Institut für Formale Methoden der Informatik

Universität Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit Nr. 90

Algorithmen zur Optimierung von geometrischen Packungsproblemen

Sergej Geringer

Studiengang:	Informatik
Prüfer/in:	Prof. Dr. Stefan Funke
Betreuer/in:	Prof. Dr. Stefan Funke
Beginn am:	4. Oktober 2013
Beendet am:	5. April 2014
CR-Nummer:	F.2.2

Kurzfassung

Geometrische Packungs-Probleme sind in vielen industriellen Prozessen anzutreffen. Dadurch motiviert wird in dieser Arbeit untersucht, wie geometrische Formen möglichst oft in einem rechteckigen Gebiet platziert werden können.

Um zu garantieren, dass Platzierungen von Formen ohne Überschneidungen sind, wird ein rasterbasierter Schnitttest realisiert, der unter Verwendung der OpenGL-API die Erkennung von Schnitten komplett auf der Grafikkarte durchführt. Die Problemstellung wird anhand von einfachen Polygonen modelliert und mithilfe der geometrischen Eigenschaften der Polygone untersucht. Dafür werden mögliche Platzierungen von Polygonen eingeschränkt und mit Hilfe des Schnitttests auf Überschneidungen geprüft. Eine Datenstruktur zur Verwaltung schnittfreier Polygon-Stellungen wird entwickelt; damit zusammenhängend werden Heuristiken vorgestellt, anhand derer solche Polygon-Stellungen bewertet werden können. Weiterhin werden Strategien diskutiert, die die Anzahl betrachteter Polygon-Stellungen, und somit nötiger Schnitttests, erheblich reduzieren. Diese Strategien orientieren sich an den geometrischen Eigenschaften der Polygone, sowie an strukturellen Eigenschaften der verwendeten Datenstruktur. Durch das iterative Aufbauen lokal enger und schnittfreier Polygon-Platzierungen werden globale Lösungen für das rechteckige Gebiet konstruiert.

Anhand einer Software-Implementierung werden die dargelegten Strategien evaluiert und als effizient erachtet.

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

1 Einleitung

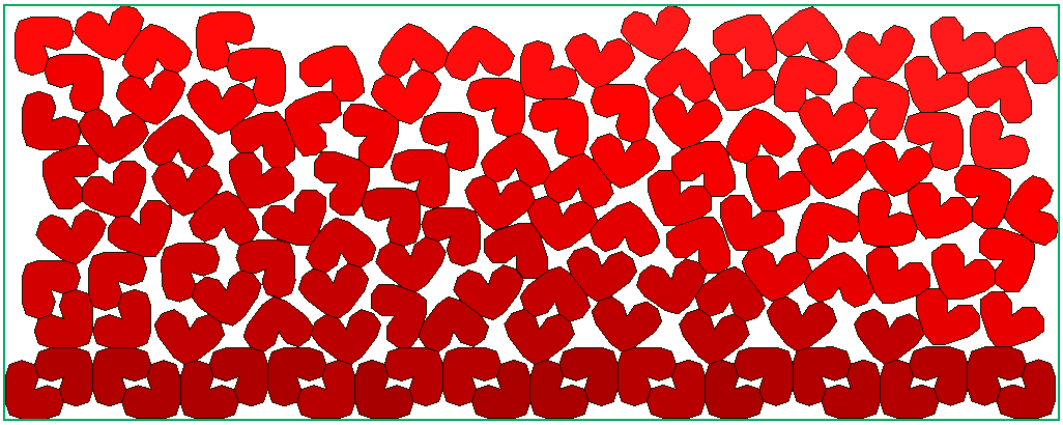

Abbildung 1.1: Eine Platzierung von 128 Formen auf einem rechteckigen Gebiet.

Problemstellung

Man stelle sich vor es ist Weihnachten und es werden Plätzchen gebacken. Aus dem *rechteckig* ausgerollten Plätzchenteig sollen *maximal viele* Plätzchen ausgestochen werden, ohne den Teig neu auszurollen. Eine andere Variante dieser Problemstellung lautet, aus einem rechteckigen Metallblech möglichst viele vorgegebene Formen zu schneiden, um den Verschnitt an Material gering zu halten und somit Materialkosten zu sparen.

In beiden Fällen liegt das Optimierungsproblem vor, auf einem gegebenen Gebiet eine Anordnung von vorgeschriebenen Formen zu finden, die den genutzten Platz im Gebiet maximiert, indem maximal viele Formen platziert werden. Die Formen können beliebig verschoben und gedreht werden, sollen aber platziert werden, ohne sich gegenseitig zu überdecken, bzw. zu überschneiden. Ein Beispiel einer Platzierung von Formen ist in Abbildung 1.1 zu sehen. Dieses *Packungs-Problem*, *Nesting-Problem*, oder *Schachtelungs-Problem* ist NP-schwer[AI12]. Das bedeutet, dass im Grunde alle möglichen Stellungen von Formen ausprobiert werden müssen um eine optimale Verteilung auf dem Gebiet zu finden [1]. Statt Menschen mit dem Finden solcher Platzierung zu beauftragen liegt es nahe sich von entsprechenden Computer Programmen Lösungen für gegebene Probleminstanzen berechnen zu lassen. Viele oder alle Möglichkeiten für Platzierungen durchzuprobieren kann, selbst mit Computern, hoffnungslos

[1] Dies gilt natürlich nur unter der Annahme, dass $P \neq NP$. Diese Annahme wird hier getroffen.

lange dauern, wenn nicht besondere Strategien zum Einsatz kommen, die eine Auswahl von nur vielversprechenden Lösungswegen sicherstellen.

Diese Arbeit untersucht Methoden und Algorithmen, wie das Nesting-Problem von Formen auf einem rechteckigen Gebiet gelöst werden kann. Unter Einschränkung der möglichen Platzierungen werden Heuristiken entwickelt, die eine Lösungsfindung beschleunigen. Es werden Datenstrukturen und Vorgehensweisen erläutert um Platzierungen von Formen zu verwalten und zu bewerten, und es wird gezeigt wie unter Nutzung dieser Strukturen überflüssiger Rechenaufwand vermieden werden kann. Um zu erkennen wann sich Platzierungen von Formen überlagern wird ein Schnitttest entwickelt der komplexe Berechnungen auf die Grafikkarte auslagert, und somit Gebrauch von deren spezialisierter Hardware macht. Anhand einer Implementierung der vorgestellten Verfahren wird evaluiert, dass diese eine schnellere Lösungsfindung ermöglichen als ein naiver Brute-Force Ansatz.

Bekannte Ansätze

Packungs-Probleme gibt es in zahlreichen Varianten, die in verschiedenen Anwendungsbereichen auftauchen. Vom optimalen Zuschneiden von Blechen und Textilien[Nie07], über das optimale Packen von Kartons auf Paletten bis hin zum optimalen Packen von Kofferräumen[EFRS03][EFK+05] sind verschiedene Packungs-Probleme anzutreffen, sowohl in zwei- als auch in drei Raumdimensionen und mit zu platzierenden Formen verschiedenster Arten. Die Relevanz solcher Probleme für industrielle Fertigungsprozesse ist leicht ersichtlich, und so verwundert es nicht, dass der Autor dieser Arbeit bei ersten Recherchen zu diesem Thema auf einen Artikel des *Spiegel* stieß[2], der dieses Thema behandelt. Neben der naheliegenden, in dieser Arbeit eingangs ebenfalls gewählten, Einführung in die Problemstellung anhand des weihnachtlichen Plätzchen Backens geht der Spiegel-Artikel auf eine Software namens *AutoNester*[3] ein, die vom Frauenhofer Institut für Algorithmen und Wissenschaftliches Rechnen entwickelt wird. Der AutoNester ist ein Programm das zweidimensionale Packungs-Probleme lösen kann, und gleichzeitig die Einhaltung verschiedener Randbedingungen erlaubt. Als solches ist der AutoNester ein Beispiel für den Bedarf an schnellen Lösern für Packungs-Probleme.

Es besteht ein breites Forschungsgebiet, das stets neue Lösungsstrategien für Nesting-Probleme entwickelt. Neben der Behandlung grundsätzlicher Strategien[BO08] liegt der Fokus auf dem Finden schneller Algorithmen[OGF00], die gute Lösungen in weniger Zeit berechnen. Hierbei werden verschiedene Ansätze verfolgt, die Lösungen zum Beispiel mittels Lineare Programmierung[EFRS03][GO06] oder *Simulated Annealing*[OF93][GO06] finden, und oft mit geometrischen Konstruktionen wie No-Fit Polygons[Gho91] arbeiten. Die meisten Optimierungsverfahren für Packungs-Probleme müssen früher oder später auf eine randomisierte Lösungsfindung zurück greifen. Hierbei wird eine vorhandene Lösung der Probleminstanz benutzt, um über zufällige Veränderungen der platzierten Formen zu einer bessere Lösung zu gelangen. Solche Verfahren verfolgen die Strategie mit mehr investierter Rechenzeit potentiell bessere Lösungen zu finden. Um eine nötige initiale (Teil-)Lösung für solche

[2] Dambek, Holger. Spiegel Online. *Mathematik im Advent: Plätzchen backen für Perfektionisten.* Dezember 2010. http://www.spiegel.de/wissenschaft/mensch/a-733067.html, letzter Zugriff: 1. April 2014.

[3] Frauenhofer SCAI, http://www.scai.fraunhofer.de/geschaeftsfelder/optimierung/produkte.html, letzter Zugriff: 1. April 2014.

randomisierten Verfahren zu erhalten sind wiederum weitere Strategien nötig, die nicht zwingend eine optimale, aber eine gute Lösung in kurzer Zeit liefern sollen.

Das Finden solcher initialen Lösungen ist häufig stark von der genauen Formulierung der Problemstellung abhängig. Fast alle Varianten des Packungs-Problems haben aber gemeinsam, dass schon das Finden einer guten initialen Lösung einen erheblichen Rechenaufwand bedeuten kann. Hinzu kommen weitere berechnungsintensive Verfahren, die zum Beispiel das Einhalten von Randbedingungen garantieren müssen. Vor allem muss sichergestellt sein, dass gefundene Lösungen keine Formen enthalten, die sich überlagern oder überschneiden. Dies erfordert einen Test der erkennt, wann Platzierungen Überschneidungen enthalten; und weil dieser Test häufig durchgeführt werden muss, sollte dieser ebenfalls möglichst performant sein.

Herangehensweise und Aufbau der Arbeit

Diese Arbeit untersucht, wie geometrische Formen auf ein rechteckiges Gebiet möglichst oft platziert werden können. Hierbei können die Formen beliebig auf dem Gebiet verschoben und rotiert werden, sie dürfen sich jedoch nicht überschneiden. Um die Problemstellung etwas zu vereinfachen, werden nur solche Formen betrachtet, die keine Löcher und keine runden Stellen haben. Hierdurch können diese Formen durch einfache Polygone modelliert werden. Das rechteckige Gebiet sei durch seine Höhe und Breite gegeben.

Statt einen globalen Ansatz zum Platzieren der Formen zu verfolgen, werden Eigenschaften der gegebenen Formen (Polygone) genutzt, um zunächst lokal kleine Teillösungen zu erzeugen. Vorrangig wird untersucht, wie Polygone miteinander möglichst eng platziert werden können, und wie der Rechenaufwand für solche Platzierungen minimiert werden kann. Um zu garantieren, dass Platzierungen, bzw. Stellungen von Polygonen schnittfrei sind, wird ein Rasterbasierter Schnitttest entwickelt und unter Verwendung der OpenGL-API auf Grafikkarten umgesetzt, hierauf wird in Kapitel 3 ausführlich eingegangen. Kapitel 2 widmet sich der Aufarbeitung mathematischer und algorithmischer Verfahren und Strukturen, die für die Modellierung des Problems und für die Untersuchung von Polygonen und deren Platzierungen notwendig sind.

Die Untersuchung des Packungs-Problems erfolgt in Kapitel 4. Hier wird zunächst eine Einschränkung möglicher Platzierungen vorgenommen, um die Komplexität der Problemstellung zu reduzieren. Diese Einschränkung besteht darin, nicht grundsätzlich alle Platzierungen von Polygonen zu betrachten, sondern nur solche, bei denen die Polygone direkt zusammenliegen. Polygone sollen sich nur an den sie definierenden Punkten berühren und aneinander ausgerichtet sein. Dies kann als Einschränkung auf *diskrete* Stellungen der Polygonene zueinander aufgefasst werden. Solche Polygon-Stellungen bieten eine vereinfachte Betrachtung von möglichst engen Platzierungen von Polygonen, müssen aber anhand des Schnitttests auf Überschneidungen geprüft werden. Anhand der diskreten Stellungen werden in Kapitel 4 Strategien entwickelt, um mögliche aber irrelevante Stellungen zu vermeiden, um somit die zahlreichen und zeitintensiven Schnitttests zu reduzieren. Hierfür werden einerseits geometrische Eigenschaften der Polygone aus Kapitel 2.2 aufgegriffen, und andererseits werden strukturelle Eigenschaften der Polygon-Stellungen genutzt, um irrelevante Gebiete von Polygon-Stellungen von Berechnungen auszuschließen und alte Informationen aus Schnitttests wiederzuverwenden.

Aus iterativ erstellten Teillösungen können so schneller globale Lösungen für das gesamte Gebiet zusammengesetzt werden.

Die in dieser Arbeit entwickelten Strategien werden anhand einer Software-Implementierung evaluiert. Dies geschieht durch das Betrachten von Testinstanzen und daraus erhobener Daten in Kapitel 5. Im darauf folgenden und letzten Kapitel schließt die Arbeit mit einer Betrachtung und Zusammenfassung der Ergebnisse, die im Laufe dieser Ausarbeitung entstanden sind.

2 Mathematische und algorithmische Grundlagen

Die Modellierung und Darstellung von Formen, also Polygonen, erfolgt anhand von Punkten und Richtungen im zweidimensionalen Raum. Dies sind Orts- und Richtungsvektoren v, r in der zweidimensionalen Ebene $\mathbb{R}^2$, sprich $v, r \in \mathbb{R}^2$. Um das Drehen und Verschieben von Polygonen zu realisieren, werden Transformationen in Form von Rotationen und Translationen benutzt.

Nach der Einführung von Transformationen folgt eine Definition der hier betrachteten Polygone und ihrer Eigenschaften, gefolgt von Definitionen und Analysen weiterer geometrischer Objekte und Algorithmen. Diese geometrischen Objekte und Algorithmen dienen im weiteren Verlauf der Arbeit als Hilfsmittel, um Eigenschaften von Polygonen (und die Konstellation vieler dicht nebeneinander platzierter Polygone) zu betrachten und zu bewerten. Namentlich werden einfache Polygone betrachtet, deren Konvexe Hülle, das minimale umschliessende Rechteck, die Triangulierung eines Polygons, und die Verdeckung eines Punktes im Raum durch ein oder mehrere Polygone.

2.1 Vektorrechnung und Transformationen

Da die Problemstellung dies bedingt, ist es nötig das Drehen und Verschieben von Objekten im zweidimensionalen Raum zu realisieren. Hierfür werden zunächst die Transformationen *Rotation* und *Translation* eingeführt und anschliessend in eine leicht handhabbare Form gebracht. Dies passiert, indem die nötigen Transformationen in sogenannten *Homogenen Koordinaten* durchgeführt werden.

Translation, Skalarprodukt, Winkel, Drehung, Skalierung

Gegeben seien Punkte, bzw. Ortsvektoren $v \in \mathbb{R}^2$, (ggf. normierte) Richtungsvektoren $r \in \mathbb{R}^2$ und sich auf einen solchen Richtungsvektor beziehende Normale $n_r \in \mathbb{R}^2$. Die Normale wird meistens ebenfalls als normiert angenommen, als Konvention wird im Folgenden auch davon ausgegangen, dass eine Normale nach *links* vom zugehörigen Richtungsvektors zeigt.

Die erste Transformation, die für unsere Anwendungen benötigt wird, ist die Verschiebung, also *Translation* $T(v, r)$ eines Vektors $v \in \mathbb{R}^2$ um einen bestimmten Richtungsvektor $r \in \mathbb{R}^2$.

$$\begin{aligned} v' &= T(v, r) \\ &= v + r = \begin{pmatrix} x_v \\ y_v \end{pmatrix} + \begin{pmatrix} x_r \\ y_r \end{pmatrix} = \begin{pmatrix} x_v + x_r \\ y_v + y_r \end{pmatrix}. \end{aligned}$$

Das *Skalarprodukt* zwischen zwei Vektoren $r_1, r_2 \in \mathbb{R}^2$ ist gegeben durch

$$\langle r_1, r_2 \rangle = r_1 \cdot r_2 = \begin{pmatrix} x_1 \\ y_1 \end{pmatrix} \cdot \begin{pmatrix} x_2 \\ y_2 \end{pmatrix} = x_1 \cdot x_2 + y_1 \cdot y_2 = s \in \mathbb{R}.$$

Das Skalarprodukt zweier Vektoren ist Null, wenn die beiden Vektoren orthogonal aufeinander stehen, wenn also einer der beteiligten Vektoren eine Normale des anderen ist. Das Skalarprodukt hat eine besondere Beziehung zum Winkel zwischen zwei Vektoren, die anhand der Berechnung des Winkels klar wird. Der *Winkel* $\alpha \in [0, \pi]$ zwischen zwei Vektoren $r_1, r_2 \in \mathbb{R}^2$ ist gegeben durch

$$cos(\alpha) = \frac{\langle r_1, r_2 \rangle}{|r_1| \cdot |r_2|}.$$

Der durch $acos(\frac{\langle r_1, r_2 \rangle}{|r_1| \cdot |r_2|})$ berechnete Winkel α bezieht sich stets auf den kleineren der beiden Winkel zwischen r_1 und r_2.

Anhand des Skalarproduktes lässt sich testen, auf welcher Seite einer Linie ein gegebener Punkt liegt. Seien gegeben ein Punkt $p \in \mathbb{R}^2$ und eine Linie die durch zwei Punkte $v_1, v_2 \in \mathbb{R}^2$ verläuft. Sei r_v der Richtungsvektor von v_1 nach v_2 und r_p der Richtungsvektor von v_1 nach p. Das Skalarprodukt zwischen r_p und der (links orientierten) Normale n_{r_v} zur Linie gibt Auskunft über die Lage von p bezüglich der Linie.

$$\langle r_p, n_{r_v} \rangle > 0 \Leftrightarrow p \text{ liegt links von der Linie durch } v_1, v_2.$$

Ist dieses Skalarprodukt Null, liegt der Punkt auf der Linie, ist das Skalarprodukt negativ, liegt der Punkt auf der rechten Seite der Linie. Eine Darstellung der hier betrachteten Punkte und Vektoren ist in Abbildung 2.1 zu sehen.

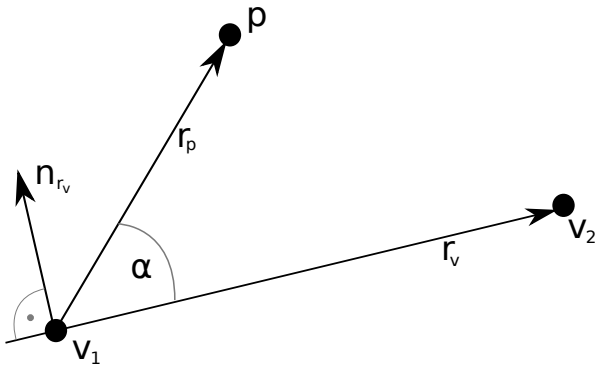

Abbildung 2.1: Zwei Richtungsvektoren r_p, r_v, der eingeschlossene Winkel α und die Normale n_{r_v} zu r_v.

Die *Rotation* eines Vektors $v \in \mathbb{R}^2$ um einen Winkel α ist gegeben durch die Multiplikation mit der Rotations-Matrix $R(\alpha)$

$$R(\alpha) = \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) \\ \sin(\alpha) & \cos(\alpha) \end{pmatrix}$$
$$v' = R(\alpha) \cdot v.$$

Der Vektor v wird um den *Ursprung* und *gegen den Uhrzeigersinn* gedreht. Eine Rotation *im Uhrzeigersinn* ergibt sich aus der Inversen von $R(\alpha)$. Die Inverse der Rotation $R(\alpha)$ ist eine Rotation um $R(-\alpha)$. Setzt man nun $-\alpha$ ein, ergibt sich über die Symmetrie vom Cosinus und der Punktsymmetrie vom Sinus

$$\begin{aligned} R(\alpha)^{-1} = R(-\alpha) &= \begin{pmatrix} \cos(-\alpha) & -\sin(-\alpha) \\ \sin(-\alpha) & \cos(-\alpha) \end{pmatrix} \\ &= \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix} \\ &= R(\alpha)^\top \end{aligned}$$

Ein normalisierter Richtungsvektor, der zwischen sich und der x-Achse einen Winkel α einschließt, hat in seinen Koordinaten den entsprechenden $\cos(\alpha)$ und $\sin(\alpha)$ Wert, dies ist am Einheitskreis leicht zu erkennen. Durch diesen Zusammenhang lassen sich nun einerseits gegebene Richtungsvektoren r sehr einfach in die Richtung $\begin{pmatrix} 1 & 0 \end{pmatrix}^\top$ rotieren, und andererseits lässt sich $\begin{pmatrix} 1 & 0 \end{pmatrix}^\top$ in jede gegebene andere Richtung rotieren. Dies ist angenehm, da sich hierdurch grundsätzlich Rotationen in gegebene Richtungen durchführen lassen, ohne je explizit die nötigen Winkel ausrechnen zu müssen. Sei also $r(\alpha) = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \end{pmatrix}^\top \in \mathbb{R}^2$ ein normalisierter Richtungsvektor, so rotiert folgende inverse Rotationsmatrix den Vektor r in Richtung $\begin{pmatrix} 1 & 0 \end{pmatrix}^\top$:

$$R(r(\alpha))^{-1} = \begin{pmatrix} r_x & r_y \\ -r_y & r_x \end{pmatrix} = \begin{pmatrix} \cos(\alpha) & \sin(\alpha) \\ -\sin(\alpha) & \cos(\alpha) \end{pmatrix}$$

Die *Skalierung* eines Vektors $v \in \mathbb{R}^2$ erfolgt durch die Skalierungsmatrix

$$\begin{aligned} S(s_x, s_y) &= \begin{pmatrix} s_x & 0 \\ 0 & s_y \end{pmatrix} \\ v' &= S(s_x, s_y) \cdot v \\ s_x, s_y &> 0 \end{aligned}$$

Ein Beispiel für eine hintereinander Ausführung aller Transformationen ist in Abbildung 2.2 zu sehen.

Ein grundsätzliches Problem der Translation ist, dass diese als Vektoraddition und nicht als Matrixmultiplikation realisiert ist. Für eine einfache Handhabung der Transformationen ist jedoch ein ausschließliches Rechnen mit Matrizen erwünscht, das wie folgt aussehen sollte:

$$\begin{aligned} v' &= S(s_x, s_y) \cdot R(\alpha) \cdot T(r) \cdot v \\ v'' &= T(r) \cdot S(s_x, s_y) \cdot R(\alpha) \cdot v. \end{aligned}$$

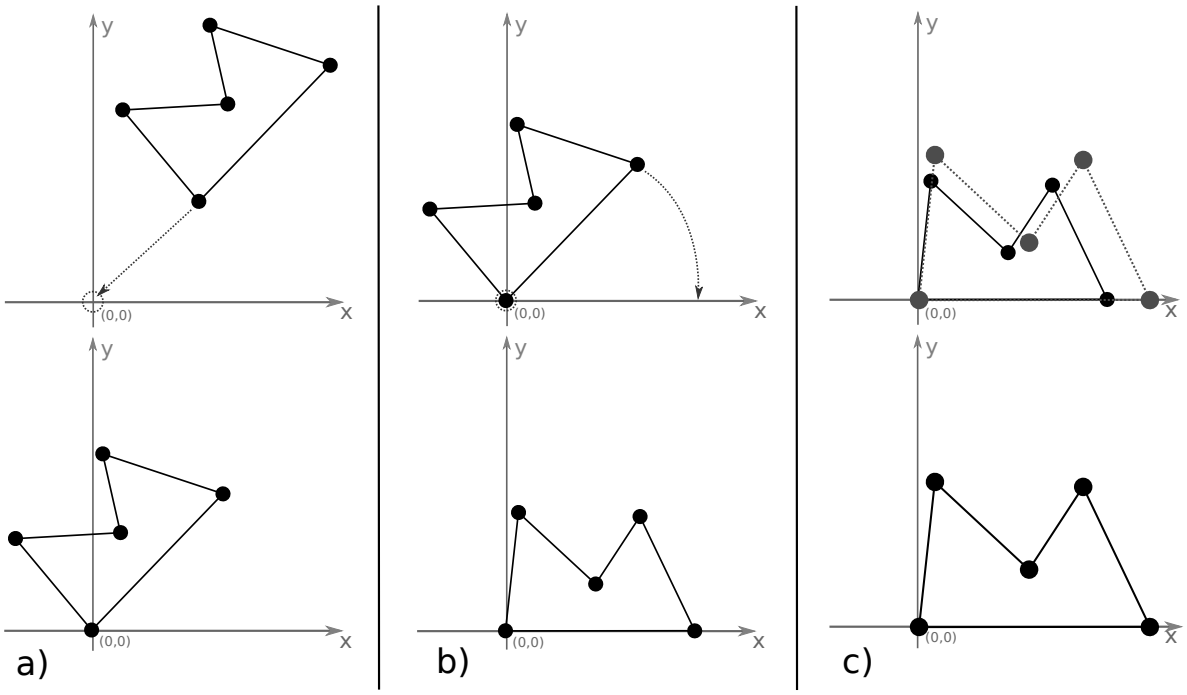

Abbildung 2.2: a) Translation von Punkten. **b)** Rotation von Punkten im Uhrzeigersinn. **c)** Skalierung einer Punktmenge.

