Mendelova univerzita v Brně
Provozně ekonomická fakulta

Řešení pro distribuovanou detekci objektů v obraze

Bakalářská práce

Vedoucí práce:
Ing. David Procházka, Ph.D.

Tomáš Iša

Brno, 2017

Chtěl bych poděkovat své rodině, která mě podporovala po celou dobu bakalářského studia a také vedoucímu své bakalářské práce panu Ing. Davidu Procházkovi, Ph.D., za poskytnutí pomoci a cenných vědomostí v průběhu vývoje aplikace, a především za jeho trpělivost a pečlivost.

Čestné prohlášení

Prohlašuji, že jsem tuto práci: **Řešení pro distribuovanou detekci objektů v obraze**
vypracoval samostatně a veškeré použité prameny a informace jsou uvedeny v seznamu použité literatury. Souhlasím, aby moje práce byla zveřejněna v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách ve znění pozdějších předpisů, a v souladu s platnou *Směrnicí o zveřejňování vysokoškolských závěrečných prací.*
Jsem si vědom, že se na moji práci vztahuje zákon č. 121/2000 Sb., autorský zákon, a že Mendelova univerzita v Brně má právo na uzavření licenční smlouvy a užití této práce jako školního díla podle § 60 odst. 1 Autorského zákona.
Dále se zavazuji, že před sepsáním licenční smlouvy o využití díla jinou osobou (subjektem) si vyžádám písemné stanovisko univerzity o tom, že předmětná licenční smlouva není v rozporu s oprávněnými zájmy univerzity, a zavazuji se uhradit případný příspěvek na úhradu nákladů spojených se vznikem díla, a to až do jejich skutečné výše.

místo a datum prohlášení .....

Abstract

IŠA, T. Solution for distributed object detection. Brno, 2017. Bachelor thesis. Mendel University in Brno.

This thesis is focused on the development of mobile application and server service for distributed image processing. The mobile application gathers image data that are sent to the service which provides information about objects detected within the scene. The mobile application is developed in C++ using Qt framework. The server service is also implemented in C++ and uses the OpenCV library for object detection.

Keywords: distributed image processing, object detection, Qt, OpenCV

Abstrakt

IŠA, T. Řešení pro distribuovanou detekci objektů v obraze. Brno, 2017. Bakalářská práce. Mendelova univerzita v Brně.

Závěrečná práce je zaměřena na vytvoření mobilní aplikace a serverové služby pro distribuované zpracování obrazu. Mobilní aplikace zasílá obrazová data serverové službě, která provádí detekci objektů v předané scéně. Mobilní aplikace je implementována v C++ s pomocí knihovny Qt. Serverová služba je také implementována v C++ a využívá knihovnu OpenCV pro zpracování obrazu.

Klíčová slova: distribuované zpracování obrazu, detekce objektů, Qt, OpenCV

Obsah

1 Úvod

Lidé jsou v dnešní době zaplavováni velkým množstvím informací a z tohoto důvodu se hledají metody, jak filtrovat informace, tak aby uživatel dostal vždy správnou informaci, ve správný čas a na správném místě. V každodenním životě je tohoto principu využíváno například u lokačních služeb. Jednoduše se dají vyhledat obchody, nebo restaurace v okolí. Na rozdíl od identifikace místa, problém nastává v situaci, kdy uživatel potřebuje informace o konkrétním objektu. V ten okamžik musí zařízení chápat, na co se se uživatel dívá. Jednou z prvních aplikací tohoto typu jsou *Google Goggles*.

Objektů, na které se uživatel dívá, může být spousta, z tohoto důvodu se hodí cloudové zpracování. Díky němu může být na jednom místě uložena celá řada snadno aktualizovatelných vzorů, které mohou být využity ke zpracování nejrůznějšími metodami.

Tato bakalářská práce se zabývá distribuovaným zpracováním obrazu. V rámci této problematiky bude vyvinuta aplikace, skládající se ze dvou částí, klienta a serveru.

Klientská část aplikace bude mít za úkol pořídit fotografii a zaslat ji pomocí zvolené metody síťové komunikace na stranu serveru. Jakmile klientská strana získá odpověď od serveru, zobrazí uživateli detekované objekty. Taková klientská aplikace může mít po zdokonalení řadu využití, například zobrazování informací o nalezeném objektu. Takovým objektem může být v praxi například výrobek v obchodním domě, umělecké dílo v muzeu, nebo neznámá rostlina, o které se bude chtít uživatel při svém cestování něco dozvědět.

Serverová část aplikace bude přijímat požadavky od klienta a zasílat zpět informace o detekovaných objektech. Detekce bude probíhat pomocí vybrané metody, za pomoci předem vybraného vzoru. Server může v praxi implementovat více metod zpracování obrazu, aby bylo dosaženo lepších výsledků. Zajímavým zdokonalením může být například ukládání nalezených objektů, pro vytvoření zajímavých statistik, nebo vytvoření databáze hledaných vzorů. Taková databáze by mohla pomoci v detekování více objektů najednou. Server by měl být výkonný a měl by dokázat obsluhovat několik připojených uživatelů v jeden moment.

2 Cíl práce

Navržení a implementace desktopové serverové aplikace a klientské mobilní aplikace ve zvoleném programovacím jazyce, která umožní přijmutí obrazu, vyhledání relevantního objektu, který se v obraze nachází a vrácení popisu objektu klientovi. Pro vyhledání objektu bude použit vybraný algoritmus podporovaný vhodnou knihovnou pro zpracování obrazu.

2.1 Dílčí cíle

Vývoj výsledné aplikace se skládá z následujících dílčích cílů.

1. Nastudování relevantních vývojových nástrojů a knihoven. Provedení rešerše zaměřené na detekci objektů na mobilních zařízeních.
2. Navržení metody komunikace mezi klientem a serverem, která umožní přenos pořízených fotografií a výsledných dat.
3. Navržení a implementace serverové a mobilní aplikace, která umožní zpracování pořízeného obrazu pomocí vybraného algoritmu.
4. Vytvoření a otestování mobilní aplikace na operačním systému iOS nebo Android.
5. Vyhodnocení výsledného projektu. Vyzdvihnutí všech omezení daných hardwarem, softwarem a implementací aplikace.

3 Zpracování obrazu

V dnešní době se stále více využívá zpracování různých obrazů pomocí počítačových nástrojů. Existuje spousta oblastí, kde má lidské vnímání navrch oproti tomu strojovému, ale naopak také existují ty, kde je tomu naopak. Právě v těchto případech se vývojáři systémů, které detekují informace v obrazech snaží dosáhnout takových výsledků, díky kterým by mohlo být lidské vnímání posunuto kupředu a předčilo tak naše biologické předpoklady. Existuje spousta knihoven, která obsahuje různorodé algoritmy, díky kterým může člověk dosáhnout zajímavých výsledků. V následujících kapitolách budou diskutovány a představeny vybrané knihovny a také způsob, kterým se obrazy před samotným zpracováním přenášejí.

Moderní trendy v oblasti detekce obrazu

V dnešní době jsou základními stavebními kameny informatiky následující oblasti: zpracování obrazu, počítačové vnímání a rozpoznávání vzorků (Deligiannidis, 2014).

Algoritmy zpracovávající obraz obklopují každého, kdo je součástí některé z internetových sociálních sítí. Veškeré uživatelské příspěvky jsou analyzovány velkou škálou algoritmů. Sociální síť si tímto způsobem o uživateli získává informace a následně pak aplikuje cílenou reklamu. Například uživatel, který se rád prezentuje různými fotkami z adrenalinových horských výprav zpozoruje zvýšený výskyt reklam, které lákají na nejrůznější zájezdy a slevy na potřebné vybavení. Další cílovou skupinou mohou být lidé, kteří rádi prezentují fotkami svoje kulinářské umění. Vývojáři pracují na algoritmech, které umožní z jedné fotografie získat různorodé informace o jídle. Uživatel se pak díky těmto algoritmům může dozvědět, co jeho kamarád vlastně jedl, jaké to dané jídlo mělo nutriční hodnoty a kolik ho bylo (Slyce.it, 2015).

Tyto přístupy nejsou vázány pouze k reklamě v prostředí internetu. Využívat je mohou i majitelé kamenných obchodů. Díky pořízeným fotografiím mohou pomocí detekce analyzovat, kde a jak dlouho se určitý zákazník nacházel. Z těchto informací mohou později hromadně vytvořit tzv. heat mapy, které tyto informace přehledně zobrazí a nejvíce frekventovaná místa mohou být použita k prezentaci nových produktů.

Samozřejmostí je také využití těchto technologií ve vědě a výzkumu. Národní ústav pro letectví a kosmonautiku (NASA) využívá nový software a algoritmy pro analýzu historických fotek, pořízených Hubblovým teleskopem, ve snaze objevit na noční obloze nové hvězdy a další vesmírné útvary (Slyce.it, 2015).

V neposlední řadě může rozpoznávaní obrazu sloužit také v lékařství. V tomto odvětví jsou analyzovány fotografie pořízené rentgenem, ultrazvukem apod. Výsledkem analýzy je zpráva obsahující informace o abnormálních útvarech nacházejících se v lidském těle. Díky ní pak mohou lékaři lépe diagnostikovat pacienta a lépe připravit léčebný plán. V některých případech bylo zjištěno, že tato technologie našla zárodky rakoviny prsu dříve a přesněji, než zkušený lékař (Slyce.it, 2015).

Detekci obrazu a zjišťování vzorků je možno použít také pro zkrácení dlouhých chvílí. Na tomto principu funguje například zařízení Kinect pro Xbox, které v reálnem čase promítne osobu do virtuálního prostředí a umožní uživateli být součástí hry. Zařízení hledá obrys osoby vzhledem k jejímu okolí a díky tomu přesně vypočítá, kde se právě osoba nachází. Tomuto algoritmu se říká Skeletal Tracking algorithm (Murino, 2013).

