

HF-ICT

IT-PROJEKT DES 6. SEMESTERS

Kassenzettel-Management App

Timo Dörflinger
9. Juni 2018

Inhaltsverzeichnis

1 Einleitung	3
2 Projekt-Management	4
3 Basis der App	4
4 SwiftOCR	5
5 Tesseract	6
5.1 Optionen von Tesseract-OCR-iOS	8
6 git	9
6.1 git lfs	9
7 Festgestellte Probleme	11
7.1 Die richtige Perspektive	11
7.2 Nach einem Grosseinkauf	12
8 OpenCV und Stitching	12
8.1 OpenCV	12
8.2 Stitching	15
9 Master App kassenzettel-management	18
10 Bildaufbereitungs-Funktionen - OpenCV	18
11 Weitere verwendete Software	20
11.1 Pods/CocoaPods	20
11.2 Homebrew	20
12 Mock-Up	21
13 Aufgaben für die zweite Phase - Diplomarbeit	23
14 Was ich gelernt habe	24
15 Quellenverzeichnis	24
16 Bilderverzeichnis	25

1 Einleitung

In meiner Diplomarbeit erstelle ich eine App zur Kassenzettel-Verwaltung. Für meinen Auftraggeber und mich persönlich sind die Kassenzettel verlorene Daten, ausser man verwendet Sie zur persönlichen Analyse bzw. zum Führen eines Haushaltsbuches. Daher erstelle ich in meiner Diplomarbeit eine iOS-App, mit welcher ein Kassenzettel fotografiert werden kann. Aus diesem erstellten Abbild werden dann gewisse Werte bzw. Daten mit Hilfe eines bereits bestehenden OCR-Frameworks ausgelesen und zur weiteren Verarbeitung zwischengespeichert. Hier kann also unter anderem der Gesamtbetrag ausgelesen und mit einem gesetzten monatlichen Budget verrechnet werden. Dies soll dann auch kategorisiert werden können. Daraus ist ersichtlich, was in dem laufenden Monat bereits in den verschiedenen Kategorien ausgegeben wurde. Die Kategorien können jeweils individuell erstellt werden.

Der Designaufbau bzw. die Benutzeroberfläche wird in der ersten Phase nicht beachtet. Hier wird aber mit dem Auftraggeber mit einem Mockup(Fussnote) ein Design erstellt, welches dann in der zweiten Phase umgesetzt werden soll.

Beide Apps setzen auf die selbe Basis. Mit der Basis der App kann ein Foto erstellt oder ein bereits erstelltes Foto aus der Fotobibliothek ausgewählt werden. Für die OCR-Funktion, also das Auslesen des Textes aus den Bildern, sind zuvor zwei verschiedene OCR-Frameworks ausgewählt worden. Allerdings wollte ich mich nicht nur nach den Dokumentationen der Frameworks entscheiden. Daher habe ich entschieden, zwei Test-Apps aufzubauen.

Nach der Entscheidung bin ich auf diverse Probleme mit den Kassenzetteln gestossen. Wird ein Kassenzettel nicht genau vertikal und horizontal ausgerichtet abfotografiert, entsteht ein verzerrtes Bild, das von dem OCR nicht oder nur schlecht ausgelesen werden kann. Hier habe ich diverse Funktionen gefunden, mit denen diese Probleme gelöst werden können.

Des Weiteren können die erstellten Abbilder der Kassenzettel mit weiteren Bildaufbereitungs-Funktionen weiter bearbeitet werden, sodass die Ausleserate steigt. So kann nun ziemlich erfolgreich der bezahlte Endbetrag ausgelesen werden. Dieser steht somit für die weitere Verarbeitung in der zweiten Phase der Diplomarbeit bereit.

2 Projekt-Management

Für dieses Projekt hatte ich für das Management die Hermes5-Management-Methode ausgewählt. Diese Methode hatten wir im 5. Semester kennengelernt und auch bereits in einem Projekt im gleichen Semester erfolgreich anwenden können. In dem Projekt im 5. Semester habe ich die Hermes5-Methode zu schätzen gelernt. Es werden zwar viele Dokumente verlangt, jedoch bringt Hermes5 auch einen hilfreichen roten Faden mit sich. Dieser kann einen erfolgreich durch ein Projekt führen.

Zu Beginn des Projekts lief es auch gut mit Hermes5. Der Projektauftrag für den Auftraggeber konnte schnell nach dem Hermes5-Template erstellt werden. Und auch die Planungsphase hat gut funktioniert, wobei ich hier bezüglich dem zeitlichen Rahmen viele Dokumente nicht erstellen konnte. Als ich aber dann in der Realisierungsphase Probleme erkannt hatte, bin ich wieder zurück in die Planungsphase um diese Probleme zu lösen. Das hat mich völlig aus dem Management geworfen.

Ich empfinde Hermes5 immer noch als eine hervorragende Projektmanagement-Methode. Hermes5 ist für reine Software-Projekte aber nicht so geeignet. Daher werde ich die zweite Phase der Diplomarbeit mit Scrum managen.

3 Basis der App

Für die Evaluierung der beiden OCR-Frameworks habe ich eine Basis-App erstellt. Auf dieser können dann die beiden Frameworks eingebaut und getestet werden.

Die iOS App wird mit einem SingleView-Template aufgebaut. Diesem Template habe ich einen Button hinzugefügt. Dieser Button hat zwei Optionen. Mit der ersten Option kann ein Bild erstellt werden, es wird also die Kamera-App geöffnet. Mit der zweiten Option wird die Bildergalerie geöffnet und es kann ein bereits erstelltes Foto ausgewählt werden. Diese zwei Button-Optionen sind nach einer YouTube-Tutorial von Brian Advent nachgebaut worden. Zu dem Button habe ich ein ImageView hinzugefügt. In diesem ImageView wird das erstellte oder ausgewählte Bild angezeigt. Dieses ImageView soll nur in dieser Projektphase bestehen. Es dient zur Anschauung des Bildes, bevor es von dem OCR-Framework ausgelesen wird. Unter dem ImageView befindet sich noch ein TextView, in dem der ausgelesene Text dargestellt wird.

4 SwiftOCR

SwiftOCR ist ein sehr starkes OCR, welches auf einer lernfähigen Basis aufbaut. Dies ist in dieser Hinsicht interessant, als das SwiftOCR besonders bei Fotos mit verschiedenen Schatten verwendet werden kann. Das ist für das Ziel dieses Projekts besonders interessant, da die App für das Auslesen von Kassenzetteln ausgerichtet ist. SwiftOCR verspricht hier also, den Text einwandfrei auslesen zu können, trotz Knitterfalten und damit einhergehender unterschiedlicher Schatten.

Das Framework war zu Beginn relativ schwierig in das Swift-Projekt einzubinden. Das Framework baut auf einem weiteren Framework, dem GPUImage-Framework, auf. Dieses ist bereits in dem SwiftOCR-Framework enthalten. Um SwiftOCR also einbinden und verwenden zu können, musste das GPUImage-Framework auch in das Projekt integriert werden.

Das SwiftOCR Framework ist auf der github-Seite(Fussnote mit der Quelle) zu finden. In dieser Repository ist auch ein Textfile, welches beschreibt, wie das Framework in das eigene Projekt eingebaut werden kann.

