Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Extrakce údajů z heterogenních dokumentů pomocí šablon

Plzeň 2020 Patrik Patera

Místo této strany bude zadání práce.

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 19. května 2020

Patrik Patera

Poděkování

Tímto bych rád poděkoval Ing. Kamilu Ekšteinovi, Ph.D. za jeho čas, cenné rady a ochotu při vedení této diplomové práce. Dále děkuji své rodině za podporu během celého studia.

Abstract

This master's thesis deals with the challenges of automatic content extraction from regions of interest located in scanned documents (images) on the basis of user's defined templates, as a part of the computer vision domain. The main goal was to analyse common techniques and frameworks used for digital image processing followed by optical character recognition (OCR) performed in the text areas. In consonance with the analysis, the software for template creation with an extensive user graphics interface was designed and implemented as well as the module to handle and extract the regions of interest defined by an appropriate template from scanned documents and subsequently passing them to the OCR system. The implemented algorithms were evaluated to get an overview of their functionality and robustness with regard to the subject matter, the results of which are summarized in the conclusion. As a result of the evaluation, the best-rated algorithms with configurable input parameters are set as the default ones in the application.

Abstrakt

Tato diplomová práce se zabývá problémy z oblasti počítačového vidění k automatizované extrakci užitečných informací z naskenovaných dokumentů (obrazových dat) dle uživatelsky definovaných šablon. Hlavním cílem bylo analyzovat používané techniky a nástroje zaměřující se na zpracování digitálních snímků s následným optickým rozpoznáním znaků (OCR) z textových oblastí. Na základě analýzy byl navržen a implementován software pro tvorbu šablon dokumentů s grafickým uživatelským rozhraním a modul pro práci s naskenovanými dokumenty, který podle příslušné šablony extrahuje oblasti s užitečnými informaci a ty předá OCR systému. Implementované algoritmy byly podrobeny evaluačním testům k získání přehledu o jejich funkčnosti a robustnosti s ohledem k zamýšlenému účelu, jejichž výstup byl shrnut v závěru této práce. Nejlépe vyhodnocené algoritmy s konfigurovatelnými vstupními parametry jsou v aplikaci nastaveny jako výchozí.

Obsah

1 Úvod **10**

2 Požadavky **11**

3 Zpracování dokumentů **12**
3.1 Haru . . . 12
3.2 Apache PDFBox . . . 13
3.3 PoDoFo . . . 13

4 Nástroje počítačového vidění **15**
4.1 OpenCV . . . 15
4.2 DLib . . . 17

5 Analýza a zpracování obrazu **19**
5.1 Barevný prostor a hloubka . . . 19
5.1.1 Binární obrázek . . . 20
5.1.2 Šedotónový . . . 20
5.1.3 RGB . . . 20
5.1.4 YC_bC_r . . . 21
5.2 Histogram . . . 21
5.3 Filtrování obrázku . . . 22
5.3.1 Lineární filtrování . . . 23
5.3.2 Hraniční efekt . . . 23
5.3.3 Nelineární filtrování . . . 24
5.4 Prahování . . . 25
5.4.1 Otsu . . . 25
5.4.2 Ekšteinova metoda . . . 27
5.4.3 Wolfova metoda . . . 29
5.5 Morfologické operace . . . 29
5.5.1 Eroze . . . 30
5.5.2 Dilatace . . . 31
5.5.3 Otevření . . . 32
5.5.4 Uzavření . . . 32
5.5.5 Obrys . . . 32
5.6 Detekce hran . . . 33
5.6.1 Robertsův operátor . . . 33

5.6.2 Prewittové operátor . . . 34
5.6.3 Sobelův operátor . . . 34
5.6.4 Robinsonův operátor . . . 34
5.6.5 Kirschův operátor . . . 34
5.6.6 Cannyho hranový detektor . . . 34
5.7 Odšumění . . . 36
5.7.1 Non-Local Means Denoising . . . 36
5.7.2 Stochastic Image Denoising . . . 37
5.8 Škálování . . . 39
5.8.1 Interpolace nejbližším sousedem . . . 39
5.8.2 Bilineární interpolace . . . 40
5.8.3 Bikubická interpolace . . . 40
5.8.4 Lanczosova interpolace . . . 41
5.8.5 Super-Resolution CNN . . . 42
5.9 Detekce natočení . . . 43
5.9.1 Houghova transformace pro detekci přímek . . . 43
5.9.2 Detekce natočení ohraničujících rámečků . . . 45
5.10 Hledání vzoru . . . 46
5.10.1 Maskování . . . 47
5.10.2 Normalizovaná suma rozdílu čtverců . . . 48
5.10.3 Normalizovaná vzájemná korelace . . . 48
5.10.4 Normalizované koeficienty vzájemné korelace . . . 48
5.10.5 Index strukturální podobnosti . . . 49

6 Nástroje OCR **50**
6.1 Optické rozpoznání znaků . . . 50
6.1.1 Předzpracování . . . 50
6.1.2 Segmentace . . . 52
6.1.3 Extrakce klíčových bodů . . . 53
6.1.4 Rozpoznání . . . 54
6.1.5 Postprocessing . . . 55
6.2 Tesseract-OCR . . . 55
6.2.1 Verze 4 . . . 55

7 Nástroje pro šablonovací software **58**
7.1 Qt . . . 59
7.2 JavaFX . . . 60

8 Modul počítačového vidění a zpracování dokumentů 61
8.1 Funkce nástroje OpenCV . . . 61
8.1.1 Struktura matice . . . 61
8.2 Funkce nástroje Tesseract-OCR . . . 66
8.2.1 Třída TessBaseAPI . . . 66
8.3 Implementace modulu . . . 69
8.3.1 Třída s konfiguracemi . . . 69
8.3.2 Práce s dokumenty . . . 70
8.3.3 Struktura šablony dokumentu . . . 81
8.3.4 Práce s obrázkem . . . 83
8.3.5 Rozpoznání textu . . . 91

9 Software pro práci se šablonami 93
9.1 Implementace aplikace . . . 95
9.1.1 Model dokumentu . . . 95
9.1.2 Ovladače aplikace . . . 97
9.1.3 Pohledy . . . 98
9.1.4 Struktura šablony . . . 99
9.1.5 Grafické uživatelské rozhraní . . . 100

10 Dosažené výsledky 102
10.1 Detekce natočení . . . 102
10.2 Odšumění obrazu . . . 105
10.3 Binarizace obrazu . . . 107
10.4 Hledání vzoru . . . 110
10.5 OCR . . . 112

11 Závěr 116

Literatura 117

Seznam zkratek 122

Seznam obrázků 126

Seznam tabulek 126

A Uživatelská příručka 127
A.1 Struktura projektu . . . 127
A.2 Překlad aplikace . . . 127
A.3 Manuál k obsluze aplikace . . . 128

1 Úvod

Zadání této práce vzešlo z potřeb firmy Palaxo Development s. r. o. za účelem rozvoje její digitální platformy CIRCULARO pro zpracování dokumentů v rámci tzv. bezpapírové kanceláře komunikující prostřednictvím REST API (Representational State Transfer API) s pokročilým webovým rozhraním. Platforma umožňuje nejenom vytvářet, upravovat, archivovat a podepisovat dokumenty, ale i je skenovat (např. mobilním zařízením), zasílat pomocí e-korespondence, extrahovat a analyzovat jejich obsah s využitím technik v oblasti zpracování přirozeného jazyka (NLP – *Nature Language Processing*).

Nicméně dosavadní implementované řešení je schopné zpracovat pouze dokumenty textového charakteru. Obsah jakéhokoliv naskenovaného dokumentu není možné analyzovat, a tudíž je pro platformu takřka nedostupný. Pakliže je zkonstruován rozsáhlejší fulltextový dotaz, např. nalezení dostupných dokumentů obsahujících konkrétní jméno osoby, nebude dokument tohoto typu zahrnut v nalezených výsledcích. K dispozici mohou být pouze metadata, která jsou ovšem závislá na konkrétním typu souboru.

Proto je hlavním cílem této diplomové práce navrhnout a implementovat modul umožňující zpracovávat naskenované dokumenty běžně používaného formátu PDF, respektive TIFF, JPG, PNG apod., s heterogenním obsahem (obecně obrazová data) a pomocí metod počítačového vidění extrahovat oblasti s užitečnými informacemi podle uživatelsky definovaných šablon a integrovat systém pro rozpoznávání textu. Taktéž je zapotřebí vyvinout interaktivní software pro vytváření a manipulaci se šablonami pro konkrétní vstupní dokument. Mimo jiné bude disponovat funkcí pro nalezení nejvhodnější šablony a následným provedením zvolených akcí nad každou definovanou oblastí v rámci šablony, jako je extrakce či anonymizace informací (např. osobních údajů). Implementované řešení bude podrobeno evaluačním testům ověřujícím stabilitu algoritmů a úspěšnost shody nalezené šablony, ale také přesnost extrahovaného obsahu vůči referenčnímu dokumentu z testovací množiny dat.

2 Požadavky

V prvé řadě by měla výsledná aplikace podporovat soubory obsahující obrazová data běžně dostupných formátech, a to:

- **rastrové formáty** – TIFF (TIF), PNG, JPEG (JPG), BMP a PGM,
- **přenosný formát dokumentů** – PDF (*Portable Document Format*).

Na základě těchto specifikovaných formátů budou vybrány vhodné knihovny a nástroje, které umožňují zpracovat a provádět základní operace nad obrazovými daty. Většina skenovacích zařízení nabízí uložit skenovaný dokument (snímek) v PDF jako výchozí volbu. PDF je v dnešní době takřka standardem pro archivaci a přenos dokumentů. Aplikace bude pracovat výhradně s obrazovými daty, proto není nutné uvažovat jakoukoliv textovou vrstvu v PDF formátu, tuto vlastnost již zmíněná platforma poskytuje s využitím nástroje **Apache Tika**.

Hlavní funkcí aplikace bude především extrakce informací ze zájmových oblastí, respektive textových oblastí, ve vstupním dokumentu, které budou dále distribuovány v JSON (*JavaScript Object Notation*) formátu. Interní reprezentace šablony obsahující uživatelsky definované zájmové oblasti je čistě v kompetenci autora této práce. Každopádně součástí musí být software – šablonovací nástroj – pro práci s těmito šablonami.

Samotná šablona by měla splňovat následující požadavky:

- **export do formátu JSON** – uložení struktury a obsahu šablony ve formátu JSON,
- **pozice a velikost oblastí** – obdélníkový box reprezentující zájmovou oblast čtveřicí celých čísel $\{x, y,$ šířka a výška$\}$,
- **identifikace dokumentu** – textový identifikátor originálního dokumentu, pro který byla šablona vytvořena,
- **typ akce** – definování akce pro konkrétní oblast: **extrakce** nebo **anonymizace údajů**, jež bude v dané oblasti provedena.

3 Zpracování dokumentů

Zpracování, tj. načtení, editování a uložení vstupního souboru je klíčovou premisou aplikace, respektive modulu počítačového vidění. Z tohoto důvodu je nezbytné analyzovat spolehlivé softwarové knihovny podporující práci s dokumenty v požadovaných formátech (viz kapitola 2). Zatímco pro většinu rastrových formátu existují již spolehlivé a odladěné knihovny (**libjpeg**, **libtiff**, **libpng**, atd.), pro formát PDF je značně omezené množství použitelných knihoven (zejména těch volně dostupných). Následující výčet obsahuje stručný popis, výhody a nevýhody potenciálně volně dostupných knihoven.

3.1 Haru

Haru (**libHaru**) je volně dostupná, multiplatformní knihovna s otevřeným zdrojovým kódem vyvinutá nezávislou komunitou vývojářů s podporou *Exploratory Software Project of Information-technology Promotion Agency, Japan* a je implementována v programovacím jazyce ANSI C. Základními funkcemi knihovny jsou:

- generování PDF souborů s textovou vrstvou,
- anotace textu a odkazů,
- bezztrátová komprese dokumentů,
- podpora vkládání PNG a JPEG obrázků,
- podpora fontů Type1, TrueScript a CJK (Type0),
- šifrování dokumentu,
- podpora znakových sad – ISO8859-1 16, MSCP1250 8 a KOI8-R.

Jak je patrné z výše uvedeného výčtu, hlavní nevýhodou je chybějící možnost editování již existujícího dokumentu. Kromě toho byla poslední stabilní verze vydána v roce 2015 ze zdrojových kódu[1] naposledy aktualizovaných v roce 2013.

[1] Zdroj `https://github.com/libharu/libharu`

3.2 Apache PDFBox

Knihovna **Apache PDFBox** je multiplatformní, volně dostupný nástroj Java pro práci s dokumenty ve formátu PDF vydaný pod licencí Apache 2.0. Jedná se o dlouhodobě vyvíjený projekt s celou řadou užitečných funkcí a vlastností, jako je:

- generování nových dokumentů a úprava již existujících dokumentů,
- extrakce textové vrstvy Unicode,
- vkládaní PNG a JPEG obrázků,
- elektronické podepsání dokumentu,
- vkládaní vlastních fontů,
- vyplňování formulářů,
- validace dokumentu podle standardu PDF/A-1b.

Funkcionalita je dostačující pro určený záměr, tj. načtení, editace a uložení dokumentu. Navíc je projekt stále udržovaný a poslední stabilní verze 2.0.19 byla vydána v lednu 2020.

3.3 PoDoFo

Poslední a nejvhodnější kandidát je multiplatformní knihovna **PoDoFo** implementovaná v programovacím jazyce C++, která je volně dostupná s otevřenými zdrojovými kódy. Knihovna je licencována jako LGPL a vyvíjena nezávislými vývojáři od roku 2006 pod vedením autora projektu Dominika Seichtera. Mezi podstatné funkce patří:

- čtení, modifikace a uložení dokumentu PDF,
- úplná syntaktická analýza dokumentu do paměti,
- šifrování dokumentu,
- podpora různých typů komprese obsahu,
- extrakce textu, obrázků a metadat dokumentu.

Obrovskou výhodou je úplná syntaktická analýza obsahu dokumentu, ačkoliv nejsou přímo připravené funkce/metody pro extrakci či vkládání obrázku z existujícího dokumentu, díky připraveným příkladům a rychle reagující podpoře se stává implementace potřebné funkcionality trivialitou. To přináší potenciálním vývojářům svobodu v modifikaci téměř jakéhokoliv objektu bez nutnosti znalosti interní struktury formátu PDF. Poslední verze 0.9.6 byla vydána v polovině roku 2018 s podporou v operačním systému Ubuntu 18.04. Dle veřejně dostupného repozitáře[2] vývoj a podpora stále pokračuje, to dokazují i poslední zveřejněné úpravy zdrojového kódu do standardu C++14.

[2] `https://sourceforge.net/projects/podofo/`

4 Nástroje počítačového vidění

Analýza potenciálně vhodných nástrojů/frameworků z oblasti počítačového vidění pro zpracování obrazu a strojové učení je poněkud přímočařejší, neboť existuje jen několik robustních a sofistikovaných řešení, které v dnešní době patří k technologickým standardům v dané oblasti. Algoritmy pro práci s obrázky jsou často z podstaty věci časově a paměťově náročné, proto je nutné věnovat pozornost i jejich optimalizovaným řešením. V následujících dílčích podkapitolách jsou detailněji popsány knihovny splňující požadované nároky.

4.1 OpenCV

OpenCV (*Open Source Computer Vision Library)* [3][33] je volně dostupná softwarová knihovna s otevřeným zdrojovým kódem vydaná pod licencí BSD, což umožňuje použití pro komerční účely a potažmo i modifikaci samotného kódu. Obsahuje přes 2500 optimalizovaných algoritmů zaměřených na tradiční techniky, ale i nejaktuálnější přístupy v doménách počítačového vidění a strojového učení. Vývoj byl oficiálně zahájen v roce 1999 a první vydání alfa verze je datováno k červnu roku 2000 z iniciativy *Intel Research Labs*[1]. Použitým programovacím jazykem je C++ kompatibilní s STL (*Standard Template Library*) a podporou i jiných často používaných jazyků jako je Java a Python. Knihovna je multiplatformní a široce používaná na platformách Linux, Mac OS, Windows a Android. Velké množství implementovaných algoritmů je připraveno k použití v tzv. real-time (reálný čas) podmínkách, např. při zpracování videa s okamžitou detekcí objektů, je dosaženo využitím dostupných procesorových instrukcí MMX (*Multi Media Extension*) a SSE (*Streaming SIMD*[2] *Extensions*), ale i grafických akcelerátorů s rozhraním kompatibilním s **CUDA**[3] (*Compute Unified Device Architecture*) a **OpenCL**[4] (*Open Computing Language*. Knihovna je rozdělena do dvou modulů – jádro a dodatečné moduly. Jádro je hlavní kostra s přísnou revizí nových algoritmů, které musejí splňovat určité nároky na kvalitu a výkon.

[1] `https://www.intel.com/content/www/us/en/research/overview.html`
[2] Single Instruction Multiple Data – vektorové instrukce CPU.
[3] Paralelní výpočty na GPU, `https://developer.nvidia.com/cuda-zone`.
[4] Knihovna pro paralelní výpočty, `https://www.khronos.org/opencl`.

Naopak dodatečné moduly obsahují algoritmy implementované širokou komunitou vývojářů a přispět může téměř kdokoliv, kvalitní a stabilní z nich jsou posléze v novějších verzích přesunuty do jádra knihovny. To přináší celému projektu obrovskou výhodu, protože právě dodatečné moduly obsahují různé modifikace algoritmů publikované v nejaktuálnějších vědeckých publikacích.

Jádro knihovny disponuje celou škálou submodulů pro práci s obrázkem nebo sérií obrázků. V následujícím výčtu jsou uvedeny převážně moduly obsahující funkce, které jsou relevantní k charakteru této práce:

- **core** – vektorové a maticové operace, struktury, makra, asynchronní rozhraní a mnoho dalšího,
- **imgproc** a **imgcodecs** – zpracování a základní operace s obrázkem, jako je filtrování, binarizace, morfologické operace, histogramy, transformace, apod.,
- **features2d** a **objdetect** – detekce objektů a klíčových bodů v obrázku,
- **ml** – algoritmy strojového učení,
- **stitching** – afinní transformace obrázků,
- **dnn** – naučené modely hlubokých neuronových sítí pro práci s obrázky,
- **photo** – techniky odšumění snímků.

Ovšem dodatečných modulů je mnohonásobně více, za zmínku stojí především:

- **ximgproc** – jedná se o rozšíření modulu **imgproc** z jádra[5],
- **xfeatures2d** a **xobjdetect** – stejně jako v předchozím bodu jde o rozšíření modulů **features2d** a **objdetect**,
- **dnn_objdetect** – detekce objektů v obrazázku pomocí hlubokých neuronových sítí,
- **dnn_superres** – naučené modely hlubokých neuronových sítě řešící problém kvality snímku při škálování,
- **shape** – techniky (metriky) pro porovnání nalezených tvarů v obrázku,

[5]x v názvu modulu znamená extended – rozšíření

- **text** – detekce a rozpoznání textu v obrázku.

Za vývojem této robustní knihovny stojí v posledních letech obrovská komunita vývojářů s rozsáhlou řadou příkladů a návodů na webu s aktivním diskuzním fórem. Díky tomu jsou nové aktualizace publikovány velice často, poslední stabilní verze 4.3.0 vyšla v dubnu 2020.

4.2 DLib

DLib [26][12] je multiplatformní knihovna primárně zaměřená na algoritmy strojové učení, ale obsahuje i ostatní nástroje pro vývoj komplexního softwaru s otevřeným zdrojovým kódem s licencí Boost Software. Využití nachází jak v akademické, tak i průmyslové sféře v řadě různých odvětí jako je robotika, embedded (vestavěné) a mobilní zařízení, a to především díky své optimalizaci. Vývoj započal v roce 2002 v čele s hlavním autorem Davisem Kingem a od té doby byl obohacen o řadu užitečných nástrojů nejen v oblasti strojového učení. Nativním programovacím jazykem je C++ s podporou jazyka Python. Stejně jako **OpenCV**, i **DLib** spoléhá na širší komunitu vývojářů, kteří dobrovolně přispívají nově implementovanými algoritmy a rozšiřují knihovnu o nové funkce. Knihovna se pyšní obšírnou a detailní dokumentací, jež přináší srozumitelné a uživatelsky přivětivé rozhraní jednoduché k použití, ale i obsáhlou sadou příkladů.

Následující přehled představuje implementované domény:

- **Bayesovské sítě** – konstrukce a evaluace Bayesovských sítí,
- **komprese** – podpora komprese a dekomprese proudu dat,
- **datové kontejnery** – speciální datové kontejnery s rozšířenou funkcionalitou (pole, fronta, zásobník, apod), které nejsou zpětně kompatibilní s C++ STL,
- **zpracování obrazu** – algoritmy pro práci s obrázkem – histogram, detekce objektů, morfologické operace, filtrace, binarizace, extrakce klíčových bodů a mnoho dalšího,
- **lineární algebra** – základní matematické, vektorové a maticové operace, ale i 2D a 3D geometrické operace (např. afinní transformace, matice rotace, atd.),
- **strojové učení** – algoritmy strojového učení pro klasifikaci, regresi, shlukovou analýzu, transformaci dat, hluboké neuronové sítě, zpětnovazební učení, skryté Markovovy modely, apod.,

- **metaprogramování** – pomocné šablonové funkce podle metodiky zvané „programovaní kontraktem“,
- **síťování** – funkce a objekty poskytující vyšší úroveň síťové abstrakce a služeb,
- **optimalizace** – algoritmy řešící problém matematické optimalizace úloh,
- **parsování** – funkce pro syntaktickou analýzu, ale i řetězcové (textové) operace,
- **grafové algoritmy** – objekty zapouzdřující grafové struktury a algoritmy, např. souvislost grafu, maximální klika grafu, atd.,
- **ostatní algoritmy** – algoritmy, které nespadají do výše uvedených domén, jako je řazení, kryptografické hashování, kvantové výpočty, atd.

Knihovna navíc obsahuje funkce pro obousměrnou konverzi mezi strukturami knihoven **DLib** a **OpenCV** . To umožňuje jednodušší integraci obou knihoven, které se tak mohou vzájemně doplňovat, jak ukazují již publikované články [29][36]. Knihovna je stále udržovaná a pravidelně aktualizovaná, poslední stabilní verze 19.19 vyšla na konci roku 2019.

5 Analýza a zpracování obrazu

Tato kapitola je zaměřena na představení používaných pojmů, technik a algoritmů pro práci s obrázkem relevantní k zadání práce, které budou následně implementovány a otestovány pro konkrétní případy. Vstupní snímek často prochází dvěma, respektive třemi fázemi, a to předzpracováním, samotným zpracováním a následným postprocessingem. Předzpracování má za úkol připravit vstupní obrázek na samotnou fázi zpracování (např. nalezení a extrakce klíčových bodů), postprocessing většinou přímo vylepšuje kvalitu vstupního obrázku na základě dvou předešlých fází. Drtivá většina poznatků zmíněných v této kapitole vychází z [41] a [2].

5.1 Barevný prostor a hloubka

Rastrový obrázek je popsán obrazovými body, tzv. „pixely“ , jako nejmenší zobrazitelný prvek obrázku. Hodnota pixelu je zakódovaná určitým počtem bitů závislým na barevné (bitové – bpp[1]) hloubce. Čím větší počet bitů je použit, tím je k dispozici širší barevná škála. Barevný prostor udává, kolik kanálů obraz obsahuje. Každá složka kanálu nese hodnotu pixelu, což se dá vyjádřit jako vektor $px = (h_1, h_2, ..., h_n)$, kde h_i je hodnota pixelu v i-tém kanálu a hodnota n je celkový počet kanálů. Běžně používané barevné (bitové) hloubky obrázku znázorňuje tabulka 5.1.

Bitová hloubka	**Počet barev**	**Označení**
1 bit	1	monochromatický obrázek
8 bitů	256	odstíny šedi (grayscale)
16 bitů	65 536	Hi-color
24 bitů	16 777 216	True Color
32 bitů	16 777 216 a alfa kanál	Super True Color

Tabulka 5.1: Přehled základních barevných hloubek obrazu.

[1]Bits Per Pixel – počet bitů na jeden pixel.

5.1.1 Binární obrázek

Binární (černobílý) obrázek obsahuje pouze bílou a černou barvu. Pixely v tomto barevném prostor jsou definovaný pouze 1 bitem (1 bpp) – 0 nebo 1 (proto název binární). Jedná se o jednoduchou a paměťově nenáročnou reprezentaci obrazu, nicméně převod z obrázku do binární podoby není úplně triviální záležitost, jak se na první pohled může zdát. Je zapotřebí zvolit/nalézt ideální prahovou hodnotu definující hranici mezi černou a bílou barvou, detailnější rozbor viz sekce 5.4.

5.1.2 Šedotónový

Jedná se obrázek ve stupních šedi (*grayscale*), kde hodnota pixelu určuje intenzitu jasu. Obrázek 5.1 znázorňuje stupnici šedi se 7 různými hodnotami od hodnoty 255 (bílá) po hodnotu 0 (černá) v 8-bitové barevné hloubce. Valná většina algoritmů počítačového vidění pracuje právě s tímto typem obrazů. Hodnota jasu může být definovaná číselnou hodnotou dle bitové hloubky nebo intenzitou danou v procentech – 0% (černá) - 100% (bílá), respektive $< 0; 1 > \in \mathbb{R}$ v plovoucí desetinné čárce.

Obrázek 5.1: Stupnice šedi.

5.1.3 RGB

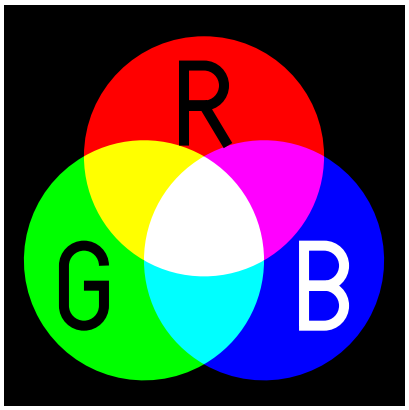
RGB je barevný prostor reprezentovaný třemi základními barevnými složkami – červená (R), zelená (G) a modrá (B). Někdy je také přidána ještě čtvrtá transparentní složka, tzv. *alfa kanál*, jehož hodnota udává průhlednost pixelu. Výsledná barva pixelu je stanovena na základě aditivního směšování jasu těchto tří, respektive čtyř barevných složek, jak je možné vidět na obrázku 5.2.

Pro převod z barevného prostoru RGB do stupňů šedi se používá jasová metoda podle následujícího vzorce:

$$px_g = 0.2126R + 0.7152G + 0.0722B, \tag{5.1}$$

kde px_g je výsledná hodnota jasu pixelu ve stupních šedi.

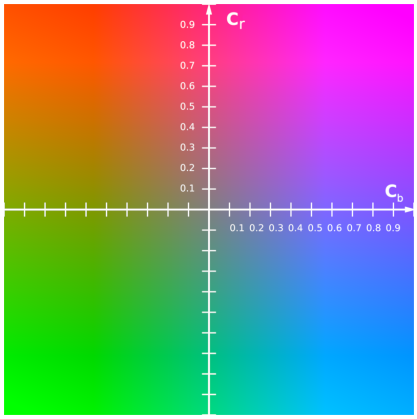
[2]Převzato z `https://en.wikipedia.org/wiki/Color_space`.



Obrázek 5.2: Aditivní míchání barev[2].

5.1.4 YC_bC_r

Barevný prostor obsahující tři složky – Y je světelná složka, C_b a C_r jsou chrominanční složky obsahující informaci o intenzitě modré a červené barvy (viz obrázek 5.3).



Obrázek 5.3: Barevný prostor YC_bC_r [3] s konstantní hodnotou Y = 0.5.

Podle [2] je definován převodní vzorec mezi RGB a YC_bC_r takto:

$$\begin{bmatrix} Y \\ C_b \\ C_r \end{bmatrix} = \begin{bmatrix} 16 \\ 128 \\ 128 \end{bmatrix} + \begin{bmatrix} 65.481 & 128.553 & 24.996 \\ -37.797 & -74.203 & 122.00 \\ 112.00 & -93.786 & -18.214 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{5.2}$$

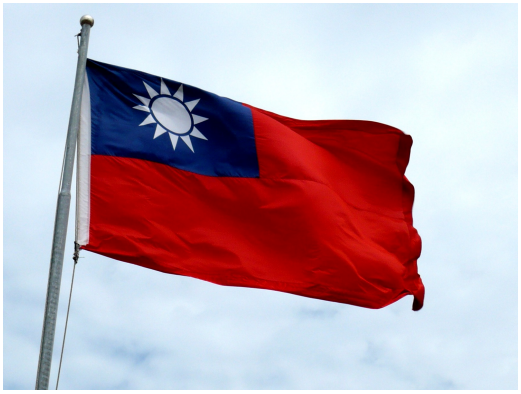
5.2 Histogram

Histogram udává distribuci intenzit pixelů v obrázku, tj. jejich absolutní četnost (zastoupení). Normalizací histogramu získáme relativní četnosti.

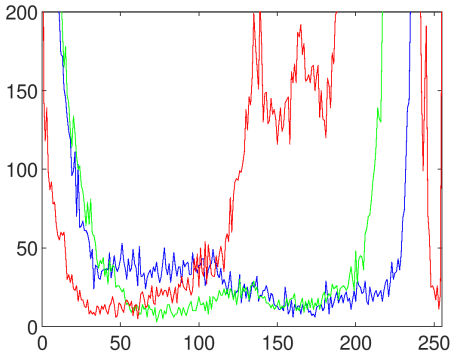
[3] Převzato z `https://en.wikipedia.org/wiki/YCbCr`.

Jedná-li se o histogram celého obrázku, pak lze říci, že poskytuje globální popis obrázku. Na základě této distribuce je možné spočítat různé statistické ukazatele, jako je minimální, maximální a průměrná intenzita pixelu, ale i rozptyl a odchylku. Důležitý je také tvar histogramu, který může být pozitivně či negativně sešikmený, ale také jeho exces (špičatost).

Podle počtu vrcholů jsou definovány typy histogramu – *unimodální* (jeden globální vrchol), *bimodální* (dva signifikantní vrcholy) a *multimodální* (více než dva signifikantní vrcholy). Histogram využívá řada algoritmů např. pro vylepšení kontrastu nebo nalezení prahové hodnoty u binarizace k detekci hran a segmentaci obrázku. Histogram se dá také využít jako jednoduchá metoda porovnání dvou obrazů. Na obr. 5.4a je příklad barevného obrázku v prostoru RGB znázorňující vlajku Tchaj-wanu s příslušným histogramem všech tří kanálů na obr. 5.4b.



(a) Obrázek vlajky Tchaj-wanu[4]v prostoru RGB.



(b) Příslušný histogram kanálů RGB.

Obrázek 5.4: Barevný obrázek a jeho příslušný histogram.

5.3 Filtrování obrázku

Proces filtrování [41] aplikuje tzv. *operátor sousedství* nebo také *lokální operátor*, který bere v potaz soubor (skupinu) pixelů v okolí zadaného referenčního pixelu k určení vlastní finální hodnoty. *Lokální* operátor se používá v přídech, kdy je potřeba vyhladit obrázek, zaostřit (zvýraznit) detaily, zdůraznit hrany nebo odstranit šum.

[4]Převzato z `https://zh.wikipedia.org/wiki/`中華民國國旗

5.3.1 Lineární filtrování

Lineární filtrační operátor patří mezi základní operátory v procesu filtrování. Výsledná hodnota pixelu využitím *lineárního filtru* je určena na základě vážených kombinací hodnot pixelů v sousedství, tj. vážené sumy hodnot v malé oblasti kolem referenčního pixelu; to je definováno takto:

$$g(i,j) = \sum_{k,l} f(i+k, j+l) \cdot h(k,l), \tag{5.3}$$

kde $h(k,l)$ představuje tzv. *jádro* či *masku* obsahující hodnoty vah často nazývané *filtrační koeficienty*. Výše uvedený vzorec (5.3) se označuje jako *korelační* operátor

$$g(i,j) = f \otimes h. \tag{5.4}$$

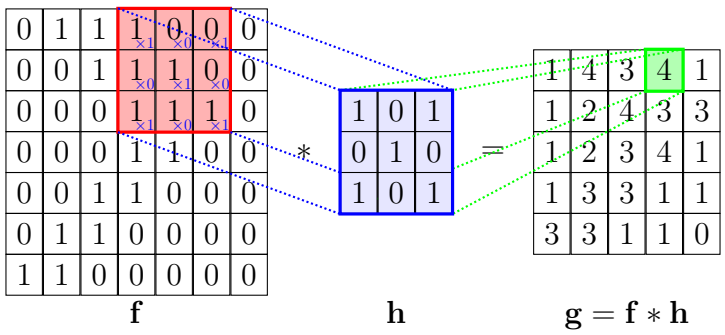
Avšak častěji používaná varianta vzorce (5.3) vypadá takto:

$$g(i,j) = \sum_{k,l} f(i-k, j-l) \cdot h(k,l) = \sum_{k,l} f(k,l) \cdot h(i-k, j-l), \tag{5.5}$$

kde se v tomto případě jedná o *konvoluční* operátor

$$g(i,j) = f * h, \tag{5.6}$$

kde h se nyní nazývá *impulzní odezva*. Obrázek 5.5 znázorňuje průběh *diskrétní konvoluce* s jádrem h o velikosti 3×3.



Obrázek 5.5: Příklad diskrétní konvoluce[5].

5.3.2 Hraniční efekt

Z obrázku 5.5 je zřetelně vidět, že výsledný obrázek g je, co se velikosti týče, menší. Jakou velikost bude mít g, závisí na velikost jádra h, jelikož

[5]Převzato z `https://github.com/PetarV-/TikZ`.

posouváním se nesmí jádro dostat přes hranice vstupního obrázku f, mimo obrázek nejsou definovány žádné hodnoty. Z toho důvodu zavedeme definici velikosti výstupního obrázku g takto:

$$\begin{aligned} w_g &= \frac{w_f - w_h + 2P}{S} + 1, \\ h_g &= \frac{h_f - h_h + 2P}{S} + 1, \end{aligned} \tag{5.7}$$

kde w_f a w_h je velikost vstupního obrazu, h_f a h_h je velikost jádra, S je velikost kroku posunu (*stride*) jádra a P je velikost tzv. „vycpávky“ (*padding*).

Díky parametru P nyní můžeme přidat extra pixely kolem vstupního obrazu a jádro se tak může dostat přes hranice f, kde již budou definovány hodnoty pixelů. Jaké hodnoty pixelů budou přidány, závisí na zvoleném typu „vycpávky“ :

- **nulový** – přidané pixely budou mít hodnotu 0,
- **konstantní** – přidané pixely budou mít zvolenou konstantní hodnotu,
- **replikace** – replikace hodnot hraničních pixelů,
- **reprodukce** – použitím *toroidu* kolem hraničních pixelů, tj. hodnoty hraničních pixelů protilehlé hrany obrázku,
- **zrcadlení** – použití hodnoty pixelu ve stejné vzdálenosti od hrany obrázku, ale v opačném směru.

5.3.3 Nelineární filtrování

Na rozdíl od *lineárního filtrování*, *nelineární filtrování* ve většině případů dosahuje lepších výsledků odšumění a vyhlazení, a to převážně díky zachování hran v obrázku, tj. zachování ostrosti hran.

Mediánový filtr

Vybere medián z okolí pixelů kolem referenčního pixelu (pomocí *lokálního* operátoru). Tento medián se stává výslednou hodnotou referenčního pixelu. *Mediánový filtr* se hodí především pro potlačení náhodného šumu. Pro výpočet je použit následující vzorec:

$$g(i,j) = \sum_{k,l} w(k,l) \cdot f(i+k, j+l), \tag{5.8}$$

kde $w(x,y)$ je jádro a $f(x,y)$ je jasová funkce vstupního obrázku.

