier also eines Polygons, in Simplexe. Das Simplex der zweiten Dimension das Dreieck und damit ist die Triangulation ein Verfahren zur Zerlegung eines Poylgons in Dreiecke. Es sei erwähnt, dass es Computern durchaus möglich ist, Flächen und Körper darzustellen, welche nicht aus Dreiecken bestehen. Dies ist jedoch wesentlich speicher- und rechenaufwendiger, als es bei Dreiecken der Fall ist. Für die Darstellung eines Objektes ließe sich alternativ auch eine sogenannte Punktwolke nutzen. Wie der Name bereits andeutet, wird

das Objekt dabei aus einer großen Menge von Einzelpunkten gebildet. Für einen hohen Detailgrad sind dafür allerdings auch sehr viele Punkte nötig, was den Speicheraufwand stark erhöht. Bei der Verwendung von Dreiecken handelt es sich, vorallem bei runden Objekte, eher um eine Approximation der Form. Eine Kugel wäre dann nicht vollständig rund, sondern würde als Polyeder repräsentiert werden. Dadurch spart man jedoch sehr viel Speicherplatz. Man kann auch hier den Detailgrad steigern, indem man die Anzahl der Dreiecke erhöht und ihre Größe reduziert. Da Dreiecke Flächen sind, benötigt man von ihnen jedoch eine geringere Anzahl, um ein Objekt darzustellen, als wenn dies mittels einer Punktwolke geschieht.

Um eine Triangulation per Computer durchzuführen, bedarf es eines Algorithmus, der das Verfahren beschreibt. Von diesen gibt es viele verschiedene, welche unterschiedlichste Herangehensweisen nutzen. Hier seien der Ear-Clipping-Algorithmus (ECA) und die monotone Triangulation als Beispiel für Algorithmen genannt. Des weiteren sollen hier die Delaunay Triangulation (DT) als Triangulation mit besonderen Eigenschaften und das Voronoi-Diagramm als duale Form zur DT angeführt werden. Diese unterscheiden sich erheblich in ihrer Komplexität und Effizienz. Mit einer Laufzeit von $O(n^2)$[5], ist der ECA bei weitem nicht so effizient wie beispielsweise ein Algorithmus zur Erzeugung einer DT mit $O(n \log n)$. Für den ECA spricht jedoch seine relative Einfachheit im Vergleich zu anderen Algorithmen.

In dieser Arbeit soll jedoch nicht die Laufzeitoptimierung im Vordergrund stehen, sondern die Anschaulichkeit. Sie ist ein wichtiger Punkt, wenn es um Didaktik geht. Anschauliche Lehrmaterialen förderen das Verständnis und bieten Interaktivität. So hat diese Ausarbeitung zum Ziel, eine interaktive Visualisierung für die Triangulation von Polygonen zu schaffen. Dafür ist der ECA aufgrund seiner relativen Einfachheit gut geeignet. Er lässt sich schrittweise durchlaufen und ist somit sehr anschaulich, da in jedem Schritt ein Dreieck der Triangulierung erzeugt wird. Es liegt in der Natur dieses Algorithmus, dass Uneindeutigkeiten auftreten, was die Auswahl des nächsten Dreiecks angeht. Diese führen zum zweiten wichtigen Punkt in dieser Arbeit - der Interaktivität. Der Nutzer der Visualisierungssoftware soll interaktiv entscheiden können, welches das nächste Dreieck ist, welches bearbeitet wird. Er beeinflusst somit direkt das endgülige Resultat. Neben dem dirketen Eingriff des Nutzers sollen auch Heuristiken zu Einsatz kommen, um diese Auswahl zu treffen. Beispielsweise kann hierfür die Größe des Dreiecks im Bezug auf seinen Flächeninhalt genutzt werden oder auch die Innenwinkel. Es ist angestrebt, dass die Nutzerauswahlen ausgewertet und in eine Heuristik überführt werden. Um dies zu bewerkstelligen, soll die Qualität der Triangulation mittels verschiedener Metriken beurteilt werden. Hierfür kann ein Vergleich zum Voronoi-Diagramm ebenso wie zum Beispiel die Anzahl der

sogenannten *Slivers*[3] betrachtet werden. Letztere führen in Anwendungen der Computergrafik oft zu Fehlern, welche vermieden werden sollten.

Es soll somit nicht nur eine Visualisierung für Triangulationen geschaffen werden, sondern es sollen auch die Auswirkungen einfahcer Heuristiken auf die Qualität dieser Zerlegungen betrachtet werden. Es steht dabei, wie bereits erwähnt, nicht die Laufzeit des Algorithmus im Vordergrund, welche üblicherweise Ziel der Optimierung ist.

2 Theoretische Grundlagen

2.1 Polygone

2.1.1 Definition

Ein geschlossener Streckenzug, also eine Folge von Strecken, welche jeweils einen Endpunkt mit ihrem Vorgänger bzw. Nachfolger gemeinsam haben, bilden ein **Polygon**.

Dabei ist es wichtig, dass die Anzahl von Strecken endlich ist. "Das Polygon, [zu Deutsch Vieleck], ist also eine durch eine Folge von Strecken begrenzte ebene Fläche."[10] Das einfachste Beispiel hierfür ist ein Dreicke. Es besitzt die Eckpunkte A, B und C und wir daher vom Streckenzug aus den Strecken $\overline{AB}$, $\overline{BC}$, $\overline{CA}$ begrenzt (s. Abbildung 1). Mit genau diesem Prinzip lassen sich beliebig komplexe Polygone erzeugen und beschreiben. Die Strecken werden auch als **Seiten** und die Endpunkte dieser Strecken als **Ecken** bezeichnet. Es sei angemerkt, dass Kreise, obwohl sie ebenfalls ebene Flächen sind, keine Polygone sind. Das folgt daraus, dass Kreise weder Ecken noch eine Begrenzung aus Strecken besitzen.

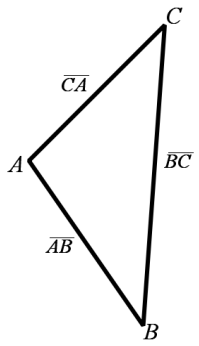


Abbildung 1: Dreieck mit Ecken A, B, C als Beispiel für ein Polygon

2.1.2 Klassifikation von Polygonen

Es ist denkbar, dass sich die Seiten des Polygons schneiden oder berühren. Man bezeichnet dieses Polygon als überschlagen.[10] Des weiteren kann man Polygone in regulär und nicht regulär unterteilen. Ein Polygon mit den n Seiten $a, b, c, \ldots$ und den Innenwinkeln $\alpha, \beta, \gamma, \ldots$ heißt regulär, wenn

$$a = b = c = \ldots \text{ und } \alpha = \beta = \gamma = \cdots$$

gilt. In einem regelmäßigen Polygon sind demnach alle Seiten zueinander kongruent und alle Winkel gleich groß. [11]

Eine weitere Unterteilungsmöglichkeit lautet wie folgt. Ein Polygon heißt **konvex**, wenn für alle Innenwinkel α_i ($i \in \mathbb{N}$) gilt: $\alpha_i < 180°$ Anderenfalls heißt es **konkav**. [12]

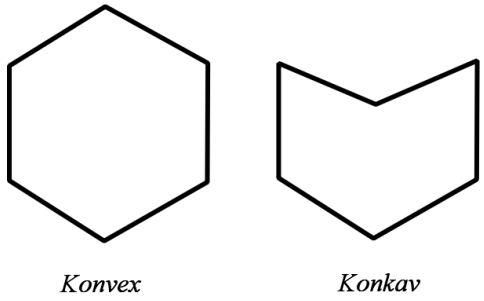


Abbildung 2: Zwei Secksecke: links konvex, rechts konkav

2.1.3 Diagonalen

Außer des Streckenzuges, welcher die äußere Grenze des Polygons bildet, kann man im Polygon selbst auch weitere Strecken definieren, welche dann als **Diagonalen** bezeichnet werden. Mittels dieser Diagonalen ist es möglich, jedes Polygon in Dreiecke zu zerlegen. Das wird in den nächsten Kapiteln noch näher erläutert, da dieser Sachverhalt die Grundlage für sämtliche Zerlegungsalgorithmen darstellt.

2.1.4 Ear und Ear Tips

Für den in Kapitel 3.3 beschriebenen ECA ist es der Begriff des **Ear** (Ohr) relevant. Ein Dreieck, welches aus drei aufeinanderfolgenden Ecken $v_{i_0}, v_{i_1}, v_{i_2}$ des Polygons gebildet wird, ohne dass andere Ecken innerhalb dieses Dreiecks liegen oder dass der äußere Streckenzug des Polygons durch die Seiten des Dreiecks geschnitten wird, nennt man Ear. Dabei bildet die Strecke $\{v_{i_0}, v_{i_2}\}$ eine Diagonale des Polygons. Die Ecke v_{i_1} heißt dann **Ear Tip**.[20]

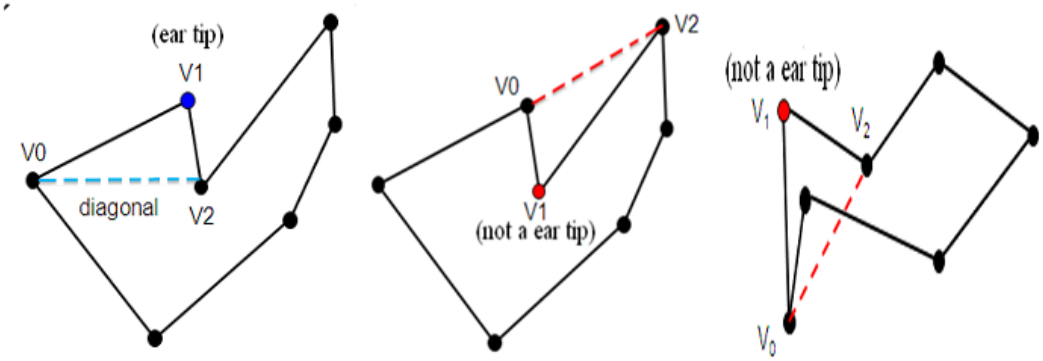


Abbildung 3: Links ist $\triangle v_{i_0}, v_{i_1}, v_{i_2}$ Ear und v_{i_1} Ear Tip. In der Mitte und rechts ist $\triangle v_{i_0}, v_{i_1}, v_{i_2}$ kein Ear, da $\{v_{i_0}, v_{i_2}\}$ keine Diagonale ist. [9]

2.1.5 Sätze über Polygone

Für Polygone gibt es einige Erkenntnisse, welche für die allgemeine Strukturanalyse von eben diesen oder auch für die Triangulation mittels ECA von Bedeutung sind.

Satz 1 (Jordan'scher Kurvensatz):
In der euklidischen Ebene $\mathbb{R}^2$ zerlegt jede geschlossene Jordan-Kurve $C \subset \mathbb{R}^2$ deren Komplement $\mathbb{R}^2 \setminus C$ in zwei disjunkte Gebiete, deren gemeinsamer Rand die Jordankurve C ist und deren Vereinigung zusammen mit die ganze Ebene $\mathbb{R}^2$ ausmacht.
Genau eines der beiden Gebiete, das sogenannte **Innengebiet**, ist eine beschränkte Teilmenge von $\mathbb{R}^2$.
Das andere dieser beiden Gebiete ist das sogenannte **Außengebiet** und unbeschränkt. [17]

Satz 2 (Dreieckszerlegung:)
Jedes Polygon P mit n Ecken kann mittels Hinzunahme von null oder mehr Diagonalen vollständig in Dreiecke zerlegt werden. [9]

Satz 3 (Anzahl der Diagonalen:)
Jede Triangulation eines Polygons P mit n Ecken besteht nutzt $(n-3)$ Diagonalen und besteht aus $(n-2)$ Dreiecken. [9]

Satz 4 (Two Ears Theorem:)
Jedes Polygon P mit $n \geq 4$ Ecken besitzt mindestens zwei nicht überlappende Ears. [19]

2.1.6 Polytope

Zuletzt sei an dieser Stelle angemerkt, dass ein Polygon die zweidimensionale Ausprägung des topologischen Begriffs des **Polytops** ist. Betrachtet man die räumlichen Dimensionen null bis vier in aufsteigender Reihenfolge, so sind ein Punkt, eine Strecke, ein Quadrat, ein Würfel und ein Tesserakt. Dies ist in der nachstehenden Abbildung zu sehen.

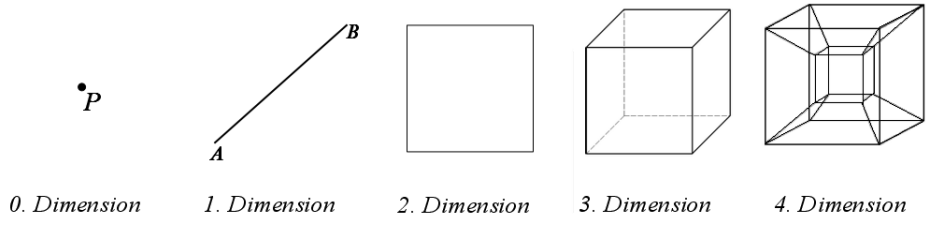


Abbildung 4: Beispiele für Polytope der Dimensionen 0 bis 4

2.2 Simplexe und Triangulation

Wie bereits angesprochen, kann man durch Hinzufügen von Diagonalen ein Polygon in Dreiecke oder allgemeiner in Unterpolygone zerlegen. Diese Eigenschaft macht sich die **Triangulation** zu nutze. Allgemein beschreibt der Begriff Triangulation die Zerlegung eines topologischen Raumes in **Simplexe**.[13] Der topologische Raum ist in diesem Fall das Polygon, welches durch einen Streckenzug gebildet wird.

Als Simplex bezeichnet man das einfachste Polygon einer Dimension.[14] Für die nullte Dimension ist das trivialerweise der Punkt. Da keine räumliche Ausdehnung möglich ist, ist der begrenzende Streckenzug hier nur der Punkt selbst. In der ersten Dimension, in welcher Objekte eine Länge, aber keine Breite besitzen, ist der Streckenzug eine einzelne Strecke. Diese ist somit auch das Simplex dieser Dimension. Für die zweite Dimension ist nun das Dreieck das Simplex. Es ist die Fläche, welche aus den wenigsten Punkten, verbunden duch Strecken, erzeugt werden kann und daher das einfachste Polygon dieser Dimension.