Nun kommt es für einen erfolgreichen Gebrauch des SwiftOCR darauf an, die zwei richtigen Ordner von SwiftOCR und GPUImage in der richtigen Reihenfolge auszuwählen, da es sonst nicht funktioniert. Denn nach der Integration des Framework-Ordners erscheinen in der Projekt-Struktur in XCode mehrere Ordner für SwiftOCR und GPUImage. Dieser Prozess hat mich sehr viel Zeit gekostet, da ich nach drei eigenen fehlgeschlagenen Versuchen die richtigen Ordner zu finden, externe Hilfen verwenden musste. Auf einer langen Suche nach einer Anleitung habe ich dann noch zwei weitere Versuche mit einer vielversprechenden Anleitung benötigt, bis die Framework-Umgebung richtig eingebunden war. Die Integration des Codes war dann kein Problem, da SwiftOCR lediglich fünf Zeilen Code braucht.

```
1 import SwiftOCR
2
3 let swiftOCRInstance = SwiftOCR()
4
5 swiftOCRInstance . recognize ( myImage )
6 {
7     recognizedString in print ( recognizedString )
8 }
```

Auch der erste Test mit den bereits mitgelieferten Test-Bildern war erfolgreich und für mich sehr vielversprechend. An diesem Punkt dachte ich sogar bereits, ich habe mich bereits für eine Option entschieden, ohne Tesseract bisher getestet zu haben.

Leider kommt es jedoch häufig anders als zuerst gedacht. Es stellte sich nach weiteren Tests mit Kassenzettel-Bildern jedoch leider heraus, das SwiftOCR nicht mit Text bzw. mit mehrzeiligem Text umgehen kann. Nach wiederholtem Durchlesen der Beschreibung von SwiftOCR hatte ich dann auch im Kleingeschriebenen gefunden, dass SwiftOCR lediglich für einzeiligen Text gemacht wurde. Dieser Stand kann auch nicht durch weitere Einstellungen geändert werden.

Mein Fazit von SwiftOCR: Wenn die Beschreibung wirklich genau gelesen wird und damit sicher ist, dass SwiftOCR für den eigenen Gebrauch genau das richtige ist, dann ist SwiftOCR eine wirklich starke Software. Es bringt viele interessante und potenzielle Punkte mit sich wie das erfolgreiche Auslesen trotz verschiedener Schatten oder der bereits eingeübten lernfähigen Basis. Jedoch könnte SwiftOCR noch besser und genauer beschrieben werden und die Herausforderung mit dem integrieren der Framework-Ordnerstruktur könnte auch noch um einiges vereinfacht werden. Damit eignet es sich leider nicht als OCR-Lösung für dieses Projekt.

5 Tesseract

Auf Tesseract bin ich aufmerksam geworden, da SwiftOCR sich auf der eigenen GitHub-Seite mit Tesseract vergleicht. Tesseract selbst hat bereits eine lange Entwicklungsphase hinter sich. Angefangen hatte es zwischen 1985 und 1995 bei Hewlett-Packard (HP). Danach hatte HP kein Interesse mehr daran und nach ein paar weiteren Jahren kam die Software dann 2005 zu Google, wo es als OpenSource mit der Apache-Lizenz auf SourceForge freigegeben wurde. Seit diesem Zeitpunkt fand Tesseract viele weitere Verbesserungen und Erweiterungen, unter anderem auch die Verfügbarkeit für Smartphones. So wurde für dieses Projekt *Tesseract OCR iOS* als zweite Möglichkeit für die OCR-Komponente getestet.

Tesseract-OCR-iOS hat auf der GitHub-Seite ein eigenes Wiki, in dem auch eine Installations-Anleitung beschrieben ist. Diese ist aber nicht mehr ganz auf dem aktuellen Stand. Daher habe ich Tesseract dann in der Test-App nach einer Video-Anleitung auf YouTube aufgebaut. Brian Advent beschreibt in diesem Video die Installation von Tesseract-OCR-iOS selbst auf der Basis der Anleitung der Tesseract GitHub-Seite.

Für den Aufbau der Test-App habe ich die Basis-App als neues Projekt kopiert und darin Tesseract installiert. Dieser Schritt ist um einiges einfacher als bei SwiftOCR. Denn Tesseract kann über Cocoa Pods (siehe Kapitel: Weitere verwendete Software) installiert werden. Dafür wird der Terminal auf dem Mac geöffnet und in den Projekt-Ordner der Test-App gewechselt.

In diesem Ordner wird die Installation mittels Cocoa Pods mit dem Befehl *init pods* initialisiert. Dieser Befehl erstellt eine Pods-Datei in dem Projekt-Ordner. Diese kann nun geöffnet und die zu installierenden Frameworks oder Erweiterungen eingetragen werden. Somit wird die Pods-Datei mit dem Befehl *pod 'TesseractOCRiOS', '4.0.0'* eingetragen. Danach wird die Pods-Datei geschlossen und mit dem Befehl *pods install* installiert. Nun installiert Cocoa Pods das Framework in dem Projekt und damit in der Test-App. Das dauert nur wenige Sekunden. Dann ist die Installation des Frameworks bereits abgeschlossen. Wird nun das Projekt in XCode wieder geöffnet, steht alles für den Tesseract-Test bereit.

Ich bin der Video-Anleitung weiter gefolgt und habe zuerst eine einfache Benutzeroberfläche erstellt, die lediglich ein *TextField*, also ein Textfeld, enthält. Dieses gibt später den ausgelesenen Text wieder. Ist das erledigt, kann das Textfeld mit dem Code verbunden werden. Nun kann der Tesseract-Code eingebaut werden. Hier war ich überrascht, dass es wie SwiftOCR lediglich ein paar einzelne Zeilen Code benötigt, bis Tesseract dann gleich verwendet werden kann.

```
1 import TesseractOCR
2
3 var tesseract : G8Tesseract = G8Tesseract
4   (language: "eng+deu")
5 tesseract.delegate = self
6 tesseract.image = UIImage(named: "image_sample.jpg")
7 tesseract.recognize()
8 print(tesseract.recognizedText)
```

5.1 Optionen von Tesseract-OCR-iOS

Tesseract-OCR-iOS bringt viele Einstellungs-Möglichkeiten. Unter anderem gibt es mittlerweile viele verschiedene Sprachpakete, die eingebunden werden können. Tesseract selbst bietet nach der Installation lediglich das Englisch-Sprachpaket. Alle weiteren Sprachpakete, so auch das deutsche Sprachpaket, müssen einzeln nachinstalliert werden.

Tesseract bietet bereits selbst einige Optionen, um die Bilder, die ausgelesen werden sollen, zu bearbeiten. Damit sind also Bildaufbereitungs-Funktionen gemeint. So bietet Tesseract eine Grau- oder Schwarz-Weiss-Option. Dies kann einfach hinter dem Auslese-Befehl angehängt werden.

```
1 tesseract.image = UIImage  
2     (named: "b_1_q_0_p_0_2")?.g8_blackAndWhite()
```

Für dieses Projekt habe ich mich allerdings dagegen entschieden und dafür als Alternative OpenCV (siehe Kapitel: „OpenCV“) gewählt. Der Grund ist, dass die Bildaufbereitungs-Funktionen relativ begrenzt sind und nur mit dem direkten Auslese-Befehl verwendet werden können. So können diese Bildaufbereitungs-Funktionen von Tesseract nicht noch mit weiteren externen Bildaufbereitungs-Funktionen kombiniert werden.

