VŠB – Technická univerzita Ostrava
Fakulta elektrotechniky a informatiky
Katedra informatiky

Algoritmy pro detekci dělící čáry v obraze

Lane Detection Algorithms

2012 Bc. Radek Tesař

Prohlašuji, že jsem tuto diplomovou práci vypracoval samostatně. Uvedl jsem všechny literární prameny a publikace, ze kterých jsem čerpal.

V Ostravě 1. května 2012 .....

Rád bych na tomto místě poděkoval všem, kteří mi z prací pomohli, ať cennými radami, návrhy nebo nápady, protože bez nich by tato práce nevznikla. Jmenovitě bych rád poděkoval vedoucímu mé práce ing. J Michalu Krumniklovi za doporučení a pomoc při vývoji i psaní mé práce, ing. Milanu Brejlovi, Ph. D. za neutuchající entuziasmus při organizaci soutěže FRC, oponentovi mé práce doc. Dr. Ing. Eduardovi Sojkovi, za rady a nápady týkající se zpracování obrazu, všem soutěžícím, se kterými jsme vedli nekonečné diskuze o dané problematice, rodičům a přítelkyni, kteří se mnou měli trpělivost a všem ostatním, kteří se mi sem nevešli.

Abstrakt

Tato práce se zabývá vývojem algoritmů pro detekci dělících čar v obraze snímaném kamerou umístěném v automobilu. Typické použití je sledování středových a krajních dělících čar nebo kolejí. Cílem je dosažení rozumných výsledků při použití omezených zdrojů (procesor, paměť) tak, aby výsledný produkt mohl být použit jako autonomní zařízení například pro řízení malého soutěžního automobilu FRC. Úkolem v soutěži je zajet co nejrychleji stanovený počet kol na neznámé trati. Pro řízení a zpracování obrazu se předpokládá použití některých z moderních procesorů ARM řady Cortex, operační systém Linux, nebo Windows CE a knihovna pro obrazové funkce OpenCV.

Klíčová slova: OpenCV, Linux, Windows CE, procesor, detekce dělící čáry, Algoritmus, hranový detektor, Canny, Houghova transformace, binární obraz, zpracování obrazu, vážený průměr, plovoucí průměr.

Abstract

This work deals with the development of Lane detection Algorithms in pictures, captured inside the car. Typical application is the monitoring lanes, or rail. The aim is to achieve reasonable results using limited resources (CPU, memory), so that the resulting product could be used for example as an autonomous control of a small car in the FRC competition. The challenge in the competition is to go as fast as possible at unknown number track. For control and image processing is expected to use some of the many modern processors ARM Cortex, Linux, or Windows CE as an operating system and imaging library for OpenCV.

Keywords: Open CV, Linux, Windows CE, processor, Lane detection, Algorithm, Edge Algorithm, Canny, Hough transformation, Binary picture, picture detection, weighted average, moving average.

Seznam použitých zkratek a symbolů

BSD	–	Berkeley Software Distribution – Licence pro svobodné šíření software
DCT	–	Diskrétní Cosinova transformace
DNA	–	Deoxyribonukleová kyselina
DPS	–	Deska plošných spojů
DSP	–	Digitální signálový procesor – vhodný pro zpracování matematických dat v reálném čase (audio nebo video)
EDF	–	Edge Distribution Function
FFT	–	Rychlá Fourierova transformace
FPGA	–	Programovatelné logické pole
FPS	–	Frames per second – počet snímků za sekundu
GB	–	Giga Byte, miliarda bytů (přesně 1 048 576)
GHz	–	Giga Hertz – Miliarda Hertz (Hertz je jednotka frekvence)
GPS	–	Global Positioning System – satelitní navigační systém
HW	–	Hardware- elektronické zařízení, například počítač
PC	–	Personal Computer – Osobní počítač
QNX	–	RTOS operační systém pro embedded aplikace
QVGA	–	čtvrt VGA rozlišení(320x240 bodů)
RANSAC	–	Random Sample Consensus – algoritmus pro nalezení stejných vzorků
RGB	–	Red, Green, Blue – červená, zelená, modrá (základní barvy)
ROI	–	Region of Interest – oblast zájmu
RTOS	–	Real Time Operating System – operační systém pro zpracování v reálném čase
SIMD	–	Single Instruction Multiple Data
SW	–	Software- kód vykonávaný počítačem
VGA	–	VGA (Video Graphics Adaptor) – V současné době označení základního rozlišení monitoru (640x480 bodů)

Obsah

Seznam tabulek

Seznam obrázků

Seznam výpisů zdrojového kódu

1 Úvod

Práce se zabývá vývojem algoritmů pro detekci dělících čar v obraze snímaném kamerou umístěném v automobilu. Typické použití je sledování středových a krajních dělících čar nebo kolejí. Cílem je dosažení rozumných výsledků při použití omezených zdrojů (procesor, paměť) tak, aby výsledný produkt mohl být použit jako autonomní zařízení například pro řízení malého soutěžního automobilu FRC. Úkolem v soutěži je zajet co nejrychleji stanovený počet kol na neznámé trati. Pro řízení a zpracování obrazu se předpokládá použití některých z moderních procesorů ARM řady Cortex A, operační systém Linux nebo Windows CE a knihovna pro obrazové funkce OpenCV. V práci jsem se zaměřil nejprve na teoretické aspekty problému a rešerše jiných řešení a autorů, dále pak na vývoj vhodného algoritmu pomocí knihovny na zpracování obrazu OpenCV a klasického desktopového PC. Po dosažení uspokojivých výsledků bude následovat portace na vhodnou platformu a ověření funkčnosti na vhodném vývojovém kitu. Poslední fází pak bude návrh DPS a vytvoření referenční platformy pro ověření celého řešení.

V kapitole 2 nastiňuji nejdůležitější kritéria pro vývoj algoritmu, omezení kterými jsme v daném kontextu limitováni a dále se zabývám rešerší již publikovaných řešení a jejich rozborem. U každého algoritmu také posuzuji jejich výhody a nevýhody, případně vhodnost použití. Následně hodnotím používané metody zaměřené na podobné účely a jejich vhodnost pro uvedenou oblast. V průběhu času také dochází ke změnám v oblibě jednotlivých druhů algoritmů – důvodem může být například dostupnost výkonnějších počítačů nebo vhodných knihoven pro zpracování obrazu a podobně. Příkladem budiž například různých hranových detektorů, jejich výhody, nevýhody, oblíbenost v určitém období, atd. Důraz přitom kladu na rychlost zpracování a nároky na paměť procesoru.

V kapitole 3 se věnuji teoretické stránce úlohy, bez detailního zkoumání principů detekce a zpracování obrazu. Výsledky předkládám čtenáři tak, aby byly srozumitelné i laikům, kteří se nezabývají teorií zpracování obrazu, ale zajímají se uvedenou problematikou a znají základy programování v jazyce C. Pro zájemce uvádím také dostatek teoreticky zaměřené literatury jak v anglické, tak v české verzi.

Ve čtvrté kapitole pak předvedu způsob a výsledky testů algoritmu. Testoval jsem jej jak z časového hlediska (náročnost na strojový čas a výkon procesoru), tak z hlediska spolehlivosti a odolnosti proti rušení, nebo nepříznivým světelným podmínkám. Detailní výsledky testů jsou pak přehledně zpracovány do grafů v příloze.

V poslední, páté kapitole, se pak věnuji možnostem zlepšení algoritmu, případně návrhům na přepracování tak, jak mě napadly v průběhu vývoje algoritmu. Při vývoji jsem také narazil na řadu úskalí a problémů, z nichž některé také zmiňuji. V neposlední řadě se zabývám vhodnou platformou pro běh algoritmu – předpokládám totiž nasazení v embedded aplikacích (formou instaluj a zapomeň), kde se předpokládá minimální obsluha a péče o zařízení, maximální spolehlivost a odolnost proti vnějším vlivům a dlouhodobá spolehlivost bez výpadků.

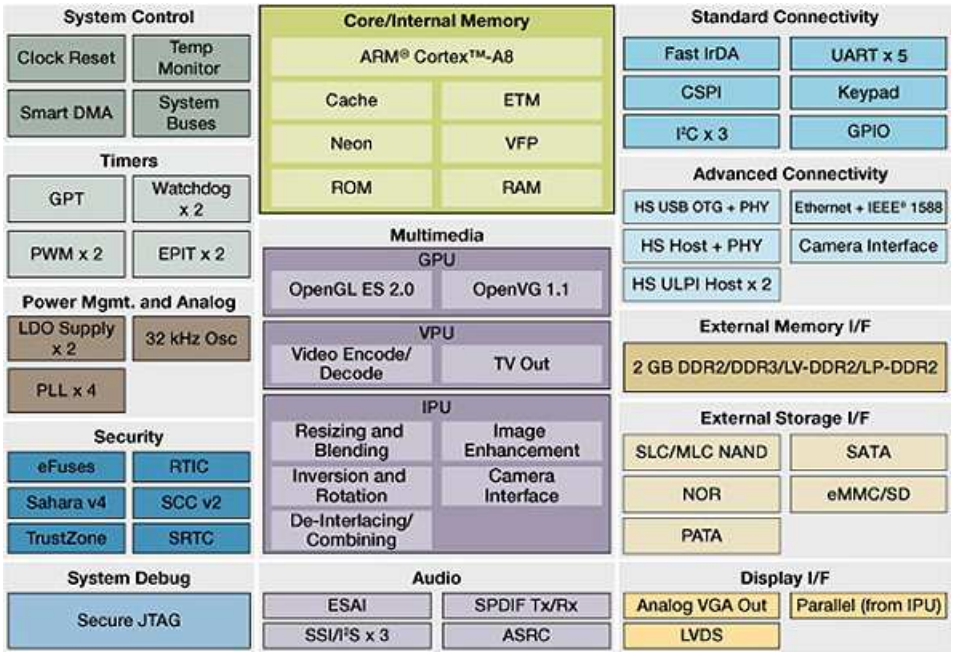
2 Kritéria pro výběr platformy, druhy algoritmů, rešerše

Složitost a náročnost zpracování obrazu je pro počítače enormní. To co se lidskému oku zdá jednoduše a rychle identifikovatelné, není běžný počítač schopen stoprocentně identifikovat a často k tomu potřebuje enormní výpočetní výkon a komplikovaný matematický aparát. Vývoj detekčních algoritmů je stále v počátcích a stále se objevují nové řešení a způsoby detekce obrazu. Některé z nich zmiňuji dále. U většiny z nich také naznačuji použitý matematický aparát a připomínám potíže a problémy se kterými se autoři setkali. Mnoho z nich jsem musel při svém vývoji také řešit. Jisté však je, že neexistuje (a pravděpodobně nikdy existovat nebude) žádný univerzální algoritmus, který by umožnil snadnou detekci obrazu počítačem.

Současné počítače také nejsou pro detekci obrazu příliš vhodné a proto si budeme muset počkat na jiný systém počítačů (kvantové, nebo založené na DNA), pomocí nichž bude možné provádět lepší detekci obrazu. Počítače používané v současnosti zpracovávají data přesně podle programu a jsou tedy plně deterministické. To není pro zpracování obrazu (a stochastických jevů obecně) příliš vhodné. V takovém prostředí je totiž pro každý stochastický jev nutno vytvořit novou podmínku, nebo navrhnout algoritmus, který zajistí pokrytí alespoň větší části stochastických jevů (například průměrování hodnot, váha objektu, atd.). Pro zpracování a detekci obrazu je pak potřeba extrémní výkon a velikost paměti běžných počítačů.

Algoritmy pro zpracování obrazu jsou poměrně novou, ale překotně se vyvíjející oblastí. První práce zabývající se touto problematikou (které ve své práci zmiňuji) pocházejí z roku 1996. V té době si autoři museli vystačit s procesory okolo 200MHz a několika desítkami MB RAM. O deset let později již měli k dispozici procesory s rychlostí 3GHz a 1GB RAM. Dnes jsou podobné algoritmy testovány na vícejádrových systémech s 4–8GB RAM. Tento značný posun ve výkonu počítačů (cca desetinásobek za posledních 10 let) znamená, že použité funkce u novějších algoritmů jsou mnohem pokročilejší nebo náročnější i při zachování schopnosti zpracovávat větší obrazy v reálném čase.

Pokud chceme zpracovávat obraz v realtimových aplikacích (a počítáme embedded aplikace), potřebujeme výkonné zařízení (procesor, DSP, FPGA), na kterém budeme zpracování obrazu provozovat. Pro každou platformu se pak bude lišit jak použitý HW tak SW. Různé platformy poskytují různý výkon a každá z nich má své výhody a nevýhody. Předpokládejme však, že si chceme co nejvíce zjednodušit práci a nebudeme tedy vymýšlet a programovat základní (známé) algoritmy. Proto je nejvýhodnější použít vhodný procesor, na kterém můžeme provozovat některou z knihoven pro zpracování obrazu. Takových knihoven je možno nalézt několik, ale nejznámější a asi nejlépe udržovaná je openCV. Autoři, kteří předpokládají zpracování obrazu pouze na PC (nebo při vývoji nezohledňují embedded aplikace), mají zjednodušenou práci ve výběru platformy. Navíc mají k dispozici výkonově a cenově prakticky neomezené prostředky, které se navíc velmi rychle vyvíjí.



Obrázek 1: Blokové schéma procesoru i.MX535

2.1 Kritéria pro výběr platformy

V posledních letech dochází k obrovskému boomu ve vývoji výkonných procesorů určených pro embedded aplikace (téměř výhradně architektury ARM). Výkon takových procesorů se velmi blíží výkonu klasických pracovních stanic (známých pod zkratkou PC – Personal Computer). Výhodou však je, že nepotřebují aktivní a většinou ani pasivní chlazení a jejich spotřeba je v desetinách spotřeby klasických PC. Příkladem může být procesor Cortex A8 i.MX535 od firmy Freescale, který pracuje na kmitočtu 1,2GHz, může adresovat až 4GB RAM a při teplotě jádra $125°$ pak spotřebuje pouhé 3 Watty! Na obrázku 1 je pak blokové schéma tohoto procesoru s vyznačenými periferiemi. Je vidět, že je velmi dobře vybaven pro multimediální aplikace, a proto se často používá ve smartphonech nebo tabletech.

Tyto procesory se dnes objevují v tzv smartphonech, tabletech, netboocích a jiných zařízeních běžného života. Často slouží pro připojení k internetu, přehrávání audio nebo video streamů, nebo k využívání jinými náročnými aplikacemi, jejichž doménou byly donedávna právě PC. Pro jejich běh je vyžadován „klasický“ operační systém (tak jak jej známe z PC, nejznámější je některý typ UNIXu, typicky určitá distribuce Linuxu, nebo Windows, a podobně). Výhodou takových procesorů však je, že mohou pracovat s nějakým RTOS, například upravený Linux, QNX a podobně.

Kapacita paměti se pak také blíží PC – typicky mívají RAM okolo 1GB a flash v řádu desítek GB. Rychlost takových procesorů je pak okolo 1GHz (pro ARM Cortex A8), nebo pro starší ARM11 pak okolo 600 MHz. Nové jádra (Cortex A8, A9) bývají také více

jádrové (v současnosti dvou až čtyřjádrové). Pokud budeme chtít použít takový procesor pro detekci a zpracování obrazu, pak z uvedeného vyplývá:

1. Tyto procesory umožňují běh reálných operačních systémů (Linux, Win CE, atd).
2. Jejich výkon a paměť je nižší než výkon typických PC.

Bod 1 nám tedy umožní vytvářet a spouštět na takovém zařízení běžné programy, které využívají služeb systému, nebo například knihovnu OpenCV. Bod 2 naopak znamená, že musíme vybírat takové postupy a algoritmy, které umožní minimalizaci spotřeby paměti a strojového času. Dalším kritériem je pak rychlost algoritmu. Pokud vezmeme v úvahu, že kamera snímá 25–30 snímků za sekundu, máme na zpracování jednoho snímku přibližně 33–40 milisekund. Snížení počtu snímků za sekundu není možné z toho důvodu, že čím vyšší rychlostí vozidlo pojede, tím delší úsek za tuto dobu ujede a je tedy nutno vyhodnocovat delší úsek cesty. To však často není možné (např. když se blížíme nebo se přímo nacházíme v zatáčce), protože sledovaný úsek je prostě příliš krátký.

Pro představu: při rychlosti 90 km/h ujede vozidlo dráhu 25 metrů za sekundu, což odpovídá posunu o 1 metr mezi jednotlivými snímky, při 25 snímcích za sekundu. Často se ale můžeme setkat s mnohem vyššími rychlostmi (1,5–2 násobné). Jak dále ukáži, často není možné správně detekovat stávající snímek a je nutno pokračovat zpracováním následujícího. Důvodem jsou většinou nevhodné světelné podmínky, nebo ztráta dělící čáry (překážka na vozovce, chybějící čára, atd.). Díky tomu vozidlo ujede větší vzdálenost, než získá opět reálnou informaci o trati.

Dále pak je možno snižovat výpočetní náročnost zmenšením zpracovávaného obrazu, což má ovšem za následek snížení rozlišovací schopnosti a to pak bude mít za následek selhání detekčního algoritmu. Jako rozumné řešení se jeví velikost obrazu QVGA (320x240 obrazových bodů) s možností snížení na polovinu, pokud by to bylo nezbytně nutné. Dále má pak vliv na velikost zpracovávaných dat typ obrazu. Nelze použít žádný kompresní algoritmus, protože při kompresi/ dekompresi by docházelo k dalšímu zdržení.

Další úspory paměti a času dosáhneme použitím černobílého obrazu (budeme zpracovávat pouze jasovou složku, oproti třem při použití barevného obrazu). Ideální je tedy použití černobílé kamery, nebo alespon její nastavení do černobílého režimu. Další otázkou pak je samotná kamera – její výběr je klíčový. Protože se jedná o zpracování rychle se měnících scén, je nutno mít kvalitní čip i objektiv. Pro rychlé scény se používají kratší časy závěrky, což ale vede ke snížení citlivosti. To je pak problém za šera, deště, mlhy nebo jiných nepříznivých vlivech a v některých případech může vést k fatálním chybám ve zpracování obrazu. Autoři některých algoritmů kladou největší důraz právě na předzpracování obrazu (před převodem do binární podoby).

Pokud dojde k chybám v předzpracování obrazu, algoritmus se z nich nevzpamatuje a celý snímek je pak nepoužitelný. Výpočet takového snímku nelze brát v úvahu. Chyby způsobené světelnými problémy (a nízkou citlivostí), se nejvíce projeví právě ve fázi předzpracování obrazu. Problémem však může být i opačný jev, označovaný jako přeexpozice. To znamená, že světla je naopak příliš. Navíc tento jev se může projevit i velmi nečekaně– v případě, že se lidskému oku zdá, že je světla málo nebo tak akorát. Je

to způsobeno vysokou citlivostí kamery na infračervené záření (nejvíce jsou na ně citlivé právě černobílé kamery). Tohoto jevu se využívá právě při nočním vidění, kdy se scéna osvětluje infračerveným světlem, na které člověk nereaguje. Tento jev se pak projevil i v mém testovacím videu.

Shrnu-li výše uvedené požadavky, vyjdou nám jako limitující faktory následující:

- Použití černobílé kamery, nebo barevné, která umožňuje černobílý režim.
- Velikost obrazu QVGA nebo menší (s možností sw nastavení).
- Co nejvyšší počet snímků za sekundu (alespoň 25–30) a z toho vycházející limit zpracování jednoho snímku za 33–40 milisekund.
- Kvalitní kamera, spolehlivá a rozměrově malá, s nízkou spotřebou.
- Složitost algoritmu taková, že bude stávající snímek zpracován do příchodu dalšího. S tím souvisí nutnost použití dostatečně výkonného procesoru.
- Možnost ztráty informace o trati – nutnost vycházet z posledních dostupných informací, tedy nutnost výpočtu odhadu, jak bude vypadat trať v následujícím snímku.

