AG Medieninformatik

Bachelorarbeit

Ein mobiler Augmented-Reality-Ansatz zur Erkennung kontextabhängiger Teilbilder mittels Feature-Matching

Andre Sanders

16. November 2017

Erstgutachter: Prof. Dr. Oliver Vornberger
Zweitgutachterin: Prof. Dr. Elke Pulvermüller

Abstract

Augmented-Reality-basierte Ansätze finden mittlerweile in vielen Bereichen des Alltags ihre Anwendung, wie zum Beispiel dem Erkennen von Objekten, um einem Anwender interessante Informationen über diese zu liefern. Dabei kann ein Bild durch eine Kamera aufgenommen und mit einem internen Datenbestand an Bildern abgeglichen werden. Die Bilder lassen sich durch sogenannte Features beschreiben und miteinander vergleichen (Feature-Matching). Interessant jedoch ist nicht nur die Erkennung der Gesamtheit dieser Bilder, sondern ebenfalls die derer Teilaspekte.
Die Erkennung der Studenten auf den Algorithmenfotos der Universität Osnabrück mit Hilfe der Smartphone-Kamera eines Besuchers bildet die Grundlage für die Frage dieser Arbeit, ob sich eine mobile Lösung entwickeln lässt, Teilbilder aus einem Gesamtkontext zu erkennen und dem Benutzer hierzu hinterlegte Informationen zu präsentieren.
Kann eine solche Anwendung generisch entwickelt werden, könnte mit dem selben Schema, beispielsweise an die Erkennung von Teilobjekten eines Gemäldes in einem Museum, herangegangen werden.
Zum Zeitpunkt dieser Arbeit hegt das Felix-Nussbaum-Museum der Stadt Osnabrück großes Interesse die herkömmliche Museumsentdeckung durch Audioguides auf eine Augmented-Reality-Ebene zu verlagern.
Das auf Feature-Matching basierende Abfragen von Teilinformationen der Algorithmenfotos, sowie denen eines Gemäldes des Felix-Nussbaum-Hauses, soll Anwendern durch eine mobile Website zugänglich gemacht werden. Des Weiteren wird untersucht, ob eine Verlagerung der Kernfunktionalität auf eine native App dazu führt, dass sogar eine mobile Echtzeitanwendung mittels Feature-Matching-Verfahren entwickelt werden kann. Diese könnte die Basis für eine Erweiterung oder die Ersetzung der bisher in Museen verwendeten Audioguides bedeuten.
Die Umsetzung der Bilderkennung und des Teilbild-Matchings geschieht mittels der Open Source Bibliothek OpenCv, die unter Anderem Module zur Erkennung und Verarbeitung mehrdimensionaler Objekte beinhaltet. Durch die Evaluierung der Anwendungsfälle und eine darauf aufbauende Implementierung der zwei verschiedenen Ansätze wird gezeigt, dass solche Lösungen möglich, jedoch je nach Ansatz zur Zeit noch beschränkt in ihrer Funktion sind.

Inhaltsverzeichnis

1 Einleitung

Zu Beginn dieser Arbeit wird die Motivation, sowie das experimentelle Umfeld erklärt. Zusätzlich wird die genaue Zielsetzung definiert.

1.1 Motivation

Mobile Lösungen erleichtern Menschen in vielen Situationen den Alltag. In Verbindung mit modernen Methoden der Computer Vision (CV) vor allem im Bereich der Objekterkennung ist es dem Anwender möglich mehr über sein sichtbares Umfeld zu erfahren, als das Auge erfassen und das Gehirn verarbeiten kann. Augmented-Reality-Applikationen versuchen die Aufgaben des menschlichen Sehens zu simulieren, indem sie Objekte in der Umwelt erkennen und die aufgenommenen Daten mit Informationen zu bestücken, die über die Wissensbasis des Anwenders hinausgehen[1]. Diese *augmentierten* Objekte können einem Anwender zum Beispiel auf dem Display eines Smartphones präsentiert werden.
Anstatt Informationen über ein Objekt in seiner Gesamtheit zu benötigen, gibt es ebenfalls Situationen in denen Informationen über dessen Teilaspekte benötigt werden. Zwei solcher Szenarien bilden die Grundlage dieser Arbeit:

Die Algorithmenfotos der Universität Osnabrück:
An der Universität Osnabrück werden bereits seit 1992 jährlich Gruppenfotos der Studenten der Vorlesung *Informatik A* gemacht, die eingerahmt einen Flur der Universität zieren. Diese Fotos gaben den Anstoß für die zu erarbeitende Lösung dieser Arbeit.
Einem Anwender sollen Teilinformationen über das Gruppenfoto (Die Namen der Personen) bereit gestellt werden, indem er die smartphone-interne Kamera auf den Bildausschnitt mit der Person, dessen Namen er wissen möchte, hält. Es bedarf also einer mobilen Anwendung, die die Teilbilder eines Gesamtbildes verarbeiten und dem Nutzer die benötigten Informationen auf dem Display präsentieren kann. Die Bilder sind auf dem Server der Universität gespeichert[2]. Im entsprechenden HTML-Quelltext eines jeden Fotos befinden sich die Konturen der enthalten Personen als Koordinatenfolge, sowie die dazugehörigen Textinformationen (die Namen der Personen) Diese Rohdaten lassen sich zur Entwicklung verwenden.

[1] Was ist Augmented Reality? http://www.augmented-minds.com/de/erweiterte-realitaet-anwendung/was-ist-augmented-reality - (Abrufdatum: 9.10.17)
[2] www.informatik.uni-osnabrueck.de/arbeitsgruppen/medieninformatik/algorithmen_photos.html

(a)

(b)

Abbildung 1.1: Beispiel-Instanz bestehend aus Urbild (a) und Teilbild (b) des Informatik A Kurses im WS 2015/2016

In Abb. 1.1 ist eine beispielhafte Instanz zu sehen, bestehend aus einem Bild vor dem der User stehen könnte in digitaler Form (*Urbild*) und einem Ausschnitt (*Teilbild)*, den er mit seiner Smartphonekamera selektiert .

Gemälde in einem beliebigen Museum:
Kann eine solche Anwendung für den o.g. Anwendungsfall umgesetzt werden, entsteht die Annahme, dass diese sich auch auf andere Anwendungsfälle übertragen lässt.
Die speziellere Annahme ist, dass die selbe Anwendung zur Erkennung von Teilbildern in den Gemälden eines Museums, bei einem Museumsbesuch, geeignet sein könnte.
Lässt sich eine Applikation entwickeln, die einem Anwender zuverlässig, beispielsweise geschichtliche, Hintergrundinformationen über die Teilaspekte eines Gemäldes liefert, könnten die herkömmlichen Audioguides durch diese erweitert, oder sogar vollkommen ersetzt werden.

1.2 Umfeld der Arbeit

Die Entwicklung der Applikationen zur Erkennung der Algorithmenfotos erfolgt an den realen Bildinstanzen der Universität selbst.
In Kooperation mit dem Felix-Nussbaum-Haus der Stadt Osnabrück[3] ist es außerdem möglich eine Testinstanz für den Anwendungsfall eines Museumsbesuches aufzustellen. Die prototypische Anwendung soll verwendet werden können, um Teilbilder eines vorab mit dem Museum ausgewählten Gemäldes zu erkennen, und einem hypothetischem Besucher die Informationen über diese zu präsentieren.
Ist das entwickelte Verfahren stabil und liefert akzeptable Ergebnisse, könnte dieses eventuell die Grundlage für ein *Benutzerleitsystem* des Museums bieten. Laut eigenen Aussagen der Museumsleitung soll die Museumsentdeckung in Zukunft mehr auf die digitale Ebene verlagert werden, so dass sich seitens der Institution ebenfalls die Frage der generellen Umsetzbarkeit und Komplexität solcher Verfahren stellt.

[3]Felix-Nussbaum-Haus www.osnabrueck.de/fnh/start-fnh.html (Abrufdatum: 9.10.17)

1.3 Zielsetzung

Das Ziel dieser Arbeit ist es zu überprüfen, ob und inwieweit eine mobile Anwendung entwickelt werden kann, die die Grundfunktion für das in Abschnitt 1.1 vorgestellte Szenario der Algorithmenfotos bereit stellt. Unter der Verwendung des Frameworks *OpenCv*[4] soll die Umsetzbarkeit einer solchen Anwendung mittels Feature-Matching-Prozessen gezeigt werden. Außerdem wird überprüft, inwieweit sich das erstellte Verfahren auf das ebenfalls vorgestellte Szenario eines Museumsbesuches anwenden lässt.
Zum Einen soll dem Anwender in Form einer mobilen Website ermöglicht werden, ein Teilbild mit der Kamera seines Smartphones aufnehmen zu können, welches anschließend zur Verarbeitung an den Server geschickt wird. Dort soll der Inhalt mittels eines Feature-Matching-Prozesses erkannt und dem Anwender die dazu hinterlegten Informationen auf dem Smartphone präsentiert werden.
Eine weitere darauf aufbauende Zielsetzung ist die Entwicklung einer nativen Applikation, die aufgrund der Popularität und bereits vorhandener Vorkenntnisse für die Android-Plattform[5] entwickelt wird.
Ausschlaggebend für diese zusätzliche Entwicklung ist die Frage, inwiefern eine Echtzeit-Anwendung gemäß üblicher Augmented-Reality-Applikationen, basierend auf Feature-Matching-Prozessen möglich ist.
Dies ist bei einer Website schon aufgrund des entstehenden Internettraffics durch permanentes Hochladen der Teilbilder, und der Latenzen durch eine ggf. schlechte Internetverbindung an den erwähnten Orten, nicht denkbar.
Beide Anwendungen werden nur prototypisch entwickelt und nicht vollends optimiert, da dies der Rahmen dieser Arbeit nicht erlaubt. Mit Hilfe des Frameworks soll zunächst untersucht werden, ob überhaupt ein Verfahren entwickelt werden kann, das die Teilbilder in den gegebenen Bildinstanzen zuverlässig erkennt, im Urbild findet und in angemessener Zeit eine wahrheitsgemäße Ausgabe generiert.
Ist ein Verfahren gefunden das vielversprechende Ergebnisse liefert, soll es in die Website integriert und anschließend auf ein Verfahren erweitert werden, das mehrere Bilder pro Sekunde verarbeiten kann. Letzteres soll dann als Kern der nativen Echtzeit-Anwendung fungieren.
Das Verfahren soll möglichst robust, besonders gegenüber unterschiedlicher Abstände des Anwenders zum gewünschten Bild, sowie der Verarbeitung mit unterschiedlichen Bildqualitäten (der Auflösung, dem *Verschwommenheitsgrad* oder auch der *Skalierung*) sein.
Die zwei eigenständigen Ansätze zur Lösung des vorangestellten Problems sollen dem Leser präsentiert und somit die Umsetzbarkeit mobiler Augmented-Reality-Anwendungen zur Erkennung von Teilbildern gezeigt werden.

[4] OpenCv Official opencv.org/ (Abrufdatum: 9.10.17)
[5] Official Android android.com

1.4 Aufbau der Arbeit

Zunächst wird dem Leser in Kapitel 2 ein Einblick in verschiedene Methoden der Bild- und Objekterkennung, sowie das verwendete Framework OpenCv gegeben. Sowohl das Prinzip des Beschreibens als auch das des Vergleichens zweier Bilder wird anhand bereits bestehender Algorithmen verdeutlicht. Dabei wird beispielhaft auf zwei Detektions-Algorithmen eingegangen, die als Vorbereitung auf den Scale Invariant Feature Transform (SIFT)-Algorithmus dienen, der in der späteren Implementierung seine Anwendung findet. Eine Vorabuntersuchung (siehe 2.7) des Vorhabens dieser Arbeit hat gezeigt, dass der SIFT-Algorithmus hinreichend gute Resultate für das bereits beschriebene Problem liefert.
Das detaillierte Extrahieren von Features, sowie das eigentliche Matching werden aufgrund des beschränkten Rahmens dieser Arbeit nur anhand eben dieses Algorithmus veranschaulicht, obgleich es noch viele andere Detektions- und somit auch Matchingverfahren gibt.
In Kapitel 3 wird das Konzept, sowie die Architekturen, der zu entwickelnden Anwendungen vorgestellt und ein integrierbares Verfahren entwickelt, welches Teilbilder eines 2D-Kontextes zuverlässig erkennt und dem Anwender die dazu hinterlegten Informationen präsentiert. Es wird gezeigt, dass ein solcher Anwendungszyklus in eine webbasierte Lösung integriert werden und als Grundlage eines mobilen Echtzeit-Verfahrens dienen kann. Die Implementierung der plattformspezifischen Entwürfe wird in Kapitel 4 vorgestellt. Zum Ende der Arbeit erfolgt in Kapitel 5 eine Zusammenfassung der Ergebnisse, sowie ein kurzer Ausblick.

2 Grundlagen der 2D-Bilderkennung

Dieses Kapitel behandelt die notwendigen Grundlagen für die Entwicklung der in Kapitel 1 erwähnten Anwendungen, die in den nachfolgenden Kapiteln vorgestellt werden.
Dem Leser wird zunächst das Konzept von Features anhand bekannter Detektionsverfahren nahe gebracht, um anschließend auf den in der Implementierung verwendeten Algorithmus einzugehen.
Es wird vorausgesetzt, dass der Leser bereits grundlegendes Verständnis im Bereich der Bildverarbeitung, sowie für diverse mathematische Konzepte, wie z.B. für lineare und affine Abbildungen besitzt.

2.1 Features

Lokale Merkmale, Features, oder auch *Keypoints* genannt, sind die charakteristischen Orte eines Bildes. Die Merkmale sind möglichst einzigartig in ihrer Umgebung und eignen sich besonders in Situationen, in denen mehrere Bilder / Objekte miteinander verglichen werden sollen, die sich im Betrachter-Blickwinkel, der Form, der Helligkeit, der Auflösung oder anderen Faktoren unterscheiden.
Das Vergleichen zweier Bilder mit Hilfe von Features kann in drei Teilschritte gegliedert werden [Szel 11, S. 183-184]:
Zunächst werden mittels eines Algorithmus in beiden Bildern die markanten Merkmale ausfindig gemacht (Feature-Detektion) und in einem zweiten Schritt in einer geeigneten Form dargestellt, indem für jedes Feature ein zugehöriger Deskriptor-Vektor generiert wird (Feature-Deskription) der je nach Algorithmus die Umgebung um das zugehörige Feature in einer kompakten Form beschreibt. In einem letzten Schritt, dem *Feature-Matching* können Sets von Deskriptoren verschiedener Bilder mit entsprechenden Distanzfunktionen miteinander verglichen werden.
Features und die zur Ermittlung benötigten Algorithmen müssen bestimmten Anforderungen genügen, um das gewünschte Ergebnis zu erzielen. Generell können unter anderem Ecken, Kanten oder allgemeine interessante Orte, sogenannte Point of interest (POI), Region of interest (ROI), oder *Blobs* dienen[Ahad 11, S.11]. Diese haben je nach Anwendungsfall einen unterschiedlich hohen Nutzen, wie durch die folgende Abbildung veranschaulicht wird. Die Patches A und B in Abb. 2.1 stellen keine nützlichen Features dar, da sie nicht einzigartig sind. Es fällt schwer zu entscheiden, welche Stelle des Bildes durch diese Patches beschrieben wird. C und D, sind als Kanten schon besser zu lokalisieren, für einige Fälle ausreichend, jedoch anfällig gegenüber geometrischen Transformationen. E und F, stellen als Ecken die am eindeutigsten lokalisierbaren Orte des Bildes dar [Undea]. Wie aus Abb. 2.1 hervorgeht, ist ein Feature bestenfalls resistent (invariant) gegenüber Änderungen des Betrachter-Blickwinkels, geometrischen und perspektivischen Transformationen, sowie veränderten Lichtverhältnissen und Rauschen [Isik 14, S. 1]. Eine zweite wichtige Eigenschaft eines Merkmals ist die Unterscheidbarkeit,

Abbildung 2.1: Unterscheidbarkeit von Features
(Quelle: [Undea])

die jedoch in gewissem Maße mit der Robustheit in Widerspruch steht [Effi, S. 12].
Soll eine Anwendung z.B. Rechtecke erkennen und sind die Algorithmen invariant gegenüber Rotation, sinkt die Unterscheidbarkeit, da ein horizontales Rechteck nicht mehr von einem vertikalen Rechteck zu unterscheiden ist. Die Anforderungen an die Algorithmen zur Ermittlung und Beschreibung von Features lassen sich somit wie folgt zusammenfassen [Hass 16, S. 14-15]:

Robustheit: Die Robustheit der Detektion und der Deskription wird durch möglichst vielfältige Invarianz erreicht.
Wiederholbarkeit: In zwei Szenen mit gleichem Inhalt sollte ein Algorithmus die selben Feature-Punkte finden.
Generalität: Die detektierbaren Features sollten in mehreren Anwendungen verwendet werden können.
Effizienz: Um (mobile) Echtzeit-Anwendungen zu ermöglichen sollten Features schnell und Ressourcensparend berechnet werden.
Quantität: Ein Algorithmus sollte möglichst alle existierenden Features, auf die er ausgelegt ist, im untersuchten Bild ausfindig machen.

Im Folgenden soll besonders das Problem der Invarianz eines Detektionsverfahrens anhand verschiedener Algorithmen verdeutlicht werden.

2.2 Nicht-skaleninvariante Merkmalsdetektion

Seit den 1980er Jahren werden sukkzessive erfolgreichere Methoden entwickelt möglichst unterscheidbare und robuste Features zu detektieren. Dabei werden häufig Gradienten und höhere Ableitungen der Bildfunktion verwendet um Ecken, Kanten und andere `POI` zu erkennen [Burg 09, S. 155-156]. Eine Kante ist dabei häufig eine Region, in der der Gradient der Bildfunktion $I(x, y)$ in einer Richtung signifikant hoch ist, wobei die Funktion den Pixelwert an Position (x, y) zurückliefert. Ein Eckpunkt wiederum weist einen in mehrere Richtungen hohen Gradientenwert auf [Burg 09, S. 155-156].
Dieser Abschnitt behandelt aufgrund des begrenzten Rahmens dieser Arbeit nur zwei

der vielen Entwicklungen im Bereich der Merkmalsdetektion, um die allgemeine Funktionsweise von Detektionsverfahren zu veranschaulichen und anschließend den für diese Arbeit wichtigen SIFT-Algorithmus zu erläutern.

2.2.1 Moravec-Detektion

Einer der ersten Feature-Detektoren geht auf H.P. Moravec von 1980 zurück [Mora 80]. Dieser beschreibt einen markanten Punkt (i.e. eine Ecke) als eine Region mit geringer *Selbstähnlichkeit*. Zur Ermittlung dieser Ecken (und somit auch Kanten) wird in einem Graustufenbild pro Region ein kleines Fenster in verschiedene Richtungen (senkrechter, waagerechter, diagonaler shift) geschoben und mittels der Summe der kleinsten Quadrate nach Intensitätsveränderungen in den Graustufen gesucht.
Die Intensitätsänderung E um einen Pixel (x, y) in Richtung (u, v) (Shift) ist mit der Pixelintensität I gegeben durch

$$E_{u,v} = \sum_{x,y} w(x, y) \left| I(x + u, y + v) - I(x, y) \right|^2 \tag{2.1}$$

mit einer binären Funktion w

$$w(u, v) = \begin{cases} 1 & \text{wenn } (u, v) \text{ in lokalem Fenster,} \\ 0 & \textit{sonst.} \end{cases}$$

Der Moravec-Detektor findet eine Ecke, indem nach lokalen Maxima in min $E\{xy\}$ gesucht wird, die über einem Schwellwert liegen. [Zhen 99, S. 153].

2.2.2 Harris-Corner-Detektion

Eine der vielen Erweiterungen des Moravec-Detektors wurde 1988 von Harris und Spephens vorgestellt [Harr 88]. Den Wissenschaftlern zufolge weist der Moravec-Operator einige Schwächen auf. Er erzeugt unter Anderem *verrauschte* Antworten und ist nicht isotrop, da nur eine diskrete Anzahl von Shifts betrachtet wird. Aus diesem Grund ist der Detektor zwar invariant gegenüber Translationen, jedoch nicht gegenüber beispielsweise Rotationen des Eingangsbildes [Effi, S. 13].
Mit dem Ziel unter Anderem das Rotationsproblem zu lösen entstand der in der Literatur häufig nur unter *Harris-Corner-Detektor* bekannte Algorithmus.

Ein Beispiel von extrahierten Merkmalen eines Eingangsbildes mittels dieses Verfahrens ist Abb. 2.2 zu entnehmen. Als Antwort auf das Problem der Anisotropie verwenden die Autoren keine diskrete Berechnung der Shifts um jeden Pixel, sondern approximieren eine analytische Lösung [Harr 88]. Eine Taylorreihenentwicklung führt zu einer symmetrischen 2x2-*Auto-korrelationsmatrix* M, die auf den Termen der Taylorreihe basiert.
Mit ihr ist es möglich Intensitätsveränderungen E in jede beliebige Richtung zu berechnen. Für einen kleinen shift (u, v) gilt, wie in [Harr 88] und [Harr] erläutert:

$$E(u, v) \approx \begin{bmatrix} u & v \end{bmatrix} M \begin{bmatrix} u \\ v \end{bmatrix} \tag{2.2}$$

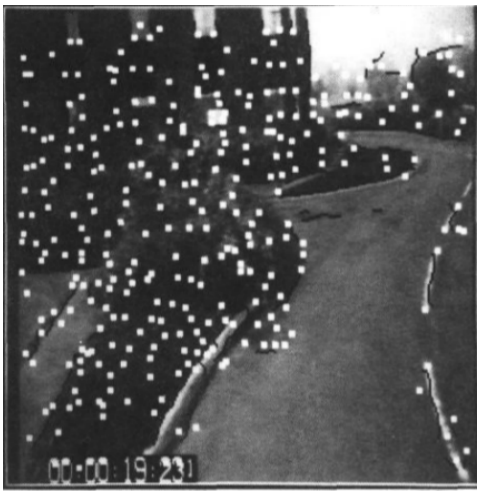


Abbildung 2.2: Harris-Corner-Detektor am Beispiel einer Live-Szene (Erkannte Ecken in Weiß, sowie erkannte Kanten in Schwarz dargestellt)
(Quelle: [Harr 88, S.151])

mit

$$M = \sum_{x,y} w(x,y) \begin{bmatrix} I_x^2 & I_x I_y \\ I_x I_y & I_y^2 \end{bmatrix} = \sum_{x,y} w(x,y) \begin{bmatrix} A & C \\ C & B \end{bmatrix} \tag{2.3}$$

und

$$w(x,y) = \exp(-\frac{x^2+y^2}{2\sigma^2}) \tag{2.4}$$

wobei $w(x,y)$ eine Gewichtungsfunktion in Form eines zweidimensionalen Gaussfilters darstellt. Dieser glättet die *verrauschten* Antworten, indem die Intensitäten nicht in einer rechteckigen, sondern in einer kreisartigen Nachbarschaft untersucht werden. Im Radius σ wird jeder Pixel mit Nähe zu seinem Zentrum gewichtet [Burg 09, S. 104]. Über die Berechnung der beiden Eigenwerte α und β der Matrix M sowie deren Eigenvektoren lassen sich nun Regionen unterscheiden, die entweder keine signifikanten Veränderungen aufweisen, eine Ecke oder eine Kante beinhalten. Harris und Stephens beschreiben eine sogenannte *Corner-Response-Function*, die auf jedes Pixel angewendet wird und anstatt der aufwendigen Berechnung der Eigenwerte der Matrix, deren Determinante und die Spur verwendet [Harr 88]:

$$R = Det(M) - k * Spur(M)^2 \tag{2.5}$$

mit

$$Spur(M) = A + B = \alpha + \beta \tag{2.6}$$

und

$$Det(M) = AB - C^2 = \alpha\beta \tag{2.7}$$

Dabei beschreibt der Parameter k die Empfindlichkeit des Detektors. Ist R groß, handelt es sich mit einer hohen Wahrscheinlichkeit um eine Ecke. Der Leser sei für weiter reichende Theorie auf die angegebenen Arbeiten verwiesen [Mora 80][Harr 88].

Da die bisherigen Detektoren vor allem Ecken als Features ausfindig machen, sind diese wiederholbar trotz diverser Bildtransformationen, jedoch nicht bei großen Skalenveränderungen, wie auf Abb. 2.3 zu sehen.

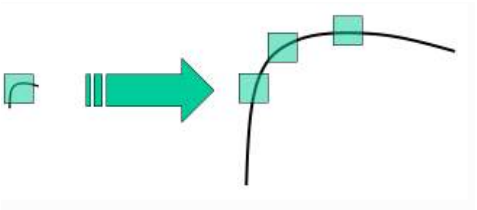


Abbildung 2.3: Skalenveränderungen führen zur nicht-Erkennung von Ecken (Quelle: [Intra])

Dies ist darauf zurückzuführen, dass die Algorithmen einen Gaussfilter einer bestimmten fixen Größe(Skalierung) σ verwenden und außer der Position und Intensität des Features keine weiteren Informationen liefern (siehe Gleichung 2.2 und [Burg 09, S. 643]). Der folgende Abschnitt behandelt deshalb ein Verfahren, das dieses Problem löst und in der späteren Implementierung der Anwendung verwendet wird, um zu ermöglichen, dass ein Anwender ein Teilbild einer Bildinstanz z.B. aus verschiedenen Entfernungen aufnehmen kann.

2.3 Scale Invariant Feature Transform

Dieser Abschnitt erläutert einen Ansatz zum Erreichen von Skaleninvarianz und bezieht sich hauptsächlich, sofern nicht anders angegeben, auf die Arbeiten von David Lowe der Jahre 1999 und 2004 [Lowe 99][Lowe 04], sowie den Ausführungen Dr. Burgers *et.al.* [Burg 09, S. 643-708]. Der SIFT-Algorithmus wurde 2004 in den USA von Lowe patentiert [Goog] und ist auch in darauf aufbauenden Implementierungen, wie beispielsweise im Framework *OpenCv* nicht ohne Weiteres kommerziell nutzbar. Die folgenden Absätze geben die Erkenntnisse Lowes aus seiner Arbeit wieder, da mit diesem Algorithmus in der in Kapitel 4 vorgestellten Implementierung das Feature-Matching geschieht.
Der Algorithmus nach Lowe kann in folgende Phasen gegliedert werden [Lowe 04]:

1. `Merkmalssuche im DoG-Skalenraum`:
   Zuerst wird ein *Skalenraum* aufgebaut, in dem mittels des Difference of Gaussian (DoG) über alle Pixel und Skalierungen nach potenziellen Merkmalskandidaten gesucht werden kann.

2. `Genaue Lokalisierung der Merkmalskandidaten`:
   Für jeden Merkmalskandidaten wird dessen genaue Position und Intensitätsstärke bestimmt und alle Kandidaten nach bestimmten Kriterien gefiltert.

3. `Ermittlung der Orientierungen`:
   Jedem Merkmal wird ein Richtungsvektor zugewiesen, so dass alle weiteren Operationen auf diesen invarianten Merkmalen ausgeführt werden können.

4. `Berechnung der Deskriptoren`:
   Für jedes Merkmal werden die Bildgradienten in der entsprechenden Region berechnet und in eine effiziente Darstellung überführt, die unter anderem invariant gegenüber Skalierung, Rotation, Rauschen und Beleuchtung ist und in weiteren Operationen, wie dem Matching verwendet werden kann.

Die Funktionsweise des Algorithmus wird im Folgenden genauer erläutert.

