Západočeská univerzita v Plzni
Fakulta aplikovaných věd
Katedra informatiky a výpočetní techniky

Diplomová práce

Pokročilé algoritmy počítačového vidění pro robotický fotbal

Plzeň 2013

Robert Eckstein

Prohlášení

Prohlašuji, že jsem diplomovou práci vypracoval samostatně a výhradně s použitím citovaných pramenů.

V Plzni dne 16. května 2013

Robert Eckstein

Ochranné známky

NVIDIA a CUDA jsou ochranné známky společnosti NVIDIA Corporation. Ochranné známky Microsoft a Windows patří firmě Microsoft Corporation. Intel a Thread Building Blocks jsou známky chránící produkty firmy Intel Corporation. IDS Imaging a uEye jsou ochranné známky společnosti IDS Imaging Development Systems GmbH. V textu se mohou vyskytovat další jména produktů a firem, které jsou chráněny ochrannými známkami příslušných vlastníků.

Abstract

The topic of this thesis is to design and implement an image recognition module that infers a current state of a real FIRA MiroSot robotic soccer game from a digital camera with frame rate of 50 frames per second. The module is written in C++ language and takes advantage of the OpenCV library for real-time computer vision processing. Furthermore, acceleration techniques for image processing on GPU with the CUDA framework are inspected. The recognition module is integrated into robotic soccer platform written in `C#` language developed at the University of West Bohemia and the communication is held over network via sockets.

Obsah

1 Úvod **1**

2 Teoretická část **2**
2.1 Robotický fotbal 2
2.1.1 Pravidla pro MiroSot Middle League 3
2.2 Požadavky 5
2.3 Předchozí práce 7
2.3.1 Bakalářská práce Bc. Vojtěcha Friče 7
2.3.2 Předchozí práce ostatních týmů 8
2.4 Řídicí software robotického fotbalu 8
2.4.1 Moduly a zprávy 8
2.4.2 Komunikace přes sockety 10
2.5 Barevné prostory 10
2.5.1 Prostor RGB 12
2.5.2 Prostor HSV (HSI) 13
2.6 CUDA 15
2.6.1 Úvod 15
2.6.2 Architektura 17
2.6.3 Programátorské rozhraní 18
2.7 OpenCV 21
2.7.1 Základní operace s maticemi a obrázky 22
2.7.2 Zpracování a filtrace obrazu 27
2.7.3 Segmentace a hledání obrysů 28
2.7.4 Minimální ohraničující obdélník 30
2.7.5 Modul GPU 31
2.8 Thread Building Blocks 33
2.9 Kamera 34

3 Realizace **37**
3.1 Celkový popis činnosti 37
3.2 Síťová komunikace 38

3.2.1 Zprávy . . . 38
3.3 Rozpoznávací modul . . . 43
3.3.1 Celkové schéma činnosti . . . 43
3.3.2 Algoritmus rozpoznávání . . . 47
3.3.3 Identifikace našich robotů . . . 56
3.3.4 Perspektivní korekce a normalizace souřadnic . . . 57
3.3.5 Implementace . . . 59
3.4 Klientský modul . . . 64
3.4.1 Grafické uživatelské rozhraní . . . 64
3.4.2 Implementace . . . 71
3.5 Testování . . . 82
3.5.1 Příprava hřiště, robotů a kamery . . . 82
3.5.2 Nastavení kamery . . . 83
3.5.3 Testování rychlosti . . . 84
3.6 Zhodnocení výsledků . . . 86

4 Závěr **91**

A Uživatelská příručka **98**
A.1 Spuštění . . . 98
A.2 Síťové nastavení kamery . . . 100
A.3 Nastavení rozpoznávacího modulu ve Visual Studiu . . . 102
A.4 Nastavení řídicí platformy . . . 107
A.5 Instalace knihoven . . . 109
A.5.1 Thread Building Blocks . . . 109
A.5.2 CUDA . . . 110
A.5.3 OpenCV . . . 110
A.5.4 Instalace knihoven kamery . . . 111

B Obrazové přílohy **112**

Poděkování

Děkuji Ing. Kamilu Ekšteinovi, Ph.D. a Bc. Petru Altmanovi za vytvoření technického zázemí pro robotický fotbal a podporu při jeho vývoji, neboť bez nich by celý projekt, ani tato práce nemohly vzniknout. Dále děkuji své rodině za podporu během studia.

1 Úvod

Robotický fotbal je celosvětová soutěž technických univerzit, která si klade za cíl rozvíjet a porovnávat schopnosti studentů v oblastech robotiky, umělé inteligence, multiagentních systémů [1], počítačového vidění a dalších souvisejících oborů.

Prvotní nápad na vytvoření týmu Západočeské univerzity v Plzni vznikl zhruba před třemi lety jako společná iniciativa studentů Katedry informatiky a výpočetní techniky Fakulty aplikovaných věd ZČU a Fakulty elektrotechnické ZČU ve spolupráci s Katedrou mechaniky Fakulty aplikovaných věd ZČU. Během prvního roku běhu projektu na Katedře informatiky a výpočetní techniky byla vytvořena řídicí modulární platforma, která umožňuje vyvíjet herní strategie, komunikovat s roboty a zobrazovat herní stav prostřednictvím 3D vizualizačního modulu, který kromě zobrazování stavu hry může sloužit i jako virtuální kamera, která hledí na simulovanou hru robotického fotbalu. Návrh a implementace tohoto modulu byl náplní mé bakalářské práce.

V té době nebyl stále ještě k dispozici hrací stůl, kamera, ani roboti, takže vývoj se musel odehrávat na virtuálních datech. Na základě obrazových dat generovaných modulem virtuální kamery vznikl v druhém roce vývoje rozpoznávací modul, jehož autorem je Bc. Vojtěch Frič, který tedy jako první prozkoumal úskalí rozpoznávání pozic a natočení robotů a polohy míčku při hře robotického fotbalu, byť jen na základě syntetických dat generovaných 3D počítačovou grafikou. V současné době však již je k dispozici skutečný hrací stůl a digitální kamera schopná snímat průběh skutečné hry. Roboti, o jejichž vývoj se stará Fakulta elektrotechnická, však k datu 16. května 2013 stále ještě nejsou k dispozici.

Cílem této práce je tedy navrhnout a implementovat rozpoznávací modul, který ze skutečných vizuálních dat z digitální kamery umístěné ve výšce 2,5 metru nad hřištěm dokáže rozpoznat herní stav a předat informaci o něm řídicí platformě tak, aby s ní mohly v budoucnu pracovat moduly herní strategie. Tato práce tedy poslouží jako základ pro budoucí rozvoj projektu robotického fotbalu a tím i výzkumné činnosti studentů na katedře.

2 Teoretická část

2.1 Robotický fotbal

Robotický fotbal je prestižní celosvětová soutěž mezi technickými univerzitami, která si klade za cíl porovnávat a rozvíjet dovednosti studentů v oblastech umělé inteligence, robotiky, mechatroniky, multiagentních systémů [1], softwarového inženýrství a dalších souvisejících oborů. Základním cílem hry je sestavit tým robotů a jejich řídicí software tak, aby dopravily míček do brankoviště soupeře na hracím stole.

Počet robotů v jednotlivých týmech, jejich velikost, i další technická omezení a herní pravidla pak závisí na konkrétní lize a typu soutěže v ní definované. Mezi největší celosvětové asociace zaštiťující robotický fotbal patří *RoboCup* [2] a *Federation of International Robot-soccer Association* (dále jen *FIRA*) [3]. První oficiální soutěž RoboCupu se odehrála v roce 1997, FIRA organizovala svůj první pohár již o rok dříve.

FIRA rozděluje pohár do několika dílčích soutěží, v závislosti na velikosti robotů a požadavcích na jejich autonomii. V lize *HuroCup* soutěží roboti, kteří se svým vzhledem i stylem hry podobají nejvíce svým lidským protějškům. Roboti v této lize mohou být až 150 cm vysocí a vážit až 30 kg. V lize *AmireSot* soutěží plně autonomní roboti vybavení systémem počítačového vidění přímo uvnitř těla robota. V kategorii *MiroSot* soutěží roboti, kteří již nejsou plně autonomní, ale jsou vzdáleně ovládáni centrálním počítačem, jehož součástí je i modul počítačového vidění. Velikost podstavy robotů je omezena na 75 × 75 mm, výška robota je rovněž shora omezena na 75 mm.

Liga *NaroSot* je velmi podobná lize MiroSot, avšak rozměry robotů jsou omezeny na 40 × 40 × 55 mm. V lize *AndroSot* je omezující hmotnost robota, která smí být maximálně 600 g. V lize *RoboSot* hrají jeden až tři roboti, kteří mohou být polo- či plně autonomní. Jejich podstava smí mít maximálně 200 × 200 mm a jejich výška nemá žádná omezení. Liga *SimuroSot* se hraje pouze na virtuálním hřišti uvnitř počítačového serveru a týmy se do ní připojují jako softwaroví klienti. Ukázkové otografie z kategorie *MiroSot* se nachází na obrázku B.2.

2.1.1 Pravidla pro MiroSot Middle League

Tým Západočeské univerzity soutěží v lize FIRA, konkrétně v kategorii MiroSot Middle League. Dále se proto budeme zabývat pravidly a technickými aspekty souvisejícími pouze s touto kategorií. Pravidla [4] jsou poměrně rozsáhlá, a proto se v této podsekci omezíme pouze na pravidla přímo související s potřebami modulu počítačového vidění.

Hřiště

Hrací pole musí mít obdélníkový tvar o rozměrech 2,2 × 1,8 m a musí být natřeno matnou černou barvou. Hřiště ohraničuje stěna o výšce 50 mm a tloušťce 25 mm. Jeho vnitřní stěny jsou bílé a vrchní část ohraničujících stěn je nabarvena černě. Hřiště musí být ve všech místech ploché a vodorovné, což zjistíme tak, že se míček v jakémkoliv místě na hřišti po položení nezačne samovolně pohybovat. Do rohů hřiště jsou umístěny hranoly s podstavou pravoúhlého rovnoramenného trojúhelníka, jehož odvěsny mají délku 70 mm. Jejich účel je zabránit tomu, aby míček uvízl v rozích hřiště.

Hřiště by mělo obsahovat bílé hrací čáry nakreslené přesně podle náčrtku na obrázku 2.1. Všechny čáry musí být 3 mm tlusté. Středová kružnice má poloměr 250 mm. Hřiště musí být umístěno uvnitř budovy.

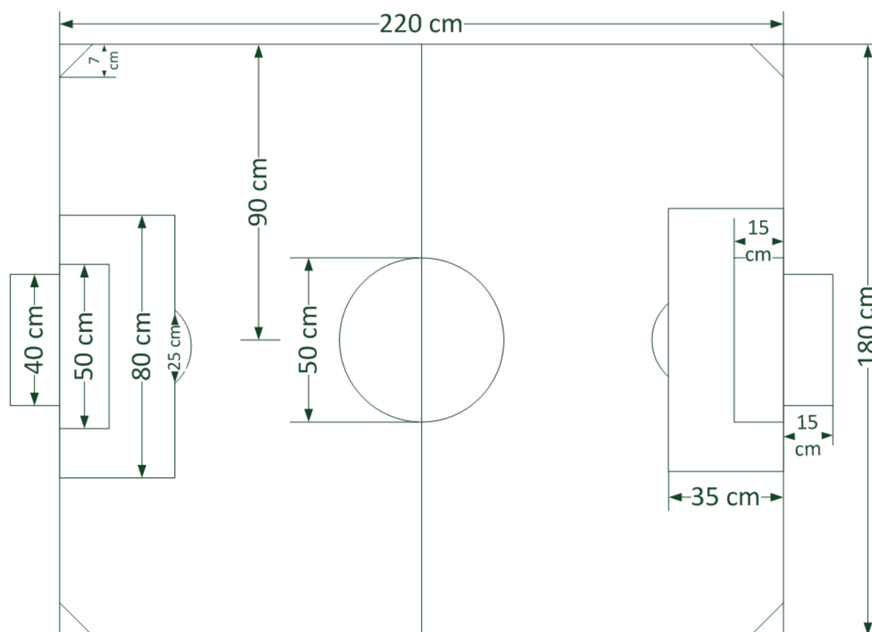
Brankoviště je 400 mm široké a nesmí obsahovat síťku. Gólová linie je umístěna těsně před začátek brankoviště.

Ke hře smí být použit oranžový golfový míček o průměru 42,7 mm a váze 46 g.

Osvětlení a kamera

Hrací plochu osvětluje difúzní světlo, jehož intenzita ve všech místech hrací plochy musí dosahovat alespoň 500 Lx. Je doporučeno použít světelný zdroj, který nebliká.

Každý tým smí použít k rozpoznávání stavu hru pouze jednu kameru, která smí být umístěna kdekoliv nad polovinou stolu náležící danému týmu, včetně středové čáry. Pokud si oba týmy přejí mít kameru nad středem hřiště,



Obrázek 2.1: Náčrtek s rozměry hřiště v kategorii MiroSot Middle League.

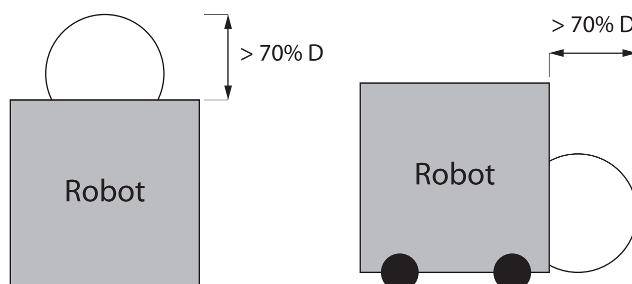
musí být umístěny vedle sebe tak, aby od středu měly stejnou vzdálenost, a zároveň byly co nejblíže u sebe. Kamera je umístěna ve výšce 2,5 m nad plochou hřiště.

Roboti a štítky

Každý tým se skládá z 5-ti robotů, z nichž jeden smí mít roli brankáře. Rozměry každého robota jsou omezeny krychlí o rozměrech $75 \times 75 \times 75$ mm. Délka komunikační antény se do rozměrů nezapočítává. Roboti smí obsahovat ruce, nohy a další vylepšení, jejich celková velikost však musí respektovat uvedené rozměry, i ve stavu s plně rozvinutými končetinami.

Hmotnost robota smí být maximálně 650 g. Stěny robota by měly mít světlou barvu, aby bylo umožněno infračervené snímání. Roboti by měly nosit uniformu, jejíž velikost je omezena na $80 \times 80 \times 80$ mm. Tyto uniformy nesmí mít jiný účel než chránit robota a nést identifikační štítek. Robot musí být plně funkční i bez této uniformy a musí ji být snadné vyměnit.

Každý robot má vlastní napájení a motory uvnitř svého těla a s centrál-



Obrázek 2.2: Pravidlo 30% – robot nesmí při pohledu z boku ani seshora zakrývat svým tělem více jak 30% průměru míčku.

ním počítačem smí komunikovat pouze bezdrátově. Žádný robot nesmí míček držet tak, aby zakrýval více jak 30% z průměru míčku při pohledu seshora ani ze stran (viz obrázek 2.2).

Vrchní strana robota nesmí mít oranžovou barvu. Každý tým před začátkem zápasu dostane od organizátorů přidělenu modrou nebo žlutou barvu, čímž bude odlišena týmová příslušnost jednotlivých robotů. Všechny identifikační štítky musí obsahovat minimálně 35 mm velkou oblast vyplněnou odpovídající týmovou (tedy modrou, nebo žlutou) barvou. Barevná identifikace se může každou hru měnit, a proto musí být identifikační štítky snadno vyměnitelné. Roboti jednoho týmu nesmí na svém identifikačním štítku mít stejné nebo podobné barvy soupeřova týmu.

2.2 Požadavky

Cílem této práce je:

- Navrhnout a implementovat rozpoznávací algoritmus, který bude z obrazových dat digitální kamery se snímkovací frekvencí 50 snímků za sekundu rozpoznávat herní stav v souladu s pravidly (viz podsekce 2.1.1).

Obrázek 2.3: Typické geometrické návrhy identifikačních štítků robota používané v reálných zápasech v kategorii MiroSot. [8][9][10][11]

- Algoritmus musí snímek zpracovávat dostatečně rychle, tj. doba zpracování musí být kratší než 20 ms (protože máme snímkovací frekvenci kamery 50 snímků za sekundu). Ideálně by rychlost rozpoznávání měla být okolo 6 ms, aby zbyl dostatek času pro výpočet herní strategie.
- Implementovat modul v jazyce C++ za použití knihovny OpenCV ([17]).
- Vytvořit rozpoznávací modul jako serverovou aplikaci a zajistit integraci tohoto modulu do řídicí platformy robotického fotbalu.
- Otestovat funkčnost celého řešení na reálných datech z digitální kamery ([18]).

S přihlédnutím k pravidlům robotického fotbalu v kategorii MiroSot musí být algoritmus dostatečně obecný, aby dokázal rozpoznat typické geometrické tvary a designy identifikačních štítků, které se používají ve skutečných zápasech. Typické vzhledy identifikačních štítků jsou znázorněny na obrázku 2.3.

2.3 Předchozí práce

2.3.1 Bakalářská práce Bc. Vojtěcha Friče

Tato práce navazuje na bakalářskou práci Bc. Vojtěcha Friče [5] a klade si za úkol eliminovat její známé nedostatky. V době, kdy byla bakalářská práce vyvíjena, nebyl k dispozici herní stůl a rozpoznávací algoritmus byl laděn pouzen na syntetických snímcích generovaných vizualizačním modulem.

Vizualizační modul [6] generuje snímky tak, že renderuje scénu s trojrozměrnými modely hřiště a robotů tak, jak by je viděla skutečná kamera umístěná nad hřištěm ve výšce 2,5 m skrze ortogonální projekci. Z tohoto důvodu se generované snímky realitě pouze přibližují, a tedy funkčnost na simulovaných datech nemusí nutně znamenat funkčnost na reálných datech ze skutečné kamery.

Zjednodušující předpoklady využity v bakalářské práci oproti skutečnosti jsou tyto:

- Snímek je umělý a neobsahuje žádný šum, na rozdíl od obrazových dat ze skutečné kamery.
- Snímky z vizualizačního modulu použité pro rozpoznávání byly vykresleny ortogonální projekcí, takže v každém místě hřiště vidíme pouze vrchní identifikační štítek robota. Ve skutečnosti kamera vnímá perspektivu, takže kromě štítku robota jsou ve výsledném snímku vidět i jeho stěny. Kvůli perspektivnímu zkreslení je navíc potřeba provést korekci výsledků, neboť objekty blíže kameře (např. identifikační štítek) se v promítnutém obrazu jeví větší a na jiném místě, než doopravdy jsou. Rozdíl mezi snímkem generovaným perspektivní a ortogonální projekcí ilustruje obrázek B.1.
- Rozpoznávací algoritmus je navržen a odladěn pouze pro jeden speciální typ štítku. Pravidla však povolují každému týmu navrhnout si svůj vlastní štítek dle libosti (za předpokladu splnění požadavků z podsekce 2.1.1).

První dva zmíněné nedostatky nepovažuji za příliš zásadní, obrovským problémem je však třetí bod, tedy nedostatečná obecnost algoritmu, který

je připraven pouze pro jeden konkrétní druh štítku, neboť u ostatních typů by algoritmus selhal. Cílem této práce je proto všechny tyto zmíněné nedostatky odstranit a navrhnout algoritmus, který bude dostatečně obecný, aby dokázal pracovat pro téměř libovolný druh identifikačního štítku navrženého soupeřem v souladu s pravidly.

2.3.2 Předchozí práce ostatních týmů

Konkrétní řešení ostatních týmů je poměrně těžké dohledat, pravděpodobně kvůli tomu, že si autoři přejí ponechat konkurenční výhodu před ostatními týmy. Některé přístupy však byly zveřejněny ve vědeckých článcích [8][9][10][11].

Prakticky všechny dohledané články řeší problém na základě segmentace robotů prahováním barev a s tím související volbou barevného prostoru. Některé přístupy používají pro klasifikaci tvar nalezených komponent nebo kombinaci obou přístupů. Algoritmy založené na Fourierově transformaci [12] mohou podávat robustnější výsledky, avšak pro reálné nasazení bývají zamítnuty kvůli příliš velké časové náročnosti výpočtu [13].

2.4 Řídicí software robotického fotbalu

Pro účely týmu Západočeské univerzity byla vytvořena řídicí platforma robotického fotbalu. Rozpoznávací modul musí s touto platformou komunikovat, aby rozpoznaný herní stav mohl být předán ke zpracování dalším modulům, zejména herní strategii. Jádro robotického fotbalu je napsáno v jazyce `C#` a je postaveno na modulární bázi tak, že každá logická jednotka (např. modul pro rozpoznávání obrazu, modul herní strategie, modul komunikace s roboty apod.) je umístěna ve svém vlastním projektu (překládaném jako DLL knihovna). Moduly se tak mohou dynamicky přidávat a načítat. Podrobné informace o řídicím software robotického fotbalu lze nalézt v [7].

2.4.1 Moduly a zprávy

Ve jmenném prostoru *RoboSoccer.Core.Modules* se nachází předpřipravené třídy pro různé typy modulů. Jejich základem je vždy abstraktní třída *Robo-*

SoccerModule. Nejdůležitějšími metodami této třídy jsou *SendMessage*, *RegisterMessageEvent*, *OnInitialize* a *OnDispose*.

Metoda *SendMessage* slouží k odeslání zprávy ostatním modulům. Parametrem je pouze zpráva, kterou chceme odeslat. Každý modul si sám definuje druhy zpráv, které chce přijímat. K tomu slouží generická metoda *RegisterMessageEvent*. Při volání se do špičatých závorek umisťuje třída zprávy, kterou si přeje modul přijímat. Parametrem této metody je pak konkrétní metoda, která bude tento typ zpráv obsluhovat.

Metody *OnInitialize* a *OnDispose* se volají automaticky po spuštění modulu, resp. ihned před uvolněním modulu z jádra robotického fotbalu. V metodě *OnInitialize* můžeme například registrovat druhy zpráv, které chceme modulem přijímat. Metoda *OnDispose* by měla uvolnit použité zdroje vzniklé při běhu modulu.

Rozpoznávací modul musí být potomkem abstraktní třídy *ImageRecognitionModule*, který ostatním modulům posílá zprávu definovanou ve třídě *MessageGameState*. Tato zpráva nese informaci o herním stavu (definovaném ve třídě *GameState*) – tedy pozice a natočení robotů (třída *RobotState*) a pozici míčku (třída *BallState*). Pozice robotů i míčku jsou vyjádřeny jako X-ové a Y-ové souřadnice v normalizovaném rozsahu $\langle -1, 1\rangle$. Z pohledu kamery tedy souřadnice $[-1, -1]$ značí levý horní roh hřiště a souřadnice $[1, 1]$ označují pravý dolní roh. Úhly natočení robotů jsou vyjádřeny v radiánech.

Každý modul se překládá jako dynamická knihovna (tj. s příponou `.dll`), která se umísťuje do adresáře se spustitelným programem robotického fotbalu. Ten v konfiguračním souboru *config.xml* přečte seznam knihoven s moduly, které si přejeme používat, a provede jejich inicializaci. Pomocí reflexe pak v načtené knihovně určí instanci třídy *RoboSoccerModule*, se kterou bude dále komunikovat.

Knihovna s modulem navíc může obsahovat třídu, která je potomkem třídy *ModuleForm*. Ta definuje okno představující prezentační vrstvu modulu. Po načtení knihovny s modulem se toto okno inicializuje a zobrazí se v platformě robotického fotbalu.

2.4.2 Komunikace přes sockety

Vzhledem k tomu, že rozpoznávací modul z důvodu rychlosti existuje jako samostatná aplikace napsaná v jazyce C++, není možné ji integrovat do řídicího software robotického fotbalu přímo. Místo toho spolu komunikuje rozpoznávací modul a platforma robotického fotbalu na bázi síťové architektury *klient – server*, kde rozpoznávací modul běží jako samostatný proces (teoreticky i na jiném počítači) a představuje server, ke kterému se řídicí software robotického fotbalu připojuje jako klientská aplikace.

Sockety jsou mechanizmus umožňující aplikacím komunikovat a předávat balíky informací (tzv. *packety*) v počítačové síti prostřednictvím protokolů rodiny *TCP/IP* [14]. Socket je popsán:

- Lokální IP adresou a portem.
- IP adresou a portem cílového uzlu, se kterým chceme komunikovat.
- Transportním protokolem.

Ke komunikaci se zpravidla používají protokoly *TCP (Transmission Control Protocol)* a *UDP (Universal Datagram Protocol)*, jejichž nejdůležitější vlastnosti a rozdíly shrnuje tabulka 2.1.

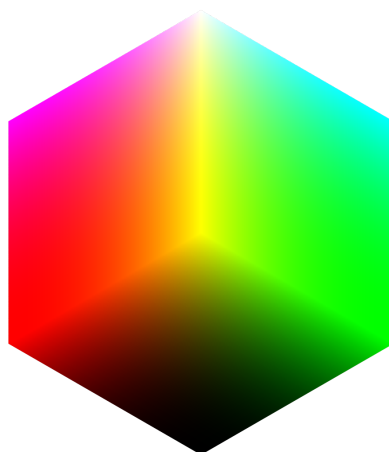
Pro vytváření aplikací využívající sockety pod operačním systémem Microsoft Windows v jazyce C++ existuje knihovna *Windows Sockets 2* deklarovaná v hlavičkovém souboru *winsock2.h.* Podrobné informace o programovacím rozhraní a architektuře socketů pro systém Microsoft Windows lze nalézt v [15].

2.5 Barevné prostory

Rozpoznávání štítků dle pravidel vyžaduje správné odlišení různých odstínů barev – přinejmenším žluté a modré (viz sekce 2.1.1). Barva může být v počítači reprezentována jako vektor v několika barevných prostorech, například v prostoru *RGB*, *HSV*, *HLS* a dalších [16]. Jejich vlastnosti a způsoby reprezentace barev se mohou významně lišit a volba vhodného barevného prostoru

	TCP	*UDP*
Druh spojení	Ke komunikaci je nejprve nutné navázat spojení.	Není vyžadováno navázání spojení mezi komunikujícími entitami.
Pořadí doručení packetů	V pořadí, jakém byly odeslány.	Nedefinované, řazení zajišťuje aplikace.
Rychlost přenosu	Obecně pomalejší z důvodu vyšší režie na přenos.	Obecně rychlejší než TCP.
Spolehlivost	Garantuje doručení dat tak, jak byla odeslána. Ztratí-li se po cestě, jsou vyžádána znovu.	Data nemusí dorazit vůbec, případně mohou dorazit v jiném pořadí.
Potvrzení o doručení	TCP vždy potvrzuje doručení zprávy.	UDP nedokáže doručení ověřit.
Velikost hlavičky	20 bytů	8 bytů

Tabulka 2.1: Porovnání vlastností a hlavních rozdílů protokolů komunikačních protokolů TCP a UDP.



Obrázek 2.4: Krychle znázorňující barevný prostor RGB. Osy prostoru představují hodnoty základních barevných složek - červené, zelené a modré. V počátku se nachází černá barva, v protilehlém rohu krychle bílá barva.

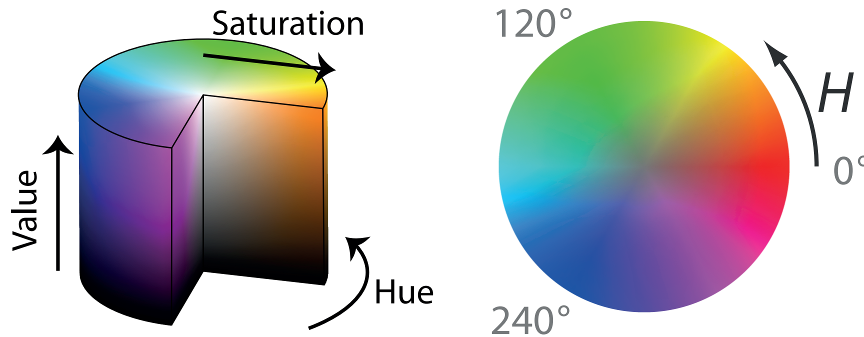
může zjednodušit proces rozpoznávání. V této sekci se proto budeme zabývat volbou vhodného prostoru barev pro účely rozpoznávání robotů ve hře robotického fotbalu.

2.5.1 Prostor RGB

Barevný prostor RGB (z anglického *Red, Green, Blue*) vyjadřuje barvu jako kombinaci tří základních barev – červené, zelené a modré. Tyto složky jsou v počítači typicky vyjádřeny jako nezáporná celá čísla v rozsahu 0 až 255 (hloubka 8 bitů). Představíme-li si každou ze zmíněných základních barev jako osu v trojrozměrném prostoru, můžeme všechny barvy vyjádřit uvnitř krychle, v jejíchž rozích se nachází černá (všechny složky nulové), červená, modrá, fialová, zelená, žlutá, tyrkysová a bílá barva (všechny složky mají maximální hodnotu), viz obrázek 2.4.

Výhody:

- Jedná se o standardní barevný prostor, a není proto třeba snímek získaný z kamery převádět do jiného barevného prostoru, což může být časově náročné.



Obrázek 2.5: Válec znázorňující barevný prostor HSV (vlevo). Úhel na kruhové výseči kolmo k ose válce vyjadřuje odstín, vzdálenost od osy válce představuje nasycení a vzrůstající hodnoty od spodní podstavy podél osy válce vyjadřují intenzitu barvy. Obrázek vpravo znázorňuje posloupnost odstínů barev se vzrůstající hodnotou úhlu *Hue*. (Autor obrázků: Jacob Rus.)

Nevýhody:

- Je obtížné vyjádřit odstín barvy, tj. vymezit například oblast, která obsahuje všechny odstíny žluté barvy.

2.5.2 Prostor HSV (HSI)

Barevný prostor HSV (z anglického *Hue, Saturation, Value*, někdy též označován jako HSI – *Hue, Saturation, Intensity*) vyjadřuje barvu jejím odstínem (*Hue*), nasycením (*Saturation*) a intenzitou (*Value / Intensity*). Barevný prostor má tvar válce. Odstín barvy je znázorněn na kruhu a je vyjádřen úhlem od 0 do 360 °. Nasycení je zobrazeno jako poloměr této kružnic, nejmenší nasycení představuje odstíny šedi, nejvyšší nasycení pak plné odstíny barev (jako je žlutá, fialová, oranžová, zelená apod.). Osou válce je intenzita, která určuje světlost dané barvy. Prostor HSV je ilustrován na obrázku 2.5.

Pro převod z barevného prostoru RGB do prostoru HSV můžeme využít níže uvedené vzorce. Nejprve definujeme maximální složku (M, viz vzorec 2.1) a minimální složku (m, viz vzorec 2.2) ze všech RGB hodnot. Chromacitu C, pak určíme jako rozdíl maximální a minimální složky (viz vzorec 2.3).

$$M = max(R, G, B) \tag{2.1}$$

$$m = min(R, G, B) \tag{2.2}$$

$$C = M - m \tag{2.3}$$

Hodnoty pro odstín (H), nasycení (S) a intenzitu (V) pak určíme ze vzorců 2.4, 2.5, 2.6 a 2.7.

$$H' = \begin{cases} \text{nedefinováno}, & \text{když } C = 0 \\ \frac{G-B}{C} \; mod \; 6, & \text{když M=R} \\ \frac{B-R}{C} + 2, & \text{když M=G} \\ \frac{R-G}{C} + 4, & \text{když M=B} \end{cases} \tag{2.4}$$

$$H = 60^\circ \times H' \tag{2.5}$$

$$S = \begin{cases} 0, & \text{když } C = 0 \\ \frac{C}{V}, & \text{jinak} \end{cases} \tag{2.6}$$

$$V = M \tag{2.7}$$

Výhody:

- Odstín barvy je jasně definovaný jako jedna ze základních složek. Tento model díky tomu blíže odpovídá lidskému vnímání barev.
- Při klasifikaci barev na štítku robota nás zajímá převážně odstín barvy, neboť intenzita se může vlivem nerovnoměrného osvětlení v různých místech hřiště měnit.

Nevýhody:

- Je nutné obraz z kamery převést z prostoru RGB do prostoru HSV, což zvyšuje dobu nutnou ke zpracování snímku.

