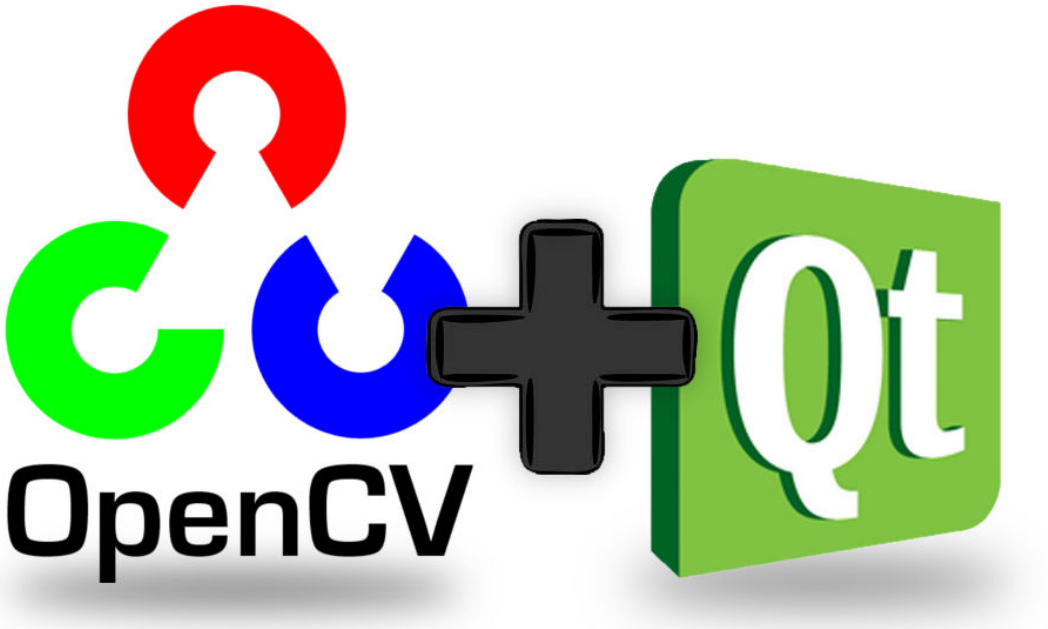
Puzzle Solver mit OpenCV und Qt



Maschinelles Sehen und Bildverarbeitung

14. Januar 2013

Puzzle Solver Projekt von

Cyril Stoller

Marcel Bärtschi

Dozenten

D. Lanz

W. Jenni

Inhalt

Problemstellung

Gegeben ist ein Bild mit einem Puzzle. Nun soll ein Algorithmus geschrieben werden, der herausfindet, wie die Puzzleteile untereinander zusammenpassen. Als Bildverarbeitungsbibliothek wird die frei verfügbare und aktiv weiterentwickelte Plattform *OpenCV* in C++ verwendet. Als Programmierumgebung wird der *Qt Creator* verwendet, welcher mit der Qt GUI-Bibliothek später eventuell auch noch eine graphische Oberfläche für das Programm ermöglicht.

Lösungsansatz

Zuerst wird das Bild binarisiert. Anschliessend werden aus dem binarisierten Bild die einzelnen Puzzleteile extrahiert. Nun werden aus der Kontur des einzelnen Puzzleteiles die vier Eckpunkte des Grundrechtecks gefunden. Anhand dieser Eckpunkte können nun die vier Seitenkonturen des Puzzleteils bestimmt werden. Diese Seitenkonturen werden nun auf verschiedene Features untersucht. Schlussendlich werden die Seitenkonturen von allen Puzzleteilen miteinander verglichen und die am besten passenden entsprechend markiert.

Die Implementation des Algorithmus konnte gut nach der Vorlage aus dem Lösungsansatz entwickelt werden.