Alternativou výpočtu je minimalizace vážené funkce:

$$\sum_{k,l} w(k,l) \cdot |f(i+k, j+l) - g(i,j)|^p, \tag{5.9}$$

kde $g(i,j)$ je žádoucí výstupní hodnota a hodnota p se nastavuje na 1 nebo 2.

Bilaterální filtr

Vypočte výslednou hodnotu pixelu na základě vážené kombinace okolních pixelů kolem referenčního pixelu:

$$g(i,j) = \frac{\sum_{k,l} f(k,l) \cdot w(i,j,k,l)}{\sum_{k,l} w(i,j,k,l)}. \tag{5.10}$$

Vážený koeficient $w(i,j,k,l)$ závisí na výsledku $d(i,j,k,l)$ a $r(i,j,k,l)$:

$$w(i,j,k,l) = \exp\left(-\underbrace{\frac{(i-k)^2 + (j-l)^2}{2\sigma_d^2}}_{\text{d(i,j,k,l)}} - \underbrace{\frac{||f(i,j) - f(k,l)||^2}{2\sigma_r^2}}_{\text{r(i,j,k,l)}}\right). \tag{5.11}$$

5.4 Prahování

Jedná se o jednoduchou segmentaci obrazu. V procesu *prahování* jde o nalezení prahové hodnoty, tj. takové intenzity jasu pixelu, která definuje hranici mezi pozadím a objektem v 8-bitovém jednokanálovém obrázku (šedotónový). Všechny pixelu obrázku jsou nastaveny v závislosti na prahové hodnotě takto:

$$g(i,j) = \begin{cases} 1, & \text{jestliže } f(i,j) \geq t \\ 0, & \text{jestliže } f(i,j) < t, \end{cases} \tag{5.12}$$

kde i a j aktuální pozice v obraze, $g(i,j)$ je intenzita výsledného pixelu, $f(i,j)$ jasová funkce výchozího obrazu a t je prahová hodnota (*threshold*).

Tomuto procesu se také jinak říká *binarizace*, jelikož rozděluje pixely v obrázku do dvou možných tříd – 0 a 1. Dále je popsáno několik používaných technik.

5.4.1 Otsu

Otsu [34] představil v roce 1979 metodu, která hledá prahovou hodnotu automaticky na základě histogramu, který je na začátku normalizován, tzn. je bezparametrická. Algoritmus se snaží zvolit takovou prahovou hodnotu,

která minimalizuje rozptyl uvnitř tříd C_0 a C_1 (pozadí a objekty) v histogramu podle vzorce:

$$\sigma_w^2(t) = \omega_0(t)\sigma_0^2(t) + \omega_1(t)\sigma_1^2(t), \tag{5.13}$$

kde t je aktuální prahová hodnota (intenzita pixelu), ω_0 a ω_1 jsou pravděpodobnosti, že daná hodnota pixelu spadá do třídy C_0 nebo C_1 s rozptylem σ_0^2, respektive σ_1^2. Jednotlivé váhy lze zapsat pomocí sumy:

$$\begin{aligned} \omega_0 = P(C_0) &= \sum_{i=0}^{t-1} p(i) = \omega_0(t), \\ \omega_1 = P(C_1) &= \sum_{i=t}^{N-1} p(i) = 1 - \omega_0 = \omega_1(t), \end{aligned} \tag{5.14}$$

kde N je velikost histogramu. Následně je možné vypočítat střední hodnoty:

$$\begin{aligned} \mu_0 &= \sum_{i=0}^{t-1} iP(i|C_0) = \sum_{i=0}^{t-1} \frac{ip(i)}{\omega_0(t)} = \mu_0(t), \\ \mu_1 &= \sum_{i=0}^{N-1} iP(i|C_1) = \sum_{i=t}^{N-1} \frac{ip(i)}{\omega_1(t)} = \mu_1(t), \\ \mu_t &= \sum_{i=0}^{N-1} ip(i), \end{aligned} \tag{5.15}$$

a zároveň platí:

$$\begin{aligned} 1 &= \omega_0 + \omega_1, \\ \mu_t &= \omega_0\mu_0 + \omega_1\mu_1. \end{aligned} \tag{5.16}$$

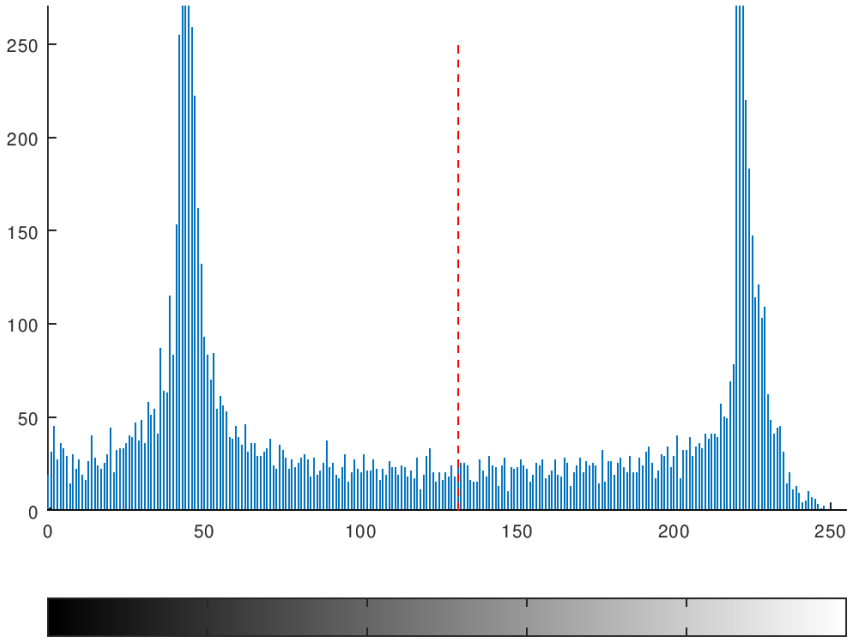
Rozptyl tříd je dán takto:

$$\begin{aligned} \sigma_0^2 &= \sum_{i=0}^{t-1} (i - \mu_0)^2 \frac{p(i)}{\omega_0(t)} = \sigma_0^2(t), \\ \sigma_1^2 &= \sum_{i=t}^{N-1} (i - \mu_1)^2 \frac{p(i)}{\omega_1(t)} = \sigma_1^2(t). \end{aligned} \tag{5.17}$$

Nyní lze dosadit do (5.13), a stále platí to, že hledáme minimální hodnotu součtu vážených středních hodnot uvnitř tříd C_0 a C_1:

$$\sigma_w^2(t^*) = \min_{0 \le t \le N-1} \sigma_w^2(t), \tag{5.18}$$

kde t^* je ideální prahová hodnota. Tato metoda přináší stabilní výsledek pro *bimodální* histogram, jenž obsahuje právě dvě významné třídy (vrcholy), jak je možné vidět na obr. 5.6.



Obrázek 5.6: Bimodální histogram s prahem podle Otsua.

5.4.2 Ekšteinova metoda

Ekštein [15] představil v roce 2016 další metodu založenou na histogramu a automatickém nalezení prahové hodnoty, tj. bez vstupních parametrů.

Před zahájením výpočtu odhadu prahové hodnoty je doporučeno redukovat nežádoucí šum ve vstupním obrázku pomocí *mediánového filtru* počítaného konvolucí:

$$f(x, y) = T_{Med}(f_{inp}(x, y), w), \tag{5.19}$$

kde $f_{inp}(x, y)$ je jasová funkce vstupního obrázku ve vstupních šedi a T_{Med} je *mediánový filtr* o velikosti $w \times w$. Dále je z obrázku získán histogram intenzit pixelů $h(i), i \in \langle 0, M \rangle$, kde pro 8-bitový obrázek ve stupních šedi platí, že $M = 255$.

Podle charakteru vstupního obrázku může histogram obsahovat extrémní hodnoty na obou stranách, tj. vysoké četnosti intenzit krajních pixelů (0 a 255), proto je žádoucí aplikovat *váhování* pomocí *Hannova okénka*:

$$h_w(i) = h(i) \cdot w_{Hann}(i), i \in \langle 0, 255 \rangle, \tag{5.20}$$

kde w_{Hann} je definováno takto:

$$w_{Hann}(n) = 0.5 \left(1 - \cos\left(\frac{2\pi n}{N-1}\right)\right), n \in \langle 0, N-1 \rangle. \tag{5.21}$$

Vážený histogram $h_w(i)$ je posléze vyhlazen *Gaussovým jádrem* pomocí konvoluce:

$$h_s(i) = (h_w * G)(i)), \tag{5.22}$$

kde *Gaussovo jádro* G je definováno následovně:

$$G(x) = \frac{1}{\sqrt{2\pi}\sigma} e^{-\frac{x^2}{2\sigma^2}}. \tag{5.23}$$

Pro histogram o velikost 256 je zvolena velikost jádra 7 s hodnotou σ nastavenou na 1.

Vyhlazený vážený histogram je numericky derivovatelný pro získání odhadu první derivace. Zderivovaný histogram $h'_s(i)$ je opět vyhlazen *Gaussovým filtrem* jako v (5.22).

Na zderivovaný histogram je aplikována funkce $g(x)$:

$$h'_g(i) = g(h'_s(i)) \tag{5.24}$$

kde funkce $g(x)$ je definována takto:

$$g(x) = \begin{cases} 0: & x < \epsilon \\ x: & x \geq \epsilon, \end{cases} \tag{5.25}$$

kde hodnota ϵ je rovna 2.5% maximální hodnoty v $h'_s(i)$.

Kandidáty pro prahovou hodnotu jsou všechny hodnoty i, které splňují podmínku $h'_g(i) = 0$. Pokud existují pouze 3 kandidáti, jedná se o *unimodální* histogram, optimálně by mělo být nalezeno alespoň 5 kandidátů, což je ve většině případů běžně dosaženo. Jeden z kandidátů vždy odpovídá globálnímu maximu v histogramu $h_s(i)$, proto je prahová hodnota nastavena na hodnotu nejbližšího souseda ve směru, kde se nachází více kandidátů (pro případ pěti a více kandidátů).

Pokud byly nalezeni pouze 3 kandidáti a prostřední z nich je roven globálnímu maximu histogramu, je nutné povést výpočet navíc, a to:

$$w_L = \int_0^a h(i)di, w_R = \int_b^M h(i)di, \tag{5.26}$$

kde mezní hodnoty integrálů jsou definovány:

$$\begin{aligned} a &= \max\{i \in \langle 0, M\rangle : h'_g(i) = 0 \wedge \forall k \in \langle 0, a\rangle : h'_g(k) = 0\}, \\ b &= \min\{i \in \langle 0, M\rangle : h'_g(i) = 0 \wedge \forall k \in \langle b, M\rangle : h'_g(k) = 0\}. \end{aligned} \tag{5.27}$$

Hodnoty a a b odpovídají nejkrajnějším kandidátům na levé, respektive pravé straně histogramu. Na základě těchto dvou hodnot je zvolena taková

prahová hodnota, která odpovídá vyšší váhové hodnotě w_L (levá strana), respektive w_R (pravá strana).

Na rozdíl od Otsuovy metody si tento přístup dokáže poradit i s histogramem, který se blíží *unimodálnímu*, tj. obě významné třídy (vrcholy) histogramu se z velké části překrývají.

5.4.3 Wolfova metoda

(Wolf et al., 2002) [44] představili vylepšení metody (Sauvola et al., 1997) [38]. Celý vstupní obrázek je procházen tzv. *posuvným okénkem* o velikosti $w \times w$. Prahová hodnota T pro středový pixel okénka je vypočtena pomocí střední hodnoty a odchylky intenzit pixelů nacházejících se v okénku.

Původní vzorec dle Sauvola pro nalezení ideální prahové hodnoty vypadá takto:

$$T(x,y) = m(x,y) \cdot \left(1 - k\left(1 - \frac{s(x,y))}{R}\right)\right), \tag{5.28}$$

kde x a y je pozice pixelu ve výchozím obrázku odpovídající středovému pixelu aktuálního okénka, $m(x,y)$ je lokální střední hodnota, $s(x,y)$ je lokální směrodatná odchylka, k je konstanta rovna -0.2 a R je taktéž konstanta s hodnotou 128 (pro obraz ve stupních šedi).

Wolfova metoda vychází z rovnice (5.28), která odstranila konstantu R následovně:

$$T(x,y) = m(x,y) - k\alpha(m - M), \tag{5.29}$$

kde α je definována takto:

$$\alpha = 1 - \frac{s}{R}, R = \max(s), \tag{5.30}$$

kde M je minimální intenzita pixelu vstupního obrazu a hodnota R je maximální hodnota ze všech možných lokálních směrodatných odchylek (ze všech okének).

Autoři uvedené metody tvrdí, že je méně náchylná na okolní šum ve skenovaných dokumentech obsahujících text než Sauvolova a Otsuova metoda. To dokazují na provedených testech, jenž dosahují vyšší přesnosti rozpoznání znaků. Na rozdíl od Ekšteinovy a Otsuovy metody je zde povinný vstupní parametr w reprezentující velikost posuvného okénka.

5.5 Morfologické operace

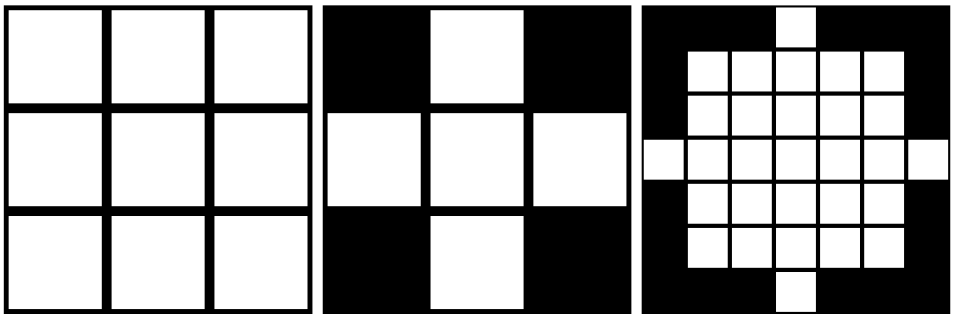
Morfologické operace [37] se používají především na binárním obrázku, tj. tyto operace jsou použity po procesu *prahování* (viz sekce 5.4). Morfologie

je naukou o tvarech. Pomocí morfologických operací jsme schopni měnit tvar zájmových objektů v obraze (popředí), tj. provést např. ztenčování, zesílení, ale i nalezení kostry, odstranění šumu a detekci hran.

Mezi základní morfologické operace patří *eroze* a *dilatace*, kombinací těchto dvou operací dostaneme *otevření*, *uzavření* a *gradient*. Všechny zmíněné operace jsou v následujících podsekcí popsány.

Uvažujeme vstupní binární obrázek jako bodovou množinu, kde hodnota pixelu 0 reprezentuje pozadí (černá barva) a 1 reprezentuje popředí, kdy zájmové objekty (bílá barva). K realizaci morfologické operace používáme tzv. *strukturní element*, který si lze představit jako binární jádro u konvoluce (viz sekce 5.3), tj. jinou (menší) bodovou množinu.

Strukturní element může být různé velikosti, ale především různého tvaru – obdélník, kříž a elipsa. Tvar určuje, jak jsou umístěny pixely hodnoty 1 v elementu, jak je možné vidět na obrázku 5.7.



(a) Obdélník. (b) Kříž. (c) Kruh.

Obrázek 5.7: Přehled tvarů strukturního elementů.

Strukturní element je ve vstupním obrázku systematicky posouván, v každém kroku posunu jsou porovnány hodnoty pixelů vstupního obrázku a elementu a podle příslušné operace je vypočtena výsledná hodnota pixelu, která se překrývá se středovým pixelem elementu, ve vstupním obrázku. Obecně není nutné se řídit podle středového pixelu elementu, ale lze nastavit libovolný pixel uvnitř elementu jako výchozí.

Následující obrázky příslušných operací s výchozím obr. 5.8 jsou převzaty z dokumentace **OpenCV**[6].

5.5.1 Eroze

Operaci *eroze* lze vyjádřit následovně:

$$I \ominus S = \{p \in E^2 : p + s \in I, \forall s \in S\}, \tag{5.31}$$

[6] `https:/docs.opencv.org/trunk/d9/d61/tutorial_py_morphological_ops.html`

Obrázek 5.8: Výchozí obrázek pro demonstraci morfologických operací.

kde I je vstupní obraz, S je *strukturní element* a E^2 je dvojrozměrný Eukleidovský prostor. Z obr. 5.9 je patrné, že se jedná o ztenčení objektu.

Obrázek 5.9: Příklad eroze.

5.5.2 Dilatace

Operaci *dilatace* lze vyjádřit následovně:

$$I \oplus S = \{p \in E^2 : p = i + s, i \in I, s \in S\}, \tag{5.32}$$

kde I je vstupní obraz, S je *strukturní element* a E^2 je dvojrozměrný Eukleidovský prostor. Z obr. 5.10 je možné vidět, že se jedná o zesílení objektu.

Obrázek 5.10: Příklad dilatace.

5.5.3 Otevření

Operace *otevření* je složena z *eroze* a následné *dilatace*:

$$I \circ S = (I \ominus S) \oplus S. \tag{5.33}$$

Z obr. 5.11 je zřetelné, že jde o odstranění šumu. V první fázi (erozi) je odstraněn šum a ztenčen objekt, ale v druhé fázi (dilataci) je objekt zpět zesílen, šum už nikoliv (žádný nezbyl).

Obrázek 5.11: Příklad otevření.

5.5.4 Uzavření

Operace *uzavření* je složena z *dilatace* a následné *eroze*:

$$I \bullet S = (I \oplus S) \ominus S. \tag{5.34}$$

Na obr. 5.12 je vidět, že *dilatace* zesílí objekt, což vede i k vyplnění děr uvnitř objektu. Následná *eroze* zpět ztenčí objekt.

Obrázek 5.12: Příklad uzavření.

5.5.5 Obrys

Operace *obrys* je složena z odečtení výsledků *dilatace* a *eroze*:

$$G(I, S) = (I \oplus S) - (I \ominus S). \tag{5.35}$$

Jak je možné vidět na obr. 5.13, operace *obrys* nám jednoduše umožňuje zachovat pouze ohraničení objektu.

Obrázek 5.13: Příklad detekce hran.

5.6 Detekce hran

Detekce hran zájmových objektů v obraze slouží k nalezení pixelů, které ohraničují objekty. Hrany v obrázků jsou oblasti, ve kterých dochází k významným změnám jasových hodnot pixelů, z toho vyplývá, že za hranu je považován i přechod mezi tmavou a světlou plochou či rozdílnou texturou.

Segmentace obrazu [41] do souvislých (ucelených) oblastí je netriviální a komplexní úloha. K detekci hran si vystačíme čistě s lokálními informacemi v obraze. Představíme-li si obraz jako výškovou mapu, pak hrany budou tvořeny strmými svahy, definovanými sklonem a směrem povrchu. Matematicky to lze vyjádřit gradientem:

$$J(x) = \nabla I(x) = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y}\right)(x) = \begin{bmatrix} g_x \\ g_y \end{bmatrix}, \tag{5.36}$$

kde ∇ je tzv. *nabla operátor*, což je diferenciální operátor a I je vstupní obrázek. Pro každý lokální gradient J je možné spočítat velikost vektoru $||J||$, což udává míru sklonu a orientaci gradientu θ, obě hodnoty lze spočítat takto:

$$\begin{aligned} ||J|| &= \sqrt{g_x^2 + g_y^2}, \\ \theta &= \arctan\left(\frac{g_y}{g_x}\right). \end{aligned} \tag{5.37}$$

Dále budou popsány konvoluční jádra sloužící k aproximaci první derivace jasové funkce (v ose x a y) a konkrétní algoritmus realizující detekci hran.

5.6.1 Robertsův operátor

Jedná se o jádro velikosti 2×2, které je definováno:

$$h_x = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, h_y = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}. \tag{5.38}$$

5.6.2 Prewittové operátor

Jedná se o jádro velikosti 3×3. Je definováno takto:

$$h_x = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}, h_y = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}. \tag{5.39}$$

Popřípadě velikosti 5×5, které je definováno takto:

$$h_x = \begin{bmatrix} 2 & 2 & 2 & 2 & 2 \\ 1 & 1 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 \\ -1 & -1 & -1 & -1 & -1 \\ -2 & -2 & -2 & -2 & -2 \end{bmatrix}, h_y = \begin{bmatrix} -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \\ -2 & -1 & 0 & 1 & 2 \end{bmatrix}. \tag{5.40}$$

5.6.3 Sobelův operátor

Jedná se o jádro velikosti 3×3, které je definováno takto:

$$h_x = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}, h_y = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix}. \tag{5.41}$$

5.6.4 Robinsonův operátor

Jedná se o jádro velikosti 3×3, definováno takto:

$$h_x = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -2 & 1 \\ -1 & -1 & -1 \end{bmatrix}, h_y = \begin{bmatrix} -1 & 1 & 1 \\ -1 & -2 & 1 \\ -1 & 1 & 1 \end{bmatrix}. \tag{5.42}$$

5.6.5 Kirschův operátor

Jedná se o jádro velikosti 3×3, definováno takto:

$$h_x = \begin{bmatrix} 3 & 3 & 3 \\ 3 & 0 & 3 \\ -5 & -5 & -5 \end{bmatrix}, h_y = \begin{bmatrix} -5 & 3 & 3 \\ -5 & 0 & 3 \\ -5 & 3 & 3 \end{bmatrix}. \tag{5.43}$$

5.6.6 Cannyho hranový detektor

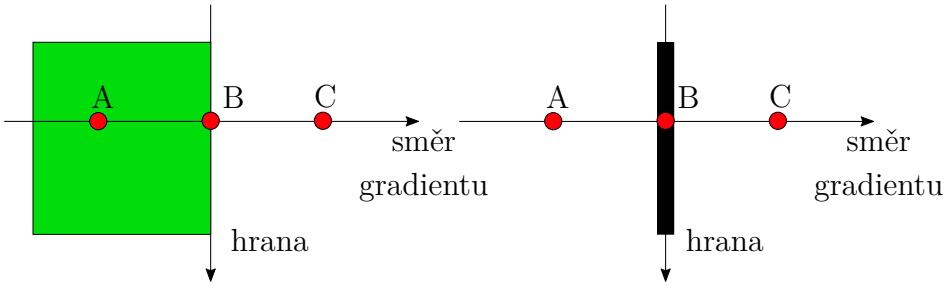
Canny [6] představil v roce 1986 algoritmus pro detekci hran, který je dodnes jeden z nejpoužívanějších (popř. jeho modifikace). Algoritmus má 4 fáze dané přesně v tomto pořadí:

1. Vyhlazení obrazu *Gaussovým* filtrem k potlačení šumu,
2. výpočet velikosti a směru gradientu podle rovnice (5.36) a (5.37) s využitím *Sobelova* operátoru (obecně méně náchylný na okolní šum),
3. ztenčení, tj. nalezení lokálních maxim,
4. prahování potenciálních hran s hysterezí.

Gaussův filtr $G(x, y)$ pro vyhlazení obrazu je definován takto:

$$G(x, y) = \frac{1}{2\pi\sigma^2} \cdot e^{\left(-\frac{x^2+y^2}{2\sigma^2}\right)}. \tag{5.44}$$

K nalezení lokálních maxim (*Non-Maximum Suppression*) je vždy zkoumána lokální oblast kolem aktuálního pixelu, aby se eliminovaly nežádoucí pixely, jenž nepředstavují okraj hrany. K tomu je zapotřebí znát již vypočtený gradient, jak znázorňuje obrázek 5.14, kde zelená plocha vlevo představuje část zájmového objektu. Pixel označený B je, v porovnání s jeho

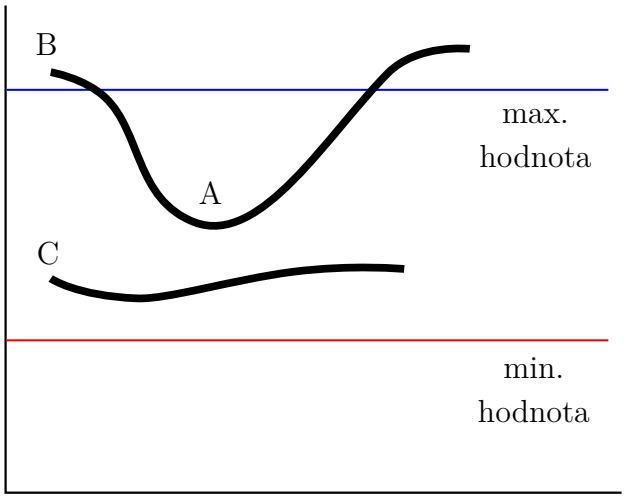


Obrázek 5.14: Příklad zeštíhlení hran.

lokálním okolím vůči pixelu A a C, lokálním maximem, a proto je ponechán jako potenciální pixel hrany. Všechny 3 pixely A, B a C leží ve směru gradientu, který je kolmý k hraně objektu.

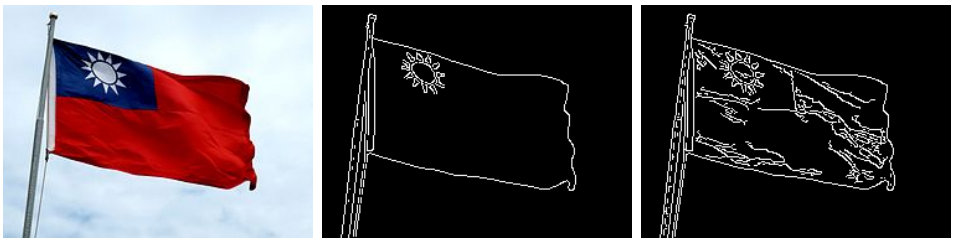
V poslední fázi je provedeno prahování s hysterezí, kde jsou definovány dvě prahovací hodnoty *max* a *min*. Pokud má pixel vyšší hodnotu jasu než *max*, pak je určitě považován za hranu a je ponechán, naopak pokud má nižší hodnotu jasu než *min*, pak se určitě o hranu nejedná. Pixel s hodnotou mezi *min* a *max* je potenciálně považován za část hrany na základě toho, zda li je spojen s pixelem, který byl již označen za hranu, tj. jeho hodnota byla vyšší než *max*.

Obr. 5.15 vystihuje uvedený popis, kde pixel B je určitě součástí hrany, kdežto A a C jsou pouze kandidáti. Na základě pixelu B je posléze i pixel A označen za součást hrany (je spojen s B a leží mezi *min* a *max*), kdežto pixel C není spojen s žádným takovým pixelem, a tudíž je zahozen.



Obrázek 5.15: Prahování pixelů s hysterezí.

Na obrázku 5.16 je možné vidět srovnání výsledků Cannyho hranového detektoru s vhodně a nevhodně zvolenými prahovými hodnotami *min* a *max*.



(a) Výchozí obrázek. (b) Vhodně zvolené parametry min a max (c) Nevhodně zvolené parametry min a max.

Obrázek 5.16: Porovnání zvolených parametrů u Cannyho algoritmu.

5.7 Odšumění

V této sekci jsou popsány sofistikovanější metody k odšumění obrazu než s využitím klasického *filtrování* (viz sekce 5.3).

5.7.1 Non-Local Means Denoising

Buades et al. [4] představili v roce 2011 metodu odšumění obrazu, která je implementována v knihovně **OpenCV**. I přesto, že se v názvu metody nachází slovo *non-local* (*nelokální*), stále se hledání podobných pixelů provádí

v lokálním (větším) okolí. Daleko přesnější by bylo pojmenování *semi-local* (*částečně lokální*), ale autoři se rozhodli zachovat původní název (jak je popisují v zmíněné publikaci).

Odšumění cílového pixelu p je dáno funkcí $u_i(p)$ takto:

$$\begin{aligned} u_i(p) &= \frac{1}{C(p)} \sum_{q \in B(p,r)} u_i(q) w(p,q), \\ C(p) &= \sum_{q \in B(p,r)} w(p,q), \end{aligned} \tag{5.45}$$

kde i je kanál obrazu (1 pro šedotónový a $< 1,3 >$ pro barevný obraz), $B(p,r)$ reprezentuje lokální okolí bodu p (centrální pixel) s velikostí $(2r+1) \times (2r+1)$ pixelů (čtvercová lokální oblast) a $C(p)$ je normalizační faktor.

Vážená funkce $w(p,q)$ závisí na druhé mocnině *eukleidovské vzdálenosti* $d^2 = d^2(B(p,f), B(q,f))$ lokálních oblastí velikosti $(2r+1) \times (2r+1)$ se středovými pixely p a q:

$$d^2(B(p,f), B(q,f)) = \frac{1}{3(2f+1)} \sum_{i=1}^{c} \sum_{j \in B(0,f)} \left(u_i(p+j) - u_i(q+j)\right)^2, \tag{5.46}$$

kde c je počet kanálů obrazu.

K finálnímu výpočtu konkrétní váhy $w(p,q)$ na základě centrálního pixelu je použito exponenciální jádro, které definováno takto:

$$w(p,q) = e^{\left(-\frac{max\left(d^2 - 2\sigma^2, 0.0\right)}{h^2}\right)}, \tag{5.47}$$

kde σ označuje standardní odchylku a h je filtrační parametr nastavený v závislosti na hodnotě σ. Oblasti s hodnotou vzdálenosti d^2 menší než $2\sigma^2$ jsou nastaveny na hodnotu váhy 1, zatímco delší vzdálenosti rychle snižují hodnotu váhy z důvodu exponenciálního průběhu funkce jádra. Váha referenčního pixelu p je nastavena na maximální hodnotu vah v lokálním okolí $B(p,r)$.

5.7.2 Stochastic Image Denoising

Estrada et al. [16] publikovali v roce 2009 *stochastické odšumění obrazu*[7]. Metoda je spojena s technikou anizotropního vyhlazení obrazu. Autoři tvrdí, že algoritmus je schopný výrazně redukovat viditelný šum v obraze při zachování detailů a ostrosti hran.

[7] stochastické = náhodné

Algoritmus je založen na tzv. *náhodných cestách* (*random walks*) kolem zvoleného lokálního okolí začínajících v aktuálně zkoumaném pixelu. Velikost a tvar lokálního okolí je dán konfigurací a podobností blízkých pixelů. Hlavní myšlenka je, že obraz je reprezentován jako graf $G(V, E)$, kde V a E je množina uzlů (tj. pixelů), respektive množina hran mezi sousedícími pixely ve vstupním obraze. Hodnota váhy či síly hrany $e(i, j) \in E$ spojující pixel i s pixelem j je úměrná jejich podobnosti. Graf G je reprezentován *Markovovou* maticí M.

Výpočet t kroků náhodné cesty v grafu G je provedeno spočtením matice M^t, což lze efektivně dosáhnout využitím vlastních vektorů matice M. Díky tomu je možné provádět simulaci kroků z jakéhokoliv zvoleného bodu x_0 a konkrétním počtem kroků.

Definice náhodně cesty z výchozího pixelu x_0 je sekvencí pixelů $T_{0,k} = \{x_0, x_1, ..., x_k\}$ navštívených na cestě z x_0 do x_k. V rámci této sekvence je pravděpodobnost přechodu mezi dvěma bezprostředně sousedícími pixely x_j a x_{j+1} definována takto:

$$p(x_{j+1}|x_j) = \frac{1}{K} \cdot \underbrace{e^{(-\frac{d(x_0,x_{j+1})^2}{2\sigma^2})}}_{t_1} \cdot \underbrace{e^{(-\frac{d(x_j,x_{j+1})^2}{2\sigma^2})}}_{t_2}, \tag{5.48}$$

kde K je normalizační koeficient, $d(x_i, x_j)$ je míra odlišnosti mezi dvěma pixely vypočtená dle *eukleidovské* vzdálenosti a σ je škálovací parametr. Složka t_1 zabraňuje rozmazání oblastí s barevným přechodem, načež složka t_2 vytváří plynulé přechody a zabraňuje cestám přejít přes silné hrany.

Na základě *Markovovy podmínky* je pravděpodobnost sekvence začínající v x_0 dána takto:

$$p(T_{1,k}|x_0) = \Pi_{j=1}^{k} p(x_j|x_{j-1}). \tag{5.49}$$

Odhad odšumění pro $\vec{I}(x_0^*)$ zkoumaného pixelu x_0 je vypočten z m náhodných cest $T_{0,k}^i, i = 1, ..., m$ začínajících v x_0. Pro každý pixel $x_j, j = 1, ..., k$ v sekvenci $T_{0,k}^i$ je vypočtena váha W_j^i na základě $p(T_{1,j}^i|x_0)$. Nicméně hodnota výsledku $p(T_{1,j}^i)$ není použita přímo, jelikož pro homogenní oblasti obrazu tato hodnota klesá příliš rychle, což vede k nežádoucím hodnotám vah. Dalším, důležitějším důvodem je, že pixel může být navštíven více náhodnými cestami s rozdílnou délkou; každopádně váha pro konkrétní pixel by měla záviset pouze na tom, zda li existuje nepřerušovaná cesta z pixelu x_0 do pixelu x_j a ne na délce cesty. Na základě toho je použit geometrický průměr pro $p(T_{1,j}^i)$, který je definován takto:

$$W_j^i = p(T_{1,j}^i)^{\frac{1}{j}}, \tag{5.50}$$

kde j reprezentuje počet kroků v sekvenci.

Nakonec je finální odhad odšumění $\vec{I}(x_0^*)$ dán takto:

$$\vec{I}(x_0^*) = \frac{1}{C}\sum_{i=1}^{m}\sum_{j=1}^{k} W_j^i I(x_j), \tag{5.51}$$

kde $C = \sum_{i=1}^{m}\sum_{j=1}^{k} W_j^i$ je normalizační konstanta a $I(x_j)$ je hodnota jasové funkce obrázku pro pixel x_j.

5.8 Škálování

Škálováním obrazu se rozumí změna velikosti původního obrazu (zmenšení či zvětšení). Vektorový obraz tvořen geometrickými primitivy (bod, přímka, obecná křivka a polygony) při tomto procesu neztrácí na kvalitě, to však neplatí pro rastrový obraz. Vzhledem k charakteru zadání jsou dále v této sekci popsané interpolační metody pro zvětšení obrazu, což se hodí u dokumentů, které byly naskenované s nízkým *ppi* (počet pixelů na palec). Výsledná kvalita rastrového obrazu závisí na zvolené interpolační metodě, také známé jako převzorkování (*Resampling*). To je dosaženo pomocí konvolučního operátoru:

$$g = f * h, \tag{5.52}$$

kde g je transformovaný obrázek, f je původní obrázek a h je jádro (maska). Rovnice 5.52 se dá taktéž rozepsat takto:

$$g(i,j) = \sum_{k,l} h(k,l) f(i - rk, j - rl), \tag{5.53}$$

kde i, j je aktuální pozice pixelu, k, l velikost jádra a r je vzorkovací hodnota.

Typ jádra konvoluce určuje různé interpolační metody, které byly podrobněji rozebrány a porovnány v [35][40]. Obecně nelze říci, která z metod je lepší či horší, primárně záleží na účelu.

5.8.1 Interpolace nejbližším sousedem

Jedná se o velice jednoduchou a výpočetně nenáročnou interpolaci, kde hodnota nového pixelu je rovna hodnotě nejbližšího známého pixelu v originálním obrázek podle:

$$g(i,j) = f(i/r, j/r), \tag{5.54}$$

kde $g(i,j)$ je hodnota výsledného pixelu a r je vzorkovací faktor > 0.

Nicméně kvalita výsledného obrazu je mizerná. Hrany v obraze jsou tzv. „rozpixelované“, jak je možné vidět v porovnání na obrázku 5.17.

(a) Originální obrázek.

(b) Třikrát zvětšený obrázek.