2.6 CUDA

2.6.1 Úvod

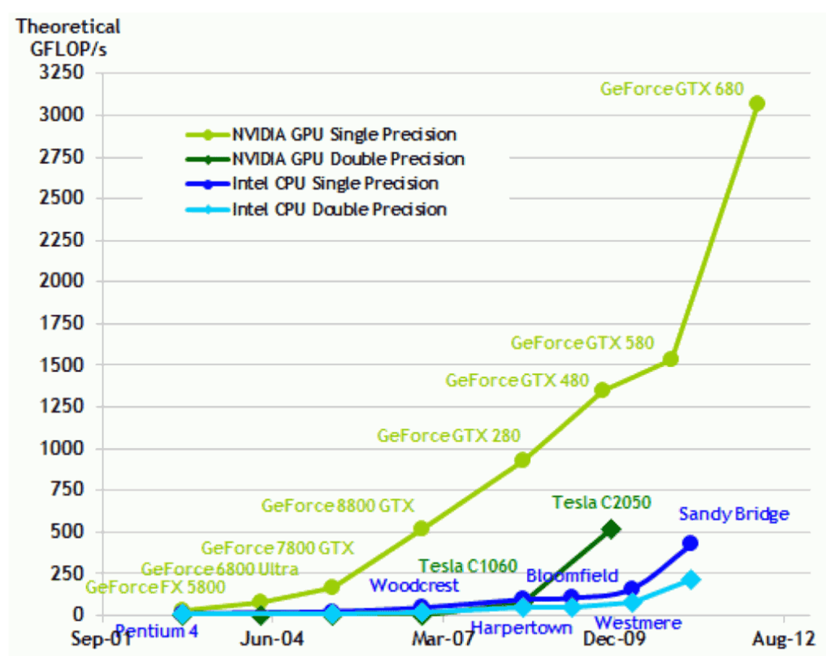
Compute Unified Device Architecture (zkráceně CUDA) je platforma od společnosti NVIDIA pro paralelizaci výpočtů prostřednictvím GPU (*Graphics Processing Units*). Pod pojmem GPU se označují moderní grafické karty schopné provádět uživatelem definované mikroprogramy zvané *shadery* či *kernely.*

GPU oproti standardním procesorům (CPU) obsahuje daleko více úzce specializovaných výpočetních jader, jejichž počet se pohybuje v řádech stovek až tisíců. To se samozřejmě projevuje na znatelně vyšším teoretickém (i praktickém) výkonu (viz obrázek 2.6).

CUDA v současné době podporuje pouze grafické karty od společnosti NVIDIA. Hlavním konkurentem platformy CUDA je jazyk *Open Computing Language* [22] (zkráceně OpenCL), který umožňuje paralelizaci výpočtů i na grafických procesorech od jiných výrobců.

CUDA obsahuje několik modulů a knihoven, které implementují často používané algoritmy z různých odvětví matematiky, informatiky a zpracování digitálního signálu:

- *cuFFT* – Modul, který provádí rychlou Fourierovu transformaci signálu za použití GPU.
- *cuBLAS* – Modul implementující práci s maticemi, vektory a rozšířenými algoritmy z lineární algebry na GPU.
- *CULA* – Značně optimalizovaná varianta modulu *cuBLAS*, která je však placená.
- *cuRAND* – Generátor kvalitních náhodných čísel implentovaný na GPU.
- *NPP* (NVIDIA Performance Primitives) – Vybrané algoritmy z oblasti počítačového vidění optimalizované pro GPU.



Obrázek 2.6: Srovnání maximálního teoretického výkonu moderních CPU a GPU. GPU nabízí daleko vyšší teoretický výkon, což je dáno masivním počtem úzce specializovaných jader. Za povšimnutí stojí výrazný rozdíl ve výkonu při užití výpočtů s jednoduchou přesností (angl. *single precision*, v normě ANSI C odpovídá datovému typu *float*) oproti dvojnásobné přesnosti (angl. *double precision*, v normě ANSI C odpovídá datovému typu *double*). (Graf převzat z [21])

- *GPU AI* – Sada algoritmů urychlených na GPU pro potřeby umělé inteligence a multiagentních systémů – plánování cesty, vyhýbání se překážkám apod.
- *Thrust* – Knihovna sloužící jako nadstavba nad základní programátorské rozhraní CUDA. Odstiňuje programátora od problémů se synchronizací, nabízí řadu užitečných postupů, jako je např. *Map-Reduce* schéma pro paralelní zpracování dat, nabízí možnost využití šablon a dalších specifických vlastností jazyka C++. Knihovna tedy zvyšuje produktivitu a rychlost vývoje aplikací pro platformu CUDA.

Popis všech modulů a detailů architektury není možné v této práci obsáhnout, neboť by byl značně rozsáhlý a komplikovaný. Cílem této sekce je pouze velmi letmo čtenáře seznámit se základními principy fungování a možností platformy CUDA tak, aby bylo možné pochopit její přednosti a úskalí. Zájemcům o detailní popis architektury, programovacího rozhraní a jednotlivých modulů doporučuji [19] a [20].

2.6.2 Architektura

Pro pochopení architektury CUDA a moderních grafických karet je užitečné znát jejich historický vývoj. Grafické karty sloužily (a dodnes slouží) primárně k urychlení výpočtů při vykreslování scény v počítačových hrách a jiných rozšířených aplikacích trojrozměrné počítačové grafiky. Vykreslovaná scéna se typicky skládá ze stovek tisíc až milionů trojúhelníků potažených *texturami* (bitmapovými obrázky). Ke zpracování těchto primitiv stačí výpočetní jednotka, která zvládá základní operace s vektory a maticemi.

Průběh zpracování jednoho trojúhelníka nebo pixelu je navíc zcela nezávislý na stavu zpracování ostatních trojúhelníků a pixelů, úlohu je proto možné snadno paralelizovat. Grafické karty tedy obsahovaly velké množství jednodušších a vysoce specializovaných výpočetních jader pro práci s vektory a maticemi, oproti několika pomalejším jádrům centrálního procesoru, který však mohl být použit pro libovolné výpočty.

S postupem času výpočetní jádra grafických karet již umožňovala být programována uživatelem tzv. *shader* programy. Tyto programy dávají uživateli možnost ovlivnit zpracování každého trojúhelníku či pixelu. Výpočetní jádra v moderních GPU se stala natolik obecná, že se již schopnostmi velmi přibližují centrálním procesorům. CUDA je tedy logickým pokračovatelem tohoto

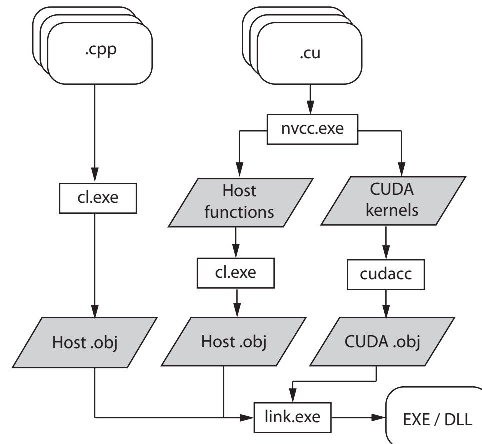
trendu a dává uživateli možnost psát libovolné programy složené ze základních aritmetických a logických operací, čtení a zápisů do paměti, cyklů apod. Ačkoliv se moderní GPU čím dál tím více podobají CPU, je vzhledem k jejich povaze třeba stále dbát na následující omezení:

- Centrální procesor a GPU spolu komunikují přes sběrnici PCI-Express. Ke zpracování snímku z kamery je nutné proto snímek (či jiná data) nejprve zkopírovat z CPU na GPU a po zpracování je přenést zpátky. To i přes vysoké přenosové rychlosti v řádech stovek MB/s může způsobit několika milisekundové latence.
- Několik výpočetních jader (typicky 32) tvoří tzv. *warp blok*, který sdílí malou, ale extrémně rychlou cache paměť. Pokud si jádro vyžádá načtení dat z globální paměti, která je velká, ale znatelně pomalejší, dochází k drastickému snížení výkonu jádra, které operaci vyžádalo, ale i okolních jader ve stejném bloku. Ačkoliv tedy CUDA podporuje náhodný přístup do paměti, zdaleka nejrychleji funguje, pokud čteme a zpracováváme data sekvenčně.
- Výpočetní jádro je schopno uchovat program jen s omezeným počtem instrukcí, a proto není možné psát neomezeně dlouhé programy.
- Výpočty s datovým typem *float* jsou výrazně rychlejší (typicky až 10×) než stejné výpočty s datovým typem *double*. Je tedy vhodné všechny výpočty provádět s nižší přesností.

2.6.3 Programátorské rozhraní

CUDA programy se vytváří v jazyce velmi podobném normě ANSI C. Typicky se tyto programy umisťují do samostatného souboru s příponou `.cu`. Ty jsou následně přeloženy programem *nvcc.exe* a výsledné objekty se linkují dohromady se standardně přeloženými objekty v jazyce C nebo C++. Postup překladu ilustruje obrázek 2.7.

Mikroprogramy, které běží na GPU, se označují pojmem *kernel*. CUDA vyžaduje při jejich deklaraci užít klíčové slovo __*global*__, které dává najevo, že definovaná procedura bude spouštěna na GPU. Velmi jednoduchý kernel lze vidět na fragmentu CUDA kódu níže:



Obrázek 2.7: Schéma kompilace a vytvoření spustitelného programu nebo knihovny s použitím CUDA. Klasické *.cpp* soubory kompiluje nativní překladač (např. Visual Studio), CUDA kernel programy (přípona *.cu*) se kompilují zvlášť překladačem *nvcc*. Výsledné přeložené objekty se slinkují dohromady do výsledného spustitelného programu či knihovny. (Schéma převzato z [20])

```
1  #define N 10
2
3  // Priklad kernel programu, ktery paralelne scita
4  // prvky z~pole A~a B a uklada vysledek do pole C.
5  __global__ void VecAdd( int *A, int *B, int *C ) {
6    int tid = threadIdx.x;
7    if (tid < N)
8    C[tid] = A[tid] + B[tid];
9  }
10
11 int main() {
12   // Pripravime pole pro vysledek
13   int cpu_C[N];
14
15   // Alokujeme na GPU pamet pro pole A, B a C
16   int *A, *B, *C;
17   cudaMalloc( (void**)&A, N * sizeof(int) ) );
18   cudaMalloc( (void**)&B, N * sizeof(int) ) );
19   cudaMalloc( (void**)&C, N * sizeof(int) ) );
20
21   // Spustime 1 blok po N vlaknech zpracovavajici kernel.
22   VecAdd<<<1, N>>>(A, B, C);
23
24   // Prekopirujeme vysledek z~GPU do centralni pameti.
```

```
25	cudaMemcpy( cpu_C, C, N * sizeof(int), cudaMemcpyDeviceToHost
	    );
26
27	// Vypisime vysledek
28	for (int i=0; i<N; i++) printf("%d ", cpu_C[i]);
29
30	// Uvolnime pamet na GPU
31	cudaFree(A);
32	cudaFree(B);
33	cudaFree(C);
34 }
```

Kernel spouštíme speciální konstrukcí ve tvaru:

`jméno_kernelu<<<počet_bloků, počet_vláken_v_bloku>>>(parametry)`

Hodnota *počet_bloků* určuje kolik výpočetních bloků se má spustit. Každý výpočetní blok spouští *počet_vláken* krát zadaný kernel v samostatném vlákně na GPU. Kernel pak může zjistit číslo bloku, ve kterém je spouštěn, z předdefinované proměnné *blockIdx* a číslo vlákna z proměnné *threadIdx*. Tyto proměnné mají tři složky - x, y a z, které značí číslo bloku, resp. vlákna v odpovídajících dimenzích. Při spouštění kernelu totiž nemusíme zadat pouze jedno číslo (tj. pokud se hodnoty pohybují pouze v jedné dimenzi), ale můžeme zadat i dvourozměrný či trojrozměrný rozsah čísel datovými strukturami *dim2* či *dim3*. Dvourozměrná dimenze je například výhodná pro zpracovávání obrázků, protože jí můžeme definovat čtvercovou oblast bodů, které chceme zpracovat.

Kernely mohou pracovat pouze s ukazateli do paměti, která je alokována na GPU. Tu můžeme spravovat voláním funkcí *cudaMalloc* a *cudaFree*, které jsou analogií ke standardním funkcím *malloc* a *free* v jazyce ANSI C. Kopírování mezi GPU a CPU (nebo naopak) zajišťuje funkce *cudaMemcpy*, která funguje analogicky k funkci *memcpy* v jazyce ANSI C, rozdíl je pouze v tom, že čtvrtý parametr určuje směr kopírování dat:

- *cudaMemcpyDeviceToHost* – kopírujeme-li data z GPU do centrální paměti, aby k nim bylo možné přistupovat z CPU.
- *cudaMemcpyHostToDevice* – kopírujeme-li data z centrální paměti do paměti GPU.

Pokročilejší ukázky práce s programátorským rozhraním CUDA lze nalézt například v [19][20][21].

2.7 OpenCV

Open Computer Vision (dále jen *OpenCV* [17]) je jedna z nejpoužívanějších knihoven, která implementuje algoritmy z oblasti počítačového vidění. Standardní varianta je napsána v jazyce C++ a je šířena pod BSD licencí [23], která umožňuje volné akademické i komerční využití. Existuje i varianta pro jazyk `C#` nazvaná *EmguCV* [24].

Knihovna nabízí celou škálu algoritmů, které spadají do několika základních modulů:

- *Core* – definuje základní datové typy (matice, body, vektory apod.) a implementuje nad nimi základní matematické operace jako je sčítání, násobení atd.
- *Image Processing* – implementuje algoritmy pro zpracování digitalizovaného obrazu, jako je například prahování, morfologické operace, filtrace, geometrické transformace, histogramy a popis ploch.
- *High-level GUI and Media* – umožňuje vytváření jednoduchých grafických uživatelských rozhraní, které slouží především k zobrazování zpracovaných obrazových dat, a také umožňuje načítat obrázky a video sekvence ze souborů v běžně používaných datových formátech.
- *Video* – implementuje algoritmy pro detekci pohybu, zejména pro účely bezpečnostních kamer.
- *Camera Calibration and 3D Reconstruction* – algoritmy pro kalibraci kamery a hledání korespondujících bodů v problému trojrozměrné rekonstrukce ze stereo kamery [25].
- *2D Features framework* – algoritmy pro extrakci a popis obrazových příznaků (SIFT, STAR, SURF, ORB apod.) a jejich párování s příznaky ve vzorovém obrazu a následné hledání geometrické transformace mezi nimi.

- *Object Detection* – detekce komplexních objektů v obraze, jako jsou chodci, lidské obličeje apod., zároveň nabízí možnost natrénování klasifikátoru pro uživatelem definované objekty.
- *Machine Learning* – modul implementuje algoritmy strojového učení a klasifikace, jako jsou například rozhodovací stromy, Bayesův naivní klasifikátor, Expectation-Maximization a umělé neuronové sítě [26][27] [28].
- *GPU-Accelerated Computer Vision* - nabízí vybrané algoritmy z předchozích modulů urychlené technologií CUDA (viz sekce 2.6)

Popis všech algoritmů a modulů by byl značně obsáhlý a značně nad rámec této práce. Existuje několik publikací, které se algoritmy v daných modulech zabývají po teoretické, i praktické stránce, zejména [29] a [30]. Rovněž lze odkázat na oficiální online dokumentaci na webu [31]. Dále však budeme popisovat algoritmy a operace nutné pro potřeby této práce.

2.7.1 Základní operace s maticemi a obrázky

Inicializace matice

Třída *cv::Mat* definuje matici, kterou však v OpenCV můžeme používat nejen například pro definování matice transformace, ale i pro samotné obrázky. Matici inicializujeme konstruktorem ve tvaru:

```
1 cv::Mat image = cv::Mat(height, width, format);
```

Je třeba si dávat pozor na definování rozměrů, neboť výška obrázku musí být uvedena jako první argument – pracujeme s maticí, takže je konvencí, aby první souřadnice vždy označovala řádky a druhá souřadnice sloupce. Počet řádků matice (výšku obrázku) pak můžeme číst z atributu *rows* a počet sloupců (šířku) z atributu *cols*. Formát matice je třetím argumentem a může typicky nabývat hodnot:

- *CV_8U* – Prvky matice jsou osmibitová celá čísla bez znaménka. Často využívané pro šedotónový obraz.

- *CV_8UC2* až *CV_8UC4* – Prvky matice jsou osmibitová celá čísla bez znaménka, které jsou však umístěny do několika kanálů (počet kanálů je definován za znakem *C*). Typicky můžeme využít formát *CV_8UC3* pro RGB obrázky, či jiné tříkanálové barevné systémy.
- *CV_16S* – Prvky matice jsou 16-bitová celá čísla se znaménkem.
- *CV_32S* – Prvky matice jsou 32-bitová celá čísla se znaménkem.
- *CV_32F* – Prvky matice jsou reálná čísla typu *float*.
- *CV_64F* – Prvky matice jsou reálná čísla typu *double*.

Po inicializaci se uvnitř matici mohou vyskytovat libovolné hodnoty, protože se prvky matice při inicializaci nenastavují na nulu. Toho můžeme docílit voláním statického konstruktoru *cv::Mat::zeros* nebo analogicky *cv::Mat::ones*, pokud chceme, aby po inicializaci všechny prvky měly hodnotu 1. Operátorem '=' můžeme přiřadit všem prvkům již inicializované matice uživatelem definovanou hodnotu. Tyto přístupy ilustruje fragment kódu níže:

```
1 cv::Mat zeroImage = cv::Mat::zeros(height, width, format);
2 cv::Mat oneImage = cv::Mat::ones(height, width, format);
3 cv::Mat fiftyFiveImage = cv::Mat(height, width, format);
4 fiftyFiveImage = 55;
```

Načítání obrázků a barevné prostory

Pro načtení obrázků ze souboru můžeme používat funkci *cv::imread*.

```
1 cv::Mat image = cv::imread("obrazek.jpg");
```

Je důležité dávat pozor na to, že obrázek se standardně načítá ve formátu *BGR*, nikoliv *RGB* (pořadí červené a modré složky je tedy prohozeno). K převodu mezi jednotlivými formáty můžeme použít funkci *cv::cvtColor*. Jejím prvním argumentem je zdrojový obrázek, druhým argumentem cílová matice, do které se zapíše převedený obrázek a třetím argumentem je kód, který definuje druh převodu mezi barevnými systémy. Následující fragment kódu ilustruje převod načteného obrázku v barevného prostoru BGR do prostoru HSV:

```
1 cv::Mat bgrImage = cv::imread("obrazek.jpg");
2 cv::Mat hsvImage;
3 cv::cvtColor(bgrImage, hsvImage, cv::COLOR_BGR2HSV);
```

Kód pro převod je vždy ve tvaru `cv::xxx2yyy`, kde *xxx* určuje barevný prostor vstupního obrázku (první argument funkce) a *yyy* určuje cílový barevný prostor. Při užití barevného prostoru HSV (či HLS nebo dalších, které vyjadřují některou ze složek jako úhel na jednotkové kružnici) je třeba dbát na to, že je úhel zmenšen dvojnásobně (z intervalu $\langle 0, 360 \rangle$ do intervalu $\langle 0, 180 \rangle$ stupňů), aby jej bylo možné uložit do datového typu *byte*.

Chceme-li barevný obrázek rozdělit na jednotlivé barevné roviny (tedy například získat 3 samostatné obrázky s rovinou B, G a R), můžeme použít proceduru *cv::split*, která vrátí původní obrázek rozdělený do obrázků s jednotlivými rovinami uloženými v seznamu. Analogicky můžeme roviny sloučit zpět do jednoho barevného obrázku procedurou *cv::merge*.

```
1 cv::Mat bgrImage = cv::imread("obrazek.jpg");
2 std::vector<cv::Mat> bgrPlanes;
3 cv::split(bgrImage, bgrPlanes);
4 cv::imshow("Modra rovina", bgrPlanes[0]);
5 cv::Mat mergedImage;
6 cv::merge(bgrPlanes, mergedImage);
7 cv::imshow("Sjednoceny obraz", mergedImage);
```

Pro zobrazení matice jako obrázku můžeme použít proceduru *cv::imshow*. Jejím prvním argumentem je řetězec identifikující okno (a zároveň nadpis tohoto okna), ve kterém se zadaný obrázek zobrazí. Druhým parametrem je obrázek, který si přejeme zobrazit. Obrázky s jedním kanálem se zobrazí v odstínech šedi, tříkanálové matice se považují za obrázky ve formátu BGR.

```
1 cv::Mat image = cv::imread("obrazek.jpg");
2 cv::imshow("Okno", image);
```

Čtení a zápis

K samotným datům uvnitř matice (obrázku) pak můžeme přistupovat šablonovou funkcí *at*. Při použití je do špičatých závorek třeba specifikovat základní datový typ, který chceme přečíst nebo zapsat do matice na danou pozici. Jako datový typ můžeme uvést libovolný základní datový typ v jazyce C++, nebo odvozené typy, jako jsou vektory, např.:

- *cv::Vec3b* – Třísložkový vektor složený z datového typu *byte*. Typicky používané pro čtení a zápis pixelů v třísložkových barevných prostorech, jako je BGR či HSV.

- *cv::Vec3f* – Třísložkový vektor složený z datového typu *float*. Typicky používané pro čtení a zápis trojrozměrných vektorů.
- *cv::Point2i* – Bod s celočíselnými souřadnicemi ve formátu *int*.
- *cv::Point2f* – Bod s reálnými souřadnicemi ve formátu *float*.
- *cv::Point2d* – Bod s reálnými souřadnicemi ve formátu *double*.

Chceme-li projít celý obrázek a změnit hodnotu R-složky všech pixelů na hodnotu 255, můžeme použít následující fragment kódu:

```
cv::Mat image = cv::imread("obrazek.jpg");
for (int y = 0; y < image.rows; y++)
{
    for (int x = 0; x < image.cols; x++)
    {
        // Nacteme aktualni hodnotu.
        cv::Vec3b color = image.at<cv::Vec3b>(y, x);
        // Prepiseme R-kanal. (treti kanal - obrazek je BGR)
        color[2] = 255;
        // Zapiseme upravenou barvu zpet do matice.
        image.at<cv::Vec3b>(y, x) = color;
    }
}
```

OpenCV nedokáže zjistit, jaká je skutečná struktura dat v matici. Je proto důležité při čtení a zápisu uvést ve funkci *at* správný datový typ. Je rovněž důležité dávat si pozor na pořadí souřadnic – opět při indexaci je prvním argumentem *řádek* – tedy *Y-ová* souřadnice obrázku. K datům matice můžeme přistupovat i přímo pomocí ukazatelů, což může být rychlejší. Ukazatel na první prvek matice se nachází v atributu *data*. Následující fragment kódu přebarví první pixel na bílou barvu:

```
cv::Mat image = cv::imread("obrazek.jpg");
cv::Vec3b whiteColor = cv::Vec3b(255, 255, 255);
cv::Vec3b* firstPixel = (cv::Vec3b*)image.data;
*firstPixel = whiteColor;
```

Oblasti zájmu a kopírování

Oblasti zájmu, nebo také *Regions of Interest (ROI)*, umožňují definovat podmatici (tedy výseč obrázku), nad kterou chceme, aby funkce a procedury

OpenCV volané v následujících krocích pracovaly. Chceme-li například zpracovat jen část obrázku kolem robota, můžeme definovat ohraničující obdélník, v němž se robot nachází, a tento obdélník pak představuje naši oblast zájmu v obrazu. Tím je možné snížit dobu zpracování, protože algoritmus pak pracuje jen na malé výseči původního obrazu.

Zájmovou část obrazu získáme z původního obrázku C++ operátorem *()*, do kterého předáme jako argument obdélník (struktura *cv::Rect*) definující oblast našeho zájmu. Následující fragment kódu vybere z původního obrázku pouze jeho horní polovinu a zobrazí ji:

```
1 cv::Mat image = cv::imread("obrazek.jpg");
2 cv::Rect upperHalf = cv::Rect(0, 0, image.cols, image.rows / 2);
3 cv::Mat upperHalfImage = image(upperHalf);
4 cv::imshow("Horni polovina", upperHalfImage);
```

Matice definovaná v předchozím příkladu je však tzv. *mělká kopie*, což znamená, že změny v ní provedené se projeví i v původním obrazu. To nemusí být vždy žádoucí chování, a proto OpenCV nabízí možnosti kopírování a klonování obrázků. Nechceme-li původní obraz dalšími úpravami poškodit, můžeme vytvořit jeho přesnou kopii voláním metody *clone*.

```
1 cv::Mat image = cv::imread("obrazek.jpg");
2 cv::Mat clonedImage = image.clone();
```

Oblast zájmu můžeme také použít při kopírování. Metoda *copyTo* zkopíruje daný obrázek do cílového obrázku. Jako cílový obraz však můžeme definovat podobraz za užití oblasti zájmu. Následující fragment kódu ilustruje vytvoření prázdného obrázku, který je dvakrát tak velký, jako původní obraz. Do prostředka pak zkopírujeme původní obrázek metodou *copyTo*:

```
1 cv::Mat image = cv::imread("obrazek.jpg");
2 cv::Mat bigImage = cv::Mat::zeros(image.rows * 2, image.cols *
    2);
3 cv::Rect roi = cv::Rect(image.cols / 2, image.rows / 2,
4     image.cols, image.rows);
5 image.copyTo(bigImage(roi));
```

Obrázek 2.8: Fotografie před (vlevo) a po převodu na šedotón a aplikaci binárního prahování s prahem 128 (vpravo). Prahováním v tomto případě můžeme ve snímku zhruba odlišit popředí od pozadí.

2.7.2 Zpracování a filtrace obrazu

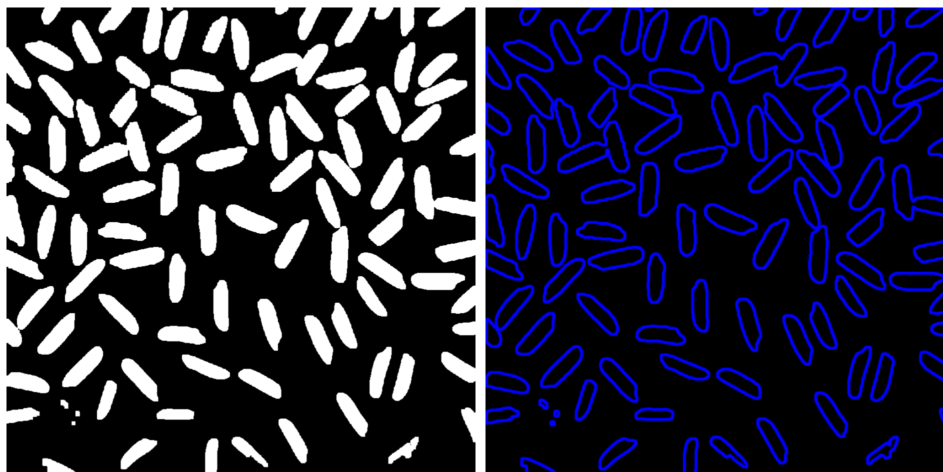
Prahování

Prahování je operace, která mění intenizitu každého bodu ve vstupním obrazu tzv. *prahovací funkcí*, jejímž smyslem je nastavit všechny intenzity nad uživatelem definovaným prahem (reálným číslem) či pod ním na určitou hodnotu. Přiřazujeme-li ve výsledném snímku pouze 2 hodnoty (například černou a bílou barvu), mluvíme o tzv. *binárním prahování* (viz obrázek 2.8). To se dá například využít k odlišení výrazných barevných štítků robotů od tmavého pozadí hracího stolu.

V OpenCV lze prahovat pouze jednokanálové obrázky, tedy například matice ve formátu *CV_8U*. OpenCV nabízí 5 módů prahování, v práci je však využita pouze základní prahovací funkce s kódem *cv::THRESH_BINARY* (viz vzorec 2.8). Tento mód do výstupního obrazu *dst* zapíše hodnotu *maxVal*, je-li hodnota ve vstupním obrázku *src* na dané pozici ostře větší než prahovací hodnota *threshold*:

$$dst\,(x,y) = \begin{cases} maxVal & \text{když } src(x,y) > threshold \\ 0 & \text{jinak} \end{cases} \tag{2.8}$$

Prahování v OpenCV zajišťuje procedura *cv::threshold*, jejíž prvním parametrem je vstupní obrázek (*src*), druhým parametrem je výstupní obraz



Obrázek 2.9: Obraz vzniklý binárním prahováním zrnek rýže na tmavém pozadí (vlevo) a nalezené kontury souvislých komponent (vpravo).

po prahování (*dst*), třetím parametrem je práh (*threshold)*, čtvrtým argumentem je zapisovaná hodnota pro hodnoty nad prahem (*maxVal*) a pátým parametrem je kód prahovací funkce. Použití ilustruje následující fragment zdrojového kódu:

```
1 cv::Mat image = cv::imread("obrazek.jpg");
2 cv::Mat grayImage;
3 cv::cvtColor(image, grayImage, cv::BGR2GRAY);
4 cv::Mat thresholdImage;
5 int thresholdVal = 50;
6 cv::threshold(grayImage, thresholdImage, thresholdVal, 255,
7     cv::THRESH_BINARY);
8 cv::imshow("Obraz po prahovani", thresholdImage);
```

2.7.3 Segmentace a hledání obrysů

Binárním prahováním vznikne obraz, který má pouze dva druhy pixelů – body, které patří do myšleného popředí (typicky znázorněné bílou barvou), a body, které tvoří pozadí (černá barva). Segmentace je proces, který najde všechny souvislé komponenty v popředí.

Každý segment (komponenta) je pak vymezen body, které tvoří jeho obrys (tj. body na pomezí popředí a pozadí, viz obrázek 2.9). Obrys (kontura)

je v OpenCV vyjádřen jako seznam bodů (uložených ve struktuře *cv::Point*). V obraze vzniklém binárním prahováním je možné najít všechny obrysy voláním procedury *cv::findContours*. Jejím prvním parametrem je matice s obrazem vzniklým prahováním, druhým argumentem je seznam, do kterého se uloží nalezené obrysy (vzhledem k tomu, že obrys je v OpenCV ukládán jako seznam, vznikne tak *seznam seznamů bodů*). Třetí parametr udává způsob, jakým se mají nalezené obrysy prohledávat a ukládat do seznamu. Může nabývat hodnot:

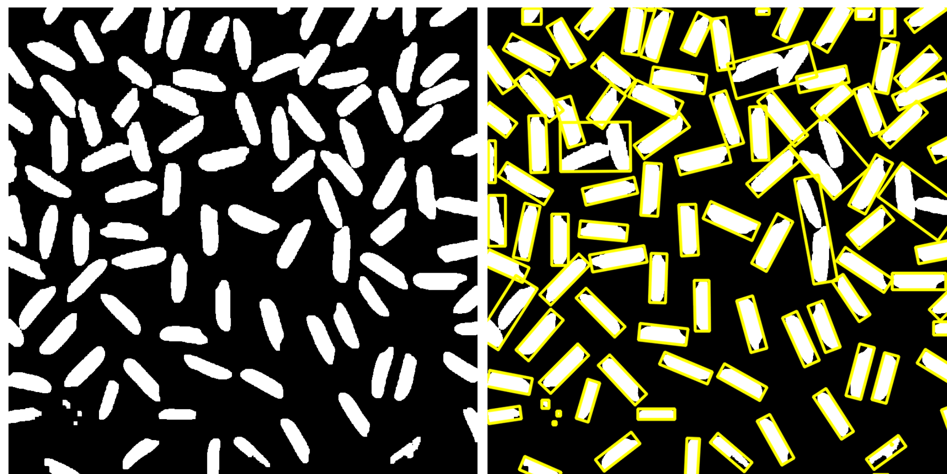
- *CV_RETR_EXTERNAL* - hledá pouze vnější obrysy komponent a ignoruje jakékoliv další možné vnořené komponenty.
- *CV_RETR_LIST* - nalezne všechny obrysy, včetně obrysů vnořených komponent, neuchovává však informaci o hierarchickém uspořádání nalezených komponent.
- *CV_RETR_TREE* - totéž jako CV_RETR_LIST, avšak navíc uchovává informaci o hierarchii ve stromové struktuře.

Čtvrtý parametr definuje způsob, jakým se mají body obrysu ukládat:

- *CV_CHAIN_APPROX_NONE* - uloží do seznamu všechny body obrysu.
- *CV_CHAIN_APPROX_SIMPLE* - komprimuje seznam bodů tak, že u horizontální, vertikálních a diagonálních čar ukládá pouze počáteční a koncové body.

Příklad použití procedury *cv::findContours* ilustruje fragment zdrojového kódu níže:

```
1  // Vytvorime obraz prahovanim.
2  ...
3  // Najdeme v~nem obrysy vsech souvislych komponent.
4  std::vector<std::vector<cv::Point>> contours;
5  cv::findContours(thresholdImage, contours, cv::RETR_LIST,
6      cv::CHAIN_APPROX_NONE);
7  for (int i=0; i<contours.size(); i++)
8  {
9      std::vector<cv::Point> contour = contours[i];
10     // Zde zpracuj obrys nalezene komponenty.
11     ...
12 }
```



Obrázek 2.10: Obraz vzniklý binárním prahováním zrnek rýže na tmavém pozadí (vlevo) a minimální ohraničující obdélníky nalezených souvislých komponent (vpravo žlutě).

2.7.4 Minimální ohraničující obdélník

Funkce *cv::minAreaRect* dokáže najít minimální ohraničující obdélník – tj. obdélník s minimálním obsahem, který je natočen tak, že těsně obepíná zadanou množinu bodů. Jediným parametrem je množina bodů, pro které chceme nalézt minimální ohraničující obdélník. Ten je pak vrácen funkcí ve třídě *cv::RotatedRect*.