Wichtige OpenCV Routinen

Die Folgenden Routinen wurden zum Dreh- und Angelpunkt unseres Projektes:

findContours

Dieser Funktion wird ein binarisiertes Bild übergeben. Sie versucht nun nach einem speziellen Algorithmus Konturen im Bild zu finden. Mit den diversen Parametern kann noch gewählt werden, ob hierarchische Konturen (also Konturen innerhalb anderer, geschlossener Konturen) ebenfalls gesucht werden und wie diese hierarchisch abgespeichert werden sollen. Ausserdem kann die Art der Konturlinie gewählt werden – also zum Beispiel 4-connected, 8-connected etc.

approxPolyDP

Diese Funktion versucht nach dem Ramer–Douglas–Peucker Algorithmus, eine Kontur zu vereinfachen. Es muss noch ein Parameter *epsilon* angegeben werden. Mit diesem kann man steuern, wie stark eine Kontur vereinfacht wird.

matchShape

Mit dieser Funktion kann die Ähnlichkeit zweier Konturen bestimmt werden. Die Funktion vergleicht die Bild Momente der beiden Konturen nach der sog. Hu-Menge invarianter Momente. Das bedeutet, dass sie resistent gegenüber Skalierung, Rotation und Spiegelung der beiden gegebenen Konturen ist. Je tiefer der zurückgegebene Wert der Funktion, desto besser stimmen die beiden überein.

Implementation

Hier werden nun die einzelnen Schritte genauer beschrieben:

Binarisieren

Zuerst wird das Bild in ein 8-Bit Graustufenbild umgewandelt. Um grobe Verrauschungen zu entfernen, wird das Bild mit einem Blur-Algorithmus verschmiert.

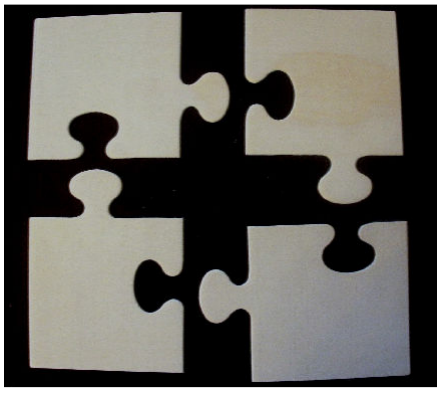
```
// Grauwert Bild erzeugen
cv::Mat imgGrey;
cv::cvtColor(img, imgGrey, CV_BGR2GRAY);

// Weichzeichnen (vor-entrauschen)
cv::Mat imgGreyBlur;
cv::blur(imgGrey, imgGreyBlur, cv::Size(3,3));
```

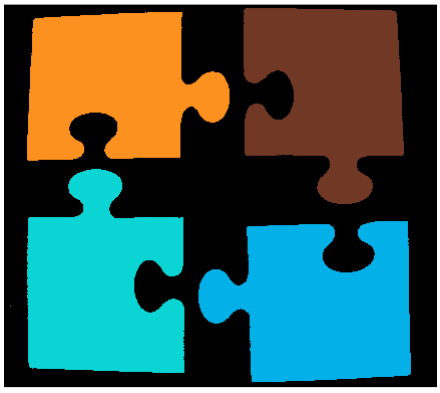
Die nachfolgende Binarisierung geschieht mit der Threshold funktion, welcher als Schwelle der Mittelwert des Bildes mitgegeben wird. Um die Binarisierung flexibler zu gestalten, könnte man ebenfalls die Funktion adaptiveThreshold benutzen. Als weitere Verbesserung könnte hier auch das Histogramm des Bildes ausgeglichen und anschliessend eine Binarisierung vorgenommen werden. Jedoch funktioniert die Funktion equalizeHist nicht für Bilder, welche nur wenige Farben enthalten und bereits fast Binärbilder sind, so wie unser Testbild.

```
// Bild binarisieren
cv::threshold(imgGreyBlur, imgBin, cv::mean(imgGreyBlur)[0]-10, 255, cv::THRESH_BINARY);
```

Nun wird das Bild noch einmal entrauscht, diesmal mit morphologischen Operationen: Mit einem 3x3 Filterkern wird das Bild zwei Mal geöffnet (zweimal erodieren und zweimal dilatieren).

```
// Entrauschen: 2x öffnen mit 3x3 Kernel
cv::Mat kernel = cv::getStructuringElement(cv::MORPH_CROSS, cv::Size(3, 3), cv::Point(1, 1));
cv::morphologyEx(imgBin, imgBin, cv::MORPH_OPEN, kernel, cv::Point(-1,-1), 2);
```





