VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

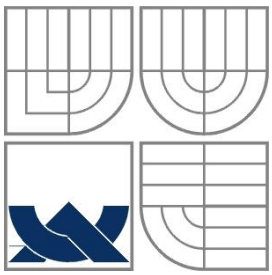
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KONTROLA BEZPEČNÉ VZDÁLENOSTI V AUTĚ PRO PLATFORMU ANDROID

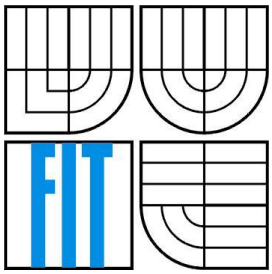
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE MICHAL PRACUCH
AUTHOR

BRNO 2013



VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

KONTROLA BEZPEČNÉ VZDÁLENOSTI V AUTĚ PRO PLATFORMU ANDROID

IN-CAR CONTROL OF MAINTAINING SAFE DISTANCE ON THE ANDROID PLATFORM

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR
MICHAL PRACUCH

VEDOUCÍ PRÁCE
SUPERVISOR
ING. ALEŠ LÁNÍK

BRNO 2013

Abstrakt

Tato bakalářská práce se věnuje detekování automobilů na platformě Android a popisuje několik způsobů detekce objektů. Aplikace využívá kameru mobilního telefonu a zpracovává obraz pomocí OpenCV knihovny. Detekce je založena na porovnávání dvou snímků s využitím ORB detektoru a deskriptoru a je implementována v nativním kódu.

Abstract

This bachelor's thesis presents car detection on Android platform and describes few techniques of object detection. Application uses mobile phone camera and processes images by using OpenCV library. Detection is based on matching of two images with ORB detector and descriptor and is implemented in native code.

Klíčová slova

Android, mobilní telefony, NDK, OpenCV, počítačové vidění, kamera, detekce automobilů, ORB detektor, porovnání obrázků

Keywords

Android, mobile phones, NDK, OpenCV, computer vision, camera, car detection, ORB detector, image matching

Citace

Pracuch Michal: Kontrola bezpečné vzdálenosti v autě pro platformu Android, bakalářská práce, Brno, FIT VUT v Brně, rok 2013

Kontrola bezpečné vzdálenosti v autě pro platformu Android

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením Ing. Aleše Láníka. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

.....
Michal Pracuch
15. května 2013

Poděkování

Chtěl bych poděkovat vedoucímu mé bakalářské práce Ing. Aleši Láníkovi za odbornou pomoc, rady a věnovaný čas při řešení problémů.

Obsah

1 Úvod

Cílem této práce je seznámit se s platformou Android, možnostmi snímání a zpracování obrazu, možnostmi detekce automobilů, prací s GPS na této platformě a implementovat aplikaci, která hlídá porušení bezpečné vzdálenosti. Zařízení s operačním systémem Android se stala velmi rozšířená, proto je zajímavé umožnit lidem je využívat ke zvýšení bezpečnosti na silnicích. Moderní mobilní zařízení získávají stále větší výkon a jsou schopna zpracovávat i náročné algoritmy počítačového vidění.

Tato technická zpráva v kapitole 2 popisuje platformu Android, její architekturu, rozšíření různých verzí tohoto operačního systému, dostupné a podporované vývojářské nástroje a možnosti vývoje v nativním kódu C/C++, knihovnu pro počítačové vidění OpenCV a její zabudování do aplikace pro Android. Dále různé metody detekce objektů, jako odečítání pozadí, kaskádový klasifikátor nebo porovnávání snímků a popis ORB detektoru a deskriptoru klíčových bodů.

Kapitola 3 obsahuje popis navrženého systému detekce, tedy samotný způsob detekování, filtrování výstupu a hlídání dodržení bezpečné vzdálenosti. Pak následuje návrh aplikace, vývojový diagram průběhu inicializace, zpracování obrazu z kamery a zobrazení výsledků na displeji.

Poté je popsána implementace aplikace v kapitole 4. Ta je rozdělena na část v jazyce Java a část v nativním kódu C++. Tyto části obsahují popis jednotlivých tříd a funkcí zajišťujících běh aplikace, zobrazování náhledu kamery, vykreslování na obrazovce, porovnání klíčových bodů a detekci automobilů. Nakonec jsou zobrazeny výsledky testování.

2 Úvod do problematiky

2.1 Platforma Android

Android je open source operační systém, který vznikl pro mobilní zařízení s dotykovou obrazovkou. Vyvíjí ho skupina *Open Handset Alliance*, složená z výrobců zařízení, ze softwarových firem v čele s *Google Inc.* a telekomunikačních firem. Seznam všech členů a souhrn informací o Androidu je dostupný na webové stránce OHA [1]. Aplikace pro Android jsou psány v programovacím jazyce Java, případně v nativním kódu C/C++.

2.1.1 Architektura

Operační systém Android je složen z pěti hlavních vrstev zobrazených na obrázku **2.1** a popsaných na webové stránce Androidu pro vývojáře [2].

Nejspodnější vrstvou je linuxové jádro, nyní ve verzi 2.6, které zajišťuje systémové služby jako bezpečnost, správu paměti, správu procesů, síťové služby a ovladače hardwaru. Jádro slouží jako abstraktní vrstva mezi hardwarem a ostatními softwarovými vrstvami.

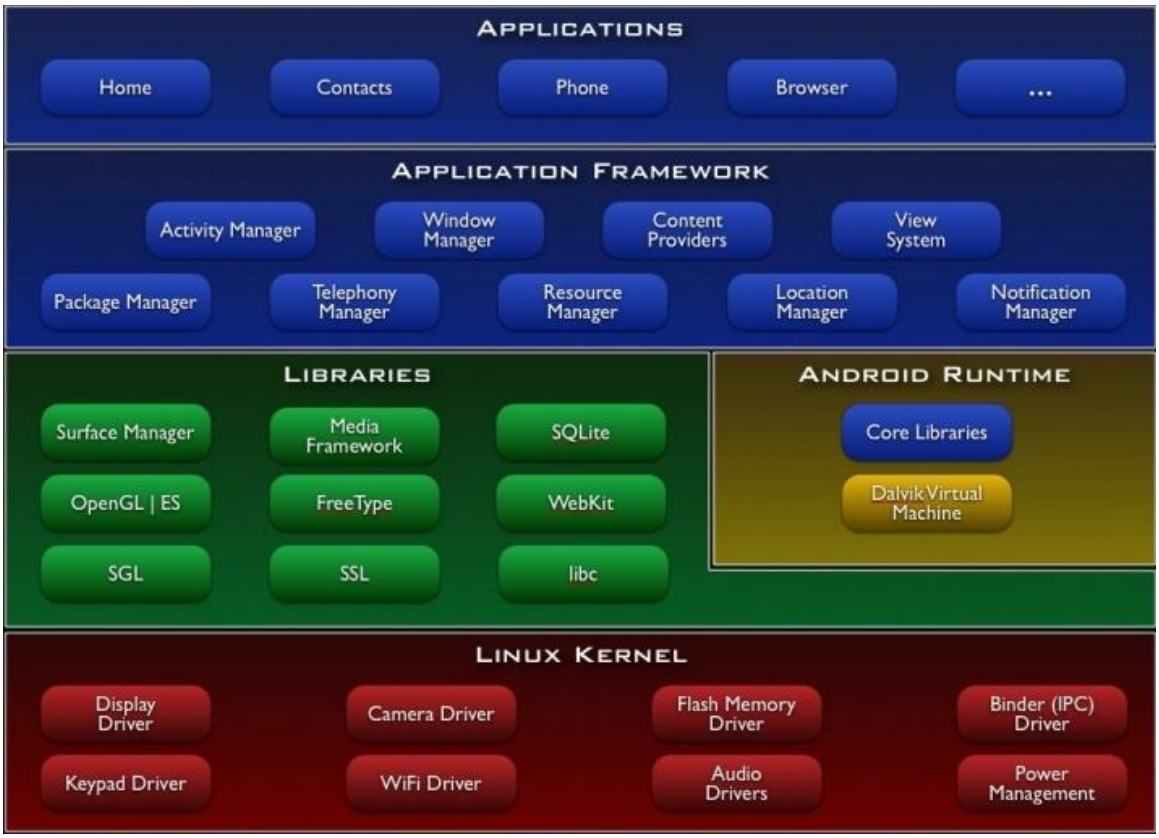
Android zahrnuje množství C/C++ knihoven, které využívají jeho různé komponenty. Vývojáři k nim mají přístup pomocí *Android Application Framework*. Nejvýznamnějšími knihovnami jsou: *System C Library* – implementující standardní C knihovny upravené pro zařízení s Linuxem, *Media Librariees* – zajišťující přehrávání a nahrávání velkého množství audio a video formátů i práci s obrázky, *Surface Manager*, *LibWebCore* atd.

Další vrstvou je *Android Runtime*, která je složena z *Dalvik* virtuálního stroje (angl. *Dalvik Vitrutal Machine* – DVM) a základních knihoven, zajišťujících funkcionalitu knihoven programovacího jazyku Java. Každá aplikace na Androidu běží ve vlastním procesu ve vlastním DVM, založeném na registrově orientované architektuře. Virtuální stroj je napsán tak, aby jich zařízení mohla efektivně pouštět více. DVM využívá funkce linuxového jádra jako vlákna procesů a správu paměti.

Díky otevřené platformě Android nabízí vývojářům využití senzorů zařízení, přístup k poloze, běh služeb na pozadí, atd. Pomocí vrstvy *Application Framework* mají vývojáři plný přístup ke službám, stejně jako základní aplikace operačního systému. Architektura aplikací je navržena tak, aby zjednodušovala znovupoužití různých komponent. Každá aplikace může nabídnout své služby, nebo použít služby jiných aplikací. Bezpečnost využívání služeb zajišťuje samotný *Framework*.

V podstatě všechny aplikace mohou využívat tyto služby a systémy:

- Systém pro zobrazování *View*, obsahující tlačítka, textová pole a další grafické komponenty.



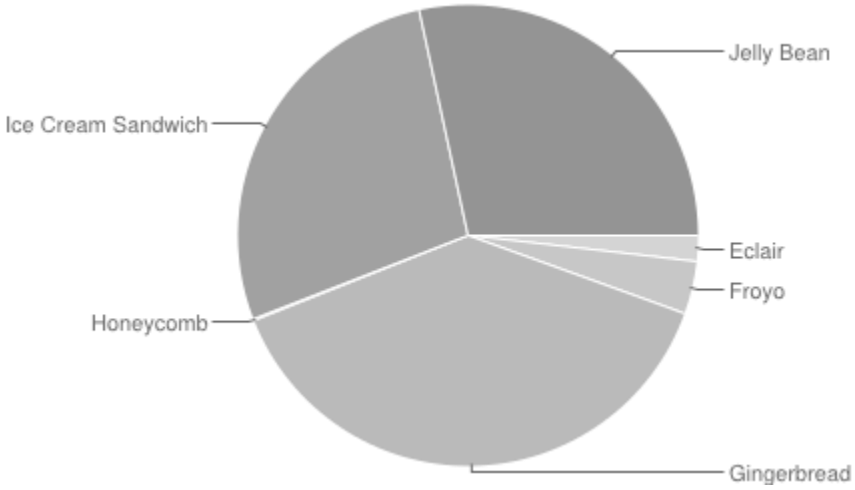
Obrázek 2.1: Architektura Androidu, převzato z [2].

