

Human Centered Computing Bildverarbeitung

Sommersemester 2020

- Dokumentation -

Magic: The Gathering Kartendetektor durch Feature Matching in OpenCv

Robin Connor Schramm

761392

Dozent: Prof. Dr.-Ing. Cristóbal Curio

Eingereicht: 17.07.2020

Inhaltsverzeichnis

1 Einleitung	1
1.1 Magic: The Gathering	1
1.2 Motivation	1
1.3 Ziele	3
1.4 Vorgehensweise	4
2 Grundlagen	6
2.1 Canny Edge	6
2.2 Contour detection	7
2.3 Feature Matching	7
2.4 Bag of visual Words	9
3 Umsetzung	10
3.1 Datensatz	10
3.2 Erkennen von Karten	11
3.2.1 Canny edge detector	11
3.2.2 Contour detection	13
3.3 Feature Matching	14
3.4 Bag of visual Words	15
4 Diskussion und Ausblick	20
4.1 Zukunftsaussichten	20
Literatur	

1 Einleitung

1.1 Magic: The Gathering

Magic: The Gathering¹ (MTG) ist eines der populärsten und profitabelsten Spiele der Welt [CWP12]. Dabei handelt es sich um ein Sammelkartenspiel, bei dem jeder Spieler sein eigenes Deck an Karten verwendet und mit diesem versucht, in rundenbasierten Partien seinen Gegner auf verschiedene Arten zu besiegen. In den Turnierfähigen Varianten des Spiels treten in der Regel zwei Spieler gegeneinander an, wobei auch Formate existieren, bei dem eine beliebige Anzahl an Spielern gleichzeitig an einer Partie mitmachen kann. So ist das Format *Commander* der beliebteste Weg MTG zu spielen, wobei in der Regel vier Spieler an einer Partie von Commander teilnehmen. Ein großer Aspekt eines solchen Mehrspielerspiels ist neben dem bauen eines eigenen Decks und der rundenbasierten Strategie die Soziale Komponente. Das Spielfeld jedes Spielers befindet sich jeweils unmittelbar vor dem Spieler, wobei es sich Prinzipiell um eine beliebige Fläche handeln kann. Diese Fläche ist oft einfach nur ein Tisch. Viele Spieler, die das Spiel öfter spielen, verwenden jedoch auch sogenannte Spielmatten, oft bunt bedruckte Matten, wie in Abbildung 1 zu sehen sind. Karten existieren in verschiedenen Farben und Typen, sind dabei aber in der Regel gleich aufgebaut. Abbildung 1 zeigt den Spielstand einer Partie mit zwei Spielern ungefähr in der Mitte des Spiels.

MTG umfasst zum aktuellen Zeitpunkt ungefähr 16000 funktionell einzigartige Karten. Die Anzahl der existierenden einzigartigen Karten ist jedoch höher, da einige Karten mehrmals in verschiedenen Versionen mit unterschiedlichem Aussehen gedruckt wurden. Die Anzahl der Karten mit einzigartigem Aussehen beträgt somit ungefähr 26000 und ist für dieses Projekt die relevante Metrik, da zwei Karten die funktional gleich sind, jedoch unterschiedlich aussehen für einen Featurebasierten Ansatz verschiedene Karten darstellen.

1.2 Motivation

Das Format Commander ist mein bevorzugter weg, mit Freunden und Bekannten MTG zu spielen. Durch verschiedene Umstände wie eine hohe Distanz

¹MTG Website: <https://magic.wizards.com/>



Abb. 1: Beispiel eines MTG Spielfelds während des Spiels.

zwischen den Wohnorten und die aktuelle Corona Situation, ist es unmöglich zusammen an einem Tisch zu spielen. Die einzige Alternative ist somit das gemeinsame Spielen über das Internet. Es existieren mehrere digitale MTG Ableger, diese verfügen aber entweder nicht über die Kapazität mit mehr als zwei Spielern zu spielen oder kosten zusätzlich Geld, da jedes Deck, das man physisch besitzt, ein weiteres Mal in digitaler Form kaufen müsste. Eine weitere Methode stellt das Spiel über Kamera und Sprachchat dar. Hier können zwar die eigenen Karten verwendet werden, jedoch ist die Interaktion mit den Spielbereichen der anderen Spielern nicht möglich. Das Spielfeld bei einer Partie Commander ist üblicherweise unübersichtlich und die Funktionsweise zwischen verschiedenen Karten komplex. Die Karte eines anderen Spielers in die Hand nehmen um sie zu lesen und zu verstehen ist oft notwendig und beim Spiel über Kamera schlicht nicht möglich. Eine Möglichkeit, auf eine Karte des Gegenübers zu klicken und sie leicht lesen zu können wie in Abbildung 2 dargestellt, wäre sehr hilfreich und würde einen großen Nachteil des Onlinespiels Eliminieren. Das Konzept, dass man auf eine Karte klickt und diese in groß und lesbar angezeigt wird, kommt bereits in den zahlreichen digitalen MTG Ablegern zum Einsatz, wie in Abbildung 3 am Beispiel von MTG Arena zu sehen ist.

1.3 Ziele

Meine Ziele sind es, einen Detektor mit verschiedenen Methoden der Bildverarbeitung zu entwickeln. Dabei sollen generell MTG Karten erkannt, auf dem Kamerabild markiert und anklickbar gemacht werden. Die angeklickte Karte soll mit einer Referenzdatenbank an Karten anhand der Merkmale (Features) verglichen und gefunden werden. Die gefundene Karte soll zum Schluss groß und in guter Qualität auf dem Bildschirm angezeigt werden, wie in Abbildung 2 illustriert wurde. Der Prozess sollte Echzeittauglich sein und während des Spielens einfach verwendet werden können.



Abb. 2: Konzept des MTG Detektors.



Abb. 3: Größere Ansicht einer Karte in MTG Arena.

