FAKULTA STROJNÍ TUL

Bakalářská práce

Digitalizace pracovního prostoru pomocí počítačového vidění

Studijní program: B0715A270008 Strojírenství
Autor práce: **Jakub Šír**
Vedoucí práce: Ing. Andrii Shynkarenko, Ph.D.
Katedra výrobních systémů a automatizace

Liberec 2022

FAKULTA STROJNÍ TUL

Zadání bakalářské práce

Digitalizace pracovního prostoru pomocí počítačového vidění

Jméno a příjmení:	**Jakub Šír**
Osobní číslo:	S19000195
Studijní program:	B0715A270008 Strojírenství
Zadávající katedra:	Katedra výrobních systémů a automatizace
Akademický rok:	2022/2023

Zásady pro vypracování:

Cílem bakalářské práce je návrh softwaru, který řeší problematiku manuálního zadávání počátečních parametrů pro automatizované procesy, například polohu předmětů v pracovním prostoru. Daný software by měl být kompatibilní s jinými softwary. Samotná detekce by měla být realizována pomocí počítačového vidění. Téma má aplikační potenciál a je určeno pro zvýšení automatizace a snížení lidského faktoru v automatizovaných procesech.

1. Proveďte rešerši dané problematiky. Seznamte se s technologií počítačového vidění.
2. Na základě studie navrhněte vhodný postup a model pro realizaci.
3. Realizujte návrh, naprogramujte navrhnuté softwarové řešení.
4. Zajistěte kompatibilitu s jinými softwary.
5. Otestujte navržené zařízení.

Rozsah grafických prací:	dle potřeby
Rozsah pracovní zprávy:	cca 40 stran
Forma zpracování práce:	tištěná/elektronická
Jazyk práce:	Čeština

Seznam odborné literatury:

[1] MOORE, Alan D. *Python GUI Programming with Tkinter: Develop responsive and powerful GUI applications with Tkinter.* [online]. Birmingham: Packt Publishing, 2018 [vid. 2021-10-13]. ISBN 978-1-78883-568-8.
[2] LUTZ, Mark. *Programming Python*. Fourth edition. Beijing Boston Farnham Sebastopol Tokyo: O'Reilly, 2019. ISBN 978-0-596-15810-1.
[3] TALVERDI, P. *CNC lathe G-Code and M-Code illustrative handbook*. ? Lulu Enterprises, ///. ISBN 978-0-557-64836-8.
[4] KAEHLER, Adrian a Gary R. BRADSKI. *Learning OpenCV 3: computer vision in C++ with the OpenCV library*. First edition, Second release. Sebastopol, CA: O'Reilly Media, 2017. ISBN 978-1-4919-3799-0.
[5] LI, Qing. *Biomaterials for implants and scaffolds*. New York, NY: Springer Berlin Heidelberg, 2016. ISBN 978-3-662-53572-1.
[6] STANISLAV, Lukáš. *PRODUKCE NANOVLÁKEN METODOU TAŽENÍ*. Liberec, 2015. Disertační práce. Technická Univerzita v Liberci.

Vedoucí práce:	Ing. Andrii Shynkarenko, Ph.D. Katedra výrobních systémů a automatizace
Datum zadání práce:	20. listopadu 2022
Předpokládaný termín odevzdání:	20. května 2024

L.S.

doc. Ing. Jaromír Moravec, Ph.D.
děkan

Ing. Petr Zelený, Ph.D.
vedoucí katedry

V Liberci dne 20. listopadu 2022

Prohlášení

Prohlašuji, že svou bakalářskou práci jsem vypracoval samostatně jako původní dílo s použitím uvedené literatury a na základě konzultací s vedoucím mé bakalářské práce a konzultantem.

Jsem si vědom toho, že na mou bakalářskou práci se plně vztahuje zákon č. 121/2000 Sb., o právu autorském, zejména § 60 – školní dílo.

Beru na vědomí, že Technická univerzita v Liberci nezasahuje do mých autorských práv užitím mé bakalářské práce pro vnitřní potřebu Technické univerzity v Liberci.

Užiji-li bakalářskou práci nebo poskytnu-li licenci k jejímu využití, jsem si vědom povinnosti informovat o této skutečnosti Technickou univerzitu v Liberci; v tomto případě má Technická univerzita v Liberci právo ode mne požadovat úhradu nákladů, které vynaložila na vytvoření díla, až do jejich skutečné výše.

Současně čestně prohlašuji, že text elektronické podoby práce vložený do IS/STAG se shoduje s textem tištěné podoby práce.

Beru na vědomí, že má bakalářská práce bude zveřejněna Technickou univerzitou v Liberci v souladu s § 47b zákona č. 111/1998 Sb., o vysokých školách a o změně a doplnění dalších zákonů (zákon o vysokých školách), ve znění pozdějších předpisů.

Jsem si vědom následků, které podle zákona o vysokých školách mohou vyplývat z porušení tohoto prohlášení.

16. listopadu 2022 Jakub Šír

Digitalizace pracovního prostoru pomocí počítačového vidění

Abstrakt

Tato práce se zabývá problematikou využití strojového vidění při procesu výroby syntetických vláken metodou tažení. Cílem práce je návrh a realizace systému strojového vidění pro detekci přípravků v pracovním prostoru a zjištění jejich souřadnic vzhledem k souřadnicovému systému stroje k produkci syntetických vláken. Získané souřadnice jsou následně předány řídícímu systému stroje prostřednictvím souboru JSON. V práci jsou nejdříve popsány různé metody detekce objektů a také je vysvětleno, proč a jak kalibrovat kameru. Rešeršní část se zaměřuje na stručné shrnutí dostupných komerčních systémů strojového vidění v průmyslu a hledání potenciálních alternativ pro tuto problematiku. Následuje detailní popis realizace samotného systému, včetně popisu ovládání systému, který může sloužit jako příručka pro obsluhu. V poslední části je realizovaný systém otestován k ověření jeho funkčnosti.

Klíčová slova: strojové vidění, detekce objektů, OpenCV, ArUco, metoda tažení, drawing, Python

Digitizing a workspace with computer vision

Abstract

This thesis solves the issue of usage of machine vision in the process of synhetic fibers production using the drawing method. Main objective of the thesis is to develop machine vision system that detects devices in a workspace of a machine used in synhetic fibers production and determining their coordinates with respect to the machine's coordinate system. The acquired data is then sent to the machine's control system through a JSON file. In theoretical part, various methods of object detection are described, as well as an explanation of why and how to calibrate the camera. The research part focuses on a brief summary of available commercial machine vision systems and the search for potential alternatives for this issue. This is followed by the main part: implementation of the system itself, including a description of how to operate the system, which can serve as an operator's manual. In the last part, the implemented system is tested to verify its functionality.

Keywords: machine vision, object detection, OpenCV, ArUco, fiber drawing , Python

Poděkování

Mé poděkování patří Ing. Andrii Shynkarenkovi, Ph.D., za odborné vedení a ochotu, kterou mi v průběhu zpracování bakalářské práce věnoval. Zároveň bych rád poděkoval rodině za trpělivost a poskytnutou podporu při studiu.

Obsah

Seznam obrázků

Seznam tabulek

Seznam zdrojových kódů

Seznam zkratek

AIO	All in One, Vše v jednom
ArUco	Augmented reality University of Cordoba; Rozšířená realita, univerzita v Cordobě
BRIEF	Binary Robust Independent Elementary Features, Binární robustní nezávislé elementární rysy
ChArUco	Checkerboard and ArUco markers , Šachovnice se znaky ArUco
CPU	Central Processing Unit, Centrální procesor
CVL	Cognex vision library, Cognex knihovna pro zpracování obrazu
FCL	Fully Connected Layer, Plně propojená vrstva
FPS	Frames Per Second, Snímky za sekundu
FLANN	Fast Library for Approximate Nearest Neighbors, Rychlá knihovna pro aproximativní vyhledávání nejbližších sousedů
GPU	Graphics Processing Unit, Grafický procesor
Mpx	megapixel
px	pixel
RAM	Random Access Memory, Paměť s přímým přístupem
RCNN	Region-based Convolutional Neural Network, Konvoluční neuronová síť založená na regionech
RGB	Red, Green, Blue; Červená, Zelená, Modrá
ORB	Oriented FAST and Rotated BRIEF, Orientovaný FAST a natočený BRIEF
SIFT	Scale-Invariant Feature Transform, Transformace rysů nezávislá na měřítku
SSD	Single Shot MultiBox Detector, Detektor jednoho snímku s více boxy
SURF	Speeded-Up Robust Features, Zrychlené robustní charakteristické rysy
USB	Universal Serial Bus, Univerzální sériová sběrnice
VRAM	Video RAM
YOLO	You Only Look Once, Podíváš se jen jednou

1 Úvod

Když si uvědomíme, že je zrak nejdůležitějším smyslem pro většinu lidí, není divu, že se technologie strojového vidění stává klíčovým faktorem moderního průmyslu. Strojové vidění umožňuje strojům vnímat a interpretovat vizuální data, jako jsou obrázky a videa, a využít tuto informaci k řešení úloh v průmyslovém prostředí. Díky pokročilým algoritmům a rostoucímu množství dat, které jsou dnes k dispozici, mohou průmyslové podniky využívat strojové vidění k různým účelům, jako je například řízení kvality výroby, rozpoznávání defektů výrobků a optimalizace logistiky v průmyslových procesech. S tím, jak se technologie strojového vidění stává stále pokročilejší, mohou průmyslové podniky získávat výhody v konkurenci, zvyšovat produktivitu a dosahovat vynikajících výsledků v různých oblastech průmyslu.[1]

Syntetická vlákna v měřítku mikrometrů až nanometrů se stávají stále více populárními díky svému širokému spektru využití, ať už se jedná o biomedicínské aplikace, vysokoúčinné filtry, nebo materiály s vynikajícími vlastnostmi pro elektroniku a optiku. Vědci po celém světě se snaží nejen zlepšit výrobu nanovláken, ale také pochopit fyzikální a chemické vlastnosti těchto materiálů a jejich interakce s okolním prostředím. Zajímavé výzkumy například ukázaly, že nanovlákna mohou být použita jako účinný způsob záchytu škodlivých plynů a částic v ovzduší, čímž přispívají k ochraně životního prostředí. A co více, nové technologie výroby syntetických vláken umožňují vytvářet materiály s přesně definovanou strukturou a vlastnostmi, což otevírá dveře k novým možnostem, například v oblastech materiálového nebo biomedicínského inženýrství. Celkově lze tedy říci, že má studium mikrovláken a nanovláken potenciál přinést mnoho zajímavých a přínosných objevů do vědy a praxe.[2]

Cílem práce je tvorba systému strojového vidění, který by vedl ke zjednodušení přípravy při procesu tvorby syntetických vláken metodou tažení v laboratorních podmínkách. Při tomto procesu je nutno manuálně zadávat souřadnice počátečních a koncových fixačních ploch a v návaznosti na to tyto plochy umisťovat relativně přesně do pracovního prostoru, respektive souřadnicového systému stroje, abychom věděli, kde přesně se nachází. Počítačové vidění nabízí způsoby jak automaticky detekovat tyto plochy a zjistit jejich souřadnice vzhledem k souřadnicovému systému stroje. To by umožnilo ulehčení práce při přípravě procesu a případně by mohlo vést k částečné až úplné automatizaci procesu.

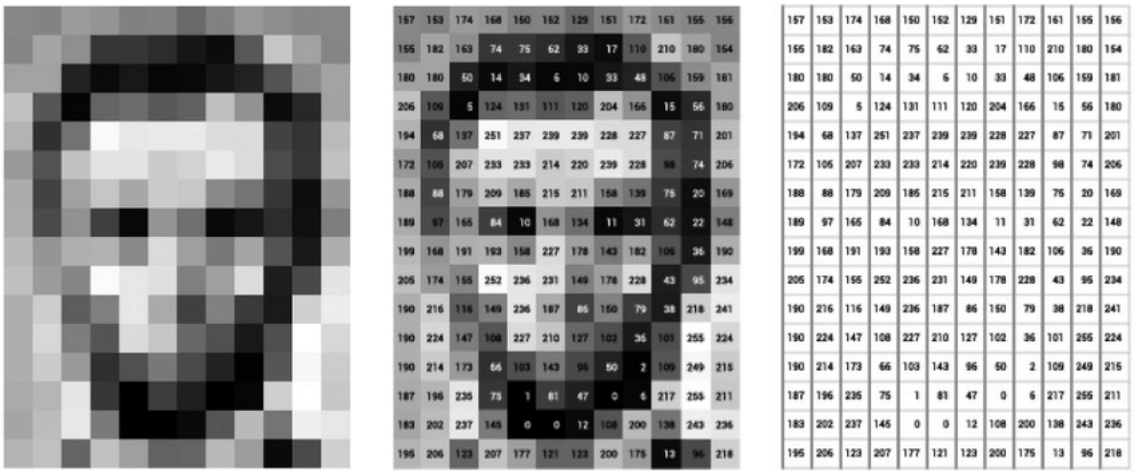
2 Teoretický základ

K navržení a zkonstruování systému strojového vidění, se je nutné seznámit se základními principy zobrazováni obrazu a počítačového vidění. Stručně bude popsáno co je to strojové vidění a jaké přístupy k řešení můžeme zaujmout. Následuje seznámení s problematikou kalibrace kamery a se samostatnou problematikou tažení syntetických vláken.

2.1 Počítačové vidění

Počítačové vidění je odvětví zabývající se transformací dat získaných ze stacionárního obrazu či videa na rozhodnutí nebo transformací dat do nové podoby za nějakým dalším účelem. Rozhodnutí by například mohlo znamenat:"Na obrázku je šroub." Nebo: "V pracovním prostoru angulárního robotu se nachází nechtěný předmět." Nová reprezentace pak může znamenat převod původního obrazu do obrazu převráceného nebo z obrazu barevného do obrazu černo-bílého apod.[3]

Na rozdíl od oka a mozku, kde dochází ke komplexním biologickým procesům, dostane počítač z kamery nebo z obrazu sít čísel - matici. Každý prvek matice reprezentuje jeden pixel. Hodnota prvku pak u černo-bílého obrazu určuje odstín šedi(viz obr.2.1).[3]



Obr. 2.1: Reprezentace obrazu pomocí matice[4]

U barevného obrazu prvek matice obsahuje vektor o 3 číslicích. Například u

barevného modelu RGB určuje první číslo odstín červené barvy, druhé odstín zelené a třetí odstín modré. [3]

Úkolem počítačového vidění je taková data zpracovat prostřednictvím velké škály algoritmů a výsledná data dále využít v požadované aplikaci. Mezi takové aplikace, ve kterých se počítačové vidění čím dál častěji uplatňuje, může být: rozpoznávání lidských obličejů, registračních značek automobilů, předmětů, v samo řiditelných vozidlech, robotice, medicíně.[5]

2.2 Strojové vidění

Strojové vidění by se dalo nazvat podoborem počítačového vidění, které se snaží využívat této existující technologie k řešení problémů reálného světa. Nyní se hojně podílí na rozmachu automatizace v průmyslu,logistice,dopravě a dalších odvětvích.

Ve výrobě dovoluje strojové vidění mnohem větší flexibilitu celé výrobní linky, například odpadá nutnost přemisťovat a orientovat jednotlivé součásti do přesných pozic, aby je robot mohl uchopit - strojové vidění předá robotu informaci, kde se součást nachází a jak je orientovaná. Vysoká flexibilita může do budoucna přinášet možnost využití jedné výrobní linky na výrobu několik různých výrobků. Nic takového doposud nebylo běžné, většinou z důvodu ekonomické nevýhodnosti. Strojové vidění také nabízí možnost monitorování v reálném čase, například tak odpadá nutnost přesné kadence linky - můžeme sledovat aktuální množství polotovarů v konkrétních místech výrobní linky a na základě toho pak dynamicky upravovat chod linky. Takto získaná data se dále dají využít k analýze a následné optimalizaci výrobního procesu. V návaznosti na pokroky v oblasti hlubokého učení může strojové vidění provádět komplexní operace a efektivně se přizpůsobit novým situacím, což může celému systému dodávat o to více flexibility v porovnání s tradičními systémy.[5]

Díky vysoké poptávce se také stává důležitou technologií pro sledování kvality výrobků. Již delší dobu se ve výrobě vyskytují automatizované stroje nebo výrobní linky, které mají za úkol rozpoznávat zmetky, které by jinak skončily u zákazníka či zamezit jejich dalšímu použití ve vlastní výrobě.[5]

V neposlední řadě se automatizace jako celek stává ekonomicky výhodnějším řešením i přes vyšší počáteční náklady, jak ve výrobě tak v inspekci. Vyžaduje méně lidské práce a může běžet nepřetržitě ve vysoké rychlosti bez nutnosti přestávek.

Mimo průmyslové použití se strojové vidění také hojně využívá v logistice, například ke skenování čárových kódů na zásilkách nebo ke zjištění absence balíku na dopravníku, atd. V dnešní době již existují plně autonomní sklady, kde se hojně využívá umělé inteligence, strojového vidění, velkých dat, atd.[6]

2.3 Algoritmy pro detekci objektů

Detekce objektů je jedním z fundamentálních úkolů strojového vidění, umožňuje nám totiž popsat, kde se jaký objekt nachází ve sledovaném obrazu. Tento proces obecně obsahuje dvě části - detekci polohy a klasifikaci. Detekce polohy, jak již

název napovídá, slouží k určení polohy předmětu. Klasifikace pak slouží ke zjištění, o jaký objekt se jedná - např. zda jde o šroub nebo o matici.

K detekci se využívají již vytvořené algoritmy, které by se daly rozdělit do několika kategorií. Mezi méně komplexní, avšak neméně užitečné přístupy patří detekce pomocí kontur a detekce pomocí substrakce pozadí. Mezi více komplexní pak algoritmy využívající charakteristických rysů a algoritmy využívající metod strojového učení.

Vzhledem k tomu, že bude v této práci využita knihovna OpenCV(viz kapitola 4.4.4), budou k jednotlivým algoritmům odkázány funkce právě z této knihovny.

2.3.1 Detekce pomocí kontur

K jednoduché detekci předem daného předmětu na jednoduchém pozadí lze využít detekce kontur. Můžeme vytvořit algoritmus, který bude hledat jen již předdefinované kontury. Takový přístup je vhodný v okamžiku, kdy víme, že budeme hledat například jen dva předměty, které budou mít ve své stálé pozici vždy stejný tvar, například krychle a koule. Konturu si lze jednoduše představit jako křivku spojující všechny body se stejným rozpětím barev nebo intenzity.[7] V knihovně OpenCv slouží k hledání kontur funkce:cv.findContours()[8] a jejich následnému vykreslení funkce: cv.drawContours()[9].

Obr. 2.2: Znázornění detekce kontur[10]

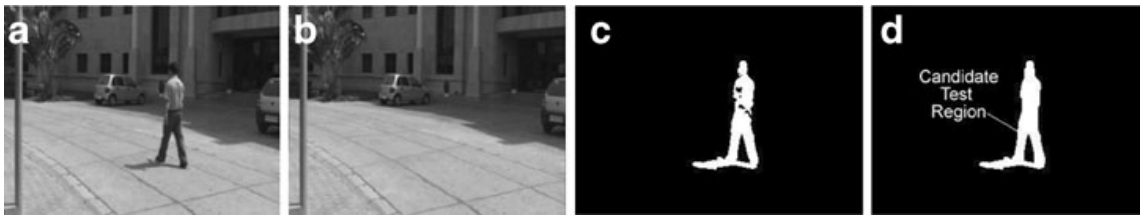
2.3.2 Detekce pomocí substrakce pozadí

Obecně je jako pozadí chápána oblast sledovaného prostoru, která není hlavním předmětem zájmu. Popředím je pak tedy předmět zájmu - hledaný/sledovaný předmět. V případě této metody je pozadím neměnná část obrazu a popředím je pohybující se předmět - předpokladem této metody je tedy fakt, že se hledané předměty pohybují.

K tomu abychom zjistili, které předměty se pohybují, potřebujeme vícero po sobě jdoucích snímků - videozáznam. Algoritmus pak porovnává referenční snímek(viz

obr.2.3b), kde se žádný pohybující předmět nenachází(většinou několik prvních snímků videozáznamu), s právě získávanými snímky(viz obr.2.3a). V místech obrazu, kde se hodnoty liší od referenčního snímku více než je nastavená prahová hodnota se pravděpodobně nachází pohybující se předmět(viz obr.2.3c). Algoritmus pak odstraní statické pozadí a vykreslí pouze detekovaný předmět(viz obr.2.3d).

Využitím abstrakce pozadí je možné detekovat pohybující se objekty bez nutnosti použití dalších algoritmů k detekci a následnému trasování objektů. [11]



Obr. 2.3: Jednotlivé kroky při substrakci pozadí[12]

2.3.3 Detekce pomocí klasických přístupů detekce charakteristických rysů

Jak již název napovídá, jde o klasické algoritmy využívající charakteristických rysů, hledaných a popsaných pomocí matematických metod. Charakteristické rysy mohou být -rohy, hrany nebo další jedinečné vzory, které je možné v obraze nalézt a identifikovat. Většina algoritmů využívá jako charakteristické body rohy. Rohy lze v obraze popsat jako body, kde ve všech směrech dochází k výrazné změně intenzity obrazu. Jedná se o původní přístup detekce objektů před rozvojem umělé inteligence. Většina z těchto algoritmů byla vytvořena před rokem 2010. Obecný popis fungování těchto algoritmů by se dal rozdělit do následujících kroků:[13]

1.**Extrakce charakteristických rysů:** V tomto kroku algoritmus identifikuje a extrahuje charakteristické rysy ze vstupního obrazu. Tyto rysy mohou být také použity jako referenční body k trasování objektu. K extrakci charakteristických rysů bylo postupně vymyšleno několik algoritmů, např. metody: SIFT, SURF, ORB (viz níže).

2.**Porovnání charakteristický rysů:** Extrahované rysy jsou porovnány se sadou charakteristických rysů získaných z referenčního snímku, který může obsahovat jeden nebo více objektů zájmu. Algoritmus využívá tohoto srovnání k vyhodnocení pozic shodných rysů ve vstupním obraze. Tento krok se v angličtině nazývá Feature matching (viz níže).

3.**Lokalizace objektu:** Algoritmus využije těchto pozic ke stanovení přibližné pozice/případně i tvaru výsledného objektu na vstupním obraze. Pozice objektu je pak znázorněna vykreslením ohraničujícího obdélníku kolem objektu.

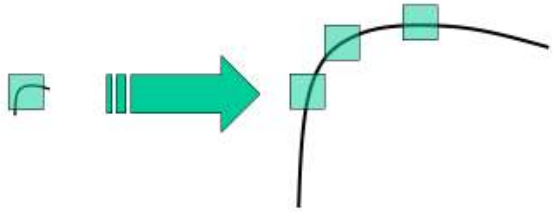
4.**Klasifikace objektu:** V některých případech může algoritmus využít extrahovaných rysů k určení o jaký typ objektu se jedná nebo do jaké třídy spadá.

Harris Corner Detector: je prvním algoritmem, který se snažil lokalizovat rohy v obraze, vytvořený Chrisem Harrisem a Michaelem Stephensenem v roce 1988.

Vychází z jednoduché myšlenky, že lze rohy popsat jako body, kde dochází ve všech směrech k výrazné změně intenzity. Algoritmus je pak založený na hledání této změny intenzity vztažené ke změně polohy ve všech směrech. Na základě těchto dat určí, zda se v dané oblasti/okně vyskytuje roh, hrana nebo rovina.[13] Knihovna OpenCV obsahuje funkci, která hledá rohy právě tímto způsobem: cv.cornerHarris()[14].

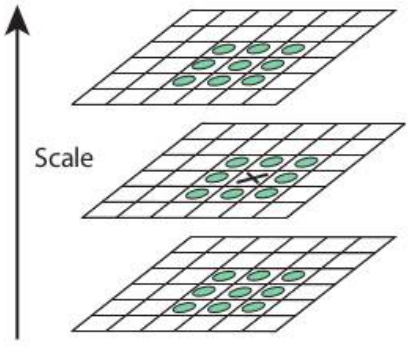
SIFT:

S čím však Harris corner detector nepočítá, je škálování. Roh nemusí být rohem pokud dojde ke změně měřítka obrazu. To lze znázornit na následujícím obrázku. Roh na snímku o malé velikosti vyskytující se v malém okně se při zachování velikosti okna a zvetšeni snímku stane přímkou.



Obr. 2.4: Vliv měřítka[13]

Tento problém řeší algoritmus SIFT:Scale-Invariant Feature Transform. Funguje na základě hledání klíčových bodů v obraze, které jsou výrazné a neměnné vůči změnám měřítka, rotace a osvětlení. Toho je docíleno vytvořením měřítko-prostorové reprezentace obrazu pomocí Gaussových filtrů. Klíčové body jsou identifikovány nalezením lokálních extrémů v rozdílech mezi Gaussovými obrazy v různých měřítkách.Například jeden pixel v obrázku je porovnán s jeho 8 sousedními pixely, stejně jako 9 pixelů v dalším měřítku a 9 pixelů v měřítku předchozím. Pokud se jedná o lokální extrém, jedná se o potenciální klíčový bod. V podstatě to pak znamená, že je klíčový bod nejlépe zastoupen v daném měřítku, ve kterém je nalezen. Knihovna OpenCV obsahuje několik funkcí pro detekci pomocí SIFT.[15]



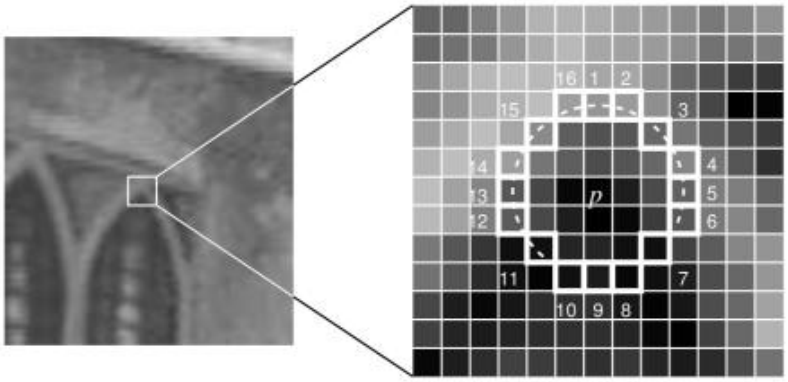
Obr. 2.5: Znázornění pixelů v jednotlivých vrstvách měřítek[13]

Jakmile jsou body identifikovány, SIFT je zpřesní pomocí procesu známého jako lokalizace klíčových bodů. Nakonec je pro každý klíčový bod vypočítán deskriptor, který kóduje místní orientace gradientu a velikosti obrazu v jeho blízkosti.[15]

SURF: Stejně jako SIFT hledá SURF klíčové body, které jsou výrazné a neměnné vůči změnám měřítka, rotace. SURF je v podstatě vylepšeným SIFTem. Hlavní výhodou SURFU je jeho výpočetní efektivnost, SURF využívá rychlejší aproximace operátoru Laplacian of Gaussian používaného v SIFT, což dovoluje mnohem rychlejší zpracovaní obrazu. SURF navíc používá kompaktnější deskriptor, než který využívá SIFT, což také přispívá k vyšší rychlosti zpracování. V jednoduchosti je tedy SURF rychlostním vylepšením algoritmu SIFT při zachování přesnosti detekce. Knihovna OpenCV obsahuje několik funkcí pro detekci pomocí SURF[16].

FAST: Předchozí algoritmus je rychlý, ale není dostatečně rychlý k použití pro aplikace v reálném čase. Jako řešení tohoto problému byl navržen algoritmus FAST (Features from Accelerated Segment Test).