Ein wichtiger Teil, der für dieses Projekt eine grosse Steigerung in der Ausleserate brachte, ist die sogenannte *Page Segmentation Method*. Das bedeutet, Tesseract kann verschiedene Text-Formate auslesen. Für Tesseract ist es ein Unterschied, ob es einen mehrzeiligen Text oder nur ein Wort in einem Bild auslesen muss. Dafür hat Tesseract verschiedene Auslesetechniken. In der Regel versucht Tesseract den Unterschied bzw. das in dem Bild vorliegende Textformat selbst zu erkennen. Da es in dieser App aber nur um Kassenzettel geht, kann Tesseract auf das für diesen Fall passendste Format angepasst werden. Das steigert in diesem Fall enorm die Ausleserate, da so immer das richtige Textformat verwendet wird.

6 git

Bei der Versionsverwaltung habe ich mich für Git entschieden. Hauptsächlich hatte ich mich für Git entschieden, da es bereits im 5. Semester im Unterricht von Herrn Tanner Thema war und wir damit gearbeitet haben. Zuvor hatte ich noch keinen Umgang mit solchen Versionsverwaltungs-Programmen. Mir war immer bewusst, dass diese ein Hauptbestandteil bei der Softwareentwicklung sind. Daher war mir gleich zu Beginn dieser Diplomarbeit bewusst, dass ich auf eine Versionsverwaltung setze.

Zu Beginn hatte ich noch mit dem GitHub-Desktop Programm gearbeitet. Dies hat mir den Einstieg noch etwas weiter vereinfacht. Schnell habe ich jedoch festgestellt, dass im diesem GitHub-Desktop Kleinigkeiten gegenüber des Managements von Git über den Terminal fehlen. Zum Beispiel ist die Übersicht der Datenraten-Übertragung im Terminal ersichtlich, die im GitHub-Desktop gefehlt hatte. Da ich aber später auf `git lfs` setzen musste und das im GitHub-Desktop nicht unterstützt war, bin ich dann auf das Management im Terminal übergegangen.

6.1 git lfs

Bei dem Pushen der fertigen Test-App mit dem Foto-Stitchen auf mein GitHub-Repository musste ich feststellen, dass das beinhaltete OpenCV2-Framework, welches in der App enthalten sein muss, nicht auf GitHub gepusht werden konnte, weil das Framework mit 250MB das 100MB-Limit von GitHub-Dateien überschritt. Das Problem hierbei ist nun, dass das in der App eingebaute Framework schnell Probleme macht, sobald es mal aus der Ordnerstruktur herausgenommen und wieder eingebaut wird. Daher musste ich hier eine Lösung finden, wie ich die Test-App (Stitching) trotz des grossen Frameworks am besten in einer Projektstruktur auf mein GitHub-Repository bringen kann.

Hier hat Git glücklicherweise bereits seit längerer Zeit ein passendes OpenSource-Projekt dafür integriert. Es nennt sich *git-lfs*. lfs steht hierbei für *Large file system*. Git LFS lagert grosse Dateien, die auf GitHub gepusht werden möchte, in einen Remote-Server von GitHub. Dabei spielt es keine Rolle welche Datei-Typen das. Es können auch ganze Ordner mit dem Git lfs genutzt werden. Im eigenen Repository wird dann an Stelle der grossen Datei ein Text-Pointer erstellt, der auf die ausgelagerte Datei auf dem Remote-Server verweist. Beim Klonen oder herunterladen des Repositorys wird die ausgelagerte Datei zusätzlich heruntergeladen und an die Stelle des Text-Pointers gesetzt.

Für die Installation und Verwendung bin ich der Anleitung der git lfs-Seite

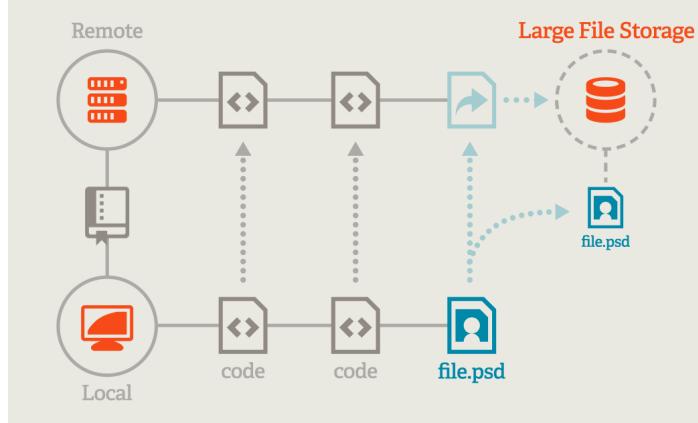


Abbildung 1: git lfs Ablauf

gefolgt (<https://git-lfs.github.com>). Hier habe ich als ersten Schritt das Paket heruntergeladen und bin dem beinhalteten README.txt gefolgt. Dieser nach zu folge habe ich das *git lfs*-Paket zuerst mit *Homebrew* auf meinem MacBook über den Terminal installiert. Danach musste ich das *git-lfs* noch mit dem Befehl *git lfs install* auf meinem MacBook aus dem Paket heraus installieren.

Ist das erledigt, habe ich mit dem Befehl *git lfs track *.framework* dem installierten git mitgeteilt, dass es jede framework-Datei, die mit .framework endet, mit *lfs* auf mein GitHub-Repository pushen soll. Das hat aber nicht gleich funktioniert. Danach habe ich festgestellt, dass die Datei opencv.framework gar keine Datei, sondern ein Ordner ist. So habe ich den Befehl für git lfs erweitert, dass git lfs mit dem Befehl *git lfs track opencv.framework/* den Ordner mit dem kompletten Inhalt auf das Repository pusht. Dies hat jedoch erneut nicht auf Anhieb funktioniert. Nach einer langen Suche in verschiedenen Foren bin ich dann auch die Lösung gestossen, die für mich funktionierte. Ich muss den Ordner mit dem Befehl *git lfs track opencv.framework/*** angeben. Damit konnte ich den Ordner erfolgreich auf mein GitHub-Repository laden.

Git lfs ist eine gute Lösung für das Problem mit zu grossen Dateien. Allerdings bringt git lfs viele Herausforderungen mit sich. So muss git lfs zuerst mitgeteilt werden, welche grosse Datei oder welcher grossen Ordner auf GitHub gepusht werden möchte. Und dies muss geschehen, bevor diese Datei oder der Order sich bereits auf dem GitHub-Repository befand oder sogar ein Push-Versuch gestartet wurde. Ist die Datei bereits ohne git lfs auf das Repository gepusht worden, ergibt das die Fehlermeldung, dass doch git lfs verwendet werden soll und der Vorgang bricht ab, wie bereits beschrieben. Nun steht diese Datei aber als Fehlgeschlagener Push in den Log-Dateien von

Git und kann damit nachträglich nicht mehr mit git lfs auf das Repository gepusht werden. Dafür müssen die Log-Dateien gelöscht werden. Dann kann ein neuer Versuch mit git lfs erfolgreich gestartet werden.

7 Festgestellte Probleme

7.1 Die richtige Perspektive

Die richtige Perspektive entscheidet über den Erfolg oder das Scheitern des Auslesens. Hier ist es also wichtig, den Winkel des Kassenzettels auf dem Bild so zu ändern, dass der Kassenzettel genau horizontal und vertikal ausgerichtet ist und soweit wie möglich parallel zum Handy gehalten wird, damit der komplette Kassenzettel auf dem Foto abgebildet wird.

Hier gibt es Funktionen, die diese Perspektive anpassen können. Diese Funktion ist bereits in sogenannten *Document-Scanner*-Apps enthalten. So können Dokumente auf dem Schreibtisch fotografiert werden, ohne gross auf die eben angesprochenen vertikalen und horizontalen Punkte zu achten.