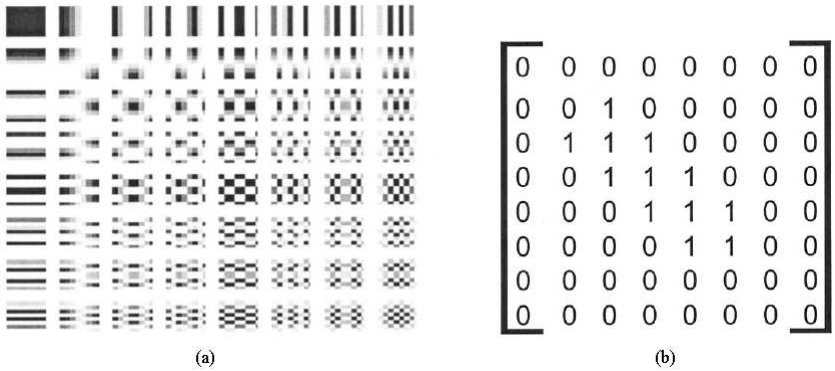
Tyto kritéria pak budou klíčové při návrhu algoritmu.

2.2 Druhy algoritmů

V této části se budu zabývat existujícími algoritmy a možností jejich implementace v závislosti na námi definovaných kritériích. Oblast zpracování obrazu v dopravě a speciálně detekce dělících čar je poměrně novou a rychle se rozvíjející oblastí. Za tak krátké období (něco málo přes 10 let) již existuje řada více či méně zdařilých algoritmů, které byly vyvíjeny na půdách různých univerzit nebo jako diplomové práce. Většinou se však jedná o experimentální práce, zkoušené pouze krátkodobě. Tento výběr není vyčerpávající a mapuje pouze část dostupných prací. U každé práce také krátce hodnotím spolehlivost a výkon uvedených algoritmů. Budeme se také zabývat pouze algoritmy založenými na detekci dat, získaných z obrazu snímaného kamerou. Existuje totiž celá řada komplexních systémů, které využívají k detekci a identifikaci polohy celé řady dalších senzorů, od GPS systémů, přes ultrazvukové, laserové nebo radarové dálkoměry, radionavigační prostředky a další.

Počátek vývoje algoritmů pro detekci dělících čar se datuje ke konci minulého tisíciletí (v 90. létech 20. století), tedy poměrně krátký čas. Za tu dobu prošly tyto algoritmy bouřlivým vývojem a rozdělily se do několika generací. Obecně se dá říct, že se algoritmy dělí do následujících skupin:

- Detekce na bázi hran.
- Detekce založené na frekvenční doméně.
- Detekce založená na adaptivních silničních šablonách.
- Detekce na bázi statistických kritérií.



Obrázek 2: Základní elementy DCT

2.2.1 Detekce na bázi hran

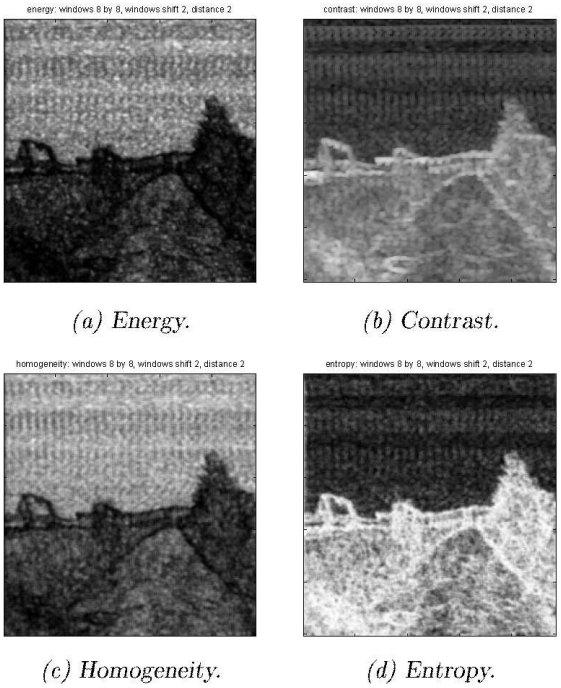
Detekce na bázi hran je založena na prahování obrazu(princip bude objasněn později).Tím se vytvoří z černobílého nebo barevného obrazu binární obraz. Taková detekce může dobře fungovat v případě jasných a segmentovaných čar. Nebude však dobře fungovat pokud bude v obraze mnoho rušivých čar. Problematická je také detekce vzdálenějších objektů (čar), proto se doporučuje rozdělit obraz na vzdálenější a bližší regiony. Existuje také řada filtrů, zaměřených na správnou detekci hran, zvyšujících cenu hrany ve správném (očekávaném) směru a snižujícím (potlačujícím) cenu hrany v jiných směrech. Tyto filtry jsou pak výpočetně poměrně nenáročné a umožňují úspěšné prahování za různých světelných podmínek (za jasného světla i ve stínu). První generace detekčních algoritmů byla většinou tohoto typu. Typickým příkladem takového algoritmu může být například [1] z roku 2004, nebo [2] z téhož roku.

2.2.2 Detekce založené na frekvenční doméně

Většinou se jedná o algoritmy založené na Fourierově transformaci. Úspěšně zpracovají nadbytečné data, ale mívají problémy s komplexním stínováním. Data se převádí do frekvenční domény a takové algoritmy jsou odolné na rušivé hrany. Příkladem hezkého algoritmu je LANA ([10]), založená na DCT. Daný obraz je rozdělen na bloky 8x8 pixelů, a pak ortogonálně rozdělen na 64 DCT základních elementů (viz obrázek 2). Je však zaměřený výhradně na diagonální hrany a má sníženou účinnost například při změně jízdního pruhu.

2.2.3 Detekce založená na adaptivních silničních šablonách

Je založena na předdefinovaných šablonách, kde s každým pixelem obrazu se provede logický součin odpovídajícího pixelu šablony. Před tím se však provádí inverzní perpektivní zakřivení pro odstranění perspektivy z obrazu. Tím se sníží počet potřebný šablon. Předpokládá se ovšem konstantní silniční vzory, proto je tato metoda málo odolná



(a) Energy. *(b) Contrast.*

(c) Homogeneity. *(d) Entropy.*

Obrázek 3: Statistické kritéria (převzato z [4]).

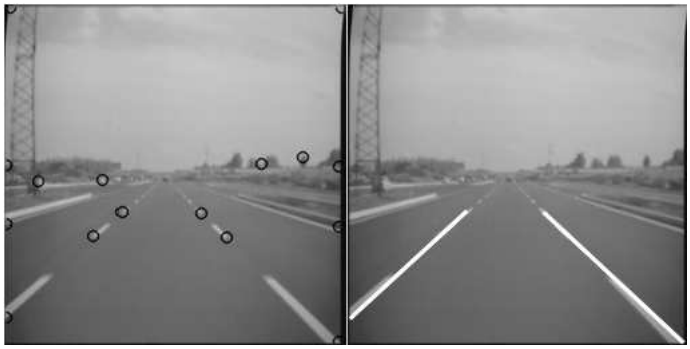
proti měnícím se (neznámým) obrazům, například při změně technologie povrchu silnice (kostky, asfalt, beton). Je však velmi dobře použitelná, pokud jsou dělící čáry málo výrazné a tedy obtížně detekovatelné jinými metodami. Její výhodou je vysoká rychost, protože operace logického součinu je elementární operace na všech typech procesorů. Takový algoritmus prezentovali například [3] v roce 2006.

2.2.4 Detekce na bázi statistických kritérií

Jsou to například energie, homogenita, nebo kontrast a používají se k odlišení silniční oblasti a oblasti která k silnici nepatří (okolní krajina, stromy u silnice nebo pole). Tato metoda se používá pro detekci venkovských cest, kde často chybí silniční čáry. V takovém prostředí ostatní algoritmy selhávají, protože jim chybí typické znaky silnice (například zmiňované krajnice nebo středové dělící čáry). Krásnou prací na toto téma je [4] z roku 1997, ze které uvádím obrázek 3 statistických kritérií.

2.3 Rešerše existujících algoritmů

Pro rešerši jsem si vybral několik stávající algoritmů vytvořených v období od roku 1999 do 2011. Většinou se jedná o experimentální algoritmy patřících do různých skupin. Zajímavý je také přístup autorů ke zpracování obrazu – od jednoduchých vlastních funkcí pro zpracování obrazu, až po komplexní algoritmy využívající například knihoven openCV.



Obrázek 4: Vlevo detekce rohů, v pravo výsledek RANSAC (převzato z [6]).

Tam, kde byly dostupné podklady, uvádím také hodnocení časové náročnosti algoritmů, většinou na počítačích dostupných v době vzniku algoritmu. Zde se nejvíce projevil pokrok v dostupné technice. Pro srovnání LANA z roku 1999 byla testována na Pentiu 266 v 96MB RAM a zpracování obrazu 640x480 bodů trvalo 30 sekund, zatímco ve [9] z roku 2010 testovali autoři stejně veliký obraz na Pentiu 4 běžícím na 3MHz s 1GB RAM. Na tomto počítači však stihli zpracovat přibližně 15 snímků za sekundu. Pro další studium doporučuji například [5], nebo další algoritmy dostupné na webu.

2.3.1 Lane Detection Based on the Random Sample Consensus

Tento algoritmus pochází z roku 2011 a byl uveřejněn v [6]. Pro zvýšení robustnosti a zlepšení detekce v reálném čase používá při předzpracování obrazu filtr pro posílení detekce čar a odstranění nezajímavých informací. Zajímavá je funkce Contrast Enhancing pro zvýšení kontrastu před převodem obrazu do binární podoby. Autor uvádí, že to má pozitivní dopad na rozpoznávání při prahování, zvláště v případě horší viditelnosti (za šera, v mlze, dešti a podobně). Nejprve tedy převedou barevný obraz na černobílý, dále provedou výše zmiňované vstupní předzpracování. Pomocí Cannyho detektoru jej převedou do binární podoby.

Následně provedou detekci hran a vyhledají významné rohy. Nakonec uplatní RANSAC (Random Sample Consensus). Ten počítá počet bodů mezi dvěma nalezenými rohy. Pro největší počet nalezených vzorků je pak spojí dohromady. Tato funkce je výhodná například pro přerušované čáry, nebo tam, kde jsou čáry binárního obrazu přerušované v důsledku chybné detekce hran při předzpracování obrazu. Na obrázku 4 je pak vidět detekce rohů a výsledek algoritmu RANSAC.

Výhody algoritmu:

- Odstraní rušivé pixely mimo oblast zájmu.
- Přesněji identifikuje hledané čáry v obraze.
- Snižuje časovou náročnost zpracování.

Obrázek 5: výsledný obraz algoritmu Line Following Robot (převzato z [7])

- Dobře reaguje za šera nebo ve stínu.

Nevýhody:

- Horší reakce za jasného dne a na přímém slunci.
- Má problémy v případě poškozených nebo chybějících krajnic, případně středových čar.

2.3.2 Architecture of the Vision System of a Line Following Mobile Robot Operating in Static Environment

V [7] uvádí autoři řešení s minimálními HW nároky, velmi levně realizovatelné za pomoci obyčejné web kamery. Cílem je vytvořit v poslední době velmi populárního robota sledující čáru na podlaze. Zajímavý algoritmus je navržen pro sledování barevné čáry, proto předpokládá vstupní barevný obraz, který nepřevádí na černobílý a následně na binární, jako většina algoritmů. V obrazu hledá čáru dané barvy. V případě, že dojde k její segmentaci, tak ji následně spojuje. Umístění kamery se doporučuje vertikálně, pro sledování pouze podlahy a také eliminaci perspektivy. Pro naše účely není příliš vhodné, i když stojí za prostudování.

Je zajímavý také tím, že se zabývá všemi třemi barevnými složkami a nepřevádí barevný obraz na černobílý, jako to dělá většina algoritmů. Nejprve tedy vytvoří tři obrazy pro každou barvu zvlášť, odečte modrý a zelený obraz od červeného (tím dostanou pouze červené objekty, tedy teoreticky pouze vodící čáru), provede prahování, dilataci a detekci kontur. Nakonec spojí jednotlivé čáry. Na obrázku 5 je vlevo vstupní obraz, a vpravo pak výstup algoritmu s vyznačenou vodící čárou.

Výhody algoritmu:

- Zajímavý nápad s přímým zpracováním barevného obrazu.

- Využívá ostatních složek obrazu k rychlé detekci vodící čáry.
- Rychlý a levný algoritmus, s minimálními HW nároky.

Nevýhody:

- Předpoklad specifického umístění kamery (netypicky ve vertikálním směru).
- Není odolný na rušivé objekty stejné barvy jako je vodící čára.
- Algoritmus je přísně statický (v záběru kamery nesmí být jiné pohybující se objekty).

2.3.3 A Practical Method of Road Detection for Intelligent Vehicle

Autoři [8] prezentují jako klíčový prvek svého algoritmu zlepšené jádro Sobelova operátoru pro zvýšení robustnosti algoritmu. Aby mohl vhodně reagovat na změnu osvětlení, zavádí také dvojité prahování. Z binárního obrazu pak extrahují hrany pomocí Houghovy transformace a algoritmu SUSAN, založeném na adaptivním prahování s různým kontrastním poměrem. SUSAN algoritmus slouží pro detekci okolních vozidel a varování při směně jízdního pruhu, proto není pro tuto práci zajímavý. Autoři prezentují algoritmus jako robustní a efektivní současně.

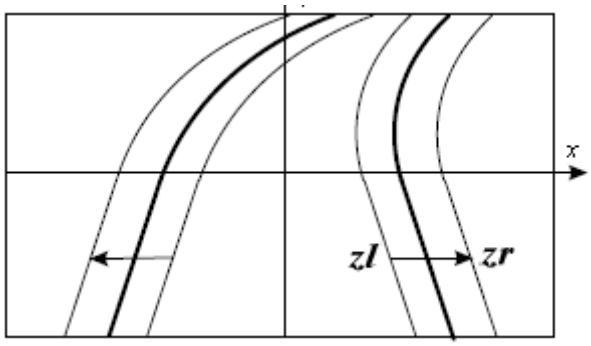
Originální obraz filtrují mediánovým filtrem, následně EDF. Jak již bylo řečeno, klíčovým prvkem je vylepšené jádro Sobelova operátoru, v $45°$, $135°$ a horizontálním směru. Důležitým krokem je také výběr hodnoty prahu v procesu získávání binárního obrazu. Binární obraz je získán pomocí metody adaptivního dvojnásobného prahování, které je možno použít pro různé osvětlení a má lepší výkon v reálném čase. Principem je výběr dvou sousedních pravoúhlých oken v dolní části obrazu a výpočtu průměrné hodnoty jasu (šedi) v každém okně.

Pro univerzální detekci čar v obraze předpokládají autoři lineární model silnice v blízké a zakřivený ve vzdálené části obrazu (obrázek 6). V blízké části je použit lineární model $x = ky + b$, ve vzdálené pak model třetího stupně $x = k_1y^3 + k_2y^2 + k_3y + k_4$. Dále předpokládají pro přerušovanou čáru, že čára pokračuje jako nepřerušovaná. Z důvodů velké šumové imunity a necitlivosti na přerušení čáry využili pro detekci čar v blízké části obrazu Houghovu transformaci.

Algoritmus byl testován na videosekvencích 180 VGA obrazů (640x480 bodů) pořízených při jízdě vozidlem při rychlosti 80 km/h pro zjištění spolehlivosti algoritmu. Výsledek činnosti algoritmu je vidět na obrázku 7. Na obrázku 7a je vidět detekce silničních čar a jednoho (vlevo) nebo více (vpravo) vozidel v přímém směru. Na 7b je vidět detekce za šera a v případě, že je silnice pokryta stíny. Na poslední části 7c je vidět výsledek detekce za deště a v případě, že je silnice pokryta dalšími značkami (například šipky).

Výhody algoritmu:

- Zajímavý nápad s využitím upraveného Sobelova operátoru (pracující v diagonálním směru).
- Využívá adaptivní dvojnásobné prahování pro zvýšení robustnosti.



Obrázek 6: Rozdělení obrazu na lineární a zakřivený model

- Implementace varování nebezpečí srážky v přímém směru.

Nevýhody:

- Pro odstranění falešného varování nebezpečí srážky jsou nutné další senzory a čidla.
- Málo citlivý při dešti nebo v noci.

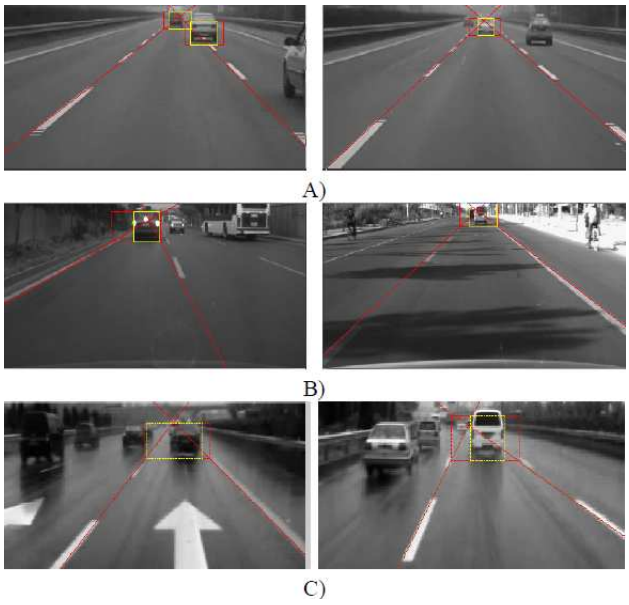
2.3.4 The Implementation of Lane detective Based on OpenCV

Autoři velmi hezkého algoritmu [9] používají knihovnu openCV a založili jej na Houghově transformaci. Jejich algoritmus je založen na modelu lineárních čar a součástí jejich dokumentu je i jeho praktické ověření. Autoři předpokládají, že cesta v obraze je tvořena lineárními čarami v obraze, přičemž zatáčky mohou být rozděleny do kratších částí. Každá taková část zatáčky pak může být proložena tečnou a vypočtena jako část přímky. Předpokládá se, že na silnici nebudou zatáčky s malým poloměrem (typicky je algoritmus určen pro dálnice nebo rychlostní silnice). Proto může algoritmus dodávat poměrně přesné výsledky. Pro vlastní detekci pak byla zvolena Houghova transformace z důvodu relativně rychlého vyhledání vektoru přímky binárním obrazu.

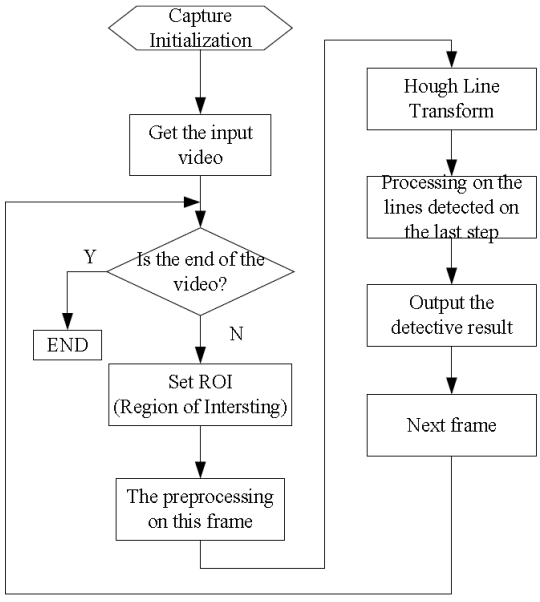
Po sejmutí obrazu kamerou je překonvertován na černobílý a následně je z něj vytvořen binární obraz pomocí Cannyho detektoru. Poté se provede Houghova transformace, ze které jsou detekovány předpokládané kraje vozovky. Úsečky získané Houghovou transformací, jejichž sklon je větší než nula, jsou rozděleny do dvou skupin podle sklonu. Protože se předpokládá lineární model, musí se sklon levé a pravé hranice vozovky lišit. Z toho se pak vychází při třídění úseček.

