Rozpoznávanie dopravných značiek

Mário Kapusta

1. mája 2013

Obsah

1 Úvod **8**

2 Rozpoznávanie objektov **9**

2.1 Rozpoznávanie dopravných značení 9

2.2 Rozpoznávanie iných objektov 10

2.2.1 Rozpoznávanie tvárí 11

3 Výskum **11**

3.1 Matematické metódy 11

3.1.1 Konvolúcia 11

3.1.2 Aproximácia 12

3.1.3 Greenová veta 14

3.2 Funkcionalita OpenCV 14

3.2.1 cvtColor 15

3.2.2 Canny 16

3.2.3 GaussianBlur 17

3.2.4 inRange 18

3.2.5 bitwise_not 20

3.2.6 threshold 20

3.2.7 findContours 22

3.2.8 boundingRect 22

3.2.9 drawContours 23

3.2.10 contourArea 23

3.2.11 fitEllipse 23

4 Návrh riešenia **25**

4.1 Návrh algoritmov 25

4.1.1 Návrh algoritmu pre detekciu farby 25

4.1.2 Návrh algoritmu pre detekciu kruhov 28

4.2 Návrh objektov - UML 30

4.3 Návrh užívteľského prostredia 31

5 Implementácia **33**

5.1 Android implementácia 33

5.1.1 BaseClass 33

5.1.2 ActivityView 34

5.1.3 ActivityFunctionality 34

5.2 Rozpoznávacie jadro 35

5.2.1 Detection 36

5.2.2 Color 36

5.2.3 Shape 37

5.2.4 Traffic 39

6 Výsledky aplikácie **40**

6.1 Detekcia kruhových značiek 40

6.1.1 Značky modrej farby 40

6.1.2 Značky červenej farby 40

7 Záver **41**

Zoznam tabuliek

1 Tabulka znázorňuje vstupy funkcie cvtColor 15
2 Konverzia RGB modelu na HSV[10][28][29] 16
3 Tabulka znázorňuje vstupy funkcie canny 18
4 Tabulka znázorňuje vstupy funkcie GaussianBlur 19
5 Tabulka znázorňuje vstupy funkcie inRange 19
6 Tabulka znázorňuje vstupy funkcie bitwise_not 20
7 Tabulka znázorňuje vstupy funkcie threshold 21
8 Tabulka znázorňuje vstupy funkcie findContours 22
9 Tabulka znázorňuje vstupy funkcie boundingRect 23
10 Tabulka znázorňuje vstupy funkcie drawcontours 24
11 Tabulka znázorňuje vstupy funkcie contourArea 24
12 Tabulka znázorňuje vstupy funkcie fitEllipse 24

Zoznam obrázkov

1 Algoritmus vyhľadávania farby v obraze 26
2 Algoritmus vyhľadávania kruhov v obraze 29
3 Návrh objektov Android aplikácie 32

Abstrakt

Cieľom bakalárskej práce je podrobne informovať čitateľa s problematikou rozpoznávania dopravného značenia. Predstaviť vývoj a aktuálny stav v tejto oblasti a dôkladne oboznámiť s momentálne najlepšou technológiou rozpoznávania objektov pomocou počítačového videnia. V práci je predstavený návrh a taktiež implementácia technológie podľa existujúceho algoritmu rozpoznávania dopravného značenia, upravený pre naše potreby. Súčasťou je plne funkčná Android aplikácia, ktorá dokáže vyhľadať príkazové a zákazové dopravné značenia. Funkčnosť a úspešnosť hľadania aplikácie bola otestovaná a zdokumentovaná.

Kľúčové slová: dopravné značky, počítačové videnie, rozpoznávanie objektov, detekcia, Android, OpenCV

Abstract

Cieľom bakalárskej práce je podrobne informovať čitateľa s problematikou rozpoznávania dopravného značenia. Predstaviť vývoj a aktuálny stav v tejto oblasti a dôkladne oboznámiť s momentálne najlepšou technológiou rozpoznávania objektov pomocou počítačového videnia. V práci je predstavený návrh a taktiež implementácia technológie podľa existujúceho algoritmu rozpoznávania dopravného značenia, upravený pre naše potreby. Súčasťou je plne funkčná Android aplikácia, ktorá dokáže vyhľadať príkazové a zákazové dopravné značenia. Funkčnosť a úspešnosť hľadania aplikácie bola otestovaná a zdokumentovaná.

Kľúčové slová: traffic signes, computer vision, object recognition, detection, Android, OpenCV

1 Úvod

2 Rozpoznávanie objektov

Rozpoznávanie objektov v počítačovom videní sa zaoberá problematikou rozpoznania už bežne známych objektov v reálnom svete, pomocou technológií. Táto úloha je prekvapivo ťažká. Ľudia vedia rozpoznávať objekty reálneho sveta bez námahy a okamžite. Problém nastáva však pri algoritmickom opise tejto úlohy, ktorej by chápali stroje. [23]

Okrem zložitého algoritmického problému, tu existuje aj skutočnosť, že pre kvalitné rozpoznanie akéhokoľvek objektu sme taktiež závislí na hardware. Keďže rozpoznávame z obrazu, tento obraz je potrebné zachytávať kamerou. Čím lepšie vie kamera pracovať so svetlom a má kvalitnejšie parametre, tým je väčšia šanca že rozpoznanie objektu bude kvalitnejšie. Pri rozpoznávaní objektov je veľmi dôležité aj umiestnenie kamery. Tam platí, čím kolmejšie k objektu je kamera smerovaná, tým je objekt viac viditeľný a tak aj ľahšie rozpoznateľný. [23]

Problematika rozpoznávania objektov sa ďaľej delí na konkrétnejšie problémy. V našom prípade sa jedná o problematiku rozpoznávania dopravných značení. [23]

2.1 Rozpoznávanie dopravných značení

Problematika rozpoznávania dopravných značení je pomerne nová a pri písaní práce som tak mohol čerpať len z veľmi malého počtu literatúry, zaoberajúcou sa touto témou. Bol som donútený študovať prolematiku podrobne a vytvoriť riešenie, hlavne na poznatkoch z rozpoznávania iných objektov. Popri tvorbe práce, vznikali rôzne ďaľšie práce a publikácie zaoberajúce sa konkrétne témou rozpoznávania dopravných značení. Nebolo však možné sledovať všetky nové objavy a zistenia v tejto oblasti, ktoré vznikali popri tvorbe tejto bakalárksej práci. Rozbor témy rozpoznávania dopravných značení je tak postavený hlavne na informáciách z jesene 2012.

Výskum témy rozpoznávania dopravných značení má v praxi silné uplatnenie. V oblasti dopravy, by to znamenalo pomoc vodičom vnímať a rozpoznávať, dopravné značenia, ktoré si vodič nevšimol, nepochopil poprípade im nerozumie. Napriek tomu, že máme k

dispozícii už dlhý výskum v oblasti počítačového videnia a množstvo kvalitného hardwaru, je úloha rozpoznávania dopravných značení pomerne zložitá. Skutočnosť, že systém musí reagovať pomerne rýchlo na dynamické zmeny ktoré pri jazde autom nastávajú a taktiež s rôznorodým pozadím a viditeľnosťou, ktoré nám prostredie vytvára, nám vytvára komplexnú problematiku v počítačovom videní, ktorej sa oplatí venovať pozornosť.[21]

Konkrétne, problematika rozpoznávania dopravných značiek obsahuje dve fázy, z ktorých sa celý proces skladá:

1. **Detekcia dopravného značenia** - detekuje umiestnenie dopravného značenia v obraze.

2. **Rozpoznanie detekovaného dopravného značenia** - rozpoznáva významovú hodnotu dopravného značenia.

