Problematik eines Puzzlespiels aus der Sicht der digitalen Bildverarbeitung

Christian Schafleitner

BAKKALAUREATSARBEIT

Nr. 03-1-0238-058-A

eingereicht am
Fachhochschul-Bakkalaureatsstudiengang

Medientechnik und -design

in Hagenberg

im Februar 2006

Diese Arbeit entstand im Rahmen des Gegenstands

Digital Imaging

im

Wintersemester 2005/06

Betreuer:

DI Dr. Wilhelm Burger

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen und Hilfsmittel nicht benutzt und die aus anderen Quellen entnommenen Stellen als solche gekennzeichnet habe.

Hagenberg, am 7. Februar 2006

Christian Schafleitner

Inhaltsverzeichnis

Erklärung iii

Vorwort vi

Kurzfassung vii

Abstract viii

1 Einleitung 1
- 1.1 Geschichtliches . . . 1
- 1.2 Motivation & Bedeutung . . . 1
- 1.3 Aufbau der Arbeit . . . 2

2 Lösungsansätze zum Puzzle Projekt 3
- 2.1 Voraussetzungen und Anforderungen an das Puzzle . . . 3
  - 2.1.1 Die Puzzleteile . . . 3
  - 2.1.2 Einschränkungen . . . 3
- 2.2 Der Algorithmus – ein Überblick . . . 5
  - 2.2.1 Erster Ansatz . . . 5
  - 2.2.2 The Big Picture . . . 5
  - 2.2.3 Kurzbeschreibung der einzelnen Schritte . . . 6
- 2.3 Analyse eines einzelnen Puzzleteiles . . . 7
  - 2.3.1 Finden der Eckpunkte und Ausrichtung . . . 7
  - 2.3.2 Analyse einer Kante . . . 11
  - 2.3.3 Aus- und Einbuchtungen . . . 11
- 2.4 Lösen des Puzzles – Matchingverfahren . . . 12
  - 2.4.1 Das erste Eckteil . . . 13
  - 2.4.2 Überprüfung zweier Kanten . . . 14
  - 2.4.3 Der Matchingfaktor . . . 14

3 Umsetzung des Puzzle Projekts 16
- 3.1 Verwendete Software . . . 16
  - 3.1.1 ImageJ . . . 16
  - 3.1.2 `PlugIn` Interface . . . 17

3.1.3 `ImagePlus` und `ImageProcessor` Objekt 17
3.2 Struktur des Plugins 18
3.2.1 `PuzzleProject_.java` 18
3.2.2 `Puzzle.java` 19
3.2.3 `PuzzlePiece.java` 19
3.2.4 `Edge.java` 19
3.2.5 `ResultPuzzle.java` 20
3.2.6 `PPLocator.java` 21
3.3 Anwendung 22
3.4 Tests 22
3.4.1 Grundlagen 22
3.4.2 Probleme beim Matching 24
3.4.3 Zeit und Speicherkomplexität 24
3.4.4 Resümee und Ergebnis 24

4 Konzepte zur Verbesserung 27
4.1 Scanning-Techniken 27
4.2 Rand bilden & *Fiducial Points* 28
4.2.1 Randteile suchen und vergleichen 28
4.2.2 Wahrscheinlichkeitsfaktor 29
4.2.3 *Fiducial Points* 29
4.2.4 Resümee 30
4.3 Übereinanderlegen der Teile 30
4.3.1 Visuelles Lösungsverfahren 30
4.3.2 Vergleich mit Robotor und Drucksensoren 31
4.4 *Isthmus Critical Points* 31
4.5 *Curve Matching* Verfahren 32
4.5.1 Vorteile 34

5 Zusammenfassung 35

A Inhalt der CD-ROM 37
A.1 Bakkalaureatsarbeit 37
A.2 `PuzzleProject`-Plugin 37
A.3 Testbilder 38
A.4 Ergebnisbilder 38

Literaturverzeichnis 39

Vorwort

Die Grundlage dieser Bakkalaureatsarbeit ist ein Projekt aus dem *Digital Imaging* Kurs im Sommersemester 2005. Nicht nur um mein Wissen im Bereich der digitalen Bildverarbeitung noch vor dem Berufspraktikum etwas aufzufrischen, sondern auch, weil ich schon während des Projektes im vergangenen Jahr von den Möglichkeiten, die uns die Technik der digitalen Bildverarbeitung bietet, fasziniert war, entschloss ich mich in diesem Bereich meine Bakkalaureatsarbeit zu schreiben.

An dieser Stelle möchte ich mich bei meinen Kollegen Markus Klopf und Dominik Steiner bedanken, mit denen ich dieses Projekt, welches die Grundlage für diese Arbeit bildet, im 4. Semester durchgeführt habe.

Weiters bedanken möchte ich mich bei meinen Bakkalaureatsarbeitsbetreuer und Studiengangsleiter Herrn Dr. Wilhelm Burger, der mir nicht nur bei Fragen zur digitalen Bildverarbeitung, sondern auch bei der Erstellung und Formatierung dieser Arbeit in LaTeX helfend zur Seite stand.

Kurzfassung

In dieser Bakkalaureatsarbeit wird ein automatischer Lösungsprozess eines Puzzles behandelt. Dabei werden die Konturen der einzelnen Puzzleteile analysiert und anhand dieser das Puzzle gelöst. Die Techniken digitaler Bildverarbeitung und menschlichen Denkens werden so kombiniert, um Probleme wie dieses zu lösen.

Die Algorithmen, welche zur Analyse des Bildes, zum Extrahieren der einzelnen Puzzleteile, zum Finden der Eckpunkte, sowie schließlich zum Lösen des Puzzles verwendet werden, sind in dieser Arbeit beschrieben. Der Prototyp der Anwendung wurde in Java als ImageJ Plugin implementiert.

Diese Arbeit stellt auch weitere Methoden und Lösungsverfahren vor, mit denen schon Puzzles gelöst wurden. Die unterschiedlichen Ergebnisse und auftauchenden Probleme werden diskutiert.

Abstract

This paper focuses on solving jigsaw puzzles using contour matching. The author discusses the developement of an application which solves problems such as solving puzzles in a way which can be compared with human cognition using methods of computer vision and digital imaging.

The algorithm described in the paper is based on a project completed at the Upper Austrian University of Applied Sciences Hagenberg. The algorithms used for analyzing the image, extracting pieces, detecting corners of puzzle pieces and finally solving a whole puzzle are described in this paper. The prototype of this application was written in Java and implemented as an ImageJ plugin.

This paper also discusses other methods and approaches to solve puzzles, as well as illustrating problems which may occur.

Kapitel 1

Einleitung

1.1 Geschichtliches

Schon 1964 beschäftigten sich H. Freeman und L. Gardner [FG64] mit der automatischen Lösung eines Puzzlespieles. Dabei wurde lediglich auf die Konturen der einzelnen Steine eingegangen. Von da an beschäftigten sich viele Autoren wissenschaftlicher Abhandlungen mit der Lösung dieses nicht ganz trivialen und faszinierenden Problemes.

1.2 Motivation & Bedeutung

Einem Computer „menschliches Sehen“ und „Vorstellungskraft“ beizubringen ist mitunter einer der Gründe, welche Informatiker in den Bann ziehen, dieses Problem, so höchst effizient wie möglich, zu lösen.

Das Puzzlelösungsproblem ist allerdings nicht nur eine Spielerei bzw. ein Wettkampf, wer die meisten Teile so schnell wie möglich lösen kann, sondern hat vor allem auch wissenschaftlich gesehen große Bedeutung. In der Archäologie, der Medizin, sowie in der Robotik und Automatisierungstechnik haben visuelle und logische Denkprozesse, die bisher nur von Menschen gelöst werden konnten, hohes Interesse in der heutigen Forschung geweckt.

Im Vorfeld müssen diese mühsamen und schwierigen Probleme genau analysiert werden, bevor damit begonnen werden kann, einem Computer beizubringen Probleme zu lösen, die bislang nur Menschen vorbehalten waren.

Zur Lösung dieses in erster Linie visuellen Problemes werden viele Grundlagen der digitalen Bildverarbeitung verwendet. Weiters sind auch ausgeklügelte Logik-Algorithmen notwendig um letztendlich ein ganzes Puzzle zusammen zu bauen.

1.3 Aufbau der Arbeit

Nach dieser Einführung in das Thema wird im nächsten Kapitel auf die Grundlagen eingegangen, die es ermöglichen sollten ein Puzzle zusammenzubauen, ehe in Kapitel 3 dann die Umsetzung und Implementierung als ImageJ Plugin Thema ist. Das Projekt führte ich mit zwei Kollegen im Rahmen des Gegenstandes „Digital Imaging“ im 4. Semester an der Fachhochschule Hagenberg durch.

In der Folge werde ich dann auf Verbesserungsansätze der präsentierten Lösung eingehen, sowie Probleme erläutern, welche nicht nur wir, sondern auch andere, die sich mit der automatischen Lösung eines Puzzles beschäftigten, hatten.

In Kapitel 4 wird dann noch auf andere und weiterführende Lösungsansätze eingegangen, ehe ich im Schlussteil dieser Arbeit ein Resümee über die vorgestellten Lösungsmöglichkeiten ziehen werde. Weiters wird auch ein Ausblick auf weitere Anwendungen derartiger Methoden gegeben.

Im Anhang ist eine Erläuterung zum Inhalt der beigelegten CD-ROM, auf der sich neben dieser Arbeit auch der Java-Quellkode des *Puzzle Projekts* befindet, zu finden.

Kapitel 2

Lösungsansätze zum Puzzle Projekt

In diesem Kapitel wird der in Kapitel 3 umgesetzte Lösungsansatz zum Lösen eine Puzzles erläutert. Die Implementierung erfolgt als ImageJ Plugin.

2.1 Voraussetzungen und Anforderungen an das Puzzle

2.1.1 Die Puzzleteile

Das Puzzle hat einen Rand, somit gibt es Eck- und Randteile. Die Eckteile verfügen über lediglich 2 Nachbarn, die Randteile weisen 3 Nachbarn auf. Alle anderen Teile haben genau 4 Nachbarn, die dieselben Eckpunkte aufweisen und somit eine annähernd gleichlange gemeinsame Kante besitzen.

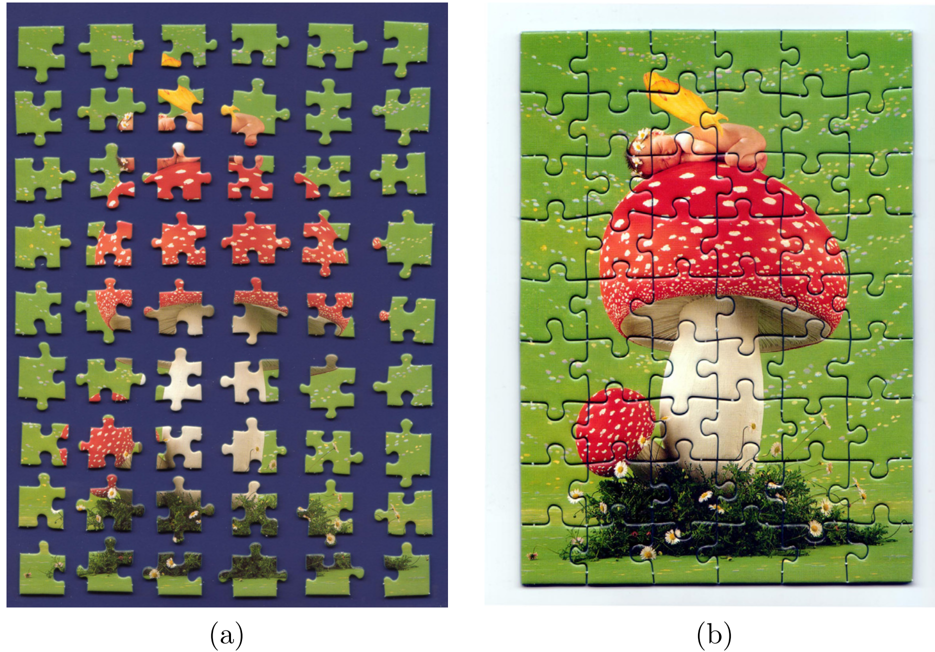
Die Ein- und Ausbuchtungen sollten möglichst markant und unterschiedlich ausgeführt sein. Da das Plugin das Puzzle nur anhand der Kontur versucht zu lösen, können Teile mit lauter gleichen Konturen nicht gelöst werden (Abb. 2.2(a)). Abbildung 2.2 (b) zeigt ein optimaleres Puzzle.