Výstupem Houghovy transformace jsou počáteční $(x0, y0)$ a koncové $(x1, y1)$ body úseček. Pak je možné vypočíst

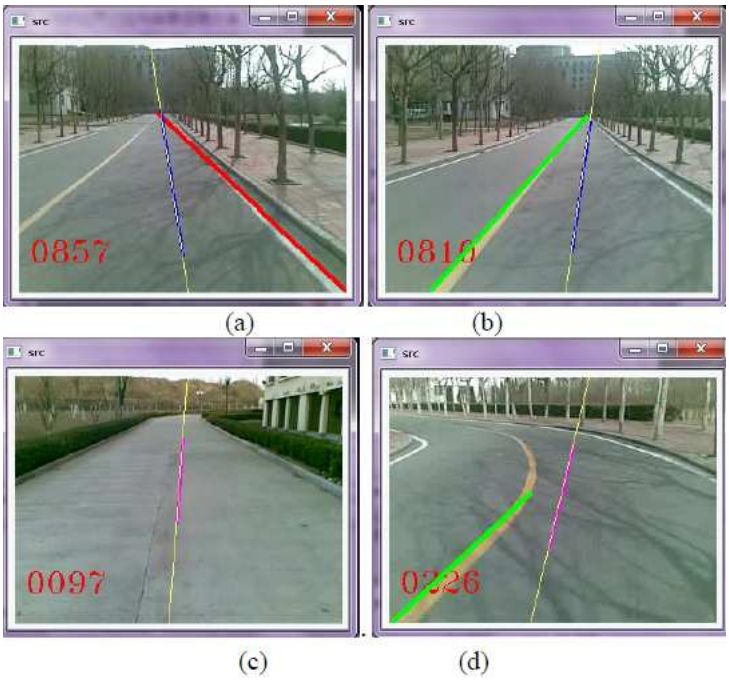
$$\mid \mathrm{tg}\, P \mid = \frac{(y1 - y0)}{(x1 - x0)} \tag{1}$$



Obrázek 7: Výsledek algoritmu Intelligent Vehicle (převzato z [8])



Obrázek 8: Vývojový diagram algoritmu (převzato z [9])



Obrázek 9: Výsledek algoritmu Lane detective (převzato z [9])

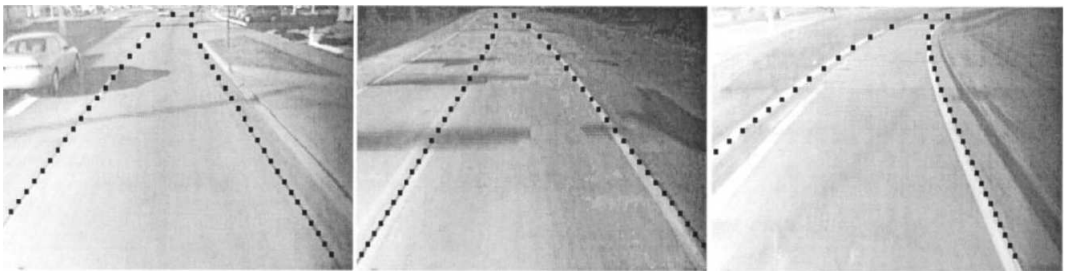
Pokud je $|\ \mathrm{tg}\, P\ | < 0.1$, předpokládá se, že je úsečka rovnoběžná s osou x. Takové se ovšem v lineárním modelu nevyskytují, jedná se tedy pravděpodobně o překážku na silnici a vyřadí se. Pokud je $|\ \mathrm{tg}\, P\ | > 0$, uloží se v levé skupině, jinak v pravé. Z levé a pravé skupiny se pak vypočítá centrální linie a zobrazí se. Budou existovat tři možnosti výpočtu:

- Existují obě hranice.
- Existuje pouze jedna hranice.
- Neexistuje ani jedna hranice.

Podrobnosti výpočtu je možno nalézt v [9], vývojový diagram pak na obrázku 8. Na obrázku 9 je pak vidět výsledek algoritmu v případě, že (a) byl nalezen pouze pravý okraj, (b) byl nalezen pouze levý okraj, (c) nebyl nalezen žádný okraj a (d) silnice zatáčí. Algoritmus byl testován na Pentiu 4 běžícím na 3MHz s 1GB RAM. Autoři využívají knihovnu openCV a byli schopni zpracovat 15 snímků za sekundu. Spolehlivost algoritmu je okolo 90%.

Výhody algoritmu:

- Jednoduchý algoritmus založený na openCV.
- Funguje i tam, kde chybí jedna nebo obě silniční čáry.



Obrázek 10: Výsledek algoritmu LANA (převzato z [10])

- Poměrně vysoká spolehlivost (90%).

Nevýhody algoritmu:

- Špatně funguje v ostrých zatáčkách.
- Potřebuje výkonný počítač, i s ním dosahuje rychlosti zpracování pouze 15 snímků za sekundu.
- Zjednodušený silniční model (předpokládá se pouze lineární model).

2.3.5 LANA: A Lane Extraction Algorithm that Uses Frequency Domain Features

LANA je starší algoritmus z roku 1999, uveřejněný v [10], založený na zpracování dat ve frekvenční doméně. Metoda je založena na sadě funkcí frekvenční oblasti (diskrétní Cosinova transformace), které zachycují důležité informace o síle a orientaci prostorových hran. Daný obraz je rozdělen na bloky 8x8 pixelů, a pak ortogonálně rozdělen na 64 DCT základních elementů. Každý z těchto prvků odpovídá prostorové doméně hrany určité síly a orientace. Z uvedených 64 prvků je diagonálně dominantních pouze 12 prvků. Všech 64 prvků je na obrázku 2a a 12 diagonálně dominantních je pak v matici na obrázku 2b. Algoritmus je tedy zaměřen převážně na diagonální hrany. Velmi působivý výsledek činnosti algoritmu je vidět na obrázku 10.

V závěru pak autoři porovnávají výsledky LANA s LOIS ([11]) včetně výkonového porovnání obou algoritmů. Zajímavé je porovnání, které proběhlo (v té době) na moderním počítači Pentium 266Mhz, s 96MB RAM. Testovací obraz byl VGA (640x480). Oba algoritmy prohledávaly přibližně stejný počet (400 000) možných tvarů dráhy, složených z devíti možných zakřivení, 50 různých směrů a 30 různých míst pro levý a pravý pruh zvlášť. Algoritmus LANA to zvládl v čase přibližně 30 sekund, zatím co LOIS to trvalo přibližně 2 hodiny. Přestože se ani v jednom případě nejedná o realtime zpracování, je evidentní, jaký výkonový pokrok udělalo zpracování obrazu v posledních deseti letech.

Výhody algoritmu:

- Zajímavý algoritmus založený na diskrétní Cosinově transformaci.
- Velmi přesně kopíruje dělící čáry.

- Velmi robustní i v případě detekce přerušovaných čar.

Nevýhody algoritmu:

- Reaguje především na diagonální hrany.
- Výpočetně velmi náročný, nelze použít pro zpracování v reálném čase.
- K detekci potřebuje krajnice a středovou čáru.

2.3.6 Lane boundary detection using statistical criteria.

Jedná se o jeden z mála algoritmů založený na statistických kritériích, uveřejněný v [4]. Vychází z význačného rozdílu mezi silničním a nesilničním povrchem. K detekci tohoto rozdílu využívají statistické kritéria druhého řádu. Nevýhoda statistických kritérií je, že jejich odhad je časově velmi náročný a vyžaduje velký výpočetní výkon. Pro zmenšení předpokládané oblasti se používá silniční model, který se neustále přizpůsobuje detekovaným hranicím silnice. Pro odhad parametrů silničního modelu se používá χ^2 test dobré shody s hyperbolickou funkcí silničního modelu.

Pro statistické kritéria používají:

- Energii.
- Kontrast.
- Homogenitu.
- Entropii.

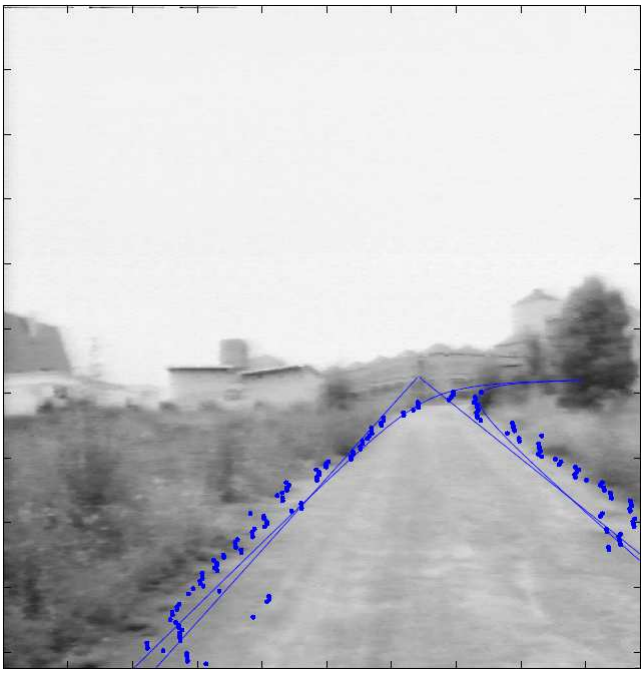
Tyto kritéria jsou znázorněna na obrázku 3. Jak již bylo řečeno, pro výpočet statistických kritérií druhého řádu je zapotřebí velkého výpočetního výkonu. Pro každý čtverec 8x8 bodů je nutno počítat matici pravděpodobnosti o rozměru N x N, kde N je počet úrovní šedi u černobílého obrazu nebo počet barev u barevného. Proto autoři zavedli silniční model, pro který se pak výrazně redukuje počet nutných výpočtů (omezí se pouze na oblasti předpokládané hranice silnice).

Pro další snížení výpočetní náročnosti hledají v každém řádku lokální extrémy (lokální minima pro energii a homogenitu a lokální maxima pro kontrast a entropii). Na tyto data pak použijí test dobré shody χ^2. Výsledek činnosti algoritmu je pak vidět na obrázku 11, na cestě v polích. Nevýhodou tohoto algoritmu je, že v době jeho vzniku (rok 1997) jej nebylo možno počítat v reálném čase.

Výhody algoritmu:

- Neobvyklé vyhodnocování založené na výpočtu statistických kritérií.
- Použitelné pro silnice bez dělících čar (venkovské a polní cesty).
- Využívá silniční model pro zrychlení výpočtu.

Nevýhody algoritmu:



Obrázek 11: Cesta v polích. Výsledek algoritmu statistických kritérií (převzato z [4])

- Autoři nepoužívají korelaci pro levý a pravý okraj silnice. To způsobuje nestabilitu algoritmu.
- Výpočetně velmi náročný, nelze použít pro zpracování v reálném čase.
- Použitelné pouze s dalšími metodami (například pro korekci dead reckoning).

3 Teorie detekčního algoritmu

V této kapitole uvedu jednotlivé části detekčního algoritmu tak, jak se vykonávají v průběhu zpracování obrazu. Předpokládám, že čtenář je obeznámen s teoretickými základy zpracování obrazu a proto při jejich popisu budu uvádět pouze nezbytnou teorii důležitou k pochopení uvedeného algoritmu. Pro případné zájemce uvádím zdroje, kde lze detailně nastudovat uvedenou problematiku. Dále pak budu uvádět části kódu nebo funkce, důležité pro tento algoritmus.

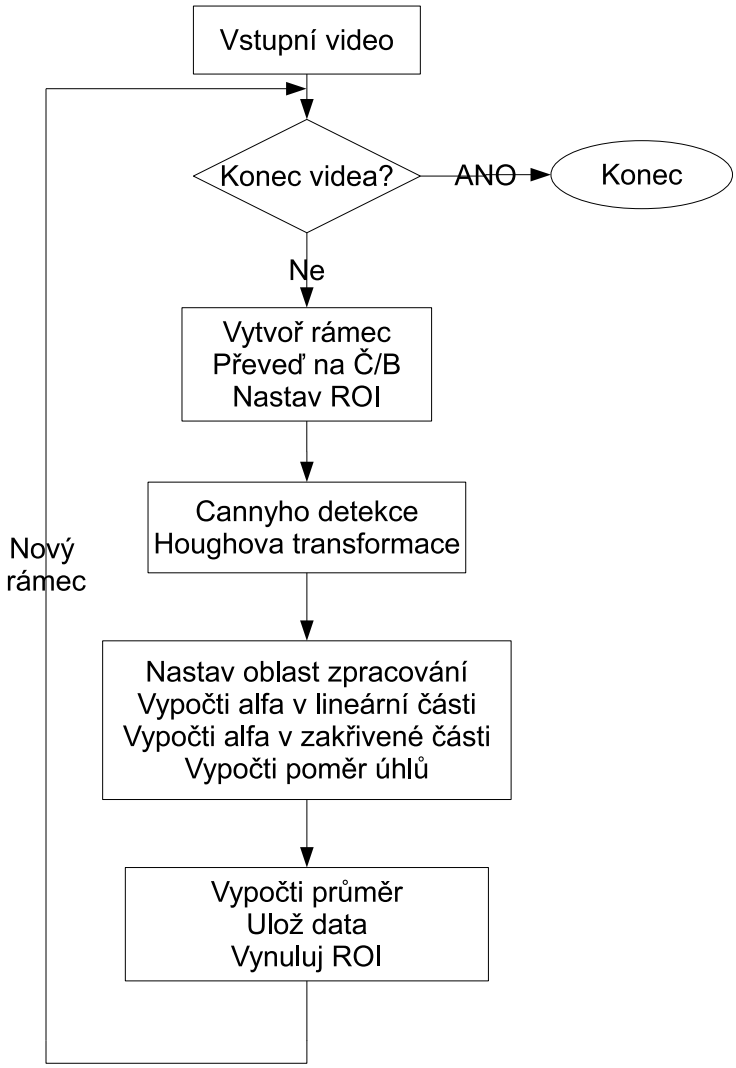
Většina uvedených funkcí jsou součástí standardní knihovny OpenCV, která je šířena pod BSD licencí a je volně ke stažení. Je to svobodná a otevřená multiplatformní knihovna pro manipulaci s obrazem. Je zaměřena především na počítačové vidění a zpracování obrazu v reálném čase. Tuto knihovnu vyvíjeli programátoři firmy Intel pro předvedení výkonu jejich procesorů a jako součást projektu zahrnující takzvaný real–time ray tracing. Poprvé byla uvedena v roce 2000, v roce 2006 byla uvedena první verze a v roce 2009 druhá verze, která je stále aktuální. Tato knihovna obsahuje přes 500 funkcí, které jsou optimalizovány na rychlost a spotřebu paměti. Dokáže se zrychlit spoluprací s knihovnou Integrated Performance Primitives (Intel IPP). Cílem této knihovny je, aby další vývojáři znovu „neobjevovali kolo“. Knihovny OpenCV jsou využívány prakticky po celém světě a webové stránky projektu zaznamenaly již více než 3 miliony stažení. Její výhodou je, že snadno komunikuje s kamerou a podporuje množství formátů pro obrázky a videa.

3.1 Stručný popis algoritmu

Algoritmus předpokládá vstupní videosekvenci z kamery. Ve vstupní sekvenci si musíme vyznačit ROI (Region Of Interest), tedy oblast zájmu s důležitými daty. Z videosekvence pak vyseparujeme jednotlivé snímky, které budeme zpracovávat. Ty je nutno převést na černobílý obraz, z něhož se pomocí hranové detekce vytvoří binární obraz. Protože však binární obraz je rastrový, který nám nedává žádnou informaci o směru a velikosti čar v obraze, je nutno z rastrového obrazu vytvořit vektorový. Tento obraz se může vyhodnocovat různými metodami. Já jsem zvolil statistické metody průměrováním, z důvodu jejich jednoduchosti a rychlosti. Vývojový diagram algoritmu je znázorněn na obrázku 12.

Dále je pak nutno filtrovat data, protože se můžou vyskytovat místa, ve kterých nelze spolehlivě detekovat vstupní obraz. To později ukážu na reálných příkladech. K tomu nám poslouží opět statistické metody, hezké příklady zpracování statistických dat můžeme najít například v [15]. Při zpracování obrazu je však nutno mít základní povědomí o operacích s obrazy. Ideálním studijním materiálem tak můžou být skripta *Matematické základy digitálního zpracování obrazu* [16] nebo [14], [17]

Na každý obraz máme ovšem limitovaný čas – přibližně 40 milisekund. Za tuto dobu musíme tedy zpracovat celý obraz a vyhodnotit jej. Proto se budu v závěru práce zabývat i časovou analýzou algoritmu, jeho spolehlivostí a návrhy na jeho zlepšení.



Obrázek 12: Vývojový diagram algoritmu.

3.2 Příprava obrazu

Při jízdě vozidla je dráha nacházející se před tímto vozidlem snímána kamerou a převáděna do digitálního streamu. Ve své podstatě se jedná o nepřetržitý tok diskrétních obrazů, které na sebe navzájem navazují. Z tohoto streamu je pak nutno tyto jednotlivé obrazy vyseparovat, protože budou dále zpracovávány každý zvlášť.

To se pak provádí pomocí volání funkcí knihovny openCV, jak je ukázáno na výpise 1. Na řádcích 03 až 08 se pokusíme otevřít vstupní videosekvenci (s názvem *test.avi*). Pokud pokus o otevření selže, vypíše se chybová hláška a ukončí běh programu. Na řádku 09 vyseparujeme aktuální rámec do struktury (obrazu) IplImage s názvem tst. Na následujícím řádku vytvoříme opět strukturu IplImage nazvanou src, se stejnou velikostí jako tst.

```
01.  void captureFrame(void)
02.  {
03.      CvCapture* capture = cvCaptureFromFile("test.avi");
04.      if (!cvGrabFrame(capture))
05.      {
06.          printf ("Could not grab a frame\n\7");
07.          exit (0);
08.      }
09.      IplImage* tst =cvRetrieveFrame(capture);
10.      IplImage* src = cvCreateImage(cvGetSize(tst), IPL_DEPTH_8U, 1);
11.  }
```

Výpis 1: Zachycení rámce z videosekvence pomocí openCV knihovny

Vzhledem k tomu, že veškeré zpracování a analýza obrazu probíhá digitálně, budeme tedy pro naše potřeby uvažovat diskrétní dvourozměrný obraz tak, jak jej obdržíme z kamery. Obraz bude buď černobílý, kde se uplatní jen jasová složka nebo barevný, nejčastěji se složkami RGB (můžou být i ve formě YUV nebo podobné, ale pro jednoduchost budeme předpokládat pouze RGB). Obor hodnot funkcí f černobílého obrazu, nebo jednotlivých složek barevného obrazu bývá obvykle z intervalu $< 0; 255 >$, ale může být i jiný. Přepokládejme tedy funkci černobílého obrazu

$$f_{bw}(x, y) \qquad \forall x \in < 0; x_{max} >, \forall y \in < 0; y_{max } > \tag{2}$$

Kde f_{bw} je hodnota jasové složky, která nabývá hodnot $< 0; f_{max} >$ a definičním oborem (x, y) jsou souřadnice všech bodů obrazu f_{bw}. Jak již bylo řečeno ve většině případů je $f_{max} = 255$. V případě barevného obrazu pak máme:

$$f_c(f_r(x, y), f_g(x, y), f_b(x, y)) \qquad \forall x \in < 0; x_{max} >, \forall y \in < 0; y_{max } > \tag{3}$$

kde definičním oborem x, y jsou stejně jako u černobílého obrazu souřadnice jednotlivých pixelů v obraze. Jednotlivé barvotvorné složky $f_r(x, y), f_g(x, y), f_b(x, y)$ jsou červená, zelená a modrá. Každá nabývá hodnot $< 0; f_{max} >$. Pro tzv. True Color (plně barevný obraz) je $f_{max} = 255$, pro každou jasovou složku. Z důvodů úspory paměti může být i menší, nebo pro každou složku jiný. Nyní tedy převedeme barevný obraz na černobílý. Použijeme funkci:

$$f_{bw} = (0.299 f_r + 0.587 f_g + 0.114 f_b) \tag{4}$$

Pokud budeme obraz snímat v černobílém režimu, není nutno provádět převod funkcí (4), čímž ušetříme strojový čas procesoru nutný pro převod každého pixelu na černobílý. Vzhledem k tomu, že se jedná o operace plovoucí čárce, bude časová náročnost funkce (4) značná. Pokud by přesto bylo nutno provádět uvedený převod, je výhodnější použít například čísla *int* a jednotlivé koeficienty násobit vhodnou konstantou (například 1024), což se dá v procesoru realizovat 10x posuvem vlevo (většina procesorů takové posuvy zvládá v jedné instrukci). Po dokončení výpočtu se pak s výsledkem provede stejný posuv vpravo. Dojde tím ke ztrátě desetinné části, nicméně výpočet bude proveden mnohem rychleji. Při použití knihovny OpenCV je možno také využít funkce z výpisu 2, řádek 3.

```
01.  void openCVfce(void)
02.  {
03.      cvCvtColor(src, img, CV_RGB2GRAY);
04.      cvSetImageROI(img, cvRect(10, 15, 150, 250));
05.      /* Zde proběhne zpracování obrazu */
06.      cvCanny( img, dst, 400, 800, 5 );
07.      CvSeq lines = cvHoughLines2( dst, storage, CV_HOUGH_PROBABILISTIC,
08.                                                  1, CV_PI/360, 30, 10, 10 );
09.      cvResetImageROI(img); //Vždy vynulujeme ROI
10.      cvThreshold(img, thrhold, 0, 255, CV_THRESH_BINARY | CV_THRESH_OTSU);
11.  }
```

Výpis 2: Použité funkce openCV knihovny

Kde *src* je zdrojový obraz (source), *img* je cílový obraz do kterého funkce uloží výsledek převodu a *CV_RGB2GRAY* je konstanta, která říká této funkci, že jde o převod z formátu RGB do odstínů šedi.

