VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

MOBILNÍ APLIKACE PRO ROZPOZNÁNÍ LEUKOKORIE ZE SNÍMKU LIDSKÉHO OBLIČEJE

MOBILE APP FOR RECOGNITION OF LEUKOCORIA IN AN IMAGE OF HUMAN FACE

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

AUTOR PRÁCE **Bc. PAVEL HŘEBÍČEK**
AUTHOR

VEDOUCÍ PRÁCE **prof. Ing. ADAM HEROUT, Ph.D.**
SUPERVISOR

BRNO 2019

Ústav počítačové grafiky a multimédií (UPGM) Akademický rok 2018/2019

Zadání diplomové práce

Student: **Hřebíček Pavel, Bc.**
Program: Informační technologie Obor: Počítačová grafika a multimédia
Název: **Mobilní aplikace pro rozpoznání leukokorie ze snímku lidského obličeje**
Mobile App for Recognition of Leukocoria in an Image of Human Face
Kategorie: Zpracování obrazu
Zadání:

1. Prostudujte možnosti diagnostiky leukokorie z fotografií lidské tváře.
2. Vyhledejte a prostudujte dostupné nástroje a knihovny pro rozpoznání částí lidské tváře.
3. Opatřete dostupná data - fotografie lidí s leukokorií.
4. Experimentujte s možnostmi pořizování fotografií lidského obličeje tak, aby z nich bylo možné usuzovat na přítomnost leukokorie.
5. Vytvořte nástroj pro pořizování vhodné datové sady.
6. Iterativně vyviňte algoritmy pro zpracování lidských tváří v obraze a pro rozpoznání leukokorie.
7. Navrhněte mobilní aplikaci, která umožní laickému uživateli snadno zjistit případnou přítomnost leukokorie z fotografie lidského obličeje.
8. Zhodnoťte dosažené výsledky a navrhněte možnosti pokračování projektu; vytvořte plakátek a krátké video pro prezentování projektu.

Literatura:

- Gary Bradski, Adrian Kaehler: Learning OpenCV; Computer Vision with the OpenCV Library, O'Reilly Media, 2008
- Richard Szeliski: Computer Vision: Algorithms and Applications, Springer, 2011
- Steve Krug: Don't Make Me Think, Revisited: A Common Sense Approach to Web Usability, ISBN-13: 978-0321965516
- Davis E. King. Dlib-ml: A Machine Learning Toolkit. Journal of Machine Learning Research 10, pp. 1755-1758, 2009

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 5, značné rozpracování bodů 6 a 7.

Podrobné závazné pokyny pro vypracování práce viz http://www.fit.vutbr.cz/info/szz/

Vedoucí práce: **Herout Adam, prof. Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, doc. Dr. Ing.
Datum zadání: 1. listopadu 2018
Datum odevzdání: 22. května 2019
Datum schválení: 5. prosince 2018

Abstrakt
Cílem této práce je návrh a implementace multiplatformní multijazyčné mobilní aplikace pro rozpoznání leukokorie ze snímku lidského obličeje pro platformy iOS a Android. Leukokorie je bělavý svit zornice, který se při použití blesku může na fotografii objevit. Včasnou detekcí tohoto symptomu lze zachránit zrak člověka. Samotná aplikace umožňuje analyzovat fotografii uživatele a detekovat přítomnost leukokorie. Cílem aplikace je tedy analýza očí člověka, od čehož je také odvozen název mobilní aplikace – Eye Check. K vytvoření multiplatformní mobilní aplikace byl použit framework React Native. Pro detekci lidského obličeje a očí byla zvolena knihovna Dlib, pro práci s fotografií pak knihovna OpenCV. Ke klasifikaci očí na případný výskyt leukokorie byla použita konvoluční neuronová síť. Komunikace mezi klientem a serverem je řešena pomocí architektury REST. Výsledkem je mobilní aplikace, která v případě detekce leukokorie uživatele upozorní, že by měl navštívit svého lékaře.

Abstract
The goal of this thesis is to design and implement a multiplatform multilingual mobile application for detecting leukocoria in an image of human face for iOS and Android platforms. Leukocoria is a whitish light of the pupil, which can be seen on the photo when the flash is used. Early detection of this symptom can save human eyesight. The application itself allows to analyze a user's photo and detect the presence of leukocoria. The goal of the application is to analyze eyes of the human, from which the mobile application name – Eye Check is derived. React Native framework was used to create a multiplatform mobile application. The Dlib library was chosen for human face and eye detection, the OpenCV library for working with the photo. The convolutional neural network was used to classify the eyes for the possible presence of leukocoria. Client-Server communication is solved using the REST architecture. The result is a mobile application that detects leukocoria and allerts the user to visit his doctor if leukocoria is detected.

Klíčová slova
Mobilní aplikace, Eye Check, Leukokorie, iOS, Android, React Native, Dlib, OpenCV, REST, Django REST framework, Konvoluční neuronová síť, Keras, Tensorflow

Keywords
Mobile application, Eye Check, Leukocoria, iOS, Android, React Native, Dlib, OpenCV, REST, Django REST framework, Convolutional neural network, Keras, Tensorflow

Citace
HŘEBÍČEK, Pavel. *Mobilní aplikace pro rozpoznání leukokorie ze snímku lidského obličeje.* Brno, 2019. Diplomová práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce prof. Ing. Adam Herout, Ph.D.

Mobilní aplikace pro rozpoznání leukokorie ze snímku lidského obličeje

Prohlášení

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně pod vedením pana prof. Ing. Adama Herouta, Ph.D. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

....
Pavel Hřebíček
19. května 2019

Poděkování

Rád bych poděkoval svému vedoucímu práce panu prof. Ing. Adamu Heroutovi, Ph.D. za odborné vedení, čas, ochotu a cenné rady při tvorbě této práce. Dále bych zde chtěl poděkovat MUDr. Barboře Žajdlíkové, za její čas, spolupráci a za poskytnutí odborné konzultace v oblasti očního lékařství, bez které by byla celá tato práce jen těžko realizovatelná. V neposlední řadě bych zde rád poděkoval své rodině a přítelkyni za jejich velkou a nenahraditelnou podporu a všem testerům, kteří se podíleli na testování aplikace a poskytli mi zpětnou vazbu.

Obsah

1 Úvod **3**

2 Leukokorie, její příčiny a analýza současné mobilní aplikace zaměřené na detekci leukokorie **4**
2.1 Leukokorie a její příčiny 4
2.2 CRADLE White Eye Detector 6

3 Funkční a nefunkční požadavky na mobilní aplikaci **9**
3.1 Funkční požadavky 9
3.2 Nefunkční požadavky 10

4 Možnosti detekce objektů, obličeje a očí z fotografie **11**
4.1 Detekce objektů v obraze 12
4.2 Detekce obličeje a očí z fotografie 18

5 Metody pro rozpoznání leukokorie a implementace nástroje pro pořizování vhodné datové sady **23**
5.1 Metoda prohledávání pixelů ve fotografii oka 24
5.2 Použití konvoluční neuronové sítě pro detekci leukokorie 27
5.3 Nástroj pro pořizování vhodné datové sady 29

6 Návrh mobilní aplikace **34**
6.1 Návrh uživatelského rozhraní 34
6.2 Testování a vyhodnocení uživatelského rozhraní 38
6.3 Návrh architektury mobilní aplikace 39

7 Implementace mobilní aplikace **41**
7.1 Možnosti vývoje multiplatformní mobilní aplikace 41
7.2 React Native 42
7.3 Redux 46
7.4 Eslint 47
7.5 Implementace klientské části 48
7.6 Implementace serverové části 54
7.7 Klient-Server komunikace 58
7.8 Dostupnost 59

8 Testování mobilní aplikace **60**

9 Rozšíření práce o webovou aplikaci **62**

9.1 Klientská část 62
9.2 Administrační část 63

10 Závěr **64**

Literatura **65**

A Diagram případů užití **68**

B Plakát **69**

C Obsah přiloženého DVD **70**

D Instalační manuál mobilní aplikace Eye Check **71**

E Instalační manuál webové aplikace **73**

F Instalační manuál nástroje pro pořizování vhodné datové sady **74**

G Instalační manuál, ukázky trénování a ověření úspěšnosti modelu konvoluční neuronové sítě **75**

Kapitola 1

Úvod

Cílem této diplomové práce je navrhnout a implementovat multiplatformní multijazyčnou mobilní aplikaci pro detekci leukokorie ze snímku lidského obličeje. Leukokorie se může projevit na fotografii, která byla pořízena za zhoršených světelných podmínek. Místo klasického tzv. *efektu červených očí* se může projevit bělavý svit, který znamená, že oko člověka není zdravé a je nutná okamžitá návštěva lékaře. Včasnou diagnostikou lze předcházet očním onemocněním a zachránit tak zrak člověka. Právě tento fakt mě vedl k vytvoření této práce.

Důležitou součástí celé této práce bylo vyhledání odborníka z oboru očního lékařství. Podařilo se mi kontaktovat MUDr. Barboru Žajdlíkovou, která pracuje ve Fakultní nemocnici v Brně na dětském oddělení a specializuje se přímo na oční lékařství. Díky této spolupráci jsem se dozvěděl veškeré informace o leukokorii, které vedly ke zdokonalení celého díla.

Pro vytvoření multiplatformní aplikace byl zvolen framework *React Native*, který umožňuje vytvářet multiplatformní mobilní aplikace pro platformy *iOS* a *Android*. Pro detekci lidského obličeje a očí byla zvolena knihovna *Dlib*, pro práci s fotografií pak knihovna *OpenCV*. Ke klasifikaci očí na případný výskyt leukokorie byla použita *konvoluční neuronová síť*. *Konvoluční neuronová sít* byla vytvořena pomocí knihovny *Keras* běžící na backendu *Tensorflow*. Komunikaci mezi klientskou a serverovou části zajišťuje architektura `REST`.

Kapitola 2 se věnuje popisu leukokorie a onemocněním, které jsou příčinou leukokorie. Dále je zde analyzována již existující mobilní aplikace sloužící k detekci leukokorie. Kapitola 3 definuje funkční a nefunkční požadavky na mobilní aplikaci. Kapitola 4 popisuje, analyzuje a porovnává metody, které se používají k detekci objektů, obličeje a očí z fotografie. Kapitola 5 se věnuje metodám pro detekci leukokorie, které byly experimentálně vyzkoušeny a následně vyhodnoceny. Dále je zde popsán nástroj, který byl vytvořen za účelem pořizování vhodné datové sady. Kapitola 6 je zaměřena na návrh uživatelského rozhraní a architektury mobilní aplikace. Uživatelské rozhraní je zde navrženo, otestováno a vyhodnoceno. Kapitola 7 se zabývá samotnou implementací mobilní aplikace. Jsou zde shrnuty a porovnány možnosti vývoje multiplatformní mobilní aplikace a je zde zdůvodněno, proč byl pro tuto práci zvolen framework *React Native*. Je zde popsána klientská i serverová část mobilní aplikace a jsou zde shrnuty a popsány všechny nástroje, které byly k implementaci mobilní aplikace použity. Kapitola 8 popisuje způsoby testování aplikace, nedostatky, které byly při testování nalezeny a úpravy, které byly provedeny k odstranění těchto nedostatků. Kapitola 9 popisuje responzivní multijazyčnou webovou aplikaci, která byla vytvořena jako rozšíření této práce. Kapitola 10 shrnuje dosažené výsledky a obsahuje návrh možných vylepšení do budoucna.

Kapitola 2

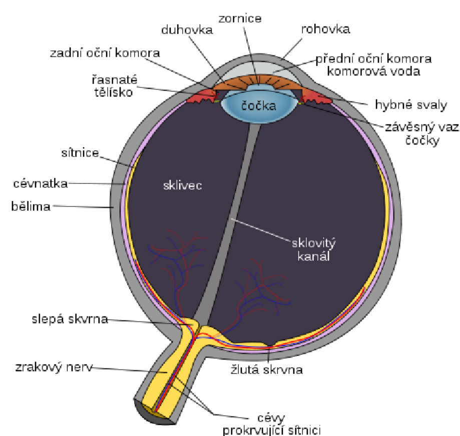
Leukokorie, její příčiny a analýza současné mobilní aplikace zaměřené na detekci leukokorie

První část této kapitoly čtenáře seznamuje s jevem, který se nazývá leukokorie a s onemocněními, které jsou příčinou leukokorie. Druhá část této kapitoly je věnována již dostupné mobilní aplikaci pro detekci leukokorie, která se nazývá *CRADLE White Eye Detector*.

2.1 Leukokorie a její příčiny

Leukokorie, nebo-li bělavý svit zornice, je vidět na obrázku 2.1. Představuje jeden z nejzávažnějších příznaků nitrooční patologie. Leukokorie znamená pouze symptom, nikoliv diagnózu [33]. Na obrázku 2.2 lze vidět schéma lidského oka. Pokud je fotografie pořízena ve tmě nebo za zhoršených světelných podmínek a je použit blesk, dochází k tzv. *efektu červených očí*. Tento jev vzniká tak, že paprsek světla projde zornicí a odrazí se od sítnice, která je prokrvená. Existuje-li však nějaká překážka mezi sítnicí a předním segmentem, vzniká bělavý svit zornice nebo-li leukokorie [42].

Obrázek 2.1: Bělavý svit zornice, který představuje symptom leukokorie. Převzato z [34].



Obrázek 2.2: Schéma lidského oka. Převzato z [20].

Mezi nejčastější příčiny leukokorie patří kongenitální katarakta, retinoblastom, retinopatie nedonošených (V. stádium), perzistující hyperplastický primární sklivec (PHPV). Mezi méně časté příčiny patří infekční onemocnění – toxoplazmóza a toxokaróza, kolobomy sítnice a cévnatky, Coatsova choroba.

Kongenitální katarakta

Je zkalení čočky, které může být jednostranné nebo oboustranné. Jako nejčastější příčina leukokorie je buď hereditární bez jiných anomálií, nebo je součástí syndromů (nejčastěji Downův syndrom) a metabolických vad (nejčastěji galaktosemie), nebo vzniká následkem intrauterinní infekce. Pro správný vývoj zrakových funkcí je nutné co nejdříve zahájit léčbu, kterou je v tomto případě chirurgický zákrok. Dochází totiž k útlumu zrakových vjemů v oku, které je postiženo a následkem toho může dojít až ke vzniku amblyopie a poruše binokulárních funkcí [21, 42].

Retinoblastom

Je maligním nádorem sítnice, patří mezi nejčastěji se vyskytující nádor oka v dětském věku. Leukokorie je jeho nejčastějším prvotním příznakem. Může se projevit i šilháním, nitroočním zánětem s glaukomem, může vést až ke ztrátě vidění. Léčba probíhá komplexně ve spolupráci s onkology a její možnosti jsou chemoterapie, lokální terapie, radioterapie či enukleace. Díky včasné diagnostice je retinoblastom nyní nejúspěšněji léčitelný maligní nádor v dětství. Pokud je nádor zachycen v bezpečně intraokulárním stádiu, je jeho vitální prognóza excelentní [42].

Retinopatie nedonošených

Ohrožuje zejména těžce nedonošené novorozence. Leukokorie se zde projevuje u pozdějších stádií onemocnění, při kterých dochází k odchlípení sítnice [42].

Persistující primární hyperplastický sklivec (PHPV)

Je kongenitální nedědičná anomálie, nález je typicky jednostranný, většinou spojen s mikroftalmem. Onemocnění může být doprovázeno kataraktou nebo odchlípením sítnice. Chirurgické odstranění sklivcových zákalů pomáhá ke zlepšení zraku [42].

Toxokaróza

Je v našich podmínkách způsobená parazity *Toxocara cati* (škrkavka kočičí) a *Toxocara canis* (škrkavka psí), svůj celoživotní cyklus prodělává v zažívacím traktu psů a koček. Z hlediska klinického se rozlišuje forma toxokarózy viscerální, oční nebo smíšená. Nakazit se dítě může kontaminovanou potravou nebo vajíčky ze zvířecích exkrementů v půdě nebo písku. Léčba je chirurgická a bývá doplněna léčbou systémovou, která spočívá v kombinaci kortikosteroidů s antihelmintikem, které snižuje pohyblivost larvy [42].

Oční toxoplazmóza

Je nejčastější formou zadních uveitid. Jedná se o infekční onemocnění způsobené parazitickým prvokem *Toxoplasma gondii*. Přenos na člověka je možný při nedodržení hygienických zásad. Rizikem je primární infekce matky v graviditě, kdy může přestoupit infekce na plod

a může tedy dojít k předčasnému porodu nebo potratu. Záleží na období gravidity, ve kterém matka onemocněla (čím dříve onemocněla, tím bude poškození plodu větší). Mezi subjektivní potíže pacienta patří výrazný pokles zrakové ostrosti (makulární ložisko), zamlžení obrazu a plovoucí zákalky [42].

Coatsova choroba

Je nedědičná vrozená anomálie sítnicových kapilár. Postižení je v 90 % jednostranné, vzácně bývá oboustranné. Manifestuje se nejčastěji mezi šestým až patnáctým rokem života, postihuje obvykle chlapce. Léčba spočívá v odstranění abnormálních cév pomocí kryoterapie [42].

2.2 CRADLE White Eye Detector

Před samotným návrhem a implementací mobilní aplikace jsem provedl průzkum trhu, při kterém jsem hledal aplikace zaměřené na detekci leukokorie, popřípadě jestli nějaká aplikace vůbec již existuje. Jedinou aplikaci, kterou jsem objevil, byla aplikace *CRADLE White Eye Detector*[1], kterou vytvořili profesoři Bryan Shaw a Greg Hamerly z univerzity Baylor. Mobilní aplikace je dostupná jak pro operační systém *iOS*[2], tak *Android*[3] a lze ji vidět na obrázku 2.3.

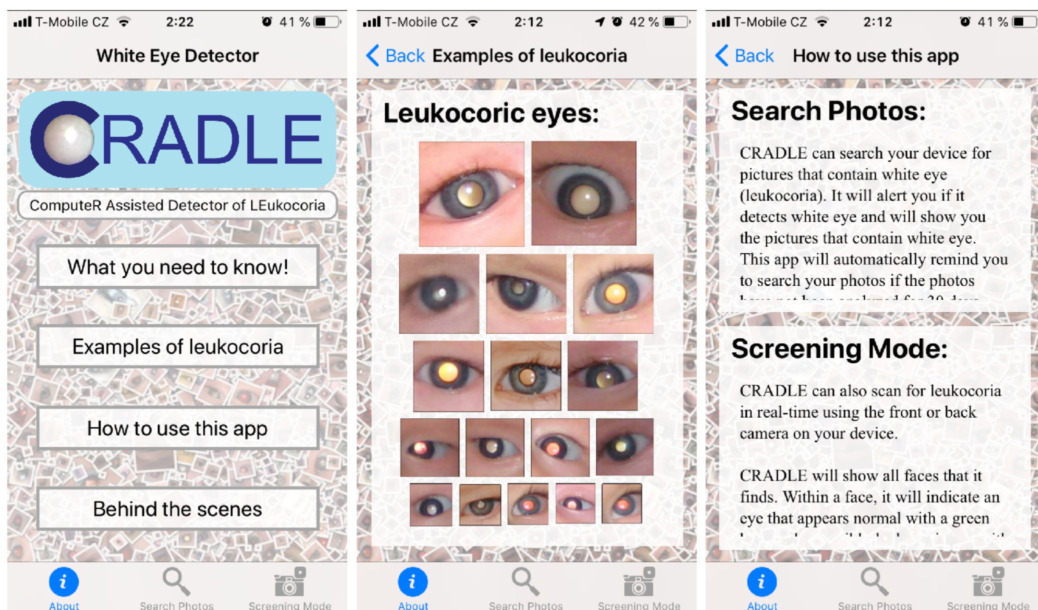
Aplikaci jsem si stáhl a nainstaloval na zařízení s oběma podporovanými operačními systémy. Následně jsem aplikaci pozoroval a testoval, jak po stránce uživatelského rozhraní, tak po stránce funkcionality a ovládání aplikace. Pro lepší přehled čtenáře jsem všechny informace, klady, zápory a nasbírané poznatky o aplikaci *CRADLE White Eye Detector* shrnul do tabulky 2.1.

Oproti již existující mobilní aplikaci jsem se v této práci soustředil především na návrh a vytvoření uživatelsky přívětivějšího a graficky modernějšího uživatelského rozhraní. Další prioritou této práce bylo poskytnout aplikaci v českém jazyce pro české uživatele, a dostat tak leukokorii do většího povědomí lidí. V neposlední řadě jsem se snažil o přesnější detekci případné leukokorie a dosáhnutí co nejmenšího počtu falešně pozitivních (false positive) výsledků.

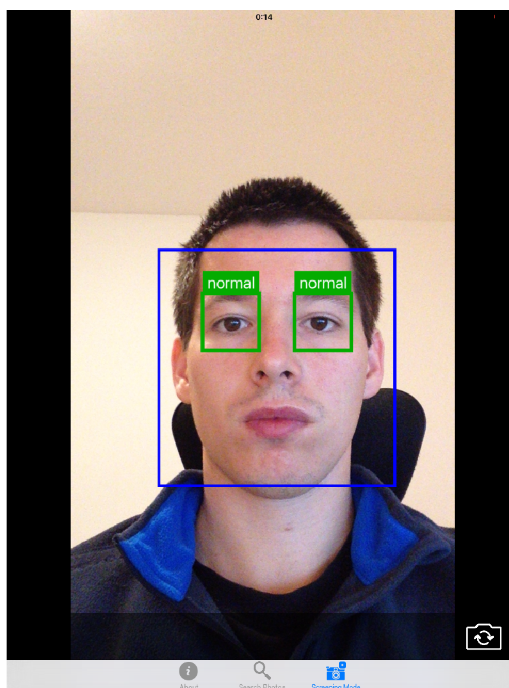
Funkcionalita

Samotná aplikace dokáže analyzovat případný výskyt leukokorie jak z jedné, uživatelem vybrané fotografie, tak z celé galerie fotografií v mobilním zařízení. Velmi povedeným a užitečným je tzv. „*Screening Mode*“, který je ukázán na obrázku 2.4, a který dokáže analyzovat případný výskyt leukokorie v reálném čase, bez jakéhokoliv dlouhého čekání na výsledek. Výsledek analýzy se uživatel dozví buď formou textového dialogu (v případě výběru fotografie nebo fotografií z galerie mobilního zařízení) nebo formou grafického výstupu (v případě použití již zmíněného *Screening Mode*), jak lze vidět na obrázku 2.4.

[1] `http://cs.baylor.edu/~hamerly/leuko/`
[2] `https://itunes.apple.com/us/app/white-eye-detector/id904042354?mt=8`
[3] `https://play.google.com/store/apps/details?id=net.leuko.leuko_android`



Obrázek 2.3: Ukázka uživatelského rozhraní mobilní aplikace *CRADLE White Eye Detector* na mobilním zařízení iPhone 6S s operačním systémem *iOS* 12.4.1.



Obrázek 2.4: Ukázka tzv. „*Screening Mode*“ funkce v aplikaci *CRADLE White Eye Detector* na iPadu mini s operačním systémem *iOS* 9.3.5, která umožňuje analyzovat případný výskyt leukokorie v reálném čase.

CRADLE White Eye Detector	**iOS**	**Android**
Dostupnost*	Ano	Ano
Cena*	Zdarma	Zdarma
Hodnocení*	3.1/5 (15 hodnocení)	3.7/5 (143 hodnocení)
Aktuální verze*	1.1	1.4
Poslední modifikace*	24. 1. 2015	19. 12. 2018
Požadovaná verze OS*	7.0 a vyšší	4.1 a vyšší
Velikost*	6 MB	9,1 MB
+	• Aplikace obsahuje ukázky očí, na kterých se projevila leukokorie. • Funkce *Screening Mode*, která dokáže analyzovat fotografii v reálném čase. • Možnost detekce leukokorie z vybrané fotografie nebo z celé galerie.	
-	• Texty v aplikaci, které slouží k vysvětlení, jak celá aplikace funguje, jsou příliš dlouhé.	
Testováno na	• iPad mini – *iOS* 9.3.5 • iPhone 5S – *iOS* 12.1.1 • iPhone 6S – *iOS* 12.4.1	• Sony Xperia Z1 Compact – *Android* 5.1.1 • Sony Xperia X – *Android* 8.0.0
Uživatelské rozhraní	Působí zastarale	Moderní
Ovládání aplikace	Intuitivní	
Funkcionalita	• iPhone 5S, iPhone 6S – Pád aplikace ve funkci „*Screening Mode*“, velmi často nefunguje ani fotoaparát. • iPhone 6S – Zamítnutý přístup ke galerii bez jakéhokoliv předešlého vyzvání k udělení povolení.	• Sony Xperia Z1 Compact – Pád aplikace ihned po spuštění.
Multijazyčnost	• Angličtina • Španělština *Pozn.:* Pouze na základě nastavení jazyka v telefonu – nelze měnit v aplikaci.	

* Údaje jsou platné k 6. 5. 2019

Tabulka 2.1: Shrnutí informací a poznatků o aplikaci *CRADLE White Eye Detector*, které byly nasbírány při testování aplikace.

Kapitola 3

Funkční a nefunkční požadavky na mobilní aplikaci

Důležitým faktorem před samotným návrhem a implementací mobilní aplikace je definice požadavků. Z tohoto důvodu jsou v této kapitole definovány funkční a nefunkční požadavky na mobilní aplikaci Eye Check.

Illa Neustadt a Jim Arlow ve své knize [26] rozdělují požadavky na funkční a nefunkční. Rozdělení na funkční a nefunkční požadavky patří mezi základní a nejjednodušší dělení. Funkční požadavky určují jaké chování bude systém nabízet, nefunkční požadavky specifikují omezení kladená na konkrétní systém.

3.1 Funkční požadavky

V následujících odstavcích jsem shrnul funkční požadavky na mobilní aplikaci Eye Check, která je součástí této práce.

- Uživatel bude moci pořídit fotografii obličeje přímo v mobilní aplikaci.
- Uživatel bude moci vybrat fotografii obličeje z galerie fotografií v mobilním telefonu.
- Uživatel bude moci analyzovat fotografii na výskyt leukokorie.
- Uživatel bude moci provést výběr jazyka přímo v mobilní aplikaci – aplikace bude vícejazyčná. Podporované jazyky budou čeština a angličtina. Prvotní nastavení jazyka bude záležet na nastavení jazyka v operačním systému mobilního zařízení.
- Uživatel bude moci přejít na webové stránky projektu.
- Uživatel si bude moci zobrazit aktuální verzi aplikace.
- Uživatel si bude moci zobrazit výčet a stručné informace o příčinách leukokorie.
- Uživatel bude moci zobrazit ukázky očí, na kterých se projevila leukokorie.