Obrázek 5.17: Zvětšení obrázku pomocí nejbližšího souseda.

5.8.2 Bilineární interpolace

Bilineární interpolace uvažuje jádro o velikosti 2×2, tj. 4 nejbližší sousední pixely. Na základě vzore (5.55) je vypočten výsledný pixel:

$$g(i,j) = (1-i)(1-j)f_{0,0}(I)+i(1-j)f_{1,0}(I)+(1-i)jf_{0,1}(I)+ijf_{1,1}(I), \quad (5.55)$$

kde $g(i,j)$ je výsledná hodnota pixelu, i,j odpovídají relativní vzdálenosti mezi dvěma sousedícími pixely ve výchozím obrázku a $f_{x,y}(I)$ je odpovídající hodnota pixelu na pozici x,y aktuálně sousedících pixelů z výchozího obrázku.

Na rozdíl od interpolace nejbližším sousedem v 5.8.1 u *bilineární interpolace* nedochází tolik k „rozpixelování“ obrázku zejména kolem hran, což lze vidět na obr. 5.18.

5.8.3 Bikubická interpolace

Bikubická interpolace je ideově podobná bilineární, avšak s tím rozdílem, že se používají kubické polynomy, tzv. *kubická spline*. Keys [25] představil v roce 1981 interpolaci kubickou konvolucí s následujícím jádrem:

$$h(x) = \begin{cases} 1-(a+3)^2+(a+2)|x|^3, & \text{jestliže } |x|<1 \\ a(|x|-1)(|x|-2)|^2, & \text{jestliže } 1 \leq |x| < 2 \\ 0, & \text{jinak,} \end{cases} \quad (5.56)$$

kde parametr a se většinou nastavuje na hodnotu $-0.5, -0.75$ nebo -1. Jádro je velikosti 4×4, tj. 16 pixelů celkem. K výpočtu je tentokrát nutný

(a) Originální obrázek.

(b) Třikrát zvětšený obrázek.

Obrázek 5.18: Zvětšení obrázku pomocí bilineární metody.

i pixel předcházející a následující každou dvojici sousedících pixelů, celkově tedy 4 pixely, které se podílejí na výsledném tvaru křivky.

Na obr. 5.19 lze vidět, že zvětšený obraz je více rozmazaný (hladší) než u předcházejících interpolací.

(a) Originální obrázek.

(b) Třikrát zvětšený obrázek.

Obrázek 5.19: Zvětšení obrázku pomocí bikubické metody.

5.8.4 Lanczosova interpolace

Jádro *Lanczosovy interpolace* je možné použít jak pro x složku, tak i y složku. Předpis jádra vypadá následovně:

$$h(x) = \begin{cases} 1, & \text{jestliže } x = 0, \\ \frac{a \sin(\pi x) \sin(\pi x/a)}{\pi^2 x^2}, & \text{jestliže } -a \leq x < a \text{ a zároveň } x \neq 0, \\ 0, & \text{jinak,} \end{cases} \tag{5.57}$$

kde a je velikost jádra. Pro dvourozměrnou funkci platí:

$$h(x, y) = h(x)h(y). \tag{5.58}$$

Předpis jádra v (5.58) je již možné dosadit do (5.8), detailnější popis je možné nalézt v [5].

Na obr. 5.20 je možné vidět porovnání třikrát zvětšeného obrázku pomocí *Lanczosovy interpolace*, kde hrany jsou hladké, ale zbytek obrázku zůstává ostřejší, na rozdíl od *bikubické interpolace* (viz sekce 5.8.3).

(a) Originální obrázek.

(b) Třikrát zvětšený obrázek.

Obrázek 5.20: Zvětšení obrázku pomocí Lanczosovy interpolace.

5.8.5 Super-Resolution CNN

Super-rozlišení pomocí konvolučních neuronových sítí (CNN) je nový přístup škálování obrazu s využitím hlubokého učení. Dong et al. [13] popsali v roce 2014 fungování hlubokých konvolučních neuronových sítí pro *Super-rozlišení*. V poslední době bylo publikováno několik modelů, jako je EDSR [31], ESPCN [39], FSRCNN [14] nebo LapSRN [28]. Všechny zmíněné modely jsou navíc implementovány v knihovně **OpenCV**. Každý z modelů poskytuje již natrénované neuronové sítě, které je možné ihned použít, nicméně s pevně daným škálovacím faktorem (většinou 2, 3 a 4).

Na obr. 5.21 je možné vidět porovnání třikrát zvětšeného obrázku pomocí *Super-Rozlišení* hlubokou konvoluční neuronovou sítí s modelem EDSR (podporuje barevný obraz). Oproti všem výše zmíněným interpolačním metodám dosahuje bezpochyby nejlepší kvality obrazu se zachováním určité ostrosti obrazu.

(a) Originální obrázku.

(b) Třikrát zvětšený obrázku.

Obrázek 5.21: Zvětšení obrázku pomocí modelu EDSR Super-Resolution CNN.

5.9 Detekce natočení

Při procesu skenování dokumentu formátu A4 může nastat situace, že dokument je na skenovacím zařízení mírně natočen, což se také promítne do jeho digitální podoby (snímku). Ačkoliv se nemusí jednat o okem pozorovatelné natočení, některé algoritmy počítačového vidění mohou dosahovat nepřesných či nežádoucích výsledků. Jedním takovým příkladem může být metoda *optického rozpoznání znaků* (více v sekci 6.1), která nemusí znaky (text) rozpoznat správně nebo vůbec.

Z tohoto důvodu je více než žádoucí pokusit se automaticky detekovat natočení naskenovaného dokumentu a případně provést rotaci obrázku na základě detekovaného úhlu a jeho orientaci. Dále v této sekci jsou popsány dva možné přístupy k nalezení úhlu natočení. První z nich je založen na tzv. *Houghově transformaci* (*Hough transform*) [22], druhý pak na detekci ohraničujícího rámečku kolem objektů (textu v dokumentu) a podle jejich natočení odhadnout výsledný úhel.

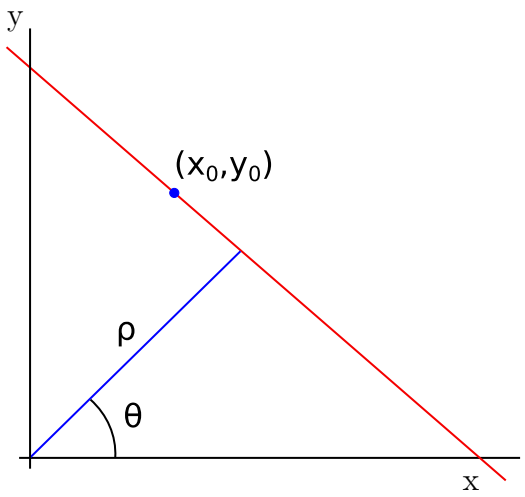
5.9.1 Houghova transformace pro detekci přímek

Houghova transformace byla úspěšně použita k detekci přímek v obrázku (*Hough Lines*), respektive její pravděpodobnostní verze (Galamhos aj., 1999) [17]. Právě na základě nalezených přímek je možné zjistit jejich úhel natočení a odvodit tak natočení celého obrázku, jak je zmíněno v [18]. Nicméně k detekci přímek v dokumentu je možné využít i řádky textu, jelikož ne každý dokument obsahuje horizontální přímky.

Předpokladem je vstupní obraz (binární) s nalezenými hranami (viz sekce 5.6), aby se snížil počet potenciálních pixelů (bodů) tvořících přímky a zachovaly se pouze ty podstatné. Definice přímky je v tomto případě v polárních souřadnicích:

$$\rho = x_0 \cos\theta + y_0 \sin\theta, \tag{5.59}$$

kde x_0 a y_0 jsou souřadnice konkrétního bodu a θ je hodnota úhlu, jak je možné vidět na obrázku 5.22.



Obrázek 5.22: Reprezentace přímky pomocí θ a ρ a konkrétních souřadnic bodu (x_0,y_0) v kartézské soustavě.

Pro každý bod (x_0, y_0) vypočteme hodnotu ρ s různým úhlem θ (vstup algoritmu). Výsledný *Houghův prostor* je dán dvojicí (θ, ρ), která tvoří tzv. *akumulátor*, kde se akumulují výskyty dvojice (θ, ρ) , tj. absolutní četnosti. Pokud bychom hodnoty θ a ρ vynesli do grafu, budou opisovat tvar *sinusoidy* (s dostatečně malým krokem inkrementace úhlu θ, viz obrázek 5.23).

Ty hodnoty četností, které jsou větší než určitá prahová hodnota (je vstupem algoritmu), jsou považovány za přímku. Každý bod (x_0, y_0) v kartézské soustavě, který odpovídá dvojici (θ, ρ) leží na stejné přímce. Vzhledem k tomu, že známe body vyskytující se na této přímce, můžeme určit i úsečku, a tudíž i její úhel natočení podle:

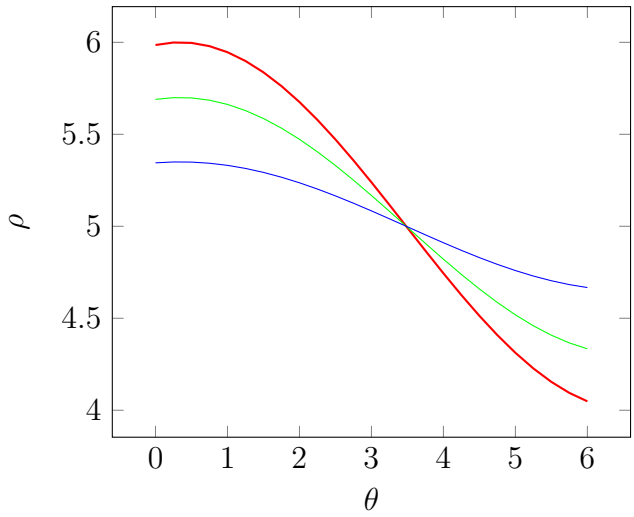
$$\alpha = \text{atan2}(y_1 - y_0, x_1 - x_0), \alpha \in (-\frac{\pi}{2}, \frac{\pi}{2}), \tag{5.60}$$

kde (x_0, y_0) a (x_1, y_1) jsou dva různé body na stejné přímce, α je úhel natočení

přímky v radiánech a $atan2(y, x)$ je definován:

$$\text{atan2}(y,x) = \begin{cases} \arctan\left(\frac{y}{x}\right), & \text{jestliže } x > 0, \\ \arctan\left(\frac{y}{x}\right) + \pi, & \text{jestliže } x < 0 \text{ a } y \geq 0, \\ \arctan\left(\frac{y}{x}\right) - \pi, & \text{jestliže } x < 0 \text{ a } y < 0, \\ +\frac{\pi}{2}, & \text{jestliže } x = 0 \text{ a } y > 0, \\ -\frac{\pi}{2}, & \text{jestliže } x = 0 \text{ a } y < 0, \\ \text{nedefinováno}, & \text{jestliže } x = 0 \text{ a } y = 0. \end{cases} \tag{5.61}$$

Na obrázku 5.23 lze vidět 3 různé sinusoidy, které byly vygenerovány na základě 3 různých bodů podle rovnice (5.59). Průsečík těchto sinusoid naznačuje, že příslušné body (x, y) a parametry (θ, ρ) tvoří příslušnou přímku.



Obrázek 5.23: Sinusoidy vygenerované pomocí 3 různých bodů.

5.9.2 Detekce natočení ohraničujících rámečků

Ohraničující rámečky (*bounding boxes*) jsou obyčejné obdélníky, které v tomto případě budou ohraničovat text či samotný znak v dokumentu (obrázku). Aby bylo možné takto objekt ohraničit, je potřeba nalézt jeho krajní pixely, a to lze dosáhnout např. pomocí *kontur*.

Kontury obsahují soubor pixelů (bodů) tvořící uzavřenou křivku kolem objektu. Kass et al. [24] představili v roce 1988 metodu *hada*, která minimalizuje energii dvourozměrné *spline křivky*[8], jež je přitahována k hranám

[8] jedná se o aproximaci křivky.

objektu. Celková energie diskrétní verze *hada* je definována takto:

$$E_{had} = \sum_{i=0}^{N-1} E_{int}(i) + E_{ext}(i), \tag{5.62}$$

kde N je celkový počet hranových pixelů, E_{int} je *interní energie* a E_{ext} je *externí energie*.

Interní energii lze zapsat takto:

$$E_{int}(i) = \alpha(i)\frac{||f(i+1) - f(i)||^2}{h^2} + \beta(i)\frac{||f(i+1) - 2f(i) + f(i-1)||^2}{h^4}, \tag{5.63}$$

kde h je hodnota kroku, $f(i)$ je jasová funkce vstupního (binárního) obrázku, $\alpha(i)$ a $\beta(i)$ jsou váhovací funkce.

Externí energii lze zapsat jako součet:

$$E_{ext}(i) = w_{line}E_{line}(i) + w_{edge}E_{edge}(i) + w_{term}E_{term}(i), \tag{5.64}$$

kde w_i reprezentují zadané váhy, E_{line} přitahuje křivku k tmavým nebo světlým částem obrázku (na základě znaménka w_{line}), E_{edge} přitahuje křivku k hranám a E_{term} je omezení, např. na základě pružinové síly.

E_{line}, E_{edge} a E_{term} jsou definovány takto:

$$\begin{aligned} E_{line}(i) &= -||G_\sigma * \nabla^2 I(f(i))||^2, \\ E_{edge}(i) &= -||\nabla I(f(i))||^2, \\ E_{term}(i) &= k_i||f(i) - d(i)||^2, \end{aligned} \tag{5.65}$$

kde G_σ je *Gaussovo* jádro se standardní odchylkou σ, k_i je koeficient tuhosti a $d(i)$ je kotvící (pevný) bod, ke kterému se kontura přibližuje.

Na základě této metody bylo publikováno mnoho modifikacích, např. [9] nebo [8].

Nakonec ohraničíme nalezené kontury obdélníkem kolem krajních pixelů, spočteme jejich úhel natočení a průměrná hodnota všech úhlů pak reprezentuje odhad celkového natočení dokumentu (obrazu).

5.10 Hledání vzoru

Hledání vzoru (*Template Matching*) [32] je technika, kde vstupem je vzor (šablona) T s šířkou w_T a výškou h_T, a obrázek I s šířkou w_I a výškou h_I. Přesněji řečeno vzor je v tomto významu malá část obrázku, kterou se snažíme nalézt ve větším obrázku, tj. lokalizovat oblast maximální podobnosti, proto musí být splněna podmínka:

$$(0 < w_T \leq w_I) \wedge (0 < h_T \leq h_I). \tag{5.66}$$

V tomto procesu není nutné projít celý vstupní obrázek, nýbrž pouze ty oblasti, pro které stále platí podmínka (5.66), tj. vzor se stále nachází uvnitř výchozího obrázku. To znamená, že výsledkem je obrázek, jehož šířka a výška je:

$$\begin{aligned} w_g &= w_I - w_T + 1, \\ h_g &= h_I - h_T + 1. \end{aligned} \tag{5.67}$$

Procházení obrazu je obecně dosaženo systematickým posunem *strukturního elementu* (jádra), kterým je přímo hledaný vzor T s výpočtem podobnosti vzoru vzhledem k podoblasti ve vstupním obrázku I:

$$g(i,j) = T \otimes I = \sum_{k}^{w_T} \sum_{l}^{h_T} T(k,l) \cdot I(i+k, j+l) \tag{5.68}$$

kde T a I je jasová funkce vzoru, respektive vstupního obrázku.

Předpokladem je, že vzor i vstupní obrázek jsou v binární podobě s invertovanými barvami (pozadí černé a popředí bílé). Dále budou popsány používané metody, které definují různou míru podobnosti hledaného vzoru. Nutno podotknout, že zmíněné metody jsou citlivé na jakoukoliv transformaci obrázku, tj. pokud hledaný vzor je ve výchozím obrázku např. v jiném velikostním poměru či natočen, pak tyto metody budou selhávat. Z charakteru zadání práce nutné takové situace řešit, jelikož vstupní dokument je vždy normalizován na velikost šablony a je provedena automatická korekce natočení obrazu (viz sekce 5.9).

5.10.1 Maskování

Jedná se o velice jednoduchou metodu, která aplikuje bitový součin (operace **AND**) mezi vzorem a podoblastí ve vstupním obraze:

$$g_i = T \wedge I_i, \tag{5.69}$$

kde i představuje i-tou podoblast ve vstupním obrázku. Jelikož popředí je definováno nenulovou hodnotou pixelu (1), pak součet nenulových hodnot v g_i reprezentuje počet shodných pixelů popředí, tj. zájmových oblastí, vstupního obrázku se vzorem:

$$n_i = \sum_{j=0}^{N-1} \left(g_i(j) \wedge 1 \right), \tag{5.70}$$

kde N je velikost podoblasti g_i, která je rovná velikosti vzoru $w_T \times h_T$.

Snahou je nalézt je maximální hodnotu n_i:

$$n_{i^*} = \max_{0 \leq i \leq N-1} n_i \tag{5.71}$$

kde hodnota i^* je pozice i-té podoblasti obsahující maximální shodu vzoru, což také udává pozici ve vstupním obrázku, kde podoblast začíná. Pokud chceme dostat procentuální shodu, stačí hodnotu n_{i*} normalizovat:

$$n_{i^*\_norm} = \frac{n_{i*}}{N_T}, \tag{5.72}$$

kde N_T je počet nenulových hodnot (1) ve vzoru.

5.10.2 Normalizovaná suma rozdílu čtverců

Metoda *normalizované sumy rozdílu čtverců* (*Normalized Sum of Square Differences*) [20] je definovaná takto:

$$g(x,y) = \frac{\sum_{x',y'}(T(x',y') - I(x+x',y+y'))^2}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}, g(x,y) \in \langle 0,1 \rangle. \tag{5.73}$$

Čím je hodnota $g(x,y)$ blíže k nule, tím vyšší je shoda vzoru s podoblastí vstupního obrázku, tj. hledáme **minimální hodnotu**.

5.10.3 Normalizovaná vzájemná korelace

Metoda *normalizované vzájemné korelace* (*Normalized Cross-Correlation*) [43] je definovaná takto:

$$g(x,y) = \frac{\sum_{x',y'}(T(x',y') \cdot I(x+x',y+y'))}{\sqrt{\sum_{x',y'} T(x',y')^2 \cdot \sum_{x',y'} I(x+x',y+y')^2}}, g(x,y) \in \langle 0,1 \rangle. \tag{5.74}$$

Čím je hodnota $g(x,y)$ blíže k 1, tím více si je vzor podobný s podoblastí výchozího obrázku, tj. hledáme **maximální hodnotu**.

5.10.4 Normalizované koeficienty vzájemné korelace

Jedná se o metodu *normalizované vzájemné korelace* s využitím *Pearsonova korelačního koeficientu* (*Normalized Cross-Correlation Coefficients*), která je definována takto:

$$g(x,y) = \frac{\sum_{x',y'}(T'(x',y') \cdot I'(x+x',y+y'))}{\sqrt{\sum_{x',y'} T'(x',y')^2 \cdot \sum_{x',y'} (I'x+x',y+y')^2}}, g(x,y) \in \langle -1,1 \rangle \tag{5.75}$$

kde T' a I' jsou definovány následovně:

$$\begin{aligned} T'(x',y') &= T(x',y') - \underbrace{\frac{1}{w_T \cdot h_T} \sum_{x'',y''} T(x'',y'')}_{\mu_{T'}}, \\ I'(x+x',y+y') &= I(x+x',y+y') - \underbrace{\frac{1}{w_I \cdot h_I} \sum_{x'',y''} I(x+x'',y+y'')}_{\mu_{I'}}, \end{aligned} \tag{5.76}$$

kde $\mu_{T'}$ je střední hodnota intenzit pixelů vzoru a $\mu_{I'}$ je střední hodnota intenzit pixelů výchozího obrázku. Střední hodnoty jsou odečtené od vzoru, respektive výchozího obrázku.

Čím je hodnota $g(x,y)$ blíže k 1, tím více si je vzor podobný s podoblastí výchozího obrázku, tj. hledáme **maximální hodnotu**.

5.10.5 Index strukturální podobnosti

Index strukturální podobnosti (*Structural Similarity Index Measurement* – SSIM) [46] udává index podobnosti dvou obrázků, v tomto případě vzoru a podoblasti I' výchozího obrázku:

$$\mathrm{SSIM}(T,I') = \underbrace{\frac{(2\mu_T\mu_{I'} + C_1)}{(\mu_T^2 + \mu_{I'}^2 + C_1)}}_{\text{jasová složka}} \cdot \underbrace{\frac{(2\sigma_{TI'} + C_2)}{(\sigma_T^2 + \sigma_{I'}^2 + C_2)}}_{\text{kontrastová složka}}, \mathrm{SSIM}(T,I') \in \langle -1, 1 \rangle, \tag{5.77}$$

kde μ_T a $\mu_{I'}$ jsou příslušné střední hodnoty, σ_T^2 a $\sigma_{I'}^2$ jsou příslušné odchylky, $\sigma_{TI'}$ je kovariance vzoru a podoblasti výchozího obrázku, hodnoty C_1 a C_2 jsou definovány takto:

$$\begin{aligned} C_1 &= (K_1 L)^2 \\ C_2 &= (K_2 L)^2, \end{aligned} \tag{5.78}$$

kde L je konstanta zvolena dle bitové hloubky obrazu $(2^{bpp} - 1)$, K_1 a K_2 jsou konstanty typicky rovné hodnotě $0,01$, respektive $0,03$.

Konstanty C_1 a C_2 jsou ve vzorci (5.77) zahrnuty proto, aby se předešlo nestabilitě právě tehdy, když $\mu_T^2 + \mu_{I'}^2 \ll 1$, respektive $\sigma_T^2 + \sigma_{I'}^2 \ll 1$.

Čím je hodnota indexu $\mathrm{SSIM}(T,I')$ blíže k 1, tím více si je vzor podobný s podoblastí výchozího obrázku, tj. hledáme **maximální hodnotu**.

6 Nástroje OCR

V této kapitole je popsána metoda *optického rozpoznání znaků* (*Optical Character Recognition* – OCR) [7] a existující systémy (knihovny) realizující OCR. Z důvodu, že existuje jen několik málo volně dostupných a robustních knihoven k realizaci OCR, bylo přihlédnuto k již provedeným analýzám, např. [11], a dále je popsán pouze nástroj **Tesseract-OCR** [42], který dle studie dosahuje nejlepších výsledků. Navíc se jedná o vyzrálý systém (vyvíjen od roku 1984) a v posledních verzi byla implementována i neuronová síť, která by měla dosahovat ještě vyšší úspěšnosti a podpory více jazyků pro širší škálu heterogenních dokumentů (dle tvrzení autorů knihovny).

6.1 Optické rozpoznání znaků

Jedná se o techniky, pomocí nichž lze docílit nalezení a rozpoznání znaků v obrázku s následnou transformací do strojově kódovaného textu, ať už z naskenovaných dokumentů či z fotografických snímků. V dnešní době je výzvou především lokalizace a rozpoznání textu z snímků reálných scén (viz obrázek 6.1), tj. různých nápisů na značkách, vozidlech, budovách, billboardech, ale i menších předmětech, apod. Další takovou výzvou je rozpoznání textu z tzv. *CAPTCHA* (*Completely Automated Public Turing Test to tell Computers and Humans Apart*) přeloženo jako „*plně automatický veřejný Turingův test k odlišení počítačů a lidí*“, což je obrázek obsahující deformovaný text s vysokým procentem nežádoucího šumu zabraňující OCR systémům rozpoznat text.

Nicméně tato práce je zaměřena na zpracování naskenovaných dokumentů, tj. jde hlavně o rozpoznání strojově tištěného textu, popř. ručně psaného. Z tohoto důvodu jsou dále popsány techniky související se zmíněným charakterem dokumentů (obrázků).

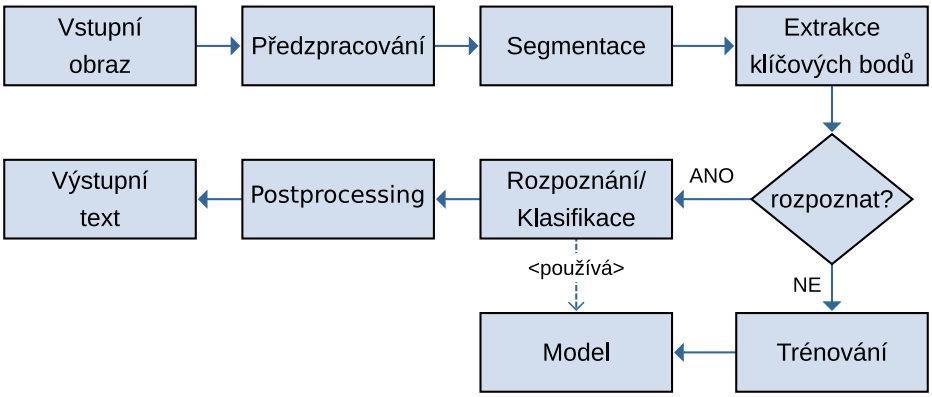
Na obrázku 6.2 je znázorněno blokové schéma s jednotlivými kroky typickými pro OCR systémy, které jsou dále blíže popsány.

6.1.1 Předzpracování

V této části se využívají techniky počítačového vidění popsané v kapitole 5, které mají za úkol vylepšit kvalitu vstupního obrázku (převážně textových

[1]Převzato z `https://cs.adelaide.edu.au/~yaoli/?page_id=111`

Obrázek 6.1: Příklady obrázku reálných scén[1].



Obrázek 6.2: Blokové schéma OCR systému.

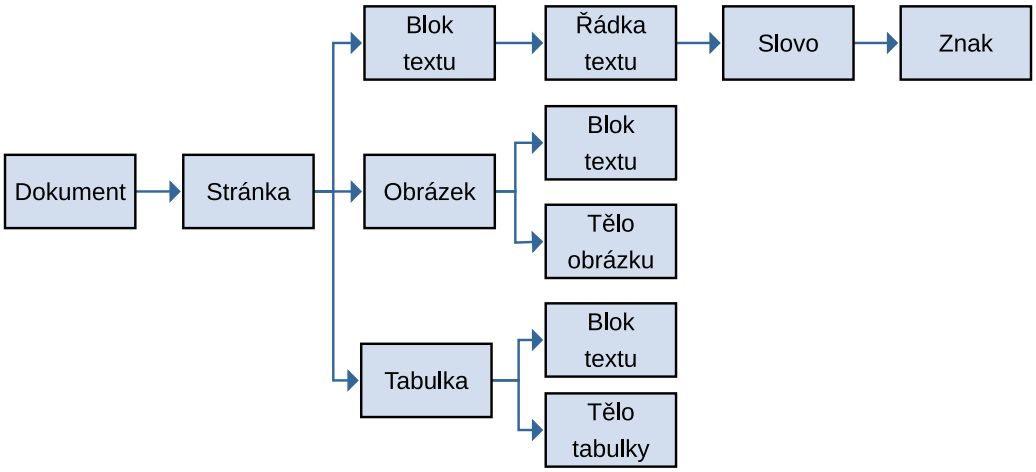
oblastí) a připravit tak obrázek na fázi segmentace s následnou extrakcí klíčových bodů. V následujícím výčtu jsou v uvedeny nezbytné techniky:

- detekce a korekce natočení,
- odstranění nežádoucího šumu,
- prahování (binarizace),
- aplikování morfologických operací – ztenčení, zesílení či vyplnění znaků,
- škálování obrazu – alespoň takové rozlišení, aby odpovídalo rozměrům naskenovaného A4 dokumentu při 300 dpi, tj. 2480×3508 pixelů (A4 z důvodu, že většina skenovaných dokumentů je v tomto formátu).

6.1.2 Segmentace

Segmentace je proces, který se snaží rozdělit obrázek do homogenních oblastí. V případě skenovaných dokumentů je záměrem detekovat bloky textu, obrázky, tabulky, sloupce a řádky. Obrázek 6.3 zobrazuje jednotlivé typy segmentace dokumentu, které lze pro textový blok seřadit takto:

- segmentace stránky,
- segmentace bloků textu,
- segmentace řádek,
- segmentace slov,
- segmentace znaků.



Obrázek 6.3: Segmentace dokumentu.

U segmentace řádek lze uvažovat i vertikálně umístěný text, tj. jedná se o sloupec, který je hojně používaný v čínském, japonském či korejském jazyce. Pokud by se v takovém případě využívaly různé slovníkové metody, slova čtená po řádcích by nemusela dávat vůbec žádný smysl, např. v čínském jazyce samotný znak (slovo) nemusí sám o sobě znamenat nic, ale v kombinaci s jiným znakem (slovem) již nabývá významu. K nalezení řádek či sloupců textu lze použít techniky jako je *vertikální a horizontální projekce* (histogramu), *Houghova transformace* nebo *nalezení kontur* (spojené komponenty).

Řádky či sloupce pak lze dělit na slova a posléze na jednotlivé znaky. K rozeznání velikosti mezer mezi slovy, respektive mezi znaky, jsou většinou použity heuristické přístupy k odhadu průměrné šířky slova, respektive

znaku. Existují také tzv. *skryté Markovovy modely*, které zastávají jak segmentaci, tak rozpoznání najednou, a mohou ihned a automaticky reagovat na změny (měnit model).

Segmentace textových bloků je v tomto případě suplovaná uživatelsky definovanými šablonami nad vstupními dokumenty, tj. uživatel si sám vybírá zájmové oblasti k zpracování (viz kapitola 2).

6.1.3 Extrakce klíčových bodů

Kumar a Bhatia [27] představili v roce 2014 přehled používaných metod k extrakci klíčových bodů reprezentující zájmové oblasti v obraze s přihlédnutím k potřebám OCR. Metody pro extrakci klíčových bodů lze rozdělit do několika skupin:

- statistické,
- globální transformace,
- geometrické a topologické,
- konvoluční neuronové sítě.

Statistické metody využívají různé statistické ukazatele k popisu znaku. Jedna z nich je metoda *zón*, která rozděluje potenciální znak do několika (ne)překrývajících se oblastí (*zón*), kde bere v potaz distribuci pixelů, tj. zaplnění těchto oblastí. Další taková metoda je založena na překryvu znaku sadou přímek s počtem přechodů z pozadí do popředí a jejich relativní vzdálenosti.

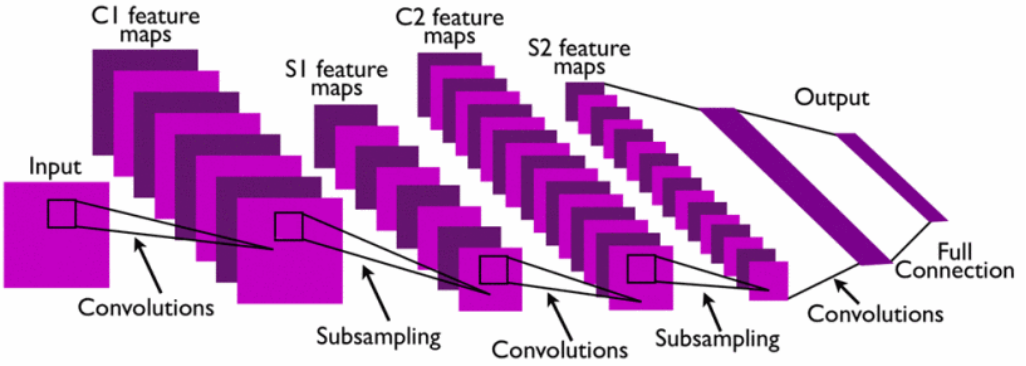
Globální transformace jsou neměnné vzhledem k různým transformacím obrázku, např. posunu či rotaci. Známé a používané metody jsou:

- Fourierova transformace,
- Houghova transformace,
- Gáborova transformace,
- vlnková transformace.

Geometrické a topologické metody se zabývají popisem tvaru znaku. Tvar znaku závisí především na použité technice při předzpracování obrázku. Běžně se používá *kontura*, *kostra* znaku nebo *hrany* znaku. *Konturu* lze např. popsat centrálními momenty. Kostru a hrany znaku můžeme popsat *řetězovým kódem* (*Freemanův kód*), jenž je konstruován na základě konkrétního

směru pohybu po hraně, kde s každým pohybem je zaznamenám příslušný symbol podle směru pohybu. Sekvence těchto symbolů tvoří *řetězový kód*.

Konvoluční neuronové sítě se samy učí nalézt vhodnou reprezentaci znaku, tj. vstupem je přímo obrázek, nikoliv jeho příznakový popis. Vstupní obrázek musí samozřejmě obsahovat popis zájmových oblastí (tzv. *labelování*). Obecně je k natrénování tohoto typu sítí potřeba velké množství trénovacích dat, nicméně LeCun aj. [30] představili v roce 2010 konvoluční síť, která je schopná se naučit i z menšího počtu trénovacích dat. Na obrázku 6.4 lze vidět architekturu takové konvoluční neuronové sítě.



Obrázek 6.4: Architektura konvoluční neuronové sítě[2].

6.1.4 Rozpoznání

Extrahované klíčové body jednotlivých znaků jsou v této fázi použity ke klasifikaci znaků s použitím natrénovaného modelu. To je docíleno buďto pomocí tradičních technik strojového učení nebo hlubokého učení (využití neuronových sítí).

Ke klasických technikám patří tyto klasifikátory:

- Bayesův klasifikátor,
- k-NN klasifikátor (k nejbližších sousedů),
- SVM (*Support Vector Machines*),
- skryté Markovovy modely (patří k jedněm ze základních technik, které byly pro OCR použity).

Mezi hluboké učení lze především zařadit:

[2]Převzato z [30].

- konvoluční neuronové sítě (CNN),
- rekurentní neuronové sítě (RNN),
- rekurentní konvoluční neuronové sítě (RCNN – kombinace RNN a CNN),
- Long Short-Term Memory (LSTM – speciální typ RNN).

6.1.5 Postprocessing

Postprocessing může omezovat výslednou množinu rozpoznaných slov pomocí různých slovníků či vzorů slova (např. pomocí regulárního výrazu), což může přispět k vyšší úspěšnosti. Slovníky obecně mohou být omezeny na specifickou množinu slov (např. technických výrazů, ekonomických, atd.), přesný výčet slov nebo pouze upřesnění daného jazyka. Výsledek nemusí být pouze čistý text, ale lze definovat i různé formáty či export do PDF s textovou vrstvou, která překrývá původní obrázek. Toto a mnohem více může být součástí výsledného zpracování, avšak záleží na konečném účelu.

6.2 Tesseract-OCR

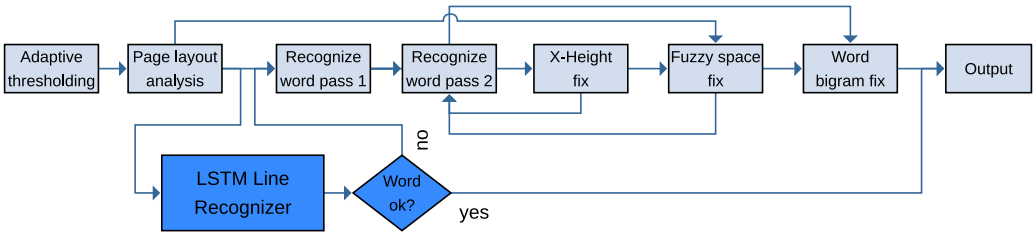
Vývoj nástroje **Tesseract-OCR** započal v roce 1984 jako postgraduální výzkumný projekt v laboratořích firmy Hewellet Packard (HP), který měl být součástí jejich nových skenovacích zařízeních. V roce 1995 byl nástroj podroben testu v rámci *Annual Test of OCR Accuracy* prováděném Nevadskou univerzitou v Las Vegas (UNLV) a překonal tehdejší komerční dostupné OCR systémy. Následně v roce 2005 byl nástroj vydán jako volně dostupný s otevřeným zdrojovým kódem pod licencí Apache verze 2 a od roku 2006 je vývoj podporován firmou Google.