Wie bereits beschrieben, kann jedes komplexere Polygon so durch Diagonalen zerlegt werden, dass es vollständig von Dreiecken repräsentiert wird. Das ist besonders günstig für eine Bearbeitung durch Computer, da ein Dreieck immer eindeutig durch seine drei Eckpunkte beschrieben wird. Wie später noch zu sehen sein wird, kann man in einem Polygon sehr einfach Dreiecke dadurch erzeugen, dass man Diagonalen einfügt. Man kann spezielle Dreiecke, die sogenannten Ears, sogar durch Hinzufügen von nur einer Diagonalen generieren, was algoritmisch gut beschreibbar ist.

Da in der Praxis nicht nur zweidimensionale sondern auch dreidimensionale Objekte eine Rolle spielen, stellt sich die folgende Frage. Kann man diese 3D-Objekte nicht auch in ebenfalls dreidimensionale Simplexe zerlegen?

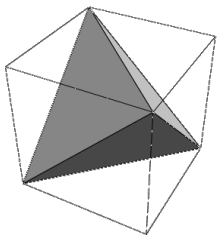


Abbildung 5: Zerlegung eines Würfels in vier Tetraeder [18]

Die Antwort ist ja, jedoch ist das nicht sonderlich nützlich. Natürlich existiert in der dritten Dimension auch ein Simplex. Dieses ist der Tetraeder. Man kann auch jedes Polytop dieser Dimension in Tetraeder zerlegen. Diese Zerlegung benötigt man durchaus für spezielle Anwendungen. Wöllte man das Innere eines Objektes durch die Zerlegung ebenfalls erhalten, beispielsweise einen Vollwürfel aus Holz in der Realität zersägen, dann wäre eine Tetraederzerlegung notwendig. Ein Beispiel für diese Art der Zerlegung ist in Abbildung 5 zu sehen. Man benötigt diese Form der Zerlegung zwar für einige Anwendungen, wie etwa bei der finite Elemente Methode, jedoch soll das nicht Gegenstand der Betrachtung in dieser Arbeit sein. Man kann die räumlich orientierte Oberfläche eines Würfels topologisch isomorph zu einem Würfelgitter aus quadratischen

Flächen beschrieben. Diese Gesamtfläche lässt sich dann wiederum in Dreiecke zerlegen. Somit lässt sich auch die Oberfläche dreidimensionaler Objekte durch eine Triangulierung beschreiben. Das ist besonders gut geeigent für Computer, da man nur einen einzigen Algorithmus zur Bearbeitung von Flächen und Körpern benötigt, wenn man diese in Dreiecke zerlegen möchte.

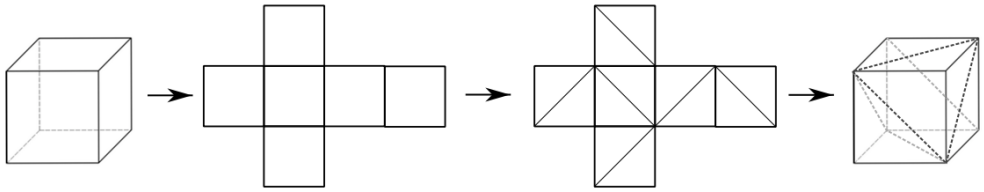


Abbildung 6: Zerlegung der Oberfläche eines Würfels in Dreiecke mittels Würfelgitter

2.3 Slivers

Bei Triangulationsalgorithmen wie dem ECA liegt der Fokus zunächst nicht in der Qualität der Zerlegung. Natürlich gibt es verbesserte Versionen wie beispielsweise in Kapitel 3.1 beschreiben wird. In jedem Fall sind sogenannte **Slivers** jedoch ein negativer Einflussfkator, wenn es um qualitative Gesichtspunkte geht. Dreiecke haben in der Computergrfaik eine bestimmte Eigenschaft. Legt man eine Scanline durch ein Dreieck, dann schneitet diese die Kanten des Dreiecks in zwei Punkten. Diese zwei Schnittpunkte werden von zwei unterschiedlichen Pixeln auf dem Bildschirm repräsentiert. Somit lässt sich definieren, wo eine Fläche beginnt und wo sie endet und dies auf dem Bildschirm darstellen. Bei Slivers ist das nicht der Fall. Ihre Innenfläche ist so schmal, dass die beiden Schnittpunkte der Scanline mit den Kanten des Dreiecks auf den selben Pixel fallen. [4] Das führt in letzter Instanz zu Grafikfehlern.

Der Begriff Sliver beschränkt sich nicht nur auf Dreiecke. Auch andere Simplexe wie Tetraeder können Slivers sein. Sie sind so flach, dass auch hier Darstellungsfehler entstehen. Zusätzlich dazu gibt es noch den verwandten Begriff der **Neadle**, der einen sehr schmalen aber auch sehr spitzen Tetraeder bezeichnet.[3]

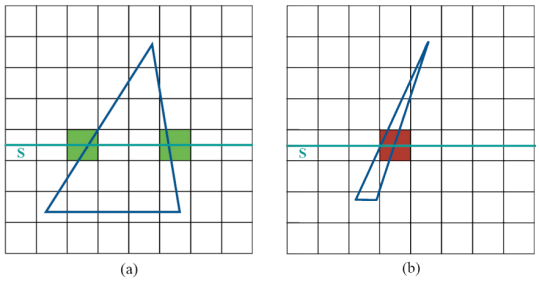


Abbildung 7: (a) Scanline mit zwei separaten Schnittpunktpixeln (b) Sliver mit nur einem Pixel für beide Schnittpunkte

2.4 Der Traditionelle Ear-Clipping-Algorithmus

Der zentrale Fokus in dieser Arbeit liegt auf dem ECA. Dieser wurde von Meister in seiner Abhandlung *Polygons have ears* [20] in seiner ursprünglichen Form beschrieben. Der Algorithmus bestimmt Dreiecke, welche die Eigenschaft eines Ears erfüllen, fügt die dafür nötige Diagonale in eine Liste ein und löscht den Ear Tip aus der Liste aller noch nicht bearbeiten Punkte. Dies wird solange wiederholt, bis das Restpolygon nur noch aus drei Punkten besteht. Zuletzt wird die Liste der Diagonalen ausgegeben, da diese die Triangulation des Polygons erzeugt. Der Algoritmus ist im Folgenden in Pseudocode dargestellt.

Algorithmus 1: Traditionelles Ear-Clipping [8]

Eingabe: Polygon P mit n Ecken in einer Liste L
Ausgabe: Liste D mit $n-3$ Diagonalen, die eine Triangulierung bilden.
Schritt 1: Sei $D := \emptyset$ Liste der Diagonalen.
Schritt 2: **while** $|L| > 3$ **do**
(a) Finde ein Ear v_{i-1}, v_i, v_{i+1}
(b) $D := D \cup \{v_{i-1}v_{i+1}\}$
(c) $L := L \setminus v_i$
endwhile
Schritt 3: Ausgabe von D als triangulierende Diagonalen.

Dieser Algoritmus hat einen Zeitaufwand von $O(n^3)$ mit einem Aufwand von $O(n^2)$ für das Ermitteln des Ear-Status eines Dreiecks. In dieser Formulierung wird nicht auf die Klassifikation eines Ears im speziellen eingegangen. Hierfür beginnt man klassisch beim ersten ersten Punkt in L. Man überprüft ob dieser Punkt v_i konvex ist. Ist das der Fall, dann muss die Strecke $\{v_{i-1}v_{i+1}\}$ die Eigenschaft haben, eine Diagonale von P zu sein. Wenn das ebenfalls zutrifft, dann ist das Dreiecke v_{i-1}, v_i, v_{i+1} ein Ear. Man kann die Klassifikation so darstellen:

Algorithmus 2: Ear Klassifikation

Eingabe: Ecken $v_i, v_{i-1}, v_{i+1} \in L$
Ausgabe: $\triangle v_{i-1}, v_i, v_{i+1}$ ist Ear oder nicht.
Schritt 1: **if** v_i konvex **AND** $\{v_{i-1}v_{i+1}\}$ ist Diagonale von P
Ausgabe $\triangle v_{i-1}, v_i, v_{i+1}$ ist Ear.
else Ausgabe $\triangle v_{i-1}, v_i, v_{i+1}$ ist kein Ear.
endif

Diese Klassifikation könnte man auch zuerst über alle Punkte v_i in L laufen lassen. Man spricht dann von der Klassifikationsphase. Danach kann man in einer zweiten Phase, der Cutting-Phase, Dreicke auswählen, welche die Ear-Eigenschaft erfüllen und sich nicht überschneiden, und diese dann abschneiden. Mit diesen beiden Phasen im Wechsel kann man ebenfalls eine Triangulation erreichen. O'Rourke beschreibt in seinem Buch einen Ansatz, der einige Zeitersparnis bei diesem Algorithmus bewirkt.[22] Anstatt nach dem Abtrennen eines Ear Tip Punkts den Status jedes Eckpunktes erneut zu überprüfen, muss man nur den Status von v_{i-1} und v_{i+1} erneut betrachten. Nur diese beiden Punkte sind nämlich vom Abtrennen von v_i beeinflusst. Somit benötigt man insgesamt nurnoch eine Zeit von $O(n^2)$.[9] In jedem Cutting-Schritt kann man dann zusätzlich entscheiden, welches Dreieck als nächstes ausgewählt werden soll. So könnte man nur die Dreiecke auswählen, welche einer bestimmten Heuristik entsprechen. Beispielsweise könnten so nur Dreiecke gewählt werden, bei denen der kleinste Innenwinkel das Maximum aller aktuell verfügbaren Innenwinkel ist. Auf diese Weise ist es denkbar, dass man Sliver vermeiden könnte. Andere Ansätze sind exemplarisch in Kapitel 3 aufgeführt.

2.5 Preprocessing für Polygone mit Löchern

Neben einfachen Polygonen, wie sie bisher in dieser Arbeit beschrieben worden sind, gibt es auch komplexere Formen. Held bezeichnet sie in seiner Abhandlung über FIST als *multiply-connected polygonal areas*, also als mehrfach verbundene polygoniale Flächen.[24] Der Einfachkeit halber sollen eben diese zusammengesetzten Polygone in dieser Arbeit entweder als **komplexe Polygone** oder als **Polygone mit Loch** bezeichnet werden. Ein solches Loch wird dabei ebenfalls durch einen geschlossenen Streckenzug begrenzt und ist somit selbst ein Polygon. Ein Loch H hat allerdings die Eigenschaft bereits im Inneren eines anderen Polygons P zu liegen, ohne dabei mit dem äußeren Streckenzug von P verbunden zu sein oder diesen zu schneiden.

Da diese beiden Streckenzüge keine Kante besitzen, welche sie verbindet, ergibt sich ein Problem für den ECA, welcher nur mit einem geschlossenen Polygonzug und dessen Ecken arbeitet. Um den ECA dennoch auch auf komlexe Polygone anwenden zu können, bedarf es der Vorbereitung durch einen anderen Algorithmus, um aus den endlich vielen verschiedenen Streckenzügen von P und den i Löchern H_i zu erzeugen. M. Held beschreibt in seiner Arbeit zwei verschiedene Verfahren mit dem selben Grundgedanken. Man fügt dazu zusätzliche Kanten, sogenannte **Brücken** (contour bridges) zwischen je einer Ecke v_i von P und einer Ecke u_j eines Loches H ein. Diese Kante sei dann $\{v_i u_j\} \in E(P')$ Damit man den Streckenzug, der daraus entsteht weiter entlang laufen kann, ohne eine Kante doppelt zu nutzen, muss man die Brücke sozusagen doppelt einfügen. Hierzu erzeugt man Kopien

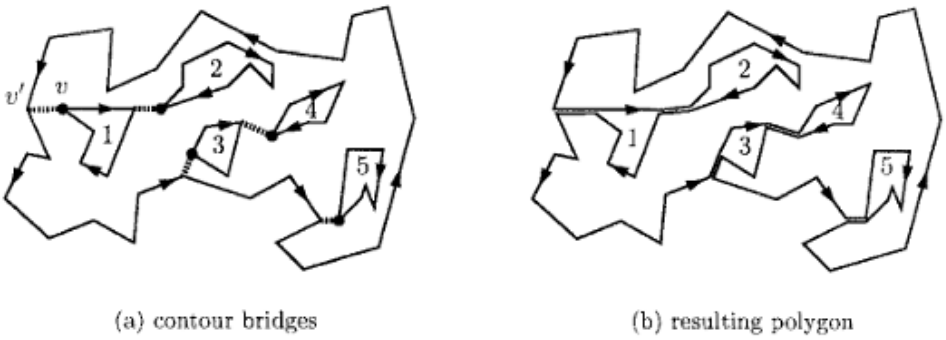


Abbildung 8: (a) Polygon mit mehrerern Löchern. Brücken gestrichelt dargestellt. (b) Resultierendes Polygon (Durchlaufrichtung entlang des Streckenzugs mittels Pfeilen gekennzeichnet)[24]

v_i', u_j' von v_i, u_j und verbindet diese ebenfalls mit einer Brücke, allerdings mit umgekehrter Orientierung, $\{u_j' v_i'\} \in E(P')$. P' sei dabei das neu entstandene Polygon aus der Vereinigung $E(P) \cup E(H) \cup \{\{v_i u_j\}, \{u_j' v_i'\}\}$.

Der erste Ansatz, um ein Loch H mit dem Polygon P zu verbinden, beruht auf einer vollständigen Enumeration. Man bildet alle Paare von Punkten (v_i, u_j) und verbindet diese mit einer vorläufigen Strecke. Dann berechnet man die Länge aller dieser Strecken und wählt die kürzeste von ihnen als Brücke. Dabei muss überprüft werden, dass die gewählte Strecke keine Kante von P oder H schneidet. Sollte es einen Schnittpunkt geben, wird die zweit längste Strecke überprüft und so weiter. Bei diesem Verfahren kommt man auf eine Komplexität von $O(n^3)$ wobei es $O(n^2)$ viel mögliche Strecken gibt und man lineare Zeit benötigt, um eine Brücke als valide zu klassifizieren. Hat man mehr als ein Loch, so beginnt man mit einem beliebigen und verbindet es mit P und wählt dann das nächste aus, bis alle Löcher und P zum Polygon P' vereinigt worden sind.

Da diese Laufzeit sehr unzufriedenstellend ist, schlägt Held ein zweites Verfahren vor, welches er auch bei der Implementierung von FIST verwendet. Hierfür bestimmt man für jedes Loch H_j den am weitesten links liegenden Eckpunkt u_{left}. Dann werden die Löcher nach eben diesen Punkten sortiert, von links nach rechts in aufsteigender Reihenfolge. Eine solche Sortierung ist in Abbildung 8 zu sehen. Man beginnt dann mit dem Loch mit der Nummer eins H_1 mit der Verbinung mit P. Dazu bestimmt man alle Punkte v_i von P, welche links von u_{left} liegen und sortiert diese nach ihrem Abstand zu u_{left}. Beginnen mit dem am wenigsten entfernten Punkt, v_1, wird dann überprüft, ob $\{v_i u_{left}\}$ eine Brücke ist. Die Sortierung der Punkte v_i soll dafür sorgen, dass möglichst wenig solcher Paare überprpft werden müssen. Führt man diesen Algorithmus für alle i Löcher durch, so erhält man im schlimmsten Fall eine Laufzeit von $O(i \cdot n^2)$.