Diagram případů užití

Funkční požadavky se zobrazují pomocí diagramu případů užití [26]. Mezi hlavní prvky diagramu případů užití patří:

- Subjekt (Hranice systému) – Hranice, která určuje, co je součástí systému a co naopak není.
- Aktér – Vyjadřuje roli (např. Zákazník), která bezprostředně používá daný systém.
- Případ užití – Popisuje funkce, které systém nabízí.

Diagram případů užití pro mobilní aplikaci Eye Check, jejíž návrh a implementace je součástí této práce, je zobrazen v příloze A na obrázku A.1.

3.2 Nefunkční požadavky

- Mobilní aplikace bude spustitelná na operačních systémech *iOS* a *Android* – multiplatformní.
- Jednoduché a intuitivní uživatelské rozhraní.
- Moderní grafické zpracování.

Kapitola 4

Možnosti detekce objektů, obličeje a očí z fotografie

Proto, aby bylo možné uživatele aplikace informovat, zda-li byla nalezena ve snímku lidského obličeje leukokorie nebo ne, je potřeba z fotografie detekovat obličej a oči člověka. Před samotnou detekcí těchto oblastí je nutné čtenáře seznámit se samotnými principy, které se používají pro detekci objektů. Z tohoto důvodu je první část této kapitoly věnována právě metodám, které se pro detekci objektů využívají.

Druhá část této kapitoly se zabývá možnostmi detekce lidského obličeje a očí. Vysvětluje, proč jsem k detekci a pro práci s fotografiemi využil knihovny *Dlib* [18] a *OpenCV* [4], resp. rozhraní těchto knihoven, které je dostupné pro programovací jazyk *Python*.

OpenCV

OpenCV (Open Source Computer Vision Library) je knihovna pro manipulaci s obrazem, která je vydána pod licencí BSD. Knihovna poskytuje rozhraní pro programovací jazyky *C++*, *Python*, *Java* a byla navržena pro výpočetní efektivitu se silným zaměřením na aplikace v reálném čase [4]. K implementaci nástroje pro rozpoznání leukokorie jsem použil rozhraní pro programovací jazyk *Python* – *opencv-python*[1].

Dlib

Dlib je knihovna obsahující algoritmy pro strojové učení a nástroje pro vytváření komplexního softwaru k řešení problémů reálného světa. Knihovna poskytuje rozhraní pro programovací jazyky *C++* a *Python*. K implementaci nástroje pro rozpoznání leukokorie jsem použil rozhraní pro programovací jazyk *Python* – *dlib*[2].

[1] https://pypi.org/project/opencv-python/
[2] https://pypi.org/project/dlib/

4.1 Detekce objektů v obraze

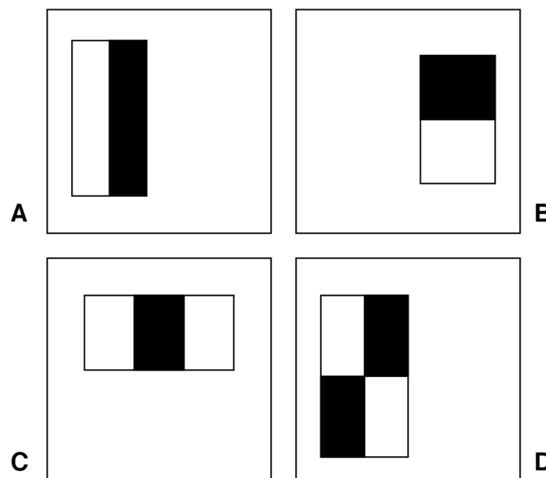
Před samotnou detekcí obličeje a očí z fotografie, která patří mezi důležité součásti této práce, je potřeba čtenáře seznámit se základními metodami a principy používanými pro detekci samotných objektů. Nastínění těchto metod je důležitou součástí, jelikož některé z nich se používají pro samotnou detekci obličeje a očí.

4.1.1 Detektor Viola-Jones

V roce 2001 představili Paul Viola a Michael Jones ve svém článku [38] detektor pro detekci objektů, konkrétně tento detektor představili na problému detekce lidského obličeje. Detektor je charakteristický svoji rychlostí a vysokou mírou úspěšnosti. Je založen na strojovém učení a je potřeba jej natrénovat na mnoha pozitivních a negativních příkladech. Následně lze tento natrénovaný detektor použít k detekci libovolného natrénovaného objektu. K základním stavebním kamenům tohoto detektoru patří Haarovy příznaky (*Haar-like features*), integrální obraz (*Integral Image*), kaskádový klasifikátor a algoritmus *AdaBoost*.

Haarovy příznaky

Hlavní motivací k zavedení těchto příznaků byl fakt, že práce s příznaky je výrazně rychlejší, než práce s jednotlivými pixely. Paul Viola a Michael Jones představili ve svém článku [38] tři základní druhy příznaků, které se dělí na základě typu informace, která má být s daným příznakem detekována. Tyto 3 základní obdélníkové druhy příznaků jsou zobrazeny na obrázku 4.1 a patří mezi ně hranové příznaky (*Edge Features*), čárové příznaky (*Line Features*) a diagonální příznaky (*Four-rectangle features*). Příznakem se myslí rozdíl intenzit jedné oblasti od intenzity jiné oblasti a počítá se jako součet pixelů ležících uvnitř bílých obdélníků odečtený od součtu pixelů ležících uvnitř černých obdélníků.



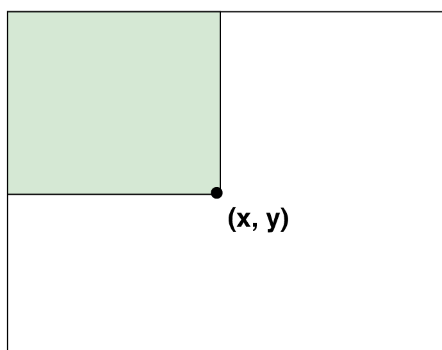
Obrázek 4.1: Ukázka 3 základních obdélníkových příznaků zobrazených vzhledem k detekčnímu oknu. Obrázek (A) a (B) ukazuje hranové příznaky (*Edge Features*). Obrázek (C) čárový příznak (*Line Feature*) a obrázek (D) diagonální příznak (*Four-rectangle feature*).

Integrální obraz

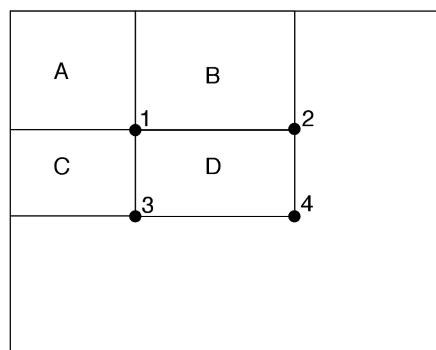
Příznaky mohou být vypočítány velmi rychle s využitím přechodné reprezentace vstupního obrazu, která se nazývá integrální obraz. Hodnota integrálního obrazu v bodě (x, y) je rovna součtu všech pixelů nad a vlevo od pozice (x, y), viz. obrázek 4.2. Tato definice se dá zapsat vzorcem 4.1. Pomocí integrálního obrazu lze libovolný obdélníkový součet vypočítat ve 4 průchodech, jak je ukázáno na obrázku 4.3.

$$ii(x,y) = \sum_{x' \leq x, y' \leq y} i(x', y'), \tag{4.1}$$

kde $ii(x, y)$ je integrální obraz a $i(x, y)$ je vstupní obraz.



Obrázek 4.2: Hodnota integrálního obrazu v bodě (x, y) je rovna součtu všech pixelů nad a vlevo od pozice (x, y).



Obrázek 4.3: Součet pixelů ležících uvnitř obdélníku D lze vypočítat pomocí 4 průchodů. Hodnota integrálního obrazu v bodě 1 je rovna součtu pixelů v obdélníku A. Hodnota v bodě 2 je $A + B$, v bodě 3 pak $A + C$ a v bodě 4 je rovna $A + B + C + D$. Součet v oblasti D může být pak vypočítán jako $4 + 1 - (2 + 3)$.

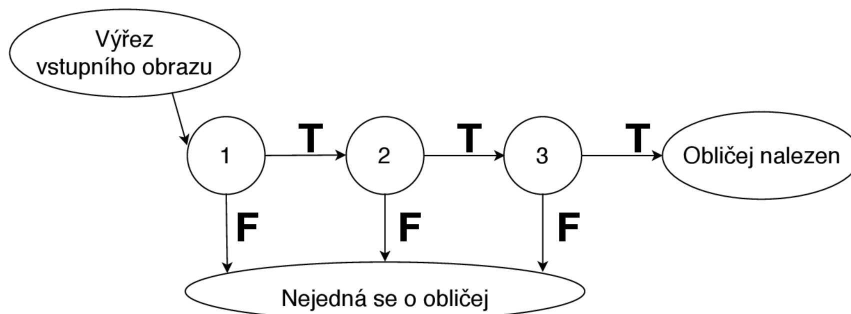
Kaskádový klasifikátor

Paul Viola a Michael Jones představili ve svém článku [38] kaskádový klasifikátor na příkladu detekce obličeje, který pracoval s úspěšností 93,9 %. Samotný kaskádový klasifikátor slouží k detekci libovolných objektů v obraze, dosahuje vysokého výkonu a zároveň radikálně zkracuje dobu výpočtu. Kaskáda se skládá z několika stupňů, tedy řadou slabých klasifikátorů. Slabé klasifikátory jsou tvořeny malým množstvím příznaků (v ideálním případě jde pouze o jeden příznak). Řada těchto slabých klasifikátorů je později sloučena do jednoho silného klasifikátoru.

Samotná klasifikace začíná od prvního stupně. Pokud je zde hledaný objekt nalezen, přechází se k dalšímu stupni. V opačném případě (pokud není objekt v prvním, nebo v jakémkoliv následujícím stupni nalezen) je výřez vstupního obrazu zahozen. Objekt je prohlášen za nalezený, pokud všechny stupně kaskády odpoví pozitivně. Příklad kaskádového klasifikátoru pro detekci obličeje je zobrazen na obrázku 4.4.

Klasifikátor na prvním stupni eliminuje velký počet negativních výřezů za velmi krátký čas, jelikož se zpravidla jedná o obecnější klasifikátor. Klasifikátor na dalším stupni vyžaduje

delší čas na zpracování. Obecně lze říci, že pokud existují stupně kaskády označené jako F_1, F_2 a F_3, pak platí vztah $|F_1| < |F_2| < |F_3|$, nebo-li každý další stupeň je složitější a vyžaduje delší čas na zpracování.



Obrázek 4.4: Znázornění kaskádového klasifikátoru pro detekci obličeje. Samotná klasifikace začíná od prvního stupně. Pokud je zde hledaný objekt nalezen, přechází se k dalšímu stupni. V opačném případě (pokud není objekt v prvním, nebo v jakémkoliv následujícím stupni nalezen) je výřez vstupního obrazu zahozen. Objekt je prohlášen za nalezený, pokud všechny stupně kaskády odpoví pozitivně.

Algoritmus AdaBoost

AdaBoost (zkratka pro *Adaptive Boosting*) je klasifikační algoritmus, který vychází z metody zvané *boosting* [29]. Umožňuje kombinovat několik slabých klasifikátorů do jednoho silného klasifikátoru [6]. Právě tohoto využili ve své práci [38] Paul Viola a Michael Jones, kdy pro vytvoření jednotlivých stupňů kaskády použili k natrénování klasifikátorů právě algoritmus *AdaBoost*.

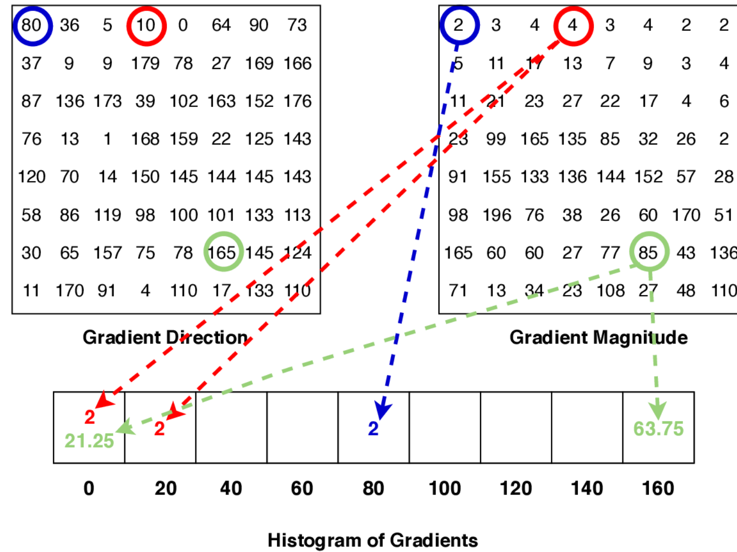
4.1.2 Histogram orientovaných gradientů

Histogram orientovaných gradientů (HOG, z anglického *Histogram of Oriented Gradients*) představili v roce 2005 Navneet Dalal a Bill Triggs na příkladu detekce lidí v obraze [5]. Satya Mallick ve svém článku [22] popisuje základní kroky, které provádí algoritmus HOG při rozhodování, zda-li se ve vstupním obraze vyskytuje hledaný objekt, mezi které patří:

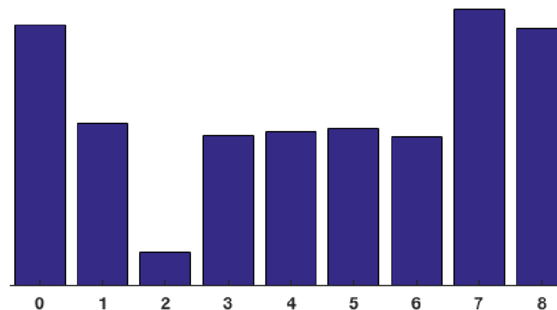
1. **Předzpracování** – Tato fáze zahrnuje např. změnu velikosti [40] obrazu, redukci barevného modelu `RGB` (převedením obrazu do stupňů šedi), nebo použití filtru pro úpravu nerovnoměrného osvětlení a jasu [25].

2. **Výpočet gradientů** – Je nejdůležitější fázi metody HOG, jelikož hledaný objekt je charakterizován právě pomocí gradientních vektorů. V této fázi tedy dochází k výpočtu gradientního vektoru každého pixelu, jeho magnitudy a směru [40].

3. **Výpočet histogramu gradientů v buňkách o rozměru 8×8 pixelů** – V tomto kroku je vstupní obraz rozdělen do buněk o velikosti 8×8 pixelů. Histogram gradientů je následně vypočítán pro každou takovou buňku. Celý proces výpočtu je ilustrován na obrázku 4.5.



Obrázek 4.5: Ukázka výpočtu histogramu gradientů. Histogram gradientů obsahuje 9 zásobníků (*bins*), které odpovídají úhlům 0, 20, 40 . . . 160. První pixel označený modrým kruhem má směr 80 stupňů a magnitudu 2. Jelikož směr 80 stupňů odpovídá přesně jednomu zásobníku, přidá se do tohoto zásobníku hodnota 2. Druhý pixel označený červeným kruhem má směr 10 stupňů a magnitudu 4. Vzhledem k tomu, že 10 je v polovině cesty mezi zásobníkem 0 a 20, dojde k rovnoměrnému rozdělení pixelu do dvou zásobníků. Pro pixely, které mají směr větší jak 160 stupňů (pixel označený zeleným kruhem), dojde k poměrnému rozdělení mezi zásobníky 0 a 160. Stejný proces se provádí pro následující pixely, přičemž hodnoty v jednotlivých zásobnících se sčítají. Výsledkem je histogram, který je zobrazen na obrázku 4.6. Inspirováno z [22].



Obrázek 4.6: Ukázka histogramu, který je výsledkem výpočtu v buňce o rozměru 8×8 pixelů z obrázku 4.5. Převzato z [22].

4. **Normalizace bloků** – Slouží k dosáhnutí lepších výsledků. Nerovnoměrné osvětlení nebo rozdílný kontrast popředí a pozadí má za následek, že velikost gradientů se v rámci celého obrazu významně liší. Tyto nežádoucí vlivy je tedy potřeba nějakým způsobem odstranit [25]. Normalizace probíhá přes bloky o velikosti 16×16 pixelů. Bloky o velikosti 16×16 pixelů obsahují 4 histogramy, které jsou zřetězeny do jednorozměrného vektoru o 36 hodnotách a následně normalizovány.

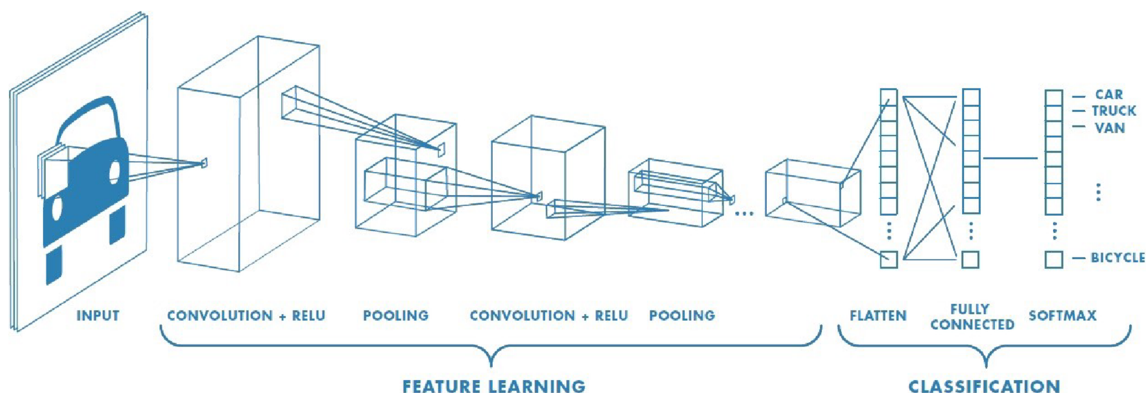
5. **Výsledný HOG deskriptor** – Vektor normalizovaných histogramů pro jeden blok se nazývá deskriptor [25]. Z každého bloku se získá jeden deskriptor. Poslední fází je předání získaných deskriptorů nějakému klasifikátoru [25]. Navneet Dalal a Bill Triggs použili ve svém článku [5] *lineární verzi metody podpůrných vektorů* (LSVM, z anglického *Linear Support Vector Machines*).

6. **LSVM** – Je pouze jednodušší verze SVM metody, což je metoda strojového učení, která se často využívá pro klasifikaci.

4.1.3 Konvoluční neuronové sítě

Konvoluční neuronové sítě (CNN, z anglického *Convolutional Neural Network*) jsou zvláštním druhem neuronových sítí, které se používají ke zpracování dat s pevně danou strukturou [12]. Příkladem pevně dané struktury mohou být např. pixely, které jsou definovány souřadnicemi. Tyto souřadnice je třeba zachovat, protože pokud by došlo k jejich záměně, obrázek by ztratil svůj původní význam [30]. CNN našly v praxi široké využití, mezi které patří např. detekce objektů, obličeje nebo klasifikace obrazových dat [28]. Název pro *konvoluční neuronové sítě* pochází z matematické operace *konvoluce*, která je blíže popsána v následujících odstavcích.

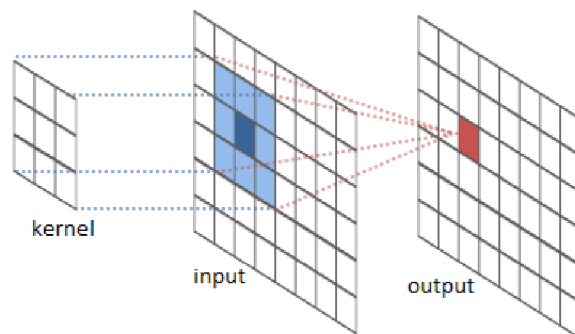
Architektura *konvoluční neuronové sítě* silně závisí na řešeném problému. Na obrázku 4.7 je zobrazena ukázková architektura pro zpracování vstupního obrazu, která se používá ke klasifikaci objektů, konkrétně jde o klasifikaci typu automobilu.



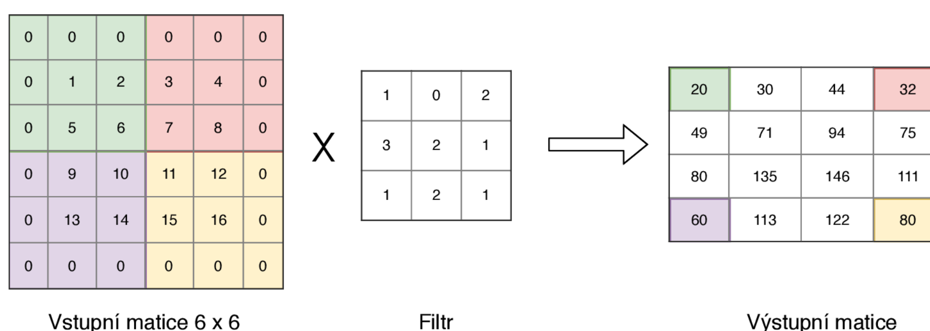
Obrázek 4.7: Ukázková architektura CNN pro zpracování vstupního obrazu, která se používá ke klasifikaci objektů, konkrétně jde o klasifikaci typu automobilu. Převzato z [28].

Konvoluce

Petr Rek, který se ve své diplomové práci [30] zabýval konvolučními neuronovými sítěmi, definoval, že konvoluční operaci v CNN si lze představit jako posouvání okna po vstupní matici a násobení překrývajících se hodnot, což je zobrazeno na obrázku 4.8. Oknu se říká *filtr* (lze se setkat i s označením *jádro* nebo *kernel*) a je ve většině případů mnohem menší než vstupní obrázek. Hodnoty uvnitř *filtru* se nazývají *váhy*. Výsledkem každého posunutí okna je jedna výstupní hodnota. Z těchto hodnot je následně vytvořena výstupní matice, tzv. *feature map*. Uvažujme např. o vstupní matici 6×6 a matici pro *filtr* o rozměrech 3×3. Výsledek operace konvoluce je pak znázorněn na obrázku 4.9.



Obrázek 4.8: Znázornění konvoluční operace v CNN. Okno (*filtr* nebo též *kernel*) je posouváno po vstupní matici a dochází k násobení překrývajících se hodnot. Výsledkem každého posunutí okna je jedna výstupní hodnota. Z těchto hodnot je následně vytvořena výstupní matice, tzv. *feature map*. Převzato z [35].

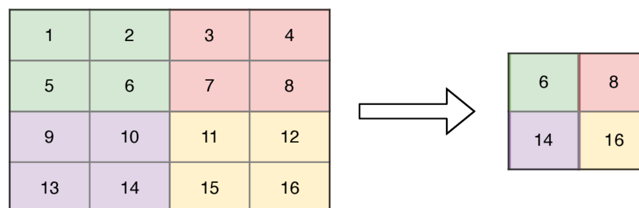


Obrázek 4.9: Ukázka operace konvoluce se vstupní maticí o rozměru 6×6 a *filtrem* o rozměru 3×3. Inspirováno z [30].

Stavební bloky

CNN sestává z vrstev, které jsou typicky řazeny lineárně za sebe (existují i implementace podporující grafové uspořádání). Mezi 4 základní vrstvy patří:

- **Konvoluční vrstva** – Jedná se o základní a nejpodstatnější vrstvu CNN. Skládá se z předem definovaného počtu trojrozměrných *filtrů*. Tyto *filtry* se postupně posouvají po vstupní matici a provádí sčítání násobků odpovídajících hodnot. Výsledkem každého posunutí okna je jedna výstupní hodnota. Z těchto hodnot je následně vytvořena výstupní matice, tzv. *feature map*. Tento proces je znázorněn na obrázku 4.8. Názorný příklad operace konvoluce je zobrazen na obrázku 4.9.
- **Poolingová vrstva** – Cílem této vrstvy je snížení velikosti vstupu, což v praxi znamená, že následné operace mají menší časovou a paměťovou náročnost. Pro dosažení snížení vstupu se používá několik operací, mezi které patří:
  1. *Max pooling* – Je vybrána maximální hodnota. Ukázka této operace je zobrazena na obrázku 4.10.
  2. *Average pooling* – Výstupní hodnota je dána průměrem.



Obrázek 4.10: Ukázka operace *max pooling*. Inspirováno z [30].

- **Aktivační vrstva** – Zavádí do sítě nelinearitu. Velmi často využívanou aktivační funkcí je funkce *ReLU* (*Rectified Linear Unit*). Existují další aktivační funkce jako např. *tanh*, *sigmoid* nebo *softmax*. Aktivační funkce *ReLU* je často používaná hlavně kvůli její výkonnosti [28].
- **Plně propojená vrstva** – Typicky se využívá na konci CNN.

4.2 Detekce obličeje a očí z fotografie

Detekovat lidský obličej a oči z fotografie je možné provádět několika způsoby. V následujících odstavcích jsou zmíněny, porovnány a ukázány na příkladech ty nejpoužívanější. Nejprve je ale nutné zadefinovat pojem, který se v následujících odstavcích vyskytuje ve velké míře.

- **Orientační body obličeje (Facial Landmarks)** = používají se k lokalizaci a reprezentaci oblastí obličeje člověka jako jsou **oči**, **obočí**, **nos**, **ústa** a **čelist** [31]. Orientační body obličeje jsou znázorněny na obrázku 4.11.

Obrázek 4.11: Zelené křivky znázorňující orientační body obličeje (*facial landmarks*). Mezi tyto oblasti patří oči, obočí, nos, ústa a čelist. Převzato z [31].

Pro detekci orientačních bodů obličeje je nutné provést 2 kroky: [31]

1. Detekovat obličej z fotografie.
2. Z detekovaného obličeje následně identifikovat orientační body obličeje.

Detekovat obličej (krok č. 1) lze pomocí: [31, 41]

- Knihovny *OpenCV*
  - S využitím kaskádového klasifikátoru Haar. Klasifikátory jsou uloženy ve formátu `XML` a k detekci obličeje lze využít např. `haarcascade_frontalface_default.xml`.
  - S využitím neuronové sítě.
- Knihovny *Dlib*
  - S využitím `API`, které poskytuje metodu `get_frontal_face_detector()`. Tato metoda využívá k detekci obličeje algoritmus HOG + LSVM.
  - S využitím neuronové sítě.
- Algoritmů založených na hlubokém učení.

Detekovat orientační body obličeje (krok č. 2) lze pomocí:

- Knihovny *OpenCV*
  - S využitím kaskádových klasifikátorů Haar.
- Knihovny *Dlib*
  - S využitím předtrénovaného detektoru přímo v knihovně *Dlib*.