Nástroj podporuje 116 světových jazyků včetně češtiny, čínštiny (tradiční a zjednodušené), arabštiny, apod. Momentálně se dá rozdělit na dvě hlavní verze, a to **3.x.x** a **4.x.x**. Zatímco verze **3.x.x** používá tradiční techniky, verze **4.x.x** je postavena na hlubokých neuronových sítí (typu LSTM).

6.2.1 Verze 4

Verze **4.x.x** zpětně podporuje i starou verzi **3.x.x**. V konfiguraci lze přesně nastavit, zda li používat kombinaci obou verzí současně, nebo pouze jednu z nich. Na rozdíl od předchozí verze, je možné rozpoznat také vertikální text,

hojně používaný převážně v čínském, japonském a korejském jazyce. Vylepšení přišlo také v oblasti rozpoznání vícejazyčných dokumentů. Na obrázku 6.5 lze vidět architekturu nástroje **Tesseract-OCR** ve verzi **4.x.x**.



Obrázek 6.5: Architektura Tesseract-OCR[3].

Samotný nástroj obsahuje metody pro segmentaci obrazu (viz podsekce 6.1.2), které lze různě konfigurovat, ale předzpracování obrazu je ponecháno na samotném vývojáři.

Neuronová síť je kompatibilní s frameworkem **TensorFlow**[4], jelikož je použit jazyk VGSL (*Variable Graph Specification Language*[5]) popisující (specifikaci) architekturu sítě. Jak již bylo zmíněno, tato verze je postavena na síti LSTM, která spadá do kategorie rekurentních neuronových sítí (RNN). Bližší popis sítě LSTM publikovali Hochreiter a Schmidhuber v roce 1997 [21]. Součástí jsou již natrénované jazykové modely, nicméně lze sítě tzv. *doučit* či *natrénovat* na svých vlastních datech.

K dispozici je spousta nastavitelných parametrů, které ovlivňují konečnou úspěšnost rozpoznaných znaků (slov). Celkový výčet lze nalézt v dokumentaci [6], každopádně mezi užitečné lze zařadit:

- **metoda segmentace** – 13 možných nastavení, jak bude nástroj segmentovat vstupní obrázek (např. obraz jako blok textu, řádka textu, samotný znak, automatická detekce, atd.),
- **load_system_dawg** – načtení hlavního **Tesseract-OCR** slovníku pro zvolený jazyk,
- **user_words_suffix** – načtení uživatelsky zvoleného slovníku,
- **language_model_penalty_non_dict_word** – penalizace slov, které nejsou v hlavním či uživatelském slovníku,

[3] Zdroj `https://github.com/tesseract-ocr/docs`

[4] `https://www.tensorflow.org/`

[5] `https://tesseract-ocr.github.io/tessdoc/VGSLSpecs`

[6] `https://github.com/tesseract-ocr/tesseract/tree/master/tessdata/configs`

- **language_model_penalty_non_freq_dict_word** – penalizace slov, které nejsou v seznamu frekventovaných slov.

7 Nástroje pro šablonovací software

Jak již bylo popsáno v sekci s požadavky (viz kapitola 2), šablonovací nástroj má sloužit k interaktivnímu navrhování uživatelsky definovatelných šablon pro konkrétní typ vstupního dokumentu. Jedná se o jakousi grafickou nadstavbu k modulu počítačového vidění a zpracování dokumentů (obrazů). Uživatel by měl být schopen nahrát dokument, který se v aplikaci vykreslí, a následně mu bude umožněno označovat významné oblasti čistě dle jeho preferencí. Nebude se jednat o označování ve smyslu přímého kreslení do obrázku, tj. jeho modifikaci, ale o vytvoření pomyslné vrstvy nad vstupním dokumentem (obrázkem). Pro každou oblast bude možnost zvolit její konkrétní typ, předdefinované akce, jméno a popis. Celá šablona bude obsahovat vlastní identifikátor, název, popis a jazyk textu v dokumentu. Aplikace musí umožnit uložení celé šablony na disk a také načtení z disku. Dále bude také disponovat funkcí nalezení nejvhodnější šablony na základně vstupního dokumentu (pokud taková již existuje).

Klasický příklad použití aplikace je následovný (uvažujme např. bankovní společnost):

1. Bankovní společenost vytvoří nový typ formuláře,
2. operátor v šablonovacím nástroji nahraje takto nový formulář,
3. operátor označí a definuje významné oblastí, většinou ty oblasti, ze kterých chce extrahovat (popř. anonymizovat) informace, např. pole pro jméno, bydliště, apod.,
4. operátor označí tzv. *synchronizační primitiva*, což jsou oblasti, které dokážou přesně identifikovat daný formulář. Jedná se o statické (neměnné) oblasti, jako je obrázek, logo, název společnosti, ale i vertikální a horizontální křivky,
5. operátor uloží šablonu, ta se exportuje do datového setu s ostatními šablonami a zkopíruje i originální formulář,
6. nyní je možné automaticky zpracovávat tyto vyplněné formuláře a provádět zvolené akce definované v konkrétních oblastech, např. pro adresu byla nastavena akce extrakce textu a anonymizování údaje, tj.

nejdřív se provede rozpoznání a extrakce adresy a následně se v dokumentu anonymizuje (odstraní).

Podle výběru programovacího jazyka C++ nebo Java, lze použít knihovnu Qt [10] nebo JavaFX [23] k realizaci GUI (*grafického uživatelského rozhraní*) aplikace.

7.1 Qt

Jedná se o volně dostupnou multiplatformní knihovnu implementovanou v programovacím jazyce C++ s otevřeným zdrojovým kódem pro vývoj aplikací s GUI. Existují i řešení pro jazyky Python, Ruby, C, Perl, Pascal, C#, Java a Haskell. Vývoj započal v roce 1991 ve společnosti Trolltech, v roce 2008 vývoj převzala firma Nokia a od roku 2014 spadá pod The Qt Company. Vývoj stále pokračuje, poslední stabilní verze 5.14 byla vydaná v prosinci 2019. Paralelně s tím je aktivní i *Qt Project* pro širší komunitu nezávislých vývojářů, kteří mohou přispět svými návrhy a implementací.

Pro snadný návrh GUI s Qt Widgets byla vytvořena aplikace **Qt Designer**, jedná se o WYSIWYG (*what you see is what you get*) editor. V editoru lze jednoduše rozvrhnout grafické komponenty a navrhovat vlastní komponenty. Umožňuje také definovat chování jednotlivých komponent pomocí *signálů a slotů* přímo v editoru. Existuje také vývojové prostředí **Qt Creator** přizpůsobený pro vývoj aplikací v nástroji Qt.

Knihovna se skládá z těchto základních modulů:

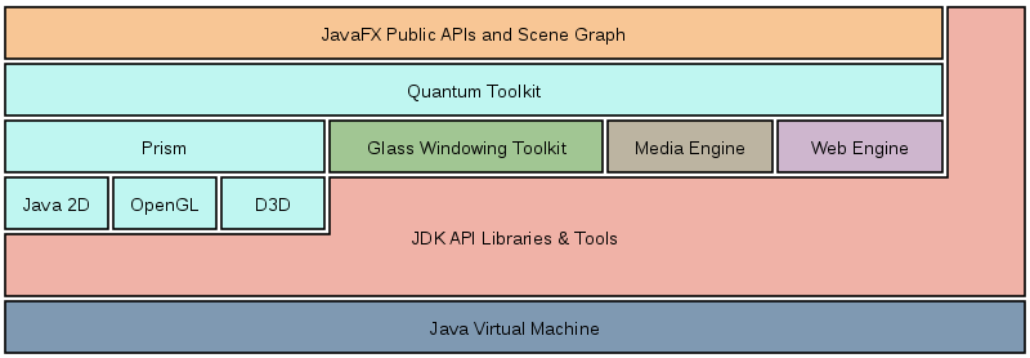
- **QtCore** – jádro obsahující implementaci datových struktur, kontejnerů, vláken, událostí, atd.,
- **QtGui** a **QtWidgets** – základní a rozšířená sada grafických prvků UI,
- **QtQuick** – vývoj aplikací v jazyce QML s podporou jazyka JavaScript a C++ s vlastním API,
- **QtNetwork** – zajišťuje síťovou komunikaci,
- **QtWebkit** – vykreslování webového obsahu v aplikaci,
- **QtSQL** – plně funkční abstraktní vrstva SQL RDBM (*Relational Database Management System*), podpora databáze typu ODBC, SQLITE, MySQL a PostrgeSQL,
- **QtMultimedia** – sada nástrojů pro zpracování multimediálního obsahu.

7.2 JavaFX

Knihovna je implementovaná v programovacím jazyce Java, byla původně vyvíjena samotnou firmou Sun Microsystems, která byla posléze odkoupena společností Oracle Corporation. Současně s vydáním Javy verze 8 v roce 2014 se knihovna JavaFX stala její nedílnou součástí; každopádně začátek vývoje je datován ještě před rokem 2008. Jedná se o náhradu zastaralého modulu Swing, což je sada nástrojů k realizaci GUI. Nicméně od verze Java 11 se o její vývoj stará firma Gluon zabývající se vývojem UI (uživatelského rozhraní) a již není součástí JDK (*Java Development Kit*).

Existuje také **JavaFX Scene Builder**, což je aplikace pro pohodlnější návrh samotného GUI (klikáním – *drag and drops*). Výsledkem není zdrojový kód v Javě, ale pouze FXML – deklarativní popis GUI pomocí XML. Samotná *logika* musí být ručně implementována a následně spárována s FXML soubory; to přináší oddělení *pohledů* od *logiky aplikace*. Podporuje také komponentu Swing, práci s dotykovými zařízeními, 2D a 3D grafiku, HTML/CSS, média (zvuk a video) a RTF (*Rich Text Format*).

Jak je možné vidět na obrázku 7.1, základem je *Scene Graph*. Organizuje a zařizuje vykreslování grafických prvků uložených ve stromové struktuře a také reaguje na uživatelské vstupy. Každý prvek je tzv. *node* (*uzel*), každý *uzel* má vlastní identifikátor, styl a ohraničující objekt. Kromě kořenového prvku (*Scene Graph*), obsahuje každý prvek rodiče a 0 či více potomků. K vykreslení je možné využít grafické knihovny DirectX 9, DirectX 10, OpenGL či Java2D. Více informací lze nalézt v dokumentaci.



Obrázek 7.1: Architektura JavaFX[1].

[1]Zdroj `https://docs.oracle.com/javafx/2/architecture/`

8 Modul počítačového vidění a zpracování dokumentů

V této kapitole je popsána implementace modulu, jenž zpracovává vstupní dokumenty a následně aplikuje techniky počítačového vidění k předzpracování dokumentu (obrázku), který je následně předán OCR systému k rozpoznání a extrakci textových oblastí. Implementace dodržuje zásady OOP (*objektově orientované programování*).

K implementaci byl vybrán programovací jazyk C++ (ve standardu C++17) s přihlédnutím k dostupným a vhodným nástrojům z analytické části. Nicméně rozhodoval i fakt, že autor práce se lépe orientuje v programovacím jazyce C++ než v Javě.

8.1 Funkce nástroje OpenCV

V této části jsou popsány základní metody, funkce a struktury knihovny **OpenCV** používané napříč celým modulem. Popsány jsou především maticové struktury k uchování vstupního obrázku a jim příslušné funkce (např. přístup k elementům matice, matematické operace, škálovaní, apod.). Funkce spojené s konkrétními algoritmy jsou popsány v rámci tříd tohoto modulu, které konkrétní algoritmus implementují.

Všechny třídy a funkce jsou umístěny v jmenném prostoru `cv`, z toho důvodu je všude před každou funkcí a třídou uveden specifikátor `cv::`. K používání knihovních struktur a funkcí je zapotřebí hlavičkový soubor:

```
#include <opencv2/opencv.hpp>
```

8.1.1 Struktura matice

Matice je implementována v třídě `cv::Mat`, kterou lze reprezentovat vstupní obrázek. Třída tak sama dynamicky spravuje paměť, aby se předcházelo časové náročnému kopírování obsahu. Obsahuje několik přetížených konstruktorů, mezi nejvíce používané v této práci patří:

```
cv::Mat(int rows, int cols, int type);
cv::Mat(int rows, int cols, int type, const cv::Scalar &s);
cv::Mat(int rows, int cols, int type, void* data, size_t step
    ↪ =0);
```

První parametrem konstruktoru je výška obrázku, až poté šířka. Je to opačně, než je u definování obrázku zvykem. To je z důvodu, že matice se obecně definují nejprve počtem řádků a pak až počtem sloupců. U druhého konstruktoru je možné využít inicializační hodnotu pro každý element matice. Třetí konstruktor dovoluje zkonstruovat matici na základě surových dat, tj. bloku paměti se specifickým krokem (velikost dimenze v ose z – počet kanálů obrázku). Každá matice má jasně definovaný datový typ její vnitřní struktury, tj. bitová hloubku a počet kanálů.

Příklady některých typů matice:

```
CV_8UC1, CV_8SC1, CV_32FC3, CV_64FC4, ...
```

`CV_` je prefix, následuje hodnota bitové hloubky zakončená znakem určující, zda li se jedná o číslo se znaménkem či bez znaménka – `S` (znaménkové) či `U` (bez znaménka), dále datový typ – `F (float)` číslo v plovoucí desetinné čárce, a zakončeno počtem kanálů – `C` jako *channels*.

Nutno zmínit, že většina parametrů funkcí a metod nepředpokládá na vstupu pouze datovou strukturu `cv::Mat`, nýbrž také tyto obecné datové struktury:

```
//Vstupní datová struktura
typedef const cv::_InputArray& InputArray;;
typedef cv::InputArray cv::InputArrayOfArrays;

//Výstupní datová struktura
typedef const cv::_OutputArray& cv::OutputArray;
typedef cv::OutputArray cv::OutputArrayOfArrays;

//Vstupní a zároveň výstupní datová struktura
typedef const cv::_InputOutputArray& cv::InputOutputArray;
typedef cv::InputOutputArray cv::InputOutputArrayOfArrays;
```

Datové struktury `InputArray`, `OutputArray` a `InputOutputArray` reprezentují nejenom datové struktury knihovny, ale i kontejnery (pole, seznamy, atd.) *STL* C++.

Přístup k prvkům matice

```
template<typename T> T &at(int row, int col);
template<typename T> const T &at(int row, int col) const;
```

Přistupovat k jednotlivým elementům matice není možné přes operátor `[]`, jak je tomu zvykem u běžných polí. Pro přístup k prvku je připravena šablonová metoda `at`, kde opět první parametr je řádek a poté sloupec.

V následujícím fragmentu kódu jsou uvedeny příklady volání metody `at`, kde parametrem šablony je konkrétní datový typ struktury matice:

```
int ele_i = (int)mat_i.at<uchar>(0,0); //Typ CV_8UC1
float ele_f = mat_f.at<float>(0,0);    //Typ CV_32FC1
double ele_d = mat_d.at<double>(0,0);  //Typ CV_64FC1
```

Pokud matice obsahuje více kanálů, pak návratovou hodnotu není primitivní datový typ, ale datová struktura `cv::Vec`:

```
cv::Vec2i vec2 = mat_i.at<cv::Vec2i>(0,0); //Typ CV_8UC2
cv::Vec3f vec3 = mat_f.at<cv::Vec3f>(0,0); //Typ CV_32FC3
cv::Vec4d vec4 = mat_d.at<cv::Vec4d>(0,0); //Typ CV_64FC4
```

Ke struktuře `cv::Vec` již lze přistupovat jako k běžnému poli, tj. je definován operátor `[]`:

```
cv::Vec3f vec = mat.at<cv::Vec3f>(0,0); //Typ CV_32FC3
float e1 = vec[0], e2 = vec[1], e3 = vec[2];
```

Předání dat matice

Pokud přiřadíme proměnnou jedné matice do proměnné jiné matice, pak datová struktura obecně předá pouze ukazatel na data. To znamená, že jakákoliv změna těchto dat se projeví v obou maticích. K vytvoření kopie slouží tyto metody:

```
cv::Mat new_mat = old_mat; //Předá ukazatel na data bez kopie
cv::Mat new_mat = old_mat.clone(); //Kopie dat
old_mat.copyTo(new_mat); // Kopie dat
```

To je dobré mít na paměti zejména tehdy, předáváme-li matici jako konstantní referenci parametru funkce či metody, což ve výsledku nezaručuje, že nemohou být modifikována data vstupní matice. Toto chování je demonstrováno v následujícím fragmentu kódu:

```
void func(const cv::Mat &m) {
  cv::Mat m2 = m; //Přiřazení matice m lokální proměnné
  m2.at<uchar>(0,0) = 123; //Přiřazení hod. lokální proměnné
}

cv::Mat mat(1,1,CV_8UC1,cv::Scalar(0)); //Matice 1x1 s hod. 0
func(mat); //Funkce modifikovala data v mat (mat[0,0] == 123)
```

Získání podmatice

```
cv::Mat operator() (const cv::Rect &roi) const;
```

Podmatice nebo také *zájmová oblast* (*Region of Interest* – ROI) lze z výchozí matice získat pomocí přetíženého operátoru `()`, kde vstupem je struktura `cv::Rect` definující obdélník (rectangle). Výstupem je struktura `cv::Mat`, ale data se opět nekopírují, tj. změna v podmatici se projeví i ve výchozí matici:

```
//Uvažujeme definovanou matici mat
cv::Rect roi = {0,0,mat.cols,1}; //Vytvoření obdélníku o šíř-
    ↪ ce matice mat a výšce 1, tj. řádka matice
cv::Mat sub_mat = mat(roi); //Získaní podmatice
cv::Mat sub_mat_c = mat(roi).clone(); //Kopie podmatice
```

Načtení a uložení obrazu

```
cv::Mat cv::imread(const std::string &filename, int flags);
```

Knihovna dovoluje načíst obrazová data přímo ze souboru, nicméně u barevných obrázků se předpokládá RGB barevný model, který ale knihovna ukládá v opačném pořadí, tj. BGR. Prvním parametrem je cesta k souboru a druhý určuje barevný model obrázku, jelikož není možné zjistit barevný model vstupního obrázku.

```
cv::Mat cv::imwrite(const std::string &filename, InputArray
    ↪ img, const std::vector<int> &params);
```

K uložení je zapotřebí cesty s konkrétním pojmenování a formátem výsledného obrázku. Pomocí parametru `params` lze předat hodnoty parametrů pro kompresi obrazu (dle výstupního obrazového formátu).

Konverze matice

```
void cv::cvtColor(cv::InputArray src, cv::OutputArray dst,
    ↪ int code, int dstCn=0);
```

Pro konverzi obrázku mezi barevnými modely lze použít výše uvedenou funkci, kde první parametr je vstupní obrázek, který chceme konvertovat, druhý je výstupní obrázek. Třetí parametr je číselný kód, který specifikuje výchozí a cílový barevný model ve formátu `cv::COLOR_in2out`, kde `in` je výchozí barevný prostor a `out` je cílové barevný prostor. Poslední parametr určuje počet cílových kanálu, pokud je roven nule, pak je počet kanálů odvozen z parametru `code`.

V následujícím úseku kódu je uveden příklad konverze barevného obrázku BGR do šedotónového:

```
//Uvažujeme definovanou matici obrazu bgr
cv::Mat gray; //Proměnná pro cílový šedotónový obraz
cv::cvtColor(bgr, gray, cv::COLOR_BGR2GRAY); //Konverze
```

Pro konverzi datového typu vnitřní struktury matice slouží metoda:

```
void cv::Mat::convertTo(OutputArray m, int rtype, double
    ↪ alpha=1, double beta=0) const;
```

Parametr `m` je cílová matice, `rtype` je cílový datový typ, `alpha` je škálovací faktor a `beta` je hodnota přičtená k výsledné matici. Příkladem může být následující fragment kódu:

```
cv::Mat mat(100,100,CV_8UC1,cv::Scalar(0)); //Typ CV_8UC1
cv::Mat mat2;
mat.convertTo(mat2,CV_32FC1); //Konverze z CV_8UC1 do
    ↪ CV_32FC1
```

Každopádně lze využít i následující konverzi obrázku z jednoho formátu do jiného (komprese obrázku):

```
bool cv::imencode(const std::string &ext, InputArray img,
    ↪ std::vector<uchar> &buff, const std::vector<int> &
    ↪ params = std::vector<int>());
```

Vstupem funkce je cílový formát `ext`, výchozí obrázek `img`, paměť cílového obrázku `buff` a seznam parametrů `params` pro konkrétní kompresi (dle parametru `ext`). Návratová hodnota udává, zda li komprese proběhla úspěšné či neúspěšně (např. nepodporovaný formát).

Následující funkce `imdecode` je opakem zmíněné funkce `imencode`:

```
cv::Mat cv::imdecode(cv::InputArray buf, int flags);
cv::Mat cv::imdecode(cv::InputArray buf, int flags, cv::Mat *
    ↪ des);
```

Díky této funkci lze načíst obrázek ze vstupní paměti `buf` a ten uložit do struktury `cv::Mat`. Parametr `flags` je stejný jako u funkce `imread`.

Matematické a logické operace

```
void cv::bitwise_and(InputArray src1, InputArray src2,
    ↪ OutputArray dst); //==  des = src1 & src2;
void cv::bitwise_or(InputArray src1, InputArray src2,
    ↪ OutputArray dst); //==  des = src1  src2;
void cv::bitwise_xor(InputArray src1, InputArray src2,
    ↪ OutputArray dst); //==  des = src1 ^ src2;
void cv::bitwise_not(InputArray src, OutputArray dst); //==
    ↪ des = !src;
```

Výše uvedené funkce zprostředkovávají logické operace and, or, xor a not. Parametry `src1` a `src2` jsou vstupní obrázky a `dst` je matice výsledku operace.

Pro matematické operace používáme klasické operátory, tak jak je zvyklostí. V následujícím fragmentu jsou popsány rozdíly použití matematických operací (příklady násobení matic):

```
//Uvažujeme definovanou matici mat a mat2
cv::Mat res = mat * 2; //Vynásobení prvků matice hodnotou 2
cv::Mat res2 = mat * mat2; //Maticové násobení
cv::Mat trans = mat.t(); //Transpozice matice
cv::Mat mat.mul(mat2); //Násobení matic po prvcích
cv::Mat inv = mat.inv(); //Inverze matice
```

Pro zobecněné násobení matic existuje funkce `gemm`:

```
void cv::gemm(cv::InputArray src1, cv::InputArray src2,
    ↪ double alpha, cv::InputArray src3, double beta,      cv
    ↪ ::OutputArray dst, int flags=0);
```

Příkladem volání uvedené funkce může být:

```
//Uvažujeme definované matice src1, src2 a src3
double alpha = 1., beta = 2.;
cv::Mat des;
cv::gemm(src1,src2, alpha, src3, beta, dst, cv::GEMM_1_T +
    ↪ cv::GEMM_3_T);
```

Výše uvedené volání funkce `gemm` odpovídá této rovnici:

$$dst = alpha \cdot src_1^T \cdot src_2 + beta \cdot src_3^T. \tag{8.1}$$

8.2 Funkce nástroje Tesseract-OCR

V této sekci jsou popsány základní funkce systému OCR – **Tesseract-OCR**, který byl vybrán na základě analýzy (viz kapitola 6). Vybrané funkce a datové struktury, jež jsou dále popsány, se vztahují k účelům rozpoznání a extrakci textu ze vstupního obrázku a jsou definovány v jmenném prostoru `tesseract`. Nástroj obsahuje širokou škálu vnitřních datových struktur pro práci jak s obrázkem, tak s textem, nicméně pro účely této práce je dostačující použití veřejného rozhraní z hlavičkových souboru:

```
#include <tesseract/baseapi.h>
#include <tesseract/ocrclass.h>
```

Nástroj využívá k vnitřní reprezentaci obrazu datovou strukturu `Pix`, která je součástí knihovny **Leptonica**[1].

8.2.1 Třída TessBaseAPI

Jedná se o základní třídu, jež implementuje rozpoznání textu z vstupního obrazu. Před samotným použitím této třídy je nutné zavolat inicializační metodu `init`:

```
int Init(const char* datapath, const char* lang, tesseract::
    ↪ OcrEngineMode mode);
```

Parametr `datapath` je cesta ke kořenovému adresáři s natrénovanými jazykovými modely, `lang` je textový řetězec definující jazyky (může obsahovat i více než 1 jazyk) a `mode` definuje, jaký typ klasifikátoru pro rozpoznávání

[1] Zdroj `http://www.leptonica.org/`

bude použit. Návratová hodnota určuje, zda li inicializace proběhla úspěšně či nastala chyba (nula značí úspěšnou inicializaci). Typy klasifikátoru jsou uvedeny v následujícím výčtovém typu `OcrEngineMode`:

```
enum OcrEngineMode {
  OEM_TESSERACT_ONLY, //Využití tradičních technik; zastaralé
  OEM_LSTM_ONLY, //Vzužití pouze LSTM(hluboké neuronové sítě)
  OEM_TESSERACT_LSTM_COMBINED, //Kombinace LSTM_ONLY a
   ↪ OEM_TESSERACT_ONLY
  OEM_DEFAULT, //Výchozí typ je OEM_TESSERACT_ONLY
};
```

Podle použitého typu klasifikátoru je nutné uvést správné cesty k natrénovaným modelům. Nástroj obsahuje několik natrénovaných modelů v repozitáři[2] dle účelu, a to:

- **tessdata** – jazykové modely pro `OEM_TESSERACT_LSTM_COMBINED` a `OEM_TESSERACT_ONLY`,
- **tessdata_fast** – jazykové modely pro `OEM_LSTM_ONLY`, který je kompromisem mezi rychlostí a přesností rozpoznání,
- **tessdata_best** – jazykové modely pro `OEM_LSTM_ONLY` dosahující vysoké přesnosti, ale pomalý v procesu rozpoznání,
- **tessdata_contrib** – neoficiální jazykové modely od širší komunity vývojářů.

Nastavení parametrů

```
//Nastavení parametrů ovlivňující rozpoznání textu
bool SetVariable(const char* name, const char* value);
//Nastavení parametrů pro ladění nástroje
bool SetDebugVariable(const char* name, const char* value);
```

Jak již bylo zmíněno v analýze nástroje **Tesseract-OCR** (viz 6.2), lze nastavit řadu parametrů, které ovlivňují výsledné rozpoznání textu z obrázku dle konkrétního účelu. Tyto parametry lze nastavit využitím výše uvedených metod, kde parametr `name` je název konkrétního parametru a `value` je příslušná hodnota, kterou chceme nastavit. Tyto metody je možné volat až po inicializaci!

Nastavení vstupního obrázku

[2]Zdroj `https://github.com/tesseract-ocr`

```
void SetImage(const unsigned char* imgdata, int w, int h, int
    ↪ bpp, int bpl);
void SetImage(Pix *pix);
```

Před samotným procesem rozpoznávání textu je nutné nastavit vstupní obrázek a k tomu slouží dvě výše uvedené metody. První metoda dokáže nastavit obrázek přímo z bloku paměti `imgdata` se zadanou šířkou `w`, výškou `h`, počtem bytů na pixel `bpp` a počtem bytů na řádku obrazu `bpl` ($bpp \times w$).

Druhá metoda nastaví vstupní obrázek z datové struktury `Pix`, ačkoliv je vstupem ukazatel, metoda provede nejdříve kopii. Vývojáři nástroje doporučují používat spíše tuto metodu, jelikož nástroj využívá tuto datovou strukturu jako vnitřní reprezentaci obrazu (dochází pouze ke kopii).

Následně je vhodné nastavit rozlišení vstupního obrázku, tj. *ppi* (počet pixelů na palec). Díky tomu je metoda schopna zvolit příslušnou velikost fontu, kterou lze pak využít k vytvoření textové vrstvy nad obrázkem. To lze nastavit metodou `SetSourceResolution`:

```
void SetSourceResolution(int ppi);
```

Hodnota *ppi* se volí na základě velikosti výstupního *plátna*, kde bude obraz s textovou vrstvou vykreslen.

Pokud chceme rozpoznat text pouze z určité oblasti vstupního obrázku, pak je možné nastavit *oblast zájmu* (obdélník) pomocí metody `SetRectangle`:

```
void SetRectangle(int left, int top, int width, int height);
```

Parametry `left` a `top` určují souřadnice horního levého rohu, kde oblast začíná, `width` a `height` je šířka, respektive výška oblasti.

Rozpoznání textu

```
int Recognize(ETEXT_DESC* monitor);
```

Potom, co je vše správně nastaveno, je možné začít s procesem rozpoznání textu z obrázku pomocí výše uvedené metody `Recognize`. Vstupem je ukazatel na datovou strukturu `ETEXT_DESC`, která umožňuje sledovat proces rozpoznání (zpětná vazba), nicméně vstupem může být i `nullptr` (proces není třeba sledovat). Výstupem je číselná hodnota určující, zda li rozpoznání proběhlo úspěšně nebo během procesu nastala chyba (nula značí úspěšné rozpoznání).

V tuto chvíli je již možné zavolat metodu `GetUTF8Text`, která vrátí nalezený text jako ukazatel na datový typ `char`:

```
virtual char* GetUTF8Text(PageIteratorLevel level) const;
```

Vstupní parametr `level` je výčtový typ `PageIteratorLevel`, který reprezentuje požadovanou úroveň textu ve struktuře, kde je uchován veškerý rozpoznaný text:

```
enum PageIteratorLevel {
  RIL_BLOCK, //Celý blok text
  RIL_PARA, //Odstavec v rámci bloku textu
  RIL_TEXTLINE, //Řádka textu v rámci odstavce
  RIL_WORD, //Slovo v rámci řádky textu
  RIL_SYMBOL //Symbol či znak v rámci slova
};
```

Díky těmto úrovním můžeme procházet rozpoznaný text v libovolném pořadí. To samozřejmě závisí na přesnosti lokalizace rozpoznaného textu a odhadu velikosti mezer mezi slovy, respektive znaky.

8.3 Implementace modulu

Modul obsahuje třídy a struktury, které jsou systematicky popsané dokumentačním komentářem přímo ve zdrojových kódech. V následujících podsekcí budou zmíněny především datové struktury, metody a funkce tvořící kostru celého modulu.

Pro práci s dokumenty typu PDF byla podle analýzy (viz kapitola 3) vybrána knihovna **PoDoFo**, která splňuje veškeré parametry na základě požadavků (viz kapitola 2). Knihovna umožňuje načíst dokument ve formátu PDF, zpětně uložit, uchovat ho v paměti, ale hlavně modifikovat vnitřní objekty.

8.3.1 Třída s konfiguracemi

Třída `ConfigLoader` se nachází v adresáři `utils` v jmenném prostoru `cfg` realizující načtení a uchování konfigurace z konfiguračního souboru `configs/modul_config.cfg`. Je implementována podle návrhového vzoru *jedináček* (*singleton*), aby nebylo možné vytvořit více instancí, tj. je pouze jediná globální konfigurace pro celý modul. Implementace využívá knihovnu `libconfig`[3], která zajišťuje parsování a logickou strukturu konfiguračního souboru.

Třída je inicializována na začátku životního cyklu aplikace v hlavní funkci `main` pomocí metody `init`, která vytvoří její instanci načtením konfiguračních parametrů do příslušných struktur. Nejsou-li uvedeny konkrétní para-

[3]Zdroj `https://hyperrealm.github.io/libconfig/`

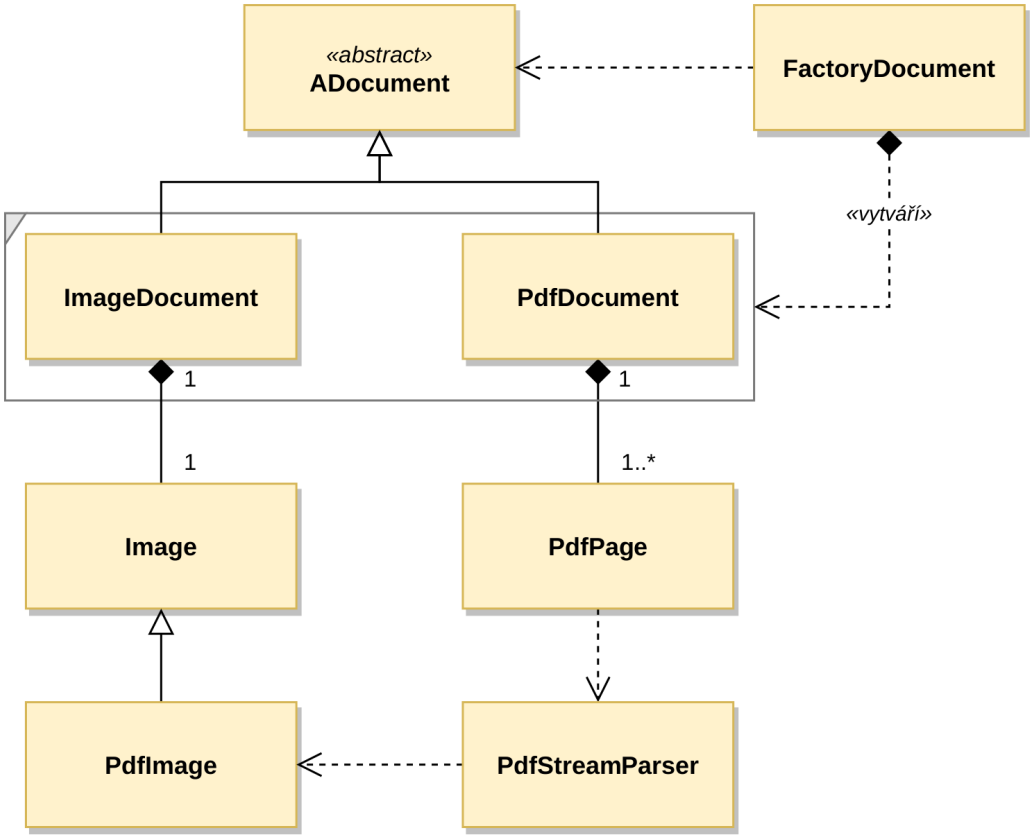
metry v konfiguračním souboru, budou použity výchozí hodnoty, které jsou definované přímo ve zdrojovém kódu příslušných struktur.

Veškeré konfigurační parametry jsou již uvedeny v konfiguračním souboru s jejich datovým typem, popisem a intervalem přípustných hodnot. Zejména je důležité dodržovat jejich datový typ, tj. nezapomenout uvozovat textové řetěze a tečkovou notaci desetinných čísel.

Každá metoda pro přístup ke struktuře je statická a veřejná, jejich konvence pojmenování je `get_`*název_struktury*`_config`.

8.3.2 Práce s dokumenty

Třídy obsažené v diagramu 8.1 jsou základními datovými strukturami pro vnitřní reprezentaci dokumentů implementující základní operace pro práci s dokumenty.



Obrázek 8.1: UML diagram tříd pro práci s dokumenty.

Třída Image

Třída `Image` je základní datová struktura pro práci se vstupním obrázkem. Všechny atributy třídy mají přístupový specifikátor `protected`, aby bylo případně možné pracovat s atributy i v odvozených třídách. Instanci této třídy lze vytvořit pomocí těchto čtyř konstruktorů:

```
explicit Image();
explicit Image(const std::string &f_name);
explicit Image(const char *bytes, size_t size);
explicit Image(const std::vector<unsigned char> &bytes);
```

Využitím prvního konstruktoru je možné načíst obrázek přímo ze souboru. Druhý a třetí konstruktor dokáže načíst obrázek z bloku paměti.

Obrázek 8.2: UML diagram třídy Image používající ImageProcessing.

