



TANSZÉKVEZETŐ

SZAKDOLGOZAT FELADAT

Makay Petra

szigorló mérnökinformatikus hallgató részére

Kamerával felszerelt robotkar távoli vezérlése

A hallgató feladata egy kamerával felszerelt ipari robotkar távoli vezérlésének megvalósítása és működésének bemutatása egy konkrét alkalmazás implementálásával. A feladat magában foglalja a robotkar mozgásának irányítását egy távoli hálózati kapcsolat segítségével, amely lehetővé teszi a valós idejű képalkotást és a vizuális adatok alapján történő precíz mozgásvezérlést.

A robotnak képesnek kell lennie egy rajzolt ábra felismerésére és kiegészítésére (pl. színezés). A gépi látás funkció a robot megfogójára szerelt kamera segítségével oldandó meg.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be a laborban elérhető UR3e robotkar és Intel RealSense D405 kamera képességeit.
- Hozza létre és mutassa be azt a hálózati környezetet, amelyben a robot vezérlése és a kamera képfeldolgozás funkciói egy távoli (hálózati) eszközön valósul meg.
- Tervezzen és valósítson meg egy alkalmazást a rendelkezésre álló eszközök segítségével, ahol a robotkar képes felismerni majd beszínezni egy papírlapon elé tett sokszög vonalábrát.
- A megvalósított rendszerrel végezzen teszteket, melyek során határozza meg a megoldás képességeit és korlátait.
- Munkáját részletesen dokumentálja.

Tanszéki konzulens: Dr. Vidács Attila egyetemi docens

Külső konzulens:

Budapest, 2024. október 2.

Dr. Varga Pál
tanszékvezető

Budapesti Műszaki és Gazdaság tudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Mesterséges Intelligencia Tanszék



1117 Budapest, Magyar tudósok krt. 2.
Tel.: +36 1 463 2448 – Web: <https://www.tmit.bme.hu>
E-mail: pvarga@tmit.bme.hu



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar
Távközlési és Mesterséges Intelligencia Tanszék

Makay Petra

**KAMERÁVAL FELSZERELT
ROBOTKAR TÁVOLI VEZÉRLÉSE**

KONZULENS

Dr. Vidács Attila

BUDAPEST, 2024

Tartalomjegyzék

Összefoglaló	5
Abstract	6
1 Bevezetés	7
1.1 UR3e	7
1.1.1 Robotkar fizikai jellemzői	8
1.1.2 Teach Pendant.....	8
1.1.3 Tool Center Point (TCP).....	9
1.1.4 Robotok koordináta rendszerei	10
1.2 OnRobot RG2FT gripper	10
1.3 RealSense D405 kamera	11
1.3.1 RealSense D405 jellemzői	11
1.3.2 Szoftvertámogatás.....	12
1.4 OpenCV	13
2 Fejezet	15
2.1 UR3e robotkar távvezérlésének lehetőségei	15
2.1.1 TCP/IP socket	15
2.1.2 Dashboard szerver	15
2.1.3 ROS (Robot Operating System)	16
2.2 Távvezérlés megvalósítása	17
2.2.1 Real-Time Data Exchange (RTDE) interfész	18
2.2.2 RTDE működése.....	18
2.2.3 RTDE konfigurálása Pythonban	18
2.2.3.1 SocketTest.....	20
2.2.4 Miért az RTDE?.....	21
3 Fejezet	22
3.1 RealSense USB-vel.....	22
3.1.1 Python implementáció	22
3.1.2 Eredmény és további lehetőségek.....	24
3.2 OpenCV GStreamer	25
3.3 Távoli megjelenítés megvalósítása	26
3.3.1 Technikai tudnivalók, kezdeti nehézségek	26

3.3.2 Kliens oldal	27
3.3.2.1 GStreamer integrálása OpenCV-be	29
3.3.3 Szerver oldal	31
3.3.4 Stream működése.....	32
4 Fejezet	36
4.1 Négyszögek felismerése	36
4.2 Színezés megvalósítása.....	39
4.2.1 Színezés koncepciója és megvalósítása	39
4.2.2 Koordináták átszámítása	40
4.2.3 Működés	42
5 Fejezet	45
5.1 Tesztelés.....	45
5.2 Továbbfejlesztési lehetőségek	47
6 Fejezet	48
Irodalomjegyzék	49
Függelék.....	50

HALLGATÓI NYILATKOZAT

Alulírott **Makay Petra**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettettem, egyértelműen, a forrás megadásával megjelöltetem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervezet esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2024. 12. 05.

.....
Makay Petra

Összefoglaló

A mai világban egyre nagyobb szerepet kapnak a különböző, mesterséges intelligenciával felruházott, önállóan cselekedni képes robotok. Bár elterjedésük a hétköznapi életben még meglehetősen korlátozott, ipari felhasználásuk széles körű, számos tevékenység elvégzésére alkalmazzák őket.

Az informatikai ipar folyamatos fejlődésével lehetőség van ezen robotok működését számos szenzor és kamera segítségével kiegészíteni, így akár képessé válnak a környezet változásaihoz alkalmazkodni és önálló döntéseket hozni. Egyre többet találkozhatunk a képfeldolgozás és az érzékelés különböző alkalmazásaival, az önvezető járművektől kezdődően, a gyárakban használt robotokon át, egészen az orvosi képalkotó rendszerekig és az okoseszközökben alkalmazott arcfelismerő technológiáig. Népszerűségüket növeli, hogy a hálózatok fejlődésének köszönhetően lehetséges vált ezen robotok távolról való vezérlése, így olyan területeken is megállják a helyüket, ahol nincs lehetőség közvetlen utasításokat adni nekik.

Szakdolgozatom témája ötvözi a hálózati megoldások használatát és a képfelismerés technológiáit egy robotkar által végzett funkció megvalósítása esetén. Alapját egy UR3e típusú robotkar képezi, mely fel van szerelve az Intel RealSense D405-ös kamerájával. Feladataim egy olyan alkalmazás készítése, mely segítségével 5G-s hálózaton keresztül a robotkar feldolgozza az elé tett papírlap képét, majd ezt követően a rajta található sokszögeket valamilyen módszer alkalmazásával kiszínezi.

Ennek megvalósítása során a kamera által érzékelt képet távolról, egy Python programot futtató eszközzel feldolgozom OpenCV segítségével, majd szintén távvezérlés megvalósításával a robotkart odamozdítom a felismert sokszögek fölé. Ezt követően a robotkar valamilyen meghatározott módszer alapján kiszínezi az alakzatokat. Fontos, hogy a robotkar kezdőpozíciójából az előtte lévő A4-es papír teljes egészében látható legyen, majd a feladat elvégzését követően visszakerüljön a kezdőpozícióba, esetleges további feladatok elvégzése céljából.

Abstract

Artificial intelligence-driven robots with autonomous capabilities are playing an ever-growing role in contemporary society. Although their spread in everyday life is still quite limited, their industrial applications are widespread, and they are used for carrying out numerous activities.

With the continuous development of the IT industry, it is possible to enhance the operation of these robots with several sensors and cameras, enabling them to adapt to environmental changes and make autonomous decisions. We increasingly encounter various applications of image processing and sensing, ranging from self-driving vehicles, through robots used in factories, to medical imaging systems and facial recognition technologies used in smart devices. Their popularity is enhanced by the fact that, thanks to the development of networks, it has become possible to control these robots remotely, allowing them to perform well even in areas where there is no possibility to give them direct instructions.

My work combines the use of networking solutions and image recognition technologies in implementing a function performed by a robotic arm. It is based on a UR3e robotic arm equipped with Intel's RealSense D405 camera. My task is to develop an application that enables the robotic arm to process the image of a paper placed in front of it through a 5G network, and afterwards color the polygons found on it using a specific method.

During implementation, I process the image captured by the camera remotely using a Python program with OpenCV, then through remote control, I move the robotic arm above the detected polygons. Then the robotic arm colors the shapes according to a predetermined method. It is essential that the A4 paper placed in front of the robotic arm is fully visible from its home position, and that the arm returns to this position after completing the task, enabling potential further operations.

1 Bevezetés

Elméleti összefoglaló

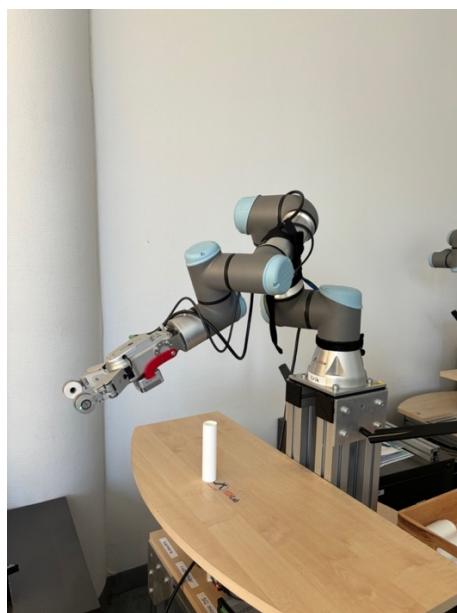
Az alábbi fejezetben a szakdolgozatom elkészítéséhez alkalmazott eszközök, valamint a gyakran használt fogalmak és fontosabb elméleti tudnivalók kerülnek bemutatásra.

1.1 UR3e

Feladatom elvégzéséhez a Universal Robots [1] által gyártott UR3e típusú robotkart használtam, mely elsősorban kisebb automatizálási feladatok elvégzésére alkalmazott robotkar. Kompakt alakú kobot, melynek köszönhetően könnyen illeszkedik szűk munkaterekbe, kisméretű munkaállomásokba.