Existuje viac techník pre detekciu dopravných značení, ale ako najúčinnejšia metóda sa ukázalá farebná segmentácia a následné rozpoznanie podľa tvaru. Pri získavaní významovej hodnoty značky je najefektívnejšie využiť neurónové siete. [21]

Prvý rozpoznávač dopravných značení bol do praxe nasadený v roku 2008, spoločnosťou BMW.[2] Nasledujúci rok nasadila svoj prvý systém pre rozpoznávanie dopravných značení aj spoločnosť Mercedes.[22] Tieto prvé rozpoznávače dopravných značení vedeli rozpoznávať len značky ktoré hovorili o rýchlostnom obmedzení. Hneď na to boli tieto automobilky nasledované ďalšími konkurentami, ktorí systém rozpoznávania vylepšovali o rôzne funkcionality. Od roku 2012 vyvýja firma Volvo komplexný systém rozpoznávania dopravných značení pomocou počítačového videnia, ktorý nazvala Road Sign Information. Tento systém je implementovaný vo všetkých nových modeloch spomínanej automobilky.[31]

2.2 Rozpoznávanie iných objektov

Rozpoznávanie jednotlivých objektov v obraze spočíva v tom, objaviť potrebné obrazové regióny hľadaného modelu, pričom sa snažiť ignorovať pozadie. Tieto regióny sú však pre každý objekt iné. Niekedy je potrebné sledovať farbu, inokedy tvary alebo pohyb. Pri

niektorých špecifických objektoch je potrebné hľadať rôzne body a počítať uhly, ktoré zvierajú. Medzi tieto špecifickejšie objekty patrí napríklad problematika rozpoznávania tvárí. [25]

2.2.1 Rozpoznávanie tvárí

Proces rozpoznávania tvárí je postavený na takom princípe, že sa snažíme hľadať už známe kontrasty medzi regiónmy na tváry a ich priestorových vzťahov ktoré vyjadrujú. Tieto vzťahy nie sú také jednoznačné a tak sa museli vypracovať algoritmy, a spôsoby ktoré problematiku riešia. Najpoužívanejšia metóda je používať neurónovú sieť, ktorej posielame vzorky správnych a nesprávnych vzťahov a naučíme stroj rozpoznávať tváre sám. V súčasnosti už nie je problém nájsť takúto technológiu použítú kdekoľvek vo svete. [30]

3 Výskum

Cieľom práce je vypracovať komplexný návrh riešenia pre vyhľadávanie a rozpoznávanie dopravného značenia a taktiež vytvoriť funkčnú aplikáciu, ktorá bude schopná rozpoznať zvislé dopravné značenia. Táto aplikácia bude naprogramovaná v jazyku Java a bude spustiteľná na operačnom systéme Android 2.3, ktorý je určený pre mobilné zariadenia. Computer vision (počítačové videnie), nám zaručí open-source knižnica OpenCV.

3.1 Matematické metódy

Mnoho matematických metód sa bude priamo vysvetlovať pri predstavovaní danej OpenCV funkcionality. V tejto sekcii si predstavíme také matematické metódy ktoré nám pomůžu lepšie sa orientovať pri opise konkrétnych funkcionalít OpenCV.

3.1.1 Konvolúcia

Konvolúcia je matematická metóda, ktorá systematicky prechádza celý obraz a na výpočet novej hodnoty bodu využíva malé okolie O reprezentatívneho bodu. Táto hodnota

je zapísaná do nového obrazu. Diskrétna konvolúcia má tvar:

$$g(x,y) = \sum_{(m,n)} \sum_{(e^0)} h(x-m, y-n) f(m,n)$$

kde f predstavuje obrazovú funkciu pôvodného obrazu, g predstavuje obrazovú funkciu nového obrazu, h predstavuje konvolučnú masku alebo konvolučné jadro, h nám udáva koeficienty jednotlivých bodov v okolí O. Najčastejšie sa používajú obdĺžnikové masky s nepárnym počtom riadkov a stĺpcov, pretože v tom prípade môže reprezentatívny bod ležať v strede masky.

Transformácie v lokálnom okolí bodu sa delia na dve skupiny:
Vyhladzovanie – tieto metódy sa snažia potlačiť šum v obraze, ale rozostrujú hrany.
Ostrenie – detekcia hrán a čiar, ale zosilňuje šum.

Podľa matematických vlastností môžeme metódy predspracovania rozdeliť na
Lineárne metódy – novú jasovú hodnotu bodu počítajú ako lineárnu kombináciu vstupných bodov. Napr.: priemerovací filter
Nelineárne metódy – berú do úvahy len body s určitými vlastnosťami. Napr.: mediánový filter. [4]

3.1.2 Aproximácia

Aproximácia je matematická metóda pri ktorej sa snažíme vyjadriť zložitú funkciu jednoducho. Túto metódu sa snažíme uskutočniť aritmetickými operáciami, ktoré dokze uskutočniť počítač. Jednou z najlepšou metódou vyjadrenia funkcie jednoduchšie je cez polynómy, čo sú vlastne najjednoduchšie funkcie, ktoré možno na počítači vypočítať priamo. Taktiež sa dajú ľahko integrovať a derivovať a vo všeobecnosti sa s nimi jednoducho zaobchádza. Každá aproximácia je presná na určitom intervale, mimo intervalu sú funkcie odlišné. [11]

Predstavme si, že našou úlohou je opísať rozloženie pôdneho znečistenia istou chemikáliou. K dispozícii máme samozrejme meracie prístroje. Jednotlivými vrtmi odoberáme vzorky pôdy, ktoré potom podrobíme analýze. Prolém spočíva v tom, že nie je možné, aby sme takto zmapovali celú oblesť dokonale, keďze sme časovo aj finančne obmedzení. Takže našou úlohou bude dostať dostatočne presný opis znečistenia celého územia z konečného počtu meráni. Musíme tak nejakým spôsobom preniesť namerané hodnoty na celú oblasť. O toto sa nám stará aproximácia. Existuje mnoho spôsobov ako tento prenos uskutočniť. Voľba metódy závisí od konkrétnej situácie. [3]

Existuje niekoľko delení aproximácie, predstavíme si niekoľko základných typov a delení. Výber danej metódy závisí od konkrétneho problému.

1. Rozdelenie aproximácií podľa aproximačnej funkcie

   - **Lineárny typ**

   $$f(x) \approx a_0 g_0(x) + a_1 g_1(x) + \ldots + a_m g_m(x)$$

   - **Racionálny typ**

   $$f(x) \approx \frac{a_0 g_0(x) + a_1 g_1(x) + \ldots + a_m g_m(x)}{b_0 g_0(x) + b_1 g_1(x) + \ldots + b_m g_m(x)}$$

2. Rozdelenie aproximácií podľa zvolených konštánt

   - **Interpolačná aproximácia** - Pri interpolácii si vyberieme nejaké body na vzore, čo je vlastne zložitá funkcia, z ktorej chceme dostať jednoduchšiu, napríklad obraz. Obraz musí tými bodmi na vzore prechádzať. Funkcie musia mať rovnakú deriváciu v danom bode.
   - **Aproximácia metódou najmenších štvorcov** - Funkcia *f(x)* a jej aproximácia majú pri tomto type aproximácie podobný obsah pod krivkou.
   - **Čerbyševova aproximácia** - Tento typ aproximácie sa snaží o najmenší rozdiel medzi funkciou *f(x)* a jej aproximáciou v určitom intervale.

V našej práci budeme využívať aproximáciu nepriamo. Väčšinou pôjde o aproximáciu pomocou bodov a teda interpoláciu. [11]

3.1.3 Greenová veta

Greenová veta bude použitá v práci opäť nepriamo. Bude ju používať OpenCV knižnica napríklad na výpočet veľkosti ľubovolnej kontúry. Greenová veta nám umožnuje previesť výpočet krivkového integrálu druhého druhu po jednoduchej uzavretej krivke na výpočet dvojného integrálu. [5]

Formulácia - *Nech $D \subset R^2$ je regulárna uzavrená oblasť, $\partial\vec{D}$ kladne orientovaná hranica oblasti D. Nech $F = (F_1, F_2)$ je vektorové pole triedy c^1 v D. Potom platí:*

$$\int_{\partial\vec{D}} F d\vec{s} = \int\int_D \mathrm{rot} F d\mu,$$

kde μ značí Jordanovú mieru. V zložkovom tvare.

$$\int_{\partial\vec{D}} F_1 dx + F_2 dy = \int\int_D \left(\frac{\partial f_2}{\partial x} - \frac{\partial f_1}{\partial y}\right) dxdy. \text{ [5]}$$

3.2 Funkcionalita OpenCV

OpenCV je open source knižnica počítačového videnia. Knižnica je napísaná v programovacích jazykoch C a C++. Aktívne sa pracuje na rozhraniach pre Python, Ruby, Matlab, Javu a iných programovacích jazykoch. V našej práci sme sa sústredili na verziu pre programovací jazyk Java, ktorý sa používa pri tvore aplikácii pre Android OS. [6]

OpenCV knižnica bola navrhnutá tak, aby funkcie použité v tejto knižnici, boli čo najefektívnejšie a čo najviac zamerané na real-time aplikácie. Knižnica je napísaná v optimalizovanom jazyku C a tak môže jednoducho využiť aj silu viacjadrových procesorov. Taktiež existujú knižnice, špeciálne určené pre procesory s architektúrou Intel. IPP (Integrated Performance Primitives) knižnice sa skladajú z nízko levelových optimalizovaných postupov a rôznych algoritmických olastí, ktoré pracujú na procesoroch s architektúrou

Premenná	Dátový typ	Popis
src	Mat	Vstup je 8-bitový, 16-bitový obraz alebo formát čísla s plávajúcou desatinou čiarkou.
dst	Mat	Výstupný obraz s rovnakými parametrami ako na vstupe.
code	int	Farebné spektrum ktoré do ktorého požadujeme obraz previesť.

