TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Návrhový systém pro zpracování a analýzu obrazu

Diplomová práce

Studijní program: N2612 – Elektrotechnika a informatika
Studijní obor: 1802T007 – Informační technologie

Autor práce: **Bc. Martin Snětivý**
Vedoucí práce: doc. Ing. Josef Chaloupka, Ph.D.

Liberec 2021

TECHNICKÁ UNIVERZITA V LIBERCI
Fakulta mechatroniky, informatiky
a mezioborových studií

Zadání diplomové práce

Návrhový systém pro zpracování a analýzu obrazu

Jméno a příjmení: **Bc. Martin Snětivý**
Osobní číslo: M19000159
Studijní program: N2612 Elektrotechnika a informatika
Studijní obor: Informační technologie
Zadávající katedra: Ústav informačních technologií a elektroniky
Akademický rok: **2020/2021**

Zásady pro vypracování:

1. Seznamte se s problematikou zpracování a rozpoznávání obrazu.
2. Navrhněte a realizujte systém (v Pythonu) pro poloautomatické zpracování a rozpoznávání obrazu.
3. Tento systém by měl obsahovat graficky přívětivé prostředí, ve kterém budou použity jednotlivé algoritmy pro zpracování a rozpoznávání obrazu z knihovny OpenCV.
4. Každý algoritmus bude realizován v rámci samostatného modulu. Moduly by mělo být možné propojovat do složitějších celků.
5. Celý systém musí být navržen tak, aby byl „otevřený“, tj. aby se v budoucnu daly do programu přidávat nové moduly.

Rozsah grafických prací:	Dle potřeby dokumentace
Rozsah pracovní zprávy:	40-50 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] ŠONKA, M., Hlaváč, V., Boyle, R.: Image processing, analysis, and machine vision. Fourth Edition. Australia: Cengage Learning, ISBN 978-1-133-59369-0, 2015.
[2] GONZALEZ, Rafael C. a Richard E. WOODS. Digital image processing. Global edition. New York: Pearson, ISBN 978-1-292-22304-9, 2017.
[3] HLAVÁČ, V., Sedláček, M.: Zpracování signálů a obrazů. 2. přeprac. vyd. Praha: ČVUT, 255 s. ISBN 978-80-01-03110-0, 2007.

Vedoucí práce:	doc. Ing. Josef Chaloupka, Ph.D. Ústav informačních technologií a elektroniky
Datum zadání práce:	19. října 2020
Předpokládaný termín odevzdání:	17. května 2021

L.S.

prof. Ing. Zdeněk Plíva, Ph.D.
děkan

prof. Ing. Ondřej Novák, CSc.
vedoucí ústavu

V Liberci dne 19. října 2020

Prohlášení

Prohlašuji, že svou diplomovou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé diplomové práce a konzultantem.

Jsem si vědom toho, že na mou diplomovou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé diplomové práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li diplomovou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má diplomová práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

13. 5. 2021 Bc. Martin Snětivý

Návrhový systém pro zpracování a analýzu obrazu

Abstrakt

Tato diplomová práce se zabývá vytvořením uživatelsky přívětivé aplikace pro analýzu, rozpoznávání a zpracování obrazových dat. Projekt je psán v jazyce Python a za pomoci knihoven Numpy, OpenCV, Dlib a dalších se provádí transformace obrazových dat. Transformací obrazových dat se rozumí například od prostých změn barevných prostorů, rotací obrazů, škálování obrazů, translací obrazů, vyhlazování obrazů až po složitější funkce jako jsou hranové detektory. Obsahuje grafické rozhraní definované frameworkem PyQt5, což je nadstavba frameworku Qt pro jazyk Python. Aplikace si uchovává historii provedených operací, která se dá poté exportovat/importovat ve formátu XML pomocí knihovny lxml. Dále byl brán ohled na dostatečnou "otevřenost"celého systému, aby následné implementace dodatečných algoritmů vyžadovaly co nejmenší čas úprav a zásahů do struktury. Systém je připraven i na jazykové rozšíření. Momentálně podporuje češtinu a angličtinu. Složitější operace běží na odlišném vlákně od hlavního za pomoci rozšíření třídy QThread, která komunikuje se zbytkem aplikace pomocí signálů knihovny PyQt5. Vytvořená aplikace je tedy dostatečně vyspělá na případné rozšíření o další nespočet algoritmů a jazykových mutací.

Klíčová slova: Python, OpenCV, PyQt5, lxml, Počítačové vidění, Python GUI, Numpy, Dlib

Project system for image processing and analysis

Abstract

This thesis is about creation of a user-friendly application for the analysis, recognition and processing of image data. It incomporates operations from simple changes of color spaces, image rotation, image scaling, image translation, image smoothing to more complex functions such as edge detectors. The project is written in Python and with the help of Numpy, OpenCV libraries it performs various image data transformation. It contains graphical user interface defined by PyQt5 framework, which is wrapper of Qt language for Python. Application maintains ordered history of all called operations, which can be exported or imported in XML format thanks to lxml library. The whole system is designed for future scalability, so further new implementations of algorithms are done at ease. System is also multilanguage ready, currently supporting Czech and English. More complex operations run on a different thread than the main thread using the extended QThread class, which communicates with the rest of the application using PyQt5 signals. The included application is more than ready for further expansions of new algorithms and languages.

Keywords: Python, OpenCV, PyQt5, lxml, Computer Vision, Python GUI, Numpy, Dlib

Obsah

Seznam zkratek

MVC	Model-View-ViewModel
OCR	Optical Character Recognition
mocap	Motion capture
CMYK	Cyan, Magenta, Yellow, Key(blacK)
RGB	Red, Green, Blue
HSV	Hue, Saturation, Value
NIR	Near-infrared
CIE	Commission internationale de l'éclairage
NTSC	National Television Standards Committee
PAL	Phase Alternating Line
PSF	Point Spread Function
FFT	Fast Fourier Transform
UI	User Interface
GUI	Graphic User Interface
API	Application Programming Interface
GPL	General Public License
LGPL	Lesser General Public License
MLL	Machine Learning Library
IPP	Integrated Performance Primitives
IA32	Intel Architecture, 32bit
IA64	Intel Architecture, 64bit
MMX	Multi Media Extension
XML	Extensible Markup Language
XSLT	Extensible Stylesheet Language Transformations
SAX	Simple API for XML
GC	Garbage Collector
WPF	Windows Presentation Foundation

Seznam obrázků

1 Úvod

Ze všech pěti smyslů–zrak, sluch, čich, chuť a hmat je zrak nepochybně jedním z klíčových smyslů, které využíváme častěji než zbyle uvedené. Oči poskytují nejen jednotky megabitů informací na první pohled, ale rychlost přenosu dat pro nepřetržité vidění pravděpodobně přesahuje 10 megabitů za sekundu (mbit/s)[1]. Nicméně většina těchto informací je nadbytečná a je filtrována různými částmi vizuální mozkové kůry. Takže vyšší centra mozku musí interpretovat podstatně menší zlomek dat. Nicméně množství informací, která vyšší centra mozku obdrží z očí, musí být alespoň o dva řády větší než všechny informace, které získají od ostatních smyslů.

Další kladnou vlastností lidské vizuální soustavy je jednoduchost jakou je prováděna analýza obrazu. Vidíme scénu přesně tak jak je: stromy v krajině, knihy na stole. K interpretaci každé scény není třeba žádného složitého úsilí. Odpovědi jsou takřka okamžité a obvykle jsou k dispozici v řádu desítek sekund. To se samozřejmě nevztahuje na optické iluze. Například Neckerova krychle, která nemá žádná vodítka vůči její orientaci a dá se tedy vyložit různými způsoby, kde se nachází její přední strana. Předešlý případ a celá řada dalších optických iluzí jsou dobře známé a z větší části je lze považovat za kuriozity. Iluze jsou důležité, neboť poukazují na skryté domněnky, které mozek dělá při svém boji s obrovským množstvím komplexních vizuálních dat, které přijímá. Důležitou pointou tohoto příběhu je to, že si většinou neuvědomujeme složitost vidění. Vidění není jednoduchý proces. Vize se vyvinula v průběhu milionů let a v naší evoluci nebyl důvod, abychom si byli vědomi složitosti tohoto úkolu. Zbytečně by naše mysl byla zaplněna irelevantními informacemi a naše reakční doba by se zcela jistě zpomalila [1].

Odborníci v oblasti počítačového vidění vyvíjeli paralelně matematické techniky pro získání trojrozměrného objektu a jeho vzhledu ve snímcích. Nyní máme tedy k dispozici spolehlivé techniky jak přesně vytvořit částečný 3D model prostředí z tisíce překrývajících se fotografií. Vzhledem k dostatečně velké sadě pohledů na konkrétní objekt nebo fasádu, lze vytvořit přesné a detailní 3D povrchové modely pomocí metody stereo matching.[1] Můžeme sledovat pohybující se osobu vůči složitému pozadí. S částečným úspěchem se můžeme pokusit najít a pojmenovat i všechny lidi na fotografii pomocí kombinace detekce a rozpoznávání obličeje, oblečení a vlasů. I přes všechny tyto pokroky však sen o počítači, který dokáže interpretovat obraz na stejné úrovni jako dvouleté dítě, zůstává nedosažitelný. Zčásti je to proto, že otázka vize je inverzní problém, ve kterém se snažíme získat neznámé, vzhledem k

[1] stereo matching - proces nalezení pixelů v různých pohledech, které odpovídají stejnému 3D bodu ve scéně.

nedostatečným informacím k úplnému řešení problému. Musíme se proto uchýlit k fyzikálním a pravděpodobnostním modelům, abychom rozlišili potenciální řešení. Nicméně modelovat fyzický svět ve své bohaté komplexnosti je daleko obtížnější než modelování hlasového traktu, který produkuje mluvené zvuky.

Dopředné modely, které používáme v počítačovém vidění, se obvykle vyvíjejí ve fyzice (radiometrie, optika a konstrukce senzorů) a v počítačové grafice. Obě tato zaměření modelují, jak se objekty pohybují a animují, jak se světlo odráží od jejich povrchů, jak se světlo rozptyluje v atmosféře a zároveň jak se láme přes objektiv fotoaparátu (nebo přes lidské oči) a nakonec jak se promítá na rovnou nebo zakřivenou obrazovou rovinu. Počítačová grafika stále ještě není dokonalá. Nicméně v limitovaných oblastech, jako je například vykreslení statické scény složené z každodenních objektů nebo animace vyhynulých stvoření, jako jsou dinosauři je iluze reality dokonalá.

V počítačovém vidění se snažíme dělat opak. Tedy popsat svět, který vidíme v jednom nebo více obrazech a rekonstruovat jeho atributy jako například tvar, osvětlení a barevnou distribuci. Je neuvěřitelné, že lidé a zvířata to dělají bez námahy, zatímco algoritmy počítačového vidění jsou tak náchylné k chybovosti. Lidé, kteří nepracují v tomto oboru často podceňují, jak složitý problém je například detekovat všechny osoby na jedné fotce. Tato mylná představa, že vize je snadná, existuje už od počátku vzniku umělé inteligence, kdy se předpokládalo, že kognitivní části inteligence (plánování a logické ověřování) jsou skutečně těžší než percepce [2].

Příklady použití počítačové vize v problémech reálného světa s praktickými příklady:

- Optické rozeznávání znaků (OCR): čtení ručně psaných poštovních směrovacích čísel a automatická detekce poznávacích značek automobilů.
- Kontrola kvality: kontrola dílů pro zajištění kvality pomocí stereo vize[2] se speciálním osvětlením k měření tolerancí na křídlech letadel, nebo na částech karoserie automobilu, nebo hledání defektů v ocelových odlitcích pomocí rentgenového vidění.
- Obchodní řetězce: rozpoznávání objektů pro automatické kasy.
- Vytváření 3D modelů (fotogrammetrie): plně automatizovaná 3D konstrukce modelů z leteckých snímků používaných v systémech jako Google maps.
- Automobilová bezpečnost: rozpoznávání neočekávaných překážek, jako jsou chodci na ulici, za podmínek, kdy aktivní techniky vidění jako radar nefungují.
- Motion capture (mocap): používání reflexních značek pozorovaných z vícero kamer, nebo jiných technik k zachycení herců pro počítačovou animaci.
- Sledování: monitorování podezřelých osob, dálničního provozu,tonoucích osob.
- Rozpoznávání otisků prstů a biometrie: automatická autentikace a také forenzní aplikace.

[2]Stereo vize - vnímání do hloubky a trojrozměrné struktury

2 Teoretická část

V následujících kapitolách bude představena teorie za použitými algoritmy v implementované aplikaci. Všechny přítomné ukázky výsledků operací pochází právě z navrhnuté aplikace.

2.1 Barevné prostory

Když dopadající světlo dopadne na snímač, tak světla z různých částí spekter se zakomponují do jednotlivých diskrétních hodnot červené, zelené a modré (RGB) barvy, které vidíme na digitálním obraze [2]. Jak tento proces funguje a jak můžeme analyzovat a manipulovat s barevnými hodnotami?

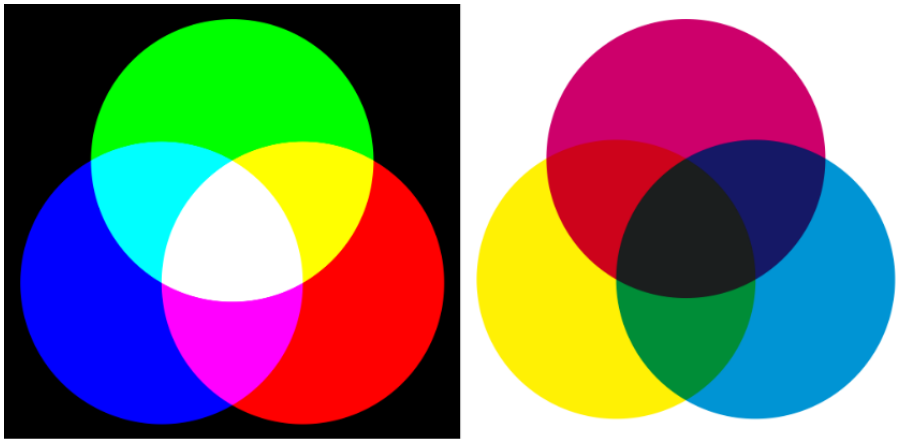
Jednoduše, z dětství si pravděpodobně pamatujete proces míchání barev k získání zcela jiné. Kombinace modré a žluté vytváří zelenou, červená a modrá dělá fialovou a červená a zelená tvoří hnědou. Později jsme se setkali se zcela jiným přístupem a tím je subtraktivní míchání barev, kde hlavní barvy jsou ve skutečnosti azurová (světle modrozelená), purpurová (růžová) a žlutá, ačkoliv černá se také často používá ve čtyřbarevném tisku (CMYK) a to z důvodu úspory pigmentu. Jinak by k vytvoření černé bylo potřeba všech tří základních barev (obrázek 2.1b).

Subtraktivní barvy se nazývají subtraktivními, protože pigmenty v barvě pohlcují určité vlnové délky v barevném spektru. Princip aditivního míchání barev se uplatňuje například v televizorech a počítačových monitorech.

Jakým postupem je možné dosáhnout dvěma rozdílnými barvami například červenou a zelenou k vytvoření třetí barvy jako je žlutá? Míchají se nějak vlnové délky k vytvoření jiné vlnové délky? Správná odpověď nemá nic společného s fyzickým mícháním vlnové délky. Existence tří primárních barev je výsledkem trichromatické (trojbarevné) nátury lidského vizuálního systému. Jelikož máme tři různé druhy kuželů, kde každý z nich reaguje odlišně na různé části barevného spektra [2]. Poznámka, pro aplikace strojového vidění jako je dálkový průzkum a klasifikace terénu, je vhodnější použít mnohem více vlnových délek. Podobně u sledovacích aplikací můžeme těžit ze snímání v blízkosti infračervené oblasti (NIR).

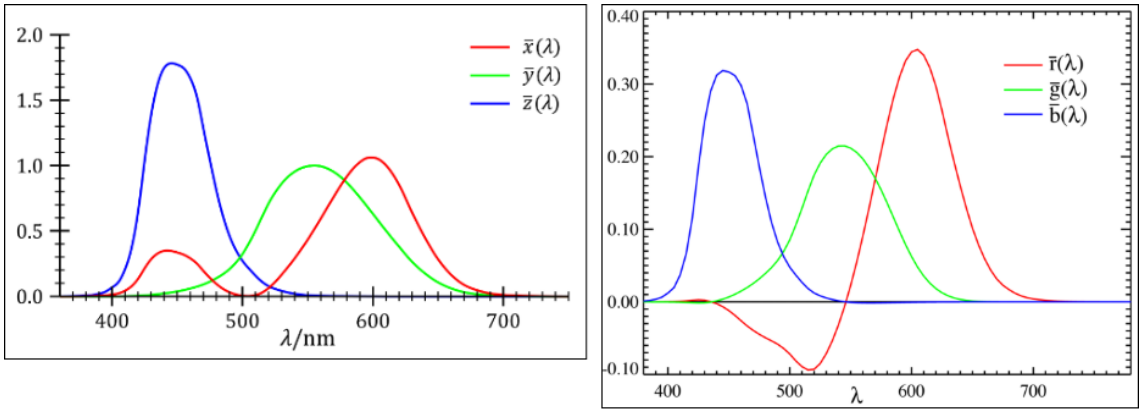
2.1.1 CIE RGB a XYZ

K otestování a kvantifikování trojbarevné teorie vnímání, se můžeme pokusit reprodukovat všechny jednobarevné barvy (jedna vlnová délka) jako směs tří vhodně zvolených primárních barev.



(a) (b)

Obrázek 2.1: Primární a sekundární barvy: (a) aditivní míchání barev červené, zelené a modré lze vytvořit azurovou, purpurovou, žlutou a bílou; (b) subtraktivní barvy azurová, purpurová a žlutá může být smíchána k vytvoření červené, zelené, modré a černé.



(a) (b)

Obrázek 2.2: Funkce shody barev CIE [2]: (a) $\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda)$ funkce shody barev, které jsou lineární kombinací $\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda)$ spektra;
(b) $\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda)$ barevná spektra získána porovnáním čistých barev s R=700.0nm, G = 546.1nm a B=435.8nm.

V roce 1930 komise *Commission Internationale d'Eclairage* (CIE) standardizovala RGB reprezentaci provedením experimentů s porovnáváním barev za použití hlavních barev červené (vlnová délka 700nm), zelené (546.1nm), modré (435.8nm). Grafy 2.2 ukazují výsledky na základě provedených experimentů zprůměrování percepčních výsledků na velké množině subjektů. Všimněte si, že pro určitá spektra v modrozeleném rozsahu je potřeba přidat záporné složky červeného světla, tedy určité množství červené barvy musí být přidáno k porovnávané barvě, aby se získala barevná shoda.

Tyto výsledky přinesly jednoduché vysvětlení existence *metamerů*, což jsou barvy s různými spektry, které jsou nerozlišitelné. Za zmínku stojí, že dvě látky nebo barvy, které jsou *metamery* pod jedním osvětlením, již nemusí být pod jiným [2].

Kvůli problému spojenému s mícháním negativního světla CIE také vyvinula nový barevný prostor nazývaný XYZ. Ten obsahuje všechny čisté spektrální barvy v kladném oktantu. Také mapuje osu Y na jasovou složku, to je vnímaný relativní jas. Čistou bílou mapuje na diagonální vektor (se stejnou hodnotou). Transformace RGB na XYZ je dána:

$$\begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = \frac{1}{0.17697} \begin{bmatrix} 0.49 & 0.31 & 0.20 \\ 0.17697 & 0.81240 & 0.01063 \\ 0.00 & 0.01 & 0.99 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} \tag{2.1}$$

Kdežto oficiální definice standardu CIE XYZ má matici normalizovanou, takže hodnota Y odpovídající čisté červené je rovna 1. Běžnější formou je vynechání vedoucího zlomku, takže druhý řádek se nasčítá na jedničku, tedy RGB vektor $(1, 1, 1)$ odpovídá Y hodnotě 1. Lineárním mícháním křivek $\bar{r}(\lambda), \bar{g}(\lambda), \bar{b}(\lambda)$ z obrázku 2.2b podle vzorce 2.1 dostáváme výsledné křivky $\bar{x}(\lambda), \bar{y}(\lambda), \bar{z}(\lambda)$ ukázané v obrázku 2.2a. Povšimněte si, jak všechna tři spektra mají nyní pouze pozitivní hodnoty a jak křivka $\bar{y}(\lambda)$ odpovídá jasu vnímaného lidmi.

Pakliže vydělíme hodnoty XYZ sumou jejich hodnot, získáme chromatické souřadnice, které se nasčítají na 1.

$$x = \frac{X}{X+Y+Z}, \quad y = \frac{X}{X+Y+Z}, \quad z = \frac{X}{X+Y+Z} \tag{2.2}$$

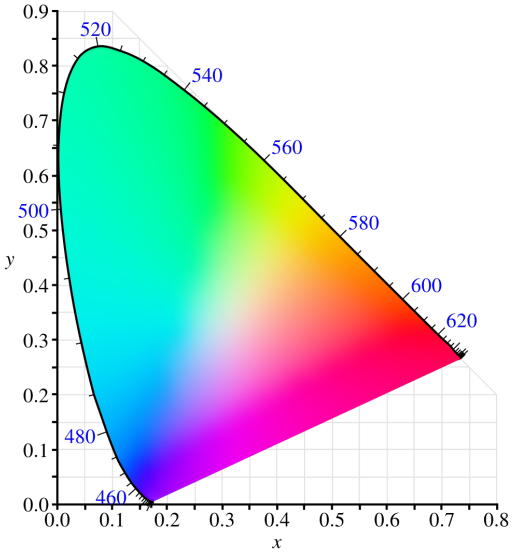
Chromatické souřadnice zahazují absolutní hodnotu intenzity daného barevného vzorku a pouze reprezentují jeho čistou barvu. Pokud zaměníme parametr z obrázku 2.2a z $\lambda = 380nm$ na $\lambda = 800nm$, získáme chromatický diagram zobrazený na obrázku 2.3. Tento obrázek promítá (x,y) hodnoty pro každou rozeznatelnou barvu vetšinou lidí.