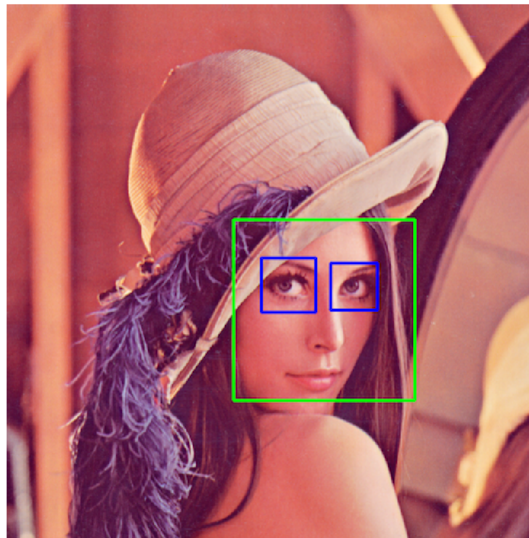
Detekce obličeje a očí pomocí knihovny OpenCV s využitím kaskádového klasifikátoru Haar

Knihovna *OpenCV* již obsahuje mnoho klasifikátorů ve formátu `XML`, mezi které patří i klasifikátory pro detekci obličeje a očí [13]. Ukázka detekce obličeje a očí pomocí knihovny *OpenCV* s využitím kaskádového klasifikátoru Haar je zobrazena na výpisu 4.1. K detekci obličeje se ve výpisu 4.1 používá klasifikátor `haarcascade_frontalface_default.xml`, jak lze vidět na řádku 5. K detekci očí se následně používá klasifikátor `haarcascade_eye.xml`, což lze vidět na řádku 7 výpisu 4.1. Výsledný obrázek, kde je pomocí zeleného, resp. modrého obdélníku vizualizován obalový obdélník (bounding box) obličeje, resp. očí, je zobrazen na obrázku 4.12.

```
1  import numpy as np
2  import cv2
3
4  # nahraj klasifikator pro rozpoznani obliceje
5  face_cascade = cv2.CascadeClassifier("haarcascade_frontalface_default.xml")
6  # nahraj klasifikator pro rozpoznani oci
7  eye_cascade = cv2.CascadeClassifier("haarcascade_eye.xml")
8
9  img = cv2.imread("lena.png") # nahraj obrazek
10 gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # preved obrazek do odstinu sedi
```

```
11 faces = face_cascade.detectMultiScale(gray, 1.3, 5) # detekuj obliceje
12
13 for (x,y,w,h) in faces:
14   # nakresli zeleny obdelnik kolem obliceje
15   cv2.rectangle(img,(x,y),(x+w,y+h),(0,255,0),2)
16   roi_gray = gray[y:y+h, x:x+w]
17   roi_color = img[y:y+h, x:x+w]
18   eyes = eye_cascade.detectMultiScale(roi_gray) # detekuj oci
19   for (ex,ey,ew,eh) in eyes:
20     # nakresli modry obdelnik kolem oci
21     cv2.rectangle(roi_color,(ex,ey),(ex+ew,ey+eh),(255,0,0),2)
22
23 cv2.imshow("Face And Eye Detection Using OpenCV library", img) # zobraz obrazek
24 cv2.waitKey(0)
25 cv2.destroyAllWindows()
```

Výpis 4.1: Ukázka kódu pro detekci obličeje a očí v knihovně *OpenCV* s využitím kaskádového klasifikátoru Haar.



Obrázek 4.12: Detekce obličeje a očí na fotografii pomocí knihovny *OpenCV* s využitím kaskádového klasifikátoru Haar. Obrázek je výsledkem ukázkového výpisu 4.1.

4.2.1 Detekce obličeje a očí pomocí knihovny Dlib s využitím HOG a LSVM

Knihovna *Dlib* také dokáže detekovat lidský obličej a oči z fotografie. Na rozdíl od knihovny *OpenCV* je zde k detekci obličeje využito `API` knihovny *Dlib*. Konkrétně se jedná o metodu `get_frontal_face_detector()`, což je vidět na řádku 8 výpisu 4.2. Tato metoda využívá k detekci obličeje algoritmus HOG + LSVM [41]. Pro detekci očí je použit předtrénovaný detektor, který je součástí knihovny *Dlib*. Tento předtrénovaný detektor slouží k detekci orientačních bodů obličeje a identifikuje tyto body ve formě **68 souřadnic** [31]. Vizualizace těchto souřadnic je zobrazena na obrázku 4.14. *Dlib* detektor byl natrénován na datové

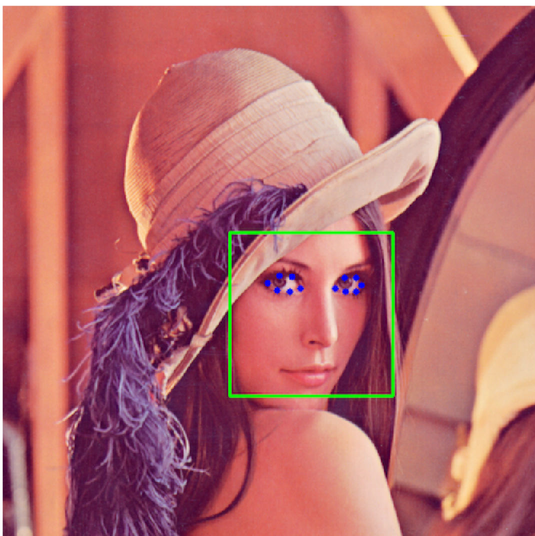
sadě iBUG 300-W[3] a pyšní se velmi vysokou úspěšností detekce orientačních bodů obličeje z fotografie člověka v reálném čase [31]. Inicializace tohoto předtrénovaného detektoru lze vidět na řádku 10 výpisu 4.2.

Ukázka detekce obličeje a očí v knihovně *Dlib* je zobrazena na výpisu 4.2. Výsledný obrázek, kde je pomocí zeleného obdélníku vizualizován obalový obdélník (bounding box) obličeje a pomocí zabudovaného detektoru vizualizovány jednotlivé body pravého, resp. levého oka, je zobrazen na obrázku 4.13. Výhodou knihovny *Dlib* oproti *OpenCV* je zabudovaný předtrénovaný detektor, díky kterému není potřeba žádných dalších klasifikátorů ve formě XML souborů, jak je tomu u knihovny *OpenCV*.

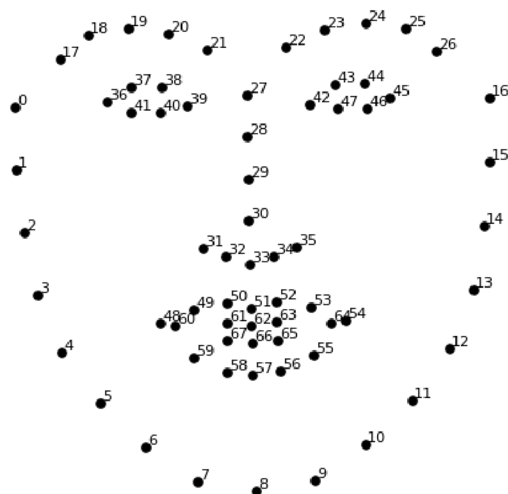
```
1  import imutils
2  from imutils import face_utils
3  import dlib
4  import cv2
5  import numpy as np
6
7  # nahraj klasifikator pro rozpoznani obliceje
8  detector = dlib.get_frontal_face_detector()
9  # nahraj predtrenovany detektor k detekci orientacnich bodu
10 predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")
11
12 img = cv2.imread("lena.png") # nahraj obrazek
13 gray = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY) # preved obrazek do odstinu sedi
14 faces = detector(gray, 1) # detekuj obliceje
15
16 for (i, rect) in enumerate(faces): # pro vsechny nalezene obliceje
17   # preved na OpenCV bounding box
18   (x, y, w, h) = face_utils.rect_to_bb(rect)
19   # nakresli zeleny obdelnik kolem obliceje
20   cv2.rectangle(img, (x, y), (x + w, y + h), (0, 255, 0), 2)
21
22   # detekuj orientacni body
23   shape = predictor(gray, rect)
24   # preved souradnice (x, y) na NumPy array
25   shape = face_utils.shape_to_np(shape)
26
27   # iteruj pres vsechny orientacni body
28   for (name, (i, j)) in face_utils.FACIAL_LANDMARKS_IDXS.items():
29     if (name == "left_eye" or name == "right_eye"):
30       for (x, y) in shape[i:j]:
31         # zobraz modry kruh na souradnici
32         cv2.circle(img, (x, y), 3, (255, 0, 0), -1)
33
34 cv2.imshow("Face And Eye Detection Using Dlib library", img) # zobraz obrazek
35 cv2.waitKey(0)
36 cv2.destroyAllWindows()
```

Výpis 4.2: Ukázka kódu pro detekci obličeje a očí pomocí knihovny *Dlib* s využitím metody `get_frontal_face_detector()` k detekci obličeje a předtrénovaného detektoru k detekci očí.

[3] https://ibug.doc.ic.ac.uk/resources/facial-point-annotations/



Obrázek 4.13: Detekce obličeje a očí na fotografii v knihovně *Dlib*. Obrázek je výsledkem ukázkového výpisu 4.2.



Obrázek 4.14: Vizualizace 68 souřadnic identifikující orientační body obličeje v knihovně *Dlib*. Převzato z [2].

Do této chvíle byly knihovny, které umožňují detekci obličeje, porovnávány spíše podle toho, co nabízejí za `API`, v čem jsou navzájem odlišné a jaké mají klady a zápory. V dnešní době, a především u aplikace, kde hraje detekce obličeje zásadní roli, je velmi důležité porovnat i další extrémně důležité faktory, jako je výkon a přesnost detekce. Z tohoto důvodu jsem provedl analýzu jednotlivých metod, které se používají pro detekci obličeje a všechny nasbírané poznatky jsem shrnul do tabulky 4.1.

Metoda detekce	Výkon	Přesnost
Kaskádový klasifikátor Haar (KKH)	Rychlý.	Méně přesný.
HOG + LSVM	Pomalejší než KKH.	Přesnější než KKH s méně falešně pozitivními (false positive) výsledky.
Neuronová sít	Může být velmi pomalý v závislosti na hloubce a složitosti modelu.	Při správném natrénování výrazně přesnější a robustnější než KKH a HOG + LSVM.

Tabulka 4.1: Porovnání metod sloužících k detekci obličeje podle výkonu a přesnosti [32].

Na základě analýzy metod určených k detekci obličeje, která je zobrazena v tabulce 4.1, osobních zkušeností a experimentů s knihovnami *OpenCV* a *Dlib* jsem se rozhodl v této práci využít k detekci obličeje knihovnu *Dlib*. Přesněji metodu `get_frontal_face_detector()`, která k detekci obličeje využívá HOG a LSVM. Mezi hlavní důvody výběru knihovny *Dlib* patří větší přesnost detekce obličeje, než u knihovny *OpenCV*, která využívá kaskádový klasifikátor Haar a méně falešně pozitivních (false positive) výsledků. Mezi další důvody patří fakt, že knihovna *Dlib* obsahuje zabudovaný a velmi přesný předtrénovaný detektor pro detekci orientačních bodů obličeje, který jsem následně použil pro detekci očí.

Kapitola 5

Metody pro rozpoznání leukokorie a implementace nástroje pro pořizování vhodné datové sady

První část této kapitoly je věnována dvěma metodám, které jsem experimentálně vyzkoušel použít k detekci leukokorie z fotografie lidského obličeje. Obě metody jsou zde popsány, porovnány a následně vyhodnoceny.

Druhá část této kapitoly je věnována nástroji, který slouží pro pořizování vhodné datové sady a významně urychluje proces trénování neuronové sítě. Jsou zde definovány funkční a nefunkční požadavky pro tento nástroj. Dále je zde popsána implementace a spuštění nástroje. Poznatky a vědomosti, které byly posbírány při vytváření nástroje posloužili jako základ pro serverovou část mobilní aplikace.

Jak bylo zmíněno v kapitole 4, k detekci obličeje a očí jsem použil knihovnu *Dlib*. Díky této knihovně je detekce obličeje a získání obalového obdélníku lidského oka velmi přesný a snadný proces. Nyní následuje část, která byla v této práci nejtěžší a zároveň nejvíce experimentální. Tato část je zobrazena na obrázku 5.1 a znázorňuje, že ze získané vstupní fotografie (obalový obdélník lidského oka) je nutné najít vhodný algoritmus nebo způsob, který bude co nejpřesněji detekovat případný výskyt leukokorie.

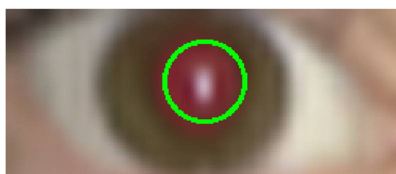
Obrázek 5.1: Nejtěžší a zároveň nejvíce experimentální část celé této práce. Na vstupu je výřez oka, který byl získán pomocí knihovny *Dlib*. Nyní je nutné najít vhodný algoritmus, metodu nebo způsob, který bude co nejpřesněji detekovat případný výskyt leukokorie.

5.1 Metoda prohledávání pixelů ve fotografii oka

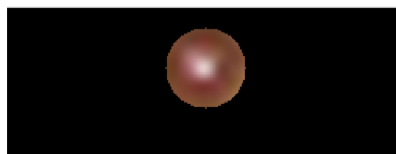
První metodu, kterou jsem pro detekci leukokorie vyzkoušel, byla metoda prohledávání pixelů ze vstupní fotografie, kdy vstupní fotografií je myšlen obalový obdélník lidského oka získaný pomocí knihovny *Dlib*. Celý proces detekce jsem si rozdělil na podproblémy, které jsou zobrazeny na obrázku 5.4.

Po získání obalového obdélníku lidského oka ze vstupní fotografie bylo nutné nalézt zornici. Tento úkol byl velmi složitý, a i přes různé experimenty a použité transformace jsem nikdy nedosáhl požadovaného výsledku. Tento fakt připisuji skutečnosti, že jsem experimentoval i s fotografiemi, které byly pořízeny starším mobilním zařízením s horším fotoaparátem. Na vstupu jsem tedy neměl fotografie lidského oka ve vysokém rozlišení, ale samotné výřezy některých fotografií byly velmi nekvalitní, až rozmazané. Pro nalezení zornice jsem tedy zvolil jiné řešení. Využil jsem skutečnosti, že na fotografii, která je pořízena uživatelem, je uživatel v tzv. *frontální pozici*[1] a jeho zornice je tak v pomyslném středu lidského oka. Dále jsem využil skutečnosti, že knihovna *Dlib* dokáže identifikovat body lidského oka, které jsou ukázány na obrázku 4.14. Z těchto bodů jsem vypočítal střed a pomocí kružnice s tímto vypočítaným středem a poloměrem 25 jsem vymezil prostor, ve kterém se budou prohledávat pixely na případný výskyt leukokorie. Hodnota poloměru byla určena experimentálně podle dostupných testovacích dat. Obalový obdélník s vymezeným prostorem, který se bude prohledávat na případný výskyt leukokorie je ukázán na obrázku 5.2.

Před samotným prohledávání pixelů je potřeba na získaný obalový obdélník lidského oka aplikovat masku. Vstupním obrázkem je tedy výřez oka, který je ukázán na obrázku 5.2, výstupem je potom obrázek ukázaný na obrázku 5.3. Aplikace masky tedy ze vstupního obrázku ponechá pouze prostor, ve kterém se budou porovnávat jednotlivé pixely se zadaným prahem. Každému pixelu mimo tento prostor je přiřazena černá barva.



Obrázek 5.2: Obalový obdélník (bounding box) lidského oka, který je získán z fotografie uživatele. Zelený kruh značí zornici nebo-li prostor, ve kterém se prohledává, a porovná každý pixel s `RGB` prahem.

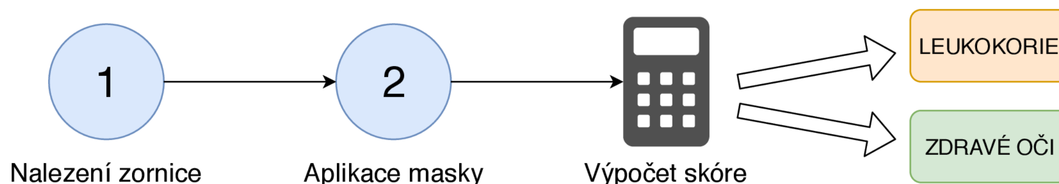


Obrázek 5.3: Obrázek, který je výsledkem aplikace masky na získaný obalový obdélník lidského oka z obrázku 5.2

Následně dojde k prohledávání a porovnávání pixelů s `RGB` prahem. Tento práh reprezentuje různé odstíny červené barvy, která je očekávána u fotografie, která je pořízená bleskem za zhoršených světelných podmínek u zdravých očí jedince. Hodnoty jednotlivých odstínů červené barvy byly nastaveny experimentálně podle dostupných testovacích dat. Pokud prohledávaný pixel leží v zadaném `RGB` rozmezí, dojde k inkrementaci počítadla, které si značí počet nalezených červených pixelů. Jakmile se prohledají a porovnají všechny pixely ve vymezeném území, je známo výsledné skóre – počet nalezených červených pixelů.

[1] **Frontální pozice** je taková pozice, kde se člověk dívá přímo do objektivu fotoaparátu a nemá obličej natočen do strany.

Po analýze obou očí je podle výsledného skóre nalezeného v pravém a levém oku určen finální výsledek, který je uživateli zobrazen. Finální výsledek, který je uživateli zobrazen, je získán na základě podmínek, které jsou zobrazeny v tabulce 5.1.



Obrázek 5.4: Jednotlivé podproblémy, které bylo potřeba vyřešit v rámci metody prohledávání pixelů, aby bylo možné detekovat případný výskyt leukokorie. Nejprve bylo potřeba nalézt zornici, nebo-li vymezené území, ve kterém se bude prohledávat. Následně bylo potřeba aplikovat masku, která zachová pouze pixely nalezeného území. V této masce následně probíhá prohledávání a porovnávání pixelů s RGB prahem. Na základě získaného skóre – počtu červených pixelů se určí konečný výsledek.

Podmínka	**Výsledek (Třída)**
RIGHT_EYE_SCORE < 5 and LEFT_EYE_SCORE < 5	NO_RED_EYES
RIGHT_EYE_SCORE >= 5 and LEFT_EYE_SCORE >= 5	NO_LECORIA
JINAK	LECORIA

Tabulka 5.1: Podmínky, podle kterých je uživateli na základě posbíraného skóre zobrazen výsledek. Skórem se myslí počet nalezených červených pixelů, které jsou očekávány u fotografie pořízené bleskem za zhoršených světelných podmínek u zdravých očí jedince (tzv. *efekt červených očí*).

5.1.1 Vyhodnocení úspěšnosti metody

Podmínkou, kterou musí vstupní fotografie splňovat, aby se zařadila do testovací sady je přítomnost tzv. *efektu červených očí*. Kdyby tato podmínka nebyla požadována, bylo by velmi těžké metodou prohledávání pixelů určit, zda-li se jedná v zornici člověka o leukokorii, nebo o pouhý odraz světla způsobený bleskem. Metoda tedy musí rozeznat fotografie, na kterých se tzv. *efekt červených očí* neprojevil, aby mohla uživatele informovat o skutečnosti, že pro samotnou analýzu je potřeba fotografie s tímto jevem.

Dalším složitým úkolem bylo najít fotografie, na kterých se projevila leukokorie. Z těchto důvodů je testovací sada velmi malá. Výsledky a vyhodnocení úspěšnosti metody prohledávání pixelů na testovací datové sadě jsou uvedeny v tabulce 5.2. Fotografie, které jsou součástí této testovací sady, byly posbírány z následujících zdrojů:

- Osobní zdroje.
- Fotografie od uživatelů, kteří aplikaci testovali nebo používali.
- Server *Flickr*[2].

[2]https://www.flickr.com

Třída	**Počet fotografií**	**Úspěšnost**
Zdravé oči (obsahující *efekt červených očí*)	105	99,05 %
Zdravé oči (bez *efektu červených očí*)	208	97,12 %
Leukokorie	14	100,00 %
Celkem	327	98,72 %

Tabulka 5.2: Výsledky a vyhodnocení úspěšnosti metody prohledávání pixelů na testovací datové sadě.

5.1.2 Problémy metody

Metoda prohledávání pixelů byla první jednoduchou metodou, kterou jsem pro detekci leukokorie vyzkoušel. Jak lze vidět v tabulce 5.2, úspěšnost metody byla velmi dobrá, i když samotná úspěšnost může být zavádějící, jelikož testovací datová sada byla velmi malá. Datovou sadu jsem ale již dále nezvětšoval, protože při experimentování s touto metodou jsem narazil na 3 hlavní problémy, díky kterým by tato metoda v praxi nenašla uplatnění. Mezi hlavní problémy, které jsem při experimentování s touto metodou objevil patří:

1. **Nalezení zornice** – Jak jsem uvedl výše, při pokusu o nalezení zornice ve fotografii s nízkým rozlišením, jsem i přes mnoho experimentů nedospěl k požadovanému výsledku. K detekci zornice jsem tedy využil knihovnu *Dlib* a předpokladu, že uživatel pořídí fotografii v tzv. *frontální pozici*. Samotnou zornici se mi tedy ve výsledku nepodařilo detekovat, ale pomocí experimentů jsem našel vymezený prostor, který ke klasifikaci případné leukokorie postačoval. Jelikož cílem této práce je vytvořit aplikaci medicínského charakteru, přišel mi tento způsob detekce zornice nedostačující, proto jsem hledal další možné způsoby a řešení.

2. **Definování prahu (`RGB` rozmezí barev)** – Při experimentování a hledání fotografií, na kterých se projevil tzv. *efekt červených očí* jsem vypozoroval, že červená barva zornice je u každého jedince velmi odlišná. Odstín červené zornice při tomto efektu záleží na mnoha faktorech, přičemž samotný odstín červené je od velmi světlé až po velmi tmavou. Definovat tedy všechny možné odstíny červené barvy zornice všech lidí na světě pomocí `RGB` prahu je velmi složitý, možná až neřešitelný problém.

3. **Možnost detekce pouze na fotografiích s tzv. efektem červených očí** – Samotná leukokorie se projeví při použití blesku. Leukokorie se ale může projevit i na fotografiích, které byly pořízeny bleskem při dobrém osvětlení. V některých případech lze leukokorii vidět dokonce i makroskopicky bez jakéhokoliv nasvícení [42]. Pořizovat fotografii za zhoršených světelných podmínek je pouze doporučováno, jelikož na fotografii s tzv. *efektem červených očí* lze leukokorii velmi dobře identifikovat. V praxi to znamená, že by bylo potřeba definovat `RGB` práh pro všechny možné barvy zornice, což je znovu velmi složitý, možná až neřešitelný problém.

Kvůli výše zmíněným omezením a problémům související s metodou prohledávání pixelů, jsem hledal další možná a efektivnější řešení pro detekci leukokorie.

5.2 Použití konvoluční neuronové sítě pro detekci leukokorie

Druhou metodou, kterou jsem pro detekci leukokorie vyzkoušel, byla metoda využívající *konvoluční neuronovou síť*. *Konvoluční neuronovou síť* jsem vytvořil pomocí knihovny *Keras*[3] běžící na backendu *Tensorflow*[4].

5.2.1 Keras

Keras je knihovna napsaná v programovacím jazyce *Python*, která nabízí vysokoúrovňové `API` pro práci s neuronovými sítěmi. Je schopná běžet na backendu *Tensorflow*[5], *CNTK*[6] nebo *Theano*[7] [36]. Hlavním důvodem, proč jsem pro tuto práci zvolil právě knihovnu *Keras* je, že tato knihovna byla vytvořena s důrazem na uživatelskou přívětivost a možnost rychlého experimentování. Knihovna je schopna běžet na CPU i GPU a podporuje jak *konvoluční*, tak *rekurentní* sítě, popřípadě jejich kombinace.

5.2.2 Tensorflow

Tensorflow je knihovna vyvinutá společností Google poskytující `API`, které usnadňuje *strojové učení* [24]. Původně byla vyvinuta k velkým numerickým výpočtům, ale právě v oblasti *strojového učení* se stala tato knihovna velmi populární a využívaná. Knihovna je schopna běžet na CPU i GPU.

5.2.3 Vytvoření modelu

Knihovna *Keras* nabízí 2 základní `API` k vytvoření modelu, mezi které patří:

- **Sekvenční (sequential) API** – Nejjednodušší způsob, jak vytvořit model v knihovně *Keras*. Sekvenční `API` umožňuje přidávat jednu vrstvu za druhou [36]. Problémem sekvenčního `API` je, že neumožňuje modelům mít více vstupů nebo výstupů, což je při řešení některého problému vyžadováno. Sekvenční `API` je ideální volbou pro řešení většiny problémů, mezi které patří i problém detekce leukokorie.

- **Funkční (functional) API** – Používá se k vytvoření složitých modelů, mezi které patří např. modely s více výstupy.

K vytvoření *konvoluční neuronové sítě* pomocí sekvenčního `API` stačí vytvořit pouze objekt `Sequential`. Pro přidání jednotlivých vrstev se využívá metoda `add()`. Před trénováním modelu je potřeba nakonfigurovat proces učení pomocí metody `compile()`. Ukázka vytvoření modelu, který byl v této práci použit pro detekci leukokorie je ukázán na výpisu 5.1. Řádek 1 vytváří objekt `Sequential`. Na řádcích 3 – 11 dochází k přidání jednotlivých vrstev do *konvoluční neuronové sítě*. Model, který jsem použil k detekci leukokorie má tedy 9 vrstev. Pro lepší přehled čtenáře jsem celou architekturu *konvoluční neuronové sítě*, která se používá k detekci leukokorie, zobrazil na obrázku 5.5. Na řádku 13 výpisu 5.1 je konfigurovaný proces učení.