Minimální ohraničující obdélník je užitečný, chceme-li najít *natočení* určité komponenty. To můžeme zjistit jednoduše tak, že nalezneme obrys zkoumané komponenty a obepneme kolem něj minimální ohraničující obdélník (viz obrázek 2.10).

Třída *cv::RotatedRect* obsahuje atribut *angle*, který udává úhel natočení ohraničujícího obdélníku ve stupních, a jeho hodnoty se vždy pohybují (poněkud nelogicky) v rozmezí od 0 do $-90\,^{\circ}$. Šířku a výšku obdélníka lze číst z atributů *width*, resp. *height*. Vrcholy minimálního obdélníka lze získat voláním procedury *points*, jejímž jediným argumentem je ukazatel na pole čtyř bodů, do kterých se vrcholy mají zapsat.

Následující fragment kódu ilustruje, jak z obrysu komponenty získat minimální ohraničující obdélník:

```
// Nalezneme obrysy vsech souvislych komponent.
std::vector<std::vector<cv::Point>> contours;
...
// Pro kazdy nalezeny obrys urcime minimalni
// ohranicujici obdelnik.
for (int i=0; i<contours.size(); i++)
{
    std::vector<cv::Point> contour = contours[i];
    // Najdeme ohranicujici obdelnik.
    cv::RotatedRect boundingBox = cv::minAreaRect(contour);
    printf("Uhel natoceni: %f\n", boundingBox.angle);
    // Nalezneme vrcholy ohranicujiciho obdelnika.
    cv::Point2f rectanglePoints[4];
    boundingBox.points(rectanglePoints);
    // Pouzijeme je pro nejaky dalsi vypocet.
    ...
}
```

2.7.5 Modul GPU

OpenCV implementuje vybrané algoritmy počítačového vidění na platformě CUDA (viz sekce 2.6). Tím lze dobu zpracování snímku výrazně urychlit, v porovnání s CPU až v řádu tisíců procent v závislosti na druhu použitého algoritmu (viz obrázek 2.13). Z principu však nelze všechny algoritmy efektivně převést na platformu CUDA, protože ne všechny se dají paralelizovat. Mezi dobře paralelizovatelné (a tím pádem dosahující i největší zrychlení) algoritmy patří primitivní operace s obrázky, jako je prahování, převod formátů barev, aritmetické operace s obrázky apod. Naopak algoritmy s typicky sekvenční povahou zpracovávání obrazu, jako je hledání obrysů, v balíku *cv::gpu* chybí, a proto jejich zpracování musí zařídit centrální procesor.

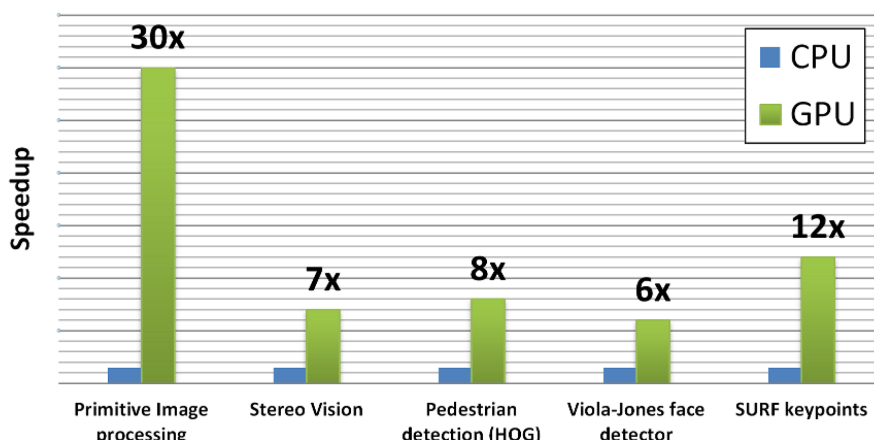
Protože algoritmy pracují na GPU, používají paměť na grafické kartě. Matice *cv::Mat* oproti tomu vždy reprezentuje blok hodnot uložených v paměti, která je určena pro CPU. Jako alternativa se tedy používá třída *cv::GpuMat*, se kterou se pracuje stejně jako s běžnou maticí, nabízí však ještě možnost nahrání dat z centrální paměti metodou *upload*, jejímž parametrem je klasická *cv::Mat* matice, jejíž data se do GPU matice zkopírují. Analogicky funguje metoda *download*, která zkopíruje data z GPU matice do matice v paměti, ke které má přístup CPU. Tyto operace lze pro pohodlnost nahradit i operátorem ’=’, který vnitřně tyto metody volá.

Následují fragment zdrojového kódu ilustruje nahrání snímku na GPU, následné prahování urychlené technologií CUDA a zpětné zkopírování výsledku do centrální paměti pro další zpracování:

```
1  // Nacteme snimek do matice v~centralni pameti.
2  cv::Mat cpuImage = cv::imread("obrazek.jpg");
3
4  // Alokujeme a nahrajeme snimek do pameti na GPU.
5  cv::gpu::GpuMat gpuDst, gpuSrc;
6  gpuSrc.upload(cpuImage);
7
8  // Provedeme prahovani na GPU
9  cv::gpu::threshold(gpuSrc, gpuDst, 128.0, 255.0,
       CV_THRESH_BINARY);
10
11 // Zkopirujeme vysledek z~GPU do matice pristupne z~CPU
12 // a zobrazime vysledek.
13 cv::Mat cpuResult = gpuDst;
14 cv::imshow("Vysledek po prahovani", cpuResult);
```

Při psaní programů využívající modul GPU je velmi důležité se vyhnout opakované alokaci a uvolňování paměti na GPU, neboť to je velmi drahá operace, v důsledku čehož může „urychlená“ varianta programu běžet pomaleji než na CPU. Dobrým přístupem je tedy před hlavní výpočetní smyčkou programu všechny potřebné matice předem alokovat a v těle smyčky již volat jen algoritmy, které zařizují vlastní zpracování snímku. Zároveň je třeba se vyhnout tomu, aby například při prahování byla zdrojová a cílová matice shodná, neboť to ve výsledku způsobuje, že si OpenCV vnitřně vytvoří dočasnou GPU matici (čímž způsobí drahou operaci alokace paměti), do které zapíše výsledek a ten pak zkopíruje do původní matice a uvolní dočasnou matici z CUDA paměti.

Při měření výkonu je rovněž nutné dbát na to, že první volání kterékoliv metody využívající technologii CUDA způsobí několika sekundovou prodlevu, neboť prostředí CUDA se inicializuje až při prvním použití. Před měřením je tedy vhodné zavolat například nahrání libovolných dat do paměti GPU, čímž se prostředí inicializuje, a teprve potom začít měřit.



Obrázek 2.11: Typická zrychlení doby výpočtů oproti CPU pro jednotlivé druhy algoritmů při použití OpenCV modulu GPU. K největšímu zrychlení dochází při použití základních obrazových transformací, jako je prahování nebo převod mezi barevnými formáty. (Převzato z [32])

2.8 Thread Building Blocks

Thread Building Blocks (dále jen *TBB*) je knihovna od společnosti Intel určená pro jazyk C++, jejímž cílem je usnadnit vývoj aplikací využívající paralelní výpočty na CPU [34]. Nabízí řadu generických algoritmů, jako jsou paralelní *for* cykly, operace typu *map - reduce*, paralelní algoritmy řazení a datové struktury uzpůsobené pro paralelní přístup, jako jsou například prioritní fronty, seznamy či hashovací tabulky.

Na rozdíl od klasických knihoven pro správu vláken jako *Pthreads* nebo *Boost threads* nepracuje uživatel s vlákny přímo, ale místo toho definuje *úlohy* (anglicky *task*), které se mají vykonat. TBB obsahuje plánovač, který tyto úlohy pak mapuje automaticky na volná jádra procesoru, tak aby se maximalizoval výkon a zefektivnilo se používání cache paměti. Navíc se tak minimalizuje režie při vytváření vláken, neboť o jejich vznik se stará plánovač, který pouze přiřazuje jednotlivým úlohám již předem vytvořená vlákna. Knihovna je zejména optimalizovaná pro moderní procesory od firmy Intel, na kterých podává nejvyšší výkon.

Úloha může být jakákoliv třída, která přetěžuje operátor *()*. V těle metody obsluhující tento operátor se musí nacházet kód, který má tvořit tělo úlohy,

tj. například nějaký časově náročný výpočet. Třída schopná vykonávat úlohu může vypadat například následovně:

```
1 class MyTask
2 {
3 public:
4     void operator()()
5     {
6         // Zde nejaky slozity vypocet.
7     }
8 };
```

Chceme-li paralelně spustit skupinu úloh, použijeme třídu *tbb::task_group*, ve které spouštíme jednotlivé úlohy metodou *Run*. Chceme-li počkat na dokončení všech úloh ve skupině, zavoláme nad instancí skupiny úloh metodu *wait*. Následující fragment kódu spustí paralelní výpočet dvou úloh definovaných třídou *MyTask*:

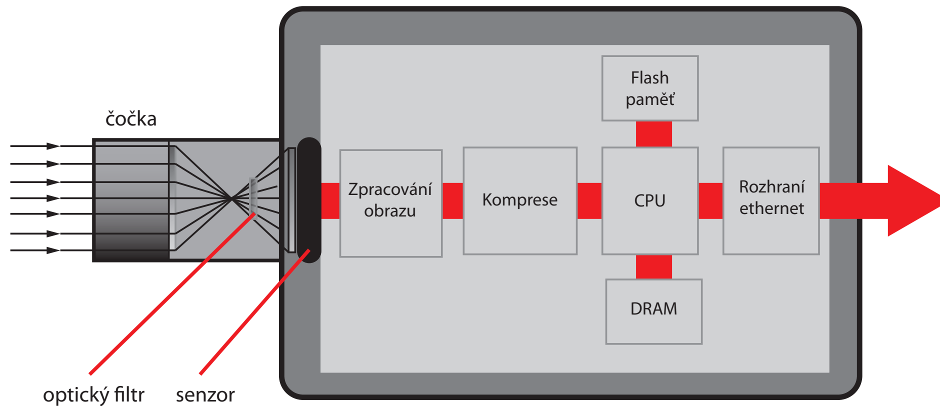
```
1 tbb::task_group group; // Vytvorime skupinu uloh.
2 group.run(MyTask()); // Spustime prvni ulohu MyTask.
3 group.run(MyTask()); // Spustime druhou ulohu MyTask.
4 group.wait(); // Pockame na dokonceni vypoctu uloh ve skupine.
```

2.9 Kamera

Kamera snímá scénu tak, že skrze optickou soustavu soustřeďuje světelné paprsky na snímací čip, který zaznamená obrazová data. Výsledek se dále zpracovává, například zesiluje, vyhlazuje, komprimuje, apod. Nakonec se obrazová data ukládají do paměti nebo se přenáší přes ethernet či jiné datové rozhraní do počítače. Schéma zpracování obrazu kamerou ilustruje obrázek 2.12.

Druh a velikost snímacího čipu významně ovlivňuje kvalitu výsledného obrazu, zejména množství šum, který se v něm nachází. Mezi nejpoužívanější technologie pro výrobu snímačů patří *CMOS* (Complementary Metal Oxide Semiconductor) a *CCD* (Charged Coupled Device). CCD snímače obecně poskytují lepší světelnou citlivost a kvalitnější obraz s menším množstvím šumu, jsou ale výrazně dražší na výrobu než čipy vyrobené technologií CMOS.

U čipů vyrobených technologií CMOS může dále také docházet k artefaktům, které se projevují zkosením obrazu ve směru pohybujícího se předmětu.

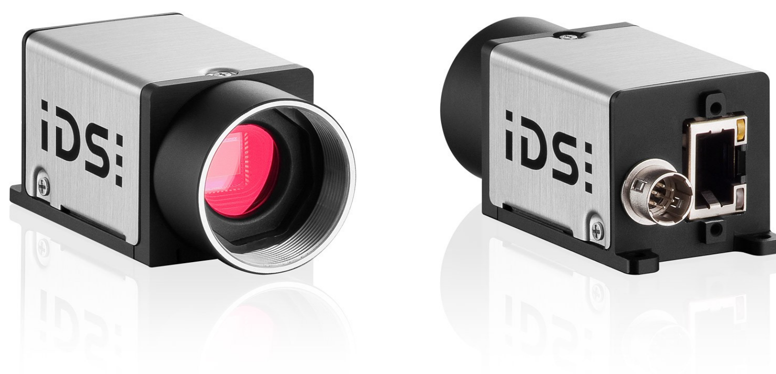


Obrázek 2.12: Schéma zpracování obrazu digitální kamerou.

Tento artefakt vzniká v tzv. *Rolling Shutter* snímacím módu, kdy kamera snímaná data ze senzoru čte řádek po řádku. Mezi tím, než však přečte celý obraz, se může snímaná scéna měnit, zejména vlivem velmi rychlého pohybu předmětů. Tím pádem je každý řádek sejmut v jiném okamžiku, což můžeme způsobit zkosení u horizontálně pohybujícího se předmětu. Rolling Shutter efekt je ilustrován na obrázku B.3.

Při použití metody *Global Shutter* je obraz přečten ze senzoru najednou, takže k tomuto jevu nedochází. Je tedy zřejmé, že pro účely robotického fotbalu je mód *Rolling Shutter* nežádoucí, protože roboti se pohybují velkou rychlostí a pro účely rozpoznávání potřebujeme získat co nejméně zkreslený obraz.

Ke snímání hřiště týmu Západočeské univerzity v Plzni byla zvolena digitální kamera *UI-5240CP* od společnosti *IDS Imaging* [35]. Ta nabízí rozlišení až 1280 × 1024 pixelů při snímací frekvenci 50 snímků za sekundu. Obraz je snímán CMOS čipem. Kamera umožňuje snímat scénu jak v módu *Rolling Shutter*, tak i *Global Shutter*.



Obrázek 2.13: Kamera UI-5240CP od společnosti IDS Imaging použitá pro snímání hřiště při hře hře robotického fotbalu týmu Západočeské univerzity v Plzni. Na obrázku vpravo lze vidět zadní část kamery, do které se připojuje napájení (kruhový vstup) a ethernetový kabel pro přenos dat. (Fotografie převzaty z [35])

3 Realizace

3.1 Celkový popis činnosti

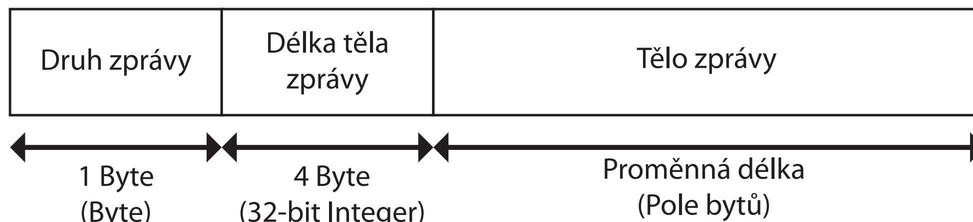
Celé řešení se skládá ze dvou samostatných aplikací, které spolu vzájemně komunikují. První z nich je řídicí software robotického fotbalu implementovaný v jazyce `C#`, pro který byl vytvořen *klientský modul*. Druhou z nich je rozpoznávací server naprogramovaný v jazyce C++, který je od řídicí platformy oddělen a pro dosažení maximálního výkonu funguje jako samostatná aplikace.

Rozpoznávací modul a řídicí software robotického fotbalu však spolu potřebují komunikovat. Rozpoznávací modul ze snímku kamery určí aktuální herní stav – tedy informaci o pozicích a natočení všech robotů a míčku. Rozpoznaný herní stav je pak nutné předat řídicí platformě, která o něm informuje ostatní moduly – zejména modul herní strategie, který na jeho základě naplánuje trajektorie pohybu robotů.

K tomu, aby rozpoznávací modul a řídicí software spolu mohly komunikovat, je do řídicího softwaru nutné integrovat *klientský modul*, jehož úkolem je od serveru získat rozpoznaný herní stav prostřednictvím síťové komunikace a předat jej jádru řídicí platformy robotického fotbalu, která o něm informuje ostatní moduly. Jako mechanismus komunikace mezi klientem a serverem byly zvoleny *sockety*, které dokáží zajistit rychlou komunikaci, zejména v případě, že klientská i serverová aplikace běží na stejném počítači.

Před začátkem procesu rozpoznávání stavu hry uživatel nastaví v grafickém uživatelském rozhraní, které je součástí klientského modulu, parametry robotů, jejich identifikační štítky, týmové barvy a další důležité parametry. Během nastavování server zasílá klientskému modulu aktuální snímky kamery, aby s nimi mohl uživatel dále pracovat.

Uživatelem zvolená nastavení pak klientský modul přemění na proud binárních dat a odešle je serveru, čímž se spustí proces rozpoznávání. V tomto módu již server neposílá klientovi snímky z kamery, pouze zprávy s aktuálním herním stavem. Klient tyto zprávy čte a rekonstruuje z binárního proudu dat rozpoznaný herní stav. Ten je pak předán v rámci řídicí platformy herním strategiím a jiným modulům, které o něj mají zájem (například vizualizačnímu modulu).



Obrázek 3.1: Obecný formát zprávy předávané mezi klientským modulem a rozpoznávacím serverem.

3.2 Síťová komunikace

Klientský modul v řídicím software robotického fotbalu a rozpoznávací modul spolu komunikují prostřednictvím TCP socketů. Protokol TCP byl zvolen proto, že zajišťuje na rozdíl od protokolu UDP spolehlivou komunikaci a respektuje pořadí, v jakém byly zprávy odeslány. Pro dosažení maximální rychlosti odezvy byl vypnut *Nagleův algoritmus*, takže se zpráva odešle okamžitě a nečeká se, než vznikne dostatečně velký datový *packet*.

3.2.1 Zprávy

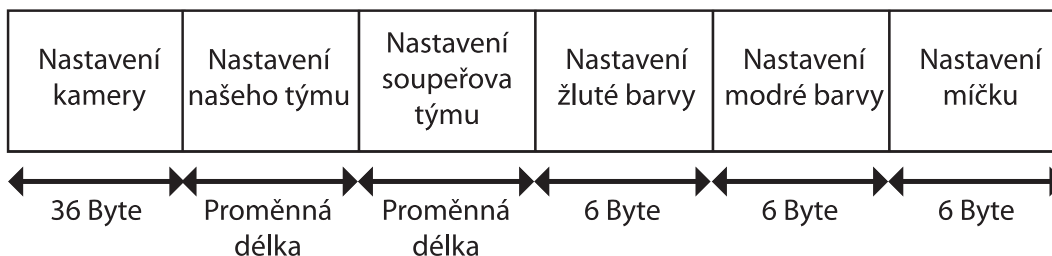
Klient a server mezi sebou komunikují mechanismem zasílání zpráv. Každá zpráva má formát ilustrovaný na obrázku 3.1. První *byte* zprávy určuje typ odeslané či přijaté zprávy. Číselné hodnoty a významy jednotlivých druhů zpráv shrnuje tabulka 3.1.

Po druhu zprávy následuje celé číslo datového typu 32-bitový *Integer*, které určuje délku těla zprávy v bytech. Po této hodnotě následuje pole bytů s tělem zprávy. Je-li hodnota délky těla nulová, nenásleduje po ní tělo zprávy a zpráva má tak celkovou délku pouze 5 bytů.

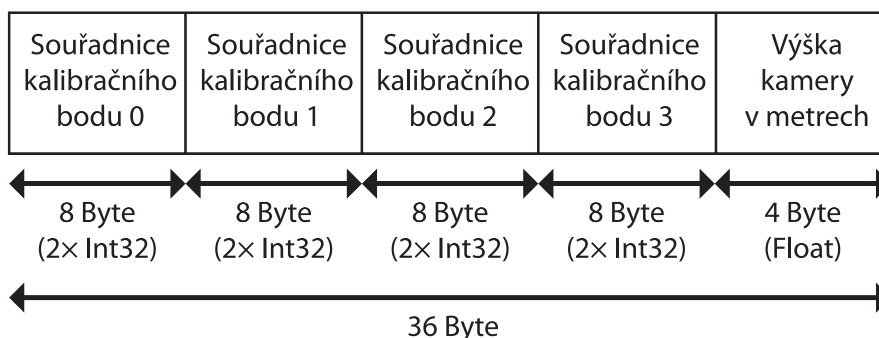
Typickým příkladem zprávy, která neobsahuje tělo, je zpráva požadující započetí detekce, která je reprezentována konstantou *MSG_START_DETECTION* (viz tabulka 3.1). Zpráva požadující od serveru aktuální snímek

ID zprávy	Konstanta	*Význam zprávy*
0	*MSG_START_ DETECTION*	Klient přikazuje serveru, aby začal s detekcí a zasíláním zpráv o rozpoznaném herním stavu.
1	*MSG_GAME_STATE*	Server posílá zprávu s aktuálním herním stavem.
2	*MSG_CHANGE_ SETTINGS*	Klient posílá serveru nastavení zadaná uživatelem v grafickém uživatelském rozhraní.
3	*MSG_CAMERA_ FRAME*	Klient touto zprávou žádá server o zaslání aktuálního snímku z kamery. Server odpovídá zprávou s tímto ID a v těle zprávy zasílá požadovaný snímek.
4	*MSG_CAMERA_ THRESHOLD_ FRAME*	Klient žádá touto zprávou server o zaslání aktuálního snímku z kamery po aplikaci prahování na základě zaslaných HSV prahů. Server odpovídá zprávou s tímto ID, která v těle obsahuje požadovaný snímek vzniklý prahováním.

Tabulka 3.1: Přehled druhů zpráv, jejich číselné identifikátory a konstanty, ve kterých jsou tyto hodnoty uloženy v rozpoznávacím a klientském modulu.



Obrázek 3.2: Formát těla zprávy s nastavením.

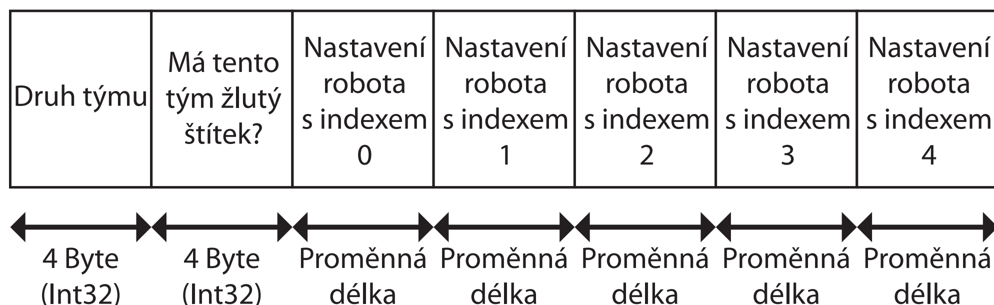


Obrázek 3.3: Formát přenášených dat s nastavením kamery.

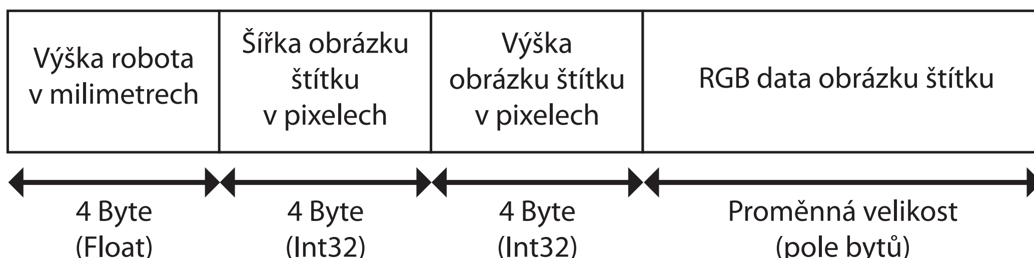
kamery též neobsahuje žádné tělo.

Tělo zprávy, ve které klient zasílá serveru uživatelem zvolená nastavení, má formát ilustrovaný na obrázku 3.2. Tělo zprávy je složeno z několika částí. První z nich je část obsahující nastavení kamery, ve které je uložena informace o souřadnicích rohů kalibračního obdélníka stolu. Každý ze čtyř bodů kalibračního obdélníka je vyjádřen jako dvojice 32-bitový hodnot datového typu *Integer*. Tyto hodnoty vyjadřují souřadnice X a Y daného rohu obdélníka ve snímku kamery. Dále nastavení kamery obsahuje informaci o tom, v jaké výšce v metrech nad hřištěm je kamera umístěna – viz obrázek 3.3.

Následují dvě části s nastavením našeho a soupeřova týmu. Formát nastavení každého týmu je ilustrován na obrázku 3.4. První položka je 32-bitový *Integer*, který má hodnotu 0, jedná-li se o nastavení našeho týmu, a hodnotu 1, jedná-li se o soupeřův tým. Následuje 32-bitový *Integer*, který rozhoduje,



Obrázek 3.4: Formát přenášených dat s nastavením týmu (našeho nebo soupeřova).

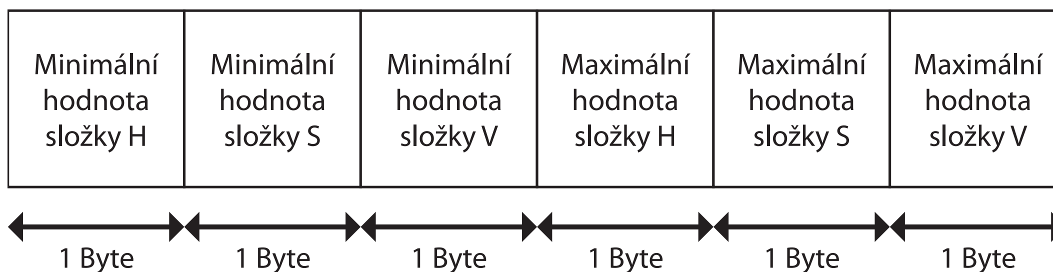


Obrázek 3.5: Formát přenášených dat s nastavením robota.

zda tento tým má od rozhodčího přidělenu žlutou (hodnota 1), nebo modrou (hodnota 0) barvu.

Poté následuje pětice nastavení – pro každého robota v týmu jedno. Každý robot ve svém nastavení nese informaci o svoji výšce v milimetrech reprezentován datovým typem *Float* a dále uchovává svůj identifikační štítek jako bitmapový obrázek v barevném prostoru RGB. Formát části zprávy s nastavením robota ilustruje obrázek 3.5.

Poslední tři části zprávy s celkovým nastavením definují prahy v barevném prostoru HSV, které vymezují odstín žluté, modré nebo oranžové barvy (pro míček). Toto vymezení je dáno trojicí HSV hodnot představující minimální povolenou HSV hodnotu pro danou barvu a trojicí HSV hodnot představující maximální povolenou hodnotu složek barvy v tomto prostoru.

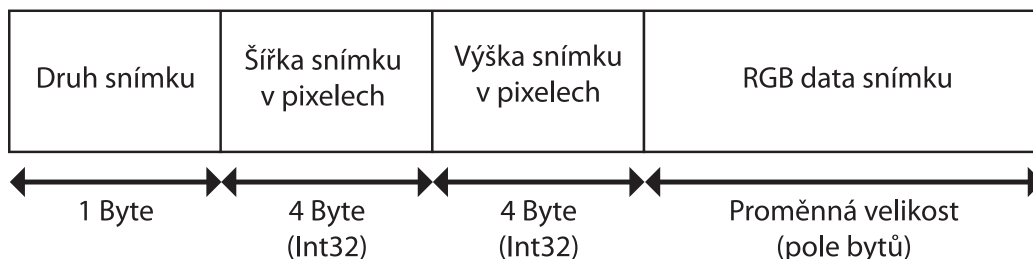


Obrázek 3.6: Formát přenášených dat s nastavením HSV prahů. Tento formát je použit pro nastavení žluté, modré a oranžové barvy (pro míček).

Formát těla zprávy s těmito nastaveními je ilustrován na obrázku 3.6.

Zpráva pro vyžádání snímku vzniklého prahováním, kterou posílá klient serveru, ve svém těle definuje minimální a maximální HSV prahy pro barvu, a má proto podobný formát, jako je ilustrován na obrázku 3.6. Jediný rozdíl je v tom, že před touto šesticí hodnot předchází ještě jeden *byte*, který určuje, v které části uživatelského rozhraní klientského modulu se má tento snímek po doručení zobrazit – viz sekce 3.4.2.

Server na žádost klienta o zaslání prahovaného či standardního snímku z kamery odpoví zprávou, jejíž tělo má formát ilustrovaný na obrázku 3.7. V prvním byte se nachází druh snímku, který je přenášen (viz sekce 3.4.2), a za ním následuje dvojice hodnot typu 32-bitový *Integer*, které představují šířku a výšku posílaného snímku v pixelech. Poslední část nese BGR bitmapová data s obrázkem.



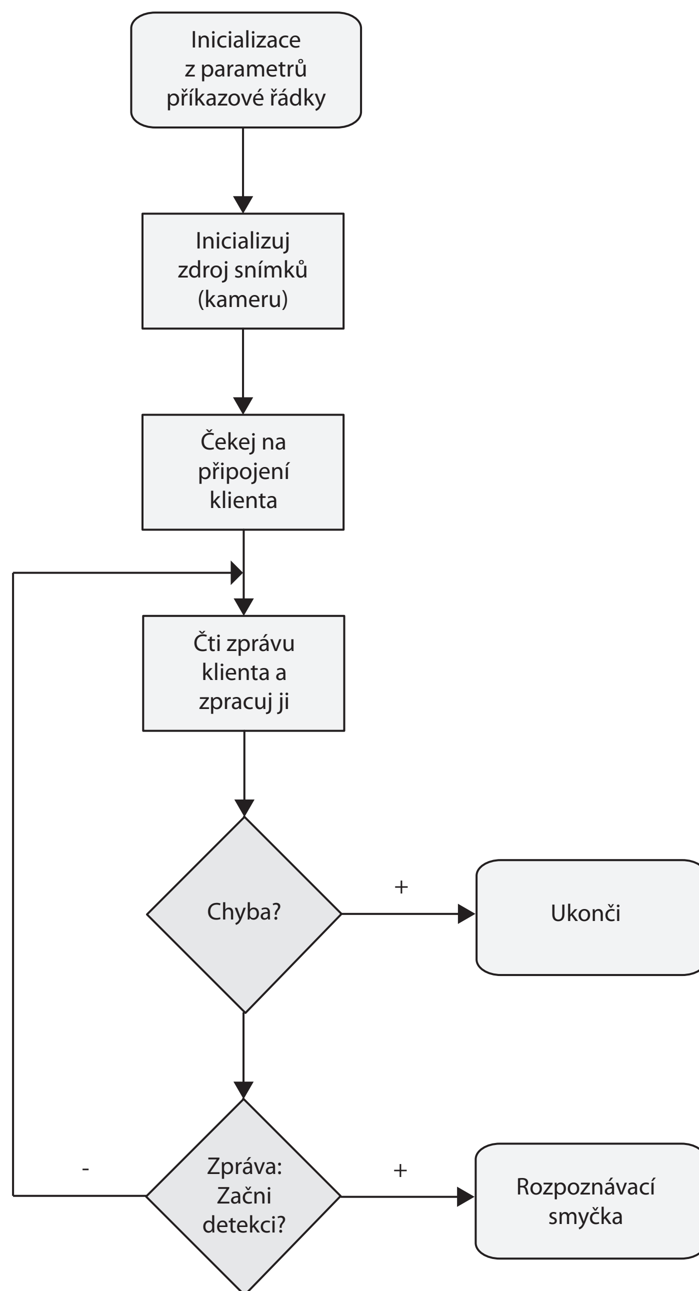
Obrázek 3.7: Formát těla zprávy, ve které je klientovi zaslán snímek (prahovaný i standardní) získaný z kamery.

3.3 Rozpoznávací modul

3.3.1 Celkové schéma činnosti

Rozpoznávací modul funguje jako samostatně spustitelná aplikace. Po spuštění se zpracují argumenty příkazové řádky a vyhodnotí se jejich validita. V případě, že jsou korektní, začne rozpoznávací modul vykonávat svoji činnost dle schématu, který je ilustrován na obrázku 3.8.

Po načtení argumentů příkazové řádky (viz podsekce 3.3.1) se inicializuje zdroj snímků (kamera). Rozpoznávací modul pak inicializuje *sockety* a začne jako server naslouchat a čekat na připojení klienta. Po jeho připojení začne rozpoznávací modul vykonávat smyčku, ve které čte zprávy a požadavky klienta a reaguje na ně odpovídajícím způsobem (viz sekce 3.2). Nastane-li při čtení zprávy chyba nebo se klient odpojí, server ukončí svoji činnost. V případě obdržení zprávy požadující okamžitý start detekce, začne server rozpoznávat objekty ze snímku a odesílat klientovi rozpoznaný herní stav (viz sekce 3.3.2).



Obrázek 3.8: Schéma činnosti rozpoznávacího modulu.

Argumenty příkazové řádky

Program s rozpoznávacím modulem čte argumenty příkazové řádky ve formátu:

`VisionModule.exe [`*port*`]`

Jediným argumentem příkazové řádky je číslo portu, na kterém má server naslouchat a čekat na připojení klienta.