Homogene Koordinaten für Transformationen

Die oben geforderte Eigenschaft für Transformationen kann durch Verwenden *Homogener Koordinaten* sichergestellt werden. Im Folgenden werden die bereits vorgestellten Transformationen in homogene Koordinaten überführt. Hierbei handelt es sich um eine Erweiterung der bekannten Matrizen und Vektoren um eine weitere Dimension, sodass Rechnungen fortan im dreidimensionalen $\mathbb{R}^3$ statt finden, aber immer noch für Operationen im zweidimensionalen $\mathbb{R}^2$ stehen. Die zusätzliche dritte Dimension wird in unserer Anwendung durchgehend durch eine hinzugefügte 1 realisiert. Sei $v \in \mathbb{R}^2$, so ist v in homogenen Koordinaten dargestellt durch

$$v = \begin{pmatrix} x \\ y \end{pmatrix} \mapsto \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \in \mathbb{R}^3$$

Folgende *Translations-Matrix* gibt eine Translation um den Vektor r in homogenen Koordinaten an.

$$
\begin{aligned}
T(r) &= \begin{pmatrix} 1 & 0 & r_x \\ 0 & 1 & r_y \\ 0 & 0 & 1 \end{pmatrix} \\
v' &= T(r) \cdot v \\
&= \begin{pmatrix} 1 & 0 & r_x \\ 0 & 1 & r_y \\ 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} x \\ y \\ 1 \end{pmatrix} \\
&= \begin{pmatrix} x + r_x \\ y + r_y \\ 1 \end{pmatrix}
\end{aligned}
$$

Die Rotations- und Skalierungsmatrizen werden in homogenen Koordinaten um die dritte Dimension und die homogene 1 erweitert, bleiben ansonsten aber unverändert.

$$
\begin{aligned}
R(\alpha) &= \begin{pmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
v' &= R(\alpha) \cdot v \\
\alpha &\in [0, 2\pi)
\end{aligned}
$$

Die Inverse dieser Rotationsmatrix ist, unter der gleichen Argumentation wie oben, ebenfalls die Transponierte: $R(\alpha)^{-1} = R(\alpha)^\top$.

Die Skalierung ist gegeben durch

$$
\begin{aligned}
S(s_x, s_y) &= \begin{pmatrix} s_x & 0 & 0 \\ 0 & s_y & 0 \\ 0 & 0 & 1 \end{pmatrix} \\
v' &= S(s_x, s_y) \cdot v \\
s_x, s_y &> 0
\end{aligned}
$$

Diese Operationen in Form von $3 \times 3-$Matrizen ermöglichen eine einfache Modellierung und Darstellung von zueinander verschobenen Objekten. So kann für jedes Objekt *eine* Matrix gespeichert werden, die alle für das Objekt benötigten Transformationen beinhaltet. Bei weiteren, hinzukommenden Transformationen werden die entsprechenden Matrizen mit einander verrechnet. Die hier ebenfalls erwähnte Skalierung findet Verwendung im Schnittest von Polygonen, ist jedoch nicht relevant für das Darstellen von Stellungen von Polygonen.

2.2 Polygone und Geometrische Algorithmen

Wie bereits erwähnt, werden in dieser Arbeit zur Darstellung von zu platzierenden Formen zweidimensionale Polygone verwendet. Denkbar wären auch andere Darstellungsformen, zum Beispiel eine direkte Verwendung von gegebenen CAD[1] Daten der modellierten Form. Diese Darstellung könnte auch Rundungen und Löcher enthalten, die üblicherweise durch analytische Funktionen gegeben sind. Eine Verarbeitung solcher Daten würde jedoch den Rahmen dieser Arbeit sprengen, und ist nicht Teil der eigentlichen Problemstellung.

Da das Gebiet, in das gegebene Polygone platziert werden sollen, rechteckig ist, erfordert dessen Modellierung nicht viel Mühe. Daher werden im Folgenden ausschließlich Eigenschaften der Polygone untersucht.

Durch die Einschränkung auf einfache Polygone, die im folgenden Unterkapitel genauer erläutert werden, werden gleichzeitig auch bestimmte geometrische Eigenschaften dieser Polygone als analytisches Werkzeug nutzbar. Namentlich sind diese Eigenschaften die Konvexe Hülle, die Triangulierung, das minimale umfassende Rechteck, und die Verdeckung eines Punktes durch ein Polygon. Für die Berechnung dieser Eigenschaften wurden Algorithmen gewählt und implementiert, die eine vertretbare Balance zwischen Implementierungsaufwand und (Laufzeit-)Komplexität aufweisen. Auf die Verwendung von vorhandenen Software Bibliotheken (zum Beispiel CGAL[2]), die solche Verfahren bereits implementieren, wurde bewusst verzichtet, um eine größere Flexibilität der eigenen Implementierung zu bewahren.

[1] CAD, Computer-Aided Design
[2] CGAL, Computational Geometry Algorithms Library, www.cgal.org

2.2.1 Einfache Polygone

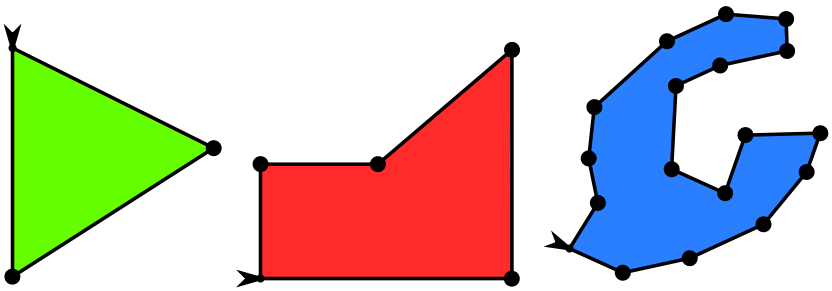

Abbildung 2.3: Einfache Polygone gegeben durch Punkte die links orientierte Linienzüge definieren.

In dieser Arbeit werden einfache, nicht zwingend konvexe Polygone betrachtet. Abbildung 2.3 zeigt einige solche Polygone. Ein Polygon $P_n = (p_1, \ldots, p_n)$, $p_i \in \mathbb{R}^2$ sei also gegeben durch eine Folge von n paarweise verschiedenen Punkten so, dass jeweils zwei Punkte ein Segment (eine Linie) $S_i = (p_i, p_{i+1}), i \in 1, \ldots, n-1$, $S_n = (p_n, p_1)$, bilden, die vom ersten Punkt zum zweiten orientiert ist. Dieser Linienzug ist geschlossen (durch S_n) und keine zwei Segmente schneiden sich , ausser an einem Punkt, der Endpunkt beider Segmente ist. Als Konvention wird davon ausgegangen, dass der Linienzug gegen den Uhrzeigersinn orientiert ist. Das bedeutet, dass links der Orientierung eines Segments stets das Innere des Polygons liegt. Diese Konvention lässt sich, wenn nötig, bei einem gegebenen Linienzug überprüfen und herstellen. Zwingend hierfür ist allerdings, dass eine Information gegeben ist, wo das Innere des Polygons liegt. Da das Polygon nur durch den Linienzug definiert ist, kann es im Inneren des Polygons keine Löcher geben, diese sind für eine allgemeine Betrachtung der Aufgabenstellung dieser Arbeit auch nicht relevant. Rundungen am Polygon können durch entsprechend viele, kurze Segmente dargestellt werden. Dies erhöht jedoch die Komplexität des Polygons und führt zu mehr Rechenaufwand, vor allem bei Brute-Force Ansätzen. Um diesen Effekt abzumildern, ist eine Aufarbeitung von Polygonen denkbar, die Rundungen vereinfacht oder anderweitig in weitere Berechnungen einfließen lässt. Diese Spezialfälle werden hier nicht behandelt, da ein allgemeiner Ansatz verfolgt wird.

Zwei erste Eigenschaften, die ein Polygon dieser Form hat sind:

- Die (per Konvention nach links zeigenden) Normalen der Segmente zeigen in das Innere des Polygons.
- An jedem Punkt gibt es einen inneren Winkel - dies ist derjenige Winkel, der durch beide angrenzenden Segmente aufgespannt wird und im Inneren des Polygons liegt.

Von nun an wird für einen Eckpunkt eines Polygons der Begriff Vertex benutzt, um zu verdeutlichen, dass der Punkt nicht etwa ein beliebiger Punkt am oder im Polygon ist, sondern einer, der als Eckpunkt maßgeblich für das Polygon ist.

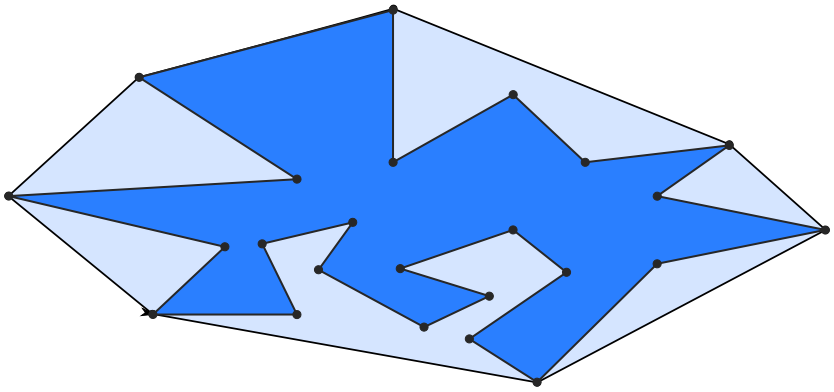

Abbildung 2.4: Konvexe Hülle (grau) eines einfachen Polygons (blau).

2.2.2 Konvexe Hüllen von Polygonen

Die Konvexe Hülle einer Punktmenge M ist definiert als die kleinste konvexe Menge, die M ganz enthält. Eine Menge K ist genau dann konvex, wenn für zwei Punkte $p_1, p_2 \in K$ dieser Menge gilt, dass die Verbindungslinie $L = \{x | x = p_1 + \lambda \cdot (p_2 - p_1),\ \lambda \in [0, 1]\}$ dieser Punkte ebenfalls in K liegt: $L \subset K$. Für unsere Anwendung benötigen wir als Darstellung der Konvexen Hülle diejenigen Segmente, die die konvexe Menge eingrenzend definieren. Abbildung 2.4 zeigt eine solche durch Segmente gegebene Konvexen Hülle eines Polygons. Dies ist also wieder ein Polygon nach unserer Definition, und die definierenden Vertices sind eine Teilmenge der Vertices des zugrundeliegenden Polygons.

Für ein gegebenes Polygon bedeutet dies, vereinfacht ausgedrückt, dass man Segmente von jedem Vertex zu allen anderen ziehen kann, und dann diejenigen Segmente als Konvexe Hülle übernimmt, die am äußersten Rand liegen. Oder noch einfacher ausgedrückt: Um die Vertices des Polygons wird ein enges Gummiband gelegt, das sich eng um die Vertices schmiegt. Die Vertices, die dieses Gummiband berührt, und die dadurch definierten Segmente, sollen berechnet werden.

Zur Berechnung von Konvexen Hüllen von Punktmengen gibt es zahlreiche Algorithmen. Die schnellste Berechnung der Konvexen Hülle von (unstrukturiert gegebenen) n Punkten ist in einer Zeitkomplexität von $\mathcal{O}(n \log h)$ möglich, wobei h die Anzahl der Vertices ist, die auf der Konvexen Hülle liegen[Cha96]. Der Einfachheit halber wurde für diese Arbeit eine Variante von *Andrews monotone chain*[And79] implementiert, um die Konvexe Hülle eines Polygons in $\mathcal{O}(n \log n)$ zu berechnen. Die Berechnung der Konvexen Hülle selbst ist hierbei in $\mathcal{O}(n)$ möglich, nachdem die Eingabe der Punkte mit Aufwand $\mathcal{O}(n \log n)$ sortiert wurde.

Die Punkte der Eingabe seien aufsteigend nach x-Koordinate sortiert, und Punkte mit gleicher x-Koordinate seien absteigend nach y-Koordinate sortiert. Der Algorithmus berechnet die untere und obere Hälfte der Konvexen Hülle separat, im Folgenden wird nur das Vorgehen für die untere Hälfte erläutert; die obere Hälfte wird analog berechnet. Zunächst ist ersichtlich, dass der Punkt p_{min} am weitesten links und p_{max} am weitesten rechts zur Konvexen Hülle gehören. Sind unterhalb der Linie $\overline{p_{min}p_{max}}$ keine weiteren Punkte, ist die untere Hälfte der Konvexen Hülle fertig. Gibt es jedoch Punkte unterhalb der Linie, werden diese in ihrer sortierten Reihenfolge betrachtet. Ein Punkt p_n

kann entweder innerhalb der bisher gefundenen Konvexen Hülle liegen, oder ausserhalb. Liegt er ausserhalb, wird die Konvexe Hülle um diesen Punkt erweitert, und es ist nötig sicherzustellen, dass alle bisherigen Punkte der Konvexen Hülle noch zum Rand dieser gehören. Um das zu prüfen wird vom aktuellsten Segment der Konvexen Hülle rückwärts gehend getestet, welche Segmente der aktuellen Konvexen Hülle durch Hinzufügen von p_n überflüssig werden. Diese werden entfernt, indem die hiermit korrespondierenden Punkte aus der konvexen Hülle entfernt werden. Diese Rückwärtssuche endet, wenn p_{min} erreicht oder ein Segment der Konvexen Hülle gefunden wird, das durch p_n nicht geändert werden muss (weil p_n links von der durch das Segment definierten Gerade liegt). Da jeder Punkt auf diese Weise höchstens ein mal zur Konvexen Hülle hinzugefügt und ggf. wieder entfernt werden kann, sind nach $\mathcal{O}(n)$ Schritten alle an der Konvexen Hülle beteiligten Punkte gefunden. Abbildung 2.5 skizziert die beschriebene Konstruktion der Konvexen Hülle.

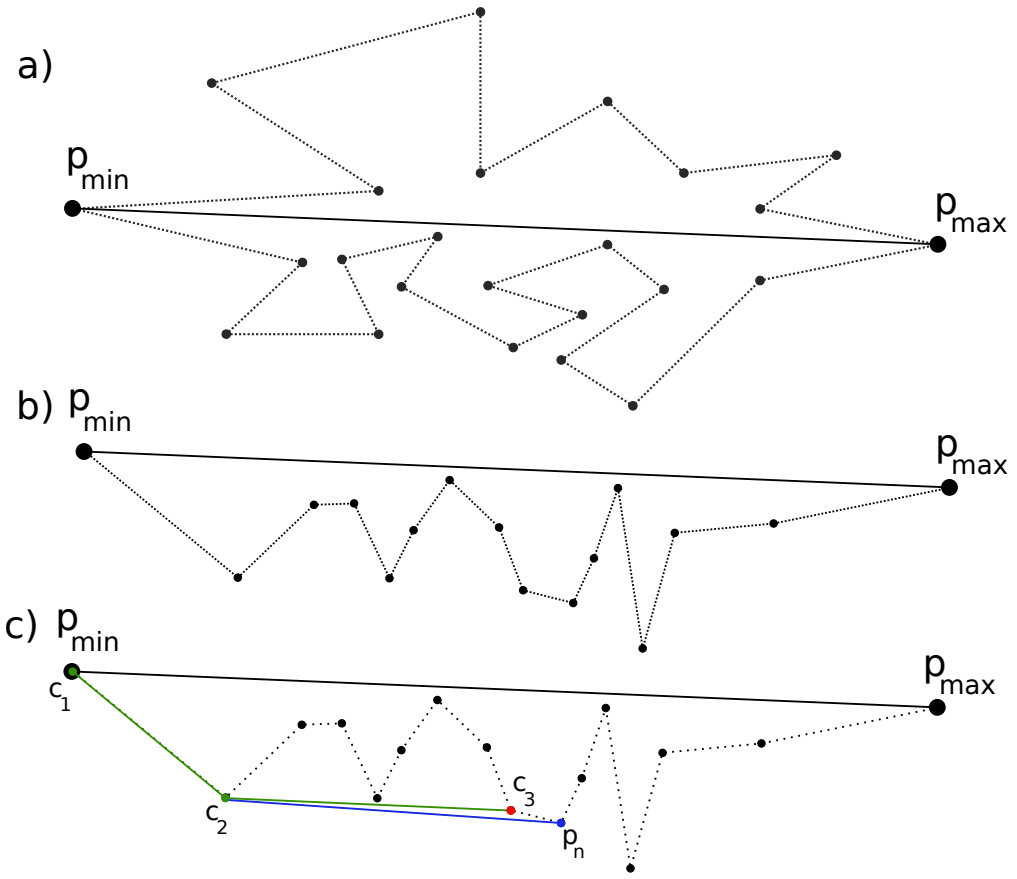

Abbildung 2.5: a) p_{min} und p_{max} des Polygons sind auf jeden Fall Teil der Konvexen Hülle. **b)** Einteilen der Punkte in untere und obere Hälfte (nur untere Hälfte dargestellt) und Sortieren der Punkte anhand der x-Koordinaten. **c)** Die Punkte c_1, c_2 und c_3 sind Teil der Konvexen Hülle aller bis c_3 betrachteten Punkte. Durch Hinzunahme von p_n wird die Konvexe Hülle um p_n erweitert und c_3 wird entfernt.

Nach Ablauf dieses Algorithmus liegt die Konvexe Hülle als Folge von Vertices vor, also als Linienzug, der gegen den Uhrzeigersinn verläuft.

Um eine Konstellation von zwei Polygonen zu bewerten, muss später die Konvexe Hülle von *zwei Polygonen* P_n und P_m (mit je n und m Vertices) berechnet werden. Dies direkt über die Vertices der Polygone zu berechnen, würde wieder eine erneute Sortierung der Punkte benötigen, was eine teure Operation ist. Da aber zu beiden gegebenen Polygonen jeweils eine Konvexe Hülle berechnet wurde, lässt sich eine Sortierung der für uns relevanten Vertices, und daher die Berechnung der gesamten Konvexen Hülle, in $\mathcal{O}(n+m)$ realisieren. Dies ist der Fall, da die gemeinsame Konvexe Hülle beider Polygone nur aus Vertices bestehen kann, die bereits zu der Konvexen Hülle jedes einzelnen Polygons gehören. Da die Konvexen Hüllen als Linienzug gegeben und die Vertices jeweils sortiert sind, lassen sich diese durch einen Merge-Sort Schritt in eine gemeinsame Sortierung überführen. Dies hat einen Aufwand von $\mathcal{O}(n+m)$. Im Anschluss wird wieder der bereits beschriebene Algorithmus verwendet, um die Konvexe Hülle der sortierten Punktmenge zu berechnen. Dies hat ebenfalls einen Aufwand von $\mathcal{O}(n+m)$. Abbildung 2.6 zeigt die Konvexe Hülle zweier konvexer Polygone.

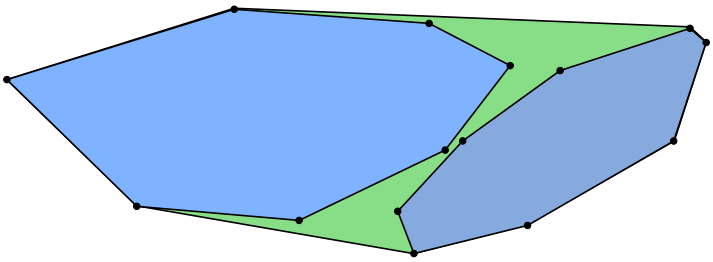

Abbildung 2.6: Konvexe Hülle (in grün hinterlegt) zweier konvexer Polygone. Weil die Linienzüge der beiden Polygone bereits einer Sortierung folgen, lässt sich die gesamte Konvexe Hülle in $\mathcal{O}(n)$ berechnen.

2.2.3 Minimales umfassendes Rechteck

Das Minimale Umfassende Rechteck (kurz: minimales Rechteck) einer Punktmenge M mit n Punkten ist das Rechteck mit der kleinsten Fläche, das alle Punkte aus M enthält. Die Orientierung des minimalen Rechtecks kann beliebig sein, und ist nicht an die Hauptachsen gebunden. Das minimale Rechteck einer Punktmenge enthält auch die Konvexe Hülle dieser Punktmenge, wie in Abbildung 2.7 dargestellt.

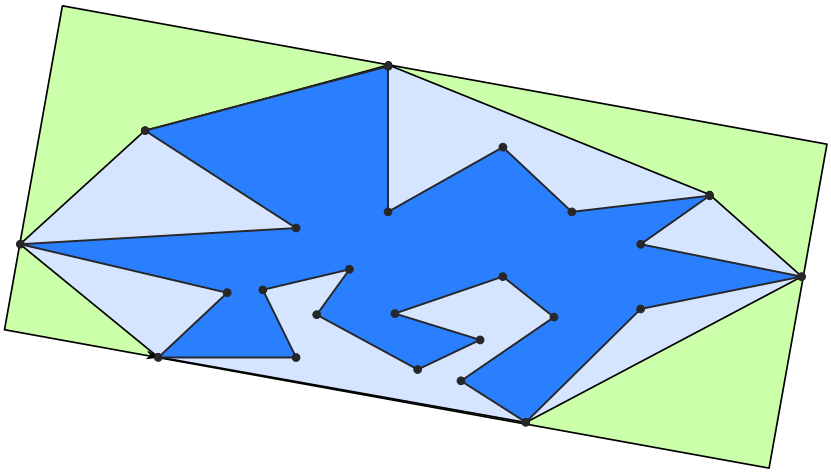

Abbildung 2.7: Das minimale umfassende Rechteck eines Polygons wird anhand der Konvexen Hülle des Polygons berechnet.

Insbesondere ist die Berechnung des minimalen Rechtecks in einer Zeitkomplexität von $\mathcal{O}(n)$ möglich, wenn die Konvexe Hülle der entsprechenden Punktmenge gegeben ist[FS75][Tou83]. Dies ist der Beobachtung geschuldet, dass das minimale Rechteck mit jeder Seite mindestens einen Vertex der Konvexen Hülle der Punktmenge berühren muss, und mit mindestens einer Seite an einem Segment der Konvexen Hülle anliegt[FS75]. Ist also eine Konvexe Hülle (bzw. ein konvexes Polygon) gegeben, lässt sich sehr schnell und relativ einfach das minimale Rechteck berechnen. Es wird wieder die Konvention verwendet, dass eine Konvexe Hülle durch einen Linienzug gegeben ist.

Zur Berechnung des minimalen Rechtecks wird die Methode der *Rotating Callipers*[Tou83] verwendet. Zwei jeweils parallele Linienpaare, die gemeinsam ein Rechteck bilden, werden hierbei um das konvexe Polygon rotiert. Als Startkonfiguration können an den Hauptachsen orientierte Linien gewählt werden, die jeweils an dem Vertex anliegen, der am weitesten unten, rechts, oben und links ist. Die Linien liegen stets an mindestens einem Vertex oder an einem Segment an, und werden stets um den kleinsten möglichen Winkel so weiter rotiert, dass keine Linie ein Segment des konvexen Polygons schneidet, sondern höchstens an einem Segment ausgerichtet ist. Hierbei muss jede Linie jeden Vertex genau ein mal berühren, bis die Ausgangsstellung wieder erreicht ist. Dies entspricht einer vollen Umrundung des konvexen Polygons durch alle Linien. Während dieser Umrundung lassen sich alle durch die vier Linien gebildeten Rechtecke aufstellen (es sind höchstens n Rechtecke), und das kleinste wird als Ergebnis des Algorithmus ausgegeben. Tatsächlich ist nur eine viertel-Umrundung des Polygons nötig, bis alle relevanten Rechtecke betrachtet wurden, da alle weiteren Drehungen nur

Rechtecke erzeugen, die äquivalent zu bereits bekannten sind. Abbildung 2.8 zeigt zwei Schritte des beschriebenen Algorithmus.

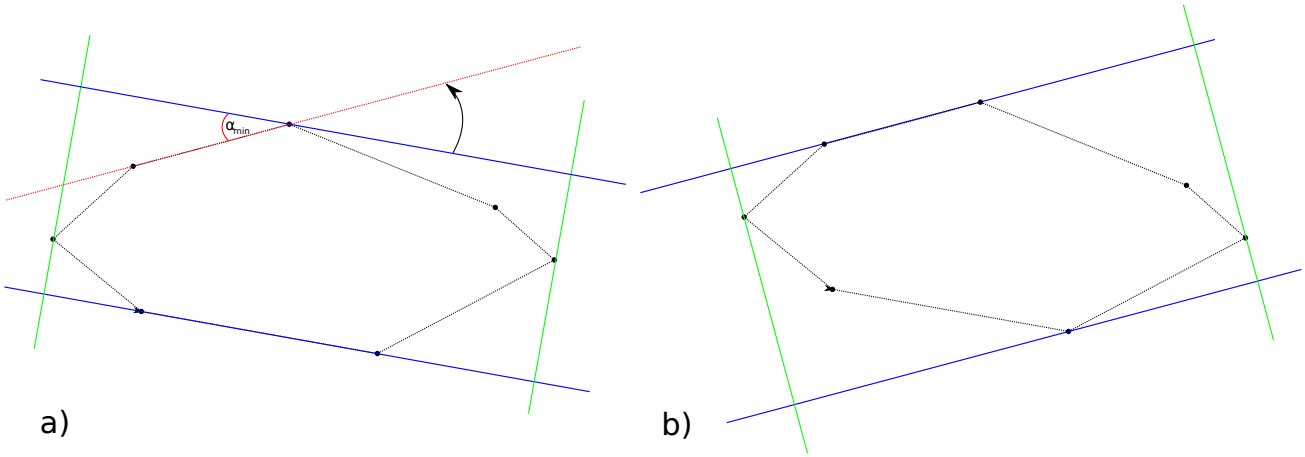

Abbildung 2.8: Die im Rechteck angeordneten Parallelen rotieren um den minimalen Winkel α um das konvexe Polygon.

Ein berechnetes minimales Rechteck zu einem Polygon wird mit den Seitenlängen (x, y) und zwei Vektoren $p, d \in \mathbb{R}^2$ gespeichert. Der Vektor p gibt einen Eckpunkt des Rechtecks an und der Vektor d eine Richtung, in die von p aus die längere Seite des Rechtecks zeigt. Der Einfachheit halber wird p so gewählt, dass es der Eckpunkt links unten ist, nachdem das Rechteck anhand der Richtung p durch eine Rotation $R(p)^{-1}$ auf die Hauptachsen ausgerichtet wurde. Die Seitenlänge x ist hierdurch die längere der beiden Seiten und kann als Skalierungsfaktor verwendet werden, wie es im Schnitttest nötig wird.