V tento okamžik máme tedy připraven černobílý obraz, na kterém vyznačíme oblast zájmu (takzvaný ROI – Region Of Interest). To provedeme pomocí OpenCV funkce *cvSetImageROI()*, které předáme obraz *img* a obdélník, který nás zajímá (*cvRect()*), viz výpis 2 řádky 4 a 9.

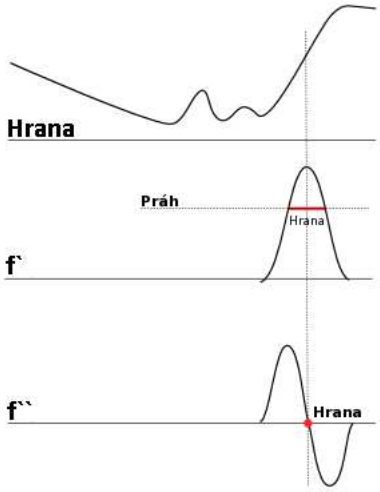
Pokud nás zajímá celý obraz, použijeme *cvSetImageROI()* na celý obraz, nebo lépe *cvResetImageROI()*.

3.3 Segmentace obrazu

Segmentací obrazu rozumíme operace prováděné s obrazem, za účelem rozpoznání objektů. V našem případě potřebujeme „dolovat“ objekty kolejí, nebo dělících čar a ostatní objekty „zahazovat“. Nejčastější způsob segmentace je prahování. První operací kterou tedy musíme provést, je hranová detekce. Tak dostaneme binární obraz. Ten má všechny body v popředí s hodnotou logická 1, všechny ostatní body (pozadí) mají hodnotu logická 0. Tedy

$$f_{bw}(x,y) = \begin{cases} 1, \text{pokud} & G(x,y) > E \\ 0, \text{pokud} & G(x) \leq E \end{cases} \tag{5}$$

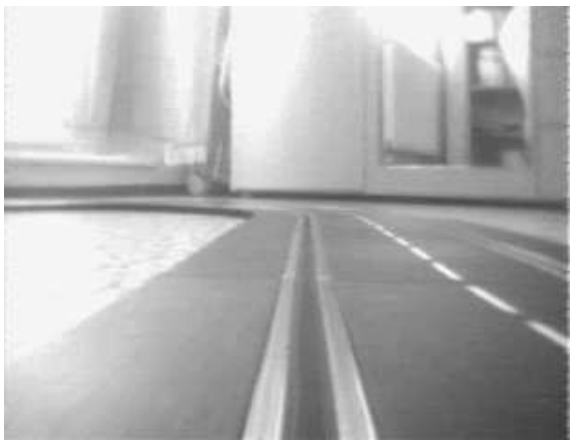
Kde E je takzvaný práh, tj úroveň jasu, nad níž považujeme bod za bod náležící popředí a pod níž považujeme bod za bod pozadí, f_{bw} je binární hodnota daného bodu a $G(x, y)$ je převodní funkce. Celé to názorně vidíme na obrázku 13, kde je nahoře vidět průřez hranou (průběh jasové funkce), pod ní je pak vidět první derivace této hrany a dole její druhá derivace. Při použití první derivace tedy hledáme lokální maximum, kdežto při druhé derivaci průchod nulou.



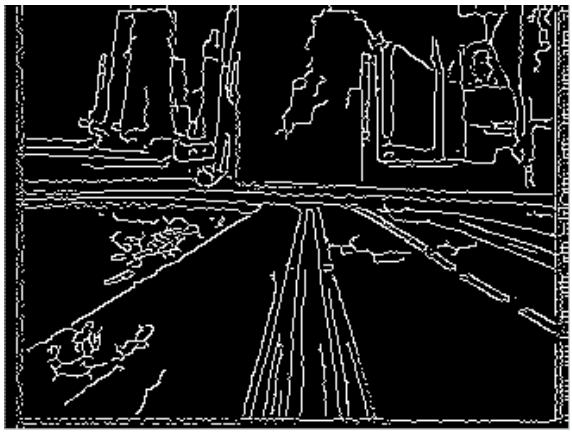
Obrázek 13: Hrana a její první a druhá derivace

Pokud aplikujeme funkci (5) na náš původní černobílý obraz, získáme požadovaný binární obraz. Funkce $G(x, y)$ pak může být některý typ hranového operátoru. Princip činnosti hranového operátoru a způsoby detekce hran jsou ukázány na obrázku 13. Často používaný je například Sobelův operátor (použitý v [8]), operátor Prewittové nebo Cannyho detektor [12], [13]. Dříve byly hranové detektory definováný více méně intuitivně. V roce 1986 definoval John F. Canny [12], [13] tři požadavky, které by měl splňovat ideální detektor hran (minimalizovat pravděpodobnost chybné detekce, najít polohu hrany co nejpřesněji a bod hrany identifikovat jednoznačně). Předpokládal, že hranový detektor bude založen na výpočtu konvoluce vstupního signálu s funkcí $f(x)$, kterou zatím neznáme. Výsledkem pak je aproximace funkce f pomocí první derivace gaussiánu, jehož hodnoty jsou jen o málo horší než hodnoty optimální. Jeho výhodou je ale snadný výpočet. Proto jsem zvolil tento hranový detektor. Použil jsem opět knihovní funkci OpenCV, na výpise 2 řádek 6, s parametry (*source, destination, lowThreshold, highThreshold, apertureSize*).

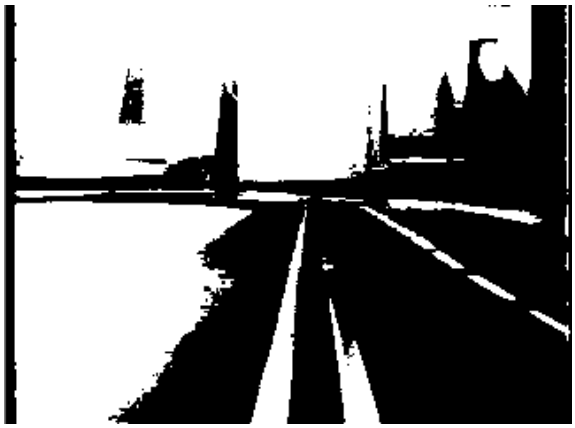
Na obrázku 14 máme vstupní obraz snímaný kamerou, výsledek Cannyho detekce je pak na obrázku 15. Pro porovnání uvádím ještě na obrázku 16 prahování obrazu pomocí funkce *cvThreshold()*, na výpise 2 řádek 10.



Obrázek 14: Vstupní obraz snímaný kamerou



Obrázek 15: Cannyho detekce vstupního obrazu



Obrázek 16: Threshold detekce vstupního obrazu

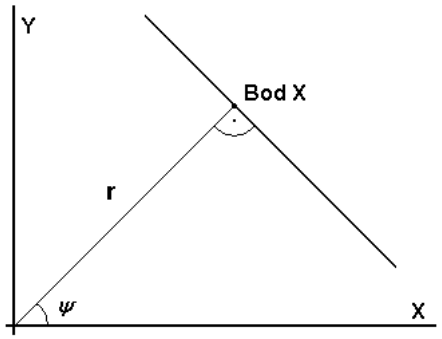
3.4 Rozpoznání obrazu

V této části se budu zabývat způsobem rozpoznání jednotlivých čar v obraze. Předpokládáme, že námi hledané čáry (koleje, vedení, dělící čáry na silnici) vedou ze spodní části obrazu vzhůru, kde různě zatáčejí (viz obrázek 14) a také se díky perspektivě sbíhají. Z předchozího bodu máme ovšem binární obraz (obrázek 15), který nám vytváří hrany pro každý bod zvlášť. Musíme tedy spojit jednotlivé body které spolu souvisí tak, aby vám vytvořily souvislé hrany (takzvané globální hrany) a zjistit jejich směr. K tomu využijeme Houghovu transformaci.

Houghova transformace se používá velmi často ve zpracování obrazu a slouží k rozpoznávání jednoduchých parametrizovatelných objektů, jako jsou přímky, úsečky a podobně. Každý bod v binárním obraze, který jsme obdrželi předchozími úpravami, je definován v prostoru souřadnic (x, y). Pro zmiňované nalezení přímky, která je tvořena jednotlivými body, potřebujeme provést transformaci. Přímka je v dvourozměrném prostoru definována několika způsoby, například směrnicový tvar přímky $y = kx + q$, kde k je směrnice přímky a q je posun v ose y. Pro nás je však výhodnější normálový tvar

$$y = \frac{x \cdot \cos(\psi)}{\sin(\psi)} + \frac{r}{\sin(\psi)} \qquad \sin(\psi) \neq 0 \tag{6}$$

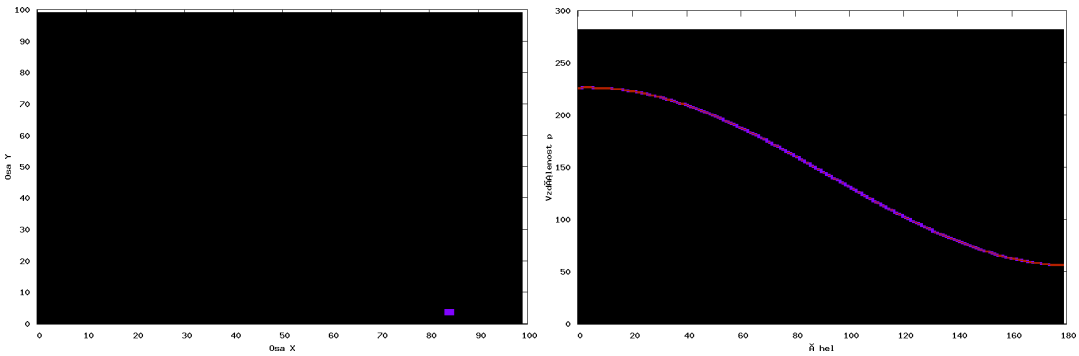
Kde r je vzdálenost bodu (x, y) od počátku O a ψ je velikost orientovaného úhlu kolmého na hledanou přímku (viz obrázek 17). Tento normálový tvar má výhodu, že je nezávislý na orientaci os.



Obrázek 17: Houghův prostor

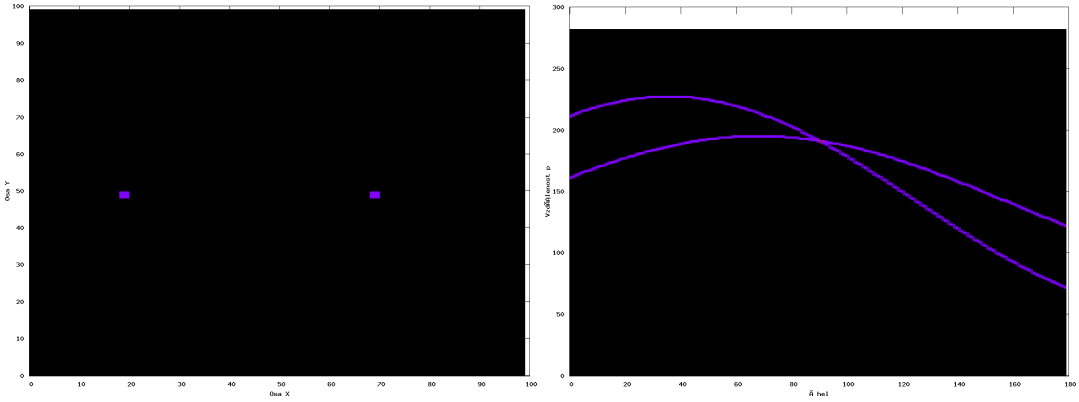
Nyní každý bod v obraze převedeme z prostoru souřadnic (x, y) do souřadnic (ψ, r) a daným bodem povedeme všechny možné přímky – tím nám vznikne sinusoida. Každý bod této sinusoidy tedy představuje jednu přímku procházející tímto bodem. Jak bude vypadat Houghova transformace pro jeden bod si můžeme prohlédnout na obrázku 18. Vlevo je bod v prostoru, vpravo všechny křivky proložené tímto bodem tvořící sinusoidu.

Je jasné, že pro každý bod budeme mít jednu křivku (sinusoidu) v Houghově prostoru. Pokud však leží dva body na stejné přímce, musí mít stejné parametry – jedná se o stejnou přímku, ze všech možných, které procházejí danými body. Bude se tedy jednat o průnik



Obrázek 18: Princip Houghovy transformace- jeden bod.

dvou sinusoid a ty se protnou právě v jednom bodě (ψ, r). Tento bod tedy definuje vektor hledané přímky. Jak to bude vypadat v praxi, je vidět na obrázku 19. Vlevo jsou dva body v prostoru, vpravo pak jejich sinusoidy, které se protínají v bodě přímky, na níž tyto body leží.

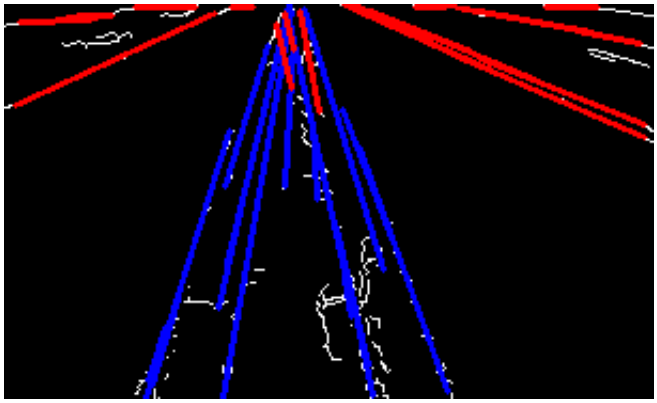


Obrázek 19: Princip Houghovy transformace- dva body.

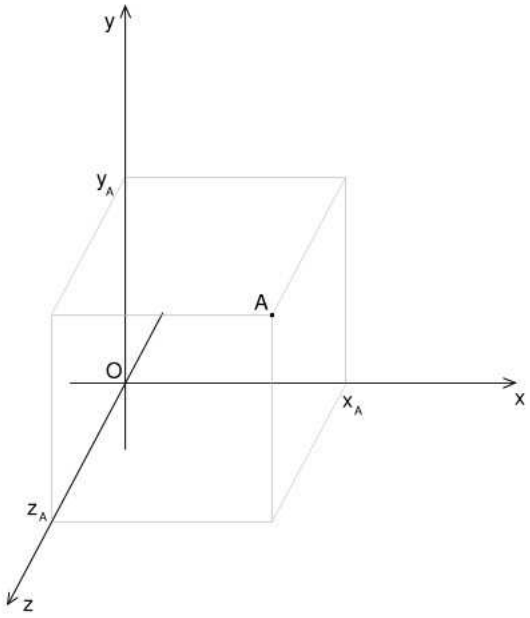
Na výpise 2, řádek 7 a 8 nám ukazuje použití Houghovy transformace v knihovně openCV. Datová struktura *CvSeq* nazvaná lines pak obsahuje jednotlivé úsečky (přímky), souřadnice jejich začátku a konce a řadu dalších údajů. Např. pomocí ukazatele *lines→total* získáme počet nalezených úseček v obraze. Na obrázcích 20 je pak ukázán výsledek Houghovy trasformace, kde jsou barevně (červeně a modře) zvýrazněny jednotlivé úsečky.

3.5 Normalizace

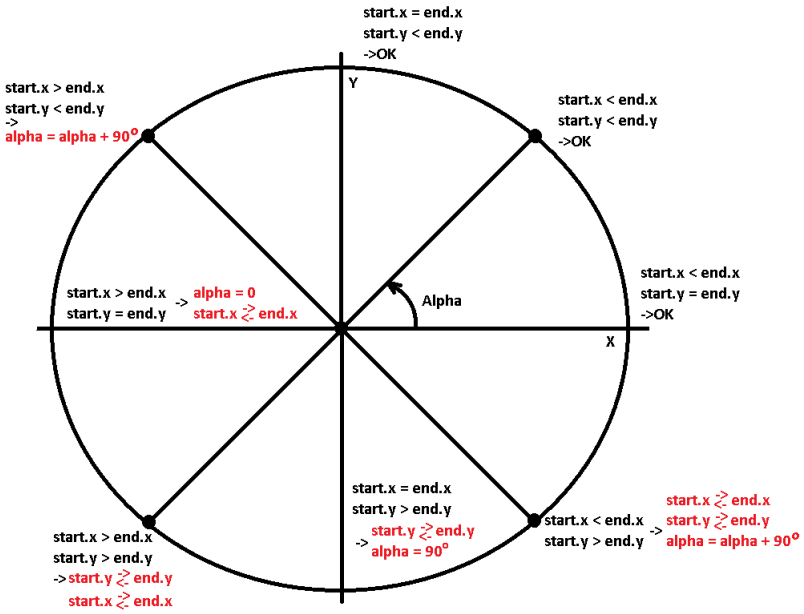
V běžném životě jsme zvyklí používat kartézský systém souřadnic (viz obrázek 21), to znamená, že ve 2D obraze máme 4 kvadranty po 90 stupních, proti směru hodinových ručiček a nula obou souřadných os (x,y) je v jejich průsečíku. Problém openCV spočívá v tom, že považuje za nulovou souřadnici levý horní roh, takže náš „běžný“ první kvadrant



Obrázek 20: Houghova transformace- přímý směr



Obrázek 21: 3D kartézský souřadný systém



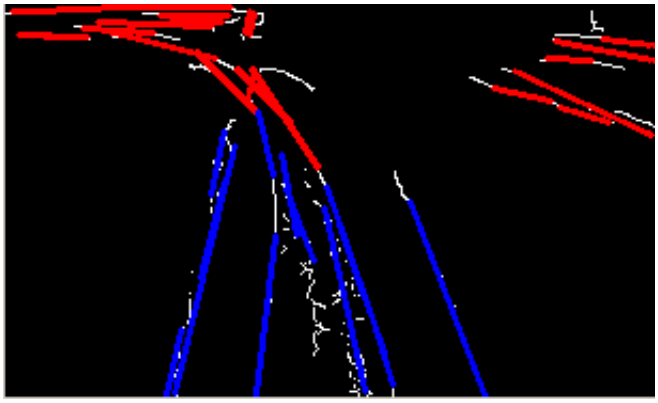
Obrázek 22: Normalizace souřadnic v openCV

se nám díky tomu promítne do čtvrtého. Díky tomu se nám pak může v openCV stát, že bude koncová souřadnice (x,y) úsečky Houghovy transformace menší (v jedné nebo obou osách) než její počáteční souřadnice. To by nám pak dělalo problémy při stanovení úhlu vektoru a při výpočtu průměrného úhlu. Musíme proto provést normalizaci souřadného systému, jak je naznačeno na obrázku 22.

