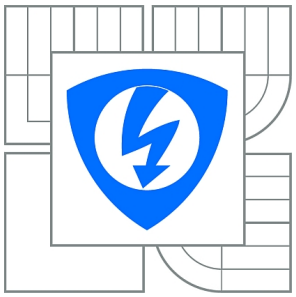
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA ELEKTROTECHNIKY A KOMUNIKAČNÍCH TECHNOLOGIÍ

ÚSTAV TELEKOMUNIKACÍ

FACULTY OF ELECTRICAL ENGINEERING AND COMMUNICATION

DEPARTMENT OF TELECOMMUNICATIONS

OVLÁDÁNÍ LINUXU POMOCÍ KAMERY

LINUX USER INTERFACE USING CAMERA

BAKALÁŘSKÁ PRÁCE

BACHELOR'S THESIS

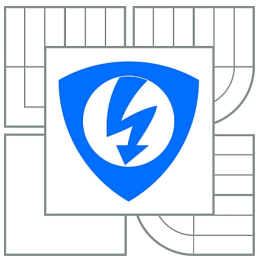
AUTOR PRÁCE — PETR DOLNÍČEK

AUTHOR

VEDOUCÍ PRÁCE — Ing. ONDŘEJ ŠMIRG

SUPERVISOR

BRNO 2010



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ

Fakulta elektrotechniky a komunikačních technologií

Ústav telekomunikací

Bakalářská práce

bakalářský studijní obor
Teleinformatika

Student: Petr Dolníček ***ID:*** 111033
Ročník: 3 ***Akademický rok:*** 2009/2010

NÁZEV TÉMATU:

Ovládání Linuxu pomocí kamery

POKYNY PRO VYPRACOVÁNÍ:

Pomocí kamery připojené přes USB detekujte pohyb uživatelovy ruky (může být vylepšeno například barevným značením) a pomocí zjištěných dat ovládejte kurzor myši. Program vytvořte v nějakém programovacím jazyce (JAVA, C++)

DOPORUČENÁ LITERATURA:

[1] ŘÍHA, K.: Pokročilé techniky zpracování obrazu. Elektronické texty VUT, Ústav telekomunikací FEKT VUT v Brně, 2007.
[2] ŠMIRG, O., Zpracování obrazu za účelem řízení Visikonu, Bakalářská práce,Brno: VUT, Fakulta elektrotechniky a komunikačních technologií, 2006,50,pp.3
[3] KOHOUTEK,M. Metody analýzy obrazové scény pro řízení videokonferenčních zařízení, Pojednání o disertační práci.Brno: VUT Fakulta elektrotechniky a komunikačních technologií, 2006,21s.

Termín zadání: 29.1.2010 ***Termín odevzdání:*** 2.6.2010

Vedoucí práce: Ing. Ondřej Šmirg

prof. Ing. Kamil Vrba, CSc.
Předseda oborové rady

UPOZORNĚNÍ:

ABSTRACT

The goal of this was to create a fully functional program coded in C++, which is capable of real time object detection and mouse positioning in operating system Linux. Object detection is based on recognizing desired color and shape from webcam input. In this case it was a red circle. The main part of source code was generated via application Harpia. This is an application especially created for purposes of object tracking, border detection and picture processing. Most of used functions belong to OpenCV library. This library, as well as Harpia application, was created for computer vision, so it has many functions especially for purposes of my program. You can find many information about edge detection, color filtering and noise reduction in this document. I have also managed to control mouse cursor according to data that program detects. My program fulfils its purpose.

Cílem projektu bylo vytvořit plně funkční program v jazyce C++, který je schopen detekce objektů a ovládání kurzoru myši v operačním systému Linux. Tato detekce je založena na rozpoznávání objektů požadované barvy a tvaru ze vstupu webkamery, v tomto případě sledování červeného kruhu. Hlavní část kódu byla psaná v programu Harpia, který je pro účely zpracovávání obrazu speciálně vytvořen. Většina použitých funkcí je z knihovny OpenCV, která se zabývá počítačovým viděním. V mé práci naleznete informace o způsobech detekce hran, filtraci obrazu a vyhlazovacích filtrech. Program splňuje stanovené zadání, na základě zjištěné polohy detekovaného objektu v obraze ovládá pohyb kurzoru myši.

Prohlášení

Prohlašuji, že svoji bakalářskou práci na téma *Ovládání Linuxu pomocí kamery* jsem vypracoval samostatně pod vedením vedoucího semestrálního projektu 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 uvedeného semestrálního projektu dále prohlašuji, že v souvislosti s vytvořením tohoto projektu 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

Citace práce:

DOLNÍČEK, P. *Ovládání Linuxu pomocí kamery.* Brno: Vysoké učení technické v Brně, Fakulta elektrotechniky a komunikačních technologií, 2010. 42 s. Vedoucí bakalářské práce Ing. Ondřej Šmirg.

Poděkování:

Děkuji vedoucímu bakalářské práce Ing. Ondřeji Šmirgovi za velmi dobrý přístup k projektu, vedení, které mi pomohlo při řešení úkolů a objevování nových poznatků a také za zapůjčení webkamery a rady při seznamování s novým softwarem.

1 Obsah

2 Úvod

V mojí bakalářské práci mám za úkol nastudovat a ověřit metody, které by se daly použít pro ovládání počítače pomocí připojené kamery v programu MATLAB, poté tyto znalosti využít při tvorbě programu v operačním systému Linux. Program MATLAB je velice sofistikovaný a je v něm implementováno obrovské množství příkazů. Vzhledem k zadání bylo velmi výhodné, že tento program zvládá i zpracovávání videosignálu přímo z webkamery laptopu. Dále uvidíme, že při zpracování obrazu používáme množství různých filtrů a detektorů. Tyto funkce pracují s obrazem jako s maticí hodnot reprezentující jednotlivé pixely a naprogramování těchto funkcí není sice nijak zvlášť obtížné, ale je velmi zdlouhavé. Hlavní výhoda MATLABU je ta, že tyto funkce již program má ve svých knihovnách, takže nám stačí pouze funkci vyvolat a zadat parametry. Místo složité mnohořádkové funkce máme jednoduchý příkaz na 1 řádek. Navíc má tento program velice obsáhlou nápovědu, ze které jsem čerpal většinu informací.

Ve svém projektu jsem nejprve začal ověřovat funkce pro zpracování obrazu na obrázcích. Různé vyhlazovací filtry, hranové detektory a podobné funkce pracovaly bezchybně, ale stejně bylo poznat, že počítač na vygenerování výsledných obrazů potřebuje nemalou chvíli a také je velmi zatěžován procesor a paměti. Proto se vyskytly problémy, když jsem tyto funkce implementoval na signál z kamery, který jsem chtěl upravovat v reálném čase. Počítač už nemohl stíhat zpracovávat každý snímek z webkamery dříve, než obdržel následující snímek. Proto jsem musel uplatnit radikální změny, které sice zmenšily nároky na zpracování snímku, ale zhoršily kvalitu výsledného obrazu. Mezi tyto změny patřilo zmenšení rozlišení webkamery, redukce velikosti matice se zpracovávaným obrazem a také zmenšení počtu zachytávaných snímků za sekundu.

Simulace jsem prováděl na svém laptopu, který sice není nejnovější, ale stále má dobré parametry. To znamená, že na současných stolních počítačích poběží program rychleji a může zde být nastavena i větší kvalita zobrazení.

V první části projektu jsem pracoval pouze v MATLABu, ale všechny simulace byly pouze pro seznámení s funkcemi a způsoby detekce objektů, tvarů a barev. Hlavní úkol byl připravit se na implementování těchto funkcí a postupů v operačním systému Linux pomocí programovacího jazyka C++.
K tomu je stvořená knihovna OpenCV (Open Source Computer Vision), ve které

nalezneme všechny funkce potřebné pro zpracování obrazu v reálném čase, rozpoznávání objektů, detekce tvarů, sledování pohybu a gest a mnoho dalších.

V mém hlavním programu jsem si dal za úkol detekovat v reálném čase polohu červeného kruhu přímo z výstupu webkamery a podle toho ovládat kurzor myši. Nejtěžší bylo vytvořit funkci, která detekuje červenou barvu, neboť každý pixel byl uložen pomocí tří hodnot, které reprezentovaly zastoupení jednotlivých základních spektrálních barev.

Téma této bakalářské práce je velmi zajímavé. V současné době se totiž fenomén zvaný *Computer Vision*, neboli počítačové vidění velmi rozmáhá. Na trhu již můžeme vidět herní konzole ovládané různými gesty pomocí kamery, fotoaparáty detekující lidské obličeje a rozpoznávající úsměvy a různé další technické výdobytky. Velké uplatnění má tato technologie v robotice a automatizaci, kde může pomáhat robotům rozpoznávat jednotlivé předměty a automobilům sledovat okraje silnice a překážky ve vozovce.

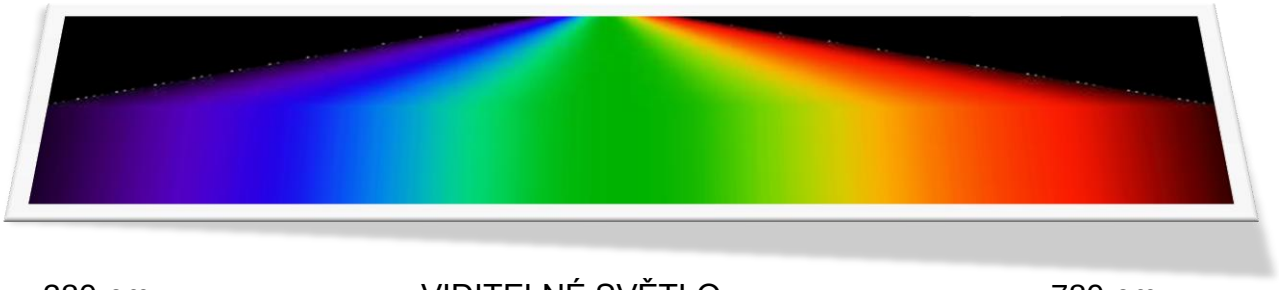
Zmiňován byl také operační systém Linux. Od jeho stvoření kolem roku 1990 ušel obrovskou cestu a v současnosti se stává stále více oblíbeným i na osobních a přenosných počítačích. Jeho největší výhodou je bezesporu to, že je distribuován zdarma a to včetně zdrojového kódu. Na webu najdeme obrovské množství aplikací šířených jako freeware, které si můžeme sami upravovat a dále šířit. Tohle umožňuje v rukou schopných programátorů rychlý rozvoj softwaru pro Linux. V současnosti se začínají objevovat také chytré mobilní telefony pracující s tímto operačním systémem, které zatím sice nemají příliš velkou softwarovou podporu, ale díky vlastnostem a filosofii Linuxu se to doufejme rychle změní.