Tabuľka 1: Tabulka znázorňuje vstupy funkcie cvtColor

Intel oveľa efektívnejšie. [6]

Jeden z hlavných cieľov OpenCV je sprístupniť jednoducho použiteľné prostredie ktoré pomôže developerom ľahko a rýchlo budovať aplikácie s použitím počítačového videnia pre rôzne použitia v oblasti, medicíny, bezpečnosti, robotiky, dopravy, priemyselnej výroby a iných, pre ktoré ma OpenCV dokonca aj špecifické funkcionality. [6]

Pre olasť rozpoznávania ojektov sú taktiež mnohé špecifické funkcionality. Pri problematike rozpoznávania zvislích dopravných značení sme niektoré z nich použili a preto je potrebné si pre lepšie pochopenie problematiky tieto funkcie vysveliť podrobnejšie.

3.2.1 cvtColor

Funkcia *cvtColor* prevedie obraz z jedného farebného spektra do iného. Je to jedna z najpoužívanejších funkcií, keďže na rozpoznávanie objektov je potrené si obraz pripraviť cez mnohé farebné filtre. Vstupné parametre je možné pozorovať pri tabuľke 1. [28] [29]

Pri používaní funkcie *cvtColor*, je potrebné si určiť o akú konverziu ide. OpenCV, už má k dispozícii predpripravené konštanty, ktoré konverziu lepšie vyjadrujú. Matematický prepočet si OpenCV už spraví v jadre. Konverzií je v OpenCV naprogramovaných už mnoho, my si predstavíme matematický model konverzie, ktorú v našom prípade rálne využijeme. Jedná sa o konverziu z BGR(pri OpenCV je poradie kanálov pre model RGB

$$BGR \leftrightarrow HSV$$

$$V \leftarrow max(R, G, B)$$

$$S \leftarrow \begin{cases} \frac{V - min(R,G,B)}{V}, & \text{pokiaľ } V \neq 0 \\ 0, & \text{pokiaľ } V = 0 \end{cases}$$

$$H \leftarrow \begin{cases} \frac{60(G-B)}{V - min(R,G,B)}, & \text{pokiaľ } V = R \\ \frac{120+60(B-R)}{V - min(R,G,B)}, & \text{pokiaľ } V = G \\ \frac{240+60(R-G)}{V - min(R,G,B)}, & \text{pokiaľ } V = B \end{cases}$$

Pokiaľ $H < 0$, tak $H = H + 360$

Na výstup pôjde $0 \leq V \leq 1, 0 \leq S \leq 1, 0 \leq H \leq 1$

Tabuľka 2: Konverzia RGB modelu na HSV[10][28][29]

zoradený opačne) do farebného modelu HSV a späť. [28] [29]

V prípade 8 a 16 bitového obrazu je potrené jednotlivé kanály R,G a B previesť do formátu s plávajúcou desatinou čiarkou a zmenšiť rozsah od 0 do 1.

3.2.2 Canny

Hlavná úloha funkcie *Canny* je vyhľadávať okraje, kontúry a hrany všetkých objektov. Pri kombinácii s rôznymi filtrami, môžeme docieliť, vyhľadanie hrán úmyselného objektu. Na rozoznávanie sa využíva algoritmus *Canny86*. [27] [29]

Kontúrový alebo hranový detektor by mal spĺňať tri kritéria, ktoré určil John Canny.

1. Detekčné kritérium, detektor nesmie zabudnúť na významnú hranu a na jednu hranu môže byť maximálne jedna odozva.

2. Lokalizačné kritérium, rozdiel medzi skutočnou a nájdenou hranou má byť minimálny.

3. Kritérium jednej odozvy.

Cannyho detektor využíva konvolúciu s dvojrozmerným Gaussianom a deriváciu v smere gradientu. Poskytuje informácie o smere a veľkosti hrany. Nech G je dvojrozmerný Gaussian. Nech Gn je prvá derivácia G v smere gradientu

$$G_n = \frac{\delta G}{\delta n} = n \bigtriangledown G$$

kde n je smer gradientu, ktorý dostaneme nasledovne

$$n = \frac{\bigtriangledown (G * f)}{| \bigtriangledown (G * f)|}$$

Hranu dostaneme v bode, kde funkcia $G_n * f$ dosiahne lokálne maximum, a druhá derivácia sa rovná nule.

$$\frac{\delta^2}{\delta n^2} G * f = 0$$

Pre silu hrany platí:

$$|G_n * f| = | \bigtriangledown (G * f)|$$

Kritérium jednej odozvy sa dosahuje následne prahovaním. [4] [7] [8]

Vstupné parametre je možné pozorovať pri tabuľke 3. Najmenšia hodnota medzi *threshold1* a *threshold2* je použitá na prepájanie kontúr. Tá najväčšia hodnota je použitá ako začínajúci segment najsilnejších kontúr. Pri správnom nastavení, sa dá dosiahnuť pomerne kvalitné odstránenie nepotrených kontúr. [27] [29]

3.2.3 GaussianBlur

Vyhladzuje obraz pomocou *GaussianBlur* filtra. [18] [29]

Premenná	Dátový typ	Popis
image	Mat	Vstup je 8-bitový obraz s jedným farebným kanálom.
edges	Mat	Výstup je mapa všetkých nájdených kontúr.
threshold1	double	Prvá prahová hodnota (threshold).
threshold2	double	Druhá prahová hodnota (threshold).

Tabuľka 3: Tabulka znázorňuje vstupy funkcie canny

GaussianBlur filter funguje na princípe *N * N* konvolúcie pri ktorej sa každý pixel prehodnotí na základe *Gaussian* funkcie. Táto funkcia tak prevedie rozostrenie pre každý pixel obrazu. [24]

$$H(x, y) = \frac{1}{2\pi\sigma^2} e^{\frac{(x^2)+(y^2)}{2\sigma^2}}$$

Princíp konvolúcie 2D obrazu je postavený na tom, že sa systematicky snažíme spracovávať okolie pixelu a dostať výslednú hodnotu z okolia reprezentatívneho bodu. Konvolúcia sa často používa pri spracovávaní obrazu, ako je vyhladzovanie obrazu, ostrenie, detekcia hrán a obrázkov. [1] [4]

Vstupné parametre pre *GaussianBlur* je možné pozorovať pri tabuľke 4. Pri premennej *ksize* si môžeme napríklad nastaviť veľkosť matice, ktorá sa bude pri konvolúcii používať. Veľkosť matice pri konvolúcii ovplivní rozostrenie. Čím väčšiu maticu používame, tím väčšie rozostrenie dostaneme. [18] [29]

3.2.4 inRange

Funkcia *inRange* zisťuje, či sa prvky poľa nachádzajú medzi prvkami ďalších dvoch polí.

Funkcia kontroluje rozsah nasledujúco:

Premenná	Dátový typ	Popis
src	Mat	Vstup je obraz s ľubovolným počtom farebných kanálov.
dst	Mat	Výstup s rovnakými parametrami ako bol vstup.
ksize	Size	Veľkosť Gaussian jadra. Matica konvolúcie.
sigmaX	double	Smerodajná odchýlka Gaussian jadra v smere X.
sigmaY	double	Smerodajná odchýlka Gaussian jadra v smere Y.

Tabuľka 4: Tabulka znázorňuje vstupy funkcie GaussianBlur

Premenná	Dátový typ	Popis
src	Mat	Vstupné zdrjové pole.
lowerb	Scalar	Spodná hranica poľa alebo skalárna veličina.
upperb	Scalar	Vrchná hranica poľa alebo skalárna veličina.
dst	Mat	Výsledné pole, rovnako veľké ako vstup.