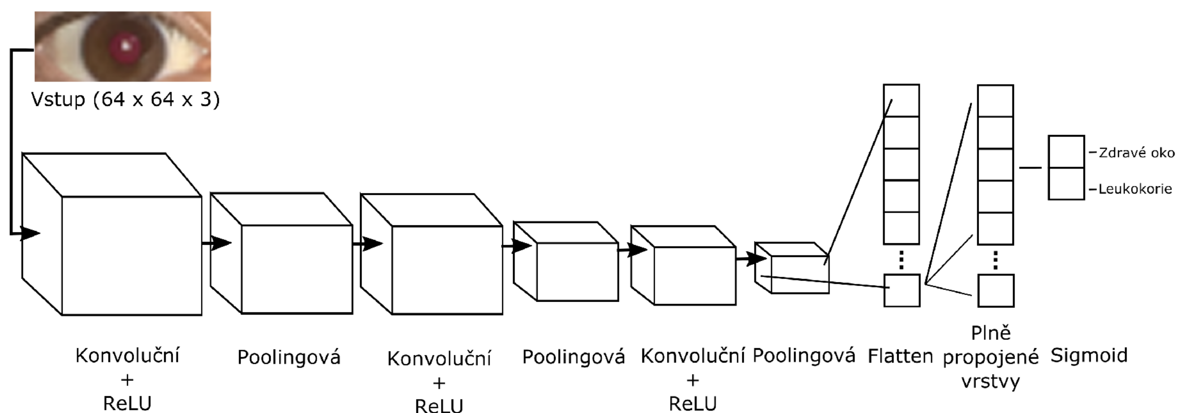
[3] https://keras.io
[4] https://www.tensorflow.org
[5] https://www.tensorflow.org
[6] https://docs.microsoft.com/en-us/cognitive-toolkit/
[7] http://www.deeplearning.net/software/theano/

```
model = Sequential()

model.add(Conv2D(32, (3, 3), input_shape = (64, 64, 3), activation = "relu"))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Conv2D(32, (3, 3), activation = "relu"))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Conv2D(64, (3, 3), activation = "relu"))
model.add(MaxPooling2D(pool_size = (2, 2)))
model.add(Flatten())
model.add(Dense(units = 128, activation = "relu"))
model.add(Dense(units = 1, activation = "sigmoid"))

model.compile(
  optimizer = "adam",
  loss = "binary_crossentropy",
  metrics = ["accuracy"]
)
```

Výpis 5.1: Příklad vytvoření modelu *konvoluční neuronové sítě* pomocí knihovny *Keras* s využitím sekvenčního `API`. Tento model byl v této práci použit k detekci leukokorie.



Obrázek 5.5: Architektura *konvoluční neuronové sítě*, kterou jsem použil pro detekci leukokorie.

5.2.4 Augmentace datové sady

Získání velkého množství dat (výřezů očí) lidí, u kterých se projevila leukokorie, bylo velmi složité. Z tohoto důvodu jsem použil proces *augmentace*. *Augmentace* je proces, který se používá k získání více dat za účelem vytvořit model robustnější, a vyřešit tak problém nedostatku dat. Jedná se tedy o proces, který z jednoho obrázku vytvoří X dalších obrázků za pomocí transformací jako např. rotace, přiblížení obrázku nebo přidání šumu [36]. Pro *augmentaci* dat v knihovně *Keras* existuje metoda `ImageDataGenerator()`. Ukázka augmentace datové sady pomocí metody `ImageDataGenerator()` v knihovně *Keras* je zobrazena na výpisu 5.2.

```
train_datagen = ImageDataGenerator(
  rescale = 1./255,
  shear_range = 0.2,
  zoom_range = 0.2,
  horizontal_flip = True
)
```

Výpis 5.2: Ukázka *augmentace* datové sady v knihovně *Keras*, kterou jsem použil z důvodu nedostatečného počtu dat (výřezů očí) lidí, u kterých se projevila leukokorie.

5.2.5 Trénování modelu

Jakmile je model vytvořený a přeložený, lze začít s trénováním vytvořeného modelu. K trénování modelu nabízí knihovna *Keras* metody `fit()` a `fit_generator()`. Jelikož jsem pro datovou sadu využil *augmentace* pomocí metody `ImageDataGenerator()`, tak jsem pro trénování použil metodu `fit_generator()`. Trénovací datová sada obsahuje následující 2 třídy:

- **Healthy (Zdravé oči)** – Výřezy očí zdravých jedinců. Množina obsahuje 1082 fotografií (výřezů očí).
- **Leukocoria (Leukokorie)** – Výřezy očí jedinců, u kterých se projevila leukokorie. Množina obsahuje 153 fotografií (výřezů očí).

5.2.6 Vyhodnocení úspěšnosti metody

Vyhodnocení úspěšnosti detekce leukokorie s použitím *konvoluční neuronové sítě* je ukázáno v tabulce 5.3. Celková úspěšnost je sice o 0,58 % horší, než u předchozí metody prohledávání pixelů, datová sada je však o 229 fotografií větší. Největší výhodou metody, která používá ke klasifikaci leukokorie *konvoluční neuronovou síť*, spočívá v eliminaci problémů, které byly nalezeny u předchozí metody prohledávání pixelů, a které jsou zmíněny v kapitole 5.1.2.

Uživatel tak může analyzovat fotografii na případný výskyt leukokorie, která byla pořízena s použitím blesku i v případech, kde se neprojevil tzv. *efekt červených očí*. Z tohoto důvodu jsem k detekci leukokorie využil metodu, která využívá *konvoluční neuronovou síť*.

Třída	Počet fotografií	Úspěšnost
Zdravé oči	536	96,27 %
Leukokorie	20	100,00 %
Celkem	556	98,14 %

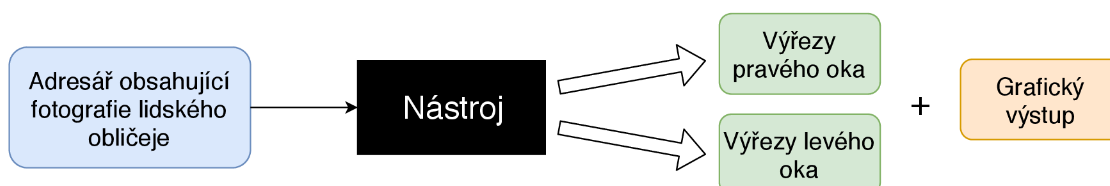
Tabulka 5.3: Výsledky a vyhodnocení metody na testovací datové sadě, kde byla pro detekci leukokorie využita *konvoluční neuronová síť*.

5.3 Nástroj pro pořizování vhodné datové sady

Ke klasifikaci možného výskytu leukokorie jsem z výše uvedených důvodů vybral použití *konvoluční neuronové sítě*. Před samotným použitím vytvořeného modelu je potřeba tento model natrénovat. V praxi to znamená, že jsou potřebná vstupní data pro natrénování modelu. Právě z toho důvodu je součástí této práce implementování nástroje, který ve fotografii

lidského obličeje najde obalový obdélník (bounding box) pravého, resp. levého oka, který následně uloží. Celý proces je zobrazen na obrázku 5.6. Kdyby takový nástroj neexistoval, bylo by potřeba každý obrázek zpracovat manuálně a získat obalové obdélníky pravého, resp. levého oka.

Nástroj je implementován ve formě konzolové aplikace. Kromě uložení jednotlivých obalových obdélníků nalezených očí, poskytuje nástroj grafický výstup, na kterém jsou vidět všechny zpracované výřezy očí. Díky tomuto grafickému výstupu lze jednoduše a rychle zkontrolovat, zda-li došlo ke správné detekci obličeje a očí. Celkový grafický výstup lze vidět na obrázku 5.7. Před samotnou implementací jsem definoval funkční a nefunkční požadavky na samotný nástroj.



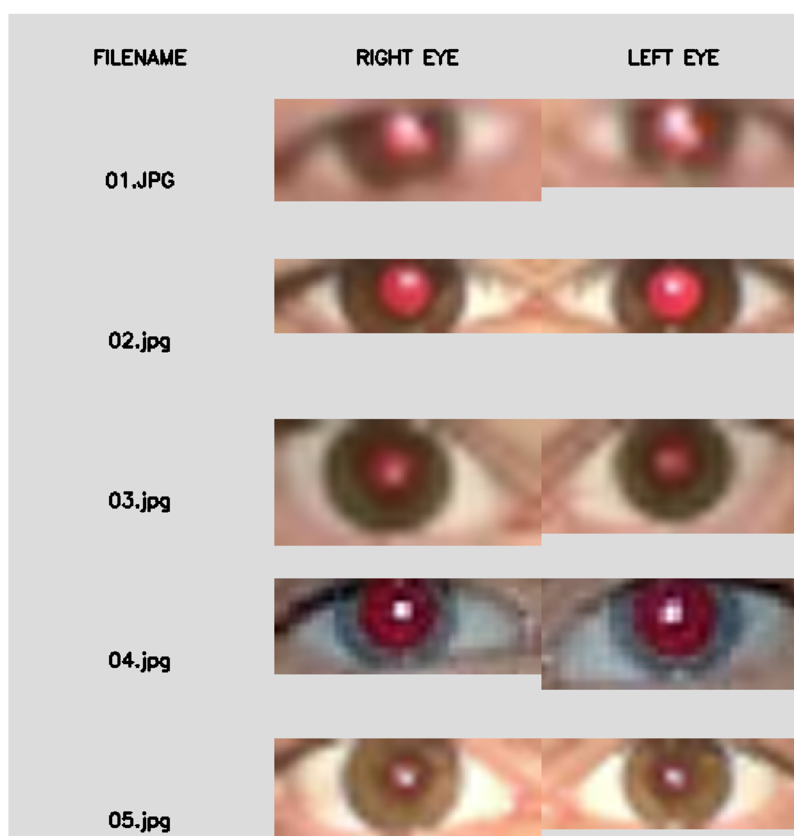
Obrázek 5.6: Kompletní proces nástroje pro pořizování vhodné datové sady, který je součástí této práce. Vstupem je adresář, který obsahuje fotografie lidského obličeje. Výstupem jsou výřezy pravého, resp. levého oka pro každou fotografii z adresáře. Kromě výřezů poskytuje nástroj ještě jeden grafický výstup, na kterém jsou vidět všechny zpracované výřezy očí. Díky tomuto výstupu lze jednoduše a rychle zkontrolovat, zda-li došlo ke správné detekci obličeje a očí.

5.3.1 Funkční požadavky

- Vstupní fotografie bude obsahovat jeden lidský obličej v tzv. *frontální pozici*.
- Uživateli se při spuštění nástroje budou v příkazové řádce vypisovat následující údaje:
  - Bude vypsán adresář, ze kterého chce uživatel zpracovat fotografie.
  - Bude vypsán počet obrázků v adresáři.
  - Při zpracování jednotlivé fotografie se bude vypisovat název aktuálně zpracovávané fotografie a pořadí v datové sadě.
- Uživateli se po ukončení nástroje uloží výřezy pravého, resp. levého oka pro všechny fotografie ze zadaného adresáře.
- Uživateli se po ukončení nástroje zobrazí grafický výstup, na kterém bude zobrazeno následující:
  - Název fotografie.
  - Výřez pravého oka z fotografie.
  - Výřez levého oka z fotografie.

5.3.2 Nefunkční požadavky

- Vstupní fotografie budou mít koncovku `png`, `jpg` nebo `jpeg`.
- Nástroj bude implementován v programovacím jazyku *Python*, protože tento jazyk nabízí rozhraní pro knihovny *Dlib* a *OpenCV*.
- Pro detekci obličeje a očí bude bude použita knihovna *Dlib*.



Obrázek 5.7: Ukázka grafického výstupu z nástroje pro pořizování vhodné datové sady, který je součástí této práce, a na kterém jsou vidět všechny zpracované výřezy očí. Díky tomuto výstupu lze jednoduše a rychle zkontrolovat, zda-li došlo ke správné detekci obličeje a očí.

5.3.3 Implementace nástroje

K implementaci tohoto nástroje jsem zvolil programovací jazyk *Python*. Mezi hlavní důvody, proč jsem si vybral právě tento jazyk patří:

- Jedná se o jeden z nejpopulárnějších jazyků v oblasti zpracování obrazu.
- Obě známé a populární knihovny pro zpracování obrazu – *OpenCV* a *Dlib* obsahují rozhraní právě pro jazyk *Python*.
- Má velmi dobrou přenositelnost programů.

Prvním krokem je zpracování samotných vstupních parametrů nástroje. Konvence povinných a volitelných vstupních parametrů je uvedena níže v tabulce 5.4. Nástroj ověří, že uživatel zadal všechny povinné parametry. Pokud jsou zadány všechny potřebné parametry, dochází ke zpracování fotografie, v opačném případě je uživateli do příkazového řádku zobrazena chybová hláška.

Před samotným zpracováním fotografie dojde ve `for` cyklu k iteraci přes všechny fotografie z adresáře, jehož cestu zadal uživatel jako vstupní parametr. Každá fotografie z adresáře je následně zpracována. K otevření fotografie a ověření, zda-li uživatel zadal správnou cestu k fotografii, slouží funkce `load_img()`. K samotnému otevření fotografie jsem použil knihovní funkci `cv2.imread()` knihovny *OpenCV*. Zde je nutné poznamenat, že knihovna *OpenCV* využívá ve výchozím nastavení barevné spektrum v pořadí `BGR`, místo klasického `RGB`. Pro následující práci s obrázkem je nutné na tuto skutečnost myslet. Dojde-li k úspěšnému otevření fotografie, přechází se k samotnému zpracování, o které se stará funkce `process_image()`, v opačném případě je uživateli zobrazena chybová hláška.

Funkce `process_image()` zmenší fotografii pomocí knihovní funkce `imutils.resize()` knihovny *imutils*. Následně je fotografie převedena do odstínu šedé (grayscale), jelikož k samotné detekci obličeje není zapotřebí barevné fotografie. O převod do odstínu šedé se stará knihovní funkce `cv2.cvtColor()` knihovny *OpenCV*. Jako první parametr vyžaduje fotografii, která se má převést do odstínu šedé. Jako druhý parametr se očekává barevný prostor, do kterého má být fotografie převedena. Jelikož, jak již bylo výše zmíněno, knihovna *OpenCV* pracuje ve výchozím nastavení s barevným spektrem v pořadí `BGR`, je potřeba nastavit tento parametr na hodnotu `COLOR_BGR2GRAY`. Po převodu fotografie do odstínu šedé následuje samotná detekce obličeje. K detekci obličeje z fotografie jsem využil `API` knihovny *Dlib*, konkrétně jsem použil k inicializaci detektoru funkci `get_frontal_face_detector()`, který k detekci obličeje používá metodu HOG + LSVM. Hlavní důvody, proč jsem si vybral pro detekci obličeje knihovnu *Dlib* jsem shrnul v kapitole 4.2, konkrétně v tabulce 4.1. Pokud detektor nenalezne ve fotografii žádný obličej, resp. nalezne více než jeden obličej, dojde k vypsání této skutečnosti na výstup a fotografie se dále nezpracovává. Pokud je na fotografii nalezen právě jeden obličej, je fotografie zpracovávána dále a následuje proces detekce očí.

K detekci očí jsem použil taktéž knihovnu *Dlib*, která obsahuje již předtrénovaný detektor pro detekci orientačních bodů obličeje ve formě 68 souřadnic [31]. Celý proces nalezení obalového obdélníku (bounding box) lidského oka je tak díky knihovně *Dlib* velmi přesný a snadný. O nalezení obalového obdélníku se stará funkce `get_eye_bb()`. Hlavní důvody, proč jsem si vybral pro detekci očí knihovnu *Dlib* jsem shrnul v kapitole 4.2.

Jak již bylo výše zmíněno, samotný nástroj poskytuje grafický výstup, který je zobrazen na obrázku 5.7. K vytvoření tohoto výsledného obrázku se podílí několik funkcí. Nejprve je zavolána funkce `create_blank_image()`, která vytvoří prázdný obrázek, na který se budou postupně zapisovat jednotlivé výřezy očí a jména souborů. Jedním z parametrů této funkce je počet obrázků, které se budou na prázdný obrázek zapisovat. Další funkcí, která se následně volá je `draw_headers()`. Tato funkce doplní do zatím prázdného obrázku hlavičku. Mezi další funkce, které se používají v průběhu vytváření grafického výstupu, patří `move_in_row()` a `move_in_col()`. Výsledný grafický obrázek si lze představit jako tabulku, která obsahuje 3 sloupce (název souboru, pravé oko, levé oko) a x řádků, kde x představuje hlavičku + počet zpracovávaných obrázků. Právě díky těmto funkcím pro posun se dá v „tabulce“ pohybovat a zapisovat (názvy fotografií, samotný výřez oka) do jednotlivých „buněk“.

5.3.4 Spuštění nástroje

Nástroj je implementován jako konzolová aplikace v programovacím jazyce *Python* a má název `eyechecktool`. Konvence povinných a volitelných vstupních parametrů nástroje je zobrazena v tabulce 5.4. Příklad spuštění nástroje je ukázán na výpisech 5.3, 5.4 a 5.5.

Parametr	Povinný	Výchozí hodnota	Popis
--shape-predictor	Ano	-	Cesta k *Dlib* shape prediktoru.
--image-folder-path	Ano	-	Cesta k adresáři s fotografiemi.
--output-folder-path	Ne	out	Cesta k uložení výstupů nástroje.
--output-image-name	Ne	results.png	Název grafického výstupu.
--start-sequence-number	Ne	-	Počáteční číslo sekvence fotografií.

Tabulka 5.4: Konvence povinných a volitelných vstupních parametrů nástroje pro pořizování vhodné datové sady – `eyechecktool`.

```
$ python3 eyechecktool.py --shape-predictor shape_predictor_68_face_landmarks.dat
--image-folder-path /Users/pavelhrebicek/Desktop/DIP_faces/NO_LECORIA/
```

Výpis 5.3: Příklad spuštění nástroje `eyechecktool`. Výsledný grafický výstup, obsahující výřezy pravého, resp. levého oka pro všechny fotografie ze zadaného adresáře, bude uložen do adresáře `out` a soubor bude mít název `results.png`.

```
$ python3 eyechecktool.py --shape-predictor shape_predictor_68_face_landmarks.dat
--image-folder-path /Users/pavelhrebicek/Desktop/DIP_faces/NO_LECORIA/ --output-
folder-path /Users/pavelhrebicek/Desktop/Results --output-image-name vyrezy_oci.png
```

Výpis 5.4: Příklad spuštění nástroje `eyechecktool`. Výsledný grafický výstup, obsahující výřezy pravého, resp. levého oka pro všechny fotografie ze zadaného adresáře, bude uložen do adresáře `/Users/pavelhrebicek/Desktop/Results` a soubor bude mít název `vyrezy_oci.png`.

```
$ python3 eyechecktool.py --shape-predictor shape_predictor_68_face_landmarks.dat
--image-folder-path /Users/pavelhrebicek/Desktop/DIP_faces/NO_LECORIA/ --start-
sequence-number 100
```

Výpis 5.5: Příklad spuštění nástroje `eyechecktool`. Výsledný grafický výstup, obsahující výřezy pravého, resp. levého oka pro všechny fotografie ze zadaného adresáře, bude uložen do adresáře `out` a soubor bude mít název `results.png`. Jednotlivé výstupy levého, resp. pravého oka budou mít místo názvu souboru číselné označení. První získaný výřez bude mít číslo 100. Parametr `--start-sequence-number` byl zaveden z důvodu, aby nebylo nutné získané výřezy manuálně přejmenovávat, ale rovnou kopírovat do datové sady, jelikož fotografie v datové sadě jsou seřazeny podle číselného názvu souboru.

Kapitola 6

Návrh mobilní aplikace

První část této kapitoly se věnuje návrhu mobilní aplikace z pohledu uživatelského rozhraní. Uživatelské rozhraní je zde navrhnuto, otestováno a vyhodnoceno. Druhá část této kapitoly se věnuje návrhu z pohledu samotné architektury mobilní aplikace.

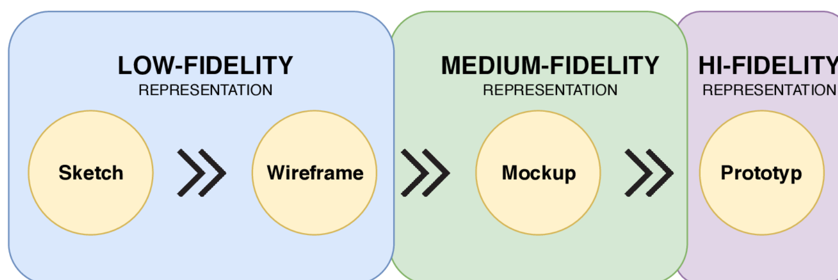
6.1 Návrh uživatelského rozhraní

Před samotným návrhem je důležité zadefinovat termíny, které jsou nutné k pochopení následujících odstavců:

- **Low-Fidelity** reprezentace = jde o jednoduchý koncept, vytvořený většinou pouze pomocí pera a papíru. Cílem je proměnit nápady do vizuální podoby, díky které lze v prvotních fázích projektu dostat zpětnou vazbu od uživatelů [8].

- **High-Fidelity** reprezentace = jde o vysoce funkční a interaktivní koncept, který je velmi podobný cílovému produktu. Používá se často v pozdějších fázích k testování použitelnosti a identifikaci problémů [8].

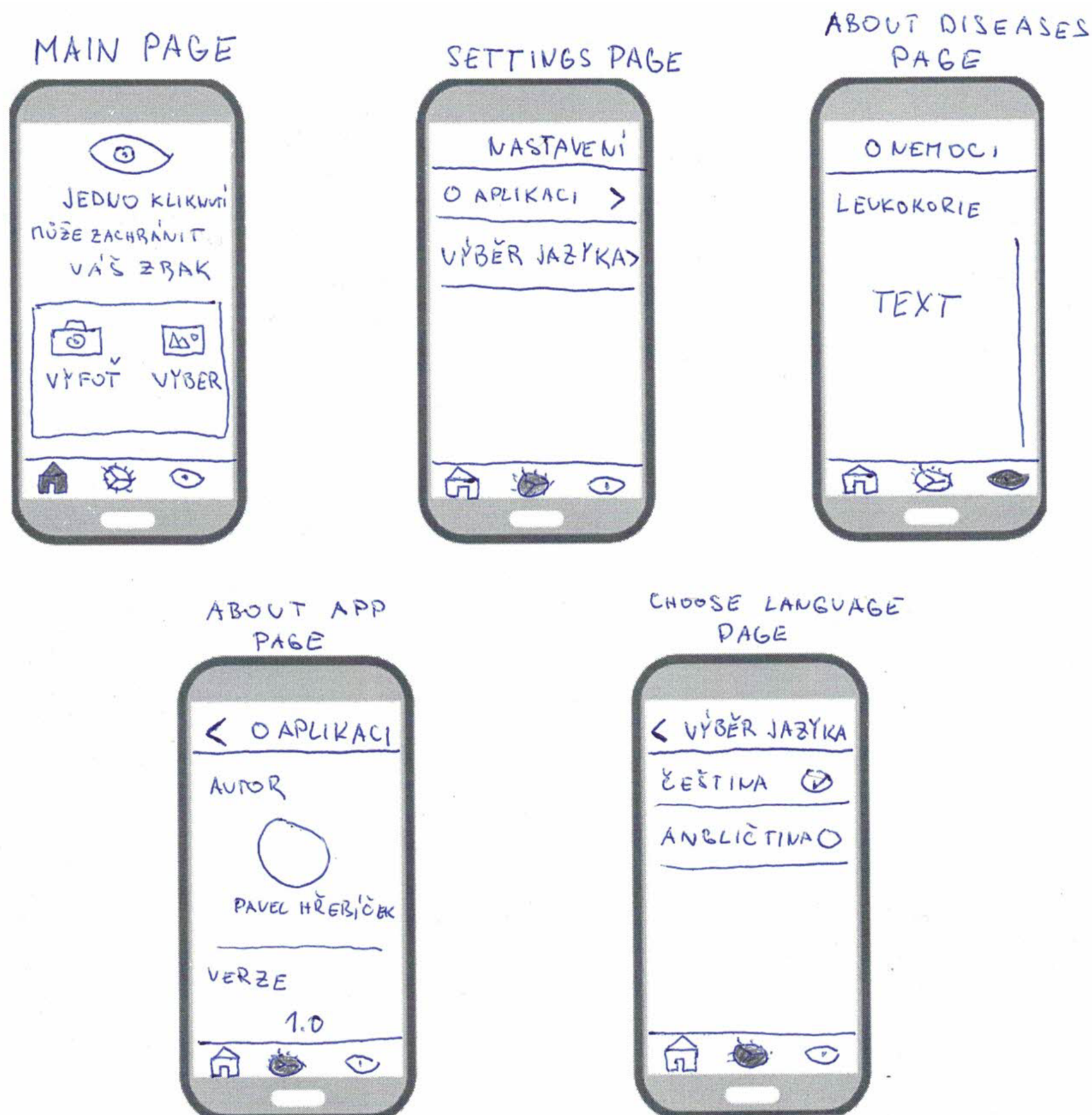
Při návrhu uživatelského rozhraní pro mobilní aplikaci Eye Check jsem se řídil procesem návrhu, který ve svém článku [39] popsal Matt Warcholinski. Tento proces je ukázán na obrázku 6.1 a zobrazuje všechny etapy typické pro vývoj designu a uživatelského rozhraní webové nebo mobilní aplikace.



Obrázek 6.1: Etapy vývoje designu a návrhu uživatelského rozhraní od prvotního náčrtu na papír (*Sketch*) až po samotný prototyp aplikace. V každé etapě se využívá rozdílný návrh, který se liší v účelu použití. Mezi tyto formy návrhu patří *Sketch*, *Wireframe*, *Mockup* a *Prototyp*. Inspirováno z [39].

6.1.1 Sketch

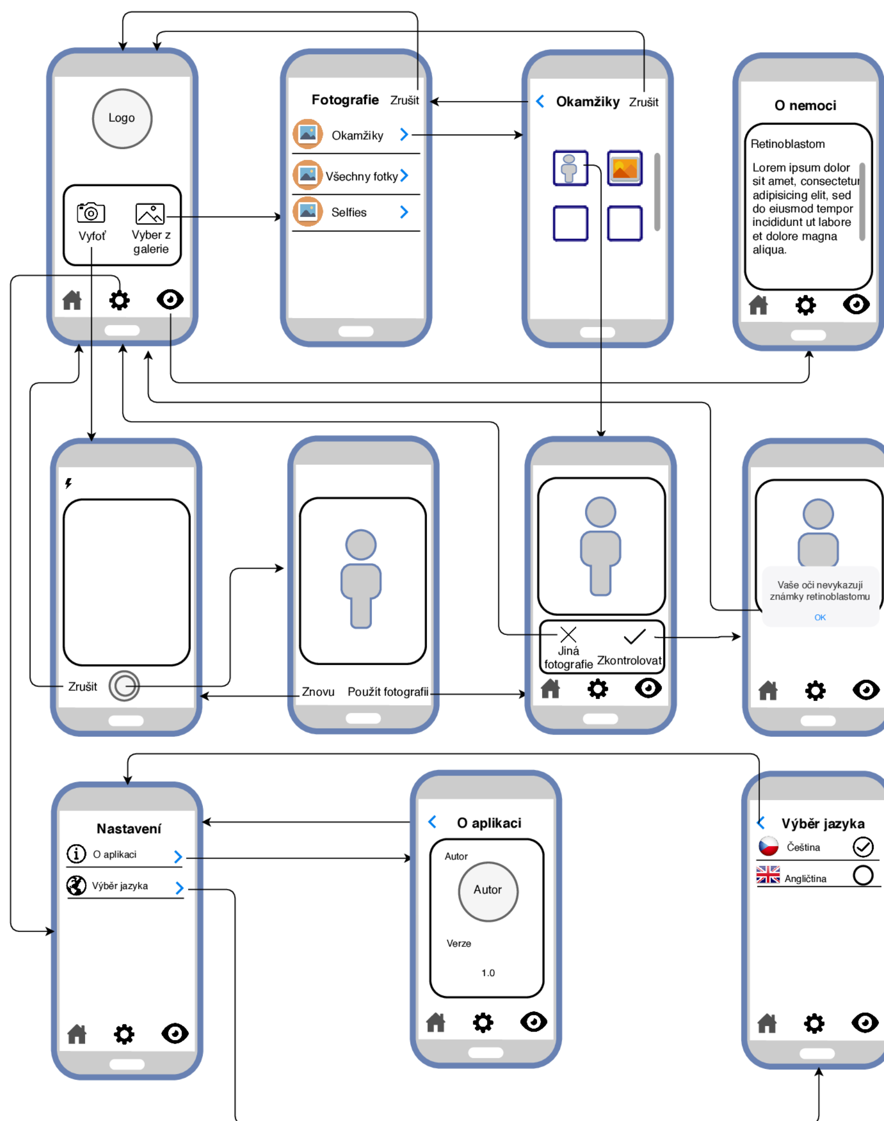
Jedná se v podstatě o ruční kresbu na papír, která poskytuje *Low-Fidelity* reprezentaci aplikace. Jde o velice rychlý a snadný způsob, jak lze vizualizovat první návrh samotné aplikace. Je založen na principu, který říká, že i jednoduchý náčrt může vyjádřit a popsat nápad mnohem lépe než slova. Při vytváření *Sketche* vznikají nové nápady na vylepšení aplikace. Velkou roli zde hraje představivost a fantazie. Vytvořit *Sketch* je nezbytný krok pro přechod k *Wireframu* [39]. *Sketch*, který jsem vytvořil pro mobilní aplikaci Eye Check je ukázán na obrázku 6.2.