Ein noch effizienterer Ansatz, welcher auch bei der Umsetzung dieser Arbeit berücksichtigt wird, wurde von David Eberly in seiner Veröffentlichung *Triangula-*

tion by Ear Clipping beschrieben.[27] Zum Bestimmen der zwei Punkte, welche das Loch H und das äußere Polygon P mit einer Brücke verbinden sollen nutzt er eine Herangehensweise, welche mit einer Halbgerade als Sichtlinie arbeitet. Zunächst bestimmt der Algoritmus von Eberly den Eckpunkt u des Loches H mit der größten x-Koordinate. Dieser soll Ausgangspunkt für die Sichtlinie sein. Diese Verläuft entlang positiver X-Richtung und kann somait als $r_u = u + t \cdot (1, 0)$ beschrieben werden. Der sich ergebende Schnittpunkt von r_u mit den Kanten des Polygons P soll mit i bezeichnet werden.

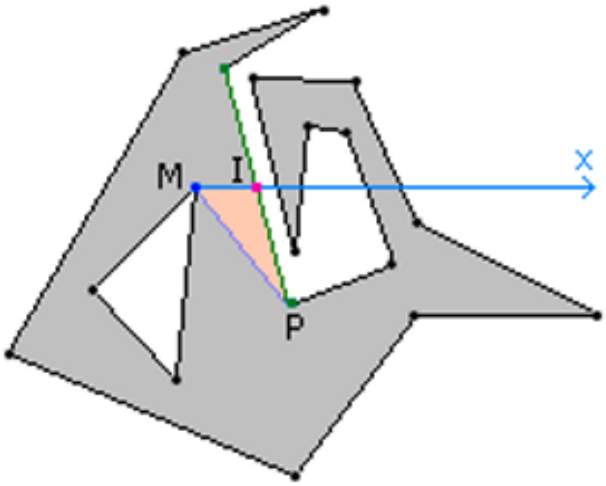


Abbildung 9: Nähester Sichtbarer Punkt w zu Punkt u des Lochs. [27]

Für diesen Punkt gibt es nun zwei Möglichkeiten. Entweder i ist Eckpunkt von P oder i liegt auf einer Kante von P. Im ersten Fall muss i für u sichtbar sein und somit wird die Kante zwischen diesen beiden Punkten die Brücke zwischen H und P. Auch diese muss doppelt vorhanden sein, wie bereits zuvor beschrieben. Der zweite Fall, bei dem i auf einer Kante $v_i v_{i+1}$ liegt bedarf näherer Betrachtung. Hierfür wird ebenfalls wieder der eine der beiden Punkte bestimmt, welcher die größere x-Koordinate besitzt. Es kann sein, dass der so ausgewählte Punkt w nicht für u sichtbar ist. Das ist dann der Fall wenn sich im Dreieck $\triangle uwi$ konkave Eckpunkte von P befinden. Die Kanten, zu denen diese Eckpunkte gehören, blockieren dann die Sicht auf w. Daher muss überprüft werden, ob keine solchen PUnkte im inneren des besagten Dreiecks liegen. Ist dem so, dann ist w der gesuchte Punkt für die Brücke zu u. Sollten sich konkave Punkte im Inneren von $\triangle uwi$ befinden, dann wird der jenige gesucht, der den Abstand zu u minimiert. Einen solchen Punkt muss es immer geben und dieser ist dann sichtbar für u und eigent sich somit für eine Brücke. In Abbildung 10 kommen die Punkte a, b und c also neue Kandidate in Frage. Für a ist der Abstand zu u minimal. Der eben beschriebene Algorithmus ist im Folgenden noch einmal in Pseudocode dargestellt.

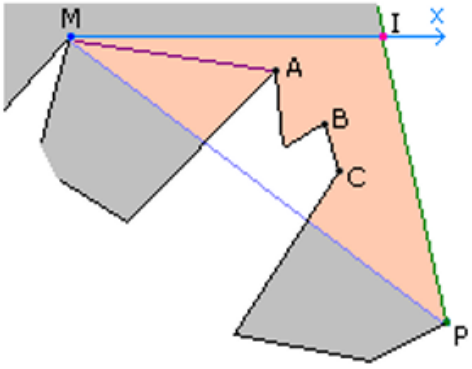


Abbildung 10: Sichtlinie von u nach w ist durch äußeres Polygon blockiert. Konkave Punkte a, b, c sind Kandidaten für die Brückenbildung. Abstand von a nach u ist minimal.[27]

Algorithmus 3: Ermittlung von Brückenknoten mittels Sichtlinie[27]

Eingabe: Ecken $v_i \in V(P)$, Ecken $u_i \in V(H)$
Ausgabe: Kanten $E(P')$
Schritt 1: Wähle u_i mit maximaler x-Koordinate aller $u_i \in V(H)$
Schritt 2: Bestimme ersten Schnittpunkt i von $r_u = u_i + t \cdot (1, 0)$, $(t \in \mathbb{R})$ mit $e \in E(P)$
Schritt 3: Bestimme Eckpunkte von e: $v_{1,e},\ v_{2,e}$
Schritt 4: **if** $i = v_{1,e}$ OR $i = v_{1,e}$
Ausgabe $E(P') := E(P) \cup iu_i \cup E(H) \cup u_i i$
else
Schritt 5: Bestimme Punkt mit maximaler x-Koordinate von $v_{1,e},\ v_{2,e}$ als Punkt w
Schritt 6: Bestimme $\triangle u_i i w$
Schritt 7: **if** Keine Punkte von P im Inneren $I_\triangle u_i i w$ von $\triangle u_i i w$
Ausgabe $E(P') := E(P) \cup wu_i \cup E(H) \cup u_i w$
else
Schritt 8: Bestimme alle konkaven Punkte in $I_\triangle u_i i w$
Schritt 9: Bestimme Punkt $a \in I_\triangle u_i i w \cap E(P)$ mit minimalem Abstand z
Schritt 10: Ausgabe $E(P') := E(P) \cup au_i \cup E(H) \cup u_i a$
endif
endif

Durch das Einfügen von Brückenkanten können zwar das Polygon P und alle Löcher H_i verbunden werden und als ein einziges Polygon P' behandelt werden, nur ergibt sich ein neues Problem. Laut der Definition eines einfachen Polygons (s. Kapitel 2.1.1) ist es nicht erlaubt, dass sich Kanten schneiden oder berühren. Da aber die Brückenkanten doppelt vorhanden sind, also an der exakt

gleichen Position befinden, ist das Polygon P' kein einfaches solches mehr. Man bezeichnet es als **schwach einfach** (weakly simple)[28]. Diese Polygone können per Definition durch eine beliebige minimale Änderung ϵ in einfache Polygone umgewandelt werden. Diese Änderung ϵ kann eine Verschiebung der Eckpunkte der Brücken um eine kleine Konstante sein, sodass sich die doppelten Kanten nicht mehr berühren. Dazu ist es notwendig die ursprünglichen Positionen dieser Punkte zu speichern, damit diese in der Nachbearbeitung wieder an ihre eigentlichen Koordinaten zurück verschoben werden können.

3 Verwandte Arbeiten

3.1 Verbesserungen des Ear-Clipping-Algorithmus

Wenn man einen Algorithmus mathematisch betrachtet, dann ist die Zeit, welcher er bis zur Terminierung benötigt, zumeist der Gegenstand der Betrachtung. Mittels der Komplexitätstheorie lässt sich eine vom Computertyp unabhängige Beschreibung dafür finden. Das Referenzmodell ist dabei zumeist die Turingmaschine (TM).[21] Ziel ist es, dass ein Algorithmus auf einer TM in Polynomialzeit abläuft. Diese Zeit hängt zumeist von der Eingabegröße ab. Für den ECA ist die entscheidende Größe die Anzahl der Ecken n des Polygons P. Betrachtet man den ECA auf einer TM, so hat er eine Komplexität von $O(n^3)$. Zwar ist dieser Term ein Polynom in n, jedoch ist das kein Grund für die Wissenschaft, hier mit der Optimierung aufzuhören. Wie O'Rourke in seiner Arbeit zeigt, kann man den ECA durch kleine Änderungen so modifizieren, dass dieser eine Komplexität von $O(n^2)$ aufweist.[22] Diese Erkenntnis nutzen Mei, Tipper und Xu, um den Algorithmus auf andere Art zu verbessern.[15]

Für einen Algorithmus wie den ECA ist nicht nur seine Laufzeit entscheidend. Während andere Algorithmen beispielsweise Entscheidungsprobleme lösen, bei denen es nur um die Frage nahc der Existenz der Lösung geht, ist bei einer Triangulation bereits bekannt, dass es unterschiedliche Lösungen gibt. Daher ist die Frage nicht, ob eine Lösung existiert, sondern ob eine optimale solche gefunden werden kann. Optimal ist dabei ein relativer Begriff, der stark von den Rahmenbedingungen abhängt. Für den ECA ist der Speicherbedarf ebenfalls entscheidend. Dieser ist bei Mei, Tipper und Xu durch $O(n)$ begenzt. Das Ziel ihrer Arbeit war es, qualitytiv hochwertige Triangulationen für komplexe Polygone zu erzeugen. Der Qualitätsparameter war dabei der kleinste Innenwinkel der erzeugten Dreiecke in der Zerlegung. Genauer ging es darum, die sogenannten Slivers zu vermeiden (s. Kapitel 2.3).

Ihr Ansatz war es, den verbesserten Algorithmus von O'Rourke so zu modifizieren, dass er über die Option des **Edge Swappings** verfügt. Wird ein Ear erkannt, dann wird für jeden seiner Innenwinkel überprüft, ob dieser kleiner ist als ein zuvor festgelegter Grenzwert. Ist das der Fall, dann muss bei diesem Dreieck Edge Swapping durchgeführt werden. Dafür werden der größte Innenwinkel des Dreiecks und die, ihm gegenüberliedende, längste Seite bestimmt. Daraufhin wird überprüft, ob es ein Nachbardreieck gibt, welches sich mit dem Ursprünglichen eben diese längste Seite teilt. Gibt es einen solchen Nachbarn, dann wird in dem von den beiden Dreiecken gebildeten Viereck eine Diagonale zwischen den beiden Ecken gezogen, welche jeweils der zuvor bestimmten längsten Seite gegenüber lagen. Auf diese Art entstehen wieder zwei Dreiecke, von denen nun die Innenwinkel auf ihre Größe überprüft werden. Ist jeweil der kleinste Winkel größer als der

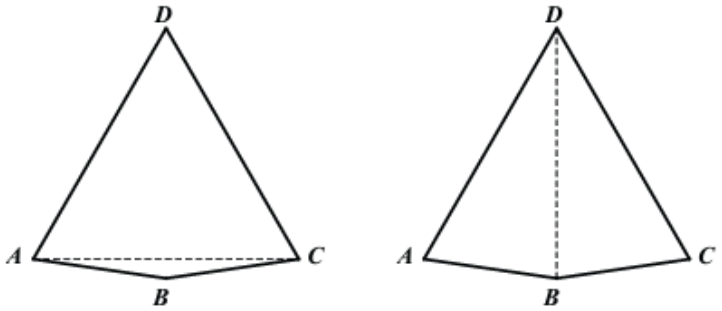


Abbildung 11: Edge Swapping [15]

Grenzwert, dann ist die Qualität der Dreiecke nun besser als die des ursprünglich Ausgewählten. Wenn dem nicht so ist, bleibt alles unverändert und das Ear wird wie es war gewählt und abgeschnitten.

Auf diese Art kann die Qualität der Dreiecke in der Triangulation stark erhöht werden. Sie zeigen an Beispielen, dass sich die Innenwinkelgröße durchschnittlich verdoppelt. Teilweise können Dreiecke mit minimalem Innenwinkel von $< 15°$ ganz eliminiert werden. Das hängt jedoch vom eingegebenen Poylgon ab und hat keine Allgemeingültigkeit.

Eine weitere wichtige Verbesserung für den ECA stellt die Verwendung einer Gitterdatenstruktur, dem sogenannten **Grid** dar. Sie wurde beispielsweise bei der Implementierung von FIST verwendet. Dieses Framework ist ein in C++ verfasster Code für Polygontriangulation basierend auf dem ECA.[7] FIST wurde von M. Held entwickelt und beinhaltet verschiedenste Techniken, um Triangulationen zu verbessern.[24] Eine solche Technik sind die angesprochenen Grids. Eine der möglichen Überprüfungsmöglichkeiten ob ein Dreieck in einem Polygon ein Ear ist oder nicht ist die folgende:

Satz 5 (Ear-Bedingung:)[24]
Dreiaufeinanderfolgende Punkte $v_{i-1},\ v_i,\ v_{i+1} \in V(P)$ bilden ein Ear von P genau dann, wenn
(1) v_i ist konvex,
(2) die abgeschlossene Hülle des Dreiecks $\triangle v_{i-1}v_iv_{i+1}$ enthält keine konvexen Punkte aus $V(P)$ (außer v_{i-1}, v_{i+1})

Benutzt man diese Definition, muss man also für jedes mögliche Dreieck in P überprüfen ob ein oder mehrere konkave Punkte in der abgeschlossenen Hülle des Dreiecks liegen. Hier kommt das Grid ins spiel. Es unterteilt das minimal umgebende Rechteck, welches das gesamte Polygon einschließt, in rechteckige Zellen. Für jeden konkaven Punkt von P wird dann die Zelle bestimmt, in welcher er sich befindet und speichern den Punkt in dieser. Da man am besten auf Nummer sicher geht, wird ein Punkt der sehr nah an der Grenze zwischen zwei Zellen

liegt auch in die Nachbarzelle gespeichert. Was die Auflösung des Grids angeht, ist klar, dass man keinen Speicherplatz mit einem sehr hochauflösenden Gitter verschwenden möchte. Nach Held wurde experimentell ein Optimum gefunden mit $w\sqrt{n} \times h\sqrt{n}$ als Anzahl der Zellen, wobei w und h von den Proportionen des umgebenden Rechtecks von P abhängen und n die Anzahl der Ecken von P ist. Zusätzlich wird gefordert, dass $w \cdot h = 1$ gilt.

