VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Měření vzdáleností mezi obrazy

Chamfer Matching

2016

Miroslav Ježík

VŠB - Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Zadání diplomové práce

Student: **Bc. Miroslav Ježík**

Studijní program: N2647 Informační a komunikační technologie

Studijní obor: 2612T025 Informatika a výpočetní technika

Téma: Měření vzdáleností mezi obrazy
Chamfer Matching

Jazyk vypracování: čeština

Zásady pro vypracování:

Cílem práce je popsat a následně naimplementovat některou z metod pro vyhledávání shodných tvarů v obraze (ang. chamfer matching).

V práci proveďte následující kroky:
1. Seznamte se s metodami pro měření vzdálenosti (nebo také nepodobnosti) mezi dvěma obrazy.
2. Vyberte několik aktuálních metod a důkladně se s nimi seznamte, např. [1-3].
3. Detailně popište a názorně vysvětlete princip, výhody a nevýhody vybraný postupů v textu diplomové práce.
4. Proveďte implementaci alespoň dvou vybraných postupů v jazyce C++ v prostředí VS2013 za využití knihovny OpenCV. Zvažte možnosti paralelizace a vektorizace implementovaných algoritmů pro dosažení zpracování snímků v reálném čase [4]. Zdrojové kódy okomentujte pomocí nástroje Doxygen a dodržte Google C++ kódovací styl.
5. Jednotlivé postupy důkladně otestujte na sadě vhodných příkladů. Výsledky s důkladným porovnáním a vyhodnocením jednotlivých metod přiložte k textu práce.

Seznam doporučené odborné literatury:

[1] Liu, M., et al. Fast Directional Chamfer Matching. In CVPR, pp. 1696-1703, 2010.
[2] Cai, H., Werner, T., Matas, J. Fast Detection of Multiple Textureless 3-D Objects. In ICVS, pp. 103-112, 2013.
[3] Hinterstoisser, S., et al. Gradient Response Maps for Real-Time Detection of Textureless Objects. IEEE TPAMI, pp. 876-888, 2012.
[4] Rauter, M., Schreiber, D. A GPU Accelerated Fast Directional Chamfer Matching Algorithm and a Detailed Comparison with a Highly Optimized CPU Implementation. In CVPRW, pp. 68-75, 2012.

Formální náležitosti a rozsah diplomové práce stanoví pokyny pro vypracování zveřejněné na webových stránkách fakulty.

Vedoucí diplomové práce: **Ing. Tomáš Fabián, Ph.D.**

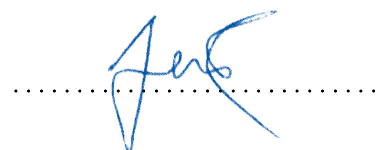
Datum zadání: 01.09.2015

Datum odevzdání: 29.04.2016

doc. Dr. Ing. Eduard Sojka	prof. RNDr. Václav Snášel, CSc.
vedoucí katedry	*děkan fakulty*

Prehlasujem, že som túto diplomovú prácu vypracoval samostatne. Uviedol som všetky literárne pramene a publikácie, z ktorých som čerpal.

V Ostrave 29. apríla 2016 ....



Súhlasím so zverejnením tejto diplomovej práce podľa požiadavkov čl. 26, odst, 9 Študijného a skúšobného poriadku pre štúdium v magisterských programoch VŠB-TU Ostrava.

V Ostrave 29. apríla 2016 ....

Na tomto mieste by som rád poďakoval všetkým, ktorí mi s prácou pomohli, predovšetkým vedúcemu práce pánovi Ing. Tomášovi Fabiánovi, Ph.D. za neskutočnú trpezlivosť.

Abstrakt

Táto práca sa zaoberá meraním vzdialenosti medzi obrazmi, ktorú je možné využiť pre detekovanie objektov v obraze. V práci sa venujeme metódam, ktoré umožnujú detekovať netextúrované objekty, ktorých jediným popisným prvkom je ich kontúra. Metódy, ktorými sme sa zaoberali, umožňujú detekovať objekty aj v scénach so zložitým pozadím a pri rôznych zmenách osvetlenia. V texte práce sú predstavené štyri algoritmy požiteľné pre detekciu. Dva z týchto algoritmov sú v práci popísané veľmi podrobne a v praktickej časti práce sú tieto dve metódy naprogramované. Implementované algoritmy sú ohodnotené na základe bežne používaných metrík a výsledky sú porovnané s voľne prístupnými implementáciami algoritmov z knižnice OpenCV.

Kľúčové slová: vzdialenost medzi obrazmi, detekcia objektov, počítačové videnie, OpenCV

Abstract

This thesis deals with issues related to measuring distance in images using chamfer matching. This measure is useful in detection of image templates in test images. The aim of this thesis is on methods, which are capable of detecting textureless objects. Only discriminative clue for these objects is their contour. Methods described in thesis can detect objects in scenes with cluttered background and in scenes with different illumination changes. There are four algorithms described in the text. Two methods are described comprehensively and they are implemented in practical part of thesis. Implemented algorithms are evaluated using common metrics and the results are compared to public implementations of algorithms presented in OpenCV library.

Key Words: chamfer matching, object detection, computer vision, OpenCV

Obsah

Zoznam použitých skratiek a symbolov **9**

Zoznam obrázkov **10**

1 Úvod **11**

2 Podobné práce **12**

3 Cumulative orientation feature **14**
3.1 Extrakcia orientácií gradientu 14
3.2 Deskriptor a jeho výpočet 15
3.3 Funkcia podobnosti 17
3.4 Prehľadávanie obrázka 17

4 Detekcia s využitím gestalistických princípov **20**
4.1 Gestalistické princípy 20
4.2 Reprezentácia hrán pomocou kodónov 21
4.3 Moment operátor 23
4.4 Tvarový kontext 28
4.5 Detekcia objektov 32

5 Ostatné algoritmy **34**
5.1 Fast Directional Chamfer Matching 34
5.2 Line-2D 37

6 Praktická implementácia **39**
6.1 Všeobecné informácie 39
6.2 Architektúra aplikácie 43
6.3 Ukážkový program 52

7 Vyhodnotenie a porovnanie **55**
7.1 Definícia pojmov 55
7.2 Porovnanie 57

8 Záver **62**

Literatúra **63**

Prílohy **65**

Zoznam použitých skratiek a symbolov

BRIEF	–	Binary Robust Independent Elementary Features
CM	–	Chamfer Matching
COF	–	Cumulative Orientation Feature
DCM	–	Directional Chamfer Matching
DOG	–	Difference of Gaussians
DOT	–	Dominant Orientation Template
DT	–	Distance Transform
FAST	–	Features from Accelerated Segment Test
FDCM	–	Fast Directional Chamfer Matching
FFT	–	Fast Fourier Transform
FPPI	–	False positives per image
GFTT	–	Good Features to Track
HOG	–	Histograms of Oriented Gradients
IGFTT	–	Iproved Good Features to Track
LSD	–	Line Segment Detector
OCM	–	Oriented Chamfer Matching
ORB	–	Oriented FAST and Rotated BRIEF
RANSAC	–	Random Sample Consensus
SC	–	Shape Context
SIFT	–	Scale-invariant feature transform
SIMD	–	Single Instruction Multiple Data
SSE	–	Streaming SIMD Extensions
SURF	–	Speeded Up Robust Features
VGA	–	Video Graphics Array
TBB	–	Threading Building Blocks

Zoznam obrázkov

1 Porovnanie veľkostí gradientu pre čiernobiely obrázok a farebný obrázok 15
2 Kvantizácia 16
3 Výpočet kumulatívnych orientácií pre 5-bitové slovo 16
4 Váhy COF deskriptora 17
5 Pyramída pre vyhľadávanie s použitím COF 18
6 Pyramída vytvorená na obrázku veľkostí gradientu 18
7 Príklady pre demonštráciu Gestalistických princípov 21
8 Stenčovanie hrán 22
9 Výpočet momentu 23
10 Moment operátor vypočítaný pre jednoduché tvary 24
11 Detekcia potencionálnych objektov 27
12 Tvarové kontexty pre podobné tvary 29
13 Rotačná invariancia s použitím FFT 30
14 Výpočet mapy vzdialeností 35
15 Výpočet máp vzdialeností a ich integrálnych obrazov 36
16 Vyhľadanie maximálnej hodnoty kosínusu rozdielu uhlov pre orientáciu (←) . . . 38
17 Triedny diagram implementácie návrhového vzoru továreň 43
18 Triedny diagram parametrov 44
19 Ukážka `xml` súboru s nastaveniami pre metódu COF 53
20 Ukážka aplikácia pre detekciu v reálnom čase 54
21 Vyhodnotenie detekovanej oblasti 56
22 Krivky zobrazujúce závislosť FPPI a senzitivity na prahu detekcie 58
23 Výsledky detekcie s použitím metódy Gestalist s prahom 0.2 59
24 Výsledky detekcie s použitím metódy COF s prahom 0.2 59

1 Úvod

Vnímanie obrazu je jedným zo základných vnemov človeka. Deje sa podvedome a obraz je prenášaný zo sietnice oka až do mozgu, kde je vyhodnocovaný. Človek nemá problém automaticky, bez zložitého zamyslenia, odpovedať na otázky ako: *"Sú tieto dva objekty podobné?"*, alebo *"Koľko objektov je na obrázku?"*. Odpovede na tieto otázky sú formulované na základe predchádzajúcich poznatkov, skúsenosti a dobrej pamäti. Úplne odlišná situácia nastáva pri spracovávaní digitalizovaného obrazu pomocou počítača. Aj napriek tomu, že dnešné počítače disponujú značným výkonom a stále viac rozširovanou pamäťou, je detekcia a rozpoznávanie veľkou výzvou v oblasti počítačovej grafiky. Pre to, aby počítač dokázal určiť, či sú dva obrázky podobné, musíme ich podobnosť nejako číselne ohodnotiť. Inak povedané, musíme nájsť funkciu, ktorá definuje kvázi-vzdialenosť dvoch obrazov. To ale nie je úplne jednoduchá úloha, najmä keď si uvedomíme, koľko rôznych faktorov môže ovplyvniť celkový vzhľad scény. Jeden objekt môže pri rôznych zmenách osvetlenia a uhla pohľadu kamery nadobúdať úplne iné farebné odtiene alebo tvar.

V súčasnosti sa počítačovej detekcii objektov v obraze venuje veľa autorov. Každým rokom sa objavujú nové algoritmy, ktoré zlepšujú doposiaľ vyvinuté metódy pre meranie vzdialenosti medzi obrazmi. Celkovo je oblasť, v ktorej sa algoritmy môžu zlepšovať veľmi široká. Je ťažké nájsť algoritmus, ktorý by dokázal pokryť celú túto oblasť a praxi často ani taký algoritmus nepotrebujeme. Pri väčšine aplikácií dokážeme odhadnúť, aké typy objektov detektor bude musieť rozpoznávať, aké sú ich charakteristické znaky, z akej vzdialenosti bude objekt snímaný a podobne. Preto sa prevažná časť algoritmov zaoberá len jednou časťou tejto oblasti a metódy detekovania objektov sa dajú rozdeliť do rôznych kategórií. Veľmi zaujímavú kategóriu tvoria netextúrované objekty, ktorých jediným identifikačným prvkom je ich tvar. Úspešne detekovať takéto objekty nie je až taká jednoduchá úloha, ako by sa mohlo na prvý pohľad zdať. Jedným z problémov je, že objekty môžu byť snímané z rôznych uhlov a ich tvar sa na rôznych obrázkoch potom líši. Navyše ich tvar je často jednoduchý a môžu v obrázku splynúť s pozadím. Medzi objekty spadajúce do tejto kategórie môžeme zaradiť každodenne používané veci ako napríklad kuchynské alebo kancelárske vybavenie, rôzne náradie, elektronické súčiastky, alebo malé objekty ako skrutky a matice. Detekcia práve takýchto objektov je často potrebná pri automatizovaní výrobných procesov pomocou autonómnych robotov. V práci sme sa rozhodli venovať predovšetkým detekcii týchto objektov.

V úvode práce sú v krátkosti zanalyzované najznámejšie algoritmy, ktoré dosahujú v oblasti detekcie objektov dobré výsledky. Predovšetkým je pri nich uvedený základný princíp ich funkcie a oblasť, pre ktorú sú algoritmy využiteľné. V ďalšom texte práce sú veľmi podrobne popísané teoretické základy potrebné pre pochopenie vybraných algoritmov, spolu s potrebnými ilustráciami a matematickými vzorcami. Tieto kapitoly sú nasledované kapitolami, ktoré popisujú postup pri praktickej implementácii vybraných algoritmov. V závere práce sú algoritmy otestované na bežne používanej sade obrázkov a úspešnosť detekcie je porovnaná s úspešnosťou voľne prístupných implementácií algoritmov.

2 Podobné práce

Porovnávaniu obrazov alebo detekcii objektov v obrazoch sa doteraz venovalo veľké množstvo článkov, v ktorých bolo predstavené veľké množstvo detektorov. Prvým typom detektorov sú také, ktoré na obrázku extrahujú množinu lokálnych príznakov, tzv. *feature points.* Tieto body charakterizujú hľadaný objekt a v ideálnom prípade sú tieto príznaky invariantné voči zmene merítka, rotácii aj posunu. Veľmi zaujímavý algoritmus pre vyhľadanie lokálnych príznakov je SIFT[24], ktorý už je takmer 16 rokov starý. Tento pri extrakcii kľúčových bodov používa funkciu *difference of gaussians* a lokálne okolie týchto bodov je popísané použitím deskriptora, vypočítaného vďaka *histograms of oriented gradients.* Pri vyhľadávaní sa potom postupuje tak, že sa príznaky extrahujú na modelovom aj testovanom obrázku a hľadajú sa korešpondencie medzi týmito dvoma množinami. Pre vyhľadanie korešpondujúcich bodov je používaný algoritmus RANSAC[16]. Niekoľko autorov tento algoritmus zdokonalilo, hlavne čo sa týka rýchlosti výpočtu. Algoritmy SURF[2] alebo FAST[31] predstavujú nový spôsob výpočtu kľúčových bodov, a tým zrýchľujú výpočet celkového algoritmu. Zdokonalenie deskriptora prináša napríklad algoritmus BRIEF[5]. Kombináciu a zdokonalením FAST a BRIEF algoritmov vzniká algoritmus ORB. V pomerne aktuálnom článku [10] je prezentovaná veľmi zaujímavá metóda IGFTT, zdokonaľuje veľmi starý algoritmus pre získavanie príznakov, ale dosahuje lepšie výsledky ako všetky uvedené metódy. Celkovo je algoritmov zaoberajúcich sa týmto spôsobom detekcie neuveriteľné množstvo, môžeme to usúdiť napríklad z toho, že článok [24] je podľa stránky `scholar.google.com` citovaný skoro 35000-krát. Fakt, ktorý je spoločný pre všetky tieto algoritmy je ten, že dosahujú dobré výsledky len v prípade detekcie textúrovaných objektov. V netexturovaných objektoch nie je možné dobre popísať lokálne okolie použitím niektorého z deskriptorov.

Tým sa dostávame ku kategórii algoritmov pre detekciu objektov, v ktorej vyhľadávaný objekt je popísaný iba svojou kontúrou. U netexturovaného objektu je kontúra jediný popisný prvok, navyše je veľmi vhodná, pretože u nej nezáleží na farbe objektu alebo na zmenách osvetlenia v testovanom obrázku. Jediným problémom pri využití kontúry je, že jej tvar sa mení pri zmene uhla pohľadu kamery. S týmto problémom sa algoritmy vysporadúvajú použitím rôznych spôsobov.

Porovnávanie kontúry modelu a kontúr v testovacom obrázku bolo prezentované napríklad už v článku [1], kde je hlavnou metrikou vzdialenosť bodu kontúry modelu k najbližšiemu bodu v kontúrach prehľadávaného obrázka. Nedostatkom tohto algoritmu je, že produkuje veľa nesprávnych detekcií vďaka tomu, že na testovanom obrázku môžu vznikať náhodné hrany z textúry pozadia alebo vplyvom šumu. Neskôr bol tento výpočet zrýchlený využitím akceleračnej štruktúry *distance transform.* V článku [23] je prezentovaná metóda FDCM, ktorá metriku používanú pri *chamfer matching* rozširuje o orientáciu hrany v kontúre. Tým značne redukuje počet falošne pozitívnych detekcií. Navyše je v článku prezentovaný spôsob ďalšieho urýchlenia výpočtu pomocou diskretizácie orientácií a použitím akceleračných štruktúr. Princíp celého algoritmu je popísaný v kapitole 5.1.

Veľmi zaujímavé články prezentovala skupina okolo Stefana Hinterstoissera. Článok [19] prezentuje metódu DOT. V nej sú pre ohodnotenie podobnosti modelu a testovaného obrázka používané orientácie gradientov. V článku je použitý zaujímavý spôsob urýchlenia prehľadávania, kde v testovanom obrázku nie sú brané do úvahy všetky gradienty, ale len tie, ktoré majú vo svojom lokálnom okolí najvyššiu magnitúdu gradientu. Ďalšie zrýchlenie je dosiahnuté diskrtizáciou orientácií gradientu. Druhým článkom od týchto autorov je [20], v ktorom je predstavená metóda LINE-2D. Postup je v porovnaní s DOT zdokonalený tým, že funkcia podobnosti využíva kosínus rozdielu orientácie gradientu pre body v modelovom a testovanom obrázku. Navyše je prezentovaný veľmi efektívny algoritmus výpočtu, kde hodnoty kosínusu rozdielov uhlov sú vypočítané ešte pred začatím detekcie. Oba algoritmy dosahujú dobrých výsledkov aj pri detekcii objektov v zložitých scénach. Algoritmus LINE-2D je podrobne popísaný v kapitole 5.2. Článok [22] prezentovaný inými autormi v roku 2015 predstavuje algoritmus založený na veľmi podobných princípoch ako DOT a LINE-2D. Orientácie gradientov sú v ňom rovnako reprezentované, odlišný je spôsob výberu bodov pre deskriptor. Tento algoritmus je súčasťou implementácie praktickej časti tejto práce a viac podrobností k jeho výpočtu je v kapitole 3.

Ďalší algoritmus má názov BOLD[39] a vo výsledkoch dosahuje lepšie hodnoty v porovnaní s algoritmom LINE-2D. V algoritme BOLD sú v modelovom aj testovanom obrázku extrahované hranové segmenty pomocou algoritmu LSD[40]. Deskriptor modelu je tvorený relatívnym rozdielom uhlov, ktoré zvierajú susediace segmenty. V segmentoch testovaného obrázka sú rozdiely uhlov extrahované podobne a sú porovnané s deskriptorom. Veľmi podobný algoritmus je prezentovaný v [9], kde sú rovnako využívané hranové segmenty. Susedné segmenty sú spájané do tzv. konštalácií (n-tica segmentov). Následne je vyhľadaná cesta, ktorá prechádza stredmi všetkých segmentov v konštalácii. Deskriptor pre objekt je tvorený relatívnymi uhlami jednotlivých segmentov cesty.

Porovnávanie jednoduchých kontúr je veľmi presné s použitím tvarového kontextu [3]. Deskriptor pre jeden bod kontúry je definovaný ako histogram vzdialeností všetkých bodov kontúry od stredu log-polárneho súradnicového systému vo vybranom bode. Tento algoritmus dosahuje veľmi dobrých výsledkov na kontúrach, ktoré nie su rušené zložitým pozadím alebo rozostrením obrazu. Jeho použitie je bežné v oblasti porovnávania písma alebo rôznych znakov.

Článok [27] prezentuje nový operátor použiteľný pre detekciu objektov. Operátor pre svoju funkciu využíva moment sily, známy z fyziky alebo matematiky. Pomocou tohto operátora sú detekované oblasti, v ktorých je vysoká pravdepodobnosť výskytu objektov. V článku [26] sú prezentované ďalšie zaujímavé využitia tohto operátora. V článku [36] je tento operátor využitý v kombinácii s tvarový kontextom [3]. Takto rozšírený algoritmus je podrobne popísaný v kapitole 4 a jeho implementácia je súčasťou praktickej časti práce.

3 Cumulative orientation feature

Prvá metóda porovnávania obrazu, ktorú sme naimplementovali má názov *"cumulative orientation feature"* (ďalej len COF) a je prezentovaná v pomerne aktuálnom článku [22]. Metóda je v mnohom podobná algoritmu LINE-2D [20]. Rovnako využíva len orientáciu a veľkosť gradientu, takže je robustná voči obrázkom so zložitým pozadím a zmenám osvetlenia. Odolnosť voči malým posunom a deformáciám je dosiahnutá tým, že vstupný model je pri výpočte deskriptora v malom rozsahu posúvaný, rotovaný a následne sú orientácie gradientov kumulované. Výsledný deskriptor tvoria body, v ktorých je počet akumulovaných orientácií najväčší. Metóda porovnávania je pomerne nenáročná pre výpočet, preto robustnosť voči zmenám v merítku alebo rotácii je dosiahnutá tým, že sa pre hľadaný model vytvorí množina obrázkov, obsahujúca všetky rotácie a zmeny v merítku. V testovacom obrázku sa potom hľadajú všetky objekty z tejto množiny. Optimalizácia prehľadávania je dosiahnutá použitím algoritmu *"coarse to fine strategy"*[4]. Na nasledujúcich riadkoch je popísaný teoretický základ tohto algoritmu.

3.1 Extrakcia orientácií gradientu

Orientácie sú z obrázkov extrahované s použitím Sobelovho operátora. Sobelov operátor je jeden zo základných operátorov v oblasti počítačovej grafiky a je využívaný mnohými algoritmami. Bol predstavený už v roku 1968 a jeho presnú definíciu a históriu môžeme nájsť aj v článku [34], ktorý bol nedávno publikovaný samotným autorom Irwinom Sobelom.

Operátor vypočítava aproximované hodnoty gradientov k funkcii jasu obrázku v smeroch x a y. Keďže počítačová reprezentácia obrazu je diskretizovaná do jednotlivých pixelov, môžeme hodnotu gradientu získať odčítaním hodnôt jasu u susedných pixelov. Pre obraz $\mathcal{I}$ môžeme výpočet Sobelovho operátora zapísať ako konvolúciu obrázka $\mathcal{I}$ s jadrami o rozmeroch 3x3 ako

$$\mathcal{I}_x = \begin{bmatrix} -1 & 0 & +1 \\ -2 & 0 & +2 \\ -1 & 0 & +1 \end{bmatrix} * \mathcal{I} \quad \text{a} \quad \mathcal{I}_y = \begin{bmatrix} -1 & +2 & -1 \\ 0 & 0 & 0 \\ +1 & +2 & +1 \end{bmatrix} * \mathcal{I} \, ,$$

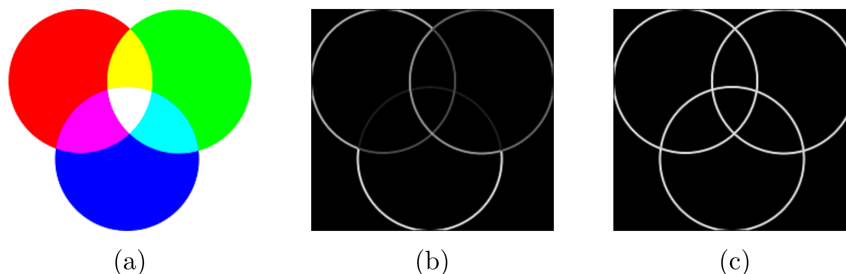
kde $*$ je operácia konvolúcie. Touto konvolúciou dostaneme dva obrazy $\mathcal{I}_x$ a $\mathcal{I}_y$, ktoré obsahujú hodnoty derivácií v x a y. Obraz $\mathcal{C}$ obsahuje derivácie v oboch s smeroch $\mathcal{C} = (\mathcal{I}_x, \mathcal{I}_y)$. Z takto vypočítaných obrazov môžeme získať veľkosť (magnitúdu) gradientu ako

$$\|\mathcal{C}\| = \sqrt{\mathcal{I}_x + \mathcal{I}_y}$$

a orientáciu gradientu ako[1]

$$\text{ori}(\mathcal{C}) = \text{atan2}(\mathcal{I}_y, \mathcal{I}_x) \; .$$

[1]V rovnici je zámerne uvedená funkcia atan2, podobne ako v C++ zápise, aby bolo jasné, že orientácia môže byť z rozsahu $[-\pi, +\pi]$.



Obr. 1: Porovnanie veľkostí gradientu pre čiernobiely obrázok a farebný obrázok

Pretože vstupom pre testovací program sú farebné obrázky, ktoré majú tri farebné kanály RGB, môžeme pre zvýšenie robustnosti detektora vypočítať orientácie gradientov pre každý kanál osobitne ako v [8] a pre ďalšie spracovanie potom použiť orientácie z toho kanála, ktorý má v danom bode maximálnu veľkosť gradientu. Ak vstupný obraz $\mathcal{I}$ obsahuje kanály RGB, hodnota $\mathcal{I}_{\mathcal{G}}$ v bode x je vypočítaná ako

$$\mathcal{I}_{\mathcal{G}}(x) = \mathrm{ori}(\hat{\mathcal{C}}(x)) ,$$

kde

$$\hat{\mathcal{C}}(x) = \underset{\mathcal{C} \in \{R,G,B\}}{\mathrm{argmax}} \|\mathcal{C}\| .$$

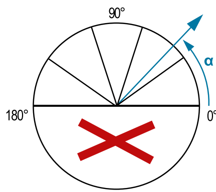
Rozdiel je zobrazený na obrázku 1. Pre vstupný obrázok 1a boli spočítané veľkosti gradientu. Na obrázku 1b je zobrazený výstup z programu[2], kde sme najprv previedli farebný obrázok na stupne šedi a na ňom vyhľadávali veľkosť gradientu. Pre výpočet obrázka 1c bola použitá práve popísaná metóda využívajúca RGB kanály. Je zjavné, že veľkosti gradientov majú na obrázku 1c približne rovnaké hodnoty a na obrázku 1b je napr. hranu medzi červenou a fialovou plochou len ťažko spozorovať.