3 Vlastnosti obrazu

3.1 Definice obrazu

Obraz všeobecně vzniká analýzou paprsků energie odražených od zkoumaného objektu. Obvykle to bývá elektromagnetická energie. Tyto paprsky při kontaktu s objektem ztrácí určitou část svojí původní energie a tím se do nich zaznamená informace o objektu. Po odrazu od objektu, nebo po průchodu tímto objektem zaznamenáme paprsky pomocí nějakého zařízení, nejčastěji kamery. Jednotlivé paprsky v sobě nesou informaci pouze o tom bodě námi zkoumaného objektu, se kterým byly ve styku. To znamená, že pro dostatečné prozkoumání objektu musíme mít také dostatečné množství paprsků a nakonec i schopnost toto množství přijmout a rozlišit. Někdy zkoumáme objekt, který je sám zdrojem elektromagnetické energie, tudíž sám produkuje paprsky, které rovnou zaznamenáváme a zkoumáme. V běžných případech se jedná o spektrum elektromagnetické energie o vlnové délce od 380 nm do 780 nm, což je lidským okem viditelné světlo. Na světlo můžeme pohlížet dvěma různými způsoby. Buď jako na vlnění, nebo jako na proud elementárních částic, kterým říkáme fotony.

Dalším důležitým aspektem světla je jeho barva. Ta je závislá pouze na frekvenci zkoumaného světla. Světlo o frekvenci 380 nm se lidskému oku jeví jako fialové a frekvence 780 nm zase reprezentuje barvu červenou a zbytek barev je rozložen v tomto intervalu.



Obr. 3-1 Barevné spektrum světla

Kromě frekvence světla také zaznamenáváme jeho intenzitu. Je to vlastně množství dopadajících fotonů za jednotku času. V zobrazovací technice se jí říká jas. Záznamová zařízení typu kamery zaznamenávají o dopadajícím světle dva údaje, barvu a jas. V našem případě budeme používat i monochromatické světlo (černobílé), u kterého zaznamenáváme pouze hodnoty jasu a barvu ignorujeme. Toto monochromatické světlo popisujeme pomocí stupňů šedi, které tvoří souvislou stupnici od černé až po bílou. Černá zde vyjadřuje nulovou hodnotu jasu, to znamená, že na snímač nedopadají žádné rozpoznatelné fotony a bílá vyjadřuje maximální hodnotu jasu. Teoreticky tato maximální hodnota není omezena, ale dále uvidíme, že kvůli elektronickému zpracování obrazu musíme maximum stanovit.

Také se nejspíš setkáme s pojmy kontrast a ostrost. Kontrast může být jasový nebo barevný, ale v obou případech se jedná o rozdíl hodnot jasu nebo barvy mezi dvěma sousedními oblastmi, které mají tyto hodnoty konstantní a navzájem rozdílné. Čím větší tento rozdíl je, tím větší je kontrast. Pojem ostrost se týká jasu hran a okolí. Hodnota ostrosti, nebo také zaostření stoupá, pokud se zvyšuje rozdíl mezi jasem hrany zkoumaného objektu a jasem okolního pozadí.

3.2 Zaznamenávání obrazu

Pro naše účely nám postačí, abychom se zabývali pouze zaznamenáváním obrazu do elektronické formy. To probíhá pomocí elektronických součástek citlivých na elektromagnetické záření, které jsou uspořádány většinou do obdélníku. V tomto obdélníku bývá několik tisíc až milionů těchto součástek a každá z nich zaznamenává jeden obrazový bod neboli pixel. Ten většinou nese informaci o jasu dopadajícího záření a také o barvě. V případě monochromatického obrazu tak dostáváme matici o rozměrech a x b, přičemž každý prvek této matice nese informaci o jasu tohoto pixelu. Tato hodnota vyjadřuje celkovou energii zaznamenávaného elektromagnetického záření dopadajícího na plochu tohoto pixelu. Jedná se tedy o jasovou funkci dvou prostorových souřadnic $f(x,y)$. U barevného obrazu pracujeme třírozměrnými maticemi, které dostaneme přidáním informace o barvě k monochromatické matici. V programu MATLAB, se kterým jsem

pracoval, se jedná o přidání dalších tří proměnných ke každému pixelu obrazové matice. Tyto hodnoty nesou informaci o zastoupení jednotlivých barev RED, GREEN a BLUE v každém obrazovém bodě.

První větší problém, se kterým se kterým se zde setkáme, je objem dat. Abychom dosáhli dobré kvality obrazu, musíme snímat pomocí dostatečného množství pixelů. V případě videa musí následovat tyto pixely i dostatečně rychle za sebou. Pokud chceme ještě s obrazem provádět různé úpravy v reálném čase, klade to obrovské nároky na přenos dat a na výpočetní techniku.

3.3 Šum

Pokud dojde k degradaci obrazu, ať už při jeho snímání nebo při zpracování, obraz obsahuje různé nežádoucí poruchy, kterým se říká šum. Kvůli šumu musíme používat při detekování objektů různé filtry, které poruchy vyhladí, ale také zmenší rozlišení detailů v obraze. Při popisu se používá pravděpodobnostních charakteristik a podle vzniku a výskytu dělíme šum do pěti různých skupin.

Bílý šum

Je to matematicky ideální model šumu. Jsou v něm obsaženy všechny frekvence ve stejném zastoupení. To znamená, že se dá nejhůře identifikovat a odstranit.

Aditivní šum

Tento šum je nezávislý na obrazovém signálu a je tvořen přenosovými kanály. Tyto kanály totiž nejsou ideální a přidávají do signálu, který jimi přenášíme šum, jehož intenzita a vlastnosti jsou na přenášeném signálu nezávislé.

Aditivní šum lze popsat vztahem

$$\boldsymbol{f} = \boldsymbol{g}(\boldsymbol{x}, \boldsymbol{y}) + \boldsymbol{v}(\boldsymbol{x}, \boldsymbol{y}) \quad (3\text{-}1)$$

Platí, že šum *v* a vstupní obraz *g* jsou nezávislé veličiny

Multiplikativní šum

Je to případ, kdy velikost vzniklého šumu je na obrazovém signále závislá a úroveň tohoto šumu je oproti signálu dostatečně veliká.

Lze ho popsat vztahem

$$\boldsymbol{f} = \boldsymbol{g} + \boldsymbol{v}\boldsymbol{g} \quad (3\text{-}2)$$

Kvantizační šum

Vzniká při procesu zvaném kvantování, kdy se jedna úroveň signálu, která může být vyjádřena pomocí nekonečně mnoha hodnot, převádí na jinou úroveň, jež se dá vyjádřit pouze omezeným počtem hodnot. V našem případě převádíme jas do matice, kde jeho hodnota může nabývat pouze 256 různých hodnot.

Impulsní šum

Je způsoben chybami v binárních obrazech a jedná se o typ šumu zvaný *pepř a sůl.* Tento typ byl použit při generování šumu pro testování hranových detektorů.

Informace čerpány z [2]

4 Odstraňování šumu a prahování

4.1 Prahování

Již jsem se zmiňoval o problému, který vzniká, když chceme zpracovávat obraz s velmi vysokým rozlišením. Nároky na výpočetní techniku u těchto operací zpracovávajících obraz nejsou malé. Proto v případech, kde si to můžeme dovolit, použijeme operaci prahování. Je to funkce, která nám upraví jasovou matici tak, že hodnoty pod určitou námi stanovenou hranici nastaví na nulu a hodnoty nad tuto hranici na jedničku. Bereme potom pixely tak, že jsou buďto černé nebo bílé a dostaneme tak matici s hodnotami 1 nebo 0. Může to pak snížit systémové nároky a zvýšit rychlost dále prováděných operací.

4.2 Mediánový filtr

K odstranění šumu zde byl použit 2-D mediánový filtr. Jako vstupní parametr pro tento filtr zadáme pouze vstupní matici jasových hodnot a velikost masky filtru. Tato maska udává velikost okolí pixelu, se kterým filtr počítá. Princip je ten, že filtr pro jednotlivé body z původní matice vezme jejich okolí o předem definované velikosti. Poté vytvoří matici o této velikosti, např. 3x3 a do ní vloží hodnotu právě počítaného bodu a bodů z jeho okolí, tyto hodnoty poskládá do řady postupně od nejmenší po největší. Pokud je počet hodnot lichý, výsledek mediánu je prostřední hodnota z řady, pokud je počet hodnot sudý, výsledek je průměr ze dvou prostředních hodnot této řady.

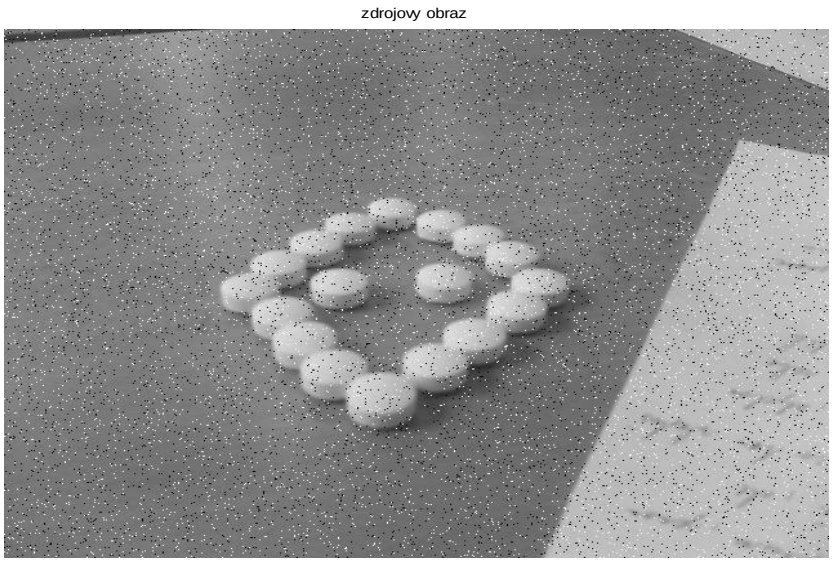
Ve výsledku pak platí, že tento filtr dokáže odstranit osamocené body v obraze, avšak rozmazává hrany. Čím větší je velikost masky, tím větší je patrný efekt. Z matematického hlediska můžeme říct, že filtr se chová jako dolní propust.

Příklad masky filtru pro okolí 3x3

$$\begin{bmatrix} 1 & 1 & 1 \\ 1 & 1 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

Dále jsou uvedeny výsledky mé simulace mediánového filtru. Obrázek byl vytvořen jednoduchým digitálním fotoaparátem a převeden do monochromatického zobrazení. Poté jsem do něj přidal impulsní šum typu *pepř a sůl* a výsledný obraz uvedený na obrázku (4-1) jsem použil jako vstup pro filtr. Nejprve pro filtr s maskou 3x3 a poté, aby bylo lépe vidět princip filtru, jsem použil masu o velikosti 60x60. To ukazují obrázky (4-2) a (4-3).