2.2.4 Triangulierung von Polygonen

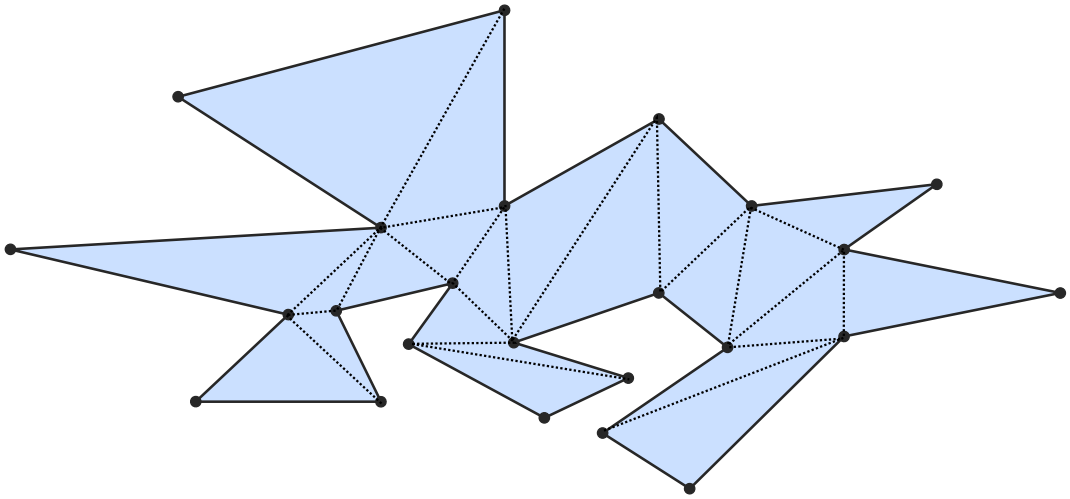

Abbildung 2.9: Triangulierung eines Polygons.

Die Zerlegung eines Polygons P_n in Dreiecke wird Triangulierung genannt und ist in Abbildung 2.9 gezeigt. Jeweils drei Vertices des Polygons definieren hierbei ein Dreieck, Vertices können an mehreren Dreiecken beteiligt sein, und es gibt genau $n-2$ solcher Dreiecke. Die hier betrachtete Form einfacher Polygone ermöglicht eine unkomplizierte Zerlegung in Dreiecke, da keine Löcher im Inneren, oder gar sich schneidende Segmente berücksichtigt werden müssen. Da die Triangulierung nur für Eingabe-Polygone durchgeführt werden muss, und dies auch nur ein mal, wird ein Algorithmus in einer Zeitkomplexität von $\mathcal{O}(n^2)$ benutzt. Die Grundidee des verwendeten Algorithmus ist, fehlende Segmente zwischen Vertices hinzuzufügen und entweder so entstandene Dreiecke am Polygon P_n wegzuschneiden, oder das Polygon am eingefügten Segment aufzuspalten. Dies entspricht einerseits einem *Ear-Clipping*[Mei75] Ansatz, verwendet aber auch eine rekursive Strategie an Stellen des Polygons, an denen nicht direkt ein Dreieck entfernt werden kann.

Da genau drei Vertices an einem Dreieck beteiligt sein müssen, läuft der Algorithmus den Linienzug des Polygons entlang, und betrachtet jeweils drei aufeinander folgende Vertices v_0, v_1, v_2. Bilden diese Vertices ein gültiges, am Polygon abstehendes Dreieck, kann dieses Dreieck abgeschnitten werden. Die beteiligten Vertices werden vermerkt und der Algorithmus fährt fort mit einem Polygon P_{n-1}, in dem der Vertex v_1 fehlt, da er abgeschnitten wurde. Ein Dreieck kann nur abgeschnitten werden, wenn innerhalb dieses Dreiecks keine weiteren Punkte des Polygons liegen. Liegt allerdings mindestens ein weiterer Punkt im Dreieck, wird derjenige Punkt v ausfindig gemacht, der den kürzesten Abstand zu v_1 hat. Entlang der Linie $\overline{v_1 v}$ wird das Polygon P_n nun in zwei kleinere Polygone geteilt, und das Segment (v_1, v) wird in beide Polygone hinzugefügt. Der Algorithmus ruft sich nun rekursiv mit den erzeugten Teilpolygonen als Eingabe auf und fügt die Ergebnisse der Rekursionen der Dreiecksliste hinzu.

In jedem Schritt wird also entweder ein Vertex weggeschnitten oder das Polygon geteilt. Dies ist in Abbildung 2.10 dargestellt. Da ein neues Teilpolygon mindestens drei Vertices hat, sind solche

erzeugten Teilpolygone gültig. Der Algorithmus (und rekursive Aufrufe davon) terminiert, wenn er ein Polygon P_3 erzeugt, oder Teillösungen aus einer Rekursion empfangen hat.

Angenommen für ein Polygon P_n geht der Algorithmus nie in eine Rekursion, so muss er maximal alle n Vertices betrachten, bis ein erster Vertex mit dem ersten dazugehörigen Dreieck abgeschnitten werden kann. Dies ist garantiert durch *Meisters Two Ears Theorem*[Mei75], das besagt, dass (bis auf Dreiecke) jedes einfache Polygon mindestens zwei abstehende Dreiecke besitzt.

Für das Finden eines abstehenden Dreiecks benötigt der Algorithmus also maximal n Schritte, hinzu kommen pro Dreieck $n-3$ Tests, ob weitere Punkte in dem gefundenen Dreieck liegen. Ohne Rekursion benötigt der Algorithmus also $\mathcal{O}(n^2)$ Schritte, um alle Dreiecke, bzw. Vertices abzuschneiden. Wird eine Rekursion mit Polygonen P_{n_1}, P_{n_2} nötig, werden insgesamt

$$\begin{aligned} n &= n_1 + n_2 + 2 \\ n^2 &= n_1^2 + n_2^2 + 2n_1n_2 + 2n_1 + 2n_2 + 4 \end{aligned}$$

n Vertices bei Berechnungen betrachtet. Somit sind

$$\mathcal{O}(n_1^2) + \mathcal{O}(n_2^2) + \mathcal{O}(n_1) + \mathcal{O}(n_2) + \mathcal{O}(1) = \mathcal{O}(n_1^2 + n_2^2) \in \mathcal{O}(n^2)$$

Schritte nötig, bis die Rekursionen terminieren.

Ist das Eingabepolygon also ein gültiges einfaches Polygon, terminiert der Algorithmus nach $\mathcal{O}(n^2)$ Schritten.

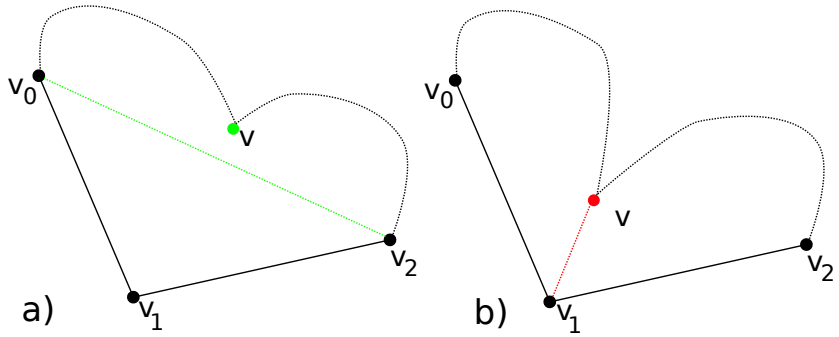

Abbildung 2.10: a) Das Dreieck kann vom Polygon abgeschnitten werden. **b)** v liegt innerhalb des Dreiecks, daher wird das Polygon entlang $\overline{v_1 v}$ getrennt und die zwei Teilpolygone einzeln trianguliert. Die abgerundeten Segmente stellen einen Verlauf des restlichen Linienzugs dar, in dem v enthalten ist.

Nach dem Algorithmus liegt eine Liste von Dreiecken vor, die angibt durch welche Vertices ein Dreieck gebildet wird, sodass alle angegebenen Dreiecke das ursprüngliche Polygon bilden. Die Triangulierung wird für den grafischen Schnittest benötigt, liefert aber auch den Flächeninhalt des Polygons.

2.2.5 Punktverdeckung durch Polygone

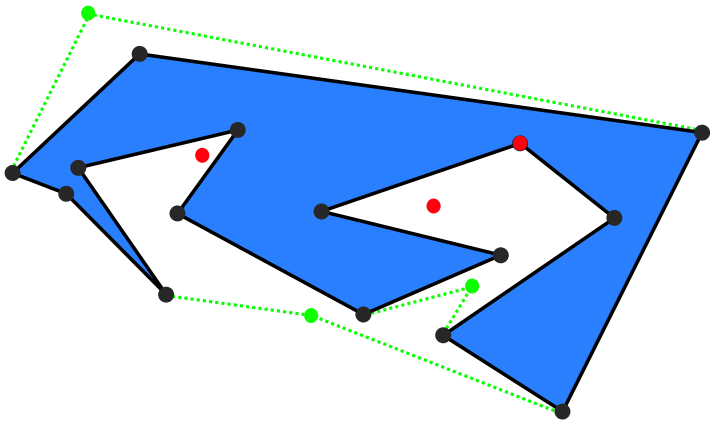

Abbildung 2.11: Rote Punkte ”sehen” in alle Richtungen nur das Polygon, grüne Punkte ”sehen” das Polygon nur in einem bestimmten Bereich ihres Horizonts.

Gegeben seien ein beliebiger Punkt $v \in \mathbb{R}^2$ und ein Polygon P_n. Der Punkt v sei nicht im Inneren des Polygons positioniert, er kann jedoch entweder komplett ausserhalb des Polygons liegen, oder in einer lokal konkaven Stelle. Weiterhin sei ein Kreis gegeben, der groß genug ist, sodass sowohl v, als auch P_n in diesem Kreis liegen. Dieser Kreis ist sozusagen der Horizont von v und P_n.

Für v und P_n ist nun zu entscheiden, wieviel vom Horizont v erblicken kann, sozusagen welchen Winkel das Blickfeld von v auf den Horizont hat. Liegt der Punkt in einer lokal konkaven Stelle des Polygons, gibt es zwei Möglichkeiten. Entweder der Punkt sieht in alle Richtungen nur Segmente des Polygons, in diesem Fall wird der Horizont in einem Winkel von 0 gesehen. Oder der Punkt kann auf den Horizont blicken, in diesem Fall gibt es einen Winkel $\alpha \in (0, 2\pi)$, der durch zwei Richtungsvektoren aufgespannt wird, die das Blickfeld von v angeben. Abbildung 2.11 zeigt Beispiele, in denen Punkte vom Polygon verdeckt sind oder nach außen ”sehen” können.

Dies ist gleichzeitig auch ein Test darauf, ob der gegebene Punkt von außen gesehen durch das Polygon verdeckt wird, oder zu sehen ist. Dieser Test wird im Laufe der Arbeit benutzt, um von aussen nicht mehr erreichbare Punkte in einem Polygon, oder in einer Polygonkonstellation, zu erkennen. Für den folgenden Algorithmus konnte der Autor keine Beschreibung in der Literatur finden, die ihm zugänglich war. Es wird aber davon ausgegangen, dass der Algorithmus unter anderen Stichwörtern, oder für eine andere Verwendung bereits bekannt ist.

Der Algorithmus zur Lösung der Problemstellung berechnet nicht direkt den Blickwinkel von v auf den Horizont. Statt dessen wird der komplementäre Winkel berechnet, der das Blickfeld von v *auf das Polygon* angibt. Zur Vereinfachung wird dieser Blickwinkel auf das Polygon im Folgenden (ebenfalls) α genannt.

Um an diesen Winkel α zu gelangen, werden zwei Richtungsvektoren $r_{cw}, r_{ccw} \in \mathbb{R}^2$ gesucht, die das Blickfeld von v angeben. Gleichzeitig wird im Verlauf des Algorithmus die Information mitgetragen, ob $\alpha \leq \pi$, oder ob $\alpha > \pi$ gilt, um aus den Richtungen eindeutig den Winkel bestimmen zu können. Die Grundidee besteht darin, alle Segmente des gegebenen Polygons P_n auf den Horizont von v zu

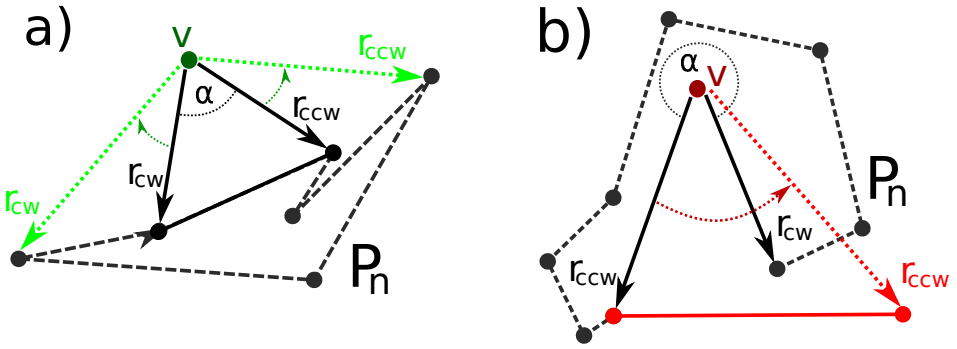

Abbildung 2.12: a) Die in P_n durchlaufenen Segmente erweitern das initiale Blickfeld von v ohne Überschneidungen zu erzeugen. **b)** Das aktuell betrachtete Segment aktualisiert r_{ccw} und erzeugt dadurch ein Blickfeld für v von mehr als 2π. Hierdurch wird erkannt, dass v komplett von P_n umschlossen wird.

projezieren, und Buch zu führen, zu welchem Teil der Horizont bereits durch das Polygon verdeckt ist. Abbildung 2.12 skizziert dieses Vorgehen.

Initial wird das erste Segment $S_1 = (p_1, p_2)$ von P_n auf den Horizont projiziert, indem die Richtungen $(p_1 - v)$ und $(p_2 - v)$ als Startwerte für r_{cw} und r_{ccw} gewählt werden. Gleichzeitig wird notiert, wo bezüglich der beiden Richtungen das Innere des Polygons liegt. Dies ist initial durch das erste Segment des Polygons gegeben. Sei nun S_i das im Linienzug nächste Segment. Von diesem nächsten Segment wird der nächste, noch nicht verarbeitete Punkt p_{i+1} betrachtet. Liegt p_{i+1} *innerhalb* des durch r_{cw} und r_{ccw} markierten inneren Bereiches, muss keiner der beiden Richtungsvektoren angepasst werden. Liegt der nächste Punkt *ausserhalb* dieses inneren Bereiches, wird eine der Richtungen so angepasst, dass der Punkt wieder innen liegt (liegt ein Punkt genau auf einer der durch die Richtungen gegebenen Geraden, liegt dieser Punkt ebenfalls *innen*). Dies passiert so, dass abhängig von der Laufrichtung des nächsten Segments entweder r_{cw} oder r_{ccw} angepasst wird, je nachdem, ob der nächste aussen liegende Punkt den eingegrenzten inneren Bereich in Richtung *im (CW, r_{cw})* oder *gegen (CCW, r_{ccw})* den Uhrzeigersinn verlässt. Weil das Polygon sich in besagte Richtung erstreckt, wird die entsprechende begrenzende Richtung zu $r = (p_{i+1} - v)$ aktualisiert. Auf diese Weise werden alle Segmente des Polygons betrachtet, implizit auf den Horizont des Punktes v projiziert und dadurch das freie Blickfeld von v eingegrenzt.

Im Laufe des Algorithmus kann es passieren, dass eine der Richtungen r so aktualisiert werden soll, dass sie die jeweils andere Richtung übertritt. Dies bedeutet, dass aus Sicht von v zwei Segmente des Polygons sich am Horizont aus verschiedenen Richtungen überlappen, und zwar an einer Stelle des Horizonts, die bisher sichtbar war. Da das Polygon ein Linienzug ist, ist v hierdurch komplett vom Polygon umgeben. Der Horizont ist also nicht sichtbar, der gesuchte Winkel α ist gleich null und der Algorithmus terminiert mit diesem Ergebnis.

Läuft der Algorithmus alle Segmente durch, ohne auf so eine Überkreuzung von r_{cw} und r_{ccw} zu treffen, ist der gesuchte Winkel größer null, und die resultierenden Richtungsvektoren und der Winkel werden als Ergebnis ausgegeben. Die Laufzeitkomplexität dieses Vorgehens beträgt $\mathcal{O}(n)$, da maximal n Segmente betrachtet werden müssen, bis der Algorithmus ein Ergebnis ausgibt.

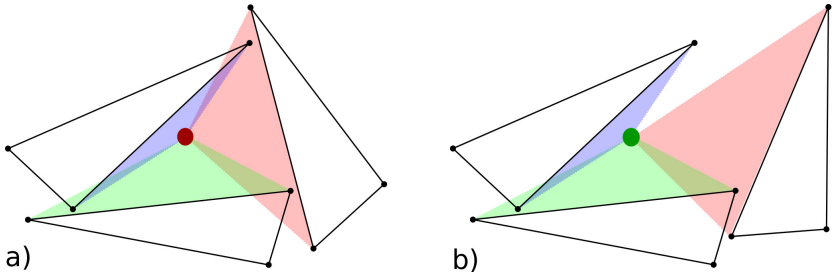

Abbildung 2.13: a) Der Punkt ist von Polygonen so umgeben, dass sein Horizont ausgefüllt ist. **b)** Die Polygone verdecken den Horizont des Punktes nicht vollständig.

Ist ein Punkt v gegeben, und mehrere Polygone $P^i_{n_i}$, kann für diesen Punkt berechnet werden, ob sein Horizont von allen Polygonen gleichzeitig verdeckt wird (siehe Abbildung 2.13). Hierfür werden zunächst für jedes Polygon die Richtungen r^i_{cw} und r^i_{ccw} samt der Winkel α^i berechnet. Diese Richtungen werden nun so interpretiert, dass sie den Teil des Horizonts als Intervall angeben, in dem das Polygon P^i den Horizont verdeckt. Eine Liste von Intervallen auf dem Horizont wird zunächst aufgestellt und bezüglich einer Referenz-Richtung (z.B. bezüglich $\begin{pmatrix}1 & 0\end{pmatrix}^\top$ und entgegen dem Uhrzeigersinn) sortiert. Die Intervalle können anschließend in einem Durchlauf der sortierten Liste mit einander verschmolzen werden. Intervalle, die sich überlappen werden hierbei zu einem Intervall zusammen gefasst. Dieses Intervalle-Mergen ist in einem Durchlauf der sortierten Intervall-Liste möglich, also in $\mathcal{O}(n)$, wenn n die Anzahl der Intervalle ist.

Während dieses Merge-Schrittes kann passieren, dass nur ein Intervall übrig bleibt, das sich auch noch selber überlappt, weil es auf dem Horizont einen Winkel von mehr als 2π einnimmt. In diesem Fall ist der Punkt v von allen Polygonen so umgeben, dass kein Horizont über bleibt. Anderenfalls bleibt eine Liste von Intervallen, zwischen denen zu einem gewissen Bereich jeweils ein Stück Horizont sichtbar ist. Mithilfe dieses Vorgehens kann untersucht werden, wie erreichbar ein Punkt v in Konstellation mit einem oder vielen Polygonen ist.

3 Polygon-Schnitttests auf der Grafikkarte

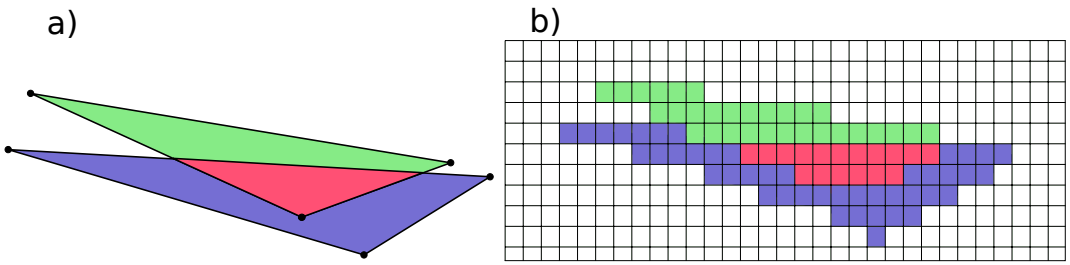

Abbildung 3.1: a) Zwei Polygone in einer Stellung, in der sie sich schneiden. Die Schnittfläche ist rot markiert. **b)** Die Polygone werden anhand eines Rastergitters dargestellt.

Ein wesentlicher Bestandteil der Problemstellung dieser Arbeit ist es, dass platzierte Formen nicht übereinander liegen dürfen. Um dies sicher zu stellen, müssen in einer Stellung gegebene Polygone darauf getestet werden, ob diese sich schneiden. Hierfür existieren analytische Methoden, die für zwei Polygone die beteiligten Segmente darauf testen, ob diese sich schneiden. Dies ist mit einer optimalen Komplexität von $\mathcal{O}(k + n \log n)$ möglich[CE92], wobei n die Anzahl aller gegebenen Segmente ist, und k die Anzahl der resultierenden Schnittpunkte. Einerseits ist die Implementierung solcher Algorithmen fehleranfällig und daher zeitraubend, andererseits sind analytisch genaue Schnitttests auch nicht immer nötig.

In dieser Arbeit wird daher ein Schnitttest verwendet, der auf der Rasterisierung von Polygon-Segmenten beruht. Dies bedeutet, dass ein Polygon in ein zweidimensionales Gitter gelegt wird, und anhand des Linienzugs und des Inneren des Polygons Gitterzellen markiert werden, in denen das Polygon liegt[OF93]. Der Vorgang der Abbildung von geometrischen Formen auf Gitterzellen wird Rasterisierung genannt. Ein Beispiel einer Rasterisierung von Polygonen ist in Abbildung 3.1 zu sehen. Ein solches Verfahren ist einfacher umzusetzen als ein analytisches und es bietet gleichzeitig Kontrolle über einen gewissen Fehler, den man im Schnitttest erlauben kann.

Grafikkarten (GPUs[1]) sind hoch spezialisierte Hardwareeinheiten, die auf schnelle Rasterisierung und Verwaltung einfacher geometrischer Objekte spezialisiert sind. Eine Rasterisierung von gegeben Polygonen kann zwar auch über eine Implementierung eines Software-Rasterisierers erfolgen, oder durch die Benutzung von vorhandenen Software-Bibliotheken, jedoch stellen diese nur suboptimale Lösungen dar. In dieser Arbeit wird daher der Ansatz verfolgt, eine Rasterisierung der Polygone unmittelbar auf der Grafikkarte geschehen zu lassen, und auf diese Weise einen Schnitttest zu

[1]GPU, Graphics processing unit

realisieren, der sich die Spezialisierung der Grafikhardware zunutze macht. Hierfür wird der *OpenGL-Standard*[2] benutzt, der eine weitestgehend direkte Kommunikation mit der Grafikkarte erlaubt. Der Schnitttest erfolgt dann anhand durch die Grafikkarte erstellter Bilder der Polygone, und dies alles geschieht *auf dem Speicher und den Recheneinheiten der GPU*. Weitere Funktionalitäten von OpenGL ermöglichen anschliessend das Ergebnis eines solchen Schnitttests im Hauptspeicher der Anwendung schnell verfügbar zu machen.

Im Folgenden wird die *OpenGL-API*[3] und deren Verwendung erläutert. Darauf aufbauend wird, unter Nutzung aktueller technischer Möglichkeiten der *OpenGL-Version 4.2*, ein Schnitttest auf der GPU entwickelt und optimiert. Über die bereits bestehende OpenGL Anbindung wird zusätzlich eine einfache Visualisierung von Polygonen und Polygon-Schachtelungen realisiert, sowie eine Maus basierte Eingabe von Polygonen durch den Benutzer.

3.1 Grundlagen in OpenGL

Grafikkarten sind darauf spezialisiert Berechnungen auf einfachen geometrischen Objekten schnell und hochparallel durchzuführen und als Rastergrafik auszugeben. OpenGL ist ein durch die Khronos Group verwalteter Standard zur Darstellung von 2D und 3D Grafik. Aus Programmierersicht ist *OpenGL* eine Bibliothek und Programmierschnittstelle die es erlaubt bestimmte geometrische Primitive und dazugehörige Daten auf den Speicher der Grafikkarte zu übertragen, diese durch die GPU verarbeiten zu lassen, und daraus resultierende Bilder anzuzeigen oder anderweitig weiter zu verwenden. Hierfür definiert OpenGL eine Reihe von Funktionen, die einem Programm ermöglichen, mit einer gegebenen OpenGL-Implementierung zu interagieren um definierte OpenGL-Zustände zu verändern, Speicher anzufordern und Daten auf die GPU zu übertragen, und letztendlich Objekte durch die GPU erzeugen (rendern) zu lassen. Die OpenGL-Implementierung wird meist durch den Grafikkarten Treiber bereitgestellt und dient als Server, der verschiedenen Anwendungen, den Clients, OpenGL-Funktionalität zur Verfügung stellt und Ressourcen der Grafikkarte verwaltet. OpenGL ist auf kein bestimmtes Betriebssystem beschränkt und namhafte Hersteller von Grafikkarten bieten über entsprechende Treiber OpenGL-Unterstützung für gängige Betriebssysteme. Eine OpenGL-Implementierung bezieht sich stets auf eine bestimmte Version des OpenGL-Standards. Neuere Versionen des Standards zeichnen sich durch mehr verfügbare und meist technisch aktuelle Funktionalität aus. In dieser Arbeit wird Funktionalität der *OpenGL-Version 4.2* verwendet, hierauf wird im nächsten Kapitel genauer eingegangen.