Z diagramu 8.2 lze vidět, že třída `Image` využívá třídu `ImageProcessing` (algoritmy pro práci s obrazem), kde v `type_utils` jsou pomocné struktury. K interní reprezentaci obrazu je využitá datová struktura `cv::Mat` z knihovny **OpenCV**, což umožňuje přímo pracovat s jejími algoritmy bez další konverze do jiné datové struktury. Nicméně je uchován jak originální (původní) obraz, tak i modifikovaný z toho důvodu, aby bylo možné vrátit změny do původního stavu bez opětovného načítání obrázku. Pokud nejsou provedeny žádné změny v původní matici `original_mat`, pak zůstává matice `edited_mat` prázdná.

```
cv::Mat original_img, edited_mat;
```

Jelikož knihovna `OpenCV` reprezentuje obraz v barevném modelu BGR je použita implementovaná metoda `set_image`, která převede obrázek do barevného modelu RGB:

```
void Image::set_image(const cv::Mat &img) {
  if (img.channels() == 3) {
    cv::cvtColor(img, original_img, cv::COLOR_BGR2RGB);
  } else if (img.channels() == 4) {
    cv::cvtColor(img, original_img, cv::COLOR_BGRA2RGBA);
  }
}
```

Pokud se již obraz nachází v šedotónovém barevném modelu, pak není třeba žádné konverze.

Třída obsahuje metodu `anonymize_area`, která je určená k anonymizaci (skrytí) specifické oblasti v obrázku:

```
void Image::anonymize_area(const cv::Rect &area, const cv::
    ↪ Mat &mask) {
if (!has_img_edited) edited_mat = original_img.clone();
cv::Mat roi = edited_mat(area); //Získání podoblasti
//Převod do šedotónového barevného modelu
cv::Mat roi_g = ia::ImageProcessing::autoconvert_color(roi),
    ↪ bin;
ia::ImageProcessing::binarization(roi_g, bin, true);

if (!mask.empty()) { //Jedná se o oblast typu FORM
  //Získání originální obrázku formulářového typu
  cv::Mat roi_mask = ia::ImageProcessing::autoconvert_color(
    ↪ mask), mask_bin;
  ia::ImageProcessing::binarization(roi_mask, mask_bin, true)
    ↪ ;
  bin = bin ^ mask_bin;
  //Jemné odstranění šumu pomocí meidánového filtrování
  cv::medianBlur(bin, bin, 5);
}
//Pokud je oblast malá, tak zmenši počet iterací dilatace
size_t ele_it = 7;
if (area.width < 100 || area.height < 20) ele_it = 2;

// Dilatace k zesílení objektů popředí
cv::dilate(bin, bin, cv::getStructuringElement(cv::MORPH_RECT
    ↪ , {5, 5}), {-1, -1}, ele_it);
// Nahrazení popředí pozadím v původním obraze
// Textové údajé v obraze zmízí, zbyde jen pozadí
cv::inpaint(roi, bin, roi, 7, cv::INPAINT_TELEA);

has_img_edited = true;
}
```

Jak již bylo zmíněno v podsekci 8.2.1, nástroj **Tesseract-OCR** reprezentuje vnitřní obrázek v datové struktuře `Pix`, proto je v rámci této třídy implementována metoda `get_image_pix` pro převod do tohoto formátu. V rámci metody je rovnou obraz připraven pro proces OCR, zejména detekce a korekce natočení, ale také zvětšení obrázku, pomocí techniky zvolené v konfiguračním souboru (viz 5.8):

```
//Část implementace metody get_image_pix
  ...
//Detekce a korekce natočení vstupního obrazu src
ia::ImageProcessing::deskewing_scanned_document(src, src,
    ↪ config_des);
//Výpočet škálovacího faktoru (pokud je obraz malý)
```

```
size_t sc = calculate_scale();
//Převod do šedotónového barevného modelu
cv::Mat img_g = ia::ImageProcessing::autoconvert_color(source
    ↪ );
  ...
if (cfg::ConfigLoader::get_ocr_do_resizing() && (sc > 1)) {
  //Zvětšení obrazu dle škálovacího faktoru a metody z
    ↪ konfigurace
  ia::ImageProcessing::resize_image(img_grayscale, res, {-1,
    ↪ -1}, sc, sc,cfg::ConfigLoader::get_ocr_resizing_method
    ↪ ());
} else {
 //Pokud není třeba škálování, použije se šedotónový obraz
  res = img_grayscale.clone();
}
  ...
```

Třída ADocument

Základem je abstraktní třída `ADocument`, která implementuje obecné operace jak pro dokumenty typu PDF, tak i obrazové dokumenty. V následujícím fragmentu kódu jsou uvedeny nejzákladnější ryze virtuální metody:

```
virtual bool load(const std::string &doc_path)=0;
virtual bool load(const char *bytes, size_t size)=0;
virtual bool load(const std::vector<unsigned char> &bytes)=0;

virtual bool store(const std::string &doc_path)=0;
virtual bool export_pdf(const std::string &path)=0;

virtual std::shared_ptr<Image> get_image_page(size_t page)=0;

virtual void set_overlay_text_page(size_t page, const std::
    ↪ string &text)=0;
```

K načtení dokumentu lze využít 3 různé přetížené metody `load` – ze souboru (dle cesty k souboru) a z paměti. Načtení souboru z paměti je využit tehdy, je-li potřeba stáhnout dokument se vzdáleného serveru, tj. není dostupný na lokálním zařízení.

K uložení dokumentu je určená metoda `store` a `export_pdf`. Obě metody mají vstupní parametr cílovou cestu k souboru. Zatímco `store` slouží k uložení dokumentu ve formátu v jakém byl načten, metoda `export_pdf` exportuje aktuální dokument do formátu PDF a pokud je dostupná i textová vrstva, tj. bylo provedeno OCR s nastaveným parametrem `overlay`, bude ve výsledném dokumentu zahrnuta.

Získat obrázek z konkrétní stránky dokumentu (pro formát PDF) lze pomocí metody `get_image_page`, kde vstupní parametr je konkrétní stránka.

Nastavit textovou vrstvu (překrývající původní obrázek) pro konkrétní stránku je možné pomocí metody `set_overlay_text_page`, kde vstupem je číslo stránky a samotná textová vrstva. Textová vrstva není obyčejný text, nýbrž speciální formát pro PDF (viz příručka PDF [1]).

Třída PdfDocument

Třída `PdfDocument` veřejně dědí od abstraktní třídy `ADocument` a implementuje její virtuální metody. Jejím účelem je práce s naskenovanými dokumenty typu PDF, tj. načtení a extrahování obrázku z jednotlivých stránek dokumentu. Třída obsahuje jeden bezparametrický konstruktor `PdfDocument`, který zajistí vytvoření instance.

Nicméně pro načtení dokumentu je nutné zavolat jednu z tří metod `load` představených v popisu abstraktní třídy `ADocument`. Načtení dokumentu PDF ze souboru vypadá následovně:

```
bool pdf::PdfDocument::load(const std::string &doc_path) {
 try {
  //Načtení dokumentu do datové struktury PdfMemDocument
  mem_document.Load(doc_path.c_str());
  total_pages = mem_document.GetPageCount();

  //Serializace dokumentu do vyhrazeného bloku paměti
  size_t size = 0;
  auto *buff = serialize_to_bytes(size, true);
  mem_buffer = std::make_unique<PoDoFo::PdfRefCountedBuffer>(
   ↪ buff, size);
  format = "pdf";
  original_path = path = doc_path;

  LDEBUG << "created pdf::PdfDocument from file";
  return true;
 } catch (const PoDoFo::PdfError &error) {
  LERROR << "Error while loading PDF document(" << doc_path
   ↪ << "): Mem document error.";
 } catch (const pdf_excpetion::PDFException &err) {
  LERROR << "Error while loading PDF document: " << err.what
   ↪ ();
 }

 return false;
}
```

Ve výše uvedeném fragmentu kódu se načtení realizuje pomocí třídy `PoDoFo`:

:PdfMemDocument a implementované metody Load, která má vstupní parametrem cestu k dokumentu. Každopádně je ještě na řádce 9 provedena serializace dokumentu do datové struktury PdfRefCountedBuffer z toho důvodu, že je to jediná možná cesta, jak zjistit, zda li je vnitřní struktura vstupního dokumentu správně formátována podle standardu PDF [1]. V opačném případě není knihovna **PoDoFo** schopná pracovat s „poškozeným“ dokumentem, respektive s těmi objekty, které jsou nekorektně formátovány. Na základě toho lze zkusit dokument opravit externími nástroji jako je třeba PDFtk, k tomu slouží následující metoda repair_document:

```
static std::shared_ptr<PdfDocument> repair_document(const std
    ↪ ::string &fname, const std::string &fname_r, bool
    ↪ delete_file = true);
```

Parametr fname je cesta ke vstupnímu dokumentu, fname_r je cesta pro opravený dokument a delete_file určuje, zda li se má opravený dokument odstranit.

Třída extrahuje stránku až tehdy, je-li zavolána metoda get_page a tu pak udržuje v *cache* paměti, která je realizována asociativním polem (mapou) takto:

```
std::map<size_t, std::shared_ptr<pdf::Page>> pages;
```

Třída PdfPage

Třída PdfPage realizuje jednu stránku ve třídě PdfDocument vstupního dokumentu PDF. Jejím hlavním úkolem je extrahovat a uchovat obrazová data nacházející se na konkrétní stránce vstupního dokumentu. Jakmile je zavolána metoda get_image této třídy, je zahájen proces extrahování a nalezené obrázky jsou uchovány ve dynamickém poli:

```
std::vector<std::shared_ptr<PDFImage>> cached_images;
```

Nicméně každá stránka naskenovaného dokumentu obsahuje pouze jeden obrázek, který je stejné velikosti jako celá stránka, takže se momentálně nepředpokládá více než jeden obrázek na jednu stránku dokumentu.

Obsah PDF stránky je uložen v tzv. „streamu“ (proud dat) a to jak text, tak i obrazová data, která bývají běžně komprimovaná. Bohužel knihovna **PoDoFo** nedisponuje funkcemi k získání obsahu z těchto proudů dat, a tak je nutné implementovat vlastní parser (viz třída PdfStreamParser).

Předání obsahu (obrázku) z třídy PdfStreamParser je implementováno v metodě extract_information:

```
void Page::extract_information() {
  //Kontrola, zda li již byly extrahovány obrázky
```

```
  if (has_images_extracted) return;
  LDEBUG << "extracting images from page";

  //Extrakce obrázků pomocí třídy PdfSreamParser
  PdfStreamParser sp;
  cached_images = sp.extract(pdf_document->mem_document,
   ↪ pdf_page);
  has_images_extracted = true;
}
```

Třída PdfStreamParser

Jedná se o komplexní třídu, která parsuje proud dat z vnitřní struktury dokumentu PDF na základě volně dostupné příručky [1]. Dekódování proudu dat z struktury PDF do čitelného textu zajišťuje knihovna **PoDoFo** pomocí třídy `PdfContentsTokenizer`. V následujícím fragmentu pseudokódu je stručně popsán proces extrakce obrazu z stránky PDF:

```
//PageDictionary obsahuje veškerý obsah stránky
IF EXISTS "Matrix" IN PageDictionary DO
  mat := extract("Matrix") //Extrahuj transformační matici
THEN
  mat := create_identity_matrix() //Vytvoř jednotkovou matici
END IF

//Iteruj v proudu dat stránky
LOOP IN PageStream AS ps DO
  //Není-li další operátor v proudu dat, ukonči smyčku
  IF NOT ps.has_next_operator() DO BREAK LOOP END IF

  op := ps.next_operator() //Vrať nadcházející operátor

  //Operátor "q" určuje začátek grafického prvku
  IF op == "q" DO parse_graphics_information() END IF

  //Extrahuj transformační matice obrazu
  IF op == "cm" DO extract_matrix() END IF
  //Získej referenci na obraz (ID obrazu uvedeno před operá
   ↪ torem)
  IF op == "Do" DO extact_image_by_reference() END IF

  //Operátor "Q" ukončuje grafický prvek
  //Ulož informace o obraze
  IF op == "Q" DO
    save_graphics_information()
    BREAK LOOP
  END IF
```

```
END LOOP
```

Z výše uvedeného pseudokódu lze vidět, že veškeré informace o obrázku na stránce jsou uvedeny mezi závorkami `q` a `Q`. Příkladem může být následující úsek dat:

```
q
 1 0 0 1 100 200 cm
 /Im1 Do
Q
```

kde řádka 2 odpovídá takovéto transformační matici:

$$mat = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 100 & 200 & 1 \end{bmatrix}, \tag{8.2}$$

kde poslední řádek [100 200 1] matice mat udává posun v ose x o 100 jednotek a v ose y o 200 jednotek. Struktura PDF má počáteční vykreslovací bod v levém spodním rohu.

Řádka 3 předchozího pseudokódu udává identifikaci (referenci) obrazu (`Im1`), který se opět nachází ve slovníku stránky. Každý takový obrázek je definován datovou strukturou `XObject`, která odpovídá heterogennímu asociativnímu poli, kde jsou uloženy potřebné informace o obrázku. Postup získání obrazu je uveden v následujícím úseku pseudokódu:

```
//Získání "XObject" ze stránky
xobject := page.get_object("XObject")

//Kontrola existence objektu "XObject"
IF EXISTS xobject DO
  //Získání reference obrazu dle jeho id (Im1)
  img_ref := xobject.get_reference(img_id)

  IF img_ref IS "Image" DO //Kontrola zda-li jde o obrázek
    //Získání velikosti obrazu
    width := img_ref.get_value("Width")   //Šířka
    height := img_ref.get_value("Height") //Výška

    //Vytvoření obrazu pomocí třídy PdfImage
    //img_mat je transformační matice
    //document je aktuální instance třídy PdfDocument
    create_pdf_image(img_ref, img_mat, document)
  END IF
END IF
```

Na řádce 17 se vytváří instance konkrétního obrázku pomocí třídy `PdfImage`, která je posléze uložena do dynamického pole:

```
std::vector<std::shared_ptr<PDFImage>> images;
```

Toto dynamické pole `images` je na konci životnosti této třídy předáno již zmíněné třídě `PdfPage`.

Celý tento proces extrakce obrázků z proudu dat PDF stránky je realizován voláním metody `extract`, která vrací výše zmíněné pole obrázků a předává její vlastnictví třídě `PdfPage`:

```
std::vector<std::shared_ptr<PDFImage>> &&extract(PoDoFo::
    ↪ PdfDocument &document, PoDoFo::PdfPage *page);
```

Třída PdfImage

Z UML diagramu tříd (viz 8.1) je patrné, že třída `PdfImage` je potomkem třídy `Image`. Třída `PdfImage` navíc umožňuje načíst a uchovat obraz z struktury PDF. K tomu slouží jediný konstruktor:

```
explicit PDFImage(PoDoFo::PdfObject *obj, const cv::Mat &mat,
    ↪  PoDoFo::PdfDocument &doc);
```

Část implementace k získání obrazu uvnitř konstruktoru je na následujícím fragmentu kódu:

```
//Knihovna PoDoFo nepodporuje kódování CCITT
if (m_filter == "CCITTFaxDecode") {
  throw pdf_excpetion::PDFImageFormatException();
}

//Příprava paměti pro extrahovaní obrazu
PoDoFo::PdfRefCountedBuffer ref_buff;
PoDoFo::PdfBufferOutputStream img_buff_stream(&ref_buff);

//Dekomprese uloženého obrazu v PdfPage datovém proudu
//p_object je reference na obraz získaná v PdfStreamParser
dynamic_cast<PoDoFo::PdfMemStream*>(p_object->GetStream())->
    ↪ GetFilteredCopy(&img_buff_stream);

//Získání barevné hloubky obrazu
int channels = 1;
if (m_color_space == "DeviceGray") { channels = 1; }
else if (m_color_space == "DeviceRGB") { channels = 3; }
else if (m_color_space == "DeviceCMYK") { channels = 4; }
else { LINFO << "Unknown image color space - " <<
    ↪ m_color_space << "; Used default = " << channels; }

//Uložení do datové struktury cv::Mat
cv::Mat img(m_height, m_width, CV_MAKETYPE(bpp, channels), (
    ↪ void *) ref_buff.GetBuffer());
```

Obrázek je také možné uložit do datového proudu struktury PDF pomocí metody `save_image_in_pdf_stream`:

```
//Příprava paměti pro kompresi obrazu
std::vector<unsigned char> buffer;
//Komprese do JPG formátu pomocí DCT
//JPG používá diskrétní kosínovou transformaci
bool compress_ret = jpeg_compress(edited_mat, buffer);
if (!compress_ret) return false;

//Vytvoření objektu PdfImage k uložení
PoDoFo::PdfImage img_pdf(m_img_obj);
//Uložení obrazu z buffer do ss
PoDoFo::PdfMemoryInputStream ss((const char *) buffer.data(),
    ↪  (PoDoFo::pdf_long) buffer.size());
//Předání dat obrazu
img_pdf.SetImageDataRaw(m_width, m_height, bpp, &ss);
//Přidání informace o kompresi obrazu do PDF struktury
m_img_obj->GetDictionary().AddKey("Filter", PoDoFo::PdfName("
    ↪ DCTDecode"));
```

Třída ImageDocument

Jedná se o potomka abstraktní třídy `ADocument`, který se stará o načtení a uchování dokumentů čistě obrazového charakteru. Třída vždy reprezentuje jen jeden obrázek dokumentu. De facto se jedná o *obalovou* třídu pro `Image`, která navíc realizuje export do dokumentu PDF a uložení textové vrstvy na základě výsledku procesu OCR.

Export do formátu PDF je implementován v metodě `export_pdf` (realizace virtuální metody z abstraktní třídy `ADocument`). V principu jde o stejný proces jako v třídě `PdfDocument`, pouze je nutné vytvořit zcela nový dokument. Odlišnost implementace ukazuje následující úsek programového kódu:

```
//Část implementace metody export_pdf

//Vytvoření PDF dokumentu a stránky v A4 formátu
PdfStreamedDocument pdf_doc(path.c_str());
PdfPage *page = pdf_doc.CreatePage(PdfPage::
    ↪ CreateStandardPageSize(ePdfPageSize_A4, false));
  ...
  ...
//Uložení obrazu ve formátu PDF struktury
std::stringstream ss;
ss.precision(15L);
ss << "q\n" //Začátek grafického objektu
```

```
    //Formát transformační matice
    << (sc * xo->GetPageSize().GetWidth()) << " 0 0 "
    << (sc * xo->GetPageSize().GetHeight()) << " "
    << x << " "
    << y << " cm\n"
    //Uložení identifikátoru obrazu
    << "/" << xo->GetIdentifier().GetName() << " Do\n"
    << "Q\n"; //Ukončení grafického objektu

text_in_image = m_oss.str() + "\n\n" + text_in_image;

//Vytvoření textové vrstvy v dokumentu a uložení obrazu
//O textovou vrstvu se stará třída tesseract_renderer
PdfReference ref;
ocr::tesseract_renderer::append_text_layer(text_in_image,
    ↪ page, pdf_doc, ref, false);
```

Třída FactoryDocument

Jedná se o třídu vytvářející instance konkretní implementace datové struktury `ADocument` podle návrhového vzoru *továrna* (*factory*). Mimo jiné definuje seznam podporovaných obrazových formátů na základě použitých knihoven:

```
{
"bmp","dib","jpeg","jpg","jpe","jp2","png","pbm","pgm","ppm",
    ↪ "pxm","pnm","sr","ras","tiff", "tif","exr","hdr","pic"
};
```

Třída obsahuje čtyři veřejné statické metody k vytvoření dokumentu:

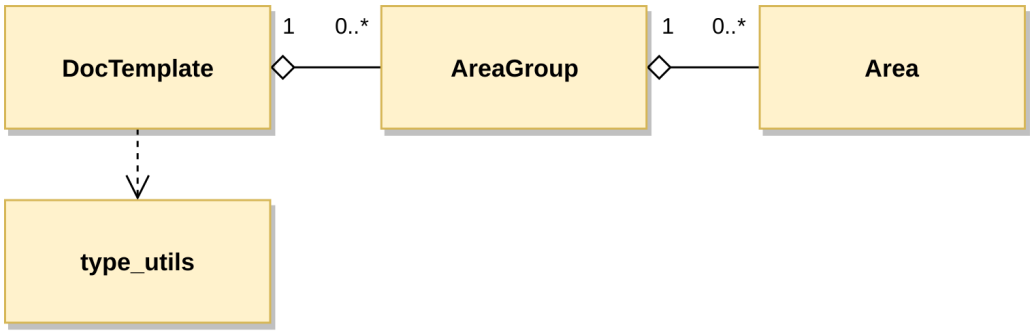
```
//Vytvoření z cesty k dokumentu (odvození formátu)
static std::shared_ptr<ADocument> create_document(const std::
    ↪ string &path);

//Vytvoření z cesty k dokumentu, typu a formátu
static std::shared_ptr<ADocument> create_document(const std::
    ↪ string &path, DocumentType type, const std::string &
    ↪ format);

//Vytvoření dokumentu z bloku paměti, typu a formátu
static std::shared_ptr<ADocument> create_document(const char
    ↪ *bytes, size_t size, DocumentType type, const std::
    ↪ string &format);
static std::shared_ptr<ADocument> create_document(const std::
    ↪ vector<unsigned char> &bytes, DocumentType type, const
    ↪ std::string &format);
```

8.3.3 Struktura šablony dokumentu

Struktura šablony (viz kapitola 2) je realizována v třídě `DocTemplate`, která je zcela nezávislá na třídě `ADocument`. Třída uchovává uživatelsky definované šablony s využitím dalších dvou pomocných tříd – `Area` a `AreaGroup`, jak je možné vidět na diagramu 8.3. Pro práci s formátem JSON je použita knihovna **jsoncpp**[4] a pro generování unikátního identifikátoru knihovna **uuid**[5].



Obrázek 8.3: UML diagram třídy DocTemplate používající AreaGroup a Area.

Třída Area

Třída `Area` je datová struktura, která reprezentuje vytvořenou zájmovou oblast v šabloně:

```
struct area_t {
  bool read_only; //Pro účel GUI
  //Unikátní ID, název oblasti a její popis
  std::string uuid, name, description;
  //Type oblasti
  Type type; //Typ oblasti
  Actions actions; //Zvolené akce pro oblast
  area_dimension_t dimension; //Rozměry a umístění oblasti
  size_t page; //Stánka, kde se oblast nachází
}
```

Typy oblasti jsou:

```
enum class Type : unsigned int {
  TEXT,   //Textová oblast
  FORM,  //Textová oblast formulářového charakteru
  IMAGE, //Obecně obrázek
```

[4]Zdroj `https://github.com/open-source-parsers/jsoncpp`
[5]Zdroj `https://launchpad.net/ubuntu/disco/+package/uuid-dev`

```
  LOGO,  //Logo společnosti
  SYNC,  //Synchronizační primitivum
};
```

Typy jako je `IMAGE`, `LOGO` a především `SYNC` jsou nezbytné k nalezení nejvhodnější šablony vstupního dokumentu, tj. v šabloně musí být zastoupen alespoň jeden výskyt těchto typů.

Pro oblast je možné zvolit tyto akce (i jejich):

```
enum class Actions : unsigned int {
  OCR = 0x01,
  ANONYM = 0x02,
};
```

Třída AreaGroup

Slouží k uchování všech oblastí v šabloně, tj. datových struktur `Area`, pro jednu stránku dokumentu v asociativním datovém poli s vygenerovaným unikátním identifikátorem:

```
std::map<std::string, std::shared_ptr<Area>> areas_map;
```

Třída DocTemplate

Reprezentuje celou šablonu a uchovává globální informace o dokumentu s jednotlivými stránkami pomocí datových struktur `AreaGroup`, které jsou uloženy v dynamickém poli, kde index určuje stránku v dokumentu:

```
std::vector<std::shared_ptr<AreaGroup>> area_groups;
```

Třída umožňuje vytvořenou šablonu uložit do formátu JSON, ale i z ní šablonu zpětně načíst pomocí těchto dvou metod:

```
//Vytvoření JSON formátu
bool convert2json(Json::Value &doc_json);
//Načtení šablony z JSON formátu
bool load_from_json(const std::string &path, bool clear);
```

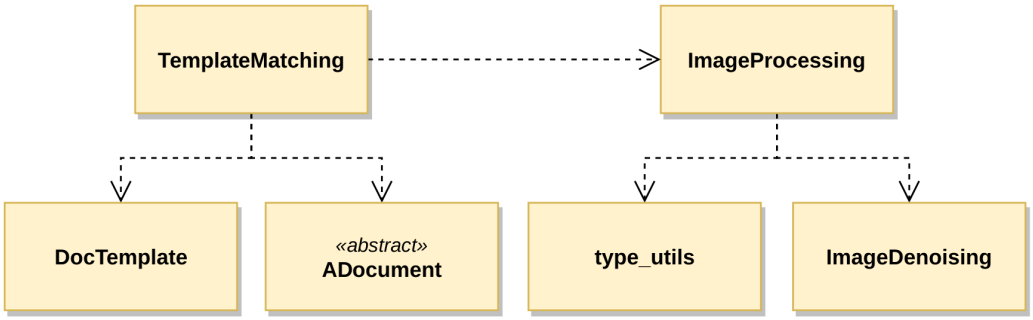
Načtenou šablonu lze otestovat, zda li rozměrově nepřesahuje velikost dokumentu pomocí metody `is_template_fit`:

```
//w a h jsou rozměry aktuálního dokumentu
//w2 a h2 jsou rozměry originální dokumentu
bool is_template_fit(size_t w, size_t h, size_t w2, size_t h2
    ↪ , size_t page);
```

Šablona obsahuje ještě jeden důležitý atribut, a to `confidence_score`, jehož hodnota reprezentuje úroveň shody nalezené šablony pro aktuálně

zpracovávaný dokument vůči referenčnímu dokumentu. Čím více se hodnota blíží 1, tím více se shodují. Tento atribut šablony je nastaven v procesu *automatického nalezení šablony*.

8.3.4 Práce s obrázkem



Obrázek 8.4: UML diagram tříd pro zpracování obrazu.

V diagramu 8.4 lze vidět závislosti mezi třídami `TemplateMatching`, `Image-Processing` a `ImageDenoising`, které implementují algoritmy pro práci s obrázkem. Navíc třída `TemplateMatching` realizuje nalezení nejvhodnější šablony na základě vstupního dokumentu, tudíž obsahuje závislost jak na `DocTemplate` (šabloně), tak i `ADocument` (dokumentu) (viz sekce 8.3.2).

Třída ImageProcessing

Třída `ImageProcessing` je hlavní třída implementující algoritmy pro práci s obrazem. Parametry jednotlivých algoritmů jsou definovány v `type_utils` ve strukturách, které lze jednoduše editovat ve zmíněném konfiguračním souboru `modul_config.cfg`.

V následujícím popisu jsou probrány veškeré implementované algoritmy s prototypem metody a ukázkou části kódu realizující konkrétní algoritmus.

Prahování obrázku je implementováno v metodě `binarization`:

```
1 static double binarization(const cv::Mat &img, cv::Mat &
    ↪ img_out, bool inverted, const ThresholdingConfig &
    ↪ config);
```

Parametr `img` je vstupní obrázek, `img_out` je výstupní binarizovaný obrázek, `inverted` udává, jestli mají být invertovány barvy (černé pozadí, bílé popředí) a `config` obsahuje parametry algoritmu prahování. V následujícím fragmentu kódu jsou obsaženy všechny metody prahování z analýzy (viz 5.4):

```
//Proměnná pro výslednou ideální prahovou hodnotu
double thr_val = -1.;
if (config.method == TH_OTSU) {
  //Otsu metoda
  thr_val = cv::threshold(img_in, img_out, 0, 255, thr + cv::
   ↪ THRESH_OTSU);
} else if (config.method == TH_EKSTEIN) {
  //Ekšteinova metoda
  thr_val = ekstein_binarizatoin(img_in, img_out, thr);
} else if (config.method == TH_WOLF) {
  //Wolfova metoda
  cv::Mat tmp_mat;
  cv::ximgproc::niBlackThreshold(img_in, tmp_mat, 255, thr,
   ↪ config.wolf_block_size, config.wolf_k_param, cv::
   ↪ ximgproc::BINARIZATION_WOLF);
  img_out = tmp_mat.clone();
} else {
  img_out = img_in.clone();
}

return thr_val;
```

Detekce a korekce natočení je na základě analýzy (viz 5.9) implementována v metodě `angle_hough_transformation` (*Houghovy transformace*):

```
std::vector<cv::Vec4i> lines;
//Získání linií pomocí HoughLines metody
cv::HoughLinesP(img, lines, 1, CV_PI / 180., config.threshold
   ↪ , img.cols * config.min_perc_line, mg.cols * config.
   ↪ max_perc_gap);

cv::Mat angles; //Seznam všech úhlů
for (const auto &line : lines) { //Iterace přes linie
  //Výpočet úhlu
  double angle = atan2((double) line[3] - line[1], (double)
   ↪ line[2] - line[0]) * 180. / CV_PI;
  if (std::abs(angle) < config.max_angle) angles.push_back(
   ↪ angle);
}
```

nebo metody `angle_contours` (natočení ohraničujících obdélníků):

```
std::vector<std::vector<cv::Point>> contours;
//Nalezení kontur
cv::findContours(img, contours, cv::RETR_EXTERNAL, cv::
   ↪ CHAIN_APPROX_TC89_KCOS);

cv::Mat angles; //Seznam všech úhlů
for (const auto &it : contours) {
  //Nalezení ohraničujícího obdélníku kolem kontury
```

```
  cv::RotatedRect rect = cv::minAreaRect(it);
  //Objekt příliš malý, nezapočítávat úhel
  if (rect.size.width < 10 && rect.size.height < 10)continue;

  double angle = rect.angle;
  //Pokud je výška větší než šířka, nutné přičíst 90
  if (rect.size.width < rect.size.height) {
    angle = 90 + angle;
  }
  angles.push_back(angle); //Ulož nalezený úhel
}
```

Hledání vzoru je implementované v metodě `template_matching`:

```
static double template_matching(const cv::Rect &area, doc::
    ↪ Type type, const cv::Mat &orig_img, const cv::Mat &
    ↪ current_img, MatchMethod method);
```

Vstupem metody je zájmová oblast `area`, typ oblasti `type`, původní obrázek šablony `orig_img`, akutálně zpracovávaný obraz `current_img` a konkrétní metoda `method` k nalezení vzoru. Významná část implementace je ukázaná v následujícím fragmentu kódu:

```
//desired_roi je oblast z aktuálního obrazu
//cropped_tmp_roi je oblast z originálního obrazu

//Nalezení vzoru v zájmových oblastí
cv::matchTemplate(desired_roi, cropped_tmp_roi, output,
    ↪ cv_method);
//Nalezení max. a min. hodnoty a jejich pozice v obraze
//Metoda i zarovná vzor s hledanou oblastí
cv::minMaxLoc(output, &min, &max, &pmin, &pmax);
if (method == TM_SQDIFF) pmax = pmin; //Pro SQDIFF hledáme
    ↪ min.
cv::Rect found_tmp_rect = {pmax.x, pmax.y, cropped_tmp_roi.
    ↪ cols, cropped_tmp_roi.rows};

cv::Mat final_desired_roi;
if (type == doc::Type::FORM) {
  //U oblasti typu FORM zachovat pouze statický text
  cv::bitwise_and(cropped_tmp_roi, desired_roi(found_tmp_rect
    ↪ ), final_desired_roi);
  //Pro následující metody opět spočítat podobnost
  if (method == TM_COEFF || method == TM_CORREL || method ==
    ↪ TM_SQDIFF) {
    cv::matchTemplate(final_desired_roi, cropped_tmp_roi,
    ↪ output, cv_method);
    cv::minMaxLoc(output, &min, &max, &pmin, &pmax);

```

```
21      //Opět pro SQDIFF hledáme min.
22      if (method == TM_SQDIFF) pmax = pmin;
23    }
24  } else {
25    //Pro ostatní typy oblastí
26    final_desired_roi = desired_roi(found_tmp_rect);
27  }
28
29  if (method == TM_MASKING) {
30    //Výpočet pomocí maskování
31    //Operace AND nad oblastmi a výpočet poměru pixelů popředí
32    max = (double) cv::countNonZero(cropped_tmp_roi &
        ↪ final_desired_roi) / nonz_tmp_roi.size();
33  } else if (method == TM_SSIM) {
34    //Výpočet podobnosti pomocí SSIM
35    max = calc_SSIM(cropped_tmp_roi, final_desired_roi)[0];
36  }
37
38  return max;
```

Funkce `cv::matchTemplate` na řádce 5 spočte nejen podobnost obou oblastí, ale pokud se v nich nachází stejný vzor, jsou oblasti zarovnány podle hledaného vzoru.

Škálování obrazu se nachází v metodě `resize_image`:

```
1  void resize_image(const cv::Mat &img, cv::Mat &out_img, const
       ↪  cv::Size &size, double sx, double sy, ResizingMethod
       ↪ method);
```

Parametr `img` je vstupní obrázek, `out_img` je výstupní přeškálovaný obrázek, `size` je rozměr (šířka a výška), `sx` a `sy` jsou škálovací parametry (o kolik se má obrázek škálovat), a `method` je specifická metoda škálování. Byly implementovány všechny možné metody dle analýzy (viz 5.8). Pro interpolaci pomocí *nejbližšího souseda*, *bilineární*, *bikubickou* a *Lanczosovu* je použita knihovní funkce `cv::resize`, nicméně pro techniku *Super-rozlišení* využitím *konvoluční neuronových sítí* je nutné použít datovou strukturu a s ní související metody z modulu `dnn_superres` **OpenCV**:

```
1  if (method >= ResizingMethod::TR_NN && method <=
       ↪ ResizingMethod::TR_LANCZOS) {
2    //Klasická interpolace dle zmíněných technik
3    cv::resize(img, out_img, size, sx, sy, map_resize_to_cv(
       ↪ method));
4  } else {
5    //Obraz nutné převést do šedotónového
6    cv::Mat src = autoconvert_color(img);
7    //Škálovací faktor jako maximum z (sx, sy), nebere se v
       ↪ potaz konkrétní zvětšení
```

```
  auto sc = static_cast<size_t>(std::max(sx, sy));
  //Datová struktura implementující Super-Resolution
  cv::dnn_superres::DnnSuperResImpl sr;
  //Konkrétní cesta k modu z konfigurace
  std::string model = cfg::ConfigLoader::
   ↪ get_super_ress_model_path() + std::to_string(sc)+ cfg::
   ↪ ConfigLoader::get_super_ress_ext();
  //Načtení příslušného modelu
  sr.readModel(model);
  //Nastavení modelu dle názvu a škálovacího parametru
  sr.setModel(cfg::ConfigLoader::get_super_ress_model_name(),
   ↪  sc);
  //Zvetšení obrazu
  sr.upsample(src, out_img);
}
```

Nutno podotknout, že škálování pomocí *Super-rozlišení* má natrénované modely převážně pro *upsamling*, tj. zvětšení obrazu.

Odšumění obrazu je realizováno v metodě `denoising_image`:

```
void denoising_image(const cv::Mat &img, cv::Mat &out_img,
   ↪ const DenoisingConfig &config);
```

Parametr `img` je vstupní obraz, `out_img` je výstupní odšuměný obraz a `config` je struktura s parametry algoritmu. V rámci této metody jsou implementovány obě techniky zmíněné v analýze (viz 5.7). Pro upřesnění je implementována pouze technika *Stochastického odšumění obrazu*, jelikož technika *Non-Local Means Denoising* je součástí knihovny **OpenCV**. Tělo metody `denoising_image` vypadá takto:

```
if (config.method == DenoisingMethod::DE_STOCHASTIC) {
  //Stochastické odšumění je implementováno ve třídě
   ↪ ImageDenoising
  out_img = ImageDenoising::run_denoising(img, config.
   ↪ stoch_dev, config.num_walks);
} else {
  //Funkce pro Non-Local Means odšumění z knihovny
  cv::fastNlMeansDenoising(img, out_img, config.h, config.
   ↪ temp_win_size, config.search_win_size);
}
```

Ve struktuře `config` se nacházejí parametry pro obě techniky, které je možné definovat v konfiguračním souboru `modul_config.cfg`.