3.1 Knihovny pro zpracovávání obrazu

V informatice se knihovnou nazývá souhrn různých zdrojů, které používají počítačové programy. Tento souhrn může obsahovat konfigurační data, dokumentaci, pomocná data, části předem napsaného kódu a různé třídy a funkce (Rouse, 2005). Knihovny jsou většinou poskytovány ve formě zdrojových souborů a jsou určené pro práci s určitým programovacím jazykem. Tyto zdrojové soubory si musí vývojář nechat přeložit (zkompilovat) nativním kompilátorem, kterým jeho počítač disponuje. Poté je knihovna připravena k používání a připojením základních definičních souborů do programu, může být bezpečně používána. Takových knihoven je samozřejmě celá řada a netýkají se pouze zpracování obrazu, ale také mnoha dalších odvětví. V následujícím textu bude uvedeno několik knihoven pro zpracovávání obrazu.

OpenCV

OpenCV je zkratkou pro *Open Source Computer Vision Library*. Jak je již z názvu čitelné, jedná se o neplacenou a volně dostupnou knihovnu. *OpenCV* obsahuje stovky počítačových algoritmů, které zpracovávají multimediální data (Laganiére, 2011). Knihovna využívá programovací jazyk C++ a je jedna z celosvětově nejrozšířenějších knihoven. Poslední dostupná verze byla vydána v červnu roku 2015.

VLFeat

VLFeat je další volně dostupnou knihovnou. Využívá programovací jazyk C a je nadstavbou pro *MATLAB*. *MATLAB* (*matrix laboratory*) je interaktivní programové prostředí a skriptovací programovací jazyk čtvrté generace. Knihovna obsahuje řadu oblíbených algoritmů. Poslední dostupná verze byla vydána v lednu roku 2015 (Vedaldi a Fulkerson, 2008).

BoofCV

BoofCV je taktéž volně dostupná knihovna. Využívá programovací jazyk Java a slouží pro zpracování počítačového vnímání a robotiky v reálném čase. Knihovna má širokou škálu funkcí od nízko úrovňového zpracování obrazu, přes kalibraci kamery, až po rozpoznávání obrazu (Abeles, 2012). Poslední dostupná verze byla vydána v listopadu roku 2015.

scikit-image (SciKit)

SciKit je dalším zástupcem volně šiřitelné knihovny pro zpracování obrazu. Knihovna využívá programovací jazyk Python a obsahuje taktéž širokou škálu algoritmů. Vývojáři této knihovny se pyšní vysoce kvalitními ukázkami zdrojových kódů, které zpracovává aktivní komunita dobrovolníků (scikit-image, 2011).

Výsledek srovnání

Jak bylo zjištěno z dokumentací jednotlivých knihoven, velká část algoritmů je obsažena v každé z nich. Je to z toho důvodu, že se vývoj tohoto typu aplikací tolik rozšířil, že knihovny, které by neobsahovaly některé důležité algoritmy, by nikdo nepoužíval a neměly by si jak vybudovat svoji dobrou pověst. Některé rozdíly bylo možné zpozorovat v kvalitě zpracování dokumentace a potom samozřejmě ve formě implementace knihoven. Právě díky rozdílné implementaci si přijdou na své vývojáři programující rozdílnými jazyky.

Název	Jazyk	Dokumentace	Komunita
OpenCV	C++	ANO	Rozsáhlá
VLFeat	Matlab	ANO	Malá
BoofCV	Java	ANO	Střední
SciKit	Python	ANO	Střední

3.2 Síťová komunikace

Jelikož jsou v moderní době čím dál více oblíbená různá přenosná zařízení, která většinou nemají tak výkonný hardware jako klasické stolní počítače a už zdaleka ne takovou výdrž baterie, je potřeba, aby část operací, kterou by musela zařízení provést, byla přesunuta na výkonný server. K tomuto procesu výborně slouží síťová architektura klient–server. Strana klienta, která je většinou představována nějakou mobilní aplikací, provádí pouze jednoduché operace, jako jsou pořízení a odeslání fotografie, nebo přijímání výsledků. Strana serveru si na sebe bere zodpovědnost v podobě zpracování přijaté zprávy ze strany klienta. Server od klienta přijme ve zprávě nový obraz a provede na něm některé z vybraných algoritmů. Výsledky, které jsou poskytnuty algoritmy, jsou poté odeslány jako odpověď na stranu klienta, kde je uživatel získá v přehledné podobě prostřednictvím uživatelského grafického rozhraní.

Síťová komunikace v prostředí Qt

V rámci této práce bude síťová komunikace zpracována ve vývojovém prostředí *Qt Creator*. Samotné vývojové prostředí obsahuje spoustu již předem připravených tříd, které řeší jednotlivé části komunikace. Správným použitím těchto tříd, si vývojář velmi usnadní práci a dosáhne lepších výsledků než při vlastní implementaci. Přenos dat bude fungovat na bázi protokolu TCP (*Transmission Control Protocol*). Díky kontrolním součtům nám protokol mimo jiné zajistí správnost přenesených dat. Obě

strany si posílají zprávy o přijatých datech a dá se tak jednoduše zjistit, kdy skončilo odesílání jednoho souboru a začalo odesílání dalšího, to se hodí obvzlášť při odesílání více výsledků, které se týkají jednoho obrazu. Qt podporuje i multivláknovost, která zajišťuje pohodlné připojení více uživatelů zároveň.

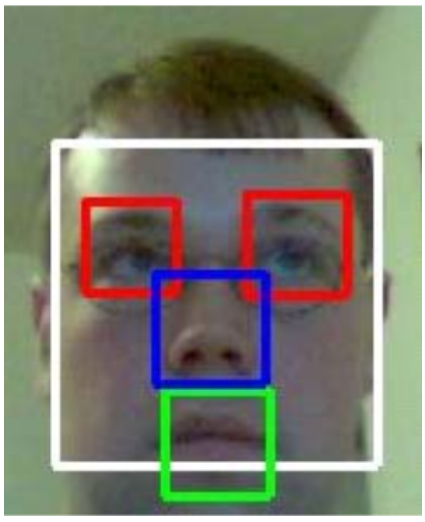
3.3 Existující řešení zaměřené na zpracování obrazu

Existujících řešení na zadané téma je mezi akademickými články opravdu velké množství. Nejčastějším tématem těchto prací je především rozpoznávání lidského obličeje, nebo lidské postavy. Problematika různých řešení zpracování obrazu bude představena v následujících odstavcích.

Detekce obličeje pomocí Haar klasifikátorů

První aplikaci detekce obrazu je možné vidět v práci *Facial feature detection using Haar classifiers* (Wilson a Fernandez, 2006), jejíž název je volně přeložen jako Detekce obličeje pomocí Haar klasifikátorů.

Všeobecně známé algoritmy detekce obličeje jsou založeny na rozpoznávání obrysů, různých odstínů pleti, nebo aplikování filtrů pomocí neuronových sítí. Všechny zmíněné algoritmy ale mají nevýhody, především se jedná o jejich výpočetní náročnost. Výpočetní náročnost je tak vysoká, protože se analyzují všechny pixely v obrázku. Haar metoda, která je touto prací zkoumána ale analyzuje pixely odlišným způsobem. Při detekci jsou porovnávány změny kontrastu mezi přilehlými skupinami pixelů. Více těchto skupin se nazývá *Haar vlastnost*. K plnému rozpoznání obličeje je zapotřebí vypočítat pouze pár takovýchto Haar vlastností. K implementaci je stejně jako v této práci využita knihovna *OpenCV*. Obličej detekovaný pomocí této metody může vypadat následovně. (Wilson a Fernandez, 2006)



Obrázek 1: Obličej detekovaný Haar metodou (Wilson a Fernandez, 2006)

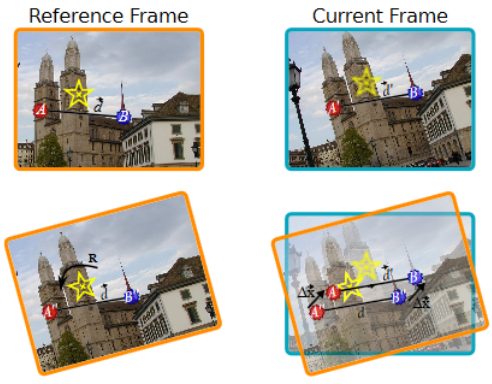
Pro optimalizování rychlosti detekce je využito následujícího postupu. Nejdříve je detekovaný celý obličej, k tomuto je potřeba analyzování celého obrazu, oblast obličeje je zaznačena bílým obdélníkem. Poté se již pracuje pouze s vybranou oblastí, ve které jsou detekovány jednotlivé části obličeje, jako je nos, oči, nebo ústa. (Wilson a Fernandez, 2006)

Existující klient-server řešení

Další prací, která se zabývá danou problematikou je práce s názvem *Server-side object recognition and client-side object tracking for mobile augmented reality* (Gammeter, Gassmann, Bossard, Quack a Van Gool, 2010), který může být volně přeložen jako Rozpoznávání objektů na straně serveru a sledování objektů na straně klienta v rámci mobilního zařízení.

Tato aplikace klade důraz na efektivní využití klient-server architektury, implementaci serverové databáze, spolehlivé a rychlé rozpoznání objektu a finální implementaci pro mobilní zařízení s operačním systémem *Android*. Server obsahuje již zmíněnou databázi, ve které je uloženo mnoho vzorů, na základě kterých jsou vraceny informace o objektech. (Gammeter, Gassmann, Bossard, Quack a Van Gool, 2010)

Klientská část této aplikace je velmi propracovaná. Objekty jsou po získání informací nadále sledovány. Objekt je při snímání prostoru kamerou označen značkami, které musí být pro každých pár snímků aktualizovány, aby ukazovaly na správný objekt, ve správném místě. Tohoto sledování je dosaženo pomocí různých moderních senzorů, které má většina chytrých telefonů zabudované. Tyto senzory mohou být například v podobě gyroskopů a kompasů. Čtení senzorů je doplněno o tzv. *Visual Feature Tracking*. Ná následujícím obrázku je vyobrazeno, jak detekce objektu funguje při změně perspektivy kamery. (Gammeter, Gassmann, Bossard, Quack a Van Gool, 2010)



Obrázek 2: Sledování objektu při změně perspektivy (Gammeter, Gassmann, Bossard, Quack a Van Gool, 2010)

Jedna z hlavních výhod tohoto řešení byla ta, že při implementaci nebylo nutné využívat data z GPS (Gammeter, Gassmann, Bossard, Quack a Van Gool, 2010).