Horní vnější zakřivený okraj představuje všechny čisté monochromatické barevné hodnoty mapované v (x, y) prostoru. Spodní přímka, která spojuje dva koncové body, je známá jako *fialová čára*. Když chceme oddělit jas a chromatičnost, je pohodlné znázornění barevných hodnot pomocí Yxy (jas plus dvě nejvýraznější barevně syté komponenty).

2.1.2 LAB

Zatímco XYZ barevný prostor má spoustu vhodných vlastností, jako třeba schopnost oddělit jas od sytosti barvy. Na druhou stranu nepředpovídá, jak dobře lidé vnímají rozdíly v barvě nebo jasu.

Protože odezva lidské vizuální soustavy je zhruba logaritmická (můžeme vnímat relativní rozdíly okolo 1%), CIE definovala nelineární mapování XYZ prostoru na prostor zvaný L*a*b (také CIELAB), kde rozdíly v jasu a sytosti jsou vnímány více jednotně. Současně byl vyvinut a standardizován další percepčně motivovaný barevný prostor L*u*v [3].



Obrázek 2.3: CIE chromatický diagram zobrazující barvy a jejich příslušné (x, y) hodnoty. Čisté spektrální barvy jsou uspořádány kolem vnějšku křivky.

L* komponenta *světlosti* je definována jako:

$$L^* = 116 f(\frac{Y}{Y_n}) \tag{2.3}$$

kde Y_n je hodnota jasu pro nominální bílou [3] a

$$f(t) = \begin{cases} t^{1/3} & t > \delta^3 \\ t/(3\delta^2) + 2\delta/3 & jinak, \end{cases} \tag{2.4}$$

je aproximace konečného sklonu k mocnině tří s parametrem $\delta = 6/29$. Výsledná škála 0 až 100 měří zhruba stejné množství vnímání světelnosti. Podobně jsou komponenty a^* a b^* definovány jako

$$a^x = 500 \left[f\left(\frac{X}{X_n}\right) - f\left(\frac{Y}{Y_n}\right) \right] \quad b^* = 200 \left[f\left(\frac{Y}{Y_n}\right) - f\left(\frac{Z}{Z_n}\right) \right] \tag{2.5}$$

2.1.3 Ostatní barevné prostory

RGB a XYZ jsou primární barevné prostory používané k popisu barevných signálů. Existuje ale celá další řada prostorů, které byly vyvinuty jak pro kódovaní videí a statických obrázků tak pro počítačovou grafiku.

YIQ standard byl jako první vyvinut pro přenos videa pro NTSC[1] v Severní Americe a s ním úzce související standard YUV pro PAL[2] v Evropě. V obou případech

[1] National Television System(s) Committee - standard kódovaní analogového televizního signálu. Oblast použití převážně Amerika, ale i Japonsko a Jižní Korea.

[2] Phase alternating line - standard analogového barevného kódovaní pro Velkou Británii a Německo.

bylo žádoucí mít dedikovaný kanál *luminance* Y, který by byl srovnatelný s běžným černobílým televizním signálem, spolu se dvěma nízkofrekvenčními barevnými kanály [2].

V obou systémech je signál Y získán z:

$$Y_{601} = 0.299R + 0.587G + 0.114B \tag{2.6}$$

kde RGB je trojice gama komprimovaných barevných komponent. Při použití novějších definic barev pro HDTV BT.709 je výpočet:

$$Y_{709} = 0.2125R + 0.7154G + 0.0721B \tag{2.7}$$

UV komponenty jsou odvozeny od zmenšených verzí $(B - Y)$ a $(R - Y)$

$$U = 0.492111(B - Y) \quad a \quad V = 0.877283(R - Y) \tag{2.8}$$

kde fázové a kvadraturní komponenty jsou UV komponenty pootočené o 33°. U kompozitního videa (NTSC, PAL) byly barevné signály poté horizontálně filtrovány dolní propustí před modulací a navrstveny na luminační signál Y. Zpětní kompatibilita byla zajištěna tím, že starší černobílé televize zkrátka ignorovaly vysokofrekvenční barevný signál (kvůli pomalé elektronice), nebo v nejhorším případě se navrstvily jako vysokofrekvenční signál na hlavní signál.

Tyto převody byly důležité v počátcích počítačové vize, kdy kompozitní TV signál byl přímo digitalizován. Dnešní kompresní metody pro video a statické snímky jsou založeny na novějším YCbCr barevném prostoru. YCbCr velice souvisí s YUV (C_b a C_r signály nesou rozdíl modré a červené a mají užitečnější mnemotechnické pomůcky než UV) ale používá rozdílné hodnoty, aby se vešly do osmibitového rozsahu dostupného pro digitální signály.

Pro video je Y signál škálován, aby se vešel do rozsahu hodnot [16...235], kdežto signály Cb a Cr jsou upraveny tak, aby se vešly do [16...240] [3, 4]. Pro statické obrazy používá JPEG standard plný osmibitový rozsah bez rezervovaných hodnot,

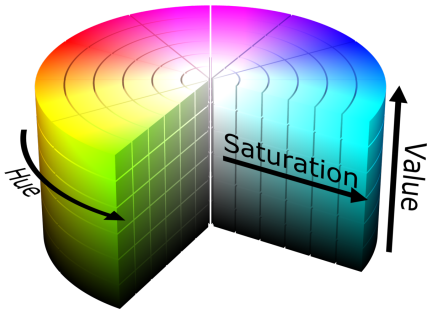
$$\begin{bmatrix} Y \\ Cb \\ Cr \end{bmatrix} = \begin{bmatrix} 0.299 & 0.587 & 0.114 \\ -0.168736 & -0.331264 & 0.5 \\ 0.5 & -0.418688 & -0.081312 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix} + \begin{bmatrix} 0 \\ 128 \\ 128 \end{bmatrix} \tag{2.9}$$

kde RGB hodnoty jsou osmibitové komprimované gama barevné komponenty (tj. skutečné RGB hodnoty, které získáme otevřením JPEG).

Další barevný prostor je *Hue, Saturation, Value*[3] (HSV), což je projekce RGB barevné kostky na nelineární úhel sytosti, radiální (jdoucí ve směru poloměru) procento sytosti a hodnotu inspirovanou jasem. Jednodušeji je hodnota definována jako střední nebo maximální hodnota barvy, sytost jako zmenšená vzdálenost od úhlopříčky a odstín je definován jako směr kolem barevného kola [5, 6].

Ačkoliv se všechny tyto barevné prostory mohou zdát matoucí, nakonec často nezáleží na tom, který z nich je použit. Percepčně motivovaný systém L*a*b je

[3]Hue - odstín; Saturation - sytost, přímes jiné barvy; Value - hodnota jasu, množství bílého světla.

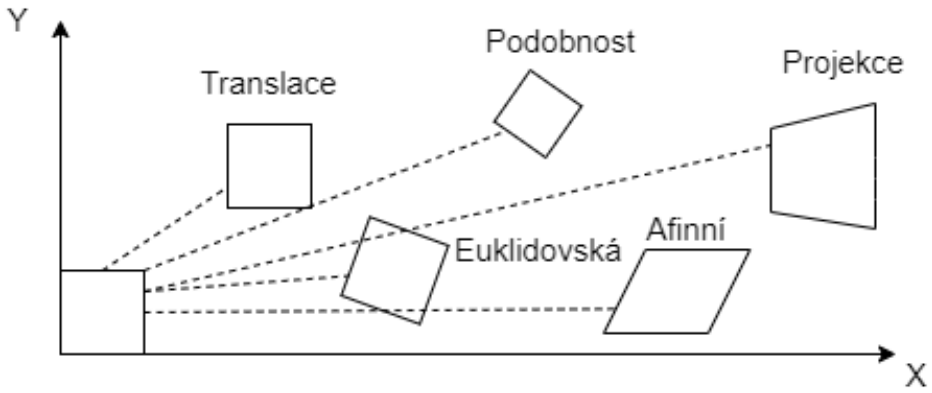


Obrázek 2.4: Grafická reprezentace barevného prostoru HSV.

kvalitativně podobný gama komprimovanému RGB systému, se kterým se většinou potýkáme. Oba mají škálování podle zlomkové mocniny (která aproximuje logaritmické) mezi skutečnými hodnotami intenzity a manipulovanými čísly [2].

2.2 2D transformace

Přehled nejjednodušších transformací se kterými lze pracovat v 2D prostoru je zobrazena níže na obrázku 2.5.



Obrázek 2.5: Základní množina 2D transformací.

2.2.1 Translace obrazu

2D translace může být napsána jako $x' = x + t$ nebo

$$x' = \begin{bmatrix} I & t \end{bmatrix} \bar{x} \tag{2.10}$$

kde I je (2×2) jednotková matice nebo

$$\bar{x}' = \begin{bmatrix} I & t \\ 0^T & 1 \end{bmatrix} \bar{x} \tag{2.11}$$

kde 0 je nulový vektor. Použitím matice 2×3 vede ke kompaktnějšímu zápisu notace, kdežto plná 3×3 matice (je získána z 2×3 přidáním řádku $[0^T 1]$) umožňuje řetězové transformace pomocí násobení matic.

2.2.2 Rotace a translace obrazu

Tato transformace je též známá jako 2D *rigid body motion*, nebo jako 2D Euklidovská transformace (Euklidovské vzdálenosti jsou zachovány)[2]. Lze jí interpretovat jako $x' = Rx + t$ nebo

$$x' = \begin{bmatrix} R & t \end{bmatrix} \bar{x} \tag{2.12}$$

kde

$$R = \begin{bmatrix} \cos\theta & -\sin\theta \\ \sin\theta & \cos\theta \end{bmatrix} \tag{2.13}$$

je ortonormální rotační matice s $RR^T = I$ a $|R| = 1$

2.2.3 Škálovaná rotace

Též známá jako podobnost, tato transformace může být reprezentována jako $x' = sRx + t$, kde s je libovolný faktor měřítka. Může být také napsána jako

$$x' = \begin{bmatrix} sR & t \end{bmatrix} \quad \overline{x} = \begin{bmatrix} a & -b & t_x \\ b & a & t_y \end{bmatrix} \bar{x} \tag{2.14}$$

kde nadále nepotřebujeme vlastnost $a^2 + b^2 = 1$. Podobnostní transformace zachovává úhly mezi úsečkami [2].

2.2.4 Afinní transformace

Afinní transformace se zapisuje jako $x' = A\bar{x}$, kde A je libovolná 2×3 matice, tj.

$$x' = \begin{bmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \end{bmatrix} \bar{x} \tag{2.15}$$

Paralelní řádky zůstávají paralelní po afinní transformací.

2.2.5 Projekce

Tato transformace je známá též jako perspektivní transformace nebo homografická, pracují nad homogenními souřadnicemi.

$$\tilde{x}' = \tilde{H}\tilde{x} \tag{2.16}$$

kde $\tilde{H}$ je libovolná 3×3 matice. $\tilde{H}$ je homogenní a definována pouze měřítkem, dvě matice $\tilde{H}$ jsou ekvivalentní, pakliže se liší pouze měřítkem. Výsledná homogenní souřadnice $\tilde{x}'$ musí být normalizována k dosažení nehomogenního výsledku x tedy

$$x' = \frac{h_{00}x + h_{01}y + h_{02}}{h_{20}x + h_{21}y + h_{22}} \quad \text{a} \quad y' = \frac{h_{10}x + h_{11}y + h_{12}}{h_{20}x + h_{21}y + h_{22}} \tag{2.17}$$

2.3 2D Konvoluce

Konvoluce je výkonná a široce používaná technika při zpracování obrazu a dalších oblastech. Definice konvoluce pro dvě funkce $f(x)$ a $g(x)$ je integrál:

$$f(x) * g(x) = \int_{-\infty}^{\infty} f(u)g(x-u)du \tag{2.18}$$

Funkce tohoto integrálu je obvykle popsána jako výsledek aplikace rozptylové funkce[4] $g(x)$ na všechny body funkce $f(x)$ a akumulaci podílů v každém bodě. Pokud je rozptylová funkce příliš úzká[5], pak je konvoluce identická s původní funkcí $f(x)$. Díky tomu je přirozené uvažovat o funkci $f(x)$ jako o funkci, která byla rozprostřena vlivem funkce $g(x)$. Toto může budit dojem, že konvoluce nutně rozmazává originální funkci, ale není tomu tak vždy. Například pokud má rozptylová funkce distribuci kladných a záporných hodnot [1].

Když se konvoluce aplikuje na digitální obraz, tak se výše uvedený vzorec 2.18 musí změnit ve dvou bodech. Vzhledem k dvourozměrnosti musí být použit dvojitý integrál. Integrace musí být změněna na diskrétní součet. Nová forma konvoluce je tedy následující:

$$F(x,y) = f(x,y) * g(x,y) = \sum_i \sum_j f(i,j)g(x-i, y-j) \tag{2.19}$$

kde g se nyní označuje jako maska prostorové konvoluce. Skutečnost, že maska musí být před použitím invertována, je pro vizualizaci procesu konvoluce nežádoucí. Pokud budeme uvažovat předem invertované masky ve formě:

$$h(x,y) = g(-x,-y) \tag{2.20}$$

Konvoluce je zjednodušena na více intuitivnější podobu:

$$F(x,y) = f(x,y) * g(x,y) = \sum_i \sum_j f(x+i, y+j)h(i,j) \tag{2.21}$$

což zahrnuje násobení odpovídajících hodnot v modifikované masce s přihlédnutím na sousední hodnoty. Vyjádření tohoto výsledku pro 3×3 masku a její koeficienty ve formě:

$$\begin{bmatrix} h_4 & h_3 & h_2 \\ h_5 & h_0 & h_1 \\ h_6 & g_7 & h_8 \end{bmatrix} \tag{2.22}$$

[4] anglicky point spread function, PSF

[5] Formálně to může být funkce delta, která je nekonečná v jednom bodě a nulová jinde a má integrál rovný jedné.

Nyní jsme schopni aplikovat konvoluci na skutečnou situaci. Použijeme následující masku na známý obraz Leny. Tato maska bude průměrovat přes sousední pixely, tedy bude se snažit odstranit šum, ale zároveň přinese značné rozmazání.

$$\frac{1}{25}\begin{bmatrix}1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\\ 1 & 1 & 1 & 1 & 1\end{bmatrix} \tag{2.23}$$

kde zlomek před maticí ovlivňuje všechny koeficienty v masce a zajišťuje, že po použití konvoluce bude průměrná intenzita obrazu zachována.

Obrázek 2.6: Ukázka potlačení šumu průměrováním sousedních hodnot. Dosaženo použitím 5×5 jednotkové matice s váhou 25. Šum je potlačen za cenu silného rozmazání obrazu.

Z výše uvedených poznatků je zřejmé, že konvoluce jsou lineárními operátory. Ve skutečnosti jsou to nejobecnější prostorově nezávislé lineární operátory [2], které lze použít na signály, například obrazový signál.

2.4 Filtrace obrazu

Filtrování obrazu zahrnuje použití okénkových operací k dosažení užitečných efektů jako je odstranění šumu nebo vylepšení obrazu. Ačkoliv se těmto typům operací lze v průmyslových aplikacích vidění vyhnout, je užitečné je prozkoumat do určité hloubky, kvůli jejich širokému použití v různých jiných aplikacích pro zpracování obrazu a protože předurčují scénu pro následující úpravy. Kromě toho se objevují některé problémy, které mají zásadní význam.

Je zřejmé, že ve skutečných obrazech může vznikat šum, a proto je nutné mít po ruce efektivní techniky pro jeho potlačení. V elektrotechnických aplikacích se

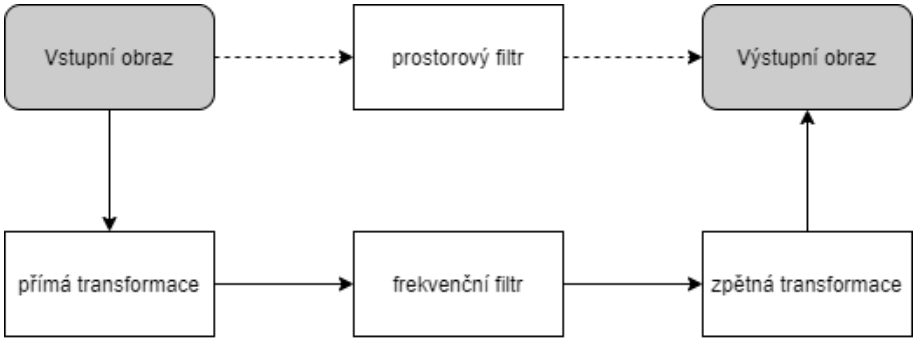
hluk typicky odstraňuje použitím dolní propusti nebo jiných filtrů, které pracují ve frekvenční doméně. Použití těchto filtrů na 1-D signály je přímočaré, protože je nutné je pouze správně umístit ve vhodných stádiích v sekvenci černých skříněk, kterými signály procházejí. Pro digitální signály se situace trochu komplikuje, protože nejprve musí být vypočtena frekvenční transformace signálu, poté musí být použita dolní propust a nakonec signál získaný z modifikované transformace převést zpět do prostorové (časové) domény [1]. Musí být tedy vypočteny dvě Fourierovy transformace. Modifikace ve frekvenčním doméně je potom jednoduchá 2.7. Ve skutečnosti je časová složitost výpočtu diskrétní Fourierovy transformace na signálu o N vzorcích řádu N^2, zapisujeme jako $O(N^2)$. Tato složitost se dá dále snížit až na $O(N \log_2 N)$ použitím rychlé Fourierovy transformace (FFT). Tento postup je potom praktický přístup jak eliminovat šum.

Při použití těchto myšlenek na obraz musíme brát v potaz, že signál je prostorový nežli časově závislý. Matematicky to nepřináší velký rozdíl, ale přesto existují významné problémy. Za prvé, neexistuje adekvátní analogová zkratka a celý proces musí být proveden digitálně. Zde ignorujeme metody optického zpracování navzdory jejich zjevné síle, rychlosti a vysokému rozlišení, protože v žádném případě je není triviální spojit s digitální počítačovou technologií. Za druhé, v případě obrazu $N \times N$ je počet operací potřebných k vypočítání Fourierovy transformace $O(N^3)$ a FFT redukuje tento počet na $O(N^2 \log_2 N)$, což je stále nezanedbatelné množství výpočtů (zde se předpokládá, že 2-D transformace jsou implementovány postupnými průchody 1-D transformací). Všimněte si také, že pro účely potlačení šumu jsou nutné dvě Fourierovy transformace (2.7). Nicméně v mnoha aplikacích zpracování obrazu je výhodné postupovat tímto způsobem, protože dokážeme odstranit nejen šum, ale mohou být odfiltrovány také další artefakty. Tato situace platí zejména pro dálkové snímání a kosmickou technologii. V průmyslu je však vždy kladen důraz na zpracování v reálném čase, takže v mnoha případech není praktické odstraňovat šum pomocí operací v prostorové doméně. Další nepříjemností je to, že dolní propust je vhodná k odstranění Gaussovského šumu, ale zkresluje obraz, pokud se používá k odstranění impulzního šumu [2].

2.4.1 Gaussův filtr

Filtrování dolní propustí se obvykle rozumí eliminaci složek s vysokými frekvencemi, a proto je přirozené provádět ji ve frekvenční doméně. Přesto je možné ji implementovat přímo v prostorové doméně. To je proveditelné díky známému faktu, že vynásobení signálu funkcí v prostorové doméně je ekvivalentní konvoluci s Fourierovskou transformací funkce v prostorové oblasti (2.7). Pokud je poslední konvoluční funkce v prostorové doméně dostatečně úzká, nebude počet výpočtů příliš složitý [1]. Tímto způsobem lze hledat uspokojivou implementaci dolní propusti. Nyní zbývá jen najít vhodnou konvoluční funkci.

Pokud má mít dolní propust vlastnost ostrého ořízntutí, bude její transformace v obrazovém prostoru oscilační. Extrémním případem je funkce *sine cardinal* neboli sinc $(\sin x/x)$, což je prostorová transformace dolní propusti obdélníkového tvaru. Oscilační konvoluční funkce jsou nežádoucí, protože mohou kolem objektu vytvořit halové jevy,



Obrázek 2.7: Filtrovat obraz lineárními filtry lze buď v prostorové, nebo frekvenční oblasti. Cesta přes frekvenční transformaci je znázorněna dole plnou čárou. Pro filtraci v prostorové oblasti (čárkovaná čára) je základním matematickým nástrojem konvoluce. Možných transformací je více, postačí si představit Fourierovu transformaci a její inverzní podobu.

které obraz velmi hrubě zkreslují. Marr a Hildreth [7] navrhl, že správné typy filtrů na obrazová data jsou ty, které neoscilují jak ve frekvenční tak v prostorové doméně. Gaussovské filtry jsou schopny optimálně splnit toto kritérium. Mají identické formy v prostorové a frekvenční doméně. V 1-D jsou tyto formy následující:

$$f(x) = \frac{1}{(2\pi\sigma^2)^{1/2}} \exp(-\frac{x^2}{2\sigma^2}) \tag{2.24}$$

$$F(\omega) = \exp(-\frac{1}{2}\sigma^2\omega^2) \tag{2.25}$$

Tedy typ prostorového konvolučního operátoru, který je vyžadován za účelem potlačení šumu dolní propustí je ten, který aproximuje Gaussovu profilu [1]. V literatuře se objevuje mnoho takových aproximací. Ty se liší na základě velikosti vybraného okolí a přesných hodnotách koeficientů konvoluční masky. Jedna z nejčastějších je následující maska, která se používá spíše pro jednoduchost výpočtu nežli pro věrnost Gaussova profilu:

$$\frac{1}{9}\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.26}$$

Další běžně používaná maska, která se blíží více Gaussovu profilu:

$$\frac{1}{16}\begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.27}$$

V obou případech se zlomky, které předcházejí masce, používají k vážení všech koeficientů masky. Tyto váhy jsou zvoleny tak, aby po aplikaci konvoluce na obraz nedošlo k ovlivnění průměrné intenzity obrazu. Předešlé dvě masky konvoluce pravděpodobně představují více něž 80% všech diskrétních Gaussových aproximací. Jelikož se jedná o 3×3 masky, mají relativně malou výpočetní složitost.