Odstranění čar realizuje nalezení a odstranění horizontálních a vertikálních čar v obraze:

```
static void line_detection_and_removing(const cv::Mat &img,
   ↪ cv::Mat &img_out, const cv::Size &size);
```

Vstupní parametr `img` je vstupní obrázek, `img_out` je obrázek zbavený nalezených čar a `size` udává velikost zkoumané oblasti.

Nalezení je realizováno využitím morfologických operací takto:

```
cv::Mat horizontal;
//Binarizace vstupního obrazu
ia::ImageProcessing::binarization(img, horizontal, true);
cv::Mat vertical = horizontal.clone();

//Vytvoření elementu (okénka) obdélníkového tvaru
//Výška je 1 - jedná se o element pro horizontální čáry
cv::Mat hs = cv::getStructuringElement(cv::MORPH_RECT, cv::
    ↪ Size(size.width, 1));
cv::Mat hs2 = cv::getStructuringElement(cv::MORPH_RECT, cv::
    ↪ Size(size.width, 2));
//Element s šířkou 1 - pro vertikální čáry
cv::Mat vs = cv::getStructuringElement(cv::MORPH_RECT, cv::
    ↪ Size(1, size.height));
cv::Mat vs2 = cv::getStructuringElement(cv::MORPH_RECT, cv::
    ↪ Size(2, size.height));

//Aplikace eroze a dilatace pro horizontální čáry
erode(horizontal, horizontal, hs, cv::Point(-1, -1));
dilate(horizontal, horizontal, hs2, cv::Point(-1, -1));
//Aplikace eroze a dilatace pro verikální čáry
erode(vertical, vertical, vs, cv::Point(-1, -1));
dilate(vertical, vertical, vs2, cv::Point(-1, -1));

//Aplikování součtu s originálním obrazem to odstraní čáry
img_out = img + horizontal + vertical;
```

Třída ImageDenoising

Jelikož technika *stochastické odšumění obrazu* není součástí knihovny **OpenCV**, byla nutná její implementace v rámci třídy `ImageDenoising`. Technika je výpočetně náročná již pro celkem malý obraz (cca od 400×400 pixelů), proto byla pro výpočet použita knihovna **TBB**[6] pro paralelní výpočty.

Jednu řádku obrazu zpracovává právě jedno vlákno (řízeno knihovnou), kde pro každý pixel v řádce je vypočteno k *náhodných cest* (jako konfigurační parametr); délka cesty závisí na podobnosti navštívených pixelů v cestě, tj. nelze pouze na základě znalosti velikosti obrázku odvodit dobu výpočtu. Metoda `run_denoising` realizuje vykonání algoritmu:

```
void run_denoising(const cv::Mat &image, cv::Mat &out_img,
    ↪ double dev, int samples);
```

[6]Zdroj `https://software.intel.com/en-us/tbb`

Parametr `image` je vstupní obrázek, `out_img` je výsledný odšuměný obraz, `dev` je standardní odchylka a `samples` určuje počet cest pro každý pixel.

Třída TemplateMatching

Třída `TemplateMatching` se stará o proces nalezení nejvhodnější šablony na podle dokumentu. Každé zpracované šabloně se nastaví hodnota `confidence_score`, která určuje úroveň toho, jak moc je šablona shodná s právě zpracovávaným dokumentem; čím vyšší hodnota, tím lepší šablona (maximum je 1).

Třída má pouze jednu veřejnou statickou metodu `start_matching`, která spouští proces hledání šablony:

```
static bool start_matching(std::shared_ptr<MatchingConfig>
    ↪ config);
```

Obsahuje jediný vstupní parametr, a to strukturu `config`, která obsahuje tyto atributy:

```
struct MatchingConfig {
  //Konfigurační parametry
  tmatch::TMConfig tm;
  //Indikuje konec hledání (pro paralelní proces)
  bool finished;
  //Callback funkce která je zavolána s nalezenou šablonou
  std::function<void(int)> func; //Callback funkce
  //Index do pole s nejvhodnější šablonou
  size_t idx_best_match;
  //Pole nalezených šablon
  std::vector<DocTemplate> template_list;
  //Vstupní dokument
  std::shared_ptr<ADocument> document;

  //Přidání nově nalezené šablony
  void push_template(DocTemplate &&temp);
  //Indikuje, jestli pole šablon je prázdné
  inline bool is_empty() const;
  //Vrací nejideálnější šablonu (konstantní reference)
  inline const DocTemplate &get_best_match() const;
  //Vrací nejideálnější šablonu (referenci)
  inline DocTemplate &get_best_match();
}
```

Samotný algoritmus hledání ideální šablony je implementován v privátní metodě `matching`:

```
double matching(MatchingConfig &config, DocTemplate &doc_temp
    ↪ ) {
```

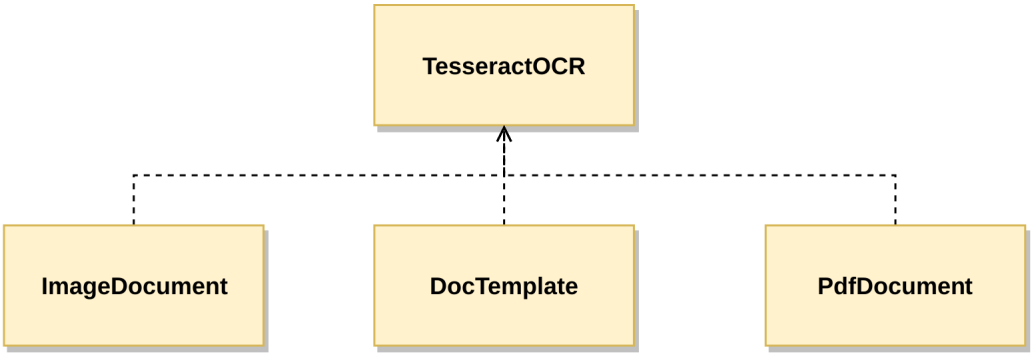
```
 ...
const auto &areas = doc_temp.get_templates();
//Iterace přes stránky šablony
for (auto &page : areas) {
  ...
  //Získání originálního a aktuální obrazu v dokumentu
  auto current_img = config.document->get_image_page(page_num
   ↪ );
  auto orig_image = orig_doc->get_image_page(page_num);

  //Normalizování aktuálního obrazu na velikost originálního
  cv::Size rescale = {orig_image->get_width(), orig_image->
   ↪ get_height()};
  const ia::DeskewingConfig conf = cfg::ConfigLoader::
   ↪ get_deskewing_config();
  //Příprava obrazů pro výpočet jejich podobnosti
  auto proc_current_img = ia::ImageProcessing::
   ↪ process_image_for_matching(current_img->get_img(), conf
   ↪ , rescale);
  auto proc_orig_img = ia::ImageProcessing::
   ↪ process_image_for_matching(orig_image->get_img(), conf)
   ↪ ;
  //Iterace přes všechny oblasti na stránce šablony
  for (const auto &area : page->get_map_areas()) {
    if (config.tm.force_stop) break; //Vynucené ukončení
    //Struktura zkoumané oblasti
    const auto &area_struct = area.second->get_area_struct();
    //Oblast s dynamickým textem se neporovnává
    if (area.second->get_area_struct().type == TEXT)continue;
    //Velikost oblasti (x,y,šířka,výška)
    const auto &area_dimension = area_struct.dimension;
    cv::Rect area_rect(area_dimension._x * conf.sx,
   ↪ area_dimension._y * conf.sy, area_dimension._w * conf.
   ↪ sx, area_dimension._h * conf.sy);
    //Volání metody z ImageProcessing pro výpočet podobnosti
    score += ia::ImageProcessing::template_matching(area_rect
   ↪ , area_struct.type, proc_orig_img, proc_current_img,
   ↪ config.tm.matching_method);

    ++total_areas;
  }
}
//Normalizovaná výsledná hodnota confidence_score
if (score == 0 || total_areas == 0) return 0;
return (score / total_areas);
}
```

8.3.5 Rozpoznání textu

Z výsledku analýzy 6.1 byl jednoznačné vybrán nástroj `Tesseract-OCR`, který je použit k rozpoznání textu z obrázku v třídě `TesseractOCR` ve jmenném prostoru `ocr`. V diagramu 8.5 lze vidět, které třídy využívají třídu `TesseractOCR`.



Obrázek 8.5: UML diagram závislostí OCR třídy.

Třída slouží také k vytvoření textové vrstvy ve výstupním souboru PDF, kde implementace byla převzata z již hotového řešení přímo z nástroje `Tesseract-OCR`, respektive z třídy `TessPDFRenderer`. Umístění výsledné textové vrstvy je uloženo v datových strukturách zmíněného nástroje (výpočet rotace textu, výpočet *baseline* slova, kódování do UTF-16 a výpočet velikosti písma). K tomu `Tesseract-OCR` používá vlastní definovaný font, který je umístěn v adresáři `models/fonts/pdf.ttf`.

Vlastní implementované metody jsou tyto:

```
private:
  //Implementace procesu OCR
  static bool impl_ocr_template(ocr::extract_settings &config
   ↪ );
  //Anonymizování konkrétní oblasti
  static void anonymize_area(Image &img, Image &img_orig,
   ↪ const doc::area_t &area);
public:
  //Volání procesu OCR + anonymizace (je-li nastaveno)
  static bool run_ocr(std::shared_ptr<ocr::extract_settings>
   ↪ config);
  //Pouze anonymizace oblastí (je-li nastaveno)
  static bool anonymize_areas(std::shared_ptr<ocr::
   ↪ extract_settings> config);
  //Vložení textové vrstvy do PDF souboru
  static bool append_text_layer(const std::string &content,
   ↪ PoDoFo::PdfPage *page, PoDoFo::PdfDocument &doc, PoDoFo
   ↪ ::PdfReference &font, bool create_font = false);
```

Zásadní fragmenty implementace procesu OCR jsou popsány v následujícím úseku kódu:

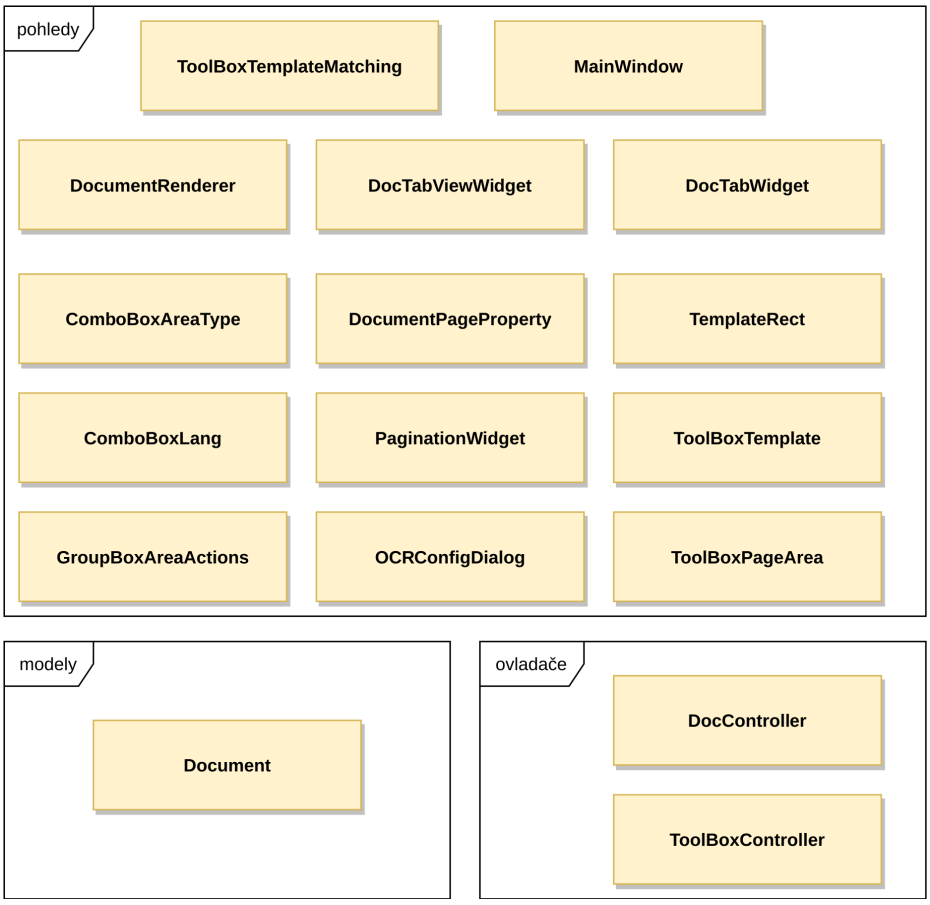
```
//Implementace impl_ocr_template
 ...
//Inicializace tesseractu s příslušným jazykem
std::string lang = config.lang_auto ? "auto" : config.
    ↪ doc_template->get_lang();
if (api->Init(cfg::ConfigLoader::get_lang_models_ocr().c_str
    ↪ (), lang.c_str(), tesseract::OcrEngineMode::
    ↪ OEM_LSTM_ONLY) != 0) {
  throw pdf_excpetion::PDFTesseractException();
}
 ...
///Nastavením paremetrů tesseraktu (penalizace slov)
api->SetVariable("language_model_penalty_non_dict_word",
    ↪ penalty.c_str());
 ...
//Nastavení vstupního obrazu
api->SetImage(pix_img.get());
//Nastavení Výsledného rozlišení obrazu dle vykreslovacího pl
    ↪ átna, většinou se jedná o velikost PDF stránky
api->SetSourceResolution(std::max(pix_img->xres, pix_img->
    ↪ yres));
 ...
//Nastavení pozice oblasti v obraze pro rozpoznání textu
api->SetRectangle(dim._x * sc, dim._y * sc, dim._w * sc, dim.
    ↪ _h * sc);
//Volání samotného procesu rozpoznání
if (api->Recognize(config.use_monitoring ? config.monitors[i
    ↪ ]->monitor.get() : nullptr) == 0) {
   ...
  if (config.overlay && !doc::is_action_set(area.actions, doc
    ↪ ::Actions::ANONYM)) {
    //Vytvoření textové vrstvy pro PDF
    const char *text = GetPDFTextObjects(api.get(), page_h);
    page_text += text;
  }

  //Extrahování nalezeného textu
  std::string extracted_block = get_text_line(api->
    ↪ GetIterator());
   ...
}
//Ukončení tesseraktu a uvolnění paměti
api->End();
 ...
```

9 Software pro práci se šablonami

V této kapitole je popsána implementace softwaru pro tvorbu a manipulaci se šablonami, který využívá modul počítačového vidění a zpracování dokumentů (viz kapitola 8). Kvůli kompatibilitě se zmíněným modulem a na základě stejných důvodů byl k implementaci zvolen programovací jazyk C++ a knihovna **Qt5** popsaná v analýze (viz kapitola 7).

Architektura softwaru dodržuje principy architektonického vzoru `MVC` (Model-View-Controller, tj. model-pohled-ovladač), což znamená důsledné oddělení logiky od uživatelského rozhraní. V diagramu tříd na obrázku 9.1 lze vidět přehled veškerých tříd aplikace.

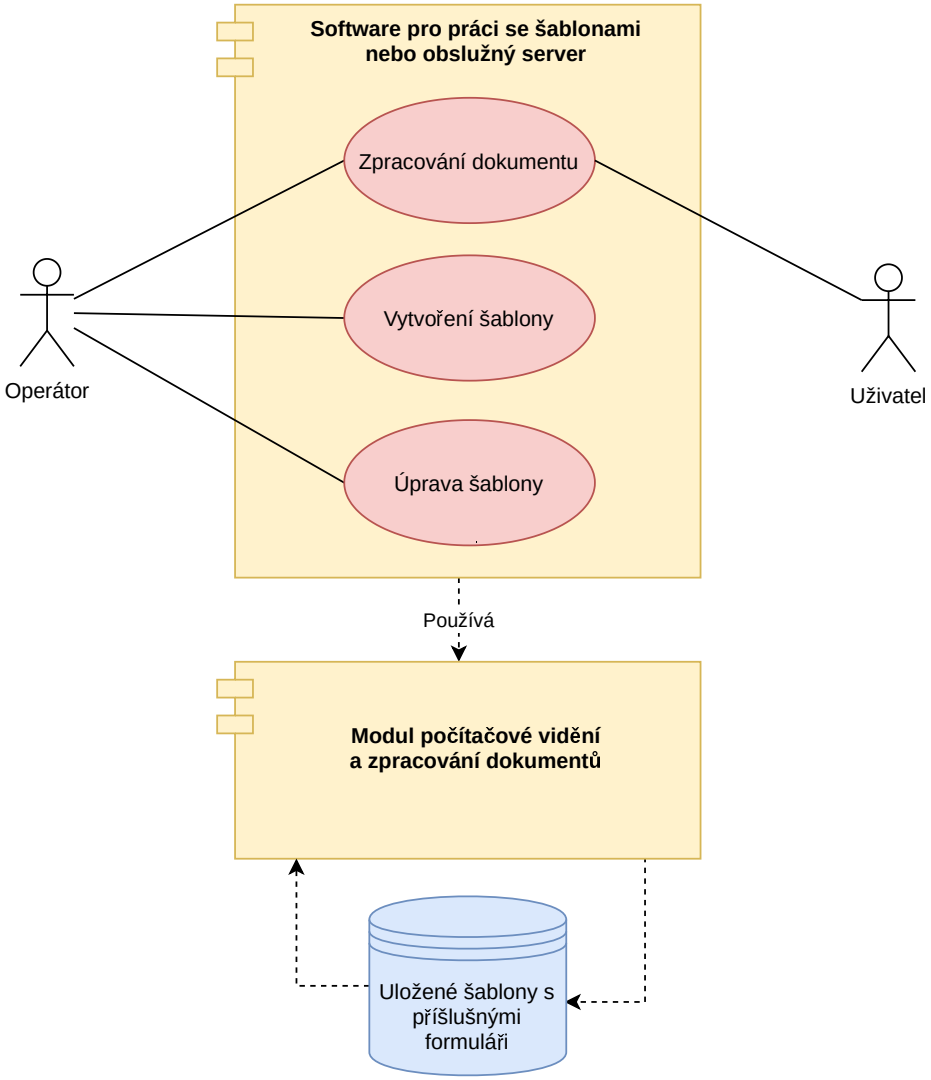


Obrázek 9.1: Diagram tříd softwaru pro práci se šablonami.

Model v tomto kontextu reprezentuje načtený dokument s příslušnou

šablonou v aplikaci. Ovladač implementuje logiku jednotlivých akcí podporovaných v aplikaci a stará se o zpracování uživatelského vstupu. Pohledy vytvářejí grafické prvky a definují základní metody jejich obsluhy.

Na obrázku 9.2 je typický případ užití software pro práci se šablonami společně s modulem počítačového vidění a zpracování dokumentů. Operátor je většinou ten, kdo vytváří šablony např. na základě nového typu dokumentu, popř. upravuje aktuální šablony či odstraňuje již nepotřebné, tj. jde o osobu oprávněnou používat veškeré funkce systému. Na druhou stranu uživatel je osoba, která zpracovává pouze vyplněné dokumenty. Jako reálný



Obrázek 9.2: Příklad případu užití aplikace pro práci se šablonami.

příklad si lze představit pracovníka na bankovní překážce, jenž potřebuje zpracovat naskenovanou smlouvu, tj. extrahovat užitečné informace nebo je

anonymizovat (či obojí) na základě již vytvořené šablony.

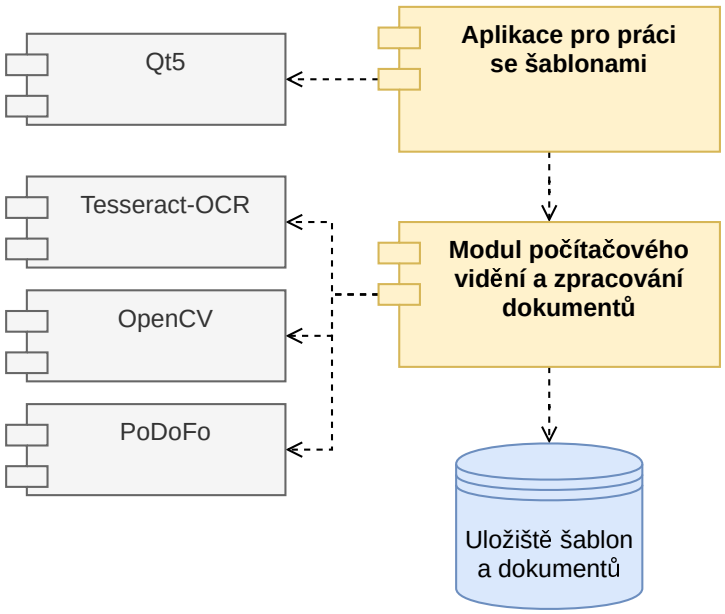
Během celého vývoje této aplikace bylo prováděno uživatelské testování. Uživatel, který se podílel na vytváření testovacího datového souboru ohlašoval nalezené chyby. Veškeré tyto chyby a nedostatky aplikace byly opraveny a uživateli byla dodána nová verze s opravenými chybami, popř. novými funkcemi (podle fáze vývoje).

9.1 Implementace aplikace

V této části jsou popsány pouze významné třídy, jenž vykreslují a pracují se šablonami, respektive dokumenty. Pohledy jsou implementovány využitím tzv. *widgetů*, popř. jejich konkrétními typy, které definují správu grafických komponent v aplikaci. Ovladače a modely jsou potomkem objektu `QObject` zejména z toho důvodu, aby bylo možné využívat mechanismu *Signals & Slots* (*signálů a slotů*), který slouží pro komunikaci mezi objekty.

9.1.1 Model dokumentu

Model dokument realizuje pouze jedna třída, která obsahuje objekt odpovídající aktuálně zpracovávanému dokumentu a zároveň příslušnou šablonu. Obě tyto datové struktury jsou součástí modulu pro zpracování dokumentů, který je integrován v této aplikaci, jak je možné vidět v UML digramu na obrázku 9.3.



Obrázek 9.3: UML digram závislostí aplikace a modulu.

Třída Document

Třída `Document` obsahuje datové struktury `ADocument` a `DocTemplate`, které jsou součástí modulu počítačového vidění a zpracování dokumentu, takže funguje jako *obalová třída*. Navíc obsahuje informaci o aktuálně zpracovávané stránce dokumentu a jednotlivých obrázcích ze stránek, které jsou konvertovány do datové struktury `QPixmap` a následné uchovány v *cache* paměti; k tomu slouží asociativní pole `QMap`, kde klíčem je konkrétní stránka:

```
QMap<size_t, QPixmap> cached_images;
```

Třída se stará hlavně o konverzi datových struktur mezi modulem a knihovnou **Qt**. Ty jsou pak pomocí ovladače `DocController` zpracovány a předány příslušným pohledům, popř. se pohledy sami dotazují k získání obsahu přímo modelu. Dále jsou implementovány metody pro přidání, modifikaci či odstranění šablony, respektive oblastí v rámci šablony. Třída definuje také tři signály, které oznamují změnu interních dat, na které mohou reagovat ostatní objekty:

```
signals:
  // Oznámení o změně konkrétní oblasti
  void notify_update_area(const doc::area_t &area);
  //Oznámeni o odznačený oblasti
  void notify_clear_selection();
  //Oznámení o změně šablony
  void notify_update_template(const doc::template_t &temp,
   ↪ doc::doc_t &doc);
```

Třída DocumentList

Třída `DocumentList` je kontejner, který uchovává veškeré načtené dokumenty, tj. šablony. K uchování dokumentu slouží asociativní pole `QHash`, kde klíčem je identifikátor šablony:

```
//Hash struktura pro uchování datové struktury Document
QHash<QString, Document *> map_documents;
```

V aplikaci není dovoleno upravovat dvě stejné šablony (se stejným identifikátorem) současně, aby se zabránilo nechtěnému přepisování atributů stejných šablon.

Signály této třídy jsou totožné s těmi v datové struktuře `Document` a při vytvoření jsou navzájem propojeny, což znázorňuje následující fragment kódu:

```
//Propojení signálů s třídou Document
connect(doc, &Document::notify_clear_selection, this,
   ↪  &DocumentList::notify_clear_selection);
```

```
connect(doc, &Document::notify_update_area, this,
    ↪ &DocumentList::notify_update_area);
connect(doc, &Document::notify_update_template, this,
    ↪ &DocumentList::notify_update_template);
```

9.1.2 Ovladače aplikace

Ovladače obsahují většinu logiky celé aplikace a zajišťují integritu mezi modelem a pohledy, tj. jedná se o prostředníky mezi nimi. Uživatelské vstupy jsou zpracovávány přes tyto ovladače `DocController` a `ToolBoxController`. Dále je popsána implementace ovladače `DocController`.

Třída DocController

Třída `DocController` realizuje veškerou logiku aplikace. Zajišťuje načtení, uložení a modifikaci dokumentu a šablony. Dále také zpracování dokumentu podle příslušné šablony, nalezení ideální šablony, detekci a korekci natočení obrázku, odšumění obrazu, načtení výchozího obrázku a export do souboru PDF.

Jedná se o komplexní třídu, tudíž budou dále popsány jen důležité či zajímavé úseky implementace. Veškeré veřejné metody jsou označeny jako `sloty`, takže je možné spojit je pomocí signálu s jiným objektem (např. jiným ovladačem). Metody začínající slovem `update` se starají o aktualizaci modelu a pohledů, jako např.:

```
//Aktualizace modelu v DocumentList
void update_area(const QString &doc_id, const doc::area_t &
    ↪ area);
//Aktualizace informací o oblasti v pohledu
void update_toolbox_area_properties(const doc::area_t &area);
```

Naopak metody, které žádají jiné objekty o aktualizaci začínají metodou `notify` (signály). Třída obsahuje čtyři takové signály:

```
signals:
  //Žádost o aktualizování popisku o aktuální stránce
  void notify_changed_page(size_t page);
  //Žádost o aktualizování popisku o celkovém počtu stránek
  void notify_changed_total_pages(size_t total_pages);
  //Žádost o aktualizaci vykreslovaného dokumentu a šablony
  void notify_update_renderer();
  //Žádost o změnu stránky
  void notify_change_page();
```

Aby grafické uživatelské rozhraní zůstalo responsivní při náročnějších výpočtech, jako je zpracování dokumentu podle šablony či nalezení šablony,

jsou využity prostředky knihovny pro paralelizaci pomocí třídy `QtConcurrent` následovně:

```
  ...
//Vytvoření sdíleného ukazetele s nastavením
auto config = std::make_shared<ocr::extract_settings>(
    ↪ ocr_dialog.get_current_config());
  ...
//Spuštění OCR ve vlákně se sdíleným ukazatelem
QtConcurrent::run([&] {
  ocr::TesseractOCR::run_ocr(config);
});
```

Díky použitému sdílenému ukazateli `std::shared_ptr` je možné k němu přistupovat z více vláken s jistotou, že jiné vlákno neuvolní jeho paměť.

9.1.3 Pohledy

Pohledů je celá řada a starají se o vykreslení grafických komponent v aplikaci. Jedná se zejména o objekty typu `QWidget`. Avšak nejdůležitější je především třída `DocumentRenderer` vykreslující vstupní obraz a šablonu. Šablona se skládá ze zájmových oblastí vymezených pomocí třídy `TemplateRect`. Tyto dvě třídy budou dále blíže popsány.

Třída DocumentRenderer

Třída `DocumentRenderer` je potomkem knihovní třídy `QGraphicsView` a stará se o vykreslení vstupního dokumentu společně se šablonou, která překrývá obraz stránky. K vizualizace je využita třída `QGraphicsScene` (lze si představit jako plátno). V pozadí je vykreslen obrázek (dokument) a do popředí je možno umisťovat zájmové oblasti, které tvoří vrstvu nad vstupním obrazem (nekreslí se do obrazu).

Tento pohled obsahuje také ukazatel na konkrétní dokument který je v ní vykreslován. Veškeré zájmové oblasti v šabloně jsou uloženy v seznamu:

```
//Jedná se stále oblasti v šabloně
QList<QGraphicsItem *> items_list;
//Jedná se o dočasné oblasti (náhled)
QList<QGraphicsItem *> temp_items_list;
```

Pro přidání nové a odstranění oblasti jsou implementovány tyto metody:

```
//Přidání oblasti na základě pozice kurzoru myši na platně
void add_template_area(const QPointF &pos);
//Přidání oblasti na základě konkrétní struktury
void add_template_area(const doc::area_t &area_struct);

```

```
//Odstraní právě vybranou oblast
void remove_template_area();
//Odstraní všechny oblasti
void clear_areas();
```

První metoda slouží k vytvoření zcela nové oblasti, naopak druhá je použita pro již vytvořenou oblast, která je např. načtena ze souboru.

Poslední důležitou metodou pro překreslení aktuálního plátna novým obrázkem, tj. stránkou dokumentu, je tato metoda:

```
//Vykreslení obrazu pomocí datové struktury QPixmap
void set_document_pixmap(const QPixmap &image);
```

Obraz je přeškálován tak, aby se vešel na aktuální plátno.

Třída TemplateRect

Třída `TemplateRect` reprezentuje zájmovou oblast a je potomkem knihovní třídy `QGraphicsRectItem`, který obecně reprezentuje grafický prvkek tvaru obdélníku. Každá taková oblast je definována svými rozměry a počátečním bodem umístěným v levém horním rohu. Třída využívá výhody rodičovské třídy, která již implementuje funkce jako posun, označení, apod. Nicméně musela být implementována vlastní metoda `resize_area` pro zvětšování oblasti.

9.1.4 Struktura šablony

Na základě požadavků (viz kapitola 2) byla definována struktura JSON, které reprezentuje uloženou šablonu. V následujícím fragmentu kódu je struktura popsána:

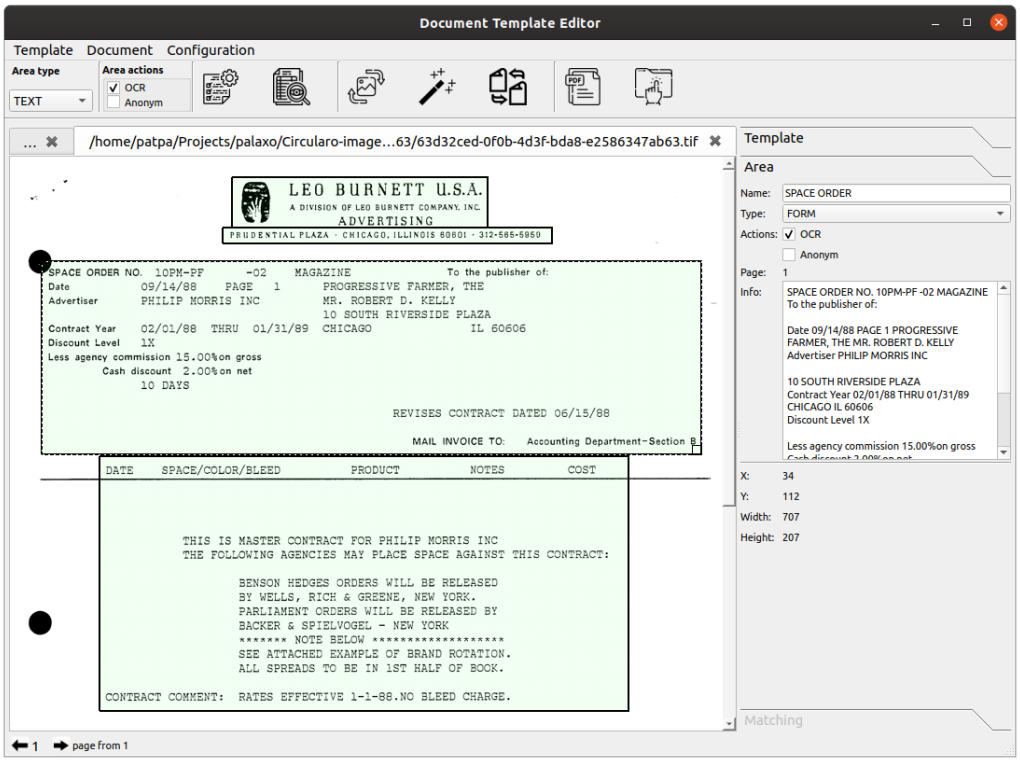
```
{
"areas" :
{
  "1" : //Konkrétní stránka
  [
    {
      "actions" : "OCR", //Akce oblasti
      "description" : "Date: 4/10/98", //Popis či výstup OCR
      "dimension" : //Velikost a pozice oblasti
       {
         "h" : 23,   //Výška
         "w" : 106,  //Šířka
         "x" : 113,  //Pozice na ose x
         "y" : 187   //Pozice na ose y
       },
       //Identifikátor oblasti
```

```
        "id" : "{c0630a78-036e-4f93-97ea-ebfa52e3d28b}",
        "name" : "Date", //Název oblasti
        "page" : 0, //Index stránky (stránka - 1)
        "type" : "FORM" //Typ oblasti
      }
  ]
},
"description" : "", //Popis šablony
"doc_format" : "tif", //Formát vstupního obrazu
//Identifikátor šablony
"id" : "{8cda5321-6939-4fa2-80a2-a7bd2f7f18e5}",
"lang" : "eng", //Jazyk textu obsažený v dokumentu
"name" : "Invoice", //Název šablony
//Cesta k originálnímu dokumentu
"original_doc" : "./documents/doc123.pdf",
"page_num" : 1 //Celkový počet stran
}
```

9.1.5 Grafické uživatelské rozhraní

Návrh grafického uživatelského rozhraní (GUI) této aplikace bylo inspirováno používanými grafickými editory. Na obr. 9.4 je možné vidět snímek obrazovky implementované aplikace pro práci se šablonami.

Uprostřed aplikace je umístěno plátno, kde je vykreslen dokument (obrázek) a šablona (zájmové oblasti). Nad plátnem se nachází záložky s načtenými dokumenty a jejich šablonami. O něco výše jsou umístěny tlačítka, která obsluhují implementované techniky (OCR, hledání vzoru, atd.). V menu je možno načítat/ukládat dokumenty společně se šablonou. V pravé části obrazovky se nacházejí prvky s editovatelnými informacemi/vlastnostmi aktuálně označené zájmové oblasti, popř. celé šablony. Zápatí obsahuje dvě tlačítka pro změnu stránky načteného dokumentu. Podrobnější popis aplikace lze nalézt v uživatelské příručce (viz příloha A).



Obrázek 9.4: Snímek obrazovky aplikace pro práci se šablonami.

10 Dosažené výsledky

V této kapitole jsou shrnuty dosažené výsledky a porovnání jednotlivých technik a algoritmů pro práci s obrázky implementovaných v modulu počítačového vidění (viz kapitola 8), jež byly předmětem analýzy (viz kapitola 5 a 6). Nicméně cílem těchto evaluačních testů je ověřit stabilitu a funkčnost implementovaných algoritmů vzhledem k charakteru a účelu této práce.

Některé algoritmy, jako je *detekce úhlu natočení obrazu* či *odšumění obrazu* obsahují konfigurovatelné parametry, které musejí být nastaveny. Takovéto parametry ovlivňují výsledek algoritmu, a to hlavně tehdy, jedná-li se o různé transformace vstupního obrázku, kde špatně zvolené parametry mohou způsobit jeho výraznou nečitelnost, tj. opačný efekt než byl očekáván.

