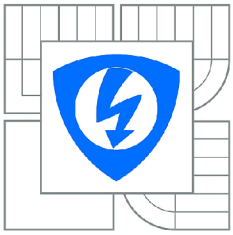
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ
ÚSTAV AUTOMATIZACE A MĚŘICÍ TECHNIKY

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION
DEPARTMENT OF CONTROL AND INSTRUMENTATION

PROPOJENÍ KNIHOVNY PRO ZPRACOVÁNÍ OBRAZU S JAZYKEM LUA

IMAGE PROCESSING LIBRARY WRAPPER FOR LUA

DIPLOMOVÁ PRÁCE
MASTER'S THESIS

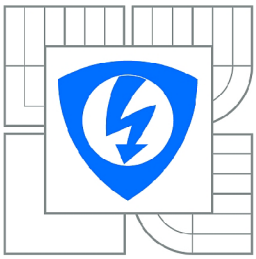
AUTOR PRÁCE
AUTHOR

Bc. JIŘÍ PRYMUS

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. PETR PETYOVSKÝ

BRNO 2012



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav automatizace a měřicí techniky

Diplomová práce

magisterský navazující studijní obor
Kybernetika, automatizace a měření

Student:	Bc. Jiří Prymus	***ID:***	109712
Ročník:	2	***Akademický rok:***	2011/2012

NÁZEV TÉMATU:

Propojení knihovny pro zpracování obrazu s jazykem Lua

POKYNY PRO VYPRACOVÁNÍ:

Cílem práce je navrhnout a realizovat vhodné přemostění mezi skriptovacím jazykem Lua a knihovnou algoritmů pro zpracování obrazu OpenCV.

1. Prostudujte problematiku zpracováním obrazu v rozsahu kurzu Počítačové vidění.
2. Nastudujte možnosti knihovny pro zpracování obrazu OpenCV využívané ve výuce kurzu.
3. Realizujte a průběžně aktualizujte přemostění komponent knihovny OpenCV s interpretrem skriptovacího jazyka Lua.
4. Vytvořené přemostění využijte pro realizaci úloh pro studenty, kurzů Počítačového vidění.
5. Začleňte realizované přemostění do výuky kurzů ve formě alespoň tří úloh pro studenty.
6. Průběžně vyhodnocujte připomínky a návrhy k jednotlivým úlohám z výuky kurzů (ZS2011/12).
7. Implementujte připomínky a návrhy do konečné verze výukových úloh a do realizovaného přemostění. Publikujte vytvořené nástroje GNU komunitě.
8. Zhodnoťte dosažené výsledky, uveďte výhody a nevýhody jednotlivých řešení a navrhněte další možná rozšíření.

DOPORUČENÁ LITERATURA:

[1] Šonka, M.; Hlaváč, V.: Počítačové vidění, Grada, Praha 1992, ISBN 80-85424-67-3
[2] Horák, K. a kol.: Elektronické texty ke kurzu Počítačové vidění MPOV, VUT 2008
[3] Bradski, G.; Kaehler, A.: Learning OpenCV, O'Reilly, 2008, ISBN 978-0-596-51613-0
[4] Ierusalimschy, R.: Programming in Lua, 2 edition; Lua.org, 2006, ISBN 978-8590379829

Termín zadání:	6.2.2012	***Termín odevzdání:***	21.5.2012

Vedoucí práce: Ing. Petr Petyovský
Konzultanti diplomové práce:

doc. Ing. Václav Jirsík, CSc.
Předseda oborové rady

Abstrakt
Předmětem této diplomové práce je seznámení se s knihovnou OpenCV a s jejím přemostěním do skriptovacího jazyka Lua. Prvá část práce popisuje kurz počítačového vidění MPOV a základní matematické aparáty používané v tomto kurzu. Dále následuje popis knihovny OpenCV a její využití ve výše zmíněném kurzu. Třetí část se věnuje stručnému popisu programovacího skriptovacího jazyka Lua.

Praktická část se zabývá přemostěním knihovny OpenCV do jazyka Lua pomocí Lua C API, vývoji podpůrných programů pro snazší kompilaci a distribuci binárních souborů. Program CMake byl použit jako multiplatformní generátor kompilačních projektů nutných pro různá vývojová prostředí a framework NSIS pro tvorbu instalátoru pro platformu MS Windows. Součástí práce je také generátor dokumentace implementovaný v Lua.

V poslední části práce se věnuji testování knihovny LuaCV v praxi a následné analýze kritických připomínek ze strany studentů.

Summary
The thesis deals with OpenCV library and its implementation into scripting language Lua. The first part of the thesis concentrates on description of the course Computer vision MPOV and description of mathematical basics needed for further understandings. The second part describes OpenCV library and its potential usage in the MPOV. Next chapter examines the programming scripting language Lua.

The description of the implementation of binding the OpenCV library to Lua language along with its overall functionality is included in the practical part of the thesis. The use of LuaCV is more comfortable thanks to Open Source projects for cross-platform compilation and distribution. Part of the thesis is also generator of Latex documentation for LuaCV binding.

The last chapter deals with testing LuaCV in course MPOV and analysis of criticism from students.

Klíčová slova
zpracování obrazu, OpenCV knihovna, Lua, C++, přemostění C++ knihovny do jazyka Lua, CMake instalátor, NSIS instalátor, generování dokumentace, svobodný software, SourceForge, Lua C API, kurz MPOV, GNUPlot

Keywords
image processing, OpenCV library, Lua, C++, C++ library wrapper to Lua, CMake, NSIS installer, generation of documentation, Open Source, SourceForge, Lua C API, course MPOV, GNUPlot

PRYMUS, J. *Propojení knihovny pro zpracování obrazu s jazykem Lua.* Brno: Vysoké učení technické v Brně, FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ, 2012. 84 s. Vedoucí diplomové práce Ing. Petr Petyovský.

Prohlášení

Prohlašuji, že svou diplomovou práci *Propojení knihovny pro zpracování obrazu s jazykem Lua* jsem vypracoval samostatně a s použitím odborné literatury a dalších informačních zdrojů, které jsou všechny citovány v práci a uvedeny v seznamu literatury na konci práce.

Jako autor uvedené diplomové práce dále prohlašuji, že v souvislosti s vytvořením této diplomové práce jsem neporušil autorská práva třetích osob, zejména jsem nezasáhl nedovoleným způsobem do cizích autorských práv osobnostních a jsem si plně vědom následků porušení ustanovení § 11 a následujících autorského zákona č. 121/2000 Sb., včetně možných trestněprávních důsledků vyplývajících z ustanovení § 152 trestního zákona č. 140/1961 Sb.

V Brně dne ....
(podpis autora)

Poděkování

Na tomto místě bych velmi rád poděkoval Ing. Petyovskému, který je vedoucím této práce. Díky svým zkušenostem a znalostem v dané problematice velmi přispěl svými připomínkami ke koncepčnosti textu a návrhu praktické části práce.

Bc. JIŘÍ PRYMUS

Obsah

1 Úvod **7**

2 Teoretický úvod **8**
2.1 Kurz MPOV . . . 8
2.1.1 Základní matematické vztahy . . . 8
2.1.2 Současná realizace úloh . . . 16
2.2 Knihovna OpenCV . . . 21
2.2.1 Rozdíly mezi předchozími verzemi . . . 22
2.2.2 Podporované platformy a software . . . 24
2.2.3 Moduly . . . 26
2.2.4 Využití OpenCV v kurzu MPOV . . . 36
2.3 Jazyk Lua . . . 41
2.3.1 Rozdíly mezi verzemi 5.1 a 5.2 . . . 41
2.3.2 Základy syntaxe . . . 42
2.3.3 Meta tabulky . . . 44
2.3.4 Garbage collector . . . 44

3 Realizovaná řešení **45**
3.1 Knihovna LuaCV . . . 45
3.1.1 Implementace . . . 46
3.1.2 Ukázkový skript . . . 59
3.1.3 LuaCV a GNUPlot . . . 60
3.1.4 Kompilace zdrojových souborů . . . 62
3.2 Instalátor . . . 63
3.2.1 Populární instalátory . . . 63
3.2.2 NSIS instalátor . . . 63
3.2.3 Základní instrukce . . . 64
3.2.4 Instalační skript . . . 66
3.3 Dokumentace . . . 71
3.4 LuaCV v kurzu MPOV . . . 72
3.4.1 Nepovinná cvičení . . . 72
3.4.2 Zápočtový projekt . . . 73
3.4.3 Převod cvičení MPOV do LuaCV . . . 73
3.4.4 Vypracování . . . 74

4 Zhodnocení dosažených výsledků **78**
4.1 Zhodnocení knihovny LuaCV . . . 78
4.2 Porovnání výkonnosti OpenCV a jeho přemostění . . . 79
4.3 Shrnutí použití LuaCV v kurzu MPOV . . . 80

5 Závěr **81**

Seznam příkladů

2.1 Prahování obrazu . . . 16
2.2 Adaptivní prahování . . . 17
2.3 Diskrétní konvoluce matic . . . 18
2.4 Gradientní operátory . . . 19
2.5 Dilatace . . . 20
2.6 Morfologické operace . . . 20
2.7 Kreslící funkce knihovny Core . . . 26
2.8 Maticové počty . . . 27
2.9 Detekce hran Laplaceovým operátorem . . . 28
2.10 Detekce kůže pomocí HSV histogramu . . . 29
2.11 Detekce rohů šachovnice . . . 31
2.12 Detekce významných bodů . . . 32
2.13 Detekce lidí pomocí Haar wavelets a HOG . . . 33
2.14 KNN a SVM klasifikátor . . . 35
2.15 Prahování obrazu . . . 36
2.16 Adaptivní prahování obrazu . . . 37
2.17 Diskrétní konvoluce matic . . . 38
2.18 Gradientní operátory . . . 39
2.19 Morfologické operace . . . 40
2.20 Syntaktická definice Lua[15] . . . 42
2.21 Využití syntaxe Lua . . . 43
3.1 Hlavní funkce modulu LuaCV . . . 47
3.2 Hlavní funkce modulu Imgproc . . . 47
3.3 Kontejner LuaCV objektu . . . 48
3.4 Kontrola typu objektu . . . 48
3.5 Zaslání objektů na zásobník . . . 49
3.6 Funkce na získání šířky textu . . . 49
3.7 Alternativní porovnání typů . . . 50
3.8 Alternativní detekce typů . . . 50
3.9 Využití alternativního porovnávání . . . 50
3.10 Indexovací funkce . . . 51
3.11 Meta tabulka . . . 51
3.12 Vyhledávání v index funkcích . . . 52
3.13 Registrace vlastnosti . . . 52
3.14 Header CvPoint . . . 53
3.15 Objekt CvPoint . . . 53
3.16 Třída cv::Mat . . . 54
3.17 Tvorba uživatelské matice . . . 55
3.18 Konverzní funkce . . . 56
3.19 Univerzální callback funkce . . . 57
3.20 Trackbar . . . 57
3.21 Chytání výjimek . . . 58
3.22 Zpracování výjimek . . . 58
3.23 Houghova transformace . . . 59
3.24 LuaCV a grafy v GNUPlot . . . 61

3.25 Vyhledávání knihoven 62
3.26 Nastavení překladu 62
3.27 Nastavení základních parametrů 66
3.28 Použití grafického rozhraní 68
3.29 Callback funkce 68
3.30 Kontrola instalačního adresáře 69
3.31 Stahování souborů 69
3.32 Instalační sekce 70
3.33 Po-instalační sekce 70

Seznam obrázků

2.1 Zdrojový obraz . . . 9
2.2 Ekvalizovaný obraz . . . 9
2.3 Zdrojový histogram . . . 9
2.4 Ekvalizovaný histogram . . . 9
2.5 Kumulativní histogram . . . 10
2.6 Zdrojový obrázek . . . 11
2.7 Gaussovské průměrování . . . 11
2.8 Sobelův operátor . . . 12
2.9 Laplaceův operátor . . . 12
2.10 Prahování s optimalním prahem . . . 13
2.11 Prahování s vysokým prahem . . . 13
2.12 Prahování s více prahy . . . 13
2.13 Prahování s více prahy 2 . . . 13
2.14 Zdrojový obrázek . . . 14
2.15 Dilatace . . . 15
2.16 Eroze . . . 15
2.17 Otevření . . . 15
2.18 Uzavření . . . 15
2.19 Benchmark jednotlivých knihoven . . . 21
2.20 Staré modulární rozřazení . . . 22
2.21 Posloupnost závislostí modulů od verze 2.2 . . . 22
2.22 Nové grafické prvky . . . 23
2.23 Rozšířený grafický mód . . . 23
2.24 Normální grafický mód . . . 23
2.25 Vykreslené logo pomocí OpenCV . . . 26
2.26 Zdrojový obrázek detekce kůže . . . 28
2.27 Nalezená maska kůže . . . 28
2.28 Zdrojový obrázek kalibrace . . . 31
2.29 Nalezené rohy šachovnice . . . 31
2.30 Optický tok . . . 32
2.31 Detekce lidí . . . 33
2.32 SVM klasifikátor . . . 34
2.33 KNN klasifikátor . . . 34
3.1 Schéma fungování LuaCV . . . 45
3.2 Zdrojový obraz . . . 59
3.3 Detekované čáry . . . 59
3.4 GUI v kooperaci s GNUPlotem . . . 60
3.5 Histogram v OpenCV . . . 60
3.6 Histogramy jasu obrazu v GNUPlot . . . 60
3.7 Grafický design MUI1 . . . 66
3.8 Grafický design MUI2 . . . 67
3.9 Výběr komponent . . . 67
3.10 Průběh instalace . . . 67
3.11 Vstupní obraz pro zadání dodatečných úloh . . . 73
4.1 Počet implementovaných funkcí a objektů . . . 78

4.2 Používané platformy s LuaCV 78
4.3 Oblasti s nejvyšším počtem stažení 79
4.4 Graf výkonnosti jednotlivých implementací 80

Seznam tabulek

2.1 Seznam podporovaného multimediálního softwaru 30
2.2 Základní datové typy 43
2.3 Seznam callback funkcí metatabulky 44
3.1 Dynamické načítání sdílené knihovny 46
3.2 Nejčastěji používané instalační programy 63
4.1 Výsledky výkonnosti implementací OpenCV 79

1. Úvod

Má diplomová na téma *Propojení knihovny pro zpracování obrazu s jazykem Lua* navazuje na mou bakalářkou práci s názvem *Rozšíření knihovny pro zpracování obrazu* z roku 2010, ve které jsem byl veden Ing. Petyovským. Mým cílem bylo nastudovat a realizovat postupy tvorby přemostění pro jazyk Lua. Práce byla do jisté míry novátorská, do té doby pro knihovnu OpenCV neexistovalo přemostění do jazyka Lua. Výsledkem bylo přemostění, které nebylo zcela kompletní, projevovaly se u něho chyby se správnou paměti a použité API nebylo zcela jednotné.

Proto jsem se rozhodl v tomto tématu pokračovat s tím, že přemostění LuaCV bude dopracováno a budou odstraněny výše zmíněné nedostatky.

V úvodní kapitole bude obsaženo učivo kurzu Počítačového vidění MPOV a nastíním základní matematické aparáty používané ve cvičeních. Dále vypracuji některé z úloh vyučovanými v tomto kurzu v prostředí Matlab.

Knihovně OpenCV bude věnována další část práce a cílem bude seznámit s její modulární strukturou a běžně používanými technikami díky příkladům reprezentující charakteristické funkce z jednotlivých modulů. Následně budou vypracovány prototypové úlohy z kurzu MPOV do jazyka C/C++ a knihovny OpenCV.

pak bude vypracováno a detailně popsáno přemostění OpenCV knihovny do skriptovacího jazyka Lua. Ta knihovna by měla obsáhnout běžně používanou funkcionalitu z OpenCV. V jejím rámci budou realizovány mechanismy pro snadnou distribuci a kompilaci, tak aby byla přístupná co největšímu počtu uživatelů. Předpokládá se využití některých běžně dostupných aplikací vydaných pod svobodnou licencí.

Vytvořené přemostění bude zahrnuto ve výuce kurzu MPOV formou nepovinného cvičení a zápočtového projektu. V rámci cvičení budou realizovány tři úlohy v LuaCV dle učiva probíraném v tomto kurzu. Je možné, že se vyskytnou komplikace kvůli syntaktické odlišnosti jazyka Lua a prostředí Matlab, na které jsou studenti zvyklí. Problémy také mohou nastat s konzolovým ovládáním interpretru jazyka Lua. V rámci těchto aktivit v kurzu MPOV je počítáno s odhalením nedostatků knihovny LuaCV nebo jejich chyb. Tyto vady budou průběžně zkoumány a opravovány.

Knihovna LuaCV bude vydána pod svobodnou licencí, a proto se počítá s její distribucí skrze některý z populárních serverů. Mezi nejpravděpodobnější servery patří *SourceForge, GitHub* nebo *SoftPedia*. Od těchto serverů se slibuje rozšíření povědomí o knihovně LuaCV a tím zvětšení její uživatelské základny.

V poslední části práce bude vyhodnocena úspěšnost knihovny LuaCV a její vlastnosti. Zejména se počítá s porovnáním výkonnosti přemostění oproti nativnímu kódu a přemostění v jazyce Python a závěrem s vyhodnocením výsledků.

2. Teoretický úvod

Tato část práce zahrnuje základní matematické vztahy a metody využívané v kurzu Počítačového vidění MPOV na Fakultě elektrotechniky a komunikačních technologií (FEKT) VUT v Brně. Dále jsou vypracovány prototypové příklady vypracované v jazycích Matlab a C++ dle zadání ze školního roku 2010/2011.

Dalším cílem je snaha přiblížit základy použití knihovny OpenCV, její možnosti a principy používané pro realizaci projektů zabývajících se počítačového vidění, maticové algebry či strojového učení. V poslední části této kapitoly je ve stručnosti ukázán skriptovací jazyk Lua a jeho výhody pro tvorbu přemostění a rozšíření v jazyku C/C++.

Kapitola by měla obsáhnout potřebné teoretické základy pro snazší pochopení praktické části práce.

2.1. Kurz MPOV

Kapitola se zabývá praktickými úlohami vyučovanými v kurzu Počítačového vidění. Klade si za cíl teoreticky popsat používané techniky a následně je realizovat ve formě prototypových úloh pro prostředí Matlab s využitím *Image Processing Toolbox*[1], které je v tomto kurzu hojně využíváno. V kurzu MPOV se vyučuje spousta různých technik pro práci s obrazem, ale tato práce se bude věnovat pouze těm vyučovaným ve cvičeních viz elektronické texty předmětu[7].

2.1.1. Základní matematické vztahy

Tato sekce pojednává o základních matematických metodách, jenž jsou nezbytné pro realizaci úloh z počítačového vidění. Bude se jednat zejména o zpracování obrazu pomocí histogramů, segmentačních technik, využití hranových detektorů a některé z matematických morfologických transformací.

Histogramy

Histogramy se široce využívají v analýze obrazu, kde zobrazují distribuci četnosti jasových úrovní. Můžeme tedy říci s jakou pravděpodobnosti má obrazová funkce $f(x, y)$ v bodě daném souřadnicemi x a y určitou hodnotu jasu. Z těchto informací můžeme například vydedukovat velikosti jasových úrovní objektu a pozadí v obrazu, najít optimální osvětlení scény pro pořízení dat a určit vhodné prahy pro segmentaci.

Nutno podotknout, že histogramy neberou v potaz uspořádání a pozici jednotlivých bodů v obraze, a proto se histogramy různých obrazů mohou rovnat. U histogramů lze určit i jiné vlastnosti, než jen rozložení jasových úrovní. Zejména jde o průměrnou hodnotu, kontrast, energii a entropii[2], které mohou být v některých případech více vypovídající než jen hodnoty jasu, více viz [5].

[1] Image Processing Toolbox je modul pro prostředí Matlab, který se zabývá zpracováním a analýzou obrazu. Jeho výhodou je naprostá integrace s prostředím Matlab a snadné používání jeho funkcí. Nevýhodou ovšem zůstává paměťová náročnost a rychlost celého vývojového prostředí Matlab.

[2] Entropie se používá pro měření míry chaosu, kde s jeho stoupající mírou se entropie zvyšuje a události jsou tím méně předvídatelné. Na tomto přístup je založena spousta stochastických metod strojového učení.

Ekvalizace histogramu

Ekvalizace histogramu je algoritmus, při kterém se snažíme histogram přetransformovat tak, aby jednotlivé jasové úrovně byly zastoupeny zhruba stejně. Tím pádem se zvýší kontrast pro maxima jasových úrovní a sníží pro minima. Tato transformace se používá také z důvodu, že odpovídá citlivosti lidského oka. V rovnici 2.1 můžeme vidět vyjádření ekvalizace digitálního obrazu, kde q_0, q_k je interval výstupních jasů, p_0, p interval vstupních jasů a $H(i)$ je histogram obrazu více viz publikace[1].

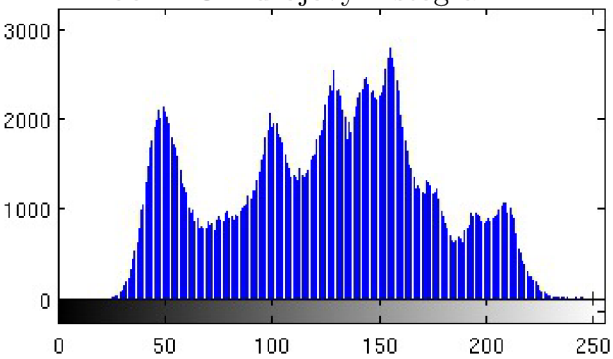
$$q = \frac{q_k - q_0}{N^2} \cdot \sum_{i=p_0}^{p} H(i) + q_0 \tag{2.1}$$

Na obrázku 2.1 a 2.2 můžeme vidět rozdíl mezi zdrojovým obrazem a ekvalizovaným. Dále pod nimi můžeme vidět jejich histogramy 2.3 a 2.4, kde je zřetelně vidět rovnoměrné rozložení jasových úrovní po celé stupnici oproti původnímu obrazu.

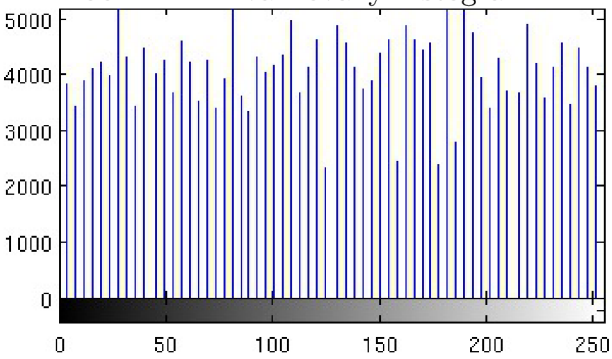
obr. 2.1: Zdrojový obraz

obr. 2.2: Ekvalizovaný obraz

obr. 2.3: Zdrojový histogram



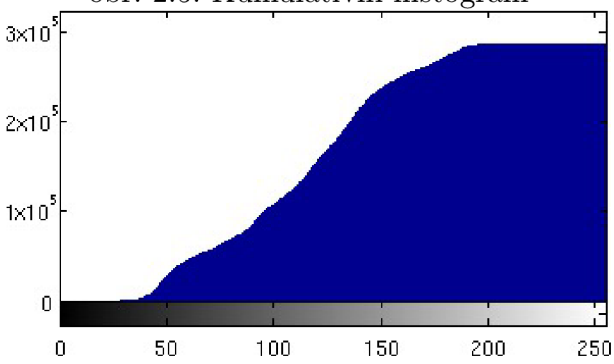
obr. 2.4: Ekvalizovaný histogram



Kumulativní histogram

Vedle klasických histogramů se ve zpracování obrazu používá i kumulativní histogram. V něm každá složka p obsahuje sumu všech předchozích složek normálního histogramu, z čehož vyplývá, že kumulativní histogram je vyjádřen rostoucí křivkou. Matematické vyjádření lze vidět v rovnici 2.2 a jeho znázornění na obrázku 2.5.

obr. 2.5: Kumulativní histogram



$$H_k(i) = \sum_{i=p_0}^{p} H(i) \tag{2.2}$$

Z rovnice vyplývá, že převod mezi těmito histogramy je bezztrátový, a proto mezi nimi můžeme přecházet, aniž by nám informace degradovaly.

Pokud bychom znázornili kumulativní histogram ekvalizovaného obrazu, zjistili bychom, že je téměř lineární. Je to z toho důvodu, že při ekvalizaci je intenzita v histogramu distribuována rovnoměrně.

Konvoluční filtry

Základem těchto filtrů je úprava jasu pomocí blízkého okolí. Podle typu okolí můžeme tyto filtry dělit na lineární a nelineární. Lineární filtry počítají blízké okolí jako lineární kombinaci jasů v obraze $f(x, y)$. Matematickým vyjádřením je rovnice 2.3, kde x, y je aktuální souřadnice bodu v obraze, h je konvoluční jádro. Tato rovnice vyjadřuje „diskrétní konvoluci“ viz [4]. Nejčastěji se využívá pravoúhlého jádra, aby výsledná hodnota bodu byla symetrická vůči okolním bodům. Jádra $h(x, y)$ bývají výrazně menší než filtrovaný obraz $f(x, y)$.

$$g(x, y) = f(x, y) * h(x, y) = \sum_{i=-\frac{S}{2}}^{\frac{S}{2}} \sum_{j=-\frac{R}{2}}^{\frac{R}{2}} f(x - i, y - j) \cdot h(i, j) \tag{2.3}$$

Vyhlazování šumu

Tato metoda je založena na potlačování vyšších frekvencí v obraze, kde požadavkem je potlačení šumu. Nevýhodou je ovšem potlačení všech vyšších frekvencí, a tedy i hran a čar. I přes svoji jednoduchost se stále používá obyčejné průměrování, ta výslednou hodnotu jasu vypočítává jako aritmetický průměr okolních bodů. Pokud máme k dispozici více obrázků se stejnou scénou můžeme výsledný bod průměrovat jako průměr stejných bodů ze všech obrazů, viz rovnice 2.4, kde m je počet obrazů více viz [1].

Pokud máme pouze jeden obraz, jsme nuceni použít „lokální průměrování“, kde předpokládáme, že v blízkém okolí bodu jsou jasové úrovně podobné. Obyčejnou maskou je například 2.6 nebo maska s „Gaussovským rozdělením“ 2.7 vypočtena pomocí rovnice 2.5. Aplikace Gaussovského konvolučního jádra je vidět na obrázku 2.7, jenž je vypočítán z obrazu 2.6.

$$f(x, y) = \frac{1}{m} \cdot \sum_{k=1}^{m} g(x, y) \tag{2.4}$$

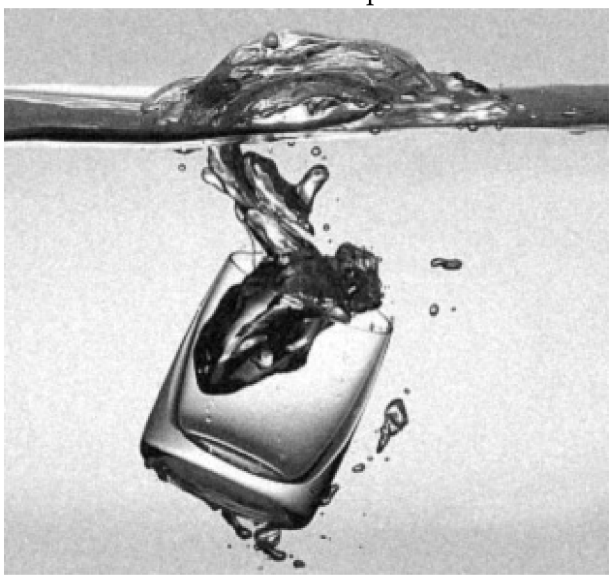
$$f(x, y) = \frac{1}{2\pi\sigma^2} \cdot e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{2.5}$$

$$h = \frac{1}{9} \cdot \begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.6}$$

$$h = \frac{1}{16} \cdot \begin{bmatrix} 1 & 2 & 1 \\ 2 & 4 & 2 \\ 1 & 2 & 1 \end{bmatrix} \tag{2.7}$$

obr. 2.6: Zdrojový obrázek

obr. 2.7: Gaussovské průměrování



Hranové operátory

Gradientní metody se používají pro potlačení nižších frekvencí, a tedy ke zvýraznění hran. Negativním výsledkem je ale i zvýraznění šumu viz [3]. V digitálním obraze se pro výpočet gradientu místo parciálních derivací aproximuje diferencí. Velikost a směr gradientu jsou dány rovnicemi 2.8 a 2.9 viz publikace [1].

$$\Delta_i g(i,j) = g(i,j) - g(i-n,j) \tag{2.8}$$

$$\Delta_j g(i,j) = g(i,j) - g(i,j-n) \tag{2.9}$$

Mezi hranové detektory založených na konvoluci patří „Robertsův operátor“ 2.10, „Sobelův operátor“ 2.12, „Prewittův operátor“ 2.11 a „Laplaceův operátor“ 2.13. Posledně jmenovaný se od ostatních liší tím, že neaproximuje první derivaci obrazu, ale druhou, která není závislá na směru viz konvoluční jádra.

Porovnání hranového detektoru první a druhé derivace je ukázáno na obrázcích 2.8 a 2.9. Jako zdrojový obrázek byl použit 2.6.

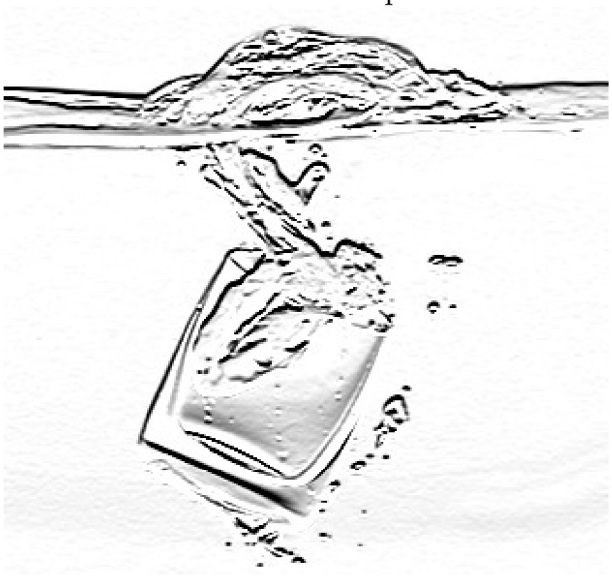
$$h_R = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \tag{2.10}$$

$$h_S = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \tag{2.12}$$

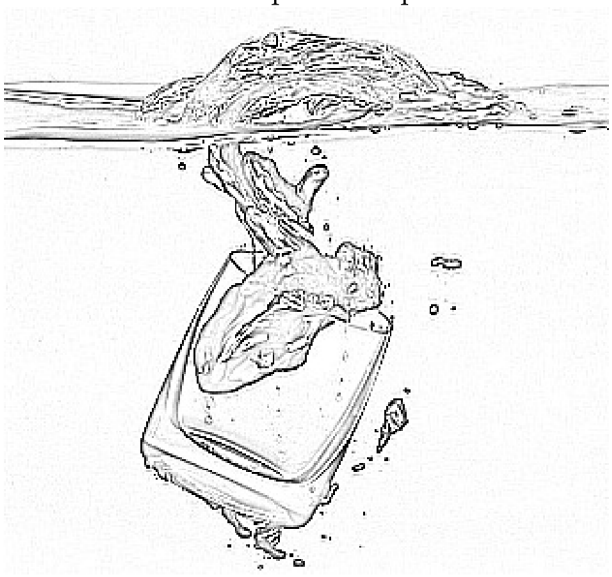
$$h_P = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{2.11}$$

$$h_L = \begin{bmatrix} 1 & 1 & 1 \\ 1 & -8 & 1 \\ 1 & 1 & 1 \end{bmatrix} \tag{2.13}$$

obr. 2.8: Sobelův operátor



obr. 2.9: Laplaceův operátor



Segmentace obrazu

Segmentaci chápeme jako jeden z nejdůležitějších kroků při identifikaci objektů v obraze. Snažíme se obrazová data upravit tak, aby jejich části korespondovaly se skutečnými objekty a bylo možno je snadno rozlišit. Častým případem bývá rozlišení objektů pomocí jasových úrovní v obraze. Při takovýchto postupech bohužel nemůžeme zaručit dokonalé rozlišení objektů, častou a stěžující eventualitou je také přítomnost šumu a nedokonalost pořízených dat, a proto mluvíme o částečné segmentaci.

Prahování

Prahování je nejjednodušší ze segmentačních technik. Vyznačuje se nenáročností na výpočetní výkon, jednoduchostí implementace algoritmu, ale také nepřesností segmentovaných dat. Tento postup vychází ze skutečnosti, že se objekty a pozadí navzájem liší v jasových úrovní, pak je možno je jednoduše odlišit. Matematickým vyjádřením prostého prahování by byla rovnice 2.14, kde $g(i,j)$ je segmentovaný obraz a $f(i,j)$ původní. Velikost prahu pak určuje konstanta t.

$$g(i,j) = \begin{cases} 0 & \in f(i,j) < t \\ 1 & \in f(i,j) \geq t \end{cases} \tag{2.14}$$

Z rovnice vyplývá, že výsledkem je binární obraz, kde obvykle bílou barvou označujeme popředí a černou pozadí. Na obrázcích 2.10 a 2.11 můžeme vidět výsledek prahování s různými prahy obrázku 2.6. Z obrázků je jasně vidět hlavní nevýhoda prostého prahování. Pokud má pozadí podobné jasové úrovně jako objekt, je segmentováno také. Neméně záleží i na analýze obrazu a následné volbě nejvýhodnějšího prahu. V triviálních úlohách prahování postačí jako segmentační technika, ale v komplexnějších případech jej používáme pouze pro předzpracování scény. Problémy s výběrem univerzálního prahu nám ulehčuje následují algoritmus, a to prahování s více prahy.

obr. 2.10: Prahování s optimalním prahem

obr. 2.11: Prahování s vysokým prahem

Prahování s více prahy

Prahování s více prahy je segmentační algoritmus, kdy výsledkem už není binární obraz jako v předchozím případě, ale obraz s tolika jasovými úrovněmi jako počet prahů. Díky tomu můžeme lépe rozlišit objekty od pozadí i pokud jejich jasové úrovně nejsou příliš kontrastní. V rovnici 2.15 vidíme matematické vyjádření tohoto prahování a na obrázcích 2.12 a 2.13 můžeme vidět rozdíly v úspěšnosti segmentace oproti prostému prahování z minulé sekce 2.1.1.

$$g(i,j) = \begin{cases} 1 & f(i,j) \in t_1 \\ 2 & f(i,j) \in t_2 \\ n & f(i,j) \in t_n \\ 0 & f(i,j) \quad jinak \end{cases} \tag{2.15}$$

obr. 2.12: Prahování s více prahy

obr. 2.13: Prahování s více prahy 2

Určení vhodného prahu

Určení vhodné velikosti prahu se opírá o znalost obrazu. Pokud například víme, že hledaný objekt zaujímá určité procento v obraze, můžeme pak snáze najít v histogramu hodnotu jasu odpovídající barvě objektu. Této metodě se říká „procentní prahování“.

Jiné metody se opírají například o znalost tvaru histogramu, kde předpokládáme, že každý z objektů v obraze má podobné jasové úrovně. Pak v histogramu můžeme vidět maxima dle počtů objektu a práh určíme mezi nimi.

Morfologické transformace

Matematická morfologie je poměrně širokou oblastí v analýze obrazu, která vychází z vlastností bodových množin. Můžeme ji aplikovat na obrazy s více jasovými úrovněmi, ale pro jednoduchost budeme brát v potaz pouze binární obrazy. Ke každé morfologické transformaci existuje její „duální transformace“.

obr. 2.14: Zdrojový obrázek

Pro popis objektů v rovině používáme „Euklidův prostor“ viz [1], kde binární obraz je reprezentován bodovou množinou. Body příslušejícím objektům jsou popsány souřadnicemi. Zbytek pak má hodnotu 0.

Prakticky využíváme relace obrazu s jinou menší bodovou množinou, která se nazývá „strukturní element“. Převážně mají symetrický vzhled jako čtverec, obdélník nebo kříž, ale mohou být i nesymetrické. Obrázky v této kapitole jsou vypočteny pomocí strukturního elementu tvaru čtverce o velikost hrany $15px$ a zdrojového obrazu 2.14.

Většina standardních algoritmů implementujících morfologické operace pracuje s bílými body v digitálním obraze na rozdíl od černých. Pro větší přehlednost byly barvy v těchto obrázcích invertovány.

Dilatace a Eroze

Dilatace a eroze jsou duální matematické transformace[3]. Zatímco dilatace představuje součet dvou množin, tak eroze množiny odečítá. Obě metody jsou invariantní vzhledem k posunutí. Ze vztahů dilatace 2.16 a eroze 2.17[1] je jasné, že jsou k sobě duální, kde X, Y jsou množiny bodů a x, y jsou jednotlivé body množin. Proměnnou E je označen „Euklidův prostor“. Porovnání výsledků obou transformací můžeme vidět na obrázcích 2.15 a 2.16.

$$X \oplus Y = \{d \in E^2 : d = x + y, x \in X, y \in Y\} \qquad X \oplus Y = \bigcup_{y \in Y} X_y \tag{2.16}$$

$$X \ominus Y = \{d \in E^2 : d + y \in X \; pro \; \forall \; y \in Y\} \qquad X \ominus Y = \bigcap_{y \in Y} X_{-y} \tag{2.17}$$

[3]Duální operace jsou takové, které dělají pravý opak, ale jejich vzájemnou aplikací nezískáme původní data. Tedy nejsou svými inverzní funkcemi.

obr. 2.15: Dilatace

obr. 2.16: Eroze

Otevření a uzavření

Dilatace a eroze nejsou inverzní transformace. Proto jejich kombinací můžeme získat další typy morfologických transformací, zejména jde o otevření a uzavření. Otevření je definováno jako eroze následována dilatací. Pak v 2.18 můžeme vidět rovnici „otevření“ množiny X strukturním elementem Y. Naopak dilatace následována erozí se nazývá „uzavření“ a lze vidět v rovnice 2.19.[1]

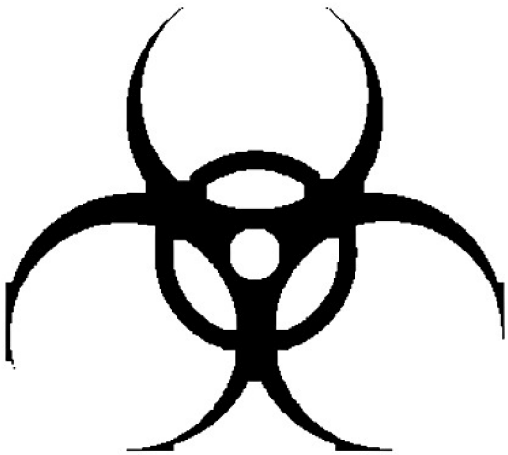
$$X \circ Y = (X \ominus Y) \oplus Y \qquad (2.18) \qquad X \bullet Y = (X \oplus Y) \ominus Y \qquad (2.19)$$

Někdy se otevření a uzavření používá k potlačení detailů v obraze tím, že použijeme větší strukturní element než detaily. Tvar obrazu se příliš nezmění. Obrázky 2.17 a 2.18 ukazují aplikaci otevření a uzavření na zdrojovém obrázku 2.14.