Tabuľka 5: Tabulka znázorňuje vstupy funkcie inRange

- Pre každý prvok vstupného pola s jedným kanálom

$$dst(I) = lowerb(I)_0 \leq src(I)_0 \leq upperb(I)_0$$

- Pre každý prvok vstupného pola s dvomi kanálmy

$$dst(I) = lowerb(I)_0 \leq src(I)_0 \leq upperb(I)_0 \wedge lowerb(I)_1 \leq src(I)_1 \leq upperb(I)_1$$

- A tak ďaľej...

Vstupné parametre pre *inRange* je možné pozorovať pri tabuľke 5. [19] [29]

Premenná	Dátový typ	Popis
src	Array	Vstupné pole plné bitov.
dst	Array	Výstupné pole plné invetovaných bitov

Tabuľka 6: Tabulka znázorňuje vstupy funkcie bitwise_not

3.2.5 bitwise_not

Je jednoduchá funkcia, ktorá invertuje všetky bity v poli ktoré jej pošlete. Taktiež má aj jednoduché vstupné parametre, ktoré vidieť aj v tauľke 6. [12] [29]

3.2.6 threshold

Aplikuje pevnú prahovú úrovní pre každý prvok poľa. Zvyčajne sa používa na získanie binárnej úrovne obrazu v odtieňoch sivej, alebo pre odstránenie šumu. Funkcia *treshold* funguje na princípe filtrovania pixelov ktoré majú príliž veľkú, alebo príliž malú hodnotu. Existuje niekoľko možností ako tento šum odstrániť.

- THRESH_BINARY

$$dst(x,y) = \begin{cases} maxval, & \text{pokiaľ } src(x,y) > tresh \\ 0, & \text{inak} \end{cases}$$

- THRESH_BINARY_INV

$$dst(x,y) = \begin{cases} 0, & \text{pokiaľ } src(x,y) > tresh \\ maxval, & \text{inak} \end{cases}$$

Premenná	Dátový typ	Popis
src	Mat	Vstupný 8-bitový obraz s jedným kanálom.
dst	Mat	Výstupný 8-bitový obraz s jedným kanálom.
thresh	double	Prahová hodnota
maxval	double	Maximálna hodnota ktorú môže použiť na niektoré typy výpočtu.
type	int	Typ výpočtu

Tabuľka 7: Tabulka znázorňuje vstupy funkcie threshold

- THRESH_TRUNC

$$dst(x,y) = \begin{cases} trashold, & \text{pokiaľ } src(x,y) > tresh \\ src(x,y), & \text{inak} \end{cases}$$

- THRESH_TOZERO

$$dst(x,y) = \begin{cases} src(x,y), & \text{pokiaľ } src(x,y) > tresh \\ 0, & \text{inak} \end{cases}$$

- THRESH_TOZERO_INV

$$dst(x,y) = \begin{cases} 0, & \text{pokiaľ } src(x,y) > tresh \\ src(x,y), & \text{inak} \end{cases}$$

Parametre ktoré táto funkcia akceptuje a s ktorými pracuje sú viditeľné v tauľke 7 [12] [29]

Premenná	Dátový typ	Popis
image	Mat	Vstup je 8-bitový obraz ktorý má len jeden kanál, kde všetky hodnoty tohoto kanála ktoré sú väčšie ako 0, sa správajú ako keby mali hodnotu 1.
contours	List:MatOfPoint	Výstup je zoznam kontúr. Každá kontúra je uložená ako vektor bodov.
hierarchy	Mat	Voliteľný výstupný vektor obsahujúci informácie o typológii obrazu. Pre každú kontúru obsahuje množstvo elementov.
mode	int	mód, aleo skôr typ kontúr ktoré budeme chcieť rozpoznávať.
method	int	Metóda aproximácie.

Tabuľka 8: Tabulka znázorňuje vstupy funkcie findContours

3.2.7 findContours

Funkcia *findContours* je prepracovaná metóda hľadania obrysov. Jednoducho nájde obrysy, aleo kontúry v binárnom obraze pomocou algoritmu od Satoshi Suzukiho pre vyhľadávanie čiar v binárnom obraze. Vyhľadané obrysy sú veľmi užitočné pri rozpoznávaní tvarov a objektov. Pri rozpoznávaní dopravných značení je našou snahou taktiež rozpoznať napríklad kruhové tvary zákazových dopravných značení. [16] [29]

obkec o suzuky algoritme - musim si nastudovat jeho pracu [26]

Parametre funkcie vidiet v tabuľke 8

3.2.8 boundingRect

Funkcia *boundingRect* je ďalšia jednoduchá funkcia. Dokáže jednoducho vypočítať a ohraničiť nejaké zoskupenie bodov do odĺžnika. V našom prípade funkciu využijeme na to, aby sme vedeli získať výrez dopravného značenia. Vstup pre funkciu je len samotné

Premenná	Dátový typ	Popis
points	MatOfPoint	Zoskupenie 2D bodov vo vektore.

Tabuľka 9: Tabulka znázorňuje vstupy funkcie boundingRect

zoskupenie bodov, ako vidieť aj na tabuľke 9. [13] [29]

3.2.9 drawContours

Funkcia *drawContours* je vykreslovacia funkcia. Kreslenie kontúr pracuje s maticami. Dokáže vykresliť akýkoľvek tvar, ktorý je definovaný vektorom. [15] [29]

Funkcia je pomerne zložitá na parametre. Podrobnejšie je rozobratá v tabuľke 10

3.2.10 contourArea

Jednoduchá funkcia, ktorá prepočítava veľkosť kontúry. Túto veľkosť sa dá jednoducho využiť pri eliminácii malých kontúr, ktoré pri rozpoznávaní dopravného značenia nevyužijeme. Pri výpočte je použitá Greenová veta. Funkcia nám vracia počet pixelov, ktoré kontúra obsahuje. Sú to pixely ktoré nemajú nulovú hodnotu. Obsahuje len jeden parameter, ktorý je opísaný v tabuľke 11. [14] [29]

3.2.11 fitEllipse

Funkcia *fitEllipse* opäť patrí medzi ľahšie použiteľné funkcie. Jej hlavnou úlohou je vykresliť elipsu okolo skupiny 2D bodov. Pri vykreslení sa snaží o to, aby bola vykreslená najmenšia možná elipsa pri čom využíva algoritmus Fitzgibbon95. Parametre funkcie je vidieť v tabuľke 12.

Obkec o Fitzgibbon95 algoritme. [17] [29]

Premenná	Dátový typ	Popis
image	Mat	Obraz do ktorého budú kontúry vkreslené.
contours	List:MatOfPoint	Zoznam všetkých kontúr ktoré chceme vykresliť. Každá kontúra je uložená ako vektor bodov.
contourIdx	int	Index, ktorý určuje ktorú kontúru chceme vykresliť. Negatívne číslo hovorí o tom, že chceme vykresliť všetky kontúry.
color	Scalar	Farba vykreslenej kontúry.
thickness	int	Šírka kontúry.
lineType	int	Typ vykreslenej čiary.
hierarchy	Mat	Voliteľný výstupný vektor obsahujúci informácie o typológii obrazu. Pre každú kontúru obsahuje množstvo elementov.
maxLevel	int	Maximalný level vykreslených kontúr. Tento parameter je funkčný, len v prípade že je použitá hierarchia.
offset	Point	Voliteľný parameter posunov. Posunie všetky kontúry podľa zadaných súradníc.

Tabuľka 10: Tabulka znázorňuje vstupy funkcie drawcontours

Premenná	Dátový typ	Popis
contour	Mat	Kontúra ktorú chceme prepočítať.

Tabuľka 11: Tabulka znázorňuje vstupy funkcie contourArea

Premenná	Dátový typ	Popis
points	MatOfPoint2f	Vektor 2D bodov okolo ktorých chceme elipsu vykresliť.

Tabuľka 12: Tabulka znázorňuje vstupy funkcie fitEllipse

4 Návrh riešenia

Po dôkladnom naštudovaní literatúry a potrebných algoritmov je našim cieľom vyhotoviť riešenie, ktoré by dokázalo detekovať zvislé dopravné značenia. Návrh bude pozostávať z návrhu algoritmov, návrhu objektov a návrhu užívateľského prostredia. Všetky algoritmy boli navrhnuté na základe práce [21] z ktorej sme sa inšpirovali.