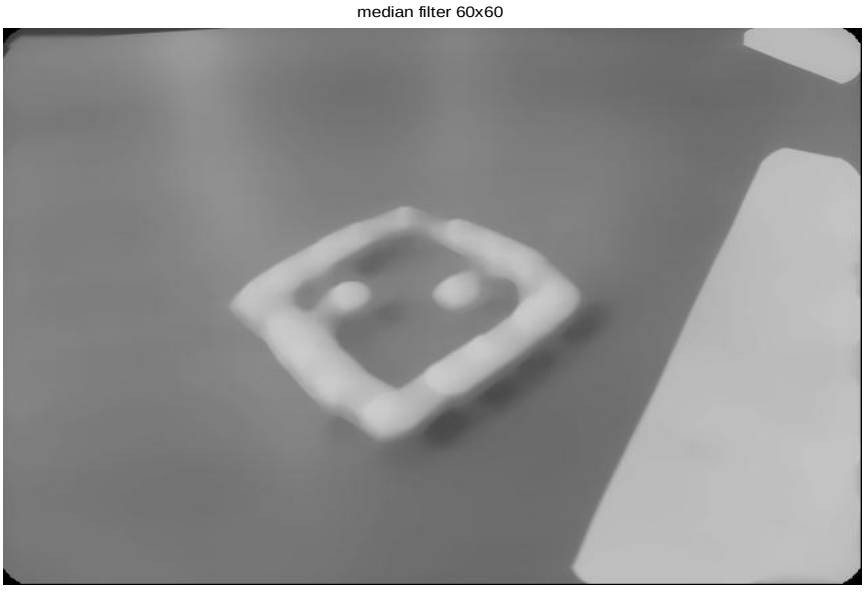
Obr. 4-1 Původní obrázek bez úprav



Obr. 4-2 Upravený zdrojový obraz před filtrováním

Zde jsou výsledky po aplikování mediánového filtru na obrázek (4-2)

Obr. 4-3 Mediánový filtr s maskou 3x3

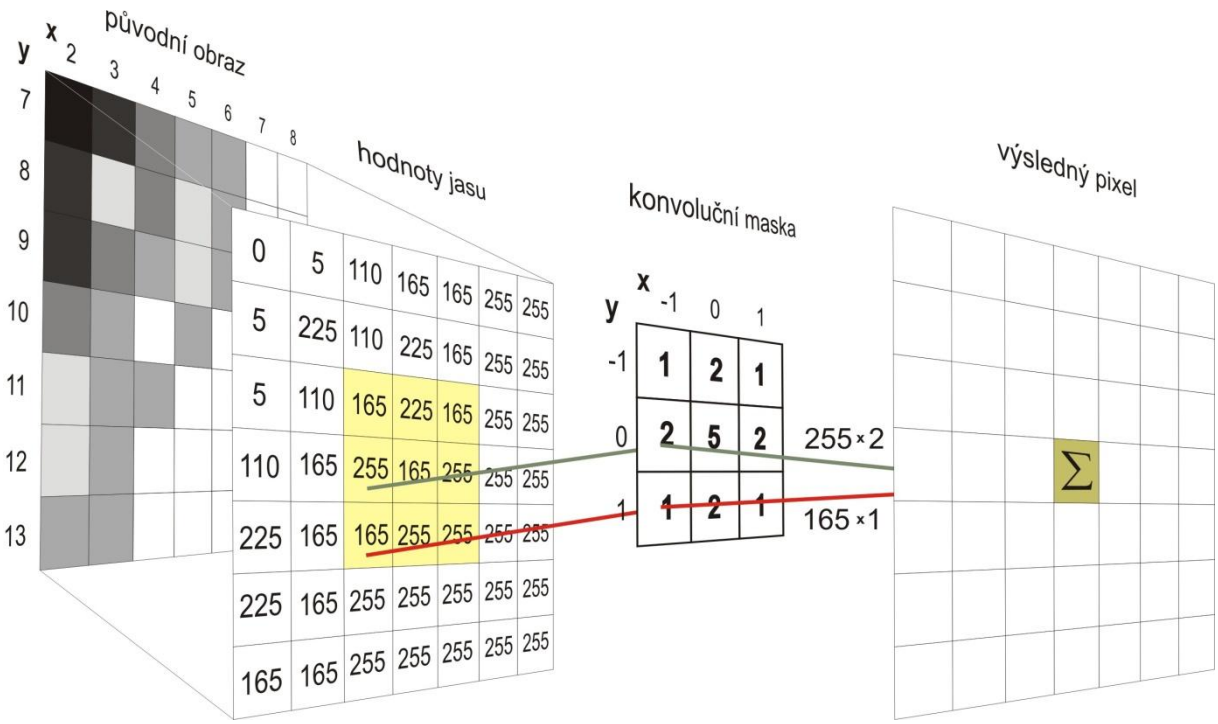


Obr. 4-4 Mediánový filtr s maskou 60x60

Informace čerpány z [3]

5 Maska filtru

Setkali jsme se, a dále ještě mnohokrát setkáme, s pojmem maska filtru. Jedná se většinou o matici o rozměrech 3x3, kterou filtr použije na každý pixel původního obrazu, aby vytvořil výsledný obraz. Princip je vidět na obrázku. Každý pixel původního obrazu a 8 pixelů z jeho nejbližšího okolí se zde násobí čísly v konvoluční masce. Nakonec se sečtou a vyjde hodnota pixelu nového výsledného obrazu. U obrazových filtrů jsou některé hodnoty masky záporné, takže nevychází nesmyslná čísla.



Obr. 5-1 Princip činnosti filtrů

Zdroj: Wikimedia Commons, GNU Free Documentation

Na tomto principu fungují všechny zde uvedené filtry, od mediánového filtru vyhlazujícího šum až po hranové detektory.

6 Hrany a gradientní operátory

6.1 Definice hrany

Nejprve bude asi vhodné popsat si, jak vypadá takzvaná ideální hrana. Je to oblast obrazu, kde mezi dvěma sousedními pixely existuje skoková změna jasu. Většinou jedna oblast pixelů má minimální hodnotu jasu vzhledem k použité soustavě a sousední oblast má maximální hodnotu jasu. Ovšem v reálném obraze takovéto přechody nenajdeme, pokud obraz nepřevedeme do formátu *black and white*.

Mnohem častěji se tedy v praxi setkáme s jevem, kdy mezi těmito dvěma oblastmi najdeme plynulý jasový přechod. Hrana se zdá rozostřená. Hranové detektory pracují často s první a druhou derivací funkce jasu mezi sousedními oblastmi. Představme si hranu jako funkci jasových hodnot závislou na poloze. V případě neostré hrany je tato funkce spojitá a plynule rostoucí z minimální hodnoty na jedné straně hrany až po maximální hodnotu na straně druhé. První derivace této funkce je v oblasti přechodu z jedné hodnoty na druhou nenulová. V oblastech, kde je funkce lineárně rostoucí nebo klesající je první derivace konstantní a v oblastech, kde je funkce jasu konstantní je derivace nulová. To znamená, že první derivace je vhodná k určení, zda se daný obrazový bod nachází v oblasti s gradientním přírůstkem. Pokud chceme určit, zda je pixel na světlé, či tmavé straně hrany, potřebujeme k tomu druhou derivaci. Její funkce je kladná v tmavé části přechodu a záporná na světlé straně. Nulová je tam, kde je jasová funkce konstantní nebo lineárně rostoucí či klesající.

Těchto operací využívají takzvané gradientní operátory, které se dají považovat za opak vyhlazovacích filtrů a chovají se jako horní propust. Na základě první a druhé derivace detekují největší gradienty jasových funkcí a na základě toho označují hrany v obraze. Dělíme je na dvě skupiny.

Izotropní operátory – reagují na všechny hrany bez ohledu na určení směru, tedy horizontální nebo vertikální orientace

Anizotropní operátory – reagují pouze na hrany v určitém směru.

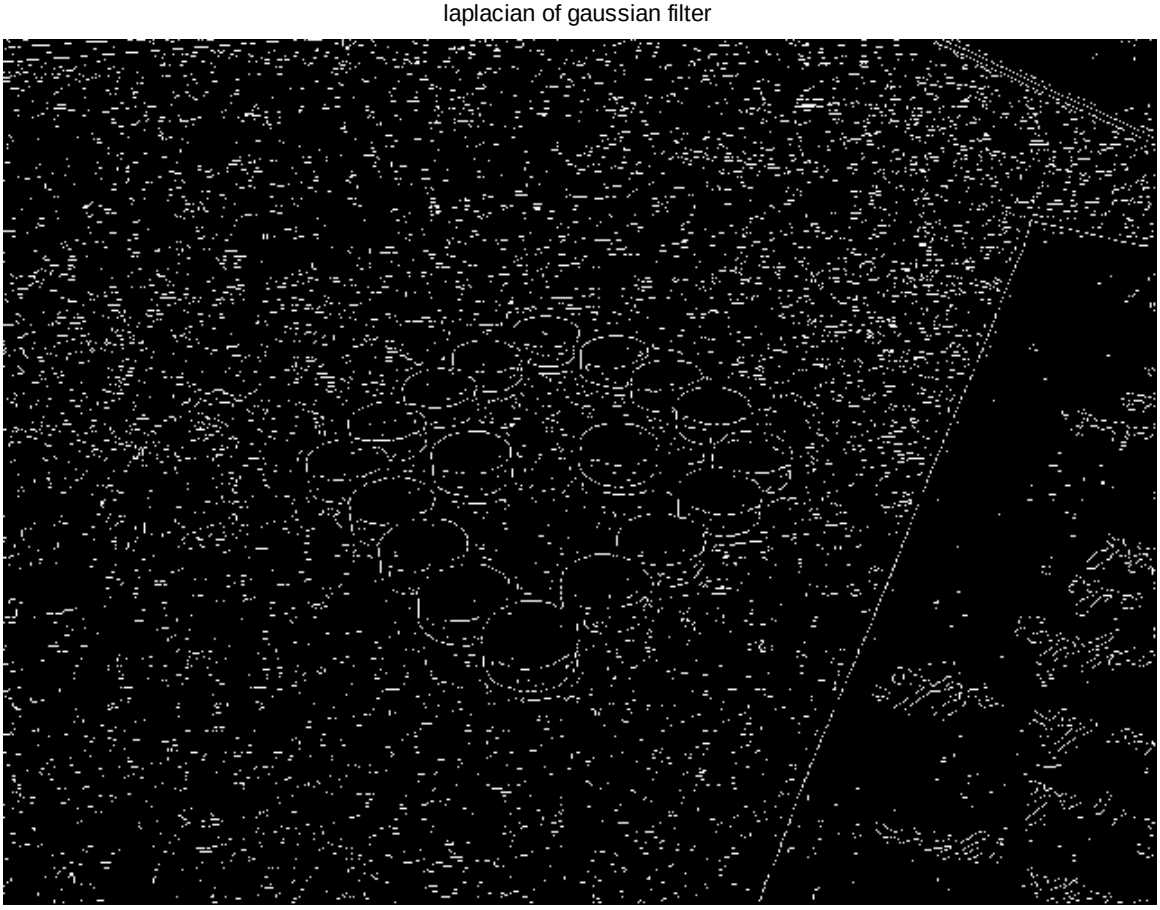
Informace čerpány z [1] a [2]

6.2 Hranové detektory

Následují příklady některých hranových detektorů. Jako vstup pro všechny detektory sloužil již výše uvedený obrázek (4-1). Ten se pak převedl do monochromatického zobrazení a byl uměle přidán šum (viz obrázek 4-2). Každý z těchto detektorů má v programu MATLAB nastavitelné parametry, jejichž nastavováním dosahujeme různých výsledků. V následujících příkladech byly ponechány výchozí nastavení.

6.2.1 Laplacián

Derivativní operátor, jenž zvýrazňuje jas tam, kde se nachází obrazové nespojitosti a naopak tlumí jas tam, kde se jeho hodnota mění pomalu. Metoda, kterou jsem použil, se nazývá *Laplacian of Gaussian* a hledá hrany v obraze přechodem nulou po použití filtru laplacián. Jak vidíme z obrázku, tato metoda je velmi citlivá na šum.