Detekce objektu a jeho orientace

Poslední příklad je publikován v knize zabývající se využitím knihovny *OpenCV*.

Jedna z kapitol je věnována problematice, která je velmi podobná této práci. Cílem kapitoly je vytvoření aplikace, která dokáže vyhledat objekt ve videu a následně zjistit jeho orientaci v prostoru a vykreslit nový 3D objekt (Lelis Baggio, 2012). Vyhledání shody probíhá na principu srovnání klíčových bodů. Nejdříve je analyzován obraz s objektem který je hledán a poté jsou analyzovány jednotlivé snímky videa. Jakmile jsou známy klíčové body, může dojít k jejich srovnání a k pokusu o zobrazení shody. O srovnání klíčových bodů se stará *OpenCV* třída `cv::FlannBasedMatcher`. K dispozici je i uživatelské rozhraní, které dokáže vyobrazit shody mezi vybranými obrazy. Takové zobrazení vypadá následovně. (Lelis Baggio, 2012)

Obrázek 3: Shody dvou obrazů (Lelis Baggio, 2012)

Jestliže je fotoaparát, kterým bylo video pořízeno správně nakalibrován a vývojář má přístup k údajům o perspektivě scény, dokáže pomocí těchto informací a již výše zmíněných klíčových bodů zakreslit do obrazu trojrozměrný objekt, který se stává součástí scény (Lelis Baggio, 2012). K tomuto se dá dopracovat pomocí složitějších kalkulací, které jsou již v *OpenCV* naimplementovány. Takto upravená scéna může vypadat následovně.

Obrázek 4: Vykreslení 3D objektu do scény (Lelis Baggio, 2012)

Jak bylo možno ze všech předchozích příkladu zjistit, má zpracování obrazu nekonečné množství použití a je tedy odvětvím, o které se začíná zajímat stále více lidí.

4 Metodika práce

4.1 Architektura klient–server

Při implementaci popsaného problému bude využito architektury klient-server. V dnešní době je to jedna z nejčastěji používaných architektur. Rozděluje funkcionalitu aplikace na dvě různé části, které mají své dané specifické vlastnosti a plní určité úkoly.

Klient

Klient je program, který zasílá serverové části aplikace požadavky, na základě kterých, potom provádí uživatelem vybrané operace (Held, 2000). Klientem mohou být různá zařízení. V případě této práce bude klienta představovat mobilní zařízení od společnosti *Apple*, s operačním systémem *iOS*. Toto zařízení pomocí vestavěného fotoaparátu pořídí obraz vybraného objektu. Fotografie bude následně odeslána na server, kde dojde ke zpracování. Aby bylo ušetřeno místo na mobilním zařízení, nebude docházet k ukládání pořízených fotografií, ale pouze informací o nich.

Server

Server je zodpovědný za přijímání požadavků (tzv. request) od klienta, na základě kterých provede procedury, které umožní klientovi uskutečnit vybrané úkoly.

Serverová část aplikace běží na velmi výkonných zařízeních, které se od běžných klientských zařízení liší především hardwarovými specifikacemi. Protože server nepotřebuje zobrazovat výsledky a vykreslovat různá uživatelská rozhraní, tak ve většinu případů nepotřebuje grafické karty a jiné součástky zodpovědné za vykreslení obrazu. Oproti klientskému zařízení obsahuje především více procesorových jader, operační paměti a pevných disků, určených pro ukládání zpracovaných dat. Tyto disky jsou zálohovány, aby v případě poruchy nebyl porušen tok a integrita dat.

V případě této práce bude server představovat stolní počítač, který bude naslouchat na vybrané IP adrese a portu. Díky podpoře více vláken bude moci odpovídat na více požadavků zároveň. Komunikace bude probíhat pomocí protokolu TCP a data budou uchovávána ve formátu JSON.

4.2 Integrované vývojové prostředí Qt Creator

Vývojové prostředí je aplikace, která programátorovi ulehčuje práci a poskytuje širokou škálu nástrojů, které uživatele usměrňují k tomu, aby psal přehledný kód a vyvaroval se různým chybám v logice programu. Umožňuje do hloubky procházet zdrojový kód, největším přínosem je především tzv. debugging, který v reálném čase uživateli prezentuje, jaká data se v danou chvíli nachází v proměnných využitých v aplikaci. Díky tomuto nástroji se značně zkrátí čas věnovaný zjišťování různých nedostatků a chyb v kódu. Když se i přes toto vše programátorovi podaří zanést

do kódu nějakou chybu a selže překlad programu, tak vývojové prostředí napoví uživateli kde chybu hledat a jak ji vyřešit.

Při implementaci bude využito vývojového prostředí *Qt Creator*. Toto vývojové prostředí bylo vybráno především kvůli jeho podpoře multiplatformního vývoje. Výhoda multiplatformnosti spočívá v tom, že programátor není vázán na určitý typ operačního systému, ať už na počítači, kde běží vývojové prostředí, nebo na cílovém zařízení, pro které je program vyvíjen. Dále není potřeba vyvíjet několik druhů uživatelských rozhraní, protože *Qt Creator* převezme nativní grafické prvky, jako jsou například tlačítka, nebo checkboxy, z cílového operačního systému a vykreslí je tak, jak je daný uživatel zvyklý. Pro tyto účely obsahuje IDE funkcionality *Qt Designer* a *Qt Quick Designer*, které umožňují navrhovat a testovat skladbu uživatelského rozhraní a dále zkrášlovat pomocí různých grafických efektů a animací (Qt Dokumentace, 2017).

Společnost Qt nabízí dvě verze vývojového prostředí a to open-souce a komerční. Při implementaci této práce bude využito open-source verze. Komerční (placená verze) musí být využita v případě, že budeme chtít mít na vyvinutém programu zisk (Qt Dokumentace, 2017).

Qt pro mobilní operační systém iOS

Vyvíjená mobilní aplikace poběží na operačním systému *iOS* od firmy *Apple*. Pro vývoj a nasazení aplikace na mobilní zařízení podporující *iOS*, je potřeba, aby vývojové prostředí *Qt Creator* dokázalo komunikovat s *Xcode* (Qt Dokumentace, 2017). Tato funkcionalita je přístupná pouze vývojářům, kteří si stáhnou přídavnou *iOS* knihovnu k základní verzi *Qt Creatoru*. Programátor musí vlastnit osobní počítač s operačním systémem *macOS* a vývojářský certifikát, který je v rámci akademických prací zdarma na určité časové období.

4.3 Programovací a značkovací jazyky

Vybrané vývojové prostředí podporuje vývoj v jazyce *C++*, z tohoto důvodu byl tento jazyk vybrán za hlavní a téměř všechny části práce v něm budou napsány. Jiného jazyka bude použito pouze při definici uživatelského rozhraní a při přístupu k nativním funkcionalitám *iOS*. Konkrétně se bude jednat o značkovací jazyk *QML* a programovací jazyk *Objective-C*.

Programovací jazyk Objective-C

Objektový jazyk *Objective-C* je derivací programovacího jazyka C. Používá se především pro vývoj mobilních aplikací pro zařízení od firmy *Apple* (Objective-C Dokumentace, 2017). Díky jeho syntaxi je velmi čitelným jazykem a proto není zapotřebí tak rozsáhlé dokumentace, jako u jiných jazyků. V nedávné době byl tento jazyk nahrazen jeho modernější verzí, která byla pojmenována *Swift*, ale pro účely této práce

postačí základy z *Objective-C*. Pomocí tohoto jazyka připojíme aplikaci k fotoaparátu obsaženém v mobilním zařízení a získáme pomocí něj důležitá obrazová data, která budou po odeslání na server zpracována. V budoucnu firma *Qt* plánuje plnou podporu *iOS* fotoaparátu, která umožní, aby se vývojář vyhnul psaní *Objective-C* kódu.

Značkovací jazyk QML

Jazyk *QML* umožňuje popis uživatelských rozhraní a jejich vzájemných propojení (Qt Dokumentace, 2017). Uživatel si může vytvářet vlastní komponenty, které mohou být znovupoužitelné v nových aplikacích.

QML je velmi čitelný jazyk, který se vyznačuje svojí syntaxí podobnou JSONu a podporuje také spouštění částí Javscriptového kódu (Qt Dokumentace, 2017).

Uživatelské rozhraní může programátor psát ručně a definovat všechny elementy pomocí kódu, nebo si práci může ulehčit a využít implementovaného nástroje *Qt Designer*, který zajišťuje, aby rozhraní bylo symetrické a vypadalo na první pohled čistě a použitelně. (Qt Dokumentace, 2017)

4.4 Knihovna OpenCV

OpenCV bylo vybráno z následujících důvodů. Je nejvíce rozšířenou knihovnou pro zpracování obrazu a díky tomu je jeho učící křivka strmá, protože začátečník má k dispozici mnoho kvalitních návodů k použití. Programovací jazyk C++ zvolený pro vytvoření této práce je knihovnou *OpenCV* nejvíce podporovaný a v neposlední řadě *OpenCV* disponuje propracovanou dokumentací.

Knihovna *OpenCV* obsahuje algoritmy, které slouží ke zpracování obrazu, proto je její použití v této práci velmi důležité. K práci s touto knihovnou lze využít několika programovacích jazyků, ale v tomto případě se bude jednat o C++. Knihovna jde přeložit na téměř všech operačních systémech, v případě této práce konkrétně *macOS* a *iOS*. Je dostupná pod BSD licencí, což znamená, že je volně použitelná ke komerčním i akademickým potřebám. (OpenCV team, 2017)

Knihovna má modulární strukturu, což znamená, že jednotlivé moduly se věnují různým problematikám zpracování obrazu (OpenCV team, 2017). Jádro knihovny obsahuje základní datové struktury a metody používané jinými moduly. Mezi nejpoužívanější patří moduly pro zpracování fotografií, videa a pro generování výstupů v podobě jednoduchých grafických uživatelských rozhraní.

