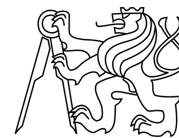
ESKÉ VYSOKÉ U ENÍ TECHNICKÉ V PRAZE
FAKULTA INFORMA NÍCH TECHNOLOGIÍ



ZADÁNÍ BAKALÁ SKÉ PRÁCE

Název: Aplikace pro porovnávání algoritm po íta ového vid ní
Student: Adam Kubišta
Vedoucí: Ing. Ji í Chludil
Studijní program: Informatika
Studijní obor: Softwarové inženýrství
Katedra: Katedra softwarového inženýrství
Platnost zadání: Do konce letního semestru 2016/17

Pokyny pro vypracování

1. Analyzujte možnosti využití robotické platformy (6-kolové vozítko se senzorovou výbavou: stereo kamera, ultrazvukový m i vzdálenosti atd.) jako zdroje vstupních dat pro porovnávací aplikaci.
2. Analyzujte rozhraní voln dostupných implementací algoritm používaných pro po íta ové vid ní (detekce p ekážky - ano-ne, kategorizace p ekážky - musí se objet x lze p ekonat) a definujte porovnávací metodiku.
3. Definujte funk ní a nefunk ní požadavky pro porovnávací aplikaci zmín ných implementací algoritm .
4. Na základ metodiky z bodu 2 navrhn te porovnávací aplikaci, která bude na vstupní data z robotické platformy postupn aplikovat jednotlivé algoritmy a shromaž ovat m ená data pro následné porovnání. Výsledky porovnání budou dostupné p es uživatelské rozhraní podporující velmi vysoké rozlišení.
5. Implementujte prototyp porovnávací aplikace.
6. Otestujte prototyp porovnávací aplikace s využitím vybraných implementací algoritm po íta ového vid ní.

Seznam odborné literatury

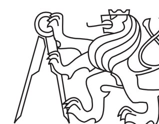
Dodá vedoucí práce.

L.S.

Ing. Michal Valenta, Ph.D.	prof. Ing. Pavel Tvrdík, CSc.
vedoucí katedry	d kan

V Praze dne 23. prosince 2015

České vysoké učení technické v Praze
Fakulta informačních technologií
Katedra Softwarového inženýrství



Bakalářská práce

Aplikace pro porovnávání algoritmů počítačového vidění

Adam Kubišta

Vedoucí práce: Ing. Jiří Chludil

16. května 2016

Poděkování

Chci poděkovat Ing. Jiřímu Chludilovi a Ing. Radkovi Richtrovi za cenné připomínky k práci a Bc. Jiřímu Kubištovi za asistenci při testování a sběru dat v SAGElabu.

Prohlášení

Prohlašuji, že jsem předloženou práci vypracoval(a) samostatně a že jsem uvedl(a) veškeré použité informační zdroje v souladu s Metodickým pokynem o etické přípravě vysokoškolských závěrečných prací.

Beru na vědomí, že se na moji práci vztahují práva a povinnosti vyplývající ze zákona č. 121/2000 Sb., autorského zákona, ve znění pozdějších předpisů. V souladu s ust. § 46 odst. 6 tohoto zákona tímto uděluji nevýhradní oprávnění (licenci) k užití této mojí práce, a to včetně všech počítačových programů, jež jsou její součástí či přílohou, a veškeré jejich dokumentace (dále souhrnně jen „Dílo“), a to všem osobám, které si přejí Dílo užít. Tyto osoby jsou oprávněny Dílo užít jakýmkoli způsobem, který nesnižuje hodnotu Díla, a za jakýmkoli účelem (včetně užití k výdělečným účelům). Toto oprávnění je časově, teritoriálně i množstevně neomezené. Každá osoba, která využije výše uvedenou licenci, se však zavazuje udělit ke každému dílu, které vznikne (byť jen zčásti) na základě Díla, úpravou Díla, spojením Díla s jiným dílem, zařazením Díla do díla souborného či zpracováním Díla (včetně překladu), licenci alespoň ve výše uvedeném rozsahu a zároveň zpřístupnit zdrojový kód takového díla alespoň srovnatelným způsobem a ve srovnatelném rozsahu, jako je zpřístupněn zdrojový kód Díla.

V Praze dne 16. května 2016 . . . .

České vysoké učení technické v Praze
Fakulta informačních technologií
© 2016 Adam Kubišta. Všechna práva vyhrazena.
Tato práce vznikla jako školní dílo na Českém vysokém učení technickém v Praze, Fakultě informačních technologií. Práce je chráněna právními předpisy a mezinárodními úmluvami o právu autorském a právech souvisejících s právem autorským. K jejímu užití, s výjimkou bezúplatných zákonných licencí, je nezbytný souhlas autora.

Odkaz na tuto práci

Kubišta, Adam. *Aplikace pro porovnávání algoritmů počítačového vidění*. Bakalářská práce. Praha: České vysoké učení technické v Praze, Fakulta informačních technologií, 2016.

Abstrakt

Tato práce vytvořila nástroj pro porovnávání algoritmů počítačového vidění se zaměřením na rozpoznávání překážek. Práce analyzuje robotické vozítko, které je součástí inventáře SAGElabu na ČVUT. Data ze senzorů tohoto vozítka zpracovává různými algoritmy počítačového vidění a analyzuje výsledky. Na základě tohoto poznání definuje způsob, kterým se algoritmy v nástroji budou porovnávat. S prací vzniká software, který uživateli umožňuje porovnávat svoje algoritmy určenou metodikou, zobrazovat výsledek ve velmi vysokém rozlišení a tak hledat nejvhodnější implementaci algoritmu pro jeho záměry.

Klíčová slova počítačové vidění, rozpoznávání překážek, porovnávání implementací algoritmů, velmi vysoké rozlišení, SAGElab

Abstract

In this thesis a tool for users to compare their implementations of computer vision algorithms for obstacle detection was crafted. Analysis of CTU's SAGElab robotic buggy in regard of data providing for implementations of such algorithms delivers the basic inspiration for defining the metrics used for comparing in this tool. Then, based on this analysis the tool was created for users to help them compare their algorithms with ability to display results in ultra high resolution to provide a method for determining most suitable implementation for user's need.

Keywords computer vision, obstacle detection, comparing algorithms implementations, ultra high resolution, SAGElab

Obsah

Úvod **1**

1 Cíl práce **3**

2 Analýza **5**
2.1 Zadání práce 5
2.2 Robotické vozítko 6
2.3 Použíté technologie 9
2.4 Volně dostupné implementace algoritmů počítačového vidění . . 13

3 Návrh **21**
3.1 Porovnávací metodika 21
3.2 Požadavky na aplikaci 23
3.3 Architektura 26
3.4 Podaplikace v SAGE2 32

4 Implementace **35**
4.1 Základní rozdělení 35
4.2 Databázové řešení 36
4.3 Lokální aplikace 38
4.4 Vzdálená SAGE2 aplikace 46
4.5 Debugování 47
4.6 Vlastnosti implementací 47

5 Testování **49**
5.1 Příprava 49
5.2 Pokrytí funkčnosti případy užití 50

6 Navázání na tuto práci **55**

7 Závěr **57**

A Seznam použitých zkratek **59**

B Obsah přiloženého média **61**

C Uživatelská příručka **63**
C.1 Uživatelská příručka pro GUI lokální aplikaci 63
C.2 Uživatelská příručka pro SAGE2 aplikaci 67

D Instalační příručka **69**
D.1 Instalační příručka pro GUI aplikaci 69
D.2 Instalační přířučka pro SAGE2 aplikaci 69

Literatura **71**

Seznam obrázků

2.1 Robotické vozítko 7
2.2 Vizualizace postupu algoritmu Kontury + AABB. 16
2.3 Vizualizace postupu algoritmu Hough + Kontury + Area. 19

3.1 Jednoduchý doménový model 26
3.2 Databázový doménový model 27
3.3 Obrazovka s formulářem 29
3.4 Obrazovka s tabulkou 30
3.5 Struktura obrazovek 31
3.6 Zobrazování na SAGE2 33

4.1 Rozhraní lokální aplikace 37
4.2 Adresářová struktura databáze 38
4.3 Logika výběru obrazovky 39
4.4 Jednoduchý případ se třemi vlákny, třemi vstupními daty a jedním algorimtem 46

5.1 Porovnání testů různě nakonfigurovaných verzí analyzovaných implementací nad daty z běžného fotoaparátu. 53

C.1 Adresářová struktura aplikace 67

Seznam kódů

2.1 Část C++ kódu pro Kontury + AABB 15
2.2 Část C++ kódu pro Cascade classification 16
2.3 Část C++ kódu pro Hough + Kontury + Area 18
4.1 Pseudokód provedení jednoho testu 42
4.2 Deklarace rozhraní uživatelského algoritmu 43
4.3 Příklad přečtení parametrů rozdílných velikostí 44
C.1 Příklad deklarace DLL funkce 63

Seznam tabulek

3.1 Mapování případů užití na funkční požadavky 25

Úvod

Odvětví humanoidních robotů a robotických pomocníků s umělou inteligencí se stále více a více rozšiřuje. Od automatizovaných topících systémů pro domácnost se dostáváme ke strojům NAO[2] a obdivuhodným kreacím z Boston Dynamics[3]. Stejně tak jako před sto lety umělci předvídali budoucnost ve svých obrázcích[4], tak i v dnešní době vznikají v umění pohledy do budoucna - např. film Chappie[5], nebo dílo The Ables[6]. I v dnešní době lze očekávat, že z moderní umělecké tvorby se také vyplní některá přání a zmiňovaná technologie budocnosti bude tu.

Počítačové vidění je jeden z fundamentálních prvků moderních samostatně rozhodujících se robotů. Bez tohoto „smyslu" by se mnohem hůře, zda-li vůbec orientovali v prostoru. Když zmiňujeme počítačové vidění, máme na mysli především kamery a učinění motorického rozhodnutí na základě analýzy obrazu v reálném čase. Oproti lidskému vidění však moderní stroje mohou používat další metodiky, např. měřit vzdálenost ultrazvukem nebo laserem, a také sledovat okolí termovizí či dokonce sdílet informace o prostoru s ostatními roboty a tím si analýzu obrazu konkrétního okolí potvrdit či nezaznamenané objekty objevit. Nejobecnější využití pro zpracování takovýchto dat je pro pohyb v prostoru - rozhodnutí o tom, kde jsou překážky a kudy je možno se pohybovat. Na toto se v této práci zaměřím.

Protože prostorů, ve kterých se roboti mohou pohybovat je obrovské množství, tak je nemožné zaručit správnou analýzu vstupních dat ze senzorů absolutně ve všech případech. Celá problematika korektnosti algoritmů počítačového vidění se tedy přesouvá do prostorů pravděpodobností. Podobně to také funguje u lidského vidění - existují optické klamy, které alterují prostorový vjem a tak mění podvědomou analýzu okolního prostoru. Tedy ani lidé nejsou vždy stoprocentně schopni poznat, jestli se jedná o překážku či nikoliv.

Abychom zaručili největší kvalitu, je třeba algoritmy testovat na různorodých datech s co největším pokrytím typických případů. V této práci se na takovéto testovaní zaměřím.

Kapitola 1

Cíl práce

Cílem této práce je zde nejprve analyzovat a porozumět zadání. Poté se práce dělí do teoretické a praktické části. V teoretické části probíhá analýza robotického vozítka na FIT ČVUT, a analýza volně dostupných implementací algoritmů počítačového vidění. Na základě těchto analýz pak vzniká definice porovnávací metodiky, která bude určovat vzájemnou jakost implementací. Dále se v teoretické části zaměřuje na mělkou analýzu použitých technologií a návrh architektury pro praktickou část.

V praktické části vzniká software podle rozborů, definic a návrhů z teoretické části. Práce zde popisuje zvolené metody implementace a jejich otestování. V praktické části lze také naleznout uživatelskou a instalační příručku k vyvinutému softwaru. Na konci je práce shrnuta a naznačuje možnosti navázní.

Kapitola 2

Analýza

Tato kapitola se dělí na čtyři části: v první části dochází k rozboru zadání této práce, v druhé části je analyzované robotické vozítko z inventáře SAGElab na ČVUT, ve třetí části je zmínka o použitých technologiích pro software, který s touto prací vzniká a v poslední části jsou vyhledávány volně dostupné implementace algoritmů počítačového vidění.

2.1 Zadání práce

1. **Analyzujte možnosti využití robotické platformy (6-kolové vozítko se senzorovou výbavou: stereo kamera, ultrazvukový měřič vzdálenosti atd.) jako zdroje vstupních dat pro porovnávací aplikaci.**

   V tomto bodu je rozebráno, čím vozítko disponuje a jaká data relevantní pro tuto práci může poskytnout. Práce se bude zajímat i o způsob, kterým se data dají získat.

2. **Analyzujte rozhraní volně dostupných implementací algoritmů používaných pro počítačové vidění (detekce překážky - ano-ne, kategorizace překážky - musí se objet x lze překonat) a definujte porovnávací metodiku.**

   Zde dochází k výběru volně dostupných implementací algorimtů počítačového vidění. Výsledkem této analýzy zdaleka není koreknotst algoritmů, nýbrž jejich rozhraní. Předmětem zájmu je, jak se implementace algoritmů používají, a jaké zdroje vyžadují. Na základě tohoto rozboru je pak definována porovnávací metodika, která bude v rámci rozhraní proveditelná.

3. **Definujte funkční a nefunkční požadavky pro porovnávací aplikaci zmíněných implementací algoritmů.**

   V této části se práce přesouvá do části věnované tvorbě softwaru. Vznikají zde požadavky na aplikaci a tím se určuje funkční schopnost díla. Zde také dochází k definici způsobu otestování vytvořené aplikace.

4. **Na základě metodiky z bodu 2 navrhněte porovnávací aplikaci, která bude na vstupní data z robotické platformy postupně aplikovat jednotlivé algoritmy a shromažďovat měřená data pro následné porovnání. Výsledky porovnání budou dostupné přes uživatelské rozhraní podporující velmi vysoké rozlišení.**

   V tomto bodu vzniká architektura aplikace.

5. **Implementujte prototyp porovnávací aplikace.**

   Zde jsou popsány způsoby implementace jednotlivých částí architektury a občas je polemizováno nad zvoleným řešením.

6. **Otestujte prototyp porovnávací aplikace s využitím vybraných implementací algoritmů počítačového vidění.**

   V této části dochází k testování aplikace dle zvolené testovací metody se vstupními daty z robotického vozítka a analyzovanými algoritmy v této práci.

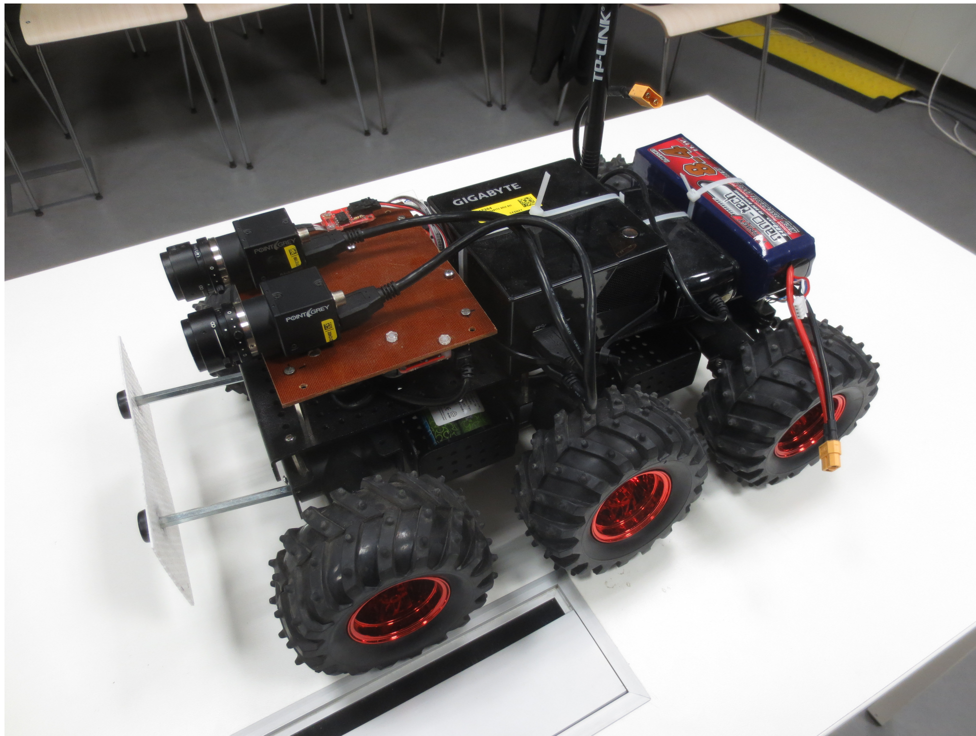
2.2 Robotické vozítko

Robotické vozítko (obr. 2.1) ze SAGElab je typické zařízení, které by mohlo používat různé algoritmy počítačového vidění. Obsahuje dvě kamery Point Grey [8] ve standardním stereoskopickém rozložení, tedy od sebe vzdálené jako lidské oči. Takováto vzájemná pozice kamer je nutná pouze pro správný vjem lidmi. Pokud by kamery byly mimo standardní rozložení tak by došlo k deformaci obrazu oproti běžnému lidskému vidění, protože zobrazovací zařízení (3D televize) vysílá záznam z každé kamery do každého oka zvlášť[9] a tím předpokládá, že záznam má stejnou vzájemnou polohu čoček jako u pozorovatele. Při jiném rozložení stereoskopické kamery by lidské oči přijímaly dva obrazy v jiné vzájemné vzálenosti (vzájemná vzdálenost očí pozorovatele), než byly zaznamenány (vzájemná vzdálenost čoček stereoskopické kamery) a tím by byl vjem deformován. Pokud by však stereoskopcická kamera byla výhradně určena pro zpracování algoritmy, tak by vzájemné rozpoložení čoček nebylo podstatné, protože by nebylo potřeba záznam zobrazovat. V případě vozítka na FIT jsou standardně umístěny, protože je záznam v SAGElab potřeba i zobrazovat na 3D televizích.

Vozítko dále obsahuje ultrazvukový měřič vzdálenosti, který je přesný pouze s většími plochými objekty [8]. Přítomen je i akcelerometr, gyroskop, měřič světla, teploty, tlaku, nadmořské výšky a magnetometr [8, 10].

Na vozítku je minipočítač GIGABYTE napájený dobíjecí baterií[10], který celé zařízení zaštituje. Obě kamery jsou připojeny do minipočítače. Ostatní senzory jsou připojeny do adekvátní desky Arduino [16, 10]. Deska Arduino poté poskytuje sériovou komunikací data ze senzorů minipočítači. Do počítače jsou ještě zapojeny motory, které mohou být ovládány, aby se vozítko mohlo pohybovat. Motory jsou poháněny samostanou baterií [10].