Uzavření spojuje body, které jsou blízko sebe a tím zaplní úzké mezery, naproti tomu otevření tyto blízké body odděluje.

obr. 2.17: Otevření

obr. 2.18: Uzavření



2.1.2. Současná realizace úloh

V rámci kurzu MPOV je během semestru vypracováno deset cvičení, kdy každé z nich se zaměřuje na jiné okruhy látky. Vybral jsem tři z nich dle obsahu kapitol, které jsem popsal výše v kapitole 2.1.1. Všechny tyto úlohy jsou vypracovány pomocí prostředí *Matlab* a *Image processing toolbox*. Zde uvedená zadání jsou kopií z elektronických textů používaným na cvičeních viz [7].

Úloha 4

Úloha 4 se zabývá segmentací obrazu, pomocí různých typů prahů. Zejména jde o prahování s jedním a více prahy, adaptivní prahování s jedním prahem a o adaptivní prahování s určením prahu z blízkého okolí pomocí histogramu.

Zadání

1. Naprogramujte kód pro prahování libovolného obrazu s jedním a dvěma pevnými prahy.
2. Naprogramujte jasově adaptivní prahování s jedním prahem na libovolném obrazu tak, aby ve výsledném obrazu byly objekty správně segmentovány.
3. Obdobně jako v předchozím bodě stanovte adaptivní práh z hlediska prostorového rozložení, čili počítejte jasově adaptivní práh vždy jen pro určitou oblast obrazu a pro další oblast jej stanovte znovu.

Vypracování

Ve skriptu 2.1 vidíme použití prahovacích technik ze zadání. Pro adaptivní prahování s využitím proměnného prahu určeného z blízkého okolí byla sestavena funkce viz př. 2.2.

př. 2.1: Prahování obrazu

```
1  clc; clear all; close all;
2  img = imread('lena_norm.png');
3  [height,width,depth]=size(img);
4  if (depth>1)
5    img=rgb2gray(img);
6    depth=1;
7  end

9  thresh_1=logical(zeros(height,width,'uint8'));
10 thresh_1(img>100)=1;

12 thresh_2=zeros(height,width,'uint8');
13 thresh_2 (:,:) =127;
14 thresh_2(img>150)=255;
15 thresh_2(img<100)=0;

17 adap_thresh=logical(zeros(height,width,'uint8'));
18 derivace=abs(diff(diff(imhist(img))));
19 level=find(derivace == max(derivace));
20 if (length(level)>1)
21   level=max(level);
22 end
23 adap_thresh(img>level)=1;

25 figure
26 subplot(2,2,1);imshow(thresh_1);title('Jeden_pevny_prah');
27 subplot(2,2,2);imshow(thresh_2);title('Dva_pevne_prahy');
28 subplot(2,2,3);imshow(adap_thresh);title('Adaptivni_prahovani');
29 subplot(2,2,4);imshow(adaptivethresh(img,16));title('Adaptivni_
       prahovani_dle_his.');
```

Komentář

V první části skriptu načítáme zdrojový obrázek a pokud je to třeba, převádíme jej na šedotónový pomocí funkce *rgb2gray*.

Prahování s jedním pevným prahem je triviální úloha a lze implementovat jednoduše na jeden řádek.

Pro prahování s dvěmi pevnými prahy postupujeme obdobně jako s jedním prahem. Ve výsledku vznikne tříbarevný obrázek a ne binární jako v předchozím případě.

Pro výpočet vhodného prahu pro adaptivní prahování je použito hledání druhé derivace četnosti jasových složek z histogramu *imhist*. Tento příklad se nijak neliší od prahování s jedním pevným prahem, až na to, že práh je vypočítáván dynamicky.

Nakonec skriptu vše vykreslíme. Pro výpočet adaptivního prahování s různými prahy pro části obrazu je použita funkce *adaptivethresh* z př. 2.2.

př. 2.2: Adaptivní prahování

```
1  function [res]=adaptivethresh(img,varargin)
2  %USAGE: adaptivthresh(img,sub_count=8)
3  % compute adaptive threshold on image with variable count of
       subsections.
4  %EXAMPLE:
5  %Adaptive thresh in 4 regions adaptivethresh(img,4);
6  %Adaptive thresh in 8 regions adaptivethresh(img);

8  sub_count=8;

10 if length(varargin) == 1
11  sub_count=varargin{1};
12 end

14 [height,width,depth]=size(img);
15 if (depth>1)
16   img=rgb2gray(img);
17   depth=1;
18 end

20 res = logical(zeros([height,width]));

22 for j = [0 : sub_count-1]
23     y = height/sub_count*j;
24     for i = [0 : sub_count-1];
25         x = width/sub_count*i;
26             sub = img(y + 1:y + height/sub_count , x + 1:x + width
                   /sub_count);

28             windowSize = 3;
29             hist = filter(ones(1,windowSize)/windowSize,1,double(
                   imhist(sub)));
30             derivace = abs(diff(diff(hist)));
31             level=find(derivace == max(derivace));

33             if (length(level)>1)
34               level=max(level);
35             end

37             res(y + 1:y + height/sub_count , x + 1:x + width/
                   sub_count)=im2bw(sub,level/255);
38     end
39 end
```

Komentář

Tato funkce realizuje adaptivní prahování pro variabilní počet subsekcí obrazu, kde počet subsekci je nepovinný parametr, a pokud není zadán je automaticky použit počet 8.

Kontrola zda byl zadán nepovinný parametr *sub_count*.

Pokud není vstupní obrázek šedotónový, musíme ho konvertovat, protože prahovat můžeme pouze obrazy s jedním kanálem.

Vytvoření výstupního obrazu, protože je to prahování s jedním prahem, můžeme jej převést na typ *logical*.

V této části se vypočítávají rozměry subsekci obrazu, protože je pouze jedna proměnná *sub_count*, budou tyto oblasti čtvercové. následně si do proměnné *sub* uložíme konkrétní segment obrazu a nad ním dále pracujeme.

Pro vypočítání optimálního prahu pro konkrétní segment je použita funkce *imhist*. Protože výsledný vektor jasových složek není spojitý, je na něj aplikován filtr *filter*, který chybějící data aproximuje.

Následně hledáme maxima a minima druhé derivace dat a tak vypočteme optimální prah.

Do výstupního obrazu pak uložíme výsledek prahování funkce *im2bw*.

Úloha 6

Tato úloha je zaměřena na procvičení diskrétní konvoluce a práci s různými konvolučními jádry. Zejména půjde o hranové detektory první, druhé derivace a porovnání úspěšnosti oproti „Cannyho detektoru“. Také dojde na seznámení s knihovními funkcemi, které jsou součástí základní výbavy Matlabu aImage Processing Toolboxu.

Zadání

1. Pomocí funkce vytvořené v minulém cvičení pro konvoluci dvou matic navrhněte konvoluční masku (jádro) pro výpočet velikosti hran v horizontálním a vertikálním směru a masce o rozměru $3x3$.

2. Nalezněte v obrazu hrany konvolucí obrazu a těchto hranových operátorů: *Robertsův*, *Prewittův*, *Sobelův*, *Robinsonův*, *Kirschův* a *Laplaceův*.

3. Vyzkoušejte tvorbu filtrů (konvolučních jader) různých velikostí a parametrů pomocí knihovní funkce *fspecial*.

4. Filtry vytvořené podle předchozího bodu aplikujte na obraz pomocí konvoluční funkce vytvořené v minulém cvičení a pomocí knihovních funkcí *conv2* a *imfilter*.

5. Nalezněte v obrazu hrany pomocí knihovní funkce *edge* s použitím metod *Laplace-Gauss*, *Canny* a *Zero-cross*.

Vypracování

V př. 2.4 je navrženo řešení zadání pro detekci hran pomocí gradientních operátorů. Byly vyzkoušeny jak operátory založené na první derivaci, tak i na druhé. Pro vypracování zadání bylo nutné implementovat funkci realizující diskrétní konvoluci viz př. 2.3. Tato funkce je stavěná tak, aby bylo možno použít jakékoliv konvoluční jádro.

př. 2.3: Diskrétní konvoluce matic

```
1  function [result]=convolution(image,kernel)
2  %USAGE: convolution(image,kernel)
3  % Computes convolution of two matrices. Image can be 2D or 3D array and
       kernel must be 2D array.
4  %EXAMPLE:
5  % Gaussian blur convolution(imread('im.png'),(1/(16)) * [1 2 1;2 4 2;1 2 1])
6  % Laplace edge detector convolution(imread('im.png'),[1 1 1;1 -8 1;1 1 1])

8  imtype=class(image);

10 switch imtype
11   case {'uint8','uint16','uint32','uint64'}
12   otherwise
13     error('Image must be integer type.');
14 end

16 kern=struct('data',rot90(rot90(kernel(:,:,1))),'size',size(kernel));
17 if (length(kern.size)>2)
18   error('Convolution kernel must be 2D array.');
19 end

21 im=struct('data',image,'size',size(image));
22 g=round(kern.size(1:2)/2)+1;

24 if (length(im.size)>2)
25   i=im.size(3);
26 else
27   i=1;
28 end

30 result=zeros(im.size,imtype);

32 for j=1:i
33   ret=zeros(im.size(1:2)+(kern.size(1:2)-1),class(kernel));
34   ret(g(1):g(1)+im.size(1)-1,g(2):g(2)+im.size(2)-1) = im.data(:,:,j);

36   for m = [1:im.size(1)]
37     for n = [1:im.size(2)]
38       result(m,n,j) = cast(sum(sum(ret(m:m+kern.size(1)-1,n:n+kern.size(2)
             -1).*kern.data)),imtype);
39     end
40   end
41 end
```

Komentář

Tato funkce realizuje matematickou operaci diskrétní konvoluce dvou matic. Je napsána tak, aby zvládla i barevné obrázky a chovala se stejně jako knihovny funkce *conv2*. Výkonnostně se ale samozřejmě nemohou rovnat, protože *conv2* je implementována v C, a proto je podstatně rychlejší. Navíc jsem byl nucen při implementaci použít cykly *for*, které výpočet podstatně zpomalily.

V první části funkce probíhá kontrola, zda zdrojová data jsou ve správném formátu. Následně pro větší přehlednost vytvářím struktury *im* a *kern*, ze kterými poté pracuji. Nutno poznamenat, že se konvoluční jádro musí otočit o *180°* opakovanou aplikací příkazu *rot90*.

Proměnná *g* pomáhá vypočítat konečnou velikost obrazu, protože oproti původnímu se bude mírně lišit v závislosti na velikosti jádra.

Dále se v cyklu pro každý barevný kanál vypočítá diskrétní konvoluce dle vzorce 2.3. Nakonec se vše zkopíruje do výstupní proměnné *result*.

př. 2.4: Gradientní operátory

```
clc, clear all, close all

img = imread('grayimage.png');

edges=[[-1:1; -1:1; -1:1];[1,0,-1; 2,0,-2; 1,0,-1];
       [1,1,-1; 1,-2,-1; 1,1,-1];[3, 3,-5; 3,0,-5; 3,3,-5];
       [0,1,0; 1,-4,1; 0,1,0]];
titles ={'Prewitt','Sobel','Robinson','Kirsch','Laplace'};

figure(1)
for i=1:length(titles)
  subplot(2,3,i); imshow(convolution(img,edges(i:i+3,:))); title( titles {i
      });
end

titles ={'prewitt','sobel','laplacian'};
functions={@convolution,@conv2,@imfilter}

figure(2);
for i=1:3:length(titles)*3
  for j=1:3
    subplot(3,3,i+j-1); imshow(functions{j}(img,fspecial( titles{(i+2)/3})
        ));
    title( titles {(i+2)/3});
  end
end

titles ={'canny','zerocross','log'};

figure(3);
for i=1:3
 subplot(1,3,i); imshow(edge(img,titles{i},0.007)); title( titles {i});
end
```

Komentář

V první části skriptu se načítá obrázek, u kterého předpokládáme pouze jeden kanál resp. šedotónový obraz.

Následně je do matice *edges* uloženo postupně všech pět hranových operátorů. Stejně tak do proměnné *titles* popisky k těmto operátorům.

Postupně v cyklu necháme vykreslit pomocí funkce *convolution* z př. 2.3 výsledek konvoluce.

V této části definuje pole popisků a pole ukazatelů na funkce *functions*.

V cyklech pak vykreslíme devět výsledků konvoluce pomocí dříve definovaných ukazatelů.

K naplnění celého zadání zde zkoumáme výsledky detektorů hran definovaných pomocí pole *titles*. Následně opět v cyklu vykreslíme.

Úloha 10

Tento úkol se zaměřuje na procvičování morfologických operací a na volbu strukturního elementu. Teoretická část byla popsána v kapitole 2.1.1.

Zadání

1. Vytvořte blok programu řešící dilataci binárního obrazu strukturním elementem čáry o třech horizontálních bodech.
2. Vygenerujte uvedený strukturní element pomocí knihovní funkce *strel*. Pomocí knihovní funkce *imdilate* ověřte správnost výsledků předchozího bodu.
3. Aplikujte na jeden libovolný obraz několikrát operaci eroze *imerode* a najděte různě široké obrysy objektů.
4. Proveďte operaci otevření jako erozi následovanou dilatací a poté také pomocí funkce *imopen*. Porovnejte výsledek těchto operací s výsledkem operace uzavření *imclose*.

Vypracování

V rámci zadání bylo nutné navrhnout funkci řešící dilataci binárního obrazu viz př. 2.5, ta ale neumožňuje měnit strukturní element. Tato funkce je dále použita ve vypracování jednotlivých úkolů viz př. 2.6.

př. 2.5: Dilatace

```
1  function [dilated]=dilatation(im)
2  %USAGE: dilatation(image)
3  % Computes dilatation of binary image and structuring element [1 1 1]
4  %EXAMPLE:
5  % dilatation(imread('image.png'))
6  image=struct('data',im(:,:,1),'size',size(im));
7  if (length(image.size)>2)
8    warning('Converting image to grayscale.');
9    image.data=rgb2gray(im);
10   image.size(3)=1;
11 end
12 if (class(image.data)~='logical')
13   image.data=im2bw(image.data,graythresh(image.data));
14 end

16 dilated=logical(zeros(image.size,'uint8'));

18 for i=1:image.size(1)
19   for j=1:image.size(2)
20     if image.data(i,j)==1
21       if (i~=1)
22         dilated(i-1,j)=1;
23       elseif (i~=image.size(1))
24         dilated(i+1,j)=1;
25       end
26       dilated(i,j)=1;
27     end
28   end
29 end
```

Komentář

Tato funkce realizuje dilataci dvou matic, respektive obrazu a strukturního elementu. V rámci zadání není vyžadováno, aby tato funkce uměla pracovat s libovolným strukturním elementem, a proto je definován napevno ([111]).

Tato dilatace počítá pouze s binárním obrazem, a proto je nutné případná zdrojová data konvertovat. Nejprve zjistíme, zda má obrázek více kanálu než jeden a pokud ano, tak zavoláme funkci *rgb2gray*, která data převede na šedotónová.

Další podmínka zjišťuje, jestli jsou data logická nebo neznaménkový int. Pokud nejsou logická, tak obraz naprahujeme funkci *im2bw* s optimálním prahem získaným z *graythresh*.

Zde vytváříme výstupní proměnnou, nutno poznamenat, že oproti jiným metodám tento nepočítá se zvyšování řádu matice v krajních bodech, a proto výsledný obrázek bude vždy stejně velký.

Nakonec pro všechny body obrazu zkoumáme jejich hodnotu a v případě, že jsou na hodnotě 1, tak jejich okolí také přenastavíme. Pouze pokud je počátkem strukturního elementu krajní bod, tak za okrajem se nestane nic.

př. 2.6: Morfologické operace

```
1  %morfologicke operace
2  clear all;close all;clc;

4  img=imread('srcimg.png');
5  [height,width,channel]=size(img);

7  if channel ~=1
8    img=im2bw(img,graythresh(img));
9  end

11 se=strel('line',3,0);

13 figure(1)
14 subplot(1,2,1); imshow(dilatation(img)); title('fnc_dilatation');
15 subplot(1,2,2); imshow(imdilate(img,se)); title('fnc_imdilate');

17 figure(2)
18 subplot(1,2,1); imshow(img); title('source_image');
19 subplot(1,2,2); imshow(imerode(imerode(imerode(img,se),se),se));
       title('3_x_erode_image');

21 figure(3)
22 subplot(2,2,1); imshow(imdilate(imerode(img,se),se)); title('erode
       ->dilalate');
23 subplot(2,2,2); imshow(imopen(img,se)); title('imopen');
24 subplot(2,2,3); imshow(imerode(imdilate(img,se),se)); title('dilate
       ->erode');
25 subplot(2,2,4); imshow(imclose(img,se)); title('imclose');
```

Komentář

Na začátku skriptu načteme vstupní obrázek a zajistíme, aby byl binární pomocí funkce *im2bw* a *graytresh*.

Následně vygenerujeme strukturní element tvaru *čára* o šířce $3px$ a naklonění 0 funkce *strel*.

Do okna *Figure 1* necháme vykreslit výsledky knihovní funkce *imdilate* a funkce z př. 2.5.

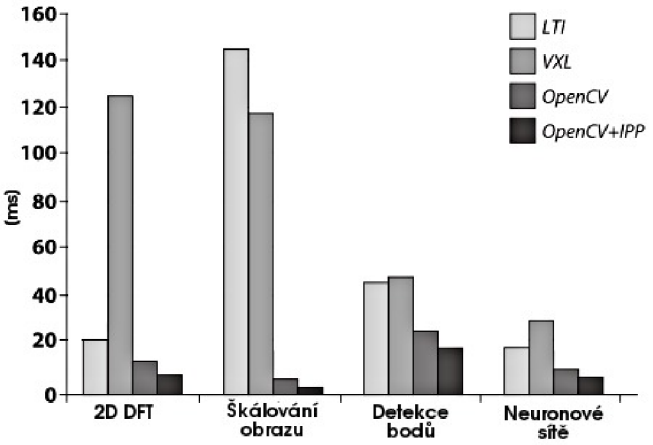
Okno *Figure 2* obsahuje původní obrázek a jeho transformaci po trojité aplikaci eroze funkcí *imerode*.

V poslední části programu je důkaz, že morfologická operace otevření, uzavření je kombinací eroze a dilatace, protože mezi výslednými obrazy není znatelný rozdíl.

2.2. Knihovna OpenCV

OpenCV je knihovna realizující algoritmy využívané v počítačovém vidění. Začala být vyvíjena společností Intel od roku 1999 do 2008, poté ji převzala společnost Willow Garage[4], která ji spravuje dodnes. Samotné základy knihovny byly položeny tak, aby knihovna byla multiplatformní, obsahovala zpravidla otevřený kód a hlavně byla efektivní, co se týče využití systémových prostředků. Úplné počátky leží v knihovně IPL[5], ze které byly převzaty některé algoritmy a dodnes můžeme vidět jejich pozůstatky v kódu. Intel využil svého místa na trhu s mikroprocesory a nabízí ke koupi knihovnu akceleračních funkcí IPP[6] pro své procesory. Díky těmto vlastnostem se OpenCV stala velmi rychle populární. Jako možnou konkurencí, která ale nedosahuje takových kvalit, mohou být například knihovny LTI a VXL. Při porovnání výkonu těchto knihoven je OpenCV v některých případech rychlejší až o 20% viz [6]. Na obrázku 2.19 můžeme vidět orientační výsledky benchmarků pro jednotlivé knihovny na několika typech úloh.

obr. 2.19: Benchmark jednotlivých knihoven



OpenCV svou funkcionalitou zasahuje do mnoha vědních oborů[7], a proto je vhodným komplexním řešením pro mnoho aplikací bez potřeby použití dalších knihoven třetích stran. Obsahuje více než 700 funkcí rozdělených do 13 modulů. Na rozšiřování se podílí jak jedinci, tak i velké korporace jako SONY, Microsoft, IBM a Google. To je umožněno otevřenou licencí, která vychází z „BSD licence“ a dovoluje nám sdílet a upravovat zdrojový kód, používat knihovnu v komerčních aplikacích apod.

Díky oblibě OpenCV mezi vývojáři byla tato knihovna integrována do mnoha větších projektů. K těm známějším patří projekt ROS[8], který implementuje operační systém pro robotické aplikace. Protože velká část knihovny je implementována v C, je OpenCV možné zkompilovat i na takových zařízeních jako jsou signálové procesory. Většina pokročilejší funkcionality je ale realizována v C++, a pokud použijeme překladač jazyka C, přijdeme o velkou část funkcionality.

[4]Firma Willow Garage se zabývá vývojem softwaru a hardwaru pro robotiku. Zajímavostí je, že někteří z prvotního týmu Intelu nyní pracují na OpenCV v této firmě. Zejména jde o Garyho Bradskiho a Vadima Pisarevského, kteří pracovali na OpenCV od začátku.

[5]Image Processing Library byla knihovna pro zpracování obrazu optimalizovaná pro MMX architekturu vyvíjenou společností Intel. Nikdy ale nebyla příliš populární, a proto od ní společnost upustila. Přestala se vyvíjet koncem roku 1998.

[6]Modul Integrated Performance Primitives optimalizuje aritmetické úlohy přímo pro procesory Intel a díky tomu můžeme výkon OpenCV znatelně zvýšit. Integrace do OpenCV je velice jednoduchá, protože si jej knihovna sama najde a začne využívat.

[7]Využití OpenCV najdeme například v robotice, automatizaci, zdravotnictví, bezpečnosti a samozřejmě i ve školství.

[8]Robot Operating System realizuje knihovny a nástroje pro tvorbu robotických aplikací. Hlavními částmi jsou hardwarová abstrakce, ovládače zařízení, vizualizace atd.

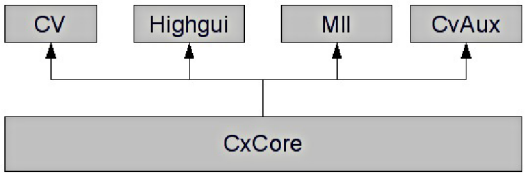
2.2.1. Rozdíly mezi předchozími verzemi

S příchodem verze 2.2 byla OpenCV z velké části přepracována. Funkcionalita se příliš nezměnila, ale byl vylepšen modulární systém, takže již není třeba načítat velké množství funkcí, které v danou chvíli nevyužijeme. Mezi další velmi očekávané změny patří podpora knihovny *Qt* pro zobrazování GUI aplikací, oficiální podpora operačního systému Android a první implementaci paralelizace *CUDA* na grafických kartách NVIDIA.

Nový modulární systém

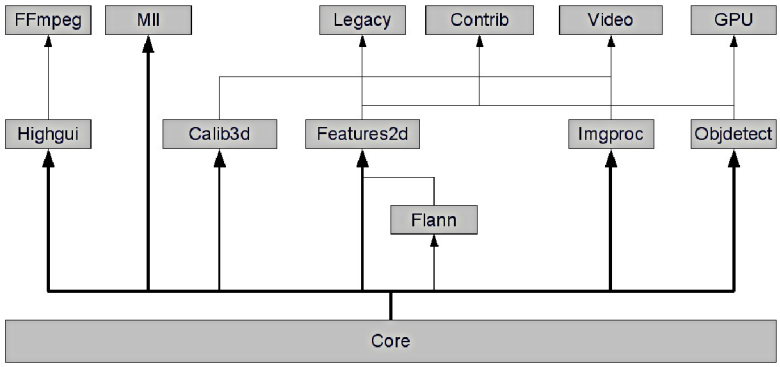
Struktura rozdělení modulů byla ve verzi 2.2 úplně přepracována. Od první verze OpenCV byla knihovna strukturována do pěti částí, viz obrázek 2.20. Knihovna *CxCore* obsahovala základní maticový počet a obslužné funkce z OpenCV API. Všechny ostatní moduly na ní byly závislé. Modul *CV* obsahoval všechny funkce, které měly něco společného s počítačovým viděním. Modul *CvAux* obsahoval testovací funkce, které čekaly na odzkoušení a zařazení do ostatních modulů. *HighGui* obsahoval vrstvu nad grafickým rozhraním a přístupu ke kamerám. Poslední byla knihovna *Mll*, která realizovala algoritmy strojového učení.

obr. 2.20: Staré modulární rozřazení



Na obrázku 2.21 je znázorněno nové rozdělení modulů a jejich vnitřní závislosti. Pouze moduly *Core* a *Highgui* zůstaly ve složení zhruba stejné. Všechny ostatní byly rozděleny do tématicky malých knihoven, kde například bývalý modul *CV* byl rozdělen do 7 modulů (*Imgproc*, *Tracking*, *Features2d*, *Flann*, *Calib3d*, *Objdetect* a *Legacy*). Za povšimnutí také stojí modul *Contrib*, který obsahuje všechen software třetích stran, který není plně kompatibilní s licenčním ujednáním OpenCV a při jeho použití se zavazujete respektovat jejich speciální požadavky.

obr. 2.21: Posloupnost závislostí modulů od verze 2.2



Knihovna Qt

Od OpenCV verze 2.2 vývojáři zařadili nové grafické rozhraní pomocí knihovny Qt. Tato knihovna byla vybrána oproti předchozímu GTK, protože je nejen multiplatformní, ale i snáze portovatelná na různé architektury. Společnosti Nokia a Google již dříve implementovaly knihovnu Qt do svých mobilních zařízení, a proto vývojáři OpenCV neměli problémy s kompatibilitou. I přes zjevné výhody Qt je staré grafické rozhraní GTK stále možno při kompilaci zapnout. Nové rozhraní pracuje se stejným API, a proto je kompatibilní se starými aplikacemi. Na obrázku 2.23 vidíme okno v rozšířeném režimu *CV_GUI_EXPANDED*, to obsahuje nejméně dva prvky *toolbar* a *statusbar*. Obrázek 2.24

zobrazuje okno v normálním režimu *CV_GUI_NORMAL*. Grafické rozhraní se skládá z těchto prvků viditelných na obrázcích:

1. Panel nástrojů *toolbar* s integrovanými funkcemi (přiblížení, oddálení, uložení, posun). Užitečné je tlačítko *Display properties window*, pomocí kterého můžeme zobrazit uživatelsky definovaný formulář například. viz obrázek 2.22.

2. Překrývací *overlay* widget pro výpis informací. Jak můžeme vidět na obrázku 2.24, tak je možno použít omezené formátovací znaky, například i pro odsazování nebo odřádkování.

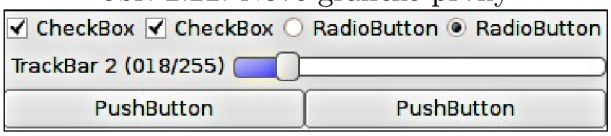
3. Nové umístění posuvného ovládacího prvku *trackbar*, předchozí verze umísťovaly trackbar nad obrázek.

4. Informační panel *statusbar* s polohou kurzoru myši a barevnou skladbu daného aktuálního pixelu. Dostupné pouze v rozšířeném režimu.

5. Vyskakovací *popup* menu při zmáčknutí pravého tlačítka. Obsahuje všechny funkce z panelu nástrojů, ale je dostupné i v normálním režimu.

Novinkou je možnost vytvoření omezeného formuláře, který může obsahovat tři druhy typů widgetů (trackbar, radio button, checkbox, push button). Posloupnost jednotlivých prvků je dána pořadím, jakým je zapíšeme do zdrojového kódu. Na obrázku 2.22 vidíme jedno z možných rozložení formuláře.

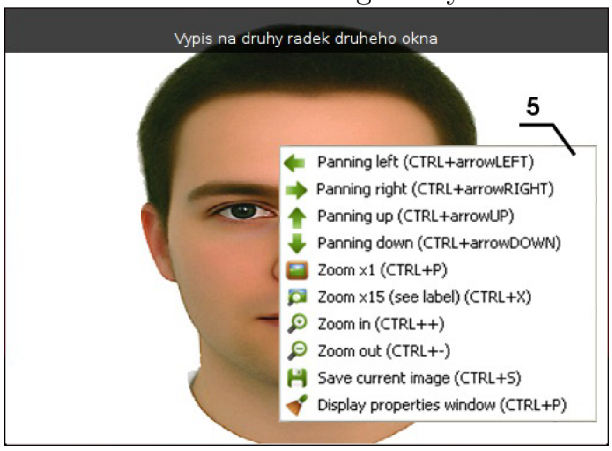
obr. 2.22: Nové grafické prvky



Všechny prvky se vkládají automaticky za sebe do řádku kromě trackbaru, ten je vždy na samostatném. Ve starém grafickém rozhraní GTK bylo možno pouze vytvořit widget typu trackbar, a proto toto rozšíření a velká modernizace přináší nové možnosti ve tvorbě strohého uživatelského prostředí. Pokud ale budeme chtít vyvíjet náročnější grafické aplikace ve smyslu uživatelského rozhraní, je nutné se vzdát tohoto zjednodušeného API a pracovat s rozhraním knihovny QT nebo GTK přímo. Obě tyto knihovny jsou multiplatformní na rozdíl od prostředí Carbon a Windows API, které jsou specifické pouze pro jednu platformu.

obr. 2.23: Rozšířený grafický mód

obr. 2.24: Normální grafický mód



Podpora CUDA

CUDA je architektura vyvinutá společností NVIDIA pro své grafické karty na paralelní zpracování matematických výpočtů pro grafické procesory s vysokým výpočetním výkonem. Podporovány jsou takové GPU, které podporují standardy CUDA 1.1[9] a novější. Pro vývoj algoritmů pro CUDA se používá jazyk *C for CUDA*[10], což není nic jiného než klasická norma C s přidanými rozšířeními od NVIDIA. Zdrojový kód je překládán kompilátorem *PathScale Open64*[11], díky kterému vznikne binární program kompatibilní s grafickými procesory. Implementace v OpenCV realizuje pouze některé datové typy a funkce používané pro počítačové vidění, například jde o geometrické transformace, histogramy nebo Haarovy kaskády. V současnosti je podporována CUDA v OpenCV pouze pro platformy Windows a Linux, plánuje se i implementace pro Mac OS X.

2.2.2. Podporované platformy a software

OpenCV je multiplatformní knihovna, a proto je podporována širokou škálou systémů. Patří mezi ně Windows, Linux, BSD, Mac OS X a nově podporovaný Android. Mezi základní závislosti společné pro všechny platformy patří:

- *Kompilátor C/C++* - Visual Studio (VS2008-2010), MinGW, Xcode 3.2[12], GCC 4.3 a vyšší.
- *CMake 2.6* - Nástroj pro generování projektů pro velkou škálu kompilátorů a platforem.

Nepovinnými závislostmi jsou:

- *Python 2.6.x nebo 2.7.x* - Nutný pro kompilaci přemostění OpenCV do jazyka Python.
- *Knihovna TBB 2.2* - Knihovna implementující sofistikovanou správu a manipulaci vláken procesů od společnosti Intel. V předchozích verzích se používala knihovna *OpenMP*.
- *Knihovna Qt 4.6* - Knihovna grafického prostředí napsána v C++ na rozdíl od GTK. Nutná pro nové grafické API, které bylo představeno v OpenCV 2.2, viz kapitola 2.2.1.
- *Knihovna IPP 5.1-6.1* - Vysoce optimalizovaná knihovna společnosti Intel, které dokáže zrychlit některé úlohy jako konverzi barev, učení rozhodovacích stromů nebo DFT transformaci.

[9] Jedním z důvodů pro podporu CUDA až verze 1.1 je to, že obsahuje atomické instrukce pro práci s 32 bitovým celočíselným typem v globální paměti, které předchozí verze neimplementovaly.

[10] *C for CUDA* rozšiřuje jazyk C o možnost definovat funkce nazývané *kernels*, které když jsou zavolány, tak jsou spuštěny v určitém počtu CUDA vláken. Funkce se definují pomoci klíčového slova *__global__* a následně při zavolání se určí přesný počet vláken pomocí $<<< 1, N >>>$, který udává interval použitých vláken. Výsledné zavolání pak vypadá např. $function <<< 1, 5 >>> (param)$;.

[11] Open64 je vysoce optimalizovaný open source kompilátor jazyků C/C++ a Fortran 77/95 pro 64 bitové architektury s podporou knihovny OpenMP používanou pro lepší paralelizaci.

[12] Xcode je vývojové prostředí pro Mac OS X, které obsahuje upravené GCC nebo LLVM. Lze v něm vyvíjet i grafické rozhraní a programy, které jsou velmi dobře integrovatelné do systému.

Tento seznam je společný pro všechny platformy, na kterých jde OpenCV přeložit. Každá z platforem má ale ještě své specifické závislosti.

Operační systémy odvozené od Unixu

Mezi tyto podporované architektury se řadí zejména Linux, BSD, Mac OS X a nově i Android. Překlad OpenCV na těchto platformách je víceméně stejný až na Android. Mezi specifické závislosti systémů Linux, BSD, Mac OS X patří:

- *pkg-config* - Program pro generování parametrů pro GCC z hlaviček programů. Příkaz $pkg-config--libs--cflagsopencv$ vytvoří výstup $-I/usr/include/opencv-lopencv_core-lopencv_imgproc-lopencv_highgui-lopencv_ml-lopencv_objdetect$, který obsahuje cestu k hlavičkám OpenCV a s jakými knihovnami se má program dynamicky slinkovat.
- *Knihovna GTK+ 2.x* - GTK+ je vrstva nad GTK napsaná v C++. Tato grafická knihovna je nutná pro staré highgui API na BSD a Linuxu.
- *Knihovna Carbon* - Grafická knihovna používaná na Mac OS X pro staré highgui API.
- *Doprovodné knihovny obrazu* - Zejména jde o libjpeg, libtiff, libjasper, libpng a zlib.
- *Programy pro kódování/dekódování videa* - Programy ffmpeg, libgstreamer, libxine, unicap, libdc1394 2.x zajišťují možnost real-time komprimace videa při zpracování obrazu pro zmenšení velikosti zaznamenaných dat.

V případě Androidu je nutno poznamenat, že se často používá tzv. cross-compilation neboli kompilace pro jinou architekturu. Takže můžeme vyvíjet aplikace pro Android na běžných a výkonnějších platformách. Pro správné přeložení je potřeba mít takovéto dodatečné programové vybavení:

- *Android NDK r5b* - Souhrn vývojových nástrojů pro Android pro kompilaci v nativním kódu.
- *SWIG* - Nástroj pro automatické vytváření přemostění dle šablon.
- *JDK 5 or JDK 6* - Java Development Kit je nutný pro kompilaci OpenCV Java přemostění, pokud bychom nechtěli knihovnu používat v nativním kódu.
- *Ant* - Program pro generování kompilačních projektů. Principem podobný nástroji Make, ale s tím rozdílem, že Ant je primárně určen pro Javu a používá XML formát.

Operační systémy Windows

Nejčastěji se pro kompilaci OpenCV na platformách Windows používá IDE MS Visual Studio, kde je postup naprosto bezproblémový. Pokud nechceme platit za licenci společnosti Microsoft, můžeme využít překladače GCC z balíku MinGW. Ten přináší do architektury Windows základní unixové nástroje a postup pro kompilaci je u něj stejný jako například na Linuxu nebo BSD. Musíme ale počítat s komplikacemi, protože tyto nástroje nejsou příliš integrované do systému. Vývoj aplikací pro Windows můžeme provádět i ze systémů odvozených na Unixu pomocí přemostění Wine, které implementuje Windows API nad X serverem.

2.2.3. Moduly

Jak už bylo poznamenáno v kapitole 2.2.1, s verzí OpenCV 2.2 byl celý modulární model přepracován. V této kapitole se blíže podíváme na obsah a funkcionalitu těch nejvýznamnějších z nich. Vytvořené příklady jsou psány ve starém C API z důvodu, že na novém C++ API se stále intenzivně pracuje a není zaručena jeho dopředná kompatibilita.

Modul Core

Modul Core je základem celé knihovny. Dá se dále rozdělit na dvě knihovny *types* a *core*. Knihovna *types* implementuje většinu všech datových typů používaných v OpenCV. Důležitou součástí jsou také inicializační funkce k těmto typům a základní pracovní funkce a makra. Mezi nejčastěji používané struktury můžeme řadit například *CvMat*, *IplImage*, *CvPoint*, *CvSize* nebo *CvSeq*. Objektů obsažených v této knihovně je celá řada a jejich celkový počet je cca 100, kde ale mnoho z nich jsou pouze určeny k internímu použití a nemají žádný uživatelský přínos. C++ API implementuje pouze ty nejpoužívanější maticové počty nad objektem *Mat*. Knihovna *Core* implementuje funkcionalitu nad knihovnou *types*. Najdeme zde všechny maticové operace, manipulaci s dynamickými typy, kreslící funkce, základní matematické operace, interní registraci nových modulů a obsluhu výjimek[13]. Na příkladu 2.7 můžeme vidět, jak se používají základní funkce OpenCV a je pomocí nich na obrázku 2.25 vykresleno logo skupiny Počítačového vidění na ústavu UAMT.

obr. 2.25: Vykreslené logo pomocí OpenCV

př. 2.7: Kreslící funkce knihovny Core

```
1  #include <opencv2/core/core_c.h>

3  int main( int argc, char** argv )
4  {

6    CvMat *image=cvCreateMat(400,400,CV_8UC1);
7    CvScalar white=CV_RGB(255,255,255),black=CV_RGB(0,0,0);
8    CvPoint center=cvPoint(200,170);
9    CvFont font;

11   cvSet(image,white);

13   cvCircle(image,center,150,black,CV_FILLED);
14   cvEllipse(image,center,cvSize(110,70),360,0,360,white,CV_FILLED);
15   cvCircle(image,center,40,black,40);

17   cvEllipse(image,cvPoint(250,276),cvSize(200,200),140,0,90,white,3);
18   cvEllipse(image,cvPoint(150,64),cvSize(200,200),50,270,360,white,3);

20   cvInitFont(&font,CV_FONT_HERSHEY_DUPLEX,1,1,0,2);
21   cvPutText(image,"Computer_Vision_Group",cvPoint(15,370),&font,black);

23   cvReleaseMat(&image);
24   return EXIT_SUCCESS;
25 }
```

Komentář

Na začátku programu provádíme inicializaci zdrojového obrazu *image* pomocí funkce *cvCreateMat()*. Ta má za parametry šířku, výšku a velikost pixelu/počet kanálů. Dále definujeme vektory *white* a *black*, které představují barvu pozadí a popředí, pomocí makra *CV_RGB()*. Následně definujeme středový bod *center* a strukturu *font*. Pomocí funkce *cvSet()* vyplníme celý obraz bílou barvou.