1.4 Vorgehensweise

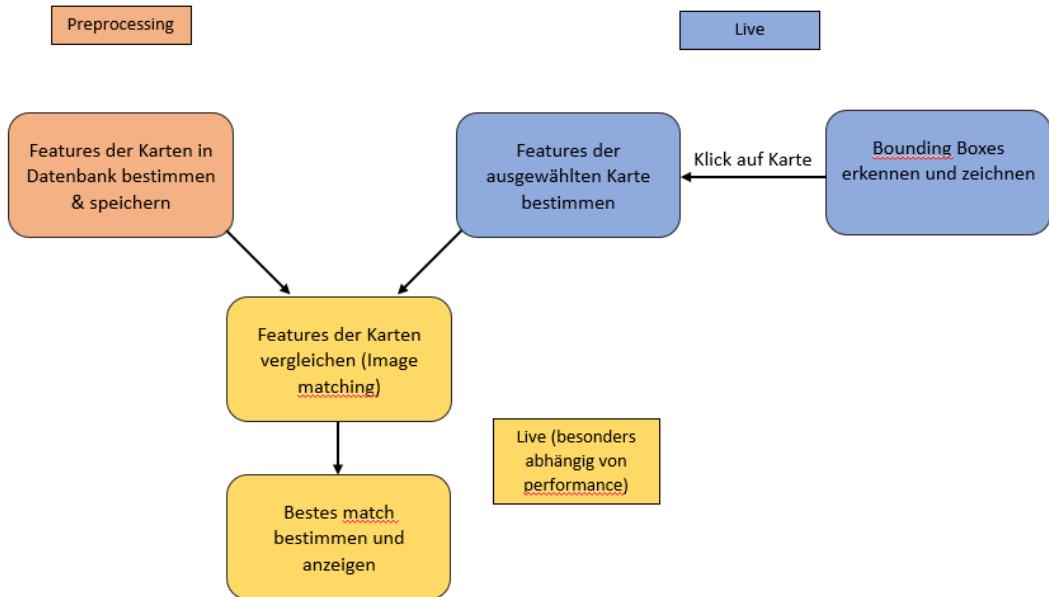


Abb. 4: Geplanter interner Ablauf des Detektors.

Der Prototyp wird in der Sprache Python und der Bibliothek OpenCv entwickelt. Die Umrisse der nötigen Schritte sind in Abbildung 4 dargestellt. Im Schritt der Vorverarbeitung (Preprocessing) werden die Features aller Referenzkarten bestimmt, gefiltert und abgespeichert. Beim Echtzeit (live) Schritt muss zuerst das Kamerabild eines Smartphones oder einer Kamera live auf den Computer übertragen werden. Auf jedes Kamerabild wird voraussichtlich eine Kantenerkennung und Umrisserkennung angewendet um die Positionen und Dimensionen der Karten zu ermitteln. Um jede Karte wird für die Interaktion ein Rechteck (bounding Box) gezeichnet, die den anklickbaren Bereich eingrenzt. Wenn nun der Bereich einer Karte angeklickt wird, werden die Features dieses Bereichs bestimmt und mit den Features aller Referenzkarten verglichen. Dieser Prozess wird abhängig von der Zahl der Karten und Features sehr Zeitintensiv sein und muss zum Schluss vermutlich entsprechend optimiert werden. Wurden alle Features Verglichen wird die beste Zuordnung (das beste *Match*) als die gesuchte Karte gekennzeichnet und dem Nutzer angezeigt. Die ersten Tests werden mit 40 Referenzkarten ausgeführt werden, um schnell verschiedene Methoden auszuprobieren. Sind die Versuche mit 40 Karten erfolgreich, werden weitere Tests mit 1000 zufällig ausgewählten Karten durchgeführt, um die Geschwindigkeit während des live Schritts zu Prüfen und zu optimieren. Wurde eine performante Methode gefunden, eine Karte von 1000 Karten zu erkennen, werden die Versuche auf alle ungefähr 26000 aktuell existierenden

Karten mit einzigartigem Aussehen ausgeweitet. Dieser Schritt ist optional und wird nur bei genügend Zeit durchgeführt werden.

2 Grundlagen

2.1 Canny Edge

Der Canny edge detector wurde im Jahr 1986 von John Canny in [Can86] beschrieben. Dabei handelt es sich um einen mehrstufigen Algorithmus, der verwendet wird, um verschiedene Arten von Kanten in Bildern zu detektieren. Der Canny edge detector wird in diesem Projekt als erste Wahl verwendet, da es sich um einen der am häufigsten genutzten Kantendetektoren handelt [SFM02]. Er lässt sich leicht mit Hilfe von zwei Parametern anpassen und ist bereits in OpenCv implementiert, was die Nutzung vereinfacht. Bei den Anpassbaren Parametern beim Canny edge detector handelt es sich um zwei Schwellenwerte (Thresholds), den *high Threshold* und den *low Threshold*. Sie helfen dabei, Pixel, die durch Rauschen und Farbvariation als Kanten erkannt werden, zu filtern. Wenn der Verlaufswert eines *Kantenpixels* höher als der high Threshold ist, wird dieser als starker Kantenpixel markiert. Wenn der Wert des Pixels zwischen den beiden Thresholds liegt, wird er als schwacher Kantenpixel markiert. Liegt der Pixel unter dem low Threshold wird er nicht als Kante gewertet. Die beiden Schwellenwerte müssen durch Beobachtung und Ausprobieren gesetzt werden und je nach Anwendungsfall angepasst werden. Das Ergebnis des Canny edge detectors ist ein Bild bestehend aus einem schwarzen Hintergrund und den detektierten Kanten in weiß wie in 5 am Beispiel von MTG Karten zu sehen ist.