Für die Überprüfung der Ear-Bedingung wird zunächst das minimal umgebende Rechteck bestimmt, welches das betreffende Dreieck einschließt. Dann werden alle Zellen überprüft, welche dieses Rechteck überdeckt, ob sie einen konkaven Punkt enthalten. Ist das Dreieck ein Ear kann das Grid nach dem Clipping einfach aktualisiert werden, da für jedes Dreieck maximal zwei Punkte ihren Status von konkav zu konvex ändern können. Wird ein Punkt konvex, dann wird er einfach aus dem Grid gelöscht.

Man bezeichnet die Verwendung eines Grids als *spatial hashing Verfahren* oder *geometrical hashing Verfahren*. Allgemein ist spatial hashig die Erweiterung der Idee hinter einer Hash Tabelle auf höhere Dimensionen. Statt einem einfachen Schlüsselwert (Key) werden die Koordinaten des zu speichernden Objekts als Key verwendet.[29] Das oben beschriebene Verfahren von Held wird in der Umsetzung dieser Arbeit verwendet.

3.2 Parallelisierung des Ear-Clipping-Algorithmus

Anstatt den Algorithmus selbst in seiner Laufzeit zu verbessern, ist es ein Gedanke, die Abarbeitung aufzuteilen. Vorallem mit der technischen Entwicklung mehrerer Prozessorkerne in einer Central Processing Unit (CPU) ist verteiltes Rechnen ein gängiges Konzept. Hierzu haben Eder, Held und Palfrader eine Arbeit verfasst, die sich mit der Umsetzung des ECA unter dem Gesichtspunkt der coarse-grain parallelization, zu Deutsch grobkörnigen Parallelisierung, befasst.[7] Dieses Prinzip beschreibt die Aufteilung eines Programms in längere Unteraufgaben. Das ist ein für Multicore Computer sehr geeignetes Konzept. Andere Arbeiten befassten sich auch mit der Umsetzung der Arbeitsteilung, aber dort speziell mit dem Konzept der fine-grain parallelization im Bezug auf die DT. Die fine-grain parallelization, also die feinkörnige Parallelisierung, beschreibt die Aufteilung eines Programms in eine Vielzahl kleinerer Aufgaben. Hier ist beispielsweise M. Goodrich[23] zu nennen.

Eder, Held und Palfrader haben ihre Arbeit auf FIST aufgebaut. Wie bereits in Kapitel 3.1 erwähnt, bastiert FIST auf dem ECA.[7] Sie beschränkten sich dabei mit der Parallelisierung auf den Bereich des Algorithmus, welcher sich mit der Klassifikation und dem Clipping der Ears befasst. Dieser Teil macht etwa 80% des Rechenaufwandes aus. Um eine Aufteilung in k Threads zu erreichen, welche dann auf den k Kernen der CPU abgearbeitet werden sollen, nutzen Sie

drei verschiedene Ansätze und vergleichen diese miteinander und mit der nicht parallel laufenden Form des Algorithmus in FIST.

Ihr erster Ansatz beruht auf dem *divide-and-conquer-Prinzip*. Anstatt das Polygon P allerding durch Diagonalen in etwa gleich große Unterpolygone P_k zu unterteilen, nutzen Sie $k-1$ viele senkrechte Geraden dafür. Dies ist weit weniger aufwendig in der Berechnung, da das Finden von geeigneten Diagonalen relativ rechenintensiv ist. Sie berufen sich dabei auf einen Algorithmus von Sutherland und Hodgman [25]. Bei dieser Form der Unterteilung enstehen sogenannte Steiner Punkte (SP), welche die Schnittpunkte der senkrechten Geraden mit den Strecken der äußeren Begrenzung darstellen. Dafür benötigt man eine Zeit von $O(n)$ pro Gerade l und fügt im schlimmsten Fall $O(n)$ SP ein. Diese werden als neue Eckpunkte in den Unterpolygonen eingefügt und damit vom ECA auch als Eckpunkte der Dreiecke in der Zerlegung benutzt. Das führt dazu, dass Dreiecke in der Gesamtzerlegung von P entstehen, welche unzulässige Eckpunkte besitzen, da diese im Ursprünglichen Polygon nicht existieren. Dafür muss eine Bereinigung der Zerlegung durchgeführt werden, nachdem alle k Threads ihre Triangulation der P_k Unterpolygone geliefert haben. Durch den Schnitt des Poylgons mit einer senkrechten Geraden entstehen zwei SP s_a und s_b. Um diese wieder zu löschen, werden alle Dreiecke, welche zu einem dieser beiden Punkte inzident sind, aus der Triangulation gelöscht. Auf diese Weise erzeugt man ein Loch H in der Zerlegung, welches wieder ein Polygon ist. Diese kann man nun durch erneute Triangulation mit validen Dreiecken füllen.

Einen *partition-and-cut-Ansatz* zu verwenden, war die zweite Variante, um die Triangulation auf die k Threads aufzuteilen. Dabei wird nicht wie bei *divide-and-conquer* das gesamte Polygon P, sondern nur sein begrenzender Streckenzug unterteilt. Hierfür werden **Landmarks**, also Wegpunkte, eingeführt. Dies geschieht anhand der Indizierung der n Ecken. Die Eckpunkte mit den Indizes $\left\{0, \frac{n}{k}, \frac{2n}{k}, \ldots, \frac{(k-1)n}{k}\right\}$ werden die Landmarks. Der Streckenzug zwischen je zwei dieser Markierungen wird jeweils einem Thread zugewiesen. Die Wegpunkte gehören dabei jeweils zu zwei benachtbarten Teilstreckenzügen gleichzeitig. Jeder Thread durchläuft dann eine Klassifikations- und eine Clippingphase, bei denen darauf geachtet wird, dass die Landmarks nicht gelöscht werden. Ist das geschehen und alle Threads beendet, dann bleibt ein Teil des Polygons noch unbearbeitet. Dieser Teil wird bei Eder, Held und Palfrader nicht nocheinaml in Abschnitte für verschiedene Threads unterteilt sondern wird dann vom sequentiellen ECA in FIST bearbeitet. Zwischen den Threads wird keine Synchronisation benötigt, da sowohl Klassifikation als auch Clipping völlig undabhängig von anderen Threads ablaufen und nur in ihrem jeweiligen Abschnitt Dreiecke erzeugt werden. Dabei sei angemerkt, dass das Überprpüfen der Ear-Eigenschaft nur Lesezugriff auf die

Globale Liste aller Eckpunkte des Polygons benötigt. Der Vorgang der Aufteilung in Threads und deren bearbeitung ist in der nachstehenden Abbildung zu sehen.

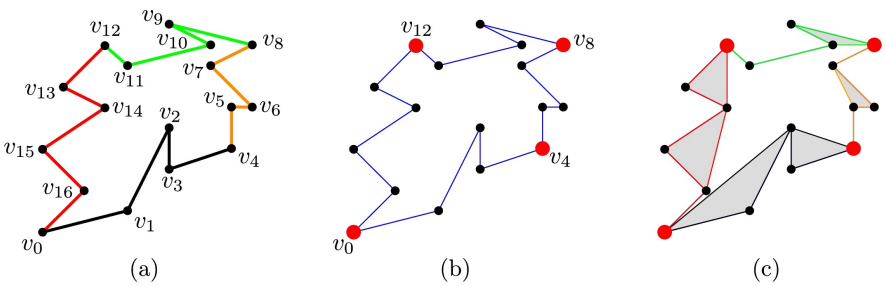


Abbildung 12: (a) Einfaches Polygon P unterteilt in vier Streckenzüge (b) Landmarks vervorgehoben (c) Triangulierung durch Threads

Der dritte Ansatz, betreffend dem ECA in FIST ist der sogenannte *mark-and-cut-Ansatz*, welcher Ähnlichkeiten zum vorher erwähnten *partition-and-cut-Ansatz* aufweist. Auch in diesem Fall werden einige Eckpunkte von P als Markierungen genutzt. In der Marierungsphase, durchläuft ein Thread den Streckenzug von P und speichert jeden zweiten konvexen Eckpunkt in einer Liste. Hat dieser Thread die Hälfte aller Punkte überprüft, werden die Cut-Threads gestartet, welche nur die Cutting-Phase durchlaufen. Bildet ein Punkt in der Liste mit seiner gegenüberliegenden Seite ein Dreiecke, dann wird dieses sofort als valide gespeichert und abgeschnitten. Jeder dieser Punkte darf nur einmal bearbeitet werden, damit es nicht zu Asynchronität und Redundanz kommt. Wenn die Cut-Threads alle ihre Arbeit getan haben, werden sie neu gestartet und bearbeiten dann alle Punkte, die Seit ihrem letzten Start zur Liste hinzugefügt worden sind. Während dessen durchläuft der Mark-Thread das Polygon erneut und fürgt neue konvexe Punkte zu Liste hinzu und so weiter, bis nurnoch weniger Dreiecke erkannt werden, als vorher mit einem Grenzwert festgelegt. Dieser lag bei Eder, Held und Palfrader bei 20 Dreiecken. Der Rest von P, welcher noch nicht bearbeitet wurde, wird dann wie im *partition-and-cut-Ansatz* von einem sequentiellen Aufruf von FIST bearbeitet.

Der finale Vergleich aller drei Ansätze hinsichtlich ihrer Qualität zeigt, dass sie

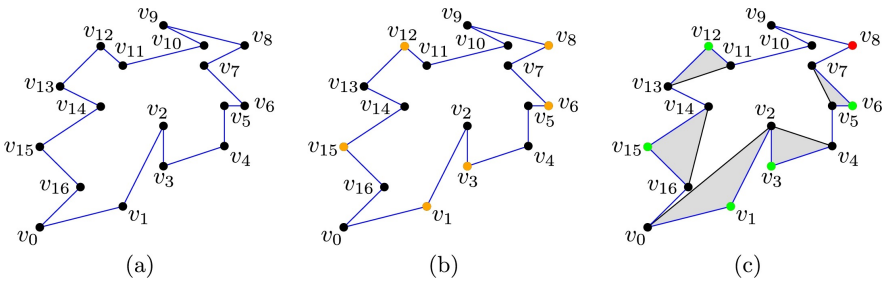


Abbildung 13: (a) Einfaches Polygon P (b) Erste Markierungsphase, ausgewählte Ecken in orange (c) Erste Cutting-Phase

in etwa gleich gut sind. Als Vergleich wurde von ihnen noch eine Variante der DT, die sogenannte Constrained Delaunay Triangulation (CDT) durchgeführt, auf die an dieser Stelle allerdings nicht genauer eingegangen werden soll. Die folgende Tabelle zeigt das Ergebnis.

	CDT	FIST's top	D&C	P&C	M&C
(1) Durchschn. Abw. 60°	30.79°	31.53°	35.29°	34.97°	38.38°
(2) Durchschn. min. Winkel	24.60°	23.40°	20.07°	21.32°	21.07°

Tabelle 1: Vergleich der FIST Trianulation inklusive CDT und FIST's top Heuristik. Aufgeführt sind folgende Parallele Versionen von FIST: Divide-and-conquer D&C, Partition-and-cut P&C und Mark-and-Cut M&C. (1) Durchschnittliche Abweichung aller Innenwinkel von 60° über alle Triangulationen (je kleiner, desdo besser) (2) Durchschnittliche Größe des kleinsten Innenwinkels aller Dreiecke über alle Trinagulationen (je größer, desdo besser) [7]

3.3 Alternative Verfahren und Ansätze

3.3.1 Delauny Triangulation

Spricht man von Triangulationen, so stößt man zwangsläufig auf den Begriff der Delauny Triangulation. Sie ist, qualitativ gesehen, die Triangulation mit den ausgewogensten Dreiecken.

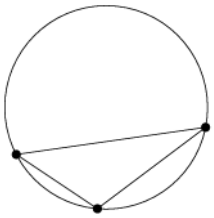


Abbildung 14: Umkreis eines Dreiecks [26]

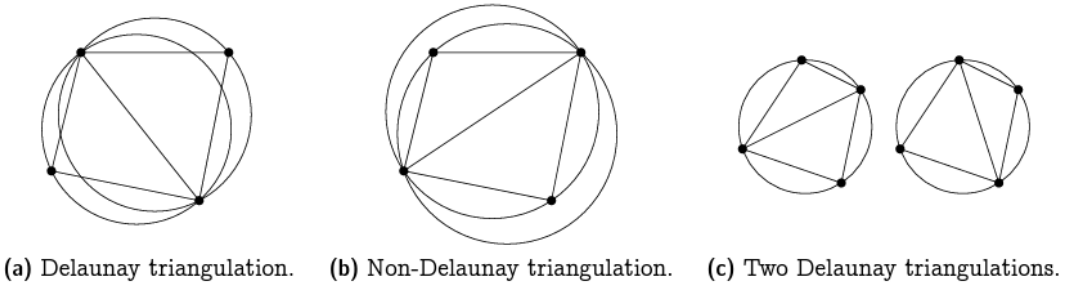
Um jedoch zu verstehen wie die DT gebildet wird, benötigt man den Begriff des *leeren Umkreises*. Sie ist die grundlegende Bedingung für eine DT, da man nur solche Triangulationen, in denen alle Dreiecke eben diese Bedingung des leeren Umkreises erfüllen, eine DT nennt.[26]

Man betrachte eine Menge aus Punkten V in der euklidischen Ebene $\mathbb{R}^2$. Ein Dreieck $\triangle v_1 v_2 v_3$ mit $v_1, v_2, v_3 \in V$ erfüllt die Bedingung des leeren Umkreises, wenn der Kreis, welcher durch die drei Punkte v_1, v_2, v_3 geht, keine anderen Punkte $v_i \in V$ beinhaltet.

Kann man also eine Menge aus Dreiecken T finden, sodass jeder Punkt der Menge V teil mindestens eines Dreiecks ist und jedes Dreieck wenigstens eine Seite mit einem anderen Dreieck teilt, dann ist die T eine DT von V, wenn jedes Dreieck in T einen leeren Umkreis besitzt.

Dass diese Triangulation nicht eindeutig ist, kann mann an einem einfachen Beispiel zeigen. Hat man vier Punkte im $\mathbb{R}^2$, welche alle zueinander konvex sind, dann ist es möglich, dass alle diese Punkte auf dem selben Umkreis liegen. In einem solchen Fall sind alle möglichen Kombinationen aus sich nicht gegenseitig schneidenden Dreiecken als DT zulässig. Jedoch gibt es auch den Fall, dass nur je drei Punkte auf einem Kreis liegen. Dafür gibt es immer zwei Möglichkeiten, von

denen meist nur eine eine DT ist. In Abbildung 15 ist sind in den Fällen (a) und (b) diese Möglichkeiten für Umkreise zu sehen, welche einmal eine DT erzeugen und einmal nicht. In Fall (c) ist der zuvor beschriebene Fall aufgeführt, bei dem die vier Punkte alle auf dem selben Kreis liegen.