Für ein Programm lässt sich die Verwendung von OpenGL in fünf Schritte einteilen:

1. Initialisiere OpenGL Kontext.
2. Fordere von OpenGL Speicher an und fülle diesen mit geometrischen Objekten und dazu gehörenden Daten. Initialisiere nötige Shader-Programme und übergib diese an OpenGL.

[2] OpenGL, Open Graphics Library, http://www.opengl.org/
[3] API, Application programming interface

3. Setze aktuell zu zeichnendes Objekt, nötige Einstellungen und Zustände, aktiviere aktuell zu nutzendes Shader-Programm.
4. Rendere geometrisches Objekt über *Draw Call*.
5. Gib Speicher frei und beende OpenGL Kontext.

Während Schritt 1, 2 und 5 häufig nur ein mal nötig sind, werden Schritt 3 und 4 stets dann ausgeführt, wenn eine neue Ausgabe aktualisierter Bilddaten gewünscht ist. Für interaktive Echtzeitanwendungen ist zum Beispiel eine Bildwiederholrate von mindestens 30 Bilern pro Sekunde wünschenswert, sodass mindestens 30 mal pro Sekunde nötige Datenstrukturen aktualisiert werden und ein *Draw Call* für zu zeichnende Objekte erfolgen muss.

Die Initialisierung eines OpenGL Kontexts erfolgt über Helfer-Bibliotheken, die von dem Betriebssystem nötige Ressourcen anfordern und technisch umfangreiche Funktionen abstrahieren, wie etwa die Funktionalität einer bestimmten OpenGL-Version zugänglich zu machen. Hierfür wurden in dieser Arbeit die Bibliotheken GLFW[4] und GLEW[5] genutzt. OpenGL kann nur einige wenige geometrische Primitive zeichnen: Punkte, Linien und Dreiecke. Um also komplexe Objekte wie Polygone zeichnen zu können, müssen diese als Dreiecke vorliegen. Um diese Daten an OpenGL zu übergeben und effizient verwalten zu können gibt es viele Mechanismen, auf die hier nicht näher eingegangen wird. Grundsätzlich teilt ein Programm OpenGL mit, welche Art von Daten es zu zeichnen beabsichtigt. Auf Anfrage stellt OpenGL reservierten Speicher bereit, auf den Vertices, Farben, Texturen und vieles mehr übertragen werden kann. Anschliessend werden diese Daten über OpenGL (auf der GPU) in sogenannten *Shader-Programmen* zu einem Bild verarbeitet.

Shader sind eine Arbeitsanweisung an die Grafikkarte und werden als Programme realisiert die auf speziellen, parallelen Recheneinheiten der Grafikkarte ausgeführt werden, und auf Eingabedaten in einem definierten Rahmen arbeiten. Diese Verarbeitungsweise von Daten entspricht dem Single-Instruction-Multiplie-Data Prinzip. Shader werden in der C ähnlichen Programmiersprache GLSL[6] geschrieben, an OpenGL zur Kompilierung übergeben und anschliessend als zu benutzende Shader-Programme registriert. OpenGL definiert verschiedene Arten von Shadern, die auf verschiedenen Ebenen in OpenGL unterschiedliche Aufgaben übernehmen. Solche verschiedenen Shader werden gemeinsam zu einem Shader-Programm zusammengefasst, das für OpenGL vereinfacht als 'Arbeitsanweisung an die Grafikkarte' umschrieben werden kann. Während also ein Client-Programm von CPU-Seite für OpenGL Daten vorbereitet und nötige Einstellungen trifft, verarbeiten Shader-Programme diese Daten *auf der Grafikkarte* und führen zum fertigen Bild. Der in Schritt 4 nötige *Draw Call* bewirkt, dass zu malende Daten in Form von Vertices unter dem aktuell aktiven Shader-Programm verarbeitet werden. Diese Verarbeitung erfolgt auf der GPU und durchläuft die sogenannte *OpenGL Rendering-Pipeline*.

Die in Abbildung 3.2 gezeigte Pipeline ist eine extrem verkürzte Darstellung der tatsächlichen OpenGL Pipeline. Im weiteren Verlauf wird nur auf Aspekte der Pipeline und der Shader eingegangen, die für diese Arbeit wichtig sind und in der Implementierung eingesetzt wurden. Abbildung 3.2 zeigt also nur für diese Arbeit relevante Teile der Rendering-Pipeline: die programmierbaren Vertex- und

[4] http://www.glfw.org/
[5] http://glew.sourceforge.net/
[6] GLSL, OpenGL Shading Language

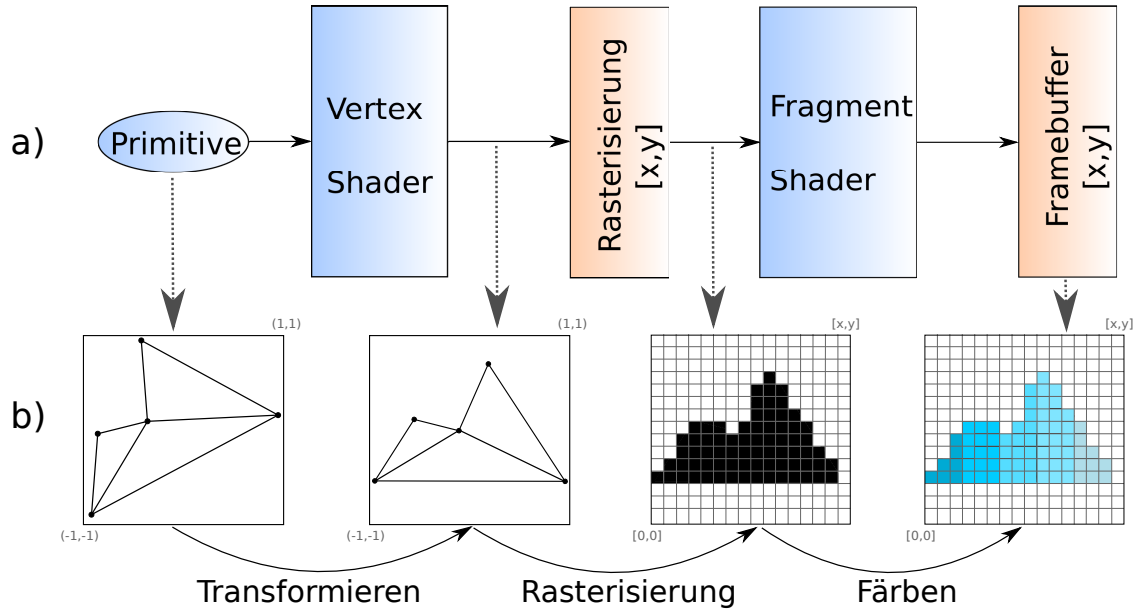

Abbildung 3.2: a) Vereinfachte OpenGL Rendering-Pipeline. Blau hinterlegte Schritte unterliegen der Kontrolle durch den Programmierer. Orange hinterlegte Schritte sind unter Kontrolle von OpenGL und können nur bedingt konfiguriert werden. **b)** Skizzenhafte Darstellung der Schritte in der OpenGL Rendering-Pipeline. Die Schritte *Transformieren* und *Färben* sind durch den Vertex- und Fragment-Shader weitestgehend frei programmierbar. *Quelle: Eigene Darstellung in Anlehnung an die OpenGL 4.4 Spezifikation[SA13].*

Fragment-Shader. Die Rasterisierung erfolgt automatisiert durch OpenGL und bildet geometrische Primitive, deren Vertices durch den Vertex-Shader bearbeitet wurden, auf ein Rasterbild ab. Aus der Rasterisierung gehen sogenannte Fragmente hervor, dies sind unfertige Pixel des Bildes, die das rasterisierte Objekt darstellen. Fragmente werden im Fragment-Shader mit einer Farbe versehen und, falls kein anderer Mechanismus das Gegenteil bewirkt, in das resultierende Bild übernommen. Alle Shader können, neben durch die Pipeline definierten Eingabewerten, auch von der CPU bestimmte, für den aktuellen Aufruf relevante Eingaben erhalten. Dies sind sogenannte Uniform Variablen. Diese Uniform Variablen erlauben es, einen einmal geschrieben und geladenen Shader mit für verschiedene Objekte wechselnden Parametern auszuführen. Auf diese Weise können zum Beispiel aktuell nötige Transformationsmatrizen in den Vertex-Shader übergeben werden.

Der Vertex-Shader wird für jeden Vertex des zu zeichnenden Primitivs einmal aufgerufen. Dies erlaubt es, alle Vertices in parallelen Recheneinheiten gleichzeitig zu verarbeiten. Diese Recheneinheiten können nicht gegenseitig auf Informationen und Zwischenergebnisse voneinander zugreifen. Ein Vertex kann im Vertex-Shader verschiedenen Transformationen unterzogen werden, und auch für den Vertex wichtige Attribute können manipuliert werden. Ein Vertex kann, neben seiner Position als Vektor in maximal $\mathbb{R}^4$, noch weitere, frei wählbare Eingabeparameter mitbringen, wie etwa die eigene Farbe, die lokale Normale des Objekts, oder Texturkoordinaten. Nach der Verarbeitung im Vertex-Shader wird aber erwartet, dass die Koordinaten aller Vertices in $\mathbb{R}^3$ in homogenen Koordinaten, also als Vektor

$\begin{pmatrix} x & y & z & w \end{pmatrix}^\top \in \mathbb{R}^4$, in sogenannten Clip-Koordinaten vorliegen. Die x-y-Koordinaten entsprechen hierbei den x-y-Achsen des Ausgabebildes, $-z$ entspricht der Blickrichtung des Betrachters, ist also die Tiefe. Die w-Koordinate wird für die perspektivische Division benutzt. Ist w ungleich null, so bringt das Teilen aller Koordinaten durch w einen Vertex in *normalisierte Gerätekoordinaten*. Für die Anwendung in dieser Arbeit sind nur die x-y-Koordinaten relevant.

OpenGL rasterisiert nur Teile von Objekten, deren x-y-Koordinaten nach dem Vertex-Shader in dem Rechteck $[-1.0, 1.0]\times[-1.0, 1.0]$ liegen[7]. Wurden Vertices so transformiert, dass sie dieser Konvention nachkommen, so befinden sie sich in *normalisierten Gerätekoordinaten*. Fragmente werden nur für diesen Bereich generiert und alles ausserhalb dieses Rechtecks wird durch Clipping beschnitten. Der Fragment-Shader arbeitet auf interpolierten Vertex-Attributen und soll als Ergebnis einen Farbwert in den Framebuffer übergeben. Die Vertex-Attribute werden im zu zeichnenden Dreieck für die Stelle interpolierten, an der ein Pixel gesetzt wird. Dies ermöglicht für Grafikanwendungen pixelgenaue Berechnung von Beleuchtung. Ein erzeugtes Fragment muss nicht in jedem Fall zu einem Pixel im fertigen Bild werden. Ein Fragment kann sich selber als ungültig markieren oder aufgrund anderer Mechanismen von anderen generierten Fragmenten überdeckt oder im Bild überschrieben werden. In beiden Fällen landet ein Fragment nicht im resultierenden Bild. Wird ein Fragment jedoch gezeichnet, wird es zwingend im Framebuffer gespeichert. Der Framebuffer muss jedoch nicht unbedingt im Hauptfenster der Anwendung angezeigt werden. Auch ein oder mehrere Texturen im Grafikspeicher können als Framebuffer genutzt werden, und im weiteren Verlauf können diese Texturen als Quelle für weitere Berechnungen dienen (sogenanntes Off-Screen Rendering).

Hat ein Framebuffer eine Breite von w Pixeln und eine Höhe von h Pixeln, so werden die normalisierten Gerätekoordinaten auf die Größe des Framebuffers skaliert, bzw. es wird ein Gitter der Größe $(w + 1, h + 1)$ auf das Einheitsrechteck gelegt, und die resultierenden Zellen stellen die Pixel dar. Für Teile von zu malenden Dreiecken, die nun in Zellen liegen, werden Fragmente erzeugt, die vom Fragment-Shader eine Farbe erhalten. Für die Anwendung in dieser Arbeit haben alle Texturen und Framebuffer quadratische Form, also gleiche Höhe und Breite.

3.2 Ein einfacher Schnitttest

Gegeben seien zwei Polygone und dazugehörige Transformationsmatrizen, die eine Konstellation der Polygone realisieren. Für diese Konstellation soll nun geprüft werden, ob sich die beiden Polygone schneiden. Die Darstellung der Polygone wird zum Programmstart einmalig OpenGL mitgeteilt, hierfür ist vor allem die Triangulierung des Polygons nötig. Alle jeweils nötigen Transformationen für Polygone eines Schnitttests werden stets als Uniform-Variablen an den zuständigen Vertex-Shader übergeben. Die Farbe, in der ein Polygon gezeichnet werden soll, wird ebenfalls als Uniform-Variable übergeben.

Im folgenden wird ein Schnitttest beschrieben, der die Polygone in ein Bild projiziert und erkennt für welche Pixel des Bildes beide Polygone einen Farbwert beisteuern wollten. Solche Pixel gelten

[7] Hier wird bewusst die Tiefendimension und der Zusammenhang zur Near- und Far-Clipping Plane ausgelassen, da für diese Arbeit die Verarbeitung in x-y-Koordinten ausreichend ist.

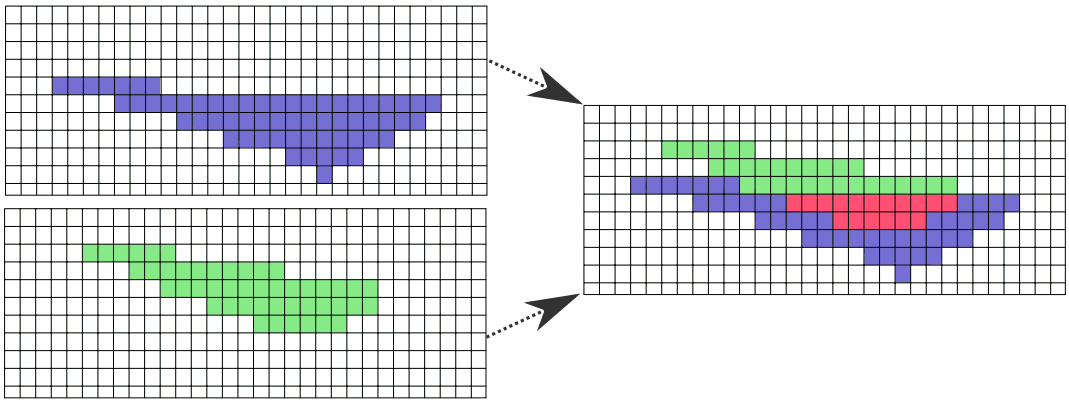

Abbildung 3.3: Anhand zwei rasterisierter Polygone wird die Schnittfläche erkannt.

als Nachweis, dass die Polygone sich in dieser Stellung schneiden. Eine konzeptionelle Darstellung des Schnitttests ist in Abbildung 3.3 zu sehen. Es kann natürlich sein, dass Teile der Polygone, die sich nicht schneiden, auf denselben Pixel gezeichnet werden sollen. Dies wäre ein *false-positive*, also ein Schnitt, der tatsächlich keiner ist, und dieses Verhalten ist eine unvermeidliche Folge der Rasterisierung. Durch eine höhere Auflösung des Bildes lassen sich solche fälschlich erkannten Schnitte besser vermeiden.

Die Schnitterkennung erfolgt idealerweise komplett auf der GPU unter OpenGL, da ein Austausch von Daten zwischen Hauptspeicher und Grafikkarte lange dauert und daher zu minimieren ist. Ein naiver Ansatz, beide Polygone in ein Bild zu malen, funktioniert aus verschiedenen Gründen nicht. Für jedes zu malende Polygon ist (lässt man fortgeschrittene Techniken der Computergrafik ausser Acht) ein eigener Draw Call nötig. Das bedeutet, dass die Dreiecke des Polygons die Rendering-Pipeline durchlaufen, im Vertex-Shader auf Clipping-Koordinaten, anschliessend auf normalisiere Gerätekoordinaten transformiert werden, im Fragment-Shader eine Farbe erhalten, und letztendlich in einem Ausgabebild landen. Die Schwierigkeit besteht darin, dass ein Fragment, welches an einen bestimmten Ort im Ergebnisbild als Pixel geschrieben werden soll, den dort vorher befindlichen Pixel überschreiben wird. Der Fragment-Shader für dieses Fragment hat auch keine einfache Möglichkeit heraus zu finden, ob an der entsprechenden Stelle im Zielbild bereits ein Pixel eine bestimmte Farbe hat. Ähnliche Mechanismen sind zwar zum Beispiel für Tiefentests[8] zugänglich, stellen sich aber als nicht hilfreich heraus, da sie im Kontext von OpenGL für andere Aufgaben ausgelegt sind. Es reicht also nicht, beide Polygone hintereinander per Draw Call in das selbe Bild zu malen, da kein einfacher Mechanismus zu Verfügung steht um bei diesem Vorgehen einen Schnitt zu erkennen.

Wie bereits ausgeführt, unterstützt OpenGL verschiedene Arten von Framebuffern. Dies lässt die Möglichkeit zu, in Texturen zeichnen zu lassen, und diese Texturen anschliessend für weitere Berechnungen zu verwenden. Ein solches Vorgehen wird im Allgemeinen *Multipass Rendering* genannt. Folgender Schnitttest liegt hierdurch nahe:

[8] Jedes Fragment kann eine Tiefe erhalten, im Sinne einer Entfernung des gezeichneten Objektes vom Betrachter in der dargestellten Szene. Sollen zwei Fragmente an derselben Stelle gezeichnet werden, kann anhand des Tiefentests dasjenige Fragment gezeichnet werden, das näher zum Betrachter liegt und somit das andere verdeckt.

1. Initialisiere zwei gleich große, schwarze Texturen A und B.
2. Erster Pass: Zeichne das *erste* Polygon in Textur A.
3. Zweiter Pass: Zeichne das *zweite* Polygon in Textur B.
4. Vergleiche beide Texturen pixelweise. Falls ein Pixel in beiden Texturen gezeichnet wurde, schneiden sich die Polygone.

Der Vergleich der beiden Texturen soll nicht auf der CPU stattfinden, da hierfür die Texturen in den Hauptspeicher übertragen werden müssen. Sollen die Texturen durch OpenGL verglichen werden, muss dies im Rahmen der Rendering-Pipeline geschehen. Dies ist möglich, indem ein dritter Verarbeitungsschritt, also ein dritter Pass angefügt wird. Im dritten Pass wird ein Rechteck (bestehend aus zwei Dreiecken) so gezeichnet, dass es die normalisierten Gerätekoordinaten ganz ausfüllt. Hierdurch wird für jeden Pixel des Framebuffers genau ein Fragment erzeugt. Zusätzlich wird die Auflösung des Framebuffers gleich der Auflösung der Texturen gesetzt, in die beide Polygone gezeichnet wurden. Im Fragment-Shader werden die Texturen A und B verfügbar gemacht, und abhängig von den Werten in den jeweiligen Pixeln kann das Fragment einen Schnitt der Polygone kommunizieren. Weil ein Fragment seine Position im resultierenden Bild kennt, kann es anhand dieser Koordinaten auf die korrespondierenden Pixel in den Texturen zugreifen und diese vergleichen. Mithilfe dieses Vorgehens kann zum Beispiel in eine Dritte Textur pixelweise das Ergebnis des Vergleichs geschrieben werden. Der Nachteil daran, eine gesamte dritte Textur als Resultat abzuspeichern, besteht darin, dass man diese Textur wieder auf der CPU auswerten muss, um an das Ergebnis des Schnitttests zu gelangen. Dies kann mithilfe sogenannter *Atomic Counters* vermieden werden.

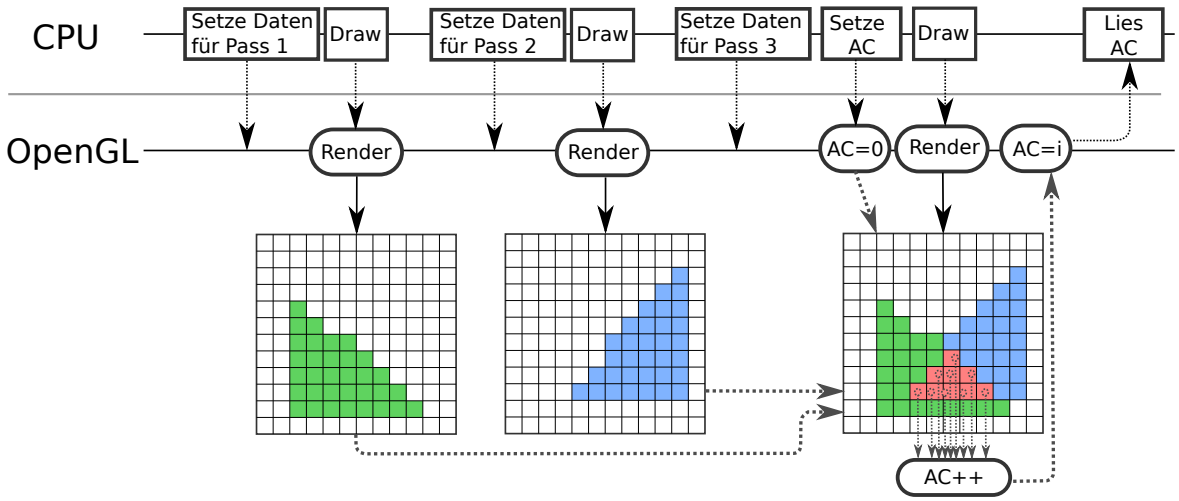

Abbildung 3.4: Schnitterkennung anhand rasterisierter Polyone und eines Atomic Counters (AC). In den ersten zwei Pässen werden die Polygone in separate Texturen gerendert, anschliessend werden die Texturen auf der GPU verglichen und anhand eines Atomic Counter Überschneidungen erkannt. Das Ergebnis des Tests wird von der CPU aus dem Atomic Counter gelesen.

Abbildung 3.4 zeigt die drei Pässe des entstandenen Schnitttests, skizziert die Verwendung der Teilresultate und das Erkennen von Überschneidungen durch einen *Atomic Counter*. *Atomic Counters*

sind ein Feature der OpenGL Version 4.2 und erlauben es, in parallel ausgeführten Shader Instanzen einen gemeinsamen, zentralen Zahlenwert (*Unsigned Integer*) atomar auslesen und hochzählen zu lassen. Dieser Atomic Counter liegt im Speicher der GPU, da dort der Zugriff darauf erfolgt. Zusätzlich kann der Wert eines solchen Atomic Counters über die CPU initial gesetzt und bei Bedarf wieder ausgelesen werden. Statt also eine große Textur auf den Hauptspeicher zu übertragen, bietet es sich an, im dritten Pass im Fragment-Shader einen Atomic Counter immer dann hochzählen zu lassen, wenn ein Fragment einen Schnitt gefunden hat. Durch die garantiert atomare Operation bleibt sichergestellt, dass korrekte Ergebnisse entstehen. Andere mögliche Alternativen, die ein solches Verhalten eines Zählers nachahmen, sind generell auch ohne Atomic Counter möglich, zum Beispiel durch Schreiben in eine weitere Textur. Solche schreibenden Speicherzugriffe auf Texturen werden jedoch als teure Operationen erachtet und sind nicht zwingend atomar. Der bisher aufgebaute Schnitttest wird also insofern modifiziert, dass vor dem dritten Pass die CPU einen Atomic Counter initial auf Null setzt, und der Fragment-Shader im dritten Pass diejenigen Fragmente den Zähler inkrementieren lässt, die einen Schnitt gefunden haben. Nach dem dritten Pass liest die CPU den Zahlenwert des Atomic Counters aus. Dies bedeutet, dass der Unsigned Integer des Atomic Counters von der GPU in den Hauptspeicher übertragen werden muss, um das Resultat des Schnitttests für die CPU verfügbar zu machen. Ist diese gelesene Zahl größer Null, schneiden sich die Polygone in der gegebenen Konstellation.

3.3 Details und Erweiterungen des Schnitttests

Der beschriebene Schnitttest lässt sich durch einige Änderungen konzeptionell vereinfachen und dadurch merklich beschleunigen.

So wurde bisher nicht darauf eingegangen, wie genau die Transformationen aussehen, die die gegebenen Polygone in die zu testende Konstellation bringen. Grundsätzlich reicht es, wenn diese Transformationen sicherstellen, dass eines der Polygone voll im Bild liegt. Dies stellt sicher, dass eine Überschneidung in jedem Fall erkannt wird. Eine entsprechende Konvention kann sein, dass stets das kleinere Polygon durch eine Skalierung in das Bild eingepasst wird. Die bestmögliche Einpassung eines Polygons in das quadratische Bild ist mithilfe des *minimalen umfassenden Rechtecks* eines Polygons gegeben. Das minimale Rechteck ist schnell zu berechnen und wird auch für andere Zwecke eingesetzt, liegt also für jedes Polygon bereits vor. Anhand dieses minimalen Rechtecks wird für eine Konstellation zweier Polygone also das kleinere Polygon optimal auf das Ausgabebild eingepasst und skaliert. Hierdurch wird für die eingestellte Auflösung der Rasterisierung die bestmögliche Qualität erreicht, da kein Platz des Bildes ungenutzt bleibt. Dadurch werden für dieses Polygon auch maximal viele Fragmente für die aktuelle Auflösung erzeugt. Dieses Polygon sei fortan das *fixierte* Polygon. Das nicht fixierte Polygon sei das *freie* Polygon, im Sinne von frei schwebend.

Für das freie Polygon gibt es viele Möglichkeiten, in einer bestimmten Lage zum fixierten Polygon zu sein. Auf die Generierung dieser Konstellationen wird in Kapitel 4.1 genau eingegangen, hier wird jedoch vorweg genommen, dass alle als relevant betrachteten Stellungen der beiden Polygone in einem Zuge erfasst und bearbeitet werden sollen. Dies entspricht einer Sammlung, einem *Test-Batch* von zu testenden Konfigurationen. Die nötigen Transformationsmatrizen für das fixierte und das freie Polygon werden in einer Batch-Datenstruktur gespeichert, gemeinsam mit nötigen OpenGL Daten, um die Polygone zeichnen zu können.