- *Content Provider* zajišťující přístup k datům jiných aplikací a sdílení vlastních dat.
- *Resource Manager*, který umožňuje přístup ke zdrojům, jako jsou textové řetězce v různých jazycích, grafika a obrázky v odlišných rozlišeních a rozdílné styly vzhledu aplikace.
- *Notification Manager*, díky kterému je možné zobrazovat upozornění ve stavovém řádku.
- *Activity Manager*, který řídí životní cyklus aplikace a zajišťuje základní navigaci.

Poslední vrstvou jsou samotné aplikace. Android obsahuje základní aplikace pro kontakty, telefonování, SMS, klienta elektronické pošty, kalendář, mapy, internetový prohlížeč atd. Další aplikace třetích stran je možné pořídit na *Google Play Store* [3], což je internetový obchod poskytující aplikace placené i neplacené.

2.1.2 Zastoupení verzí Androidu na trhu

Podle oficiální webové stránky Androidu [4] již bylo prodáno a je aktivních přes 400 milionů zařízení s tímto operačním systémem a je to nejrozšířenější mobilní platforma. Zatím bylo vydáno 17 verzí tzv. *API levels* (*Application Programming Interface* – programové rozhraní aplikace), které identifikují revize *API Frameworku* a ty určují, jaké vestavěné funkce lze využívat. Některá API obsahují více verzí samotného Androidu s několika různými kódovými jmény.



Obrázek 2.2: Graf zastoupení verzí Androidu na trhu. Převzato z [2].

V sekci *Dashboards* na webové stránce Androidu pro vývojáře [2] je zobrazena statistika verzí Androidu, která je vytvářena shromažďováním informací ze zařízení, jenž byla za posledních 14 dní připojena na *Google Play Store*. Neuvedené verze mají nejmenší podíl, a to menší nebo roven 0,1%. Verze 2.1 *Eclair* (API7) má podíl 1,7%, verze 2.2 *Froyo* (API8) 4,0%, verze 4.1.x *Jelly Bean* (API16) 23% a k tomu nejnovější 4.2.x (API17) 2%, verze 4.0.x *Ice Cream Sandwich* (API15) 29,3% a stále nejrozšířenější jsou verze 2.3.3 – 2.3.7 *Gingerbread* (API10) s největším podílem 39,7%. Protože mám k dispozici mobilní telefon *LG Optimus One P500* s verzí 2.3.3, která je také stále nejrozšířenější, rozhodl jsem se pro podporu minimálně této verze a kompatibilitu s verzí 2.2.

2.1.3 Vývojářské nástroje, Android SDK

Android poskytuje sadu nástrojů pro Java integrované vývojové prostředí (angl. *Integrated Development Environment* – IDE) s pokročilými vlastnostmi pro vývoj, testování, ladění a vytváření balíků aplikací. Použitím IDE lze vyvíjet aplikace pro všechny dostupné Android zařízení nebo vytvořit virtuální zařízení, které emuluje jakoukoliv hardwarovou konfiguraci. Tyto nástroje jsou zdarma, open source a fungují na většině operačních systémů.

Plná podpora v Java IDE. Nabídka rychlých oprav kódu, zabudovaná navigace mezi Javou a XML zdroji (textové řetězce, základní popis aplikace, seznam povolení), rozšířené XML editory pro Android XML zdroje, analytické nástroje výkonu, použitelnosti a opravování problémů.

Návrh grafického rozhraní. Vytváření UI (*User Interface*) pomocí přetahování grafických komponent z nabídek, náhled možného zobrazení na mobilních telefonech, tabletech a dalších zařízeních, přepínání grafických témat a verzí systému bez kompilace, podpora editoru pro práci s vlastními uživatelskými UI komponentami.

Možnosti vývoje na zařízeních. Ladění přes USB kabel, zobrazení využití procesoru, zachytávání chybových zpráv. Jsou dva typy vývoje:

- Vývoj na hardwarovém zařízení – použití jakéhokoliv Android zařízení, instalace aplikací přímo z IDE, testování a ladění přímo na zařízení.
- Vývoj na virtuálním zařízení – použití uživatelských velikostí obrazovky, klávesnice a jiných hardwarových komponent, pokročilá emulace zahrnující kameru, senzory, multitouch (více dotyků na obrazovce zároveň), telefonování, účelem je testování pro vysokou kompatibilitu.

2.1.4 Android NDK

NDK je soubor nástrojů, které umožňují implementaci částí aplikace s použitím nativního kódu v programovacích jazycích C a C++. Výhodou je znovupoužití existujících knihoven napsaných v těchto jazycích. Použití NDK však neznamená vždy zvýšení výkonu aplikace. NDK by mělo být použito pro samostatné operace, které hodně zatěžují procesor a nealokují mnoho paměti.

NDK využívá JNI (*Java Native Interface*) [5], což je standardní programovací rozhraní pro psaní Java nativních metod a zabudování Java virtuálního stroje do nativních aplikací. Pomocí tohoto rozhraní je možné z Javy volat nativní funkce napsané v C/C++, případně asembleru.

Nativní metody jsou v Javě deklarovány pomocí klíčového slova `native`, např.:

```
native long initHolder(int width, int height);
```

Tato funkce musí být implementována v nativní sdílené knihovně, která musí být pojmenována podle unixových konvencí s předponou „`lib`“ a příponou „`.so`“, např. `libNative.so`. Aplikace musí explicitně načíst knihovnu, např. pro načtení na začátku běhu aplikace stačí přidat kód:

```
static {
   System.loadLibrary("Native");
}
```

Zde je však jméno knihovny uvedeno bez předpony a přípony.

Soubor `Android.mk`. Tento soubor slouží k popisu zdrojových C a C++ souborů pro NDK. Je to malý fragment *GNU Makefile* používaný při překladu a kompilaci. Umožňuje vytváření modulů ze zdrojových souborů a to statické knihovny a sdílené knihovny. Statické knihovny jsou používány jen ke generaci sdílených knihoven, které jsou instalovány do balíčku aplikace. Překladový systém sám řeší mnoho detailů, např. není nutné uvádět hlavičkové soubory nebo závislosti mezi generovanými soubory, NDK je doplní automaticky.

Syntaxe `Android.mk`. Soubor musí začínat definicí proměnné `LOCAL_PATH`, ta je použita pro zjištění umístění zdrojových souborů v hierarchii projektu. Lze využít makro funkci `my-dir` (`LOCAL_PATH := $(call my-dir)`) pro zjištění aktuálního adresáře (ve kterém je umístěn samotný `Android.mk`). Následuje `include $(CLEAR_VARS)` a slouží k vyčištění proměnných, protože překlad je proveden najednou a všechny proměnné jsou brány jako globální. Dále následuje proměnná pro definici jména modulu: `LOCAL_MODULE := Native` a zdrojových souborů:

`LOCAL_SRC_FILES := native.cpp`. Jsou zde uvedeny pouze C/C++ soubory, hlavičkové soubory ne. Nakonec je vybrán druh knihovny, buď sdílená `include $(BUILD_SHARED_LIBRARY)`, nebo statická `include $(BUILD_STATIC_LIBRARY)`.

Soubor `Application.mk`. Popisuje nativní moduly, které aplikace potřebuje. Některé základní proměnné:

- `APP_ABI` – standardně je generován kód pro *armebi*, které představuje zařízení s procesorem založeným na ARMv5 se softwarovými operacemi s plovoucí desetinnou čárkou. Pro podporu hardwarových operací u ARMv7 procesorů lze nastavit `APP_ABI := armeabi-v7a` a pro podporu více druhů zařízení je možné zadat najednou `APP_ABI := armeabi armeabi-v7a`, k dispozici je ještě instrukční sada IA-32 (parametr `x86`) a MIPS instrukce (`mips`), případně pro podporu všech najednou parametr `all`.
- `APP_PLATFORM` – jméno cílové platformy Androidu. Např. „`android-8`“ představuje Android 2.2 (API8).
- `APP_STL` – samotný systém poskytuje jen velmi malé množství standardních C++ knihoven, kde nejsou podporovány např. ani `std::string` nebo `std::vector`. Také nejsou podporovány výjimky. NDK však poskytuje několik pomocných knihoven: *GAbi++*, *STLport* a *GNU STL*, které tyto vlastnosti podporují. Knihovny jsou ve statických i sdílených verzích. Nastavením `APP_STL` je zařízeno použití některé z těchto knihoven místo systémové.
- `APP_CPPFLAGS` – podpora výjimek je standardně vypnutá kvůli zpětné kompatibilitě skriptů, proto je nutné explicitně nastavit parametr `-fexceptions`.

Samotný překlad a kompilace zdrojových souborů je prováděn pomocí *shell* skriptu `ndk-build` na unixových operačních systémech, nebo `ndk-build.cmd` na operačních systémech Windows. Skript je umístěn v hlavním adresáři NDK a měl by být spouštěn z adresáře projektu aplikace.

Výhody použití C++. Nativní funkce mají jako první parametr `JNIEnv *env`, což je ukazatel na prostředí Javy a tabulky funkcí. Při použití jazyka C jsou funkce volány např.:

```
cArray = (*env)->GetByteArrayElements(env, javaByteArray, 0);
```

ale při použití C++ je práce usnadněna:

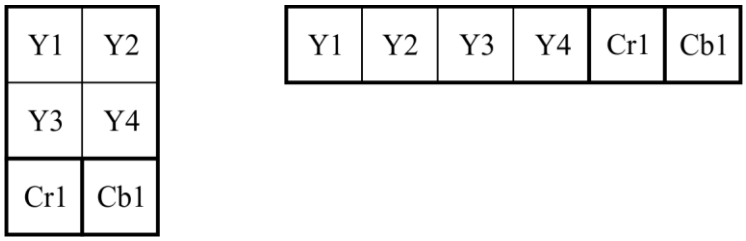
```
cArray = env->GetByteArrayElements(javaByteArray, 0);
```

již není nutné předávat `env` jako parametr a přístup k funkcím JNI je jednodušší. Další velkou výhodou je možnost použití výjimek.

2.1.5 YCbCr formát obrazu

Systém Android standardně používá pro obrázky náhledu z kamery formát YCbCr s kódováním NV21, které mění pořadí Cb a Cr složek. Y značí jasovou složku, Cb modrou barevnou složku a Cr červenou složku obrazu. Popis formátu je na webových stránkách video kodeku FOURCC [6]. V tomto barevném modelu jsou v paměti za sebou uloženy nejprve všechny pixely s informacemi o jasu, pak o červené barvě a nakonec o modré, zobrazeno na obrázku **2.3**.

Jeden pixel je uložen na osmi bitech. Počet pixelů jasové složky je roven velikosti obrázku – výška krát šířka, pak následují barevné složky, každá s čtvrtinovým počtem pixelů. Barvy jsou podvzorkované kvůli komprimaci, jeden pixel barevné složky obsahuje informaci o barvě pro blok 2 krát 2, tedy 4 pixely s jasem.



Obrázek 2.3: Formát YCbCr s kódováním NV21. Zleva: Zobrazení obrázku, uložení v paměti.

2.2 OpenCV knihovna

OpenCV [7] je zkratkou pro *Open Source Computer Vision* – open source knihovna počítačového vidění a softwaru pro strojové učení. Je vydáváno pod BSD licencí a je zdarma k dispozici pro akademické i komerční účely. Má rozhraní v C, C++, Pythonu a nyní i Javě a podporuje Windows, Linux, Mac OS, iOS a Android. OpenCV bylo navrženo pro výpočetní efektivitu a se silným zaměřením na aplikace v reálném čase a poskytnutí infrastruktury pro aplikace počítačového vidění. Je nativně napsáno v C++ a má šablonové rozhraní, které hladce pracuje s STL (*Standard Template Library*) kontejnery a může využít výhod vícejádrových procesorů.

