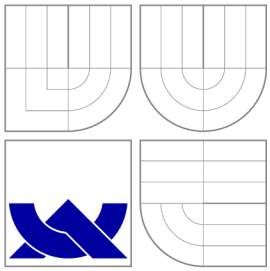
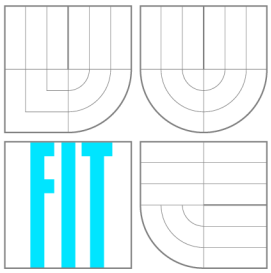


VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

TUTORIÁL PRÁCE S OPENCV

TUTORIAL OF OPENCV

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ BĚHAL

VEDOUCÍ PRÁCE
SUPERVISOR

Ing. VÍTĚZSLAV BERAN

BRNO 2010

Abstrakt

Tato bakalářská práce se zabývá identifikací a popisem základní činnosti doposud nezdokumentovaných metod v knihovně OpenCV. Konkrétně jsou zde popsány algoritmy FAST corner detector, Maximally stable extremal regions, LDetector, HOG people and object detector a One-way descriptor. Každá metoda obsahuje teoretickou část, deklaraci funkcí s popisem jejich parametrů a v neposlední řadě rovněž příklad použití. Pro každou z metod byla také vytvořena vzorová demo aplikace.

Abstract

This bachelor thesis deals with identification and description of undocumented algorithms in OpenCV library. It contains description of FAST corner detector, Maximally stable extremal regions, LDetector, HOG people and object detector and One-way descriptor. Each method description can be divided into parts as: Theory part, declaration of method with arguments description and also code example. Demo application has been created for each of these methods.

Klíčová slova

OpenCV, FAST corner detektor, Maximálně stabilní extrémní regiony, LDetector, HOG detektor osob a objektů, One-way deskriptor

Keywords

OpenCV, FAST corner detector, Maximally stable extremal regions, LDetector, HOG people and object detector, One-way descriptor

Citace

Lukáš Běhal: Tutoriál práce s OpenCV, bakalářská práce, Brno, FIT VUT v Brně, 2010

Tutoriál práce s OpenCV

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Vítězslava Berana.

.....
Lukáš Běhal
19. května 2010

Poděkování

Velmi rád bych poděkoval vedoucímu mé bakalářské práce Ing. Vítězslavu Beranovi za odborné vedení, připomínky a cenné rady, které mi během tvorby této práce poskytl.

Obsah

Kapitola 1

Úvod

S neustálým zdokonalováním výpočetní a video techniky dochází k jejímu nasazení v mnoha různorodých odvětvích, kde je velmi často využíváno počítačového vidění. Počítačové vidění je věda, která se snaží technicky napodobit lidské vidění, přičemž získává informace z obrazů. Obrazová data se mohou vyskytovat v mnoha formách, např. jako jednotlivé obrázky, videosekvence či pohledy z více kamer. Navrženy byly systémy schopny řídit procesy, detekovat události nebo také modelovat objekty.

V této době již existuje nespočet knihoven pro počítačové vidění, jako jsou například Vigra, NCV[1], Blepo a v neposlední řadě také OpenCV, na kterou je zaměřena tato bakalářská práce.

1.1 OpenCV knihovna

OpenCV[2] je volně šiřitelná knihovna pro počítačové vidění, která obsahuje velké množství metod a algoritmů pro zpracovávání reálně běžícího videa. Tato knihovna byla vyvinuta společností Intel, je napsána v jazyce C a C++ a pracuje pod operačními systémy Linux, Windows i Mac OS. Knihovna OpenCV je šířena pod BSD[3] licencí a její první alpha verze byla vydána v lednu roku 1999.

Jedním z klíčových cílů OpenCV je vytvořit jednoduše použitelnou infrastrukturu algoritmů počítačového vidění. Knihovna nyní obsahuje více jak 500 funkcí z mnoha odvětví počítačového vidění. OpenCV taktéž zahrnuje knihovnu pro strojové učení[4], jelikož ta je s počítačovým viděním úzce spjata.[2]

1.2 Cíl práce

Cílem této práce je identifikovat a popsat základní činnosti prováděné při práci s knihovnou OpenCV. V září roku 2009 byla vydána nová verze 2.0[5], kde se nacházela spousta nových, leč nezdokumentovaných metod. Po konzultaci s vedoucím jsme se dohodli, že bude vhodné se pokusit zdokumentovat některé z nich. Konkrétně jsem se zaměřil na detektory význačných bodů a objektů.

[1] Nokia CV

[2] Open Source Computer Vision

[3] **Berkeley Software Distribution licence** umožňuje volné šíření licencovaného obsahu, přičemž požaduje pouze uvedení autora a informaci o licenci včetně informace o zřeknutí se odpovědnosti za dílo.

[4] Machine Learning Library

[5] http://opencv.willowgarage.com/wiki/OpenCV%20Change%20Logs

Současně bylo mým úkolem k dokumentovaným metodám vytvořit programy, které je budou využívat, a demonstrovat jejich činnost. Výstupem mé práce by měl být tutoriál sestávající z více příkladů umístěný na web, což by usnadnilo přístup k informacím pro ostatní uživatele OpenCV.

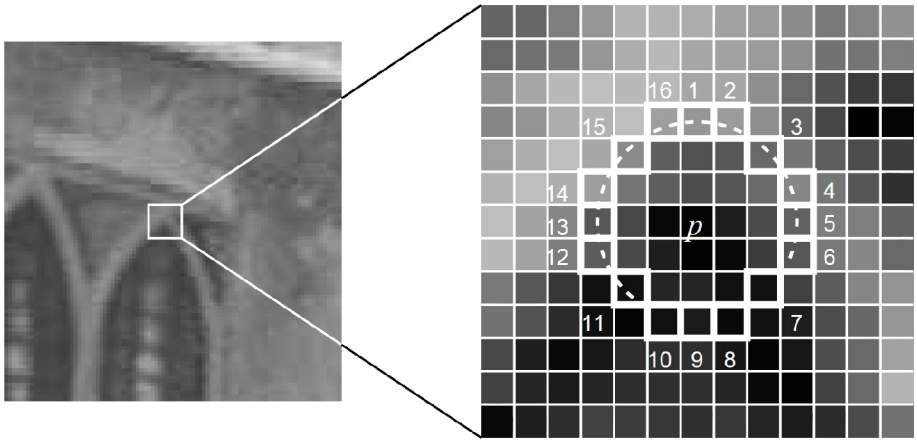
Práce je pojata poněkud netradičně, jelikož ji nelze jako standardní bakalářské práce rozdělit na část ryze teoretickou a ryze praktickou. Každá z kapitol 2 až 6 je věnována jedné z mnou dokumentovaných metod, přičemž obsahuje teoretický úvod, deklaraci funkcí dané metody a také její příklad použití. V poslední 7. kapitole uvádím návrh demo aplikace a její ovládání.

Kapitola 2

FAST corner detector

Detektory význačných bodů jsou ve většině případů v praxi používány v aplikacích běžících v reálném čase. Z tohoto důvodu je jedním z hlavních požadavků požadavek na rychlost detektoru. Edward Rosten a Tom Drummond vytvořili nový detektor, který je ve srovnání s ostatními detektory značně rychlejší. Detektor je schopen zpracovávat reálně běžící video ve formátu PAL, přičemž vyžaduje *méně jak 7%* procesorového času. V porovnání s ostatními detektory není schopna většina z nich video zpracovat. (Harris detektor 115%, SIFT 195%) Informace, vztahující se k FAST detektoru popsané v této kapitole, byly čerpány z [11],[12] a [13].

Hlavní podstatou FAST[1] detektoru je segmentový test, který pracuje s *Bressenhamovým kruhem* šestnácti pixelů soustředěného okolo bodu p považovaného za roh. Bod p je prohlášen za roh, pokud existuje alespoň n sousedících bodů, které jsou světlejší než součet intenzity testovaného bodu I_p a prahu t, nebo naopak tmavší než $I_p - t$, jak je zobrazeno na obrázku 2.1. Za n se obvykle volí číslo dvanáct, protože při menším n se vyskytuje hodně bodů, které rohy nejsou.



Obrázek 2.1: Znázornění segmentového testu rohové detekce v části obrázku. Vyznačené čtverce jsou pixely použité při rohové detekci. Pixel p je právě testovaný bod považovaný za roh.[12]

[1] Features from Accelerated Segment Test

Jako první jsou prověřovány body 1 a 9. Jestliže jsou intenzity obou pixelů v mezích $I_p + t$ resp. $I_p - t$, tento bod není rohem. Naopak je však stále možné, že testovaný bod může být roh, proto se dále prověřují body 5 a 13. K tomu, aby mohl být bod označený za rohový, musí být nejméně 3 z těchto 4 otestovaných pixelů světlejší než $I_p + t$, nebo naopak tmavší než $I_p - t$. Pokud je splněna i tahle podmínka, jsou již otestovány všechny ostatní pixely. Tento typ detektoru vykazuje podle [12] vysoký výkon, má však některé slabiny:

1. Tento přístup neodmítne v krátkém čase tolik kandidátů pro $n < 12$, protože v tomhle případě stačí, aby byly pouze 2 ze 4 pixelů výrazně světlejší, či tmavší.
2. Výkon detektoru závisí na pořadí prověřování jednotlivých bodů, ležících na Bresenhamově kružnici. Je nepravděpodobné, že pořadí 1,9,5,13 je optimální.
3. Je zjištěno více význačných bodů blízko sebe.

2.1 Zvýšení obecnosti a rychlosti za pomoci strojového učení

Proces se skládá ze dvou kroků. Nejprve jsou rohy detekovány ze sady obrázků, kde se používá pomalý algoritmus, který jednoduše testuje všech 16 bodů na kružnici soustředných kolem testovaného bodu. Pro každou relativní pozici na kružnici $x \in \{1 \dots 16\}$ může mít pixel 3 stavy vzhledem k středovému pixelu p:

$$\begin{array}{ll} \text{tmavší:} & I_{p\to x} \leq I_p - t \\ \text{podobný:} & I_p - t < I_{p\to x} < I_p + t \\ \text{světlejší:} & I_p + t \leq I_{p\to x} \end{array}$$

Ve 2. fázi je použit ID3[2] algoritmus, který vytvoří rozhodovací strom schopný klasifikovat všechny rohy nalezené v sadě trénovacích obrázků, díky čemuž je i schopen správně popsat pravidla *FAST detektoru*.

Rozhodovací strom je posléze překonvertován do C kódu, který je tvořen velkou řadou `if-else` podmínek a také `goto` nepodmíněných skoků. Tento kód je poté zkompilován a používán jako detektor rohů. Většina bodů je odmítnuta již po testech dvou pixelů, což vede k značnému navýšení rychlosti.

Na stránkách[3] Edwarda Rostena lze nalézt zdrojové kódy, sloužící pro generaci vlastního FAST detektoru ze svého setu obrázků. Výstupem může být i detektor v jiném jazyce.

2.2 Potlačení blízkých bodů

Velmi užitečnou vlastností *FAST detektoru* je potlačení blízkých bodů, které je založeno na postupném zvyšování prahu. Při tomto ději totiž dochází ke snižování počtu detekovaných rohů. Rohy jsou ohodnoceny maximální hodnotou prahu t, pro kterou jsou ještě detekovány jako roh. Detekce blízkých bodů je prováděna v matici 3×3.

[2] **Iterative Dichotomiser 3** je algoritmus sloužící k generování rozhodovacích stromů stvořený Rossem Quinlanem.

[3] http://mi.eng.cam.ac.uk/~er258/work/fast.html

2.3 Deklarace funkce a její parametry

Zdrojový kód *FAST detektoru* lze nalézt v souboru `cvfast.cpp` v adresáři `cvaux`.

```
void FAST( const Mat& image, vector<KeyPoint>& keypoints, int
   threshold, bool nonmax_suppression );
```

image Zdrojový obrázek, ve kterém se budou detekovat rohové body. Detektor pracuje pouze s **gray-scale obrázky**.

keypoints Výstupní vektor bodů, do kterého budou uloženy body detekované detektorem.

