VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
FACULTY OF INFORMATION TECHNOLOGY

ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

DETEKTOR OBJEKTŮ PRO ROBOTICKÉ PRACOVIŠTĚ

OBJECT DETECTOR FOR ROBOTIC WORKPLACE

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE **MARTIN KNESLÍK**
AUTHOR

VEDOUCÍ PRÁCE **Ing. ZDENĚK MATERNA, Ph.D.**
SUPERVISOR

BRNO 2023

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
FAKULTA INFORMAČNÍCH TECHNOLOGIÍ

Zadání bakalářské práce

146827

Ústav: Ústav počítačové grafiky a multimédií (UPGM)
Student: **Kneslík Martin**
Program: Informační technologie
Specializace: Informační technologie

Název: **Detektor objektů pro robotické pracoviště**
Kategorie: Zpracování obrazu
Akademický rok: 2022/23

Zadání:

1. Seznamte se s algoritmy používanými pro detekci a sledování objektů na základě RGBD dat.
2. Seznamte se se systémem pro vizuální programování robotů ARCOR2.
3. Proveďte návrh modulu do systému ARCOR2 umožňující detekci barevných kostiček pomocí zařízení Kinect Azure.
4. Navržené řešení implementujte.
5. Proveďte testování.
6. Vytvořte video prezentující vaši práci, její cíle a výsledky.

Literatura:

- Zhou, Qian-Yi, Jaesik Park, and Vladlen Koltun. "Open3D: A modern library for 3D data processing." *arXiv preprint arXiv:1801.09847* (2018).
- Wang, Yangfan, et al. "Recent Advances in 3D Object Detection based on RGB-D: A Survey." *Displays* (2021): 102077.
- Ward, Isaac Ronald, Hamid Laga, and Mohammed Bennamoun. "RGB-D image-based Object Detection: From traditional methods to deep learning techniques." *RGB-D Image Analysis and Processing*. Springer, Cham, 2019. 169-201.

Při obhajobě semestrální části projektu je požadováno:

- Body 1 až 3.

Podrobné závazné pokyny pro vypracování práce viz https://www.fit.vut.cz/study/theses/

Vedoucí práce: **Materna Zdeněk, Ing., Ph.D.**
Vedoucí ústavu: Černocký Jan, prof. Dr. Ing.
Datum zadání: 1.11.2022
Termín pro odevzdání: 17.5.2023
Datum schválení: 31.10.2022

Abstrakt

Hlavním cílem této práce bylo vytvoření algoritmu umožňujícího detekci a sledování barevných kostiček pomocí hloubkové kamery Kinect Azure a integrování tohoto algoritmu do systému ARCOR2. Řešení používá filtrování vstupního obrazu podle barev, algoritmus DBSCAN pro hledání shluků v mračně bodů a RANSAC pro detekci rovin. Detekce je vyhodnocena na vlastním datasetu, přičemž byla dosažena přesnost 91%. Uživatel systému ARCOR2 může výsledky této práce používat na demonstračním pracovišti v robotické laboratoři, kde budou roboti manipulovat s barevnými kostičkami, které algoritmus detekuje.

Abstract

The main goal of this thesis was to create an algorithm for detection and tracking of colored cubes using the Kinect Azure depth camera and integrate this algorithm into the ARCOR2 system. The solution uses color filtering of the input image, the DBSCAN algorithm for finding clusters in the pointcloud, and RANSAC for plane detection. The detection is evaluated on a custom dataset with an accuracy of 91%. The user of the ARCOR2 system can use the results of this work in a demonstration workstation in the robotics lab, where robots will manipulate the colored cubes detected by the algorithm.

Klíčová slova

detekce objektů, trackování, ARCOR2, Azure Kinect, mračno bodů, python, Open3D

Keywords

object detection, tracking, ARCOR2, Azure Kinect, pointcloud, python, Open3D

Citace

KNESLÍK, Martin. *Detektor objektů pro robotické pracoviště.* Brno, 2023. Bakalářská práce. Vysoké učení technické v Brně, Fakulta informačních technologií. Vedoucí práce Ing. Zdeněk Materna, Ph.D.

Detektor objektů pro robotické pracoviště

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Zdeňka Materny, Ph.D. Uvedl jsem všechny literární prameny, publikace a další zdroje, ze kterých jsem čerpal.

. . . .
Martin Kneslík
16. května 2023

Poděkování

Tímto bych rád vyjádřil své poděkování panu Ing. Zdeňku Maternovi, Ph.D. za jeho ochotu mi poskytnout veškerou potřebnou pomoc při řešení této práce. Dále bych chtěl poděkovat své rodině za podporu při studiu.

Obsah

1 Úvod **3**

2 Problematika **4**
2.1 RGB-D 4
2.1.1 Hloubkové kamery 4
2.1.2 Mračno bodů 6
2.2 Existující řešení 9
2.3 ARCOR2 10
2.3.1 Architektura 10
2.3.2 AREditor 12
2.3.3 Robotické pracoviště 14

3 Návrh řešení **15**

4 Implementace **18**
4.1 Detektor 18
4.1.1 Vizualizér 19
4.1.2 Filtrování obrazu podle barev 20
4.1.3 Rozdělení mračna bodů do shluků 21
4.1.4 Segmentace rovin 22
4.1.5 Výpočet úhlů mezi rovinami 23
4.1.6 Hledání kandidátů 23
4.1.7 Klasifikace kandidátů 24
4.1.8 Problémy při implementaci 25
4.2 Tracker 26
4.3 Object Type 29

5 Testování **30**
5.1 Integrační a unit testy 30
5.2 Dataset a jeho vyhodnocení 30
5.3 Experimenty na robotickém pracovišti 35

6 Závěr **39**

Literatura **40**

A Obsah paměťového média **42**

Seznam obrázků

2.1 RGB a hloubkový obraz 4
2.2 Hloubková kamera Azure Kinect [8] 5
2.3 Mračno bodů získané z RGBD dat na obrázku 2.1 6
2.4 Ukázka DBSCAN algoritmu. Převzato z [3] 7
2.5 Iterativní nalezení rovin 8
2.6 Rozdělení metod podle autorů [11] 9
2.7 Program v aplikaci AREditor 10
2.8 Architektura systému ARCOR2. 11
2.9 Scény v aplikaci AREditor 12
2.10 Vytvoření projektu založeného na scéně v aplikaci AREditor 13
2.11 Větvení programu podle pravdivostní hodnoty 14
2.12 Robotické pracoviště v univerzitní laboratoři O104 14

3.1 Viditelné strany kostek 16
3.2 Příklad komunikace modulů 17

4.1 Aplikace s uživatelským rozhraním 19
4.2 Mračna bodů rozdělená podle barev 20
4.3 Všechny shluky 21
4.4 Jednotlivé shluky 21
4.5 Nalezené roviny (vizualizovány odlišnými barvami) 22
4.6 Kostky se dvěma a třemi stranami 23
4.7 Všechny detekované kostky 24
4.8 Detekované kostky 25
4.9 Deformace kostek 26
4.10 Příklad ukládání kostek 27

5.1 Ukázka snímků z datasetu (pouze RGB obrázky) 31
5.2 Vyhodnocení datasetu 33
5.3 Problémy na různých snímcích 34
5.4 Průměrný čas detekce 34
5.5 Procentuální trvání jednotlivých kroků 35
5.6 Porovnání pozice vytvořeného bodu a pozice kostky získaná přes API 36
5.7 Kostka se nachází blízko bodu, akce vrací `true` 36
5.8 Kostka se nenachází blízko bodu, akce vrací `false` 37
5.9 Akce vrací pozici kostky 37
5.10 Program třídící kostky podle barvy 38

Kapitola 1

Úvod

Detekce objektů pomocí technik počítačového vidění má mnoho reálných využití – od autonomního řízení vozidel až po průmyslovou automatizaci a robotiku. Oproti klasické detekci na 2D obrazu nebo videu se detekce objektů na hloubkových datech liší v tom, že data obsahují informace o vzdálenosti každého pixelu ke kameře. Lze tak třeba rozlišit tvar objektu nebo jiné vlastnosti, které není možné z normálního 2D obrazu vyčíst.

Cílem této práce je navrhnout algoritmus umožňující detekci barevných kostek a implementovat ho jako samostatný modul do systému ARCOR2[1]. Tento systém pracuje s akčními body v rozšířené realitě. Akční body mají nastavenou pozici a orientaci a přiřazují se k nim jednotlivé akce, které ovládají různá zařízení (např. akce zvedni/polož pro robota). Momentálně lze tyto body vytvářet buďto manuálně nastavením pozice bodu pomocí rozšířené reality a nebo vytvořením bodu na místě, kde se zrovna nachází robotické rameno. Tato práce umožní pozice získat pomocí detekce kostek z kamery a nebude tak nutné vytvářet akční body na přesném místě, kde se nachází kostky. Dále bude možné ověřovat, zda se v okolí akčního bodu vůbec nějaká kostka nachází, což umožní rozvětvování výsledného programu.

Ze začátku je v kapitole 2 představena problematika s důrazem na vysvětlení informací o RGB-D datech, hloubkových kamerách a jejich principech. Dále jsou zde představeny existující řešení rozdělené na tradiční metody a metody hlubokého učení. Je zde představen systém ARCOR2 a jeho architektura. Kapitola 3 se věnuje návrhu modulu. Zde je vysvětleno rozdělení na dva samostatné moduly pro detekci a sledování kostek. Dále je zde představen princip, podle kterého se budou detekovat kostky. V kapitole 4 se probírají implementační detaily a vysvětlují různé překážky, na které se během vývoje narazilo. Na závěr se v kapitole 5 probírá testování, vytvoření a vyhodnocení vlastního datasetu a různé experimenty prováděné na robotickém pracovišti.

[1] https://github.com/robofit/arcor2

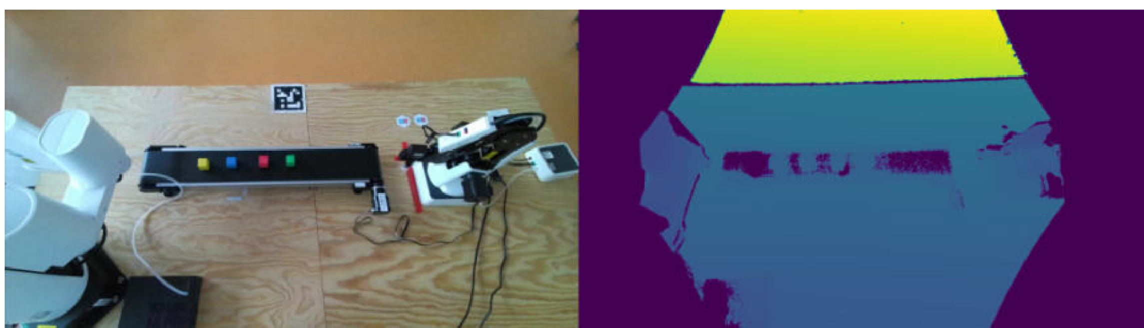
Kapitola 2

Problematika

Detekce objektů má v dnešní době mnoho různých využití. S rostoucí dostupností různých hloubkových kamer a 3D skenerů se poslední dobou rozšiřuje i detekce v RGB-D datech [10]. Oproti klasické detekci na 2D obrazu je zde rozdíl v tom, že se musí nyní prohledávat i hloubková data. Nevýhodou však můžou být neúplná nebo řídká data, limitace senzoru a hlavně požadavky na výpočetní čas. Mnoho aplikací požaduje výpočet v reálném čase, ale algoritmy založené na RGB-D detekci bývají kvůli přidání prostorové dimenze pomalejší, než detekce na 2D obrazu [11]. Další překážkou jsou pak různé deformace způsobené nedokonalým měřením hloubky.

2.1 RGB-D

RGB-D (Red, Green, Blue – Depth) představuje datový formát, kterou tvoří dvojice klasického RGB obrazu a hloubkových dat. Hloubková data obsahují informace o vzdálenosti/hloubce každého pixelu. Tyto data jsou po úpravě vzájemně zarovnané s RGB obrazem, takže pro každý pixel je známá jeho barva a hloubka.



Obrázek 2.1: RGB a hloubkový obraz

2.1.1 Hloubkové kamery

Hloubková data lze získat různými způsoby. Mezi nejrozšířenější způsoby patří hloubkové kamery. Ty v dnešní době nachází čím dál tím větší uplatnění jak už v robotice, tak i rozšířené realitě nebo lékárnictví [11, 8]. Hloubkové kamery se dále rozlišují podle metod, kterými zachycují a zpracovávají informace o hloubce.

Hloubkové kamery používají pro detekci hloubky především tři různé metody [1]:

- **Stereo senzory** – získávání hloubkových informacích je založeno na princiou stereo vidění, které napodobuje lidské vidění. Stereo senzory využívají dvě kamery s určitým odstupem. Snímky z kamer jsou následně používány k extrakci vizuálních rysů a pro získání mapy disparity mezi pohledy kamer. Pomocí disparity jsme pak schopni získat hloubkovou mapu.
- **Time-of-Flight (TOF)** – princip této metody spočívá v tom, že kamera vysílá pulzy světla a hloubku vypočítá pomocí doby, za kterou se odražený signál vrátí zpět ke snímači kamery. Nevýhoda této metody je citlivost na různé různé typy povrchů, tmavé nebo odrazivé povrchy tak nelze spolehlivě snímat a na hloubkovém obrazu jsou tak jako neplatná data. Výhodou je však odolnost vůči slabému osvětlení.
- **Structured light** – hloubková mapa se získává pomocí deformace vzorů vysílaných senzorem. Senzor promítá různé tvary (například kruhy nebo mřížky), které jsou následně odražené od objektu zpátky do kamery. Ta následně vzory zachytí a analyzuje deformace a posunutí vzorů.

Azure Kinect

Data v této práci jsou získávány z hloubkové kamery Azure Kinect [8]. Tato kamera od společnosti Microsoft obsahuje 12megapixelovou RGB kameru a 1 megapixelový hloubkový senzor. Pro zachycení hloubky používá již zmíněnou metodu Time-of-Flight.

Obrázek 2.2: Hloubková kamera Azure Kinect [8]

2.1.2 Mračno bodů

Mračno bodů (pointcloud) je množina bodů v prostoru, kdy každý bod má pozici v soustavě kartézských souřadnic a případně i odpovídající barvu. Tento formát dat jde získat například převedením z RGB-D obrazu. K vytvoření mračna bodů je potřeba znát vnitřní (intrinsické) parametry kamery, kterou byl RGB-D obraz zachycen. Tyto parametry obsahují rozlišení, ohniskovou vzdálenost, clonu, zorné pole.

Obrázek 2.3: Mračno bodů získané z RGBD dat na obrázku 2.1

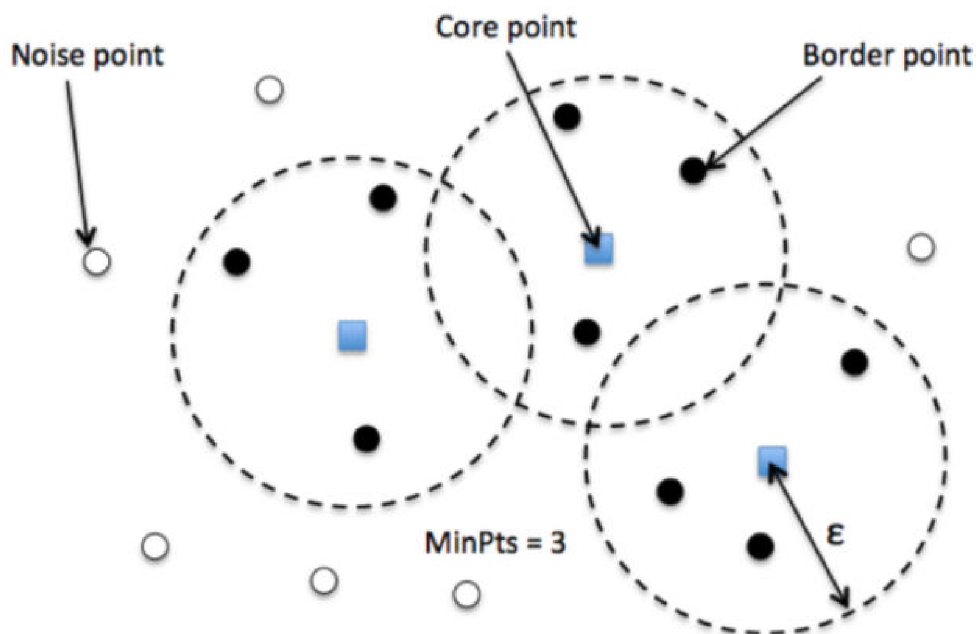
Pro mračna bodů existuje několik knihoven, které s nimi dokáží pracovat. Jedna z nich je open-source knihovna Open3D [12] pro Python a C++, která umožňuje snadnou práci a manipulaci s 3D daty. Poskytuje nástroje pro načítání, zpracování a vizualizaci dat. Dále implementuje důležité algoritmy pro zpracování mračen bodů jako je například algoritmus RANSAC pro detekci rovin nebo algoritmus DBSCAN pro rozdělení mračna na shluky. Nevýhodou mračen bodů může být jejich velikost a náročnost na výpočetní výkon. Data se tedy musí většinou před dalším zpracováním napřed upravit. Open3D poskytuje několik metod, díky kterým lze vstupní data převzorkovat na menší nebo na uniformní rozlišení.