4.5 CMake

CMake je nástroj, kterým budou přeloženy používané knihovny. Většina knihoven je distribuována ve formě zdrojového kódu a pro jejich použití je potřeba je nejdříve přeložit do spustitelné podoby. *CMake* je nástroj, který nám umožní překládat

aplikace nezávisle na platformě, lze modifikovat jednoduchými konfiguracemi a jeho použití je založeno na freewarové licenci (CMake team, 2017).

Výhoda tohoto postupu je taková, že operační systém může použít svůj nativní kompilátor a vyvarovat se pak následným chybám při používání knihovny, která byla přeložena na jiném operačním systému, nebo jeho jiné verzi. Ručním přeložením knihovny můžeme také snížit nároky na paměť počítače, protože si necháme přeložit pouze moduly, které opravdu k práci potřebujeme. V případě, že bychom při implementaci aplikaci zjistili, že potřebujeme přidat další modul, není problém nechat knihovnu přeložit znovu a přiložit další moduly.

4.6 Textový formát JSON

JSON je zkratka pocházející z anglického *JavaScript Object Notation*. Jedná se o textový formát, který slouží ke strukturování a ukládání nejrůznějších typů dat (W3C, 2017). V dnešní době je čím dál více rozšířený a používá se především u webových aplikací. Název tohoto formátu nám napovídá, že je odvozen z JavaScriptu a to přesněji z jeho objektů. Objekt definujeme pomocí složených závorek, pole pomocí hranatých závorek a hodnoty fungují jako klíč a hodnota. Klíč i hodnota jsou mezi uvozovkami a jsou odděleny dvojtečkou. Jednotlivé páry hodnot oddělujeme čárkami.

V této aplikaci bude formát použit k výměně dat mezi serverem a klientem. Budou zde uchovávána data o celém objektu, údajích o jeho předzpracování a následně budou přidány souřadnice vyhledaného objektu, společně s dalšími užitečnými hodnotami.

JSON byl vybrán z důvodu jeho univerzality a přenositelnosti. Stejnou datovou strukturu může jiný vývojář využít k sestavení odlišeného druhu aplikace a poté se díky totožné struktuře dat mohou aplikace navzájem doplňovat.

5 Implementace aplikace

V této části práce bude popsáno, jakým způsobem byly implementovány jednotlivé části aplikace a jaké nástroje k tomu byly použity. Kapitola bude rozdělena do dvou individuálních celků, a to do implementace serverové části a následně implementace klientské části.

5.1 Implementace serverové části aplikace

Qt Creator nabízí při vytvoření projektu zvolení možnosti, jaký druh aplikace chceme vyvíjet. Máme na výběr z několika možností (Qt Dokumentace, 2017).

- Konzolová aplikace
- Aplikace s uživatelským rozhraním
- Aplikace s widgety
- Mobilní aplikace
- Aplikace pro práci s trojrozměrnou scénou

Pro tvorbu serverové části aplikace byla zvolena hned první možnost z předchozího výběru, a to konzolová aplikace. Konzolová aplikace byla zvolena hned z několika důvodů, které ovlivní především její výkon.

Aplikace funguje jako HTTP server, k jehož implementaci bude využito knihovny *QtWebApp*. Aplikace potřebuje pouze naslouchat a obsluhovat požadavky ze strany klienta, k tomuto procesu není nutné, aby obsahovala uživatelské rozhraní, protože všechna nastavení jsou uložena v jednoduchém konfiguračním souboru, který je součástí knihovny *QtWebApp* (Frings, 2017). Tento config vypadá následovně:

```
[listener]
host=195.113.218.159
port=8090
minThreads=4
maxThreads=100
cleanupInterval=60000
readTimeout=60000
maxRequestSize=10485760
maxMultiPartSize=10000000

[sessions]
expirationTime=3600001
cookieName=sessionid
cookiePath=/
cookieComment=Identifies the user
cookieDomain=
```

Údaje obsaženy v konfiguračním souboru jsou použity v následujících případech:

- **Host a Port** – tyto údaje představují IP adresu a port serveru, na kterém serverová část aplikace poběží. Podmínkou obsluhy požadavků od klienta je připojení obou částí aplikace do stejné sítě. (Frings, 2017)
- **minThreads** – nejnižší počet vláken, který server obsluhuje po celou dobu jeho běhu. Počet běžících vláken se nikdy nesmí dostat pod toto číslo. (Frings, 2017)
- **maxThreads** – maximální počet vláken, které server obslouží. Počet běžících vláken nikdy nesmí přesáhnout toto číslo. (Frings, 2017)
- **cleanupInterval** – časový interval v milisekundách, po jehož uběhnutí se zruší vždy jedno neaktivní vlákno. Tento proces probíhá tak dlouho, dokud není dosaženo minimálního počtu vláken. (Frings, 2017)
- **readTimeout** – časový interval po kterém může klient otevřít nové připojení k serveru, slouží jako ochrana před denial-of-service útoky. V případě těchto útoků útočník záměrně vytěžuje server, v tomto případě by se jednalo o spuštění spousty vláken, která by byla nečinná a blokovala by ostatní uživatele, kteří server chtějí opravdu používat. (Frings, 2017)
- **maxRequestSize** – konstanta představující maximální možnou velikost požadavku od klienta v bytech. V tomto případě představuje požadavek především zasílaný obraz a jeho velikost je omezena na 10 MB. (Frings, 2017)
- **maxMultiPartSize** – představuje maximální počet části, v případě že by byl požadavek zasílán na více částí. (Frings, 2017)
- **Sessions** – všechny proměnné v této části slouží k identifikaci uživatele. Jsou uchovány po určitou dobu a znovupoužity v případě opětovného připojení uživatele služby. (Frings, 2017)

Všechny proměnné obsažené v tomto konfiguračním souboru mohou být kdykoliv změněny a projeví se po restartování aplikace. Uživatelé tak mohou aplikaci přizpůsobit výkonnostním normám jejich serverů a vyvarovat se tak problémům s nedostatkem operační paměti nebo přetížení procesoru.

Výhodou konzolové aplikace může být také její běh na pozadí a ovládání pomocí skriptů. Jako příklad lze uvést případ, kdy uživatel bude chtít službu provozovat pouze v daném časovém rozmezí. Serverový administrátor si tedy napíše skript, který v určitou hodinu aplikaci spustí, ta pak v tomto časovém rozmezí obsluhuje požadavky klientů a po uplynutí specifikované doby bude zase aplikace vypnuta. Toto všechno může probíhat automaticky, bez jediného zásahu člověka.

Výhodou konzolové aplikace může být, v neposlední řadě, také její přenositelnost. Jelikož neobsahuje uživatelské rozhraní, nenaskytnou se problémy s jeho implementací na různých operačních systémech. Taková aplikace se dá potom přeložit pomocí nativních překladačů jednotlivých operačních systémů a následnému spuštění už nebude nic bránit.

Po vytvoření projektu v *Qt Creatoru* je k dispozici soubor `main.cpp`, který obsahuje základní výpis, který otestuje funkčnost kompilátoru. Výchozím kompilátorem operačního systému *macOS* je *Clang*. Podmínkou pro použití tohoto kompilátoru je instalace nativního vývojového prostředí firmy *Apple*, které se jmenuje *Xcode* a tento kompilátor v sobě obsahuje. Po otestování správné funkčnosti začal vývoj samotné aplikace.

Struktura serverové aplikace

Serverová část aplikace obsahuje dva adresáře, které obsahují zdrojové soubory.

- Adresář `etc` – Tento adresář obsahuje výše zmíněný konfigurační soubor.
- Adresář `src` – Tento adresář obsahuje zdrojové soubory aplikace.

Soubor main.cpp

Soubor `main.cpp`, který po vytvoření projektu sloužil pouze k otestování funkčnosti kompilátoru, je ve skutečnosti hlavním stavebním kamenem každého C++ programu. V tomto souboru se specifikují základní instrukce, které celý program uvedou do chodu. Tento soubor by měl obsahovat čitelný kód, kterého by zde nemělo být mnoho. V ideálním případě by se mělo jednat pouze o jeden řádek obsahující instanci hlavní třídy aplikace a zavolání metody, která provede inicializaci a spuštění.

V případě naší aplikace je potřeba při spuštění programu správně načíst konfigurační soubor, zapnout správu sessions a uvést do provozu naslouchání HTTP serveru. Tuto funkcionalitu obsluhuje následující kód.

```
int main(int argc, char *argv[])
{
    QCoreApplication app(argc, argv);
    QString configFileName= "../etc/webapp1.ini";

    // Session store
    QSettings* sessionSettings
        = new QSettings(configFileName, QSettings::IniFormat,&app);
    sessionSettings -> beginGroup("sessions");
    RequestMapper::sessionStore
        = new HttpSessionStore(sessionSettings,&app);

    // HTTP server
    QSettings* listenerSettings
        = new QSettings(configFileName, QSettings::IniFormat,&app);
    listenerSettings -> beginGroup("listener");
    new HttpListener(listenerSettings, new RequestMapper(&app), &app);

    return app.exec();
}
```

Vytvoření instance třídy `QCoreApplication` zajistí to, že budeme moci po inicializaci aplikaci spustit. Přesněji tato třída zajišťuje vytvoření smyčky pro konzolové aplikace, která opakovaně provádí operace držící aplikaci při životě (Qt Dokumentace, 2017). Proměnná `configFileName` obsahuje relativní cestu ke konfiguračnímu souboru. Správa sessions je zapnuta vytvořením nové instance třídy `HttpSessionStore`, která je posléze přiřazena do statické proměnné sessionStore třídy RequestMapper, o které bude pojednáno později. Naslouchání http serveru je zapnuto vytvořením instance třídy `HttpListener`.

Jakmile jsou splněny výše uvedené prerekvizity, může být zavolána funkce `exec()`, která spouští provádění aplikace a vrací návratový kód. Podle návratového kódu zjistíme, zda aplikace byla zapnuta bez chyb. Číslo, představující bezproblémový start aplikace je 0.