Puzzleteile extrahieren

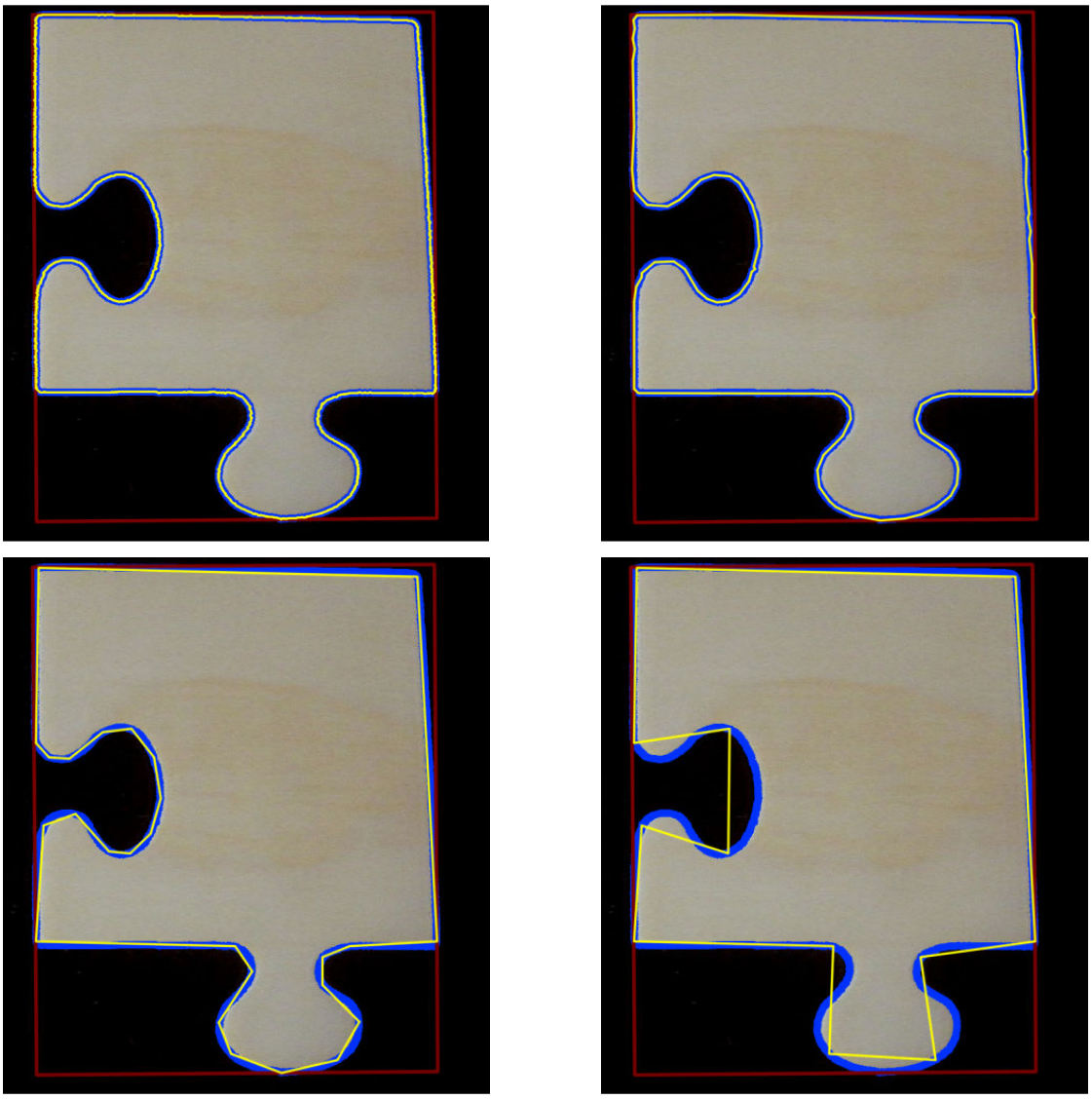
Um die Puzzleteile zu erkennen, nutzen wir die Funktion findContours. Diese liefert alle gefundenen Konturen als Array eines Arrays von Punkten zurück.

```
 // Konturen suchen
std::vector<std::vector<cv::Point> > contours;
cv::findContours(imgBinTemp, contours, CV_RETR_EXTERNAL, CV_CHAIN_APPROX_SIMPLE);
```