Rozdělení na shluky pomocí algoritmu DBSCAN

DBSCAN (Density-based spatial clustering of applications with noise) [4] je shlukovací algoritmus, který seskupuje body v oblasti s vysokou hustotou do shluků. Shluky jsou mezi sebou oddělené právě místy s nízkou hustotou bodů.

Pro nalezení shluků využívá dva klíčové parametry, epsilon a MinPts. Epsilon určuje radius okolí, které je prozkoumáváno kolem každého bodu. V tomto okolí musí být nalezen minimální počet bodů (MinPts), aby se oblast považovala za hustou.

Podle těchto parametrů se rozdělují body na tři typy. Jestliže má bod v okolí minimální počet bodů, tak se jedná o jádrový bod (core point). Z tohoto bodu algoritmus hledá v okolí epsilon další jádrové body, které pak označí za součást shluku. Dalším typem bodů jsou hraniční body. To jsou takové body, které mají ve svém okolí alespoň jeden jádrový bod. Pokud není bod jádrový ani hraniční, tak se jedná o šum. Tyto body tedy nepatří do žádného shluku.



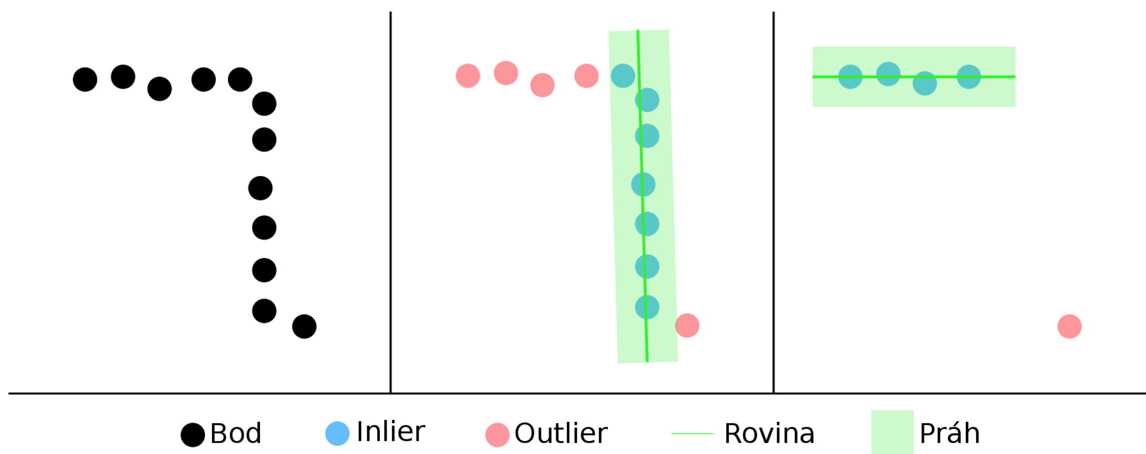
Obrázek 2.4: Ukázka DBSCAN algoritmu. Převzato z [3]

Segmentace rovin pomocí algoritmu RANSAC

RANSAC (Random sample consensus) [5] je iterativní algoritmus pro odstraňování šumu a nalezení nejlepšího matematického modelu z množiny dat, která může obsahovat odlehlé body nebo šum.

Algoritmus pracuje tak, že se náhodně vybere určený počet bodů, ze kterých se vypočítá matematický model roviny. Po určení roviny se zkontrolují všechny body tak, že se vypočítá vzdálenost od bodu k rovině. Pokud je tato vzdálenost menší než určená prahová hodnota, tak se bod považuje za inlier, tedy bod, který je blízko modelu a můžeme ho považovat jako součást roviny. V opačném případě se bod považuje za outlier. Tyto kroky se v cyklu opakují dokud se nedosáhne předem daného počtu iterací. Výsledná rovina se pak vybera podle toho, který model má nejvíce inlierů.

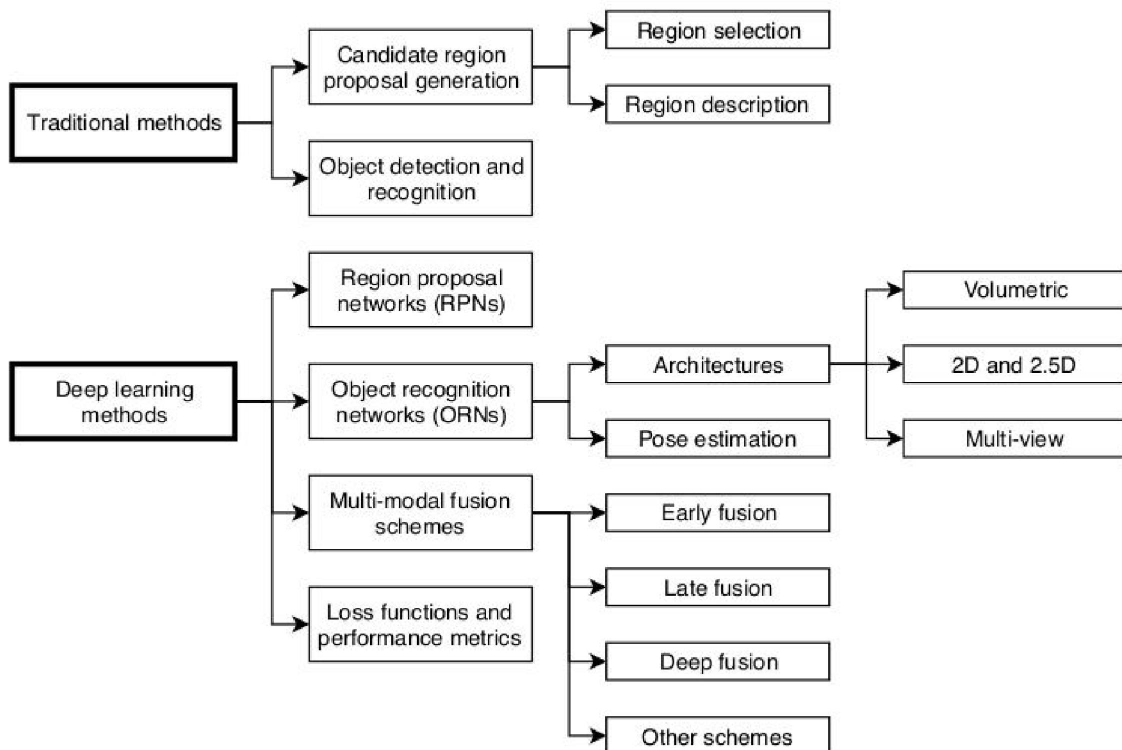
Jestliže je potřeba najít všechny roviny v mračnu bodů, tak se musí celý tento proces několikrát opakovat, RANSAC totiž nalezne pouze jednu rovinu. Pokud se bude volat RANSAC vícekrát a vždy se mu jako vstup nastaví outlier získaný z předešlého volání, tak se získají právě všechny roviny. Podle počtu inlierů pak lze roviny i vyfiltrovat. Díky tomu se třeba může odstranit rovina, která má velký počet bodů nebo naopak příliš malý počet bodů.



Obrázek 2.5: Iterativní nalezení rovin

2.2 Existující řešení

Stávající řešení pro detekci objektů v hloubkových datech se dají rozdělit do dvou hlavních kategorií [11]. První kategorii tvoří takzvané klasické metody, které pomocí strojového učení a ručně vytvořených funkcí detekují polohu a orientaci objektů. Tato metoda pracuje ve dvou krocích. V prvním kroku se vytvoří sada kandidátních oblastí, které mohou obsahovat hledaný objekt. Z těchto oblastí se pak ve druhém kroku natrénuje klasifikátor, který kandidáty klasifikuje na pravdivé nebo neplatné detekce. Druhou kategorii pak představují metody hloubkového učení. Ty nahrazují některé kroky tradičních metod sítěmi hlubokého učení.



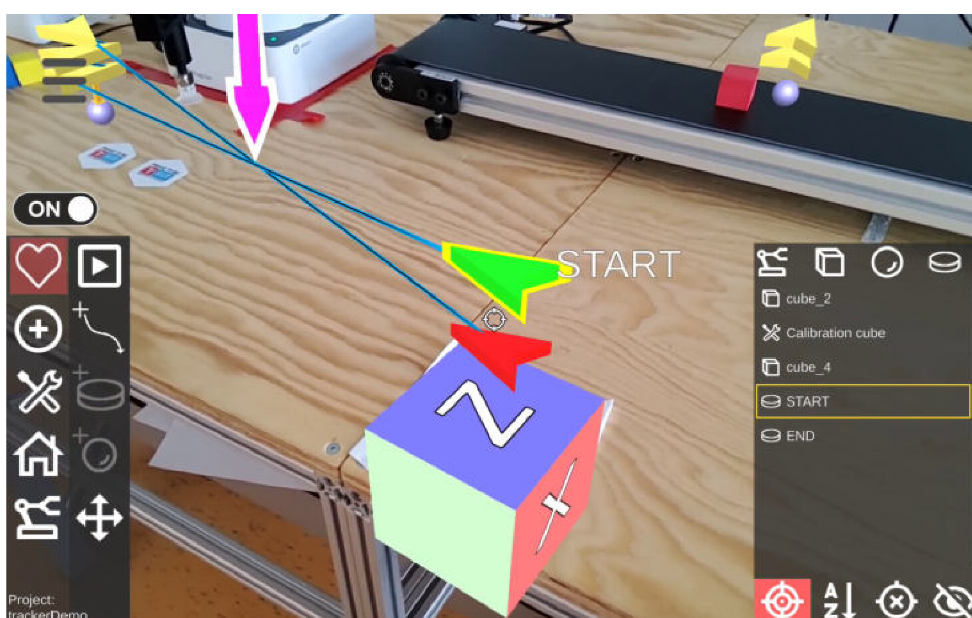
Obrázek 2.6: Rozdělení metod podle autorů [11]

Kromě strojového nebo hloubkové učení lze detekci také provádět analyzováním geometrických vlastností objektů v hloubkových datech. Podobným problémem, který se řeší v této práci, se zabýval Kunt [7], který v hloubkových datech detekoval cihly. Řešení rozdělil do tří částí, kdy napřed nalezl normálový vektor podložky, poté segmentoval obraz na jednotlivé cihly pro které následně nalezl jejich ohraničující boxy. Pro každou část řešení navíc navrhl více různých postupů. Hlavním krokem bylo odstranit rovinu reprezentující zem. Normálový vektor roviny představující zem odhadoval třemi způsoby. Prvně vyzkoušel přístup založený na výšce, pak vyzkoušel detekovat normálový vektor pomocí algoritmu RANSAC a na konec prozkoumal přístup založený na změně výšky. Cihly pak detekoval pomocí nejmenšího ohraničujícího obdélníku a opět pomocí algoritmu RANSAC. Pro testování těchto metod pak si pak anotoval pozice jednotlivých cihel ve snímcích, na kterých zkoušel kombinace všech jeho metod.

2.3 ARCOR2

ARCOR2 (Augmented Reality Collaborative Robot) [6] je systém pro zjednodušené programování kolaborativních robotů v rozšířené realitě vyvíjený výzkumnou skupinou Robo@FIT na fakultě informačních technologií Vysokého učení technického v Brně.

Díky tomuto systému lze programovat různé zařízení v rpzšířené realitě, například průmyslové roboty nebo dopravníkový pás. Systém je modulární a lze tak vytvářet a přidávat podporu pro nové zařízení. Programování probíhá především přes aplikaci AREditor[1] na mobilním zařízení, které pomocí kamery zobrazuje uživateli scénu s rozšířenou realitou. Uživatel vytváří body ve scéně, ke kterým poté přidává specifické akce, které se mají vykonávat. Tyto akce pak uživatel propojí a vytvoří tím přesné kroky, které za sebou bude program vykonávat. Systém také umožňuje spolupráci více uživatelů na scéně naráz.



Obrázek 2.7: Program v aplikaci AREditor

2.3.1 Architektura

Systém se dělí na dvě hlavní části – nezávislé služby tvořící backendovou část a AREditor (frontendová část). Tato kapitola přebírá informace z GitHub wiki těchto projektů[2].

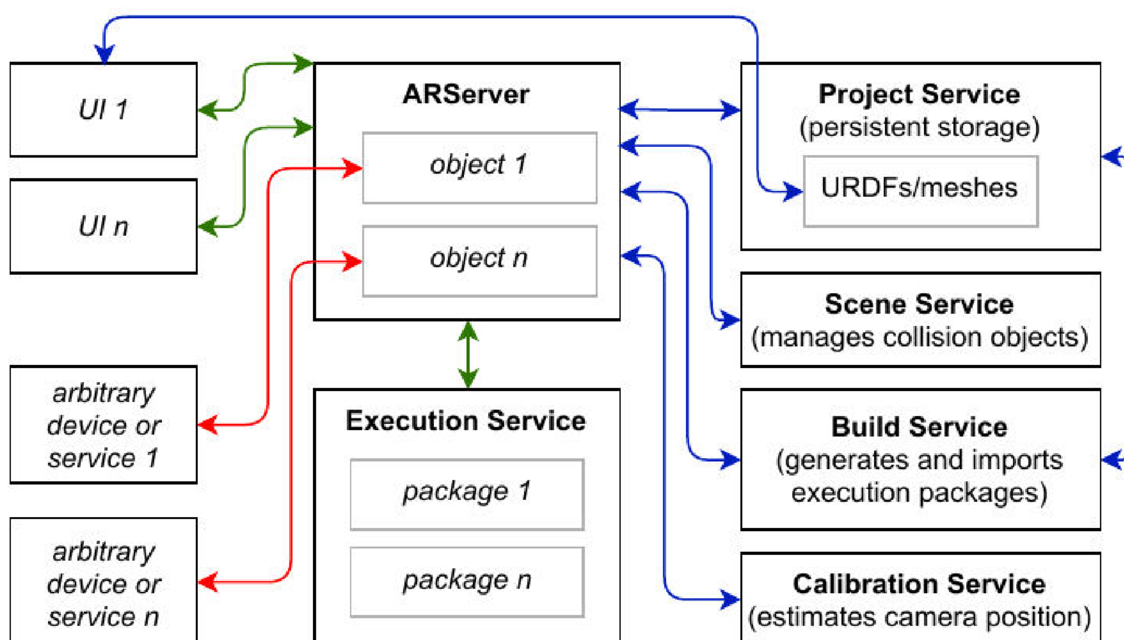
Nejdůležitější součástí systému je služba ARServer, která je zodpovědná za veškerou komunikaci mezi uživatelským rozhraním a ostatními službami. S uživatelským rozhraním komunikuje pomocí WebSocket API rozhraní, které umožňuje oboustrannou komunikaci. S ostatními službami ARServer komunikuje pomocí REST API rozhraní. Na obrázku 2.8 je tato komunikace zobrazena pomocí modrých čar, zatímco WebSocket rozhraní je zobrazena zeleně. Mezi další důležité služby systému patří:

- služba **Project** – ukládá potřebná data pro pracoviště, například scény, projekty, modely a různé objekty,

[1] https://github.com/robofit/arcor2_areditor

[2] https://github.com/robofit/arcor2/wiki, https://github.com/robofit/arcor2_areditor/wiki

- služba **Scene** – je zodpovědná za správu kolizních objektů,
- služba **Build** – vytváří samostatné spustitelné balíčky pro daný projekt,
- služba **Execution** – stará se o balíčky vytvořené Build službou,
- služba **Calibraction** – umožňuje odhadnout polohu kamery a robota pomocí kamery na základě ArUco značek.



Obrázek 2.8: Architektura systému ARCOR2.