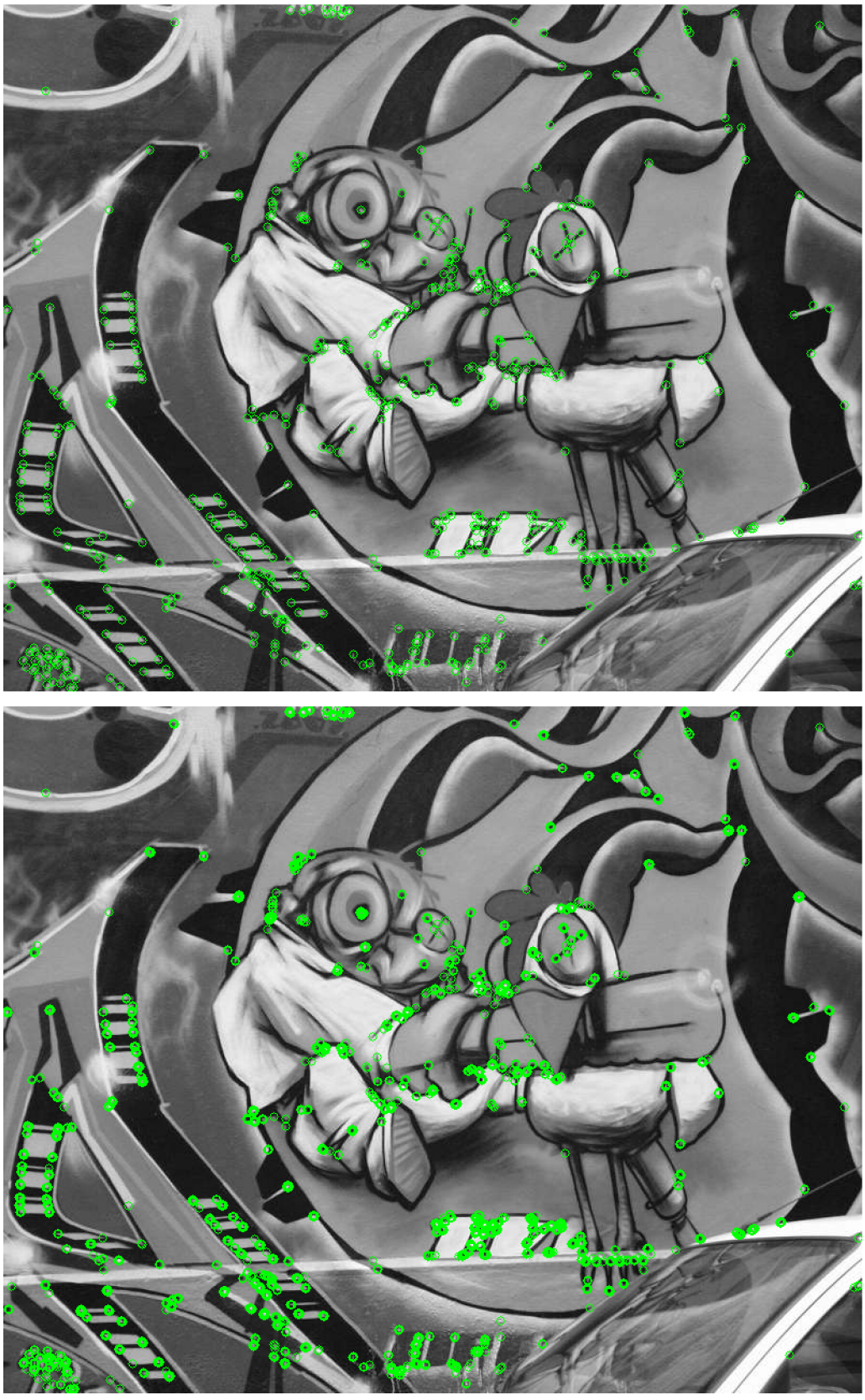
supression Určuje, zda bude po detekci použito potlačení blízkých bodů.

2.4 Příklad použití

```
// nacte obrazek ve stupnich sedi
cv::Mat img = cv::imread( "jmenoSouboru"
   ,CV_LOAD_IMAGE_GRAYSCALE );
// pokud se podarilo obrazek nacist, provede detekci
if( img.data )
{
    // vektor slouzici pro ukladani nalezenych klicovych bodu
    std::vector<cv::KeyPoint> keypoints;
    std::vector<cv::KeyPoint>::iterator it;
    keypoints.clear();
    // provede detekci rohovych bodu pomoci FAST detektoru v
       matici obrazku img s prahem 50 a zaplym potlacenim
       blizkych bodu
    cv::FAST( img, keypoints, 50, true );
    // vsechny nalezene body postupne zaznamena do obrazku
       jako kruznice o polomeru 3
    cv::Point point;
    for ( it = keypoints.begin();  it < keypoints.end(); it++
       ) {
        point.x = it->pt.x;
        point.y = it->pt.y;
        cv::circle( img, point, 3, CV_RGB(255,0,0) );
    }
   // zobrazi obrazek a pocka na stisk klavesy
   cv::namedWindow( "img", CV_WINDOW_AUTOSIZE );
   cv::imshow( "img",img );
   cv::waitKey(0);
}
```

2.5 Porovnání detekce se zapnutou a vypnutou funkcí potlačení blízkých bodů

FAST detektor jsem testoval na obrázku o rozměrech 800×640 bodů, přičemž detekce bez potlačení blízkých bodů trvala $4,0ms$ oproti $4,5ms$ se zapnutým potlačením blízkých bodů. Práh detektoru byl nastaven na hodnotu 60. Výstup obou detektorů je zobrazen na obrázku 2.2.



Obrázek 2.2: Detekce rohových bodů pomocí FAST algoritmu se zapnutým/vypnutým potlačením blízkých bodů.

Kapitola 3

Maximally stable extremal regions

Metoda Maximally stable extremal regions byla navržena pro zjišťování podobností elementů obrázků s rozdílnými úhly pohledu. Tato metoda extrakce mnoha vzájemně si odpovídajících elementů v obraze vedla k lepší identifikaci totožných objektů v různých pohledech shodné scény. Metodu jako první prezentoval Jiří Matas. Informace o této metodě byly čerpány z [10] a [15].

Základní princip MSER je následující. Představme si všechny možné prahy obrázku v úrovních šedi. Všechny body pod úrovní prahu označíme jako černé, ostatní nad úrovní prahu jako bílé. Pokud si po té zobrazíme posloupnost obrázků, kde budeme postupně měnit práh z nejmenší hodnoty na největší, uvidíme první obrázek jako bílý. S postupným zvedáním prahu se na obraze budou tvořit černé shluky bodů. Tyto shluky se budou se zvyšováním prahu zvětšovat a v určitém bodě se regiony se shodnými lokálními minimy sloučí. Poslední obrázek bude celý černý. Soubor všech propojených komponent ze snímků je množina všech maximálních regionů. Pokud bychom chtěli dostat minimální regiony, získali bychom je stejným postupem, pouze s převrácením intenzity bodů.

Metoda MSER má tyto vlastnosti:

- Invariantní vůči afinním transformacím obrázku. To znamená, že obrázek může být různě zdeformován či zkosen.
- Stabilní na mnoha prazích různě se lišících regionů.
- Je schopna stejně dobře detekovat jak malé, tak i naopak velmi velké regiony, čehož lze využít při detekci v obrázcích s rozdílnými měřítky.
- Soubor všech extrémních regionů lze vyčíslit v $\mathcal{O}(n\ log(log\ n))$, kde n je počet pixelů v obrázku.

Vyčíslení extrémních regionů vypadá následovně. Nejprve jsou pixely seřazeny podle intenzity. Složitost tohoto kroku je $\mathcal{O}(n)$. Toto řazení bývá často realizováno jako *BINSORT*[1]. Po seřazení jsou pixely umísťovány do obrazu (buď v narůstajícím, či snižujícím se pořadí) a seznam propojených částí a jejich ploch se udržuje pomocí *union-find algoritmu*[2]. Algoritmus je podle [10] velmi rychlý v praktickém použití.

[1] **BINSORT** je řadící algoritmus založený na insert sortu, který však pro vyhledání místa pro zatřídění právě řazeného prvku do již seřazeného pole používá binární půlení.

[2] **union-find algoritmus** je algoritmus, který dokáže určit, ve které množině se nachází daný prvek(find) a také umí spojit dvě množiny dohromady(union).

3.1 Navržený robustní algoritmus

Pro porovnávání detekovaných regionů byl navržen algoritmus, který pro každý detekovaný význačný region vytvoří několik regionů měřitelných, které normalizuje tak, aby je následně bylo možné porovnávat. Pro význačný region na prvním obrázku je nalezen takový význačný region z druhého obrázku, kde je největší shoda jejich měřitelných regionů.

Detekce význačných regionů

Prvním krokem je detekce *význačných regionů*, při které se MSERy počítají jak z normálního (MSER+), tak z invertovaného obrázku (MSER–).

Měřitelné regiony

Měřitelné regiony s libovolnou velikostí mohou být spojeny s každým *význačným regionem*, jestliže je jejich stavba affině-kovariantní[3]. Menší regiony jsou používány spíše ke splnění planární podmínky, nikoliv pro spojení nesouvislostí v hloubce či orientaci. Naopak zvětšování regionů s sebou nese riziko zahrnutí části pozadí, které může být na více porovnávaných obrázcích zcela odlišné. Optimální velikost *měřitelných regionů* závisí na scéně obrazu a je odlišná pro každý *význačný region*.

V tomto algoritmu jsou *měřitelné regiony* vybírány v několika měřítcích: jako velikost samotného význačného regionu, $1,5$, 2 a 3 násobek *význačného regionu*.

Invariantní popis

Po aplikaci transformace, která diagonalizuje kovariantní matici význačných regionů, jsou použity rotační invarianty. Tuto kombinaci lze označit afinně-invariantní procedurou.

Robustní porovnávání

Pro všechny měřitelné regiony M_A^i, vytvořené na význačném regionu A, jsou nalezeny význačné regiony $B_1, \ldots, B_k$ v jiném obrázku s odpovídajícími i-tými měřitelnými regiony $M_{B_1}^i, \ldots, M_{B_k}^i$ nejbližšími k M_A^i. Jejich ohodnocení potom udává shodu mezi A a každým z $B_1, \ldots, B_k$.

Ohodnocení jsou sečteny ve všech měřitelných regionech. Význačné regiony s největším počtem ohodnocení jsou potom kandidáty na možnou shodu. Pomocí pravděpodobnostní analýzy je vybrán odpovídající význačný region.

Invariantní popis je použit jako předběžný test korespondence regionů, přičemž konečný výběr je založen na korelaci. Nejprve jsou aplikovány transformace, jenž diagonalizují kovariantní matice význačných regionů. Výsledné kruhové regiony jsou korelovány, což je efektivně prováděno v polárních souřadnicích pro odlišné velikosti kružnic.

Přibližnou epipolární geometrii[4] lze vypočítat aplikací RANSAC[5] na těžiště význačných regionů. Přesnost epipolární geometrie lze zlepšit tak, že jsou nejprve spočítány afinní

[3] **Kovariance** udává míru vzájemné vazby mezi veličinami.

[4] **Epipolární geometrie** je geometrií dvou středových promítání. Je teoretickým základem pro určení vztahu mezi dvěma obrázky téže scény a pro rekonstrukci scény v prostoru.

[5] **RANdom SAmple Consensus** je iterativní metoda pro odhad parametrů matematického modelu z řady pozorovaných dat, které obsahují odlehlé hodnoty. Algoritmus byl poprve zvěřejněn Fischlerem a Bollesem v roce 1981.

transformace mezi potenciálně si odpovídajícími význačnými regiony. Podobnost kovariantních matic udává afinní trasformace současně s rotací, přičemž ta je určena epipolárními linkami. Dále jsou vybrány pouze regiony s korelací jejich transformovaných obrázku nad vybraným prahem.

Přesnost epipolární geometrie je odhadována *eight-point algoritmem*[6].

3.2 Řešení pro gray-scale a barevný obrázek

V OpenCV 2.0 jsou implementovány dva algoritmy MSER, jeden pro gray-scale a druhý pro barevný obrázek.

- Gray-scale algoritmus je založen na *Linear Time Maximally Stable External Regions*.
- Algoritmus pro barevný obrázek je založen na *Maximally Stable Colour Regions*[4], který nepracuje s prahováním intenzity, ale s *analýzou shluků*[7], založených na barevných gradientech. Udává se, že tento algoritmus je 3 – 4x pomalejší než standardní gray-scale algoritmus.
- Druh algoritmu je automaticky zvolen podle matice obrázku, ve kterém provádíme detekci.

3.3 Deklarace funkce a její parametry

Zdrojový kód algoritmu lze nalézt v souboru `cvmser.cpp` v adresáři `cv`.

```
MSER();
```

Defaultní nastavení je uvedeno v závorkách za jednotlivými parametry u následujícího konstruktoru.

```
MSER( int _delta, int _min_area, int _max_area, float
   _max_variation, float _min_diversity, int _max_evolution,
   double _area_threshold, double _min_margin, int
   _edge_blur_size );
```

_delta (5) Používá se při výpočtu odchylky regionů. Určuje, po jakých hodnotách se bude algoritmus posouvat při hledání správného prvku.

_min_area (60) Minimální velikost regionu

_max_area (14400) Maximální velikost regionu

_max_variation (0.25) Maximální odchylka regionů, které lze spojit do jednoho.

min_diversity (0.2) Minimální odlišnost

[6] **Eight-point algoritmus** je algoritmus sloužící k odhadu základní matice ze setu odpovídajících si bodů ve dvou obrázcích při stereografii. Algoritmus byl představen v roce 1981 Christopherem Longuet-Higginsem.

[7] **Cluster analysis** spočívá v rozkladu daného souboru dat na několik téměř stejnorodých podmnožin

_max_evolution (200) Maximální rozvoj udává počet prahů, které budou použity při detekci.

_area_threshold (1.01) Práh oblasti

_min_margin (0.003) Margin udává počet prahů, pro které je region stabilní. K tomu, aby byl region stabilní, se musí tedy vyskytnout minimálně u `_min_margin` prahů.

_edge_blur_size (5) Hodnota *Gaussova šumu*, která se aplikuje na barevný obrázek ještě před samotnou detekcí.

Stabilní region je takový, jestliže je větší jako `_min_area` a zároveň menší jako `_max_area`. Přitom se musí lišit od svého předchůdce o více jak `min_diversity`.

Parametry `_max_evolution`, `_area_threshold`, `_min_margin` a `_edge_blur_size` se používají pouze u detekce v barevném obrázku.

```
void operator()(Mat& image, vector<vector<Point> >& msers,
   const Mat& mask) const;
```

image Zdrojový obrázek, ve kterém bude prováděna detekce.

msers Výstupní vektor regionů. V každém z vektorů jsou uloženy body, patřící jednomu regionu.

mask Operační maska. Specifikuje, v jakých částech matice obrázku se budou regiony vyhledávat.