Die Teile können jedoch im Bild, welches dem Plugin am Start übergeben wird, beliebig angeordnet sein. Die Teile können beliebig rotiert und verschoben werden. In Abbildung 2.1 (a) sieht man eines der an das Plug-In übergebenen Testbilder, (b) zeigt den gewünschten Output des Plugins.

2.1.2 Einschränkungen

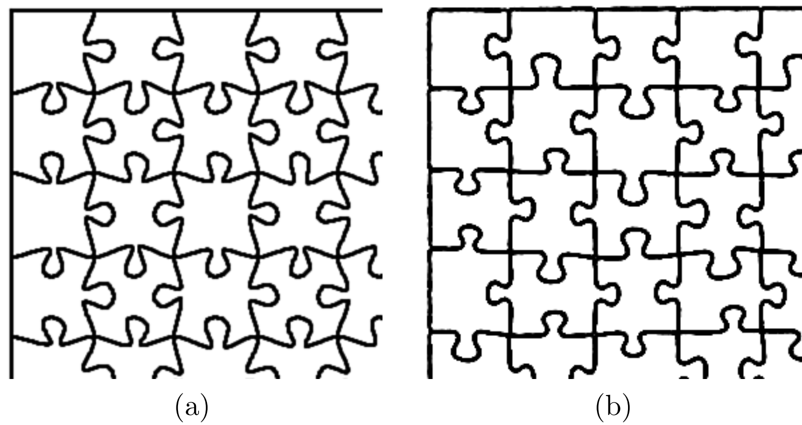
Folgende Kriterien muss ein Bild (siehe Abb. 2.1 (a)), welches an das Plugin übergeben wird, erfüllen:

- möglichst einheitlicher Hintergrund, im Idealfall schwarz oder weiß



(a) (b)

Abbildung 2.1: (a) Ausgangssituation, (b) gelöstes Puzzle.



(a) (b)

Abbildung 2.2: (a) Unbrauchbare Konturen, da die Teile gleich sind, bzw. es zu wenige Unterschiede in den Teilkonturen gibt. (b) Unterschiedliche Konturen, welche sich zum Lösen des Puzzles mit Hilfe von Konturmatching sehr gut eignen.

- die Teile dürfen nicht verzerrt sein, haben also das gleiche Größenverhältnis
- die Teile dürfen sich nicht überlappen
- die Eckpunkte sollten eindeutig erkennbar sein
- das Bild muss in einer möglichst hohen Auflösung vorliegen, sodass die Kantenlängen der einzelnen Puzzleteile mindestens 200 Pixel betragen

Je höher die Auflösung ist, umso mehr Testwerte können genommen werden und umso genauer kann auch das Matching erfolgen. Bei unseren Tests hat sich herausgestellt, dass es bei Kantenlängen unter 200 Pixel vor allem am Beginn Probleme mit der Zuordnung geben kann.

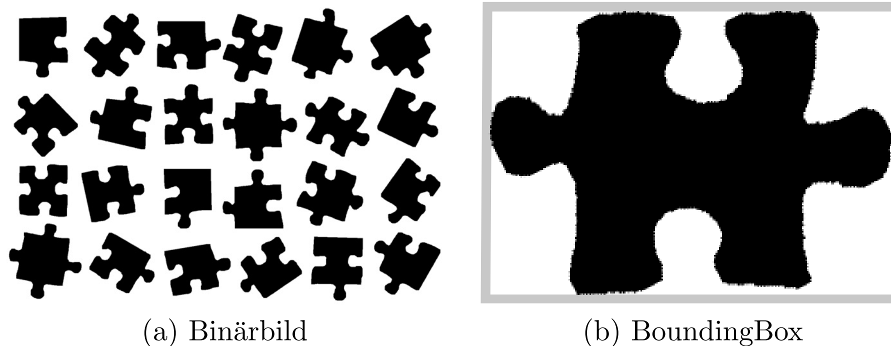
2.2 Der Algorithmus – ein Überblick

2.2.1 Erster Ansatz

Für jeden im Bild sichtbaren Baustein wird zunächst die umhüllende Kontur analysiert. Die Kontur wird durch Auffinden der Ecken in vier Teilkonturen zerlegt, die jeden Baustein beschreiben. Das Matching wird auf Basis dieser Teilkonturen durchgeführt, wobei zunächst der Abstand zwischen den Eckpunkten übereinstimmen muss). Die Kodierung der Teilkonturen könnte als Verlauf der Distanz der Kontur von der zwischen den Eckpunkten verlaufenden Geraden erfolgen.

2.2.2 The Big Picture

1. Binärbild erstellen
2. Puzzleteile markieren und Bounding Box finden
3. einzelnes Puzzleteil speichern
4. Eckpunkte finden und einzelnes Puzzleteil drehen
5. einzelne Kante speichern und Kontur analysieren
6. Eckpuzzleteil suchen und mit dem Vergleich aller Kanten mit allen Puzzleteilen (Matching) beginnen
7. am besten passendes Teil hinzufügen und mit nächstem Matching-Versuch beginnen



(a) Binärbild (b) BoundingBox

Abbildung 2.3: Binärbild und Boundingbox: Vom Binärbild ausgehend werden die Teile erkannt, extrahiert und analysiert. Die Boundingbox wird zum *Ausschneiden* eines Teiles berechnet.

2.2.3 Kurzbeschreibung der einzelnen Schritte

Schritt 1: Binärbild erstellen

Im ersten Schritt muss aus dem übergebenen Farbbild, welches in weiterer Folge nur mehr zur visuellen Ausgabe des Puzzles verwendet wird, ein Binärbild[1] erstellt werden.

Sollte kein eindeutiger schwarzer oder weißer Hintergrund vorhanden sein, wird ein Mittelwert (die erste Pixelzeile dient dabei als Referenz) aus den Farbtönen des Hintergrundes errechnet. Das Wegfiltern des Hintergrundes alleine reicht nicht aus, da bei einem gescannten Bild am Rand der einzelnen Teile dunkle Schatten geworfen werden. Diese werden in diesem ersten entscheidenden Schritt ebenfalls weggefiltert.

Da noch immer störende Pixel übrig bleiben, sowie die Kontur der Puzzleteile immer wieder durch fehlende einzelne Pixel unterbrochen wird, werden auf das schon erhaltene Binärbild noch mehrere Erosionen und Dilationen angewendet. Das Ergebnis dieser Arbeitsschritte ist in Abbildung 2.3 (a) zu sehen. Die schwarzen Puzzleteile befinden sich nun auf weißem Hintergrund.

Schritt 2: Labeling durchführen und Bounding Box

Um die einzelnen Puzzle Teile zu erkennen wird ein *Flood-Filling* Algorithmus [BB05, S. 195-206] angewendet. Dieser führt zugleich ein *Labeling*[2] der einzelnen Teile durch.

Dabei werden alle Pixel des Bildes durchlaufen: Sobald das aktuelle Pixel ein Schwarzes ist, also ein Puzzleteil darstellt, wird an dieser Stelle ein

[1] Ein Binärbild besteht nur aus 2 Werten, schwarz und weiß.

[2] Als *Labeling* bezeichnet man das Markieren und Durchnummerieren einzelner Regionen in einem Bild.

iterativer *Flood-Filling* Algorithmus ausgelöst, welcher alle Pixel dieses Teiles mit einer Nummer versieht. Im selben Programmschritt wird auch die Bounding Box (siehe Abb. 2.3 (b)) des Puzzleteiles berechnet.

Schritt 3: Einzelnes Puzzleteil speichern

Sobald das *Flood-Filling* mit einem Teil fertig ist, wird dieses Teil sofort mit Hilfe der errechneten Boundingbox herausgeschnitten und als Puzzleteil Objekt gespeichert. In dieser Klasse wird es weiter analysiert.

Schritt 4: Eckpunkte finden und Puzzleteil ausrichten

Für jedes gefundene Puzzleteil werden die Eckpunkte gesucht, sowie das Puzzleteil so ausgerichtet, dass eine Kante eine Parallele mit der y-Achse bildet.

Schritt 5: Kanten speichern und Kontur analysieren

Zwei Eckpunkte ergeben eine Seitenkante, deren Kontur analysiert und gespeichert wird.

Schritt 6: Eckpuzzleteil suchen und Matching beginnen

Nachdem alle Puzzleteile analysiert wurden, wird mit dem Matching begonnen. Zuerst wird ein Eckpuzzleteil gesucht und in das Lösungsraster gesetzt. Von diesem ausgehend, wird das am besten passende Teil auf der rechten Seite gesucht. Dieser Vorgang wird solange durchgeführt, bis wieder ein Eckteil gefunden wird. Nun wird in die nächste Zeile gesprungen und das Matching beginnt für diese Reihe. Die Breite des Puzzles steht nun fest. Das Puzzle wird also zeilenweise aufgebaut.

2.3 Analyse eines einzelnen Puzzleteiles

2.3.1 Finden der Eckpunkte und Ausrichtung

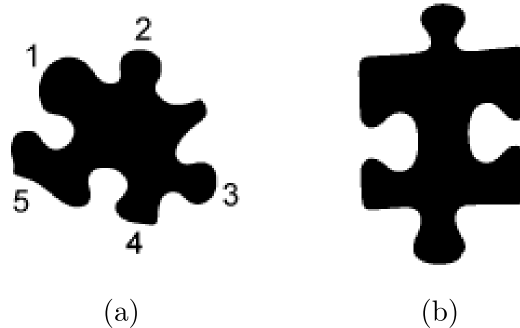
Das wohl Schwierigste beim gesamten Analyse-Prozess eines einzelnen Teiles ist es, die richtigen Eckpunkte zu finden.

Abbildung 2.5 zeigt in (a) wie problematisch es ist, die richtigen Stellen zu finden. Viele Aus/Einbuchtungen können falsch erkannt werden (Stellen 1, 4 und 5), bzw. werden viele Ausbuchtungen als Ecken identifiziert (2 und 3). Auch in (b) gibt es Probleme, allerdings ist hier die eindeutige Rechtecksform des „Hauptkörpers“ erkennbar.

Zu Beginn wurde versucht schon erprobte Corner-Detektoren, wie den *Harris-Corner-Detector* [BB05, Kapitel 8] einzusetzen. Da dieser *Corner-Detektor* mit unnötig hoher Genauigkeit arbeitet, die wir nicht benötigen

```
Gegeben: Binärbild eines Puzzleteiles
Gesucht: Eckpunkte der Kanten
Suchen der Eckpunkte
    1. Überprüfe alle Pixel:
         Für jedes Pixel überprüfe vier Quadranten
         in der Größe von 20 Pixel
         Wenn ein Quadrant schwarz und drei Quadranten weiß sind:
             Speichere Punkt als möglichen Eckpunkt
    2. Fasse ähnliche Eckpunkte zu einem Punkt zusammen
         Speichere ähnliche Eckpunkte in einer Liste
         Berechne Durchschnitt für je eine Region von Punkten
    3. Suche Punkte entlang der Diagonale
         von den 4 Ecken des Bildes ausgehend
Überprüfe die gefunden Punkte auf Gültigkeit
Wenn nicht gültig:
    Rotiere das Bild um 10° und starte den Vorgang erneut
Sonst:
    Eckpunkte erfolgreich gefunden
```

Abbildung 2.4: Pseudo-Code: Algorithmus zum Finden der Eckpunkte.