2.4.2 Mediánový filtr

Cílem zde je najít pixely v obraze, které mají extrémní a proto velice nepravděpodobné intenzity a nahradit je vhodnějšími hodnotami. Toto se podobá nakreslením křivky skrz sadu dat a ignorováním těch, která jsou zjevně daleko od této křivky. Zřejmým způsobem jak toho dosáhnout, je použitím limitního filtru, který zabrání tomu, aby jakýkoli pixel měl intenzitu mimo rozsah intenzity svých sousedů. K vytvoření této techniky je nezbytné prozkoumat rozdělení lokální intenzity v konkrétním sousedství. Body na hranici této intenzity jsou pravděpodobně způsobeny impulzním šumem. Je tedy rozumné tyto body eliminovat, ale také postupovat dále a odstranit stejné oblasti na druhé straně minimální hranice a použít nakonec medián. Dostáváme se tedy k mediánovému filtru, který vezme všechna rozdělení lokální intenzity a vygeneruje nový obrázek s odpovídajícími hodnotami mediánu. Medián filtr je vynikající v potlačení impulzního šumu, což se potvrzuje v praxi 2.8. S ohledem na rozmazání

Obrázek 2.8: Aplikace mediánového filtru s 5×5 maskou. Obraz přišel o malé ztráty jemných detailů a vypadá poněkud „změkčeně". Snímek pochází z výřezu aplikace s pomocí funkcí knihovny OpenCV.

způsobené Gaussovými operátory je vhodné se zeptat, zda medián filtr též vyvolává rozmazání. Obrázek 2.8 ukazuje, že jakékoli rozmazání je jen mírné, ačkoliv dochází k jisté ztrátě jemných detailů, které mohou nakonec působit výsledným obrázkům „změkčený"vzhled. Medián filtr bez rozostřování svým způsobem opravuje hlavní nedostatek Gaussova filtru [1], a tím se stává pravděpodobně nejrozšířenějším filtrem v aplikacích pro zpracovaní obrazu.

2.4.3 Bilaterální filtr

Nejedná se o nic jiného než spojení myšlenky o vážených filtrech s lepší verzí zamítání odlehlých hodnot. Zjednodušeně, místo zamítnutí fixního procenta α, odmítneme pixely, jejichž hodnoty se příliš liší od střední hodnoty pixelu. Toto je základní myšlenka v bilaterálním filtrování, která byla popularizována v komunitě počítačového vidění [8, 9].

Podobně jako u Gaussova filtru je i bilaterální filtr také definován jako vážený průměr pixelů. Rozdíl je v tom, že bilaterální filtr bere v úvahu variaci intenzit k zachování hran. Bilaterální filtrování vychází z toho, že dva pixely jsou k sobě blízké

nejen v případě když okupují blízké prostorové body, ale také v případě, že mají určitou podobnost ve fotometrickém[6] rozsahu [10].

Bilaterální filtr s označením $BF[.]$ je definován jako

$$BF[I]_p = \frac{1}{W_p} \sum_{q \in S} G_{\sigma_s}(\|p - q\|) G_{\sigma_r}(I_p - I_q) I_q \tag{2.28}$$

kde W_p je normalizační faktor

$$W_p = \sum_{q \in S} G_{\sigma_s}(\|p - q\|) G_{\sigma_r}(I_p - I_q) \tag{2.29}$$

Parametry σ_s a σ_r udávají míru filtrace pro obraz I. Rovnice 2.28 je normalizovaný vážený průměr, kde G_{σ_s} je Gaussova funkce prostoru, která snižuje vliv vzdálených pixelů. G_{σ_r} je Gaussova funkce pro rozsah(intenzitu) snižuje vliv pixelů q s intenzitou rozdílnou od I_p. Jak už bylo řečeno, pojmem rozsah je zde myšlena hodnota pixelu, kdežto prostorem je zde myšleno umístění samotného pixelu. Na obrázku 2.9 je vidět příklad použití bilaterálního filtru s parametry $\sigma_s = 75$, $\sigma_r = 75$.

Obrázek 2.9: Výřez výsledku z aplikace po aplikaci bilaterálního filtru z knihovny OpenCV s parametry $\sigma_s = 75$, $\sigma_r = 75$. Vlevo je vstupní obraz a vpravo je výsledný.

Bilaterální filtr je ovládán dvěma parametry σ_s a σ_r.

- Při zvětšování parametru σ_r se bilaterální filtr blíží Gaussovu rozostření, protože rozsah Gaussovy funkce je plošší, neboli téměř konstantní v celém intervalu intenzit obrazu.
- Zvyšováním prostorového parametru σ_s vyhlazujeme větší rysy.

[6]Fotometrie - oblast optiky popisující světlo a jeho účinky na lidské oko.

2.5 Hranové detektory

Pro vnímání člověka jsou velmi důležitá místa v obraze, kde se náhle mění hodnota jasu. Těmto pixelům říkáme hrany. Technikám k nalezení takových míst v obraze slouží právě lokální předzpracování hledání hran.

Hrana je charakterizována vlastnostmi obrazového elementu a jeho okolím a je určena tím, jak se prudce mění hodnota obrazové funkce $f(x, y)$. Nástrojem pro prozkoumání změn dvou proměnných jsou parciální derivace. Gradient udává změnu funkce, vektorová veličina ∇ zase udává směr největšího růstu funkce (směr gradientu) a strmost tohoto růstu (velikost, modul gradientu). Pixely, které mají velkou strmost jsou právě hrany [11].

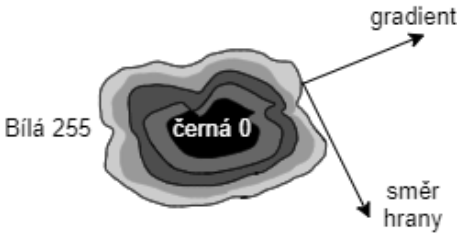
Pro spojitou obrazovou funkci f(x,y) je velikost gradientu $|\nabla f(x, y)|$ a jeho směr ψ dán vztahy:

$$|\nabla f(x, y)| = \sqrt{\left(\frac{\partial f}{\partial x}\right)^2 + \left(\frac{\partial f}{\partial y}\right)^2} \tag{2.30}$$

$$\psi = arg\left(\frac{\partial g}{\partial x}, \frac{\partial g}{\partial y}\right) \tag{2.31}$$

kde $arg(x, y)$ je úhel v radiánech mezi osou x a polohovým vektorem k bodu (x, y). Pokud nás zajímá pouze velikost gradientu, neboli hrany, bez ohledu na její směr, používá se všesměrový lineární Laplaceův operátor, Laplacián ∇^2. Vychází z druhé parciální derivace:

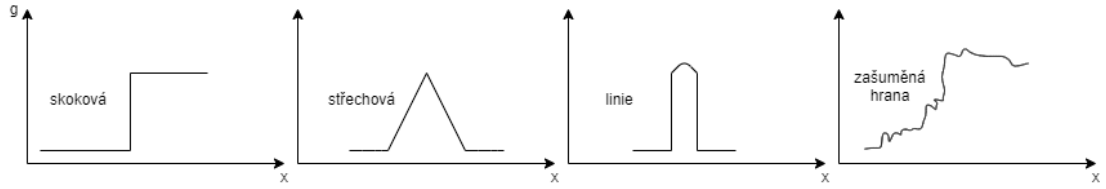
$$\nabla^2 g(x, y) = \frac{\partial^2 g(x, y)}{\partial x^2} + \frac{\partial^2 g(x, y)}{\partial y^2} \tag{2.32}$$



Obrázek 2.10: Mezi směrem hrany Φ a směru gradientu Ψ je pravý úhel.

Nalezené hrany v obraze se sporadicky používají k hledání hranic objektů. A to právě tehdy, pokud má objekt homogenní hodnoty jasu. Body hranice jsou potom pixely s vysokou hodnotou gradientu. Na obrázku 2.10 se hraniční pixely spojují do hranic, a proto je někdy směr hrany Φ definován jako kolmý na směr gradientu Ψ [11].

Na obrázku 2.11 je vidět přehled, jak lze jednotlivé hrany roztřídit podle jejich 1D jasového profilu ve směru gradientu u konkrétního pixelu. První tři typy jsou spíše ukázkového typu. Reálně se setkáme až se čtvrtým, tedy zašuměnou hranou.



Obrázek 2.11: Nejběžnější hrany a jejich jasové profily.

Gradientní operátory nachází použití i u ostření obrazu. Ostřením obrazu se rozumí upravení obrazu tak, aby obraz měl strmější hrany. Ostření odpovídá zdůraznění vysokých frekvencí. Pro obraz f, na který bylo aplikováno ostření obrazu g, lze definovat:

$$f(x,y) = g(x,y) - C\,S(x,y) \tag{2.33}$$

kde C je kladný koeficient udávající sílu ostření a $S(x,y)$ je operátor udávající strmost změny obrazové funkce v daném bodě. Strmost může být definována Laplaciánem nebo modulem gradientu.

Gradientní operátory definující strmost obrazových funkcí lze rozdělit do tří kategorií [11]:

- Operátory aproximující derivace pomocí diferencí. Některé aproximují první derivaci, které používají vícero masek odpovídající konkrétní rotaci. Z nich je vybrána ta, která nejlépe lokálně aproximuje obrazovou funkci $f(x,y)$. Další operátory typu Laplacián aproximující druhou derivaci jsou invariantní vůči rotaci a mohou být počítány pomocí konvoluce s jedinou maskou.
- Operátory založené na hledání hran v místech kde druhá derivace prochází nulou[7]. Příkladem je Cannyho hranový detektor viz. kapitola 2.6.
- Operátory snažící se lokálně aproximovat obrazovou funkci poměrně jednoduchým parametrickým modelem, například polynomem dvou proměnných.

2.5.1 Robertsův operátor

Nejstarším a zároveň velmi jednoduchým operátorem je právě Robertsův. Používá masku s okolím 2×2 pro daný pixel. Masky jsou následující:

$$h1 = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix}, \quad h2 = \begin{bmatrix} 0 & -1 \\ 1 & 0 \end{bmatrix} \tag{2.34}$$

Velikost gradientu se získá ze vzorce:

$$|g(x,y) - g(x+1,y+1)| + |g(x,y+1) - g(x+1,y)| \tag{2.35}$$

Hlavní nevýhodou tohoto gradientu je velká citlivost na šum, protože okolí 2×2 použité pro aproximaci je malé.

[7] angl. zero crossing

2.5.2 Sobelův operátor

Nachází využití při detekci vodorovných a svislých hran, na což postačí masky h_1, h_3:

$$h_1 = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \; h_2 = \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix} \; h_3 = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.36}$$

2.5.3 Laplaceův operátor

Jak už bylo řečeno, Laplaceův gradientní operátor ∇^2 aproximující druhou derivaci, je invariantní vůči rotaci a udává velikost hrany nikoliv její směr. Dvě častá 3×3 konvoluční jádra (pro 4-sousedství a 8-sousedství) jsou:

$$h_1 = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix} \quad h_2 = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.37}$$

Lze použít i Laplacián s větší váhou pixelů blíže reprezentativnímu bodu masky, pak ale ztrácíme invariantnost vůči rotaci:

$$h_1 = \begin{bmatrix} 2 & -1 & 2 \\ -1 & -4 & -1 \\ 2 & -1 & 2 \end{bmatrix} \quad h_2 = \begin{bmatrix} -1 & 2 & -1 \\ 2 & -4 & 2 \\ -1 & 2 & -1 \end{bmatrix} \tag{2.38}$$

Laplacián má dvě nevýhody, první je velká citlivost na šum. Pochopitelné vzhledem ke snaze aproximovat druhou derivaci primitivními prostředky. Další nevýhodou jsou dvojité odezvy na hrany odpovídající tenkým liniím v obraze [11].

2.6 Cannyho hranový detektor

Od svého zvěřejnění v roce 1986 se Cannyho hranový detektor stal jedním z nejpoužívanějších detektorů hran, protože se snaží odpoutat od tradičních metod založených na maskách [12]. Součástí metody je pečlivé stanovení prostorové šířky pásma v němž se předpokládá, že bude fungovat, a také vyloučení zbytečných prahových hodnot. To umožňuje vznik tenkých linií struktur, které jsou pokud možno vzájemně propojené a skutečně mají smysl v konkrétním měřítku a šířce pásma. V důsledku těchto úvah metoda zahrnuje řadu fází zpracování [1]:

1. Filtrování dolní propustí nad frekvenční prostorové oblasti.
2. Aplikace diferenciálních masek prvního řádu.
3. Nemaximální suprese zahrnující subpixelovou interpolaci inzenzit pixelů.
4. Hysterezní prahování.

Obrázek 2.12: Výřez výsledku z aplikace po převedení obrazu do stupňů šedi a použití metody cv2.Canny(vstup, min, max). Zadané parametry $min = 80$ a $max = 200$. Další parametry aperture_size a L2gradient jsou nepovinné a byly nechány v defaultních hodnotách, tedy 3 a false. Aperture_size je velikost Sobelovy masky a L2gradient představuje použití přesnějšího vzorce.

Filtrování dolní propustí se musí v zásadě provádět pomocí Gaussovy konvoluce u které je standardní odchylka (nebo prostorová šířka pásma) σ známá a předem specifikována. Poté je třeba použít diferenciální masky prvního řádu. Pro tento účel je Sobelův operátor dostatečný. V této souvislosti lze považovat Sobelovu masku za konvoluci základní masky typu $\begin{bmatrix}-1 & 1\end{bmatrix}$ s maskou vyhlazovací $\begin{bmatrix}1 & 1\end{bmatrix}$. Vezmeme-li derivaci ze Sobela podle x, máme:

$$\begin{bmatrix}-1 & 0 & 1\\ -2 & 0 & 2\\ -1 & 0 & 1\end{bmatrix} = \begin{bmatrix}1\\ 2\\ 1\end{bmatrix} \begin{bmatrix}-1 & 0 & 1\end{bmatrix} \tag{2.39}$$

kde

$$\begin{bmatrix}1 & 2 & 1\end{bmatrix} = \begin{bmatrix}1 & 1\end{bmatrix} \circledast \begin{bmatrix}1 & 1\end{bmatrix} \tag{2.40}$$

a

$$\begin{bmatrix}-1 & 0 & 1\end{bmatrix} = \begin{bmatrix}-1 & 1\end{bmatrix} \circledast \begin{bmatrix}1 & 1\end{bmatrix} \tag{2.41}$$

Tyto rovnice potvrzují, že Sobelův operátor sám o sobě nese dostatečné množství nízkofrekvenčního filtrování, takže množství dalšího filtrování potřebného ve fázi 1 může být přiměřeně sníženo. Filtrování dolní propustí může být provedeno pomocí vyhlazovací masky zobrazené v tabulce 2.1(b). Je zajímavé, jak blízko je tato maska úplnému 2-D Gaussovi ukázanému na 2.1(a). Všimněte si také, že šířka pásma masky v tabulce 2.1(b) je známá hodnota (0.707) a pokud je kombinována s šířkou pásma Sobela, je celková šířka pásma téměř 1,0.

Dále se zaměříme na fázi 3, tedy nemaximální potlačení. Pro tento účel musíme určit normálu lokální hrany podle rovnice:

$$\theta = \arctan \frac{g_y}{g_x} \tag{2.42}$$

a pohybovat se v obou směrech podél normály, abychom určili, zda současná lokace je či není lokálním maximem. Pokud není, tak potlačíme výstup hrany na aktuálním umístění a ponecháme pouze body hrany, které jsou osvědčené lokální maxima podél hranové normály. Protože pouze jeden bod v tomto směru by měl být lokálním maximem, tak tento postup nutně ztenčí hrany šedi na jednotky šířky. Zde vzniká mírný problém v tom, že směr normály obecně neprochází středy sousedních pixelů a Cannyho metoda vyžaduje, aby intenzity podél normály byly odhadnuty interpolací. V 3×3 masce je toho jednoduše dosaženo. V každém oktantu musí normála hrany ležet uvnitř dané dvojice pixelů, jak je ukázáno na obrázku 2.13(a). Ve větší masce může interpolace probíhat mezi několika páry pixelů. Například pro 5×5 masku, bude muset být určeno, který ze dvou párů je relevantní (obrázek 2.13(b)) a použít vhodný interpolační vzorec. Málokdy však bude potřeba používat větší masku než 3×3, jelikož ta obsahuje všechny relevantní informace a při dostatečném předešlém vyhlazení ve fázi 1 dojde k zanedbatelné ztrátě přesnosti. Pokud je přítomný impulsní šum, mohlo by to vést k zásadní chybě, ale filtrování dolní propustí rozhodně nezaručuje eliminaci impulsního šumu, takže při použití menší masky nedochází k žádné zvláštní ztrátě. Tyto úvahy je třeba pečlivě prozkoumat s ohledem na konkrétní obrazová data a jejich šum. Obrázek 2.13 ukazuje dvě vzdálenosti l_1 a l_2, které musíme určit. Intenzita pixelu podél normály hrany je získána vážením odpovídajících hodnot intenzit pixelů v opačném poměru vzdálenostem:

$$I = \frac{l_2 I_1 + l_1 I_2}{l_1 + l_2} = (1 - l_1) I_1 + l_1 I_2 \tag{2.43}$$

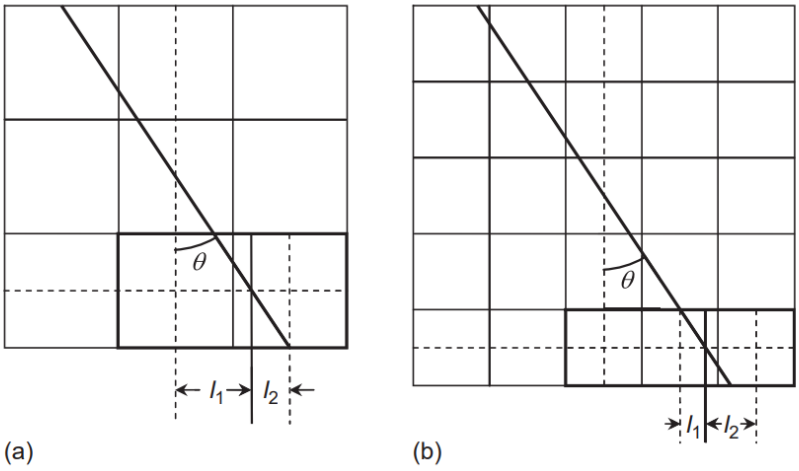
kde

$$l_1 = \tan \theta \tag{2.44}$$

To nás přivádí k poslednímu bodu a to k prahování podle hystereze. Do tohoto bodu jsme se kompletně obešli bez použití prahování. V tomto bodě je nezbytné učinit tento poslední krok. Účelem použití dvou prahů hystereze je omezení výsledků, které mohou být vyhodnoceny jedním prahem a opravit je druhým. Vybereme horní práh pro zachycení spolehlivých hran, pak vybereme další body, které mají vysokou pravděpodobnost, že budou použitelné body hrany, protože sousedí se spolehlivými body hran. Ve skutečnosti se stále jedná o ad hoc přístup. V praxi to přináší uspokojivé výsledky. Jednoduchým pravidlem pro výběr dolního prahu je to, že by se měl rovnat přibližně polovině horního prahu [1]. Toto je pouze orientační pravidlo, které nebude platit na všechna obrazová data.

2.7 Binární morfologické operace

Binární obraz je takový, který obsahuje pouze přesně dvě hodnoty barev typicky černou a bílou. Takový obraz může nést řadu nedokonalostí zejména šum a objekty mohou být necelistvé po použití jednoduchého prahování. Morfologické operace se tedy zabývají odstraněním těchto nedokonalostí s ohledem na strukturu a formu obrazu.



Obrázek 2.13: Interpolace pixelů pro Cannyho metodu. (a) Interpolace mezi dvěma zvýrazněnými pixely v pravém dolním rohu pro 3×3 masku. (b) Interpolace pro 5×5 masku, existují dvě možnosti interpolace mezi páry sousedních pixelů, relevantní vzdálenosti jsou označeny vpravo [1].

Tabulka 2.1: Přesné 3×3 vyhlazovací jádra. Tato tabulka ukazuje vyhlazovací jádro (a), které je nejblíže známému 3×3 vyhlazovacímu jádru (b).

$$\begin{bmatrix} 0.000 & 0.000 & 0.004 & 0.008 & 0.004 & 0.000 & 0.000 \\ 0.000 & 0.016 & 0.125 & 0.250 & 0.125 & 0.016 & 0.000 \\ 0.004 & 0.125 & 1.000 & 2.000 & 1.000 & 0.125 & 0.004 \\ 0.008 & 0.250 & 2.000 & 4.000 & 2.000 & 0.250 & 0.008 \\ 0.004 & 0.125 & 1.000 & 2.000 & 1.000 & 0.125 & 0.004 \\ 0.000 & 0.016 & 0.125 & 0.250 & 0.125 & 0.016 & 0.000 \\ 0.000 & 0.000 & 0.004 & 0.008 & 0.004 & 0.000 & 0.000 \end{bmatrix} \qquad \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix}$$

a) b)

Jedná se o kolekci nelineárních operací, které spoléhají pouze na relativní řazení hodnot pixelů, nikoli na jejich konkrétní číselné hodnoty, a proto jsou zvláště vhodné pro binární obrazy. Tyto techniky lze také rozšířit na obrázky ve stupních šedi.