S minipočítačem je možno komunikovat skrz programové rozhraní Yuri. [8, 10, 17]



Obrázek 2.1: Robotické vozítko

2.2.1 Software Yuri

Yuri (yuri-light) je volně dostupná open source knihovna v C++ pro získávání, zpracování a síťový přesun vizuálních [11] a zvukových dat [13]. Knihovna podporuje především operační systém Linux. Funkčnost na operačním systému Windows nezaručuje. Knihovna je nutně závislá na volně dostupné C++ knihovně boost. Dále je zbytně závislá na dalších volně dostupných knihovnách. Yuri je modulární a tak umožňuje zamezit použití některých části kódu a tím zbytné závislosti vyřadit. Uživatel knihovy si musí zdrojové kódy sám zkompilovat.[12]

Yuri umožňuje především získávat živá data z připojených periferií, přes síť HTTP, RTP nebo OSC protokolem či získávat data ze souborů. Získávaná

data umožňuje upravovat a převádět do různých formátů. Data je pak možné přesouvat přes síť HTTP POST požadavky nebo vysílat RTP či OSC protokolem. Možné je také data uložit do souboru či přímo zobrazit. Knihovna také zprostředkovává sériovou komunikaci s připojenými periferiemi. Výše zmíněné funkčnosti rozhodně nejsou všechny, ale dostatečně naznačují oblasti, kterými se software zabývá. Díky dostupnému zdrojovému kódu si může uživatel funkčnosti doplnit nebo dohledat.[13] [18]

Zvolené funkčnosti knihovny je buďto možné přímo využít ve vlastním kódu formou volání API zkompilované knihovny a nebo využít předpřipravené real-time aplikace reprezentující toto API. Chod aplikace se řídí speciálním konfiguračním souborem ve formátu XML odkázaným při spuštění této aplikace. Druhá forma použití knihovny je vhodná pro rychlé prototypování a bezprostřední používání. [15] Dostupná uživatelská dokumentace softwaru je minimální.

2.2.2 HTTP

S HTTP (Hypertext Transfer Protocol) se v dnešní době běžně setkáváme - používá se především pro komunikaci mezi uživatelem a webovým serverem pro zisk obsahu webových stránek. Výše zmíněné HTTP POST požadavky znamenají konkrétní využítí možnosti HTTP prokotolu pro přesun dat od klienta na server. Existují také HTTP GET požadavky, které uchovávají data přímo v URL, tento způsob je vhodný pro navigaci po stránkách a v této práci ho využíváme (např. „http://localhost/index?page=detail“ je HTTP GET požadavek s parametrem „page“, který má hodnotu „detail“).[21] Více detailů v [21].

2.2.3 RTP

Real-time transport protocol popisuje způsob přenosu realtimeových dat mezi vysílačem a přijímačem v počítačové síti. Typická realtimová data jsou zvukové a obrazové signály právě přijatá senzory. RTP podporuje i vysílání pro více přijímačů pokud to síť umožňuje. RTP zahrnuje dva úzce související protokoly - RTP samotný, který popisuje přenos realtimových dat a RTCP, který monitoruje aktuální komunikaci mezi účastníky.[19] Více detailů v [19].

2.2.4 OSC

Open Sound Control je síťový protokol, založený na zasílání zpráv především mezi multimediálními zařízeními. OSC definuje jednoduché pakety, jako elementy komunikace, které obalují zasílaná data jistým způsobem. Více v [20]

2.2.5 Zisk dat a ovládání

Na minipočítači vozítka je operační systém Linux Gentoo, který po svém nahrání spustí předpřipravené programové rozhraní Yuri s konfiguračním souborem. Yuri pak běží v reálném čase podle konfigurace a posílá data skrze lokální síť. Na přijímacím počítači je také spuštěno rozhraní Yuri, které díky nastavitelnému konfiguračnímu souboru jedná podle potřeb uživatele. Většinou data přímo zobrazuje na místní 3D televizi. Robot se ovládá signály poslanými z přijímacího počítače OSC protokolem po místní síti. Přijaté signály minipočítačem robota jsou poté přesměrovány do ovladače motorů. Motory pak adekvátně posouvají vozítkem. Zdrojové signály na přijímacím počítači se obvykle generují na základě analogového vstupu z herního ovladače. [23] [10] [24] Díky vysoké konfigurovatelnosti knihovny Yuri je možné například připravit sekvence signálů pro motory do souboru a ty pak odesílat. Dá se tím docílit co nejpřesnější opakování pokusu zahrnující pohyb vozítka (např. otočení o 90 stupňů). Signály by se také daly generovat proceduálně s využitím algoritmů počítačového vidění například vygenerovat manévr objetí překážky široké 30 cm zleva. Pro potřeby této práce pouze využíjeme snímky pořízené kamerami z robotického vozítka nebo snímky z obyčejného fotoaparátu ve formátu JFIF.

2.2.6 JFIF

JFIF (JPEG File Interchage Format) je známý formát dat kompresního algoritmu JPEG. JPEG (Joint Photographics Experts Group) je ztrátová kompresní metoda pro snímky. [22]. Více v [25].

2.3 Použíté technologie

2.3.1 HTML

HTML (Hypertext Markup Language) je kódovací jazyk popisující strukturu webových stránek. Umožňuje různými způsoby formátovat data, která chce autor zobrazit (např. tabulky, nadpisy, paragrafy), definovat hypertextové odkazy pro zisk obsahu webové stránky z jiných adres, vytvářet formuláře, díky kterým může návštěvník webových stránek se serverem interagovat či vkládat obsahy jiných webových stránek, videozáznamy, audiozáznamy nebo různé aplikace. [28] HTML kód obsahuje jak data k zobrazení, tak jejich toužený formát definovaný tagy. [29]

2.3.2 CSS

CSS (Cascading Style Sheets) je kódovací jazyk popisující vzhled webových stránek - včetně použitého písma, rozmístění, barev, velikostí apod. CSS umožňuje adaptovat prezentaci různým zobrazovacím zařízením.[28] CSS umožňuje definovat třídy a jejich stylování, které se dají jednotlivým elementům přiřazovat nebo umožňuje stylovat elementy přímo. CSS se dá jednoduše provázat s HTML. Způsoby inegrace jsou v tři: definice přímo v HTML tagu v atributu „style“, definice tříd a stylů v HTML kódu nebo v externím CSS souboru, na který se HTML soubor odkazuje. V posledních dvou případech se HTML element styluje pomocí tříd nebo přímým výběrem. [31]

2.3.3 JavaScript

JavaScript je dynamický objektově orientovaný skriptovací jazyk který nemá koncept vstupu a výstupu jako běžné objektově orientované jazyky. Tento skriptovací jazyk je určen k interpretaci v hostitelském prostředí, které by mělo zaručovat vstupy a výstupy tohoto skriptu - především se jedná o dostupné webové prohlížeče. Běžně se s ním můžeme setkat na webových stránkách, které prohlížíme skrze populární a moderní webové prohlížeče. JavaScript umí úzce spolupracovat s HTML kódem a tím například zařizovat jakékoliv změny obsahu bez přechodu na jinou HTTP adresu - může se jednat o změny textu, stránkování obsahu, či animace přesouvající prvky.[33] Více o základech JavaScriptu se lze dozvědět v [33] a případně [34].

2.3.4 C++

C++ je populární objektově orientovaný programovací jazyk. Vznikl jako rozšíření programovacího jazyka C. [27] C++ je otevřený programovací jazyk standardizovaný ISO. Zdrojový kód se kompiluje kompilátorem přímo do instrukcí cílového hardwaru. Tento jazyk nabízí obrovské možnosti programátorovi za cenu svoji nekonkrétnosti. Lze zvolit různé programátorské přístupy, není ani nutné používat objektově orientované paradigma. C++ podporuje téměř všechny vlastnosti jazyka C a kompilátory jsou schopny tento C kód zkompilovat jako C++ kód. Díky své popularitě existuje spoustu knihoven, které může programátor využít. [26]

2.3.5 Libuv

Libuv je multiplatformní open source knihovna v jazyce C se zaměřením na asynchronní vstup a výstup dat. Primárně tato knihovna byla vyvinuta pro Node.js, ale využívá ji i ostatní software. Tato knihovna umožňuje například základní operace se systémovými soubory, obstarávání síťových socketů TCP nebo UDP či obsluhu vláken. [35] Knihovnu je možné zkompilovat se zdrojovým kódem a tím přímo využít funkčnosti formou volání implementovaných

funkcí. Díky vlastnostem C++ ji lze beze změny použít při kompilaci C++ kódu. Protože je knihovna open source, lze její rozhraní identifikovat přímo ze zdrojových kódů, nebo vyčíst z příkladů[37][36] či oficiální dokumentace[38].

2.3.6 RAPIDXML

RAPIDXML je C++ open-source knihovna se zaměřením na rychlé parsování XML kódu. K C++ zdorjovému kódu stačí explicitně vložit jediný soubor a tím využít funkcionalit, které knihovna nabízí.[39] Dokumentace API je dostupná v [39].

2.3.7 V8, Node.js a npm

Node.js je JavaScriptové prostředí naimplementované s využítím JavaScriptového enginu V8.[41] V8 je vysokovýkonostní open source engine vytvořený v C++. Mimo Node.js je například použit i v populárním prohlížeči Google Chrome. Tento engine může fungovat samostatně a nebo může být připojen k jakékoliv C++ aplikaci.[40] Při spuštění Node.js aplikace se zobrazí konzole, která je schopna přijímat JavaScriptový kód a následně ho interpretovat. JavaScriptový kód je také možné spouštět ze souborů. Prostředí poskytuje mechanizmus pro nalinkování dynamicky linkované knihovny (DLL) napsané v C++, využívající povinné rozhraní knihovny V8. Více o konkrétních funkčnostech lze dohledat v API dokumentaci [42].

Node.js podporuje npm, [41] což je balíčkový systém pro JavaScript. npm umožňuje jednoduchými příkazy instalovat, sdílet a distribuovat JavaScriptový kód. Umožňuje také spravovat jednotlivé závislosti v rámci projektu. [43]

2.3.8 Electron

Electron je framework používající Node.js a Chromium (open source webový prohlížeč[44]). Díky této kombinaci jádra frameworku umožňuje Electron vytvářet aplikace pomocí webových technologií. S použitím HTML a CSS kódu je možné data zobrazit díky Chromium a kvůli Node.js nad nimi JavaScriptovým kódem provádět libovolné operace. [45]

2.3.9 Handlebars

Handlebars je JavaScriptová knihovna pro usnadnění vkládání dat mezi HTML tagy. Zavádí textové proměnné, které se při volání API nahradí touženými daty. [46]

2.3.10 jQuery

jQuery je populární JavaScriptová knihovna ulehčující práci s HTML kódem. Její API nabízí jednoduché metodiky pro výběr HTML prvků, jejich změn

a animací. Také zlehčuje AJAXová volání či ulehčuje navázání JavaScriptového kódu k interpretaci při událostech vyvolaných uživatelem (např. kliknutí na prvek, posun myši nad prvek apod.) [47]

2.3.11 SAGE2

SAGE2 (Scalable Amplified Group Environment) je server využívající webových technologií (HTML, CSS, JavaScript) pro vzdálenou kolaboraci jednotlivých stran. Umožňuje na straně serveru spouštět aplikace nebo zobrazovat sdílené obrazovky jednotlivých klientů ve vysokém rozlišení. Jednotliví klienti se mohou připojit na server přes webový prohlížeč a odtud sdílet svoji obrazovku nebo spouštět vzdálené aplikace. Obrazovky připojené k serveru SAGE2 poté tyto prvky zobrazují.[48][49] Pro SAGE2 je možné psát vlastní aplikace, které lze spouštět a zobrazovat. Aplikace musí být naimplementované v JavaScriptovém kódu a interpretovatelné z jednoho souboru. Zdrojové kódy aplikací musí být dostupné na serveru.[50] V prostorách ČVUT existuje SAGElab laboratoř, ve které je takovýto SAGE2 server dostupný. Je k němu připojeno 20 obrazovek v celkovém rozlišení 9600px x 4320px [51].

2.3.12 Websocket protokol

Websocket protokol umožňuje oboustrannou komunikaci mezi klientem, a serverem, který takovouto komunikaci povolil. Jedná se o vrstvu nad TCP protokolem, spočívající v prvotním handshake a následné sériové komunikaci formou zpráv. Účelem tohoto protokolu je poskytnout mechanizmus oboustrané komunikace pro aplikace založené na prohlížečovém principu (např. Aplikace v Electron, či SAGE2) bez nutnosti navázání opakovaného HTTP spojení. [52]

2.3.13 CSV a XML

CSV a XML jsou populární formáty pro ukládání dat. V této práci je CSV použit pro zaznamenávání výsledků testování a XML pro metadata popisující jednotlivé entity v aplikaci. Více o CSV v [53] a o XML v [54].

2.3.14 OpenCV

OpenCV je open source knihovna v C++ s hlavním zaměřením na zpracování obrazu. Knihovna se dělí do modulů, které lze většinou nezávisle použít (některé moduly vyžadují ke své funkčnosti jiné moduly). Protože OpenCV je open source, je možné si zvolené moduly přikompilovat přímo do svého programu, nebo je využívat formou API volání srkze dynamicky či staticky nalinkovanou knihovnu. Za zmínku stojí moduly:

- **core**

Základní modul, který především definuje jednotlivé struktury (například třídu pro matice, která reprezentuje obraz) a operace nad nimi.

- **highgui**

  Modul umožňuje otevírat soubory s videem nebo obrázky a také je ukládat. Možné je i obrazy zobrazit na monitoru.

- **imgproc**

  Modul poskytuje například transformace s obrázky, jako je zvětšování/zmenšování nebo nabízí filtraci - vyhlazení, rozmazání. Hlavně také umožňuje vyhledávat kontury, vypočítávat jejich konvexní obálku, nebo také poskytuje vyhledávání hran.

- **objdetect**

  Modul zaměřen na detekci konkrétních objektů v obrazu.

- **calib3d**

  Modul poskytuje práci se stereo kamerou, umožňuje ji kalibrovat a případně zpracovat záznam tak, že jeho část rekonstruuje do trojrozměrného prostoru.

Další moduly se zaměřují třeba na učení strojů, tvoření obrovských panoramatických fotografií či vyhledávání důležitých bodů zájmu. Více funkcionalit se dá vyčíst z API dokumentace. [56] V této práci jsou využívány moduly core, highgui, imgproc a objedetect, které jsou vyžadovány pro správnou funkčnost testovaných implementací algoritmů. Moduly jsou zkompilovány jednotlivě, jako DLL, které se pak dynamicky linkují při nahrávání.

2.4 Volně dostupné implementace algoritmů počítačového vidění

Existuje rozmanité množství volně dostupných implementací algoritmů počítačového vidění. Pro C++ např. VLFeat[74], OpenCV[75], VXL[76], Point Cloud Library[77], CVIPtools[78], v Javě např. BOOFCV[79], pro C# AForge.NET[80] nebo SimpleCV[81] pro Python. Na doporučení vedoucího práce je využito pouze knihovny OpenCV. Tyto implementace jsou dostatečně populární a dostatečně zdokumentované.

Nyní je potřeba zvolit množinu různých implementací algoritmů, které detekují a kategorizují překážky. Na základě vlastností zvolené množiny nebo individuálních implementací bude definovaná porovnávací metodika, která bude brát ohled i na jiné implementace algoritmů počítačového vidění. Porovnávací metodika tedy bude dostatečně obecná, aby se dala aplikovat na libovolné relevantní implementace algoritmů. Později také tato zvolená množina algoritmů bude porovnávána v prototypu vzniklém v rámci této práce.

Práce se pro jednoduchost zabývá algoritmy detekujícími jeden snímek, protože záleží na rozhraní a ne na samotných algoritmech, může si práce takovýto předpoklad klást. Knihovna OpenCV nenabízí příliš implementací, které jednoduše přímo detekují nebo kategorizují objekty [56]. Proto jsou některé implementace intuitivně zkonstruované tak, aby alespoň v některých případech představovaly použitelný algoritmus vhodný k porovnání. Protože implementace při použití v této aplikaci dodává uživatel, příliš na jejich naivnosti a nekorektnosti v případě demostrace aplikace nezáleží.

Pro tento rozbor a intuitivní konstrukci algoritmů nebyly použity fotografie pořízené robotickým vozítkem, ale fotografie pořízené běžným fotoaparátem. Oba dva zdroje snímků v této práci dodávají fotografie ve formátu JFIF a proto nemají na analyzované rozhraní vliv. Použitím jiných fotografií se navíc později potvrdí analyzované rozhraní, protože bude využit jiný zdroj snímků. Pro pozdější testováni budou samozřejmě použity snímky z robotického vozítka.

2.4.1 Algoritmus: Kontury + AABB

Tento složený algoritmus se skládá ze čtyř podstatných volání funkcí API: vyžaduje nahrání černobílého obrázku, což umožňuje funkce „imread“, dále pro smysluplné nalezení kontur je potřeba v obrazu nalézt hrany, toho se dá docílit například pomocí implementace algoritmu „Canny“[57]. Samostatné nalezení kontur poté proběhne implementací algoritmu popsaném v [58]. Poté se pomocí volání funkce „boundingRect“ vypočte AABB nalezené kontury a pokud obsah AABB je dostatečně velký, považuje algoritmus detekovanou věc za překážku. Část implementace složeného algoritmu by pak mohla vypadat jako v kódu 2.1.

V kódu 2.1 si také lze všimnout tří parametrů modifikujících tento algoritmus: „threshold1“ a „threshold2“ jsou parametry algoritmu „Canny“ pro citlivost při detekci hran (více v [59] a [57]) a „threhsold3“ je parametr komplexního algoritmu, který určuje jak moc veliká plocha AABB musí být, aby algoritmus označil detekci za překážku. U funkce „findContours“ jsou dva povinné modifikující parametry: „CV_RETR_TREE“ určuje, že detekované kontury se ukládají v hiearchické struktuře a „CV_CHAIN_APPROX_SIMPLE“ určuje, že nalezené kontury, pokud reprezentují např. úsečku se místo každého bodu uloží jen body hraniční, které úsečku určí. Druhý parametr nemá nijak velký vliv, akorát se může stát, že se ztratí přesnost pozice a tvaru při aproximaci úseček. Více o parametrech funkce „findContours“ v [60].

Kategorizace překážky je v tomto případě řešena tak, že pokud byla překážka nalezena, a plocha AABB je v určitém rozsahu definovaném dvěma dalšími parametry, tak je detekce označena jako překážka, která se dá objet. Pokud je plocha větší, než horní rozsah, jedná se o překážku, která se nedá objet a pokud je plocha menší, než spodní rozsah, nebo překážka nebyla nalezena, nejedná se o překážku. Obrázek 2.2 znázorňuje průběh tohoto algoritmu

Kód 2.1: Část C++ kódu pro Kontury + AABB

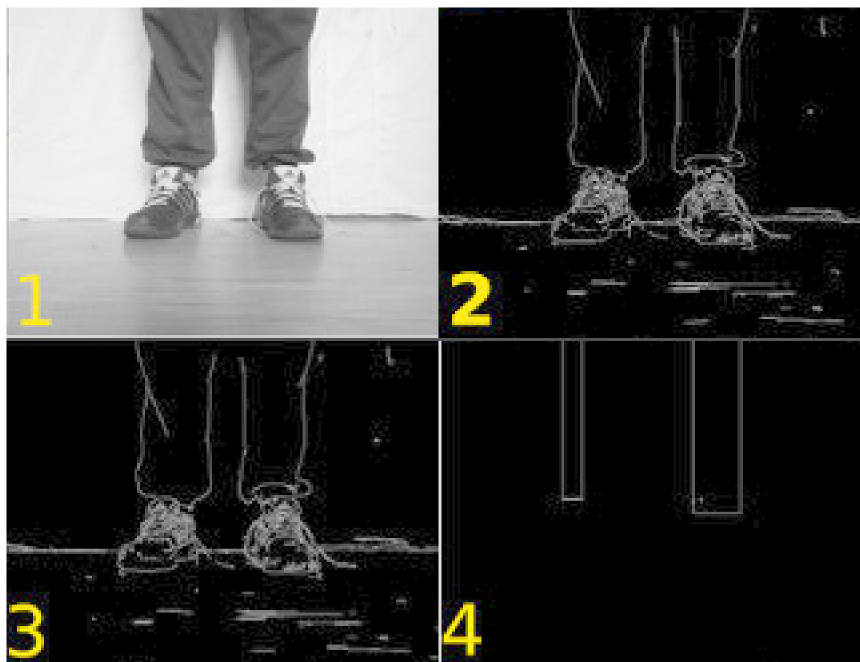
```
original = imread("image.jpg", 0)
Canny(original, edges, threshold1, threshold2);

vector<Mat> contours;
findContours(edges, contours,
          CV_RETR_TREE, CV_CHAIN_APPROX_SIMPLE);

for(int i = 0; i < contours.size(); i++){
          Rect bound = boundingRect(contours.at(i));
          int area = bound.width * bound.height;
                    if(area > threshold3){
                    //an obstacle found, do something
                    }
}
```

s parametry „threshold1“ = 50 a „threshold2“ = 150. První sekce znázorňuje pouhé nahrání černobílého obrázku, druhá výsledek funkce „Canny“, třetí výsledek funkce „findContours“ a čtvrtá jednotlivé AABB.