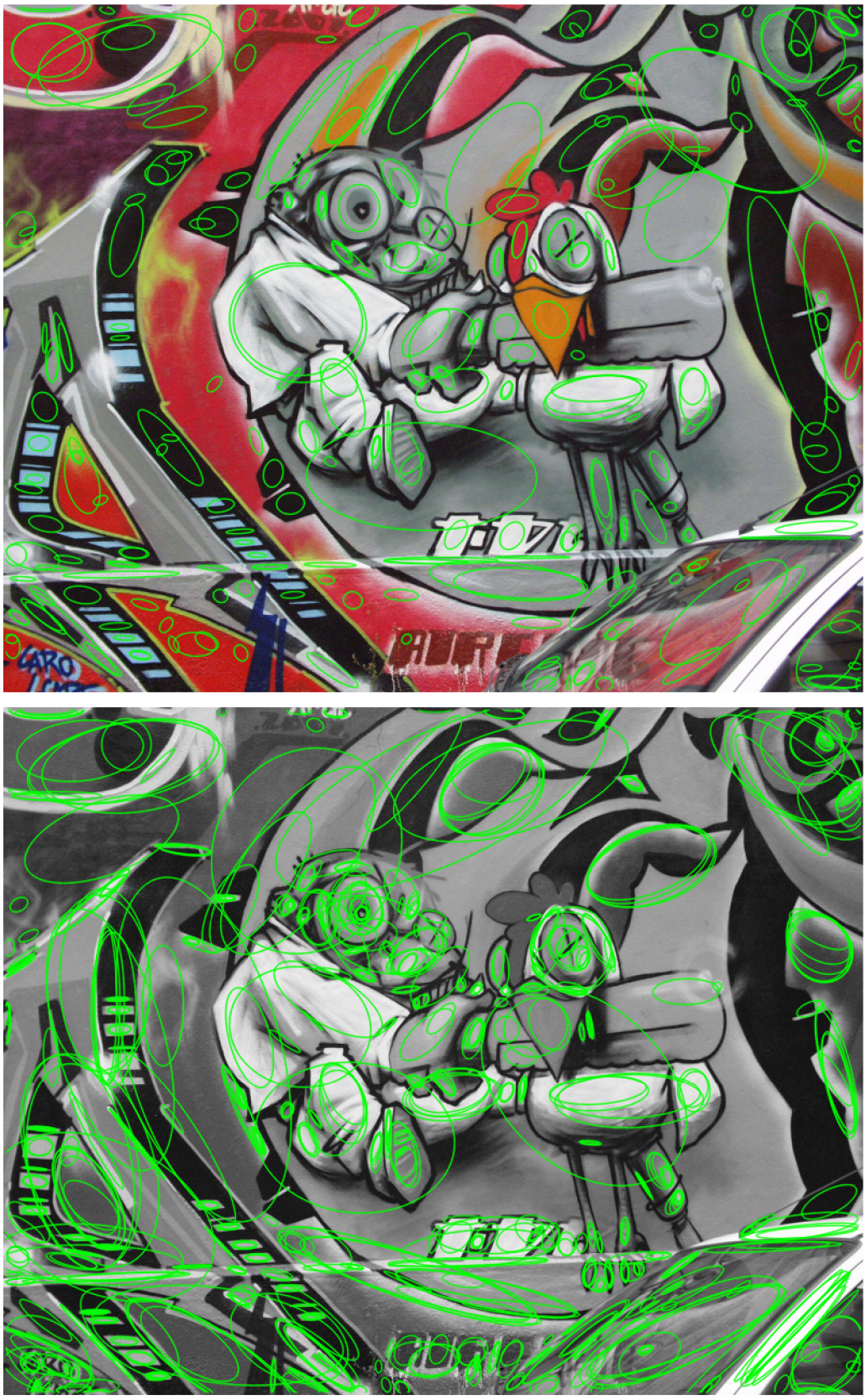
3.4 Příklad použití

```
// nacte obrazek
cv::Mat img = cv::imread("jmenoSouboru",CV_LOAD_IMAGE_COLOR);
if( img.data )
{
    // vektor slouzici k ukladani nalezenych klicovych bodu
    std::vector<std::vector<cv::Point>> keypoints;
    std::vector<std::vector<cv::Point>>::iterator it;
    std::vector<cv::Point>::iterator it2;
    // provede detekci
    cv::Mat tmp2;
    cv::MSER mser;
    mser(img,keypoints,tmp2);
    // kazdy region zobrazi odlisnou barvou
    for (it = keypoints.begin(); it < keypoints.end(); it++)
    {
        cv::Scalar color(rand()&255,rand()&255,rand()&255);
        for ( it2 = it->begin(); it2 < it->end(); it2++ )
        {
            cv::circle(img,cv::Point(it2->x,it2->y),0,color);
```

```
        }
    }
    // zobrazi obrazek a pocka na stisk klavesy
    cv::namedWindow( "img", CV_WINDOW_AUTOSIZE );
    cv::imshow("img",img);
    cv::waitKey(0);
}
```

3.5 Porovnání detekce regionů gray-scale a barevného obrázku

Jak jsem se již zmínil v sekci 3.2, MSER používá dvě odlišné metody pro detekci regionů v gray-scale a barevném obrázku, proto jsem se rozhodl porovnat výkonnost a výstup těchto dvou přístupů. Porovnání jsem prováděl na obrázku o rozměrech 800×640 bodů s defaultním nastavením parametrů zmíněným výše v 3.3, přičemž zpracování černobílého obrázku trvalo $136ms$ oproti $491ms$ u obrázku barevného. Z obrázků je také zřejmé, že metody používají opravdu odlišné přístupy, jelikož se stejným nastavením nejsou detekované regiony totožné.



Obrázek 3.1: Porovnání detekce regionů v barevném a grayscale obrázku algoritmem MSER se shodným nastavením.

Kapitola 4

LDetector

LDetector je rychlý detektor význačných bodů založený na kruhovém testu, který byl publikován Vincentem Lepetitem. Přestože bylo již publikováno mnoho kvalitních metod k detekci význačných bodů v obraze, byl *LDetector* publikován z několika důvodů:

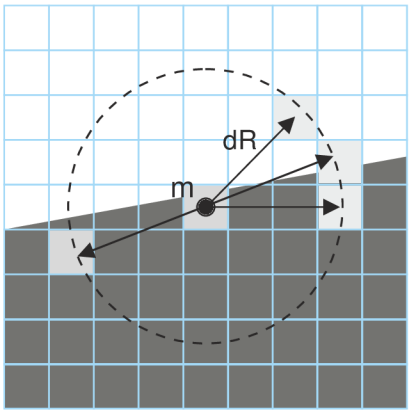
- Je rychlý
- Jeví se jako velmi stabilní v praxi
- Může určit orientaci význačného bodu

Informace vztahující se k *LDetectoru* byly čerpány z [9].

Základní myšlenkou této metody je porovnávání intenzity bodů, které leží na kružnici, soustředěné kolem testovaného klíčového bodu. Pokud dva zcela odlišné body na tomto kruhu mají přibližně stejnou intenzitu, potom je bod prohlášen za neklíčový. Algoritmus provádí test typu:

$$\begin{aligned} \text{pokud } |I(m) - I(m + dR_\alpha)| &\leq +\tau \\ \text{a zároveň } |I(m) - I(m - dR_\alpha)| &\leq +\tau \end{aligned}$$

potom bod m není význačný bod, kde $dR_\alpha = (R\,cos\,\alpha; R\,sin\,\alpha)$. R je rádius a α může nabývat hodnot $[0; \pi]$.



Obrázek 4.1: Zobrazení nutného testu sousedů[9]

V praxi se neporovnávají pouze přímo protilehlé pixely, ale je nutno porovnávat i jejich sousedy, jak je znázorněno na obrázku 4.1. Místo toho, aby byly prohledávány všechny body

na kružnici, je zde využito *rozhodovacích stromů*, které pracují *gaussovskými pyramidami* daného obrázku. Obvykle je test bodů, které nejsou klíčovými, ukončen velmi rychle bez prohledávání celého kruhu.

Tento přístup produkuje několik pozitivních reakcí v okolí význačného bodu, proto je zde, stejně jako u jiných detektorů, pro každou odezvu počítáno její ohodnocení. K dosažení stabilních výsledků je k ohodnocení význačných bodů použit *Laplacián*[1].

Pro určení orientace význačných bodů je použita stejná konstrukce. Orientace α_m je brána jako:

$$\alpha_m = \underset{\alpha \in [0;2\pi]}{\operatorname{argmax}} |I(m) - I(m + dR_\alpha)|$$

Orientace je dostatečně stabilní k normalizaci sousedních význačných bodů s ohledem na 2D orientaci. Tato metoda pracuje pouze s **gray-scale obrázkem**, jelikož se při detekci význačných bodů porovnávají intenzity bodů.

Gaussovské pyramidy

K tomu, aby byl algoritmus použitelný, je nutné provádět detekci bodů v pyramidách obrázku, jak jsem již uváděl výše. Třída `LDetector` obsahuje přetížený operátor, kterému lze předat jak samotný obrázek, tak předem vygenerované pyramidy. LDetector využívá pro generaci jednotlivých pyramid funkci `pyrDown()`, která je založena na postupném zmenšování obrázku na polovinu, což provádí tak, že jej nejprve rozmaže pomocí gausiánu a následně vypustí každý druhý řádek a sloupec.

metoda operator

Nejzákladnější metodou je `operator`, která je schopna velice rychle nalézt význačné body v obrázku. Algoritmus pro všechny pyramidy pracuje následnovně:

1. Sestaví další pyramidovou vrstvu
2. Detekuje význačné body, spočítá jejich ohodnocení
3. Provede potlačení blízkých bodů
4. Přizpůsobí počet význačných bodů

metoda getMostStable2D

Metodu `getMostStable2D` je vhodné použít pro detekci význačných bodů před samotným během reálně běžící aplikace, jelikož je tento výpočet značně náročný. Princip metody spočívá ve snaze najít nejvíce stabilní body z množiny obrázků, které jsou vytvořeny affinními transformacemi ze vstupního obrázku. Postup algoritmu pro jeden pohled je následující:

1. Vygeneruje náhodnou afinní trasformaci
2. Tuto transformaci aplikuje na zdrojový obrázek
3. Spustí detektor význačných bodů v pyramidách (metoda `operator`)
4. Vytvoří inverzní afinní trasformaci

[1] **Laplacian of Gaussian** je metoda pro detekci rohů, při kterém se provádí konvoluce obrázku s Gaussovým šumem

5. Namapuje detekované body zpátky do původního obrázku pomocí inverzní transformace

Na konci algoritmu jsou všechny body seřazeny podle jejich ohodnocení, přičemž je vybrán pouze požadovaný počet bodů.

4.1 Deklarace funkce a její parametry

Zdrojový kód algoritmu lze nalézt v souboru `cvmser.cpp` v adresáři `cv`.

```
LDetector();
```

Defaultní nastavení je uvedeno v závorkách za jednotlivými parametry u následujícího konstruktoru.

```
LDetector(int _radius, int _threshold, int _nOctaves, int
   _nViews, double _baseFeatureSize, double
   _clusteringDistance);
```

_radius (7) Poloměr kružnice, která bude použita při testování jednotlivých bodů, jako je zobrazeno na obrázku 4.1.

_threshold (20) Práh

_nOctaves (3) Počet Gaussovských pyramid, které budou vytvořeny z obrázku, pro následnou detekci.

_nViews (1000) Počet pohledů, ze kterých bude metoda `getMostStable2D` získávat stabilní význačné body.

_baseFeatureSize (32) Základní velikost význačného bodu

_clusteringDistance (2) Vzdálenost jednotlivých seskupení při metodě `getMostStable2D`

```
void operator()(const Mat& image, vector<KeyPoint>&
   keypoints, int maxCount=0, bool scaleCoords=true) const;
```

Nejprve spočítá Gaussovské pyramidy pro matici `image` a poté volá operátor, který již pracuje s pyramidami.

image Zdrojový obrázek, ve kterém se budou detekovat význačné body.

keypoints Výstupní vektor význačných bodů

maxCount Detektor vrátí `maxCount` počet význačných bodů.

scaleCoords Úprava měřítka podle toho, ve které pyramidě byl klíčový bod detekován

---

```
void operator()(const vector<Mat>& pyr, vector<KeyPoint>&
    keypoints, int maxCount=0, bool scaleCoords=true) const;
```

pyr Gausovské Pyramidy, ve kterých se budou detekovat význačné body.

keypoints Výstupní vektor význačných bodů

maxCount Detektor vrátí `maxCount` počet význačných bodů.

scaleCoords Úprava měřítka podle toho, ve které pyramidě byl klíčový bod detekován.

```
void getMostStable2D(const Mat& image, vector<KeyPoint>&
    keypoints, int maxCount, const PatchGenerator&
    patchGenerator) const;
```

Získá nejvíce stabilní body z matice obrázku `image`. Tento postup provádí pro `_nViews` afinních transformací obrázku.

image Zdrojový obrázek, ve kterém se budou detekovat význačné body.