Obr. 6-1 Hranový detektor laplacián

Matematická formulace filtru laplacián:

$$\triangle = \frac{\partial^2}{\partial x^2} + \frac{\partial^2}{\partial y^2} \rightarrow \begin{pmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{pmatrix} \tag{6-1}$$

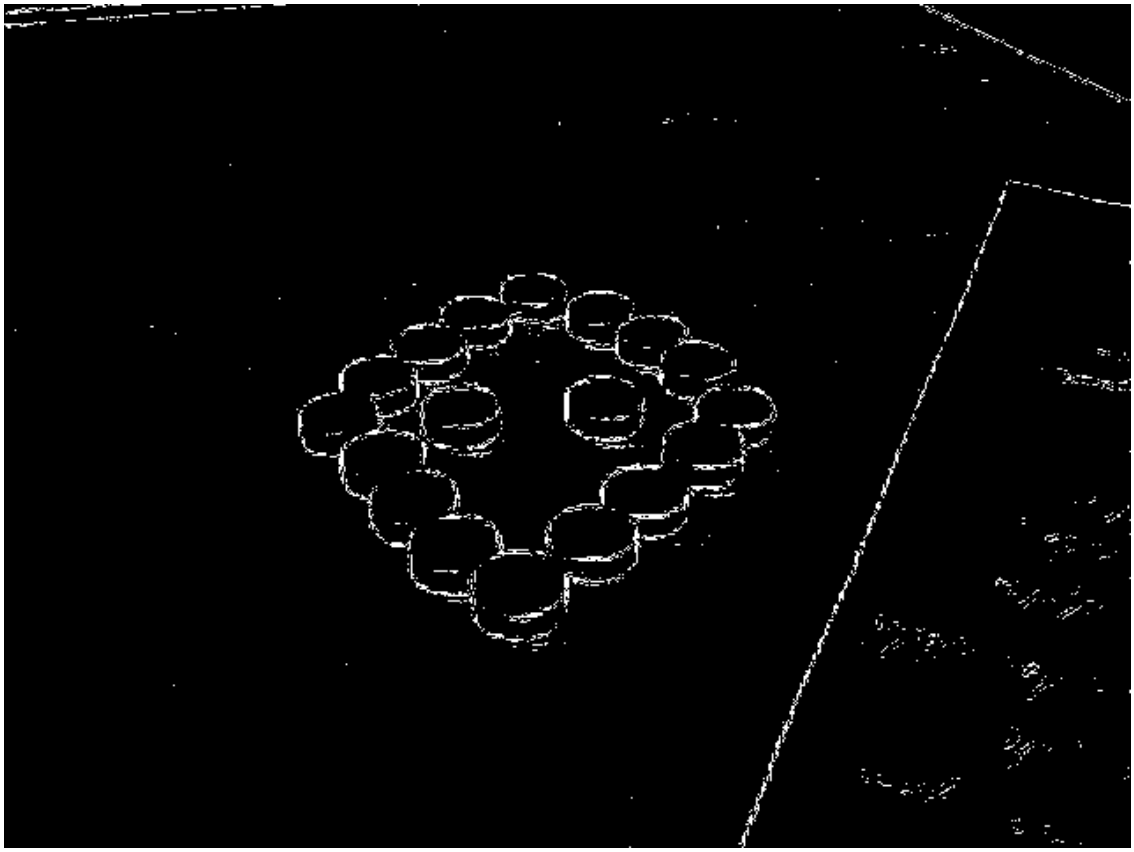
$$f - \alpha\triangle f \rightarrow \begin{pmatrix} 0 & -\alpha & 0 \\ -\alpha & 1+4\alpha & -\alpha \\ 0 & -\alpha & 0 \end{pmatrix} \tag{6-2}$$

Maska filtru: $\begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}$

6.2.2 Robertsův operátor

Tento operátor detekuje hrany tam, kde jsou maximální změny hodnot jasu. Je velmi podobný operátorům Prewittové,které jsou uvedeny dále, liší se opět pouze jinými maskami filtru. V případě mého programu ovšem dosáhl naprosto nejlepšího výsledku.

roberts filter

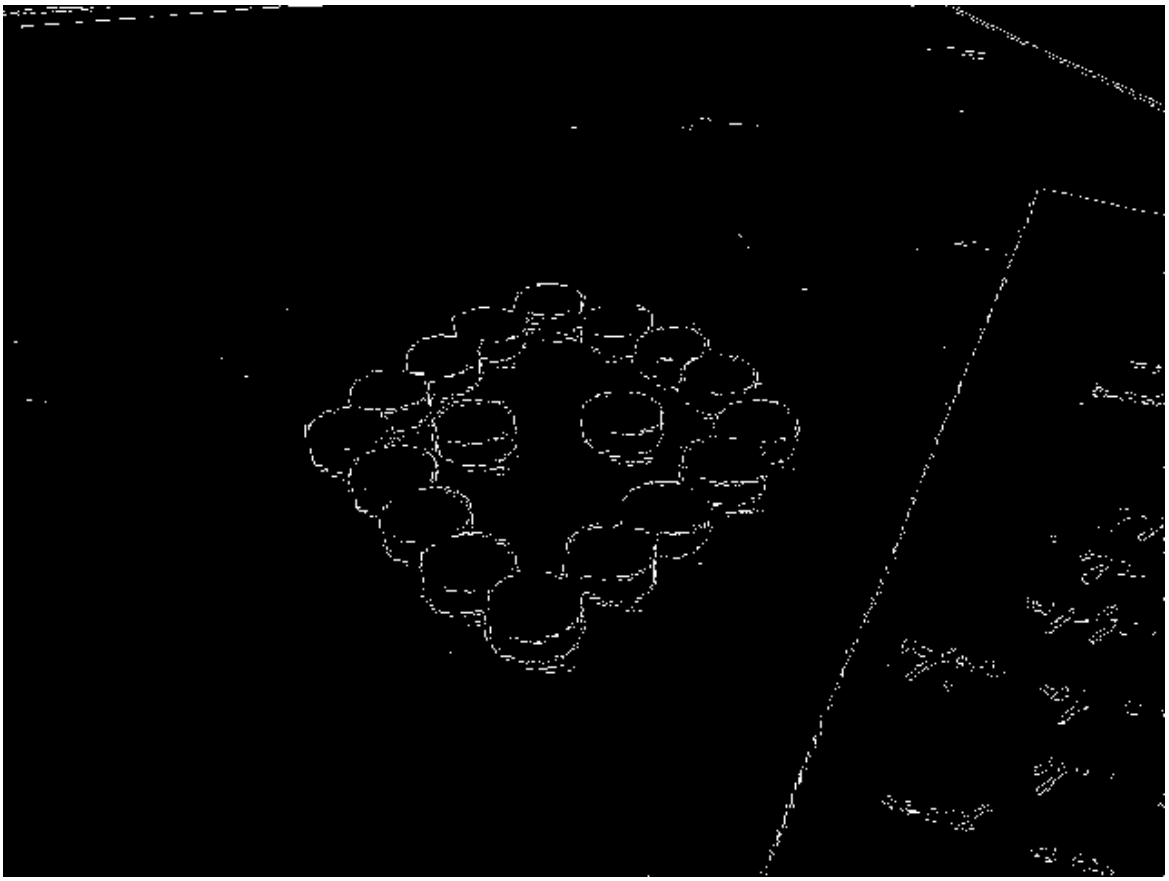


Obr. 6-2 Robertsův hranový detektor

6.2.3 Operátor Prewittové

Operátor Prewittové je anizotropní operátor. To znamená, že detekuje hrany pouze v horizontálním, nebo ve vertikálním směru. V ukázce jsem ovšem použil funkci implementovanou do programu matlab, kde tento operátor detekuje oba druhy hran, pokud nenastavíme jinak. Tato metoda zvýrazňuje oblasti tam, kde je změna jasových hodnot nejvyšší. Jak vidíme z obrázku, není tato metoda příliš citlivá na šum a také celkem dobře detekuje hrany.

prewitt filter



Obr. 6-3 Hranový detektor Prewittové

Masky filtru:

$$\begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix}$$

$$\begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix}$$

6.2.4 Sobelův operátor

Je velmi podobný operátoru Prewittové. Pro detekci hran používá stejného principu, pouze tvar masky filtru je trochu odlišný. Jejich podobnost je jasně vidět i z obrázku.

sobel filter

Obr. 6-4 Sobelův detektor hran

Masky filtru:

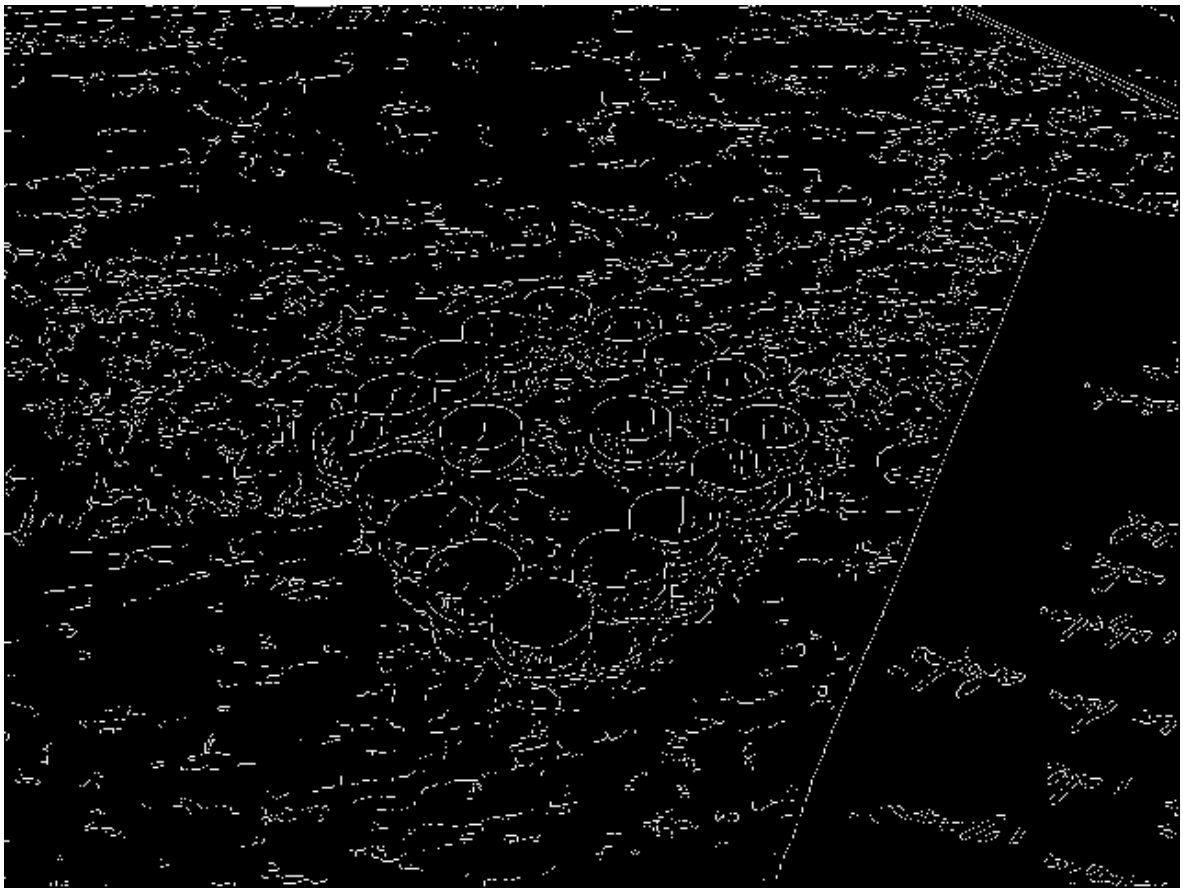
$$\begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix}$$