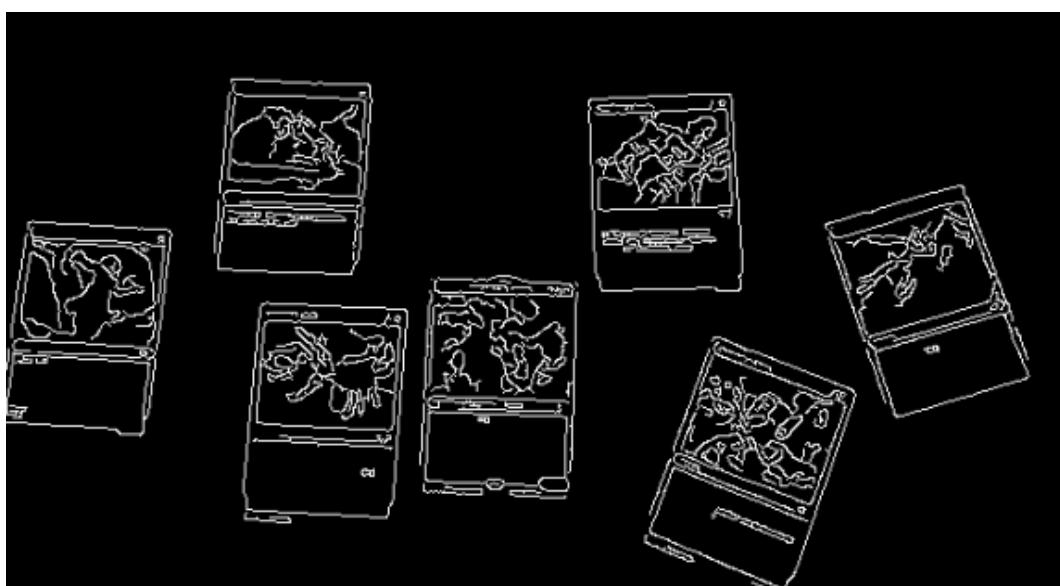


Abb. 5: Canny edge Repräsentation am Beispiel von MTG Karten.

2.2 Contour detection

Die contour detection, auch Umrisserkennung, wird in diesem Projekt auf das Ergebnis des Canny edge detectors angewendet, um geschlossene Flächen innerhalb des Bildes zu finden. Verwendet wird die Methode `findContours()` der OpenCv Bibliothek, welche auf dem von Suzuki et al. in [SB85] vorgestellten Vorgehen basiert. Die Funktion benötigt als Eingabe ein zweifarbiges Bild, wobei praktischerweise direkt die Ausgabe des Canny edge detectors als Eingabe verwendet werden kann. Die `findContours` Funktion beinhaltet des Weiteren verschiedene Modi in zwei Parametern. Dabei handelt es sich zum einen um die *Retrieval Modes*, die Beschreiben, wie die Umrisse erlangt werden sollen. Dabei können die nur die absolut äußersten Umrisse oder auch mehrere Hierarchien in verschiedenen Formen zurückgegeben werden. Zum anderen handelt es sich um die *Contour Approximation Modes*, die Beschreiben, wie die Umrisse approximiert werden sollen. Dabei können zum Beispiel alle Konturpunkte oder nur rechteckige Umrisse mit vier Punkten gespeichert werden. Die im Projekt verwendetet Modi sind die RETR_EXTERNAL und CHAIN_APPROX_NONE Modi. CHAIN_APPROX_NONE bedeutet, dass einfach alle Punkte des Umrisses gespeichert werden, was wichtig ist, da die Karten zwar annährend rechteckig sind, jedoch über abgerundete Ecken verfügen und dadurch nicht durch ein einfaches Rechteck repräsentiert werden können. Beim RETR_EXTERNAL Modus werden nur die äußersten Umrisse einer Hierarchie gespeichert. Dieser Modus ist bei den MTG Karten sehr hilfreich, da die geschlossenen Umrisse innerhalb der Karte für das Detektieren der Karte nicht relevant sind.

2.3 Feature Matching

Feature Matching ist eine im Bereich der Bildverarbeitung häufig eingesetzte Methode um z.B. Objekte in Bildern zu erkennen oder leicht unterschiedliche Bilder einander zuzuordnen. Dazu werden die Merkmale (Features) eines Bilds detektiert, indem z.B. die Position und gegebenenfalls die Ausrichtung von Ecken, Kanten und anderen interessanten Punkten in einem Bild extrahiert werden. Der in diesem Projekt verwendete Algorithmus um Features zu erkennen ist der von Rublee et al. in [RRKB11] vorgestellte Oriented FAST and Rotated BRIEF (ORB) Algorithmus. ORB wurde als eine schnellere und effiziente Alternative zum populären scale-invariant feature transform (SIFT) Algorithmus entwickelt. Ein weiterer Grund neben der Geschwindigkeit, warum

ORB in diesem Projekt verwendet wird, ist die freie Verfügbarkeit von ORB und die in OpenCv integrierte Implementation. SIFT dagegen wurde Patentiert und ist so schwieriger verfügbar. Die Ausgabe des ORB Feature Detektors besteht aus Schlüsselpunkten (Keypoints) und Desktiptoren (Descriptors). Ein Keypoint beinhaltet dabei Metadaten des Features wie Position, Skalierung und Größe. Ein Descriptor beinhaltet hingegen die visuelle Beschreibung der Stelle an der sich das Feature befindet und wird für den Vergleich der Ähnlichkeit von zwei Bildern verwendet. In Abbildung 6 sind einige Features auf zwei Karten als bunte Punkte dargestellt. Die Linien stellen eine Verbindung zwischen jeweils zwei Features dar und werden im folgenden als *Match* bezeichnet.

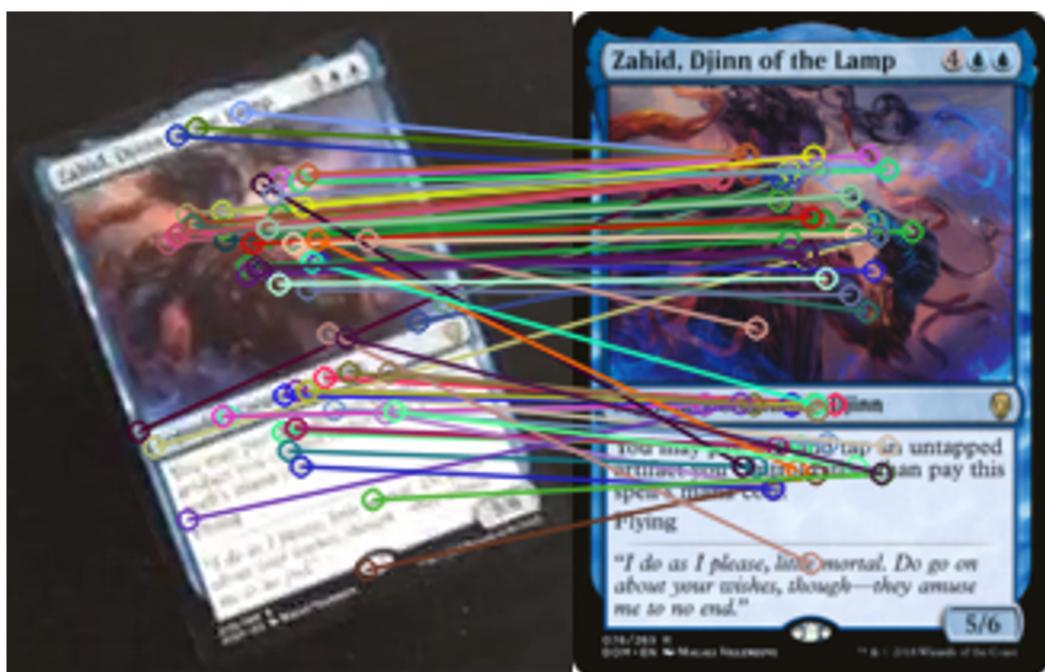


Abb. 6: Feature Matching am Beispiel eines Kamerabilds (links) und eines Referenzbilds (rechts) von einer MTG Karte. Die Punkte stellen Features dar, die Linien Matches zwischen jeweils zwei Features.