Třída RequestMapper

Třída `RequestMapper` představuje v serverové aplikaci takovou pomyslnou křižovatku. Je to místo, kde aplikace zjistí, co přesně je po ní požadováno. Tato informace je zjišťována z URL adresy, na kterou byl požadavek zaslán. Jako příklad zvolíme následující adresu:

```
http://195.113.218.159:8090/image
```

Z této adresy vyčteme hned několik informací. Podle protokolu zjistíme, zda je komunikace šifrovaná nebo ne. V případě, že bz byla šifrovaná, by byl použit protokol https, v tomto případě ale prozatím stačí nešifrovaná komunikace http. Dále můžeme zjistit IP adresu (195.113.218.159) stroje, který službu provozuje a také na jakém portu (8090) bude požadavky přijímat. Nejdůležitější částí adresy je klíčové slovo uvedené za posledním lomítkem. V tomto případě se jedná o slovo `image`. Na základě tohoto slova `RequestMapper` určí, co se má s přijatými daty stát.

`RequestMapper` obsahuje jednu hlavní metodu, která se jmenuje `service()`. Tato metoda obsluhuje výše zmíněnou funkcionalitu a její hlavní část vypadá následovně:

```
QByteArray path = request.getPath();
if (path == "/" || path == "/hello")
{
    helloWorldController.service(request, response);
}
else if(path == "/image")
{
    imageController.service(request, response);
}
else
{
    response.setStatus(404,"Not found");
    response.write("The URL is wrong, no such service.");
}
```

Přijatý požadavek udržuje aplikace v proměnné request. Pomocí metody `getPath()` získá aplikace hodnotu za lomítkem včetně samotného lomítka. Tuto hodnotu si poté aplikace udržuje v proměnné path. Nyní již aplikace vybírá, kam přijatý požadavek pošle. V tomto případě byly nadefinovány tři URL adresy, které požadavky přijímají. Jedná se o tyto tři adresy:

- http://195.113.218.159:8090/
- http://195.113.218.159:8090/hello
- http://195.113.218.159:8090/image

Pomocí jednoduché `if-elseif-else` podmínky bylo zjištěno, jaký požadavek byl přijat. V případě, že přišel požadavek na špatnou adresu, bude uživatel upozorněn odesláním návratového kódu 404, který je všeobecně známý tím, že představuje nenalezení požadované informace. Návratový kód je doprovázen chybovou hláškou, popisující typ chyby.

V případě, že je zaslán požadavek na správnou adresu, proběhne delegování požadavku do části aplikace, která se specializuje na daný požadavek. Těmto částem se říká *controllery* a bude o nich pojednáno v následující podkapitole. Požadavky zaslané na adresu končící na lomítko, nebo `/hello` zpracovává `helloWorldController`, kterým si uživatel může otestovat funkčnost aplikace. Samotné zpracování obrazu probíhá na adrese končící na `/image` a obsluhuje ho `imageController`.

Controllery

Jak již bylo řečeno v předchozí podkapitole, controllery slouží ke zpracování požadavků se specifickými daty a vytvářejí k nim odpovědi. Každý controller dědí veškeré proměnné a metody po třídě `HttpRequestHandler`. Tato třída je součástí knihovny `QtWebApp`. Každá třída, která po `HttpRequestHandleru` dědí musí mít vlastní implementaci metody `service()`, protože tato metoda je definována jako čistě virtuální (vynucuje implementaci uživatelem). Metoda service přijímá dva parametry. Těmito parametry jsou výchozí `HttpRequest` a konečný `HttpResponse`, do kterého jsou vkládána zpracovaná data.

Jako příklad bude popsán `ImageController`, který provádí zpracování přijatého obrazu na server. Metoda `service()` tohoto controlleru vypadá následovně:

```
void ImageController::service(HttpRequest &request, HttpResponse &
    response) {
    // vytvoreni image processoru
    ImageProcessor* imgProcessor = new ImageProcessor();
    // nacteni dat
    QByteArray requestBody = request.getBody();

    QString imageString = QString::fromUtf8(requestBody);
    std::string stdImageString = imageString.toStdString();
    // prevedeni textu na obraz
```

```
    cv::FileStorage fs(stdImageString, cv::FileStorage::READ + cv::
        FileStorage::MEMORY);
    cv::Mat openCvImage;
    fs["myImage"] >> openCvImage;
    // zpracovani obrazu
    BoundingBox vysledneSouradnice = imgProcessor->showDifferences(
        openCvImage);

    //prevedeni vysledku na text
    cv::FileStorage fss(".xml", cv::FileStorage::WRITE + cv::
        FileStorage::MEMORY);
    fss << "myImage" << vysledneSouradnice.result;
    std::string matInString = fss.releaseAndGetString();
    QString strImage = QString::fromStdString(matInString);

    //ulozeni do JSONu
    QJsonObject jobject;
    jobject["result"] =  strImage;
    jobject["coordinates"] = vysledneSouradnice

    QJsonDocument doc(jobject);
    QString strJson(doc.toJson(QJsonDocument::Compact));
    // odeslani
    response.write(strJson.toUtf8(), true);
}
```

Prvním krokem po přijetí požadavku tímto controllerem, je vytvoření instance třídy `ImageProcessor`, tato třída zpracovává obraz a bude o ní pojednáno v následující podkapitole. Dalším krokem je deserializace obrazu z textové podoby, do formátu `cv::Mat`, se kterým pracuje `ImageProcessor`. Deserializace je provedena tak, že je pomocí metody `getBody()` získáno tělo požadavku. Toto tělo požadavku je převedeno z pole bytů na čistý neformátovaný text, ve kterém je zakódován uživatelem odeslaný obraz.

Datové typy `cv::FileStorage` a `cv::Mat` patří knihovně *OpenCV*. Která je prostředkem zpracování obrazu (viz Metodika). `FileStorage` dokáže ze získaného neformátovaného textu vytvořit obrázek. Tímto je zjednodušený proces deserializace a obrázek neztrácí na kvalitě, protože proběhne pouze jedna konverze.

Jakmile je obrázek deserializován do proměnné `openCvImage`, je následně předložen jako parametr metodě `showDifferences()`, o které bude taktéž pojednáno v následující podkapitole. Tato metoda vrací souřadnice tzv. bounding boxu a upravený obraz. Bounding box představuje místo v obrázku, kde byla nalezena shoda s šablonou obrázku na serveru.

Souřadnice a obraz jsou poté převedeny do textového formátu JSON a odeslány jako odpověď zpět na klientské zařízení.

Třída ImageProcessor

Tato třída je nejdůležitější třídou celé serverové části aplikace. Pomocí controlleru je do ní uživatelem zaslán zachycený obraz. Při zpracování je možno provést s obrazem různé operace. O všechny operace se stará knihovna *OpenCV*, která byla zmíněna v metodice. V následujícím textu bude popsáno, jakých funkcí *OpenCV* je v jednotlivých metodách použito.

Nejdůležitější metodou ImageControlleru je samotné vyhledání obrazu. Vstupním parametrem této metody je proměnná datového typu `cv::Mat` obsahující zachycený obraz. Prvními kroky zpracovávání je vybrání metody detekce a extrakce klíčových bodů, v tomto případě se jedná o metodu *SURF* (OpenCV team, 2017), následně vytvoření instance externí třídy *FeatureProcessing*, která obstarává přepočítávání obrazu, dále načtení šablony, která představuje hledaný objekt. V komplexnější aplikaci by mohla existovat databáze obsahující širokou škálu šablon, v této kostře aplikace bude použito přímé načtení ze souborového systému počítače.

```
std::string methodDetect = "SURF";
std::string methodExtract = "SURF";

FeatureProcessing processor = FeatureProcessing();
// řpijatý obraz
cv::Mat img_scene = image;
// hledaný obraz
cv::Mat img_object = cv::imread("/Users/tomasisa/Desktop/pencil_crop.
    jpg");
```

Po určení metody detekce a extrakce, naplnění proměnných s obrazy a vytvoření processoru může být započata samotná detekce a extrakce klíčových bodů. Prvním krokem je vytvoření detektoru a následně vytvoření dvou dynamických polí, do kterých budou detekované body vloženy. Z těchto polí budou klíčové body extrahovány do nových proměnných datového typu `cv::Mat`.

```
// detektor
cv::Ptr featureDetector = cv::FeatureDetector::create(methodDetect);
// dynamicka pole (vektory)
std::vector kp_obj;
std::vector kp_scn;

// detekce klicovych bodu
featureDetector->detect(img_scene, kp_scn);
featureDetector->detect(img_object, kp_obj);

// extraktor
cv::Ptr featureExtractor = cv::DescriptorExtractor::create(
    methodExtract);

cv::Mat desc_scn;
cv::Mat desc_obj;

// extrakce klicovych bodu
```

```
featureExtractor->compute(img_scene, kp_scn, desc_scn);
featureExtractor->compute(img_object, kp_obj, desc_obj);
```

K vyhledání shod v obraze je použito *OpenCV* třídy `FlannBasedMatcher`, která je využívána k rychlému účinnému srovnání obrazů za pomoci metody *FLANN* (OpenCV team, 2017). Z vyhledaných shod jsou potom vyfiltrovány ty nejpodobnější klíčové body, na základě kterých je vygenerována matice homografie. Homografie je transformace, která promítne body z jednoho obrazu do druhého a počítá i s případnými zkresleními druhého obrazu (Mallick, 2017). Ve vstupním obraze je následně vytvořen čtverec o velikosti hledaného objektu, který je na základě nalezené matice homografie zkreslen. Rohy tohoto čtverce jsou uloženy jako souřadnice, které jsou poté odeslány na klienta kvůli následnému zobrazení.

5.2 Implementace klientské části aplikace

Klientská část aplikace je mobilní aplikace, která běží na mobilním zařízení od firmy *Apple*, konkrétně se jedná o *iPhone*. K testování aplikace je možno využít více přístupů.

Prvním přístupem je testování aplikace na vestavěném emulátoru, který je na počítači s *macOS* přístupný. Emulátor má ale jistá omezení. Tím, že zařízení nemá vývojář fyzicky v rukou, nemůže pořizovat fotografie, naopak má na výběr pouze z pár výchozích předdefinovaných. Tento přístup je sice jednodušší pro nastavování, ale není pro typ naší aplikace vhodný.

Druhým přístupem, který bude využit i v naší aplikaci, je využití fyzického mobilního zařízení. Protože bylo vybráno zařízení od firmy *Apple* bude jeho nastavení a spárování s *Qt Creatorem* poněkud složitější než například v případě *Android* zařízení. Párování telefonu s vývojářským účtem bude popsáno v následující kapitole.