Számos fejlett funkcióval rendelkezik, melyek lehetővé teszik a pontos, hatékony, biztonságos működést emberi munkaerővel és automatizált rendszerekkel egyaránt. Kollaboratív jellegéből adódik, hogy használatához nem kell bonyolult biztonsági intézkedéseket foganatosítani, úgy terveztek, hogy megfeleljen számos erre vonatkozó előírásnak.

A rendelkezésemre álló robotkar az egyetemen az Erik nevet kapta, mely az 1.1-es ábrán látható.



1.1. ábra – Laborban található Erik nevű robotkar

1.1.1 Robotkar fizikai jellemzői

A UR3e egy 6 szabadsági fokkal rendelkező robotkar. Ennek lényege, hogy a robot 6 olyan pontból áll, melyek 360 fokban tudnak elfordulni, így nagy mozgathatóságot tesznek lehetővé. Ezeket más néven szervomotoroknak, vagy jointoknak is nevezzük. Maga a robotkar könnyű, illetve méretét tekintve is a legkisebb UR kobot címét viseli, ebből adódóan teherbírása korlátozott, maximum 3kg. Kinyúlása 500 mm, ami lehetővé teszi, hogy kisebb munkaterületeken is közel hibamentesen és precízen tudjon működni.

1.1.2 Teach Pendant

A UR Teach Pendant egy felhasználóbarát interfész, melyet a UR3e, UR5e és UR10e kollaboratív robotok programozására és irányítására terveztek. Gyakorlatilag egy érintőképernyővel rendelkező táblagép, ami egy kábelén keresztül kapcsolódik a robot vezérlőjéhez.

Az UR3e robot mozgatása alapvetően két módon történhet: a Teach Pendant segítségével lokálisan, vagy egy távolról csatlakoztatott eszközön keresztül. A robotkart pusztán fizikai erő alkalmazásával - biztonsági okokból - nem lehet elmozdítani, ez az ipari robotok egyik alapvető védelmi funkciója. A kézi mozgatás lehetővé tételeért a Teach Pendant hátoldalán található engedélyező kapcsoló (enabling device) felel. Ez egy rugós kialakítású biztonsági gomb, amelynek folyamatos nyomva tartása mellett a robotkar szabadon, kézzel is mozgathatóvá válik. Ez a megoldás biztosítja, hogy a robot pozíciójának megváltoztatása kizárolag a kezelő tudatos irányítása mellett történhessen. Feladatom során többször is alkalmaztam ezt a funkciót, például amikor megkerestem azt a pozíciót, ahonnan a robotra szerelt kamera az egész papírlapot látja.

A Teach Pendant-on futó grafikus szoftver neve PolyScope [2]. Ennek két fajtája létezik jelenleg: a PolyScope 5 és a PolyScope X. A laborban található UR3e robotkar Teach Pendant-jén a PolyScope 5.16-os verziója érhető el. Az 1.1.2-es ábrán látható a Teach Pendant-on futó PolyScope szoftver.



1.1.2. ábra – PolyScope a Teach Pendant-on

Számos hasznos funkciója közül fontos megemlíteni azt, melyet dolgozatom elkészítése során alkalmaztam. A PolyScope felületén lehetőség van beállítani, hogy a robotot a Teach Pendant segítségével lokálisan, vagy egy másik eszköz használatával, távolról kívánjuk vezérelni. Ezt a képernyő jobb felső sarkában egy ikonra kattintva tudjuk állítani, ahol név szerint a 'Local' és a 'Remote Control' lehetőségek között tudunk váltani. Az én esetbenben, mivel távolról, a saját eszközöm segítségével irányítom a robotkart, így a program futása során végig 'Remote Control' módban kell lennie.

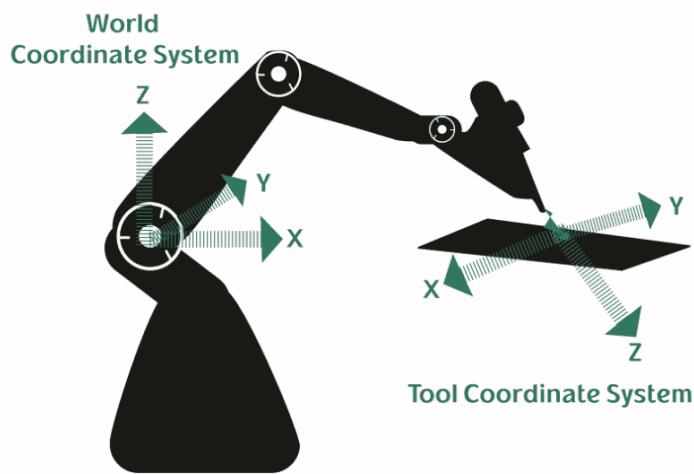
1.1.3 Tool Center Point (TCP)

A Tool Center Point (eszköz központ) a robotkar végén található eszköz vagy szerszám képzeletbeli pontja, amelyet a robot vezérlőrendszere a mozgások koordinálásához használ. Ez a pont általában ott helyezkedik el, ahol a tényleges interakció zajlik, például ahol a megfogó eszköz megragad egy tárgyat, vagy egy hegesztőfej dolgozik.

A TCP rendkívül fontos a robotok programozásában és vezérlésében, mivel a robot mozgása ezen a ponton alapul. A robotkar által végrehajtott pozícióváltozások, elmozdulások és forgások mindenkor a TCP-hez viszonyítva történnek. Amikor egy robotot programoznak, a mozgási utasításokat gyakran úgy adják meg, hogy a TCP-nek mely pozícióba vagy orientációba kell kerülnie. A TCP pontos meghatározása és kalibrálása elengedhetetlen a robot precíz működéséhez, különösen olyan alkalmazásoknál, ahol kis toleranciák vannak, mint például a hegesztés, szerelés vagy akár festés, rajzolás.

1.1.4 Robotok koordináta rendszerei

Egy robot számos koordináta rendszerrel rendelkezik, mely alapján a pozíóját, illetve a mozgását meg lehet határozni, be lehet állítani. A két alapvető rendszer, amely létezik, a robot saját koordináta rendszere, valamint a TCP koordináta rendszer, melyek az 1.1.4-es ábrán láthatóak.



1.1.4. ábra – Koordináta rendszerek

A robot mozgását mindenkor a robot koordináta rendszerhez viszonyítva irányítják, de a szerszám (TCP) helyzete és orientációja kritikus a feladatok végrehajtásában. Amikor a robot egy bizonyos feladatot végez, a vezérlőrendszer kiszámítja, hogyan kell a robotnak mozognia az alap koordináta rendszerben ahhoz, hogy a TCP a megfelelő pozícióba és orientációba kerüljön.

1.2 OnRobot RG2FT gripper

Az RG2FT az OnRobot egyik legnépszerűbb elektromos megfogója [3]. Társai közül sokoldalú felhasználhatóságával és előnyös tulajdonságaival tűnik ki, így a robotikában gyakran alkalmazzák.

A projekt megvalósítása során a gripper-t egy filctoll megfogására használtam. A megfogó beállítását a Teach Pendant-on keresztül végeztem el, ahol a PolyScope felületén a jobb felső sarokban található ikon segítségével lehetséges a megfogó ujjai közötti távolság beállítása. Az 1.2-es ábrán látható az általam használt robotkar, ujjai között a színezéshez használt filctollal.



1.2. ábra – Gripper, filccel az ujjai között

1.3 RealSense D405 kamera

Dolgozatom másik legfontosabb eszköze a RealSense D405 típusú kamerája volt. Feladatom megoldása során ezzel a kamerával kellett dolgoznom, így ebben az alfejezetben ezen eszköz tulajdonságairól, képességeiről és funkcióiról lesz szó.

1.3.1 RealSense D405 jellemzői

Az Intel RealSense D405 mélységérzékelő kamera a modern gépi látás és térbeli érzékelés egyik kiemelkedő technológiai megoldása, mely az 1.3.1-es ábrán látható [4].



1.3.1. ábra – RealSense D405 kamera

Ez a kompakt eszköz forradalmasítja a mélységérzékelés területét, különösen a rövid hatótávolságú alkalmazásokban, ahol a precizitás és a megbízhatóság kulcsfontosságú szerepet játszik.

A kamera ideális működési tartománya 7 cm és 50 cm közé tehető, ahol képes akár 1 mm-nél kisebb objektumok érzékelésére is. Felbontása 1280x800 pixel, ami lehetővé teszi a finom részletek megkülönböztetését és a magas színvonalú adatfeldolgozást. Mélységérzékelését infravörös fény használata is segíti, így a kamera alkalmazható sötétebb, vagy gyengén megvilágított körülmények között is.

Mélységérzékelés mellett tökéletesen használható színes képek feldolgozására is. Az RGB érzékelő felbontása 1920x1080 pixel, amely kiváló minőségű képeket biztosít, így részletgazdag látványt nyújt a színérzékelést igénylő alkalmazásokhoz.

A kamera rendelkezik egy modern USB 3.1 csatlakozóval, melynek két nagyon fontos funkciója van. Egyrészt ennek használatával nagyon gyors adatátvitelt biztosít, amely a valós idejű, vagy késleltetésérzékeny alkalmazások esetén rendkívül fontos, másrészt megoldja a kamera tápellátását is, így nincsen szüksége külső áramforrása.

A kamera kis méretének, tartósságának és kivételes energiahatékonyságának köszönhetően rendkívül sokoldalúan alkalmazható. Úgy terveztek, hogy ellenálljon a külső hatásoknak, így nemcsak ipari környezetben, hanem laboratóriumi alkalmazásokban is megbízható teljesítményt nyújt. Kompakt kialakítása lehetővé teszi, hogy könnyedén integrálható legyen akár hordozható eszközökbe is.