3.2 Deskriptor a jeho výpočet

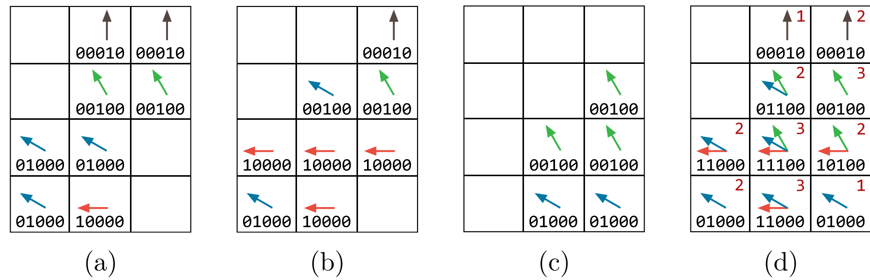
Výpočet deskriptora COF začína tým, že z obrázku hľadaného modelu $\mathcal{I}_M$ je vytvorené veľké množstvo jeho transformácií (rotácia a posuv). Rozsah transformácií je volený z malého rozsahu, okolo 1-2 pixely pre posuv a 5-10 stupňov pre rotáciu a počet transformácií je okolo 200. Orientácie sú extrahované na všetkých týchto transformovaných obrázkoch a sú kumulované pre každý pixel. Priestor možných orientácií je kvantizovaný do 8 oblastí. Počet 8 nie je volený náhodne, ale preto, aby orientácie bolo možné reprezentovať pomocou jedného bajtu tak, že ak orientácia spadá do danej oblasti, je príslušný bit nastavený na 1, ostatné bity sú nastavené na 0.

Na obrázku 2 je znázornená kvantizácia vektoru orientácie. Orientácia reprezentovaná modrým vektorom je priradená do oblasti s indexom 1 (pri indexovaní od 0). Jej bitové kódovanie

[2]Výstup je možné prehliadnuť si po spustení demo programu s parametrom `-fig=gradient img.png`. Parametre pre ostatné ukážky sú na strane 52



Obr. 2: Kvantizácia



Obr. 3: Výpočet kumulatívnych orientácií pre 5-bitové slovo

by teda odpovedalo binárne zapísanému číslu `00010`. Pri kvantizácii neberieme do úvahy smer gradientu, ale len jeho orientáciu. Teda orientácie, ktoré na obrázku 2 spadajú do oblasti s červeným krížikom sú kvantizované k odpovedajúcej oblasti na druhej polovici. To nám zabezpečí, že objekt môže byť úspešne detekovaný aj v tom prípade, ak bol deskriptor extrahovaný pre svetlý objekt na tmavom pozadí a na testovacom obrázku sa vyskytuje tmavý objekt na svetlom pozadí (alebo naopak). Hrany, ktoré majú veľkosť gradientu nižšiu ako určené minimum nie su brané do úvahy - týmto prahovaním sa odstránia náhodné hrany, ktoré mohli vzniknúť z náhodného šumu v obrázku. Konečná kumulácia je prevedená tak, že na bajtoch, reprezentujúcich jednotlivé orientácie je prevedená operácia logický súčet ($\vee$), pričom je počítaný aj počet orientácií kumulovaných pre každý pixel. Tento počet je ďalej nazývaný aj frekvencia alebo váha. Výpočet pre 5-bitové slovo je znázornený na obrázku 3, kde 3a, 3b, 3c sú transformované obrázky modelu a 3d je ich kumulovaný súčet. Šípkou je reprezentovaná orientácia (smer neberieme do úvahy), číselným kódom pod šípkou je reprezentovaná daná orientácia v bitovom slove a červené čísla na obrázku 3d sú frekvencie pre odpovedajúci pixel. Body, v ktorých je frekvencia väčšia ako zvolená hodnota, tvoria deskriptor $\mathcal{M}_i$ pre daný objekt, zapísaný ako

$$\mathcal{M}_i : \{x_j, y_j, \mathrm{ori}_j, w_j | j = 1, ..., n\}, i = 1, ..., m \; .$$

Deskriptor objektu $\mathcal{M}_i$ sa skladá zo štyroch parametrov, kde x_j, y_j je pozícia relatívna k stredu objektu, ori_j sú kumulované orientácie, w_j je frekvencia orientácií v danom bode, m je počet hľadaných modelov, kde každý ma svoj vlastný deskriptor (alebo môže byť pre jeden model spočítaných viac deskriptorov v rôznych rotáciách a merítkach. Viď kapitola 3.4).

Na obrázku 4b sú zobrazené váhy deskriptora [3]. Model bol 200 krát náhodne transformovaný - rotovaný v rozmedzí $\pm 10°$ a posunutý v rozmedzí $\pm$3px. Obrázok 4c zobrazuje prahované váhy s prahom 0.5, teda váhy, ktorých normalizovaná hodnota prekročila 0.5.

[3] V testovacom programe je možné zobraziť váhy deskriptora a ich orientácie pri spustení aplikácie s parametrom `-fig=ori img.png`. Parametre pre ostatné ukážky sú na strane 52

3.3 Funkcia podobnosti

Pred porovnávaním deskriptoru a testovaného obrázka sú z testovaného obrázka extrahované hrany a kvantizované do bitových slov rovnako ako pre deskriptor. Funkcia podobnosti určuje podobnosť medzi testovaným obrázkov v bode x, y a deskriptorom. Ak sa orientácia gradientu z testovaného obrázka nachádza na odpovedajúcej pozícii v kumulovaných orientáciách deskriptora, je príslušná váha pripočítaná do celkovej sumy, ktorá určuje podobnosť modelu a testovaného obrázka. Vyššia hodnota funkcie s značí vyššiu podobnosť obrázkov. Nakoniec je celková suma normalizovaná celkovým súčtom váh deskriptora. Podobnosť medzi testovaným obrazom $\mathcal{I}$ v bode x, y a deskriptorom $\mathcal{M}_i$ je určená ako

$$s(x, y, i) = \frac{\sum_{j=1}^{n} \delta_j \left(\text{ori}^{\mathcal{I}}_{(x+x_j, y+y_j)} \in \text{ori}_j{}^{\mathcal{M}_i} \right)}{\sum_{j=1}^{n} w_j}, \tag{1}$$

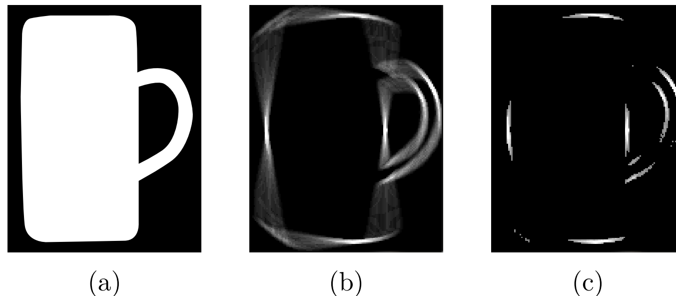
kde

$$\delta_j(\text{ori}^{\mathcal{I}} \in \text{ori}_j^{\mathcal{M}_i}) = \begin{cases} w_j & \text{ak ori}^{\mathcal{I}} \wedge \text{ori}_j^{\mathcal{M}_i} > 0 \\ 0 & \text{inak.} \end{cases} .$$

Je badateľné, že pre každý bod deskriptora sa musí spočítať funkcia logického súčinu $\wedge$ s bodom testovaného obrázka. Výpočet je na CPU pomerne rýchly, naviac výpočet môže byť ďalej optimalizovaný pomocou SIMD inštrukcií ako napríklad SSE2.

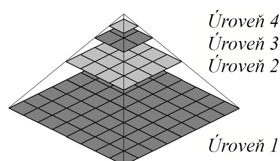
3.4 Prehľadávanie obrázka

Keďže funkcia porovnávania je pomerne rýchla na výpočet, môžeme dosiahnuť invarianciu voči merítku a rotácii tým, že je deskriptor $\mathcal{M}_i$ vypočítaný pre každú možnú orientáciu a merítko modelového obrázka $\mathcal{I}_M$. Takto vytvorená množina deskriptorov je označená $\mathcal{M}$. Aby sme v obrázku vyhľadali potencionálne objekty, potrebujeme nájsť pozíciu x, y a deskriptor $\mathcal{M}_i$, pre ktoré funkcia $s(x, y, i)$ nadobúda maximálnu hodnotu a zároveň je vyššia ako predom definovaný prah. Množinu obsahujúcu detekované objekty označíme $\mathcal{P}$. Extrém $p = (x, y, i) \in \mathcal{P}$ určuje pozíciu x, y a index i určuje deskriptor $\mathcal{M}_i \in \mathcal{M}$, na základe ktorého môžeme zistiť tvar objektu.



(a) (b) (c)

Obr. 4: Váhy COF deskriptora



Obr. 5: Pyramída pre vyhľadávanie s použitím COF

(a) Úroveň 1 (b) Úroveň 2 (c) Úroveň 3

Obr. 6: Pyramída vytvorená na obrázku veľkostí gradientu

Testovaný obrázok by sme mohli prehľadávať bod po bode a pre každý bod spočítať funkciu podobnosti s. V takto definovanom algoritme by sa funkcia s musela vypočítať približne $|\mathcal{M}| \times |\mathcal{I}|$ krát, kde $|\mathcal{M}|$ označuje počet deskriptorov a $|\mathcal{I}|$ počet bodov testovacieho obrázku. To je veľmi naivný spôsob, ktorý by dosahoval primeraný čas výpočtu len pre obrázky s veľmi nízkym rozlíšením a pre veľmi malé $\mathcal{M}$. Napríklad ak by boli vygenerované možné rotácie hľadaného objektu v rozmedzí $\pm 180°$ pre každých $18°$ a merítka v rozmedzí $\pm 25\%$ po 5%, dostali by sme 200 deskriptorov pre jeden objekt, teda $|\mathcal{M}| = 200$. Už pre obraz vo VGA rozlíšení, by program musel funkciu podobnosti $s(x, y, i)$ vykonať $200 \times 640 \times 480$ krát.

Preto je pre optimalizáciu vyhľadávania použitý algoritmus *"coarse to fine strategy"*[4], ktorý je pomerne často používaný pri prehľadávaní obrázkov. Prvým krokom tohto algoritmu je, že sa skonštruujú tzv. pyramídy obrázkov pre obrázok modelu a aj pre testovací obrázok. Pyramída je konštruovaná postupným zmenšovaním vstupného obrázka o 50%, takže počet bodov sa v každej novej úrovni zníži na $\frac{1}{4}$. Počet úrovní pyramídy k je priamo určený, alebo konštrukcia pyramídy sa ukončí, ak je veľkosť obrázka menšia ako zvolené minimum. Príklad takto skonštruovanej pyramídy je na obrázku 5, kde $k = 4$. Je jasne viditeľné, že počet bodov v každej úrovni je $\frac{1}{4}$ z predchádzajúcej. V prípade COF konštruujeme pyramídu pre oba, modelový aj testovací obrázok. Následne je spočítaný deskriptor pre modelové obrázky na každej úrovni pyramídy. Pre testovací obrázok sú extrahované kvantizované orientácie takisto na každej úrovni. Pyramída pre deskriptor môže byť vypočítaná pred začatím porovnávania, pyramída pre testovaný obrázok musí byť vypočítaná v reálnom čase. Vzhľadom na to, že orientácia je počítaná pomocou pomerne náročnej funkcie `atan2`, orientácie testovacieho obrázka sa nepočítajú pre každú vrstvu znova, ale stačí ak sa pyramída konštruuje zmenšovaním obrázka, ktorý obsahuje nekvantizované orientácie a spolu s obrázkom veľkostí gradientu (znázornené na obrázku 5). V našej implementácii bola pre orientácie použitá metóda lineárneho zmenšovania, kde sa hodnota pixelu vypočíta ako aritmetický priemer orientácií odpovedajúich štyroch pixelov na vyššej úrovni a výsledná priemerná orientácia sa kvantizuje spôsobom popísaným v kapitole 3.2.

Pri vyhľadávaní modelového obrázku v testovacom obrázku sa začne na najvyššej k-tej úrovni pyramídy, teda na tej, ktorá obsahuje najmenej pixelov a pokračuje sa smerom dole až k najnižšej úrovni, ktorej počet bodov je zhodný s počtom bodov vstupného obrázka $\mathcal{I}$. Na najvyššej úrovni sa pre všetky body a všetky deskriptory spočíta funkcia podobnosti $s(x, y, i)$ a vyhľadajú sa extrémy tejto funkcie. Nájdené extrémy $p_j = (x, y, i)$ sú uložené do pracovnej množiny P. Na

každej ďalšej úrovni sa zistí množina bodov, ktoré odpovedajú extrémom na predchádzajúcej úrovni, funkcia podobnosti sa spočíta len na týchto bodoch a extrémy sa zase uložia do množiny P. Funkcia $s(x, y, i)$ je rátaná len v okolí bodov so súradnicami x, y a len na modeloch s podobnou orientáciu a merítkom ako model s indexom i. Po spracovaní poslednej vrstvy pyramídy, množina P obsahuje extrémy funkcie s, ktoré by mali odpovedať extrémom, ktoré by sme našli pri prechádzaní naivným spôsobom - popísaným na začiatku kapitoly. Z množiny extrémov P môžeme zistiť pozície, rotáciu a merítko objektov. Takýmto prístupom sa výrazne zníži počet celkovo prehľadávaných bodov a deskriptorov, ale schopnosti detektora ostanú rovnaké.

Pre porovnanie uvedieme podobný príklad ako bol uvedený na začiatku kapitoly. Pre testovací obrázok vo VGA rozlíšení a model s 200 transformáciami, pyramídou s tromi úrovňami a s tým, že na každej ďalšej úrovni pyramídy sa vyhľadá 5 extrémov a prehľadá sa 9 susedných bodov v okolí extrému, sa porovnávacia funkcia $s(x, y, i)$ bude musieť vypočítať len okolo $200 \times 160 \times 120 + (5 \times 9)^2$ krát, čo približne 16-krát menej, ako v prvom prípade. Pri porovnávaní modelu s testovacím obrázkom naviac môžeme využiť fakt, že pri výpočte je deskriptor náhodne posúvaný v rozmedzí $\pm t$ bodov. Vďaka tomuto je deskriptor invariantný voči malému posunu a pri prehľadávaní najvyššej vrstvy pyramídy nie je nutné prehľadávať všetky body, ale len každý t-tý bod. Takto sa počet bodov prehľadávaných na najvyššej vrstve pyramídy zredukuje t-krát.

V prílohe A.1 je zapísaný pseudoalgoritmus pre detekciu objektov pomocou COF. Algoritmus je rozdelený na dva kroky. Prvý krok sa týka výpočtu deskriptorov a je ho možné vypočítať offline, teda ešte pred začatím samotnej detekcie. Druhý krok je samotné prehľadávanie obrázka, ktoré sa vykonáva v reálnom čase, preto je u neho dôležitá časová náročnosť operácií.

Algoritmus COF je pomerne jednoduchý na pochopenie a pre implementáciu. Zjednodušuje prístupy z algoritmu LINE-2D, ktorý je bližšie predstavený v kapitole 5.2. Napriek tomu sú výsledky detekcie porovnateľné s LINE-2D ako je prezentované v kapitole 7.

4 Detekcia s využitím gestalistických princípov

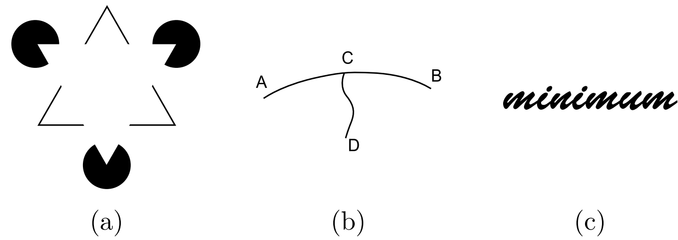
Druhá metóda, ktorú sme sa rozhodli naimplementovať, je detekcia objektov pomocou gestalistických princípov, ktoré podrobnejšie predstavíme v podkapitole 4.1. Táto metóda detekcie sa snaží pri detekcii objektov simulovať vnímanie človeka. Teda najprv sa zameriava len na hrubé tvary objektov, ktoré by po zoskupení mohli tvoriť hľadaný objekt a až potom potencionálnu zhodu porovná voči objektu, ktorý už pozná z predchádzajúcej skúsenosti. Pre prvý krok detekcie využíva tzv. moment operátor (*torque operator*). Definícia momentu je rovnaká ako moment sily vo fyzike alebo mechanike. To, že moment môže byť využitý aj v spracovaní obrazu, bolo ukázané v článku [27], kde autori predstavili tento operátor ako možnosť pri detekcii významných oblastí (*saliency detecion*), detekcii hrán alebo pre segmentácii obrazu. My sme konkrétne implementovali detekciu objektov podľa článku [36], kde je moment operátor využitý najprv pre detekciu potencionálnych objektov bez ohľadu na konkrétny hľadaný objekt (*bottom-up searching*). Potom sú potencionálne objekty porovnané s modelom a pre hrany potencionálnych objektov sú ohodnotené váhami. Na ich výpočet sa využíva často používaný deskriptor: tvarový kontext (*shape context*)[3]. Nakoniec sa znova pre celý obrázok vypočíta momentová mapa pomocou moment operátora, tentoraz ale už s využitím ohodnotených váh. Extrémy v tejto momentovej mape predstavujú pozície detekovaných objektov.

Prvý krok detekcie, teda vyhľadávanie významných oblastí bez ohľadu na konkrétny hľadaný objekt, býva označovaný ako prehľadávanie zdola-hore (*bottom-up*). Naopak vyhľadávanie konkrétneho tvaru býva označované ako prehľadávanie zhora-dole (*top-down*). Postup, ktorý je predstavený v tejto kapitole, teda kombinuje oba tieto spôsoby vyhľadávania.

4.1 Gestalistické princípy

Pôvod slova Gestaltismus je v nemeckom výraze Gestalt, čo v slovenčine znamená podoba, tvar, útvar, celok, štruktúra. Často sa stretneme aj s pojmom tvarová psychológia alebo tvarové vnímanie. Princípy Gestalizmu popisujú ako človek vníma svoje okolie. Prikláňajú sa k tomu, že človek často vnímané objekty spája do celkov a vníma ich ako jeden tvar. Korene Gestalizmu siahajú k začiatku dvadsiateho storočia [21]. Čoraz častejšie sa ale s týmito princípmi stretávame aj dnes v oblasti počítačov, najmä v oblasti dizajnu [7] a návrhu užívateľských rozhraní [33]. Gestalizmu sa venovalo viac autorov a v rozdielnych publikáciach sú uvedené rôzne princípy. My uvedieme len tie, ktoré sa najviac týkajú tejto práce alebo počítačovej oblasti všeobecne.

- **Princíp uzatvorenosti** hovorí o tom, že objekty zoskupené blízko u seba často vnímame ako celok. A to aj také, ktoré vôbec nemusia existovať. Napríklad na obrázku 7a nie je žiaden celý trojuholník alebo kruh, náš mozog si ale informáciu doplní podľa tvarov, ktoré sú mu všeobecne známe. Tento princíp je pri popisovanom detektore využívaný hlavne v prvom kroku, kde ako potencionálne objekty sú vybrané hrany, ktoré sú blízko seba a spolu tvoria nejakú uzavretú oblasť.



Obr. 7: Príklady pre demonštráciu Gestalistických princípov

- **Princíp kontinuálnosti** popisuje, že objekty, ktoré tvoria nejakú postupnosť sú vnímané ako jeden tvar. Demonštráciu môžeme vidieť na obrázku 7b. Ak by sme mali obrázok rozdeliť na viac častí, určite by každého z nás skôr napadlo rozdeliť obrázok na dve krivky: AB, CD. Toto využívame aj pri detekcii pomocou moment operátora, kde hrany objektov, ktoré majú podobnú orientáciu, prispievajú veľkou hodnotou k celkovému skóre potencionálneho objektu.

- **Princíp podobnosti** úvadza, že je veľká pravdepodobnosť, že malé objekty, ktoré majú podobný tvar, pri vnímaní zoskupíme k sebe. To je často využívané pri návrhu užívateľských rozhraní, napríklad pri rozložení zaškrtávacích alebo prepínacích tlačidiel (*radio button*).

- **Princíp predchádzajúcej skúsenosti** vysvetľuje, že objekty, ktoré poznáme a ktoré sa vyskytujú často pri sebe, zvykneme zoskupovať do jedného objektu. Často sa tak deje napríklad prí písmenách abecedy, kde slová zvykneme vnímať ako jeden objekt, alebo zoskupovať písmená, ktoré sa často vyskytujú pri sebe. Typický príklad je na obrázku 7c, slovo minimum napísané takýmto druhom písma by sme možno mali problém prečítať pri čítaní po jednom písmene. Z predchádzajúcej skúsenosti ale písmená vnímame ako jedno celé slovo. V našom programe by sme tento princíp mohli aplikovať na to, že program v testovacom obrázku vyhľadáva objekt, ktorého tvar je vopred známy.

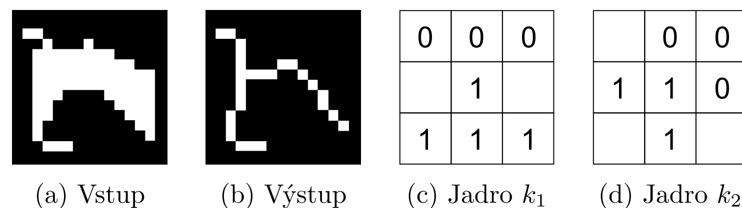
4.2 Reprezentácia hrán pomocou kodónov

Kontúry objektov detekovaných týmto detektorom sú reprezentované pomocou tzv. kodónov. Názov kodón bol inšpirovaný článkom [30]. Jedná sa o časti kontúr objektov, ktorých krivosť je vo všetkých ich bodoch menšia ako predom stanovené minimum. Prakticky to znamená to, že kodóny sú konštruované tak, že kontúra objektu je rozdelená v bodoch s najväčšou krivosťou.

Pri výpočte krivosti sme uvažovali dva spôsoby. Prvý spôsob je z článku [12]. Krivosť je v ňom pre kontúru $\mathcal{K} = (q_i, ..., q_m)$ v bode q_i so susednými bodmi $q_{(i-s)}$ a $q_{(i+s)}$ spočítaná ako prevrátená hodnota priemeru kružnice, ktorá prechádza týmito tromi bodmi. Pričom s je vzdialenosť susedných bodov, v praxi malé číslo $s \approx 5$. Takto definovaná krivosť dosahuje dobré

výsledky, jej výpočet sa ale ukázal ako neefektívny. Druhý spôsob, ktorý sme nakoniec aj použili pri implementácii, je z článku [3], kde je krivosť v bode q_i definovaná ako $\|q_{(i-s)} - 2q_i + q_{(i+s)}\|$.

Prvým krokom pri konštrukcii kodónov je vyhľadanie hrán modelového aj testovaného obrázka niektorým z mnohých hranových detektorov. V našej implementácii sme použili Cannyho detektor hrán [6]. Výstupom detektora hrán je binarizovaný obraz I_e, ktorý v hranových bodoch obsahuje nenulové hodnoty, naopak miesta, kde sa hrany nenachádzajú, sú označené 0. Hrany detekované Cannyho detektorom sú stenčené, ale nie až tak, ako by sme potrebovali pre ďalšie spracovanie, teda tak, aby sa v osem-okolí každého hranového bodu nachádzali maximálne 2 hranové body. Preto je na výstupný obraz I_e použitá jednoduchá morfologická operácia [38][15] - stenčovanie hrán alebo skeletonizácia. Charakteristickým znakom tejto operácie je, že hrany sú stenčené na maximálnu úroveň a to tak, že každý hranový bod má vo svojom 8-okolí maximálne 2 hranové body, pritom ale dĺžka hrán ostáva rovnaká a žiadna hrana nie je rozdelená. Výstup stenčovania je znázornený na obrázku 8b, kde vstupom pre program[4] bol obrázok 8a.



(a) Vstup (b) Výstup (c) Jadro k_1 (d) Jadro k_2

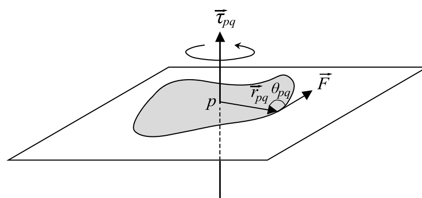
Obr. 8: Stenčovanie hrán

Základom tejto morfologickej operácie sú dve jadrá k_1, k_2 zobrazené na obrázkoch 8c a 8d. Body označené ako 1 predstavujú hranové pixely, 0 sú body pozadia. Body, ktoré nemajú označenie, neberieme pri stenčovaní do úvahy. Pri tejto morfologickej operácii je jadro k_1 posúvané po obraze I_e pre každý možný bod obrazu a v každej pozícii sú porovnané hodnoty bodov v jadre s korešpondujúcimi bodmi v obraze I_e. Ak body v obraze I_e presne odpovedajú hodnotám bodov v jadre, je aktuálny bod nastavený na 0. Ak sa aspoň jeden líši, ostáva aktuálny bod nezmenený. Po prevedení operácie s jadrom k_1 pre všetky body z I_e sa použije jadro k_2. Po aplikovaní oboch jadier sú jadrá rotované o 90° a upravené jadrá sú aplikované na obrázok znova. Celkovo teda stenčovanie používa 8 jadier. Táto operácia je cyklicky opakovaná a skončí vtedy, keď konverguje k tomu, že nie je zmenený žiaden bod obrazu I_e.

V testovanom obraze I_e obsahujúcom takto stenčené hrany sú následne vyhľadané kontúry a tieto kontúry sú rozdelené v bodoch s najväčšou krivosťou na menšie časti - kodóny.

Pre modelový obrázok sa základná množina kodónov $\mathcal{C}_{mo} = \{\mathcal{M}_1, ..., \mathcal{M}_l\}$ vytvorí podobne s tým, že kodóny sú orientované v smere hodinových ručičiek (pri zoraďovaní kodónov uvažujeme stred ako ťažisko hranových bodov modelového obrázka). Množina kodónov je rozšírená tak, že susediace trojice kodónov sú zoskupené do jedného kodónu a priradené do množiny. Novovytvorené kodóny sú ďalej zoskupované so susediacimi kodónmi a takto sa pokračuje dovtedy, kým

[4]Výstup je možné prehliadnuť si po spustení demo programu s parametrom `-fig=thin img.png`. Parametre pre ostatné ukážky sú na strane 52



Obr. 9: Výpočet momentu

množina neobsahuje prvok $\mathcal{M}$, ktorý je rovný kontúre celého objektu. Dôvod tohto zoskupovania je podrobnejšie vysvetlený v kapitole 4.5.