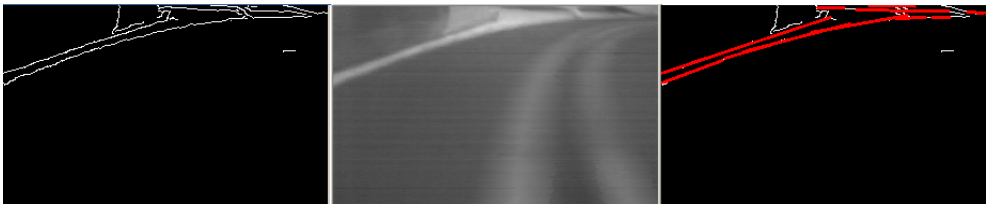
3.6 Analýza obrazu

V tento okamžik máme zpracovaný obraz, musíme jej však analyzovat. Cílem je zjištění, jestli se nacházíme na rovince nebo v zatáčce. Na základě toho pak bude možno provádět řízení vozidla tak, jak by je řídil člověk. Předpokládejme, že jedeme po rovince. Rychlost není v podstatě nijak omezená, můžeme jet (v rámci možností, předpisů, atd.) libovolně rychle. Jakmile se začne blížit zatáčka, musíme na ni reagovat snížením rychlosti tak, aby jsme ji mohli bezpečně projet. Při pohledu na obrázek 23 jednoduchou úvahou zjistíme, že zřejmě snadné bude testovat poměr úhlů mezi „modrými“ a „červenými“ úsečkami Houghovy transformace.

Vycházíme přitom z toho, že pokud jedeme po rovince, je spodní část obrazu právě ta rovinka, která nám zbývá do začátku zatáčky. Teoreticky by měly mít úsečky této rovinky úhel $90°$, ale díky perspektivě budou mít tento úhel mírně odlišný. Prakticky se pohybuje mezi $70°$ a $80°$. V místě, kde se začne rovinka měnit na zatáčku, dojde ke změně úhlu úseček. Je však nutné počítat s tím, že můžeme průměrovat pouze úsečky blízkých



Obrázek 23: Houghova transformace- detekce v levé zatáčce



Obrázek 24: Houghova transformace- chybná detekce v zatáčce

objektů (v našem případě vodící slot a napájecí koleje), protože díky perspektivě budou mít vzdálenější objekty výrazně odlišný úhel, i když se jedná o objekty stejného směru (typicky kraje dráhy).

Vzhledem k tomu, že při zpracování obrazu dochází často k rušení a různým anomáliím viz například obrázek 24. Na něm je vidět úplná ztráta obrazové informace v důsledku špatných světelných podmínek. Vlevo je binární obraz získaný Cannyho detekcí, uprostřed je pro porovnání obraz sejmutý kamerou a vpravo pak výsledek Houghovy transformace. Je vidět, že zde nejsou žádné smysluplné informace k analýze obrazu, proto je nutno eliminovat takové neobvyklé údaje. Abychom tedy dosáhli co nejlepších výsledků, které nejsou ovlivněny takovými chybnými údaji, použijeme vážený průměr nejprve pro výpočet „modrých“ a pak pro výpočet „červených“ úseček.

Než se k tomu ovšem dostaneme, budeme muset najít slot ve spodní části obrazu. V jeho okolí se pak budou nacházet objekty napájecích kolejí. Budeme vycházet z obrazu rozděleného na čtverce 8x8 bodů, na kterých provedeme funkci *cvCountNonZero()* a výsledek uložíme do dvourozměrného pole. Každý čtverec obsahující aktivní body bude označen číslem. Čtverce, které neobsahují žádný bod jsou označeny jako nulové. Slot a napájecí koleje jsou význačné uskupení objektů v obraze (viz například obrázek 15). Budeme tedy hledat takové objekty (nenulové čtverce 8x8 bodů), které mají ve své blízkosti další objekty (nenulové čtverce). Pro výpočet takového seskupení objektů využijeme

vážený průměr pozice daného čtverce a převrácené hodnoty vzdálenosti jeho nejbližšího souseda. Tím získáme pozici slotu (střed takového význačného uskupení), který bude minimálně ovlivněn odlehlým pozorováním (váha odlehlého pozorování je převrácenou hodnotou jeho vzdálenosti). Na výpise 3 je pak vidět vlastní postup výpočtu pozice slotu. Výpočet váženého průměru pro nalezení pozice slotu provedeme následovně:

$$P_{slot} = \frac{\sum_{i=1}^{n} P_i \frac{1}{D_i}}{\sum_{i=1}^{n} \frac{1}{D_i}} \qquad D_i > 0, P_i \in < 0; y_{max} > \tag{7}$$

Kde P_{slot} je pozice slotu, obor hodnot $P_{slot} \in < 0; y_{max} >$. P_i je pozice nenulových čtverců, D_i je vzdálenost P_i od nejbližšího dalšího nenulového čtverce. Číslo $n = y_{max}$, což je počet všech čtverců v ose y (i těch nenulových). Pro čtverce 8x8 bodů pak platí $n = \frac{y_h}{8}$, kde y_h je horizontální velikost obrazu.

```
01.     //********* FILL ARRAY OF 8x8 RECTANGLES AND PROCESS ZOOMED IMAGE
    ********//
02.     int i, j;
03.     double sum = 0;
04.     int count = 0;
05.
06.     for(j = 0; j < dst->height; j += 8)
07.     {
08.
09.         for(i = 0; i < dst->width; i += 8)
10.         {
11.
12.             cvSetImageROI(dst, cvRect(i, j, 8, 8));
13.             int white = cvCountNonZero(dst);
14.             arr[i/8][j/8] = white;                    // save 2D array of 8x8 rectangles
15.             cvRectangle(track, cvPoint(i, j), cvPoint(i + 8, j + 8), CV_RGB(white<<3, white
    <<3, white<<3), CV_FILLED, 8, 0);
16.             cvResetImageROI(dst);
17.         }
18.         // printf ("\n");
19.     }
20.
21.     //********* FIND SLOT AT LAST ROW IN PICTURE **********//
22.     arraySize = cvPoint(i/8, j/8);
23.     double distance = 0;
24.     for(int i = 0; i < arraySize.x; i++)
25.     {
26.         if(arr[i][arraySize.y - 1])
27.         {
28.             int temp = arraySize.x;
29.             for(int j = 0; j < arraySize.x; j++)
30.             {
31.                 if(arr[j][arraySize.y - 1])
32.                 {
33.                     int dist = (abs(arr[j][arraySize.y - 1] - arr[i][arraySize.y - 1]));
34.                     if(dist && dist < temp) temp = dist;
35.                 }
36.             }
```

```
37.             double fl = 1.0/(temp);
38.             sum += ((double)i * fl);
39.             distance += fl;
40.             count++;
41.         }
42.     }
43.
44.     int avg = (int)((sum/distance)*8.0 + 0.5);
45.     int rectNum = avg / 8;
```

Výpis 3: Výpočet pozice slotu v obraze

Vlastní výpočet váženého průměru Houghovy transformace provedeme následovně:

$$\alpha_{avg} = \frac{\sum_{i=1}^{n} \alpha_i c_i}{\sum_{i=1}^{n} c_i} \qquad \sum_{i=1}^{n} c_i > 0 \tag{8}$$

Kde α_{avg} je požadovaný vážený průměr, obor hodnot $\alpha_{avg} \in < 0; 180)$. Dále α_i je úhel dané úsečky, c_i je délka úsečky $c_i \in < 0; O_d >$ a n je počet úseček v dané oblasti (v okolí vodícího slotu a kolejí). O_d je diagonální rozměr vstupního obrazu a definiční obor $\alpha_i \in < 0; 180)$. Pokud je poměr úhlů roven jedné, jsou stejné a je jasné, že jedeme stále po rovince. Pokud se začne poměr úhlů měnit (podle toho, jestli se blíží levá, nebo pravá zatáčka bude úhel buď větší nebo menší než jedna), bude následovat zatáčka. Po jednoduché úvaze je ovšem možno celý výpočet zjednodušit, protože úhel úseček ve spodní části obrazu („modrých“) je konstantní, proto jej stačí spočítat pouze jednou a počítat pouze průměr vzdálenějších úseček.

3.7 Řízení

Nyní máme veškeré potřebné informace potřebné pro řízení vozidla. Typická jízda člověka autem po silnici probíhá následovně (předpokládáme vozidlo s předním náhonem, nebo pohonem 4x4):

- Řidič jede po dlouhé rovince, zvyšuje rychlost (fakticky je omezen pouze předpisy).
- V dálce vidí zatáčku, zpomalí tedy na vhodnou rychlost pro průjezd zatáčkou (Zkušený řidič brzdí vždy před zatáčkou).
- Při průjezdu zatáčkou v její druhé části může mírně akcelerovat.
- Za zatáčkou vidí opět rovinku, proto opět zvyšuje rychlost.
- Činnost řidiče nezávisí na směru zatáčky (je–li levá nebo pravá).

Protože máme již v podstatě všechny informace k dispozici, je možno implementovat řízení vozidla, tj nějaký regulátor, k řízení rychlosti. Takových regulátorů je možno najít na webu poměrně značné množství, navíc nesouvisí se zpracováním obrazu, proto se jím nebudu v této části zabývat. Co je však důležitější, je rychlost odezvy a spolehlivost dat.

Rychlost odezvy závisí na počtu snímků za sekundu, které získáme z kamery a následně je zpracujeme. S tím souvisí doba zpracování obrazových dat, ve které se nejdéle trvá Houghova transformace. Dobu zpracování dat je možno ovlivnit velikostí obrazu-čím menší obraz, tím rychlejší zpracování a tím větší počet snímků za sekundu. To na druhou stranu znamená snížení rozlišení a následně snížení spolehlivosti detekce.

Protože se však obraz mění postupně, nikoliv skokově, je řešením zavedení průměrování obrazů s předchozími snímky. Pokud je v obraze a_i rovinka, může být v obraze a_{i+1} opět rovinka, nebo většinu obrazu rovinka a v horní části začátek zatáčky. Pokud je v a_{i+1} rovinka a v horní části zatáčka, bude v a_{i+2} opět ve spodní části rovinka, ale v horní části bude větší část zatáčky. Zavedením plovoucího průměru pak odstraníme krátkodobé výpadky nebo chybnou detekci v jednotlivých obrazech. Nevýhodou bude ovšem zpoždění, které tím do systému vneseme a tedy delší reakce na zjištění zatáčky.

Pro naše účely se pak hodí plovoucí průměr s exponenciálním zapomínáním, protože si nemusíme pamatovat k hodnot a počítat z nich průměr, navíc díky tomu, že nejnovější hodnota má největší váhu, ovlivňuje nejvíce nově tvořený průměr. Budeme tedy počítat:

$$avg_{float} = avg_{float} + \frac{1}{k}(\alpha_{avg} - avg_{float}) \qquad k > 0 \tag{9}$$

Kde avg_{float} je požadovaný plovoucí průměr, jehož obor hodnot je $\alpha_{float} \in < 0; 180)$. Pak α_{avg} je úhel alfa (vážený průměr) nejnovějšího obrazu podle výpočtu 8 s definičním oborem $\alpha_{avg} \in < 0; 180)$ a $k, k > 0$ je korekce, tedy kolik obrazových dat chceme zahrnout do našeho výpočtu. Čím vyšší hodnota, tím víc obrazů bude v našem průměru obsaženo a data budou tedy „hladší“, ale tím více bude naše hodnota „zaostávat, plavat“ za skutečnou hodnotou. Výsledek takového „plavání“ je vidět na obrázcích 34 pro $k = 6$ a 35 pro $k = 12$. Díky tomu dochází k pozdní detekci zatáčky a v některých případech dokonce k detekci zatáčky až v okamžiku, kdy se v ní vozidlo nachází. Pro praktické využití je dostatečná hodnota 4, maximálně 8.

```
01.  int cumulAlpha = 0;
02.  int cumulLen = 0;
03.  for(int i = 0; i < lines->total; i++)
04.  {
05.      int tempX1 = lanes[i].start.x / 8;
06.      int tempX2 = lanes[i].end.x / 8;
07.      int tempY1 = lanes[i].start.y / 8;
08.      int tempY2 = lanes[i].end.y / 8;
09.      {
10.          if ((tempX1 < right && tempX2 > left) || (tempX2 < right && tempX1 > left))
11.          {
12.              int al = lanes[i].alpha;
13.              if (al > 90) al = 180 - lanes[i].alpha;
14.              cumulAlpha += (al * lanes[i].c);
15.              cumulLen += lanes[i].c;
16.          }
17.      }
18.  }
19.  int res = 0;
20.  if (cumulLen) res = cumulAlpha / cumulLen;
```

```
21.    avgAlfa = (int)((float)avgAlfa + ((1.0/3.0) * ((float)res - (float)avgAlfa)));
```

Výpis 4: Použité funkce průměrování dat

Na výpise 4 pak vidíme praktickou realizaci jak váženého průměru (řádky 01 až 18), tak plovoucího průměru. U váženého průměru je také vidět způsob výběru dat z pole struktur *lanes*, definovaných ve výpise 5. V této struktuře jsou uloženy všechny informace k příslušné úsečce, ze kterých jsou pak počítány potřebné data. Pole struktur *lanes* je naplněno ihned po provedení Houghovy transformace v průběhu vyčítání *lines*. Protože máme k dispozici pouze počáteční a koncové souřadnice (x, y) každé úsečky, musíme provést pomocné výpočty, jako je výpočet délky úsečky *lanes[index].c* nebo výpočet úhlu α (označený jako *lanes[index].alpha*)

$$\alpha = \frac{180}{\pi}(\arcsin\frac{a}{c}) \tag{10}$$

Kde α je úhel úsečky ve stupních, a je délka úsečky v ose x a c je vypočtená délka úsečky. Následně se provede také normalizace popsaná v předchozím odstavci. Protože pole má pro každý obraz jinou délku, uloží se také skutečná délka pole (to znamená ukazatel za poslední strukturu v poli).

```
01.  struct lane
02.  {
03.      int       position;
04.      CvPoint   start;
05.      CvPoint   end;
06.      int       a;
07.      int       b;
08.      int       c;
09.      int       alpha;
10.  } lanes[128];
```

Výpis 5: Struktura dat Houghovy transformace

4 Testování algoritmu

Pro testovací účely jsem vytvořil dráhu podle obrázku 25 (nebo 36 v příloze). Směr dráhy a pozice rámce 0 je v sejmutém videu zvýrazněn. Protože máme konstantní snímání rámců (fps = 25), můžeme bez újmy na obecnosti prohlásit jednotlivé rámce za jednotku času (1 rámec = 40 ms). Grafy dráhy jsou proto udávány v rámcích. Perida grafu vychází z délky okruhu a je 200 rámců (8 sekund). V testovacím videu pak projíždí autíčko cílovou páskou v rámcích 48, 248, 447 a 648 (viz přiložené CD). Video má celkovou délku 834 rámců. Při sledování videa je vidět velký vliv světelných podmínek na kvalitu videa, což mělo za následek chyby detekčního algoritmu.

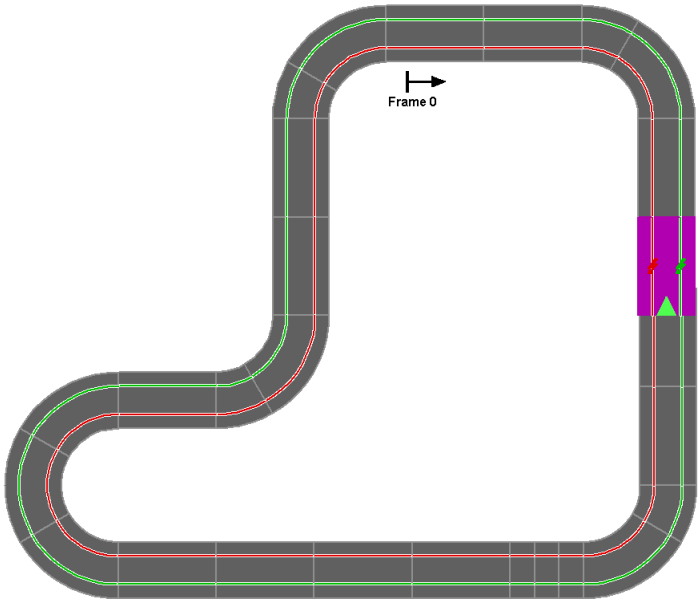
Z testovací videosekvence byl vytvořen graf, tak jak by měl detekční algoritmus vypočítat dráhu trati. Do stejného grafu pak byl vynesen výstup algoritmu pro porovnání výsledků (viz graf na obrázku 26 a podrobněji viz příloha A). Jak je uvedeno výše, algoritmus porovnává vážené průměry úhlů v dolní části obrazu (rovinka) a v horní části (potencionální zatáčka). Pokud je úhel v horní části menší než 50 stupňů (přičemž 90 stupňů je vertikální osa, tj fakticky rovinka), dochází k detekci zatáčky. Tento graf je přímo řízený, tedy nemá plovoucí průměrování rozebírané v předchozí kapitole.

Z grafu na obrázku 26 je vidět, že občas dochází k detekci zatáčky i v místě kde není, například při protisvětle, kde na dráhu není dobře vidět. V takovém případě algoritmus zareaguje snížením rychlosti, což není nic neobvyklého, protože člověk by ve takové situaci reagoval stejně – pokud dobře nevidí, sníží rychlost na bezpečnou úroveň. Další detekce zatáčky nastane na cílové rovince, což je mnohem nepříjemnější – ty bývají vždy poměrně dlouhé (ne–li nejdelší), takže zde by mělo autíčko jet plnou rychlostí. Z Houghovy transformace (viz obrázek 27) je jasné, proč ji algoritmus považuje za zatáčku – skutečně má takové parametry. Ale cílová rovinka má však také jednoznačně identifikovatelný layout (bílé pásy po stranách a při průjezdu cílem šachovnicovou čáru (která má na svědomí chybnou detekci zatáčky).