(a) (b)

Abbildung 2.5: Problematische Puzzleteile: (a) Eckpunkte sind nicht eindeutig erkennbar, (b) Eckpunkte deutlich erkennbar, aber die Ausbuchtungen bereite immer noch Probleme. Bilder aus [GMB02].

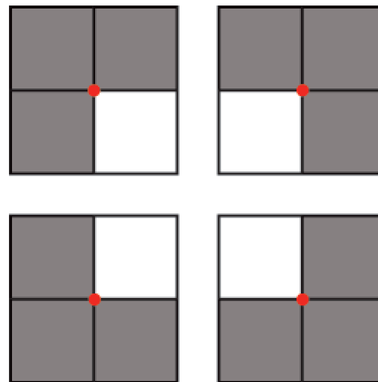


Abbildung 2.6: Ein Eckpunkt muss einem dieser 4 Patterns entsprechen.

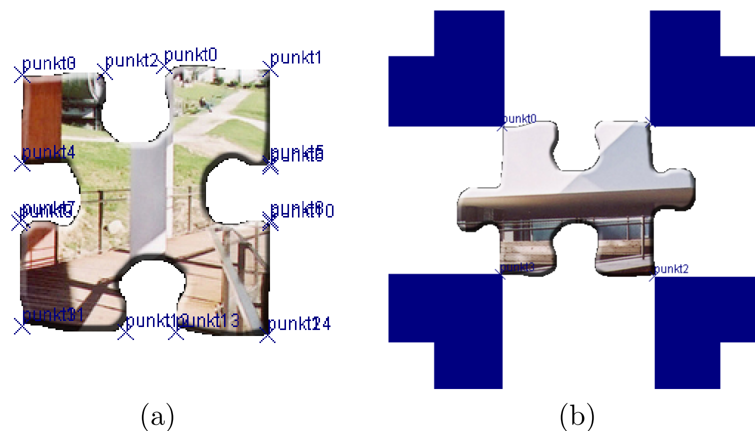


Abbildung 2.7: Eckpunkte finden: (a) gefundene Punkte nach Schritt 1, (b) gefundene Eckpunkte nach Schritt 3.

(er findet auch Punkte wenn sich nur wenige Pixel ändern und so ein 90° Winkel entsteht), erschien es einfacher einen eigenen *Corner-Detektor* für Puzzleteile zu implementieren.

Corner Detection für Puzzleteile

Der „Eckpunkte-Findungs-Prozess“ läuft in 3 Schritten ab:

1. **Schritt**
   Für jedes Pixel im Bild wird ein Bereich von 40×40 Pixel überprüft. Dieser Bereich wird in 4 Viertel mit je 20×20 geteilt, wobei einer dieser 4 Felder zu 94 % aus weißen und die anderen 3 Felder zu 94 % aus schwarzen Pixel bestehen müssen, um das aktuelle Pixel als möglichen Eckpunkt markiert wird. Diese *Patterns* sind in Abbildung 2.6

sichtbar.

Da es sich bei den Ecken eines Puzzleteiles es sich nur selten um 90° Winkel handelt, muss man einem Schwellwert (in diesem Fall 94 % weiße/schwarze Pixel) arbeiten, damit auch falsche Pixel zugelassen werden. In Abb. 2.7 (a) sieht man die so gefundenen Eckpunkte.

2. **Schritt**
   Wie man in Abbildung 2.7 (a) erkennen kann, findet das Programm durch die Zuhilfenahme eines Schwellenwertes an den Ecken mehrere Punkte. Durch eine Approximierung dieser einzelnen Punkte wird lediglich ein Punkt als tatsächlicher Eckpunkt dieser Ecke bestimmt.

3. **Schritt**
   Da auch an anderen markanten Stellen eines Puzzleteiles (z. B. an den Ausbuchtungen) ein 90° Winkel auftritt, müssen im letzten Schritt diese falschen Punkte aussortiert werden. In Abbildung 2.7 (a) werden die fälschlicherweise erkannten Punkte gezeigt.

   Bei diesem letzten Schritt wird davon ausgegangen, dass das Bild bereits richtig ausgerichtet ist und sich somit die Eckpunkte in den Diagonalen des Bildes befinden. Eine Abfrage über die Abstände zu den Ecken hat kein erfolgreiches Ergebnis geliefert, da es sehr häufig vorkommt, dass gefundene Punkte an den Ausbuchtungen den Ecken näher liegen als die tatsächlichen Eckpunkte eines Teiles.

   In Abbildung 2.7 (b) wurde der nach den Eckpunkten abgesuchte Bereich blau markiert und zeigt die nun gefundenen Eckpunkte.

Verdrehte Teile

Wie schon im 3. Schritt erläutert, wird immer von einem bereits ausgerichteten Puzzleteil ausgegangen. Daher liefert die oben beschriebene Methode bei verdrehten Teilen zwar ein Ergebnis, bei denen es sich aber aber nicht um die richtigen Eckpunkte handelt. Bevor die gefundenen Eckpunkte gespeichert werden, müssen diese bestimmte Kriterien erfüllen, um nicht falsifiziert zu werden:

- Mindestlänge einer Kante
- Diagonalen und gegenüberliegende Kanten
  müssen annähernd gleich lang sein
- Jeder Punkt muss auf einem eigenen Pixel liegen

Dadurch wird die „Quadratförmigkeit“ des Puzzleteiles bestätigt. Falls eine der oben gelisteten Bedingungen nicht zutrifft, wird davon ausgegangen,

dass das Teil verdreht vorliegt. Es wird sodann um 10° gedreht und die Eckpunkte werden erneut gesucht.

Die Eckpunkte müssen nicht genau in den Diagonalen liegen, eine Abweichung von ± 5° wird vom Algorithmus toleriert. Das Teil muss also maximal 9 mal (90°) gedreht werden, um die richtigen Eckpunkte zu finden.

Sobald gültige Eckpunkte vorliegen, müssen noch die Winkel der einzelnen Kanten zueinander bestimmt werden, um die erste Kante gerade zu richten, bevor die Analyse der einzelnen Kanten beginnt.

2.3.2 Analyse einer Kante

Die 4 Kanten eines jeden Teiles werden als eigenständig betrachtet und somit auch getrennt voneinander analysiert.

Bei der Analyse wird zuerst überprüft, ob es sich um ein Randteil oder um eine Aus-/Einbuchtung handelt. Beim Analysieren einer Kante werden entlang der gedachten Linie zwischen den beiden Eckpunkten die einzelnen Pixel abgetastet. Zuerst wird das Teil so gedreht, dass von einer komplett waagrechten Linie ausgegangen wird. Nun werden die Unterschiede von der gedachten Linie zum tatsächlichen Puzzlerand gespeichert. Das so erhaltene Profil gibt einen ersten Aufschluss über die Beschaffenheit einer Kante und teilt diese in 3 Kategorien ein.

- Rand
- Einbuchtung / weiblich
- Ausbuchtung / männlich

Randteile

Bei Randteilen wird der Analyseprozess sofort beendet, jedoch ist dies für die Klassifizierung eines Teiles extrem wichtig, da so Puzzleteile als Eck- bzw. Randteile markiert werden. Dies ist eine wichtige Grundlage für den Matching-Prozess.

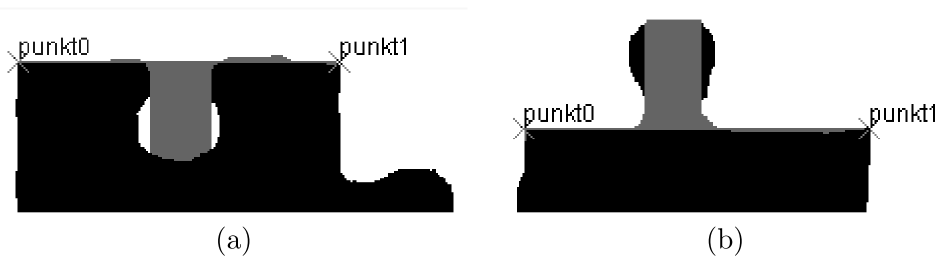
Der Analyse der Aus- und Einbuchtungen wird der nächste Abschnitt gewidmet.

2.3.3 Aus- und Einbuchtungen

Die Grundlage beim Matching ist bei diesem Lösungsansatz die geometrische Vermessung der Kante, bzw. der Aus- und Einbuchtungen im Besonderen.

Überblick

Folgende Informationen werden analysiert und gespeichert:



(a) (b)

Abbildung 2.8: Kantenanalyse einer (a) Einbuchtung (weiblich) und (b) einer Ausbuchtung (männlich)

- Länge der Kante zwischen den Eckpunkten
- Tiefe/Länge einer Aus-/Einbuchtung
- Abstand vom Eckpunkt zur Aus-/Einbuchtung
- Fläche einer Aus-/Einbuchtung

Abtasten einer Kante

Beim Analysieren einer Kante wird entlang der gedachten Linie zwischen den beiden Eckpunkten der Rand des Puzzleteiles abgetastet.

Abbildung 2.8 zeigt die Abtastwerte (grau) einer Ein-(a) und Ausbuchtung(b). Das so erhaltene Höhenprofil lässt nun auch auf den Beginn und das Ende einer Aus- und Einbuchtung schließen. Somit wird die Lage und der Abstand der Ein-/Ausbuchtung bestimmt. Weiters wird der Mittelpunkt der kritischen Region berechnet und von diesem ausgehend wird die Tiefe gemessen, sowie die Fläche mittels Flood-Filling errechnet.

2.4 Lösen des Puzzles – Matchingverfahren

Die erste Überlegung um zur richtigen Lösung des Puzzles zu gelangen, war einfach alle Möglichkeiten durchzuprobieren und somit diejenige Lösung, welche die geringsten Unterschiede zwischen den Kanten aufweist, als die richtige zu erkennen. Wie und ob zwei Kanten zusammenpassen wird später erläutert.

Beim Durchprobieren all dieser Lösungen stößt man sehr schnell an das Problem des Handelsreisenden (*Travelling-Salesman-Problem*[3]), welches

[3] Das Travelling Salesman Problem (TSP) [Lan01] besteht darin, daß ein Handelsreisender eine Rundreise durch n Städte (Entfernungen sind bekannt) unternehmen und dabei einen möglichst kurzen Weg zurücklegen soll. Es gibt also $(n-1)!$ Möglichkeiten, beginnend bei einer bestimmten Stadt alle anderen Städte zu besuchen und wieder zur Ausgangsstadt zurückzukehren. Dies sind zu viele, um alle durchzuprobieren.

```
Lösungsbild anlegen
Lösungsfeld anlegen, in dem die Teile gespeichert werden
Suche erstes Eckteil
Solange es freie Teile gibt:
    Aktuelle Spalte == 0:
        Speichere letzte obere Kante
        Suche nach passendem Randteil
    Sonst:
        Speichere letzte rechte Ecke
        Wenn rechte Ecke vorhanden und kein Randteil ist
            Aktuelle Reihe == 0:
                Suche passendes Randteil mit passender linker Kante
            Aktuelle Reihe != 0:
                Speichere letzte obere Kante
                Suche passendes Randteil mit passender linker
                                             und oberer Kante
    Wenn gültiges Teil gefunden wurde:
        Setze Teil an bestimmten Platz und zeichne dieses
        Verringere Anzahl der vorhandenen Teile
        Wenn rechte Kante ein Rand ist:
            Gehe in die nächste Reihe
        Sonst:
            Gehe zur nächsten Spalte
    Wurde kein gültiges Teil gefunden:
        Lösungsprozess wird abgebrochen
Puzzle wurde gelöst
```

Abbildung 2.9: Pseudo-Code: Matchingverfahren – allgemeiner Vorgang.

in [Lan01] beschrieben wird. Daher können nicht alle Teile durchprobiert werden, bis die am besten passende Lösung gefunden wurde.