Šablona zvaná strukturní prvek je matice, která pracuje nad vybraným pixelem a definuje jeho sousedství. Toto sousedství se poté používá při zpracování každého pixelu a prozkoumávání obrazu v těchto morfologických technikách [13].

Typicky jsou strukturní elementy lichých rozměrů tvaru čtverce a počátek je definovaný uprostřed. Lze se ale setkat s odlišnými tvary typu elipsa, kříž, kruh atp. Strukturní elementy hrají podobnou roli jako konvoluční jádra při lineárním filtrování obrazu.

2.7.1 Dilatace

Označuje se jako $\oplus$. Strukturovací element se používá k rozšiřování tvarů obsažených ve vstupním obraze. Konkrétně funguje jako filtr lokálního maxima. Dilatace má opačný účinek než eroze. Přidává vrstvu pixelů k vnitřní i vnější hranici objektu.

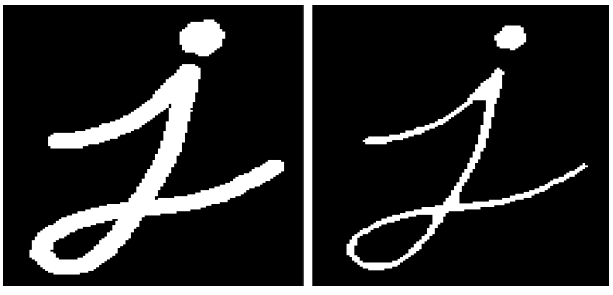
To znamená, že hodnota výstupního pixelu je maximální hodnotou všech pixelů v sousedství. V binárním obraze to tedy znamená, že pixel má hodnotu 1, pokud má některý ze sousedních pixelů hodnotu 1. Dilatace dělá objekty více viditelnými a zaplňuje mezery v objektech [13].

Obrázek 2.14: Ukázka z aplikace za použití knihovny OpenCV a metody cv2.dilate(vstup, jádro, iterace). Jádro je čtvercové 5×5 vytvořené pomocí metody cv2.getStructuringElement a iterace byla nastavena na 1.

2.7.2 Eroze

Přímý opak dilatace, značeno $\ominus$. Strukturní element redukuje tvary obsáhlé v obraze. Konkrétně funguje jako filtr lokálního minima. Strukturovací element nutně zmenšuje obraz, neboť odstraňuje vrstvy pixelů jak z vnitřní tak i vnější hranice oblasti. Pomocí eroze můžeme eliminovat díry a mezery mezi různými oblastmi.

Hodnota výstupního pixelu je minimální hodnota všech pixelů v sousedství. V binárním obraze to tedy znamená, že pixel má hodnotu 0, pakliže jakýkoliv sousední pixel má hodnotu 0. Eroze maže ostrůvky a miniaturní pozůstatky [13], takže pouze významnější oblasti jsou zachovány.



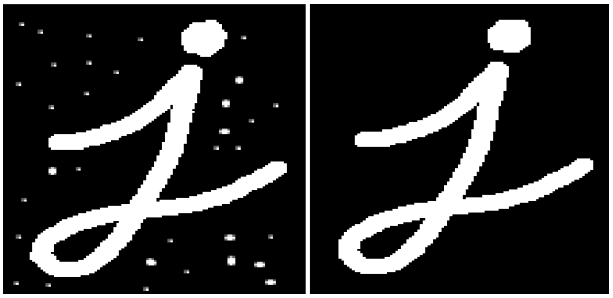
Obrázek 2.15: Ukázka z aplikace za použití knihovny OpenCV a metody cv2.erode(vstup, jádro, iterace). Jádro je čtvercové 5×5 vytvořené pomocí metody cv2.getStructuringElement a iterace byla nastavena na 1.

2.7.3 Otevření

Jedná se o dvě operace skloubené dohromady a to sice eroze a dilatace v tomto pořadí. Formálně tedy

$$A \, o \, B = (A \ominus B) \oplus B. \tag{2.45}$$

Otevření odstraní všechna úzká spojení a čáry mezi dvěma oblastmi [13]. Funkce otevření eroduje obraz a poté tento obraz dilatuje za použití stejného strukturního elementu. Morfologické otevření je užitečné k odstranění malých objektů z obrazu při zachování tvaru a velikosti větších objektů v obraze.



Obrázek 2.16: Ukázka z aplikace za použití knihovny OpenCV. Jádro je čtvercové 5×5.

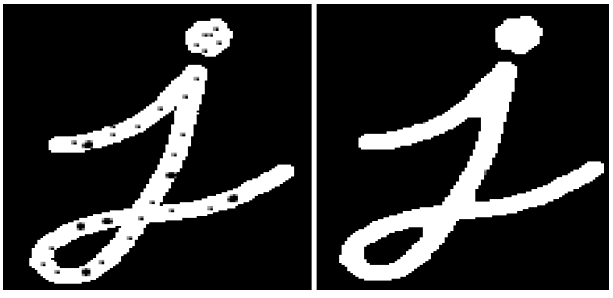
2.7.4 Uzavření

Opět kombinace předem zmíněných operací. V tomto případě opačná kombinace otevření, nejdříve dilatace a poté eroze tedy

$$A \bullet B = (A \oplus B) \ominus B \tag{2.46}$$

Uzavření vyplní úzké černé oblasti nebo díry v obraze [13].

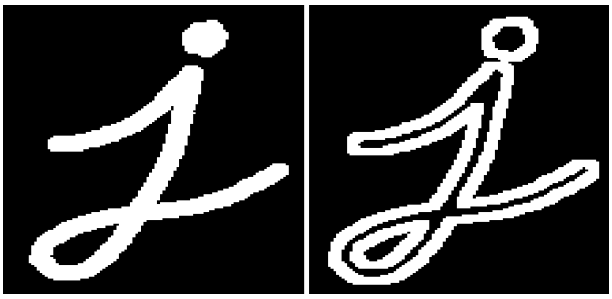
Operace uzavření nejprve dilatuje obraz a poté tento obraz eroduje, za použití stejného strukturovacího elementu. Používá se k vyplnění malých otvorů v obraze při zachování tvaru a velikosti objektů v obraze.



Obrázek 2.17: Ukázka z aplikace za použití knihovny OpenCV. Jádro je čtvercové 5×5.

2.7.5 Gradient

Rozdíl mezi dilatací a erozí obrazu. Slouží k vyhledání hranic nebo hran v obraze. Tato metoda je velice senzitivní na šum [13], je tedy doporučeno předem použít filtrování.



Obrázek 2.18: Ukázka z aplikace za použití knihovny OpenCV. Jádro je čtvercové 9×9.

2.7.6 Top hat

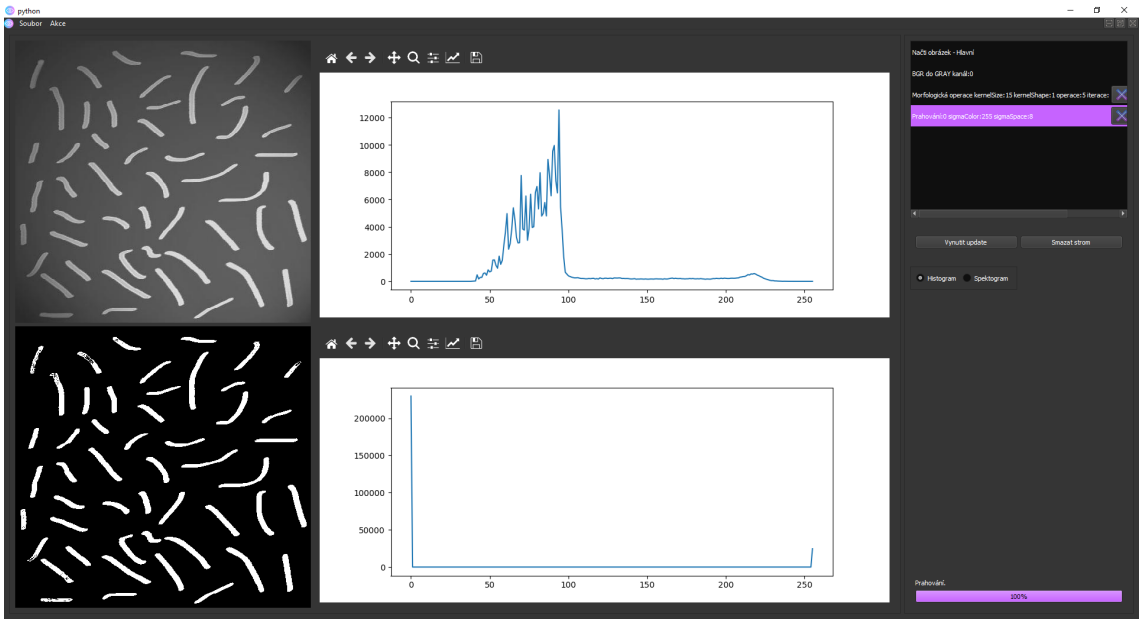
Rozdíl vstupního obrazu a otevření vstupního obrazu. Zdůrazňuje úzké cesty mezi různými oblastmi. Tato transformace použije operaci otevření na obraz a poté odečte tento obraz od vstupního obrazu. Používá se pro zvýšení kontrastu obrazu ve stupních šedi s nerovnoměrným osvětlením. Může také izolovat malé světlé objekty v obraze [13].

Následuje praktická ukázka na obraze, který je zatížen nerovnoměrným osvětlením. Chceme z něho oddělit objekty od pozadí. Běžným postupem by bylo použití segmentace na základě hodnot intenzity. Pakliže však obraz obsahuje nerovnoměrné osvětlení, může se stát, že se vyskytnou chyby, protože některé objekty v tmavších oblastech mají blízké hodnoty intenzity jako pozadí. Tedy místo přímého použití Otsuvy segmentace použijeme nejdříve top hat operaci.

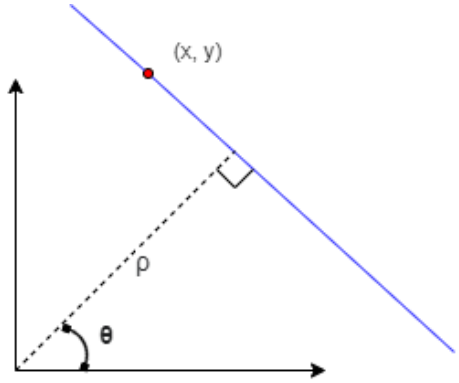
Výsledný řetězec operací je následující. Obraz převedeme do stupňů šedi, aplikujeme top hat operaci s maskou typu kříže velikosti 15×15, tím opravíme nerovnoměrné osvětlení, protože kontrast mezi pozadím a objektem zájmu je nyní zřetelný a naposled aplikujeme prahování pomocí Otsuva algoritmu.

2.8 Houghova transformace

Houghova transformace je metoda extrakce charakteristických rysů pro detekci jednoduchých tvarů jako jsou kruhy, čáry a tak dále. Jednoduchým je myšleno tvar, který lze reprezentovat několika parametry. Například přímku lze parametrizovat dvěma parametry a kruh třemi. Hlavní výhodou Houghovy transformace je necitlivost na okluzi [14].



Obrázek 2.19: Ukázka z aplikace za použití knihovny OpenCV. Snaha o extrakci rýže. Obraz převedeme do stupňů šedi, aplikujeme top hat operaci s maskou typu kříže velikosti 15×15, tím opravíme nerovnoměrné osvětlení, protože kontrast mezi pozadím a objektem zájmu je nyní zřetelný. Nakonec aplikujeme prahování pomocí Otsuva algoritmu.



Obrázek 2.20: Přímka v polárních souřadnicích.

polární forma přímky je reprezentována jako:

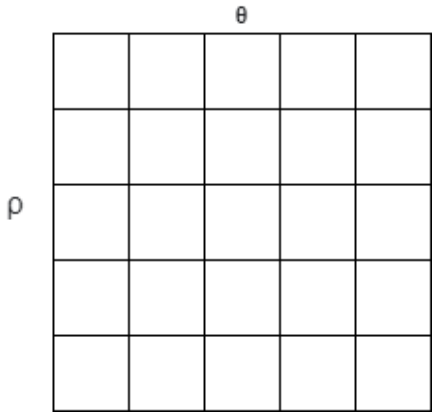
$$\rho = x\cos(\theta) + y\sin(\theta) \tag{2.47}$$

kde ρ představuje kolmou vzdálenost čáry od počátku v pixelech a θ je úhel v radiánech, který čára vytváří s počátkem, jak je zobrazeno v obrázku 2.20. Důvodem proč se nepoužívá směrnicová rovnice přímky:

$$y = kx + q \tag{2.48}$$

je ten, že sklon k může nabývat hodnot mezi $-\infty$ až $+\infty$. Pro Houghovu transformaci musí být parametry ohraničené. To vede k otázce, zda ve formě (ρ, θ) nemůže nastat podobný případ. θ je omezená, ale teoreticky ρ může nabývat hodnot mezi 0 až $+\infty$. Teoreticky to může být pravda, ale v praxi je to omezeno tím, že samotný obraz je konečný.

Když řekneme, že přímka v 2D prostoru je parametrizována pomocí ρ a θ, znamená to, že pokud vybereme libovolnou hodnotu (ρ, θ), bude jí odpovídat jedna přímka. Představme si 2D pole, kde osa x má všechny možné hodnoty θ a osa y má všechny možné hodnoty ρ. Jakákoliv (x, y) položka představuje jednu přímku.



Obrázek 2.21: Akumulátor.

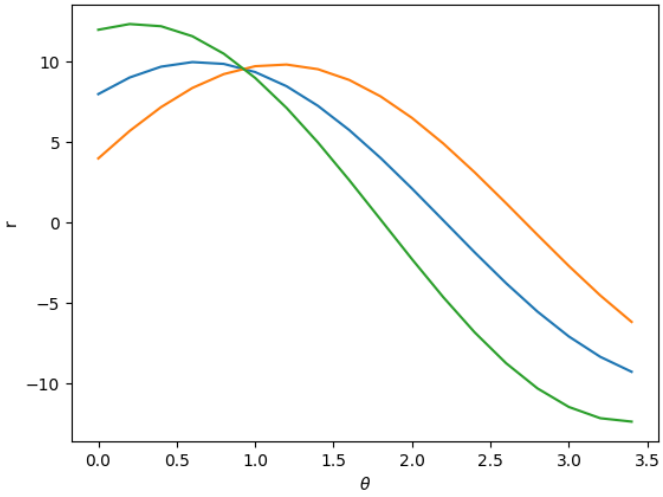
Toto 2D pole se nazývá akumulátor, protože budeme používat přihrádky tohoto pole ke shromažďovaní údajů, které přímky existují v obraze. Levá horní buňka odpovídá $(-R, 0)$ a dolní pravá (R, π). Hodnota uvnitř buňky (ρ, θ) se bude zvětšovat s přibývajícími údaji o přítomnosti přímky s parametry ρ a θ. Následující kroky jsou provedeny k nalezení přímky v obraze [14]:

- Inicializace akumulátoru - počet buněk je návrhové rozhodnutí, uvažujme 10×10 akumulátor. To znamená, že θ i ρ mohou pojmout 10 rozdílných hodnot a tedy budou schopny detekovat 100 druhů přímek. Velikost akumulátoru závisí též na velikosti vstupního obrazu.

- Detekce přímek - myšlenka je taková, že pokud je v obraze viditelná přímka, měl by hranový detektor určit její okraj. Tyto hraniční pixely poskytují důkaz o přítomnosti přímky. Výstup hranového detektoru je pole hraničních pixelů $[(x_1, y_1), (x_2, y_2)...(x_n, y_n)]$.

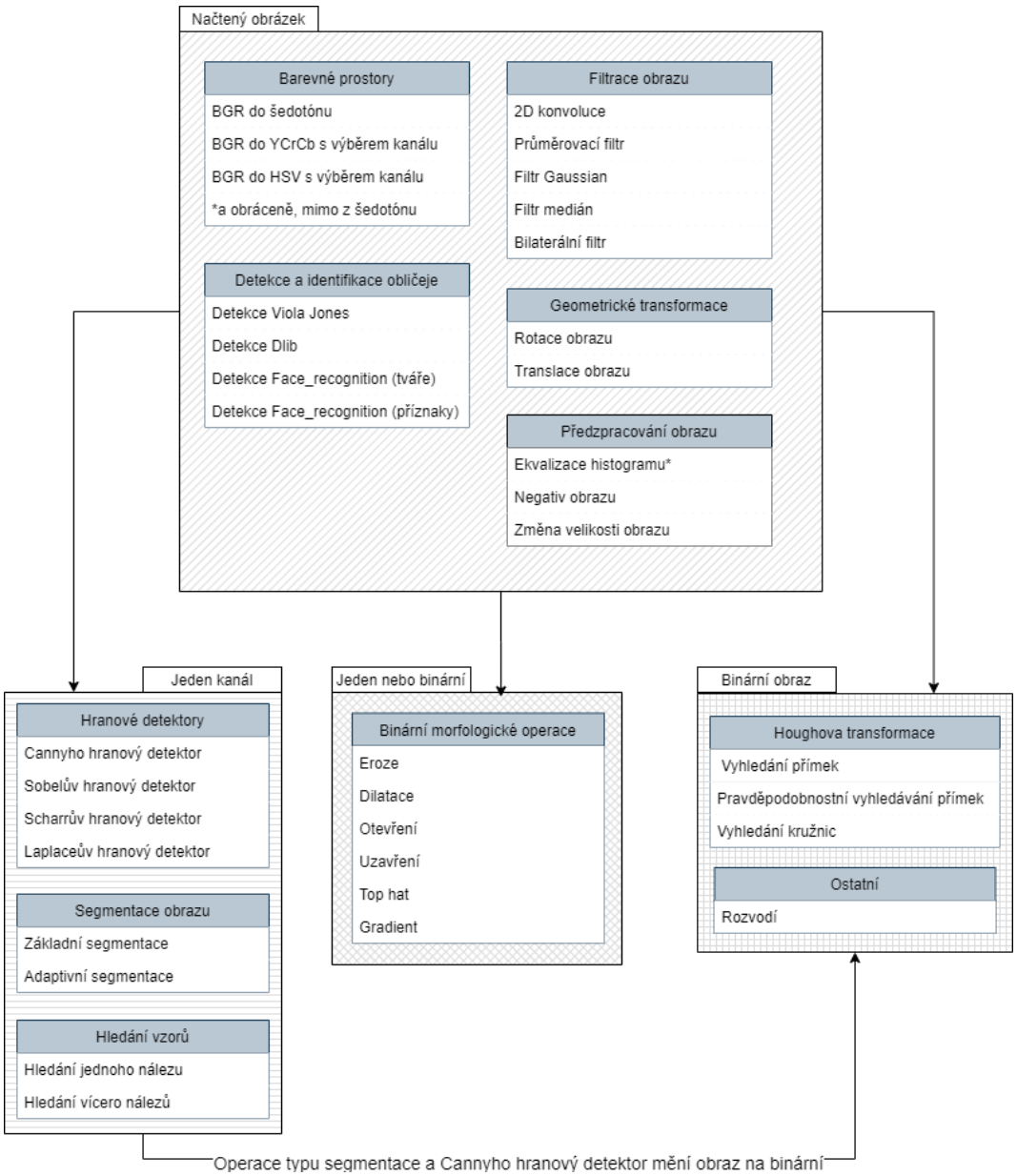
- Hlasování - pro každý hraniční pixel (x, y) ve výše uvedeném kroku měníme hodnoty θ od 0 do π a dosazujeme je do rovnice 2.47, abychom získali hodnotu ρ. V grafu 2.22 uvažujeme θ pro tři pixely a získáme hodnoty pro ρ z rovnice 2.47. Je viditelné, že tyto křivky se protínají v bodě, což znamená, že jimi vede přímka s parametry $\theta = 1$ a $\rho = 9.5$. Typicky máme stovky hranových pixelů a akumulátor se používá k nalezení průsečíků všech křivek, které jsou generovány hranovými pixely.

Uvažujme akumulátor velikosti 20×20. Tedy je zde 20 unikátních hodnot θ a tedy pro každý hranový pixel (x, y) můžeme spočítat 20 párů (ρ, θ) pomocí rovnice 2.47. Políčko v akumulátoru odpovídající těmto 20 (ρ, θ) se inkrementuje. Toto uděláme pro každý hranový pixel a nyní máme akumulátor, který má všechny informace o všech možných přímkách v obraze. Můžeme jednoduše vybrat políčka v akumulátoru nad určitou prahovou hodnotou, abychom našli přímky v obraze. Pokud je práh vyšší, najdeme méně silných čar a pokud je nižší, najdeme velké množství přímek včetně slabých.

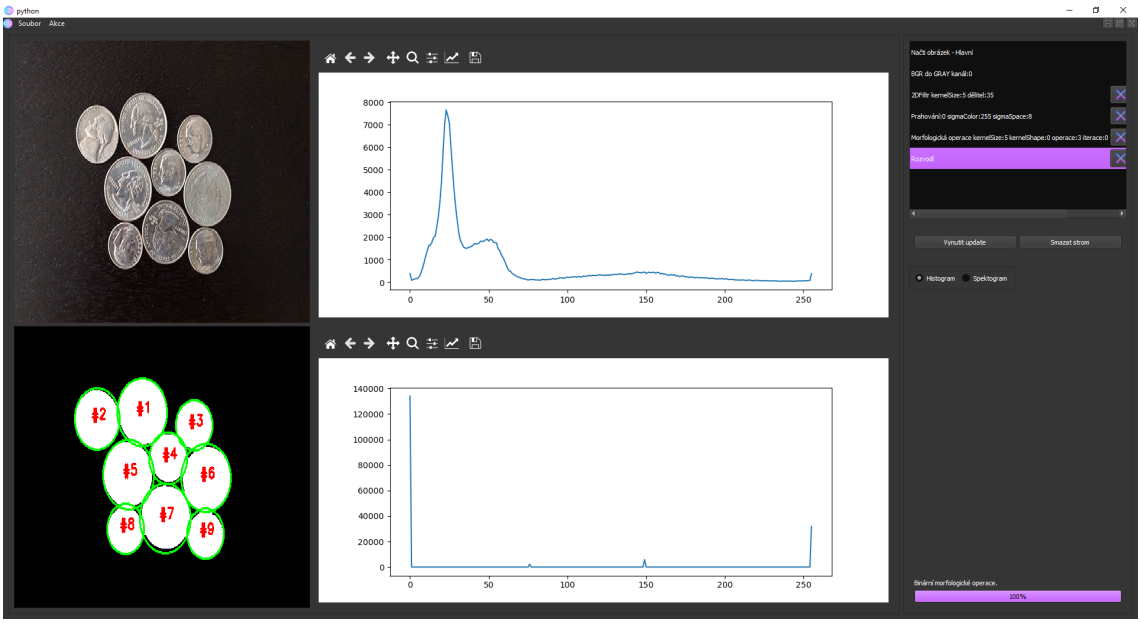


Obrázek 2.22: Průnik v bodě $\theta = 1$ a $\rho = 9.5$ značí přítomnost přímky [14].