2.3.1 Skalenräume

Um skalierungsinvariante Features zu detektieren hat es sich als sinnvoll herausgestellt Bilder nicht zweidimensional, sondern dreidimensional zu behandeln, wobei die dritte Dimension σ die Skalierung darstellt. So ist es möglich einen sogenannten *Skalenraum* $L(x, y, \sigma)$ aufzustellen, der das selbe Bild in unterschiedlichen Auflösungen beinhaltet. Die *scale space theory* wurde 1994 von Tony Lindeberg erörtert [Lind 94]. Laut Lindeberg, ist der einzige Kandidat zur Aufstellung eines Skalenraumes eine Gaussfunktion variabler Skalierung, die für eine sukkzessive Faltung sorgt und Bilder mit einer konstanten Skalierungsdifferenz k erzeugen kann. Die Gaussfunktion bringt zwei weitere Vorteile mit sich [GREM 08]: Zum Einen ist sie *separierbar*. Statt ein Bild einmal mit einer zweidimensionalen Funktion zu falten, lässt sich das selbe Ergebnis durch die effizientere zweimalige Faltung mit einer eindimensionalen Gaussfunktion erreichen. Zum Anderen ist die Funktion rekursiv berechenbar, so dass die Folgeebenen jeweils aus der vorherigen Ebene berechnet werden können.
Ein Gauss-Skalenraum, bestehend aus geglätteten Bildern $L(x, y, \sigma)$ eines Eingangsbildes I ist somit gegeben durch

$$L(x, y, \sigma) = G(x, y, \sigma) * I(x, y) \tag{2.8}$$

mit $*$ als Faltungsoperator und

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2+y^2)}{2\sigma^2}} \tag{2.9}$$

wobei mit dem variablen Skalierungsparameter σ die entsprechende Skalierungsebene bestimmt werden kann (siehe auch [Lowe 04]). Da die Bilder nach einer gewissen Anzahl von Ebenen Q und somit der mehrfachen Anwendung von Glättungen immer verschwommener werden, wird ein Unterraum mit einer verkleinerten Version des Bildes erzeugt, eine weitere sogenannte *Oktave*. Das Anfangsbild der Folgeoktave ist eine um 50 Prozent herunterskalierte Version der vorherigen Oktave, bei der nur jedes zweite Pixel in x- und in y-Richtung verwendet wird [Burg 09, S.650-664]. Dieses einfache Subsampling kann ohne Informationsverlust erreicht werden, da durch den vorher angewendeten Gauss-Filter, alle direkten Nachbarn eines Pixels bei der Glättung mit einbezogen werden. Das Startbild wird wieder sukkzessive gefaltet und die Schritte wiederholt, bis keine sinnvolle Verkleinerung des Bildes mehr durchgeführt werden kann. Die Abb. 2.4 zeigt beispielhaft die Faltung eines Bildes durch einen Gauss-Filter mit Varianzen des Skalierungsparameters σ.

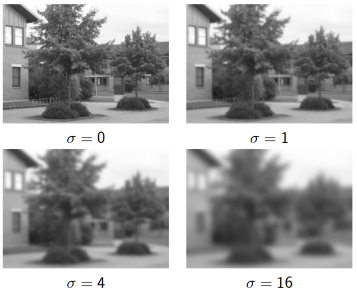


Abbildung 2.4: Unterschiedliche Varianzen des Gauss-Kerns
(Quelle: http://dustsigns.de/CMS/wp-content/uploads/Merkmalerkennung.pdf)

Neben der Gaussfunktion wird in der Literatur eine weitere wichtige Funktion beschrieben. Der Laplace-Operator einer Funktion ∇^2 ist gegeben durch deren zweiten partiellen Ableitungen

$$\nabla^2 L(x, y, \sigma) = L_{xx}(x, y, \sigma) + L_{yy}(x, y, \sigma) \tag{2.10}$$

Der auf diesem basierende Laplacian of Gaussian (LoG) ist ein zur Merkmalsdetektion auf einen Gauss-Skalenraum angewandter Detektor und reagiert vor allem auf Regionen, die signifikant dunkler bzw. heller als ihre Nachbarschaft sind [Burg 09, S.647]. Der LoG kann als Kombination mit anderen Detektoren verwendet werden, um diese invariant gegenüber Skalierung zu gestalten, wie der Harris-Laplace-Ansatz, Hessian-Laplace-Ansatz und andere zeigen (z.B. von Mikolajczyk beschrieben [Miko 04]). Wird der normierte Laplace-Operator auf eine zweidimensionale Gaussfunktion angewendet, erhält man die LoG-Funktion, mit der es außerdem möglich ist für jedes Merkmal eine charakteristische Skalierung zu schätzen [Grau 11, S. 29].
Wie in Abb. 2.5 zu sehen ist, lässt sich durch die Anwendung des LoG ebenfalls ein Skalenraum aufbauen, in dem es möglich ist nach skaleninvarianten Extrema zu suchen.

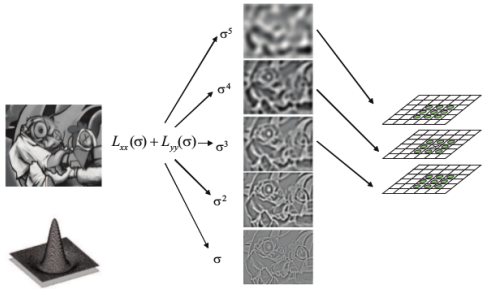


Abbildung 2.5: Beispiel der Detektion von Skalenraum-Extrema im LoG space
(Quelle: [Grau 11, S. 30])

Damit die Ergebnisse verschiedener Skalenebenen miteinander vergleichbar sind und der Operator volle Skaleninvarianz erreicht muss dieser normiert werden [Lind 94]

$$\nabla^2_{norm} L(x, y, \sigma) = \sigma^2 (L_{xx}(x, y, \sigma) + L_{yy}(x, y, \sigma)) \tag{2.11}$$

. Laut Mikolajczyk liefern die Maxima und Minima dieser Funktion die stabilsten Features im Vergleich zu anderen Detektionsverfahren, wie Harris-Corner, Hesse-Matrix oder Gradientenfunktionen, wie er 2002 beschreibt [Miko 02]. Der sehr rechenaufwendige LoG kann, wie durch Lowe im SIFT-Algorithmus verwendet, durch den DoG bis auf eine Konstante angenähert werden, indem die Differenz zweier Graubilder aus dem Gauss-Skalenraum gebildet wird. Diese bilden eine neue Ebene, so dass ein neuer, sogenannter DoG-Skalenraum entsteht[Lowe 04]:

$$D(x, y, \sigma) = (G(x, y, k\sigma) - G(x, y, \sigma)) * I(x, y) = L(x, y, k\sigma) - L(x, y, \sigma) \tag{2.12}$$

2.3.2 Merkmalssuche im DoG-Skalenraum

In den inneren Ebenen des aufgestellten DoG-Raum können nun Minima und Maxima gefunden werden, indem die Hesse-Matrix der Umgebung betrachtet wird.
Laut Lowe kann diese effizient angenähert werden, indem die 26 Nachbarpixel des untersuchten Pixels einer 3x3-Umgebung, also die acht unmittelbaren, sowie die neun umgebenden Nachbarn der Skalierungsebene jeweils ober- und unterhalb der Bildebene untersucht werden (siehe Abb. 2.6). Der Rechenaufwand ist relativ gering, da viele Punkte bereits nach dem Vergleich mit einigen wenigen Nachbarn abgelehnt werden [Scal].

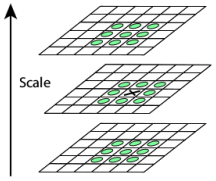


Abbildung 2.6: Minima- und Maxima-Suche im DoG-Raum durch Nachbarschaftsanalyse (Quelle: [Lowe 04])

Durch Experimente von Lowe wurden nahezu optimale Werte der Parameter ermittelt, die zu guten Ergebnissen in der Praxis führen: Die Anzahl der Oktaven $P = 5$, Anzahl der Ebenen $Q = 5$, eine initiale Skalierung des Raumes von $\sigma = 1.6$ und für die konstante Skalierungsdifferenz k zweier Ebenen $k = \sqrt{2}$ [Intra].

2.3.3 Lokalisierung der Merkmale

Nach dem vorherigen Schritt ist eine Menge potenzieller Merkmalskandidaten mit ihren Skalierungen gefunden. Diese sind jedoch mit ihren diskreten 3D-Koordinaten x, y, σ des herunterskalierten DoG-Raumes gegeben und können auch zwischen zwei Pixeln liegen [Brow 02]. Mit einer kontinuierlichen Funktion muss deshalb ein genauerer lokaler Positionswert geschätzt werden [Burg 09, S. 670][Scal]. Lowe verwendet hier die Taylorreihenentwicklung, bis auf den Grad zwei, der Skalenraum-Funktion $D(x, y, \sigma)$, um den entsprechenden Merkmalskandidaten $\vec{x} := (x, y, \sigma)$[Lowe 04]:

$$D(\vec{x}) = D + \frac{\delta D^T}{\delta \vec{x}}\vec{x} + \frac{1}{2}\vec{x}^T\frac{\delta^2 D}{\delta \vec{x}^2}\vec{x} \tag{2.13}$$

Anschließend kann das Offset $\hat{x}$ bestimmt und das daraus resultierende 3x3-System effizient gelöst werden.

$$\hat{x} = -\frac{\delta^2 D^-1}{\delta \vec{x}^2} \frac{\delta D}{\delta \vec{x}} \tag{2.14}$$

Ist das Offset größer als 0.5 liegt das untersuchte Extremum näher an einem Nachbarpixel, als am gedachten Pixel. Das finale Offset $\hat{x}$ wird an die ursprüngliche Position addiert und man erhält so die interpolierte Echtposition des Pixels.
Im Anschluss an die Schätzung müssen instabile Kandidaten aus der gefundenen Menge entfernt werden, indem die Stelle $\hat{x}$ auf folgende Kriterien geprüft wird, wobei die Gleichungen wieder der Ausgangsarbeit Lowes entnommen sind [Lowe 04]:

Löschung zu schwacher Kandidaten:
Indem das Ergebnis aus 2.14 in 2.13 eingesetzt wird, lässt sich der Intensitätswert des Merkmals ermitteln. Zuerst werden diejenigen Merkmale entfernt, deren Kontrast unter einem bestimmten Schwellwert liegt. Lowe verwendet in seiner Arbeit einen Schwellwert von drei Prozent und lehnt alle darunter liegenden Features ab [Lowe 04].
Löschung der auf Kanten liegenden Merkmale:
Die DoG-Methode reagiert ebenfalls stark auf selbst schwache, kantenartige Strukturen in Bildern. Diese sind jedoch nicht hinreichend stabil und eignen sich nicht zur zuverlässigen Lokalisation von Bildinhalten [Burg 09][Lowe 04]. Wie Lowe ausführt, weist die DoG-Funktion entlang einer Kante starke und andernfalls schwache Hauptkrümmungen (*principle curvatures)* auf. Diese sind proportional zu den Eigenwerten der $2x2$ Hesse-Matrix H der DoG-Funktion, mit

$$H = \begin{bmatrix} D_{xx} & D_{xy} \\ D_{x,y} & D_{yy} \end{bmatrix} \tag{2.15}$$

und können, wie bereits bei Harris und Stephens (siehe 2.2.2) in effizienterer Weise durch deren Determinante sowie der Spur berechnet werden. Sei r das Verhältnis zwischen des größten Eigenwerts α und des kleineren Eigenwerts β, so dass $\alpha = r\beta$, lässt sich eine Kante durch den Test

$$\frac{Spur(H)^2}{Det(H)} < \frac{(r+1)^2}{r} \tag{2.16}$$

erkennen. Der Test ist effizient, da er sich mit weniger als 20 Fließkommaoperationen ausführen lässt. Lowe verwendet in seinen Testreihen des Weiteren den Wert $r = 10$. In diesem Fall werden alle Merkmale mit $r > 10$ zur weiteren Verarbeitung abgelehnt.

In Abbildung 2.7 ist die Merkmalsextraktion mit SIFT eines Graustufenbildes verdeutlicht. Zu sehen ist das Urbild (a), gefolgt von den nach der Extrema-Suche aus dem DoG-Raum extrahierten 832 Features (b). In (c) ist das Resultat der Löschung schwacher Kandidaten bei einem Schwellwert zu sehen, nach der noch 729 Features vorhanden sind. Die letztendlichen 536 Merkmale nach Filterung der auf Kanten liegenden Kandidaten sind in (d) zu sehen.

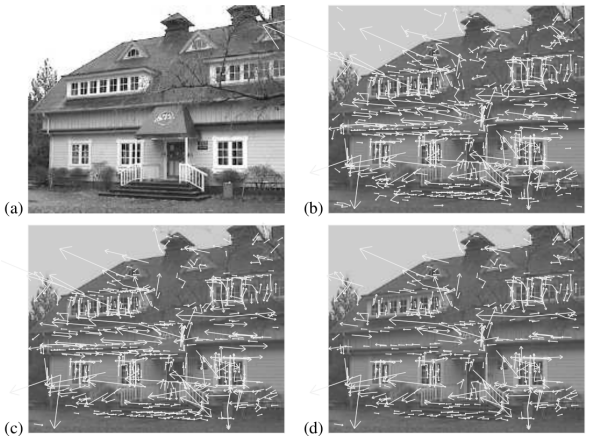


Abbildung 2.7: Beispiel der Merkmalsextraktion eines 233x189-Pixel-Grauwert-Bildes (Quelle: [Lowe 04, S. 12])

2.3.4 Ermittlung der Orientierungen

Das bis zu diesem Schritt gefilterte Set von Merkmalen beinhaltet skaleninvariante Merkmale. Damit diese auch Rotationsinvarianz erlangen, wird für jedes Merkmal eine Orientierung der lokalen Bildstruktur berechnet [Lowe 04, S. 13-14]. Hierzu wird für jedes Merkmal durch die bekannte Skalierung, die Gauss-Ebene L gewählt, die dieser am nächsten ist, damit alle Folgeoperationen in einer skaleninvarianten Umgebung stattfinden.
Für jede Merkmalsumgebung werden nun die Längen $m(x, y)$ und Orientierungen $\theta(x, y)$ der Gradienten pro Pixel mittels einfacher Euklidabstände der Nachbarpixel berechnet

$$m(x,y) = \sqrt{(L(x+1,y) - L(x-1,y))^2 + (L(x,y+1) - L(x,y-1))^2} \tag{2.17}$$

und

$$\theta(x,y) = tan^{-1} \frac{((L(x,y+1) - L(x,y-1))}{(L(x+1,y) - L(x-1,y))} \tag{2.18}$$

, wie dem Originalpaper entnommen werden kann [Lowe 04]. Für diese wird anschließend ein Histogramm erstellt, bestehend aus 36 Bins, die jeweils 10° Grad der 360°-Orientierung abdecken. Die Länge jedes Gradienten wird nun jeweils in den entsprechenden Bin der Gradientenorientierung addiert. Dadurch entstehen Spitzen im Histogramm, wobei die größte Spitze die Hauptorientierung des Merkmals darstellt. Sollten jedoch weitere Spitzen auftauchen, die mehr als 80 Prozent in Relation zum Hauptorientierungsbin gefüllt sind, wird für diese ein neues Merkmal erzeugt. Laut Lowe werden hierdurch mehr Features und ebenfalls mehr Stabilität des Verfahrens erreicht.

2.3.5 Berechnung der Deskriptoren

Das Verfahren des letzten Abschnitts liefert ein Merkmal $k = (x, y, p, q)$ zu jeder dominanten Orientierung θ mit der Position x,y in der Ebene q der Oktave p des Gauss-Skalenraums [Burg 09, S. 680]. Aus dem Gradientenumfeld dieser Merkmale können nun möglichst unterscheidbare SIFT-Deskriptoren erzeugt werden, die auf diesen Daten

basieren.
Dieser Deskriptor gewährleistet unter anderem ebenfalls Skalierungs- und Rotationsinvarianz, da das Beschreibungsverfahren auch in Kombination mit anderen Detektionsverfahren nutzbar sein soll [Lowe 04].
Um das gegebene Merkmal der entsprechenden Skalenebene werden in den Subquadranten der umgebenden quadratischen Region die Gradienten berechnet [Hert 12, S. 85]. Diese werden quadrantenweise in einem Gradientenhistogramm zusammengefasst. Hier wird ähnlich wie in 2.3.4 vorgegangen, jedoch mit einer anderen Aufteilung der Histogramm-Bins.

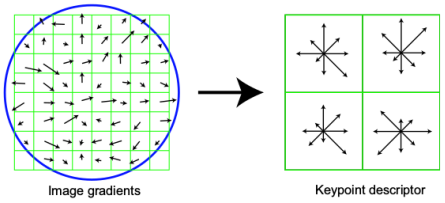


Abbildung 2.8: Erzeugung eines 2x2x8-SIFT-Deskriptors
(Quelle: [Lowe 04, S. 15])

Typischerweise wird eine 4x4-Region um das Merkmal betrachtet, in der für jedes Pixel ein Histogramm mit acht Orientierungsbins berechnet wird. Damit ergibt sich also ein 128-elementiger Merkmalsvektor zur Beschreibung [Hert 12, S. 85]. Abb. 2.8 zeigt den Extraktionsprozess der Histogramme aus den Bildgradienten.
Aus dem Merkmalspunkt k ergibt sich nach diesem Schritt also ein SIFT-Deskriptor $s = (x', y', \sigma, \theta, f_{sift})$, wobei x' und y' die interpolierten Pixelkoordinaten im Ursprungskoordinatensystem darstellen. f_{sift} beschreibt den 128-elementigen Merkmalsvektor [Burg 09, S. 688]. Der Leser sei für tiefergehendes Verständis auf die Arbeiten [Burg 09] und [Hert 12], sowie das Originalpaper von Lowe [Lowe 04] verwiesen.

2.4 Speeded Up Robust Features

Der Speeded Up Robust features (SURF)-Algorithmus geht auf das Jahr 2006 zurück, mit dem Versuch den von Lowe entwickelten Algorithmus performanter zu gestalten [Bay 06]. Lowe approximiert die LoG-Funktion mittels des DoG. Die SURF-Entwickler gehen weiter und benutzen sogenannte *Box-Filter* zur Approximation. Die Faltung mittels Box-Filtern kann einfach durch Integralbilder berechnet werden [Bay 06]. Dieses Vorgehen ist ebenfalls rotations-und skaleninvariant sowie invariant gegenüber einigen anderen Deformationen und soll laut Literatur um bis zu drei mal schneller sein als SIFT [Intrb]. In einer Vorabuntersuchung dieser Arbeit wurde ein simples Matchingverfahren sowohl mit SIFT als auch mit SURF implementiert, dessen Resultate dazu führen, dass das SIFT-Verfahren, dem SURF-Verfahren in dieser Arbeit vorgezogen wird (siehe 2.7).

2.5 Feature-Matching

Sind für zwei Bilder die jeweiligen Merkmalskandidaten bestimmt und in eine Deskriptordarstellung überführt worden, können diese mittels einer definierten Metrik verglichen indem der (Euklid-)Abstand zweier Merkmalsvektoren berechnet wird, wie an der Formel

$$dist(s_i, s_j) = ||f_{sift_1} - f_{sift_2}|| \tag{2.19}$$

aus [Burg 09, S. 698] zu sehen ist. Dabei ist das einfachste Verfahren, das in OpenCv sogenannte *Bruteforce*-Verfahren, bei dem jede mögliche Kombination von Deskriptorpaaren beider Bilder mit der gewählten Metrik verglichen wird und *gute* Matches anhand eines Distanzschwellwerts akzeptiert oder verworfen werden, wie auch in Abb. 2.9 anhand des SIFT-Beispiels zu sehen.

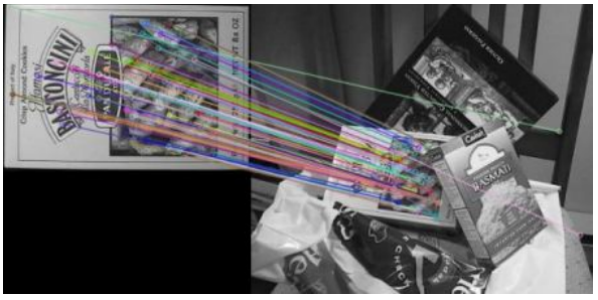


Abbildung 2.9: Feature-Matching mit SIFT-Detektor und SIFT-Deskriptor (Quelle: [offi])

Die Matching-Mechanismen generieren in der Regel zwei Arten von Matches. Positive und negative Matches.
Die einzelnen Matches (in der oberen Abbildung durch Kanten gekennzeichnet), können jedoch für z.B. Genauigkeitstests auch als *True Positive*, *False Negative*, *False Positive* oder *True Negative* klassifiziert werden [Szel 11, S. 200-2002].

- `True Positive (TP)`: Richtige Matches, die auch als richtig erkannt wurden.
- `False Negative (FN)`: Richtige Matches, die als nicht richtig erkannt wurden.
- `False Positive (FP)`: Falsche Matches, die als richtig erkannt wurden.
- `True negative (TN)`: Falsche Matches, die auch als falsch erkannt wurden.

Alle *negative*-Matches sind folglich nicht auf Abb. 2.9 zu finden, da diese nicht existent sind. Die Abbildung zeigt jedoch hauptsächlich *TP*-Matches, und vereinzelt (z.B. auf der rechten Seite) *FP*-Matches.

Verbesserung durch KNN-Verfahren:
Die Matchqualität kann unter Umständen durch die K-Nearest-Neighbour (KNN)-Methode verbessert werden. Anstatt den besten Match durch das Standard-Bruteforce-Verfahren zu ermitteln, lassen sich in der Verbindung mit einer k-Nachbarschaft die

Anzahl der ermittelten Matches erhöhen. Für $k = 2$ befinden sich also nach dem Matching für jedes erkannte Feature die zwei besten Matches in der resultierenden Menge aller Matches M, wobei $\{M\} = \{TP\}^* + \{FP\}^*$ gilt.

Ratio-Test nach Lowe:
Wird die KNN-Methode mit $k = 2$ verwendet, wie in der späteren Implementierung, kann der *Ratio-Test* nach Lowe auf die daraus resultierenden Matches angewendet werden [Lowe 04][Undeb]. Mit dieser Methode können die FP-Matches aus der Menge aller Matches herausgefiltert werden, indem das Distanzverhältnis der zwei ähnlichsten Deskriptoren (KNN-Match) betrachtet wird. Ein guter Match m wird selektiert, falls gilt :

$$m.distance < r * n.distance \tag{2.20}$$

Lowe schlägt hier ein $r = 0.7$ vor, welcher auch in dieser Arbeit verwendet wird.

Transformationsmatrix mit Ransac:
Nach dem Ratio-Test ist das Feature-Matching beendet und eine Auswahl guter Matches getroffen. Für diese Arbeit wird zusätzlich der RANdom SAmple Consensus (RANSAC)-Algorithmus nach Fischler und Bolles [Fisc 81] angewendet. Mit diesem ist es möglich anhand gegebener Punkte zweier verschiedener Koordinatensysteme die Transformationsmatrix H (Homografie) zwischen diesen zu berechnen, so dass

$$\vec{x'} = H\vec{x} \tag{2.21}$$

für möglichst alle Punkte $\vec{x}$ aus dem Teilbild-, sowie allen Punkten $\vec{x'}$ aus dem Urbildraum gilt.

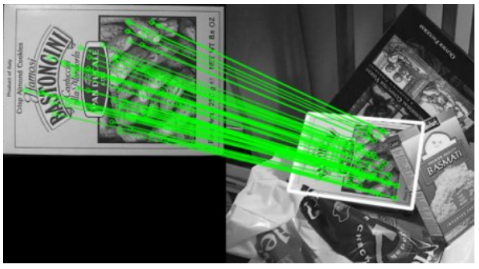


Abbildung 2.10: Beispiel Transformation RANSAC-Matrix
Quelle: [Feata]

Wird diese Matrix auf einen Punkt bezüglich der Basis des ersten Koordinatensystems angewendet, wird dieser in das andere Koordinatensystem transformiert. In Abb. 2.10 wurde die gefundene Transformationsmatrix auf die vier Eckpunkte des linken Bildes angewendet, diese so in die Szene (rechts) transformiert und die Region als weißes Rechteck kenntlich gemacht [Feata].
Der RANSAC-Algorithmus zur Berechnung von H und die gleichzeitige Eliminierung von *Ausreißern* benötigt lediglich genügend Daten (die Matches), um die Parameter des

Modells zu schätzen [Moul 12, S. 260-261]. Dabei wird iterativ ein Modell bestimmt, indem die Anzahl der Merkmale, die in der tatsächlich transformierten Region liegen maximiert und der Fehler zwischen dem geschätzten und dem echten Modell minimiert wird [Moul 12].

2.6 Bildverarbeitung mit OpenCv

Dieser Abschnitt dient dazu dem Leser einen kurzen Einblick in die verwendete Bibliothek zu geben, auf der das spätere mobile Teilbild-Matching-Verfahren basiert.
OpenCv liefert viele Implementierungen der in der CV angewandten Techniken[1]. Aufgrund der BSD-Lizensierung kann diese sowohl für Forschungszwecke als auch kommerziell genutzt werden. Unter anderem bietet die Bibliothek ein Modul, welches Algorithmen im Bereich des Feature-Matchings zur Verfügung stellt.
Das Framework wurde 1999 von der Firma Intel zu Forschungszwecken ins Leben gerufen und beinhaltet aktuell mehr als 2500 Algorithmen, die permanent optimiert werden[2]. Die Hauptbibliothek ist ursprünglich in optimiertem C++ geschrieben. Mittlerweile sind jedoch Schnittstellen zu gängigen Programmiersprachen, wie *C*, *C++*, *Java*, *MATLAB* und *Python* vorhanden. Des Weiteren werden die gängigsten Plattformen, wie Windows, Linux, Android und Mac OS unterstützt. Die in dieser Arbeit verwendeten OpenCv-Module lassen sich auf einige wenige beschränken [Open]:

Core: Das Core-Modul beinhaltet grundlegende mathematische Strukturen, wie Vektoren und Matrizen, sowie Rechenoperationen, die auf diese angewendet werden können. OpenCv bietet unterschiedliche Methoden für unterschiedliche Wertebereiche, wie 8-,16, oder 32-Bit-Integer-Werte und Fließkommazahlen.
Imgproc: Hier befinden sich alle bildverarbeitungsbezogenen Algorithmen, wie Funktionen zum Zeichnen von Bildern, für deren Transformation oder Reduktion.
Imgcodecs: Dieses Modul beinhaltet Funktionen um Ressourcen zu lesen, oder zu schreiben. Diese sind speziell auf die internen Datenstrukturen von OpenCv zugeschnitten, so dass beispielsweise bestimmte Bildformate aus dem Speicher direkt in eine vorgefertigte Matrix-Struktur der entsprechenden Programmiersprache umgewandelt werden können.
FLANN: Das Modul Fast Library for Approximate Nearest Neighbors (FLANN) liefert eine Sammlung optimierter Algorithmen für schnelle KNN-Methoden [Featb].
Features2D: Das für diese Arbeit wohl wichtigste Modul bietet Implementierungen für nahezu alle bekannten Methoden der Detektion bzw. Deskription von Merkmalen[3]. Die patentierten Algorithmen SIFT und SURF befinden sich jedoch seit OpenCv 3 in einem seperaten Paket der Extramodule (*xfeatures2D*) und sind nicht kostenfrei kommerziell nutzbar. Dementsprechend muss ein spezieller Installationsvorgang für die zwei Zielplattformen verwendet werden, wie im Abschnitt 3.1 näher ausgeführt wird. Diese Module stellen alle für diese Arbeit wichtigen Strukturen und Funktionen in C++ bereit.

[1] OpenCv Official opencv.org (Abrufdatum: 9.10.17)
[2] Intel Official software.intel.com/en-us/articles/what-is-opencv (Abrufdatum: 9.10.17)
[3] Cv-features2D docs.opencv.org/3.1.0/d0/d13/classcv_1_1Feature2D.html (Abrufdatum: 9.10.17)

Matrixrepräsentation:
Ein häufig genutztes Konstrukt ist die C++-Klasse `Mat` zur Repräsentation einer Bildmatrix, wie in Listing 2.1 zu sehen[4]. .

```
1       Mat (int rows, int cols, int type)
```

Listing 2.1: Konstruktor der CV-Klasse Mat

Die Anzahl der Spalten (cols) wird durch die Breite und die Anzahl der Zeilen (rows) durch die Höhe des Bildes in Pixeln bestimmt. Der Typ eines `Mat`-Objekts wird einerseits durch dessen Farbtiefe (z.B. 8-Bit-Kodierung), sowie durch die Anzahl der *Kanäle* (engl. channel) bestimmt. Jeder Kanal kann als Grauwertbild in einer Hauptgrundfarbe des zugrundeliegenden Farbraumes (hier der RGB-, BGR- oder auch RGBA-Farbraum) gesehen werden. Kombiniert man die Kanäle, erhält man die tatsächlichen Bildfarben[5].
.

```
1     Mat greyscale_mat = new Mat(640, 480, 32FC1);
```

Listing 2.2: Erzeugung einer Grauwertmatrix in OpenCv

Listing 2.2 erzeugt somit eine 480x640-Matrix. `32FC1` bestimmt eine 32-Bit-Float-Farbkodierung und der einzige Channel führt zu einem Grauwertbild.