1.3.2 Szoftvertámogatás

Az Intel RealSense kamerák szoftveres támogatásának gerincét az Intel RealSense SDK 2.0 adja, amely egy nyílt forráskódú fejlesztőkészlet. A szoftver több operációs rendszeren is működik, beleértve a Windows 10-et és 11-et, különböző Linux disztribúciókat, Androidot, valamint korlátozottan a macOS-t is.

A fejlesztők számos programozási nyelvben dolgozhatnak a kamerákkal, mivel az SDK támogatja a C++, Python, C#/.NET, MatLab és LabVIEW nyelveket. Ezek az eszközök lehetővé teszik a mélységérzékelést és -feldolgozást, az RGB képfeldolgozást, valamint a pontfelhők kezelését is.

A RealSense kamerák jól integrálhatók népszerű fejlesztői keretrendszerekbe is. Ezek közé tartozik többek között a Robot Operating System (ROS) platform, ahol ezen

kamerák alkalmazása különösen előnyös robotikai alkalmazásokhoz. A RealSense-ROS csomag lehetővé teszi a kamera mélységi adatainak közvetlen felhasználását a robotok navigációjához, objektumok felismeréséhez és környezeti térképezéshez. Ezen ROS támogatás kiterjed mind a ROS 1, mind a ROS 2 verziókra. Másik nagyon hasznos platform az OpenCV könyvtár, mellyel való együttműködés esetén a fejlesztők komplex képfeldolgozási műveleteket végezhetnek. A RealSense által szolgáltatott RGB és mélységi képek közvetlenül feldolgozhatók az OpenCV függvényeivel, ami lehetővé teszi például a fejlett objektumdetekciót, képszegmentálást vagy akár a gépi tanuláson alapuló képelemzést. Harmadik keretrendszer a Unity játékmotorban való támogatottság. Különösen hasznos kiterjesztett és virtuális valóság alkalmazások fejlesztéséhez. A RealSense Unity csomagja tartalmaz előre elkészített komponenseket és szkripteket, amelyekkel könnyen megvalósítható például a valós környezet 3D szkennelése vagy a felhasználó mozgásának követése.

A GitHub platformon található dokumentáció és példakódok minden említett keretrendszerhez részletes útmutatót és gyakorlati példákat tartalmaznak, beleértve a telepítési folyamatot, az alapvető használatot és a haladó funkciókat is. Az Intel rendszeresen frissíti ezeket az anyagokat az új funkciók és a hibajavítások követésével.

1.4 OpenCV

A dolgozatom egyik lényegi eleme a négyzetek felismerése és detektálása, melyhez az OpenCV-t alkalmazom [5]. Az OpenCV (Open Source Computer Vision Library) egy nyílt forráskódú számítógépes látást és gépi tanulást támogató programkönyvtár, mely 2000-ben indult az Intel kezdeményezésére. C++, Python, Java és MATLAB nyelveken is használható, és számos platformot támogat, beleértve a Windows, Linux, Android és iOS rendszereket is. Rengeteg funkcióval és képességgel rendelkezik, melyeknek köszönhetően napjainkra széles körben elterjedt, világszerte használják kutatók, fejlesztők és vállalatok.

Főbb funkciói közé tartozik a képfeldolgozás, amely lehetővé tesz éldetektálást, képszűrést, valamint transzformációkat. Támogatja az objektumkövetést és az objektumfelismerést, ráadásul képes mindezt valós időben is végrehajtani. Alkalmas több kamera nézőpontjából készített képek alapján 3D objektumok rekonstrukciójára és mélységinformációk feldolgozására, valamint a gépi tanulást segítő modulja számos

algoritmust tartalmaz (neurális hálók, döntési fák), melyek támogatják a gépi tanulás alapú képfeldolgozást és objektumfelismerést.

Mivel az OpenCV erős támogatással rendelkezik a valós idejű képfeldolgozásban, így számos területen elterjedt a használata, például orvosi képfeldolgozásban, robotikában és autonóm járművek esetén. Ráadásul könnyen kombinálható más könyvtákkal is, ezért összetett látási feladatokhoz is alkalmazható.

Erősségei közé tartozik, hogy ingyenes a használata és aktív közösséggel rendelkezik, ezért folyamatosan fejlődik, lépést tartva a számítógépes látás legújabb trendjeivel és igényeivel.

A robotika területén is előszeretettel alkalmazzák, mivel az OpenCV által nyújtott képességek segítenek abban, hogy a robotok vizuális információk alapján hatékonyan érzékeljék, felismerjék és kövessék a környezetükben lévő tárgyakat, valamint navigáljanak az adott térben.

2 Fejezet

Robot távoli vezérlése

Ebben a fejezetben a robotkar távolról való eléréséről és vezérléséről lesz szó. Bemutatásra fognak kerülni a különböző lehetséges módszerek, majd részletesen fogom tárgyalni az általam választott megoldást.

2.1 UR3e robotkar távvezérlésének lehetőségei

A UR3e robotkarok távoli vezérlésére számos módszer áll rendelkezésre, amelyek lehetőséget nyújtanak a felhasználóknak arra, hogy különböző mértékben és rugalmassággal irányítsák a robotot [6]. A választás folyamatát különböző tényezők határozzák meg. Ilyenek például, hogy milyen gyors legyen a távoli interakció, a hálózati infrastruktúrának mekkora a kapacitása, az adott munkafolyamat mennyire komplex, valamint milyen biztonsági intézkedéseknek kell eleget tenni.

Három alapvető módszer létezik, melyekről az alábbiakban adok rövid áttekintést.

2.1.1 TCP/IP socket

A TCP/IP socket alapú kommunikáció az egyik leginkább elterjedt megoldás. Előnye, hogy gyors adatátvitelt tesz lehetővé, amely különösen fontos valós idejű alkalmazások esetén, ahol a robotkar azonnali reagálása kulcsfontosságú. Ez a mód lehetőséget ad arra, hogy saját alkalmazáson keresztül, C++ vagy Python segítségével irányítsuk a robotot. A feladatom megvalósítása során ezt a módszert alkalmaztam, melyet a 2.2-es fejezetben részletezlek.

2.1.2 Dashboard szerver

Az egyik legpraktikusabb és legegyszerűbb megoldás a Universal Robots Dashboard Server használata, amely egy alapvető, beépített szolgáltatás a UR robotok vezérlőin. Mivel ennek segítségével csak egyszerű műveleteket tudunk végrehajtani a robottal, mint programok elindítása és leállítása, a pozíció lekérdezése, vagy a szervomotorok működésének engedélyezése, ezért ez a módszer elsősorban felügyeleti rendszerek integrálásánál hasznos.

A Dashboard szerver a 29999-es porton található, mellyel közvetlenül a PolyScope-nak küldjük az utasításokat. Összetett programok írására nem alkalmas, csak a PolyScope-ra előre feltelepített, vagy azon készített scripteket képes elindítani és leállítani, így valós idejű alkalmazások készítésére nem ajánlott.

2.1.3 ROS (Robot Operating System)

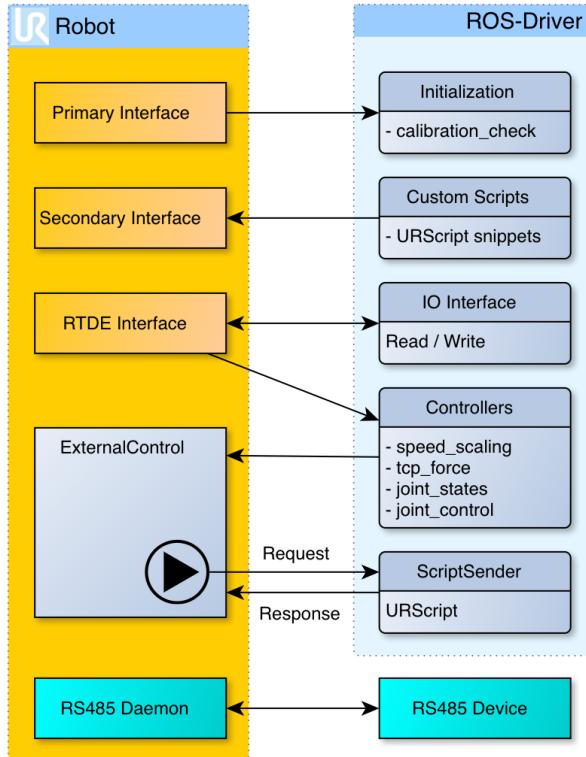
A Robot Operating System (ROS) egy nyílt forráskódú, moduláris keretrendszer, melyet robotikai alkalmazások fejlesztésére és üzemeltetésére terveztek [7]. A ROS egy központi kommunikációs architektúrát biztosít, amely megkönnyíti a különböző szoftverkomponensek (úgynevezett csomópontok) közötti együttműködést. A csomópontok egymástól független folyamatokként működnek, és képesek üzeneteket küldeni és fogadni egy közös kommunikációs rétegen keresztül.

A ROS egy decentralizált kommunikációs rendszert biztosít, ahol a ROS Master koordinálja a csomópontok (nodes) közötti adatcserét. A csomópontok szoftvermodulok, amelyek üzeneteket küldenek és fogadnak témákon (topics) keresztül, vagy szolgáltatások (services) segítségével kérés-válasz alapú interakciót folytatnak. Az üzenetek előre meghatározott formátumú adatcsomagok, amelyek a robot funkcióinak végrehajtását segítik.

A ROS hivatalosan támogatja a Python és a C++ nyelveket, melyekkel a csomópontok könnyen fejleszthetők.