Zu kurze Konturstücke werden danach noch mit einem Längenkriterium herausgefiltert.

Kontur auf Eckpunkte analysieren

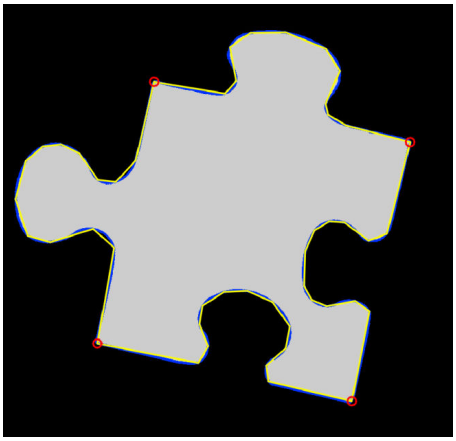
Zuerst wird hier bei jedem Puzzleteil die gefundene Kontur mit der Routine approxPolyDP vereinfacht. Der frei einstellbare Parameter *epsilon*, welcher der Funktion mitgegeben wird, wurde so gewählt, dass nur gerade an den vier Eckpunkten des Grundrechtecks der Winkel zwischen zwei Geraden ungefähr 90° beträgt. Überall sonst soll er kleiner sein. Hier mit den Werten *epsilon* = 0, 2, 10, und 64:

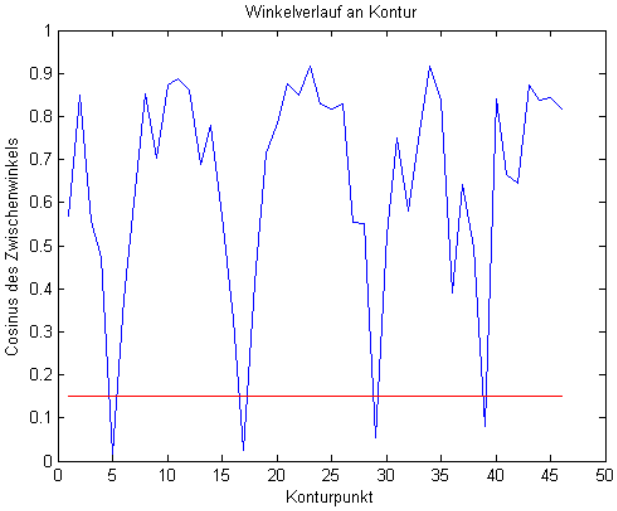


Die blaue Kontur wurde mit der Routine findContours gefunden, das rote Rechteck ist das minimale umfassende Rechteck und die gelbe Linie ist die approximierte Kontur. Wie man gut sehen kann, eignet sich der Algorithmus bei e*psilon* gleich ungefähr 10 am besten, da sich dort die Ecken klar als 90° Winkel von den anderen Winkeln abheben.