Hierfür gibt es zwei Möglichkeiten zur Verwendung dieser Technik. Zum einen gibt es Software wie diese von OpenCV, bei der der Benutzer manuell die Eckpunkte auswählen muss, aus denen dann die Perspektive transformiert wird. Dies empfinde ich aber als nicht sehr Benutzerfreundlich für eine App. Denn schliesslich möchte man nur schnell den Kassenzettel fotografieren und dann sollte eigentlich alles weitere automatisch ablaufen, sodass dann gleich die Werte ausgelesen und auf der Benutzeroberfläche bereitgestellt werden. Hier möchte der Benutzer nicht noch lange Zeit für die Transformation der Perspektive aufwende.

Hier gibt es in der Zwischenzeit auch Software bzw. Frameworks, die die Anwahl der Ecken automatisch erkennt und damit die Perspektive automatisch nach gewissen Algorithmen berechnen und ausführen kann. Leider konnte dies in diesem Projekt nicht realisiert werden. Das automatische Erkennen und Ausführen dieser Transformation ist in der Regel nicht kostenlos und steht damit nicht zur Auswahl für dieses Projekt. Natürlich ist so etwas im Internet auch als OpenSource erhältlich. Die OpenSource-Perspective-Transformation-Frameworks die ich mir aber angeschaut habe, hatte die Transaktion bei Kassenzetteln nicht zuverlässig verarbeiten können.

Für eine produktive App ist die Transformation der Perspektive aber sicherlich ein wichtiger Punkt, so müssen Kassenzettel nicht komplett parallel zum Handy und auch nicht vertikal und horizontal ausgerichtet werden. Dies würde die Benutzerfreundlichkeit und auch die Produktivität der App steigern.

7.2 Nach einem Grosseinkauf

Für dieses Projekt habe ich von meiner Familie und Freunden viele verschiedene Kassenzettel sammeln lassen. Da waren auch sehr lange Kassenzettel dabei. Bei diesen Kassenzetteln bin ich auf ein Problem gestossen.

Die App ist mit Tesseract bisher so ausgelegt, dass ein Kassenzettel komplett auf einem Foto abgebildet wird. Wird aber ein langer Kassenzettel so fotografiert, dass dieser komplett auf ein Foto passt, ist die Schrift zu klein für Tesseract. Hier ist mir nach langen Überlegungen lediglich die Panorama-Funktion des iPhones eingefallen. Damit kann durch das Ziehen des iPhones ein Panoramabild erstellt werden. Auch Herr Tanner hatte diese Idee. Leider war nach den ersten direkten Tests mit der Panorama-Funktion nicht viel positives festzustellen. Um ein erfolgreiches Bild mittels der Panorama-Funktion erstellen zu können, benötigt es eine absolut ruhige Hand, damit das Handy komplett parallel zum Kassenzettel abgefahren werden kann. Ist dies nicht der Fall, gibt es im Panoramabild Verzerrungen, wie sich auch bei allen Tests zeigte. Das ist absolut nicht benutzerfreundlich und auch nicht produktiv.

Nach einer langen Suche im Internet bin ich auf die sogenannte *Stitching*-Funktion gestossen. Dies ist die passende Lösung für dieses Problem. Im folgenden Kapitel ist dies genauer beschrieben.

8 OpenCV und Stitching

8.1 OpenCV

Beim Stitching werden mehrere Fotos zu einem Panoramabild zusammen *gestickt*. Dafür müssen sich die Fotos in einem gewissen Teil überschneiden. In diesen Überschneidungen werden dann besondere Punkte, also spezielle Pixel-Kombinationen, in beiden Bildern in den überschneidenden Bereichen gesucht und mittels diesen Punkten zusammengefügt (im nachfolgenden Bild sind diese Punkte in beiden Bildern bereits ermittelt und durch grüne Striche dargestellt). An diesen Punkten werden die beiden Bilder danach *zusammenge näht*. Dies funktioniert auch bei langen Kassenzetteln.

Herr Tanner hat mir in einer Präsentation zu einem Zwischenstand, als ich gerade den Punkt der Bildaufbereitungsmöglichkeiten angesprochen hatte, zwei Bildbearbeitungs-Frameworks empfohlen. Eines davon war OpenCV. Da OpenCV bereits eine Stitching-Funktion enthält, habe ich mich für dieses Framework entschieden.

Da ich das Problem der langen Kassenzettel bisher als kritisch für die App gesehen hatte, wollte ich dieses unbedingt lösen. Die Lösung war hierfür dann

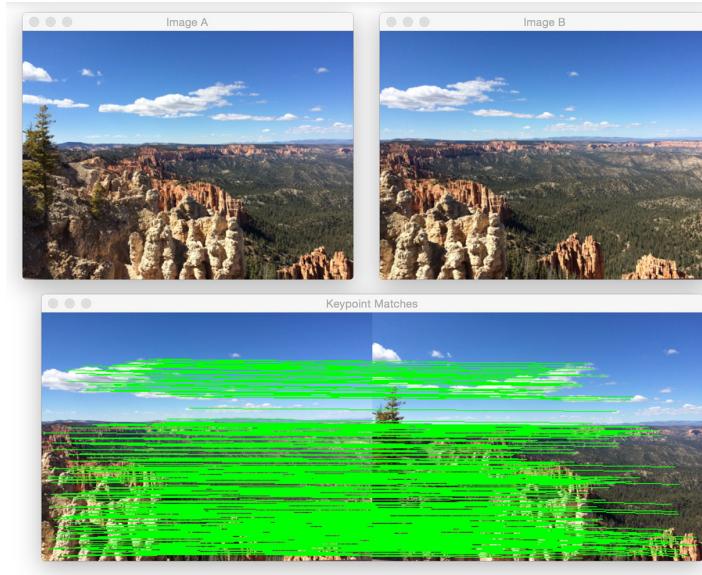


Abbildung 2: Stitching-Technik

die Stitching-Funktion von OpenCV. Dies habe ich wieder in einer Test-App eingebaut, die auch auf der Basis-App aufgebaut wird.

Auf GitHub ist ein Repository [foundry/OpenCVSwiftStitch¹](https://github.com/foundry/OpenCVSwiftStitch), in dem beschrieben wird, wie das Stitching mittels OpenCV in ein Swift-Projekt inkludiert werden kann. Dies startet mit der Installation von OpenCV mittels Cocoa Pods. Leider musste ich feststellen, dass dies nicht erfolgreich war. Wahrscheinlich war die OpenCV-Cocoa Pods-Datei defekt. So konnte ich das OpenCV nicht mittels Cocoa Pods in die Test-App einbinden. Daher habe ich es nach einer weiteren Anleitung im Internet manuell eingebunden².

Die Integration von OpenCV bringt allerdings eine gewisse Schwierigkeit mit sich. OpenCV ist in C++ geschrieben. Vor der Einführung von Swift war die Verwendung von OpenCV in einer iOS App, die mit Objective-C geschrieben war, kein Problem. Objective-C ist als Erweiterung zu der Programmiersprache C erstellt worden. Da auch C++ auf der Programmiersprache C basiert, konnte Objective-C mit C++ zusammen genutzt werden. Swift basiert zwar auf Objective-C, kann aber C++-Code nicht direkt verarbeiten. Dafür benötigt es eine gemischte Form von Objective-C und C++, dem sogenannten Objective-C++. Nun ist noch ein weiterer sogenannter *Bridging Header* nötig. Dieser vermittelt zwischen dem Swift-Teil und dem Objective-C++ Teil. Das folgende Bild veranschaulicht das nochmal, wobei auf diesem

¹<https://github.com/foundry/OpenCVSwiftStitch>

²<https://medium.com/@yiweini/opencv-with-swift-step-by-step-c3cc1d1ee5f1>

Bild der *Bridging Header* nicht abgebildet ist.