Algoritmus funguje tak, že vybere jeden pixel v obrázku a poté porovná jeho intenzitu se sousedními pixely nacházející se v kruhovém poli(viz Obr.2.6). Pokud se intenzita liší u dostatečného počtu pixelů(jsou tmavší nebo světlejší), je centrální bod klasifikován jako charakteristický bod. FAST je navržen jako výpočetně efektivní alternativa k předchozím algoritmům a lze jej použít pro aplikace v reálném čase, jako je například sledování a navigace robotů. [17]



Obr. 2.6: Kruhové pole metody FAST[13]

BRIEF: (Binary Robust Independent Elementary Features) je binární deskriptor charakteristických rysů. To znamená, že charakteristické rysy popisuje v binárním kódu spíše než tradičně pomocí číselných hodnot. To mu dodává rychlost a efektivnost pro aplikace v reálném čase. Nedokáže však dobře pracovat s různým natočením a změnou měřítka, tím pádem se může stát, že bude jeden a ten samý prvek reprezentován odlišně v závislosti na jeho orientaci a velikosti. Důležité je zdůraznit, že jde o DESKRIPTOR, ne o detektor - neobsahuje žádnou metodu k detekci a lokalizaci charakteristických rysů, pouze k jejich popisu. Je ho tedy nutné použít v kombinaci s algoritmem, který takovou metodu obsahuje, například SURF nebo FAST. [18]

ORB: (Oriented FAST and Rotated BRIEF) je právě kombinací předchozího deskriptoru BRIEF a detektoru FAST. ORB funguje tak, že nejprve detekuje klíčové body(rohy) v obraze pomocí algoritmu FAST. Orientace každého z těchto bodů je dána vektorem s počátkem v detekovaném rohu, procházejícím bodem s nejvyšší intenzitou(v okolí rohu). Deskriptor je pak vypočítán pomocí párů pixelů kolem klíčového bodu, obdobně jako v původním algoritmu BRIEF. Na rozdíl od samostatného BRIEF však ORB bere v úvahu zjištěnou orientaci klíčových bodů, což zaručuje neměnnost vůči rotaci.[19]

Celkově je ORB oblíbeným algoritmem díky své robustnosti, účinnosti a je navíc volně dostupný. Knihovna OpenCV obsahuje několik funkcí pro detekci pomocí ORB[20].

Feature matching: neboli porovnávání charakteristických rysů, jíž zmíněné v bodě 2, slouží k porovnání detekovaných charakteristických rysů mezi dvěma nebo vícero snímky. Příkladem může být Brute-Force matcher nebo FLANN matcher, které lze použít i skrze funkce v knihovně OpenCV.[13]

2.3.4 Detekce pomocí neuronových sítí

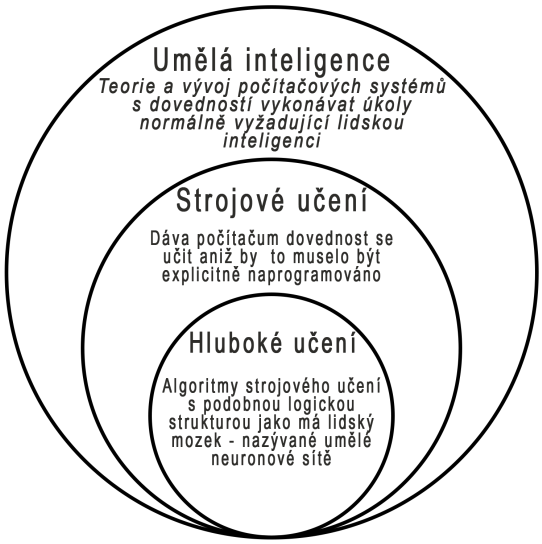
Neuronové sítě jsou nedílnou součásti hlubokého učení,které je podoborem strojového učení. Strojové učení je odvětvím umělé inteligence a soustředí se na tvorbu

systémů, které jsou schopny automaticky zlepšovat své výsledky pomocí zkušeností. Algoritmy strojového učení využívají statistických metod, díky kterým se může počítač učit s využitím velkého množství dat, aniž by musela být taková schopnost explicitně naprogramována.[21]

Hluboké učení by se dalo popsat tak, že trénuje neuronovou síť složenou z více vrstev tak, aby se naučila předpovídat nebo rozhodovat na základě velkého souboru vstupních dat. Neuronová síť se skládá z propojených uzlů neboli „neuronů“ organizovaných do vrstev, kde každá vrstva zpracovává a transformuje vstupy z vrstvy předchozí.[22]

V posledních letech došlo k rapidnímu pokroku v oblasti hlubokého učení, zvláště díky dostupnému, zvyšujícímu se výpočetnímu výkonu. To přispělo k rychlému vývoji algoritmů pro detekci objektů pomocí této technologie.

Obecný postup implementace hlubokého učení by se dal shrnout do několika následujících bodů:[23][24]



Obr. 2.7: Odvětví umělé inteligence

1.**Shromáždění dat:** Prvním krokem je shromáždění velkého souboru dat. V tomto případě jde o snímky obsahující anotace. Anotace zahrnuje ohraničující obdélník kolem objektu a slovní štítek třídy(např. automobil, motocykl, letadlo, atd.), do které objekt spadá.

2.**Příprava dat:** Dalším krokem je přerozdělení souboru dat do dílčích souborů určených pro trénink, validaci a testovaní.

3.**Návrh architektury neuronové sítě:** V tomto kroku dochází k návrhu sítě modelu hlubokého učení-to zahrnuje výběr typu neuronové sítě. Počet vrstev, aktivační funkce a další hyperparametry.

4.**Trénování:** Model je trénován na předem připraveném souboru dat z kroku 2,

přičemž je pod dohledem člověka využit učící algoritmus. Během tréninku se model naučí mapovat vstupní snímky k odpovídajícím anotacím.

5.**Validace:** Po tréninku je na model nasazen soubor dat určených k validaci. To slouží ke zjištění přesnosti modelu a ke zjištění případných problémů spojených s pře-určeností/generalizací nebo pod-určeností modelu.

6.**Testování** Nakonec je model otestován. Je tak zjištěn výkon modelu na datech, se kterým se v minulosti ještě nesetkal.

7.**Implementace:** V případě úspěšné validace a dostatečném výkonu při testovaní, může být model použit ke svému účelu v reálném světě.

Algoritmy k detekci objektů postavené na hlubokém učení řeší dva navazující úkoly. Prvním úkolem je nalézt a lokalizovat libovolný počet objektů nacházejících se na vstupním snímku. Druhým úkolem je zvlášť klasifikovat(určit jeho třídu) každý objekt a odhadnout jeho velikost. Detektory se dělí na jednofázové a dvoufázové. Dvoufázové detektory plní tyto úkoly ve dvou po sobě jdoucích krocích, jednofázové pak naopak pouze v jednom kroku. Výhodou jednofázových je vyšší rychlost za cenu snížené přesnosti oproti detektorům dvoufázovým.[25]

Dvoufázové detektory:

R-CNN: Region-based Convolutional Neural Network je jedním z prvních významných dvoufázových detektorů využívajících konvulačních neuronových sítí vyvinutý v roce 2015. V prvním kroku algoritmus rozdělí vstupní snímek na 2000 oblastí pomocí algoritmu zvaného Selective Search algorithm. Ten vytvoří velké množství prvotních podoblastí, které jsou následně za různých pravidel zredukovány do větších oblastí -> soubor 2000 oblastí.[26]

V druhém kroku je tento soubor vložen do konvulační neuronové sítě, která slouží jako extraktor charakteristických rysů. Charakteristické rysy jsou vstupem pro algoritmus zvaný Support Vector Machines (metoda podpůrných vektorů), který predikuje zda se v této oblasti objekt nachází a případně ho klasifikuje s určitou pravděpodobností. Následně je odhadnuta velikost a pozice ohraničujícího obdélníku. [27]

Vzhledem k tomu, že je pro každý snímek nutno vkládat 2000 prvků-oblastí do neuronové sítě, je tento algoritmus velice pomalý a naprosto nevhodný pro použití v reálném čase. Později byl algoritmus vylepšen, kdy dokázal pracovat při rychlostech až 4 FPS, toto řešení bylo však také velice rychle překonáno.[27]

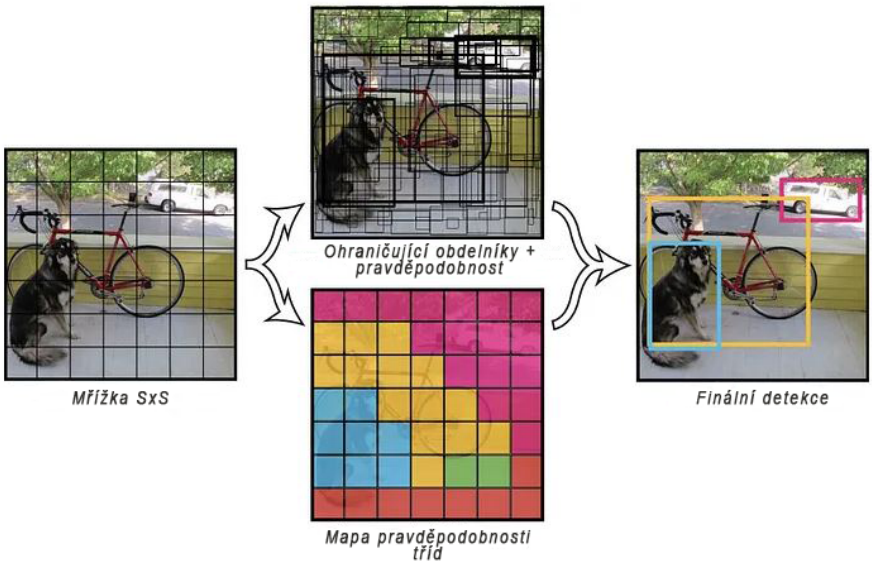
Faster R-CNN: U R-CNN byl hlavním problémem způsob určování oblastí pomocí fixního algoritmu. Model Faster R-CNN již v prvním kroku využívá jednu neuronovou síť sloužící k tvorbě oblastí, kde je každá z oblastí ohodnocena podle pravděpodobnosti výskytu objektu. Oblasti s pravděpodobností výskytu přesahující určitou prahovou hodnotu, jsou podobně jako u R-CNN vloženy do další

neuronové sítě,sloužící k extrakci charakteristických rysů, které jsou následně klasifikovány pomocí FCL - plně propojené vrstvy. Následně je odhadnuta velikost a pozice hraničního obdélníku. [28]

Díky těmto úpravám došlo k rapidnímu zlepšení rychlosti ale i přesnosti. Oproti předchozím iteracím je tento algoritmus použitelný pro aplikace v reálném čase.

Jednofázové detektory:

YOLO: Předchozí dvoufázové detektory využívají oblastí k lokalizaci objektů. Neuronová síť neprohledávala snímek jako celek ale pouze oblasti s největší pravděpodobností výskytu objektu. Detektor YOLO - You Only Look Once funguje velice odlišně. Využívá jedné jediné neuronové sítě, která odhaduje pozici hraničního obdélníku, ale i třídu do které ohraničený oraz spadá. Algoritmus nejdříve rozdělí vstupní snímek do mřížky o velikosti SxS. Z jednotlivých polí mřížky je vytvořeno m hraničních obdélníků různých rozměrů, kde každému z nich neuronová síť přiřadí pravděpodobnosti výskytu nějaké třídy a navrhne úpravu jeho velikosti. Hraniční obdélníky s pravděpodobností přesahující zvolenou prahovou hodnotu, jsou vybrány k lokalizování objektu uvnitř obdélníku.[29]



Obr. 2.8: YOLO[29]

Je jedním z nejvýznamnějších představitelů jednofázových detektorů a v průběhu posledních let prošel mnoha iteracemi, dosahujících vyšších rychlostí ale i přesností. Hlavní výhodou algoritmu YOLO oproti předešlým algoritmům je jeho mnohonásobně vyšší rychlost, dokáže totiž běžně dosahovat rychlostí 45FPS a při sníženém rozlišení vstupního obrazu dokáže pracovat až při 150 FPS. Jeho hlavní nevýhodou je, že není schopen detekovat objekty malých rozměrů. [30]

SSD: je jednofázový detektor velice podobný algoritmu YOLO, kde je hlavním rozdílem přístup k rozdělení vstupního snímku. SSD využívá sadu předdefinovaných

kotevních obdélníků s různými poměry stran pro různá měřítka. Díky tomuto přístupu dokáže efektivněji detekovat objekty různých velikostí. Každému obdélníku je přiřazena predikovaná třída a případné přesahy hraničního obdélníku. Výhodou SSD oproti YOLO je lepší detekce objektů v různých měřítkách a poměrech stran za cenu snížené rychlosti.[31]

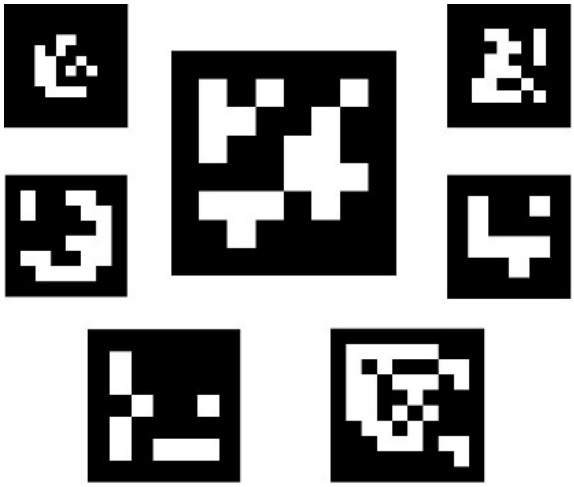
2.3.5 Detekce pomocí umělých obrazců(ArUco)

Vzhledem k dříve zmíněným metodám jde o odlišný způsob detekce objektů. Jde o uměle vytvořené obrazce (fidicual markers) vložené ve sledovaném prostoru, které jsme schopni snadno a efektivně detekovat, zjistit jejich polohu a ID. Metoda nám dovoluje vytvořit až několik stovek obrazců, nesoucí své jedinečné ID zakódované v matici uvnitř obrazce. My tak získáváme možnost přiřadit konkrétní ID k jednotlivým objektům a získat jejich polohu. Knihovna OpenCV obsahuje funkce ke generování a detekci takovýchto obrazců - ArUco markerů.[32]

ArUco marker je čtvercového tvaru a je tvořen širokou černou hranicí po obvodu. Uvnitř ohraničení se nachází binární matice, která určuje jeho identifikátor(id). Černá hranice ve zkoumaném obrazu usnadňuje rychlou detekci markeru. Velikost matice určuje celkovou velikost markeru, to znamená že čím větší vnitřní matice, tím větší množství možných markerů můžeme vytvořit. Například marker o velikosti 4x4 bude složen z 16 bitů. Nedoporučuje se však překračovat velikost 8x8(64bitů) z důvodu nečitelnosti/nutnosti kamery s vyšším rozlišením.[32]

Detekce probíhá následovně: algoritmus vyhledá ve zkoumaném obraze všechny tvary, které splňují podmínky čtverce (4 strany stejně dlouhé navzájem kolmé). Následuje rozhodnutí, zda se opravdu jedná o marker. Toho je dosaženo pomocí analýzy binárního kódu uvnitř čtverce. Pokud se podaří marker přiřadit do určitého slovníku, funkce vrátí polohu jednotlivých rohů markeru a jeho ID. V opačném případě jsou data zahozeny.[32]

Modul nabízí několik předdefinovaných slovníků - například DICT_4X4_50[33] obsahuje 50 markerů o velikosti vnitřní matice 4x4 Doporučuje se používat co nejmenší velikost slovníků. Čím více markerů - tím větší množství různých sobě podobných markerů a tím pádem větší pravděpodobnost chybné detekce - způsobené záměnou . Pokud bychom tedy potřebovali například pouze 10 markerů, nejlepší volbou je vytvořit vlastní slovník obsahující jen tyto markery. [32]



Obr. 2.9: Příklady ArUco markerů[13]

2.3.6 Shrnutí detektorů

V této kapitole bylo představeno několik různých přístupů detekování objektů, nejedná se o kompletní výčet, ale spíše o nejpopulárnější zástupce jednotlivých přístupů.

Prvním z nich byla detekce kontur, která může být dostatečným, velmi nenáročným řešením v případech, kdy víme že budeme potřebovat detekovat pouze pár tvarů na jednoduchém jednolitém pozadí. Toto řešení nemá jak klasifikovat o jaký objekt se jedná, nýbrž pouze detekovat jeho konturu a z ní určit polohu objektu. Dalším algoritmem byla substrakce pozadí, která je z podstaty věci nejvhodnějším kandidátem pro detekci pohybujících se předmětů.

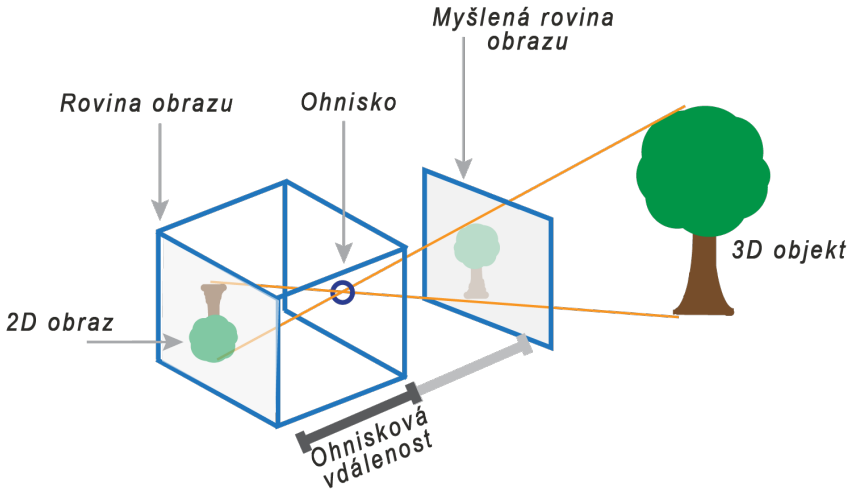
Následovaly klasické přístupy detekce. Ty prošly postupem času řadou vylepšení. Mezi nejrobustnější a často používané řešení patří algoritmus ORB, který dosahuje rychlé detekce a pracuje dobře s různými natočeními a měřítky objektů. Hlavní výhodou těchto algoritmů je, že nevyžadují vysoké nároky na výpočetní výkon, mohou však pokulhávat s rozmanitými obrazy obsahující více různých objektů, které se navzájem překrývají nebo mají chaotické pozadí. V takových případech mohou být charakteristické rysy těžko rozpoznatelné a porovnatelné. Detektory založené na strojovém učení mohou být v takových případech lepší volbou.

Předposledním přístupem byly algoritmy využívajících neuronových sítí. Dnes jde o algoritmy, které rok co rok procházejí řadou vylepšení a jde o nejrychleji se rozvíjející mezi zmíněnými algoritmy. Jsou schopné řešit komplexní scény a dosahovat vysokých přesností. Jsou však velmi náročné na výpočetní výkon a vyžadují velký objem vstupních dat. Pro použití v reálném čase jsou použitelné hlavně jednofázové detektory, jako jsou zmíněné algoritmy YOLO nebo SSD, ty jsou již dnes schopny dosahovat vysokých rychlostí.

Posledním, trochu odlišným přístupem byly ArUco markery, které lze jejich umístěním na objekt použít k nepřímé detekci. ID markeru pak může poukazovat například na třídu objektu. Může jít o jednoduché avšak efektivní řešení v mnoha aplikacích. Dnes se hojně využívá například v robotice.

2.4 Kalibrace kamery

Po celou dobu práce se bude pracovat s modelem dírkové kamery. Jde o model jednoduché kamery bez objektivu s jediným malým otvorem. Světelné paprsky procházejí tímto otvorem(ohniskem) a promítají převrácený obraz na protější straně kamery(rovina obrazu). Parametry kamery zahrnují vnitřní a vnější parametry a koeficienty zkreslení. [34]



Obr. 2.10: Model dírkové kamery[34]

Vnitřní parametry kamery zohledňují konstrukci kamery, tedy ohniskovou vzdálenost a polohu optického středu. Pomocí těchto parametrů můžeme vytvořit tzv. matici vnitřních parametrů. Matice vnitřních parametrů K je jedinečná pro konkrétní kameru, tím pádem nám ji stačí jednou zjistit/vypočítat a dále neomezeně používat ke snímaní různých obrazů.[34]

$$K = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix}$$

kde: f_x,f_y *- ohnisková vzdálenost;* c_x,c_y *- poloha optického středu*

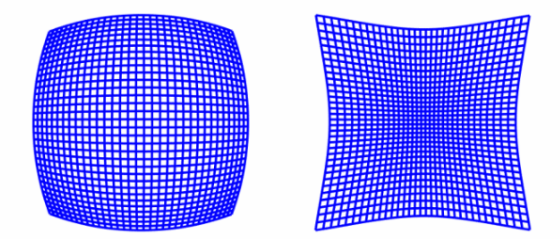
Vnější parametry popisují polohu a orientaci kamery vzhledem ke snímané scéně. Vynásobením vnitřních parametrů s vektorem pozice a orientace získáme matici kamery.

$$Matice\ kamery = K[Rt] \tag{2.1}$$

kde: R *- orientace kamery,* t *- poloha kamery*

Matice kamery však nebere v potaz zkreslení vzniklé v čočce/objektivu, protože model dírkové kamery čočku ani objektiv nemá. Dva hlavní typy takového zkreslení jsou radiální zkreslení a tangenciální zkreslení.[34]

Radiální zkreslení vzniká zakřivením čočky a způsobuje, že se z rovných linií stanou křivky. Toto zkreslení se dále dělí na dva další typy. Prvním je negativní dislokace, také nazývané jako soudkové zkreslení. Jde o stav, kdy jsou body ze své správné pozice posunuty směrem od středu ke krajům obrazu. Druhým typem je pozitivní dislokace, kde jsou body naopak posunuty směrem do středu obrazu. Je nazýváno také jako poduškové zkreslení.[35]



Obr. 2.11: Soudkové zkreslení(vlevo) Poduškové zkreslení(vpravo)[36]

Zkreslení jednoho bodu se dá popsat rovnicemi.

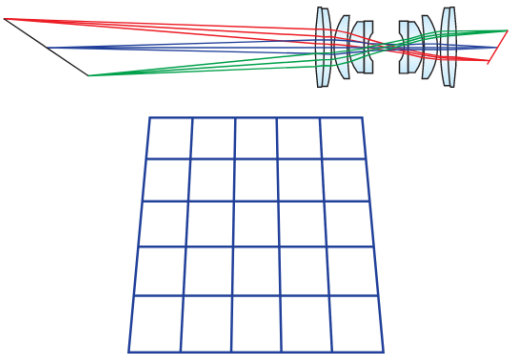
$$x_{zkres} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.2}$$

$$y_{zkres} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \tag{2.3}$$

$$r^2 = x^2 + y^2 \tag{2.4}$$

kde: x,y - nezkreslené souřadnice bodu; k_1,k_2,k_3 - koeficienty radiálního zkreslení

Tangenciální zkreslení vzniká nedokonalou rovnoběžností čočky se snímačem kamery. To způsobí, že se některé body zdají být blíže nebo dále než je tomu tak ve skutečnosti.



Obr. 2.12: Tangenciální zkreslení[37]

Pro tangenciální zkreslení.

$$x_{zkres} = x + [2p_1xy + p_2(r^2 + 2x^2) \tag{2.5}$$
$$y_{zkres} = x + [p_1(r^2 + 2y^2) + 2p_2xy) \tag{2.6}$$
$$r^2 = x^2 + y^2 \tag{2.7}$$

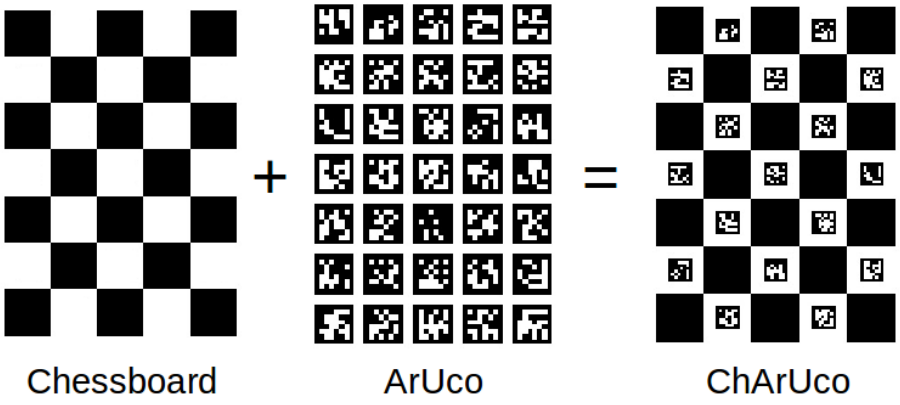
kde: x,y - nezkreslené souřadnice bodu; p_1,p_2 - koeficienty radiálního zkreslení

Takováto zkreslení jsou nežádoucí pro aplikace strojového vidění, kde je naší prioritou dosahovat co nejpřesnějších měření získaných z obrazu. Jako příklad je možné použít vyobrazení tangenciálního zkreslení, kde si je možné všimnout, že se vzdálenost mezi spodními rohy mřížky a horními rohy mřížky razantně liší, ačkoliv v realitě by tato vzdálenost byla stejná. [36]

Kalibrací kamery je docíleno částečné nebo úplné eliminace zkreslení. Ke kalibraci bychom potřebovali znát všechny výše zmíněné parametry, ty však neznáme. K odhadu parametrů a následné kalibrací je možné využít závislostí mezi body v trojrozměrném prostoru a jejich odpovídajícím obrazovým bodům v prostoru dvojrozměrném. K získání takovýchto bodů můžeme použít množinu snímků obsahujících kalibrační vzor. Jde o opakující se vzor se známými velikostmi a rozestupy. [34]

Nejčastěji zmiňovaným vzorem je šachovnice, která se skládá ze střídajících se bílých a černých čtverců stejné veliksoti. Rohy čtverců ležící uvnitř vzoru jsou využity jako kontrolní body. Tyto rohy mohou být v dvojrozměrném obrazu detekovány automaticky s využitím algoritmu pro detekci rohů(např Harris Corner Detector). Za předpokladu, že je například levý horní roh počátkem, jsme schopni získat také souřadnice těchto rohů ve trojrozměrném prostoru s využitím faktu neměnné velikosti čtverců šachovnice. Díky tomu jsme schopni vyřešit parametry kamery a obraz pomocí nich zkalibrovat. [13]

Knihovna OpenCV obsahuje vestavěné funkce ke kalibraci pomocí šachovnice, ale i pomocí jiných vzorů jako je ChaArUco board. ChArUco je kombinací klasické šachovnice a ArUco markerů umístěných uvnitř bílých čtverců. Oproti kalibraci pomocí klasické šachovnice tento vzor nevyžaduje kompletní visibilitu vzoru a umožňuje částečná zakrytí. Pro koncového uživatele to tak usnadní celý proces kalibrace.[38]



Obr. 2.13: Vzor ChArUco[13]

2.5 Produkce syntetických vláken metodou tažení

Pro pochopení cílů práce je vhodné stručně uvést do tématu produkce syntetických vláken v měřítku mikrometů až nanometrů.

Vlákna v měřítkách nanometrů spadají do oboru nanotechnologií, který se zabývá manipulací a studiem materiálů na nanometrové úrovni, tedy v rozsahu od jedné miliardtiny až do několika desítek nanometrů. Tento obor zahrnuje širokou škálu disciplín, jako jsou chemie, fyzika, biologie, elektronika a materiálové vědy.[39]. Díky rozvoji nanotechnologií bylo dosaženo mnoha inovací a objevů v různých oblastech, jako je zdravotnictví, elektronika, energetika a mnoho dalších. V oblasti zdravotnictví se například nanotechnologie využívá k vývoji nových léků, diagnostických nástrojů a materiálů pro regeneraci tkání. V oblasti elektroniky se nanotechnologie využívá k vývoji nových komponentů, jako jsou například nanočipy.[40]

Jednou z metod výroby syntetických vláken je metoda tzv. tažením, při které je možno produkovat vlákna v měřítku mikrometrů a v určitých případech až nanometrů. Jde o výrobu jednotlivých vláken prostřednictvím tažení vlákna z polymerního roztoku nebo z taveniny. Výhodou této metody je možnost výroby ojedinělých vláken a tím kontrolovat vzniklou strukturu jejich následným skládáním a překládáním. Další výhodou je minimální náročnost na zařízení: mikropipeta, podkladový materiál. Je důležité podotknout, že tato technologie umožňuje vytváření vláken o velikosti submikronů a za přesných technologických podmínek dokonce i o veliksoti nanometrů.[41]

Obr. 2.14: Robotický postup tvorby vlákna metodou tažení [42]

Automatizovaný proces tvorby syntetických vláken metodou drawing lze rozdělit na tyto kroky[42]:

1.pohyb mikropipety směrem k pracovní ploše

2.nanesení kapky polymerního roztoku na podkladový materiál

3.tažení vlákna z kapky polymeru určitou rychlostí

4.fixace vytaženého vlákna dotykem s pracovní plochou nebo s podkladovým materiálem

Popisovaná technologie umožňuje vytvoření sjednocených, přesně definovaných a umístěných vláken, což přináší řadu výhod. Jednou z těchto výhod je schopnost vytvářet komplikované struktury, které jsou navrženy speciálně pro konkrétní aplikace. Tato technologie se využívá zejména v oblastech senzoriky, fyziky a tkáňového inženýrství.[41]

3 Rešerše trhu se systémy strojového vidění

Pro získání přehledu o dostupných možnostech v oblasti strojového vidění v průmyslu je nezbytné prozkoumat současné produkty na trhu a posoudit, zda by některý z nich mohl představovat vhodné řešení pro problematiku produkce syntetických vláken metodou tažení.