Další služby pak přímo implementují konkrétní zařízení. Takové služby jsou do systému integrované pomocí takzvaného Object Type (dále jen OT). OT slouží jako plugin, který poskytuje novou funkcionalitu a jde dynamicky načítat. Integrace nového zařízení může být provedena buďto pouze pomocí samostatného OT, nebo pomocí OT s vlastní API službou. Samostatný OT má nevýhodu v tom, že nemůže mít žádnou závislost třetí strany. Tento způsob je dostačující pouze pokud již pro zařízení existuje nějaký druh API se všemi potřebnými funkcemi. Vytvoření nové API služby zároveň s novým OT se tedy používá při integrování složitějších zařízení, které potřebují dodatečné závislosti.

OT může s ostatními službami komunikovat buďto pomocí WebSocket API nebo pomocí REST API rozhraní (na obr. 2.8 zobrazeno červenou šipkou). Pro REST API rozhraní je v systému implementován klient, který podporuje typové anotace a datové třídy. Při vývoji nového OT se musí dodržovat předem dané postupy. OT musí používat pouze standardní python knihovny, funkce ze základního modulu `arcor2` a samostatné API služby. Každý OT musí být odvozený ze základní třídy `Generic` (přímo nebo nepřímo). Třída `Generic` definuje základní vlastnosti a funkce. Na této obecné třídě staví další abstraktní třídy:

- třída `GenericWithPose` – rozšiřuje základní třídu o vlastnost `pose`, která obsahuje pozici a orientaci objektu,
- třída `CollisionObject` – umožňuje přidat objektu kolizní model,

- třída `Robot` – přidává API související s roboty,
- třída `MultiArmRobot` – rozšiřuje třídu `Robot` o podporu více ramen,
- třída `Camera` – přidává API pro kameru Azure Kinect.

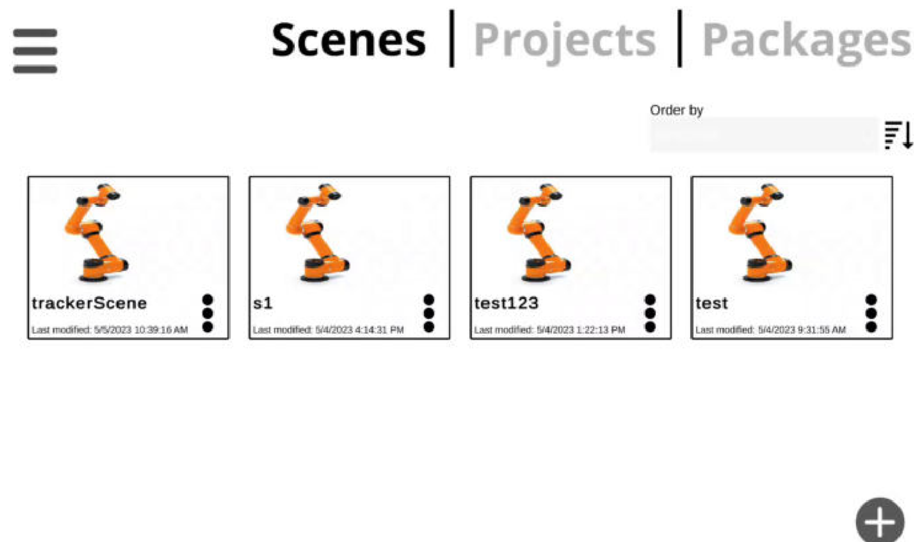
Cílem OT je přidat nějaké akce do systému. Aby se metoda třídy mohla považovat jako akce, tak musí splňovat určité vlastnosti – musí mít například definovaný docstring, musí být typově anotované a musí mít speciální atribut `__action__`.

2.3.2 AREditor

AREditor představuje hlavní uživatelské rozhraní systému ARCOR2. Přesněji se jedná o aplikaci založenou na Unity pro mobilní zařízení s podporou rozšířené reality. Díky tomu, že je aplikace kolaborativní, tak se všem uživatelům připojeným na stejný ARServer zobrazují totožné věci. Aplikace se podle funkčnosti dělí do tří hlavních částí – anotace zařízení v prostoru a přidávání jednotlivých OT v sekci **Scenes**, vytváření programu spojováním akcí v sekci **Projects** a správa balíčků v sekci **Packages**.

Scéna

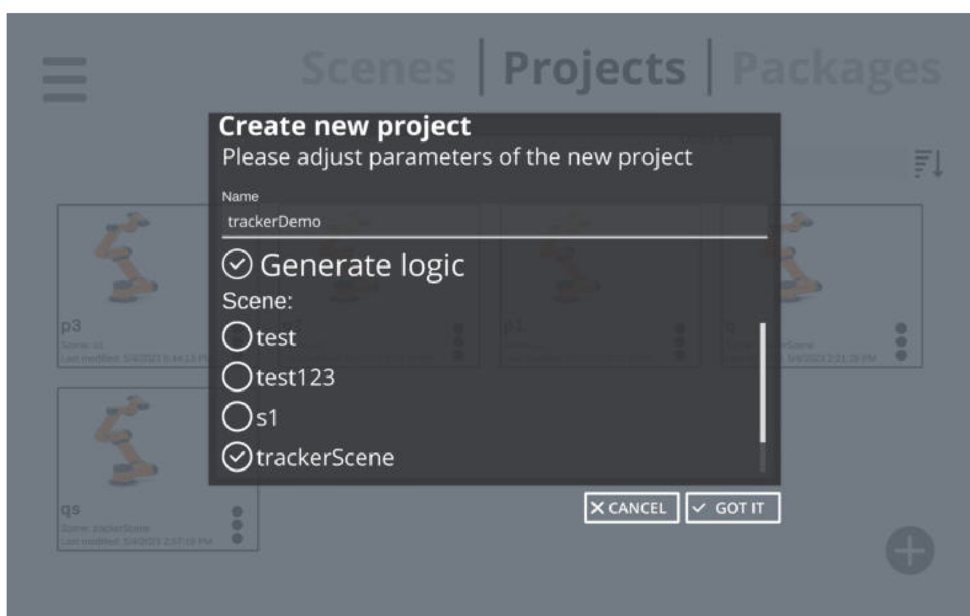
Scéna představuje anotovaný prostor, ve kterém se nachází všechny objekty, se kterými chce uživatel pracovat. Objekt může představovat abstraktní třídy nebo jednotlivé zařízení, které se na pracovišti opravdu nachází (robot, kamera). Takovým zařízením se musí nastavit ve scéně pozice. To lze udělat manuálním přemístěním objektu v rozšířené realitě. Pozici některých objektů lze nastavit pomocí speciální kalibrace. Lze přidat i virtuální kolizní objekty (speciální typ OT), které slouží pro anotaci kolizní zóny.



Obrázek 2.9: Scény v aplikaci AREditor

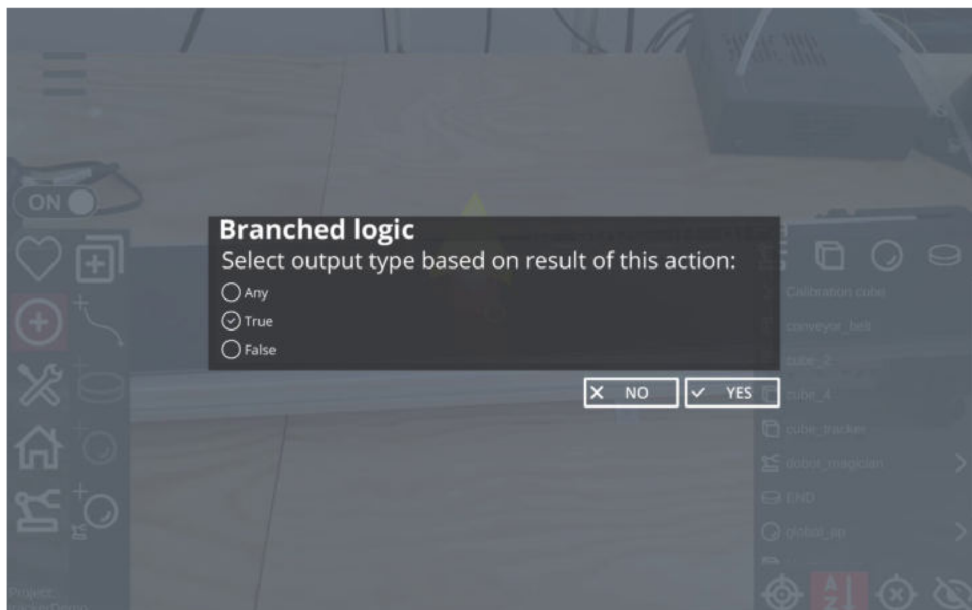
Projekt

Projekt vychází z již vytvořené scény. Právě zde probíhá programování funkčnosti daných objektů. Jako první se v prostoru musí vytvořit akční body. Akční bod lze vytvořit buďto vytvořením v místě, kam je namířena kamera mobilního zařízení (tedy ve středu obrazovky), a nebo vytvořením akčního bodu v místě, kde se nachází robotické rameno. Tato možnost funguje pouze v případě, že se ve scéně nachází robot. Body pak lze podobně jako objekty ve scéně libovolně posouvat. Vytvořeným akčním bodům poté uživatel přiřazuje specifické akce. Tyto akce jsou definované v OT, které jsou nahrané na ARServeru. Akcím lze nastavit různé parametry, které upřesňují chování dané akce. Uživatel poté propojováním jednotlivých akcí vytváří přesný tok programu, kde akce `START` značí začátek programu a akce `END` konec programu. Akce na sebe navazují a je tak možné výstup jedné akce nastavit jako vstupní parametr nadcházející akce.



Obrázek 2.10: Vytvoření projektu založeného na scéně v aplikaci AREditor

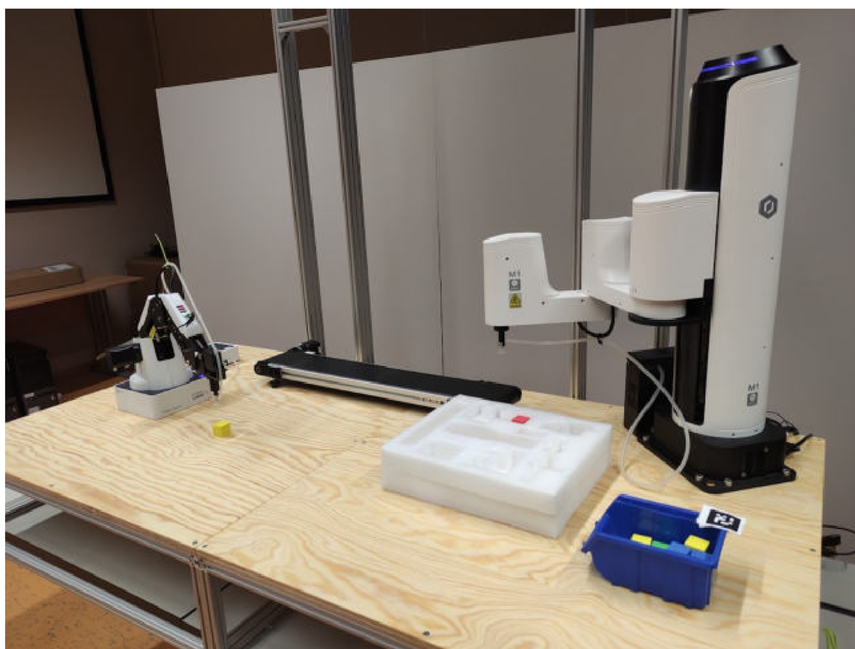
Vhodné je taky zmínit, že některé akce mohou vracet pravdivostní hodnotu. Při propojování takové akce se uživateli zobrazí nabídka, pro kterou pravdivostní hodnotu chce program propojit. Díky tomuto lze program větvit a je tak možné vytvořit například nějaké cykly s určitou podmínkou.



Obrázek 2.11: Větvení programu podle pravdivostní hodnoty

2.3.3 Robotické pracoviště

Robotické pracoviště představuje místo pro demonstraci systému ARCOR2 na kterém se nachází zařízení, se kterými chceme pracovat. V této práci se jako robotické pracoviště označuje robotická laboratoř O104 na FIT VUT. V laboratoři se nachází roboti Dobot M1 a Dobot Magician, dopravníkový pás a kamera Azure Kinect. Důležité jsou i kalibrační body sloužící pro kalibraci scény rozšířené reality a kamery.



Obrázek 2.12: Robotické pracoviště v univerzitní laboratoři O104

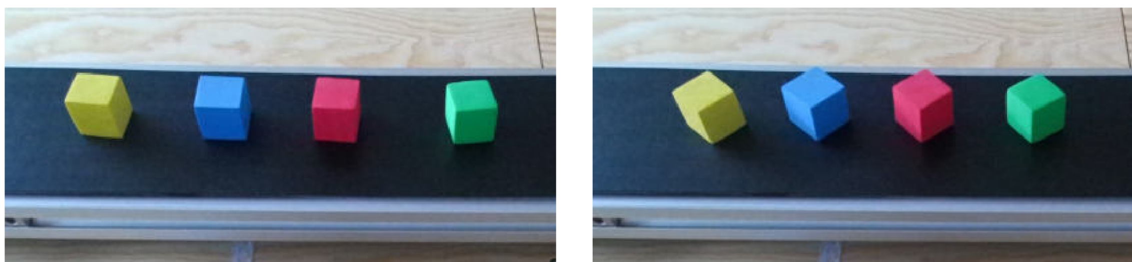
Kapitola 3

Návrh řešení

Cílem této práce je rozšířit systém ARCOR2 o modul umožňující detekci barevných kostek. Do systému ARCOR2 budou implementovány dva nové moduly. Hlavní modul bude zajišťovat samotnou funkcionalitu detektoru. V modulu se budou nacházet i podpůrné skripty jako třeba aplikace s uživatelským rozhraním, kde bude možné vyzkoušet a vizualizovat jednotlivé kroky algoritmu. Druhým modulem bude tracker. Ten si bude periodicky získávat data z detektoru (pozice detekovaných kostek) a sledovat je. Díky trackeru bude možné sloučit výsledky z více detektorů. Na pracoviště tak může být více kamer snímajících scénu z různých úhlů.

Základním krokem k detekci kostek je příprava dat. Modul pro obsluhu kamery Azure Kinect je už v systému implementovaný, takže data pro detektor půjde jednoduše získat přes API službu Kinectu.

Samotná detekce bude fungovat na jednoduchém principu. Kostky, které chceme detekovat, budou mít jen jednu ze čtyř barev -- červenou, zelenou, modrou nebo žlutou. Díky tomu lze rozdělit samotný obraz na čtyři, kde na každém obrazu bude pouze vyfiltrované barevné spektrum. Filtrace bude probíhat na barevném modelu HSV. Pro každou barvu se nalezne přesný rozsah, podle kterého se vytvoří maska obsahující pouze danou barvu. Tato maska se aplikuje na RGB obraz i na hloubkový obraz, čímž se vytvoří vyfiltrovaný RGB-D obraz. Po filtraci obrazu podle barev se vytvoří čtyři mračna bodů. Tato mračna bodů se nejdříve převedou do uniformního rozlišení pomocí Open3D funkce `voxel_down_sample(voxel_size=0.0015)`. Každé mračno bodů se poté rozdělí na shluky pomocí algoritmu DBSCAN. Výsledkem tohoto kroku budou jednotlivé shluky -- opět jako mračna bodů. Za ideálních podmínek by ve shluku měla být pouze jedna kostka bez jiných objektů. Stačí tedy projít každý jeden shluk a zjistit, jestli představuje hledaný geometrický tvar. To zjistíme tak, že budeme hledat roviny představující stěny kostky. Úhly těchto rovin se pak mezi sebou porovnají. Obyčejná kostka má viditelně dvě stěny při pohledu na kostku kolmo a tři stěny, když je kostka šikmo.