Ein einfacher Weg, Feature Matching zu betreiben, ist über einen Brute-Force Matcher (BF-Matcher). Dabei werden die Features des gesuchten Bilds mit jedem Set an Features in der Datenbank verglichen. Eine einfach Metrik, um die am besten passenden Bilder zu finden, ist die euklidische Distanz zwischen einzelnen Features die einander zugeordnet wurden. Diese Herangehensweise liefert bereits brauchbare Ergebnisse, kann aber noch verbessert werden, da oft einige Features nur durch zufälliges Rauschen zustande kommen und für verfälschte Ergebnisse sorgen können. Eine verbreitete Methode um Matches zu filtern bzw. die *guten* Matches von denen zu Unterscheiden, die durch Rauschen zustande kamen, ist Lowe's ratio Test, der in [Low04] beschrieben wurde. Für

Lowe's ratio Test muss für jedes Feature im gesuchten Bild (Query), die beiden besten Matches mit dem Referenzbild gefunden werden. Wenn die beiden besten Matches eines Features sehr ähnlich sind, werden sie beide eliminiert. Die Ähnlichkeit wird dabei durch eine selbst festgelegte Konstante gesteuert. Es wird angenommen, dass das Feature aus dem Query genau ein äquivalent in dem Bild aus der Datenbank hat, wodurch alle anderen Matches folglich äquivalent zu zufälligem Rauschen haben. Das Match mit der niedrigsten Distanz ist dabei das *gute* Match. Sollten sich beide Matches nun aber sehr ähnlich sein, ist somit das "gute" Match nicht von zufälligem Rauschen zu unterscheiden, wodurch es keinen Mehrwert bringt. Ist dies der Fall werden beide Matches verworfen. Im Prinzip muss also genug Unterschied zwischen dem besten und dem zweitbesten Match bestehen, damit es sich bei dem besten Match um ein relevantes Match handelt. Nachdem die Matches alle gefiltert wurden, ist das Bild, bei dem die meisten relevante Matches vorkamen, das vorgeschlagene Bild, das dem Query entspricht.

2.4 Bag of visual Words

Ein Bag of visual Words (BoVW) Ansatz für das Feature Matching wurde als schnellere Alternative zum Brute-Force Matcher für das Projekt vorgeschlagen und gegen Ende teilweise implementiert. Der BoVW Ansatz wurde auf der Grundlage des Bag of Words Ansatz in der Textverarbeitung entwickelt. Dabei wird ein Vokabular aus Bildern der Referenzdatenbank aufgebaut, welches aus gruppierten Histogrammen der Features der Referenzbilder besteht. Soll nun ein Bild mit Hilfe des BoVW gesucht werden, muss nicht jedes Feature einzeln verglichen werden. Stattdessen wird von dem Query ebenfalls ein Histogramm erstellt und mit allen Gruppenhistogrammen verglichen. Dieser Ansatz ist potentiell schneller als ein BF-Matcher und kann in Kombination mit anderen Matchern verwendet werden, um die Genauigkeit weiter zu steigern.

3 Umsetzung

3.1 Datensatz

Um eine große Zahl an Referenzbildern von Magic: The Gathering (MTG) Karten zu gelangen, wurde die Website Scryfall¹ verwendet. Dabei handelt es sich um eine MTG Karten Suchmaschine, in der jede existierende MTG Karte sowohl in Textform als auch als Bild in verschiedenen Größen, Versionen und Sprachen einsehbar ist. Scryfall bietet zusätzlich eine REST ähnliche API² an, die den einfachen programmatischen Zugriff auf die gesamte Scryfall Datenbank mit REST anfragen ermöglicht.

Zuerst wurde eine JSON Datei mit Metadaten aller englischsprachigen MTG Karten mit einzigartigem Aussehen von Scryfall heruntergeladen. Die Datei enthielt zum Zeitpunkt des Projekts 26389 Zeilen, wobei jede Zeile eine einzigartige Karte darstellt und viele Informationen zu dieser Karte beinhaltet, wie der Englische Name der Karte, Datum des Erscheinens, URLs zu Bildern der Karte und viele weitere für das Projekt irrelevante Informationen. Eine stark komprimierte Version einer einzelnen Zeile mit den relevanten Informationen ist im Codeauszug ?? für die Karte *Fury Sliver* dargestellt. Die einzigen Karten, die vorerst in diesem Projekt nicht erkannt werden können, sind spezielle Karten, die je auf vorder- und Rückseite relevanten Text und Artwork besitzen (sog. double faced cards). Diese Karten zählen im Spiel als eine einzelne Karte, müsste für diese Projekt jedoch als zwei separate Karten behandelt werden, was mit zusätzlichem Aufwand verbunden wäre. Die Anzahl dieser doppelseitigen Karten beträgt zum Zeitpunkt dieser Dokumentation 111. Die doppelseitigen Karten werden gegebenenfalls in einer Zukünftigen Version des Projekts eingebunden.

```
1 { "object": "card", "name": "Fury Sliver",
2   "lang": "en", "released_at": "2006-10-06",
3   "large": "https://img.scryfall.com/cards/large/front
 /0/0/0000579f-7b35-4ed3-b44c-db2a538066fe.jpg
 ?1562894979", [...] },
```

Codeauszug 3.1: Relevante Teile für das Projekt der JSON Datei mit Metadaten der Karte *Fury Sliver*

¹Scryfall Website: <https://scryfall.com>

²Scryfall API Dokumentation: <https://scryfall.com/docs/api>

Die wichtigsten Attribute in jeder Zeile sind der Name der Karte, die Erweiterung, in der die Karte erschienen ist, die Sprache der Karte sowie die URL zu einer Hochauflösenden Bilddatei der Karte, hier im Attribut "large". Diese Informationen werden alle, ausgenommen der URL, in einen String gepackt, der später auch als ID für die Karten verwendet wird, dargestellt in Codeauszug 3.2. Mit Hilfe des Strings können nun die gewollte Anzahl an Karten heruntergeladen werden, indem die ersten n Zeilen der JSON Datei gelesen werden.

```
1 '5dn_Auriok_Salvagers_en.jpg'
```

Codeauszug 3.2: Beispiel für einen generierten Karten-Dateinamen.