2.4.2 Algoritmus: Cascade classification

Tento algorimus je schopný přímo detekovat konkrétní překážku. Vyžaduje nahrání černobílého obrázku a XML souboru definující parametry překážky k detekci. Poté je schopna detekovat. [61]. Více o algoritmu v [64] nebo [63]. Výhodou této implementace je, že přijímá definici jakékoliv překážky koreknktě popsané v XML souboru. OpenCV nabízí aplikaci, která umí takováto data vytvořit na základě trénování a testování nad množinou obrazů [62]. Bylo by tak možné tuto implementaci použít třeba k detekování konkrétní plyšové hračky. Nevýhodou pak na druhou stranu je, že implementace je vhodná pouze pro detekci konkrétní věci a nikoliv obecně k detekci překážky. OpenCV distrubuuje svoje implementace s předpřipravenými naučenými definicemi například pro: lidské obličeje, spodní části těla, ruské státní poznávací značky nebo i obličeje kočky.

Zde práce využívá definic pro detekci spodní části těla, kvůli případnému reálnému použití robotickým vozítkem. Část implementace složeného algoritmu by pak mohla vypadat jako v kódu 2.2. Zde nedochází k žádným mezikrokům předzpracovávající obraz před detekcí. Metoda „detectMultiScale“ má volitelné parametry popsané v [61]. Rozhraní implementace určuje detekovanou překážku také AABB, jako v minulém algoritmu. Nyní však není nutné přepočítávat plochu obdelníku k určení překážky, protože metoda „detectMultiScale“ překážku už určila.



Obrázek 2.2: Vizualizace postupu algoritmu Kontury + AABB.

Kód 2.2: Část C++ kódu pro Cascade classification

```
image = imread("image.jpg", 0);
CascadeClassifier cascade("haarcascade_lowerbody.xml");
std::vector<Rect> legs;
cascade.detectMultiScale(image, legs);

for ( size_t i = 0; i < legs.size(); i++ ){
//this is a lower body obstacle
}
```

Kategorie u této detekce je implicitní, protože detekuje spodní část těla. Nalezne-li implementace překážku, překážka se dá objet, nenalezne li implementace nic, nejedná se o překážku.

2.4.3 Algoritmus: Hough + Kontury + Area

Jedná se o další naivní složený algoritmus. Nejprve vyžaduje nahrání černobílého obrazu, poté detekuje hrany pomocí implementovaného algoritmu Canny [57]. Z určených hran poté vyhledává úsečky pomocí funkce „HoughLinesP“ která implementuje pravděpodobnostní Hough transformaci pro detekci přímek [65]. Více o tomto algoritmu v [66]. Nalezené úsečky poté vykreslí do prázdného obrázku, kde se z úseček detekují kontury funkcní „findContours“ a naposled zkontroluje, zda-li plocha obepsaná konturami přesahuje určitou mez. V takovém případě pak považuje detekovanou věc za překážku. Část implementace složeného algoritmu by pak mohla vypadat jako v kódu 2.3. U tohoto algoritmu je dokonce až osm modifikujících parametrů, „threshold1“, a „threshold2“ opět určují citlivost detekce hran (více v [59] a [57]), „rho“, „theta“, „threshold3“, „minLineLength“ a „maxLineGap“ jsou parametry algoritmu pro detekci úseček - více v [66] a [65]. Poslední parametr „threshold4“ je parametrem celého algoritmu, který určuje jak veliká plocha musí být, aby byl objekt detekován jako překážka. Kategorizace překážky je řešena stejným způsobem, jako u algoritmu Kontur + AABB. Obrázek 2.3 znázorňuje průběh tohoto algoritmu s parametry „threshold1“ = 50, „threshold2“ = 150, „rho“ = 1, „theta“ = 1, „threshold3“ = 50, „minLineLenght“ = 200 a „maxLineGap“ = 40. První sekce znázorňuje pouhé nahrání černobílého obrázku, druhá výsledek funkce „Canny“, třetí výsledek funkce „HoughLinesP“ a čtvrtá nalezené kontury, ze kterých se počítá oblast.

OpenCV ještě nabízí rozhraní pro práci se stereo kamerami. Kamera se musí kalibrovat a poté je možnost rekonstruovat body v 3D prostoru. Žádnou implementaci pro rozpoznání překážky nenabízí. Byl bych schopen opět vymyslet nějaký naivní algoritmus, který funkcionalitu spojuje a nekorektně rozpoznává překážky například podle objemu AABB rekonstruovaných bodů. Jenže takovýto algoritmus, který pracuje se stereo obrazem by mezi ostatními byl jen jeden a porovnání s kategoricky jinými algoritmy nemusí dávat smysl. Vstupní data této teoretické implementace jsou jiné, než u ostatních implementací (2 fotografie vs. 1 fotografie).

Rozhodně ale aplikace se stereo záznamem počítá a plně se jeho kompatibilitě v rozhraní věnuje.

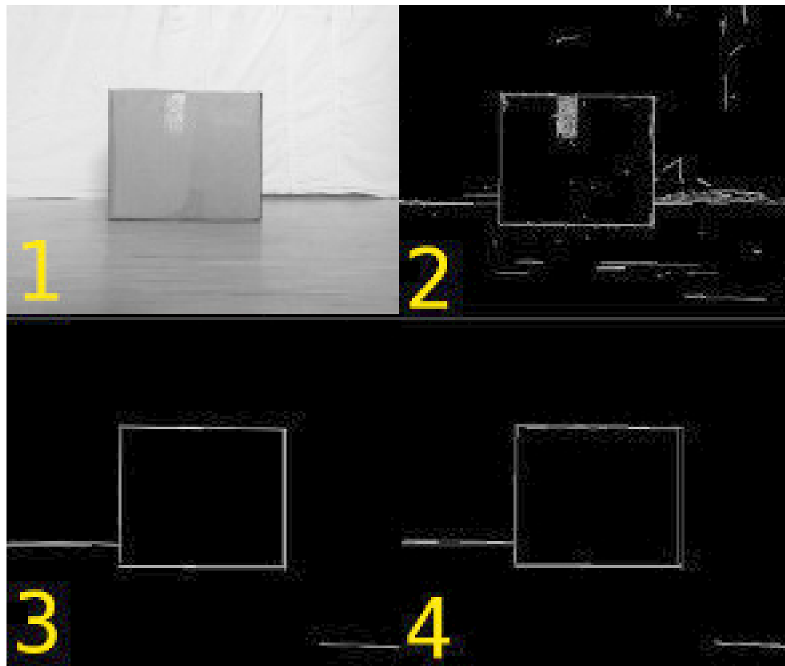
2.4.4 Rozhraní

Pro rozhraní těcho tří implementací vyplývá:

- Implementace jsou v C++ kódu,

Kód 2.3: Část C++ kódu pro Hough + Kontury + Area

```
image = imread("image.jpg", 0);
Mat edges;
Canny(image, edges, threshold1, threshold2);
vector<Vec4i> lines;
HoughLinesP(edges, lines, rho,
        theta * CV_PI/180, threshold3,
        minLineLength, maxLineGap );

Mat empty = Mat::zeros(data->image->size().height,
                       data->image->size().width,
                       data->image->type());

for( size_t i = 0; i < lines.size(); i++ ){
  Vec4i l = lines[i];
  line( empty,
          Point(l[0], l[1]),
          Point(l[2], l[3]),
           Scalar(255,255,255),
           1, CV_AA);
}

vector<vector<Point>> contours;
findContours(empty, contours, CV_RETR_TREE,
         CV_CHAIN_APPROX_SIMPLE);
for(int i = 0; i < contours.size(); i++){
  data->area = contourArea(contours.at(i));
  if(data->area > threshold4){
          //this is an obstacle
  }
}
```



Obrázek 2.3: Vizualizace postupu algoritmu Hough + Kontury + Area.

- vyžadují svůj paměťový prostor,
- vyžadují čtení souborů,
- mohou mít paramery různých primitivních datových typů
- a vyžadují přikompilované nebo dynamicky či staticky nalinkované relevantní OpenCV moduly.

Kapitola 3

Návrh

Tato kapitola se dělí na čtyři části. V první části se věnuje definici metody, kterou se implementace algoritmů budou v aplikaci porovnávat. V druhé části definuje požadavky, které aplikace musí splňovat a určí metodu, kterou se zaručí kvalita. Třetí část se věnuje návrhu architektury tvořeného softwaru a poslední část je věnována návrhu aplikační části pro SAGE2.

3.1 Porovnávací metodika

Jak je zmíněno v úvodu, tato práce se zabývá algoritmy počítačového vidění a jejich využitelností pro rozpoznávání okolí. Tento fakt také ovlivňuje určení porovnávací metodiky. Existují kategorie vlastností, které jsou nezávislé na účelu algoritmu. Z čistě teoretické části např. složitost algoritmu v nejhorším případě vstupu nebo paměťová složitost apod. Z praktické části může být zajímavé např. velikost implementace v bytech či doba, kterou trvá průměrný běh na určitém hardwaru. Dále jsou kategorie vlastností, které se dají přidělovat exkluzivně implementacím algoritmů počítačového vidění pro rozpoznávání okolí - např. použitelnost v noci nebo schopnost detekce překážky.

Protože účelem aplikace je právě porovnávání implementací algoritmů z této kategorie, zvolil jsem následující vlastnosti, ve kterých se budou implementace porovnávat:

- **detekce překážky**

  Atributem vlastnosti je celé číslo. Vlastnost popisuje schopnost detekovat překážku. Číslo reprezentuje počet nalezených překážek. Tato kategorie je naprosto kritickým ukazatelem použitelnosti. Pokud se má libovolný robot pohybovat v prostoru a nebude schopen rozpoznat překážku, nebo ji detekuje na špatném místě, bude mít milnou představu o okolí a s největší pravděpodobností do překážky naboura a něco poškodí. V opačném případě implementace s vysokou úspěšností v této kategorii je silným kandidátem pro konečné využití.

- **kategorizace překážky**

  Překážky se budou dělit do libovolného počtu typů: například na ty co se dají vozítkem objet, na ty co ne. Atributem vlastnosti je celé číslo, které specifikuje kategorii. Např. 0 znamená překážku, co se dá objet, 1 překážku, která se nedá objet atd. Tuto vlastnost si uživatel určuje sám, protože kategorii překážky detekují jeho dodané implementace a tyto detekce jsou porovnávány s kategoriemi jeho dodaných dat, ve kterých kategorii definoval.

- **čas**

  Poslední, významnou vlastností je doba běhu implementace algoritmu nad určitými daty. Atributem této vlastnosti je čas implementace strávený jednorázovou přípravou (např. nahrání XML souboru), čas strávený nad nahráváním zdrojů pro konkrétní detekci (např. nahrání obrázku), čas strávený nad detekcí překážky, čas strávený nad kategorizací překážky, čas strávený nad uvolňěním zdrojů pro konkrétní detekci (např. uzavření obrázku, případný zápis dat do paměti umělé inteligence apod.), čas strávený nad uzavřením algoritmu (např. uzavření XML souboru, uložení mezivýpočtů apod.) a čas strávený nad snímkem celkově.

  Vlastnost se bude měřit v milisekundách. Toto rozhodnutní není nijak kritické, protože jednotka by se v případě nevhodnosti dala jednoduše převést. Zvolené byly milisekundy, protože odhaduji, že běžné kamery zaznamenávají video mezi 20 - 60 FPS (frames per second, snímků za sekundu). Kamery z robotického vozítka analyzovaném v této práci zaznamenávají video při 21 FPS[7].

  Pokud by se implementace algoritmů testovaly na cílovém hardwaru, bylo by zajímavé, kolik milisekund robot strávil nad jedním snímkem, tedy zda-li by stíhal zpracovávat data v reálném čase. Na jeden snímek by při odhadované frekvenci měl totiž zhruba 16-50 milisekund. V případě zmíněného vozítka by to bylo 47ms (časové intervaly vyplývají ze základního vztahu frekvence).

  Implementace, kterým by detekce trvala příliš dlouho jsou v reálném případě nepoužitelné. Pokud by detekce nad jedním snímkem trvala např. dvaced sekund, tak by libovolný robot byl schopný dělat motorické rozhodnutí jednou za zmíněných dvacet sekund a navíc by musel rozhodnout o kolik se může posunout a byl by v podstatě neschopný detekce pohybujících se překážek, protože by je nejspíše nestihl ani zaznamenat (např. kutálející se míč kompletně přes celý záběr během výpočtu staršího snímku). Zde především záleží na cílové platformě využívající tuto implementaci, účel robota definuje takovouto hranici doby běhu a tím i použitelnost testované implementace.

> Ještě si je u této vlastnosti potřeba uvědomit, že čas strávený implementací nad testovacími daty je na hardwaru, který testování provádí. Pokud to není cílový hardware, který by implementace reálně používal, je nutné, spíše než čas absolutní, porovnávat čas vzájemných implementací, tedy studovat, jaký algoritmus je rychlejší a který je pomalejší.

Aplikace bude zaznamenávat výsledky iterací algoritmů nad jednotlivými snímky ve výše zmíněných kategoriích. Zpracování pak bude spočívat v agregaci jednotlivých výsledků pro každý algoritmus zvlášť. Výsledky bude reprezentovat tabulka přehledu zobrazující průměrné časy, úspěšnosti klasifikace překážky a kategorie. Úspěšnost klasifikace znamená poměr počtu správně určených atributů ku počtu všech určování (špatné i správné). Pro možnost takto testovat implementace musí k datům (ke každému obrazu) existovat metadata, která říkají pravdu o dané fotografii, tedy informace o existenci a kategorii překážky. Pro měření času nejsou metadata potřeba.

3.2 Požadavky na aplikaci

V této části se určují funkční a nefunknčí požadavky na aplikaci a případy užití. Funkční požadavky jsou takové, které definují funkčnost aplikace, tedy možnosti, které aplikace musí nabízet. Nefunknčí požadavky jsou požadavky, které aplikaci limitují, ale neposkytují funkčnost. Případy užití jsou cílové úkony, které lze v aplikaci provést.

Pro zajištění kvality tohoto software se využije metoda mapování případu užití na požadavky. Tato metoda umožňuje zaručení především funkčních požadavků. Funguje tak, že se určí typické případy užití, které nadefinované funkčnosti pokrývají. Je potřeba určit dostatečné množství tak, aby pokrylo veškeré požadavky. Pokud se při testování podaří všechny případy užití provést, považují se pokryté funkční požadavky za splňěné.

Z analýzy a zadání práce plynou následující požadavky:

Funkční požadavek FP1 Aplikace umožňuje definovat a využívat definované algoritmy

Funkční požadavek FP2 Aplikace umožňuje definovat a využívat fotografie ke zpracování algoritmy

Funkční požadavek FP3 Aplikace měří jednotlivé běhy uživatelem dodaných algoritmů nad uživatelem dodanými fotografiemi a naměřená data ukládá.

Funkční požadavek FP4 Aplikace porovnává (metodikou zmíňěnou výše) agregované jednotlivé běhy konkrétních implementací algoritmů Aby uživatel mohl určovat vhodné algoritmy pro jeho potřeby, potřebuje formou uživatelského rozhraní porovnané výsledky běhů zobrazit.

Funkční požadavek FP5 Aplikace umožňuje uživateli permutovat nad definovanými parametry algoritmu Tato funkčnost slouží k nalezení optimálních parametrů implementace, tak aby podávala nejlepší výsledky

Nefunkční požadavek NP1 Aplikace podporuje velmi vysoké rozlišení v serverovém prostředí SAGE2 Algoritmů koneckonců může být velké množství a tabulka přehledu tím pádem větší, než je běžná obrazovka schopna zobrazit. Přehled ve velmi vysokém rozlišení bude pro takovéto případy vhodný.

Dále jsou určeny další požadavky, které jsou doplněny tak, aby software tvořil smysluplný celek.

Funkční požadavek FP6 Aplikace umožňuje spravovat fotografie a algoritmy Důležité pro užitečnost aplikace je poskytnout uživateli možnost měnit data, která se testují nebo algoritmy, které jsou porovnávány. Spravováním rozumím možnost mazat, a přidávat testované forografie či implementace algoritmů ať už dávkově, nebo jednotlivě.

Funkční požadavek FP7 Aplikace nabízí možnost výběru algoritmů a dat, které se mají testovat. Aplikace poskytuje uschovávání rozmanitých algoritmů a dat, ale při testování uživatel nemusí chtít v testu porovnávat všechny dostupné algoritmy a data.

Nefunkční požadavek NP2 Uživatelské rozhraní aplikace je v anglickém jazyce. Tímto požadavkem usnadním dostupnost pro uživatele. Základní angličtinu považuji za lehký společný jazyk, který by každý uživatel této aplikace měl znát.

Nefunkční požadavek NP3 Aplikace, nebo její části, které mohou být open source, by tak měly být publikovány Protože aplikace bude prototypem, považuji za nutné, aby byla open source pro případné úpravy uživatelem. V případě, že by potřeboval měřit další veličiny k porovnání, tak aby si aplikaci mohl upravit, nebo pokud by chtěl porovnávat data jinak, aby měl možnost.

Funkční požadavek FP8 Aplikace umožňí uživateli provádět testy paralelně. Moderní hardware typicky obsahuje alespoň dvoujádrové procesory a vícevláknové testování urychlí celkový proces.

Zde se definují případy užití, které pokrývají veškeré funkční požadavky. Detailnější scénáře případů užití jsou k nalezení v kapitole Testování.