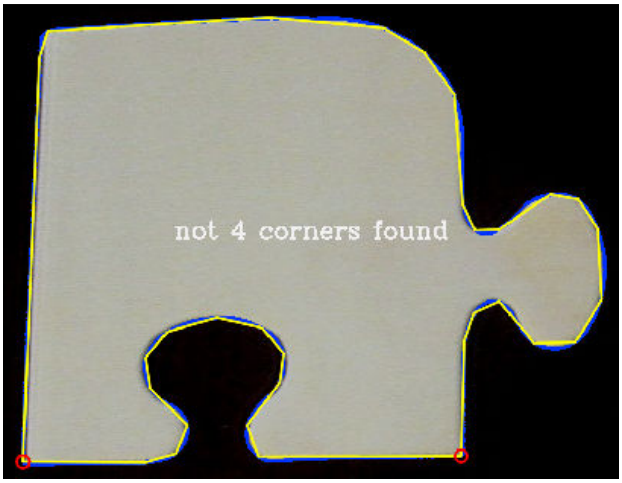
```
// Ecken erkennen, Winkel berechnen zwischen diesem und den zwei nächsten Vertices.
double cosine = std::fabs(angle(pointsApprox[i][(j+2)%pointsApprox[i].size()],
pointsApprox[i][j], pointsApprox[i][(j+1)%pointsApprox[i].size()]));

// wenn Winkel > 81°, als Ecke abspeichern
if(cosine < 0.15)
{
    corners[i].push_back(pointsApprox[i][(j+1)%pointsApprox[i].size()]);
}
```

An dieser vereinfachten Kontur wird jetzt entlanggefahren und wenn der Zwischenwinkel zwischen zwei Konturpunkten eine gewisse Schwelle überschreitet, wird dieser Punkt als Ecke interpretiert.





Werden an einer Kontur einmal nicht vier Eckpunkte gefunden, wird die Kontur verworfen und im Folgenden nicht mehr verarbeitet. Ausserdem wird bei der visuellen Ausgabe ein Text über das entsprechende Puzzleteil geschriebenen:

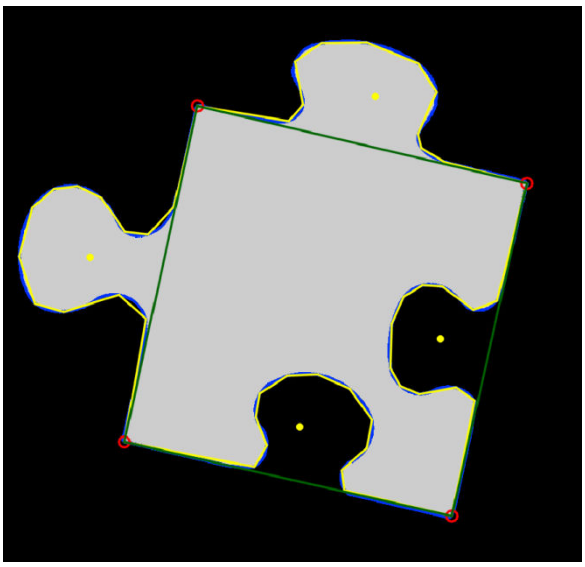


Seitenwände analysieren

Jetzt wird die Kontur jedes Puzzleteils an den Eckpunkten geteilt und so vier Seitenwände pro Puzzleteile extrahiert.

Danach werden die Seitenwände auf ihr Geschlecht analysiert. Herausragende Teile definieren wir als „positiv“, Einbuchtungen als „negativ“ und gerade Wände als „neutral“. Dies dient zur Reduktion der Anzahl Vergleiche, welche so mehr als halbiert wird. Um zu erkennen, ob ein Teil herausragt, wird der Schwerpunkt der Seite berechnet und überprüft, ob dieser innerhalb des Grundrechtecks liegt. Das Grundrechteck wird mit den zuvor gefundenen vier Eckpunkten definiert.

```
// Gender finden: positiv = 1, negativ = -1, neutral = 0
double dist = cv::pointPolygonTest(cv::Mat(corners[i]), sideCentroids[i][j], true);
if(std::fabs(dist) < 5)
{
    // wenn praktisch auf der Linie -> wir als gerade Kante angenommen = Neutral
    genders[i].push_back(0);
}
else
{
    // wenn innerhalb des Grundrechtecks, Gender = Negativ, ansonsten Gender = Positiv
    genders[i].push_back(dist < 0? 1 : -1);
}
```



Die blaue Kontur wurde mit der Routine findContours gefunden, die rot markierten Eckpunkte mit der zuvor erwähnten Methode. Das grüne Quadrat ist das Grundrechteck und die gelben Punkte sind die Schwerpunkte der vier Seitenwände.