Dále pak pomocí kružnic *cvCircle()* a elips *cvEllipse()* vytvoříme základní tvar. Přerušení základního kruhu je uděláno posunutými elipsami s pouze částečným vykreslením.

Funkce *cvInitFont()* inicializuje strukturu *font* definovanou dříve a nastaví typ fontu, velikost, zkosení a šířku. Funkce *cvPutText()* vykreslí text s fontem do obrázku, kde počátek je dán bodem a barva textu černou barvou. Nakonec odalokujeme obraz pomocí funkce *cvReleaseMat()*.

[13] V základním C API jsou výjimky implementování pouze jako *assert* makro s dodatečným výpisem. Pokud používáme C++ API, tak v mnoha objektech jsou standardní výjimky zakomponovány.

V příkladu 2.8 jsou uvedeny některé matematické úlohy s maticemi. Konkrétně je vidět řešení lineárních rovnic „Gaussovou eliminací“, výpočet inverzní matice a její vynásobení se zdrojovou. Dle výstupu programu je vidět, že výsledky jsou korektní. Protože násobení matic $A * A^{-1} = 1$. Na tomto příkladě je i vidět přístup k jednotlivým prvkům matice a přiřazení již existujících dat k matici.

př. 2.8: Maticové počty

```
1  #include <stdio.h>
2  #include <opencv2/core/core_c.h>

4  #define vars "ABCDEF"
5  #define DIM 6
6  int main( int argc, char** argv )
7  {

9    float data[DIM*DIM]={4,3,2,1,1,2,
10                        1,2,3,4,4,3,
11                        8,1,2,8,8,2,
12                        5,3,6,9,9,6,
13                        9,8,1,2,4,5,
14                        5,6,1,2,7,8};
15   CvMat mat;
16   cvInitMatHeader(&mat,DIM,DIM,CV_32FC1,&data);
17   CvMat *res=cvCreateMat(DIM,1,CV_32FC1);
18   CvMat *right=cvCreateMat(DIM,1,CV_32FC1);
19   cvSet(right,cvScalarAll(3));

21   cvSolve(&mat,right,res);

23   printf("Matrix roots by gaussian decomp.\n");
24   for(int i=0;i<DIM;i++)
25       printf("%c = %f\n",vars[i],cvGet2D(res,i,0).val[0]) ;

27   float tmp=0;
28   for(int i=0;i<DIM;i++)
29     tmp+=data[i]*cvGet2D(res,i,0).val[0];

31   printf("Accuracy check, result must be 3\nresult=%f\n",tmp);

33   cvReleaseMat(&res);
34   cvReleaseMat(&right);

36   printf("Find Inverse matrix by gaussian decomp\nMultiply source and
         inverse matrix must be identity matrix.\n");

38   CvMat *inv=cvCreateMat(DIM,DIM,CV_32FC1);
39   res=cvCreateMat(DIM,DIM,CV_32FC1);

41   cvInvert(&mat,inv);
42   cvMatMulAdd(&mat,inv,NULL,res);

44   for(int i=0;i<DIM;i++)
45     for(int j=0;j<DIM;j++)
46       printf("%d%s",cvRound(cvGet2D(res,i,j).val[0]),((j+1)%DIM==0?"\n
             ":" "));

48   cvReleaseMat(&inv);
49   cvReleaseMat(&res);

51   return EXIT_SUCCESS;
52 }
```

Komentář

V první části programu definujeme makra pro popisky kořenů matice a počet dimenzí čtvercové matice.

Dále definujeme pole zdrojových dat, respektive jednotlivé koeficienty lineárních rovnic. Matice *mat* je inicializována pomocí funkce *cvInitMatHeader()*. Dále alokujeme matici/vektor výsledků a pravých stran pomocí *cvCreateMat()*. Vektor pravých stran nastavíme na hodnotu 3.

Funkce *cvSolve()* vyřeší lineární rovnice a výsledek uloží do matice *res*.

Dále provedeme kontrolní výpočet, zda se levá strana shoduje s pravou.

Nakonec najdeme inverzní matici *inv* matice *mat* pomocí funkce *cvInvert()* a vynásobíme je pomocí *cvMatMulAdd()*. Výsledkem je jednotková matice, která je i kontrolou správnosti výpočtu.

Funkcí *cvReleaseMat()* uvolníme alokovanou paměť.

Výstup programu

```
Matrix roots by gaussian decomp.
A = -0.480001
B = 2.568002
C = 5.848005
D = -7.808009
E = 8.816010
F = -7.744007
Accuracy check, result must be 3
result=3.000000
Find Inverse matrix by gaussian decomp
Multiply source and inverse matrix must
      be identity matrix.
1 0 0 0 0 0
0 1 0 0 0 0
0 0 1 0 0 0
0 0 0 1 0 0
0 0 0 0 1 0
0 0 0 0 0 1
```

Modul Imgproc

Po modulu *Core* je tento modul druhým nejdůležitějším. Obsahuje přes 130 funkcí zabývající se počítačovým vidění a zpracováním obrazu. Můžeme zde nalézt velmi často používané algoritmy jako „Houghova transformace“, Cannyho hranový filtr, Harrisův detektor rohů, adaptivní prahování, matematické morfologie, strukturální analýzu nebo geometrické transformace. S tímto programovým vybavením většinou není třeba psát vlastní algoritmy, protože modul *Imgproc* je komplexní řešení.

V příkladě 2.9 můžeme vidět jednoduchou implementaci aplikace Laplaceova operátoru. Na obrázcích 2.26 a 2.27 vidíme vstupní a výstupní data složitější úlohy 2.10, která implementuje detekci kůže pomocí histogramu. Díky vstupním datům je tato úloha značně zjednodušena, protože obraz je vyretušován a je na jednobarevném pozadí.

př. 2.9: Detekce hran Laplaceovým operátorem

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

3  int main( int argc, char** argv )
4  {
5      IplImage *im=cvLoadImage("edge.png",
           CV_LOAD_IMAGE_GRAYSCALE);
6      if (!im) return EXIT_FAILURE;

8      CvSize imsize=cvGetSize(im);
9      IplImage *tmp=cvCreateImage(imsize,IPL_DEPTH_8U,1);
10     IplImage *laplace=cvCreateImage(imsize,IPL_DEPTH_16S,1);

12     cvSmooth(im,tmp,CV_BLUR,3,3,0,0);
13     cvNot(tmp,im);
14     cvLaplace(im,laplace,3);

16     cvReleaseImage(&im);
17     cvReleaseImage(&tmp);
18     cvReleaseImage(&laplace);

20     return EXIT_SUCCESS;
21 }
```

Komentář

Na začátek programu načteme vstupní obraz *cvLoadImage()* a zkontrolujeme zda nenastala chyba

Dále definujeme proměnou *imsize* pomocí funkce *cvGetSize()*, která obsahuje rozměry původních dat. Poté alokujeme pomocný obraz *tmp* a výstupní obraz *laplace* funkcemi *cvCreateImage()*.

V této části rozostřujeme obraz Gaussovským filtrem *cvSmooth()* pro snížení šumu a poté jej invertujeme *cvNot()*.

Nakonec zavoláme funkce *cvLaplace()* a vypočteme gradientní obraz hran. V poslední části skriptu uvolníme paměť pomocí *cvReleaseImage()*.

obr. 2.26: Zdrojový obrázek detekce kůže

obr. 2.27: Nalezená maska kůže

př. 2.10: Detekce kůže pomocí HSV histogramu

```
1  #include "opencv2/opencv.hpp"

3  int main(int argc,char**argv)
4  {

6    IplImage *img=cvLoadImage(FILENAME);
7    if (!img) return EXIT_FAILURE;

9    CvSize imsize=cvGetSize(img);
10   IplImage* back_img=cvCreateImage(imsize ,IPL_DEPTH_8U,1);
11   IplImage* hsv=cvCreateImage(imsize,IPL_DEPTH_8U,3);
12   cvCvtColor(img,hsv,CV_BGR2HSV);

14   IplImage* h_plane=cvCreateImage(cvGetSize(img),IPL_DEPTH_8U,1);
15   IplImage* s_plane=cvCreateImage(cvGetSize(img),IPL_DEPTH_8U,1);
16   IplImage* v_plane=cvCreateImage(cvGetSize(img),IPL_DEPTH_8U,1);
17   IplImage* planes[]={h_plane,s_plane};
18   cvSplit(hsv,h_plane,s_plane,v_plane,NULL);

20   int h_bins=30,s_bins=32;
21   int  hist_size []={h_bins, s_bins };
22   float h_ranges[]={0,180};
23   float s_ranges[]={0,255};
24   float* ranges[]={h_ranges,s_ranges};
25   CvHistogram* hist=cvCreateHist(2,hist_size,CV_HIST_ARRAY,ranges,1);
26   cvCalcHist(planes,hist ,0,0) ;
27   cvNormalizeHist(hist,25*255);

29   cvCalcBackProject(planes,back_img,hist);
30   cvNormalizeHist(hist,1.0);

32   cvThreshold(back_img,back_img,65,255,CV_THRESH_BINARY);

34   IplImage *mask=cvCreateImage(imsize,IPL_DEPTH_8U,1);
35   cvZero(mask);

37   uchar * ptr1,*ptr2,*ptr3;

39   for(int i=0;i<img->height;i++)
40     for(int j=0;j<img->width;j++)
41     {
42       ptr1=cvPtr2D(v_plane,i,j);
43       ptr2=cvPtr2D(back_img,i,j);
44       ptr3=cvPtr2D(mask,i,j);

46       if (*ptr2==255)
47         if (*ptr1==*ptr2)
48           *ptr3=0;
49         else
50           *ptr3=255;
51     }

53   cvSmooth(mask,mask);
54   cvErode(mask,mask,NULL,3);
55   cvDilate(mask,mask,NULL,5);

57   cvReleaseImage(&hsv);
58   cvReleaseImage(&h_plane);cvReleaseImage(&s_plane);
59   cvReleaseImage(&v_plane);cvReleaseImage(&img);
60   cvReleaseImage(&mask);cvReleaseImage(&back_img);
61   return EXIT_SUCCESS;
62 }
```

Komentář

V počátku programu načteme zdrojová data pomoci *cvLoadImage()* a zkontrolujeme zda nenastala chyba.

Dále vytvoříme struktu *CvSize*, která obsahuje rozměry zdroje. Poté alokujeme pomocné proměnné *back_img* a *hsv* funkcemi *cvCreateImage()*. Pak konvertujeme původní data do hsv barevného prostoru funkcí *cvCvtColor()*.

Zde vytváříme pomocné obrázky, které ponesou jednotlivé složky hsv prostoru. Ty separujeme pomocí funkce *cvSplit()*. Do pole *planes* ukládáme pouze dvě složky, protože budeme vytvářet 2D H-S histogram.

V této části alokujeme parametry pro 2D histogram a určujeme rozmezí jeho os. Samotný histogram inicializujeme funkcí *cvCreateHist()*.

Funkce *cvCalcHist()* vypočte histogram H-S složek původního obrazu a následně pomocí funkce *cvNormalizeHist()* je histogram normalizován. Normalizace hodnotou $25 * 255$ je důležitá pro citlivost funkce *cvCalcBackProject()*, která se snaží procentuálně odlišit pozadí od popředí. Protože se hodnoty kůže v H-S složce příliš nemění, můžeme této vlastnosti využít.

Výsledný obraz naprahujeme *cvThreshold()* a pomocí jednoduchého algoritmu separujeme pozadí původního obrazu. K tomu nám dopomůže informace *v_plane* složka hsv prostoru. Tam kde je segmentovaná kůže v *back_img* obrazu a souhlasí s hodnotou bíle ve *v_plane* složce nastavíme hodnotu 0 a jinak 255 obrázku masky *mask*.

Následně masku rozostříme pro potlačení šumu Gaussovským filtrem *cvSmooth()* a aplikujeme morfologické operace eroze *cvErode()* a dilatace *cvDilate()*, abychom se zbavili nechtěných bodů v masce.

Na konec programu uvolníme paměť *cvReleaseImage()* všech alokovaných obrazů.

Modul Highgui

Knihovna Highgui implementuje univerzální vrstvu grafického rozhraní mezi OpenCV a platformou počítače, tak aby zdrojový kód programu byl stejný i na různých architekturách. Knihovna počítá s podporou konkretního API specifického pro danou platformu. Tím máme na mysli WinAPI pro systémy Windows, GTK pro BSD/Linux a Carbon pro Mac OS X. Od verze OpenCV 2.2 bylo implementováno univerzální API využívající knihovnu Qt za čímž stojí snaha o redukci platformě závislého kódu v knihovně. Více o implementaci knihovny Qt v OpenCV viz kapitola 2.2.1.

Základní Highgui API obsahuje pouze několik grafických widgetů (window, image a trackbar) a základní události pro jejich manipulaci. Další důležitou částí knihovny je implementace dekodérů obrazu pro načítaní obrázků a videí v různých formátech. Podporováno je i kódování tzv. *raw* obrazu do běžně používaných formátů. To je zajištěno programy specifickými pro dané platformy (FFmpeg[14], Quick Time, GStreamer, DShow, V4l a Xine). V tabulce 2.1 můžeme vidět podporované multimediální frameworky a jejich závislosti na platformách.

tab. 2.1: Seznam podporovaného multimediálního softwaru

Framework	**Podporované platformy**				
	Windows	*Mac OS X*	*BSD*	*Linux*	*Android*
DShow	•				
FFmpeg	•	•	•	•	•
Quick Time	•	•			
GStreamer	•	•	•	•	•
Xine	•	•	•	•	•
V4l				•	

OpenCV podporuje nejznámější formáty obrazu jako Portable Network Graphics (PNG), Joint Photographic Experts Group (JPEG), Sun Raster (SUNRAS), Bitmap (BMP), Tagged Image File Format (TIFF) a Portable Pix Map (PXM resp. PBM, PGM nebo PPM). Mezi formáty videa najdeme MPEG-1, MJPEG, MPEG-4.2 (MP4-2), MPEG-4.3 (DIV-3), MPEG-4 (DIV-X), H263 (U263), H263I (I263) a FLV1. Díky podpoře těchto nejznámějších kodeků není třeba pro většinu případů použít externího softwaru na konverzi videa a obrazu.

Důležitou vlastností knihovny Highgui je také bezesporu podpora sběrnic používaných v nejběžnějších kamerách (firewire, usb), proto nebývá velký problém v interakci OpenCV s běžnými zařízeními. Mezi nejdůležitější funkce tohoto modulu patří:

- *cvLoadImage()* - Načítání obrázku ze souboru.
- *cvSaveImage()* - Ukládání obrázku do souboru.
- *cvNamedWindow()* - Vytvoření uživatelského okna.
- *cvShowImage()* - Zobrazení obrázku do okna.
- *cvWaitKey()* - Reakce na uživatelský vstup nebo časový interval.

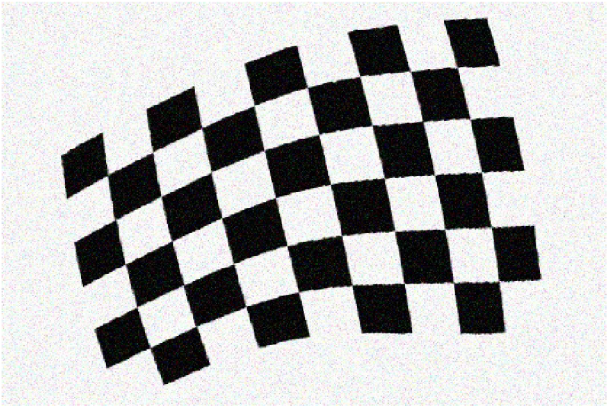
[14]FFmpeg je open source program pro zpracování obrazu vydaný pod svobodnou licencí LGPL. Jeho výhodou oproti ostatním programům používaným v OpenCV je bezesporu jeho nezávislost na architektuře. Je jej možno provozovat na všech běžných systémech.

Modul Calib3d

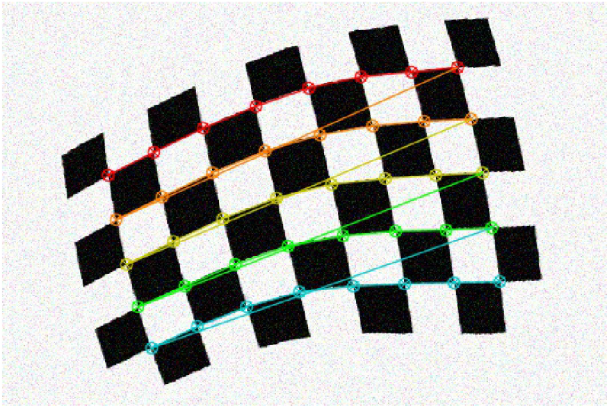
Tento modul obsahuje funkce pro kalibraci kamer a 3D rekonstrukci obrazu. Obsahuje nástroje pro měření parametrů kamery, výpočet transformační matic pro redukci zkreslení.

V příkladu 2.11 můžeme vidět program realizující detekci rohů na šachovnici. Na obrázcích 2.28 a 2.29 pak vstupní a výstupní obrazy, je vidět, že program detekoval správně rohy i při značném zkreslení a zašumění obrazu. Tento program realizuje pouze první krok v kalibraci kamer. Dalším bodem by byl výpočet parametrů kamery, respektive instrikce a distorze. Následně bychom provedli zpětnou transformaci na každý snímek obrazu.

obr. 2.28: Zdrojový obrázek kalibrace



obr. 2.29: Nalezené rohy šachovnice



př. 2.11: Detekce rohů šachovnice

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

3  #define CORN_HOR 8
4  #define CORN_VER 5

6  int main(int argc, char* argv[])
7  {
8    CvSize board_sz=cvSize(CORN_HOR,CORN_VER);
9    CvPoint2D32f corners[CORN_HOR*CORN_VER];

11   int corner_count;

13   IplImage *image=cvLoadImage("chess2.png");
14   if (!image) return EXIT_FAILURE;

16   IplImage *gray=cvCreateImage(cvGetSize(image),IPL_DEPTH_8U,1);

18   int found=cvFindChessboardCorners(image,board_sz,corners,&
         corner_count,CV_CALIB_CB_ADAPTIVE_THRESH|
         CV_CALIB_CB_FILTER_QUADS);

20   cvCvtColor(image,gray,CV_BGR2GRAY);

22   cvFindCornerSubPix(gray,corners,corner_count,cvSize( 11, 11 ),cvSize
         (-1,-1 ),cvTermCriteria(CV_TERMCRIT_EPS+
         CV_TERMCRIT_ITER,30,0.1));

24   cvDrawChessboardCorners(image,board_sz,corners,corner_count,found);
25   cvSaveImage("calib.png",image);

27   cvReleaseImage(&image);
28   cvReleaseImage(&gray);
29   return EXIT_SUCCESS;
30 }
```

Komentář

Na začátku programu definujeme počet rohů šachovnice v horizontálním *CORN_HOR* a ve vertikálním směru *CORN_VER*. Je důležité určit co nejpřesnější počet, jinak dojde k větší nepřesnosti nebo k detekci rohů.

Zde alokujeme proměnou *board_sz* typu *CvSize*, která nese informaci o počtu rohů. Dále vytváříme pole bodů, které bude uchovávat informaci o pozici detekovaných rohů, je tedy důležité alokovat dostatečně velké pole, aby obsáhlo všechny body. Proměnná *corner_count* bude nést informaci o dynamicky detekovaných rozích.

Dále načteme vstupní obrázek do proměnné *image* a kontrolujeme, zda nenastala chyba. Proměnná *gray* bude konvertovaný vstupní obraz do stupňů šedi.

Funkce *cvFindChessboardCorners()* vyhledává rohy v obraze upraveným předzpracováním dle parametrů funkce.

Zde konvertuje vstupní obraz do stupňů šedi pomocí funkce *cvCvtColor()* a následně funkcí *FindCornerSubPix()* zpřesníme polohu bodů. Tato funkce vyhledává pozici na přesnější rozměr než je pixel pomocí gradientních metod na základě kritérií daných funkcí *cvTermCriteria()*.

Nakonec body vykreslíme do vstupního obrazu funkcí *cvDrawChessboardCorners()* a uložíme do souboru *cvSaveImage()*. Poté odalokujeme načtené obrázky *cvReleaseImage()*.

Modul Features2d

Tento modul implementuje funkce a třídy pro detektory významných bodů. Zejména jde o Scale Invariant Feature Transform (SIFT), Speed Up Robust Features (SURF) nebo Maximally Stable Extremal Regions (MSER). Tento modul je napsán pouze v C++, a proto je jedním z těch, který nebude přeložen, pokud OpenCV zkompilujeme pouze pomocí překladače jazyka C.

Na obrázku 2.30 můžeme vidět výstup příkladu 2.12, který detekuje korespondující body a páruje je přímkou. Vstupními daty jsou navzájem posunuté krychle.

př. 2.12: Detekce významných bodů

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

3  int main(int argc,char**argv)
4  {
5    IplImage *image1=cvLoadImage("cube.png");
6    IplImage *image2=cvLoadImage("cube2.png");
7    if ((!image1)||(!image2)) return EXIT_FAILURE;

9    CvSize winsize=cvSize(15,15);
10   CvSize imsize=cvGetSize(image1);
11   CvTermCriteria crit=cvTermCriteria(CV_TERMCRIT_ITER|
         CV_TERMCRIT_EPS,20,0.03);
12   IplImage *gray1=cvCreateImage(imsize,IPL_DEPTH_8U,1);
13   cvCvtColor(image1,gray1,CV_BGR2GRAY);
14   IplImage *gray2=cvCreateImage(imsize,IPL_DEPTH_8U,1);
15   cvCvtColor(image2,gray2,CV_BGR2GRAY);
16   IplImage *imflow=cvCreateImage(imsize,IPL_DEPTH_8U,3);

18   IplImage *eig=cvCreateImage(imsize,IPL_DEPTH_32F,1);
19   IplImage *tmp=cvCreateImage(imsize,IPL_DEPTH_32F,1);

21   int count=500;
22   CvPoint2D32f *points1=(CvPoint2D32f*)cvAlloc(count*sizeof(
         CvPoint2D32f));
23   CvPoint2D32f *points2=(CvPoint2D32f*)cvAlloc(count*sizeof(
         CvPoint2D32f));

25   cvGoodFeaturesToTrack(gray1,eig,tmp,points1,&count
         ,0.05,5,0,3,0,0.04);
26   cvFindCornerSubPix(gray1,points1,count,winsize,cvSize(-1,-1),crit
         );

28   cvReleaseImage(&eig);cvReleaseImage(&tmp);
29   char found[500];

31   cvCalcOpticalFlowPyrLK(gray1,gray2,NULL,NULL,points1,points2,
         count,winsize,5,found,NULL,crit,0);

33   cvAnd(image1,image2,imflow);
34   CvRNG rng=cvRNG(-1);
35   for( int i=0; i<count; i++)
36   {
37     if (!found[i]) continue;
38     CvPoint p0=cvPoint(cvRound(points1[i].x),cvRound(points1[i].y))
           ;
39     CvPoint p1=cvPoint(cvRound(points2[i].x),cvRound(points2[i].y))
           ;
40     cvLine(imflow,p0,p1,CV_RGB(cvRandInt(&rng)%255,cvRandInt
           (&rng)%255,cvRandInt(&rng)%255),1);
41   }

43   cvFree(&points1);cvFree(&points2);
44   cvReleaseImage(&image1);cvReleaseImage(&gray1);
45   cvReleaseImage(&image2);cvReleaseImage(&gray2);
46   cvReleaseImage(&imflow);
47   return EXIT_SUCCESS;
48 }
```

Komentář

V první části programu se načítají zdrojové obrázky *cvLoadImage()* a kontroluje se, zda se načetly. Dále definuje velikost okna pro další algoritmy *cvSize()*, ukládáme si velikost obrázku *cvGetSize()*, definujeme kritéria pro algoritmy *cvTermCriteria()*.

Zde vytváříme kopie vstupních obrázku s tím rozdílem, že je konvertujeme do stupňů šedi pomocí *cvCvtColor()*.

Proměnná *imflow* bude obsahovat oba vstupní obrázky, do kterých se vykreslí spárované body. Obrázky *eig* a *tmp* jsou pomocné proměnné. Dále definujeme maximální počet bodů *count* na 500, poté alokujeme místo pro detekované body *points1* a *points2* pomocí *cvAlloc()*.

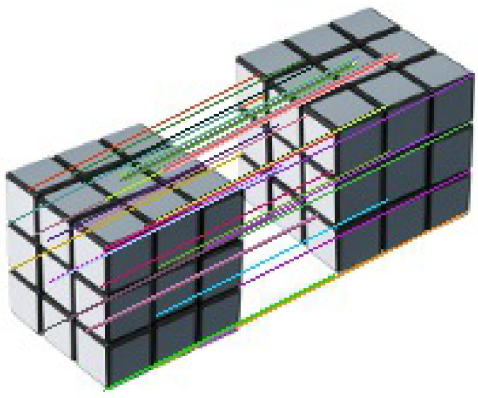
Dále detekuje vhodné body pomocí *cvGoodFeaturesToTrack()* a předem definovaných kritérií *crit*. Následně zpřesníme pozice pomocí *cvFindCornerSubPix()*.

Proměnná *found* je určena k uchování informace zda byl daný pár nalezen. Poté už můžeme vypočítat optický tok pomocí *cvCalcOpticalFlowPryLK()*. Tatu funkce najde korespondenční body k těm nalezeným v *cvGoodFeaturesToTrack()*.

Následně spojíme oba vstupní obrazy pomocí *cvAnd()* a inicializuje generátor pseudonáhodných čísel *cvRNG()*.

Nakonec pro všechny nalezené body, konvertujeme z *CvPoint2D32f* na *CvPoint* a vykreslíme přímku *cvLine()* s náhodnou barvou. Poté uvolníme alokovanou paměť *cvReleaseImage()* a *cvFree()*.

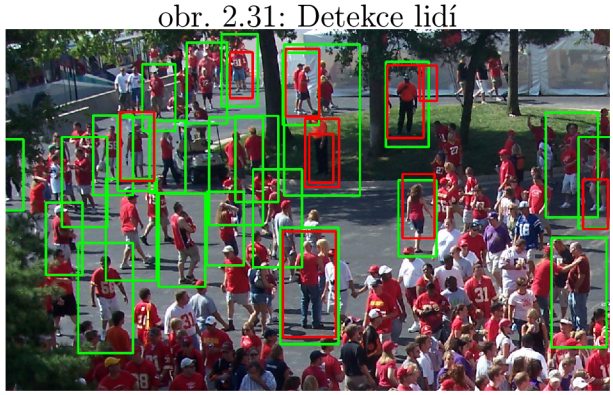
obr. 2.30: Optický tok



Modul Objdetect

Tento modul implementuje algoritmy pro detekci objektů pomocí různých metod. Najdeme zde například Haarovy kaskády, Support Vector Machine (SVM) nebo Histogram Of Oriented Gradients (HOG) detektor. Na obrázku 2.31 můžeme vidět porovnání úspěšnosti detekce pomocí HOG detektoru (zelené čtverce) a Haarových kaskád (červené čtverce). Implementace této úlohy je vidět v příkladě 2.13. HOG detektor je realizován v C++, zatímco Haar detektor v C, proto je v kódu vidět spojení obou OpenCV API.

obr. 2.31: Detekce lidí



př. 2.13: Detekce lidí pomocí Haar wavelets a HOG

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

3  using namespace std;
4  using namespace cv;
5  int main(int argc,char**argv)
6  {

8    Mat img=imread("crowd.png");
9    if (!img.data) return EXIT_FAILURE;

11   HOGDescriptor hog;
12   vector<Rect> found, filtered;

14   hog.setSVMDetector(HOGDescriptor::getDefaultPeopleDetector());
15   hog.detectMultiScale(img,found,0,Size(8,8),Size(32,32),1.05,2);

17   for(int i=0,j;i<found.size();i++)
18   {
19     for(j=0;j<found.size();j++)
20       if (j!=i && (found[i] & found[j])==found[i]) break;
21     if (j==found.size())
22       filtered.push_back(found[i]);
23   }

25   CvMat img_tmp=img;

27   for(int i=0;i<filtered.size();i++)
28   {
29     Rect rect=filtered[i];
30     rectangle(img,rect.tl(),rect.br(),CV_RGB(0,255,0),3);
31   }

33   CvHaarClassifierCascade *cascade=(CvHaarClassifierCascade*)cvLoad("
         fullbody.xml");
34   if (!cascade) return EXIT_FAILURE;
35   CvMemStorage *storage=cvCreateMemStorage();
36   CvSeq *people=cvHaarDetectObjects(&img_tmp,cascade,storage,1.1,2,0,
         Size(30,30));

38   for(int i=0;i<people->total;i++)
39   {
40     Rect *rect=(Rect*)cvGetSeqElem(people,i);
41     rectangle(img,rect->tl(),rect->br(),CV_RGB(255,0,0),3);
42   }

44   cvReleaseData(&img_tmp);
45   cvReleaseMemStorage(&storage);
46   cvReleaseHaarClassifierCascade(&cascade);
47   return EXIT_SUCCESS;
48 }
```

Komentář

Na počátku programu definujeme používané jmenné prostory, abychom nemuseli psát dlouhé názvy objektů.

Zde načítáme zdrojový obraz do objektu *Mat* a kontrolujeme zda nenastala chyba.

Dále alokujeme objekt HOG deskriptor a pomocné dynamické datové typy *vector* knihovny stl. Funkcí *hog.setSVMDetector()* aplikujeme naučenou bázi znalostí pro detekci lidí statickou funkcí *hog.getDefaultPeopleDetector()* a spustíme samotnou detekci *hog.detectMultiScale()*. Ve *for* cyklu filtrujeme nalezené duplicity a ukládáme do pomocného vektoru *filtered*.

Zde vytváříme kopii objektu *img* pomocí konverzního operátoru do struktury *CvMat* pro kompatibilitu s Haarovou kaskádou.

Pro všechny nalezené prvky HOG deskriptoru vykreslíme zelené čtverce do obrazu.

Funkcí *cvLoad()* načteme bázi znalostí pro Haarovu kaskádu z xml souboru a zkontrolujeme zda nenastala chyba. Dále alokujeme paměť pro Haarovu kaskádu *cvCreateMemStorage()* a spustíme detekci pomocí *cvHaarDetectObjects()*.

Pro všechny nalezené prvky Haar detektoru vykreslíme červené čtverce do obrazu.

Nakonec uvolníme paměť objektů z OpenCV C API pomocí release funkcí *cvReleaseData()*, *cvReleaseMemStorage()* a *cvReleaseHaarClassifierCascade()*.

Modul Ml

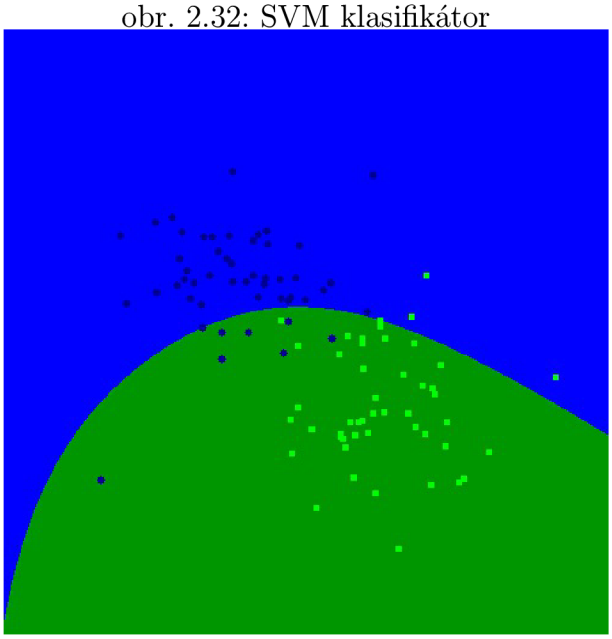
V knihovně OpenCV nalezneme nejběžnější algoritmy používaných ve strojovém učení. Modul Ml je téměř celý implementován v C++ a je navržen tak, aby všechny třídy měly společného předka *CvStatModel*. Tím je zajištěna jistá míra kompatibility mezi třídami. Mezi implementovanými algoritmy najdeme zejména podpůrné vektory (SVM), rozhodovací stromy[15] (Decision trees), boosting algoritmy[16] nebo neuronové sítě [17](Artificial Neural Network).

SVM patří do tzv. jádrových metod analýzy vzorků. Tato metoda se vyznačuje efektivitou nalezení lineárních i nelineárních hranic mezi skupinami. Principem je převod vstupního prostoru do více-rozměrného prostoru, kde lze nalézt vhodnou lineární funkci na třídění vzorků.

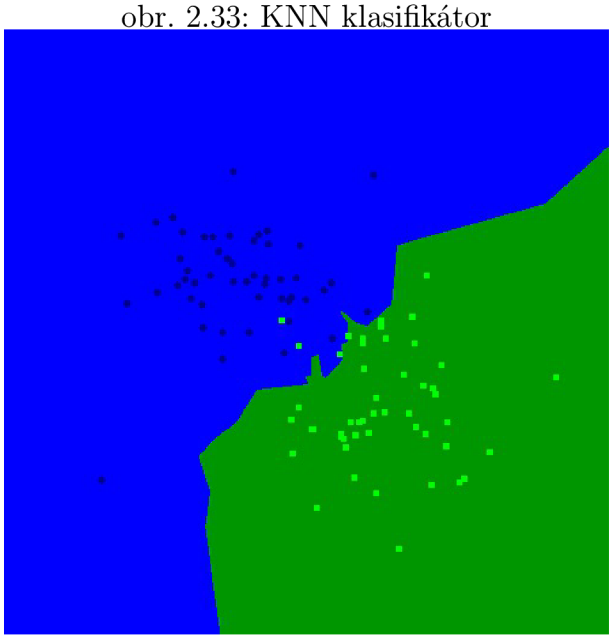
Klasifikace nejbližších sousedů KMeans je algoritmus pro třídění vzorků. Ten sbírá všechny trénovací vzorky a predikuje odezvu pro nový vzorek, pomocí daného počtu K nejbližších sousedů. Jedná se o často používanou metodu.

Na obrázcích 2.32 a 2.33 vidíme výsledky příkladu 2.14. Konkrétně jde o výstup z SVM klasifikátoru, který je geometricky vyjádřen polygonem a o KNN klasifikátor.

obr. 2.32: SVM klasifikátor



obr. 2.33: KNN klasifikátor



[15]Rozhodovací stromy patří mezi slabé klasifikátory. Používají se jako podpůrný nástroj, jehož základem jsou dynamické datové typy. Ten realizuje model rozhodnutí a jich následků. Obsahují tři základní typy uzlů, rozhodovací uzel, pravděpodobnostní uzel a koncový uzel.

[16]Boosting algoritmy jsou postavené na předpokladu, že z několika slabých klasifikátorů můžeme udělat jeden silný, který má větší úspěšnost, než několik slabších. Silný klasifikátor lze matematicky definovat jako lineární kombinací slabých klasifikátorů.