(a) Delaunay triangulation. **(b)** Non-Delaunay triangulation. **(c)** Two Delaunay triangulations.

Abbildung 15: Delauny Triangulation für vier konvexe Punkte. (a) Delauny Triangulation (b) keine Delauny Triangulation (c) mehrere gleichwertige Triangulationen [26]

Eine sehr vorteilhafte Eigenschaft der DT, welche diese für praktische Anwendungen sehr interessant macht, ist, dass sie den minimalen Innenwinkel jedes Dreiecks der Triangulation maximiert. Das bedeutet, dass die DT die qualitativ hochwertigste aller möglichen Triangulationen einer Menge von Punkten ist. Aber auch so ist es nicht möglich, dass jede DT die Bedingung erfüllt, keine Slivers zu enthalten. Es bedeutet lediglich, dass, wenn ein Sliver teil einer DT ist, dann wäre auch in jeder anderen Triangulation mindestens ein Sliver enthalten. Das ist immerhin in sofern eine gute Eigenschaft, da eine DT einer Punktmenge immer die minimale Anzahl an Slivers enthält. Sie dient damit als optimaler Vergleich für beispielsweise den ECA, da man damit die Abweichung der Triangulation des ECA zu der durch die DT erzeugten bestimmen kann.

Möchte man, wie in dieser Arbeit, ein Polygon in Dreiecke zerlegen, so stößt man auf ein Problem. Ein Polygon in zwar eine Menge aus Punkten, jedoch sind diese bereits durch einen Streckenzug fest miteinander verbunden. Man kann in dem meisten Fällen also keine echte DT erzeugen, da die Dreiecke nicht frei wählbar sind. Man spricht dann von der sogenannten CDT. Diese versucht ebenfalls die Bedingung des leeren Umkreises zu erfüllen, allerdings mit einer Einschränkung, welche sie für vorgegebene Polygone umsetzbar macht. Der Umkreis eines Dreiecks $\triangle$ darf andere Punkte des Polygons enthalten, wenn diese von innerhalb von $\triangle$ nicht sichtbar sind. Ein Punkt p heißt **sichtbar**, wenn eine Punkt q im

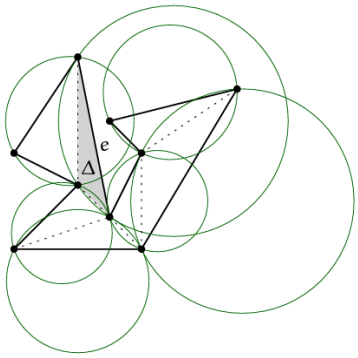


Abbildung 16: Constrained Delauny Triangulation eines Polygons[26]

Inneren des Dreiecks $\triangle$ existiert, so dass die Strecke $\overline{pq}$ keine andere Strecke $e \in E$ schneidet. Anderenfalls blockiert e sozusagen die Sicht auf den Punkt p.

3.3.2 Algorithmus basierend auf Sichtbarkeit

Wie schon bei der DT ist hier der Begriff der Sichtbarkeit von Eckpunkten entscheidend. Anders als zuvor werden hier jedoch keine Dreiecke ermittelt, welche die Bedingung des leeren Umkreises erfüllen. In diesem Algorithmus, beschrieben von Ran Liu, wird das Polygon P in zwei Unterpolygone P_1 und P_2 unterteilt, welche dann solange rekursiv weiter unterteilt werden, bis die entstandenen Unterpolygone P_i Dreiecke sind.[9] Hierfür muss in einem ersten Schritt die Sichtbarkeit jedes Punkte gegenüber einem Referenzpunkt v_i überprüft werden. Dazu betrachtet man zunächst die Nachbarn v_{i-1} und $v_i + 1$ von v_i. Diese begrenzen das sogenannte **Sichtfeld** von v_i, welches mit α bezeichnet wird und dem Winkel in v_i entspricht. Für spätere Betrachtungen zählen v_{i-1} und v_{i+1} als nicht sichtbar im BEzug auf v_i. Entlang der Kanten von P wird nun ausgehend von v_{i+1} entgegen dem Uhrzeigersinn überprüft, ob ein Punkt v_j im Sichtfeld von v_i liegt. Ist das der Fall, dann gilt der Punkt v_j als sichtbar, wenn die Strecke $v_i v_j$ eine Diagonale von P ist. Zusätzlich begrenzt dieser Punkt nun das Sichtfeld und es muss verkleinert werden. Das neue Sichtfeld α berchnet sich also durch $\alpha = v_{i-1}, v_i, v_j$. Der Vorgang wird fortgesetzt, bis alle Punkte überprüft sind. Ist diese Überprüfung für alle v_i von P abgeschlossen, wird die Anzahl der sichtaberen Punkte gegenüber dem jeweiligen Referenzpunkt bestimmt. Diese dient als Vergleichskriterum für die Punkte untereinander. Als Beispiel ist in der nachfolgenden Abbildung einmal die Ermittlung von *alpha* und die Überprüfung mehrerer Punkte bezogen auf den Punkt v_0 dargestellt.

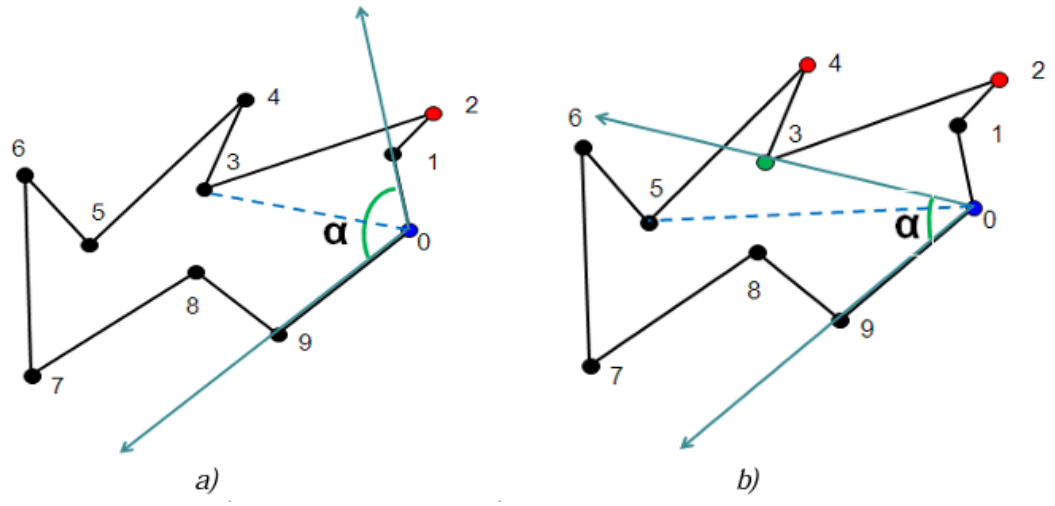


Abbildung 17: (a) v_3 ist sichtbar für v_0 (b) Überprüfung der Sichtbarkeit von v_5 mit neuem Sichtfeld α (nicht sichtbar entspricht rot, sichtbar entspricht grün) [9]

Für die Anzahl sichtbarer Punkte bezogen auf v_i wird die hier Bezeichnung

$s(v_i)$ verwendet. Man kann diesen Vorgang der Sichtbarkeitsanalyse wie folgt in Pseudocode beschreiben.

Algorithmus 4: Sichtbarkeitsanalyse für einen Punkt v_i

Eingabe: Ecken $v_i \in V(P),\ n = |V(P)| - 3,\ s(v_i) = 0$
Ausgabe: Anzahl sichtbarer Punkte $s(v_i)$
Schritt 1: Wähle Referenzpunkt v_i
Schritt 2: $\alpha = \angle v_{i-1} v_i v_{i+1}$
Schritt 3: **while** $n > 0$ **do**
if $\angle v_{i-1} v_i v_j > 0 \wedge \angle v_{i-1} v_i v_j < \alpha$
if $v_i v_j$ Diagonale von P
(a) $s(v_i) = s(v_i) + 1$
(b) $\alpha = \angle v_{i-1} v_i v_j$
(c) $n = n - 1$
Schritt 4: Ausgabe von $s(v_i)$

Ist $s(v_i)$ für jeden Eckpunkt v_i von P ermittelt, wird der Punkt mit der größten solchen Anzhal ausgewählt. Gibt es mehrere solche Punkte, dann kann ein beliebige dieser Punkte gewählt werden. Dieser Punkt wird dann als **First Head** bezeichnet. Als nächstes wird noch der Punkt mit der zweit höchsten Anzahl $s(v_i)$ gewählt. Er wird als **Second Head** bezeichnet. Sollte es hier Uneindeutigkeiten bei der Auswahl geben, dann wird ein Test durchgeführt, um diesen Punkt auszuwählen. Durch die Diagonale zwischen First und Second Head soll das Polygon in zwei Unterpolygone zerlegt werden. Man testet, bei welchem der möglichen Punkte für den Second Head, der Betrag der Differenz zwischen der Anzahl an Strecken in den beiden Unterpolygonen P_1 und P_2 am geringsten ist. Das ist in der nächsten Abbildung zu sehen.

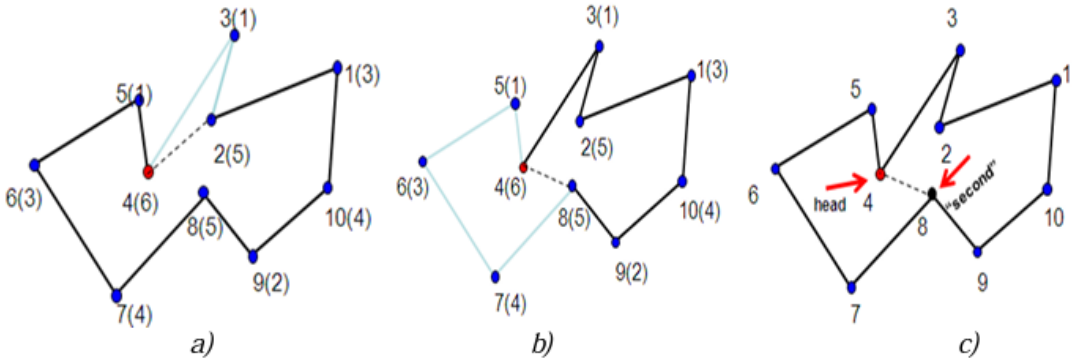


Abbildung 18: (a) und (b) Vergleich der Differenz zwischen der Länge des schwazen und des türkiesen Strecknabschnitts (c) Auswahl des Second Head [9]

Für die Zerlegung werden First und Second Head dann dupliziert, damit beide Polygone vollständig begrenzt sind. In beiden Unterpolygonen muss dann die Sichtbarkeitsanalyse erneut durchgeführt werden. Damit dies ein wenig schneller

geschieht, kann man die sichtbaren Punkte im Bezug auf einen Punkt v_i in einer sogennanten *single circular list* gespeichert werden. In einer solchen Liste zeigt der Pointer des letzten Elements auf das erste Element der Liste. Beim Update der Sichtbarkeiten müssen dann für jeden Punkt nur die Punkte in seiner jeweiligen Liste überprüft werden und jetzt nicht mehr sichtbare Punkte aus der Liste gelöscht werden. Damit wird die Laufzeit der Sichtbarkeitsanalyse von $O(n^2)$ auf $O(n)$ begrenzt. Nichtsdesdotrotz hat der neue Algorithmus von Ran Liu, nach seiner eigenen groben Analyse, eine Komplexität von über $O(n^3)$. Im Vergleich mit dem ECA, welcher eine Komplexität von $O(n^2)$ besitzt, schneidet er damit schlechter ab. In Tabelle 2 und 3 werden zwei Beispiele des Vergleichs zwischen dem Sichtbarkeitsalgorithmus und dem ECA aus der Arbeit von Liu angeführt. Die Algorithmen wurden auf zufällig generierten Polygone unterschiedlicher Knotenanzahl und Form getestet. Dabei waren die getesteten Polygonformen einmal *rund*, das heißt die Eckpunkte konnten nur in einem quadratischen Koordinatenbereich liegen. Der andere Typ war *länglich*, wobei die Punkte eher in ihrer x-Koordinate weiter streuten, nicht so sehr jedoch in der y-Koordinate. Es ist in den Ergebnissen dieser Tests ersichtlich, dass die Qualität der Dreiecke bezogen auf ihre minimalen Innenwinkel bei Liu's Algorithmus besser ist, als die des ECA. In Sachen Laufzeit jedoch schneitet der neue Algorithmus jedoch schlechter ab, wie bereits die Komplexitätsanalyse zeigte.

In der nachfolgenden Abbildung ist zunächst einmal exemplarisch dargestellt, wie eine solche Zerlegung eines einfachen Polygons durchgeführt wird. Danach folgen die angesprochenen Tabellen.