4.3 Moment operátor

Dôležitým krokom algoritmu je vyhľadať pozície a merítko potencionálnych objektov. Na to je využitý moment operátor predstavený v [27]. V tejto podkapitole je uvedená definícia operátora a sú predstavené jeho základné vlastnosti.

4.3.1 Definícia

Podľa fyzikálnej definície je moment vektorová veličina, ktorá vyjadruje mieru otáčavého účinku sily. Matematicky je moment definovaný ako vektorový súčin (*cross product*) vektora sily a vektora posunutia z bodu, v ktorom je moment počítaný do bodu, na ktorý je sila aplikovaná. Moment sily vypočítame ako

$$\vec{\tau} = \vec{r} \times \vec{F},$$

kde $\vec{\tau}$ je výsledný moment, $\vec{r}$ je vektor posunutia a $\vec{F}$ je vektor sily.

V ďalšom výklade často pracujeme s momentom na úrovni bodov obrázka, preto budeme používať označenie, kde p je stred momentu q je ľubovolný bod na obrázku, vektor $\vec{r}_{pq}$ je vektor smerujúci z bodu p do bodu q a $\vec{F}_q$ je vektor sily pre bod q (znázornené na obrázku 9). Pre obrázky je vektor sily definovaný ako jednotkový vektor, kolmý na gradient hrany (vektory gradientov môžeme získať pomocou Sobelovho operátora, ktorý je podrobne popísaný v kapitole 3.1). Trojrozmerný súradnicový systém môžeme na obrázku definovať tak, že osi x, y ležia na ploche tvorenej obrázkom a os z, kolmá na obrázok smeruje nahor. Pre zvolený stred momentu p môžeme pre ľubovolný bod obrázka vypočítať vektor momentu sily $\vec{\tau}$ ako

$$\vec{\tau}_{pq} = \vec{r}_{pq} \times \vec{F}_q \; . \tag{2}$$

Keďže vektory p, q ležia na ploche tvorenej obrázkom, ich vektorový súčin je vždy kolmý na obrázok a veľkosť momentu je vlastne určená veľkosťou z-tovej súradnice vypočítaného momentu. Preto s takto definovaným momentom pri trochu voľnejšom zápise môžeme pracovať ako so skalárnou hodnotou. Pri používaní termínu orientácia hrany berieme do úvahy aj smer hrany

(narozdiel od prístupu v kapitole 3.1), preto skalárna hodnota momentu nadobúda kladné aj záporné hodnoty. Znamienko je určené smerom vektora $\vec{r}_{pq}$. Veľkosť momentu je teda definovaná aj takto

$$\tau_{pq} = |\vec{r}_{pq}| \sin \theta_{pq} \ , \tag{3}$$

kde θ_{pq} je uhol, ktorý zvierajú vektory $\vec{r}_{pq}$ a $\vec{F}_q$.

Vektory $\vec{r}_{pq}$ a $\vec{F}_q$ ležiace v ploche obrázka majú zložku $z = 0$. Ak rozpíšme vektorový súčin po zložkách, vidíme, že po dosadení 0 za z-tové zložky vektorov $\vec{r}_{pq}$ a $\vec{F}_q$ do nižšie uvedenej rovnice, zložky $\vec{\tau}_x$ a $\vec{\tau}_y$ nadobúdajú nulovú hodnotu. Jediná zložka, ktorá nadobudne nenulovú hodnotu je $\vec{\tau}_z$, čo predstavuje skalárnu veľkosť momentu. Vektorového súčin je možné rozpísať po zložkách ako

$$\vec{\tau}_x = \vec{r}_y \vec{F}_z - \vec{r}_z \vec{F}_y, \quad \vec{\tau}_y = \vec{r}_z \vec{F}_x - \vec{r}_x \vec{F}_z, \quad \vec{\tau}_z = \vec{r}_x \vec{F}_y - \vec{r}_y \vec{F}_x \ .$$

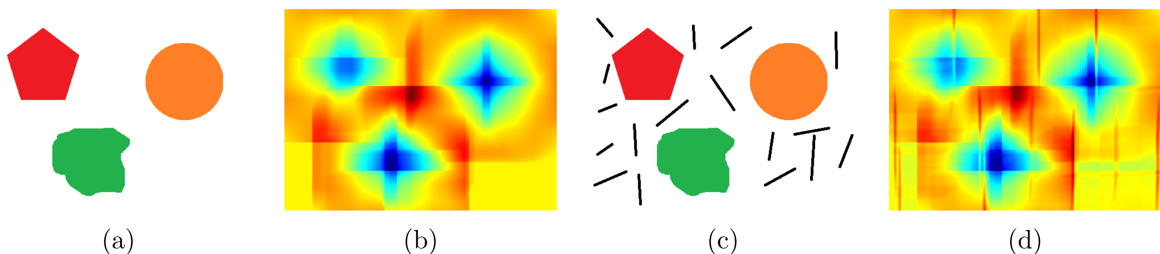
Moment operátor je definovaný ako suma momentov spočítaných pre všetky hranové body v určitej oblasti P, ktorá môže mať ľubovolný tvar. Vypočítaná veľkosť je normalizovaná hodnotou $2|P|$.

$$\tau_P = \frac{1}{2|P|} \sum_{q \in E(P)} \tau_{pq} \tag{4}$$

4.3.2 Vlastnosti moment operátora

Prvá vlastnosť moment operátora je zjavná z rovnice (3). Je jasne vidieť, že veľkosť výsledného momentu priamo úmerne závisí od dĺžky vektora $|\vec{r}_{pq}|$. V praxi to znamená, že ak sú hrany detekovaného objektu blízko k hranici prehľadávanej oblasti P, dĺžka vektoru $|\vec{r}_{pq}|$ nadobúda maximálnych hodnôt a tieto hranové pixely teda prispievajú do celkovej sumy najväčšou hodnotou.

Druhá vlastnosť súvisí s uhlom θ_{pq}, ktorý zviera vektor posunutia a vektor sily. Hranové body z oblasti P, ktorých vektor posunutia zviera s vektorom sily podobný uhol, prispievajú do celkovej sumy moment operátora väčším príspevkom. V ideálnom prípade - objekt tvaru kruhu so stredom v bode p všetky hranové body z oblasti zvierajú rovnaký uhol θ_{pq} a ich suma bude pre bod p nadobúdať maximálnu hodnotu. Naopak náhodné hrany, ktoré mohli v obrázku vzniknúť vplyvom šumu alebo vplyvom náhodných objektov v pozadí, sú týmto krokom odfiltrované.



(a) (b) (c) (d)

Obr. 10: Moment operátor vypočítaný pre jednoduché tvary

Na obrázku 10 je znázornený výpočet moment operátora pre jednoduché tvary. Test [5] bol prevedený na obrázku s rozmermi 500×350px. Prehľadávaná oblasť P má tvar štvorca so stranou o veľkosti 150px. Pre kódovanie farby je použitá farebná paleta z OpenCV `cv::COLORMAP_JET`. Obrázok 10a je vstupný obrázok a hodnoty funkcie τ_P k nemu sú na obrázku 10b. Najtmavšia modrá farba na mape hodnôt reprezentuje lokálne minimá. Vidíme, že minimá funkcie približne odpovedajú pozíciám objektov. Objekty sa nachádzajú v minimách preto, lebo sú reprezentované tmavou farbou na svetlom pozadí. Problémom pre tento jednoduchý obrázok ale je, že do úvahy musíme brať aj maximá funkcie pre prípad, že by detekovaný objekt bol reprezentovaný svetlou farbou na tmavom pozadí. Minimálne jeden potencionálny objekt by bol detekovaný v strede, medzi útvarmi. Preto tieto maximá nazývame len ako pozície potencionálnych objektov, a tieto pozície sú podrobené ďalšiemu spracovaniu.

Obrázok 10c znázorňuje podobný obrázok ako 10a s tým rozdielom, že v pozadí sú pridané náhodné hrany. Na obrázku 10d je výsledná mapa hodnôt, na ktorej môžeme pozorovať, že extrémy funkcie neboli ovplyvnené náhodnými hranami a boli by detekované rovnaké potencionálne objekty ako na obrázku 10b.

4.3.3 Optimalizovaný výpočet

Výpočet mapy hodnôt $V(x, y)$ pre všetky x, y a pre všetky $s \in \mathcal{S}$ je veľmi časovo náročná operácia jej výpočet je možné značne urýchliť využitím integrálnych obrazov. Pripomenieme si tento pomerne jednoduchý, ale veľmi účinný nástroj pre optimalizáciu výpočtu súm v obrázku.

Integrálny obraz je dátová štruktúra, ktorá v každom svojom bode obsahuje súčet hodnôt funkcie $k(x, y)$ od ľavého horného bodu $(0, 0)$. Integrálny obraz k vstupnému obrazu je ľahké vypočítať jedným prechodom všetkých bodov obrazu. Matematicky je integrálny obraz definovaný takto

$$K(x, y) = \sum_{u \leq x} \sum_{v \leq y} k(u, v) \ .$$

S využitím tejto definície je možné dvojrozmernú sumu v oblasti $(a...b, c...d)$ vyrátať ako

$$\sum_{u=a}^{b} \sum_{v=c}^{d} k(u, v) = K(a, c) + K(b, d) - K(a, d) - K(b, c) \ .$$

Už sme prezentovali vzorec pre výpočet hodnoty moment operátora. Sumu v tomto vzorci môžeme po drobných úpravách vypočítať pomocou integrálnych obrazov. Rovnica (4) sa po

[5] Výstup je možné prehliadnuť si po spustení demo programu s parametrom `-fig=torque img.png min max`. Parametre pre ostatné ukážky sú na strane 52

presunutí normalizačnej konštanty na ľavú stranu dá postupne rozpísať takto

$$\begin{aligned}\vec{\tau}_P \cdot 2|P| &= \sum_{q\in P} \overrightarrow{pq} \times \vec{F}_q \\ &= \sum_{q\in P} (\overrightarrow{po} + \overrightarrow{oq}) \times \vec{F}_q \\ &= -\overrightarrow{op} \times \sum_{q\in P} \vec{F}_q + \sum_{q\in P} \overrightarrow{oq} \times \vec{F}_q \ , \end{aligned} \tag{5}$$

kde bod p je stred prehľadávanej oblasti P, vektor $\overrightarrow{op}$ a $\overrightarrow{oq}$ sú vektory posunutia z bodu $(0,0)$ do bodu p, v respektíve q. Ak zapíšeme vektory z rovnice (5) po zložkách $\overrightarrow{op} = (x, y)$, $\overrightarrow{oq} = (u, v)$ a $F_q = (F_x(u, v), F_y(u, v))$, môžeme rovnicu prepísať ako

$$\vec{\tau}_P \cdot 2|P| = -x \sum_{(u,v)\in P} F_y(u,v) + y \sum_{(u,v)\in P} F_x(u,v) + \sum_{(u,v)\in P} uF_y(u,v) - \sum_{(u,v)\in P} vF_x(u,v) \ . \tag{6}$$

V rovnici (5) nie ja až tak ľahké spozorovať, že pre výpočet sa dajú použiť integrálne obrazy. V rovnici (6) to môžeme zbadať oveľa ľahšie. Výpočet súm pre zložky $F_x(u, v)$, $F_y(u, v)$, $vF_x(u, v)$, $uF_y(u, v)$, môžeme vypočítať vopred bez toho, aby sme vôbec vedeli, aká veľká bude oblasť P, pre ktorú budeme chcieť vypočítať hodnotu moment operátora. Veľkosť hodnoty $\vec{\tau}_P$ zistíme na základe štyroch bodov v lineárnom čase pre akúkoľvek oblasť P. Oproti tomu v prístupe bez využitia integrálnych obrazov by sme museli hodnotu momentu τ spočítať $W \times H \times |P|$ krát pre každé merítko oblasti z $s \in \mathcal{S}$.

Ďalšie zrýchlenie je podobne ako v kapitole 3 a v mnohých ďalších algoritmoch, dosiahnuté diskretizáciou hrán. Diskretizované orientácie hrán označíme θ_i, pričom $i \in \mathcal{E}$. Zložky vektora $\vec{F}_g$ určujúceho smer hrany sú $(\cos\theta_i, \sin\theta_i)$. Pre takto definované značenie je hodnota moment operátora so stredom $p = o = (0, 0)$ vypočítaná takto

$$\tau_{op} = \sum_{i\in\mathcal{E}} (x \sin\theta_i - y \cos\theta_i) \cdot \delta(q, i) \ , \tag{7}$$

kde $\delta(q, i) \in 0, 1$ je funkcia, ktorá vracia 1 ak sa v bode q nachádza hrana s diskretizovanou hranou s orientáciu θ_i. Potom môže byť výpočet moment operátora prepísaný ako

$$\vec{\tau}_P \cdot 2|P| = \sum_{i\in\mathcal{E}} \left((-x\sin\theta_i + y\cos\theta_i) \sum_{(u,v)\in P} \delta((u,v), i) \right) + \sum_{i\in\mathcal{E}} \sum_{(u,v)\in P} \tau_o((u,v), i) \ . \tag{8}$$

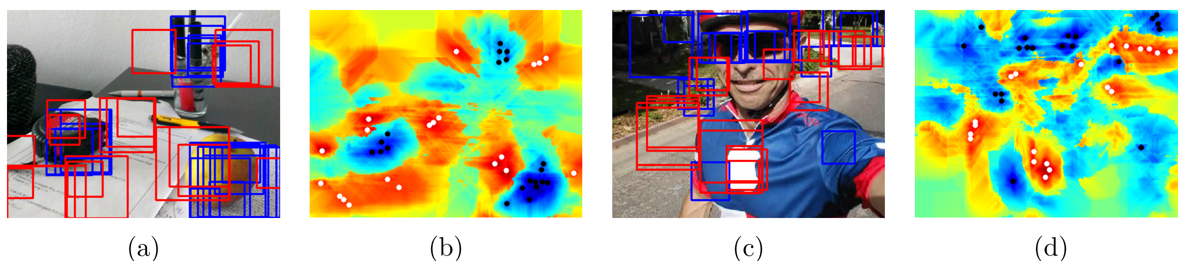
V praxi môžeme hodnoty $(-x\sin\theta_i + y\cos\theta_i)$ vypočítať pre každú orientáciu θ_i vopred. Takisto aj integrálne obrazy pre funkcie $\delta(q, i)$ a $\tau_o(q, i)$ pre každé θ_i. Možno sa zdá, že týchto predpočítaných hodnôt je priveľa, ale pre detekciu objektov s použitím moment operátora potrebujeme otestovať veľké množstvo oblastí P. Tieto integrálne obrazy nám zabezpečia výpočet pre akúkoľvek oblasť P v lineárnom čase.

4.3.4 Detekcia potencionálnych objektov

Moment operátor sa pri detekcii využíva tak, že jeho hodnota sa vypočíta pre všetky body x, y vstupného obrázka, pre rôzne veľkosti oblastí P_s, kde s je merítko z množiny $s \in \mathcal{S}$. Vypočítaním moment máp pre všetky x, y a s dostaneme $|\mathcal{S}|$ matíc s vypočítanými hodnotami momentu Celková mapa hodnôt $V(x, y)$ sa získa vyhľadaním extrémov naprieč parameterom s. Okrem toho je udržiavaná aj matica $S(x, y)$, v ktorej sú uložené merítka s, pri ktorých dané extrémy nastali. Matematicky môžeme mapu hodnôt $V(x, y)$ a mapu merítok $S(x, y)$ zapísať takto

$$
\begin{aligned}
V(x, y) &= \tau(x, y, \hat{s}(x, y)) \\
S(x, y) &= \hat{s}(x, y) \\
\hat{s}(x, y) &= \underset{s}{\operatorname{argmax}} |\tau(x, y, s)| \ .
\end{aligned}
$$

Ďalej v texte budeme používať aj termín moment mapa $T_I(x, y) = \{V(x, y), S(x, y)\}$. Pre ďalšiu detekciu vyhľadáme extrémy funkcie $V(x, y)$. Body x, y, v ktorých funkcia nadobúda extrémne hodnoty, tvoria pozície p_c čiastočne uzavretých kontúr - potencionálne objekty. Množinu potencionálnych stredov označíme $\mathcal{P}_c$. Merítko detekovaného objektu v bode p_c môžeme získať z mapy merítok $S(p_c)$. Pre extrém $p_c \in \mathcal{P}_c$ môžeme získať príspevky jednotlivých hranových pixelov $\vec{\tau}_{pq}$ použitím rovnice (2). Keď nastavíme pre príspevok hrán prah t_c a do úvahy berieme len body, v ktorých hodnota $\vec{\tau}_{pq}$ je väčšia ako prah t_c, pre každý stred p_c dostaneme množinu najviac prispievajúcich hrán $\mathcal{Q}_{pc}$. Na obrázku 11 sú zobrazené potencionálne objekty nájdené pomocou mapy hodnôt. Na mapách 11b, 11d sú minimá zobrazené čiernymi bodkami, maximá bielymi. V obrázkoch 11a, 11c červené štvorce predstavujú objekty detekované v lokálnych extrémoch funkcie, modré sú detekované v minimách. Vidíme, že na obrázku 11a by nás ďalej zaujímali objekty v lokálnych minimách, na obrázku 11c, ak by sme hľadali biele logo, by nás zaujímali objekty v maximách funkcie. Preto sa pri detekcii musíme zaoberať všetkými extrémami funkcie $V(x, y)$.



(a) (b) (c) (d)

Obr. 11: Detekcia potencionálnych objektov

V tejto podkapitole sme si ukázali, že pomocou moment operátora dokážeme vyhľadať množinu potencionálnych objektov. Tieto objekty sú vyhľadané len na základe všeobecných vlastností popísaných v podkapitole 4.3.2, a nie sú porovnané voči hľadanému objektu. Aby sme zabezpečili porovnávanie voči hľadanému objektu môžeme rovnicu pre výpočet momentu (2) rozšíriť

o funkciu $\vec{f}(\cdot)$, ktorá modifikuje vektor $\vec{F_q}$ na základe podobnosti testovanej hrany s modelom obrázka. A to tak, že veľkosť vektora $\vec{F_q}$ je väčšia ak je hrana podobná hľadanému objektu, v opačnom prípade je menšia, prípadne až nulová. Rovnicu rozšírenú o váhy hrán môžeme zapísať ako

$$\tau_{pq}^{\omega} = \vec{r}_{pq} \times \vec{f}(\vec{F_q}) \ , \tag{9}$$

Pre porovnávanie objektov v tak ako v [36] použijeme tvarový kontext (*shape context* [3]).

4.4 Tvarový kontext

Lokálne podobnosti dvoch tvarov môžeme porovnať pomocou tvarového kontextu, ktorý bol prezentovaný v [3]. Tvarový kontext je deskriptor, ktorý sa vypočíta pre vybrané body porovnávaných tvarov. Dva vizuálne podobné tvary by mali obsahovať v odpovedajúcich bodoch podobné tvarové kontexty. V uvedenom článku je tvarový kontext použitý predovšetkým na porovnávanie znakov, písmen a číslic. V tejto podkapitole ukážeme definíciu tvarového kontextu a jeho využitie na hranách získaných pomocou moment operátora. Tiež to, ako bola táto definícia rozšírená o rotačnú invarianciu a citlivosť bodov na polohu voči stredu objektu v článku [36].

4.4.1 Definícia

Základom tohto lokálneho deskriptora je, že na množine bodov $\mathcal{Q}_{p_c} = \{q_1, ..., q_n\}$ porovná relatívne vzdialenosti všetkých bodov q_i k ostatným $n-1$ bodom. Tieto vzdialenosti medzi bodmi sú porovnávané v log-polárnom súradnicovom systéme pomocou histogramov. Log-polárny systém je určený v rovine, kde súradnica ρ udáva logaritovanú vzdialenosť bodu q od stredu p systému a súradnica θ udáva uhol, ktorý zviera spojnica z bodu q do p a počiatok súradnicového systému (najčastejšie os x v karteziánskych súradniciach). Tento systém znázornený na obrázku 12e je veľmi podobný polárnemu súradnicovému systému, s tým rozdielom, že vzdialenosť bodu od stredu je v log-polárnom systéme naviac logaritmovaná. Prevod bežne používaných karteziánskych súradníc do log-polárnych súradníc je definovaný ako

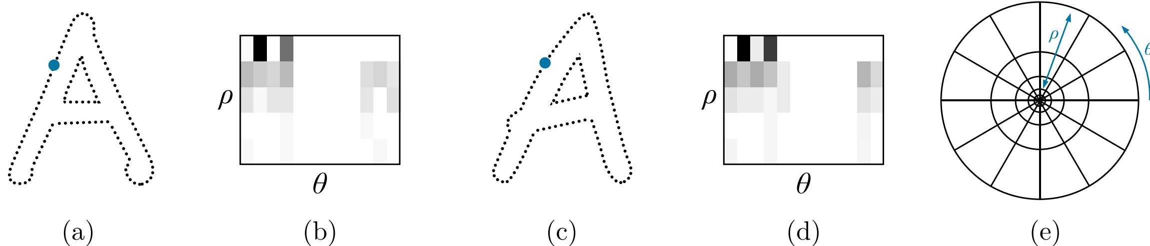
$$\begin{aligned} \rho &= \log \sqrt{x^2 + y^2} \\ \theta &= \arctan y/x \qquad \text{ak} \qquad x > 0 \ . \end{aligned}$$

Histogram $h_i^{sc}(k)$ pre tvarový kontext bodu q_i je určený takto

$$h_i^{sc}(k) = \#[q_j \neq q_i : (q_j - q_i) \in \text{bin}(k)], j \neq i \ . \tag{10}$$

Teda možné rozdiely bodov p a q v log-polárnych súradniciach sú rozdelené do k intervalov a pre každý interval v histograme sa uchováva počet bodov spadajúcich do jednotlivých intervalov.

Na obrázku 12 sú zobrazené dva tvarové kontexty pre veľmi podobné reprezentácie písmena A. Body, pre ktoré sú tvarové kontexty vyrátané sú zvýraznené modrou bodkou. Intervaly k



Obr. 12: Tvarové kontexty pre podobné tvary

sú zobrazené na obrázku 12e. Vidíme, že priestor k sa skladá z 5 intervalov pre logaritmovanú vzdialenosť ρ a 12 intervalov pre uhol θ. Tomuto odpovedajú veľkosti histogramov 12b a 12d. Oba histogramy sú veľmi podobné napriek tomu, že obrázky 12a a 12c sú mierne rozdielne. Môžeme z toho usúdiť, že tvarový kontext je invariantný voči malým deformáciám.

4.4.2 Rozšírenie tvarového kontextu o pozíciu stredu

Tvarový kontext, tak ako je definovaný v [3], neobsahuje žiadnu informáciu o pozícii stredu objektu. Teda histogramy sú spočítané pre všetky body útvaru a je jedno, kde v rámci objektu sa bod nachádza. V práci [36] je tvarový kontext $h_i^{sc}(k)$ rozšírený o uhlový tvarový kontext, v ktorom hodnoty histogramu sú vážené podľa pozície bodu $q_i \in \mathcal{Q}_{p_c}$ vzhľadom na pozíciu stredu p_c. Takto rozšírený tvarový kontext je označovaný ako momentový tvarový kontext a je vypočítaný ako

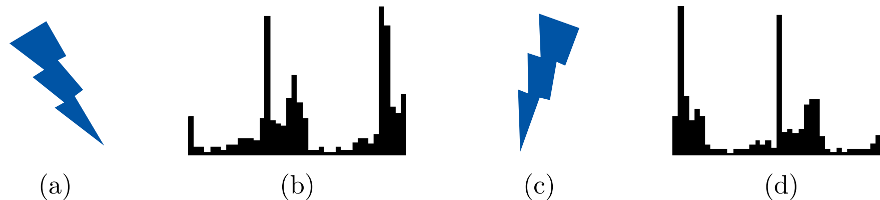
$$h_i^{\tau} = h_i^{sc}(k) + \mathcal{K}(\angle\text{bin}(k), \theta_{p_c q_i}) \; ,$$

kde $\angle\text{bin}(k)$ je uhlová zložka tvarového kontextu, $\theta_{p_c q_i}$ je uhol, ktorý zviera vektor $\vec{r}_{p_c q_i}$ a počiatok log-polárneho súradnicového systému O_g. A funkcia $\mathcal{K}(\cdot, \theta)$ je definovaná ako Gaussova funkcia $N(\theta_{p_c q_i}, \sigma_{\mathcal{K}}^2)$ orezaná do intervalu $[\theta_{p_c q_i} - \pi/2, \theta_{p_c q_i} + \pi/2]$. Táto funkcia zmení hodnoty uhlovej zložky, tak že najväčší vplyv majú intervaly histogramu smerujúce k stredu p_c a naopak nulový vplyv majú intervaly, ktoré smerujú na druhú stranu. Vďaka tomuto tvarový kontext neberie do úvahy hrany, ktoré sa v okolí objektu mohli detekovať náhodne zo šumu alebo zo zložitého pozadia.

4.4.3 Rotačná invariancia

Deskriptor použiteľný pre detekciu objektov by mal byť invariantný voči posunu, merítku a rotácii. To že je tvarový kontext invariantný voči malým rotáciám sme videli na obrázku 12. Je jasné, že tvarový kontext je invariantný voči posunu už z definície tvarového kontextu - všetky pozície sa medzi sebou porovnávajú len relatívne, v rámci jedného tvaru. Tvarový kontext tak ako sme ho definovali v rovnici (10) nie je invariantný voči merítku. Túto vlastnosť ale môžeme dostať tak, že všetky logaritmované vzdialenosti normalizujeme priemernou vzdialenosťou medzi bodmi v danom tvare.