Knihovna obsahuje více než 2500 optimalizovaných algoritmů, které zahrnují obsáhlý soubor klasických algoritmů počítačového vidění a strojového učení. Tyto algoritmy mohou být použity pro detekci a rozpoznání obličejů, identifikace objektů, klasifikace lidské činnosti ve videích, sledování pohybů kamery nebo objektů, extrakci 3D modelů objektů, vytváření 3D bodů ze stereo kamer, spojování obrázků pro vytvoření obrazu celé scény s vysokým rozlišením, vyhledání podobných obrázků z databáze, odstranění efektu červených očí po použití blesku, následování pohybu očí atd.

Třída `Mat`. Základní třída OpenCV, reprezentující n-dimenzionální pole. Lze ji využít k uložení barevných obrázků, vektorů, matic, histogramů atd. a pracuje s ní většina OpenCV funkcí.

Obsahuje dvě části. Hlavičku, která obsahuje informace o rozměrech matice, datový typ uložených dat a adresu uložení matice, počet referencí na matici atd. Druhou částí jsou uložená data, nebo ukazatel na ně. Objekt `Mat` lze vytvořit i nad již existujícími daty (pole, pixely obázku), při tom je vytvořena pouze hlavička a data nejsou kopírována. `Mat` automaticky alokuje i uvolňuje paměť, není nutné to provádět manuálně. Přitom je hlídán počet referencí na matici, při vytvoření dalšího `Mat` objektu nad stejnými daty je počet referencí zvýšen, při zrušení objektu je počet snížen. Při dosažení nulového počtu referencí poslední objekt využívající data je uvolní, pokud již data neexistovala při vytvoření matice.

2.2.1 OpenCV pro Android

OpenCV lze na Android platformě používat jak v jazyku Java, tak i v nativním kódu C++.

Použití v Javě. OpenCV využívá aplikaci *Android OpenCV Manager*, která poskytuje ostatním aplikacím nejnovější verze knihoven OpenCV. Doporučeným postupem vývoje aplikace je s *Async Initialization* (asynchronní inicializace), která používá manažer k přístupu k externě uloženým knihovnám. Inicializace je v kódu spuštěna pomocí příkazu:

```
OpenCVLoader.initAsync(OpenCVLoader.OPENCV_VERSION_2_4_5,
   this, mLoaderCallback);
```

Instance třídy `BaseLoaderCallback - mLoaderCallback` musí implementovat metodu `OnManagerConnected()`, která je vyvolána po dokončení inicializace pro obsloužení výsledku. Funkce OpenCV knihovny mohou být volány až po vyvolání obsluhy.

Použití v nativním kódu. Do souboru `Android.mk`, popsaného v kapitole 2.1.4, je nutné přidat cestu k *OpenCV makefile*:

```
include <Cesta_k_OpenCV>\<verze_OpenCV>-android-sdk
   \sdk\native\jni\OpenCV.mk
```

za řádek obsahující `include $(CLEAR_VARS)`. Ještě před vložením cesty k `OpenCV.mk` je možné nastavit několik proměnných:

- `OPENCV_INSTALL_MODULES` – nastavením na `on` jsou zkopírovány dynamické knihovny do adresáře *libs* v projektu, aby byly instalovány do balíku aplikace.
- `OPENCV_CAMERA_MODULES` – nastavením na `off` je vypnuto kopírování OpenCV nativních knihoven souvisejících s kamerou.
- `OPENCV_LIB_TYPE` – standardně je použito dynamické linkování a projekt pak závisí na knihovně `libopencv_java.so`. Pro použití statického linkování lze nastavit tuto proměnnou na `STATIC`.

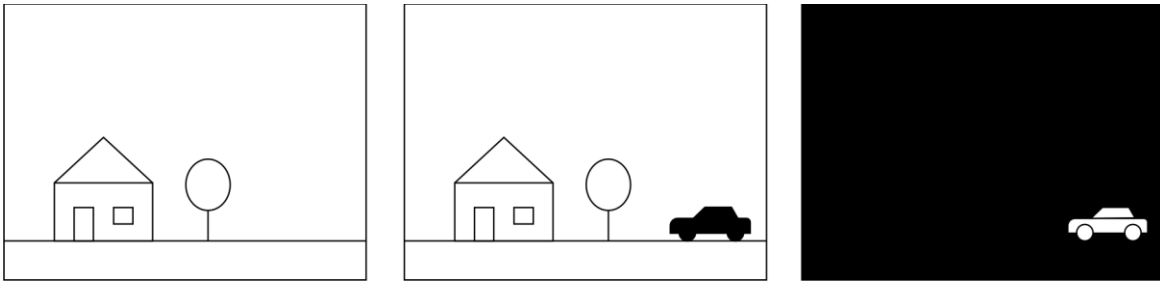
Pokud je v OpenCV použit nejen jazyk C, ale i C++ a výjimky, měly by být nastaveny také proměnné `APP_STL` a `APP_CPPFLAGS` v souboru `Application.mk`.

2.3 Možnosti detekce objektů

Uvedu zde přehled základních metod detekce objektů ve videu, se kterými jsem se seznámil a případně vyzkoušel použít.

2.3.1 Odečítání pozadí

Jak popisuje Piccardi ve svém souhrnu technik pro tuto metodu [8], jedná se o velmi rozšířenou metodu detekce pohybujících se objektů ve videu. Základním principem je výpočet rozdílu mezi retenčním obrázkem (pozadím) a aktuálním snímkem. Pozadí musí být scéna bez pohybujících se objektů a musí být aktualizováno kvůli změnám osvětlení. Pokud se do scény dostane objekt, rozdíl pixelů obrázků není nulový, a tím je objekt detekován (znázorněno na obrázku **2.4**). Také je možné nastavit hranici velikosti rozdílu pro detekci. Tato metoda však vyžaduje statickou kameru, proto jsem ji nemohl použít pro detekci v jedoucím vozidle.

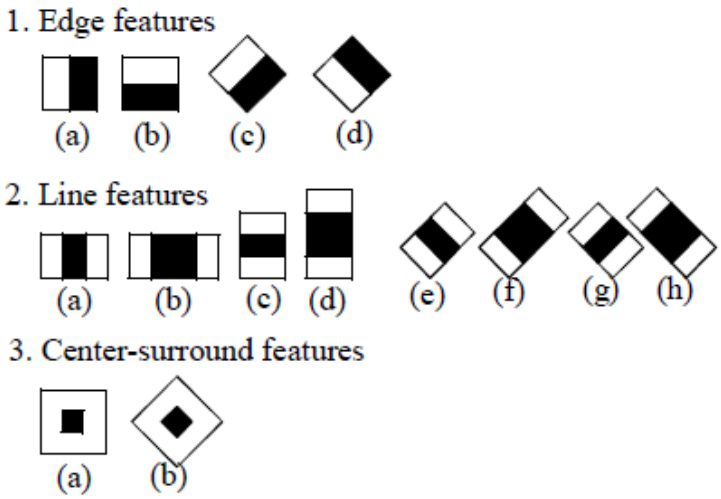


Obrázek 2.4: Odečítání pozadí. Zleva: referenční pozadí, aktuální snímek s pohybujícím se objektem, rozdíl obrázků.

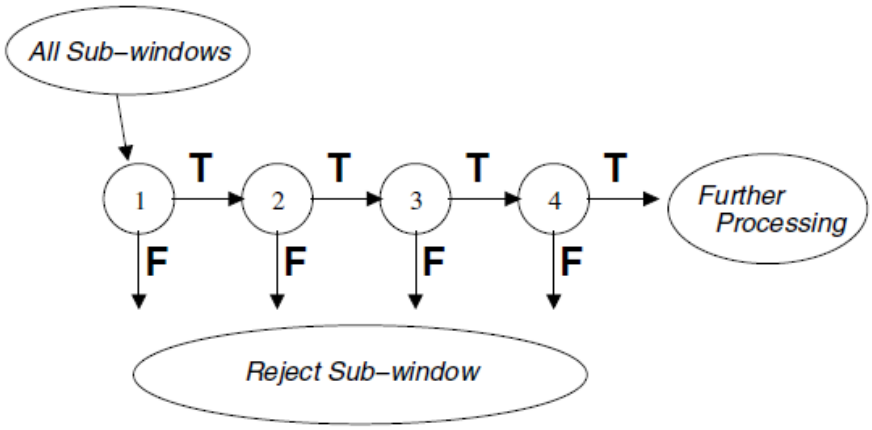
2.3.2 Kaskádový klasifikátor

Poprvé popsaný Violem a Jonesem [9]. Princip spočívá v natrénování klasifikátoru několika stovkami obrázků hledaného objektu, nazývaných pozitivní vzorky, které jsou upravené do stejné velikosti a negativními vzorky – libovolné obrázky stejné velikosti. Klasifikátor si vytváří sadu příznaků, které popisují vlastnost oblasti, např. rozdíl intenzity světla mezi očima a tváří. Klasifikátor využívá jen tyto příznaky z obrázku **2.5**: příznaky hrany (1a) a (1b) a příznaky čáry (2a) a (2c). Klasifikátor s více příznaky bude mít vyšší počet správných detekcí a menší počet falešných detekcí, ale bude potřebovat více výpočetního času.

Po natrénování je klasifikátor použit na část obrázku (stejně velkou jako během trénování). Výstupem klasifikátoru je logická jednička, pokud je možné, že část obrázku objekt obsahuje, jinak logická nula. Pro vyhledání objektu v celém obrázku je posunováno hledací okno. Klasifikátor je navržen tak, aby bylo možné jednoduše měnit jeho velikost a tím vyhledávat objekty zájmu v různých



Obrázek 2.5: Prototypy příznaků. Převzato z [10].



Obrázek 2.6: Schéma detekční kaskády. Převzato z [9].

velikostech. Toto je více efektivní než měnit velikost obrázku. Pro vyhledání objektu v neznámé velikosti je potřeba prohledat obrázek několikrát s klasifikátorem ve více měřítcích.

Kaskádový znamená, že klasifikátor je složen z více jednodušších, které jsou postupně aplikovány na část obrázku až do doby, kdy je oblast zamítnuta, nebo jsou všechny stupně kaskády splněny. Schéma kaskády je zobrazeno na obrázku **2.6**. Toto rozdělení do kaskády umožňuje rychle vyřadit pozadí obrázku a využít větší část výpočetního času na oblast, kde je možný výskyt objektů.

Lienhart a Maydt ve své práci [10] přidali sadu příznaků čar (2b) a (2d), příznak obklopeného středu (3a) a představili sadu otočených příznaků, zobrazených na obrázku **2.5**. Tímto zmenšili počet falešných detekcí o průměrně deset procent.

Tato metoda detekce je hojně používána k detekci obličejů, protože lidské obličeje mají stejné proporce a nejsou velmi odlišné. Bývá používána i pro detekci automobilů, ale u nich je problém

v tom, že automobily mají různé tvary, velikosti, pozice zadních světel a státní poznávací značky apod.

2.3.3 Porovnání klíčových bodů ve snímcích videa

Principem této metody je vyhledání klíčových bodů a jejich deskriptorů v referenčním obrázku s objektem, případně označení sledovaného objektu přímo ve videu nebo na kameře a následné hledání stejných klíčových bodů v dalších snímcích videa. Nevýhodou této metody je nutnost předem znát hledaný objekt a označit ho. To ovšem není vhodné pro uživatele v automobilu, který by musel za jízdy označovat vozidla před ním. Přesto jsem tuto metodu použil jako základ mého systému pro rozpoznávání, ale dále jsem ji upravil a rozšířil, aby nebylo nutné objekty označovat předem.