A ROS gazdag ökoszisztémája számos előre definiált eszközt kínál a robotika fejlesztéséhez. A Gazebo valósághű szimulációs környezet, az RViz vizualizálja a robot állapotát és érzékelt adatait, míg a rosbag adatgyűjtésre és elemzésre szolgál. A MoveIt! egy fejlett mozgástervezési keretrendszer, amelyet robotkarok vezérlésére használnak.

A ROS egyik legnagyobb előnye, hogy a hálózaton kereszthü kommunikációt támogatja, így távvezérlés megvalósítására is alkalmas. A ROS keretrendszer a kommunikációhoz használja a fent említett megoldások mindegyikét, mely a 2.1.3-as ábrán látható.



2.1.3. ábra – ROS komponensek és az adatáramlás

A képen látható robot-architektúrában az RTDE (Real-Time Data Exchange) interfész a ROS-vezérlő és a robot közötti valós idejű kommunikációért felelős, mely TCP/IP alapú kommunikációt használ. Az RTDE interfész teszi lehetővé, hogy a ROS-vezérlő valós időben olvashassa és írhassa a robot állapotát és vezérlési paramétereit.

Emellett a diagramon nem látható, de a ROS-vezérlő a hálózati kommunikációhoz általában használ egy Dashboard szervert is. A Dashboard szerver lehetővé teszi a ROS-rendszer távoli elérését és monitorozását, így a robotot a hálózaton keresztül is lehet irányítani és felügyelni. A Dashboard szerver gyakran magába foglalja a robot állapotának megjelenítését, a parancsriadást, a szimulációt és más hasznos funkciókat, amelyek megkönnyítik a robot távoli kezelését.

2.2 Távvezérlés megvalósítása

A távvezérlés megvalósítása során megoldásom alapjaként az RTDE (Real-Time Data Exchange) interfész szolgált, melyet ebben a fejezetben részletesebben is bemutatok. Szó lesz arról, hogy milyen módon alkalmaztam, és a fejezet végén megmagyarázom, hogy miért erre esett a választásom.

2.2.1 Real-Time Data Exchange (RTDE) interfész

A Real-Time Data Exchange (RTDE) interfész a Universal Robots által biztosított protokoll, amely lehetővé teszi a UR robotkarok külső rendszerekkel történő valós idejű adatcseréjét. Az RTDE egy TCP/IP-alapú kommunikációs protokoll, amely optimalizálva van a gyors és megbízható adatátvitelhez, és lehetőséget nyújt arra, hogy a robot állapotát, szenzoradatait és egyéb fontos információit gyorsan megoszthassuk külső alkalmazásokkal. Ezeken kívül támogatja, hogy parancsokat küldjünk a robotnak, így a robotvezérlés is megoldható a használatával.

2.2.2 RTDE működése

Az RTDE interfész segítségével kétirányú kommunikáció valósítható meg, mely két fő folyamat köré épül [8]:

- Input Package: ezekkel a csomagokkal parancsokat küldhetünk a robotnak: beállíthatjuk a célpozíciót, sebességet, és egyéb vezérlési paramétereket. Az RTDE input csomagok lehetőséget adnak arra, hogy a robotot részletesen, precíz módon vezéreljük.
- Output Package: ezekkel a csomagokkal adatokat kérhetünk le a robotról, beleértve a jelenlegi pozíciót, sebességet, nyomatékadatokat és egyéb fontos diagnosztikai információkat. Az RTDE output csomagok rendszeresen küldik a kiválasztott adatokat a vezérlőből az alkalmazás felé, lehetővé téve a robot valós idejű állapotának folyamatos monitorozását.

Az RTDE interfészt általában a 30004 vagy 30005 porton keresztül érhetjük el. Használat előtt a robotkar Teach Pendant-jén engedélyezni kell a távoli vezérlést (Remote Control), amely ezután lehetővé teszi, hogy a megfelelő porton keresztül elérjük, és vezérelni tudjuk a robotkart, valamint információkat kapjunk a jelenlegi állapotáról.

2.2.3 RTDE konfigurálása Pythonban

Az RTDE interfész rendelkezik Python könyvtárakkal, így saját alkalmazásból is vezérelhetjük a robotot, a megfelelő paraméterek beállításával. Lehetőség van parancsokat adni, és adatokat fogadni a robottól az RTDE API segítségével [9].

Az alábbi kódrészlet bemutatja, hogy hogyan kapcsolódtam a robotkarra egy Python kóddal az RTDE könyvtárak használatával:

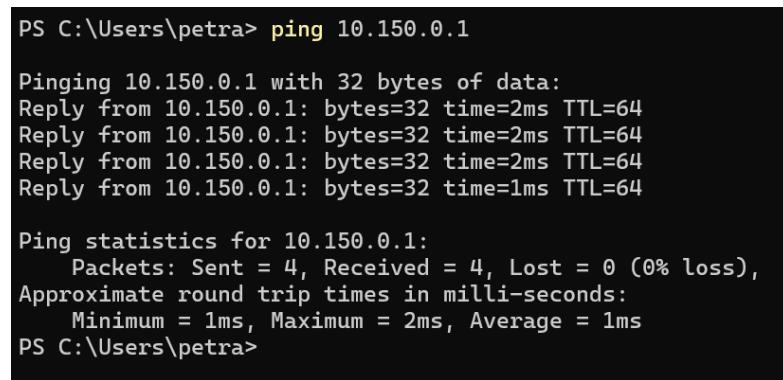
```
import rtde_control
import rtde_receive

robot_ip = '10.150.0.1'

rtde_c = rtde_control.RTDEControlInterface(robot_ip)
rtde_r = rtde_receive.RTDEReceiveInterface(robot_ip)
```

A kód elején levő importok felelnek az RTDE interfész használatáért. Az `rtde_control` könyvtár biztosítja a közvetlen vezérlést a robotkar felett, lehetővé téve annak mozgatását, valamint parancsok beállítását. Az `rtde_receive` szolgál arra, hogy valós időben információkat kérdezzünk le a robot állapotáról. Ezek együtt teszik lehetővé, hogy a robottal kétirányú kapcsolatot tudjak létesíteni. Ez a feladatom szempontjából különösen fontos, hiszen a robotkarnak dinamikusan kell reagálnia a kamera által érzékelte változókra.

Ahogy a fenti kódban látható, szükség van a robot IP címének megadására, e nélkül nem tudunk kapcsolódni a robotra. Fontos, hogy a robotkarnak és a távvezérlésért felelős eszközök (jelen esetben a laptopomnak) egy hálózaton kell lenniük, különben nem tudjuk az RTDE interfészt alkalmazni. A laborban a robotkar rendelkezik egy saját 5G Wifi hálózattal, így a laptopommal erre rákapcsolódva tudtam elérni a robotot. A legjobb ellenőrzési módszer arra, hogy az eszközünk látja-e a robot IP címét az, ha egy terminálból megpróbáljuk megpingelni a robotkart. Ahogy azt a 2.2.3-as ábra mutatja, nálam működött a ping, kaptam választ a robotkartól.



```
PS C:\Users\petra> ping 10.150.0.1

Pinging 10.150.0.1 with 32 bytes of data:
Reply from 10.150.0.1: bytes=32 time=2ms TTL=64
Reply from 10.150.0.1: bytes=32 time=2ms TTL=64
Reply from 10.150.0.1: bytes=32 time=2ms TTL=64
Reply from 10.150.0.1: bytes=32 time=1ms TTL=64

Ping statistics for 10.150.0.1:
    Packets: Sent = 4, Received = 4, Lost = 0 (0% loss),
    Approximate round trip times in milli-seconds:
        Minimum = 1ms, Maximum = 2ms, Average = 1ms
PS C:\Users\petra>
```

2.2.3. ábra – Robotkar pingelése

Végül a kódban beállítottam az rtde_c és rtde_r változókat, amelyek a robot IP címét használva kapcsolódnak a robotkarra. Ezek lesznek felelősek a tényleges kommunikáció lebonyolításáért a robotkar és a laptopom között.

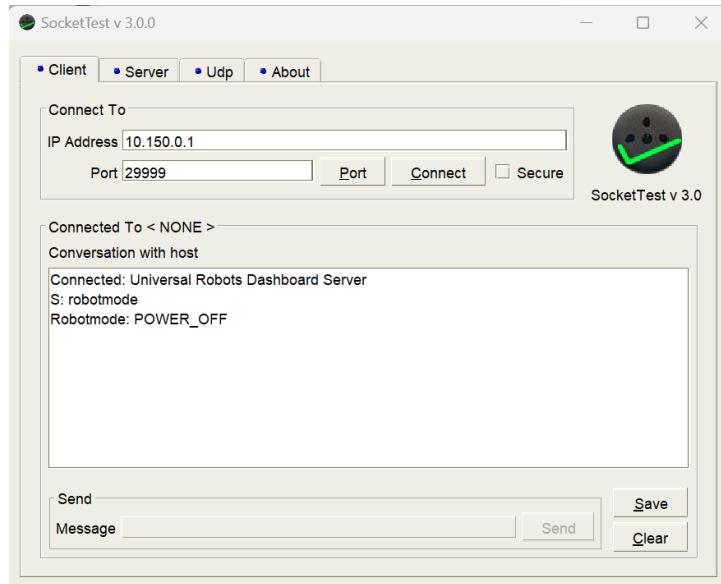
Felmerül a kérdés, hogy a port beállítására nincsen-e szükség, hiszen a Dashboard szervert csak annak megadását követően tudjuk elérni. Mivel a UR robotok gyárilag egy alapértelmezett portot használnak az RTDE kommunikációhoz, és maguk a Python könyvtárak is úgy vannak kialakítva, hogy ezt az alapértelmezett portot használják, így ezt a kódban nem fontos megadni.