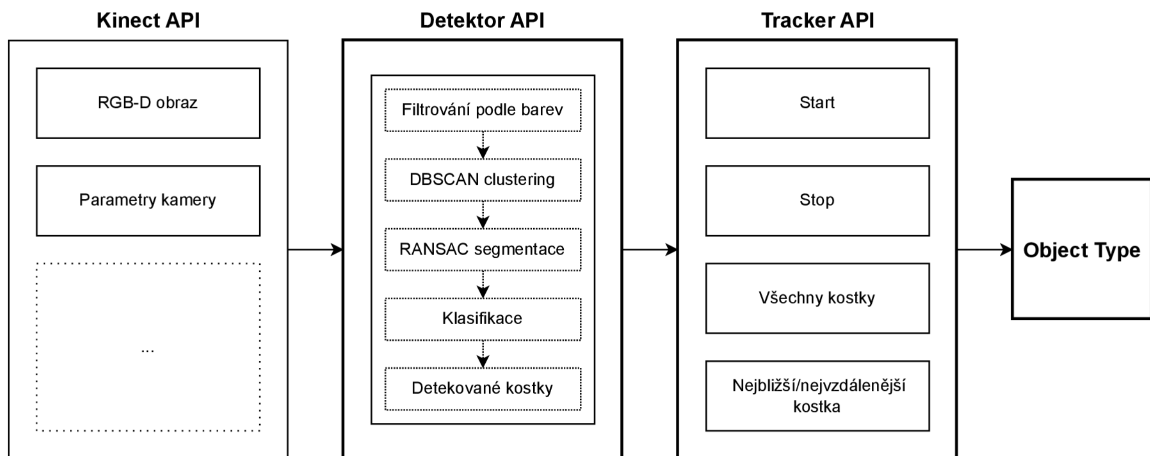


(a) Dvě viditelné strany (b) Tři viditelné strany

Obrázek 3.1: Viditelné strany kostek

Pro každý shluk se tedy musí zjistit, jestli se v něm nachází alespoň dvě roviny. K tomu slouží další algoritmus RANSAC, díky kterému lze najít všechny roviny ve shluku. Pokud bude ve shluku nalezena pouze jedna rovina, tak se určitě nejedná o kostku. V ostatních případech se porovnají úhly mezi rovinami. Pokud budou roviny aspoň částečně na sebe kolmé, přejde se k dalším kontrolám. Jelikož bylo mračno bodů převedeno do uniformního rozlišení, tak lze kontrolovat počet bodů v rovině. Počet bodů v rovině musí být v určitém rozmezí, jinak by se jednalo o větší nebo menší rovinu, než tvoří stěna kostky. Takové kontroly by měly odhalit i potenciální falešně pozitivní detekce. Posledním krokem bude vypočítat souřadnice středu kostky. U získaných rovin jsou známy i jejich rovnice. Je tak možní najít všechny body, které leží na více rovinách zároveň. Takové body tvoří hranu kostky. Z toho lze lehce zjistit souřadnice středu hrany a souřadnice středu každé stěny kostky. Pak už stačí jen vypočítat směrový vektor od středu hrany ke středům stěn a přičíst velikosti odpovídající půlce kostky.

Tracker bude navazovat na detektor. Jeho funkcí bude periodické volání detektoru, ze kterého si bude ukládat detekované kostky. Kostky se mezi iteracemi mohou libovolně posouvat. Tracker vždy aktualizuje každou uloženou kostku nejbližší detekovanou kostkou stejné barvy. Při aktualizaci kostky se provede průměrování staré a nové pozice. Jelikož není detekce deterministická, tak se může stát, že v některých iteracích se správně nedetekuje kostka, která má být detekovaná. Tracker s tím musí počítat a proto si bude uchovávat kostky až po dobu 5 iterací. Pokud se uložená kostka do 5 iterací nedetekuje, tak se ze seznamu odstraní. Služba bude poskytovat REST API rozhraní, přes které půjde zapnout a vypnout periodické volání detektoru a koncové body pro získání všech uložených kostek nebo pro získání nejbližší a nejvzdálenější kostky. Zároveň bude s trackerem vytvořen nový Object Type, který bude právě k tomuto API rozhraní přistupovat.



Obrázek 3.2: Příklad komunikace modulů

Pro integraci do systému ARCOR2 se používá Object Type. Zde bude vytvořený OT, který bude pracovat s API rozhraním trackeru. Tento OT bude poskytovat akce, které usnadní práci při manipulování s kostkami. K tomu budou sloužit akce, které vrátí pozice detekovaných kostek v reálném čase. Bude tak možné vrátit pozici nejbližší nebo nejvzdálenější kostky od určitého akčního bodu. Kostky bude možné i filtrovat podle barvy, díky tomu půjde pro každou barvu vytvořit samostatná akce. Další možnosti bude poskytovat akce, která zjistí, zda se v okolí akčního bodu nachází kostka. Tato akce bude vracet pravdivostní hodnotu, která umožní rozvětvit program.

Kapitola 4

Implementace

Jak již bylo zmíněno v kapitole 3, řešení je rozděleno do dvou modulů, `arcor2_cube_detector` a `arcor2_cube_tracker`. Výsledný Object Type využívající funkcionalitu těchto modulů je pak implementovaný v modulu `arcor2_fit_demo`. Pro oba tyto moduly byly také vytvořeny Docker kontejnery. Kód je dokumentován pomocí komentářů. Veškeré funkce a metody obsahují docstringy ve formátu sphinx [1]. Kód také splňuje typovou kontrolu Mypy [2].

4.1 Detektor

Jako první byla vytvořena aplikaci s uživatelským rozhraním (vizualizér) s pomocí již zmíněné knihovny Open3D. Díky této aplikaci bylo možné prohlížet mračna bodů a ladit jednotlivé kroky detekce. Postupně byla rozšiřována funkcionalita, aby mohl být zobrazen každý krok algoritmu. Algoritmus 1 zobrazuje pseudokód popisující jednotlivé kroky algoritmu.

Algoritmus 1: Hlavní smyčka detektoru

```
1  načti obraz z kinectu
2  vyfiltruj obraz podle barev
3  foreach barva do
4      rozděl mračno bodů na shluky
5      foreach shluk do
6          najdi všechny roviny
7          vypočítej úhel mezi rovinami
8          najdi kandidáty
9          if existují kandidáti then
10             zjisti, jestli se jedná o kostku
11             if jedná se o kostku then
12                 přidej polohu kostky do seznamu
13             end
14         end
15     end
16 end
17 return seznam pozic detekovaných kostek
```

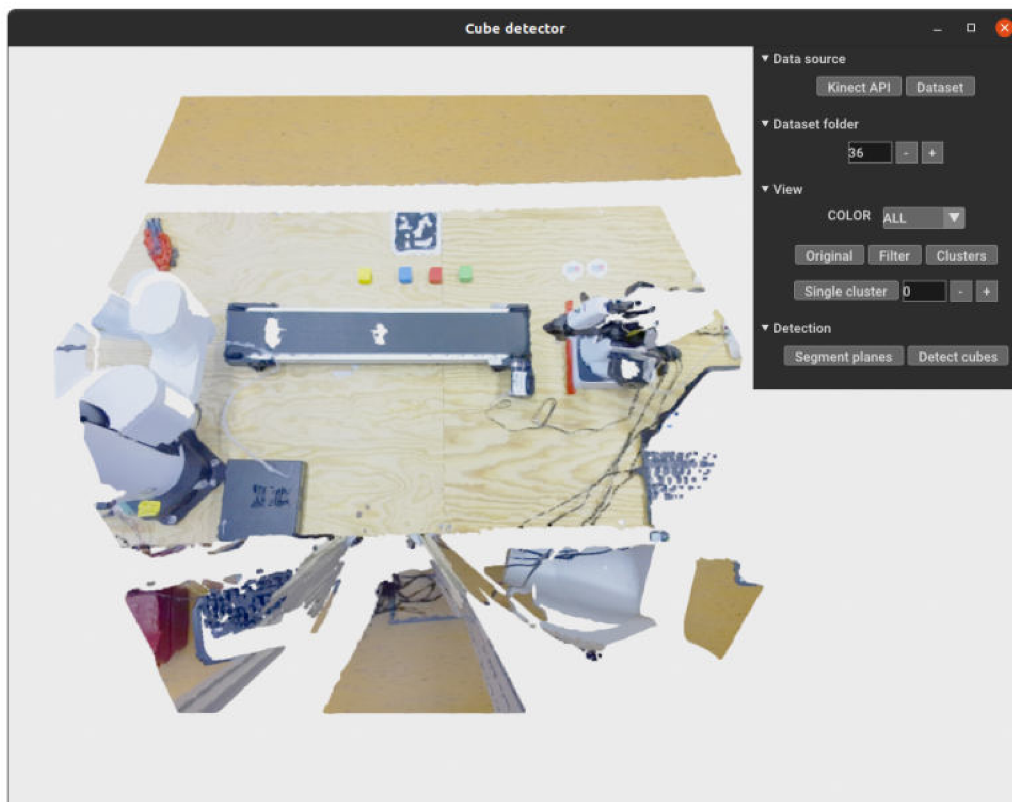
[1] https://www.sphinx-doc.org/

[2] https://mypy.readthedocs.io/

4.1.1 Vizualizér

Před implementací do systému ARCOR2 byla vytvořena aplikace s uživatelským rozhraním, která slouží pro zobrazování a ladění jednotlivých kroků algoritmu.

Samotná aplikace je vytvořena s pomocí knihovny Open3D. Scéna je vytvořena pomocí třídy `open3d.visualization.gui.SceneWidget` a rendrováni zajišťuje třída `open3d.visualization.rendering.Open3DScene`. Všechny snímky v následujících kapitolách jsou pořízené právě z této aplikace.



Obrázek 4.1: Aplikace s uživatelským rozhraním

Pomocí následujících ovládacích prvků lze zobrazit jednotlivé kroky algoritmu:

- **Kinect API** – načte RGB-D data z API služby Kinect Azure
- **Dataset** – načte RGB-D data ze souborů ve složce `data/dataset/`. V této složce se nachází celkem 45 RGB-D snímků, které lze pomocí vstupu pod tlačítkem přepínat.
- **COLOR** – výběrem z možnosti lze změnit zobrazovanou barvu.
- **Original** – zobrazí původní mračno bodů.
- **Filter** – zobrazí mračno bodů vyfiltrované podle barvy (viz obr. 4.2).
- **Clusters** – zobrazí všechny shluky (viz obr. 4.3).
- **Single cluster** – zobrazí jednotlivé shluky (viz obr. 4.4). Pomocí číselného vstupu vedle tlačítka lze mezi shluky přepínat.

- **Segment lanes** – zobrazí nalezené roviny (viz obr. 4.5).
- **Detect cubes** – zobrazí detekované kostky (viz obr. 4.7).

4.1.2 Filtrování obrazu podle barev

Tento krok pracuje primárně s knihovnou OpenCV [2].
Pomocí funkce `hsv_obraz = cv2.cvtColor(rgb_obraz, cv2.COLOR_RGB2HSV)` je napřed RGB obraz převeden do formátu barevného modulu HSV. Poté je pro každou ze čtyř barev vytvořena maska podle vlastně definovaného HSV rozsahu každé barvy.
`maska = cv2.inRange(hsv_obraz, spodní_HSV_limit, horní_HSV_limit)`.

Barva	Spodní HSV limit	Horní HSV limit
Červená	[0, 100, 50]	[10, 255, 255]
	[165, 100, 50]	[180, 255, 255]
Zelená	[40, 100, 50]	[85, 255, 255]
Modrá	[100, 175, 75]	[135, 255, 255]
Žlutá	[28, 100, 50]	[40, 255, 255]

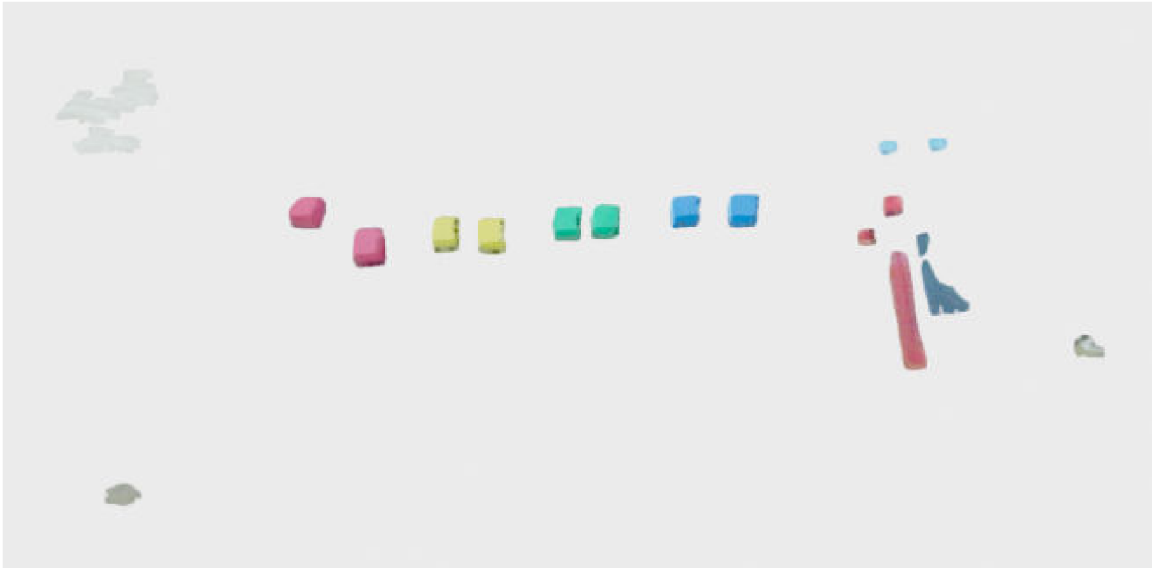
Tabulka 4.1: HSV rozsah pro barvy kostek

Maska se aplikuje na původní RGB a hloubkový obraz, čímž vznikne vyfiltrovaný RGB-D obraz, ze kterého se vytvoří mračno bodů. Na toto mračno bodů se zavolá funkce z knihovny Open3D `voxel_down_sample(voxel_size=0.0015)`, která sníží počet bodů v mračnu a zároveň převede mračno na uniformní rozlišení.

Obrázek 4.2: Mračna bodů rozdělená podle barev

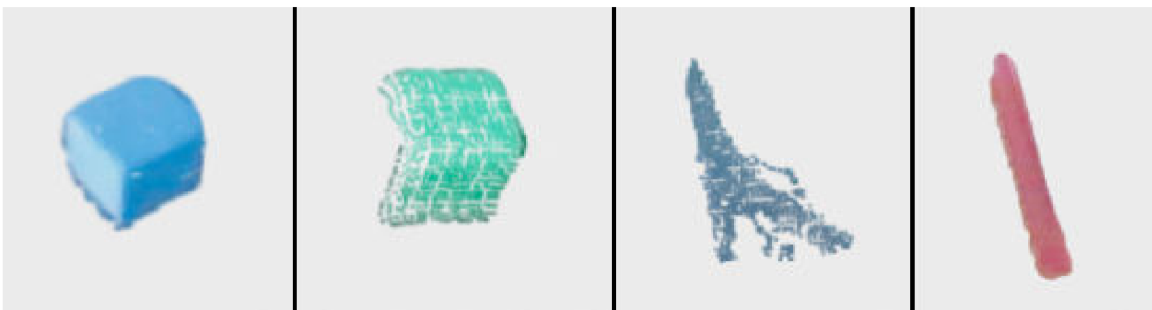
4.1.3 Rozdělení mračna bodů do shluků

Pro každé mračno bodů se provede DBSCAN shlukování. Tento algoritmus je implementován v knihovně Open3D, stačí pouze nad mračnem bodů zavolat funkci `cluster_dbscan()`. V sekci 2.1.2 jsou zmíněné parametry *epsilon* a *MinPts*. Hodnoty těchto parametrů mohou hodně ovlivnit výsledky. S těmito hodnotami se experimentovalo a nejlepších výsledků bylo dosaženo s hodnotami `epsilon = 0.015` a `MinPts = 100`.



Obrázek 4.3: Všechny shluky

Výsledkem toho kroku jsou jednotlivé shluky. Místo jednoho mračna bodů pro každou barvu je jich po tomto kroku tedy několik. Tímto krokem se zároveň odstranil šum (všimněte si různých bodů na obrázku 4.2 v porovnání s obrázkem 4.3). Shluky se dále budou zpracovávat individuálně. Na obrázku 4.4 jsou ukázány čtyři shluky. V prvním shluku je kostka se třemi viditelnými stranami, ve druhém je kostka se dvěma stranami a na dalších dvou se vůbec nejedná o kostky.