keypoints Výstupní vektor význačných bodů

maxCount Maximální počet hledaných nejvíce stabilních význačných bodů

patchGenerator Určuje, jaké transformace budou generovány

```
void setVerbose(bool verbose);
```

verbose Nastaví detektor jako výřečný

```
void read(const FileNode& node);
```

node Načte nastavení detektoru z uzlu `node`

```
void write(FileStorage& fs, const String& name=String())
    const;
```

fs Datové úložiště

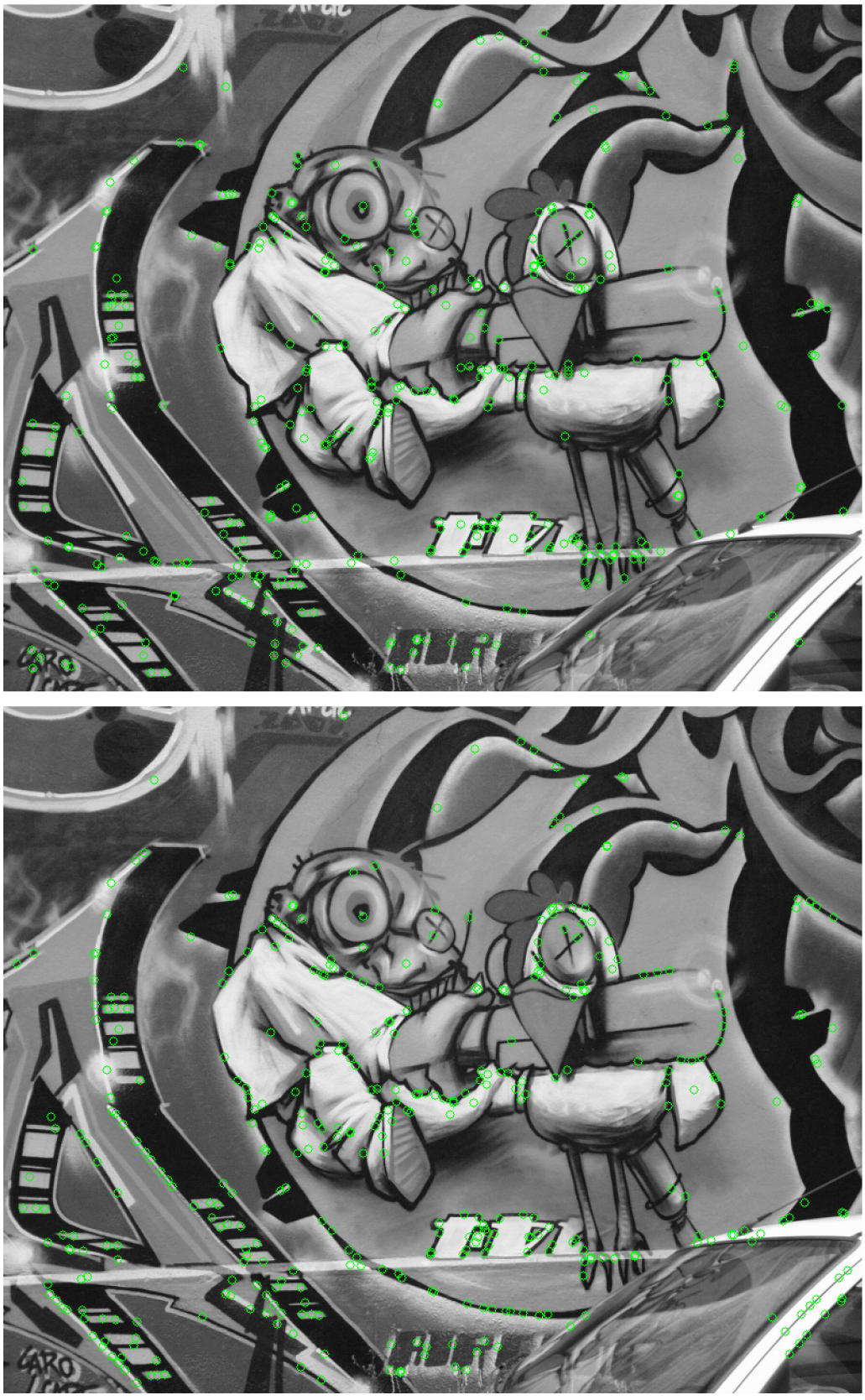
name Jméno, pod kterým bude uloženo nastavení detektoru

4.2 Příklad použití

```
// nacte obrazek ve stupnich sedi
cv::Mat img =
   cv::imread("jmenoSouboru",CV_LOAD_IMAGE_GRAYSCALE);
if( img.data )
{
    cv::Mat img1 =  img.clone();
    cv::Mat img2 =  img.clone();
    // vektor slouzici k ukladani nalezenych klicovych bodu
    std::vector<cv::KeyPoint> keypoints;
    std::vector<cv::KeyPoint>::iterator it;
    cv::LDetector ldetector;
    // ziska nejvice stabilni body z defaultne nastavenych
       1000 transformaci
    ldetector.getMostStable2D(img,keypoints,200,*(new
       cv::PatchGenerator()));
    for (it = keypoints.begin();  it < keypoints.end(); it++)
    {
        cv::circle(img1,cv::Point(it->pt.x,it->pt.y), 3,
           cv::Scalar(255,0,0));
    }
    // provede detekci 200 nejsilnejsich bodu pyramidovym
       detektorem
    ldetector(img,keypoints,200);
    for (it = keypoints.begin();  it < keypoints.end(); it++)
    {
        cv::circle(img2,cv::Point(it->pt.x,it->pt.y), 3,
           cv::Scalar(255,0,0));
    }
    cv::namedWindow("most stable poses",CV_WINDOW_AUTOSIZE);
    cv::imshow("most stable poses",img1);
    cv::namedWindow("operator",CV_WINDOW_AUTOSIZE);
    cv::imshow("operator",img2);
    cv::waitKey(0);
}
```

4.3 Srovnání metod getMostStable2D a operator

Na obrázku 4.2 jsou zobrazeny výstupy dvou metod LDetectoru. Metodu `getMostStable2D` je vhodné použít pro získání nejvíce stabilních bodů před samotným během reálně běžící aplikace, jelikož detekce 400 nejvíce stabilních bodů pro 100 pohledů trvala $23666ms$ na obrázku o rozměrech 800×640 bodů. Oproti tomu rychlá metoda `operator`, založená na pyramidové detekci, zpracovala stejný obrázek za $127ms$. Detekce probíhala s defaultním nastavením uvedeným v 4.1, přičemž změněna byla pouze hodnota nViews = 100.



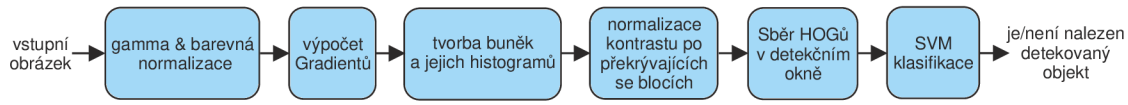
Obrázek 4.2: Porovnání detekce význačných bodů metodami `getMostStable2D` a `operator` algoritmu LDetector

Kapitola 5

HOG people and object detector

Histogram of Oriented Gradients je deskriptor význačných rysů, často používaný při zpracovávání obrazu sloužící k detekci objektů, který byl prezentovaný Navneetem Dalalem a Billem Triggsem v roce 2005. Nejčastěji je používán k detekci osob, zvířat či automobilů jak ve videu, tak ve statických obrázcích. Informace, vztahující se k teorii HOG deskriptoru, byly čerpány z [3, 16] a [14].

Základní myšlenkou HOG deskriptoru je to, že objekt vyskytující se v obrázku lze popsat distribucí intenzity gradientů nebo směry hran. Implementačně je toho dosaženo rozdělením obrázku na malé propojené regiony, zvané také buňky(cells). Každá buňka shrnuje histogram směru gradientů, či hranové reprezentace pixelů, jenž jsou uvnitř buňky. Kombinace těchto histogramů tvoří deskriptor. Pro zlepšení výsledku se histogramy kontrastně normalizují tak, že je spočítána intenzita ve větším regionu, než je buňka, nazývaném blok. Tato intenzita je poté použita k normalizaci všech buněk uvnitř bloku. Normalizace vede k lepším výsledkům při změně intenzity, či stínů.



Obrázek 5.1: Diagram práce HOG deskriptoru[3]

Gamma/barevná normalizace

Tyto normalizace mají pouze malý vliv na výkon deskriptoru, jelikož pozdější normalizace deskriptoru vykazují podobné výsledky.

Výpočet gradientu

HOG deskriptor používá jednu z nejběžnějších metod, kterou je 1-D centrovaná bodová diskrétní maska $[-1, 0, 1]$. Masku lze použít v horizontálním, vertikálním směru, nebo také v obou. V [3] jsou zmíněny i jiné více komplexní masky, které však nedosahovaly takového výkonu.

Tvorba buněk a jejich histogramů

Dalším krokem je vytvoření histogramů pro jednotlivé buňky. Každý pixel v buňce ovlivňuje výsledný hranově orientovaný histogram, který je založen na orientaci gradientu středového

elementu. Buňky mohou být obdélníkového či radiálního tvaru. Orientace gradientu je v mezích od 0° do 180° v případě neznaménkového gradientu, či od 0° do 360° stupňů v případě znaménkového gradientu. Dalal a Triggs zjistili, že pro detekci osob v obraze je nejlepší použít bezznaménkový gradient současně s 9 kanálovým histogramem. V některých případech, jak je např. detekce aut, či motorek, může zahrnutí znaménkové informace přinést jisté zlepšení.

Normalizace kontrastu po překrývajících se blocích

Kvůli změnám jasu a kontrastu musí být gradienty jednotlivých buněk lokálně normalizovány, což nás vede k seskupování jednotlivých buněk do bloků. HOG deskriptor je potom vektor normalizovaných buněk histogramu ze všech blokových oblastí. Jelikož se bloky překrývají, některé buňky do výsledného deskriptoru mohou přispívat i vícekrát. Jako nejoptimálnější se jeví nastavení, kdy je použito 6×6 pixelů pro jednu buňku, 3×3 buňky pro jeden blok a histogram o devíti kanálech.

Pro normalizaci bloků lze použít několik různých normalizačních schémat:

(a) L2-*norm*, $v \rightarrow \frac{v}{\sqrt{||v||_2^2+\epsilon^2}}$

(b) L1-*norm*, $v \rightarrow \frac{v}{(||v||_1+\epsilon)}$

(c) L1-*sqrt*, $v \rightarrow \sqrt{\frac{v}{||v||_1+\epsilon}}$,

kde v je neznormalizovaný vektor obsahující všechny histogramy v daném bloku, $||v||_k$ jeho k-norm pro $k = 1, 2$ a ϵ malá konstanta. L2-*hys* lze spočítat jako L2-*norm*, kde je výsledek ořízntut (limitován maximální hodnotou v na $0, 2$) a renormalizován.

Podle [3] vykazují L2-*hys*, L2-*norm* a L1-*sqrt* podobné výsledky. V knihovně OpenCV je zatím implementováno pouze normalizační schéma L2-*hys*.

Sběr HOGů v detekčním okně

Jednou z posledních částí dektekčního řetězce je sběr HOGů, jelikož je finální deskriptor tvořen všemi buňkami uvnitř všech bloků v detekčním okně. K detekci osob je v [3] použito detekční okno o rozměru 64×128 pixelů, kolem kterého je vytvořen okraj 16ti pixelů na všech čtyřech stranách.

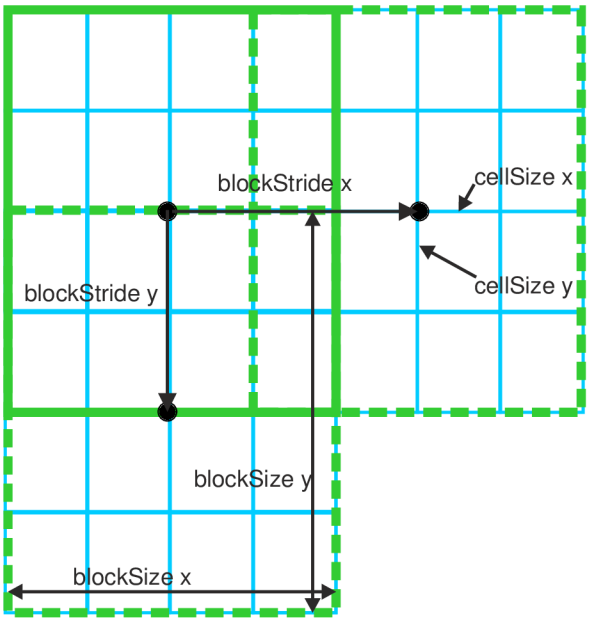
SVM klasifikace

Konečnou fází při rozpoznávání objektů pomocí HOG deskriptorů je předání deskriptorů nějakému rozpoznávacímu systému, založenému na *supervised learning*[1]. SVM[2] klasifikátor je binární klasifikátor, který hledá optimální nadrovinu jako rozhodovací funkci. Pokud je jednou natrénován na obrázcích s detailní reprezentací objektu, dokáže určit, zda je objekt přítomný, či nikoli. V HOG deskriptoru se používá lineární SVM klasifikátor, trénovaný pomocí SVM^{light}[3], který je lehce modifikován pro práci s velkými vektory.

[1] učení s učitelem

[2] **Support Vector Machine** je soubor metod strojového učení, který se používá pro klasifikaci.

[3] http://svmlight.joachims.org/



Obrázek 5.2: Zobrazuje interpretaci obrázku HOG deskriptorem. Modré obdélníky znázorňují jednotlivé buňky(cells), pro které jsou vytvářeny histogramy, zatímco zelené obdélníky vyobrazují normalizační bloky. Parametr `blockStride` definuje posun normalizačního bloku po obrázku.

5.1 Deklarace funkce a její parametry

Zdrojový kód algoritmu lze nalézt v souboru `cvhog.cpp` v adresáři `cvaux`.

```
HOGDescriptor() : winSize(64,128), blockSize(16,16),
   blockStride(8,8),cellSize(8,8), nbins(9),
   derivAperture(1), winSigma(-1),histogramNormType(L2Hys),
   L2HysThreshold(0.2), gammaCorrection(true);
```

```
HOGDescriptor(Size _winSize, Size _blockSize, Size
   _blockStride, Size _cellSize, int _nbins, int
   _derivAperture=1, double _winSigma=-1, int
   _histogramNormType=L2Hys, double _L2HysThreshold=0.2, bool
   _gammaCorrection=false);
```

_winSize Velikost detekčního okna

_blockSize Velikost bloku

_blockStride Definuje posun bloku po obrázku

_cellSize Velikost buňky

_nbins Počet kanálů v histogramu

_derivAperture Nemá žádný vliv na práci deskriptoru

_winSigma Počet detekčních oken

_histogramNormType Typ histogramu. Zatím lze použít pouze `L2Hys`.

_L2HysThreshold Práh histogramu. Používá se při normalizaci blokových histogramů.

_gammaCorrection Gamma korekce

```
HOGDescriptor(const String& filename);
```

Načte nastavení deskriptoru ze XML/YAML souboru `filename`. Nastavení se načítá z prvního uzlu na nejvyšší úrovni.

filename Jméno souboru

```
size_t getDescriptorSize() const;
```

Vrátí velikost deskriptoru.

```
bool checkDetectorSize() const;
```

Zkontroluje velikost detektoru. Velikost musí být nulová nebo rovna velikosti deskriptoru, popřípadě o jedna větší.

```
double getWinSigma() const;
```

Vrátí hodnotu `winSigma`. Pokud nebyla zadána, tak je spočítána jako:

$$(\text{šířka bloku} + \text{výška bloku})/8$$

```
virtual void setSVMDetector(const vector<float>&
    _svmdetector);
```

Nastaví SVM detektor na `_svmdetector`.