Miután a kapcsolat létrejött, szabadon tudunk különböző parancsokat adni a robotkarnak, valamint az állapotáról is számos információt le tudunk kérdezni. Az alábbiakban bemutatom azt a két parancsot, amelyeket alkalmaztam a feladat megoldása során:

- moveL(position, a, v): ez a parancs felel a lineáris mozgásért. A position változóban kell megadni a célpozíciót [x, y, z, rx, ry, rz] formátumban. Lehetőség van a gyorsulás (a) és sebesség (v) értékek megadására is, azonban ez opcionális.
- getActualTCPPose(): ez a függvény a robot aktuális pozíójának és orientációjának lekérdezésére szolgál.

2.2.3.1 SocketTest

Mielőtt az RTDE interfész kipróbáltam volna, a 2.1.3-as fejezetben említett Dashboard szerver segítségével ellenőriztem, hogy a TCP/IP kommunikáció működik-e. Ehhez a SocketTest nevű alkalmazást használtam, amely bármilyen platformra telepíthető, és használata nagyon könnyű [10]. Csak a robot IP címét kell a megfelelő mezőbe beírni, majd a port szám megadását követően kiírja, hogy a kapcsolódás sikeres volt-e vagy sem. Én még lekértem a robot állapotát is, hogy lássam, a kommunikáció is működik-e. Ez látható a 2.2.3.1-es ábrán.



2.2.3.1. ábra – SocketTest használata

2.2.4 Miért az RTDE?

Választásomat erősen befolyásolta, hogy én egy Windows 11 operációs rendszert futtató laptopon készítettem a szakdolgozatomat. Mivel rengeteg, a robotikához köthető programot és alkalmazást csak Linux operációs rendszerekre fejlesztettek ki, így kellett találnom egy olyan megoldást, amellyel Windows esetén is tudok dolgozni. Sokáig próbálkoztam a WSL2 (Windows Subsystem for Linux 2) és a ROS kombinációjával, ahol a WSL segítségével Ubuntu-t futtattam, azonban mindenkor mindenkor hibába ütköztem, melyek kiküszöbölése sokáig elhúzódott volna, ezért más opciót kellett keresnem. Végül az RTDE interfész bizonyult a legkézenfekvőbb megoldásnak, mivel TCP/IP alapú kommunikációt használ, így teljesen független az azt használó operációs rendszertől.

A másik ok, amely a választásomat alátámasztja, az volt, hogy mindenkor mindenkor a kamera képénék feldolgozásához OpenCV-t szerettem volna alkalmazni, amely Python könyvtárral egyszerűen és könnyen használható. Így a szép megoldás érdekében a robotmozgatáshoz is valamilyen Python-nal kezelhető lehetőség után kutattam, és ahogy a 2.2.3-as fejezetből látszik, az RTDE interfész használatával lehet olyan Python kódot írni, amely képes kapcsolatot létesíteni a robottal és azon keresztül vezérelni azt.

3 Fejezet

Kamera távoli elérése

Ezen fejezet keretein belül fogom bemutatni, hogy a saját laptopomon milyen eszközök és programok segítségével jelenítettem meg távolról a robotkarra rögzített RealSense D405 típusú kamera képét. Részletesen tárgyalni fogom a megoldás menetét, meglemlítve a nehézségeket és kihívásokat, melyekbe ütköztem a folyamat során.

3.1 RealSense USB-vel

Mivel egy elég bonyolult feladatról van szó, melynek megvalósításával korábban még nem próbálkoztam, ezért első lépésként a képet a kamera USB csatlakozójának segítségével jelenítettem meg. Erre a lépésre azért volt szükség, hogy amennyiben a kamera távolról való elérése nem sikerül rövid időn belül, attól függetlenül a dolgozatom további részfeladatainak megoldásával tudjak foglalkozni.

Ahogy az 1.3.2-es fejezet leírja, a RealSense kamerák szoftvertámogatottsága meglehetősen széles körű. Az USB-s megoldás egyik alappillére a RealSense SDK 2.0 telepítése, melyet a megfelelő operációs rendszer kiválasztását követően a RealSense hivatalos oldaláról lehetőség van megtenni [11]. Mivel én Python alkalmazáson keresztül szerettem volna megjeleníteni a képet, hogy OpenCV segítségével fel tudjam azt dolgozni, így az SDK telepítése önmagában nem volt elég.

3.1.1 Python implementáció

Ahhoz, hogy a RealSense kamera képét megfelelően tudjuk kezelní egy Python program segítségével, ahhoz telepíteni kell a szükséges könyvtárakat. A program működéséhez az alábbi importok kellenek:

- pyrealsense2
- numpy
- cv2

A pyrealsense2 könyvtár szolgál a RealSense SDK interfészüként, amely gyakorlatilag lehetővé teszi a RealSense kamerák funkcionálisának kezelését és használatát Pythonban. Fontos, hogy helyes működésének egyik feltétele a RealSense

SDK 2.0 telepítése. Kizárálag ennek segítségével tudunk hozzáférni a kamera mélységerzékelési és színes képalkotási funkcióihoz. A NumPy a Python egyik legfontosabb könyvtára a numerikus és mátrixalapú számításokhoz. Lehetővé teszi a képadatok hatékony tárolását és manipulálását, ezért a kód egyszerűsége érdekében használata nélkülözhetetlen. Végül importálni kell magát az OpenCV-hez tartozó könyvtárat, mely a képek és videók megjelenítéséhez és a detektálási, felismerési feladatok ellátásához szükséges.

Fontos megemlíteni, hogy a pyrealsense2 könyvtárnak Python 3.7-es és 3.8-as verziója esetén van megfelelő támogatottsága. Mivel nálam a Python 3.12-es verziója futott, így a program működése érdekében ezt le kellett cserélnem a fent említett verziók egyikére.

Az importok megadását követően fel kell konfigurálni magát az alkalmazást, ami a kamera képének megjelenítéséhez elengedhetetlen lépés. Létre kell hozni a pipeline-t, ami a kamera felől érkező adatokat fogja fogadni, és az adatok áramlásáért lesz felelős. Ezt követően lehetőségünk van beállítani, hogy a képet milyen módon szeretnénk megjeleníteni (például fekete-fehérben vagy színesben), megadhatjuk a megjelenő ablak méreteit, majd ha ezt mind megtettük, végezetül el kell indítanunk magát a pipeline-t.

A megfelelő beállítások elvégzését követően egy végtelen ciklus indítása szükséges, amely folyamatosan lekéri és fogadja az adatokat a kamerától, majd megjeleníti az élő képet addig, amíg a program fut. Leállítást követően fontos, hogy az összes megjelenő OpenCV ablak be legyen zárva, valamint, hogy a pipeline le legyen állítva.

A kamera képének megjelenítéséhez egy példa program itt található:

```
import pyrealsense2 as rs
import numpy as np
import cv2

#Pipeline létrehozása
pipeline = rs.pipeline()

#Config beállítása
config = rs.config()

#A kamera indítása, színes képadatok kérése
config.enable_stream(rs.stream.color, 640, 480, rs.format.bgr8, 30)

#A pipeline indul
pipeline.start(config)
```

```

try:
    while True:
        #Egy képkocka Lekérése
        frames = pipeline.wait_for_frames()
        color_frame = frames.get_color_frame()

        if not color_frame:
            continue

        #Képkocka konvertálása numpy tömbbe
        color_image = np.asarray(color_frame.get_data())

        #Kép megjelenítése OpenCV-vel
        cv2.imshow('RealSense', color_image)

        #Kilépés a 'q' billentyűvel
        if cv2.waitKey(1) & 0xFF == ord('q'):
            break
finally:
    #Pipeline leállítása
    pipeline.stop()
    cv2.destroyAllWindows()

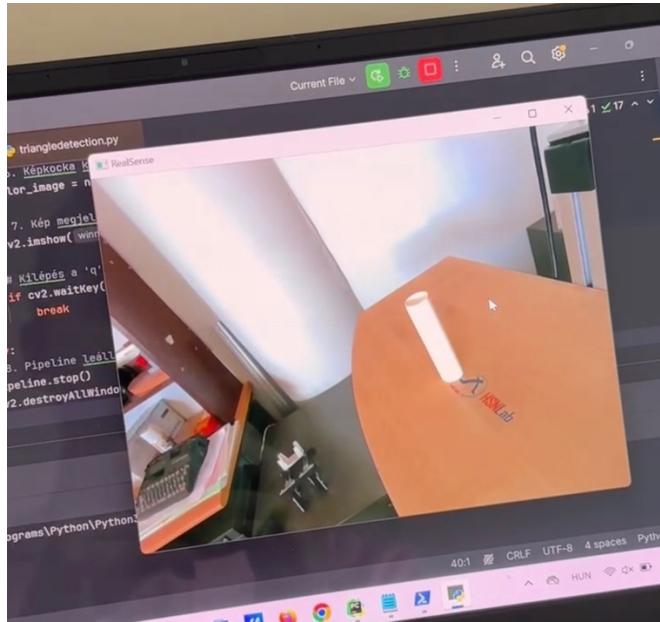
```

Ez a kód az eddig tárgyalt folyamatokat végrehajtja. Ebben az esetben pluszban beállításra került, hogy a 'q' billentyű lenyomására álljon le a program, illetve található benne egy 'if' ág is, ami ellenőrzi, hogy érkezik-e megfelelő adat a kamerától, amit meg lehet jeleníteni.