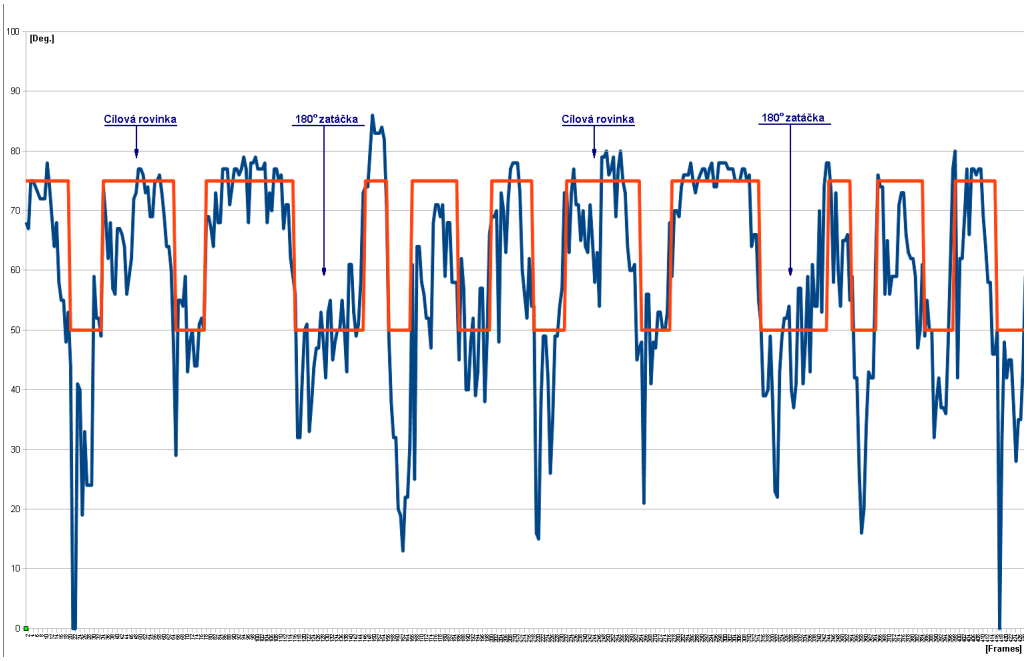
4.1 Časový rozbor algoritmu

Jak bylo již dříve řečeno, je nutno, aby algoritmus prováděl výpočty co nejrychleji. Čím rychleji je totiž provádí, tím více snímků za sekundu je možno pořizovat a tím přesnější máme detekci. Pokud totiž bude narůstat rychlost autíčka, může dojít k situaci, kdy ujetá dráha bude delší, než ta, která je zobrazena na předcházejícím snímku. Tím by byla porušena kontinuita snímků, ze které jsme vycházeli (o postupném nasouvání zatáčky v horní části obrazu) a současně by nemuselo k detekci zatáčky dojít, nebo by k ní došlo příliš pozdě. Pokusil jsem se tedy měřit časouvou náročnost jednotlivých kroků algoritmu. Měření jsem prováděl na stolním počítači s procesorem CORE2DUO T9400, 2,53 GHz, osazeným 4GB RAM. Dále uvádím typické a kde je to možné i minimální a maximální hodnoty. Z následujících výsledků je zřejmé, že časová náročnost algoritmu je poměrně proměnlivá.

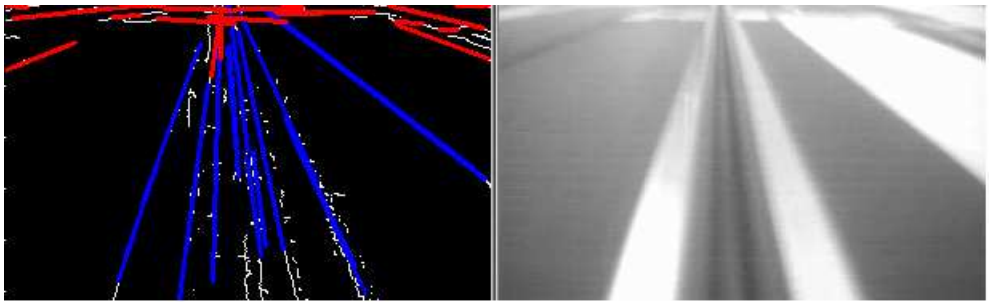
Nejvíce času zabírá Houghova transformace – průměrně 30 ms. Minimální čas je 12,7 ms a maximální pak 77,5 ms. Cannyho detekce je o řád rychlejší, zabírá průměrně 4 ms. Minimální čas je 2,8 ms a maximální 6,5 ms. Příprava obrazu (převod do odstínů



Obrázek 25: Testovací dráha



Obrázek 26: Graf testovací dráhy



Obrázek 27: Houghova transformace při průjezdu cílovou rovinkou

šedi, nastavení ROI, atd.) trvá průměrně 0,3 ms. Minimální čas je 0,2 ms a maximální 0,7 ms. Ostatní výpočty probíhají průběžně a jsou tak prakticky neměřitelné. Praktické výsledky měření pro jednotlivé funkce v každém rámci je možno vidět v grafu (obrázek 28 a podrobněji 40).

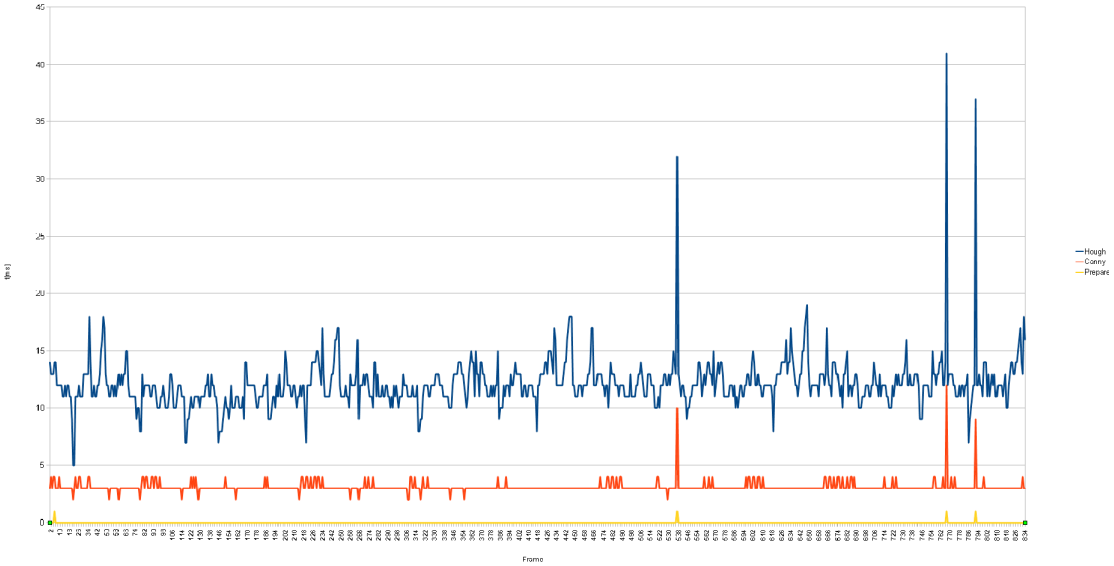
Tyto hodnoty neposkytují příliš prostoru na snížení časové náročnosti algoritmu, tak aby bylo možno zvýšit počet zpracovaných snímků za sekundu. Jediným možným řešením je zmenšení obrazu.

4.2 Odolnost algoritmu

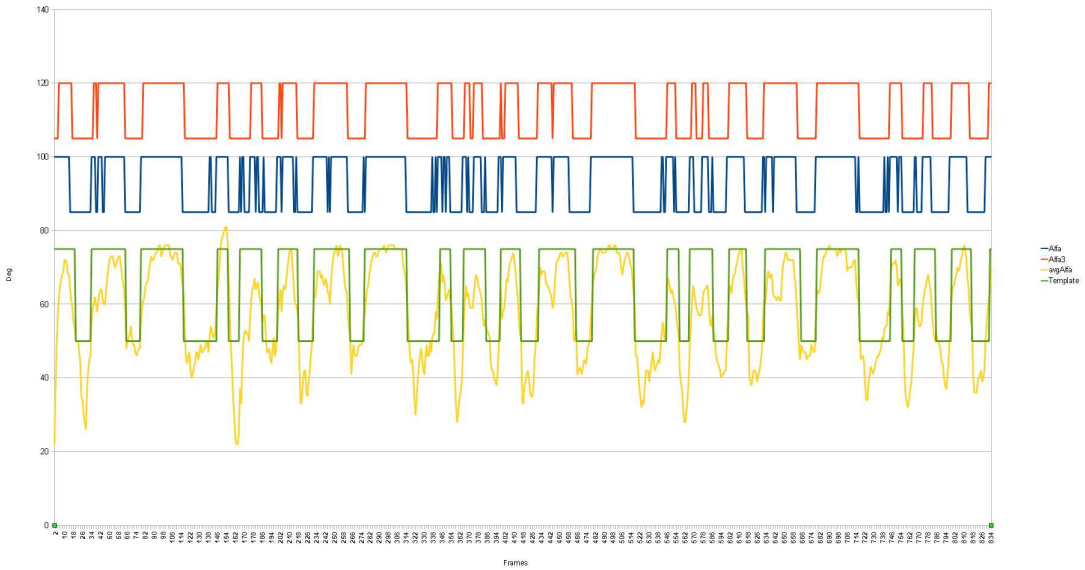
V testovacím videu je několik úseků se špatným světlem (ostré protisvětlo, podobné oslnění řidiče zapadajícím sluncem) nebo rozmazané úseky v důsledku slabého světla a nekvalitní kamery (kamera se pokouší zlepšit citlivost prodloužením závěrky, což při rychlém pohybu vede k rozmazání obrazu). V prvním případě dochází stále k poměrně spolehlivé detekci, kdy algoritmus stále reaguje na rovinky nebo zatáčky, i když s menší citlivostí. V druhém případě dojde k výpadku detekce (na obrázku 24). Výpadek je však tak krátký (pouze jeden snímek), že neovlivní funkci algoritmu. Takové chyby jsou pak zachyceny a ošetřeny statisticky, v tomto případě plovoucím průměrem.

Ten předpokládá, že změny v obraze jsou plynulé (že obraz neobsahuje výrazné změny), proto se snaží novou hodnotu průměrovat s předchozími hodnotami. Pokud by však takový výpadek trval déle, mohl by nepříznivě ovlivnit funkci algoritmu. Řešením by bylo v případě ztráty orientace použít předcházející snímek doplnění o předpokládanou změnu (posun obrazu).

Provedl jsem několik testů, jak ovlivní plovoucí průměr spolehlivost algoritmu. Na obrázcích (29a podrobněji pak v 41) je vidět jak vyhodnocuje testovací dráhu algoritmus bez plovoucího průměru, s plovoucím průměrem o $k = 3$ a jak by měl být vyhodnocen v ideálním případě. Graf je záměrně posunut na hodnoty 85 až 100 pro *alfa* a 105 až 120 pro *alfa3*, aby je bylo možno odlišit. Při stejných hodnotách by se křivky překrývaly a graf by byl nečitelný. Grafem je pro zajímavost opět proložena hodnota *avgAlfa* (tedy výsledek algoritmu s plovoucím průměrem o $k = 3$). Dále jsem všechny hodnoty porovnal a v tabulce 1 je vidět výsledek včetně procentuální spolehlivosti detekce. Je zřejmé, že plo-



Obrázek 28: Graf časové složitosti



Obrázek 29: Graf detekce zatáček pro plovoucí průměr s k=3 a bez něj

Algoritmus	**Správně hodnocených rámců**	**Celkem rámců ve videu**	**Procent**
Vzor	835	835	100
Alfa	707	835	85
Alfa3	748	835	90

Tabulka 1: Tabulka úspěšnosti různých verzí algoritmů

voucí průměr pozitivně ovlivní chování algoritmu. Pro vyšší index k se pak spolehlivost detekce zvyšuje, ale to zase s sebou přináší problémy popsané výše.

5 Hodnocení algoritmu

5.1 Předpoklady pro funkci algoritmu

Pro funkci algoritmu tedy potřebujeme počítač s operačním systémem, na kterém poběží knihovna openCV. V úvahu tedy připadá systém Windows nebo Linux. Protože se předpokládá použití v embedded aplikacích, přichází v úvahu pravděpodobně Linux, protože tak výkonné procesory pro embedded aplikace jsou téměř výhradně architektury ARM, na které ovšem dnešní Windows neběží (nebereme na zřetel Windows for Mobile, nebo některé ze starších Windows CE). To s sebou přináší další komplikace, protože openCV byla původně vyvíjena Intelem, pro demonstraci schopností jejich procesorů právě při grafických operacích. Využívá tedy specializované instrukce jejich procesorů, což při přechodu na jinou architekturu může znamenat výrazný výkonový pokles. Její použití má však značné výhody

- Je velmi důkladně optimalizována, částečně psána v assembleru nebo v C.
- Odstiňuje programátora od low level funkcí – stačí pouze využívat odladěné hotové funkce.
- Usnadňuje a urychluje práci.
- Udržuje ji tým vývojářů, kteří poskytují rozsáhlou a rychlou podporu.
- Na webu je možno najít řadu příkladů a tutoriálů, usnadňujících s ní práci pro začátečníky.

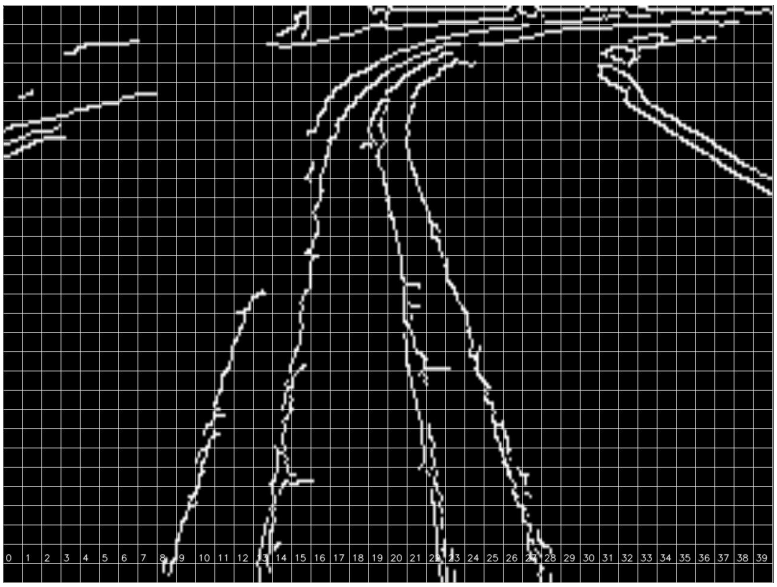
Má ale také nevýhody, o kterých jsem se již zmiňoval. V některých případech je tedy na zvážení, jestli nepoužít proprietární funkci nebo algoritmus, který by mohl být v konečném důsledku rychlejší.

```
01.     /* use sobel to find derivatives */
02.     cvSobel( src, sobel_x, 1, 0, 3);
03.     cvSobel( src, sobel_y, 0, 1, 3);
04.
05.     /* Convert signed to unsigned 8*/
06.     cvConvertScaleAbs( sobel_x , dest_x, 1, 0);
07.     cvConvertScaleAbs( sobel_y , dest_y, 1, 0);
08.
09.     //******** FILL ARRAY OF 8x8 RECTANGLES AND PROCESS ZOOMED IMAGE
    *********//
10.     int i, j;
11.     unsigned char arr[128][64];
12.     CvPoint arraySize;
13.
14.     for(j = 0; j < dst->height; j += 8)
15.     {
16.
17.         for(i = 0; i < dst->width; i += 8)
18.         {
```

```
19.             cvSetImageROI(dst, cvRect(i, j, 8, 8));
20.             int white = cvCountNonZero(dst);
21.             arr[i/8][j/8] = white;                    // save 2D array of 8x8 rectangles
22.             cvResetImageROI(dst);
23.         }
24.     }
25.     arraySize = cvPoint(i/8, j/8);
```

Výpis 6: Další funkce navrhované pro zlepšení algoritmu

5.2 Návrhy na zlepšení



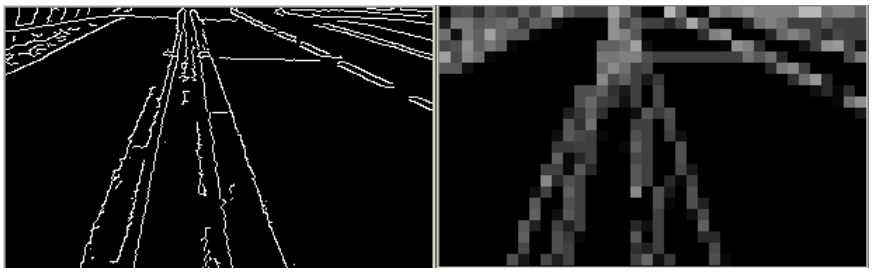
Obrázek 30: Binární obraz, rozdělený na čtverce 8x8 pixelů

V průběhu ladění algoritmu jsem přišel na řadu myšlenek, které se pokusím zapracovat do dalších verzí. Nejdůležitější myšlenka je vynechání Houghovy transformace – důvodem je její velká časová náročnost, která limituje výkon celého algoritmu. Jednou z variant je nahrazení transformace logickým součinem (operací AND) malými částmi obrazu (typicky 8x8 pixelů), kde budeme pomocí vzorů vyhledávat dominantní směr v daném čtverci.

Díky této myšlence bude možno pro každý čtverec 8x8 pixelů provést jen několik operací AND (které jsou velmi rychlé, z důvodu nativní podpory v každém moderním procesoru – pro takový vstup se jedná typicky jen o několik strojových cyklů). Celý QVGA obraz o rozměrech 320x240 bodů se pak rozdělí do 1200 čtverců 8x8 (což je dvojrozměrné pole o 30x40 prvcích). Každý prvek ponese informaci o dominantním směru čar v něm umístěných. Tím pak bude možno velmi rychle vytvořit vektor dráhy, navíc nebude nutno

zpracovávat všechny čtverce, ale jen ty, které obsahují nějaké (nebo určitý počet, typicky alespoň 8) aktivních pixelů (viz obrázek 30 a detailněji pak na obrázku 39).

Na výpise 6, řádky 10 až 25 je pak vidět způsob zpracování vstupního obrazu do čtverců 8x8 bodů. Tyto čtverce jsou pak uloženy ve dvourozměrném poli *arr* a jeho velikost v proměnné *arraySize*, která je typu *CvPoint*. Vstupní obraz se zpracovává funkcí *cvCountNonZero()*, která vrací počet bodů v daném čtverci. Pokud v tomto čtverci není žádný aktivní pixel, vrací nulu, což je pak výhodné pro následné zpracování obrazu (zpracovávají se pouze nenulové čtverce). Na obrázku 31 je vlevo původní binární obraz a vpravo pak graficky zpracované pole *arr*. Každý počet aktivních pixelů v daném čtverci pak znázorňuje intenzita tohoto čtverce.



Obrázek 31: Vlevo binární obraz, vpravo rozdělený na čtverce 8x8 pixelů

Další myšlenkou, jak snížit časovou náročnost algoritmu, je požít Sobelův filtr. Ten je velmi často používaný a je velmi citlivý na směr hrany (na obrázku 32 vlevo je původní binární obraz, uprostřed pak Sobelův filtr v ose x a vpravo Sobelův filtr v ose y). Vzorec 11 názorně ukazuje jak vypadá jádro 3x3 Sobelova filtru pro osu x (vlevo), pro osu y (uprostřed) a v diagonálním směru (vpravo). Existují také jádra o rozměru 5x5 a 7x7.

Existují také modifikované jádra (vzorec 11 vpravo) pro Sobelův filtr, které jsou citlivé v diagonálním směru. Ty by byly pro náš účel také vhodné. Použití Sobelova filtru v openCV pak ukazuje opět výpis 6, řádky 1 až 7. Po odprahování obrazu pomocí Sobelova filtru by se pak v každém směru vytvořila dominantní přímka. Tyto se pak protnou v konkrétním bodě, kterým je zakřivení zatáčky v obraze.

$$\begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \quad \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad \begin{bmatrix} 0 & 1 & 2 \\ -1 & 0 & 1 \\ -2 & -1 & 0 \end{bmatrix} \tag{11}$$

5.3 Možnosti implementace

Na základě výše uvedeného je zřejmé, že pro funkci algoritmu v reálném čase potřebujeme poměrně výkonný procesor, který ovšem musí být implementovatelný do malého embedded zařízení (autíčko do kterého je zamýšleno zabudování celé desky, má rozměry cca 100x50 mm). Dalším limitujícím faktorem je nízná spotřeba, provoz bez chladiče a odolnost proti rušení. Z uvedeného nám vychází jako nejvýhodnější moderní procesory

Obrázek 32: Vlevo binární obraz, uprostřed Sobelův filtr v ose x, vpravo v ose y

ARM řady Cortex A8 nebo A9. V současné době je k dispozici velmi výkonný procesor i.MX535 firmy Freescale.