Obrázek 6.2: *Sketch*, který byl vytvořený pro mobilní aplikaci Eye Check. *Sketch* je velice rychlý a snadný způsob, jak lze vizualizovat první návrh samotné aplikace. Typicky se vytváří pomocí tužky nebo pera na papír.

6.1.2 Wireframe

Wireframe je ekvivalent ke kostře či jednoduchému návrhu webové nebo mobilní aplikace. Mezi hlavní důvody, proč se *wireframe* vytváří, patří zobrazení vztahů mezi jednotlivými obrazovkami aplikace – kam se přejde, nebo co se vykoná po kliknutí na tlačítko. V této fázi se také rozhoduje o umístění tlačítek a prvků na obrazovce. *Wireframe* nezahrnuje design aplikace. Nemusí zde tedy být přesné fonty písma, loga a barvy, které budou použity ve výsledné aplikaci. Používá se k rychlé komunikaci mezi vývojáři, popřípadě zákazníkem [39, 16]. *Wireframe*, který jsem vytvořil pro mobilní aplikaci Eye Check je ukázán na obrázku 6.3. Pro vytvoření wireframe jsem využil službu draw.io[1].

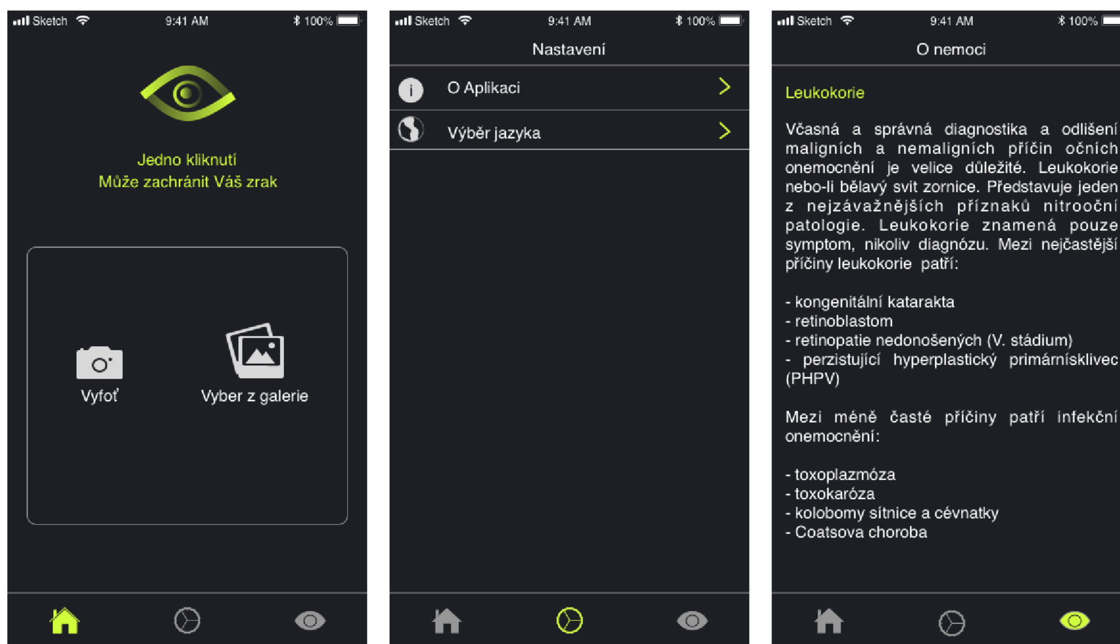


Obrázek 6.3: *Wireframe*, který byl vytvořený pro mobilní aplikaci Eye Check. Mezi hlavní důvody, proč se *wireframe* vytváří, patří zobrazení vztahů mezi jednotlivými obrazovkami aplikace.

[1] https://www.draw.io

6.1.3 Mockup

Mockup představuje velmi blízkou vizualizaci výsledné aplikace se statickou reprezentací funkcionality [16]. Samotný *mockup* slouží jako velmi dobrý nástroj pro poskytnutí zpětné vazby od uživatelů, jelikož testovaní uživatelé si aplikaci dokáží velmi dobře představit [16]. Oproti *wireframe* již *mockup* obsahuje barvy, fonty, obrázky a loga, které budou zastoupeny i ve výsledné aplikaci [39]. *Mockupy* hlavních obrazovek, které jsem vytvořil pro mobilní aplikaci Eye Check, jsou ukázány na obrázku 6.4. Pro vytvoření *mockupů* jsem využil nástroj *Sketch*[2].



Obrázek 6.4: *Mockupy* hlavních obrazovek mobilní aplikace Eye Check. *Mockupy* již představují velmi blízkou vizualizaci výsledné aplikace.

6.1.4 Prototyp

Prototyp poskytuje *High-Fidelity* reprezentaci aplikace. Jde v podstatě o *mockup*, který je obohacený o interakce, animace a vše, co je při kliknutí na tlačítka očekáváno. Matt Warcholinski ve svém článku [39] zmiňuje, že vytvoření prototypu **není vždy nutným krokem** k vytvoření aplikace.

Pokud člověk, který aplikaci vytváří, je zároveň vývojář, může místo vytváření prototypu výslednou aplikaci již přímo naprogramovat na základě vytvořených *mockupů*. Právě toho jsem využil v této práci, kdy namísto vytváření prototypu jsem rovnou aplikaci naprogramoval na základě vytvořených *mockupů*.

Pokud ovšem nejde o vývojáře je velmi doporučeno prototyp vytvořit. Jedná se totiž o velmi detailní představení nápadu, který lze ukázat např. kamarádům, rodině nebo potenciálnímu zákazníku. Jediné, co zde chybí, je samotná funkčnost aplikace [39].

[2]https://www.sketchapp.com

6.2 Testování a vyhodnocení uživatelského rozhraní

Mobilní aplikace se vytváří hlavně pro uživatele, kteří ji budou používat. Z tohoto důvodu je velmi důležité provést testování na samotných uživatelích. Cílem testování bylo zjistit, jak se uživatelům s aplikací pracuje a získat jejich zpětnou vazbu. Všechny poznatky, připomínky a nápady na vylepšení jsem si poznamenal, vyhodnotil a následně provedl úpravu stávajícího uživatelského rozhraní.

6.2.1 Testování

Testování proběhlo s 11 uživateli patřící do věkové skupiny 20-55 let. Mezi testovanými uživateli byli zastoupeni velmi zdatní jedinci, kteří využívají mobilní aplikace každý den, ale i uživatelé, kteří aplikace používají jen velmi zřídka. Testování bylo prováděno pomocí *mockupů*, které jsem vytvořil v rámci návrhu uživatelského rozhraní.

Pro testování jsem zvolil metodu *Usability testing*, kterou ve své knize [37] popisují Russ Unger a Carolyn Chandler. Tato metoda je založena na velmi snadném principu:

1. Vytvoření sady úkolů.
2. Zadání těchto vytvořených úkolů uživatelům, kteří je budou plnit.
3. Zaznamenávat si, kde uživatel uspěl a kde měl naopak problémy.

Pro testování jsem tedy vytvořil sadu úkolů, které jsou sepsány v tabulce 6.1. Úkoly byly zaměřené především na ověření, zda-li se uživatel v aplikaci snadno orientuje a dokáže jednoduše analyzovat fotografii na případný výskyt leukokorie. Při plnění úkolů jsem si na papír zaznamenával poznatky z průběhu testování (co uživateli šlo, kde se zastavil).

Po konci testování jsem každému uživateli položil otázky ohledně právě skončeného testování, které jsem shrnul do tabulky 6.2. Zajímaly mě především dojmy a pocity z testování a ze samotné aplikace.

Číslo úkolu	**Úkol**
1	Pořiďte fotografie člověka v mobilní aplikaci.
2	Vyberte fotografii člověka z galerie mobilního telefonu.
3	Vybranou fotografii z úkolu 2 analyzujte na výskyt leukokorie.
4	Změňte jazyk v mobilní aplikaci na angličtinu.
5	Nalezněte v mobilní aplikaci informace o leukokorii.
6	Nalezněte, o jakou verzi aplikace se jedná.

Tabulka 6.1: Úkoly, které byly kladeny uživatelům v rámci testování uživatelského rozhraní.

Číslo otázky	**Otázka**
1	Přišla Vám orientace v aplikaci intuitivní?
2	Dělalo Vám splnění nějakého úkolu problém?
3	Co se Vám nelíbilo (a proč) a co byste změnil/a?

Tabulka 6.2: Otázky, které byly kladeny uživatelům po ukončení testování uživatelského rozhraní.

6.2.2 Vyhodnocení testování

Poznatky z testování jsem si sepsal a vyhodnotil. V tabulce 6.3 jsou uvedeny všechny poznámky a podněty k vylepšení, které byly v průběhu testování nasbírány. Ke každému bodu je uvedeno i jeho řešení. Tyto poznámky a náměty jsem zapracoval a vytvořil tak další verzi uživatelského rozhraní.

Poznámka/Podnět k vylepšení	**Řešení**
Není ukázáno, jak leukokorie vypadá.	Do sekce „O nemoci“ přidat obrázek s leukokorií.
Položka „Výběr jazyka“ je prioritnější.	Prohodit položky v sekci „Nastavení“.

Tabulka 6.3: Poznámky a podněty k vylepšení aplikace nasbírané při testování uživatelského rozhraní.

6.3 Návrh architektury mobilní aplikace

Nedílnou součástí návrhu mobilní aplikace byl samotný návrh architektury. Klientskou aplikaci jsem se rozhodl implementovat ve frameworku *React Native (JavaScript)*. Proč jsem se rozhodl právě pro tento framework, a jaké přináší výhody, jsem popsal níže v kapitole 7. Serverovou část mobilní aplikace jsem implementoval v programovacím jazyku *Python*. Důvody, proč jsem se rozhodl právě pro tento jazyk, jsem již popsal v kapitole 5.

Pro komunikaci jsem tedy potřeboval vybrat architekturu, která odděluje implementaci a nabízené služby. Přesně tuto podmínku naprosto splňuje architektura `REST` [1], která je dále popsána v následujících odstavcích. Pro úplné pochopení je zde také zmíněn protokol `HTTP`, který je úzce spjat s architekturou `REST`.

6.3.1 HTTP/1.1

Jedná se o textový, bezstavový protokol typu dotaz – odpověď. Klient zasílá požadavek a server odpovídá. Protokol `HTTP` tedy rozlišuje 2 druhy zpráv: [10]

1. Požadavek klienta – *Request*.
2. Odpověď serveru – *Response*.

Mezi metody protokolu `HTTP` patří `GET`, `HEAD`, `POST`, `PUT`, `DELETE`, `CONNECT`, `OPTIONS` a `TRACE` [10].

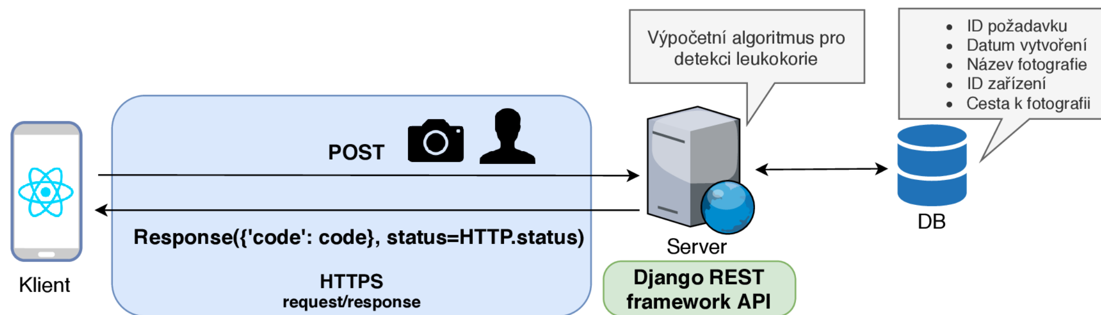
6.3.2 Representational State Transfer (REST)

Architekturu `REST` popsal ve své disertační práci v roce 2000 Roy Thomas Fielding [9], který je spoluautorem protokolu `HTTP`. Architektura sama o sobě nedefinuje formát přenosu, obvykle se však využívají formáty `JSON` či `XML`. `REST` má velmi úzkou vazbu na protokol `HTTP`, protože pro komunikaci využívá `HTTP` metody. Mezi metody, které jsou typicky použity s `REST API` patří: [1]

- `GET` – Získává data ze serveru.
- `PUT` – Nahraje data na server.
- `POST` – Odesílá uživatelská data na server.

- DELETE – Smaže uvedený objekt ze serveru.

Mezi hlavní výhody architektury REST patří separace klienta a serveru. Realizace klienta a serveru může být provedena nezávisle, aniž by jeden o druhém věděl [1]. To znamená, že jak klient, tak server mohou být implementovány v jiném programovacím jazyku, ale mohou spolu komunikovat přes formát, který oba podporují – např. JSON. Právě tohoto jsem využil v mé práci, kdy klient je naimplementovaný v *JavaScriptu*, server v *Pythonu* a přes architekturu REST spolu komunikují. Tato komunikace a celá architektura mobilní aplikace je znázorněna na obrázku 6.5.



Obrázek 6.5: Architektura mobilní aplikace Eye Check. Klient zasílá POST požadavek s fotografií uživatele na server. Server následně detekuje obličej, oči a s využitím *konvoluční neuronové sítě* klasifikuje získané výřezy očí na případný výskyt leukokorie. Výsledek analýzy je serverem zaslán zpět na stranu klienta. Ke komunikaci mezi klientem a serverem je využita architektura REST.

Kapitola 7

Implementace mobilní aplikace

Na světovém trhu s mobilními telefony v dnešní době převládají zařízení s operačními systémy *iOS* a *Android*[1]. Tato skutečnost, cíl této práce poskytnout aplikaci co nejvíce uživatelům a existence moderních frameworků pro tvorbu multiplatformních mobilních aplikací, mě přiměla k tomu, že jsem se rozhodl vytvořit **multiplatformní** mobilní aplikaci dostupnou právě pro operační systémy *iOS* a *Android*. Jelikož jeden z hlavních cílů této práce je poskytnout aplikaci co nejširšímu spektru uživatelů, rozhodl jsem se vytvořit aplikaci **multijazyčnou** – konkrétně si uživatel může v současné době vybrat mezi češtinou a angličtinou.

V úvodu této kapitoly jsou nastíněny různé možnosti vývoje multiplatformní mobilní aplikace. Dále je zde vysvětleno a zdůvodněno, proč byl pro implementaci zvolen právě framework *React Native* a je zde ukázán základní koncept tohoto frameworku. Zbytek této kapitoly se soustředí na konkrétní implementaci klientské a serverové části aplikace.

7.1 Možnosti vývoje multiplatformní mobilní aplikace

Dnešní doba nabízí celou škálu programovacích jazyků a frameworků pro vytvoření multiplatformní mobilní aplikace. V následujících odstavcích jsou představeny a porovnány ty nejznámější způsoby a mobilní frameworky, pomocí kterých se multiplatformní mobilní aplikace dnes vyvíjí.

7.1.1 Nativní mobilní aplikace

Jeden ze způsobů, jak vytvořit aplikaci pro obě platformy, je vytvořit tzv. nativní mobilní aplikaci pro obě platformy odděleně. Tedy pro platformu *iOS* vytvořit aplikaci v programovacím jazyku *Swift*[2] nebo *Objective-C*[3] a pro platformu *Android* vytvořit aplikaci pomocí programovacího jazyku *Java*. V tabulce 7.1 jsem shrnul klady a zápory vývoje multiplatformní aplikace jako dvou oddělených nativních aplikací.

[1] http://gs.statcounter.com/os-market-share/mobile/worldwide

[2] https://swift.org

[3] https://developer.apple.com/library/archive/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html

Nativní mobilní aplikace	
+	-
Výkon.	Vývoj pro obě platformy současně je drahý.
Přístup ke všem nativním funkcím.	Pro jednoduché aplikace se nehodí.
Lepší user experience (UX).	Potřeba umět složitější jazyk (např. *objective-C*).

Tabulka 7.1: Srovnání kladů a záporů při vývoji multiplatformní aplikace jako dvou oddělených nativních aplikací [7].

V dnešní době však již existují přístupy, které dokáží ten samý kód přeložit, jak pro platformu *iOS*, tak pro platformu *Android*. Mezi tyto přístupy patří tzv. *Hybridní mobilní aplikace* nebo např. framework *React Native*.

7.1.2 Hybridní mobilní aplikace

Hybridní aplikace jsou aplikace vytvořené pomocí webových technologií jako *HTML*, *CSS* a *JavaScript*. Hlavní rozdíl oproti nativním mobilním aplikacím je v tom, že hybridní mobilní aplikace běží v tzv. *WebView* – mobilním prohlížeči [27]. Mezi zástupce hybridních frameworků patří např. *Phonegap*[4], *Ionic*[5], *Cordova*[6]. V tabulce 7.2 jsem shrnul klady a zápory vývoje multiplatformní aplikace pomocí hybridních mobilních frameworků.

Hybridní mobilní aplikace	
+	-
Jazyky *HTML*/*CSS*/*JavaScript*.	Jsou pomalejší.
Levnější než vývoj nativních aplikací.	Nepůsobí zcela nativně.
Jeden kód pro obě platformy.	
Rychlejší vývoj než nativní aplikace.	

Tabulka 7.2: Srovnání kladů a záporů při vývoji multiplatformní aplikace pomocí hybridních mobilních frameworků [7].

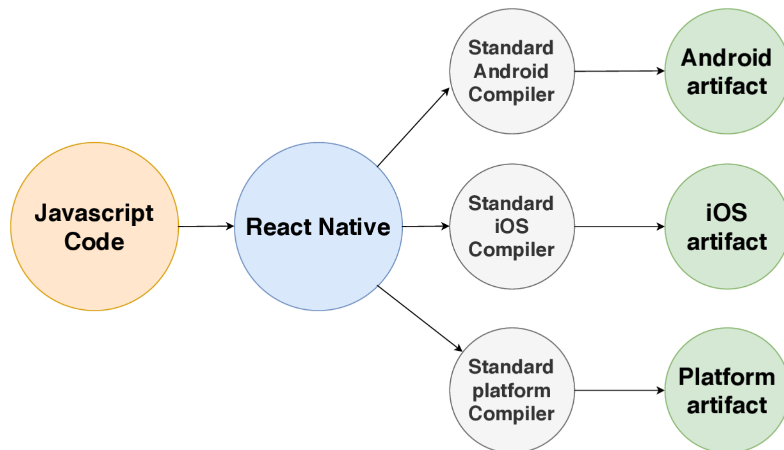
7.2 React Native

Pro vytvoření mobilní aplikace Eye Check jsem použil framework *React Native*[7]. *React Native* je framework pro vytváření mobilních aplikací za pomocí programovacího jazyka *JavaScript* a javascriptové knihovny *React* udržovaný společností *Facebook*. *React Native* pracuje na velmi podobném principu jako *React* s tím rozdílem, že k zobrazení aplikace nepoužívá *HTML*, ale poskytuje velmi podobné stavební bloky – komponenty [23]. *React Native* používá stejné základní stavební bloky uživatelského rozhraní jako běžné aplikace pro *iOS* a *Android*. Místo použití nativních programovacích jazyků jako *Swift* nebo *Java* lze však tyto stavební bloky dát dohromady pomocí *Javascriptu* a *Reactu*.

Hlavní výhodou *React Native* oproti konkurenčním hybridním frameworkům (*Phonegap*, *Ionic*, *Cordova*) zaměřených na vývoj multiplatformních mobilních aplikací spočívá ve skutečnosti, že *React Native* nepoužívá tzv. *WebView* – mobilní prohlížeč, ale používá nativní

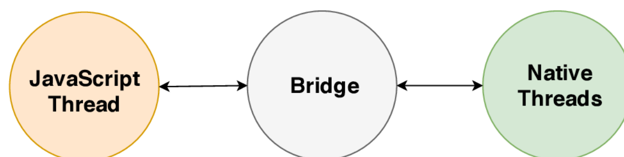
[4]https://phonegap.com
[5]https://ionicframework.com
[6]https://cordova.apache.org
[7]https://facebook.github.io/react-native/

bloky uživatelského rozhraní jako jsou v *iOS* a *Android* aplikacích. Když se v *React Native* vytváří *Android* aplikace, použije se ke zkompilování standardní Android kompilátor. Když se vytváří *iOS* aplikace, použije se standardní iOS kompilátor. Tento koncept je znázorněn na obrázku 7.1.



Obrázek 7.1: Základní koncept frameworku *React Native*, kdy se nepoužívá *WebView*, ale dochází ke kompilaci kódu platformě specifickým kompilátorem. Inspirováno z [11].

Srdcem celé *React Native* architektury je bezpochyby tzv. *bridge* [11]. Jedná se o koncept, který umožňuje obousměrnou a asynchronní komunikaci mezi javascriptovým vláknem a vlákny nativními [11], jak je zobrazeno na obrázku 7.2. Javascriptová strana posílá asynchronní zprávy ve formátu `JSON` popisující akci, kterou má nativní část vykonat. *Bridge* tedy zastává roli zprostředkovatele zpráv a ovládá asynchronní příkazy mezi javascriptovou a nativní stranou.



Obrázek 7.2: Koncept architektury *React Native*, kdy *JS* vlákno komunikuje s nativními vlákny pomocí *bridge*. Inspirováno z [11].

Mezi další důvody, proč jsem k implementaci použil právě framework *React Native* oproti hybridním frameworkům založených na *WebView*, patří jeho výkon, který umožňuje dosáhnout až 60 fps a tzv. *Hot Reload*. Díky funkci *Hot Reload* lze vyvíjet aplikaci velmi rychle, jelikož namísto opětovné kompilace lze aplikaci okamžitě znovu načíst při zachování současného stavu. Funkce *Hot Reload* je velmi užitečná a sám jsem ji mnohokrát využil především při vytváření uživatelského rozhraní, kdy namísto opětovné kompilace lze pozorovat změny v reálném čase a ušetřit tak mnoho času.

Jak již bylo zmíněno výše, *React Native* je založen na komponentách. Při vytváření uživatelského rozhraní lze využít znovupoužitelnosti těchto komponent a výrazně si ulehčit

práci. Samotný *React Native* má velkou vývojářskou komunitu a nabízí již mnoho hotových komponent, které lze snadno nainstalovat přes správce balíčků *npm*[8].

V neposlední řadě je třeba zmínit, že v případě potřeby optimalizovat část aplikace napsané v *React Native* lze vytvořit nativní kód napsaný v programovacím jazyku *Swift*, *Objective-C* nebo *Java* a tento kód potom využít v samotné aplikaci. Tento přístup je velmi užitečný, když existují platformně specifické komponenty – jako např. kamera a lze tak naplno využívat platformně specifické `API`.

7.2.1 Komponenta

Základním stavebním blokem celé *React Native* aplikace je komponenta. Z čistě programátorského hlediska jde o třídu, která dědí od třídy *Component* z knihovny *React* a obsahuje povinnou metodu `render()`. Celá aplikace je potom poskládána a vytvořena z několika těchto komponent. *React Native* již poskytuje řadu vytvořených a zabudovaných komponent, mezi které patří např. `View`, `Text`, `Image`.

React Native rozeznává dva typy komponent, funkcionální (*functional*) a třídní (*class*) [17]. Funkcionální komponenty jsou často označovány jako „hloupé“ (*dumb*), třídní jako „chytré“ (*smart*) [3]. Funkcionální komponenty mají za úkol pouze zobrazit data. Třídní komponenty mají vnitřní stav (`state`) a umožňují přístup k metodám životního cyklu (*lifecycle methods*) komponenty.

7.2.2 Životní cyklus komponent

Životní cyklus komponent lze definovat jako čas od okamžiku, kdy je komponenta poprvé vložena do `DOM`[9], po celou dobu, kdy je komponenta v `DOM` a bod, kdy je komponenta odebrána z `DOM` [15].