Problémom tvarového kontextu ale je, že nie je rotačne invariantný. To znamená, že pre úspešnú detekciu objektu, ktorý je na testovanom obrázku inak rotovaný ako v modelovom obrázku, by sme museli prejsť všetky možné orientácie objektu, tak ako pre COF v kapitole 3. To ale nepripadá do úvahy, pretože kalkulácia a porovnanie tvarového kontextu je výpočetne náročná. Takisto by sme mohli rotačnú invarianciu dosiahnuť tým, že by sme počiatok log-polárneho súradnicového systému definovali relatívne k orientácii hrany v bode p, pre ktorý kontext počítame. To by ale nemuselo viesť k dobrým výsledkom, pretože presné orientácie hrán môžu byť znehodnotené šumom.



Obr. 13: Rotačná invariancia s použitím FFT

V implementácii sme využili spôsob s využitím Fourierovej transformácie, rovnako ako v článku [36]. Prvým krokom je vypočítanie 1D uhlovej zložky tvarového kontextu so stredom v ťažisku modelu pre kodóny modelového obrázka a uhlovej zložky pre podmnožinu kodónov testovaného obrázka v strede p_c. Uhlová zložka tvarového kontextu je vypočítaná podobne ako tvarový kontext, s tým rozdielom že pri log-polárnych súradniciach neberieme do úvahy zložku ρ, teda vzdialenosť od stredu súradnicového systému. Tým pádom sa dvojrozmerný priestor intervalov histogramu k mení na jednorozmerný. Uhlová zložka je teda vektor $\vec{a} = \langle h^{\angle}_{p_c}(1), ..., h^{\angle}_{p_c}(k) \rangle$. Na obrázkoch[6] 13a a 13c sú dva rovnaké objekty, pričom druhý je rotovaný o 49°. Pre oba obrázky sú spočítané uhlové zložky tvarového kontextu (13b, 13d) s počtom intervalov 40. Keďže sa jedná o rovnaké, rotované objekty, vektory uhlových zložiek sú potencionálne rovnaké, len posunuté o veľkosť uhla, ktorý bol použitý pri rotácii. Veľkosť tohto uhla potrebujeme zistiť. V terminológii spracovania signálov by sme povedali, že sa jedná o dva rovnaké signály a hľadáme veľkosť fázového posunu. Pre vyhľadanie veľkosti fázového posunu môžeme použiť fázovú koreláciu. Postup je nasledovný. Získať vektory uhlových tvarových kontextov pre model a testovaný objekt $\vec{a}_m$ a $\vec{a}_t$. Vypočítať Fourierovu transformáciu pre oba vektory: $\mathbf{G}_m = \mathcal{F}\{\vec{a}_m\}$, $\mathbf{G}_t = \mathcal{F}\{\vec{a}_t\}$. Po zložkách prenásobiť výsledky Fourierových transformácií a výsledok po zložkách normalizovať podľa

$$R = \frac{\mathbf{G}_m \circ \mathbf{G}_t^*}{|\mathbf{G}_m \circ \mathbf{G}_t^*|} \; .$$

Na výslednom obraze previesť inverznú Fourierovu transformáciu, čím vznikne normalizovaná kroskorelácia

$$r = \mathcal{F}^{-1}\{R\} \; ,$$

[6] Obrázky sú výstupom ukážkového programu po spustení s parametrom `-fig=fft img1.png img2.png`. Parametre pre ostatné ukážky sú na strane 52

potom výsledný fázový posun je

$$\Delta x = \underset{x}{\text{argmax}}(r) \ .$$

Výsledok Δx je počet uhlových intervalov o koľko sú dva obrazy posunuté. Previesť na uhol v radiánoch môžeme ako

$$o_g = \frac{2\pi\Delta x}{k} \ ,$$

kde k je veľkosť vektora $\vec{a}$, alebo počet uhlových intervalov v tvarovom kontexte. Týmto sme zistili veľkosť uhla, o ktorý bol objekt rotovaný. V praxi sa vypočíta táto veľkosť uhla pre kodóny modelu a pre kodóny z oblasti, ktorá je pomocou moment operátora označená ako potencionálny objekt a pred porovnaním tvarových kontextov modelu a objektu sa uhol o_g použije pre rotáciu testovaných kodónov.

4.4.4 Porovnanie kodónov pomocou tvarových kontextov

V tejto kapitole je uvedený spôsob akým sa porovnávajú dva tvarové kontexty. Pripomeňme si, že kodón je usporiadaná množina bodov. V tejto podkapitole je prezentovaná funkcia ϕ, ktorá ku každému bodu testovaného kodónu priradí bod z modelového kodónu. Cena tohto priradenia je definovaná ako

$$C_\phi(\mathcal{G}', \mathcal{M}) = \gamma_{sc} C_{sc}(\mathcal{G}', \mathcal{M}) + \gamma_\angle C_\angle(\mathcal{G}', \mathcal{M}) \ ,$$

kde $\mathcal{M}$ je kodón modelu, $\mathcal{G}'$ je kodón z testovaného obrázka, $C_{sc}(\cdot)$ je vzdialenosť tvarového kontextu a $C_\angle(\cdot)$ je vzdialenosť váženej uhlovej zložky tvarového kontextu. Vplyv tvarových kontextov je rozdelený pomocou váh $\gamma_{sc} + \gamma_\angle = 1$. Pre nájdenie funkcie ϕ musíme hodnotu ceny $C_\phi(\mathcal{G}', \mathcal{M})$ minimalizovať. To je možné dosiahnuť pomocou Viterbiho algoritmu, ktorý je popísaný v článku v [3]. Tento algoritmus pri ohodnocovaní funkcie ϕ berie do úvahy vzdialenosť dvoch tvarových kontextov, ale aj dva faktory naviac. Prvým je vzájomná pozícia dvoch susedných bodov. Ak je vzdialenosť medzi dvoma bodmi na testovanom kodóne malá (teda body sú susediace), musia byť aj odpovedajúce dva body v modelovom kodóne takisto susediace. Druhý faktor je krivosť kontúry v okolí bodu. To teda znamená, že hodnota krivosti v okolí bodu na testovanom kodóne musí byť podobná ako hodnota krivosti na odpovedajúcom bode v modelovom kodóne. Hodnota krivosti je vypočítaná ako $k(u_i) = \|g_{(i-1)} - 2g_i + g_{(i+1)}\|$.

Keďže tvarový kontext aj uhlová zložka tvarového kontextu sú histogramy, ich vzdialenosti sú porovnávané pomocou chí-kvadrát vzdialenosti. Vzorce pre $C_{sc}(\mathcal{G}', \mathcal{M})$ a $C_\angle(\mathcal{G}', \mathcal{M})$ sú prakticky identické, preto je zavedené označenie $C_t(\cdot) \in \{sc, \angle\}$, pomocou ktorého je možné zapísať oba vzorce. Vzdialenosť kodónov $\mathcal{G}'$ a $\mathcal{M}$ pre zložku t je definovaná ako suma chi-kvadrát vzdialeností medzi bodmi testovaného kodónu a ich odpovedajúcimi bodmi z modelového kodónu takto

$$C_t(\mathcal{G}', \mathcal{M}) = \sum_{i=1}^{n'} \chi^2(g_i^t, m_{\phi(i)}^t) \ ,$$

kde g_i^t a $m_{\phi(i)}^t$ sú body testovacieho kodónu a ich odpovedajúce body v modelovom kodóne a chí-kvadrát vzdialenosť je definovaná ako

$$\chi^2(g_i^t, m_{\phi(i)}) = \frac{1}{2} \sum_{k=1}^{K} \frac{[h_i^t(k) - h_{\phi(i)}^t(k)]^2}{h_i^t(k) + h_{\phi(i)}^t(k)} ,$$

kde $h(k)$ je početnosť k-teho intervalu v histograme. Hodnota porovnávacej funkcie $D_{sc}^{\tau}(\mathcal{G}', \mathcal{M})$ pre testovanú a modelovú kontúru je definovaná ako priemer cien C_ϕ v jednotlivých bodoch ako

$$D_{sc}^{\tau}(\mathcal{G}', \mathcal{M}) = \frac{1}{n'} \sum_{i=1}^{n'} C_{\phi(i)}(\mathcal{G}', \mathcal{M}) . \tag{11}$$

Použitím tejto funkcie sú kodóny, ktoré boli vyhľadané v mape hodnôt momentovej mapy porovnávané voči špecifickému objektu. Vieme, že kodón je časť kontúry rozdelená v oblasti vysokej krivosti, takže sa objekt neporovnáva ako celok ale porovnávajú sa len jeho časti. Toto je výhodou v prípade, že časť vyhľadávaného objektu je zatienená, alebo sú časti hrán objektu inak poškodené. Podrobný popis sa nachádza v ďalšej kapitole.

4.5 Detekcia objektov

Táto kapitola vysvetľuje celý princíp detekcie objektov s využitím moment operátora a tvarového kontextu. V nasledujúcom texte je princíp popísaný slovami, kde snahou je zachytiť všetky detaily. Celý algoritmus je potom zosumarizovaný v prílohe A.2.

Prvým krokom je výpočet moment mapy $T_I = V, S$ pre vstupný hranový obrázok I_e pre všetky merítka oblastí z množiny $s \in \mathcal{S}$. Vyhľadaním extrémov v mape hodnôt V, v kombinácii s mapou merítok S získame pozície a veľkosti potencionálnych objektov $\mathcal{P}$. Často krát sa stane, že objekt, ktorý chceme detekovať, je zložený z viacerých potencionálnych objektov, preto sú tieto prvotne detekované oblasti zoskupené so susednými oblasťami. Ako sme uviedli v kapitole 4.3.4, pre každý objekt sú vybrané hranové body $\mathcal{Q}_{p_c}$, ktoré prispievajú k extrémnej hodnote najvyššou hodnotou. Susediace oblasti $\mathcal{Q}_{r_c}$, ktorých stred r_c spadá do oblasti objektu $\mathcal{Q}_{p_c}$, sú zoskupené do oblastí $\mathcal{R}_{N_c}$, kde N_c je stred novovytvorenej oblasti, vypočítaný ako ťažisko hranových bodov v oblasti.

Na základe hranových bodov v oblasti $\mathcal{R}_{N_c}$ je vytvorená množina testovacích kodónov $\mathcal{C}_g = \{\mathcal{R}'_1, ..., \mathcal{R}'_d\}$. Tieto testovacie kodóny sú ďalej, podobne ako v kapitole 4.2, zoradené v smere hodinových ručičiek a porovnávané s kodónmi modelu $\mathcal{C}_{mo} = \{\mathcal{M}_1, ..., \mathcal{M}_l\}$. Testovacie kodóny nie sú porovnávané len po jednom, ale sú vytvorené aj rôzne kombinácie s $\mathcal{J}$ susednými kodónmi. Kombinácie pre prvý kodón $\mathcal{R}'_1$ môžu byť zapísané takto: $\{\mathcal{R}'_{\{1\}}, ..., \mathcal{R}'_{\{1+\mathcal{J}\}}\}$. Každá z týchto vytvorených kombinácií sa porovnáva s kodónmi modelu využitím rovnice (11) dosadením $D_{sc}^{\tau}(\mathcal{R}'_{\{1\},...\{1+\mathcal{J}\}}, \mathcal{M}_{\{1\},...,\{l\}})$. Tento výpočet je opakovaný pre všetky kodóny z $\mathcal{C}_g$, čím vzniká matica hodnôt D_{sc}^{τ} o rozmeroch $W \times H \times d \times \mathcal{J} \times l$. Rozmer matica nabáda k dojmu, že zložitosť algoritmu sa bude pohybovať v rade $O(n^5)$. V skutočnosti tomu tak nie je, pretože veľkosť $\mathcal{J}$ je

volená ako malé číslo z rozsahu 3 až 5 a čísla l a d sú tiež malé, menšie ako 10. Uložením minimálnych hodnôt skrz rozmery $d \times \mathcal{J} \times l$ do matice $E_{D^{\tau}_{sc}}$ s rozmerom $H \times W$ dostaneme najmenšie vzdialenosti testovaných kodónov pre jednotlivé hranové body r_i. Vzdialenosti sú prevedené na váhy pomocou rovnice

$$W_{D^{\tau}_{sc}}(r_i) = \beta_c + \beta_f(\exp(-E^{\tau}_{D_{sc}}(r_i)/2\sigma))) \;, \tag{12}$$

kde $\beta_c + \beta_f = 1$ a β_c je vlastne hodnota, ktorou budú ohodnotené hrany, ktoré nemajú žiadnu podobnosť s hľadaným objektom. Tieto váhy súžia ako funkcia $\vec{f}(\cdot)$, ktorá mení veľkosť jednotkového vektoru v rovnici (9). Nahradením $\vec{f}(\cdot)$ v rovnici (9) získavame rovnicu pre výpočet veľkosti váženého momentu, ktorý je citlivý na tvar špecifického modelového objektu

$$\tau^{\omega}_{pq} = \vec{r}_{pq} \times (W_{D^{\tau}_{sc}}(q)\vec{F}_q) \tag{13}$$

a zároveň rovnicu pre výpočet váženého moment operátora

$$\tau^{\omega}_{P} = \frac{1}{2|P|} \sum_{q \in E(P)} \tau^{\omega}_{pq} \;. \tag{14}$$

Po získaní váh $W_{D_{sc}}$ je opakovaný postup pre výpočet moment mapy, tentoraz ale s využitím váženého operátora τ^{ω}_{P}. Vyhľadané extrémy $p^{\omega}_{c} \in \mathcal{P}^{\omega}$ v mape hodnôt váženej moment mapy predstavujú konečné stredy detekovaných objektov. V príslušnej mape merítok je možné zistiť ich merítka. V praxi je opakovaný výpočet moment mapy veľmi rýchly, pretože dátové štruktúry pre výpočet moment operátora sú uložené v pamäti a pri počítaní s váženými hranami sa môžu využiť.

Po prečítaní tejto kapitoly by malo byť pomerne jasné ako algoritmus využíva *bottom-up* prístup pri hľadaní potencionálnych objektov záujmu a potom využíva *top-down* prístup pri porovnávaní kodónov s konkrétnym modelom, ako je využitý moment operátor a tvarový kontext. Algoritmus dosahuje dobré výsledky detekcie pre zložité scény čo je prezentované v kapitole 7. Výpočetná zložitosť algoritmu je ale stále vysoká aj napriek tomu, že je prezentovaný optimalizovaný výpočet moment mapy. Nehovoriac o tom, že porovnávanie tvarových kontextov je ešte náročnejšia operácia. Preto by sme tento algoritmus mohli len ťažko využiť pre detekciu objektov v reálnom čase.

5 Ostatné algoritmy

V tejto kapitole sú veľmi stručne popísané dva algoritmy: *fast directional chamfer matching* (ďalej FDCM)[23] a LINE-2D[20]. Cieľom tejto práce nebolo tieto algoritmy implementovať, pretože sú voľne prístupné v knižnici OpenCV. Cieľom je porozumieť týmto algoritmom a predstaviť ich základné koncepty, pretože veľa z nich je spoločných s tými, ktoré využívajú algoritmy implementované v tejto práci.

Detektory FDCM a LINE-2D sú síce už staršieho dátumu, ale v mnohých publikáciách sú uvádzané ako *"state of the art"*, teda detektory s najlepšími výsledkami. A aj dnešné detektory [36][22][39] dosahujú len o niečo lepšie výsledky detekcie ako LINE-2D prípadne jeho alternatíva s rozšírením o hĺbkovú mapu: LINE-MOD. Takisto implementácia a výsledky tejto práce sú v kapitole 7 porovnané voči týmto dvom algoritmom.

5.1 Fast Directional Chamfer Matching

Chamfer matching ako metóda porovnávania obrazov je známa už od vydania článku [1]. Odvtedy bola táto základná metóda rozšírená napríklad o orientácie hrán [17]. V článku [23] je predstavený algoritmus FDCM, ktorý skracuje dobu výpočtu, pričom dosahuje ešte lepšie výsledky ako jeho predchodcovia. Porovnávacia funkcia hľadá pre body modelu najbližšie korešpondujúce body v testovanom obrázku, pričom zohľadňuje aj ich orientáciu. Pre zrýchlenie výpočtu algoritmu sú hrany v algoritme reprezentované ako segmenty a využívajú sa zaujímavé optimalizačné štruktúry.

5.1.1 Chamfer matching

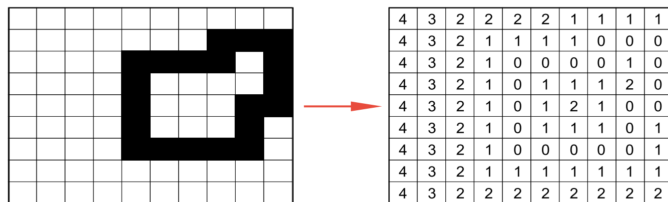
Vzdialenosť medzi hranovým obrazom modelu U a testovaným obrazom V je v základnom algoritme *chamfer matching*(CM)[1] definovaná ako aritmetický priemer vzdialeností medzi bodmi $u_i \in U$ a najbližšieho hranového bodu v obraze V ako

$$d_{CM}(U,V) = \frac{1}{|U|} \sum_{u_i \in U} \min_{v_j \in V} |u_i - v_j| \ .$$

Takto definovaná vzdialenosť medzi obrazmi má tendenciu detekovať nesprávne objekty v zložitých scénach s rušivým pozadím. Preto autori rozširujú bežný *chamfer matching* tak, že pre každý bod x je definovaná orientácia $\phi(x)$ a do výpočtu vzdialeností je pridaný rozdiel takto definovaných orientácií. Tento prístup je pomenovaný *directional chamfer matching (DCM)* a je definovaný takto

$$d_{DCM}(U,V) = \frac{1}{|U|} \sum_{u_i \in U} \min_{v_j \in V} |u_i - v_j| + \lambda |\phi(u_i) - \phi(v_j)| \ ,$$

kde λ je váha vplyvu orientácií na celkovú vzdialenosť.



Obr. 14: Výpočet mapy vzdialeností

5.1.2 Mapa vzdialeností

Samozrejme vyhľadávania najbližšieho bodu v_i pre bod u_i hrubou silou je výpočetne náročná operácia a počet operácií je kvadraticky závislý na počte bodov modelového a testovaného obrázka. Pre zrýchlenie je využitý koncept mapy vzdialeností (*distance transform / distance map (DT)*), čo je obraz, ktorý je vypočítaný pre testovaný obraz V a v každom bode obsahuje hodnotu vzdialenosti k najbližšiemu bodu. Takýto obraz je pri použití vhodného algoritmu [25] možné vypočítať jediným priechodom cez všetky body obrazu V. Mapa vzdialeností je určená takto

$$DT_V(x) = \min_{v_j \in V} |x - v_j| \ .$$

Na obrázku 14 je znázornený výpočet mapy $DT_V(x)$ pre jednoduchý útvar. Pomocou takto definovanej mapy $DT_V(x)$ môžeme získať veľkosť vzdialenosti k najbližšiemu bodu v konštantnom čase, len odčítaním hodnoty z mapy. Mapa $DT_V(x)$ môže ale byť použitá len pre obyčajný CM, teda bez ohľadu na orientácie. Preto sú orientácie diskretizované do q orientácií v rozsahu $(0, \pi)$ podobne ako v kapitole 3. Funkcia, ktorá pre daný bod vráti jej kvantizovanú orientáciu je označená $\hat{\phi}(x)$. Mapa vzdialeností, ktorá berie ohľad na orientácie je definovaná ako

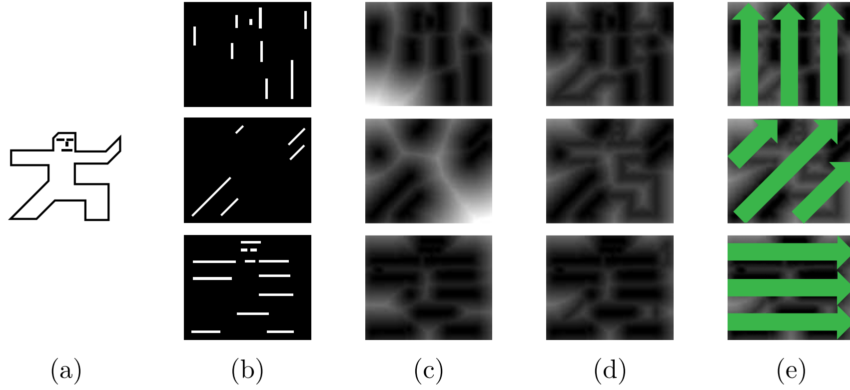
$$DT3_V(x, \phi(x)) = \min_{v_j \in V} |x - v_j| + \lambda |\hat{\phi}(x) - \hat{\phi}(v_j)| \ .$$

V článku [23] je popísaný spôsob ako vypočítať mapu $DT3_V(x, \phi(x))$ pomocou q priechodov obrázka V. Hodnota funkcie $d_{DCM}(U, V)$ môže byť z takto vypočítanej mapy vypočítaná ako

$$d_{DCM}(U, V) = \frac{1}{n} \sum_{u_i \in U} DT3_V(u_i, \hat{\phi}(u_i)) \ .$$

5.1.3 Integrálny obraz mapy vzdialeností

Mapa hrán V je zložená z jednotlivých bodov reprezentujúcich vysoké zmeny jasu na obrázku. Počet bodov môže byť veľký, preto sú hrany s podobnými orientáciami zoskupené do hranových segmentov pomocou RANSAC [16] algoritmu. Týmto sa výrazne zníži počet bodov, ktorý sa musí spracovať. Hranové segmenty sú teda v obraze reprezentované množinou $L_U = \{l_{[s_j, e_j]}\}_{j=1\ldots m}$, kde s_j a s_e sú začiatočný a koncový bod segmentu l_j.



Obr. 15: Výpočet máp vzdialeností a ich integrálnych obrazov

Pre ďalšie urýchlenie výpočtu je použitý integrálny obraz vypočítaný na mape vzdialeností (podobne ako v kapitole 4.3.3). Keďže počet máp vzdialeností je rovný počtu kvantizovaných orientácií q, potrebujeme integrálny obraz vypočítať pre každú orientáciu $\hat{\phi}_i$. Aby sa integrálna mapa dala použiť pre zistenie sumy pre segmenty z L_U, sú jednotlivé integrálne obrazy integrované v smere prislúchajúcej orientácie ϕ_i. Výpočet integrálneho obrazu pre mapu vzdialeností je určený ako

$$IDT3_V(x_i, \hat{\phi}_i) = \sum_{x_j \in l_{[x_0, x]}} DT3_V(x_j, \hat{\phi}_i) \ .$$

Na obrázku 15 je znázornený postup výpočtu integrálnych obrazov pre jednoduchý obrázok 15a. Najprv sú orientácie diskretizované, v tomto prípade len do 3 orientácií (obrázok 15b). Pre každú orientáciu je vypočítaná mapa vzdialeností $DT_V(x)$ (obrázok 15c). Mapy vzdialeností sú rozšírené o orientáciu hrán a je vypočítaná $DT3_V(x)$ (obrázok 15d). Nakoniec sú obrazy integrované v smere orientácie (obrázok 15e).

V integrálnom obraze $IDT3_V$ môžeme zistiť hodnotu $d_{DCM}(U, V)$ podľa

$$d_{DCM}(U, V) = \frac{1}{n} \sum_{l_{[s_j, e_j]} \in L_U} \left[IDT3_V(e_j, \hat{\phi}(l_{[s_j, e_j]})) - IDT3_V(s_j, \hat{\phi}(l_{[s_j, e_j]})) \right] \ .$$

Hodnotu $d_{DCM}(U, V)$ pre jeden segment teda dokážeme zistiť v konštantnom čase, len odčítaním hodnôt $IDT3_V$ pre začiatočný a koncový bod. To znamená, že pre všetky segmenty získame hodnotu $d_{DCM}(U, V)$ v $O(m)$ čase, kde m je počet segmentov v testovanom obrázku. To je veľký rozdiel oproti spôsobu, ktorý sme predstavili na začiatku, kde pre každý bod u_i by sa museli prejsť všetky body testovaného obrázka v_i.

Článok [23] znamenal posun v oblasti porovnávania obrazov. Rozširuje metódu CM, kde k porovnávacej funkcii pridáva orientácie gradientu. S použitím tejto funkcie dosahuje lepšie výsledky ako predtým prezentované metódy. Zároveň však ukazuje veľmi efektívny spôsob výpočtu, ktorým je možné kalkuláciu zrýchliť až 45-krát. Celkovo je tento algoritmus veľmi zaujímavý a ukazuje možnosti optimalizačných štruktúr ako sú mapa vzdialeností a integrálne obrazy.

5.2 Line-2D

Tento algoritmus je vlastne zdokonalenie algoritmu DOT[19] od tých istých autorov. Hlavná porovnávacia funkcia pri tomto algoritme porovnáva orientácie gradientov pre modelový a testovaný obrázok. Pre zrýchlenie využíva diskretízáciu orientácií, lookup tabuľky vopred vypočítaných hodnôt a vhodné uloženie dát v pamäti, ktoré umožňuje optimalizáciu pomocou SSE inštrukcií. Algoritmus COF, ktorého implementácia je súčasťou tejto práce a ktorého princíp je popísaný v kapitole 3, je inšpirovaný práve algoritmom LINE-2D.

5.2.1 Porovnávacia funkcia

Prvým krokom je extrakcia orientácie gradientov, rovnako ako v kapitole 3.1. Základom metriky pre ohodnotenie podobnosti medzi modelovým obrázkom a testovaným obrázkom je suma rozdielu orientácií gradientov. Táto metrika bola prvý krát prezentovaná v [35] a je vypočítaná ako

$$\mathcal{E}_S(\mathcal{I}, \mathcal{T}, c) = \sum_{r \in \mathcal{P}} |\cos(\mathrm{ori}(\mathcal{O}, r) - \mathrm{ori}(\mathcal{I}, c + r))| \ ,$$

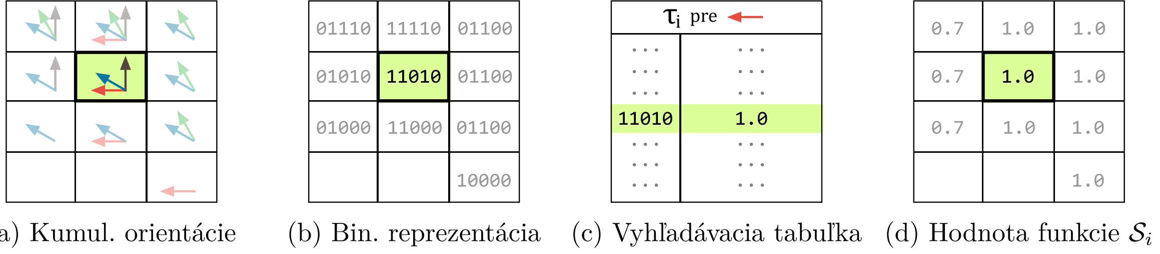
kde $\mathcal{T} = (\mathcal{O}, \mathcal{P})$ je deskriptor hľadaného objektu zložený z modelového obrázka $\mathcal{O}$ a množiny bodov $\mathcal{P}$, ktorá sa berie do úvahy pri detekcii. Funkcia $\mathrm{ori}(\mathcal{O}, r)$ vracia orientáciu gradientu v bode r modelového obrázka $\mathcal{O}$, podobne $\mathrm{ori}(\mathcal{I}, c+r)$ je orientácia gradientu v bode c posunutom o r.