4.1 Návrh algoritmov

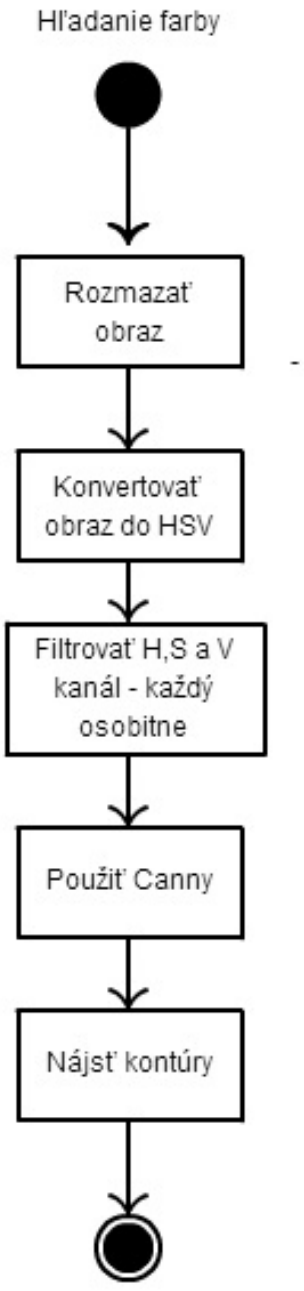
Ako metódu rozpoznávania som si zvolil detekciu dopravného značenia podľa tvaru a farby. Algoritmy ktoré som navrhol, sú postavené na princípe rozpoznania farebného rozhrania hľadaného objektu a následné detekovanie potrebného tvaru. Pri opise som sa zameral na detekciu značiek, ktoré sú na cestách najviac početné. Na cestách prevládajú dopravné značenia, ktoré sú červenej a modrej farby. Z tvarov prevládajú kruhy a trojuholníky. Samotné rozpoznanie bolo uskutočnené pomocou neurónových sietí. Táto metóda je najlepšia na počítačové učenie objektov. Na rozdiel od detekcie dopravného značenia, pre riešenie neurónových sietí použijeme už existujúcu knižnicu.

4.1.1 Návrh algoritmu pre detekciu farby

Ako prvý algoritmus som si vybral detekciu červenej farby. Pre detekciu farieb sa v literatúre odporúča najprv previesť vstup na farebný model HSV. Vstup prichádza vo farebnom formáte RGB. Farebný model HSV je jeden z dvoch najpoužívanejších valcovo súradnicových reprezentácii bodov pre RGB model. [9]

Na začiatok by sa mal vstup(bitmapa) konvertovať na binárnu maticu.

Najväčšia výhoda dopravného značenia je, že je silne kontrastné od ostatného prostredia. Túto vlastnosť môžeme perfektne využiť v náš prospech a pomocou pomocou rozmazania obrazu, môžeme dosieliť to, že sa zbavíme slabších kontú hneď na začiatku. V OpenCV je pre rozmazávanie obrazu na výber viacero metód, no my použijeme matódu *GaussianBlur*, ktorá už názvom prezrádza použitie známeho filtra *Gaussian blur*



Obr. 1: Algoritmus vyhľadávania farby v obraze

Keďže sa snažíme dostať náš vstupný obraz do formátu HSV, o ktorú sa stará funkcionalita *cvtColor* potrebujeme mu nastaviť vstup tak, aby obraz vedel bez problémov spracovať. Keďže na väčšine mobilných zariadení prichádza do zariadenia obraz vo formáte RGBA, ďalší krok bude nepríklad konvertovanie formátu RGBA na formát RGB.

Ďalej bude nasledovať samotná konverzia obrazu do HSV pomocou už spomínanej metódy *cvtColor*.

Ďalší krok bude spracovať každý kanál farebného modelu HSV samostatne. Ako prvý spracujeme *Hue* kanál, ktorý sa stará o farebný odtien každého pixelu. *Hue* Farba sa v tomto kanáli určuje podľa stupňov. Primárne sa začína na stupni 0°, čo predstavuje zelenú farbu, postupne prechádza do modrej, ktorá sa nachádza na 120° stupňoch z kade prechádza cez červenú na 240° a keďže je to model kruhový, vracia sa do zelenej na 360°. Pomocou funkcie *inRange* by nemal byť problém určiť rozhranie stupňov, ktoré sme schopný akceptovať ako hľadanú farbu pre hľadané naše dopravné značenia. Ďalší kanál je *Saturation*, ktorý predstavuje sýtosť farby. Táto sýtosť sa vyjadruje v percentách, kde 0% predstavuje šedú a 100% je plne sýta farba.[10] V našom prípade je postacuje metóda *threshold*. Posledný kanál *Value* vyjadruje hodnotu jasu. Keďže v praxi znamená znižovanie jasu pridávanie čiernej do základnej farby, pre hľadanie červenej farby na dopravnom značení nie je potrebné s týmto kanálom pracovať, lebo červená farba použitá na dopravných značeniach je pomerne svetlá. Pri hľadaní modrej je túto farbu potrebné trochu stmaviť a tak použijeme opäť funkciu *threshold*.

Na koniec potrebujeme dostať len kontúry hľadanej farby. Najpr si budeme musieť spojiť jednotlivé kanály späť do jednej binárnej matice použitím metódy *Canny*. Po tomto kroku by nám mali ostať len čierny obraz a biele škvrny predstavujúce červenú farbu v požadovanom rozsahu. Z týchto bielych objektov, budeme potrebovať len okraje a tak použijeme metódu *findContours*, ktorá sa postará o to, že dostaneme pole kontúr z celého obrazu. S týmito kontúrami potom ďalej pracujeme a rozoznávame z nich hľadané útvary.

4.1.2 Návrh algoritmu pre detekciu kruhov

Pri detekcii dopravného značenia v tvare kruhu, je dôležité počítať s tým, že nehľadáme úplný kruh. Kruhové dopravné značenia sú vyrábané ako dokonalý kruh, no pri ich rozpoznávaní si je potrené uvedomiť, že na objekt sa pozeráme z rôznych uhlov. Táto skutočnosť nám prináša do prolematiky dôležitý fakt, že v skutočnosti to nie sú kruhy čo hľadáme, ale sú to elipsy. Celý algoritmus je možné vidieť na orázku č. 2

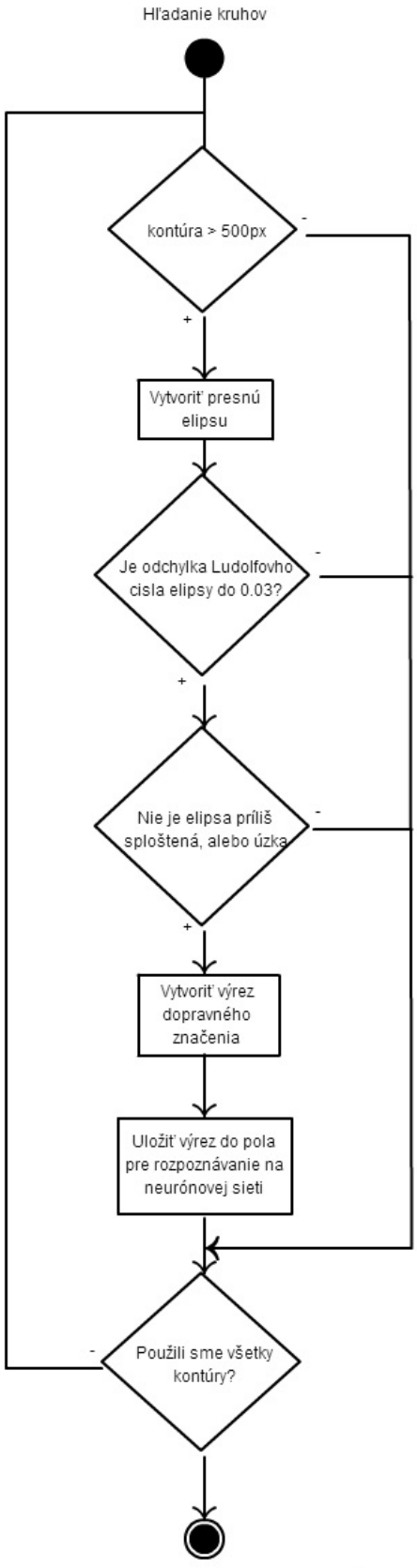
Keďže v predchádzajúcej kapitole sme si navrhli riešenie, ktoré nám vracia len kontúry hľadanej farby, môžeme pokračovať od tohto bodu. Ako prvé si spravíme cyklus, ktorým budeme prechádzať všetky naše vyhľadané kontúry farieb. Aby sme eliminovali počet prebytočných kontúr, je potrené spracovávať čo najrelevantnejšie výsledky. Tento úkon vykoná metóda *contourArea*, vďaka ktorej budeme posielať na ďalšie spracovanie len kontúry väčšie ako 500 pixelov.