Punktrepräsentation:
OpenCv stellt eine eigene Klasse zur 2D-Punktrepräsentation zur Verfügung[6]. Dabei sind verschiedene Datentypen als Einträge möglich, wie beispielsweise `Point2D` oder `Point2F`, deren 2D-Koordinaten jeweils aus `Double`-oder `Float`-Werten bestehen.
Eine Repräsentation für detektierte Merkmalspunkte ist durch die Klasse `Keypoint` gegeben: .

```
1       cv::KeyPoint::KeyPoint(
2       Point2f _pt,
3       float   _size,
4       float   _angle = -1,
5       float   _response = 0,
6       int   _octave = 0,
7       int   _class_id = -1)
```

Listing 2.3: OpenCv-Feature-Klasse

Durch den Konstruktor in Listing 2.3 lässt sich für alle herkömmlichen Detektoren eine Merkmalsrepäsentation erzeugen[7]. Dabei setzt sich *pt* aus der x- und y-Koordinate des extrahierten Merkmals zusammen. *angle* gibt die Orientierung θ, sowie *octave* die entsprechende Oktave des Merkmals an. *response* ist der Rückgabewert des Detektors, der angibt wie *ausgeprägt* das Merkmal ist.

[4]OpenCv-Mat-Klasse docs.opencv.org/3.1.0/d3/d63/classcv_1_1Mat.html#details (Abrufdatum: 9.10.17)
[5]Channels http://www.imagemagick.org/Usage/color_basics/#channels (Abrufdatum: 9.10.17)
[6]OpenCv-Punkt-Klasse docs.opencv.org/3.1.0/db/d4e/classcv_1_1Point___.html#details (Abrufdatum: 9.10.17)
[7]OpenCv-Keypoint docs.opencv.org/3.1.0/d2/d29/classcv_1_1KeyPoint.html (Abrufdatum: 9.10.17)

Feature-Matching:
OpenCv stellt Implementierungen für die gängigsten Extraktions- und Matchingalgorithmen zur Verfügung.

```
1  static Ptr<SIFT> cv::xfeatures2d::SIFT::create(
2  int   nfeatures = 0,
3  int   nOctaveLayers = 3,
4  double  contrastThreshold = 0.04,
5  double  edgeThreshold = 10,
6  double  sigma = 1.6)
```

Listing 2.4: OpenCv-SIFT-Detektor

Listing 2.4 zeigt den Konstruktor des bereits vorgestellten SIFT-Detektors[8]. In der nativen C++-Implementierung können durch den Konstruktor Einstellungen am Detektor vorgenommen werden. Dazu gehören die maximale Anzahl detektierter Features *nfeatures*, die Anzahl der Oktaven, die nach Lowe standardmäßig auf drei eingestellt ist, einen Kantenschwellwert *edgethreshold*, mit dem sich die Stärke der Kantenunterdrückung einstellen lässt, sowie einen Kontrastschwellwert für die Extremabestimmung im DoG-Raum [Lowe 04]. Mit dieser SIFT-Instanz lassen sich die Merkmale sowie die Deskriptoren eines Bildes extrahieren. Das Matching zweier Deskriptorensets liefert eine Liste aus Matches, die durch die Klasse `DMatch` repräsentiert, wie Listing 2.5 zu entnehmen ist.

```
1        cv::DMatch::DMatch  (
2        int _queryIdx,
3        int   _trainIdx,
4        float   _distance)
```

Listing 2.5: OpenCv-Klasse-DMatch

Dabei lässt sich auf die zum Match zugehörigen Features über die Felder *queryIdx* und *trainIdx* zugreifen. In OpenCv wird das zuvor gelernte Bild als *trainimage* (i.e. das Urbild) und das *abfragende* Eingangsbild (i.e. das Teilbild) auch als *queryimage* bezeichnet. Das Feld *distance* gibt die Ähnlichkeit beider verglichener Merkmale an[9]. Auf die genaue Anwendung dieser und weiterer Funktionen wird in der Implementierung in Kapitel 4 eingegangen.

2.7 Wahl des Detektionsverfahrens

Die Wahl des Detektors ist abhängig von dessen Einsatzgebiet. Da für die hier entwickelte Applikation besonders Abstände und Qualitätsunterschiede des Ein- und Ausgangsbildes eine Rolle spielen, wird sich, wie aus den vorherigen Abschnitten bereits hervorgeht, für ein skalierungsinvariantes Verfahren entschieden.

[8] OpenCv-SIFT-Detektor docs.opencv.org/3.2.0/d5/d3c/classcv_1_1xfeatures2d_1_1SIFT.html (Abrufdatum: 9.10.17)
[9] OpenCv-DMatch-Klasse docs.opencv.org/3.1.0/d4/de0/classcv_1_1DMatch.html (Abrufdatum: 9.10.17)

Wie in Abschnitt 2.4 erwähnt, gilt der SURF-Detektor in der Literatur im Allgemeinen als schnelleres und effizienteres Verfahren als SIFT. Die Vorabuntersuchung zeigt jedoch, dass SIFT für das Szenario dieser Arbeit die zuverlässigeren Ergebnisse liefert. Dabei wurden Urbild-Teilbild-Tupel testweise an ein OpenCv-Python-Skript übergeben. Dieses wandelt das Bild in ein Graustufen-Bild um und führt an diesem das Feature-Matching mit dem ebenfalls in ein Graubild umgewandelten Urbild durch. Der Test an verschiedenen Instanzen hat zeigt, dass der SURF-Algorithmus OpenCv's zwar deutlich mehr Features produziert und im Verhältnis dazu eine kürzere Bearbeitungszeit aufweist, jedoch keine so zuverlässigen Antworten liefert, wie SIFT. Auffällig ist, dass eine große Anzahl der Instanzen mit SURF *false-positive*-Antworten generiert.
Auch in einer Arbeit von Saleem et. al. aus der International Conference for Image Analysis and Recognition (ICIAR) zur Untersuchung von Matchingverfahren an Spektralbildern wird experimentell gezeigt, dass das SIFT-Verfahren SURF gegenüber einige Vorteile bietet [Sale 12]. Dabei wird als Maß für die Matchinggenauigkeit p die Formel

$$p = \frac{corrent\_matches}{correct\_matches + false\_matches} \tag{2.22}$$

verwendet [Sale 12, S.168].
Eine Zusammenfassung der Ergebnisse kann Abb. 2.11 entnommen werden.

	Scale	**Rotation**	**Noise**
SIFT	best	best	good
SURF	good	good	best

Abbildung 2.11: SIFT vs. SURF Ergebnisse
(Quelle [Sale 12])

Auch Teile der OpenCv-Community konnten, anhand diverser Matchingverfahren mit SIFT und SURF, ähnliche Beobachtungen machen[10].
Da das SIFT-Verfahren sich hier nach einer Iteration von Testinstanzen als sehr stabil herausstellt, wird sich hier auf diesen Algorithmus festgelegt.

[10] Diskussion SIFT-SURF stackoverflow.com/questions/11172408/surf-vs-sift-is-surf-really-faster (Abrufdatum: 14.10.17)

3 Entwurf und Konzeptionierung

Dieses Kapitel behandelt die Anforderungen beider Anwendungen, sowie das Konzept und den funktionalen Zusammenhang aller beteiligten Komponenten. Die Anwendungen sollen der Zielsetzung aus 1.3 genügen.
Hierzu wird ein webbasiertes *Teilbild-Matching*-Verfahren basierend auf dem in Kapitel 2 vorgestellten Feature-Matching entworfen, das als Eingabe ein Teilbild erwartet, und als Ausgabe, die hinterlegten Informationen für die darin enthaltenen Polygone liefert. Der entwickelte Prozess wird anschließend auf ein Echtzeit-Matching-Verfahren übertragen, welches als Kern der nativen Android-Applikation fungieren soll.
Die Grundkenntnisse der jeweiligen Programmiersprache werden in den folgenden Kapiteln als vorausgesetzt angesehen und nur oberflächlich auf die system- und sprachenabhängigen Besonderheiten eingegangen. Vielmehr wird lediglich die Umsetzbarkeit einer Anwendung mittels eines Feature-Matching-Verfahrens basierend auf dem Framework OpenCv untersucht.
Eine Beschreibung der Anwendungsszenarien ist bereits 1.1 zu entnehmen. Abb. A.1 zeigt, wie sich diese auf ein allgemeines Anwendungsfalldiagramm reduzieren lassen, das für beide Applikationen gilt. Gemäß der Zielsetzung werden mobile Lösungen für diesen Anwendungsfall in Form einer Website sowie einer nativen Android-Applikation benötigt. Die notwendigen Komponenten sowie deren Zusammenhänge sind der Abb. 3.1 zu entnehmen.

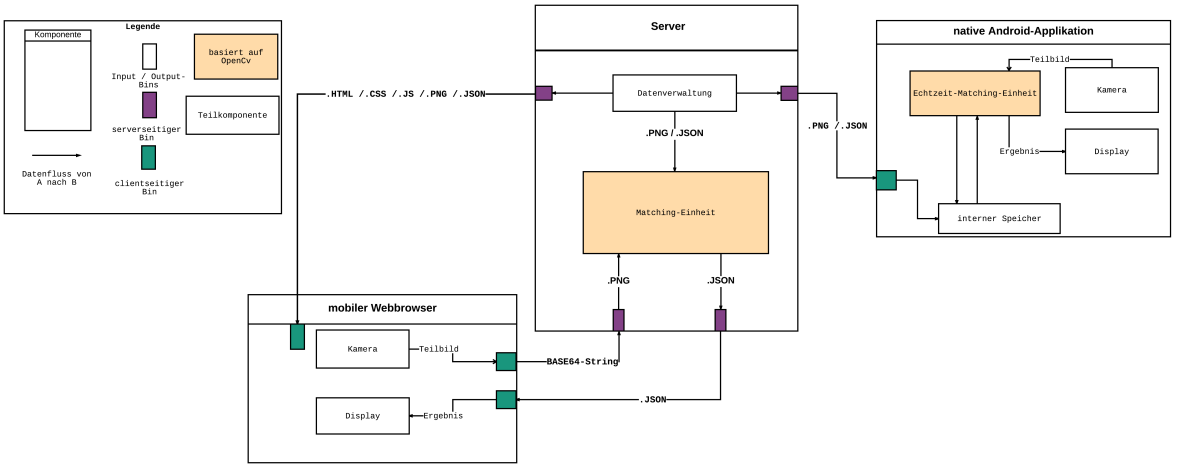


Abbildung 3.1: Aufbau und Zusammenhang der Architekturen

Die Bins einer Komponente stellen dabei Kommunikationsschnittstellen, wie beispielsweise clientseitige HTTP-Anfragen an PHP-Skripte des Servers zu anderen Komponenten dar. Die serverseitigen Bins dienen oft z.B. in Form von PHP-Skripten nur der Weiterleitung von Daten. Der Datenfluss der Kommunikation ist durch gerichtete Kanten, sowie die Übertragung relevanterer Datentypen durch Beschriftung dieser gekennzeichnet.

Die Funktionalität der Webapplikation ergibt sich aus den clientseitigen Funktionalitäten des Browsers, sowie denen des Servers. Dabei geschehen z.B. die Aufnahme des Teilbildes, sowie dessen Augmentierung um hinterlegte Textinformationen, im mobilen Browser und das Matching des Teilbildes mittels OpenCv seitens der Serverkomponente.
Die native Applikation stellt ein weitestgehend von externen Komponenten unabhängiges System dar (s. Abb. 3.1).
Es bedarf lediglich einer einmaligen Versorgung mit den zu den Bildinstanzen hinterlegten Daten. Alle weiteren notwendigen Funktionalitäten, insbesondere das Teilbild-Matching selbst, ergeben sich durch die lokalen Teilkomponenten der Android-App.
Diese lokale Umsetzung ist vor allem auf Aussagen des Felix-Nussbaum-Hauses zurückzuführen, das das einmalige Beziehen der Daten und eine ansonsten verbindungslose Lösung wünschenswerter findet, als z.B. eine Webapplikation. Aufgrund der nicht flächendeckenden WLAN-Versorgung und der durch die Bauart bedingte meist fehlende UMTS-Versorgung ist für das Museum besonders interessant, dass die benötigten Daten über alle Gemälde des Hauses nur einmal zu Anfang auf das Gerät geladen werden. Da die native Applikation, wie im Falle des Felix-Nussbaum-Hauses auf speziell dafür ausgelegten Geräten laufen könnte spielt der hohe Speicheraufwand, der durch das Kopieren aller bekannten Daten auf das Gerät entsteht, im Weiteren vorerst keine Rolle.
Im folgenden Verlauf wird die Vorbereitung der Architekturen aus Abb. 3.1 sowie das Konzept vorgestellt, das beiden Anwendungen zugrunde liegt, bevor die plattformspezifischen Implementierungen im darauf folgenden Kapitel betrachtet werden.

3.1 Vorbereitung der Architektur

Dieser Abschnitt beschreibt die Serverarchitektur, die Vorbereitung der Datenschicht die von beiden Applikationen verwendet wird sowie die Einrichtung OpenCvs, das für die Konstruktion der beiden Matching-Einheiten notwendig ist (s. Abb. 3.1).

3.1.1 Der Server

Für das Hosting der Website, sowie zur Verwaltung der vorhandenen Daten (s. Abb 3.1) wird Webspace an der Universität Osnabrück gemietet. Dieser befindet sich auf einem Apache-Server und ist ansprechbar unter der Adresse:

project2.informatik.uni-osnabrueck.de/ardetection/

Auf diesem werden alle Berechnungen bezüglich der Bildverarbeitung mithilfe von OpenCv seitens der Website laufen. Des Weiteren werden auf diesem vorerst alle hinterlegten Daten zusammengeführt, die von verschiedenen Anwendungen genutzt werden können.
Da es sich um einen Apache-Server handelt, bietet sich PHP[1] als serverseitige Programmiersprache an. Weil jedoch kein offizielles OpenCv-PHP-Interface für OpenCv exisitert, wird das Python-Interface[2] für die serverseitige Matching-Einheit verwendet.

[1] Hypertext PreProcessor php.net (Abrufdatum: 11.10.17)
[2] Python OpenCv opencv-python-tutroals.readthedocs.io/en/latest/ (Abrufdatum: 11.10.17)

Für die Entwicklung der nativen App wird das offizielle Android bzw. das Java-Interface[3][4] verwendet.
Je nach Interface und benötigten Funktionen OpenCvs muss ein spezieller Installationsvorgang des Frameworks stattfinden.

3.1.2 Installation von OpenCv

Die Einrichtung der Bibliothek gestaltet sich für unterschiedliche Plattformen und Ansprüche unterschiedlich aufwendig.
Da für die Applikation Zugang zu dem patentierten SIFT-Algorithmus benötigt wird, muss die Kompilierung der Bibliothek manuell geschehen. Hierzu wird sowohl für die App als auch die Website das öffentliche Repository von OpenCv[5] verwendet, welches auf die aktuelle Version OpenCv 3 aufbaut.
Seit OpenCv 3 befinden sich die patentierten Algorithmen jedoch in einem Paket namens *Extra Modules* des seperaten Repositorys *OpenCv Contrib*[6] und müssen deshalb manuell in den Kompiliervorgang von OpenCv integriert werden. Da OpenCv nativ in C++ programmiert wurde, muss die Bibliothek außerdem erst auf die entsprechende Plattform portiert werden.

3.1.2.1 OpenCv auf dem Server

Sowohl die Hauptbibliothek, als auch die Extramodule werden auf den Server geladen. Mittels der internen *CMakeLists.txt* lassen sich verschiedene Parameter für die Kompilierung mitgeben.
Neben Flags für die Aktivierung von Optionen für bestimmte Grafikkarten, lassen sich ebenfalls umgebungsabhängige Variablen setzen. Mit der Zeile
cmake -DOPENCV_EXTRA_MODULES_PATH=<opencv_contrib>/modules <opencv_source_directory>
werden die Extra-Module in das Haupt-Modul von OpenCv integriert. Ebenfalls wichtig ist die Konstante *CMAKE_BUILD_TYPE*, die auf RELEASE gesetzt werden sollte, da nur so die optimierten Versionen der Algorithmen bei der Kompilierung verwendet werden. Weitere Eintellungsmöglichkeiten können der CMake[7]-Konfigurationsdatei und dem offiziellen OpenCv-Tutorial[8] entnommen werden[9]. Nach der Kompilierung lässt sich OpenCv einfach in das gewünschte Python-Skript importieren.

[3] Java OpenCv docs.opencv.org/2.4/doc/tutorials/introduction/desktop_java/java_dev_intro.html (Abrufdatum: 11.10.17)
[4] Android OpenCv opencv.org/platforms/android/ (Abrufdatum: 11.10.17)
[5] OpenCv Root Repository github.com/opencv (Abrufdatum: 11.10.17)
[6] Extra Modul Repository github.com/opencv/opencv_contrib (Abrufdatum: 11.10.17)
[7] CMAKE cmake.org/ (Abrufdatum: 11.10.17)
[8] docs.opencv.org/trunk/d7/d9f/tutorial_linux_install.html (Abrufdatum: 11.10.17)
[9] Opencv CMake github.com/opencv/opencv/blob/master/CMakeLists.txt (Abrufdatum: 11.10.17)

3.1.2.2 OpenCv für Android

Da die Echtzeit-Matching-Einheit (s. Abb. 3.1) ihre Berechnungen in der nativen App lokal ausführen soll, muss auch hier die Bibliothek eingerichtet werden. Für die Android-Plattform gibt es wieder unterschiedliche Möglichkeiten das OpenCv-System aufzusetzen [Deve].

Asnychrone Initialisierung:
Durch wenige Anweisungen können nach Start der Anwendung, durch den sogenannten *OpenCv-Manager*, alle notwendigen OpenCv-Module geladen werden. Ist der Manager nicht auf dem Gerät installiert, wird er automatisch aus dem Google-Playstore geladen. In diesem Fall wird jedoch nur die Hauptbibliothek selbst geladen. Um die Extramodule für den SIFT-Algorithmus zu integrieren, wird auch hier das manuelle CMake-Schema angewendet und die bei der Kompilierung erzeugte *.so-Datei* anschließend *statisch* in der benötigten Android-Activity geladen werden. Dieses hier verwendete Schema nennt sich *statische Initialisierung*.

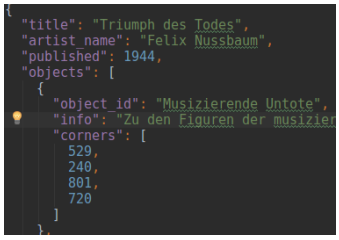
Statische Initialisierung:
Die notwendigen Java-Dateien können wieder via CMake aus den Extramodulen und dem Hauptmodul erzeugt werden, wie es bereits auf dem Server der Fall ist. Jedoch wird hier kein Python, sondern ein Java-Interface benötigt. Wie dem angegebenen Tutorial zu entnehmen ist, muss CMake hier zusätzlich unter anderem das `Android-SDK`[10], sowie das `Android-NDK`[11] mitgegeben werden, da der Zugriff auf die C++-Funktionen ermöglicht werden soll. Mit Hilfe des `JNI`[12] und nach Integration der kompilierten .so-Datei in ein neu aufgesetztes Android-Projekt, können die nativen C++-OpenCv-Funktionen durch einen Java-Wrapper angesprochen werden.
Hier fällt auf, dass dieser Kompiliervorgang nur für eine Prozessorfamilie pro Vorgang ausgelegt ist. Um Multi-Plattform-Support zu gewährleisten, lässt sich jedoch die erzeugte Android-*APK* aufteilen, so dass für alle verschiedenen Prozessorarchitekturen unterschiedliche Versionen der Applikation zum Download angeboten werden können. Hierzu wird die passende *.so*-Datei aus dem Kompilierungsvorgang erzeugt und ebenfalls in das Projekt integriert, um mittels *Gradle* anschließend den *Multiple-Apk-Build* durchzuführen[13].
Da die Veröffentlichung der App jedoch vorerst nicht vorgesehen ist und die Entwicklung an einem Samsung Galaxy S8 erfolgt, wird sich hier auf die `x86`-Serie von Intel festgelegt.

3.1.3 Aufbereitung der Rohdaten

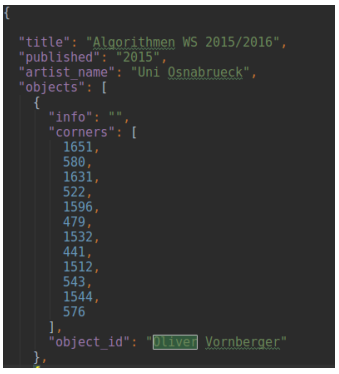
Damit der Anwendung unabhängig vom Szenario Hintergrundinformationen jeder realen Bildinstanz zur Verfügung stehen, werden diese in einem vereinheitlichten und übertragbaren Format in der Datenschicht des Servers abgelegt.
Zu Beginn dieser Arbeit war lediglich die Reihe der Algorithmenfotos der Universität

[10] Android Software Development Kit developer.android.com/studio/index.html (Abrufdatum: 11.10.17)
[11] Android Native Development Kit developer.android.com/ndk/index.html (Abrufdatum: 11.10.17)
[12] Java Native Interface docs.oracle.com/javase/6/docs/technotes/guides/jni/ (Abrufdatum: 11.10.17)
[13] developer.android.com/studio/build/configure-apk-splits.html (Abrufdatum: 11.10.17)

Osnabrück, sowie die zugehörigen Polygoninformationen gegeben, die im Quelltext des entsprechenden HTML-Dokumentes zu finden sind. Aufgrund einer Umstellung des internen Systems der Universität ist dies jedoch nur für die Jahrgänge 2013 bis 2016 der Fall. Es ist möglich, in allen Fällen in denen die Polygoninformationen nicht vorhanden sind, mittels eines Tools die Polygonkanten um jedes Objekt oder eine allgemeinere Region im Bild einzuzeichnen, um anschließend die Koordinaten samt Urbild in einem einheitlichen Format abzuspeichern. Dies ist jedoch nicht Teil der Zielsetzung, weshalb die Anwendungen vorerst nur auf den genannten Jahrgängen aufbauen. Da der Aufbau eines effizienten Datenbanksystems zur einheitlichen und effizienten Verwaltung der Daten den Rahmen dieser Arbeit sprengen würde, wird hier ein lokales dokumentenorientiertes Datenverwaltungssystem verwendet. Das Datenformat wird so gewählt, dass es sowohl die Algorithmenfotos, als auch die Gemälde eines Museums beschreibbar macht. Die Anwendungen benötigen zwei Ressourcen zu jeder Bildinstanz, zum Einen das Urbild, und zum Anderen eine Repräsentation der Information des Bildinhaltes.
Unter dem Pfad */public/resources/data* des Webspaces kann für jeden gewünschten Ort ein Ressourcenordner angelegt werden, wie in diesem Fall entsprechend der zwei Modi */studentMode* und */museumMode*. In jedem dieser Ordner sind die n Urbilder des Ortes im `PNG`[14]-Format abgelegt.
Zur Vereinheitlichung der gegebenen Informationen zu jedem Bild wird das JSON[15]-Format verwendet. Mit diesem wird ein einheitliches Datenformat über mehrere Anwendungen gewährleistet.



(a)



(b)

Abbildung 3.2: JSON-Daten der verschiedenen Modi

Wie in Abb. 3.2 zu sehen wurden die Informationen des Gemäldes (a) und des Gruppenfotos (b) noch um Daten des Gesamtbildes erweitert, wie dem Künstlernamen, dem Erscheinungsjahr und dem Titel des Bildes.
Für die Studenten bleibt bis auf den Namen und deren Polygon-Koordinaten weitere Info über jedes Teilobjekt aus. Über ein Museum sind neben dem Namen und Koordinaten eines Teilobjektes bzw. einer Region noch weitere geschichtliche Info vorhanden.
Die Anwendung ist so konstruiert, dass lediglich das Urbild, sowie die zugehörige JSON-Datei im entsprechenden Ordner abgelegt werden müssen, um diese in beide Applikationen zu integrieren.

[14]Portable Network Graphics http://www.libpng.org/pub/png/ (Abrufdatum: 11.10.17)
[15]JavaScript Object Notation http://www.json.org/ (Abrufdatum: 11.10.17)

3.2 Anforderungsbeschreibung

Die Anforderungen an die Komponenten aus Abb. 3.1 können aus dem Anwendungsfalldiagramm in Abb. A.1 in abstrahierter Weise abgeleitet und somit in funktionale und nicht-funktionale Anforderungen gegliedert werden, die für beide Anwendungen gelten:

notwendige funktionale Anforderungen:

- `Selektion des Urbildes`: Da sich der User vor n unterschiedlichen Bildern befinden könnte, muss die Anwendung das *Urbild* der kommenden Teilbilder erkennen können.
- `Aufnahme des Teilbildes`: Um ein Teilbild zu verarbeiten, muss es zunächst erfasst werden. Je nach Komponente ergeben sich hier unterschiedliche Möglichkeiten.
- `Matching des Teilbildes mit dem Urbild`: Pro Applikation wird eine *Matching-Einheit* benötigt, die als Input ein Teilbild erwartet und dieses im Urbild sucht. Ist ein Match entstanden, müssen die hinterlegten Polygone die in der gefundenen Urbild-Region liegen selektiert und deren Information als Antwort zurück geliefert werden.
- `Darstellung der Matching-Antwort`: Je nach Anwendungsfall und den damit einhergehenden darzustellenden Informationen sollen dem User die resultierenden Informationen adäquat präsentiert werden.

notwendige nicht-funktionale Anforderungen:

- `Effizienz`: Der User soll mit möglichst wenig Kosten konfrontiert werden. Dementsprechend gilt es bei der Implementierung sowohl Rücksicht auf die Verarbeitungszeit der Algorithmen, als auch hinsichtlich des entstehenden Internettraffics zu nehmen.
- `Stabilität`: Die Anwendungsergebnisse sollen möglichst unabhängig von Deformationen des aufgenommenen Bildes sein, da sich z.B. die Aufnahmequalität älterer und neuerer Kameras unterscheidet und die Umgebungsbeleuchtung von Ort zu Ort variiert. Außerdem lässt sich davon ausgehen, dass jeder Anwender mit einer unterschiedlichen Distanz an das Originalbild herantreten / zoomen wird. Der bereits im Abschnitt 2.3 beschriebene Algorithmus liefert relativ stabile Ergebnisse bezüglich Beleuchtungs-und Skalierungsänderungen des Bildes.
- `Erweiterbarkeit`: Das entwickelte Kernverfahren soll möglichst allgemein gehalten werden. Eine modularisierte Entwicklung und ein stabiler Anwendungsprozess sollen gewährleisten, dass die Applikation auf möglichst viele Anwendungsfälle erweiterbar ist.
- `Usability`: Es sollen intuitive Oberflächen generiert werden, die einem User einen einfachen Zugang zu den Anwendungen bieten. Ein Algorithmus der außerdem länger als zehn Sekunden in seiner Verarbeitung benötigt wäre für solche Szenarien nicht akzeptabel aber vor allem bei Bildern größerer Auflösung durchaus realistisch.

Aus diesem Grund wird ebenfalls versucht die Ausführungszeit der Matching-Einheiten möglichst gering zu halten. Besonders bei einer Echtzeit-Applikation führen hohe Ausführungszeiten zur nicht-flüssigen Darstellung des augmentierten Inhalts.