3 Praktická část



Obrázek 3.1: Přehled všech implementovaných algoritmů a jejich požadavků na vstup. *Pro ekvalizaci histogramu se předpokládá obraz ve stupních šedi.



Obrázek 3.2: Vzhled aplikace s konkrétními provedenými operacemi. Načtení obrázku; převod do stupňů šedi z RGB (ve skutečnosti BGR z důvodu OpenCV implementace); 2D konvoluce s jádrem 5×5 a váhou 1/25; binární Otsu prahování; binární morfologická operace uzavření; rozvodí.

Projekt byl psán v jazyce Python s grafickou nadstavbou PyQt5 a doplňkovými knihovnami OpenCV, lxml, Numpy. Proběhla i rešerše dalších nástrojů pro tvorbu UI, jako jsou TKinter, PySide2. TKinter přichází nativně s Pythonem a obsahuje velice jednoduché API pro práci s ním, to je výhoda i zápor. Díky tomuto neobsahuje pokročilejší widgety[1] a jeho výchozí vzhled vypadá přinejmenším zastarale. TKinter se doporučuje spíše pro menší projekty, skripty, které potřebují mít jednoduché a rychle nasazené grafické rozhraní.

PyQt5 a PySide2 jsou poté skoro identické frameworky. Prakticky stačí změnit jmenné prostory a program by měl být zaměnitelný. Existují však mezi nimi rozdíly, které jsou rozebrány v následující kapitole 3.1.

Byla použita distribuce Pythonu, Anaconda k usnadnění instalace a práce s různými knihovnami za pomoci balíčkovacího systému conda. Visual Studio Code bylo zvoleno jako vývojové prostředí. Dále byly využity vývojářské nástroje PyQt5 frameworku Qt Designer pro rozhraní, Qt Linguist pro překlad a nástroje pro příkazovou řádku, které nemají grafické rozhraní. Celý program je verzován verzovacím systémem git na serveru GitLab.

[1] Widget - ovládací prvkek, kontrolka uživatelského rozhraní

3.1 PyQt5 a PySide2

PySide2 a PyQt5 jsou velice podobné frameworky. Jsou dokonce navrhovány tak, aby byly mezi sebou zaměnitelné. PyQt5 existuje dlouhou dobu a je podporován společností RiverBank Computing Limited, což je velice malá poradenská firma. V dobách, kdy společnost Qt patřila Nokii, se Nokie rozhodla, vytvořit si vlastní port Qt, PySide2. Velmi pravděpodobně bylo důvodem, že PyQt5 je licencováno pod licencí GPL a komerční licencí. Společnost Nokia byla odkoupena společností Microsoft a Qt, který se primárně používal pro vývoj GUI Symbianu, byl prodán společnosti Digia. Od té doby byl PySide2 podporován výhradně open-source komunitou a z velké části zaostával za PyQt5. Christian Tismer převzal práci na podpoře Qt5 pro Pyside2 v roce 2015 pro AutoDesk a vytvořil první použitelnou verzi. V roce 2016 byl projekt interně převzat společností Qt. V roce 2018 byla podpora pro PySide2 vyhlášena oficiální společností Qt. Následují rozdíly v bodech [15]:

- Licence
- Kompletnost API
- Nástroje
- Komunita
- Generátor vazeb[2]

Asi nejhlavnějším rozdílem mezi PyQt5 a PySide2 je licence. PyQt5 je vydáno pod GPL a Riverbank komerční licencí. PySide2 je k dispozici pod licencí LGPL a Qt komerční licencí. Obchodní licence PyQt5 stojí 550 $ čistého s roční podporou. Cena za komerční Qt licenci se liší v závislosti na velikosti firmy a aplikace. Při vývoji otevřeného softwaru to není zcela zásadní. Nicméně v některých případech, kdy je potřeba vytvořit open-source, který lze použít v proprietárních[3] komerčních projektech může být klíčové, že nelze použít knihovny s GPL licencí. U komerčních aplikací, záleží na tom, zda je zvoleno GPL, LGPL nebo některá z komerčních licencí. U GPL licence musí být publikován kód aplikace a co je důležitější, aplikace musí být také licencována pod kompatibilní GPL licencí. To je pro většinu komerčních aplikací nepřijatelné. Komerční licence umožňuje distribuovat aplikaci bez nutnosti jakýchkoli pokynů. To je obzvláště důležité pro vestavěná zařízení, pokud je třeba zamknout přístup k souborovému systému. Poznámka, komerční licence Riverbank se vztahuje pouze na binding PyQt5 nikoliv na samotný jazyk Qt. V případě, že aplikace musí být uzamčená, možná bude muset být zakoupena i Qt komerční licence. Pyside2 tedy vede co se týče licencí [15].

Na nástroje které jsou k dispozici oběma knihovnám se při diskuzích často zapomíná. PyQt5 přichází s celou řadou nástrojů. Jedním z nejzajímavějších je pyqtdeploy, který prohlašuje, že je schopný vytvořit aplikace pro Windows, Linux,

[2]Binding - poskytuje prostředky pro použití jiných než nativních knihoven pro daný jazyk
[3]Proprietární - také někdy označováno jako software s uzavřeným kódem, closed-source.

OS X, Android a i iOS. Navíc PyQt5 přichází s dalšími doplňky a knihovnami včetně QScintilla2 a dip. Většina těchto knihoven je zaměřena na práci s Qt widgety. PySide2 momentálně nemá žádné dedikované nástroje pro nasazení aplikace.

Pokud jde o API, PyQt5 je velmi podobný PySide2, avšak obě implementace mají své nedostatky. PySide2 je stále relativně nový, a proto některé základní funkce API včetně například qmlRegisterSingletonType stále chybí. To samé platí i pro PyQt5, kde chybí například QValidator.

Velice důležitým faktorem výběru projektu nebo jakékoli knihovny před jinou je komunita a podpora za projektem. Zde se PyQt5 a PySide2 značně liší. Z hlediska komunity má PyQt5 pravděpodobně větší komunitu. Zdá se však, že proces vývoje není příliš otevřený a je veden jedinou entitou. PySide2 ztratil většinu komunity od dob Nokie, ale protože byla vybrána jako oficiální Qt vazba pro Python, je velmi pravděpodobné, že se komunita rychle vzpamatuje. Proces vývoje je otevřen a je velmi blízko vývoji jazyka Qt. Komerční podpora pro PyQt5 je levnější než ta pro PySide2. Jak kvalitní je podpora je však diskutabilní. Společnost Qt na druhé straně může poskytovat podporu všem typům zákazníků, včetně projektového poradenství a konzultací na místě. Důležité je také si uvědomit, že podpůrná společnost stojící za PyQt5, Riverbank Computing, je malá společnost pravděpodobně o jedné osobě.

Který generátor vazeb pro Python použít nemusí být relevantní otázka, pokud je třeba použít tuto knihovnu bez přidání dalších C++ knihoven. Pokud je však plánováno přidat další C/C++ knihovny do projektu, může být výběr generátoru relevantní. PyQt5 používá SIP generátor a PySide2 Shiboken generátor vazeb. Z výsledků je nejkritičtější rozdíl v tom, že vazby generované Shiboken mohou přijmout jakýkoli druh vstupu, zatímco SIP přijímá pouze typ, který je mu známý. Kromě toho lze SIP použít pro generování Python vazeb pro C i C++ knihovny. Shiboken na druhé straně je navržen pro práci s C++.

Jak už bylo zmíněno v horní části textu PyQt5 a PySide2 jsou si velice podobné. API Pyside2 je dokonce navrženo tak, aby bylo kompatibilní s PyQt5. Naučná křivka obou frameworků je velice strmá. Obě dokumentace pro Python nejsou dokončené a často je potřeba pracovat s oficiální dokumentací jazyka Qt C/C++, která je kompletní. Pro účely tohoto projektu byl zvolen framework PyQt5, jelikož jeho komunita je opravdu prokazatelně větší a tomuto projektu limitace licencí nevadí.

3.2 OpenCV

OpenCV je knihovna s otevřeným zdrojovým kódem pro počítačové vidění. Je psána v jazyce C, C++ a běží pod systémy Linux, Windows, Mac OS X [16]. Pracuje se na dalších rozhraních pro jazyky Python, Ruby, Matlab a dalších.

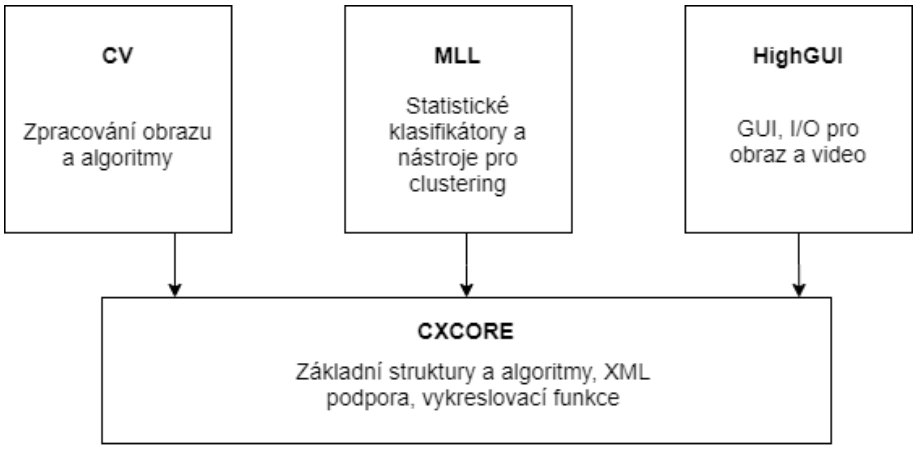
OpenCV byl navržen tak, aby byl výpočetně účinný a se silným zaměřením na aplikace v reálném čase. OpenCV je napsán v optimalizovaném jazyce C a může využívat více jádrových procesorů. Pokud je uvažováno o další automatické optimalizaci na architektuře Intel, lze dokoupit knihovny Intelu IPP (Integrated Performance Primitives), které se skládají z nízkoúrovňových optimalizovaných rutin v mnoha různých algoritmických oblastech. OpenCV automaticky používá tuto

knihovnu, pokud je nainstalována.

Jedním z cílů OpenCV je poskytnout snadnou infrastrukturu počítačového vidění, která lidem pomáhá rychle vytvářet poměrně sofistikované aplikace. Knihovna OpenCV obsahuje více než 500 funkcí, které pokrývají mnoho oblastí vidění. Protože počítačové vidění jde obvykle ruku v ruce se strojovým učením, obsahuje OpenCV také úplnou univerzální knihovnu strojového učení MLL (Machine Learning Library). Tato knihovna je zaměřena na statistické rozpoznávání vzorů a clustering. MLL je velmi užitečné pro úkoly vidění, které jsou jádrem OpenCV, ale jsou dostatečně abstraktní, aby se daly použít pro jakýkoli problém strojového učení.

3.2.1 Struktura

OpenCV je široce strukturován do pěti hlavních komponent [16], z nichž čtyři jsou zobrazené na obrázku 3.3. CV komponenta obsahuje základní zpracování obrazu a algoritmy počítačového vidění z vyšší úrovně. ML je knihovna pro strojové učení, která obsahuje mnoho statistických klasifikátorů a clustrovacích nástrojů. HighGUI obsahuje vstupní/výstupní rutiny a funkce pro ukládání a nahrávání videí a obrazů. CXCore obsahuje základní datové struktury a obsah. Obrázek 3.3 neobsahuje



Obrázek 3.3: Základní struktura OpenCV.

komponentu CvAux, která obsahuje jak zaniklé oblasti (Skrytý Markovův model, rozpoznávaní obličeje), tak experimentální metody (segmentace pozadí/popředí). CvAux není příliš dobře zdokumentovaná, dokonce chybí dokumentace. CvAux obsahuje:

- Eigen objekty - výpočetně efektivní technika rozpoznávání, která je v podstatě založena na porovnání vzorů.
- 1D a 2D Skryté Markovy modely - technika statistického rozpoznávání řešená dynamickým programováním.
- Zabudované skryté Markovy modely
- Rozpoznávání gest pomocí stereo vidění

- Rozšíření Delaunayovy triangulace, sekvence, atd.
- Stereo vize
- Shoda tvarů s obrysy regionů
- Deskriptory textur
- Sledování očí a pusy
- 3D sledování
- Hledání koster (středních linií) objektů ve scéně
- Segmentace pozadí/popředí
- Video monitorování
- Kalibrace kamery, C++ třídy

Některé z těchto metod mohou v budoucnu přejít do komponenty CV, některé pravděpodobně nikoli.

3.2.2 Portabilita

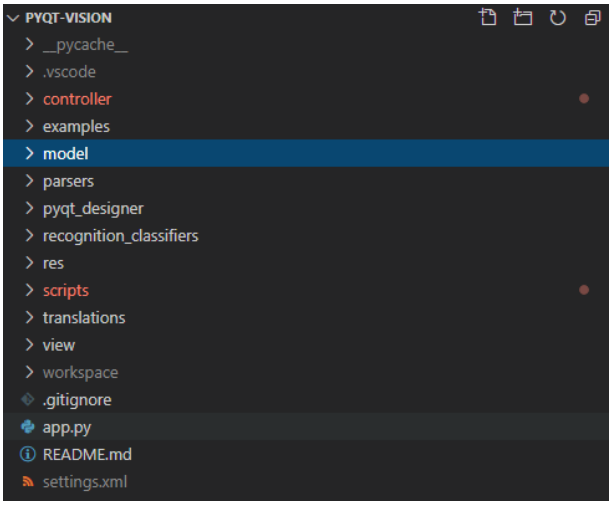
OpenCV byl navržen tak, aby byl přenosný. Původně byl vytvořen pro kompilaci napříč Borland C++, MSVC++ a kompilátory Intelu [16]. To znamená, že kód C a C++ musel být docela standardní, aby se usnadnila podpora napříč platformami. Podpora 32 bitové architektury Intel (IA32) pro Windows je nejvyspělejší následovaná Linuxem na stejné architektuře. Přenositelnost pro systém Mac OS x se stala prioritou až poté, co společnost Apple začala používat procesory Intel. Pak následuje 64 bitová podpora na rozšířených pamětích (EM64T) a 64 bitové Intel architektuře (IA64). Nejhorší je pak přenositelnost na hardwaru Sun a jiných operačních systémech.

OpenCV byl přenesen do téměř každého komerčního systému od PowerPC[4] až do robotických psů. OpenCV běží dobře na procesorech od AMD a další optimalizace dostupné v IPP využijí multimediální rozšíření (MMX) v procesorech AMD, které tuto technologii obsahují.

3.3 Struktura programu

Program byl psán podle návrhového vzoru MVC (Model-View-Controller). Oficiálně je PyQt5 však prezentováno spíše stylem Model-View. To nic nemění na věci, že přehlednější architektury MVC a separace kontroleru od vzhledu se dá docílit MVC poměrně snadno. A to sice předáním reference na požadované okno kontroleru. V tomto případě existuje právě jeden kontrolér, tedy na hlavní MainWindow, protože zbylá okna nemají žádnou interní logiku. Reagují pouze na propojené signály a

[4]PowerPC - starý název pro počítač typu Mac od společnosti Apple



Obrázek 3.4: Navrhnutá struktura programu.

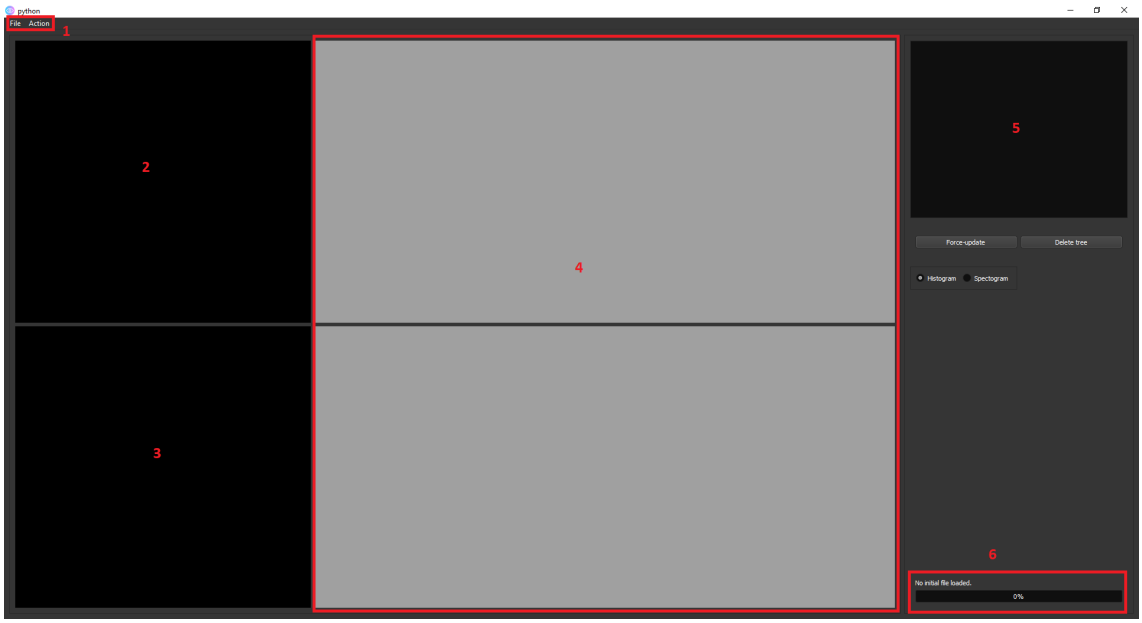
odesílají data hlavnímu kontroléru, který poté tyto napojené signály řádně zpracuje. Navrhnutý program se tedy skládá z následujících komponent:

- controller - mainWindowController logika pro view.
- examples - důležité nebo zajímavé příklady získané z různých internetových zdrojů k dokončení projektu
- model - obsahuje globální programové proměnné hlavně pro XML export, import a ucelenou definici XML tagů (programVariables). Pomocnou generickou třídu, která implemetuje vlákna QThread (generic_worker_QThread). Čtyři modely, jeden pro uložení a spravování výsledků widgetu QListWidget (operationsModel). Druhý abstraktní model (availableOperations) definuje, jak se jednotlivý implementovaný algoritmus chová, respektive co umí za operace (export, import, provedení operace, získání lokalizovaných textů atp.). Třetí (progressFlags), se stará o to jaké operace lze v konkrétním kroku spouštět (bez šedotonového obrazu nelze provádět binární morfologické operace například). Poslední (settings) drží uživatelsky nastavené hodnoty aplikace (překlady, úložné cesty).
- parsers - statickou třídu pro export a import modelu ve formátu XML (xmlParser).
- pyqt_designer - obsahuje .ui soubory formátu XML generované pomocí PyQt5 designeru.
- res - testovací obrázky pro jednotlivé algoritmy a samotné ikony programu.
- scripts - samotné implementované algoritmy z OpenCV (image_scripts) a další užitečné podprogramy (utils).

- translations - překladové soubory jak XML formátu .ts tak i binárního formátu .qm
- view - styly a vzhledy pro okna a dialogy, každá implementovaná funkce má vlastní view.
- workspace - defaultní cesta pro ukládání provedených operací (obrazů), lze tuto cestu změnit v nastavení.

Další položky v obrázku 3.4 jsou samo vypovídající. Gitignore je klasický generovaný soubor aplikací GitKraken pro jazyk Python a obsahuje vlastní pravidla pro vynechání složky workspace, XML souboru s nastavením a soubory pro Visual Studio Code. App.py je zavádějící soubor celé aplikace, který vytváří vzhled hlavního okna a nastavuje celkový tmavý vzhled aplikace. Zde je poté nutno vytvořit pomocnou proměnnou windows, do které se ukládají všechna otevřená okna. Pokud je vytvořené pouze samotné okno a jeho reference není nikde uložena GC (garbage collector) jej okamžitě zrecykluje.

3.4 Navržená aplikace



Obrázek 3.5: Vzhled a představení realizované aplikace pomocí jazyka PyQt5.

Na obrázku 3.5 je možné vidět vzhled cílové aplikace, která byla navržena knihovnou PyQt5. Většina komponent byla vygenerována PyQt5 Designerem. Tmavé barevné rozložení aplikace je realizováno však přímo v kódu. Jedná se o vlastní nadstavbu existujícího stylu fusion v tmavém režimu. Složitější vzhled jako je tento, který obsahuje několik definic rozložení prvků, se přímý zásah do kódu přímo nabízí

k zavedení chyby. Vygenerovaný kód není přehledný a je lepší ho znovu vygenerovat s požadovanými změnami. Zde je důvod proč se vyplatí odpoutat od PyQt5 Model-View architektury a rozdělit logiku do samostatné třídy.