Zdroje snímků

Rozpoznávací modul umožňuje načítat snímky ze dvou zdrojů – kamery a video záznamu. Video záznam však slouží pouze k ladicím účelům a standardně se bere jako zdroj snímků kamera. Během inicializace kamery se otevře souborový dialog, který od uživatele vyžaduje zadání konfiguračního souboru s nastavením kamery. Ten je na PC určeném pro robotický fotbal standardně uložen v souboru:

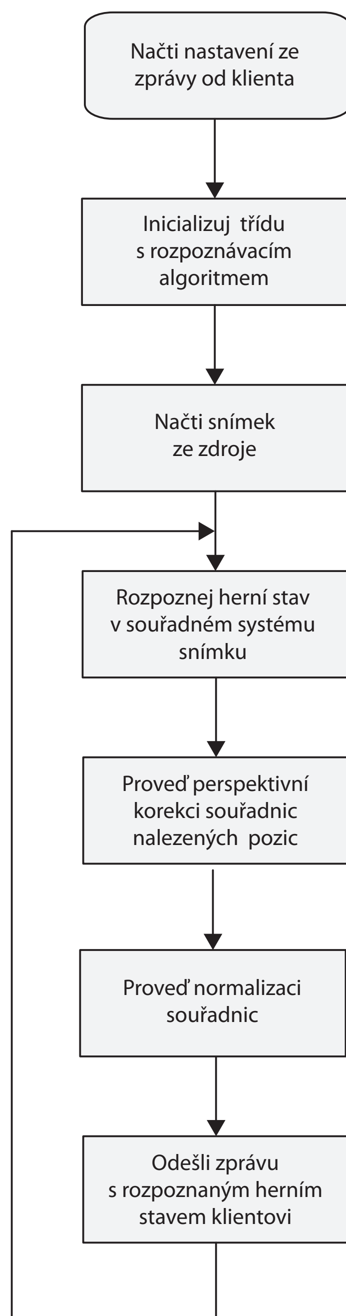
`C:\Robofotbal\kamera.ini`

Rozpoznávací smyčka

Před spuštěním detekční smyčky obdrží rozpoznávací modul od klienta zprávu s požadovaným nastavením, které uživatel specifikoval grafickém uživatelském rozhraní v klientské aplikaci. Tato nastavení pak ovlivňují chod rozpoznávacího algoritmu.

Na obrázku 3.9 je znázorněno schéma rozpoznávací smyčky. Před začátkem smyčky rozpoznávací modul zvolí druh algoritmu, který se k rozpoznávání použije (modul lze v budoucnu snadno rozšířit o další zcela odlišné algoritmy, experimentováno bylo například s algoritmem akcelerovaným na GPU), a předá mu nastavení, která jsou nutná pro chod rozpoznávacího algoritmu.

Úkolem rozpoznávacího algoritmu je zjistit pozice a natočení objektů ve snímku. Rozpoznané pozice tedy po detekci ještě nejsou v normalizované podobě (v intervalu $\langle 0, 1 \rangle$), ale závisí na rozlišení snímku, a nejsou určeny



Obrázek 3.9: Schéma rozpoznávací smyčky.

relativně vůči pozici stolu. Tento problém řeší následující fáze rozpoznávací smyčky, která nejprve provede perspektivní korekci nalezených pozic štítků robotů (pozice štítku ve snímku neodpovídá pozici podstavy robota vlivem perspektivního zkreslení) a následně provede normalizaci pozic vzhledem k rozměrům a středu hřiště. Výsledný herní stav po perspektivní korekci a normalizaci souřadnic je pak odeslán klientovi, který jej pak předá ostatním modulům. Celý postup se opakuje po načtení dalšího snímku ze zdroje.

3.3.2 Algoritmus rozpoznávání

Algoritmus rozpoznávání řeší tři základní úkoly – rozpoznání pozice míčku, nalezení pozic a natočení všech robotů našeho týmu a nalezení pozic a natočení všech robotů soupeřova týmu. Tyto úlohy na sobě nezávisí, takže je možné je vykonávat paralelně.

Vyhledávání kontur barevných objektů

Jedním ze základních problémů, které je nutné pro nalezení míčku a robotů obou týmů vyřešit, je nalezení ploch ve snímku s určitou barvou (žlutou, modrou nebo oranžovou). K vyhledávání barevných ploch byl využit převod snímku do barevného prostoru HSV, který je pro vymezení odstínu barvy vhodnější než prostor RGB, a následně byly oblasti hledané barvy určeny binárním prahováním. Prahování je řešeno tak, že se definuje *minimální* (dané vektorem $[minH, minS, minV]$) a *maximální* složky HSV barvy (dané vektorem $[maxH, maxS, maxV]$), které definují výseč uvnitř barevného prostoru HSV, ve které se nachází všechny přijatelné barvy. Prahováním se tedy zdrojový barevný obrázek v prostoru HSV (označeno *src*) převede na jednokanálový obrázek (*dst*) využitím vzorců 3.1.

$$
\begin{gathered}
srcH = src(x, y).H \\
srcS = src(x, y).S \\
srcV = src(x, y).V
\end{gathered}
$$

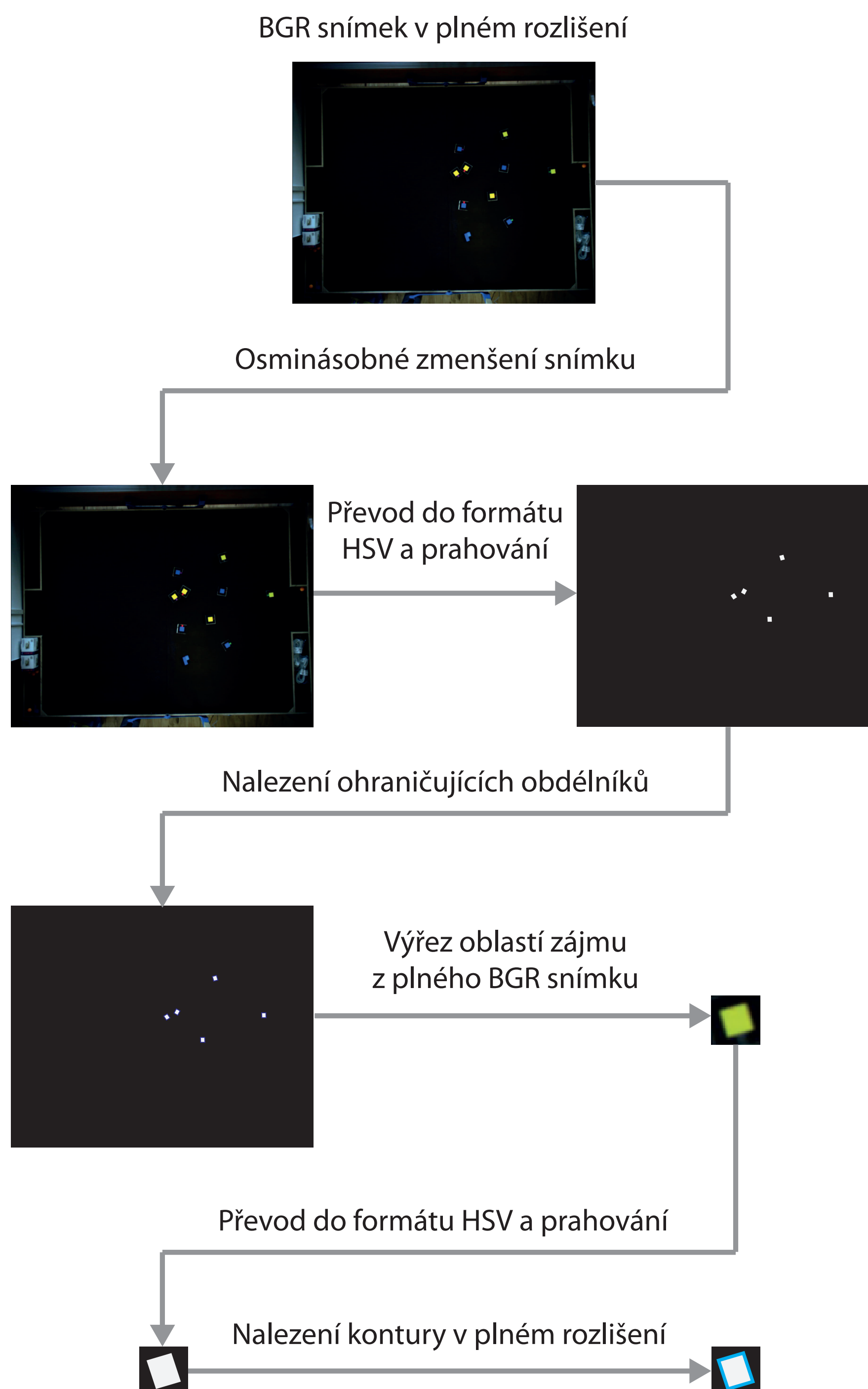
$$
dst\,(x, y) = \begin{cases} 255 & \text{když} \quad srcH \geq minH,\ srcS \geq minS,\ srcV \geq minV, \\ & \qquad\quad srcH \leq maxH,\ srcS \leq maxS \text{ a } srcV \leq maxV \\ 0 & \text{jinak} \end{cases} \tag{3.1}
$$

Takto prahovaný snímek je již možné použít k určení kontur nalezených barevných ploch. Z hlediska rychlosti však tento postup není optimální, neboť procházení snímku v plném rozlišení je časově příliš náročné. Snímek kamery má rozlišení 1280 × 1024 pixelů, šířka stolu i s brankovištěm je 2,5 metru a výška stolu je 1,8 metru. Kamera je umístěna ve výšce 2,5 metru nad hracím stolem. Při těchto údajích 1 pixel na hřišti zhruba odpovídá čtverci o hraně 2 mm. Všechny hledané objekty – míček i plochy s týmovou barvou robotů – musejí mít velikost alespoň 35 mm.

Z toho vyplývá, že hledat kontury je teoreticky možné až v $17,5\times$ zmenšeném snímku. Vzhledem k *Nyquist-Shannonově vzorkovacímu teorému* však musí být vzorkovací frekvence bodů obrazu dvojnásobná než je nejvyšší frekvence v něm obsažená, aby byl obraz korektně navzorkován. Jinými slovy, krok, se kterým se obraz prochází, musí být zmenšen na polovinu, aby nějaký objekt nebyl „přeskočen“. Obraz tedy pro dosažení korektního vzorkování smí být zmenšen pouze osminásobně.

Vzhledem k tomu, že převod snímku z barevného prostoru RGB do prostoru HSV je časově náročná operace a v plném rozlišení trvá několik milisekund, je RGB obraz nejprve osminásobně zmenšen a teprve poté je převeden do barevného prostoru HSV. V tomto zmenšeném HSV obrazu jsou prahováním na základě minimální a maximální povolené HSV barvy určeny hledané barevné plochy, u nichž jsou pak nalezeny jejich obrysy. Tyto obrysy jsou však taktéž osminásobně zmenšené, a je proto třeba je vyhledat znovu v plném rozlišení. Nejprve se určí těsně ohraničující obdélník kolem nalezené kontury ve zmenšeném obraze. Tento obdélník se osminásobně zvětší a umístí do odpovídajícího místa ve snímku v plném rozlišení. Tato oblast se rozšíří o 4 pixely, aby nebyl žádný barevný bod oříznutím vynechán. Tato obdélníková oblast RGB snímku v plném rozlišení se převede do barevného prostoru HSV, znovu se provede prahování a naleznou se výsledné kontury, tentokrát již v plném rozlišení.

Tímto postupem je možné výrazně snížit časovou náročnost prohledávání obrazu, a přesto naleznout obrysy hledaných barevných ploch v plném rozlišení. Postup ilustruje schéma 3.10.



Obrázek 3.10: Schéma rychlého vyhledávání kontur barevných oblastí ve snímku.

Detekce míčku

Míček se detekuje vyhledáváním obrysů oranžových ploch algoritmem, který byl popsán v předchozí podsekci. Minimální a maximální hodnoty v prostoru HSV pro oranžovou barvu definuje uživatel v klientské aplikaci a rozpoznávací modul pak čte tyto hodnoty z předaného nastavení.

Nalezené kontury se filtrují podle velikosti – zamítají se všechny souvislé komponenty, jejichž šířka nebo výška je menší než 10 pixelů. V takovém případě se totiž s největší pravděpodobností jedná o šum. První nalezená souvislá komponenta větší než tyto rozměry je považována za míček. Kolem nalezené kontury se najde minimální ohraničující kružnice (tj. ta, která má nejmenší poloměr a všechny body obrysu do ní spadají). Střed této kružnice je pak považován za střed (pozici) míčku.

Detekce robotů našeho a soupeřova týmu

Detekce robotů našeho i soupeřova týmu se řeší téměř stejným algoritmem, a nebude proto postup popisován samostatně pro každý tým. Jediný rozdíl při rozpoznávání spočívá v tom, že u robotů našeho týmu je potřeba určit i jejich identifikaci – tedy přiřadit každému z nalezených robotů jeho identifikační číslo, které slouží k bezdrátové komunikaci s ním.

Klientská aplikace zasílá rozpoznávacímu modulu nastavení se všemi potřebnými údaji pro rozpoznání robotů obou týmů – minimální a maximální HSV hodnoty pro modrou a žlutou barvu, dále je k dispozici údaj o tom, který tým má modrou a který má žlutou barvu. V nastavení je u každého robota taktéž k dispozici obrázek jeho štítku v základním natočení vyextrahovaný přímo z reálných dat kamery.

Rozpoznávací algoritmus pro každý tým při prvním spuštění projde obrázky všech identifikačních štítků robotů v týmu a provede jejich analýzu. Následně vyhledá ve snímku všechny kontury oblastí týmové barvy, která danému týmu byla přiřazena. Z kontur odvodí pozici středu štítku a jeho natočení (tj. pozici a natočení robota). Trackovacím algoritmem dále zajišťuje kontinuitu v očíslování robotů v čase. Jedná-li se o náš tým, je očíslování robotů určeno z dalších barevných značek na štítku.

Rozpoznané pozice štítků jsou určeny souřadnicemi ve zpracovávaném snímku. Vzhledem k tomu, že štítky se nachází v jiné výšce, než je rovina

hřiště, je třeba provést perspektivní korekci nalezených pozic, aby se všechny pozice promítly do roviny podlahy hřiště. Upravené pozice jsou na závěr normalizovány a výsledný herní stav je odeslán klientovi přes sockety.

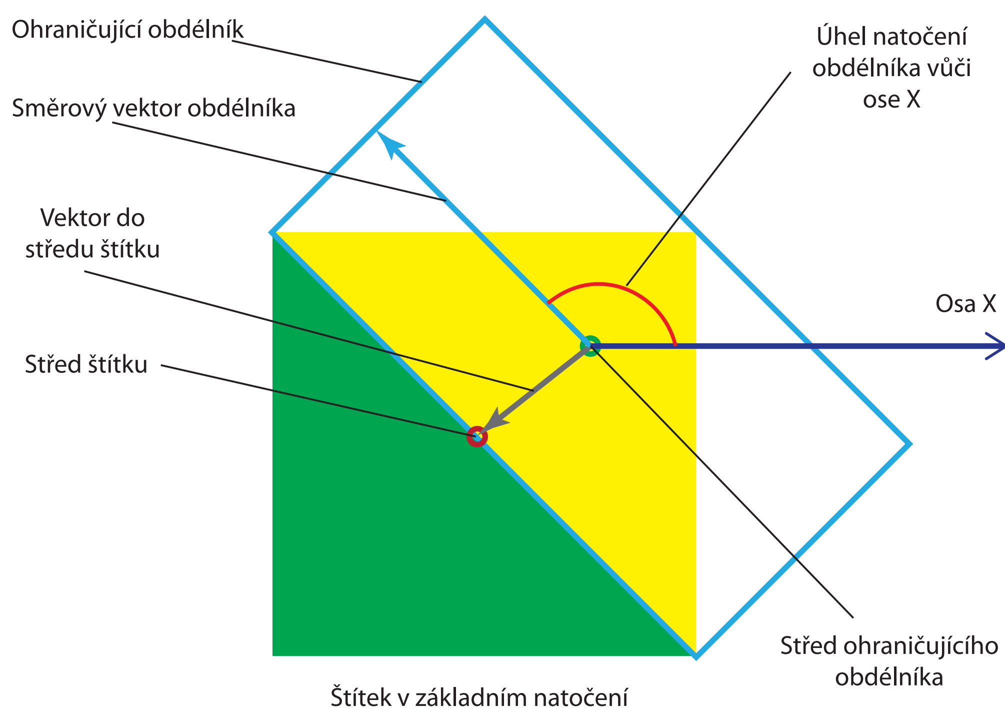
Analýza štítku

Pro odvození pozice a natočení štítku robota je nejprve nutné vytvořit jeho vhodný popis. V nastavení se nachází u každého robota obrázek s jeho štítkem. Cílem analýzy obrázku štítku je najít jeho popis, který umožní odvodit natočení a střed štítku, je-li dán pouze obrys oblasti s jeho týmovou barvou. Při rozpoznávání algoritmus vychází pouze z nalezené množiny obrysů oblastí týmové (modré nebo žluté) barvy.

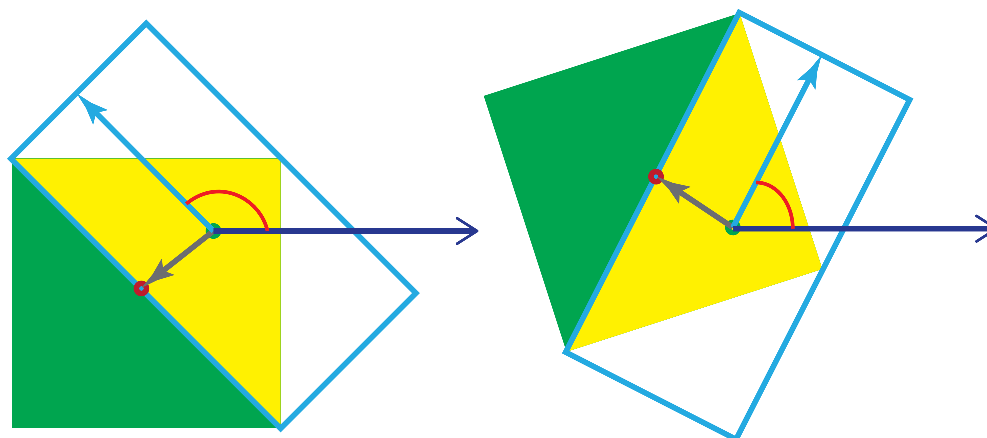
Pro lepší názornost postupu při analýze štítku jsou klíčové pojmy ilustrovány na obrázku 3.11. Během analýzy se vychází z obrázku štítku v tzv. *základním natočení*, což znamená, že přední strana štítku (tj. ta, ve směru které robot jezdí dopředu) míří doprava, tedy ve směru osy X, a štítek má proto úhel natočení nulový (vychází se z úhlů a os na jednotkové kružnici).

V obrázku štítku v základním tvaru se určí prahováním oblast týmové barvy (na obrázku 3.11 je týmová barva žlutá) a kolem ní se najde minimální ohraničující obdélník. Poté se určí směrový vektor tohoto obdélníka, který míří vždy ve směru rovnoběžném s delší stranou obdélníka. Existuje však více možností volby tohoto vektoru (u obdélníka se úly natočení těchto vektorů liší o 180 stupňů), a proto je vždy vybrán směrový vektor, který má nejmenší úhel na jednotkové kružnici. Pro obdélník existují 2 možné směrové vektory, pro čtverec existují čtyři možné směrové vektory. Tento úhel bude dále označován jako *základní úhel natočení ohraničujícího obdélníka.*

Střed ohraničujícího obdélníka oblasti s týmovou barvou však nemusí nutně odpovídat středu štítku robota, který určuje jeho skutečnou pozici. Je proto vypočten a uložen vektor ze středu nalezeného ohraničujícího obdélníka do středu štítku (analyzovaného obrázku). Tento vektor je tedy vyjádřen v pixelech a lze z něj správně vyvodit střed štítku robota pouze v případě, že byl analyzovaný obrázek štítku extrahován přímo ze snímku kamery v plném rozlišení. V klientské aplikaci je tedy nutné štítek extrahovat přímo ze snímku a nenačítat zmenšené, zvětšené či jinak upravené verze štítků, má-li algoritmus správně fungovat, protože by docházelo ke změně velikosti tohoto vektoru a tím by se odvodila nesprávná pozice středu robota.



Obrázek 3.11: Hledané veličiny při analýze obrázku identifikačního štítku.



Obrázek 3.12: Natočení robota je dáno rozdílem základních úhlů (červená křivka) nalezeného ohraničujícího obdélníka (vpravo) a ohraničujícího obdélníka ve štítku v základním natočení (vlevo).

Odvození pozice a natočení štítku

Natočení úhlu robota lze spočítat z natočení ohraničujícího obdélníka nalezené oblasti s týmovou barvou tak, že porovnáme jejich *základní úhly natočení*. Úhel natočení robota je pak rozdíl základního úhlu natočení nalezeného ohraničujícího obdélníka a základního úhlu natočení ohraničujícího obdélníka oblasti týmové barvy ve štítku v základním natočení (viz obrázek 3.12.

V natočeném štítku je natočen i vektor mířící ze středu ohraničujícího obdélníka do středu štítku. Je-li známo celkové natočení robota, stačí provést rotaci vektoru mířícího do středu získaného analýzou šablony o úhel natočení robota. Tento natočený vektor je pak přičten ke středu ohraničujícího obdélníka nalezené kontury a tím je získán střed štítku, takže i pozice robota ve snímku.

Identifikace robotů

Identifikace robotů je nutná k tomu, aby bylo možné zjistit, k jakému robotovi nalezená oblast týmové barvy patří. Není-li přiřazení známo, nelze provádět

ani odvození pozice a natočení štítku, protože není zřejmé, z jaké šablony obrázku štítku má algoritmus vycházet.

Úkolem identifikačního algoritmu je tedy přiřadit každému štítku nejvýše jednu konturu (resp. její minimální ohraničující obdélník). Tento problém je řešen tak, že se sestrojí *matice možných přiřazení.* Řádky matice určují číslo robota a sloupce index kontury v seznamu všech nalezených kontur oblastí týmové barvy. Hodnota v matici na řádku i a ve sloupci j pak nabývá hodnoty:

- 1 – pokud je možné, aby ohraničující obdélník (kontura) j patřil štítku robota číslo i.
- 0 – pokud ohraničující obdélník (kontura) j za žádných okolností nemůže patřit robotu číslo i.

Na počátku jsou všechny hodnoty v matici rovny jedné, protože není známo, která kontura patří kterému robotovi. V následující fázi identifikační algoritmus může provést několik *diskriminačních testů*, které rozhodnou, zda určitá přiřazení nejsou možná.

Nejdůležitějším testem je test velikosti nalezených ohraničujících obdélníků. Každý ohraničující obdélník je porovnán s ohraničujícím obdélníkem v šabloně každého robota. Liší-li se jejich výška nebo šířka o více než přednastavený počet pixelů (tato mez byla nastavena na hodnotu 7 pixelů), je přiřazení dané kontury robotovi s testovanou šablonou zamítnuto (do matice je na odpovídající místo zapsána hodnota nula).

Během vývoje bylo ještě experimentováno s použitím dalších diskriminačních testů, například na základě tvaru kontury definované tzv. *momenty invariance* (viz [18]), ale nakonec nebyly použity kvůli malé odolnosti vůči šumu.

Matice možných přiřazení po sérii diskriminačních testů může vypadat například jako tabulka 3.2. Tato konkrétní matice určuje například, že prvnímu robotovi lze přiřadit nalezenou konturu v seznamu na pozici 2, 3, 4 a 5 – není tedy ještě zřejmé, která kontura tomuto robotovi patří. Naopak pátému robotovi lze přiřadit pouze kontura číslo 3.

Lze-li robotovi přiřadit pouze jednu z nalezených kontur, přiřadíme ji, čímž ji logicky nebude možné přiřadit jinému robotovi. Proto se řádek i

0	1	1	1	1	1	0
0	1	1	0	1	0	0
0	1	1	1	0	0	0
0	1	1	0	0	0	0
0	0	1	0	0	0	0

Tabulka 3.2: Ukázka *matice možných přiřazení.* Každý z pěti řádků představuje jednoho robota a každý sloupec představuje nalezenou konturu týmové barvy (těch může být nalezeno víc než je robotů vlivem šumu). Hodnota 1 znamená, že přiřazení je možné, 0 přiřazení zamítá.

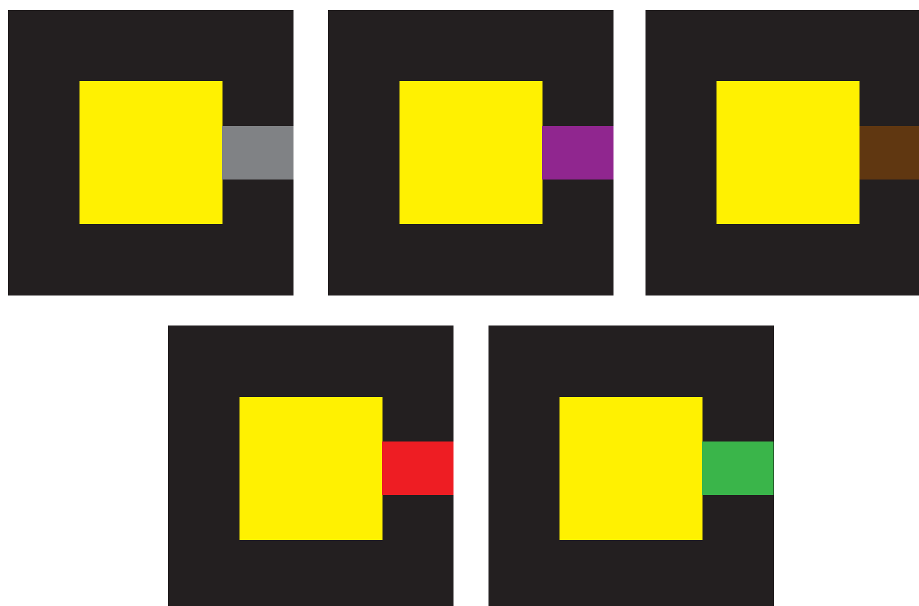
0	1	**0**	1	1	1	0
0	1	**0**	0	1	0	0
0	1	**0**	1	0	0	0
0	1	**0**	0	0	0	0
0	**0**	**0**	**0**	**0**	**0**	**0**

Tabulka 3.3: Ukázka matice možných přiřazení z tabulky 3.2 po přiřazení kontury číslo 3 robotovi 5. Všechny ostatní hodnoty v řádku a sloupci s touto hodnotou se vynulují a algoritmus znovu začne hledat v matici jednoznačné přiřazení.

sloupec dané tímto přiřazením vynulují. Tím může vzniknout další řádek, který obsahuje pouze jedno možné přiřazení, které se opět provede a tabulka se vynuluje.

V případě, že některým robotům lze přiřadit několik nalezených kontur, je třeba o jejich přiřazení rozhodnout *trackováním.* Tento přístup vychází z předpokladu, že u dvou po sobě jdoucích snímcích se štítek robota pohne jen o malou vzdálenost. Takže nejpravděpodobnější přiřazení je přiřadit konturu s nejnižší Eukleidovskou vzdáleností středu od středu naposledy přiřazené kontury danému robotovi.

Tento předpoklad byl uplatněn i u úhlu natočení robota. Algoritmus dokáže rozpoznat pouze úhel natočení od 0 do 90-ti stupňů. Skutečný úhel však může být o 90, 180 nebo 270 stupňů větší. Algoritmus zkouší všechny možné varianty a zvolí takový úhel, který je k poslednímu rozpoznanému úhlu natočení robota nejbližší. Tím je zajištěno plynulé sledování úhlu natočení robota v celém rozsahu 0 až 360 stupňů.

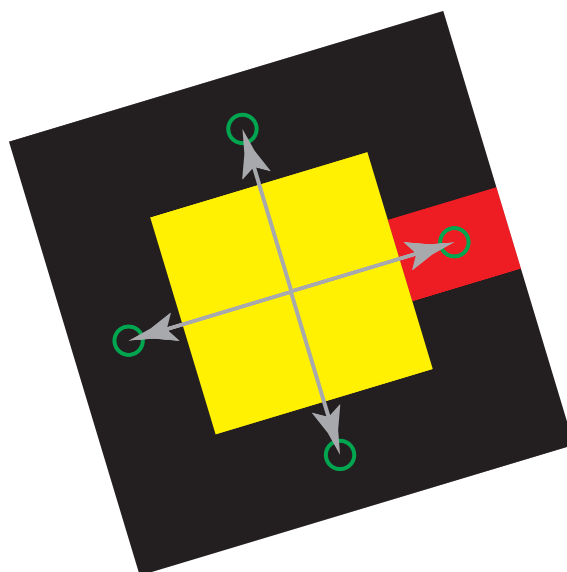


Obrázek 3.13: Vzhled štítků a identifikačních barev navržených pro roboty našeho týmu (ve variantě se žlutou týmovou barvou). Čtverec s identifikační barvou míří ve směru jízdy robota.

3.3.3 Identifikace našich robotů

Algoritmus identifikace našich robotů je stejný jako u robotů soupeře, navíc však každý robot ještě podstupuje diskriminativní test na sekundární barvu svého štítku, která je unikátní pro každého robota v našem týmu. Štítky obsahující plošky se sekundární barvou navržené pro roboty našeho týmu se nachází na obrázku 3.13.

Po prahování je nalezen ohraničující čtverec oblasti s naší týmovou barvou. Barevná identifikační ploška se nachází u hrany štítku ležící na přední straně robota. Nalezením této plošky tedy lze určit i směr jízdy robota. Ve všech čtyřech možných směrech natočení (tj. základní úhel natočení rozpoznaného čtverce $+0°$, $+90°$, $+180°$ a $+270°$) se vytvoří vektor, který zasahuje do středu plochy mezi čtvercem s týmovou barvou a okraji štítku robota. Z těchto 4 míst se přečte HSV odstín barvy. Pokud barva není černá,



Obrázek 3.14: Postup při hledání identifikační barevné plošky, která určuje identifikaci a směr natočení robota. Ze základního natočení a středu čtverce s týmovou barvou je určen směrový vektor do oblasti mezi týmovou barvou a okrajem robota a hledá se, ve kterém ze čtyř možných směrů se nachází identifikační ploška se sekundární barvou robota.

značí aktuální místo přední stranu robota, čímž se určí úhel natočení robota. Postup hledání barevné plošky ilustruje obrázek 3.14.

Barva nalezené plošky se pak klasifikuje podle vzdálenosti hodnot HSV složek od vzorové barvy u každého štítku robota našeho týmu. Robotovi, jehož štítek má barvu identifikační plošky nejbližší nalezené HSV barvě, je přiřazena pozice a natočení rozpoznané čtvercové plochy.

3.3.4 Perspektivní korekce a normalizace souřadnic

Vlivem použití kamery dochází k perspektivnímu zkreslení, což se projevuje tak, že ve snímku nejsou vidět pouze vrchní štítky robotů, ale i část jejich těla po stranách. Objekty v různých výškách se navíc zobrazují různě velké a na různých pozicích. Vlivem toho se štítky například mohou zobrazit částečně mimo hřiště, i přesto, že robot celou svojí podstavou stojí na hřišti (viz obrázek B.1).

Pozice kamery je známá, neboť dle pravidel musí být umístěna nad středem hřiště ve výšce 2,5 metru. Dále jsou známé rozměry stolu -– $2,2 \times 1,8$ metru a jeho velikost v pixelech díky kalibračnímu obdélníku, který zasílá klientský modul. Označme okraje stolu v pixelech *left*, *right*, *top* a *bottom* odpovídající umístění levého, pravého, horního a dolního okraje hrací plochy v pixelech od osy Y, resp. osy X pro *top* a *bottom*. Šířka stolu *width* se pak určí jako vzdálenost pravé stěny stolu od pozice levé stěny ($right - left$) a výška stolu *height* se určí jako vzdálenost spodní stěny od horní stěny (*bottom* – *top*). Výška každého robota (označme jako *robotHeight*) je rovněž známa a předává se ve zprávě s nastavením. Tyto údaje stačí k tomu, aby bylo možné perspektivou zkreslený čtverec s pozicemi rohů P_1, P_2, P_3 a P_4 korigovat (viz obrázek 3.15).