Bei dem Entwurf und der Implementierung gilt es die funktionalen Anforderungen prototypisch zu erfüllen, sowie eine ausgeglichene Realisierung der nicht-funktionalen Anforderungen zu erreichen.
Aus den funktionalen Anforderungen kann der notwendige Kernprozess abgeleitet werden, der die Basis beider Applikationen bildet und je nach Anwendung in die dieser integriert wird, unterschiedlich oft durchlaufen wird, wie in Abb. 3.3 zu sehen ist.

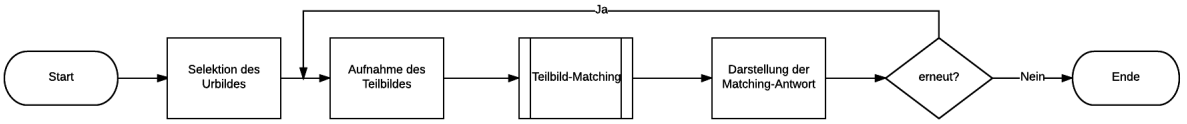


Abbildung 3.3: Allgemeiner Anwendungszyklus pro Urbild. Der Kernprozess *Aufnahme-Teilbild-Matching-Darstellung* kann beliebig oft durchlaufen werden

Im folgenden Abschnitt wird die Integration eines solchen Prozesses in eine Webapplikation, sowie anschließend die Übertragbarkeit der Lösung auf eine native Echtzeitanwendung, die den Kernprozess mehrmals pro Sekunde durchführen kann, vorgestellt.

3.3 Entwurf einer webbasierten Lösung

Zunächst wird das webbasierte Matching-Verfahren vorgestellt, das ein vom mobilen Webbrowser übertragenes Teilbild analysiert und diesem die von einer Matching-Einheit detektierten Informationen in einem einheitlichen Format zurückliefert (s. Abb. 3.1). Der Kernprozess in Abb. 3.3 wird in der Webapplikation genau einmal durchlaufen, ohne dass der Anwender einen erneuten Vorgang verifiziert. In Abb. A.2 ist das zugehörige Aktivitätsdiagramm zu sehen. Aus dem notwendigen Kernprozess lässt sich außerdem ein Sequenzdiagramm der webbasierten (Teil-)Komponenten ableiten, welches der Abb. A.5 entnommen werden kann. Nachfolgend wird der einfache Anwendungszyklus und dessen Integration in eine Webanwendung vorgestellt.

3.3.1 Selektion des Urbildes

Der erste Teil des Anwendungszyklus befasst sich mit der Erkennung der Bildinstanz, vor der der Anwender steht. Der genaue Anwendungsfall ist in Tabelle A.1 zu sehen. Um die notwendigen Daten aus dem Teil- und dem Urbild im späteren Teilbild-Matching zu extrahieren, muss die Anwendung das Urbild und die zugehörigen Informationen auf dem Server finden. Wäre die Menge der möglichen Daten beschränkt und die Wartezeit für die Berechnung entsprechend vertretbar, hätte auch hier ein Feature-Matching-Prozess verwendet werden können. Um die Applikation jedoch um ein Matching-Verfahren zu entlasten, wird sich für die Benutzung eines QR-Codes entschieden. Dieser könnte vor

der jeweiligen Bildinstanz, sowohl in der Universität, als auch im Museum angebracht werden.
Die Codes beinhalten einen einzelnen Link als Textinformation. Dieser besteht aus dem Host der Daten, dem Modus in dem die Anwendung aufgerufen werden soll und dem Namen des Urbildes. Die im QR-Code enthaltene Zeichenkette für das Algorithmenfoto 2015 sieht z.B. wie folgt aus:

project2.informatik.uni-osnabrueck.de/ardetection/studentMode/2015/

Der Modus dient zur Unterscheidung in welcher Form die Matching-Antwort präsentiert wird und zur Identifikation des zuständigen Ressourcenordners. Zudem kommt dies der Erweiterbarkeit zugute, da für jedes Szenario eine eigene Darstellungsmöglichkeit angelegt werden kann.
Der Name des Urbildes dient zur Identifikation einer Ressource auf dem Server. Der Bildname ist zwar eventuell kein eindeutiger Schlüssel, wird jedoch aufgrund der kleinen Datenmenge dieser Arbeit vorerst vernachlässigt. Aufgrund dessen erwartet die Anwendung, dass das Urbild sowie die zugehörige JSON-Datei abzüglich der Dateiendung unter dem selben Namen abgespeichert sind. Mittels eines externen QR-Scanners kann sich der Nutzer auf die entsprechende *index.php* des entsprechenden Urbildes weiterleiten lassen, wie Abb. B.1b zeigt. Dort werden dem Anwender Hinweise zur Funktionalität der Anwendung, sowie einige Hintergrundinformationen zur Bildinstanz, vor der er sich befindet, in einem Infofeld gegeben.

3.3.2 Aufnahme des Teilbildes

Nachdem die Anwendung registriert vor welchem Bild sich der Anwender befindet, kann mit der Smartphone-Kamera ein Teilbild des Bildes aufgenommen werden und der eigentliche Prozesszyklus beginnen. Aufgrund der Struktur des folgenden Teilbild-Matching-Prozesses der Matching-Einheit kann das Teilbild sogar dem gesamten Urbild entsprechen und darüber hinaus. Tabelle A.2 zeigt noch einmal den zugehörigen Use-Case.
Das Öffnen einer externen Kamera-Applikation aus dem mobilen Webbrowser heraus senkt erfahrungsgemäß vor allem bei mehrfacher Aufnahme die angestrebte Usability. Dementsprechend wird sich hier für ein eingebettetes System entschieden, um die Kernanwendung als einfache `Single-Page-Application`[16] gestalten zu können.
Zur Integration der Kamera wird ein möglichst plattformunabhängiger Standard benötigt, um die Kamera möglichst unabhängig von Hardware und Betriebssystem des Endgerätes ansprechen zu können.

Die verbreitetsten Betriebssysteme sind die Android-, sowie die IOS[17][18]-Plattform (s. Abb. 3.4). Auf IOS-Geräten befindet sich der vorinstallierte Safari- sowie auf Android-Geräten der Chrome-Browser, weshalb sich bei der Entwicklung auf diese Browser konzentriert wird.

[16] SPA www.codeschool.com/beginners-guide-to-web-development/single-page-applications (Abrufdatum: 11.10.17)
[17] Android official www.android.com/ (Abrufdatum: 11.10.17)
[18] IOS official www.apple.com/de/ios/ios-11/ (Abrufdatum: 11.10.17)

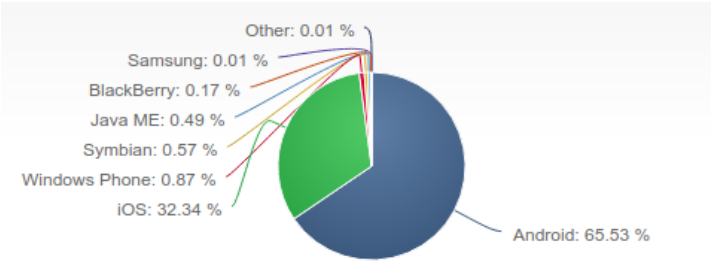


Abbildung 3.4: Verteilung der verbreitetsten mobilen Betriebssysteme, Stand 2017 www.netmarketshare.com

Hier wird der moderne `WebRTC`-Standard[19] als Lösung, plattformunabhängige Hardwarezugriffe zu ermöglichen, gewählt. Mittels der bereit gestellten Funktion *getUserMedia()* wird die smartphone-interne Kamera angesprochen. Zwar befindet sich der Standard noch in der Entwicklungsphase, und ist noch nicht mit allen Plattformen und Browserversionen kompatibel (s. Abb. 3.5), jedoch ausreichend für diese Arbeit, da das Ziel der Arbeit nicht die marktreife Entwicklung, sondern die Untersuchung der generellen Umsetzbarkeit ist. In Kombination mit HTML5[20], das unter Anderem neue Mediaele-

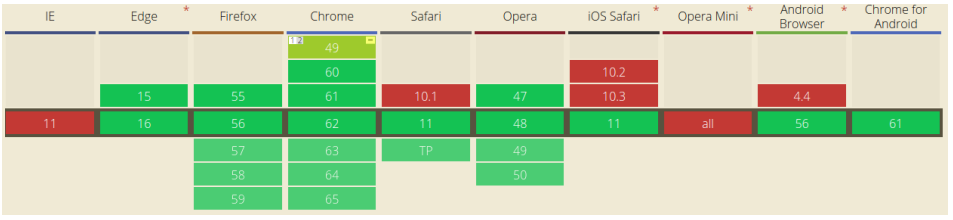
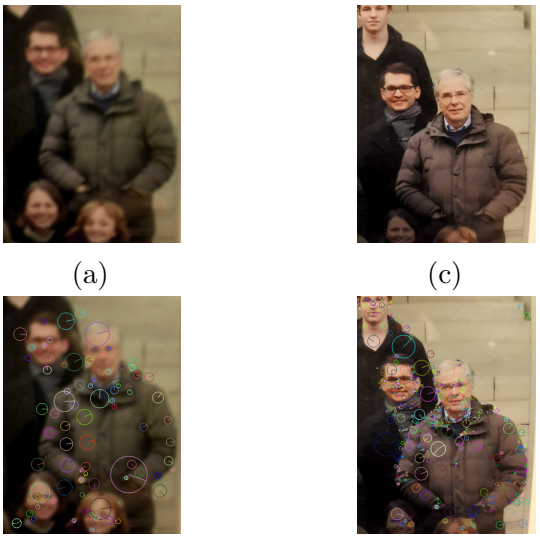


Abbildung 3.5: GetUserMedia-Support-Tabelle (Quelle caniuse.com/#feat=stream)

mente, wie das *<video>*-Element, zur Verfügung stellt, und WebRTC ist es möglich einen Videofeed in die Weboberfläche zu integrieren.
Damit dieser funktioniert muss, z.B. aufgrund der Sicherheitsrichtlinien von Google-Chrome., die HTTP-Anfrage der index.html unter `HTTPS` erfolgen[21]. Außerdem muss der Nutzer manuell die Kameraberechtigung freigeben, nach der WebRTC fragt, siehe Abb. B.1a. Sind alle Berechtigungen erteilt, kann der Anwender Teilbilder des gewünschten 2D-Kontextes aufnehmen.
Die Bilder entsprechen Momentaufnahmen des Streams, der durch die native Kamera empfangen wird. Die Aufnahme und die anschließende Weiterleitung des Bildes an den Server geschieht durch die Verifizierung des Anwenders, indem dieser mit seinem Finger auf das Videoelement tippt. Das Video wird aus- und das aufgenommene Teilbild eingeblendet, bevor der Browser das Teilbild automatisch an den Server sendet. Der Anwender wird nach der Aufnahme des Teilbildes in einen wartenden Zustand versetzt, der durch ein Ladeoverlay signalisiert wird, wie in Abb. B.2a zu sehen ist. Die Qualitätsansprüche an das aufgenommene Teilbild sind entscheidend. Je mehr Deformationsaspekte auf

[19] Web Real-Time Communication webrtc.org/ (Abrufdatum: 11.10.17)
[20] HTML5 www.w3schools.com/html/html5_intro.asp (Abrufdatum: 11.10.17)
[21] HTTP Secure www.elektronik-kompendium.de/sites/net/1811281.htm (Abrufdatum: 11.10.17)

das Bild wirken, desto stärker wird die Anzahl, sowie die Stabilität der extrahierten Features durch die Matching-Einheit, beeinflusst. In Abb. 3.6 ist exemplarisch der Qua-



(a) (c)

(b) (d)

Abbildung 3.6: Beispiel unterschiedlicher Bildqualitäten durch verschiedene Kameras des Samsung Galaxy S3 (a) und Samsung Galaxy S8 (c) mittels WebRTC der Webapplikation. Aufgenommen aus einem ähnlichen Betrachterwinkel unter gleicher Beleuchtung und ähnlicher Skalierung (Differenzen durch Verwackelung). Die Kreise in (b) und (d) stellen die extrahierten SIFT-Merkmale mit ihrer Größe (die in den Deskriptor einbezogene Region um das Merkmal) und Orientierung dar.

litätsunterschied zwei verschiedener Smartphonekameras der Firma Samsung und die daraus resultierenden Unterschiede der Merkmalsextraktion zu sehen. Während aus dem qualitativ hochwertigerem Teilbild (d) 531 Features extrahiert werden können, werden aus Teilbild (b) nur noch lediglich 108 Merkmale gewonnen.
Der negative Einfluss der Bildqualität auf das Matchingergebnis durch den skalierungsinvarianten Detektor kann jedoch, wie aus der Funktionalität der entwickelten Anwendung hervorgeht, gering gehalten werden.

3.3.3 Matching des Teilbildes

Dieser Abschnitt stellt die Funktionsweise der Matching-Einheit der Webapplikation, in der das Vergleichen des Teilbildes mit dem Urbild (Teilbild-Matching) stattfindet, vor. Die Einheit entspricht hier einem Python-Skript auf dem verwendeten Server.
Mittels einer HTTP-Anfrage an ein PHP-Skript des Servers wird das aufgenommene Teilbild aus dem mobilen Browser nach Verifizierung durch den Anwender zur Verarbeitung in Form eines Base64-Strings an den Server geschickt. Durch PHP wird ein Python-Skript aufgerufen, um mit Hilfe von OpenCv alle notwendigen Teilbild-Matching-Operationen durchzuführen. Tabelle A.3 kann nochmals der genaue Anwendungsfall entnommen werden. Der daraus abgeleitete Programmablaufplan ist A.3 zu entnehmen.
Wie im Folgenden zu sehen, lässt sich das Teilbild-Matching in zwei Phasen unterteilen:

Phase 1 - Feature-Matching des Teil-und Urbildes:
Zunächst werden die Merkmale des Ur- sowie des Teilbildes, mittels des in 2 vorgestellten SIFT-Verfahrens durch Funktionen des OpenCv-Python-Interface extrahiert und die zugehörigen Deskriptoren berechnet, um die beiden Bilder anschließend miteinander vergleichen zu können. Die notwendigen Urbilddateien liegen im .png-Format in der Datenverwaltung des Servers.
Der Vergleich geschieht effizient mit der KNN-Methode des OpenCv-FLANN-Moduls. Nachdem mit dem KNN-Resultat ein Ratio-Test nach Lowe [Lowe 04] durchgeführt wird, sind bestenfalls nur noch gute Matches vorhanden (s. 2.5). Die zugehörigen Features der Matches können einfach mittels der Raumkoordinaten selektiert werden.

Phase 2 - Lokalisierung der sichtbaren Polygone:

Um den Matching-Prozess zu vervollständigen, müssen die gewonnenen Informationen über die Lage des Teilbildes im Urbild analysiert werden, um die sichtbaren Polygone (Personen der Algorithmenfotos, oder Regionen/Objekte in einem Gemälde) auf dem Teilbild zu identifizieren.
Aus den nach dem eigentlichen Feature-Matching übrig gebliebenen Merkmalen der besten Matches kann mittels des vorgestellten RANSAC-Algorithmus die Transformationsmatrix bestimmt werden, wie bereits in 2.5 erläutert. Die Matching-Einheit der Applikation kann diese Matrix nun auf die Eckkordinaten des Teilbildes anwenden um diese perspektiv zu transformieren. Nach erfolgreicher Verarbeitung entsteht so eine Zielregion Z im Urbild-Raum, wie in Abb. 3.7 zu sehen ist.

Abbildung 3.7: Als Beispiel das Algorithmenfoto 2015 (rechts), sowie das gewünschte Teilbild (links). Die hinterlegten Polygondaten (rot) sind mit ihrem Zentrum eingezeichnet. Aus dem Feature-Matching des ersten Schrittes (grün) kann die transformierte Region (weiß) gefunden werden.

Z kann durch Einstellung des RANSAC-Schwellwerts T relativ genau bestimmt werden, so dass der Fehler zwischen Z und der tatsächlichen Region minimal wird. Hier hängt die erkannte Region offensichtlich stark von den Inputdaten ab.
Da die Transformationsmatrix auf die Ecken des Teilbildes angewendet wird, entsteht

im Urbildraum eine ebenfalls viereckige Region, die ggf. irrelevante Information beinhaltet. Eine andere Möglichkeit ist, das Eingangsbild auf die darin enthaltenen Polygone zuzuschneiden, so dass nur die Polygone selbst als Eingangsbild fungieren. Dies würde als Ausgabe der Transformation in den Urbildraum eine Region produzieren, die ebenfalls den Polygonen selbst ähnelt und keine irrelevante Information beeinhaltet. Für den Anwendungsfall der Algorithmenfotos beispielsweise hätte sich ein Gesichtserkennungsalgorithmus vor die Matching-Einheit schalten lassen, um das Teilbild nur auf die gefundenen Gesichter zuzuschneiden. Der genauere Input würde auch zu genauerem Output führen, bedeutet jedoch Einschränkungen in den Use-Cases, da ein Gesichtserkenner nicht für ein Museum geeignet ist. Da die Anwendung übertragbar auf verschiedene Szenarien sein soll, müssten prinzipiell sowohl konvexe, als auch nicht konvexe Polygone berücksichtigt werden.
Einen Konturendetektor zu entwickeln, der diese zuverlässig erkennt bedeutet jedoch unabsehbaren Mehraufwand in der Programmierung auf den aufgrund des begrenzten Rahmens dieser Arbeit verzichtet wird. Hier wird sich deshalb für die regions- und nicht objektbasierte Transformationsmethode entschieden.
Ist die Zielregion Z als Liste von Punkten bestimmt, können die Polygone innerhalb der Region gefunden werden indem überprüft wird, ob ihr Zentrum in Z liegt. Ein noch genaueres Resultat würde man erhalten, wenn man für jede Kante der hinterlegten Polygone prüft, ob sie in Z liegt und das Polygon nur ab einem Schwellwert in der Region enthaltener Kanten akzeptieren. Bei wenigen Polygonen scheint dies denkbar, hat sich jedoch als zu teuer in der Verarbeitungszeit herausgestellt, falls es sich um ein informationsreiches Bild mit vielen hinterlegten Polygonkanten handelt. Die hier verwendete Herangehensweise liefert jedoch für die Szenarien dieser Arbeit hinreichend gute Ergebnisse.
Sind die Polygone in der Urbildregion gefunden, basieren diese auf Modellkoordinaten des Urbildraumes und müssen erst in Bildschirmkoordinaten transformiert werden. Dies kann mit der noch vorhandenen Transformationsmatrix geschehen, die durch den RANSAC-Algorithmus gefunden wurde. Wird diese Matrix invertiert, so erhält man die Umkehrfunktion, die einen Punkt aus dem Urbildraum zurück in den Teilbildraum transformieren kann.
Diese Inverse wird nun auf jede Koordinate eines jeden gefundenen Polygons angewendet, so dass diese anschließend auf dem Display dargestellt werden können.
Die Matching-Einheit, in diesem Fall das Python-Skript, liefert das Resultat des Teilbild-Matchings an das PHP-Skript aus dem es aufgerufen wurde zurück.
Das Resultat besteht aus einem Statuscode des Matchingprozesses, sowie einem JSON-Objekt. Ist der Matching-Prozess ohne Fehler verlaufen, und der Status somit positiv, beinhaltet das JSON-Objekt die im Teilbild gefundenen, bereits in Bildschirmkoordinaten transformierten Polygone.
Andernfalls ist das JSON-Objekt leer und der Code beinhaltet den Grund, weshalb das Matching nicht funktioniert hat. Dies ist bei folgenden Umständen der Fall:

1. Aus dem Teilbild konnten keine Features extrahiert, und somit keine Deskriptoren extrahiert werden. Dies ist beispielsweise auf die Qualität des Teilbildes zurückzuführen.

2. Durch das Feature-Matching sind nicht genügend Matches entstanden, die zur Modellschätzung durch RANSAC benötigt werden.

3. Die Modellschätzung mittels RANSAC liefert bei Misserfolg eine leere Transformationsmatrix. Ist dies der Fall, kann keine Urbild-Region und somit auch kein Polygon lokalisiert werden.

4. Es konnten keine Polygone in der Urbildregion gefunden werden. Dies ist beispielsweise bei false-positive-Matches der Fall, da diese die Modellschätzung beeinflussen und zu einer falschen Transformationsmatrix führen.

Das verantwortliche PHP-Skript schickt die erhaltene Matching-Antwort als HTTP-response zurück an den mobilen Browser.

3.3.4 Darstellung der Matching-Antwort

Nach erfolgreicher Verarbeitung des aufgenommenen Teilbildes kann der Prozess abgeschlossen werden, indem das Ausgangsbild um die Polygoninformationen der gefundenen Polygone augmentiert wird (s. Tabelle A.4). Diese Aufgabe wird vom mobilen Webbrowser selbst übernommen, der die erhaltenen Polygoninformationen an die entsprechende Stelle des Teilbildes schreibt. Je nach Szenario wird hier eine unterschiedliche Darstellungsart gewählt, indem der durch den QR-Code eingelesene Modus ausgewertet wird. Befindet sich der Anwender vor einem Algorithmenfoto in der Universität, wird das Ausgangsteilbild um die Namen der Personen erweitert, wie in Abb. B.2b zu sehen. Durch viele verschiedene Smartphonetypen treten unterschiedlich große Displays auf. Des Weiteren ist es möglich, dass ein Student beispielsweise einen längeren Namen besitzt. Aus diesen Gründen wird die Anzahl angezeigter Polygoninformationen im Falle der Algorithmenfotos beschränkt, so dass das Display nicht überladen wird.
Sind zu viele Polygone vorhanden, wird dem Anwender ein entsprechender Hinweis zur erneuten Aufnahme gegeben.
Bei einem Museumsbesuch sind oftmals so viele Informationen über die Teilaspekte eines Bildes zu erwarten, dass diese nicht mehr auf dem Display dargestellt werden können. Hier wird sich deshalb für eine Sprachausgabe entschieden, durch die dem Anwender die Teilinformation präsentiert wird. Liefert die Matching-Einheit mehr als ein sichtbares Polygon im Bild, kann die Sprachausgabe nicht entscheiden, welches im Bild enthaltene Objekt das vom Anwender gewünschte ist. In diesem Fall werden die informationsreichen Regionen des aufgenommenen Bildes markiert, so dass der Anwender eine erneute, diesmal gezieltere Aufnahme der gewünschten Teilregion unternehmen kann.
Entscheidet sich der Anwender erneut ein Teilbild des selben Urbilds untersuchen zu wollen, gelangt dieser mittels eines anschließend eingebelendeten Buttons wieder zurück zum Videofeed der Kamera.

3.4 Entwurf eines Echtzeit-Verfahrens

Der entwickelte Kernprozess der Webanwendung (s. Abb. 3.3), sowie der Aufbau der Matching-Einheit (s. 3.3.3) lassen sich auf ein Echtzeit-Verfahren übertragen, das gemäß der Zielsetzung 1.3 in eine native Android-App integriert wird. Dem Anwender wird ermöglicht die smartphone-Kamera wie eine Lupe auf den gewünschten Teilaspekt des

Bildes zu halten, um die hinterlegten Informationen auf dem Display zu sehen. Da hier nur zu Anfang eine einmalige HTTP-Verbindung zum Server notwendig ist (s. Abb. 3.1), kann der allgemeine Anwendungszyklus lokal bei 30 Inputbildern pro Sekunde (FPS) n Mal durchlaufen werden ohne Kosten, außer der notwendigen Rechenleistung, zu verursachen.
Da das Teilbild-Matching hier auf der geräteeigenen CPU stattfindet und zusätzlich der verwendete Java-Wrapper für OpenCv (s. 3.1) die Ausführungen einzelner Funktionen der Bibliothek pro Aufruf verlangsamt, wird sich hier für asynchrone Verarbeitungstechniken entschieden, die ein Echtzeit-Verfahren gewährleisten können. In Abb. A.7 ist ein Sequenzdiagramm der Applikation beschrieben, das im Vergleich zur Webapplikation die Integration der Phasen des Kernprozesses veranschaulichen und im Folgenden genauer erklärt werden soll.

3.4.1 Selektion des Urbildes

Auch der nativen App muss zu Beginn mitgeteilt werden, vor welchem Bild sich der Anwender befindet und in welchem Modus die App gestartet wird, um die notwendigen Daten zu laden und die Darstellungsart des von der Matching-Einheit produzierten Outputs zu wählen.
Ruft der Anwender die Applikation auf, wird er zunächst auf einen Willkommens-Bildschirm geleitet. Abb. B.3 zeigt den Appzustand nach Aufruf durch den Anwender. Dabei müssen zunächst notwendige Berechtigungen erteilt, sowie anschließend der selbe QR-Code eingelesen werden, der bereits für die Webapplikation vor den Bildinstanzen angebracht ist. Eine QR-Scanner-Einheit wird direkt in die Applikation integriert, so dass nach dem Download dieser keine weitere externe App benötigt wird. Die Applikation liest den Code und lädt, sofern noch nicht geschehen, anschließend alle notwendigen Daten des Modus, bzw. des Ortes, vom Server auf das lokale Gerät, wie auch in Abb. B.4a gezeigt wird. Der Lösung liegt das bereits erwähnte Problem der unzureichenden Internetversorgung der unterschiedlichen Orte, sowie ein möglichst HTTP-unabhängiges Konzept zugrunde. Nachdem alle Daten heruntergeladen sind, ist die native Applikation in ihrer Funktionsweise unabhängig von weiteren Internetverbindungen.
Zu den auf dem Gerät gespeicherten Daten zählen die heruntergeladenen JSON-Informationen zu jedem Urbild, sowie die extrahierten Merkmale der Urbilder.
Die Merkmale eines Urbildes verändern sich nicht, sobald es in der Datenverwaltung vorhanden ist. Aus diesem Grund, und um nicht für jeden Teilbild-Matching-Zyklus die Urbild-Merkmale extrahieren zu müssen, werden diese zu Appstart einmal durch einen asynchronen Prozess extrahiert, und anstelle der Urbilder selbst auf dem lokalen Gerät abgespeichert. Die Merkmale lassen sich hier in kompakter Form durch Matrizen beschreiben und ebenfalls als JSON-Format im internen Speicher ablegen. Bei einem erneuten Start der App sind bereits alle Daten vorhanden, so dass die Datenakquisition nicht mehr notwendig ist. Der Anwender wird in diesem Fall direkt zur Hauptoberfläche weitergeleitet und kann mit dem Echtzeit-Teilbild-Matching beginnen.

3.4.2 Aufnahme der Teilbilder

Anders als bei der Konstruktion der Website wird hier kein plattformunabhängiger Standard für Hardwarezugriffe benötigt. Da die Anwendung nur für die Android-Plattform ausgelegt ist, wird auch nur mit bestimmten Hardwaretypen gerechnet. Die Aufnahme der Teilbilder geschieht hier in der Hauptoberfläche der Applikation, wie auch in Abb. B.4b zu sehen ist. Die Hauptanwendung kann somit auch hier wieder als Single-Page-Application gestaltet werden. Der Videofeed wird hier durch eine einfache Java-Repräsentation der nativen Kamera erreicht. Nachdem die notwendigen Funktionen OpenCvs geladen sind (s. Statische Initialisierung 3.1.2), kann die Kamera aktiviert werden und die Anwendung mit 30 Bildern pro Sekunde versorgen, die für eine flüssige Wahrnehmung der Inhalte notwendig sind. Durch die Hardwareeinschränkung ergeben sich außerdem Möglichkeiten, wie die genaue Einstellung der Fokussierungs-Modi, oder der Support des Heranzoomes, wie in Abb. 3.8 gezeigt.