Pri konštrukcii deskriptora $\mathcal{T}$ sa vyberá množina $\mathcal{P}$ bodov z modelového obrázka $\mathcal{O}$ na základe veľkosti gradientu, čo vytvorí množinu bodov, ktoré nesú najväčšiu informáciu. Vidíme, že pri výpočte porovnávacej funkcie $\mathcal{E}_S$ sa už veľkosť gradientu nepoužíva. Takto je možné detekovať aj objekty, ktorých hrany sa strácajú v zložitom pozadí. Celkovo funkcia podobnosti $\mathcal{E}_S$ je vhodná pre detekciu objektov na zložitom pozadí, zlyháva ale pri detekcii objektov, ktoré sú rotované alebo deformované čo i len v malom rozsahu. Preto autori článku [20] predstavili mierne upravenú porovnávaciu funkciu, ktorá je odolná voči malým deformáciám a rotáciám

$$\mathcal{E}(\mathcal{I}, \mathcal{T}, c) = \sum_{r \in \mathcal{P}} \left(\max_{t \in \mathcal{R}(c+r)} |\cos(\mathrm{ori}(\mathcal{O}, r) - \mathrm{ori}(\mathcal{I}, t))| \right) \ , \tag{15}$$

kde $R(c + r) = [c + r - \frac{T}{2}, c + r + \frac{T}{2}] \times [c + r - \frac{T}{2}, c + r + \frac{T}{2}]$ je okolie bodu veľkosti T, so stredom v bode $c+r$. Takže pre každý bod modelu r je prehľadané jeho okolie korešpondujúcich bodov v testovanom obrázku a do úvahy sa berú tie body, ktorých orientácia gradientu je najviac podobná orientácii bodu r. Týmto je dosiahnutá odolnosť voči malým rotáciám a deformáciám.

Pri implementácii by sme mohli prehľadať pre každý bod r jeho okolie, spočítať kosínus rozdielu orientácií pre všetky body v okolí a vyhľadať maximum. Toto by ale viedlo k zložitosti v $O(|J| \times |P| \times T)$. Pre urýchlenie výpočtu sa preto postupuje nasledujúcim spôsobom. Orientácie gradientov sú prevedené na bitové slová a kumulované do obrázka $\mathcal{J}$ podobne, ako sme vysvetlili v kapitole 3.2. S tým rozdielom, že v tomto prípade nie sú obrazy transformované do n náhod-



(a) Kumul. orientácie (b) Bin. reprezentácia (c) Vyhľadávacia tabuľka (d) Hodnota funkcie $\mathcal{S}_i$

Obr. 16: Vyhľadanie maximálnej hodnoty kosínusu rozdielu uhlov pre orientáciu (←)

ných pozícií, ale obrazy sú len posúvané v oblasti $[-\frac{T}{2}, +\frac{T}{2}] \times [-\frac{T}{2}, +\frac{T}{2}]$. Ďalej sa vypočítajú vyhľadávacie tabuľky $\mathcal{S}_i$ (*lookup table*) ukladajúce hodnoty funkcie (15) pre všetky možnosti bitovo kódovaných kumulovaných orientácií a všetky diskretizované orientácie gradientu $\text{ori}(\mathcal{O}, r)$. Vyhľadávacia tabuľka τ_i pre každú z kvantizovaných orientácií sa vypočíta ako

$$\tau_i[\mathcal{L}] = \max_{l \in \mathcal{L}} |\cos(i - l)| ,$$

kde i je index kvantizovanej orientácie (pre jednoduchosť notácie i predstavuje aj samotný uhol). A $\mathcal{L}$ sú kumulované orientácie z okolia gradientu, teda bod z obrázka $\mathcal{J}$. Pre každú orientáciu i je možné maximálnu hodnotu kosínusu rozdielu uhlov pre bod c jednoducho prečítať z príslušnej vyhľadávacej tabuľky $\mathcal{S}_i$ ako

$$\mathcal{S}_i(c) = \tau_i[\mathcal{J}(c)] .$$

Nakoniec môžeme teda dostať hodnotu funkcie (15) takto

$$\mathcal{E}(\mathcal{I}, \mathcal{T}, c) = \sum_{r \in \mathcal{P}} \mathcal{S}_{\text{ori}(\mathcal{O}, r)}(c + r) .$$

Na obrázku 16 je znázornený postup pre vyhľadanie hodnoty funkcie $\mathcal{S}_i$ pre orientáciu v smere (←). Kumulované orientácie 16a sú v pamäti binárne reprezentované ako na obrázku 16b. Podľa binárneho kódu (`11010`) sa vyhľadá hodnota funkcie $\mathcal{S}_i$ vo vyhľadávacej tabuľke τ_i (obrázok 16c) pre príslušnú orientáciu (←). Konečná hodnota funkcie $\mathcal{S}_i$ je na obrázku 16d.

V ďalšom texte článku [20] autori popisujú spôsob uloženia vyhľadávacích tabuliek $\mathcal{S}_i$. Dnešné procesory nečítajú dáta z pamäti po jednom bajte ale po zoskupených bajtoch, ktoré sú ukladané do cache, je teda vhodné dáta v pamäti organizovať tak, aby procesor nemusel na bajty pristupovať na náhodné miesta do pamäte, ale mohol pracovať s dátami, ktoré sú uložené za sebou v cache. Vyhľadávacie tabuľky $\mathcal{S}_i$ sú teda v pamäti linearizované (odtiaľ aj názov LINE-2D), tzn. dáta sú uložené za sebou tak, aby ich procesor mohol efektívne prečítať. Navyše ak sú dáta uložené za sebou, môže sa výpočet ešte viac zrýchliť použitím SSE inštrukcií.

Algoritmus LINE-2D teda upravuje porovnávaciu funkciu, aby bola odolná voči malým deformáciám a posunom a navyše predstavuje koncept vyhľadávacích tabuliek, čím sa dosahuje neporovnateľné zrýchlenie výpočtu.

6 Praktická implementácia

V tejto kapitole je stručne popísaný postup pri implementovaní porovnávacích metód. V úvode kapitoly sú popísané konvencie dodržiavané v celom zdrojovom kóde aplikácie. V ďalšom texte sú spomenuté použité nástroje a knižnice. V zbytku kapitoly sú popísané jednotlivé moduly aplikácie, predovšetkým riešenie netriviálnych problémov pri implementácii. Pred čítaním tejto kapitoly je vhodné prečítať si teoretické kapitoly 3, 4 a 5, pretože implementácia sa často odkazuje na teoretické detaily predstavené v týchto kapitolách a často sú uvedené korešpondencie medzi matematickým zápisom a zápisom v zdrojovom kóde. Výsledná aplikácia `DiplomaThesis` je na CD priložená vo forme *solution* pre Visual Studio 2015 a ako spustiteľný súbor spolu s `bat` súbormi, ktoré program spúšťajú s rôznymi parametrami.

6.1 Všeobecné informácie

Prvým krokom pri implementácii praktickej časti práce bol výber vhodných nástrojov a knižníc. Tieto boli z časti určené zadaním práce, ale pri implementácii boli vyžité aj iné nástroje. Ich využitie spolu s dodržiavanými konvenciami je zhrnuté v tejto kapitole.

6.1.1 Dodržiavané konvencie

Celá práca bola implementovaná v jazyku C++. Pri implementácii sme sa snažili dodržiavať konvencie *Google C++ Style Guide*[18], ktoré dodržiavajú všetky open-source projekty zastrešované firmou Google. Konvencie sú jasné a striktné, definujú nielen mennú konvenciu, ale aj tzv. *best practises*, ktorých dodržiavaním sa kód stáva veľmi dobre čitateľný a dobre udržiavateľný. To, aby bol kód dobre čitateľný, je obzvlášť dôležité u projektu ako prezentujeme v tejto práci. Teda projekt, ktorý okrem mňa ako autora budú čítať pedagógovia alebo ostatní študenti. V nasledujúcom texte je vymenovaných pár konvencií z množstva, ktoré sú v zdrojovom kóde aplikácie dodržiavané.

Používanie hlavičkových súborov je neoddeliteľnou súčasťou projektu, ak má byť projekt uvoľnený ako statická alebo dynamická knižnica. Všetky potrebné triedy sú deklarované v hlavičkových súboroch a pri použití týchto tried v inom projekte je nutné pridať cestu k týmto hlavičkovým súborom. Ich názvy odpovedajú názvu triedy, ktorú definujú. Rozdielom v názve je to, že pre názov triedy sme použili tzv. *camel case*, teda každé nové slovo v názve triedy je zvýraznené veľkým písmenom, pre názvy súborov je použité pomenovanie, kde všetky písmená sú písané malým písmom a slová sú oddelené podtržníkmi. Každý hlavičkový súbor v projekte obsahuje aj ochranu voči viacnásobnému vloženiu do zdrojovného kódu pomocou direktív preprocesoru `#ifndef`, `#define`. Do hlavičkových súborov sú pomocou direktív `#include` vložené len nezbytne nutné hlavičkové súbory. Snažíme sa vyvarovať tomu, že trieda zbytočne použije hlavičkový súbor obsahujúci všetky deklarácie OpenCV, namiesto toho radšej vložíme napr. len `opencv2/core/mat.hpp`. Takisto nie sú v hlavičkových súboroch použité príklazy `using`

`namespace`, pretože tieto príkazy by boli vykonané vo všetkých `cpp` súboroch, ktoré hlavičkový súbor používajú a ľahko by mohlo dôjsť k menným konfliktom. Vyhýbame sa tomu, aby hlavičkové súbory obsahovali definície metód, jedinou výnimkou sú generické metódy, pri ktorých pre správnu funkciu musí byť definícia práve v hlavičkovom súbore.

Definície jednotlivých tried sa nachádzajú v `cpp` súboroch s rovnakým názvom. Dodržiavame menné konvencie tak ako sú definované v [18], názvy tried, metód a typov používajú *camel case*. Výnimkou tvoria jednoduché metódy, ktoré napríklad len vracajú referenciu na objekt. Ich názvy sú písané malými písmenami. Názvy privátnych členov tried sú označované malými písmenami s podtržníkom na konci.

Aplikácia a aj všetky statické knižnice sú kompilované s použitím príznaku `\WX`. Ten nastavuje, že správy, ktoré by sa pri normálnej kompilácii zobrazili len ako upozornenia, berie ako chyby. Takže pri výskyte obyčajného upozornenia sa program alebo knižnicu nepodarí skompilovať. Toto nastavenie nám pomáha dodržiavať všetky zásady pri písaní dobrého kódu. Jedným z príkladov je priradenie hodnoty typu `double` do premennej typu `float`. Bez použitia príznaku `\WX` je program skompilovaný s tým, že do výstupu sa zobrazí chyba. Podobné prípady môžu viesť k ťažko dohľadateľným chybám. Tým sa dostávame k tomu, že premenné pretypovávame tak, ako to radia konvencie [18], teda používame C++ zápisy ako napríklad `static_cast<float>(hodnota)`. Táto definícia je jasnejšia v porovnaní s C operátormi `int(a)` alebo `(int)a`, kde nie je moc jasné, či sa vykoná operácia pretypovania alebo konverzie.

Štandard [18] hovorí o tom, že tam, kde je to vhodné, môžu sa využívať rozšírenia jazyka C++11. Projekty programované v tejto práci sú kompilované pomocou Visual Studia 2015, ktorého kompilátor obsahuje plnú podporu pre štandard C++11. V našom programe sme z týchto výhod využili len pár, v ďalšom texte vymenujeme najčastejšie používané z nich

- Veľmi často využívanou výhodou štandardu C++11 je zjednodušený zápis cyklu `for` pre iterovateľné objekty ako `std::vector` alebo iný podobný objekt. Tento zápis je veľmi pohodlný a čitateľný, pričom jeho výkonnosť je rovnaká ako jeho ekvivalent, obyčajný cyklus `for`.

- V programe využívame pre paralelizáciu knižnicu `tbb`[37]. Do jej paralelizovaných metód je nutné predať objekty s preťaženým operátorom `operator()`, tzv. funktory alebo lambda výrazy. V našom prípade sme sa rozhodli pre využitie lambda výrazov, ktorých zápis je výrazne kratší. V celom programe sa snažíme, aby bolo kopírovaných čo najmenej objektov, preto aj pri využívaní lambda výrazov predávame do *capture listu* parametere pomocou referencie.

- Ďalšia výhoda je kľúčové slovo `override`, ktoré je použité pri deklarácii metód, ktoré nahradzujú alebo implementujú virtuálnu metódu predka, kde je označená kľúčovým slovom `virtual`. Kombinácia týchto kľúčových slov zaistí, že metóda bude skutočne implementovať metódu predka a nevytvorí sa nová metóda. Týmto môžeme predísť zbytočným nepríjemnostiam a problémom pri hľadaní chýb v programe.

- V C++03 je povolené využívanie inicializačných listov `{1, 2}` napríklad pre polia. Štandard C++11 tento koncept rozširuje a umožňuje použitie inicializačných listov pre rôzne objekty, napríklad štandardné kontajnery ako `std::vector`. Túto výhodu v aplikácii často využívame.

6.1.2 Použité knižnice

Štandardná knižnica jazyka C++ je veľmi bohatá na rôznorodé funkcie, mnoho funkcií ale v nej chýba. Preto je bežnou praxou používanie rôznych knižníc. Pre projekt zaoberajúci sa spracovaním obrazu a počítačovým videním je jasná voľba knižnica OpenCV[28]. Jedná sa o *open-source* knižnicu, ktorá obsahuje množstvo užitočných funkcií od načítania obrázku, jeho zobrazenia, cez nespočetné množstvo matematických operácií až po celé algoritmy pre detekciu ako LINE-2D (kapitola 5.2), FDCM (kapitola 5.1) alebo SIFT[24], SURF[2] a pod. V aplikácii je použitá posledná verzia knižnice OpenCV 3.1.0.

Keďže sme sa pri implementácii zaoberali aj optimalizáciu a snažili sme sa o zrýchlenie výpočtu, hľadali sme knižnicu, s použitím ktorej by sa dal program paralelizovať pre viacjadrové procesory. Zvolili sme knižnicu Intel Threading Building Blocks (ďalej TBB). Táto knižnica je pre naše potreby možno zbytočne rozsiahla, voľba na ňu ale padla preto, lebo jej online dokumentácia je veľmi zrozumiteľne popísaná a paralelizovaný program je možné napísať pomocou jednoduchých príkazov. Takisto sme použili najnovšiu verziu, to je pre TBB verzia 4.4.

Obe dynamické knižnice sú priložené na CD v adresári `\demo\bin` tak, aby ich mohol použiť ukážkový program `demo.exe`. Všetky Visual Studio projekty na priloženom CD majú nastavené relatívne cesty k hlavičkovým súborom oboch knižníc. Ukážková aplikácia má v `Post-Build Event` nastavený príkaz pre prekopírovanie potrebných `dll` knižníc do výstupného adresára. Takto je možné Visual Studio projekt spustiť aj bez toho, aby na počítači bolo nainštalované OpenCV alebo TBB.

6.1.3 Použité nástroje

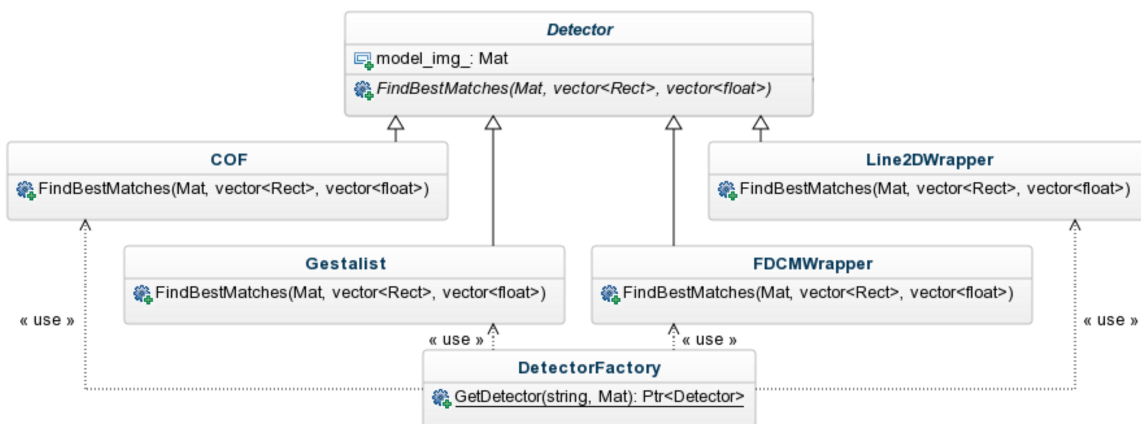
Pre vývoj ukážkovej aplikácie bolo použité prostredie Visual Studio 2015. Vývoj v tomto prostredí je veľmi pohodlný a prostredie obsahuje základné veci ako je našepkávanie (*intellisense*) možných typov a premenných pri písaní kódu, generovanie prázdnych častí kódu pre cykly, deklarácie tried a podobne. V prostredí je veľa ďalších pomôcok a nástrojov, bohužiaľ väčšina z nich je len pre programovanie .NET aplikácií s využitím manažovaného kódu. Pre C++ Visual Studio v základnej inštalácii postráda viac možností pre generovanie kódu, navigáciu v kóde a podobne. Našťastie množstvo funkcií je možné doinštalovať pomocou rozšírení.

Veľmi dobré rozšírenie pre Visual Studio je ReSharper C++[29]. Do vývojového prostredia dodáva veľké množstvo funkcií. Pri písaní kódu rozšírenie analyzuje celkový program a v reálnom čase ponúka užitočné informácie alebo návrhy na úpravu kódu, tzv. *quick-fixes*. Napríklad pri použití neznámej triedy ponúkne možnosti: či chceme pridať `#include` direktívu, vygenerovať

telo novej triedy a podobne. Vzhľadom na to, že v C++ máme celkovú voľnosť v tom, kde triedu deklarujeme, kde ju definujeme, ReSharper môže len hádať, kam napr. telo triedy chceme definovať. Vo väčšine prípadov sú ale odhady správne. Ďalšou vecou, ktorú Visual Studio postráda, je zrýchlená navigácia v kóde. ReSharper pridáva klávesovú skratku, ktorá vyvolá vyhľadávacie dialógové okno, do ktorého môžeme písať názvy tried, metód, súborov. Ihneď pri písaní sú ponúkané možnosti a je možné rýchlo prejsť na vybraný symbol alebo súbor. Veľmi mocným nástrojom je vyhľadávanie použití symbolov a s tým spojený refaktoring kódu. Premenovávanie vybraných symbolov prebieha veľmi rýchlo a má ďalšie možnosti ako napríklad premenovanie symbolov nájdených aj v zakomentovanom kóde. Jedinou nevýhodou ReSharperu je, že sa jedná o platené rozšírenie. Pre študentov je však po zadaní školského emailu na stránke projektu[29] zadarmo.

Pri vývoji aplikácií, ktoré pracujú s knižnicou OpenCV[28], je určite vhodné do Visual Studia inštalovať rozšírenie s názvom Image Watch. Je všeobecne známe, že v ladiacom móde (*debugging*) môžeme pozastaviť beh programu, krokovať beh po jednotlivých riadkoch a volaniach funkcií a nahliadnuť do obsahu premenných. Vďaka tomu môžeme odhaliť a opraviť veľa nepríjemných chýb. Horšie je to ale s ladením aplikácií, ktoré pre svoju funkciu využívajú obrázky, uložené v maticiach typu `cv::Mat`. Pre ich zobrazenie musíme neustále volať funkciu `cv::imshow` alebo si vytvárať iné testovacie metódy. Rozšírenie Image Watch umožňuje pri pozastavenom behu programu zobraziť obsah matíc typu `cv::Mat`, pričom obsahuje aj ďalšie funkcie, ktoré by sme museli pracne programovať - normalizovanie a zobrazenie jasovej zložky obrázka, zobrazenie jasovej zložky zakódovanej v pseudofarbách vo farbenej škále RGB, alebo uloženie obsahu matice do obrázka typu `jpg`, `png` alebo priamy binárny obsah obrázka do súboru typu `bin`. Po prejdení kurzorom nad zobrazený obrázok sa zobrazujú číselné hodnoty v príslušných bodoch. Pri mnohonásobnom zväčšení zobrazuje maticu ako tabuľku, s presným zobrazením číselných hodnôt v jednotlivých bodoch.

Pre vytvorenie dokumentácie k praktickej časti práce bol použitý nástroj Doxygen[11], ktorý na základe zdrojového kódu a komentárov v ňom dokáže vygenerovať html stránku a LaTeX dokument. Podľa základných nastavení Doxygenu je vygenerovaná dokumentácia len pre triedy a ich verejné metódy. Pre náš prípad je vhodnejšie, aby dokumentácia obsahovala aj privátne atribúty a metódy. Preto sme pri generovaní dokumentácie použili nastavenia, ktoré zahrnú všetky symboly. Dokumentácia je na priloženom CD v adresári `\doxygen`. Manuálne vypisovanie hlavičiek pre okomentované metódy a funkcie ve veľmi pracné, preto sme pri písaní komentárov pre funkcie použili rozšírenie Atomineer, ktoré dokáže prázdne hlavičky pre komentované metódy vygenerovať jedným kliknutím. Ich obsah je potom ľahké zmeniť a rozšírenie sa potom znova postará o formátovanie textu, odsadenie a zalamovanie riadkov. Atomineer tiež bohužiaľ nie je zadarmo, ale ponúka 30 dňovú skúšobnú verziu, ktorá na vygenerovanie hlavičiek bohate stačí. Okrem iného rozšírenie veľa nastavení, my sme použili také nastavenia, aby bolo formátovanie komentárov podobné tým, ktoré sú v knižnici OpenCV.



Obr. 17: Triedny diagram implementácie návrhového vzoru továreň

6.2 Architektúra aplikácie

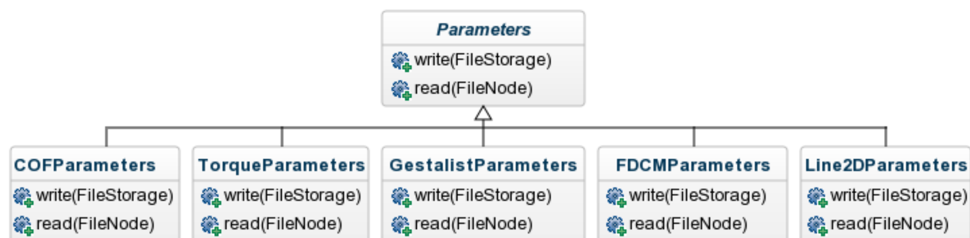
Riešenie sa skladá zo šiestich projektov, ktorými je program rozdelený do logických celkov. Päť z týchto projektov sú statické knižnice, ktoré sa ľahko prenositeľné a dali by sa použiť v akomkoľvek C++ projekte. Posledným projektom je spustiteľná demo aplikácia, ktorá využíva ostatné statické knižnice a prezentuje výsledky dosiahnuté v tejto práci.

6.2.1 Projekt Common

Prvý projekt, ktorý si rozoberieme je statická knižnica s názvom `Common`. Ako názov napovedá, táto knižnica obsahuje funkcionalitu, ktorá je spoločná pre všetky ostatné knižnice.

Trieda `Detector` je abstraktná trieda, z ktorej dedia všetky implementované detektory. Na obrázku 17 je znázornený diagram tried. Sú zobrazené len naozaj nutné atribúty a metódy. Triedy samozrejme obsahujú viac metód a atribútov. Všetky triedy, ktoré dedia z triedy `Detector`, musia implemetovať metódu `FindBestMatches()`. Táto metóda vlastne realizuje vyhľadávanie modelových obrázkov v testovacích obrázkoch. Metóda ma jeden vstupný a dva výstupné parametere. Vstupným parametrom je testovací obrázok typu `cv::Mat` a výstupné parametre sú vektor obsahujúci detekované obdĺžnikové oblasti a vektor obsahujúci skóre jednotlivých detekcií. Vektory musia mať rovnakú veľkosť a sú zoradené tak, že detekovaný objekt s najvyšším skóre je vo vektoroch na pozícii `[0]`. Obrázok modelu je do detektorov predávaný priamo v konštruktore a je uložený do atribútu `model_img_` bázovej triedy. Dôvodom je, že pri ohodnocovaní algoritmov vyhľadávame jeden model vo viacerých testovacích obrázkoch. Keď obrázok modelu predáme do konštruktoru, deskriptor pre neho môže byť vypočítaný vopred a v rôznych testovacích obrázkoch využívať stále ten istý deskriptor. Odpadá tak výpočet deskriptora modelu pre každý jeden testovací obrázok.