Mezi přední poskytovatele systémů strojového vidění patří: Cognex, Keyence, Basler, SICK, Omron, National Instruments, atd.[43]

Přední výrobci mechatronických systémů jako je FANUC, KUKA, ABB a další, také nabízejí velké množství produktů strojového vidění, primárně v kategorii optického navádění robotů. Systémy těchto výrobců bývají často vytvořené k přímé implementaci do jejich mechatronických celků a poskytují univerzální řešení použitelné pro širokou škálu úkolů v kategorii optického navádění robotů. [44] [45] [46]

Strojové vidění v průmyslu by se všeobecně dalo rozdělit do 2 kategorií: optická kontrola/měření/detekce a optické navádění robotů.

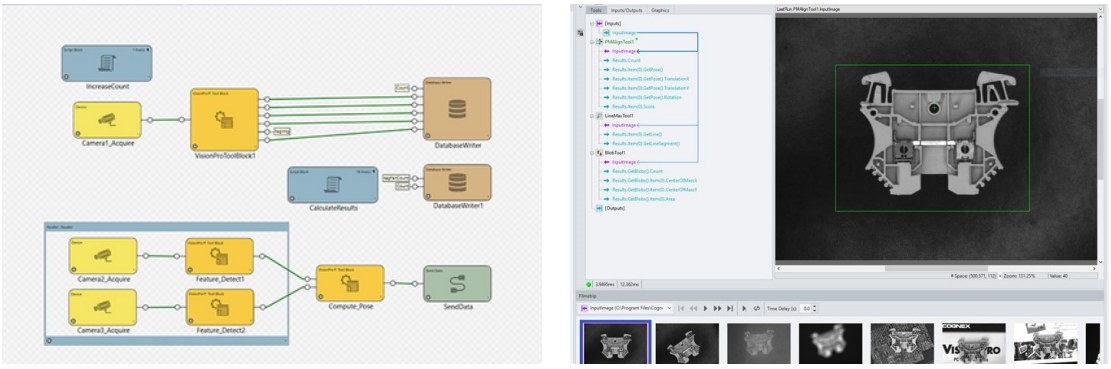
3.1 Systémy vše v jednom(AIO)

Na trhu se často můžeme setkat se softwary, které nabízejí částečnou až naprostou flexibilitu a je možné je tak použít jen v některých respektive ve všech kategoriích strojového vidění v průmyslu. Produkty od různých výrobců nabízejí navzájem podobné variace funkcionalit. Jako příklad použijme produkty společnosti Cognex, která by mohla sumarizovat produkty dostupné na trhu. Ta nabízí 3 typy softwaru.

Prvním je Cognex vision Library(CVL). Jedná se o maximálně flexibilní knihovnu napsanou v programovacím jazyce C++ která by měla pomoci výrobcům v tvorbě systémů strojového vidění pro to možná nejširší spektrum aplikací. CVL slouží k tvorbě dalšího softwaru, to znamená že se nejedná o jednoduchý plug&play systém s grafickým rozhraním. Vyžaduje pokročilou znalost programování, tím pádem i vysokou kvalifikaci personálu. Knihovna poskytuje nástroje vhodné k tvorbě systémů pro použití ve všech kategoriích i aplikacích. [47]

Druhým je COGNEX DESIGNER SOFTWARE. Dalo by se říct, že jde o nadstavbu CVL s grafickým rozhraním. Software by měl podpořit rychlejší a snadnější tvorbu systému za pomocí metody blokového programování. To usnadňuje jednodušší podporu a údržbu systému. Výkonné systémy je možno vytvořit pomocí metody přetahování komponent na stránku a jejich vzájemným propojováním (viz

obr 3.1a). K tvorbě systému není potřebná znalost programování, tím pádem není nutná tak vysoká kvalifikace personálu jako v předešlém případě.[48]



(a) Cognex Designer (b) Cognex VisionPro

Obr. 3.1: Grafická rozhraní[48][49]

Posledním z nich je Cognex Vision-Pro. Jendá se o univerzální software s grafickým rozhraním. Pro využití v širokém spektru aplikací. Software nabízí velké množství nástrojů a možností detekce. První možností je vytvoření pravidel, na jakých místech a jaké patterny/vzorce hledat. Podle nich pak software dále vyhodnocuje polohu předmětu nebo rozpoznává defekty, chybějící součástky, atd. Další možností je využití umělé inteligence a systém předem naučit co hledat pomocí metod hlubokého učení. To se hodí především při detekci složitějších vzorů, kde by systém založený na principu pravidel jednoduše nestíhal. Software lze používat bez jakékoliv znalosti programování a provede operátora krok po kroku potřebným nastavením. Tím pádem odpadá nutnost vysoce kvalifikovaného personálu pro ovládání systému.[49]

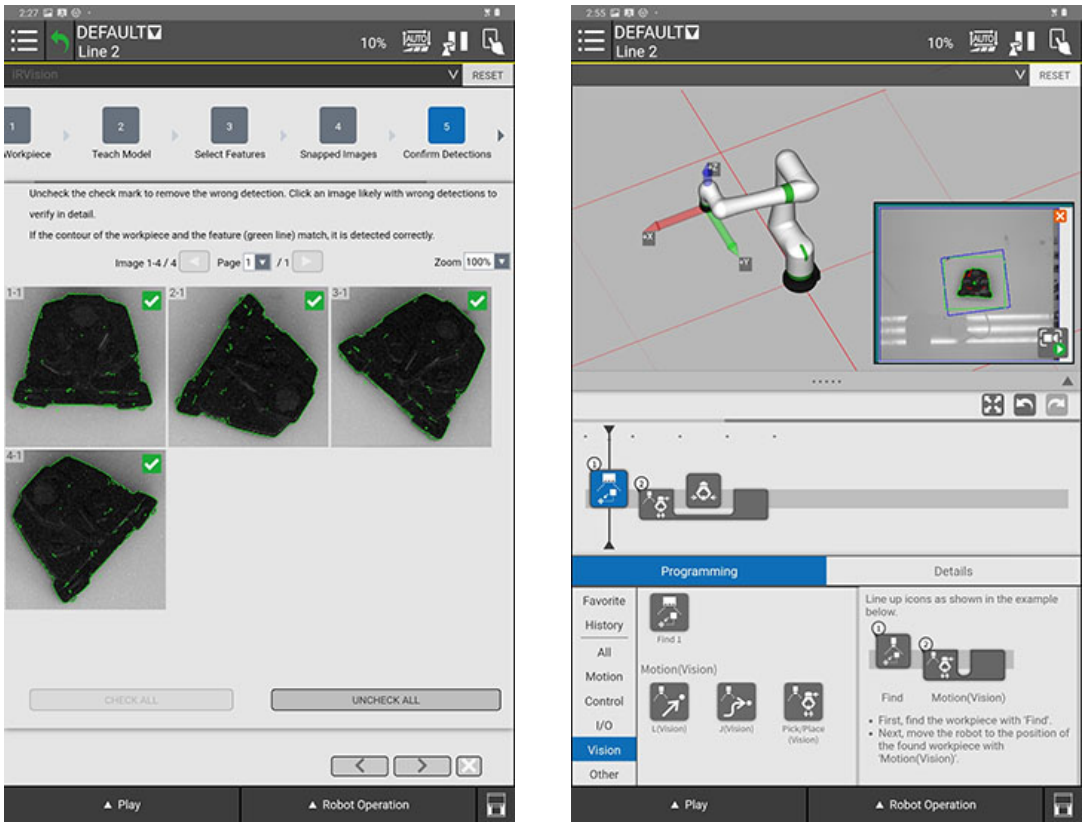
Vše uvedený software je jen příklad a na trhu se dá nalézt software, který je například určený k více konkrétnímu účelu.

3.2 Integrované systémy k navádění robotů

Vzhledem k tomu, že jsou si produkty nabízející řešení optického navádění navzájem podobné, popíšeme primárně systémy jen jednoho z výrobců k získání základní představy o produktech existujících na trhu v této kategorii. Z důvodu nejlépe dostupných informacích o produktech byly z výše uvedených společností vybrány systémy firmy FANUC.

Firma FANUC nabízí balík strojového vidění nazývaný iRVision obsahující několik typů systémů, kde každý slouží ke svému konkrétnějšímu účelu. Mělo by se jednat o "připoj a hraj"(Plug&Play) systém bez nutnosti pořizování jakéhokoliv dalšího hardwaru, jelikož se všechno nastavovaní odehrává v ovladači robotu - pendantu. Nastavení má být jednoduché a intuitivní, kde Pendant operátora provede jednotlivými kroky nastavení. Všechny typy sdílí stejné grafické rozhraní. To všechno by mělo vést k usnadnění procesu implementace bez potřeby komplikovaného pro-

gramovaní a potřeby vysoce kvalifikovaného personálu. Aktuálně FANUC nabízí 7 typů systémů(viz.[44]). Všechny typy systémů jsou podporovány simulačním softwarem ROBOGUIDE, který umožňuje simulaci požadovaných procesů, což nabízí posouzení proveditelnosti a efektivnosti celého navrženého procesu, ještě před koupí systému.[44]



Obr. 3.2: Grafické rozhraní iRVIsion v pendantu robotu[44]

Popsaný balík je jen jedním z příkladů integrovaného řešení optického navádění robotů, které jsou na trhu dostupné. Je možné si vybrat buď systém integrovaný: FANUC-FANUC, nebo k strojovému vidění používat systémy 3 stran: KUKA-Cognex. Pokud již firma disponuje roboty od jednoho z těchto výrobců, bývá často výhodné používat i jejich systémy strojového vidění z důvodu jednoduché implementace a podpory. Každý má však v určitých situacích své výhody i nevýhody. Na trhu existují i menší výrobci, kteří ke svým strojům systém strojového vidění vůbec nevytvářejí. V takovém případě jsou jasnou volbou systémy 3 stran jako je například Cognex, nebo Keyence. Jako příklad se dají uvést produkty společnosti Cognex, která nabízí jak systémy pro roboty konkretních výrobců:ABB, Yaskawa, KUKA, Mitsubishi, tak i univerzální AIO systém(viz kapitola 2.9.1). A nebo systém vytvořený čistě k navádění robotů schopný fungovat s robotem jakéhokoliv výrobce, který však oproti přímo implementovaným nebo universálním postrádá hned několik funkcí[50].

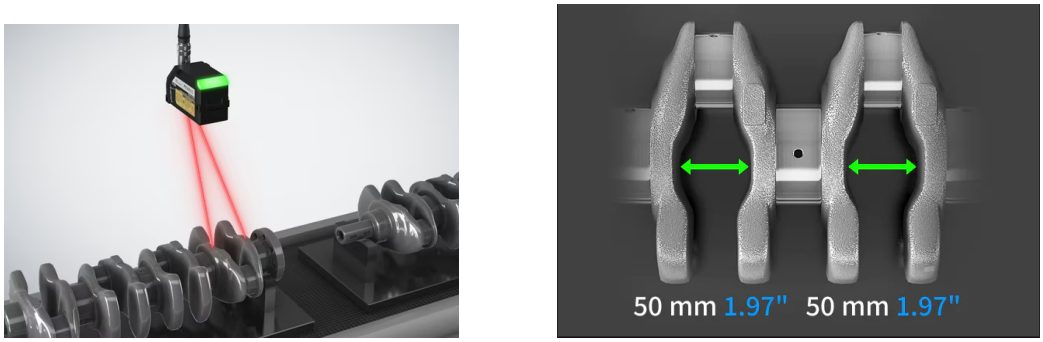
3.3 Specifická řešení

Jednoduché univerzální řešení může vyvodit problém u velmi specifických aplikací, kde se může stát z jednoduchosti spíše přítěž. V takových případech jsou možné dvě řešení. Využít některého z AIO softwaru(viz kapitola 3.1). a systém si vytvořit interně nebo použít systém "na míru", který se na danou problematiku přímo specializuje.

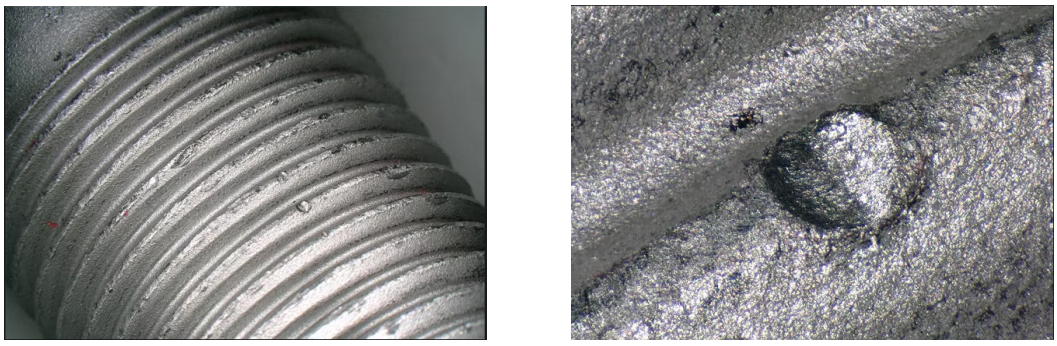
V kategorii optického navádění robotů může být příkladem hned několik takových specifických aplikací. V oblasti elektroniky: optické navádění při umisťování elektronických součástek na tištěné spoje, zarovnávání před laserovým vrtáním, zarovnávání při laminování OLED displayů. Nebo v oblasti automotive: Laminování a přemisťování baterií pro elektromobily a další.[51]

V kategorii optické inspekce bychom našli příkladů mnohem více. Jako příklad zde použijeme produkty společnosti Keyence, která nabízí systémy strojového vidění pro optickou kontrolu v oblasti automotoive pro měření odchylek rozměrů klikových hřídelů nebo ojnic, kontrolu správného zajištění pojišťovacích kroužků hřídelů, kontrolu gumových těsnění, detekci chybějících šroubů uchycení kol,atd. Nabízí také optické přeměřovaní rozměrů ozubených kol, měření rozměrů a kontrolu geometrické přesnosti lisovaných plechů nebo optickou kontrolu defektů závitů. [51]

Uvedení poskytovatelé (viz kapitola 3.0) často přímo nabízejí možnost vytvoření poptávky po specifickém řešení pro zákazníkovu aplikaci. Výše uvedené aplikace slouží jen jako příklad toho, co je na trhu možné získat.



Obr. 3.3: Měření odchylek rozměrů klikových hřídelů[51]



Obr. 3.4: Kontrola defektů závitů [51]

3.4 Shrnutí

Počet aplikací, které strojového vidění využívají nebo by mohli využít, je v podstatě neomezený a pravděpodobně bude dále růst. Každá z aplikací však může vyžadovat velmi odlišný přístup než ta předchozí.

Volba řešení bude záviset hlavně na aplikaci, ke které má strojové vidění sloužit a na její složitosti. Od toho se bude odvíjet který konkrétní systém a od kterého poskytovatele využít. U "vyřešených" specifických aplikací může být vhodné takového řešení využít. Vždy však také bude záležet na kvalifikaci personálu, finančních možnostech případně i na preferencích.

K řešení problematiky produkce syntetických vláken metodou tažení by bylo možné zvolit jeden ze systémů "vše v jednom", např. knihovnu CVL. V případě, že bychom k tažení využívali například robotické rameno bychom mohli využít integrovaného řešení. Specifické řešení, které by se zabývalo konkrétně touto problematikou nebylo nalezeno.

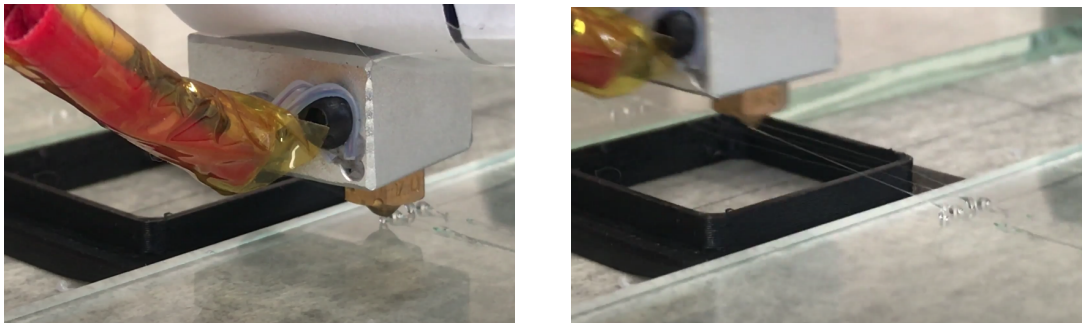
4 Návrh řešení

Tato kapitola se bude věnovat návrhu řešení dané problematiky. Nejprve bude popsán samotný problém, poté budou stanoveny kroky vedoucí k jeho vyřešení a nakonec budou popsány nástroje, které budou k realizaci využity.

4.1 Obecný popis fungování systému

Jak již bylo zmíněno v úvodu, cílem práce je vytvořit systém strojového vidění pro usnadnění výroby syntetických vláken metodou tažení. Systém strojového vidění je z velké části algoritmus, což je sekvence instrukcí, které popisují, jak provést konkrétní úlohu nebo operaci.

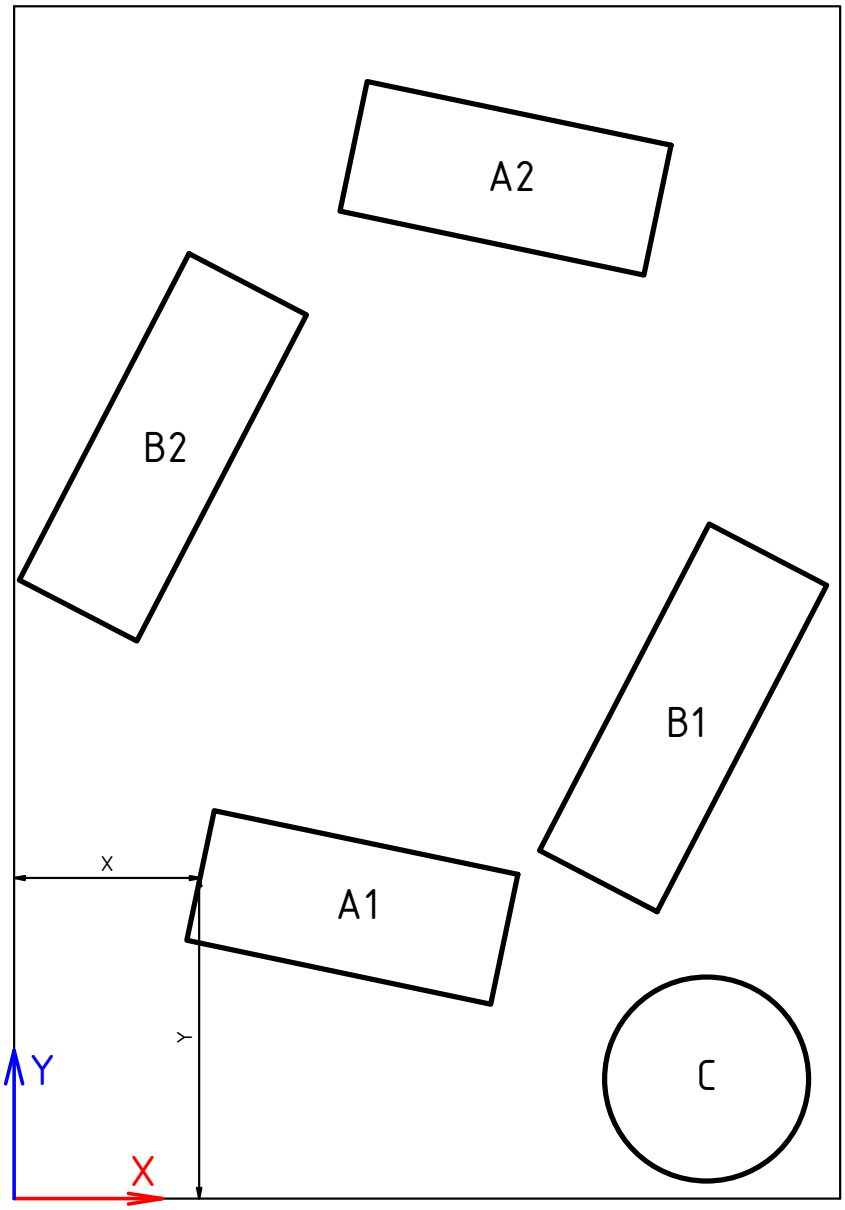
V prvním kroku bude obecně popsána podstata problému, co by měl vlastně výsledný systém dělat. Velmi jednoduše řečeno při procesu produkce nanovláken metodou nanovláken dochází k tažení vlákna od jedné fixační plochy ke druhé. Tisková hlava nebo pipeta nejdříve položí kapku na fixační plochu(viz obr 4.2a) a poté z této kapky táhne vlákno směrem ke druhé fixační ploše(viz obr 4.2b).Podrobněji je tato metoda popsána v kapitole 2.5. Následující obrázek



(a) položení kapky (b) tažení vlákna

Obr. 4.1: Tažení vlákna

znázorňuje pracovní prostor, ve kterém probíhá operace tažení vláken.



Obr. 4.2: Znázornění pracovní plochy při tažení nanovláken

Označené dvojce ploch A1,A2 a B1,B2 jsou dvojce zmíněných fixačních ploch, které slouží k zachycení počátku nebo konce taženého vlákna. Číslice 1 indikuje startovací plochu. Úkolem systému strojového vidění je tyto plochy nalézt, identifikovat a zjistit jejich souřadnice vztažené k souřadnicovému systému produkčního stroje. V případě tažení pomocí pipety navíc detekuje zásobník s roztokem, na obrázku označen písmenem C. Je nutno podotknout že tažení v tomto případě probíhá pouze v dvojrozměrném prostoru, veškeré fungování algoritmu bude tedy pracovat pouze v rovině XY.

Vstupem pro algoritmus je obraz získaný z kamery namontované na produkčním

stroji. Z tohoto obrazu pak algoritmus extrahuje potřebná data a zpracuje je k dosažení požadovaného výstupu.

Výstupem algoritmu jsou data o dvojicích ploch a jejich souřadnicích a o poloze zásobníku s roztokem. Tyto data pak budou vstupem pro řídící algoritmus stroje prováděcího samotnou operaci tažení vláken.

4.2 Postup řešení

Jako u mnoha větších systémů je klíčovým krokem rozdělení celkového problému na menší a snadněji zvládnutelné části. To umožňuje lépe porozumět jednotlivým částem problému a věnovat se jim podrobněji. Pro řešení problematiky byl navržen následující postup řešení shrnutý do několika kroků:

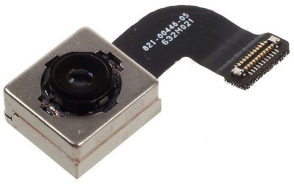
1. Popis strukturu souborů a funkcí
2. Kalibrace kamery
3. Stanovení a detekce pracovního prostoru
4. Detekce fixačních ploch a zásobníku s roztokem
5. Zpracování získaných dat

4.3 Použitá zařízení, knihovny a nástroje

Před samotnou realizací bude představeno zvolený software, zařízení a knihovny.

4.3.1 Kamera

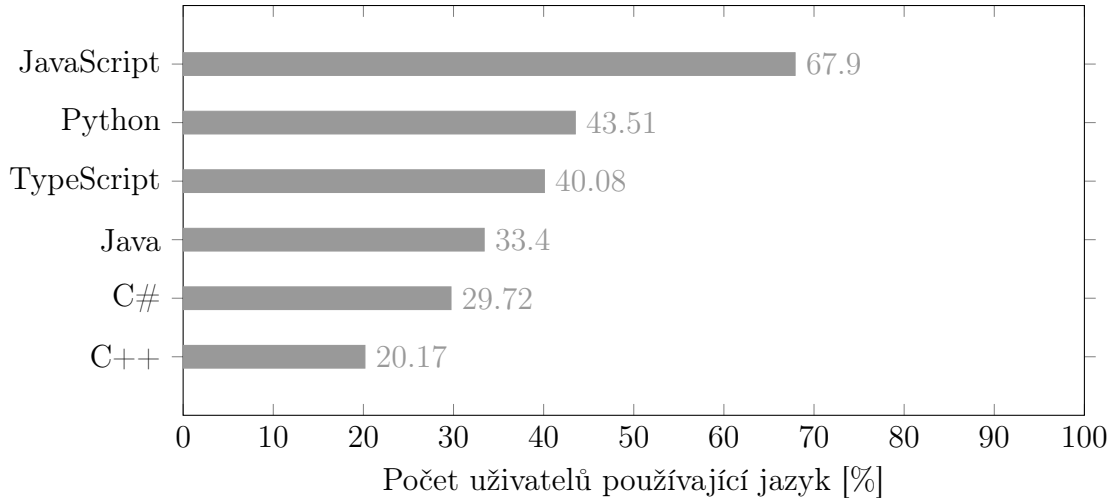
Při tvorbě programu byla použita kamera mobilního telefonu iPhone 7, který byl instalován na tuhém stativu. Jde o kameru s rozlišením 12MPx s ohniskovou vzdáleností 1.8 mm. Taková kamera by v žádném případě nebyla vhodná do ostrého provozu, k účelům tvorby programu však byla dostačující. Při řešení této problematiky je však možné využít kameru libovolného typu, přičemž je žádoucí, aby měla co nejmenší zkreslení.



Obr. 4.3: Použitá kamera

4.3.2 Programovací jazyk Python

Python je vysokoúrovňový objektově orientovaný programovací jazyk, který se staví mezi ty nejpopulárnější programovací jazyky mezi vývojáři po celém světě. Python má čistou a jednoduchou syntaxi, která zjednodušuje psaní a čtení kódu, je velmi intuitivní a snadno se učí. Obsahuje mnoho vestavěných knihoven a modulů, které usnadňují a urychlují vývoj aplikací. V neposlední řadě je velkou výhodou, že má obrovskou komunitu vývojářů, kteří sdílejí své znalosti, zkušenosti a nástroje, což usnadňuje řešení problémů a zlepšuje kvalitu kódu.



Obr. 4.4: Nejpoužívanější programovací jazyky dle StackOverflow survey.[52]

4.3.3 Vývojové prostředí PyCharm Community Edition

PyCharm Community Edition je vývojové prostředí pro programování v jazyce Python. Jde o nástroj, který v několika různých ohledech usnadňuje práci při tvorbě kódu. Podobných softwarů je dostupných hned několik, např: Vsiual Studio Code, Anaconda, Eclipse a další. Tento software nijak nemění fungování samostatného kódu a vetšinou závisí čistě na preferencích každého individuálního vývojáře. PyCharm vyvíjí česká společnost JetBrains s.r.o.[53]

4.3.4 Knihovna OpenCV

OpenCV(Open Computer Vision) je otevřená, volně stažitelná knihovna, sloužící k operacím na poli počítačového vidění. Umožňuje manipulaci s obrazem, díky které lze vykonávat širokou škálu dalších úkolů. Knihovna obsahuje více jak 2500 algoritmů, které mohou být použity například k detekci a rozpoznávání obličejů, rozpoznávání lidských činů na videu, sledování pohybů kamery, sledování pohybujících se objektů a mnoha dalšího. Knihovna je napsána v jazyce C++ a má také plnohodnotné rozhraní pro jazyky Python, Java a MATLAB. Podporuje Windows, Linux, Adnroid a Mac OS.[13]

Open-source licence je u OpenCV navržena tak, že je kompletně celá knihovna volně stažitelná a je ji možno použít k soukromým, ale i komerčním účelům kompletně zdarma.[13] Velkou výhodou knihovny je její rozsáhlá komunita, díky níž je řešení problémů mnohem snadnější.V této práci jde o stěžejní knihovnu, jelikož bude velká část algoritmu využívat právě jejich funkcí, na které bude v průběhu odkazováno.