Der im Folgenden beschriebene und angewendete Lösungsalgorithmus sucht von einer Stelle ausgehend, das an dieser Stelle am besten passendste Teil und fügt dieses an der aktuellen Stelle hinzu. Sobald ein Fehler gemacht wird, gibt es deshalb kein Zurück mehr. Darum muss schon bei der Analyse auf große Genauigkeit geachtet werden.

Der Pseudo-Code in Abbildung 2.9 beschreibt den Matching Vorgang im Allgemeinen.

2.4.1 Das erste Eckteil

Zu Beginn wird das zuerst gefundene Eckteil ausgewählt, richtig gedreht und an die erste Stelle links oben gesetzt.

Vom ersten Eckteil ausgehend wird versucht die erste Reihe zusammen zu bauen. Die Fehlerwahrscheinlichkeit ist hier besonders hoch, da noch alle Teile im *Teile-Pool* liegen. Eine Vergleichskante ist dabei fix: die obere Kante

muss ein Rand sein, die zweite Kante ist immer die rechte Kante des zuletzt gesetzen Teiles.

So werden nun alle passenden Kanten jedes vorhandenen Randteiles mit der rechten Kante des schon fixierten Eckteiles verglichen. Das Teil mit dem geringsten *Matchingfaktor*, welcher für alle in Frage kommenden Kanten errechnet wird, wird am Ende des Prozesses dem Eckteil hinzugefügt.

Dieser Vorgang wird solange wiederholt, bis wieder an ein Eckteil gesetzt wird. Sobald dieses gesetzt wird, ist die Breite des Puzzles festgelegt.

2.4.2 Überprüfung zweier Kanten

Nachdem die erste Reihe vollständig gesetzt wurde, wird mit der nächsten Reihe begonnen. Ab diesem Zeitpunkt kann jedes neue Teil mit 2 Kanten verglichen werden. Dazu wird auf die untere Kante des darüberliegenden Teiles referenziert, sowie auf den rechten „Kantennachbar“.

Schon verbaute Teile werden markiert. Auf diese kann später nicht mehr zugegriffen werden. Der oben beschriebene Vorgang wiederholt sich solange, bis eine der folgenden Bedingungen zutrifft:

- Es sind keine Teile mehr vorhanden — das Puzzle wurde vollständig (und somit richtig) gelöst.
- Es kann kein passendes Teil mehr gefunden werden, das den Grundkriterien (Einbuchtung verlangt Ausbuchtung, Ränder) entspricht — das Puzzle konnte nicht vollständig gelöst werden, die schon gesetzen Teile, sowie der gemachte Fehler, sind aber in der Ausgabe erkennbar.

2.4.3 Der Matchingfaktor

Wie schon erwähnt, wird für jede Kante im Teil ein Matchingfaktor errechnet, welcher die einzelnen Teile mathematisch vergleichbar macht.

Ausschlussbedingungen

Zuerst werden die beiden Kanten auf bestimmte Ausschlussbedingungen überprüft. Sollten die Längen der Kanten oder die Positionen und Längen der Aus-/Einbuchtungen auf keinen Fall zusammenpassen, wird der Vergleich abgebrochen und ein eindeutiger Matchingfaktor mit `Integer.MAX_VALUE` zurückgegeben.

Tests haben ergeben, dass diese Ausschlusskriterien mit bestimmten Toleranzwerten die Puzzleteile so weit aus filtern, sodass nur mehr ganz wenige für die genauere Betrachtung übrig bleiben, unter diesen sich aber das richtige Teil befindet.

Am Beginn des Matching-Verfahrens sind die Werte noch sehr streng. Sollte gar kein Teil gefunden werden, wird der Toleranzbereich dynamisch

erhöht und das Matching ein weiteres Mal durchgeführt, bis schließlich mindestens ein passendes Teil gefunden wird.

Unterschiedliche Gewichtungen

Nun bleiben also nur mehr Teile übrig, die sich tatsächlich sehr ähnlich sind. Um diese objektiv bewerten und vergleichen zu können, wird der Matchingfaktor für jede Kante getrennt errechnet.

Sollte aber eine der beiden Kanten nicht passen, wird immer der `Integer.MAX_VALUE` zurückgegeben, das Teil ist also nicht verwendbar.

Bei der Berechnung des Matching-Faktors wird wieder auf die oben erwähnten Werte zurückgegriffen, um die Unterschiede zuerst zu messen und dann für jede Eigenschaft verschieden zu gewichten. Die unterschiedlichen Gewichtungen sind notwendig um die verschiedenen Eigenschaften in einem Wert vergleichbar zu machen.

Der Ausdruck 2.1 zeigt die unterschiedlichen Gewichtungen, wobei A und B die beiden zu vergleichenden Kanten sind, l die Länge einer bestimmten Ausbuchtung bezeichnet, f die Füllfläche und w die Breite einer Aus-/Einbuchtung. φ bezeichnet den jeweiligen Gewichtungsfaktor einer Eigenschaft.

$$\begin{aligned} \mathit{matchingfaktor} &= |l_A - l_B| \cdot \varphi_l \\ &+ |f_A - f_B| \cdot \varphi_f \\ &+ |w_A - w_B| \cdot \varphi_w \end{aligned} \tag{2.1}$$

Kapitel 3

Umsetzung des Puzzle Projekts

3.1 Verwendete Software

Das Projekt wurde als Plugin für das Java basierte Bildbearbeitungsprogramm ImageJ implementiert. ImageJ ist open-source und plattformunabhängig. Es bietet dem Entwickler eine einfache Schnittstelle, um selbst Plugins zu programmieren. Es wird vielfach für medizinische und wissenschaftliche Bildanalyse genutzt, wozu es schon Hunderte von Plugins gibt. Im Bereich der Druckvorstufe wird es für Farbraumanalysen verwendet.

3.1.1 ImageJ

Bildverarbeitungsalgorithmen zu schreiben und zu testen war in der Vergangenheit keine leichte Angelegenheit, da es noch vor Jahren an teurer Hardware benötigte, um eine schnelle Rechnerleistung gewährleisten zu können. Weiters ist es gerade für Programmiereinsteiger nur sehr schwer möglich für kommerzielle Softwareprodukte wie *Adobe Photoshop* oder *Corel Photo-Paint*, welche über ein ansprechendes User-Interface verfügen, selbst Plugins zu erstellen. Andere Entwicklungsprogramme wie *Matlab*[1] oder *ImageMagick*[2] hingegen helfen dem Entwickler bei der Algorithmenerstellung, bieten aber für Endanwender nur wenig Komfort.

Am *U.S. National Institute of Health* entwickelte daher Wayne Rasband ImageJ, ein auf Java basierendes Open-Source Projekt. Die aktuelle Version von ImageJ, sowie Updates, Dokumentation, Testbilder, sowie eine große Auswahl an Plugins findet man auf der offiziellen Homepage[3].

ImageJ ist frei verfügbar und somit das ideale Tool, um in Forschung und

[1] www.mathworks.com
[2] www.imagemagick.org
[3] http://rsb.info.nih.gov/ij/

Abbildung 3.1: Die ImageJ Programmoberfläche.

```
public int setup (String arg, ImagePlus imp) {
    return DOES_RGB; // this plugin accepst 24-bit color images
}
```

Abbildung 3.2: `setup`-Methode aus dem *Puzzle Projekt*.

Studium selbst Plugins zu entwickeln und zu testen. Weitere Hilfestellungen und Informationen über ImageJ und der Plugin Erstellung findet man auch unter [BB05, Kapitel 3]. Die Programmoberfläche ist in Abbildung 3.1 zu sehen.

3.1.2 `PlugIn` Interface

Um nun ein Plugin für ImageJ zu schreiben, muss man das Interface `PlugIn` bzw. `PlugInFilter` in seiner eigenen Java-Klasse implementieren. Der Unterschied zwischen diesen beiden ist, dass dem `PlugIn` kein Bild übergeben werden muss, dem `PlugInFilter` hingegen wird das aktuell in ImageJ geöffnete Bild übergeben.

Folgende Methoden muss ein `PlugInFilter` implementieren:

- `int setup (String arg, ImagePlus img)`
  In dieser Methode werden die Spezifikationen eines Bildes, welches dem Plugin übergeben wird, überprüft. Dazu wird ein Bit-Vektor, der die Eigenschaften des Plugins beschreibt, zurückgegeben. Im Codestück 3.2 wird die Methode aus dem Puzzle Projekt gezeigt.

- `void run (ImageProcessor ip)`
  Die `run` Methode wird dann gestartet, wenn das Bild den oben genannten Test besteht. Dieser Methode wird das aktuell ausgewählte Bild aus ImageJ übergeben, welches sodann über den `ImageProcessor ip` angesprochen werden kann.

3.1.3 `ImagePlus` und `ImageProcessor` Objekt

Ein `ImagePlus` Objekt speichert nicht nur die Bildinformationen an sich (wie der `ImageProcessor`, der nur das Pixel Array beinhaltet), sondern

PuzzleProject_.java	Plugin Klasse, welche aus ImageJ aufgerufen wird
Puzzle.java	beschreibt das Puzzle und bietet Methoden zum Lösen desjenigen
PuzzlePiece.java	speichert und analysiert ein einzelnes Puzzleteil
Edge.java	speichert eine einzelne Kante, analysiert diese und macht sie vergleichbar
ResultPuzzle.java	speichert die Beziehungen und Position der gelösten Puzzleteile
PPLocator.java	Schnittstelle zwischen den Positionen der Kanten des `PuzzlePiece` und eines gelösten Teiles
Matrix.java	Hilfsklasse um mit Matrizen zu rechnen
Vector.java	Hilfsklasse um mit Vektoren zu rechnen

Abbildung 3.3: Dateiliste: Die verschiedenen Klassen des Projektes.

auch Informationen wie z. B. Bildtyp, oder auch mehrere Bilder in Form eines `ImageStacks`. Um ein neues Fenster mit einem Bild zu erzeugen ist ebenfalls ein `ImagePlus` Objekt notwendig.

Im nächsten Abschnitt wird auf die Struktur des Plugins eingegangen.

3.2 Struktur des Plugins

Abbildung 3.3 zeigt eine Liste der einzelnen Dateien bzw. Klassen, welche das Puzzle Projekt umfasst. Im Folgenden werden die einzelnen Klassen und deren wichtigsten Funktionen und Methoden beschrieben.

3.2.1 `PuzzleProject_.java`

Wie schon in Abschnitt 3.1.2 erläutert, implementiert diese Klasse die vorgeschriebenen Methoden.

In der `run()` Methode wird ein Objekt `puzzle` vom Typ `Puzzle` erzeugt, welchem das Farbbild übergeben wird. Die gesamte Verarbeitung und Analyse wird von dieser Klasse ausgehend gemacht. Zum Schluss wird in `run()` die Methode `puzzle.solveIt()` aufgerufen, welche den Lösungsprozess startet.

Weiters können hier durch Ein- und Auskommentieren einzelner Befehle auch verschiedene Bilder und Zwischenergebnisse ausgegeben werden. Somit

kann man sich z. B. das Binärbild oder einen Stack mit den analysierten Eckpunkten ausgeben lassen.

3.2.2 `Puzzle.java`

Diese Klasse beschreibt das gesamte Puzzle und von hier aus werden alle Analyse und Lösungsprozesse gestartet. Im Konstruktor werden die Methoden `makeBinaryPicture()` und `splitToPieces()` aufgerufen.

In der Methode `splitToPieces()` werden dann in einer Liste `pieces` die einzelnen Teile als `PuzzlePiece` Objekte angelegt.