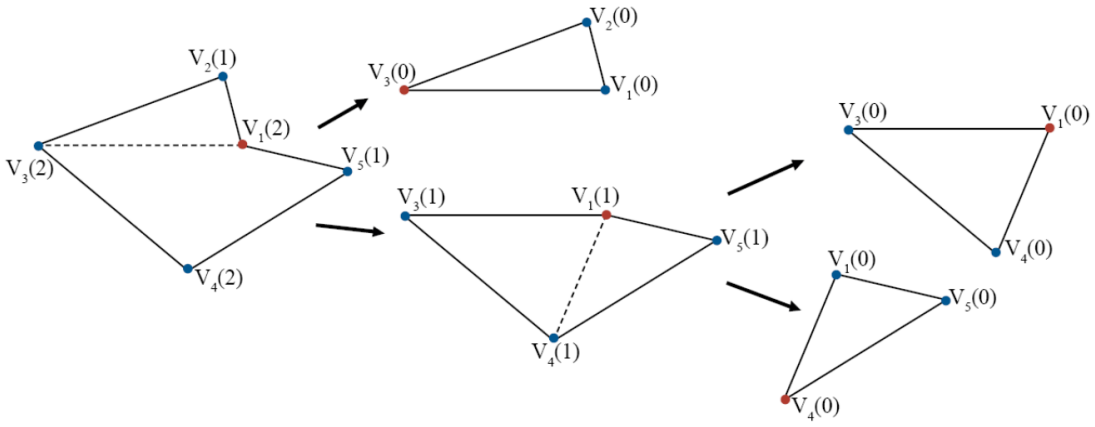


Abbildung 19: Von links nach rechts die Schritte der Unterteilung eines Polygons in Unterpolygone bis zur Triangulierung mittels Sichtbarkeit der Eckpunkte (rot die First Heads)[9]

Eckenanzahl: 30 $v_i = \{(x,y) \mid -50 < x < 50, -50 < y < 50\}$				
Algorithmus	Polygon-fläche	Durchschn. Dreiecksfläche	Standartabw. der Dreiecksfläche	Durchschn. min. Winkel
Ear-Clipping	4128,5px	147,446px	163,63px	10,29°
Sichtbarkeit	4128,5px	147,446px	148,91px	15,04°

Tabelle 2: Vergleich des ECA und des Sichtbarkeitsalgorithmus bei einem rundem Polygon mit 30 Ecken anhand der Standartabweichung der Dreiecksfläche in Pixeln (je kleiner desdo besser) und der durchschnittlichen minimalen Innenwinkelgröße in Grad (je größer desdo besser) [9]

Eckenanzahl: 30 $v_i = \{(x,y) \mid -100 < x < 100, -30 < y < 30\}$				
Algorithmus	Polygon-fläche	Durchschn. Dreiecksfläche	Standartabw. der Dreiecksfläche	Durchschn. min. Winkel
Ear-Clipping	5066px	180,93px	185,50px	9,23°
Sichtbarkeit	5066px	180,93px	146,03px	16,45°

Tabelle 3: Vergleich des ECA und des Sichtbarkeitsalgorithmus bei einem länglichen Polygon mit 30 Ecken anhand der Standartabweichung der Dreiecksfläche in Pixeln (je kleiner desdo besser) und der durchschnittlichen minimalen Innenwinkelgröße in Grad (je größer desdo besser) [9]

4 Praktische Implementierung

4.1 Programmiersprache und Bibliotheken

4.1.1 Rust

Wie bereits im Abschnitt über verwandte Arbeiten angesprochen, sind wichtige Punkte bei der Implementierung von Algorithmen die Geschwindigkeit und die Speichernutzung. Eine Programmiersprache, welche darauf ausgelegt ist in diesen Bereichen besonders effizient zu sein, ist Rust. [30] Entworfen von Graydon Hoare, welcher seine Neuentwicklung im Juli 2010 das erste Mal vorstellte, ist Rust eine sehr vielseitige Sprache. Mit dem Grundsatz einer open-source Multiparadigmen-Systemprogrammiersprache ist sie nicht nur für die breite Masse der Programmierer zugänglich, sondern auch speziell für die Umsetzung von hardwarenaher Programmierung geeignet. Die Sprache setzt viele verschiedene Paradigmen der praktischen Programmierung um. So unterstützt Rust sowohl funktionale als auch objektorientierte, sowie nebenläufige Programmierung. Auch ein hoher Abstraktionsgrad ist möglich. Vor allem aber wurde beim Entwurf der Sprache darauf geachtet, dass die Kosten der Abstraktion zu Laufzeit so gering wie möglich sind. Man spricht von *zero-cost-abstractions*, welche beispielsweise auch bei der weitverbreiteten Programmiersprache *C++* umgesetzt wurden. [31] Ein weiterer Vorteil von Rust ist, das die Sprache für *cross-platform* Benutzung geeignet ist. Das bedeutet sie ist Betriebssystem unabhängig. Das macht Rust zu einer sehr einfach zugänglichen Sprache, da es nicht notwendig ist, ein UNIX-Basiertes- oder ein Windows-Betriebssystem zu besitzen oder gegebenen Falls ein Subsystem oder eine Virtual Machine zu nutzen. All das würde zusätzlichen Aufwand bedeuten. Trotz allem sind natürlich, wie in keiner Programmiersprache, alle gewünschten Funktionen bereits implementiert. Bei Rust stellt das, durch den open-source Charakter, jedoch kein Problem dar. Die Community kann unter Nutzung des originalen Funktionsumfangs, der Standardbibliotheken, weitere neue Bibliotheken erstellen. So ist beispielsweise die Bibliothek *Iced* entstanden, welche für den Entwurf von GUIs gedacht ist. Im nächsten Kapitel ist darüber mehr zu lesen.

4.1.2 Iced

Wie bereits angedeutet ist Iced eine Bibliothek für Rust, welche sich auf die Umsetzung von grafischen Nutzerinterfaces spezialisiert ist. Der spanische Programmierer Héctor Ramón ließ sich bei Iced von der Sprache Elm inspirieren. Es ist zu erwähnen, dass sich Iced zum Zeitpunkt, da diese Arbeit verfasst wird, mit der Version 0.4 noch im experimentellen Status befindet. Dennoch sind bereits die verschiedensten Funktionen nebst anschaulichen Beispielen für die Implemen-

tierung umgesetzt. Auch sind zwei verschiedene Renderer, also Software zum Darstellen von Grafiken auf dem Computerbildschirm, vorhanden. Namentlich *iced-wgpu* und *iced-glow*, unterstützt der erste der beiden die Verwendung von Vulkan [33], Metal [34] und DirectX 12 [37] und der zweite die Verwendung von OpenGL 2.1+ [35] und OpenGL ES 2.0+ [36]. [32]

4.2 Entwurf des grafischen Nutzerinterfaces

Bevor man an die praktische Umsetzung eines GUI gehen kann, benötigt man einen einen Entwurf. Eine Möglichkeit, um bereits in der Entwurfsphase auf Aspekte der Nutzerfreundlichkeit eingehen zu können, stellen Papierentwürfe dar. Dabei wird das GUI, statt aus digitalen Fenstern, aus einzelnen Blättern Papier gezeichnet und die Funktionen so simuliert. Dabei fallen negative Punkte wie zu tiefe oder breite Menüs auf und können direkt behoben werden, bevor sie schon im Code umgesetzt sind. Das vermeidet aufwendiges Umstrukturieren von hunderten Zeilen Code.

Bei dieser Arbeit gab es zwei konkurrierende Entwurfsideen, welche hier aus Gründen der Anschaulichkeit und Qualität nicht auf Papier dargestellt werden. Statt dessen werden digitale Skizzen verwendet, bei denen eventuelles Auffalten des Papiers durch Pfeile und Beschriftung gekennzeichnet werden. Die Ideen entscheiden sich vor allem durch das erste Menü, welches einmal die Optionen für die Triangulation inkludiert - im Folgenden als *Menü Entwurf - Optionen inkludiert* und einmal ein Entwurf, bei dem die Optionen auf einem extra Menü nach Eingabe des Polygons aufgeführt werden. Dieser zweite Entwurf soll als *Menü Entwurf - Optionen nachfolgend* bezeichnet werden. Klar war jedoch von Beginn an, wie die Algorithmusiteration aussehen soll. Sie wird hier nicht einzeln aufgeführt, ist aber im Abschnitt Finalentwurf zusehen. Zuletzt soll das Resultat des Algorithmus gezeigt werden. Auch hierfür gab es zwei unterschiedliche Konzepte. Einmal sollten zwei Anzeigen umgesetzt werden, zum einen das Ergebnis des gewählten Algorithmus und zum anderen das Ergebnis aus einer DT, welche als optimales Ergebnis gilt und einen guten Qualitätsvergleich ermöglicht. Dieser Entwurf wird als *Resultat Entwurf - Vergleichsfenster* bezeichnet. Der andere Entwurf wird aufgrund der hier nur textlich dargestellten Metadaten als *Resultat Entwurf - Metadaten* bezeichnet.

4.2.1 Menü Entwurf - Optionen inkludiert

4.2.2 Menü Entwurf - Optionen nachfolgend

4.2.3 Resultat Entwurf - Vergleichsfenster

4.2.4 Resultat Entwurf - Metadaten

4.2.5 Finalentwurf

4.3 GUI und Pages

Wie bereits in Kapitel 4.2.5 beschrieben, besteht das GUI aus drei Teilen. Diese werden durch einen Aufzählungstypen `enum Page` dargestellt. Dieser besitzt drei Ausprägungen, wie im folgenden zu sehen, welche zusätzlich eigene Felder für die weitere Implementierung der Funktionalitäten besitzen. Dieses Struct wird in der `main.rs` deklariert, sowie auch die anderen in diesem Abschnitt besprochenen Komponenten des Codes.

```
enum Page {
    Menu {
        tools: Tools,
        progset: ProgramSettings,
        draw_panel: DrawPanel,
        confirm_button: button::State,
        undo_buffer: Vec<PageMessage>,
        action_buffer: Vec<PageMessage>,
        undo_performed: bool,
        dark_mode: bool,
    },
    Iteration {
        preview_panel: PreviewPanel,
        prevoius_button: button::State,
        next_button: button::State,
        end_button: button::State,
        dark_mode: bool,
        current_step: usize,

    },
    Result {
        result_panel: ResultPanel,
        repeat_button: button::State,
        exit_button: button::State,
        dark_mode: bool,
    },
}
```

Auf jede dieser Ausprägungen wird in den folgenden Kapiteln gesondert eingegangen. Als übergeordneten Datentyp existiert das `struct Pages`. Es besitzt zwei Felder. Zum einen einen Vektor aus allen Seiten, welche Ausprägungen des

`enum Page` sind. Zusätzlich gibt es noch den Zähler `current_page: usize` als Index für den eben beschriebenen Vektor. Er wird später verwendet, um von einer Seite des Programms zur nächsten zu wechseln.

```
struct Pages {
    pages: Vec<Page>,
    current_page: usize,
}
```

Das `struct Pages` erhält eine Implementierung mit verschiedenen Funktionen mit dem nachfolgenden Aufruf. Die Funktionen sollen an dieser Stelle in vollständigem Umfang aufgeführt werden. Eine namentliche Erwähnung soll hier genügen. Nur auf die `fn new()` soll einmal genauer eingegangen werden, damit sie exemplarisch für alle weiteren ähnlichen Funktionen erklärt wird. Wichtig für die Funktionalität des Programms ist vor allem, dass `current_page` mit `0` initialisiert wird. Das ist gleichbedeutend mit dem Setzen der Startseite auf `Page::Menu`.

```
impl Pages {
    fn new() -> Pages { ... }
    fn update(&mut self, msg: PageMessage, ) { ... }
    fn view(&mut self) -> Element<PageMessage> { ... }
    fn advance(&mut self) { ... }
    fn return_to_menu(&mut self) { ... }
    fn can_continue(&self) -> bool { ... }
    fn title(&self) -> &str { ... }
}
```

Mittels der `fn new()` wird, bei einer Initialisierung einer Variable vom Typ `Pages`, die Standardbelegung aller Felder dieses structs festgelegt. Dies geschieht beispielsweise beim Start des Programms. Man bezeichnet diese Funktion auch als Konstruktor. Die Datentypen, welche hier zugewiesen werden sowie andere Standardkonstruktoren, werden an anderer Stelle noch Erwähnung finden, wenn sie eine zentralere Rolle spielen.

Noch ist das Programm allerdings nicht lauffähig. Dazu fehlen noch drei wichtige Bestandteile. Als erstes das `struct Gui`. Dieses umfasst zwei Felde, einmal `pages: Pages` und `dark_mod: bool`. Des weiteren wird für dieses Struct eine Implementierung einer `iced::Sandbox` vorgenommen. Die `Sandbox` ist ein Applikationstyp aus der Iced-Bibliothek, welcher ein Programm mit GUI erzeugt, jedoch keine asynchronen Aktionen unterstützt. Hierfür müsste man eine `iced ::Application` nutzen, was für einfache Anwendungen, wie diese Arbeit, nicht notwendig ist. Eine solche Implementierung sieht wie folgt aus:

```
impl Sandbox for Gui {
    type Message = Message;

    fn new() -> Gui { ... }
    fn title(&self) -> String { ... }
```

```
        fn update(&mut self, event: Message) { ... }
        fn view(&mut self) -> Element<Message> { ... }
    }
```

Zuerst wird eine Typendefinition für `Message` durchgeführt. Diese ist wichtig für zwei der vier obligatorischen Funktionen einer `Sandbox`, denn mittels Nachrichten werden alle Rückmeldungen zu Nutzerinteraktionen abgebildet. Die `fn new()` ist wieder, wie bereits beschrieben ein Standardkonstruktor.

Die `fn title(&self)` setzt den Text, welcher oben über einer Seite des Programms im Header angezeigt wird. Dieser wird je nach dem, welche Seite aktiv ist, festgelegt. Dafür findet sich in der Implementierung `impl Page` eine Funktion mit gleichem Namen. Diese ist nachfolgend abgebildet.

```

    impl<'a> Page {

        fn title(&self) -> &str {
            match self {
                Page::Menu { .. } =>
                    "Triangulation for Polygons - Menu",
                Page::Iteration { .. } =>
                    "Triangulation for Polygons
                        - Algorithm Iteration",
                Page::Result { .. } =>
                    "Triangulation for Polygons - Result",
            }
        }
    }
```

Die `fn update(&mut self, event: Message)` hat die Aufgabe, die Übergänge zwischen den einzelnen Seiten zu bewerkstelligen und die jeweilige Update-Funktion der einzelnen Pages aufzurufen. Für die Übergänge werden bestimmte Nachrichten abgefangen, welche von Buttons auf den jeweiligen Seiten generiert werden. Dazu aber in späteren Kapiteln mehr.

Zu guter Letzt gibt es noch die `fn view(&mut self)`. Diese ist für alle visuellen Aspekte des Programms zuständig. Hier wird das übergeordnete Layout der Pages festgelegt und die zur aktiven Page gehörende View-Funktion aufgerufen. Diese legt dann das spezielle Layout jeder Seite fest.