Zunächst ist wünschenswert, von Seiten der CPU Verzögerungen im Schnitttest zu minimieren. Das heisst, eine möglichst hohe Lokalität herzustellen. Bevor OpenGL auf der GPU zeichnet, muss die CPU vor jedem Draw Call für jedes Polygon und jede Transformationsmatrix nötige Informationen bereit stellen. Dies ist in Abbildung 3.4 durch das CPU-seitige Setzen von Daten skizziert.

Wenn die nötigen Daten für die Matrizen und Polygone nicht lokal schnell verfügbar sind, muss auf diese unnötig gewartet werden. Daher ist es sinnvoll, einen Test-Batch als lokal zusammenhängenden Speicher, also als Array, anzulegen. Für Draw Calls nötige OpenGL Daten und Transformationsmatrizen werden hier für jeden Test hinterlegt, dies teilweise redundant, aber möglichst lokal. Statt zum Beispiel Zeiger auf Matrizen oder Polygondaten zu verwalten, werden diese Daten direkt in das Array lokal hinterlegt. Auch ein Feedback-Feld für das Ergebnis des Schnitttests wird lokal vorgehalten.

Da das fixierte Polygon nur eine Transformation braucht, wird diese nur ein mal gesondert gespeichert, und nicht für jeden Test des Batches nochmal gesondert. Lediglich für das freie Polygon müssen variierende Transformationen mitgeführt werden. Diese Unterteilung in fixes und freies Polygon vereinfacht auch den Schnitttest.

So muss das fixierte Polygon nur *ein mal* für den gesamten Test-Batch gezeichnet werden. Es entfällt also faktisch ein Pass pro Schnitttest. Der bisher dritte Pass, das Vergleichen beider Texturen, kann ebenfalls beseitigt werden. Der dritte Pass bringt darüber hinaus auch den Nachteil mit sich, dass er für die jeweilige Auflösung die maximal möglich Anzahl an Fragmenten generieren lässt, und jedes dieser Fragmente führt auch noch zwei teure Speicherzugriffe in Texturen durch. Einfacher ist es, bereits beim Zeichnen des nun freien Polygons nach Überschneidungen zu suchen. Der Fragment-Shader wird also so angepasst, dass für Fragmente des freien Polygons ein Texturzugriff auf das gezeichnete fixe Polygon erfolgt und auf einen Schnitt getestet wird. Der Atomic Counter wird gesetzt, inkrementiert und ausgelesen wie bisher, nur geschieht dies direkt beim Zeichnen des freien Polygons.

Die eben beschriebenen Erweiterungen des Schnitttests sind in Abbildung 3.5 dargestellt. Sie führen dazu, dass statt drei Draw Calls pro Schnitttest faktisch nur noch ein Draw Call pro Schnitttest nötig ist, zuzüglich initialem Draw Call für das fixierte Polygon. Das ist aber nur möglich, weil das freie Polygon *viele verschiedene Stellungen* zum fixierten haben kann. Zusätzlich wird versucht, über eine hohe Lokalität die CPU als Engpass im Schnitttest zu beseitigen [9].

[9] Bei technisch gut ausgestatteten Testsystemen erwies sich die beschriebene lokale Datenstruktur als weniger ausschlaggebend für die Geschwindigkeit. Auf dem technisch verhältnismäßig schwach bestückten privaten Notebook des Autors war jedoch eine Beschleunigung der Schnitttests nicht zu übersehen. Daher möchte der Autor bei dieser sowieso nötigen Datenstruktur auch die positiven Effekte für den CPU Cache betonen.

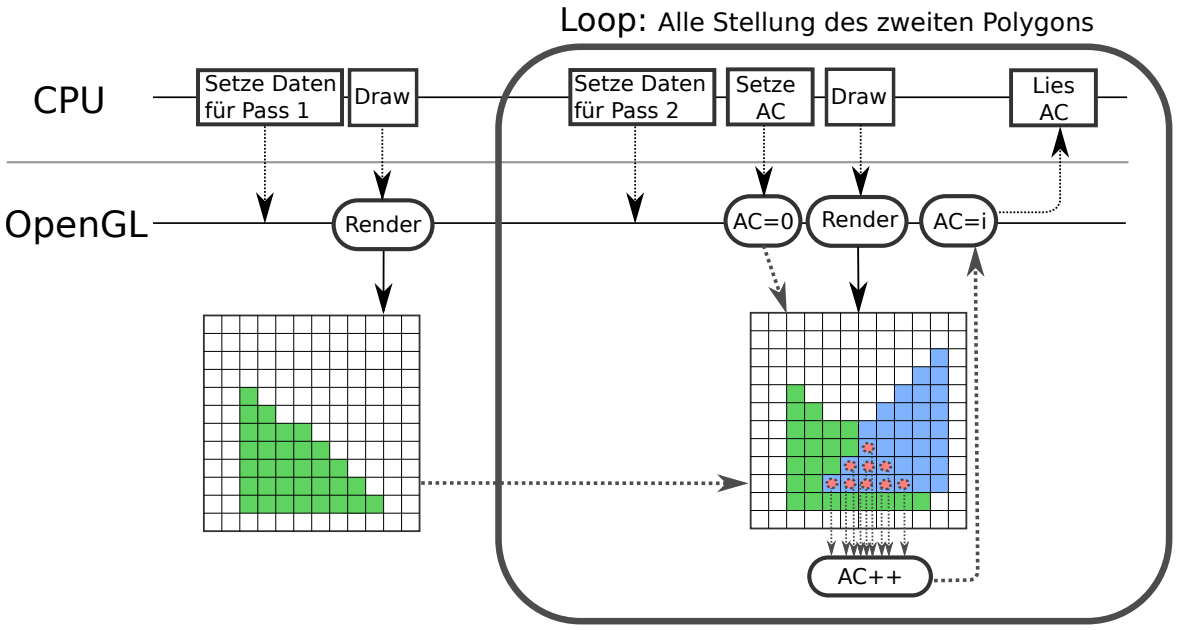

Abbildung 3.5: Vereinfachter Schnitttest. Statt für jede Stellung zweier Polygone beide zu zeichnen wird das erste Polygon in einer Position fixiert und das zweite relativ dazu verschoben. Hierdurch sind nur noch zwei Pässe nötig, einer davon nur ein mal, der zweite für jede Stellung des zweiten Polygons.

3.4 Visualisierung und Benutzereingaben

Durch die OpenGL-Anbindung für den Schnitttest sind bereits alle nötigen Mittel vorhanden, um eine Visualisierung von Polygonen und gefundenen Schachtelungen zu realisieren. Dies erwies sich im Laufe dieser Arbeit mehrfach als vorteilhaft, weil so die Fehlersuche bei implementierten Verfahren wesentlich vereinfacht wurde. Durch die verwendete Bibliothek GLFW ist es möglich, Benutzereingaben über Tastatur und Maus zu verarbeiten. Das ermöglicht eine interaktive Visualisierung von Polygonen und ihrer Eigenschaften. Die implementierte Visualisierung ist in Abbildung 3.6 zu sehen und unterstützt das Anzeigen einfacher Polygone und Linienzüge, Navigieren und Zoomen in der Darstellung, die Anzeige der Konvexen Hülle, des minimalen Rechtecks und das hervorheben verdeckter Vertices.

Über die durch GLFW verfügbaren Benutzereingaben wurde auch eine Maus basierte Eingabe von Polygonen implementiert. Der Benutzer kann per Mausklick im Anwendungsfenster Vertices definieren. Diese werden laufend an OpenGL kommuniziert und der entstehende Linienzug wird angezeigt. Der eingegebene Linienzug muss entgegen dem Uhrzeigersinn orientiert sein, um später als gültiges Polygon verwendet werden zu können. Anderenfalls kann das Polygon nicht als Eingabe für eine Schachtelung genutzt werden, da die Triangulierung des Polygons fehl schlägt. Ist das eingegebene Polygon gültig, wird es vom Programm direkt für eine Schachtelung benutzt. Gleichzeitig werden die Vertices des Polygons in Textform ausgegeben, sodass sie in eine Datei übertragen werden können, um das Polygon anhand dieser Datei zu einem anderen Zeitpunkt zu verwenden. Abbildung 3.7

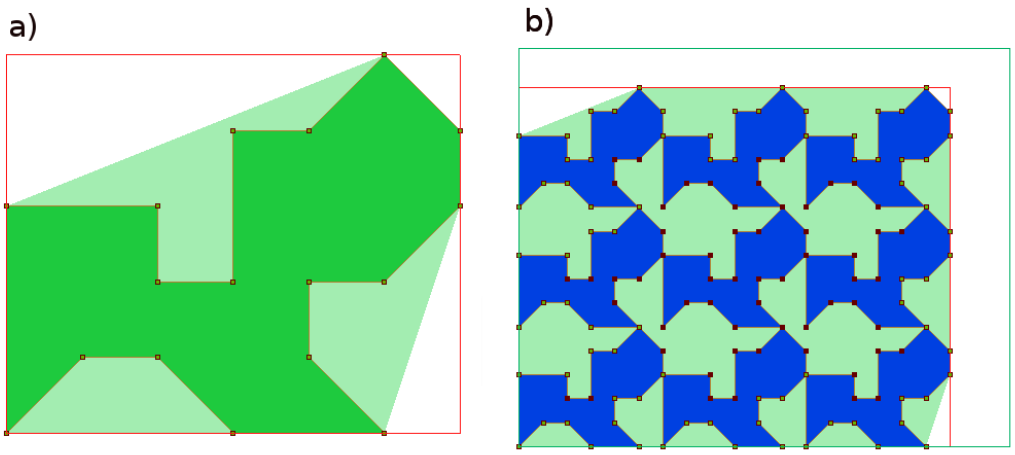

Abbildung 3.6: a) Visualisierung eines Polygons (dunkelgrün) mit Konvexer Hülle (hellgrün) und minimalen Rechteck (rot). **b)** Visualisierung von angeordneten Polygonen (blau) mit Konvexer Hülle aller Polygone (grün), minimalen Rechteck (rot) und dem Gebiet, in dem die Polygone anzuordnen sind (grünes Rechteck).

zeigt einen eingegebenen Linienzug und die dadurch resultierende Darstellung des Polygons im Programm.

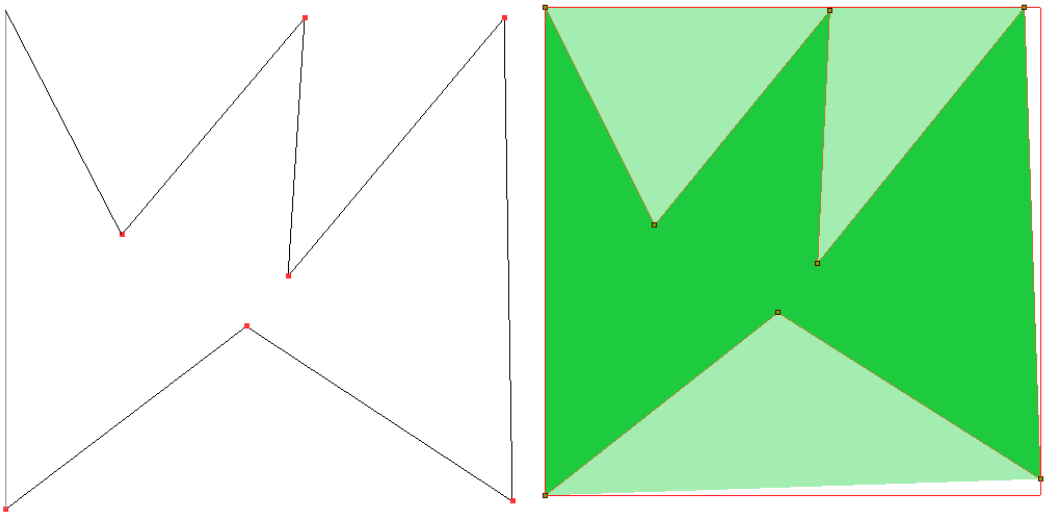

Abbildung 3.7: Eingabe eines links orientierten Linienzuges *(links)* und das daraus erzeugte Polygon mit dessen Eigenschaften *(rechts)*.

4 Nesting von Polygonen

Nachdem der Schnitttest und Algorithmen für den Umgang mit Polygonen erläutert sind, wird das tatsächliche Platzieren von Polygonen behandelt. Gegeben seien also ein rechteckiges Gebiet, eine Domäne $\mathcal{G}$ anhand seiner Seitenlängen $(x, y) = \mathcal{G}$ und ein einfaches Polygon P_n. Im Laufe der nächsten Unterkapitel werden Methoden vorgestellt, die zum Ziel haben das Polygon P_n möglichst oft in das Gebiet $\mathcal{G}$ zu platzieren, ohne dass sich zwei Instanzen von P_n überschneiden.

Es wird versucht anhand geometrischer Eigenschaften der gegebenen Polygone gute Nestings zunächst für wenige Polygone zu finden, statt sich direkt global an der gegebenen Domäne $\mathcal{G}$ zu orientieren. Statt beliebige Verschiebungen und Drehungen von Polygonen zu erlauben, werden ausschließlich diskrete Stellungen von Polygonen betrachtet. Das bedeutet, dass Polygone zunächst *nur an Vertices* zusammen gelegt werden, und anschließend so rotiert, dass die anliegenden Seiten der Polygone einander berühren (jedoch nicht überschneiden). Da keine beliebigen Stellungen mehr betrachtet werden, wird die Anzahl möglicher Lösungen erheblich eingeschränkt. Gleichzeitig erfolgt durch das diskrete Vorgehen eine Einschränkung auf Polygon-Konstellationen, die durch die Form und Eigenschaften der Polygone vorgegeben sind. Dies ermöglicht eine einfachere Betrachtung der Aufgabenstellung. Eine Rückkehr zu beliebigen Verschiebungen und Rotationen ist konzeptionell jederzeit möglich. Allerdings bewirken zusätzliche, randomisierte Konstellationen beliebig mehr Aufwand bei der Lösungsfindung.

Im Folgenden werden also Methoden entwickelt, um unter diskreten Stellungen den Rechenaufwand einer Lösungsfindung zu reduzieren. Hierfür wird zunächst untersucht, wie Polygone zusammen zu legen sind, sodass diese Konstellation als optimal angesehen werden kann. Stellungen der Polygone, in denen diese sich schneiden, sind ungültig und werden nicht weiter betrachtet. Weitere Verfahren, um ungültige Stellungen frühzeitig zu erkennen und dauerhaft zu vermeiden werden entwickelt, sodass Rechenzeit für teure Schnitttests nur für tatsächlich nötige Polygon-Stellungen investiert werden muss. An schnell gefundene, vielversprechende Polygon-Konstellationen sollen iterativ weitere Polygone hinzugefügt werden, mit dem Ziel, möglichst wenig Platz zwischen den Polygonen ungenutzt zu lassen. So gefundene, iterativ erweiterte Lösungen werden anschließend verwendet, um eine gültige globale Lösung zu finden.

4.1 Ein lokaler diskreter Brute-Force Ansatz

Gegeben seien zwei Polygone $P_n = (v_1, \ldots, v_n)$ und $Q_m = (w_1, \ldots, w_m)$. Gesucht wird die beste, im Sinne von engste, Zusammenlegung dieser beiden Polygone. Wie bereits erläutert, soll dieses enge Nesting von P und Q realisiert werden, indem nur diskrete Stellungen der beiden Polygone betrachtet werden. Die Polygone werden also an Vertices v_i und w_j zusammengelegt, und die an den

Vertices anliegenden Segmente werden so rotiert, dass sich die Polygone an diesen Seiten schnittfrei berühren. Hieraus ergeben sich pro Vertex-Paar zwei Möglichkeiten, die Segmente zu Kombinieren, ohne dass sich die Polygone auf triviale Weise überschneiden. Für die Polygone gibt es somit genau

$$|P_n \times Q_m| = n \cdot m \cdot 2 \ \in \mathcal{O}(nm)$$

Möglichkeiten, wie sie diskret gepaart werden können. Hierbei wird mit $P_n \times Q_m$ ausgedrückt, dass alle möglichen Stellungen der Polygone betrachtet werden. Weil die Polygone sich an jeweils einem Segment berühren, können sie im Sinne des diskreten Nestings nicht noch enger zusammen gelegt werden. Abbildung 4.1 zeigt verschiedene diskrete Stellungen zweier Polygone, wie sie in dieser Arbeit untersucht werden.

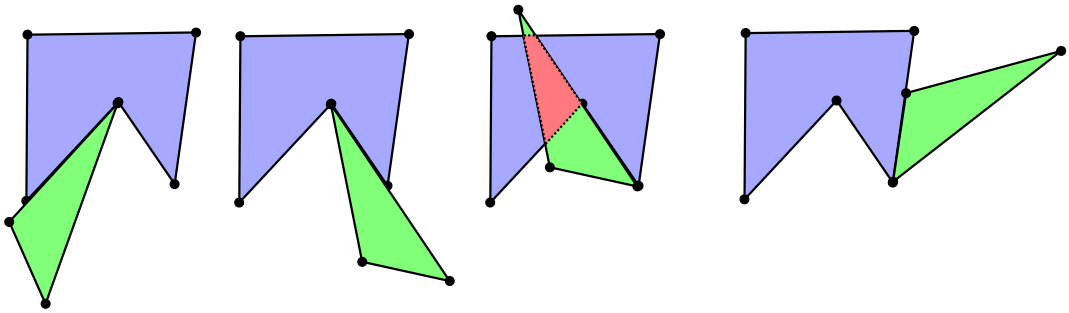

Abbildung 4.1: Zwei Polygone in verschiedenen diskreten Stellungen.

Alle solche Stellungen werden erstellt und in einer Datenstruktur gespeichert. Neben den Indizes der beteiligten Vertices wird auch die Ausrichtung der Segmente notiert. Diese Ausrichtung kann zum Beispiel durch eine gewählte Durchnummerierung der zwei Möglichen Segment-Kombinationen erfolgen. Eine (Polygon-)Stellung ist also ein Tupel

$$(\text{P Vertex ID}, \text{Q Vertex ID}, \text{Segment-Kombination}) = (v\_id,\ w\_id,\ c)$$

und alle möglichen Stellungen werden aufgezählt durch folgenden Pseudocode:

for $v\_id = 1 \ldots n$ **do**
  for $w\_id = 1 \ldots m$ **do**
    for $c = -1, 1$ **do**
      Speichere Polygon-Stellung $(v\_id,\ w\_id,\ c)$
    end for
  end for
end for

Diese Stellungen sind jedoch nicht zwangsläufig schnittfrei, es wird also Stellungen geben, in denen sich die Polygone schneiden. Aus den Informationen der Stellungen können entsprechende Transformationsmatrizen erzeugt werden, die diese Stellung realisieren. Zwar können solche Transformationen auch bereits bei der Aufzählung aller Stellungen erzeugt werden, aber für weitere Verfahren, die noch erläutert werden, ist es einfacher die aufgezählten Stellungen und die dazugehörigen Transformationen getrennt zu behandeln. Die Transformationen werden erzeugt, indem für jede Stellung die Polygone an den durch v_id und w_id angegebenen Vertices zum Nullpunkt verschoben werden,

und die durch c gegebenen Segmente so rotiert werden, dass sie in die selbe Richtung zeigen (der Einfachheit halber nach $\begin{pmatrix}1 & 0\end{pmatrix}^\top$).

$$
\begin{aligned}
M_P &= R\Big(\frac{r_p}{|r_p|}\Big)^{-1} \cdot T(-v_{v\_id}) \\
M_Q &= R\Big(\frac{r_q}{|r_q|}\Big)^{-1} \cdot T(-w_{w\_id}) \\
r_p &= v_{v\_id-c} - v_{v\_id} \\
r_q &= w_{w\_id+c} - w_{w\_id} \\
c &\in \{-1, 1\}
\end{aligned}
$$

Die Notation von r_p und r_q berechnet die Richtung, in die das Segment zeigt, das an v_{v_id}, bzw. w_{v_id} liegt. Um den in Kapitel 3.3 entwickelten Schnitttest mit einem *fixierten* und einem *freien* Polygon durchführen zu können, ist eine leicht angepasste Generierung der Transformationen nötig. Hierfür wird zunächst angenommen, dass P_n dasjenige Polygon von beiden ist, dessen minimales Rechteck kleiner ist. Dann ist P_n das fixierte Polygon und dessen Transformation ist die Einheitsmatrix. Q_m muss am Vertex w_{w_id} zunächst in den Nullpunkt verschoben werden, dort werden Rotationen verrechnet, die Q_m passend am Segment von P_n ausrichten, und anschließend wird Q_m an die richtige Stelle an P_n gesetzt.

$$
\begin{aligned}
M_P &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \\
M_Q &= T(v_{v\_id}) \cdot R\Big(\frac{r_p}{|r_p|}\Big) \cdot R\Big(\frac{r_q}{|r_q|}\Big)^{-1} \cdot T(-w_{w\_id}) \\
r_p &= v_{v\_id-c} - v_{v\_id} \\
r_q &= w_{w\_id+c} - w_{w\_id} \\
c &\in \{-1, 1\}
\end{aligned}
$$

Anhand gegebener Daten zum minimalen Rechteck von P_n kann die Schnitttest-Funktion eine optimale Einpassung von P_n auf den Zeichenbereich vornehmen, durch die hierfür genutzte Transformation wird auch Q_m für den Schnitttest korrekt transformiert und die Stellung der Polygone zueinander bleibt erhalten. Mit den Transformationsmatrizen für die Stellungen können auch nötige OpenGL Daten generiert und die in Kapitel 3.3 eingeführte Test-Batch Datenstruktur aufgestellt werden.

Wie bereits erwähnt, können sich unter den generierten Konstellationen solche befinden, bei denen sich die Polygone schneiden. Solche ungültigen Konstellationen müssen entfernt werden. Dies passiert, indem der erstellte Test-Batch mit allen Konfigurationen an den Schnitttest übergeben wird. Der Test-Batch vermerkt für jede Stellung das Ergebnis des Schnitttests und ungültige Stellungen werden in einem weiteren Schritt gelöscht. Aufgrund der hohen Anzahl möglicher Stellungen ist der Schnitttest der zeitaufwändigste Schritt der Lösungsfindung.

Die nun verbliebenen Stellungen können ausgewertet und weiterverwendet werden. Von nun an stellt sich die Frage, welche Konfigurationen der Polygone als engste angesehen werden können, und somit für eine weitere Verarbeitung und Schachtelung besonders interessant sind.

4.2 Bewerten und Verwalten von Polygon-Stellungen

Ein wichtiger Aspekt in der Handhabung von Polygon-Konfigurationen ist deren tatsächliche Verwaltung in Datenstrukturen. Unter Betrachtung der nötigen Eigenschaften einer solchen Struktur, und der zur Verfügung stehenden Mittel aus Kapitel 2.2 wird im Folgenden eine Verwaltung von Polygon-Stellungen realisiert, die das Nesting mit weiteren Polygonen einfach gestaltet und Optimierungen für diesen Prozess motiviert.

Es ist wünschenswert nur solche schnittfreien Konfigurationen zweier Polygone weiter zu verwenden, die vielversprechend für weitere Nestings sind. Ein Weiterrechnen mit allen gefundenen Konfigurationen ist zwar möglich, bedeutet aber mit steigender Zahl beteiligter Polygone einen enormen Zuwachs an benötigter Rechenzeit. Dieses Kapitel stellt einfache Heuristiken vor, wie gefundene Konfigurationen qualitativ bewertet werden können. Anhand solcher Heuristiken kann die Anzahl der zu betrachtenden möglichen Polygon-Stellungen reduziert werden, indem nur vielversprechende Konfigurationen für weitere Berechnungen verwendet werden. Trotz dieser Einschränkung soll die Möglichkeit erhalten bleiben, eine optimale Lösung zu finden, also maximal viele Polygone im Gebiet zu positionieren.

4.2.1 Modellierung zusammengesetzter Polygone

Eine Stellung mehrerer Polygone wird ausgedrückt durch die beteiligten Polygone (in Form von Linienzügen) und Transformationsmatrizen, die die Polygone entsprechend positionieren. Diese Datenstruktur wird fortan als *Polygon-Patch* $\mathcal{P}$ bezeichnet, also als Ansammlung von Polygonen. Ein Polygon-Patch ist eine schnittfreie Konstellation von Polygonen, die einen Verweis auf die beteiligten Polygone enthält, zusammen mit den dazugehörigen Transformationsmatrizen und weiteren Eigenschaften der Konstellation.