Párování iOS zařízení s počítačem

K tomu, aby mohlo být zařízení spárováno je potřeba, aby vývojář vlastnil speciální vývojářský účet u firmy *Apple* tzv. *developer account*. Jak již bylo řečeno, tak registrace takového účtu v praxi stojí určitý obnos peněz, ale v rámci akademických prací je zdarma. Po vytvoření vývojářského účtu je potřeba, aby správce univerzitního účtu pozval vývojáře do týmu. Jakmile je vývojář přijat do týmu, musí poskytnout unikátní identifikační číslo svého zařízení. Na základě tohoto čísla je následně zařízení zaregistrováno pod tým a je vytvořen osobní certifikát, který musí být nainstalován na počítači vývojáře.

Po splnění všech těchto kroků, je další postup jednoduchý. Vývojář definuje typ a verzi operačního systému zařízení ve vývojovém prostředí *Xcode* a to tyto změny následně propíše i do vývojového prostředí *Qt Creator*. S takto připraveným *Qt Creatorem* se vývojář může přesunout k vytvoření projektu.

Vytvoření projektu

Stejně jako u serverové části vytvoříme nový projekt určitého typu. V tomto případě se jedná o *Qt Quick Controls Application*. Tento typ projektu obsahuje navíc definici uživatelského rozhraní. Uživatelské rozhraní je definováno značkovacím jazykem *QML* a je pro něj v projektu vytvořen zvláštní soubor, který je napojen na hlavní funkcionality aplikace. *QML* dává vývojáři plnou kontrolu nad definováním vzhledu aplikace, obsahuje širokou škálu různých tlačítek, textových polí, štítků apod. Všechny tyto komponenty mohou být přizpůsobovány potřebám uživatele pomocí aplikace stylů. O této problematice bude detailněji pojednáno při popisu uživatelského rozhraní.

Po vytvoření projektu může být definována struktura aplikace.

Struktura klientské aplikace

Klientská část aplikace obsahuje následující třídy a soubory, které se starají o její chod.

- main.cpp
- IOSCamera
- Sender
- main.qml

Soubor main.cpp

Stejně tak jako u serverové části aplikace je tento soubor spouštěčem celé aplikace. Obsahuje pouze pár odlišností, které spojují logiku aplikace s uživatelským rozhraním.

```
int main(int argc, char *argv[])
{
    QApplication app(argc, argv);
    qmlRegisterType("IOSCamera", 1, 0, "IOSCamera");
    qmlRegisterType("Sender", 1, 0, "Sender");
    QQmlApplicationEngine engine;
    engine.load(QUrl(QStringLiteral("qrc:/main.qml")));
    return app.exec();
}
```

Vytvořením instance třídy `QApplication` jsou načteny potenciální vstupní parametry aplikace a je připravena proměnná, která provede na konci `mainu` start celé aplikace.

Metodou `qmlRegisterType()` jsou zpřístupněny interní třídy aplikace uživatelskému rozhraní. Díky volání této funkce je pak možno v *QML* souboru přistupovat k metodám těchto tříd. Tato metoda přijímá při svém volání 4 následující parametry:

- uri – Název třídy k zpřístupnění (Qt Dokumentace, 2017)
- versionMajor – hlavní verze třídy (Qt Dokumentace, 2017)
- versionMinor – vedlejší verze třídy (Qt Dokumentace, 2017)
- qmlName – název, pod kterým bude třída přístupná v *QML* souboru (Qt Dokumentace, 2017)

Po zpřístupnění tříd uživatelskému rozhraní je vytvořena instance třídy `QQml Application Engine`. Tato třída provádí načtení a vykreslení uživatelského rozhraní z *QML* souboru (Qt Dokumentace, 2017). Parametrem při volání konstruktoru třídy je relativní cesta ke *QML* souboru.

Spuštění aplikace je provedeno zavoláním metody `exec()` na instanci třídy `QApplication`.

Získání fotografie

Získání fotografie zajišťuje třída `IOSCamera`. Jelikož je operační systém iOS naprogramován v programovacích jazycích *Objective-C* a *Swift* je zapotřebí, aby byl k připojení na fotoaparát zařízení použit *Objective-C* kód. Výchozím programovacím jazykem projektu je ale zvolen jazyk *C++*, jak je tedy možné docílit spuštění *Objective-C* kódu v takovém projektu? Vývojáři vývojového prostředí *Qt Creator* na tento problém mysleli a umožnili spuštění *Objective-C* kódu za těchto podmínek:

- Vývojář musí do hlavičky souboru přiložit hlavičkové soubory, které jsou zapotřebí k překladu *Objective-C* kódu.
- Implementační soubor musí být přejmenován a to tak, že je koncovka `.cpp` změněna na `.mm`.

Takto upravená třída je poté připravena pro psaní kódu jiného jazyka.

Pro umožnění překladu byl soubor `ioscamera.cpp` přejmenován na `ioscamera.mm` a byly přiloženy následující hlavičkové soubory:

- `qpa/qplatformnativeinterface.h`
- `UIKit/UIKit.h`

Nejdůležitějšími metodami třídy `IOSCamera` jsou metody `open()` a metoda `didFinishPickingMediaWithInfo`.

Metoda `open()` otevře novou obrazovku, přesněji nativní *iOS* aplikaci pro pořizování fotografií, která překryje dosavadní uživatelské rozhraní a bude jej překrývat do té doby, než uživatel zvolí fotografii. Metoda vypadá následovně:

```
// inicializace kamery
void IOSCamera::open()
{
    // vytvoreni obrazovky, ktera prekryje stavajici obrazovku
    UIView *view = static_cast(
```

```
                QGuiApplication::platformNativeInterface()
                ->nativeResourceForWindow("uiview", window()));
    UIViewController *qtController = [[view window] rootViewController
        ];

    // vytvoreni kontroleru, ktery slouzi k~zachyceni obrazu
    UIImagePickerController *imageController = [[[
        UIImagePickerController alloc] init] autorelease];
    [imageController setSourceType:
        UIImagePickerControllerSourceTypePhotoLibrary];
    [imageController setDelegate:id(m_delegate)];

    // zobrazeni kontroleru
    [qtController presentViewController:imageController animated:YES
        completion:nil];
}
```

Jakmile je zvolena fotografie, je vyvolán signál, který zahájí provádění metody `didFinishPickingMediaWithInfo`. Metoda vypadá následovně:

```
- (void)imagePickerController:(UIImagePickerController *)picker
    didFinishPickingMediaWithInfo:(NSDictionary *)info
{
    Q_UNUSED(picker);

    // vytvoreni cesty, kam se fotografie ulozi
    NSString *path = [NSSearchPathForDirectoriesInDomains(
        NSDocumentDirectory, NSUserDomainMask, YES) objectAtIndex:0];
    path = [path stringByAppendingString:@"/test.jpg"];

    // ulozeni fotografie
    UIImage *image = [info objectForKey:
        UIImagePickerControllerOriginalImage];
    [UIImagePNGRepresentation(image) writeToFile:path options:
        NSAtomicWrite error:nil];

    // aktualizace promenne s~cestou k~souboru a~vyvolani Qt signalu
    m_iosCamera->m_imagePath = QString::fromNSString(path);
    emit m_iosCamera->imagePathChanged();


    // vraceni obrazovky s~nasi aplikaci
    UIViewController *rvc = [[[UIApplication sharedApplication]
        keyWindow] rootViewController];
    [rvc dismissViewControllerAnimated:YES completion:nil];
}
```

Nejdříve je vytvořena cesta, kam bude na zařízení fotografie uložena. Pomocí `NSDocumentDirectory` je získáno výchozí umístění dokumentů a je do něj vložena fotografie s vybraným názvem. V tomto případě se jedná o název **test.jpg**. Po uložení fotografie je vyvolán signál, který vybranou cestu k souboru uloží do běžící

instance třídy `IOSCamera` pro pozdější užití. Po odeslání signálu je nativní aplikace fotoaparátu skryta a aplikace se přesune zpět na úvodní obrazovku.

Tento přístup zachycení a odeslání obrazu fungoval pouze do určité verze vývojového prostředí *Qt* a to přesněji do verze 5.7. Od verze 5.8 byla totiž plně zprovozněna *Qt Multimedia* pro *iOS*, která výrazně ulehčují práci s fotoaparátem a umožňují připojení jednoduše přes *QML*.

V tuto chvíli je fotografie připravena ke zpracování a odeslání na server.

Odeslání fotografie

O odeslání fotografie na stranu serveru se stará třída `Sender`. Třída obsahuje proměnnou typu `QNetworkAccessManager`, která se stará o propojování obou stran. Třída také obsahuje 3 metody, které připravují odesílaná data a zpracovávají přijatá data. Jako první bude popsána metoda `serializeMat(cv::Mat mat)`.

Metoda `serializeMat(cv::Mat mat)` přijímá jako parametr obraz datového typu `cv::Mat` a jejím výstupem je proměnná datového typu `string` obsahující obraz v textové podobě. Metoda vypadá následovně:

```
std::string serializeMat(cv::Mat mat){
    cv::FileStorage fs(".xml", cv::FileStorage::WRITE + cv::FileStorage
        ::MEMORY);
    fs << "myImage" << mat;
    std::string matInString = fs.releaseAndGetString();
    return matInString;
}
```

Nejdříve je vytvořena proměnná `fs` datového typu `cv::FileStorage`. Tato proměnná zajišťuje pojmenování a převod obrazu. Pomocí `<<` operátoru je vložen do *FileStorage* název obrazu (*myImage*) a následně celý obraz `mat`.

Takto definovaný *FileStorage* je následně převeden na `string` pomocí metody `releaseAndGetString()`. Tato stringová proměnná je vrácena k dalšímu zpracování aplikací.