_svmdetector Vektor hodnot tvořících SVM detektor.

```
virtual bool load(const String& filename, const String&
    objname=String());
```

Načte nastavení deskriptoru z XML/YAML souboru `filename` a a uzlu `objname`.

filename Jméno souboru

objname Jméno uzlu

```
virtual void save(const String& filename, const String&
    objname=String()) const;
```

Uloží nastavení deskriptoru do XML/YAML souboru `filename` a uzlu `objname`. V případě, že není `objname` zadáno, ukládá nastavení do uzlu `filename`.

filename Jméno souboru

objname Jméno uzlu

```
virtual void compute(const Mat& img, vector<float>&
    descriptors, Size winStride=Size(), Size padding=Size(),
    const vector<Point>& locations=vector<Point>()) const;
```

Spočítá deskriptory z obrázku

img Matice zdrojového obrázku

descriptors Výstupní deskriptory

winStride Definuje posun okna po obrázku

padding Rozšíří detekční obrázek o okraje definované `padding`

locations Body, jejichž deskriptory bude metoda počítat. V případě, že žádné body nejsou zadané, spočítá deskriptory z celého obrázku.

```
virtual void detect(const Mat& img, vector<Point>&
    foundLocations, double hitThreshold=0, Size
    winStride=Size(), Size padding=Size(), const
    vector<Point>& searchLocations=vector<Point>()) const;
```

Provede detekci objektů podle předem nastaveného SVM v matici obrázku `img`.

img Matice zdrojového obrázku

foundLocations Výstupní vektor bodů, na kterých byl objekt detekován

hitThreshold Práh shody detekovaného objektu s SVM deskriptorem

winStride Definuje posun okna po obrázku

padding Rozšíří detekční obrázek o okraje definované `padding`

searchLocations Body, na kterých budeme detekovat objekty. V případě, že žádné nezadáme, provádí se detekce v celém obrázku.

```
virtual void detectMultiScale(const Mat& img, vector<Rect>&
    foundLocations, double hitThreshold=0, Size
    winStride=Size(), Size padding=Size(), double scale=1.05,
    int groupThreshold=2) const;
```

Provede detekci objektů podle předem nastaveného SVM klasifikátoru. Detekce je prováděna několikrát v postupně zmenšovaném obrázku `img`. Krok zmenšení obrázku je definován hodnotou `scale` a detekce se provádí výše zmiňovanou metodou `detect`. Maximální počet transformací je 64, přičemž nejmenší obrázek může mít nejméně velikost okna.

img Matice zdrojového obrázku

foundLocations Výstupní vektor obdelníků, ve kterých byl objekt detekován

hitThreshold Práh shody detekovaného objektu s SVM deskriptorem

winStride Definuje posun okna po obrázku

padding Rozšíří detekční obrázek o okraje definované `padding`

scale Udává, po jakých krocích bude zmenšován původní detekční obrázek. Např. hodnota 1.05 znamená, že každý následující obrázek bude zmenšen o 5%.

groupThreshold Práh seskupení podobných oblastí.

```
virtual void computeGradient(const Mat& img, Mat& grad, Mat&
    angleOfs, Size paddingTL=Size(), Size paddingBR=Size())
    const;
```

Spočítá gradienty z obrázku.

img Matice zdrojového obrázku

grad Výstupní matice velikosti gradientů

angleOfs Výstupní matice orientace gradientů

paddingTL Ohraničení obrázku nahoře a vlevo

paddingBR Ohraničení obrázku dole a vpravo

```
static vector<float> getDefaultPeopleDetector();
```

Vrátí vektor detektoru osob.

5.2 Příklad použití

HOG detektor lze použít pro detekci mnoha odlišných objektů. Níže uvedený příklad demonstruje detekci osob v obraze pomocí metody `detectMultiScale`, přičemž výstup detektoru je zobrazen na obrázku 5.3.

```
// nacte obrazek
cv::Mat img = cv::imread("jmenoSouboru",CV_LOAD_IMAGE_COLOR);
if( img.data )
{
    std::vector<cv::Rect> rects;
    cv::HOGDescriptor hog;
    // nastavi SVM detektor na detektor osob
    hog.setSVMDetector(
        cv::HOGDescriptor::getDefaultPeopleDetector() );
    // provede detekci
    hog.detectMultiScale(img, rects, 0, cv::Size(8,8),
        cv::Size(24,16), 1.1, 2);
    // nalezene obdelniky zaznemena do obrazku
    for( std::vector<cv::Rect>::iterator it=rects.begin();
        it<rects.end() ; it++)
    {
        cv::rectangle(img,it->tl(), it->br(),
            cv::Scalar(0,255,0),1);
    }

    // zobrazi obrazek a pocka na stisk klavesy
    cv::namedWindow( "img", CV_WINDOW_AUTOSIZE );
    cv::imshow("img",img);
    cv::waitKey(0);
}
```

Obrázek 5.3: Detekce osob v obraze za pomocí metody `detectMultiScale` třídy HOGDescriptor.

Kapitola 6

One-way descriptor

V minulosti bylo prezentováno již mnoho detektorů affiních regionů. V kombinaci s deskriptory, jako je např. SIFT[1], jsou použitelné pro většinu typů aplikací. Nedávno byl také v [6] publikován detektor založený na strojovém učení, který je mnohem rychlejší. V případě, že však chceme za běhu integrovat nové význačné body, které se staly viditelnými, je tento přístup nepoužitelný. Nevýhodou tohoto typu dektektorů je jeho pomalá fáze trénování. Hinterstoisser,Kutter,Navab,Fua a Lepetit přišli s přístupem, který je taktéž založen na strojovém učení. Vzhledem k předchozímu návrhu je ale schopen trénovat nové body za běhu, což je velmi užitečné pro SLAM[2] aplikace. Informace k teorii one-way deskriptoru jsou čerpány z [7].

Koncepce deskriptorů, založených na strojovém učení, je zpravidla tvořena dvěma kroky. První porovnává vstupní význačný bod s databází již detekovaných bodů. Jde v podstatě o získání přibližné pozice bodu. Druhý krok tuto pozici upřesňuje.

Získání přibližné pozice bodu

Základní myšlenkou k odhadnutí identity a pozice bodu je vytvoření množiny průměrných ploch(mean patches) z různých referenčních pohledů význačného bodu. Každá plocha je získána pokřivením původní plochy, jejíž středem je význačný bod, pro který jsou plochy generovány. Tato množina průměrných ploch pro jeden význačný bod je označována jako *One-way deskriptor*. Porovnáním příchozího bodu s množinou průměrných ploch dostaneme přibližný odhad pozice tohoto bodu.

Výpočet průměrných ploch je obvykle velice časově náročný. V této metodě je použito techniky PCA[3] spolu s trénovací fází. V trénovací fázi jsou spočítány průměrné PCA hodnoty a také PCA eigenvektory, které jsou následně použity k velice rychlému vytvoření průměrných ploch nového význačného bodu za běhu aplikace. Jde v podstatě jen o promítnutí význačného bodu do eigenprostoru.

Jelikož je pro rychlost algoritmu důležité, aby pokřivovací funkce použitá při generaci nových ploch byla lineární, je brána původní plocha větší než výsledná pokřivená plocha. Tím je zaručeno, že se v pokřivené ploše nevyskytnou žádné nové části.

[1] **Scale-Invariant Feature Transform** je algoritmus, který je schopen detekovat a popsat význačné body v obrázku.

[2] **Simultaneous Localization and Mapping** je technika používaná v robotismu, která slouží k sestavení mapy bez prioritních znalostí či aktualizování mapy s prioritními znalostmi, zatímco ve stejném čase je sledována aktuální poloha.

[3] **Principal Component Analysis** je matematická procedura schopna transformovat velký počet korelovatelných hodnot do menšího dále již nekorelovatelného počtu. Používá se ke snížení dimenze dat.

Upřesnění pozice bodu

Jakmile je pro příchozí bod získána přibližná pozice s její celkovou identitou $\widehat{id}$, je použit IC[4] *algoritmus* společně s aproximací nadroviny[5] k jejímu upřesnění.

6.1 Deklarace funkce a její parametry

Zdrojový kód *One-way deskriptoru* lze nalézt v souboru `cvoneway.cpp` v adresáři `cvaux`, přičemž implementace obsahuje celkem 3 třídy `OneWayDescriptor`, `OneWayDescriptorBase` a `OneWayDescriptorObject`. Pro přehlednost je zde uvedena pouze třída `OneWayDescriptorBase`, která je použita u demonstračního příkladu 6.2. Dokumentaci ostatních dvou tříd lze nalézt v příloze A.

Třída `OneWayDescriptorBase` zahrnuje funkcionalitu pro trénování a načítání několika one-way deskriptorů. Dokáže také vyhledat nejbližší deskriptor k vstupnímu význačnému rysu.

```
OneWayDescriptorBase(Size patch_size, int pose_count, const
    char* train_path = 0, const char* pca_config = 0, const
    char* pca_hr_config = 0, const char* pca_desc_config = 0,
    int pyr_levels = 2, int pca_dim_high = 100, int
    pca_dim_low = 100);
```

patch_size Velikost vstupních ploch

pose_count Počet póz, které se budou generovat pro každý detektor

train_path Cesta k trénovacím souborům

pca_config Jméno souboru, který obsahuje PCA pro malé plochy.

pca_hr_config Jméno souboru, který obsahuje PCA pro velké plochy.(velikosti vstupních ploch)

pca_desc_config Jméno souboru, který obsahuje deskriptory PCA složek

pyr_levels Úroveň pyramid generovaných z obrázku

pca_dim_high Počet PCA složek určených pro generaci afinních trasformací

pca_dim_low Počet PCA složek sloužících k porovnávání

```
void Allocate(int train_feature_count);
```

Naalokuje paměť pro `train_feature_count` deskriptorů.

```
void AllocatePCADescriptors();
```

[4] Inverse Compositional
[5] Hyperplane approximation

Naalokuje paměť pro PCA deskriptory.

```
Size GetPatchSize();
```

Vrátí velikost každé obrázkové plochy.

```
int GetPoseCount();
```

Vrátí počet póz pro každý deskriptor.

```
int GetPyrLevels();
```

Vrátí počet pyramidových úrovní.

```
void CreateDescriptorsFromImage(IplImage* src, const
   vector<KeyPoint>& features);
```

Vytvoří deskriptory pro každý vstupní význačný bod.

src Vstupní obrázek

features Vektor význačných bodů

```
void CreatePCADescriptors();
```

Vytvoří PCA deskriptory potřebné pro generaci deskriptorů význačných bodů.

```
const OneWayDescriptor* GetDescriptor(int desc_idx) const;
```

Vrátí deskriptor význačného rysu na indexu `desc_idx`.

```
void FindDescriptor(IplImage* patch, int& desc_idx, int&
   pose_idx, float& distance) const;
```

Najde nejbližší deskriptor v obrázku `patch`.

patch Vstupní plocha obrázku

desc_idx Výstupní index nejbližšího deskriptoru

pose_idx Výstupní index pózy nejbližšího deskriptoru

distance Vzdálenost nalezené pózy od vstupní plochy

```
void FindDescriptor(IplImage* src, Point2f pt, int& desc_idx,
   int& pose_idx, float& distance) const;
```

Najde nejbližší deskriptor v obrázku `patch`.

src Vstupní obrázek

pt Středový bod význačného bodu

desc_idx Výstupní index nejbližšího deskriptoru

pose_idx Výstupní index pózy nejbližšího deskriptoru

distance Vzdálenost nalezené pózy od vstupní plochy

```
void InitializePoses();
```

Vygeneruje náhodné pózy

```
void InitializeTransformsFromPoses();
```

Vygeneruje afinní matice pro pózy

```
void InitializePoseTransforms();
```

Zavolá metody `InitializePoses` a `InitializeTransformsFromPoses`.

```
void InitializeDescriptor(int desc_idx, IplImage*
    train_image, const char* feature_label);
```

Inicializuje deskriptor.

desc_idx Index deskriptoru

train_image Plocha obrázku

feature_label Jméno deskriptoru

```
void InitializeDescriptors(IplImage* train_image, const
    vector<KeyPoint>& features, const char* feature_label =
    "", int desc_start_idx = 0);
```

Inicializuje deskriptory pro každý z předaných význačných bodů.

train_image Trénovací obrázek

features Plocha obrázku

feature_label Jméno deskriptorů

desc_start_idx Začátek indexace deskriptorů

```
int LoadPCADescriptors(const char* filename);
```

Načte PCA deskriptory ze souboru `filename`.

```
void SavePCADescriptors(const char* filename);
```

Uloží deskriptory do souboru `filename`.

```
void SetPCAHigh(CvMat* avg, CvMat* eigenvectors);
```

Nastaví největší rozlišení pro PCA matice.

```
void SetPCALow(CvMat* avg, CvMat* eigenvectors);
```

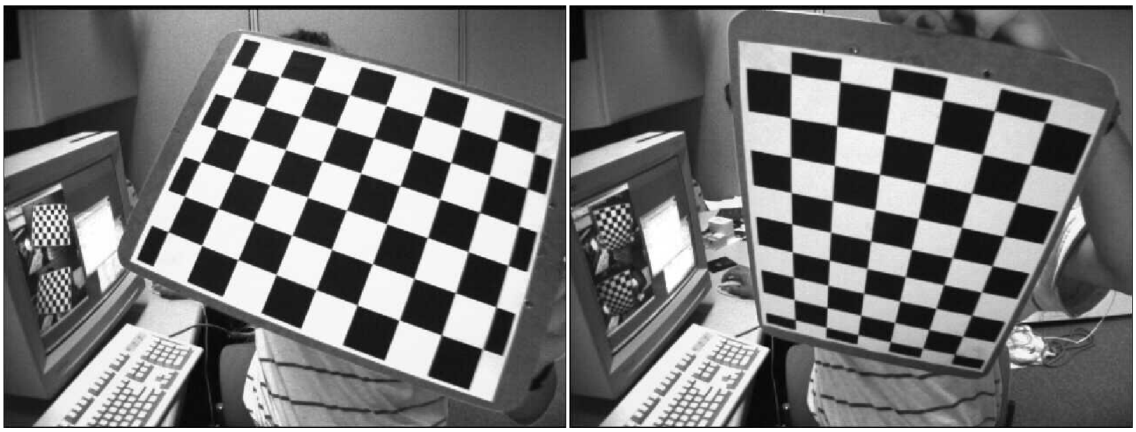
Nastaví nejmenší rozlišení pro PCA matice.