Oblast 1 obsahuje programové menu, jehož položky jsou soubor, akce. V položce soubor je možné vytvořit nový soubor, nahrát obrázek nad kterým se budou vykonávat transformace, exportovat/importovat řetězec transformací ve formátu XML, dále otevřít dialog s nastavením, kde se momentálně nachází přepínání mezi jazykovými mutacemi, kam ukládat výsledky operací a na závěr, zda přepočítávat celý řetězec automaticky pokud se vyskytne zásadní změna v předešlých operacích (posunutí, smazání operací). Položka akce obsahuje všechny implementované metody z knihoven OpenCV, Dlib a face_recognition a dalších pomocných jako Numpy, ty budou představeny dále v 3.5.

Oblast 2 a 3 představuje vždy nahoře nahraný snímek a pod ním upravený snímek. Momentálně není v aplikaci evidován žádný snímek, nad kterým by se prováděly operace, oblasti jsou tedy prázdné. Prázdnými se stanou i v momentě, že by došlo k zásadní změně v pořadí operací (smazání, změnění pořadí). Tato změna se projeví vysláním události a změněním příslušných dat v modelu QListWidgetu tedy oblasti 5. Model má pak svůj callback propojený na view hlavního okna v kontroléru, kterým upozorní, že se změnila data a řádky v oblasti 5 výrazně obarví fialovou barvou. V tomto momentě je nutné kliknout na tlačítko Force update (pod oblastí 5) a přeložit celý model s novými operacemi, případně pokud je v nastavení povolena položka automaticky obnovovat, vše bude provedeno automaticky. Celý proces nové generace je monitorován v oblasti 6 příslušným textovým a grafickým doprovodem. Jelikož může být tento proces značně zdlouhavý, je implementován na samostatném vlákně pomocí pomocné třídy generic_worker_QThread, který implementuje základní prostý QObject z QtCore knihovny. Jedná se vlastně o základní objekt všech objektů v PyQt5. Taková třída má definované vlastní signály pro komunikaci s okolím, jmenovitě v tomto případě o dokončení, chybě, výsledku, průběhu a možnosti o přidání vlastních callbacků dle libosti. Jelikož chceme tuto třídu spouštět jako vlákno s vlastními signály, musí mít také naimplementované metody run a init. Tuto třídu pak jen stačí deklarovat na vhodném místě, připojit metodu kterou bude vlákno vykonávat a připojit metody reagující na odchozí signály a přidat takto vytvořené vlákno na QThreadpool. Zavoláním signals.start.emit() se spouští proces vykonávání. Metoda start obsahuje spousty kódu, který skutečně vytváří vlákno a nakonec volá metodu run. Při zavolání metody start tedy vzniká vlákno, které má svůj vlastní paměťový prostor a kódový prostor a poté volá metodu run. Pokud je zavolána přímo metoda run, operace vlastně poběží na hlavním grafickém GUI vlákně.

Oblast 4 zobrazuje poté k obrázkům jejich histogramy, případně spektogramy v závislost na hodnotě přepínače, který se nachází pod tlačítky a pod oblastí 5. Histogram i spektrogram je poté vždy brán a vypočítáván z šedotónového/jedno kanálového obrazu nehledě na to zda je na vstupu například klasický RGB formát. Widget, který drží tato okna, se nazývá QMdiArea a díky tomu lze s grafy interagovat, jelikož se vlastně jedná o podokno v okně. Lze tedy grafy uložit, zvětšovat, zmenšovat a analyzovat jednotlivé (x, y) hodnoty za běhu.

Oblast 5 slouží pro přehled provedených operací z hlavního menu. Konkrétní

příklad je zobrazen na obrázku 3.2. Každý jeden řádek je typ QListWidgetItem implementace vzhledu z view/listRowView.py, který je poté přidáván do widgetu QListWidget. Na tento QListWidget je právě připojen model operationsModel, který umožňuje na základě události vyvolaného z tlačítka řádku reagovat posunutím nahoru, dolu, smazáním nebo upravením příslušných dat.

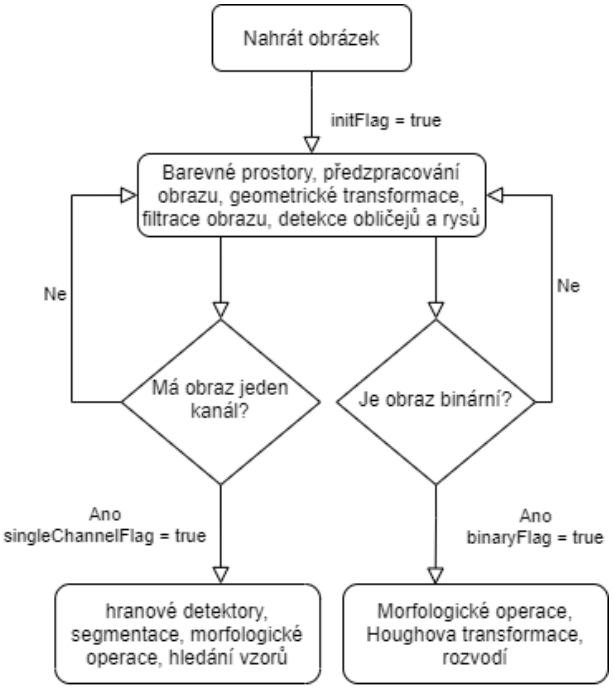
Oblast 6 slouží k notifikaci uživatele co právě probíhá, typicky obsahuje chybové hlášky, případně průběh nové kalibrace dat na samostatném vlákně.

Jak celá obsluha aplikace vypadá? Uživatel nahraje obrázek, klikne na jakoukoliv dostupnou akci v menu, vygeneruje se nové okno na základě přiděleného signálu položky a její obsluhy a okno vyskočí navrch. V praxi to znamená, že každá operace (až na velice parametrově podobné) má svůj vlastní vzhled. Zde je problematické udržet okna „naživu"a zde také přichází trik, který byl zmíněn v předešlé kapitole. Pokud je vytvořeno samostatné nové okno mimo hlavního a jeho reference není uložena, tak GC automaticky nově vytvořené okno smaže z paměti. Této drobnosti se těžko dohledává, protože někdy ani probliknutí není viditelné. Celá tato problematika je ošetřena uložením reference okna do pole. Ve skutečnosti by postačila jakákoliv datová struktura. Zbytek procesu funguje následovně: uživatel zadá parametry; kliknutím na tlačítko „OK"se vyvolá na straně otevřeného okna signál; příslušný kontrolér na tento signál reaguje a provede požadovanou operaci; vytvoří se nový záznam do modelu a přidá se řádek QListWidgetItem do QListWidget. V případě chybných parametrů ze strany uživatele dojde k vytvoření chybového okna dle definice errorDialogView. To obsahuje chybovou hlášku, funkci, soubor, řádek, kód a podrobný popis chyby. Pokud nastane chyba jsou informace od uživatele zahozeny a celý proces se musí opakovat.

3.5 Implementované algoritmy

Jednotlivé algoritmy obsahující implementace operací z knihoven OpenCV, Dlib a face_recognition jsou implementované v souboru scripts/image_scripts.py. Jedná se o statickou třídu se statickými metodami, které jsou natolik samostatné, že je lze se správnými parametry spouštět z příkazové řádky a tak byla víceméně taktéž testována samotná funkcionalita. Povinným atributem každé metody je vždy cesta ke vstupnímu obrazu zjednodušeně. Další parametry které jsou vždy přítomny avšak jsou nepovinné, jsou cesta kam výsledek uložit, název souboru a úložný formát (mimo pomocných metod typu testování, zda je obraz ve stupních šedi nebo finalizace operace). Výše tři uvedené parametry mají tovární hodnoty None, None a textový řetězec png. Existuje zde privátní statická metoda _finalize(), která se vyskytuje na konci každého algoritmu a v závislosti zda jsou parametry úložná cesta, název souboru, úložný formát vyplněny či nikoliv se provádí rozhodnutí, zda výsledný obraz zobrazit prostým přímým vykreslováním cv2.imshow(), či ho uložit na disk.

Logicky v závislosti na složitosti operace má potom každá svá metoda proměnný počet dalších vstupních parametrů mimo výše uvedených. Hodnoty jsou předávány metodám za pomoci signálů knihovny PyQt5. Návratové hodnoty implementovaných metod jsou cesta k upravenému obrazu a parametr error. Na parametr error se



Obrázek 3.6: Omezení operací na základě předem provedených výsledků.

reaguje v kontroléru přehledným zobrazením chybové hlášky ve vyskakovacím okně, jak už bylo řečeno.

```
@staticmethod
def imageFaceRecognitionModule(input, saveLocation=None, fileName=None, extension="png"):
    error = None
    try:
        image = cv2.imread(input)
        face_locations = face_recognition.face_locations(image)
        # loop over the recognized faces
        for i, (top, right, bottom, left) in enumerate(face_locations):
            # draw the predicted face name on the image
            cv2.rectangle(image, (left, top), (right, bottom), (0, 255, 0), 2)
            y = top - 15 if top - 15 > 15 else top + 15
            cv2.putText(image, f'#{i}', (left, y), cv2.FONT_HERSHEY_SIMPLEX,
    0.75, (0, 255, 0), 2)
    except Exception as e:
        error = e
    if(error == None):
        return ImageScripts.__finalize(saveLocation, fileName, image), error
    else:
        return None, error
```

Obrázek 3.7: Výřez z Visual Studio Code. Konkrétní implementace algoritmu na vyhledávání tváří pomocí knihovny face_recognition, která používá funkce Dlib. Tato metoda vyžaduje pouze jediný povinný atribut a tím je cesta ke vstupnímu obrazu. Jedná se tedy o příklad metody s minimálním počtem atributů, který je v dosavadní aplikaci možný.

Výčet implementovaných algoritmů lze nalézt na obrázku 3.1. Operace jsou dále

zamknuty v závislosti na použitých předešlých metodách 3.6. Například je zbytečné povolovat k použití metody hranových detektorů, pokud není k dispozici šedotónový respektive jedno-kanálový obraz. Omezení jsou dána v třídě model/progressFlags.py. Některé operace typu segmentace a Cannyho hranový detektor nutně mění obraz na binární.

3.6 Škálovatelnost systému

Byl brán ohled na případnou rozšířitelnost celého systému o další nespočet neimplementovaných algoritmů. Zjednodušeně lze říci, že k implementaci nového algoritmu je potřeba:

1. Vytvořit vzhled okna pro metodu se vstupními parametry.
2. Implementovat logiku algoritmu do scripts/image_scripts.py
3. V model/availableOperations.py definovat chování metody při exportu/importu objektu a dalších metod dle abstraktní třídy basicOperation.
4. V kontroléru, napojit signály a zajistit vytvoření okna.
5. Vytvořit nové lokalizace (volitelné).

Vytvořit vzhled okna je jednoduchý krok, neboť se dají částečně znovu použít předem vytvořená okna pro operace zmíněné v kapitole 3.5. Jelikož se každá operace liší proměnným počtem vstupních hodnot a i kontrolky, ze kterých je potřeba načítat uživatelská data, byla snaha o vytvoření dostatečně abstraktního okna rychle opuštěna. Pokud člověk dobře zná jednotlivé metody a případně kolik mají vstupních parametrů, je tvorba nového rozhraní pomocí PyQt5 designera bezproblémová. Výstupem PyQt5 designera je XML soubor typu ui. Ten se převádí do Python kódu za pomoci nástroje pyuic5, který přichází s distribucí PyQt5. Celý příkaz ve vhodném prostředí je následující pyuic5 -soubor. Vygenerovaný Python kód je poté potřeba drobně upravit. V závislosti na počtu tlačítek je třeba importovat z PyQt5.QtCore pyqtSignal a definovat všechny možné signály pro komunikaci s kontrolérem. Přidat inicializační metodu, která obsahuje metodu setupUi(). A naposled třída bude dědit od QObject na rozdíl od object. Pokud třída obsahuje combo box[5], pak je třeba jej zde naplnit, nelze učinit tak z PyQt5 designera. Definice pro tyto prvky se nachází v models/programVariables.py a jsou ve formě slovníku. Klíčem je zobrazený text a hodnotou je často skrytá hodnota integer za výčtovým typem. Příklad páru by bylo například $\{"GRAY" : cv2.COLOR\_BGR2GRAY\}$.

V kapitole 3.5 byla podrobně vysvětlena struktura jednotlivých algoritmů. Je potřeba dodržet existenci parametrů input, saveLocation, fileName, extension. Na konci nové metody se poté předají tyto parametry privátní statické metodě v též třídě zvané _finalize(saveLocation, filename, image, extension), která řeší problematiku

[5]Combo box - česky jako kombinované pole.

zakončení výsledku jak už název napovídá. Konkrétně tedy na základě vstupních parametrů buďto obraz uloží na disk, nebo předpokládá, že operace je volána samostatně a pouze jí tedy vykreslí pomocí metody cv2.imshow(). Poslední nutností je zachovat výstup ve formátu výstupní obraz, chyba. Nebude-li tak dodrženo, nelze pak uživateli přikládat přehledné údaje o chybě. Jelikož je každý algoritmus jedinečný a zamezit všem vstupním chybám od uživatele je pracné, doporučuje se provádět celou kritickou část v try-catch bloku.

V souboru availableOperations.py se nachází předpis dle třídy basicOperation, jak musí být definována nová třída pro nový algoritmus, tak aby jí bylo možné správně začlenit do systému. Všechny následující operace (třídy) dědí od třídy basicOperation. Tato základní třída vlastně představuje též načtení obrazu. Nachází se zde definice pro export/import ve formátu XML, jak spouštět metodu při vícevláknovém zpracování, jaké parametry editovat v případě editace, lokalizované textové řetězce pro QListWidget a chybové hlášky.

V kontroléru zbývá pak na příslušná tlačítka a jejich signály připojit správné metody, které říkají, jaké konkrétní okno otevřít a tomuto oknu namapovat signál dialogSuccesSignal na implementovanou operaci, kterou požadujeme, aby byla obsloužena.

Volitelným krokem je přidat všechny nové textové řetězce do souboru eng-cz.ts a ten pak konvertovat do binární podoby a vytvořit tak překlad. Tento proces je obsáhle vysvětlen v kapitole 3.9. Jedná se o volitelný krok, často lze některé překlady znovupoužít.

3.7 PyQt5 vlákna

Vzhledem k výpočetní složitosti některých algoritmů z kapitoly 3.5 typu detekce Dlib se celá aplikace může stát neodpovídající, neboli zamrzne z důvodu zahlcení hlavního grafického vlákna, na kterém běží vzhled PyQt5. Kdyby aplikace neobsahovala tyto rozpoznávací algoritmy a spol., šlo by jí navrhnout uspokojivě bez použití vláken, jelikož nebylo možné si neresponzivity do této doby všimnout. Přesto však bylo výhodné se touto cestou vydat, neboť šli bychom do extrémních případů, kde by byly sériově spojeny desítky algoritmů za sebou, nastal by ten samý problém, například při snaze změnit prostřední uzel.

Prvotní experiment spočíval v použití rozhraní QRunnable z QtCore a spouštěním tohoto workera na QThreadPool. Byla vytvořena nová třída Worker, která dědila od QtCore.QRunnable. Tato třída musela mít tedy povinně definované metody init() a run(). Byla definována i pomocná třída WorkerSignals, která poskytovala definice signálů pro komunikaci s okolím typu o dokončení, výskytu chyby, výsledku a průběhu. Legitimní myšlenka, nicméně s drobným zádrhelem, ztráta možnosti ukončit tuto operaci. Jakmile byla operace spuštěna, nebyla možnost jí jakkoliv jinak ukončit než dokončením. QRunnable zkrátka nebylo dosti flexibilní, a proto bylo nakonec použito třídy QThread.

Následoval tedy přechod na QThread, který vyžadoval prakticky změnu třídy od které bylo děděno a to tedy na QtCore.QObject. Zbytek kódu zůstal nezměněn. S tímto

objektem lze tedy vlákna ukončovat a dotazovat se zda jsou stále aktivní. Projekt obsahuje obě implementace ve složce model. Tato třída je dostatečně abstraktní na spouštění libovolného počtu vláken. Stačí vlastně připojit jen správné metody na signály.

Stejný problém, který však nebyl vyřešen, se vyskytl i u vykreslování histogramu a spektrogramu. Jejich přepínání viditelně pozastavuje hlavní GUI vlákno. Zde je však problém, že vlákno QThread nemůže používat objekty GUI k vykreslování a malování obrazu.

3.8 Export a import operací

Pro usnadnění práce byla použita knihovna lxml. Místo použití této knihovny se však naskytovala ještě jedna možnost a tou je ElementTree. ElementTree je součástí standardní knihovny Python, která zahrnuje další typy datových modulů, jako csv nebo json. To znamená, že modul je přítomen v každé instalaci Pythonu. Pro většinu běžných XML operací, včetně vytváření stromové struktury dokumentů a jednoduchého vyhledávání a parsování atributů, hodnot uzlů dokonce i jmenných prostorů, je ElementTree dostačující.

Lxml je knihovnou třetí strany a vyžaduje instalaci. V mnoha ohledech lxml dokonce rozšiřuje ElementTree, protože je k dispozici většina operací. Hlavním rysem tohoto rozšíření je, že lxml podporuje XPath 1.0 i XSLT 1.0. Kromě toho může lxml parsovat HTML dokumenty, které nejsou kompatibilní s XML, a proto se používá i pro web scraping operace a je dokonce jako parser v BeautifulSoup a knihovně Pandas. Mezi další užitečné vlastnosti lxml patří pretty_print výstup, objectify a SAX podpora.

Xpath bylo výhodné v projektu využít. Nebylo důvodu tedy této knihovny nevyužít. Momentální podpora pro export je ve formátu XML. Přidáním dalšího parseru by nevyžadovalo zásadní zásah do kódu, pouze vlastně definovat nové chování parsování. Části kódu které definují chování procesu XML exportu/importu, jsou definovány na třech místech:

- controller/mainWindowController.py
- parsers/xmlParser.py
- model/availableOperations.py

Příklad bude vysvětlen na XML parseru. V prvním bodě se opět jedná o propojení vhodného signálu tlačítka z menu a metody pro parsování. V druhém bodě je třeba definovat tyto metody export a import. Metoda export pracuje nad každou použitou operací v programu a definice jaké elementy má metoda export použít pro jednotlivou operaci, jsou definovány právě v posledním bodě souboru availableOperations. Všechny použité elementy pro export se nachází v model/programVariables.py. Metoda import pracuje potom se stejnými elementy a jednoduše tedy v případě potřeby lze změnit pojmenování. Hledání elementů probíhá na základě Xpath. Každá operace

má v availableOperations.py dáno, jaké elementy má hledat pomocí Xpath v metodě import/export.

```xml
<?xml version='1.0' encoding='UTF-16'?>
<actions>
  <_action type="basicOperation">
    <start_path>C:/Users/sneti/Documents/GitLab/pyqt-vision/res/lenna.png</start_path>
    <end_path>C:/Users/sneti/Documents/GitLab/pyqt-vision/res/lenna.png</end_path>
  </_action>
  <_action type="colorSpaceOperation">
    <start_path>C:/Users/sneti/Documents/GitLab/pyqt-vision/res/lenna.png</start_path>
    <end_path>workspace/2347797.png</end_path>
    <original_colorspace>BGR</original_colorspace>
    <changed_colorspace>GRAY</changed_colorspace>
  </_action>
  <_action type="simpleThresholdingOperation">
    <start_path>workspace/9681025.png</start_path>
    <end_path>workspace/206193.png</end_path>
    <threshold_value>120</threshold_value>
    <maxVal>255</maxVal>
    <method>0</method>
  </_action>
  <_action type="morphologyOperation">
    <start_path>workspace/206193.png</start_path>
    <end_path>workspace/312456.png</end_path>
    <kernel_size>5</kernel_size>
    <kernel_shape>0</kernel_shape>
    <method>-91</method>
    <iterations>0</iterations>
  </_action>
</actions>
```

Obrázek 3.8: Ukázka exportovaných operací z aplikace.

Kořenovým elementem je zde actions. Ten může mít neomezené množství elementů _action. Každá akce má vždy atribut type, který udává o jakou operaci se jedná. Vnořený element akce povinně musí obsahovat vstupní cestu obrazu start_path a výstupní cestu obrazu end_path. Ostatní elementy a atributy jsou kompletně závislé na tom, jak je jejich export definován a jaké parametry vlastně exportovat v souboru availableOperations.py. Je esenciální zkontrolovat, že importované hodnoty jsou čísla a ne textové řetězce tam kde je potřeba (většina případů). Pro tuto problematiku existuje pomocná metoda string2Number v souboru utils. Jejím jediným vstupním parametrem je textový řetězec. Metoda navrací hodnotu z funkce literal_eval z modulu ast. Tato metoda slouží k bezpečnému vyhodnocení řetězců obsahujících výrazy Pythonu z nedůvěryhodných zdrojů bez nutnosti analyzovat hodnoty ručně. Narozdíl od metody eval, je tato metoda bezpečná a nelze podvrhnout jiná data do aplikace.

3.9 Dynamický překlad a nastavení

Nastavení celé aplikace používá jednoduchý model o třech položkách, jazyk aplikace, kam uložit pracovní soubory a zda automaticky přepočítávat změny bez nutnosti klikat na tlačítko. Soubor je ve formátu XML a předpokládá se název settings.xml. Aplikace se snaží tento soubor nahrát při každém spuštění, pakliže neexistuje, vytvoří se výchozí s hodnotami anglický jazyk, ukládání do složky s aplikací workspace a automatické přepočítávání je povoleno.