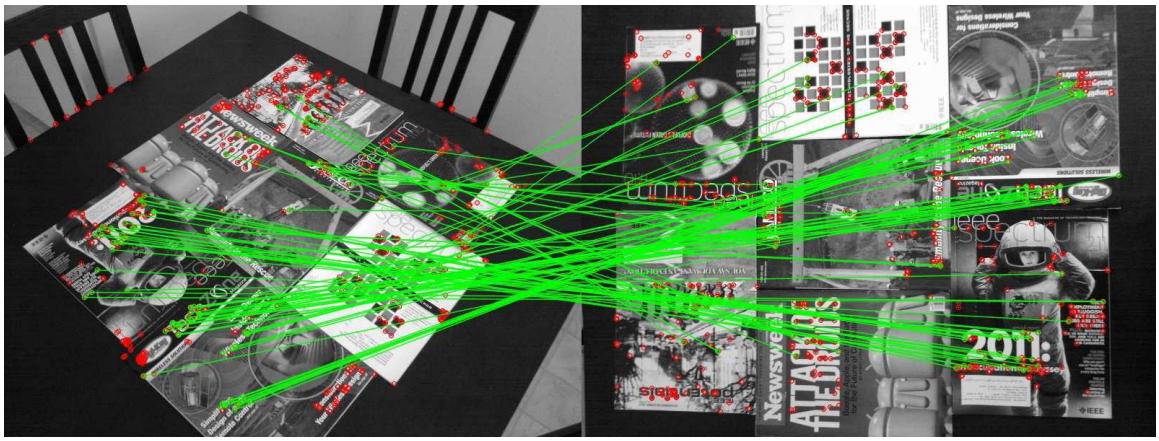
K získání klíčových bodů je možné využít několik různých tzv. detektorů a deskriptorů příznaků. Rozšířené detektory jsou SIFT (*Scale-invariant feature transform*) [11] a SURF (Speeded *Up Robust Features*) [12], ty jsou však chráněny patentem a v OpenCV byly přesunuty do tzv. *nonfree* modulu. Ten jednak není v OpenCV pro Android vůbec obsažen a ještě by bylo nutné akceptovat licenční smlouvu. Další možností je použití detektoru ORB, popsaného v kapitole 2.4, který je i implementován v OpenCV. Ukázka použití ORB k porovnání dvou snímků je na obrázku **2.7**.

2.4 ORB – alternativa k SIFT nebo SURF

ORB je detektor klíčových bodů a binární deskriptor. Jak píše Rublee [13], ORB má být schopný efektivně nahradit SIFT, který má podobný výkon, avšak ORB je méně náchylný na ruchy v obraze a je možné ho využít v aplikacích reálného času nebo na mobilních zařízeních nebo nízko výkonných zařízeních bez grafické akcelerace. Při porovnávání obrázků a detekci objektů si vede stejně dobře jako SIFT a je lepší než SURF, zatímco je o skoro dva řády rychlejší.

Základem je FAST detektor klíčových bodů a BRIEF deskriptor, odtud zkratka ORB (*Oriented FAST and Rotated BRIEF*). Obě techniky mají dobrou výkonnost a zároveň nízké požadavky na výpočetní výkon. Další výhodou je, že ORB nepodléhá žádné licenci.

Klíčové body. Metodou pro hledání klíčových bodů v reálném čase je FAST. Je efektivní a hledá rozumné rohové body, přesto musí být rozšířen o Harrisův rohový filtr, kvůli odstranění hran. Mnoho detektorů klíčových bodů obsahuje operátor orientace, ale FAST ne. Je několik možných způsobů popisu orientace bodu, mnoho jich zahrnuje výpočet histogramu gradientu (např. u SIFT) nebo aproximace blokových vzorů u SURF. Tyto metody jsou buď výpočetně náročné, nebo v případě SURF jde o slabou aproximaci. ORB využívá tzv. těžiště intenzity. Na rozdíl od operátoru orientace u SIFT, který může mít více hodnot pro jediný klíčový bod, těžišťový operátor dává jeden dominantní výsledek.



Obrázek 2.7: Typický výsledek porovnání obrázků pomocí ORB. Zelené čáry značí nalezené shody, červené tečky značí klíčové body bez nalezených shod. Převzato z [13].

Deskriptory. BRIEF je nový deskriptor příznaků, který používá jednoduché binární testy mezi pixely vyhlazeného políčka obrázku. Jeho výkony jsou podobné deskriptoru SIFT v mnoha ohledech, včetně odolnosti vůči změně osvětlení, rozmazání a pokřivení perspektivy. Avšak je velmi citlivý na rotaci. BRIEF vznikl z výzkumu, který používá binární testy k natrénování sady klasifikačních stromů. Po natrénování na množině 500 apod. typických klíčových bodů, stromy mohou být použity pro získání popisu jakéhokoliv klíčového bodu.

2.4.1 Orientace klíčových bodů pro FAST

FAST detektor. Nejprve jsou detekovány FAST body v obraze. Parametrem pro funkci FAST je intenzita prahu mezi centrálním pixelem a pixely v kruhu kolem středu. Rublee použil FAST-9 (poloměr 9) [13], který má dobrou výkonnost.

Orientace pomocí těžiště intenzity (angl. *Intensity Centroid*). Metoda těžiště intenzity předpokládá, že intenzita rohu je posun od jeho středu a tento vektor může být použit k výpočtu orientace. Momenty políčka obrázku s rohem jsou definovány jako:

$$m_{pq} = \sum_{x,y} x^p y^q I(x,y) \tag{1}$$

a s pomocí těchto momentů lze najít těžiště:

$$C = \left(\frac{m_{10}}{m_{00}}, \frac{m_{01}}{m_{00}}\right) \tag{2}$$

Lze sestrojit vektor ze středu rohu O do těžiště C, $\overrightarrow{OC}$, orientace políčka pak je:

$$\theta = \tan^{-1}\left(\frac{m_{01}}{m_{10}}\right) \tag{3}$$

Pro zlepšení nezávislosti na rotaci, jsou momenty počítány s x a y, které zůstávají v kruhu o poloměru r, ten je nastaven na velikost políčka, takže x a y jsou v rozmezí $[-r, r]$. Jak se $|C|$ blíží nule, rozsah se stává nestabilním, ale s FAST rohy jen velmi vzácně.

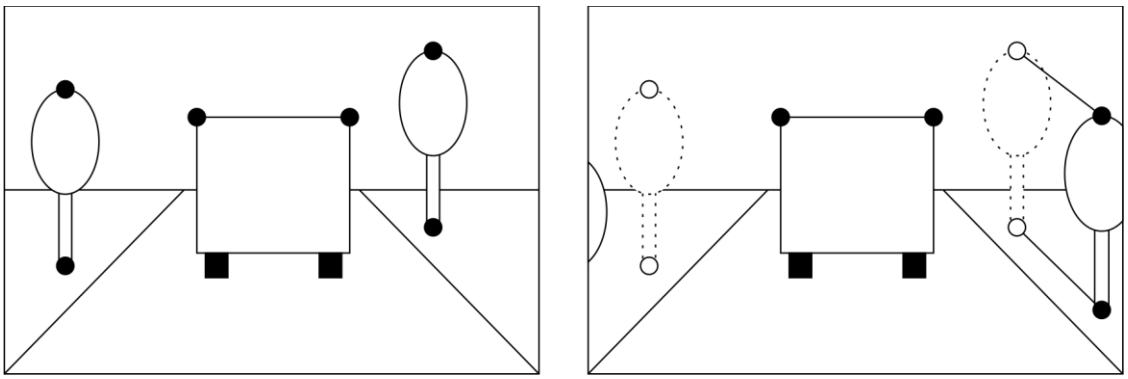
3 Návrh systému detekce

Rozhodl jsem se navrhnout systém rozpoznávající automobily jedoucí před vozidlem s mobilním telefonem s kamerou, který nepotřebuje natrénovat na vzorku dat obrázků automobilů a ani není pro uživatele aplikace nutné označovat vozidla před ním. Zároveň jsem chtěl dosáhnout toho, aby byl tento systém univerzální pro rozpoznání osobních a nákladních automobilů i různých druhů přívěsů, případně jiných překážek na silnici.

3.1 Detekce objektů a dodržení bezpečné vzdálenosti

3.1.1 Detekce objektů

Detekce je založena na porovnávání po sobě jdoucích snímků z kamery mobilního telefonu, které je popsáno v kapitole 2.3.3 a s využitím ORB detektoru a deskriptoru popsaného v kapitole 2.4. Základním předpokladem tohoto principu v ideálních podmínkách je, že všechny různé objekty kolem silnice se před kamerou míhají, avšak automobil při konstantní rychlosti zůstává vůči kameře statický, jak je zobrazeno na obrázku **3.1**. Černé tečky znázorňují nalezené klíčové body, v pravém obrázku jsou také vyneseny vektory posunu těchto bodů. V reálných podmínkách však dochází k otřesům vozidla nebo změnám směru jízdy apod.



Obrázek 3.1: Posun objektů ve snímcích kamery.

Maska pro vyhledávání. Objekty v pozadí na horizontu ovšem také zůstávají statické, a tak je nelze rozeznat od statických vozidel. Proto je vhodné vytvořit masku a jen v ní hledat klíčové body. Jejím ideálním tvarem je tvar silnice nebo jízdního pruhu v obraze. Maska je přednastavena, ale uživateli je umožněno masku změnit podle umístění mobilního telefonu v automobilu a perspektivy

výhledu. Tak je jednak zamezeno detekci objektů nad horizontem, ale i objektů kolem silnice a tím je detekce automobilu jednodušší a přesnější.

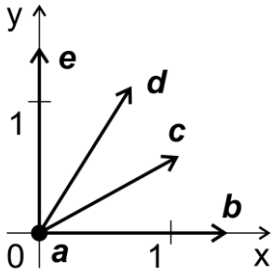
Příprava shodných deskriptorů. Nejprve jsou detekovány klíčové body v prvním snímku a vypočítány jejich deskriptory. To samé je provedeno pro další snímek. Pak jsou porovnány deskriptory a jsou hledány shodné, které se vyskytují v obou snímcích. Z nich je vytvořen seznam *Shod* deskriptorů, které se odkazují na deskriptory z prvního a druhého obrázku a obsahují vzdálenost těchto deskriptorů.

3.1.2 Filtrace otřesů kamery a rychle se pohybujících objektů

Pokud se vozidlo v obraze pohne (otřesem kamery nebo mírným zatočením), všechny jeho klíčové body se posunou stejným směrem, takže vektory posunu mají stejnou velikost a směr. Rychle se pohybující objekty v obraze (stromy v okolí silnice, přechody pro chodce) mají v poměru k otřesům velký vektor posunu mezi snímky. Čím blíže kameře objekty jsou, tím větší je posun.

Pro filtrování je vytvořen seznam vektorů posunu klíčových bodů z druhého snímku oproti prvnímu a zároveň jsou sčítány velikosti těchto vektorů. Ze sečtených velikostí a velikosti seznamu je vypočítána průměrná velikost vektoru. Ze seznamu jsou pak odstraněny všechny vektory s větší velikostí než průměrnou a jsou odstraněny i *Shody* odkazující se na klíčové body s těmito vektory posunu. Tím jsou vyfiltrovány objekty, které se v obraze pohnuly rychleji, než vozidlo před kamerou.