Na konštruovanie detektorov využívame triedu `DetectorFactory`, ktorú sme umiestnili do projektu `Demo`. Ak by bola trieda umiestnená v projekte `Common`, musel by projekt referencovať



Obr. 18: Triedny diagram parametrov

všetky ostatné projekty, a tým by sme vytvorili nepríjemný problém zvaný kruhové referencie. Ako jej názov napovedá, trieda `DetectorFactory` implementuje návrhový vzor továreň. Statická metóda `GetDetector()` slúži na získanie ukazovateľa na objekt, ktorého bázovým typ je `Detector`. Konkrétne to, aký typ detektora bude skonštruovaný, sa rozhodne na základe prvého parametera metódy, ktorý obsahuje cestu k `xml` súboru s nastaveniami porovnávacej metódy. Druhým parametrom je obrázok modelu typu `cv::Mat`. Ten sa predá do konštruktora konkrétneho detektora a bude sa vyhľadávať v testovacích obrázkoch

Spoločná funkcionalita je dobre reprezentovaná aj v triede `Logger`. Je to trieda využívaná všetkými ostatnými projektami a slúži pre výpis informácií o priebehu programu na štandardný výstup a zároveň do súboru `log.txt`. Textový súbor je otvorený v móde `append`, takže všetky nové správy je možné nájsť na konci tohto súboru. Trieda obsahuje metódy na meranie času `StartTimer()` a `EndTimer()`. Počas behu programu môže byť spustených viacero časovačov a ich hodnoty sú merané pomocou funkcií zo štandardnej knižnice `chrono`.

Ako sme uviedli v kapitole 6.1.2, pre paralelizáciu používame knižnicu TBB. Zápis jej metód je pomerne jednoduchý, ale my sme medzi knižnicou TBB a našimi knižnicami vytvorili ďalšiu vrstvu, ktorá je definovaná pomocou funkcií deklarovaných v súbore `for_helper.h`. Týmto sme dosiahli praktický a jednotný zápis cyklov `for`. Funkcie majú názvy `for_1d`, `for_2d`, `for_mat` a `for_vec`. Z ich názvov je možné usúdiť ich účel. Telo funkcie, ktoré sa bude vykonávať paralelne je predávané pomocou lambda výrazov. Každá z vytvorených funkcií má parameter s názvom `parallel` typu `bool`. Týmto príznakom je možné kontrolovať, či sa má funkcia vykonávať paralelne. Takto je možné program spustiť s nastavením, aby sa nevykonávali funkcie paralelne a jednoducho porovnať časy alebo zaťaženie procesorov pri behu s knižnicou TBB alebo bez nej.

Pri oboch implementovaných algoritmoch často potrebujeme v matici prehľadať obdĺžnikovú oblasť veľkosti T so stredom v bode p. Napr. pri vyhľadávaní lokálnych extrémov, pri výpočte moment operátora, alebo pri porovnávaní deskriptora COF s testovaným obrázkom. Na zjednodušenie zápisu týchto situácií je vytvorená trieda `RectIterator`. Konštruktor triedy prijíma dva parametere: stred oblasti p a veľkosť oblasti T. Následne je iterátor možné použiť zápisom `for(PointPair &pp: iterator)`, kde `pp.first` obsahuje bod v súradnicovom systéme matice (počiatok je v ľavom hornom rohu) a `pp.second` obsahuje bod relatívny na prehľadávanú oblasť (počiatok v ľavom hornom rohu oblasti).

Každá z porovnávacích metód má veľké množstvo nastavení, ktoré ovplyvňujú jej správanie.

V prvotnom návrhu sme nastavenia predávali do konštruktora po jednej hodnote, čo viedlo k tomu, že konštruktor mal aj 10 vstupných parametrov. Nehovoriac o tom, že počas vývoja programu sa počet parametrov menil a bolo niekoľkokrát nutné signatúru konštruktora prepísať. Preto sme parametre zlúčili do štruktúr. Pri takomto prístupe sa do konštruktora predá len jeden objekt obsahujúci všetky parametre. Naviac sú potom parametre ľahko serializované a deserializované do `xml` súborov pomocou OpenCV modulu `Persistence`. Na to, aby štruktúra mohla byť serializovaná, musí implementovať metódy `write()` a `read()`. Pre zjednotenie bola vytvorená abstraktná štruktúra `Parameters` a štruktúry pre jednotlivé porovnávacie algoritmy z tejto štruktúry dedia (zobrazené na obrázku 18).

Trieda, ktorá implementuje stenčovanie hrán popísané v kapitole 4.2, má názov `ThinHelper`. Vykonávanie afinných transformácií na obrázku je uľahčené pomocou triedy `AffineHelper`. Prevod akejkoľvek hodnoty na RGB pseudofarbu v palete `cv::COLORMAP_JET` je realizovaný pomocou triedy `ColorHelper`. Ostatné súbory v tejto knižnici definujú makrá, konštanty a ďalšie pomocné funkcie, ktoré je zbytočné popisovať v tomto texte. Ich podrobný popis je možné nájsť v Doxygen dokumentácii.

6.2.2 Projekt COF

Obsahom projektu s názvom COF je detektor objektov naprogramovaný podľa kapitoly 3. Najprv popíšeme triedu `Discretizer`, ktorá slúži na vyhľadanie orientácií gradientov a ich následnú diskretizáciu. Pre objekt sú spočítané orientácie gradientov pomocou Sobelovho operátora implementovaného v OpenCV vo funkcii `cv::Sobel()`. Magnitúdy gradientu sú získané pomocou funkcie `cv::magnitude()`. Orientácie sú následne diskretizované pomocou triedy `Discretizer`. Diskretizácia do 8 intervalov je prevedená pomocou metódy `BinaryDiscretize()`, kde orientácie sú reprezentované jedným bajtom. Jednotlivé bity predstavujú prítomnosť/neprítomnosť orientácie. Trieda obsahuje aj pomocú metódu `ShowAngles()`, použitím ktorej je možné zobraziť interaktívnu ukážku diskretizovaných orientácií gradientov.

Trieda, v ktorej sa nachádza značná časť implementácie, má názov `COF`. Jej konštruktor prijíma dva parametre: maticu obsahujúcu obrázok modelu a inštanciu štruktúry `COFParameters`. Vstupný obrázok modelu musí byť v stupňoch šedi, kde nulové pixely predstavujú pozadie a nenulové pixely predstavujú hľadaný objekt. Štruktúra `COFParameters` obsahuje všetky potrebné nastavenia[7]. Algoritmus používa pre rotačnú invarianciu a nezávislosť na merítku prístup, kde sú vopred vygenerované všetky jeho možné rotácie a zmeny merítka. Toto sa deje hneď v konštruktore, kde je objekt transformovaný na základe parametrov `min_scale`, `max_scale`, `scale_step` a `rot_step_deg` a všetky transformácie sú uložené do vektora `models_`. Pre každý z týchto obrázkov sa vypočíta deskriptor, definovaný triedou `COFDescriptor`. Matematicky je výpočet deskriptora podrobne popísaný v kapitole 3.2. Pri výpočte deskriptora je matica so vstupným obrázkom zase transformovaná, tentoraz je počet transformácií určený parametrom

[7]Všetky parametre sú podrobne popísané v doxygen dokumentácii priloženej na CD

`n_modifications`. Objekt je rotovaný o náhodne zvolený uhol v rozsahu obmedzenom parametrom `desc_angle_deg` a náhodne posunutý v intervale, ktorý je obmedzený parametrom `desc_translation`. Pre každý transformovaný obrázok sú extrahované orientácie vďaka triede `Discretizer`. Takto získané orientácie sú prechádzané pre každý objekt a sú kumulované do atribútu `angles_`. Početnosť orientácií pre každý pixel je uchovávaná v matici `weights_`. Nakoniec sú početnosti normalizované tak, že sú podelené celkovým súčtom početností. Týmto spôsobom dostaneme n deskriptorov pre každú možnú orientáciu a merítko objektu v obrázku. Na porovnanie deskriptora voči testovaciemu obrázku slúži metóda `Match()`, ktorá vychádza z rovnice (1). Vstupným parametrom pre túto metódu je matica obsahujúca diskretizované hrany testovaného obrázka a bod (x, y), pre ktorý bude spočítaná hodnota podobnosti. Hodnota podobnosti je spočítaná ako suma váh bodov, ktorých orientácia z testovaného obrázka sa nachádza v kumulovaných orientáciách deskriptora.

Keďže trieda `COF` implementuje abstraktnú triedu `Descriptor`, musí implementovať metódu `FindBestMatches()`. Pomocou tejto metódy sú vyhľadané objekty, ktorých porovnávacia funkcia je vyššia ako minimum nastavené parametrom `detection_threshold`. Vyhľadávanie bodov s najvyššou hodnotou funkcie podobnosti prebieha pri nastavení parametra `optimized = false` hrubou silou. Pri tomto výpočte je hodnota funkcie $s(x, y, i)$ vypočítaná pre každý pixel a uložené v matici typu `cv::Mat`. Pozície detekovaných objektov sú tvorené extrémami v tejto matici. Na základe pozícií extrémov a veľkosti deskriptorov je naplnený výstupný vektor `rects`. Hodnoty extrémov sú uložené do výstupného vektora funkcie `extrema` (ich vyhľadanie je uľahčené použitím triedy `ExremaFinder`). Pre malú optimalizáciu tohto vyhľadávania je možné využiť fakt, že pri výpočte deskriptora je posúvaný v rozsahu `desc_translation` pixelov t a orientácie sú kumulované. Nie je teda potrebné prechádzať všetky pixely, stačí prejsť len každý t-tý pixel.

Vyhľadávanie je oveľa zaujímavejšie pri nastavení parametra `optimized = true`. Vtedy je pre vyhľadávanie použitý algoritmus *coarse to fine strategy* z kapitoly 3.4. Pomocou metód `InitTestPyramid()` a `InitModelPyramid()` sú skonštruované pyramídy pre deskriptory a pre testovaný obrázok. Porovnávacia logika je v tomto prípade vykonávaná vo funkcii `MatchPyramids()`. Využitím tejto metódy nie sú prehľadávané všetky body vstupného obrázka, ale vždy len tie, ktoré sú uložené v premennej `points`. Vyhľadávanie začína na najvyššej vrstve, pre túto vrstvu sú do premennej `points` vložené všetky body najnižšej vrstvy pyramídy (pri dobre nastavených vstupných parametroch je tento počet oveľa menší ako počet bodov vo vstupnom obrázku). Pre všetky body z `points` je vypočítaná hodnota funkcie $s(x, y, i)$ a vyhľadané extrémy sú uložené do vektora `extrema`. Následne sú pomocou funkcie `PointsForNextPyramidLevel()` body z `extrema` prevedené na ich odpovedajúce body v nižšej vrstve pyramídy a prehľadávanie vrstiev pyramídy pokračuje, až kým sa neprehľadá aj najnižšia vrstva. Extrémy na poslednej vrstve tvoria pozície detekovaných objektov a sú podobne ako v odseku vyššie naplnené do výstupných parametrov funkcie `rects` a `extrema`.

6.2.3 Projekt Torque

Tento projekt obsahuje triedu `Torque`, ktorá implementuje výpočet moment mapy predstavenej v kapitole 4.3. Vstupným parametrom pre konštruktor je inštancia štruktúry `TorqueParameters`. Pomocou tejto štruktúry je možné predať všetky potrebné nastavenia. Jedinou funkcionalitou, ktorú konštruktor vykonáva je, že na základe vstupných parametrov `patch_size`, `patch_min_scale`, `patch_max_scale`, `patch_step_size_px` vytvorí vektor merítok oblastí `patch_sizes_`, pre ktoré sa vypočíta moment mapa. V matematickom zápise tento vektor odpovedá množine merítok $s \in \mathcal{S}$.

Výpočet moment mapy je realizovaný pomocou funkcie `Compute()`. Jej jediným parametrom je vstupný obrázok, ktorý môže byť RGB alebo v stupňoch šedi. Na obrázku sa vyhľadajú hrany s použitím funkcie `cv::Canny()`. Parametre ovplyvňujúce detekciu hrán sú `canny_t1` a `canny_t1`. Orientácie gradientov sú vypočítané pomocou Sobelovho operátora `cv::Sobel()`. Orientácie hrán sú kolmé na orientácie gradientov, takže orientácia hrán je dosiahnutá odčítaním $\pi/2$ od uhla orientácie gradientu.

Pre výpočet moment mapy sú naprogramované dva spôsoby. Prvý spôsob nie je optimalizovaný a veľkosť moment operátora sa vypočíta hrubou silou. Tento výpočet sa spustí pri nastavení parametra `optimized = false`. Výpočet pre jedno merítko oblasti z $\mathcal{S}$ je v tomto prípade naprogramovaný presne tak, ako je uvedený v rovnici 4, teda pre každý bod p je hodnota moment operátora získaná ako suma všetkých bodov q z definovaného okolia P. Pre jeden bod je jeho moment získaný pomocou `GetEdgeContribution()`. Tento výpočet je nutné opakovať pre všetky merítka oblastí z $\mathcal{S}$ a výsledky pre jednotlivé merítka sú ukladané v poli matíc `torque_all_patches_`. Nakoniec sú v tomto poli matíc vyhľadané extrémy naprieč merítkom oblasti s a sú vytvorené matice: mapa hodnôt $V(x, y)$ a mapa merítok $S(x, y)$. V zdrojovom kóde im odpovedajú matice `value_map_` a `scale_map_`. Mapa hodnôt obsahuje hodnoty momentov typu `float`, mapa merítok obsahuje veľkosti oblastí, prislúchajúcich k extrémom typu `cv::Size`. Tento postup je veľmi neefektívny, už pre malé obrázky sa jeho výpočet pohybuje v desiatkach sekúnd, preto je naprogramovaný aj optimalizovaný výpočet, ktorý sa spustí pri nastavení vstupného parametra `optimized = true`.

V optimalizovanom výpočte je prvým krokom diskretizácia hrán do poľa matíc `discrete_edges_`. Veľkosť tohto poľa je daná počtom intervalov, do ktorých sú hrany diskretizované. Veľkosť matíc odpovedá veľkosti vstupného obrázka. Ak je orientácia hrany v bode x, y diskretizovaná na uhol θ_i, ktorý je programe reprezentovaný indexom `i`, je na pozíciu `discrete_edges_[i].at(x, y)` v poli matíc nastavená hodnota `1`.

Ďalším krokom je predpočítanie hodnôt $(x \sin - y \cos)$ z rovnice (7) pre všetky diskretizované orientácie do poľa matíc `x_sin_y_cos_`. S využitím tohoto poľa matíc sú pre všetky orientácie vypočítané hodnoty momentu so stredom v bode nula τ_{op} podľa rovnice (7). Následne sú pomocou funkcie `cv::integral()` spočítané integrálne obrazy `integral_edges_` a `integral_torque_` pre diskretizované hrany a momenty. Tieto nám umožnia rýchly výpočet veľkosti momentu pre

akýkoveľvek rozmer oblasti P s využitím vzorca (8). Podobne ako v neoptimalizovanom spôsobe sú hodnoty moment operátorov pre všetky veľkosti oblastí z množiny $\mathcal{S}$ uložené do vektora matíc `torque_all_patches_`. Nakoniec sú získané matice `value_map_` a `scale_map_` rovnako ako v odseku vyššie.

K mape hodnôt a mape merítok je možné pristupovať pomocou metód `value_map()` a `scale_map()`. Jedným z krokov detekčného algoritmu z kapitoly 4 je vypočítanie mapy hodnôt s použitím obrazu váh jednotlivých hranových pixelov. Toto je v kóde umožnené použitím funkcie `SetWeightsAndRecompute()`, ktorej parametrom je matica, odpovedajúca váham hrán $W_{D_{sc}^{\tau}}$ z rovnice (12). Pri optimalizovanom spôsobe je už tento výpočet veľmi rýchly, pretože polia matíc sú uložené v atribútoch objektu.

6.2.4 Projekt Gestalist

Detektor objektov popísaný v kapitole 4 je implementovaný v projekte `Gestalist`. Časti algoritmu pre konštruovanie kodónov a ich porovnávanie pomocou tvarového kontextu (kapitola 4.2) sú implementované v pomocných triedach. Úvod tejto podkapitoly je venovaný práve týmto triedam.

Kontúry modelového aj prehľadávaného obrázka sú v algoritme reprezentované pomocou kodónov. Tie v implementácii predstavuje vektor bodov `vector<cv::Point>`. Pre ich vytáranie slúži trieda `CodonFactory`, ktorej konštruktor prijíma maticu obsahujúcu hrany obrázka. Trieda obsahuje metódu `ConstructCodons()`, ktorej výstupným parametrom je vektor kodónov `codons`. Vyhľadávanie začína vyhľadaním kontúr v obrázku hrán použitím funkcie `cv::findContours()`. Vyhľadané kontúry sú rozdelené v miestach s veľkou hodnotou krivosti, ktorá je vyrátaná pomocou triedy `Curvature`. V kontúrach vyhľadaných pomocou funkcie `cv::findContours()` sú niektoré pixely duplikované, niekedy sú duplikované celé kodóny. Preto sme vytvorili funkciu `RemoveDuplicates()`, ktorá tieto duplikáty odstráni. Jej priebeh je pomerne jednoduchý. Vytvorí sa matica, ktorej všetky prvky sú na začiatku inicializované na 0. Následne sa prechádzajú body všetkých kodónov, pričom pri každom navštívení bodu je hodnota na príslušnej pozícii v matici inkrementovaná. Ak sa v kodóne nachádzajú body, ktorých počet navštívení je väčší ako 1, je kodón odstránený. Trieda navyše obsahuje funkcie pre ďalšiu prácu s kodónmi: ich rotáciu, zobrazovanie a pod.

Podobnosť medzi dvoma kodónmi je určená pomocou tvarového kontextu. Teoretický základ je podrobne popísaný v kapitole 4.4. V zdrojovom kóde je táto logika implementovaná v triede `ShapeContext`. Jej základným parametrom pri konštrukcii je počet intervalov, do ktorých bude rozdelený log-polárny súradnicový priestor k. Do konštruktoru sú tieto parametre predávané pomocou parametrov `n_angular_bins` a `n_distance_bins`. Vypočítanie tvarového kontextu pre jeden bod kodónu je realizované pomocou metódy `GetShapeContext()`. Výstupom je matica obsahujúca histogram rozdielov v log-polárnych súradniciach medzi bodom `reference_point` a všetkými ostatnými bodmi z kodónu `codon`. V práci sa používajú aj ďalšie typy tvarových kontextov: uhlový a vážený uhlový. Pre ich výpočet slúžia metódy

GetAngularSC() a GetWeightedAngularSC(). Pre váženie uhlového tvarového kontextu sa využíva Gaussova funkcia, ktorej hodnoty sú predpočítané pri inicializácii do premennej gauss_. Vzdialenosť medzi dvoma tvarovými kontextami je vypočítaná pomocou chi-kvadrát vzdialenosti implementovanej vo funkcii DistanceSC(). Konečná vzdialenosť dvoch kodónov (kapitola 4.4.4) je s použitím Viterbiho algoritmu počítaná vo funkcii DistanceCodons(). Funkcia vracia vektor, kde zložky vektoru typu float odpovedajú najmenším vzdialenostiam pre jednotlivé body vstupného kodónu. Ak je tejto funkcii predaný voliteľný parameter phi_out, je matica funkcie ϕ, ktorá ku každému bodu testovaného kodónu priradí bod s najmenšou vzdialenosťou z kodónu modelu, vrátená použitím tohto výstupného parametra. Následne je možné jej výsledok zobraziť pomocou metódy ShowPhi(). Táto metóda bola často používaná pri ladení aplikácie, ukážková demo aplikácia zobrazí výsledok funkcie ShowPhi() pri spustení s parametrom -fig=sc.

Hlavný algoritmus sa nachádza v triede Gestalist. Jej konštruktor prijíma ako parameter binarizovaný obrázok modelu a nastavenia metódy GestalistParameters. Ihneď v konštruktore sú pre obrázok modelu vytvorené kodóny pomocou triedy CodonFactory. Pre samotnú detekciu slúži, rovnako ako pri iných triedach dediacich z triedy Detector, metóda FindBestMatches(). Vstupným parametrom pre pre metódu je testovaný obrázok. Prvým krokom je výpočet moment mapy pomocou triedy Torque. Toto odpovedá kroku č.1 z algoritmu v prílohe A.2.

Po výpočte moment mapy nasleduje konštrukcia kodónov pre testovaný obrázok. Následne sú v moment mape s pomocou triedy ExtremaFinder vyhľadané extrémy, ktoré sú reprezentované pomocou štruktúry Extremum. Táto štruktúra obsahuje informácie o pozícii a hodnote extrému, veľkosť oblasti, ktorá je pre príslušný extrém zistená z mapy mierítok, ale aj vektor bodov contributing_edge_pixels, obsahujúci body, ktoré do celkovej hodnoty extrému prispievajú najväčšou hodnotou. Extrémy sú ukladané v dvoch vektoroch typu vector<Extremum>. Jeden vektor predstavuje minimá funkcie, druhý vektor obsahuje maximá. V algoritme z prílohy A.2 je tento krok označený č. 2. Extrémy, ktorých oblasti sa prekrývajú, sú zoskupené pomocou funkcie GroupNearbyPatches() a podľa najviac prispievajúcich bodov sú vybrané kodóny z testovaného obrázka. Na to, aby bol kodón vybraný, musí mať v oblasti dostatok podporujúcich bodov. Minimálny počet týchto bodov je definovaný parametrom codon_vote_threshold. Po vytvorení množiny testovacích kodónov sú vo funkcii ComputeWeights() spočítané podobnosti medzi kodónmi modelu a kodónmi testovaného obrázka. Výpočet váh začína výpočtom vzdialeností medzi kodónmi pomocou funkcie DistanceCodons() triedy ShapeContext. Vzhľadom na to, že tvarový kontext nie je rotačne invariantný (kapitola 4.4.3), je uhol rotácie medzi kodónom modelu a kodónom testovaného obrázka vyhľadaný pomocou OpenCV funkcie cv::phaseCorrelate() vo funkcii EstimateRotAngles() triedy ShapeContext. Použitím tohto uhla môžu byť testovacie kodóny rotované vďaka funkcii Rotate() v triede CodonFactory. Po výpočte minimálnych vzdialeností sú vzdialenosti prerátané na váhy pomocou rovnice (11). Návratovou hodnotou ComputeWeights() je teda matica obsahujúca váhy hranových bodov $W_{D^{\tau}_{sc}}$.

Moment mapa torque_ je prerátaná s použitím funkcie SetWeightsAndRecompute(), ktorej vstupným parametrom sú práve vyrátané váhy. Vyhľadaním extrémov v mape hodnôt dostávame

konečné polohy a merítka detekovaných objektov. Tie sú vhodným spôsobom pretransformované do výstupných parametrov funkcie `FindBestMatches()`.

6.2.5 Projekt OpenCVModules

Tento projekt slúži na vytvorenie vrstvy medzi voľne prístupnými implementáciami algoritmov FDCM (kapitola 5.1) a LINE-2D (kapitola 5.2) a nami implementovanou testovacou aplikáciu. Hlavný dôvod je to, že sme nami implementované algoritmy chceli porovnať s inými algoritmami.

Pre knižnicu OpenCV existuje GitHub repozitár s názvom `opencv_contrib`[8], v ktorom sú voľne prístupné rozširujúce moduly pre OpenCV. Po stiahnutí tohto repozitára potom môžeme dynamické `dll` OpenCV knižnice skompilovať pomocou nástroja CMake tak, že obsahujú tieto prídavné moduly. Vzhľadom na to, že v tejto práci sme z tohto repozitára potrebovali len LINE-2D detektor, pristúpili sme k tomu, že zdrojový kód `linemod.hpp` spolu s ostatnými potrebnými súbormi sme z `opencv_contrib` prekopírovali do tohto projektu. Výhodou je, že môžeme použiť ľubovolnú inštaláciu OpenCV knižníc a v prípade potreby dynamické knižnice OpenCV zameniť za novšie.

Zaujímavosťou je, že zdrojový kód pre FDCM býval súčasťou starších verzií OpenCV. V najnovšej verzii 3.1.0 sa žiadny algoritmus podobný ako FDCM nenachádza ani v základnej knižnici OpenCV, ani v rozširujúcich moduloch `opencv_contrib`. Preto sme do tohto projektu skopírovali aj zdrojový kód `chamfermatching.cpp` z OpenCV knižnice verzie 2.4.12[9].

Pre oba algoritmy sú v projekte vytvorené obalové triedy, ktoré implementujú abstraktnú triedu `Detector` a pre detekciu volajú funkcie, ktoré sú prekopírované z OpenCV. Konkrétne pre algoritmus FDCM je to trieda `FDCMWrapper`, pre LINE-2D je to trieda `Line2DWrapper`. Vstupným parametrom konštruktora oboch tried je obrázok modelu v stupňoch šedi a objekt typu `FDCMParameters` v respektíve `Line2DParameters` s nastaveniami príslušného algoritmu. Trieda `FDCMWrapper` veľa funkcionality naviac nepridáva. V podstate len predáva parametre zo štruktúry `FDCMParameters` do volania funkcie `chamferMatching()` a výsledok detekcie transformuje do formátu používanom v celej aplikácii. Podobne je to pri triede `Line2DWrapper`, tá ale pred detekciou mení rozmer testovaných obrázkov. Keďže autori tohto algoritmu optimalizovali výpočet pomocou SIMD inštrukcií, musia byť matice v pamäti zarovnané, inak detekcia skončí s chybou. V ich implementácii pracujú len s obrázkami vo VGA rozlíšení. Pre vyriešenie tohto problému táto obalová trieda mení rozmer testovaných obrázkov na najbližší väčší násobok VGA rozlíšenia.

6.2.6 Projekt Demo

Jedinou spustiteľnou aplikáciu je tento projekt. Správanie programu je ovplyvnené argumentami predanými do príkazovej riadky pri spustení aplikácie. Niektoré nastavenia programu sú

[8]Dostupné z: `https://github.com/Itseez/opencv_contrib`
[9]Dostupné z: `http://opencv.org/downloads.html`

realizované pomocou `xml` súborov. Projekt spúšťa algoritmy naprogramované v iných moduloch a zobrazuje ich výsledky buď do okna, alebo uložením obrázkov a nameraných hodnôt do vybraného adresára.

Pre získanie argumentov príkazovej riadky je použitá OpenCV trieda `cv::CommandLineParser`. Tá umožňuje pohodlné čítanie argmentov s tým, že prečítané argumenty sú ihneď pretypované na správny dátový typ. Všetky možné argumenty sú definované v globálnej premennej `keys`.