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

6.2.5 Cannyho operátor

Tento filtr určuje hrany pomocí hledání maxim v gradientu vstupního obrazu. Gradient se počítá pomocí Gaussian filtru. Cannyho metoda používá pro určení hran dva prahy, jeden pro určení silných hran a druhý pro určení hran slabých, které jsou do výsledku zahrnuty pouze v případě, že jsou v kontaktu se silnými hranami. Jak vidíme z obrázku, tato metoda byla v tomto případě příliš citlivá a zobrazuje kromě hran v obraze i malé změny v textuře okolí.

canny filter



Obr. 6-5 Cannyho detektor hran

Informace čerpány z [2] a [3]

7 Popis řešení v programu MATLAB

V hlavním programu jsem měl za úkol vytvořit detektor, který by identifikoval určité objekty dané barvy. Vybral jsem si červené kruhy. Na začátku programu se nastaví vstupní zařízení, takže při změně hardwaru se musí tento řádek programu přepsat. Poté je nastaven interval zachytávání snímků, protože pokud bychom nesnížili FPS, program by byl příliš náročný. Začne zachytávání obrazu, které skončí po 100 snímcích. Nejprve je na obraz použita mnou vytvořená funkce pro detekci červené barvy. Tato funkce je kvůli problémům, se kterými jsem se potýkal při programování v MATLABu, celkem jednoduchá, ale pro náš demonstrativní účel postačí. Její funkce spočívá v tom, že pro každý pixel obrazové matice najde tři hodnoty základních barev Red, Green a Blue. Poté určí pixely, které mají danou minimální hodnotu proměnné Red a zároveň o hodně menší hodnoty proměnných Green a Blue. Pro ideální funkci programu by se musel definovat určitý minimální poměr mezi těmito barvami, aby byla červená barva zastoupena v pixelu maximálně a ostatní barvy v poměru k ní mnohem méně.

Zde je tabulka, která udává poměry jednotlivých hodnot RGB pro určité ideální barvy. Pro reálné použití musíme počítat s určitými tolerancemi.

Red	Green	Blue	Color
0	0	0	Black
1	1	1	White
1	0	0	Red
0	1	0	Green
0	0	1	Blue
1	1	0	Yellow
1	0	1	Magenta
0	1	1	Cyan
0.5	0.5	0.5	Gray
0.5	0	0	Dark red
1	0.62	0.40	Copper
0.49	1	0.83	Aquamarine

Tabulka 1 Poměry hodnot RGB pro barvy

Pixely, pro které toto kritérium platí, jsou označeny logickou hodnotou jedna a ostatní pixely hodnotou nula. Tím vznikne obrazová matice s hodnotami buďto bílé, nebo černé barvy. Nemusíme tak počítat s 3x256 možnými hodnotami pro každý pixel, ale pouze se dvěma možnými hodnotami. To výrazně urychlí další zpracování.

Dále je na obraz uplatněn mediánový filtr s maskou 3x3 a také filtr odstraňující všechny objekty menší než 60 pixelů. Pro praktické použití nejsou potřeba oba dva tyto filtry zároveň, ale pro demonstrativní účely jsou v programu ponechány oba. Rozměr masky mediánového filtru byl volen po několika testech tak, aby byl zachován ideální poměr mezi rychlostí a mírou vyhlazení šumu. Potom je na obraz použita speciální funkce MATLABu, která má dva výstupy. Jeden z nich je obrazová matice s jedničkami tam, kde jsou červené objekty a s nulami tam, kde je barva jiná. A druhý výstup je informace o počtu detekovaných objektů a se souřadnicemi jejich hran. Tahle část programu je již převzatá z nápovědy pro MATLAB, kde je tento problém ukázkově řešen. Nakonec se pro každý detekovaný objekt vypočítá koeficient, který udává poměr mezi změřeným obvodem objektu a předpokládaným obvodem kruhu při obsahu, který objekt má. Pokud se tento koeficient blíží jedničce, je objekt označen jako kruh a do jeho středu je vykreslena malá kružnice.

Bohužel program trpí dvěma závadami. První je problém s výkonem, aby program fungoval, musel jsem velmi omezit rozlišení obrazu a také rychlost jeho zachytávání. Druhý problém je ten, že webkamera mění jasové hodnoty celého obrazu v závislosti na tom, co detekuje uprostřed obrazu a také na okolním osvětlení. Toto se v programu nedá kontrolovat a zavádí to další chyby v detekci.

Základní funkci, které jsme chtěli docílit, ovšem program splňuje a najdeme v něm proměnnou, která nese souřadnice detekovaného kruhu, nebo i více kruhů pro teoretické budoucí použití při ovládání například kurzoru počítače.

Informace čerpány z [3]

8 Linux

8.1 Způsob tvorby programu

Všechno, co zde bylo dosud napsáno a demonstrováno, se týkalo implementace v operačním systému Windows pomocí programu MATLAB. Bylo to zadání mého semestrálního projektu a nyní poslouží jako první polovina mé bakalářské práce. Výsledek tohoto projektu byl program v MATLABu, který byl schopen načítat obrazová data z předem definované kamery nebo jiného zdroje a poté v reálném čase vyhledat a lokalizovat polohu červeného kruhu v obraze.

V druhé polovině bakalářské práce jsem měl za úkol vyrobit program se stejnou funkcí v operačním systému Linux. Tento program měl navíc použít souřadnice nalezeného červeného kruhu k ovládání kurzoru myši. Jako nejvhodnější se ukázalo použít jazyk C++ a knihovnu OpenCV, která je pro podobné účely speciálně navržená. Bohužel, v tomto bodě se vyskytly mírné komplikace, neboť k této problematice již není k dispozici obrovská a přehledná nápověda nebo dokumentace, jako u případu s MATLABem. Následkem toho byl postup při tvorbě programu poněkud zdlouhavý a odkázán hlavně na střípky informací z internetových fór a metodu pokus-omyl.

Pro vytvoření základu mého programu jsem nakonec použil aplikaci Harpia. Tato volně šiřitelná Linuxová aplikace se používá při tvorbě programů založených na funkcích z knihovny OpenCV. Má grafické uživatelské rozhraní a poměrně velké množství předdefinovaných funkcí. Ty jsou reprezentovány barevnými bloky se vstupy a výstupy, které už pak jen podle potřeby propojíme. Každý takový blok reprezentuje kus zdrojového kódu v jazyce C++, který je optimalizovaný pro použití na běžných stolních počítačích a to v reálném čase. Bohužel tento program neumí všechno a tak další krok byl export zdrojového kódu vytvořeného Harpií do obyčejného textového souboru, aby mohl být poté upraven v jakémkoli textovém editoru a zkompilován pomocí terminálu, čili příkazové řádky. Získáme tím totiž plnohodnotný zdrojový kód psaný v C++, který můžeme klidně zkopírovat do našeho textového IDE (Integrated Development Environment) a dále s ním pracovat.

Jako IDE při editaci kódu jsem používal aplikaci Geany, ale v konečné fázi se vyskytly malé potíže při používání více externích knihoven současně, tak jsem přešel na variantu editace zdrojového kódu v obyčejném textovém editoru a následné kompilace pomocí příkazové řádky.

Informace čerpány z [4] a [5]

8.2 Linux versus Windows

Jak bylo již uvedeno výše, zabýval jsem se řešením programu v operačním systému Linux. Konkrétně Ubuntu verze 9. Používal jsem programovací jazyk C++ a kromě dalších knihovnu OpenCV. Hlavní výhoda je to, že C++ i OpenCV jsou tzv. cross-platform, takže jsou univerzální a používají se ve všech operačních systémech stejně. To znamená, že můj program je sice psaný v Linuxu, ale verze pro Windows by vypadala identicky, nebo téměř identicky (kvůli funkcím mimo OpenCV). Mohl jsem tak při psaní programu čerpat informace z internetových zdrojů, které nebyly určeny pro Linux. Takových je totiž stále drtivá většina.

Jeden veliký rozdíl mezi těmito operačními systémy ovšem existuje. A to je softwarová podpora výrobců počítačových komponent. K naprosté většině hardwaru, který jsem používal, totiž neexistuje oficiální ovladač pro systém Linux. V případě USB webkamery Logitech, kterou jsem také používal, nebyly žádné problémy. Kamera totiž podporuje UVC, což je zkratka pro *USB Device Class Definition for Video Devices*, neboli *USB Video Class*. Součástí tohoto ovladače je i V4L2 kernel device driver, takže systém Linux si s webkamerou poradil bez nejmenšího problému i bez softwaru od výrobce. Na druhé straně stojí ovšem podpora grafických karet. Na stolním PC byla stará grafika od společnosti ATI a i přes veliké úsilí jsem nebyl schopen najít a nainstalovat vhodný ovladač pro tuto grafickou kartu. Podle zkušeností ostatních uživatelů na internetu to ovšem není způsobenu stářím grafické karty, ale obecně přístupem ATI k operačním systémům Linux a hrubým nedostatkem použitelných ovladačů. Nejsem si jistý, zda to bylo způsobeno právě tímto problémem, nebo jen slabým výkonem mého stolního počítače, ale při tvorbě složitějších aplikací na tomto počítači jsem se setkal s neúspěchem. Proto jsem při tvorbě svého programu přešel na laptop, který má integrovanou webkameru od firmy Hewlett-Packard, se kterou také nebyly nejmenší potíže, a hlavně grafickou kartu nVidia od společnosti Intel. K těmto grafikám je přímo na oficiálních internetových stránkách plná softwarová podpora i pro operační systém Linux. To je poměrně důležitá informace pro ty, kteří pořizují nový hardware a chtějí pracovat i pod Linuxem. Každopádně na laptopu, který měl lepší grafiku i celkově lepší parametry než stolní PC již žádné potíže se složitějšími a náročnějšími aplikacemi nebyly.

Čerpáno z [6]

8.3 Zvláštní knihovny

Ve svém programu jsem zpracovával obrazová data a také jsem potřeboval získat přístup k ovládání kurzoru myši. Za tímto účelem jsem použil dvě speciální knihovny, které se v obyčejných C++ programech běžně nevyskytují a musí se většinou dodatečně stáhnout a nainstalovat. Jsou to knihovny OpenCV a Xlib.

8.3.1 OpenCV

OpenCV, neboli Open Computer Vision je knihovna speciálně zaměřená na zpracování obrazu v reálném čase a manipulaci s obrazem. Je to svobodný software, takže jej lze bezplatně stáhnout, nainstalovat, používat a upravovat. Jak již bylo dříve uvedeno, OpenCV je multiplatformní, stejná pro Windows, Linux i ostatní operační systémy. Zpočátku byla vyvíjena společností Intel a pro zrychlení je možné použít Intel IPP (Integrated Performance Primitives). Knihovna OpenCV je stále ve vývoji, takže vznikají docela často nové verze, které implementují nové funkce, ale zároveň některé funkce již nenabízejí. Při tvorbě programu jsem používal verzi 2.0. Níže jsou uvedeny nejdůležitější funkce knihovny OpenCV použité v mém programu, které už nebudou popsány v dalších kapitolách.