- Přidání implementace algoritmu (UC1)
- Přidání dvou implementací algoritmu, které iterují nad parametry (UC2)
- Přidání fotografie (UC3)
- Přidání více forografií (UC4)
- Odebrání algoritmu (UC5)
- Odebrání fotografie (UC6)
- Spuštění testování s více vlákny (UC7)
- Spuštění testování s jedním vláknem (UC8)
- Spuštění vybrané množiny algoritmů nad vybranou množinou dat (UC9)
- Zobrazení obrazovky porovnávající jeden test (UC10)

Tabulka 3.1: Mapování případů užití na funkční požadavky

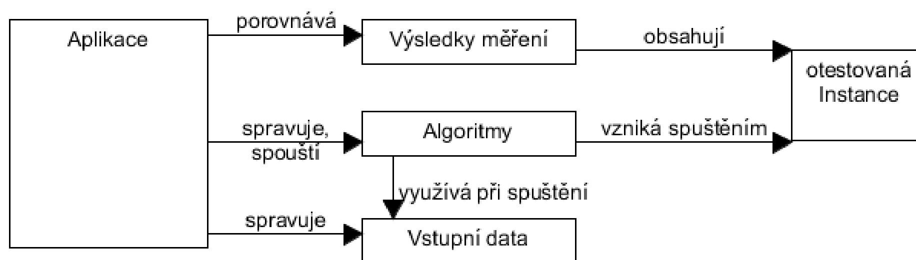
	UC1	UC2	UC3	UC4	UC5	UC6	UC7	UC8	UC9	UC10
FP1	x	x								
FP2			x	x						
FP3							x			
FP4										x
FP5		x								
FP6		x		x	x	x				
FP7									x	
FP8								x		

Důležité je, aby každý případ užití (sloupec) pokryl alespoň jeden funkční požadavek, jinak by takovýto i úspěšně provedený případ užití nic neprokázal. Také je důležité, aby každý funkční požadavek (řádek) byl pokryt alespoň jedním případem použití. Pokud by se tak nestalo, nebyly by všechny případy pokryty a tím pádem by se nedala kvalita zaručit. V tomto případě jsou obě podmínky splňěny a tím pádem při úspěšných testech bude kvalita zaručena.

3.3 Architektura

Funkční a nefunknční požadavky zásadně ovlivňují návrh architektury softwaru. Cílem je navrhnout takovou architekturu, aby splňovala veškeré explicitní funknčí a nefunkční požadavky. Dobrým zvykem je dodržovat i jiné implicitní požadavky na zdrojový kód (případně architekturu) jako jsou: dobrá čitelnost (vhodné názvy entit), dobrá udržovatelnost kódu - prostor pro jednoduché přidávání a změnu funkčností a jiné. Tyto implicní požadavky však nejsou povinné, protože nemají vliv na současné požadavky aplikace. Jejich cílem je usnadnít implementaci budoucích změn (jak architektury, tak funkčnosti) v softwaru. O těchto implictních požadavcích se zmiňovat nebudu, protože jich je velké množsví, ale ty které ze své zkušenosti znám a považuji za vhodné při vývoji softwaru vyujžiji.

Definice požadavků a případů užití nám již naznačuje, jak by software mohl vypadat. V jednoduchém doménovém modelu (obrázek 3.1) se o tom můžeme přesvědčit.

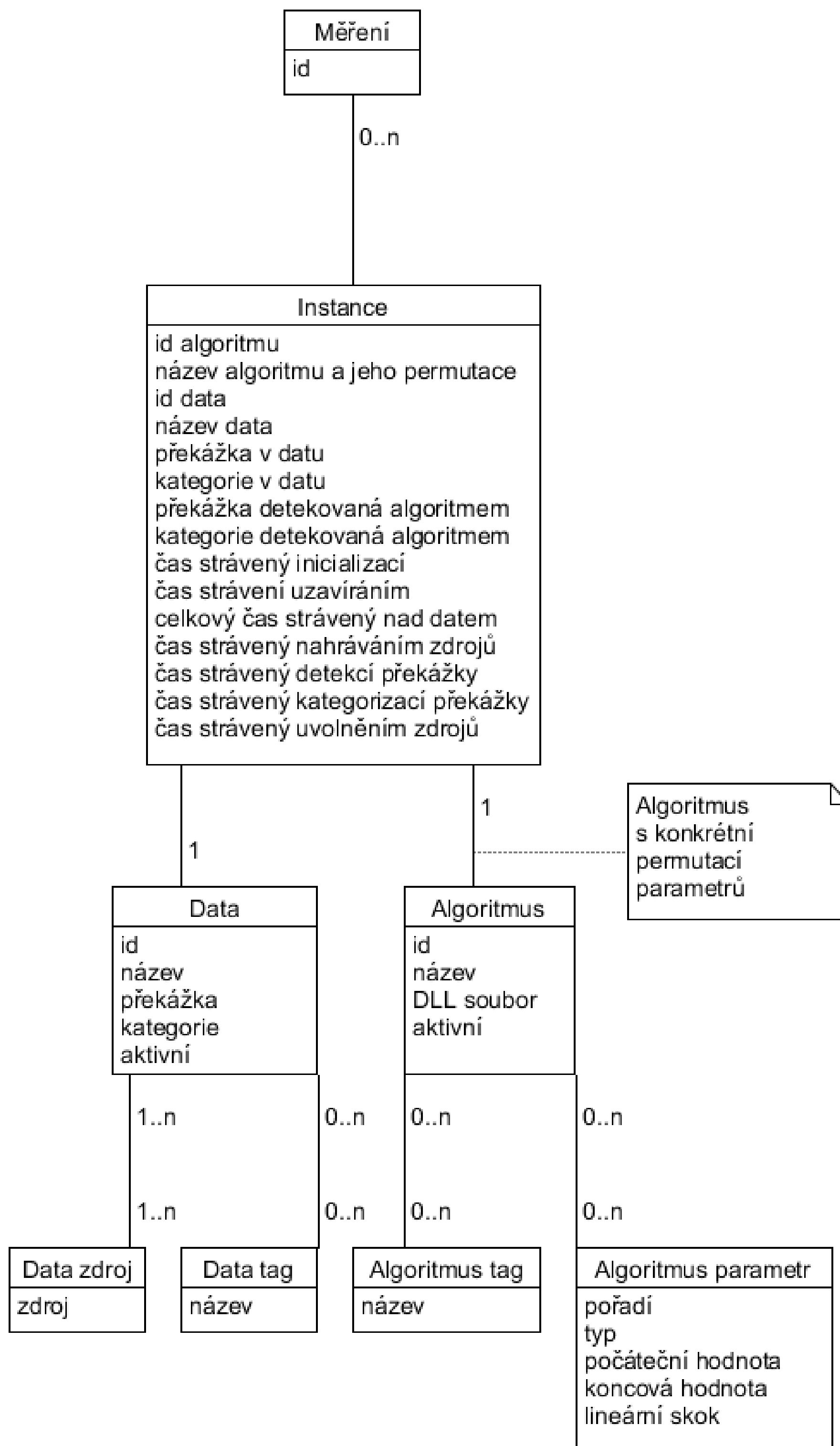


Obrázek 3.1: Jednoduchý doménový model

Existuje tedy aplikace, která spravuje algoritmy a data. Shromažďuje výsledky měření a umožňuje je porovnat. Aby mohla aplikace ukládat metadata k fotografiím a uschovávat výsledky testů, bude potřeba zavést datové úložiště.

3.3.1 Databáze

Z databázového modelu (obrázek 3.2) vyplývá, že aplikace počítá s osmi datovými entitami. Obsahuje měření, které aplikace porovnává. Jednotlivá měření



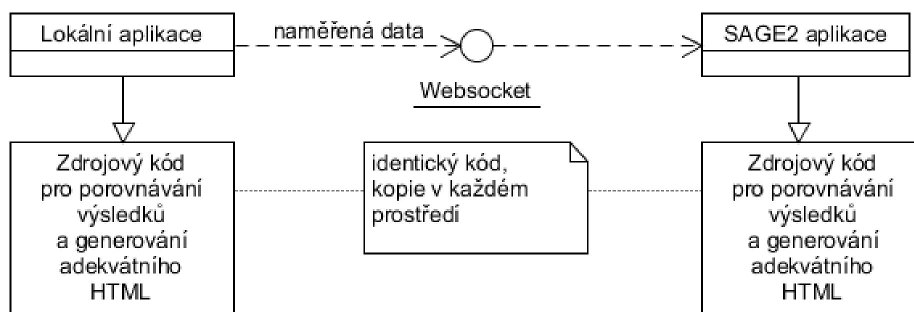
Obrázek 3.2: Databázový doménový model

obsahují všechny spuštěné instance a naměřené hodnoty. Aplikace také pracuje s daty, které definují konečné zdroje a mohou jim být přidleny tagy. Obdobným způsobem pracuje i s algoritmy, kterém také mohou být přiděleny tagy a mohou vyžadovat iterace parametrů různých typů. Libovolné množství parametrů u algoritmu již vyplývá z analýzy použitých implementací v této práci, některé algoritmy vyžadovali i více parametrů, některé žádné. Proč ale data umožňují více zdrojů nemusí být hned zřejmé. Představme si, že uživatel chce testovat algoritmy pracující se stereo záznamem: k tomu je potřeba jeden pár fotografií. Tento návrh umožňuje takovouto věc provést tak, že pár zůstane pohromadě, zaštítěn entitou „Data“.

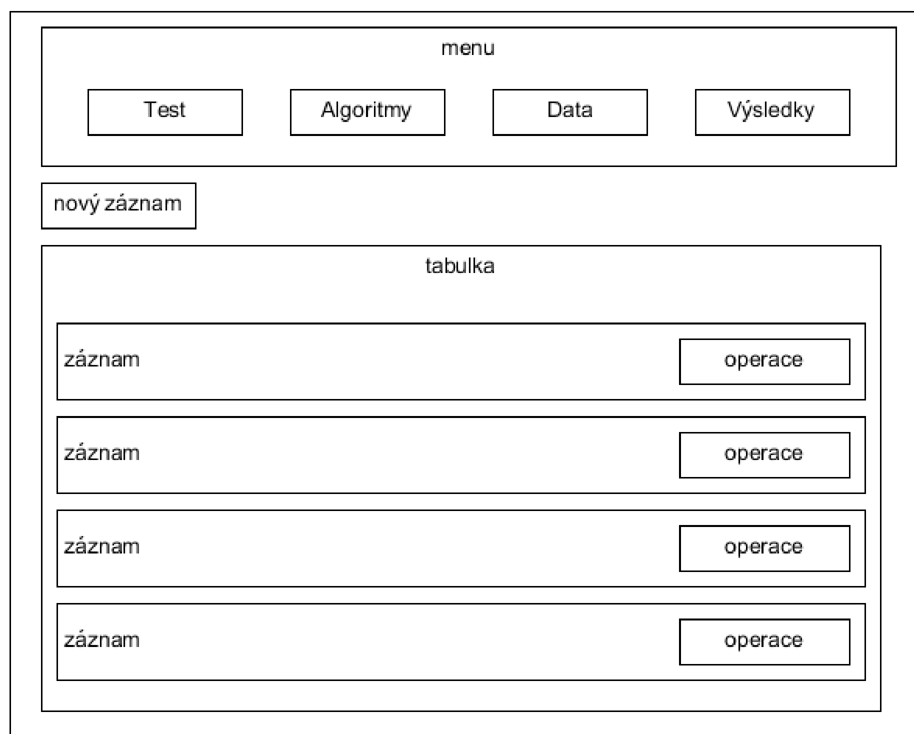
Přidělování tagů a vlastnost active pro algoritmy a data má význam pro výběr entit k testování. Přidělí-li uživatel k datům tag „překážka“ pouze tam, kde se vyskytuje a ve spuštění testu vybere takováto data (s tagem „překážka“), tak implementace algoritmů budou testovány pouze nad daty s překážkami. Obdobně bude mít možnost testovat pouze entity, které jsou aktivní (mají active nastaveno na „1“).

3.3.2 Grafické uživatelské rozhraní

Aby mohla aplikace výsledky zobrazit a porovnat, potřebuje grafické uživatelské rozhraní (GUI). Pro jednoduchou možnost spravovat algoritmy a data, spouštět testy a zobrazovat výsledky porovnání, aplikace poskytne grafické uživatelské rozhraní.



Obrázek 3.3: Obrazovka s formulářem

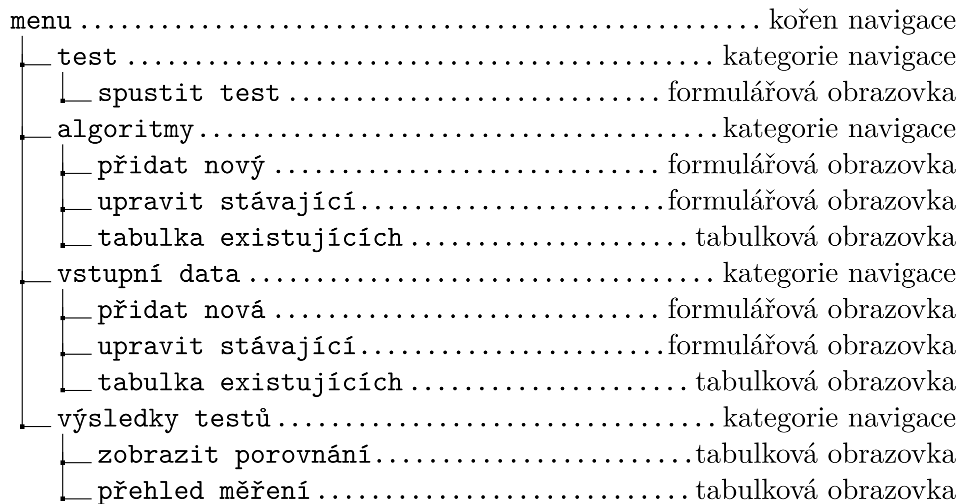


Obrázek 3.4: Obrazovka s tabulkou

Aplikace v podstatě nabízí dvě verze obrazovek. Obrazovka bude vždy zobrazovat menu ve vrchní části a ve spodní části obsah. Obsahem je buďto formulář (obrázek 3.3) a nebo tabulka (obrázek 3.4). Aplikace tak celkem nabízí 9 obrazovek. Jejich struktura je zobrazena v obrázku 3.5.

3.3.3 Obrazovka „spustit test“

Obrazovka „spustit test“ obsahuje formulář s nastavením testu. Uživatel má možnost ve formulářovém poli pojmenovat test, který se bude provádět. Výchozí hodnota tohoto pole je aktuální rok, měsíc, den a čas. Dále uživateli nabízí možnost vypisovat logy do souboru nebo konzole. Logy se zaznamenávají v průběhu testování. Slouží k uvědomění uživatele o aktuální činnosti testeru a případných chybách. Zpráva v logu je vždy na jednom řádku, obsahuje čas, kdy byla vypsána, její typ (ERROR, NOTICE, WARNING) a zprávu. Dále si ve formuláři může uživatel zvolit, jestli se má kontrolovat atribut „active“ a tím pádem vybrat jenom aktivní prvky k testování, nebo jestli se má ingorovat. Případně také může vypsat tagy, na základě kterých budou vybrány relevantní prvky. Tyto dva formulářové elementy pro výběr k testování jsou zde jak pro algoritmy, tak pro vstupní data. Posledním formulářovým po-



Obrázek 3.5: Struktura obrazovek

lem je pak určení počtu vláken, které mají jednotlivé testy provádět, výchozí hodnota je nastavená na 1. Uživatel pak může spustit tlačítkem testování. Obrazovka se nahradí prázdnou stránkou s žádostí o trpělivost, než se dokončí testování. Po skončení testu se zobrazí obrazovka s přehledem měření.

3.3.4 Obrazovky přehledu

Obrazovky přehledu („přehled měření", 2x „tabulka existujících") zobrazují tabulky s adekvátními sloupci k databázovým entitám a operace, které se nad nimi dají provádět. Pro algoritmy a data to je editace a odebrání. Pro výsledky měření je to zobrazit porovnání a vysílání do SAGE2 (více o SAGE2 později). „Tabulky existujících" také nabízí možnost přidání nové odpovídající entity. Tlačítko pro přidání se nachází nad tabulkou.

3.3.5 Obrazovky pro přidání/úpravu

Mezi obrazovkami přidání nové entity a upravení stávající je pouze jeden rozdíl: formuláře pro nové entity obsahují jedno pole navíc, a to id entity, tedy název souboru s metadaty. U formulářových prvků budou vždy přitomné vysvětlivky, případně i přiklady vyplnění.

Obrazovka s přidáním nového algoritmu ve formuláři obsahuje pole (mimo id):

Název algoritmu Pojmenováni algoritmu uživatelem.

Soubor implementace Soubor, ve kterém je uživatelova implementace algoritmu, jedná se o DLL. (více v implementaci)

Aktivní Možnost volby: ANO/NE, slouží pro výběr při testování

Tagy Možnost přidání libovolného množství tagů, také slouží při výběru testování

Atributy Libovolné množství atributů, nad kterými bude implementace iterovat. Jejich popis a chování je pospáno přímo u formulářového prvku a nebo v uživatelské příručce.

Obrazovka s přidáním vstupních dat obsahuje ve formuláři prvky (mimo id):

Název zdroje Uživatelovo pojmenování

Zdrojové cesty Libovolné množství zdrojů, které bude předáno implementaci algoritmu

Aktivní Možnost volby: ANO/NE, slouží pro výběr při testování

Tagy Možnost přidání libovolného množství tagů, také slouží při výběru testování

Počet překážek Počet překážek, které tyto data obsahují.

Kategorie překážky Kategorie překážek, které tyto data obsahují.

3.3.6 Obrazovka porovnání výsledků

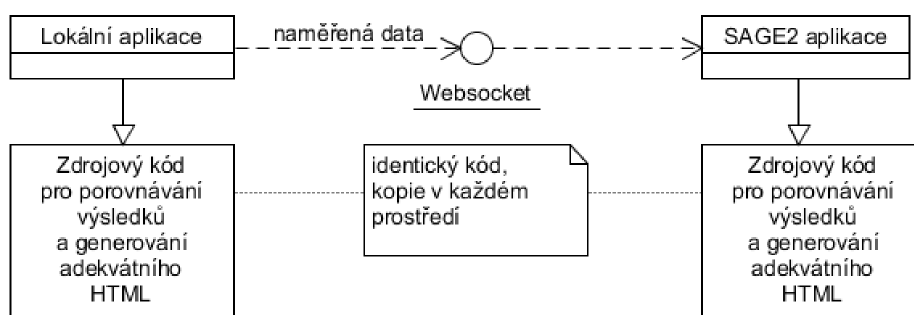
Obrazovka porovnání výsledků („zobrazit porovnání") obsahuje tabulku, která má na každém řádku jinou formu implementace algoritmu (podle iterace jejich atributů) která agreguje výsledky proběhnuté nad všemi relevantními daty. Zobrazuje název této formy a hodnoty atributů (pokud má), úspěšnost klasifikace počtu překážek a kategorie překážky, čas inicializace a čas uzavření implementace algoritmu, pak průměrné časy jednotlivých operacích nad zdroji. Všechny časy jsou zobrazeny v milisekundách. Po všech formách jednoho algoritmu následuje řádek, průměrující veškeré formy toho algoritmu. Tento řádek je zvýrazněn.

3.4 Podaplikace v SAGE2

Aby mohla aplikace zobrazovat porovnané výsledky i v prostředí SAGE2, potřebuje k tomu dceřinnou část pro SAGE2. Software se tedy dělí na dva samostatně spustitelné programy: lokální aplikaci, o které bylo zmiňováno doposud a vzdálenou aplikaci pro SAGE2.

Celý systém umožní prohlížet porovnávání výsledků jak lokálně, tak vzdáleně v SAGE2. Přibývá tím desátá obrazovka. Obrazovka bude zobrazovat status ve vrchní části, a tabulku porovnání ve zbytku, tedy spodní části. V podstatě se jedná o obrazovku porovnání výsledků, kde je místo menu zobrazen status připojení.