Vzhľadom na to, že výsledky, ktoré dostávame ešte nemôžeme nazvať elipsami, musíme si naše kontúry na elipsy upraviť. Tento úkon vykonáva metóda *fitEllipse*, ktorá upraví kostrbaté kontúry, ktoré sa aspoň trochu podobajú elipse, na matematicky presnú elipsu.

Keď už máme detekované elipsy, nastáva posledný krok, a tým krokom je, určiť si toleranciu elipsy dopravného značenia, ktorú vyhľadávam. Táto tolerancia, je vlastne tolerancia nepresnosti, pri výpočte Ludolfovho čísla. Ďalším krokom je tak výpočet už spomínaného ludolfovho čísla a následné overenie jeho nepresnosti. Pokiaľ je výsledná hodnota vyhovujúca, nájdený objekt vyrežeme, a zasielame na rozpoznanie neurónovej sieti, ktorá zistí o akú značku sa presne jedná.

Výpočet Ludolfovho čísla:

$$\pi = \frac{o}{d}$$



Obr. 2: Algoritmus vyhľadávania kruhov v obraze

Úprava výpočtu Ludolfovho čísla pre elipsu:

$$p = \frac{o}{d} = \frac{o}{(\frac{1}{2}y) * (\frac{1}{2}x)}$$

Získanie tolerancie:

$$\pi - p < 0.03$$

Pre určovanie tolerancie elipsy, je možné použiť ešte jednu metódu, a tou je overovanie podľa osí. Pokiaľ je x-ová os dvoj-násobne väčšia ako y-ová, ide už o elipsu, ktorú by sme ďalej len ťažko identifikovali. Takýto nežiaduci stav môže nastať, pokiaľ sa na značku pozeráme na dopravné značenie z príliž veľkého uhlu.

Dva nežiaduce stavy tvaru dopravného značenia:

$$1.) \ \frac{\frac{1}{2}x}{\frac{1}{2}y} > 2$$
$$2.) \ \frac{\frac{1}{2}y}{\frac{1}{2}x} > 2$$

4.2 Návrh objektov - UML

Algoritmy, ktoré boli navrhnuté, je potrebne správne zakomponovať do objektovo orientovaného modelu našej android aplikácie. Je potrebné si správne navrhnúť ako budú jednotlivé algoritmy a funkcionality rozdelené do tried a taktiež aj to, ako budú triedy medzi sebou spolupracovať. Celá štruktúra je najlepšie viditeľná na obrázku 3.

- **SurfaceView** - Táto trieda je v jadre Android OS, ale je si potrebné vysvetliť jej probližnú funkcionalitu, keďže z nej priamo dedíme a upravujeme ju v ďalších triedach našej aplikácie. Poskytuje špecializovanú kresliacu plochu. Podoba tejto plochy

sa dá meniť, dá sa jednoducho meniť napríklad veľkosť alebo samotné umiestnenie. [20]

- **Activity** - Táto trieda je taktiež priamo v jadre Android OS, ale keďže z nej taktiež dedíme, vasvetlíme si jej hlavný princíp. Táto trieda má na starosti komunikáciu s užívateľom. Všetko čo užívateľ s aplikáciou spraví, napríklad čo sa stane s aplikáciou pokiaľ užívateľ pozastaví priebeh vykonávania, alebo čo sa bude diať po výbere jednotlivých položiek v menu. [20]

- **BaseClass** - Táto trieda dedí z triedy *SurfaceView* a nastavuje našu plochu. Keďže na našej aplikácii máme v pláne mať na ploche hlavne pohľad z kamery, v tejto triede sa staráme o všetky správne nastavenia kamery a zobrazenia pre hlavnú plochu aplikácie.

- **ActivityView** - Trieda dedí z triedy *Activity*. V tejto aplikácii sa nastavujú všetky akcie ktoré v našej android aplikácii môžu nastať a taktiež aj naše hlavné menu z ktorého si užívateľ vyberá akcie.

- **Detection** - Je to hlavná trieda zodpovedná za rozpoznanie jednotlivých objektov, nastavujeme v nej taktiež rôzne parametre, ktoré sa použijú neskôr v triedach, ktoré z našej triedy *Detection* dedia.

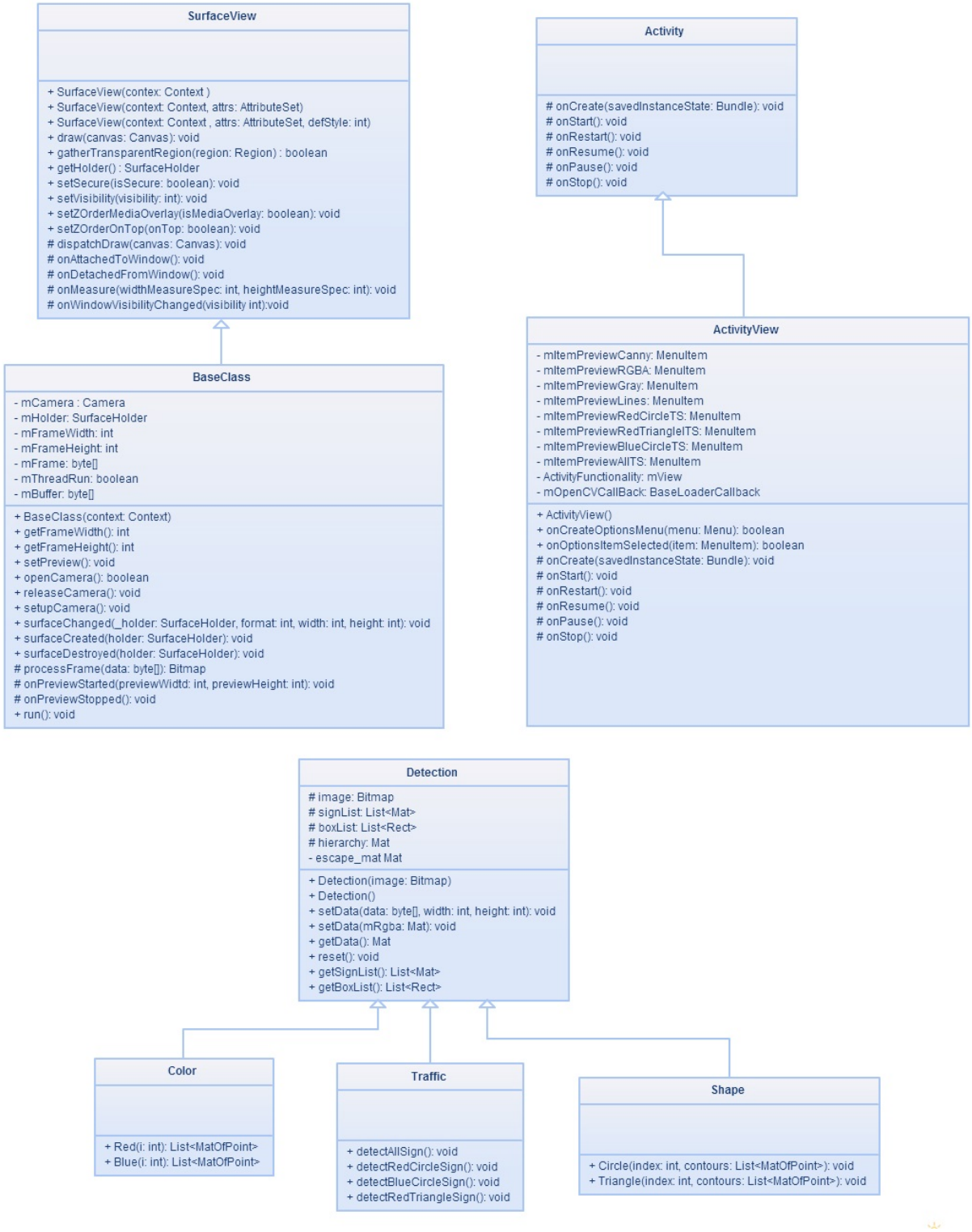
- **Color** - Trieda *Color* bude dediť z *Detection*. Jej hlavná úloha je rozpoznávať jednotlivé farby.

- **Shape** - Trieda *Shape* bude dediť z *Detection*. Jej hlavná úloha je rozpoznávať tvary z obrazu.