CvCapture* **cvCaptureFromCAM**(int *index*)

Zahájí snímání videa z obrazového zařízení. Parametr index určuje zařízení, ze kterého se mají obrazové informace odebírat. V Linuxu to mohou být dvě zařízení typu V4L2 nebo FireWire. Výhoda je ta, že pokud máme k počítači připojeno pouze jedno takovéto zařízení, a to je většina případů, nemusíme parametr vůbec definovat a zařízení se vyhledá automaticky. Funkce cvCaptureFromCAM alokuje pro obrazová data strukturu cvCapture, která se nakonec opět dealokuje funkcí ReleaseCapture.

int **cvGrabFrame**(CvCapture* *capture*)

Tato funkce získá snímek z obrazového zařízení. Snímek není zobrazován a je uložen vnitřně.

IplImage* **cvRetrieveFrame**(CvCapture* *capture*)

Získá obrázek ze zdroje předem zachyceného pomocí předchozí funkce cvGrabFrame.

IplImage* **cvCloneImage**(const IplImage* *image*)¶

Vytvoří naprosto identickou kopii původního obrazu. Parametr *image* udává zdroj kopie a funkce vrací ukazatel na vytvořenou kopii obrazu. Program Harpia používá tuto funkci při napojování funkčních bloků. Při práci s obrazem neprovádí změny na výstupu předchozího bloku, nýbrž vytvoří pro následující blok kopii a s tou potom pracuje. Výstup předchozího bloku tak zůstane zachován a může být opět použit v další části programu. V případě mé aplikace je tento postup ovšem poněkud zbytečný, pokud ovšem nepředpokládáme žádné další úpravy kódu. Při testování jsem zjistil, že tento způsob neovlivňuje rychlost zpracování dat a nezatěžuje počítač více, než varianta bez kopírování, proto jsem tuhle strategii neměnil a ponechal tyto části kódu tak, jak je vytvořila aplikace Harpia.

int **cvNamedWindow**(const char* *name*, int *flags*)

Jednoduchá funkce pro vytvoření nového okna na obrazovce. Toto okno poté slouží jako výstup pro obrazová i ostatní data. Můžeme zde umístit i vstupní ovládací prvky. Parametr *name* udává název okna a druhý parametr *flags* bude udávat vlastnosti tohoto okna. Záměrně uvádím bude, neboť v současné verzi 2.0 je zatím použitelný pouze parametr *CV_WINDOW_AUTOSIZE*, který automaticky přizpůsobí velikost okna v něm zobrazenému obrázku.

IplImage* **cvCreateImage**(CvSize *size*, int *depth*, int *channels*)¶

Tato funkce vytvoří obrázek. Konkrétně jeho hlavičku a alokuje obrazová data. Parametr *size* udává šířku a výšku obrázku, parametr *depth* bitovou hloubku pixelu.

Zde je seznam podporovaných bitových hloubek pro OpenCV:

- IPL_DEPTH_8U - Unsigned 8-bit integer
- IPL_DEPTH_8S - Signed 8-bit integer
- IPL_DEPTH_16U - Unsigned 16-bit integer
- IPL_DEPTH_16S - Signed 16-bit integer
- IPL_DEPTH_32S - Signed 32-bit integer
- IPL_DEPTH_32F - Single-precision floating point
- IPL_DEPTH_64F - Double-precision floating point

Poslední parametr *channels* udává počet kanálů na obrazový bod. Pro barevný obraz ve formátu RGB použijeme 3 kanály, každý pro jednu barevnou složku.

void **cvPutText**(CvArr* *img*, const char* *text*, CvPoint *org*, const CvFont* *font*, CvScalar *color*)

Funkce PutText vykreslí řadu znaků. V podstatě tyto znaky vloží do nějakého zobrazovaného obrázku, který je definován parametrem *img*. Parametr *text* udává požadovaný text, který chceme vložit a parametr *org* představuje souřadnice levého spodního rohu prvního znaku řetězce. Další parametr představuje font písma, jež chceme použít. Je to ukazatel na krátkou strukturu typu CvFont, která obsahuje informace o vlastnostech textu a musíme si ji předem vytvořit. To realizujeme nejprve konstruktorem CvFont a poté funkcí cvInitFont, která do struktury doplní informace o fontu jako třeba velikost písma, typ písma, řádkování, prokládání, sklon, tloušťku a další. Poslední parametr udává barvu textu.

void **cvCircle**(CvArr* *img*, CvPoint *center*, int *radius*, CvScalar *color*, int *thickness=1*, int *lineType=8*, int *shift=0*)

Funkce pro vykreslování kružnice. První operátor označuje obraz do kterého se má kržnice vykrestit, druhý souřadnice její střed a třetí poloměr. Operátor *color* udává barvu a *thickness* tloušťku kružnice. Může vykreslovat jak kružnice, tak i plné kruhy.

void **cvLine**(CvArr* *img*, CvPoint *pt1*, CvPoint *pt2*, CvScalar *color*, int *thickness=1*, int line*Type=8*, int *shift=0*)

Funkce pro kreslení úseček. Kromě operátorů shodných s cvCircle, jsou zde body *pt1* a *pt2*. Udávají souřadnice výchozího a konečného bodu přímky.

Knihovnu OpenCV do programu vložíme v hlavičce kódu pomocí

```
#include <cv.h>
#include <cvaux.h>
#include <highgui.h>.
```

Při překladu poté použijeme příkaz

```
gcc program1.c `pkg-config --libs --cflags opencv` -o program2,
```

kde *program1* je název souboru se zdrojovým kódem a *program2* název souboru, který vytvoří kompilátor.

Informace čerpány z [5]

8.3.2 Xlib

Xlib je knihovna, kterou využívá systém X Window. Tuto knihovnu jsem v programu použil pouze kvůli jedné funkci, a to XWarpPointer. Pokud máme knihovnu Xlib správně nainstalovanou, do hlavičky ji vložíme příkazem

```
#include "/usr/include/X11/Xlib.h".
```

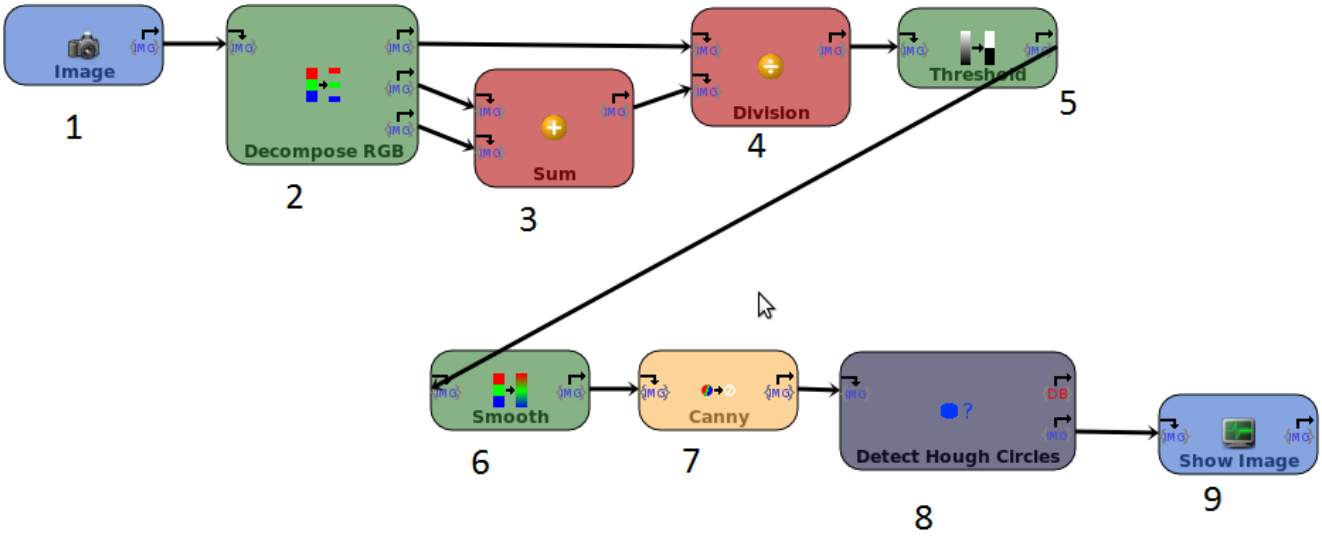
Jelikož jsem ve své práci použil pouze jednu funkci z této knihovny, příliš jsem se jí nezabýval. Ovšem ukázalo se, že tato knihovna nabízí obrovskou škálu možností včetně práce s okny a kreslení objektů či vypisování textů. V systému X Window, zjednodušeně vysvětleno, existuje vždy klient a server. Server ovládá grafické rozhraní a klient posílá serveru zrávy o tom, co má udělat. Může existovat více klientů, serverů, nebo můžeme mít například klient i server na různých počítačích. Knihovnu Xlib použijeme tehdy, pokud chceme ovládat X Server pomocí nějaké svojí aplikace. Například pokud tvoříme program, který má samostatně pracovat s okny na ploše či s kurzorem myši.

Zdroj informací [7]

9 Popis řešení v Linuxu

Hlavní úkol mého programu je nalézt v obraze kruh červené barvy a podle jeho pozice měnit polohu kurzoru myši v Linuxu. Kvůli povaze funkce, která ovládá kurzor myši je změna polohy kurzoru prováděna relativně, nikoliv absolutně. To znamená, že pokud se změní poloha detekovaného kruhu doleva, změní se i pozice kurzoru myši doleva oproti jeho předchozí poloze. Celá detekce přitom probíhá pro každý snímek znovu v cyklu while. Přerušení nastane při stisku klávesy. Použitá je funkce cvWaitKey, protože v jejím parametru můžeme definovat dobu čekání a tím i framerate programu.

Základ detekční části programu byl vytvořen v aplikaci Harpia, která pracuje v uživatelském rozhraní s grafickými bloky. Grafické schéma tedy vypadá následovně.



Obr. 9-1 Schéma detekčního programu

9.1 Vstupní funkce

V této části programu je definován zdroj a formát dat pro program. Definujeme zde vstupní zařízení a formát vstupu. OpenCV zvládá vše plně automaticky, takže definice je zde značně jednoduchá.

Pokud máme k počítači připojenou pouze jednu kameru, program si ji detekuje sám. Zdrojový kód této části vypadá přibližně takto:

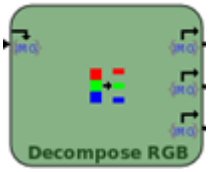
```
IplImage * block1_img_o1 = NULL;
CvCapture * block1_capture = NULL;
IplImage * block1_frame = NULL;
block1_capture = cvCaptureFromCAM(0); // Parametr pro volbu
                                      kamery, pokud je jen jedna, staci (0)
cvGrabFrame (block1_capture);
block1_frame = cvRetrieveFrame (block1_capture);
```

Proměnná *block1_frame* je již běžný obrázek, se kterým dále pracujeme.

Nejlépe popíšeme činnost jednotlivých bloků na příkladu jednoho obrázku, na kterém jsou vidět různé předměty různých barev. Snažíme se nalézt červené kolečko v pravém spodním rohu obrázku a přitom se vyhnout chybné detekci ostatních barev. Nejvíce bude dělat problémy bílá v podobě odrazu světla od desky stolu.