Z tohoto důvodu byly pro zmíněné algoritmy empiricky nalezeny ideálních parametry z množství jejich kombinací s využitím připravených testovacích dat. Jako testovací data byly použity převážně staré formuláře a faktury v anglickém jazyce ze souboru dat (dokumentů) **RVL-CDIP** [19], které byly připraveny jiným uživatelem k testování s využitím implementovaného modulu pro práci se šablonami (viz kapitola 9), tudíž testovací data nebyla přizpůsobena výsledkům evaluačních testů prezentovaných v této kapitole.

Testovací datový soubor obsahuje celkem 78 odlišných dokumentů, z toho 6 formátu PDF a 72 v obrazovém formátu TIFF, pro které navíc byly vytvořeny dvě různé rotace obrázku s odlišným směrem natočení a velikostí úhlu. Ke všem dokumentům byl vytvořen i odpovídající formulář, tj. byly ručně odstraněny variabilní (dynamické) bloky textu; tento typ dokumentu je pak využit jako výchozí. V součtu datový soubor obsahuje 292 dokumentů k testování.

Ke každému výchozímu dokumentu ve skupině byla vytvořena příslušná šablona, která obsahuje množinu zájmových oblastí různých typů s různými akcemi. Pro každou oblast obsahující text, tj. zvolená akce je `OCR`, byl tento text přepsán a slouží jako referenční zdroj k vyhodnocení úspěšnosti rozpoznávání textu pomocí OCR systému **Tesseract-OCR**.

10.1 Detekce natočení

Jak již bylo popsáno v úvodu kapitoly, testovací data obsahují různě natočené dokumenty, jedná se celkem o 144 různých rotací (hodnota úhlu natočení je součástí názvu souboru), uvážíme-li i výchozí dokument jedná se dohromady o 216 rotací dokumentů. Přestože byla snaha výchozí dokument

vycentrovat, i tak v mnoha případech stále zůstává nepatrný úhel natočení, proto je nalezený úhel výchozího dokumentu zohledněn ve výsledném výpočtu.

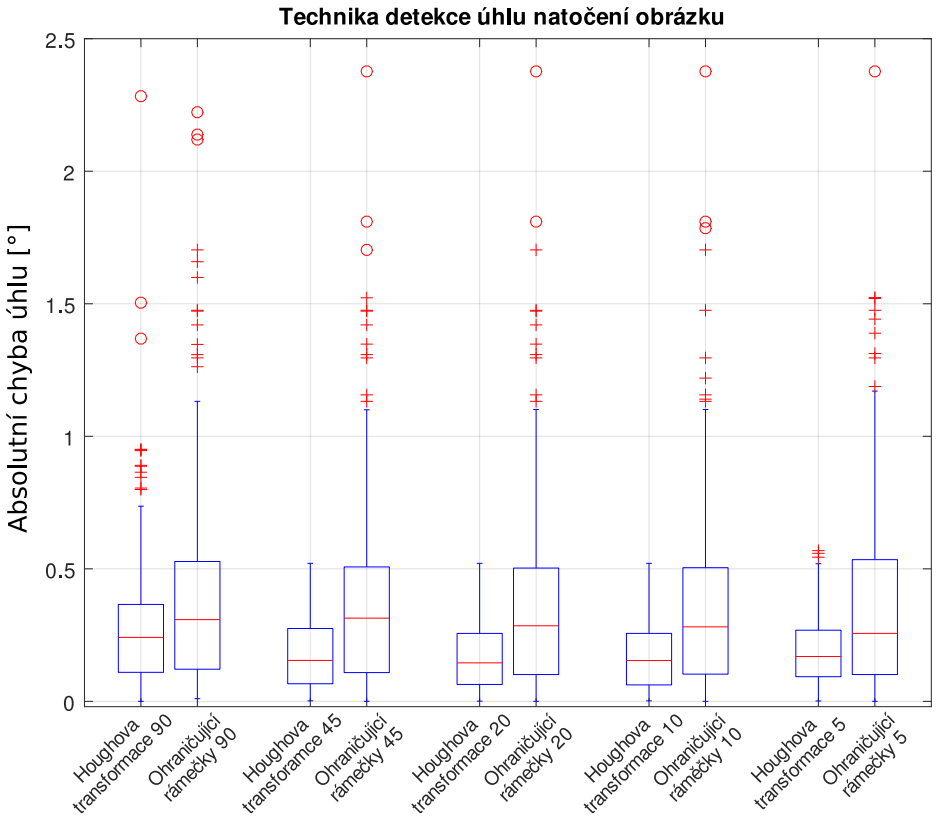
Nalezená hodnota úhlu natočení je vypočtena na základě absolutní chyby:

$$\theta_i = |y_i - x_i|, \tag{10.1}$$

kde $y_i = |\theta_{\text{referenční}} - \theta_{\text{výchozí}}|$ a $x_i = |\theta_{\text{nalezený}}|$. Celý testovací soubor je reprezentován sumou těchto absolutních chyb takto:

$$\Theta = \frac{\sum_{i=1}^{N} |y_i - x_i|}{N} = \frac{\sum_{i=1}^{N} \theta_i}{N}, \tag{10.2}$$

kde je N je celkový počet dokumentů v souboru dat.



Obrázek 10.1: Porovnání přesnosti technik k detekci úhlu natočení obrázku.

Graf v obrázků 10.1 s příslušnou tabulkou hodnot 10.1 ukazují absolutní chyby nalezených hodnot úhlů natočení obrázku pomocí techniky *Houghovy transformace* a *ohraničujících rámečků*. Bylo zjištěno, že omezíme-li maximální možnou hodnotu úhlu shora, tj. a priori známe horní hranici, je detekce stabilnější – zejména v případě techniky *Houghovy transformace*. Použitá

	Absolutní chyba úhlu [°]				
Metoda	Medián	Průměr	Odchylka	Min.	Max.
Hough 45	0,293	0,502	0,660	0,002	4,499
Rámečky 90	0,380	0,488	0,482	0,013	2,545
Hough 45	0,182	0,240	0,232	0,004	1,418
Rámečky 45	0,249	0,400	0,444	0,007	2,278
Hough 20	0,173	0,203	0,173	0,003	1,195
Rámečky 20	0,247	0,401	0,431	0,007	2,024
Hough 10	0,181	**0,196**	**0,145**	0,007	1,098
Rámečky 10	0,234	0,394	0,425	0,003	2,024
Hough 5	**0,162**	0,204	0,152	**0,001**	**1,015**
Rámečky 5	0,311	0,456	0,492	0,001	2,311

Tabulka 10.1: Přehled absolutních chyb uhlů natočení obrázku.

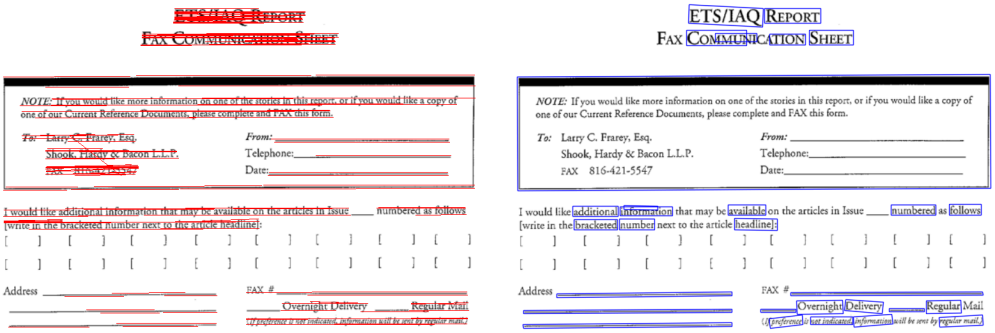
hodnota maximálního možného úhlu natočení je v obr. 10.1 umístěna za názvem techniky na ose x, tj. 90, 45, 20, 10 a 5 stupňů. Nejstabilnější výsledky přináší technika *Houghovy transformace* pro maximální zvolený úhel 10 a 5 stupňů s chybou pouze $0,197$ a $0,204$ stupně. Na druhou stranu metoda *ohraničujícím ráměčků* obsahuje v každém případě množství extrémních hodnot (úhlů) s větším rozptylem a chybou úhlu vždy kolem $0,4$ stupně, což z ní nedělá robustní metodu.

Pro techniku *Houghovy transformace* byly empiricky nalezeny tyto ideální hodnoty parametrů:

- **minimální délka úsečky** – $9,99$ % šířky obrazu,
- **maximální velikost díry v úsečce (mezery)** – $2,7$ % šířky obrazu,
- **prahová hodnota akumulátorů** – 28.

Metoda *ohraničujících rámečků* je bezparametrická, nicméně jsou odfiltrovány ty *rámečky*, které jsou příliš malé. Empiricky bylo dosaženo ideální minimální velikosti 100×100, natočení *rámečků* menší velikosti není bráno v potaz.

Na obrázku 10.2 lze vidět grafický výstup obou metod, kde v levém obrázku 10.2a jsou znázorněny červené linie a jejich natočení a v pravém obrázku 10.2b jsou modře vyznačeny natočené *ohraničující rámečky*.



(a) Houghova transformace. (b) Ohraničující rámečky.

Obrázek 10.2: Grafický výstup metod k detekci úhlu natočení obrazu.

10.2 Odšumění obrazu

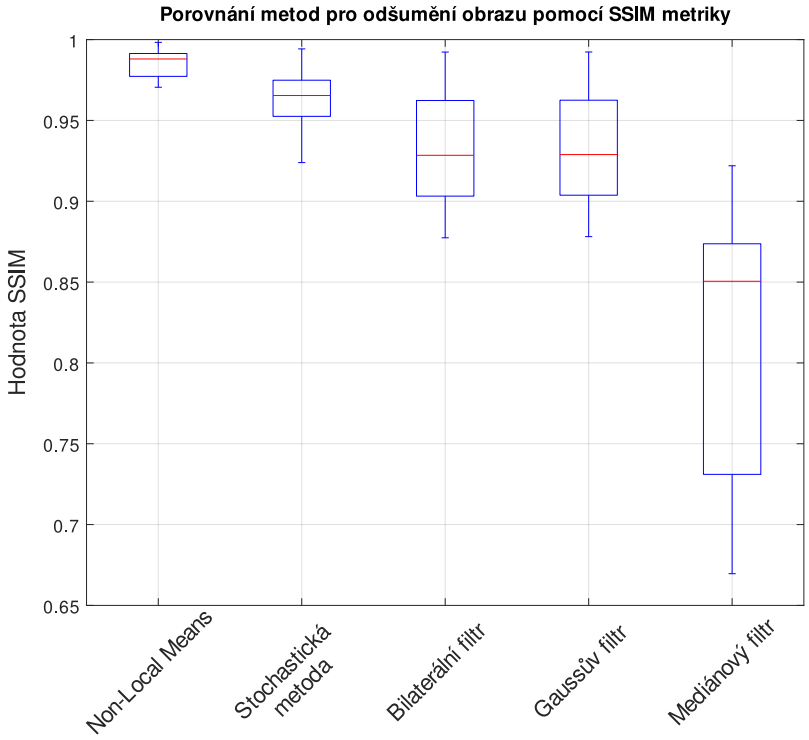
Pro odšumění obrazu byla implementována *stochastická* metoda a použit algoritmus *Non-Local Means Denoising* (obsažen v knihovně **OpenCV**). Autoři metody *stochastického odšumění obrazu* [16] provedli v rámci prezentovaných výsledků také porovnání navrženého algoritmu s ostatními používanými metodami – *Non-Local Means*, *Bilateral filtering*, *Block matching* a *Total variation*.

Na základě zmíněné publikace byly použity stejné či podobné vstupní parametry metod, avšak s tím rozdílem, že metody *Block matching* a *Total variaton* byly nahrazeny *Gaussovým filtrem* a *mediánovým filtrem*.

Z testovací sady dat bylo vybráno 10 viditelně zašuměných dokumentů při procesu skenování (z [19]). Každý dokument byl manuálně odšuměn a označen jako „čistý“ (referenční) dokument.

Graf na obrázku 10.3 s přehledem hodnot v tabulce 10.2 znázorňuje porovnání zmíněných metod, kde metrikou je hodnota *SSIM* (viz 5.10.5), která udává hodnotu podobnosti referenčního dokumentu vůči zašuměnému dokumentu, respektive podobnost jejich obrazu. Čím je hodnota *SSIM* blíže k 1, tím více jsou si dokumenty podobné. Z toho vyplývá, že nejlépe si vedla metoda *Non-Local Means*, která jako jediná dosahuje průměrné hodnoty $0,984$ s směrodatnou odchylkou $0,015$. O něco hůře je na tom *stochastická metoda*, nicméně její střední hodnota je $0,963$ s vyšší odchylkou $0,019$. Zatímco *bilaterální filtr* a *Gaussův filtr* dosahují podobných výsledků $0,947$ a $0,932$ se stejnou odchylkou $0,037$, *mediánový filtr* skončil s nejhorším výsledkem, jeho střední hodnota $0,864$ nedosáhla ani na minimální hodnoty ostatních metod, navíc hodnoty jsou výrazně rozptýlené.

Na základě zmíněné publikace a empirického měření byly nastaveny ná-



Obrázek 10.3: Porovnání metod pro odšumění obrazu.

	SSIM Index				
Metoda	Medián	Průměr	Odchylka	Min.	Max.
Non-Local Means	**0,992**	**0,984**	**0,015**	**0,948**	**0,998**
Stochastická	0,967	0,963	0,019	0,923	0,994
Bilaterální	0,947	0,931	0,037	0,877	0,992
Gaussova	0,947	0,932	0,037	0,878	0,992
Medián	0,864	0,813	0,086	0,669	0,921

Tabulka 10.2: Přehled SSIM hodnot technik pro odšumění obrazu.

sledující hodnoty vstupních parametrů:

- **Stochastické odšumění** – počet náhodných cest 25 a standardní odchylka $0,3$,
- **Non-Local Means** – parametr $h = 15$, velikost okénka pro výpočet vah 17 a velikost okénka pro výpočet lokálního průměr 28,
- **bilaterální filtr** – velikost průměru 7 a parametr $\sigma_{\text{color}} = 1$,
- **Gaussův filtr** – velikost 21×21 a parametr $\sigma_x = 0,3$,

- **mediánový filtr** – velikost 21.

Na obrázku 10.6 lze vidět grafické porovnání výsledných odšuměných obrazků pomocí metody *Non-Local Mean* (obrázek 10.4b) a *stochastického odšumění* (obrázek 10.4c) se vstupním zašuměným obrázkem na obrázku 10.4a.

(a) Zašuměný obraz.

(b) Odšumění Non-Local Means.

(c) Stochastické odšumění.

Obrázek 10.4: Grafické porovnání metod pro odšumění obrazu.

10.3 Binarizace obrazu

K vyhodnocení nejvhodnější techniky pro prahování (binarizaci) byl použit datový soubor **DIBCO**[1], který obsahuje 20 obrázků s příslušným referenčním obrázkem k otestování technik.

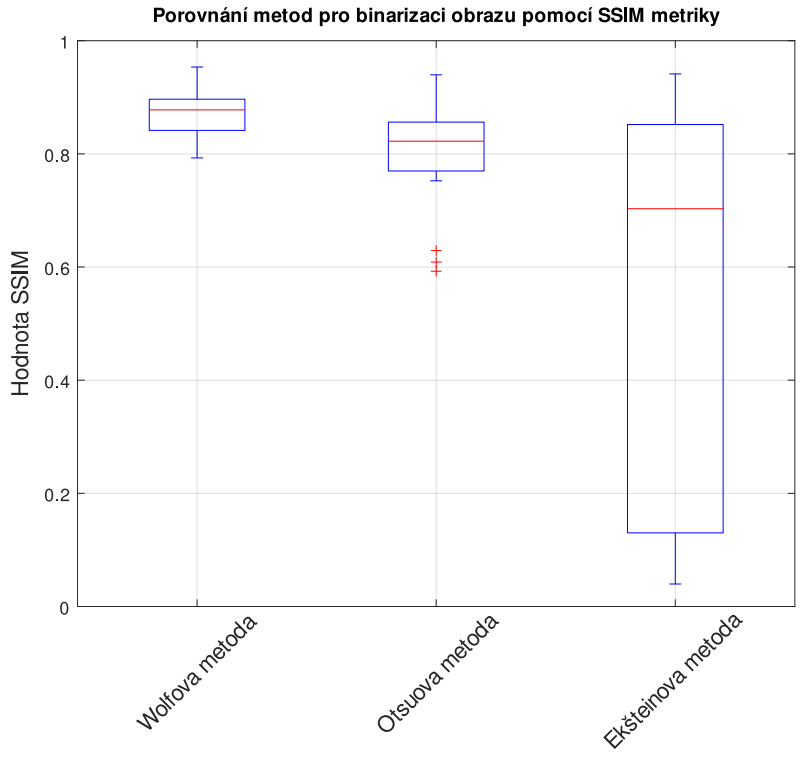
Otestovány byly všechny metody zmíněné v analýze (viz 5.4). *Otsuova* metoda i *Wolfova* metoda je součástí knihovny **OpenCV**, *Ekšteinova* metoda musela být implementována.

[1]Zdroj `https://vc.ee.duth.gr/dibco2019/`

	SSIM Index				
Metoda	Medián	Průměr	Odchylka	Min.	Max.
Wolfova	**0,882**	**0,871**	**0,041**	**0,792**	**0,953**
Otsuova	0,822	0,801	0,092	0,592	0,939
Ekšteinova	0,733	0,523	0,345	0,039	0,941

Tabulka 10.3: Přehled SSIM hodnot technik pro binarizaci obrazu.

Z grafu na obrázku 10.5 a příslušnými hodnotami v tabulce 10.3 je zřejmé, že nejlépe je na tom *Wolfova* metoda, jejíž binarizované obrázky se nejvíce shodovaly s referenčním obrázkem se střední *SSIM* hodnotou $0,871$. Podobně je na tom *Otsuova* metoda s hodnotu těsně nad 0.801, která ale měla problém se třemi obrazy (proto také větší odchylka). Nejhůře skončila *Ekšteinova* metoda s rozptýlenými hodnotami téměř po celém intervalu a střední *SSIM* hodnotou $0,523$, avšak polovina výsledných hodnot je $\geq 0,733$



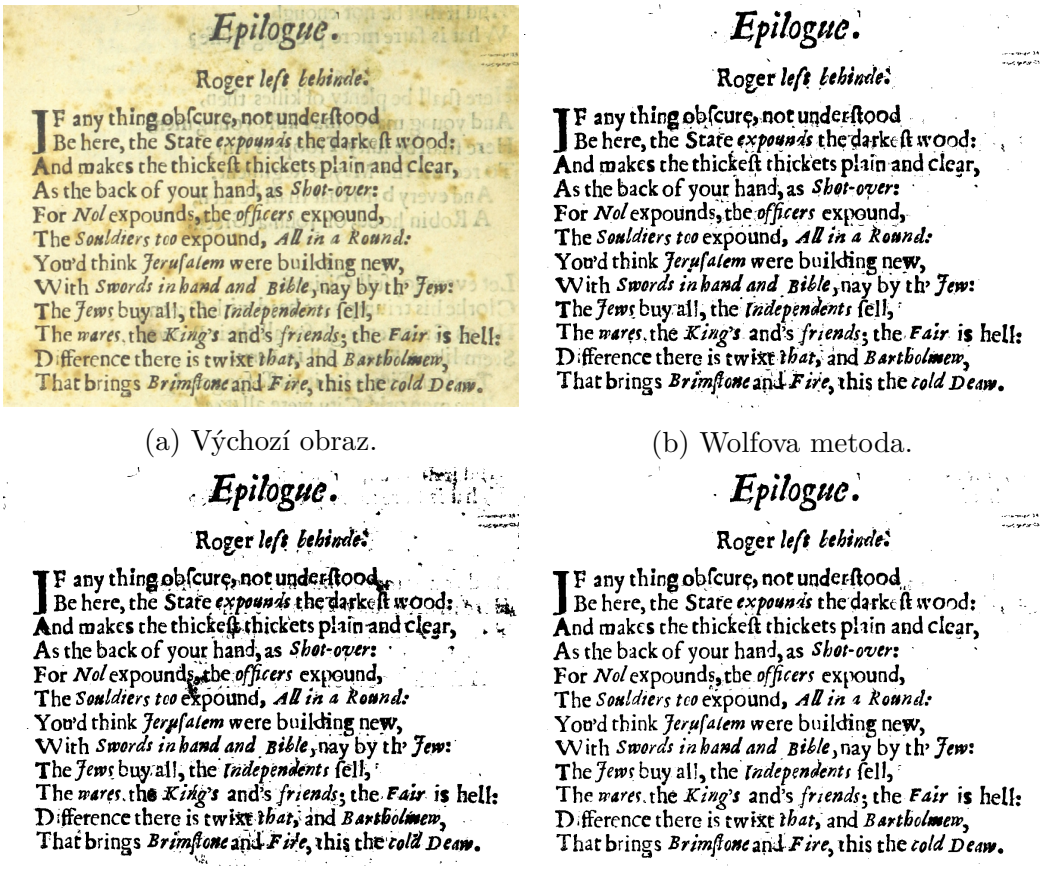
Obrázek 10.5: Porovnání výsledků metod pro binarizaci obrazu.

U *Wolfovy* metody byly empiricky nalezeny následující ideální parametry:

- **velikost lokálního okolí** – 39,
- **parametr K** – $0,5$.

Je vhodné zmínit, že uvedený datový soubor k otestování binarizovaných obrazků obsahuje velice staré snímky textu s netradičním pozadím a šumem, což není v dnešní době u skenovaných dokumentů vůbec běžné.

Na obrázku 10.6 je možné vidět příklad obrázků, kde všechny zmíněné metody binarizovaly obraz víceméně správně.



(a) Výchozí obraz.

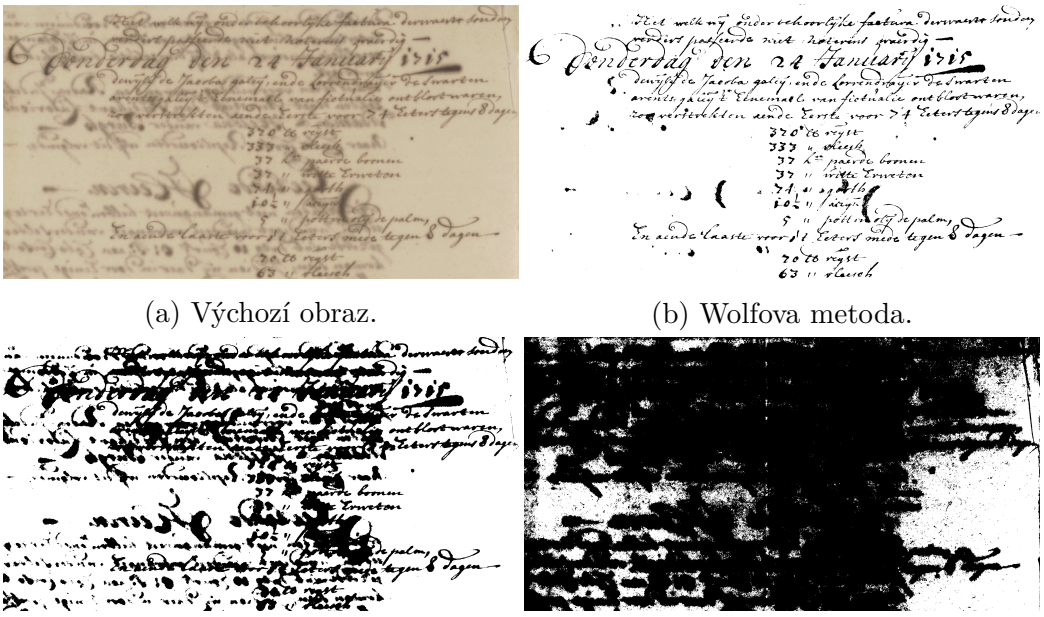
(b) Wolfova metoda.

(c) Otsuova metoda.

(d) Ekšteinova metoda.

Obrázek 10.6: Grafické porovnání metod pro binarizaci dokumentu.

Naopak na obr. 10.7 lze vidět příklad obrázku z datového souboru, u kterého uspěla pouze *Wolfova* metoda, *Otsuova* a *Ekšteinova* metoda si s výchozím obrázkem neporadila. Nicméně jedná se o bezparametrické metody, u *Wolfovy* metody je stále potřeba zvolit vhodné vstupní parametry.



(a) Výchozí obraz. (b) Wolfova metoda.

(c) Otsuova metoda. (d) Ekšteinova metoda.

Obrázek 10.7: Grafické porovnání metod pro binarizaci dokumentu.

10.4 Hledání vzoru

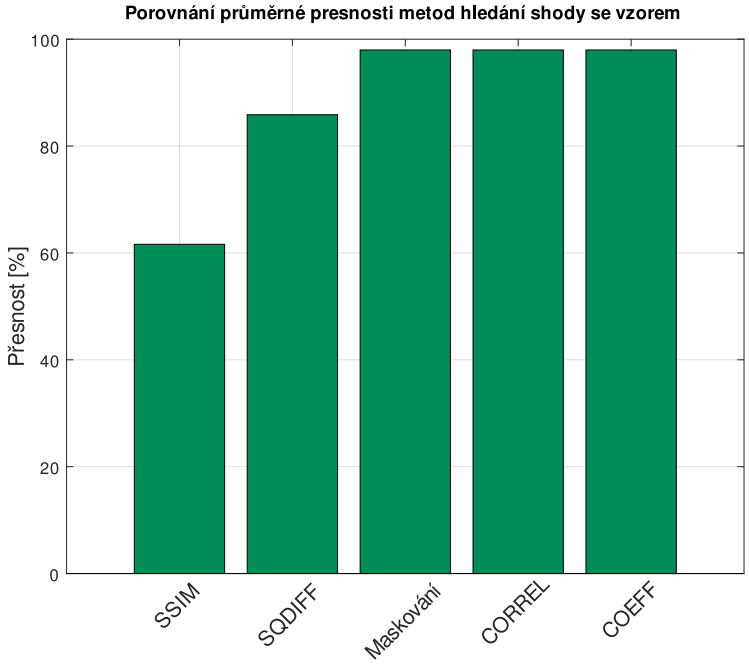
K vyhodnocení úspěšnosti nalezení nejvhodnější šablony byly použity techniky popsané v analýze (viz 5.10) založené na výpočtu podobnosti obrazů, respektive jejich oblastí v rámci šablony. Jedná se o tyto výpočty podobnosti; *maskování*, *normalizovaná suma rozdílů čtverců*, *normalizovaná vzájemná korelace*, *normalizované koeficienty vzájemné korelace* a *index strukturální podobnosti* (*SSIM*).

Před samotným procesem nalezení vhodné šablony byly aplikovány techniky pro předzpracování obrázku – *normalizace* rozlišení (na velikost výchozího dokumentu), *binarizace*, a *detekce a korekce natočení*. Na základě již provedených testů byly použity ideální hodnoty vstupních parametrů konkrétních algoritmů.

Co se týče metod, jako je *normalizovaná suma rozdílů čtverců*, *normalizovaná vzájemná korelace* a *normalizované koeficienty vzájemné korelace*, byla k jejich realizaci použita funkce `matchTemplate` knihovny **OpenCV**, načež metody *SSIM* a *maskování* byly implementovány.

Graf na obrázku 10.8 s přehledem hodnot v tabulce 10.4 zobrazuje střední hodnotu přesnosti v procentech. Nejvyšší přesnosti $97,9$ % dosahují metody *maskování*, *normalizovaná vzájemná korelace* (*CORREL*) a *normalizované koeficienty vzájemné korelace* (*COEFF*). O něco menší přesnosti $85,8$ %

dosahuje metoda *normalizované sumy rozdílů čtverců* (*SQDIFF*). Naopak nejnižší přesnosti 61, 6 % nabývá metoda *SSIM* (*strukturální podobnost*).

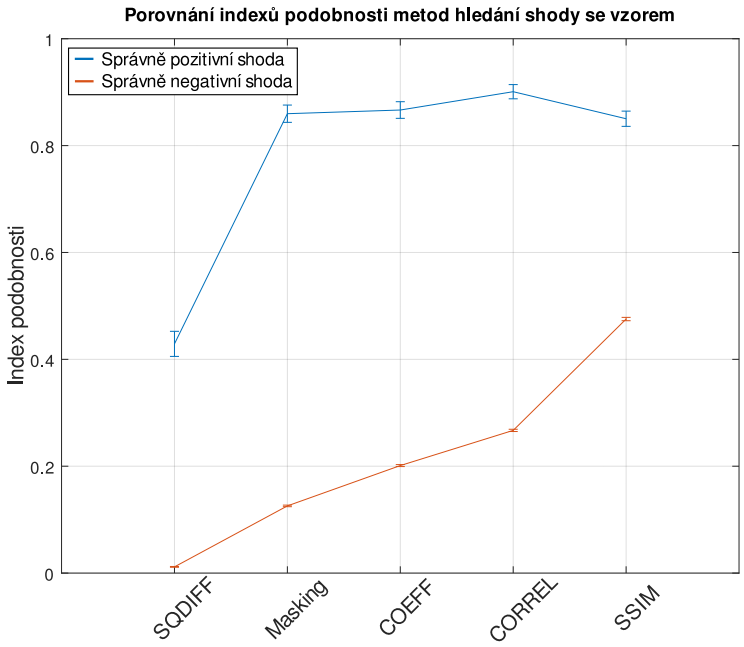


Obrázek 10.8: Průměrná přesnost metod hledání shody se vzorem.

	Shoda	**Skóre podobnosti**				
Metoda	Přesnost	Medián	Průměr	Odchylka	Min.	Max.
SSIM	61,6%	0,883	0,902	0,049	0,844	1
SQDIFF	85,8%	0,403	0,434	0,225	0	0,961
Maskování	97,9%	0,899	0,863	0,138	0,305	1
CORREL	**97,9%**	0,934	0,908	0,097	0,543	1
COEFF	97,9%	0,903	0,873	0,123	0,382	1

Tabulka 10.4: Přehled hodnot přesností technik hledání shody se vzorem.

S tím koresponduje graf na obrázku 10.9, který zobrazuje střední hodnoty indexu podobnosti se střední chybou. Modrá křivka reprezentuje hodnoty podobnosti správně nalezeného vzoru pro konkrétní obrázek, hodnoty na červené křivce patří obrázku odlišném od hledaného vzoru. Ideální je metoda, která má rozdíl mezi hodnotami na červené a modré křivce maximální možný, tj. hodnoty 1 pro modrou křivku s hodnotami 0 pro červenou křivku. Nejvyšší takový rozdíl dosahuje *maskovací* metoda, která i podle grafu na předchozím obrázku 10.8 dosahuje nejvyšší přesnosti.



Obrázek 10.9: Průměrné hodnoty podobnosti metod hledání shody se vzorem.

10.5 OCR

Jako jeden z posledních testů se zabývá vyhodnocením přesnosti OCR systému při použití různých technik počítačového vidění pro předzpracování vstupního obrázku s textovými oblastmi podle definovaných šablon. Jak již bylo zmíněno na začátku kapitoly, testovací sada dat obsahuje 292 dokumentů, avšak 78 z nich tvoří formulářové (výchozí) dokumenty. Ve výsledků bylo otestováno 214 dokumentů, z toho 78 bez výrazného natočení obrázku a 136 s různými úhly natočení obrázku v intervalu od -5 do 5 stupňů.

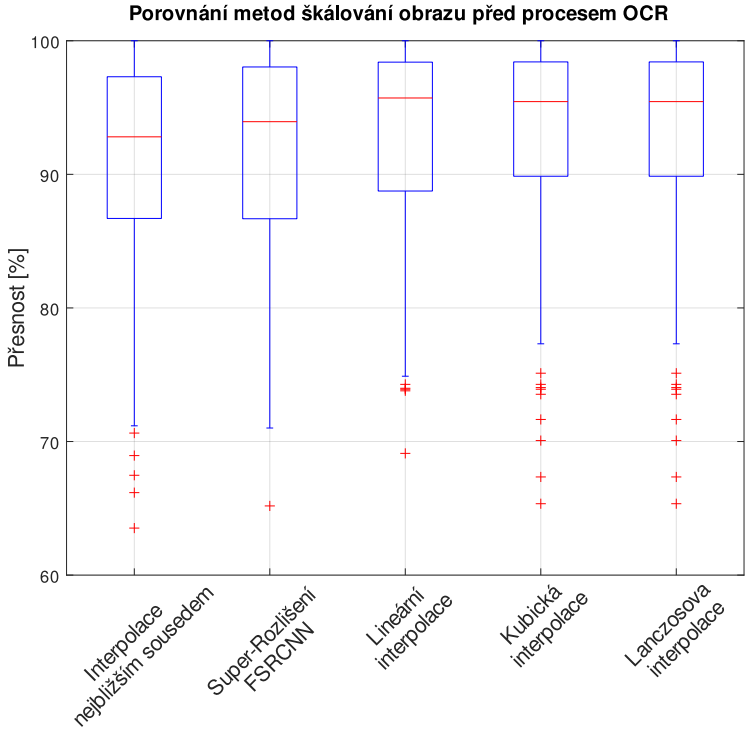
Před samotným procesem rozpoznání znaků v obrázku je aplikována řada technik k vylepšení obrázku s cílem dosáhnout vyšší přesnosti OCR systému. Dále v této sekci jsou popsány výsledné hodnoty úspěšnosti OCR s využitím různých technik předzpracování obrazu, popř. jejich kombinacemi, jenž byly představeny v analýze (viz kapitola 5 a 6). Díky předchozím testům byly vybrány pouze nejúspěšnější algoritmy s konkrétními vstupními parametry.

Jednotlivé definované šablony ke každému dokumentu obsahují ručně přepsaný text z oblastí, kterým je nastavena akce `OCR`. Tento referenční text je následně porovnán s výstupem OCR systému a jako metrika podobnosti bloku textu (slov) je zvolena *normalizovaná Levenshteinova vzdálenost* [45].

Jelikož jsou dvě třetiny dokumentů různě natočené, každá testovaná metoda nejdříve aplikuje korekci natočení obrazu pomocí *Houghovy transfor-*

mace, jež byla vyhodnocena jako nejpřesnější (viz test 10.1). Před samotným porovnáním možných technik předzpracování obrazu jsou otestovány metody zmíněné v analýze (viz sekce 5.8) pro zvětšení velikosti obrazu – *interpolace nejbližším sousedem*, *lineární interpolace*, *kubická interpolace*, *Lanczosova inerpolace* a *Super-rozlišení* s modelem *FSRCNN*.

Jako metrika je zvolena hodnota úspěšnosti OCR systému na úrovni celých slov. Jak je možné vidět v grafu na obrázku 10.10 s přehledem hodnot v tabulkce 10.5, střední hodnota přesnosti všech metod je nad 90 %, respektive 90, 9 % (nejnižší hodnota odpovídající *interpolaci nejbližším sousedem*). O něco vyšší hodnoty přesnosti 91, 9 % dosahuje metoda *Super-rozlišení* s natrénovaným modelem *FSRCNN*. Podobně je na tom *kubická interpolace* s úspěšností 92, 8 %. Naopak *lineární interpolace* a *Lanczosova interpolace* dosahují nejvyšší přesnosti – 92, 9 % s rozdílem v odchylce o 0, 4 %.



Obrázek 10.10: Porovnání metod škálování obrazu před procesem OCR.

Na obrázku 10.11 lze vidět graf, který porovnává úspěšnosti OCR systému při různém předzpracování obrázku s příslušným přehledem hodnot v tabulce 10.6. I přesto, že nebylo aplikováno žádné předzpracování obrázku, úspěšnost dosáhla 74, 8 %, nicméně odchylka je nejvyšší, a to 15, 5 %. Jak již bylo zmíněno, veškeré metody aplikovaly korekci natočení obrazu, nicméně pro porovnání je v grafu také znázorněno samotné natočení, které zlepšilo

	Přesnost OCR systému [%]				
Metoda	Medián	Průměr	Odchylka	Min.	Max.
Nejbližší soused	92,8	90,9	7,9	63,5	100
Super-rozlišení	93,9	91,9	7,5	65,1	100
Lineární interpolace	**95,7**	**92,9**	**7,0**	**69,1**	**100**
Kubická interpolace	95,2	92,8	7,2	68,2	100
Lanczosova interpolace	95,4	92,9	7,4	65,3	100

Tabulka 10.5: Přehled přesností OCR pro různé techniky škálování obrazu.