3.1.2 Eredmény és további lehetőségek

Miután a kamerát csatlakoztattam, egy egyszerű Windows terminált megnyitva ellenőriztem, hogy a csatlakoztatás sikeres volt-e, az USB bemeneteken megjelenik-e az Intel RealSense kamerája. Miután ez sikerült, futtattam a programot. Ahogy a 3.1.2-es ábrán látható, a kamerától érkező élő adatfolyam megjelent a laptopom képernyőjén.

Azonban a feladatleírás hangsúlyozza a kamera képének távoli elérését, így a közvetlen USB csatlakoztatás helyett egy vezeték nélküli megközelítést kellett alkalmaznom. A megoldás során az USB interfészöt megtartottam, azonban a kamerát nem a fejlesztői számítógéphez, hanem közvetlenül a robot vezérlőegységéhez csatlakoztattam. Ennek a konfigurációnak a részletes leírása és implementációs folyamata a következő fejezetben kerül bemutatásra.



3.1.2. ábra – Futtatott kód és a megjelenő kamera kép

3.2 OpenCV GStreamer

Dolgozatom legnagyobb kihívásának a képfelismeréshez használt kamera képének távolról való megjelenítése bizonyult. Az előzőekben tárgyaltam, hogy USB használatával sikerült megoldani a problémát, azonban a közvetlen vezeték nélküli eset már nem volt olyan egyszerű.

Míg a távoli robotvezérlésnek számos, kézhez kapott lehetősége létezik (melyeket a 2.1-es fejezet tárgyal részletesebben), addig a kamera képének hálózaton történő elérése annál összetettebb, és bonyolultabb megközelítést kíván. A legkézenfekvőbb megoldást a ROS használata jelentette volna, hiszen egyetlen ROS-csomag importálásával megoldható lenne a kamera képének megjelenítése. Azonban, mint azt a 2.2.4-es fejezetben kifejtettem, részletesen tárgyalt okok miatt az RTDE interfész mellett döntöttem, így egy olyan lehetséges módszert kellett keresnem, amely során a két eszköz használatát kombinálni tudtam.

A ROS használata nélkül az egyetlen járható utat a kamera képének streamelése jelentette. Így megoldásom alapja az OpenCV Gstreamer, mely egy rendkívül hatékony, nyílt forráskódú multimédia keretrendszer, amely az OpenCV-vel együttműködve komplex videó feldolgozási és streaming megoldásokat tesz lehetővé [12]. A GStreamer egy pipeline-alapú architektúrát használ, ahol az adatok különböző feldolgozási elemeken (plugin-eken) keresztül áramlanak.

Számos előnnyel rendelkezik a többi multimédiás keretrendszerrel szemben: széleskörű formátum- és protokolltámogatást nyújt, beleértve az RTSP, RTP, UDP és TCP protokollokat, a H.264, MJPEG, VP8 és VP9 kodekeket, valamint különböző konténerformátumokat. Moduláris felépítéssel rendelkezik, amely lehetővé teszi pluginek széles választékának használatát, miközben az architektúra könnyedén bővíthető és a pipeline-ok tetszés szerint testreszabhatók. Továbbá a teljesítménye kiemelkedő, amit a hardware-alapú kódolás és dekódolás támogatása, az alacsony késleltetés, valamint a hatékony memóriakezelés biztosít.

3.3 Távoli megjelenítés megvalósítása

A kamera képének hálózati streamelése egy szerver-kliens architektúrát igényel, ahol a szerver szerepét a UR3e robotkar vezérlője tölti be, míg kliensként a saját eszközöm szolgál. Ebben a fejezetben részletesen be fogom mutatni mind a kliens, mind a szerver szükséges beállításait ahhoz, hogy egy közös hálózaton keresztül megvalósítható legyen a kamera képének távoli lekérdezése.

3.3.1 Technikai tudnivalók, kezdeti nehézségek

Mivel a GStreamer támogatja számos protokollnak a használatát, ezért alap koncepciónak azt választottam, hogy az UDP 5000-es portján fogom a stream közvetítését lebonyolítani.

Az OpenCV GStreamer funkcionalitásának használatához először telepítenünk kell magát a GStreamer keretrendszert. Bár ez minden operációs rendszeren lehetséges, a Linux disztribúciók kínálják a legegyszerűbb telepítési folyamatot. Fontos megjegyezni, hogy Python környezetben történő használat esetén nem elegendő csupán a GStreamer telepítése, az OpenCV könyvtárat kifejezetten úgy kell fordítani, hogy támogassa a GStreamer integrációt. Enélkül az OpenCV Python modulja nem lesz képes hozzáférni a GStreamer által biztosított funkciókhöz, ami megakadályozza a stream megjelenítését és feldolgozását Python alkalmazáson keresztül.

A megvalósítást a saját, Windows környezetben kezdtem el, ahol elsődleges feladatom a GStreamer telepítése, majd annak OpenCV-be való beimportálása volt. Munkám során hivatalos útmutatókból és linkekkel tájékozódtam, azonban az alkalmazás telepítését követően az OpenCV-vel való összehangolás nem volt sikeres. Mivel maga a telepítési és konfigurálási folyamat rengeteg időmet felemészette, és nem sikerült a

problémát záros határidőn belül orvosolnom, ezért a haladás érdekében más útvonalat kellett választanom.

3.3.2 Kliens oldal

Mint ahogy a 3.3.1-es fejezetben említettem, a GStreamer telepítése és Python környezetbe történő integrálása Linux rendszereken valósítható meg a legegyszerűbben. Ezért döntöttem a WSL2 (Windows Subsystem for Linux 2) használata mellett, amely lehetővé tette egy Ubuntu alapú fejlesztői környezet kialakítását, ahol minden szükséges komponenst és eszközt telepítettem a feladat megvalósításához.

Ezen környezet kialakításának legfontosabb része a GStreamer keretrendszer volt, melyet az alábbi parancssal installáltam:

```
sudo apt install gstreamer1.0-plugins-base gstreamer1.0-plugins-good  
gstreamer1.0-plugins-bad gstreamer1.0-plugins-ugly gstreamer1.0-libav
```

Ez tartalmazza a GStreamer rendes működéséhez szükséges plugin-csomagok telepítését, melyek együtt biztosítják a legtöbb multimédiás formátum kezelését és a különböző streaming protokollok támogatását.

Azonban az én esetemben a működéshez még elengedhetetlen egy plusz csomag telepítése, mely az alábbi parancssal valósítható meg:

```
sudo apt install gstreamer1.0-tools
```

A gstreamer1.0-tools csomag a GStreamer alapvető parancssori eszközeit tartalmazza. A legfontosabb közülük a gst-launch-1.0, amely pipeline-ok gyors tesztelésére szolgál, valamint a gst-inspect-1.0, amivel részletes információkat kaphatunk a telepített pluginekről. A csomag része még a gst-discoverer-1.0, amely médiafájlok tulajdonságait vizsgálja, és a gst-device-monitor-1.0, ami az elérhető média eszközök felderítésében segít. Ezek az eszközök nélkülözhetetlenek a fejlesztés és a hibakeresés során. Számomra azért volt különösen fontos, ugyanis a stream működését terminálból kiadott parancsokon keresztül tudtam ellenőrizni.

Miután minden szükséges csomag telepítve lett, az alábbi parancssal lehet lekérdezni, hogy sikeres volt-e az installáció.

```
gst-launch-1.0 --version
```

Kimenetként megjeleníti a GStreamer telepített verzióját és egyéb információkat, mely a 3.3.2-es képen látható. Az én eszközömön az 1.20.3-as verzió fut.

```
gst-launch-1.0 version 1.20.3
GStreamer 1.20.3
https://launchpad.net/distros/ubuntu/+source/gstreamer1.0
```

3.3.2. ábra – GStreamer verzió

Miután a fent említett plugin-ek sikeresen letelepültek, az eszközünk képes a hálózaton lévő stream-ek elkapására és megjelenítésére. A továbbiakban arról lesz szó, hogy milyen egyéb alkalmazásokra van még szükség a feladatom megvalósítása során.

A fejlesztői környezet teljessé tételehez telepítenünk kell a Python programozási nyelvet, valamint a működéshez szükséges függvénykönyvtárakat és modulokat. A 3.1-es fejezetben ismertetett megoldáshoz hasonlóan a kamera képének megjelenítéséhez és feldolgozásához szükség van a cv2 és a numpy könyvtárak telepítésére, valamint az rtde_control és rtde_recieve importokra is. Így a WSL2 rendszeren keresztül nem csak a kamera képét jelenítem meg távolról, de a robot mozgatását is elvégzem. Az alábbi parancsok kiadása szükséges ehhez:

```
sudo apt install python3
sudo apt install python3-pip
pip3 install opencv-python numpy rtde_rtde
```

Az utolsó eszköz, amire szükségem volt, az egy fejlesztői platform, ahol lehetőség van könnyedén és átláthatóan kódot írni, azt debug-olni és futtatni. Ehhez a Visual Studio Code-ot választottam, melyet a WSL2 rendszerben telepítettem, majd onnan megnyitva ugyanúgy tudtam szerkeszteni a programomat, mintha a saját, Windows környezetben dolgoznék.

Az eddig tárgyalt alkalmazások és eszközök elengedhetetlenek a feladatom megoldásához. Azonban miután minden elem telepítésre került, a Python program még nem fog megfelelően működni. Ahogy a fejezet elején említettem, a GStreamer alkalmazást bele kell integrálni az OpenCV könyvtárába, hogy használat során együtt tudjon fordulni a kettő, így lehetőségünk legyen a GStreamer funkcióit az OpenCV eszközeivel kezelni egy saját alkalmazáson keresztül.