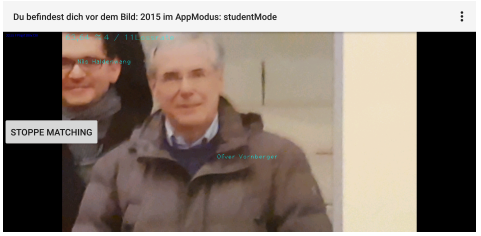


Abbildung 3.8: Android-Kamera Samsung Galaxy S8 mit maximalem Zoom

3.4.3 Matching der Teilbilder

Die Echtzeit-Matching-Einheit, die in der Android-App durch eine Java-Manager-Instanz realisiert ist, basiert auf dem gleichen Verfahren der Matching-Einheit der bereits vorgestellten Webapplikation.
Die aufgenommenen Bilder der Android-Kamera können abgefangen und verarbeitet werden. Wie bereits erwähnt führen Umstände, wie der Java-Wrapper um die OpenCv-Funktionalität, sowie der besonders kleine smartphone-eigene Prozessor zu Leistungsreduzierungen in den Berechnungen. Allein ein Teilbild-Matching-Zyklus würde den Hauptprozess während der Verarbeitungszeit der Anwendung so blockieren, dass keine flüssige Darstellung der Informationen mehr möglich ist. Die Entwicklung der Anwendung hat gezeigt, dass allein der OpenCv-Befehl zur Extraktion der Merkmale des ersten aufgenommenen Teilbildes in der Verarbeitung so lange dauert, dass statt 30 nur noch wenige Bilder pro Sekunde angezeigt werden können.
Um dennoch eine Echtzeit-Verarbeitung möglichst vieler Bilder pro Sekunde zu ermöglichen wird sich hier für eine asynchrone Lösung entschieden.

Feature-Matching durch Nebenläufigkeit

Die Simulation eines Echtzeit-Verhaltens wird durch ein Producer-Consumer-Design-Pattern erreicht und wird durch ein Sequenzdiagramm in Abb. A.8 veranschaulicht. n der 30 empfangenen Teilbilder können pro Sekunde vom Hauptprozess (Produzent) an die vorher initierte Matching-Einheit gesendet werden. Diese hält einen Datenspeicher,

eine Queue, der letzten k empfangenen Teilbilder auf die extern zugegriffen kann. Zur Simulation eines Echtzeit-Verhaltens werden von der Matching-Einheit eine beliebige Anzahl an Kindprozessen generiert, die sich um das eigentliche Teilbild-Matching kümmern können. Das Teilbild-Matching wird bereits in 3.3.3 beschrieben. Jeder erzeugte Kindprozess kann auf die zentrale Teilbild-Queue der Matching-Einheit zugreifen und fungiert somit im verwendeten Pattern als Konsument.
Hat der Kindprozess die sichtbaren Polygone aus dem Teilbild erkannt, sowie die zugehörigen Informationen im Urbild gefunden, wird das Ergebnis in einen ebenfalls in der Matching-Einheit enthaltenen Datenspeicher geschrieben und anschließend das nächste zu verarbeitende Bild von der Matching-Einheit angefragt.
Sobald es beim Teilbild-Matching eines Kindprozesses zu Fehlern kommt, wie sie bereits bei der Webapplikation auftreten können, wird kein Ergebnis produziert und der Kindprozess startet seinen Zyklus erneut. Durch ein weiteres Producer-Consumer-Pattern, in welchem der Hauptprozess als Konsument und die Matching-Einheit als Produzent fungieren, können die aktuellen Ergebnisse der verarbeiteten Teilbilder durch die Kindprozesse aus dem zuständigen Datenspeicher der Matching-Einheit abgefragt werden.
Werden die Anzahl der verarbeiteten Inputbilder pro Sekunde, die Anzahl der erzeugten Kindprozesse, sowie die Größe der Datenspeicher aufeinander abgestimmt, entsteht ein Echtzeit-Verhalten der Anwendung, das abgesehen von kleineren Mängeln eine hinreichend gute Annäherung an eine flüssige Verarbeitung der Teilbilder, sowie die gleichzeitige Darstellung Matching-Antwort liefert. Das verwendete Verfahren sorgt gleichzeitig dafür, dass eventuell durch das Teilbild-Matching entstandene Fehler ausgeglichen werden und die zuverlässige und durchgängige Darstellung des aktuellen Ergebnisses nur in kleinem Maße beeinflussen.

3.4.4 Darstellung der Matching-Antwort

Wie bereits aus dem letzten Unterabschnitt hervor geht, steht dem Hauptprozess der Android-Anwendung nahezu permanent ein durch die Matching-Einheit erzeugtes Ergebnis (Liste im Teilbild sichtbarer Polygone) zur Verfügung.
Dieses Ergebnis soll den möglichst aktuellen Inhalt der gerade aufgenommenen Region repräsentieren. Da die Verarbeitungszeit der Zyklen der Kindprozesse für jedes Teilbild unterschiedlich ist, kann dies nur bedingt erreicht werden. Bewegt der Anwender beispielsweise während des Matching-Prozesses sein Smartphone von Position A zu einer weiter entfernten Position B, liegen noch die Inputbilder der ursprünglichen Position im Datenspeicher der Matching-Einheit. Diese werden von den Kindprozessen verarbeitet, bevor die Teilbilder der aktuellen Position verarbeitet werden können. Dies lässt sich beispielsweise durch Integration des smartphone-internen Beschleunigungssensors lösen, der Bewegungen registriert, und den Zwischenspeicher der Matching-Einheit bei größeren Bewegungssprüngen entleert. Aufgrund des zeitlich begrenzten Rahmens dieser Arbeit wurden derartige Optimierungen jedoch vernachlässigt.
Verhält sich der Anwender bei Aufnahme des Bildes relativ statisch, werden hinreichend gute Ergebnisse produziert, die wieder je nach Modus der Anwendung unterschiedlich dargestellt werden können.
Da es sich um eine prototypische Anwendung handelt, wird hier unabhängig vom Modus vorerst nur die bekannte Augmentierung des Displays bevorzugt, wie auch in Abb. B.4c

dargestellt ist. Dies lässt sich jedoch durch entsprechenden Implementierungsaufwand auf eine Sprachausgabe erweitern.
Die Augmentierung des Bildes geschieht, indem für jedes der 30 Bilder pro Sekunde, das aktuelle Ergebnis aus der Matching-Einheit abgefragt wird. Ist ein Ergebnis vorhanden, wird mittels OpenCv die entsprechend gewünschte Polygoninformation jedes Ergebnispolygons in das Bild geschrieben, bevor es ausgegeben wird. Ist kein Ergebnis vorhanden, wird dies dem Anwender mitgeteilt. Befinden sich zu viele sichtbare Polygone im Ausgangsbild, werden diese nicht dargestellt, um eine Überladung des Bildschirmes zu vermeiden. Anstatt der Namen erscheint dann ein Text auf dem Bildschirm, der den Anwender zum Heranzoomen auffordert.

4 Implementierung

Dieser Abschnitt behandelt die plattformspezifischen Implementierungen und Integration des einfachen Teilbild-Matchings in die Website, sowie des Echtzeitverfahrens in die Android-Applikation. In Abschnitt 3 wird bereits der Aufbau, sowie die Funktionsweise der für die Applikationen notwendigen Komponenten vorgestellt. Aufgrund des begrenzten Rahmens der Arbeit wird im Folgenden nur auf die wesentlichen Teile der Implementierung eingegangen. Der vollständige Quellcode ist dem beiligenden Datenträger zu entnehmen (s. C).

4.1 Entwicklungsumgebung

Die Entwicklung der Website wird aufgrund des begrenzten zeitlichen Rahmens nur anhand des Chrome-Browsers durchgeführt. Getestet werden die Versionen 59, 61 und 62. Der WebRTC-unterstützte Safari 11 Browser ist erst kürzlich erschienen und noch im Entwicklungsstadium. Die Algorithmen der Webanwendung laufen auf der CPU, einer der Apache-Server der Universität (s.3.1.1).
Die Entwicklung der nativen Echtzeit-App und der Weboberfläche wird an einem Samsung Galaxy S8 mit einer 12-Megapixel-Kamera, einem Octa-core (2.3GHz Quad + 1.7GHz Quad) 64-Bit-Prozessor mit der Android-Version 7.0 durchgeführt. Über Endgeräte anderer Prozessorart, und somit einer anders kompilierten OpenCv-Bibliothek wird im Weiteren nicht eingegangen.
Die Anwendungen werden anhand der Bildinstanzen der Universität (s. 1.1) sowie der exemplarischen vom Felix-Nussbaum-Haus zur Verfügung gestellten Testinstanz, die in Abb. B.5 zu sehen ist, entwickelt.
Das Kunstwerk ist urheberrechtlich geschützt, kann jedoch hier nach Absprache mit dem Museum zu Testzwecken dem Werksverzeichnis [Feli] entnommen werden. Das Werk wird in keiner anderen Form außer in dieser Arbeit selbst veröffentlicht oder an Dritte weitergegeben. In Abb. B.6 sind die, für das Gemälde, zur Verfügung gestellten informationsreichen Regionen gegeben. Zu jeder dieser Regionen stellt das Museum außerdem eine geschichtliche Hintergrundinformation in Textform zur Verfügung, die wie die Algorithmenfotos der Universität im JSON-Format auf dem Server abgespeichert werden.

4.2 Implementierung eines webbasierten Matching-Verfahrens

Zunächst wird die Umsetzung der clientseitigen (mobiler Browser) Aufgaben der Website vorgestellt, bevor auf das dazugehörige serverseitige Teilbild-Matching eingegangen wird.

4.2.1 Implementierung des Clients

Der Client übernimmt, wie in 3.3 erläutert, die Aufgaben der Teilbildaufnahme, den Transfer des Bildes zum Server, sowie die Darstellung des erhaltenen Ergebnisses.
Die Weboberfläche selbst besteht aus einer einzigen *index.php*, die sich aus einem *Header*, einem *footer* und einer Hauptkomponente zusammensetzt. Die PHP-Funktion `renderTemplateWithContent(\$contentFile, \$variables = array())` macht den Hauptinhalt der Seite austauschbar und ermöglicht, dass je nach Modus und Urbild mit dem die Website aufgerufen wird, entsprechende Hintergrundinformation (Autor, Erscheinungsjahr, ..) des Urbildes präsentiert wird.

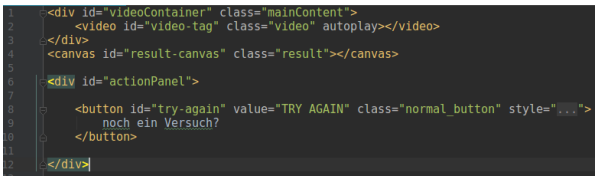


Abbildung 4.1: Hauptinhalt der Weboberfläche

Der Hauptinhalt selbst besteht, wie in Abb. 4.1 zu sehen ist, im Wesentlichen aus dem HTML5-Video-Element, sowie einer Zeichenfläche *result-canvas*, die vorerst mit dem CSS-Attribut `display:none` ausgestattet ist und als Platzhalter für das aufgenommene Teilbild dient. Das Attribut *autoplay* des Video-Elements sorgt dafür, dass das Video in WebRTC-unterstützten Browsern automatisch startet, sobald die Kameraberechtigung erteilt wird. Des Weiteren ist im Hauptinhalt ein ebenfalls vorerst unsichtbarer Button integriert, der dem Nutzer nach Erhalt eines Ergebnisses zur erneuten Aufnahme verhilft.
Die *index.php* integriert außerdem die notwendigen Javascript-Dateien. Darunter befinden sich z.B. die fremde Datei *adapter-latest.js*[1], die alle aktuellen Funktionen bzgl. WebRTC bereit stellt, die fremde Datei *responsivevoice.js*[2], die für die Audiowiedergabe des späteren Museumszenarios verwendet wird, sowie ein fremdes Overlay, das dem Anwender den Prozess der Verarbeitung anzeigt[3].
Außerdem werden die eigenen Dateien *main.js*, die alle Funktionen zum Setup der Weboberfläche und WebRTC, sowie der Aufnahme des Teilbildes liefert und die eigene Datei *style.css*, die das Design der einzelnen UI-Komponenten definiert, verwendet.
Ist der `DOM-Tree` der aktuellen *index.php* geladen, wird den `HTML`-Komponenten eine Funktion zugewiesen. .

```
1    videoContainer.addEventListener('click', takePhoto);
2    document.getElementById('try-again').addEventListener("click",
         restoreStream);
3    startStream();
```

Listing 4.1: Setup der Komponenten in *main.js*

Listing 4.1 zeigt die wichtigen clientseitigen Funktionen. Dabei spielen die Funktionen `takePhoto()` und `restoreStream()` eine Rolle sobald die Methode `startStream()` mittels `WebRTC` für einen Videofeed sorgt. .

[1] Adapter-latest.js https://webrtc.github.io/adapter/adapter-latest.js (Abrufdatum: 26.10.17)
[2] Sprachausgabe mit ResponsiveVoice https://responsivevoice.org/ (Abrufdatum: 26.10.17)
[3] Overlay https://cdn.jsdelivr.net/npm/gasparesganga-jquery-loading-overlay@1.5.3 (Abrufdatum: 26.10.17)

```
1     //older browsers dont support mediaDevices or enumerateDevices
2     if (!navigator.mediaDevices || !navigator.mediaDevices.
          enumerateDevices) {
3         alert("Deine aktuelle Browserversion wird leider zurzeit nicht
              unterstuetzt.");
4         return;
5     }
```

Listing 4.2: Nicht unterstützte Browser-Versionen durch WebRTC

Dabei werden noch nicht alle Browser-Versionen durch den Standard versorgt, wie Listing 4.2 zeigt. Wird die Methode `enumerateDevices()` unterstützt, kann mit dieser die Rear-Kamera des Smartphones selektiert werden. Standardmäßig wird von `WebRTC` die Front-Kamera verwendet. .

```
1   var constraints = {audio: false,
2                      video: {deviceId: {exact: cameraDevice.deviceId}
3                 },optional:[
4                     {minWidth: 320},
5                     {minWidth: 640},
6                     {minWidth: 1024},
7                     {minWidth: 1280},
8                     {minWidth: 2560}]};
```

Listing 4.3: Constraints des Videofeeds

Die gefundene Rear-Kamera-Id kann dem `Constraints`-Array hinzugefügt werden, welches die Art des Videofeeds bestimmt (s. Listing 4.3). Die `Constraints` beinhalten des Weiteren die optionalen Felder `minWidth`. Durch einen Bug seitens `WebRTC` wird nur so gewährleistet, dass die größtmögliche unterstützte Auflösung für das Video gewählt wird. Das Array kann anschließend der `WebRTC`-Funktion `getUserMedia(constraints)` übergeben werden, welche die Anbindung der Smartphonekamera an den Videofeed, sowie die Abfrage der Kameraberechtigung ermöglicht und den Stream anschließend startet. Dem Container des Videos wird, wie in Listing 4.1 zu sehen, ein Listener hinzugefügt, so dass der Anwender durch Klicken des Videos die Methode `takePhoto()` aktiviert.

.

```
1   if (localMediaStream) {
2           \$('#videoContainer').hide();
3           \$('#result-canvas').show();
4           result_context.drawImage(video, 0, 0, video.videoWidth, video.
                videoHeight);
5           data = result_canvas.toDataURL('image/png');
6           \$.LoadingOverlay("show");
7           sendResultRequest(data);
8       }
```

Listing 4.4: Teilbildaufnahme mit HTML5-Video

Listing 4.4 zeigt den Ablauf Teilbildaufnahme. Das Video wird aus- und die Ergebniszeichenfläche eingeblendet, in die mit der Methode `drawImage()` eine Momentaufnahme des Streams gezeichnet wird. Diese kann nach Erhalt des Ergebnisses um die gefundenen hinterlegten Informationen erweitert werden. Die Größe der gezeichneten Aufnahme bzw. des aufgenommenen Teilbildes ergibt sich aus der Größe des Videos, die je nach mobilem

Gerät variiert. Wird die Website von einem Gerät mit größerem Display aufgerufen, kann die serverseitige Verarbeitungszeit gering gehalten werden, indem das Bild mit Javascript herunterskaliert und die anschließend erhaltenen Ergebnispolygone um den selben Faktor hochskaliert werden.
In 4.4 ist zusätzlich zu sehen, wie die Momentaufnahme durch die Methode `toDataURL()` in ein übertragbares `Base64`-PNG-Format umgewandelt wird. Der String ist je nach Menge der zu kodierenden Daten entsprechend groß. Die kodierten Daten werden anschließend durch die Methode `sendResultRequest()` zur Verarbeitung an den Server weitergeleitet.

```
1 type: "POST",
2         url: "api/processing.php",
3         data: {mode: mode,
4             src: sourceId,
5             image: data}
```

Listing 4.5: Transfer der Daten durch JQuery

Wie in Listing 4.5 zu sehen ist, wird mittels `JQuery` eine HTTP-Anfrage an das verantwortliche Skript des Servers gestartet. Übertragen werden der `Base64`-String, der das Teilbild beschreibt sowie der Modus und die Urbild-Id, damit die Matching-Einheit die entsprechend hinterlegten Daten finden kann.
Hat die Anfrage Erfolg, wird je nach Modus die Information eines jeden Ergebnis-Polygons planar auf der Zeichenfläche ergänzt, die noch das ursprünglich abgesendete Teilbild enthält. Für den Fall, dass der Modus einem Museumsbesuch entspricht wird zunächst getestet, ob genau ein Polygon enthalten ist. In diesem Fall wird die hinterlegte Information als Sprachausgabe, gemäß eines Audioguides, präsentiert. Ist mehr als ein Polygon enthalten werden die Regionen des Teilbildes die Information enthalten markiert und der Anwender zur erneuten, diesmal genaueren Aufnahme verwiesen.

4.2.2 Serverseitiges Teilbild-Matching

Das Skript `processing.php` auf dem Server kann die Query-Parameter des `JQuery`-Requests (s. Listing 4.5) auslesen und mit diesen Parametern und dem PHP-Befehl `exec()` die Matching-Einheit bzw. das Python-Skript aktivieren.

```
1 $command = "python detector.py $mode $sourceId $fileName";
2 exec($command, $script_output, $return_code);
```

Listing 4.6: Aufruf der Python-Matching-Einheit aus einem PHP-Skript

Der Base64-String ist zu lang um ihn als Kommandozeilenparameter zu übergeben, weshalb das Bild dekodiert und in einem temporären Ordner unter fortlaufender Nummer während des Matchings zwischengespeichert wird. Der relative Pfad kann der Matching-Einheit *detector.py* als Parameter übergeben werden, wie in Listing 4.6 zu sehen ist. Des Weiteren werden die Parameter *mode*, der den Ort bestimmt, sowie die Id der Bildinstanz vor der der Anwender steht, übermittelt. Die Antwort der Matching-Einheit wird nach

der Verarbeitung in die Variable *script_output* geschrieben. Der *return_code* ist nach dem Teilbild-Matching auf 1 gesetzt, falls es interne Probleme bei der Verarbeitung wie z.B. unvorhergesehene serverseitige Probleme o.ä. gibt, und ansonsten auf 0. Im weiteren Verlauf wird das Urbild auch als *Source*, und das Teilbild als *Scene* bezeichnet.

Die Matching-Einheit detector.py

Das für das Matching zuständige Skript liest die übergebenen Kommandozeilenparameter aus und lädt die zugehörigen Urbild-Daten, wie in Listing 4.7 zu sehen ist.

```
1 # create internal source object
2 src_obj = data_structures.SourceObject(sys.argv[1], sys.argv[2])
```

Listing 4.7: Python Urbild-Repräsentation

Dabei stellt das Untermodul *SourceObject* eine Repräsentation des Urbildes dar und hält die zugehörige JSON-Datei, die unter Anderem aus einer Liste von hinterlegten (in das Urbild eingezeichneten) Polygonen besteht, sowie den Pfad zum Urbild selbst.

```
1 source_image_gray = cv2.imread(src_obj.image_path,
     CV_LOAD_IMAGE_GRAYSCALE)
2 source_image_gray = cv2.resize(source_image_gray, (760, 620))
3 scene_image_gray = cv2.imread(tmp_image_path, CV_LOAD_IMAGE_GRAYSCALE)
```

Listing 4.8: Laden des Urbildes als Grauwertmatrix

Durch die Funktion `imread()` werden das Ur- und das Teilbild als Grauwertmatrizen in das Skript geladen (s. Listing 4.8). Dabei wird das Urbild auf 760x620 Pixel reduziert, sowie die Koordinaten der hinterlegten Polygone anschließend um den Skalierungsfaktor angepasst. Die niedrigere Auflösung bedeutet eine geringere Anzahl detektierter Merkmale und somit auch eine kürzere Matchingdauer.
Mit Hilfe einer SIFT-Instanz werden die Merkmale der beiden Graubilder detektiert und die entsprechenden Deskriptoren berechnet, siehe Listing 4.9.

```
1 sift_instance = cv2.xfeatures2d.SIFT_create()
2 src_features, src_descriptors = sift_instance.detectAndCompute(
     source_image_gray, None)
3 scene_features, scene_descriptors = sift_instance.detectAndCompute(
     scene_image_gray, None)
```

Listing 4.9: Initialisierung der SIFT-Instanz und Extraktion der Merkmale

Der dafür zuständigen Methode `detectAndCompute()` kann dazu eine zusätzliche Bitmaske mitgegeben werden. Durch Verundung der Bitmaske mit der Bildmarix, kann eine ROI selektiert werden, so dass ausschließlich in dieser Region nach Merkmalen gesucht wird. Hier wird jedoch darauf verzichtet und in der ganzen aufgenommenen Teilbildregion nach Inhalten gesucht.

```
matches = flann.knnMatch(scene_descriptors, src_descriptors, k=2)
better_matches = []
for m, n in matches:
    if m.distance < 0.7*n.distance:
        better_matches.append(m)
```

Listing 4.10: Feature-Matching mit KNN-Match und anschließendem Ratio-Test nach Lowe [Lowe 04]

Die Deskriptoren-Sets werden mit einer Euklid-Metrik in einem 2-Nächste-Nachbarn-Verfahren des FLANN-Moduls miteinander verglichen, wie Listing 4.10 zeigt. Der Ratio-Test nach Lowe filtert die besten Matches aus der Menge heraus, so dass mittels RANSAC eine möglichst genaue Schätzung des Modells vorgenommen werden kann (s. Listing 4.11).

.

```
    H = cv2.findHomography(scene_points, target_points, cv2.RANSAC, 2
       .0,2000)
    if H.size > 0:
        target_corners = cv2.perspectiveTransform(scene_corners, H)
```

Listing 4.11: Schätzung des Modells und Transformation des Teilbildes in den Urbildraum

Listing 4.11 zeigt, wie die OpenCv-Methode `findHomography()` das Modell der gefunden Matches aus Teil- und Urbild schätzt. Dabei hat sich durch empirisches Vorgehen ein Schwellwert von 2.0 bei 2000 Iterationen ergeben. Durch den Schwellwert entscheidet RANSAC, ob das aktuell untersuchte Merkmal innerhalb oder außerhalb der Region liegt und bestimmt somit den zulässigen Fehler zwischen geschätztem und tatsächlichen Modell. Mit der resultierenden Matrix H wird anschließend aus der Liste der *clockwise* geordneten Teilbildecken die Liste der Ecken im Urbildraum gefunden, indem die Methode `perspectiveTransform()` auf die Teilbildecken angewendet wird.

.

```
    inverse_mat = np.linalg.inv(H)
    for poly in src_obj.objects:
        if poly.is_visible(target_corners):
            transformed_polygon = poly
            poly.centroid = cv2.perspectiveTransform(poly.centroid,
               inverse_mat)
            transformed_corners = []
            for corner in poly.corners:
                transformed_corner = cv2.perspectiveTransform(corner,
                   inverse_mat)
                transformed_corners.append(transformed_corner)
            poly.corners = transformed_corners
            visible_polygons.append(transformed_polygon)
```

Listing 4.12: Lokalisierung der Polygone und Transformation der Daten

Listing 4.12 zeigt wie die in der Zielregion enthaltenen Polygone gefunden werden. Dabei wird durch die Liste der hinterlegten Polygone iteriert und jedes Polygon einem Lokalitätstest unterzogen. Dieser Test untersucht, ob das Zentrum des hinterlegten

Polygons innerhalb der gefundenen Region liegt. Wenn ja, ist es ein im Teilbild sichtbares Polygon. Befindet sich ein Polygon im sichtbaren Bereich können dessen Daten, die noch auf dem Urbildraum basieren, durch die invertierte Transformationsmatrix H zurück in den Teilbildraum transformiert werden. Das transformierte Polygon wird der Ergebnisliste hinzugefügt und kann durch JSON über PHP zur Darstellung an den Browser gesendet werden. Die Ausführungszeit eines typischen Anwendungszyklus ist Tabelle A.5 zu entnehmen.

4.3 Implementierung eines nativen Echtzeit-Verfahrens

Das Grundprinzip der Matching-Einheit der Webapplikation lässt sich, wie in 3.4 beschrieben, auf eine lokal arbeitende Echtzeit-Applikation übertragen. Ein aufgrund der Übersichtlichkeit reduziertes Klassendiagramm der dazu entwickelten Android-App ist Abb. A.4 zu entnehmen. Die Grundfunktionalität der Anwendung wird bereits in 3.4 beschrieben. Im Folgenden wird auf die wichtigsten Strukturen des Klassendiagramms aus Abb. A.4 eingegangen, aus denen sich die Funktionalität der Echtzeit-Teilbild-Matching-App ergibt.

4.3.1 LaunchActivity

Die `LaunchActivity` ist die erste, die dem Anwender nach dem Start präsentiert wird. Hier werden alle notwendigen Berechtigungen erfragt, sowie der QR-Code des Urbildes vom Anwender eingelesen (s. Abb. B.4).
Mit Hilfe des integrierbaren `zxing`[4]-QR-Code-Scanners wird der Anwender durch das Einlesen des Codes zur `LoadingActivity` weitergeleitet.

4.3.2 LoadingActivity

Bevor der Anwender zur Hauptoberfläche `OpenCvActivity` gelangt, wird ihm durch diese Activity signalisiert, dass alle notwendigen Daten, falls noch nicht vorhanden, heruntergeladen werden (s. Abb. B.4a).
Hier wird die, nach dem Singleton-Pattern konstruierte, zentrale Klasse `ResourceManager` initiiert, dem bei der Instanziierung die Daten des QR-Codes übergeben werden, um die notwendigen Daten herunterzuladen. Die Klasse implementiert zusätzlich ein Interface, um dem Anwender den Downloadstatus mitzuteilen, sowie ihn nach beendetem Download zur Hauptoberfläche weiterzuleiten (s. Abb. A.4 oder das Sequenzdiagramm A.6).

4.3.3 ResourceManager

Der Ressourcenmanager ist während des ganzen Anwendungszyklus für die interne Datenverwaltung zuständig.

[4]ZXing QR- and Barcode-Scanner https://github.com/zxing/zxing (Abrufdatum: 26.10.17)

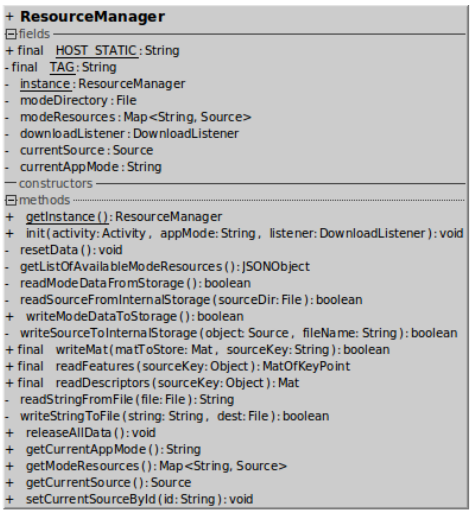


Abbildung 4.2: Der ResourceManager als Singleton

Wie Abb. 4.2 zeigt, liefert die Instanz verschiedene Methoden zum Lesen und Schreiben von Daten, insbesondere bereits berechneter Merkmalsmatrizen und der Polygoninformationen der Urbilder. Wird die App das erste Mal gestartet sind zur Instanziierung des Manager noch keine Daten auf dem Gerät vorhanden. Deshalb wird mittels der Methode `getListOfAvailableResources()` eine HTTP-Anfrage, über alle verfügbaren Ressourcen zum aktuellen Modus, an das PHP-Skript *get_mode_data.php* des Servers geschickt. Die Antwort besteht aus einer Liste von URLs der vorhandenen Ressourcen. Diese können vom Manager mittels eines asynchronen Prozesses (Android-`AsyncTask`) heruntergeladen und in interne Objekte umgewandelt werden, die in einer Liste *modeResources* gespeichert werden.