- **Traffic** - Trieda *Traffic* bude taktiež dediť z triedy *Detection*. Jej hlavná úloha je rozpoznávať dopravné značenia. Využíva na to spoluprácu tried *Shape* a *Color*.

4.3 Návrh užívteľského prostredia

Prostredie aplikácie bude veľmi jednoduché. Celé rozhranie, bude pozostávať z dvoch častí. Jedna časť bude naša plocha, tam sa budú graficky znázorňovať nájdené objekty, v



Obr. 3: Návrh objektov Android aplikácie

našom prípade dorpavné značenia, ale taktiež aj rôzne filtre, ktoré budeme chcieť prezentovať ako ukážku. Druhá časť bude menu, v ktorom si užívateľ bude vyberať z možností ktoré tam budú a následne sa budú vykonávať rôzne. V menu bude napríklad prepínanie rôznych filtrov alebo spôsobov vyhľadávania dopravných značení.

5 Implementácia

V tejto sekcii sa budem snažiť popísať implementáciu a správne nastavenie technológií, ktoré sa v aplikácii využijú.

5.1 Android implementácia

Pri aplikácii pre android je potrebné upraviť v triedach aj samotnú spoluprácu s mobilom a jadrom Androidu. Rozdelil som túto spoluprácu do dvoch logických celkov. Rozdelil som to na triedu *BaseClass* a triedu *ActivityView*.

5.1.1 BaseClass

V tejto triede nastavujeme plochu našej android aplikácie. Trieda obsahuje mnoho metód ktoré sa venujú napríklad otváraniu kamery a nastaveniam kamery. Metóda zaoberajúca sa otvorením kamery je *openCamera*. V tejto metóde sa zisťuje či sa kamera dá otvoriť, pokiaľ áno, tak ju samozrejme otvoríme a pokiaľ nie, tak ukončíme aplikáciu a dáme vedieť užívateľovi čo sa stalo.

```
1  public boolean openCamera() {
       Log.i(TAG, "openCamera");
3      releaseCamera();
       mCamera = Camera.open();
5      if(mCamera == null) {
         Log.e(TAG, "Nemozem otvorit kameru");
7        return false;
       }
9
       mCamera.setPreviewCallbackWithBuffer(new PreviewCallback() {
```

```
11              public void onPreviewFrame(byte[] data, Camera camera) {
                    synchronized (BaseClass.this) {
13                      System.arraycopy(data, 0, mFrame, 0, data.length);
                        BaseClass.this.notify();
15                  }
                    camera.addCallbackBuffer(mBuffer);
17              }
            });
19          return true;
        }
```

Listing 1: Spustenie kamery

Ďalšia metóda *releaseCamera* uvoľní, alebo reštartne kameru. V metóde *setupCamera* nastavíme potrebné nastavenia kamery a obrazu v ktorom sa bude zobrazovať. My si kameru nastavíme na celú plochu aplikácie a taktiež jej nastavíme automatické ostrenie.

5.1.2 ActivityView

Trieda *ActivityView* rieši všetky aktivity vykonávané v aplikácii. *onPause* metóda zastaví kameru a reštartne ju. Metóda *onResume* nám rieši situáciu, keď sa k aplikácii vrátime. Spúšta znovu kameru a overuje či je možné ďalej pokračovať. Pri negatívnom stave upozorní užívateľa o ukončení aplikácie. Pri metóde *onCreate* načítavame OpenCV a zistujeme, aktuálnu verziu OpenCV v zariadení. Pokiaľ je zastaralá, alebo vôbec neexistuje, vyzveme užívateľa aby si ju dodatočne stiahol. Samozrejme mu pomôžeme s nájdením správnej verzie a pokiaľ má záujem ďalej pokračovať, dovedieme ho ku správnej verzii my. Samozrejme pokiaľ aj tu nastane nejaká chyba, je potrebné aplikáciu ukončiť a upozorniť na to užívateľa. V metóde *onCreateOptionsMenu* nastavíme položky v menu a texty ktoré sa majú vypisovať. Samotná aktivita po kliku na menu sa rieši v metóde *onOptionsItemSelected*.

5.1.3 ActivityFunctionality

V tejto triede sa rieši logika prepájania interakcie s jadrom rospoznávania. Taktiež sa tu rieši to, čo sa stane počas načítavania samotnej aplikácie. Je dobŕe si čo najviac vecí

načítať práve v tejto fáze, aby sme sa odbremenili od záťaže pri behu samotnej aplikácie. Práve toto sa rieši v metóde *onPreviewStarted*, kde si načítame všetky rozmery obrazu pre rôzne filtre. To čo sa stane pokiaľ aplikácia zastane sa rieši v metóde *onPreviewStopped*. Tu je potrebné všetky premenné, polia a zoznamy uvoľniť. Veľmi dôležitá robota sa vykonáva v metóde *processFrame*. Tu komunikujeme s menu a rozdelujeme čo sa stane po zvolení jednotlivých položiek. Technika prepínania bola zvolená pomocou funkcie *switch*, ktorá nás rozdeľuje do rôznych stavov. Pri detekovaní dopravných značení, je to práve tu, kde spúštame vyhľadávanie a následne vykesľujeme na plochu štvorce s nájdenými objektami.

```
Imgproc.cvtColor(mYuv, mRgba, Imgproc.COLOR_YUV420sp2RGB, 4);
//nastavene dat
traffic.setData(mRgba);
//najdene cervenych objektov
traffic.detectRedCircleSign();
//vycistenie zoznamu so stvorcami
boxList.clear();
//naplnenie zoznamu so stvorcami
boxList = traffic.getBoxList();
//vycistenie zoznamu so znackami
signList.clear();
//naplnenie zoznamu so znackami
signList = traffic.getSignList();
//kreslenie stvorcov
for(int i = 0; i < boxList.size(); i++){
  Rect r=boxList.get(i);
  Core.rectangle(mRgba, r.tl(), r.br(), new Scalar(0, 255, 0, 255), 3);
}
```

Listing 2: Výber z menu - hľadanie dopravných značiek

5.2 Rozpoznávacie jadro

Rozpoznávacie jadro je rozdelené do logických celkov do tried, podľa druhu objektov ktoré rozpoznávajú.

5.2.1 Detection

Je to hlavná trieda z ktorej všetky ďalšie triedy ktoré rozpoznávajú konkrétne objekty dedia. V tejto triede sa nastavujú základné informácie potrebné pre rozpoznávanie. Obsahuje dva konštruktory, ktoré sa spúštajú na základe parametrov a pri každom spustení reštartuje všetky objavené dopravné značenia. Keďže aplikácia potrebuje rôzne vstupy, je potrebné nastaviť pre jednotlivé ukážky rôzne vstupné premenné obsahujúce vstupný obraz.

5.2.2 Color

Trieda *Color* je zodpovedná za rozpoznávanie farieb na dopravných značeniach. Obsahuje metódy, ktoré sú pomenované podľa farby ktorú vie daná metóda rozpoznávať. Už ako z názvu vyplíva, metóda *Red* bude rozpoznávať červenú farbu v obraze. V metóde si najprv nastavíme potrebné premenné s ktorými budeme neskôr pracovať. Ako prvé potrebujeme previesť obraz z bitmapy do binárnej matice typu *Mat*, pomocou funkcie *bitmapToMat*. Ďaľej sa pokúsime rozmazať obraz, pomocou *GaussianBlur*. Táto funkcia nám vracia formát obrazu ako RGBA, no pre ďalšiu prácu budeme potrebovať RGB a taktiež HSV. Toto docielime použitím *cvtColor*. Funkcia *split* nám docieli to, že náš obraz rozdelí na 3 kanály, ktoré filter HSV obsahuje a ktoré potrebujeme pre ďalšie filtrovanie obrazu. H kanál budeme filtrovať pomocou funkcie *inRange*, keďže vstup môže byť aj vektor.

```
//konverzia vstupneho obrazu
Utils.bitmapToMat(image, mRGBA);
//rozmazanie obrazu
Imgproc.GaussianBlur(mRGBA,mRGBA,new Size(5, 5),1.5,1.5);
//prevedenie RGBA na RGB
Imgproc.cvtColor(mRGBA,mRGB,Imgproc.COLOR_RGBA2RGB);
//prevedenie RGB na HSV
Imgproc.cvtColor(mRGB,mTemp,Imgproc.COLOR_RGB2HSV);
//rozdelenie
Core.split(mTemp,lHSV);
//filtrovanie kanalu H
mTemp=new Mat();
Core.inRange(lHSV.get(0), new Scalar(90), new Scalar(130), mTemp);
```

```
14  lHSV.set(0, mTemp);
```

Listing 3: Konverzia obrazu cez rôzne filtre

Pre ďalšie dva kanály bude stačiť použiť funkciu *threshold*. Funkcia *Canny* nám prekonvertuje obraz na 8-bitový čierny obraz, na ktorom sa nachádza mnoho bielych objektov. Tieto biele objekty predstavujú červenú farbu. Na záver je potrebné získať všetky červené objekty len ako kontúry. S týmto problémom nám pomôže funkcia *findContours*, ktorá získa z nájdených objektov červenej farby len ich kontúry. Metóda na záver vracia zoznam všetkých nájdených kontúr, ktoré považujeme za okraje za červených objektov. Pri metóde *Blue* a taktiež v prípade vyhľadávania iných farieb, by stačilo zmeniť parametre fo funkciách, ktoré spracovávajú jednotlivé kanály HSV filtra.