Každá komponenta má několik tzv. metod životního cyklu (*lifecycle methods*), které umožňují reagovat na změny životního cyklu komponenty. Samotný životní cyklus je zobrazen na obrázku 7.3 a lze jej rozdělit do 3 kategorií: [15]

1. **Mounting** – Tyto metody jsou volány v následujícím pořadí při vytváření instance komponenty a vkládání do `DOM`:
    - `constructor()`
    - `componentWillMount()` – Tato metoda je již zastaralá a bude nahrazena metodou `static getDerivedStateFromProps()` od verze 17 knihovny *React*.
    - `render()`
    - `componentDidMount()`
2. **Updating** – Aktualizace může být způsobena změnou atributu (`props`) nebo stavu (`state`). Tyto metody jsou volány v následujícím pořadí, když je komponenta znovu vykreslována:
    - `componentWillReceiveProps()` – Tato metoda je již zastaralá a bude nahrazena metodou `static getDerivedStateFromProps()` od verze 17 knihovny *React*.
    - `shouldComponentUpdate()`

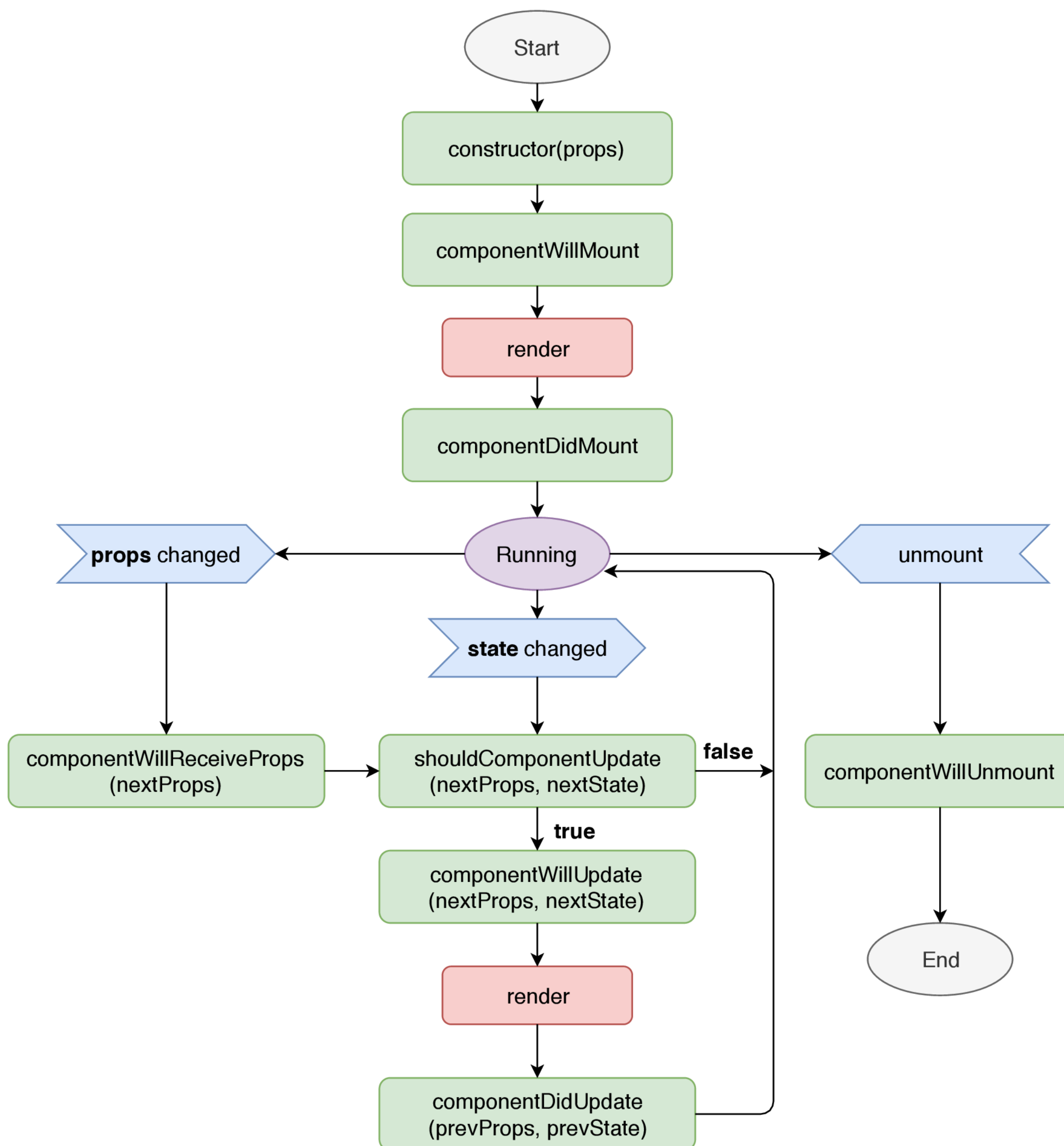
[8] https://www.npmjs.com

[9] **Document Object Model (DOM)** je rozhraní, které umožňuje přistupovat k obsahu dokumentů a rovněž tento obsah modifikovat.

- `componentWillUpdate()` – Tato metoda je již zastaralá a bude nahrazena metodou `getSnapshotBeforeUpdate()` od verze 17 knihovny *React*.
- `render()`
- `componentDidUpdate()`

3. **Unmounting** – Následující metoda je volána, když je komponenta odebrána z `DOM`.

    - `componentWillUnmount()`



Obrázek 7.3: Životní cyklus komponenty ve frameworku *React Native*. Inspirováno z [15].

7.3 Redux

Redux[10] je stavový kontejner pro aplikace napsané v programovacím jazyku *JavaScript*. Lze jej velmi dobře použít jak pro aplikace napsané v *Reactu*, tak v *React Native*. Hlavní idea tohoto přístupu je ukázána na obrázku 7.4. *Redux* je založen na 3 základních principech:

- **Single source of truth** – Stav celé aplikace je uložen ve stromu objektů v jediném úložišti zvaném *store*.
- **State is read-only** – Jediný způsob, jak změnit stav, je vykonat akci (*actions*), což je objekt, který popisuje co se stalo.
- **Changes are made with pure functions** – Ke specifikaci, jak se stav změní na základě provedené akce, slouží tzv. reducery (*reducers*). *Reducery* jsou čisté (*pure*) funkce, které přijímají dva parametry, současný stav, akci a vrací nový stav.

7.3.1 Actions

Akce slouží k odeslání dat z aplikace do *store*. Pro samotný *store* jsou jediným zdrojem informací. Odeslání samotných dat probíhá pomocí funkce `dispatch()`. Z programátorského hlediska jde o klasický *JavaScriptový* objekt, který musí obsahovat povinný atribut `type`, který rozlišuje, o jaký typ akce se jedná. Další předávané atributy jsou potom zcela v režii programátora.

7.3.2 Reducers

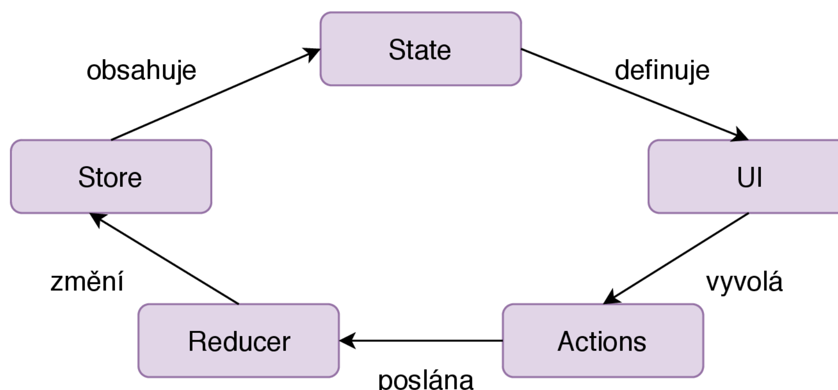
Reducery určují, jak se stav aplikace změní v reakci na odeslanou akci do *store*. *Reducer* je funkce, která přijímá dva parametry, současný stav, akci a vrací nový stav. Velmi důležitým faktorem je, aby *reducer* zůstal čistou (*pure*) funkcí. V samotném těle *reduceru* by nikdy nemělo docházet k tzv. *side efektům* jako např. k volání `API`. Dále by se zde neměly volat žádné nečisté (*non-pure*) funkce jako např. `Date.now()` nebo `Math.random()`.

7.3.3 Store

V předchozích odstavcích byly definovány akce (*actions*), které odesílají data z aplikace do *store*, na základě provedení nějaké akce a reducery (*reducers*), které změní stav aplikace na základě provedení těchto akcí. Store je objekt, který tyto dva přístupy spojuje dohromady. Důležité je také poznamenat, že v *Redux* aplikaci je možné mít pouze jeden *store*, který se vytvoří voláním funkce `createStore()` a má následné odpovědnosti:

- Udržuje stav aplikace.
- Umožňuje přístup ke stavu pomocí funkce `getState()`.
- Umožňuje modifikaci stavu pomocí funkce `dispatch(action)`.
- Registruje tzv. *listenery* pomocí funkce `subscribe(listener)`.
- Odregistrovává tzv. *listenery* pomocí funkce, kterou vrací `subscribe(listener)`.

[10] https://redux.js.org/introduction/getting-started



Obrázek 7.4: Hlavní idea přístupu *Redux*.

7.4 Eslint

Eslint[11] je open source[12] *JavaScriptový linting*[13] nástroj, který vytvořil v roce 2013 Nicholas C. Zakas. *Linting* kódu je typ statické analýzy, který se používá k nalezení problematických vzorů nebo kódu, který nedodržuje určitý styl tzv. *style guide*[14]. Při vytváření softwarového produktu se pracuje velmi často v týmu. Každý programátor v tomto týmu má však jiné zvyky a používá jiné konstrukce. Příkladem může být např. odsazení zdrojového kódu. Někdo pro samotné odsazení používá mezery, někdo tabulátory. V reálném světe je však problém odsazení zdrojového kódu jenom jeden z mnoha.

Právě tyto důvody byly motivací pro vytvoření nástroje *Eslint*, který pomáhá týmu dodržovat stejná pravidla, která zvyšují čitelnost a čistotu kódu. Vše funguje na jednoduchém principu, kdy jsou nejprve definována pravidla pro zdrojový kód, nebo-li styl (*style guide*) zdrojového dokumentu. Následně *Eslint* podtrhuje místa ve zdrojovém kódu, které definovaným pravidlům neodpovídají. Aby nebylo nutné procházet všechny soubory v projektu a kontrolovat, co *Eslint* podtrhl, lze samotný nástroj spustit jako příkaz v příkazovém řádku. Po ukončení *Eslint* vypíše názvy souborů, řádků a problémů, které se vyskytli a jsou v rozporu s definovanými pravidly.

Při implementaci aplikace Eye Check jsem použil právě *Eslint*, jelikož bylo mým cílem mít zdrojový kód konzistentní, dobře čitelný a čistý. Jako *style guide* jsem použil jeden z nejznámějších a nejpoužívanějších – *Airbnb JavaScript Style Guide*[15]. Spuštění *Eslint* nástroje pro projekt, který je součástí této práce, je ukázáno na výpisu 7.1.

```
$ npm run lint
```

Výpis 7.1: Příklad spuštění nástroje *Eslint* pro projekt, který je součástí této práce.

[11] https://eslint.org

[12] **Open source** je počítačový software s otevřeným zdrojovým kódem. Zdrojový kód je poskytnut vývojářům, kteří jej mohou upravovat a dále vylepšovat.

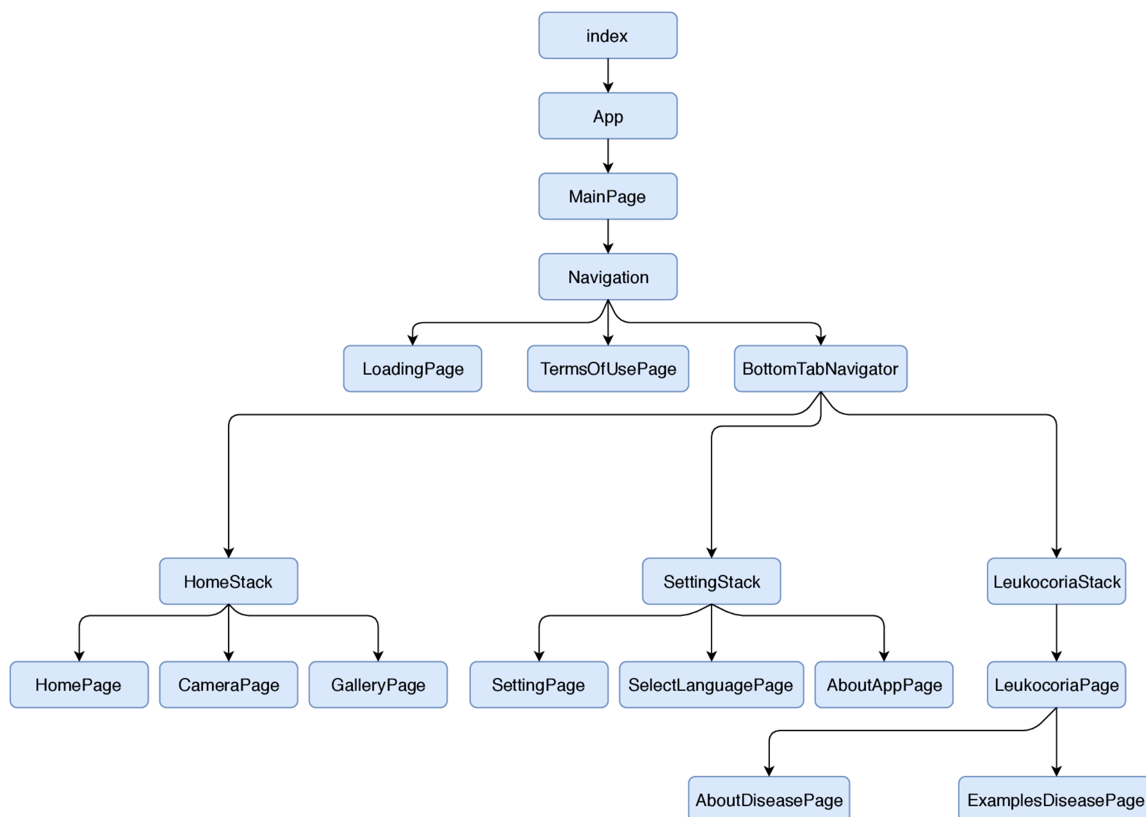
[13] **Lint** nebo **Linter** je nástroj, který analyzuje zdrojový kód a upozorňuje programátora na programátorské a stylistické chyby, popřípadě špatné konstrukce.

[14] **Style guide** je soubor pravidel pro psaní a úpravu zdrojového kódu.

[15] https://github.com/airbnb/javascript

7.5 Implementace klientské části

Klientská část mobilní aplikace reprezentuje tu část aplikace, která běží na mobilním zařízení uživatele. Je zodpovědná za pořízení nebo vybrání fotografie, která se bude analyzovat na případný výskyt leukokorie, poslání požadavku na server a zobrazení výsledku uživateli na základě odpovědi ze serveru. Dále zde uživatel může zvolit jazyk aplikace, zobrazit si základní informace o leukokorii nebo se podívat na ukázky, jak samotná leukokorie vypadá. Klientská část aplikace byla rozdělena do několika komponent, které jsou detailněji popsány v následujících řádcích. Na obrázku 7.5 jsou tyto komponenty zobrazeny včetně jejich vztahů ve formě stromové struktury.



Obrázek 7.5: Komponenty klientské části aplikace Eye Check, jejichž vztahy jsou vyjádřeny pomocí stromové struktury.

- **index** – Vstupní bod aplikace, který je zavolán vždy při startu aplikace.
- **App** – Tato komponenta je zodpovědná za vytvoření *Redux store*. `Provider` zpřístupní *Redux store* všem vnořeným komponentám. Vnořené komponenty následně mohou přistupovat k *Redux store* pomocí funkce `connect()`.
- **MainPage** – Tato komponenta je zodpovědná za inicializaci celé aplikace. Při inicializačním procesu se kontroluje a nastavuje jazyk aplikace.
- **Navigation** – Zodpovědnost za vytvoření navigace v rámci celé aplikace.

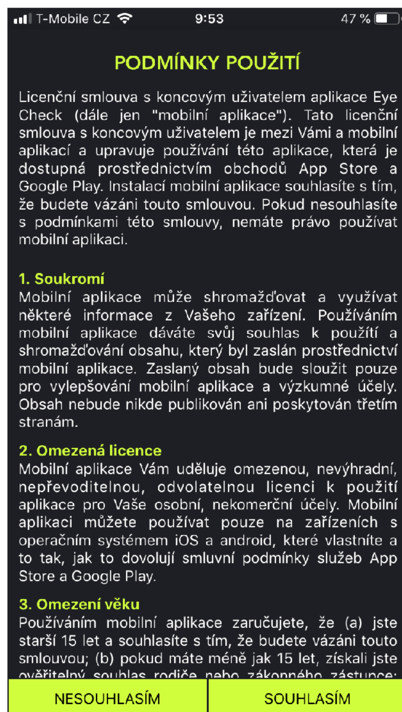
- **LoadingPage** – Tato komponenta je zodpovědná za přepnutí na obrazovku s podmínkami užití aplikace (TermsOfUsePage) nebo na hlavní obrazovku aplikace (HomePage), podle toho, zda-li uživatel přijal podmínky použití. Jelikož je proces zjištění, zda-li uživatel přijal podmínky použití aplikace asynchronní, je uživateli zobrazen `ActivityIndicator` do doby, než přijde odpověď.
- **TermsOfUsePage** – Tato komponenta zobrazuje uživateli podmínky použití mobilní aplikace Eye Check.
- **BottomTabNavigator** – Zodpovědnost za vytvoření spodní navigace aplikace.
- **HomeStack** – Zodpovědnost za vytvoření stacku pro hlavní obrazovku aplikace.
- **HomePage** – Tato komponenta zobrazuje úvodní obrazovku aplikace Eye Check.
- **CameraPage** – Tato komponenta umožňuje uživateli pořídit fotografii pro následnou analýzu na případný výskyt leukokorie.
- **GalleryPage** – Umožňuje uživateli vybrat fotografii z galerie mobilního zařízení pro následnou analýzu na případný výskyt leukokorie.
- **SettingStack** – Zodpovědnost za vytvoření stacku pro nastavení aplikace.
- **SettingPage** – Tato komponenta zobrazuje úvodní obrazovku nastavení.
- **SelectLanguagePage** – Tato komponenta umožňuje uživateli vybrat jazyk aplikace.
- **AboutAppPage** – Tato komponenta zobrazuje informace o aplikaci.
- **LeukocoriaStack** – Zodpovědnost za vytvoření stacku pro obrazovku o leukokorii.
- **LeukocoriaPage** – Tato komponenta zobrazuje úvodní obrazovku o leukokorii.
- **AboutDiseasePage** – Tato komponenta zobrazuje informace a příčiny leukokorie.
- **ExamplesDiseasePage** – Tato komponenta zobrazuje ukázky leukokorie ve formě fotografií.

7.5.1 TermsOfUsePage

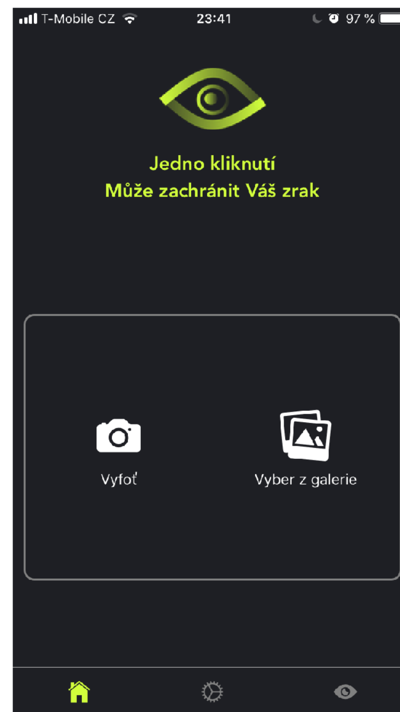
Tato komponenta zobrazuje podmínky použití mobilní aplikace Eye Check a je zobrazena na obrázku 7.6. Uživatel je zde rovněž upozorněn, že aplikace má pouze informativní charakter a v žádném případě neslouží jako plnohodnotné lékařské vyšetření, nespecifikuje příčinu leukokorie, ani nedoporučuje konkrétní léčbu.

7.5.2 HomePage

Tato komponenta zobrazuje úvodní obrazovku aplikace Eye Check a je ukázána na obrázku 7.7. Uživatel si zde může vybrat, zda-li chce fotografii pro následnou analýzu na výskyt leukokorie pořídit přímo v aplikaci, nebo fotografii vybrat z galerie svého zařízení.



Obrázek 7.6: Obrazovka, která uživateli zobrazuje podmínky použití mobilní aplikace Eye Check. Uživatel je zde kromě samotných podmínek upozorněn, že aplikace má pouze informativní charakter a v žádném případě neslouží jako plnohodnotné lékařské vyšetření.



Obrázek 7.7: Úvodní obrazovka aplikace Eye Check. Na této obrazovce si uživatel může vybrat fotografii, která bude dále analyzována na výskyt leukokorie. Fotografii může pořídit přímo v aplikaci, nebo si může fotografii vybrat z galerie zařízení.

Pokud se uživatel rozhodne pořídit fotografii přímo v mobilní aplikaci, dojde k volání metody `goToCameraPage()`. Při prvotním požadavku uživatele o pořízení fotografie v mobilním zařízení požádá aplikace Eye Check o přístup k fotoaparátu. Při každém dalším požadavku je jako první krok v metodě `goToCameraPage()` kontrolováno, zda-li má uživatel povolen přístup k fotoaparátu pro aplikaci Eye Check. Pro kontrolu povolení přístupu jsem využil knihovnu *react-native-permissions*[16], konkrétně knihovní funkci `Permissions.check('camera')`. Pokud má uživatel přístup k fotoaparátu pro mobilní aplikaci Eye Check povolen, je navigován na obrazovku pro pořízení fotografie (CameraPage), v opačném případě je uživateli přístup zamítnut a je o tom informován pomocí dialogového okna.

V případě, že chce uživatel pro následnou analýzu na výskyt leukokorie vybrat fotografii z galerie svého zařízení, dojde k volání metody `chooseFromLibrary()`. Při prvotním požadavku uživatele o přístup ke galerii požádá aplikace Eye Check o přístup k fotografiím. Při každém dalším požadavku je jako první krok v metodě `chooseFromLibrary()` kontrolováno, zda-li má uživatel povolen přístup ke galerii pro aplikaci Eye Check. Pro kontrolu povolení přístupu jsem využil opět knihovnu *react-native-permissions*, konkrétně knihovní funkci `Permissions.check('photo')`. Pokud má uživatel přístup ke galerii pro mobilní aplikaci Eye Check povolen, je navigován na obrazovku pro výběr fotografie z galerie za-

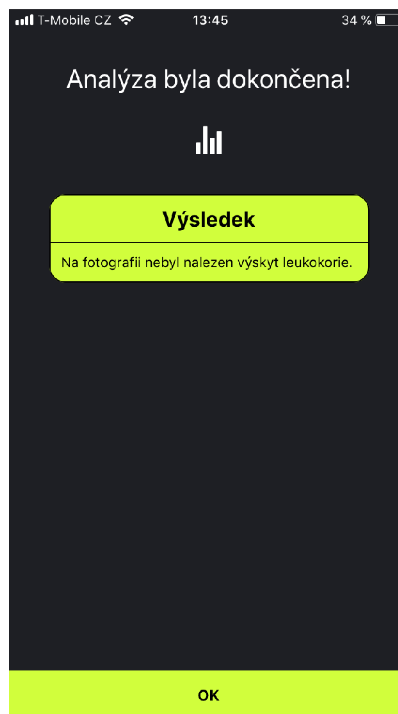
[16]https://github.com/yonahforst/react-native-permissions

řízení, v opačném případě je uživateli přístup zamítnut a je o tom informován pomocí dialogového okna. Pro přístup ke galerii jsem využil knihovnu *react-native-image-picker*[17], konkrétně knihovní funkci `ImagePicker.launchImageLibrary()`. Důležitým parametrem této funkce je parametr `quality`. Tento parametr určuje, jestli má být vybraná fotografie nějak komprimována za cílem zmenšení její velikosti, nebo má zůstat v původní kvalitě a velikosti. Hodnota parametru je v rozmezí 0 (nejnižší kvalita) až 1 (zachování původní kvality a velikosti). Parametr `quality` jsem nastavil na hodnotu 1, abych pro následné zpracování a analýzu pracoval s co nejkvalitnější fotografií.

Jakmile uživatel pořídí fotografii přímo v mobilní aplikaci, nebo fotografii vybere z galerie zařízení, dojde k zobrazení fotografie, což je vidět na obrázku 7.8. V případě, že uživatel vybral špatnou fotografii nebo chce vybranou fotografii změnit, může využít tlačítko „Jiná fotografie“. Po stisku tlačítka je uživateli znovu zobrazena nabídka pro pořízení fotografie v aplikaci, nebo pro vybrání fotografie z galerie zařízení, jak je ukázáno na obrázku 7.7.

Tlačítko „Zkontrolovat“ odešle fotografii k analýze na případný výskyt leukokorie. Po jeho stisknutí dojde k volání funkce `checkEyes()`, která vybranou fotografii odešle na server, počká na odpověď a následně zobrazí uživateli výsledek analýzy. Na obrázku 7.9 je ukázán výsledek analýzy, která nenašla výskyt leukokorie na uživatelem vybrané fotografii. V případě, že je na vybrané fotografii nalezen výskyt leukokorie, je uživateli doporučena návštěva lékaře.

Obrázek 7.8: Obrazovka, která je uživateli zobrazena po pořízení fotografie v mobilní aplikaci, nebo po vybrání fotografie z galerie zařízení. Uživatel zde může vybrat jinou fotografii, nebo vybranou fotografii přímo analyzovat.



Obrázek 7.9: Obrazovka ukazující výsledek analýzy, která nenašla výskyt leukokorie na uživatelem vybrané fotografii.

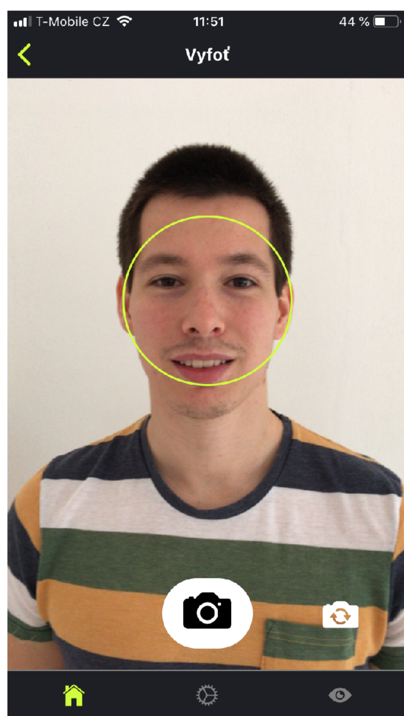
[17] https://github.com/react-native-community/react-native-image-picker

7.5.3 CameraPage

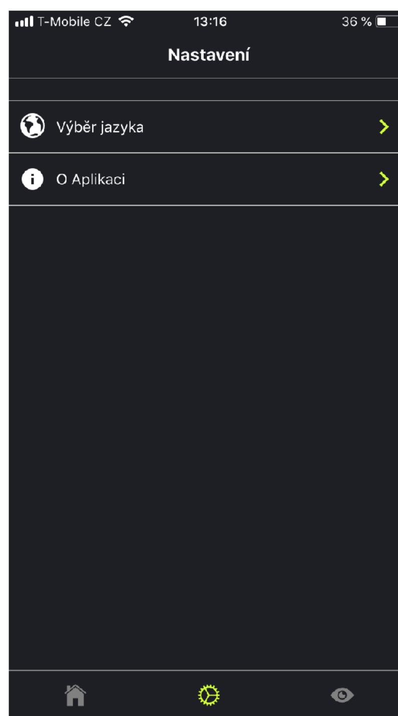
Tato komponenta umožňuje pořídit fotografii pro následnou analýzu případného výskytu leukokorie přímo v mobilní aplikaci a je ukázána na obrázku 7.10. Pro možnost pořízení fotografie jsem využil knihovnu *react-native-camera*[18]. Uživatel může pořídit fotografii jak předním, tak zadním fotoaparátem. Pro přesnější analýzu je však doporučeno použít zadní fotoaparát s bleskem a pořizovat fotografii ve tmě nebo za snížených světelných podmínek, aby se projevil tzv. *efekt červených očí*, protože v takovém případě je leukokorie velmi dobře rozpoznatelná.