[17]Umělé neuronové sítě jsou nástrojem na paralelní zpracování dat. Skládají se z umělých neuronů, které jsou specifické tím, že mají pouze jeden vstup mnoho výstupů. Tato metoda je jedna z nejstarších používaných v umělé inteligenci.

př. 2.14: KNN a SVM klasifikátor

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

3  int K=2,sample_count=100,response;
4  CvSize imsize=cvSize(500,500);
5  CvMat trainData1,trainData2,trainClasses1,trainClasses2;

7  void drawImage(IplImage *image,void* classifier,bool knn=true)
8  {
9    CvMat *sample=cvCreateMat(1,2,CV_32FC1);

11   for(int i=0;i<imsize.height;i++)
12     for(int j=0;j<imsize.width;j++)
13     {
14       sample->data.fl[0]=(float)j;
15       sample->data.fl[1]=(float)i;
16       if (knn)
17         response=((CvKNearest*)classifier)->find_nearest(sample,K,0,0,
               NULL,0);
18       else
19         response=((CvSVM*)classifier)->predict(sample);
20       cvSet2D(image,i,j,(response==1?CV_RGB(0,0,255):CV_RGB(0,150,0)
             ));
21     }
22   for(int i =0;i<sample_count/2;i++)
23   {
24     CvPoint pt;
25     pt.x=cvRound(trainData1.data.fl[i*2]);
26     pt.y=cvRound(trainData1.data.fl[i*2+1]);
27     cvCircle(image,pt,3,CV_RGB(0,0,150),CV_FILLED);

29     pt.x=cvRound(trainData2.data.fl[i*2]);
30     pt.y=cvRound(trainData2.data.fl[i*2+1]);
31     cvRectangle(image,cvPoint(pt.x-2,pt.y-2),cvPoint(pt.x+2,pt.y+2),
           CV_RGB(0,255,0),CV_FILLED);
32   }
33   cvReleaseMat(&sample);
34 }

36 int main(int arc,char**argv)
37 {
38   CvRNG rng=cvRNG(-1);

40   CvMat* trainData=cvCreateMat(sample_count,2,CV_32FC1 );
41   CvMat* trainClasses=cvCreateMat(sample_count,1,CV_32FC1 );
42   IplImage* img_knn=cvCreateImage(imsize,IPL_DEPTH_8U,3);
43   IplImage* img_svm=cvCreateImage(imsize,IPL_DEPTH_8U,3);

45   cvGetRows(trainData,&trainData1,0,sample_count/2);
46   cvRandArr(&rng,&trainData1,CV_RAND_NORMAL,cvScalar(200,200),
         cvScalar(50,50));
47   cvGetRows(trainData,&trainData2,sample_count/2,sample_count);
48   cvRandArr(&rng,&trainData2,CV_RAND_NORMAL,cvScalar(300,300),
         cvScalar(50,50));
49   cvGetRows(trainClasses,&trainClasses1,0,sample_count/2);
50   cvSet(&trainClasses1,cvScalar(1));
51   cvGetRows(trainClasses,&trainClasses2,sample_count/2,sample_count);
52   cvSet(&trainClasses2,cvScalar(2) );

54   CvSVMParams params(CvSVM::C_SVC,CvSVM::POLY,5,1,0,5,0,0,NULL
         ,cvTermCriteria(CV_TERMCRIT_ITER,sample_count,0.001));
55   CvSVM svm(trainData,trainClasses,NULL,NULL,params);

57   CvKNearest knn(trainData,trainClasses,0,0,K);

59   drawImage(img_knn,(void*)&knn,true);
60   drawImage(img_svm,(void*)&svm,false);

62   cvReleaseImage(&img_knn);cvReleaseImage(&img_svm);
63   cvReleaseMat(&trainClasses);cvReleaseMat(&trainData);
64   return EXIT_SUCCESS;
65 }
```

Komentář

V začátku programu definujeme globální proměnné pro generování testovacích vzorků a počet sousedů K pro *KMeans*.

Tato funkce realizuje vykreslování klasifikovaných pixelů pro všechny body obrazu. Proměnná *sample* obsahuje informaci o pozici aktuálního bodu. Poté se dle typu klasifikátoru *classifier* zavolá funkce pro zjištění příslušnosti bodu ke skupině. Jde o funkce *knn.find_nearest()* a *svm.predict()*. Poté funkcí *cvSet2D()* zapíšeme barevné hodnoty do výstupního obrázku.

Dále vykreslíme jednotlivé trénovací body konvertované z matic *trainData1/2* a přiřadíme jim symbol kruhu *cvCircle()* nebo čtverce *cvRectangle()*.

Zde inicializuje generátor pseudonáhodných čísel *cvRNG()*. Poté alokujeme matice pro trénovací data *trainData* a výstupní obrazy *img_svm*/*img_knn*.

Funkcemi *cvGetRows()* si vytvoříme referenci na část pole *trainData*, které následně vyplníme náhodně generovanými body pomocí funkce *cvRandArr()*. Toto provedeme pro obě skupiny trénovací dat. Poté přiřadíme identifikační hodnoty trénovacím skupinám *cvSet()*.

Zde vytváříme třídu *CvSVMParams*, pomocí které nastavujeme *SVM* klasifikátor. Samotné vytvoření klasifikátoru a natrénování provedeme v konstruktoru třídy *CvSVM*.

Konstruktor třídy *CvKNearest* plní stejnou úlohu jako u klasifikátoru svm výše. Pokud zadáme patřičné parametry, tak se klasifikátor při vytvoření rovnou natrénuje.

Nakonec vykreslíme výstupní data pomocí funkce *drawImage()*, kterou jsme definovali výše. Poté už jenom odalokujeme data.

2.2.4. Využití OpenCV v kurzu MPOV

V této kapitole jsou vypracovány referenční úlohy kurzu MPOV pomocí knihovny OpenCV. Tyto úlohy kopírují dílčí zadání z kapitoly 2.1.2 se snahou o co nejpodobnější funkcionalitu, aby bylo možno obě vypracování navzájem porovnat.

Úloha 4

Zadání z kapitoly 2.1.2 klade důraz na vypracování algoritmů prahování. Prahování s pevným prahem můžeme vidět v příkladu 2.15 a funkci pro adaptivní prahování v příkladu 2.16. V rámci zadání bylo naprogramovat své algoritmy prahování, a proto v příkladu nebyly použity optimalizované knihovní funkce *cvThreshold()* a *cvAdaptiveThreshold()*.

př. 2.15: Prahování obrazu

```
46 int main(int argc,char**argv)
47 {
48   IplImage *img=cvLoadImage((argc>1?argv[1]:"lena.png"),
         CV_LOAD_IMAGE_GRAYSCALE);
49   if (!img) return EXIT_FAILURE;

51   IplImage *thresh=cvCloneImage(img);

53   for(int i=0;i<thresh->width*thresh->height;i++)
54     if (cvGet1D(thresh,i).val[0]>100)
55       cvSet1D(thresh,i,cvScalar(255));
56     else
57       cvSet1D(thresh,i,cvScalar(0));

59   cvShowImage("1_pevny_prah",thresh);

61   cvSet(thresh,cvScalar(127));

63   for(int i=0;i<thresh->width*thresh->height;i++)
64   {
65     uchar val=cvRound(cvGet1D(img,i).val[0]);
66     if (val>150)
67       cvSet1D(thresh,i,cvScalar(255));
68     else if (val<100)
69       cvSet1D(thresh,i,cvScalar(0));
70   }

72   cvShowImage("2_pevne_prahy",thresh);

74   adaptiveThresh(img,thresh,1);
75   cvShowImage("Adaptivni_pevny_prah_1",thresh);

77   adaptiveThresh(img,thresh,8);
78   cvShowImage("Adaptivni_pevny_prah_8",thresh);
79   cvWaitKey();

81   cvReleaseImage(&img);cvReleaseImage(&thresh);
82   cvDestroyAllWindows();
83   return EXIT_SUCCESS;
84 }
```

Komentář

V první části programu načítáme vstupní data *cvLoadImage()* ve stupních šedi *CV_LOAD_IMAGE_GRAYSCALE* a kontrolujeme, zda nenastala chyba. Poté si alokujeme data pro ukládání výsledků *cvCloneImage()*

Tento jednoduchý for cyklus implementuje prahování s jedním prahem. Využívá přístupu k matici jako k vektoru *cvGet1D()*, a tím si ušetříme jeden for cyklus pro druhou souřadnici. Funkce *cvGet1D()* vrací type *CvScalar*, a proto se omezíme ve výsledku pouze pro první kanál. Na základě stanoveného prahu zapíšeme novou hodnotu *cvSet1D()* do výstupního pole *thresh*. A výsledek vykreslíme *cvShowImage()*.

Algoritmus prahování s dvěma pevnými prahy je naprosto stejný až na to, že výstupní pole předem celé naplníme jednou výstupní hodnotu *cvSet()*. Poté už jen vymezíme ve for cyklu interval hodnot na který se nemá vůbec zapisovat. Tím vznikne výstupní obraz se třemi barvami.

Zbytek příkazů volá funkci *adaptiveThreshold()* pro adaptivní prahování v určitém regionu obrazu popsanou v příkladě 2.16.

Nakonec uvolníme data *cvReleaseImage()* a zavřeme uživatelská okna *cvDestroyAllWindows()*.

př. 2.16: Adaptivní prahování obrazu

```
1  #include "opencv2/opencv.hpp"

3  void adaptiveThresh(IplImage *src,IplImage *dst,unsigned int subs)
4  {
5    CV_DbgAssert((src->width!=dst->width)||(src->height!=dst->
         height))

7    CvSize sub_size=cvSize(src->width/subs,src->height/subs);
8    CvMat *src_header=cvCreateMatHeader(sub_size.width,sub_size.
         height,CV_8UC1);
9    CvMat *dst_header=cvCreateMatHeader(sub_size.width,sub_size.
         height,CV_8UC1);

11   int  hist_size []={255},index=0;
12   float ranges[]={0,255},*hist_ranges[]={ranges},deriv[ hist_size
         [0]-1], deriv2[ hist_size [0]-1], max=0;
13   CvHistogram* hist=cvCreateHist(1,hist_size,CV_HIST_ARRAY,
         hist_ranges,1);

15   for(int k=0;k<subs;k++,max=0,index=0)
16     for(int j=0;j<subs;j++,max=0,index=0)
17     {
18     CvRect rect=cvRect(k*sub_size.width,j*sub_size.height,sub_size.
           width,sub_size.height);
19     src_header=cvGetSubRect(src,src_header,rect);
20     dst_header=cvGetSubRect(dst,dst_header,rect);

22     cvCalcHist((IplImage**)&src_header,hist,0,0);

24     for(int i=0;i<hist_size[0]-1;i++)
25       deriv[i]=cvGetReal1D(hist->bins,i+1)-cvGetReal1D(hist->
             bins,i);

27     for(int i=0;i<hist_size[0]-1;i++)
28     {
29       deriv2[i]=deriv[i+1]-deriv[i];
30       if (deriv2[i]>max)
31       {
32         max=deriv2[i];
33         index=i;
34       }
35     }

37     for(int i=0;i<dst_header->width*dst_header->height;i++)
38       if (cvGet1D(src_header,i).val[0]>index)
39         cvSet1D(dst_header,i,cvScalar(255));
40       else
41         cvSet1D(dst_header,i,cvScalar(0));
42     }
43   cvReleaseHist(&hist);
44 }
```

Komentář

Tato funkce realizuje algoritmus adaptivního prahování s ohledem na prahy určené v několika regionech. Počet regionů je dán vstupním parametrem *subs*, který označuje počet v jedné ose. Například při hodnotě $sub = 3$ bude celkový počet regionů 9. Na začátku funkce je ukázáno použití knihovního makra *CV_DbgAssert()* ke kontrole shody parametrů vstupního a výstupního obrazu.

Dále definujeme velikost jednoho regionu strukturou *CvSize* a vytvoříme si hlavičky matic *cvCreateMatHeader()* s velikostí jednoho regionu.

Zde si alokujeme proměnné důležité pro funkci histogramu *CvHistogram*. Jde zejména o počet binů v ose x *hist_size*, rozsah osy y *ranges*. Dále si zde alokujeme pole pro výpočet první *deriv* derivace a druhé *deriv2* derivace histogramu. Funkcí *cvCreateHist()* pak inicializujeme 1D uniformní histogram *hist*.

V těchto dvou for cyklech vypočítáváme vhodný práh pro každý region zvlášť pomocí histogramy *cvCalcHist()*. Nejdříve ale vytvoříme ukazatele na správnou část obrazu funkcí *cvGetSubRect()*.

Poté vypočítáme pro každý pár bodů histogramu diferenci a uložíme ji do pole *deriv*. Samotné hodnoty histogramu extrahujeme pomocí funkce *cvGetReal1D()*.

Dále hledáme druhou derivaci a její maxima, kde nás zajímá jeho index. Tento *index* představuje současně i vhodný práh pro danou oblast.

Následným for cyklem aplikujeme algoritmus prahování s jedním prahem. Nakonec uvolníme paměť histogramu *cvReleaseHist()*.

Úloha 6

V této kapitole vidíme vypracování úlohy 2.1.2, která se zabývá diskrétní konvolucí a gradientními operátory prvního a druhého řádu. Příklad 2.17 implementuje diskrétní konvoluci a v příkladu 2.18 vidíme její použití spolu s knihovními funkcemi konvoluce a detekce hran.

př. 2.17: Diskrétní konvoluce matic

```
1  #include "opencv2/opencv.hpp"

3  IplImage* convolution(IplImage*img,CvMat *kernel)
4  {
5    CvMat *rot_mat=cvCreateMat(2,3,CV_32FC1);
6    CvMat *tmp=cvCloneMat(kernel);

8    cv2DRotationMatrix(cvPoint2D32f(kernel->width/2,kernel->
         height/2),-180,1,rot_mat);
9    cvWarpAffine(tmp,kernel,rot_mat);

11   cvReleaseMat(&tmp);
12   cvReleaseMat(&rot_mat);

14   int g[]={cvRound(kernel->width/2+1),cvRound(kernel->height
         /2+1)};

16   IplImage *ret=cvCreateImage(cvSize(img->width+kernel->width
         -1,img->height+kernel->height-1),img->depth,img->
         nChannels);
17   IplImage *result=cvCloneImage(img);
18   cvZero(result);
19   CvMat *kern_t=cvCreateMat(kernel->width,kernel->height,kernel
         ->type);

21   tmp=cvCreateMat(kernel->width,kernel->height,kernel->type);
22   cvZero(tmp);
23   for(int j=0;j<img->nChannels;j++)
24   {
25     cvZero(ret);
26     cvSetImageCOI(img,j);
27     cvSetImageCOI(result,j);

29     for(int k=g[0];k<g[0]+img->width-1;k++)
30       for(int l=g[1];l<g[1]+img->height-1;l++)
31         cvSet2D(ret,k,l,cvGet2D(img,k-g[0],l-g[1]));

33     for(int k=0;k<img->width;k++)
34       for(int l=0;l<img->height;l++)
35       {
36         for(int a=k;a<k+kernel->width;a++)
37           for(int b=l;b<l+kernel->height;b++)
38             cvSet2D(kern_t,a-k,b-l,cvGet2D(ret,a,b));

40         cvMul(kern_t,kernel,tmp);
41         cvSet2D(result,k,l,cvSum(tmp));
42       }
43   }
44   cvReleaseImage(&ret);
45   cvReleaseMat(&kern_t);
46   cvReleaseMat(&tmp);

48   cvSetImageCOI(img,0);
49   cvSetImageCOI(result,0);

51   return result;
52 }
```

Komentář

V tomto programu je implementována funkce realizující diskrétní konvoluci dvou obrazu *img* a konvolučního jádra *kernel*. Pro potřeby konvoluce je nutné jádro rotovat o 180 stupňů. K tomu je využita afiní transformace *cvWarpAffine()*, která jádro konvertuje pomocí rotační matice vytvořené funkcí *cv2DRotationMatrix()*. Pro účely tohoto algoritmu je nutné alokovat pomocná pole *rot_mat* a *tmp*. Po výpočtu rotace již není potřeba tato pole držet v paměti, a proto je uvolníme *cvReleaseMat()*.

Zde definujeme pomocná pole *ret*, *kern_t* a výstupní obraz *result* funkce *cvCreateImage()* a *cvCloneImage()*.

Pro všechny kanály vstupního obrazu vypočteme konvoluci zvlášť ve for cyklu a nastavíme aktivní kanál funkcí *cvSetImageCOI()*. Dále zkopírujeme *cvSet2D()* a *cvGet2D()* vstupní pole *img* do pomocného pole *ret*, které je větší, aby pojmulo i okraje konvolučního jádra.

V těchto cyklech dále kopírujeme oblasti *cvSet2D()* a *cvGet2D()* vstupního obrazu, tak abychom je mohli vynásobit s konvolučním jádrem *cvMul()*. Poté do výstupního pole *result* zapíšeme *cvSet2D()* sumu vynásobeného pole *cvSum()*.

Nakonec uvolníme paměť *cvReleaseMat()* a resetujeme preferovaný kanál *cvSetImageCOI()*.

př. 2.18: Gradientní operátory

```
54 #define MASK_SIZE 3
55 #define KERNEL_COUNT 5
56 int main(int argc,char**argv)
57 {
58   IplImage *img=cvLoadImage((argc>1?argv[1]:"lena.png"),
         CV_LOAD_IMAGE_GRAYSCALE);
59   if (!img) return EXIT_FAILURE;

61   float data[KERNEL_COUNT][MASK_SIZE*MASK_SIZE]=
62                   {{-1,0,1,   -1,0,1,   -1,0,1},
63                    {1,0,-1,   2,0,-2,   1,0,-1},
64                    {1,1,-1,   1,-2,-1,  1,1,-1},
65                    {3,3,-5,   3,0,-5,   3,3,-5},
66                    {0,1,0,    1,-4,1,   0,1,0}};

68   const char *titles[KERNEL_COUNT]={"Prewitt","Sobel","
         Robinson","Kirsch","Laplace"};
69   const char *titles2[KERNEL_COUNT]={"Prewitt_cvFilter2D","
         Sobel_cvFilter2D","Robinson_cvFilter2D","Kirsch_cvFilter2D"
         ,"Laplace_cvFilter2D"};

71   CvMat kernel;
72   IplImage *out;

74   for(int i=0;i<KERNEL_COUNT;i++)
75   {
76     kernel=cvMat(MASK_SIZE,MASK_SIZE,CV_32FC1,data[i]);

78     out=convolution(img,&kernel);
79     cvShowImage(titles[i], out);

81     cvZero(out);

83     cvFilter2D(img,out,&kernel);
84     cvShowImage(titles2[i], out);

86     cvReleaseImage(&out);
87   }

89   out=cvCreateImage(cvSize(img->width,img->height),
         IPL_DEPTH_32F,1);
90   cvLaplace(img,out);
91   cvShowImage("Laplace_cvLaplace",out);

93   cvSobel(img,out,1,1);
94   cvShowImage("Sobel_cvSobel",out);

96   cvReleaseImage(&out);

98   out=cvCreateImage(cvSize(img->width,img->height),
         IPL_DEPTH_8U,1);
99   cvCanny(img,out,100,150);
100  cvShowImage("Canny_cvCanny",out);

102  cvWaitKey();
103  cvReleaseImage(&img);
104  cvReleaseImage(&out);
105  cvDestroyAllWindows();
106  return EXIT_SUCCESS;
107 }
```

Komentář

Na začátku programu načítáme zdrojová data *cvLoadImage()* a kontrolujeme, zda nenastala chyba.

V těchto polích definujeme konkrétní gradientní operátory, pro další použití v programu. Každý operátor je v jednom poli typu *float*.

K vytvoření oken pro vykreslování výstupních obrazů je nutno nadefinovat jejich popisky. K tomu slouží pole *titles* a *titles2*.

Zde vytváříme proměnné *out* a *kernel*, které budou naplněny vstupními a výstupními daty.

V tomto for cyklu je implementován výpočet jednotlivých výstupních obrazů. Nejdříve musíme do matice *kernel* uložit konkrétní konvoluční jádro z pole *data* a zavolat funkci *convolution*. Výsledek konvoluce vykreslíme do okna s popiskem daným polem *titles*.
Jakmile je výpočet hotov, použijeme knihovní funkci *cvFilter2D()* pro porovnání úspěšnosti s námi implementovanou funkcí. Nakonec uvolníme paměť alokovanou funkcí *convolution* pomocí *cvReleaseImage()*.

Pro výpočet Laplaceova a Sobelova operátoru musíme redefinovat výstupní proměnnou *out*, protože tyto funkce předpokládají, že obraz bude mít data uložena jako typ *float*. Poté zavoláme funkce *cvLaplace()* a *cvSobel()* a výsledek vykreslíme do oken *cvShowImage()*.

Posledním bodem zadání je vyzkoušení pokročilejších detektorů hran, tím je v našem případě Cannyho detektor implementovaný funkcí *cvCanny()*.

Nakonec po vykreslení všech výsledků uvolníme paměť *cvReleaseImage()* a okna *cvDestroyAllWindows()*.

Úloha 10

V této kapitole je implementováno poslední zadání z kapitoly 2.1.2. V příkladu 2.19 vidíme realizované algoritmy dilatace a použití knihovních funkcí morfologických operací.

př. 2.19: Morfologické operace

```
1  #include "opencv2/opencv.hpp"

3  void dilatation(IplImage *img)
4  {
5    if (img->nChannels!=1)
6      cvCvtColor(img,img,CV_BGR2GRAY);

8    cvThreshold(img,img,120,255,CV_THRESH_BINARY);

10   for(int i=0;i<img->width;i++)
11     for(int j=0;j<img->height;j++)
12       if (cvGet2D(img,i,j).val[0]==255)
13       {
14         if (i!=0)
15           cvSet2D(img,i-1,j,cvScalar(255));
16         else
17           if (i!=img->width-1)
18             cvSet2D(img,i+1,j,cvScalar(255));
19         cvSet2D(img,i,j,cvScalar(255));
20       }
21 }

23 int main(int argc,char**argv)
24 {

26   IplImage *img=cvLoadImage("biohazard.png",
         CV_LOAD_IMAGE_GRAYSCALE);
27   if (!img) return EXIT_FAILURE;
28   IplImage *copy=cvCloneImage(img);

30   dilatation(copy);
31   cvShowImage("dilatation",copy);

33   IplConvKernel *se=cvCreateStructuringElementEx(3,1,1,0,
         CV_SHAPE_RECT);
34   cvDilate(img,copy,se);
35   cvShowImage("Dilate",copy);
36   cvErode(img,copy,se,3);
37   cvShowImage("Erode",copy);

39   cvErode(img,copy,se);
40   cvDilate(copy,copy,se);
41   cvShowImage("Erode&Dilate",copy);
42   cvMorphologyEx(img,copy,NULL,se,CV_MOP_OPEN);
43   cvShowImage("Open",copy);

45   cvDilate(img,copy,se);
46   cvErode(copy,copy,se);
47   cvShowImage("Dilate&Erode",copy);
48   cvMorphologyEx(img,copy,NULL,se,CV_MOP_CLOSE);
49   cvShowImage("Close",copy);

51   cvWaitKey();

53   cvReleaseImage(&img);
54   cvReleaseImage(&copy);
55   cvReleaseStructuringElement(&se);
56   cvDestroyAllWindows();
57   return EXIT_SUCCESS;
58 }
```

Komentář

Na začátku programu je implementována funkce pro matematickou morfologii dilatace *dilatation* strukturním elementem obdélníku o rozměrech stran $3x1px$. Ta počítá s binárními daty, respektive s obrazem, který neobsahuje více než dvě úrovně hodnot. Pro zjednodušení je vstupní obraz konvertován do stupňů šedí *cvCvtColor()* a výsledek je naprahován *cvThreshold()*.

Poté již pro všechny body obrazu vypočítáváme výslednou transformaci. Pro body, které mají hodnotu bíle *cvScalar(255)* kontrolujeme, zda to nejsou krajní body a pokud ne, nastavíme okolním bodům v horizontálním směru bílou barvu.

V hlavní funkci programu nejprve načítáme zdrojový obrázek ve stupních šedi *cvLoadImage()* a vytváříme si pomocnou kopii *cvCloneImage()*.

Zde voláme funkci *dilatation()* a výsledek dilatace zobrazíme v okne *cvShowImage()*.

Pro vytvoření strukturního elementu ve tvaru dle zadání je třeba použít funkci *cvCreateStructuringElementEx()* s přednastaveným tvarem *CV_SHAPE_RECT*. Tento strukturní element je poté využíván knihovními funkcemi pro morfologické operace dilatece, eroze, otevření a uzavření.

Zde počítáme operace dilatace *cvDilate* a eroze *cvErode()*. Výsledky jsou poté vykresleny do okna *cvShowImage()*.

V této části kontrolujeme zda po sobě jdoucí eroze a dilatace tvoří operaci otevření *cvMorphologyEx()*. Otevření je definováno parametrem *CV_MOP_OPEN*.

Stejně jako v předchozí části kontrolujeme po sobě jdoucí dilataci a erozi s uzavřením. K uzavření opět slouží funkce *cvMorphologyEx()*, ale tentokrát s parametrem *CV_MOP_CLOSE*.

Nakonec uvolníme paměť *cvReleaseImage()* a zavřeme uživatelská okna *cvDestroyAllWindows()*.

2.3. Jazyk Lua

Tento skriptovací jazyk vznikl v roce 1993 na univerzitě PUC v Brazílii jako akademický projekt a byl vydán pod MIT licencí. Během následujících let se prosadil nejvíce, jako podpůrný mechanismus pro grafické aplikace, pro svou malou velikost a vysokou rychlost. To potvrzuje množství vývojářů her[18], jenž jazyk Lua využily ve svých projektech jako skriptovací platformu zejména pro zpracování grafických operací. Hlavními rysy jazyka Lua jsou:

- *Rychlost vykonávání instrukcí*
- *Malá velikost binárních souborů*
- *Malá paměťová náročnost*
- *Implementace v ANSI C*
- *Registrový virtuální stroj (VM)*
- *Inkrementální garbage collector (GC)*
- *Externí kompilátor*[19]

Knihovna jazyka Lua implementuje pouze základní mechanismy z C knihoven *stdio.h*, *stdlib.h* a *math.h*. Obsáhlejší funkcionalita je implementována v řadě uživatelských modulů, jako například *socket* nebo *lfs*. Tyto moduly mohou být i Lua skripty, ale zpravidla jsou to přemostění (wrapper) C/C++ knihoven, tak jako LuaCV.

2.3.1. Rozdíly mezi verzemi 5.1 a 5.2

Verze jazyka 5.1 byla vydána roku 2006 a odstupem času byly přidávány pouze opravné balíčky. Od té doby nebyl jazyk nikterak rozšířen. Nová stabilní verze *5.2* byla vydána 16. 12. 2011, tato verze byla ve vývoji přes pět let a přidala některé významné změny. Hlavními změnami jsou:

- *Bit32* - Oficiální podpora bitových operací pomocí tabulky *bit32*.
- *Slabé tabulky* - *Weak tables* se liší od regulérních tabulek tím, že obsahují tzv. *weak references*. Tyto reference jsou pak garbage collectorem vynechávány.
- *Lehké C funkce* - Nový datový typ, prakticky jde o pouhý ukazatel na C funkci bez uživatelských dat.
- *Goto* - Podpora goto výroku pro skok na návěští.

[18]Mezi nejznámější studia, která využila jazyk Lua pro své hry můžeme zařadit Valve (Half-Life), Blizzard (World of Warcraft) nebo Black Isle (Planescape-Torment,Baldur's Gate I a II, IceWind Dale I a II).

[19]Pomocí externího kompilátoru lze libovolný lua skript zkompilovat do binárního souboru. Tento soubor tzv. *binary chunk* se vyznačuje rychlejším zpracováním instrukcí a možnosti skrýt obsah skriptu před jinými uživateli.

- *Generativni režim GC* - Možnost změny priorit garbage collectoru se zaměřením na mladé proměnné.
- *Odebrání tabulky prostředí* - Nyní mohou tzv. *Environment table* vlastnit pouze funkce, a proto byly odstraněny C API funkce *lua_setenv()* a *lua_getenv()*. Do budoucna se počítalo s využitím environmentální tabulky pro ukládání počtu referencí objektů v LuaCV, ale s touto změnou se bude muset využít jiná metoda.

2.3.2. Základy syntaxe

Jazyk Lua syntakticky vychází z kombinace jazyků Modula (nepříliš známý zaniklý nástupce Pascalu) a C, proto je programátorovi jazyk od pohledu povědomý. Nicméně Lua oproti jeho vzorům implementuje, jak imperativní styl programování, tak i funkcionální. V příkladu 2.20 můžeme vidět zápis kompletní syntaxe vyjádřené ve formě *BNF* (Backus Normal Form).

př. 2.20: Syntaktická definice Lua[15]

```
chunk ::= block
block ::= {stat} [retstat]

stat ::= ';'
  varlist '=' explist |
  functioncall |
  label |
  break |
  goto Name |
  do block end |
  while exp do block end |
  repeat block until exp |
  if exp then block {elseif exp then block} [else block]
      end |
  for Name '=' exp ',' exp [',' exp] do block end |
  for namelist in explist do block end |
  function funcname funcbody |
  local function Name funcbody |
  local namelist ['=' explist]
retstat ::= return [explist] [';']

funcname ::= Name {'.' Name} [':' Name]
varlist ::= var {',' var}
var ::= Name | prefixexp '[' exp ']' | prefixexp '.' Name
namelist ::= Name {',' Name}
```

```
label ::= '::' Name '::'

explist ::= exp {',' exp}
exp ::= nil | false | true | Number | String | '...' |
      functiondef |
prefixexp | tableconstructor | exp binop exp | unop
     exp
prefixexp ::= var | functioncall | '(' exp ')'

functioncall ::= prefixexp args | prefixexp ':' Name
      args
args ::= '(' [explist] ')' | tableconstructor | String
functiondef ::= function funcbody
funcbody ::= '(' [parlist] ')' block end
parlist ::= namelist [',' '...'] | '...'

tableconstructor ::= '{' [fieldlist] '}'
fieldlist ::= field {fieldsep field} [fieldsep]
field ::= '[' exp ']' '=' exp | Name '=' exp | exp
fieldsep ::= ',' | ';'

binop ::= '+' | '-' | '*' | '/' | '^' | '%' | '..' |
'<' | '<=' | '>' | '>=' | '==' | '~=' |
and | or
unop ::= '-' | not | '#'
```

Protože syntaxe z výčtu v příkladu 2.20 nemusí být na první pohled zřejmá, na následujícím Příkladu 2.21 je ukázáno několik základních konstrukcí. Tento příklad lze napsat jednoduše, ale jeho účelem bylo ukázat základní možnosti jazyka Lua. Funkcí skriptu je implementovat morfologickou operaci eroze strukturním elementem □⊠. Ve skriptu bylo využito pouze základních knihoven jazyka.

Jak je z příkladu vidět, tak jazyk Lua umožňuje bohatý způsob zápisu a záleží jen na programátorovi, na který je zvyklý a využije ho. Syntaxe Lua obsahuje velkou míru benevolence vůči stylu zápisu, jako nepovinné středníky na konci příkazů nebo takřka neomezeně odsazovaní příkazů ve skriptu. Tímto mohou vznikat na první pohled velmi neobvyklé konstrukce, ale díky možnosti funkcionálního zápisu lze mnohé algoritmy přepsat do efektivnějších variant.

př. 2.21: Využití syntaxe Lua

```
1  #!/usr/bin/env lua

3  erode=function (matrix)
4    local mat,i=matrix,1
5    while i<=#mat do
6      for j=2,#mat do
7        if (mat[i][j]==1) and (mat[i][j-1]==1) then
8          mat[i][j-1]=0
9        end
10     end
11     i=i+1
12   end
13   return mat
14 end

16 function printMat(mat)
17   for _,tmp in pairs(mat) do
18     local str,j="",1
19     repeat
20       str=str..' '..tmp[j]
21       j=j+1
22     until #mat<j
23     print(str)
24   end
25 end

27 print("Original matrix")
28 local matrix={{0,1,1,0,1},
29                {0,0,1,1,0},
30                {0,0,1,1,1},
31                {1,0,1,1,0},
32                {1,1,1,1,1}}

34 printMat(matrix)
35 print([[----------
36 Eroded matrix]])
37 printMat(erode(matrix))
```

Komentář

Na počátku skriptu vidíme přiřazení anonymní funkce k proměnné *erode()*. Tato funkce má jeden vstupní argument matice. V těle funkce inicializujeme lokální proměnné *mat* a *i*. Zde můžeme vidět vícenásobné přiřazení. Dále v cyklu *while* iterujeme index *i* od dvou do délky tabulky definované operátorem #. Následuje vnořený cyklus for a v něm podmínka if, která realizuje strukturní element. Zde můžeme vidět přístup k 2D poli. Nakonec iterujeme index *i* a poté vracíme vypočtenou matici *mat*.
Dále vidíme definici funkce *printMat* pro vypsání matice, v ní je opět pomocí různých zápisů cyklů vidíme algoritmus pro vypsání všech položek matice. Je zde ukázán průchod matice pomocí iterátoru *pairs()* ve for cyklu. Poté inicializujeme lokální proměnné, kde *str* je řetězec. Následně v cyklu repeat/until přidáváme do proměnné *str* aktuální prvek pole. Po konci každého řádku ho vytiskneme *print()*.
V hlavním těle skriptu pak vytvoříme tabulku *matrix* jako 2D pole a vypíšeme, jak původní tak i erodovanou matici. Můžeme vidět způsob zápisu jak krátkého řetězce, tak i dlouhého.

Výstup skriptu

```
Original matrix
0 1 1 0 1
0 0 1 1 0
0 0 1 1 1
1 0 1 1 0
1 1 1 1 1
----------
Eroded matrix
0 0 1 0 1
0 0 0 1 0
0 0 0 0 1
1 0 0 1 0
0 0 0 0 1
```

V tabulce 2.2 můžeme vidět všechny základní datové typy v Lua. Je dobré poznamenat, že prakticky všechny komplexnější uživatelsky-definované typy postavené na základních typech obsahují metatabulku (metatable).

tab. 2.2: Základní datové typy

Typ	Popis	C/C++
Number	číslo s plovoucí desetinou čárkou	double
String	textový řetězec	char[]
Nil	nulová adresa	NULL
UserValue	datový objekt z C/C++	void *
Tables	hash pole/struktura	struct *
Weak references	Userdata, která GC vynechává	void *
Lua functions	čisté Lua funkce	
C functions	C/C++ funkce	(static int)(lua_State *L)
Light C function	Ukazatel na C funkci bez uživ. dat	(static int)(lua_State *L)

2.3.3. Meta tabulky

Meta tabulky slouží jako mechanizmus pro definici objektu, dědičnosti, operátorů a jiných vlastností objektového návrhu v jazyce Lua. V Lua API mohou tyto meta tabulky být přiděleny pouze jiným *tabulkám* nebo uživatelské paměti z C tzv. *uservalue*. Přidělení meta tabulky uživatelské paměti je možné pouze z Lua C API, a proto pouze z nativního kódu. Nelze tedy přetypovat jednu uživatelskou paměť za jinou pomocí meta tabulek přímo z Lua skriptu.

V tabulce 2.3.3 je vidět seznam callback funkcí, které jsou definované v API. Pro přemostění LuaCV jsou důležité ty tučně označené. Funkce *__index* je volána vždy, když se snažíme přistoupit k metodě tabulky. Funkci *__newindex*, když se snažíme vlastnosti tabulky přiřadit novou hodnotu. Obě tyto funkce jsou pouze jedny pro všechny volané nebo zapisované metody, je tedy na vnitřním algoritmu, aby vrátil požadovaný výsledek.

V Lua se meta tabulky přiřazují pomocí knihovní funkce *setmetatable()* a vybavují pomocí *getmetatable()*.

tab. 2.3: Seznam callback funkcí metatabulky

Funkce	Operátor	Popis
__index	. a :	přístup k proměnné tabulky
__newindex	=	za účelem změny hodnoty
__add	+	operace sčítání
__sub	-	operace odčítání
__mul	*	operace násobení
__div	/	operace dělení
__mod	%	zbytek po celočíselném dělení
__pow	^	operace s exponentem
__unm	-	operace jednočlenného odčítání
__concat	..	operace spojování tabulek
__len	#	získání délky tabulky
__eq	==	podmínkové - ekvivalentní
__le	<=	podmínkové - menší nebo ekvivalentní
__lt	<	podmínkové - menší
__call		volá funkci při každém načtení tabulky
__gc		volá funkci až před uvolněním paměti garbage collectorem
__tostring		volán při funkci *print()*

2.3.4. Garbage collector

Lua od verze 5.1 obsahuje tzv. inkrementální garbage collector (GC). Ten se od jiných GC liší tím, že je schopen uklízecí cyklus přerušit a následně na něj navázat. Mezi tím je možno spouštět uživatelský kód. Ve verzi 5.2 přibyl i experimentální generační GC mód. Ten se snaží efektivně uklízet objekty, které dle statistik budou nejpravděpodobněji brzy nepotřebné. V základním nastavení je GC nastaven na inkrementální mód. Vzhledem k tomu, že žádný návrh GC není dokonalý, existuje i manuální ovládání, které je v některých situacích nutno použít. Jde o funkci *collectgarbage()* a její parametr *collect*.

3. Realizovaná řešení

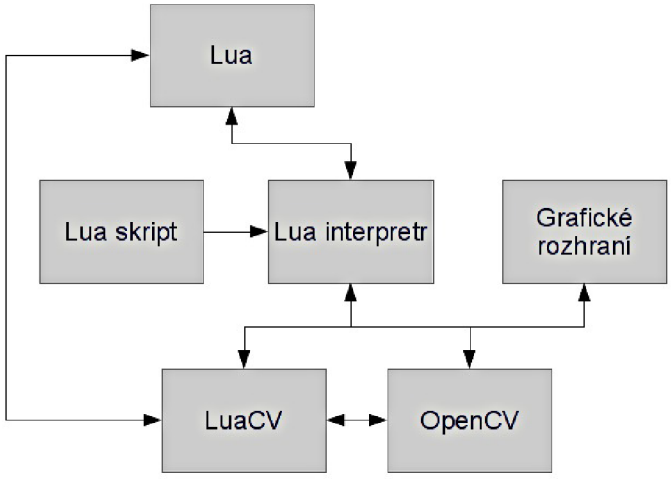
Tato část práce popisuje přemostění LuaCV a jeho mechanismy pro tvorbu multiplatformní knihovny zprostředkovávající knihovnu OpenCV do jazyka Lua. Jsou popsány jednotlivé implementované moduly a objekty s nimi spojené. Dále pak multiplatformní aplikaci pro generování pomocných souborů, projektů pro překlad této knihovny a následně vývoj instalátoru pro platformu MS Windows. Poslední část této kapitoly je věnována popisu skriptu pro generování dokumentace přímo ze zdrojového kódu.

3.1. Knihovna LuaCV

Knihovna LuaCV si klade za cíl realizovat přemostění knihovny OpenCV do jazyka Lua v jazyce C++. Jazyk Lua obsahuje základní C API pro komunikaci mezi binárními soubory a interpretrem Lua, proto je možné vytvořit přemostění mezi knihovnou OpenCV. Výsledné přemostění tak nabývá formu dynamické sdílené knihovny[1], která je dynamicky načítána interpretrem Lua, při vykonávání každého skriptu.

Na obrázku 3.1 lze vidět schématicky funkčnost knihovny LuaCV a její propojení s ostatními komponentami. Lua interpretr, který načítá skript jazyka Lua provádí instrukce na základě obsahuj skriptu. Jakmile dojde k funkci *require('luacv')* dynamicky načte knihovnu LuaCV. Knihovna LuaCV neobsahuje funkcionalitu počítačového vidění, ale využívá knihovních funkcí a objektů OpenCV knihovny. LuaCV je tedy úzce svázána s knihovnou OpenCV. Aby mohla knihovna LuaCV komunikovat s interpretrem Lua, je zapotřebí použít Lua C API definované v knihovně Lua. Grafické rozhraní je ovládáno skrze OpenCV knihovnu v modulu *highgui*, knihovna LuaCV pouze definuje sadu callback funkcí nutných pro předání parametrů knihovně OpenCV.

obr. 3.1: Schéma fungování LuaCV



Velký důraz při tvorbě toho přemostění je kladen na možnost multiplatformního využití, a tedy možnosti kompilace na nejrozšířenějších platformách. Výhodou je, že jazyk Lua je psán v ANSI C, a proto je podporován všude tam, kde je libovolný kompilátor jazyka C na rozdíl od knihovny OpenCV, která je silně vázána na podporu operačního systému[2]. Proto nároky na běh přemostění LuaCV udává převážně knihovna OpenCV.

[1] Přípony dynamické knihovny jsou podobné na všech kromě MS Windows. Na Unixových/Linuxových systémech je přípona **.so* a na MS Windows **.dll*. Bohužel přenositelnost dynamických knihoven je omezená oproti staticky linkovaným z důvodů, že v systému mohou chybět důležité knihovny nebo jejich správné verze.

[2] Knihovna OpenCV je nejvíce vázána na programy a knihovny třetích stran jako FFMpeg, GTK, Qt, libjpeg nebo libtiff. Pokud bychom OpenCV zkompilovali bez těchto závislostí, přišli bychom o velkou část klíčové funkcionality.

3.1.1. Implementace

Implementace LuaCV je realizována v jazyce C++, který je binárně kompatibilní s jazykem C. Proto je možné spolu linkovat knihovnu Lua a OpenCV. Nicméně je nutné použít příkaz *extern "C"* pro knihovnu Lua, protože jazyk C++ a C má jinak uspořádaný zásobník a tím by došlo k chybě. Celý projekt LuaCV se skládá z několika dílčích částí viz seznam níže.

- Pomocná knihovna *luacvaux* společná pro všechny moduly LuaCV.
- *Přemostěné objekty*, které jsou nezbytné pro chod jednotlivých funkcí z modulů.
- *Moduly*, které kopírují rozdělení funkcionality dle OpenCV a používají přemostěných objektů.
- *Modul LuaCV*, který je linkován se všemi moduly, dynamicky je načítá a zajišťuje tak přístup ke všem funkcím a objektům LuaCV modulů v jediné globální tabulce.

Důležitým faktem je, že knihovna LuaCV nemá žádnou funkci *main()*, ke které by se linkovala. Tato problematika je více rozvedena v následující kapitole.

Načtení sdílené knihovny

Lua načítá sdílenou knihovnu v případě, že uživatel zavolá příkaz *require()*. Ten dle parametrů dynamicky načte knihovnu podle jména na předem určených obvyklých umístěních. Proto je nutné mít v každém modulu hlavní funkci, která bude následně interpretrem zavolána. V případě modulu LuaCV, který je volán *require('luacv')* musí být ve sdílené knihovně přítomna funkce *luaopen_luacv()*. V případě platformy MS Windows tato funkce musí být zařazena do exportní tabulky sdílené knihovny. Toho docílíme makrem *__declspec(dllexport)*, u jiných platforem to není nutné.

Mechanismus dynamického načítání se liší dle platformy. V tabulce 3.1 jsou vidět obě API v závislosti na platformě.

tab. 3.1: Dynamické načítání sdílené knihovny

	POSIX	**MS Windows**
Definice API	libdl.so	kernel32.dll
Otevření knihovny	dlopen()	LoadLibrary()
Načtení obsahu	dlsym()	GetProcAddress()
Uzavření knihovny	dlclose()	FreeLibrary()

V příkladu 3.1 je vidět exportní funkce základního modulu LuaCV a v příkladu 3.2 exportní funkci modulu Imgproc. Jako všechny Lua C API funkce musí tyto exportní funkce být návratového typu *int* a mít jediný vstupní argument *lua_State *L*.

Číselný výstup interpretru říká kolik bude mít funkce výstupních argumentů, tedy kolik má interpretr vybrat ze zásobníku proměnných. Výstupem exportní funkce musí být konstanta 1 a na zásobník musí být přivedena tabulka, která bude obsahovat přístupné funkce a objekty knihovny. Od verze Lua 5.2 již tabulky modulů nejsou globální, takže pokud si při volání funkce *require()* neuložíme odkaz na tabulku, musíme ji najít v globální tabulce modulů *package.loaded.nazev_modulu*. Vstupní argument reprezentuje ukazatel na aktuální zásobník interpretru Lua.

př. 3.1: Hlavní funkce modulu LuaCV