5.2.3 Shape

V tejto triede sa venujeme problematike rozpoznávania tvarov. Tvary nerozpoznávame priamo z obrazu, ale zo zoznamu kontúr ktoré sme pri rozpoznaní farby už dostali. Rovnako ako v predchádzajúcej triede, sme pomenovali jednotlivé metódy podľa objektu aký rozpoznáva. Vznikli tak metódy *Circle* a *Triangle*. Síce spracovávame už existujúce kontúry budeme potrebovať aj samotný snímaný obraz a tak je potrebné si tento obraz opäť previesť na binárny obraz typu *Mat*, pomocou funkcie *bitmapToMat*. Nasleduje cyklus s podmienkou na konci, kedy zisťujeme, či už sme na konci, alebo, či nie sme len v dieťati nejakej rodičovskej kontúry. Vytvoríme si buffer, do ktorého načítame dodatočné informácie kontúr, ktoré sa nachádzajú v premennej *hierarchy*. Načítame si potrebnú kontúru s ktorou ideme pracovať. Ako prvé, ju overíme, či je veľkosť kontúry dostatočne veľká, aby malo zmysel sa s ňou zaoberať. Tu sme sa rozhodli, pracovať s kontúrami, ktoré su väčšie ako 500 pixelov. Veľkosť kontúry sa nám podarí získať pomocou funkcie *contourArea*.

```
int buff[] = new int[4];
//i sme dostali so vstupu
hierarchy.get(0, i, buff);
Mat contour = contours.get(i);
int id = i;
i = buff[0];
```

```
if(Imgproc.contourArea(contour) > 500){
```

Listing 4: Načítanie a overenie veľkosti kontúry

Po tento bod je každá metóda v triede *Shape* rovnaká, od tadeto sa mení spôsob rozpoznania jednotlivých tvarov. Pre rozpoznanie kruhov, je našou úlohou nastaviť si najmenšiu možnú elipsu pre danú kontúru. Nie je to však tak jednoduchá úloha. Ako prvé je potrebné konvertovať naše kontúry na maticu typu *MatOfPoint2f*. Toto prevedieme pomocou cyklu, v ktorom iba rozširujeme počet bitov a následne konvertujeme pomocou *toArray*. Hneď ako sa nám konverzia podarí, môžeme použiť našu funkciu *fitEllipse*, ktorá nám vykreslí najmenšiu možnú elipsu.

```
1  //pocet kontur
   int num = (int)contour.total();
3  //vytvorime si pole o dvojnasobnej velkosti samotnej kontury
   int temp[] = new int[num * 2];
5  //nacitame si konturu do docasnej premennej
   contour.get(0, 0, temp);
7  //konvertujeme  List<Point> do MatOfPoint2f pre pouzitie fitEllipse
   for(int j = 0; j < num * 2; j = j + 2){
9    points.add(new Point(temp[j], temp[j+1]));
   }
11 MatOfPoint2f specialPointMtx = new MatOfPoint2f(points.toArray(new Point
       [0]));
   //do premennej bound ukladame dokonalu - najmensiu moznu elipsu
13 RotatedRect bound = Imgproc.fitEllipse(specialPointMtx);
```

Listing 5: Konverzia do MatOfPoint2f a použitie fitEllipse

Je potrebné si vypočítať a overiť toleranciu k hodnote PI. Výpočet sa spraví podľa vzorca opísaného v časti 3.1.2, a následne sa overí tolerancia.

```
1  //vypocita sa hodnota pi
   double pi = Imgproc.contourArea(contour) / ((bound.size.height / 2) * (
       bound.size.width / 2));
```

Listing 6: Vypočet PI

Spraví sa ešte jedno overenie pre elipsu a následne sa vyreže obraz okolo značky a ukladá sa do zoznamu *boxList* s ktorým neskôr môžeme ďalej pracovať.

```
double longAxis;
double shortAxis;
//ziska dve osy elipsy
if (bound.size.height < bound.size.width){
  shortAxis = bound.size.height / 2;
  longAxis = bound.size.width / 2;
} else {
  shortAxis = bound.size.width / 2;
  longAxis = bound.size.height / 2;
}
//this could stop the searching when is elipse too oval
if ((longAxis / shortAxis) < 2.0){
  signList.add(roi);
  boxList.add(box);
}
```

Listing 7: Overenie oválnosti našej elipsy

5.2.4 Traffic

V triede sa nachádzajú jednotlivé metódy rozpoznávania spojené po logických celkoch. Napríklad rozpoznanie kruhových červených značení, alebo všetkých značení. V takej metóde *detectRCS* sa použijú len metódy z tried *Color* a *Shape*.

```
//vypocita sa hodnota pi
List<MatOfPoint> contours = Color.Red(i);
if(contours.size()>0){
  Shape.Circle(contours, 0);
}
```

Listing 8: Detekcia červených okrúhlich dopravných značení

6 Výsledky aplikácie

6.1 Detekcia kruhových značiek

6.1.1 Značky modrej farby

6.1.2 Značky červenej farby

7 Záver

Literatúra

[1] Song Ho Ahn. Convolution, 2005.

[2] BMW Automobiles. Traffic sign recognition, 2010.

[3] Slodička M. Weisz J. Babušíková, J. *Numerické Metódy.* Univerzita Komenského v Bratislave, Bratislava, SK, 1998.

[4] Gábor Blázsovit. Digital image processing - interaktívna učebnica spracovania obrazu, February 2006.

[5] Hrůza B. Brabec J. *Matematická analýza II.* SNTL, Praha, 1986.

[6] Kaehler A. Bradski, G. *Learning OpenCV: Computer Vision with the OpenCV Library.* O'Reilly Media, Inc., Gravenstein Highway North, Sebastopol, CA, 2008.

[7] J. Canny. A computational approach to edge detection. 1986.

[8] Wikipedia Contributors. Canny edge detector, March 2013.

[9] Wikipedia Contributors. Hsl and hsv, February 2013.

[10] Wikipedia Contributors. Hsv, March 2013.

[11] Claudia Csollárová. Aproximácia, July 2005.

[12] OpenCV dev team. bitwise_not, March 2013.

[13] OpenCV dev team. boundingrect, March 2013.

[14] OpenCV dev team. contourarea, March 2013.

[15] OpenCV dev team. drawcontours, March 2013.

[16] OpenCV dev team. findcontours, March 2013.

[17] OpenCV dev team. fitellipse, March 2013.

[18] OpenCV dev team. gaussianblur, March 2013.

[19] OpenCV dev team. inrange, March 2013.

[20] Android Developers. Android dev team, 2013.

[21] Suthakorn J. Lorsakul, A. Traffic sign recognition using neural network on opencv: Toward intelligent vehicle/driver assistance system. 2012.

[22] Mercedes-Benz. Road sign information, 2010.

[23] Rangachar K. Brian S. Ramesh, J. *Machine Vision*. MIT Press and McGraw-Hill, Inc., USA, 1995.

[24] Daniel Rákos. Gaussian blur, September 2010.

[25] Torralba A. Liu C. Fergus R. Freeman W. Russell, B. Object recognition by scene alignment. July 2001.

[26] S. Suzuky. A computational approach to edge detection. 1983.

[27] OpenCV Dev Team. canny, March 2013.

[28] OpenCV Dev Team. cvtcolor, March 2013.

[29] OpenCV Dev Team. *Opencv documentation*, March 2013.

[30] Jones M. Viola, P. Robust real-time object detection. July 2001.

[31] Volvo. Road sign information, 2010.