3.2 Erkennen von Karten

Die wichtigste Voraussetzung für den Kartendetektor ist das Erkennen von MTG Karten auf einem beliebigen Hintergrund. Da jede MTG Karte die selben Maße und Dimensionen hat und in der Regel jeweils einen einfarbigen Rand hat, war der erste Ansatz für das Erkennen der Karten eine Kombination aus John Canny's Kantenerkennung (Canny edge detection) [Can86] und einer Umrisserkennung (contour detection). In Abbildung 7 sind die verschiedenen Schritte der Kartenerkennung visualisiert. Sektion 5 ist zwei mal zu sehen, da die verwendete Methode mehrere Fenster in OpenCv darzustellen sechs Fenster benötigt, hier jedoch nur fünf verwendet wurden.

Canny edge detector

Der Canny edge detector wurde in Kapitel 2.1 bereits erklärt. OpenCv verfügt bereits eine Canny Implementation, welche mit einem unscharf gemachten, grauen Frame und den beiden Thresholds aufgerufen werden kann. Die Thresholds wurden Anfangs auf zufällige Startwerte gesetzt und per Hand angepasst. Für die live Anpassung wird ein extra Fenster 8 für Parameter erstellt. Als robuste Werte für die Thresholds wurden nach einigem Testen 121 für den maximalen und 72 für den minimalen Threshold gewählt. Das Ergebnis ist in Abbildung 7 in Sektion 3 dargestellt.



Abb. 7: Alle Schritte der Kartenerkennung visualisiert. 1: Originales Kamerabild. 2: Unscharfe Version von 1. 3: Ergebnis der Canny edge detection. 4: Ergebnis der contour detection. 5: Blau: Äußerste Umriss aus 4.; Grün: Bounding Boxes um die Umrissse. (5): siehe 5.

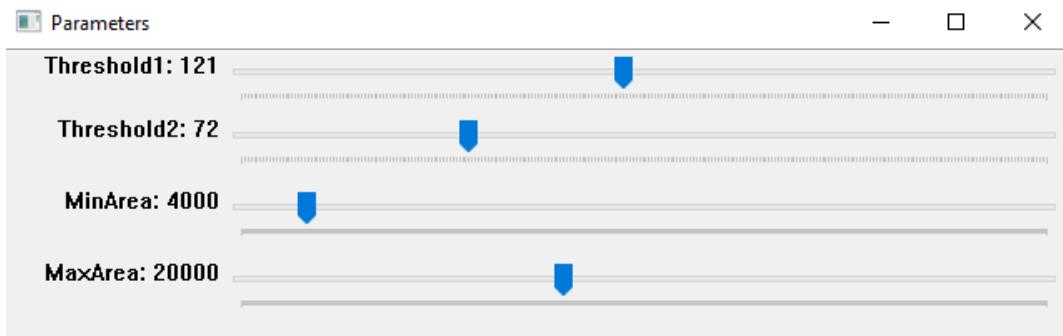


Abb. 8: Parameter für die live Kartenerkennung.

```

1 frameBlur = cv.GaussianBlur(frame, (7,7), 1)
2 frameGray = cv.cvtColor(frameBlur, cv.COLOR_BGR2GRAY)
3
4 threshold1 = cv.getTrackbarPos("Threshold1", "Parameters")
5 threshold2 = cv.getTrackbarPos("Threshold2", "Parameters")
6 #Edge Detection
7 frameCanny = cv.Canny(frameGray, threshold1,
threshold2)

```

Codeauszug 3.3: Canny edge detection & bounding Boxes in OpenCv

Auf schwarzem Hintergrund lieferte dieser Ansatz sehr gute Ergebnisse, für Karten auf bunten Spielmatten, wie sie häufig eingesetzt werden, ist dieser Ansatz jedoch ungeeignet.

Contour detection

Der Schritt nach der Kantenerkennung ist das Detektieren der Umrisse der Karten. Dazu werden zuerst die in Sektion drei der Abbildung 7 erkannten Kanten dicker gemacht, damit die Umrisse besser erkennbar sind. Die dickeren Umrisse sind in Sektion vier in Abbildung 7 zu sehen.

```

1 contours, hierarchy = cv.findContours(inputFrame, cv.
2                                     RETR_EXTERNAL, cv.CHAIN_APPROX_NONE)
3
4 for contour in contours:
5     area = cv.contourArea(contour)
6     if minArea < area < maxArea:
7         cv.drawContours(frameContour, contour, -1,
8                         (255, 0, 0), 7)
9         #contour lenght - true=contour closed
10        perimeter = cv.arcLength(contour, True)
11        approx = cv.approxPolyDP(contour, 0.02 *
12                                  perimeter, True)
13
14        #bounding Box
15        x, y, w, h = cv.boundingRect(approx)
16        cv.rectangle(frameContoured, (x, y), (x+w, y+
17                                              h), (0, 255, 0), 3)
18        boundingBoxes.append(boundingBox(x, y, h, w))

```

Codeauszug 3.4: Contour detection in OpenCv

Der *RETR_EXTERNAL* Modus wurde für die *findContours* Funktion als retrieval Methode gewählt, um die absolut äußersten Umrisse eines Objekts zu erlangen, da nur der äußere Umriss einer Karte für eine bounding box nötig ist und sonst verschiedene Ebenen innerhalb einer Karte als eigene Karten erkannt werden können. Der *CHAIN_APPROX_NONE* Modus wurde für die *findContours* Funktion als Näherungsalgorithmus gewählt, da MTG Karten zwar annäherungsweise Rechteckig sind, jedoch abhängig vom Hintergrund

mit abgerundeten Ecken erkannt werden, wie in den Abbildungen 5 und 7 zu sehen ist.