Der Aufbau eines Polygon-Patches erfolgt hierarchisch durch Zusammenlegen zwei bereits bestehender Polygon-Patches, wie in Abbildung 4.2 dargestellt. Initial besteht ein Polygon-Patch also aus nur einem Polygon, dessen Transformationsmatrix die Identitätsmatrix ist. Der Flächeninhalt, die Konvexe Hülle, und das minimale Rechteck des Polygons charakterisieren somit auch initial den Polygon-Patch. Ist eine schnittfreie Konstellation von zwei Polygonen gefunden, wird diese Konstellation benutzt, um einen neuen Polygon-Patch zu erzeugen. Seien also $P^{\mathcal{P}}$ und $Q^{\mathcal{P}}$ Polygon-Patches, dazu zwei Matrizen M_P und M_Q, die eine Konstellation der Polygon-Patches realisieren, indem sie auf alle enthaltenen Polygone eines Patches angewendet werden. Ein neuer Polygon-Patch $N^{\mathcal{P}}$ wird erzeugt, indem alle in $P^{\mathcal{P}}$ und $Q^{\mathcal{P}}$ beteiligten Polygone in $N^{\mathcal{P}}$ eingetragen werden.

$$N^{\mathcal{P}} = P^{\mathcal{P}} \cup Q^{\mathcal{P}}$$

Hier ist $P^{\mathcal{P}} \cup Q^{\mathcal{P}}$ als Zusammenlegen von Polygon-Patches zu interpretieren, bzw. als Vereinigung der in den Polygon-Patches enthaltenen Mengen von Polygonen. Zusätzlich werden die zu den Polygonen gehörigen Transformationsmatrizen aus $P^{\mathcal{P}}$ und $Q^{\mathcal{P}}$ mit M_P, bzw. mit M_Q verrechnet und ebenfalls in $N^{\mathcal{P}}$ eingetragen, um für die in $N^{\mathcal{P}}$ enthaltenen Polygone korrekte Transformationen zu erhalten. Da $P^{\mathcal{P}}$ und $Q^{\mathcal{P}}$ jeweils eine Konvexe Hülle $\mathrm{kh}(P^{\mathcal{P}})$, bzw. $\mathrm{kh}(Q^{\mathcal{P}})$ haben, wird aus diesen mithilfe des in Kapitel 2.2.2 vorgestellten Algorithmus die Konvexe Hülle $\mathrm{kh}(N^{\mathcal{P}})$ der gesamt Konstellation

berechnet. Die Konvexen Hüllen der beiden ursprünglichen Patches müssen hierfür mithilfe der Transformationsmatrizen in die benötigte Stellung gebracht werden.

$$\mathrm{kh}(N^{\mathcal{P}}) = \mathrm{kh}\left(\mathrm{kh}(P^{\mathcal{P}}) \cdot M_P \ \cup \ \mathrm{kh}(Q^{\mathcal{P}}) \cdot M_Q\right)$$

Hieraus kann wiederum das minimale Rechteck $\mathrm{mr}(N^{\mathcal{P}})$ der Konstellation errechnet werden.

$$\mathrm{mr}(N^{\mathcal{P}}) = \mathrm{mr}(\mathrm{kh}(N^{\mathcal{P}}))$$

Der Flächeninhalt $\mathrm{area}(N^{\mathcal{P}})$, der durch die Polygone in $N^{\mathcal{P}}$ eingenommen wird, ergibt sich als Summe der Flächeninhalte der Polygone von $P^{\mathcal{P}}$ und $Q^{\mathcal{P}}$. Als Flächeninhalt wird also die Summe der Flächeninhalte aller Polygone mitgetragen, etwaige Lücken zwischen Polygonen im Patch werden nicht zum Flächeninhalt gezählt.

$$\mathrm{area}(N^{\mathcal{P}}) = \mathrm{area}(P^{\mathcal{P}}) + \mathrm{area}(Q^{\mathcal{P}})$$

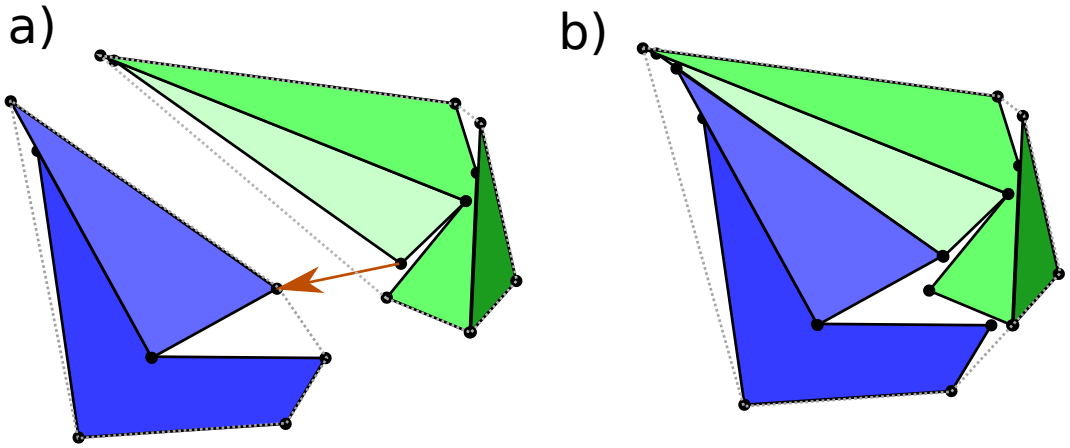

Abbildung 4.2: Zwei Polygon-Patches mit jeweils zwei und drei Polygonen werden anhand einer Stellung zusammen gelegt.

Das Erzeugen von neuen Konstellationen für zwei gegebene Polygon-Patches geschieht wie in Kapitel 4.1 beschrieben anhand der Vertices der enthaltenen Polygone. Die Polygone in einem Polygon-Patch sind anhand intern vergebener Indizes identifizierbar, und das Konstrukt für Test-Generierungen wird um diese Polygon-IDs erweitert. Eine Stellung von zwei Polygon-Patches $P^{\mathcal{P}}$ und $Q^{\mathcal{P}}$ wird dann dargestellt als folgendes Tupel:

$$(P^{\mathcal{P}} \text{ Polygon P ID, P Vertex ID, } Q^{\mathcal{P}} \text{ Polygon Q ID, Q Vertex ID, Segment-Kombination})$$

$$= (poly\_p\_id,\ v\_id,\ poly\_q\_id,\ w\_id,\ c)$$

Die Kombination von interner Polygon-ID und der Vertex-ID zu diesem Polygon kann auch als Meta-Index für den gesuchten Vertex im Polygon-Patch verstanden werden. Anhand dieser Identifizierung von Vertices können nötige Test-Batches auf dieselbe Weise für Polygon-Patches aufgestellt werden, wie sie für einzelne Polygone vorgestellt wurden. Alle möglichen Stellungen zweier Polygon-Patches werden über mögliche Stellungen aller beteiligter Polygone aufgezählt.

for $poly\_p\_id = 1 \ldots (\max \mathrm{P}^{\mathcal{P}}$ Polygons$)$ **do**
  for $poly\_q\_id = 1 \ldots (\max \mathrm{Q}^{\mathcal{P}}$ Polygons$)$ **do**
    for $v\_id = 1 \ldots n$ **do**
      for $w\_id = 1 \ldots m$ **do**
        for $c = -1, 1$ **do**
          Speichere Polygon-Stellung ($poly\_p\_id$, v_id, $poly\_q\_id$, w_id, c)
        end for
      end for
    end for
  end for
end for

Die so generierten Tests werden über einen leicht angepassten Schnitttest von ungültigen Konfigurationen bereinigt. Die Konvexe Hülle eines Polygon-Patches ist leicht zu berechnen, weshalb auch das minimale Rechteck verfügbar ist. Das minimale Rechteck kann für den bereits entwickelten Schnitttest benutzt werden, um ein ganzes Polygon-Patch auf Überschneidung mit einem zweiten zu prüfen. Der Schnitttest wird also so modifiziert, dass er einen Polygon-Patch mithilfe seines minimalen Rechtecks in das Zeichenfeld einpasst, und anschließend alle im Patch enthaltenen Polygone zeichnet. Aufgrund des einfachen Aufbaus des Schnitttests ist dies leicht umsetzbar, denn statt nur einem Polygon werden nun alle Polygone eines Polygon-Patches pro Pass gezeichnet. Ein Polygon-Patch benötigt also auch Informationen, anhand derer er per OpenGL gezeichnet werden kann. Dies ist im Grunde eine Ansammlung der OpenGL-Daten der beteiligten Polygone.

Von nun an wird für Polygon-Konstellationen angenommen, dass diese als Polygon-Patches umgesetzt werden. Auch werden die Begriffe Polygon und Polygon-Patch synonym füreinander verwendet. Im weiteren Verlauf der Arbeit wird es darauf ankommen Polygon-Patches so zu manipulieren, dass beim Nesting mit weiteren Polygon-Patches anfallende Testfälle zahlenmäßig verringert werden, um Rechenzeit zu sparen.

4.2.2 Einfache Bewertungsfunktionen

Soll ein Polygon P_n auf einem Gebiet $\mathcal{G}$ möglichst oft platziert werden, so ist die optimale Lösung nach oben Beschränkt durch die Flächeninhalte des Polygons und des Gebiets.

$$\left\lfloor \frac{\mathrm{area}(\mathcal{G})}{\mathrm{area}(P)} \right\rfloor = k_{max}$$

Sollen k gleiche Polygone in einem Gebiet verteilt werden, müssen diese (im Kontext dieser Arbeit) in diskreten Stellungen zueinander platziert sein. Für $k \geq 2$ Polygone gibt es

$$2n^2 \underbrace{\cdot 2n \cdot \ldots \cdot 2n}_{k-2 \text{ mal}} = \; n^k \cdot 2^{k-1} \quad \leq n^{k_{max}} \cdot 2^{k_{max}-1}$$

Möglichkeiten, die Polygone diskret miteinander zu kombinieren. Nicht alle diese Stellungen sind schnittfrei oder einzigartig. Selbst wenn Stellungen mit Überschneidungen entfernt werden, sollten nur vielversprechende Konstellationen tatsächlich erzeugt und betrachtet werden. Statt also über

einen Brute-Force Ansatz global exponentiell viele Stellungen zu betrachten, ist es nötig anhand von Heuristiken eine Auswahl von Polygon-Stellungen zu treffen, die trotz der getroffenen Einschränkung eine gute (oder gar optimale) Lösung ermöglichen.

Eine Möglichkeit Polygon-Stellungen zu bewerten ist die Konvexen Hülle der gesamten Stellung, beziehungsweise der Flächeninhalt der Konvexen Hülle. Der Flächeninhalt $\mathrm{area}(\mathrm{kh}(P^{\mathcal{P}}))$ der Konvexen Hülle eines Polygon-Patches muss mindestens so groß sein wie der Flächeninhalt $\mathrm{area}(P^{\mathcal{P}})$ des Polygon-Patches selber:

$$\mathrm{area}(\mathrm{kh}(P^{\mathcal{P}})) \geq \mathrm{area}(P^{\mathcal{P}})$$

Je enger die Polygone im Polygon-Patch zusammen liegen, desto kleiner wird auch der Flächeninhalt der Konvexen Hülle. Die Fläche der Konvexen Hülle wird also als Maß dafür verwendet, wie eng Polygone zusammenliegen. Dieser Sachverhalt ist in Abbildung 4.3 skizziert. Solche Stellungen können bevorzugt betrachtet werden, denn sind Polygone schon in einer gewissen Konstellation platziert, ist es unwahrscheinlich, dass in Lücken zwischen den Polygonen später noch etwas passend platziert werden kann. Liegen also schnittfreie Stellungen für k Polygone vor, werden diejenigen Stellungen für weitere Iterationen mit $k+1$ Polygonen verwendet, deren Konvexe Hülle minimal ist. Die hiermit verfolgte Strategie ist es also, von Aussen an einen Polygon-Patch weitere Polygone so anzulegen, dass diese eng anliegen. Ideal ist es, wenn neue Polygone in die bisherige Stellung schnittfrei so eingefügt werden können, dass sie in lokale konkave Stellen anderer Polygone passen, und somit die Konvexe Hülle des gesamten Polygon-Patches gar nicht oder nur wenig erweitern.

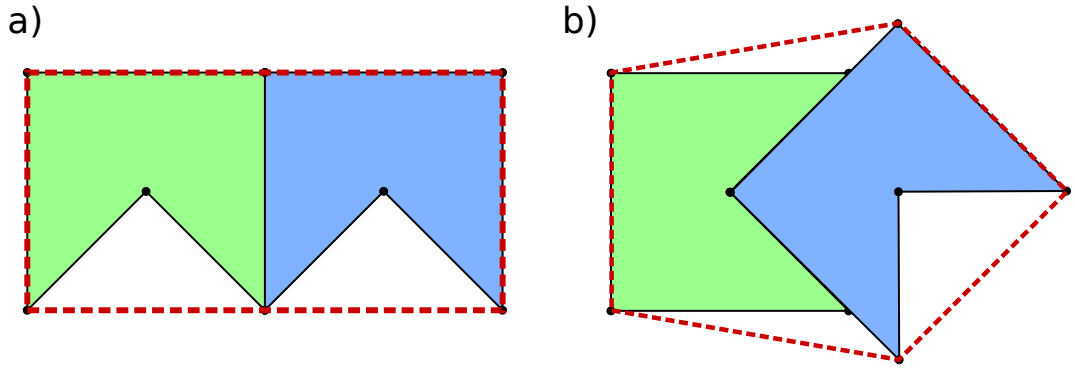

Abbildung 4.3: Zwei Stellungen von Polygonen. Die Fläche der Konvexen Hülle in *b)* fällt minimal kleiner aus als in *a)*

Da das Gebiet $\mathcal{G}$ rechteckig ist, ist es vielversprechend solche Konstellationen zu bevorzugen, deren Form möglichst rechteckig ist. Während die Konvexe Hülle für verschiedene Stellungen den gleichen Flächeninhalt aufweisen kann, sind rechteckige Stellungen mit minimaler Konvexer Hülle interessant, um global eine einfache Lösung zu erzeugen. Hierfür wird eine rechteckig Stellung einfach anhand des minimalen umfassenden Rechtecks dicht auf das Gebiet $\mathcal{G}$ gepackt, ohne einzelne Vertices der Polygone betrachten zu müssen.

Für den Flächeninhalt $\mathrm{area}(\mathrm{mr}(P^{\mathcal{P}}))$ eines minimalen Rechtecks für einen Polygon-Patch gilt, dass er mindestens so groß ist wie der Flächeninhalt $\mathrm{area}(\mathrm{kh}(P^{\mathcal{P}}))$ der Konvexen Hülle:

$$\mathrm{area}(\mathrm{mr}(P^{\mathcal{P}})) \geq \mathrm{area}(\mathrm{kh}(P^{\mathcal{P}}))$$

Es ist also sinnvoll solche Stellungen zu suchen, deren Konvexe Hülle möglichst dicht im minimalen Rechteck liegt. Abbildung 4.4 zeigt ein Beispiel für zwei Stellungen, von denen die bessere mittels des minimalen Rechtecks erkannt werden kann. Als Maß wird folgendes Verhältnis der Flächeninhalte verwendet:

$$\frac{\text{area}(\text{mr}(P^{\mathcal{P}}))}{\text{area}(\text{kh}(P^{\mathcal{P}}))} \geq 1$$

Polygon-Patches, die eine rechteckige Form aufweisen, minimieren dieses Verhältnis.

Anhand der beiden Heuristiken werden schnittfreie Polygon-Stellungen bewertet und nur solche als Polygon-Patches für weitere Nestings in Betracht gezogen, deren Konvexe Hülle minimal ist, und die gleichzeitig das minimale Rechteck maximal ausfüllen. In der Software-Implementierung zu dieser Arbeit zeigt sich, dass durch diese Heuristiken eine Auswahl vielversprechender Teillösungen sichergestellt wird.

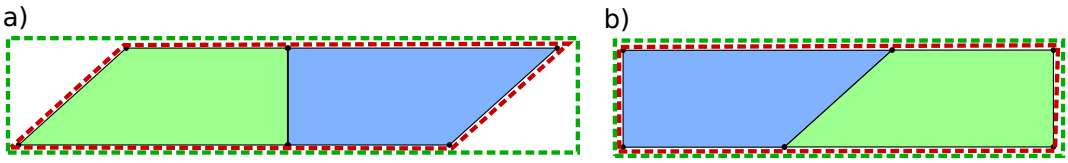

Abbildung 4.4: Die Fläche der Konvexen Hülle (rot) ist in *a)* und *b)* gleich, aber *b)* passt enger in das eigene minimale Rechteck (grün). Stellung *b)* ist deswegen für ein direktes Füllen der rechteckigen Domäne besser geeignet.

4.3 Einschränkung auf relevante Vertices

Komplexe Polygone werden durch viele Vertices dargestellt. Abhängig von der Anzahl an Vertices sind jedoch quadratisch viele Konstellationen möglich, die für ein diskretes Nesting von zwei Polygonen betrachtet werden müssen. Schon für verhältnismäßig kleine Polygone mit wenigen Vertices ist somit die Anzahl möglicher Stellungen sehr groß. Erzeugte Stellungen müssen darauf geprüft werden, ob sich die beteiligten Polygone schneiden, und der Schnitttest ist trotz vieler Optimierungen sehr zeitaufwändig. Ohne auf andere Varianten von Schnitttests auszuweichen, bleibt nur die Möglichkeit die Anzahl anfallender Testfälle zu minimieren, indem bei der Testerzeugung auf Eigenschaften der Polygone eingegangen wird. Hierfür sind vor allem die Vertices der Polygone relevant, und im Folgenden werden Strategien erläutert, wie unter Betrachtung von Vertices unnötige Schnitttests vermieden werden können. Dies führt dazu, dass man für Polygone in Polygon-Patches eine Datenstruktur von Vertices mitführt, in der gültige, also *valide* Vertices gespeichert werden. Nur solche validen Vertices müssen für zu generierende Stellungen betrachtet werden, wodurch viel überflüssige Arbeit entfällt.

4.3.1 Winkelsummen an Vertices

Ein erster Ansatz überflüssige Stellungen zu erkennen ist das Betrachten der Innenwinkel zusammengelegter Vertices. Werden zwei Polygone an Vertices zusammen gelegt, so kann diese Stellung nur

dann schnittfrei sein, wenn die Polygone sich nicht schon lokal an den beteiligten Vertices schneiden. An jedem Vertex lässt sich ein Innenwinkel abmessen, der angibt, in welchem Winkel die beiden angrenzenden Segmente das Innere des Polygons einschliessen. Wenn zwei Vertices von Polygonen zusammen gelegt werden, darf die Summe dieser Innenwinkel nicht mehr als $360°$ (oder 2π) betragen, da sich die Polygone sonst lokal überschneiden. Bei der Generierung von Stellungen muss also geprüft werden ob die Winkelsummen einen Schnitt implizieren, und die betrachtete Stellung wird gegebenenfalls nicht erzeugt.

Dies ist ein lokal eingeschränkter Ansatz und muss für jede anfallende Konfiguration neu berechnet werden. Dennoch können mithilfe dieses simplen Vorgehens viele unnötige Schnitttests eingespart werden.

4.3.2 Erkennen nicht erreichbarer Gebiete

Ein Polygon-Patch wird unter der Annahme erstellt, dass die beiden beteiligten Polygone (oder Polygon-Patches) in der engsten möglichen schnittfreien Stellung vorliegen. Das bedeutet, dass zwischen Teilpolygonen des Patches keine weiteren Polygone gesetzt werden können ohne Überschneidungen zu erzeugen. In einem Polygon-Patch gibt es also Vertices die *außen* liegen, an die weitere Polygone gelegt werden können, und Vertices die *innen* liegen, die nicht mehr erreichbar sind. Vertices die *innen* liegen können für die Generierung weiterer Stellungen somit ignoriert werden. Dies spart eine signifikante Anzahl an Stellungen, die aufgrund der Lage der Polygone sowieso Überschneidungen enthalten. Diese Annahme ist jedoch nur vertretbar, falls nur ein Polygon derselben Sorte platziert werden soll, oder alle gegebenen Polygone eine vergleichbare Größe haben. Für kleinere Polygone, die in Lücken eines Polygon-Patches passen könnten, würde ein solches Vorgehen eventuell gute Nestings verhindern, sofern nicht weitere Informationen über besagte Lücken zur Verfügung stehen. Eine solche detailierte Analyse von Polygon-Stellungen und enthaltenen Lücken überschreitet jedoch den Umfang dieser Arbeit.

Aus diesen Überlegungen heraus wird die Datenstruktur des Polygon-Patches so erweitert, dass für jedes enthaltene Polygon ein Array mitgeführt wird, das angibt, welche Vertices des Polygons *aussen* liegen. Da dies die Stellen am Polygon sind die für eine gültige Konfiguration in Frage kommen, werden diese Vertices fortan *valide Vertices* genannt. Abbildung 4.5 zeigt valide Vertices (grün) und nicht valide Vertices (rot) für Polygone.

Die Berechnung von validen Vertices wird mittels der Algorithmen für Punktverdeckung durch Polygone aus Kapitel 2.2.5 realisiert. Initial werden bei einem Polygon-Patch mit nur einem Polygon alle Vertices des Polygons darauf untersucht, ob diese von aussen sichtbar sind. Diese Sichtbarkeit wird interpretiert als Erreichbarkeit der Vertices bei Stellungen. Ist ein Vertex erreichbar, wird er im valide-Vertices-Array für dieses Polygon eingetragen. Für ein Polygon mit n Vertices benötigt die Berechnung valider Vertices $\mathcal{O}(n^2)$ Schritte, da für die Verdeckung eines Vertex alle durch Vertices definierten Segmente betrachtet werden müssen. Hat ein Polygon-Patch mehrere Polygone, werden für einen Vertex alle Polygone darauf geprüft, ob sie den Vertex verdecken. Sobald ein Vertex von einem Polygon, oder von allen Polygonen, komplett verdeckt wird, wird es von weiteren Berechnungen für Stellungen ausgeschlossen, indem es *nicht* im valide-Vertices-Array eingetragen wird. Haben alle

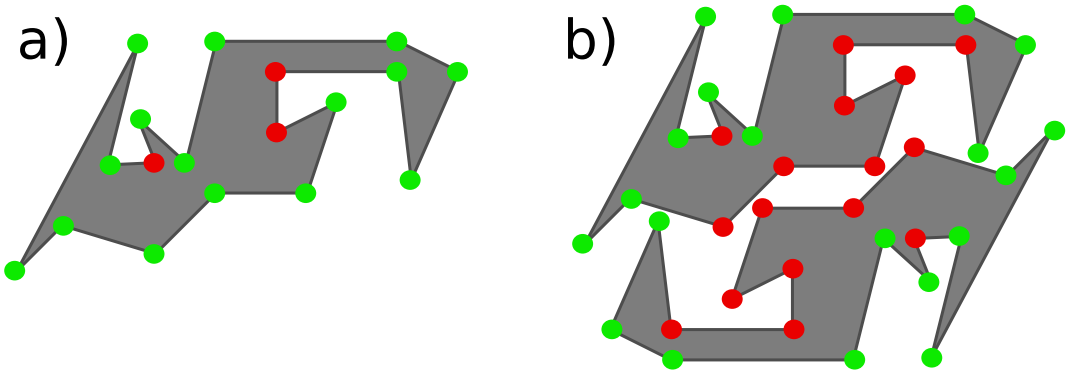

Abbildung 4.5: a) Valide Vertices (grün) in einem Polygon. **b)** Valide Vertices (grün) in einem Polygon-Patch. Eigen- und durch andere Polygone verdeckte Vertices (rot) werden von weiteren Berechnungen ausgeschlossen.

im Patch vorhandenen Polygone insgesamt n Vertices, benötigt die Verdeckungsberechnung aller validen Vertices ebenfalls $\mathcal{O}(n^2)$ Schritte.

Bei der Generierung von neuen Stellungen und Testfällen werden nun nur noch valide Vertices von Polygonen in einem Polygon-Patch verwendet. Valide Vertices müssen nur einmal bei der Erstellung eines Polygon-Patches berechnet und gespeichert werden. Wird ein Polygon-Patch mit einem weiteren zusammengelegt, werden die bisherigen validen Vertices aller beteiligten Polygone übernommen und in der neuen Konstellation darauf geprüft, ob sie verdeckt werden. Vor allem bei großen Polygon-Patches mit vielen Polygonen ist dieses Vorgehen von Vorteil, da so nur Vertices am Rand des Patches für weitere Berechnungen herangezogen werden.

Gemeinsam mit dem Erkennen zu großer Winkelsummen an Vertices bieten valide Vertices ein effektives Mittel, um die quadratisch vielen Stellungen, und daher quadratisch viele Schnitttests, auf nur tatsächlich sinnvolle zu begrenzen.

4.4 Vermeiden unnötiger Schnitttests

Eine weitere Möglichkeit unnötige Polygon-Stellungen und Schnitttests zu vermeiden ist das Ausnutzen struktureller Eigenschaften von Polygon-Patches. Neben der Betrachtung symmetrischer Eigenschaften von erzeugten Stellungen können vor allem Informationen aus alten Testfällen benutzt werden, um Voraussagen über neu anfallende Schnitttests zu treffen. Dies führt zu einer erheblichen Reduzierung von erstellten Polygon-Stellungen.

4.4.1 Redundante symmetrische Polygon-Stellungen

Für Polygone und Polygon-Patches die nicht mit einem strukturell unterschiedlichen, sondern mit einem identischen Polygon(-Patch) zusammengelegt werden sollen, ist die Abschätzung möglicher Stellungen aus Kapitel 4.1 zu hoch. Beim Erstellen diskreter Stellungen für zwei identische Polygone

muss nur rund die Hälfte der Stellungen betrachtet werden, die für zwei unterschiedliche Polygone anfallen würden. Gegeben ein Polygon P_n mit nummerierten Vertices, so müssen nur Vertices v_i, v_j ($i, j \leq n$) des Polygons aneinander gelegt werden, bei denen $j \leq i$ gilt, und für jedes Vertex-Paar gibt es zwei Ausrichtungen der anliegenden Segmente.

for $i = 1 \ldots n$ **do**
  for $j = 1 \ldots i$ **do**
    for $c = -1, 1$ **do**
      Speichere Polygon-Stellung $(i,\ j,\ c)$
    end for
  end for
end for

Dies führt zu

$$2 \cdot \sum_{j=1}^{n} j = 2 \cdot \frac{n \cdot (n+1)}{2} = n \cdot (n+1)$$

Stellungen, die auf Schnitte getestet werden müssen. Im Vergleich zur ursprünglichen Abzählung von $n \cdot m \cdot 2$ (hier $m = n$) Stellungen ist dies gerade die Hälfte.