Jedná se procesor Cortex A8 s frekvencí až 1,2GHz se schopností dekódovat 1080p HD video, se dvěma grafickými jádry, NEON SIMD media akcelerátorem a vektorovým floating point koprocesorem. Je možno k němu připojit až 2GB DDR2 nebo DDR3 paměti. Maximální spotřeba procesoru je 2,2A při 1,37V a 1,2GHz, což jsou pouhé 3W při teplotě jádra $125\,°C$!

Pro testování jsem proto pořídil i.MX53 Quick Start Board, s procesorem i.MX535, 1GHz Cortex A8, osazený 1GB RAM za cenu 149USD. Deska o rozměrech 77 mm x 77 mm je napájena z 5V/2A zdroje a součástí kitu je SD karta s BSP a Linuxem. Pro srovnání jsem vybral v současné době velmi populární Rapsberry PI. To obsahuje čip Broadcom BCM2835, což je ARM11 (starší verze ARMv6 proti ARMv7 v Cortex A8) s floating point na 700Mhz a má 256MB RAM. Měří 84 mm x 54 mm. Pro porovnání je na obrázcích 37 i.MX53 Quick Start Board a na obrázku 38 Rapsberry PI rev. B.

Součástí i.MX53 Quick Start Boardu je 4GB flash SD karta s operačním systémem Linux, je tedy možno začít testovat výkon algoritmu na reálném HW. Pro nasazení v autíčku však bude nutno vytvořit specializovanou DPS (výrazně menší než vývojový kit), včetně řízení DC motoru.

6 Závěr

Předmětem mé práce bylo vytvoření funkčního algoritmu pro sledování dělících čar v obraze snímaného kamerou, bez pomocných senzorů (inerční systémy, GPS, radar, ultrazvuk). Algoritmus je určen jako testovací platforma (se specializovaným hardware), pro soutěže samořídících autíček pro Freescale Race Challenge.

Po prostudování řady materiálů jsem dospěl k názoru, že nejvýhodnější je použít knihovny openCV která obsahuje řadu algoritmů, určených právě pro zpracování obrazu a odstiňuje programátora od tvorby nízkoúrovňových funkcí. Použil jsem funkce pro převod obrazu na odstíny šedi, Cannyho detektor hran, Houghovu transformaci a řadu pomocných funkcí. Také jsem experimentoval s jinými detektory (thresholding, Sobelův operátor, atd.). Při vytváření algoritmu mne také napadla řada zlepšení a úprav, které hodlám do tohoto algoritmu zapracovat.

V nynější podobě algoritmus pracuje spolehlivě a detekuje zatáčky v dostatečném předstihu. Pro praktické nasazení je však nutno vytvořit specializovaný HW, dále použít kvalitní a menší kameru. Důvodem je rozměrové a hmotnostní omezení autíčka při soutěži. Dále hodlám pracovat na úpravě detekce zatáček tak, aby nebylo nutno používat Houghovu transformaci – důvodem je její časová náročnost a tím nemožnost zvýšit zpracovatelný počet snímků za sekundu. Pro filtraci hodnot pak používám statistické metody založené na váženém průměru a plovoucím průměru s exponenciálním zapomínáním. Za úvahu stojí také použití populárního Kalmanova filtru, používaného například v [18], který ovšem na druhou stranu zvyšuje časovou složitost algoritmu.

Radek Tesař

7 Reference

[1] Bellino M., Y.L. de Meneses, P. Ryser, J. Jacot, 2004. *Lane detection algorithm for an onboard camera.* Proc. SPIE, 5663:102—111, 2004.

[2] McCall JC, MM Trivedi, 2004. *An integrated, robust approach to lane marking detection and lane tracking.* Intelligent Vehicles Symposium, 2004 IEEE. Strana 533—537. ISBN: 0-7803-8310-9

[3] McCall JC, MM Trivedi, 2006. *Video-based lane estimation and tracking for driver assistance: Survey, system, and evaluation.* IEEE Transactions on intelligent transportation systems, 2006. Strana 20–37. ISSN : 1524-9050

[4] Kaske A., D. Wolf, R. Husson, 1997. *Lane boundary detection using statistical criteria.* International Conference on Quality by Artificial Vision, QCAV9, 1997. Strana 28–30.

[5] Su Chung–Yen, Fan Gen–Hau, 2008. *An Effective and Fast Lane Detection Algorithm*, Lecture Notes in Computer Science, Advances in Visual Computing, Springer Berlin / Heidelberg, 2008. Strana 942–948. ISBN: 978-3-540-89645-6

[6] Guo Keyou, Li Na, Zhang Mo, 2011. *Lane Detection Based on the Random Sample Consensus* International Conference of Information Technology, Computer Engineering and Management Sciences, 2011. Strana 38–41. ISBN 978-1-4577-1419-1.

[7] Miftahur Rahman, Md. Hasnaeen Rizvi Rahman, Abul L. Haque, M. Towhidul Islam. *Architecture of the Vision System of a Line Following Mobile Robot Operating in Static Environment*, Bangladesh: Department ofComputer Science and Engineering, North South University Banani, Dhaka–1213

[8] Dezhi Gao, Wei Li, Jianmin Duan, Banggui Zheng, 2009. *A Practical Method of Road Detection for Intelligent Vehicle*, International Conference on Automation and Logistics Shenyang, China, August 2009. Strana 980–985. ISBN: 978-1-4244-4794-7.

[9] Wu Ye, Shan Yuetian, Xu Yunhe, Wang Shu,Zhuang Yuchen, 2010. *The Implementation of Lane detective Based on OpenCV*, Second WRI Global Congress on Intelligent Systems, 2010. Strana 278–281. ISBN: 978-1-4244-9247-3.

[10] Chris Kreucher, Sridhar Lakshmanan, 1999. *LANA: A Lane Extraction Algorithm that Uses Frequency Domain Features*, IEEE TRANSACTIONS ON ROBOTICS AND AUTOMATION, VOL. 15, NO. 2, APRIL 1999. Strana 343–350. ISSN : 1042-296X

[11] Kluge Karl, Lakshmanan Sridhar, 1996. *Lane boundary detection using deformable templates: Effects of image subsampling on detected lane edges*, Recent Developments in Computer Vision, Springer Berlin/ Heidelberg, 1996. Strana 329–339. ISBN: 978-3-540-60793-9

[12] Canny, J. F.,1983. *Finding edges and lines in images.* Technical Report AI-TR-720, MIT, Artificial Intelligence Laboratory, Cambridge, MA, 1983. 149 stran.

[13] Canny, J. F.,1986. *A computational approach to edge detection.* IEEE Transactions on Pattern Analysis and Machine Intelligence, 1986. Strana 679–698. ISSN: 0162-8828

[14] Rafael C. Gonzales, Richard E. Woods, 2002. *Digital Image Processing, Second Edition*, Prentice–Hall, Inc., 2002. 793 stran. ISBN: 0-201-18075-8.

[15] Martina Litschmannová, 2011. *Úvod do statistiky*,[Online]. [cit. 10.4.2012]. `http://mi21.vsb.cz/modul/uvod-do-statistiky`

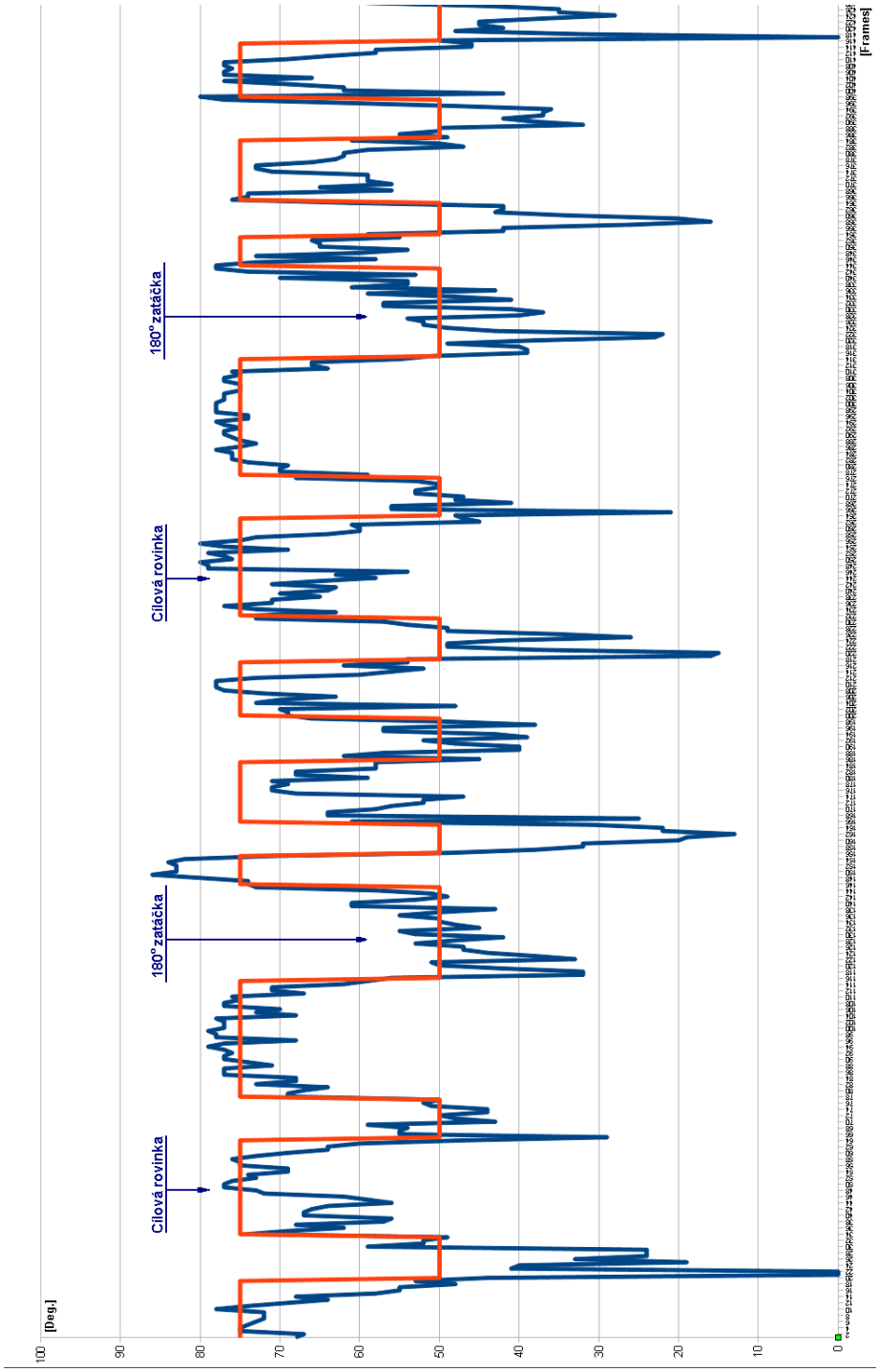
[16] Eduard Sojka, Jan Gaura, Michal Krumnikl, 2011 *Matematické základy digitálního zpracování obrazu*,[Online]. [cit. 10.4.2012]. `http://mrl.cs.vsb.cz/people/sojka/dzo/mzdzo.pdf`

[17] J. R. Parker, 1996. *Algorithms For Image Processing And Computer Vision*, Wiley, 1996. 432 stran. ISBN–10: 0471140562.

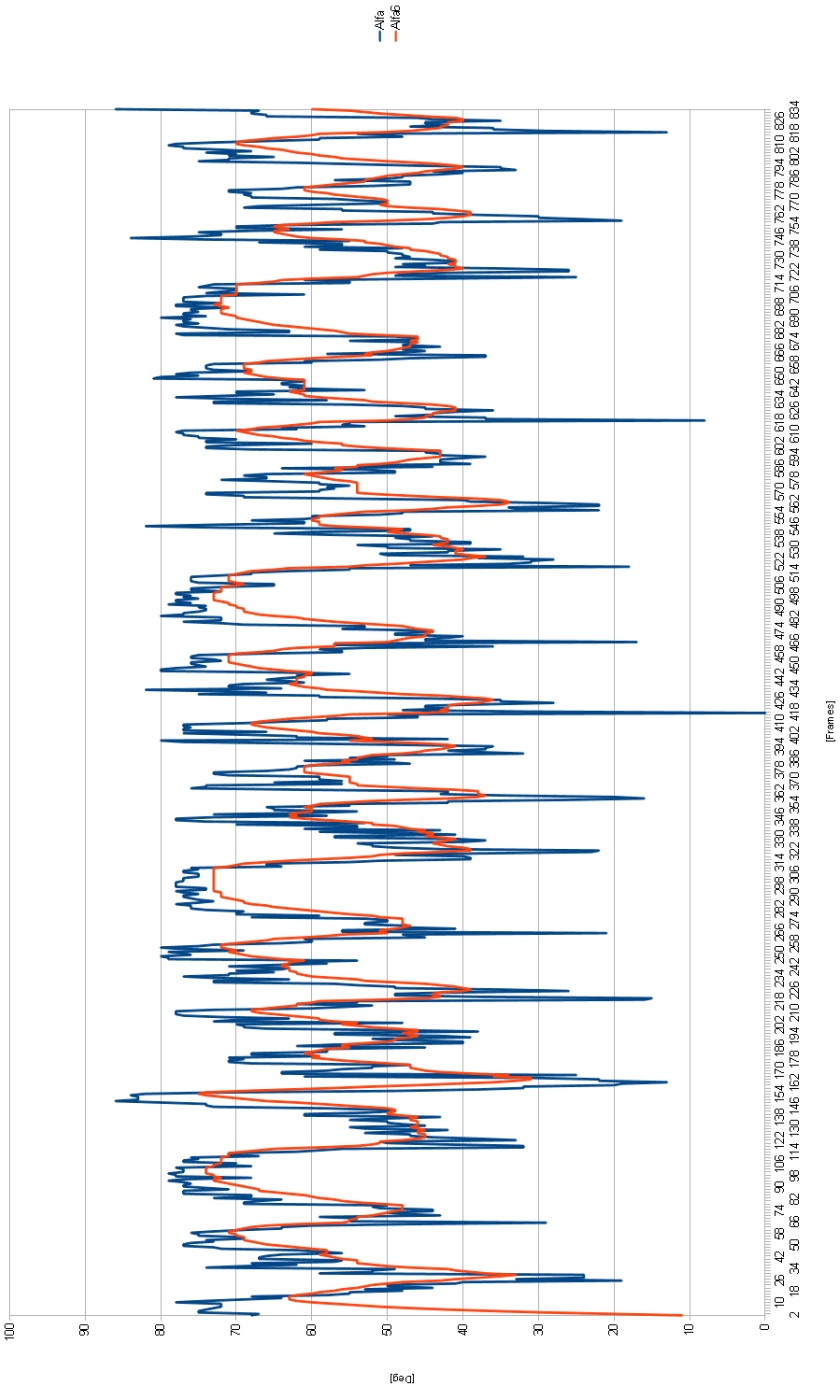
[18] Suttorp T., Bucher T., 2006. *Learning of Kalman Filter Parameters for Lane Detection*, IEEE Intelligent Vehicles Symposium, 2006. Strana 552–557. ISBN: 4-901122-86-X

A Grafy a tabulky

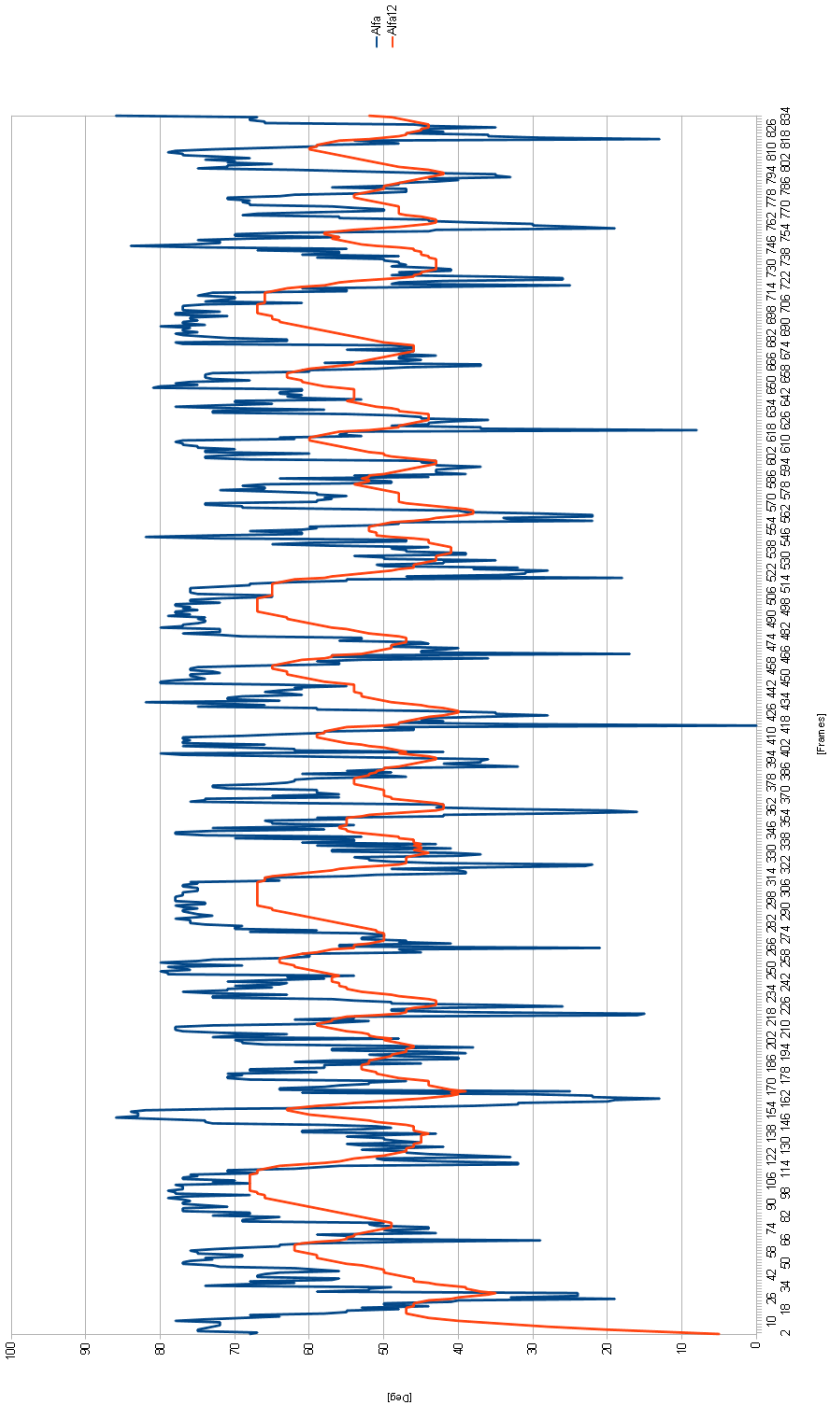
Obrázky a grafy ve větším rozlišení, určené k podrobnějšímu zkoumání, nebo jejichž velikost by překážela v textu diplomové práce.