Folgende Methoden sind `public`:

- `boolean solveIt()`
  Diese Methode startet den gesamten Lösungsprozess wie in Abschnitt 2.4 beschrieben.
- `ImagePlus getPiecesStack()`
  Hier wird ein `ImagePlus` Objekt zurückgegeben, welches einen Stack mit den einzelnen Teilen, welche schon richtig gedreht wurden, ausgibt. Auf Wunsch können auch die gefundenen Eckpunkte eingezeichnet werden.
- `int getNumberOfPieces()`
  gibt die Anzahl der gefundenen Teile zurück.
- `ImageProcessor getOriginal()/getBinary()/getResult()`
  gibt das Original oder Binärbild bzw. das gelöste Puzzlebild zurück.

3.2.3 `PuzzlePiece.java`

Wenn ein Objekt eines `PuzzlePiece` erzeugt wird, wird dieses wiederum im Konstruktor komplett analysiert (siehe Abb. 3.4).

Weiters stellt diese Methode Funktionen zur Verfügung, welche es möglich machen, ein Teil mit einer oder zwei übergebenen Kanten zu überprüfen. Wie schon in Abschnitt 2.4.2 beschrieben, wird hier der Algorithmus (in Abb. 3.5 dargestellt) angewendet. Dabei wird zur Berechnung die Methode `matchingValue()` der Klasse `Edges` verwendet, welche in Abschnitt 2.4.2 erklärt wird.

Im `PuzzlePiece` werden neben der im Konstruktor errechneten Eigenschaften auch das Farb- und Binärbild zur Weiterverarbeitung gespeichert.

3.2.4 `Edge.java`

Die `Edge`-Klasse beschreibt eine einzelne Kante eines Puzzleteiles. Beim Anlegen eine Objektes werden dem Konstruktor die beiden Eckpunkte, die die Kante beschreiben, die `id` um welche Kante es sich handelt, sowie das schon

```
PuzzlePiece(ImageProcessor colorPic, ImageProcessor binPic) {
    // initialisiert Variblen im Standardkonstruktor
  this();
    // fügt den Bildern einen weißen Rand hinzu,
    // um ohne Problem rotieren zu können
  addWhiteSpace(colorPic, binPic);
    // findet die Eckpunkte und rotiert das Bild,
    // sodass eine Kante waagerecht ist
  findCornersAndRotatePicture();
    // berechnet die Winkel und die Eckpunkte jeweils einer Kante
  calculateAngleAndCornerPerEdge();
    // erstellt aus den berechneten Kantenpunkte
    // die einzelnen Kanten als Objekte
  createEdges();
    // berechnet den Typ des Teiles (Eckteil, Rand oder Normal)
  checkAndOrderEdges();
}
```

Abbildung 3.4: Java-Quellkode: Konstruktor der `PuzzlePiece` Klasse.

richtig gedrehte Bild als `ImageProcessor` übergeben. Richtig gedreht heisst in diesem Falle, dass die gedachte Linie zwischen den beiden Punkten genau paralell zur y-Achse ist.

Die Analyse der Kante, welche wieder im Konstruktor erledigt wird, läuft wie in Abschnitt 2.3.2 erläutert ab.

Die Methode `matchingValue(Edge edge2, int moreTreshold)` hat einen Integer als Rückgabewert, welcher den Matchingfaktor beschreibt. Der Parameter `edge2` referenziert die zweite Kante, mit der verglichen werden soll, der zweite Parameter `moreTreshold` wird dazu verwendet, um den Schwellwert dynamisch von der `Puzzle` Klasse aus zu erhöhen. Am Beginn wird hier 0 übergeben, der Schwellwert kann um 50 Schritte erhöht werden, was einer Fehlertoleranz von ca. 50 Pixel entspricht.

3.2.5 `ResultPuzzle.java`

Der `ResultPuzzle`-Klasse wird eine Referenz auf das Löungsbild in der `Puzzle`-Klasse übergeben. Diese Klasse ist für das grafische Zusammensetzen des Lösungsbildes zuständig. Hierfür werden Methoden angeboten, welche es ermöglichen, ein Teil rechts vom letzten Teil zu setzen oder ein Teil in die nächste Zeile zu setzen.

Um dies zu ermöglichen müssen die Eckpunkte der betroffenen Teile gespeichert werden und der Winkel bezogen auf des neue Teil und das schon gesetzte Teil errechnet werden. Nach der Rotation wird das neue Teil an den letzten Eckpunkt gesetzt. Eventuelle Verschiebungen müssen dabei berücksichtigt werden.

```
public int[] matchPieceWith2Edges(Edge edgeLeft, Edge edgeTop,
                                  int moreTreshhold) {
  int matchingValueSum = Integer.MAX_VALUE;
  int matchingEdge = 99;
  for (int i = 0; i < 4; i++) {
    int matchingValue = Integer.MAX_VALUE;

    int matchValLeft
      = Math.abs(edges[i].matchingValue(edgeLeft, moreTreshhold));
    int matchValTop
      = Math.abs(edges[(i+1)%4].matchingValue(edgeTop, moreTreshhold));

    if (matchValLeft > 200000 || matchValTop > 200000)
     matchingValue = Integer.MAX_VALUE;
    else
     matchingValue = matchValLeft + matchValTop;

    if (matchingValue < matchingValueSum) {
      matchingValueSum = matchingValue;
      matchingEdge = i;
    }
  }
  return new int[] {matchingValueSum, matchingEdge};
  // Es wird der matchingValue und der Index der Kante zurückgegeben,
  // welche nach oben ausgerichtet sein muss.
}
```

Abbildung 3.5: Java-Quellkode: `matchPieceWith2Edges()`-Funktion der `PuzzlePiece` Klasse.

3.2.6 `PPLocator.java`

Da in einem Objekt der Klasse `PuzzlePiece` ein Puzzleteil lediglich einmal so rotiert wird, dass die Kanten analysiert werden können, wird es sehr kompliziert sobald das Teil anders als in der Ausrichtung, wie es in `PuzzlePiece` gespeichert wurde, gesetzt wird.

Wie schon in Abschnitt 3.2.4 beschrieben hat eine Kante eine bestimmte `id`, die ihre Lage am Puzzleteil beschreibt. Die `id` wird im Uhrzeigersinn, wie in Abbildung 3.6 erkennbar, vergeben.

Beim Setzen eines Puzzleteiles in das Lösungsbild wird nun ein `PPLocator`[4] angelegt, welcher das zu setzende Puzzleteil enthält, sowie die `id` der Kante, welche nun im ausgerichteten Zustand die obere Kante ist.

Der `PPLocator` bildet nun die Schnittstelle zwischen `Puzzle` und `ResultPuzzle` mit dem `PuzzlePiece`. Man kann über den `PPLocator` wieder ganz normal z. B. auf die obere Kante mit der `id 0` zugreifen, der `PPLocator`

[4] kurz für: *PuzzlePieceLocator*

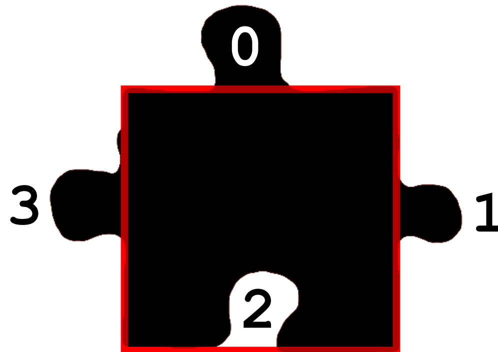


Abbildung 3.6: Die Nummerierung der 4 Kanten erfolgt von der oberen Kante ausgehend bei 0 beginnend im Uhrzeigersinn.

gibt dann die richtige Kante bezogen auf die Ausrichtung im gelösten Zustand zurück.

3.3 Anwendung

Bevor das Plugin gestartet werden kann, muss das Bild mit den zu lösenden Puzzleteilen in ImageJ geöffnet werden. Sodann kann über das Plugin Menü das `PuzzleProject` ausgewählt werden. Der Benutzer kann in der Dialogbox, die nun erscheint, den Hintergrund auswählen, den Analyseprozess starten oder das Plugin abbrechen.

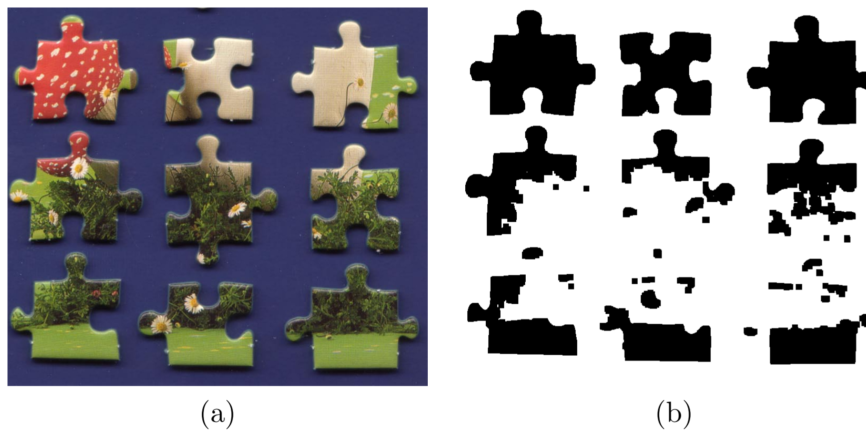
In der Konsole (Abb. 3.9) werden die Textausgaben gemacht, und etwaige Fehler die während des Lösungsprozesses auftreten ausgegeben. Je nachdem welche Einstellungen in der `PuzzleProject_.java` gemacht worden sind, werden während des Analyseprozesses diverse Test- und Debugbilder ausgegeben (z. B. Binärbild, einzelne Puzzleteile und deren Ecken).

Am Ende wird ein leeres Bild geöffnet, in welches dann in Echtzeit die gefundenen Puzzleteile gesetzt werden. Es werden solange Teile gesetzt, bis das Puzzle vollständig gelöst wurde oder kein Teil mehr gefunden wurde. Im letzteren Fall erscheint eine Fehlermeldung in der Konsole.

3.4 Tests

3.4.1 Grundlagen

Bei unseren Tests merkten wir schnell, dass es eine große Rolle spielt, in welcher Auflösung das Ausgangsmaterial vorliegt. Wie schon in Abschnitt 2.1.2 erwähnt, ist daher einer Mindestanforderung für das Ausgangsbild notwendig. Die Auflösung eines Steines ist mit den Abtastwerten beim Matching-



(a) (b)

Abbildung 3.7: Diese beiden Grafiken zeigen, dass dunkle Bereiche im Bild nicht von den Schatten, die beim Scannen auftreten unterschieden werden können: (a) zeigt das Ausgangsbild, (b) das erstellte Binärbild.

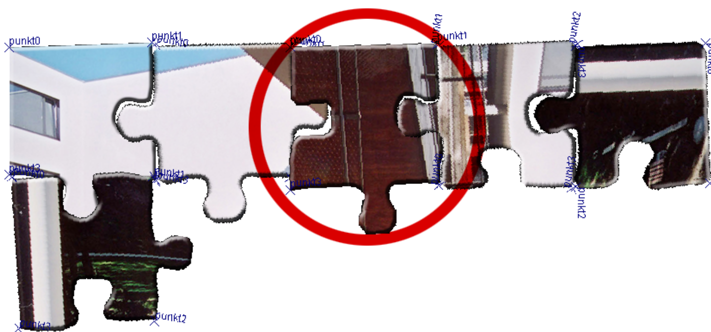


Abbildung 3.8: ein gescheiterter Versuch.

prozess gleichzusetzen und deswegen so entscheidend.

Auch ein einheitlicher Hintergrund ist zur fehlerfreien Erkennung der einzelnen Teile notwendig. Abbildung 3.7 zeigt einen Testfall, bei dem es nicht möglich war, automatisiert die Puzzleteile vom Hintergrund zu trennen, da zu viele ähnliche Farben im Bild waren.