Diese Einsparung lässt sich auch auf Polygon-Patches übertragen, die mit sich selber gepaart werden sollen. Eine Stellung wird dann nur weiterverwendet, wenn sie durch folgende Funktion $AkzeptiereStellung$ akzeptiert wird.

$$(P^{\mathcal{P}} \text{ Polygon P ID, P Vertex ID, } P^{\mathcal{P}} \text{ Polygon Q ID, Q Vertex ID, Segment-Kombination})$$

$$= (poly\_p\_id,\ v\_id,\ poly\_q\_id,\ w\_id,\ c) = s$$

$$\text{AkzeptiereStellung}(s) = \begin{cases} poly\_p\_id < poly\_q\_id, & \text{akzeptiere} \\ (poly\_p\_id = poly\_q\_id) \text{ und } (v\_id \leq w\_id), & \text{akzeptiere} \\ sonst, & \text{verwerfe} \end{cases}$$

4.4.2 Alte Schnitttests wiederverwerten

Eine Weiterführung des Gedankens, beim Kombinieren äquivalenter Polygon-Patches redundante Stellungen zu filtern, führt zu dem Ergebnis, anhand der Zusammensetzung eines Polygon-Patches einmal bereits erkannte Überschneidungen nicht wieder zu erzeugen. Dies hat auch mit der Einsicht zu tun, dass vor allem bei großen Polygon-Patches, selbst unter Benutzung von validen Vertices, ein großer Teil erzeugter Stellungen Überschneidungen aufweist. Die Erkennung dieser Überschneidungen kostet Rechenzeit, und die gewonnene Information über die Überschneidung sollte wieder verwertet werden. Dies ist aufgrund der hierarchischen Zusammensetzung von Polygon-Patches umsetzbar.

Hierfür werden für zwei Polygon-Patches $A^{\mathcal{P}}$ und $B^{\mathcal{P}}$ alle Stellungen gespeichert, die schnittfrei sind. Zusätzlich ist es nötig, jedem im Verlauf des Programms erzeugten Polygon-Patch eine eindeutige Identifikationsnummer zu geben. Die Tupel für gültige Stellungen werden gemeinsam mit den eindeutigen IDs der beteiligten Polygon-Patches abgespeichert und können wiederverwendet werden, wenn wieder genau diese beiden Polygon-Patches miteinander kombiniert werden sollen.

Ein Polygon-Patch $P^{\mathcal{P}}$ entsteht hierarchisch durch die Kombination zwei weiterer Polygon-Patches $A^{\mathcal{P}}$ und $B^{\mathcal{P}}$.

$$P^{\mathcal{P}} = A^{\mathcal{P}} \cup B^{\mathcal{P}}$$

Für eine formale Betrachtung werden $A^{\mathcal{P}}$ und $B^{\mathcal{P}}$ fortan Eltern-Patches $P^{A\mathcal{P}}$ und $P^{B\mathcal{P}}$ von $P^{\mathcal{P}}$ genannt.

$$P^{\mathcal{P}} = A^{\mathcal{P}} \cup B^{\mathcal{P}} = P^{A\mathcal{P}} \cup P^{B\mathcal{P}}$$

Sollen zwei vorhandene Polygon-Patches $P^{\mathcal{P}}$ und $Q^{\mathcal{P}}$ zu einem neuen zusammen gelegt werden, werden alle enthaltenen Polygone der Patches miteinander kombiniert. Das bedeutet implizit, dass die jeweils beteiligten Eltern beider Polygon-Patches miteinander kombiniert werden. Formal kann dies wie folgt ausgedrückt werden

$$\begin{aligned} P^{\mathcal{P}} \times Q^{\mathcal{P}} &= (P^{A\mathcal{P}} \cup P^{B\mathcal{P}}) \times (Q^{A\mathcal{P}} \cup Q^{B\mathcal{P}}) \\ &= P^{\mathcal{P}} \times (Q^{A\mathcal{P}} \cup Q^{B\mathcal{P}}) \ &= [P^{\mathcal{P}} \times Q^{A\mathcal{P}}] \cup [P^{\mathcal{P}} \times Q^{B\mathcal{P}}] \\ &= (P^{A\mathcal{P}} \cup P^{B\mathcal{P}}) \times Q^{\mathcal{P}} \ &= [P^{A\mathcal{P}} \times Q^{\mathcal{P}}] \cup [P^{B\mathcal{P}} \times Q^{\mathcal{P}}] \\ &= [P^{A\mathcal{P}} \times Q^{A\mathcal{P}}] \cup [P^{A\mathcal{P}} \times Q^{B\mathcal{P}}] \cup [P^{B\mathcal{P}} \times Q^{A\mathcal{P}}] \cup [P^{B\mathcal{P}} \times Q^{B\mathcal{P}}] \end{aligned} \tag{4.1}$$

Hierbei bedeutet $\times$ alle Stellungen zwischen zwei Polygon-Patches zu generieren, und $\cup$ kann als (Mengen-)Vereinigung verstanden werden, einerseits zwischen Polygon-Patches untereinander und andererseits zwischen generierten Mengen von Stellungen. Die mit eckigen Klammern umschlossenen Terme sind Kombinationen von Eltern-Patches (und Eltern- und Kind-Patches), die eventuell bereits einmal berechnet wurden und worüber Informationen vorliegen können. Wenn für zwei Polygon-Patches $P^{\mathcal{P}}$ und $Q^{\mathcal{P}}$ Stellungen generiert werden sollen, wird auf Ebene der Eltern-Patches implizit die letzte Zeile von Gleichung 4.1 umgesetzt. Dabei werden die Stellungen erzeugt, die durch die Terme in eckigen Klammern definiert werden. Die zweite und dritte Zeile von Gleichung 4.1 sind äquivalente Umformungen der letzten Zeile, in denen Informationen zwischen Polygon- und Eltern-Patches genutzt werden können. In Abbildung 4.6 ist eine ungültige Stellung zweier Patches gezeigt, die unter Verwendung von Informationen über die Eltern-Patches vermieden werden kann.

An dieser Stelle lassen sich zuvor getätigte Schnitttests wiederverwerten. Genauer gesagt lassen sich gültige Stellungen wiederverwerten, wenn für zwei nun zu kombinierende Eltern-Patches bereits zu einem früheren Zeitpunkt Stellungen erzeugt und von Überschneidungen bereinigt wurden. Diese gültigen Stellungen werden, gemeinsam mit den IDs für die dazugehörigen Patches, in einer Tabelle gespeichert. Sollen zwei Eltern-Patches miteinander kombiniert werden, wird anhand ihrer IDs in der Tabelle nachgeschlagen, ob bereits Stellungen für diese Kombination vorliegen. Liegen keine Stellungen vor, können für dieses Eltern-Paar keine zu generierenden Stellungen übersprungen werden. Also werden, wie bisher auch, alle möglichen Stellungen für dieses Eltern-Patch Paar erzeugt. Liegen für zwei zu testende Eltern-Patches in der Tabelle bereits Stellungen vor, werden diese für den aktuellen Kontext der beiden Patches wiederverwendet. Da Stellungen grundsätzlich als Tupel mit Polygon-IDs und Vertex-IDs gespeichert werden, müssen diese IDs aus dem Kontext, in dem sie *vorher* gestimmt haben, nun angepasst werden auf dem neuen Kontext der Eltern-Patches im Kind-Polygon-Patch. Dieser Kontext ist vor allem gegeben durch die Lage der Polygone im Polygon-Patch und ihre für jeden Patch intern neu vergebenen IDs, aber auch das Filtern von ehemals validen Vertices, die im Kontext des Kind-Polygon-Patches nicht zwingend valide sind. Ein solches Bereinigen

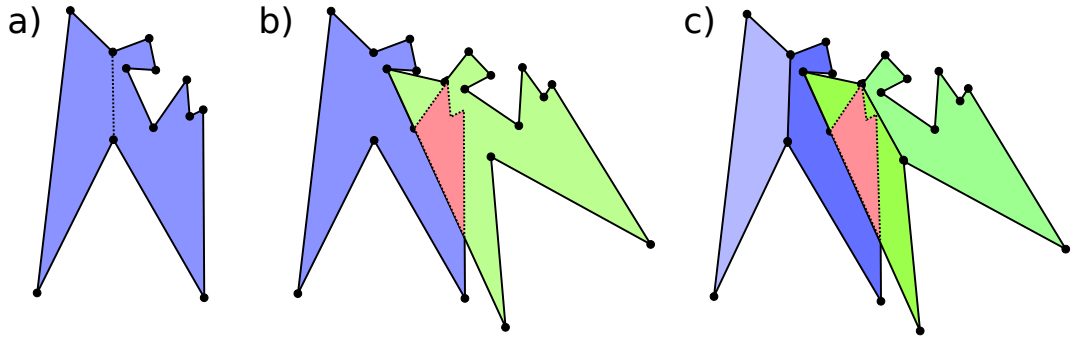

Abbildung 4.6: a) Ein Polygon-Patch $P^{\mathcal{P}}$ bestehend aus zwei Polygonen. Für die Teilpolygone wurden alle Stellungen betrachtet, bevor der Patch erzeugt wurde. **b)** $P^{\mathcal{P}}$ wird mit sich selber in einer Stellung platziert. Es entsteht ein Schnitt der über einen Schnitttest erkannt werden muss. **c)** Die sich schneidenden Teilpolygone der Patches wurden bereits einmal auf die ungültige Stellung getestet. Der anfallende Schnitttest ist vermeidbar, indem die alte Information genutzt wird.

von nicht mehr validen Vertices ist durch Sortieren und Abgleichen der gegebenen Stellungen und aktuell validen Vertices möglich.

Wurden alte Stellungen für Eltern-Patches auf diese Weise aufgearbeitet, stellen sie eine gültige Teilmenge der Stellungen für $P^{\mathcal{P}} \times Q^{\mathcal{P}}$ dar. Da die Eltern-Patches von $P^{\mathcal{P}}$ und $Q^{\mathcal{P}}$ aber nun in einer neuer Platzierung mit mehr Patches stehen, können ehemals gültige Stellungen zwischen Eltern-Patches nun ungültig sein. Dies ist aus Sicht eines Eltern-Patches in $P^{\mathcal{P}}$ der Fall, weil er sich mit neu hinzugefügten Eltern-Patches in $Q^{\mathcal{P}}$ überschneiden kann, auch wenn von genau einem Eltern-Patch in $Q^{\mathcal{P}}$ bekannt ist, dass keine Überschneidung damit vorliegt. Daher müssen auch die auf diese Weise gewonnenen und angepassten Stellungen wieder durch Schnitttests validiert werden, wie dies für Stellungen zwischen $P^{\mathcal{P}}$ und $Q^{\mathcal{P}}$ sowieso nötig ist. Da aber Überschneidungen auf Ebene der Eltern-Patches bereits vorher beseitigt wurden, fällt der Aufwand hierfür kein zweites mal an.

Im Laufe des Programms werden bei der Erstellung von Polygon-Patches also stets alle gültigen Stellungen mit den nötigen IDs der Patches in einer Tabelle gespeichert und können jederzeit wiederverwendet werden. Diese Tabelle kann als eine Art Bibliothek für gültige Stellungen zwischen bisher erstellten Patches angesehen werden. Unter einer guten Strukturierung der Stellungen und Patches sind bei p erstellten Polygon-Patches maximal

$$\frac{p \cdot (p+1)}{2}$$

Pakete mit Stellungen für Polygon-Patch Kombinationen zu speichern.

Diese Bibliothek mit Stellungen ist nicht zwangsläufig dicht befüllt, da nicht zwingend jeder Polygon-Patch mit jedem anderen kombiniert wird oder werden soll. Da auch stets neue Polygon-Patches erzeugt werden erscheint es wenig zielführend, die Bibliothek explizit zu füllen. Vielmehr sollte sie als Möglichkeit gesehen werden, altes Wissen über Stellungen praktisch wiederverwenden zu können.

Das hier beschriebene Absteigen im hierarchischen Aufbau eines Polygon-Patches ist theoretisch so weit möglich, dass die gesamte Entstehungsgeschichte eines Polygon-Patches in Form eines

Binärbaumes mitgeführt werden kann. Innerhalb dieses Binärbaumes ließen sich alle im Polygon-Patch enthaltenen Eltern-, Eltern-Eltern-, etc. auf ehemalige Überschneidungen mit Eltern eines weiteren Polygon-Patches prüfen. Dies würde jedoch eine nicht triviale Strukturierung, Suche und Abgleichung von gemeinsamen Eltern-Patches im Eltern-Binärbaum erfordern, weshalb dieser Gedanke in der Implementierung zu dieser Arbeit nicht weiter verfolgt wurde.

4.5 Finden einer globalen Lösung

Das Finden einer globalen Lösung bedeutet eine Platzierung von Polygonen auf der Domäne $\mathcal{G}$ zu finden, sodass möglichst viele Polygone enthalten sind. Mit den bisher entwickelten Methoden lassen sich enge Platzierungen von Polygonen in Form von Polygon-Patches realisieren. Unter Verwendung von validen Vertices und unter Wiederverwertung alter gültiger Stellungen wird die Anzahl teurer Schnitttests reduziert. Gleichzeitig kann mithilfe der vorgestellten Heuristiken eine Auswahl vielversprechender Polygon-Patches erfolgen, um die Komplexität der Lösungsfindung weiter zu reduzieren.

Für das Finden einer globalen Lösung wurden in dieser Arbeit zwei einfache Strategien evaluiert, die sich aus Polygon-Patches und deren Eigenschaften ergeben. Zur Vereinfachung wird angenommen, dass Instanzen von nur einem Polygon auf dem Gebiet platziert werden sollen. Müssten verschiedene Polygone platziert werden, lassen sich dieselben Strategien anwenden wie hier beschrieben, nur kommt hinzu, dass eventuell weitere Bedingungen an die Stückzahl der verschiedenen Polygone gestellt werden.

Brute-Force auf der Domäne

Es bietet sich ein iteratives Brute-Force Vorgehen an, stets Polygon-Patches um eine Polygoninstanz zu erweitern. Diese rundenbasierte Strategie erstellt für Runde i alle Polygon-Patches, die genau i Polygone beinhalten. Bei diesem Vorgehen fällt ohne einschränkende Heuristiken ein extrem hoher (Berechnungs-)Aufwand an. Es ist also nötig, nach jeder Runde nicht tatsächlich alle Polygon-Patches mit i Polygonen zu behalten, sondern nur solche, bei denen die Polygone am engsten zusammen liegen. Gleichzeitig werden unter diesen Patches diejenigen bevorzugt, deren Form am meisten einem Rechteck gleicht. Hierfür werden die in Kapitel 4.2.2 vorgestellten Heuristiken verwendet. Im Laufe der Lösungsfindung wird eine Tabelle von Polygon-Patches aufgebaut und stets erweitert. In dieser Polygon-Patch-Bibliothek können für eine gegebene Zahl k alle Polygon-Patches nachgeschlagen werden, die genau diese Anzahl an Polygonen beinhalten. Sollen also neue Polygon-Patch Kombinationen mit insgesamt i Polygonen generiert werden, werden in der Polygon-Patch-Bibliothek alle Patches mit k und j Polygonen benutzt, um Stellungen mit $i = j + k$ Polygonen zu erzeugen. Diese Stellungen werden anschließend in Schnitt-Tests umgesetzt und von Überschneidungen bereinigt. Aus gut bewerteten Stellungen werden neue Polygon-Patches erstellt und in die Polygon-Patches-Bibliothek eingetragen.

Die in Runde i neu erzeugten Polygone lassen sich aber auch einschränken, indem nicht alle Paare von Polygon-Patches betrachtet werden, die zusammen $i = j + k$ Polygone haben. Wird zum Beispiel k fest vorgegeben, beschränkt dies die Anzahl an möglichen Polygon-Patch Paaren, die kombiniert

werden können. So werden für eine aktuelle Runde i nur Polygone-Patches der letzten k Runden heran gezogen, und um maximal $j = i - k$ Polygone erweitert. Für $k = 1$ bedeutet dies zum Beispiel, dass Resultate aus der letzten Runde benutzt werden. Diese Resultate werden um genau ein Polygon so erweitert, dass dieses Polygon am engsten an das bisherige Polygon-Patch angelegt wird. Das ist ein auf die letzten k Runden beschränkter Brute-Force Ansatz.

Da das Gebiet $\mathcal{G}$ rechteckig ist, werden neu generierte Polygon-Patches anhand ihrer minimalen Rechtecke darauf getestet, ob sie in $\mathcal{G}$ platziert werden können. Hierfür müssen lediglich die Seitenlängen von $\mathcal{G}$ mit denen des minimalen Rechtecks verglichen werden. Polygon-Patches die eigentlich heuristisch gute Stellungen beinhalten, aber zu groß sind werden so erkannt und entfernt. Um unnötige Schnitttests für zu große Patches zu sparen, werden die korrespondierenden Stellungen *vor* den Schnitttests anhand ihrer minimalen Rechtecke heraus gefiltert. Dies spart besonders bei größer werdenden Patches Rechenzeit, wenn sie die Domäne bereits gut ausfüllen und weitere Polygone nicht beliebig platziert werden können. Der in Runden agierende Algorithmus terminiert, wenn *alle* neu erzeugten Polygon-Patches zu groß für $\mathcal{G}$ sind und wieder verworfen werden, oder bis eine angegebene maximale Anzahl an platzierten Polygonen erreicht wurde. Die Laufzeit dieses Vorgehens hängt also wesentlich von der Anzahl der pro Runde neu erzeugten Polygon-Patches ab.

Zusammensetzen einer Lösung anhand minimaler Rechtecke

Das iterative Aufbauen eines großen Polygon-Patches, der eine Lösung darstellt, erfordert viel Aufwand. Eine einfache Möglichkeit eine schnelle Lösung zu finden ist, einen kleineren Polygon-Patch anhand seines minimalen Rechtecks möglichst oft in $\mathcal{G}$ zu platzieren. Da das optimale Platzieren von Rechtecken verschiedener Größen selber ein NP-vollständiges Nesting-Problem darstellt[Kor03], wurde in dieser Arbeit nur das Platzieren eines stets gleich großen Rechtecks im Gebiet realisiert. Solche gleich großen Rechtecke eines Polygon-Patches können ohne Weiteres schnittfrei in die Domäne gelegt werden, und es bleibt nur noch zu zählen, wieviele Polygone hierdurch insgesamt in der Domäne platziert sind. Für dieses Vorgehen ist es sinnvoll Polygon-Patches zu bevorzugen, die sehr dicht in ihrem minimalen Rechteck liegen. Diese Forderung wird in die Sortierung und Auswahl von Polygon-Patches des rundenbasierten Verfahrens integriert. Von den Stellungen in denen die Polygone am engsten zusammen liegen werden also die bevorzugt, die ihr minimales Rechteck besser ausfüllen. Die in jeder Runde neu erzeugten Polygon-Patches werden dann anhand ihrer minimalen Rechtecke ausfüllend auf das Gebiet platziert und unter allen solchen Lösung einer Runde wird die Beste behalten. Da die hierdurch erzeugten Platzierungen auf der Domäne nicht zwangsläufig eng zusammen liegen (im Sinne von Kapitel 4.2.2), werden so gefundene Lösungen nicht für weitere Verbesserungen in Betracht gezogen. Es erscheint jedoch sinnvoll bei vielversprechenden Lösungen dieser Art wieder die zugrundeliegenden Polygon-Patches aufzugreifen und weitere Polygone hinzuzufügen unter der Bedingung, dass das minimale Rechteck nicht wesentlich größer wird.

5 Evaluation

Die in Kapitel 2.2 und 3 erläuterten und in Kapitel 4 entwickelten Methoden und Algorithmen wurden in einer prototypischen Anwendung implementiert und verifiziert. Die Implementierung ist in *C* geschrieben, nutzt die Laufzeitbibliotheken *OpenGL*, *GLEW (Version 1.10)* und *GLFW (Version 3)* und ist auf *GNU/Linux* Systemen lauffähig. Für den Schnitttest wird eine *OpenGL 4.2* kompatible Grafikkarte mit ebenfalls *OpenGL 4.2* kompatiblen Gerätetreibern vorausgesetzt. Für eine Verwendung unter *Windows* ist eine Anpassung von Betriebssystem-spezifischen Funktionen nötig, zum Beispiel Funktionen zur Zeitmessung.

Die implementierte Anwendung erwartet die Eingabe eines Polygons in Form eines Linienzugs und die Größe eines rechteckigen Gebietes. Über die in Kapitel 4.5 erläuterten Strategien zum Finden einer globalen Lösung wird versucht das Polygon möglichst oft im Gebiet zu platzieren. Hierfür werden rundenweise Polygon-Patches mit mehr Polygonen erstellt, bis keine Erweiterung von Polygon-Patches mehr möglich ist, die gleichzeitig noch in das Gebiet passen. Das Polygon kann durch den Benutzer als Linienzug gezeichnet, oder anhand einer Textdatei geladen werden. Über Eingabeparameter sind die Größe des Ausgabefensters, die Texturgröße der für den Schnitttest genutzten Texturen, und Einstellungen für das Nesting von Polygonen konfigurierbar. Da sich die Strategien zur Minimierung und Vermeidung von Schnitttests durchweg als vorteilhaft erwiesen haben, werden Polygon-Stellungen durchgehend nicht-optional unter Verwendung dieser Strategien erstellt. Über Parameter lässt sich einstellen, wieviele der am besten bewerteten Stellungen einer Runde für weitere Berechnungen verwendet werden sollen und wieviele Polygone pro Runde an einen Polygon-Patch hinzugefügt werden. Hierdurch sind vielfältige Möglichkeiten zur Auswahl von Lösungen und zur Reduzierung der nötigen Rechenzeit gegeben.

In Folgenden werden Resultate der implementierten Anwendung präsentiert. Hierfür werden Daten betrachtet, die über ausgewählte Probleminstanzen auf zwei verschiedenen Testsystemen erhoben wurden.

Testsystem **AMD** ist ein Notebook mit AMD E2-1800 APU, deren Dual-Core CPU mit maximal 1.7GHz taktet und 1MB L2 Cache besitzt. Die in der APU integrierte Grafikeinheit ist eine AMD Radeon HD 7340 deren Grafikspeicher mit maximal 512MB vom Hauptspeicher des Systems versorgt wird. Im System sind als Hauptspeicher 16GB DDR3-1333-RAM verbaut, die im Single-Channel-Modus betrieben werden. Als Betriebssystem ist GNU/Linux mit einem Arch Linux (Kernel 3.13) in Betrieb. Der verwendete Grafiktreiber ist der von AMD für Linux bereit gestellte Catalyst-Treiber 14.3.

Testsystem **NVI** ist ein Desktop PC mit einem AMD FX-4100 Quad-Core Prozessor der mit maximal 3.8GHz taktet und 4MB L2 Cache besitzt. Die verbaute dedizierte Grafikkarte ist eine NVIDIA GeForce GT 640 mit 2048MB Grafikspeicher. Im System sind 16GB DDR3-1333-RAM verbaut, die im Dual-Channel-Modus betrieben werden. Als Betriebssystem wird ebenfalls ein Arch Linux (Kernel 3.12)

verwendet. Als Grafiktreiber ist der von NVIDIA für Linux bereit gestellte Treiber in Version 331.20 in Betrieb.

Abbildung 5.1 zeigt drei Testinstanzen, bei denen verschiedene geometrische Formen auf kleinen Gebieten platziert werden sollen. Lösungen der drei Instanzen sind in Abbildung 5.2 zu sehen.

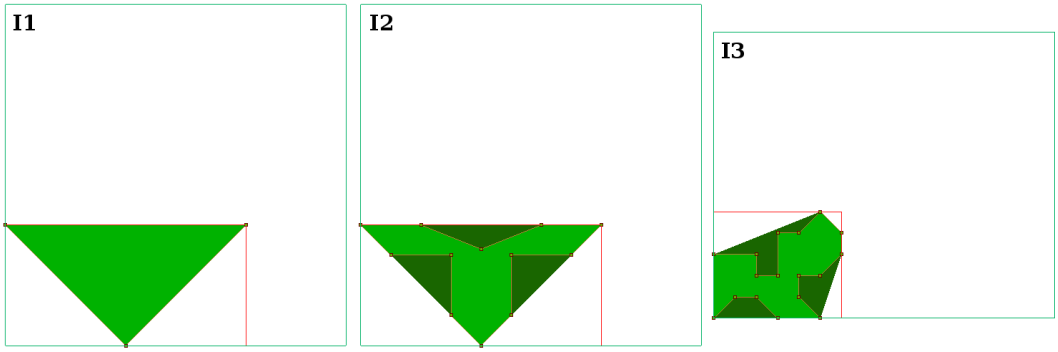

Abbildung 5.1: Drei Testinstanzen. Die Polygone (hellgrün) werden auf dem Gebiet (grünes Rechteck) möglichst oft platziert. Die Konvexe Hülle der Polygone ist dunkelgrün hinterlegt, das minimale Rechteck ist in rot eingezeichnet. **I1)** Ein rechtwinkliges Dreieck auf einem Gebiet, in dem es maximal acht mal platziert werden kann. **I2)** Wie I1 ein rechtwinkliges Dreieck, jedoch mit drei Aussparungen. Es kann maximal acht mal im Gebiet platziert werden. **I3)** Ein geometrisches Objekt mit 17 Vertices in einem Gebiet, in dem es vom Programm maximal sieben mal platziert werden konnte.

Zunächst wird der GPU-Schnitttest auf beiden Testsytemen evaluiert und die Performanz verglichen, die im wesentlichen von der Größe der intern verwendeten Bildauflösungen abhängt. Anschliessend werden Laufzeiten und der anfallende Rechenaufwand für jede Instanz betrachtet. Hierbei werden verschiedene Varianten des Programms betrachtet, bei denen unterschiedliche Beschleunigungstechnicken deaktiviert wurden. Zuletzt folgt eine Betrachtung von Instanzen auf großen Gebieten, auf denen der allgemeine Nesting Brute-Force Ansatz an seine Grenzen stößt.