Další částí filtrace je roztřídění vektorů posunů do množin podle podobnosti jejich velikostí a směrů. Podobnost je podmíněna rozdílem velikostí dvou vektorů s hodnotou maximálně $\varepsilon = 0{,}2$ (dvacet procent jednoho pixelu). Shoda směrů je porovnávána v rámci 2D kartézského souřadnicového systému (obrázek **3.2**). Je dělena podle nulové velikosti ve směru osy *x* a zároveň osy *y* (vektor ***a***), nulové velikosti jen ve směru jedné osy a nenulové v druhém směru (vektory ***b*** a ***e***), nebo nenulové v obou směrech (***c*** a ***d***). Tyto případy jsou rozlišeny pro všechny čtyři kvadranty.



Obrázek 3.2: Rozdělení vektorů podle směru.

Takto jsou vektory přiděleny do jednotlivých množin podobných vlastností. Pak je deset procent z celkového počtu vektorů použito jako minimální hranice počtu vektorů, jakou musí množina obsahovat. Pokud obsahuje méně, jsou odstraněny *Shody*, které se odkazovaly na klíčové body, z kterých byly tyto vektory vypočítány. Tak jsou eliminovány klíčové body, které se vyskytují

v obraze samostatně a s malým počtem podobných a jsou uchovány jen hlavní shluky bodů. Např. pokud se vozidlo pohne doleva, všechny vektory z klíčových bodů na něm budou mít také směr doleva, avšak několik klíčových bodů z objektu u silnice bude mít směr doprava dolů. Tyto body budou odstraněny a zůstanou jen body automobilu.

Po tomto filtrování by měly zůstat jen klíčové body automobilu jedoucího před kamerou a tím by měl být detekován.

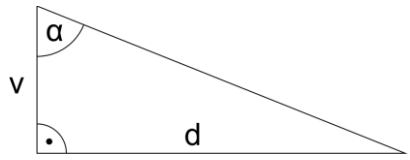
3.1.3 Dodržení bezpečné vzdálenosti

ORB detektor nachází klíčové body na celém automobilu před kamerou, ale ne vždy i na okrajích jeho siluety, proto nelze z obrysu klíčových bodů získat rozměry vozidla v pixelech a v poměru ke kameře dopočítat rozměry v metrech.

Nelze použít ani trigonometrické funkce. Jak zobrazuje obrázek **3.3**, k výpočtu vzdálenosti *d* je nutné znát výšku *v* a úhel α. Pak je možné *d* vypočítat jako:

$$d = v \tan \propto \tag{4}$$

Uživatel aplikace by však musel výšku *v* zadat ručně. Platforma Android sice poskytuje senzory, ze kterých lze zjistit úhel natočení mobilního telefonu, ale bez natáčení telefonu za jízdy na snímané vozidlo by nebyl úhel aktuální a výpočet vzdálenosti přesný. Navíc silnice může být také skloněna pod určitým úhlem. To nelze eliminovat ani uložením počátečního natočení při spuštění aplikace, kvůli možnosti náklonu už při spuštění.



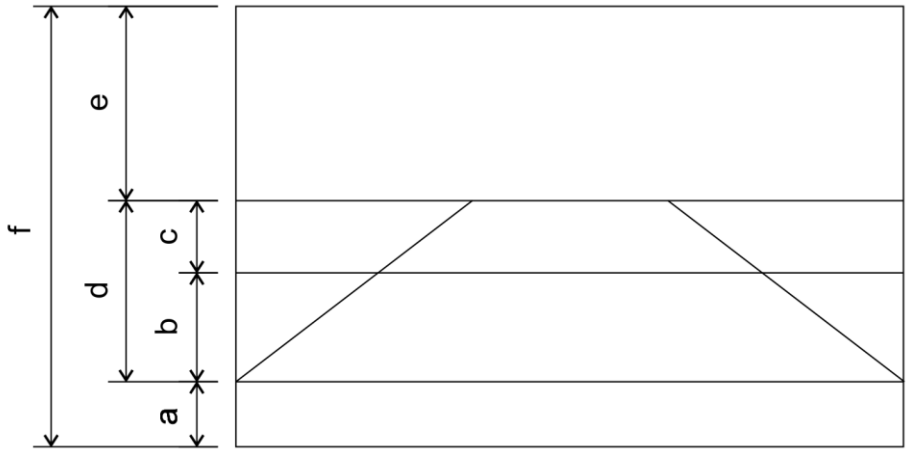
Obrázek 3.3: Výpočet vzdálenosti pomocí trojúhelníku.

Bez možnosti přesně spočítat vzdálenost k vozidlu, jsem se rozhodl experimentálně určit pevnou hranici bezpečné vzdálenosti na základě zkušebních videí s jízdou za několika automobily. Proto se ani nemění hranice v závislosti na rychlosti. Vzdálenost je tedy určena podle velikosti hranice horizontu silnice *e* zobrazené na obrázku **3.4** a začátku viditelné silnice *a*, která je ovlivněna zasahováním kapoty motoru automobilu do obrazu nebo výhledem z kabiny řidiče. Z celkové výšky obrázku *f* lze vypočítat rozměr viditelné silnice *d* jako:

$$d = f - e - a \tag{5}$$

a hranici bezpečné vzdálenosti *c* od horizontu *e*:

$$c = \frac{d}{3} \tag{6}$$



Obrázek 3.4: Rozměry silnice v obraze.

t - 2	*t - 1*	*t*	*f*
0	0	0	0
0	0	1	0
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1

Tabulka 3.1: Zpracování detekcí v čase.

nebo hranici bezpečné vzdálenosti b od hranice viditelnosti a:

$$b = d - \frac{d}{3} \tag{7}$$

Pokud se více jak dvacet procent klíčových bodů nachází za touto hranicí, dochází k detekci nedodržení bezpečné vzdálenosti. Procentuální mez je stanovena kvůli odfiltrování šumu v obraze a eliminaci jeho klíčových bodů.

Další součástí je filtrace v čase. Výsledné vyhodnocení nebezpečí f je určeno z nového – současného stavu a dvou předchozích, jak zobrazuje tabulka 3.1. Např. pokud v čase $t-2$ nedošlo k detekci (logická nula), ale v čase $t-1$ ano (logická jednička) a v čase t opět nedošlo k detekci, logická jednička je interpretována jako náhoda a výsledkem je logická nula a systém nehlásí nebezpečí. Tato technika také eliminuje náhodné detekování šumu v obraze jako nebezpečí.

3.2 Návrh aplikace

Vývojový diagram aplikace je znázorněn na obrázku **3.5**. Okno aplikace s grafickými prvky běží v hlavním vlákně. Dále jsou inicializovány potřebné struktury na uchovávání dat pro porovnávání snímků při detekci, kamera a zobrazovací komponenty a je spuštěno druhé vlákno na zpracování obrazu. Pak je započato zpracování příchozích snímků z kamery a v druhém vlákně probíhá detekce a nakonec jsou zobrazeny výsledky zpracování. Potom je načten další vstup z kamery pro zpracování.

3.2.1 Hlavní okno aplikace

Hlavní okno poskytuje rozhraní pro interakci s uživatelem a zajišťuje obsluhu životního cyklu aplikace. Jsou v něm obsaženy všechny další grafické komponenty. Také je v něm zajištěno zpracování dat o poloze a rychlosti jízdy a její vypsání. Data se snaží získat od nejlepšího možného dostupného zdroje: GPS satelitů, poloha od operátora mobilní sítě z vysílačů nebo internetového připojení.

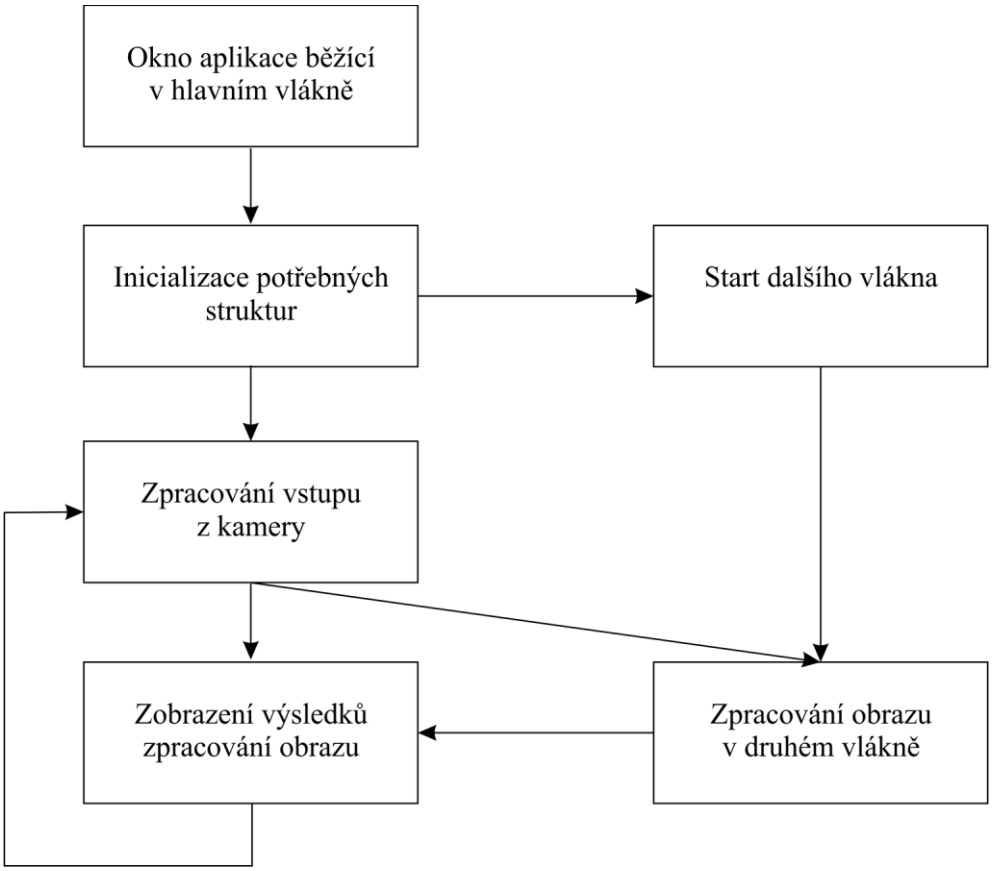
Okno obsahuje menu, jehož položky jsou *Nastavení* a *Demo*. *Nastavení* umožňuje nakreslení horizontu silnice na displeji mobilního telefonu a průsečíků, kde se levý a pravý okraj silnice (jízdního pruhu) protínají s horizontem. To slouží k nastavení uživatelské masky silnice. Při nízkých rychlostech vozidla je vypnuto zvukové varování před nedodržením bezpečné vzdálenosti, kvůli zastavování na křižovatkách, parkování apod. Pro možnosti testování lze zapnout *Demo* mód, který zapne varování i při nulové rychlosti.

3.2.2 Inicializace

Nejprve je inicializována kamera. Jsou zjištěny její parametry a vybrány ty podporované, které lze nastavit. Hlavní je určit rozlišení náhledu (angl. *Preview*), v jakém bude kamera snímat a dodávat snímky a uložit si poměr stran náhledu a nastavit i zobrazování na displeji na stejný poměr stran, aby nedocházelo k deformaci obrazu.

Pak je nutné připravit metodu, která bude volána při získání snímku (zároveň s jeho zobrazením na displeji). Při jejím prvním volání je v ní připravena grafická komponenta, která bude zobrazovat výsledky zpracování obrazu. Jsou jí nastaveny rozměry podle rozlišení náhledu a je vytvořen bitmapový obrázek pro vykreslení detekcí vozidel. Dále je inicializována struktura, ve které jsou uloženy informace o klíčových bodech a deskriptorech z vždy předchozího snímku, aby nebylo nutné ukládat aktuální snímek a znovu provádět jejich výpočet, ale aby byly vypočítány jen pro nový snímek. Také je v ní spuštěno druhé vlákno, ve kterém budou snímky zpracovávány. Při dalších voláních je předán snímek z kamery na zpracování, pokud již bylo dokončeno zpracování předchozího snímku.