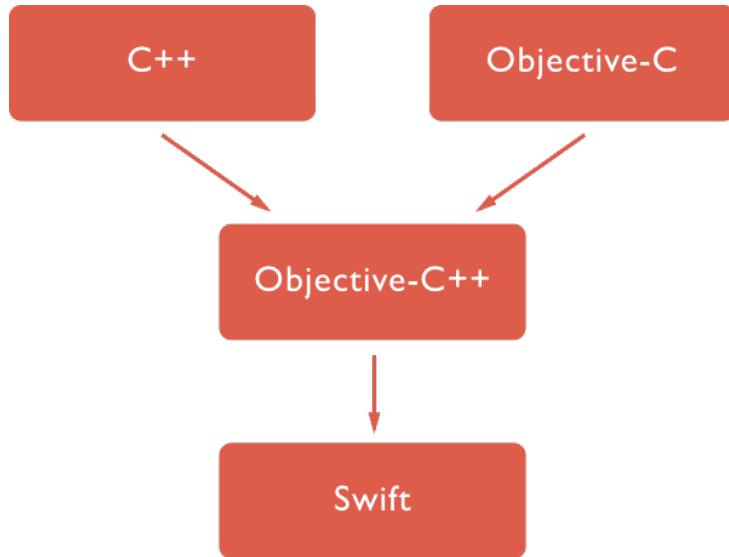


Abbildung 3: Aufbau zwischen Swift und OpenCV

Auf die Test-App bezogen bedeutet das nun, dass die OpenCVWrapper.h und OpenCVWrapper.mm die Objective-C++-Dateien sind. In diesen werden die OpenCV-Funktionen erstellt. Über den Bridging-Header können die OpenCV-Funktionen aus den OpenCVWrapper-Dateien in den Swift-Dateien verwendet werden.

Um zu testen, ob das OpenCV komplett verwendet werden kann, habe ich als Beispiel eine der OpenCV-Funktionen verwendet bzw. aufgebaut. Die Funktion soll ein farbiges Bild in ein Schwarz-Weiss-Bild umwandeln. Der Code im OpenCVWrapper sieht wie folgt aus:

```
1 +(UIImage *) makeGrayFromImage:(UIImage *) image{  
2  
3     // Create a variable from type cv::Mat  
4     cv::Mat imageMat;  
5     // Convert an UIImage to a cv::Mat  
6     UIImageToMat(image, imageMat);  
7     // Create new variable from type cv::Mat  
8     cv::Mat grayMat;  
9     // Convert the color from source imageMat  
10    // to target grayMat  
11    cv::cvtColor(imageMat, grayMat, CV_BGR2GRAY);  
12}
```

```

13 // Return the converted image as
14 return MatToUIImage(grayMat);
15 }

```

Diese Funktion kann nun über den Bridging Header im photoViewController.swift mit folgendem Code aufgerufen werden:

```

1 // The variable takenPhoto get the taken photo from
2 // the ViewController.swift
3 var takenPhoto:UIImage?
4 // The function makeGreyFromImage from OpenCVWrapper
5 // called with the taken Photo and set to the imageView
6 imageView.image = OpenCVWrapper
7 .makeGray(from: takenPhoto)

```

Diese Funktion kann nun verwendet werden. Damit ist OpenCV erfolgreich in die iOS-App integriert.

8.2 Stitching

Für die Implementierung der Stitching-Funktion kann nun in der Anleitung des GitHub-Repositorys von foundry/OpenCVSwiftStitch³, nach der Installation von OpenCV, fortgefahren werden. Um nun die Stitching-Funktion einwandfrei in die App einzubauen, habe ich die dafür nötigen sechs Dateien von foundry/OpenCVSwiftStitch in mein Projekt kopiert. Das sind in diesem Fall UIImage+OpenCV.h/-mm, UIImage+Rotate.h/-mm und stitching.h/-cpp. Zusätzlich musste noch folgende Funktion in das ViewController.swift eingebunden werden:

```

1 // original function name without parameter
2 // func stitch() {
3 // function name with additional parameter
4 func stitch(arrayParam:[ UIImage?]) {
5     DispatchQueue.global().async {
6
7     /*
8     let image1 = UIImage(named:"pano_19_16_mid.jpg")
9     let image2 = UIImage(named:"pano_19_20_mid.jpg")
10    let image3 = UIImage(named:"pano_19_22_mid.jpg")
11    let image4 = UIImage(named:"pano_19_25_mid.jpg")
12    let imageArray:[ UIImage?] = [image1,image2,image3]

```

³<https://github.com/foundry/OpenCVSwiftStitch>

```

13     , image4]
14 */
15
16 let imageArray :[ UIImage?] = arrayParam
17
18 let stitchedImage :UIImage = OpenCVWrapper. process
19   (with: imageArray as! [UIImage]) as UIImage
20
21 DispatchQueue.main.async {
22   NSLog("stichedImage %@", stitchedImage)
23   let imageView: UIImageView = UIImageView. init
24     (image: stitchedImage)
25   self. imageView = imageView
26   self. scrollView. addSubview( self .imageView!)
27   self. scrollView. backgroundColor = UIColor. black
28   self. scrollView. contentSize = self .imageView!
29     . bounds. size
30   self. scrollView. maximumZoomScale = 4.0
31   self. scrollView. minimumZoomScale = 0.5
32   self. scrollView. delegate = self as?
33     UIScrollViewDelegate
34   self. scrollView. contentOffset = CGPoint(x:
35     -(self .scrollView. bounds. size. width
36       - self .imageView!. bounds. size. width)/2.0 , y:
37     -(self .scrollView. bounds. size. height
38       - self .imageView!. bounds. size. height)/2.0)
39   NSLog(" scrollview \( self .scrollView. contentSize )")
40   // self .spinner. stopAnimating()
41 }
42 }
43 }
```

An diesem Punkt ist das Stitching erfolgreich in die App eingebaut und funktioniert mit den manuell eingebauten Bildern erfolgreich. Dafür wird ein UIImage-Array (imageArray) aus diesen Bildern erstellt, die dann in dieser Funktion zusammen gestickt werden. Für einen dynamischeren Test habe ich die Benutzeroberfläche und die Funktion stitch() noch etwas angepasst. Mein Ziel war es, dass der Funktion stitch() nicht manuell im Code Bilder übergeben werden, sondern dass diese Bilder von der Benutzeroberfläche ausgewählt, zu einem Array zusammengestellt werden können. Dieses Array soll dann als Parameter der Funktion stitch() übergeben werden können. So habe

ich im ViewController.swift ein globales UIImage Array erstellt.

```
1 var imageArrayGlobal : [ UIImage ?] = []
```

Die Funktion `stitch()` habe ich mit einem Parameter eines UIImage Arrays erweitert.

```
1 func stitch (arrayParam : [ UIImage ?]) {  
2     ....
```

Danach habe ich die Benutzeroberfläche geändert. Zu dem bereits bestehenden Button auf der Benutzeroberfläche aus der Basis-App, der ein Foto erstellt oder ein Foto aus der Bildergalerie öffnet, habe ich noch zwei weitere Buttons hinzugefügt. Der erste dieser zwei Buttons `imageAddToStitching` fügt das von Button `Photo` geöffnete Bild in das globale UIImage Array `imageArrayGlobal` hinzu. So können mit dem Button `Photo` mehrere Bilder ausgewählt werden und mit dem Button `imageAddToStitching` nach jedem ausgewählten Bild, dieses zu dem Array hinzugefügt werden.