Jakmile uživatel zmáčkne ikonu fotoaparátu pro pořízení fotografie, dojde k volání metody `takePicture()`. Důležitým krokem je nastavení parametru `quality`. Tento parametr určuje, jestli má být pořízená fotografie nějak komprimována za cílem zmenšení její velikosti. Hodnota parametru je v rozmezí 0 (nejnižší kvalita) až 1 (nejvyšší kvalita). Parametr `quality` jsem nastavil na hodnotu 1, abych pro následné zpracování a analýzu pracoval s co nejkvalitnější fotografií.

Po pořízení fotografie je uživateli fotografie zobrazena, a právě pořízenou fotografii může ihned analyzovat na případný výskyt leukokorie, jak je ukázáno na obrázku 7.8.



Obrázek 7.10: Obrazovka „Vyfoť“, která umožňuje uživateli pořídit fotografii přímo v mobilní aplikaci. Uživatel může použít přední i zadní fotoaparát.



Obrázek 7.11: Obrazovka „Nastavení“ umožňující uživateli přejít na obrazovku pro výběr jazyka aplikace, nebo přejít na obrazovku s informacemi o aplikaci Eye Check.

[18]https://github.com/react-native-community/react-native-camera

7.5.4 SettingPage

Tato komponenta je ukázána na obrázku 7.11 a umožňuje uživateli přejít na obrazovku pro výběr jazyka aplikace, nebo přejít na obrazovku s informacemi o aplikaci Eye Check.

7.5.5 SelectLanguagePage

Tato komponenta je ukázána na obrázku 7.12 a umožňuje uživateli zvolit jazyk aplikace Eye Check. Uživatel má na výběr mezi češtinou a angličtinou. K dosáhnutí multijazyčnosti celé aplikace jsem použil knihovnu *react-native-redux-i18n*[19]. Jakmile uživatel zvolí vybraný jazyk, dojde k volání metody `setLanguage()`. V této metodě dojde k vyvolání *Redux* akce, která změní stav v *Redux* store. Díky tomu, že ostatní komponenty jsou k *Redux store* připojeny pomocí funkce `connect()`, dojde ke změně jazyka ve všech těchto komponentách.

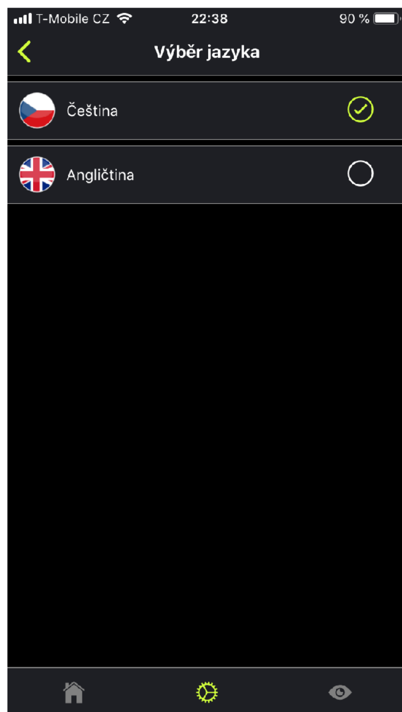
Aby bylo možné při znovu spuštění aplikace uživateli zobrazit naposledy vybraný jazyk, je potřeba aktuálně zvolený jazyk perzistovat. K perzistenci naposledy zvoleného jazyka jsem využil `API` knihovny *React Native* `AsyncStorage`[20]. Při každém spuštění aplikace je při inicializaci aplikace získána hodnota z tohoto úložiště a následně nastaven jazyk pro celou aplikaci. Při prvotním spuštění aplikace je pro uživatele, kteří mají jako systémový jazyk mobilního zařízení nastavenou češtinu, spuštěna aplikace Eye Check v českém jazyce, pro všechny ostatní je aplikace spuštěna v jazyce anglickém. Pro zjištění systémového jazyka uživatele jsem použil knihovnu *react-native-device-info*[21], konkrétně knihovní funkci `getDeviceLocale()`.

7.5.6 AboutAppPage

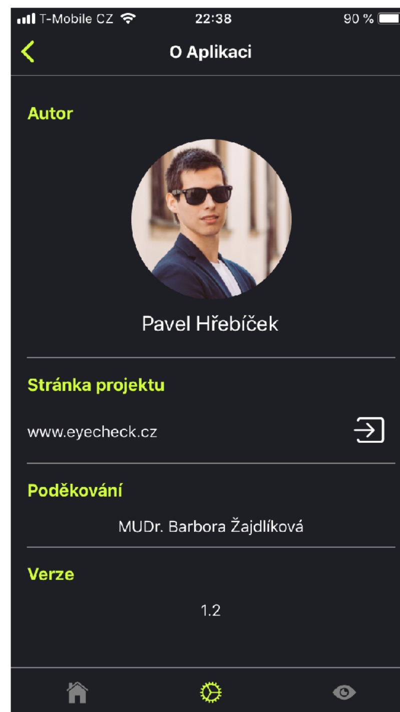
Tato komponenta je ukázána na obrázku 7.13 a umožňuje uživateli získat následující informace o aplikaci Eye Check:

- **Jméno autora aplikace** – Pavel Hřebíček.
- **Stránka projektu** – Jedná se o webovou aplikaci[22], která byla vytvořena za prezentačním účelem mobilní aplikace. Uživatel se zde dozví veškeré informace o leukokorii a o samotné aplikaci Eye Check, včetně odkazů na *App Store* a *Google Play*.
- **Poděkování** – Zde je zmíněna MUDr. Barbora Žajdlíková, která na celé práci spolupracovala. V případě jakýchkoliv informací nebo dotazů ji může uživatel kontaktovat.
- **Verzi aplikace** – Zobrazí aktuální verzi aplikace.

[19] `https://github.com/derniercri/react-native-redux-i18n`
[20] **AsyncStorage** je jednoduché API pro ukládání dat ve formě klíč – hodnota.
[21] `https://github.com/react-native-community/react-native-device-info`
[22] `https://eyecheck.cz/`



Obrázek 7.12: Obrazovka „Výběr jazyka“, která umožňuje uživateli zvolit jazyk aplikace Eye Check. Uživatel má na výběr mezi češtinou a angličtinou.



Obrázek 7.13: Obrazovka „O Aplikaci“ umožňující uživateli získat veškeré informace o aplikaci Eye Check.

7.5.7 LeukocoriaPage

Tato komponenta umožňuje uživateli získat základní informaci o leukokorii. Komponenta obsahuje dvě vnořené komponenty – AboutDiseasePage a ExamplesDiseasePage. Mezi těmito komponentami může uživatel přepínat pomocí gesta *swipe*. K vytvoření jsem použil knihovnu *react-native-scrollable-tab-view*[23].

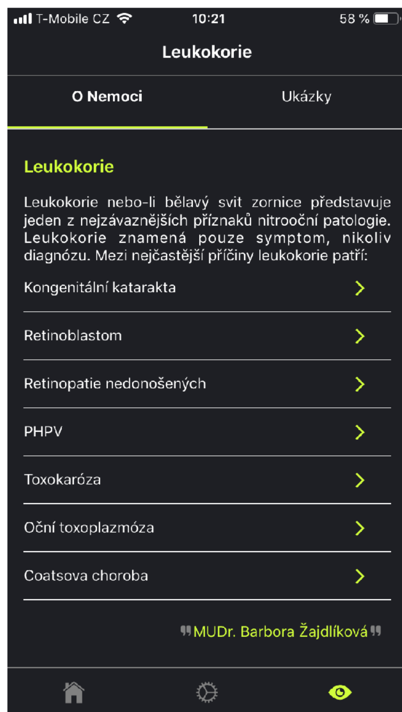
Komponenta AboutDiseasePage je ukázána na obrázku 7.14. Umožňuje uživateli získat základní informace jak o leukokorii samotné, tak o příčinách leukokorie. Ke každé příčině je napsán krátký informační text. Komponenta ExamplesDiseasePage je ukázána na obrázku 7.15 a obsahuje ukázky leukokorie, aby si uživatel dokázal tento symptom lépe představit.

7.6 Implementace serverové části

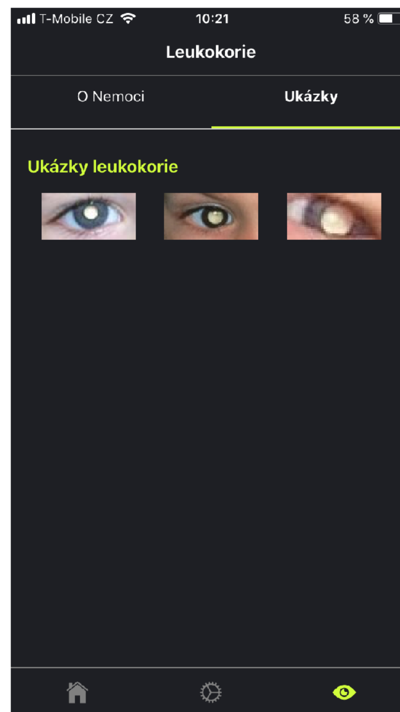
Serverová část mobilní aplikace je zodpovědná za detekci obličeje, očí a následnou klasifikaci získaných výřezů očí na případný výskyt leukokorie. Klasifikace je prováděna pomocí *konvoluční neuronové sítě*, která provede predikci výsledku pro každé oko. Konečný výsledek je poslán zpět k uživateli na jeho mobilní zařízení a konečný výsledek je přehledně zobrazen.

Pro hosting serverové části jsem zvolil službu *pythonanywhere*[24], kde jsem nainstaloval veškeré knihovny potřebné k detekci obličeje, očí a rozpoznání leukokorie. Mezi tyto knihovny patří:

[23] https://github.com/ptomasroos/react-native-scrollable-tab-view
[24] https://www.pythonanywhere.com



Obrázek 7.14: Obrazovka „O Nemoci“, která umožňuje uživateli získat základní informace jak o leukokorii samotné, tak o příčinách leukokorie.



Obrázek 7.15: Obrazovka „Ukázky“ obsahuje ukázky leukokorie, aby si uživatel dokázal tento symptom lépe představit.

- **Django** (verze 2.1.2) – Webový aplikační framework, nutný pro zprovoznění *Django REST frameworku.*
- **djangorestframework** (verze 3.9.0) – Framework pro tvorbu `REST API`.
- **django-cors-headers** (verze 3.0.1) – Přidává `CORS` hlavičky k odpovědím. Využívá se u webové aplikace, která je blíže popsána v kapitole 9.
- **dlib** (verze 19.16.0) – Knihovna potřebná pro detekci obličeje a očí z fotografie.
- **opencv-python** (verze 3.4.3.18) – Knihovna pro práci s fotografií.
- **imutils** (verze 0.5.1) – Knihovna pro provádění jednoduchých úloh při zpracování obrazu. Na serveru jsem tuto knihovnu použil pro změnu velikosti obrazu.
- **Keras** (verze 2.2.4) – Knihovna, která nabízí vysokoúrovňové `API` pro práci s neuronovými sítěmi. Je schopná běžet na backendu *Tensorflow*[25], *CNTK*[26] nebo *Theano*[27].
- **tensorflow** (verze 1.13.1) – Knihovna, která se používá pro práci s neuronovými sítěmi. Na serveru jsem využil *tensorflow* jako backend, na kterém běží knihovna *Keras*.

[25] https://www.tensorflow.org
[26] https://docs.microsoft.com/en-us/cognitive-toolkit/
[27] http://www.deeplearning.net/software/theano/

7.6.1 Django REST framework

Jedná se o velmi mocný a flexibilní nástroj pro tvorbu `REST API`, který jsem využil v této práci pro komunikaci mezi klientem a serverem. *Django REST framework* je nadstavbou frameworku *Django*[28] [14]. K vytvoření `REST API` je potřeba [19]:

- Vytvořit koncové body (*Endpoints*).
- Zadefinovat `URL` adresu, ze které bude mít uživatel přístup ke koncovému bodu.

Pro účely této práce jsem se rozhodl vytvořit pouze jeden koncový bod, kde lze zobrazit informace o všech fotografiích, které byly odeslány na analýzu, nebo vytvořit nový záznam o nově příchozí fotografii. Ukázka vytvoření koncového bodu v *Django REST frameworku* je zobrazena na výpisu 7.2. K vytvoření koncového bodu jsem použil dekorátor `@api_view()`, který je vidět na řádku 1 výpisu 7.2. Parametrem tohoto dekorátoru je seznam `HTTP` metod, na které by měl koncový bod odpovídat. V případě této práce se jedná o parametry `GET` a `POST`, jelikož koncový bod pouze získává informace, nebo vytváří nový záznam. V mobilní aplikaci Eye Check se používá pouze metoda `POST` pro odesílání fotografie na analýzu. Metoda `GET` se používá v administrační části webové aplikace, která byla implementována jako rozšíření celé této práce a je více popsána v kapitole 9, pro zobrazení statistického údaje o počtu doposud analyzovaných fotografií.

Na řádku 2 výpisu 7.2 je použit další dekorátor `@parser_classes()`, který *Django REST frameworku* říká, jaký má očekávat `Content-Type` v příchozím `HTTP` požadavku. Jelikož se při `POST` požadavku zasílá fotografie k analýze a `HTTP` požadavek má nastavený `Content-Type` na *multipart/form-data*, je jako parametr dekorátoru `@parser_classes()` nastaven `MultiPartParser`. Definování koncových bodů vytvářeného `REST API` se typicky provádí v souboru s názvem `views.py`.

```
1  @api_view(["GET", "POST"])
2  @parser_classes((MultiPartParser, ))
3  def image_list(request):
4    """
5    Seznam informaci o vsech fotografiich nebo vytvoreni noveho zaznamu
6    """
7    if request.method == "GET":
8      # Zpracovani GET pozadavku
9    elif request.method == "POST":
10     # Zpracovani POST pozadavku
```

Výpis 7.2: Ukázka vytvoření koncového bodu pro `REST API` mobilní aplikace Eye Check v *Django REST frameworku*, který slouží pro zobrazení informací o všech fotografiích, které byly odeslány na analýzu, nebo pro vytvoření nového záznamu o nově příchozí fotografii. Pro vytvoření koncového bodu byl použit dekorátor `@api_view()` s parametry `GET` a `POST`, jelikož koncový bod pouze získává informace nebo vytváří nový záznam.

Definování `URL` adresy, ze které bude mít uživatel přístup ke koncovému bodu pro získání informací o všech fotografiích, nebo zaslání fotografie na analýzu, je ukázáno na výpisu 7.3. Na řádku 6 výpisu 7.3 je ukázána definice `URL` cesty. Funkce `path()` bere jako první argument cestu a druhý argument obsahuje koncový bod. Koncový bod pro mobilní aplikaci Eye Check bude mít tedy `URL` adresu https://moje_domena.com/images/ a bude využívat koncový bod `image_list()`, který je definován na řádku 3 výpisu 7.2.

[28] https://www.djangoproject.com

```
1 from django.urls import path
2 from rest_framework.urlpatterns import format_suffix_patterns
3 from server import views
4
5 urlpatterns = [
6     path("images/", views.image_list),
7 ]
8
9 urlpatterns = format_suffix_patterns(urlpatterns)
```

Výpis 7.3: Ukázka vytvoření `URL` adresy pro mobilní aplikaci Eye Check pomocí *Django REST frameworku*, ze které bude mít uživatel přístup ke koncovému bodu pro získání informací o všech fotografiích, nebo zaslání fotografie na analýzu. O definici `URL` adresy se stará funkce `path()`, která bere jako první argument cestu a druhý argument obsahuje koncový bod.

7.6.2 Proces analýzy fotografie na případný výskyt leukokorie

Pro lepší přehled čtenáře jsem na výpisu 7.4 uvedl pseudokód, který ukazuje jednotlivé kroky, které se provádějí na straně serveru od příchodu `HTTP` požadavku až po samotnou odpověď. V následujících řádcích popíši jednotlivé kroky a řádky kódu výpisu 7.4.

Jak bylo již výše zmíněno, mobilní aplikace Eye Check používá pro odeslání fotografie na server HTTP metodu `POST`. Na řádcích 6 – 7 se provádí kontrola příchozího požadavku. Pokud se jedná o validní požadavek, je dále zpracováván, v opačném případě se vrací `HTTP` stavový kód 400 (*Bad Request*). Na řádcích 9 – 10 se získává detektor a prediktor z knihovny *Dlib*, který slouží k detekci obličeje a očí. Na řádcích 13 – 15 se načítá obrázek, který přišel v `HTTP` požadavku a převádí se do odstínu šedi, jelikož k samotné detekci obličeje není zapotřebí barevné fotografie.

Na řádku 17 se knihovna *Dlib* pokouší detekovat obličej v uživatelem poslané fotografii. Na řádcích 19 – 23 se provádí kontrola fotografie. Pro samotnou analýzu je vyžadována fotografie, kde je obličej člověka, a zároveň pouze jeden obličej. Pokud nějaká z těchto podmínek není splněna, vrací se `HTTP` stavový kód 400 (*Bad Request*) a uživatel je o této skutečnosti informován přímo v mobilní aplikaci.

Na řádku 25 dochází k načtení *Keras* modelu, který obsahuje celou architekturu a váhy natrénovaného modelu konvoluční neuronové sítě. Na řádcích 28 – 29 se získává obalový obdélník (bounding box) pravého, resp. levého oka. Na řádcích 32 – 33 se získaný výřez pravého, resp. levého oka dá na vstup neuronové sítě. Natrénovaný model následně provede predikci pro obě oči a výsledek uloží do příslušných proměnných.

Pokud natrénovaný model predikuje na pravém, resp. levém oku možnost leukokorie, vrátí server `HTTP` stavový kód 201 (*Created*) s kódem LECORIA. Výsledek je uživateli následně přehledně zobrazen v mobilním zařízením a je mu doporučena návštěva lékaře. V případě, že natrénovaný model nenajde v pravém ani levém oku symptom leukokorie, vrátí server `HTTP` stavový kód 201 (*Created*) s kódem NO_LECORIA. Výsledek je uživateli následně taktéž přehledně zobrazen v mobilním zařízením.

```
def image_list(request):
  if request.method == "GET":
    # vrati seznam informaci o vsech analyzovanych fotografiich
    return Response(all_images)
  elif request.method == "POST":
    req = process_request(request.data) # zpracuj prichozi pozadavek
    if req.is_valid():
      # ziskani detektoru a prediktoru pro detekci obliceje a oci
      detector = dlib.get_frontal_face_detector()
      predictor = dlib.shape_predictor("shape_predictor_68_face_landmarks.dat")

      # nacteni obrazku
      img = cv2.imread(req.data["image"])
      # prevod obrazku do odstinu sedi
      gray_scale_img = cv2.cvtColor(img, cv2.COLOR_BGR2GRAY)

      faces = detector(gray_scale_img, 1) # detekce obliceje

      if len(faces) == 0:
          return Response({"code": "NO_FACE_DETECT"}, status=400)

      if len(faces) > 1:
          return Response({"code": "MORE_FACES_DETECT"}, status=400)

      model = load_model("eyecheck_model.h5") # nacteni modelu keras

      # ziskani obaloveho obdelniku leveho a praveho oka
      right_eye_bb = get_eye_bb(img)
      left_eye_bb = get_eye_bb(img)

      # predikce ziskanych vyrezu oci
      right_eye_bb_result = model.predict_classes(right_eye_bb)
      left_eye_bb_result = model.predict_classes(left_eye_bb)

      if right_eye_bb_result == "LECORIA" or left_eye_bb_result == "LECORIA":
        return Response({"code": "LECORIA"}, status=status.HTTP_201_CREATED)
      else:
        return Response({"code": "NO_LECORIA"}, status=status.HTTP_201_CREATED)
    return Response(serializer.errors, status=status.HTTP_400_BAD_REQUEST)
```

Výpis 7.4: Pseudokód, který ukazuje jednotlivé kroky, které se provádějí na straně serveru od příchodu HTTP požadavku až po samotnou odpověď.

7.7 Klient-Server komunikace

Jakmile uživatel vybere ze svého zařízení fotografii, kterou chce analyzovat na výskyt leukokorie a zmáčkne tlačítko pro odeslání, dojde k poslání HTTP POST požadavku na server. Jelikož klient posílá na server fotografii, je v HTTP hlavičce požadavku jako Content-Type nastavena hodnota *multipart/form-data*. K vytvoření samotného těla požadavku je použit *FormData* objekt.

Jak již bylo zmíněno výše, metoda GET se používá v administrační části webové aplikace, která byla implementována jako rozšíření celé této práce a je více popsána v kapitole 9,

pro zobrazení statistického údaje o počtu doposud analyzovaných fotografií. Všechny dostupné koncové body (*endpoints*) `REST API`, které byly k mobilní a webové aplikaci Eye Check vytvořeny jsem shrnul do tabulky 7.3.

URI	Metoda	`HTTP` hlavička	Parametry
/images/	`GET`	Accept: "application/json" Content-Type: -	-
/images/	`POST`	Accept: "application/json" Content-Type: "multipart/form-data"	image: [string] title: [string] deviceId: [string]

Tabulka 7.3: Shrnutí všech dostupných koncových bodů (*endpoints*) vytvořeného `REST API` pro mobilní a webovou aplikaci Eye Check.

Server zpracuje požadavek a následně odpovídá klientovi pomocí formátu `JSON` jedním z následujících kódů:

- **NO_FACE_DETECT** – Na obrázku nebyl detekován žádný obličej. Je potřeba nahrát novou fotografii.
- **MORE_FACES_DETECT** – Na obrázku bylo detekováno více obličejů. Je potřeba nahrát novou fotografii.
- **NO_LECORIA** – Na obrázku nebyla detekována leukokorie.
- **LECORIA** – Na obrázku byla detekována leukokorie.

7.8 Dostupnost

Jak již bylo zmíněno, jeden z hlavních cílů této práce je poskytnout mobilní aplikaci co nejširšímu spektru uživatelů. V aplikaci je tedy dostupná multijazyčnost a uživatel si může vybrat mezi češtinou a angličtinou. Do budoucna se plánuje rozšíření o další jazyky.

Pro operační systém *iOS* je mobilní aplikace dostupná pro verzi *iOS 9.0* a novější. Na zařízeních s operačním systémem *Android* je aplikace dostupná pro verzi *Android 4.1 (Jelly Bean)* a novější. Mobilní aplikace Eye Check je dostupná pro veřejnost jak v *App Store*[29], tak na *Google Play*[30].

[29] https://itunes.apple.com/cz/app/eye-check/id1454515853?l=cs&mt=8
[30] https://play.google.com/store/apps/details?id=com.pavelhrebicek.eyecheck

Kapitola 8

Testování mobilní aplikace

Testování mobilní aplikace Eye Check se uskutečnilo pro obě dostupné platformy – *iOS* a *Android*. Pro operační systém *iOS* byla výsledná aplikace nahrána na *App Store Connect*[1]. Uživatelé následně prováděli testy na svých zařízeních pomocí aplikace *TestFlight*[2]. Pro operační systém *Android* byla aplikace testována pomocí *Google Play Console*[3], kde byla aplikace nahrána do tzv. *alfa kanálu*, který je určen k testování. Testerům byl následně odeslán odkaz k nainstalování aplikace, po jehož otevření byla aplikace nainstalována do uživatelova mobilního zařízení. Kromě výše zmíněných dostupných platforem a procesů pro testování byla celá aplikace Eye Check otestována účastníky na studentské konferenci Excel@FIT 2019.

Největší problém, který se při testování aplikace Eye Check objevil, souvisel s pořizováním a výběrem fotografie pro samotnou analýzu. Mnoho uživatelů pořizovalo fotografii přímo detailu oka nebo očí. Tento nalezený problém jsem se rozhodl vyřešit pomocí grafické vizualizace. Konkrétně jsem do obrazovky pro pořizování fotografie přidal geometrický útvar kruh, který lze vidět na obrázku 8.1. Uživatel se tedy snaží vmístit svůj obličej do tohoto kruhu, což má za výsledek, že fotografie je správně vyfocena ve smyslu detailu fotografie a připravena k analýze. Aby byl uživatel obeznámen, z jakého důvodu je při pořizování fotografie ukázána tato grafická vizualizace, a jak nejlépe pořídit fotografii pro co nejlepší přesnost výsledku, vytvořil jsem průvodce. Tento průvodce je vidět na obrázku 8.2 a zobrazí se uživateli vždy před pořízením, nebo vybráním fotografie. Uživatel tak dostane přesné instrukce, jak a za jakých podmínek je nejvhodnější samotnou fotografii pořídit, nebo vybrat. Pokud je uživatel seznámen s informacemi, může zvolit možnost nezobrazovat a dále se již průvodce nebude ukazovat.

Další věcí, na kterou upozornilo mnoho testovaných uživatelů, byla nemožnost pořídit fotografii předním fotoaparátem. Tato možnost nebyla do prvních verzí mobilní aplikace dána záměrně ze dvou hlavních důvodů. Tím prvním je, že pro analýzu je potřeba co nejkvalitnější fotografie. Některé starší modely přední fotoaparát nemají nebo je jeho kvalita velmi nízká. Druhým důvodem je, že pro rozpoznání leukokorie je ve většině případů vyžadován blesk, který také není u všech zařízení, které přední fotoaparát mají. Po konzultaci s MUDr. Barborou Žajdlíkovou jsem se nakonec rozhodl pro implementování předního fotoaparátu. Paní doktorka mi sdělila [42], že někdy jde leukokorie vidět i makroskopicky bez jakéhokoliv nasvícení. Uživatelům je však doporučeno používat zadní fotoaparát s bleskem, což je vidět na obrázku 8.2.