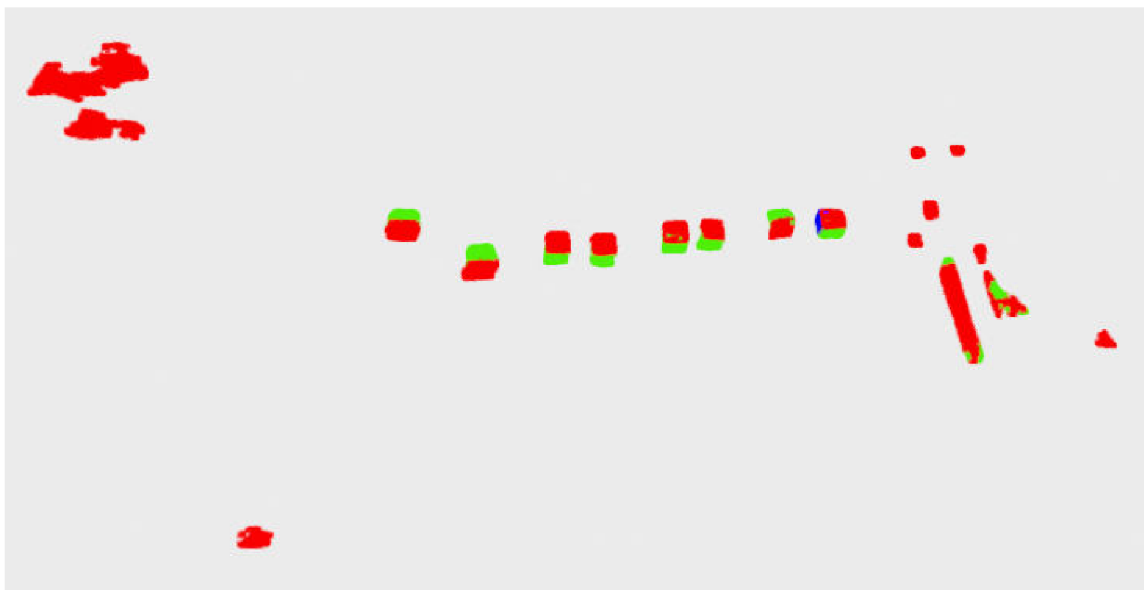
Obrázek 4.4: Jednotlivé shluky

4.1.4 Segmentace rovin

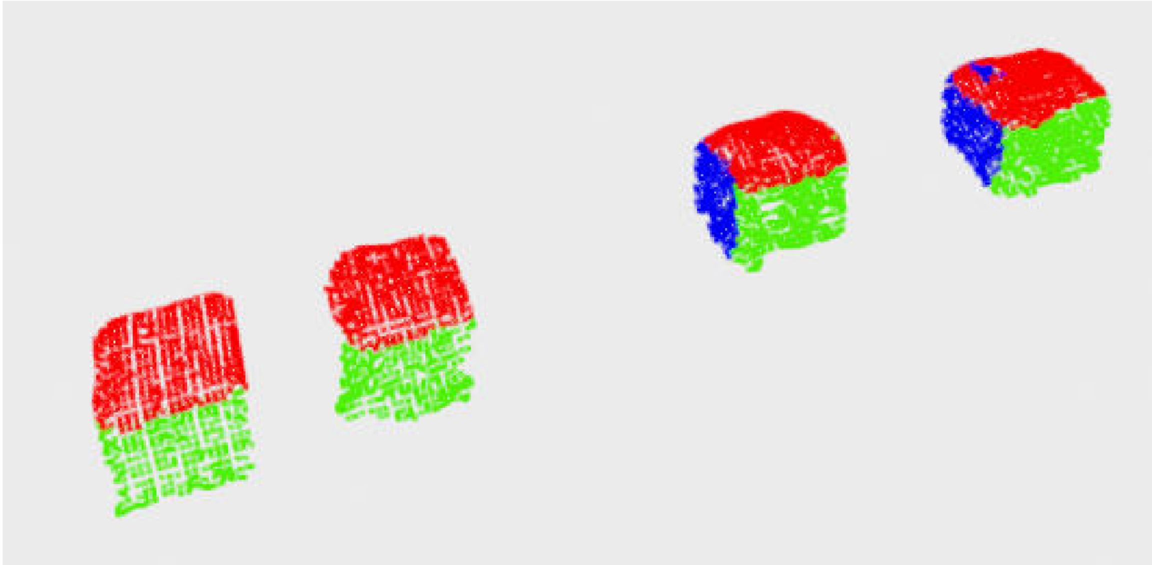
Pro každý shluk se najdou všechny roviny a jejich matematické modely. Jelikož algoritmus RANSAC vrací pouze jednu rovinu, tak je tento krok provedený ve smyčce. Před začátkem smyčky se nastaví počáteční shluk jako outlier a smyčka se opakuje, dokud má outlier více než 50 bodů. Na segmentaci byla použita funkce z knihovny Open3D
`segment_plane(distance_threshold=0.002, ransac_n=3, num_iterations=100)`.
Tato funkce vrací dvě hodnoty, seznam indexů inlier bodů a model roviny jako hodnoty (a, b, c, d). Pro každý bod (x, y, z) na rovině pak platí $ax + by + cz + d = 0$. Ze získaných indexů se získá samotný inlier (rovina) a nový outlier (tedy mračno bodů bez nalezené roviny). Pokud má rovina (inlier) více než 50 a méně než 500 bodů, tak se přidá i s jejím modelem do seznamu nalezených rovin. Touto podmínkou se vyfiltrují příliš malé roviny (menší než strana kostky) a příliš velké roviny (například podlaha, stůl). Tento proces se opakuje, dokud se v shluku nenaleznou všechny roviny.

Algoritmus 2: Segmentace rovin

```
1 nastav počáteční shluk jako outlier
2 while počet bodů v outlieru je větší než stanovené minimum do
3 │ v outlieru najdi rovinu pomocí RANSAC
4 │ aktualizuj outlier
5 │ if počet bodů na rovině je ve stanoveném rozsahu then
6 │ │ přidej rovinu do seznamu rovin
7 │ end
8 end
9 return seznam rovin
```



Obrázek 4.5: Nalezené roviny (vizualizovány odlišnými barvami)



Obrázek 4.6: Kostky se dvěma a třemi stranami

Ve většině shluků se nalezne pouze jedna rovina, z čehož lze vyhodnotit, že v takových shlucích se kostka nenachází. Shluk obsahující kostku bude mít minimálně 2 nebo 3 roviny. Takový případ lze vidět na obrázku 4.6, kde se nachází obě varianty kostek.

Jelikož RANSAC algoritmus není deterministický, tak se budou výsledky tohoto kroku pokaždé mírně lišit. Může se tedy stát, že se v jedné iteraci roviny naleznou správně ale v další iteraci už ne.

4.1.5 Výpočet úhlů mezi rovinami

Dalším krokem bude výpočet úhlů mezi nalezenými rovinami. Pomocí rovnice (4.1) [9] se vypočítají úhly mezi každou nalezenou rovinou. Tyto úhly se pak ještě musí převést z radiánů na stupně pomocí funkce `math.degrees()`. Výsledkem je vnořený slovník, kde klíče jsou indexy rovin a hodnoty vnořeného slovníku jsou úhly mezi rovinami ve stupních (tabulka).

$$A = \cos^{-1}\left(\frac{(a_1 * a_2 + b_1 * b_2 + c_1 * c_2)}{\left(\sqrt{a_1 * a_1 + b_1 * b_1 + c_1 * c_1}\right) * \left(\sqrt{a_2 * a_2 + b_2 * b_2 + c_2 * c_2}\right)}\right) \tag{4.1}$$

4.1.6 Hledání kandidátů

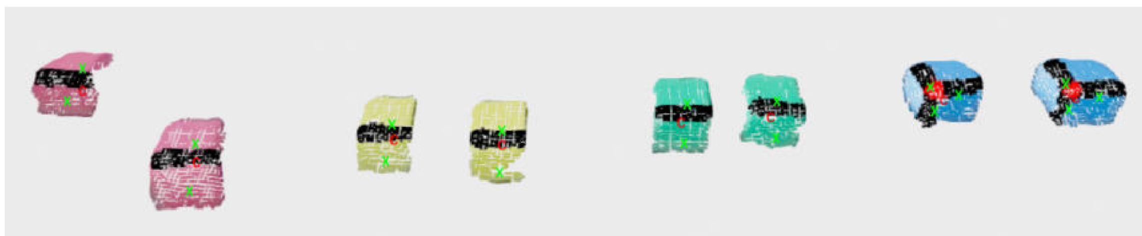
Cílem dalšího kroku je porovnat mezi sebou úhly rovin a zjistit, jestli jsou roviny mezi sebou kolmé. V minulých krocích byly odstraněny shluky, ve kterých byla nalezena pouze jednu rovina. Pracuje se zde se seznamem indexů všech rovin shluku. Kandidát představuje seznam dvojic nebo trojic indexů rovin, které jsou na sebe kolmé. Tyto skupiny kandidátů obsahují pouze indexy roviny, které splňují podmínky.

Ideálně by mezi rovinami představující strany kostek mělo být 90 stupňů, ale kvůli různým deformacím a nedeterminičnosti algoritmu RANSAC jsou tyto úhly trochu zkreslené. Neporovnává se tedy, jestli jsou roviny úplně kolmé, ale jestli je úhel mezi rovinami větší než 50 stupňů. Takto nízká hodnota je zvolena pouze pro krajní případy (viz obr. 4.9), ve většině případů je úhel mezi stěnami kostek kolem 80 stupňů.

4.1.7 Klasifikace kandidátů

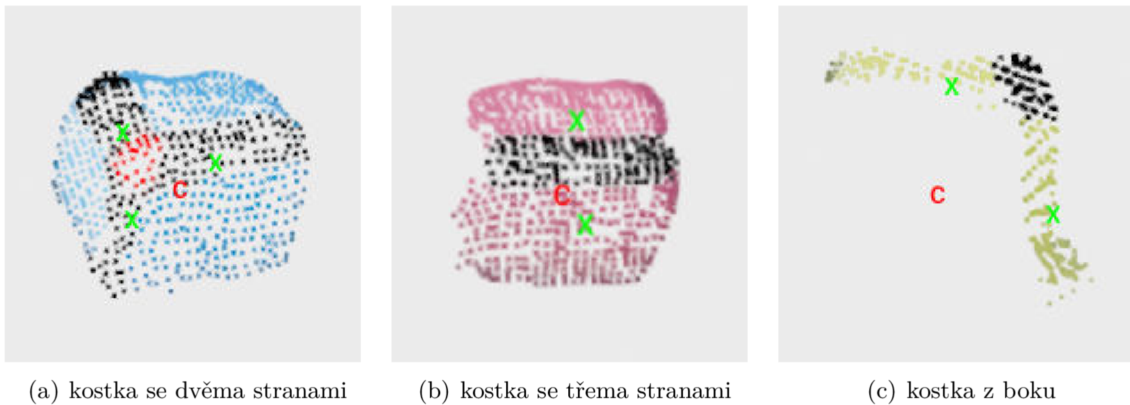
Každá trojice nebo dvojice tvořící kandidáta se převede na mračno bodů obsahující pouze dané roviny. Jako první je potřeba detekovat hrany kostky. Hrany se rozpoznají tak, že pro každý bod v mračnu zkontroluje, jestli bod leží na dvou rovinách zároveň. V tomto kroku je dostupné mračno bodů shluku a model roviny pro každou rovinu. V cyklu se projde každý bod a pomocnými funkcemi se zjistí, jestli bod patří na hranu nebo na roh kostky. Tyto body pak se pak sjednotí a získá se mračna bodů obsahující pouze jednotlivé hrany a mračno bodů obsahující rohové body. Hrany jsou na obrázku 4.7 zobrazeny černou barvou a roh červenou barvou (roh je pouze u kostek, které mají tři viditelné strany). Následující pomocné funkce byly vytvořeny a použity:

- `is_point_on_plane (leží bod na rovině)` – aby bod ležel na rovině, tak pro něj musí platit $ax + by + cz + d < 0.004$.
- `is_point_on_edge (leží bod na hraně)` – aby bod ležel na hraně, tak musí ležet na dvou rovinách zároveň.
- `is_point_on_corner (leží bod na rohu)` – aby bod ležel na hraně, tak musí ležet na třech rovinách zároveň.



Obrázek 4.7: Všechny detekované kostky

Při počítání středu kostky se trochu liší postup podle počtu viditelných hran. U kostky se třemi viditelnými stranami se vypočítají tři směrové vektory od hrany kostky ke každému středu hrany (střed hrany se získá zavoláním Open3D funkce `get_center()` nad body, které tvoří hranu). Tyto vektory se poté normalizují a nastaví se jim velikost odpovídající zhruba polovině kostky (0.0125). Vektory jsou poté přičteny k souřadnicím rohu kostky, čímž se získají souřadnice středu kostky. Dále zde probíhá kontrola, jestli je průměrná vzdálenost od rohu do středu hrany podobně velká, jako polovina kostky. U kostky se dvěma viditelnými stranami je tento proces téměř identický. Liší se akorát v tom, že se místo rohu bere střed hrany (hrana je v tomto případě pouze jedna) a místo středů hran se berou středy stran. Toto lze vidět na obr. 4.7, 4.8 – zelené X značí střed hrany/střed strany, červené C značí střed kostky.



(a) kostka se dvěma stranami (b) kostka se třema stranami (c) kostka z boku

Obrázek 4.8: Detekované kostky

4.1.8 Problémy při implementaci

Při vývoji algoritmu se vyskytlo pár problémů. Některé se podařilo vyřešit a některé ne. Jeden z prvních problému bylo správné určení HSV rozsahu pro jednotlivé barvy. Aby používaný princip fungoval, tak se musí vyfiltrovat pouze konkrétní barvy. Při špatné filtraci se kromě samotné kostky může v mračnu bodů objevit například zem, nebo naopak se nemusí objevit tmavší strana kostky. Kvůli tomu je pro nejlepší výsledky doporučeno mít prostředí co nejlépe osvětleno. Použité HSV hodnoty 4.1 byly postupně upravovány a testovány s různým osvětlením.

Další překážkou bylo vybrání správných parametrů pro funkci `segment_plane()` a `cluster_dbscan()`. Tyto parametry nejvíce ovlivňují výsledek detekce a nějaký čas trvalo, než byly nalezeny ideální hodnoty.

Největším problémem však byly různé deformace v mračnu bodů. Zadní strana kostek je v mračnu bodů při určitých pozicích, což zároveň s menší velikostí kostek ovlivňuje mračno bodů. Kvůli tomuto nastává problém při porovnávání úhlů nalezených rovin, jak je již zmíněno v sekci 4.1.6. K ještě horšímu zkreslení nastává, když jsou na sebe kostky naskládané. V tom případě nelze vůbec rozlišit, že se v mračnu bodů nachází kostka (minimálně u kostky, která je úplně na vrchu). Tyto problémy lze řešit posunutím kamery tak, aby na kostky koukala nejlépe přímo a z vyšší pozice.

Obrázek 4.9: Deformace kostek

Algoritmus nefunguje správně, když v mračnu bodů po provedení shlukování bude více než jedna kostka. Nelze tak detekovat stejnobarevné kostky, které jsou naskládané blízko u sebe. Podmínkou pro detekci tedy je, že kostka musí být samostatně a nesmí se dotýkat žádných objektů se stejnou barvou. Kostek s jinými barvami se může dotýkat (viz horní obr. 4.9), tam díky filtrovaní podle barev k problémům nedochází

Algoritmus je nastaven na detekci kostek s předem určenou velikostí. Nelze tedy bez úpravy detekovat kostky všech velikostí.

4.2 Tracker

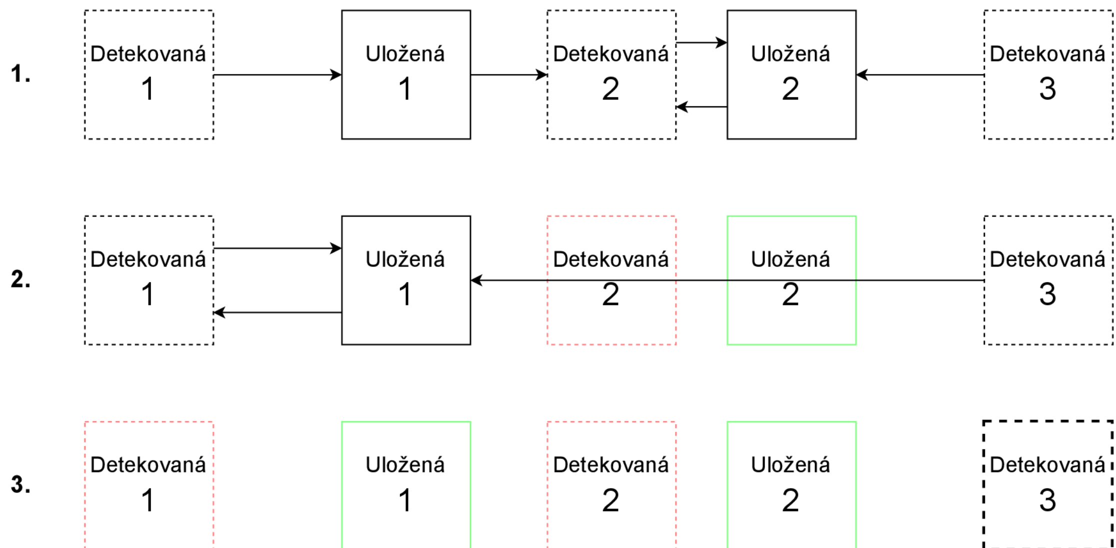
Cílem trackeru je sledovat a ukládat jednotlivé kostky. Hlavní princip spočívá v tom, že si periodicky získává detekované kostky pomocí API služby detektoru. Tyto kostky si pak ukládá a slučuje s detekovanými kostkami získané v předchozích voláních. Neimplementuje tedy žádnou důležitější funkčnost, pouze poskytuje úložiště pro pozice kostek, ze kterého pak dodává data pro Object Type.