Jeder Teil-Algorithmus wurde im Vorfeld einzeln getestet, bis ein optimales Ergebnis gefunden wurde. Beim Puzzleteilvergleichsverfahren mussten aber alle Analyseprozesse schon funktionieren um mit dem Testen beginnen zu können.

3.4.2 Probleme beim Matching

Wir versuchten zuerst 2 Teile richtig zusammenzubauen und erweiterten unser Testbild so weit, bis die erste Reihe komplett gelöst werden konnte. Problem dabei war, sobald wieder mehr Teile im *Teile-Pool* lagen, musste das Vergleichsverfahren wieder überarbeitet und verfeinert werden, da schon in der ersten Reihe immer wieder Fehler passierten. Abbildung 3.8 zeigt, dass es passieren kann, dass schon eines der ersten Teile falsch gesetzt wird, das Programm aber trotzdem versucht das Puzzle weiter zu lösen. Erst wenn es keine möglichen Teile (die im ersten selektiven Auswahlverfahren - siehe Abschnitt 2.4 - überprüft werden) vorhanden sind, wird der Suchalgorithmus abgebrochen. Bei unserem Verfahren wird also kein *Backtracking*[5] eingesetzt, welches ein nachträgliches korrigieren von Fehler möglich machen würde.

3.4.3 Zeit und Speicherkomplexität

Als sehr rechenintensiv hat sich der Corner-Findungs-Prozess herausgestellt. Hierbei müssen pro Bildpixel an die 1600 weitere Pixel untersucht werden, dies entspricht ca. 400 Millionen Pixel pro Puzzleteil, um die korrekten Eckpunkte herauszufinden. Da es bei verdrehten Bildern vorkommen kann, dass ein Bild mehrfach (maximal 9 mal – 90°) rotiert werden muss, und jedes mal wieder versucht wird, die Ecken zu finden, kostet dies besonders viel Zeit.

3.4.4 Resümee und Ergebnis

Mit unserer Methode schafften wir es aber schließlich ein 24-teiliges Puzzle in lediglich 30 Sekunden auf einem mobilen Intel Centrino 1,7 GHz Prozessor mit 1024 MB RAM zu lösen. Abbildung 3.10 zeigt das Ausgangsbild, welches wir schon im Vorfeld mit schwarzen Hintergrund gefüllt haben. Die automatisiert erstellte Lösung ist in Abbildung 3.11 zu sehen. In der Konsole (Abb. 3.9) werden die aktuellen Arbeitssschritte ausgegeben und mögliche Fehler angezeigt.

[5] *Backtracking* bedeutet, dass schon gesetzte Teile wieder zurückgenommen werden können und nach einer anderen Lösung gesucht wird, vgl. auch [WLS91].

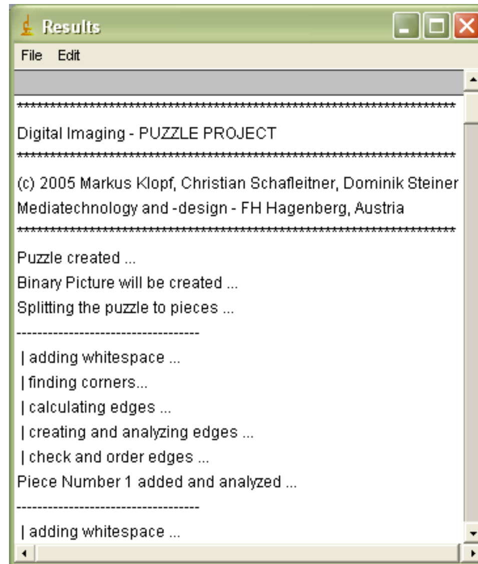


Abbildung 3.9: In der ImageJ Konsole werden die aktuellen Arbeitsschritte, sowie Fehler, ausgegeben.

Abbildung 3.10: Ausgangsbild.

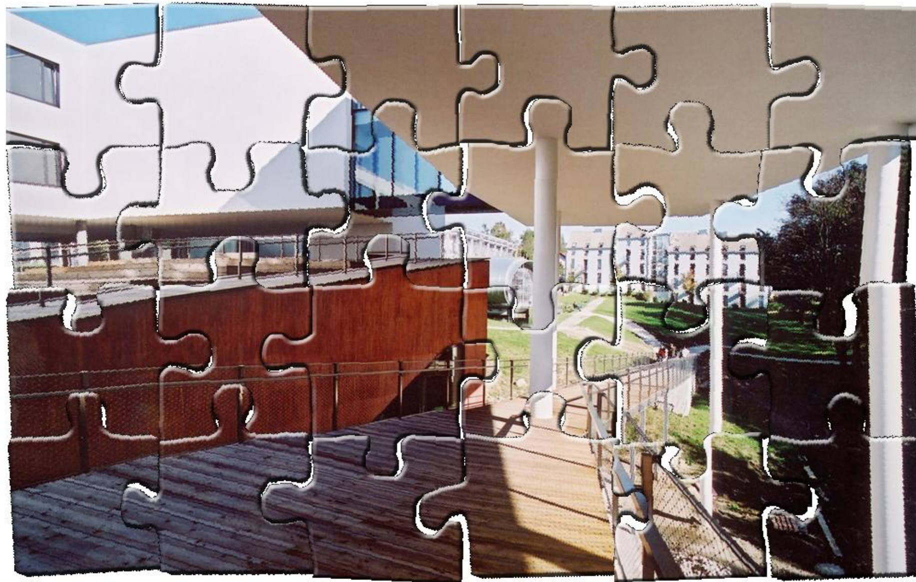


Abbildung 3.11: Lösung nach der Verarbeitung mit dem *Puzzle Projekt*, das Bild zeigt den Aufgang in den Innenhof der neuen Fachhochschule am Hagenberger Campus.

Kapitel 4

Konzepte zur Verbesserung des Puzzle Projekts

4.1 Scanning-Techniken

Schon am Beginn unserer ersten Tests erkannten wir, dass das Ausgangsmaterial eine große Schwierigkeit darstellte. Ein zufriedenstellendes fotografisches Abbild zu bekommen war nicht ganz so einfach wie wir uns das vorstellten. Vorallem lehnten wir es auch ab, die Fotos aufwendig im *Photoshop* oder anderen Bildprogrammen nachzubearbeiten.

Ein hochauflösendes Foto von den Puzzleteilen zu machen ist leider nicht möglich, da es unweigerlich zu Verzerrungen kommt. Ein Scan wiederum wirft Schatten auf Bildbereiche, die die Bestimmung eines einheitlichen Umrisses unmöglich macht. Wir versuchten durch Dilationen und Erosionen, die Schatten und fehlenden Farben zu eliminieren, was aber unweigerlich zu einer Verfälschung der Umrisse führte. Das Nachbearbeiten des Hintergrundes in *Photoshop* war eine mühsame Angelegenheit.

Goldberg et al. [GMB02] beschäftigen ebenfalls in deren „*Global Approach*“ mit diversen Scanning und Beleuchtungstechniken, ehe sie nach unzähligen Versuchen mit einer reinen Schwarz/Weiß Kopie mit einem geöffneten Kopiererdeckels und anschließendem Scan dieser Kopie die besten Ergebnisse erzielten.

Bei Webster et al. [WLS91] wird jedes Puzzleteil über eine CCD Kamera eingelesen. Die einzulesenden Teile werden hierbei auf eine von hinten beleuchtete Oberfläche gelegt, um klare Umrisse zu bekommen. Da es uns wichtig war, ein Farbbild für das dynamische Zusammensetzen der einzelnen Teile zu erhalten, konnten wir diese Methoden nicht eins zu eins umsetzen. Ein Scannen mit geöffneten Deckel brachte aber gute Ergebnisse, und der schwarze Hintergrund kann ohne Problem heraus gefiltert werden.

Dieser kurze Abschnitt sollte zeigen, dass es meist viel einfachere Lösungen für scheinbare Probleme gibt und das sich schon mehrere Personen mit

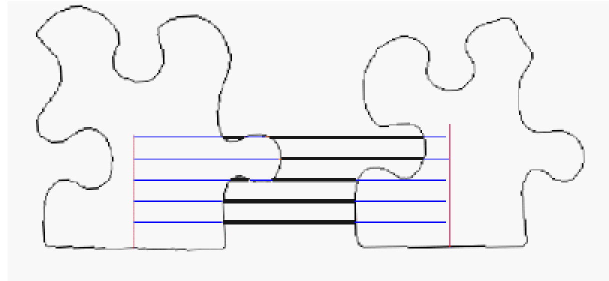


Abbildung 4.1: Berechnung der Unterschiede der Abstände. Bild aus [GMB02].

ähnlichen Problemen, wie wir sie hatten, herumgeschlagen haben.

4.2 Rand bilden & *Fiducial Points*

In unserer Lösung wird das Puzzle zeilenweise aufgebaut, somit gibt es für jedes Puzzleteil zwei Kanten, welche eindeutig sind und mit denen die möglichen weiteren Teile verglichen werden. Bei der ersten Zeile ist die obere Kante immer ein *Rand*, bei jeder weiteren Zeile kann auf eine obige Kante referenziert werden.

Es wird also Teil für Teil nach einem weiteren passenden Puzzleteil gesucht, und dieses dann an die aktuelle Stelle hinzugereiht. Besonders am Anfang, wo noch besonders viele Teile im *Teile-Pool* liegen, ist die Fehlerwahrscheinlichkeit am Größten.

Problem hierbei ist, dass es an anderen Stellen teilweise eindeutigere Lösungen gibt. Wenn ein Mensch ein Puzzle löst, legt er auch zuerst die Teile zusammen, die für ihn eine höhere Zusammengehörigkeitswahrscheinlichkeit haben.

4.2.1 Randteile suchen und vergleichen

Diesen Ansatz verfolgen Goldberg, Malon und Bern in deren Lösungsansatz in [GMB02, 3.2], indem sie, bevor sie die Teile mit mehreren Nachbarn untersuchen, den gesamten Rahmen zu bilden versuchen. Dabei werden die Randteile an der glatten Seite ausgerichtet und für jedes Teil der am besten passende Nachbar gewählt.

Hier greifen Goldberg et al. auf ein einfaches Vergleichsverfahren zurück. Es werden die Abstände wie in Abbildung 4.1 gemessen und die Unterschiede der einzelnen Messergebnisse ergeben den Wert $s(A, B)$, wobei A und B die beiden Kanten sind. Der Wert s beschreibt die Unterschiede in der Länge der verschiedenen gemessenen Abstände.

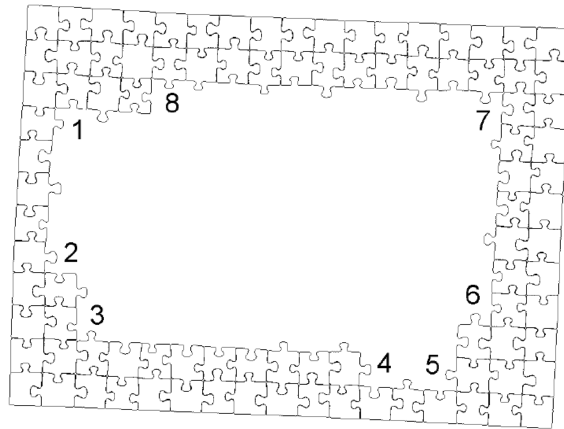


Abbildung 4.2: Stellen, an welche ein weiteres Teil angelegt werden könnte. Bild aus [GMB02].