4.3.5 Git a GitHub

Git je distribuovaný verzovací systém pro správu kódu a spolupráci na softwarových projektech. Je navržen tak, aby umožňoval rychlé a efektivní sledování změn v kódu a jejich koordinaci mezi vývojáři.

GitHub je webová platforma pro hosting projektů a správu verzí kódu pomocí Gitu. Umožňuje vývojářům sdílet své projekty a pracovat na nich s ostatními vývojáři pomocí funkcí jako jsou například pull requesty, code review a issue tracking.

4.3.6 Formát pro ukládání dat JSON

JSON je formát pro ukládání a výměnu dat, který je snadno čitelný a srozumitelný jak pro lidi, tak pro stroje. Je založen na textovém formátu, který je přehledný a strukturovaný, což usnadňuje práci s daty. JSON je nezávislý na platformě a může být používán v různých programovacích jazycích. Umožňuje reprezentovat strukturovaná data, jako jsou objekty a pole, což usnadňuje organizaci a hierarchii dat. Tím umožňuje efektivní ukládání a přenos dat mezi různými systémy a aplikacemi. Jednoduchým příkladem JSON souboru může být například tento:

```
{
  "jmeno": "Jan",
  "vek": 25,
  "mesto": "Liberec",
  "konicky": ["plavani", "cyklistika", "lyzovani"]
}
```

Zdrojový kód 4.1: Příklad JSON souboru

4.3.7 Další knihovny

Knihovna	**Popis**
NumPy	poskytuje efektivní podporu pro výpočty s maticemi, vektorizaci a jiné funkce pro vědecké výpočty.
SciPy	obsahuje nástroje pro matematické, vědecké a technické výpočty, včetně integrace, optimalizace, signálního zpracování a statistiky.

Tabulka 4.1: Další použité knihovny

5 Řešení

Tato kapitola se bude zabývat samotnou tvorbou systému strojového vidění, který se bude skládat ze dvou dílčích algoritmů. První vedlejší algoritmus bude sloužit ke kalibraci kamery. Druhý hlavní algoritmus bude sloužit k samotné detekci přípravků uvnitř pracovního prostoru a bude rozdělen na 3 části: detekci pracovního prostoru, detekci přípravků a zpracování získaných dat. Kapitola je tedy rozdělena do následujících podkapitol:

1. Popis strukturu souborů a funkcí
2. Kalibrace kamery
3. Stanovení a detekce pracovního prostoru
4. Detekce fixačních ploch a zásobníku s roztokem
5. Zpracování získaných dat

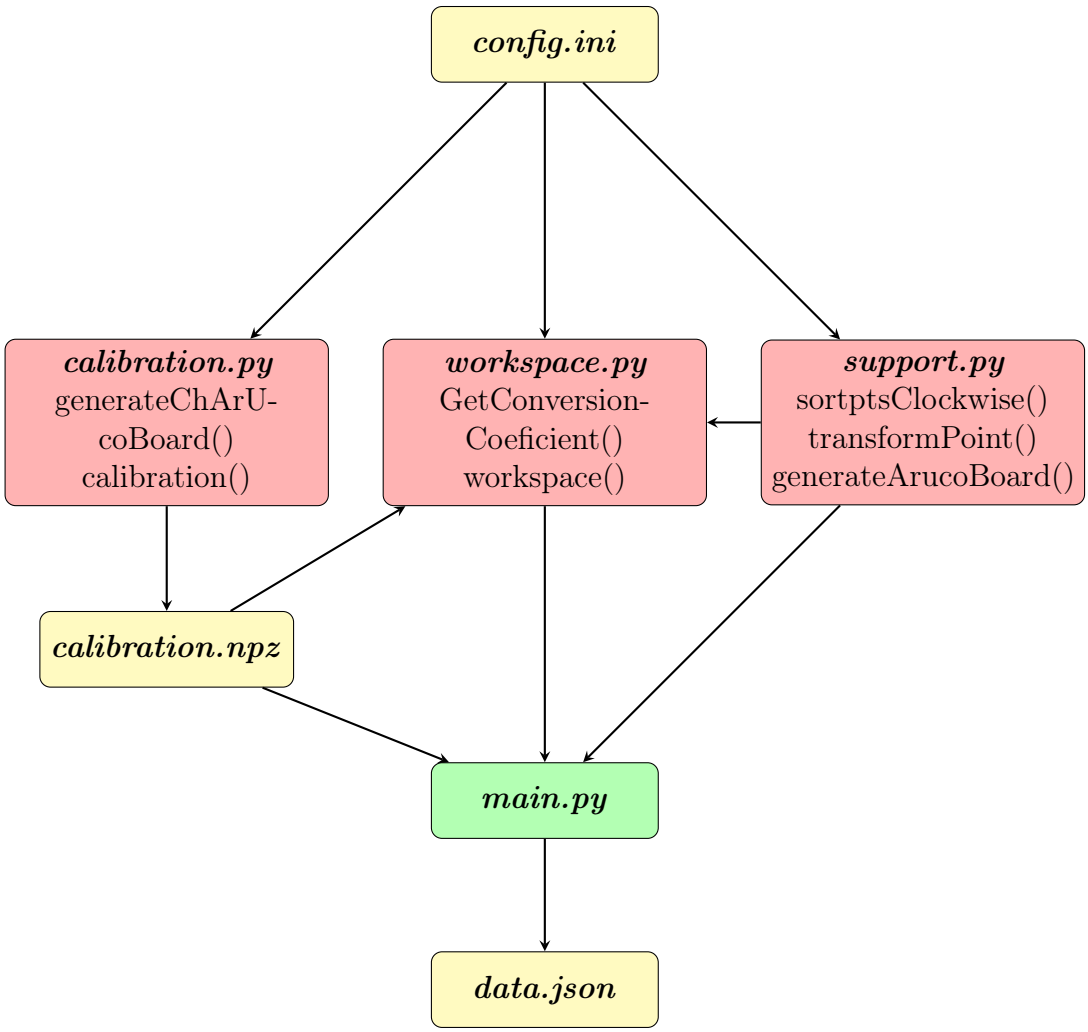
Předem je nutno dodat, že vyobrazené části kódu slouží pro ukázku hlavních funkčních částí algoritmu a neobsahují úplný výčet zdrojového kódu. Z toho důvodu se může stát, že na sebe jednotlivé části ne vždy přesně navazují. Každá komplexnější funkce popsána pomocí logického diagramu pro lepší pochopení. V příloze této práce je uveden úplný zdrojový kód(viz Příloha A).

Dalším důležitým faktorem, který si je třeba uvědomit při čtení nadcházejícího kódu, je, že popsaný proces se vždy aplikuje na jednotlivý snímek získaný z kamery v daném čase. To znamená, že algoritmus bude zpracovávat například 30 snímků za sekundu, pokud je rychlost snímání (FPS) nastavena na tuto hodnotu. V každé vteřině se tedy cyklus pro zpracování opakuje 30krát.

5.1 Struktura souborů a funkcí

Pro přehlednější strukturu kódu budou jednotlivé funkce umístěny do svých příslušejících souborů. Pro jednoduší přehlednost a orientaci mezi soubory byl vytvořen následující diagram. Hlavním souborem je soubor *main.py* označený zeleně,ze kterého se bude spouštět samotný algoritmus strojového vidění k detekci přípravků. V červených polích jsou soubory ve formátu .py. Které obsahují jednotlivé funkce, které budou vytvořeny v následujících několika kapitolách.

Ve žlutých polích jsou pak datové soubory. Soubor **config.ini** obsahuje volitelné parametry, které může uživatel změnit podle vlastních potřeb. Jednotlivé parametry jsou popsány přímo v tomto souboru. Soubor **calibration.npz** obsahuje získaná data z procesu kalibrace, nutné pro kompenzaci zkreslení. Soubor **data.json** obsahuje hlavní výstup celého algoritmu viz(kapitola 5.5).



Obr. 5.1: Diagram souborů a funkcí

5.2 Kalibrace kamery

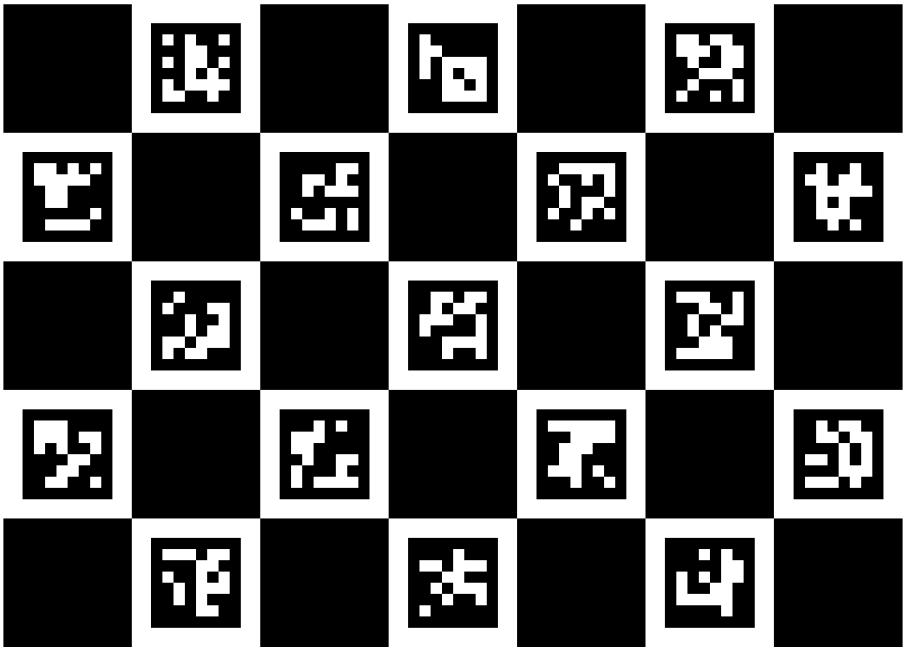
Jak již bylo zmíněno v kapitole 2.4, než se pustím do tvorby hlavního algoritmu strojového vidění, je nutné se zbavit zkreslení obrazu způsobeného kamerou. K tomuto účelu vytvořím samostatný soubor nazvaný *calibration.py*, který bude obsahovat samostatný vedlejší algoritmus skládající se ze dvou na sebe navazujících funkcí. Ke kalibraci budu používat modul OpenCv nazvaný ArUco.(viz kapitola 2.3.6)

Prvním logickým krokem je tvorba kalibračního vzoru/desky zvaného ChArUco(viz kapitola 2.4). Ve zkratce jde o šachovnici, jež má ve svých bílých polích umístěny ArUco markery. K tomuto účelu vytvořím funkci nazvanou ***generateChArUcoBoard()***.

```
6  def generateChArUcoBoard(squaresX, squaresY, filename=None):
7      #definuje velikost A4 pri 300 dpi
8      paperWidth, paperHeight = int(8.27 * 300), int(11.69 * 300)
9      #velikost ctvercu a markeru
10     squareLength = int((paperWidth-100) / squaresX)
11     markerLength = int(squareLength * 0.7)
12     #Vyber slovniku
13     dict = cv2.aruco.DICT_6X6_250
14     dictionary = cv2.aruco.getPredefinedDictionary(dict)
15     #Vytvori desku
16     board =cv2.aruco.CharucoBoard((squaresX, squaresY),
       squareLength, markerLength, dictionary)
17     if filename is not None:
18         #Vykresli desku
19         img = board.generateImage((paperWidth, paperHeight))
20         #Ulozi desku
21         cv2.imwrite(filename, img)
22     return board
```

Zdrojový kód 5.1: Tvorba kalibračního vzoru

Vstupní parametry jsou v tomto pořadí: počet čtverců ve směru X, a počet čtverců ve směru Y, posledním parametrem je volitelný parametr filename, v případě jeho vyplnění bude navíc deska uložena jako obrázek ve formátu filename.png. K vygenerování kalibračního vzoru použiji funkci ***cv2.aruco.CharucoBoard()*** [54]. Výstupem funkce je proměnná board, která mi poslouží v nadcházejících krocích.



Obr. 5.2: Vygenerovaný kalibrační vzor ChArUco

Jakmile mám desku vygenerovanou, mohu se pustit do samotné kalibrace, tedy do výpočtu matice kamery, koeficientů zkreslení a kalibrační matice. K tomu vytvořím funkci nazvanou ***calibration()***. Funkce má dva vstupní parametry: prvním je proměnná board získaná z předchozí funkce a druhým je proměnná záznamu kamery. Nahrajeme slovník dictionary a definujeme listy charucoCorner a CharucoIds.

```
23 def charlibration(board,cap):
24     dict = cv2.aruco.getPredefinedDictionary(dict)
25     dict = cv2.aruco.DICT_6X6_250
26     charucoCorners = []
27     charucoIds = []
```

Zdrojový kód 5.2: Příprava slovníku ArUco a seznamů

Ze záznamu videa z kamery získávám jednotlivé snímky a na každém z nich detekuji ArUco markery pomocí funkce ***cv2.detectMarkers()***[55]. Výstupem funkce jsou data o rozích jednotlivých markerů a jejich identifikačních číslech. Tato data pak použiji ve funkci ***cv2.interpolateCornersCharuco()***[56], která už detekuje jednotlivé rohy kalibrační šachovnice a jim příslušící identifikační čísla. V případě stisknutí klávesy '+' jsou rohy šachovnice uloženy do listu **charucoCorners** a identifikační čísla do listu **charucoIds**.

```
29     #Zatimco je zaznam videa otevren
30     while cap.isOpened():
31         ret, frame = cap.read() #cteni zaznamu z kamery
32         #Detekce markeru
33         corners, ids, _ = cv2.aruco.detectMarkers(frame, dict)
34         if len(corners) > 0: #Pokud je nalezen alespon jeden marker
35             ret, charuco_corners, charuco_ids = cv2.aruco.
   interpolateCornersCharuco(corners, ids, frame, board)
```

```
36              # Pokud je deska nalezena(vice jak 15 rohu)
37              if ret > 15:
38                  #Pokud je stisknuta klavesa +
39                  if cv2.waitKey(1) == ord('+'):
40                      charucoCorners.append(charuco_corners)
41                      charucoIds.append(charuco_ids)
42                  #Vykreslii detekovane rohy desky
43                  cv2.aruco.drawDetectedCornersCharuco(frame,
    charuco_corners,charuco_ids)
```

Zdrojový kód 5.3: Detekce kalibrační desky

Pomocí funkce ***cv2.imshow()***[57] zobrazím výsledný obraz z kamery s vykreslenými detekovanými rohy. V případě zachycení dostatečného počtu snímků může obsluha snímání obrazu ukončit pomocí klávesy 'q' a algoritmus tak přejde k samotnému výpočtu matice kamery a koeficientů zkreslení.

```
61          #Zobrazeni vysledneho obrazku
62          cv2.imshow('Calibration', frame)
63          #Pokud je stisknuta klavesa q ukonci se program
64          if cv2.waitKey(1) == ord('q'):
65              break
66      cap.release()
67      cv2.destroyAllWindows()
```

Zdrojový kód 5.4: Zobrazení obrazu a konec smyčky

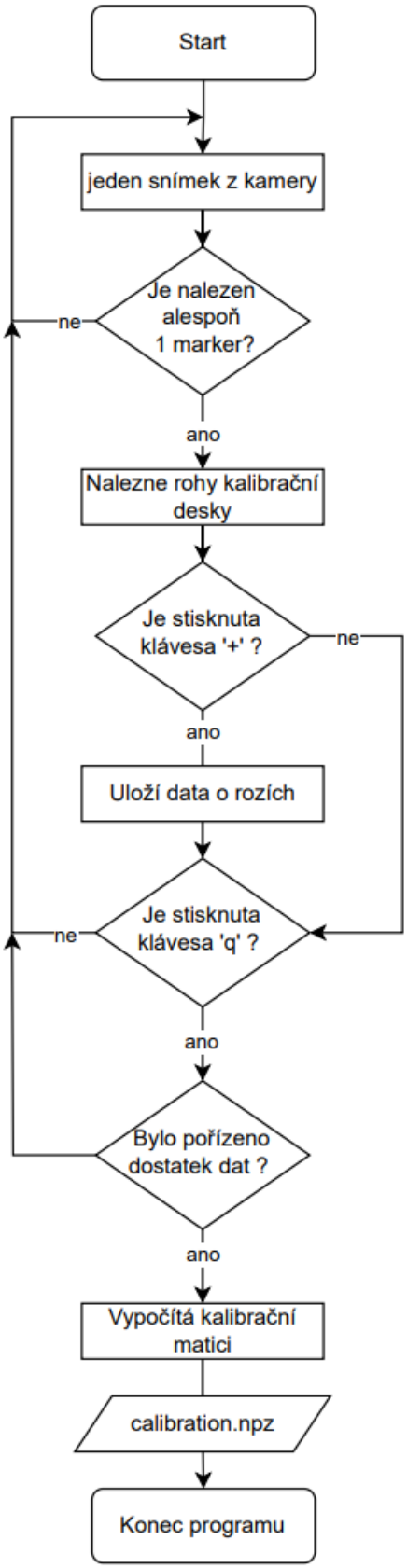
K výpočtu matice kamery a koeficientů zkreslení využiji funkci z knihovny OpenCv ***cv2.aruco.calibrateCameraCharuco()***[58], pro kterou jsou vstupem předtím získaná data z listu **charucoCorners** a **CharucoIds**. Jakmile získám matici kamery a koeficienty zkreslení, vypočítám z nich novou matici kamery, neboli kalibrační matici. Tyto 3 proměnné uložím do souboru **calibration.npz** pomocí funkce ***savez()***[59] z knihovny NumPy. Díky těmto proměnným budu později kalibrovat všechny vstupní snímky pro hlavní algoritmus.

```
71 #Ziskani matice kamery
72     h, w = frame.shape[:2]
73     ret, mtx, dist,R,T = cv2.aruco.calibrateCameraCharuco(
    charucoCorners, charucoIds, board,(w,h),None,None)
74     newmtx, roi = cv2.getOptimalNewCameraMatrix(mtx, dist, (w,h),
    1, (w,h))
75     # Ulozeni matice kamery, distorze a novy matice kamery
76     np.savez('calibration.npz', mtx=mtx, dist=dist, newmtx=newmtx)
```

Zdrojový kód 5.5: Výpočet matice kamery;koeficientů zkreslení a kalibrační matice

Proměnná	**Popis**
mtx	matice kamery
dist	koeficienty zkreslení
newmtx	nová matice kamery

Tabulka 5.1: Přehled výstupných proměnných funkce calibration()

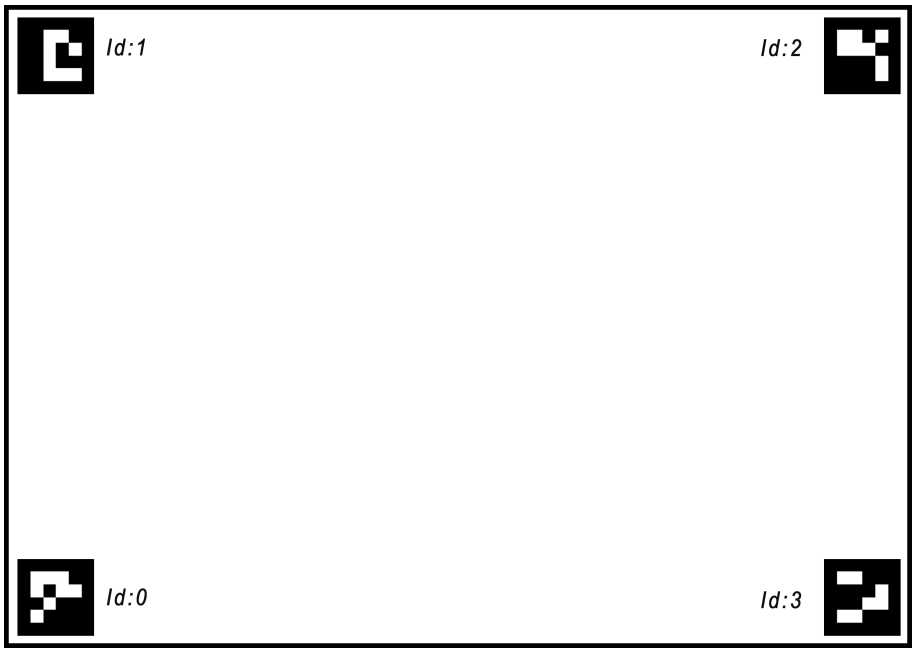


Obr. 5.3: Vývojový diagram funkce calibration()

5.3 Stanovení a detekce pracovního prostoru

Abych byl schopen detekovat požadované objekty, potřebuji si nejdříve definovat, co je to pracovní prostor, jinými slovy ho potřebuji nějakým způsobem ohraničit. Pracovním prostorem bude v mém případě vždy obdélník o libovolném poměru a velikosti stran. K jeho ohraničení opět využiji ArUco markery(viz kapitola 2.3.6), kdy do každého rohu pracovního prostoru(obdélníku) umístím jeden ArUco marker se svým unikátním ID. Požadavkem je, aby byly markery umístěny postupně podle svého ID (identifikačního čísla) od nejmenšího po největší ve směru hodinových ručiček.

Důvodem použití ArUco markerů k ohraničení pracovního prostoru je jejich snadná detekovatelnost a také existence celého modulu v knihovně OpenCv, který se věnuje ArUco markerům a nabízí nespočet funkcí, jak s markery naložit. Zároveň bude mít obsluha možnost libovolně měnit velikost pracovního prostoru v závilosti na jejich umístění. ArUco markery budou v tomto případě generovány ze slovníku `cv2.aruco.DICT_4X4_50`, slovník si lze zvolit libovolně, prostřednictvím konfiguračního souboru **config.ini**.



Obr. 5.4: Znázornění ohraničeného pracovního prostoru

Takto definovaný prostor budu prostřednictvím ArUco obrazců detekovat na obrazu z kamery. Obraz následně oříznu, čímž se zbavím okolního prostoru, který pro mě v tomto případě není nijak zajímavý. S oříznutým obrazem pak budu dále pracovat.

K tomuto účelu vytvořím funkci ***workspace()***, která bude uložena v souboru **workspace.py**. Funkce bude zahrnovat detekci ohraničujících markerů a na základě toho pak transformaci obrazu. Transformace obrazu zahrnuje kompenzaci natočení kamery kolem všech 3 os a následné oříznutí regionu zájmu - pracovního prostoru.

Vstupem pro funkci je pouze jedna proměnná **frame**, obsahující jeden samostatný snímek obrazu. Výstupem funkce boudou 3 proměnné: 2 proměnné obsahují obraz a 3. proměnná obsahuje převodní koeficient mezi pixely a milimetry(viz níže).

V prvním kroku nahraji do souboru workspace.py získaná data z procesu kalibrace.

```
data = np.load('calibration.npz')
mtx, newmtx, dist = [data[i] for i in ('mtx', 'newmtx', 'dist')]
```

Zdrojový kód 5.6: Import kalibračních dat

Dále už definuji funkci ***workspace()*** a začnu detekovat ArUco markery na vstupním snímku pomocí funkce ***cv2.detectMarkers()***[55]. Výstupem funkce jsou souřadnice rohů jednotlivých markerů a jejich identifikační čísla.

```
def workspace(frame):
    #Nahraje directory s pouzivanymi markery
    dict = cv2.aruco.DICT_4X4_50
    arucoDict = cv2.aruco.getPredefinedDictionary(dict)
    #Detekce markeru
    corners, ids,_ = cv2.aruco.detectMarkers(frame, arucoDict)
```

Zdrojový kód 5.7: Detekce ArUco markerů

Pokud je nalezen alespoň jeden marker, mohu spočítat souřadnice každého nalezeného markeru. Seznam **corners** obsahuje souřadnice rohů jednotlivých markerů, kde jsou rohy jednoho markeru vždy uspořádány v tomto pořadí: levý horní(tl), pravý horní(tr), pravý spodní(br), levý spodní(bl). Vzhledem k tomu, že má marker tvar čtverce, mohu jednoduše sečíst souřadnice levého horního a pravého spodního rohu a součet vydělit dvěma, čímž získám souřadnice středů markerů, které budou sloužit jako rohy ohraničujícího obdélníku. Souřadnice středů uložím do seznamu **MarkerCenters**, ID do seznamu **MarkerIDs** a rohy do **MarkerCorners**. U takto uložených dat vím, že data na jednotlivých pozicích ve všech třech seznamech patří k sobě.

Abych se vyhnul chybnému ukládání markerů, které nemají sloužit k ohraničení pracovního prostoru, přidám podmínku, že ID markeru musí být v předdefinovaném seznamu: **workspaceIds**.

```
    if len(corners)>0:
        #Vykresli detekovane markery na snimek
        cv2.aruco.drawDetectedMarkers(frame, corners, ids)
        for markerCorners, markerId in zip(corners,ids):
            if markerId in workspaceIds:
                markerCorners = markerCorners.reshape((4, 2))
                tl, tr, br, bl = markerCorners
                #bottom_right-pravy spodni ,top_left-levy spodni
                br = (int(br[0]), int(br[1]))
                tl = (int(tl[0]), int(tl[1]))
                #Vypocet souradnice stredu markeru
                CX = (tl[0] + br[0]) / 2
                CY = (tl[1] + br[1]) / 2
                #Ulozeni dat
                MarkerCenters.append([CX, CY])
```

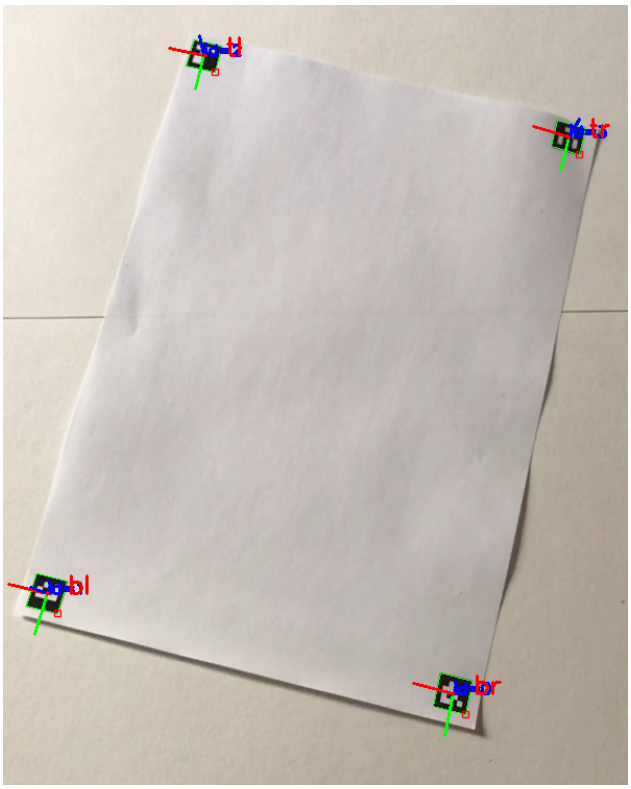
```
76                  MarkerIds.append(markerId[0])
77                  MarkerCorners.append(markerCorners)
```

Zdrojový kód 5.8: Zpracování detekovaných ArUco markerů

Pomocí funkce ***cv2.aruco.estimatePoseSingleMarkers()***[60] zjistím vektory posuvu a rotace markeru vůči kameře. Později pro mě budou důležité vektory posuvu, které uložím do seznamu **tvecs(translation vectors - vektory posuvu)**. Vstupními proměnnými pro funkci jsou v tomto pořadí: rohy jednoho markeru, reálná velikost markeru,matice kamery a koeficienty zkreslení kamery.

```
81          #Odhadne posuv a natoceni kazdeho markeru vuci kamere
82          for corner in sortedCorners:
83              rvec, tvec, _ = cv2.aruco.estimatePoseSingleMarkers(
    corner,RealMarkerSize,mtx, dist)
84              tvecs.append(tvec)
85              #Vykresli osy markeru
86              cv2.drawFrameAxes(frame, mtx, dist, rvec, tvec, 30, 2)
```

Zdrojový kód 5.9: Získání rotačních a translačních vektorů každého markeru



Obr. 5.5: Detekované ArUco markery

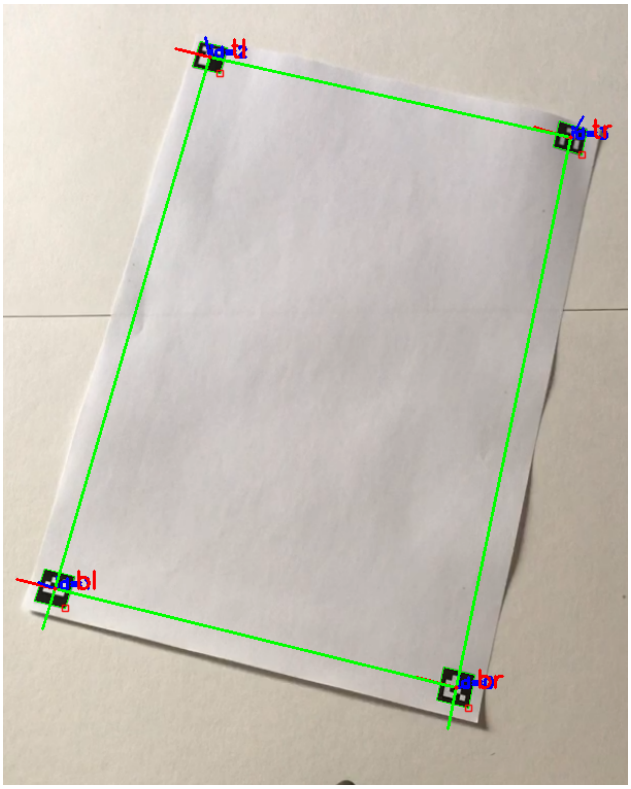
V tuto chvíli můžu přejít k vytvoření samotného pracovního prostoru z detekovaných markerů. Následující kód bude proveden, jestliže jsou nalezeny více jak 3 markery.