Die Nachrichten, welche während der Laufzeit des Programms von den einzelnen Interaktionselementen generiert und ausgegeben werden, müssen behandelt werden, um alle Funktionalitäten auch bei Aufruf auszuführen. Dafür hat der der Struct `Page` ebenfalls eine Funktion `fn update()`. Diese besteht aus einem großen `match`-Statement, welches für jeden auftretenden Nachrichtentypen vom Typ `PageMessage` die gewünschten Aktionen durchführt. Dieses wird in späteren Kapiteln in Teilen betrachtet werden. Hier seinen einmal alle Nachrichtentypen angeführt. Diese sind in der Datei `message.rs` definiert.

```
pub enum PageMessage {
    //Messegaes needed for inteactions on the menu page

    //Options
    AlgorithmSelected(Algorithm),
    HeuristicSelected(Heuristic),
    EdgeSwappingToggled(bool),
    StepTrigToggled(bool),
    DarkModeToggled(bool),

    //Drawing Tools
    DrawPressed,
    DrawHolePressed,
    UndoPressed,
    RedoPressed,
    ClearPressed, //Opens Popup
    PopUpClosed, //Closes PopUp
    AddPoint(Point),
    ConfirmPressed,
    ClearAll,
    RejectClear,

    //Messages for interactions on the iteration page
    PreviousPressed,
    NextPressed,
    EndPressed,

    //Messages for interactions on the result page
    ExitPressed,
    RepeatPressed,
}
```

Das zweite noch fehlende Element, um das Programm lauffähig zu machen ist die `fn main()`. Diese ist das Kernstück eines jeden Rust-Programms. Sie ist so lange aktiv, bis das Programm beendet wird. Sie hat in diesem Fall die Funktion, die `Sandbox` zu starten und ablaufen zu lassen. Dazu können der `fn Gui::run()`, welche durch die Implementierung der `Sandbox` hinzugefügt wurde, verschiedene Einstellungsmöglichkeiten vom Typ `Settings` übergeben werden. Darunter sind beispielsweise die Höhe und Breite des Programmfensters, ob dieses skalierbar sein soll und anderes. Alles was man nicht per Hand festlegt, wird durch `..Settings ::default()` auf einen Standardwert gesetzt. Das Ganze sieht dann wie folgt aus:

```

pub fn main() -> iced::Result {

    Gui::run(Settings {
        antialiasing: true,
```

```
            window: window::Settings {
                resizable: false,
                position: Position::Centered,
                size: (1280, 720),
                ..window::Settings::default()
            },
            ..Settings::default()
        })
    }
```

Die letzte wichtige Komponente eines Rust-Projekts ist die `Cargo.toml` Datei. Sie beinhaltet alle relevanten Informationen für den Compiler, wie zum Beispiel die Dependencies. Für dieses Projekt sieht es folgendermaßen aus:

```

    [package]
        name = "src"
        version = "0.1.0"
        author = "Christoph Pooch"
        edition = "2021"

    [dependencies]
        iced = {version = "0.4", features = ["canvas"]}
        iced_aw = { version = "0.2", default-features = false,
    features = ["card"]}
        num-traits = "0.2"
    [[bin]]
        name = "src"
        path = "main.rs"
```

Das Programm kann nun mittels des Befehl `cargo run` in der Kommandozeile ausgeführt werden.

4.3.1 Page - Menu

Wie bereits im Kapitel 4.2.5 zusehen war, setzt sich das Menü aus drei großen Bereichen zusammen, der Zeichenfläche, den Zeichenwerkzeugen und den Optionen.

Zeichenfläche

Über die Zeichenfläche, welche der zentralste Bestandteil des Menüs ist, wird das Polygon eingegeben, welches im weiteren Verlauf dann mittels des gewählten Algorithmus zerlegt werden soll. Hierfür besitzt die `Page::Menu` ein Feld `draw_panel: DrawPanel`. Dieser Struct wird in der Datei `draw_panel.rs` definiert und mehrere Felder, wie nachfolgend zu sehen. Davon ist `polygon: DrawState` der Teil, welcher, zusammen mit dem Vektor `vertices: Vec<Point>` als Eingabespeicher, die Zeichenfunktionalität umsetzt. Man kann sich den `DrawState` als Zustandsautomaten vorstellen, welcher drei Zustände vom Typ `pending: Option`

<Pending> (None, WaitNxtInput, ClipToStartVertex) und einen Cache für die Zeichenoperationen des Renderers besitzt.

```
pub struct DrawPanel {
    pub polygon: DrawState,
    pub vertices: Vec<Point>,
    pub panel_width: u16,
    pub panel_height: u16,
    pub closed: bool,
    pub ignore_input: bool,}
```

Um nun einen Mausklick über dem Zeichenfenster abzufangen, wird vor jedem Zeichenvorgang eine Überprüfung durchgeführt, ob der Input gesperrt wurde. Das geschieht, wenn das Zeichenwerkzeug *Draw* nicht aktiviert ist, damit keine ungewollten Eingaben entstehen. Wenn das nicht der Fall ist, also die Variable `ignore_input: bool` auf `false` gesetzt ist, dann wird eine zweite Überprüfung durchgeführt. Hierbei wird abgefragt, ob sich der Mauszeiger in den Grenzen des Zeichenfeldes befindet und eine relative Position bezogen auf das Zeichenfeldkoordinatensystem ausgegeben.

```
let cursor_position =
    if self.ignore_input {
        return (event::Status::Ignored, None);
    }
    else if let Some(position) = cursor.position_in(&bounds){
        position
    } else {
        return (event::Status::Ignored, None);
    };
```

Diese Information wird nun an den Zustandsautomat, der mittels einem `match` -Statement umgesetzt wurde, übergeben. Er überprüft ob ein Mausklick stattgefunden hat und führt dann einen Zustandsübergang aus, wenn das geschehen ist. Dabei wird bei Programmstart im Zustand `None` begonnen. Nach der ersten Eingabe bleibt der Automat solange im Zustand `WaitNxtInput` und gibt einen Nachricht vom Typ `PageMessage::AddPoint(Point)` aus, bis sich der Mauszeiger in einem kleinen Bereich um den zuerst Eingegebenen Eckpunkt des Polygons befindet. In diesem Fall geht er in den Zustand `ClipToStartVertex` über. Hier wird kein Punkt hinzugefügt, sondern das Polygon mit einer letzten Kante geschlossen. Dazu wird die Variable `closed: bool` des Zeichenfeldes auf `true` gesetzt. Die angesprochene Nachricht wird, wie alle Nachrichten vom Typ `PageMessage` in der Funktion `fn update()` des Structs `Page` behandelt. Dies sieht folgendermaßen aus:

```
fn update(&mut self, msg: PageMessage) {
    match msg {
        PageMessage::AddPoint(vertex) => {
            if let Page::Menu { draw_panel,
```

```
                    tools, action_buffer, undo_performed,
                    undo_buffer,.. } = self {

                    tools.clear_active = true;
                    tools.undo_active = true;

                    if *undo_performed {
                        undo_buffer.clear();
                        tools.redo_active = false;
                    }

                    Page::push_vertex_to_buffer(vertex,
                        &mut draw_panel.vertices);

                    draw_panel.polygon.request_redraw();

                    action_buffer.push(
                        PageMessage::AddPoint(vertex));
                }}
        ...
        }}
```

Es werden zunächst einmal zwei Buttons der nachfolgend besprochenen Zeichenwerkzeuge aktiviert - der *Clear*- und der *Undo-Button*. Dann wird eine Überprüfung durchgeführt, ob zuvor eine *Undo-Aktion* durchgeführt wurde. Das wird später noch einmal thematisiert. Eben dafür wird auch die Nachricht selbst am Ende des Codeblocks noch in einen Buffer geschoben. Die zentrale Aufgabe dieses Aufrufs ist jedoch das hinzufügen des in der Nachricht angegebenen Punktes. Das geschieht mittels der Funktion `fn push_vertex_to_buffer()`. Ihr wird der Punkt sowie der Buffer übergeben, in welchen er eingefügt werden soll. In diesem Fall ist es der Vektor `draw_panel.vertices`. Danach wird noch der Cache des Renderers zurückgesetzt, wodurch alle Inhalte auf dem Zeichenfenster neu gezeichnet werden.

Zeichenwerkzeuge

Der zweite Bereich des Menüs, welcher eng mit dem Zeichenfenster verbunden ist, sind die Zeichenwerkzeuge. Dafür wurde ein neuer Struct in der Datei `tools.rs` angelegt. Bei den Zeichenwerkzeug handelt es sich im Wesentlichen um Buttons, welche verschiedene Funktionalitäten aktivieren. Es gibt fünf solcher Buttons, welche jeder eine eigene Aufgabe erfüllen. Namentlich sind das der *Draw*-, der *Draw Hole*-, der *Undo*-, der *Redo*- und der *Clear*-Button. Auf jeden davon wird im folgenden einmal gesondert eingegangen. Der Tool-Struct umfasst also die Zustände der Buttons sowie einigen Boolean-Variablen, welche angeben, ob ein Button aktiv ist oder nicht. Des weiteren umfasst die Implementierung dieses Structs einen Konstruktor und eine weitere Funktion `pub fn tool_menu()`, welche das Layout der Zeichenwerkzeuge festlegt. Zu besseren Übersicht für den Nutzer

wurden diese fünf Buttons in einer umrahmten Gruppe rechts vom Zeichenfeld angeordnet.

Bevor nun auf die einzelnen Buttons und deren Funktionalität eingegangen wird, soll hier kurz beschrieben sein, wie ein `iced::button` überhaupt aufgebaut ist. Wie angedeutet, ist ein Button ein Element aus der Iced-Bibliothek. Es besitzt einen Zustand und ein Label, also eine Beschriftung. Zusätzlich gibt es noch einen Nachricht, welche beim drücken des Buttons ausgelöst wird und neben vielen weiteren Einstellungen wie Breite und Höhe auch noch einen Stile, welcher mittels eines `StyleSheets` festgelegt werden kann. Eine Nachricht wird dadurch durch den Befehl `mybutton.on_press(Message)` an den Button gebunden. Zu Stilen ist im Kapitel 4.3.4 mehr beschrieben. Hier einmal der grundsetzliche Aufruf eines neuen Buttons:

```
1    use iced::{Button, button, Text};
2    neuer_button = Button::new(button::State::new(),
3                   Text::new("Beschriftung"))
4                   .on_press(Message).style(ButtonStyle);
```

Das Layout, welches in der Funktion `pub fn tool_menu()` festgelegt wurde, ist im nachfolgenden Bild noch einmal als Ausschnitt aus dem vollständigen Menü zusehen.

Draw-Button

Dieser Button hat eine der wichtigsten Funktionen des Programms. Er aktiviert die Eingabe auf dem Zeichenfeld. Standardmäßig ist er aktiv, das heißt es ist möglich ihn zu drücken. Einmal gedrückt, wird er so lange deaktiviert, bis ein anderer Button gedrückt wurde. Während er deaktiviert ist, das heiß er gedrückt wurde, kann man nach belieben auf dem Zeichenfeld ein Polygon durch Mausklicks erzeugen. Ist dieses dann geschlossen, wird der Input wieder deaktiviert.

Draw-Hole-Button

Der *Draw-Hole-Button* soll, wie der *Draw-Button* den Input auf dem Zeichenfeld erlauben. Anders als bei *Draw* soll hier aber nicht auf dem ganzen Zeichenfeld gezeichnet werden, sondern nur innerhalb des Zuvor gezeichneten Polygons. Es sollen also Löcher zum Polygon hinzugefügt werden. Dies ist zum Zeitpunkt der Abgabe dieser Arbeit noch nicht implementiert, wird aber später hinzugefügt.

Undo-Button

Das Rückgängigmachen von Aktionen ist ein wichtiger Aspekt, wenn es um Nutzerfreundlichkeit geht. Daher wurde diese Funktion mittels des *Undo-Buttons* implementiert. Er wird erst aktiv, wenn bereits mindestens ein Punkt auf dem Zeichenfeld gezeichnet wurde. Wird er gedrückt, so wir der zuletzt gezeichnete Eckpunkt aus dem Vektor `draw_panel.vertices`, sowie die Nachricht `AddPoint`

`(Point)` aus dem `action_buffer` entfernt. In einen zweiten Buffer, den Vektor `undo_buffer` wird dann eine Kopie dieser entfernten Nachricht eingefügt. Sie enthält auch den gelöschten Punkt. Dies geschieht, damit er mittels des *Redo-Buttons* wieder hinzugefügt werden kann. Dazu im nachfolgenden Abschnitt mehr. Das drücken des *Undo-Buttons* löst die Nachricht `PageMessage::UndoPressed` aus, welche dann in der Funktion `fn update()` des Page-Struct behandelt wird. Sollte das Polygon zuvor geschlossen worden sein, dann muss dieser Zustand natürlich wieder aufgehoben werden und der Zustand des Zeichenautomaten `DarwState` muss wieder auf `Pending::WaitNxtInput` gesetzt werden. All das wird in der Update-Funktion umgesetzt. Auch wird eine Flag gesetzt, dass die zuletzt durchgeführte Aktion ein Rückgängigmachen war. Das wir für die Redo-Aktion relevant.

Redo-Button

Der *Redo-Button* stellt das logische Gegenstück zum *Undo-Button* dar. Aktionen, welche mittels Undo rückgängig gemacht wurden, können hiermit erneut durchgeführt wreden. Auch dieser Askept ist für die Nutzerfreundlichkeit sehr wichtig. Zunächst einmal ist dieser Button aber inaktiv. Erst, wenn eine Undo-Aktion durchgeführt wurde, kann der *Redo-Button* gedrückt werden. Wird der *Redo-Button* betätigt, löst er die Nachricht `PageMessage::RedoPressed` aus. Diese führt in der `fn update()` des Page-Structs zu verschiedenen Abläufen. Als erstes wird die letzte Nachricht aus dem `undo_buffer` entfernt und wieder in den `action_buffer` geschrieben. Der zuvor entfernte Punkt, wird wieder in den dafür vorgesehenen Speicher eingefügt. Sollte es noch weitere Aktionen geben, welche rückgängig gemacht wurden, dann bleibt der Button aktiv und kann erneut gedrückt werden.

In einem weiteren Fall, außer dem, dass der `undo_buffer` leer ist, wird der Button auch deaktiviert. Im Prinzip liegt das auch daran, dass der `undo_buffer` geleert wurde, aber auf andere Art. Das geschieht, wenn nach einem Undo ein Draw passiert ist. Zuvor wurde bei der Undo-Aktion eine Flag gesetzt, welche anzeigt, dass die letzte Aktion rückgängig gemacht wurde. Wird jetzt ein neuer Eckpunkt gezeichnet, dann wird der `undo_buffer` gelöscht. Das geschieht, damit keinen Punkte, welche gelöscht wurden an falscher Stelle wieder in den Vertex-Buffer eingefügt wird. Im nachfolgenden Bild ist das einmal aufgezeigt.

Clear-Button

Dieser Button bildet eine Ausnahme unter den Zeichenwerkzeugen, da seine Funktion eine extra Bestätigung durch ein Dialogfenster benötig, welches sich dann öffnet, wenn dieser Button gedrückt wurde. Dieses Dialogfenster ist kein Element der Iced-Bibliothek. Es gehört zur Bibliothek `iced_aw`, welche eine Erweiterung von `iced` darstellt und weitere weniger grundlegende GUI-Elemente

umfasst. Das verwendete Element ist eine `iced_aw::Card`, welche man sich als Fenster im Fenster vorstellen kann. Sie besteht aus Header, Body und Footer und sendet beim schließen eine Nachricht. Im Falle dieser Arbeit ist das `PageMessage::PopUpClosed`. Der Body der Karte enthält dabei die Warnung, dass nach der Bestätigung des Löschungsvorganges der gesamte Input auf dem Zeichenfenster gelöscht wird und dies nicht rückgängig gemacht werden kann. Diese Entscheidung wird mittels eines roten und eines grünen Buttons getroffen. Der rote *Reject-Button* sendet dabei die Nachricht `PageMessage::RejectClear` und setzt damit die gleich Funktionalität um, wie die Nachricht `PageMessage ::PopUpClosed`. Mit dem grünen *Yes-Clear-Button* wird die Löschung mittels der Nachricht `PageMessage::ClearAll` ausgelöst. Beide Buttons schließen die Karte mit dem Dialog und lassen den Ursprünglichen *Clear-Button* wieder erscheinen. Ein Beispiel für einen solchen Vorgang ist im folgenden Bild zu sehen.