Sind alle Bilder zu dem Array hinzugefügt worden, die zusammen ein Stitching-Bild ergeben sollen, übergibt der zweite neue Button `imageStitching` das globale Array mit den ausgewählten Bildern der Function als Parameter. So erstellt die `stitch()`-Funktion dann aus dem Bilder-Array vom Parameter das gestochte Bild.

Der Aufbau hatte direkt funktioniert. Ich hatte dann die in der `stitch()`-Funktion direkt im Code aufgerufenen Bilder in den iPhone-Simulator gespeichert und konnte diese so dynamisch über die Benutzeroberfläche in das Array hinzufügen und der angepassten `stitch()`-Funktion übergeben. Das ergab auch das gestochte Bild aus. So habe ich dann einen Test mit einem langen Kassenzettel gemacht. Von diesem habe ich vom oberen Teil des Kassenzettels ein Foto erstellt und vom unteren Teil ein Foto erstellt, so dass auf beiden Teilen ein Überschnitt der Mitte für das stitching vorhanden war. Leider hatte das nicht auf Anhieb funktioniert. Nach vielen und langen Tests und der genauen Auswertung der Fehlermeldung hat sich herausgestellt, dass das Stitching bei Kassenzetteln sehr viel überschnittenen Bereich benötigt. Anders als bei Bildern von einer Landschaft, bieten Bilder eines Kassenzettels zu wenige einzigartige Punkte bzw. Pixel-Zusammenstellungen, die für das Stitching benötigt werden.

9 Master App kassenzettel-management

Da ich mir mit dem Tesseract sicher war und das Stitching auch erfolgreich in einer Test-App eingebaut hatte, wollte ich nun eine Master-App erstellt, die auf der Basis-App aufbauend, die zwei Test-Apps von Tesseract und Stitching zusammenfügt. Auf dieser App wollte ich dann weiter aufbauen.

Dafür habe ich die Test-App von Tesseract als Basis für diese Master-App verwendet und nachträglich das OpenCV-Framework eingebaut und die erstellten bzw. geänderten Dateien aus dem Stitching-Kapitel in dieses Projekt kopiert. So musste ich lediglich die Benutzeroberfläche so anpassen, dass die der Tesseract-Test-App und die der Stitching-Test-App zusammen funktionieren.

Dieses manuelle Zusammenfügen der Test-Apps hätte ich mir mit einer konstanteren Verwendung von Git ersparen können. So hätte ich bereits die Basis-App als Master-App erstellen können und alle weiteren Test wie den SwiftOCR-, den Tesseract- und den Stitching-Test als Branch auf der Master-App im Git aufbauen können. So hätte ich im Nachhinein Git die Arbeit der Zusammenlegung der Basis-/Master-App mit einem Tesseract-Branch und danach mit einem Stitching-Branch überlassen können. Dies werde ich für die zweite Phase als grossen Lernfaktor mit einbeziehen und möchte dann die weitere Entwicklung genau so auf der nun erstellten Master-App kassenzettel-management aufbauen.

10 Bildaufbereitungs-Funktionen - OpenCV

Im Internet sind einige Seiten zu finden, die beschreiben, wie mit verschiedenen Bildaufbereitungs-Funktionen ein Bild aufbereitet werden kann, um die Ausleserate von Tesseract zu steigern.

Tesseract-OCR-iOS bietet selbst so eine Unterseite in seinem GitHub-Repository, in dem beschrieben ist wie Bilder bearbeitet werden können. Auch OpenCV ist für die Bildaufbereitung für Tesseract als geeignete Software erwähnt. Das passt natürlich, da OpenCV bereits wegen der Stitching-Funktion eingebaut ist.

Die Seite beschreibt unter anderem, dass es hilfreich ist, den Text bzw. das Bild so zu drehen, dass der auszulesende Text genau horizontal ausgerichtet ist (siehe Kapitel: Die richtige Perspektive). Des Weiteren ist die *Binarisation* ein sehr hilfreiche Bildbearbeitungs-Funktion. Diese Funktion erstellt aus dem Bild ein Schwarz-Weiss-Bild. Aus diesem Bild entfernt es dann die Graustufen indem jedes Bit entweder Weiss oder Schwarz geändert wird. Diese Auswahl wird nach gewissen Algorithmen berechnet. Das bringt allerdings

ein Risiko mit sich. Mit dieser Funktion werden Schatten auf einem Bild komplett in einen schwarzen Bereich verändert (siehe folgendes Bild)

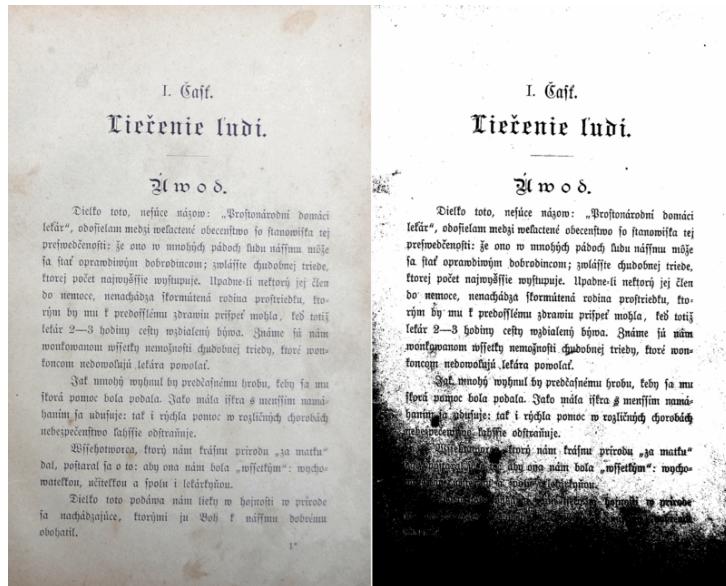


Abbildung 4: Binarisation mit Schatten

Allerdings gibt es eine erweiterte Funktion der Binarisation. Die sogenannte *Adaptive Gaussian Thresholding*. Bei dieser Funktion wird nicht nur jedes Pixel einzeln betrachtet und dann zu Schwarz oder Weiss geändert, sondern es werden auch noch die Pixel direkt nebenan miteinbezogen. Damit sind Schatten keine Herausforderung für diese Funktion und diese werden zuverlässig herausgearbeitet. Dies passt der Beschreibung nach perfekt zu den Kassenzettel-Abbildern. Dies hat sich gegen Ende des Projekts allerdings doch nicht als erfolgreiche Bildaufbereitungs-Funktion herausgestellt. Diese Funktion umrandet aber leider bei grosser bzw. dicker Schrift diese. Das erkennt Tesseract aber nicht mehr als Schrift, wodurch die *Adaptive Gaussian Thresholding*-Funktion nicht geeignet dafür ist (siehe nächstes Bild).

Leider ist mir das erst sehr spät aufgefallen. Ich hatte bereits einen Vergleichstest der verschiedenen Bildaufbereitungs-Funktionen erstellt. Jedoch hatte ich den Code nicht komplett an diese neu integrierten Funktionen angepasst. Dadurch hat sich der dieser Test in jedem Testfall immer noch auf das originale Bild bezogen. Dadurch ist dieser Test nicht brauchbar und leider war nicht mehr genug Zeit, diesen Test mit den richtigen Funktionen durchführen zu können.