Hlavní princip trackeru spočívá v ukládání kostek. Každou sekundu se musí uložit detekované kostky do seznamu `stored_cubes`. Hlavní myšlenkou toho systému je, že detekované kostky musí aktualizovat již uložené kostky – uložené jsou například kostky před jejich posunutím.

Uložené kostky mají vlastní datový typ `StoredCube`, který obsahuje samostatnou kostku (datový typ `Cube`) a takzvaný `up-to-date` index. Tento index značí takzvanou aktuálnost kostky. Jelikož je detekce nedeterministická, tak musí být kostky uloženy po dobu několika iterací. Pokud se uložená kostka nachází v detekovaných kostkách, tak se tento index vy-

nuluje, v opačném případě se inkrementuje. Pokud index přesáhne hodnotu 5 (tzn. nebyla pětkrát za sebou detekovaná), tak se kostka odstaní.

Spuštění trackeru spustí samostatné vlákno, které každou sekundu volá funkci pro uložení kostek získaných přes API službu detektoru. Jako první funkce inkrementuje hodnotu `up-to-date` všech kostek a zastaralé kostky odstraní. Dalším cílem je aktualizovat uložené kostky pomocí detekovaných kostek a přidat nové detekované kostky, které žádnou kostku neaktualizovali. Na vstupu jsou dva seznamy s uloženými pozicemi a s nově získanými pozicemi. Seznam s uloženými kostkami se bude průběžně aktualizovat, tudíž je nutno vytvořit jeho kopii. Tato kopie se bude používat na porovnávání s detekovanými kostky a originál se bude přepisovat. Dokud se neaktualizují všechny uložené kostky nebo dokud nebude seznam detekovaných kostek prázdných, tak se v cyklu budou hledat dvojice nejbližších kostek. Pro každou uloženou kostku se najde nejbližší detekovaná kostka se stejnou barvou. Pro tuto nalezenou kostku se taktéž najde nejbližší kostka stejné barvy. Pokud jsou obě kostky navzájem k sobě nejblíž, tak tato dvojice představuje tu stejnou kostku. V tom případě se uložená kostka aktualizuje. Podle enviromentální proměnné `ARCOR2_CUBE_TRACKER_AVERAGE_POSITION` buďto detekovaná kostka uloženou kostku přepíše nebo se kostka aktualizuje tak, že se zprůměruje pozice obou kostek. Detekovaná kostka se pak odstraní ze seznamu. Pokud po cyklu není seznam detekovaných kostek prázdný, tak se všechny kostky z tohoto seznamu uloží. Princip tohoto ukládání si ukážeme na jednoduchém příkladu.



Obrázek 4.10: Příklad ukládání kostek

Na obrázku 4.10 lze vidět dvě uložené kostky a tři detekované kostky. Uvažujme, že všechny kostky mají stejnou barvu a nejbližší kostky jsou vyznačeny šipkami. Na začátku platí podmínka, že všechny uložené kostky nejsou aktualizované a seznam detekovaných kostek není prázdný. V první iteraci cyklu se pro uloženou kostku 1 najde nejbližší detekovaná kostka 2, která má ale nejblíž uloženou kostku 2. Uložená kostka 1 se tedy neaktualizuje. Uložená kostka 2 a detekovaná kostka 2 jsou k sobě nejblíž, tudíž se aktualizují. Ve druhé iteraci je už detekovaná kostka 2 odstraněna, tudíž nejblíže k uložené kostce je detekovaná

kostka 1. V tomto případě se už kostka aktualizuje. V další iteraci už jsou všechny uložené kostky aktualizovány, takže se cyklus přeruší. Nakonec zbyla už pouze detekovaná kostka 3, která se uloží jako nová kostka.

Algoritmus 3: Tracker

```
1  while uložené kostky nejsou aktualizované a nové kostky nejsou zpracované do
2  |   foreach uložená kostka do
3  |   |   if kostka už byla aktualizovaná then
4  |   |   |   continue
5  |   |   end
6  |   |   najdi k uložené kostce nejbližší detekovanou kostku stejné barvy
7  |   |   if žádná kostka nebyla nalezena then
8  |   |   |   označ uloženou kostku jako aktualizovanou
9  |   |   |   continue
10 |   |   end
11 |   |   najdi k nalezené kostce nejbližší uloženou kostku
12 |   |   if uložená kostka je také nejblíž k nalezené kostce then
13 |   |   |   aktualizuj uloženou kostku pomocí detekované kostky
14 |   |   |   odstraň detekovanou kostku
15 |   |   end
16 |   end
17 end
18 foreach detekovaná kostka, která nebyla odstraněna do
19 |   ulož kostku
20 end
```

4.3 Object Type

Při integraci do sytém byl vytvořený Object Type s názvem `CubeTracker`. Object Type byl vytvořen speciálně do balíčku `fit_demo`, který obsahuje ostatní OT potřebné pro demonstraci na robotickém pracovišti. V tomto demu se pracuje s roboty Dobot M1, Dobot Magician, pásovým dopravníkem, kamerou Kinect Azure a nově i s detektorem/trackerem barevných kostek. Samotný OT je vytvořen ze základní třídy `Generic`. Při inicializaci automaticky zapíná API službu trackeru. Jelikož OT nepodporuje datový typ `list`, tak nelze udělat akci, která by vrátila seznam všech detekovaných kostek. Ve výsledku jsou vytvořeny pouze tři akce:

- `is_cube_in_area` – vrátí pravdivostní hodnotu podle toho, jestli se v prostoru akčního bodu nachází detekovaná kostka
- `get_nearest_cube` – vrátí pozici nejbližší kostku k akčnímu bodu
- `get_farthest_cube` – vrátí pozici nejvzdálenější kostku od akčního bodu

Výsledky těchto akcí lze ovlivnit různými parametry (`offset` se vztahuje pouze pro pozici nejbližší/nejvzdálenější kostky):

- `position (Position)` – akční bod, ke kterému se hledá nejbližší/nejvzdálenější kostka
- `max_distance (float)` – maximální vzdálenost, ve které můžou kostky být
- `color (Enum)` – barva kostky
- `offset_x (float)` – posun na ose x, který se přičte k souřadnicím kostky
- `offset_y (float)` – posun na ose y, který se přičte k souřadnicím kostky
- `offset_z (float)` – posun na ose z, který se přičte k souřadnicím kostky

Parametr `offset_z (float)` má ve výchozím nastavení hodnotu 0.125. Tím se docílí, že se vrátí hodnota horní strany místo samotného středu kostky, které vrací API služba trackeru. Ostatní parametry pro posun na osách pak slouží k tomu, aby šli získávat například pozice před kostkou nebo vedle kostky.

Při volání akcí pro získání nejbližší/nejvdálenější kostky může nastat, že okolo akčního bodu nebude vůbec žádná kostka. V takovém případě nelze vrátit žádnou pozici, takže se musí vrátit hodnota `None`. Taková hodnota však není v systému podporovaná, tudíž se toto musí při programování manuálně ošetřit. Právě k tomu slouží akce `is_cube_in_area`. Tato akce se musí volat před každým voláním akcí `get_nearest_cube` a `get_farthest_cube`. Ty se můžou vykonat pouze pokud akce potvrdí, že se u akčního bodu kostka nachází. Je důležité, aby pro všechny tyto akce byly parametry nastaveny stejně. Díky tomu že ARCOR2 podporuje větvení podle pravdivostní hodnoty, tak lze vytvořit v programu smyčku, která se bude provádět, dokud se budou v daném okolí kostky nacházet.

Kapitola 5

Testování

Testování probíhalo na několika úrovních. Byly vytvořeny klasické unit testy a vlastní dataset, který se vyhodnotil. Dále proběhli různé experimenty na robotickém pracovišti, kde se testovala přímo funkčnost modulů vytvořených v této práci.

5.1 Integrační a unit testy

V systému ARCOR2 se provádí testování pomocí frameworku pytest[1]. Pro samostatnou funkci detektoru i trackeru byly vytvořeny unit testy, které testují jednotlivé funkce.

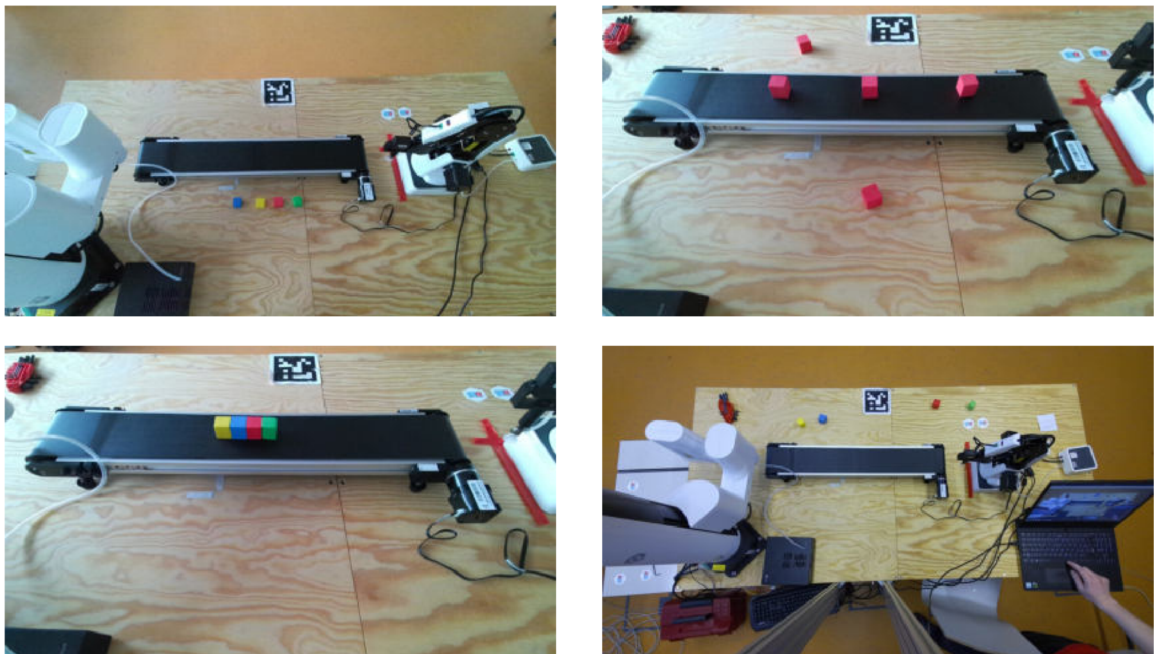
Pro detektor byly napřed vytvořeny testovací data tak, že se uložili vstupy a výstupy všech funkcí při volání detekce na ověřeném snímku. Tyto vstupy a výstupy jsou pak používány v testech, kdy se zavolá každá funkce s načtenými vstupy a porovnává se, jestli vrátí stejný výsledek jako uložený výstup. Jelikož tracker pracuje jenom s pozicemi a ne s mračny bodů, tak zde bylo lehčí vytvořit testovací data. Kromě klasických unit testů jsou pro oba moduly implementovány i testy na kontrolu API rozhraní.

Jelikož byl Object Type vytvořen v balíčku `fit_demo`, ke kterému již existují vytvořené integrační testy, tak nebylo potřeba žádné další integrační testy vytvářet.

5.2 Dataset a jeho vyhodnocení

Pro kontrolu detektoru byl vytvořen vlastní dataset. Tento dataset obsahuje celkem 45 dvojic RGB obrazu a hloubkového obrazu. Snímky byly pořízeny pomocí kamery Azure Kinect na robotickém pracovišti. Během pořizování byla kamera ve třech různých výškách, z každé výšky je 15 snímků.

[1] https://docs.pytest.org



Obrázek 5.1: Ukázka snímků z datasetu (pouze RGB obrázky)

Každý snímek je anotován pomocí CSV souboru. Anotace obsahuje informace o barvě a souřadnicích (x, y, z) kostky. Dataset byl vytvořen pouze pro testování souřadnic, takže anotace vůbec neuvádí orientaci kostek.
Dataset má pevně daný formát, `color.jpg`, `depth.png` a `annotation.csv` v samostatném souboru.

color	**x**	**y**	**z**
YELLOW	-0.31963	-0.34182	1.10498
BLUE	-0.23144	-0.35612	1.10988
RED	0.25523	-0.43349	1.13807
GREEN	0.40480	-0.43066	1.13776

Tabulka 5.1: Příklad anotovaného snímku

Dataset slouží pro evaluaci detektoru. Evaluaci provádí samostatný skript `dataset_evaluation.py`. Pro každý snímek se provede detekce a detekované kostky se porovnají s kostky načtené z anotovaného CSV souboru. Porovnávání funguje tak, že se spočítá vzdálenost mezi detekovanou kostkou a každou anotovanou kostkou. Pokud je tato vzdálenost menší než určená hodnota, tak se jedná o kostku, která je správně detekovaná. V opačném případě se detekovala kostka, která není anotovaná – tím pádem se jedná o kostku, která je chybně označena jako pozitivní (false positive). Skript si ukládá celkový počet správně detekovaných kostek, chybně detekovaných kostek a anotovaných kostek, které nebyly detekovány. Z těchto dat se pak vyvozuje úspěšnost detekce jako poměr správně detekovaných kostek a anotovaných kostek, které nebyly detekovány. Skript lze spouštět s několika parametry:

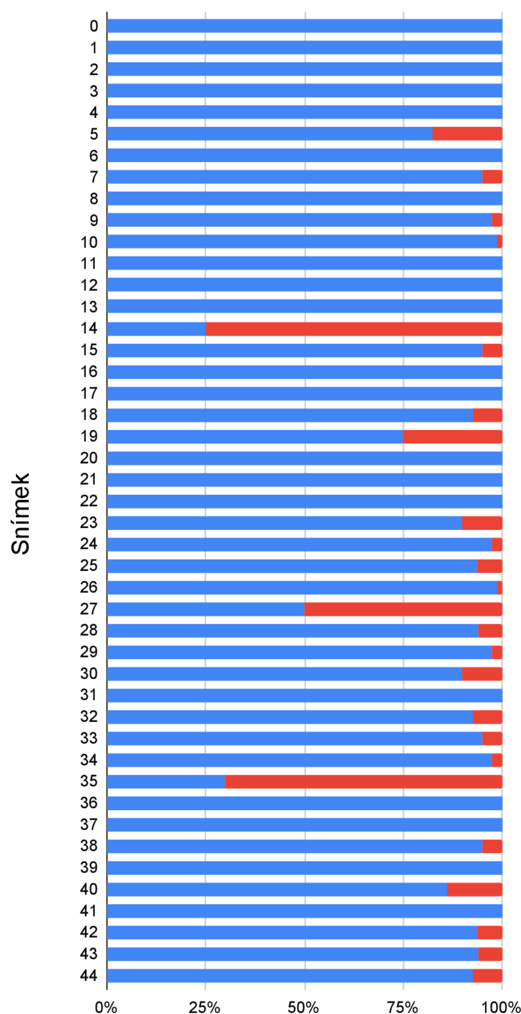
- `--debug` – do konzole se vypíší detailní informace o detekci jednotlivých snímků
- `--iterations` – počet iterací, ze kterých se zprůměrují data
- `--save` – cesta, kam se uloží CSV soubor s výsledky evaluace (pokud není zadaná, nic se neuloží)

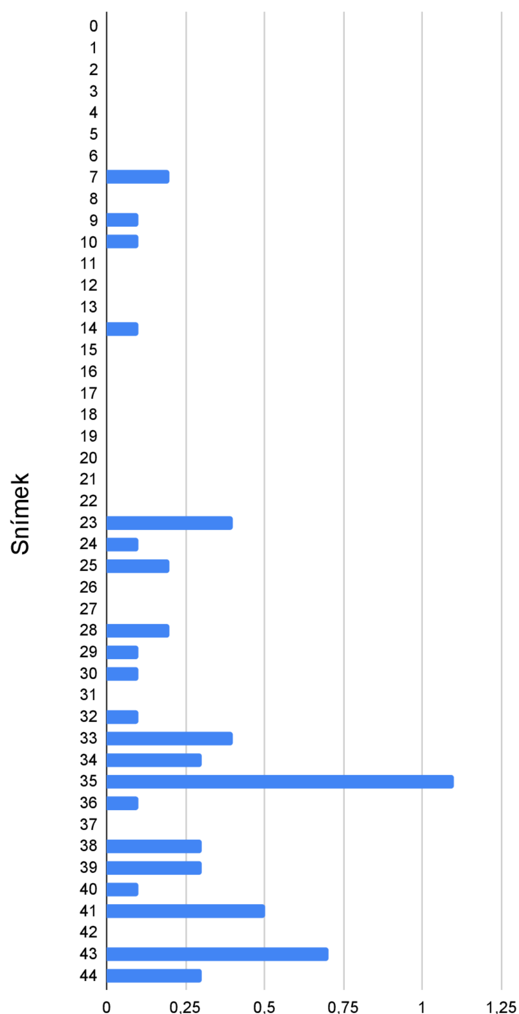
Vyhodnocení je zprůměrováno z deseti iterací. Obsahuje i výpočetní dobu pro jednotlivé snímky. Tato doba závisí na hardwarovém vybavení stroje, na kterým se spouští skript. Následující vyhodnocení bylo provedeno na serveru v robotické laboratoři O104.