Nakonec je spustěno samotné získávání snímků náhledu kamery.



Obrázek 3.5: Vývojový diagram aplikace.

3.2.3 Zpracování obrazu

Zpracování snímků z kamery probíhá v druhém vlákně, aby byl zajištěn plynulý chod uživatelského rozhraní, které běží v hlavním vlákně. Obraz je zpracován tak, že snímek a bitmapový obrázek pro výsledek jsou předány funkci, která provede detekci podle kapitoly 3.1. Dále vykreslí nalezené klíčové body a jejich obrys do připravené bitmapy a zkopíruje klíčové body a deskriptory aktuálního snímku kvůli uchování pro další detekci. Výsledek detekce dodržení bezpečné vzdálenosti (logická jednička nebo nula) je poslán jako zpráva do hlavního vlákna. Ještě je nastaven příznak dokončení zpracování snímku, aby bylo možné zpracovat další.

Pokud je aplikace v režimu *Nastavení*, vlákno pro detekci je uspáno, kvůli ušetření výpočetních prostředků procesoru.

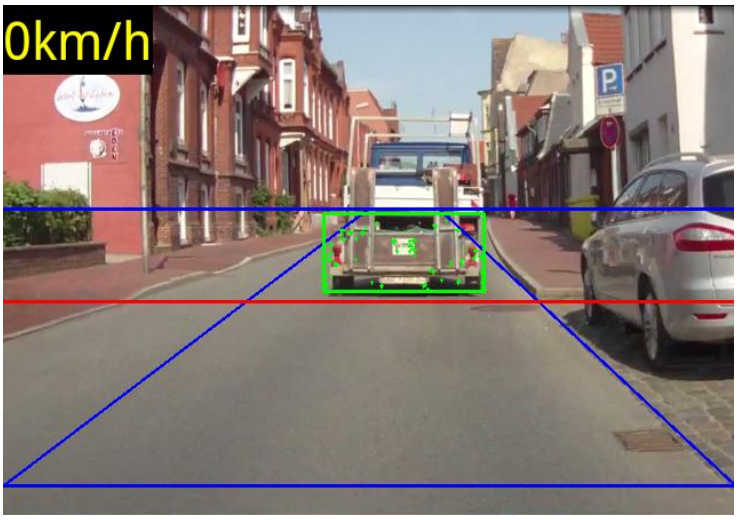
3.2.4 Zobrazení výsledků

Připravená grafická komponenta na zobrazení obrázků v hlavním vlákně čeká na zprávu z druhého vlákna o dokončení operace. Při příchodu zprávy si vyžádá překreslení s novou bitmapou. Pokud zpráva obsahuje příznak detekce nebezpečí, spustí varovný zvukový signál.

Při překreslení je zobrazena upravená bitmapa, a pak je vyčištěna a připravena pro další použití a kreslení do ní, nebo v režimu *Nastavení* jsou zobrazeny čáry, které uživatel kreslí na displeji při změně masky.

4 Implementace a testování aplikace

Obsluha aplikace, snímání obrazu a jeho vykreslení je implementováno v programovacím jazyku Java. Detekce automobilů je implementována v nativním kódu C++.



Obrázek 4.1: Ukázka běžící aplikace při detekci přívěsu.

4.1 Část aplikace v Javě

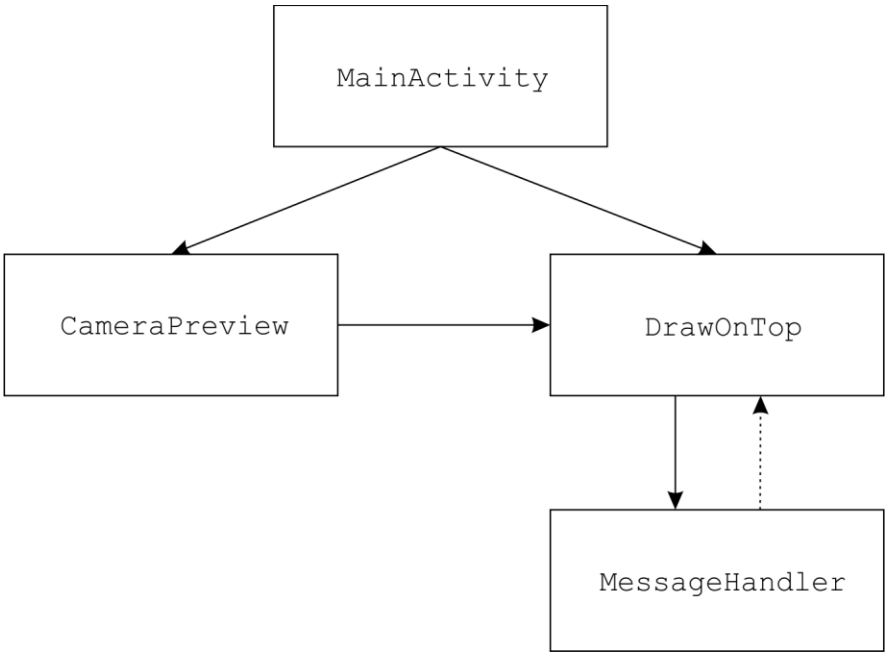
Tato část je složena ze čtyř tříd: `MainActivity`, `CameraPreview`, `DrawOnTop` a `MessageHandler`, jejich diagram je zobrazen na obrázku **4.2**.

4.1.1 Třída `MainActivity`

Rozšiřuje třídu `Activity`, která je hlavní třídou v Android systému, řídící životní cyklus aplikace. Při jejím startu je v ní načtena také knihovna s funkcemi v nativním kódu.

Správa životního cyklu aplikace. `MainActivity` implementuje metodu `onCreate()`, která je volána při spuštění. V ní je aplikace nastavena na celou obrazovku, zrušeno zobrazení názvu a vypínání displeje. Dále jsou vytvořeny instance tříd `DrawOnTop`, `CameraPreview` a `TextView` (textový popisek) pro zobrazování rychlosti jízdy. Popisku je nastavena velikost a barva písma a pozadí. Tyto komponenty jsou přidány do tzv. *ContentView* aplikace – obsahu, který je zobrazován na displeji.

Potom je získána instance třídy `LocationManager` (manažer lokací) ze systémových služeb, pro získávání informací o poloze zařízení a `LocationListener`, který naslouchá změnám polohy a při změně je vyvolána metoda `onLocationChanged()`, ve které je přepsán textový



Obrázek 4.2: Diagram tříd v Javě.

popisek s rychlostí. Rychlost je však v jednotce metry za sekundu, proto je nejprve nutné ji přepočítat na kilometry za hodinu.

Další metodou v `MainActivity` je `onResume()`. Ta je volána po `onCreate()` a připraví kritéria pro získávání polohy a od manažera lokací se snaží získat nejlepšího možného poskytovatele informací o poloze. Nejpřesnější jsou GPS satelity, pak data z vysílačů mobilních operátorů nebo bezdrátového internetu. Nakonec si vyžádá zasílání zpráv o změnách polohy.

Ještě je nutné implementovat metodu `onPause()`, volanou při pozastavení aplikace, ve které je nezbytné vypnout získávání aktualizací polohy.

Menu. Obsahuje dvě položky: *Nastavení* a *Demo*. Při jejich vyvolání je informován objekt `DrawOnTop`, který na to reaguje. V případě *Nastavení* je ještě zobrazeno dialogové okno s nápovědou, jak nakreslit čáry na displeji pro nastavení masky.

4.1.2 Třída `CameraPreview`

Slouží k zobrazování náhledu kamery na obrazovce. Rozšiřuje třídu `SurfaceView` a implementuje rozhraní `SurfaceHolder.Callback`, tzn., že musí obsahovat metody `surfaceCreated()`, `surfaceChanged ()` a `surfaceDestroyed()`, které jsou volány při změnách zobrazovací plochy náhledu. Obsahuje referenci na objekt `DrawOnTop`, vytvořený v `MainActivity`, `mDrawOnTop`.

Vytvoření plochy náhledu. Nastává při spuštění aplikace, nebo při probuzení zařízení, nebo přepnutí zpět na tuto aplikaci, např. po dokončení telefonního hovoru. Metoda `surfaceCreated()` se pokouší získat přístup k zadní kameře mobilního telefonu. Pokud je pokus

úspěšný (jiná aplikace neblokuje kameru), je nastavena nová funkce pro příjem snímků náhledu `onPreviewFrame(byte[] data, Camera camera)`, která bude volána po startu náhledu.

Pokud ještě nebyla připravena bitmapa v `mDrawOnTop` objektu, jsou získány parametry kamery a pro `mDrawOnTop` je nastavena výška a šířka obrázku podle velikosti rozlišení náhledu, je získána adresa `Holder` objektu (viz kapitola 4.2.2), vytvořena bitmapa, do které budou vykreslovány výsledky, a je spuštěno nové vlákno pro `mDrawOnTop` objekt, který bude zpracovávat snímky. Při dalších voláních `onPreviewFrame()` je `mDrawOnTop` předána reference na bytové pole `data`, které obsahuje snímek kamery v YCbCr formátu s kódováním NV21.

Změna plochy náhledu. Vyvolána po vytvoření plochy náhledu. V metodě `surfaceChanged()` jsou získány parametry kamery a z nich seznam podporovaných rozlišení náhledů. Při testování bylo zjištěno, že různé verze Android mobilních zařízení vrací seznam v různém pořadí, tzn., že některé seznamy začínají největším možným rozlišením, jiné nejmenším rozlišením. Proto jsou získány rozměry výšky a šířky náhledu prvního a posledního prvku seznamu. Větší z nich jsou pak použity, ovšem pokud je rozlišení větší než VGA rozlišení 640x480, jsou rozměry nastaveny na toto rozlišení, pokud je podporováno, aby nebylo nutné zpracovávat větší obrázky, náročnější na paměť a výpočetní výkon. Potom je vypočítán a uložen poměr stran náhledu, který může být jiný, než poměr stran displeje zařízení. Nakonec je spuštěn samotný náhled (začnou se zobrazovat snímky náhledu na obrazovce a při každém obrázku je vyvolána metoda `onPreviewFrame()`).

Zánik plochy náhledu. Nastává při vypnutí aplikace, uspání mobilního zařízení nebo při přepnutí na jinou aplikaci. Metoda `surfaceDestroyed()` přeruší běh druhého vlákna, uvolní `Holder` objekt, zastaví náhled kamery a uvolní kameru, aby ji mohly použít jiné aplikace.

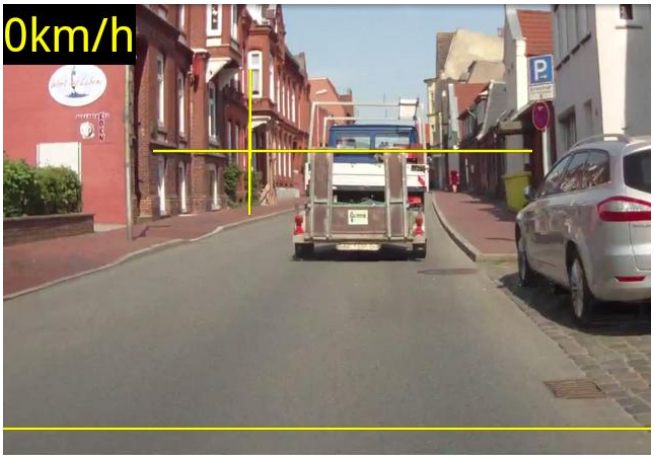
Změna rozlišení zobrazovaného na obrazovce. Při vytváření náhledu je několikrát volána funkce `onMeasure()`, ve které je možno měnit rozlišení zobrazovaného náhledu na obrazovce. Při prvním volání je nastavena výška, při druhém šířka. Výška je brána přednastavená, ale šířka je vypočítána jako výška krát poměr stran náhledu kamery. Tak je docíleno stejného poměru stran jak náhledu kamery, tak zobrazení na displeji, aby nebyl obraz deformovaný.