```
36 extern "C" {
37 #if defined(WIN32) || defined(WIN64)
38  __declspec(dllexport)
39 #endif
40 int luaopen_luacv(lua_State *L)
41 {
42   objectReg(LIBNAME,luacv_m);
43   luaL_newlib(L,luacv);

45   luaL_getmetatable(L,LIBNAME);
46   lua_setmetatable(L,-2);

48   lua_pushglobaltable(L);
49   lua_pushvalue(L, -2);
50    lua_setfield (L, -2,LIBNAME);
51   lua_pop(L, 1);

53   for (luaL_Reg *lib = (luaL_Reg*)loadedlibs; lib->func; lib++)
54   {
55     luaL_requiref(L, lib->name, lib->func, 1);
56     lua_getglobal(L,LIBNAME);
57     lua_pushnil(L);
58     while(lua_next(L,-3) !=0)
59     {
60       lua_pushvalue(L, -2);
61       lua_insert(L, -2);
62       lua_settable(L, -4);
63     }
64   }
65   return 1;
66 }}
```

Komentář

příkaz *extern "C"* je direktiva pro překladač, aby správně přerovnal pořadí na zásobníku, pro kompatibilitu C a C++.
Makro *#if defined()* zaručuje přidání exportního symbolu pomocí *__declspec(dllexport)* pouze pro platformu MS Windows.
Makro *objectReg()* vytvoří meta tabulku a zaregistruje do ní její metody z pole *luacv_m*. Funkce *luaL_newlib()* vytvoří tabulku, do které se budou ukládat ukazatele na funkce a konstanty modulu.
Funkce *luaL_getmetatable()* načte meta tabulku z registru a zařadí ji na zásobník. Pote pomocí *lua_setmetatable()* přiřadíme tuto meta tabulku tabulce vytvořené pomocí *luaL_newlib()*.
Příkazy *lua_pushglobaltable()*, *lua_pushvalue()*, *lua_setfield()* a *lua_pop()* zajistíme, aby tabulka tohoto modulu byla globální.
V následujícím *for* cyklu načítáme všechny moduly z pole *loadedlibs*, kde v položce *func* je ukazatel exportní funkce těchto modulů. Funkcí *luaL_requiref()* zaregistrujeme aktuální modul a jeho tabulku je na vrchu zásobníku. Načteme globální tabulku modulu luacv a v cyklu *while* iterujeme tabulku aktuálního modulu a vytváříme mělkou kopii jejich položek funkcemi *lua_pushvalue()*, *lua_insert()* a *lua_settable()*.

př. 3.2: Hlavní funkce modulu Imgproc

```
2236 extern "C" {
2237 #if defined(WIN32) || defined(WIN64)
2238  __declspec(dllexport)
2239 #endif
2240 int luaopen_luacv_imgproc(lua_State *L)
2241 {
2242   imgprocReg()
2243   objectReg(IMGPROC_NAME,luacv_m);

2245   luaL_requiref(L,CORE_NAME,luaopen_luacv_core,1);
2246   lua_pop(L,1);

2248   luaL_newlib(L,imgproc);

2250   luacv_objTableReg(L,imgproc_object);
2251   luacv_constReg(L,imgproc_var);

2253   luaL_getmetatable(L,IMGPROC_NAME);
2254   lua_setmetatable(L,-2);
2255   return 1;
2256 }}
```

Komentář

Makrem *imgprocReg()* zaregistrujeme do registru všechny meta tabulky objektů modulu imgproc. Dále vytvoříme meta tabulku pro modul imgproc *objectReg()*, který bude mít callback funkce shodné s modulem luacv.
Funkcí *luaL_requiref()* načteme core modul, protože je nutnou závislostí modulu imgproc. Tabulku modulu core pak odstraníme ze zásobníku *lua_pop()*.

Funkce *luaL_newlib()* vytvoří tabulku modulu imgproc, do které se budou registrovat funkce a konstanty modulu.

Funkcemi *luacv_objTableReg()* a *luacv_constReg()* vložíme do tabulky modulu parametry objektů a konstanty.
Následně na tuto tabulku aplikujeme meta tabulku vytvořenou výše *luaL_getmetatable()* a *lua_setmetatable()*.

Příklad 3.2, který je popsán výše, je velmi podobný ostatním modulům. Výjimkou je modul *core*, který se liší pouze tím, že nemá žádnou závislost na ostatních modulech.

V případě nedodržení závislostí na modul *core* by nedošlo ke kritickému selhání typu „undefined symbol“[3], pouze by v Lua chyběla zásadní podpůrná funkcionalita.

[3]Chyba „undefined symbol“ se projeví pokud se snažíme načíst dynamickou knihovnu, ve které chybí potřebná funkcionalita. Zejména se projeví mezi knihovnami závislými na konkrétní verzi jiné knihovny, ve které byly pozměněny parametry nebo odebrány funkce. Také k tomuto problému může dojít pokud se snažíme linkovat programy zkompilovanáne v C a C++ bez pomocí direktivy *extern "C"*.

Kontrola typů objektů

Pro vyřešení kontroly typovosti v netypovém jazyce Lua[4] bylo nutné využit již dříve popsaných meta tabulek. Každý objekt má přiřazenou meta tabulku a ta je kontrolována při zjišťování typu uživatelské paměti.

V podstatě je využívaná knihovní funkce *luaL_checkudata()*, která při nesprávné nebo chybějící meta tabulce vyvolá chybu a konec procesu, jinak vrací ukazatel na data. V knihovně LuaCV byla navržena šablonová funkce *luacv_checkObject()*, která snáze pracuje s kontejnerem objektu[5] viz příklad 3.3 a stará se o správně přetypovávání. Funkce je vidět v příkladu 3.4.

př. 3.3: Kontejner LuaCV objektu

```
49 template <typename cvtype>
50 struct luacv_obj
51 {
52   cvtype *data;
53   bool dealloc_data;
54   int ref;
55 };
```

Komentář

Šablonová proměnná *data* je ukazatelem na originální OpenCV data.
dealloc_data je proměnná, dle které se zjišťuje, zda se mají data ručně uvolnit, nebo se o ně postará Lua GC.
Proměnná *ref* je číslo reference v registru, dle které se dá identifikovat instance dat.

př. 3.4: Kontrola typu objektu

```
155 template<typename cvtype>
156 cvtype* luacv_checkObject(lua_State *L,int i,const char *tname)
157 {
158   return (cvtype*)((luacv_obj<cvtype>*)luaL_checkudata(L,i,tname))->
          data;
159 }
```

Komentář

Tato šablonová funkce přetypovává ukazatel *laucv_obj<cvtype*>* na OpenCV data *cvtype**, který je vrácen funkcí *luaL_checkudata()*.

Ostatní standardní typy (int/long, double/float, string) se kontrolují pomocí knihovních funkcí *luaL_checkinteger()*, *luaL_checknumber()* a *luaL_checklstring()*.

Pro posílání OpenCV dat a jejich zabalení do LuaCV kontejneru byla vyvinuta funkce *luacv_pushObject()*, která je založena na knihovní funkci *lua_newuserdata()*. V příkladu 3.5 lze vidět její přesná implementace.

Dále pro dynamické alokování je použita funkce *luacv_alloc()*, která je v současné době pouze makrem na OpenCV *cvAlloc()*. Díky tomu je LuaCV připraveno na výměnu alokačního mechanismu bez větších překážek. OpenCV alokátor má tu vlastnost, že zarovnává data, což je formou optimalizace.

Pokud uvážíme, že OpenCV alokátor je optimalizován na stejnou nejmenší velikost alokovaného bloku $4kb$ jako *malloc()* a LuaCV kontejner má velikost $9byte$ na 32 bit architektuře, dochází zde ke zbytečné alokaci místa. V některých případech LuaCV kopíruje malé objekty z OpenCV (CvPoint, CvSize, atd...), ale velikost těchto objektů nebývá větší než $20byte$. Proto je dále ponechán prostor pro výměnu a výběr alokačního mechanismu optimalizovaného pro malé objekty[6].

Možným problémem by poté mohl být rozdíl ve velikosti alokovaného bloku při uvolňování OpenCV dealokátorem na datech vytvořených jiným nestandardním alokátorem. Z toho důvodu nebyl zatím alokační mechanismus vyměněn.

[4]Netypovostí jazyka Lua je myšleno neschopnost rozeznat typy uživatelské paměti z C a C++.
[5]Technice balit ukazatel do struktury s pomocnými atributy se někdy říká tzv. „boxed pointer“.
[6]Alokátory malých objektů jsou např. projekt Loki nebo projekt jemalloc.

př. 3.5: Zaslání objektů na zásobník

```
161 template<typename cvtype>
162 void luacv_pushObject(lua_State *L,cvtype *data,const char *type_name,
        bool copy=false)
163 {
164   if (!data) lua_pushnil(L);
165   else
166   {
167     luacv_obj<cvtype> *obj=(luacv_obj<cvtype>*)lua_newuserdata(L,
            sizeof(luacv_obj<cvtype>));
168     obj->dealloc_data=true;
169     if (copy)
170     {
171       cvtype *d=(cvtype*)luacv_alloc(sizeof(cvtype));
172       *d=*data;
173       obj->data=d;
174     }
175     else
176     {
177       obj->data=data;
178       obj->ref=luaL_ref(L,LUA_REGISTRYINDEX);
179       lua_rawgeti(L,LUA_REGISTRYINDEX,obj->ref);
180     }

182     luaL_getmetatable(L,type_name);
183     lua_setmetatable(L,-2);
184   }
185 }
```

Komentář

Vstupem do této šablonové funkce je zásobník Lua *lua_State *L*, název meta tabulky *type_name* a proměnná podle které se kopírují data.
Pomocí *lua_newuserdata()* jsou na zásobníku vytvořena nová uživatelská data a je nastaven přepínač pro automatické uklizení dat pomocí GC.
Touto podmínkou zamezuje chybám způsobeným prázdným ukazatelm *NULL*, pokud se tak stane je na zásobník Lua interpretru poslán typ *nil*.
Pokud se data mají zkopírovat na místo pouhého ukazatele, je ručně dynamicky alokována paměť a její ukazatel zapsán do LuaCV kontejneru.

V opačném případě se zkopíruje adresa ukazatele, vytvoří se reference v registru *luaL_ref()* a jeho identifikátor je zapsán *lua_rawgeti()*.

Nakonec se načte meta tabulka dat *luaL_getmetatable()* a zapíše k nově vytvořeným uživatelským datům *lua_setmetatable()*.

Standardní Lua typy jsou posílány na zásobník pomocí knihovních funkcí viz. *lua_pushnil()*, *lua_pushnumber()* a *lua_pushstring()* s mírným rozšířením o práci s textovým řetězcem *f_msg*, který obsahuje Lua syntax funkce. Tyto funkce se jmenují *luacv_check*()*, ale pro zjednodušení zápisu s parametrem *f_msg* byly zkráceny makrem na *check*()*, kde je parametr *f_msg* doplňován automaticky.

Praktickým příkladem využití výše zmiňovaných funkcí je vidět na příkladu 3.6 realizující získání šířky textu implementovanou v modulu Core.

př. 3.6: Funkce na získání šířky textu

```
4113 static int luacv_cvGetTextSize(lua_State *L)
4114 {
4115   const char f_msg[]="int GetTextSize(string text, CvFont font, CvSize
          textSize)";
4116   if (lua_gettop(L)!=4)luaL_error(L,f_msg);
4117   int baseline;
4118   cvGetTextSize(checkstring(L,1),checkCvFont(L,2),checkCvSize(L,3),&
          baseline);
4119   lua_pushnumber(L,baseline);
4120   return 1;
4121 }
```

Komentář

Pomocí funkce *lua_gettop()* je realizována první stupeň ochrany, protože kontrolujeme počet parametrů na zásobníku.
Dále je samotné spouštění OpenCV funkce *cvGetTextSize()* s jejími parametry. Kontrolujeme zda je na první pozici zásobníku řetězec *checkstring()*, dále objekt Font *checkCvFont()* a nakonec objekt Size *checkCvSize()*.
Následně pomocí *lua_pushnumber()* vložíme proměnnou *baseline* na zásobník jako výstupní proměnnou.

Konstruktory objektů z C++ API v některých případech mají až 15 nepovinných argumentů nebo jsou přetížené, a proto využití metod kontroly typů, jak tomu bylo u C funkcí je nepraktické. Docházelo by totiž k velkému řetězení podmínkových výrazů.

Proto byl implementován algoritmus *luacv_compare()* pro snadnější kontrolu typů viz příklad 3.7, který ale není tak efektivní, jako původní řešení. Je využita třída *std::vector* standardní šablonové knihovny STL a funkce *luacv_getnames()* viz příklad 3.8, která získává textový popis proměnných na zásobníku a ukládá je do vektoru. V příkladu 3.9 je částečná implementace konstruktoru objektu *cv::Mat*.

př. 3.7: Alternativní porovnání typů

```
232 bool luacv_compare(std::vector<std::string>&arr1,std::vector<std::string>&
        arr2)
233 {
234   if (arr1.size()!=arr2.size())
235     return false;
236   for (size_t i=0;i<arr1.size();i++)
237   {
238       if (arr1[i].compare(arr2[i]))
239         return false;
240   }
241   return true;
242 }
```

Komentář

Vstupními argumenty funkce jsou vektory typu *std::vector*, ve kterých jsou uloženy textové reprezentace *std::string* typu objektů na zásobníku.
Nejdříve je zkontrolována délka obou polí *arr1.size(), arr2.size()* a pokud nejsou stejné je výsledek neshodný.
Pro všechny položky pole je prováděno porovnávání obsahu pomocí *std::string.compare()* metody. Pokud se liší je opět vrácena neshoda.
Pokud jsou všechny prvky stejné, je navrácena shoda obou polí.

př. 3.8: Alternativní detekce typů

```
244 std::vector<std::string>& luacv_getnames(lua_State *L)
245 {
246   size_t top=lua_gettop(L);
247   std::vector<std::string> *names=new std::vector<std::string>();
248   for(size_t i=1;i<=top;i++)
249   {
250     std::string tmp(lua_typename(L,lua_type(L,i)));
251     if(tmp=="userdata")
252     {
253       if (luaL_callmeta(L,i,"__tostring"))
254       {
255         tmp=std::string(lua_tostring(L,lua_gettop(L)));
256         names->push_back(tmp.substr(0,tmp.find(" ")));
257       }
258     }
259     else
260       names->push_back(tmp);
261   }
262   return *names;
263 }
```

Komentář

Tato funkce získává textovou reprezentaci typů proměnných na zásobníku Lua. Nejdříve je získána výška zásobníku a následně vytvořen vektor *std::vektor names* pro uložení názvů.
Poté se pro každý prvek na zásobníku Lua vytvoří dočasný řetězec *std::string tmp*, do kterého je uložen jeho název získaný funkcí *lua_typename()* a *lua_type()*.
Pokud jsou proměnné základních typů kromě uživatelských dat, uloží se na konec *std::vector.push_back()* vektoru *names*.
V opačném případě se využije popisu objektu v callback funkci *__tostring* a ta je zavolána *luaL_callmeta()*. Poté se z textové proměnné na zásobníku extrahuje název objektu *std::string.substr()*, který je umístěn na začátku popisu a uložíme jej na konec *std::vector.push_back()* vektoru *names*. Na konec vektor *names* použijeme jako výstupní parametr.

př. 3.9: Využití alternativního porovnávání

```
4  static int luacv_cvMat(lua_State *L)
5  {
6    size_t top=lua_gettop(L);
7    vector<string> names2, &names=luacv_getnames(L);
8    switch(top)
9     {
10      case 0:
11              pushMat(L,new cv::Mat());
12              return 1;
13      case 1:
14              names2.push_back(string(MAT_NAME));
15              if(luacv_compare(names,names2))
16              {
17                pushMat(L,new cv::Mat(*checkMat(L,1)));
18                return 1;
19              }
20              names2.clear();
21              names2.push_back(string(CVMAT_NAME));
22              if(luacv_compare(names,names2))
23              {
24                pushMat(L,new cv::Mat(checkCvMat(L,1)));
25                return 1;
26              }
27              break;
28      default:
29              luaL_error(L,"unknown parameters");
30    }

32    return 1;
33 }
```

Komentář

V této funkci vidíme částečnou realizaci konstruktoru třídy *cv::Mat*.
Nejprve zjístíme výšku zásobníku *lua_gettop()* a vytvoříme vektor *names2* pro ruční uložení názvů typů. V proměnné *names* je výstupní vektor funkce *luacv_getnames()*, který obsahuje názvy objektu na zásobníku.
V konstrukci *switch* pak dle počtu položek zásobníku provádíme kontrolu typů. Pro žádné parametry je zavolán výchozí konstruktor *cv::Mat()* a jeho výsledek poslán na zásobník *pushMat()*.
I pro malý počet prvku lze vidět výhoda alternativního porovnávání. V algoritmu nedochází k žádnému řetězeni podmínek. Funkcí *names2.push_back()* uložíme název požadovaného typu *string(MAT_NAME)* a provedeme porovnání *luacv_compare()*. Pokud se vektory rovnají, je použita funkce *checkMat()*, zavolán konstruktor a výsledek zaslán zpět na zásobník *pushMat()*.
Poté se musí vektor *names2* vyprázdnit *names2.clear()* pro další použití vyhledávání typu *CVMAT_NAME*.
Pokud je na zásobníku více parametrů než je definováno, zavolá se funkce *luaL_error()* a algoritmus se ukončí.

Struktury

OpenCV C API struktury jsou do LuaCV přemosťovány jako netypová uživatelská paměť, z toho důvodu nelze z Lua normálně přistupovat k jejím vlastnostem. Pro tento účel byla navržena sada funkcí využívající meta tabulky, pomocí kterých jsme schopni omezeně emulovat základní přístup k datům. Jako nejjednodušší řešení se nabízí využít samotných meta tabulek a konstanty držet přímo v nich. Tato metoda má ale nevýhodu, protože by každá instance objektu měla stejná data. Je to z toho důvodu, že všechny instance vlastní totožnou meta tabulku.

Z toho důvodu byl navržen mechanismus pomocí indexovacích funkcí *__index* a *__newindex*, dle kterého jsme schopni přistoupit k samotným datům instance struktury. Nevýhodou tohoto postupu je, že stejná indexovací funkce je volána na všechny požadované atributy struktury. Proto je tedy na algoritmu, aby rozlišil a vrátil nebo zapsal požadovaný atribut.

Indexovací funkce *__index* je volána při požadavku vrátit hodnotu atributu a *__newindex* při požadavku na zápis nové hodnoty.

V příkladu 3.10 lze vidět implementace indexovacích funkcí. Protože indexovací funkce kromě názvu funkce a názvu struktury jsou stejné, tak je bylo možno zapsat jako makro *makeIndexFunctions()*.

př. 3.10: Indexovací funkce

```
135 #define makeIndexFunctions(name)\
136 static int name##__index(lua_State *L)\
137 {\
138   int ret;\
139   if ((ret=luacv_methodSearch(lua_tostring(L,2),name##_v))!=-1)\
140     return name##_v[ret].index(L);\
141   lua_pushnil(L);\
142   return 1;\
143 }\
144 static int name##__newindex(lua_State *L)\
145 {\
146   int ret;\
147   if ((ret=luacv_methodSearch(lua_tostring(L,2),name##_v))!=-1)\
148     return name##_v[ret].newindex(L);\
149   return 0;\
150 }
```

Komentář

Když jsou tyto indexovací funkce zavolány, mají na zásobníku následující parametry: ukazatel na uživatelskou paměť, ze které se snažíme číst nebo zapisovat, název atributu, se kterým chceme operovat předaný jako řetězec a nakonec v případě *__newindex* novou hodnotu atributu.
V obou funkcích je použita funkce *luacv_methodSearch()*, která implementuje optimalizované vyhledávání v seřazeném poli. Jejím parametrem je pouze název požadované vlastnosti uložený v řetězci znaků. Pokud je nalezena shoda v názvech, je navrácen index v poli *ret* a nakonec zavolána obslužná funkce daného atributu. Pokud není atribut nalezen v *__index* funkci, je vrácen místo atributu *nil*.

Aby výše zmíněné indexovací funkce mohly fungovat, musí být zaregistrované v meta tabulce. U OpenCV C API struktur byly implementovány pouze základní callback funkce typu výpis popisu struktury, indexovací funkce a funkce GC. Operátory typu $<, >, ==, *, /, +, -$ nebyly implementovány z důvodu jejich absence v originálním návrhu OpenCV.

V příkladu 3.11 lze vidět makro na tvorbu meta tabulky objektu. Protože opět meta tabulky jsou identické, až na ukazatele funkcí v nich, bylo použito makro *makeObjectCallback()* pro jejich generování.

př. 3.11: Meta tabulka

```
 99 #define makeObjectCallback(name)\
100 const luaL_Reg name##_m[]=\
101 {\
102   {"__index",name##__index},\
103   {"__newindex",name##__newindex},\
104   {"__tostring",name##_tostring},\
105   {"__gc",name##_gc},\
106   {NULL,NULL}\
107 }
```

Komentář

Prvním atributem struktury *luaL_Reg* je název položky meta tabulky v Lua. Proto je v C šířen jako řetězec znaků. Druhým atributem je pak ukazatel na funkci realizující danou funkci.
__index a *__newindex* již byly popsány výše. Funkce *__tostring* realizuje vypsání informaci při použití funkce *print()* v Lua a *__gc* je callback funkce spouštěna před uklizením objektu pomocí GC.

Jak už bylo dříve zmíněno, vyhledávání v index funkcí funguje na způsobu Dijkstrova binárního vyhledávání[23] nad seřazeným polem. V příkladu 3.12 lze vidět přesná implementace funkce *luacv_methodSearch()*. Tato funkce se snaží do nejvyšší míry optimalizovat režii potřebnou pro vyhledání atributu struktury z řetězce znaků dodávaným interpretrem Lua.

Funkce pro získání a zápis atributu jsou uloženy ve struktuře typu *luacv_method* u každého objektu. Kvůli binárnímu vyhledávání je nutná znalost počtu položek, délky jména jednotlivých položek a musí být zajištěno jejich seřazení dle prvního znaku resp. jeho hodnoty v ASCII tabulce.

Pro omezení počtu chyb při ručním vytváření položek této struktury bylo implementováno makro *methodReg()* viz příklad 3.13, které dle jména atributu vyplní ostatní položky prvku struktury automaticky.

př. 3.12: Vyhledávání v index funkcích

```
131 int luacv_methodSearch(const char *pattern,const struct luacv_method *
        table)
132 {//table have to be alphabeticaly sorted
133   if ((!table) ||(!pattern)) return -1;
134   struct luacv_method *var=(struct luacv_method*)table;
135   unsigned char *s1=(unsigned char*)pattern,*s2=NULL;
136   size_t left=1,middle,right=table[0].len,len=strlen(pattern);

138   while(right!=(left+1))
139   {//binary dijkstr's search
140     middle=(left+right)/2;
141     if (*table[middle].name<=*s1)
142       left=middle;
143     else
144       right=middle;
145   }

147   for(var=(struct luacv_method*)&table[left];((s2=(unsigned char*)var
        ->name)&&(*(s1=(unsigned char*)pattern)==*s2));var--)
148   {//sequence search
149     if (var->len!=len) continue;
150     do
151       if (!(*(++s1))) return var-table;
152     while((*(s1))==*(++s2));
153   }

155   return -1;
156 }
```

Komentář

Vstupními parametry této fuknce je textový řetězec se jménem hledaného atributu *pattern* a ukazatel na pole struktur se funkcemi pro manipulaci s atributy *table*.
Nejprve proběhne kontrola, zda jsou vstupní parametry validní, jinak funkce vrací nedosažitelný index −1.
Vytvoří se ukazatele na první prvky pole *table* a ukazatel na první znak řetězce *pattern*. Proměnné *left*, *middle*, *right* slouží pro půlení intervalů v binárním vyhledávání.
Následně je vidět samotný algoritmus Dijkstrového binárního vyhledávání, který na rozdíl od klasického binárního vyhledávání předpokládá více stejných klíčů. Je to z toho důvodu, že binárním vyhledáváním porovnáváme pouze první znak řetězce.
Jakmile je nalezena pozice atributu se stejným počátečním znakem jako u proměnné *pattern*, zkontroluje se délka řetězce s uloženou hodnotou ve struktuře *len* a pokud souhlasí, použije se sekvenční porovnání znak po znaku. Porovnávají se ukazatele na znak řetězce *s1* a *s2*. Pokud je atribut nalezen, vrací se jeho index v poli, jinak −1.

př. 3.13: Registrace vlastnosti

```
116 #define methodReg(name,object)\
117   {#name,sizeof(#name)-1,object##_n##name,object##_##name}
```

Komentář

Pomocí makra se z názvu vytvoří řetězec jména, následně se vypočte jeho délka *sizeof()* a poté vyplní název callback funkcí.

Pro ucelení pochopení způsobu přemostění OpenCV struktur do Lua, je v příkladu 3.14 vidět hlavička a v př. 3.15 ukázáno řešení pro základní OpenCV C API strukturu typu *CvPoint*[22]. Tento typ byl vybrán s ohledem na svoji jednoduchost a snadnou čitelnost.

V těchto příkladech je vidět využití výše zmíněných pomocných maker a funkcí z pomocné knihovny *luacvaux*.

př. 3.14: Header CvPoint

```
1  #ifndef CVPOINT_NAME
2  #include "opencv2/opencv.hpp"
3  #include "luacvaux.h"

5  #define CVPOINT_NAME "CvPoint"
6  #define checkCvPoint(L,i) luacv_checkObject<CvPoint>(L,i,
       CVPOINT_NAME)
7  #define pushCvPoint(L,data) luacv_pushObject<CvPoint>(L,data,
       CVPOINT_NAME,true)

9  extern const struct luaL_Reg CvPoint_m[];
10 #endif
```

Komentář

Na prvním řádku je pomocí makra *#ifdef* zajištěno pouze jedno načtení objektu CvPoint pomocí direktivy *include*.
Následně definujeme interní název meta tabulky objektu CvPoint *CVPOINT_NAME*
Pro zjednodušení používání *luacv_checkObject()* funkce je u každého objektu definováno makro pro konkretní objekt viz *checkCvPoint()*.
Stejným způsobem je řešena i funkce *luacv_pushObject()* a tedy i *pushCvPoint()*.
Nakonec je definice meta tabulky.

př. 3.15: Objekt CvPoint

```
1  #include "CvPoint.h"

3  static int CvPoint_tostring(lua_State *L)
4  {
5    CvPoint *p=checkCvPoint(L,1);
6    lua_pushfstring(L,CVPOINT_NAME" object: %p\n\tx=%d\n\ty=%d",p,
         p->x,p->y);
7    return 1;
8  }

10 static int CvPoint_gc(lua_State *L)
11 {
12   luacv_obj<CvPoint>*obj=(luacv_obj<CvPoint>*)luaL_checkudata(L,1,
         CVPOINT_NAME);
13   if (obj->dealloc_data)
14     luacv_free(&(obj->data));

16   return 0;
17 }

19 static int CvPoint_nx(lua_State *L)
20 {
21   const char f_msg[]=CVPOINT_NAME".x=int";
22   checkCvPoint(L,1)->x=checkint(L,3);
23 return 0;
24 }

26 static int CvPoint_x(lua_State *L)
27 {
28   lua_pushnumber(L,checkCvPoint(L,1)->x);
29 return 1;
30 }

32 static int CvPoint_ny(lua_State *L)
33 {
34   const char f_msg[]=CVPOINT_NAME".y=int";
35   checkCvPoint(L,1)->y=checkint(L,3);
36 return 0;
37 }

39 static int CvPoint_y(lua_State *L)
40 {
41   lua_pushnumber(L,checkCvPoint(L,1)->y);
42 return 1;
43 }

45 static const luacv_method CvPoint_v[]=
46 {
47   {NULL,3,NULL,NULL},
48   methodReg(x,CvPoint),
49   methodReg(y,CvPoint),
50 };

52 makeIndexFunctions(CvPoint)
53 makeObjectCallback(CvPoint);
```

Komentář

Funkce *CvPoint_tostring()* jak už bylo řečeno realizuje výpis informací na obrazovku při zavolání funkce *print()*. Jak je vidět nejprve je třeba načíst adresu objektu *checkCvPoint()*. Poté stačí na zásobník vrátit formátovaný řetězec znaků *lua_pushfstring()*. LuaCV dodržuje konvenci pro formátování těchto výstupů, a proto zprávy u ostatních objektů vypadají obdobně. LuaCV se snaží vypisovat všechny relevantní atributy struktur.
Další funkcí je *CvPoint_gc()*, která je volána před uvolněním objektu pomocí GC. Nejdříve je nutno získat celý LuaCV kontejner objektu *luacv_obj* a dle proměnné *dealloc_data* je pak rozhodnuto, zda se mají OpenCV data uvolnit, nebo jejich instance existuje u jiného objektu.
Pokud data již nejsou využívána, jsou uvolněny funkcí *luacv_free()*.
Funkce *CvPoint_nx()* realizuje zápis nové hodnoty do proměnné x struktury CvPoint. V řetězci *f_msg* je uložena syntaxe pro případ nesprávného použití. Dále je zkontrolován typ objektu na zásobníku a vybavena adresa paměti *checkCvPoint()*, do které zapíšeme hodnotu ze zásobníku *checkint()* na místo proměnné x.

Pomocí funkce *CvPoint_x()* je realizováno vybavování hodnoty atributu x *checkCvPoint()* na zásobník *lua_pushnumber()*.
Funkce *CvPoint_ny()* je principiálně stejná jako předchozí funkce *CvPoint_nx()* s tím rozdílem, že zapisuje číslo ze zásobníku do atributu y.

Pole struktur typu *luacv_method* tvoří tabulku callback funkcí použitých pro seznam indexovacích funkcí. Je zde vidět využití makra *methodReg()* a položek seřazených dle hodnoty jména atributu. První položku tvoří prázdné hodnoty s tím rozdílem, že je v něm uložen celkový počet položek nutný pro binární vyhledávání.
Poté už jsou jen vytvořeny indexovací funkce *makeIndexFunctions()* a následná struktura meta tabulky *makeObjectCallback()*.

Objekty

V LuaCV verzi 0.2.0 byla dokončena implementace OpenCV C API, od této verze probíhá základní implementace OpenCV C++ API. Funkční přemostění zůstává principiálně stejné, ale přemostění tříd je odlišné od přemostění C struktur. Tato kapitola se bude zabývat metodami použitými pro C ++ API.

Základní rozdíl tříd oproti C strukturám je práce s metodami na rozdíl atributů. Protože prototypová stavba je stejná pro všechny třídy a nepoužívají se přímo jejich atributy, můžeme metody registrovat přímo do meta tabulky objektu. Odpadá tím nutnost používat vlastní indexovací funkci, jako tomu bylo u C struktur. V příkladu 3.16 lze vidět neúplná implementace přemostění třídy *cv::Mat*.

př. 3.16: Třída cv::Mat

```
63  static int Mat_diag(lua_State *L)
64  {
65    const char f_msg[]=MAT_NAME" Mat.diag(int d=0) || "MAT_NAME"
          Mat.diag("MAT_NAME" d)";
66    int top=lua_gettop(L);
67    Mat p;
68    switch(top)
69    {
70      case 1:
71              p=checkMat(L,1)->diag();
72            break;
73      case 2:
74            if (lua_isuserdata(L,2))
75              p=checkMat(L,1)->diag(*checkMat(L,2));
76            else
77              p=checkMat(L,1)->diag(checkint(L,2));
78            break;
79      default:
80            luaL_error(L,f_msg);
81    }

83    pushUserData(L,&p);
84    luaL_getmetatable(L,MAT_NAME);
85    lua_setmetatable(L,-2);
86    return 1;
87  }
88  static int Mat_clone(lua_State *L)
89  {
90    const char f_msg[]=MAT_NAME" Mat.clone()";
91    if(lua_gettop(L)!=1) luaL_error(L,f_msg);

93    Mat p=checkMat(L,1)->clone();
94    pushUserData(L,&p);
95    luaL_getmetatable(L,MAT_NAME);
96    lua_setmetatable(L,-2);
97    return 1;
98  }

100 const luaL_Reg Mat_m[]=
101 {
102   {"__tostring",Mat__tostring},
103   {"__gc",Mat__gc},
104   {"row",Mat_row},
105   {"col",Mat_col},
106   {"rowRange",Mat_rowRange},
107   {"colRange",Mat_colRange},
108   {"diag",Mat_diag},
109   {"clone",Mat_clone},
110   {NULL,NULL}
111 };
```

Komentář

V této funkci vidíme implementaci metody *diag()*, která přetížená a má dva různé prototypy. Jedním je funkce s nepovinným parametrem *int* a druhým s parametrem *Mat*. Dále je porovnávána výška zásobníku a dle počtu parametrů kontrolovány typy objektů.

Je důležité si uvědomit, jak se skládají parametry na zásobníku. Prvním parametrem jsou vždy uživatelská data ukazující na konkrétní instanci třídy *Mat*. Vychází to z Lua zápisu, kde volání metody *objekt:metoda()* je identické s voláním *objekt.metoda(objekt)*. Zápis s dvojtečkou je pouze věc syntaktického zjednodušení.

V případě výšky zásobníku rovno dvěma je jasné, že jde o případ s přetíženou funkcí. Nemůžeme v podmínce použít *checkMat()*, protože když by parametrem byl *int*, tak by funkce zahlásila chybu a ukončila program. Je proto využita funkce *lua_isuserdata()*, která pouze zjišťuje zda na zásobníku jsou libovolná uživatelská data. Pokud ano, použije se kontrolo *checkMat()*, která vrací ukazatel na data a na nich zavoláme metodu *diag()*.

Poté již vrátíme výsledek *p* na zásobník, ten ale nesmíme poslat jako *pushMat()*, ale jako uživatelská data, kterým ručně přiřadíme meta tabulku. Kdybychom použili *pushMat()*, GC by se tuto instanci snažil později uvolnit, ale tato instance je pouze dočasná. Poté by nastal tzv. „double free error“.

Zde je vytvořena meta tabulka pro tento typ. Oproti přemostěným C strukturám, nemá zaregistrované žádné indexovací funkce *__index* a *__newindex*. Všechny metody jsou zapsány přímo do meta tabulky.

Moduly

LuaCV implementuje téměř všechny OpenCV C API funkce z modulů (přes 600 funkcí). Moduly jsou v LuaCV implementovány v souborech pod jmény *lua_jméno.**. Každý z těchto modulů obsahuje jednotlivé implementace funkcí a jejich registraci do pole struktur typu *luaL_Reg*, jak tomu bylo i u meta tabulek objektů. Vedle funkcí jsou v modulech registrovány i OpenCV konstanty pro snazší zápis algoritmů.

Většina OpenCV funkcí lze přemostit do LuaCV bez větších obtíží. Vyskytlo se ale i pár funkcí, které potřebovaly buďto novou implementaci, nebo bylo třeba pozměnit jejich použití. V příkladu 3.17 lze vidět přemostění funkce *cvMat()*, která vytváří matici z předem definovaného pole čísel. U původního OpenCV řešení se počítá s tím, že uživatel definuje správný typ číselného pole a postará se o správné zarovnání dat v matici. Lua definuje pouze jeden typ čísel, a proto v rámci optimalizace bylo nepřípustné, aby uživatel vytvářel pouze matice s „floating point“ čísly. Z toho důvodu byl navržen algoritmus konverzních funkcí viz příklad 3.18, které automaticky přetypovávají Lua čísla na číselný typ definovaný uživatelem. Tyto funkce jsou generovány makrem *makeMatConvFunctionTo()* pro všechny typy, které se používají v OpenCV maticích.

př. 3.17: Tvorba uživatelské matice