Protože k tomu, aby uživatel mohl ovládat SAGE2 potřebuje být připojen k síti. Tohoto předpokladu využiji a umožňím uživateli vysílat data z lokálního programu do SAGE2 aplikace k porovnání a zobrazení. Po připojení na SAGE2 vzdálenou aplikaci, pak bude uživatel moci vysílat v obrazovce „přehled měření“ jednotlivé výsledky a tím je zobrazit stejným způsobem, jako v lokálním programu, akorát ve velmi vysokém rozlišení. Obrázek 5.1 naznačuje, jak obě aplikace spolupracují.



Obrázek 3.6: Zobrazování na SAGE2

Kapitola 4

Implementace

Tato kapitola popisuje způsoby implementace, které byly použity a nad některými polemizuje. Kapitola se dělí do šesti částí. V první diskutuje o nutném rozdělení aplikace do několika částí. V druhé části popisuje způsob řešení databáze. Poté v dalších sekcích kapitoly popisuje řešení implementací jednotlivých částí aplikace popsaných v první části kapitoly. Nakonec ve dvou částích pak zmiňuje použité metody pro debugování vyvíjených částí a vytyčuje některé vlastnosti implementace.

Implementovace se zprvu zaměřila na kompletní datový tok od počátku až do konce. Nejprve vznikla základní obrazovka v GUI aplikaci, která nahrávala přídavný C++ modul a používala z něj funkci s parametrem. Modul dočasně modifikoval přijatý parametr a používal ho jako návratovou hodnotu. V SAGE2 prostředí vznikla prvotní aplikace, která přijímala spojení a textový řetězec, který vykreslovala. Po vytvoření takového proof of concept se postupně přidávala funkčnost ve všech částech tak, aby celý datový tok stále probíhal.

4.1 Základní rozdělení

Protože SAGE2 vyžaduje, aby všechny zdrojové kódy aplikací byly v JavaScriptu, tak i implementace každé lokální GUI části je v JavaScriptu. Kdyby tomu tak nebylo, tak by existoval zbytečně kód v jiném jazyce, který by stejnou logikou zpracovával naměřená data. Navíc webové technologie (HTML, CSS, JavaScript) jsou vhodné k rychlému vývoji grafických rozhraní. Porovnání výsledků testu a generace adekvátního HTML probíhá v nativním JavaScriptu, aby se tato část kódu vyhnula jakýmkoliv závislostem a dala se použít v libovolném JavaScriptovém prostředí. S tímto kódem pracují oba dva programy. Zobrazení vygenerovaného HTML pak řeší každý svým způsobem.

Také je zřejmé, že aplikace musí spouštět C++ kód, který může vyžadovat dynamické linkování (implementace vyžadující OpenCV DLL). To JavaScrip-

tové prostředí Node.js umožňuje formou dynamického linkování C++ přídavků (addon) implementující určité rozhraní.

Nabízí se tedy možnost jednotlivé algoritmy při každém přidání přikompilovat přímo do aplikace. Takováto možnost není vhodná, protože autor takového algoritmu musí znát základní rozhraní pro výmněnu dat mezi Node.js a C++ přídavným modulem, aby implementaci mohl korektně použít. Rozhraní je nevyhnutelné, ale připadá mi vhodnější ho udržet v rámci C++. Proto vznikl jeden C++ modul pro Node.js, který nahrává implementace algoritmů, spouští je a měří je (znázorněno v obrázku 4.1).

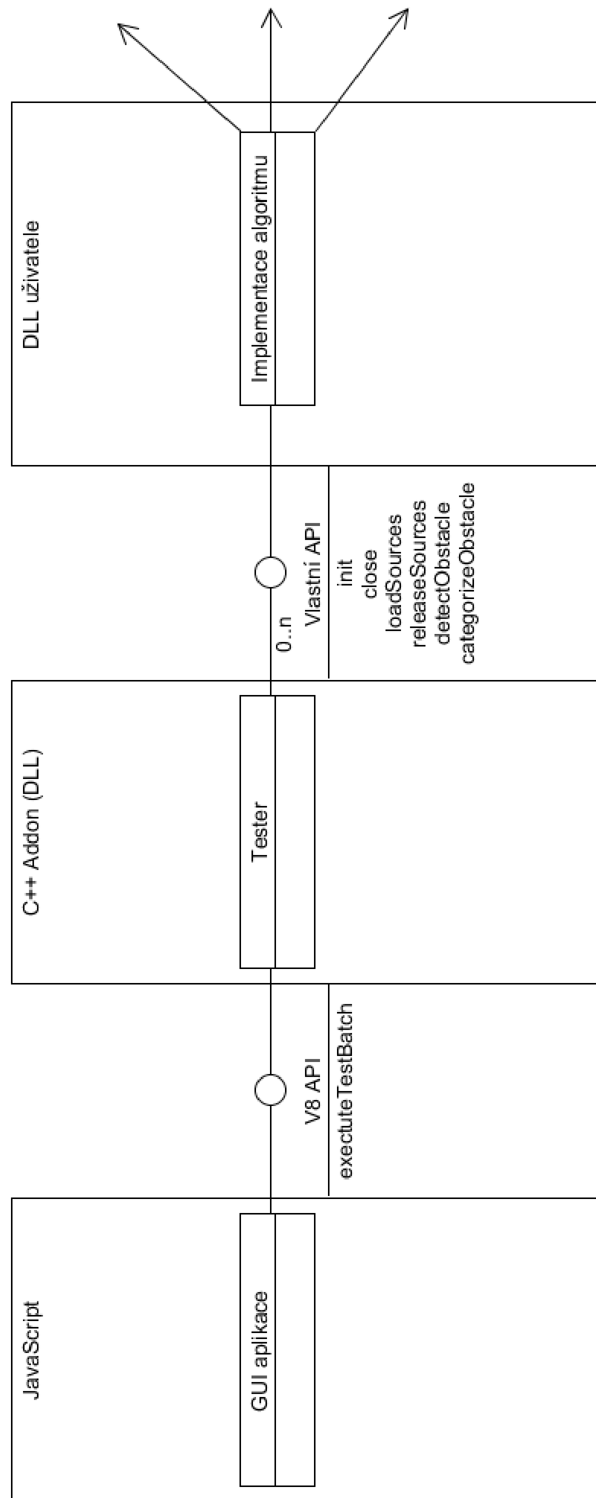
Při tomto navrhovaném řešení jsou implementace algoritmů odstíněny od rozhraní V8, které překládá C++ datové typy do JavaScriptové formy a naopak. Data, která implementace zpracovává tedy putují C/C++ rozhraním, pro uživatele známým prostředím (v C++ dodává algoritmy, předpokládám, že takovéto nativní rozhraní pro něj nebude problémové).

Lokální aplikace se tedy dělí do dvou částí: kód, který poskytuje GUI (implementované v JavaScriptu) a testeru, který spoští algoritmy a měří je (implementované v C++).

4.2 Databázové řešení

Z návrhu vyplývá, že aplikace využívá databázi. Okamžitě se nabízí možnost použít standardní relační databáze podporující jazyk SQL. Ale pozor:

Implementace algoritmů, dodané uživatelem jsou DLL soubory (více později), které mohou vyžadovat soubory ke zpracování k tomu, aby mohly korektně fungovat (fotografie, XML kódy definic objektů apod.). Ukládat takovéto soubory do databáze není dobrým zvykem a proto aplikace musí podporovat práci se soubory. Protože databázový model je triviální, dá se implementovat prací s XML a CSV soubory. Databázi pak tvoří triviální adresářová struktura (obrázek 4.2).



Obrázek 4.1: Rozhraní lokální aplikace

```
userdata ........................................... kořen databáze
├─ run .......................... adresář s metadaty pro spouštění testů
├─ data ................. adresář s metadaty pro vstupní data algoritmů
├─ algorithms ........................ adresář s metadaty pro algoritmy
└─ results .................... adresář s výslednými CSV sobory a logy
```

Obrázek 4.2: Adresářová struktura databáze

Pro každá vstupní data existuje XML soubor s metadaty pojmenovaný „X.meta“, kde X je identifikátor dat. Unikátnost tohoto atributu zaručuje zakázaná duplicita názvu souboru v jednom adresáři na hostujícím operačním systému. Pro každý algoritmus a nastavení spuštění testování platí to samé. Výsledky testů se ukládají v CSV souboru pod názvem „Y.csv“, kde Y je název testu. Každý záznam (řádek) pak reprezentuje jednu otestovanou instanci nad jedním datem.

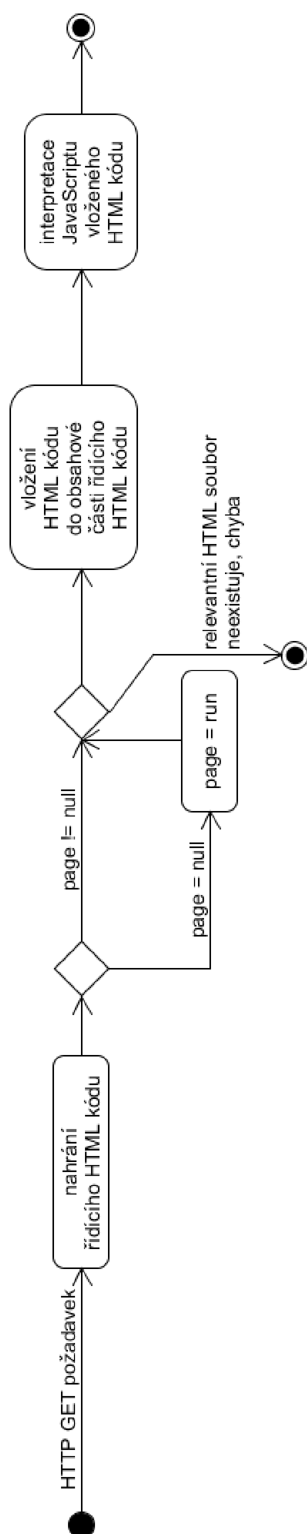
Tato implementace s sebou nese značnou výhodu: import a export algoritmů a dat je přímý, zatímco u relační databáze by muselo existovat rozhraní pro import a export. Ale přináší to i nevýhodu: pro výběr algoritmů, či dat (ať už výpis v UI, nebo testování) je třeba procházet celý adresář, každý soubor zkontrolovat a případně přečíst. V případě, že uživatel chce ať už data nebo algoritmy exportovat ši importovat, stačí, aby adekvátní metadata vkládal nebo vyjímal z relevantních adresářů. Jak tester, tak UI kód pracuje s databázovým adresářem a jeho aktuálním obsahem, tím pádem se v případě přikopírování dat nebo algoritmů změna v databázi projeví v aplikaci okamžitě.

4.3 Lokální aplikace

4.3.1 GUI aplikace

GUI aplikace je implementována v JavaScriptu, využívá prostředí Node.js a framework Electron. To umožňuje jednodušše udržovat obrazovky v HTML a CSS kódu. Jejich funkčnost je pak interpretována JavaScriptem umístěným přímo ve zdrojovém HTML kódu. Díky Electronu aplikace využívá HTTP systém navigace přes GET požadavky.

Každá obrazovka má tedy unikátní URL. Hlavní rámec aplikace nenabízí vložení HTTP adresy, jako běžné prohlížeče, čímž znemožňuje uživateli se pohybovat po nekorektních URL a proto je tento způsob navigace bezpečný. Logika výběru obrazovky k zobrazení je znázorněna na obrázku 4.3.



Obrázek 4.3: Logika výběru obrazovky

Framework při spuštění aplikace nahraje HTML kód z jednoho souboru a zpracuje ho (zobrazí data a interpretuje JavaScript). Aplikace využívá jednoho hlavního HTML kódu, který se vždy načítá. JavaScriptová část si přečte GET parametry a pokud mezi nimi existuje parametr „page“, vloží do obsahu relevantní HTML kód obrazovky. Ten opět obsahuje JavaScript, který obsah zpracovává. Tyto HTML kódy pro vložení existují 4: pro každou kategorii v menu jeden.

Algoritmy, data a výsledky ještě mají další rozhodovací logiku pro výběr obrazovek (porovnání, přidání entity, úprava entity), která je umístěna uvnitř HTML kódů, aktivuje se po vložení do hlavního kódu. Logika funguje obdobným způsobem, akorát se jedná o jiné GET parametry a místo nahrávání dalšího HTML kódu se upravuje již vložený HTML kód, podle potřebné funkčnosti.

Tento způsob udržuje zdrojový kód pro správu entit stejného typu v jednom souboru. Protože obrazovek není mnoho a obsahují jednoduchou logiku, tak je kód dostatečně krátký na to, aby se v něm efektivně programovalo. Společné funkčnosti, jako je například procházení adresáře a čtení metadat jsou abrstahovány do jednoho souboru a na adekvátních místech vloženy, aby se zdrojový kód vyhnul opakování.

Obrazovka s přehledem měření obsahuje ještě jednu speciální funkčnost. Při navigaci na tuto stránku se otevře nešifrovaný WebSocket, který přijímá jakékoliv spojení. V obrazovce se ještě nachází jedna z lokálních IP adres, port, na kterém byl WebSocket otevřen a status (zda-li se někdo připojil). Po přijmutí spojení se změní status na obrazovce a u každého měření se zobrazí tlačítko pro vyslání CSV dat přidruženého záznamu. Po stisku tohoto tlačítka se vyšle obsah souboru s měřením obalený hlavičkou (kterou SAGE2 vyžaduje) na vzdálenou stranu. WebSocket je nešifrovaný, protože se jedná o prototyp a zatím na takovouto funkčnost není požadavek. Otevřený socket přijímá jakékoliv připojení a v rámci této obrazovky si je pamatuje. Při vysílání dat se pro každé otevřené spojení pošle CSV kód s daty k porovnání.

Obrazovka pro spuštění testu vyplňěním formuláře vytvoří metadata soubor s konfigurací, nalinkuje přídavný modul pro testování a spustí v něm měření podle vytvořené konfigurace.

4.3.2 Přídavný modul pro testování

Přídavný model je naimplementován v C++ a má charakteristiky jako běžná DLL. Modulu implementuje jednu funkci „executeTestBatch“, která v začátku a konci musí implementovat rozhraní pro připojení na JavaScript, které je řešeno knihovnou V8. Modul tuto funkci také musí zaregistrovat, aby byla v JavaScriptovém prostředí dostupná. Modul využívá knihovny libuv, která je již dostupná, protože ji využívá Node.js a knihovnu pro parsování XML kódu: RAPIDXML.

Funkce volaná z tohoto modulu v JavaScriptovém prostředí vyhazuje výjimky v případě chyb a akceptuje jeden vstupní parametr: relativní cestu ke konfiguračnímu souboru.

Po zavolání funkce v modulu probíhá následující postup:

1. Transformuje JavaScriptový vstup a získá se relativní cesta ke konfiguračnímu souboru.
2. Přečte se konfigurační soubor a nastaví se prostředí.
3. Založí se soubor pro výsledky.
4. Pokud je nakonfigurováno logování do souboru, vytvoří se logovací soubor.
5. Vyberou se relevantní algoritmy, nahrajou se jejich DLL a pro každý parametr se iteruje a vytváří jednotlivé verze algoritmu, které si své parametry pamatují a sdílí nalinkovanou DLL.
6. Vyberou se relevantní data pro testování algoritmy.
7. Pro každý algoritmus se provede jeho incializace. Pokud se inicializace nezdaří, algoritmus se zneaktivní.
8. Pokud je specifikováno více jak jedno vlákno, vytvoří se specifikovaný počet vláken. Jinak testy provádí hlavní vlákno.
9. Každé testovací vlákno provádí testy (viz níže.)
10. Hlavní vlákno buďto testuje, nebo čeká na dokončení všech testovacích vláken.
11. Po skončení všech testů se pro každý algoritmus provede uzavření.
12. Výsledná měření se zapisují do CSV souboru.

Minimalistický pseudokód pro povedení jednoho testu znázorňuje kód 4.1.

Při testování se specifikovaným počtem vláken „1“ provádi testy hlavní vlákno a nevytváří žádné pracovní vlákna. Chtěl jsem umožnit pro případné rozšiřujicí práce způsob, jak se vyhnout paralelizmu. Představuji si totiž situaci, kdy uživatel potřebuje odpojit testovací modul, udělat samostatně spustitelnou testovací aplikaci a tu zprovoznit na cílovém hardwaru, který více vláken nepodporuje. Kdyby zakládání vláken bylo podmínkou, tato modifikace by byla o něco složitější. Navíc během postupu implementace testovaní nejdříve více vláken nepodporovalo a přirozeně testovalo na hlavním vláknu. Z výše zmíněného důvodu tato možnost zůstala.

Postup měření musí být bezpečný pro testování více vlákny a vyžaduje jednoznačné rozhraní a limitaci uživatelských algoritmů. Na implementaci algoritmu (DLL) se kladou pravidla:

Kód 4.1: Pseudokód provedení jednoho testu

```
dokud existuje neotestovaný pár verze algoritmu a dat
       algoritmus := neotestovana verze
       data := neotestovaná data
       pokud algoritmus.nahratData(data)
         algoritmus.ulozTrvaniNahrati(...)
         algoritmus.detekujPrekazkuUlozVysledek(...)
         algoritmus.ulozTrvaniDetekce(...)
         algoritmus.urciKategoriiUlozVysledek(...)
         algoritmus.ulozTrvaniKategorizace(...)
         algoritmus.uvolniData(...)
         algoritmus.ulozCasUvolnovani(...)
       jinak
       algoritmus.ulozCelkovyCasTrvani(...)
```

- **Zákaz používání globálních promněných.** Nejen, že je to nebezpečné pro vícevláknové testování, ale i v případě více verzí algoritmů (na základě iterací parametrů) si může uživatel nechtěně přepisovat data. Představme si například globální proměnou pro definici překážky XML kódem. Pokud by se XML kód pro překážku volil na základě iterovaného parametru, veškeré verze s touto knihovnou by měly jako definici posledně nahraný XML kód.
- **Veškeré alokování paměti na straně DLL algoritmu musí být opět korektně uvolněno.** Testovací modul dává prostor uživateli pro ukládání dat, ale zodpovědnost přenechává na něm.
- **DLL musí obsahovat minimálně 6 symbolů (funkcí) s přesnými názvy a parametry (viz níže).**
- **Obsah alokovaného parametru „localReference“ je určen pouze pro čtení.** Modifikovat se smí v metodách init nebo close. (Při testování více vlákny se při modifikaci jedná o kritickou sekci).

Kritická sekce je sekvence příkazů v programu, které pro správnou zamýšlenou činnost nesmí provádět více vláken najednou. Těmto problémům se dá předejít použitím synchronizačních primitiv. Typickou kritickou sekcí bývá inkrementace proměné. Při dodržení výše zmíňěných pravidel je rozhraní deklarované v kódu 4.2 bezpečné pro více vláken.

Kód 4.2: Deklarace rozhraní uživatelského algoritmu

```
int init(          const char * dirname,
                   void ** localReference,
                   const char ** parameters,
                   unsigned long long parametersCount);

int loadSources(          const char * dirname,
                          const char ** sources,
                          unsigned int count,
                          const void * localReference,
                          void ** sourceReference);

int detectObstacle(       const void * localReference,
                          void ** sourceReference);

int categorizeObstacle(   const void * localReference,
                          void ** sourceReference);

int releaseSources(       const void * localReference,
                          void ** sourceReference)

int close(         const char * dirname,
                   void ** localReference);
```

Funkce „init“ je volána jednou, a to na začátku, před testováním. Funkce by měla vrátit 0 v případě úspěchu, jinak bude inicializace modulem považována za neúspěšnou. Funkce jako první parametr přijímá řetězec s relativní cestou od současného pracovního adresáře (CWD) k adresáři s metadaty algoritmů, druhý parametr je ukazatel na ukazatel v paměti C++ modulu, zde si uživatel může alokovat svoji paměť, nejříve ale musí ukazatel jednou dereferencovat.