6.2 Příklad použití

Protože by byl příklad použití značně složitý, rozhodl jsem se postup, jakým lze *One-way deskriptor* použít, popsat pouze slovně v jednotlivých bodech. Při popisu v závorkách uvádím korespondující řádky kódu vzorového příkladu Victora Eruhimova, který lze nalézt v příloze B.

Trénovací fáze

Jak jsem již uvedl v popisu algoritmu, samotnému běhu předchází fáze trénování, ve které je potřeba vygenerovat PCA deskriptory ze sady trénovacích obrázků 6.1 (228–253):



Obrázek 6.1: Ukázka trénovacích obrázků převzatá z příkladů knihovny OpenCV 2.1.

1. Vygeneruje PCA význačné rysy jak pro největší rozlišení PCA matice (232), tak pro nejmenší rozlišení (236), kde je brána plocha o poloviční velikosti.
    - Načte PCA význačné rysy z trénovacích obrázků (176–219)
        - Pro všechny trénovací obrázky detekuje význačné body, např. pomocí algoritmu SURF (188–190)
        - Pro každý detekovaný význačný bod vytvoří plochu obsahující ve středu tento význačný bod (193–213)

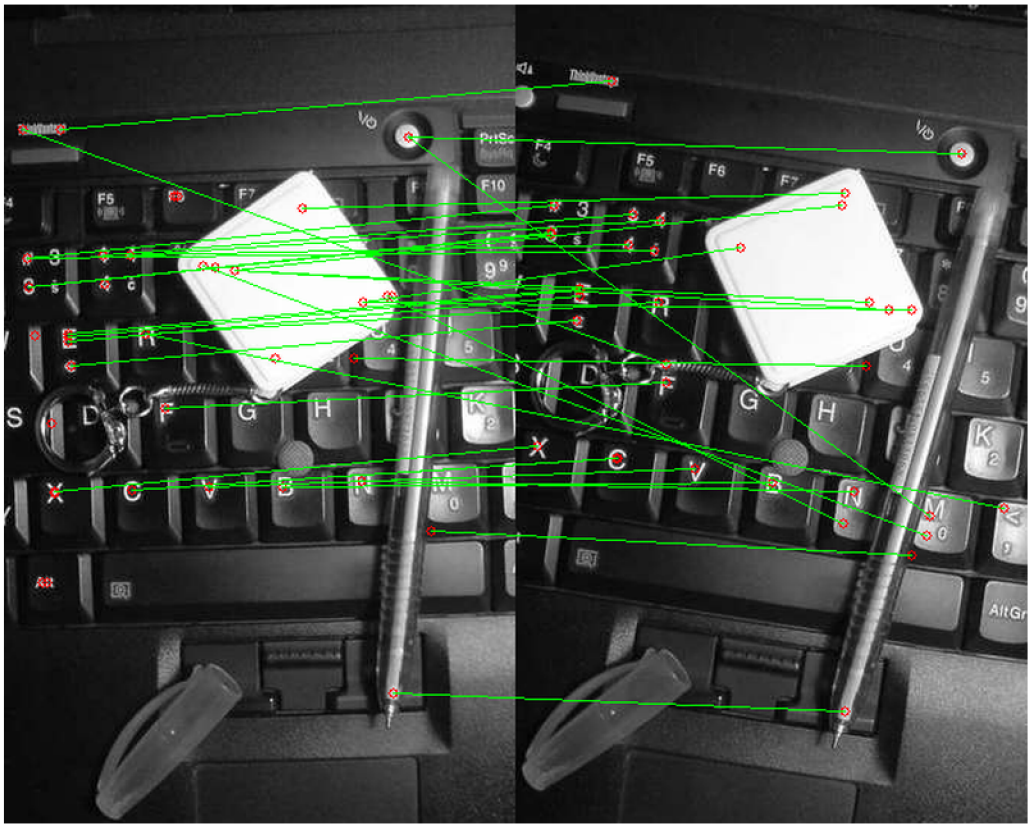
- Spočítá PCA význačné rysy (140–173)
  - Vytvoří matici pro data (147), do které postupně zapíše hodnoty všech ploch (152–162)
  - Z této matice spočítá vektor průměrných PCA hodnot a také PCA eigenvektory pomocí funkce (165)
  - Uloží PCA význačné rysy do souboru (169)

2. Zkonstruuje objekt `OneWayDescriptorBase` (240)
3. Nastaví největší/nejmenší rozlišení pro PCA tak, aby používalo již vygenerované soubory (241–242)
4. Vytvoří PCA deskriptory, které uloží do souboru (245–247)

Detekční fáze

Běh aplikace poté vypadá následovně (56–87):

1. V prvním obrázku jsou detekovány význačné body, např. algoritmem SURF (58–59)
2. Je vytvořen objekt `OneWayDescriptorBase`, přičemž jako parametry jsou mu dány PCA soubory vytvořené v trénovací fázi. (65)
3. Vytvoří deskriptory z obrázku pomocí metody `CreateDescriptorsFromImage` (66)
4. Detekuje význačné body v druhém obrázku stejným algoritmem jako u prvního obrázku. (70–71)
5. Pro všechny detekované body u druhého obrázku najde odpovídající význačné body z prvního obrázku pomocí metody `FindDescriptor`. (79–84)
6. Nakonec vykreslí podobnosti nalezené podobnosti. (87)



Obrázek 6.2: Výstupní obrázek one-way deskriptoru znázorňující korespondující význačné body.

Kapitola 7

Návrh a implementace

Tato kapitola pojednává o návrhu a implementaci demonstračních příkladů k dokumentovaným funkcím. Je zde také uvedeno základní ovládání aplikací.

Při tvorbě aplikace byly použity programovací jazyky *C* a *C++*. Dále byla použita knihovna OpenCV[8], která poskytuje nejen mnou dokumentované metody, ale i funkce pro jednoduchý přístup k obrazovým datům.

Mým úkolem bylo vytvořit programy, které budou používat identifikované metody, přičemž jsem se rozhodl vytvořit pro každou metodu jeden program, který demonstruje její základní použití.

7.1 Návrh demo aplikace

Jelikož je moje práce zaměřena na detektory, popř. deskriptory význačných bodů, jevilo se mi jako logické vytvořit jednu společnou kostru aplikace pro všechny metody.

Protože se jedná o tutoriál, který by měl usnadnit první použití algoritmů uživatelům, byl hlavní důraz kladen na jednoduchost aplikace. Aplikace se skládá ze dvou oken, přičemž po startu aplikace je zobrazeno pouze okno na obrázku 7.1, které zobrazuje základní pokyny pro práci s aplikací a trackbary. Pomocí trackbarů lze nastavit všechny parametry volané metody. Teprve po stisknutí tlačítka `Enter` pro výpočet algoritmu je zobrazeno druhé okno, které zobrazuje výsledný obrázek. Do konsole je také současně vypsána informace o době výpočtu metody.

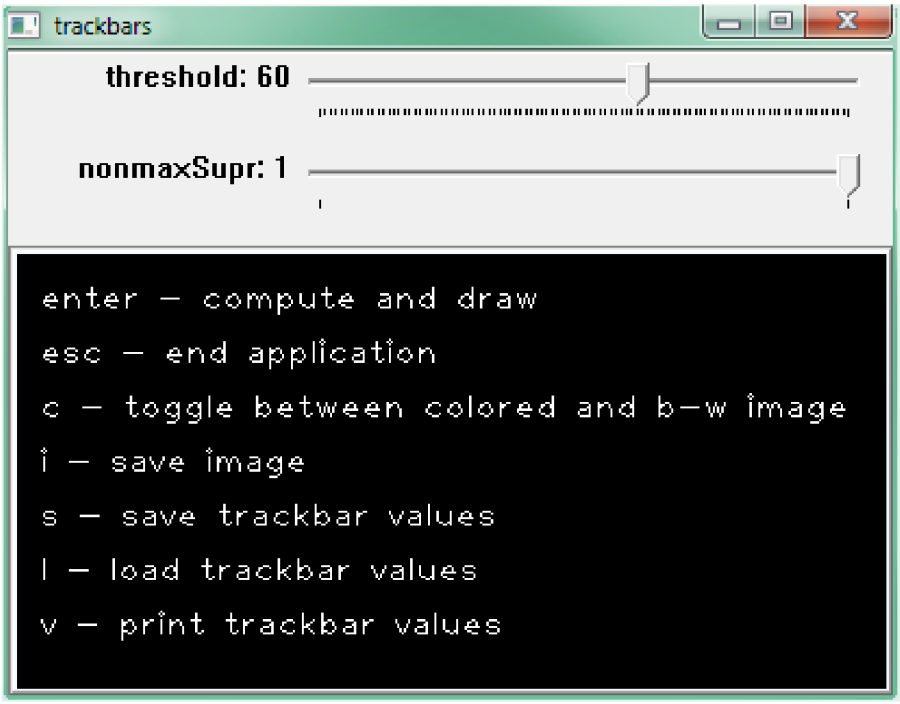
Jelikož některé parametry nabývají velice vysokých hodnot, bylo by značně nevhodné vytvářet trackbary, na kterých by bylo možné nastavit všechny tyto hodnoty. Z tohoto důvodu je číslo indikující stav trackbaru v aplikaci pouze orientační a skutečná hodnota parametru předávaná metodě je zobrazena při stisku klávesy **V**.

7.2 Ovládaní programu

Program lze ovládat pomocí stisku kláves, přičemž interaguje pouze na klávesy vypsané v základním okně 7.1. Základní ovládaní všech aplikací vypadá následovně:

Enter Provede výpočet metody s hodnotami předem nastavených trackbarů a zobrazí výsledný obrázek. V případě, že se během výpočtu vyskytla chyba, je vypsána do konsolového okna.

Esc Ukončí aplikaci.



Obrázek 7.1: Základní okno s ovládáním aplikace pro algoritmus FAST

C Přepíná mezi načítáním obrázku jako barevného nebo gray-scale.

I Uloží obrázek do souboru, jehož název zadá uživatel po výzvě do konsolového okna.

S Uloží nastavení parametrů aplikace do XML/YAML souboru, jehož jméno zadá uživatel po výzvě do konsolového okna.

L Načte nastavení parametrů aplikace z XML/YAML souboru, jehož jméno zadá uživatel po výzvě do konsolového okna.

V Vypíše nastavení parametrů aplikace do konsolového okna.

7.3 Popis jednotlivých demo aplikací

Ač byla kostra aplikace navržena stejně pro všechny metody, jsou výsledné aplikace více či méně odlišné. U každé z metod je uveden stručný popis trackbarů a jejich vliv na detekci. V případě, že aplikace interaguje navíc na některou další klávesu, je tato skutečnost také uvedena. Aplikace se spouštějí z příkazového řádku, přičemž mají jeden povinný parametr, který udává jméno vstupního obrázku. Výjimku tvoří aplikace One-way deskriptor s dvěma povinnými parametry obrázků, ve kterých bude metoda detekovat vzájemné podobnosti elementů.

FAST corner detector

Demonstrační aplikace FAST detektoru je jednou z nejjednodušších, jelikož obsahuje pouze dva trackbary. Na prvním lze nastavit práh detekce a druhý slouží k zapnutí/vypnutí funkce potlačení blízkých bodů. Ostatně právě tato aplikace je zobrazena na obrázku 7.1.

Je vhodné FAST detektor používat s aktivovaným potlačením blízkých bodů, jelikož snížení rychlosti detektoru není nikterak výrazné. Optimální hodnota prahu je velice variabilní a závisí na konkrétním obrázku. Detekci lze spustit pouze na **gray-scale obrázku**.

Maximally stable extremal regions

V MSER aplikaci lze nastavit všechny parametry konstruktoru, které jsou popsány v 3.3. Trackbary parametrů `_max_evolution`, `_area_threshold`, `_min_margin` a `_edge_blur_size` jsou zobrazeny pouze při detekci regionů v barevném obrázku, jelikož při gray-scale detekci nemají žádný vliv. Tato aplikace reaguje navíc na stisk klávesy **T**, která volí způsob interpretace detekovaných regionů. Implementováno je obarvení pixelů patřících jednomu regionu stejnou, náhodně generovanou barvou a taktéž aproximace regionů pomocí elips.

LDetector

V aplikaci znázorňující práci LDetectoru jsou zobrazeny trackbary pro nastavení všech parametrů konstruktoru a také trackbar pro nastavení maximálního počtu detekovaných bodů `maxCount`. Pomocí klávesy **T** lze přepínat mezi metodami `getMostStable2D` a `operator`. Při zvolení metody `operator` je zobrazen přepínač `scaleCoords`, jenž určuje, zda bude pozice detekovaného bodu mapována zpět do původního obrázku v závislosti na měřítku pyramidy, ve které byl bod detekován. Tuto vlastnost je výhodné mít aktivovanou.

HOG people and object detector

Demonstrační aplikace HOG detektoru znázorňuje použití metody `detectMultiScale` a vliv nastavení všech parametrů této metody při detekci osob v obraze. Objekt `HOGDescriptor` je konstruován s defaultními parametry uvedenými v 5.1. Zajímavé výsledky vykazuje kombinace parametrů `scale`, který udává míru zmenšování obrázku, a `groupThreshold`, jenž naopak sjednocuje podobné detekované obdélníky v jednotlivých měřítcích obrázku. Pro detekci osob v obraze je nutné mít parametr `hitThreshold` nastavený na 0.