Optionen

Der dritte und letzte große Bestandteil der Menü-Seite sind die Optionen. Diese sind ebenfalls noch einmal in drei Teile geteilt, welche optisch, wie auch die Zeichenwerkzeuge, durch einen Rahmen abgegrenzt werden. Bei den drei Sektionen handelt es sich um die Auswahl des Algorithmus, die Auswahl der anzuwendenden Heuristik und den weiteren Optionen. In den ersten beiden Bereichen werden die Auswahlmöglichkeiten durch eine Selektion mittels Radio Buttons abgebildet. Ein Radio Button bzw. eine Radio Button Gruppe hat die Eigenschaft, dass nur eine Option zur gleichen Zeit aktiv sein kann. Das bedeutet wenn ich drei Möglichkeiten (a), (b) und (c) habe, dann kann ich nur eine der drei, zum Beispiel (b), nicht aber zwei verschiedene, wie etwas (a) und (c), gleichzeitig auswählen. Die Iced-Bibliothek besitzt eine Implementierung für eben solche Radio Buttons. Man kann sie automatisch aus einem Aufzählungstypen generieren, wenn man für diesen eine Implementierung für `impl From<Algorithm> for String` und eine Funktion `fn all()` bereitstellt. Die Funktion gibt schlichtweg alle Ausprägungen des Aufzählungstypen aus. Die String-Implementierung generiert aus einer gebenen solchen Ausprägung eine Zeichenkette. Anhand der Algorithmus-Auswahl sieht das dann in etwa so aus:

```
1
2   pub enum Algorithm {
3       EarClipping,
4       DelaunyTriangulation,}
5
6   impl<'a> Algorithm {
7       pub fn all() -> [Algorithm; 2] {
8           [Algorithm::EarClipping,
9            Algorithm::DelaunyTriangulation,]
10      }}
```

```
11
12      impl From<Algorithm> for String {
13          fn from(algorithm: Algorithm) -> String {
14              String::from(match algorithm {
15                  Algorithm::EarClipping => "Ear Clipping",
16                  Algorithm::DelaunyTriangulation =>
17                      "Delauny Triangulation",
18              })
19          }}
```

Man bemerke, dass hier zu Beispielzwecken die DT als Option aufgeführt ist. Diese ist in der beiliegenden Version des Programms nicht anwählbar. Sie ist als Idee der Programmerweiterung im Ausblick noch einmal angeführt. Die Umsetzung einer automatischen Generierung der Radio Buttons würde dann in etwa so aussehen:

```
1       Algorithm::all().iter().cloned().fold(
2           Column::new().padding(10).spacing(15),
3           |choices, algorithm| {
4               choices.push(Radio::new(
5                   algorithm,
6                   algorithm,
7                   selection,
8                   PageMessage::AlgorithmSelected,
9               ).text_size(20).size(15))
10          },
11      )
```

Wie für die Algorithmen in der Datei `algorithm.rs` wird nach dem gleichen Schema die selbe Implementierung für die Heuristiken in der Datei `heuristc .rs` vorgenommen. In der Datei `program_settings.rs` wir nun der `struct ProgramSettings` deklariert, welcher neben `algorithm: Option<Algorithm>` und `heuristic: Option<Heuristic>` auch `bools: ProgramSettingsBools`. Dieses Feld umfasst die weiteren Optionen, welche man wahlweise aus- oder anschalten kann. In der finalen Version der Software sind dies zwei Möglichkeiten. Die Option *Stepwise Triangulation* für die Auswahl, ob der Algorithmus in Schritten oder im Ganzen ausgeführt werden soll, und die Option für den Dark Mode, welcher in Kapitel 4.3.4 beschrieben wird. Diese Optionen werde durch sogenannte Toggler abgebildet. Ein Toggler hat neben einer Beschriftung auf eine Variable vom Typ Boolean, welchen er je nach seiner eigenen Stellung auf *ein* oder *aus* auf `true` oder `false` setzt. Anhand der Option des Dark Modes sieht das dann so aus:

```
1       Toggler::new(
2           bools.dark_mode,
3           String::from("Dark Mode"),
4           PageMessage::DarkModeToggled,
5       ).size(15).text_size(20)
```

Seitenübergang

Zusetztlich zu den drei großen Bereichen des Menüs gibt es noch ein weiteres Interaktionselement. Dies ist ein Button, welcher unter den Optionen am rechten Rand des Fensters platziert wurde. Dies ist der *Confirm-Button*. Dieser ist, im Gegensatz zu den Zeichenwerkzeugen kein Button mit Sekundärstil sondern ein Primärbutton. Er betsätigt nämlich alle Einstellungen und Eingaben der Menüseite und regelt den Übergang auf die Iterartionsseite. Dazu wird die Nachricht `PageMessage::ConfirmPressed` gesendet. Damit das eingegebene Polygon auch auf der nächsten wie auch letzten Seite angezeigt wird, gibt es im Page-Struct mehrere Funktionen, welche aufgerufen werden, sobald ein Seitenübergang stattfindet. Zuerst einmal muss der Speicher, welcher alle eingegebenen Eckpunkte beinhaltet ausgelesen werden. Dies geschieht mittels der Funktion `fn get_vertex_buffer(&mut self)`. Wurde der Buffer ausgelesen, dann muss er in das Koordinatensystem des Anzeigefensters der nächsten Seite konvertiert werden. Das ist notwendig, da alle Anzeigefensters unterschiedliche Größen besitzen. Das bewerkstelligt die Funktion `fn buffer_move_center(buffer: Vec<Point>, offset_x: f32, offset_y: f32)`. Sie erhält als Eingaben neben dem Buffer auch den jeweiligen Offset in x- und y-Richtung zwischen den beiden Koordinatensystemen. Ist die Transformation geschehen, wird der Buffer mittels der Funktion `fn set_vertex_buffer(&mut self, buffer: Vec<Point>)` auf die nächste Seite kopiert.

Für den Übergang zwischen Menü und Iteration genügt das. Für den später noch folgenden Übergang zwischen Iteration und Resultatsseite muss zusätzlich noch der Speicher für die erzeugten Diagonalen übergeben werden. Dies geschieht nur mittels Kopiervorgang, da keine Koordinaten sondern Anfangs- und Endpunkte der Strecken gespeichert wurden.

4.3.2 Page - Iteration

4.3.3 Page - Result

4.3.4 Style und Dark Mode

Ein großer Aspekt bei der Umsetzung von GUIs sind Farben. So haben bestimmte Farben für den Menschen bestimmte Bedeutungen - etwa Rot für Gefahr oder Ablehnung. Dies nutzt man, um dem Benutzer bestimmte Botschaften zu übermitteln. So sind farblich hervorgehobene Buttons zum Beispiel wichtiger als graue. Hierfür müssen diese Farben festgelegt werden. Das geschieht in der Datei `style.rs` mittels der Implementierung von `StyleSheets` für verschiedene Aufzählungstypen. Dabei wird für jedes Interaktionselement wie Buttons, Radio Buttons aber auch für das Fenster selbst ein `enum` erstellt, welcher alle verschiedenen Möglichkeiten für Stile des jeweiligen Elements umfasst.

Hier soll einmal exemplarisch gezeigt werden, wie ein solches `StyleSheet` für einen `iced::container` aussieht. Dieser beinhaltet in jeder View-Funktion den Inhalt einer Seite und umfasst derher die Hintergrundfarbe `backgroundcolor` und die Schriftfarbe `text_color` des Anzeigefensters. Zuerst wird im Aufzählungstypen `enum WindowStyle` festgelegt, wie viele unterschiedliche Stile ein solcher Container haben kann. Danach wird für jeden Stil pro Eigenschaft eine Farbe festgelegt. Alle Eigenschaften, welche dem festgelegten Standard der Iced-Bibliothek folgen sollen, werden mittels `..Style::default()` abgebildet. So gibt es für Container in dieser Arbeit zwei Stile, wie für die meisten anderen Element auch. Die einzige Ausnahme bilden die Buttons, welche eine rot und grüne, sowie eine helle und eine dunkle Variante für Primär- und Sekundärbuttons besitzen. Alle anderen Elemente des GUI haben eine helle und eine dunkle Variante. Der sogenannte *Dark Mode* ist eine weit verbreitete Option für vielerlei Anwendungsprogramme. Da er für einige Nutzer ein obligatorischer Modus ist, ist er auch in dieser Arbeit implementiert. Er wird wie bereits erwähnt durch einen Toggler im Menü ein- oder ausgeschalten. Dadurch ändern sich alle Stile von der hellen auf die dunkel Variante oder umgekehrt. Standardmäßig ist der Dark Mode allerdings aus.

```
1
2    pub enum WindowStyle {
3        Light,
4        Dark
5    }
6
7    impl container::StyleSheet for WindowStyle {
8        fn style(&self) -> container::Style {
9            container::Style {
10
11               background:  Some(Background::Color(match self {
12                   WindowStyle::Light => Color::WHITE,
13                   WindowStyle::Dark => Color::from_rgb8(0x57, 0x57
   , 0x57),
14               })),
15               text_color: match self {
16                   WindowStyle::Light => Some(Color::from_rgb8(0x57
   , 0x57, 0x57)),
17                   WindowStyle::Dark => Some(Color::WHITE),
18               },
19               ..container::Style::default()
20           }
21       }
22   }
```

Literatur

[1] *Digitale Revolution*
(https://www.staatslexikon-online.de)

[2] *Darstellung von Kurven und Flächen*, Christoph Dähne
(https://www.inf.tu-dresden.de)

[3] *Geometrical Mesh Quality*
(https://www.iue.tuwien.ac.at)

[4] *Sliver In: Computer Graphics Dictionary*, Stevens, R.T., 2002.

[5] *Triangulation by Ear Clipping*, David Eberly
(https://www.geometrictools.com)

[6] *Polygon Triangulation*, Subhash Suri
(https://sites.cs.ucsb.edu/ suri/cs235/Triangulation.pdf)

[7] *Parallelized ear clipping for the triangulation and constrained Delaunay triangulation of polygons*, Günther Eder, Martin Held, Peter Palfrader
(https://www.sciencedirect.com)

[8] *Improved Algoritms For Ear-CLipping Triangulation*, Bartosz Kajak, 2005
(https://digitalscholarship.unlv.edu/thesesdissertations/1319/)

[9] *A comparison of Ear Clipping and a new Polygon Triangulation Algorithm*, Ran Liu
(https://www.diva-portal.org/smash/get/diva2:330344/FULLTEXT02.pdf)

[10] *Polygon, Definition*
(https://mathepedia.de/Polygone.html)

[11] *Regular Polygons. In: Michiel Hazewinkel (Hrsg.): Encyclopedia of Mathematics. Springer-Verlag und EMS Press, Berlin 2002*

[12] *Convex Polygon*
(https://www.mathopenref.com/polygonconvex.html)

[13] *Triangulation, Definition*
(https://encyclopediaofmath.org/wiki/Triangulation)

[14] *Simplex, Definition*
(https://encyclopediaofmath.org/wiki/Simplex)

[15] *Ear-clipping Based Algorithms of Generating High-quality Polygon Triangulation*, Gang Mei, John C.Tipper and Nengxiong Xu
(https://arxiv.org/ftp/arxiv/papers/1212/1212.6038.pdf)

[16] *Ear-Clipping Triangulierung*
(wiki.delphigl.com)

[17] *Jordanscher Kurvensatz*
(https://de.wikipedia.org)

[18] *The smallest 8 cubes to cover a regular tetrahedron*
(https://math.stackexchange.com/)

[19] *Slicing an ear using prune-and-search In: Pattern Recognition Letters*, ElGindy, H., Everett, H., and Toussaint, G. T., (1993) S. 719-722

[20] *Polygons Have Ears In: Amer. Math. Monthly*, G.H. Meisters, Ausgabe 82, S. 648–651, 1975.

[21] *Turing-Maschinen In: Der Turing Omnibus*, A. K. Dewdney, S. 211-230, 1975.

[22] *Computational Geometry In: C. Cambridge: Cambridge University Press*, O'Rourke, J., (1998).

[23] *Triangulating a polygon in parallel In: Journal of Algorithms*, M. Goodrich, Ausgabe 10, S. 327-351, 1989.
(https://www.sciencedirect.com/)

[24] *FIST: Fast Industrial-Strength Triangulation of Polygons In: Algorithmica*, Held, M., Ausgabe 30, S. 563–596, 2001.
(https://link.springer.com/article/10.1007/s00453-001-0028-4)

[25] *Reentrant Polygon Clipping*, Sutherland, Ivan E. und Hodgman, Gary W., 1974.
(https://dl.acm.org/doi/abs/10.1145/360767.360802)

[26] *Delauny Triangulation*
(https://ti.inf.ethz.ch/ew/Lehre/CG13/lecture/Chapter%206.pdf)

[27] *Triangulation by Ear Clipping*, David Eberly, Geometric Tools, Redmond WA, 2002
(https://www.geometrictools.com/Documentation/TriangulationByEarClipping.pdf)

[28] *Detecting Weakly Simple Polygons*, Hsien-Chih Chang, Jeff Erickson, Chao Xu, Proceedings of the 2015 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), S. 1655 - 1670,2015
(https://jeffe.cs.illinois.edu/pubs/pdf/weak.pdf)

[29] *Perfect Spatial Hashing*, Sylvain Lefebvre Hugues Hoppe, Microsoft Research
(https://hhoppe.com/perfecthash.pdf)

[30] *Rust, offizielle Website*
(https://www.rust-lang.org/)

[31] *Rust, Wikipedia*
(https://de.wikipedia.org/)

[32] *Iced, Github*
(https://github.com/iced-rs/iced)

[33] *Vulkan, offizielle Website*
(https://www.vulkan.org/)

[34] *Metal, offizielle Website*
(https://developer.apple.com/metal/)

[35] *OpenGL, offizielle Website*
(https://www.opengl.org/)

[36] *OpenGL ES, offizielle Website*
(https://www.khronos.org/opengles/)

[37] *DirectX 12, offizielle Website von NVIDIA*
(https://www.nvidia.com/de-de/geforce/technologies/dx12/)