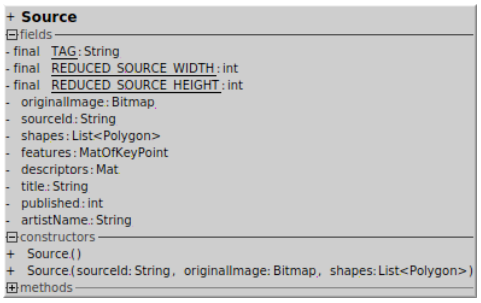


Abbildung 4.3: Java-Repräsentation des Urbildes

Zu jedem heruntergeladenen .PNG-Urbild wird die entsprechende JSON-Datei geladen und ein *Source*-Objekt erzeugt, siehe Abb. 4.3. Das Urbild wird beim Herunterladen auf 760x620 Pixel reduziert und die Polygondaten entsprechend angepasst. Eine zu hohe Auflösung sorgt, für viele extrahierte Merkmale und Deskriptoren und somit für eine erhöhte Ausführungszeit pro Matchingzyklus, die die flüssige Darstellung der Ergebnisse verhindert. Für die später extrahierten Urbild-Merkmale und deren Deskriptoren gibt es die OpenCv-`Mat`-Objekte `features` und `descriptors`. Die `Source`-Klasse beinhaltet außerdem eine Liste der im Urbild hinterlegten Polygone. Diese werden durch eine eigene Klasse repräsentiert, wie Abb. 4.4 zeigt. Neben den eigentlichen Ecken, die die Kontur

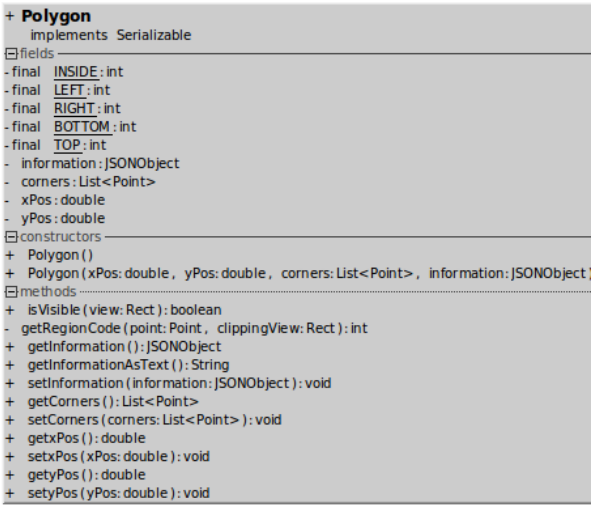


Abbildung 4.4: Java-Repräsentation eines Teilobjekts

des Polygons bestimmen und den dazu hinterlegten Textinformationen, gibt es eine Reihe von Methoden, die diese für andere Klassen zugänglich machen.
Außerdem beinhaltet sie die Methode `isVisible()`, die bei einer gegebenen Liste von anderen Ecken überprüft, ob sich das eigene Zentrum innerhalb der zu überprüfenden Region (durch RANSAC bestimmt) befindet. Des Weiteren gibt es Felder für Region-Codes, falls sich im Weiteren doch dazu entschieden wird, die Sichtbarkeit eines Polygons dadurch zu definieren, dass jede seiner Kanten in der zu überprüfenden Region liegt.
Die Repräsentation durch eigene Klassen ermöglicht zusätzlich, dass die Art der Information, die als Input der Applikation dienen, austauschbar und leicht wartbar wird.

4.3.4 OpenCvActivity

Diese Activity ist für den Kernprozess des Echtzeit-Matchings zuständig. Hier wird die zuvor kompilierte OpenCv-Bibliothek geladen, um die darin enthaltenen Datenstrukturen und Funktionen zu verwenden. Der allgemeine Lebenszyklus einer Activity, wird durch den Einsatz von OpenCv erweitert, wie Abb. 4.5 zeigt.

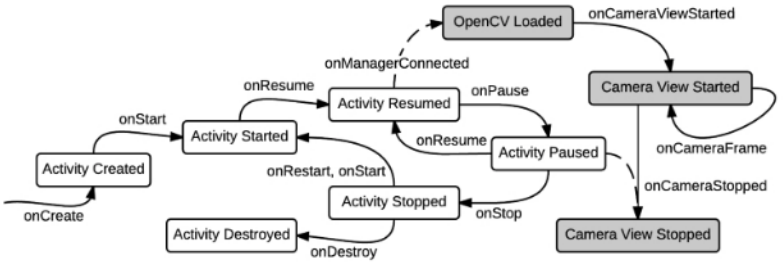


Abbildung 4.5: Lebenszyklus einer OpenCv-Activity

Die Klasse, inklusive aller Methoden und Funktionen ist in Abb. 4.6 zu sehen. Sobald die Activity in den Vordergrund tritt (`onResume()`-Methode) wird die OpenCv-Bibliothek

geladen. Da die Methode ebenfalls aktiviert wird, falls der Anwender vom *Portrait*- in

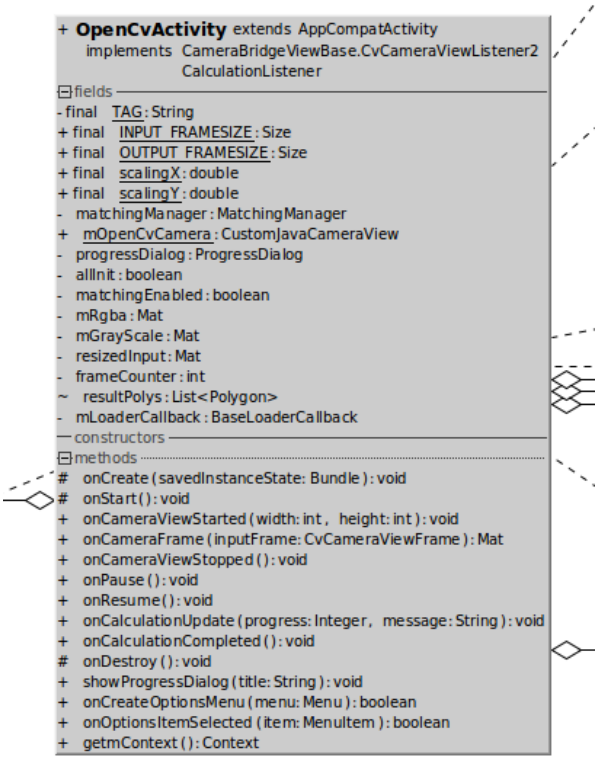


Abbildung 4.6: Die OpenCv-Activity-Klasse
(Quelle: [Hows 15, S.42]

den *Landscape*-Modus oder umgekehrt wechselt, wird sich für die App auf die *Landscape*-Ansicht festgelegt, um weitere Probleme zu vermeiden.
Nach dem Laden der Bibliothek wird der lokale `BaseLoaderCallback` mit der Methode `onManagerConnected()` aktiviert (s. Abb. 4.5). Alle weiteren Klassen, die auf OpenCv-Strukturen basieren, können erst ab dieser Stelle initialisiert werden. Wurde OpenCv korrekt geladen, wird an dieser Stelle der MatchingManager, sowie die lokale Kamera `CustomCameraView` gestartet, die in der `onCreate()`-Methode erstellt wurde. Diese erbt von der Klasse `JavaCameraView` und ermöglicht direkten Zugriff auf die native Kamera, sowie auf Einstellungen z.B. bzgl. Fokussierung und Kameralicht. Des Weiteren wird die Kamera um einen *Pinch-Zoom* erweitert, so dass der Anwender mit seinen Fingern in das gewünschte Teilbild herein- oder herausskalieren kann und um einen manuellen Autofokus, der durch das einmalige Tippen auf den Bildschirm aktiviert wird. Durch das Implementieren des Interface `CvCameraViewListener` bekommt die `OpenCvActivity` die Methoden `onCameraViewStarted()`, `onCameraFrame()`, sowie `onCameraViewStopped()` vererbt (s. Abb. 4.6). Der Ablauf der Methoden ist Abb. 4.5 zu entnehmen.

Start der Kamera:
In der Methode `onCameraViewStarted()` werden die klasseninternen `Mat`-Objekte initialisiert. Die Erzeugung dieser Matrizen ist recht teuer bzgl. Verarbeitungszeit und lokalem Speicher, so dass die mehrfache Erzeugung pro Sekunde den Grafikspeicher des Smartphones schnell zum Überlaufen bringen kann. Aus diesem Grund werden zwei

Klassenvariablen `mGrayScale` (Graubildmatrix) und `mRgba` (Farbmatrix) erstellt, die zu Kamerastart einmal erzeugt und pro empfangenen Bild immer wieder neu belegt werden.

Aufnahmezyklus:
Die Methode `onCameraFrame()` wird 30 mal pro Sekunde mit einem `CvCameraViewFrame` aufgerufen, die als Ausgabe das Bild liefert, dass dem Anwender auf dem Display angezeigt wird. Ein Auszug der Methode ist in Listing 4.13 zu sehen. Aus diesem können direkt die entsprechenden Bildmatrizen extrahiert werden.

```
1   mGrayScale = inputFrame.gray();
2        mRgba = inputFrame.rgba();
3        Imgproc.resize(mGrayScale, resizedInput, INPUT_FRAMESIZE);
4
5          if (allInit && matchingEnabled) {
6
7                if (frameCounter % 15 == 0) {
8
9                    matchingManager.pushFrame(resizedInput);
10
11               }
12               if (frameCounter % 150 == 0) {
13               System.gc();
14               System.runFinalization();
15               }
16               resultPolys = matchingManager.getCurrentResult();
```

Listing 4.13: Auszug der Methode onCameraFrame()

Dabei ist das `inputFrame` mit der Auflösung 1280x960 Pixel gegeben. Während eine Farbversion des Bildes für die Ausgabe an den User gespeichert wird, wird eine Graubildversion für die Feature-Matching-Berechnungen verwendet. Dieses wird, um Rechenzeit einzusparen, mit OpenCv auf ein 360x360-Pixelbild herunterskaliert. Der Faktor wird gespeichert, um die aus der Matching-Einheit gelieferten Daten anschließend um den selben Faktor hoch zu skalieren, damit die Textinformationen auch an der passenden Stelle ins erweitere Bild gezeichnet werden.
Hat der Anwender das Matching zugelassen, wird jedes 15. empfangene Bild (zwei Mal pro Sekunde) zur Verarbeitung in den Zwischenspeicher des Matchingmanagers geschickt. Alle fünf Sekunden wird außerdem manuell der Java-Garbage-Collector[5] aktiviert, da die Zerstörung der nicht mehr benötigten C++-OpenCv-Objekte nur unzuverlässig von Java erkannt wird und dies den internen Speicher überlädt. Bei jedem der 30 Bilder wird überprüft, ob ein aktuelles Ergebnis im Ergebnisspeicher des Matchingmanagers vorhanden ist.

```
1   for (Polygon poly : resultPolys) {
2
3       Imgproc.putText(mRgba, poly.getInformationAsText(), new Point(
           poly.getxPos() * scalingX, poly.getyPos()*scalingY),
4          Core.FONT_HERSHEY_PLAIN, 1.5, new Scalar(0, 255, 255));
```

Listing 4.14: Auszug der Methode onCameraFrame-Augmentierung des Ausgabebildes

[5]Garbage Collector http://www.oracle.com/webfolder/technetwork/tutorials/obe/java/gc01/index.html (Abrufdatum: 26.10.17)

Dieses kann anschließend, je nach Anwendungsfall durch OpenCv auf das Ausgabebild geschrieben werden, bevor es durch ein `return` auf dem Display erscheint, wie in Listing 4.14 zu sehen ist.

Zerstörung der Kamera:
Wird die Kamera durch das Verlassen der Activity, oder das Beenden der App, zerstört wird die Methode `onCameraViewStopped()` aufgerufen und alle noch aktiven OpenCv-Mat-Objekte ebenfalls zerstört. Der Java-Garbage-Collector arbeitet hier nicht zuverlässig, so dass der allokierte Speicher für die Matrizen mittels `mat.release()` manuell freigegeben wird.

4.3.5 MatchingManager

Die Klasse `MatchingManager` repräsentiert die Echtzeit-Matching-Einheit, dessen allgemeine Funktionsweise bereits in 3.4 erklärt wird. Wie das Sequenzdiagramm in A.9 zeigt, wird der Manager in der anonymen Klasse für den `BaseLoaderCallback`, innerhalb der `OpenCvActivity`, instanziiert. Abb 4.7 zeigt die Felder, sowie die Funktionen des Mana-

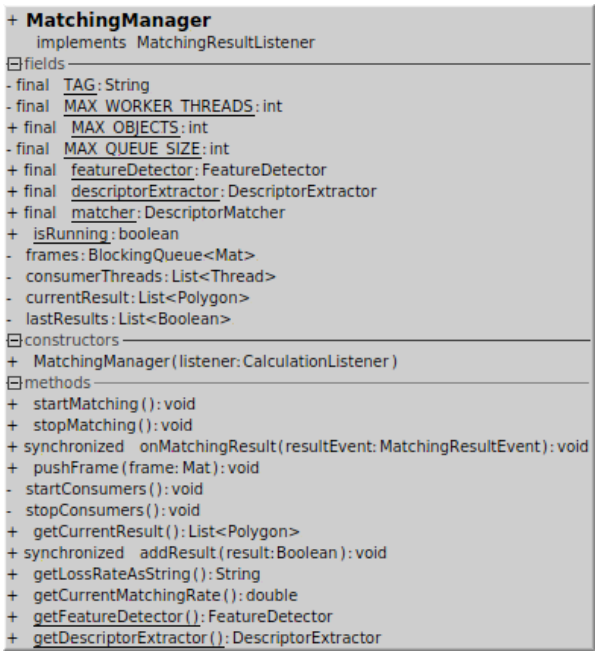


Abbildung 4.7: Java-Repräsentation der Echtzeit-Matching-Einheit

gers. Die Klasse stellt unter Anderem die OpenCv-Instanzen wie den `FeatureDetector` zur Detektion von Merkmalen, `DescriptorExtractor` zur Berechnung der Merkmalsdeskriptoren, sowie einen `DescriptorMatcher` zum Feature-Matching bereit, die von allen Matching-Prozessen verwendet werden können. Der Manager hält außerdem eine Queue vom Typ `BlockingQueue` *frames*, für die n letzten zu verarbeitenden Bilder, sowie eine Liste der aktuell für den Anwender sichtbaren Polygone.

```
1 public static final FeatureDetector featureDetector =
    FeatureDetector.create(FeatureDetector.SIFT);
2 public static final DescriptorExtractor descriptorExtractor =
    DescriptorExtractor.create(DescriptorExtractor.SIFT);
3 public static final DescriptorMatcher matcher =
    DescriptorMatcher.create(DescriptorMatcher.FLANNBASED);
```

Listing 4.15: Festlegung der Verfahren für Detektion, Deskription und Matching

Dabei wird, wie bereits im Konzept 3 erläutert, ein SIFT- `FeatureDetector`, sowie ein SIFT-`DescriptorExtractor` verwendet, wie Listing 4.15 zeigt. Im Konstruktur des Managers werden die Vorbereitungen für das Echtzeit-Matching getroffen, indem ein `PrepareMatchingTask` gestartet wird (s. Listing 4.16).

```
1 new PrepareMatchingTask(listener).execute();
```

Listing 4.16: Auszug PrepareMatchingTask: Extraktion der Urbildmerkmale

PrepareMatchingTask:

Durch den asynchronen Prozess werden die Deskriptoren des aktuellen `Source`-Objekts ermittelt. Die dafür angelegten `Mat`-Objekte sind bis zu diesem Zeitpunkt noch nicht ausgefüllt. Zunächst wird überprüft, ob zur aktuell betrachteten Bildinstanz bereits Deskriptoren im internen Speicher abgelegt wurden und speichert sie in diesem Fall im `Source-Objekt`.

```
1  Mat grayscaleSource = new Mat();
2  Bitmap bmp32 = currentSource.getOriginalImage().copy(
     Bitmap.Config.ARGB_8888, true);
3  Utils.bitmapToMat(bmp32, grayscaleSource);
4  Imgproc.cvtColor(grayscaleSource, grayscaleSource,
     Imgproc.COLOR_RGBA2GRAY);
5  ...
6  MatchingManager.getFeatureDetector().detect(grayscaleSource,
     currentSource.getFeatures());
7  ...
8  MatchingManager.getDescriptorExtractor().compute(grayscaleSource,
     currentSource.getFeatures(), currentSource.getDescriptors());
9  ...
10 ResourceManager.getInstance().writeMat(currentSource.getDescriptors(),
     sourceName)
```

Listing 4.17: Auszug PrepareMatchingTask: Extraktion der Urbildmerkmale

Sind noch keine Daten vorhanden werden die Merkmale detektiert und in Deskriptoren überführt, wie in Listing 4.17 gezeigt wird. Dazu konvertiert der Prozess das Urbild der aktuellen Bildinstanz mit Hilfe von OpenCv vorher von einer `Bitmap` in eine Grauwertmatrix. Anschließend können diese mit dem `ResourceManager` im internen Speicher abgelegt werden.
Der Prozessstatus wird durch den `CalculationListener` an die aktuelle Activity (OpenCvActivity) gegeben, die die Methoden des ListenerInterfaces implementiert. Dadurch wird dem Anwender der Berechnungsstatus angezeigt und nach Beendigung das Matching freigegeben.

MatchingTask:

Wie aus dem Sequenzdiagramm A.9 und Abb.4.7 hervorgeht wird das Matching des Managers durch ein äußeres Event (hier, ein Button als Usereingabe) gestartet, indem die Manager-Methode `startMatching()` ausgeführt wird. Sobald dies der Fall ist, erzeugt der MatchingManager durch die Methode `startConsumers()` eine Anzahl `MAX_WORKER_THREADS` an Arbeiterprozessen.
Die Methode `onCameraFrame()` der `OpenCvActivity` arbeitet auf dem Hauptprozess und sendet permanent aufgenommene Teilbilder an die `BlockingQueue` *frames* des MatchingManagers und erfüllt somit die Rolle als Produzent.
Die Arbeiterprozesse vom Typ `Thread` werden mit einem `MatchingTask`, der das Interface `Runnable` implementiert, ausgeführt. Die Prozesse erkundigen sich permanent beim Manager, ob es in *frames* Teilbilder zur Verarbeitung gibt und erfüllen somit die Rolle des Konsumenten. Sind Teilbilder zur Verarbeitung vorhanden, wird an diesem das Feature-Matching mit den Urbild-Deskriptoren durchgeführt. Andernfalls wartet der Prozess, bis vom Produzenten neue Teilbilder produziert wurden.
Die Klassenübersicht eines `MatchingTask` ist in Abb. 4.8 zu sehen.

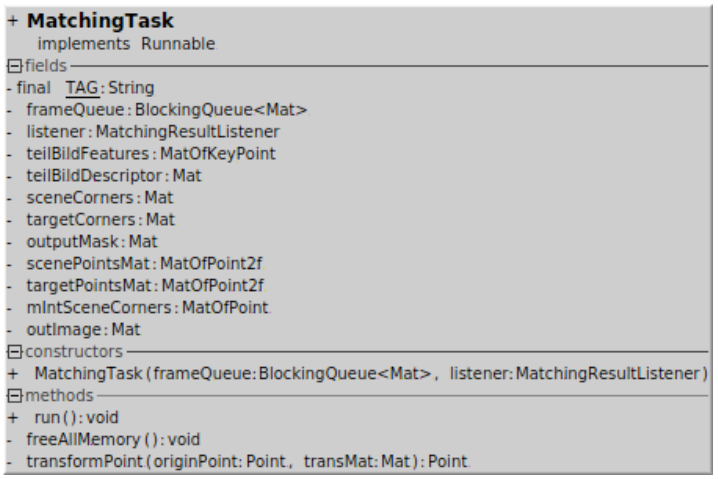


Abbildung 4.8: Java-Repräsentation MatchingTask

Das eigentliche Matching-Verfahren ist bereits hinreichend erklärt, sowie in einer Python-Variante als Matching-Skript in die Webapplikation integriert (s. Konzept 3 oder 4.2.2). Die wesentlichen Unterschiede ergeben sich aus den speziellen Java-OpenCv-Datentypen, sowie dem für die C++-Funktionen verwendetem Wrapper. Der `MatchingTask` läuft in einer Endlosschleife, bis das Matching extern, oder durch Appabbruch beendet wird. Für jeden Schleifendurchlauf wird das nächste Bild aus dem Manager angefragt und verarbeitet. Das Extrahieren der Merkmale funktioniert dabei wie bereits im `PrepareMatchingTask` bzgl. des Urbildes.

```
1
2 MatchingManager.matcher.knnMatch(teilBildDescriptor,
     currentSource.getDescriptors(), knnMatchesList, 2);
3 LinkedList<DMatch> good_matches = new LinkedList<DMatch>();
4     for (MatOfDMatch matOfDMatch : knnMatchesList) {
5         if (matOfDMatch.toArray()[0].distance / matOfDMatch.toArray()
              [1].distance < 0.7) {
6             good_matches.add(matOfDMatch.toArray()[0]);
7         }
8     }
```

Listing 4.18: Auszug MatchingTask: Feature-Matching mit OpenCv und Java

Das KNN-Matching, sowie der anschließende Ratio-Test nach Lowe sind in Listing 4.18 gezeigt. Nachdem mit Hilfe der von RANSAC gefundenen Transformationsmatrix die Polygone gefunden, sowie mit der resultierenden Inversen die Ergebnispolygone in Bildschirmkoordinaten transformiert werden, kann das aktuelle Ergebnis des Matchingmanagers überschrieben werden. Hierzu wird ein `MatchingResultEvent` geworfen das vom `MatchingManager` durch einen Listener registriert wird. Hier findet das zweite Producer-Consumer-Pattern statt, bei dem ein Arbeiterprozess ein Produzent und die OpenCv-Activity der Konsument ist. Während die Arbeiterprozesse permanent die Liste der aktuell sichtbaren Polygone überschreiben (Produzent), fragt die `OpenCvActivity` das aktuelle Ergebnis permanent ab, um es darzustellen.
Produziert ein `MatchingTask` während der Verarbeitung einen Fehler beim Matching durch zu wenige Features, Matches o.ä., wird die Schleife abgebrochen und sofort das nächste zu verarbeitende Bild aus dem Manager abgefragt.
Diese Umsetzung führt dazu, dass kleinere Fehler bei der Darstellung des Ergebnisses auf dem Bildschirm nicht ins Gewicht fallen, da das aktuelle Ergebnis nur im Falle eines Erfolges überschrieben wird.
Der Matchingzyklus setzt sich so lange fort, bis durch ein externes Ereignis die Methode `stopMatching()` des Managers aufgerufen wird. In dem Fall werden alle noch aktiven Prozesse zerstört, und der allokierte Speicher für die erzeugten `Mat`-Objekte freigegeben, wie aus dem Sequenzdiagramm A.9 hervorgeht.

```
1 private static final int MAX_WORKER_THREADS = 3;
2 //max. number of visible objects
3 public static final int MAX_OBJECTS = 10;
4 private static final int MAX_QUEUE_SIZE = 5;
```

Listing 4.19: Auszug MatchingManager: Einstellungen der Parameter

Für ein stabiles Verfahren haben sich durch empirisches Vorgehen die Werte aus Listing 4.19 ergeben. Für alle weiteren Feature-Matching, sowie RANSAC-Werte gilt die Belegung, die bereits bei der Webapplikation zum Erfolg führt. Startet der Anwender die Echtzeitberechnung, ist das erste Ergebnis frühstens nach dem ersten erfolgreichen Durchlauf eines `MatchingTask` zu sehen. Ein typischer Matchingdurchlauf ist dabei exemplarisch A.6 zu entnehmen. Zum Test der Matchingqualität werden in der App zusätzlich die letzten 10 Matching-Durchläufe gespeichert. Aus diesen kann ein prozentualer Anteil positiver / negativer Matches errechnet werden. Die Matchingrate wird auf dem Display angezeigt und beträgt, sofern der Anwender sich statisch bei hinreichender Beleuchtung verhält, 95 - 100 Prozent.