Von den Eckteilen ausgehend wird nun versucht, dass der Rand komplett gebildet wird. Da die Eckteile symmetrisch angeordnet sein müssen, kann dies nun schon am Beginn des Lösungsprozesses überprüft werden. Wenn also nur ein Fehler an einem Rand auftritt, kann dieser auf diese Weise sofort erkannt werden, da die Symmetrie nicht mehr gegeben ist. Es wird solange nach einer Lösung gesucht, bis ein passender Rahmen gefunden wird.

4.2.2 Wahrscheinlichkeitsfaktor

Durch den nun schon vorhandenen richtigen Rahmen, ergeben sich 4 Stellen, an welche ein Teil mit je 2 Kanten verglichen werden kann. Abbildung 4.2 zeigt, dass es bei weiterer Lösung sogar zu mehr Stellen kommt, bei denen ein Teil probiert werden kann. Es wird nun für jede Stelle ein passendes Teil gesucht und dann jenes Teil ausgewählt, welches den höchsten Wahrscheinlichkeitsfaktor hat.

Beim nächsten Durchlauf wird wieder das Puzzleteil mit dem nächst niedrigeren Wert verbaut, außer es wurde schon gesetzt, dann muss für diese Stelle wieder ein neues passendes Teil gesucht werden.

4.2.3 *Fiducial Points*

Um nun die innen liegenden Teile richtig zu positionieren, werden für jede Aus-/Einbuchtung sogenannte *Fiducial Points* berechnet. Dazu wird bei den Wendepunkten (*inflection points*) (siehe Abb. 4.3 (a)) eine Ellipse darübergelegt, der Mittelpunkt der Ellipse wird sodann als *fiducial point*[1] bezeich-

[1] vgl. leo.dict.org: fiducial center [metr.] - das Rahmenachsenkreuz (*Vermessungswesen*)

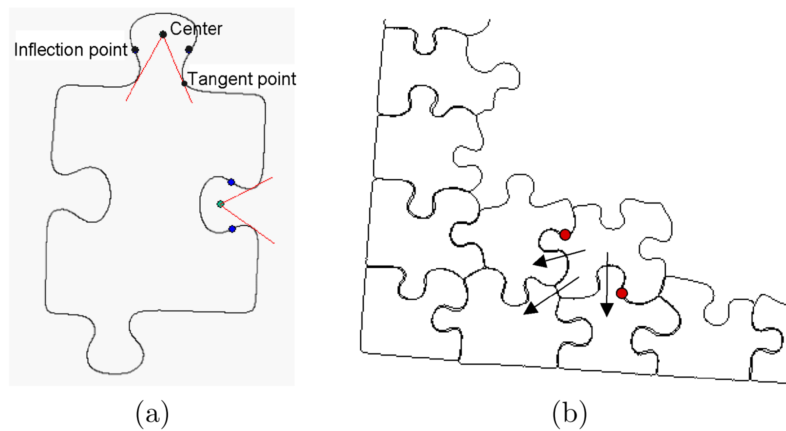


Abbildung 4.3: *Fiducial Points*: (a) zeigt die Zusammenhänge zwischen Wendepunkt, Tangentenpunkt und Mittelpunkt , in (b) wird die zu untersuchende Kante zwischen den beiden roten Punkten veranschaulicht. Somit wurde es auch möglich Puzzles mit Teilen, die mehr als 4 Nachbarn aufweisen, zu lösen. Bilder aus [GMB02].

net. Nun werden Tangenten (wie in Abb. 4.3 (a)), die durch den Mittelpunkt gehen, angelegt.

Mit Hilfe dieser Punkte werden die Teile ausgerichtet. Abbildung 4.3 (b) zeigt, dass nun die Kante zwischen zwei Tangentenpunkte (durch zwei rote Punkte gekennzeichnet) auf Passgenauigkeit überprüft wird, ehe das Teil gesetzt wird.

4.2.4 Resümee

Mit dieser Methode schafften es Goldberg et al. ein 204-teiliges Puzzle zu lösen, sowie ein 100-teiliges Puzzle mit Teilen die mehr als 4 Nachbarn haben. Somit ist dieser Lösungsweg der bisher erfolgreichste.

4.3 Übereinanderlegen der Teile

4.3.1 Visuelles Lösungsverfahren

Einen anderen interessanten Ansatz verfolgt Michael Hurnaus in seiner Bakkalaureatsarbeit [Hur06]. Er wählt einen sehr visuellen Lösungsansatz: Es werden zur Lösung ebenfalls die Eckpunkte und Kanten der einzelnen Puzzleteile gesucht. Allerdings werden dann die Teile übereinandergelegt und weiße (also freibleibende), sowie auch überlappende Pixel gezählt. Ein Teil passt also am besten zusammen wenn es möglichst wenige Überlappungen bzw. freie Stellen/Pixel gibt.

Das so erzielte Ergebnis kombiniert die meisten Messungsverfahren wie die Längen der Aus- und Einbuchtungen und liefert somit in nur einem Wert ein ziemlich aussagekräftiges Ergebnis.

Allerdings muss wieder ein gewisser Toleranzspielraum gelassen werden, da die zusammentreffenden Ecken nicht zu 100% gleich bestimmt werden können. Dadurch ergeben sich wieder Fehler und diese führen zu einer Einschränkung auf eine bestimmte Anzahl von unterschiedlichen Puzzleteilen.

4.3.2 Vergleich mit Robotor und Drucksensoren

Ein Mensch hat einen weiteren Sinn den er beim Lösen von Puzzles einsetzt – und zwar den Tastsinn. Beim Zusammenstecken einzelner Teile spürt man meist ob die beiden Teile ineinander passen oder ob es einen Widerstand gibt und das Teil nur mit starkem Druck hineingedrückt werden kann.

Burdea und Wolfson [BW89] haben einen Roboter entwickelt, der die Teile nicht nur visuell analysiert, sondern auch beim Zusammenbauen der Teile mit Hilfe von *Force Feedback*, prüft ob die Teile leicht ineinander gehen, oder nicht.

Wenn also nur Methoden der digitalen Bildverarbeitung verwendet werden können, ist die oben beschriebene Methode eine gute Möglichkeit diesen Tastsinn nachzuempfinden, da hierbei schon kleine Überlappungen auf ziemlich intuitive Weise erkannt werden können.

4.4 *Isthmus Critical Points*

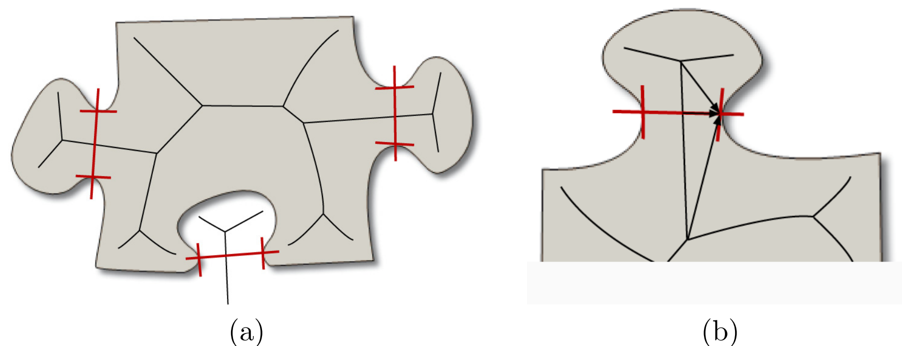
Webster et al. untersuchen bei ihrem Lösungsansatz in [WLS91] die Ein- und Ausbuchtungen im Speziellen. Die sogenannten *Isthimii Critical Points*[2] werden dabei berechnet und verglichen.

Um diese kritischen Punkte zu finden, wird ein *Euklidisches Skelett*[3] wie in Abbildung 4.4 (a) angelegt. Dabei werden die einzelnen Zweige eines Skelettes auf die Abstände zur Umrisslinie geprüft, wobei wie in Abbildung 4.4 (b) ein kritischer Punkt dadurch beschrieben wird, das der geringste Abstand nicht an den Enden, sondern innerhalb eines Zweiges liegt.

Vom *Isthmus Critical Point*, welcher in der Mitte der *Isthmus Critical Line* liegt werden nun die Konturen der Ein- und Ausbuchtungen abgetastet. Das so entstandene Profil eines negativen Isthmus (Abb. 4.5 (b) oben) wird nun mit den Profilen der positiven Isthmii (Abb. 4.5 (b) unten) überprüft. Mithilfe einer Korrelation der beiden Funktionen, welche sich aus den einzelnen Abtastwerten ergibt, können die beiden Teile optimal positioniert

[2] vgl. dict.leo.org: Isthmus (engl.) - Landenge

[3] Unter Skelettierung bezeichnet man eine größtmögliche Vereinfachung eines Objektes auf die wesentlichsten Teile, ohne dadurch Informationen der Grundform zu verlieren, siehe auch [Grä04].



(a) (b)

Abbildung 4.4: *Euklidische Skelette*: (a) zeigt die Skelettierung eines Puzzleteiles, die unterschiedlichen Abstände eines einzelnen Skelettknochens dienen zum Finden der *Isthmus Critical Points* (b).

werden, sodass ein Vergleich stattfinden kann. Ein *GAP*[4] Wert, wie in Abbildung 4.5 (c) erkennbar, wird sodann berechnet. Der Stein mit dem geringsten Durchschnittswert, welche sich aus $\frac{GAP}{Pfadl\ddot{a}nge}$ errechnet, passt also zu diesem Teil.

Mithilfe dieser Methode war es 1991 möglich ein 24-teiliges Puzzle zu lösen, wobei ein großes Problem der Autoren die mangelnde Rechenleistung der damaligen Computer war. Mithilfe einer CCD Kamera wurde ein 1024×1024 großes Bild erstellt, welches mit einer *Sun Sparcstation 330*, die mit 16 Megabyte RAM und zwei 699 Megabyte Festplatten bestückt war, verarbeitet wurde.

4.5 *Curve Matching* Verfahren

Weixing Kong und Benjamin B. Kimia versuchten einen Puzzle Lösungsalgorithmus zu finden, welcher nicht auf bestimmte Formen der Puzzleteile angewiesen ist. Deren „Curve Matching“ Verfahren [KK01] untersucht keine Teilkonturen, sondern betrachtet die gesamte Outline eines Teiles.

Wie in Abbildung 4.6 sichtbar, werden einzelne Punkte, an denen sich die Kurve markant ändert, herausgenommen und im ersten Schritt nach passenden weiteren Punkten untersucht. Mit Hilfe dieser Punkte können Start- und Endpunkte möglicher zusammengehöriger Kurvenabschnitte bestimmt werden: Die Punkte werden dazu in einer Tabelle gegenübergestellt und für je 2 Punktpaare die gemeinsame Kurve betrachtet. Wiederum wird ein Matching-Faktor errechnet, um die Kurven vergleichbar zu machen. Ein niedriger Faktor beschreibt in diesem Falle eine große Ähnlichkeit der Kurven. Sobald gemeinsame Startpunkte gefunden wurden, wird das Vergleichs-

[4] vgl. dict.leo.org: gap (engl.) - Abstand, Spalt, Fuge

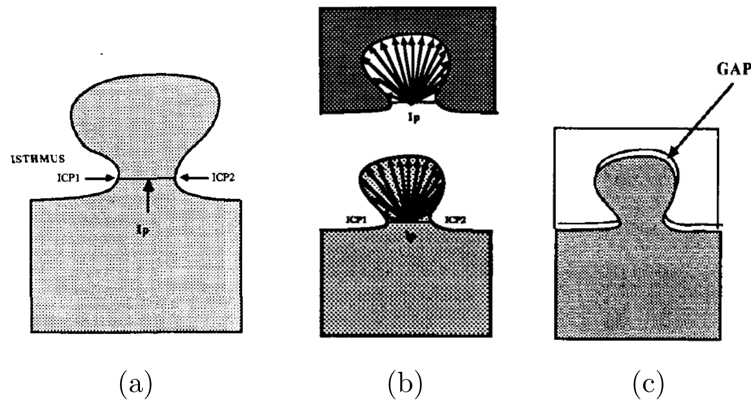


Abbildung 4.5: *Isthmus Critical Points*: (a) zeigt die *Isthmus Critical Line* mit den dazugehörigen Punkten, in (b) sieht man die Abtastwerte von einer negativen (oben) bzw. einer positiven *Isthmus Critical Line* ausgehend. Der auftretende Spalt ist in (c) erkennbar, durch Verschieben des Teiles kann dieser aber noch verringert werden. Bilder aus [WLS91].