One-way descriptor

Aplikace znázorňující použití One-way deskriptoru je schopna vyhledat podobnost vzájemných elementů ve dvou zadaných obrázcích. V aplikaci lze nastavit velikost plochy kolem jednotlivých význačných bodů, ze které budou získávány one-way deskriptory a dále také počet póz (afinních transformací), jenž budou použity na každou z ploch při tvorbě deskriptorů. Posledním parametrem, kterým lze ovlivnit výstupní obrázek, je přesnost SURF algoritmu, jenž je použit pro detekci význačných bodů před samotnou klasifikací.

Ke korektnímu běhu aplikace je nutné mít ve shodném adresáři i dva trénovací obrázky, jejichž pojmenování musí být, `one_way_train_0000.jpg` a `one_way_train_0001.jpg`. Trénování je prováděno pouze při prvním spuštění aplikace. Výstupem jsou soubory `pca_descriptors.yml`, `pca_hr.yml` a `pca_lr.yml`. Při dalším spuštění jsou tyto soubory upřednostněny před novým trénováním deskriptorů. Jelikož musí velikost plochy i počet póz odpovídat, je tato skutečnost před samotným spuštěním detekce zkontrolována. V případě nesouhlasu některého z hodnot je vypsáno varování do konsolového okna. Na toto lze reagovat dvěma způsoby: Buď můžeme změnit nastavení trackbarů aplikace tak, aby odpovídalo hodnotám v souborech. Pokud ale trváme na námi zadaném nastavení, je nutno

vymazat všechny tři YAML soubory uvedené výše. Teprve poté je korektně spuštěno nové trénování deskriptorů.

Tato aplikace vychází z vzorového příkladu One-way deskriptoru OpenCV, který je také uveden v příloze B.

Kapitola 8

Závěr

Cílem této bakalářské práce bylo identifikovat a popsat základní činnosti při práci s knihovnou OpenCV a vytvořit příklady, které budou používat identifikované operace. Na základě těchto zkušeností také vytvořit tutoriál, který umístit na web.

Po konzultaci s vedoucím mé bakalářské práce jsem se rozhodl věnovat nezdokumentovaným metodám, které přibyly v knihovně OpenCV s vydáním její verze 2.0 v září roku 2009. Nacházelo se zde mnoho zajímavých metod, které však bylo velice těžké použít. Má práce tedy z největší části spočívala v analýze zdrojových kódů těchto metod a různými experimenty s nimi.

Podařilo se mi zdokumentovat celkem 5 metod, které jsou uvedeny v kapitolách 2–6, přičemž výstupy detektorů jsou zobrazeny na obrázku získaného z [1]. Pro každou z těchto metod jsem také vytvořil aplikaci, demonstrující její základní použití. Na těchto aplikacích si může uživatel odzkoušet vliv nastavení klíčových parametrů na funkčnost metody. Nastavení parametrů lze uložit do XML či YAML souboru, což je vhodné zejména pro uchování optimální konfigurace metod. Podrobnější popis aplikace lze nalézt v kapitole 7.

Jako základ pro tutoriál na webu jsem se rozhodl použít DokuWiki[5]. Webové stránky obsahují stejně jako tato bakalářská práce pro každou metodu popis činnosti, deklaraci metod a princip použití. Navíc lze u každé z metod stáhnout její přeloženou demonstrační aplikaci, či zdrojové kódy. Výsledné webové stránky lze nalézt na adrese `http://www.stud.fit.vutbr.cz/~xbehal01/opencv/`.

Knihovna OpenCV je nepřetržitě rozšiřována, což s sebou přináší i stále nové algoritmy bez patřičné dokumentace. Nyní je aktuální dubnová verze 2.1, ve které sice přibyl příklad pro práci s One-way deskriptorem diskutovaným v kapitole 6, většina metod však bohužel zůstala dále nezdokumentována. Kompletní seznam změn lze nalézt na oficiálních stránkách knihovny[1].

Dalším pokračováním by logicky mělo být rozšíření tutoriálu o další algoritmy spolu s jejich demonstračními příklady. Dále by bylo vhodné vytvořit anglickou verzi stránek, což by usnadnilo získávání informací cizím uživatelům.

[1] `http://opencv.willowgarage.com/wiki/OpenCV%20Change%20Logs`

Literatura

[1] Robotics Research Group. 2009, [Online; navštíveno 30. 4. 2010].
URL http://www.robots.ox.ac.uk/~vgg/data/data-aff.html

[2] Bradski, G.; Kaehler, A.: *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, první vydání, September 2008, ISBN 0596516134.
URL http://www.worldcat.org/isbn/0596516134

[3] Dalal, N.; Triggs, B.: Histograms of Oriented Gradients for Human Detection. In *CVPR '05: Proceedings of the 2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05) - Volume 1*, ročník 1, Washington, DC, USA: IEEE Computer Society, 2005, ISBN 0-7695-2372-2, ISSN 1063-6919, s. 886–893.
URL http://lear.inrialpes.fr/people/triggs/pubs/Dalal-cvpr05.pdf

[4] Forssen, P.: Maximally Stable Colour Regions for Recognition and Matching. In *CVPR07*, 2007, s. 1–8.
URL http://www.cs.ubc.ca/~perfo/papers/forssen_cvpr07.pdf

[5] Gohr, A.: DokuWiki. 2004–2010.
URL http://www.dokuwiki.org/

[6] Hinterstoisser, S.; Benhimane, S.; Navab, N.; aj.: Online learning of patch perspective rectification for efficient object detection. In *2008 IEEE Conference on Computer Vision and Pattern Recognition*, IEEE, June 2008, ISBN 978-1-4244-2242-5, s. 1–8.
URL http://campar.in.tum.de/pub/hinterstoisser2008leopar/hinterstoisser2008leopar.pdf

[7] Hinterstoisser, S.; Kutter, O.; Navab, N.; aj.: Real-time learning of accurate patch rectification. *Computer Vision and Pattern Recognition, IEEE Computer Society Conference on*, ročník 0, 2009: s. 2945–2952.
URL http://cvlab.epfl.ch/publications/publications/2009/Fua,L09.pdf

[8] Intel: OpenCV (Open Source Computer Vision) Library. 2007.
URL http://opencv.willowgarage.com

[9] Lepetit, V.; Fua, P.: Towards Recognizing Feature Points using Classification Trees. Technická zpráva, 2004.
URL http://cvlab.epfl.ch/~vlepetit/papers/lepetit_tr04.pdf

[10] Matas, J.; Chum, O.; Martin, U.; aj.: Robust wide baseline stereo from maximally stable extremal regions. In *Proceedings of British Machine Vision Conference*,

London, 2002, s. 384–393.
URL http://cmp.felk.cvut.cz/~matas/papers/matas-bmvc02.pdf

[11] Rosten, E.; Drummond, T.: Machine learning for high-speed corner detection. In *In European Conference on Computer Vision*, ročník 1, 2006, s. 430–443.
URL http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=8CD599B91155F67C5E6D9B8F966F045D?doi=10.1.1.60.3991&rep=rep1&type=pdf

[12] Rosten, E.; Porter, R.; Drummond, T.: Faster and Better: A Machine Learning Approach to Corner Detection. *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, ročník 32, č. 1, November 2008: s. 105–119.
URL http://mi.eng.cam.ac.uk/~er258/work/rosten_2008_faster.pdf

[13] Wikipedia: Corner detection — Wikipedia, The Free Encyclopedia. 2010, [Online; navštíveno 15. 4. 2010].
URL http://en.wikipedia.org/w/index.php?title=Corner_detection&oldid=355117801

[14] Wikipedia: Histogram of oriented gradients — Wikipedia, The Free Encyclopedia. 2010, [Online; navštíveno 27. 4. 2010].
URL http://en.wikipedia.org/w/index.php?title=Histogram_of_oriented_gradients&oldid=354614327

[15] Wikipedia: Maximally stable extremal regions — Wikipedia, The Free Encyclopedia. 2010, [Online; navštíveno 12. 4. 2010].
URL http://en.wikipedia.org/w/index.php?title=Maximally_stable_extremal_regions&oldid=350349421

[16] Wojek, C.; Schiele, B.: A Performance Evaluation of Single and Multi-feature People Detection. In *DAGM-Symposium*, 2008, s. 82–91.
URL
http://www.mis.tu-darmstadt.de/sites/default/files/wojek08dagma.pdf

Příloha A

Dokumentace tříd OneWayDescriptor a OneWayDescriptorObject

A.1 Třída OneWayDescriptor

Třída `OneWayDescriptor` zahrnuje deskriptor pro jeden bod.

```
OneWayDescriptor();
```

```
void Allocate(int pose_count, Size size, int nChannels);
```

Alokuje pamět pro deskriptor s danými parametry.

pose_count Počet póz

size Velikost

nChannels Počet kanálů obrázku

```
void GenerateSamples(int pose_count, IplImage* frontal, int
   norm = 0);
```

Vygeneruje afinně transformované plochy, jejichž hodnota bude tvořena zprůměrováním všech transformačních změn. Pokud jsou specifikovány transformace, použije je místo náhodně generovaných.

pose_count Počet póz, které bude generovat

frontal Vstupní obrázek

norm Pokud je nenulový, provede normalizaci výstupních ploch tak, aby součet intenzit pixelů byl 1

```
void GenerateSamplesFast(IplImage* frontal, CvMat*
   pca_hr_avg, CvMat* pca_hr_eigenvectors, OneWayDescriptor*
   pca_descriptors);
```

Vygeneruje afinně transformované plochy, jejichž hodnota bude tvořena zprůměrováním všech transformačních změn, přičemž použije předpočítané transformované PCA složky.

frontal Vstupní obrázek

pca_hr_avg Vektor průměrných PCA hodnot

pca_hr_eigenvectors Pca eigenvektory

pca_descriptors Pole předpočítaných deskriptorů PCA složek, obsahující jejich afinní transformace. První je deskriptor vektoru průměrných hodnot. Zbytek vektorů odpovídá eigenvektorům.

```
void SetTransforms(AffinePose* poses, CvMat** transforms);
```

Nastaví pózy a odpovídající transformace

poses pózy

transforms transformace

```
void Initialize(int pose_count, IplImage* frontal, const
   char* feature_name = 0, int norm = 0);
```

Vytvoří deskriptor

pose_count Počet póz. Pokud byly pózy nastaveny externě, použije předem nastavené místo náhodně generovaných

frontal Vstupní obrázek

feature_name Jméno deskriptoru

norm Pokud je nenulový, provede normalizaci výstupních ploch tak, aby součet intenzit pixelů byl 1

```
void InitializeFast(int pose_count, IplImage* frontal, const
   char* feature_name, CvMat* pca_hr_avg, CvMat*
   pca_hr_eigenvectors, OneWayDescriptor* pca_descriptors);
```

Vytvoří deskriptor, přičemž použije předpočítané deskriptory PCA složek

pose_count Počet póz.

frontal Vstupní obrázek

feature_name Jméno deskriptoru

pca_hr_avg Vektor průměrných PCA hodnot

pca_hr_eigenvectors Pca eigenvektory

pca_descriptors Pole předpočítaných deskriptorů PCA složek obsahující jejich afinní transformace. První je deskriptor vektoru průměrných hodnot. Zbytek vektorů odpovídá eigenvektorům.

```
void ProjectPCASample(IplImage* patch, CvMat* avg, CvMat*
   eigenvectors, CvMat* pca_coeffs) const;
```

Normalizuje obrazovou plochu do vektoru, který promítne do PCA prostoru

patch Vstupní obrazová plocha

avg PCA vektor průměrných hodnot

eigenvectors PCA eigenvektory

pca_coeffs Výstupní PCA koeficienty

```
void InitializePCACoeffs(CvMat* avg, CvMat* eigenvectors);
```

Promítne všechny pokřivené plochy do PCA prostoru

avg PCA vektor průměrných hodnot

eigenvectors Výstupní PCA koeficienty

```
void EstimatePose(IplImage* patch, int& pose_idx, float&
   distance) const;
```

Najde největší podobnost mezi vstupní plochou a sadou odlišných ploch s jinými pohledy.

patch Vstupní obrazová plocha

pose_idx Výstupní index nejbližší pózy

distance Vzdálenost k nejbližší póze

```
void EstimatePosePCA(IplImage* patch, int& pose_idx, float&
   distance, CvMat* avg, CvMat* eigenvalues) const;
```

Najde největší podobnost mezi vstupní plochou a sadou odlišných ploch s jinými pohledy.

patch Vstupní obrazová plocha

pose_idx Výstupní index nejbližší pózy

distance Vzdálenost k nejbližší póze

avg PCA vektor průměrných hodnot

eigenvalues PCA eigenvektory

```
Size GetPatchSize() const;
```

Vrátí velikost každé obrázkové plochy po deformaci.

```
Size GetInputPatchSize() const;
```

Vrátí požadovanou velikost plochy, ze které byl deskriptor sestaven.

```
IplImage* GetPatch(int index);
```

Vrátí plochu odpovídající póze na indexu `index`.

```
AffinePose GetPose(int index) const;
```

Vrátí pózu odpovídající póze na indexu `index`.

```
void Save(const char* path);
```

Uloží všechny plochy s odlišnými pózami na cestu `path`.

```
int ReadByName(CvFileStorage* fs, CvFileNode* parent, const
   char* name);
```

Načte deskriptor z FileStorage. Vrací 1 pokud se operace podařila, jinak 0.

fs Jméno souboru

parent Jméno otcovského uzlu

name Jméno uzlu

```
void Write(CvFileStorage* fs, const char* name);
```

Zapíše deskriptor do FileStorage.

fs Jméno souboru

name Jméno uzlu

```
const char* GetFeatureName() const;
```

Vrátí jméno odpovídající význačnému rysu.

```
Point GetCenter() const;
```

Vrátí středový bod význačného rysu.

```
void SetPCADimHigh(int pca_dim_high);
```

Nastaví počet PCA složek pro generaci afinních trasformací na `pca_dim_high`.

```
void SetPCADimLow(int pca_dim_low);
```

Nastaví počet PCA složek složících k porovnávání na `pca_dim_low`.