Každý bod promítnutého čtverce se štítkem lze korigovat následujícím postupem:

- Souřadnice každého bodu P se převedou z pixelových souřadnic na odpovídající polohu P_m v trojrozměrném prostoru vyjádřenou v metrech relativně od levého horního rohu stolu:

$$P_m = \left[\frac{P_x - left}{width}, \frac{P_y - top}{height}, 0\right]$$

- Definuje se pozice kamery C_m nad středem hřiště v metrech:

$$C_m = \left[\frac{2.2}{2}, \frac{1.8}{2}, 2.5\right]$$

- Určí se vektor mířící z promítnutého bodu P_m směrem ke kameře:

$$V_m = C_m - P_m$$

- Posuneme promítnutý bod podél vektoru V_m tak, aby se jeho výška (ležící na ose Z) dostala na úroveň výšky štítku robota (konstanta *robotHeight*):
  - Provedeme normalizaci vektoru V_m:

$$V'_m = \frac{V_m}{||V_m||}$$

– Z normalizované složky Z vektoru V'_m zjistíme, kolikrát se musí V'_m zvětšit, abychom se dostali z podlahy hřiště na úroveň štítku robota ve výšce *robotHeight*:

$$scale = \frac{robotHeight}{V'_{m_Z}}$$

– Na základě koeficientu zvětšení *scale* se určí výsledná korigovaná pozice $P_{korekce}$ bodu v metrech:

$$P_{korekce} = P_m + scale \cdot V_m$$

– Pro účely vizualizace může být užitečné převést korigovanou pozici v metrech do souřadnic ve snímku:

$$P' = \left[left + P_{korekce_X} \frac{width}{2.2},\ top + P_{korekce_Y} \frac{height}{1.8} \right]$$

Na základě dříve spočtené korigované pozice $P_{korekce}$ vyjádřené v metrech lze spočítat normalizovanou pozici $P_{normalizace}$, jejíž souřadnice X a Y se nachází v rozsahu $\langle -1, 1 \rangle$. Toho lze docílit vydělením korigovaných souřadnic rozměry stolu a zarovnáním kolem počátku souřadnic:

$$P_{normalizace} = \left[\frac{P_{korekceX}}{2.2} \cdot 2 - 1,\ \frac{P_{korekceY}}{1.8} \cdot 2 - 1 \right]$$

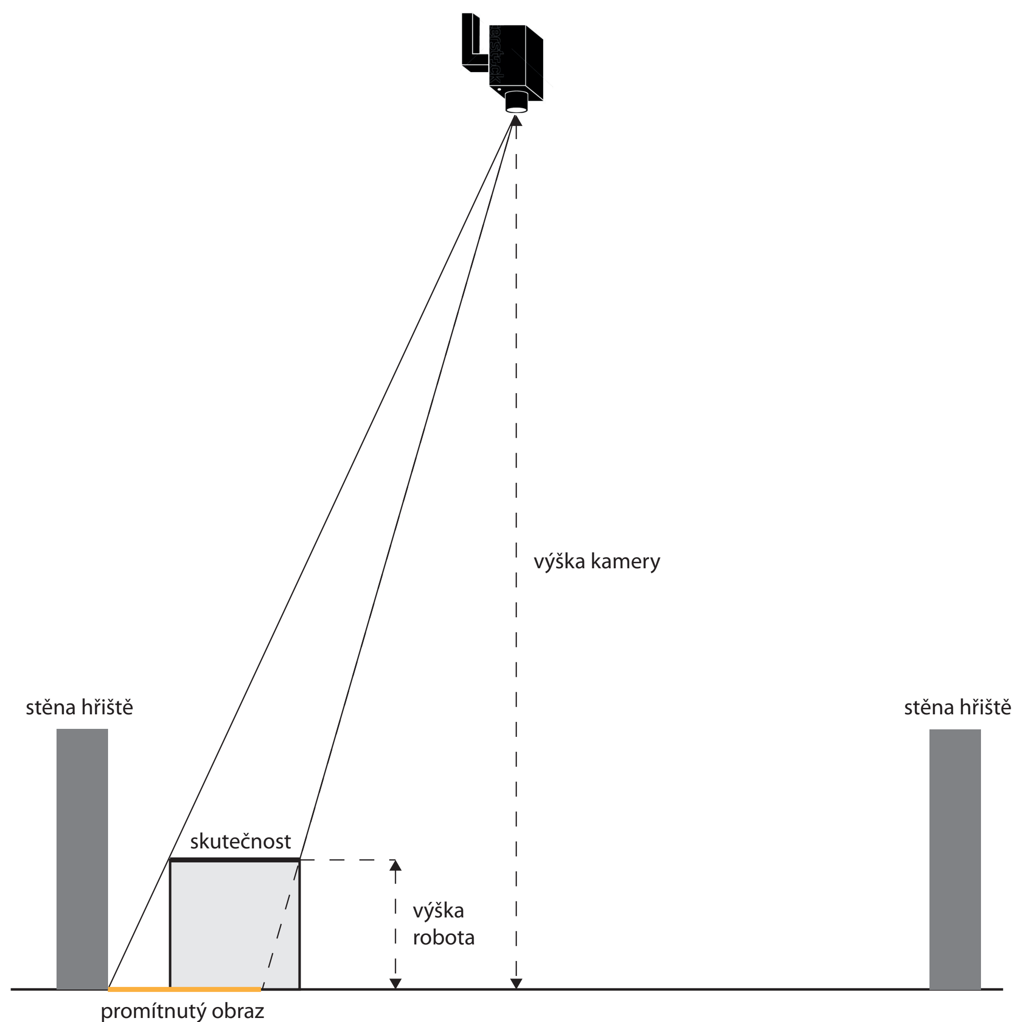
Normalizované pozice robotů a míčku pak určuje rozpoznaný herní stav, který je klientskému modulu ihned odeslán v odpovídající zprávě skrze sockety.

3.3.5 Implementace

Rozpoznávací modul byl implementován v jazyce C++ a Visual Studiu 2012 (viz sekce A.3). K vývoji byly použity knihovny OpenCV, Thread Building Blocks a knihovny pro ovládání a čtení snímků z kamery od výrobce kamery.

Získání snímku z kamery

Pro komunikaci s kamerou byla použita sada knihoven a hlavičkových souborů dodávaných v hlavním instalačním balíku ke kameře. Při implementaci



Obrázek 3.15: Perspektivní zkreslení ilustrované z bočního pohledu na hřiště. Skutečný štítek se při pohledu z kamery zobrazí těsně u stěny, i když robot je ve skutečnosti kousek od ní. Rozpoznávací algoritmus však změří pozici promítnutého obrazu. Známe-li polohu kamery a výšku robota, je možné zpětně odvodit souřadnice X a Y skutečné pozice robota v rovině podlahy hřiště.

byly využity části zdrojových kódů z příkladu *IDS Simple Acquire* dodávaného k programátorskému API kamery pro jazyk C++. Jména procedur a funkcí sloužící k nastavení a komunikaci s kamerou obsahují vždy prefix *is_*.

Nejprve byla zjištěna šířka a výška snímku v pixelech, která se předává ve struktuře *IS_SIZE_2D* jako návratová hodnota funkce *is_AOI*:

```
1 IS_SIZE_2D imageSize;
2 is_AOI(m_hCam, IS_AOI_IMAGE_GET_SIZE, (void*)&imageSize, sizeof(
      imageSize));
```

Proměnná *m_hCam* je *handle* pro ovládání aktuální instance kamery.

Dále se určí bitová hloubka pixelů ve snímku. Na základě této bitové hloubky a rozměrů snímku se alokuje odpovídající množství paměti, do které bude ukládán přijatý snímek z kamery:

```
1 // Alokujeme pamet pro snimek.
2 is_AllocImageMem( m_hCam, nAllocSizeX, nAllocSizeY,
      m_nBitsPerPixel, &m_pcImageMemory, &m_lMemoryId);
3 // Nastavime kamere alokovany blok jako cil pro ukladani snimku.
4 is_SetImageMem(m_hCam, m_pcImageMemory, m_lMemoryId );
```

Proměnné *nAllocSizeX* a *nAllocSizeY* udávají šířku a výšku snímku v pixelech. Proměnná *m_nBitsPerPixel* obsahuje bitovou hloubku pixelu ve snímku. Ukazatel *m_pcImageMemory* po dokončení těchto procedur ukazuje na první pixel snímku kamery.

Aby nebylo nutné stále kopírovat snímek z úseku paměti přiděleného knihovnou kamery do matice v OpenCV, byl matici *frame* sloužící k uchování snímku změněn ukazatel *data* na ukazatel *m_pcImageMemory*, čímž došlo k tomu, že OpenCV čte všechny hodnoty přímo z paměti se snímkem. Tím odpadá nutnost zbytečného kopírování dat, které by algoritmus zpomalovalo.

```
1 frame.data = (uchar*)m_pcImageMemory;
```

Načtení snímku do této paměti se pak provede voláním funkce *is_Freeze*, která za použití parametru *IS_WAIT* vyčká, než se se snímek kamery s handlem *m_hCam* nahraje do paměti alokované pro tento handle:

```
1 is_FreezeVideo(m_hCam, IS_WAIT);
```

Po dokončení této funkce je možné používat OpenCV matici v proměnné *frame* jako snímek kamery a předat jej rozpoznávacímu algoritmu.

Využití Thread Building Blocks

Knihovna Thread Building Blocks byla využita k měření času a k paralelizaci vyhledávání objektů ve snímku. TBB měří čas na základě časovačů s vysokou přesností. Uplynulý čas se měří jako rozdíl dvou instancí třídy *tbb::tick_count*. Statická metoda *tbb::tick_count::now* vrací aktuální čas. Doba výpočtu se pak určí změřením času před začátkem a po skončení výpočtu následujícím fragmentem kódu:

```
1 // Ulozi cas zacatku spusteni vypoctu.
2 tbb::tick_count startTime = tbb::tick_count::now();
3 // Spusti nejaky intenzivni vypocet.
4 pocitej();
5 // Precte aktualni cas.
6 tbb::tick_count endTime = tbb::tick_count::now();
7 // Vypise uplynuly cas v milisekundach.
8 printf("%.1f ms\n", (endTime - startTime).seconds() * 1000.0);
```

Hlavní využití knihovny TBB však bylo pro urychlení běhu algoritmu. Hledání míčku, robotů našeho týmu a robotů soupeřova týmu řeší algoritmus jako tři nezávislé úlohy, a je proto možné je paralelizovat. Pro každou úlohu byla vytvořena obalovací třída, jejímž jediným smyslem je dodržet konvence stanovené TBB na implementaci tříd, které se chovají jako TBB úloha (viz sekce 2.8):

```
1  // Trida definujici ulohu hledani nasich robotu.
2  class DetectOurTeamTask
3  {
4  protected:
5    StandardAlgorithm* alg;
6  public:
7    DetectOurTeamTask(StandardAlgorithm* alg) : alg(alg) {}
8      void operator()()
9      {
10       // Vykona nalezeni robotu naseho tymu.
11       alg->DetectOurTeam();
12     }
13 };
14
15 // Trida definujici ulohu hledani robotu soupere.
16 class DetectEnemyTeamTask
17 {
18 protected:
19   StandardAlgorithm* alg;
20 public:
21   DetectEnemyTeamTask(StandardAlgorithm* alg) : alg(alg) {}
```

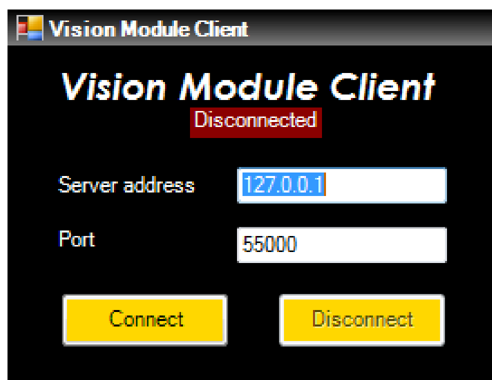
```
22      void operator()()
23      {
24        // Vykona nalezeni robotu souperova tymu.
25      alg->DetectEnemyTeam();
26      }
27 };
28
29 // Trida definujici ulohu hledani micku.
30 class DetectBallTask
31 {
32 protected:
33   StandardAlgorithm* alg;
34 public:
35   DetectBallTask(StandardAlgorithm* alg) : alg(alg) {}
36      void operator()()
37      {
38        // Vykona nalezeni micku.
39      alg->DetectBall();
40      }
41 };
```

Metody *DetectOurTeam*, *DetectEnemyTeam* a *DetectBall* třídy *StandardAlgorithm* se starají o nalezení robotů našeho týmu, resp. robotů soupeřova týmu a míčku. Třídy *DetectOurTeamTask*, *DetectEnemyTeamTask* a *DetectBallTask* tedy pouze volají metody ze standardní implementace algoritmu v přetíženém operátoru *()*, což knihovně TBB umožňuje s nimi zacházet jako s TBB úlohami.

Všechny úlohy se paralelně spustí a počká se na jejich dokončení následujícím fragmentem kódu:

```
1 tbb::task_group group;
2 group.run(DetectEnemyTeamTask(this));
3 group.run(DetectOurTeamTask(this));
4 group.run(DetectBallTask(this));
5 group.wait();
```

Kde *this* je instance třída *StandardAlgorithm*, která tyto tři úlohy spouští pro každý přijatý snímek. Všechny úlohy, které se mají vykonávat paralelně jsou umístěny do skupiny definované třídou *tbb::task_group*. Její metodou *run* jsou spuštěny paralelně a voláním metoda *wait* se vyčká, než dojde k nalezení míčku, robotů našeho týmu a robotů soupeřova týmu.



Obrázek 3.16: Po načtení klientského modulu řídicí platformou robotického fotbalu se zobrazí okno s údaji pro připojení k serveru. Nahoře je zobrazen stav připojení – *Connected*, je-li server připojen, a *Disconnected* v odpojeném stavu.

3.4 Klientský modul

Úkolem klientského modulu je integrovat výsledky získané rozpoznávacím modulem do řídicí softwarové platformy robotického fotbalu. Klient využívá grafického uživatelského rozhraní k nastavení parametrů rozpoznávacího modulu uživatelsky přívětivou formou. Dále umožňuje nastavené parametry automaticky ukládat a načítat z konfiguračního souboru *VisionModuleSettings.xml*, takže není třeba vizualizační modul nastavovat znovu při každém spuštění.

3.4.1 Grafické uživatelské rozhraní

Řídicí software nejprve načte knihovnu s klientským modulem, protože je definován v konfiguračním souboru řídicího software *config.xml*, ze souboru *RoboSoccer.Vision.Client.dll*, který vznikne překladem klientského modulu. Po startu řídicí platformy se inicializuje a zobrazí v okně s moduly celého řídicího software. Nejprve je k dispozici pouze okno s údaji a připojení k serveru, tedy jeho IP adresou a portem (viz obrázek 3.17). Po úspěšném připojení se zobrazí okno, které umožňuje měnit nastavení rozpoznávacího modulu.

Okno obsahuje několik záložek, které člení nastavení do logických celků:

- *Camera* – zobrazení dat z kamery a nastavení kalibračního obdélníka stolu.
- *Team colors* – nastavení hodnot pro prahování týmových barev (modré a žluté); navíc definuje, který tým má jakou barvu.
- *Ball* – nastavení míčku a prahovacích hodnot pro oranžovou barvu (barva míčku).
- *Our Team* – nastavení štítků a výšek jednotlivých robotů našeho týmu.
- *Enemy Team* – nastavení štítků a výšek jednotlivých robotů soupeřova týmu.
- *Server* – spuštění serveru.

Záložka Camera

Záložka *Camera* slouží primárně k zobrazení aktuálního snímku z kamery, který je odesílán ze serveru na vyžádání klienta. Dále vyznačuje *kalibrační obdélník*, což je část snímku z kamery, která obsahuje obdélníkovou část hřiště bez brankovišť. Na základě tohoto obdélníka pak rozpoznávací algoritmus převádí rozpoznané pozice ze souřadnic na snímku do normalizované pozice v intervalu $\langle -1, 1 \rangle$, se kterou pak pracuje řídicí jádro a herní strategie. Volba tohoto obdélníka tedy ovlivňuje přesnost rozpoznávání pozic. Obdélník by měl mít hrany tam, kde se ve snímku začínají zdvihat stěny hřiště (nikoliv tam, kde je vidět již horní plocha stěny, protože tato pozice je zkreslena perspektivou a nás zajímá umístění stěn ve výšce podlahy hřiště).

Kalibrační obdélník lze vybrat po stisknutí tlačítka *Calibrate*, které vyvolá zobrazení kalibračního okna, které umožňuje hýbat s jednotlivými body kalibračního obdélníka. Tyto body jsou očíslovány 0 až 3 a každá hodnota představuje jiný roh stolu:

- 0 – pravý horní roh stolu.
- 1 – levý horní roh stolu.
- 2 – levý dolní roh stolu.
- 3 – pravý dolní roh stolu.

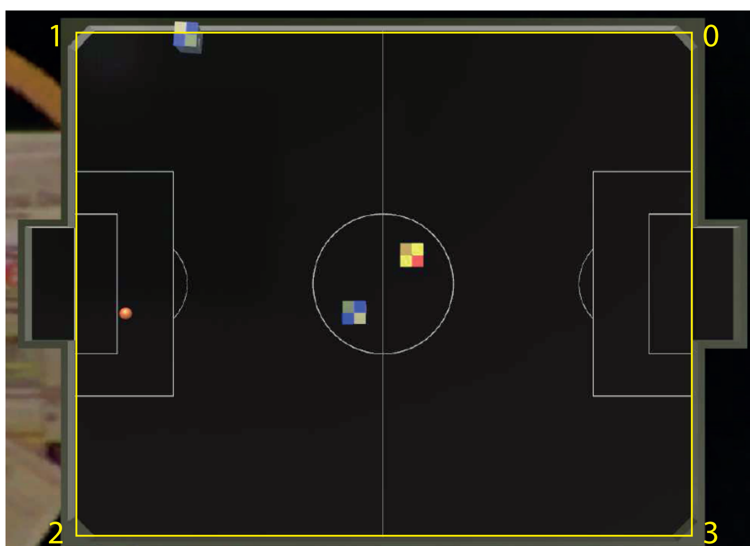
Obrázek 3.17: Záložka *Camera* v grafickém uživatelském rozhraní klientského modulu. Zobrazuje aktuální snímek z kamery a volitelně kalibrační obdélník stolu (zaškrtávací volba *Show calibration rectangle*).

Pořadí ilustruje obrázek 3.18. Pozici kalibračního obdélníka lze nejprve hrubě vyznačit tažením myši za stisku *pravého* tlačítka myši. Tažením bodů levým tlačítkem lze pak doladit jejich výslednou pozici.

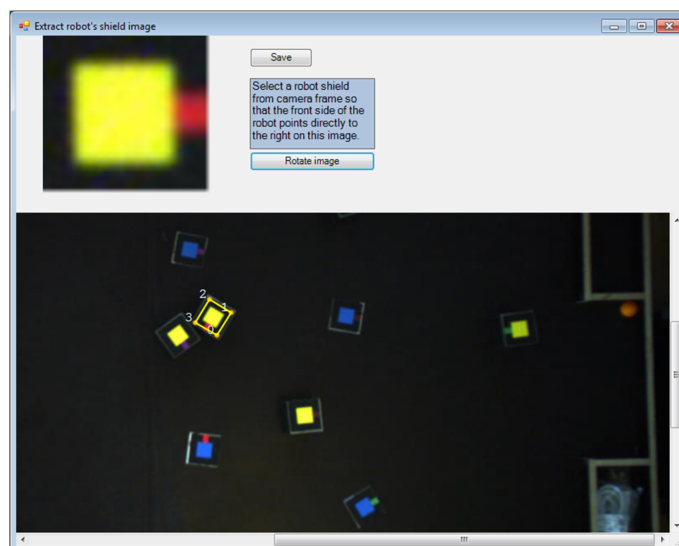
Extrakce štítku robota

Kliknutím na tlačítko *Extract robot shields* se otevře samostatné okno, které obsahuje snímek z kamery v plném rozlišení, ve kterém lze vybrat oblast se štítkem robota a uložit ji jako vzorový obrázek štítku do souboru, což přijde vhod při nastavování štítků jednotlivým robotům v obou týmech. Okno pro výběr štítku robota je ilustrováno na obrázku 3.19.

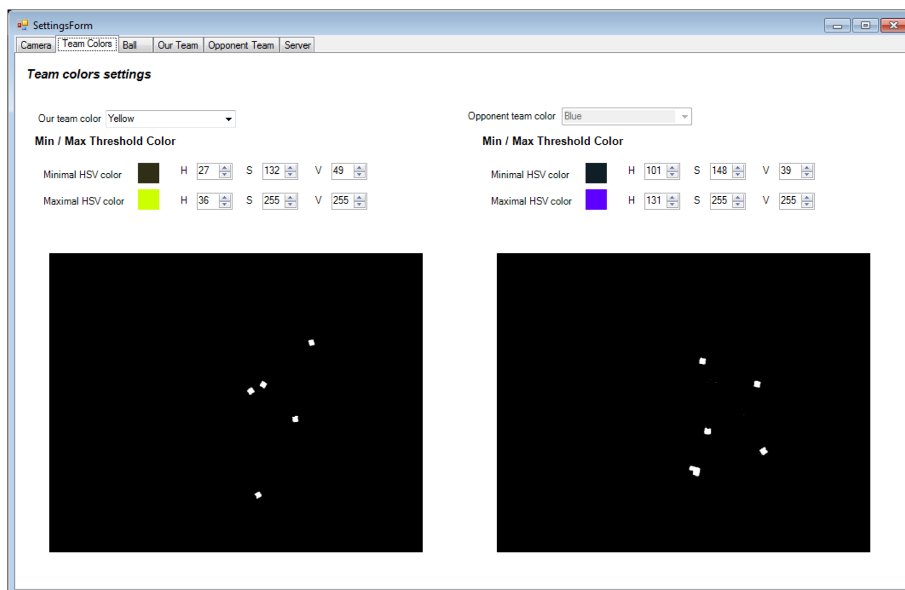
Podobně jako při volbě kalibračního obdélníka hřiště je možné nejprve tažením pravého tlačítka myši vybrat hrubou obdélníkovou část obrazu, v níž se štítek robota nachází, a později pozici rohů doladit tažením bodů levým tlačítkem myši. V levé horní části okna se ukazuje aktuální výseč štítku robota ze snímku. Stejně jako u kalibračního obdélníka stolu jsou body očíslované a nesou stejný význam. Vybraný štítek robota ze snímku musí být natočen



Obrázek 3.18: Ilustrace kalibrační obdélníka stolu. Pro co nejvyšší přesnost rozpoznávání by hrany obdélníka měly začínat tam, kde se ve snímku začínají zdvihat stěny hřiště od podlahy. Je zároveň nutné respektovat očíslování rohů tak, jak ilustruje obrázek.



Obrázek 3.19: Okno, ve kterém je možné sejmout štítek robota ze snímku kamery v plném rozlišení. Po určení rohů štítku robota ve snímku se v levé horní části okna ukáže vyextrahovaný obrázek štítku. Přední část štítku musí mířit vpravo. Štítek lze uložit do souboru stiskem tlačítka *Save*.



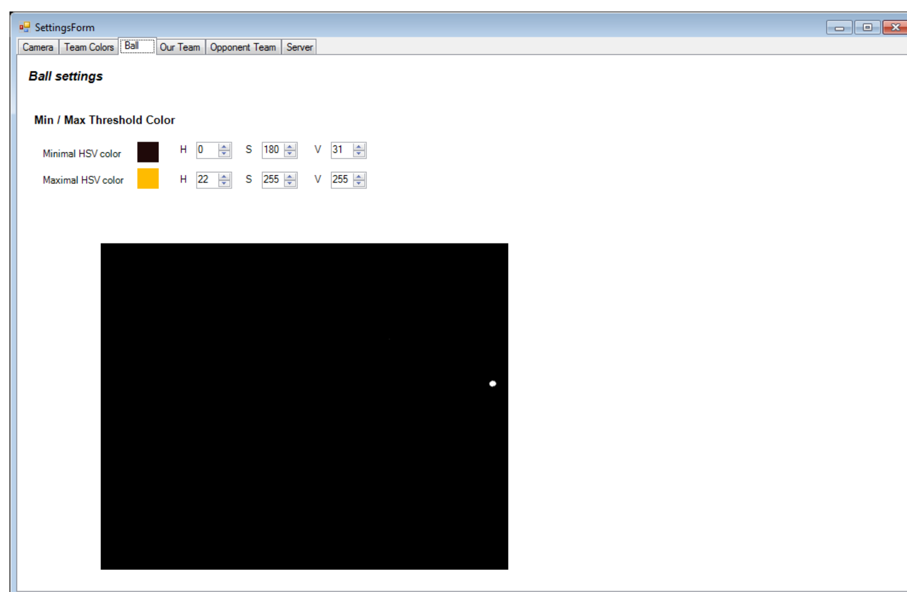
Obrázek 3.20: Záložka pro nastavení prahovacích HSV hodnot týmových barev (modré a žluté). V rolovacím seznamu *Our team color* lze vybrat, jaká týmová barva byla přidělena rozhodčím našemu týmu.

tak, aby jeho přední strana (tj. ta, ve směru které robot jezdí dopředu) mířila vpravo – tj. ve směru osy X (na jednotkové kružnici vyjadřuje nulový úhel). Je-li vybraný štítek vůči správnému natočení pouze pootočen o násobky 90°, stačí tlačítkem *Rotate image* postupně výřez otáčet, dokud přední část robota nemíří vpravo. Je-li extrahovaný štítek správně vybrán a natočen, je možné jej stiskem tlačítka *Save* uložit jako obrázek do souboru.

Záložka Team Colors

Záložka *Team Colors* slouží k nastavení HSV hodnot pro prahování týmových barev (modré a žluté) ze snímku kamery. Zároveň určuje, jakému týmu je přiřazena která barva (určuje rozhodčí před zápasem). Barvu našeho týmu lze zvolit v rolovacím seznamu *Our team color*, která je buď žlutá (volba *yellow*), nebo modrá (volba *blue*).

Záložka je rozdělena na 2 části. V levé části se nachází komponenta pro výběr minimálních a maximálních HSV prahovacích hodnot, které definují barvu našeho týmu. Posuvníky v řádku *Minimal HSV Color*, resp. *Maximal*



Obrázek 3.21: Záložka pro nastavení prahovacích HSV hodnot míčku (oranžové barvy).

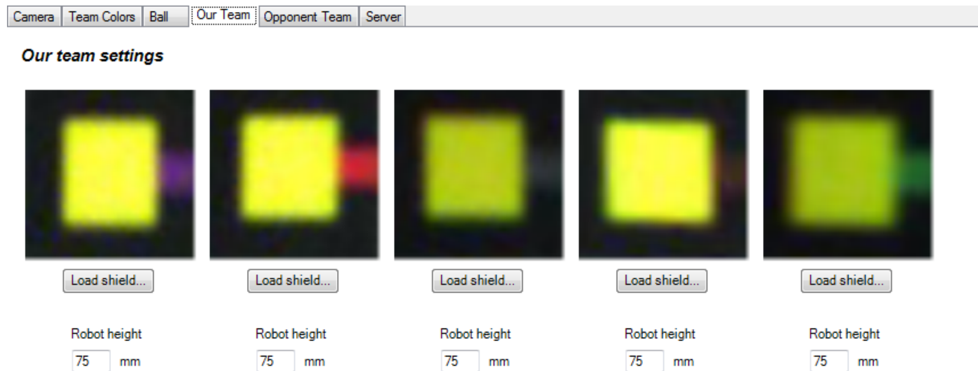
HSV Color určují minimální, resp. maximální hodnotu barvy v prostoru HSV, která vymezuje žlutou nebo modrou barvu. Pravá část okna funguje zcela analogicky, pouze vymezuje minimální a maximální hodnoty HSV prahů pro týmovou barvu soupeře. V obrázcích pod posuvníky se zobrazuje snímek vzniklý prahováním s těmito hodnotami (viz obrázek 3.4.1).

Záložka Ball

Záložka *Ball* slouží k nastavení HSV barev pro prahování míčku v obraze. Funguje zcela analogicky k záložce *Team Colors* – posuvníky vymezují minimální a maximální hodnoty HSV oranžové barvy míčku a dole se zobrazuje aktuální snímek vzniklý prahováním s těmito hodnotami. Záložka je zobrazena na obrázku 3.4.1.

Záložky Our Team a Opponent Team

V záložkách *Our Team* a *Opponent Team* lze nastavit identifikační štítky našeho, resp. soupeřova týmu. Stisknutím tlačítka *Load shield...* lze načíst



Obrázek 3.22: Záložky *Our Team* (na obrázku) a *Opponent Team* umožňují nastavení identifikačních štítků robotů našeho, resp. soupeřova týmu. Dále je každému z pětice robotů možné nastavit jeho výšku v milimetrech, aby bylo možné korektně vypočítat jeho pozici perspektivní korekcí.

identifikační štítek robota ze souboru, který vznikl v okně pro extrakci štítku robotů ze snímku kamery (viz podsekce 3.4.1). Dále je každému robotu možné přiřadit jeho výšku v milimetrech v poli *Robot height*. Tato hodnota slouží při rozpoznávání pro výpočet perspektivní korekce. Záložka *Our Team* je ilustrována na obrázku 3.4.1.

Záložka Server

Záložka *Server* slouží ke spuštění detekce serveru. Před započetím detekce však musí každý robot našeho i soupeřova týmu mít přiřazen obrázek s identifikačním štítkem, jinak není možné nastavení serveru odeslat. Po stisknutí tlačítka *Run detection* je serveru odeslána zpráva s celkovým nastavením a okno s grafickým uživatelským rozhraním s nastavením je zavřeno. Od této chvíle server detekuje pozice robotů a posílá klientskému modulu zprávy s rozpoznaným herním stavem.

3.4.2 Implementace

Obecné

Klientský modul je implementován v jazyce `C#`, aby jej bylo možné začlenit do řídicí platformy robotického fotbalu, která je na tomto jazyce postavena. Ke komunikaci se serverem využívá TCP sockety, aby byla zaručena spolehlivost a pořadí předávaných zpráv. Protokol UDP by totiž nepřinesl žádnou zásadní výhodu, ani z hlediska rychlosti, neboť by jak klientský modul, tak i rozpoznávací modul měly primárně běžet na stejném počítači, protože využití více strojů je při reálné hře zakázáno.

Klientský modul je překládán pro platformu *x86*, ve které na rozdíl od platformy *x64* funguje korektně návrhář grafického uživatelského rozhraní ve Visual Studiu. Modul nevyužívá žádné externí knihovny a pracuje pouze s třídami z *.NET Framework* verze 4, na kterém zároveň běží i celá platforma.

Všechna nastavení se ukládají a načítají do XML konfiguračního souboru *VisionModuleSettings.xml*, který vzniká automaticky *XML serializací* objektů v platformě *.NET*, takže není při přidání dodatečných nastavení měnit ostatní třídy aplikace a přidávat například metody pro načítání a ukládání nově přidaných nastavení, čímž je zajištěna snadná rozšiřitelnost v budoucnu.

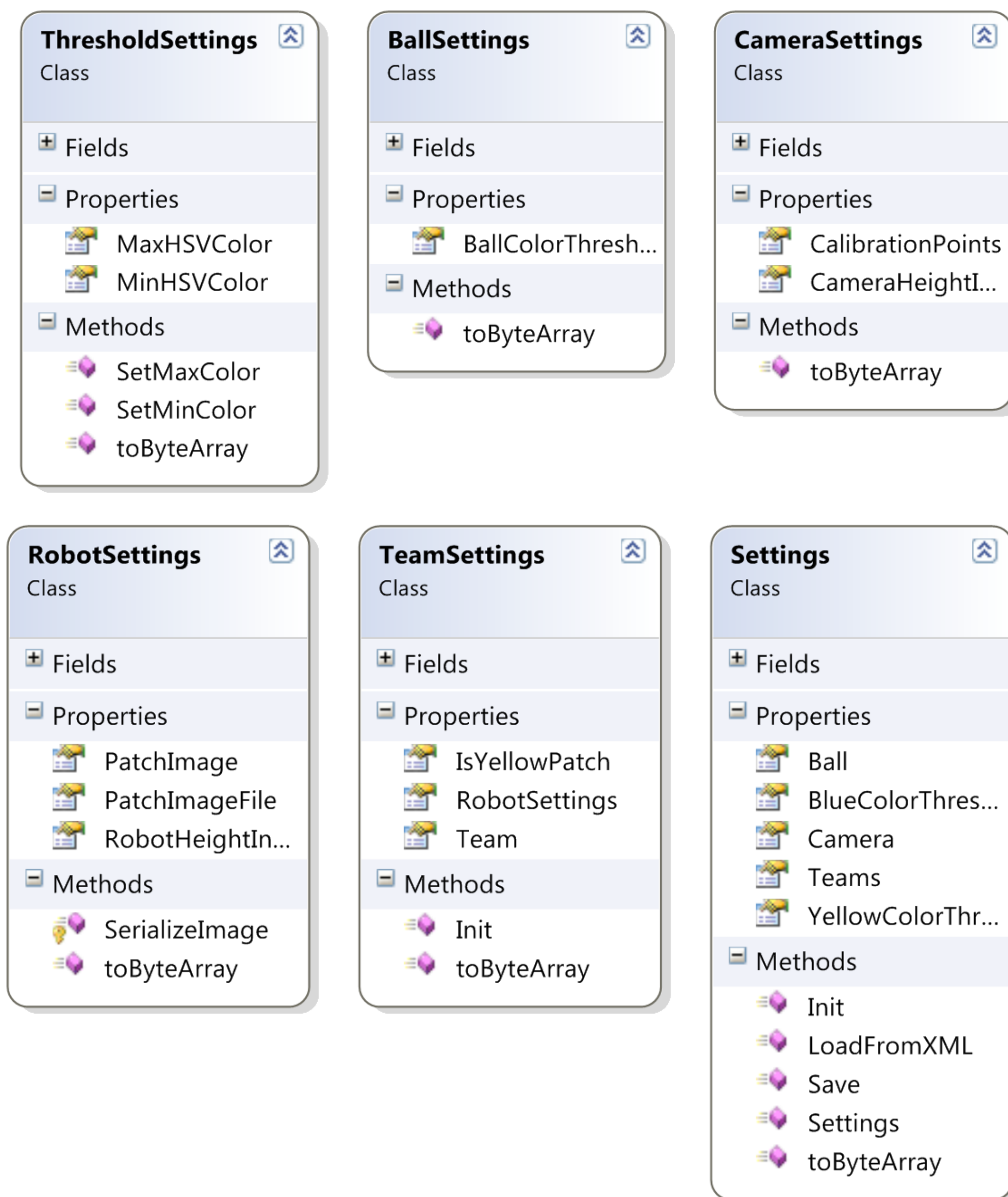
Třídy s nastavením

Třídy *ThresholdSettings*, *BallSettings*, *CameraSettings*, *RobotSettings*, *TeamSettings* a *Settings* slouží k uchování nastavení určeného pro rozpoznávací modul. Každá z nich obsahuje metodu *toByteArray*, která provádí nad daným nastavením serializaci do binárního proudu dat, aby je bylo možné odeslat skrze sockety. UML diagramy těchto tříd se nachází na obrázku 3.4.2.