Čtvrtý parametr obsahuje velikost pole, kterým je třetí parametr. Třetí parametr je pole hodnot parametrů kde každý parametr je uložený v osmi bytech. Pokud byl požadavek na iteraci pouze čtyřbytový (např. int) nachází se hodnota v prvních 4 bytech. Jinak se hodnota nachází ve všech 8 bytech. Toto řešení umožňuje přímé přetypování, jak je v kódu 4.3 naznačeno.

Takovýto přístup vyžaduje, aby v metadatech algoritmu byly korektně nastaveny iterační typy ve správném pořadí a počtu. Uživatel by si vždy měl kontrolovat počet parametrů, ať už v metadatech algoritmu, nebo v implementaci samotné.

Funkce „close“ je také volána pouze jednou a to po skončení všech testů. Oba dva parametry jsou stejné jako v případě funkce „init“. Zde by uživatel

Kód 4.3: Příklad přečtení parametrů rozdílných velikostí

```
int init(const char * dirname,
void ** localReference,
const char ** parameters,
unsigned long long parametersCount){

if(parametersCount < 2) return 1;

int firstParam = *((int*) parameters[0]);
long long scndParam = *((long long*) parameters[1]);

return 0;
}
```

měl uvolnit jakoukoliv paměť, kterou ve funkci init nad „*localReference“ alokoval. Návratová hodnota se chová jako u funkce „init“.

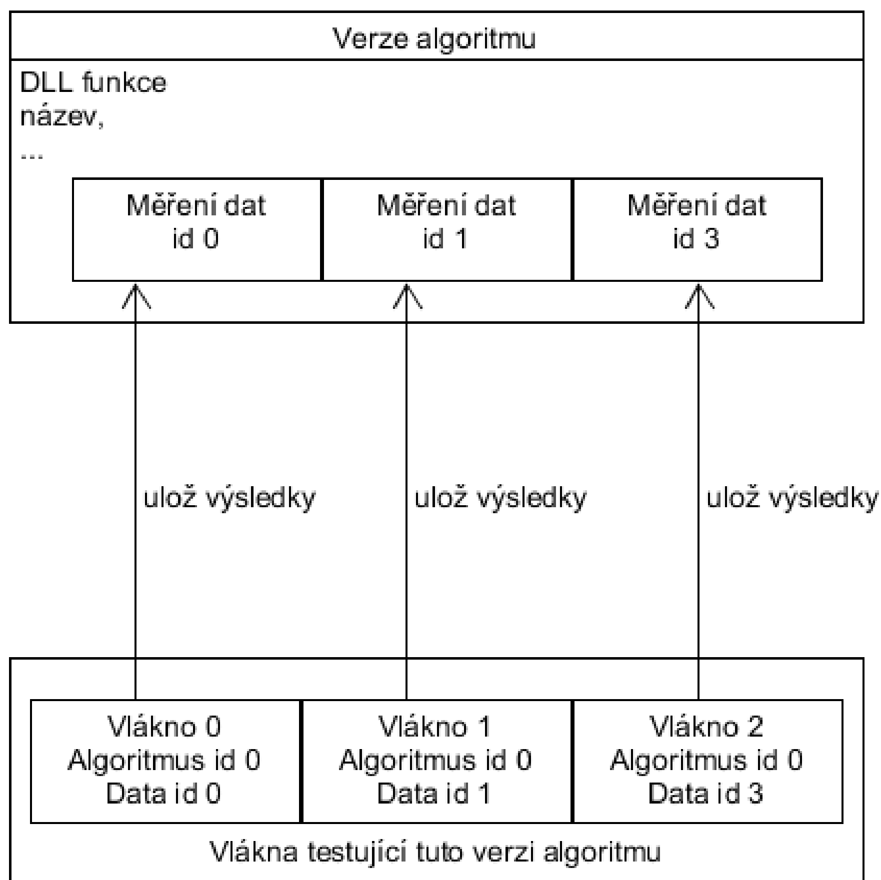
Funkce „loadSources“ je volána nad každými daty k testování, jako úvodní funkce. První parametr je relativní cesta z CWD k adresáři s metadaty vstupních dat. Třetí parametr je počet prvků v poli, kterým je druhý parametr. Druhý parametr je pole řetězců, které se nacházi jako jednotlivé zdroje v metadatech. Např. cesty k souborům s daty (fotografie). Čtvrtý parametr je lokální reference určená ke čtení, kterou si uživatel nastavil ve funkci init. Pátý parametr je volný ukazatel na ukazatel, kde si uživatel opět může alokovat paměť vzhledem k aktuálnímu zdroji. Zde by měl alokovat pamět a ukládat data, která se váží k těmto vstupním datům. Návratová hodnota se chová jako u funkce „init“.

Funkce „releaseSources“ je volána na konci každého testování jednoho vstupního data. Oba dva parametry jsou stejné jako ve funkci „loadSources“, Uživatel by měl uvolnit veškerou paměť, kterou v rámci tohoto zdroje na „*sourceReference“ alokoval. Návratová hodnota se chová jako u funkce „init“.

Poslední dvě funkce „detectObstacle“ a „categorizeObstacle“ mají totožné parametry, jsou jimi zdroje (nepřímo přes ukazatele), které byly alokovány ve funkcích „init“ a „loadSources“. Parametry mají stejné názvy, jako ve funkcích, které je alokovaly. Z těchto parametrů by uživatel měl být schopný zkonstruovat veškeré informace, k detekci a kategorizaci překážky. První volaná funkce je „detectObstacle“ a druhá „categorizeObstacle“. Tyto dvě funkce jsou typicky závislé a tak by si uživatel měl informaci o detekované překážce uložit do paměti zdroje, kterou si v „loadSources“ alokoval. Návratové hodnoty reprezentují klasifikaci. Tedy u „detectObstacle“ je to počet překážek a u „categorizeObstacle“ je to kategorie překážky. Návratová hodnota „-1“ je rezervována pro chybu a bude tak s ní v modulu nakládáno.

Pokud uživatel tyto pravidla a rozhraní dodrží, je jeho kód bezpečný pro přístup z více vláken. Na straně testeru je jediná kritická sekce. A to výběr páru neotestovaná verze - neotestovaná data. V postatě se jedná o dvě primitiva - indexy do kolekcí. Do této metody přistupují jednotlivá vlákna a interní indexaci mění, to dělá tuto sekci kritickou. Celá metoda je řádně opatřena jediným blokujícím synchronizačním primitivem (mutex). Moholo by se ještě zdát, že zápis měření je kritickou sekcí, ale není, protože každá instance algoritmu obsahuje kolekci struktur s měřením, indexovanou podle pořadí zdroje a naměřené hodnoty se zapisují do relevantní struktury podle indexu aktuální testované entity. Počet prvků kolekci měření je stejný, jako počet prvků kolekce testovaných dat. Každá tato instance struktury měření obsahuje pouze data dané dvojice. Každé takovéto měření je modifikováno standardním postupem, protože každá dvojice je testována pouze jednou (jedním vláknem). Není to tedy kritická sekce, protože modifikaci dat provádí vlákna na odlišných místech. Stav jednoduché situace je znázorněn v obrázku 4.4. Vlákna zajisté zároveň sahají do instance algoritmu, ale v momentu testování pouze pro čtení, což není považováno za kritickou sekci. Jindy, například při inicializaci, nebo uzavírání algoritmů běží pouze hlavní vlákno a tedy kritickými sekcemi se zabývat nemusí.

Měření času probíhá pomocí standardní knihovny „chrono“, která nabízí jednotku mirkosekund. V analýze je řečeno, že měření času je v jednotkách milisekund, ale protože mikrosekundy jsou přesnější, tak tento fakt nijak požadavek neomezuje. Aplikace ukládá naměřený čas v mikrosekundách a při zobrazení porovnání ho převádí do jednotek milisekund.



Obrázek 4.4: Jednoduchý případ se třemi vlákny, třemi vstupními daty a jedním algorimtem

4.4 Vzdálená SAGE2 aplikace

Rozhraní na vzdálené straně SAGE2 je triviální. Aby aplikace porovnala a zobrazila výsledky vyžaduje spojení s lokální aplikací přes WebSocket. Pro uskutečnění spojení, je třeba, aby na lokální straně uživatel přešel na obrazovku s výsledky, kde se zobrazuje jeho IP addresa a port pro připojení SAGE2, tím otevře WebSocket server, hostovaný na této adrese.

Na vzdálené straně pak standardním ovládáním otevře widgetové menu, kde zobrazenou IP vyplní a potvrdí. Poté se aplikace pokouší navázat spojení (otevírá WebSocket s cílovou adresou lokální aplikace). Při úspěšném navázání spojení má uživatel možnost měnit obsah zobrazovaný na SAGE2. Lokální WebSocket přijímá jakékoliv připojení, nijak připojené klienty nekontroluje. Uživatel musí data vyslat ručně tlačítkem „STREAM“. To vyšle data s hla-

vičkou (kterou SAGE2 vyžaduje) na všechna navázaná spojení.

Původně aplikace fungovala tak, že přijímala pouze jedno spojení. Prokázalo se však, že takto pak aplikace nefunguje v libovolné SAGE. Např. na jednoobrazovkové SAGE2 to fungovalo správně, na víceobrazovkové nikoliv, protože SAGE2 více obrazovek může dělit do jendotlivých zobrazovacích klientů a každý klient pak vytváří jednu instanci aplikace[50].

Touto cestou pak docházelo k žádostem o několik připojení, která byla zamítnuta. Data se pak zobrazovala jen v přehledovém oknu, jehož připojení bylo přijimuto jako první. Proto je třeba přijímat více, než jedno připojení. Také se prokázalo, že pro hladké a přímé použití je potřeba, aby SAGE2 display klienti byli ve stejné síti, jako lokální aplikace. SAGE2 vzdálená strana využívá IP adresu a port k připojení. Pokud nejsou aplikace připojeny ke stejné síti (LAN), tak je potřeba zadat jakoukoliv korektní IP adresu, která ve finále vede na počítač, s otevřenou lokální stranou, pokud takováto cesta v síti existuje. Pak bude třeba v jednotlivých uzlech sítě zařítit správná přesměrování.

4.5 Debugování

Debugování je proces, při kterém se prochází části kódu a paměti za jeho interpretace a kontrolují se vnitřní stavy a volání programu tak, aby vyhovovali zamýšlené předloze. Pro GUI aplikaci a SAGE2 vzdálenou stranu jsem použil neefektivní a triviální metodu debugování - výpisy promněných do konzole, kterou jak SAGE2, tak Electron poskytuje. Tato metoda spočívá v kontrolování hodnot v jistých částech kódu tak, že se její obsah vypisuje v konzoli při ručním testování. Zmíněný přístup je zdlouhavý, pomalý a ne příliš efektivní. Ale protože jsou aplikace malého rozsahu a pracují hlavně s GUI, je koreknost většiny kódu v podstatě vidět na obrazovce (správně se zobrazuje a reaguje na uživatelské příkazy).

C++ modul jsem efektivně debugoval v nástroji Microsoft Visual Studio [72]. Bohužel se ale vyskytla menší komplikace, kdy po nahrání modulu přes originální GUI aplikaci odmítal debugger spolupracovat. Framework Electron totiž potřebuje veškeré moduly pro Node.js překompilovat svým způsobem a to si myslím, že je příčina problému. Proto jsem vždy modul kompiloval pro Node.js, a v debuggeru spouštěl Node.js konzoli, ve které jsem modul nahrál a zavolal funkci pro testování, do které mě už debugger pustil. Pro použití v aplikaci jsem pak modul překompiloval a spoléhal na ekvivalentní funkčnost.

4.6 Vlastnosti implementací

Všechny tři aplikace (GUI, SAGE2, C++ modul) nijak stoprocentně nekontrolují koreknost vstupů od uživatele. Jedná se o prototyp a pokud je vyžadováno aby uživatel dodával implementace algoritmů v C++ a rozuměl výše zmíněným rozhraním a nárokům, tak předpokládám, že je dostatečně inteli-

gentní na to, aby například pro parametry spuštění testování pro počet vláken nevyplnil „DESET“, ale „10“.

V implementaci jsou použity multiplatformní knihovny a proto by lokální aplikace mohla být multiplatformní. Takovouto věc testovat nebudu a proto se o toto tvrzení nechci moc opírat, ale věřím, že by aplikace fungovaly na všech populárních operačních systémech s malými, nebo bez žádných změn.

Celý styl a duch implementace je „maximalizace nezávislosti“. Stačí menší změna zdrojového kódu k tomu, aby testovací část kódu byla místo přídavného modulu samostatně spustitelná. Databáze je řešená XML a CSV soubory, proto se v některých případech dá vyhnout používaní GUI aplikace, pokud by ji platforma nepodporovala. Platformy, které neumí spustit Node.js prostředí by pak také mohly testovat. Naměřená data jsou v CSV formátu a dají se zpracovávat vlastním způsobem, např. v populárních datamining aplikacích.

Rozhraní testovaných implementací algoritmů ani výhradně nevyžaduje, aby využívali OpenCV knihovny. Hůře by se udržovala nejnovější verze a navíc by nemusela být kompatibilní s uživatelskými implementacemi algoritmů. Se zdejším řešením je veškerá zodpovědnost závislostí na uživateli a proto si může zvolit které verze a jakým způsobem zkompiluje a použije.

Kapitola 5

Testování

5.1 Příprava

Software (GUI a C++ modul) byl testován na operačním systému Windows 10 64bit a SAGE2 část v SAGElabu na SAGE2 (linux) a v SAGE2 na Windows 10 64bit.

Testované algoritmy jsou implementované velice podobným způsobem jako v jejich analýze, obsahují zmíněně parametry, volání funkcí a zpracovávají fotografie z robotického vozítka.

Zisk dat z robotického vozítka se trochu oproti původní myšlence zkomplikoval. Původně bylo v plánu nastavit konfigurační soubor pro spustitelnou verzi Yuri, který ať už periodicky, nebo na stisk tlačíka na ovladači sbírá fotografie a po místní síti je posílá na druhý spustitelný Yuri konec, kde se uloží na disk. Vytvoření takovýchto konfiguračních souborů se z důvodu slabé uživatelské dokumentace Yuri nedařilo ani s asistencí personálu SAGElab provést. Proto jsem se po konzultaci s personálem rozhodl pro následující formu:

Upravili jsme existující konfigurační soubor tak, aby sbíral pouze data z kamer a to ve formě videa s frekvencí jeden snímek za sekundu. Upravená (snížené rozlišení, h264 kodek [71] - v SAGElab standardně používaný) sezbíraná videa pak ukládal na svoje pevné uložiště, odkud jsme je přes síť přesunuli na USB flash disk. Vždy jsme před vozítko dali předmět k zaznamenání a spustili Yuri s danou konfigurací. Po chvíli jsme aplikaci ukončili, soubory přesunuli do jiného adresáře, aby se nepřepisovali novými daty a celý proces sběru videa opakovali s různými předměty.

Z každého páru videa pak programem „ffmpeg“[67] byly vyjmuty první snímky ve formátu JFIF a ty poté sloužily jako vstupní data pro porovnávané implementace algoritmů.

Testované implementace algoritmů používají záznam pouze z levé kamery, protože vyžadují jen jednu fotografii ke zpracování. Metadata jsou ale připravena tak, aby poskytovala oboje fotografie.

5.2 Pokrytí funkčnosti případy užití

Následují minimalistické scénáře k dříve definovaným případům užití, podle kterých je aplikace testována. Pokud všechny scénáře proběhnou v pořádku, lze dle metody mapování případu užití na požadavky aplikace označit software za kvalitní.

- **Přidání implementace algoritmu (UC1)**
  1. Naviguj se na obrazovku s algoritmy z hlavního menu.
  2. Stiskni tlačítko pro přidání nového algoritmu.
  3. Vyplň formulář a odešli.
  4. Algoritmus je přidán a objevil se v seznamu dostupných algoritmů.
- **Přidání dvou implementací algoritmu, které iterují nad parametry (UC2)**
  1. Vytvoř nebo získej dva nezávislé soubory s XML kódem v korektním formátu (viz uživatelská příručka).
  2. Zkopíruj tyto metatada do relevantní složky (./userdata/algorithms).
  3. V aplikaci naviguj na přehled dostupných algoritmů, v případě, že tato obrazovka již je zobrazená, přejdi na ni znovu.
  4. Oba dva algoritmy se objeví v seznamu.
- **Přidání fotografie (UC3)**
  1. Naviguj se na obrazovku se vstupními daty z hlavního menu.
  2. Stiskni tlačítko pro přidání nového data.
  3. Vyplň formulář a odešli.
  4. Data jsou přidána a objeví se v seznamu dostupných dat.
- **Přidání více forografií (UC4)**
  1. Vytvoř nebo získej libovolný počet souborů s XML kódem v korektním formátu (viz uživatelská příručka).
  2. Zkopíruj tyto metatada do relevantní složky (./userdata/data).
  3. V aplikaci naviguj na přehled dostupných vstupních dat, v případě, že tato obrazovka již je zobrazená, přejdi na ni znovu.
  4. Všechna data se objeví v seznamu.
- **Odebrání algoritmu (UC5)**
  1. Naviguj na obrazovku s přehledem algorimů v hlavním menu.

2. V seznamu klikni na tlačítko pro odebrání a potvrď žádost o smazání.
3. Metadata jsou nyní smazána a algoritmus již v seznamu není.

Alternativní scénář:

1. V operačním systému ze složky pro algoritmy odstraň toužená metadata.
2. V aplikaci na obrazovce přehledu algoritmů již smazaný algoritmus není.

- **Odebrání fotografie (UC6)**

    1. Naviguj na obrazovku s přehledem vstupních dat v hlavním menu.
    2. V seznamu klikni na tlačítko pro odebrání a potvrď žádost o smazání.
    3. Metadata jsou nyní smazána a data již v seznamu nejsou.

    Nebo:

    1. V operačním systému ze složky pro vstupní data odstraň toužená metadata.
    2. V aplikaci na obrazovce přehledu dat již smazané data nejsou.

- **Spuštění testování s více vlákny (UC7)**

    1. V obrazovce pro spouštění testu vyplň mj. počet vláken, které testují na libovolný počet větší než 1.
    2. Spusť testování.

- **Spuštění testování s jedním vláknem (UC8)**

    1. V obrazovce pro spouštění testu vyplň mj. počet vláken, které testují na 1.
    2. Spusť testování.

- **Spuštění vybrané množiny algoritmů nad vybranou množinou dat (UC9)**

    1. Deaktivuj entity, které nechceš testovat v relevantní obrazovce.
    2. Přidej tag k entitám, které chceš testovat v relevantní obrazovce.
    3. V obrazovce pro spouštění testu vyplň například tag, nebo využij možnosti výběru pouze aktivních entit.
    4. Spusť testování.

- **Zobrazení obrazovky porovnávající jeden test (UC10)**
    1. Proveď alespoň jedno testování.
    2. Naviguj na obrazovku s přehledem měření.
    3. V seznamu klikni na tlačítko zobrazení porovnání v touženém řádku.
    4. Objeví se obrazovka s porovnáním měření.

Většina scénářu případů užití je triviálních a má zřejmou metodu kontroly. Totiž efekt lze vizuálně zkontrolovat v GUI.