Nejříve seřadím středy jednotlivých makerů, které už jsou pro mě v tuto chvíli rohy pracovního prostoru, v pořadí: levý horní(tl), pravý horní(tr), pravý

spodní(br), levý spodní(bl) a uložím je do nového seznamu ***boundaryPoints***. Na základě takto seřazených rohů přerovnám seznam ***tvecs*** do nového seznamu ***cornersTvecs*** tak, aby index vektoru odpovídal indexu korespondujícího rohu v seznamu ***boundaryPoints***. Nakonec vykreslím ohraničení pracovního prostoru do původního obrázku(viz obr.5.6).

```
91      #Pokud je detekovano vice jak 3 markery
92       if len(MarkerCenters)>3:
93           #Seradi rohy PP: tl, tr, br, bl ve smeru hod rucicek
94           boundaryPoints = sortptsClockwise(np.array(MarkerCenters))
95           corners_tvecs = []
96           for corner in boundaryPoints:
97               # nalezeni indexu tvecu, ktery odpovida danemu rohu
98               i = np.where((sortedCenters==corner).all(axis=1))[0][0]
99               # ulozeni tvecu do listu s poradim tl, tr, br, bl
100              cornersTvecs.append(tvecs[i])
101          #Vykresli ohraniceni kolem pracovni plochy a popise body na
         puvodnim obrazku
102          a=np.array(boundaryPoints).reshape((-1,1,2)).astype(np.
       int32)
103          cv2.drawContours(frame,[a], 0, color, 2)
```

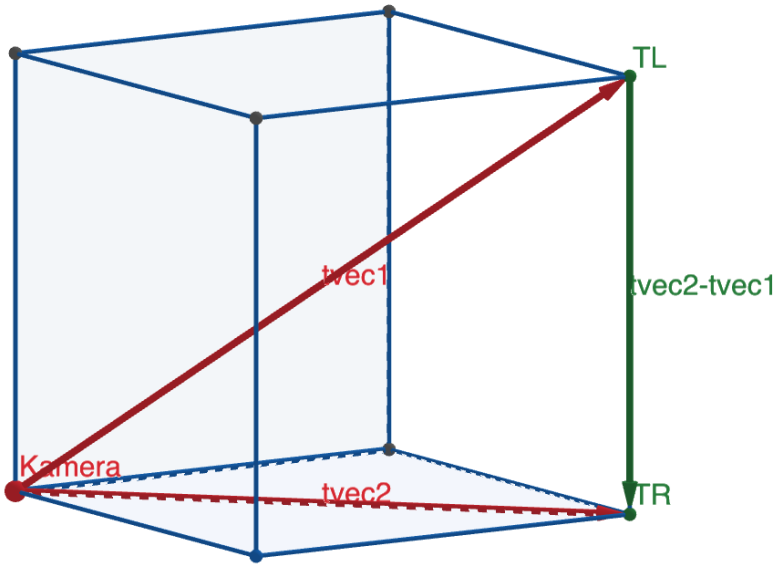
Zdrojový kód 5.10: Ohraničení pracovního prostoru



Obr. 5.6: Detekovaný pracovní prostor

Z posouvajících vektorů teď mohu spočítat délky stran obdélníku tvořeného 4 markery. Je si nutno uvědomit, že kamera může být natočena vzhledem k pracovní

rovině,tím pádem mohou být i složky vektorů ve směru Z nenulové. Předchozí obrázek je pořízen s kamerou, která je natočena ve dvou směrech vůči pracovní ploše. Délku základny spočítám jako velikost vektoru vzniklého rozdílem vektorů k levého horního(tl) a pravého horního rohu(tr). Obdobně pak získám výšku z velikosti rozdílu vektorů pravého spodního a pravého vrchního rohu. K výpočtu velikosti vektoru použiji funkci ***np.linalg.norm()***[61] z knihovny NumPy. Ze zjištěné šířky a výšky spočítám poměr stran obdélníku.



Obr. 5.7: Znázornění vektorů

```
110         # Vypocet delky stran
111         #sirka obdelniku
112         tltr = np.linalg.norm(cornersTvecs[1] - cornersTvecs[0])
113         #vyska obdelniku
114         brtr = np.linalg.norm(cornersTvecs[2] - cornersTvecs[1])
115         aspect_ratio = tltr / brtr  #pomer stran: sirka/vyska
```

Zdrojový kód 5.11: Výpočet poměru stran pracovního prostoru

Obraz detekovaného pracovního prostoru musím nyní transformovat tak, aby byl zdánlivě rovnoběžný s kamerou, z důvodu pozdější jednodušší manipulace - veškeré body budu schopen měřit už jen ve dvourozměrném prostoru a nebude třeba každou souřadnici transformovat podle natočení kamery vůči pracovní ploše . Toho docílím pomocí funkce ***cv2.warpPerspective()***[62] z knihovny OpneCv, která obraz transformuje a zároveň i ořízne - dále nazývaný jako nový obraz. Vstupem pro funkci jsou tyto proměnné: obraz k transformaci,transformační matice,šířka a výška nového obrazu a další doplňující parametry, které nejsou v tuto chvíli až tak důležité.

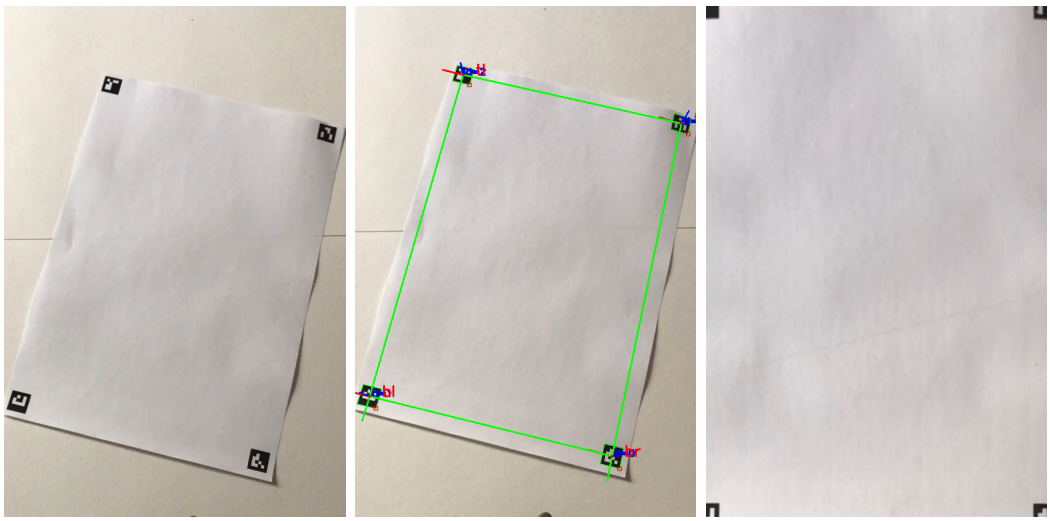
Nejdříve určím šířku a výšku nového obrazu. Šířku zvolím stejnou jako je šířka původního snímaného obrazu. Tento rozměr může být volitelný a v podstatě bude jen určovat velikost výsledného nového obrazu. Ze zvolené šířky pak dopočítám

výšku pomocí předtím spočítaného poměru stran ohraničujícího obdélníku, který musím zachovat, aby nedošlo ke zkreslení nového obrazu.

Jako druhou musím určit transformační matici, která lze jednoduše vypočítat pomocí funkce ***cv2.getPerspectiveTransform()***[63]. Vstupem pro tuto funkci jsou dvě proměnné: první je seznam 4 vstupních bodů. Druhou proměnnou je seznam 4 bodů, do kterých budou 4 vstupní body přesunuty v novém obraze. Jako vstupní body použijeme rohy ohraničujícího obdélníku a jako výstupní body použiji rohy nového obrazu. Jakmile mám matici spočítanou, mám vše potřebné k transformaci a použiji zmíněnou funkci ***cv2.warpPerspective()***[62]. Jejím výstupem je nový obraz - transformovaný a oříznutý původní obraz(viz obr.5.8), který v kódu odpovídá proměnné **warpedCropped** a bude jedním z výstupů funkce ***workspace()***.

```
116         #sirka noveho obrazu
117         new_width = int(ww)
118         #vyska n. obrazu
119         new_height = int(new_width / aspect_ratio)
120         #vstupni body
121         input = np.float32(boundaryPoints[0:4])
122         #vystupni body
123         output = np.float32([[0, 0], [new_width, 0], [new_width,
    new_height], [0, new_height]])
124         # transformacni matice
125         matrix = cv2.getPerspectiveTransform(input,output)
126         warpedCropped = cv2.warpPerspective(raw_frame, matrix,(
    new_width, new_height),cv2.INTER_LINEAR, borderMode=cv2.
    BORDER_CONSTANT, borderValue=(0, 0, 0))
```

Zdrojový kód 5.12: Transformace obrazu



(a) Původní obraz (b) Ohraničený PP (c) Oříznutý PP

Obr. 5.8: Výstupní obrázky

Posledním krokem v této kapitole je zjistit převodní koeficient mezi pixely a milimetry, který budu potřebovat později k přepočítávaní souřadnic z pixelů na

milimetry. K tomuto účelu jsem vytvořil funkci ***GetConversionCoeficient()***. Vstupy pro funkci jsou: souřadnice rohů markerů, reálná velikost strany markeru a transformační matice. Výstupem funkce je převodový koeficient říkající kolik mm se vejde do jednoho pixel[mm/px].

```
127         conversion = GetConversionCoeficient(MarkerCorners,
    RealMarkerSize,matrix)
```

Zdrojový kód 5.13: Funkce GetConversionCoeficient() uvnitř funkce workspace()

Funkce vychází ze známe velikosti strany ArUco markeru, kterou musím algoritmu dodat(skrze soubor **config.ini**). Jakmile znám velikost strany markeru v milimetrech i v pixelech mohu jednoduše získat převodní koeficient jejich podělením. Jako první přetransformuji souřadnice rohů markerů ze souřadnic odpovídajících původnímu obrazu na souřadnice odpovídající novému obrazu pomocí funkce *transformPoint()*(viz support.py). Následně spočítám velikosti jednotlivých stran markeru a sečtu je - tím získám obvod markeru, který uložím do seznamu obvodů markerů.

```
25 def GetConversionCoeficient(MultipleMarkersCorners, MarkerSize,
    TransformationMatrix):
26     obvod_list = []
27     for corners in MultipleMarkersCorners:
28         #trasformace souradnic rohu markeru
29         transformed_corners = [transformPoint(p,
    TransformationMatrix) for p in corners]
30         tl, tr, br, bl = transformed_corners
31         #vypocet delek stran markeru
32         tltr = math.hypot(tl[0] - tr[0], tl[1] - tr[1])
33         trbr = math.hypot(tr[0] - br[0], tr[1] - br[1])
34         blbr = math.hypot(bl[0] - br[0], bl[1] - br[1])
35         bltl = math.hypot(bl[0] - tl[0], bl[1] - tl[1])
36         #vypocet obvodu markeru
37         obvod = tltr + trbr + blbr + bltl
38         obvod_list.append(obvod)
```

Zdrojový kód 5.14: Definování funkce GetConversionCoeficient()

Ze všech spočítaných obvodů vypočítám průměrný obvod, ze kterého po vydělení 4 dostanu průměrnou velikost jedné strany markerů v pixelech. Nakonec podělím reálnou velikost markeru s velikostí v pixelech, čímž získám kýžený převodní koeficient v jednotkách [mm/px], tedy kolik mm se vejde do jednoho px. Koeficient je jediným výstupem funkce ***GetConversionCoeficient()***.

```
39     avrg_obvod = sum(obvod_list) / len(obvod_list)
40     a = avrg_obvod / 4
41     #konecny vypocet prevodoveho koeficientu
42     conversion = MarkerSize / a [mm/px]
43     return conversion
```

Zdrojový kód 5.15: Výpočet převodního koeficientu ze známé délky strany v mm i px

V tuto chvíli mám připraveno vše, co se týče vytyčení pracovního prostoru - přetransformovaný a oříznutý obraz obsahující pouze region zájmu a převodní koeficient k přepočtu rozměrů z px do mm.

```
128     #Konec funkce worksapce()
129     return warpedCroppedFrame, frame ,conversion
```

Zdrojový kód 5.16: Konec funkce workspace()

Proměnná	**Popis**
warpedCroppedFrame	přetransformovaný a oříznutý obraz pracovního prostoru, obsahuje pouze region zájmu(viz obr.5.8c)
frame	původní obraz doplněný o vykreslený pracovní prostor a detekované ArUco markery(viz obr.5.8b)
conversion	převodní koeficient [mm/px]

Tabulka 5.2: Přehled výstupných proměnných funkce workspace()

Obr. 5.9: Vývojový diagram funkce workspce()

5.4 Detekce fixačních ploch a zásobníku s roztokem

V této kapitole se dostávám do jádra fungovaní algoritmu. V předchozích krocích jsem kalibroval kameru a následně jsem získal obraz oříznutého pracovního prostoru, ve kterém se nyní můžu pustit do samotné detekce fixačních ploch a případně i zásobníků s roztokem. Veškerý kód v této kapitole je ze souboru ***main.py*** (viz obr.5.1), který je zároveň spouštěcím souborem pro celý algoritmus strojového vidění.

Z popsaných přístupů detekce objektů v teoretické kapitole 2.3 jsem vybral pro toto řešení opět detekci pomocí ArUco markerů v propojení s detekcí kontur. Důvodem této volby je, že potřebuji detekovat pouze velmi jednoduché tvary - obdélník: fixační plocha. Při stanovení několika pravidel jak by měly být plochy zviditelněny, budu schopen vytvořit relativně robustní řešení. Naopak při použití detektorů na principu charakteristických rysů nebo strojového učení by došlo ke zbytečnému zvýšení komplexity kódu, která není vzhledem k jednoduchosti tvarů nutná.

Fixační plochy budou označeny takovým způsobem, že po jejích okrajích bude tlustá černá linie a uvnitř tohoto ohraničení bude libovolně umístěn ArUco marker (viz obr.5.13). Fixační plocha, která s ní bude tvořit dvojici musí být označena markerem s ID o 1 vyšší. Startovací plocha je vždy označena sudým ID - dvojce budou tedy tvořeny například takto: 01;23;45;67;.., kde první číslo je ID startovací plochy a druhé je ID koncové plochy. Pro zjednodušení jsou fixační plochy ve zdrojovém kódu většinou popisovány jako desky(boards).

Zásobník s roztokem bude označen pouze pomocí ArUco markeru tím způsobem, že bude umístěn na dně zásobníku - podmínkou pro jeho detekci je tedy nutnost dostatečné transparentnosti roztoku. ArUco markery budou generovány ze slovníku `cv2.aruco.DICT_6X6_250` na rozdíl od slovníku, který byl použit k ohraničení prostoru: `cv2.aruco.DICT_4X4_50`. Slovníky lze libovolně měnit, prostřednictvím konfiguračního souboru **config.ini**

Jako první importuji používané knihovny, funkci ***workspace()*** vytvořenou v předchozí kapitole a další pomocné funkce ze souboru **support**. Jako poslední načtu kalibrační data se souboru ***calibration.npz***. Definuji seznamy ***boardIds***, které definují rozsah IDs, které se budou používat pro fixační plochy a ***reservoirIds***, které se budou používat pro zásobníky s roztokem.

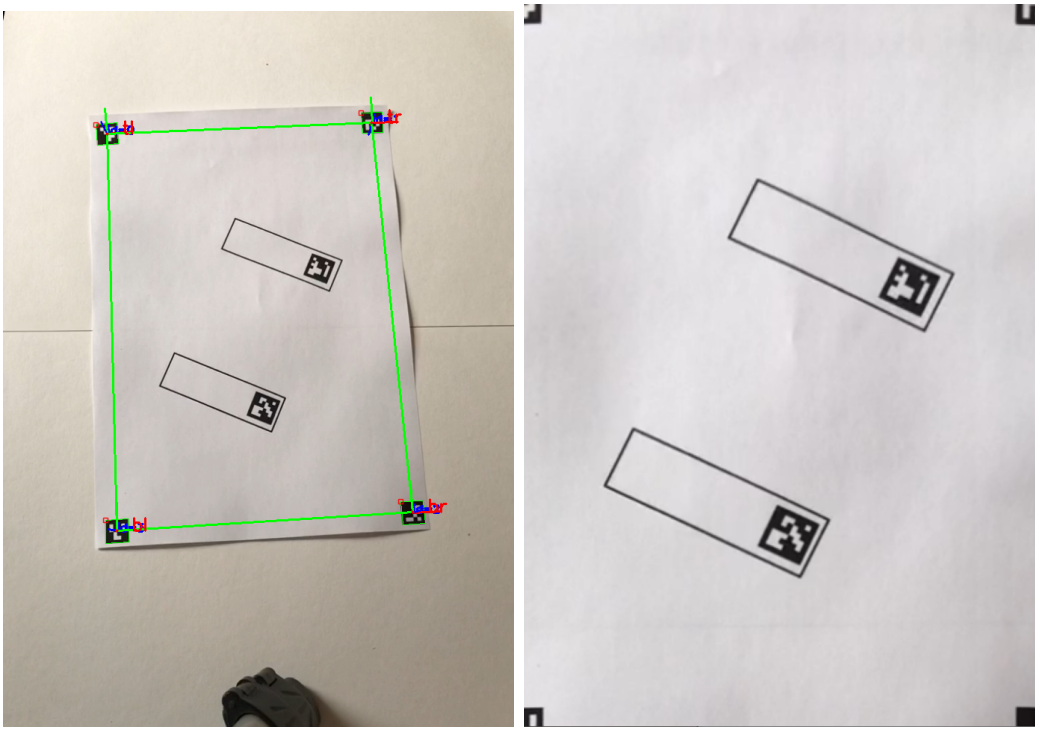
```
1  import cv2
2  import numpy as np
3  import math
4  from support import sortptsClockwise
5  from worksapce import workspace
6  #Nacte data pro undisort
7  data = np.load('calibration.npz')
8  #Definuje seznamy
9  mtx, newmtx, dist = [data[i] for i in ('mtx', 'newmtx', 'dist')]
10 boardIds = [0, 1, 2, 3,4,10]
11 reservoirIds = [240]
```

Zdrojový kód 5.17: Začátek hlavního souboru main.py - import knihoven; kalibračních dat;funkce workspace() a dalších.

V prvním korku je potřeba získat záznam z kamery pomocí funkce ***cv2.VideoCapture()*** [64]. Následně definuji smyčku while, která poběží dokud bude dostávat záznam z kamery - tím je docíleno toho, že bude každý jednotlivý snímek ze záznamu zpracován v reálném čase. Pomocí funkce ***cv2.read()***[65] načtu aktuální snímek **frame** ze záznámu **cap**, se kterým budu dál pracovat uvnitř smyčky. Snímek je následně zbaven zkreslení pomocí funkce ***cv2.undistort()***[66], kde jsou vstupními parametry data získaná během kalibrace(viz tabulka 5.1). Snímek zbavený zkreslení je vstupem pro funkci ***workspace()*** vytvořené v předchozí kapitole. Nejdůležitějším výstupem z této funkce je transformovaný a oříznutý snímek(**warpedCroppedFrame**), ve kterém se budou odehrávat všechny následující procesy. Aby však mohly probíhat, nesmí funkce ***workspace()*** vrátit prázdnou proměnnou.

```
22 if __name__ == "__main__":
23     #Ziskani videa z kamery
24     cap = cv2.VideoCapture(0)
25     #Dokud je zaznamenavani spusteno:
26     while cap.isOpened():
27         #nacteni snimku
28         ret, frame = cap.read()
29         #odstraneni zkresleni kamery
30         frame = cv2.undistort(frame, mtx, dist, None, newmtx)
31         #ziskani snimku z funkce workspace
32         warpedCroppedFrame, wholeFrame, prevod, warped = workspace(
   frame)
33         #zobrazi puvodni upraveny snimek
34         cv2.imshow('Undisorted', wholeFrame)
35         #pokud snimek neni None
36         if warpedCroppedFrame is not None:
37             frame = warpedCroppedFrame
38             raw_frame = frame.copy()
```

Zdrojový kód 5.18: Začátek algoritmu



(a) Detekovaný pracovní prostor (b) Transformovaný a oříznutý obraz

Obr. 5.10: Vstupní obraz

Jakmile je vše připraveno a snímek obsahuje obraz, mohu na něm detekovat ArUco markery a následně dopočítávat jejich středy, obdobně jako v předchozí kapitole (Kód 5.7 a 5.8). Rozdílná bude podmínka, že pokud bude detekován marker s ID, kterému přísluší zásobník s roztokem(je v seznamu reserovirIDs), bude souřadnice středu markeru brána rovnou jako souřadnice zásobníku. U zásobníku není potřeba nutně znát jeho další rozměry, stačí dodržet, že je marker uvnitř zásobníku a tak bude pipeta posílána nad jeho střed. Do slovníku uložím střed markeru a ID markeru zásobníku a tento slovník uložím do seznamu **reservoirs**. Zbytek středů uložím do seznamu **MarkerCenters** a ID do seznamu **MarkerIds**.

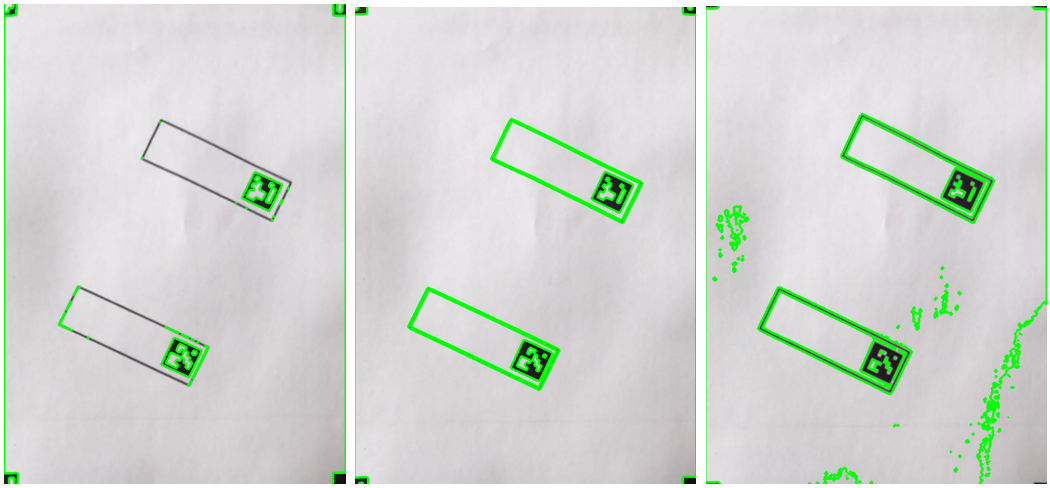
```
63              if markerId in reservoirIds:
64                  # Ulozeni informaci do slovniku
65                  #a pak do seznamu reservoirs
66                  dct =  {
67                          "center": [CX, CY],
68                          "id": markerId[0]
69                          }
70                  reservoirs.append(dct)
71              else:
72                  MarkerCenters.append([CX, CY])
73                  MarkerIds.append(markerId[0])
74                  MarkerCorners.append(cors)
```

Zdrojový kód 5.19: Zpracování detekovaných ArUco markerů uvnitř pracovního prostoru

Po detekci markerů se pustím do detekce kontur. Úkolem je nalézt černé ohraničení kolem fixačních ploch, uvnitř kterých se vyskytují ArUco markery. Základním krokem je převod snímku z barevného na černobílý pomocí funkce ***cv2.cvtColor()***. [67] Detekce kontur je tak efektivnější a zároveň následně stačí určit prahovou hodnotu intenzity pouze pro černou barvu, kterou určíme na druhé pozici ve funkci ***cv2.threshold()***[68]. Prahová hodnota jednoduše řečeno určuje jak výrazné kontury budou detekovány, tuto hodnotu je často vhodné nastavit vzhledem ke konkrétní kompozici. Funkce ***cv2.findContours()***[7] pak nalezne všechny kontury na snímku, které mají intenzitu nad prahovou hodnotou a zároveň vrátí hierarchii kontur, která popisuje vztahy mezi konturami(viz[69]).

```
84 # Detekce kontur uvnitr pracovniho prostoru
85 # Zmeni snimek na cerno bily
86 imgrey = cv2.cvtColor(raw_frame, cv2.COLOR_BGR2GRAY)
87 #Tresholding cernobileho snimku
88 ret, thresh = cv2.threshold(imgrey, 125, 255, 0)
89 #Nalezne kontury
90 contours, hierarchy = cv2.findContours(thresh, cv2.RETR_TREE, cv2.
       CHAIN_APPROX_SIMPLE)
```

Zdrojový kód 5.20: Detekce kontur uvnitř pracovního prostoru



Obr. 5.11: Znázornění detekovaných kontur při různě nastavených prahových hodnotách, zleva: 65 - 125 - 200

Dalším úkolem je z detekovaných kontur vybrat ty kontury, které přísluší ohraničeným plochámm. Jako první vyseparuji nejvyšší kontury pracovní plochy a uložím je do seznamu **highestParentContours**.

```
97 for i in range(len(contours)):
98     if hierarchy[i][3] == -1:
99         highestParentContours.append(i)
100    else:
101        continue
```

Zdrojový kód 5.21: Nalezení nejvyšších kontur v obraze

Následně najdu všechny kontury, uvnitř, kterých se vyskytuje ArUco marker. Toho docílím pomocí funkce ***pointPolygonTest()***[70], pomocí které lze zjistit vzdálenost bodu od polygonu. Pokud je zjištěná vzdálenost záporná -> bod se nachází uvnitř polygonu. Takto proti sobě ve dvou sdružených for smyčkách postavím kontury a středy markerů a zjistím všechny kontury ležící kolem jednotlivých ArUco markerů. Pokaždé, kdy je detekována kontura kolem Markeru, je uložena do seznamu **parentContours** a ID markeru do seznamu **coresponding-Markers**

```
104 for i in range(len(contours)):
105     for j in range(len(MarkerCenters)):
106         if cv2.pointPolygonTest(contours[i], tuple(MarkerCenters[j
    ]), False) >= 0:
107             parentContours.append(hierarchy[i][3])
108             corespondingMarkers.append(MarkerIds[j])
109             break
110         else:
111             continue
```

Zdrojový kód 5.22: Nalezení konur kolem ArUco markerů

Jako poslední musím získat nejvyšší konturu ze všech nalezených kontur kolem markeru. Toho docílím pomocí následující podmínky: pokud je rodič kontury markeru v seznamu nejvyšších kontur(**highestParentContours**) -> jedná se o nejvyšší konturu markeru a tedy o ohraničující konturu fixační plochy. ID kontury uložím do seznamu **markerParentContourIds** a ID příslučejícího markeru do seznamu **conturedMarkerIds**.

```
116     # Nalezne nejvyssi rodice ArUco markeru, uvnitr pracovni plochy
117     markerParentContourIds = []
118     conturedMarkerIds = []
119     for i, id in zip(parentContours, corespondingMarkers):
120         if hierarchy[i][3] in highestParentContours:
121             markerParentContourIds.append(i)
122             conturedMarkerIds.append(id)
123         else:
124             continue
```

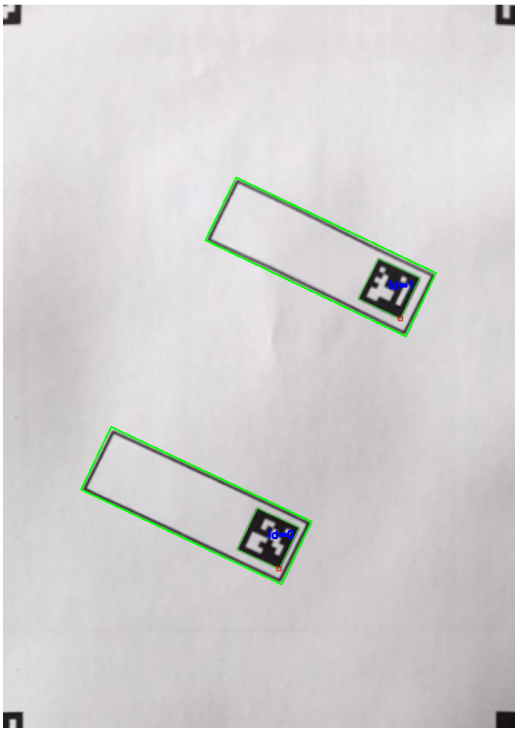
Zdrojový kód 5.23: Nalezení nejvyšších kontur kolem ArUco markeru.