Obr. 9-2 Vstupní data pro detektor.

9.2 Rozdělení barevných kanálů



Další blok slouží k rozdělení signálu RGB na jednotlivé barevné složky. V knihovně OpenCV je na to funkce cvSplit:

```
cvSplit(obrazek ,img_t3 ,img_t2 ,img_t1 , NULL);.
```

Pomocí tohoto rozdělíme původní obrázek na složky img_t3, img_t2 a img_t1, což jsou jednotlivé barvy systému RGB.

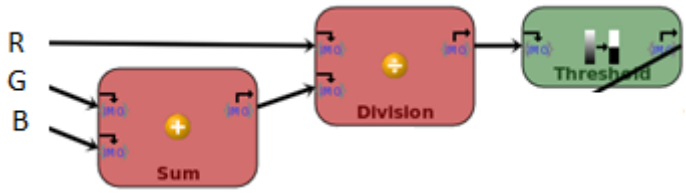
Obr. 9-3 Červená složka obrazu

Obr. 9-4 Zelená složka obrazu

Obr. 9-5 Modrá složka obrazu

Na předchozích obrázcích vidíme jednotlivé barevné složky původního obrázku (9-2). Jsou to tři odlišné černobílé obrazy, kde každý pixel má pouze jeden kanál, a to bílý. Intenzita tohoto bílého kanálu udává zastoupení té dané barvy v původním trojkanálovém RGB obrazu. Mohlo by se zdát, že pro hledání červených kruhů stačí, když budeme pracovat pouze s červenou složkou signálu (obr 9-3) a ostatní budeme ignorovat. Ovšem to by nebylo správné. Jak je vidět na obrázcích, námi hledaný červený kruh je opravdu nejsvětlejší na červené složce obrazu, ale podobně vysokou intenzitu tam mají také ostatní objekty, jako třeba tenisový míček či bílý odraz žárovky. Tyto bílé a světlé objekty mají totiž velmi vysoké zastoupení ve všech třech barevných složkách. Jak víme, nejsvětlejší bílá barva by měla R:255, G:255 a B:255, takže bychom na samotné červené složce signálu nepoznali, jestli jde o červenou barvu nebo nějakou hodně světlou či bílou. Proto je v programu implementován další blok, který vyhodnocuje červenou jen na základě poměru mezi jednotlivými barevnými složkami.

9.3 Filtrace červené barvy



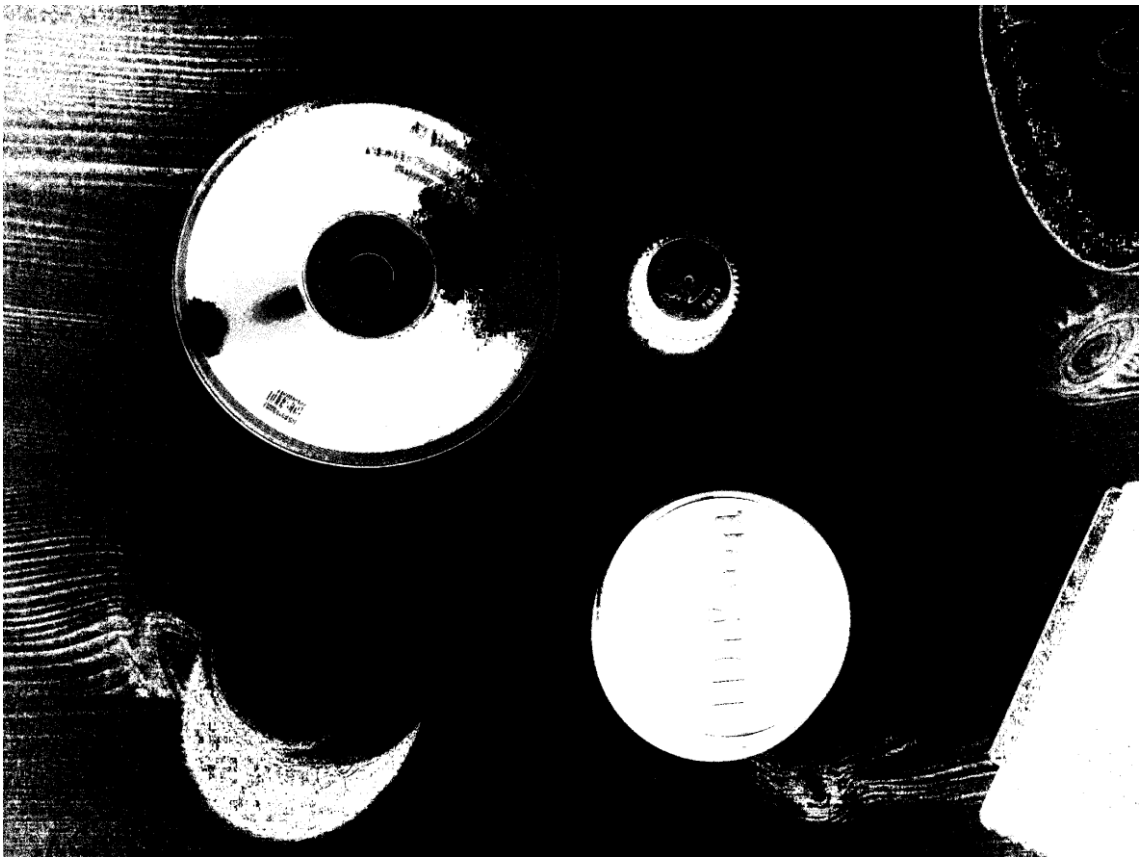
Tato část programu řeší rozlišení mezi červenou barvou a nějakou jinou barvou, která má také velké zastoupení červené složky. V kapitole 7 můžeme vidět tabulku, která udává poměrné zastoupení jednotlivých složek RGB v některých barvách. Dozvíme se tam, že červená má poměr RGB 1:0:0. Tohle je ovšem teoretické ideální číslo, se kterým se v praxi nepotkáme. Navíc by bylo obtížné shánět předmět se stejným poměrem barev, který jsme si nadefinovali. Proto jsem v programu zvolil určitou toleranci, takže program detekuje pouze červené a jí podobné barvy. Samotná filtrace probíhá na základě zjišťování poměru jednotlivých barevných složek pro každý pixel obrazu zvlášť. Pomocí prvního bloku (sum) se provede součet intenzity zelené a modré složky pixelu. Druhý blok (division) zjistí poměr červené složky pixelu a součtu modré se zelenou složkou. Provádí tedy dělení R/(G+B).

Poslední blok (threshold) nakonec rozhodne, zdali je tento poměr větší nebo menší než jedna. V případě, že je větší než jedna, označí pixel za červený a přidělí mu hodnotu jedna (bílou barvu). Jestliže je ovšem poměr menší než jedna, je pixel označen jako logická nula (černá barva). Zdrojový kód je zde již poměrně dlouhý, takže uvedu funkci pouze pro blok threshold.

```
cvThreshold(img_vstup,img_vystup,block20_arg_threshold,block20_arg_
maxValue,block20_arg_thresholdType);}
```

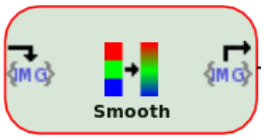
Zadáváme zde hlavně maximální a hraniční hodnotu, číslo jedna jsem zvolil na základě několika experimentů.

Po takovéto detekci červené barvy vypadá náš původní obrázek následovně:



Obr. 9-6 Obraz po detekci červené barvy

9.4 Blok vyhlazování

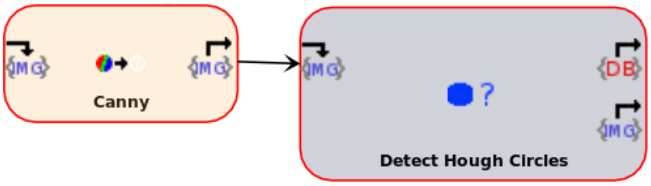


Tento blok představuje pouze jediný řádek kódu a to funkci cvSmooth.

```
cvSmooth(block33_img_i1, block33_img_o1 ,CV_MEDIAN,7,9,0,0)
```

Jedná se o mediánový vyhlazovací filtr s maskou o velikosti 7x7. Zde je vidět obrovské zrychlení běhu programu oproti aplikaci v programu MATLAB. Tam jsme si mohli dovolit filtr s maskou pouze 3x3, při větších rozměrech masek již docházelo k viditelnému zpomalení, jež si při zpracování obrazu v reálném čase rozhodně nemůžeme dovolit.

9.5 Bloky detekce hran a kruhů



Tyto dva bloky záměrně uvádím společně, neboť blok *Detect Hough Circles* by teoreticky mohl zastat potřebnou funkci sám. Blok *Canny*, který představuje Cannyho hranový detektor zde není nezbytně nutný a pokud bychom chtěli program co nejvíce zjednodušit, mohli bychom vypustit i dřívější vyhlazování pomocí mediánu, neboť podobnou funkci poslední blok obsahuje taktéž. Nicméně experimenty potvrdily, že bez vypuštění těchto bloků nedojde k viditelnému zpomalení programu a celkové výsledky při detekci jsou s nimi lepší. Takto definujeme funkci Cannyho detektoru, první dva parametry jsou vstupní a výstupní obraz, další dva jsou spodní a horní hranice pro detekci, poslední je velikost Sobelova operátoru.

```
cvCanny(tmpImg28, tmpImg28, block28_arg_threshold1,
block28_arg_threshold2, block28_arg_aperture_size);
```

Pro detekci kruhů slouží v OpenCV funkce CvHoughCircles:

```
cvHoughCircles( block41_img_t1, block41_storage, CV_HOUGH_GRADIENT,
3.0, 600.0, 100.0, 70.0,0,1000);
```

První dva operátory jsou vstup a výstup funkce, třetí je typ metody. Zatím žádná jiná, než *CV_HOUHG_GRADIENT* není v knihovně implementována. Další parametr je rozlišení při detekci středu kruhu, trojka znamená třikrát menší rozlišení, než má vstupní obraz. Další parametr je minimální vzdálenost mezi středy detekovaných kruhů. Je nastavená vysoko, protože chceme v našem programu detekovat pouze jeden kruh. Další dva parametry jsou proměnné pro vnitřní hranový detektor a vyhlazovací filtr, poslední dva jsou minimální a maximální poloměr detekovaného kruhu. Při další editaci zdrojového kódu jsem omezil velikost tak, že program sice zobrazuje, ale už nedetekuje kruhy menší než 40 pixelů. Je to potřebné pro omezení nežádoucích chybných detekcí. Detekce kruhů probíhá stejně, jak bylo popsáno v předchozí části s MATLABem. Samozřejmě bylo nutno doplnit také koeficient, který udává přijatelnou odchylku od dokonalého kruhu.