Ergebnisse und Messungen der Implementierung

Der Schnitttest ist ein sehr wichtiger Faktor für die Laufzeit des Programms und für die Korrektheit der gefundene Lösung. Im Grunde wird fast die gesamte Laufzeit des Programms dafür aufgebracht Polygon-Stellungen auf Überschneidungen zu prüfen. Daher wird zunächst die Performanz des Schnitttests untersucht. Der Schnitttest ist maßgeblich durch die Auflösung seines rasterisierten Bildes konfigurierbar. Von dieser Auflösung hängt vereinfacht gesagt ab, wieviele (parallele) Recheneinheiten auf der Grafikkarte die Bearbeitung von Pixeln (Fragmenten) des Bildes realisieren müssen. Gleichzeitig ist die dem Schnitttest zugrunde liegende Bildauflösung maßgeblich für die Korrektheit dieses Tests. Ist die Auflösung zu niedrig gewählt, können Details der Polygone nicht entsprechend abgebildet werden, sodass Überschneidungen an diesen Details nicht erkannt werden. Tabelle 5.1

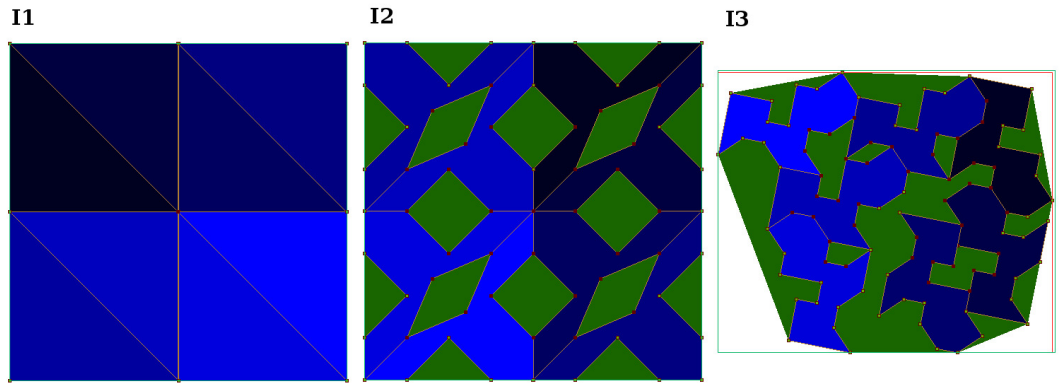

Abbildung 5.2: Lösungen der drei Testinstanzen. **I1)** Acht Dreiecke passen in das quadratische Gebiet. **I2)** Die Aussparungen ändern nichts an der dreieckigen Grundform des Polygons, weshalb ebenfalls nur acht solche Formen in das Quadrat platziert werden können. **I3)** Das Programm findet eine Lösung mit maximal sieben platzierten Formen. Das Lösen dieser Instanzen dauert auf beiden Testsystemen wenige Sekunden.

	Millisekunden pro Schnitttest					
	AMD			NVI		
Auflösung	I1	I2	I3	I1	I2	I3
1024	1.75	1.61	1.51	0.207	0.174	0.151
512	1.17	1.20	1.81	0.106	0.092	0.088
256	0.99	0.99	0.96	0.080	0.070	0.068
128	0.77	0.77	0.70	0.073	0.065	0.062
64	0.67	0.68	0.65	0.073	0.064	0.060
32	0.67	0.70	0.66*	0.072	0.062	0.059*

Tabelle 5.1: Messungen der durchschnittlichen Zeit für einen Schnitttest auf der GPU. Die Auflösung des quadratischen Bildes für den Schnitttest ist in Pixeln angegeben. Für die mit * markierten Einträge für I3 wurde vom Programm eine ungültige Lösung ausgegeben.

zeigt Messungen für die beiden Testsysteme AMD und NVI, auf denen unter verschiedenen Bildauflösungen des Schnitttests die vorgestellten Testinstanzen I1, I2 und I3 gelöst wurden. Gemessen wurde die durchschnittliche Zeit in Millisekunden, die ein Schnitttest auf dem System benötigt. Bei beiden Systemen skaliert die gemessene Zeit für einen Schnitttest nicht beliebig nach unten. Dies ist durch einen Overhead teils durch den CPU-Teil des Schnitttests zu erklären, kann aber auch mit genauen Implementierungsdetails im Bezug auf OpenGL und mit dem Grafiktreiber zusammenhängen. Eine weitere Optimierung des Schnitttests, um diesen Overhead weiter zu senken, kann in der Verwendung fortgeschrittener OpenGL-Techniken bestehen.

Für die genannten Ergebnisse wurde das Programm mit dem *GNU C Compiler 4.8* mit Optimierungsstufe O2 kompiliert. Tatsächlich spielt die Optimierungsstufe des Kompiliervorgangs aber keine

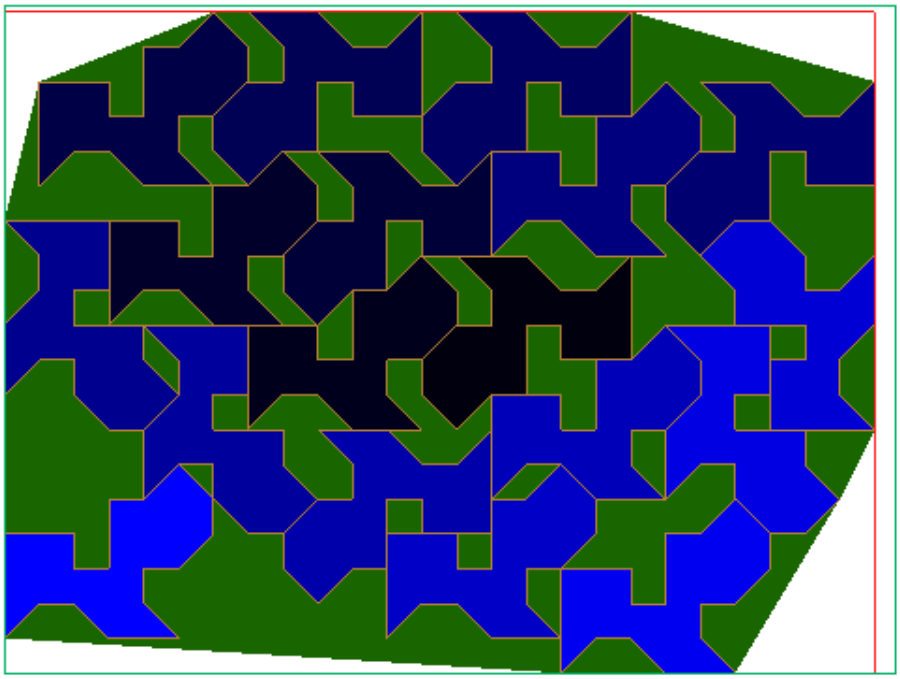

Abbildung 5.3: Eine größere Variante der Testinstanz I3. In knapp einer Minute konnten vom Programm 18 Polygone platziert werden. Die Berechnung wurde auf dem NVI System durchgeführt, die Schnitttest-Auflösung wurde auf 512 gestellt.

	Programm Laufzeit (große I3)	
Optimierung	AMD	NVI
O0	17m 26s 884ms	1m 25s 127ms
O2	17m 12s 357ms	1m 09s 500ms
O3	17m 03s 309ms	1m 09s 304ms

Tabelle 5.2: Laufzeiten der in Abbildung 5.3 dargestellten gelösten Instanz, abhängig von der Optimierungsstufe während des Kompiliervorgangs.

entscheidende Rolle für die Laufzeit des Programms. Dies ist in Tabelle 5.2 zu sehen. Gemessen wurden Ausführungszeiten für eine Abwandlung der Testinstanz I3 (siehe Abbildung 5.3) mit etwas größerem Gebiet, auf das 18 Polygone platziert werden konnten. Auf beiden Testsystemen wurden Varianten des Programms gestartet, die mit verschiedenen Optimierungsstufen kompiliert wurden. Die Bildauflösung für den Schnitttest wurde hier auf 512 gesetzt.

Die Unabhängigkeit der Programmlaufzeit von Compileroptimierungen deckt sich mit der Beobachtung, dass gemessene Laufzeiten vollständig mit Schnitttests zugebracht werden, die auf der GPU stattfinden.

Variante	deaktivierte Funktionen
V0	keine
V1	valide Vertices
V2	V1 + alte Tests wiederverwerten
V3	V2 + Stellungen vor Schnitttest auf Gebietsüberschreitung prüfen

	Anzahl erstellter Stellungen		
Variante	I1	I2	I3
V0	1 646	3 077	2 749
V1	1 715	4 459	4 194
V2	2 612	23 082	63 049
V3	6 528	147 162	146 608

Tabelle 5.3: Anzahl erstellter Stellungen während eines Programmdurchlaufs auf den drei vorgestellten Instanzen und unter Deaktivierung verschiedener Stellungen-reduzierender Funktionen.

Es wurden verschiedene Strategien entwickelt um nur tatsächlich nötige Polygon-Stellungen zu erzeugen. Dies hatte den Hintergrund, möglichst viele aus Stellungen folgende Schnitttests zu vermeiden. Um zu evaluieren, welchen Mehrwert diese Methoden zur Vermeidung von Stellungen haben, wurde die entwickelte Software in mehreren Varianten kompiliert. In den Programm-Varianten wurden jeweils verschiedene Funktionen deaktiviert, die die Anzahl generierter Stellungen reduzieren. Testläufe auf den vorgestellten Instanzen wurden durchgeführt, und alle im Programmverlauf erstellten Stellungen wurden gezählt. Die Ergebnisse hierzu sind in Tabelle 5.3 dargestellt.

Das Programm wurde stets so lange laufen gelassen, bis keine der neu erstellten Stellungen ein weiteres Polygon zum Nesting hinzufügen konnte. Für diesen Brute-Force Ansatz zeigt sich in Tabelle 5.3 deutlich, dass die in Variante V3 deaktivierte Funktion viele Schnitttests ersparen kann. Diese Funktion wurde in Kapitel 4.5 nur kurz erwähnt. Um nicht alle generierten Polygon-Stellungen an den Schnitttest zur Überprüfung geben zu müssen, werden für diese zunächst die minimalen Rechtecke erzeugt. Anhand dieser Rechtecke werden Stellungen entfernt, die nicht in das Gebiet passen, noch bevor sie einem Schnitttest unterzogen werden. Vor allem beim iterativen Nesting von Polygonen und wenn das Gebiet zu einem großen Teil bereits gefüllt ist werden so viele Schnitttests vermieden. Eine ähnliche Relevanz für die Anzahl erstellter Stellungen hat das Wiederverwerten alter Schnitttests. Vor allem für die komplexeren Polygone in I2 und I3 können extrem viele unnötige Stellungen, und somit Schnitttests, eingespart werden. Aus den erhobenen Daten wird geschlussfolgert, dass die entwickelten Methoden zur Aufwandsreduzierung sehr gut funktionieren.

Alle bisher betrachteten Testinstanzen wurden über den Brute-Force Ansatz berechnet, iterativ neue Polygone auf dem Gebiet zu platzieren. Hierbei werden in jeder neuen Runde i alle bisher erzeugten Polygon-Patches betrachtet und diejenigen Paare von Patches miteinander kombiniert, die zusammen genau $i = j + k$ Polygone haben. Da im Verlauf des Programms sehr viele Polygon-Patches erzeugt werden, sind mit jeder Runde mehr solcher Patch-Paare möglich, und der damit einhergehende Aufwand für Schnitttests nimmt enorm zu. Eine Möglichkeit diesen Aufwand zu beschränken ist,

bisher gefundene Polygon-Patches mehrmals im Gebiet zu platzieren. Eine andere Möglichkeit besteht darin, sich pro Runde auf maximal k neu hinzugefügte Polygone zu beschränken. Beschränkt man die Lösungsfindung zum Beispiel auf $k = 1$, ist eine erheblich geringere Laufzeit messbar, und es wird eine ähnlich gute Lösung gefunden wie beim Brute-Force Ansatz mit geschachtelten Patches. Das Ergebnis eines solchen Tests ist in Abbildung 5.4 zu sehen.

Der Speicherverbrauch bei Berechnungen mit einem nicht beschränkten Brute-Force steigt mit jeder Runde erheblich. Das ist einerseits auf die steigende Zahl neuer Polygon-Patches zurück zu führen, wird wahrscheinlich aber auch zu einem großen Teil durch das Speichern alter Stellungen verursacht. Beide Aspekte lassen sich abmildern, indem äquivalente Polygon-Patches, die dieselbe Platzierung von Polygonen darstellen, erkannt und gefiltert werden. Ebenfalls ist eine ausgereifte Strategie zur Einschränkung von neu erzeugten Polygonen denkbar, die sich sowohl auf den Speicherverbrauch, als auch auf den Berechnungsaufwand auswirken würde. Eine durchdachtere Strategie für das Speichern von alten gültigen Stellungen wird sicherlich auch zu einer Reduktion des Speicherverbrauchs führen.

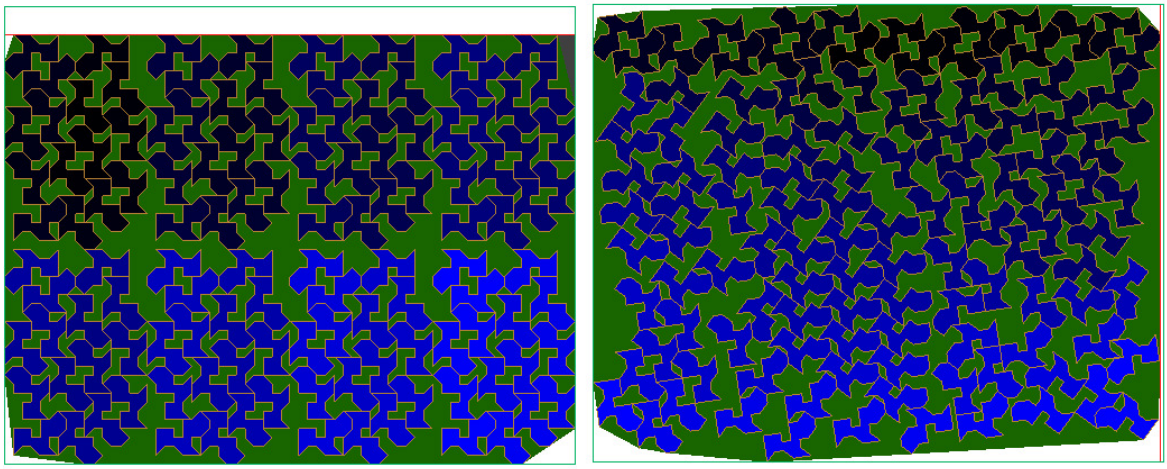

Abbildung 5.4: links) 120 Polygone konnten in 30 Minuten platziert werden. Tatsächlich wurden 8 Polygon-Patches mit jeweils 15 Polygonen im Gebiet anhand ihrer minimalen Rechtecke verteilt. Die Polygon-Patches wurden über unbeschränktes Brute-Force berechnet. Die Platzierung der minimalen Rechtecke ist sofort verfügbar. Während der 16. Runde des Programmdurchlaufs beendete das Betriebssystem den Prozess, weil es die gesamten Systemressourcen in Anspruch nahm (16GB RAM + 8GB Swap). **rechts)** Iterativ konnten in knapp 4 Sekunden 117 Polygone nacheinander platziert werden. Der Brute-Force Ansatz wurde beschränkt auf genau ein neues Polygon pro Runde. Der Speicherverbrauch belief sich auf knapp 60MB. Beide Tests wurden auf dem NVI System mit einer Schnitttest-Bildauflösung von 512 durchgeführt.

6 Fazit

Zusammenfassung

In dieser Arbeit wurde untersucht wie gegebene Formen möglichst oft in einem rechteckigen Gebiet platziert werden können. Die Problemstellung wurde durch Polygone modelliert, die möglichst eng zusammen liegen sollen. Statt beliebige Platzierungen zu erlauben, wurden nur Polygon-Stellungen betrachtet, die sich an den Vertices und Segmenten der Polygone orientieren. Verschiedene geometrische Eigenschaften wurden benutzt um Stellungen von Polygonen qualitativ zu bewerten, dies sind die Konvexe Hülle und das minimale Rechteck. Die hierfür nötigen Algorithmen wurden für häufig stattfindende Berechnungen in bestmöglicher Laufzeit-Komplexität umgesetzt und implementiert.

Um komplexe geometrische Berechnungen auf der CPU zu vermeiden wurde ein rasterbasierter Schnitttest für Grafikkarten realisiert und optimiert, der von technisch zeitgemäßen Funktionen der OpenGL-API Gebrauch macht. Mit Hilfe der dadurch bestehenden OpenGL-Anbindung wurde eine Visualisierung für Polygone und Platzierungen von Polygonen implementiert, die auch eine Maus basierte Eingabe von Polygonen durch den Benutzer zulässt.

Die hier getätigte Einschränkung auf diskrete Polygon-Stellungen hat zur Folge, dass sich zwei Polygone grundsätzlich in jeder erzeugten Stellung schneiden können. Um den hieraus resultiertenden Rechenaufwand für Schnitttests zu reduzieren wurden verschiedene Strategien betrachtet, wie die Anzahl erzeugter Stellungen verringert werden kann, ohne gleichzeitig die Qualität einer gefundenen Lösung zu beeinträchtigen. Die Datenstruktur der Polygon-Patches wurde eingeführt, um Stellungen von vielen Polygonen zu speichern und dauerhaft zu verwalten. Mithilfe der Polygon-Patches konnte eine Reduktion unnötiger Vertices bei der Erzeugung von Stellungen realisiert werden. Unnötige Vertices werden dadurch erkannt, dass sie im Polygon-Patch rundum durch Polygone verdeckt sind. Zusätzlich wurde eine Möglichkeit aufgezeigt die Information über getätigte Schnitttests wieder zu verwerten, indem gespeichert wird, welche Stellungen von Schnitttest als gültig erkannt wurden. Durch die hierarchische Struktur der Polygon-Patches wurde anhand gespeicherter gültiger Stellungen möglich, bei der Erzeugung neuer Stellungen zwischen Polygon-Patches solche Varianten zu vermeiden, die Überschneidungen von Polygonen enthalten.

Die entwickelten Strategien zum Nesting von Polygonen wurden in einer Software-Implementierung umgesetzt und evaluiert. Die entwickelte Software kann für gegebene Gebiete und Polygone Platzierungen der Polygone berechnen, anhand verschiedener Parameter sind der Schnitttest und die Strategie bei der globalen Lösungsfindung konfigurierbar. Das Laufzeitverhalten wurde anhand verschiedener Testinstanzen und Parameter-Konfigurationen untersucht. Die gemessenen Resultate entsprechen erwarteten Verbesserungen im Laufzeitverhalten, die durch die entwickelten Strategien erzielt werden sollten.

Limitierungen und mögliche Erweiterungen

Die in dieser Arbeit entwickelten Strategien wurden nur für hier so genannte diskrete Polygon-Stellungen betrachtet. Die eingangs getätigte Einschränkung auf diskreten Stellungen hat zur Folge, dass nicht tatsächlich alle möglichen Lösungen einer Probleminstanz betrachtet werden. Es liegt also nahe, die hier entwickelten Strategien auf beliebige Stellungen zu erweitern, bei denen Polygone tatsächlich beliebig orientiert und platziert sein können. Ein solche Ansatz würde aber wahrscheinlich zu der Nutzung bereits gut untersuchter Strategien führen, wie sie für das Lösen von Nesting-Problemen bereits entwickelt wurden[BO08].

Die hier entwickelten Methoden greifen den Aspekt auf, dass die Anzahl erzeugter Stellungen minimiert werden soll. Der Ansatz, alte gültige Stellungen über den Programmverlauf in einer Bibliothek zu bewahren und als Nachschlagewerk zu nutzen, lässt sich grundsätzlich ausbauen. Sofern eine sinnvolle Verwaltung gegeben ist, kann für Polygon-Patches der gesamte Eltern-Stammbaum in Form eines Binärbaumes betrachtet werden, um ungültige Stellungen in Teilbereichen des Patches zu vermeiden. Dieser Eltern-Stammbaum kann wiederum eingeschränkt werden auf Teile, die im aktuellen Patch relevant sind, also am äußeren Rand des Patches liegen.

Das Speichern alter Stellungen und das bisher nicht untersuchte Erzeugen äquivalenter Polygon-Patches führen zu einem enormen Speicherverbrauch. Diese beiden Aspekte hängen stark zusammen. Wenn über den gesamten Programmverlauf keine äquivalenten Polygon-Patches erzeugt werden, werden die nötigen Tabellen zum Speichern aller Polygone und Resultate von Polygon-Stellungen erheblich kleiner. In den Resultaten des entwickelten Programms finden sich momentan viele Polygon-Patches mit äquivalenten Stellungen, die mehrfach erzeugt wurden. Bereits eine besser durchdachte Auswahl von pro Runde neu erzeugten Polygonen könnte den aktuellen Speicherverbrauch gut beschränken, denn zur Bewertung und Sortierung von Polygon-Patches untereinander *nachdem* sie erstellt wurden sind keine Untersuchungen gemacht worden. Damit ist gemeint, dass nachdem in Runde i zwei Patches $A^{\mathcal{P}}$ und $B^{\mathcal{P}}$ kombiniert wurden und aus deren engster Stellungen ein neuer Patch $P^{\mathcal{P}}$ entstanden ist, dieser Patch $P^{\mathcal{P}}$ bisher in die Liste aller erzeugten Patches aufgenommen wird, ohne mit anderen in dieser Runde neu erzeugten Patches verglichen zu werden.

Unabhängig von der tatsächlichen Stellung der Polygone lässt sich der entwickelte OpenGL-Schnitttest weiter ausbauen. Hierfür können weitere in der Grafikprogrammierung übliche Verfahren untersucht werden, die auch mithilfe von OpenGL realisierbar sind. Über die genaue Wahl des Texturformates, eine weitere Reduktion von CPU-seitigen OpenGL-Aufrufen über spezielle OpenGL Funktionen, bis hin zu Optimierungen an verwendeten Shadern sind viele Verbesserungen denkbar. Diese grundsätzlich in der Grafikprogrammierung beheimateten Ansätze müssten für die Verwendung in einem Schnitttest genauer betrachtet und evaluiert werden.

Eigenständigkeitserklärung

Ich versichere, diese Arbeit selbstständig verfasst zu haben. Ich habe keine anderen als die angegebenen Quellen benutzt und alle wörtlich oder sinngemäß aus anderen Werken übernommene Aussagen als solche gekennzeichnet. Weder diese Arbeit noch wesentliche Teile daraus waren bisher Gegenstand eines anderen Prüfungsverfahrens. Ich habe diese Arbeit bisher weder teilweise noch vollständig veröffentlicht. Das elektronische Exemplar stimmt mit allen eingereichten Exemplaren überein.

Ort, Datum

Unterschrift

Literaturverzeichnis

[AI12] S. R. Allen, J. Iacono. Packing identical simple polygons is NP-hard. *CoRR*, abs/1209.5307, 2012. (Zitiert auf Seite 9)

[And79] A. Andrew. Another efficient algorithm for convex hulls in two dimensions. *Information Processing Letters*, 9(5):216 – 219, 1979. (Zitiert auf Seite 20)

[BO08] J. A. Bennell, J. F. Oliveira. The geometry of nesting problems: A tutorial. *European Journal of Operational Research*, 184(2):397 – 415, 2008. (Zitiert auf den Seiten 10 und 66)

[CE92] B. Chazelle, H. Edelsbrunner. An Optimal Algorithm for Intersecting Line Segments in the Plane. *J. ACM*, 39(1):1–54, 1992. (Zitiert auf Seite 31)

[Cha96] T. M. Chan. Optimal Output-Sensitive Convex Hull Algorithms in Two and Three Dimensions. *Discrete & Computational Geometry*, 16:361–368, 1996. (Zitiert auf Seite 20)

[EFK+05] F. Eisenbrand, S. Funke, A. Karrenbauer, J. Reichel, E. Schömer. Packing a trunk: now with a twist! In L. Kobbelt, V. Shapiro, Herausgeber, *Symposium on Solid and Physical Modeling*, S. 197–206. ACM, 2005. (Zitiert auf Seite 10)

[EFRS03] F. Eisenbrand, S. Funke, J. Reichel, E. Schömer. Packing a Trunk. In G. D. Battista, U. Zwick, Herausgeber, *ESA*, Band 2832 von *Lecture Notes in Computer Science*, S. 618–629. Springer, 2003. (Zitiert auf Seite 10)

[FS75] H. Freeman, R. Shapira. Determining the Minimum-area Encasing Rectangle for an Arbitrary Closed Curve. *Commun. ACM*, 18(7):409–413, 1975. (Zitiert auf Seite 23)

[Gho91] P. K. Ghosh. An Algebra of Polygons Through the Notion of Negative Shapes. *CVGIP: Image Underst.*, 54(1):119–144, 1991. (Zitiert auf Seite 10)

[GO06] M. A. Gomes, J. F. Oliveira. Solving Irregular Strip Packing problems by hybridising simulated annealing and linear programming. *European Journal of Operational Research*, 171(3):811–829, 2006. (Zitiert auf Seite 10)

[Kor03] R. E. Korf. Optimal Rectangle Packing: Initial Results. In E. Giunchiglia, N. Muscettola, D. S. Nau, Herausgeber, *ICAPS*, S. 287–295. AAAI, 2003. (Zitiert auf Seite 57)

[Mei75] G. H. Meisters. Polygons Have Ears. *The American Mathematical Monthly*, 82(6):pp. 648–651, 1975. (Zitiert auf den Seiten 25 und 26)

[Nie07] B. K. Nielsen. An Efficient Solution Method for Relaxed Variants of the Nesting Problem. In *Proceedings of the Thirteenth Australasian Symposium on Theory of Computing - Volume 65*, CATS '07, S. 123–130. Australian Computer Society, Inc., 2007. (Zitiert auf Seite 10)

[OF93] J. Oliveira, J. Ferreira. Algorithms for Nesting Problems. In R. Vidal, Herausgeber, *Applied Simulated Annealing*, Band 396 von *Lecture Notes in Economics and Mathematical Systems*, S. 255–273. Springer Berlin Heidelberg, 1993. (Zitiert auf den Seiten 10 und 31)

[OGF00] J. F. Oliveira, A. M. Gomes, J. S. Ferreira. TOPOS – A new constructive algorithm for nesting problems. *OR Spectrum*, 22(2):263–284, 2000. (Zitiert auf Seite 10)

[SA13] M. Segal, K. Akeley. The OpenGL Graphics System: A Specification (Version 4.4 (Core Profile)), 2013. http://www.opengl.org/registry/doc/glspec44.core.pdf. (Zitiert auf Seite 34)

[Tou83] G. Toussaint. Solving geometric problems with the rotating calipers, 1983. (Zitiert auf Seite 23)

Alle URLs wurden zuletzt am 1. 04. 2014 geprüft.