5 Fazit

5.1 Zusammenfassung

In dieser Arbeit wurde untersucht, ob und inwieweit sich eine mobile Anwendung entwickeln lässt, die basierend auf dem Framework OpenCv und einem Feature-Matching-Prozess zuverlässig Teilbilder in einem 2D-Kontext erkennt und dem Anwender dazu hinterlegte Informationen präsentiert.
Dabei können interessante Merkmale des Bildes (Features) dazu verwendet werden, Bildinhalte eindeutig zu beschreiben und diese miteinander zu vergleichen. Aufgrund des verwendeten SIFT-Algorithmus nach David Lowe können Bilder trotz verschiedener Deformationen, wie der Skalierung, Rotation, Beleuchtung oder Rauschen zuverlässig miteinander verglichen werden [Lowe 04]. Das Framework OpenCv stellt unter Anderem Methoden zur Merkmalsextraktion nach SIFT und zum Feature-Matching bereit. So lässt sich ein aufgenommenes Teilbild mit einem vorher abgelegten Urbild vergleichen und die entsprechende Region im Urbild finden, so dass dazu hinterlegte Informationen selektiert und zurückgegeben werden können. Diese Informationen können auf dem Display eines Smartphones als Augmentierung des aufgenommenen Teilbildes dargestellt werden.
Durch die Implementierung einer Webanwendung ist es nun möglich, mittels eines QR-Codes auf eine entsprechende Website weitergeleitet zu werden. Dort können die Teilbilder der Algorithmenfotos der Universität sowie von Gemälden des Felix-Nussbaum-Hauses aufgenommen, zur Verarbeitung zum Server geschickt und die Namen der Studenten bzw. geschichtliche Inhalte auf dem Display angezeigt werden.
Für den Fall, dass größere Datenmengen hinterlegt sind lassen diese sich im Sinne eines herkömmlichen Audioguides präsentieren. Die Weiterleitung auf die entsprechende Website erfolgt nur bei einer Internetverbindung. Sowohl eine stabile WLAN- als auch eine UMTS-Verbindung erweist sich, vor allem im Felix-Nussbaum-Haus aufgrund baulicher Strukturen als nicht zwangsläufig gewährleistet.
Die Integration der Kamera in den Browser ist plattformübergreifend möglich. Wird der Browser unterstützt kann ein akzeptables Matching-Ergebnis des gewünschten Teilbildes mit dem Urbild erzielt werden.
Die Matching-Qualität hängt vom Verhalten des Anwenders ab. Verhält dieser sich bei der Aufnahme des Teilbildes möglichst statisch und ist die Auflösung des aufgenommenen Bildes sowie die Beleuchtung hinreichend, wird erfahrungsgemäß in rund 8 von 10 Fällen ein positiver Match ermittelt. Negative Ergebnisse sind dabei z.B. auf eine unzureichende Beleuchtung zurückzuführen, weshalb nicht genügend Merkmale extrahiert und somit keine genaue Modellschätzung durch RANSAC möglich ist.
Ebenfalls negative Ergebnisse kommen zustande, falls der Anwender einen Aspekt des Urbildes untersucht, für den die Polygonkoordinaten im Vorhinein nicht genau genug eingezeichnet wurden. Somit bestimmen vor allem die Eingangsdaten die Matchingqualität. Tiefer gehende Tests bleiben im Weiteren aus, da dies der Rahmen der Arbeit nicht

erlaubt. Da die Wartezeit des Anwenders jedoch typischerweise unter fünf Sekunden pro aufgenommenen Teilbild liegt, wird dies für diese Arbeit als hinreichendes Ergebnis akzeptiert.
Aufgrund der nicht gewährleisteten Internetverbindung an unterschiedlichen Orten wurde zudem eine native Android-Applikation entwickelt, mit der es ebenfalls möglich ist die zu einer Bildinstanz hinterlegten Teilinformationen auf dem Display präsentiert zu bekommen, sobald die Smartphonekamera über einen Teilaspekt gehalten wird. Die hinterlegten Daten des entsprechenden Ortes können einmal zu Anfang eingelesen werden, damit vor allem im Museum keine weitere Internetverbindung notwendig ist. Die benötigte Zeit zum Herunterladen der Daten verhält sich proportional zur vorhandenen Datenmenge. Im Falle des Felix-Nussbaum-Hauses beschränkt sich diese auf rund 300 Exemplare, von denen meist ungefähr 50 unterschiedliche Bildinstanzen ausgestellt sind. Die App muss zunächst, sofern die Anwendung veröffentlicht wäre, aus dem Google-Playstore geladen werden. Nachdem der QR-Code der gewünschten Bildinstanz mit einem integrierten QR-Code-Scanner eingelesen ist, wird der Anwender zu einer Oberfläche weitergeleitet, in der zunächst die Vorberechnung der Urbildmerkmale zur aktuellen Bildinstanz stattfindet. Die Dauer der Vorberechnung hängt von der Größe der hinterlegten Urbilder ab. Alle Berechnungen nach Herunterladen der Daten laufen auf der smartphoneigenen CPU, weshalb ein Echtzeit-Verfahren ermöglicht wird. Mittels asynchroner Prozesse ist es möglich dem Anwender eine flüssige Darstellung hinterlegter Informationen über gewünschte Teilaspekte eines 2D-Kontextes zu liefern. Dieses Verfahren generiert, sofern der Anwender sich eine kurze Zeit pro gewünschtem Teilaspekt möglichst statisch verhält, zuverlässige Ergebnisse und eine Matchingrate zwischen 95 und 100 Prozent. Negative Matches sind auch hier unter Anderem auf unzureichende Beleuchtung und zu dynamisches Verhalten des Anwenders zurückzuführen.
Es stellt sich heraus, dass die gewählte Plattform die Möglichkeiten, sowie die Einschränkungen der Funktionalität der Anwendungen bestimmt. Der Kernprozess, des Teilbild-Matchings kann allerdings weitestgehend unabhängig von der Plattform konzeptioniert werden.
Im Falle der Website gestaltet sich, entgegen der vorherigen Erwartungen, nicht die Implementierung eines serverseitigen Feature-Matching-Prozesses, sondern die plattformunabhägige und anwenderfreundliche Integration der smartphoneeigenen Kamera in einen mobilen Browser als kompliziert. Der noch immer neue `WebRTC`-Standard zur plattformunabhängigen Kommunikation mit der Hardware des Endgerätes ist noch nicht in allen Browserversionen implementiert, weshalb die Anwendbarkeit der Applikation auf vorerst einige wenige Browser und Versionen eingeschränkt ist. Vor allem Funktionalitäten, wie das Heranzoomen an einen Teilaspekt, das Ändern der Auflösung des Videofeeds, sowie die Kameraintegration mittels WebRTC als solche sind noch nicht in jedem Browser gewährleistet.
Die native Applikation hingegen bedarf aufgrund der Echtzeitberechnungen zwar mehr Implementierungsaufwand, führt jedoch auch zu mehr Möglichkeiten. Durch die Einschränkung auf ein Betriebssystem ist mit nur einer begrenzten Menge an möglicher Hardware zu rechnen. Durch die einfache Zugänglichkeit der mobilen Kamera können Funktionen, wie das Heranzoomen an die Bildinstanz, das Ändern der Auflösung, sowie die Integration einer Autofokussierung erreicht werden. Vor allem die Autofokussierung, die durch den WebRTC-Standard noch nicht plattformübergreifend möglich ist, lässt sich vor allem beim Heranzoomen an ein Bild ein besseres Matchingergebnis erzielen,

da das Eingangsbild so eine bessere Auflösung aufweist. Die Lösung zur Integration der Kamera spielt somit in beiden Anwendungen eine wichtige Rolle um möglichst qualitativ hochwertige Eingangsdaten für einen Teilbild-Matching-Prozess zu gewährleisten. Durch die Art der Entwicklung beider Anwendungen ist es zusätzlich möglich ein beliebiges Bild, sowie dazu gehörige Informationen (Teilobjektkoordinaten und Textinformation) auf einem Server zu speichern, um sie durch einen QR-Code in die Anwendungen zu integrieren. Da die Webanwendung abgesehen von der aktuellen Browsereinschränkung stabil funktioniert, wird für die Bildinstanzen der Universität Osnabrück ein QR-Code vor jede Bildinstanz platziert, um diese zugänglich zu machen. Da die rechtliche Lage im Felix-Nussbaum-Haus bzgl. der Fotografie von Gemälden noch nicht eindeutig geklärt ist, bleibt dies vorerst aus.
Vor allem die native Android-Anwendung befindet sich noch in einem experimentellen Status und ist nur für eine Prozessorarchitektur entwickelt, könnte jedoch durch ein Multiplattform-Verfahren für verschiedene Prozessorarchitekturen im Google-Playstore veröffentlicht werden, so dass die App auf verschiedenen Android-Geräten aufrufbar ist.
Zum Ende dieser Arbeit ist also durch zwei verschiedene Ansätze gezeigt, dass ein mobiler auf Feature-Matching basierender Augmented-Reality-Ansatz möglich ist, um die Erkennung von Teilinhalten in 2D-Kontexten zu gewährleisten. Dabei kann der Matching-Prozess so konzeptioniert werden, dass dieser übertragbar auf weitere Anwendungsfälle ist.

5.2 Ausblick

Die entwickelten Anwendungen zeigen zwar schon ein relativ stabiles Verhalten, haben jedoch noch Optimierungspotenzial. Durch z.B. die Optimierung der einzelnen OpenCv-Aufrufe mittels *Mixed Processing* oder die Nutzung einer effizienten Datenbankstruktur zur Verwaltung bereits berechneter Deskriptoren können die Ergebnisse bezüglich der nicht-funktionalen Anforderungen verbessert werden.
Die Integration einer neuen Bildinstanz könnte durch eine weitere Anwendung vereinfacht werden, die es ermöglicht Formen um ein Urbild zu zeichnen und dazu Textinformationen anzugeben, welche in den entsprechenden Formaten in der Datenbank gespeichert werden.
Die eingeschränkte Browserkompabilität der Webanwendung wird höchstwahrscheinlich im Laufe der nächsten Zeit durch die Weiterentwicklung des WebRTC-Standards, sowie browsereigenen Updates verbessert.
Der notwendige QR-Code zum Starten der Anwendung könnte durch einen weiteren automatisierten Erkennungsschritt ersetzt werden, der effizient und zuverlässig das zur Bildinstanz zugehörige Urbild erkennt.
Die native App zeigt deutlich, dass es möglich ist einen stabilen anwendungstauglichen (Echtzeit-)Prozess mittels Feature-Matching-Mechanismen zu entwickeln, der in der Praxis eingesetzt werden kann.
Laut des Felix-Nussbaum-Hauses entspricht die entwickelte Lösung den Vorstellungen der zukünftigen Digitalisierung des Museums und könnte die traditionellen bis dato genutzen Audioguides ersetzen, indem die Anwendung als Basis eines Besucherleitsystems fungiert, das eine Erkundungstour eines Museums mittels Augmented-Reality ermöglicht. Die hinterlegten Informationen könnten dabei aus mehr als nur Textinformationen

bestehen und beispielsweise auf andere Bilder des selben Künstlers referenzieren, die als Information parallel zum Audioguide eingeblendet werden.
Auf die entwickelten mobilen Lösungen kann im Weiteren aufgebaut werden, indem das verwendete Verfahren optimiert und ggf. in seiner Funktionalität erweitert wird.
Insbesondere das Felix-Nussbaum-Haus der Stadt Osnabrück verfolgt eine mögliche Weiterentwicklung dieser Arbeit mit großem Interesse.

A Diagramme und Tabellen

A.1 Anwendungsfälle

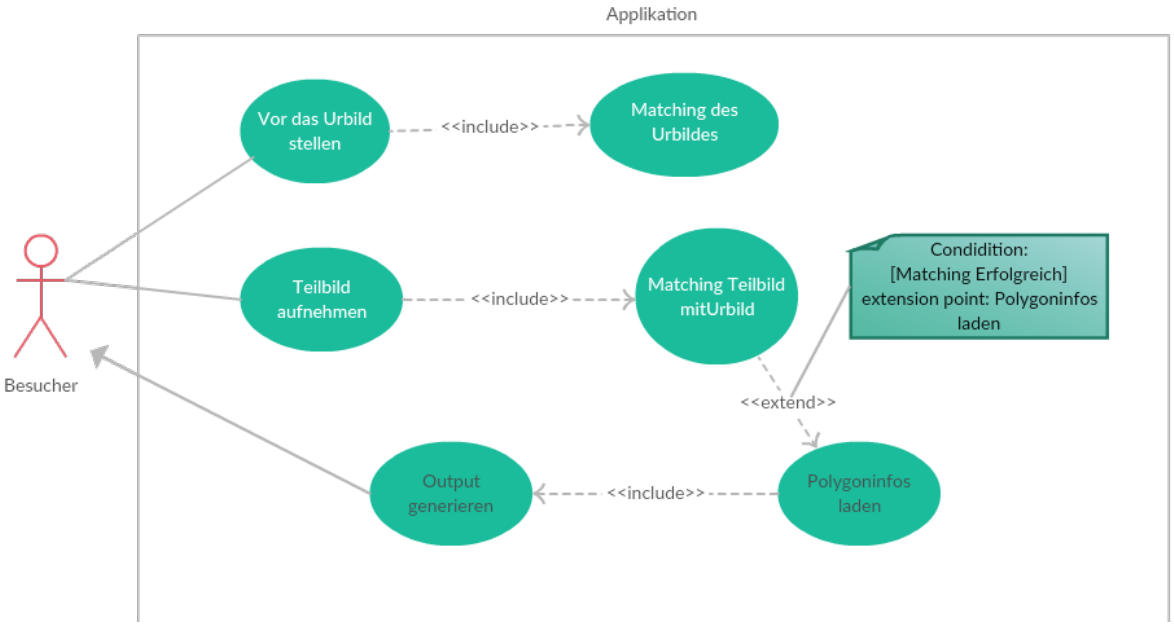


Abbildung A.1: Use-Case-Diagramm der Applikation nach UML 2.0

Use-Case-Id : 1	
Titel	Matching des Urbildes
Akteur	Der Besucher
Beschreibung	Der Anwendung muss das Urbild bekannt gemacht werden.
Vorbedingungen	1. Der User steht vor dem Bild 2. Daten zum Urbild liegen auf dem Server 3. Der User hat eine Verbindung zum Internet
Nachbedingungen	Die Anwendung kennt das Urbild
Ausnahmen & Fehler	Keine Daten vorhanden, Rückmeldung an den User

Tabelle A.1: Use-Case #1

Use-Case-Id : 2	
Titel	Aufnahme eines Teilbildes
Akteur	Der Besucher
Beschreibung	Mit der mobilen Kamera des Smartphones des Besuchers muss eine Aufnahme des Teilobjekts erfolgen, welche an die Verarbeitungseinheit weitergeleitet werden kann.
Vorbedingungen	1. Teil-sowie Urbild sind in digitaler Form vorhanden 2. Der User erteilt die Kameraberechtigung. 3. Der User hält seine Smartphone-Kamera auf einen Teilaspekt des Bildes, oder das Gesamtbild.
Nachbedingungen	Die Anwendung kennt das Teilbild und leitet es zur Verarbeitung weiter
Ausnahmen & Fehler	Die Qualität des aufgenommenen Teilbildes ist nicht ausreichend, der User macht eine Aufnahme etwas anderem als einem Teilobjekt des Bildes oder hat keine Berechtigung vergeben.

Tabelle A.2: Use-Case #2

Use-Case-Id : 3	
Titel	Matching des Teilbildes
Akteur	Matchingeinheit der Applikation
Beschreibung	Das Teilbild-Matching geschieht in mehreren Schritten: 1. Falls noch nicht geschehen, Extraktion der Urbild-Merkmale 2. Extraktion der Teilbild-Merkmale 3. Feature-Matching beider Bilder 4. Matchfiltering 5. Lokalisierung der genauen Region und enthaltenen Objekte um Urbild
Vorbedingungen	1. Teil-sowie Urbild sind in digitaler Form vorhanden. 2. **Case Website:** Es besteht eine Internetverbindung.
Nachbedingungen	Eine Matchingantwort (positiv oder negativ) ist vorhanden.
Ausnahmen & Fehler	1. Keine Features im Teilbild erkannt 2. Kein Match zwischen Teil-und Urbild 3. Keine oder falsche Polygone in der Matchingregion gefunden

Tabelle A.3: Use-Case #3

Use-Case-Id : 4	
Titel	Darstellung der Matchingantwort
Akteur	Die Applikation
Beschreibung	Das Ergebnis der Matching-Einheit wird dem Anwender auf dem Bildschirm präsentiert.
Vorbedingungen	1. Die Matching-Einheit liefert ein Ergebnis (positiv oder negativ). 2. Die Ergebnispolygone liegen in Bildschirmkoordinaten vor. 3. **Case Website:** Es besteht eine Internetverbindung.
Nachbedingungen	Der Anwender kennt die zum gewünschten Teilaspekt des Urbildes hinterlegte Information. Diese wird ihm, bei kleiner Information auf dem Display und bei größerer Datenmenge als Sprachausgabe präsentiert.
Ausnahmen & Fehler	1. Die Matching-Einheit liefert ein negatives Ergebnis. 2. Die Matching-Einheit liefert kein Ergebnis. 3. Die in der Antwort enthaltenen Polygone stimmen nicht mit der Wirklichkeit überein.

Tabelle A.4: Use-Case #4

A.2 Aktivitätsdiagramm der Webanwendung nach UML 2.0

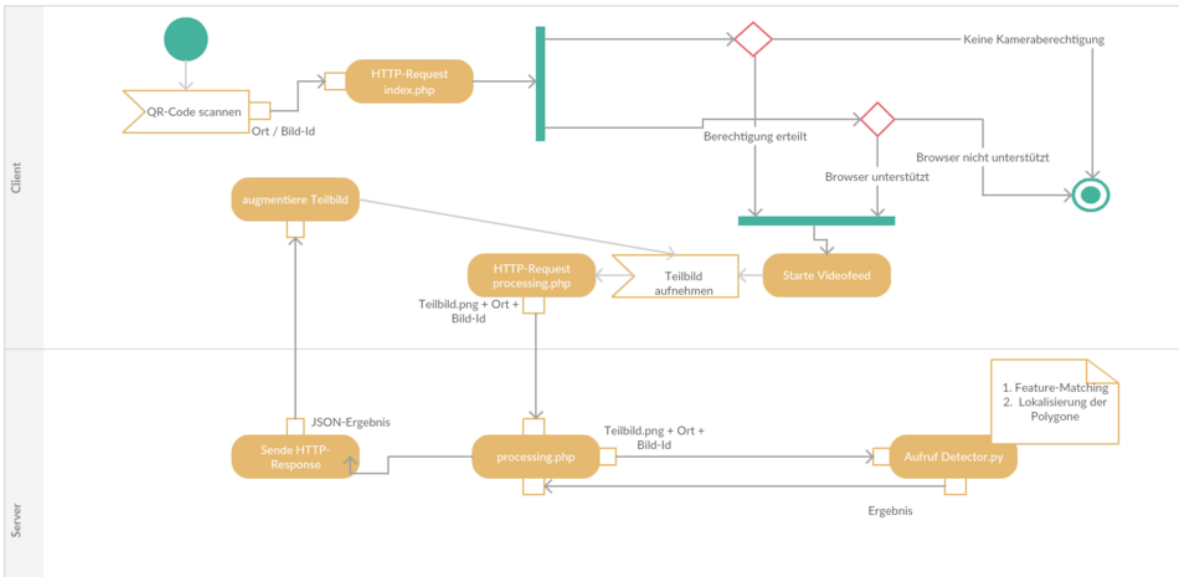


Abbildung A.2: Aktivitäsdiagramm der Webanwendung nach UML2.0

A.3 Programmablaufplan Teilbild-Matching

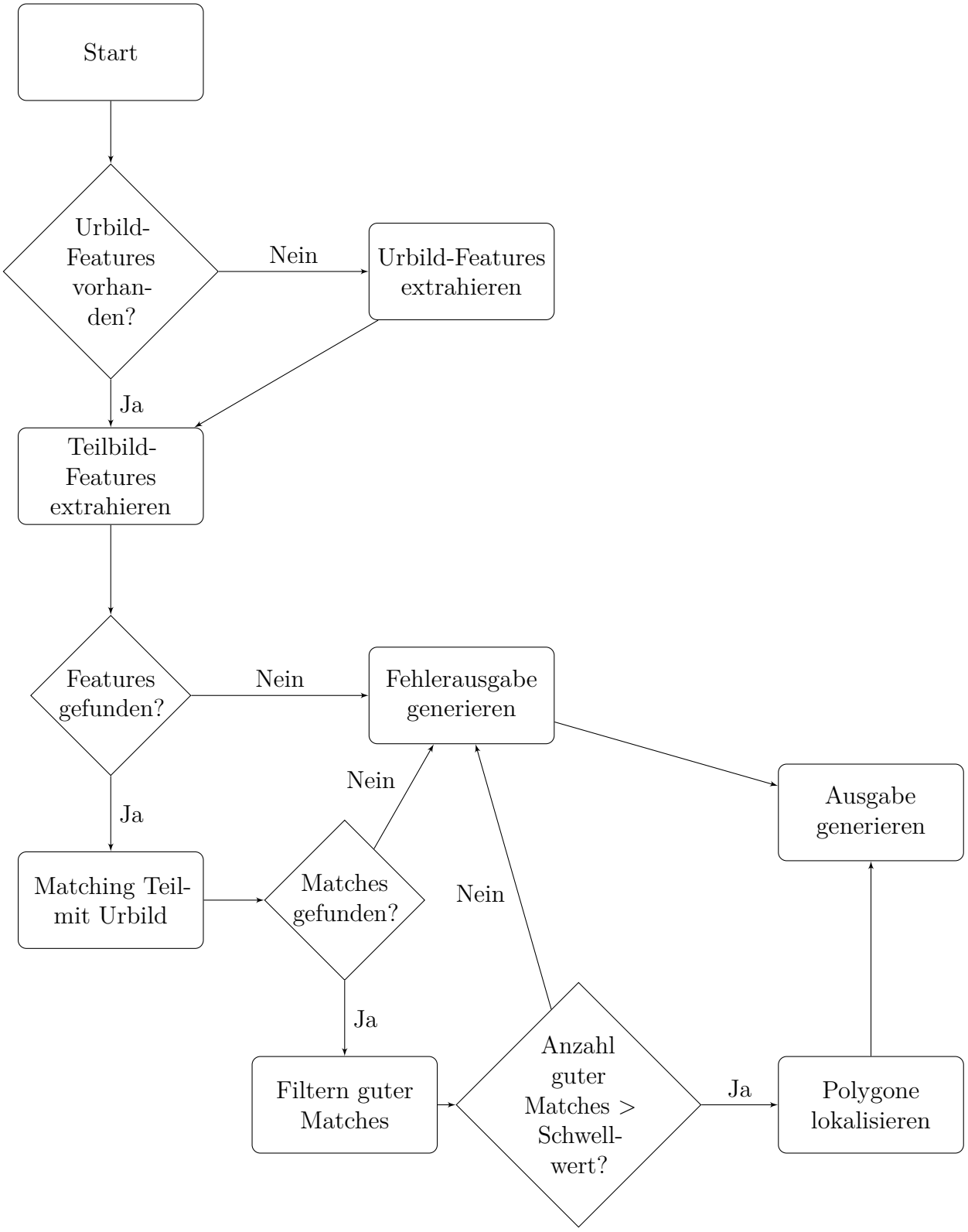


Abbildung A.3: Allgemeiner Programmablaufplan für `Teilbild-Matching`

A.4 Klassendiagramm der Android-Applikation

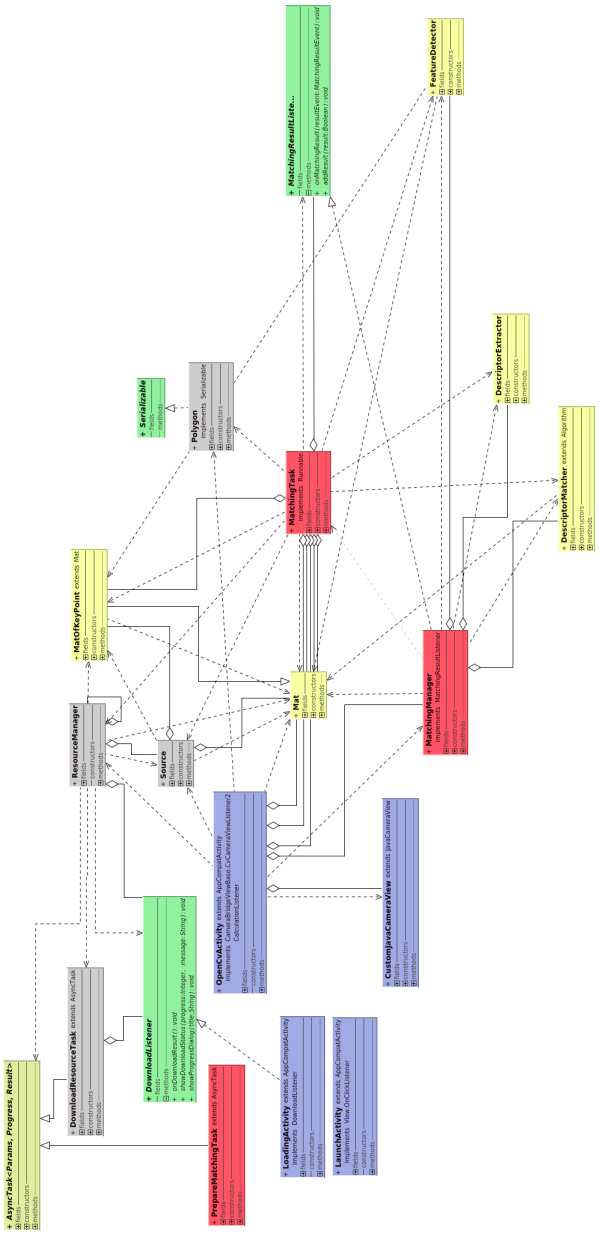


Abbildung A.4: UML-Klassendiagramm der nativen Echtzeit-App. Zu sehen sind die für die Darstellung zuständigen Android-Activities (blau), die für die Datenverwaltung notwendigen Strukturen (grau), die OpenCv-eigenen Klassen (gelb), sowie den darauf aufbauenden Matchingstrukturen (rot). Die Interfaces sind in grün dargestellt.

A.5 Sequenzdiagramme

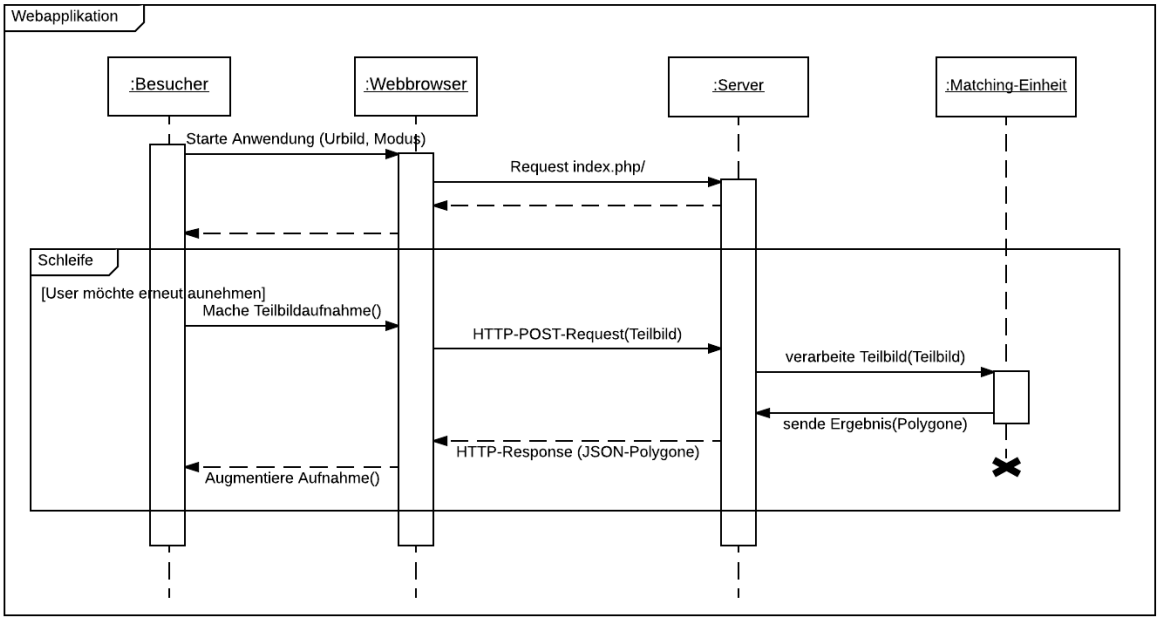


Abbildung A.5: Sequenzdiagramm der Webanwendung nach UML 2.0

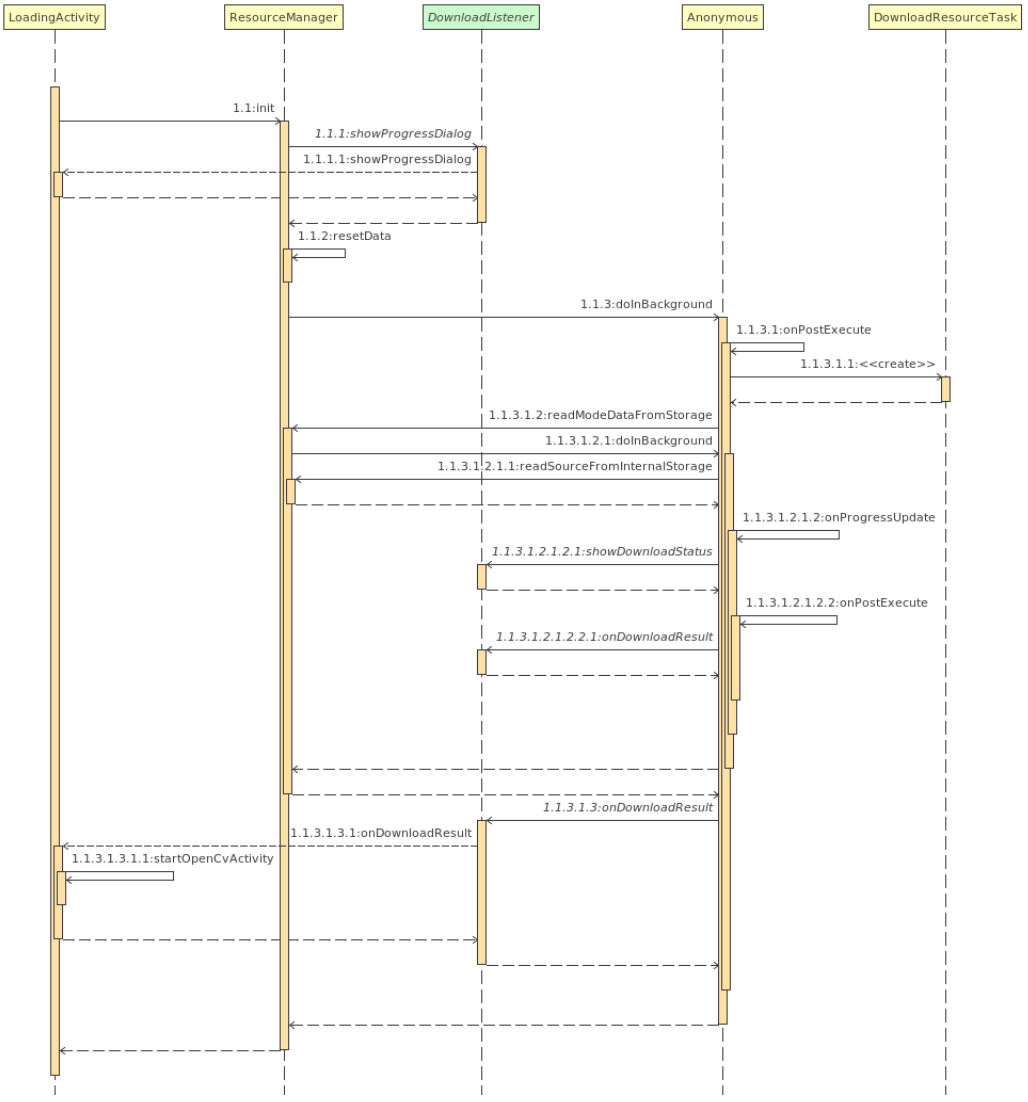


Abbildung A.6: Android-App: Initiierung der Daten (nach UML 2.0)