Iterace	**Detekováno**	**Očekáváno**	**Úspěšnost**	**Chybné detekce**	**Čas**
1	198	217	91%	4	13.869s
2	201	217	92%	9	13.436s
3	201	217	92%	7	13.469s
4	196	217	90%	8	13.229s
5	204	217	94%	3	13.274s
6	204	217	94%	4	13.314s
7	198	217	91%	5	13.423s
8	200	217	92%	4	13.400s
9	194	217	89%	9	13.380s
10	201	217	92%	5	13.211s
Průměr	**199**	**217**	**91%**	**5**	**13.403s**

Tabulka 5.2: Vyhodnocení datasetu

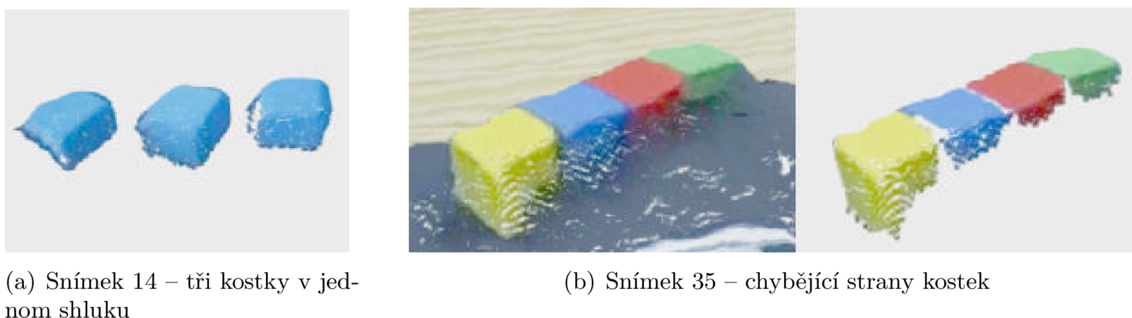
V tabulce 5.2 lze vidět, že byla dosažena úspěšnost 91% s 5 chybnými detekcemi. Z 217 anotovaných kostek jich tak v průměru bylo detekováno 199. V průměru bylo ve 45 snímcích detekováno pouze 5 chybných detekci.





Obrázek 5.2: Vyhodnocení datasetu

Z grafů na obr. 5.2 si lze všimnout, že detekce není moc úspěšná na snímcích 14 a 35. Na snímku 14 (5.3(a)) je problém v tom, že se v jednom shluku objevují tři kostky. Kvůli tomu se správně neprovede detekce. Na snímku 35 (5.3(b)) je zase problém při filtrování obrazu. Kostky se nachází na dopravníkovém pásu a senzor špatně zachytí data. Strany kostek mají špatnou barvu a po filtrování zůstane menší počet bodů v mračnu bodů, tím pádem se správně nenaleznou roviny představující strany kostek. Dále lze z grafu vyčíst počet chybně detekovaných kostek v každém snímku. Jediný snímek, ve kterém dochází k chybné detekci skoro v každé iteraci, je opět snímek 35. V ostatních snímcích nedochází k chybné detekci skoro vůbec nebo pouze jednou za několik iterací.

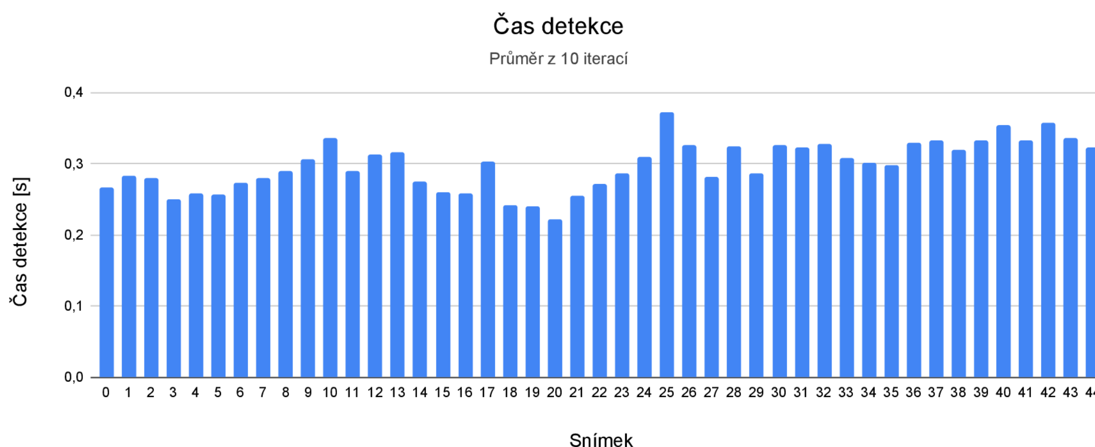


(a) Snímek 14 – tři kostky v jednom shluku

(b) Snímek 35 – chybějící strany kostek

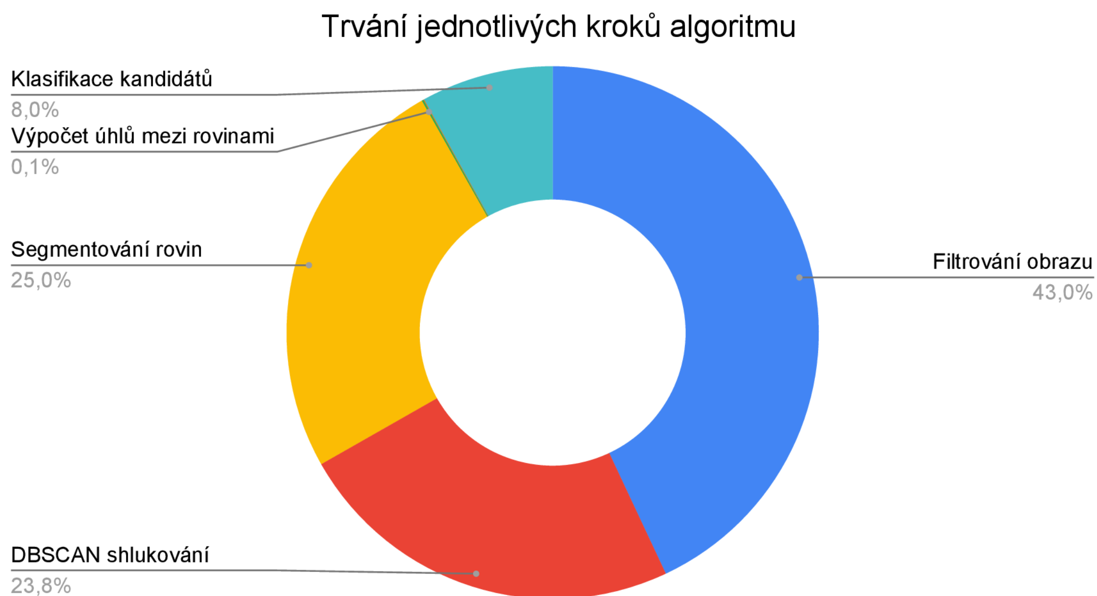
Obrázek 5.3: Problémy na různých snímcích

Dále byl změřen i celkový čas detekce. V grafu na obr. 5.4 lze vidět průměrný čas detekce každého snímku datasetu. Čas detekce je pro každý snímek podobný, průměrný naměřený čas byl 0,297s, maximální 0,424s a minimální 0,209s. Tento čas byl však dosažen na připravených datech, při detekování v reálném čase může být detekce pomalejší.



Obrázek 5.4: Průměrný čas detekce

V grafu na obr. 5.5 je zobrazeno, kolik trvají jednotlivé kroky algoritmu. Nejvíce času trvá samotné filtrování obrazu. V tomto kroku se provádí i převzorkování mračna bodů na uniformní rozlišeni, což vysvětluje tento zvýšený výpočetní čas. Shlukování a segmentování rovin pak zabírá další půlku celkového výpočetního času. Samotné hledání a klasifikace kandidátů pak zabírá malé procento výpočtu.



Obrázek 5.5: Procentuální trvání jednotlivých kroků

Kromě tohoto skriptu byl vytvořen i samostatný test `test_dataset.py`. Jedná se o zjednodušenou verzi předchozího skriptu, která pouze vyhodnocuje dataset. Aby test uspěl, tak musí být úspěšnost detekcí větší než 70%.

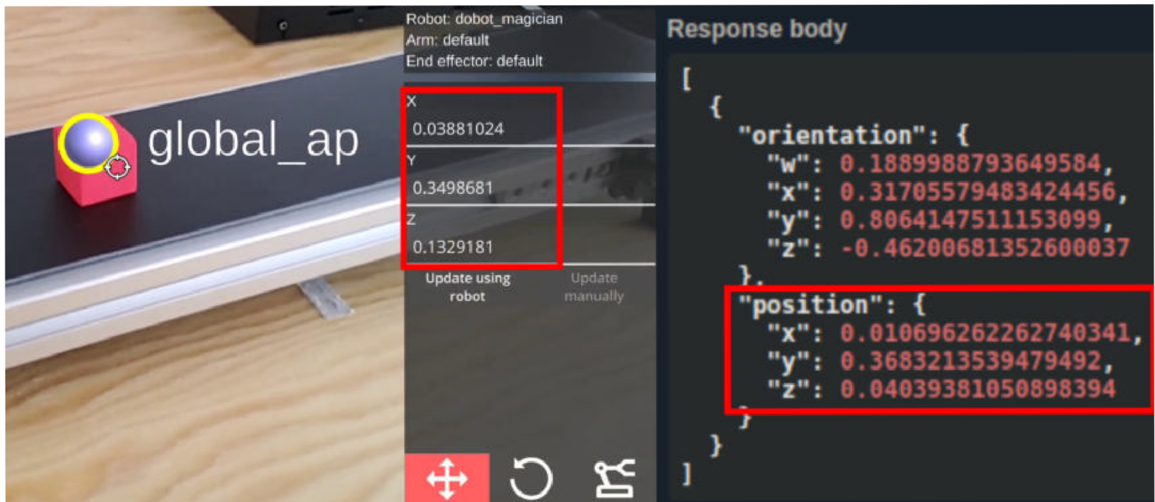
5.3 Experimenty na robotickém pracovišti

Zprovoznění detektoru a trackeru přímo na pracovišti bylo ze začátku dost komplikované. Při pokusech o zprovoznění byli objeveny chyby v kódu, kvůli kterým nefungoval nebo nešel spustit modul detektoru a trackeru. První problém byl, že instalační skript modulu (Dockerfile) neobsahoval potřebnou knihovnu. Další chyba pak byla v souboru pro sestavení Docker kontejnerů (docker-compose.yml). Pak už byli jen drobnější chyby v Object Type trackeru.

Na chybu se narazilo i v existující službě kamery Azure Kinect. Chyba byla v endpointu, který vracel synchronizovaný RGB-D obraz. Při lokálním vývoji byla tato chyba opravena. Současně s touto prací byla ale vyvíjena další bakalářská práce, která měla za úkol rozšířit právě tuto službu kamery. Služba kinectu byla v té práci kompletně předělána a byla přidána do hlavní větve systému během toho, co se testovala služba detektoru/trackeru. Zmíněná chyba se tak znovu objevila a ještě s dalšími drobnějšími chybami se ještě jednou opravila.

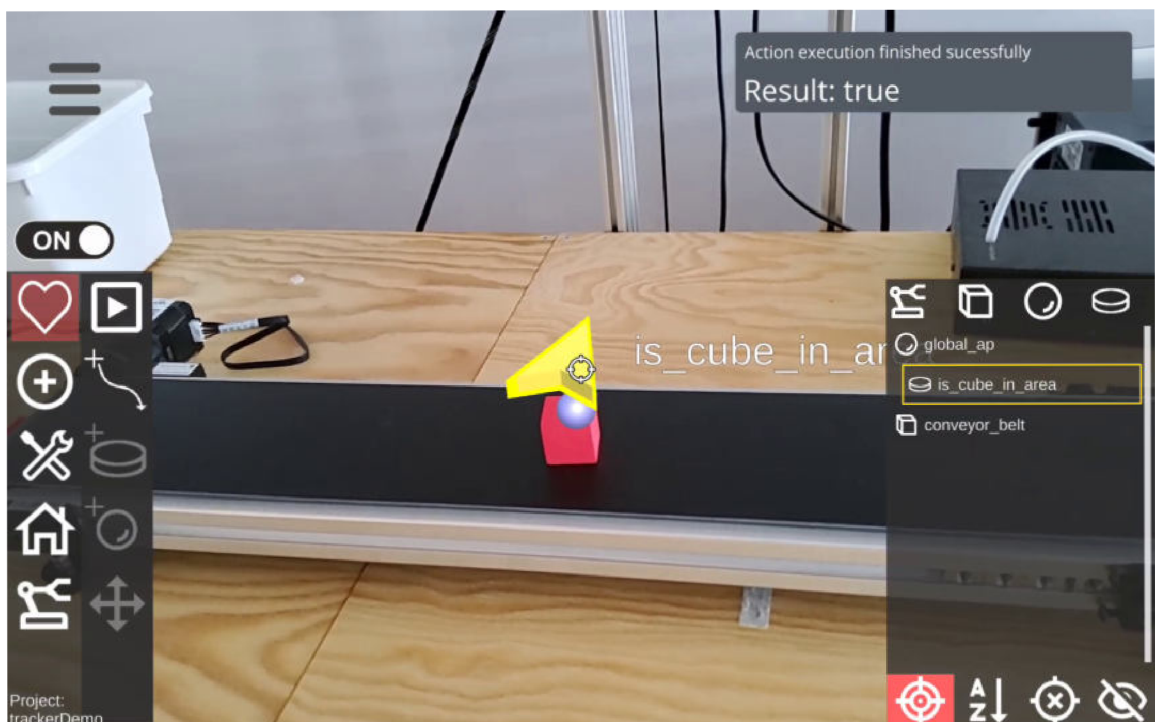
Poté, co se podařilo vše zprovoznit, mohlo začít pořádné testování. Jako první byla provedena kontrola toho, jestli detektor vrací správné pozice kostek. V rozšířené realitě byl manuálně vytvořen akční bod. Tento akční bod byl poté ručně posunut na místo, na kterém se nacházela červená kostka. Pozici akčního bodu nebylo potřeba nastavit přesně do středu kostky, stačilo když byl bod zhruba na stejném místě. Následně se ve swaggeru API služby trackeru získali pozice všech kostek. V tu dobu byla na scéně pouze jedna viditelná kostka. Pozice této detekované kostky se porovnala s pozicí akčního bodu (znázorněno na obrázku 5.6). Souřadnice kostky i bodu byly téměř stejné, malá odchylka byla způsobena nepřesnou

pozicí akčního bodu. Tento experiment kontroloval, jestli detektor správně převádí souřadnice kostek ze souřadnicového systému kamery do souřadnicového systému ARCOR2.