Poté co jsem nalezl detekovanou konturu, musím zjistit jestli jde opravdu o správnou konturu - viditelná fixační plocha by měla být obdélníkem a její ohraničení(detekovaná kontura) by tedy taky mělo kopírovat obdelník. O tom se přesvědčím prostřednictvím funkce ***cv2.approxPolyDP()***[70], která slouží ke zjednodušení kontur nebo tvarů skládáním více kontur do jedné a vrací seznam nových kontur. Pro nás je důležitou informací délka seznamu -> pokud obsahuje 4 kontury -> jedná se o 4-úhelník -> můžeme předpokládat, že jde o hledanou fixační plochu. Další příkazy budou provedeny za předpokladu, že je splněna podmínka 4-úhelníku.

```
129     for i,marker_id in zip(markerParentContourIds,
    conturedMarkerIds):
130         #aproximace kontury
```

```
131         approx = cv2.approxPolyDP(contours[i], 0.04 * cv2.arcLength
    (contours[i], True), True)
132         #Pokud je kontura(okolo ArUco markeru) 4-uhelnikem -->
133         #--> jde o desku
134         if len(approx) == 4:
```

Zdrojový kód 5.24: Kontrola ohraničení fixační plochy



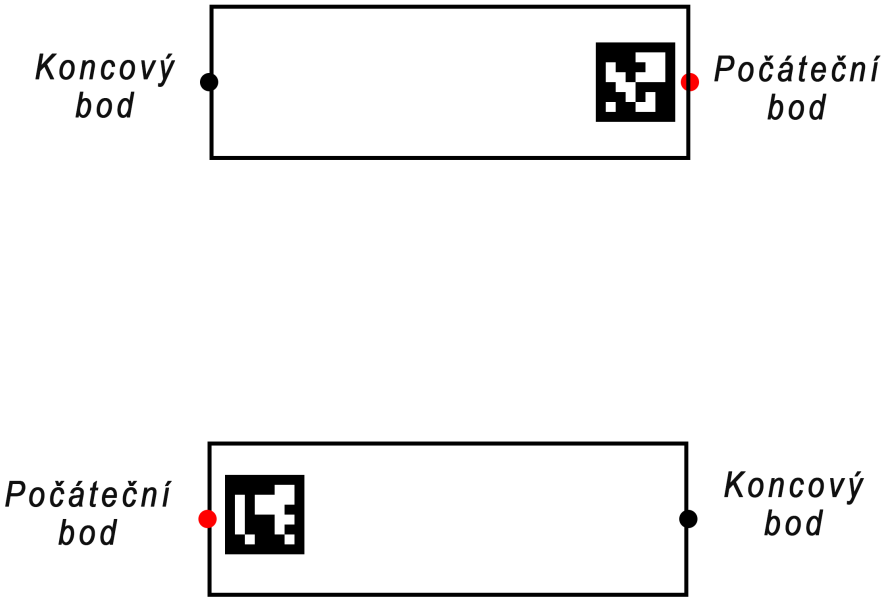
Obr. 5.12: Detekované fixační plochy

Pokud podmínka splněna není, je daná kontura vykreslena červenou barvou.

```
180         else:
181             cv2.drawContours(frame, contours, i, (0, 0, 255), 3)
```

Zdrojový kód 5.25: Vykreslení chybně nalezených kontur

V tuto chvíli mám nalezenou správnou konturu pro každý marker a můžu tak určit souřadnice fixačních ploch. Předtím než toto budu schopen provést, si nejdříve musím definovat, jaké body fixační plochy budou výstupními souřadnicemi celého algoritmu. Zvolím tedy, že startovacím či konečným bodem na jedné desce bude vždy střed kratší strany fixační plochy. O tom kde bude startovací či koncový bod rozhodne poloha markeru - na té straně obdélníku, na které se nachází marker bude startovací bod a na druhé straně bod koncový.Přehledněji je to znázorněno na následujícím obrázku .



Obr. 5.13: Logika určování bodů

Detekovanou konturu obepnu obdélníkem pomocí funkce ***cv2.minAreaRect()***[70], čímž získám pravidelný obdélník. Výstupem funkce je proměnná obsahující data o středu, délkách stran a úhlu natočení obdélníku. Tuto proměnnou mohu přímo dosadit do funkce ***cv2.boxPoints()***[71], která mi v jednom kroku vrátí seznam souřadnic všech 4 vrcholů obdélníku, se kterými budu dál pracovat. Zároveň si ze seznamu **MarkerCenters** vytáhnu, prostřednictvím ID, souřadnice středu příslušejícího markeru.

```
133         #Pokud je kontura(okolo ArUco markeru) 4-uhelnikem -->
134         #--> jde o desku
135         if len(approx) == 4:
136             rect = cv2.minAreaRect(contours[i])
137             box = cv2.boxPoints(rect)
138             box = np.int0(box)
139             index = MarkerIds.index(marker_id)
140             markerCenter = MarkerCenters[index]
```

Zdrojový kód 5.26: Aproximace kontury pomocí obdélníku

V dalším kroku si definuji dvě jednoduché funkce pro pozdější zjednodušení kódu. První z nich je funkce ***distance()***, pomocí, které spočítám vzdálenost mezi dvěma body pomocí Pythagorovy věty. Druhá je funkce ***midpoint()***, pomocí, které zjistím středový bod mezi dvěma body.

```
139             def distance(p1, p2):
140                 return math.sqrt((p1[0] - p2[0]) ** 2 +
                            (p1[1] - p2[1]) ** 2)
141             #vypocita stredovy bod mezi dvema body.
142             def midpoint(p1, p2):
```

```
143                 return [(p1[0] + p2[0]) / 2, (p1[1] + p2[1]) / 2]
```

Zdrojový kód 5.27: Definování funkcí distance() a midpoint()

Pomocí funkce ***distance()*** spočítám délky dvou na sebe kolmých stran obdélníku. Následně pomocí logických funkcí zjistím, která ze stran je kratší a naleznu souřadnice jejich středu pomocí funkce ***midpoint()***.

```
144             # Spocita velikost stran obdelniku
145             a = distance(box[0], box[1])
146             b = distance(box[1], box[2])
147             # Zjisti, ktera strana je delsi
148             if a < b:
149                 center1 = midpoint(box[0], box[1])
150                 center2 = midpoint(box[2], box[3])
151             else:
152                 center1 = midpoint(box[1], box[2])
153                 center2 = midpoint(box[3], box[0])
```

Zdrojový kód 5.28: Výpočet velikostí stran fixačních ploch

Nyní už jen stačí zjistit, který střed bude startovací bod a který bod koncový. Podle popsáné logiky bude startovacím bodem ten střed, který leží blíž od ArUco marker. Spočítám tedy vzdálenost středů od středu ArUco markeru a vzájemně je porovnám. Střed s kratší vzdáleností od markeru je vždy počátečním bodem a druhý bod je koncovým bodem. Počáteční body jsou fixačními body první operace tažení, koncové body jsou naopak fixačními body poslední operace tažení.

```
156             #Spocta vzdalenost stredu stran od stredu markeru
157             c1c = distance(center1, markerCenter)
158             c2c = distance(center2, markerCenter)
159             #porovna vzdalenost stredu stran od stredu markeru
160             if c1c < c2c:
161                 start_point = center1
162                 end_point = center2
163             else:
164                 start_point = center2
165                 end_point = center1
```

Zdrojový kód 5.29: Určení startovacích a koncových bodů na fixačních plochách

Získané informace o fixační ploše uložím do slovníku a slovník následně do seznamu **boards**. ID markeru bude od této chvíle také ID fixační plochy a uložím jej do seznamu **boardIDs**

```
166             # Ulozeni informaci o desce do slovniku
167             dct = {
168                     "start_point": start_point,
169                     "end_point": end_point,
170                     "marker_id": marker_id,
171                     "marker_center": markerCenter,
172                     "box": box
173                     }
174             # Ulozeni informaci o desce do seznamu desek
175             boards.append(dct)
176             # Ulozeni ID o desce do seznamu IDs desek
```

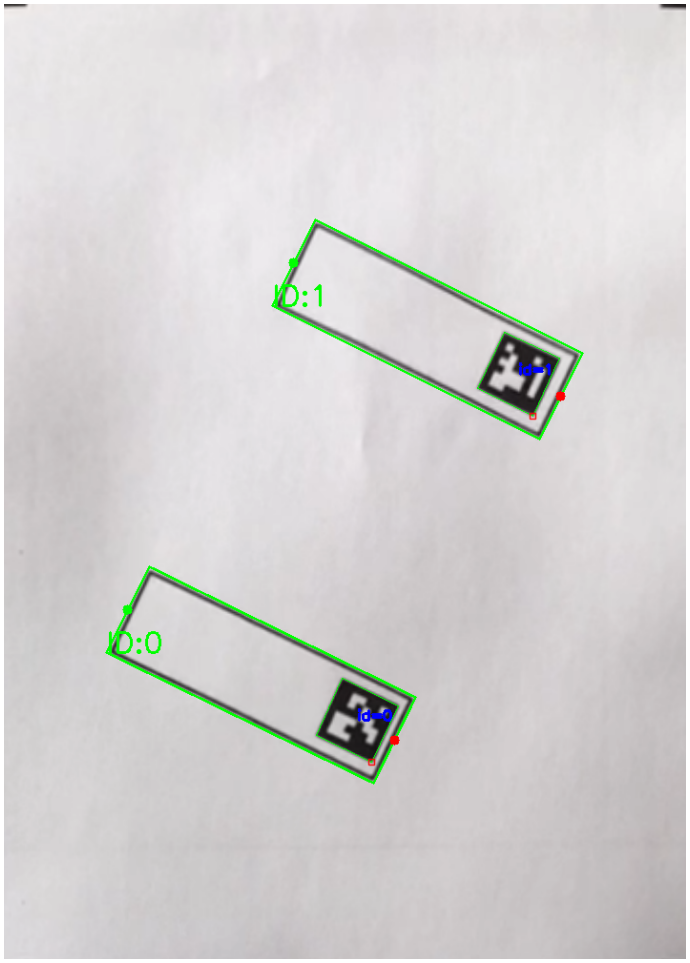
```
177                 boardIDs.append(marker_id)
```

Zdrojový kód 5.30: Uložení dat o plochách do slovníku

Jako poslední pomocí několika funkcí vykreslím detekované kontury, ID ,startovací a koncové body.

```
166 # Nakresleni kontury desky
167 cv2.drawContours(frame, [box], 0, (0, 255, 255), 2)
168 # vypise ID desky
169 cv2.putText(frame, f"ID:{marker_id}", (int(box[0][0]), int(box
    [0][1])),cv2.FONT_HERSHEY_SIMPLEX, 0.5, (0, 255, 255), 2)
170 # Vykresli startovaci a koncovy bod
171 cv2.circle(frame, (int(start_point[0]), int(start_point[1])), 5,
    (0, 0, 255), -1)
172 cv2.circle(frame, (int(end_point[0]), int(end_point[1])), 5, (0,
    255, 255), -1)
```

Zdrojový kód 5.31: Vyobrazení detekované plochy do pořízeného obrazu



Obr. 5.14: Detekované fixační plochy a startovací/koncové body

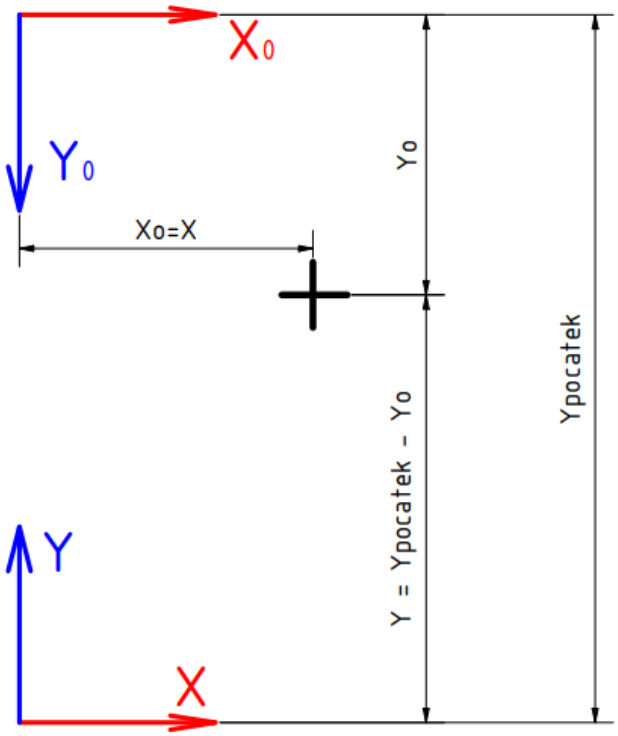
5.5 Zpracování získaných dat

Hlavní část algoritmu mám nyní vytvořenou - z obrazu jsem schopen získat všechna potřebná data o fixačních plochách a zásobnících. Ty však musím ještě zpracovat a následně exportovat v patřičném formátu.

Předtím než se pustím do práce s daty, si vytvořím funkci ***TransformToRealCoordiantes()***, která mi bude sloužit k převodu souřadnic bodu z pixelů do mm a zároveň ze souřadnicového systému snímku do požadovaného souřadnicového systému stroje. Vstupem pro funkci jsou souřadnice bodu, souřadnice počátku nového souřadnicového systému a převodní koeficient získaný z funkce ***workspace()***. Abych převedl souřadnice ze souřadnicového systému snímku do souřadnicového systému stroje, musím odečíst Y ovou souřadnici bodu od Y ové souřadnice počátku stroje. Souřadnice ve směru X jsou identické. Počátek souřadnicového systému stroje uvažujeme vlevo dole.(viz obr 5.15). Nové souřadnice už jen vynásobím převodním koeficientem [mm/px] a tak získám souřadnice v mm. Výstupem jsou tak souřadnice bodu v mm.

```
16 def TransformToRealCoordiantes(point,origin,prevod):
17     X = round(point[0]*prevod,2)
18     Y = round((origin[1] - point[1])*prevod,2)
19     return (X,Y)
```

Zdrojový kód 5.32: Definování funkce TransformToRealCoordinates()



Obr. 5.15: Přepočet souřadnic

Aby mohla probíhat samotná operace tažení vláken, je nutné mít v pracovním prostoru alespoň jednu dvojici fixačních ploch. Musím tedy zjistit, zda se v prostoru taková dvojice nachází a pokud ano, sdružit data o těchto fixačních plochách k sobě. Toho docílím pomocí několika logických funkcí.

Nejdříve zjistím pokud je v pracovním prostoru více jak 1 fixační plocha -> pokud ano, projedu postupně seznam **boardIDs** a ke každé fixační ploše hledám příslušného partnera, id partnera je ±1 od aktuálního ID -> pokud je partner v seznamu **boardIDs**, přidám dvojici do seznamu **coupleIDs**. Pokud v seznamu partner není -> přidám fixační plochu do seznamu **singleIDs**. Takto projedu celý seznam a zjistím všechny existující dvojce.

```
191     if boards != []:
192         if len(boards) >1:
193             boardIDs = list(set(boardIDs))
194             i = 0
195             # Rozdeleni desek na dvojice a jednotlive desky
196             for id in boardIDs:
197                 #pokud je ID sude --> partner je ID + 1
198                 if id % 2 == 0:
199                     partner = id + 1
200                 #pokud je ID liche --> partner je ID - 1
201                 else:
202                     partner = id - 1
203                 # Pokud je nalezen partner v boardIDs
204                 if partner in boardIDs:
205                     couple  = tuple(sorted([id, partner]))
206                     # Pokud dvojice desek jeste neni
207                     #v seznamu dvojic
208                     if couple not in coupleIDs:
209                         #prida dvojici do seznamu dvojic
210                         coupleIDs.append(couple)
211                 # Pokud neni nalezen partner v boardIDs
212                 # prida ID desky do seznamu single desek
213                 elif id not in singleIDs:
214                         singleIDs.append(id)
```

Zdrojový kód 5.33: Sdružení desek do dvojic

Jakmile mám dvojici desek, extrahuji data o obou plochách ze seznamu **boards**, který jsem vytvořil v předchozí kapitole a obsahuje data o počátečních/koncových bodech a identifikačních číslech, podle kterých data vyhledám.

```
217         # Pokud je nalezena alespon dvojice desek
218         if len(coupleIDs) > 0:
219             # Ziskani informaci o dvojici desek ze seznamu ploch,
220             # promoci ID desek
221             for couple in coupleIDs:
222                 for board in boards:
223                     # Pokud je ID desky v dvojici stejne jako ID
224                     # desky v seznamu desek jde o plochu z dvojice
225                     if board["marker_id"] == couple[0]:
226                         board1 = board
227                     elif board["marker_id"] == couple[1]:
```

```
228                         board2 = board
```

Zdrojový kód 5.34: Získání dat o fixačních plochách ve dvojici

U obou desek z dvojice převedu souřadnice startovacích a koncových bodů z px na mm pomocí vytvořené funkce ***TransformToRealCoordiantes()***.

```
230                     # Prevod pocatecnich a koncovych bodu desek
                        z pixelu do realnych souradnic
231                     start_point1= TransformToRealCoordiantes(board1
                                    ["start_point"], origin, prevod)
232                     start_point2= TransformToRealCoordiantes(board2
                                    ["start_point"], origin, prevod)
233                     end_point1 = TransformToRealCoordiantes(board1
                                    ["end_point"], origin, prevod)
234                     end_point2 = TransformToRealCoordiantes(board2
                                    ["end_point"], origin, prevod)
```

Zdrojový kód 5.35: Převod souřadnic bodů desek z px na mm

Převedené souřadnice a ID desek uložím slovníku **coupleDict**, který obsahuje klíčová slova: *'board1' 'board2'*, kdy každé obsahuje svůj podslovník s klíčovými slovy *'startPoint'*,*'endPoint'* a *'Id'*, za které dosadím patřičné proměnné. Slovník následně uložím do seznamu **couples**. Tento slovník už bude součástí výstupního datového souboru **data.json**.

```
238                     # Ulozeni informaci o dvojici do slovniku
239                     dict = {
240                             "board1": {
241                                 "start_point":start_point1,
242                                 "end_point": end_point1,
243                                 "id": board1["marker_id"],
244                             },
245                             "board2": {
246                                 "start_point": start_point2,
247                                 "end_point":end_point2,
248                                 "id": board2["marker_id"],
249                             }
250                         }
251                         #Ulozeni do seznamu dvojic desek
252                         couples.append(dict)
```

Zdrojový kód 5.36: Uložení převedených souřadnic do slovníku

Stejný postup provedu pro zásobníky s roztokem - u každého nalezeného zásobníku převedu souřadnice markeru pod klíčovým slovem *'center'* z px do mm přímo ve slovníku. Seznam **reservoirs** bude druhou součástí výstupního datového souboru **data.json**.

```
265     # Pokud jsou nalezeny rezervoary
266     if reservoirs != []:
267         for reservoir in reservoirs:
268             # Prevod stredu rezervoaru z pixelu do realnych
                souradnic
269             reservoir["center"] = TransformToRealCoordiantes(
                                    reservoir["center"], origin, prevod)
```

Zdrojový kód 5.37: Převod souřadnic zásobníků s roztokem z px do mm

Seznamy s dvojicemi desek a se zásobníky vložím do posledního slovníku **data**, který obsahuje klíčová slova *'couples'* a *'reservoirs'*. Ke klíčovému slovu *'couples'* přiřadím seznam dvojic **couples** a ke slovu *'reservoirs'* seznam **reservoirs**.

```
272     # Ulozeni infromaci o deskach rezervoarech do slovniku data
273     data = {
274             "couples": couples,
275             "reservoirs": reservoirs
276         }
```

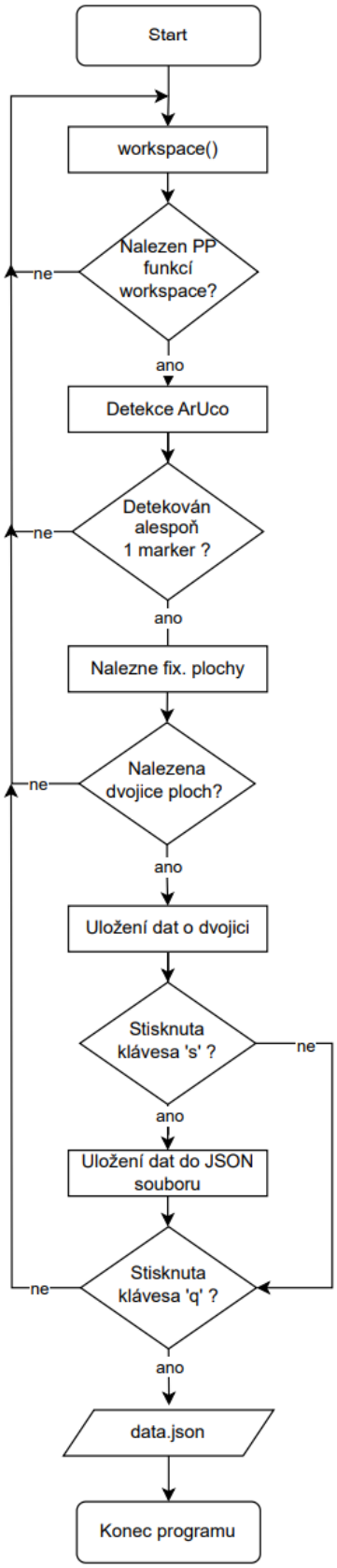
Zdrojový kód 5.38: Uložení seznamů desek do slovníku dat

V tuto chvíli se dostávám na konec algoritmu a mám veškerá potřebná data uložená ve slovníku **,** který už mi jen stačí uložit do souboru v příslušném formátu. Ten pak bude vstupem pro řídící algoritmus stroje. Data jsem se rozhodl ukládat do souboru ve formátu JSON(viz kapitola 4.4.6). Jedná se o vhodnou volbu pro dosažení co největší kompatibility, jelikož se jedná o platformně nezávislý formát, což znamená, že může být použit na různých operačních systémech a s různými programovacími jazyky. Další výhodou je jeho přehlednost pro člověka.

Data budou uloženy v případě stisknutí klávesy "s". Ve chvíli, když je stisknuta klávesa k uložení dat, se zobrazí nové okno, ve kterém se zobrazí snímek, při kterém byla data uložena.

```
290 #Pokud je stisknuta klavesa S
291 if cv2.waitKey(1) & 0xFF == ord('s'):
292     #Ulozi data do json souboru
293 
294     with open('data.json', 'w') as json_file:
295         json.dump(data, json_file, indent=4)
296 #Pokud je stisknuta klavesa S
297 if cv2.waitKey(1) & 0xFF == ord('s'):
298     #Ukonci cyklus
299     break
```

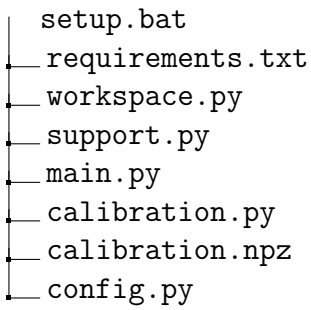
Zdrojový kód 5.39: Ukládání dat a ukončení programu



Obr. 5.16: Vývojový diagram hlavního algoritmu

5.6 Implementace

Cílem této podkapitoly je popsat implementaci systému, zejména jeho ovládání z uživatelského hlediska. Ovládání algoritmu se provádí prostřednictvím příkazového řádku uvnitř repositáře pomocí jednoduchého příkazu: **python názevSouboru.py**. Veškeré níže zmíněné soubory se vyskytují v jednom jediném repositáři(viz obr.5.17).