9.6 Ovládání kurzoru myši

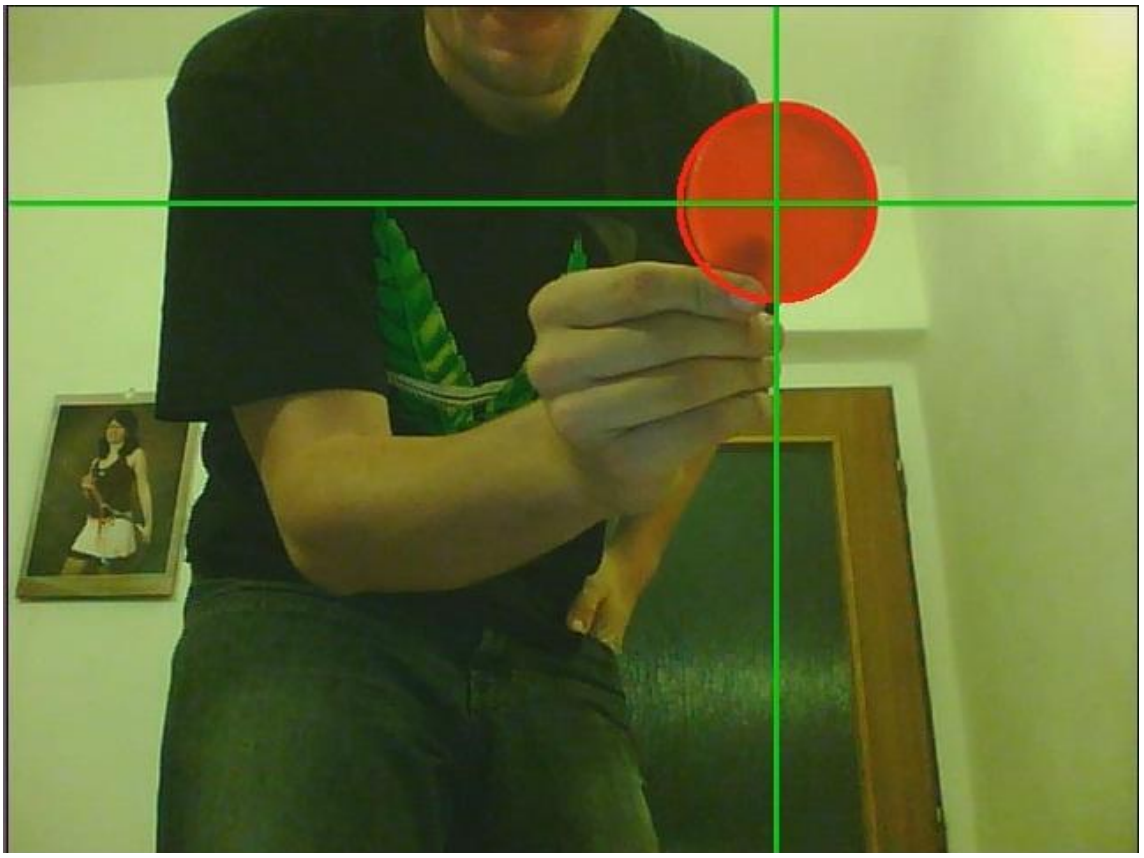
Kvůli tomuto úkolu bylo potřeba importovat knihovnu Xlib a použít její funkci XWarpPointer. Celé posunutí myši se dá vyjádřit ve zdrojovém kódu pomocí tří řádků.

```
Display *displayMain = XOpenDisplay(NULL);
XWarpPointer(displayMain, None, None, 0, 0, 0, 0, hor , ver);
XCloseDisplay(displayMain);
```

První a třetí řádek zprostředkují přístup k serveru X Window a jeho následné uvolnění po provedení změny. Prostřední řádek je funkce s vlastní změnou polohy kurzoru myši. Všimněme si pouze posledních dvou parametrů *hor* a *ver*. Udávají vzdálenost, o kterou se má poloha kurzoru změnit v horizontálním a vertikálním směru. Tato funkce zprostředkuje relativní změnu polohy, to znamená, že změna je vždy oproti předchozí poloze kurzoru.
Když potřebujeme pohyb doprava dolů, zadáme kladné hodnoty, při pohybu opačným směrem hodnoty záporné. Relativní změna polohy je oproti absolutní výhodnější, neboť můžeme nastavovat rychlost pohybu myši a také nedochází k prudkému pohybu kurzoru při chybné detekci kruhu daleko mimo jeho skutečnou polohu. V této části programu jsem také implementoval nastavení pro citlivost myši, takže můžeme rychlost jejího pohybu ovládat jednou proměnnou. Dále bylo příhodné deklarovat další

proměnnou a tou byla citlivost kamery. Představuje maximální možnou změnu polohy detekovaného kruhu za jeden cyklus programu, při kterém byl kruh skutečně detekován. Umožňuje tak programu rozlišit, jestli se jedná o plynulý pohyb kruhu po nějaké trajektorii, nebo došlo ke skokové změně polohy detekovaného kruhu. Tento skok může nastat například při chybné detekci v důsledku nekvalitního vstupního signálu, nebo jen při zakrytí kolečka. Můžeme tak zakrytím kolečka nebo jen jeho natočením realizovat stejnou věc, jako je zvednutí klasické počítačové myši ze stolu a její posunutí ve vzduchu.

V tomto bodě již máme program hotov. Použijeme funkci cvShowImage, která má pouze dva parametry. Obraz, který chceme zobrazit a okno, do kterého jej chceme umístit. Ve výsledku vypadá výstup jako na obrázku (9-7). Detekovaný kruh je označen červeným obrysem, a pokud splní podmínku minimálního poloměru, je na jeho střed zaměřen souřadnicový kříž a informace použity pro posun kurzoru myši.



Obr. 9-7 Okno s výstupem programu a detekovaným červeným kruhem.

Informace čerpány z [4], [5], [6], [7] a [8]

10 Závěr

V první části mé bakalářské práce jsem se zabýval problematikou detekování objektů v oblasti počítačového vidění. Úspěšně se mi podařilo vytvořit mnoho simulací v programu MATLAB, které demonstrují různé funkce používané při zpracování obrazu za účelem rozpoznávání objektů. Program MATLAB je velmi komplexní, takže způsobů, kterými se daly jednotlivé funkce realizovat, bylo několik. Ve většině případů má ovšem program tyto funkce již vytvořené a nám stačí je pouze použít. Největší zdroj informací byla jistě knihovna nápovědy u tohoto programu. Skoro všechny funkce, které jsem použil, jsou zde velmi dobře popsány a ukázány v praktickém použití. V případě detekce kruhových objektů dokonce existuje v této nápovědě celý kus zdrojového kódu, který řešil přesně to, co jsem potřeboval.

Bohužel jsem se potýkal s velkými problémy s výkonem mého počítače. Musel jsem zmenšovat rozlišení vstupních obrazů, kvalitu používaných filtrů a zjednodušovat některé operace, aby vůbec program fungoval. To mělo za následek mnoho různých nepřesností a chyb v detekci. V hlavním programu je již výsledná matice tak malá, že mezi kruhem a obdélníkem není až takový rozdíl, jaký bychom potřebovali.

Poté jsem se zaměřil na vytvoření programu splňujícího zadání bakalářské práce v operačním systému Linux, což se mi podařilo. Použil jsem programovací jazyk C++ a převážná část použitých funkcí byla z knihovny OpenCV. Vytvořený program funguje velmi podobně jako ten MATLABový, navíc umí ovládat kurzor myši a mnohem méně zatěžuje při práci počítač. Zvolil jsem opět detekci červených kruhů, i když po mírné úpravě kódu není problém zadání změnit na jinou barvu či jiný tvar.

Pro použití programu je nutné ho nejdříve zkompilovat. Nejjednodušeji pomocí terminálu příkazem, který je uveden jako komentář hned na začátku zdrojového kódu. Jako výhodu to přináší použitelnost jakékoli kompatibilní kamery bez nutnosti přepisování kódu. Po spuštění programu se objeví výstup webkamery doplněný o obrysy detekovaného kruhu a souřadnicového kříže a druhé okno s údaji o poloze kruhu a jeho poloměru. Zde se objevil problém, který se mi nepodařilo vyřešit. A sice to, že obraz se promítá invertovaně. Správný pohyb kurzoru to ovšem neovlivní, neboť ten jsem poté invertoval stejným způsobem.

Zadání mé bakalářské práce bylo velice zajímavé a přiblížila mi možnosti operačního systému Linux. Ovšem ještě více mě zaujala problematika počítačového vidění obecně. Nakonec třeba taková knihovna OpenCV je multiplatformní, takže na použitém operačním systému až tak nezáleží. Zabýval jsem se například otázkou, zdali by se dalo do mého programu implementovat nejenom pohyb myši, ale i ovládání nahrazující její tlačítka. První z nich byla detekce dvou objektů různých barev a ovládání tlačítek na základě jejich vzájemné polohy. Další možností bylo rozpoznávání různých gest, které by uživatel snímaným objektem prováděl, jako například přiblížení ke kameře, či změna tvaru objektu. Od obou těchto nápadů jsem ale brzy upustil, neboť kliknutí vyžaduje přesnost, se kterou bych těmito způsoby a dostupným vybavením zatím nemohl dosáhnout. Ani jedna z možností nebylo přijatelné elegantní řešení. Jedno z takových elegantních řešení teď velice nedávno představila společnost SONY pro svůj playstation. Používají webkameru pro detekci různobarevných světelných bodů umístěných na přenosných bezdrátových ovladačích s tlačítky. Výhodou je možnost použití více ovladačů s různými barvami najednou a také pohodlí několika tlačítek přímo v ruce. Ani společnost Microsoft nezůstala pozadu, její projekt Natal zahrnuje ovládání pomocí kamery, a to pouhými gesty lidského těla.

Z toho je jasně vidět, že fenomén počítačového vidění v současnosti zažívá velký vzestup a rozhodně se blízké budoucnosti bude na co těšit. Na druhou stranu, v mnohých případech tak dokonalé zařízení, jako je počítačová myš ničím nahradit nemůžeme.

11 Seznam literatury

[1] ŘÍHA, K.: Pokročilé techniky zpracování obrazu. Elektronické texty VUT, Ústav telekomunikací FEKT VUT v Brně, 2007.

[2] ŠMIRG, O., Zpracování obrazu za účelem řízení Visikonu, Bakalářská práce, Brno: VUT, Fakulta elektrotechniky a komunikačních technologií, 2006,50, pp. 3

[3] MATHWORKS, Inc., MATLAB HELP, Nápověda k programu MATLAB, 2008

[4] *Ubuntu Forums* [online]. 23.4.2008 About Xlib and window managers . Dostupné z WWW:http://ubuntuforums.org/showthread.php?t=898249>.

[5] *Willowgarage.com* [online]. 6.4.2010 OpenCV 2.1 C++ Reference. Dostupné z WWW: <opencv.willowgarage.com/documentation/cpp/index.html>.

[6] *Ideasonboard.org* [online]. 2010 Linux UVC driver and tools. Dostupné z WWW: <http://www.ideasonboard.org/uvc/>.

[7] *Kiv.zcu.cz* [online]. 19.12.1997 Xlib a úvod do X Window protokolu. Dostupné z WWW: <http://www.kiv.zcu.cz/~luki/vyuka/stare-materialy/os/oslinux/2.0.31/sak4/xx.htm>.

[8] NOAH KUNTZ, OpenCV Tutorial. www.dasl.drexel.edu/~noahKuntz/opevCVTut1.html

12 Seznam příloh

Jako příloha slouží pouze CD, kde se nachází kompletní práce provedená v programu MATLAB, celý program pro Linux včetně komentovaného zdrojového kódu a projektu v aplikaci Harpia a také krátká videoukázka zobrazující činnost programu.