Další metodou této třídy je `sendImage(QString directory)`. Jak již anglický název napovídá, tak tato metoda obstarává odeslání fotografie na server a zajišťuje správné přijetí odpovědi. Vstupním parametrem je cesta k souboru fotografie, získaná výše popsanou třídou `IOSCamera`. Struktura `Senderu` je následující:

```
void Sender::sendImage(QString directory){
    manager = new QNetworkAccessManager(this);
    connect(manager, SIGNAL(finished(QNetworkReply*)),
            this, SLOT(replyImageFinished(QNetworkReply*)));
    QNetworkRequest request(QUrl("http://195.113.218.159:8090/image"))
        ;
    request.setHeader(QNetworkRequest::ContentTypeHeader, "text/
        plaintext");

    cv::Mat mat;
    mat = cv::imread(directory.toStdString());
```

```
    std::string matInString = serializeMat(mat);

    QString str = QString::fromStdString(matInString);
    manager->post(request, str.toUtf8());
}
```

Po inicializaci výše zmíněné proměnné datového typu `QNetworkAccessManager` je volána funkce `connect()`. Tato část je velmi důležitá, protože propojuje odeslání požadavku s přijetím odpovědi. Propojení zapříčiní to, že jakmile dostane `manager` signál od serverové části aplikace, obsahující odpověď na požadavek, začne provádění metody `replyImageFinished()`, které bude odpověď předána jako parametr. Po propojení událostí a přípravě na odpověď může být odeslán konkrétní požadavek.

Požadavek je vytvořen vytvořením proměnné `request` datového typu `QNetwork Request`. Konstruktor této třídy přijímá URL, na kterou bude zaslán požadavek. Následně je `requestu` nastavena hlavička, konkrétně typ zasílaných dat. V tomto případě se jedná o prostý text. Poté je do proměnné `mat` načten obrázek pořízený fotoaparátem a pomocí metody `serializeMat` je převeden na `string`. Před odesláním je ještě `std::string` převeden na `QString`, aby manager nastavil správný formát textu. Odeslání požadavku je provedeno metodou `post()`, která jako parametr převezme vytvořený `request` a data převedena do *UTF-8*.

V tuto chvíli aplikace čeká na odpověď serveru a vyvolání signálu pro zavolání metody `replyImageFinished`.

Jakmile je získána odpověď serveru začne provádění metody `replyImageFinished(QNetworkReply *reply)`, jejímž parametrem je samotná odpověď s výslednými daty. Zpracování odpovědi probíhá následovně:

```
void Sender::replyImageFinished(QNetworkReply *reply)
{
    if (reply->error()){
        qDebug() << reply->errorString();
    }else{
        QString fullJson = reply->readAll();
        reply->seek(0);

        QJsonDocument doc = QJsonDocument::fromJson(fullJson.toUtf8());
        QJsonObject jo = doc.object();

        QString imageString = jo["result"].toString();
        std::string stdImageString = imageString.toStdString();

        cv::FileStorage fs(stdImageString, cv::FileStorage::READ + cv::
            FileStorage::MEMORY);
        cv::Mat openCvImage;
        fs["myImage"] >> openCvImage;

        QString resultPath = QStandardPaths::writableLocation(
            QStandardPaths::HomeLocation).append("/Documents/
            resultImage.bmp");
```

```
        resultPath.replace("/private", "");

        imwrite(resultPath.toStdString(), openCvImage);
        emit objectFound("file://" + resultPath);
    }
    reply->deleteLater();
}
```

Pomocí metody `error()` je zkontrolováno, zda přišla data ze serveru v pořádku. Jestliže dojde k chybě, je v konzoli vypsána chybová hláška. V opačném případě je z odpovědi načten prostý text, který je formátován jako *JSON*. Tento text je deserializován pomocí `QJsonDocument`. Pomocí `cv::FileStorage` je převeden text zpět na obrázek. Jakmile je obrázek připraven pro uložení na zařízení, tak je pomocí `QStandardPaths` zjištěna cesta ke složce dokumentů, kam je obrázek následně uložen pomocí metody `imwrite`. Po uložení obrázku je pomocí klíčového slova `emit` vyvolán signál, kterému je předána cesta k obrázku a tento signál je zaslán do uživatelského rozhraní, které následně zobrazí výsledek zpracování obrazu. Metoda `deleteLater()` ukončí vlákno, které provádělo zadaný požadavek.

Uživatelské rozhraní

Všechny výše popsané součásti se střetávají na jednom místě a to v uživatelském rozhraní, které celou aplikace spojuje v jeden celek. Uživatelské rozhraní je vytvořeno pomocí jazyka *QML*, který je součástí *Qt*. S příchodem *Qt Quick Controls 2* byla uvedena komponenta *SwipeView*, která pomáhá na mobilním zařízením rozdělit funkcionalitu aplikace na několik záložek. Aplikace obsahuje 3 následující záložky.

- **Úvod** – na úvodní obrazovce se nachází pouze tlačítko pro odeslání pořízené fotografie
- **Kamera** – na této záložce uživatel fotí zasílanou fotografii
- **Výsledek** – na této záložce je uživateli prezentován výsledek hledání objektu

Rozvržení pomocí *SwipeView* bylo vytvořeno následovně.

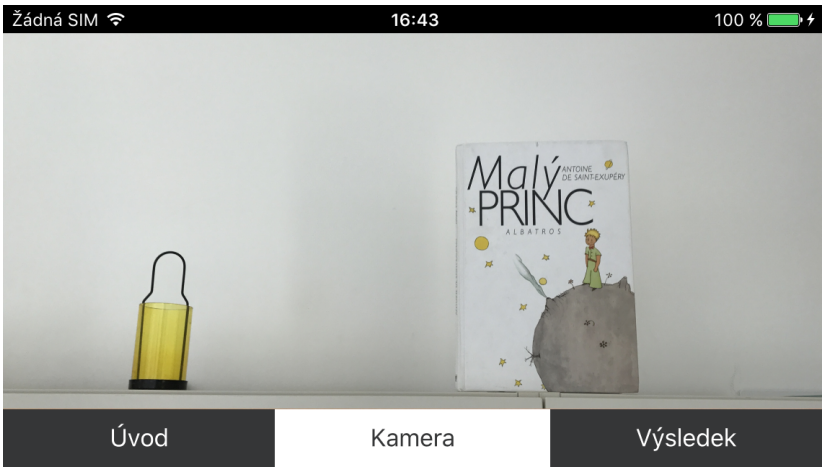
```
SwipeView {
    id: swipeView
    anchors.fill: parent
    currentIndex: tabBar.currentIndex
    Page {
        //obsah prvni zalozky
    }
    Page {
        //obsah druhe zalozky
    }
    Page{
        //obsah treti zalozky
    }
```

```
}

footer: TabBar {
    id: tabBar
    currentIndex: swipeView.currentIndex
    TabButton {
        text: qsTr("Úvod")
    }
    TabButton {
        text: qsTr("Kamera")
    }
    TabButton{
        text: qsTr("Výsledek")
    }
}
```

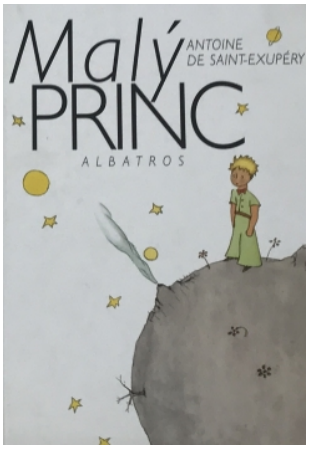
Celé rozhraní se skládá ze tří *Page* s připojenou patičkou, která zajišťuje to, že uživatel ví, na které záložce se zrovna nachází. V následujícím textu bude na jednoduchých příkladech předvedeno, jak uživatelské rozhraní a celá aplikace funguje.

Po zapnutí aplikace se uživateli jako první zobrazí záložka s fotoaparátem. V tomto případě bude zpracovávána fotografie knihy, která je postavena na polici.



Obrázek 5: Záložka s fotoaparátem

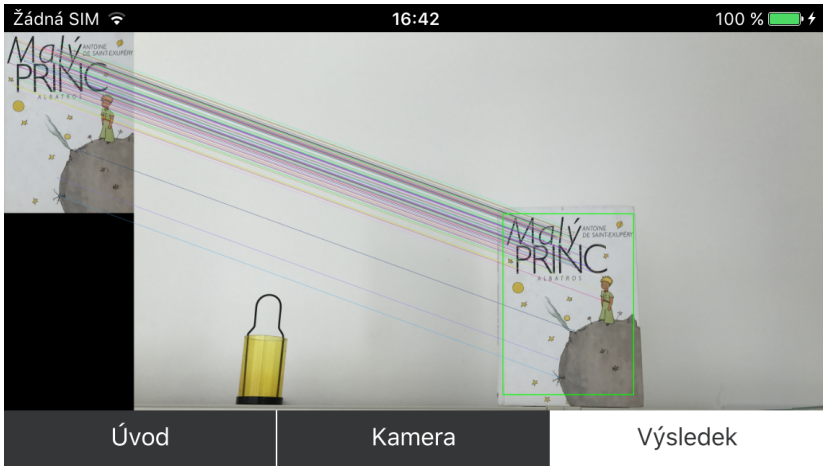
Fotografie je pořízena klepnutím na kterékoliv místo panelu. Z této fotografie byl také vyříznut vzor, který bude ve scéně hledán. Vzor vypadá následovně:



Obrázek 6: Hledaný vzor uložený na serveru

Po pořízení fotografie je uživatel přesměrován na první záložku, kde si může vybrat ze dvou možností. První možností je vrácení se zpět na záložku s fotoaparátem a pořízení nového obrázku, nebo druhou možností je kliknutí na tlačítko *Odeslat*, které zahájí síťový přenos a zpracování obrazu.

Po kliknutí na tlačítko *Odeslat* je obraz převeden do textové podoby a odeslán ve formátu *JSON* na server. Ve chvíli, kdy server zpracuje data a klient přijme odpověď, je uživatel přesměrován na poslední záložku s názvem *Výsledek*. Na této záložce je zobrazen výstup algoritmu, který obraz zpracovával. V levém horním rohu výstupu je k dispozici vzor, který byl hledán a na pravé straně obrazovky je pořízená fotografie, obohacena o nalezené klíčové body a rámeček, který ukazuje místo, kde byl objekt nalezen.