Výběr správných algoritmů a dat pro testování se dá zkontrolovat po skončení testování - zobrazit CSV seznam měření a ujistit se, že byly testovány pouze vybrané algoritmy nebo při debugování (viz implementace). Pro kontrolu počtu vláken pro testování je vhodné zapnout libovolný systémový nástroj, který umí sledovat spuštěná vlákna. V momentu testování přibude specifikovaný počet vláken, nebo nepřibude žádné v případě testování jedním vláknem. Tyto dvě funkčnosti jsem kontrolovat vizuální revizí kódu a debugováním relevantních případů užití (viz implementace).

Funkčnost na SAGE2 prostředí jsem otestoval zobrazováním porovnání testů. Obrázek 5.1 ilustruje okno aplikace v SAGE2 prostředí.

Obrázek 5.1: Porovnání testů různě nakonfigurovaných verzí analyzovaných implementací nad daty z běžného fotoaparátu.

Některé záležitosti, které se stojí za zmínku:

- Prvotní testy s pevně nakonfigurovanými implementacemi nad daty z běžného fotoaparátu jsem prováděl totožně nastavené jak na jednom vlákně, tak na více vláknech. Naměřil jsem zajímavé hodnoty: průměrná strávená doba nad zdrojem byla skoro dvojnásobná, ale na vícevláknovém testu. Tento závěr je překvapivý a fakt dávám za vinu zkompilovaným OpenCV DLL, které byly specifikovány jako vícevláknové a sdílí jednotlivé části (atribut „/MD“ v [68]). Mohu se však mýlit. V této práci to rozebírat nebudu.

- S výsledky u více verzemi algoritmů se objevil problém, kdy tabulka porovnání přesahovala okno obrazovky. Lokální aplikace ve frameworku Electron toto elegantně řeší automatickým scrollbarem, který se v takovém případě objeví. Okno v SAGE2 aplikaci tuto vlastnost nemá a tak jsem scrollovaní musel doimplementovat ručně.

- V GUI aplikaci, pokud bylo v tabulce entit nebo přehledu měření více jak pět záznamů, tak se v tabulce špatně orientovalo. Problém vyřešilo střídavé podbarvení řádků.

- Porovnávané algoritmy určené v analýze nebyly moc efektivní. Jejich úspěšnost klasifikace pro detekování překážky se v různých testech pohybovala pod 65 procenty. Navíc byly testovány na malé množině vstupních dat (zhruba 10 obrázků). Takovýto výsledek se dal očekávat, protože specifický algoritmus pro spodní část končetin ani teoreticky není naučen detekovat například krabici, která se ve vstupních datech objevila a inutuitivní algoritmy byly zjevně nedostatečně korektní.

Všechny testy nakonec proběhly korektně. Veškerá dostupná fotodokumentace nebo testovací data použitá pro demonstraci funkčnosti jsou k nalezení na přiloženém médiu.

Kapitola 6

Navázání na tuto práci

Protože vznilá aplikace je pouhým prototypem a její opravdová užitečnost se prokáže až při reálném použití, myslím si, že se vyskytne dostatek funkčností, o které by bylo vhodné aplikaci rozšířit.

Prvním, nepochybným krokem je využít tuto aplikaci k měření sofistikovaných algoritmů. Prokáže se tím relevantnost porovnávací metodiky a funkčností. Na základě zpětné vazby po tomto využití navrhnout změny.

Nyní vidím jednotlivé kroky, které by mohly funkčnost aplikace rozšířit:

Více porovnávacích metodik Při reálném využití tohoto softwaru se ukáže, po kterých informacích z porovnání uživatel prahne, otázky, které si klade. Cílem by bylo implementovat vhodné porovnávací metodiky, aby svoji odpověď uživatel našel.

Zajištění multiplatformnosti Ujistit se, že zdrojový kód je multiplatformní a pokud ne, tak ho upravit tak, aby mohla fungovat na více platformách.

Implementace samostatného testeru Jednalo by se o odpojení od Node.js prostředí a vytvoření samostatně spustitelného testeru. Testovací přídavný modul nyní nevyužívá žádné možnosti JavaScriptu mimo vstupní a výstupní rozhraní. Modul by se tak zbavil závislosti na knihovně V8. Dal by se jednodušeji použít k testování na konečném hardwaru - např. robotické vozítko, kde by se tester zkompiloval a spouštěl.

Snížení standardu Nyní se zdrojový kód kompiluje ve standardu C++11 nebo novějším. Přitom využívá jen jediné funkčnosti a to standardní knihovny „chrono“, která měří čas v mikrosekundách, protože knihovna „libuv“ nenabízí funkcionalitu pro zjištění času od prvotního bodu v mikrosekundách. Kdyby se nalezl jiný způsob, jak takto přesný čas měřit v nižším standardu, rozhodně si myslím, že by se tím zvětšil prostor cílových strojů, které by měření mohly spouštět, protože by si s nižším standardem kompilátory cílových hardwarů poradily lépe.

Podpora více vláken nejen pro testování Nyní se v aplikaci více vlákny provádí pouze elementární testování. Urychlení celkového běhu by se dalo dosáhnout zavedením vícevláknového zpracování při čtení souborů nebo iteraci parametrů.

Přívětivější podpora iterací algoritmů Nyní aplikace nabízí pouze lineární iteraci atributů. Uživatel si sice může v inicializační funkci implementace atributy přepočítat a tím získat libovolnou hodnotu, ale sofistikovanější formou, která by umožňovala iterovat přímo uživatelskou funkcí by tuto problematiku usnadnila.

Kapitola 7

Závěr

Tato práce nejprve analyzovala robotické vozítko z inventáře SAGElab umístěné na ČVUT, poznala, jaký hardware využívá, která data a jakým způsobem je možné z něj získat. Dalším cílem analýzy bylo získat přehled o dostupných algoritmech počítačového vidění. Zmínka je zde o několika knihovnách, které by mohly být vhodné pro použití, ale na doporučení vedoucího práce bylo využito OpenCV.

Analýza ukázala, že OpenCV primárně neposkytuje implementace algoritmů počítačového vidění pro detekci překážek, ale především implementace algoritmů pro zpracování obrazu. V této práci je použita jedna, která detekuje z fotografie spodní část těla. Další dva algoritmy jsou intuitivně zkonstruované z dostupných implementací různých funkcí. Celkově tak byly určeny tři algoritmy, jejichž implementace byly později porovnány v rámci testování aplikace, která s touto prací vznikla.

U těchto tří implementací algoritmů došlo k analýze programové rozhraní a na základě této analýzy vznikl návrh obecnější metody porovnávání implementací algoritmů počítačového vidění pro detekci překážek. Práce se pak pomalu přesouvala do praktické části, kdy na základě porovnávací metodiky byly určeny požadavky na aplikaci a tím vlastně vymezena její funkčnost a konkrétnější účel. Poté byl určen způsob zaručení kvality této aplikace a práce se přesunula k tvoření architektury softwaru. Zde se určilo že aplikace bude dělena na dvě podaplikace, jednu, která je určená pro prostředí SAGE2 a jednu, která je určená pro plné využívání v běžném operačním systému.

V implementační části práce pak popisuje zvolené přistupy reprezentace navržené architektury a občas polemizuje nad zvoleným řešením. Ke konci práce pak ověřuje kvalitu dříve zmíňěným způsobem a navrhuje možnosti navázání na tuto práci.

Tato práce vytvořila funkční aplikaci, která slouží k porovnávání implementací algoritmů počítačového vidění se zaměřením na detekci překážek.

Příloha **A**

Seznam použitých zkratek

API Application program interface

HTTP Hypertext transfer protocol

HTML Hypertext markup language

XML Extensible Markup Language

CSS Cascading style sheets

OCS Open sound control

RTP Real-time transfer protocol

JPEG Joint Photographic Experts Group

JFIF JPEG File interchange format

DLL Dynamic link library

AJAX Asynchronous JavaScript and XML

AABB Axis aligned bounding box

GUI Graphical user interface

UI User interface

URL Unified resource location

CWD Current working directory

FIT Fakulta informačních technologií

ČVUT České vysoké učení technické

LAN Local area network

FPS Frames per second

Příloha B

Obsah přiloženého média

`přiložené médium` . hlavní adresář, obsahuje soubory nutné ke kompilaci práce

- `botdata` adresář s jednotlivímy daty z robotického vozítka a skriptem pro vyjmutí prvního snímku
- `cvcompare` .... adresář se zdrojovými soubory pro GUI aplikaci
- `cvcompare_results` ...... adresář se zdrojovými soubory pro SAGE2 aplikaci
- `CVdemo` adresář se zdrojovými soubory, testovacími daty a skripty pro analýzu rozhraní OpenCV
- `diagramy` .... adresář se zdrojovými soubory použitých diagramů
- `fotodokumentace` fotografie pořízené jako fotodokumentace této práce
- `licences` .... adresář s licencemi distribuovaného softwaru
- `yuri_conf` .. adresář s konfiguračními soubory pro software Yuri-light využité v této práci
- `BP.tex` .... zdrojový kód práce
- `BP.pdf` .... elektronická verze práce ve formátu PDF

Příloha C

Uživatelská příručka

C.1 Uživatelská příručka pro GUI lokální aplikaci

Při spuštění aplikace se objeví úvodní obrazovka (obrazovka pro spouštění testů).

Aplikace využívá známý styl webové navigace a tak je používání aplikace velice intuitivní. Na veškeré obrazovky se dá dostat skrze hypertextovou navigaci.

Obrazovky samy zobrazují dostatečné množství instrukcí pro uživatele a tak se zde jenom zmíním o důležitých věcech, které nejsou přímo zmíněny v aplikaci.

C.1.1 Dodávání implementací algoritmů

Uživatel má možnost dodávat vlastní implementace formou DLL, jehož cestu musí správně specifikovat v metadatech implementace.

To znamená, že uživatel musí sám zkompilovat a dodat DLL, které dodržuje jistá omezení a rozhraní zmíněné dříve v této práci. Volba kompilátoru zavisí na operačním systému určeném pro testování a na volbě uživatele. Je ale nutné, aby uživatel dodržel omezení, které si tato aplikace klade.

DLL musí vyexportovat minimálně 6 metod, s přesnými názvy a parametry (viz implementace). Například v kompilátoru CL pro kompilaci C++ se to dá zajistit hlavčkou před definicí nebo deklarací funkce (obrázek C.1).

C++ verze kompilátoru CL[73] totiž obměuje názvy funkcí a tedy hlavička „extern "C"“ říká, že se název má kompilovat jako kód v C a tam se název neobměňuje. Tím se zajistí stálost názvu funkce při kompilaci. [69] Druhá

Kód C.1: Příklad deklarace DLL funkce

```
extern "C" __declspec(dllexport) void myFunction();
```

hlavička pak určuje, že funkce má být dostupná k volání aplikací, která tuto DLL nalinkovala [70].

Kód se pak musí zkompilovat jako DLL určené k dynamickému nahrávání za běhu aplikace v bitové verzi stejné, jako je testovací aplikace a Node.js.

Pokud je DLL zkompilováno tak, že potřebuje nahrát jiná DLL při nahrání samo sebe, tyto další DLL musí být umístěna v CWD, tedy v kořenovém adresáři této aplikace.

C.1.2 Streamování dat na SAGE2 vzdálenou aplikace

Tato speciální funkčnost je možná pouze z obrazovky s přehledem dostupných výsledků měření. Obrazovka ukazuje LAN IP adresu, na které se otevřelo připojení očekávající spojení s SAGE2 vzdálenou aplikací.

Po připojení vzdálené aplikace se u každého záznamu objeví možnost provést operaci „STREAM“, která dané výsledky pošle do vzdálené aplikace k porovnání. Při opětovném stisknutí libovolného tlačítka „STREAM“ se data ve vzdálené aplikaci přepíší daty novými.

C.1.3 Formáty metadat, CSV souborů s výsledky a logů

Log je textový soubor, který obsahuje na každém řádku jednu zprávu. Zprávy jsou ve formátu čas, typ zprávy a obsah zprávy.

CSV soubor s výsledky vždy obsahuje hlavičku s pojmenovanými sloupci, kterých je vždy stejný počet. Každý řádek pak reprezentuje měření jedné verze algoritmu nad jedním datem. Měřená verze implementace algoritmu vždy obsahuje ve svém názvu mj. i hodnoty jednotlivých parametrů oddělené středníkem, pokud originální implementace nad parametry iterovala. Obsah jednodlivých buněk je vždy uveden ve dvojitých uvozovkách a jednotlivé buňky jsou vždy odděleny středníkem.

Aplikace pracuje celkem se třemi typy metadat. Všechny tři typy jsou XML kód. Název souboru musí končit koncovkou „meta“ jinak aplikace nebude se soubory spolupracovat. Každé metadata reprezentuje jednu entitu a ve svém kódu má hlavní tag „metadata“. Název souboru bez koncovky „meta“ reprezentuje ID entity.

Do hlavního tagu se pak vnořují další možné tagy:

- **Konfigurační soubor testu**

  useLogging Hodnota 1 reprezentuje, že se má logovat do souporu (s názvem id této entity a koncovkou „log“) v adresáři resultsDir. Hodnota 0 zakazuje logování do souboru.

  useStdLogging Hodnota 1 loguje do standardního výstupu. Hodnota 0 po nakonfigurování tohoto souboru logování do standardního výstupu vypne.

algorithmsDir Relativní (z „CWD/pages“) a nebo absolutní cesta do adresáře s metadaty pro implementace algoritmů.

dataDir Relativní (z „CWD/pages“) a nebo absolutní cesta do adresáře s metadaty pro vstupní data.

resultsDir Relativní (z „CWD/pages“) a nebo absolutní cesta do adresáře pro ukládání logu a výsledku měření.

useActiveAlgorithms Hodnota 1 reprezentuje, že se pro testování vyberou pouze implementace algoritmů, které jsou nastaveny jako aktivní. Hodnota 0 tento přívlastek ignoruje.

useActiveData Hodnota 1 reprezentuje, že se pro testování vyberou pouze data, která jsou nastavena jako aktivní. Hodnota 0 tento přívlastek ignoruje.

algorithmsTags Textové řetězce oddělené čárkou. Každý takový texový řetězec reprezentuje tag, kterým musí být implementace algorimu označena, aby byla vybrána k testování. Stačí, aby vlastnila jeden z tagů a bude vybrána. Pokud se tento XML prvek vynechá, tak se z výběru nevyřazují žádné implementace na základě přidělených tagů.

dataTags Textové řetězce oddělené čárkou. Každý takový texový řetězec reprezentuje tag, kterým musí být data označena, aby byla vybrána k testování. Stačí, aby vlastnilo jeden z tagů a bude vybráno. Pokud se tento XML prvek vynechá, tak se z výběru nevyřazují žádné data na základě přidělených tagů.

numOfThreads Počet testovacích vláken, které mají implementace měřit.

- **Vstupní data**

name Název vstupních dat. Jedná se jen o pojmenování pro lehčí identifikaci uživatelem

sources V tomto XML tagu je vnořen libovolný počet XML tagů „source“, kde každý obsahuje textový řetězec zdroje, který bude předán implementaci skrze definované rozhraní. Vždy by měl existovat alespoň jeden zdroj.

active 1/0, označuje, zda-li jsou data aktivní a nebo neaktivní.

tags Obdobně jako u „sources“ je v tomto XML tagu vnořen libovolný počet XML tagů „tag“ obsahující textový tag přidělený tomuto datu. Tentokrát nemusí metadata obsahovat žádný tag.

obstacles Celé číslo, obsahuje počet překážek v tomto datu. Číslo by nikdy nemělo být -1, protože to je vyhrazeno pro vnitřní reprezentaci chyby.

category Celé číslo, obsahuje kategorii překážky v tomto datu. Číslo by nikdy nemělo být -1, protože to je vyhrazeno pro vnitřní reprezentaci chyby.

- **Implementace algoritmu**

name Název implementace. Jedná se jen o pojmenování pro lehčí identifikaci uživatelem

source Relativní cesta ke zdrojovému DLL souboru z adresáře pro metadata s implementacemi algoritmů uvedeném v konfiguračních metadatech testování. Pokud je zdrojové DLL ve stejném adresáři, jako tato metadata, jenom název souboru s DLL je uveden.

active 1/0, označuje, zda-li je implementace algoritmu aktivní a nebo neaktivní.

tags V tomto XML tagu vnořen libovolný počet XML tagů „tag" obsahující textový tag přidělený této implemetaci. Metadata nemusí obsahovat žádný tag.

attributes Tento XML tag obsahuje libovolný počet vnořených XML tagů „attribute". Každý takovýto vnořený XML tag musí obsahovat přesně čtyři XML tagy: „type", „initialValue", „endingValue" a „jump". Každý „atribute" zajistí vygenerování dalších verzí tohoto algoritmu k již předchozím vygenerovaným obohacený o nové hodnoty tohoto atributu z probíhající iterace. Jedná se o lineární iteraci ze začínající hodnoty („initialValue") s přičítáním hodnoty „jump" (může být záporná) dokud nepřesáhne (nepodleze) „endingValue". Smysluplnost iniciální hodnot atributů závisí na uživateli. Aplikace podporuje iteraci z obou směrů: se zápornou hodnotou skoku je možné, aby startovací hodnota byla větší, než konečná. Jinak, ale pokud atributy nebudou smysluplné, aplikace nemusí fungovat. Například pro skokovou hodnotu 0 se aplikace zasekne v nekonečné smyčce. Generování podporuje 6 typů atributů:

i32 reprezentuje C typ „int"

i64 reprezentuje C typ „long long"

u32 reprezentuje C typ „unsigned int"

u64 reprezentuje C typ „unsigned long long"

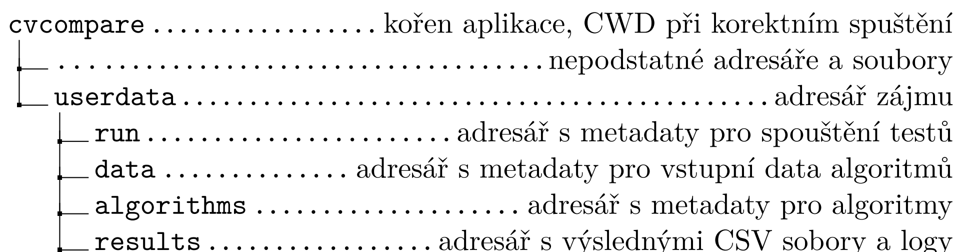
f32 reprezentuje C typ „float"

f64 reprezentuje C typ „double"

C.1.4 Adresářová struktura

Pokud by uživatel používal funkčnosti pouze skrze grafické rozhraní, není třeba ani adresářovou strukturu znát. Pokud by však chtěl metadata a jejich při-

družené soubory držet pohromadě, či importovat nebo exportovat entity nebo procházet logy testování, měl by tuto strukturu znát:



Obrázek C.1: Adresářová struktura aplikace

V adresáři „run“ lze nalézt metadata s konfiguracemi pro spuštění testování. Ty nejsou v aplikaci zobrazitelné a uživatel je nemůže nijak použít. Tato data se vytváří při spuštění testování a ihned se používají.

Zbylé tři adresáře reprezentují tabulky a metadata v nich generují záznamy v tabulkách na relevantních obrazovkách.

Pokud by uživatel chtěl v metadata držet pohromadě se zdroji, měl by je umístit do odpovídajících složek.

C.2 Uživatelská příručka pro SAGE2 aplikaci

SAGE2 aplikace nabízí jednoduché rozhraní.