[1] https://appstoreconnect.apple.com

[2] https://testflight.apple.com

[3] https://developer.android.com/distribute/console

Obrázek 8.1: Obrazovka, kde může uživatel pořizovat fotografii pro následnou analýzu na případný výskyt leukokorie byla v rámci testování rozšířena o grafickou vizualizaci kruhu. Důvodem vytvoření této grafické vizualizace bylo, že uživatel nevěděl, jak moc detailní fotografii pořizovat. Nyní se uživatel snaží vmístit svůj obličej do tohoto kruhu, což má za výsledek, že fotografie je správně vyfocena ve smyslu detailu fotografie a připravena k analýze.

Obrázek 8.2: Obrazovky ukazující průvodce, který byl vytvořen v rámci testování. Průvodce se uživateli zobrazí vždy před pořízením, nebo vybráním fotografie. Uživatel tak dostane přesné instrukce, jak a za jakých podmínek je nejvhodnější samotnou fotografii pořídit, nebo vybrat. Pokud je uživatel seznámen s informacemi, může zvolit možnost nezobrazovat a dále se již průvodce nebude ukazovat.

Kapitola 9

Rozšíření práce o webovou aplikaci

Jako rozšíření této práce byla pro mobilní aplikaci vytvořena i responzivní multijazyčná webová aplikace[1], kterou jsem vytvořil ve frameworku *Nette*[2]. K docílení responzivního webu byl použit framework *Bootstrap*[3]. První část této kapitoly se zabývá klientskou částí webové aplikace. Ve druhé části je následně popsána část administrační.

9.1 Klientská část

Jedná se o webovou prezentaci mobilní aplikace Eye Check. Webová stránka je plně responzivní a stejně jako aplikace mobilní je multijazyčná. Uživatel se zde dozví informace o leukokorii a příčinách leukokorie. Dále je zde vysvětlen princip, jak celá aplikace funguje a jsou zde uvedeny odkazy pro stažení aplikace na *App Store* a *Google Play*. Dále jsou zde uvedeny kontakty na autora projektu a na MUDr. Barboru Žajdlíkovou, která na celém projektu spolupracovala. Uživatel tak může v případě jakýchkoliv dotazů, nalezených chyb či nápadů na vylepšení mobilní nebo webové aplikace kontaktovat přímo autora projektu. V případě jakýchkoliv dotazů ohledně leukokorie může potom kontaktovat přímo paní doktorku Žajdlíkovou.

Webová stránka má v této době pouze prezentační charakter. Do budoucna se plánuje rozšířit její účel o možnost detekovat leukokorii přímo z webového prohlížeče a vytvořit tak ekosystém mobilní – webová aplikace. Uživatel by tak mohl analyzovat fotografii stejným způsobem jak v mobilní aplikaci, ale s využitím webového prohlížeče. Dalším možným vylepšením do budoucna by mohlo být zpracování celého alba fotografií uživatele. Pokud by například přijel uživatel z dovolené, mohl by pořízené fotografie archivovat např. do formátu `ZIP`. Tento soubor by následně nahrál do webového prohlížeče a došlo by k aplikování algoritmu pro analýzu leukokorie pro každou fotografii. Následně by byl uživateli zobrazen výsledek celé analýzy.

Zmíněné možné rozšíření celé této práce, které jsem popsal výše, bylo jedním z hlavních důvodů, proč jsem se rozhodl samotnou analýzu provádět na straně serveru a využít architekturu `REST`. Chtěl jsem využít co největší míry znovupoužitelnosti a s již existující `REST` architekturou je toto rozšíření do budoucna snadnou záležitostí.

---

[1] https://eyecheck.cz/
[2] https://nette.org
[3] https://getbootstrap.com

9.1.1 Multijazyčnost

Jelikož je mobilní aplikace Eye Check dostupná v češtině a angličtině, rozhodl jsem se pro vytvoření multijazyčného obsahu i v případě klientské části aplikace webové. Uživatel si tak může veškeré informace o leukokorii nebo samotné aplikaci Eye Check přečíst jak v českém, tak anglickém jazyce. K implementaci multijazyčnosti jsem použil knihovnu *Kdyby/Translation*[4].

9.2 Administrační část

Administrační část webové aplikace slouží pro rychlou a snadnou úpravu obsahu klientské části webové aplikace. Obsah lze upravovat jak pro českou, tak anglickou verzi klientské části. Vytvořil jsem tedy pomocí frameworku *Nette* vlastní systém pro správu obsahu (CMS), který je plně responzivní a lze vidět na obrázku 9.1.

Kromě samotné správy obsahu zde uživatel může zobrazit např. statistický údaj o počtu fotografií, jež byly analyzovány prostřednictvím mobilní aplikace. Administrační část webové aplikace jsem vytvářel s vizí, že by se v budoucnu mohl najít člověk, který by stránku o leukokorii spravoval. Z tohoto důvodu jsem do administrační části naimplementoval možnost správy uživatelů a rolí. Je tedy možné provádět základní operace, mezi které patří vytvoření, zobrazení, editace a smazání uživatele.

Obrázek 9.1: Ukázka administrační části webové aplikace sloužící pro rychlou a snadnou úpravu obsahu klientské části webové aplikace. Administrační část je plně responzivní a umožňuje tak uživateli pohodlnou úpravu obsahu i z mobilního zařízení.

[4] https://github.com/Kdyby/Translation

Kapitola 10

Závěr

Cílem této práce bylo navrhnout a implementovat multiplatformní multijazyčnou mobilní aplikaci pro detekci leukokorie ze snímku lidského obličeje pro platformy *iOS* a *Android*. Důležitou součástí celé této práce bylo vyhledání odborníka z oboru očního lékařství a následné konzultace ohledně symptomu leukokorie, které vedly ke zdokonalení celého díla.

V teoretické části této práce byly nastíněny, analyzovány a porovnány metody, které se používají k detekci objektů, obličeje a očí z fotografie. Na základě této analýzy byla pro detekci obličeje a očí vybrána knihovna *Dlib*. Pro práci s fotografiemi byla použita knihovna *OpenCV*. K detekci leukokorie byly experimentálně vyzkoušeny 2 metody. Na základě analýzy a vyhodnocení byla následně použita metoda využívající *konvoluční neuronovou síť*. Tato síť byla vytvořena pomocí knihovny *Keras*, běžící na backendu *Tensorflow*. Metoda byla vyhodnocena na testovací datové sadě s úspěšností 98,14 %. V rámci implementace vznikl nástroj pro pořizování vhodné datové sady, který výrazně urychlil proces trénování sítě.

Před samotným vývojem mobilní aplikace byl proveden návrh uživatelského rozhraní a architektury. Uživatelské rozhraní bylo navrhnuto, otestováno a vyhodnoceno. Pro komunikaci mezi klientem a serverem byla zvolena architektura `REST`. Před implementací mobilní aplikace byly porovnány frameworky a technologie, které v dnešní době umožňují vývoj multiplatformní mobilní aplikace. K implementaci mobilní aplikace byl na základě této analýzy vybrán framework *React Native*.

Výsledkem této práce je multiplatformní multijazyčná mobilní aplikace pro platformy *iOS* a *Android*, která je dostupná v *App Store* a na *Google Play*. Aplikaci si stáhlo a nainstalovalo 56 uživatelů se systémem *iOS* a 36 uživatelů se systémem *Android* po celém světě. Mobilní aplikace uživateli umožňuje nahrát fotografii a následně ji nechat analyzovat na případný výskyt leukokorie. V případě detekce leukokorie je uživatel o této skutečnosti informován a je mu doporučena návštěva lékaře. Celá aplikace Eye Check byla otestována jak pro platformu *iOS*, tak *Android* pomocí dostupných testovacích nástrojů a procesů. Další fáze testování aplikace proběhla na studentské konferenci Excel@FIT 2019.

Jako rozšíření této práce byla vytvořena responzivní multijazyčná webová aplikace obsahující jak klientskou, tak administrační část. Do budoucna se plánuje rozšířit její účel o možnost detekovat leukokorii přímo z webového prohlížeče a vytvořit tak ekosystém mobilní – webová aplikace. Dalším plánem je rozšíření lokalizace mobilní aplikace o další jazyky.

Tato práce byla přijata, ústně prezentována a oceněna odborným panelem na studentské konferenci Excel@FIT 2019.

Literatura

[1] What is REST? [Online; navštíveno 26.11.2018].
URL https://www.codecademy.com/articles/what-is-rest

[2] Anzalone, L.: Training alternative Dlib Shape Predictor models using Python. *MEDIUM.com*, Říjen 2018.

[3] Arnold, J.: Dumb Components and Smart Components. *MEDIUM.com*, Únor 2017.

[4] Bradski, G.: The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.

[5] Dalal, N.; Triggs, B.: Histograms of Oriented Gradients for Human Detection. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR 2005), 20-26 June 2005, San Diego, CA, USA*, 2005, s. 886–893.

[6] D'Souza, J.: A Quick Guide to Boosting in ML. *MEDIUM.com*, Březen 2018.

[7] Dua, K.: A Guide to Mobile App Development: Web vs. Native vs. Hybrid. 2018, [Online; navštíveno 11.1.2019].
URL https://clearbridgemobile.com/mobile-app-development-native-vs-web-vs-hybrid/

[8] Esposito, E.: Low-fidelity vs. high-fidelity prototyping. 2018, [Online; navštíveno 16.1.2019].
URL https://www.invisionapp.com/inside-design/low-fi-vs-hi-fi-prototyping/

[9] Fielding, R. T.: *Architectural Styles and the Design of Network-based Software Architectures*. Dizertační práce, UNIVERSITY OF CALIFORNIA, 2000.

[10] Fielding, R. T.; Gettys, J.; Mogul, J. C.; aj.: Hypertext Transfer Protocol – HTTP/1.1. RFC 2616, RFC Editor, 1999, [Online; navštíveno 10.1.2019].
URL http://www.rfc-editor.org/rfc/rfc2616.txt

[11] Frachet, M.: Understanding the React Native bridge concept. 2018, [Online; navštíveno 10.1.2019].
URL https://hackernoon.com/understanding-react-native-bridge-concept-e9526066ddb8

[12] Goodfellow, I.; Bengio, Y.; Courville, A.: *Deep Learning*. MIT Press, 2016, http://www.deeplearningbook.org.

[13] Gupta, V.: Face Detection – OpenCV, Dlib and Deep Learning (C++ / Python). 2018, [Online; navštíveno 2.4.2019].
URL https://www.learnopencv.com/face-detection-opencv-dlib-and-deep-learning-c-python/

[14] Hrončok, M.: *RESTful API v jazyce Python*. Diplomová práce, České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

[15] Huffine, T.: componentDidMakeSense – React Component Lifecycle Explanation. *MEDIUM.com*, Říjen 2017.

[16] Hřebíček, P.: *Mobilní aplikace pro anotace obrázků*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2016.

[17] Jöch, D.: Functional vs Class-Components in React. *MEDIUM.com*, Červenec 2018.

[18] King, D. E.: Dlib-ml: A Machine Learning Toolkit. *Journal of Machine Learning Research*, ročník 10, 2009: s. 1755–1758.

[19] Kings, E.: Creating a Django API using Django Rest Framework APIView. *MEDIUM.com*, Listopad 2018.

[20] Králová, M.: Lidské oko. [Online; navštíveno 8.4.2019].
URL https://edu.techmania.cz/cs/encyklopedie/fyzika/svetlo/lidske-oko

[21] Langrová, A.: *Příčiny leukokorie v dětském věku*. Diplomová práce, Masarykova Univerzita v Brně, Fakulta lékařská, 2014.

[22] Mallick, S.: Histogram of Oriented Gradients. Prosinec 2016, [Online; navštíveno 4.5.2019].
URL https://www.learnopencv.com/histogram-of-oriented-gradients/

[23] Mangin, A.: What are the main differences between ReactJS and React-Native? *MEDIUM.com*, Prosinec 2016.

[24] Messenger, M.: What exactly is TensorFlow? *MEDIUM.com*, Říjen 2018.

[25] Mrázek, Z.: *Aplikace pro demonstraci metody histogram of oriented gradients pro detekci objektů*. Diplomová práce, Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2014.

[26] Neustadt, I.; Arlow, J.: *UML 2 a unifikovaný proces vývoje aplikací*. Brno: Computer Press, druhé vydání, 2007, ISBN 978-80-251-1503-9.

[27] Patro, N.: Choose the best-Native App vs Hybrid App. 2018, [Online; navštíveno 11.1.2019].
URL https://codeburst.io/native-app-or-hybrid-app-ca08e460df9

[28] Prabhu: Understanding of Convolutional Neural Network (CNN) – Deep Learning. *MEDIUM.com*, Březen 2018.

[29] Přinosil, J.; Krolikowski, M.: Využití detektoru Viola-Jones pro lokalizaci obličeje a očí v barevných obrazech. *ELEKTROREVUE*, Srpen 2008.

[30] Rek, P.: *Knihovna pro návrh konvolučních neuronových sítí*. Diplomová práce, Vysoké učení technické v Brně, Fakulta informačních technologií, 2018.

[31] Rosebrock, A.: Facial landmarks with dlib, OpenCV, and Python. 2017, [Online; navštíveno 23.11.2018].
URL https://www.pyimagesearch.com/2017/04/03/facial-landmarks-dlib-opencv-python/

[32] Rosebrock, A.: (Faster) Facial landmark detector with dlib. 2018, [Online; navštíveno 2.4.2019].
URL https://www.pyimagesearch.com/2018/04/02/faster-facial-landmark-detector-with-dlib/

[33] Schoffer, R.; Krejčířová, I.; Autrata, R.: Diferenciální diagnostika leukokorie u dětí. 2017, [Online; navštíveno 31.3.2019].
URL https://www.pediatriepropraxi.cz/incpdfs/act-000269-0001_10_001.pdf

[34] Scott, A.: Your camera could save a child's life. [Online; navštíveno 31.3.2019].
URL http://alscotts.blogspot.com/2009/10/your-camera-can-save-childs-life-true.html

[35] Sreeram, J.; Herhut, S.; Kuper, L.: Bringing Parallelism to the Web with River Trail. *RiverTrail*.
URL http://intellabs.github.io/RiverTrail/tutorial/

[36] Tanner, G.: Introduction to Deep Learning with Keras. *TowardsDataScience.com*, Leden 2019.

[37] Unger, R.; Chandler, C.: *A Project Guide to UX Design For User Experience Designers in the Field or in the Making*. Berkeley: New Riders, druhé vydání, 2012, ISBN 0-321-81538-6.

[38] Viola, P.; Jones, M.: Robust Real-Time Face Detection. *International Journal of Computer Vision*, ročník 57(2), 2004: s. 137–154, [Online; navštíveno 2.4.2019].
URL http://www.face-rec.org/algorithms/Boosting-Ensemble/16981346.pdf

[39] Warcholinski, M.: What Is the Difference Between Wireframe, Mockup and Prototype? [Online; navštíveno 25.11.2018].
URL
https://brainhub.eu/blog/difference-between-wireframe-mockup-prototype/

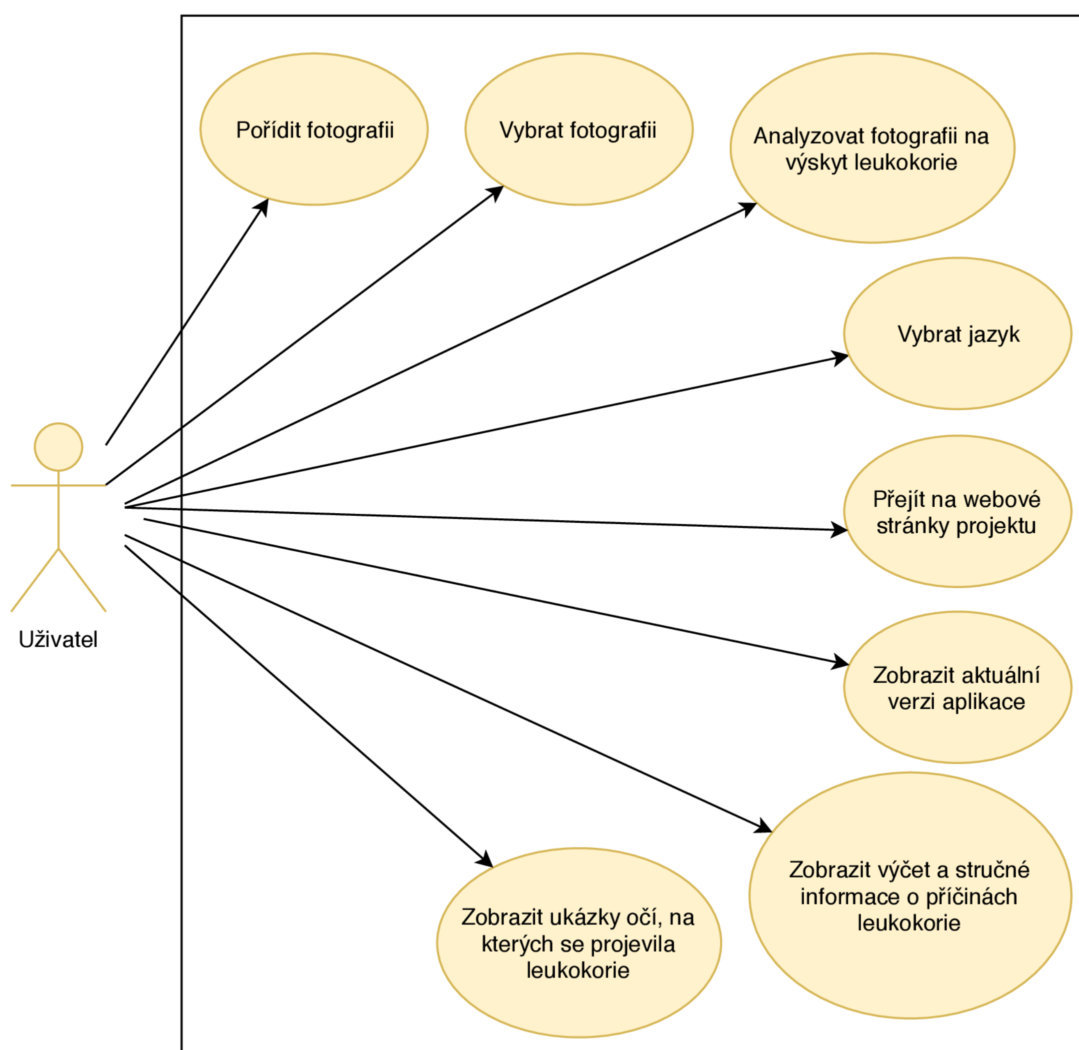
[40] Weng, L.: Object Detection for Dummies Part 1: Gradient Vector, HOG, and SS. 2017, [Online; navštíveno 10.4.2019].
URL https://lilianweng.github.io/lil-log/2017/10/29/object-recognition-for-dummies-part-1.html

[41] Zepeda, I.: Face Detection with dlib. 2018, [Online; navštíveno 2.4.2019].
URL https://topicfly.io/face-detection-with-dlib/

[42] Žajdlíková, B.: Osobní sdělení lékaře (lékaře Fakultní nemocnice Brno – Dětská nemocnice, Černopolní 9, Brno) dne 12. listopadu 2018.

Příloha A

Diagram případů užití



Obrázek A.1: Diagram případů užití zobrazující funkční požadavky na mobilní aplikaci Eye Check.

Příloha B

Plakát

Obrázek B.1: Prezentační plakát vytvořený pro mobilní aplikaci Eye Check.

Příloha C

Obsah přiloženého DVD

Přiložené DVD obsahuje následující adresáře:

- **client** – Zdrojové kódy klientské části mobilní aplikace.
- **server** – Zdrojové kódy serverové části mobilní aplikace.
- **tool** – Zdrojové kódy nástroje pro pořizování vhodné datové sady.
- **cnn** – Zdrojové kódy konvoluční neuronové sítě (natrénovaný model, skripty pro trénování a vyhodnocení modelu, trénovací a testovací datová sada).
- **webApp** – Zdrojové kódy webové aplikace.
- **presentation** – Prezentační materiály (plakát, video).
- **latex** – Zdrojové kódy technické zprávy.
- **thesis** – Technická zpráva ve formátu `PDF`.
- **auth** – Obsahuje soubor `auth.txt`, ve kterém je uveden návod a přihlašovací údaje do administrační části webové aplikace.

Příloha D

Instalační manuál mobilní aplikace Eye Check

V následujících řádcích je popsáno, jakými způsoby je možné spustit mobilní aplikaci Eye Check.

Klientská část

Mobilní aplikaci Eye Check lze spustit na mobilním zařízení několika způsoby:

1. Stažením a následnou instalací aplikace z *App Store* a *Google Play*.
   - *App Store* (*iOS*)
     https://itunes.apple.com/cz/app/eye-check/id1454515853?l=cs&mt=8
   - *Google Play* (*Android*)
     https://play.google.com/store/apps/details?id=com.pavelhrebicek.eyecheck
2. Pomocí souborů `IPA` (*iOS*) nebo `APK` (*Android*).
   - **`IPA`** (*iOS*) – Zkopírováním souboru `client/eyecheck-ios.ipa` do zařízení s operačním systémem *iOS* a následnou instalací.
     (a) Otevřít vývojové prostředí *Xcode*.
     (b) Otevřít menu *Window* → *Devices and Simulators*.
     (c) V zobrazeném okně vybrat záložku *Devices* a následně přetáhnout soubor `client/eyecheck-ios.ipa` do pole *INSTALLED APPS*. Aplikace se po přetažení do mobilního zařízení zároveň nainstaluje.
   - **`APK`** (*Android*) – Zkopírováním souboru `client/eyecheck-android.apk` do zařízení s operačním systémem *Android* a následnou instalací.
3. Sestavením aplikace pomocí přiložených zdrojových kódů.
   (a) K sestavení aplikace je nejprve nutné nastavit a nainstalovat framework *React Native* podle oficiální dokumentace, která je dostupná na https://facebook.github.io/react-native/docs/getting-started.
   (b) Z adresáře `client/EyeCheck` spustit následující příkazy:
      - **`npm install`** – Nainstaluje potřebné knihovny.

- **`react-native run-ios`** – Spustí aplikaci pro mobilní zařízení s operačním systémem *iOS*.
- **`react-native run-android`** – Spustí aplikaci pro mobilní zařízení s operačním systémem *Android*.

Serverová část

Pro spuštění serverové části mobilní aplikace Eye Check pomocí přiložených zdrojových kódů je nutné provést následující kroky:

1. Stáhnout a nainstalovat *Python 3* (Zkoušeno na verzi 3.6.3).
2. Vytvořit virtuální prostředí pomocí příkazu `python3 -m venv server_env` (Linux a macOS).
3. Aktivovat virtuální prostředí příkazem `source server_env/bin/activate`.
4. Přejít do adresáře `server` a nainstalovat potřebné knihovny pomocí příkazu `pip3 install -r requirements.txt`.
5. Přejít do adresáře `server/eyecheck` a spustit server pomocí příkazu `python manage.py runserver`.

Příloha E

Instalační manuál webové aplikace

V následujících řádcích je popsáno, jakými způsoby je možné spustit webovou aplikaci, která byla implementována jako rozšíření této práce.

1. Ve webovém prohlížeči přejít na adresu `https://eyecheck.cz`.
2. Spuštění aplikace pomocí přiložených zdrojových kódů.
   (a) Stáhnout a spustit webový server *Apache* a *MySQL* databázi, např. s využitím prostředí `XAMPP`[1].
   (b) Naimportovat DB z adresáře `webApp/db/eyecheck.sql`.
   (c) V souboru `webApp/app/config/config.local.neon` nastavit připojení a přihlašovací údaje k DB. Ukázka nastavení připojení a přihlašovacích údajů k DB lze vidět na výpisu E.1.

```
database:
  dsn: 'mysql:host=127.0.0.1;dbname=eye_check'
  user: root
  password: admin
```

Výpis E.1: Ukázka nastavení připojení a zadání přihlašovacích údajů k DB v souboru `config.local.neon`.

   (d) Nainstalovat potřebné knihovny pomocí nástroje *Composer*[2]. Z adresáře `webApp` spustit příkaz `composer install`.
   (e) Návod a přihlašovací údaje do administrační části webové aplikace jsou uvedeny v souboru `auth/auth.txt`.

[1] `https://www.apachefriends.org/index.html`
[2] `https://getcomposer.org`

Příloha F

Instalační manuál nástroje pro pořizování vhodné datové sady

V následujících řádcích je popsáno, jakým způsobem je možno spustit nástroj `eyechecktool`, který slouží k pořizování vhodné datové sady a výrazně urychluje proces trénování neuronové sítě. Konvence povinných a volitelných parametrů nástroje a ukázky spuštění jsou ukázány v kapitole 5.3.4.

1. Stáhnout a nainstalovat *Python 3* (Zkoušeno na verzi 3.6.3).
2. Vytvořit virtuální prostředí pomocí příkazu `python3 -m venv tool_env` (Linux a macOS).
3. Aktivovat virtuální prostředí příkazem `source tool_env/bin/activate`.
4. Přejít do adresáře `tool` a nainstalovat potřebné knihovny pomocí příkazu `pip3 install -r requirements.txt`.

Příloha G

Instalační manuál, ukázky trénování a ověření úspěšnosti modelu konvoluční neuronové sítě

Instalace konvoluční neuronové sítě

1. Stáhnout a nainstalovat *Python 3* (Zkoušeno na verzi 3.6.3).
2. Vytvořit virtuální prostředí pomocí příkazu `python3 -m venv cnn_env` (Linux a macOS).
3. Aktivovat virtuální prostředí příkazem `source cnn_env/bin/activate`.
4. Nainstalovat potřebné knihovny pomocí příkazu `pip3 install -r requirements.txt`.

Vytvoření a trénování modelu

Pro vytvoření a trénování modelu byl vytvořen skript `cnn/create_train_cnn.py`. Ukázka spuštění tohoto skriptu je zobrazena na výpisu G.1. Součástí adresáře `cnn` je i natrénovaný model `eyecheck_model.h5`, který je použit na serverové části mobilní aplikace, a jehož úspěšnost byla vyhodnocena v kapitole 5.2.6.

```
$ python create_train_cnn.py
```

Výpis G.1: Ukázka spuštění skriptu `create_train_cnn.py`, který vytvoří a natrénuje model *konvoluční neuronové sítě*.

Ověření úspěšnosti natrénovaného modelu

K ověření úspěšnosti natrénovaného modelu byl vytvořen skript `cnn/test_model.py`. Skript má jeden povinný parametr `--image-folder-path`, který identifikuje cestu k fotografiím (výřezům pravého, resp. levého oka). Ukázka spuštění skriptu je zobrazena na výpisu G.2.

```
python test_model.py --image-folder-path dataset/test_set/healthy/
```

Výpis G.2: Ukázka spuštění skriptu `test_model.py`, který ověří úspěšnost natrénovaného modelu na testovací datové sadě zdravých očí jedinců.