Pre zobrazenie výsledku detekcie je použitá trieda `Matching`. V konštruktore na základe mena porovnávacej metódy predanej parametrom `method_string` vytvorí inštanciu príslušného detektora. Toto uľahčuje trieda `DetectorFactory` popísaná v kapitole 6.2.1. Ukazovateľ na detektor je v triede uchovávaný v parametre `detector_`. Pre jeho uchovávanie je použitý typ `cv::Ptr`, čo je implementácia *smart pointerov*[10] v OpenCV. Po vytvorení objektu typu `Matching` je na inštancii možné zavolať funkciu `Match()`, ktorá spustí samotnú detekciu a výsledok zobrazí v okne.

Trieda `Evaluation` slúži na ohodnotenie detekcie vybranej porovnávacej metódy. Do konštruktora podobne ako trieda `Matching` prijíma cestu k `xml` súboru s nastaveniami porovnávacej metódy a na základe tohto súboru je vytvorený detektor. Druhým parametrom je cesta k `xml` súboru, ktorý obsahuje zoznam obrázkov, nad ktorými sa bude spúšťať detekčný algoritmus. Tento súbor obsahuje cestu k modelovému obrázku, k testovaným obrázkom a cestu k súborom, v ktorých sú súradnice správne detekovaných objektov. Detektor by pri ideálnej funkčnosti mal objekty detekovať práve na týchto pozíciach (ang. *ground truths*). Tretím argumentom je cesta k adresáru, kde budú uložené výsledky detekcie. Po zavolaní funkcie `Evaluate()` sa pomocou funkcie `FindBestMatches()` detekuje modelový obrázok vo všetkých testovaných obrázkoch. Obrázky, v ktorých sú znázornené pozície detekovaných oblastí, sú uložené do výstupného adresára. Zeleným obdĺžnikom sú označené správne detekcie, červeným nesprávne detekované objekty. To, či je objekt detekovaný správne, sa rozhoduje vo funkcii `IsTruePositive()` na základe Pascalovho kritéria (kapitola 7). Do výstupného adresára je uložený aj `csv` súbor, ktorý pre postupne menenú hodnotu parametra `detection_threshold` obsahuje hodnoty: celkový počet objektov, ktoré mali byť detekované, počet správne/nesprávne detekovaných objektov, FPPI, senzitivitu a presnosť. Význam týchto hodnôt a grafy vygenerované na základe nich sú v kapitole 7.

Niektoré obrázky v texte sú výstupom tohto ukážkového programu. Zobrazovanie týchto obrázkov a zabezpečuje trieda `Figure`. Do konštruktora tejto triedy je predávaný celý objekt typu `cv::CommandLineParser`, z ktorého sú potom prečítané všetky potrebné argumenty. Po zavolaní funkcie `Show()` sa na základe argumentu `fig` vyberie, ktorá ukážka sa má zobraziť. Každá ukážka má svoju vlastnú implementáciu a všetky pre svoju funkciu využívajú ostatné moduly.

[10] Smart pointer je ukazovateľ na objekt, ktorý automaticky rieši manažment dynamicky alokovanej pamäti. Zjednodušene povedané, keď pamäť už nie je potrebná, je automaticky uvoľnená. To je rozdiel v porovnaní s klasickými ukazovateľmi, kde musíme alokovanú pamäť uvolniť pomocou príkazu `delete`.

Posledná trieda, ktorú popíšeme má názov `Realtime`. Pre jej konštrukciu je potrebná cesta k `xml` súboru s nastaveniami porovnávacej metódy a obrázok modelu. Podobne ako v ostatných triedach je skonštruovaná inštancia detektora a je uložená do premennej `detector_`. Po zavolaní funkcie `Start()` ja vykonávanie aplikácie rozdelené do dvoch vlákien. V jednom vlákne prebieha zachytávanie obrazu z predvolenej web kamery počítača do premennej `frame_` a zobrazovanie zachyteného obrazu, spolu s obdĺžnikom, ktorý ohraničuje detekovaný objekt s najvyššou hodnotou funkcie podobnosti. Práve pre toto vyhľadávanie objektu je vykonávané v druhom vlákne. Pre vyhľadávanie je použitý detektor z premennej `detector_`. Priebeh oboch vlákien sa ukončí ak je stlačená ľubovolná klávesa.

6.3 Ukážkový program

Ukážková aplikácia má tri hlavné funkcie: prezentovať výsledok detekcie vybranej porovnávacej metódy s použitím ľubovoľného obrázku modelu a testovacieho obrázka, ohodnotenie vybranej porovnávacej metódy pre ľubovolný zoznam obrázkov, zobrazenie krátkych ukážok jednotlivých častí programu alebo vyhľadávanie modelu v obraze z webkamery v reálnom čase. V adresári `\program` sa nachádzajú `bat` súbory, ktoré spúšťajú ukážkovú aplikáciu s predvolenými parametrami. Spustiteľný program `demo.exe` sa nachádza na CD v adresári `\demo\bin\`. Pri jeho spustení bez parametrov je na štandardný výstup zoznam všetkých argumentov, s ktorým je možné program spustiť. V ďalšom texte je stručne popísaný význam jednotlivých parametrov.

- `-match settings.xml model.png test.png`

  Určite najdôležitejšou funkciou programu je vyskúšať implementované metódy na vlastných obrázkoch a vizuálne ohodnotiť, či je detekcia správna alebo nie. Prípadne skúmať, aký vplyv na výsledok majú jednotlivé parametre detektora. Zmena týchto parametrov je jednoduchá, keďže všetky parametre sú uložené v `xml` súboroch. Pomocou horeuvedeného príkazu sa vyhľadá modelový obrázok v testovacom obrázku s použitím porovnávacej metódy nastavenej v `xml` súbore a zobrazí sa okno so zvýrazneným výsledkom detekcie. Je vhodné, aby v `xml` súbore bol nastavený parameter `show_debug_info` na `true`, tým pádom budú zobrazené ďalšie doplňujúce okná s informáciami (hodnoty porovnávacej funkcie zakódované v pseudofarbe, deskriptor COF a pod.). Význam ostatných parametrov by z ich názvu mal byť po prečítaní teoretických kapitol pomerne jasný. V ostatnom prípade sú všetky parametre podrobne zdokumentované v Doxygen dokumentácii. Štruktúra tohto súboru s nastaveniami je na obrázku 19.

- `-eval settings.xml files.xml out`

  Druhou veľmi dôležitou funkciu je ohodnotiť výsledok detekcie nielen vizuálne na jednom obrázku, ale aj na vhodne zvolenom balíku obrázkov. Ak sú naviac dostupné skutočné pozície objektov v testovaných obrázkov, môžeme detekciu ohodnotiť aj matematicky, prípadne

```
<opencv_storage>
<method>cof</method>
<parameters>
<detection_threshold>0.1</detection_threshold>
<parallel>1</parallel>
<show_debug_info>0</show_debug_info></parameters>
</opencv_storage>
```

Obr. 19: Ukážka xml súboru s nastaveniami pre metódu COF

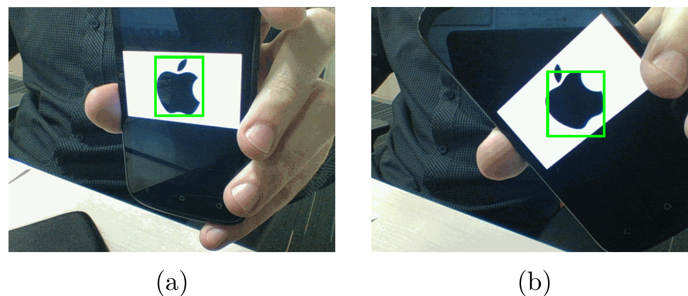
na základe výsledkov detekcie skonštruovať grafy. Horeuvedený príkaz pomocou nastavenej porovnávacej metódy detekuje modelový obrázok v testovacích obrázkoch definovaných v súbore `files.xml`. Detekcie porovná voči skutočným pozíciám takisto definovaným v súbore `files.xml`. Na základe týchto výsledkov do výstupného adresára `out` uloží očíslované obrázky, na ktorých sú zvýraznené detekované oblasti. Ďalej do adresára uloží súbor `result.csv`, v ktorom sa nachádzajú hodnoty: počet hľadaných objektov, počet skutočne pozitívnych, počet falošne pozitívnych, FPPI, senzitivita a presnosť. Na základe nich je možné vykresliť grafy z kapitoly 7.

- `-fig=<name> <param1> <param2> <param3>`

  Pri implementovaní algoritmov bolo vytvorených veľa malých funkcií zobrazujúcich priebežný stav matíc pri priebehu detekcie. Takisto boli vytvorené testovacie funkcie, ktoré overujú, či vybraná časť programu pracuje správne. Testovacie funkcie sú v kóde ponechané, a niektoré z nich sú volané pri použití parametra `-fig`. Zoznam parametrov, ktoré môžeme použiť namiesto `<name>`: `torque`, `gradient`, `fft`, `sc`, `ori`, `codon`, `thin`. Každá z ukážok má svoje vlastné parametre. Pre väčšinu z nich ale platí, že `<param1>` je cesta k vstupnému obrázku. Pre všetky tieto malé ukážkové programy sa na CD nachádzajú `bat` súbory. Výstupy programu po spustení s týmto parametrom sú použité aj v texte tejto práce ako ukážkové obrázky. V texte sa pri obrázkoch, ktoré sa dajú takto zobraziť (1, 4, 8, 10, 13), nachádza poznámka pod čiarou s potrebnými parametrami. Ukážka `ori` zobrazuje kumulované orientácie COF deskriptora a je aj interaktívna, kde po prejdení myšou po obrázku s váhami deskriptora sa v druhom okne znázornia orientácie pod aktuálnym bodom.

- `-realtime settings.xml model.png`

  Keďže metóda COF je pomerne rýchla, je možné ju využiť aj pre detekciu objektov v reálnom čase. Toto je umožnené pri spustení programu s uvedeným parametrom alebo po spustení súboru `realtime_cof.bat`. Program sníma obraz z predvolenej kamery a v snímanom obraze sa snaží detekovať modelový objekt predaný parametrom. Detekovaná oblasť je zvýraznená zeleným obdĺžnikom. V ukážkovom prípade je modelový obrázok vyhľadávaný v 16 rôznych orientáciách v merítku od 0.8 do 1.5. Pre prehľadávacia pyramída



(a) (b)

Obr. 20: Ukážka aplikácia pre detekciu v reálnom čase

má výšku 4. Pri takto nastavených parametroch a s webkamerou s VGA rozlíšením je rýchlosť detekcie okolo 0.10s, čo postačuje na detekciu v reálnom čase. Na obrázku 20 je ukážka programu.

- `-def_method=<method> out.xml`

  Súbory s nastaveniami sú špecifické pre každú metódu, týmto príkazom je možné vygenerovať `xml` súbor s predvolenými nastaveniami pre zvolenú metódu. Hodnota `<method>` môže byť nahradená jednou z hodnôt: (`cof`, `gestalist`, `line2d`, `fdcm`).

- `demo.exe -def_files files.xml`

  Pri evaluácii vybranej porovnávacej metódy sa používa `xml` zoznam súborov, na ktorých sa metóda bude testovať. Súbor s predvolenými hodnotami je možné vygenerovať spustením programu s týmto príkazom.

7 Vyhodnotenie a porovnanie

Cieľom tejto kapitoly je ohodnoť implementované algoritmy na základe bežne používaných metrík. Implementované detekčné metódy sú porovnané s dvomi voľne prístupnými algoritmami z knižnice OpenCV. Výsledky porovnania sú prehľadne zachytené pomocou grafov. V kapitole sú používané pojmy ako pascalovo kritérium, falošná/skutočná pozitivita, senzitivita, specificita alebo presnosť testu. Preto je úvod kapitoly venovaný vysvetleniu týchto pojmov.

7.1 Definícia pojmov

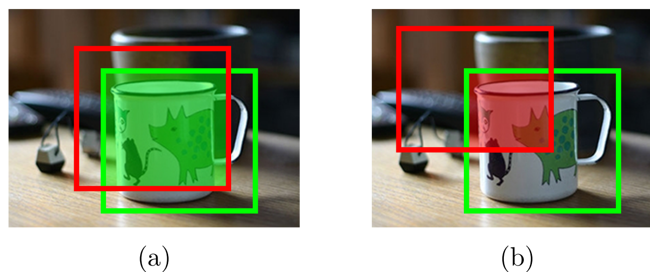
Hlavým výstupom detektora objektov sú označené pozície detekovaných objektov v testovanom obrázku. Samozrejme detektor označí niektoré pozície správne, ale môže označiť niektoré pozície aj nesprávne. Podľa pojmov používaných predovšetkým v zdravotníckej štatistike [41] môžeme takúto formu detektora označiť pojmom test. Detektor, ako špeciálny prípad testu môže nadobúdať hodnoty:

- **skutočne pozitívny** - detektor označil objekt, ktorý na sa obrázku skutočne nachádza
- **falošne pozitívny** - detektor označil nesprávny objekt
- **skutočne negatívny** - detektor neoznačil žiadny objekt a objekt sa v danom obrázku skutočne nenachádza
- **falošne negatívny** - detektor neoznačil žiadny objekt, pritom sa objekt danom v obrázku nachádza

Kolekcie obrázkov, s ktorými pri vyhodnocovaní algoritmov pracujeme, obsahujú doplnkové súbory, v ktorých je uvedený počet objektov a pozície, kde presne sa v obraze objekty nachádzajú. Anglicky sa tieto súbory nazývajú *ground truths*. Vďaka tomuto môžeme programovo rozhodnúť, či algoritmus objekt detekoval správne alebo nie. V programe používame formát súborov, v ktorých je zoznam štvoruholníkov, kde každý štvoruholník je určený dvomi bodmi. Každý štvoruholník teda ohraničuje jeden objekt. Ak by algoritmus pracoval ideálne, pri detekcii by sa pozícia a rozmer detekovaného štvoruholníka zhodoval so štvoruholníkom v súbore *ground truth*. V praxi to samozrejme takto ideálne nedopadne, preto je potrebné určiť kritérium, podľa ktorého sa určí, či je pozícia a rozmer detekovaného a ideálneho štvoruholníka dostatočne podobná, a teda výsledok testu je skutočne pozitívny. Tak ako veľa iných článkov, je aj v tejto práci použité tzv. pascalovo kritérium [13]. Aby bol objekt vyhodnotený ako skutočne pozitívny, musí byť pomer obsahu prieniku detekovanej ohraničujúcej oblasti objektu so skutočnou ohraničujúcou oblasťou a obsahu ich zjednotenia väčší ako 0.5. Pomocou vzorca môžeme toto tvrdenie zapísať ako

$$a_0 = \frac{\text{obsah}(B_p \cap B_{gt})}{\text{obsah}(B_p \cup B_{gt})},$$

kde B_p je detekovaná oblasť, B_{gt} je skutočná (*ground truth*) oblasť. Ak je teda hodnota $a_0 > 0.5$, je výsledok detekcie ohodnotený ako skutočne pozitívny. Na obrázku 21 sú zobrazené dva prípady



(a) (b)

Obr. 21: Vyhodnotenie detekovanej oblasti

vyhodnocovania pascalovho kritéria. Zelený štvoruholník ohraničuje oblasť skutočného objektu (*ground truth*), červený štvoruholník predstavuje výstup detektora objektov, polo-transparentná zelená a červená oblasť predstavuje prienik štvoruholníkov. Na obrázku 21a je výsledok detekcie vyhodnotený ako skutočne pozitívny, pretože pomer prieniku a zjednotenia štvoruholníkov je väčší ako 0.5. Naopak na obrázku 21b je tento pomer menší ako 0.5, preto je detekcia vyhodnotená ako falošne pozitívna.

Ak dokážeme určiť výsledok detekcie. Môžeme test prevedený na kolekcii obrázkov ohodnotiť. Prvým kritériom je presnosť testu, ktorá je definovaný ako pomer skutočne pozitívnych detekcií a všetkých pozitívnych detekcií. Pomocou vzorca zapísané takto

$$\text{presnosť} = \frac{\text{počet skutočne pozitívnych}}{\text{počet skutočne pozitívnych} + \text{počet falošne pozitívnych}}\ .$$

V praxi by to znamenalo, že ak by algoritmus detekoval 3 psy na scéne, kde sa nachádza 5 psov a 3 mačky, pričom 2 označené objekty sú v skutočnosti mačky, znamenalo by to, že detekčný algoritmus má presnosť $\frac{1}{4}$. Presnosť by sme teda mohli označiť ako metriku kvality detekčného algoritmu.

Druhá zaujímavá metrika je senzitivita testu (ang. tiež *recall*). Tá je definovaná ako pomer skutočne pozitívnych a všetkých objektov, ktoré sú v skutočnosti v scéne. Alebo ako pomer skutočne pozitívnych a súčet skutočne pozitívnych a falošne negatívnych. Vzorcom zapísané ako

$$\text{senzitivita} = \frac{\text{počet skutočne pozitívnych}}{\text{počet skutočne pozitívnych} + \text{počet falošne negatívnych}}\ .$$

V horeuvedenom príklade so psami a mačkami by pri uvedenej situácii by sa hodnota senzitivity rovnala $\frac{2}{5}$. Ak by sme mali detektor objektov, ktorý ako skutočne pozitívne detekcie označí všetky body obrázka a všetky možné veľkosti objektov, jeho senzitivita by bola 100%, pretože dokáže detekovať všetky objekty. Naopak jeho presnosť by sa blížila k nule. V zdravotníckej praxi sa senzitivita často používa na ohodnotenie prístrojov. Predstavme si prístroj pre odhalenie choroby, ktorý ma 100% senzitivitu. Na vyšetrenie by prišlo 10 ľudí a z nich by prístroj označil, že 3 ľudia trpia chorobou. Z troch ľudí ale niektorí chorobou trpieť nemusia. Ostatných 7 pacientov môže byť 100% istých, že chorobou netrpia. Pri detekcii senzitivita udáva schopnosť detektora odhaliť všetky objekty v scéne, pri tom nezáleží na tom, koľko objektov detekuje nesprávne. To môžeme

zistiť horeuvedenou presnosťou, alebo kritériom FPPI.

Kritérium FPPI je z akronym z anglického *false positives per image*, teda pomer falošne pozitívnych a celkového počtu obrázkov v testovanej kolekcii. Zapísané vzorcom je ako

$$\text{FPPI} = \frac{\text{počet falošne pozitívnych}}{\text{počet obrázkov v kolekcii}} \,.$$

Udáva teda koľko nesprávnych objektov priemerne označí algoritmus v jednom obrázku. V praxi sa často používa krivka, ktorá zobrazuje závislosť hodnôt senzitivity a FPPI na minimálnej hodnote prahu porovnávacej funkcie. Tento prah určuje, aká musí byť minimálna hodnota porovnávacej funkcie, aby sa detekcia objektu označila ako pozitívna. Je jasné, že pri nastavení prahu na nízku hodnotu bude mať algoritmus schopnosť odhaliť množstvo objektov, veľa z nich ale bude falošne pozitívnych a tým pádom bude mať detekcia vysokú hodnotu FPPI. Naopak pri nastavení prahu na vysokú hodnotu sa počet detekcií zníži, pritom sa zníži aj hodnota FPPI. Pri vysokom prahu by ale algoritmus mal dosahovať vysokú presnosť. Z tejto krivky je potom možné interpoláciu[11] získať hodnotu senzitivity pre zvolenú hodnotu FPPI.

7.2 Porovnanie

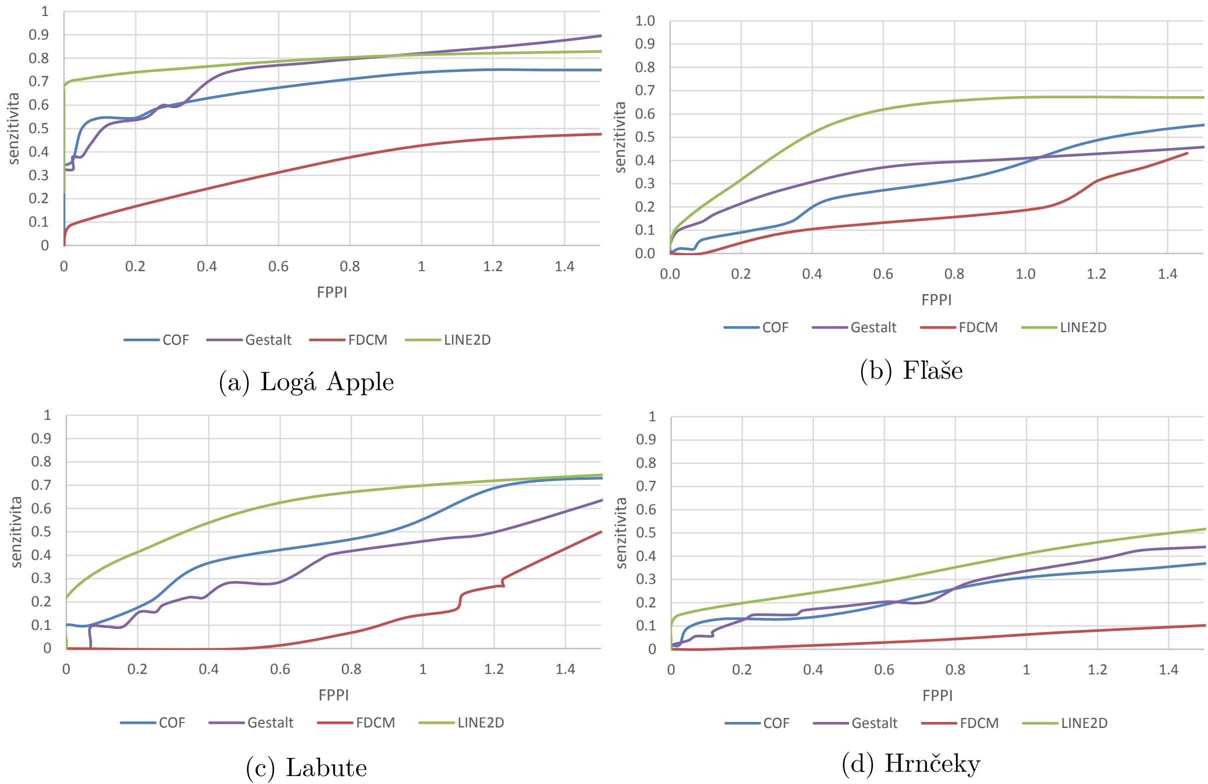
Naimplementované metódy sme porovnali na verejne prístupnej kolekcii obrázkov [14], ktorá je veľmi často používaná v článkoch zaoberajúcich sa detekciou objektov. Táto kolekcia pozostáva z 255 obrázkov a obrázky sú rozdelené do piatich rôznych tried: logo apple, fľaše, žirafy, hrnčeky a labute. V obrázkoch sa vždy nachádza jeden alebo viac z týchto objektov, na každom obrázku sú vždy len objekty z jednej triedy. Objekty sa na obrázkoch nachádzajú v rôznych veľkostiach a rôzne rotované, pričom obrázky sú zachytené pri rôznych zmenách osvetlenia. Tiež je rôzny štýl jednotlivých obrázkov. Prevažná väčšina obrázkov sú fotky, ale nachádzajú sa tu aj rukou kreslené obrázky, alebo tzv. *clip art* obrázky. Pre každú triedu obrázkov sa v kolekcii nachádza aj kontúra jednoduchého modelového obrázka, ktorá sa v vyhľadáva v testovaných obrázkoch. Výhodou tejto kolekcie je, že obsahuje aj súbory, v ktorých sú zaznačené ohraničujúce oblasti objektov, ktoré sa na obrázku skutočne nachádzajú (tzv. *ground truths*). S ich použitím môžeme ľahko vyhodnotiť, či je detekcia skutočne alebo falošne pozitívna.

7.2.1 Krivka závislosti FPPI a senzitivity na prahu detekcie

Toto porovnanie bolo prevedené pomocou ukážkového programu[12], ktorého výstupom sú obrázky s vyznačenými detekovanými oblasťami a `csv` súbory obsahujúce namerané dáta. Tieto boli následne skopírované do `xlsx` súborov, kde z nich boli vykreslené grafy 22 a interpolované hodnoty pre tabuľku 1. Pre zaistenie objektívnych porovnaní, sme spustili testovací program

[11] Pre interpoláciu sa v programe Excel nenachádza žiadna jednoduchá funkcia. Pre jej vypočítanie sme postupovali podľa návodu dostupného na: `http://peltiertech.com/excel-interpolation-formulas/`

[12] Program bol spustený s parametrom `-eval method.xml filelist.xml output_dir`. Parametre pre ostatné ukážky sú na strane 52



(a) Logá Apple

(b) Fľaše

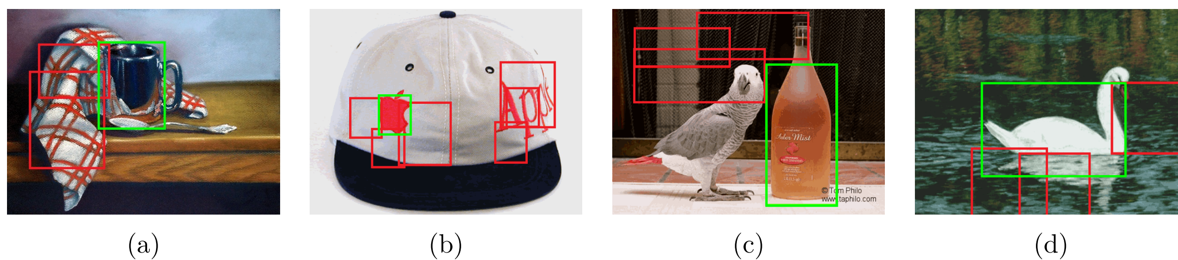
(c) Labute

(d) Hrnčeky

Obr. 22: Krivky zobrazujúce závislosť FPPI a senzitivity na prahu detekcie

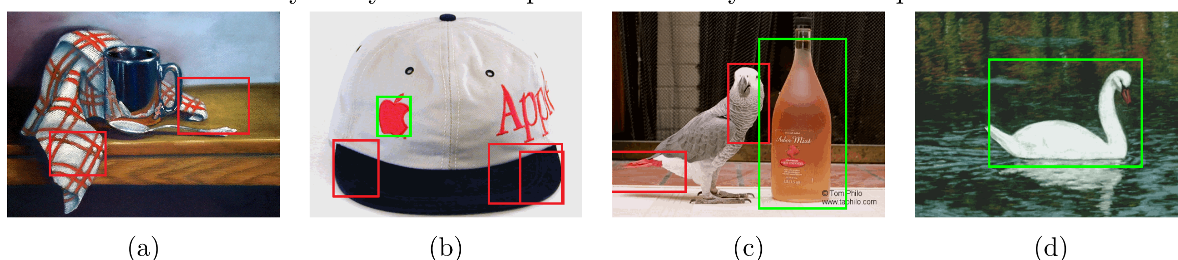
pre všetky štyri porovnávané metódy a na vybrané štyri triedy obrázkov s približne rovnakými nastaveniami. To znamená, že merítko vyhľadávaného modelu bolo v rozmedzí $(0.8, 2.0)$, rozdiel jednotlivými medzi merítkami bol 0.05, pre algoritmy, ktoré využívajú Cannyho detektor hrán boli nastavené prahy $t_1 = 60$, $t_2 = 180$. Ostatné nastavenia algoritmov boli ponechané na predvolené hodnoty. Súbory `bat` s prednastavenými parametrami spolu s výsledkami testov v `xlsx` súboroch sa nachádzajú na CD.