3.3.2.1 GStreamer integrálása OpenCV-be

Az OpenCV-vel való használat érdekében szükség van még az alábbi csomagok telepítésére is [13]:

- ubuntu-restricted-extras: ez a csomag olyan extra kodekeket és médiaformátumokat telepít, amik licensz miatt nem részei az alap Ubuntu telepítésnek, azonban egyes videóformátumok kezeléséhez elengedhetetlenek.
- libgstreamer1.0-dev és libgstreamer-plugins-base1.0-dev: minden két csomag a GStreamer fejlesztői könyvtárait tartalmazza. Ezek azért szükségesek, mert az OpenCV fordításkor ezekből építi be a GStreamer támogatást.

Az OpenCV könyvtár megfelelő újra telepítése és konfigurálása szintén kulcsfontosságú lépés. A standard telepítési módszerek helyett ebben az esetben a forráskóból történő fordítást kell választanunk, ami bár összetettebb folyamat, de lehetővé teszi a könyvtár pontos testreszabását. Ez különösen fontos a GStreamer támogatás implementálásához, mivel csak így biztosítható, hogy az OpenCV a megfelelő modulokkal és függőségekkel épüljön fel. Ehhez az alábbi parancsokra van szükség:

```
git clone https://github.com/opencv/opencv.git
cd opencv/
git checkout 4.1.0
```

A forráskódot a hivatalos Github oldalról van lehetőség letölteni. A könyvtárba belépve szükséges kiválasztani a verziót, amely szintén körültekintést igényel. Ennek kiválasztásánál két kritikus szempontot kell figyelembe vennünk: egyszerűsítve a stabilitást, amely a megbízható működés alapfeltétele, másrészt a kompatibilitást, amely biztosítja a zökkenőmentes együttműködést mind a GStreamer keretrendszerrel, mind pedig a projekt során használt egyéb komponensekkel. A nem megfelelő verzió választása stabil működéshez, vagy akár a különböző komponensek közötti konfliktusokhoz vezethet.

A telepítési folyamat utolsó szakaszában egy build könyvtárat kell létrehoznunk az OpenCV projekten belül. Ebben a könyvtárban a CMAKE segítségével meghatározhatjuk az OpenCV fordítási paramétereit és funkcionálisitását. A konfigurációs folyamat során különös figyelmet fordítottam két kulcsfontosságú elemre: a Python környezeti változók pontos beállítására, valamint a GStreamer támogatás aktiválására. A

fordítást és telepítést követően az ldconfig parancs futtatása biztosítja, hogy az operációs rendszer megfelelően felismerje és integrálja az újonnan telepített könyvtárkomponenseket, ezáltal téve működőképpessé a rendszert.

Az OpenCV GStreamer helyes működését egy egyszerű Python lekérdezéssel lehet ellenőrizni, az alábbi módon:

```
import cv2
print(cv2.getBuildInformation())
```

A kód importálja az OpenCV könyvtárat (cv2), majd a getBuildInformation() függvény segítségével részletes információt szolgáltat a telepített OpenCV verzióról. A függvény megjeleníti a könyvtár fordítási beállításait, beleértve a támogatott funkciókat, a használt fordítót, a különböző függőségek verzióit és az aktivált modulokat. Ez különösen hasznos annak ellenőrzésére, hogy a GStreamer támogatás és egyéb szükséges komponensek megfelelően kerültek-e beépítésre a telepítés során. Sikeres integrálást követően a fenti Python kód futtatása során a 3.3.2.1-es képen látható kimenetet kell kapni, ahol a GStreamer sort megkeresve a YES szónak, valamint az elérhető verziónak a számának kell szerepelni.

swscale:	YES (3.3.100)
avresample:	NO
GStreamer:	YES (1.19.90)
PvAPI:	NO
v4l/v4l2:	YES (linux/videodev2.h)

3.3.2.1. ábra – GStreamer sikeres integrálása OpenCV-be

Ezekkel a lépésekkel hoztam létre a WSL2-n futtatott Ubuntuban a munkám elvégzéséhez szükséges környezetet, mely innentől kezdve képes a kamera adatait fogadni stream formájában hálózaton keresztül, valamint TCP/IP kapcsolat létrehozását követően tudja mozgatni a robotkart egy egyszerű Python alkalmazás segítségével.

Itt még kitérnék egy fontos ellentmondásra a megoldás környezetével kapcsolatban. A 2.2-es fejezetben tárgyaltam, hogy a robotmozgatáshoz az RTDE interfész eszközeit szeretném használni, melynek elsődleges okának a Windows környezet szükségességét hoztam fel, ennek ellenére mégis WSL2-ben folytattam a munkát, mely egy Linux disztribúciót futtat. A probléma feloldása abban rejlik, hogy ahogyan említettem, az RTDE interfész platformfüggetlen, így teljesen mindegy, hogy milyen operációs rendszer alatt kívánjuk azt használni. Ezért WSL2-n keresztül is

pontosan ugyanazzal a kóddal tudtam TCP/IP kapcsolatot kezdeményezni, mint Windows rendszer esetén.

3.3.3 Szerver oldal

Ahhoz, hogy a kamera képet stream formájában az adott hálózaton távolról el tudjuk érni, szükség van egy szerverre, amely a kamera képet közvetíti. Esetben ezt a szervert a UR3e robotkar vezérlője (NUC) jelentette, ahova a RealSense kamera USB vezetéken keresztül csatlakoztatva lett.

A saját laptopomról könnyedén, az SSH segítségével tudtam a szervert beállítani. Itt arra az apró részletre kell figyelni, hogy a NUC IP címe nem egyezik meg a robot IP címével. Ez a gyakorlatban azt jelenti, hogy míg az RTDE interfész a 10.150.0.1-es IP címen keresztül kezdeményezi a TCP/IP kapcsolatot, addig a kamera képének megosztását a 10.150.0.3. IP címmel rendelkező NUC végzi.

Szerver esetén is szükség van a GStreamer csomagjainak telepítésére. Mivel a robot vezérlőjén is Linux operációs rendszer fut, ezért telepítését ugyanazzal a parancssorral tudtam megvalósítani, mint WSL2 esetén, melyet a 3.3.2-es fejezetben részleteztem.

A stream elindítására és működtetésére számos lehetőség létezik, akár Python, akár Docker alkalmazásával. Azonban én a képfelismerést és képfeldolgozást a saját eszközömön, kliens oldalon valósítom meg, így a szerver csak a stream elindítására szolgál, amit egy egyszerű, terminálból kiadott parancssal el lehet érni. Ebből kifolyólag a feladatom megoldása érdekében nincs szükség további alkalmazás és csomag telepítésére szerver oldalon.

Itt még kitérnék egy fontos kérdésre. A 3.1-es fejezetben tárgyalt közvetlen kapcsolatos megoldásban előjön a RealSense SDK 2.0 telepítésének a szükségessége. Így felmerül a kérdés, hogy a stream megosztásához nincs-e szükség ennek a keretrendszernek a letöltésére. A válasz az, hogy nem, mivel a szerverre beérkező, kamera által szolgáltatott stream már egy szabványos formátumú videó. Maga a szerver nem kommunikál közvetlenül a RealSense kamerával, így az SDK telepítése nélkül is meg tudja osztani annak képét a hálózaton. Erre abban az esetben lenne szükség, ha a kamera speciális funkcióit szeretnénk távolról vezérelni, vagy például a mélységi információkat feldolgozni.

3.3.4 Stream működése

A videotream működéséhez elengedhetetlen a megfelelő környezet konfigurálása mind a szerver, mind a kliens oldalon. Amikor minden két oldal megfelelően van beállítva, lehetővé válik a kamera képének valós idejű továbbítása hálózaton keresztül.

A GStreamer keretrendszer rugalmas lehetőséget kínál a stream konfigurálására, szabadon választható minden az átviteli protokoll típusa, minden pedig a kommunikációs port száma. Én a megvalósítás során az UDP protokollt (User Datagram Protocol) választottam, amely az 5000-es porton keresztül továbbítja az adatfolyamot.

A stream elindításához először beléptem SSH segítségével a robotkar vezérlőegységébe, ahova előtte USB-n keresztül bekötöttem a robotra felszerelt RealSense kamerát. Ezt követően az alábbi parancs kiadásával indítottam el a stream-et:

```
gst-launch-1.0 v4l2src device=/dev/video4 ! videoconvert ! x264enc tune=zerolatency bitrate=500 ! rtph264pay ! udpsink host=<IP-cím> port=5000
```

Ebben az utasításban minden kulcsfontosságú elemnek szerepelnie kell. Kötelező megadni, hogy a stream képe honnan érkezik. Ezt a 'device' használatával áll módunkban megtenni, ahol én a RealSense kamera elérési útját állítottam be. Ahogy látható, a 4-es számú eszköz jelenti az én esetben a kamerát, melyet úgy derítettem ki, hogy végig próbáltam az összes lehetőséget. Ezen kívül meg kell adni, hogy milyen formátumban szeretnénk megosztani a képet, amely itt a H.264 videótömörítő, melyet az 'x264enc' kulcsszóval állítottam be.