- Po spuštění se zobrazí obrazovka s instrukcemi pro uživatele. V SAGE2 prostředí může pravým kliknutím uživatel otevřít formulářové pole pro vyplnění adresy vypsané v lokální aplikaci na obrazovce přehledu měření. Po potvrzení formulářového políčka se v případě úspěšného připojení změní obsah obrazovek jak v lokální, tak v SAGE2 obrazovce.

- Uživatel má ještě možnost tlačíkty „+“ a „-“ na numerické klávesnici měnit velikost textu tak, aby mu to pro porovnávání vyhovovalo. Šipka nahoru/dolu slouží pro procházení delších tabulek, které se nevejdou do jednoho okna. Tyto ovládací prvky fungují jen když je zvolený standardní kurzor SAGE, nikoliv ten, který je určený k přesouvání a zvětšování.

- Obsah vzdálené SAGE2 obrazovky se ovládá přes lokální aplikaci.

Příloha D

Instalační příručka

D.1 Instalační příručka pro GUI aplikaci

Aplikace vyžaduje nainstalované JavaScriptové prostředí Node.js verze 4.4.3 LTS. Aplikace byla testována pro Windows 10 64bit verzi, kde svoji funkčnost zaručuje. Nicméně protože využívá multiplatformních knihoven, je pravděpodobné že bude fungovat i na jiných operačních systémech. Aplikace je vyvíjena pro tuto verzi Node.js a kompatibilitu s jinými verzemi nezaručuje. Je naprosto možné a i pravděpodobné, že aplikace bude fungovat i na jiných verzích. Dále se předpokládá, že Node.js přidružený software npm je dostupný v jakémkoliv CWD, Python verze 2.7 je také dostupný v jakémkoliv CWD a uživatel je připojen k internetu.

Adresář se zdrojovým a kódem aplikace je v přiloženém médiu.

1. Překopíruj zdrojový adresář „cvcompare“ do libovolného adresáře na lokálním systému
2. Otevři příkazovou řádku a v ní přejdi do takto zkopírovaného adresáře.
3. Nainstaluj, nebo se ujisti, že máš nainstalovaný nástroj „node-gyp“ příkazem „npm install -g node-gyp“
4. Nainstaluj aplikaci příkazem „npm install“.
5. Spusť skript „rebuild.cmd“
6. Spusť aplikaci pomocí příkazu „npm start“.

D.2 Instalační příručka pro SAGE2 aplikaci

Instalace vyžaduje funkční SAGE2 server a oprávnění k přístupu na server. Adresář se zdrojovým kódem aplikace je v přiloženém médiu.

1. Překopíruj zdrojový adresář „cvcompare_results“ do relativní cesty (od kořenového adresáře SAGE2 serveru): „public/uploads/apps/“
2. Zapni a nebo restartuj SAGE2 server.
3. Aplikace je dostupná přes standardní webové rozhraní pod názvem „cvcompare_results“

Literatura

[1] RICHTR, R. Emailová komunikace o úpravě textu této práce. 12. května 2016.

[2] ALDEBARAN. Aldebaran Robotics | Humanoid robotics & programmable robots. *Aldebaran.com* [online]. [cit. 2016-05-11]. Dostupné z: `https://www.aldebaran.com/en/cool-robots/nao`

[3] BOSTON DYNAMICS. Boston Dynamics: Dedicated to the Science and Art of How Things Move. *Bostondynamics.com* [online]. ©2016 [cit. 2016-05-11]. Dostupné z: `http://www.bostondynamics.com/index.html`

[4] GRIFFITHS, Sarah. From FaceTime (sort of) to smart cookers: Postcards reveal how 19th century artists accurately predicted the 21st Century (if not the changes in fashion). In: Associated Newspapers Ltd. *Mail online* [online]. June 23, 2014, 17:27 GMT [cit. 2016-05-11]. Dostupné z: `http://www.dailymail.co.uk/sciencetech/article-2665946/From-FaceTime-smart-cookers-Postcards-reveal-19th-century-artists-predicted-world-look-like-today.html`

[5] *Chappie* [film]. Directed by Neill BLOMKAMP. USA: 2015

[6] SCOTT, Jeremy. *The Ables*. Clovercroft Publishing, 2015. ISBN 978-1940262659.

[7] POINT GREY RESEARCH, INC. Flea3 8.8 MP Color USB3 Vision Camera (Sony IMX121) Point Grey USB 3.0, Gigabit Ethernet and FireWire Machine Vision Cameras. *Ptgrey.com* [online]. [cit. 2016-05-11]. Dostupné z: `https://www.ptgrey.com/flea3-88-mp-color-usb3-vision-sony-imx121-camera`

[8] MELNÍKOV, J. Emailová komunikace o specifikaci robotického vozítka. Březen 2016.

[9] WOODFORD, Ch. 3D TV. In: *Explainthatstuff* [online]. 2010 [cit. 2016-05-16]. Dostupné z: `http://www.explainthatstuff.com/how-3d-tv-works.html`

[10] HALÁK, J. Emailová komunikace o specifikaci robotického vozítka. 29. březnen 2016.

[11] INSTITUT INTERMÉDIÍ. Root [projects.iim.cz]. *Projects.iim.cz* [online]. [cit. 2016-05-11]. Dostupné z: `http://projects.iim.cz/root`

[12] INSTITUT INTERMÉDIÍ. Yuri: compiling [projects.iim.cz]. *Projects.iim.cz* [online]. [cit. 2016-05-11]. Dostupné z: `http://projects.iim.cz/yuri:compiling`

[13] INSTITUT INTERMÉDIÍ. Libyutri-light [zdrojové kódy]. [cit. 11. května 2016]. Dostupné z: `http://projects.iim.cz/yuri:compiling`

[14] INSTITUT INTERMÉDIÍ. Yuri: modules [projects.iim.cz]. *Projects.iim.cz* [online]. [cit. 2016-05-11]. Dostupné z: `http://projects.iim.cz/yuri:modules`

[15] INSTITUT INTERMÉDIÍ. Yuri: usage [projects.iim.cz]. *Projects.iim.cz* [online]. [cit. 2016-05-11]. Dostupné z: `http://projects.iim.cz/yuri:usage`

[16] ARDUINO. Arduino - Home. *Arduino.com* [online]. [cit. 2016-05-11]. Dostupné z: `https://www.arduino.cc/`

[17] FAISTAVER, P. Emailová a osobní komunikace o robotickém vozítku. SAGElab, Březen 2016.

[18] INSTITUT INTERMÉDIÍ. Yuri: modules [projects.iim.cz]. *Projects.iim.cz* [online]. [cit. 2016-05-11]. Dostupné z: `http://projects.iim.cz/yuri:modules`

[19] SCHULZRINNE, H., COLUMBIA UNIVERSITY, CASNER, S. a spol. *RTP: A Transport Protocol for Real-Time Applications.* RFC 3550, Network Working Group. Červenec 2003. Dostupné z: `https://tools.ietf.org/html/rfc3550`

[20] WRIGHT, Matt. The Open Sound Control 1.0 Specification. *Opensoundcontrol.org* [online]. March 26, 2002 [cit. 2016-05-11]. Dostupné z: `http://opensoundcontrol.org/spec-1_0`

[21] FIELDING, R., UC IRVINE, GETTYS, J. a spol. *Hypertext Transfer Protocol – HTTP/1.1.* RFC 3550, Network Working Group. Červen 1999. Dostupné z: `https://www.ietf.org/rfc/rfc2616.txt`

[22] HOLUB, J. *Image Compression* [přednáška]. 2016. In: Edux.fit.cvut.cz [cit. 2016-05-11]. Dostupé z: https://edux.fit.cvut.cz/courses/MI-KOD/_media/lectures/13/mi-kod-13-image.pdf

[23] SAGELAB. rtp_receiver_sage.xml [zdrojový kód]. Dostupné na přiloženém médiu.

[24] CHLUDIL, J. Diskuze o způsobu ovládání vozítka. SAGElab Praha, Duben 2016.

[25] INTERNATIONAL TELECOMMUNICATION UNION. *Information technology - Digital compression and coding of continuous-tone still images - Requitements and guidelines.* CCITT T.81, 1992. ISO/IEC 10918-1:1994

[26] CPLUSPLUS.COM. A Brief Description - C++ Information. *Cplusplus.com* [online]. ©2000-2016 [cit. 2016-05-11]. Dostupné z: http://www.cplusplus.com/info/description/

[27] CPLUSPLUS.COM. History of C++ - C++ Information. *Cplusplus.com* [online]. ©2000-2016 [cit. 2016-05-11]. Dostupné z: http://www.cplusplus.com/info/history/

[28] W3C. HTML & CSS - W3C. *W3.org* [online].©2016 [cit. 2016-05-11]. Dostupné z: https://www.w3.org/standards/webdesign/htmlcss

[29] REFSNES DATA. Introduction to HTML. *W3schools.com* [online]. ©1999-2016 [cit. 2016-05-11]. Dostupné z: http://www.w3schools.com/html/html_intro.asp

[30] REFSNES DATA. HTML Reference. *W3schools.com* [online]. ©1999-2016 [cit. 2016-05-11]. Dostupné z: http://www.w3schools.com/tags/default.asp

[31] REFSNES DATA. CSS How to. *W3schools.com* [online]. ©1999-2016 [cit. 2016-05-11]. Dostupné z: http://www.w3schools.com/css/css_howto.asp

[32] REFSNES DATA. CSS Reference. *W3schools.com* [online]. ©1999-2016 [cit. 2016-05-11]. Dostupné z: http://www.w3schools.com/cssref/default.asp

[33] [MYERS, Bob a spol]. A re-introduction to JavaScript (JS tutorial). *Developer.mozilla.org* [online]. Mar 27, 2016, 9:05:10 PM [cit. 2016-05-11]. Dostupné z: https://developer.mozilla.org/en-US/docs/Web/JavaScript/A_re-introduction_to_JavaScript

[34] REFSNES DATA. JavaScript and HTML DOM Reference. *W3schools.com* [online]. ©1999-2016 [cit. 2016-05-11]. Dostupné z: `http://www.w3schools.com/jsref/default.asp`

[35] Libuv/libuv: Cross-platform asynchronous I/O. *Github.com* [online]. [cit. 2016-05-11]. Dostupné z: `https://github.com/libuv/libuv`

[36] [THORSTEN, Lorenz]. Documenting types and methods of libuv, mostly by reading 'uv.h'. *Github.com* [online]. [cit. 2016-05-11]. Dostupné z: `https://github.com/thlorenz/libuv-dox`

[37] MARATHE, Nikhil. An Introduction to libuv. *Nikhilm.github.io* [online]. 2015 [cit. 2016-05-11]. Dostupné z: `https://nikhilm.github.io/uvbook/`

[38] Welcome to the libuv API documentation — libuv API documentation. *Docs.libuv.org/* [online]. [cit. 2016-05-11]. Dostupné z: `http://docs.libuv.org/en/v1.x/`

[39] KALICINSKI, Marcin. RAPIDXML Manual. *Rapidxml.sourceforge.net* [online]. ©2006,2009 [2006-2009] [cit. 2016-05-11]. Dostupné z: `http://rapidxml.sourceforge.net/manual.html`

[40] GOOGLE DEVELOPERS. Chrome V8 | Google Developers. *Developers.google.com.* [cit. 2016-05-11]. Dostupné z: `https://developers.google.com/v8/`

[41] NODE.JS FOUNDATION. Node.js. *Nodejs.org* [online]. ©2016 [cit. 2016-05-11]. Dostupné z: `https://nodejs.org/en/`

[42] NODE.JS FOUNDATION. Node.js v6.1.0 Manual & Documentation. *Nodejs.org* [online]. ©2016 [cit. 2016-05-11]. Dostupné z: `https://nodejs.org/api/`

[43] NPM, INC. Npm. *Npmjs.com* [online]. [cit. 2016-05-11]. Dostupné z: `https://www.npmjs.com/`

[44] The Chromium Projects. *Chromium.org* [online]. [cit. 2016-05-11]. Dostupné z: `https://www.chromium.org/`

[45] GITHUB. Electron. *Electron.atom.io* [online]. [cit. 2016-05-11]. Dostupné z: `http://electron.atom.io/`

[46] Handlebars.js: Minimal Templating on Steroids. *Handlebarsjs.com* [online]. [cit. 2016-05-11]. Dostupné z: `http://handlebarsjs.com/`

[47] JQUERY FOUNDATION. JQuery. *jquery.com* [online]. ©2016 [cit. 2016-05-11]. Dostupné z: `https://jquery.com/`

[48] SAGE2. Introduction – SAGE2. *Sage2.sagecommons.org* [online]. ©2016 [cit. 2016-05-11]. Dostupné z: http://sage2.sagecommons.org/project/introduction/

[49] SAGE2. *SAGE2 User documentation.* [online]. [cit. 2016-05-11]. Dostupné z: http://sage2.sagecommons.org/download/1312/

[50] SAGE2. *SAGE2 Developer Documentation.* [online]. [cit. 2016-05-11]. Dostupné z: http://sage2.sagecommons.org/download/1312/

[51] CESNET, Z. S. P. O. SAGE2 Administrator Controls. *Sage2.cesnet.cz* [online]. [cit. 2016-05-10]. Dostupné z: http://sage2.cesnet.cz:10180/admin/index.html

[52] FETTE I., GOOGLE INC., MELNIKOV, A. a spol. *The WebSocket Protocol.* RFC 6455, Internet Engineering Task Force (IETF). Prosinec 2011. Dostupné z: https://tools.ietf.org/html/rfc6455

[53] SHAFRANOVICH, Y., SOLIDMATRIX TECHNOLOGIES, INC. *Common Format and MIME Type for Comma-Separated Values (CSV) Files.* RFC 4180, Network Working Group. Říjen 2005. Dostupné z: https://tools.ietf.org/html/rfc4180

[54] REFSNES DATA. XML Tutorial. *W3schools.com* [online]. ©1999-2016 [cit. 2016-05-11]. Dostupné z: http://www.w3schools.com/xml/

[55] REFSNES DATA. XML Schema Tutorial. *W3schools.com* [online]. ©1999-2016 [cit. 2016-05-11]. Dostupné z: http://www.w3schools.com/xml/schema_intro.asp

[56] OpenCV API Reference — OpenCV 2.4.13.0 documentation. *Docs.opencv.org* [online]. ©2011-2014 [cit. 2016-05-11]. Dostupné z: http://docs.opencv.org/2.4/modules/refman.html

[57] CANNY, J. *A Computational Approach to Edge Detection.* IEEE, 1986. DOI 10.1109/TPAMI.1986.4767851

[58] SUZUKI, S., ABE, K. *Topological Structural Analysis of Digitized Binary Images by Border Following.* CVGIP 30, 1985. DOI 10.1016/0734-189X(85)90016-7

[59] Feature Detection - OpenCV 2.4.13.0 documentation. *Docs.opencv.org* [online]. ©2011-2014 [cit. 2016-05-11]. Dostupné z: http://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=canny#canny

[60] Structural Analysis and Shape Descriptors — OpenCV 2.4.13.0 documentation. *Docs.opencv.org* [online]. ©2011-2014 [cit. 2016-05-11]. Dostupné z: http://docs.opencv.org/2.4/modules/imgproc/doc/structural_analysis_and_shape_descriptors.html?highlight=findcontours#findcontours

[61] Cascade Classification - OpenCV 2.4.13.0 documentation. *Docs.opencv.org* [online]. ©2011-2014 [cit. 2016-05-11]. Dostupné z: http://docs.opencv.org/2.4/modules/objdetect/doc/cascade_classification.html?highlight=detectmultiscale

[62] OpenCV: Cascade Classifier Training *Docs.opencv.org* [online]. [cit. 2016-05-11]. Dostupné z: http://docs.opencv.org/3.1.0/dc/d88/tutorial_traincascade.html#gsc.tab=0

[63] LIENHART, R., KURANOV, A., PISAREVSKY, V. *Empirical Analysis of Detection Cascades of Boosted Classifiers for Rapid Object Detection.* Microprocessor Research Lab, Intel Labs, 2002 [online]. [cit. 2016-05-11]. Dostupné z: http://www.multimedia-computing.de/mediawiki//images/5/52/MRL-TR-May02-revised-Dec02.pdf

[64] VIOLA P., JONES M. *Rapid Object Detection using a Boosted Cascade of Simple Features.* CVPR, 2001. DOI 10.1109/CVPR.2001.990517.

[65] Feature Detection — OpenCV 2.4.13.0 documentation. *Docs.opencv.org* [online]. ©2011-2014 [cit. 2016-05-11]. Dostupné z: http://docs.opencv.org/2.4/modules/imgproc/doc/feature_detection.html?highlight=houghlinesp#cv2.HoughLinesP

[66] MATAS, J., GALAMBOS, C., KITTLER, J. *Robust Detection of Lines Using the Progressive Probabilistic Hough Transform.* Academic Press, 2000.

[67] SCHWARZ, Kyle. FFmpeg git-c8c14d0 [software]. [cit. 11. května 2016]. Dostupné z: https://ffmpeg.zeranoe.com/builds/

[68] MICROSOFT. /MD, /MT, /LD (Use Run-Time Library). *Msdn.microsoft.com* [online]. ©2016 [cit. 2016-05-11]. Dostupné z: https://msdn.microsoft.com/en-us/library/2kzt1wy3.aspx

[69] MICROSOFT. Using extern to Specify Linkage. *Msdn.microsoft.com* [online]. ©2016 [cit. 2016-05-11]. Dostupné z: https://msdn.microsoft.com/en-us/library/0603949d.aspx

[70] MICROSOFT. Exporting from a DLL Using __declspec(dllexport). *Msdn.microsoft.com* [online]. ©2016 [cit. 2016-05-11]. Dostupné z: https://msdn.microsoft.com/en-us/library/a90k134d.aspx

[71] WANG, Y.-K., EVEN, R., HUAWEI TECHNOLOGIES a spol. *RTP Payload Format for H.264 Video*. RFC 6184, Internet Engineering Task Force (IETF). Květen 2011. Dostupné z: https://tools.ietf.org/html/rfc6184

[72] MICROSOFT. Visual Studio 2015 [software]. [cit. 15. května 2016]. Dostupné z: https://www.visualstudio.com

[73] MICROSOFT. CL [software]. [cit. 15. května 2016]. In: Visual Studio 2015. Dostupné z: https://www.visualstudio.com

[74] VLFeat [software]. [cit. 15. května 2016]. Dostupné z: http://www.vlfeat.org/

[75] . ITSEEZ. OpenCV [software]. [cit. 15. května 2016]. Dostupné z: http://opencv.org/downloads.html

[76] VXL [software]. [cit. 15. května 2016]. Dostupné z: http://vxl.sourceforge.net/

[77] Point Cloud Library [software]. [cit. 15. května 2016]. Dostupné z: http://pointclouds.org/

[78] SOUTHERN ILLINOIS UNIVERSITY EDWARDSVILLE. CVIPtools [software]. [cit. 15. května 2016]. Dostupné z: http://cviptools.ece.siue.edu/

[79] ABELES, P. BoofCV [software]. [cit. 15. května 2016]. Dostupné z: http://boofcv.org/

[80] AFORGE.NET. AForge.NET [software]. [cit. 15. května 2016]. Dostupné z: http://www.aforgenet.com/

[81] SIGHT MACHINE, INC. SimpleCV [software]. [cit. 15. května 2016]. Dostupné z: http://simplecv.org/