A.2 Třída OneWayDescriptorObject

Třída `OneWayDescriptorObject` zahrnuje stejnou funkcionalitu jako třída `OneWayDescriptorBase`, popsaná v sekci 6.1, má však odlišný konstruktor a navíc tyto metody:

```
OneWayDescriptorObject(Size patch_size, int pose_count, const
   char* train_path, const char* pca_config, const char*
   pca_hr_config = 0, const char* pca_desc_config = 0, int
   pyr_levels = 2);
```

Vytvoří instanci třídy OneWayDescriptorObject ze souboru trénovacích souborů.

patch_size Velikost vstupních ploch

pose_count Počet póz, které se budou generovat pro každý detektor

train_path Cesta k trénovacím souborům

pca_config Jméno souboru, který obsahuje PCA pro malé plochy.

pca_hr_config Jméno souboru, který obsahuje PCA pro velké plochy.(velikosti vstupních ploch)

pca_desc_config Jméno souboru, který obsahuje deskriptory PCA složek

pyr_levels Úroveň pyramid generovaných z obrázku

```
void Allocate(int train_feature_count, int
   object_feature_count);
```

Naalokuje paměť pro `object_feature_count` deskriptorů.

```
void SetLabeledFeatures(const vector<KeyPoint>& features);
```

Nastaví vnitřní reprezentaci význačných bodů na `features`.

```
vector<KeyPoint>& GetLabeledFeatures();
```

Vrátí vektor význačných bodů.

```
const vector<KeyPoint>& GetLabeledFeatures();
```

Vrátí konstantní vektor význačných bodů.

```
int IsDescriptorObject(int desc_idx) const;
```

Vrací 1, pokud existuje deskriptor na indexu `desc_idx`.

```
int MatchPointToPart(Point pt) const;
```

Vrací index význačného rysu. V případě, že zadanému bodu neodpovídá žádný význačný rys, vrací -1.

```
int GetDescriptorPart(int desc_idx) const;
```

Vrací id deskriptoru na indexu `desc_idx`.

```
const vector<KeyPoint>& GetTrainFeatures();
```

Vrací konstantní set trénovacích význačných bodů.

```
vector<KeyPoint> _GetTrainFeatures() const;
```

Vrací set trénovacích význačných bodů.

```
void InitializeObjectDescriptors(IplImage* train_image, const
   vector<KeyPoint>& features, const char* feature_label, int
   desc_start_idx = 0, float scale = 1.0f);
```

Inicializuje deskriptory pro každý z předaných význačných bodů.

train_image Trénovací obrázek

features Plocha obrázku

feature_label Jméno deskriptorů

desc_start_idx Začátek indexace deskriptorů

scale Měřítko

Příloha B

Zdrojový kód použití One-way deskriptoru

Zdrojový kód včetně trénovacích a testovacích obrázků lze nalézt ve vzorových příkladech knihovny *OpenCV 2.1*.

```
1  /*
2   *  one_way_sample.cpp
3   *  outlet_detection
4   *
5   *  Created by Victor  Eruhimov on 8/5/09.
6   *  Copyright 2009 Argus Corp. All rights reserved.
7   *
8   */
9
10 #include <cv.h>
11 #include <cvaux.h>
12 #include <highgui.h>
13
14 #include <string>
15
16 using namespace cv;
17
18 IplImage* DrawCorrespondences(IplImage* img1, const
      vector<KeyPoint>& features1, IplImage* img2, const
      vector<KeyPoint>& features2, const vector<int>& desc_idx);
19 void generatePCADescriptors(const char* img_path, const char*
      pca_low_filename, const char* pca_high_filename, const
      char* pca_desc_filename, CvSize patch_size);
20
21 int main(int argc, char** argv)
22 {
23     const char pca_high_filename[] = "pca_hr.yml";
24     const char pca_low_filename[] = "pca_lr.yml";
25     const char pca_desc_filename[] = "pca_descriptors.yml";
26     const CvSize patch_size = cvSize(24, 24);
```

```
27      const int pose_count = 50;
28
29      if(argc != 3 && argc != 4)
30      {
31          printf("Format: \n./one_way_sample [path_to_samples]
               [image1] [image2]\n");
32          printf("For example: ./one_way_sample
               ../../../opencv/samples/c scene_l.bmp
               scene_r.bmp\n");
33          return 0;
34      }
35
36      std::string path_name = argv[1];
37      std::string img1_name = path_name + "/" +
           std::string(argv[2]);
38      std::string img2_name = path_name + "/" +
           std::string(argv[3]);
39
40      CvFileStorage* fs = cvOpenFileStorage("pca_hr.yml", NULL,
           CV_STORAGE_READ);
41      if(fs == NULL)
42      {
43          printf("PCA data is not found, starting
               training...\n");
44          generatePCADescriptors(path_name.c_str(),
               pca_low_filename, pca_high_filename,
               pca_desc_filename, patch_size);
45      }
46      else
47      {
48          cvReleaseFileStorage(&fs);
49      }
50
51
52      printf("Reading the images...\n");
53      IplImage* img1 = cvLoadImage(img1_name.c_str(),
           CV_LOAD_IMAGE_GRAYSCALE);
54      IplImage* img2 = cvLoadImage(img2_name.c_str(),
           CV_LOAD_IMAGE_GRAYSCALE);
55
56      // extract keypoints from the first image
57      vector<KeyPoint> keypoints1;
58      SURF surf_extractor(5.0e3);
59  //    printf("Extracting keypoints\n");
60      surf_extractor(img1, Mat(), keypoints1);
61      printf("Extracted %d keypoints...\n",
           (int)keypoints1.size());
62
```

```
63      printf("Training one way descriptors...");
64      // create descriptors
65      OneWayDescriptorBase descriptors(patch_size, pose_count,
           ".", pca_low_filename, pca_high_filename,
           pca_desc_filename);
66      descriptors.CreateDescriptorsFromImage(img1, keypoints1);
67      printf("done\n");
68
69      // extract keypoints from the second image
70      vector<KeyPoint> keypoints2;
71      surf_extractor(img2, Mat(), keypoints2);
72      printf("Extracted %d keypoints from the second
           image...\n", (int)keypoints2.size());
73
74
75      printf("Finding nearest neighbors...");
76      // find NN for each of keypoints2 in keypoints1
77      vector<int> desc_idx;
78      desc_idx.resize(keypoints2.size());
79      for(size_t i = 0; i < keypoints2.size(); i++)
80      {
81          int pose_idx = 0;
82          float distance = 0;
83          descriptors.FindDescriptor(img2, keypoints2[i].pt,
               desc_idx[i], pose_idx, distance);
84      }
85      printf("done\n");
86
87      IplImage* img_corr = DrawCorrespondences(img1,
           keypoints1, img2, keypoints2, desc_idx);
88
89      cvNamedWindow("correspondences", 1);
90      cvShowImage("correspondences", img_corr);
91      cvWaitKey(0);
92
93      cvReleaseImage(&img1);
94      cvReleaseImage(&img2);
95      cvReleaseImage(&img_corr);
96  }
97
98  IplImage* DrawCorrespondences(IplImage* img1, const
       vector<KeyPoint>& features1, IplImage* img2, const
       vector<KeyPoint>& features2, const vector<int>& desc_idx)
99  {
100     IplImage* img_corr = cvCreateImage(cvSize(img1->width +
           img2->width, MAX(img1->height, img2->height)),
           IPL_DEPTH_8U, 3);
```

```
101     cvSetImageROI(img_corr, cvRect(0, 0, img1->width,
            img1->height));
102     cvCvtColor(img1, img_corr, CV_GRAY2RGB);
103     cvSetImageROI(img_corr, cvRect(img1->width, 0,
            img2->width, img2->height));
104     cvCvtColor(img2, img_corr, CV_GRAY2RGB);
105     cvResetImageROI(img_corr);
106
107     for(size_t i = 0; i < features1.size(); i++)
108     {
109         cvCircle(img_corr, features1[i].pt, 3, CV_RGB(255, 0,
                0));
110     }
111
112     for(size_t i = 0; i < features2.size(); i++)
113     {
114         CvPoint pt = cvPoint(features2[i].pt.x + img1->width,
                features2[i].pt.y);
115         cvCircle(img_corr, pt, 3, CV_RGB(255, 0, 0));
116         cvLine(img_corr, features1[desc_idx[i]].pt, pt,
                CV_RGB(0, 255, 0));
117     }
118
119     return img_corr;
120 }
121
122 /*
123  *  pca_features
124  *
125  *
126  */
127
128 void savePCAFeatures(const char* filename, CvMat* avg, CvMat*
        eigenvectors)
129 {
130     CvMemStorage* storage = cvCreateMemStorage();
131
132     CvFileStorage* fs = cvOpenFileStorage(filename, storage,
            CV_STORAGE_WRITE);
133     cvWrite(fs, "avg", avg);
134     cvWrite(fs, "eigenvectors", eigenvectors);
135     cvReleaseFileStorage(&fs);
136
137     cvReleaseMemStorage(&storage);
138 }
139
140 void calcPCAFeatures(vector<IplImage*>& patches, const char*
        filename, CvMat** avg, CvMat** eigenvectors)
```

```
141 {
142     int width = patches[0]->width;
143     int height = patches[0]->height;
144     int length = width*height;
145     int patch_count = (int)patches.size();
146 
147     CvMat* data = cvCreateMat(patch_count, length, CV_32FC1);
148     *avg = cvCreateMat(1, length, CV_32FC1);
149     CvMat* eigenvalues = cvCreateMat(1, length, CV_32FC1);
150     *eigenvectors = cvCreateMat(length, length, CV_32FC1);
151 
152     for(int i = 0; i < patch_count; i++)
153     {
154         float sum = cvSum(patches[i]).val[0];
155         for(int y = 0; y < height; y++)
156         {
157             for(int x = 0; x < width; x++)
158             {
159                 *((float*)(data->data.ptr + data->step*i) +
                       y*width + x) = (float)(unsigned char)
                       patches[i]->imageData[y*patches[i]->widthStep
                       + x]/sum;
160             }
161         }
162     }
163 
164     printf("Calculating PCA...");
165     cvCalcPCA(data, *avg, eigenvalues, *eigenvectors,
          CV_PCA_DATA_AS_ROW);
166     printf("done\n");
167 
168     // save pca data
169     savePCAFeatures(filename, *avg, *eigenvectors);
170 
171     cvReleaseMat(&data);
172     cvReleaseMat(&eigenvalues);
173 }
174 
175 
176 void loadPCAFeatures(const char* path, vector<IplImage*>&
       patches, CvSize patch_size)
177 {
178     const int file_count = 2;
179     for(int i = 0; i < file_count; i++)
180     {
181         char buf[1024];
182         sprintf(buf, "%s/one_way_train_%04d.jpg", path, i);
183         printf("Reading image %s...", buf);
```

```
184         IplImage* img = cvLoadImage(buf,
               CV_LOAD_IMAGE_GRAYSCALE);
185         printf("done\n");
186
187         vector<KeyPoint> features;
188         SURF surf_extractor(1.0f);
189         printf("Extracting SURF features...");
190         surf_extractor(img, Mat(), features);
191         printf("done\n");
192
193         for(int j = 0; j < (int)features.size(); j++)
194         {
195             int patch_width = patch_size.width;
196             int patch_height = patch_size.height;
197
198             CvPoint center = features[j].pt;
199
200             CvRect roi = cvRect(center.x - patch_width/2,
                   center.y - patch_height/2, patch_width,
                   patch_height);
201             cvSetImageROI(img, roi);
202             roi = cvGetImageROI(img);
203             if(roi.width != patch_width || roi.height !=
                   patch_height)
204             {
205                 continue;
206             }
207
208             IplImage* patch =
                   cvCreateImage(cvSize(patch_width,
                   patch_height), IPL_DEPTH_8U, 1);
209             cvCopy(img, patch);
210             patches.push_back(patch);
211             cvResetImageROI(img);
212
213         }
214
215         printf("Completed file %d, extracted %d features\n",
               i, (int)features.size());
216
217         cvReleaseImage(&img);
218     }
219 }
220
221 void generatePCAFeatures(const char* img_filename, const
       char* pca_filename, CvSize patch_size, CvMat** avg,
       CvMat** eigenvectors)
222 {
```

```
223     vector<IplImage*> patches;
224     loadPCAFeatures(img_filename, patches, patch_size);
225     calcPCAFeatures(patches, pca_filename, avg, eigenvectors);
226 }
227
228 void generatePCADescriptors(const char* img_path, const char*
        pca_low_filename, const char* pca_high_filename, const
        char* pca_desc_filename, CvSize patch_size)
229 {
230     CvMat* avg_hr;
231     CvMat* eigenvectors_hr;
232     generatePCAFeatures(img_path, pca_high_filename,
            patch_size, &avg_hr, &eigenvectors_hr);
233
234     CvMat* avg_lr;
235     CvMat* eigenvectors_lr;
236     generatePCAFeatures(img_path, pca_low_filename,
            cvSize(patch_size.width/2, patch_size.height/2),
237          &avg_lr, &eigenvectors_lr);
238
239     const int pose_count = 500;
240     OneWayDescriptorBase descriptors(patch_size, pose_count);
241     descriptors.SetPCAHigh(avg_hr, eigenvectors_hr);
242     descriptors.SetPCALow(avg_lr, eigenvectors_lr);
243
244     printf("Calculating %d PCA descriptors (you can grab a
            coffee, this will take a while)...\n",
            descriptors.GetPCADimHigh());
245     descriptors.InitializePoseTransforms();
246     descriptors.CreatePCADescriptors();
247     descriptors.SavePCADescriptors(pca_desc_filename);
248
249     cvReleaseMat(&avg_hr);
250     cvReleaseMat(&eigenvectors_hr);
251     cvReleaseMat(&avg_lr);
252     cvReleaseMat(&eigenvectors_lr);
253 }
```