4.1.3 Třída `DrawOnTop`

Vykresluje na obrazovce další vrstvu grafiky nad obrazem náhledu z kamery. Rozšiřuje základní zobrazovací třídu `View` a implementuje rozhraní `Runnable` a jeho metodu `run()`, která umožňuje spouštět kód v jiném vlákně. Obsahuje deklaraci čtyř nativních metod: `detect()` pro detekci automobilů (popsánu v kapitole 4.2.4) a `initHolder()`, `freeHolder()` a `setMask()` pro práci s `Holder` objektem (kapitola 4.2.2).

Při vytvoření instance této třídy je získán standardní oznamovací tón mobilního telefonu, aby byla zajištěna přítomnost zvukového souboru, a je vytvořena instance třídy `MessageHandler` (kapitola 4.1.4).

Vykreslování. Hlavní funkcí je vykreslení bitmapy s označeným detekovaným automobilem, které se provádí v metodě `onDraw()` (zobrazeno na obrázku **4.1**). Ta není volána přímo, ale operačním systémem, po naplánování po zavolání funkce `invalidate()`. Při nastavování masky jsou v `onDraw()` zobrazovány čáry, které uživatel kreslí na dotykovém displeji, zobrazeno na obrázku **4.3**.



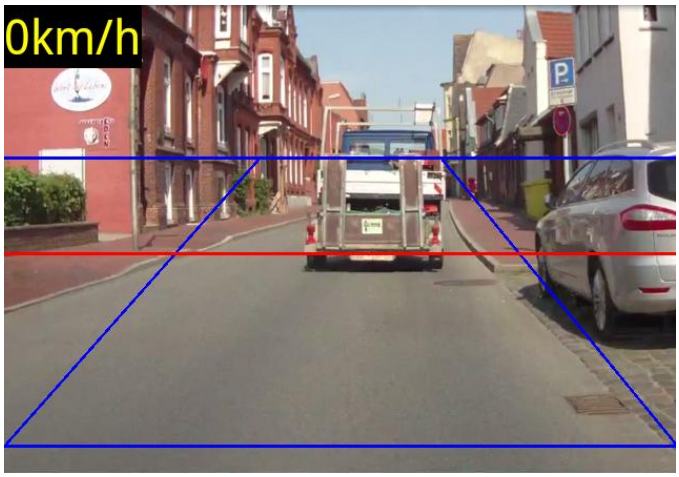
Obrázek 4.3: Režim Nastavení.

Změna masky. Ke změně masky jsou potřebné tři souřadnice: *y* souřadnice horizontu silnice, *x* není potřeba, protože horizont je přes celou šířku obrazovky a dvě *x* souřadnice, z průsečíku levého nebo pravého okraje silnice s horizontem. Spodní hranice viditelné silnice je určena předem. Tyto souřadnice jsou získány jako průměr souřadnic koncových bodů čar, které uživatel nakreslí, např.:

$$y = \frac{y_1 + y_2}{2} \tag{8}$$

S tím souvisí metoda `onTouchEvent()`, která reaguje na dotyk na obrazovku, pokud je aktivní režim *Nastavení*. Podle druhu dotyku: stisk, pohyb, uvolnění a pořadí dotyku jsou uloženy potřebné souřadnice a v případě pohybu je ještě zavolána funkce `invalidate()` pro překreslení čar s novými souřadnicemi. Ještě je nutné přepočítat souřadnice obrazovky na souřadnice náhledu kamery, pokud mají jiné rozlišení. Po nakreslení všech potřebných čar je změněna maska nativní funkcí `setMask()` a vynulovány souřadnice čar. Nová maska je zobrazena na obrázku **4.4**.

Metoda pro druhé vlákno. V této metodě `run()` je volána nativní funkce `detect()`, která provede detekci v předaném snímku z kamery a vykreslí výsledek do bitmapy. Její návratová hodnota je poslána jako druhý argument ve zprávě `Message`, spolu s argumentem žádosti aktualizace.



Obrázek 4.4: Změněná maska a obrys silnice.

4.1.4 Třída `MessageHandler`

Vnořená třída v `DrawOnTop`, obsahuje na ni slabou referenci. Běží v hlavním vlákně a zpracovává zprávy poslané z druhého vlákna v metodě `run()`, protože k uživatelskému rozhraní je možno přistupovat pouze z hlavního vlákna aplikace.

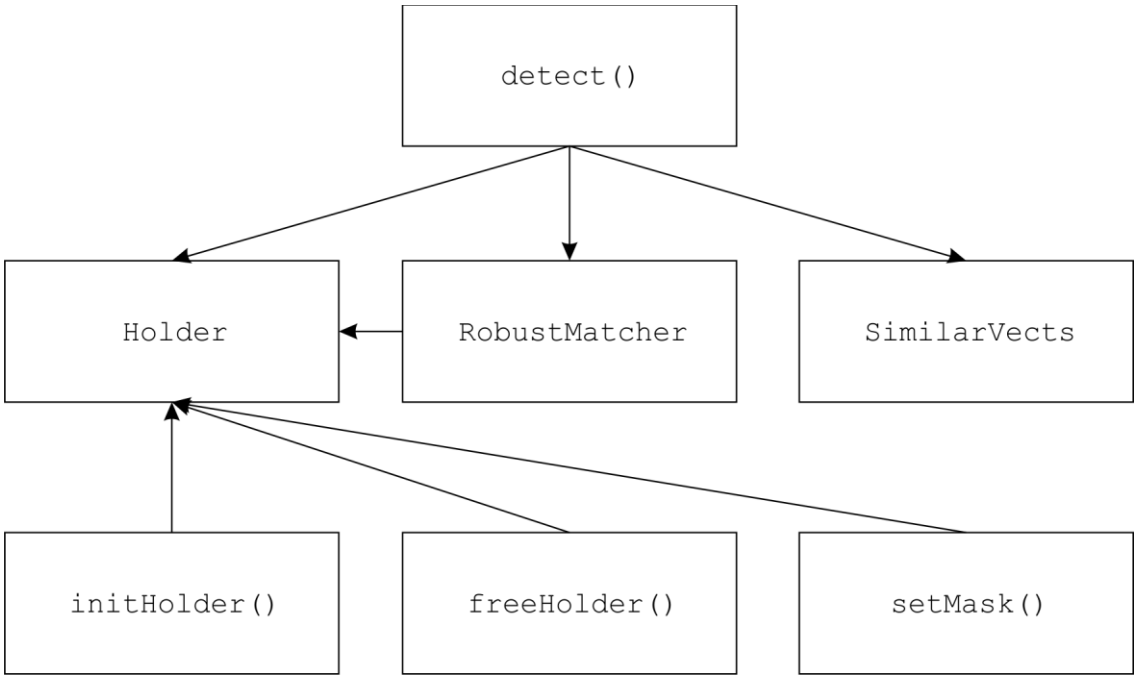
Při příchodu zprávy je vyvolána funkce `handleMessage()`. Pokud je prvním argumentem zprávy konstanta aktualizace, je provedena funkce `invalidate()`, aby byla překreslena obrazovka s nově nastavenou bitmapou po detekci. Jestliže druhý argument značí detekci nebezpečí a rychlost jízdy je větší než patnáct kilometrů za hodinu, nebo je aplikace v režimu *Demo*, je přehrán varovný signál.

4.2 Část aplikace v nativním kódu C++

Nativní funkce deklarované v `DrawOnTop` jsou implementovány v modulu `native.cpp` a využívají další tři třídy: `Holder`, `RobusMatcher` a `SimilarVects`. Diagram tříd a jejich využití funkcemi je zobrazeno na obrázku **4.5**.

4.2.1 Třída `SimilarVects`

Slouží k porovnání dvou vektorů. Přetěžuje operátor `()` jako `bool operator()(const Point2f& p1, const Point2f& p2)`, kde p1 a p2 jsou dva objekty reprezentující OpenCV body se složkami `x` a `y`, zde použity pro uchování vektoru. Nejprve jsou vypočítány velikosti vektorů, a pak jejich absolutní rozdíl. Potom je provedeno porovnání velikosti a směrů podle kapitoly 3.1.2. Pokud jsou vektory podobné, vrací operátor `()` *pravdu* (`true`), jinak *nepravdu* (`false`). Tato třída je využívána OpenCV funkcí `partition()` při třídění do množin.



Obrázek 4.5: Diagram tříd v C++ a jejich využití funkcemi.

4.2.2 Třída `Holder`

Slouží k uložení masky, ve které je prováděna detekce a seznamu klíčových bodů a jejich deskriptorů z předchozího snímku. Dále obsahuje rozměry obrázku a souřadnice některých hraničních čar. Pro prácí s `Holder` objektem v Javě jsou určeny tři nativní funkce `initHolder()`, `freeHolder()` a `setMask()`.

Při instancování třídy `Holder` je nutné znát výšku a šířku obrázku. Ty jsou použity na vytvoření nového `Mat` objektu reprezentujícího masku a její přípravu a výpočet hranice viditelné silnice.

Změna masky v objektu. Ke změně masky slouží metoda `changeMask()`, která celou masku vyplní bílou barvou, a pak podle zadaných souřadnic nakreslí polygon černou barvou tam, kde nemá probíhat detekce. Tak je vytvořena maska, kterou používají OpenCV funkce.

Inicializace. Ve funkci `initHolder()` je vytvořen nový dynamický objekt a ukazatel na něj je vrácen jako `long` adresa do Java kódu, kde je uložena v `DrawOnTop`.

Uvolnění. Funkci `freeHolder()` je předána adresa objektu, ta je přetypována zpět na ukazatel na `Holder` a je vyvolán destruktor.

Vyvolání změny masky z Javy. Funkci `setMask ()` je také předána adresa objektu a také je přetypována na ukazatel, dále je předáno pole souřadnic `coords`. K tomuto poli není možné přistupovat přímo, ale je nutné získat ukazatel na něj pomocí funkce:

```
jfloat *coordsC = env->GetFloatArrayElements(coords, 0);
```

Pak je zavolána funkce `Holder` objektu `changeMask()`, s jednotlivými souřadnicemi z pole, která změní masku. Nakonec je nutné zase uvolnit pole souřadnic:

```
env->ReleaseFloatArrayElements(coords, coordsC, 0);
```

4.2.3 Třída `RobustMatcher`

Souží k porovnání deskriptorů klíčových bodů a nalezení shodných. Je převzata z knihy *OpenCV 2 Computer Vision Application Programming Cookbook* [14] a je dále upravena pro práci s objektem `Holder`. Obsahuje OpenCV detektor klíčových bodů, extraktor deskriptorů a vyhledávač *Shod*. Tyto objekty jsou přednastaveny na práci s ORB detektorem a deskriptorem.