Bio Golden Couscous	1	3.50	3.50	0
Salat Caprese Rucola	1	6.95	6.95	0
Pistazien ger.250g	1	4.35	4.35	0
<hr/>				
TOTAL CHF			14.80	
PostFinance			14.80	
<hr/>				
Buchung			PostFinance Card	

Abbildung 5: Adaptive Gaussian Thresholding

11 Weitere verwendete Software

In diesem Kapitel möchte ich gewisse Software und Programm genauer beschreiben, die ich während dem Projekt als Unterstützung verwendet habe.

11.1 Pods/CocoaPods

Unter anderem habe ich für die Installation von Tesseract das CocoaPods verwenden können. Cocoa Pods ist ein Dependency-Manager speziell für Swift und Objective-C und bietet mit über 47.000 Frameworks eine grosse Anzahl. Es macht die Installation von Frameworks super einfach. Mit wenigen Befehlen im Terminal auf dem Mac kann ein Framework-Pod in ein bestehendes Swift- oder Objective-C-Projekt eingebaut werden.

Leider habe ich Cocoa Pods erst beim testen von Tesseract kennenlernen dürfen. Es hat sich nämlich dann herausgestellt, dass SwiftOCR auch mittels Cocoa Pods installiert werden kann. Dies hätte mir eventuell einige Zeit bei der manuellen Installation von SwiftOCR ersparen können.

11.2 Homebrew

Mit Homebrew wird, ähnlich wie bei Cocoa Pods, die Installation von freien oder quelloffenen Software-Paket vereinfacht. Damit konnte ich ohne grosse Aufwände das git lfs über den Terminal installieren.

```

log Timo$ git lfs install
git: 'lfs' is not a git command. See 'git --help'.

Did you mean one of these?
  alias
  llg
  log
Timo$ brew install git-lfs
Updating Homebrew...
=> Downloading https://homebrew.bintray.com/bottles-portable-ruby/portable-ruby-2.3.3_2.leopard_64.bottle.tar.gz
#####
 100.0%
=> Pouring portable-ruby-2.3.3.2.leopard_64.bottle.tar.gz
=> Homebrew has enabled anonymous aggregate user behaviour analytics.
Read the analytics documentation (and how to opt-out) here:
  https://docs.brew.sh/Analytics

xcrun: error: invalid active developer path (/Library/Developer/CommandLineTools), missing xcrun at: /Library/Developer/CommandLineTools/usr/bin/xcrun
Error: Failure while executing: git config --local --replace-all homebrew.analyticsmessage true
=> Downloading https://homebrew.bintray.com/bottles/git-lfs-2.4.0.high_sierra.bottle.tar.gz
=> Downloading from https://akamai.bintray.com/3c/3c402c41266dd1290c8b87ff011913dc20cc0e2eb772159d14424e25518107f2?__gda__=exp=1
#####
 100.0%
=> Pouring git-lfs-2.4.0.high_sierra.bottle.tar.gz
=> Caveats
Update your git config to finish installation:

# Update global git config
$ git lfs install

# Update system git config
$ git lfs install --system
=> Summary
  /usr/local/Cellar/git-lfs/2.4.0: 65 files, 8.4MB
Timo$ git lfs install
Updated git hooks.
Git LFS initialized.

```

Abbildung 6: git lfs Installation mit HomeBrew

12 Mock-Up

Ein Teil dieses Projekts ist ein Mock-Up mit dem Auftraggeber erstellen. Das Ziel eines solchen Mock-Ups ist es, die Anforderungen der Benutzeroberfläche in Zusammenarbeit mit dem Auftraggeber und Anwender zu ermitteln.

Zur Vorbereitung auf die Erarbeitung mit dem Auftraggeber, habe ich zuvor verschiedene Apps aus dem App-Store verglichen. Heute ist eine App erst richtig Erfolgreich und bei Kunden beliebt, wenn das Design stimmig und modern ist. Des Weiteren ist eine simple Bedienung ebenfalls ein ausschlaggebender Punkt.

Daher habe ich bei dem Vergleich verschiedenster Apps zwei herausgesucht, die zwei verschiedene aber dennoch beliebte Design darstellen. Zum einen ist das die Messer-App Whatsapp. Diese stellt am unteren Ende der App eine Menü-Leiste bereit, über die verschiedene Bereiche der App erreichbar sind. Hier braucht es keine grosse Erklärung und der Anwender findet sich sofort zurecht.

Zum anderen ist da die Musik-Erkennungs-App Shazam. Diese keine Menü-Leiste, sondern drei Bereiche. Der mittlere Bereich ist der Hauptbereich, über den die hauptsächliche Funktion der Musik-Erkennung gestartet werden kann. Des Weiteren sind links und rechts davon jeweils ein Button, über die der linke bzw. rechte Bereich gewechselt werden kann. Das ist nicht so simpel aufgebaut wie bei Whatsapp, dennoch nach wenigen Versuchen der

Steuerung auch kein Problem mehr.

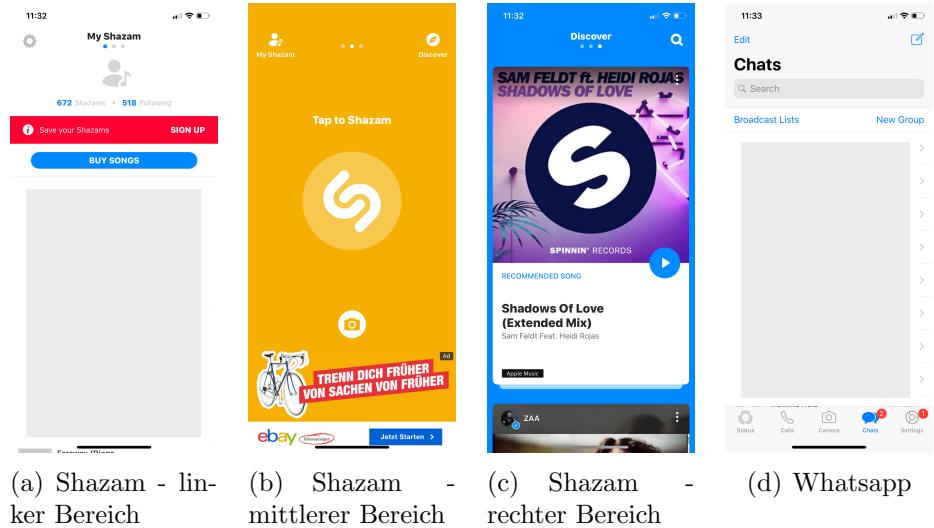


Abbildung 7: Shazam und Whatsapp

Dem Auftraggeber habe ich dann diese zwei Apps als Vorschläge für die kassenzettel-management-App vorgestellt. Der Auftraggeber hat sich gleich für die Variante von Whatsapp entschieden. Bei weiteren Gesprächen haben wir dann besprochen, dass wir ein Mix aus beiden vorgestellten Apps erstellen. So haben wir uns auf drei Bereiche geeinigt, ähnlich wie bei Shazam.