Obr. 5.17: Repositář

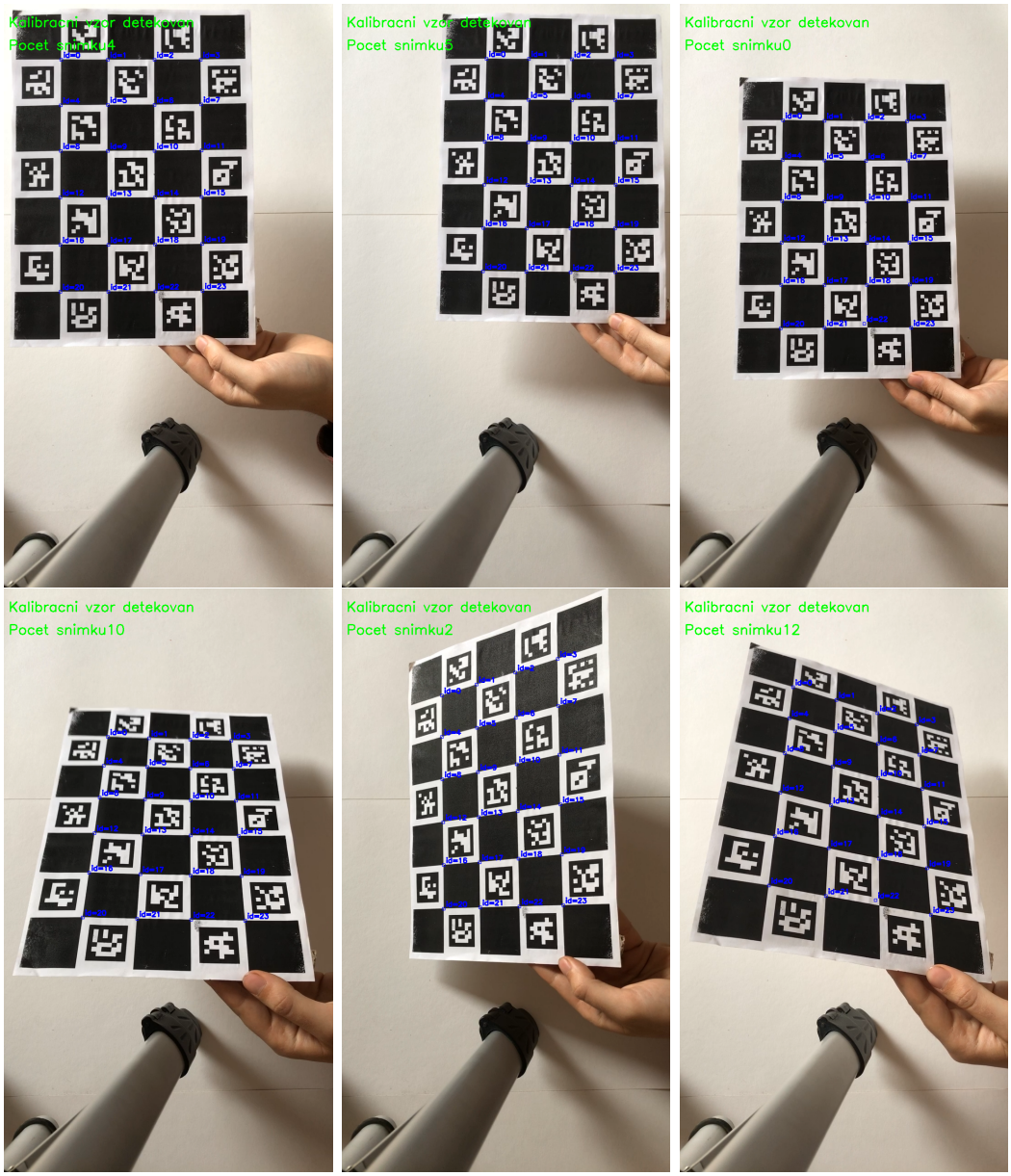
5.6.1 Nastavení

Prvním krokem je instalace potřebných programů a knihoven, na kterých algoritmus funguje. K tomuto účelu byl vytvořen soubor **setup.bat**. Ten po spuštění automaticky stáhne a nainstaluje nejnovější verzi Pythonu a nainstaluje potřebné knihovny, které jsou definované v souboru requirements.txt. Tím jsou v jednom kroku vyřešené nutné knihovny a uživatel se nyní může pusit do prvního kroku - kalibrace.

5.6.2 Kalibrace

Kamera se kalibruje, aby byla zajištěna přesnost a správné měření ve vztahu k fyzickým objektům ve scéně(viz kapitola 2.4). Postup kalibrace je následující: uživatel si při prvním spuštění souboru **calibration.py** vygeneruje kalibrační vzor. Ten se uloží v podobě obrázku do souboru **kalibracniVzor.png**. Tento obrázek následně vytiskne na výšku na papír formátu A4. Doporučuje se jej připnout na pevnější podklad, například na kartonovou desku, aby se zabránilo ohýbání papíru.

Při spuštění souboru **calibration.py** se zobrazí okno se záznamem z kamery s vykreslenými detekovanými rohy kalibračního vzoru. Uživatel polohuje kalibračním vzorem a stiskáváním klávesy '+' pořizuje jednotlivé snímky. Je doporučeno pořídit 15 snímků kalibračního vzoru v různých polohách (viz obrázek 5.2).



Obr. 5.18: Snímaní kalibračního vzoru

Jakmile je pořízen dostatek snímků, záznam kamery lze ukončit stisknutím klávesy 'q'. Tím je práce pro obsluhu hotová a algoritmus už automaticky sám dopočítá potřebná data z pořízených snímků,které pak uloží do příslušného souboru pro další použití, viz výše. Proces kalibrace je nutný provést pouze při prvotním nastavování systému nebo při změně kamery či objektivu.

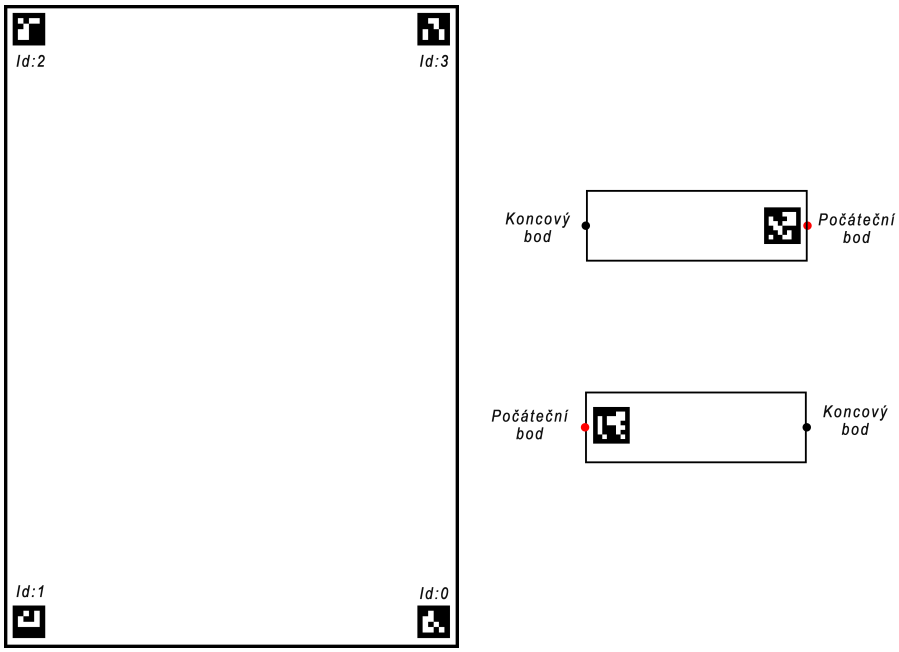
5.6.3 Příprava pracovního prostoru a fixačních ploch

Před detekcí je potřeba připravit pracovní prostor a přípravky k detekci. Prvním krokem je rozmístění 4 ArUco markerů do každého rohu pracovního prostoru ve tvaru obdélníku. Markery musí být ze slovníku přednastaveného pro ohraničení

pracovního prostoru a zároveň ID musí být v seznamu objectsIds v souboru **config.ini**. Dalším krokem je ohraničit fixační plochy tvaru obdélníku výraznou černou linií a dovnitř ohraničení umístit ArUco markery z příslušného slovníku a seznamu, který odpovídá fixačním plochám. Transparentní zásobník s roztokem stačí označit umístěním markeru pod jeho dno. Přednastavené slovníky a seznamy jde s dalšími parametry změnit v souboru **config.ini**.

K zaručení správného fungování algoritmu je nutné dodržet několik podmínek:

- Náklon kamery vůči pracovní ploše by neměl přesáhnout 30°. Algoritmus je schopen efektivně kompenzovat zkreslení způsobené náklonem do této hodnoty.
- ArUco markery k ohraničení pracovního prostoru musí být rozmístěny tak aby zhruba tvořily rohy obdélníku. Zároveň by se měly jít markery postupně podle svých identifikačních čísel od nejmenšího po největší ve směru hodinových ručiček. ID marker k ohraničení musí být v seznamu **workspaceIds** v souboru **config.ini**.
- Detekce zásobníku s roztokem je možná pouze při použití transparentního roztoku.
- Před prvním spuštěním je nutno změřit reálnou velikost strany AruCo markeru k ohraničení pracovního prostoru v milimetrech a zapsat ji do souboru **config.ini**.

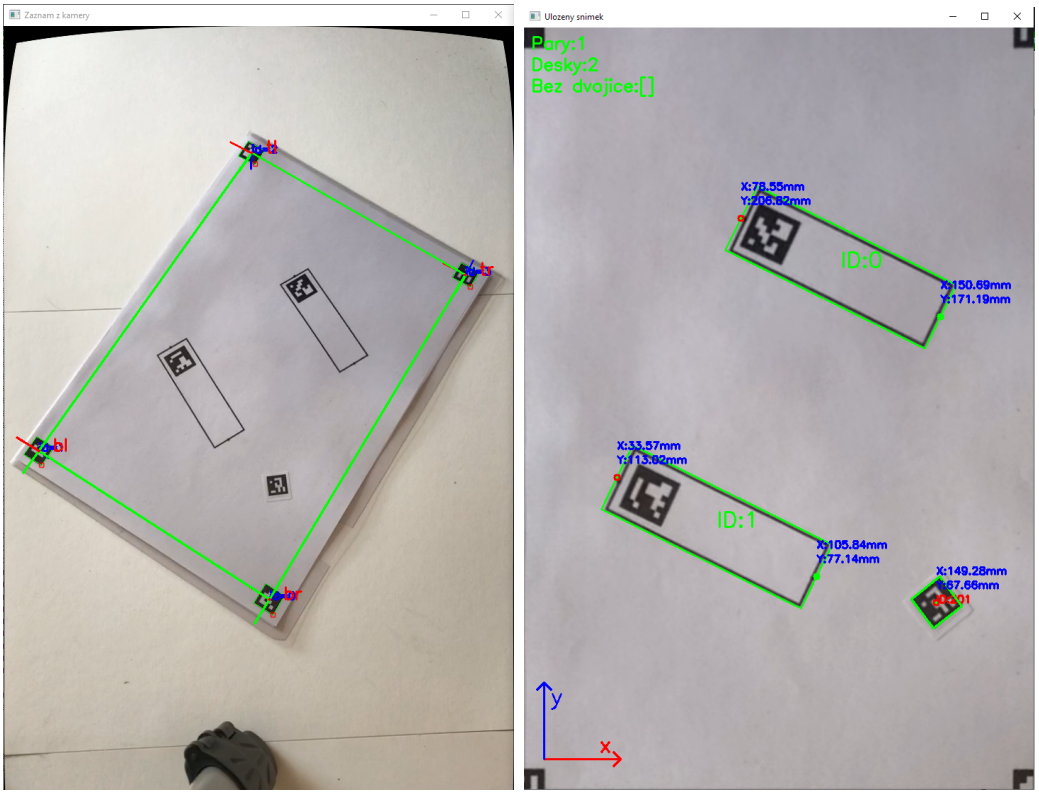


(a) Pracovní prostor (b) Fixační plochy

Obr. 5.19: Příprava před detekcí

5.6.4 Detekce a získání dat

Jakmile uživatel provede proces kalibrace, může spustit samotný detektor pomocí příkazu: **python main.py**. V tu chvíli se otevře 1 okno se záznamem z kamery, na kterém se zároveň bude vykreslovat detekovaný pracovní prostor(viz obr.5.2a). Jakmile je detekován pracovní prostor, otevře se druhé okno. Na němž bude vidět oříznutý obraz uvnitř kterého bude zobrazovat detekované fixační plochy. Jakmile je podle uživatele vše v pořádku detekováno, může získané data uložit pomocí klávesy 's', přičemž se zobrazí nové okno, v němž se zobrazí snímek, při kterém byla data uložena(viz obr.5.2b). Uživatel může v případě špatné detekce při ukládání znovu uložit nová data. Algoritmus lze ukončit pomocí klávesy 'q'. Pokud jsou data uložena, uloží se do souboru **data.json**(viz kapitola 5.5).



(a) Záznam z kamery (b) Uložený snímek

Obr. 5.20: Zobrazená okna

5.7 Shrnutí

V této kapitole byl popsán proces tvorby algoritmů jednotlivých částí systému strojového vidění a jeho následná implementace. Prvním samostatným celkem byl algoritmus ke kalibraci kamery a druhým pak samotný algoritmus k detekci přípravků uvnitř pracovního prostoru. Ten byl rozdělen na 3 části: detekci pracovního prostoru, detekci přípravků a poslední částí bylo zpracování získaných dat.

Hlavním vstupem pro detekční algoritmus je záznam z kamery v reálném čase, který je v těchto 3 zmíněných krocích zpracován. Výstupem je soubor ve formátu JSON, který obsahuje seznam sdružených dvojic fixačních ploch a seznam zásobníků s roztokem. U každé plochy ve dvojici jsou určeny souřadnice počátečního a koncového bodu. Tyto informace umožňují řídícímu algoritmu stroje získat počáteční a koncový bod plochy a určit směr a orientaci kroků pro tažení vlákna. U zásobníku s roztokem je určen pouze střed ArUco markeru, který slouží k označení zásobníku.

Popis tvorby algoritmu se zaměřoval především na kód potřebný pro jeho fungování v oblasti strojového vidění a neposkytoval nejpodrobnější informace o programování samotném. V případě zájmu je doporučeno nahlédnout do přiložených příloh(viz Příloha A) k získání podrobnějšího vhledu do fungování algoritmu jako celku.

Poslední část se zaměřovala na implementaci systému a na popis jeho ovládání z uživatelského hlediska.

```
{
#seznam dvojic desek
"couples": {
    #startovaci deska
    "board1": {
                "start_point":start_point1, #pocatecni bod 1
                "end_point": end_point1,    #koncovy bod 1
                "id": board1["marker_id"],  #id desky 1
                },
    #koncova deska
    "board2": {
                "start_point": start_point2, #pocatecni bod 2
                "end_point":end_point2,      #koncovy bod 2
                "id": board2["marker_id"],   #id desky 2
                }
            }, ...
    #seznam zasobniku s roztokem
    "reservoirs": {
                    "center": [CX, CY], #stred markeru
                    "id": markerId[0]   #id zasobniku
                    }, ...}
```

Zdrojový kód 5.40: Struktura výstupního json souboru.

6 Testování

Cílem této kapitoly je provést testování u stěžejních částí k ověření funkčnosti a zjištění případných nedostatků systému. Proces testování probíhal na stolním počítači s operačním systémem Windows a s následujícími specifikacemi:

- RAM: 8GB, 1200 MHz
- CPU: AMD Vishera FX-8350 [72]
- GPU: NVIDIA GTX 960 2GB VRAM [73]

K testování byla použita kamera Niceboy STREAM s rozlišením 1280 x 720 px.

V prvním kroku byl otestován proces instalace potřebných programů a knihoven prostřednictvím souboru **setup.bat**, který na daném počítači proběhl bezproblémově. Byla nainstalovaná požadovaná verze Pythonu 3.8 a knihovny potřebné k fungovaní systému. K řádnému otestování by však bylo vhodné instalaci otestovat na několika různých zařízeních s různými konfiguracemi a specifikacemi.

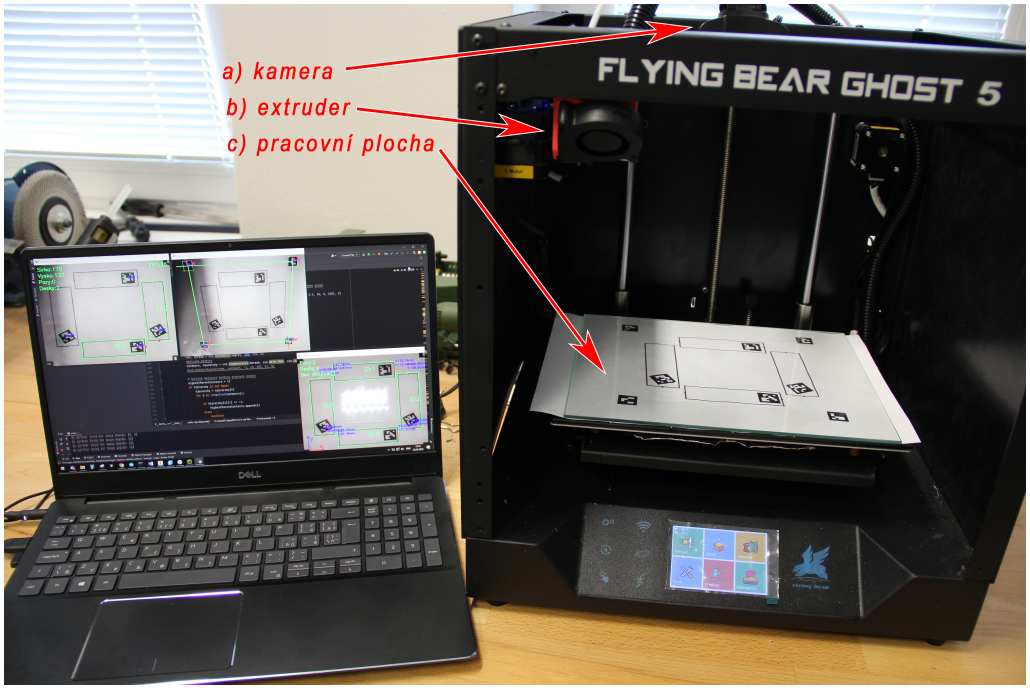
Následovalo připojení kamery prostřednictvím USB a spuštění kalibračního programu pomocí příkazu ***python3 calibration.py***. S využitím vytisknutého kalibračního vzoru a postupu popsaného v předchozí kapitole (viz podkapitola 5.6.2) byl proveden proces získávání kalibračních dat. Při získávání dat byla zjištěna nedokonalá funkčnost při stiskávání kláves, kdy v některých momentech nedocházelo k uložení dat nebo nešlo proces kalibrace ukončit pouhým stisknutím klávesy. Problém byl pravděpodobně v nedostatečném výkonu použitého počítače, jelikož systém dosahoval rychlosti při kalibraci jen 12 FPS. Pomocí získaných kalibračních dat byl kalibrován získaný obraz.

Po úspěšné kalibraci obrazu byl spuštěn hlavní modul detekce pomocí příkazu ***python3 main.py***. Pro účely testování bylo připraveno několik různých rozložení pracovních prostorů. K zjištění přesnosti měření systému byly vytvořené pracovní prostory podrobeny měření, kdy byly systémem naměřené souřadnice bodů porovnány se souřadnicemi naměřenými manuálně. Měření bylo provedeno pro 3 různá rozložení pracovního prostoru. U každého rozložení bylo měření provedeno u 5ti různých bodů a ze získaných dat byla spočítána průměrná odchylka X-ové a Y-ové souřadnice. Zjištěná průměrná odchylka pro směr X je +0.5mm a ve směru Y -0.5mm. Pro získání směrodatnějších hodnot by však bylo vhodné měření provést vícekrát. V příloze D je možné nalézt získané snímky ze systému při jednotlivých měřeních.

[mm]	1. rozložení				2. rozložení				3. rozložení			
	real		cv		real		cv		real		cv	
	x	y	x	y	x	y	x	y	x	y	x	y
bod 1	80	206	78,4	207,94	42	190	41,25	189,76	46,5	174	46,65	176,9
bod 2	151	171	150,3	172,1	145	223	143,61	220,31	48	96	48,59	95,25
bod 3	34	111	33,6	113,7	123	100	122,46	100,03	153	98	152,13	97,32
bod 4	106	76	105,5	77,75	171	85	171,4	83,79	151	177	150,32	178,96
bod 5	151	67	149,1	68,3	103	37	101,97	37	142	196	143,45	196,32
dílčí průměrná odchylka [mm]	X		Y		X		Y		X		Y	
	1,02		-1,758		0,662		0,822		-0,128		-0,75	
celková průměrná odchylka [mm]	X						Y					
	0,518						-0,562					

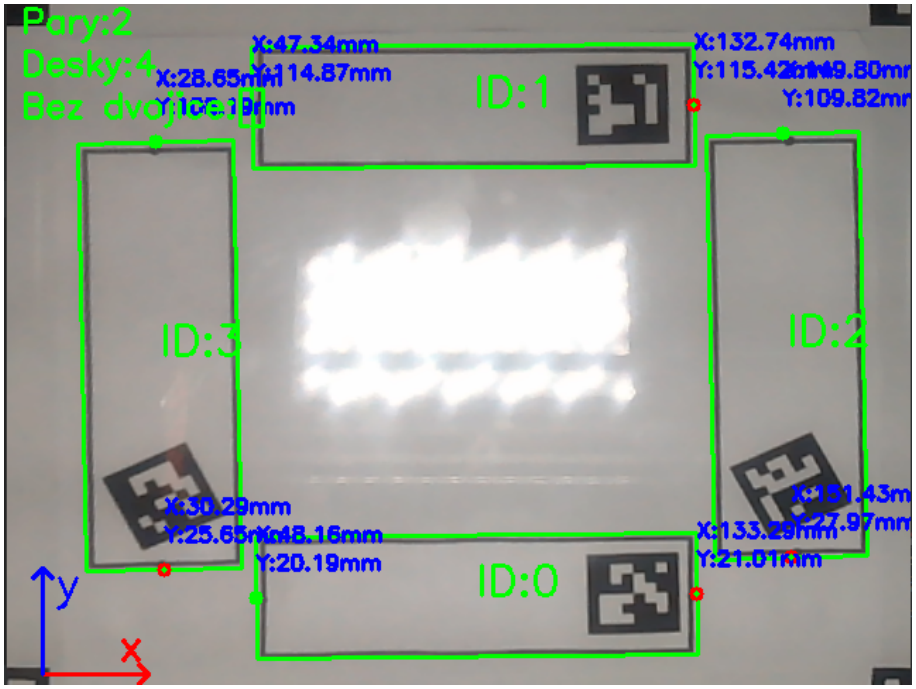
Tabulka 6.1: Naměřené souřadnice při testování

V poslední části byla otestována přímá implementace systému strojového vidění na stroj produkující vlákna metodou tažení. Šlo o upravenou 3D tiskárnu, jejíž řídící systém byl vytvořen v rámci bakalářské práce dalšího studenta Fakulty strojní Technické univerzity v Liberci: Davida Astaloše.



Obr. 6.1: Upravená 3D tiskárna při testování

V rámci testování byl vytyčen pracovní prostor a rozmístěny fixační plochy(viz podkapitola 5.6.3), které byly umístěny pod transparentní skleněnou desku(viz obr.6.1c). Následně byla otestována samotná detekce, kdy došlo k úspěšné detekci fixačních ploch (viz obr. 6.2). Musela však být upravena základní hodnota prahové hodnoty pro detekci kontur(viz obr.5.11), pravděpodobně kvůli nekonzistentní světelnosti uvnitř stroje.



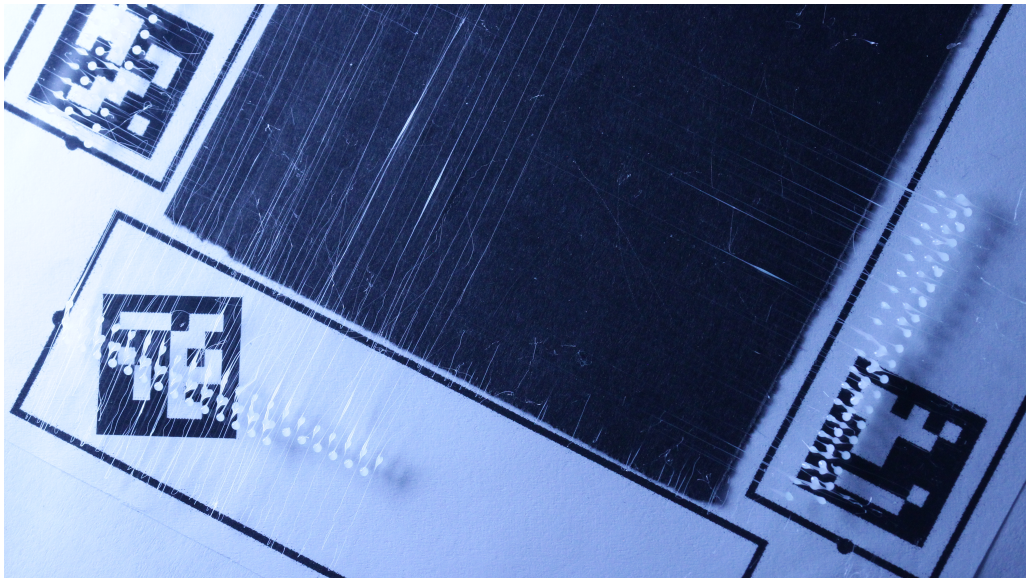
Obr. 6.2: Detekované fixační plochy při testování


```
{
"couples": [
    {
        "board1": {
            "start_point": [
                133.29,
                21.01
            ],
            "end_point": [
                48.16,
                20.19
            ],
            "id": 0
        },
        "board2": {
            "start_point": [
                132.74,
                115.42
            ],
            "end_point": [
                47.34,
                114.87
            ],
            "id": 1
        }
    },
    {
        "board1": {
            "start_point": [
                151.43,
                27.97
            ],
            "end_point": [
                149.8,
                109.82
            ],
            "id": 2
        },
        "board2": {
            "start_point": [
                30.29,
                25.65
            ],
            "end_point": [
                28.65,
                108.19
            ],
            "id": 3
        }
    }
]
}
```


Obr. 6.3: Získaný JSON soubor při testování

Při úspěšné detekci byla úspěšně uložena data do souboru JSON(viz obr.6.3), který obsahoval vstupní data pro upravený řídící systém 3D tiskárny.

Nakonec byla pomocí získaných dat provedena zkušební produkce vláken metodou tažení z taveniny. Vyprodukovaná vlákna je možno vidět na následujícím obrázku, nebo v příloze B.



Obr. 6.4: Vyprodukovaná vlákna

Během testování bylo zjištěno několik důležitých poznatků:

- Zjištěná průměrná odchylka měření je ve směru směru X: +0.5mm a ve směru Y: -0.5mm, získané hodnoty by však bylo vhodné ověřit rozsáhlejším měřením, případně zjistit vlivy ovlivňující přesnost.
- Při implementaci bylo zjištěno, že hraje významnou roli barva podkladu pod fixačními plochami. V případě bílého podkladu probíhala detekce bezproblémově, v případě použití původního černého podkladu byla detekce fixačních ploch, ale i pracovního prostoru velmi nekonzistentní.
- Velkou roli při detekci hrálo rovněž nekonzistentní osvětlení a výskyt stínů ve snímaném pracovním prostoru, vlivem netransparentního zakrytování stroje. Problém by se dal eliminovat použitím transparentního zakrytování nebo použitím několika světelných zdrojů na různých místech, např. led pásek.
- Vytvořený systém strojového vidění je za určitých předpokladů schopen detekovat přípravky v pracovním prostoru stroje, zjistit jejich souřadnice a předat tato data řídícímu systému stroje skrze JSON soubor k následné produkci syntetických vláken metodou tažení.

7 Shrnutí a diskuze

V rámci řešeného problému bylo cílem vytvořit systém, který je co nejjednodušší a nejrobustnější, a zároveň klade minimální požadavky na obsluhu. Robustnosti se snažilo docílit pomocí využití ArUco markerů k označování přípravků. Uživatel tak má téměř neomezené množství možností, kolik přípravků použít a jak je rozmístit. Nutností je však práce při ohraničování pracovního prostoru pomocí ArUco markerů, protože je nutno zajistit pravoúhlost, které spolu svírají spojnice markerů. Rovněž je nutno přesně umístit marker, který určuje počátek souřadnicového systému stroje. Tato práce je však potřeba udělat jen jednou při první implementaci systému.

Jelikož v tuto chvíli nebylo možné nalézt informace o řídících systémech strojů k produkci nanovláken metodou tažení, je struktura výstupních dat pouze orientační. Je pravděpodobné, že by řídící systém stroje nebyl schopen takové parametry přijmout, a proto by musela být upravena buď struktura výstupních dat, nebo řídící systém stroje, k zajištění přímé kompatibility. Výstupní data však poskytují dostatečné množství informací k proveditelnosti procesu.

Systém strojového vidění vytvořený v této práci by se dal považovat za první prototypovou verzi a možností, jak jej vylepšit je nespočet, zde jsou však ty nejdůležitější z nich:

- Prvním a zároveň nejdůležitějším krokem by mělo být vylepšení části detekce ohraničujících linii kolem fixačních ploch. Tato oblast představuje potenciální zdroj nejčastějších chybných detekcí.
- Dalším vylepšením by mohla být detekce nechtěných předmětů uvnitř pracovního prostoru a varovat obsluhu při jejich výskytu, nebo detekce nevhodného vzájemného postavení dvojice fixačních ploch.
- K rozšíření kampatibility by data mohla být ukládána do jiných formátu podle potřeby, např YAML[74], XML[75].
- Vhodným vylepšením by bylo vytvoření grafického rozhraní (GUI), čímž by došlo ke sjednocení všech funkčních částí do jednoho celku a napomohlo by k implementaci navržených vylepšení. Ovládání by tak bylo mnohem intuitivnější pro koncového uživatele.
- Program je nyní možné používat pouze s operačním systémem Windows. Nicméně, lze snadno rozšířit možnosti vytvořením instalačních souborů pro systémy jako Linux nebo OSX, neboť Python je podporován oběma těmito systémy.