Anhand der berechneten Umrisse werden dann bounding boxes um die erkannten Karten berechnet und später gezeichnet. Das Ergebnis ist in Abbildung 7 in Sektion 5 zu sehen. Dabei sind die berechneten Umrisse blau gekennzeichnet und die bounding boxes grün gezeichnet. In einer weiteren Funktion wird nun auf die Eingabe des Nutzers gewartet. Wenn sich der Mauszeiger bei einem Klick innerhalb der Koordinaten einer bounding box befindet, können weitere Funktionen aufgerufen werden. Zur weiteren Verarbeitung wird der Ausschnitt des Frames innerhalb der bounding box zuerst als Bilddatei abgespeichert.

3.3 Feature Matching

Wie in Kapitel 2.3 beschrieben, wird in diesem Projekt der ORB Algorithmus verwendet, um die Merkmale (Features) der Karten zu bestimmen.

Zuerst werden die ORB Keypoints und Descriptors von des gesuchten Bildes (Query) berechnet. Der Query muss dafür nicht weiter bearbeitet werden und kann direkt verwendet werden. Als nächstes werden alle Referenzbilder vor verarbeitet, dazu werden die Bilder leicht unscharf gemacht und auf 20% der Größe verkleinert, damit sie dem Query ähnlicher sind. Anschließend werden die Keypoints und Descriptors für alle Referenzbilder berechnet und in einem Array gespeichert.

Um nun die Descriptors des Query und den Referenzbildern zu vergleichen, wurde im ersten Schritt ein Brute-force Matcher (BF-Matcher) verwendet. Bei der BF-Matcher Implementation handelt es sich um den k nearest neighbours (knn) Matcher wobei $k=2$ gesetzt wurde, was bedeutet, dass für jedes Feature die beiden Matches mit den kürzesten Distanzen gespeichert werden. Die Matches werden anschließend durch Lowe's ratio Test gefiltert. Die Konstante für Lowe's ratio Test wurde nach einigem Ausprobieren auf 0.85 gesetzt. Im folgenden Codeausschnitt 3.5 ist der Ablauf des BF-Matchers und Lowe's ratio Test zu sehen.

```

1 bf = cv.BFMatcher()
2 lowe_ratio = 0.85
3 good = []
4 maxMatches, bestMatch, bestMatches = 0, 0, 0
5
6 for i in range(len(bagOfImages)):
7     matches = bf.knnMatch(queryDes, bagOfDescriptors[i],
8                           k=2)
9     for m,n in matches:
10         if m.distance < lowe_ratio*n.distance:
11             good.append([m])
12     print(i, " Good: ", len(good))
13     if len(good) > maxMatches:
14         maxMatches = len(good)
15         bestMatch = i
16         bestMatches = good.copy()
17     good.clear()

```

Codeauszug 3.5: Brute-Force Matching und Lowe's ratio Test

Bei den ersten Versuchen mit 40 Karten lieferte der BF-Matcher sehr schnell gute Ergebnisse. Ein großes Problem dabei ist aber die Skalierung der Größe des Referenzdatensatzes. Bei den Tests mit 1000 Karten dauerte ein Matching Vorgang mehrere Sekunden, was während der Echtzeitanwendung noch Nutzbar war, jedoch schon stark an der Grenze ist. Das macht den BF-Matcher für die live Verwendung mit allen 26000 Karten ungeeignet.

3.4 Bag of visual Words

Eine effizientere Alternative zum direkten Matching mit jeder einzelnen Karte durch Brute-Force Matching ist der Bag of visual Words (BoVW) Ansatz. Der Ansatz wurde gegen Ende des Projektzykluses verfolgt und weitestgehend implementiert, ist jedoch zum Zeitpunkt der Dokumentation noch unvollständig und teilweise fehlerhaft. Im Gegensatz zur oben beschriebenen Methode ist hier der offline Schritt um einiges länger und komplexer, was im Gegenzug zur schnelleren und hoffentlich besseren Performance im live schritt führt. Zuerst muss, wie beim Image Matching Ansatz, einer Feature Datenbank aufgebaut

werden. Dazu werden für jedes Referenzbild die Orb Features und Descriptors berechnet und in eine Datei gespeichert. Dieser Prozess wurde für Testzwecke mit 1000 Referenzkarten durchgeführt dauert ungefähr eine Minute und liefert eine .pck Datei mit 250 MB. Zusätzlich wurden 20 zufällige Bilder aus dem Datensatz kopiert, die später zum Testen des BovW verwendet werden. Von diesen 20 Bildern wurden ebenfalls die Features und Descriptors berechnet und in einer Datei abgespeichert. Für das Speichern von Python Dictionaries wird das Modul Pickle³ verwendet. Pickle wird verwendet, um Python Objekte als byte stream zu serialisieren und wieder zu deserialisieren. Der Bytestream wird dabei von einer Binärdatei wieder in eine Objekthierarchie umgewandelt, wodurch die Daten leicht im Code weiter verwendet werden können. Für alle Bilder werden zwei Dictionaries angelegt, einmal für die Keypoints und einmal für die Descriptors. Als Schlüssel dienen dabei jeweils der Name der Karte wie sie im Dateisystem vorliegt und wie bereits im Kapitel Datensatz beschrieben wurde. Beim erneuten Laden der Karten werden aus den Dictionaries die Descriptorwerte und die Namen gezogen und aus einer Funktion zurückgegeben. Dadurch lassen sich alle Zwischenergebnisse des kompletten Prozesses Schritt für Schritt serialisieren um an den notwendigen Stellen wieder verwendet werden kann. Dieser kleine Mehraufwand spart immens viel Zeit beim Testen sowie später in der Anwendung.