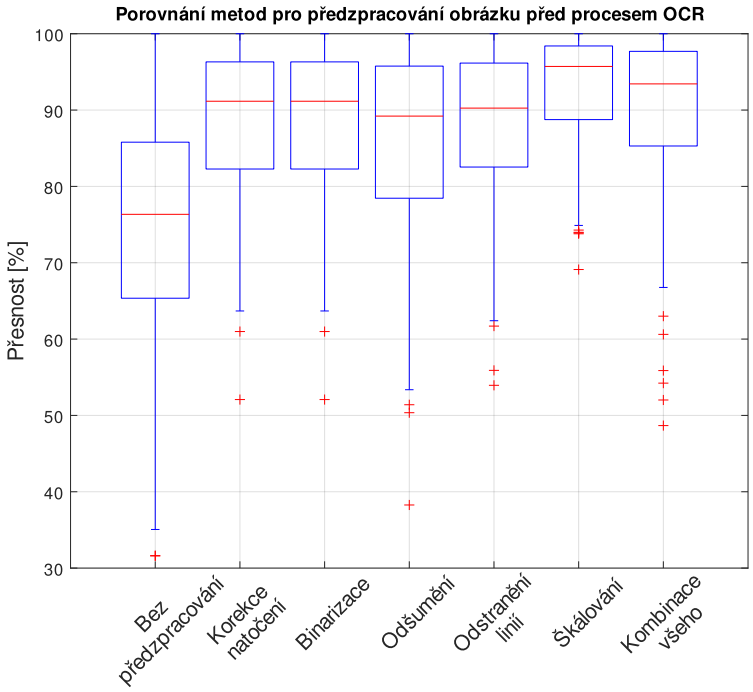
přesnost OCR na $88,2$ %. Dále pak binarize obrázku, která nepřinesla žádně zlepšení ani zhoršení přesnosti. Na rozdíl od odšumění a odstranění čar, které naopak přesnost zhoršily – $86,2$ % a $87,9$ %. Nejvyšší přesnosti $92,9$ % bylo dosaženo pouze po zvětšení obrazu (*lineární interpolací* z předešlého testování), a to alespoň na rozlišení odpovídající naskenovanému A4 dokumentu při 300 dpi. Pro zajímavost byla otestována kombinace všech zmíněných metod, jejíž přesnost je $90,1$ %.

	Přesnost OCR systému [%]				
Metoda	Medián	Průměr	Odchylka	Min.	Max.
Bez předzpracování	76,3	74,8	15,5	31,5	100
Korekce natočení	91,1	88,2	10,0	52,0	100
Odšumění	89,2	86,2	11,4	38,2	100
Binarizace	91,1	88,2	10,0	52,0	100
Odstranění čar	90,2	87,9	10,2	53,9	100
Škálování	**95,7**	**92,9**	**7,0**	**69,1**	**100**
Kombinace všeho	93,4	90,1	10,0	48,6	100

Tabulka 10.6: Přehled přesností OCR pro různé techniky předzpracování obrázku.

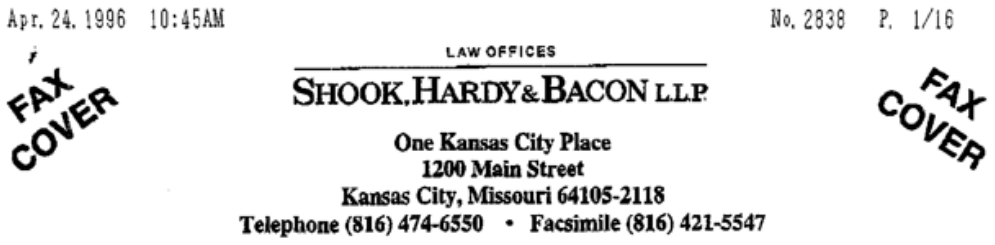
Jelikož **Tessract** od verze **4**, na rozdíl od verze **3**, používá hlubokou neuronovou síť LSTM, jejíž model je naučen přímo ze vstupního obrázku, není dosaženo vyšší přesnosti při využití binarizovaného obrazu.

I přes korekci natočení obrazu mohou nastat situace, kdy v dokumentu



Obrázek 10.11: Porovnání metod předzpracování obrazu před procesem OCR.

zůstanou textové oblasti, které jsou různě natočené, jak je vidět na obrázku 10.12. Celkově dokument není natočen, ale obsahuje dva textové bloky „FAX COVER“, které jsou stále natočené. Řešením by byla korekce jak na globální úrovni (celý dokument), tak i v rámci lokálních oblastí (např. nalezení *kontur* a korekce jejich natočení).



Obrázek 10.12: Příklad lokálního natočení textu.

11 Závěr

Cílem práce bylo seznámit se s technikami počítačového vidění a OCR použitelnými k ověření shody analyzovaného dokumentu se šablonou a následné extrakci významných oblastí podle příslušné šablony. Na základě toho navrhnout a implementovat jak modul počítačového vidění a práce s dokumenty (obrázky), tak i software pro samotnou tvorbu šablon.

V rámci analýzy byly důkladně prostudovány a popsány potenciální techniky a nástroje pro předzpracování obrázku a ověření shody dokumentu se vzorem (šablonou). Zároveň byly prostudovány dostupné OCR systémy pro rozpoznávání textu z extrahovaných zájmových oblastí.

Na základě analýzy byl navržen a implementován modul počítačového vidění a práce s dokumenty realizující uvedené techniky v programovacím jazyce C++ s využitím knihovny **OpenCV**, **Tesseract-OCR** a **PoDoFo**. Software pro tvorbu šablon byl taktéž implementován v programovacím jazyce C++ s využitím knihovny **Qt5** pro grafické uživatelské rozhraní. Obě implementovaná řešení jsou řádně popsána a zdokumentována. Součástí je konfigurační soubor s možností změny použitých technik a jejich vstupních parametrů. Nad rámec této práce byla vytvořena funkcionalita pro vygenerování souboru PDF s textovou vrstvou překrývající výchozí obrázek, takže s ním lze pracovat jako s textovým dokumentem.

K ověření funkčnosti a stability implementovaných technik posloužil vytvořený testovací soubor dat. Každý dokument obsahuje dvě kopie s různou rotací obrázku a jednu kopii jako výchozí vzor (dokument s odstraněnými formulářovými údaji) s příslušnou vytvořenou šablonou. Pro algoritmy předzpracování obrázku byly empiricky nalezené nejvhodnější vstupní parametry nebo byly převzaty z dostupných publikací; nejrobustnější techniky byly po sadě provedených testů zvoleny jako výchozí pro následné vyhodnocení přesnosti nalezení šablony a OCR. Zvolenými technikami jsou *Houghova transformace* k detekci úhlu natočení, *Non-Local Means* k odšumění obrazu a *Wolfova* metoda pro prahování (binarizaci) obrázku.

K nalezení ideální šablony byla zvolena technika *vzájemné korelace*, která dosahuje přesnosti **97,9 %**. OCR systém dosahuje přesnosti **92,9 %** s využitím *korekce natočení obrázku* a zvětšením obrázku *lineární interpolací* ve fázi předzpracování. S přihlédnutím ke kvalitě dokumentů v testovací sadě (odpovídající cca 90 dpi pro rozměr A4) lze konstatovat, že přesnost nalezení šablony i OCR jsou uspokojivé.

Literatura

[1] *PDF Reference, sixth edition: Adobe Portable Document Format version 1.7.* Adobe® Systems Incorporated, 2007. Dostupné z: https://www.adobe.com/devnet/pdf/pdf_reference_archive.html.

[2] Acharya, T. – Ray, A. K. *Image Processing: Principles and Applications.* John Wiley & Sons, Inc., 2005. ISBN 13 978-0-471-71998-4.

[3] Adrian Kaehler, G. B. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library.* O'Reilly Media, Inc., 2016. ISBN 978-1-491-93799-0.

[4] Buades, A. – Coll, B. – Morel, J. M. Non-Local Means Denoising. *Image Processing On Line.* 2011, 1, s. 208–212. Dostupné z: https://doi.org/10.5201/ipol.2011.bcm_nlm.

[5] Burger, W. – Burge, M. J. *Principles of Digital Image Processing: Core Algorithms.* Springer-Verlag London, 2009. ISBN 978-1-84800-194-7.

[6] Canny, J. A Computational Approach to Edge Detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence.* 1986, PAMI-8, 6, s. 679–698. Dostupné z: https://doi.org/10.1109/TPAMI.1986.4767851.

[7] Cao, H. – Natarajan, P. *Machine-Printed Character Recognition.* Springer-Verlag London, 2014. ISBN 978-0-85729-858-4.

[8] Chan, T. F. – Vese, L. A. Active contours without edges. *IEEE Transactions on Image Processing.* 2001, 10, 2, s. 266–277. Dostupné z: https://doi.org/10.1109/83.902291.

[9] Chenyang Xu – Prince, J. L. Snakes, shapes, and gradient vector flow. *IEEE Transactions on Image Processing.* 1998, 7, 3, s. 359–369. Dostupné z: https://doi.org/10.1109/83.661186.

[10] Company, Q. *Qt Website* [online]. Qt Company, 2020. [cit. 2020/04/22]. Qt Framework. Dostupné z: https://www.qt.io/.

[11] Dhiman, S. – Singh, A. Tesseract vs gocr a comparative study. *International Journal of Recent Technology and Engineering.* 2013, 2, 4, s. 80.

[12] *DLib library* [online]. DLib, 2019. [cit. 2020/04/09]. DLib library website. Dostupné z: http://dlib.net/.

[13] Dong, C. et al. Learning a Deep Convolutional Network for Image Super-Resolution. *Springer International Publishing*. 2014, s. 184–199. Dostupné z: https://doi.org/10.1007/978-3-319-10593-2_13.

[14] Dong, C. – Loy, C. C. – Tang, X. Accelerating the Super-Resolution Convolutional Neural Network. *CoRR*. 2016, abs/1608.00367. Dostupné z: http://arxiv.org/abs/1608.00367.

[15] Ekštein, K. Simple and Efficient Method of Low-Contrast Grayscale Image Binarization. *Springer International Publishing*. 2016, s. 142–150. Dostupné z: https://doi.org/10.1007/978-3-319-46418-3_13.

[16] Estrada, F. – Fleet, D. – Jepson, A. Stochastic Image Denoising. *Proceedings of the British Machine Vision Conference*. 2009, s. 117.1–117.11. Dostupné z: https://doi.org/10.5244/C.23.117.

[17] Galamhos, C. – Matas, J. – Kittler, J. Progressive probabilistic Hough transform for line detection. *Proceedings. 1999 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (Cat. No PR00149)*. 1999, 1, s. 554–560 Vol. 1. Dostupné z: https://doi.org/10.1109/CVPR.1999.786993.

[18] Gari, A. et al. Skew detection and correction based on Hough transform and Harris corners. *2017 International Conference on Wireless Technologies, Embedded and Intelligent Systems (WITS)*. 2017, s. 1–4. Dostupné z: https://doi.org/10.1109/WITS.2017.7934619.

[19] Harley, A. W. – Ufkes, A. – Derpanis, K. G. Evaluation of Deep Convolutional Nets for Document Image Classification and Retrieval. *International Conference on Document Analysis and Recognition (ICDAR)*. 2015. Dostupné z: https://www.cs.ryerson.ca/~aharley/icdar15/.

[20] Hisham, M. B. et al. Template Matching using Sum of Squared Difference and Normalized Cross Correlation. *2015 IEEE Student Conference on Research and Development (SCOReD)*. 2015, s. 100–104.

[21] Hochreiter, S. – Schmidhuber, J. Long Short-Term Memory. *Neural Computation*. 1997, 9, 8, s. 1735–1780. Dostupné z: https://doi.org/10.1162/neco.1997.9.8.1735.

[22] Illingworth, J. – Kittler, J. The Adaptive Hough Transform. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 1987, PAMI-9, 5, s. 690–698. Dostupné z: https://doi.org/10.1109/TPAMI.1987.4767964.

[23] *JavaFX Website* [online]. Gluon, 2020. [cit. 2020/04/22]. Java JavaFX Library. Dostupné z: https://gluonhq.com/products/javafx/.

[24] Kass, M. – Witkin, A. – Terzopoulos, D. Snakes: Active contour models. *International Journal of Computer Vision*. 1988, 1, s. 321–331. Dostupné z: https://doi.org/10.1007/BF00133570.

[25] Keys, R. Cubic convolution interpolation for digital image processing. *IEEE Transactions on Acoustics, Speech, and Signal Processing*. 1981, 29, 6, s. 1153–1160. Dostupné z: https://doi.org/10.1109/TASSP.1981.1163711.

[26] King, D. E. Dlib-ml: A Machine Learning Toolkit. *Journal of Machine Learning Research*. 2009, 10, s. 1755–1758. Dostupné z: http://jmlr.csail.mit.edu/papers/volume10/king09a/king09a.pdf.

[27] Kumar, G. – Bhatia, P. K. A Detailed Review of Feature Extraction in Image Processing Systems. *2014 Fourth International Conference on Advanced Computing Communication Technologies*. 2014, s. 5–12. Dostupné z: https://doi.org/10.1109/ACCT.2014.74.

[28] Lai, W. et al. Fast and Accurate Image Super-Resolution with Deep Laplacian Pyramid Networks. *CoRR*. 2017, abs/1710.01992. Dostupné z: http://arxiv.org/abs/1710.01992.

[29] Lashkov, I. et al. Driver Dangerous State Detection Based on OpenCV Dlib Libraries Using Mobile Video Processing. *2019 IEEE International Conference on Computational Science and Engineering (CSE) and IEEE International Conference on Embedded and Ubiquitous Computing (EUC)*. 2019, s. 74–79. Dostupné z: https://doi.org/10.1109/CSE/EUC.2019.00024.

[30] LeCun, Y. – Kavukcuoglu, K. – Farabet, C. Convolutional networks and applications in vision. *Proceedings of 2010 IEEE International Symposium on Circuits and Systems*. 2010, s. 253–256. Dostupné z: https://doi.org/10.1109/ISCAS.2010.5537907.

[31] Lim, B. et al. Enhanced Deep Residual Networks for Single Image Super-Resolution. *CoRR*. 2017, abs/1707.02921. Dostupné z: http://arxiv.org/abs/1707.02921.

[32] Obinata, G. – Dutta, A. *Vision Systems: Segmentation and Pattern Recognition*. I-Tech Education and Publishing, 2007. ISBN 978-3-902613-05-9.

[33] *OpenCV library* [online]. OpenCV team, 2020. [cit. 2020/04/09]. Open Computer Vison library website. Dostupné z: https://opencv.org/.

[34] Otsu, N. A Threshold Selection Method from Gray-Level Histograms. *IEEE Transactions on Systems, Man, and Cybernetics*. 1979, 9, 1, s. 62–66. Dostupné z: https://doi.org/10.1109/TSMC.1979.4310076.

[35] Parker, J. A. – Kenyon, R. V. – Troxel, D. E. Comparison of Interpolating Methods for Image Resampling. *IEEE Transactions on Medical Imaging*. 1983, 2, 1, s. 31–39. Dostupné z: https://doi.org/10.1109/TMI.1983.4307610.

[36] Rajesh, K. M. – Naveenkumar, M. A robust method for face recognition and face emotion detection system using support vector machines. *2016 International Conference on Electrical, Electronics, Communication, Computer and Optimization Techniques (ICEECCOT)*. 2016, s. 1–5. Dostupné z: https://doi.org/10.1109/ICEECCOT.2016.7955175.

[37] Ritter, G. X. – Wilson, J. N. *Computer Vision Algorithms in Image Algebra*. CRC Press LLC, 2000. ISBN 0-8493-0075-4.

[38] Sauvola, J. et al. Adaptive document binarization. *Proceedings of the Fourth International Conference on Document Analysis and Recognition*. 1997, 1, s. 147–152 vol.1. Dostupné z: https://doi.org/10.1109/ICDAR.1997.619831.

[39] Shi, W. et al. Real-Time Single Image and Video Super-Resolution Using an Efficient Sub-Pixel Convolutional Neural Network. *CoRR*. 2016, abs/1609.05158. Dostupné z: http://arxiv.org/abs/1609.05158.

[40] Singh, R. – Dubey, A. K. – Kapoor, R. Up sampling of an image using convolution method. *2017 6th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions) (ICRITO)*. 2017, s. 636–639. Dostupné z: https://doi.org/10.1109/ICRITO.2017.8342505.

[41] Szeliski, R. *Image Processing: Principles and Applications*. Springer Science & Business Media, 2010. ISBN 978-1-84882-935-0.

[42] *Tesseract-OCR Website* [online]. GitHub Pages, 2020. [cit. 2020/21/04]. Optical Character Recognition Library. Dostupné z: https://tesseract-ocr.github.io/.

[43] Wei, S. – Lai, S. Fast Template Matching Based on Normalized Cross Correlation With Adaptive Multilevel Winner Update. *IEEE Transactions on Image Processing*. 2008, 17, 11, s. 2227–2235. Dostupné z: https://doi.org/10.1109/TIP.2008.2004615.

[44] Wolf, C. – Jolion, J. . – Chassaing, F. Text localization, enhancement and binarization in multimedia documents. *Object recognition supported by user interaction for service robots*. 2002, 2, s. 1037–1040 vol.2. Dostupné z: https://doi.org/10.1109/ICPR.2002.1048482.

[45] Yujian, L. – Bo, L. A Normalized Levenshtein Distance Metric. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 2007, 29, 6, s. 1091–1095. Dostupné z: https://doi.org/10.1109/TPAMI.2007.1078.

[46] Zhou Wang et al. Image quality assessment: from error visibility to structural similarity. *IEEE Transactions on Image Processing*. 2004, 13, 4, s. 600–612. Dostupné z: https://doi.org/10.1109/TIP.2003.819861.

Seznam zkratek

API Application Programming Interface.

BGR Blue Greed Red.

bpp bits per pixel.

dpi Dots per inch.

GPU Graphical Processing Unit.

GUI Graphical User Interface.

HP Hewlett Packard.

ID Identification.

JSON JavaScript Object Notation.

LGPL Lesser General Public License.

LSTM Long Short-Term Memory.

MVC Model-View-Controller.

NLP Natural Language Processing.

OCR Optical Character Recognition.

OOP Object Oriented Programming.

PDF Portable Document Format.

ppi pixel per inch.

QML Qt Modeling Language.

RDMB Relitional Database Management.

REST API Representational State Transfer API.

RGB Red Green Blue.

RNN Recurent Neural Network.

ROI Region of Interest.

RTF Rich Text Format.

SQL Structured Query Language.

SSIM Structural Similarity.

STL Standard Library.

SVM Support Vector Machines.

TBB Thread Building Blocks.

UI User Interface.

UML Unified Modeling Language.

VGSL Variable Graph Specification Language.

WYSIWYG What you see is what you get.

XML Extensible Markup Language.

Seznam obrázků

5.1 Stupnice šedi. 20
5.2 Aditivní míchání barev 21
5.3 Barevný prostor YC_bC_r s konstantní hodnotou Y = 0.5. . . . 21
5.4 Barevný obrázek a jeho příslušný histogram. 22
5.5 Příklad konvoluce 23
5.6 Bimodální histogram s prahem podle Otsua. 27
5.7 Přehled tvarů strukturního elementů. 30
5.8 Výchozí obrázek pro demonstraci morfologických operací. . . 31
5.9 Příklad eroze. 31
5.10 Příklad dilatace. 31
5.11 Příklad otevření. 32
5.12 Příklad uzavření. 32
5.13 Příklad detekce hran. 33
5.14 Příklad zeštíhlení hran. 35
5.15 Prahování pixelů s hysterezí. 36
5.16 Porovnání zvolených parametrů u Cannyho algoritmu. 36
5.17 Zvětšení obrázku pomocí nejbližšího souseda. 40
5.18 Zvětšení obrázku pomocí bilineární metody. 41
5.19 Zvětšení obrázku pomocí bikubické metody. 41
5.20 Zvětšení obrázku pomocí Lanczosovy interpolace. 42
5.21 Zvětšení obrázku pomocí modelu EDSR Super-Resolution CNN. 43
5.22 Reprezentace přímky pomocí θ a ρ a konkrétních souřadnic bodu (x_0,y_0) v kartézské soustavě. 44
5.23 Sinusoidy vygenerované pomocí 3 různých bodů. 45

6.1 Příklady obrázků reálných scén 51
6.2 Blokové schéma OCR systému. 51
6.3 Segmentace dokumentu. 52
6.4 Architektura konvoluční neuronové sítě. 54
6.5 Architektura Tesseract-OCR. 56

7.1 JavaFX architektura 60

8.1 UML diagram tříd pro práci s dokumenty. 70
8.2 UML diagram třídy Image používající ImageProcessing. . . . 71
8.3 UML diagram třídy DocTemplate používající AreaGroup a Area. 81

8.4 UML diagram tříd pro zpracování obrazu. 83
8.5 UML diagram závislostí OCR třídy. 91

9.1 Diagram tříd softwaru pro práci se šablonami. 93
9.2 Příklad případu užití aplikace pro práci se šablonami. 94
9.3 UML digram závislostí aplikace a modulu. 95
9.4 Snímek obrazovky aplikace pro práci se šablonami. 101

10.1 Porovnání přesnosti technik k detekci úhlu natočení obrázku. 103
10.2 Grafický výstup metod k detekci úhlu natočení obrazu. . . . 105
10.3 Porovnání metod pro odšumění obrazu. 106
10.4 Grafické porovnání metod pro odšumění obrazu. 107
10.5 Porovnání výsledků metod pro binarizaci obrazu. 108
10.6 Grafické porovnání metod pro binarizaci dokumentu. 109
10.7 Grafické porovnání metod pro binarizaci dokumentu. 110
10.8 Průměrná přesnost metod hledání shody se vzorem. 111
10.9 Průměrné hodnoty podobnosti metod hledání shody se vzorem.112
10.10Porovnání metod škálování obrazu před procesem OCR. . . 113
10.11Porovnání metod předzpracování obrazu před procesem OCR. 115
10.12Příklad lokálního natočení textu. 115

A.1 Snímek obrazovky aplikace s zvýrazněnými oblastmi. 129
A.2 Oblast v aplikaci s nástroji. 129
A.3 Dialogové okno pro zpracování šablony. 130
A.4 Záložky v postranním panelu aplikace. 131

Seznam tabulek

5.1 Přehled základních barevných hloubek obrazu. 19

10.1 Přehled absolutních chyb uhlů natočení obrázku. 104
10.2 Přehled SSIM hodnot technik pro odšumění obrazu. 106
10.3 Přehled SSIM hodnot technik pro binarizaci obrazu. 108
10.4 Přehled hodnot přesností technik hledání shody se vzorem. . 111
10.5 Přehled přesností OCR pro různé techniky škálování obrazu. 114
10.6 Přehled přesností OCR pro různé techniky předzpracování obrázku. 114

A Uživatelská příručka

V následující uživatelské příručce je podrobněji popsána struktura aplikace, překlad ze zdrojových souborů a manuál k obsluze.

A.1 Struktura projektu

Struktura kořenového adresáře vypadá následovně:

- **src** – zde se nacházejí všechny zdrojové kódy,
- **data** – potřebná data pro chod aplikace,
- **doc** – vygenerovaná dokumentace zdrojového kódu,
- **Poster** – poster ve formátu *.pub* a *.pdf*,
- **text** – text diplomové práce,
- **get_dependecies.sh** – skript, který pomocí příkazu `apt install` stáhne potřebné knihovny,
- **build_opencv4.sh** – skript, který stáhne knihovnu **OpenCV 4.2.0**, přeloží jí a připraví pro překlad projektu (nutný i **git**),
- **build_project.sh** – přeložení projektu a vytvoření spustitelného souboru,
- **readme.txt** – popis projektu.

A.2 Překlad aplikace

Aplikace byla vyvíjena v operačním systému (OS) **Ubuntu 19.10 64-bit** v programovacím jazyce C++17 s kompilátorem **gcc** verze **9.2.1**. Pro překlad aplikace a modulu jsou použity soubory **CMake** s minimální verzí **3.5**.

Před samotnou fází překladu je potřeba stáhnout knihovny a nástroje podle následujícího výčtu: **libpodofo-dev** (**0.9.6**), **OpenCV 4.2.0** (+ **contrib**), **cmake 3.5** (či vyšší), **qtdeclerative5-dev**, **qt5-image-formats-plugins**, **libleptonica-dev**, **libtesseract4**, **libtesseract-dev**, **libtbb-dev**, **build-essential**, **libjsoncpp-dev**, **libcurl4-openssl-dev**, **libconfig++-dev**, **pdftk** a **uuid-dev**.

Veškeré uvedené knihovny jsou ve zmíněné verzi **Ubuntu** dostupné pomocí příkazu `apt install`, kromě knihovny **OpenCV 4.2.0** (to samé platí pro **contrib**). Obě knihovny musí být staženy a společně manuálně přeloženy[1].

Pro překlad jsou připraveny tři skripty, které stáhnou potřebné knihovny, přeloží aplikaci a vytvoří spustitelný soubor *document-template-editor*. Postup spuštění skriptů vypadá takto:

1. spustit skript *get_dependecies.sh*, ten pomocí příkazu `apt install` stáhne potřebné knihovny,

2. spustit skript *build_opencv4.sh*, ten stáhne knihovnu **OpenCV 4.2.0**, zajistí její překlad a zkopíruje potřebné adresáře do adresáře s projektem,

3. spustit skript *build_project.sh*, ten přeloží aplikaci a vytvoří adresář *bin*, kde se nachází spustitelný soubor pojmenovaný *document-template-editor*. Zařídí také zkopírování potřebných souborů pro chod aplikace.

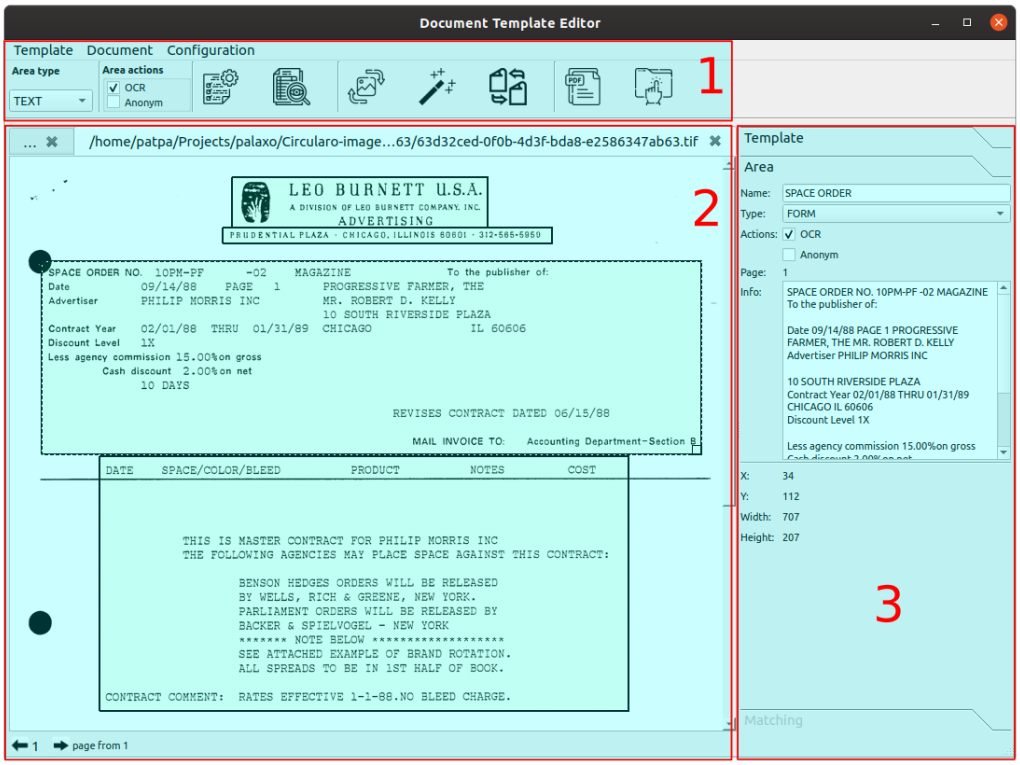
Poslední skript taktéž zkopíruje do adresáře *bin* testovací data, která se skládají ze dvou hlavních adresářů, a to: *templates* (uložené šablony) a *documents* (dokumenty). Nicméně je zkopírován i adresář s konfiguracemi *configs* a adresáře potřebné k evaluačním testům.

A.3 Manuál k obsluze aplikace

Na obr. A.1 lze vidět tři hlavní oblasti aplikace pro vytváření/úpravu šablon a dokumentů. Tyto oblasti jsou dále blíže popsány.

Na obr. A.2 se nachází zvětšená oblast (1) s menu a nástroji. Pomocí tlačítek v menu lze načíst/uložit šablonu či dokument a znovu načíst soubor s konfiguračními parametry, tj. není třeba ukončovat aplikaci při změně konfiguračních parametrů. Šablonu je možné uložit libovolně vybráním cesty (akce `Save as...`) nebo do datového adresáře s ostatními šablonami (akce `Save to DS` – tímto se zkopíruje i dokument). Pod menu se nacházejí (zleva) dvě oblasti s výchozími parametry pro každou nově přidanou oblast, tj. typ a akce.

[1] **OpenCV** `https://github.com/opencv/opencv/releases/tag/4.2.0` a **OpenCV-contrib** `https://github.com/opencv/opencv_contrib/releases/tag/4.2.0`

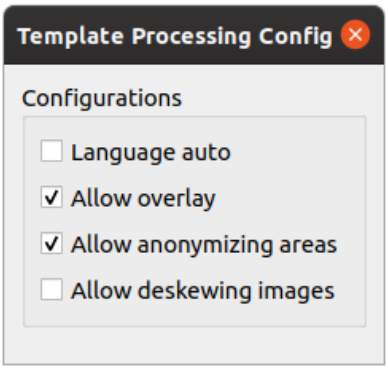


Obrázek A.1: Snímek obrazovky aplikace s zvýrazněnými oblastmi.

Následuje 7 samostatných tlačítek – zpracování šablony, nalezení ideální šablony, detekce a korekce natočení dokumentu (aktuální stránky), odšumění dokumentu (aktuální stránky), načtení původního obrázku pro aktuální stránku (zrušení provedených změn), export dokumentu do formátu PDF (i s textovou vrstvou pokud byla vytvořena OCR procesem) a otevření složky s umístěným dokumentu/šablony. Tlačítka disponují i popiskem (najetím ukazatelem myši na příslušné tlačítko).

Obrázek A.2: Oblast v aplikaci s nástroji.

Tlačítko pro zpracování šablony nabízí ještě kontextový dialog s nastavením parametrů, které povolují či zakazují určité akce, a to: automatická korekce natočení obrázků (kde je definována šablona), anononimyzování oblastí (kde je nastavena příslušná akce), vytvoření textové vrstvy (pro export do formátu PDF) a využití automatické detekce jazyka (při procesu OCR). Tyto akce lze vidět na obrázku A.3.



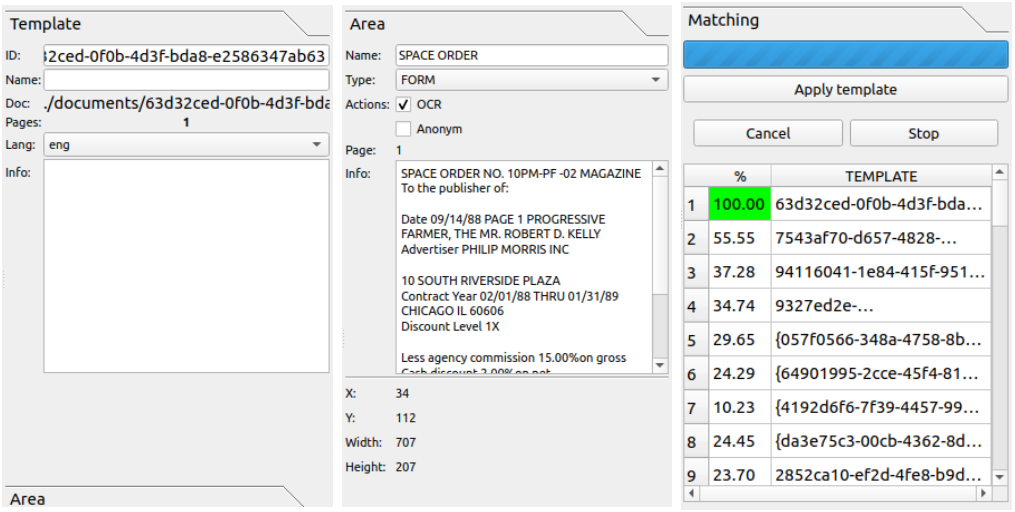
Obrázek A.3: Dialogové okno pro zpracování šablony.

Uprostřed aplikace (pod číslem 2 v obr. A.1) je vykreslen zpracovávaný dokument. Dvojklikem levého tlačítka myši lze přidávat nové zájmové oblasti. Jedním kliknutím levého tlačítka myši na příslušnou oblast dojde k označení této oblasti. V pravém dolním rohu oblasti se nachází obdélník pro škálování. Tlačítkem `delete` je možno označenou oblasti odstranit. Se stiskem tlačítka `Shift` a stiskem a táhnutím myši se lze po plátnu pohybovat. Stiskem tlačítka `Ctrl` a rolováním prostředního tlačítka myši lze vykreslený obrázek přibližovat/oddalovat. Pod plátnem se nacházejí dvě tlačítka pro změnu stránky dokumentu, k tomu je také možné využít klávesové zkratky: `Q` (předchozí stránka) a `E` (následující stránka).

Poslední důležitou oblastí (3) je pravý postranní panel, obsahuje celkem tři záložky, které jsou zachyceny na obrázku A.4. První dvě záložky (`Template` (viz obr. A.4a) a `Area` (viz obr. A.4b) obsahují informace o šabloně a aktuálně označené oblasti. Většinu těchto informací lze ručně změnit pomocí příslušných uživatelských vstupů. V záložce `Area` je textové pole `Info`, kde je uložen rozpoznaný text po procesu OCR, popř. libovolný popisek přidaný uživatelem.

Obrázek A.4c znázorňuje proces hledání ideální šablony pro načtený dokument. Na první pozici se vždy umisťuje šablona s nejvyšší hodnotou shody (označeno zeleně). Kliknutím na jakoukoliv nalezenou šablonu v tabulce se její zájmové oblasti ihned vykreslí na plátno (červenou barvou). Pomocí tlačítka `Stop` se zastaví proces hledání a `Cancel` zruší celý proces a uzavře záložku. Tlačítko `Apply template` slouží k aplikování vybrané šablony z tabulky nalezených šablon. Modrý obdélník indikuje, že proces hledání šablony je stále aktivní (jakmile zmizí, tak proces dokončil hledání).

Pokud nějaká z oblastí v šabloně obsahuje akci `Anonym` a zároveň se jedná o typ `Form`, pak jsou podle originálního dokumentu anonymizovány pouze variabilní části této oblasti (např. jméno, adresa, atd.). Pokud by originální



(a) Záložka s vlastnostmi šablony.

(b) Záložka s vlastnostmi vybrané oblasti.

(c) Záložka s nalezenými šablonami.

Obrázek A.4: Záložky v postranním panelu aplikace.

dokument chyběl, tak se anonymiazce pro oblast typu `Form` neprovede.