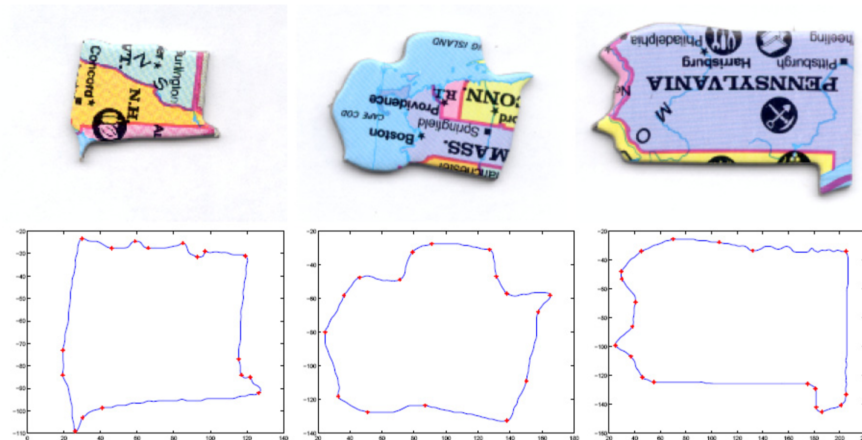
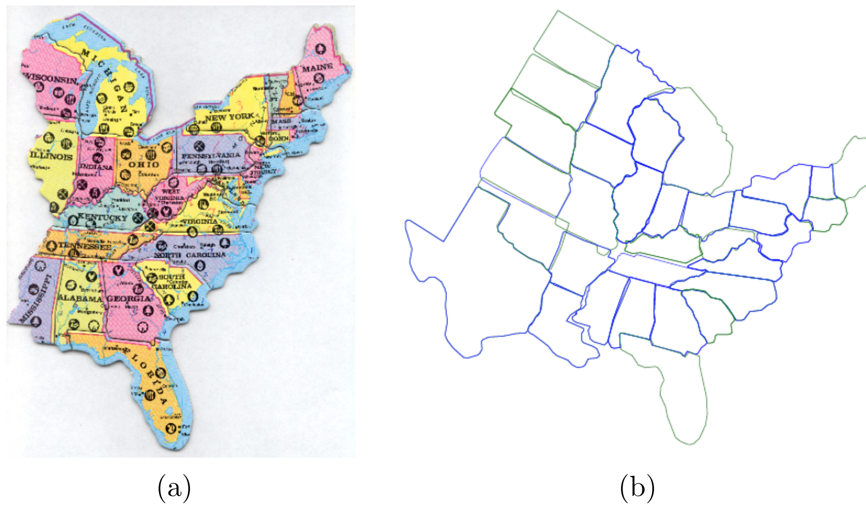


Abbildung 4.6: Stellen, an welche ein weiteres Teil angelegt werden könnte. Bilder aus [KK01].



(a) (b)

Abbildung 4.7: USA Landkarte (a), welche mit dem *Curve Matching* Verfahren (b) zusammengesetzt wurde. Bilder aus [KK01].

verfahren auf die gesamte gemeinsame Kurve ausgeweitet um ein präziseres Ergebnis zu bekommen.

4.5.1 Vorteile

Vor allem das Finden der Eckpunkte, wie schon in Abschnitt 2.3.1 beschrieben, ist eine schwierige Angelegenheit und erfordert auch hohe Ansprüche an das Ausgangspuzzle. Mit dieser Lösung kann der Algorithmus nicht nur auf Puzzleteile angewendet werden, sondern wird in der Archäologie dazu verwendet, um gefunden Tonscherben, auch im 3D-Raum, wieder zusammenzufügen. Abbildung 4.7 zeigt die fehlerfrei zusammengesetzte Landkarte der USA.

Kapitel 5

Zusammenfassung

In dieser Bakkalaureatsarbeit wurde das automatisierte Lösen von Puzzles erarbeitet. Dazu wird ein eingescanntes Bild der einzelnen Puzzleteile analysiert. Die einzelnen verwendeten Methoden und Algorithmen wurden in Kapitel 2 und 3 beschrieben. Dabei wurde lediglich auf die Analyse der Konturen und auf das anschließende Matching der verschiedenen Puzzleteile eingegangen. *Force-Feedback* Lösungsmethoden oder die Analyse der Farbinformationen des zu darstellenden Bildes wurden in dieser Bakkalaureatsarbeit nicht betrachtet.

Der hier beschriebene Lösungsansatz ist bewusst einfach gehalten, da er als Projekt der Lehrveranstaltung *Digital Imaging* im 4. Semester an der Fachhochschule Hagenberg entwickelt wurde. Er gilt daher auch auf keinen Fall als perfekt, dennoch ist es mit der vorgestellten Lösung möglich innerhalb kürzester Zeit ein Puzzle mit ca. 24 Teilen zu lösen. Ein Intel Centrino mit 1,7 GHz und 1024 MB RAM braucht für den gesamten Analyse- und Lösungsprozess ungefähr 30 Sekunden. Der Erfolg der kompletten Lösung eines Puzzles ist zum großen Teil vom Ausgangsmaterial abhängig, welches die Teile und deren Umrisse in möglichst hoher Qualität zeigen sollte.

Verglichen mit anderen Algorithmen, welche ebenfalls in dieser Arbeit vorgestellt wurden, arbeitet unser Algorithmus daher sehr schnell, trotz hoher Effizienz. Die Lösung von Goldberg et al. [GMB02] brauchte zum Vergleich ca. 3 Minuten für ein 100-teiliges Puzzle, für das 204-teilige Puzzle schon 20 Minuten auf einer *Sun Ultra 60* Workstation.

Leider sind derartige Lösungsprozesse nicht fehlerfrei, da immer mit Schwellwerten gearbeitet werden muss, um Ungenauigkeiten des analogen Ausgangsmateriales ausgleichen zu können. Bei zu vielen Puzzleteilen werden die Unterschiede der verschiedenen Konturen einfach zu gering, um mit dem vorgestellten Verfahren ein fehlerfreies Matching durchführen zu können.

Auch mit anderen Lösungen, welche in Kapitel 4 beschrieben wurden, ist es bis heute noch nicht gelungen Puzzles mit mehr als 200 Teilen (auch

mit Einschränkungen) zu lösen. Somit werden noch einige Entwickler viele Nächte lang forschen und probieren müssen, bis auch Computer 1000- und mehrteilige Puzzles, die bis heute den Menschen vorbehalten sind, richtig lösen können.

Anhang A

Inhalt der CD-ROM

File System: Joliet

Mode: Single-Session (CD-ROM)

A.1 Bakkalaureatsarbeit

Pfad: /

PuzzleProjekt.pdf . . . Bakkalaureatsarbeit (PDF-File)

A.2 `PuzzleProject`-Plugin

Pfad: /PuzzleProject/

PuzzleProject_.java . . . Plugin Klasse, welche aus ImageJ aufgerufen wird

Puzzle.java . . . . beschreibt das Puzzle und bietet Methoden zum Lösen desjenigen

PuzzlePiece.java . . . . speichert und analysiert ein einzelnes Puzzleteil

Edge.java . . . . speichert eine einzelne Kante, analysiert diese und macht sie vergleichbar

ResultPuzzle.java . . . . speichert die Beziehungen und Position der gelösten Puzzleteile

PPLocator.java . . . . . Schnittstelle zwischen den Positionen der Kanten des `PuzzlePiece` und eines gelösten Teiles

Matrix.java . . . . Hilfsklasse um mit Matrizen zu rechnen

Vector.java . . . . Hilfsklasse um mit Vektoren zu rechnen

A.3 Testbilder

Pfad: /Testbilder/

- puzzle_demo1.tif . . . . 24 teiliges Puzzle zum Lösen
- puzzle_demo2.tif . . . . ein weiteres Puzzle, die Teile wurden dabei vertauscht
- puzzle_demo3.tif . . . . im 3. Testbild sind die Teile zusätzlich verdreht

A.4 Ergebnisbilder

Pfad: /Ergebnisbilder/

- puzzle_demo1.tif . . . . Lösungsbild 1
- puzzle_demo2.tif . . . . Lösungsbild 2
- puzzle_demo3.tif . . . . Lösungsbild 3

Lösungsbilder zu den gleichnamigen Bildern aus dem Ordner /Testbilder/.

Literaturverzeichnis

[BB05] Burger, Wilhelm und Mark Burge: *Digitale Bildverarbeitung.* Springer-Verlag Berlin Heidelberg, 2005.

[BW89] Burdea, G. und H. Wolfson: *Solving Jigsaw Puzzle by a Robot.* IEEE Trans. Electronic Computers EC, 5:752–764, December 1989.

[FG64] Freeman, H. und L. Garder: *Apictorial Jigsaw Puzzles: The Computer Solution of a Problem in Pattern Recognition.* IEEE Trans. Electronic Computers EC, 13:118–127, April 1964.

[GMB02] Goldberg, David, Christopher Malon und Marshall Bern: *A global approach to automatic solution of jigsaw puzzles.* In: *SCG '02: Proceedings of the eighteenth annual symposium on Computational geometry*, Seiten 82–87, New York, NY, USA, 2002. ACM Press.

[Grä04] Gräning, Lars: *Morphologie in der Bildverarbeitung*, 2004. Hauptseminar, Technische Universität Ilmenau.

[Hur06] Hurnaus, Michael: *Algorithmisches Lösen von Puzzles*, Februar 2006. Bakkalaureatsarbeit, Fachhochschule Hagenberg.

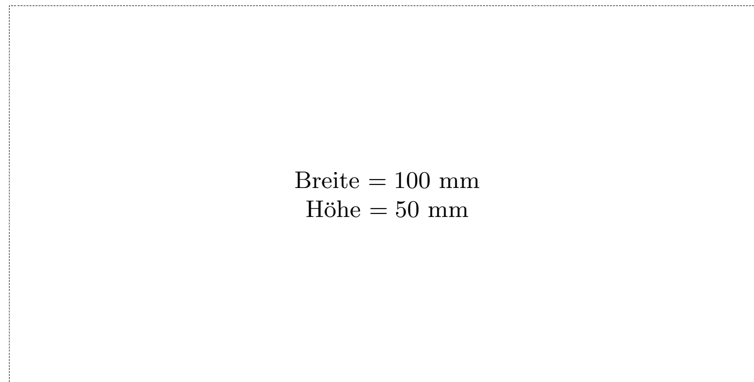
[KK01] Kong, Weixin und Benjamin B. Kimia: *On Solving 2D and 3D Puzzles using Curve Matching.* In: *Proc. of the IEEE Conference on Computer Vision and Pattern Recognition*, Hawaii, December 2001.

[Lan01] Lang, Hans Werner: *Travelling Salesman Problem - Approximationsverfahren.* http://www.iti.fh-flensburg.de/lang/algorithmen/np/tsp/roundtrip.htm, Februar 2001. Diverse Algorithmen, Fachhochschule Flensburg,.

[WLS91] Webster, R.W., P.S. LaFollette und R.L. Stafford: *Isthmus Critical Points for Solving Jigsaw Puzzles in Computer Vision.* IEEE Trans. Systems, Man, and Cybernetics, 21:1271–1278, 1991.

Messbox zur Druckkontrolle

— Druckgröße kontrollieren! —



— Diese Seite nach dem Druck entfernen! —