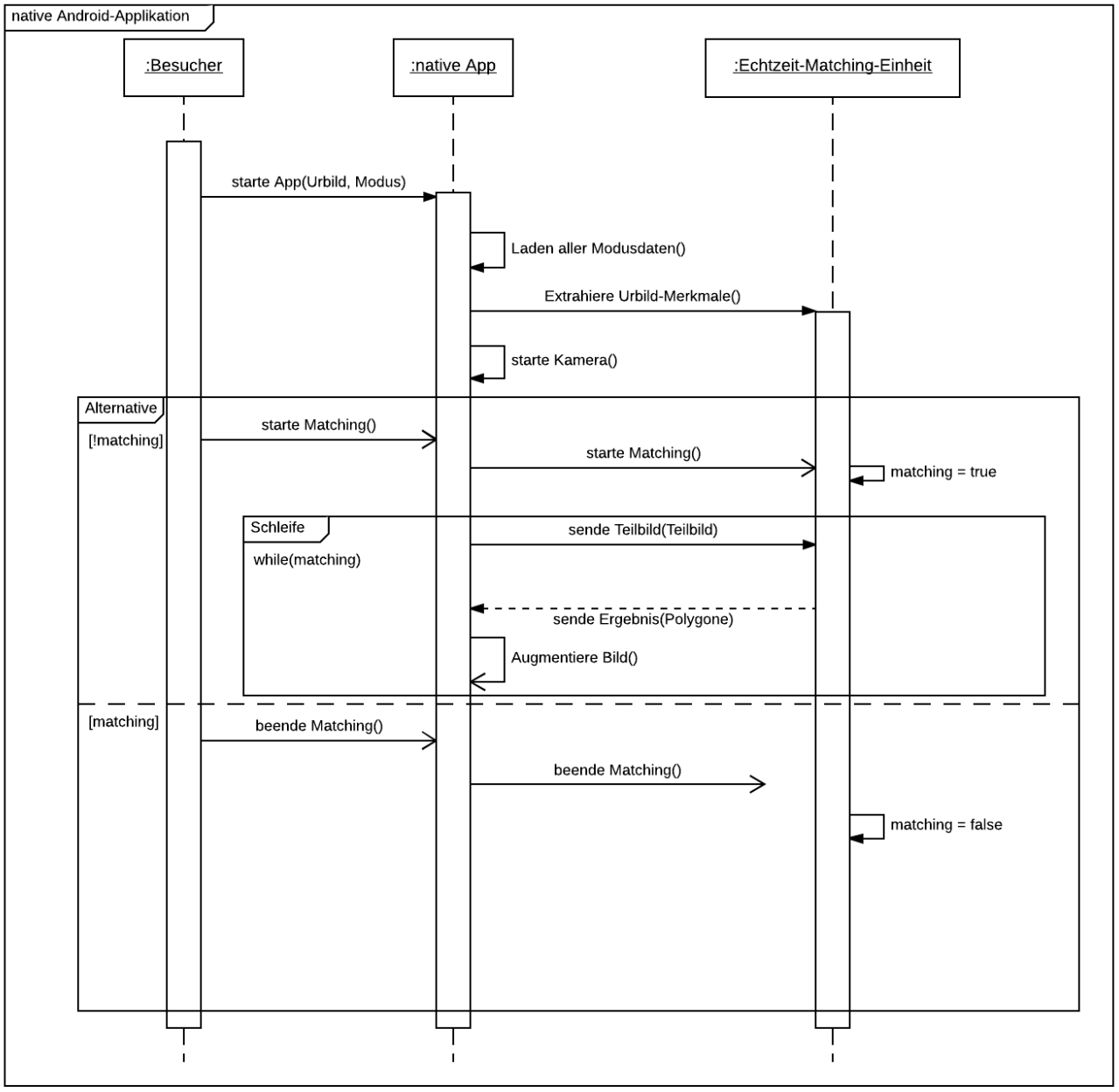


Abbildung A.7: Abstrahiertes Sequenzdiagramm der nativen Android-Applikation nach UML 2.0

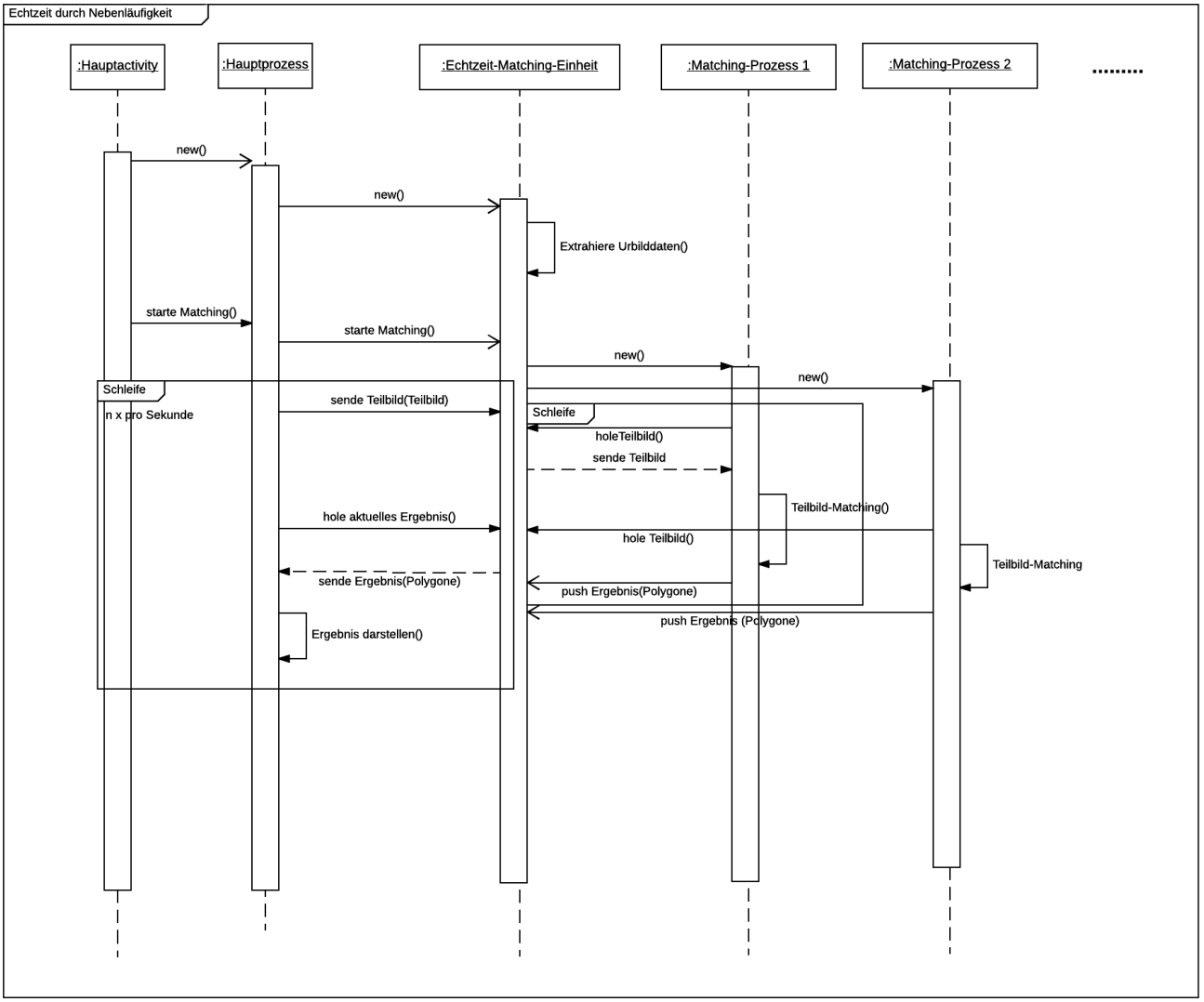


Abbildung A.8: Echtzeit durch Nebenläufigkeit. Sequenzdiagramm nach UML 2.0

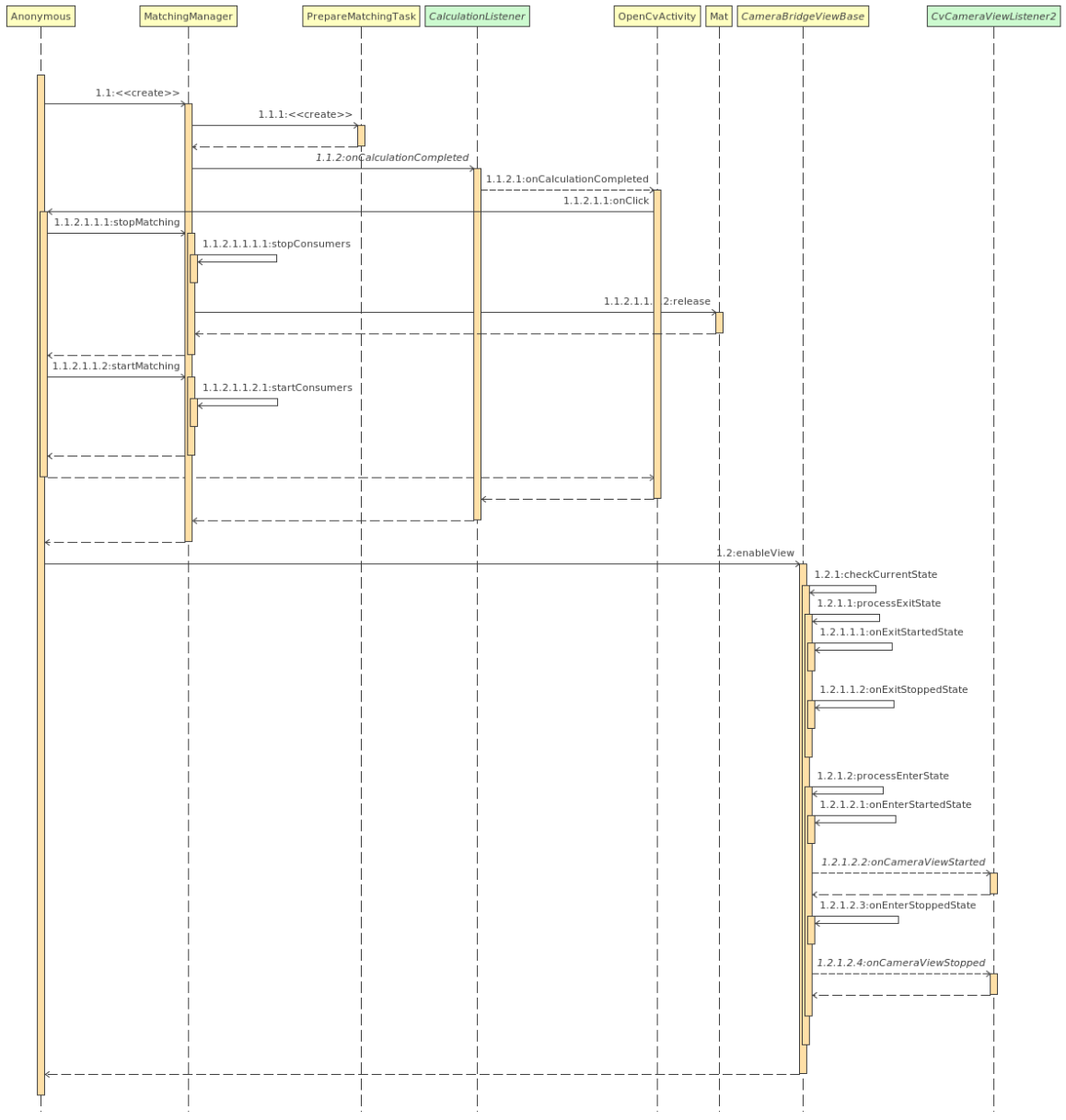


Abbildung A.9: Sequenzdiagramm der Manager-Instanziierung und Start der Kamera nach UML 2.0

A.6 Ausführungszeiten

Auflösung Urbild	760x620 Pixel
Auflösung Teilbild	480x640
Teilbild-Matching Dauer gesamt	1.4 Sekunden
Berechnung Urbild-Features u. Deskriptoren	0.6 Sekunden
Berechnung Teilbild-Features und Deskriptoren	0.22 Sekunden
Anzahl Features im Urbild	5833
Anzahl Features im Teilbild	717
Dauer KNN-Matching	0.43 Sekunden
Dauer Ratio-Test	0.00017 Sekunden
Dauer Schätzung durch RANSAC	0.0005
Dauer Lokalisierung der Polygone	0.000014 Sekunden

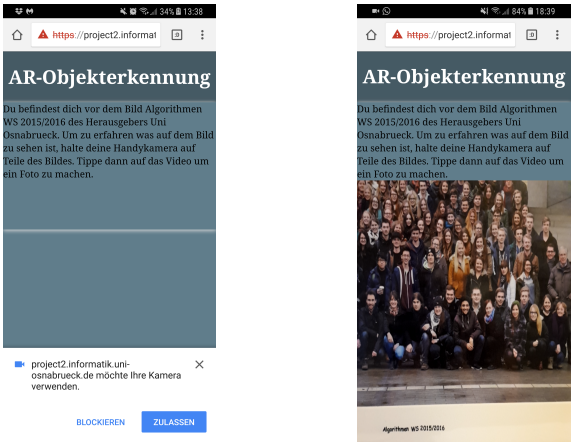
Tabelle A.5: Dauer eines typischen erfolgreichen Matching-Prozesses auf dem Server

Auflösung Urbild	760x620 Pixel
Auflösung Teilbild	360x360
Teilbild-Matching Dauer gesamt	0.9 Sekunden
Berechnung Urbild-Features u. Deskriptoren (vorab berechnet)	3.66 Sekunden
Berechnung Teilbild-Features und Deskriptoren	0.6 Sekunden
Anzahl Features im Urbild	5530
Anzahl Features im Teilbild	322
Dauer KNN-Matching	0.2 Sekunden
Dauer Ratio-Test	0.003 Sekunden
Dauer Schätzung durch RANSAC	0.000004
Dauer Lokalisierung der Polygone	0.0001 Sekunden

Tabelle A.6: Dauer eines typischen erfolgreichen Matching-Prozesses auf einem Samsung Galaxy S8

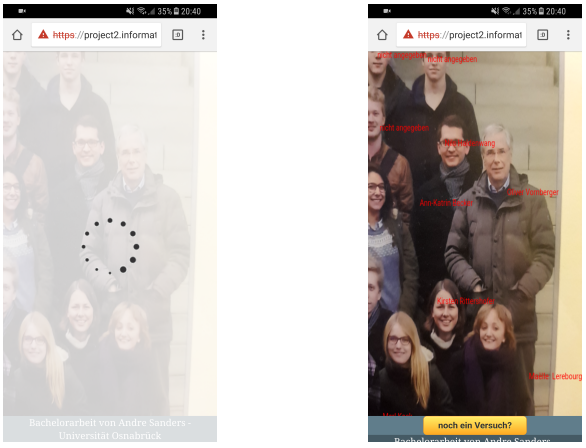
B Bildreihen

B.1 Bildreihe: Webanwendung



(a) (b)

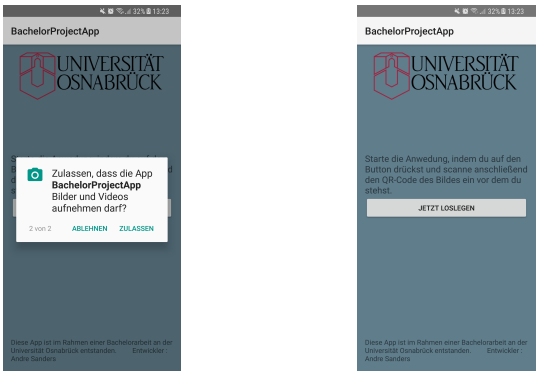
Abbildung B.1: Kameraberechtigung mittels GetUserMedia (a) und Oberfläche im Browser als Single-Page-Application (b)



(a) (b)

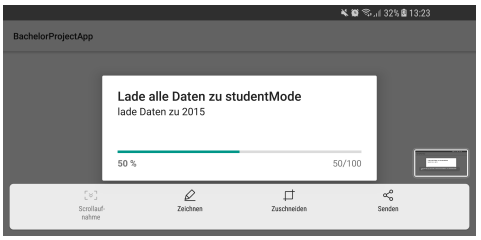
Abbildung B.2: Wartehaltung des mobilen Browsers nach Absenden des Teilbildes während des Teilbild-Matchings (a) und Augmentierung um die Matching-Antwort (b)

B.2 Bildreihe: Androidanwendung

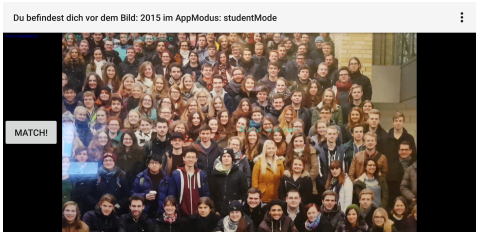


(a) (b)

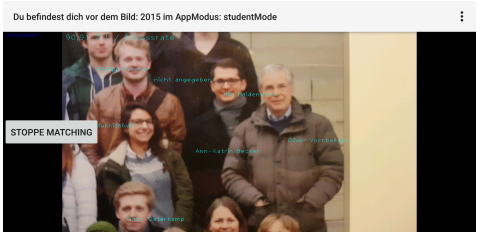
Abbildung B.3: Kameraberechtigung der Android-App (a) und Willkommensbildschirm (b) mit Aufforderung zum Einlesen des QR-Codes



(a)



(b)



(c)

Abbildung B.4: Verhalten der Android-App nach Einlesen des QR-Codes. Bestehend aus dem Herunterladen aller Daten (a), der Weiterleitung zur Hauptanwendung (b) und der Augmentierung des Bildes durch Feature-Matching (c)

B.3 Gemälde im Felix-Nussbaum-Haus

Abbildung B.5: Beispielhaftes Szenario eines Museumsbesuches. Auf dem Bild ist der "Triumph des Todes"des Künstlers Felix Nussbaum zu sehen. © VG Bild-Kunst, Bonn 2017

Abbildung B.6: Triumph des Todes inklusive interessanter Teilregionen. © VG Bild-Kunst, Bonn 2017

C Quellcode

Der Quellcode der entwickelten Webapplikation sowie der Android-App ist dem beiliegendem Datenträger zu entnehmen.

Webanwendung:
Der Code der Webanwendung ist auf dem Datenträger unter dem Pfad `MatchingWebsite` zu finden, wie er auch auf dem verwendeten Server angelegt ist. Die Struktur baut sich wie folgt auf:

- `/css`:
  Hier befindet sich das verwendete CSS-Stylesheet.
- `/js`:
  Hier befinden sich alle im Client ausgeführten Javascript-Dateien.
- `/public`:
  Hier befinden sich alle notwendigen Ressourcendateien, wie Templates und hinterlegte Urbildinformationen. Dabei sind unter `public/data/` die hinterlegten Daten des jeweiligen Modus, sowie unter `public/templates/` alle für die Oberfläche notwendigen Dateien hinterlegt.
- `api`:
  Hier befinden sich alle von außen ansprechbaren PHP-Skripte, sowie die Matching-Einheit `detector.py` und die zugehörigen Datenstrukturen. Des Weiteren ermöglicht der Order `api/tmp/` das Zwischenspeichern von Dateien.

Android-App:
Der Quellcode der Android-App ist auf dem Datenträger unter dem Pfad

    /MatchingApp/java/

zu finden. Dort befinden sich alle Klassen, auf denen die Anwendung basiert. Die kompilierten OpenCv-Module befinden sich jedoch im Verzeichnis

    /MatchingApp/jniLibs/

Unter dem Pfad

    /MatchingApp/res/

sind alle Layout-bezogenen Android-Dateien zu finden.

Abbildungsverzeichnis

Literaturverzeichnis

[Ahad 11] M. Ahad. *Computer Vision and Action Recognition: A Guide for Image Processing and Computer Vision Community for Action Understanding. Atlantis Ambient and Pervasive Intelligence*, Atlantis Press, 2011.

[Bay 06] H. Bay, T. Tuytelaars, and L. Van Gool. *SURF: Speeded Up Robust Features*, S. 404–417. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[Brow 02] M. Brown and D. G. Lowe. "Invariant Features from Interest Point Groups.". In: *BMVC*, 2002.

[Burg 09] W. Burger and M. J. Burge. *Digitale Bildverarbeitung: Eine Algorithmische Einführung Mit Java*. Springer-Verlag, 2009.

[Deve] "Development with OpenCv on Android". docs.opencv.org/2.4/doc/tutorials/introduction/android_binary_package/dev_with_OCV_on_Android.html.

[Effi] "Effiziente Repräsentation von hochdimensionalen Merkmalsvektoren für die Korrespondenzfindung". https://www.google.de/url?sa=t&rct=j&q=&esrc=s&source=web&cd=1&ved=0ahUKEwjBlrr-t-TWAhWI7hoKHUl4BZgQFggpMAA&url=https%3A%2F%2Fwww.mip.informatik.uni-kiel.de%2Ftiki-download_file.php%3FfileId%3D557&usg=AOvVaw1cFnSwTKZdJbfNkbV0Hkql.

[Feata] "Feature Matching + Homography to find Objects". OpenCv-HomografiemitRANSAChttps://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_feature_homography/py_feature_homography.html.

[Featb] "Feature Matching with FLANN". https://docs.opencv.org/3.1.0/d5/d6f/tutorial_feature_flann_matcher.html.

[Feli] "Felix Nussbaum Werkverzeichnis". http://www.osnabrueck.de/werkverzeichnis/archiv.php?lang=de&.

[Fisc 81] M. A. Fischler and R. C. Bolles. "Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography". *Communications of the ACM*, Vol. 24, Nr. 6, S. 381–395, 1981.

[Goog] "Google Patentsuche". https://www.google.com/patents/US6711293.

[Grau 11] K. Grauman and B. Leibe. "Visual object recognition - Local Features: Detection and Description". *Synthesis lectures on artificial intelligence and machine learning*, Vol. 5, Nr. 2, 2011.

[GREM 08] F. GREMSE. “Skaleninvariante Merkmalstransformation-SIFT Merkmale,"Hauptseminar Medizinische Bildverarbeitung 2005 RWTH-Aachen.“, 2005, S. 149–164”. *URL http://phobos. imib. rwth-aachen. de/lehmann/seminare/bv_2005. pdfStand: Nov*, 2008.

[Harr] “Harris corner detector”. https://docs.opencv.org/2.4/doc/tutorials/features2d/trackingmotion/harris_detector/harris_detector.html.

[Harr 88] C. Harris and M. Stephens. “A combined corner and edge detector.”. In: *Alvey vision conference*, S. 10–5244, Manchester, UK, 1988.

[Hass 16] M. Hassaballah, A. A. Abdelmgeid, and H. A. Alshazly. *Image Features Detection, Description and Matching*, S. 11–45. Springer International Publishing, Cham, 2016.

[Hert 12] J. Hertzberg, K. Lingemann, and A. Nüchter. *Mobile Roboter: Eine Einführung aus Sicht der Informatik*. Springer-Verlag, 2012.

[Hows 15] J. Howse. *Android Application Programming with OpenCV 3*. Packt Publishing, 2015.

[Intra] “Introduction to SIFT (Scale-Invariant Feature Transform)”. http://opencv-python-tutroals.readthedocs.io/en/latest/py_tutorials/py_feature2d/py_sift_intro/py_sift_intro.html.

[Intrb] “Introduction to SURF”. https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_feature2d/py_surf_intro/py_surf_intro.html.

[Isik 14] Şahin Isik. “A Comparative Evaluation of Well-known Feature Detectors and Descriptors”. *International Journal of Applied Mathematics, Electronics and Computers*, Vol. 3, Nr. 1, S. 1–6, 2014.

[Lind 94] T. Lindeberg. “Scale-space theory: A basic tool for analyzing structures at different scales”. *Journal of applied statistics*, Vol. 21, Nr. 1-2, S. 225–270, 1994.

[Lowe 04] D. G. Lowe. “Distinctive Image Features from Scale-Invariant Keypoints”. *International Journal of Computer Vision*, Vol. 60, Nr. 2, S. 91–110, Nov 2004.

[Lowe 99] D. G. Lowe. “Object Recognition from Local Scale-Invariant Features”. In: *Proceedings of the Seventh IEEE International Conference on Computer Vision*, S. 1150–1157 vol.2, 1999.

[Miko 02] K. Mikolajczyk. *Detection of local features invariant to affines transformations*. PhD thesis, Institut National Polytechnique de Grenoble-INPG, 2002.

[Miko 04] K. Mikolajczyk and C. Schmid. “Scale & affine invariant interest point detectors”. *International journal of computer vision*, Vol. 60, Nr. 1, S. 63–86, 2004.

[Mora 80] H. P. Moravec. "Obstacle avoidance and navigation in the real world by a seeing robot rover.". Tech. Rep., STANFORD UNIV CA DEPT OF COMPUTER SCIENCE, 1980.

[Moul 12] P. Moulon, P. Monasse, and R. Marlet. "Adaptive structure from motion with a contrario model estimation". In: *Asian Conference on Computer Vision*, S. 257–270, Springer, 2012.

[offi] "official OpenCv homepage". opencv.org.

[Open] "OpenCV Modules Version 3.2.0". http://docs.opencv.org/3.2.0/.

[Sale 12] S. Saleem, A. Bais, and R. Sablatnig. "A performance evaluation of SIFT and SURF for multispectral image matching". *Image Analysis and Recognition*, S. 166–173, 2012.

[Scal] "Scale Invariant Feature Transform". https://kogs-www.informatik.uni-hamburg.de/~seppke/content/teaching/sose16/bvproj/CM-SIFT15.pdf.

[Szel 11] R. Szeliski. *Computer Vision - Algorithms and Applications*. Springer, London, 2011.

[Undea] "Understanding Features". http://docs.opencv.org/3.3.0/df/d54/tutorial_py_features_meaning.html.

[Undeb] "Understanding K-Nearest-Neighbour". https://docs.opencv.org/3.0-beta/doc/py_tutorials/py_ml/py_knn/py_knn_understanding/py_knn_understanding.html.

[Zhen 99] Z. Zheng, H. Wang, and E. K. Teoh. "Analysis of gray level corner detection". *Pattern Recognition Letters*, Vol. 20, Nr. 2, S. 149–162, 1999.

Abkürzungsverzeichnis

o.g.	oben genannte/n
z.B.	zum Beispiel
Abb.	Abbildung
CV	Computer Vision
HTML	Hypertext Markup Language
CSS	Cascading Style Sheets
RTA	real-time-application
RTP	real-time-processing
POI	Point of interest
ROI	Region of interest
SIFT	Scale Invariant Feature Transform
SURF	Speeded Up Robust features
KNN	K-Nearest-Neighbour
LoG	Laplacian of Gaussian
DoG	Difference of Gaussian
BRISK	Binary Robust scalable keypoints
BRIEF	Binary robust independent elementary features
ORB	Oriented FAST and Rotated BRIEF
FAST	Features from Accelerated Segment Test
HTML	Hyptertext Markup Language
QR	Quick Response
ICIAR	International Conference for Image Analysis and Recognition
RANSAC	RANdom SAmple Consensus
FLANN	Fast Library for Approximate Nearest Neighbors
OpenCL	Open Computing Language

Ich versichere, dass ich die eingereichte Bachelorarbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommene Stellen habe ich kenntlich gemacht.

Osnabrück, den 16. November 2017

(Andre Sanders)