Zu vergleichende Seitenwände auswählen

Wir haben uns aus folgenden Gründen entschieden, nicht das ganze Puzzle kreuz und quer zu analysieren:

- Es ist kompliziert, die verschiedenen Vergleichskombinationen zwischen den Seitenwänden aller Puzzleteile zu finden.
- Es ist zeitaufwändig.
- Es kann ohnehin nicht übersichtlich dargestellt werden, vorausgesetzt wir fügen das Puzzle nicht schon ganz zusammen. Aber das ist hier nicht die Aufgabenstellung.

Mit der Maus kann auf dem Originalbild eine Seite angeklickt werden. Wir vergleichen nun diese Seitenwand mit den Seitenwänden aller anderen Puzzleteile im Bild.

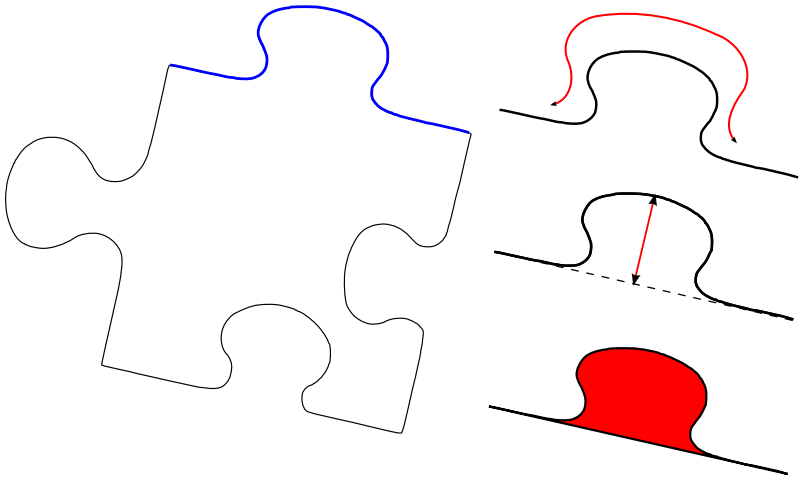
Um einen Mausklick zu registrieren muss der MouseCallback auf das entsprechende Fenster gesetzt werden. Danach kann in der Funktion onMouse die gewünschte Funktion eingefügt werden. In unserem Fall wird dort nur die Position des Klicks in einer globalen Variable gespeichert. Die eigentliche Auswertung geschieht danach in der Hauptroutine. Dort wird die Distanz der Mausposition zu den Puzzleteilen mit der Funktion pointPolygonTest gemessen und das Puzzleteil mit dem kleinsten Abstand zur Mausposition ausgewählt.

Um die Seitenwand des entsprechenden Puzzleteils zu bestimmen, wird die Position noch mit den Seitenwandschwerpunkten verglichen und auch wieder die Seitenwand mit dem kleinsten Abstand gewählt.

Seitenwände vergleichen

Die gewählte Seitenwand wird nur mit Seiten des anderen Geschlechts verglichen. Um die Seitenwände zu vergleichen, haben wir zuerst die Funktion matchShapes verwendet, welche mit den Bild-Momenten zwei Konturen vergleicht. Jedoch hatte diese Funktion eine relativ schlechte Erfolgsrate und wir entschieden uns, einige Features selbst zu definieren:

- Die Gesamtlänge der Seitenwandkontur
- Die Höhe der Ausbuchtung
- Die Fläche der Ausbuchtung



```
// Umfang (Ua/|Ua-Ub|)
double length_basis = cv::arcLength(*basis_side, false);
double length_compare = cv::arcLength(*compare_side, false);
double result_length = length_basis / std::fabs(length_basis - length_compare);

// Tiefe der Ausbuchtung (Ta/|Ta-Tb|)
double arc_basis = std::min(cv::minAreaRect(*basis_side).size.height,
                            cv::minAreaRect(*basis_side).size.width);
double arc_compare = std::min(cv::minAreaRect(*compare_side).size.height,
                              cv::minAreaRect(*compare_side).size.width);
double result_arc = arc_basis / std::fabs(arc_basis - arc_compare);

// Fläche der Ausbuchtung (Fa/|Fa-Fb|)
double area_basis = cv::contourArea(*basis_side);
double area_compare = cv::contourArea(*compare_side);
double result_area = area_basis / std::fabs(area_basis - area_compare);
```