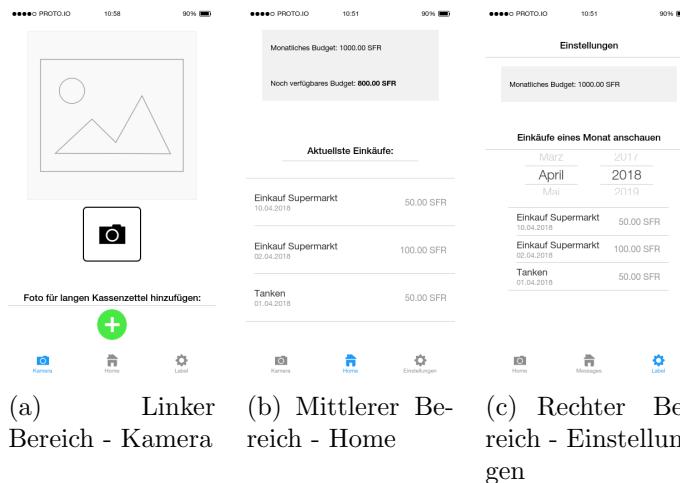


Abbildung 8: Mockup der kassenzettel-management-App

Diese drei Bereiche können über eine Menü-Leiste angewählt werden. Im

mittleren Bereich sieht man die Übersicht des gesetzten Budgets und die letzten Einkäufe, im linken Bereich kann ein neues Bild eines Kassenzettels erstellt werden und die eben erstellten Fotos zu der Stitching-Funktion hinzugefügt werden bei langen Kassenzetteln. Im rechten Bereich können Einstellungen getätigt werden wie z.B. das einstellen des monatlichen Budgets, von dem dann der ausgelesene Endbetrag eines Kassenzettels verrechnet wird. Es soll dort auch die Möglichkeit geben, bereits vergangene Monate einzusehen.

13 Aufgaben für die zweite Phase - Diplomarbeit

Da ich in dieser ersten Phase nicht so weit gekommen bin, wie zu Beginn des Projekts geplant hatte, habe ich das Projekt nun wie folgt aufgeteilt. In dieser ersten Phase habe ich die Bildaufbereitung erstellt und die OCR-Funktion eingebaut. Dazu wird der ausgelesene Betrag aus dem ausgelesenen Text herausgenommen. Somit soll in der zweiten Phase die Bereitstellung des ausgelesenen Betrags auf einer Benutzeroberfläche dargestellt werden. Die Benutzeroberfläche soll nach dem Mock-Up aufgebaut werden. Des Weiteren soll zwischen die Bereitstellung des ausgelesenen Endbetrags und der Benutzeroberfläche eine lokale Datenbank integriert werden. So können in dieser der Endbetrag und weitere wichtige Daten, wie der Erstell-Zeitpunkt des Kassenzettel-Abbilds, gespeichert werden. Dies könnte in einer späteren produktiven Version für eine zeitliche Analyse genutzt werden. Darüber hinaus soll noch eine Kategorisierung bereitgestellt werden, die der Benutzer manuell nach der Erstellung des Kassenzettel-Abbildes auswählen kann. Hierfür sollen wenige Grund-Kategorien wie *Lebensmittel*, *Klamotten*, *Tanken* zur Verfügung stehen.

Damit habe ich mir folgende Aufteilung der zweiten Phase vorgestellt. Die genaue Planung werde ich zu Beginn der zweiten Phase aufstellen.

- Datenbank-Aufbau innerhalb der App (mit Schema) (6 Stunden)
- Kategorien aufbauen (Benutzer muss nach dem Erstellen des Kassenzettel-Abbilds manuell eine passende Kategorie auswählen (da zum aktuellen Stand zu wenige Daten erfolgreich aus dem Kassenzettel gelesen werden können) (8 Stunden)
 - Ausgelesenen Text in die Datenbank pushen (4 Stunden)
 - Benutzeroberfläche nach dem Mockup (mit dem Auftraggeber erstellt) aufbauen (dafür eventuell externe Design-Software verwenden) (10 Stunden)
 - Daten aus der Datenbank und Funktionen (Photos machen und Stitching) mit der Benutzeroberfläche verbinden (8 Stunden)

- Projektmanagement (Scrum) (6 Stunden)
 - Dokumentation (30 Stunden inkl. Korrekturlesen, etc.)
 - Präsentation vorbereiten (6 Stunden)
- (Bei jedem Punkt ist die Zeit aufgerundet worden für eventuell anfallende Probleme)

14 Was ich gelernt habe

In diesem Projekt habe ich sehr viel lernen können. Zum einen hatte ich zuvor noch nicht so viel Kontakt mit der iOS-App Erstellung. Zum anderen habe ich viel über die OCR-Technik und Bildaufbereitungs-Funktionen, speziell auch mit OpenCV, kennengelernt. Auch hatte ich zuvor noch nicht so ausführlich Git genutzt. Auch aus dem fehlgeschlagene Projekt-Management werde ich für die nächste Phase, der Diplomarbeit, meine Schlüsse ziehen. Persönlich bin ich stolz auf das, was ich in diesem Projekt erreicht habe und freue mich bereits auf die zweite Phase.

15 Quellenverzeichnis

- Basis der App: <https://www.youtube.com/watch?v=4CbcMZ0SmEk>
- SwiftOCR: <https://github.com/garnele007/SwiftOCR>
- Tesseract: <https://de.wikipedia.org/wiki/Tesseract>
<https://github.com/gali8/Tesseract-OCR-iOS>
https://www.youtube.com/watch?v=DTQ1z_8KXZo
<https://github.com/gali8/Tesseract-OCR-iOS/wiki/Installation>
<https://github.com/tesseract-ocr/tesseract/wiki/ImproveQuality#page-segmentation-method>
- Git LFS: <https://git-lfs.github.com/>
- Die richtige Perspektive: https://docs.opencv.org/3.1.0/da/d6e/tutorial_py_geometric_transformations.html
<https://github.com/Breta01/handwriting-ocr/blob/master/PageDetection.ipynb>
https://docs.opencv.org/3.1.0/da/d6e/tutorial_py_geometric_transformations.html
- OpenCV und Stitching: <https://www.pyimagesearch.com/2016/01/11/opencv-panorama-stitching/> <https://medium.com/@yiweini/opencv-with-swift-step-1-ee3e3a5735b>
<https://medium.com/@borisohayon/ios-opencv-and-swift-1-ee3e3a5735b>
https://docs.opencv.org/trunk/d8/d19/tutorial_stitcher.html https://docs.opencv.org/3.1.0/d7/d4d/tutorial_py_thresholding.html

```
//docs.opencv.org/3.1.0/d4/d13/tutorial_py_filtering.html https:  
//docs.opencv.org/master/d3/def/tutorial_image_manipulation.html  
Bildaufbereitung: https://github.com/tesseract-ocr/tesseract/wiki/  
ImproveQuality#page-segmentation-method  
https://www.docs.opencv.org/master/d7/d4d/tutorial_py_thresholding.  
html  
Homebrew: https://brew.sh/  
https://de.wikipedia.org/wiki/Homebrew_(Paketverwaltung)  
Mock-Up: https://proto.io
```

16 Bilderverzeichnis

Abbildung 1: <https://git-lfs.github.com/>
Abbildung 2: <https://www.pyimagesearch.com/2016/01/11/opencv-panorama-stitching/>
Abbildung 3: <https://medium.com/@borisohayon/ios-opencv-and-swift-1ee3e3a5735b>
Abbildung 4: <https://github.com/tesseract-ocr/tesseract/wiki/ImproveQuality#page-segmentation-method>
Abbildung 5: Selbst gemacht
Abbildung 6: Selbst gemacht
Abbildung 7: Selbst gemacht
Abbildung 8: Selbst gemacht (<https://proto.io>)