Nachdem die Features der Referenzbilder bestimmt wurden, wird das visuelle Vokabular gebildet, der Zugehörige Vorgang ist im Codeauszug 3.6 dargestellt. Dabei kommen Clusteringverfahren zum Einsatz, um zufällig Mittelpunkte von einer vom Nutzer festgelegten Anzahl an Clustern zu bestimmen. Hierzu wurde die kMeans Methode von OpenCv verwendet. Zuvor müssen die Deskriptoren in einen gemeinsamen Array umgewandelt werden sowie auf den double Typ umgewandelt werden, um die Anforderungen des kMeans Algorithmus zu erfüllen. Die Anzahl der Cluster wurde Anfangs auf 20 gesetzt, hier wurde jedoch noch nicht getestet, was ein optimalen Wert darstellen würde. Die Cluster Mitten wurden, wie die anderen Zwischenergebnisse, in einer Pickle Datei für späteren Zugriff abgespeichert.

```

1 def kMeans(dataPath):
2     dsc, names = loadFeatureDatabase(dataPath)
3
4     dscFlat = np.vstack(dsc)
5     dscFlat = np.array(dscFlat, dtype="double")
6     clusters = KMeans(20)

```

³Pickle Python doc: <https://docs.python.org/3/library/pickle.html>

```

7     clusters.fit(dscFlat)

8

9     skiCenters = clusters.cluster_centers_
10    with open("data/clusters", 'wb') as file:
11        pickle.dump(skiCenters, file)

```

Codeauszug 3.6: kMeans clustering für Deskriptoren

Nachdem die Cluster Mittelpunkte definiert wurden, können Histogrammrepräsentationen für die Bilder erstellt werden. Der Programmcode in Codeauszug 3.8 beschreibt das erstellen der Histogramme und wurde unter Zuhilfenahme von zwei Tutorials^{4,5} erstellt. Da in den Tutorials sklearn anstatt OpenCv verwendet wird, mussten einige Zeilen angepasst werden. Das Erstellen der Histogramme ist der Zeitaufwändigste Schritt des Prozesses und dauert, abhängig von der Anzahl an Clustern und Bildern, bis zu 15 Minuten für 100 Referenzbilder.

```

1 def createHistograms(dsc, skiCenters, names):
2     dictFeature = {}
3     for i in range(0, len(dsc)):
4         print(i)
5         category = []
6         for j in range(len(dsc[i])):
7             histogram = np.zeros(len(skiCenters))
8             for eachFeature in dsc[i][j]:
9                 ind = findIndex(eachFeature,
10                     skiCenters)
11                 histogram[ind] += 1
12             category.append(histogram)
13             dictFeature[names[i]] = category
14
15
16     return dictFeature
17
18
19 def findIndex(featureVal, centers):
20     featureVal = featureVal.reshape(-1, 1)
21     minDist = abs(distance.euclidean(featureVal,
22         centers[0]))
23     label = 0

```

⁴<https://towardsdatascience.com/bag-of-visual-words-in-a-nutshell-9ceea97ce0fb>⁵<https://medium.com/@aybukeyalcinerr/bag-of-visual-words-bovw-db9500331b2f>

```

20     for i in range(0, len(centers)):
21         dist = abs(distance.euclidean(featureVal,
22                                     centers[i]))
23         if dist < minDist: label = i
24     return label

```

Codeauszug 3.7: Erstellen der Histogrammrepräsentationen aller Bilder

BovW Daten				
Anzahl Cluster	Auriok Salvagers	Angelsong	Knight-Captain of Eos	Gesamt Korrekt
20	0	0	0	0
60	2	3	10	15
70	0	0	0	0
80	4	2	2	8
100	6	3	8	17
109	9	95	95	285
120	0	0	0	0
200	0	0	0	0

Tab. 3.1: Ergebnisse des BovW Trainings mit verschiedenen Clustergrößen. Die Werte zu den Kartennamen und Gesamt ist die Anzahl der korrekten Vorhersagungen von 128 Möglichen Features.

Es stellte sich heraus, dass 109 die beste Anzahl an Clustern ist. 109 entspricht der Anzahl an Referenzkarten, die in diesem Test verwendet wurden. Um die Zahl zu prüfen, wurde der Test mit sechs weiteren Karten geprüft. Bis auf die Karte *Auriok Salvagers* waren die Ergebnisse aller Karten zufriedenstellend. Die Werte sind im folgenden zu betrachten:

```

1 korrekt: 719, gesamt: 1144,
2 '5dn_auriok_salvagers_en.jpg': [9, 120],
3 'ala_angelsong_en.jpg': [95, 128],
4 'ala_knight-captain_of_eos_en.jpg': [95, 128],
5 'hou_plains_en.jpg': [77, 128],
6 'ice_thermokarst_en.jpg': [92, 128],
7 'isd_avacynian_priest_en.jpg': [90, 128],
8 'kld_ceremonious_rejection_en.jpg': [93, 128],
9 'ktk_scoured_barrens_en.jpg': [89, 128],
10 'ktk_surrak_dragonclaw_en.jpg': [79, 128]

```

Codeauszug 3.8: Ergebnisse des BovW Trainings mit 9 Karten und 109 Clustern. Der linke Wert sind die Korrekt geschätzten Features und der Rechte Wert sind die Features gesamt.

Auffällig bei dem Test waren zwei Karten: Zum einen die Karte *Auriok Salvagers*, deren Vorhersagen bei fast jedem Test mit Abstand die schlechtesten

waren. Zum anderen die Karte *Manabarbs*. Wenn Manabarbs im Referenzdatensatz vorhanden war, wurde bei jeder Karte ausschließlich vorhergesagt, dass es sich um die Karte Manabarbs handele. Beide Karten haben an sich nichts für den Menschen erkennbares besonderes an sich, diese Beobachtungen sorgten jedoch für Verwirrung.

Aufgrund der begrenzten Zeit konnten nicht noch mehr Clustergrößen und Datensätze getestet werden, die Ergebnisse sind jedoch bereits vielversprechend. Des Weiteren wurde der BovW Ansatz nur mit statischen Testbildern durchgeführt, da der BovW Code ebenfalls aus Zeitgründen nicht in die Live Ansicht eingebunden werden konnte.