```
293 static int luacv_cvMat(lua_State *L)
294 {
295   const char f_msg[]=CVMAT_NAME" Mat(int rows, int cols
        , table type, num data[]=nil)";
296   if (! lua_istable (L,3)) luaL_error(L,"Table is not valid
        conversion table.");
297   uchar *data=NULL;
298   int rows=checkint(L,1),cols=checkint(L,2),type;
299    size_t  top=lua_gettop(L);
300   lua_pushstring(L,"value");
301   lua_rawget(L,3);
302   if ( lua_isnil (L,top+1)) luaL_error(L,"Table is not valid
        conversion table.");
303   type=checkint(L,top+1);
304   lua_pop(L,1);
305   CvMat mat=cvMat(rows,cols,type,data);
306   switch (top)
307   {
308     case 3:
309             break;
310     case 4:
311             if (! lua_istable (L,4)) luaL_error(L,f_msg);
312             if (( size_t )CV_MAT_CN(CV_MAT_TYPE(
                    CV_MAT_MAGIC_VAL|
                    CV_MAT_CONT_FLAG|type))*rows*cols!=
                    lua_rawlen(L,4)) luaL_error(L,"Length of new
                    matrix data don't match with matrix.");
313             lua_pushstring(L,"to");
314             lua_rawget(L,3);
315             if ( lua_isnil (L,5)) luaL_error(L,"Table is not
                    valid conversion table.");
316             lua_replace(L,2);
317             pushUserData(L,&mat);
318             lua_replace(L,3);
319              lua_call (L,2,0);
320             break;
321     default:
322             luaL_error(L,f_msg);
323   }
324   pushCvMat(L,cvCloneMat(&mat));
325   return 1;
326 }
```

Komentář

V řetězci *f_msg* je definován prototyp funkce v Lua. Dále kontrolujeme zda je na třetí pozici v zásobníku tabulka *lua_istable()* a pokud není, je vyvolána chyba *luaL_error()*.
Poté vytvoříme ukazatel *data*, který bude obsahovat konvertovaná čísla. Následně ze zásobníku získáme požadovanou výšku *rows* a šířku matice *cols* pomocí *checkint()*. Funkcí *lua_gettop()* získáme počet prvku v zásobníku a extrahujeme z konverzní tabulky *lua_rawget()* na třetí pozici položku*lua_pushstring() value*, která obsahuje OpenCV typ matice. Pokud na vrcholu zásobníku není číslo *lua_isnil()*, znamená to, že tabulka na pozici 3 není konverzní tabulka, jinak si číslo uložíme *checkint()* do proměnné *type* a uvolníme ho ze stacku *lua_pop()*.
Jakmile máme všechna potřebná data, vytvoříme lokální matici pomocí *cvMat()*.
Následně dle počtu položek na stacku porovnáváme, zda uživatel zvolil i předdefinovaná data. Pokud ne, kopie matice *cvCloneMat()* se pošle na zásobník *pushCvMat()*.
Jestliže uživatel definoval data matice, je zkontrolováno zda proměnná je tabulka *lua_istable()* a zda délka tabulky *lua_rawlen()* odpovídá požadované délce tabulky *CV_MAT_CN*. Pokud vše souhlasí, je z konverzní tabulky na pozici 3 vybavena *lua_rawget()* konverzí funkce *to*. Následně je stack přerovnáván *lua_replace()* tak, aby byl připraven pro zavolání *lua_call()* konverzní funkce na lokálních datech *pushUserData()*.

Tato konverzní funkce *luacv_TableTo*()* přetypuje čísla na správný typ.

Nakonec se kopie *cvCloneMat()*upravené matice zašlou na zásobník *pushCvMat()*.

př. 3.18: Konverzní funkce

```
221 #define makeMatConvFunctionTo(name,type_t)\
222 static int luacv_TableTo##name(lua_State *L)\
223 {\
224   const char f_msg[]="userdata_TableTo"#name"("
          CVMAT_NAME"_mat,_num[]_data,_int_row=0,_int_col
          =0)";\
225   size_t  len=0;\
226   if  (!(len=lua_rawlen(L,2))) luaL_error(L,f_msg);\
227   CvMat *mat=(CvMat*)checkldata(L,1);\
228   size_t  top=lua_gettop(L);\
229   int row,col;\
230   switch (top)\
231   {\
232     case 2:\
233       mat->data.ptr=(uchar*)lua_newuserdata(L,
              CV_ELEM_SIZE(mat->type)*len);\
234       for( size_t  i=0;i<len;i++)\
235       {\
236         lua_rawgeti(L,2,i+1);\
237         ((type_t*)mat->data.ptr)[i]=(type_t)checknumber(L,
                top+2);\
238         lua_pop(L,1);\
239       }\
240       break;\
241     case 4:\
242           row=checkint(L,3);\
243           col=checkint(L,4);\
244           for( size_t  i=0;i<len;i++)\
245           {\
246             lua_rawgeti(L,2,i+1);\
247             ((type_t*)mat->data.ptr)[mat->cols*len*row+len
                    *col+i]=(type_t)checknumber(L,top+1);\
248             lua_pop(L,1);\
249           }\
250           break;\
251     default:\
252             luaL_error(L,f_msg);\
253   }\
254   return 0;\
255 }
```

Komentář

Toto makro definuje funkci dle požadovaného typu a jména daného parametry *type_t* a *name*.
Řetězec *f_msg* obsahuje pomocnou Lua syntax. Následně kontrolujeme zda druhý parametr na zásobníku je tabulka s nenulovým počtem položek *lua_rawlen()*. Poté je získána adresa matice *checkldata()* a je přetypována na správný typ *CvMat*. *top* obsahuje velikost zásobníku a na základě jeho velikost je pak rozhodnuto, který algoritmus použít.

V prvním případě se alokují data *lua_newuserdata()* správného typu *type_t* na určité místo v matici *i*. Data z Lua matice na stacku extrahujeme pomocí *lua_rawgeti()*, které pak pomocí *checknumber()* a *type_t* uložíme.
Následně aktuální číslo odebereme ze stacku *lua_pop()*.

V druhém případě, když jsou na stacku všechny parametry, je postupně získáme *checkint()*. Poté opět pro všechny prvky matice *len* zapisujeme hodnoty ze zásobníku *checknumber()* do přesně určeného místa v paměti *mat* → *data.ptr*. Číslo je pak odebráno ze zásobníku *lua_pop()*.

Mezi další funkce, které bylo nutné razantně upravit, aby mohly vykonávat alespoň částečnou funkčnost patří tvorba callback funkcí grafického rozhraní v OpenCV modulu Highgui. Konkrétně jde o funkce *cvCreateTrackbar()* a *cvSetMouseCallback()*. Tyto funkce počítají s C/C++ funkcí s pevně definovanými parametry např. *void func(int pos)* pro vytvoření trackbaru.

Z toho důvodu není možné do těch funkcí vložit adresu Lua zásobníku potřebného pro správný chod. Implementace těchto funkcí je možná díky jejich alternativních funkcí např. *cvCreateTrackbar2()*, které mají volitelný netypový parametr *void *param* pro uživatelské hodnoty.

V příkladu 3.19 lze vidět univerzální callback funkci pro událost změny pozice trackbaru a v příkladu 3.20 implementaci vytvoření trackbaru pomocí *cvCreateTrackbar()*.

Každá callback funkce se ukládá do pole struktur typu *luacv_callback*, a proto je maximální počet callback funkcí pevně určen při kompilaci. Tyto struktury obsahují číselný identifikátor callback funkce, ukazatel na Lua stack, a název okna uživatelského rozhraní, pro který je určen.

Obdobnou metodou je také vytvořen mechanismus pro události myši skrze funkci *cvSetMouseCallback()*.

př. 3.19: Univerzální callback funkce

```
466 void luacv_cvTrackbarCallback(int pos,void *userdata)
467 {
468     luacv_callback *call=(luacv_callback*)userdata;
469     lua_State *L=call->stack;
470     lua_getglobal(L,"package");
471     lua_getfield(L,-1,"loaded");
472     lua_remove(L,-2);
473     lua_getfield(L,-1,LIBNAME);
474     lua_remove(L,-2);
475     lua_getfield(L,-1,LIBCALLBACKS);
476     lua_remove(L,-2);
477     lua_rawgeti(L,-1,call->pos);
478     lua_remove(L,-2);
479     lua_pushnumber(L,pos);

481     lua_call(L,1,0);
482 }
```

Komentář

Parametry této funkce jsou pozice trackbaru *pos* a uživatelská proměnná *userdata* obsahující strukturu typu *luacv_callback*. Nejprve *userdata* přetypujeme zpět na správný typ, abychom s nimi mohli pracovat a použijeme uloženou adresu Lua zásobníku *call* → *stack*.
Protože v Lua 5.2 již nejsou moduly zaregistrované jako globální proměnné, musí musíme se k tabulce callback funkcí dostat skrze tabulku *package.loaded* pomocí funkcí *lua_getglobal()* a *lua_getfield()*. Jakmile načteme tabulku callback funkcí *LIBCALLBACKS* skrze hlavní tabulku LuaCV *LIBNAME*, odstraníme nepotřebné položky ze stacku *lua_remove()*. Pomocí parametru *pos* ze struktury *call* získáme index správné položky, která obsahuje Lua funkci přiřazenou pro událost změny polohy trackbaru. Až je funkce na vrcholu zásobníku *lua_rawgeti()*, tak ji zavoláme *lua_call()* s parametrem změny polohy *pos*.

př. 3.20: Trackbar

```
484 static int luacv_cvCreateTrackbar(lua_State*L)
485 {
486   const char f_msg[]="int CreateTrackbar(string
          trackbarName,string windowName,int count,func
          onChange(int pos))";
487   int len=0;
488   const char *name=checkstring(L,2);
489   if ((lua_gettop(L)!=4)||(!lua_isfunction(L,4))) luaL_error(L
          ,f_msg);
490   lua_getglobal(L,"package");
491   lua_getfield(L,-1,"loaded");
492   lua_getfield(L,-1,LIBNAME);
493   lua_getfield(L,-1,LIBCALLBACKS);

495   lua_remove(L,5);
496   lua_remove(L,5);

498   lua_insert(L,5);
499   lua_insert(L,4);
500   lua_insert(L,5);

502   len=lua_rawlen(L,-2);
503   lua_rawseti(L,-2,len+1);
504   lua_rawset(L,-2);

506   callbackTable[len]=(luacv_callback*)luacv_alloc(sizeof(
          luacv_callback));
507   luacv_callback *call=callbackTable[len];
508   call->stack=L;
509   call->pos=len+1;
510   sprintf(call->wname,"%s",name);

512    int val=0;
513   lua_pushnumber(L,cvCreateTrackbar2(checkstring(L,1),name
          ,&val,checkint(L,3),luacv_cvTrackbarCallback,call));
514   return 1;
515 }
```

Komentář

Tato funkce realizuje tvorbu trackbaru pro pojmenované okno a nastavení funkce řešící události při změně polohy.
Na počátku funkce je definovaná Lua syntax. Dále kontrolujeme velikost zásobníku *lua_gettop()*, Lua funkci *lua_isfunction()* a název trackbaru *checkstring()* proměnné *name*.
Pomocí *lua_getglobal()* a *lua_getfield()* se postupně dostaneme do tabulky callback funkcí *LIBCALLBACKS* v modulu *LuaCV*.

Poté odstraníme přebytečné parametry *lua_remove()* a přeskládáme stack *lua_insert()* tak, aby event funkce byla na vrcholu stacku. Následně funkcí *lua_rawlen()* získáme aktuální počet registrovaných callback funkcí a vytvoříme novou položku *lua_rawseti()*, do které uložíme Lua funkcí *lua_rawset()*.

Poté alokujeme *luacv_alloc()* novou položku do pole struktur typu *callbackTable* a vyplníme ji aktuálními daty.

Na konec zavoláme funkce *cvCreateTrackbar2()* pro vytvoření trackbaru a předáme ji za parametry ukazatel na univerzální callback funkci *luacv_cvTrackbarCallback()* a ukazatel na strukturu obsahující informace o stacku, id a názvu okna.

Obsloužení OpenCV výjimek

Při nesprávném použitím funkcí, jejich špatných atributech nebo velikostech OpenCV knihovna vyvolává výjimku. Ta obsahuje informaci, který parametr neprošel podmínkovým výrazem a následně v jakém zdrojovém souboru se kód nalézá. Bohužel informace jsou v některých případech spíše matoucí, protože obsahují cestu na umístění, kde byla knihovna zkompilována a na soubory, jenž byly použity při překladu (pomocné funkce) mimo oficiální API.

Z těchto důvodů byl implementován algoritmus na zachytávání neobsloužených výjimek, následné vypsání dodatečných informací ze zásobníku Lua a uvolnění paměti v GC. V příkladu 3.21 lze vidět zachytávání výjimek a v příkladu 3.22 jejich obsluhu.

př. 3.21: Chytání výjimek

```
199 void laucv_sigcatcher(lua_State *L)
200 {
201   lua_sethook(L,luacv_hook,LUA_MASKCALL,0);
203 #if defined(WIN32) || defined(WIN64)
204   SetUnhandledExceptionFilter(luacv_sighandler);
205 #else
206   struct sigaction sa;
207    sigset_t  set;
208   sigemptyset(&(sa.sa_mask));
209   sa.sa_flags=0;
210   sa.sa_handler=luacv_sighandler;
211   sigaction(SIGABRT,&sa,NULL);
213   sigemptyset(&set);
214   sigaddset(&set,SIGABRT);
215   sigprocmask(SIG_BLOCK,&set,NULL);
216 #endif
217 }
```

Komentář

Na začátku funkce registrujeme tzv. „háček“ *lua_sethook()*, který se spouští pří každém volání jakékoliv funkce *LUA_MASKCALL*. V tomto háčku se kopírují informace o aktuálním stacku, aby byl přístupný ve funkci zpracování výjimek.
Protože MS Windows nemá standardní POSIX API pro zachytávání signálů, je třeba použít makro *#if defined()* a tím oddělit funkce dle platformy. V MS Windows se používá funkce *SetUnhandledExpetionFilter()* pro registraci neobsloužených výjimek.
U systémů držících se POSIX normy, neobsloužená výjimka vyvolává signál *SIGABRT*, který následně ukončí program. Z toho důvodu se zaměříme na detekci signálu *SIGABRT* pomocí struktury typu *sigaction*, ve které nastavíme adresu callback funkce *luacv_sighandler()* a poté pomocí knihovní funkce *sigaction()* tuto strukturu zaregistrujeme.
Strukturou typu *sigset_t* a funkcemi *sigaddset()*, *sigprocmask()* nastavíme blokování tohoto signálu.

př. 3.22: Zpracování výjimek

```
163 #if defined(WIN32) || defined(WIN64)
164 LONG __stdcall luacv_sighandler(LPEXCEPTION_POINTERS
        exceptionPtrs)
165 {
166 #else
167 void luacv_sighandler(int signum)
168 {
169     signum=signum;//to avoid warnings
170 #endif
171     lua_getinfo(luacv_globalstack,"nS",&luacv_debug);
172     char line[255];
173     char sep[255];
175     sprintf(line,"LuaCV_catched_OpenCV_error_in_%s_
            function_"LIBNAME".%s()",luacv_debug.short_src,
            luacv_debug.name);
176     int len=strlen(line);
177     for(int i=0;i<len;i++)
178       sep[i]='=';
179     fprintf(stderr,"%s\n%s\n%s\nReference_in_",sep,line,sep);
181     lua_gc(luacv_globalstack,LUA_GCCOLLECT,0);
183     lua_getglobal(luacv_globalstack,"package");
184     lua_getfield(luacv_globalstack,-1,"loaded");
185     lua_getfield(luacv_globalstack,-1,LIBNAME);
186     lua_getfield(luacv_globalstack,-1,luacv_debug.name);
187     lua_call(luacv_globalstack,0,1);
188 #if defined(WIN32) || defined(WIN64)
189   return EXCEPTION_EXECUTE_HANDLER;
190 #endif
191 }
```

Komentář

Tato funkce je používána ve funkci *luacv_sigcatcher()*, a protože ta je závislá na platformě, tak i tato funkce je řízená pomocí maker *#if defined()*. Pro platformu Windows má funkce jiný prototyp než pro platformy postavené na POSIX.

Funkcí *lua_getinfo()* získáme informaci o funkci, která vyvolala výjimku, respektive o poslední zavolané funkci. Parametr *nS* udává typ návratových informací pro debug rozhraní *luacv_debug*.
Proměnné *line* a *sep* jsou buffery pro výpis informací na konzolový výstup.

Funkcí *sprintf()* zkopírujeme do proměnné *line* informace o funkci, která vyvolala výjimku spolu s jejím typem.
Poté vytvoříme separátor stejné délky *strlen()* jako text, aby výstup byl graficky odlišen od OpenCV výjimky. Následně separátor a informace zapíšeme *fprintf()* na chybový kanál *stderr*.
Funkcí *lua_gc()* uvolníme paměť z GC.

Následně je využita vlastnost každé přemostěné funkce, že zavolána *lua_call()* bez parametrů vypíše svou syntax. Díky tomuto efektu se do chybového hlášení dostane syntax a místo ve skriptu, kde funkce selhala. Abychom se k této funkci dostali, musíme projít skrze globální tabulky *lua_getglobal()* do tabulky modulů *lua_getfield()*, kde dle jména z debug rozhraní zavoláme *lua_call()* danou funkci.
Na konec pro plaftormu Windows z funkce vrátíme parametr *EXCEPTION_EXECUTE_HANDLER*.

3.1.2. Ukázkový skript

V rámci demonstrace funkčnosti přemostění LuaCV byly vypracovány téměř všechny oficiální OpenCV C API příklady. Jedním z nich je i detekce čar pomocí „Houghovy transformace“ viz příklad 3.23. Tento příklad demonstruje použití nejčastěji využívaných OpenCV modulů *core*, *imgproc* a *highgui*.

Na obrázku 3.2 lze vidět vstupní obraz a na obrázku 3.3 detekované hrany spolu s vykreslenými přímkami. Ve výstupním obrázku lze také vidět nové uživatelské rozhraní realizované v knihovně *Qt*.

př. 3.23: Houghova transformace

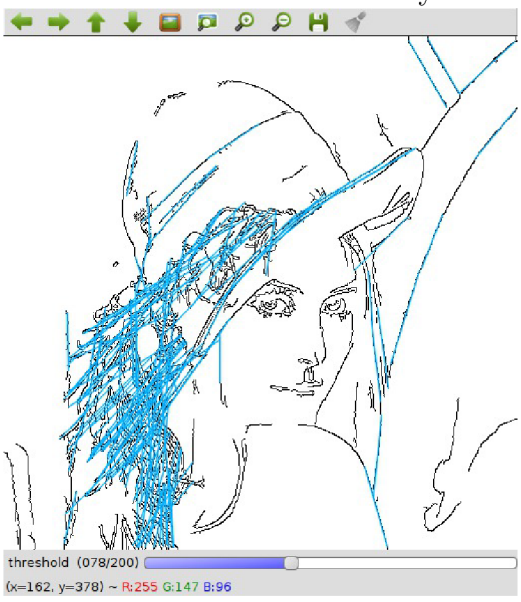
```
1  #!/usr/bin/env lua
2  cv=require('luacv')

4  src=cv.LoadImage(arg[1] and arg[1] or "lena.png",
       cv.CV_LOAD_IMAGE_GRAYSCALE)
5  if (not src) then error("Can't load source image "..arg[1])
       end

7  storage=cv.CreateMemStorage(0)
8  size=cv.GetSize(src)
9  dst=cv.CreateImage(size,cv.IPL_DEPTH_8U,1)
10 color_dst=cv.CreateImage(size,cv.IPL_DEPTH_8U,3)
11 wname="Hough Transform"
12 cv.NamedWindow(wname)

14 cv.CreateTrackbar("threshold",wname,200,function (thresh)
15   cv.Canny(src,dst,thresh,thresh*3,3)
16   cv.CvtColor(dst,color_dst,cv.CV_GRAY2BGR)
17   local lines=cv.HoughLines2(dst,storage,
         cv.CV_HOUGH_PROBABILISTIC,1,cv.CV_PI
         /180,50,50,10)

19   for i=0,lines.total do
20     local pts=cv.GetSeqElem(lines,i,cv.CvPoint["name"].."[2]")
21     cv.Line(color_dst,pts[1],pts[2],cv.CV_RGB(255,80,0),1,
           cv.CV_AA)
22   end
23   cv.ShowImage(wname,color_dst)
24 end)

26 cv.WaitKey()
27 cv.DestroyAllWindows()
```

Komentář

Funkcí *require('luacv')* načteme knihovnu LuaCV a uložíme si ji do proměnné *cv*. Načteme zdrojový obrázek*cv.LoadImage()* v odstínech šedi *cv.CV_LOAD_IMAGE_GRAYSCALE* a provedeme kontrolu zda se načetl. Pokud ne, je program ukončen chybou *error()*. Pro potřeby „Houghovy transformace“ je třeba vytvořit paměť typu *CvMemStorage* funkcí *cv.CreateMemStorage()*. Abychom mohli vytvořit výstupní obrázky *cv.CreateImage()*, musíme získat velikost zdrojového obrazu *cv.GetSize()*. Parametr *cv.IPL_DEPTH_8U* je konverzní tabulka, která říká, že data budou typu *uchar* tedy budou nabývat hodnot $0 - 255$. Poslední číselný parametr udává počet barevných kanálů v obraze. Následně vytvoříme okno grafického rozhraní *cv.NamedWindow()* se jménem daným v proměnné *wname*. Jakmile je vytvořeno okno, můžeme pro něj vytvořit trackbar *cv.CreateTrackbar()* pro změnu prahu „Cannyho detektoru“. Tato funkce požaduje za poslední parametr ukazatel na Lua funkci, v tomto případě je použita anonymní Lua funkce a definována přímo v těle funkce, což je častý způsob zápisu v Lua. Poté na obraz aplikujeme „Cannyho detektor“ *cv.Canny()* s určeným prahem a výsledek konvertujeme ze stupňů šedi do RGB *cv.CvtColor()*. Na binární hranový obraz aplikujeme „Houghovu transformaci“ *cv.HoughLines2()* a výsledné přímky uložíme do sekvence *lines* typu *CvSeq*. Následně pro všechny přímky *lines.total* získáme jejich koncové body *cv.GetSeqElem()* a vykreslíme je do barevného obrazu *cv.Line()*. Na konec obraz *color_dst* vykreslíme do okna *cv.ShowImage()*. Poté čekáme na stisknutí libovolné klávesy *cv.WaitKey()* a uvolníme všechna okna *cv.DestroyAllWindows()*.

obr. 3.2: Zdrojový obraz

obr. 3.3: Detekované čáry



3.1.3. LuaCV a GNUPlot

OpenCV *highgui* modul neimplementuje žádné algoritmy pro snadné vykreslení grafů, sice je možno grafy vykreslit pomocí základní kreslící funkcionality z modulu *core*, ale výsledek není nikdy příliš reprezentativní.

V OpenCV lze tedy generovat grafy pouze informativního charakteru viz obrázek 3.5.

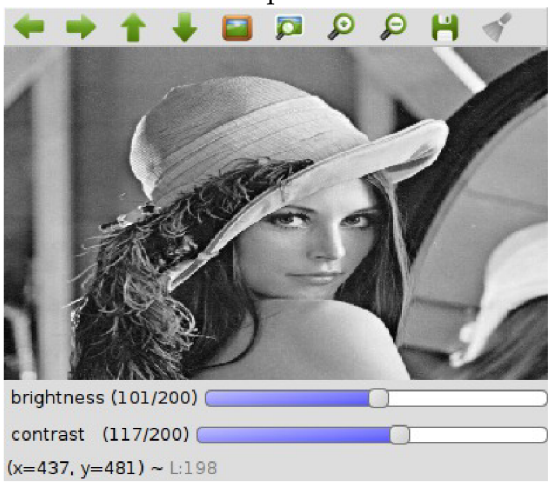
Z toho důvodu se často používá projekt *GNUPlot*, ten implementuje funkcionalitu pro vykreslení velkého množství grafů a dá se přirovnat k funkci *plot()* z Matlabu.

Jak už název napovídá, jedná se o Open Source projekt a je ho možno provozovat téměř na všech běžně používaných platformách.

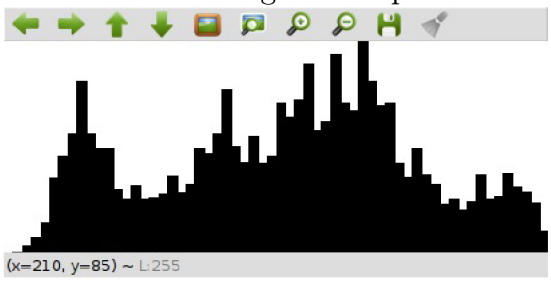
Pro využití GNUPlotu s knihovnou LuaCV se použije jeho schopnost čtení ze standardního vstupu a pomocí FIFO fronty „pipe“ jsou mu posílány příkazy přímo ze skriptu bez nutnosti použití dočasných souborů na disku.

Pro účely demonstrace kooperace LuaCV s GNUPlotem byl vypracován příklad 3.24, realizující změnu jasu a kontrastu v obraze. GNUPlot zde vykresluje příslušné histogramy v závislostech na pozicích trackbarů z grafického prostředí. Na obrázku 3.4 je vidět grafické prostředí pro manipulaci jasu a kontrastu. V obrazu 3.6 jsou příslušné histogramy vygenerované GNUPlotem.

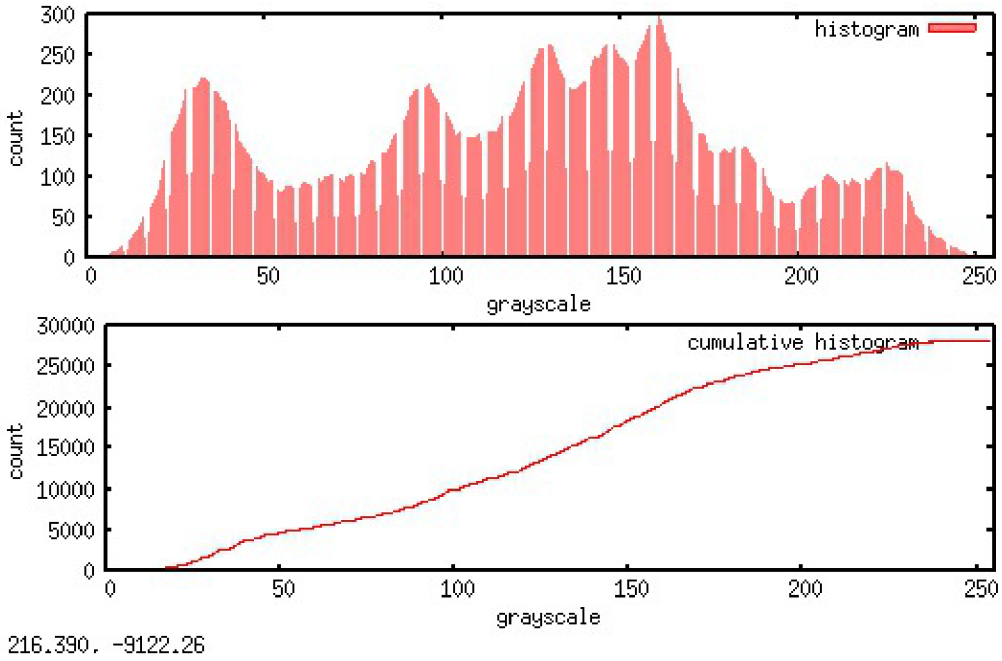
obr. 3.4: GUI v kooperaci s GNUPlotem



obr. 3.5: Histogram v OpenCV



obr. 3.6: Histogramy jasu obrazu v GNUPlot



př. 3.24: LuaCV a grafy v GNUPlot

```
#!/usr/bin/env lua
cv=require('luacv')

src_image=cv.LoadImage(not arg[1] and "lena.png" or arg[1],cv.CV_LOAD_IMAGE_GRAYSCALE)
if (not src_image) then error("Couldn't load image") end

dst_image=cv.CloneImage(src_image)
hist_width, hist_size =255,256
hist=cv.CreateHist(1,{ hist_size },cv.CV_HIST_ARRAY,{{0,hist_width}})
wname="image"

lut_mat=cv.CreateMat(1,hist_width,cv.CV_8UC1)
_brightness, _contrast=100,100

function update_hist()
  local brightness=_brightness -100
  local contrast=_contrast -100
  if (contrast>0) then
    delta=127*contrast/100
    a=255/(255-delta*2)
    b=a*(brightness - delta)
  else
    delta=-128*contrast/100
    a=(hist_width-delta*2)/255
    b=a*brightness + delta
  end

  for i=0,hist_width-1 do
    local v=cv.Round(a*i+b)
    if (v<0) then v=0
    elseif (v>255) then v=255
    end
    cv.Set1D(lut_mat,i,cv.ScalarAll(v))
  end

  cv.LUT(src_image,dst_image,lut_mat)
  cv.ShowImage(wname,dst_image)

  cv.CalcHist(dst_image,hist)
  minval,maxval=cv.GetMinMaxHistValue(hist)

  local str=''
  for i=0,hist_size-1 do
    str= str..i.. ' 0 '..cv.Round(cv.GetReal1D(hist.bins,i))..'\n'
  end

  local command=[[
  set xrange [0:]  ..hist_size..  [[]]
  set yrange [0:]
  set style fill solid 0.5
  set xlabel "grayscale"
  set ylabel "count"
  set multiplot
  set size 1, 0.5
  set origin 0.0, 0.5
  plot '-' u 1:2:3 w filledcu t 'histogram'
  ]] ..str.. [[
  e
  set origin 0.0,0.0
  plot '-' u 3 s cumulative t 'cumulative_histogram'
  ]] ..str.. [[
  e
  unset multiplot
  ]]
  gnuplot:write(command)
  gnuplot:flush()
end

cv.NamedWindow(wname)

cv.CreateTrackbar("brightness",wname,200,function(pos) _brightness=pos update_hist() end)
cv.CreateTrackbar("contrast",wname,200,function(pos) _contrast=pos update_hist() end)

gnuplot=io.popen('gnuplot','w')
update_hist()
repeat
  key=cv.WaitKey(10)
until key==27

cv.DestroyAllWindows()
gnuplot:close()
```

Komentář

Nejdříve načítáme zdrojový obraz *cv.LoadImage()* do stupňů šedi *cv.CV_LOAD_IMAGE_GRAYSCALE* a provedeme kontrolu zda se data správně načetla. Poté si vytvoříme kopii zdrojového obrazu *cv.CloneImage()* a definuje rozměry *hist_size* definiční obor *hist_width* dimenze histogramu. Následně vytvoříme uniformní 1D histogram *cv.CreateHist()*. Matice *lut_mat* je tzv. „lookup table“ vytvoříme ji přes *cv.CreateMat()*. Proměnné *_contrast* a *_brightness* budou používány pro grafické rozhraní. Funkce *update_hist()* je event funkcí pro oba trackbary nastavující kontrast a jas. Z počátku se vypočítávají převodní vztahy pro výpočet proměnných *delta*, *a* a *b*, které jsou následně použity pro inicializování *lut_mat*. Pro všechny prvky matice *lut_mat* počítáme součin indexu a proměnných *a* a *b*, který nakonec zaokrouhlíme *cv.Round()*. V podmínce ošetřujeme stav kdy je výsledek menší než 0 a větší než 255. Poté z dat vytvoříme typ *CvScalar* pomocí funkce *cv.ScalarAll()* a zapíšeme jej do matice *cv.Set1D()*. Funkcí *cv.LUT()* vyplníme výstupní obrázek ze vstupního na základě *lut_mat*. Výsledek zobrazíme do okna *cv.ShowImage()*. Funkcí *cv.CalcHist()* vypočítáme histogram výsledného obrazu a získáme jeho minimální a maximální hodnoty *cv.GetMinMaxHistValue()*. Pro všechny prvky histogramu *hist.bins* ukládáme do textového řetězce jeho data *cv.GetReal1D()*. V Proměnné *command* je uložen příkaz pro GNUPlot na vykreslení histogramů z dříve nasbíraných dat *str*. Vidíme, že transformace dat na histogram nebo kumulativní histogram je pouze ve změně funkce GNUPlotu. Pomlčka '-' naznačuje, že data přijdou ze standardního vstupu a proto se vkládají přímo za příkaz a musí být ukončené znakem *e*. Poté jsou data zapsána *gnuplot:write()* do deskriptoru spojení s GNUPlotem a je vyprázdněn buffer *gnuplot:flush()*, aby příkaz došel celý. Po konci deklarace callback funkce *update_hist()* vytvoříme okno *cv.NamedWindow()* a k němu dva trackbary pro změnu kontrastu a jasu *cv.CreateTrackbar()*. V trackbarech jsou použité anonymní funkce, které ale volají funkci *update_hist()* a předávají příslušné pozice trackbaru. Pomocí *io.popen()* otvíráme „pipe“ mezi GNUPlotem a Lua interpretrem. Deskriptor je uložen v proměnné *gnuplot*. Ručně zavoláme funkci *update_hist()* aby se inicializoval obraz v okně a poté již čekáme, kdy uživatel stiskne klávesu *cv.WaitKey()* ESC pro přerušení programu. Pak jsou uvolněna všechna okna *cv.DestroyAllWindows()* a uzavřeno spojení s GNUPlotem *gnuplot:close()*

3.1.4. Kompilace zdrojových souborů

Kompilace LuaCV je v normálním případě velmi závislá na vývojovém prostředí a platformě. Z toho důvodu byl použit Open Source projekt *CMake*, který generuje MakeFile nebo projekt v závislosti na detekované platformě a kompilátoru. Z tohoto důvodu není třeba udržovat statický MakeFile pro různé platformy nebo jiné projekty pro různá IDE. Stačí pouze definovat pravidla pro CMake a on dle situace vygeneruje potřebné soubory pro kompilaci.

Výhodou CMake je také vlastnost, že dokáže pomocí modulů nalézt umístění knihoven, ke kterým se program má linkovat. Na UNIXových systémech by to nebyl problém, ale na platformě MS Windows není unifikované umístění hlaviček a knihoven různých projektů.

V příkladu 3.25 lze vidět vyhledávání knihoven Lua a OpenCV a následně nastavení umístění hlaviček a nastavení umístění linkeru, příklad 3.26 obsahuje nastavení výstupní knihovny.

př. 3.25: Vyhledávání knihoven

```
14 set(CMAKE_MODULE_PATH "${CMAKE_SOURCE_DIR}/
       cmake_modules")
15 find_package(Lua51 REQUIRED)
16 find_package(OpenCV REQUIRED)

18 include_directories(
19     "${CMAKE_CURRENT_SOURCE_DIR}/objects"
20     "${CMAKE_CURRENT_SOURCE_DIR}"
21     "${LUA_INCLUDE_DIR}"
22     )

24 link_directories(
25    "${CMAKE_CURRENT_SOURCE_DIR}/objects"
26    "${CMAKE_CURRENT_SOURCE_DIR}"
27    "${LUA_LIBRARIES}"
28    "${OpenCV_LIB_DIR}"
29    )
```

Komentář

Funkcí *set()* přidáváme cestu pro neoficiální OpenCV CMake modul, tak aby ho CMake automaticky používal.
find_package() volá modul, který má na starosti nalezení instalace projektů Lua a OpenCV a následně uložení jejich umístění do proměnných *LUA_INCLUDE_DIR*, *LUA_LIBRARIES* a *OpenCV_LIB_DIR*.

Funkce *include_directories()* nastavuje cestu pro hlavičky projektu LuaCV a Lua, proměnná *CMAKE_CURRENT_SOURCE_DIR* obsahuje umístění hlavního adresáře LuaCV.

link_directories() je ve své podstatě podobná funkci *include_directories()* až na to, že určuje umístění, ve kterých se bude LuaCV linkovat s ostatními soubory.

př. 3.26: Nastavení překladu

```
79 set(LUACV_LIBS ${CORE} ${CALIB3D} ${FEATURES2D}
       ${HIGHGUI} ${IMGPROC} ${OBJDETECT} ${
       LEGACY} ${VIDEO})

81 add_library(${TARGET} SHARED "luacv.cpp")

83 set_target_properties(${TARGET} PROPERTIES
84   SOVERSION ${LUACV_VERSION}
85   OUTPUT_NAME "${TARGET}"
86   PREFIX ""
87   ARCHIVE_OUTPUT_DIRECTORY "${
         CMAKE_CURRENT_SOURCE_DIR}"
88   RUNTIME_OUTPUT_DIRECTORY "${
         CMAKE_CURRENT_SOURCE_DIR}"
89   LINKER_LANGUAGE CXX
90   )
91 if(WIN32)
92   set_target_properties(${TARGET} PROPERTIES
         SUFFIX ".dll")
93 else()
94   set_target_properties(${TARGET} PROPERTIES
         SUFFIX ".so")
95 endif()