Třída *ThresholdSettings* slouží k uchování hodnot pro prahování – tedy minimální HSV barvy (atribut *MinHSVColor*) a maximální HSV barvy (atribut *MaxHSVColor*).

Třída *BallSettings* uchovává nastavení míčku. U něj je nutné znát pouze prahovací hodnoty pro oranžovou barvu, které jsou uloženy v atributu *BallColorThreshold*, což je instance třídy *ThresholdSettings*.

Třída *CameraSettings* uchovává nastavení kamery – kalibrační obdélník



Obrázek 3.23: UML diagramy tříd s nastavením.

stolu a výšku kamery nad hřištěm v metrech. Kalibrační obdélník je uložen v atributu *CalibrationPoints* jako čtveřice bodů (instance třídy *System.Drawing.Point*) v pořadí, které je definované v podsekci 3.4.1. Výška umístění kamery v metrech je uložena v atributu *CameraHeightInMeters*.

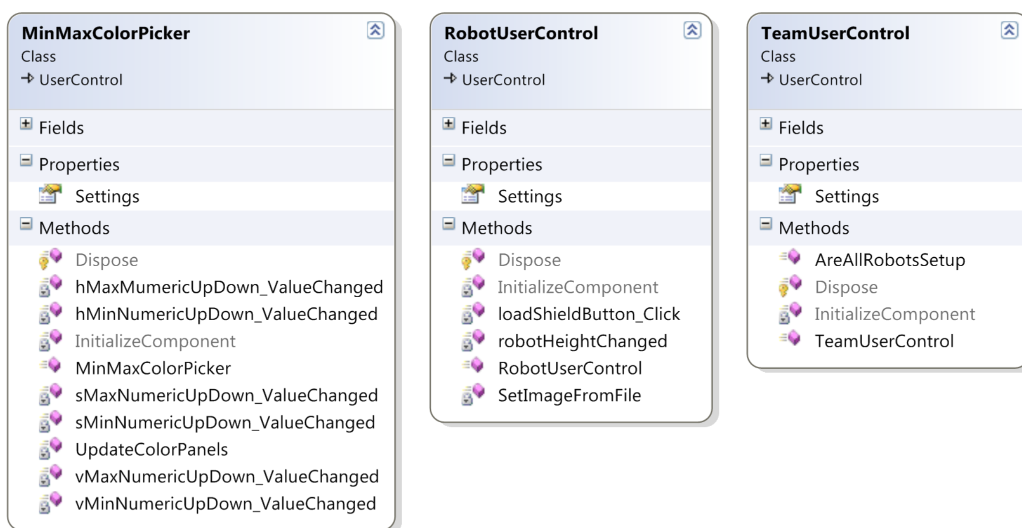
Třída *RobotSettings* uchovává nastavení jednoho robota v týmu. Atribut *PatchImageFile* určuje jméno souboru, ve kterém se nachází obrázek s identifikačním štítkem tohoto robota. Je-li cesta validní, do atributu *PatchImage* se načte odpovídající obrázek se štítkem jako instance třídy *Bitmap*, kterou je možné zobrazovat v grafickém uživatelském rozhraní.

Třída *TeamSettings* obsahuje nastavení jednoho týmu (našeho nebo soupeřova). Atribut *IsYellowPatch* je typu *bool* a značí, zda tento tým má přidělen jako týmovou barvu žlutou. Atribut *RobotSettings* je pole pěti instancí třídy *RobotSettings* tak, že každá položka v poli představuje nastavení jednoho robota v týmu. Atribut *Team* označuje, zda se jedná o náš nebo soupeřův tým (hodnota výčtu *TeamIdentificationEnum* definovaná ve jmenném prostoru jádra řídicí platformy – *RoboSoccer.Core*). Metoda *Init* inicializuje pole nastavení robotů. Z důvodů serializace tříd s nastavením to není možné provádět v konstruktoru.

Třída *Settings* sdružuje všechna dílčí nastavení ve svých atributech. Jedná se o celkové nastavení, které je odesláno serveru jako zpráva před zahájením procesu rozpoznávání. Atribut *Ball* obsahuje nastavení míčku (instance třídy *BallSettings*). Atributy *BlueColorThreshold* a *YellowColorThreshold* uchovávají nastavení prahovacích hodnot pro modrou, resp. žlutou barvu (jako instance třídy *ThresholdSettings*). Atribut *Camera* obsahuje nastavení kamery (instance třídy *CameraSettings*). Atribut *Teams* je pole obsahující dvě instance třídy *TeamSettings* – tedy nastavení našeho i soupeřova týmu.

Metoda *Init* provede inicializaci nastavení v případě, že se nezdaří načíst konfiguraci ze souboru (například proto, že ještě neexistuje – při prvním spuštění). Statická metoda *LoadFromXML* načte nastavení ze zadaného konfiguračního souboru a vrátí odpovídající instanci třídy *Settings*. Metoda *Save* uloží nastavení do konfiguračního souboru (typicky při zavření okna s nastavením v grafickém uživatelském rozhraní).

Metoda *toByteArray* serializuje celé nastavení do binárního proudu tak, že rekurzivně volá tuto metodu u svých atributů obsahující dílčí nastavení. Tím, že všechna nastavení zapisují do stejného proudu, se postupně složí tělo výsledné zprávy obsahující všechna nastavení. Server pak ve stejném pořadí



Obrázek 3.24: UML diagramy komponentových tříd *MinMaxColorPicker*, *RobotUserControl* a *TeamUserControl*.

čte dílčí nastavení z těla zprávy. Pro konkrétní formát zprávy s nastavením viz 3.2.

Komponentové třídy

Grafické uživatelské rozhraní obsahuje několik prvků, jejichž funkcionalita je stejná v několika záložkách – typicky jde o nastavování HSV hodnot pro prahování a zobrazování výsledného obrázku po prahování pod nimi. Z tohoto důvodu byly tyto celky nejprve vymodelovány samostatně jako potomci třídy *UserControl* a později byly přidány instance těchto komponent příslušných jednotlivých záložek.

Mezi tyto komponentové třídy patří třída *MinMaxColorPicker*, *RobotUserControl* a *TeamUserControl*. Jejich UML diagramy jsou zobrazeny na obrázku 3.4.2.

Všechny tyto komponenty v konstruktoru obdrží nastavení (které je po spuštění modulu přečteno z konfiguračního souboru) a na jeho základě inicializují prvky ve svém grafickém rozhraní (tj. hodnoty posuvníků, číselných polí, obrázky apod.). Zároveň je jakákoliv změna provedená v těchto komponentách ihned zapsána zpět do předané instance s nastavením, které je

sdíleno v celém klientském modulu, takže po ukončení aplikace se nastavení uloží s odpovídajícími hodnotami v grafickém uživatelském rozhraní.

Komponenta *MinMaxColorPicker* slouží k vybrání minimální a maximální HSV hodnoty pro prahování. V konstruktoru získává instanci třídy *ThresholdSettings*, na jejímž základě inicializuje komponenty ve svém grafickém rozhraní – převede uložené HSV hodnoty na RGB a obarví na jejich základě panel zobrazující vybranou minimální nebo maximální prahovací barvu. Zároveň načtené HSV hodnoty přiřadí posuvníkům pro hodnoty H, S a V.

Komponenta *RobotUserControl* zobrazuje a mění nastavení jednoho robota v týmu. V konstruktoru obdrží instanci třídy *RobotSettings*, na jejímž základě zjistí jméno souboru s obrázkem štítku tohoto robota, načte jej a zobrazí ve svém panelu. Dále načte výšku robota a upraví podle její hodnoty položku *Robot height* ve svém panelu.

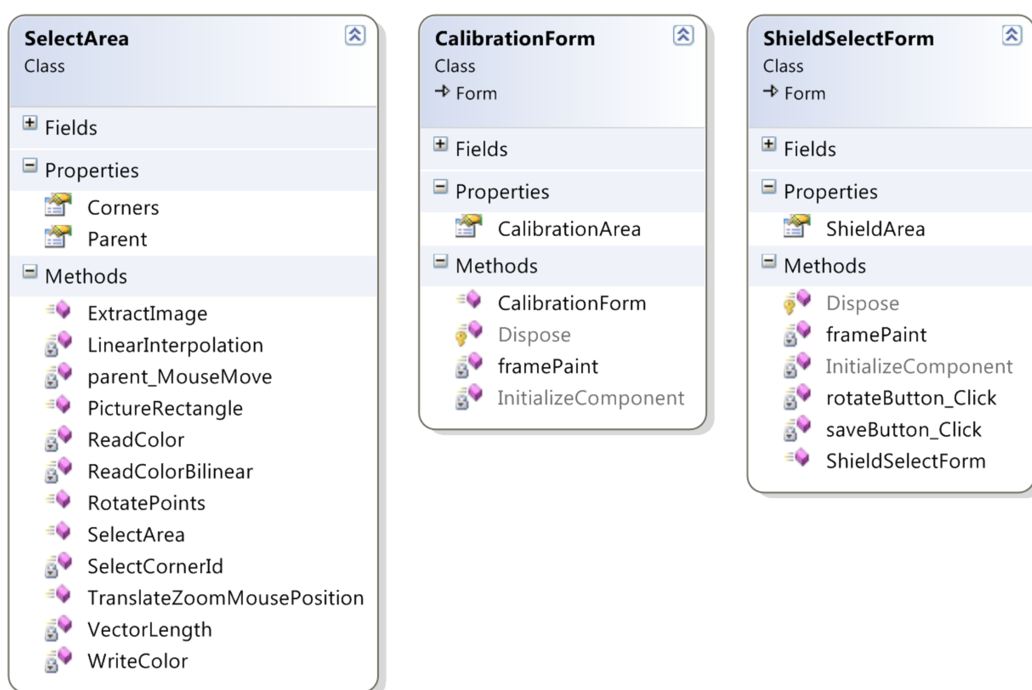
Komponenta *TeamUserControl* sdružuje 5 komponent *RobotUserControl* a kontroluje tak nastavení celého týmu. Tato komponenta je pak použita v záložkách *Our Team* a *Opponent Team*. V konstruktoru je předáno nastavení odpovídajícího týmu, na jehož základě se inicializují komponenty robotů.

Třídy pro kalibraci a výběr štítku

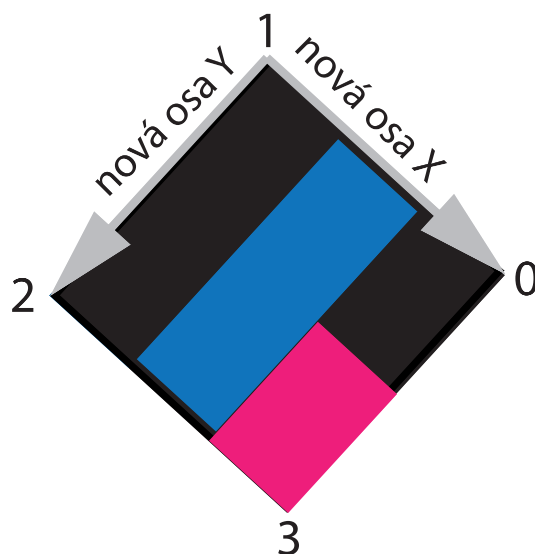
Třídy *SelectArea*, *CalibrationForm* a *ShieldSelectForm* mají společný cíl v tom, že vybírají určitou podoblast snímku z kamery. Jejich UML diagramy se nachází na obrázku 3.4.2.

Třída *SelectArea* umožňuje vybírat podoblast snímku jako polygon tvořený čtyřmi body, které jsou uloženy v atributu *Corners* – tím se dá na snímku vyznačit například kalibrační obdélník stolu nebo identifikační štítek robota. Metoda *ExtractImage* získá z vybrané oblasti dané těmito čtyřmi body obrázek, který se v něm nachází. Tento obrázek je však narovnán tak, že vektor z bodu s indexem 0 do bodu s indexem 1 bude tvořit osu X nového souřadného systému a vektor z bodu s indexem 0 do bodu s indexem 2 bude tvořit osu Y nového souřadného systému (pro lepší představu viz obrázek 3.4.2.

Lineární kombinací těchto vektorů s koeficienty a, b, jejichž hodnoty budou v intervalu $\langle 0, 1 \rangle$, se můžeme „dostat“ do libovolného pixelu v původ-



Obrázek 3.25: UML diagramy tříd *SelectArea*, *CalibrationForm* a *ShieldSelectForm*.

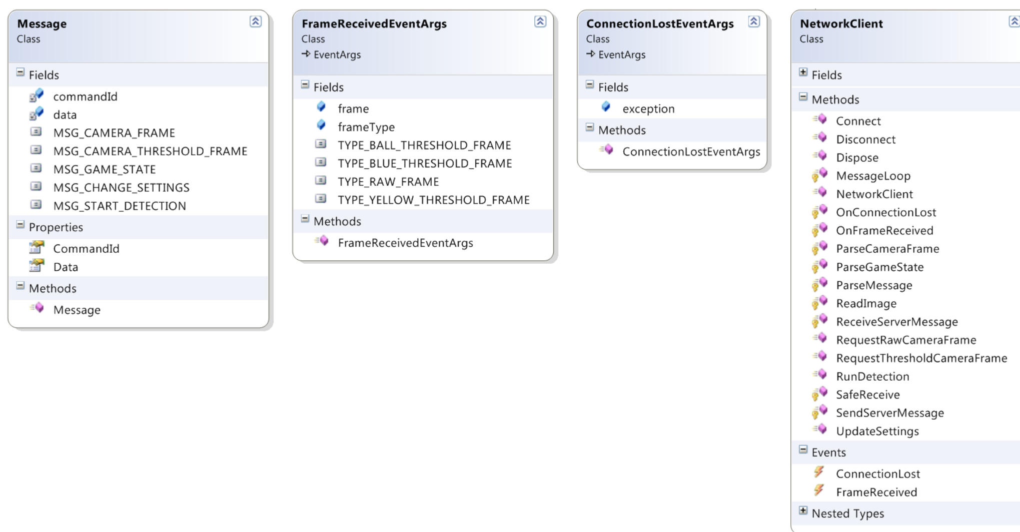


Obrázek 3.26: Určení os nového souřadného systému obrázku z bodů polygonu, který ve snímku kamery ohraničuje identifikační štítek.

ním natočeném obrázku. Budeme-li číst barvy ze souřadnic vzniklých lineární kombinací postupně rostoucích koeficientů a, b, získáme nový „narovnaný“ obraz. Ten je pak metodou *ExtractImage* vrácen. Vzniklé souřadnice lineární kombinací těchto vektorů mohou být však reálná čísla, takže s nimi nelze do paměti přistupovat přímo. Místo toho je provedena *bilineární interpolace* a výsledná barva je spočítána na základě barev čtyř nejbližších pixelů, které mají celočíselné souřadnice.

Třída *SelectArea* se zároveň stará o oblushu události stisknutí tlačítka myši a tažení bodů ohraničujícího polygonu.

Třída *CalibrationForm* představuje okno pro výběr kalibračního obdélníka stolu a využívá instanci třídy *SelectArea* k uchování rohů kalibračního obdélníka. Třída *ShieldSelectForm* reprezentuje okno, ve kterém je možné vybrat a uložit identifikační štítek robota do souboru. K získání narovnané verze štítku využívá třídu *SelectArea* a její metodu *ExtractImage*.



Obrázek 3.27: UML diagramy třídy *Message*, argumentů událostí *FrameReceivedEventArgs*, *ConnectionLostEventArgs* a třídy síťového klienta *NetworkClient*.

Třída TCP klienta, zprávy a událostí

Třídy *Message*, *FrameReceivedEventArgs*, *ConnectionLostEventArgs* a *NetworkClient* souvisí se síťovou komunikací. Jejich UML diagramy znázorňuje obrázek 3.4.2.

Třída *Message* zaobaluje přijatou či odesílanou zprávu rozpoznávacímu modulu. Obsahuje dva atributy – atribut *CommandID*, což je hodnota v datovém typu *byte*, která označuje druh zprávy nebo příkazu, který je odesílán nebo přijímán. Hodnota tohoto atributu může být jedna z konstant:

- *MSG_CAMERA_FRAME* – požaduje-li klient aktuální snímek z kamery nebo přijímá-li klient zprávu s tímto snímkem od serveru.
- *MSG_CANERA_THRESHOLD_FRAME* – požaduje-li klient aktuální prahovaný snímek z kamery se zadanými prahovacími hodnotami nebo přijímá-li klient zprávu s tímto snímkem od serveru.
- *MSG_GAME_STATE* – zpráva s rozpoznaným herním stavem přijatá od serveru.

- *MSG_CHANGE_SETTINGS* – zpráva s celkovým nastavením rozpoznávacího modulu odesílaná serveru před začátkem detekce.
- *MSG_START_DETECTION* – pokyn odesílaný serveru, aby začal detekovat a posílat aktuální herní stav.

Druhým atributem třídy *Message* je pole *bytů Data*, které obsahuje celé tělo zprávy (tj. zpráva bez hlavičky).

Třída *NetworkClientModule* je ústřední třída pro síťovou komunikaci. Definuje dvě události:

- *ConnectionLost* – při ztrátě připojení nebo chybě při komunikaci se serverem. Argumenty této události jsou definovány ve třídy *ConnectionLostEventArgs*, která předává druh výjimky, který při komunikaci nastal, v atributu *exception*.
- *FrameReceived* – je-li přijat snímek z kamery či jeho verze vzniklá prahováním. Argumenty této události definuje třída *FrameReceivedEventArgs*, která v atributu *frame* předává snímek jako instanci třídy *Bitmap* a v atributu *frameType* určuje typ přijatého snímku.

Typ přijatého snímku je vyjádřen jako datový typ *byte*, který může nabývat jedné z hodnot definované následujícími konstantami:

- *TYPE_BALL_THRESHOLD_FRAME* – přijatý snímek obsahuje obraz vzniklý prahováním s aktuálním nastavením v GUI pro míček (oranžovou barvu).
- *TYPE_BLUE_THRESHOLD_FRAME* – přijatý snímek obsahuje obraz vzniklý prahováním s aktuálním nastavením HSV mezí pro modrou barvu.
- *TYPE_YELLOW_THRESHOLD_FRAME* – přijatý snímek obsahuje obraz vzniklý prahováním s aktuálním nastavením HSV mezí pro žlutou barvu.

Na základě těchto hodnot pak dokáže grafické uživatelské rozhraní určit, v jaké záložce a pozici přijatý snímek zobrazit.

Síťový klient funguje tak, že po inicializaci získá instanci třídy *ClientModule*, což je ústřední třída modulu schopná řídicímu jádru a ostatním modulům odesílat zprávy o aktuálním herním stavu prostřednictvím metody *SendGameState*. Síťový klient si tuto instanci uloží do atributu *module*.

Klient se dokáže k serveru připojit voláním metody *Connect*, která jako argumenty požaduje IP adresu a port, na kterém server běží. Je-li síťový klient úspěšně připojen, spustí ve vlastním vlákně *listenerThread* nekonečnou smyčku, ve které čte a zpracovává přijaté zprávy od serveru. Metoda s touto smyčkou se nachází pod jménem *MessageLoop*.

Metoda *ReceiveServerMessage* přečte ze socketu hlavičku obsahující typ zprávy a délku těla zprávy. Na základě této délky pak přečte ještě tělo zprávy metodou *SafeReceive*. Na základě těchto přijatých dat je možné vytvořit instanci třídy *Message*, kterou pak zpracuje *ParseMessage*. V této metodě se na základě *CommandId* atributu přijaté zprávy rozhodne, kterou obslužnou metodu zavolat.

Metoda *ParseGameState* zpracuje tělo zprávy s rozpoznaným herním stavem. Přečte pozice a natočení jednotlivých robotů a míčku z těla zprávy a vytvoří instanci třídy *GameState*, kterou následně odešle ostatním modulům robotického fotbalu prostřednictvím instance třídy *ClientModule* metodou *SendGameState*. Herní stav je odeslán, jak pro moduly přijímající skutečný herní stav, tak i pro moduly reagující na simulovaný herní stav (tj. například vizualizační modul).

Metoda *ParseCameraFrame* zpracovává tělo zprávy s přijatým snímkem ze serveru. Jako první *byte* se v těle zprávy nachází typ přijatého snímku. Dále je přečten samotný snímek prostřednictvím metody *ReadImage* a je následně převeden na instanci třídy *Bitmap*, se kterou pak uživatelské rozhraní dokáže dále pracovat. Vzniklá bitmapa a typ přijatého snímku jsou předány jako instance třídy *FrameReceivedEventArgs* naslouchajícím třídám v GUI čekající na událost *FrameReceived*.

Metody *RequestRawFrame* a *RequestThresholdFrame* odesílají serveru zprávu se žádostí o zaslání aktuálního snímku z kamery, resp. aktuálního prahovaného snímku se zadanými HSV mezemi.

Metoda *UpdateSettings* posílá serveru zprávu s celkovým nastavením rozpoznávacího modulu (instance třídy *Settings*). Tělo této zprávy je vytvořeno serializací všech dílčích nastavení prostřednictvím volání metody *ToByteArray* ve třídě *Settings*.

Metoda *RunDetection* odešle zprávu, která serveru přikazuje, aby začal detekovat a posílat herní stav.

Třída ConnectForm a SettingsForm

Třídy *ConnectForm* a *SettingsForm* představují okno s připojovacími údaji (viz obrázek 3.17), resp. okno se všemi dílčími záložkami s nastavením.

Třída *ConnectForm* tedy slouží pouze k připojení klientovi. Na základě IP adresy a portu přečtených z textových polí v GUI se pokusí vytvořit síťového klienta (instance třídy *NetworkClient*) a připojit se s těmito údaji. V případě úspěchu se inicializuje a zobrazí okno reprezentované třídou *SettingsForm*, jíž je v konstruktoru předána instance třídy s připojených síťovým klientem.

Po zobrazení okna *SettingsForm* je načteno poslední nastavení GUI uložené v konfiguračním souboru *VisionModuleSettings.xml*. Dílčí nastavení například kamery nebo míčku jsou přiřazeny odpovídajícím komponentám GUI, které na jejich základě nastaví své grafické elementy (posuvníky, textová pole, barvy apod.) v metodě *SettingsForm_Load* (obsluha události *Load* okna – jeho první zobrazení). Nepodaří-li se nastavení ze souboru načíst, je vytvořena nová instance se standardními hodnotami.

Při ukončení okna se aktuální nastavení GUI automaticky uloží do konfiguračního souboru, aby uživatel nemusel při dalším spuštění klientského modulu znovu všechny hodnoty nastavovat.

Okno obsluhuje událost síťového klienta *FrameReceived* (přijetí klasického nebo prahovaného snímku kamery) a na základě této události aktualizuje obrázky v GUI – v záložce *Camera* ukáže aktuální snímek kamery, v záložkách *Team Colors* a *Ball* zobrazí aktuální snímek vzniklý příslušnými prahovacími hodnotami. Ve vlastním vlákně zároveň běží nekonečná smyčka, která od serveru žádá zaslání dalšího snímku z kamery a jeho variant vzniklých prahováním. Aby nedošlo k zahlcení serveru a klienta, uchovává si klient počet snímků, které vyžádal a obdržel, a na jejich základě se rozhoduje, zda bude požadovat další snímky.

Okno s nastavením zároveň obsluhuje události tlačítek a jiných grafických prvků. Zejména se stará o zobrazení oken pro výběr kalibračního obdélníka hřiště a okna pro extrakci štítků robota ze snímku kamery v plném rozlišení. Při stisku tlačítka *Run detection* v záložce *Server* se okno uzavře a serveru se

odešle zpráva s nastavením a pokyn, aby začal s detekcí a odesíláním zpráv o rozpoznaném herního stavu.

Při zavření okna se uloží nastavení do konfiguračního souboru a klient se odpojí od serveru (je-li připojen).

3.5 Testování

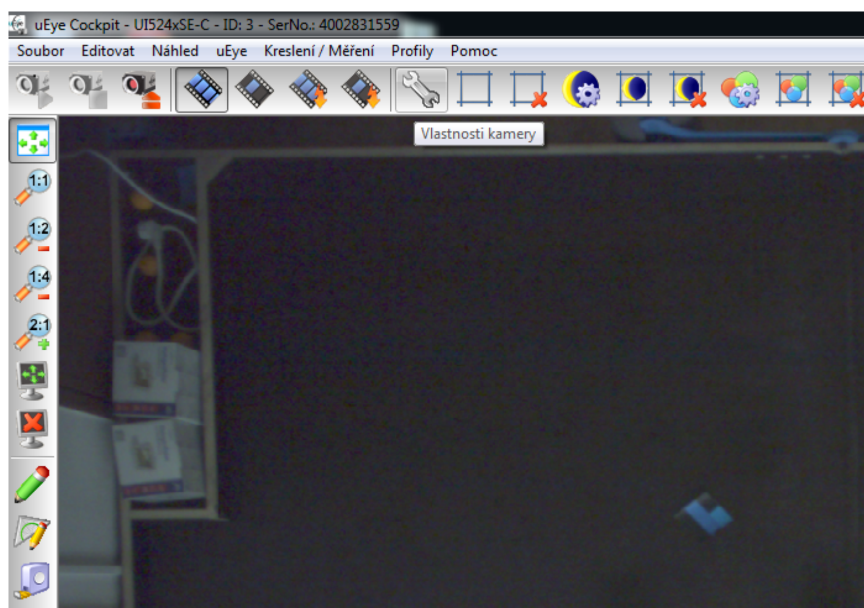
3.5.1 Příprava hřiště, robotů a kamery

V době psaní této práce nebyli skuteční roboti stále ještě k dispozici. Stůl byl dodán, ale bez konstrukce, na kterou by se dala připevnit kamera a osvětlení. Bylo tedy nutné nejprve vytvořit podmínky k tomu, aby se dal rozpoznávací modul testovat a odpovídal alespoň přibližně reálným podmínkám při skutečné hře robotického fotbalu.

Náhražky robotů byly vyrobeny z papírových krabiček o rozměrech $70 \times 70 \times 70$ mm a jejich stěny byly nasprejovány stříbrnou matnou barvou, aby alespoň zhruba připomínaly kovové tělo robota. Dále byly vytvořeny identifikační štítky robotů tak, že byl nejprve vytištěn obrázek identifikačního štítku na papír barevnou tiskárnou a vystřižený štítek byl pak přilepen na vrchní stěnu krabiček představující tělo robota.

Kamera byla umístěna do výšky 2,5 metru nad hracím stolem a upevněna na dřevěnou konstrukci tak, aby se nacházela přibližně nad středem hřiště. Kamera byla zapojena do zdroje napájení počítače určeného pro robotický fotbal a byla dále s tímto počítačem propojena ethernetovým kabelem, přes který se přenáší obrazová data. Na základě snímků z kamery zobrazených na PC bylo doladěno její natočení a pozice tak, aby ve snímcích byl vidět celý stůl a stěny hřiště byly zhruba rovnoběžné s osami výsledného snímku.

Hřiště bylo osvětleno dvojicí halogenových lamp namířených do rohů hřiště tak, aby osvětlovaly tmavší zákoutí hřiště zastíněné topením pod okny v laboratoři.



Obrázek 3.28: Okno s aktuálním proudem snímků z kamery. Kliknutím na ikonu klíče lze zobrazit menu s nastavením kamery.

3.5.2 Nastavení kamery

Jedna z dodávaných aplikací v softwarovém balíku ke kameře je *uEye Cockpit*, která dokáže zobrazit aktuální proud snímků kamery a umožňuje měnit její nastavení. Po spuštění aplikace lze změny v nastavení provádět po stisknutí ikony klíče (viz obrázek 3.28).

V záložce *Kamera* byly nastaveny položky *Pixel Clock* a *Počet snímků* na maximální hodnotu. Tím kamera začala pracovat v nejrychlejším režimu snímajícím 50,45 snímků za sekundu. Zvýšením snímkovací frekvence se však zkrátila doba expozice, tedy i množství světla dopadajícího na senzor kamery, což se projevilo zvýšeným množstvím šumu ve snímku. Položka *Expoziční doba* proto byla nastavena na maximální možnou hodnotu při zachování maximální snímkovací frekvence (viz B.6).

Na záložce *Obraz* se nachází nastavení barev a intenzity snímaného obrazu. Vzhledem k tomu, že při maximální snímkovací frekvenci je za použití standardního zesílení 1× výsledný snímek příliš tmavý, byla hodnota zesílení (položka *Gain*) zvýšena na hodnotu 2,36. Vyšší hodnoty již vykazovaly příliš velkou míru šumu. Pro další zesvětlení byla místo zesílení použita gamma ko-

Procesor	Intel Core i7-2600K, 3,4 až 3,7 GHz
Grafická karta	NVIDIA GeForce GTX 580
Paměť	16 GB, 666 MHz, CL 9
Pevný disk	Western Digital Velociraptor WD4500HLH, 10000 otáček / min
Základní deska	Gigabyte Z68XP-UD5

Tabulka 3.4: Hardwarová konfigurace počítače určeného pro robotický fotbal na ZČU.

rekce (posuvník *Gamma*) s parametrem 1,49. Tyto hodnoty byly zvoleny jako kompromis mezi příliš velkým zašuměním obrazu a ztrátou kontrastu barev vlivem příliš vysoké hodnoty gamma korekce. Tato nastavení jsou zobrazena na obrázku B.7.

Dále v záložce *Velikost* bylo v položce *Rozlišení* nastaveno nejvyšší možné rozlišení generovaného snímku – *1280 × 1024 (1.3M SXGA)*. V záložce *Formát* pak byl nastaven RGB formát se zarovnáním velikosti pixelu na 32 bitů s normální kvalitou odstranění Bayerovy mřížky (viz obrázek B.8). V záložce *Závěrka* byl nastaven mód *Global Shutter*, aby nedocházelo ke vzniku artefaktů u rychle pohybujících se robotů a míčku. Nakonec v záložce *Barva* byla vybrána v sekci *Matice korekce IR barevného filtru* možnost *HQ*, která ze všech nabízených variant poskytovala nejvěrnější podání barev ve snímku.

Výsledná nastavení byla uložena volbou v menu *Soubor* → *Ulož parametry* → *Do souboru...*. Konfigurační soubor byl uložen do adresáře `C:\Robofotbal` pod názvem `kamera.ini`.

3.5.3 Testování rychlosti

Rozpoznávací modul byl vyvíjen a testován na PC sestaveném speciálně pro účely robotického fotbalu týmu Západočeské univerzity v Plzni. Jedná se o počítač vybavený procesorem *Intel Core i7-2600K* se čtyřmi jádry taktovanými dynamicky na frekvenci 3,4 až 3,7 GHz a grafickou kartou *NVIDIA GeForce GTX 580* s 512-ti CUDA jádry. Počítač má k dispozice 16 GB paměti a pevný disk *Western Digital Velociraptor WD4500HLH* o kapacitě 250 GB a rychlosti 10000 otáček za minutu. Konfiguraci použitého počítače shrnuje tabulka 3.5.3.

Bylo provedeno několik druhů testů, jejichž cílem bylo detailněji pro-

zkoumat dobu zpracování jednotlivých částí algoritmu, a navíc prozkoumat možnosti jejich urychlení na GPU technologií CUDA. Všechny testy byly měřeny za využití časovačů v knihovně Thread Building Blocks, které poskytují měření času s vysokou přesností, zejména ve vícevláknových aplikacích. U mnoha měřených operací byly však doby zpracování i tak příliš krátké, například v řádech několika mikrosekund, a proto byly tyto operace měřeny ve smyčce, která se opakovala 10000× a jako výsledek měření byla použita průměrná doba zpracování během tohoto počtu iterací.