Obrázek 7: Obrazovka s výstupem

V úvodu práce bylo pojednáno o možných využitích takovéto aplikace v praxi. Výše uvedený příklad s knihou může být využit například v knihovně, kde si uživatel chce zjistit informace o knize. Následující dva příklady předvedou aplikaci v prostředí simulovaného obchodu s potravinami a při cestování po světě. V horní části obrázku je možnost vidět detekci obalu produktu v obchodě a ve spodní části obrázku detekci světoznámé budovy parlamentu v Londýně. Jako vzor byla vybrána věž, která skrývá známý zvon Big Ben. Její fotografie byla získána na webových stránkách firmy *Lego* (UK Parliament, 2013). Na výchozí fotografii je vyobrazena celá budova parlamentu a je získána ze stránek Londýnského průvodce (GetYourGuide, 2017).

Obrázek 8: Příklad s obchodem a budovou parlamentu

6 Diskuze

Po dokončení implementace obou aplikací bylo dosaženo následujícího výsledku. Uživateli je pomocí uživatelského rozhraní umožněno zachycení fotografie, odeslání fotografie ke zpracování na server, získání informací o detekovaném objektu a zobrazení objektu pomocí rámečku. Serverová část dokáže obsluhovat několik požadavků v jednu chvíli a má implementovanou jednu metodu zpracování obrazu.

Aplikace dokáže detekovat pouze jeden vzor, který je na serveru pevně definován a musí být změněn v případě, že by uživatel chtěl hledat jiný objekt. V této oblasti je prostor pro zlepšení, protože v tuto chvíli neexistuje jednoduché řešení detekce více objektů najednou. Další nevýhodou je doba, po kterou trvá jedna iterace požadavku mezi serverem a klientem. V tuto chvíli se tento čas pohybuje někde kolem deseti až patnácti sekund a to v dnešní uspěchané době není ideální. Po uplynutí tak dlouhé doby by běžný uživatel aplikaci vypnul a odinstaloval, protože by pro něj byla nepoužitelná. Iterace mezi serverem a klientem je takto dlouhá, protože cílem práce bylo pouze navržení architektury přenosu a její otestování. Optimalizace detekčních algoritmů je mimo rámec této práce.

Při vývoji vzniklo jedno omezení ze strany hardwaru. Nedostatkem bylo, v dnešní době, již starší mobilní zařízení *iPhone 5*. Z důvodu vadného konektoru telefonu byly zničeny dva napájecí kabely a staří telefonu se podepsalo také na výdrži baterie. Server je poháněn velmi výkonným stolním počítačem od firmy *Apple*, a fungoval bez chyb.

V průběhu implementace vzniklo několik menších problému. Prvním problémem, který byl řešen, byl překlad a přiložení knihovny *OpenCV* k projektu. Vyřešením problému bylo upravení konfigurace překladu v programu *CMake* a downgradování verze *OpenCV* z verze 3 na verzi 2.4.3.

Jedním z hlavních a opakujících se problémů bylo udržení funkčnosti a konzistence aplikace. V průběhu vývoje došlo ze strany *Qt*, *Xcode* a *OpenCV* k řadě aktualizací. Z tohoto důvodu se často stávalo, že některé části aplikace přestaly po aktualizaci fungovat. Většina těchto problémů byla vyřešena různými konfiguracemi operačního systému, nebo projektu.

V jedné z aktualizaci, byla ale bohužel zanesena chyba, která na nějakou dobu úplně znemožnila překlad aplikace pro *iOS*. Po delším testování bylo zjištěno, že se jednalo o problém s připojením k fotoaparátu mobilního zařízení. Prvotní verze připojování fotoaparátu pomocí *Objective-C* byla vyměněna za jednodušší připojování pomocí *Qt Multimedia*. Poté již bylo možné aplikaci přeložit, ale aplikace fotoaparát stále nemohla používat. Finálním řešením tedy bylo upravení souboru, který má na starosti přístupová práva k jednotlivým částem *iOS* zařízení.

Prvotní verze síťové komunikace byla naimplementována pouze pomocí čistého C++, tento přístup nebyl pro aplikaci vhodný, protože server dokázal v jednu chvíli obsluhovat pouze jednoho klienta a dokázal obsluhovat pouze jeden požadavek. Díky tomu byl server těžko rozšiřitelný a museli by se vytvořit třídy, které by podporovaly multivláknovost a obsluhu více požadavků. Řešením tohoto problému bylo použití

knihovny *QtWebApp*, která představuje multivláknový *HTTP* server a lze jednoduše definovat nové požadavky.

Další vývoj aplikace již proběhl bez větších nesnází a bylo dosaženo stanoveného cíle.

7 Závěr

Cílem práce bylo vytvořit klient–server aplikaci, která umožní detekci objektů v zachyceném obraze. Po prozkoumání existujících řešení byly vybrány a nastudovány technologie, které byly potřebné k započetí fáze implementace. Po vyřešení mnoha nesnází byla ukončena fáze implementace a byla vytvořena aplikace, která splňuje cíl práce.

Podařilo se vytvořit klient–server aplikaci, jejíž klientskou stranu může mít uživatel nainstalovánu na svém telefonu a díky tomu ji může použít kdekoliv má přístup k internetovému připojení. Uživatel pořídí fotografii, odešle ji na server a nechá si zaslat odpověď v podobě obrazu s vyznačeným detekovaným objektem. Aplikace má do budoucna mnoho prostoru pro rozšíření. Mezi nejzajímavější rozšíření může patřit například zajištění podpory pro operační systém Android, vytvoření databáze vzorů, která by rozšířila využitelnost aplikace a v neposlední řadě může být vylepšením také implementace zpracování obrazu v reálném čase.

Při psaní aplikace jsem se naučil řešit mnoho programátorských problémů, lépe komunikovat s lidmi a také si osvojil základy nejrůznějších knihoven, frameworků a vývojových prostředí.

8 Přílohy

Datové CD obsahující:

- Zdrojový kód serverové aplikace
- Zdrojový kód klientské aplikace

9 Reference

Abeles, P., *BoofCV* [online]. 2012, 2015 [cit. 2015-12-07]. Dostupné z: http://boofcv.org/index.php?title=Main_Page.

Apple, *About Objective-C* [online]. 2017 [cit. 2017-05-11]. Dostupné z: https://developer.apple.com/library/content/documentation/Cocoa/Conceptual/ProgrammingWithObjectiveC/Introduction/Introduction.html.

CMake team, *CMake* [online]. 2017 [cit. 2017-05-11]. Dostupné z: https://cmake.org/.

Deligiannidis, L., *Emerging trends in image processing, computer vision and pattern recognition* [online]. 1st edition. Waltham, MA: Elsevier, 2014, pages cm [cit. 2015-12-27]. ISBN 978-012-8020-456..

Frings, S., *QtWebApp HTTP server in C++* [online]. 2017 [cit. 2017-05-11]. Dostupné z: http://stefanfrings.de/qtwebapp/index-en.html.

Gammeter S., Gassmann A., Bossard L., Quack T. a Van Gool L., *Server-side object recognition and client-side object tracking for mobile augmented reality* [online]. Zurich, 2010 [cit. 2017-04-30]. Dostupné z: http://homes.esat.kuleuven.be/ƙonijn/publications/2010/eth_biwi_00782.pdf.

GetYourGuide, *Big Ben* In: Get Your Guide [online]. Londýn, 2017 [cit. 2017-05-16]. Dostupné z: https://cdn.getyourguide.com/niwziy2l9cvz/3TLDUtWWacU8gwW0o6m4km/7e9164a5dd180d8d29dec20142897f24/london-bigben-1500x850.jpg.

Held, G., *Server management* [online]. Boca Raton, FL: Auerbach [cit. 2017-05-11]. Best practices series (Boca Raton, Fla.). ISBN 08-493-9823-1.

Laganiére, R., *OpenCV 2 computer vision application programming cookbook: over 50 recipes to master this library of programming functions for real-time computer vision* [online]. Birmingham, U.K.: Packt Open Source Pub., 2011, iii, 287 p. [cit. 2015-12-07]. ISBN ISBN 978-1-849513-24-1.

Lelis Baggio, D., *Mastering OpenCV with practical computer vision projects* Birmingham, UK: Packt Pub., 2012.

Mallick, S., *Learn OpenCV online* [online]. 2017 [cit. 2017-05-11]. Dostupné z: http://www.learnopencv.com/homographyexamples-using-opencv-python-c/.

Murino, V., *New trends in image analysis and processing - ICIAP 2013: ICIAP 2013 International Workshops, Naples, Italy, September 9-13, 2013. Proceedings* [online]. 1st edition. pages cm [cit. 2015-12-27]. ISBN 36-424-1189-4.

OpenCV team, *OpenCV library* [online]. 2017 [cit. 2017-05-11]. Dostupné z: http://opencv.org/.

Qt Docs, *Qt Documentation* [online]. Norsko, 2017 [cit. 2017-05-11]. Dostupné z: http://doc.qt.io/.

Rouse, M., *What is library?* [online]. 2005 [cit. 2015-12-07]. Dostupné z: http://searchsqlserver.techtarget.com/definition/library.

scikit-image, *scikit-image: Image processing in python* [online]. 2011, 2015 [cit. 2015-12-07]. Dostupné z: http://scikit-image.org/.

Slyce.it, *What's Next in Image Recognition: 4 Trends to Watch* [online]. 2015 [cit. 2015-12-07]. Dostupné z: http://blog.slyce.it/2015/08/11/whats-next-in-image-recognition-%C2%AD4-trends-to-watch/.

UK Parliament, *Parliament* In: Lego [online]. Londýn, 2013 [cit. 2017-05-16]. Dostupné z: https://www.lego.com/en-us/architecture/explore/21013-big-ben.

Vedaldi, A. a Fulkerson, B., *VLFeat: An Open and Portable Library of Computer Vision Algorithms* [online]. 2008, 2015 [cit. 2015-12-07]. Dostupné z: http://www.vlfeat.org/.

W3C, *JSON Introduction* [online]. 2017 [cit. 2017-05-11]. Dostupné z: https://www.w3schools.com/js/js_json_intro.asp.

Wilson, P. a Fernandez, J., *Facial feature detection using Haar classifiers* USA, 2006.