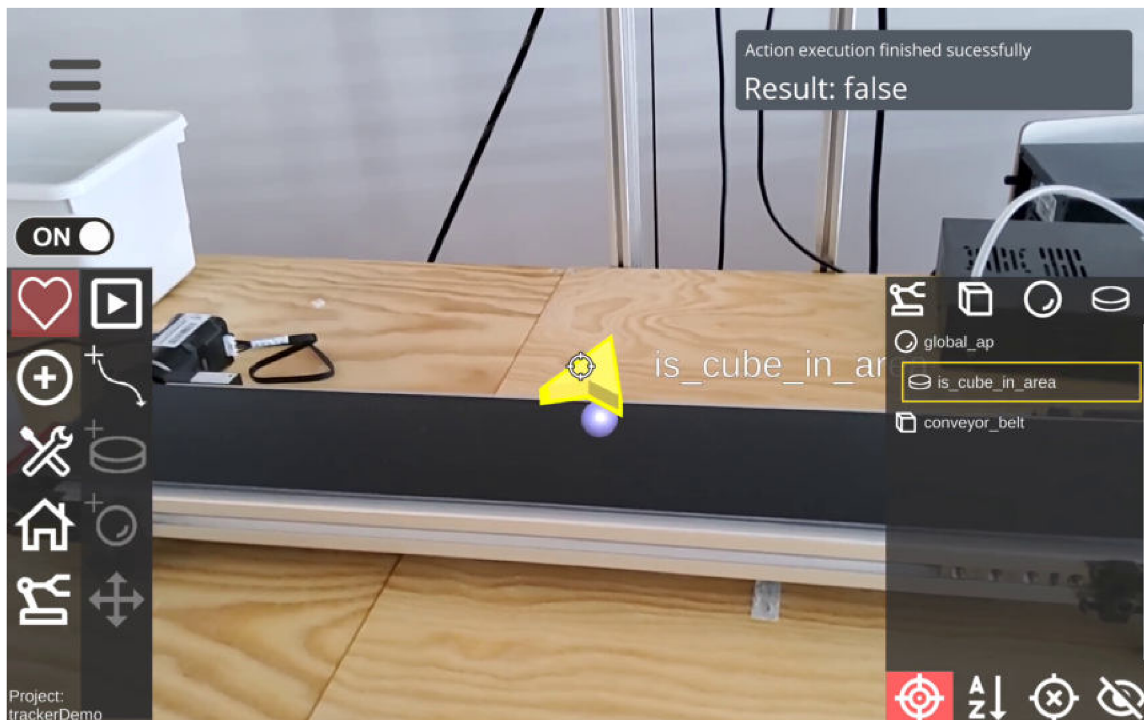


Obrázek 5.6: Porovnání pozice vytvořeného bodu a pozice kostky získaná přes API

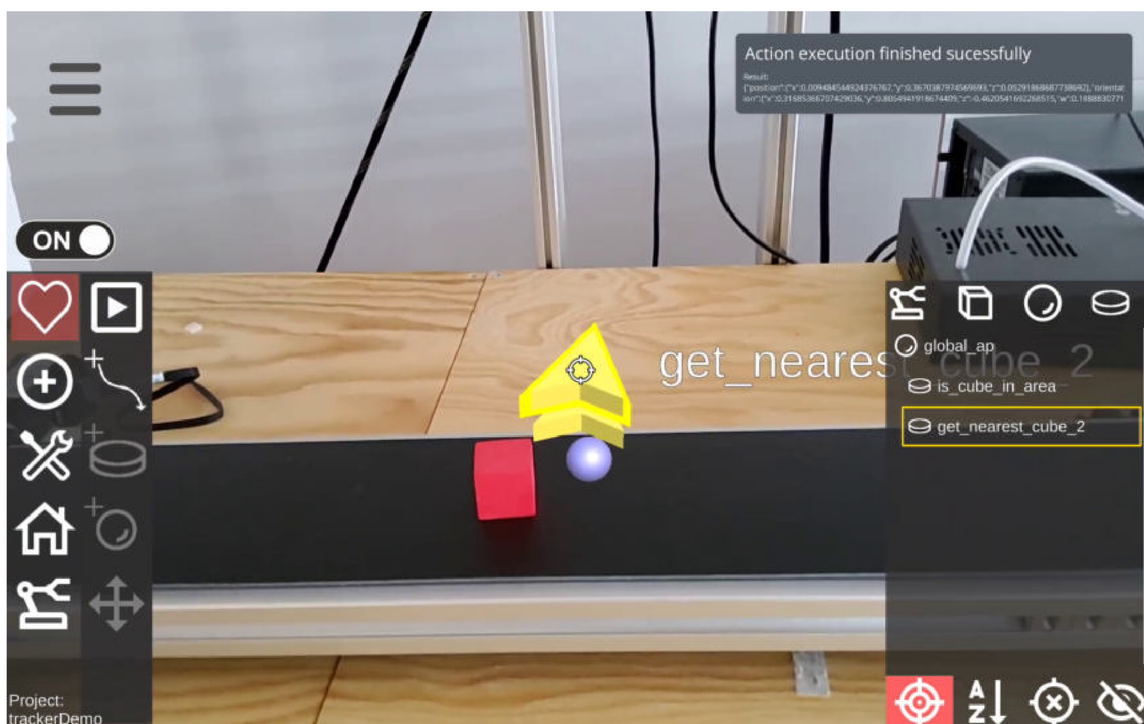
Další experimenty spočívali v jednoduchém testování OT trackeru. Cílem bylo testovat akci pro získání nejbližší kostky a akci pro zjištění, zda se kostka nachází v okolí akčního bodu. Byl vytvořen akční bod, ke kterému se přiřadili právě tyto akce. První se testovala akce `is_cube_in_area` s kostkou blízko bodu (obr. 5.7). Následovala stejná akce, ale nyní bez kostky (obr. 5.8). Kostka se pak umístila zpátky a provedla se akce `get_nearest_cube` (obr. 5.9).



Obrázek 5.7: Kostka se nachází blízko bodu, akce vrací `true`



Obrázek 5.8: Kostka se nenachází blízko bodu, akce vrací `false`

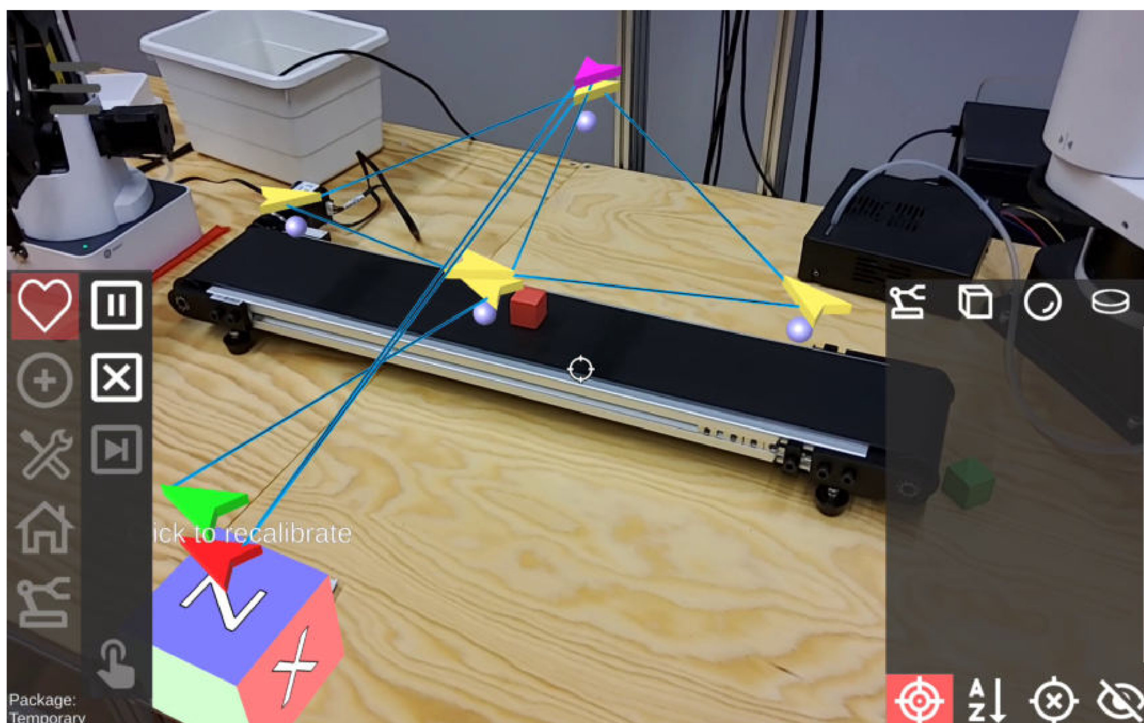


Obrázek 5.9: Akce vrací pozici kostky

V plánu byly složitější experimenty s roboty, kteří by například třídili detekované kostky podle barev na různé místa. V době testování byla bohužel v uživatelské aplikaci AREditor

chyba, kvůli které nebyly správně zobrazovány modely robotů. Nebylo tak možné nastavit robotovi správnou pozici ve scéně. Pokud není pozice správně nastavená, tak nemůže robot spolehlivě pracovat s pozicemi detekovaných kostek, jelikož pro něj budou tyto pozice na odlišném místě, než ve skutečnosti jsou. Tento problém nevadí, pokud se vytváří akční body za pomocí pozice robotického ramene. Tyto pozice jsou pro robota relevantní, takže se mezi nimi může pohybovat. Pozice kostek jsou ale absolutní ke scéně a špatně nastavený robot s nimi tak nemůže pracovat.

Složitější experimenty tak šli dělat pouze s dopravníkovým pásem. Byl vytvořen program, který testoval, zda se na dopravníkovém pásu nachází kostka. Na střed pásu byl vložen akční bod s akcí `is_cube_in_area`. Tato akce byla vytvořena dvakrát, jednou pro kontrolu červené kostky a jednou pro kontrolu zelené kostky. Jako první se kontroluje červená kostka, pokud se nachází u bodu, tak se přejde do akce, která posune pás o určenou vzdálenost doleva, V opačném případě se kontroluje zelená kostka. U té se podobně přejde do akce ovládající pás, v tomto případě se ale pás posune doprava. Pokud není v blízkosti žádná kostka, tak se program na 2 sekundy uspí a celá smyčka se znovu opakuje. Program se na pár sekund uspí i po posunutí pásu. Toto uspání umožní dokončení akce posunutí pásu a zároveň umožní aktualizaci kostky, která se na pásu posunula. Tento experiment fungoval podle představ a kostky se skutečně na pásu třídili. Tento progran je zobrazen na obr. 5.10



Obrázek 5.10: Program třídící kostky podle barvy

Kapitola 6

Závěr

Cílem této práce bylo rozšířit systém ARCOR2 o modul umožňující detekci barevných kostek. Implementace byla rozdělena do dvou samostatných modulů – modul pro detektor, který pouze detekuje kostky a modul pro tracker, který periodicky získává detekované kostky z detektoru a ukládá je. Zároveň byl vytvořen nový Object Type, který rozšiřuje funkcionalitu systému ARCOR2. Tento OT pracuje s API službou trackeru a umožňuje ověřit, zda se na určité pozici nachází kostka a případně získat pozici nejbližší nebo nejvzdálenější kostky.

Samotná detekce kostek je založena na zpracovávání mračna bodů, kdy se napřed obraz vyfiltruje podle barev, poté se rozdělí na shluky, které se dále segmentují na roviny. Tyto roviny se pak kontrolují a určuje se, zda tvoří kostku.

Pro testování byl vytvořen vlastní dataset obsahující 45 snímků. Pro všechny snímky se anotovala pozice každé kostky a následně byl dataset vyhodnocen pomocí skriptu. Výsledky ukazují 91% úspěšnost detekce s minimálním počtem chybné detekce.

Další testování proběhlo v robotické laboratoři, kde byly naživo otestovány oba moduly i nový Object Type. Bylo provedeno pár primitivních i složitějších experimentů, kdy například dopravníkový pás posouval kostky na levou nebo na pravou stranu podle barvy detekované kostky.

Řešení má určitě prostor pro zlepšení. Aktuálně detektor hledá pouze polohu kostky, nicméně v budoucnu by bylo vhodné rozšířit algoritmus také o detekci orientace kostky. Dále by bylo možné více optimalizovat rychlost detekce.

Literatura

[1] ALVES, J. *Overview of depth cameras* [online]. Aivero. Revidováno 16. 11. 2021 [cit. 2023-04-15]. Dostupné z: https://aivero.com/topic/overview-of-depth-cameras/.

[2] BRADSKI, G. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*. 2000.

[3] CHAUHAN, N. S. DBSCAN clustering algorithm in machine learning. *KDnuggets* [online]. 4. dubna 2022 [cit. 2023-05-15]. Dostupné z: https://www.kdnuggets.com/2020/04/dbscan-clustering-algorithm-machine-learning.html.

[4] ESTER, M., KRIEGEL, H. P., SANDER, J. a XIAOWEI, X. A density-based algorithm for discovering clusters in large spatial databases with noise. Prosinec 1996. Dostupné z: https://www.osti.gov/biblio/421283.

[5] FISCHLER, M. A. a BOLLES, R. C. Random Sample Consensus: A Paradigm for Model Fitting with Applications to Image Analysis and Automated Cartography. *Commun. ACM*. New York, NY, USA: Association for Computing Machinery. jun 1981, sv. 24, č. 6, s. 381–395. DOI: 10.1145/358669.358692. ISSN 0001-0782. Dostupné z: https://doi.org/10.1145/358669.358692.

[6] KAPINUS, M., BERAN, V., MATERNA, Z. a BAMBUSEK, D. Spatially Situated End-User Robot Programming in Augmented Reality. In:. říjen 2019, s. 1–8. DOI: 10.1109/RO-MAN46459.2019.8956336.

[7] KUNT, L. *Detekce objektů z hloubkové kamery* [online]. Praha, 2020. [cit. 2023-05-15]. Bakalářská práce. České vysoké učení technické v Praze, Fakulta elektrotechnická. Vedoucí práce PETR, S. Dostupné z: https://dspace.cvut.cz/handle/10467/87713.

[8] MICROSOFT. *Azure Kinect DK* [online]. [cit. 2023-04-09]. Dostupné z: https://azure.microsoft.com/en-us/products/kinect-dk.

[9] SHARMA, A. Angle between two planes in 3D. *GeeksforGeeks* [online]. Revidováno 16. 9. 2022 [cit. 2023-05-15]. Dostupné z: https://www.geeksforgeeks.org/angle-between-two-planes-in-3d/.

[10] WANG, Y., WANG, C., LONG, P., GU, Y. a LI, W. Recent advances in 3D object detection based on RGB-D: A survey. *Displays*. 2021, sv. 70, s. 102077. DOI: https://doi.org/10.1016/j.displa.2021.102077. ISSN 0141-9382. Dostupné z: https://www.sciencedirect.com/science/article/pii/S0141938221000846.

[11] WARD, I., LAGA, H. a BENNAMOUN, M. RGB-D Image-Based Object Detection: From Traditional Methods to Deep Learning Techniques. In:. říjen 2019, s. 169–201. DOI: 10.1007/978-3-030-28603-3_8. ISBN 978-3-030-28602-6.

[12] ZHOU, Q.-Y., PARK, J. a KOLTUN, V. Open3D: A Modern Library for 3D Data Processing. *ArXiv:1801.09847*. 2018.

Příloha A

Obsah paměťového média

Přiloženy jsou pouze zdrojové soubory modulů, které byly v této práci vytvořeny. Dále je přiložen soubor **git.diff**, který obsahuje všechny provedené změny. Celý zdrojový kód systému ARCOR2 je dostupný na GitHubu[1].

```
/
├── text.pdf ............................ textová část práce
├── latex/ .............................. zdrojové soubory k textové části
├── src ................................. zdrojové soubory modulů
│   ├── docker/
│   │   ├── arcor2_cube_detector/
│   │   └── arcor2_cube_tracker/
│   └── python/
│       ├── arcor2_cube_detector/
│       ├── arcor2_cube_tracker/
│       └── arcor2_fit_demo/
│           └── object_types/
│               └── cube_tracker.py
├── git.diff ............................ provedené změny
└── video.mp4 ........................... video prezentující tuto práci
```

[1] https://github.com/robofit/arcor2