97 target_link_libraries(${TARGET} ${LUA_LIBRARIES} ${
       OpenCV_LIBS} ${LUACV_LIBS})
```

Komentář

Pomocí funkce *set()* nastavíme seznam objektů *LUACV_LIBS*, ke kterým se bude LuaCV linkovat, jedná se o všechny moduly.

S *add_library()* řekneme, aby CMake vytvořil sdílenou knihovnu z hlavního souboru *luacv.cpp*.

Dále funkcí *set_target_properties()* nastavujeme vlastnosti, které se týkají názvu a umístění, kde bude knihovna zkompilována. Položka *SOVERSION* udává verzi sdílené knihovny.

Podmínka *if()* řídí název přípony knihovny dle platformy. Pokud jde o Windows, je přípona *dll* a jinak *so*.

Funkce *target_link_libraries()* spouští samotný proces kompilace a zajišťuje, aby knihovna byla zkompilována a slinkována s každou potřebnou komponentou.

3.2. Instalátor

Tato kapitola se věnuje návrhu instalátoru pro knihovnu LuaCV. Tvorba instalátoru si klade za cíl zjednodušit uživatelům instalaci knihovny LuaCV pod platformou MS Windows a její aktualizaci. Důležitým aspektem při výběru instalačního programu je svobodná licence, jednoduchost, velikost binárního souboru, kompatibilita mezi platformami MS Windows a rychlost spouštěného kódu. V praxi se používá celá řada instalátorů, ale jejich aplikace závisí na mnoha faktorech například na typu platformy, OS nebo prostředí.

3.2.1. Populární instalátory

V tabulce 3.2 můžeme vidět příklady často používaných instalačních frameworků a porovnání jejich vlastností. Nevýhodou některých instalátorů je závislost na vývojovém prostředí třetích stran například na MS Visual Studiu nebo na Java Development Kitu (JDK), pro snížení závislostí je žádoucí se těmto instalátorům vyhnout a minimalizovat tím závislosti nutné pro spuštění instalace. Pro projekt LuaCV je využit instalátor NSIS hlavně kvůli jeho malé velikosti, množství pluginů, množství dostupných příkladů, rychlému návrhu kódu a stabilitě.

tab. 3.2: Nejčastěji používané instalační programy

Framework	**Podporované vlastnosti**			
	Otevřená licence	*Nezávislost na knihovnách 3. stran*	*Jazyková podpora*	*Multiplatformní*
Installjammer	•	•	•	•
NSIS	•	•	•	○[7]
Inno Setup	•	•	•	
Install Shield		•	•	
MSI			•	

3.2.2. NSIS instalátor

Tento instalační nástroj je komplexní skriptovací systém pro platformy MS Windows. Byl vyvinut společností *Nullsoft*, která jej používala ve svých produktech, mezi které patří populární hudební přehrávač *Winamp*. Tato firma NSIS později uvolnila pod svobodnou licencí *Zlib*[8] a poté se na dalším vývoji začali podílet z velké části nezávislí vývojáři. Mezi největší přednosti NSISu patří:

- Komplexní modulární systém.
- Malá režie a velikost instalátoru.
- Podpora více jazyků v rámci instalátoru.

[7] NSIS nemá oficiální podporu jiných OS než MS Windows, ale instalátor i vývojové prostředí fungují naprosto bezchybně pod přemostěním WINE.

[8] Zlib licence je kompatibilní s licencí GPL. Oproti GPL se vyznačuje značnou jednoduchostí a definuje pouze ochranu autora oproti poškození cizího majetku zapříčiněného jeho softwarem a povinnost uvádět předchozí autory u odvozeného softwaru.

- Preprocesor kódu.
- Podpora kompresních algoritmů (bzip2[9], LZMA[10], zlib[11]).
- Nezávislost na kompilátorech třetích stran.
- Možnost bezproblémového vývoje i na oficiálně nepodporovaných platformách (Wine).

NSIS používá jednoduchý zásobníkový jazyk založený na postfixové notaci, proto se při programování musíme omezit pouze na jednoduché konstrukce a rozhodovací operace pomocí návěští, jak je tomu například u jazyku *Assembler*.

Součástí NSISu jsou i dialogy a funkce pro tvorbu odinstalátorů (uninstaller), tyto funkce a dialogy jsou z velké části paralelní s těmi pro instalaci a mnoho z nich je stejných. Tvorbou odinstalačních mechanismů se tato práce nebude zabývat.

3.2.3. Základní instrukce

Jazyk NSISu obsahuje malé množství základních funkcí, které ale postačí při návrhu jednodušších aplikací. U složitějších aplikací je potřeba využít dostupných modulů a jejich pokročilejší funkcionality. Ty v zásadě obsahují buď funkce navržené na základním NSIS API, nebo WIN32 API pomocí sdílených knihoven zkompilovaných v nativním kódu. Mezi nejdůležitější instrukce, které definuje NSIS API patří:

- *Push()* - Vložení hodnoty na vrchol zásobníku.
- *Pop()* - Odebrání hodnoty z vrcholu zásobníku.
- *StrCmp()* - Porovnávání obsahu textové proměnné.
- *StrCpy()* - Kopírování obsahu textové proměnné.
- *IntCmp()* - Porovnání obsahu číselné proměnné.
- *IntOp()* - Matematická operace s číselnou proměnnou.
- *Call()* - Volání funkce.
- *Goto()* - Skok na návěští v programu, povoleny jsou i relativní skoky např. *(+3)*.
- *Abort()* - Konec programu.

Tyto funkce definují pouze nejnutnější operace pro práci s pamětí. Všechny instalátory ale potřebují funkce pro manipulaci se soubory a adresáři na souborovém systému, k tomuto účelu slouží následující instrukce:

- *FileOpen()* - Otevření deskriptoru souboru.
- *FileClose()* - Zavření deskriptoru souboru.

[9] Bzip2 je svobodná implementace Borrows-Wheeler algoritmu. Oproti ostatním algoritmům jako zip nebo gzip má lepší kompresní poměr, ale je značně pomalejší.

[10] LZMA je relativně novým algoritmem, je vyvíjen od roku 1998 a byl prvně použit v archivačním formátu 7z. Má značně vyšší kompresní poměr než bzip2.

[11] Zlib knihovna implementuje populární gzip algoritmus.

- *FileRead()* - Čtení ze souboru.
- *FileWrite()* - Zápis do souboru.
- *Delete()* - Smazání souboru nebo složky.
- *Rename()* - Přesun souboru.

Abychom mohli spustit námi definovaný kód, je třeba jej zasadit do některého z bloků. Tyto bloky se liší místem v instalátoru, kde se vykonají. Mezi základní bloky patří:

- **Function**[12] - Definuje blok funkce, která posléze musí být zavolána *call* instrukcí, aby se kód provedl. Vstupní parametry pro funkci se dají předat pomocí zásobníku *$0-9* nebo pomocí registrů *$R0-9*, dalším způsobem je globální proměnná nebo makro.
- **Section** - Sekce instalátoru, která se spustí v závislosti na parametrech instalátoru, tyto sekce mohou být pojmenované nebo anonymní. Pokud jsou pojmenované, tak se zobrazí v dialogu pro výběr komponent a záleží na uživateli, jestli kód obsažených v nich spustí. Anonymní sekce se spouštějí vždy.
- **SubSection** - Sekce vyššího řádu, která dokáže spojit několik sekcí. Používá se zpravidla pro seskupování sekcí v dialogu výběru komponent jaký lze vidět na obrázku 3.9.

Jako nejvýhodnější volbu pro knihovnu LuaCV se jevil instalátor, který stahuje data z internetového repositáře. To má výhody hlavně proto, že při aktualizaci knihovny stačí pouze aktualizovat archiv na repositáři a nemusíme překompilovávat instalační soubor. Protože není třeba větších zásahů do operačního systému MS Windows ze strany instalátoru, tak odinstalační program nebyl naprogramován, protože stačí pouze smazat instalační adresář a v systému pak po aplikaci nic nezůstane. Pro potřeby internetového instalátoru bylo třeba vyšší funkcionality, a proto byly použity některé volně dostupné moduly pro NSIS, mezi které patří:

- **AccessControl** - Slouží k nastavení přístupových práv pro soubory na souborovém systému NTFS[13].
- **Inetc** - Vrstva nad *Inet* knihovnou, která zprostředkovává základní funkce ke komunikaci s TCP/IP sítěmi.
- **ZipDll** - Plugin starající se o manipulaci s archivy typu **.zip*.
- **UnTgz** - Plugin obsahující funkce pro extrakci **.tar.gz* archívů tzv. *tarball*.
- **Locate** - Funkce pro rekurzivní prohledávání adresářové struktury a selekce souborů na základě masky.

[12] V NSISu je definováno několik callback funkcí. Ty jsou volány v závislosti na provedené události. Mezi nejvýznamnější z nich patří *.onInit, .onUserAbort, .onInstFailed, .onVerifyInstDir*.

[13] Souborový systém NTFS byl obsažen v produktech MS Windows od verze NT4.O. Proto rutiny pro nastavení práv nebudou fungovat na starších systémech např. Windows 98 nebo Windows ME.

3.2.4. Instalační skript

Tato kapitola se bude věnovat detailnímu popisu důležitých částí internetového instalátoru knihovny LuaCV. V prvé řadě bude popsáno využití základního NSIS API a funkcí dostupných z některých volně dostupných modulů. Není zde uveden celý kód instalátoru, ale pouze jeho zajímavé části, celý lze vidět v příloze. V příkladu 3.27 vidíme základní i pokročilé nastavení vnitřních proměnných instalátoru.

př. 3.27: Nastavení základních parametrů

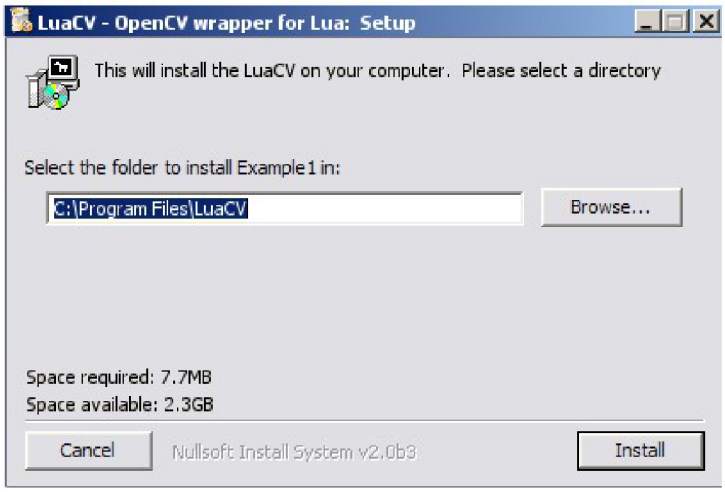
```
1  ;NSIS script for LuaCV net installer.
2  !include 'MUI2.nsh'
3  !include "FileFunc.nsh"
4  !include "EnvVarUpdate.nsh"
5  !define APP_NAME 'LuaCV'
6  !define TEMP_DIR '$TEMP\luacv'
7  Name '${APP_NAME} - OpenCV wrapper for Lua: '
8  OutFile '${APP_NAME}.exe'
9  ShowInstDetails show

11 Var /GLOBAL LUACV_IS_DOWNLOADED
12 Var /GLOBAL LUACV_SAMPLES


15 InstallDir '$PROGRAMFILES\${APP_NAME}'
16 DirText "This will install the ${APP_NAME} example on
       your computer.  Please select a directory"
17 ComponentText "This will install ${APP_NAME} on your
       computer. Select which componentsdo you want to
       install."
```

Komentář

Na prvních řádcích skriptu vidíme makro *!include*, pomocí kterého jsou zpřístupněny rozšířené sady funkcí. Jedná se o knihovny *MUI2*, *FileFunc* a *EnvVarUpdate*. Pomocí makra *!define* definujeme pomocné proměnné. Vlastnost *Name* umožňuje nastavit nadpis programu, *OutFile* zase název výstupního souboru resp. název instalátoru, *ShowInstDetails* nastaví detailní výpis operací, které se vykonávají při manipulaci se soubory.
Klíčovým slovem *Var* můžeme alokovat uživatelské proměnné, pokud přidáme i parametr */GLOBAL*, pak jsou proměnné viditelné i pro zbytek programu.
Vlastnost *InstallDir* nastaví výchozí instalační adresář, v tomto případě je využito globální proměnné *$PROGRAMFILES* místo ručně napsané absolutní cesty např. *C:/Program Files*. *DirText*, *ComponentText* a *BrandingText* nastavují uživatelské texty v různých částech instalátoru.

V následujícím příkladu 3.28 vidíme definici uživatelských dialogů použitých v instalátoru. Tyto dialogy jsou součástí sady funkcí z knihovny *MUI2* (Modern User Interface). Tato sada maker obsahuje několik před-generovaných dialogů, mezi které patří uvítací stránka, licenční ujednání, výběr instalovaných komponent, výběr instalačního adresáře, dialog pro možnost přidání aplikace do *Start menu*, dialog pro vypsání průběhu instalace a stránka zobrazovaná po skončení instalace.

obr. 3.7: Grafický design MUI1



Pro komunikaci s těmito dialogy je připravena sada callback funkcí, které můžeme měnit v pevně definovaných událostech (inicializace stránky, opuštění stránky, ...).

Při tvorbě instalátoru byla použita druhá verze těchto dialogů, v předchozí verzi *MUI* byla stejná funkcionalita, ale se starším grafickým designem podobná grafickému stylu Windows ME. Na obrázcích 3.7 a 3.8 vidíme porovnání mezi grafickým designem MUI1 a MUI2 v dialogu pro výběr instalačního adresáře.

V tomto dialogu probíhá funkce pro kontrolu existence instalačního adresáře a dostupnosti místa na disku. Pokud adresář existuje, instalátor jej smaže. Tato funkce se spustí při události zmáčknutí tlačítka *next*.

Na obrázku 3.9 vidíme dialog pro výběr instalovaných komponent. Instalátor byl navržen tak, aby bylo dosaženo co největší modularity. Hlavně z důvodu, aby se při změně v knihovně nemusely aktualizovat všechny soubory na repositáři.

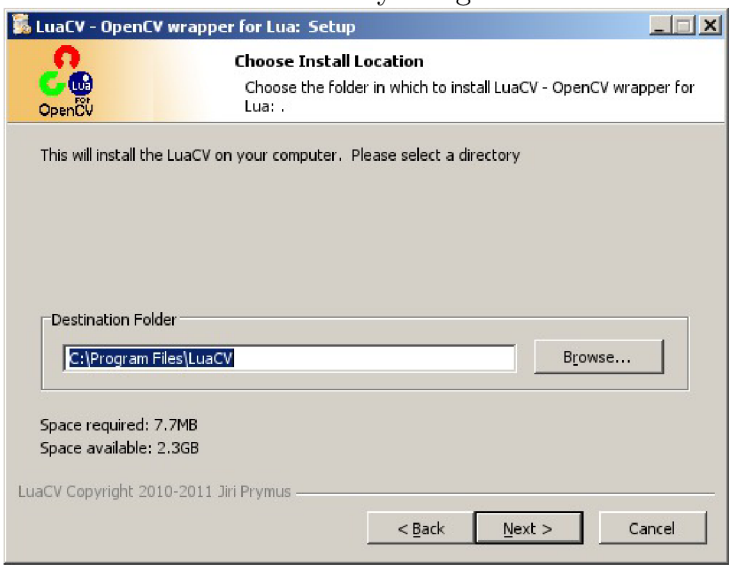
Díky tomu zůstane binární soubor instalátoru stejný během vydávání jednotlivých verzí knihovny, pokud nebudou nutné aktualizace samotného instalátoru nebo přidání nové funkcionality.

Každá komponenta v instalátoru je definována pomocí bloku *subsection, section*, kde jejich pořadí je dáno pozici v kódu. Přiřazení sekcí k jednotlivým instalačním typům *minimal*, *custom* a *full* je pomocí funkce *SectionIn()*.

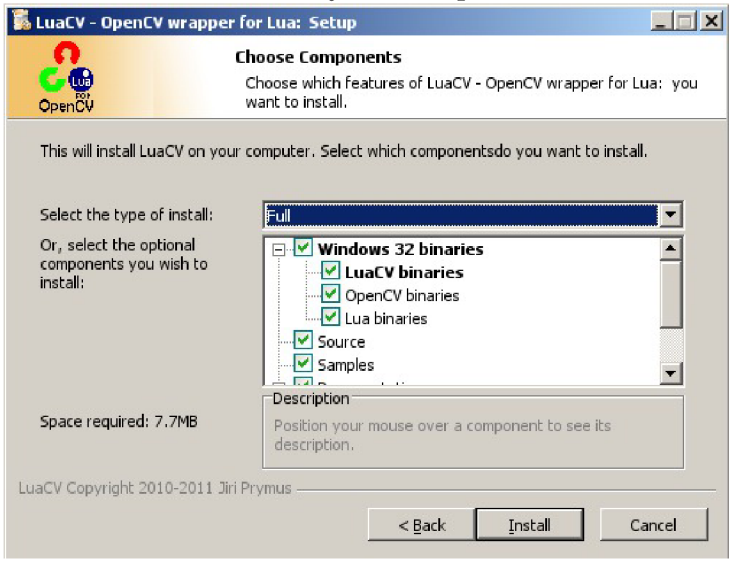
Na obrázku 3.10 je vidět dialog s výpisem průběhu instalace. Ten obsahuje nejen tzv. *progress bar*, ale i podrobné operace se soubory. Aby se během operace vypisovaly podrobné informace, bylo nutné změnit vnitřní proměnnou *ShowInstDetails* na hodnotu *show*.

Díky modulu *Inet* se v průběhu instalace zobrazuje i progressbar stahování i v četně rychlosti stahování.

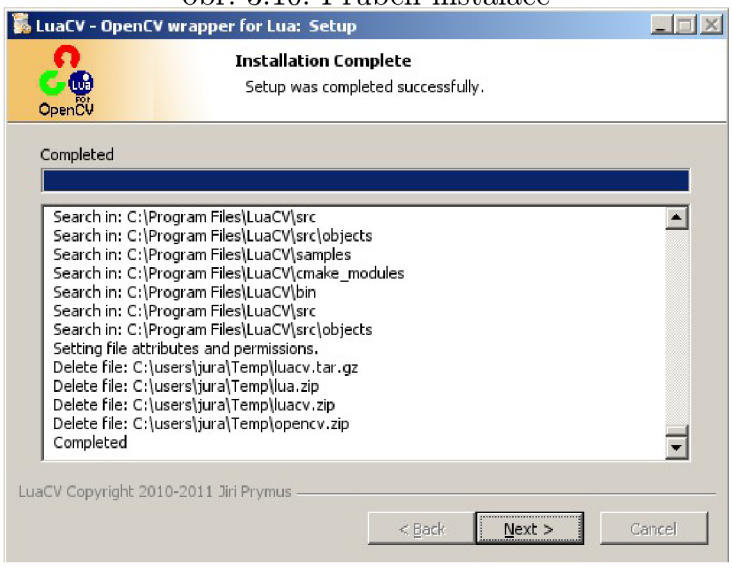
obr. 3.8: Grafický design MUI2



obr. 3.9: Výběr komponent



obr. 3.10: Průběh instalace



př. 3.28: Použití grafického rozhraní

```
20 !define MUI_ICON '${NSISDIR}\Contrib\Graphics\Icons\
       orange-install.ico'
21 !define MUI_HEADERIMAGE
22 !define MUI_HEADERIMAGE_BITMAP '${NSISDIR}\
       Contrib\Graphics\Header\luacv.bmp'
23 !define MUI_WELCOMEFINISHPAGE_BITMAP '${NSISDIR
       }\Contrib\Graphics\Wizard\luacv2.bmp'

25 !insertmacro MUI_PAGE_WELCOME

27 !define MUI_LICENSEPAGE_RADIOBUTTONS
28 !insertmacro MUI_PAGE_LICENSE '${NSISDIR}\luacv\
       LICENSE.txt'

30 !define MUI_PAGE_CUSTOMFUNCTION_LEAVE dir_check
31 !insertmacro MUI_PAGE_DIRECTORY

33 !define MUI_COMPONENTSPAGE_SMALLDESC
34 !insertmacro MUI_PAGE_COMPONENTS

36 !insertmacro MUI_PAGE_INSTFILES

38 !define MUI_FINISHPAGE_SHOWREADME '$INSTDIR\
       README.txt'
39 !define MUI_FINISHPAGE_RUN
40 !define MUI_FINISHPAGE_RUN_TEXT 'Run testing script'
41 !define MUI_FINISHPAGE_RUN_FUNCTION 'runTest'

43 !define MUI_FINISHPAGE_LINK 'LuaCV project website on
       sourceforge.net'
44 !define MUI_FINISHPAGE_LINK_LOCATION 'http://www.
       sourceforge.net/projects/luacv'
45 #!define MUI_FINISHPAGE_NOAUTOCLOSE
46 !define MUI_FINISHPAGE_TEXT_LARGE
47 !insertmacro MUI_PAGE_FINISH

49 !insertmacro MUI_LANGUAGE "English"
```

Komentář

Makro *MUI_ICON* mění přednastavenou ikonu za uživatelskou, makra *MUI_HEADERIMAGE* přidává do instalátoru grafický pruh s obrázkem do vrchní části aplikace. Makro *MUI_WELCOMEFINISHPAGE_BITMAP* udává cestu k obrázku použitému v uvítací a konečné stránce aplikace.
Makrem *MUI_PAGE_WELCOME* zaregistrujeme do instalátoru uvítací stránku. Obdobně to funguje i u ostatních stránek.
Pro zaregistrování událostí a callback funkcí, které se mají vykonat při opuštění dialogu se používá definice *MUI_PAGE_CUSTOMFUNCTION_LEAVE*.
MUI_PAGE_DIRECTORY zaregistruje dialog výběru instalačního adresáře, který můžeme vidět na obrázku 3.8. Stejně tak makrem *MUI_PAGE_COMPONENTS* vložíme do instalátoru dialog pro výběr instalovaných komponent. Parametrem *MUI_PAGE_COMPONENTSPAGE_SMALLDESC* přenastavíme velikost popisů jednotlivých komponent na menší.
Následující makra se vztahují ke konečné obrazovce instalátoru.

Makra s názvem *RUN* přidávají zaškrtávací možnost spuštění testovacího skriptu a jeho názvu.

Makra s názvem *LINK* přidávají hypertextový odkaz na webové stránky projektu.

Na konec definujeme makrem *MUI_LANGUAGE* jazykové prostředí instalátoru. NSIS obsahuje podporu pro mnoho jazyků pro své základní dialogy.

V jazyce NSIS neexistují cykly, a proto vznikly moduly, které realizují specifickou činnost a pomocí callback funkcí realizují uživatelské události. Díky tomuto mechanismu bylo možno realizovat rekurzivní dynamické nastavování práv pro každý soubor. V tomto případě jde o funkci *Locate()* a její callback funkce můžeme vidět v příkladu 3.29.

Pomocí *Locate()* funkce je také implementována konverze plain-text konců řádků v souborech z kódování Unix na Windows[14].

př. 3.29: Callback funkce

```
54 Function FilePermissionCallback
55   SetFileAttributes $R9 FILE_ATTRIBUTE_NORMAL
56   AccessControl::GrantOnFile $R9 "(S-1-5-32-545)" "
         GenericRead + GenericWrite"
57   AccessControl::GrantOnFile $R9 "(S-1-1-0)" "GenericRead
       + GenericWrite"
58   AccessControl::GrantOnFile $R9 "(BU)" "GenericRead +
         GenericWrite"
59   Push $0
60 FunctionEnd

62 Function FileEndLineCallback
63   push $R9
64   call ConvertUnixNewLines
65   Push $0
66 FunctionEnd
```

Komentář

Funkce *FilePermissionCallback()* realizuje rekurzivní nastavování práv souborů. Tato funkce je volána pro každý nalezený soubor funkcí *Locate()*. Na začátku voláme funkci *SetFileAttributes()*, abychom nastavili normální parametry souboru pro případ kdyby atributy byly špatně nastaveny. Funkce *Locate()* předává cestu k aktuálnímu souboru v registru *$R9*.
Framework NSIS v základním API neřeší nastavování práv, tak je nutno využít *AccesControl* plugin. Pomocí něho můžeme nastavit každému souboru specifická práva funkcí *GrantOnFile()*. Pro soubory je nastavena práva na zápis a čtení. Funkcí *push* vracíme z funkce návratovou hodnotu.
Funkce *FileEndLineCallback()* implementuje volání *ConvertUnixNewLines()* pro převod konce řádků pro hledané soubory. Tato funkce je volně dostupná v repositářích NSIS.

[14]Na operačních systémech MS Windows jsou konce řádků signalizovány párem znaků *hex(0D a 0A)* neboli *dec(13 a 10)*. Na Unixových/Linuxových systémech je konec řádku dán pouze znakem *0A*.

NSIS nemá vestavěnou kontrolu vybraného instalačního adresáře, proto musíme chybové stavy ošetřit sami. V příkladu 3.30 vidíme rozhodovací blok, který je volán pří opuštění dialogu pro výběr adresáře z obrázku 3.8.

Definice aby se tato funkce volala při této specifické události můžeme vidět v příkladu 3.28 pomocí makra *MUI_PAGE_CUSTOMFUNCTION_LEAVE*. Aby NSIS poznal, ke kterému dialogu patří toto makro s ukazatelem na funkci, musíme ho definovat vždy před definici dialogu.

př. 3.30: Kontrola instalačního adresáře

```
142 Function dir_check
143   GetInstDirError $0
144   ${Switch} $0
145     ${Case} 1
146       MessageBox MB_OK|MB_ICONEXCLAMATION "
              Invalid_installation_directory!"
147       Abort
148       ${Break}
149     ${Case} 2
150       MessageBox MB_OK|MB_ICONEXCLAMATION "Not_
              enough_free_space!"
151       Abort
152   ${EndSwitch}
153   IfFileExists $INSTDIR +1 +3
154   MessageBox MB_YESNO|MB_ICONEXCLAMATION "
          Installation_directory_exists,_overwrite?" IDYES +2
          IDNO +1
155   abort
156   rmdir /r $INSTDIR
157 FunctionEnd
```

Komentář

Funkce *GetInstDirError()* ukládá do zásobníku na pozici 0 ($0) stav instalačního adresáře definovaného vnitřní proměnnou *$INSTDIR*.
Pokud je název adresáře nepřípustný nebo jeho cesta, tak funkce vrací stav 1 a poté je vykresleno výstražné dialogové okno funkcí *MessageBox()*.

Stav *2* označuje nedostatek volného místa na disku a je vykreslena výstražná zpráva.

Poslední kontrola hlídá zda instalační adresář již existuje (*IfFileExists()*), pokud ano vykreslí zprávu zda má data uvnitř adresáře přepsat. Pokud uživatel potvrdí, tak se adresář smaže *rmdir()*.

Můžeme zde vidět případ relativního adresování pro různé volby tlačítek.

Základem každého internetového instalatéru je schopnost stahovat data pomocí konvenčních protokolů ze sdílených úložišť, proto je v příkladu 3.31 implementována funkce pro stahování pomocí standardní knihovny *Inet*. Zprostředkování základních funkcí má na starost stejnojmenný modul pro NSIS *inetc*.

př. 3.31: Stahování souborů

```
159 function download
160     inetc::get /caption $0 $1 $2
161     Pop $0
162     StrCmp $0 "OK" dlok
163     MessageBox MB_OK|MB_ICONEXCLAMATION "
            Download_error,_click_OK_to_abort_installation" /SD
            IDOK
164     Abort
165   dlok:
166 functionend

168 function download_luacv
169   IntCmp $LUACV_IS_DOWNLOADED '1' dlskip dlbegin
          dlskip
170   dlbegin:
171   IfFileExists '$TEMP\luacv.tar.gz' extr
172   strcpy $0 "Downloading_LuaCV_package"
173   strcpy $1 "http://switch.dl.sourceforge.net/project/luacv/
          luacv.tar.gz"
174   strcpy $2 "$TEMP\luacv.tar.gz"
175   call download
176   extr:
177   untgz::extract -d "$TEMP" "$TEMP\luacv.tar.gz"
178   exec 'cmd_/c_"mkdir"_"$INSTDIR"'
179   IntOp $LUACV_IS_DOWNLOADED
          $LUACV_IS_DOWNLOADED + 1
180   dlskip:
181 functionend
```

Komentář

Funkce *download()* implementuje obecné stahování souboru parametry přenášenými pomocí zásobníku. Základem je knihovní funkce *Get()* z modulu *intec*, která realizuje samotné stahování. Funkce *pop* vyzvedne hodnotu ze zásobníku a tu pak pomocí *StrCmp()* zkontrolujeme, zda nenastala chyba. Pokud ano, zobrazí se varovný dialog *MessageBox()* a proces je zrušen *Abort()*. V případě že chyba nenastala skočí se na návěští *dlok*, aby se nespustila výstraha.

Funkce *download_luacv()* je postavená nad dříve definovanou funkcí *download*. Protože velká část aplikace je šířena dohromady v jednom archívu (*tarballu*) a jsou v instalaci v jednotlivých sekcích, je nutné ošetřit stav, že se archív stahuje znova pro každou část.
Funkcí *IntCmp()* kontrolujeme uživatelskou proměnnou na počet výskytů LuaCV archívu na disku. V případě, že počet výskytů je 0 nebo více než 1, tak se stahování přeskočí a program přejde na návěští *dlskip*.
Funkce *IfFileExists()* kontrolujeme, zda archív neexistuje z dřívějších instalací. Pokud ano, přejde se rovnou k extrahování dat.
Pomocí *StrCpy()* nakopírujeme požadované parametry pro funkci *download()* na zásobník a následně ji zavoláme *call()*. Poté pomocí funkce *extract()* z modulu *untgz* rozbalíme tarball do dočasného umístění *$TEMP* a vytvoříme instalační adresář *$INSTDIR*. Následně inkrementuje uživatelskou proměnnou, aby se archív znova nestahoval.

Vybrané instalační komponenty jsou definovaný v sekcích a jsou zobrazovány a spouštěny v pořadí tak, jak jsou za sebou ve zdrojovém kódu např. sekce v příkladu 3.32. V příkladu 3.33 můžeme vidět anonymní sekci, která není ve výběru komponent, ale její instrukce jsou spuštěny vždy. Tato sekce je na konci skriptu, a proto se spouští jako poslední.

př. 3.32: Instalační sekce

```
188 Section "!Windows binaries" Group1
189     SectionIn 1 2 3
190     AddSize 3760
191     strcpy $0 "Downloading LuaCV binaries"
192     strcpy $1 "http://switch.dl.sourceforge.net/project/luacv/
            Binaries/luacv.zip"
193     strcpy $2 "$TEMP\luacv.zip"
194     call download
195     ZipDLL::extractall "$TEMP\luacv.zip" "$INSTDIR\bin"
196     DetailPrint "LuaCV binaries succesfully installed"

198     strcpy $0 "Downloading OpenCV binaries"
199     strcpy $1 "http://switch.dl.sourceforge.net/project/luacv/
            Binaries/opencv.zip"
200     strcpy $2 "$TEMP\opencv.zip"
201     call download
202     ZipDLL::extractall "$TEMP\opencv.zip" "$INSTDIR\bin"
203     DetailPrint "OpenCV binaries succesfully installed"

205     Strcpy $0 "Downloading Lua interpreter"
206     strcpy $1 "http://switch.dl.sourceforge.net/project/luacv/
            Binaries/lua.zip"
207     Strcpy $2 "$TEMP\lua.zip"
208     call download
209     ZipDLL::extractall "$TEMP\lua.zip" "$INSTDIR\bin"
210     DetailPrint "Lua binaries succesfully installed"
211     DetailPrint "Appending Lua binaries to PATH
            environment variable"
212     ${EnvVarUpdate} $0 "PATH" "A" "HKCU" "$INSTDIR\
            bin"
213 SectionEnd
```

Komentář

Nadpis sekce obsahuje formátovací znak „!“, který NSIS překládá jako tučné písmo. Funkcí *SectionIn()* je nastavena příslušnost sekce k jednotlivým typům instalace jako *minimal*, *full* nebo *custom*. *AddSize()* udává velikost sekce v kbyte pro kontrolu velikosti místa na disku.

Následně zkopírujeme parametry *strcpy()* na stack, abychom mohli zavolat funkci *download()* pro stažení souboru. Na vrchol stacku je zkopírován popis operace, dále odkaz na soubor a na konec umístění, kam bude uložen.
Pomocí pluginu *ZipDll* a jeho funkce *extractall()* rozbalíme archiv do instalačního adresáře $*INSTDIR*.
Funkce *DetailPrint()* zobrazuje do instalačního výpisu informaci o prováděné operaci.

Takto jsou postupně staženy binární soubory pro LuaCV, Lua a OpenCV

Na konec je nastavena cesta interpretru do enviromentální proměnné *PATH* funkcí *EnvVarUpdate()*

př. 3.33: Po-instalační sekce

```
259 Section
260   IntCmp $LUACV_IS_DOWNLOADED '0' +1 +1 +3
261   call getTxtFiles
262   goto +2
263   call rename2txt

265   DetailPrint "Converting files to windows end line format."
266   ${Locate} "$INSTDIR" "/L=F /M=*.lua /G=1" "
          FileEndLineCallback"
267   IfErrors error2
268   ${Locate} "$INSTDIR" "/L=F /M=*.txt /G=1" "
          FileEndLineCallback"
269   IfErrors error2

271   ${Locate} "$INSTDIR" "/L=FD /M=*.*" "
          FilePermissionCallback"
272   DetailPrint "Setting file attributes and permissions."
273   IfErrors error1 +4
274   error1:
275   DetailPrint "Error in settings file attributes or
          permissions"
276   goto +2
277   error2:
278   DetailPrint "Error in converting endlines."