Překlad je poměrně problematický z hlediska časové náročnosti a není řešen úplně nejlépe (proti WPF například). K PyQt5 existuje aplikace jménem Qt Linguist, ta je

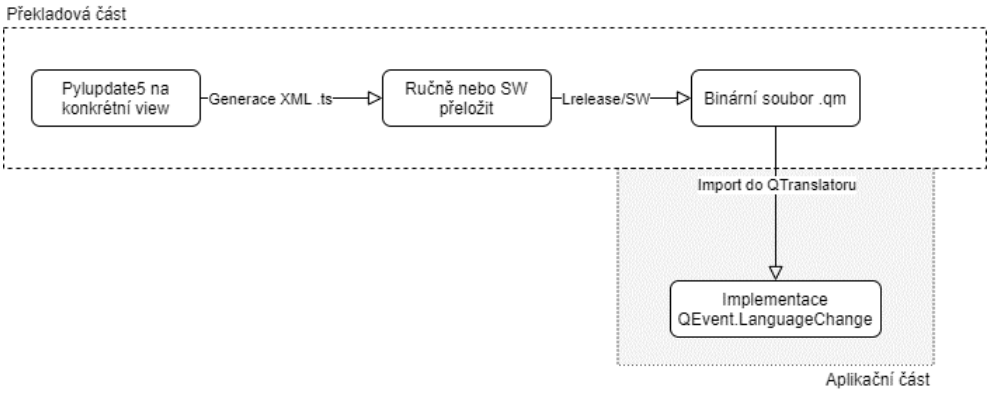
```
<context>
    <name>StatusLabels</name>
    <message>
        <location filename="availableOperations.py" line="21"/>
        <source>Basic operation.</source>
        <translation>Základní operace.</translation>
    </message>
    <message>
        <location filename="availableOperations.py" line="81"/>
        <source>Color space changing.</source>
        <translation>Změna barevného prostoru.</translation>
    </message>
```

Obrázek 3.9: Generovaný soubor .xml programem Qt Linguist, který přichází s distribucí PyQt5. Tento soubor je nakonec překládán do binární podoby a instalován do aplikace pomocí globální třídy QTranslator.

ale spíše pro laiky, kteří se nesetkali s XML a tedy čistým překladatelům. Neexistuje možnost jak přidat nový záznam o překladu. Nový záznam jde přidat pouze ručně do XML a nebo voláním z příkazové řádky pylupdate5 "python soubor"-ts "výstupní XML soubor". Zde se vyskytují dva problémy. Vygenerovaný XML soubor má zbytečné atributy právě pro lepší čitelnost v programu Qt Linguist. A druhým problémem je právě to, že pokud existuje více pohledů a rozhraní, musí se překlad dělat buďto kompletně ručně, zásahem do již nějakého generovaného překladového souboru a nebo vytvořením několika překladových souborů a ty poté skloubit do jednoho. Pro spojení několika takových souborů bez duplicit nebylo nalezeno řešení a pravděpodobně neexistuje podle dokumentace. Zbývalo tedy kopírovat z jednoho zdroje do druhého a duplicity manuálně odstranit.

XML formát popisu překladu je ukázán na obrázku 3.9. Kořenový element je vždy context a za ním musí následovat element name. Ten udává kontext, ke kterému zpráva patří (první ze vstupních parametrů metody translate). Za ním může následovat nekonečně mnoho elementů message. Vnořený element lokace je nepovinný a má využití pouze pokud je využíváno Qt Linguist, který poté dokáže poskytnout náhled. Zbylé dva elementy source a translation jsou povinné a udávají originální text a jeho překlad.

Samotný překlad už je poté otázkou vygenerování binárního souboru z XML zdroje, který používá právě PyQt5. To se dělá pomocí příkazové řádky lrelease "xml zdroj.ts"nebo v právě v PyQt Linguist. Překladové soubory se nachází ve složce translations. Přímo v mainWindowView.py se poté připojí správný binární translační soubor .qm pomocí třídy QTranslator, či se odpojí. Aplikace je momentálně psaná kompletně v angličtině, tedy anglicko-český překlad se připojuje. QTranslator je globální v celé aplikaci a v jednotlivých požadovaných místech překladu stačí volat metodu QtWidgets.QApplication.translate(kontext, zdroj textu).



Obrázek 3.10: Postup s konkrétními kroky pro vytvoření jednoho překladového souboru, který může být poté použit v aplikaci.

4 Závěr

V rámci této diplomové práce byl vytvořen systém v jazyce Python a frameworku PyQt5 pro definici uživatelského grafického rozhraní. Tento systém obsahuje algoritmy z počítačového zpracování obrazu za pomoci knihoven OpenCV, Dlib, Numpy, face_recognition a dalších a je dostatečně otevřený pro rozšíření o další funkce. Každá operace nad obrazovými daty je zaznamenána. Operace se dají mazat, měnit nebo posouvat jejich pořadí v řetězci událostí. Tento model operací je tedy potom možné exportovat/importovat ve formátu XML díky knihovně lxml. Bylo počítáno i s jazykovými mutacemi, které jsou řešeny dynamicky za běhu programu. Momentálně program podporuje angličtinu a češtinu. Přidání další jazykové podpory je pouze o tom, vytvořit správný translační soubor a v programu ho zaregistrovat. Program je dále vícevláknový a operace nelze používat bez předem splněných požadavků na vstup. Na závěr byla vytvořena nápověda k funkcím ve formě statických webových stránek.

Při připravování Python prostředí v rámci balíčkovacího systému Conda, nastal notorický problém se zprovozněním knihovny OpenCV, kde některé vydané verze zkrátka nefungují, ačkoliv jsou dle Condy nainstalovány správně. S problémem se potýkám už několik let, vždy když se vrátím k práci s touto knihovnou. Typicky potom fungují verze gcc7 nebo broken, případně když už nic jiného nezbývá, tak zvolit defaultní balíčkovací systém Pythonu, pip. Ten funguje většinou vždy, jelikož pip může využívat kompilátory cílového stroje. Nevýhodou takového řešení je, že pip si nekontroluje závislosti mezi balíky a tak rychle může nastat tzv. dependency hell.

Celý systém byl na poněkolikáté přepisován, aby odpovídal požadavkům zadání. Postupem času, jak se projekt vyvíjel se ukázalo, že původní návrh je zkrátka nedostačující. Nejsložitějším úkolem bylo vyřešit, jak spojit několik těchto naprosto odlišných metod do jakéhosi celku, nad kterým se dá jednotně pracovat a manipulovat. Zároveň zajistit čitelnost a jednoduchost pro dodatečné budoucí rozšiřování a úpravy. Bylo důležité zodpovědět též otázku, kdy už je zkrátka systém dost čitelný, neboli otevřený a přestat se zabývat jeho strukturou a začít implementovat jednotlivé algoritmy a ostatní funkcionality.

Grafické rozhraní se vytváří přes Qt Designera, který přijde s balíčkem PyQt5 a poté se pomocí příkazové řádky a příkazu pyuic5 překonvertuje do Python kódu z jazyka Qt. Dělat nějaké rozsáhlejší změny přímo uvnitř generovaného kódu složitějšího rozvržení je spíše na škodu než k užitku a je lepší otevřít designera a celý proces provést znovu s požadovanými změnami. Tedy samotný vygenerovaný kód je dostatečně komplexní a trvá dlouho se v něm zorientovat. Kód musel být stejně vždy dopsán ručně, jelikož některé parametry zkrátka nejsou přes grafické prostředí dostupné. Je

výhodné zakomponovat načtení rozhraní okna setupUi() přímo do inicializační části třídy (defaultně odděleno). Základní události neboli signály se dají nastavit uvnitř grafického rozhraní. Při pokročilejších nebo vlastních uživatelských událostí, které obsluhují metody je třeba zasáhnout do kódu. Diskutabilní je poté někdy i matoucí překlad Qt designera typu „upravit kamarády"nebo „krabice s prvky". Není třeba znovu „vynalézat kolo"a původní anglický překlad by byl optimální.

U překladu aplikace nebyla zjištěna cesta jak z více předpisů oken udělat jeden velký soubor s překlady. Nejjednodušším způsobem bylo zkrátka vygenerovat pro každé rozhraní jeden překlad a tyto dílčí překlady pak skloubit do jednoho velkého překladového souboru. Zde může nastat problém s duplicitami. PyQt Linguist, který má pomáhat s překladem, je užitečný nástroj ale chybí mu možnost přidávat vlastní nové překlady. Slouží tedy spíše pro pracovníky, kteří se zabývají čistým překladem aplikace. Přesto byl nakonec využit, neboť dokáže právě upozornit na duplicity a zlehčuje generaci finálního binárního souboru pro aplikaci, kdy není třeba spouštět příkazovou řádku.

V neposlední řadě bylo taktéž výzvou zvolení vhodné vícevláknové technologie. Teoreticky by se dalo využít nativních Python tříd pro vícevláknové/paralelní zpracovaní. Nedoporučuje se však kombinovat PyQt5 a Python těchto technologií. Druhým důvodem je to, že PyQt5 funguje na bázi signálů a to je logicky něco co Python neumí. Na výběr tedy byly dvě možnosti QThread a QRunnable. Jako první byla vyzkoušena druhá možnost, jelikož je jednodušší na implementaci. Ta se bohužel ukázala jako nedostačující, byla málo flexibilní. Zde nelze nepochválit kvalitní dokumentaci jazyka Qt, která obsahuje definitivní tabulku, kterou z technologií použít. Pakliže bylo vyžadováno vlákna, které poběží po celou dobu života hlavního vlákna a bude ovládáno signály dle dokumentace, je vhodné použít QThread. Toto řešení se nakonec projevilo jako uspokojivé a bylo implementováno.

Závěrem vstup do grafického světa nadstaveb PyQt5 a PySide2 a jazyka Qt je zprvu velice strmý a matoucí, jelikož některé postupy zkrátka nefungují tak jak by se očekávalo. Oficiální dokumentace pro binding PyQt5 prakticky neexistuje a je tedy třeba alespoň trochu rozumět syntaxi jazyka C++. Jazyk Python a grafická nadstavba PyQt5 se jeví jako nejlepší způsob, jak integrovat knihovnu OpenCV a další knihovny s ní související a prezentovat je prostřednictvím grafického rozhraní. Z vlastních zkušeností například zkoušet cestu grafického rozhraní pomocí .NET frameworku WPF a tu kombinovat s wrapperem pro Python nebo přímo wrapperem pro OpenCV bylo k datu 2020 velice nevýhodné a nepříjemné (zastaralé, teprve ve vývoji, neudržované knihovny).

Literatura

1. DAVIES, E. R. *Computer and machine vision, fourth edition: theory, algorithms, practicalities*. 4th. USA: Academic Press, Inc., 2012. ISBN 978-0-12-386908-1.
2. SZELISKI, Richard. *Computer vision: algorithms and applications*. 1st. Berlin, Heidelberg: Springer-Verlag, 2010. ISBN 978-1-84882-934-3.
3. FAIRCHILD, Mark D. *Color appearance models*. John Wiley & Sons, 2013. ISBN 978-1-118-65310-4. Google-Books-ID: 1BT9R6FjVhIC.
4. GOMES, Jonas; VELHO, Luiz. *Image processing for computer graphics* [online]. New York: Springer-Verlag, 1997 [cit. 2021-03-07]. ISBN 978-1-4757-2745-6. Dostupné z DOI: 10.1007/978-1-4757-2745-6.
5. HALL, Roy. *Illumination and color in computer generated imagery* [online]. New York: Springer-Verlag, 1989 [cit. 2021-03-07]. Monographs in Visual Communication. ISBN 978-1-4612-8141-2. Dostupné z DOI: 10.1007/978-1-4612-3526-2.
6. DAM, Andries Van; FEINER, Steven K.; HUGHES, John F. *Computer graphics: principles and practice*. 2nd edition. Ed. FOLEY, James D. Reading, Mass: Addison-Wesley Professional, 1995. ISBN 978-0-201-84840-3.
7. MARR, D.; HILDRETH, E.; BRENNER, Sydney. Theory of edge detection. *Proceedings of the Royal Society of London. Series B. Biological Sciences* [online]. 1980, roč. 207, č. 1167, s. 187–217 [cit. 2021-03-16]. Dostupné z DOI: 10.1098/rspb.1980.0020. Publisher: Royal Society.
8. TOMASI, C.; MANDUCHI, R. Bilateral filtering for gray and color images. In: *Sixth International Conference on Computer Vision (IEEE Cat. No.98CH36271)* [online]. Bombay, India: Narosa Publishing House, 1998, s. 839–846 [cit. 2021-03-15]. ISBN 978-81-7319-221-0. Dostupné z DOI: 10.1109/ICCV.1998.710815.
9. PARIS, Sylvain et al. Bilateral filtering: theory and applications. *Foundations and Trends® in Computer Graphics and Vision* [online]. 2008, vol. 4, no. 1, s. 1–75 [cit. 2021-03-15]. ISSN 1572-2740, ISSN 1572-2759. Dostupné z DOI: 10.1561/0600000020.
10. PARIS, Sylvain et al. A gentle introduction to bilateral filtering and its applications. In: *ACM SIGGRAPH 2007 courses* [online]. New York, NY, USA: Association for Computing Machinery, 2007, 1–es [cit. 2021-03-31]. SIGGRAPH '07. ISBN 978-1-4503-1823-5. Dostupné z DOI: 10.1145/1281500.1281602.

11. HLAVÁČ, Václav; SEDLÁČEK, Miloš. *Zpracování signálů a obrazu* [online]. 2. vyd. Praha: Vydavatelství ČVUT, 2007 [cit. 2021-03-16]. ISBN 978-80-01-03110-0. Dostupné z: https://search.mlp.cz/cz/titul/zpracovani-signalu-a-obrazu/2443225/.

12. CANNY, J. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*. 1986, roč. PAMI-8, č. 6, s. 679–698. ISSN 1939-3539. Dostupné z DOI: 10.1109/TPAMI.1986.4767851. Conference Name: IEEE Transactions on Pattern Analysis and Machine Intelligence.

13. JORAM, Nickson. *Morphological operations in image processing* [Medium] [online]. 2020-01-01 [cit. 2021-03-17]. Dostupné z: https://himnickson.medium.com/morphological-operations-in-image-processing-cb8045b98fcc.

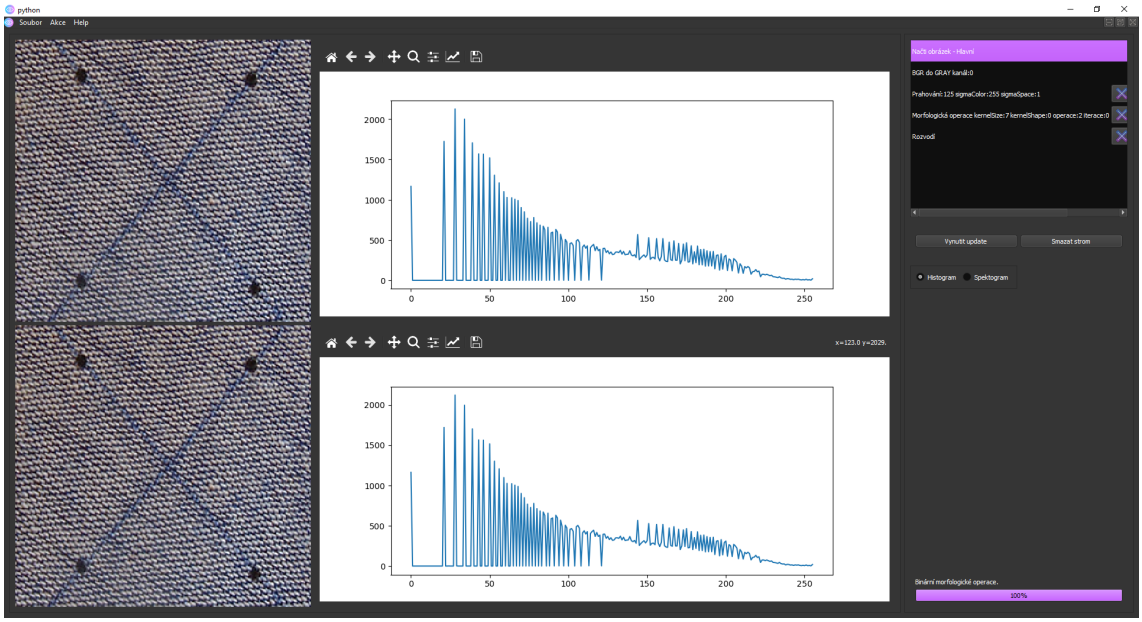
14. KRUTIKA, Bapat. *Hough transform using OpenCV — Learn OpenCV* [online]. 2019-03-19 [cit. 2021-03-17]. Dostupné z: https://learnopencv.com/hough-transform-with-opencv-c-python/.

15. 2019, Alex. *PyQt vs Qt for Python (PySide2)* [Machine Koder] [online]. 2019-02-19 [cit. 2021-03-19]. Dostupné z: http://machinekoder.com/pyqt-vs-qt-for-python-pyside2-pyside/.

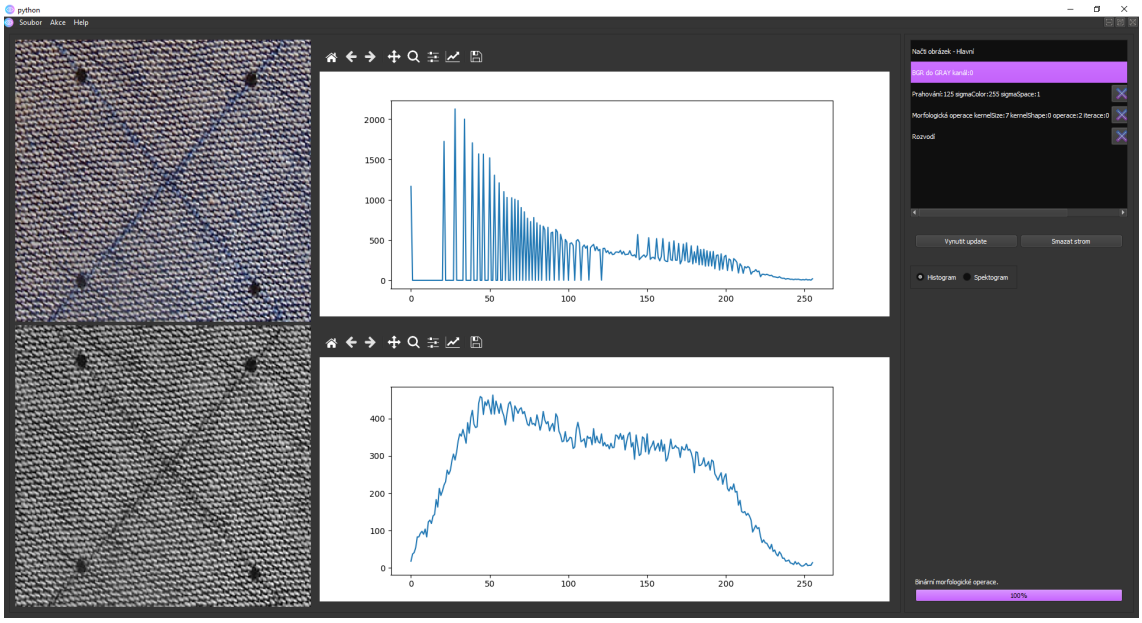
16. BRADSKI, Gary; KAEHLER, Adrian. *Learning OpenCV: computer vision in C++ with the OpenCV library*. 2nd. O'Reilly Media, Inc., 2013. ISBN 978-1-4493-1465-1.

Přílohy

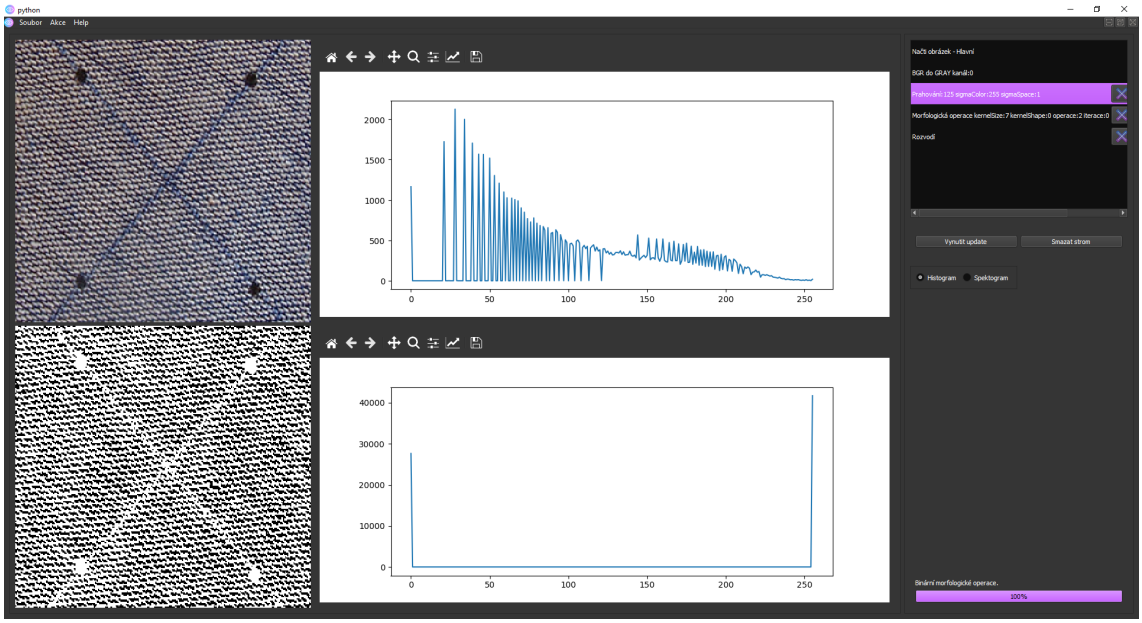
A První ukázka segmentace



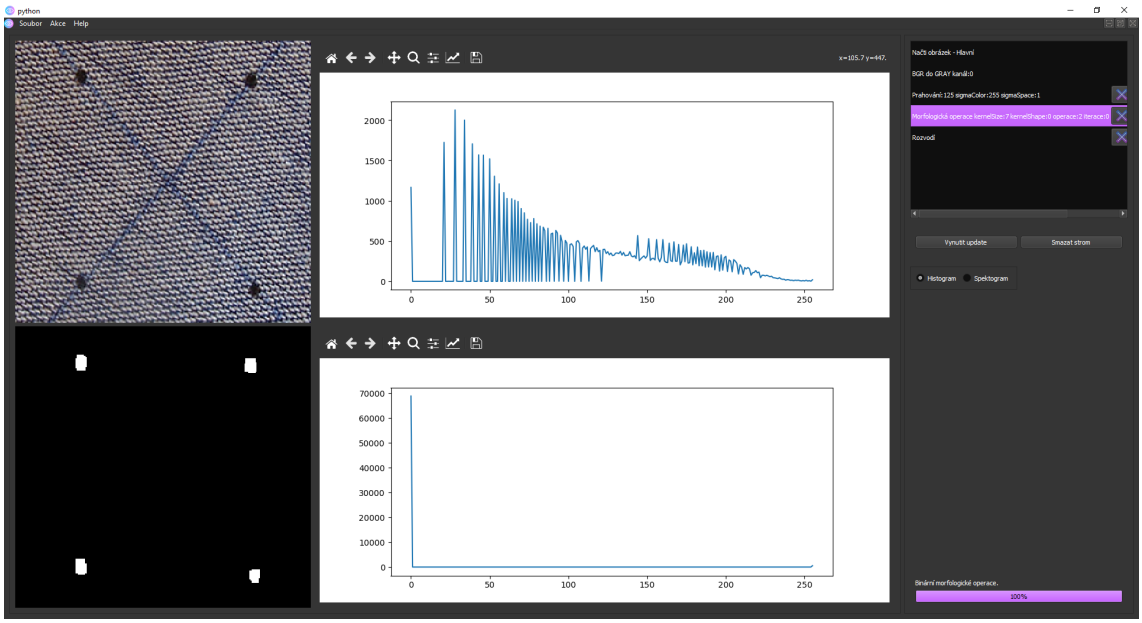
Obrázek 4.1: Operace načtení obrazu v navržené aplikaci.