- Systém by bylo vhodné podrobit dlouhodobějšímu testování k odhalení potenciálních chyb, které mohou vzniknout při původně nezamýšleném nakládání se systémem.

Posledním důležitým faktem, který je nutno zmínit je, že samostatné ArUco markery sloužící v tomto případě k označování a detekci zásobníků s roztokem by teoreticky mohly být použity k označování libovolných předmětů v pracovním prostoru. Část systému by tak mohla být použita i pro úplně jiný případ použití než pro produkci vláken. Například by se pomocí ArUco markerů daly označovat předměty, které mají být uchopeny pomocí manipulátoru, apod(viz příloha C). Tento fakt naznačuje, že se nejedná o ryze jednoúčelový systém a lze jej do jisté míry využít i v jiných aplikacích. Zároveň se tak otevírá možnost dalšího rozvoje systému směrem k jinému, možná víceúčelovému řešení.

8 Závěr

Tato bakalářská práce se zabývala návrhem, realizací a testováním systému strojového vidění s cílem zjednodušit přípravu procesu produkce syntetických vláken metodou tažení. Hlavním cílem bylo vytvořit systém schopný detekovat přípravky v pracovním prostoru stroje, získat jejich souřadnice a následně tyto informace předat dál řídícímu systému stroje.

Nejprve byla provedena studie v oblasti technologie počítačového vidění s následnou rešerší trhu s dostupnými systémy strojového vidění pro průmyslové využití.

Na základě provedené studie byl zhotoven návrh a následná realizace systému strojového vidění, skládající se ze dvou samostatných algoritmů. První vedlejší algoritmus slouží ke kalibraci kamery pomocí kalibračního vzoru ChArUco. Druhý hlavní algoritmus slouží k detekci a zjištění souřadnic přípravků: fixačních ploch a zásobníku s roztokem, v pracovním prostoru stroje. Detekce přípravků byla realizována pomocí detekce kontur a ArUco markerů, které byly popsány ve studii a implementovány pomocí knihovny OpenCV. Získané souřadnice jsou ukládány do souboru ve formátu JSON, čímž je zajištěna kompatibilita s jinými softwary. Výstupní soubor ze systému je vstupním souborem pro řídící algoritmus stroje k produkci syntetických vláken metodou tažení.

Nakonec byl systém otestován s cílem ověřit jeho funkčnost, přičemž bylo zjištěno, že je schopný detekovat přípravky a získávat jejich souřadnice v pracovním prostoru stroje.

Vytvořený systém tak plní cíle, které byly stanoveny, nabízí další možnosti, jak jej zdokonalit a mohl by přispět ke zvýšení míry automatizace při procesu produkce syntetických vláken metodou tažení. Kompletní systém je dostupný skrze platformu GitHub, kde je zároveň dostupný úplný zdrojový kód.

Bibliografie

[1] John Billingsley. *Practical Machine Vision for Industry*. Elsevier, 2015.

[2] M. Zouhair Atassi. *Micro and Nanostructured Fibers*. William Andrew, 2013.

[3] Gary R. Bradski and Adrian Kaehler. *Learning OpenCV*. 1st ed. Sebastopol: O'Reilly, c2008. ISBN: 978-0-596-51613-0.

[4] Thomas Smits and Melvin Wevers. "The visual digital turn: Using neural networks to study historical images". In: *Digital Scholarship in the Humanities* 35 (Jan. 2018). DOI: 10.1093/llc/fqy085.

[5] Carsten Steger, Markus Ulrich, and Christian Wiedemann. *Machine Vision Algorithms and Applications*. 2nd ed. Weinheim: John Wiley & Sons, 2018. ISBN: 978-3-527-41365-2.

[6] Cognex. *MACHINE VISION AND IIOT DRIVE NEXT WAVE OF LOGISTICS EFFICIENCY*. 2021. URL: https://www.cognex.com/en-cz/blogs/industrial-barcode-reader/leverage-the-power-of-iiot-in-logistics-using-machine-vision (visited on 01/16/2023).

[7] *Contour Detection using OpenCV (Python/C++)*. URL: https://learnopencv.com/contour-detection-using-opencv-python-c (visited on 02/26/2023).

[8] *cv.findContours()*. URL: https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#ga17ed9f5d79ae97bd4c7cf18403e1689a (visited on 02/28/2023).

[9] *cv.drawContours()*. URL: https://docs.opencv.org/3.4/d6/d6e/group__imgproc__draw.html#ga746c0625f1781f1ffc9056259103edbc (visited on 02/28/2023).

[10] *Extract an object on a sheet of paper*. URL: https://stackoverflow.com/questions/44806723/extract-an-object-on-a-sheet-of-paper (visited on 02/26/2023).

[11] *Background Subtraction*. URL: https://www.sciencedirect.com/topics/engineering/background-subtraction (visited on 02/26/2023).

[12] *A framework for background modelling and shadow suppression for moving object detection in complex wavelet domain*. URL: https://www.researchgate.net/figure/Moving-object-detection-using-background-subtraction-in-complex-wavelet-domain-a_fig2_257627533 (visited on 02/21/2023).

[13] *The OpenCV Library*. 2022. URL: https://opencv.org (visited on 01/16/2023).

[14] *cv.cornerHarris()*. URL: https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html#gac1fc3598018010880e370e2f709b4345 (visited on 02/28/2023).

[15] David Lowe. "Distinctive Image Features from Scale-Invariant Keypoints". In: *International Journal of Computer Vision* 60 (Nov. 2004), pp. 91–. DOI: 10.1023/B:VISI.0000029664.99615.94.

[16] Herbert Bay, Tinne Tuytelaars, and Luc Van Gool. "SURF: Speeded up robust features". In: vol. 3951. July 2006, pp. 404–417. ISBN: 978-3-540-33832-1. DOI: 10.1007/11744023_32.

[17] Yenewondim Biadgie and Kyung-Ah Sohn. "Feature Detector Using Adaptive Accelerated Segment Test". In: vol. 33. May 2014, pp. 1–4. ISBN: 978-1-4799-4441-5. DOI: 10.1109/ICISA.2014.6847403.

[18] Michael Calonder et al. "BRIEF: Binary Robust Independent Elementary Features". In: vol. 6314. Sept. 2010, pp. 778–792. ISBN: 978-3-642-15560-4. DOI: 10.1007/978-3-642-15561-1_56.

[19] Ethan Rublee et al. "ORB: an efficient alternative to SIFT or SURF". In: Nov. 2011, pp. 2564–2571. DOI: 10.1109/ICCV.2011.6126544.

[20] *cv::ORB Class Reference*. URL: https://docs.opencv.org/4.x/db/d95/classcv_1_1ORB.html (visited on 02/28/2023).

[21] *An introduction to deep learning*. URL: https://developer.ibm.com/articles/an-introduction-to-deep-learning/ (visited on 02/05/2023).

[22] Aurélien Géron. *Hands-On Machine Learning with Scikit-Learn, Keras, and TensorFlow: Concepts, Tools, and Techniques to Build Intelligent Systems*. O'Reilly Media, 2019.

[23] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. MIT Press, 2016.

[24] Francois Chollet. *Deep Learning with Python*. Manning Publications, 2018.

[25] *A guide to Two-stage Object Detection: R-CNN, FPN, Mask R-CNN*. URL: https://medium.com/codex/a-guide-to-two-stage-object-detection-r-cnn-fpn-mask-r-cnn-and-more-54c2e168438c (visited on 02/28/2023).

[26] Jasper Uijlings et al. "Selective Search for Object Recognition". In: *International Journal of Computer Vision* 104 (Sept. 2013), pp. 154–171. DOI: 10.1007/s11263-013-0620-5.

[27] Ross Girshick et al. *Rich feature hierarchies for accurate object detection and semantic segmentation*. 2013. DOI: 10.48550/ARXIV.1311.2524. URL: https://arxiv.org/abs/1311.2524.

[28] Shaoqing Ren et al. *Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks*. 2015. DOI: 10.48550/ARXIV.1506.01497. URL: https://arxiv.org/abs/1506.01497.

[29] Joseph Redmon et al. *You Only Look Once: Unified, Real-Time Object Detection*. 2015. DOI: 10.48550/ARXIV.1506.02640. URL: https://arxiv.org/abs/1506.02640.

[30] *Performance Comparison of YOLO Object Detection Models*. URL: https://learnopencv.com/performance-comparison-of-yolo-models/ (visited on 02/28/2023).

[31] Wei Liu et al. "SSD: Single Shot MultiBox Detector". In: vol. 9905. Oct. 2016, pp. 21–37. ISBN: 978-3-319-46447-3. DOI: 10.1007/978-3-319-46448-0_2.

[32] *Detection of ArUco Markers*. URL: https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html (visited on 02/28/2023).

[33] *PredefinedDictionaryType*. URL: https://docs.opencv.org/4.x/de/d67/group__objdetect__aruco.html#ga4e13135a118f497c6172311d601ce00d (visited on 01/16/2023).

[34] *What Is Camera Calibration?* URL: https://www.mathworks.com/help/vision/ug/camera-calibration.html (visited on 02/05/2023).

[35] *CORRECTION OF RADIAL DISTORTION IN DIGITAL IMAGES*. URL: https://www.researchgate.net/publication/255610429_CORRECTION_OF_RADIAL_DISTORTION_IN_DIGITAL_IMAGES (visited on 02/01/2023).

[36] *Understanding Lens Distortion*. URL: https://learnopencv.com/understanding-lens-distortion/ (visited on 02/01/2023).

[37] *Distortion*. URL: https://www.edmundoptics.eu/knowledge-center/application-notes/imaging/distortion/ (visited on 02/01/2023).

[38] *Detection of ChArUco Boards*. URL: https://docs.opencv.org/3.4/df/d4a/tutorial_charuco_detection.html (visited on 02/28/2023).

[39] Ben Rogers, Sumita Pennathur, and Jesse Adams. *Nanotechnology: Understanding Small Systems*. CRC Press, 2016.

[40] Lisa Regan. *Nanotechnology: The Future is Tiny*. ReferencePoint Press, 2018.

[41] Lukáš Stanislav. "PRODUKCE NANOVLÁKEN METODOU TAŽENÍ". Disertační práce. Liberec: Technická univerzita v Liberci, Fakulta Strojní, 2015.

[42] Andrii Shynkarenko. "Vývoj zařízení pro automatickou výrobu nanovlákenných přízí". Disertační práce. Liberec: Technická univerzita v Liberci, Fakulta Strojní, 2021.

[43] *Global Computer Vision Market*. URL: https://www.sphericalinsights.com/reports/computer-vision-market (visited on 02/28/2023).

[44] *Cover all types of vision with iRVision*. URL: https://www.fanuc.eu/~/media/files/pdf/products/robots/brochures/mbr-02856-ro%5C%20irvision%5C%20brochure-v6/irvision%5C%20brochure%5C%20en.pdf?la=en (visited on 01/16/2023).

[45] *KUKA.VisionTech*. URL: https://www.kuka.com/en-de/products/robot-systems/software/application-software/kuka_visiontech (visited on 01/16/2023).

[46] *Integrated Vision*. URL: https://new.abb.com/products/robotics/application-equipment-and-accessories/vision-systems/integrated-vision (visited on 01/16/2023).

[47] *Cognex Vision Library*. URL: https://www.cognex.com/en-cz/products/machine-vision/vision-software/cognex-vision-library (visited on 01/16/2023).

[48] *Cognex Designer Software*. URL: https://www.cognex.com/en-cz/products/machine-vision/vision-software/cognex-designer-software (visited on 01/16/2023).

[49] *Cognex VisionPro Software*. URL: https://www.cognex.com/en-cz/products/machine-vision/vision-software/visionpro-software (visited on 01/16/2023).

[50] *Cognex Vision Guided Robotics*. URL: https://www.cognex.com/en-cz/industries/automation/robotic-system-integrators (visited on 01/16/2023).

[51] *Keyence Product Selection by Industry and Application*. URL: https://www.keyence.eu/solutions/applications/ (visited on 01/16/2023).

[52] *Stack Overflow Developer Survey 2022*. URL: https://survey.stackoverflow.co/2022/#most-popular-technologies-language (visited on 05/07/2023).

[53] *JetBrains*. URL: https://www.jetbrains.com/company/ (visited on 05/16/2023).

[54] OpenCV. *cv2.aruco.CharucoBoard()*. https://docs.opencv.org/3.4/db/df8/classcv_1_1aruco_1_1CharucoBoard.html. 2021.

[55] OpenCV. *cv2.detectMarkers()*. https://docs.opencv.org/3.4/d9/d6d/tutorial_table_of_content_aruco.html. 2021.

[56] OpenCV. *cv2.interpolateCornersCharuco()*. https://docs.opencv.org/3.4/d9/d6d/tutorial_table_of_content_aruco.html. 2021.

[57] OpenCV. *cv2.imshow()*. 2021. URL: https://docs.opencv.org/3.4/d7/dfc/group__highgui.html#ga453d42fe4cb60e5723281a89973ee563.

[58] OpenCV. *calibrateCameraCharuco*. https://docs.opencv.org/3.4/d9/d6d/tutorial_table_of_content_aruco.html. 2021.

[59] NumPy. *numpy.savez()*. https://numpy.org/doc/stable/reference/generated/numpy.savez.html. 2021.

[60] OpenCV. *cv2.estimatePoseSingleMarkers()*. https://docs.opencv.org/3.4/d9/d6d/tutorial_table_of_content_aruco.html. 2021.

[61] NumPy. *numpy.linalg.norm()*. https://numpy.org/doc/stable/reference/generated/numpy.linalg.norm.html. 2021.

[62] OpenCV. *cv2.warpPerspective()*. 2021. URL: https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html#ga0203d9ee5fcd28d40dbc4a1ea4451983.

[63] OpenCV. *cv2.getPerspectiveTransform()*. 2021. URL: https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html#ga8c4d5c3e5f5f5b2596b8af42b254d8a8.

[64] OpenCV. *cv2.VideoCapture()*. 2021. URL: https://docs.opencv.org/3.4/d8/dfe/classcv_1_1VideoCapture.html.

[65] OpenCV. *cv2.imread()*. 2021. URL: https://docs.opencv.org/3.4/d4/da8/group__imgcodecs.html#gga288b8b3da0892bd651fce07b3bbd3a56a107e6bcaf625d786c5912af4100ebd53.

[66] OpenCV. *cv2.undistort()*. 2021. URL: https://docs.opencv.org/3.4/da/d54/group__imgproc__transform.html#ga69f2545a8b62a6b0fc2ee060dc30559d.

[67] OpenCV. *cv2.cvtColor()*. 2021. URL: https://docs.opencv.org/3.4/d7/d1b/group__imgproc__misc.html#ga397ae87e1288a81d2363b61574eb8cab.

[68] OpenCV. *cv2.threshold()*. 2021. URL: https://docs.opencv.org/3.4/d7/d4d/tutorial_py_thresholding.html.

[69] OpenCV. *cv2.findContours()*. 2021. URL: https://docs.opencv.org/3.4/d9/d8b/tutorial_py_contours_hierarchy.html.

[70] OpenCV. *Contour Features*. 2021. URL: https://docs.opencv.org/3.4/dd/d49/tutorial_py_contour_features.html.

[71] OpenCV. *cv2.boxPoints()*. 2021. URL: https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#gaf78d467e024b4d7936cf9397185d2f5c.

[72] AMD. *FX-8350*. URL: https://www.amd.com/en/product/1451 (visited on 05/20/2023).

[73] GPUZoo. *NVIDIA GeForce GTX 960 GPU specifications*. URL: https://www.gpuzoo.com/GPU-NVIDIA/GeForce_GTX_960.html (visited on 05/20/2023).

[74] YAML Language Development Team. *Introduction to YAML*. 2021. URL: https://yaml.org/spec/1.2.2/#chapter-1-introduction-to-yaml (visited on 05/16/2023).

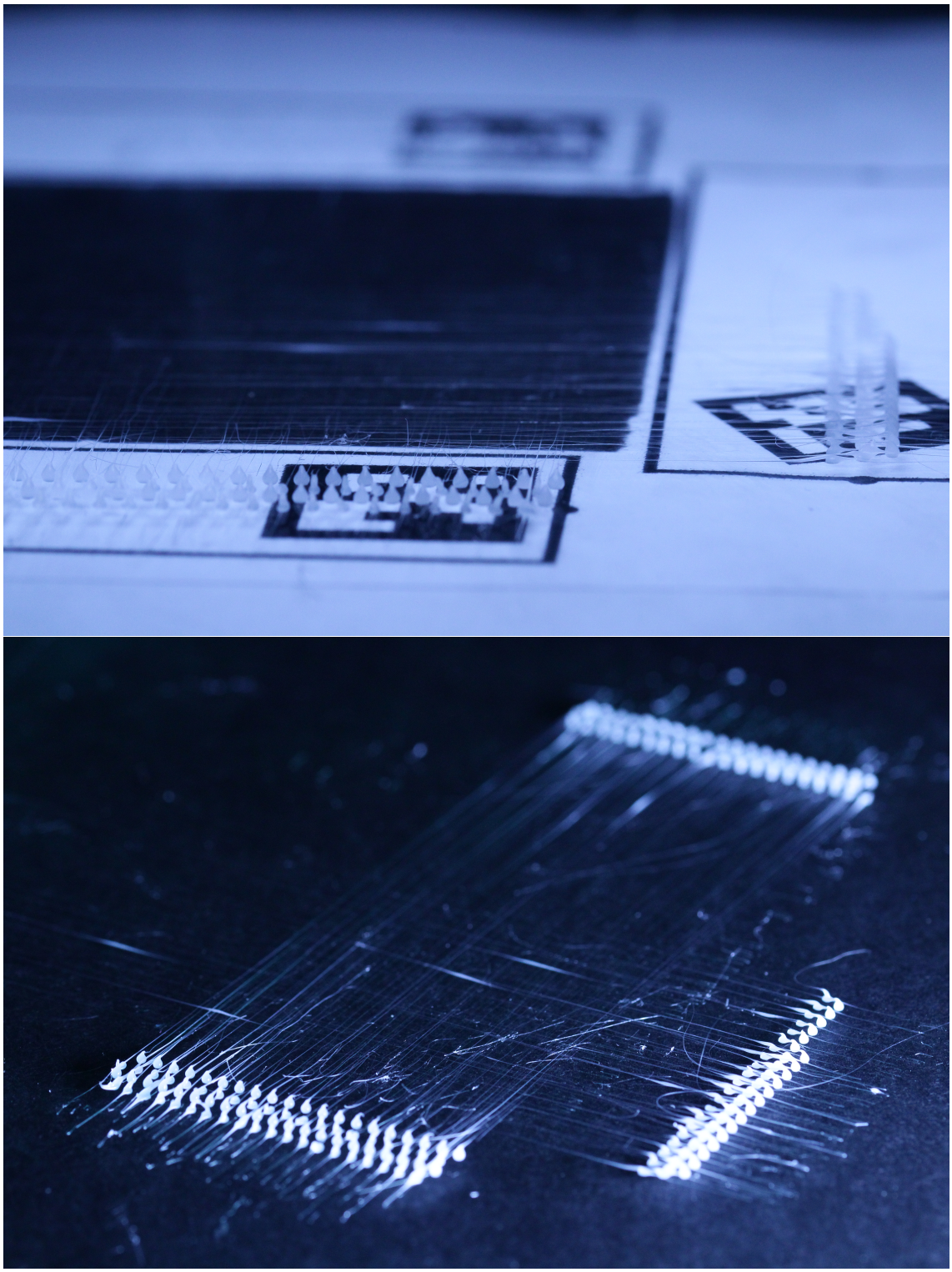
[75] *XML ESSENTIALS*. URL: https://www.w3.org/standards/xml/core (visited on 05/16/2023).

Seznam příloh

Příloha A

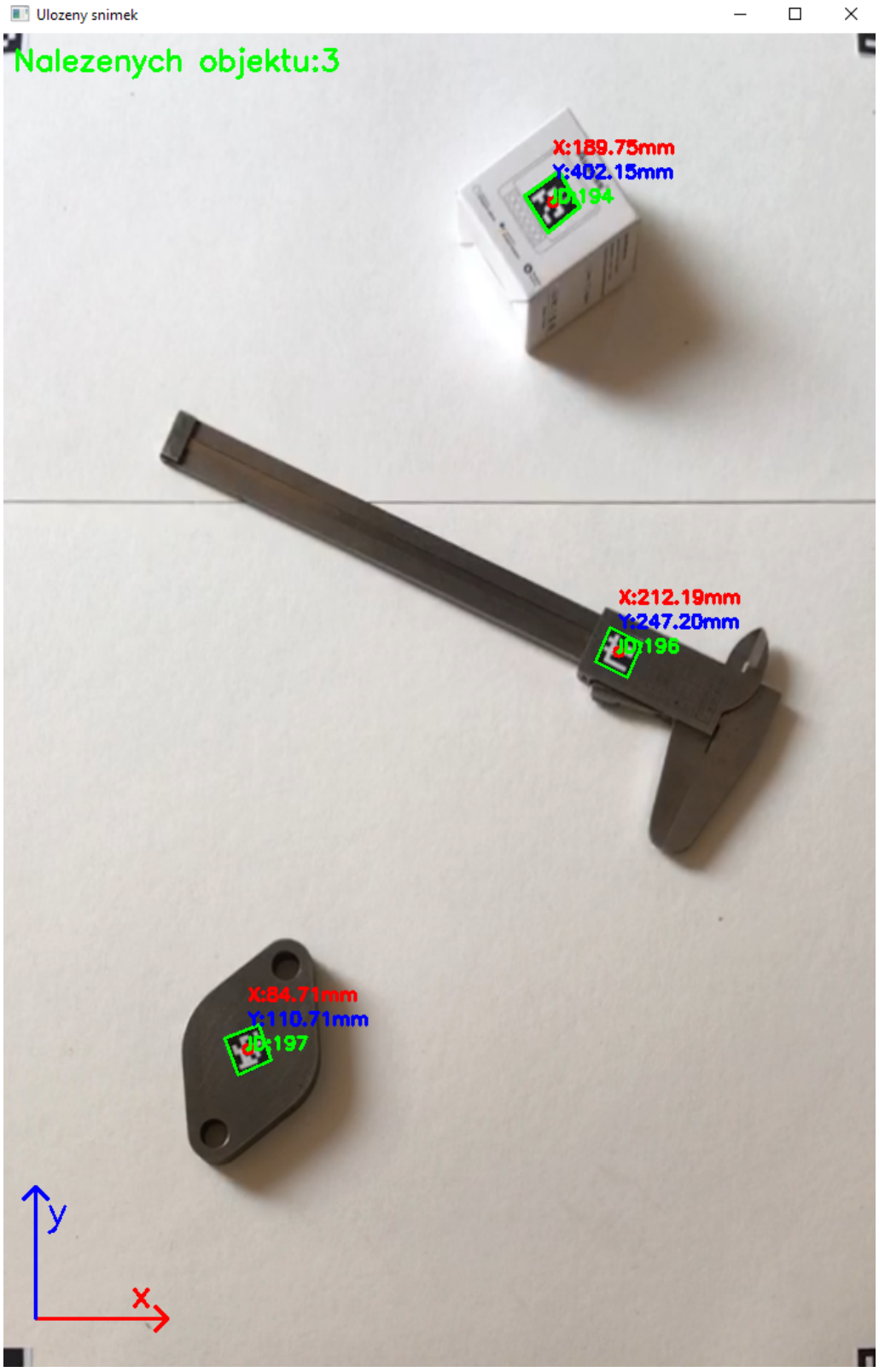
Zdrojový kód systému strojového vidění je dostupný skrze platformu GitHub:
https://github.com/kubeex/Machine-vision-system-for-nano-microfiber-drawing

Příloha B



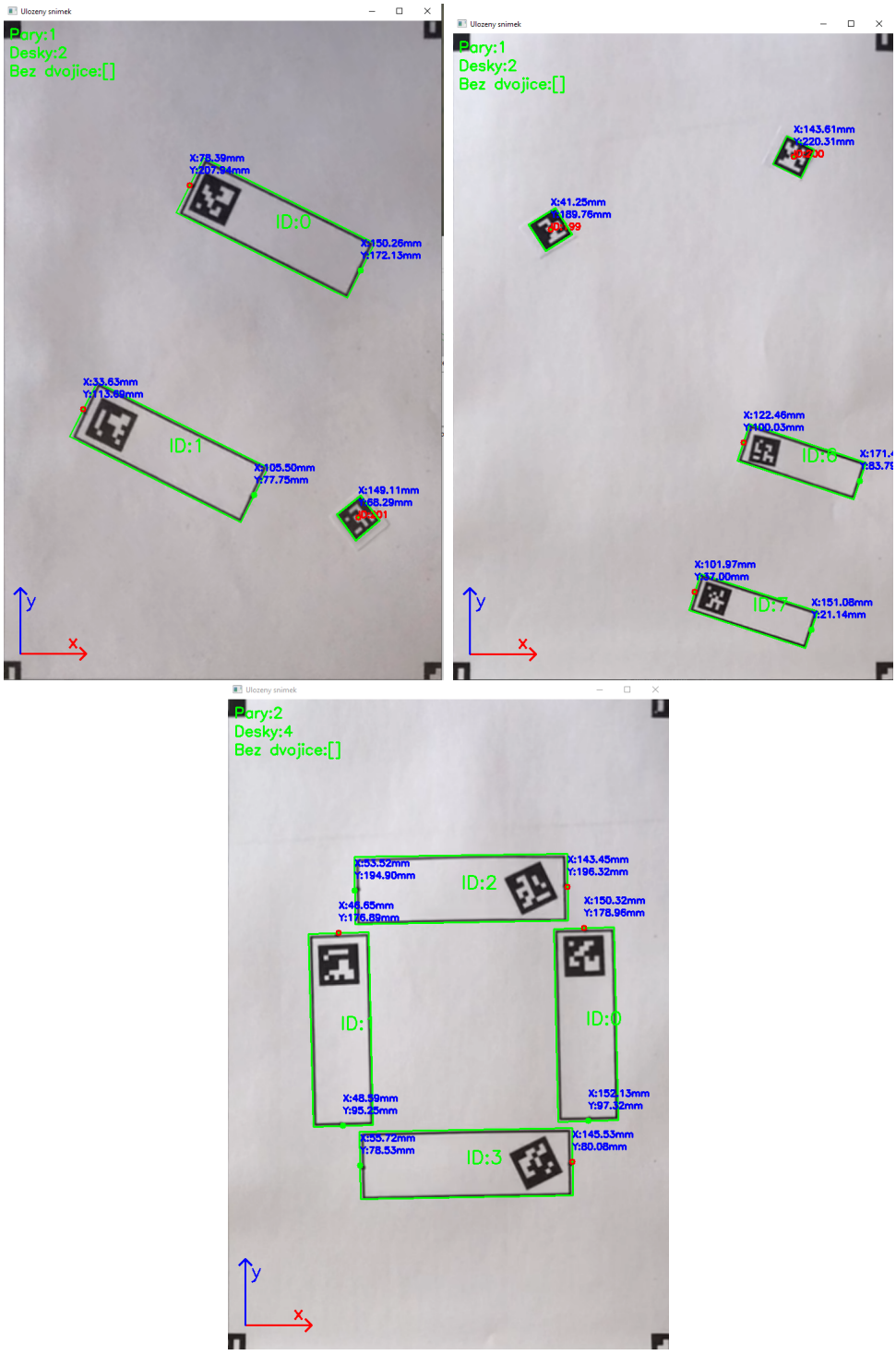
Příloha B: Vyprodukovaná vlákna

Příloha C



Příloha C: Alternativní použití systému

Příloha D



Příloha D: Snímky z testovacího měření