4 Diskussion und Ausblick

Es wurde ein Programm entwickelt, das mit Hilfe von OpenCv und feature matching Magic: The Gathering (MTG) Karten auf Kamerabildern erkennen kann. Das Projekt kann zum Teil als erfolgreich bezeichnet werden. Die Kartenerkennung durch Brute force Matching funktioniert sehr gut bei einer kleinen Anzahl (40) von Referenzkarten auf einem einfarbigen, nicht weißen Hintergrund. Bei größeren Datensätzen mit mehr als 1000 Karten ist der Ansatz jedoch so langsam, dass er nicht für die live Anwendung verwendet werden kann und ist dabei sehr ungenau. Die Notwendigkeit für einen einfarbigen Hintergrund kommt von der Funktionsweise des Canny edge detectors, der auf Hintergründen mit Mustern die Kanten von Karten nicht von den Kanten des Hintergrunds unterscheiden kann. Eine mögliche Alternative wäre eine Art der Flächenerkennung, wo die Position der Karten nicht anhand ihrer Kanten sondern, anhand ihrer Features bestimmt wird. Dieser Ansatz wäre ebenfalls vielversprechend, da die Features für das Feature matching ohnehin detektiert werden. Eine Alternative zum Brute force Matcher, die für die Geschwindigkeit der Detektion während der live Anwendung besser sein könnte, ist der im vorigen Kapitel beschriebene Ansatz des Bag of visual Words (BoW). Der BoW wurde größtenteils implementiert, jedoch nicht in der live Anwendung und nicht mit dem kompletten Datensatz von ca. 26000 Karten getestet, ist jedoch vielversprechend. Die Auswahl der besten erkannten Features ist zusätzlich ein Aspekt, an dem Veränderungen Verbesserungen bewirken könnten. So werden viele Features im Text der Karte anstatt dem Bild erkannt. Eine weitere Alternative zu dem Brute force Matcher könnte ein FLANN basierter Matcher bessere Ergebnisse bewirken. Die plane_tracker Anwendung der OpenCv Beispiele verwendet einen ORB detector zusammen mit einem FLANN Matcher und liefert beim Test mit MTG Karten vielversprechende Ergebnisse.

4.1 Zukunftsaussichten

Da es sich bei dem Projekt noch um einen Prototyp handelt, sind zahlreiche Verbesserungen möglich. Eine wichtige Erweiterung wäre das Einbringen der Doppelseitigen Karten, die der Einfachheit halber aus dem Datensatz ausgeschlossen wurden. Das Einbinden und Testen des Bag of visual Words ist ebenfalls eine naheliegende Erweiterung des Prototyps. Des Weiteren können,

wie Angesprochen, noch weitere Alternativen zur Kantenerkennung und zum Feature matching Implementiert werden. Die Verwendung und Anpassung des Projektes für weitere Karten basierte Spiele ist ebenfalls denkbar, da theoretisch nur der Datensatz und wenige Parameter für ein komplett neues Spiel angepasst werden muss.

Literatur

- [Can86] CANNY, J: A Computational Approach to Edge Detection. In: *IEEE Transactions on Pattern Analysis and Machine Intelligence* PAMI-8 (1986), nov, Nr. 6, S. 679–698. – ISSN 1939–3539
- [CWP12] COWLING, P. I. ; WARD, C. D. ; POWLEY, E. J.: Ensemble Determinization in Monte Carlo Tree Search for the Imperfect Information Card Game Magic: The Gathering. In: *IEEE Transactions on Computational Intelligence and AI in Games* 4 (2012), Nr. 4, S. 241–257
- [Low04] LOWE, David G.: Distinctive Image Features from Scale-Invariant Keypoints. In: *International Journal of Computer Vision* 60 (2004), Nr. 2, S. 91–110. – ISSN 1573–1405
- [RRKB11] RUBLEE, Ethan ; RABAUD, Vincent ; KONOLIGE, Kurt ; BRADSKI, Gary: ORB: An efficient alternative to SIFT or SURF. In: *Proceedings of the IEEE International Conference on Computer Vision* (2011), S. 2564–2571. ISBN 9781457711015
- [SB85] SUZUKI, Satoshi ; BE, Keiichi A.: Topological structural analysis of digitized binary images by border following. In: *Computer Vision, Graphics and Image Processing* 30 (1985), apr, Nr. 1, S. 32–46. – ISSN 0734189X
- [SFM02] SHARIFI, M. ; FATHY, M. ; MAHMOUDI, M. T.: A classified and comparative study of edge detection algorithms. In: *Proceedings. International Conference on Information Technology: Coding and Computing*, 2002, S. 117–120

Erklärung zur Abgabe einer Prüfungsleistung

Ich versichere, dass ich

- den „Leitfaden für gute wissenschaftliche Praxis im Studiengang huc“¹ kenne und achte,
- die von mir eingereichten Dokumente und Artefakte selbstständig ohne Hilfe Dritter verfasst habe,
- alle benutzten Quellen und Hilfsmittel - dazu zählen auch sinngemäß übernommene Inhalte, leicht veränderte Inhalte sowie übersetzte Inhalte - in Quellenverzeichnissen, Fußnoten oder direkt bei Zitaten angegeben habe,
- alle wörtlichen und sinngemäßen Zitate von Textstücken, Tabellen, Grafiken, Fotos, Quellcode usw. aus fremden Quellen als solche gekennzeichnet und mit seitengenauen Quellenverweisen versehen habe,
- die von mir eingereichten Dokumente und Artefakte noch nicht in dieser oder ähnlicher Form in einem anderen Kurs vorgelegt worden sind und ich
- alle nicht als Zitat gekennzeichneten Inhalte selbst erstellt habe.

Mir ist bekannt, dass unmarkierte und unbelegte Zitate und Paraphrasen Plagiate sind und nicht als handwerkliche Fehler, sondern als eine Form vorsätzlicher Täuschung der Prüfer gelten, da fremde Gedanken als eigene Gedanken vorgetäuscht werden mit dem Ziel der Erschleichung einer besseren Leistungsbewertung.

Mir ist bekannt, dass Plagiarismus

- die Standards guter wissenschaftlicher Praxis,
- den Leitfaden für gute wissenschaftliche Praxis im Studiengang MKI,
- die Studien- und Prüfungsordnung der Hochschule Reutlingen (§10 Täuschung und Ordnungsverstoß) sowie
- das Landeshochschulgesetz von Baden-Württemberg (§3 Wissenschaftliche Redlichkeit Abs. 5, §62 Exmatrifikation Abs. 3)

missachtet und seine

studienrechtlichen Folgen vom Nichtbestehen bis zur Exmatrifikation reichen.

Nachname: Schramm

Vorname: Robin

Matrikelnummer: 761392

abgegeben zur Lehrveranstaltung: Bildverarbeitung

für das Semester: Sose 2020 - HUC II

Datum, Ort: 17.07.2020, Balingen

Unterschrift: R. Sch.

¹ <https://bscwserv.reutlingen-university.de/bscw/bscw.cgi/d2871027/GWP.pdf>