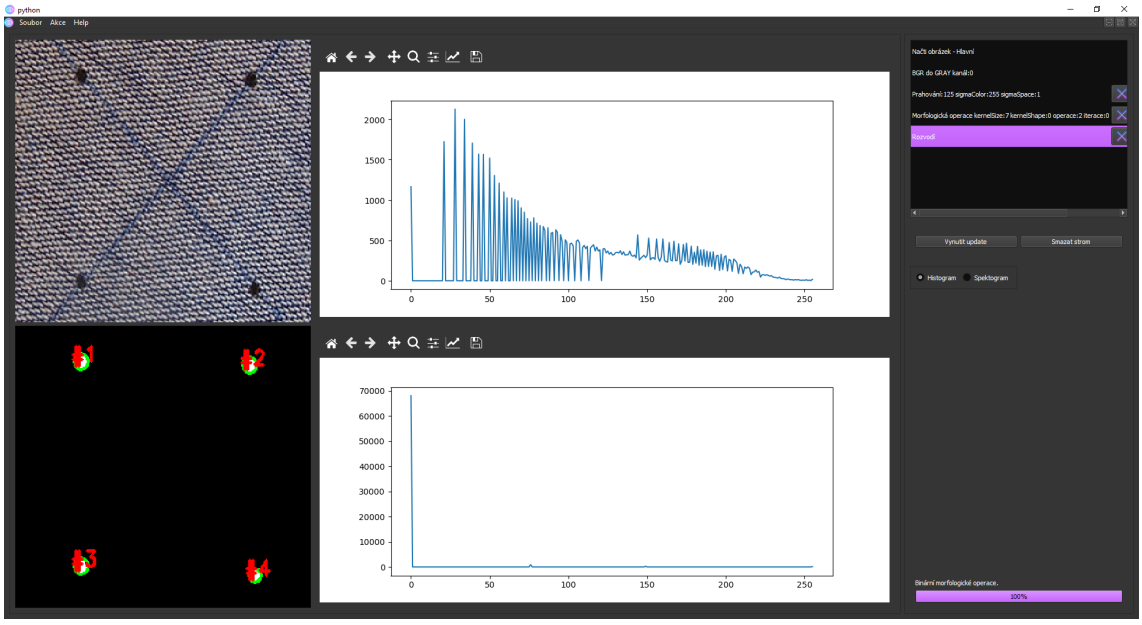
Obrázek 4.2: Operace převod vstupního obrazu do stupňů šedi v navržené aplikaci.



Obrázek 4.3: Operace prahování s parametry prahování hodnoty pixelu 125 na hodnotu 255 metodou THRESH_BINARY_INV. Inverzní metoda z důvodu, aby body zájmu (kolečka) byla bílá.

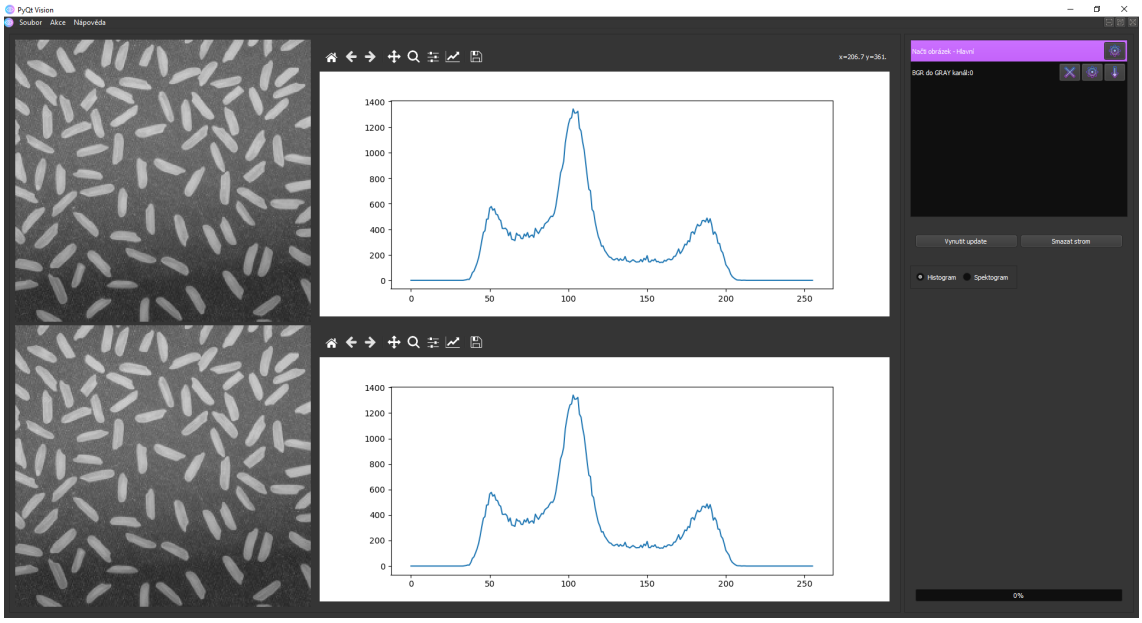


Obrázek 4.4: Binární morfologická operace otevření s maskou typu čtverce a velikosti 7. Při velikosti 6 se stále vyskytují nežádoucí skupiny pixelů.

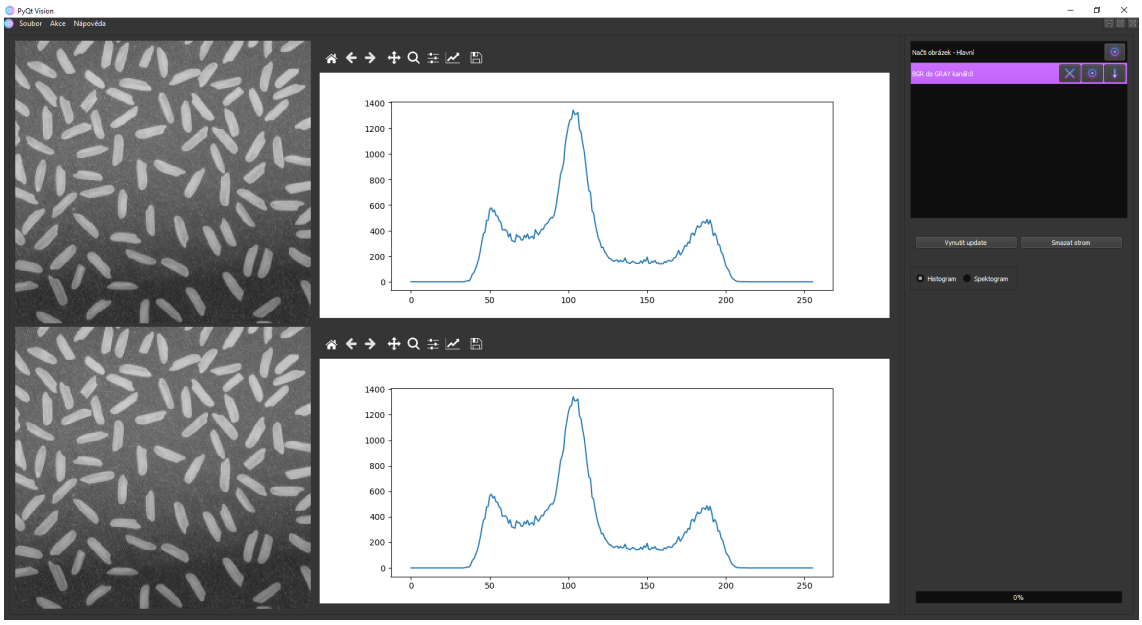


Obrázek 4.5: Aplikace rozvodí na obraz, který má oddělené pozadí od objektů.

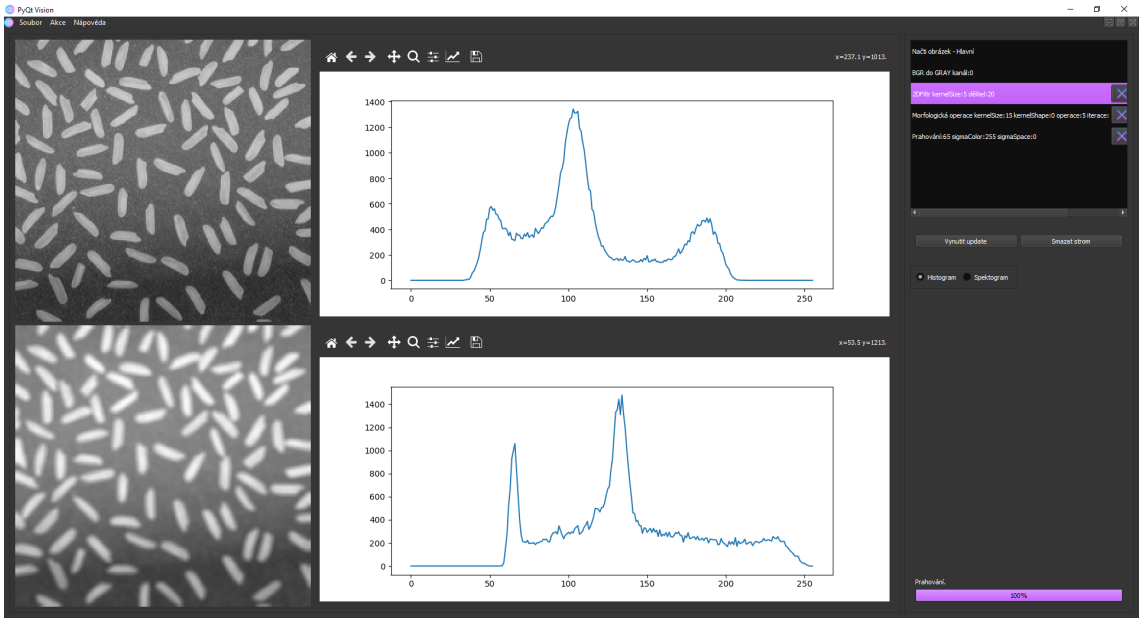
B Druhá ukázka segmentace



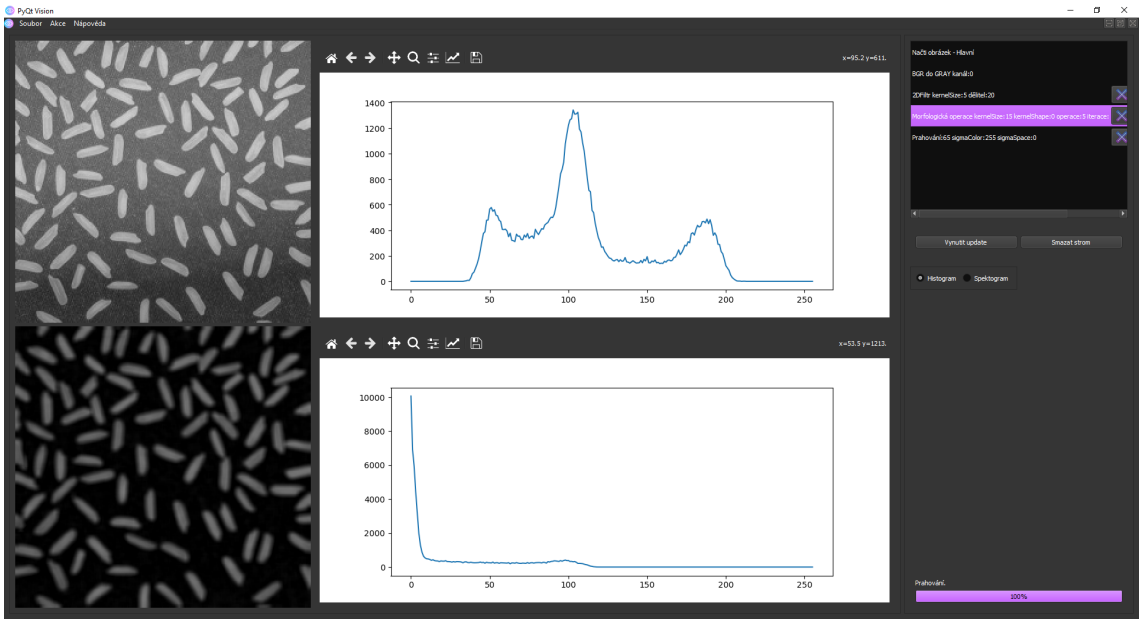
Obrázek 4.6: Načtení obrazu rýže s nerovnoměrným osvětlením, na kterém bude demonstrována operace top hat.



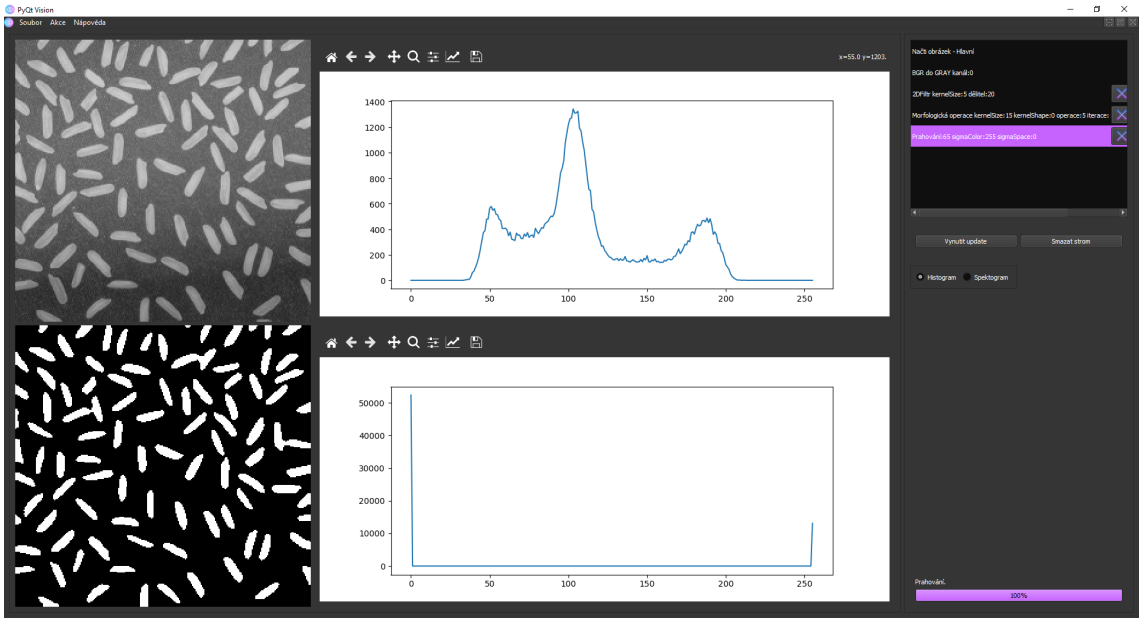
Obrázek 4.7: Aplikace předpokládá vždy práci s jedním barevným kanálem. Obraz je převeden do stupňů šedi.



Obrázek 4.8: Obraz je rozmazán konvolučním jádrem velikosti 5. Rozmazání obvykle přináší lepší výsledky segmentace, detekce hran a proto se u těchto aplikací často používá jako první krok.

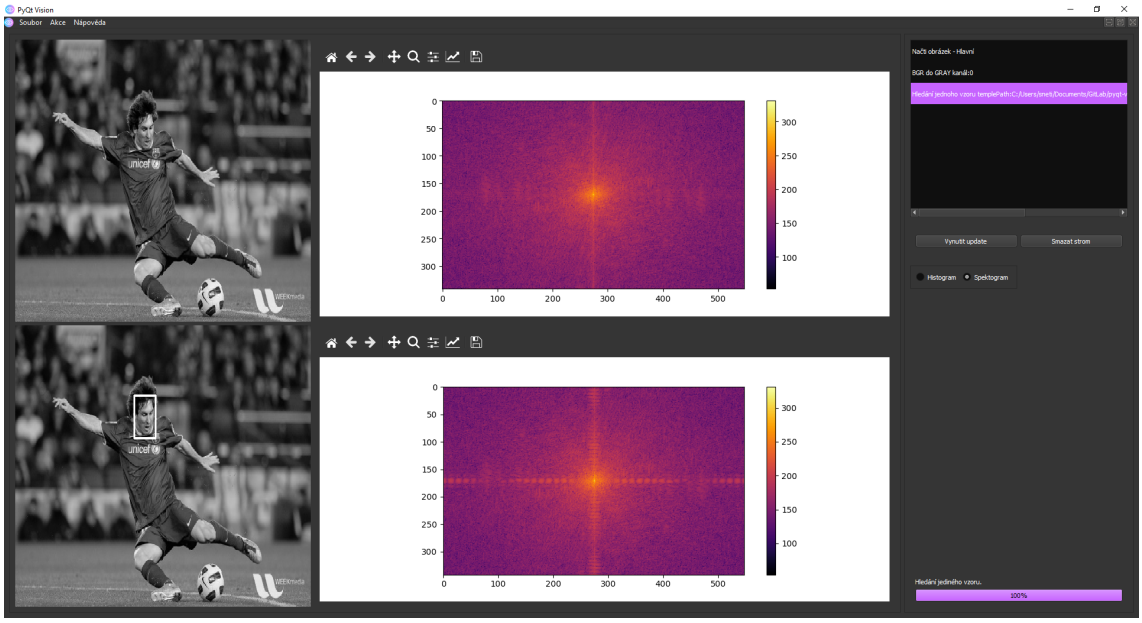


Obrázek 4.9: Vyrovnání nerovnoměrného jasu rýže pomocí binární morfologické operace top hat čtvercovou maskou velikosti 15.

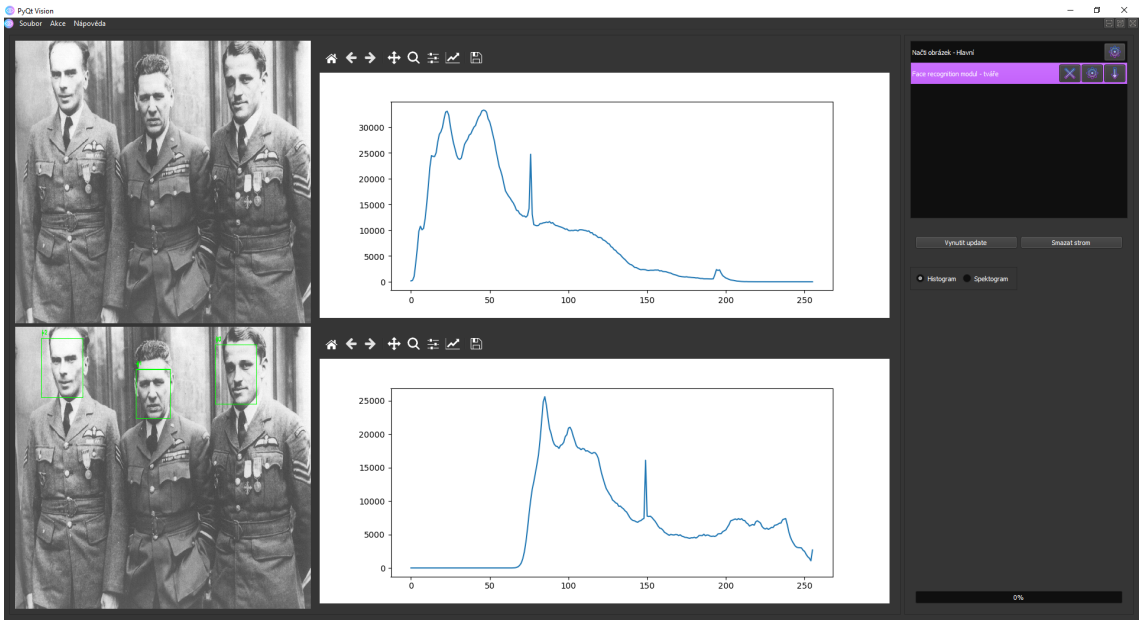


Obrázek 4.10: Aplikace jednoduchého prahování na základě histogramu z předchozí operace 4.9. Z histogramu je patrné že hodnoty rýže mají hodnotu pixelu okolo hodnoty 100. Uspokojivý práh pokus-omyl byl zvolen na hodnotu 65.

C Ostatní ukázky



Obrázek 4.11: Na základě předem vyřízlého obličeje byla provedena operace hledání jednoho vzoru. Zároveň demonstrace spektrogramu.



Obrázek 4.12: Aplikace metody z knihovny face_recognition na hledání tváří. V aplikaci existuje více implementací na hledání tváří, toto je jedna z nich.