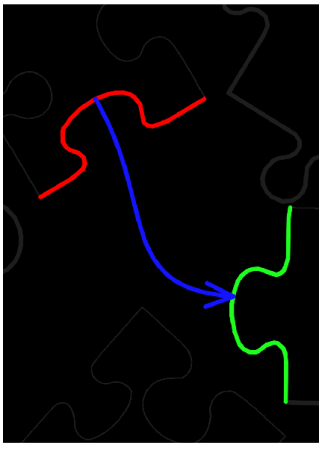
Passende Teile darstellen

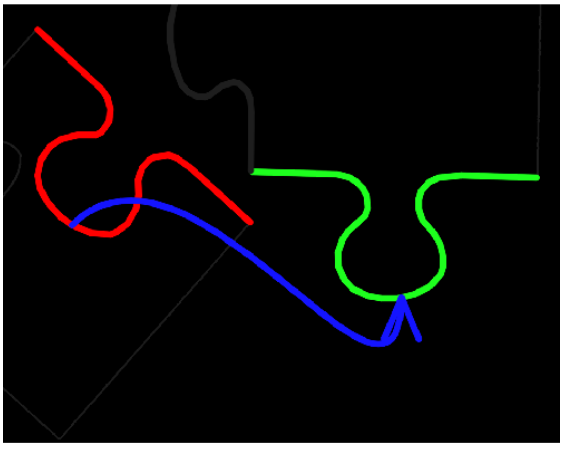
Die Kontur der Basis-Seitenwand wird Rot gezeichnet, die Übereinstimmungen mit den anderen passenden (passend heisst entgegengesetztes Geschlecht) Seitenwänden werden dann mit stärker oder schwächer gefärbtem Grün der anderen Seitenwände angezeigt. Die Seite mit der besten Übereinstimmung wird mit der Originalseitenwand mit einer blauen Bezierkurve verbunden.

```
// Zeichnen der Basis-Seitenwand
std::vector<cv::Point> side = sidesFiltered[piece][piece_side];
const cv::Point *pts = (const cv::Point*) cv::Mat(side).data;
int npts = cv::Mat(side).rows;
cv::polylines(imgContSimilar, &pts,&npts, 1, false, CV_RGB(255,0,0), 10, CV_AA);
```

Um die Bezierkurve zu zeichnen, werden zuerst der Startpunkt und der Endpunkt der best übereinstimmenden Seitenwand bestimmt. Dort wird dann jeweils ein Vektor angesetzt, der von der Seitenwand wegzeigt. Die Länge der Vektoren ist abhängig davon, wie gross die Distanz der beiden zu verbindenden Seitenwände ist. Schlussendlich wird mit einer externen Bibliothek, die wir eingebunden haben (OpenCV unterstützt von Haus aus keine Bezierkurvenberechnung), die Bezierkurve erzeugt und gezeichnet.

```
// Vektor beim Basispuzzleteil:
cv::Point startVector = start - sideCentroids[piece][(piece_side + 2)%4];
startVector *= curveFactor/cv::norm(startVector);

// Vektor beim Zeilpuzzleteil:
cv::Point endVector = end - sideCentroids[maxElement[0]][(maxElement[1] + 2)%4];
endVector *= curveFactor/cv::norm(endVector);
cv::Path2D bezierLine;
bezierLine.restart(start.x, start.y);
bezierLine.curveTo(start + startVector, end + endVector, end);
cv::drawPath2D(imgContSimilar, bezierLine, CV_RGB(20,20,255), -1, 10, CV_AA);

// Pfeil zeichnen
cv::Point orthogonal = cv::Point(-endVector.y, endVector.x)*30*(1/cv::norm(endVector));
cv::line(imgContSimilar, end, end + endVector*70*(1/cv::norm(endVector)) + orthogonal,
        CV_RGB(20,20,255), 10, CV_AA);
cv::line(imgContSimilar, end, end + endVector*70*(1/cv::norm(endVector)) - orthogonal,
        CV_RGB(20,20,255), 10, CV_AA);

cv::imshow("Aehnlichkeit", imgContSimilar);
```