Nejprve byla provedena řada testů, jejichž cílem bylo prozkoumat rychlost vybraných algoritmů OpenCV na CPU a po akceleraci na GPU užitím technologie CUDA. Všechny testy byly prováděny na snímcích s rozlišením 1280 × 1024 pixelů a třemi barevnými kanály. Byla měřena doba nahrání snímku na GPU přes sběrnici PCI-Express a doba potřebná ke stažení zpracovaného snímku z GPU. Dále pro CPU i GPU byly určeny doby převodu snímku z barevného prostoru BGR do prostoru HSV (funkce *cv::cvtColor*), doba potřebná k prahování jednoho kanálu snímku (*cv::threshold*), doba potřebná pro násobení všech hodnot v obrazové matici konstantou (*cv::multiply*) a čas potřebný k osminásobnému zmenšení rozlišení snímku (*cv::resize*). Výsledky shrnuje tabulka 3.6.

Dále byly několika testy prozkoumány možnosti urychlení části rozpoznávacího algoritmu na GPU. Většinu výpočetního času rozpoznávací algoritmus stráví hledáním barevných oblastí modré, žluté a oranžové barvy a následným určováním jejich obrysů. Byla naměřena rychlost hledání kontur barevných oblastí procházením snímku v plném rozlišení na CPU a na CPU ve spolupráci s GPU. Modul GPU v OpenCV neobsahuje proceduru pro vyhledávání obrysů (procedura *cv::findContours*), takže je nutné nejprve na GPU najít barevné oblasti převodem do barevného prostoru HSV a prahováním. Výsledný obraz je pak předán CPU, které v něm najde kontury. Následně jsou výsledky dány do kontextu s použitým algoritmem, který pracuje pouze na CPU a hledá nejprve oblasti zájmu v osminásobně zmenšeném snímku a poté v těchto oblastech nachází kontury s maximální přesností (viz sekce 3.3.2). Dosažené výsledky shrnuje tabulka 3.6.

Další sada testů určila doby potřebné pro nalezení míčku, robotů našeho týmu a robotů soupeřova týmu. Byla určena jak celková doba při sekvenčním zpracování těchto úloh, tak i při paralelním zpracování těchto úkolů knihovnou Thread Building Blocks. Výsledky jsou zaznamenány v tabulce 3.6.

Poslední sadou testů byla určena celková časová prodleva od získání snímku

z kamery až do obdržení herního stavu modulem herní strategie. Výsledky měření se nachází v tabulce 3.6.

3.6 Zhodnocení výsledků

V této práci byl navržen, popsán, implementován a otestován modul využívající metody počítačového vidění k rozpoznání herního stavu při hře robotické fotbalu z reálných snímků kamery. Byla zprovozněna kamera, počítač a bylo připraveno zázemí pro testování a budoucí vývoj algoritmů počítačového vidění pro účely týmu robotického fotbalu Západočeské univerzity v Plzni.

Nad rámec zadání této práce bylo vytvořeno komplexní grafické uživatelské rozhraní, které uživateli umožňuje nastavit nejdůležitější parametry rozpoznávacího modulu přívětivou formou. Navíc byl proveden výzkum možného dalšího zrychlení zpracování obrazu technologií CUDA, která umožňuje přenést výpočetně intenzivní části kódu na GPU.

Z tabulky 3.6 vyplývá, že na testovaném PC dokáže CUDA významně urychlit elementární operace počítačového vidění využité v rozpoznávacím algoritmu. Jako nejnáročnější operace se ukázal převod snímku z barevného prostoru BGR do prostoru HSV, který při převodu snímku v plném rozlišení 1280×1024 trvá na CPU *3,4 ms*. To je nepřijatelně dlouhá doba, protože je třeba provést ještě další časově náročné operace, jako je například vyhledávání kontur barevných oblastí modré, žluté a oranžové barvy. GPU však převod dokázalo uskutečnit přibližně $15\times$ rychleji. Násobení hodnot v obrázku dokázalo GPU provést zhruba $18\times$ rychleji a zmenšení obrazu na osminu jeho původní velikosti proběhlo na GPU $3\times$ rychleji než na CPU. Výkon GPU je však limitován přenášením dat po sběrnici PCI-Express, která u snímku kamery v plném rozlišení způsobuje latenci zhruba 1 ms, než je snímek dopraven do paměti grafické karty. Stažení zpracovaného snímku v plném rozlišení též trvá okolo 1 ms. Přenosy dat mezi CPU a GPU jsou tedy drahá operace a významně brzdí celý výpočet.

Dále byly testovány možné přístupy pro nalezení kontur barevných oblastí ve snímku, což je operace, kterou navržený rozpoznávací algoritmus využívá ve všech svých základních úlohách – hledání robotů našeho týmu, hledání robotů soupeřova týmu a při hledání míčku. Naměřené hodnoty představují dobu výpočtu nutnou pro všechny tyto úkoly dohromady. Barevné oblasti se hledají převodem snímku z prostoru BGR do prostoru HSV a následným

	CPU	*GPU*
Nahrání snímku BGR snímku v rozlišení 1280 × 1024 *pixelů*	-	1,0 ms
Převod z BGR do HSV	3,42 ms	0,23 ms
Prahování jednoho kanálu	0,003 ms	0,06 ms
Násobení prvků snímku	1,27 ms	0,07 ms
Osminásobné zmenšení	0,27 ms	0,08 ms
Stažení BGR snímku v rozlišení 1280 × 1024 *pixelů*	-	1.0 ms

Tabulka 3.5: Porovnání rychlosti CPU a GPU ve vybraných operacích. Všechny testy byly prováděny na snímku o velikosti 1280 × 1024 pixelů se třemi barevnými kanály.

binárním prahováním pixelů, které se nachází v určitém rozsahu HSV hodnot.

GPU však již nedokáže nalézt obrysy těchto barevných oblastí, a musí proto zpracovaný snímek po prahování přenést zpět, aby v něm CPU mohlo najít obrysy. Nalezení kontur ve spolupráci s GPU zabere *9,5 ms* (viz tabulka 3.6, samostatně by ten samý problém procesor počítal *713 ms*. Spoluprací s GPU bylo tedy dosaženo *75-ti násobného* zrychlení oproti variantě, která vše počítá na CPU. Ani spolupráce s GPU však nedokáže porazit efektivnější algoritmus, který byl popsán v sekci 3.3.2 a který nejprve obraz osminásobně zmenší a v něm najde hrubé obrysy oblastí s týmovou barvou, na které se pak zaměří v plném rozlišení. Na závěr je nutné poznamenat, že pokud by bylo hledání barev a převod barev z BGR do HSV na GPU optimalizováno například napsáním vlastního CUDA kernelu, došlo by pravděpodobně k dalšímu výraznému zrychlení. Limitujícím faktorem je však stále to, že pro GPU zatím neexistuje efektivní algoritmus pro vyhledávání obrysů souvislých oblastí.

Dalšími testy byly prozkoumány časové náročnosti jednotlivých základních úkolů rozpoznávacího algoritmu – hledání míčku, rozpoznání robotů našeho týmu a rozpoznání stavu robotů soupeřova týmu. Doby výpočtu při zpracování těchto základních úloh jsou zhruba podobné (viz tabulka 3.6), protože všechny řeší problém podobným způsobem, jehož nejvíce výpočetně náročná část je nalezení kontur oblastí s určitou barvou ve snímku. Jsou-li tyto úlohy zpracovány sekvenčně v jedno vlákně na procesoru, je celková doba rozpoznávání v průměru 9,2 ms. Je-li použito paralelní zpracování knihovnou Thread Building Blocks, dokáže být herní stav rozpoznán v průměru za 4,5 ms. Knihovna TBB tedy urychlila výpočet přibližně dvojnásobně.

	Doba zpracování
Nalezení barevných oblastí pro náš tým, soupeře a míček převodem do prostoru HSV a prahováním na GPU	3,7 ms
Stažení prahovaných snímků z GPU na CPU	1,0 ms
Nalezení výsledných kontur na CPU	4,8 ms
Celková doba zpracování snímku v plném rozlišení: spolupráce CPU+GPU	9,5 ms
Celková doba zpracování snímku v plném rozlišení: CPU samostatně	713 ms
Celková doba zpracování snímku na CPU při použití optimalizovaného algoritmu a osminásobného podvzorkování snímku	4,5 ms

Tabulka 3.6: Doba hledání kontur různými přístupy. GPU nedokáže vyhledávat kontury samostatně, a proto musí spolupracovat s CPU. Prohledávání snímku v plném rozlišení je i za použití GPU pomalejší, než nalezení kontur „chytřejším“ algoritmem na CPU.

	Doba zpracování
Nalezení míčku	2,7 ms
Rozpoznání robotů našeho týmu	3,3 ms
Rozpoznání robotů soupeřova týmu	3,2 ms
Celkem sekvenčně	9,2 ms
Celkem paralelně s TBB	4,5 ms

Tabulka 3.7: Doby zpracování jednotlivých fází rozpoznávacího algoritmu. Porovnání doby sekvenčního zpracování a paralelního zpracování při použití knihovny Thread Building Blocks.

	Doba zpracování
Rozpoznání herního stavu	4,5 ms
Odeslání zprávy s herním stavem	0,02 ms
Přečtení a zpracování zprávy klientským modulem v řídicím software robotického fotbalu.	0,03 ms
Celkem	4,6 ms

Tabulka 3.8: Doby zpracování jednotlivých fází celého systému

V posledním testu výkonu byla analyzována celková doba činnosti celého systému od obdržení snímku rozpoznávacím modulem do získání informace o rozpoznaném herním stavu v modulu herní strategie. Režie na přenos zpráv a jejich zpracování skrze sockety je minimální a komunikace je téměř okamžitá, jelikož doba k ní potřebá je menší než 0,1 ms. Výsledky shrnuje tabulka 3.6.

Celé řešení bylo otestováno na snímcích ze skutečné kamery umístěné ve výšce 2,5 metru nad hracím stolem. Algoritmus byl testován s ručně vyrobenými roboty, kteří se svými rozměry a vzhledem zhruba podobají při pohledu shora skutečným robotům. Funkčnost byla testována na neosvětleném hřišti i ve variantě hřiště nasvětleného dvěma lampami. Ačkoliv v obou případech bylo hřiště špatně a velmi nerovnoměrně nasvětlené, dokázal algoritmus korektně rozpoznat stav hry, dokonce i v případech, kdy uživatel zakryl štítky robotů či míček rukou, zasahoval svým tělem do hřiště či jinak rušil zpracovávaný snímek, což je pravidly vyloučeno.

Algoritmus rozpoznávání se tedy ukázal jako poměrně robustní a rychlý. Jediným nedostatkem je to, že kamera produkuje výrazně zašuměný obraz a hřiště je špatně osvětleno, což limituje možnosti rozlišení sekundárních barevných značek pro identifikaci robotů, protože z obrazu, který je kamerou generován, by tyto značky měl nejspíš problém rozpoznat i člověk. Při použití lamp se situace spíše zhoršila, protože došlo k dalšímu zkreslení barevného podání a hřiště bylo v rozích stále nedostatečně osvětleno. Do budoucna tedy bude nutné zajistit korektní nasvícení hřiště v souladu s pravidly a navrhnout design štítku s větší identifikační ploškou.

Vzhledem k tomu, že tato práce má spíše výzkumný charakter a jde o vůbec první pokus týmu Západočeské univerzity v Plzni o rozpoznání stavu hry ze skutečných snímků kamery, nepovažuji tyto nedostatky za příliš zásadní. Tato práce splnila všechny body definované v zadání a přispěla k dalšímu

rozvoji projektu robotického fotbalu na Západočeské univerzitě v Plzni. Považuji ji proto za úspěšnou.

4 Závěr

Projekt robotického fotbalu na Západočeské univerzitě v Plzni byl v době zadání této diplomové práce stále ještě ve svých počátcích. Nebyl k dispozici hrací stůl, ani skuteční roboti, pouze kamera a software, který dokázal simulovat a vizualizovat stav virtuální hry robotického fotbalu na počítači. V průběhu vytváření této práce byl dodán hrací stůl, díky němuž bylo poprvé možné prozkoumat úskalí vývoje rozpoznávacího modulu v realistických podmínkách.

Rozpoznávací modul hraje v robotickém fotbalu významnou roli. Díky němu je možné ze snímku kamery získat informaci o aktuální pozici robotů a míčku, se kterou pak můžou dál pracovat moduly s herní strategií. Robustnost a rychlost implementace rozpoznávacího modulu pak výrazně ovlivňuje schopnost konkurovat ostatním týmům v celosvětových soutěžích robotického fotbalu.

V rámci této práce bylo připraveno zázemí pro vývoj a testování rozpoznávacího modulu na základě proudu skutečných snímků z digitální kamery. Kamera byla připevněna do výšky 2,5 metru nad hracím stolem a zapojena do počítače určeného pro robotický fotbal. Na počítač byl nainstalován operační systém i všechny knihovny potřebné pro vývoj. Kolem hřiště byly umístěny lampy pro dosažení rovnoměrnějšího osvětlení. Na závěr byly vytvořeny náhražky robotů a navrženy jejich identifikační štítky, protože skutečná verze robotů během vývoje této práce stále ještě nebyla k dispozici.

Rozpoznávací modul byl implementován v jazyce C++ s důrazem na vysokou rychlost rozpoznávání, robustnost a snadnou rozšiřitelnost. Jeho funkčnost byla ověřena na hřišti a robotech tak, aby rozpoznávaná scéna odpovídala co nejblíže požadavkům definovaných v pravidlech hry robotického fotbalu. Ačkoliv hřiště bylo špatně a nerovnoměrně osvětleno, což pravidla hry vylučují, a kamera produkovala velmi zašuměný obraz, dokázal algoritmus i přesto korektně rozpoznat pozice všech robotů i míčku. Dle pravidel se u stolu nesmí nikdo pohybovat, ani jakkoliv zasahovat do prostoru hracího stolu. Rozpoznávací modul však i v takových případech reagoval korektně a například po zakrytí robota rukou se z této situace dokázal snadno zotavit.

Rozpoznávací modul dokáže zpracovat snímek a předat rozpoznaný herní stav modulu herní strategie v průměru za **4,6 ms**. Nad rámec zadání této práce bylo vytvořeno rozsáhlé grafické uživatelské rozhraní umožňující nasta-

vit všechny důležité parametry rozpoznávacího modulu. Na závěr byla navíc i prozkoumána možnost dalšího urychlení zpracování snímku přesunem části výpočtů na grafickou kartu za využití technologie *CUDA*.

Přehled zkratek

- CCD – Zkratka pro *Charge-coupled device*. Jde o druh snímacího čipu kamery, který má lepší světelnou citlivost a nabízí obecně lepší obraz než snímací čipy CMOS.
- CMOS – Zkratka pro *Complementary Metal Oxide Semiconductor*. Tato zkratka označuje snímací čipy kamery, které mají obecně menší světelnou citlivost a produkují horší obraz než čipy CCD, jsou však výrazně levnější na výrobu.
- CPU – Zkratka pro *Central Processing Unit* neboli centrální procesor počítače.
- CUDA – Zkratka pro *Compute Unified Device Architecture*, což je platforma pro akceleraci výpočtů na procesoru grafické karty.
- FIRA – Zkratka pro *Federation of International Robot-soccer Association*. Jedná se o organizaci zaštiťující soutěže robotického fotbalu a definující jejich pravidla hry.
- GPU – Zkratka pro *Graphics Processing Unit*. Jedná se o procesor grafické karty.
- GUI – Zkratka pro *Graphical User Interface*. Jedná se o grafické uživatelské rozhraní aplikace.
- HSV – Zkratka pro *Hue, Saturation, Value* – prostor definující barvu na základě jejího odstínu, nasycení a intenzity.
- IP – Zkratka pro *Internet Protocol*. Jedná se o jeden ze základních protokolů síťové komunikace.
- OpenCV – Zkratka pro *Open Computer Vision*. Je to knihovna implementující algoritmy počítačového vidění a strojového učení.

- RGB (BGR) – Zkratka pro *Red, Green, Blue* (či *Blue, Green, Red)*. Je to barevný prostor, který barvu vyjadřuje na základě intenzit tří základních barev – červené, zelené a modré. Rozdíl mezi RGB a BGR je v pořadí ukládání jednotlivých složek například v obrázku.
- TBB – Zkratka pro *Thread Building Blocks*. Jedná se o knihovnu pro paralelizaci výpočtů na CPU.
- TCP – Zkratka pro *Transmission Control Protocol*. Je to jeden ze základních síťových protokolů pro spolehlivý přenos dat.

Literatura

[1] SHOHAM, Yoav; LEYTON-BROWN, Kevin. *Multiagent Systems: Algorithmic, Game-Theoretic, and Logical Foundations*, ISBN 978-0521899437, Cambridge University Press, 2008

[2] RoboCup, *A Brief History of RoboCup*, dostupné z http://www.robocup.org/about-robocup/a-brief-history-of-robocup/

[3] Federation of International Robot-soccer Association, *FIRA overview*, dostupné z http://www.fira.net/?mid=firaoverview

[4] Federation of International Robot-soccer Association, *FIRA MiroSot Game Rules For Middle League and Large League*, dostupné z http://www.fira.net/?module=file&act=procFileDownload&file_srl=2870&sid=09c8a14e80aa45c9df6152b1cfbd534b

[5] FRIČ, Vojtěch. *Robofotbal: modul počítačového vidění*. Plzeň, 2012. Bakalářská práce (Bc.). Západočeská univerzita v Plzni, Fakulta aplikovaných věd. Vedoucí práce Kamil Ekštein.

[6] ECKSTEIN, Robert. *Vizualizační modul pro robotický fotbal*. Plzeň, 2011. Bakalářská práce (Bc.). Západočeská univerzita v Plzni, Fakulta aplikovaných věd. Vedoucí práce Kamil Ekštein.

[7] ALTMAN, Petr. *Jádro řídicího systému a virtualizační modul pro robotický fotbal*. Plzeň, 2011. Bakalářská práce (Bc.). Západočeská univerzita v Plzni, Fakulta aplikovaných věd. Vedoucí práce Kamil Ekštein.

[8] MA, Gang; LIU, Tian-shi; HAN Jia-xin; WANG Xiao-xiao. *The Color Tag Design and Color Model Study in Mirosot*, 2010 International Conference on Artificial Intelligence and Computational, 2010

[9] JUN, Zhou; HONG-SHUANG Zhang; YONG Chen; CHANG-ZHI Zhang. *Design of vision system and recognition algorithm in Mirosot*, 2004 IEEE Conference on Cybernetics and Intelligent Systems, 2004

[10] NOVAK, G.; SPRINGER, R. *An introduction to a vision system used for a MiroSOT robot soccer system*, 2004 Second IEEE International Conference on Computational Cybernetics, 2004

[11] XIAOCHUAN, Zhang; CHANZHEN, Liu; QIANLONG, Yu; ZUSHU Li. *A Identification Method for Robotics Soccer*, 2010 Second International Workshop on Education Technology and Computer Science (ETCS), March 2010

[12] SMITH, Stephen. *The Scientist and Engineer's Guide to Digital Signal Processing, Second Edition*, ISBN 0-9660176-7-6, California Technical Publishing, 1999,

[13] XIAN-XIANG, Wu; BAO-LONG, Guo. *FFT-Based Orientation Recognition Algorithm in Mirosot*, 2008 Eighth International Conference on Intelligent Systems Design and Applications, 2008

[14] STEVENS, Richard. *TCP/IP Illustrated, Vol. 1: The Protocols*, ISBN 978-0201633467, Addison-Wesley Professional, 1993

[15] MSDN. *Windows Sockets 2*, dostupné z `http://msdn.microsoft.com/cs-cz/library/windows/desktop/ms740673(v=vs.85).aspx`

[16] SHAPIRO, Linda; STOCKMAN, George. *Computer Vision*, ISBN 978-0130307965, Prentice Hall, 2001

[17] Willow Garage, OpenCV Wiki, dostupné z `http://opencv.willowgarage.com/wiki/Welcome`

[18] SZELISKI, Richard. *Computer Vision: Algorithms and Applications*, ISBN 978-1-84882-934-3, Springer-Verlag London Limited, 2011

[19] SANDERS, Jason; KANDROT, Edward. *CUDA by Example: An Introduction to General-Purpose GPU Programming*, ISBN 978-0131387683, Addison-Wesley Professional, 2010

[20] FARBER, Rob. *CUDA Application Design and Development*, ISBN 978-0123884268, Morgan Kaufmann, 2011

[21] NVIDIA Corporatio. *CUDA C Programming Guide*, dostupné z `http://docs.nvidia.com/cuda/pdf/CUDA_C_Programming_Guide.pdf`

[22] Khronos Group. *OpenCL - The open standard for parallel programming of heterogeneous systems*, dostupné z `http://www.khronos.org/opencl/`http://www.khronos.org/opencl/

[23] Open Source Initiative, *The BSD 2-Clause License*, dostupné z `http://opensource.org/licenses/bsd-license.php`

[24] Emgu CV, *Emgu CV Wiki*, dostupné z `http://www.emgu.com/wiki/index.php/Main_Page`

[25] CYGANEK, Boguslaw; SIEBERT, Paul., *An Introduction to 3D Computer Vision Techniques and Algorithms*, ISBN 978-0-470-01704-3, John Wiley & Sons Limited, 2009

[26] ALPAYDIN, Ethem. *Introduction to Machine Learning, Second Edition*, ISBN 978-0-262-01243-0, The MIT Press, Cambridge, London, 2010

[27] MITCHELL, Tom. *Machine Learning*, ISBN 0070428077, McGraw-Hill Science/Engineering/Math, 1997

[28] THEODORIDIS, Sergios; KOUTROUMBAS, Konstantinos. *Pattern Recognition, Fourth Edition*, ISBN: 978-1-59749-272-0, Elsevier, London, 2009

[29] BRADSKI, Gary; KAEHLER, Adrian. *Learning OpenCV*, ISBN: 978-0-596-51613-0, O'Reilly Media, Inc., 2008

[30] LAGANIÈRE, Robert. *OpenCV 2 Computer Vision Application Programming Cookbook*, ISBN 978-1-849513-24-1, Packt Publishing Ltd., 2011

[31] Willow Garage. *OpenCV Documentation*, dostupné z `http://docs.opencv.org/trunk/index.html`

[32] Willow Garage. *OpenCV CUDA Platform*, dostupné z `http://opencv.org/platforms/cuda.html`

[33] Willow Garage. *OpenCV Installation Guide*, dostupné z `http://opencv.willowgarage.com/wiki/InstallGuide`

[34] Intel Corporation. *Thread Building Blocks - Home Page*. Dostupné z `http://threadingbuildingblocks.org/`

[35] IDS Imaging. *UI-5240CP GigE Camera*, dostupné z `http://en.ids-imaging.com/store/produkte/kameras/gige-kameras/ui-5240cp.html`

A Uživatelská příručka

A.1 Spuštění

Před spuštěním je nutné zkontrolovat, zda je ethernetovým kabelem do počítače připojena kamera, a také zkontrolovat její síťové nastavení dle sekce A.2. Dále je nutné zajistit korektní nastavení Visual Studia pro překlad rozpoznávacího modulu (viz sekce A.3) a řídicího software platformy robotického fotbalu (viz A.4). V případě opětovné instalace počítače pro robotický fotbal je nutné nainstalovat všechny knihovny dle sekce A.5. V případě smazání projektů s rozpoznávacím modulem a řídicí platformy z počítače určeného pro robotický fotbal, je lze obnovit zkopírováním z CD, které je k této práci přiloženo.

Po překladu rozpoznávacího modulu, jehož hlavní soubor projektu *VisionModule.sln* se nachází ve složce `C:\Robofotbal\VisionModule`, lze spustit server přímo z Visual Studia 2012 nebo nalezením přeloženého programu *VisionModule.exe* v odpovídající složce se zvolenou konfigurací. V případě spouštění z příkazové řádky se server spouští ve tvaru:

`VisionModule.exe [`*`port`*`]`

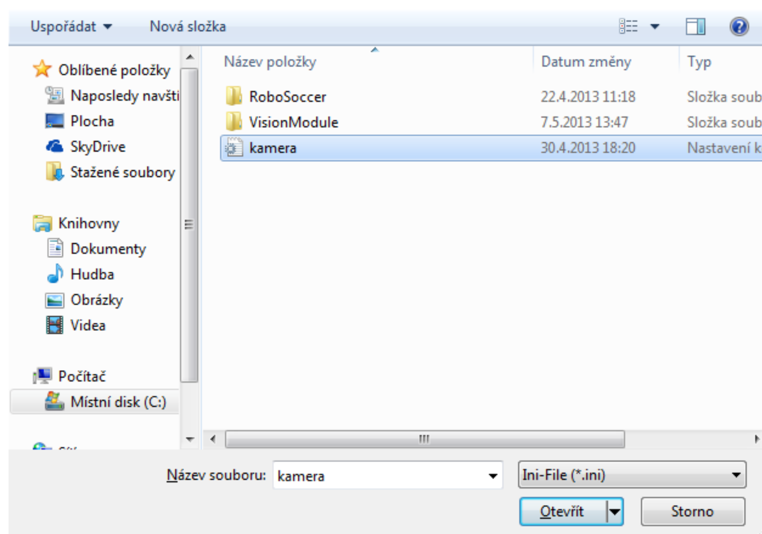
Kde *port* určuje číslo portu, na kterém má server naslouchat klientovi. Není-li uvedena žádná hodnota, je server spuštěn na jeho standardním portu 55000.

Ihned po spuštění se inicializuje kamera. Uživatel je vyzván souborovým dialogem, aby uvedl konfigurační soubor s obrazovým nastavením kamery, který se má pro rozpoznávání použít (viz obrázek A.1). Ten je na PC určeném pro robotický fotbal standardně uložen v souboru:

```
C:\Robofotbal\kamera.ini
```

V případě, že se kameru nepodaří inicializovat, server okamžitě ukončí svoji činnost. S největší pravděpodobností je chybné síťové nastavení kamery. Pro vyřešení tohoto problému je třeba postupovat dle sekce A.2.

Po spuštění serveru je nutné spustit platformu robotického fotbalu buď překladem z Visual Studia 2010 (viz sekce A.4), nebo spustit již přeložený



Obrázek A.1: Souborový dialog, který se objeví po spuštění serveru. Je nutné zadat konfigurační soubor s obrazovým nastavením kamery. Ten se standardně nachází v souboru *kamera.ini* ve složce *Robofotbal*.

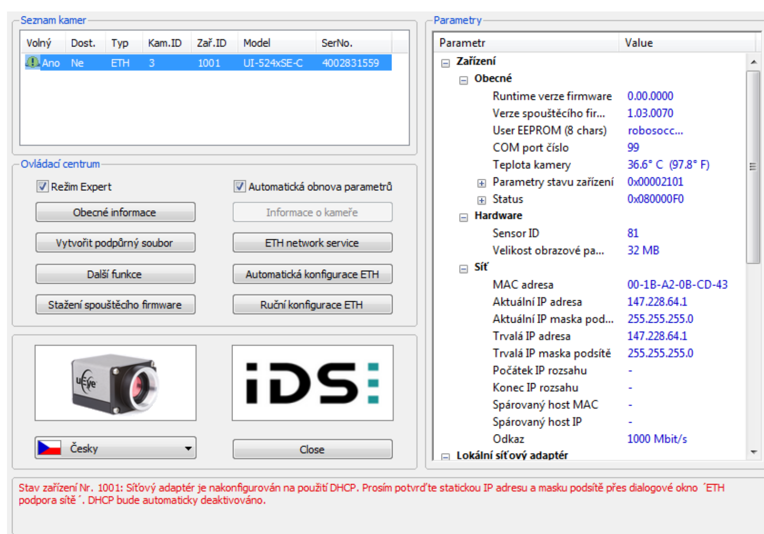
program *RoboSoccer.exe* ve složce:

```
C:\Robofotbal\RoboSoccer\bin\x86\Release
```

V případě, že se neobjeví okno s připojovacími údaji (viz obrázek 3.17), zkontrolujeme, zda se v horní liště s moduly nachází po stisknutí pravého tlačítka. Pokud ano, zvolíme jej, pokud ne, nebyl klientský modul pravděpodobně správně načten. V tom případě je nutné zkontrolovat, zda se ve složce se spustitelným programem řídicí platformy nachází knihovna *RoboSoccer.Vision.Client.dll*, který vzniká překladem klientského modulu. Dále je nutné provést kontrolu, zda se v konfiguračním souboru modulů platformy robotického fotbalu *config.xml* nachází řádek:

```
<Module file="RoboSoccer.Vision.Client.dll"/>
```

Nastavení všech náležitostí rozpoznávacího modulu je pak podrobně popsáno v sekci 3.4.1. Pro správnou funkčnost algoritmu je důležité zejména nastavení HSV prahovacích hodnot pro modrou, žlutou a oranžovou barvu tak, aby objekty byly nalezeny i v tmavších zákoutích hřiště. Před začátkem



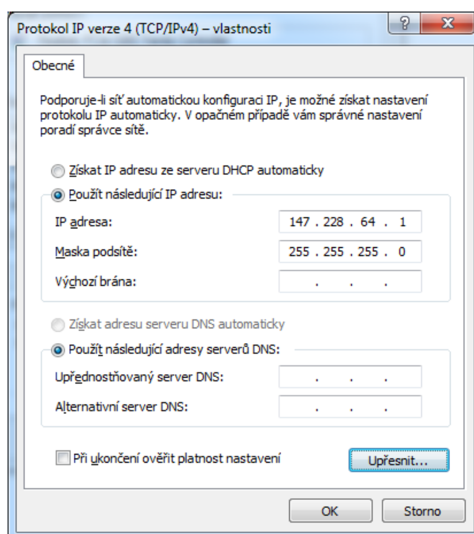
Obrázek A.2: Okno správce kamer *IDS Manager*. V horním panelu by se po spuštění měla nacházet jedna položka, která představuje připojenou kameru určenou pro robotický fotbal. Výstražný trojúhelník však značí, že síťové nastavení kamery není v pořádku.

nastavování je vhodné rozmístit roboty a míček do částí s různou intenzitou osvětlení a až potom hledat optimální prahovací hodnoty. Pro správnou funkčnost je taky nutné vyextrahovat snímky štítků robotů za aktuálního druhu osvětlení (lampy totiž barvy na štítku zkreslují). Po nastavení všech náležitostí je možné v záložce *Server* spustit samotnou detekci a pak už jen sledovat rozpoznané pozice ve vizualizačním modulu uvnitř řídicího software.

A.2 Síťové nastavení kamery

Softwarový balík určený pro ovládání kamery se nachází v nabídce *Start* → *Všechny programy* → *IDS*. Důležitým programem v této nabídce je *IDS Camera Manager*. Jedná se o nástroj pro správu kamer. Po jeho spuštění by se mělo objevit okno, v jehož horní nabídce se nachází seznam připojených kamer (viz obrázek A.2). Nachází-li se nalevo u řádku s kamerou výstražný trojúhelník, je kamera nalezena, avšak síťové nastavení počítače není správné.

Aby kamera mohla správně fungovat, je třeba, aby počítač, ke kterému je připojena, měl statickou IP adresu. Typicky je nutné následující proces



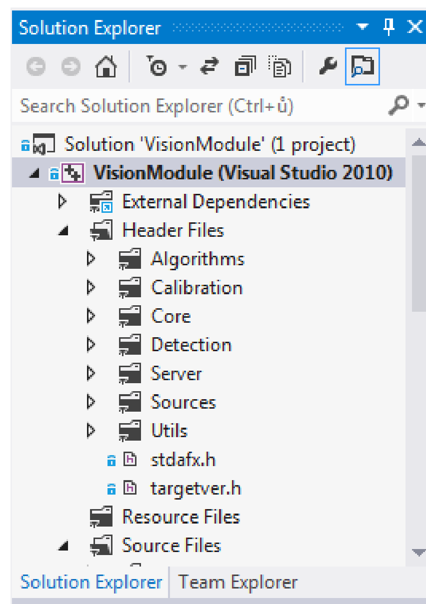
Obrázek A.3: Okno pro nastavení protokolu DHCP a statické IP adresy. Vypnutí automatického přidělování síťového nastavení a nastavení statické IP adresu lze zajistit volbu *Použít následující IP adresu* a vyplněním příslušné statické IP adresy počítače a masky podsítě.

opakovat pokaždé, když se připojí ethernetový kabel od kamery do počítače, který byl předtím připojen síťovým kabelem k internetu, a musel být povolen protokol *DHCP*, který počítači dynamicky přiděluje síťová nastavení.

Vypnutí DHCP a nastavení statické IP adresy se provede v nabídce *Start* → *Ovládací panely* → *Síť a internet* → *Centrum síťových připojení a sdílení*. Dále se v levém menu vybere volba *Změnit nastavení adaptéru* a zvolí se adaptér *Připojení k místní síti*, v jehož nabídce se zvolí možnost *Vlastnosti*. Z této nabídky se označí položka *Protokol IP verze 4 (TCP/IPv4)* a stiskne tlačítko *Vlastnosti*.