Na obrázku 22 sú znázornené výsledky prevedených testov na jednotlivých triedach objektov. Krivky z obrázka 22 boli vykreslené tak, že hodnota prahu detekcie bola postupne zvyšovaná od 0 do 1 s krokom 0.02, pre jednotlivé hodnoty prahu bol zaznamenaný počet skutočne/falošne pozitívnych detekcií a na základe celkového počtu obrázkov a objektov v obrázkoch boli vypočítané hodnoty senzitivity a FPPI. Vidíme, že pre triedu obrázkov s názvom “Logá Apple” (obrázok 22a) dosiahli všetky porovnávané algoritmy najlepšie výsledky. Nami implementované metódy COF a Gestalt dokonca v niektorých častiach grafu dosahujú lepšie výsledky ako implementácia LINE2D z knižnice OpenCV. Dôvodom týchto dobrých výsledkov je to, že na prevažnej väčšine testovaných obrázkov je objekt jasne oddelený od pozadia, jeho kontúry sú jasne viditeľné, pričom sú objekty zachytené len s veľmi malými zmenami uhlu pohľadu kamery. Na obrázkoch 22b a 22c sú zobrazené výsledné krivky pre triedy “Fľaše” a “Labute”. Výsledky pre obe triedy dopadli podobne, s tým rozdielom, že metóda Gestalt v značnom úseku krivky 22b dosahuje lepšie



(a) (b) (c) (d)

Obr. 23: Výsledky detekcie s použitím metódy Gestalist s prahom 0.2



(a) (b) (c) (d)

Obr. 24: Výsledky detekcie s použitím metódy COF s prahom 0.2

výsledky ako metóda COF. Veľmi zle dopadla pre všetky metódy detekcia na triede obrázkov s názvom "Hrnčeky". Pri vizuálnom ohodnotení výsledkov detekcie sme usúdili, že kontúra modelu pri tejto triede je veľmi odlišná od tvarov objektov hľadaných objektov. Navyše v tejto triede obrázkov sú objekty zachytené v rôznych zložitých scénach. Výsledky detekcie pre vybrané obrázky sú zobrazené na 23 a 24. Pri ich vytváraní boli nastavené rovnaké parametre ako pri vytváraní FPPI kriviek z obrázka 22. Je viditeľné, že metóda COF nebola schopná správne detekovať hrnček na obrázku 24a. Objekty z iných tried boli na obrázkoch detekované dobre s tým, že sú prítomné aj falošne pozitívne detekcie zobrazené červenými štvoruholníkmi. Ich počet je pomerne vysoký, je to dané tým, že bol zámerne nastavený minimálny prah detekcie na nízku hodnotu 0.2 aby bolo vidieť rozdielny prístup algoritmov pri detekcii. Metóda COF na obrázku 24d ale aj pri takto nízkom prahu detekovala len skutočne pozitívny objekt.

Celkovo ale nami implementované metódy dosahujú veľmi podobné výsledky ako kvalitná metóda LINE2D, v niektorých prípadoch sú výsledky dokonca lepšie. Pre niektoré prípady boli dosiahnuté veľmi slabé výsledky, čo môže byť spôsobené nesprávnym nastavením parametrov jednotlivých detektorov. Pre objektívnosť boli nastavené pre všetky objekty rovnaké parametre.

Tabuľka 1 prehľadne zobrazuje interpolované hodnoty senzitivity pre hodnoty FPPI 0.4 a 1.0 pre všetky testované algoritmy a všetky triedy obrázkov. Vidíme, že v prípade triedy obrázkov "Logá Apple", pre FPPI 1.0 dosahuje naša implementácia algoritmu Gestalt najlepší výsledok. V ostatných prípadoch dominuje implementácia LINE2D z OpenCV.

Tabuľka 1: Porovnanie interpolovaných hodnôt senzitivity pre FPPI 0.4 a 1.0

	FPPI	COF	Gestalt	FDCM	LINE2D
Apple logá	0.40	0.625	0.682	0.237	0.761
	1.00	0.736	0.822	0.425	0.812
Hrnčeky	0.40	0.143	0.171	0.017	0.241
	1.00	0.305	0.333	0.063	0.395
Labute	0.40	0.367	0.231	0.000	0.506
	1.00	0.560	0.459	0.144	0.683
Fľaše	0.40	0.188	0.302	0.103	0.497
	1.00	0.397	0.410	0.190	0.668

7.2.2 Vplyv optimalizačných štruktúr

V tejto podkapitole sú prezentované výsledky[13] dosiahnuté použitím optimalizačných štruktúr, ktoré boli implementované ako súčasť porovnávacích metód. Takisto vplyv toho, že sme výpočet paralelizovali pomocou knižnice TBB[14].

Tabuľka 2 zobrazuje počet operácií a dobu výpočtu pre kalkuláciu moment mapy z kapitoly 4.3 pre obrázok s rozmermi 300×200px a 11 merítok oblastí v rozmeroch $82 - 102$px. Vidíme, že pri použití optimalizovaného výpočtu pomocou integrálnych obrazov v kombinácii s paralelizáciou výpočet zaberie len 0.29s. Naopak pri výpočte bez obidvoch optimalizácií výpočet zaberie až 19.61s. Počet operácií je pri použití optimalizovaného výpočtu okolo 100-krát menší (počet operácií sme rozdelili na dve časti, pretože v integrálnom obrazu môžu byť niektoré výpočty vypočítane vopred). Na základe týchto pozorovaní môžeme výkon optimalizačnej štruktúry ohodnotiť za veľmi dobrý. Druhá stránka je použitie moment mapy pre detekciu objektov v reálnom čase. Čas 0.29s je pomerne nízky, ale musíme si uvedomiť, že sa jedná o obrázok v rozlíšení 300×200px a to, že vypočítanie moment mapy je len prvý krok algoritmu z prílohy A.2 a objekt sa potom porovnáva pomocou tvarového kontextu. Preto, aj napriek, tejto optimalizácii sa metóda popísaná v kapitole 4 nedá použiť pre detekciu v reálnom čase.

Tabuľka 2: Porovnanie rýchlosti a počtu operácií pri výpočte moment mapy.

Integrálny obraz	✓	✓		
TBB	✓		✓	
Čas[s]	0.29	0.79	7.17	19.61
Počet operácií pred	498560		0	
Počet operácií počas	617703		58869510	
Počet operácií spolu	1116263		58869510	

V druhom teste sme sa zamerali na zrýchlenie výpočtu a zredukovanie prehľadávaného priestoru, ktoré je dosiahnuté pri použití metódy *coarse to fine strategy* (kapitola 3.4) pre prehľa-

[13]Testy boli spúšťané na notebooku s procesorom Intel® Core™ i7-5600U CPU @2.60GHz s 7,89GB RAM.

[14]Použitie optimalizačných štruktúr a paralelizmu je možné ovplyvniť parametrami `optimized` a `parallel` v `xml` súbore s nastaveniami porovnávacej metódy.

dávanie dvojrozmerného priestoru. Pre testovanie sme vybrali vstupný obrázok s rozlíšením 1024×768px, počet vrstiev pyramídy bol 4, na každej vrstve sa prehľadávalo 5 extrémov a modelový obrázok bol vyhľadávaný v 65 rôznych rotáciách a merítkach. Výsledok testu je v tabuľke 3. Takisto ako v predchádzajúcom teste boli výsledky detekcie porovnané s výsledkami nameranými pri použití paralelizácie. V tabuľke 3 vidíme, že použitie paralelizácie zrýchlilo výpočet približne 2.7-krát, a zrýchlenie vďaka použitiu optimalizačnej štruktúry je skoro 1000-násobné. Navyše bola rýchlosť nami implementovanej metódy COF porovnaná s rýchlosťou implementácie LINE2D z knižnice OpenCV, ktorej výpočet trval 0.17 pre rovnaký obrázok a rovnaké parametre. Nami naimplementovaná metóda teda nedosahuje až takú vysokú rýchlosť, ako implementácia LINE2D. Algoritmus LINE2D je v knižnici OpenCV veľmi dobre optimalizovaný a prekonať túto rýchlosť ani nebolo v úmysle tejto práce. Rýchlosť, ktorú dosahuje implementovaný algoritmus je dostatočná a algoritmus je vhodný pre detekciu objektov v reálnom čase.

Tabuľka 3: Porovnanie rýchlosti a počtu operácií pri výpočte moment mapy.

Coarse to fine	✓	✓		
TBB	✓		✓	
Čas[s]	0.23	0.63	211.0	N/A
Počet operácií	210408		51118080	

8 Záver

Prvým cieľom tejto práce bolo detailne popísať aktuálne možnosti pre porovnávanie vzdialeností medzi obrazmi. Pri spracovávaní tejto témy sme sa stretli s veľkým množstvom zaujímavých článkov z rôznych konferencií, ktoré prezentujú algoritmy využívajúce vzdialenosť medzi obrazmi ako hlavné kritérium pri detekcii a rozpoznávaní objektov v obraze. Toto množstvo algoritmov je možné rozdeliť do rôznych kategórií a my sme si pre dôkladnejší rozbor vybrali algoritmy, ktoré umožňujú detekciu netextúrovaných objektov. V tejto kategórii nás ako prvý zaujal algoritmus COF, ktorý pre svoju funkciu využíva orientácie gradientov a pracuje veľmi podobne ako algoritmus LINE2D. Jeho detailnému popisu je venovaná tretia kapitola tejto práce. Druhý algoritmus, ktorému sme sa rozhodli podrobnejšie venovať, pre svoju funkciu využíva nedávno prezentovaný moment operátor v kombinácii s deskriptorom lokálnych podobností s názvom tvarový kontext. Potrebné matematické základy pre pochopenie funkcie tohto algoritmu sú zhrnuté vo štvrtej kapitole.

V praktickej časti boli tieto dva algoritmy naprogramované v jazyku C++ s využitím knižnice OpenCV. Pri implementácii sme narazili na viacero problémov a často jediným riešením bolo vytvoriť krátku funkciu, ktorej výstupom je obrázok zobrazujúci aktuálny priebeh detekcie. Tieto obrázky sú priložené v texte práce a pomocou nich je čitateľovi zrozumiteľnejšie vysvetlený priebeh algoritmov. Hlavným účelom vytvorenej aplikácie je ale samotná detekcia objektov s použitím popísaných algoritmov. Jej výstupy sú ukladané do súborov, na základe ktorých je možné vytvoriť grafy pre porovnanie jednotlivých metód.

Práve dôkladnému porovnaniu je venovaná posledná kapitola tejto práce v ktorej sa nachádzajú vhodne zobrazené a okomentované výstupy ukážkovej aplikácie. Na základe kritérií popísaných v rovnakej kapitole sú implementované metódy porovnané s voľne prístupnými implementáciami algoritmov LINE2D a FDCM z knižnice OpenCV. Pri porovnaní sme zistili, že nami implementované metódy dosahujú podobné, v niektorých prípadoch ešte lepšie, výsledky detekcie. V tejto kapitole sme tiež potvrdili, že metóda COF je vhodná pre aplikácie, v ktorých detekcia objektov prebieha v reálnom čase. V budúcnosti by sa ďalšie zrýchlenie tejto metódy dalo dosiahnuť využitím linearizácie pamäte podobne ako v algoritme LINE2D. U druhej implementovanej metódy sme potvrdili, že aj napriek optimalizácii nie je vhodná pre aplikácie citlivé na dĺžku výpočtu. Detekcia s využitím moment operátora ale dosahuje zaujímave výsledky a v ďalšej práci by bolo možné výsledky vylepšiť nahradením tvarového kontextu za iný deskriptor.

Hlavným prínosom tejto práce je poskytnutie veľmi detailného popisu jednotlivých algoritmov spolu s vhodnými ukážkami a voľne prístupnou implementáciu. Aplikácia je naprogramovaná tak aby ju budúci používateľ mohol ľahko rozšíriť, alebo naopak využiť jej knižnice pre svoje vlastné projekty.

Literatúra

[1] BARROW, Harry G., et al. *Parametric correspondence and chamfer matching: Two new techniques for image matching.* SRI INTERNATIONAL MENLO PARK CA ARTIFICIAL INTELLIGENCE CENTER, 1977.

[2] BAY, Herbert; TUYTELAARS, Tinne; VAN GOOL, Luc. SURF: Speeded up robust features. In: *Computer vision–ECCV 2006. Springer Berlin Heidelberg*, 2006. p. 404-417.

[3] BELONGIE, Serge; MALIK, Jitendra; PUZICHA, Jan. Shape matching and object recognition using shape contexts. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 2002, 24.4: 509-522.

[4] BORGEFORS, Gunilla. Hierarchical chamfer matching: A parametric edge matching algorithm. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 1988, 10.6: 849-865.

[5] CALONDER, Michael, et al. Brief: Binary robust independent elementary features. *Computer Vision–ECCV 2010*, 2010, 778-792.

[6] CANNY, John. A computational approach to edge detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 1986, 6: 679-698.

[7] CAO, Jerry. *Gestalt Principles for Designers – Applying Visual Psychology to Modern Day Design* [online]. 2015 [cit. 2016-04-24]. Dostupné z: `http://blog.teamtreehouse.com/gestalt-principles-designers-applying-visual-psychology-modern-day-design`

[8] DALAL, Navneet; TRIGGS, Bill. Histograms of oriented gradients for human detection. *In: Computer Vision and Pattern Recognition, 2005. CVPR 2005. IEEE Computer Society Conference on. IEEE*, 2005. p. 886-893.

[9] DAMEN, Dima, et al. Real-time Learning and Detection of 3D Texture-less Objects: A Scalable Approach. In: *BMVC.* 2012. p. 1-12.

[10] DE OLIVEIRA, Ícaro Oliveira; ONO, Keiko Veronica; TODT, Eduardo. IGFTT: towards an efficient alternative to SIFT and SURF. *WSCG International Conferences in Central Europe on Computer Graphics*, 2015

[11] *Doxygen* [online]. [cit. 2016-04-24]. Dostupné z: `http://www.stack.nl/~dimitri`

[12] DRISCOLL, Meghan K., et al. Cell shape dynamics: from waves to migration. *PLoS Comput Biol*, 2012, 8.3: e1002392.

[13] EVERINGHAM, Mark, et al. The pascal visual object classes (voc) challenge. *International journal of computer vision*, 2010, 88.2: 303-338.

[14] FERRARI V., et al. Object detection by contour segment networks. *In Proc. European Conf. on Comp. Vis., volume 3953 of LNCS*, p 14–28., 2006.

[15] FISHER, R., S. PERKINS a A. WALKER. *Morphology* [online]. 2003 [cit. 2016-04-24]. Dostupné z: `http://homepages.inf.ed.ac.uk/rbf/HIPR2/morops.htm`

[16] FISCHLER, Martin A.; BOLLES, Robert C. Random sample consensus: a paradigm for model fitting with applications to image analysis and automated cartography. *Communications of the ACM*, 1981, 24.6: 381-395.

[17] GAVRILA, Dariu M. Multi-feature hierarchical template matching using distance transforms. In: *Pattern Recognition, 1998. Proceedings. Fourteenth International Conference on. IEEE*, 1998. p. 439-444.

[18] *Google C++ Style Guide* [online]. [cit. 2016-04-24]. Dostupné z: `https://google.github.io/styleguide/cppguide.html`

[19] HINTERSTOISSER, Stefan, et al. Dominant orientation templates for real-time detection of texture-less objects. In:*IEEE Computer Society Conference on Computer Vision and Pattern Recognition*, 2010.

[20] HINTERSTOISSER, Stefan, et al. Gradient response maps for real-time detection of textureless objects. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, 2012, 34.5: 876-888.

[21] KOFFKA, K. *Principles of gestalt psychology.* Repr. London [u.a.]: Routledge, 2001. ISBN 0415209625.

[22] KONISHI, Yoshinori, Yoshihisa IJIRI, Masaki SUWA a Masato KAWADE. Textureless object detection using cumulative orientation feature. In:*2015 IEEE International Conference on Image Processing (ICIP). IEEE*, 2015, s. 1310-1313

[23] LIU, Ming-Yu, et al. Fast directional chamfer matching. In: *Computer Vision and Pattern Recognition (CVPR), 2010 IEEE Conference on. IEEE*, 2010. p. 1696-1703.

[24] LOWE, David G. Distinctive image features from scale-invariant keypoints. *International journal of computer vision*, 2004, 60.2: 91-110.

[25] MAN, Duhu, et al. Implementations of parallel computation of Euclidean distance map in multicore processors and GPUs. In: *Networking and Computing (ICNC), 2010 First International Conference on. IEEE*, 2010. p. 120-127.

[26] NISHIGAKI, Morimichi; FERMÜLLER, Cornelia. The Image Torque Operator for Contour Processing. *arXiv preprint* arXiv:1601.04669, 2016.

[27] NISHIGAKI, Morimichi; FERMÜLLER, Cornelia; DEMENTHON, Daniel. The image torque operator: A new tool for mid-level vision. In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on. IEEE*, 2012. p. 502-509.

[28] *OpenCV* [online]. [cit. 2016-04-24]. Dostupné z: `http://opencv.org/`

[29] *ReSharper C++* [online]. [cit. 2016-04-24]. Dostupné z: `https://www.jetbrains.com/resharper-cpp/`

[30] RICHARDS, Whitman; HOFFMAN, Donald D. Codon constraints on closed 2D shapes. *Computer Vision, Graphics, and Image Processing*, 1985, 31.3: 265-281.

[31] ROSTEN, Edward; DRUMMOND, Tom. Machine learning for high-speed corner detection. In: *Computer Vision–ECCV 2006. Springer Berlin Heidelberg*, 2006. p. 430-443.

[32] RUBLEE, Ethan, et al. ORB: an efficient alternative to SIFT or SURF. In: *Computer Vision (ICCV), 2011 IEEE International Conference on. IEEE*, 2011. p. 2564-2571.

[33] SHIMPENO, Peter a Neta EZER. Improving the User Interface throughGestalt Design Principles. *The 26th Annual IEEE Software Technology Conference*. 2014.

[34] SOBEL, Irwin, et al. *Sobel-Feldman Operator*. 2014

[35] STEGER, Carsten. Occlusion, clutter, and illumination invariant object recognition. *International Archives of Photogrammetry Remote Sensing and Spatial Information Sciences*, 2002, 34.3/A: 345-350.

[36] TEO, Ching L., Cornelia FERMÜLLER a Yiannis ALOIMONOS. A Gestaltist approach to contour-based object recognition: Combining bottom-up and top-down cues. *International Journal of Robotics Research*. 2015, (34), 627-652. ISSN 0278-3649.

[37] *Threading Building Blocks*[online]. [cit. 2016-04-24]. Dostupné z: `https://www.threadingbuildingblocks.org/`

[38] TCHESLAVSKI, Gleb V. *Morphological Image Processing: Basic Algorithms*, 2009

[39] TOMBARI, Federico; FRANCHI, Alessandro; STEFANO, Luigi. BOLD features to detect texture-less objects. In: *Proceedings of the IEEE International Conference on Computer Vision*. 2013. p. 1265-1272.

[40] VON GIOI, Rafael Grompone, et al. LSD: A fast line segment detector with a false detection control. *IEEE Transactions on Pattern Analysis & Machine Intelligence*, 2008, 4: 722-732.

[41] *Zdravotnická statistika* [online]. [cit. 2016-04-24]. Dostupné z: `http://www.wikiskripta.eu/index.php/Kategorie:Zdravotnick%C3%A1_statistika`

A Príloha: algoritmy

A.1 Detekcia pomocou COF

Vstup: Obrázok modelu $\mathcal{I}_M$, Testovací obrázok $\mathcal{I}$;
Výstup: Množina P obsahujúca extrémy funkcie $s(x, y, i)$ - detekované objekty;

Krok 1: *Výpočet deskriptorov*;
Vygenerovať m transformovaných obrázkov pre vstupný obrázok $\mathcal{I}_M$;
for $i \leftarrow 1$ ***to*** m **do**
  Vygenerovať n náhodných tranformácií $\mathcal{I}_M$ do obrázka I_i;
  Inicializovať deskriptor $\mathcal{M}_i$;
  for $j \leftarrow 1$ ***to*** n **do**
    Extrahovať orientácie pre obrázok I_{ij};
    Kumulovať orientácie do deskriptora $\mathcal{M}_i$;
  end
  Vypočítať pyramídu o veľkosti k pre deskriptor $\mathcal{M}_i$;
end

Krok 2: *Detekcia objektov*;
Kvantizovať hrany testovaného obrázka $\mathcal{I}$;
Vypočítať pyramídu o veľkosti k pre testovaný obrázok $\mathcal{I}_T$;
$P \leftarrow (x, y, i)$ Vložiť do množiny P všetky body z k-tej vrstvy pyramídy testovaného obrázka a všetky indexy deskriptorov z $\mathcal{M}$;
for $j \leftarrow k$ ***to*** 1 **do**
  foreach $p \in P$ **do**
    Vypočítať hodnotu porovnávacej funkcie $s(p)$;
  end
  Vyhľadať extrémy z vypočítaných hodnôt funkcie $s(p)$;
  $P \leftarrow (x, y, i)$ Vložiť do množiny P všetky body a indexy v ktorých nastal extrém prepočítané pre ďalšiu vrstvu pyramídy.
end
Na základe obsahu množiny P zobraziť detekované objekty;

A.2 Detekcia pomocou torque mapy

Vstup: Obrázok hrán vstupného obrázka I_e, kodóny modelu $\mathcal{C}_{mo} = \mathcal{M}_1, ..., \mathcal{M}_l$, vypočítané shape context deksriptory pre všetky kodóny modelu: $(m_i^{\angle}, m_i^{sc}), i \in \mathcal{M}_l$ pre všetky body i z $\mathcal{M}_l$;
Výstup: Modulovaná torque mapa, T_I^m;

Krok 1: *Výpočet základnej torque mapy* T_I;
for $(r, c) \leftarrow (1, 1)$ ***to*** (H, W) **do**
  Vypočítať rovnicu (4) pre každú oblasť $P_s(r, c)$ so stredom v (r, c) pre obrázok I_e, pre každé merítko $s \in \mathcal{S}$;
  $T_I(r, c) \leftarrow \max_{s \in \mathcal{S}} \tau_{P_s}(r, c)$;
end
Vyhľadať top $|\mathcal{P}_c|$ stredov potencionálnych $p_c \in \mathcal{P}_c$ objektov v mape T_I;

Krok 2: *Výpočet modulovanej torque mapy* T_I^m;
for $p_c \in \mathcal{P}_c$ **do**
  Vybrať hrany Q_{p_c}, ktoré prispievajú k hodnote extrému hodnotou $> t_c$;
  Zoskupiť susediace hrany z Q_{r_c} pre vytvorenie väčšej množiny $\mathcal{C}_g = \{\mathcal{R}'_1 ..., \mathcal{R}'_d\}$;
  Vypočítať shape context deksriptory pre všetky kodóny testovacieho obrázka: $(g_i^{\angle}, g_i^{sc}), i \in \mathcal{R}'_d$ pre všetky body i z $\mathcal{R}'_d$;
  Vypočítať množinu O_g pomocou kroskorelácie medzi uhlovými zložkami v shape context deskriptorov v strede p_c.;
  $\mathcal{O} \leftarrow \{O_g, O_g + 90, O_g + 180, O_g + 270\}$;
  for $o_g \in \mathcal{O}_g$ **do**
    for $\mathcal{R}'_e \in \mathcal{C}_g$ **do**
      Zoskupiť susediacich $\mathcal{J}$ testovacích kodónov: $\mathcal{R}'_{\{e\},...,\{e+\mathcal{J}\}}$;
      Rotovať všetky testovacie kodóny o uhol o_g;
      for $(a, b) \leftarrow (1, 1)$ ***to*** $(\mathcal{J}, l)$ **do**
        Vypočítať $V(a, b) = D_{sc}^T(\mathcal{R}'_{\{e\},...,\{e+a\}}, \mathcal{M}_b)$ podľa rovnice 11;
      end
      $E_{D_{sc}^T}(e, o_g) \leftarrow min_{\mathcal{J},l} V$;
    end
    $E_{D_{sc}^T}(o_g) \leftarrow min_e E_{D_{sc}^T}(e, o_g)$;
  end
  $E_{D_{sc}^T} \leftarrow min_{o_g} E_{D_{sc}^T}(o_g)$;
end

Pokračovanie na ďalšej strane →

for $e \leftarrow 1$ ***to*** d **do**

  for $r_i \in \mathcal{R}'_e$ **do**

    Prepočítať vzdialenosti $E_{D_{sc}}$ na váhy $W_{D_{sc}}(r_i)$ podľa rovnice (12);

    Vypočítať vážený moment $\tau^{\omega}_{p_c r_i}$ podľa rovnice (13);

  end

end

for $(r, c) \leftarrow (1, 1)$ ***to*** (H, W) **do**

  for $r_i \in \mathcal{R}'_e$ **do**

    Vypočítať $\tau^{\omega}_{P_s(r,c)}$ podľa rovnice (14) pre každú oblasť $P_s(r, c)$ so stredom v (r, c) pre obrázok I_e, pre každé merítko $s \in \mathcal{S}$;

    $T^m_I(r, c) \leftarrow \max_{s \in \mathcal{S}} \tau^{\omega}_{P_s}(r, c)$;

  end

end