Fazit

Schon von Beginn weg waren wir der Überzeugung, dass das Projekt in OpenCV realisierbar ist. Jedoch haben wir zuerst versucht, die OpenCV-Bibliothek selbst zu kompilieren und Microsoft Visual Studio als Entwicklungsumgebung zu verwenden. Dieses Vorhaben scheiterte und wir entschieden uns, den bereits vorhandenen Qt Creator zu verwenden. Die Software MATLAB kam für uns nicht in Frage, da wir nur die Studentenversion besitzen, bei welcher die Bildverarbeitungs-Toolbox nicht dazugehört. Zudem gingen wir davon aus, dass OpenCV besser dokumentiert ist als die MATLAB Bildverarbeitungsbibliothek (was wir aber nicht beurteilen können, da wir nicht mit MATLAB gearbeitet haben).

Die OpenCV Bibliothek ist sehr mächtig und umfassend. Beinahe für jedes Problem gibt es eine Funktion, die sehr intuitiv nutzbar ist. Auch die Dokumentation ist sehr umfangreich, es gibt eine Menge Beispielprojekte und eine riesige Community im Internet, die Foren, Tutorials und Tipps und Tricks zur Verfügung stellt. Die Herausforderung besteht darin, die richtige Funktion zu finden und sie richtig einzusetzen.

Auch der Qt Creator ist praktisch und mit dem im Kurs behandelten Tutorial fällt der Einstieg leicht.

Unser grösstes Manko lag in der Programmiersprache C++. Es war unser erstes Projekt in dieser Sprache, was vor allem die Handhabung der diversen Datentypen (insbesondere Vektoren) erschwert hat. Zudem haben wir wahrscheinlich vom Aufbau und Ablauf des Programmes eher einen C-Code in C++ geschrieben. Der objektorientierte Ansatz kommt wenig bis gar nicht zum Vorschein.

Schlussendlich sind wir mit dem Ergebnis jedoch zufrieden. Für unser vorgegebenes Puzzleteilchen funktioniert der Algorithmus recht gut. Jedoch könnte er noch mit weiteren Features verbessert werden.

Abschliessend können wir sagen, dass wir einen guten Einstieg in die Bildverarbeitung gehabt haben, für weitere Projekte gerüstet sind und mit diesem Projekt auf ein relativ umfangreiches, funktionierendes Beispiel zurückgreifen können.

Mögliche Erweiterungen

Bessere Binarisierung

Der Binarisierungs-Algorithmus funktioniert noch nicht für alle Bilder. Der nächste Schritt wäre sicher, einen intelligenteren Algorithmus hierfür zu suchen.

Anderer Lösungsansatz

Anstatt Seitenwände mit Features zu vergleichen, könnte man alle Teile auf einer Fläche auslegen und sie so anordnen, dass die „Leerräume“ und Überschneidungen an den Ein- und Ausbuchtungen zwischen den Teilen minimal werden. Diese Methode ist aber mit grösserem Rechenaufwand verbunden.

Nebst Seitenwandanalyse noch Farbanalyse

Man könnte nebst dem Analysieren der Form der Seitenwände auch noch die Farbe vergleichen, welche die Seitenwand gerade am äussersten Punkt hat. Diese muss dann innerhalb einer gewissen Toleranz mit der Farbe auf dem gegenüberliegenden Puzzleteil übereinstimmen.

Puzzle automatisch zusammensetzen

Anstatt nur zu beschreiben, welche Seitenwand wohin gehört, könnte man die Puzzleteile auch noch ausschneiden und so zusammensetzen, dass das Gesamtbild sichtbar wird.

Code Repository

Das ganze Projekt ist auf der Source Code Hosting Website GitHub verfügbar:
https://github.com/baertschi/Puzzle