V okně, které se objeví, se vypne DHCP tak, že se zaškrtne možnost *Použít následující IP adresu* a vyplní se aktuální IP adresa počítače a maska podsítě (tu stačí opsat z pravé části na obrázku A.2). Výsledné okno by mělo vypadat jako na obrázku A.3.

Poté by kamera již měla být připravena k použití, což lze ověřit v aplikaci *IDS Camera Manager* tak, že se u kamery nenachází výstražný trojúhelník a ve spodní části okna již neobjevuje chybová hláška o povoleném protokolu DHCP. Proud snímků z kamery lze zobrazit poklepáním levým tlačítkem



Obrázek A.4: Okno se seznamem souborů v projektu. Nastavit projekt lze stisknutím pravého tlačítka na projekt *VisionModule* a volbou *Properties*.

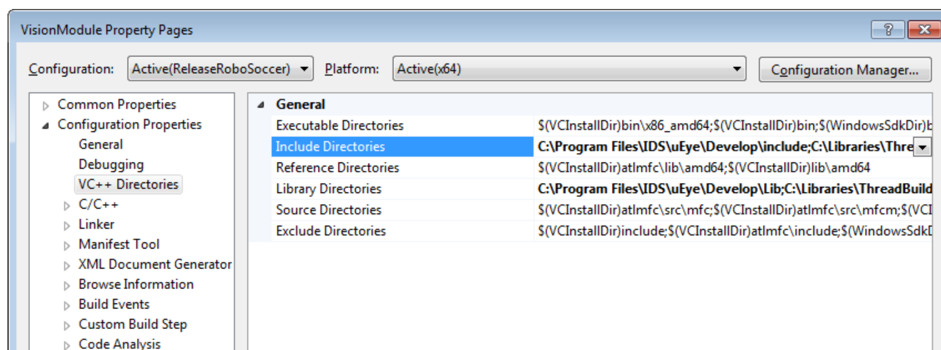
myši na její řádek v seznamu kamer.

A.3 Nastavení rozpoznávacího modulu ve Visual Studiu

Rozpoznávací modul byl vyvíjen v jazyce C++. Zdrojové kódy modulu jsou umístěny ve složce `C:\Robofotbal\VisionModule` a celý projekt ve Visual Studiu lze spustit poklepáním na hlavní soubor projektu `VisionModule.sln`.

Po spuštění se vpravo nachází okno *Solution Explorer* obsahující stromovou strukturu souborů v projektu (viz obrázek A.4). Nastavení projektu se zobrazí kliknutím pravého tlačítka na projekt *VisionModule* a výběrem možnosti *Properties*. Zobrazí se okno, ve kterém lze provádět všechna důležitá nastavení.

Po zobrazení okna s nastavením ve sloupci vlevo se vybere možnost *VC++ Directories*, kde lze nastavovat cesty k hlavičkovým a linkovacím souborům.



Obrázek A.5: Okno s nastavením cest projektu. Kliknutím na šipku u každé položky vpravo a výběrem volby <*Edit...*> lze přidávat a odebírat jednotlivé cesty. V levém horním rohu zároveň lze vybrat aktivní konfiguraci (Typicky *Release* nebo *Debug*), pro kterou se nastavení má vztahovat.

V pravé části okna se pak zobrazí položky s cestami. V levém horním rohu byla nejprve vybrána jako stávající konfigurace *Release*, a zároveň byl překlad nastaven pro 64-bitovou platformu volbou v seznamu nahoře uprostřed (*Platform: x64*) (viz obrázek A.5).

Do seznamu položek *Include Directories* byly přidány následující cesty (v případě, že byly knihovny nainstalovány do jiné složky, než jaké bylo uvedeno v předcházejících podsekcích, bude potřeba je náležitě pozměnit):

```
C:\Program Files\IDS\uEye\Develop\include
C:\Libraries\ThreadBuildingBlocks\tbb41_20130314oss\include
C:\Libraries\opencv\build\include
```

A do seznamu složek s knihovnami pro linkování *Library Directories* bylo přidány položky:

```
C:\Program Files\IDS\uEye\Develop\Lib
C:\Libraries\ThreadBuildingBlocks\tbb41_20130314oss\lib\intel64
   \vc10
C:\Libraries\opencv\build\x64\vc10\lib
C:\Libraries\opencv\build\gpu\x64\vc10\lib
```

Dále bylo třeba nastavit seznam použitých knihoven pro linkování. To lze

učinit v nabídce *Linker* a v její podnabídce *Input*, kde do seznamu *Additional Dependencies* bylo přidáno:

```
opencv_calib3d244.lib
opencv_contrib244.lib
opencv_core244.lib
opencv_features2d244.lib
opencv_flann244.lib
opencv_gpu244.lib
opencv_haartraining_engine.lib
opencv_highgui244.lib
opencv_imgproc244.lib
opencv_legacy244.lib
opencv_ml244.lib
opencv_objdetect244.lib
opencv_photo244.lib
opencv_stitching244.lib
opencv_ts244.lib
opencv_video244.lib
opencv_videostab244.lib
tbb.lib
tbbmalloc.lib
tbbmalloc_proxy.lib
ws2_32.lib
uEye_api_64.lib
```

Tento celý postup bylo nutné ještě jednou celý zopakovat pro nastavení platformy *Debug*. Tu lze vybrat z nabídky *Configuration* vlevo nahoře. Postup nastavení cest k hlavičkovým souborům a knihovnám je stejný, avšak jako knihovny pro linkování bylo třeba uvést:

```
opencv_calib3d244d.lib
opencv_contrib244d.lib
opencv_core244d.lib
opencv_features2d244d.lib
opencv_flann244d.lib
opencv_gpu244d.lib
opencv_highgui244d.lib
opencv_imgproc244d.lib
```

```
opencv_legacy244d.lib
opencv_ml244d.lib
opencv_objdetect244d.lib
opencv_photo244d.lib
opencv_stitching244d.lib
opencv_ts244d.lib
opencv_video244d.lib
opencv_videostab244d.lib
tbb_debug.lib
tbbmalloc_debug.lib
tbbmalloc_proxy_debug.lib
ws2_32.lib
uEye_api_64.lib
ueye_api.lib
```

Nyní by celý projekt měl již být přeložitelný stisknutím klávesy *F6* nebo tlačítkem z menu *Build* → *Build Solution*. Spuštění však ještě selže, protože v adresáři s přeloženým spustitelným souborem ještě chybí DLL soubory z použitých knihoven.

Přeložený projekt se nachází ve složce `x64\Release`, resp. `x64\Debug`, jedná-li se o ladicí konfiguraci. Do těchto složek bylo potřeba nakopírovat DLL soubory s knihovnami. Knihovny pro OpenCV se nachází ve složce:

```
C:\Libraries\opencv\build\x64\vc10\bin
```

Pro konfiguraci *Debug* jsou určeny ty knihovny, které před příponou *.dll* mají písmeno *d* (je-li na výběr ze dvou možností). Knihovny tedy byly roztříděny do náležitých konfiguračních složek rozpoznávacího modulu.

Pro modul GPU se nachází DLL soubory ve vlastní složce:

```
C:\Libraries\opencv\build\gpu\x64\vc10\bin
```

K možnosti využívat OpenCV s podporou GPU, bylo nutné tyto knihovny překopírovat do složky se spustitelným souborem a přepsat *Core* modul, který byl původně zkopírován ze standardní složky s knihovnami OpenCV. Kromě toho bylo ještě potřeba přidat ze složky obsahující CUDA Toolkit soubory:

```
cudart64_42_9.dll
cufft64_42_9.dll
npp64_42_9.dll
```

Tyto knihovny by se měly standardně nacházet ve složce:

```
C:\Program Files\NVIDIA GPU Computing Toolkit\CUDA\v4.2\bin
```

Knihovny pro užití Intel Thread Building Blocks se standardně nachází ve složce:

```
C:\Libraries\tbb41_20130314oss\bin\intel64\vc10
```

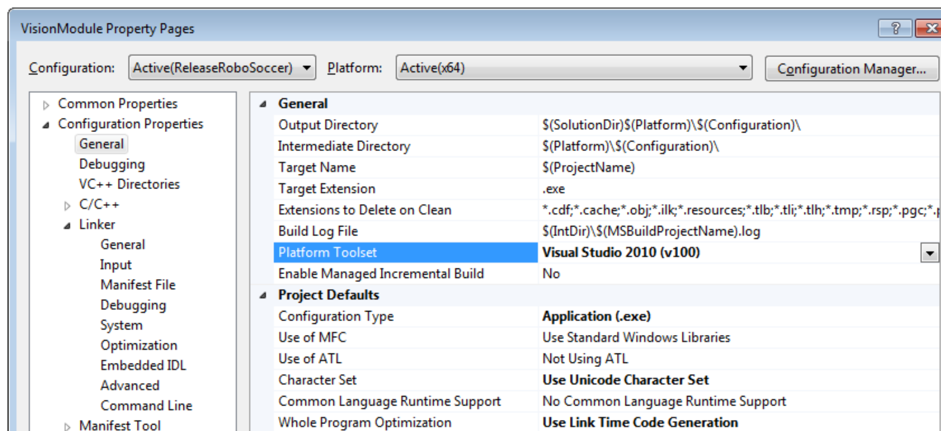
Z ní bylo třeba nakopírovat všechny DLL soubory do složek s konfiguracemi rozpoznávacího modulu. Knihovny zakončené řetězcem *_debug.dll* jsou určeny pouze pro ladicí platformu *Debug*.

Nakonec je ještě potřeba upozornit na to, že je-li pro vývoj využito Visual Studiu 2012, je nutné pro kompilaci využít sadu knihoven z Visual Studia 2010 (je třeba ho mít taktéž nainstalované), protože OpenCV není s novou verzí stále ještě kompatibilní. V nastavení projektu v nabídce *General* lze v podnabídce *Platform Toolset* vybrat možnost *Visual Studio 2010 (v100)*, čímž se k překladu využije starší verze Visual Studia. Okno s nastavení je znázorněno na obrázku A.6.

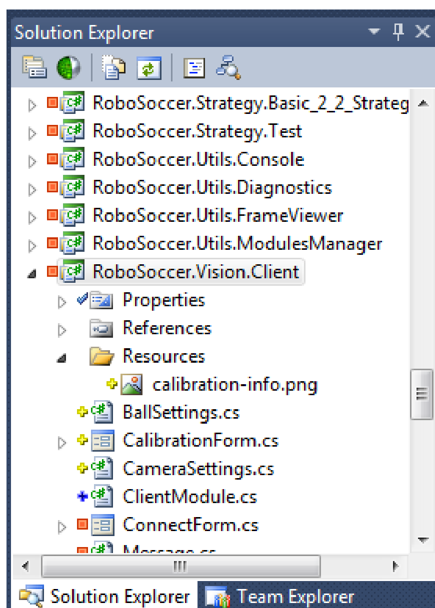
A.4 Nastavení řídicí platformy

Zdrojové kódy řídicí platformy robotického fotbalu byly umístěny do složky `C:\Robofotbal\RoboSoccer` a celý projekt ve Visual Studiu lze spustit poklepáním na soubor `RoboSoccer.sln` v kořenovém adresáři projektu.

Po spuštění se v okně *Solution Explorer* nachází všechny moduly platformy robotického fotbalu. Pro komunikaci s rozpoznávacím serverem byl založen projekt nazvaný *RoboSoccer.Vision.Client* (viz obrázek A.7). Po jeho kliknutí pravým tlačítkem a volbě *Properties* lze získat přístup k nastavení projektu.



Obrázek A.6: Při vývoji ve Visual Studiu 2012, je třeba nastavit *Platform Toolset* na knihovny Visual Studia 2010, protože OpenCV není stále ještě kompatibilní s novou verzí Visual Studia.



Obrázek A.7: Okno se seznamem projektů v řídicím software robotického fotbalu. Stiskem pravého tlačítka na projekt *RoboSoccer.Vision.Client* a volbou *Properties* lze zobrazit nastavení projektu.

Po výběru záložky *Build* v levém menu se zobrazí nabídka s možnostmi překladu. Cílový adresář pro překlad (*Output Directory*) byl nastaven na cestu:

```
..\bin\x86\Release\
```

Standardně se totiž DLL knihovna, která vznikne překladem, umisťuje do adresáře uvnitř daného projektu. S výše zmíněným nastavením se umístí do adresáře se spustitelným souborem celého řídicího software tak, aby jej mohl načíst. Tuto cestu je vhodné v případě problémů se spuštěním řídicí platformy také nastavit i u všech ostatních projektů.

Dále bylo třeba změnit nastavení v hlavní nabídce (nikoli v nastavení projektu) menu *Build* a podnabídce *Configuration Manager*. U všech projektů je nutné zatrhnout položku *Build*, čímž se dá najevo, že se má daný projekt překládat. Dále jako aktivní platforma (*Active solution platform*) byla nastavena možnost *x86*, protože v 64-bitových aplikacích (volba *x64*) Visual Studio mívá potíže s designérem grafických uživatelských rozhraní. Tatáž platforma byla nastavena u všech projektů ve sloupci *Platform* (viz obrázek A.8).

Překlad lze provést stiskem klávesy *F6*. Překlad by se měl provést do adresáře se spustitelným programem, který se nachází relativně od kořenového adresáře na cestě (pro *Release* konfiguraci):

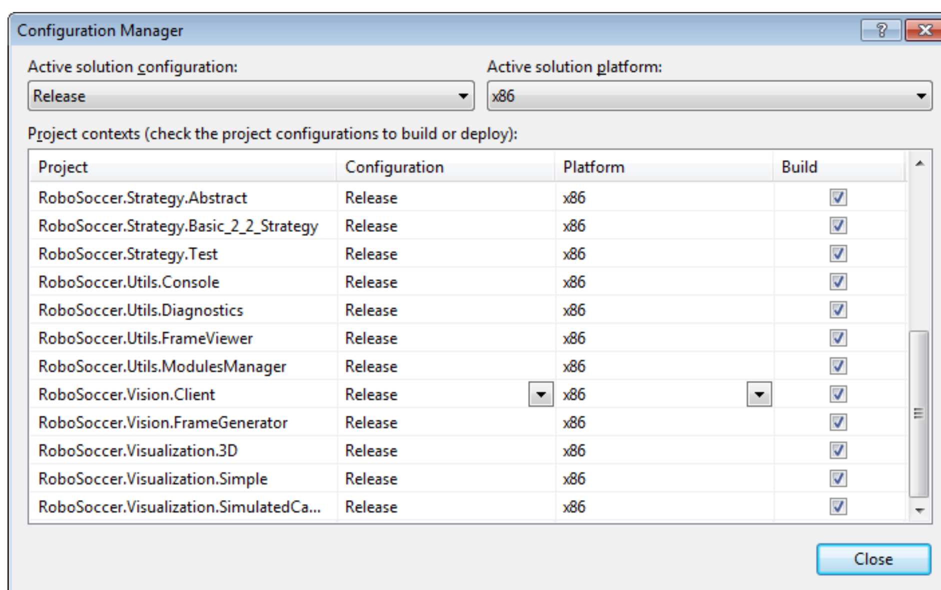
```
bin\x86\Release
```

V případě problémů je vhodné zkontrolovat, zda se v něm nachází přeložené knihovny jednotlivých modulů, zejména knihovna s přeloženým klientem rozpoznávacího modulu:

```
RoboSoccer.Vision.Client.dll
```

V této složce se také nachází konfigurační soubor modulů *config.xml*, který definuje, které moduly má řídicí software načíst. Z toho důvodu je nutné, aby se v něm nacházel řádek:

```
<Module file="RoboSoccer.Vision.Client.dll"/>
```



Obrázek A.8: Nastavení konfigurace překladu u projektů v řídicím software robotického fotbalu. Všechny projekty se musí překládat (zatržená volba *Build*) a jako platforma by měla být nastavena (*x86*).

V této složce by se měl nacházet i konfigurační soubor klienta rozpoznávacího modulu *VisionModuleSettings.xml*. Pokud tomu tak není, bude tento soubor vytvořen při prvním spuštění klienta rozpoznávacího modulu. Může se ale hodit ho překopírovat mezi jednotlivými platformami (Debug a Release), aby nebylo nutné všechny hodnoty znovu pozměňovat při změně platformy pro překlad (každá platforma má svoji složku, takže i vlastní konfigurační soubory).

A.5 Instalace knihoven

Na PC určené pro robotický fotbal byl nainstalován operační systém Microsoft Windows 7 v 64-bitové verzi, pro který je rozpoznávací modul i celá softwarová platforma robotického fotbalu primárně určena. V této sekci je popsán postup instalace všech potřebných knihoven pro správnou funkčnost rozpoznávacího modulu.

A.5.1 Thread Building Blocks

Balík s aktuální verzí (*4.1 Update 3*) knihovny Thread Building Blocks pro operační systém Microsoft Windows byl stažen z adresy:

`http://threadingbuildingblocks.org/sites/default/files/software_releases/windows/tbb41_20130314oss_win.zip`

Obsah archivu byl rozbalen do složky `C:\Libraries`.

A.5.2 CUDA

OpenCV stále ještě nepodporuje platformu CUDA v nejnovější verzi 5, a proto bylo třeba stáhnout starší SDK a ovladače ze stránek:
`https://developer.nvidia.com/cuda-toolkit-42-archive`

Nejprve bylo nutné nainstalovat CUDA Toolkit pro 64-bitové aplikace, které je možné stánout z adresy:

`http://developer.download.nvidia.com/compute/cuda/4_2/rel/toolkit/cudatoolkit_4.2.9_win_64.msi`

Vzhledem k potřebě vývoje a spouštění CUDA aplikací bylo třeba nainstalovat speciální ovladače grafické karty. Ovladač byl stažen pro desktop v 64-bitové verzi z URL:

`http://developer.download.nvidia.com/compute/cuda/4_2/rel/drivers/devdriver_4.2_winvista-win7_64_301.32_general.exe`

Nakonec bylo ještě nainstalováno *GPU Computing SDK*, které obsahuje sadu příkladů a dokumentů, které usnadňují vývoj aplikací pro GPU. Tento krok však není pro funkčnost programů využívající platformu CUDA nutný. SDK bylo staženo z webové adresy:

`http://developer.download.nvidia.com/compute/cuda/4_2/rel/sdk/gpucomputingsdk_4.2.9_win_64.exe`

A.5.3 OpenCV

Rozpoznávací modul byl vyvíjen za použití knihovny OpenCV ve verzi *2.4.4*. V době psaní práce existovala i novější verze *2.4.5*, ta však zatím neobsahuje předkompilované knihovny pro modul GPU. OpenCV lze i zkompilovat ze zdrojových kódů [33], avšak to se ukázalo jako poměrně náročné, neboť úspěšně lze zkompilovat jen určité kombinace verzí Visual Studia, OpenCV a CUDA platformy. Například pod vývojovým prostředím *Visual Studio 2012* nebo za použití platformy CUDA ve verzi 5 se mi nepodařilo zkompilovat žádnou verzi OpenCV. Pro překlad rozpoznávacího modulu je proto použito OpenCV 2.4.4, CUDA 4.2 a Visual Studio 2010.

OpenCV SDK v této verzi lze stáhnout z webové adresy:

`http://sourceforge.net/projects/opencvlibrary/files/opencv-win/2.4.4/`

Z této adresy byl stažen jak balík *OpenCV-2.4.4.exe*, tak i archiv *OpenCV-2.4.4-CUDA-vc10.7z*, který obsahuje přeložené knihovny pro GPU modul OpenCV. Nejprve bylo nainstalováno SDK ze souboru *OpenCV-2.4.4.exe*. OpenCV bylo nainstalováno do složky `C:\Libraries\OpenCV`. Takto nainstalované SDK však obsahuje pouze tzv. *dummy* knihovny pro modul GPU, které nevykonávají žádnou činnost a jsou zde proto, aby se ušetřilo místo. Knihovny přeloženého GPU modulu totiž zabírají téměř 1GB dat. Knihovny s GPU modulem byly rozbaleny do stejné složky, do které bylo nainstalováno OpenCV.

A.5.4 Instalace knihoven kamery

Software pro ovládání kamery a vývoj se nachází na stránkách IDS Imaging:

`http://en.ids-imaging.com/download-ueye.html`

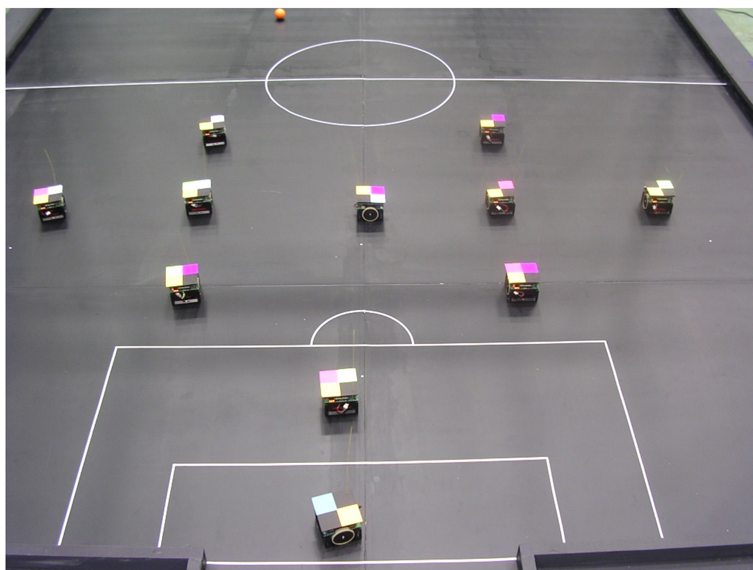
Byl stažen balík *IDS Software Suite 4.22 for Windows (64 bit)*, který obsahuje ovladače i knihovny pro vývoj, z URL:

`http://en.ids-imaging.com/download-ueye.html?file=tl_files/downloads/uEye_SDK/driver/uEye_4.22.00_64.zip`

Z rozbaleného archivu byl spuštěn instalační *.exe* soubor a knihovny byly nainstalovány do standardní složky `C:\Program Files` určené instalátorem.

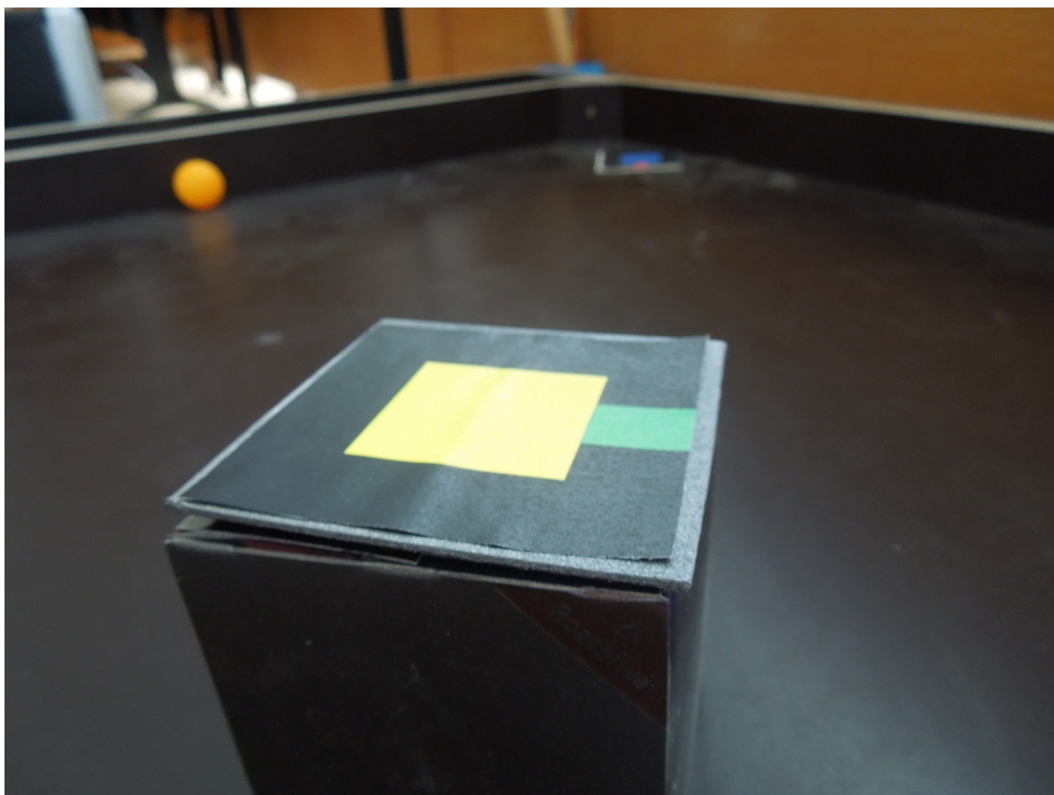
B Obrazové přílohy

Obrázek B.1: Na snímku generovaném perspektivní projekcí (nahoře) jsou vidět i stěny robota a hřiště. Ve snímku generovaném ortogonální projekcí je vidět pouze identifikační štítek a horní plochy stěn hřiště (obrázek dole).



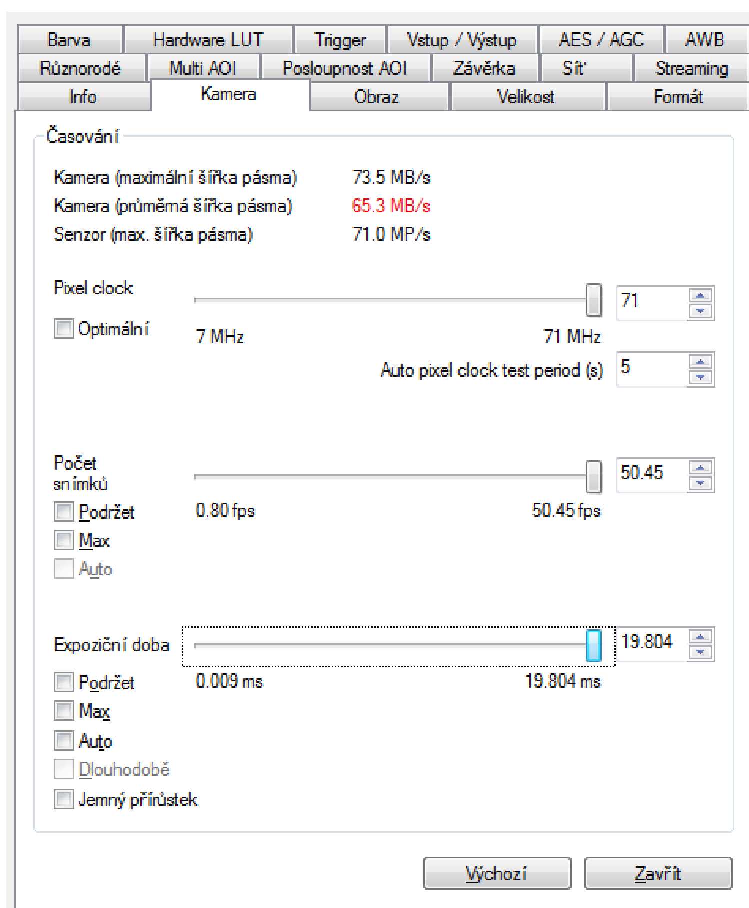
Obrázek B.2: Fotografie z reálné hry v lize *MiroSot*. Na fotografii nahoře je zobrazen detail robotů, na fotografii dole je vidět vzhled hracího stolu a konstrukce pro zavěšení kamery v souladu s pravidly soutěže.

Obrázek B.3: Artefakt vznikající u CMOS čipů ve snímacím módu *Rolling Shutter*. U rychle pohybujícího se předmětu dochází ke zkosení vlivem postupného čtení jednotlivých řádků obrazu ze senzoru v různých časových okamžicích.

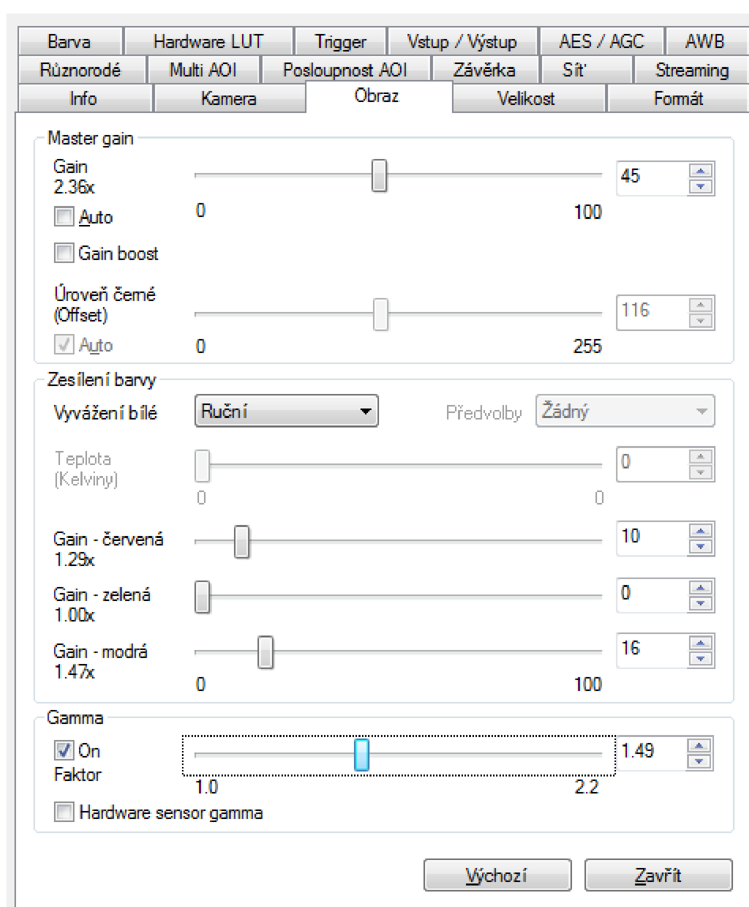


Obrázek B.4: Detail prototypu „robota“ vyrobeného pro účely testování rozpoznávacího modulu. Robot byl vytvořen z papírové krabičky o rozměrech $7 \times 7 \times 7$ cm a na jeho vrchní stranu byl nalepen obrázek s jeho identifikačním štítkem.

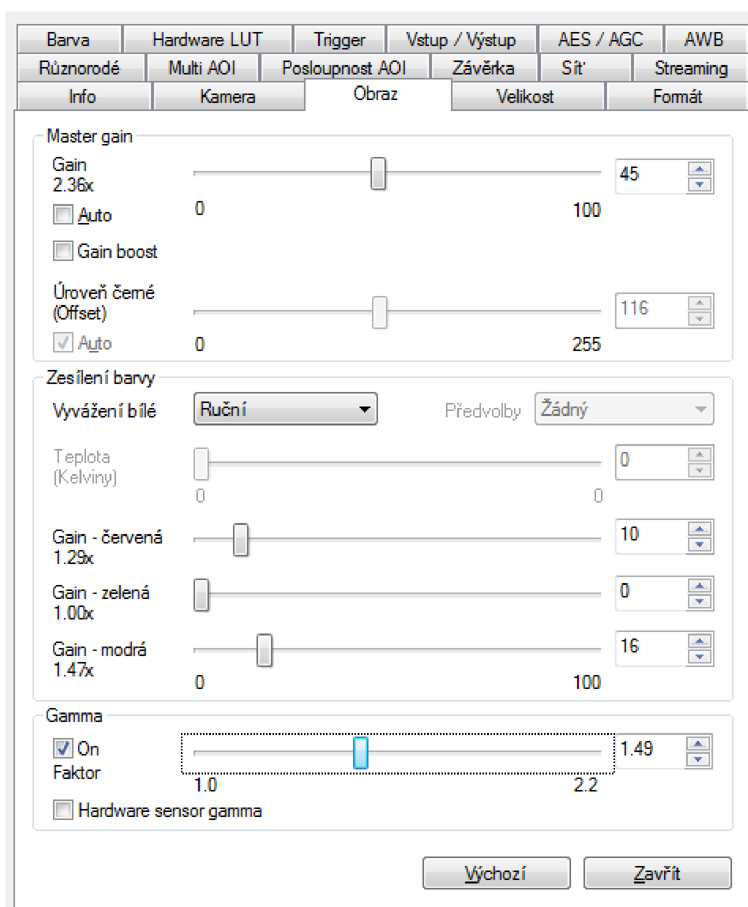
Obrázek B.5: Detail připevnění kamery. Kamera byla umístěna na dřevěnou konstrukci do výšky zhruba 2,5 metru nad hřištěm a připevněna na kabel tak, aby se dala snáze polohovat. Její pozice byla nastavena tak, aby ve snímku byl vidět celý hrací stůl a hrany stolu byly zobrazeny zhruba rovnoběžně s osami snímaného obrazu.



Obrázek B.6: Okno s nastavením snímkovací frekvence a expoziční doby. K dosažení maximální snímkovací frekvence je nutné hodnoty *Pixel Clock* a *Počet snímků* nastavit na maximální hodnotu. Zároveň je vhodné i expoziční dobu zvýšit na maximum.



Obrázek B.7: Okno s nastavením zesílení intenzity snímku a nastavením *gamma korekce.*



Obrázek B.8: Okno s nastavením formátu snímku generovaného kamerou.