280   delete '$TEMP\luacv.tar.gz'
281   delete '$TEMP\lua.zip'
282   delete '$TEMP\luacv.zip'
283   delete '$TEMP\opencv.zip'
284 SectionEnd
```

Komentář

Tato sekce v instalátoru se stará o konečné nastavení parametrů stažených souborů. Nejprve se zkontroluje, zda byla stažena libovolná komponenta z tarballu LuaCV pomocí funkce *IntCmp()* a pokud ne, stáhnou se z repositáře textové soubory obsahující *licenci* a *readme* funkcí *getTxtFiles()*.
Pokud tarball byl stažen, tak přejmenuje textové soubory bez přípony **.txt* na tuto příponu pomocí funkce *rename2txt()*, aby je bylo možné ve MS Windows otevřít.
Funkcí *DetailPrint()* vypíšeme do instalačního okna zprávu viz obrázek 3.10.
Pomocí *Locate()* funkce vyhledáme všechny soubory **.lua*, **.txt* a poté na ně aplikuje callback funkci *FileEndLineCallback()* definovanou v příkladu 3.29.
Následně funcí *Locate()* vyhledáme všechny soubory, adresáře a aplikujeme na ně *FilePermissionCallback()* pro nastavení přístupových práv.
Ošetříme stavy, kdy nastaly chyby pomocí funkce *IfErrors()* a pokud nastaly, skočíme na návěští *error2*. Jestliže nenastaly chyby, skočíme pomocí funkce *Goto()* o dvě instrukce dále.

Na konec smažeme dočasné stažené soubory pomocí funkce *Delete()*.

3.3. Dokumentace

Vzhledem ke komplexnosti knihovny LuaCV bylo nutné vytvořit dokumentaci její Lua API, ale s ohledem k její velikosti nebylo možné ji vytvořit ručně. Také nešlo využít univerzálních generátorů dokumentace jako Doxygen z těchto důvodů:

- Prototypy C funkcí neodpovídají prototypům Lua funkcí. Všechny přemostěné funkce mají stejnou hlavičku *int funkce (lua_State *L)*.
- Přemostěné objekty nemají nikde definovanou stavbu kromě řetězce ve funkci *__tostring*. A parametry struktury jsou definovány pomocí funkcí uložených v seřazeném poli pro použití v indexování.
- Konstanty jsou definovány pouze v poli typu *luacv_var* a pomocí makra *varReg()* a ne dle standardních metod jako *#define*.
- Popis funkcí a jejich argumentů nejsou uloženy ve zdrojových souborech LuaCV, ale pouze v Latexové dokumentaci OpenCV.

Proto byl navržen generátor v jazyce Lua, který extrahuje data ze zdrojových kódů knihoven LuaCV, OpenCV a oficiální dokumentace OpenCV verze 2.2. Celý tento generátor vytváří Latexový zdrojový kód, který je na konec zkompilován do pdf pomocí *pdflatex*. Jako Latexová šablona byla využita OpenCV šablona použita v oficiální dokumentaci.

Celkový průběh generování dokumentace pak probíhá v následují posloupnosti:

- Extrakce řetězců *const char f_msg[]*, které jsou definovány u každé přemostěné funkce. Získání názvu Lua funkce a jejich parametrů, které jsou uloženy do tabulky.
- Vytvoření tabulky obsahující popisy funkcí a argumentů z Latexové dokumentace OpenCV 2.2. Ve většině případů jsou názvy funkcí a argumentů shodné, takže je lze porovnávat jako klíč tabulky.
- Získání názvů souborů objektů přemostěných v LuaCV, které jsou shodné s jejich originálními názvy. Tím je vytvořena tabulka názvů objektů.
- Extrakce atributů struktur ze zdrojových souborů LuaCV.
- Nalezení popisu struktur v OpenCV dokumentaci.
- Vytvoření seznamu použitých konstant v LuaCV z jejich modulů a následně procházení hlavičky knihovny OpenCV, pro jejich hodnoty.
- Přeložení všech vygenerovaných souborů do pdf pomocí *pdflatex*.

Takto vygenerovaná dokumentace obsahuje přes 300 stran popisu knihovny LuaCV. OpenCV knihovna od verze 2.3 přešla na nový systém generování dokumentace, a proto je přemostění LuaCV také nuceno postupně přejít, aby nebylo třeba používat dokumentaci starých verzí. To v důsledku znamená vytvoření nového generátoru nebo jeho razantní upravení.

3.4. LuaCV v kurzu MPOV

Pro otestování funkčnosti implementace přemostění LuaCV byl využit prostor v kurzu počítačového vidění MPOV. Cílem těchto aktivit bylo získat zpětnou vazbu od uživatelů, kteří s jazykem Lua a knihovnou OpenCV mají různé zkušenosti a tím prověřit funkčnost a použitelnost tohoto přemostění v běžném prostředí. Očekávalo se, že velkou překážkou bude jak odlišnost jazyka Lua, tak i způsob programování v OpenCV oproti prostředí Matlab. Aktivity okolo LuaCV v tomto kurzu se dělily na nepovinné cvičení a na úlohu v zápočtových projektech.

3.4.1. Nepovinná cvičení

Cvičení proběhlo v 8. týdnu semestru, bylo zaměřeno na vypracování základních operací používaných v počítačovém vidění. Protože nebylo možno předpokládat, že studenti budou schopni sami vypracovat celé řešení z důvodů hlubší znalosti jazyka Lua a knihovny OpenCV, byly jim poskytnuty skripty, ve kterých chyběly důležité části zajišťující stěžejní funkcionalitu úlohy. Tato místa v kódu byla stručně okomentována, tak aby bylo jasné, jakou funkci a argumenty použít. Smysl tohoto typu cvičení bylo, aby si studenti prohlédli, jak jazyk Lua a knihovna LuaCV funguje a na základě pozorování dokázali aplikovat chybějící metody.

Pro stručné seznámení byly na začátku každého cvičení prezentovány základní informace, možná záludná místa v návrhu a návod, jak spustit interpretr z příkazové řádky. Jako podpůrné materiály byly k dispozici kopie tzv. *lua cheatsheet* neboli souhrn nejdůležitějších vlastností, funkcí a syntaxe jazyka Lua spolu se stručným popisem použitých OpenCV rutin. Jako další zdroje byly používány oficiální OpenCV příklady přepracovaných do LuaCV, ze zdrojového balíčku.

Během tohoto cvičení se vystřídaly tři skupiny studentů a každá měla za úkol vypracovat jiné zadání z důvodů otestování většího obsahu funkcionality. Témata zadání byla vybrána na základě cvičení kurzu MPOV z předchozího roku. Tato zadání jsou popsána v kapitole 2.1.2 a jejich OpenCV varianty pak v 2.2.4.

Častou chybou studentů během vypracování bylo mylné pochopení Lua konstrukcí a aplikování zvyklostí z Matlabu, kdy se snažili procházet prvky matice pomocí operátoru *()*. Nicméně matice přemostěné knihovnou LuaCV jsou pouze ukazatele na paměť, takže tato činnost pro interpret znamenala pouze chybný pokus o zavolání *userdat*.

Další chybou bylo vynechávání prefixu *cv.* nebo *luacv.* před voláním LuaCV funkcí. Všechny ukazatele na LuaCV funkce jsou umístěny v globální tabulce *luacv*. Pro interpret to znamenalo chybné volání globální funkce, která neexistuje.

Mezi méně častými problémy studentů se také objevovalo zaměňování interaktivního módu interpretru jazyka Lua za příkazovou řádku *cmd.exe* z MS Windows. To vyúsťovalo ke snaze volat Lua kód v příkazové řádce nebo naopak. Další z méně častých problémů byla snaha o kopírování LuaCV struktur pomocí operátoru =, přičemž studenti docílili pouze zkopírování adresy v paměti do další Lua proměnné. Tento problém je dalším ze „zvyklostí“ přinesených z Matlabu, které LuaCV nepodporuje. Pro studenty může být matoucí, že nativní datové typy Lua tyto operátory mají implementované a přemostěné nikoliv.

Během cvičení nebyly nalezeny žádné chyby v implementaci LuaCV, ale to bylo dáno povahou cvičení, kde byla snaha o seznámení studentů s jazykem Lua a jeho využitím v

knihovně OpenCV. Každý ze studentů dokázal, ať už s pomocí nebo bez ní, vypracovat zadání.

3.4.2. Zápočtový projekt

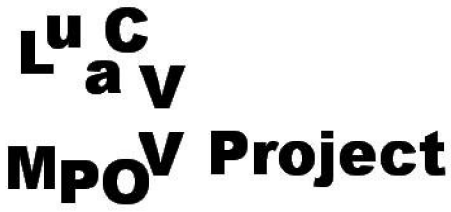
Jakmile studenti byli alespoň základně seznámeni s přemostěním LuaCV a jazykem Lua v nepovinném cvičení, byly zadány zápočtové projekty. Jedním z těchto projektů bylo i přepracování příkladů z celého kurzu MPOV do jazyka Lua. Tři dvojčlenné skupiny si toto zadání vybraly a vypracovaly. Přesné zadání můžeme vidět v kapitole níže.

3.4.3. Převod cvičení MPOV do LuaCV

Cílem projektu 6 je dodatečně realizovat všechna cvičení (1-5) z předmětu MPOV 2011 v jazyce LuaCV. Dále pak realizovat dvě další úlohy dle zadání vedoucího projektu a dále pak úlohy jsou vidět v seznamu níže.Ty pak otestujte na snímku 3.11:

- realizujte vlastní funkci pro indexování oblastí pracující obdobně jako funkce *bwlabel()* v Matlabu. Funkce vrátí indexovaný obraz a strukturu obsahující identifikátor ID oblasti a plochu.
- nastudujte a aplikujte funkci *cv.SetMouseCallback()*. Zobrazte snímek indexovaný pomocí funkce *bwlabel()* , pomocí níž zjistíte kolik objektů se v obraze vyskytuje a vhodným násobením jasu v obraze jej normalizujte na hodnoty *0-255*. Při kliknutí do obrazu vypište do konzole ID a velikost oblasti.

obr. 3.11: Vstupní obraz pro zadání dodatečných úloh



Pro zadání jednotlivých úkolů je níže vypracován jejich seznam a hrubý popis.

- *Analýza obrazu* - Cvičení se zabývalo aritmetickými operacemi nad obrazy, vytvoření histogramu a jeho ekvalizování.
- *Jasové transformace* - Navržení funkce pro transformaci obrazu, výpočet pomocí kumulovaného histogramu, kvantil intenzit, jasová korekce.
- *Geometrické transformace* - Implementace funkce rotace obrazu, navržení radonové transformace.
- *Hranové filtry* - Navržení funkce realizující Gaussovu 2D plochu, filtraci obrazu, realizování funkce Sobelova filtru.
- *Segmentace obrazu* - Prahování obrazu jedním prahem a pravděpodobnostní metodou.

3.4.4. Vypracování

V této kapitole je popsána úspěšnost jednotlivých skupin, jejich problémy a chyby v implementaci, které nalezly při vypracování zadání. Taktéž obsahuje i jejich připomínky pro zlepšení funkčnosti.

Citace závěrů studentů Síkory a Vítka[8]:

> Jeden z problémů bylo málo dokumentace k jednotlivým funkcím. Našli jsme sice na internetu popisy všech funkcí. Pokud ale ty funkce vidíte poprvé, tak jeden nebo dva řádky anglického textu nic moc neřeknou. Matematický popis co byl u některých funkcí, taky někdy moc nepřidal. Další problém byl, že hodně příkazů bylo potřeba napsat jinak, než bylo na zmíněných stránkách aniž bychom věděli jak. Některé příkazy byly sice v *cheatsheetu*, ale nenašli jsem tam všechno, co jsme chtěli použít. Dále byl problém při uvolňování paměti, což způsobovalo padání. Nevýhodou také bylo ladění programu jen pomocí funkce *print()*.
>
> Výhodu LuaCV, že má pracovat s obrázky rychleji než Matlab, jsme nestihli ověřit, výhoda oproti Matlabu je velikost celého programu. Velká část problémů bylo asi taky tím, že jsme měli z toho jazyka jenom jedno cvičení. Kdybychom s tím pracovali od začátku mohlo by to být lepší.

Rozbor připomínek:

- Dokumentace projektu LuaCV je generována z oficiální dokumentace OpenCV. Malé procento funkcí se svými parametry mohou mírně lišit, ale při jejich neplatném zavolání vždy program vypíše správnou formu. Takže ve většině případů lze použít oficiální dokumentaci, která obsahuje více detailů.

- O uvolňování paměti se stará *garbage collector*, který instance dat uklízí až tehdy, kdy na ně neexistuje žádná reference. Proto se může stát, že data zůstávají v paměti déle než uživatel zamýšlí, to je ale problém všech jazyků obsahujících GC. Nicméně pomocí sledování alokování dat v jednotlivých skriptech byla nalezena série podobných chyb v implementaci LuaCV, kde paměť zůstávala neuvolněná.

- Jazyk Lua obsahuje základní funkcionalitu pro tvorbu ladícího rozhraní. Nicméně v základní výbavě toto řešení není přímo použitelné pro samotné ladění, ale spíše pro návrh komplexnějšího debuggeru. Proto je vhodné použít některé z již existujících prostředí, které mají debugger implementovaný.

Citace závěrů studentů Voždy a Stehna[9]:

> V první úloze bylo úkolem seznámit se se základy zpracování obrazu. Aplikovali jsme tedy jednoduché algoritmy, jako vytvoření matice hodnot typu *uint8* alias obrazu s konstantní jasovou hodnotou *50*. Poté jsme tuto vytvořenou matici sčítali, odčítali a míchali s obrazem *Lena.bmp*, jenž byl součástí zadání. Následně byly napsány funkce pro zobrazení histogramu a jeho ekvalizaci. Zde bylo nutné zpracovat i barevnou verzi obrázku Leny, *LenaC.bmp*.
>
> Oproti Matlabu je zde především nutnost deklarovat dopředu matice, s nimiž se bude pracovat. Toto je ale spíše věc zvyku a pohodlnosti programátora. Dále by se nám líbila implementace operátoru indexace matice (např.

X[1,2]), jakožto elegantnějšího dvojníka k funkci *cv.Get*D()*. Pokud zde tato možnost je, nepodařilo se nám ji objevit.

Dále se nám nepodařilo přistoupit k jednotlivým kanálům barevného obrazu přes funkci *cv.SetImageCOI()* (set channel of interest). Vinu jsme přikládali nepochopení syntaxe z naší strany, a proto zvolili metodu rozložení obrazu do jednotlivých složek funkcí *cv.Split()*. Řešení v prostředí Matlab, kdy je barevný obraz de facto jen trojrozměrná matice, nám přijde elegantnější. Rovněž zpracování po jednotlivých složkách pak lze realizovat ve třech vložených for-cyklech společných pro barevný i šedotónový obraz, což přispívá k úhlednosti kódu. Tuto vlastnost však nepovažujeme za nijak zásadní. Druhým úkolem byly bodové jasové transformace. Šlo o napsání funkce pro úpravu obrazu dle dané tabulky look-up table). Poté jsme měli napsat funkci, hledající 1% a 99% kvantil v obrazu a následně obraz saturovat dle těchto kvantilů. Jelikož jsme již do programování v LuaCV trochu pronikli v minulé úloze, nezaznamenali jsme zde žádné větší překážky.

Jediné, co nás na chvíli zaujalo, bylo, jakým způsobem dochází k předávání výstupních parametrů z funkce. Při syntaxi x=funkce(y) se pravděpodobně vytváří jen mělká kopie globální proměnné vytvořené při volání funkce?

Třetím úkolem bylo vytvoření funkce pro rotaci obrazu a její následné využití k radonově transformaci obrázků s geometrickými primitivy, v němž jsme měli detekovat dominantní čáru.

S využitím interní knihovny matematických funkcí a funkce pro vykreslení čar do obrazu byl tento úkol splněn bez větších problémů.

Po našem upozornění objevil autor programu pan Prymus chybu ve funkcích *cv.Set*D()* a *cv.Get*D*, která způsobovala nadměrnou spotřebu operační paměti. Kvůli ní jsme museli zdrojový kód rozdělit na dva kratší, čímž jsme předešli pádu programu při zaplnění fyzické paměti PC. Chyba byla odhalena pouze díky existenci čtyřnásobně vnořeného for-cyklu v našem kódu, jedná se tedy o opravdu extrémní případ. Autor již tento nedostatek odstranil, lze tedy předpokládat zlepšení výkonu při výpočtu této úlohy. Čtvrtá úloha se zabývala návrhem hranového filtru. Měli jsme vytvořit 2D Gaussovu funkci a následně ji použít jako masku pro filtraci šumu. Následně bylo nutno napsat funkci pro filtraci obrazu s hysterezí.

V rozporu s původním zadáním jsme nevykreslovali Gaussovu křivku ve 3D grafu, v prostředí LuaCV by to byl úkol mimo naše schopnosti. Zaskočení jsme byli jen faktem, že aby filtrace *cv.Filter2D()* fungovala uspokojivě, je nutné mít masku deklarovanou jako matici proměnných typu 32 bitový float. Podobné nepříjemnosti s datovými konverzemi však máme v živé paměti i z MATLABu. Poslední úloha byla zaměřena na segmentaci objektů v obrazu. Šlo o návrh dvou funkcí – funkce pro iterativní stanovení hodnoty prahu a funkci pro barevnou segmentaci kůže v prostoru YCbCr.

Zajímavostí na tomto úkolu bylo využití callback funkce po kliknutí myši na obrázek. Tímto si může uživatel vybrat barvu pro segmentaci. Zde jsme museli napsat pro každý obrázek zvlášť callback funkci, nepřišli jsme totiž na to, jak by se dala do callback funkce předat matice s konkrétním obrázkem, který jsme potřebovali zpracovat.

V obecné rovině lze říct, že práce v LuaCV je přes mírně odlišnou syntaxi mixem MATLABu a C, po úvodním proniknutí do problematiky nám tedy již návrh programu nečinil větší potíže.

Vzhledem k nedostatku času jsme se nemohli seznámit s debuggerem, jako primitivní *debugger* jsme používali výstup do konzole. Jak je (snad) vidět, v takto jednoduchých případech plní službu i on. Všimli jsme si rovněž, že ve většině výpočtů je LuaCV svižnější, než MATLAB.

Rozbor připomínek:

- Indexace typu *CvMat* nebo *IplImage* pomocí operátoru *[]* by byla možná, ale tím by knihovna LuaCV přestala být kompatibilní s OpenCV C API.
- Problém *cv.SetImageCOI()* funkce je v tom, že velká část OpenCV funkcionality tuto vlastnost ignoruje, protože se nalézá pouze u typu *IplImage* a nikoliv i u matic *CvMat*. A proto spousta funkcí na tuto vlastnost nemůže spoléhat, protože argumentem funkce může být jak obrázek tak matice. Nicméně většina funkcí z modulu *Core* ji podporuje a může být užitečná. Pokud tedy potřebujeme extrahovat celý kanál, tak musíme použít *cv.Split()* nebo *cv.Copy()* s COI. Pokud ale je třeba pouze zápisu do různých kanálů, je vhodné použít *cv.Set2D()*, jehož parametrem je typ *CvScalar*, který umožňuje zapsat do více kanálů najednou.
- Problém mělké kopie, jak je popsán může nastat ve chvíli, kdy OpenCV funkce má za parametr ukazatel na objekt a tentýž objekt pak také vrací. Tím pádem v Lua vzniknou dvě proměnné, které vedou na stejnou adresu v paměti.
- Problém uvolňování paměti vznikal u funkcí *cv.Set*D()* a *cv.Get*D()* z toho důvodu, že to jsou pouze deriváty funkcí *cv.Ptr*()*. Proto se na první pohled zdálo, že výstupy z těchto funkcí jsou pouze lokální proměnné, ale nikoliv. Bylo tedy nutné u těchto funkcí provést ruční odalokování. Protože typ *CvScalar*, se kterým se pracuje v těchto funkcích je malý, tak objevení této chyby chtělo opravdu hodně vnořených cyklů.
- OpenCV nepodporuje vykreslování 3D grafů, proto je nutné využít externích řešení typu *GNUPlot*, které fungují bez problémů na všech majoritních platformách. Pokud jde přímo o GNUPlot, tak dokáže číst z „roury“, a proto je jeho využití jednoduché i bez jakéhokoliv speciálního Lua API. Můžeme použít knihovní funkci *io.popen()* a dále pracovat jako se souborem pro zápis viz kapitola 3.24.
- Už samotná podpora Callback funkcí pro myš nebo trackbary je v Lua složitá a musí se obcházet pomocí jejich ekvivalentních funkcí, které umožňují předat do funkce libovolný parametr, kterým zavedeme informace o Lua zásobníku místo uživatelského parametru. Aby bylo možné přidat další libovolný uživatelský parametr, tak by to požadovalo velkou složitost návrhu vnitřních mechanismů.
- Jak už bylo napsáno výše, Lua v základu podporuje pouze Debug API nikoli uživatelské ladící prostředí, a proto je nutné použít již některý z existujících nástrojů.

Citace závěrů studentů Kulky a Labudka[10]:

> U každého cvičení jsme přepsali do LuaCV všechny úkoly. Bohužel u cvičení č. 3 funkci ImageRadon nemáme odzkoušenou kvůli obrovské spotřebě operační paměti a u cvičení č. 5 se nepodařilo implementovat načítaní souřadnic z obrázku. Místo toho jsou zadány pevné souřadnice bodu v obrázku, který má barvu tváře dítěte. Ve cvičení č.3 je funkce *ImageRadon()* zakomentována.
>
> LuaCV při spuštění námi vytvořených kodu používá obrovské množství operační paměti. A to je asi největší problém našich programů nebo LuaCV.
>
> Zajímavé by bylo porovnat stejné úlohy vytvořené v jazyce C + OpenCV s úlohami vytvořenými v LuaCV s pohledu rychlosti a využitím operační paměti.

Rozbor připomínek:

- Funkce ImageRadon nefungovala z výše popsaného bugu ve funkce *cv.Get*D()* a *cv.Set*D()*.
- Zadávání souřadnic z obrázku je možné pomocí funkce *cv.SetMouseCallback()*.
- Je jasné že nativní kód v C++ by byl vždy rychlejší než implementace v Lua z důvodů další aplikační vrstvy. Nicméně rychlost instrukcí v Lua se blíží rychlosti nativního kódu, díky malé režie jazyka Lua.

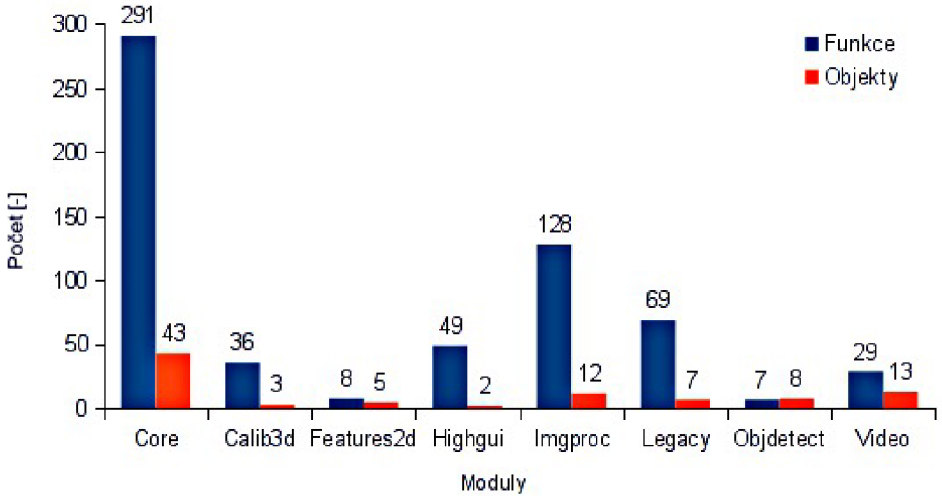
4. Zhodnocení dosažených výsledků

Účelem této kapitoly je shrnout vlastnosti a úspěchy knihovny LuaCV a stručně zopakovat implementovanou funkcionalitu a nástroje, které byly nutné při vývoji knihovny. V této kapitole je také porovnávána rychlost implementace knihovny LuaCV s nativním kódem OpenCV a přemostěním do jazyka Python dodávaného v oficiální distribuci. Toto měření proběhlo hardwaru Intel Celeron 1.7 GHz, 2 Gb DDRAM2 a na platformě GNU Linux x86. Na konec je shrnuta úspěšnost použití knihovny LuaCV a jejího příslušenství v kurzu MPOV a využití LuaCV při vypracování zápočtových projektů do tohoto kurzu.

4.1. Zhodnocení knihovny LuaCV

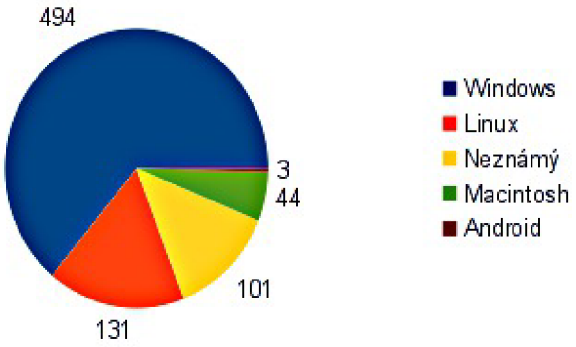
Cílem přemostění LuaCV bylo implementovat co nejvíce funkcionality z OpenCV C API. OpenCV knihovna v době začátku implementace LuaCV majoritně používala C API oproti C++ API, do budoucna se ale bude tento trend obracet. Knihovna LuaCV implementuje téměř celé OpenCV C API a v obrázku 4.1 lze vidět počty implementovaných funkcí a struktur. Dohromady byl vytvořeno přes 600 funkcí a přes 90 objektů.

obr. 4.1: Počet implementovaných funkcí a objektů



Pro snadnější distribuci bylo nutnou použít podpůrné programy. V rámci multiplatformní kompilace byl zvolen program CMake, který vytváří soubory nutné pro různá vývojová prostředí. Protože většina uživatelů LuaCV používá platformu MS Windows viz obrázek 4.1, byl navržen instalátor ve frameworku NSIS, který realizuje internetový instalační mechanismus.

obr. 4.2: Používané platformy s LuaCV



Postupem času, jak knihovna LuaCV obsahovala více a více OpenCV funkcionality, bylo nepřehledné dívat se na konkrétní syntax funkcí a objektů do zdrojového kódu LuaCV. Z toho důvodu byl navržen generátor

dokumentace, který generuje Latexové zdrojové soubory. Ty jsou poté zkompilovány Latexovým kompilátorem *pdflatex*. Tato dokumentace dosahuje rozsahu přes 300 stran a jsou v ní obsaženy informace o funkcích dodávané s knihovnou OpenCV.

Jako vhodnou prezentaci LuaCV knihovny byly vybrány populární softwarové portály SourceForge a SoftPedia a z těchto webů bylo LuaCV staženo více než 700x, viz obrázek 4.2 z velkého množství států. Nejčastěji byla stahována z Číny a České republiky, viz obrázek 4.3, jak ukazuje stupnice šedi. Počty stažení z Česka zkreslují mé vlastní testy instalátorů a ověřování funkčnosti distribuovaného balíku.

obr. 4.3: Oblasti s nejvyšším počtem stažení

Knihovna LuaCV byla také prezentována na studentské soutěži EEICT 2012 pořádané fakultami FEKT a FIT na VUT v Brně, kde byla úspěšně obhájena a byla zařazena do oficiálního sborníku soutěže[24].

4.2. Porovnání výkonnosti OpenCV a jeho přemostění

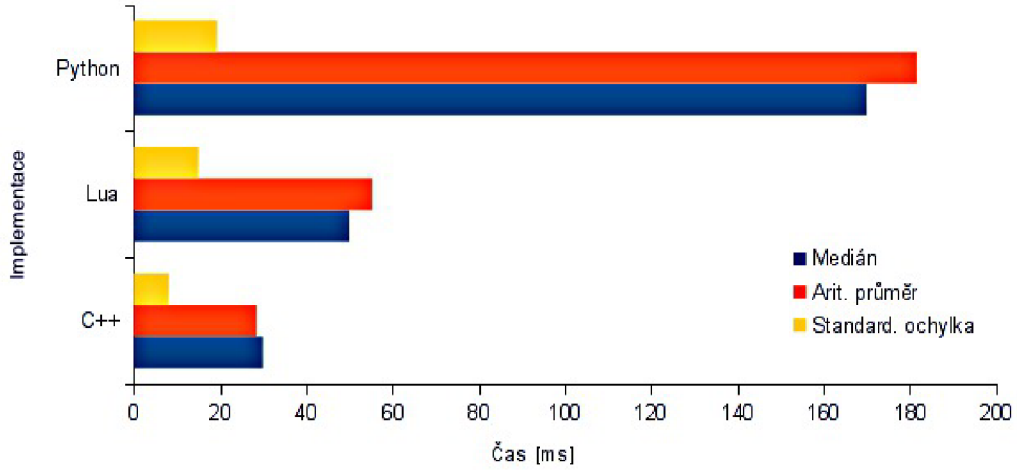
Tato kapitola porovnává výkonnost jednotlivých implementaci knihovny OpenCV v jazycích C/C++, Lua a Python. Přemostění v jazyce Python je automaticky generováno pomocí generátoru SWIG a je dodáváno v oficiální distribuci knihovny OpenCV. Cílem měření bylo zjistit dobu trvání testovacího programu, který byl napsán pro každý jazyk zvlášť, kde testovací programy jednotlivých implementací používají stejné funkce. Testovací program realizuje načítání zdrojového obrazu ze souboru z modulu *highgui*, vytváří pomocné obrazy a postupně nad nimi provádí matematické operace z modulu *core*, a poté aplikuje „Cannyho detektor“ z modulu *imgproc*.

Měření bylo sestaveno ze 100 vzorků pro každou implementaci, u kterých byla doba průběhu programu měřena konzolovým příkazem *time* dostupného na všech UNIXových platformách, tento příkaz měří s přesností na desítky milisekund. V tabulce 4.1 lze vidět výsledky jednotlivých měření a na obrázku 4.4 grafickou reprezentaci. Měření se bude na různých platformách a hardwaru lišit, v tomto případě jde spíše o prezentaci poměru výkonnosti jednotlivých implementací.

tab. 4.1: Výsledky výkonnosti implementací OpenCV

Jazyk	**Naměřené hodnoty [ms]**		
	Medián	*Arit. průměr*	*Standard. ochylka*
C++	30	28.5	8.21
Lua	50	55.3	15.07
Python	170	181.7	19.34

obr. 4.4: Graf výkonnosti jednotlivých implementací



Z naměřených dat jde jasně vidět, že poměr výkonnosti přemostění LuaCV je o 66% pomalejší než C/C++. Poměr výkonnosti přemostění v Pythonu k nativnímu C/C++ je pomalejší o 466%. LuaCV je tedy více než $3x$ rychlejší než Python.

Tímto měřením byla dokázána správnost výběru skriptovacího jazyka Lua pro tvorbu přemostění vzhledem k jeho nízké režii a paměťovým nárokům oproti jiným populárním skriptovacím jazykům. Na výsledky měření mělo také vliv, že LuaCV je implementována ručně, zatímco přemostění v Python je generováno pomocí SWIG.

4.3. Shrnutí použití LuaCV v kurzu MPOV

Nejvíce problémů studentům dělal přechod mezi jazyky Matlab a Lua respektive, styl jejich API, kdy se snažili aplikovat konstrukce vlastní pro Matlab do Lua skriptů. Dále studenti nepřímo odhalili chybu v implementaci v *cv.Set*D()* a *cv.Get*D()* funkcích, kde docházelo k neuvolňování paměti. Tento problém byl viditelný až tehdy, když studenti navrhli algoritmus, který obsahoval čtyřikrát zanořený *for* cyklus. Studentům taktéž chyběla pokročilejší funkcionalita pro vykreslování grafů či obrázků, byť toto není cíl ani jedné z knihoven, a proto je třeba využít programů třetích stran. Za zmínku jistě stojí program GNUPlot, který tyto operací z velké části řeší, a jeho integrace do LuaCV skriptů je velmi jednoduchá, jak bylo ukázáno v kapitole 3.24.

Jako další problém se také jevila dokumentace LuaCV. Ta je generována z Lua pomocí zdrojových kódů LuaCV a OpenCV . Proto je nepraktické psát dokumentaci pouze k LuaCV, když se dokumentace pro OpenCV s každou verzí stále vyvíjí. V této chvíli jsou generovány funkce, objekty a konstanty z LuaCV, kterým jsou jim přiřazeny popisy z dokumentace OpenCV viz kapitola 3.3.

Poslední velkou překážkou byla absence ladícího programu a vývojového prostředí. Takových prostředí existuje celá řad, jsou volně stažitelná a jsou kompatibilní s knihovnou LuaCV. LuaCV obsahuje pouze základní vysvětlování chyb, a proto pro komplexní hledání chyb v samotném návrhu algoritmu není vhodné.

I přes výše jmenované problémy byli všichni studenti schopni projekty vypracovat a doladit do funkčního stavu. Dokázali tím funkčnost knihovny LuaCV jako takové.

5. Závěr

V úvodu této práce jsem si kladl za cíl porovnat dosažené výsledky s prvotními předpoklady.

V kapitole teoretického úvodu jsem se věnoval předmětu MPOV a základům matematických aparátů používanými v tomto kurzu. V rámci kurzu byly realizovány prototypové úlohy dle zadání a implementovány v prostředí Matlab. Tyto úlohy byly vybrány na základě výše popsané látky. Konkrétně byly navrženy příklady na segmentaci pomocí prahování obrazu, diskrétní konvoluci a její použití s hranovými operátory a morfologické operace viz kapitola 2.1.2. Následná kapitola byla věnována Open Source knihovně OpenCV a jejímu softwarovému výbavení. V kapitole 2.2 byla shrnuta krátká historie, vývoj knihovny a zároveň i porovnání výkonnosti s konkurenčními produkty. Dále byly rozebrány strukturální změny mezi verzemi knihovny OpenCV a zároveň byla nastíněna možnost použití multiplatformní grafické knihovny Qt, která v budoucnu nahradí nativní grafické toolkity na jednotlivých platformách z důvodu větší kompatibility a čistotě implementace.

Pak bylo popsáno aktuální rozložení funkcí v modulech knihovny a některé jejich ukázky použití v příkladech. Pro každý modul byl implementován alespoň jeden charakteristický příklad. Ke konci kapitoly o knihovně OpenCV byly vypracovány příklady kurzu MPOV v jazyce C++. Ty jasně ukázaly, že lze za cenu mírně složitějšího zdrojového kódu získat vyšší výkon oproti použití prostředí Matlab. Tyto příklady slouží k porovnání složitosti implementace.

V kapitole 2.3 byly ukázány základy jazyka Lua a její syntaxi pro tvorbu programů. Na jednoduchém příkladu 2.21 realizujícím matematickou morfologickou operaci eroze jsou ukázány různé zápisy algoritmů, tak aby to vedlo k lepšímu pochopení konstrukcí používaných v jazyku Lua. Dále byly předneseny výhody jazyka Lua a její charakteristické vlastnosti, mezi které patří zejména malá režie, rychlost instrukcí, garbage collector nebo už samotná podstata skriptovacího jazyka. Jelikož na konci roku 2011 po téměř pěti letech vývoje vyšla nová verze Lua 5.2, jsou stručně popsány zásadní změny. Mezi nejviditelnější rozdíly patří například podpora generativního módu garbage collectoru a přidání nové knihovny realizující bitové operace. Protože nejdůležitější C API zůstalo zachováno je knihovna LuaCV po drobných úpravách schopna překladu i v nové stabilní verzi.

Kapitola realizovaná řešení popisuje knihovnu LuaCV, která byla vyvinuta jako přemostění OpenCV do jazyka Lua. Samotný návrh přemostění LuaCV je implementován v jazyce C++ z důvodu, že OpenCV kompilovaná pod C++ obsahuje větší funkcionalitu. V rámci této práce byly implementovány téměř všechny funkce a struktury z OpenCV C API. V důsledku to znamená více než 600 funkcí a 90 objektů. Velkou předností oproti C/C++ je možnost použití prvků funkcionálního programování dané v samotné Lua. LuaCV je navržena tak, aby byla co nejjednodušeji rozšiřitelná. V rámci otestování implementované funkcionality byly vypracovány téměř všechny oficiální příklady C API z knihovny OpenCV. LuaCV od verze 0.2.0 začíná implementovat první objekty z OpenCV C++ API, ale vývoj je ještě v počátcích.

Pro snadnější kompilaci byl použit program *CMake*, který generuje projekty nebo soubory Makefile používanými běžnými překladači jazyka C++. Tím je kompilace snadná na všech běžných platformách. Pro snadnou instalaci a správu knihovny LuaCV byl navržen instalátor pomocí frameworku NSIS formou internetového instalátoru. Ten stahuje různé komponenty LuaCV přímo z repositáře projektu SourceForge, kde je projekt hostován.

Výhodou SourceForge je také snadný přístup k vývojovým verzím LuaCV skrze webové SVN API a statistiky návštěvnosti.

Vzhledem k velikosti knihovny LuaCV bylo nutné vytvořit kvalitní dokumentaci. K tomuto účelu se bohužel nedaly využít univerzální generátory dokumentace jako např. Doxygen, a proto musel být generátor navržen ručně. Tento generátor čerpá informace ze zdrojových souborů knihovny LuaCV, OpenCV a Latexových souborů oficiální dokumentace OpenCV. Takto vygenerovaná dokumentace přesahuje rozsah 300 stran.

LuaCV byla do značné míry otestována v kurzu počítačového vidění MPOV. Nejprve proběhlo nepovinné cvičení v osmém týdnu semestru, kde bez větších obtíží všichni studenti dokázali vypracovat zadané cvičení. Mezi nejčastější problémy patřilo nepochopení syntaxe jazyka Lua ze strany studentů a její kombinace s knihovnou LuaCV. V následující kapitole můžeme vidět detailní rozbor toho cvičení.

Poté byla knihovna testována v rámci zápočtových prací kurzu MPOV, kde bylo úkolem vypracovat předešlá cvičení toho kurzu do jazyka Lua a knihovny LuaCV. Všechny tři skupiny projekt úspěšně vypracovaly a díky jejich algoritmům byly nalezeny chyby v implementaci knihovny. Tyto chyby byly triviálního charakteru, ale způsobovaly při velkém počtu zacyklení (čtyři zanořené cykly) velký nárůst nároků na paměť. Konkrétně v úloze „Radonovy transformace“ narostly nároky až v v řádu Gb. Mimo tyto chyby studenti postrádali komfort vývojového prostředí Matlab, jeho obrovskou funkcionalitu, funkce pro 2D a 3D tisk grafů a vykreslení více obrázků do jednoho okna. Nicméně tyto funkce chybí již v samotné implementaci OpenCV knihovny a zákonitě chybí i v LuaCV. Funkcionalita může být nahrazena produkty třetích stran, například GNUPlot. Pokud jde o vývojové prostředí kompatibilní s jazykem Lua a knihovnou LuaCV, lze na internetu nalézt mnoho IDE jako například Decode, LuaEdit nebo Eclipse.

V poslední kapitole byly shrnuty vlastnosti a implementované řešení knihovny LuaCV. Z těchto výsledků plyne, že LuaCV je nejvíce stahována z Číny a České republiky pro platformu MS Windows. Následně bylo realizováno měření výkonnosti implementace LuaCV oproti nativnímu kódu a oficiálního přemostění v Pythonu. Výsledky ukázaly, že LuaCV je oproti nativnímu kódu pomalejší o 66%, ale oproti Pythonu je minimálně $3x$ rychlejší. Dále je zřejmé, že přemostění LuaCV je konkurenčně schopné vůči Pythonu a je použitelné pro realizaci úloh počítačového vidění, matematické algebry nebo zpracování obrazu.

Potenciálem knihovny LuaCV by mohlo být zařazení do oficiální distribuce OpenCV jako přemostění pro jazyk Lua, ale zatím se tohoto cíle nepodařilo dosáhnout.

Knihovna LuaCV byla prezentována na studentské soutěži EEICT, kde byla úspěšně obhájena a byla zařazena do oficiálního sborníku soutěže.

Literatura

[1] ŠONKA, M., HLAVÁČ, V. *Počítačové vidění* Praha: Grada 1992. ISBN 80-85424-67-3.

[2] ŽÁRA, J., BENEŠ, B., FELKEL, P. *Moderní počítačová grafika* Praha: Computer press 1998. ISBN 80-7226-049-9

[3] HLAVÁČ, V., SEDLÁČEK, M. *Zpracování signálů a obrazů* Praha: ČVUT 2001

[4] HORÁK, K. a kol. *Elektronické texty ke kurzu Počítačové vidění MPOV* Brno: VUT 2008

[5] SONKA, M., HLAVAC, V., BOYLE, R. *Image processing, analysis and machine vision* London: Thomson Learning 2008. ISBN 0-495-24438-4

[6] BRADSKI, G., KAEHLER, A. *Learning OpenCV* Sebastopol: O'Reilly Media 2008. ISBN 978-0-596-51613-0

[7] HORÁK, K. *Počítačové vidění, počítačová cvičení* Brno: VUT 2010

[8] VITKO, P., SIKORA J. *Protokol zápočtového projektu kurzu Počítačového vidění, počítačová cvičení*

[9] VOŽDA, O., STEHNO D. *Protokol zápočtového projektu kurzu Počítačového vidění, počítačová cvičení*

[10] KULKA, B., LABUDEK D. *Protokol zápočtového projektu kurzu Počítačového vidění, počítačová cvičení*

[11] MARTIN, K., HOFFMAN B. *Mastering CMake: a cross-platform build system.* Kitwarem 2003. ISBN 978-1-930-93409-2

[12] MECKLENBURG, R. *Managing Projects with GNU Make.* O'Reilly Media Inc., 2005. ISBN 978-0-596-00610-5

[13] FRIEDL, J. *Mastering Regular Expressions.* O'Reilly Media Inc., 2006. ISBN 978-0-596-55899-4

[14] LAURENT, A.*Understanding Open Source and Free Software Licensign.* O'Reilly Media Inc., 2004. ISBN 0-596-00581-4

[15] IERUSAMLISCHY, R., DE FIGUEIREDO, L. H., CELES, W. *Lua 5.1 Reference Manual.* Lua.org, 2006. ISBN 85-903798-3-3

[16] IERUSAMLISCHY, R., DE FIGUEIREDO, L. H., CELES, W. *The evolution of Lua.* ACM HOPL III, 2007. ISBN 978-1-59593-766-X

[17] IERUSAMLISCHY, R. *Programming in Lua.* Lua.org, 2006. ISBN 85-903798-2-5

[18] IERUSAMLISCHY, R., DE FIGUEIREDO, L. H., CELES, W. *Lua Programming Gems.* Lua.org, 2008. ISBN 978-85-9037798-4-3

[19] BRADSKI, G., KAEHLER, A. *Learning OpenCV.* O'Reilly Media, Inc., 2008. ISBN 978-0-596-51613-0

[20] GOUGH, B. J., STALLMAN, R. M. *An Introduction to GCC.* Network Theory Limited, 2005. ISBN 0-9541617-9-3.

[21] IERUSAMLISCHY, R., DE FIGUEIREDO, L. H., CELES, W. *Semish'94 paper* [online]. 1994, poslední revize 26. 8. 2009 [cit. 2010-03-12]. Dostupné z: <http://www.lua.org/semish94.html>.

[22] *OpenCV reference manual* [online]. 2009, poslední revize 25. 3. 2010 [cit. 2010-04-1]. Dostupné z: <http://opencv.willowgarage.com/documentation/>.

[23] HONZÍK, J. *Algoritmy IAL, Studijní opora* ,2009

[24] *Proceedings of the 18th Conference STUDENT EEICT 2012 Volume 2*, Vysoké učení technické v Brně, 2012, ISBN 978-80-214-4461-4