Obsahuje metodu `ratioTest()`, která porovnává vzdálenosti dvou nalezených *Shod*, a pokud je jejich poměr menší než zadaná hranice, jsou odstraněny ze seznamu. Tak je zamezeno výběru *Shod*, jejichž klíčové body jsou příliš blízko sebe a mohlo by tak dojít k výběru špatného klíčového bodu.

Metoda `symmetryTest()` ze seznamu *Shod* zjišťuje, zda nalezená *Shoda* z prvního obrázku do druhého je stejná se *Shodou* nalezenou z druhého obrázku do prvního. Pokud se *Shody* odkazují na stejné deskriptory, je nová *Shoda*, odkazující se na tyto deskriptory, vložena do výsledného seznamu symetrických *Shod*.

Hlavní funkcí třídy `RobusMatcher` je funkce `match()`. Slouží k nalezení klíčových bodů a seznamu *Shod*. Využívá objekt `Holder` k získání klíčových bodů a jejich deskriptorů z prvního snímku. Proto jsou detekovány klíčové body (metodou detektoru `detect()`, s využitím masky) a vypočítány jejich deskriptory (metodou extraktoru `compute()`) vždy jen v druhém snímku. Pokud jsou v prvním i druhém snímku nalezeny nějaké klíčové body, je provedeno funkcí vyhledávače `knnMatch()` hledání *Shod* z prvního obrázku do druhého, a pak hledání *Shod* z druhého do prvního. `knnMatch()` hledá k nejlepších shod pro každý deskriptor z prvního obrázku (podle zvětšující se vzdálenosti deskriptorů), bylo použito $k = 2$. Pro oba získané seznamy *Shod* je proveden test poměru vzdáleností `ratioTest()` a test symetrie *Shod* `symmetryTest()`, který vrací finální seznam *Shod*. Nakonec jsou zkopírovány deskriptory druhého obrázku do objektu `Holder`.

4.2.4 Nativní funkce `detect()`

Provádí detekci podle kapitoly 3.1. Má tři parametry: adresu Holder objektu, která je přetypována na ukazatel, bytové pole obsahující snímek z kamery ve formátu YCbCr a bitmapu, do které se vykreslí výsledek.

Pro pole se snímkem je nejprve nutné získat ukazatel funkcí `GetByteArrayElements()`. Pak jsou využity funkce z Android knihovny pro JNI `<android/bitmap.h>` pro přímý přístup k bitmapě a kreslení do ní přímo v nativním kódu. Jsou to funkce `AndroidBitmap_getInfo()`,

která zjišťuje informace o obrázku: výšku, šířku, formát a příznaky a funkce `AndroidBitmap_lockPixels()`, která uzamkne pixely bitmapy, aby nebylo možno obrázek přesouvat v paměti.

Potom je vytvořen `Mat` objekt (pouze jeho hlavička) nad daty pole snímku z kamery, jako matice `unsigned char` o velikosti náhledu kamery. Tak je hlavička vytvořena jen nad pixely s jasovou složkou obrazu (barevné se nacházejí až za nimi) a matici lze interpretovat jako RGB obrázek převedený do stupňů šedi. Čímž odpadá nutnost převádět YCbCr do RGB, a pak ještě převádět obrázek do stupňů šedi, s kterým pak pracují některé OpenCV funkce. Další matice je vytvořena pro pixely výstupní bitmapy. Tím jsou vytvořeny matice `Mat` pro přímý přístup k pixelům a není nutné provádět žádné kopírování dat z Java části do nativní části a po dokončení zpracování obrazu kopírovat data zpět do objektů v Javě.

Pak jsou vyhledány klíčové body a *Shody* pomocí objektu `RobusMatcher` a jeho funkce `match()` a vytvořen seznam vektorů posunu těchto klíčových bodů oproti klíčovým bodům uloženým v `Holder` objektu z předchozího obrázku. Dále je vypočítána celková velikost vektorů a z ní a velikosti seznamu průměrná velikost vektoru. Vektory s větší velikostí než průměrnou jsou ze seznamu odstraněny. Dále s použitím OpenCV funkce `partition()` a objektu `SimilarVects` jsou vektory roztříděny do množin a číslo množiny pro každý vektor je uloženo do seznamu `labels`. Potom je spočítán počet vektorů v jednotlivých množinách a jsou odstraněny ty množiny, které obsahují méně vektorů než deset procent velikosti seznamu vektorů.

Poté jsou vykresleny do bitmapy čáry představující silnici a horizont spolu s hranicí bezpečné vzdálenosti a nalezené vektory posunu a obdélník ohraničující je. Dále je spočítán počet klíčových bodů, které překročily hranici bezpečné vzdálenosti jako počet detekcí. Ještě jsou zkopírovány klíčové body z druhého snímku do objektu `Holder`, aby bylo možné je použít při příštím volání `detect()`. Nakonec jsou odemčeny bytové pole snímku z kamery pomocí `ReleaseByteArrayElements()` a také pixely bitmapy pomocí `AndroidBitmap_unlockPixels()`. Pokud počet detekcí překročí dvacet procent nalezených shod v obrázcích, je vrácena logická jednička, jinak nula.

4.3 Testování aplikace

Aplikace byla testována na dvou mobilních telefonech: *LG Optimus One P500* a *LG Optimus 2X P990*. *LG Optimus One* má procesor ARMv6 s taktem 600 MHz a operační systém ve verzi 2.3.3 *Gingerbread*. *LG 2X* má dvoujádrový procesor ARMv7 s taktem 1000 MHz a operační systém ve verzi 2.2 *Froyo*. K testování bylo použito referenční video se záznamem jízdy a bylo zobrazováno na monitoru, měření probíhalo vždy ve stejné části videa.

Na *LG Optimus One* bylo dosaženo frekvence detekce automobilu 3,733 detekcí za sekundu. To je příliš pomalé na reálné použití k odvrácení nebezpečí. Byl měřen čas mezi dokončením funkce `detect()` a bylo naměřeno patnáct vzorků.

Na *LG 2X* proběhla dvě měření. První s přeloženým nativním kódem pro procesory ARMv6 s nastaveným `APP_ABI := armeabi` v `Appliacation.mk` a druhé s přeloženým kódem pro ARMv7 procesory s `APP_ABI := armeabi-v7a`. Při prvním byla frekvence 0,686 detekcí za sekundu, což je 5,4 krát rychlejší než *LG Optimus One* a při druhém 0,433 detekcí za sekundu, což je 8,6 krát rychlejší než *LG Optimus One* a 1,5 krát rychlejší než první měření *LG 2X*. To už je dostatečně rychlé pro ohlášení nebezpečí při plynulém přibližování k automobilu jedoucímu před kamerou, např. při usnutí řidiče na dálnici. Ovšem při rychlém přiblížení, např. dobrzdění na křižovatce, je posun mezi snímky příliš velký a systém nestačí automobil detekovat.

Při testování bylo také zjištěno, že systém je náchylný na špatné světelné podmínky. Za jízdy v šeru nebo stínu vysokých budov nebo stromů má obraz malý kontrast a nejsou nalezeny žádné klíčové body na automobilu, který je příliš tmavý. Dále dochází k občasným falešným detekcím čar na silnici, tomu by měla zabránit filtrace v čase, podle kapitoly 3.1.3, ale ta nebyla implementována kvůli omezení rychlostí hardwaru mobilních zařízení, protože jinak by aplikace nereagovala na nebezpečí dostatečně rychle (byla by zpožděná o jednu další detekci).

5 Závěr

Cílem této práce bylo seznámení se s platformou Android, možnostmi snímání a zpracování obrazu na mobilních zařízeních, možnostmi detekce automobilů před kamerou za jízdy, práce s GPS na této platformě a implementovat aplikaci, která hlídá porušení bezpečné vzdálenosti.

Podařilo se mi vytvořit systém pro detekci automobilů, jehož zajímavostí je, že ho není nutné natrénovat na vzorových obrázcích, ani označovat hledané objekty. Hlavním úspěchem tak je, že je zároveň univerzální pro detekci osobních a nákladních automobilů nebo i přívěsů. Při špatných světelných podmínkách však detekce nefunguje správně. Aplikace zobrazuje aktuální rychlost, kterou se snaží získat od GPS satelitů nebo jiných aktuálně nejlepších zdrojů. V závislosti na nízké rychlosti je vypínáno zvukové varování před nebezpečím.

Řídící a zobrazovací části aplikace jsou implementovány v jazyce Java. Náročné výpočty v detekci, které provádí funkce z OpenCV knihovny, jsou implementovány v nativním kódu C++ s využitím Android NDK. Při testování byl zjištěn velký nárůst výkonu při použití dvoujádrového mobilního telefonu i nárůst výkonu při optimalizaci nativního kódu pro procesory ARMv7.

V dalším vývoji aplikace by bylo možné implementovat filtraci detekcí v čase, závisí to však na zvýšení výkonnosti mobilních zařízení. Jako další vylepšení by bylo možné zvětšování jasu a kontrastu obrazu, případně jeho jiné úpravy, podle hodnoty aktuálního osvětlení obrazu.

Literatura

[1] *Open Handset Alliance* [online]. [2007] [cit. 2013-04-16]. Dostupné z: http://www.openhandsetalliance.com

[2] *Android Developers* [online]. [2013] [cit. 2013-04-16]. Dostupné z: http://developer.android.com/about/index.html

[3] *Google Play* [online]. © 2012 [cit. 2013-04-17]. Dostupné z: https://play.google.com/store

[4] *Android* [online]. [2012] [cit. 2013-04-17]. Dostupné z: http://www.android.com

[5] Java Native Interface. *Oracle Java SE Documentation* [online]. © 1993-2011 [cit. 2013-04-18]. Dostupné z: http://docs.oracle.com/javase/6/docs/technotes/guides/jni

[6] *FOURCC* [online]. © 2011 [cit. 2013-05-10]. Dostupné z: http://www.fourcc.org

[7] *OpenCV* [online]. © 2013 [cit. 2013-04-18]. Dostupné z: http://opencv.org

[8] PICCARDI, Massimo. Background subtraction techniques: a review. *IEEE International Conference on Systems, Man and Cybernetics*. 2004, č. 4, s. 3099-3104. ISSN 1062-922X.

[9] VIOLA, Paul a Michael JONES. Rapid Object Detection using a Boosted Cascade of Simple Features. *IEEE Computer Society Conference on Computer Vision and Pattern Recognition*. 2001, č. 1, s. 511-518. ISBN 0-7695-1272-0.

[10] LIENHART, R. a J. MAYDT. An extended set of Haar-like features for rapid object detection. *Proceedings. International Conference on Image Processing*. IEEE, 2002, roč. 1, č. 1, s. 900-903.

[11] LOWE, D. G. Object recognition from local scale-invariant features. *Proceedings of the Seventh IEEE International Conference on Computer Vision*. 1999, č. 2, s. 1150-1157.

[12] BAY, Herbert, Andreas ESS, Tinne TUYTELAARS a Luc VAN GOOL. SURF: Speeded Up Robust Features. *Computer Vision and Image Understanding*. 2008, roč. 110, č. 3, s. 346-359.

[13] RUBLEE, Ethan, Vincent RABAUD, Kurt KONOLIGE a Gary BRADSKI. ORB: An efficient alternative to SIFT or SURF. *2011 International Conference on Computer Vision*. IEEE, 2011, č. 1, s. 2564-2571. ISSN 1550-5499.

[14] LAGANIÉRE, Robert. *OpenCV 2 Computer Vision Application Programming Cookbook: Over 50 recepies to master this library of programming functions for real-time computer vision*. 1. vyd. Brimmingham: Packt Publishing, 2011, iii, 287 s. Quick Answers to Common Problems. ISBN 978-1-84951-324-1.