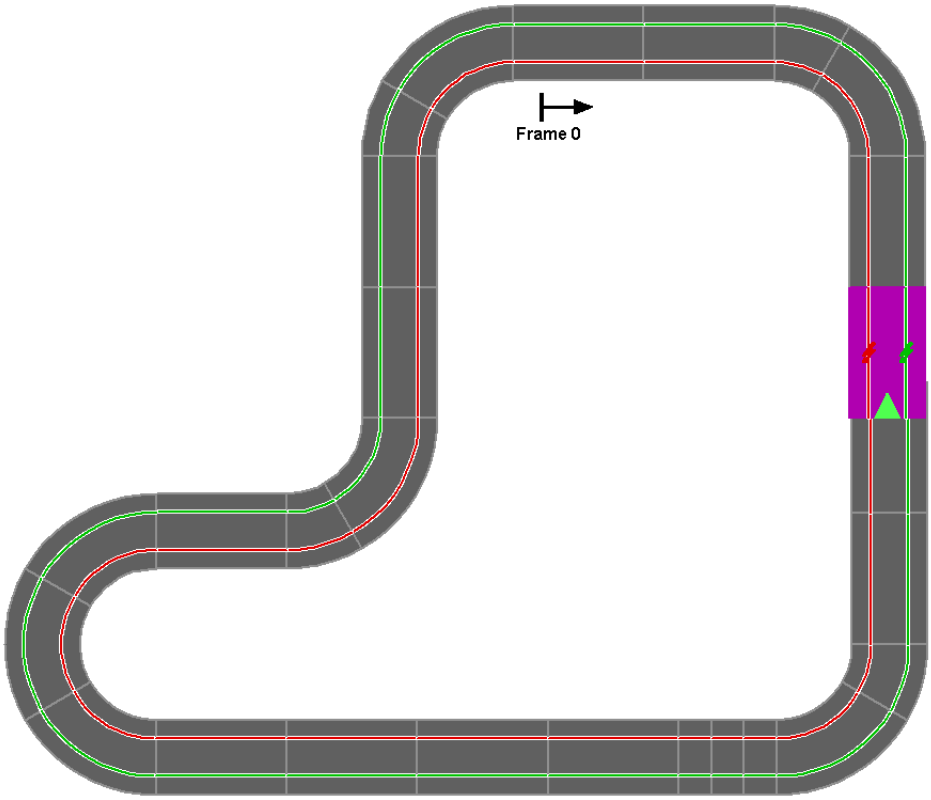
Obrázek 33: Graf testovací dráhy



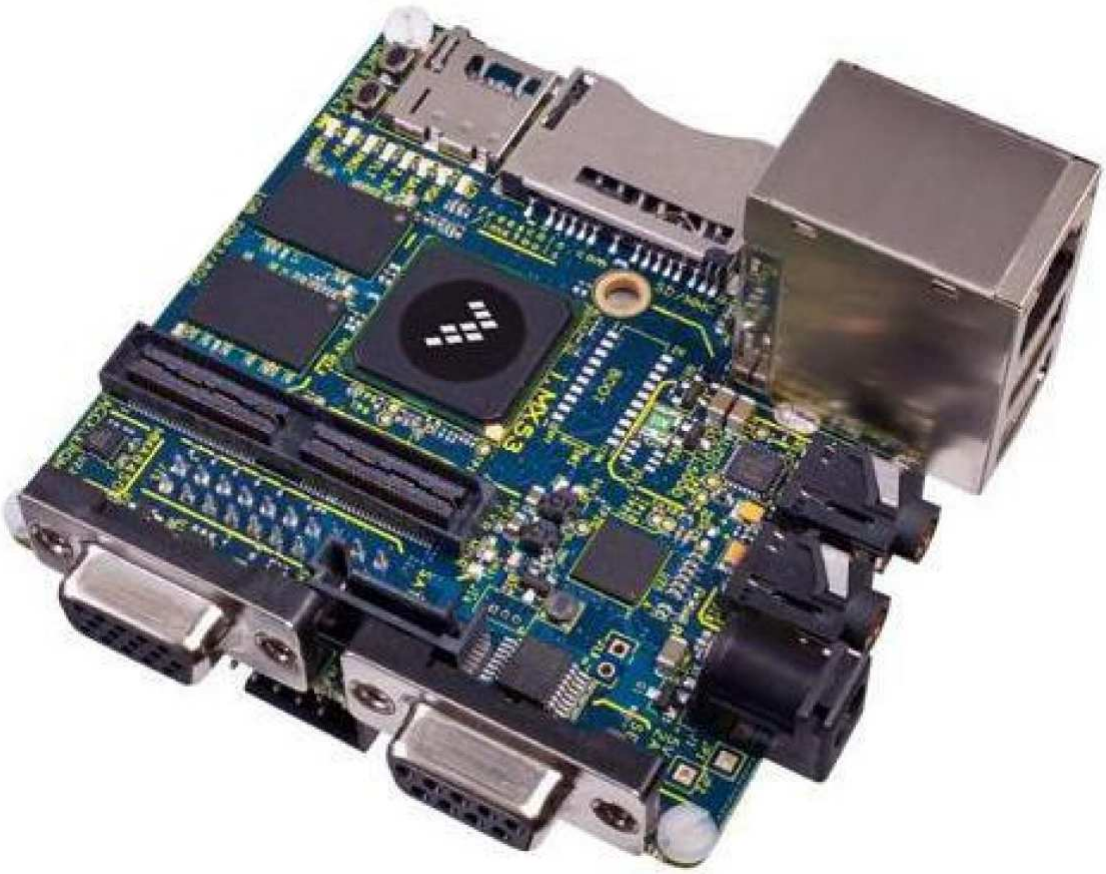
Obrázek 34: Graf dráhy pro plovoucí průměr, k=6



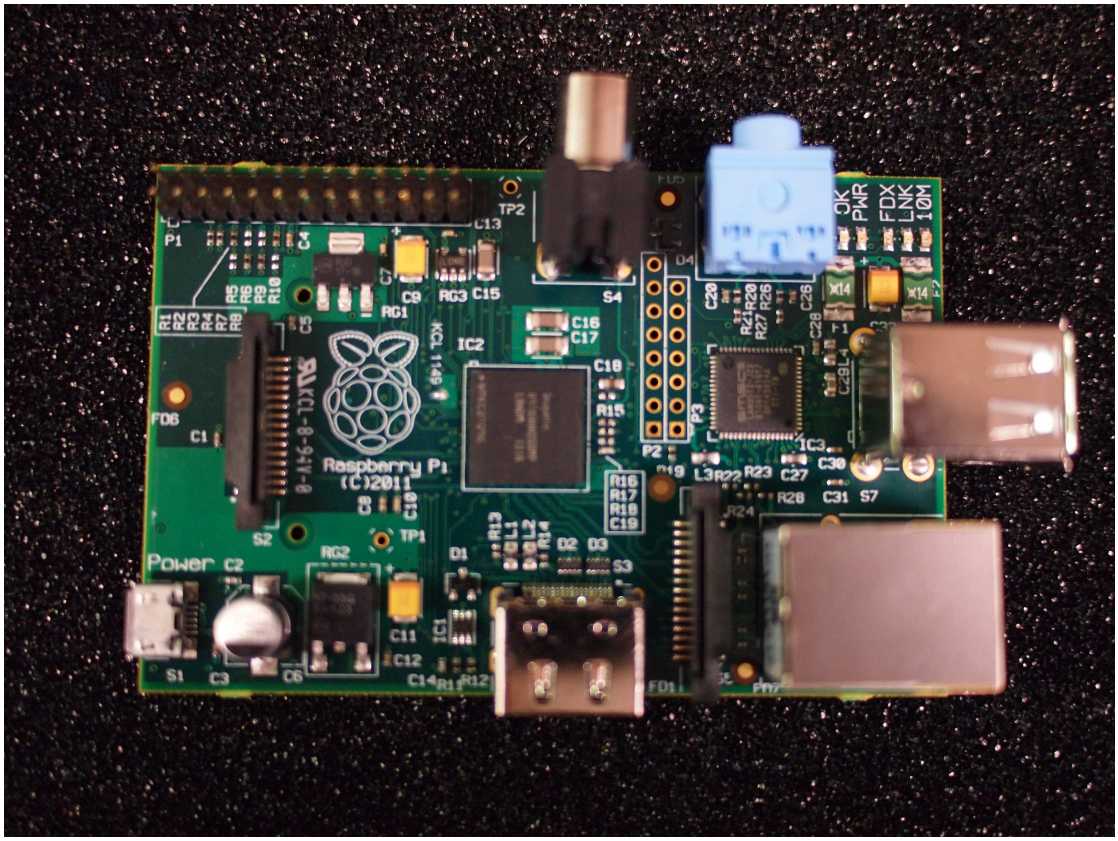
Obrázek 35: Graf dráhy pro plovoucí průměr, k=12



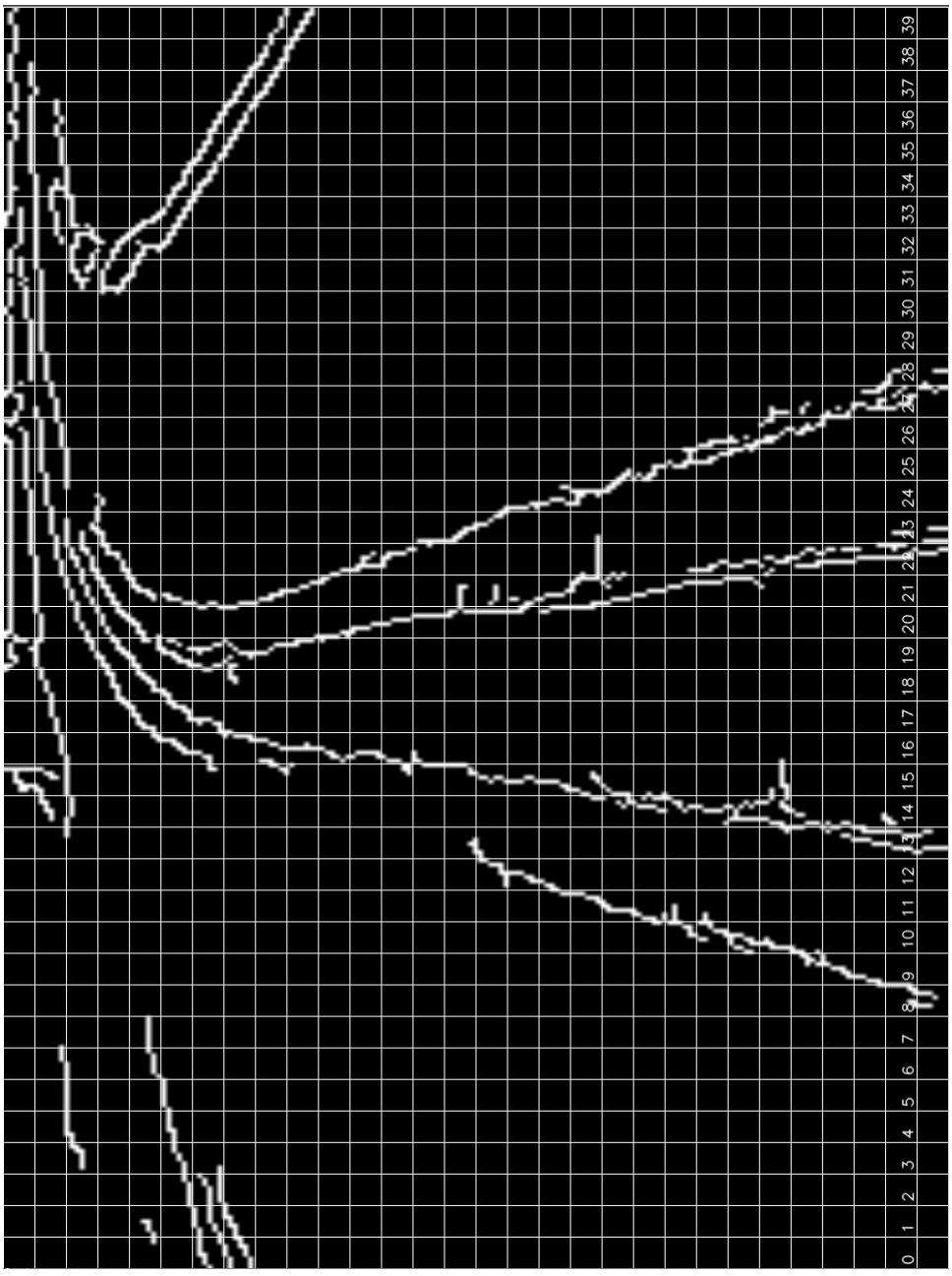
Obrázek 36: Testovací dráha



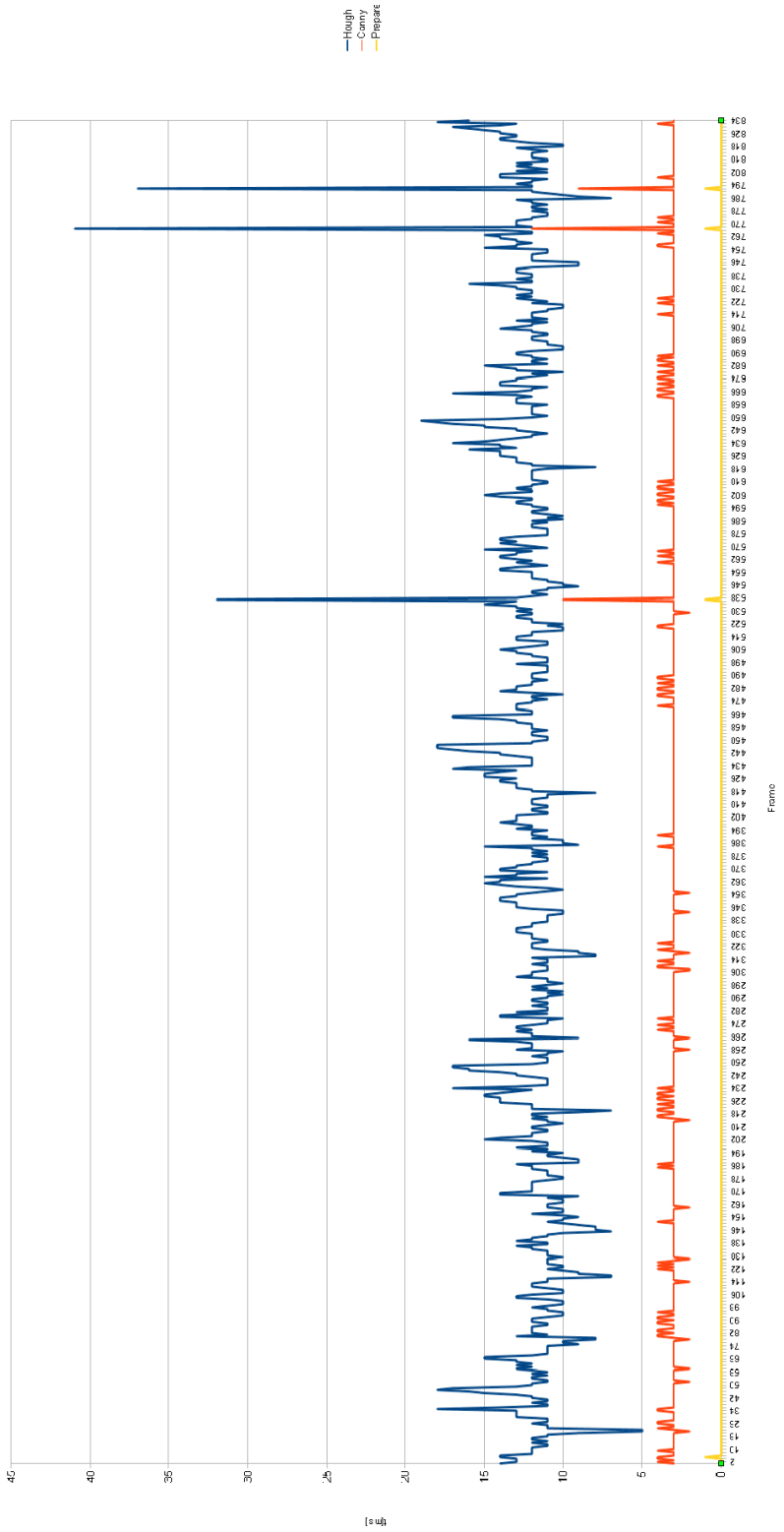
Obrázek 37: i.MX53 Quick Start Board



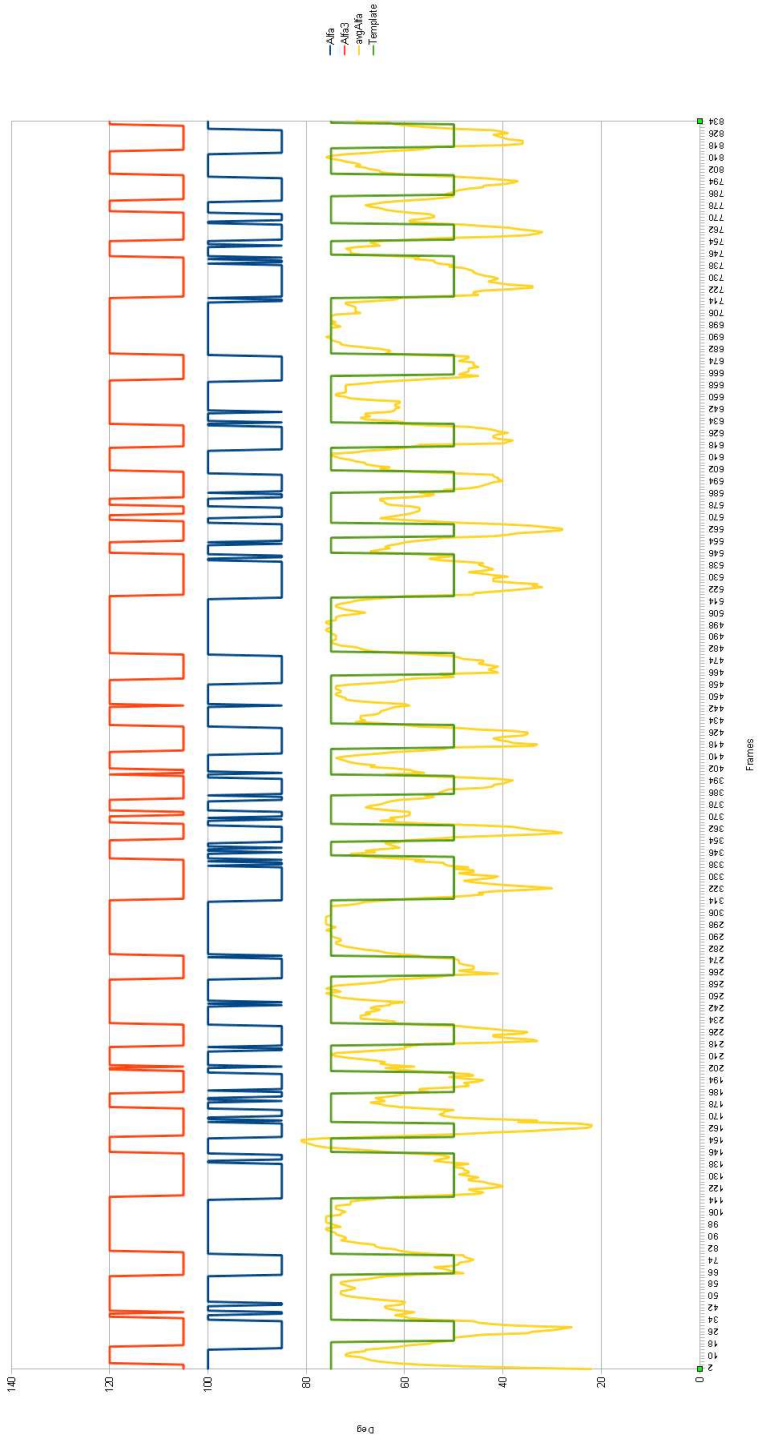
Obrázek 38: Rapsberry PI



Obrázek 39: Binární obraz, rozdělený na čtverce 8x8 pixelů



Obrázek 40: Graf časové složitosti



Obrázek 41: Graf detekce zatáček pro plovoucí průměr s k=3 a bez něj