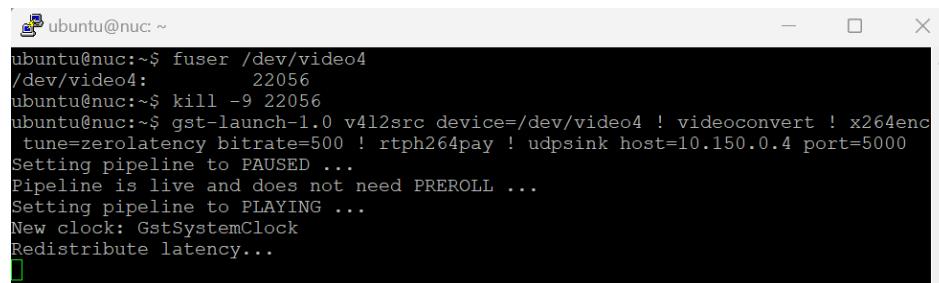
A három legfontosabb paraméter a parancs legvégén található. Először is az 'udpsink' felel azért, hogy UDP legyen a hálózati kimenet. Maga az UDP - a TCP-vel (Transmission Control Protocol) ellentétben - nem garantál megbízható adatátvitelt. Azonban mivel nem vár visszaigazolást a csomagok megérkezéséről és nem foglalkozik az újraküldéssel ezek elvesztése esetén, ezért a késleltetése alacsonyabb, valamint nincs kapcsolat-felépítési folyamat, így az UDP fejléce kisebb, mint a TCP-é. Ezen okokból kifolyólag az UDP alkalmazása késleltetés érzékeny vagy valós idejű stream-ek esetén előnyösebb választás.

A videotream megfelelő konfigurálásánál különös figyelmet kell fordítani a céleszköz IP-címének helyes megadására. Ez a cím dinamikusan kerül kiosztásra a WiFi router DHCP (Dynamic Host Configuration Protocol) szolgáltatása által, amikor egy új

eszköz csatlakozik a hálózathoz. A megvalósítás során a robot saját WiFi hálózatát használtam a stream továbbítására. Ahhoz, hogy távolról elérhessem a kameraképet, a saját laptopomat ehhez a hálózathoz kellett csatlakoztatni. A robot routere a 10.150.0.x címtartományt használja a hálózati címek kiosztására, és jellemzően a 10.150.0.4-es címet rendelte az én eszközömhöz. Fontos, hogy a stream konfigurációjában ezt a dinamikusan kiosztott IP-címet kell megadni célállomásként, hogy a videójel megfelelően eljusson a fogadó eszközre, ráadásul a hálózathoz történő minden újracsatlakozás esetén a router új IP-címet oszthat ki, így a stream beállításait ennek megfelelően kell aktualizálni a zavartalan működés érdekében.

A stream konfigurációjának utolsó paramétere a kommunikációs port meghatározása. Bár a portszám elvileg szabadon választható a teljes portszámtartományból, célszerű olyat választani, amely nem ütközik gyakran használt szolgáltatások alapértelmezett portjaival. Az optimális működés érdekében érdemes a kevésbé forgalmas, magasabb számtartományból (például 5000 feletti) választani egy szabad portot, ezzel minimalizálva az esetleges port-ütközések kockázatát és biztosítva a zavartalan kommunikációt.

A stream sikeres elindítása a 3.3.4a. ábrán látható.

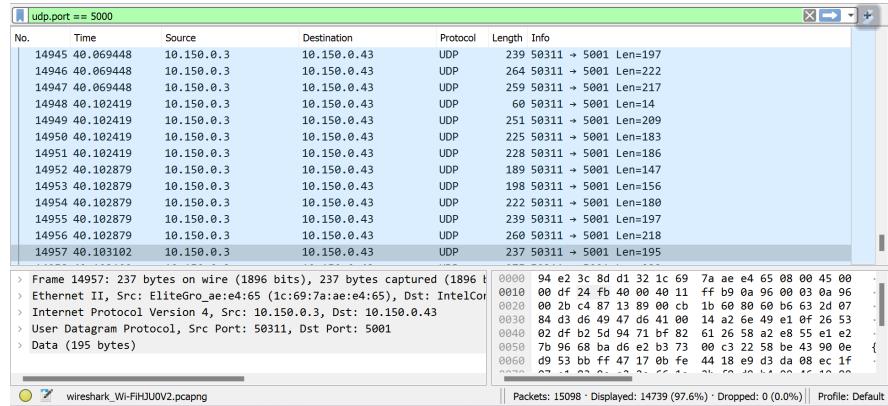


```
ubuntu@nuc:~$ fuser /dev/video4
/dev/video4:      22056
ubuntu@nuc:~$ kill -9 22056
ubuntu@nuc:~$ gst-launch-1.0 v4l2src device=/dev/video4 ! videoconvert ! x264enc
  tune=zerolatency bitrate=500 ! rtph264pay ! udpsink host=10.150.0.4 port=5000
Setting pipeline to PAUSED ...
Pipeline is live and does not need PREROLL ...
Setting pipeline to PLAYING ...
New clock: GstSystemClock
Redistribute latency...
```

3.3.4a. ábra – Stream sikeres elindítása a szerveren

A fenti kép mutat még egy említésre méltó esetet. A videotream indítása során számos hibaüzenetet kaphatunk, mely leírja, hogy miért volt sikertelen a kép hálózaton történő megosztása. A leggyakoribb eset az, amikor azt a visszajelést kapjuk, hogy a forrásként megadott eszköz már használva van egy másik erőforrás által. Ez akkor szokott előfordulni, amikor megszakad az SSH kapcsolat, mielőtt a stream-et le tudnánk állítani, vagy ha egy korábbi sikertelen kísérlet során a folyamat a háttérben még mindig fut. Ezen probléma megoldásához először azonosítanunk kell a kamerát használó folyamat azonosítóját, majd szabaddá kell tennünk az erőforrást a folyamat leállításával.

Azt, hogy a stream a megfelelő eszköz IP-címére érkezik, egyszerűen ellenőrizhetjük a Wireshark használatával. Megnyitottam az alkalmazást a saját laptopomon, majd szűrtem az UDP 5000-es portjára. Ahogy a 3.3.4b. ábrán látható, folyamatosan érkeztek az adatok, így a megosztás a szerveren helyesen lett felkonfigurálva.



3.3.4b. ábra – Wireshark a stream működése közben

Kliens oldalon a stream megjelenítéséhez nem szükséges azonnal egy Python alkalmazást összerakni. Mivel a gstreamer1.0-tools telepítésre került, így helyette terminálból kiadhatunk egy egyszerű parancsot, amely megjeleníti a hálózaton megosztott kameraképet. Ezt az alábbi utasítás kiadásával tudjuk elérni:

```
gst-launch-1.0 udpsrc port=5000 caps="application/x-rtp, encoding-name=H264, payload=96" ! rtph264depay ! avdec_h264 ! videoconvert ! autovideosink
```

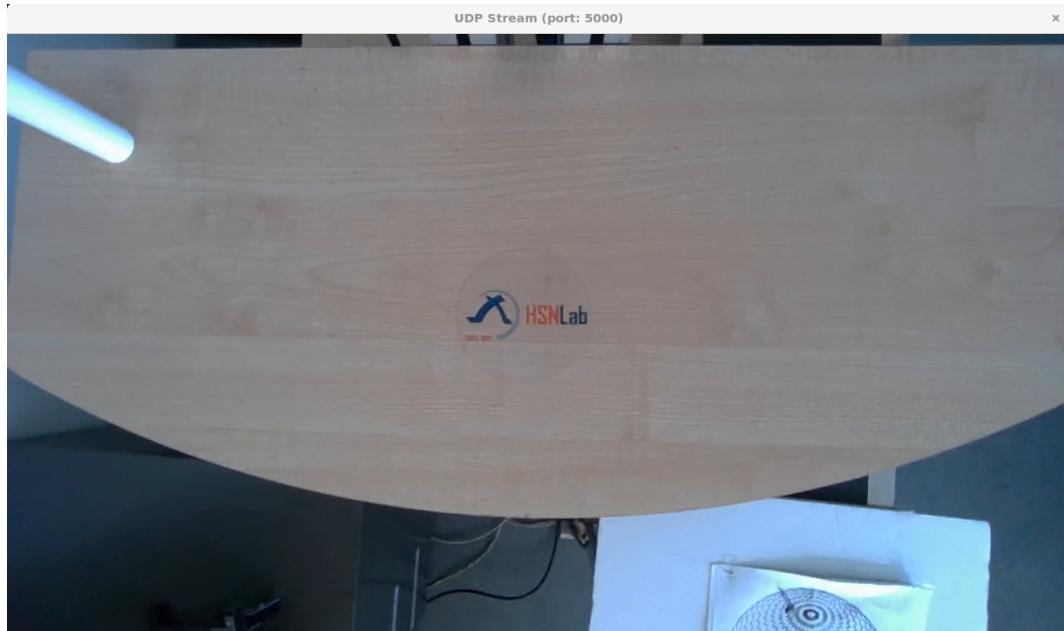
Ez a pipeline UDP stream-en keresztül érkező H.264 videót fogad és jelenít meg. Felépítése hasonlít a szerver oldalon használthoz. WSL használata esetén figyelni kell arra, hogy a WSL által futtatott rendszer nem fér hozzá közvetlenül a Windows WiFi interféseihez. Ez azért lehet gond, mert a stream elindításánál a Windows IP-címét állítottuk be célként, amely megfelelő engedélyezés nélkül a WSL-en futtatott kódban nem érhető el. Itt a megoldás az, hogy a Windows IP-címét meg kell adni a WSL-nek, mint alapértelmezett ájtáró, így el tudjuk érni belőle a külső hálózati interfészeket is.

Sikeres adatátvitel esetén a kamera által látott képnek meg kell jelennie a saját eszközünk képernyőjén. Nálam ez a 3.3.4c. ábrán látható.



3.3.4c. ábra – Laptopomon megjelenő kamerakép

Python szkripten keresztül nagyon hasonlóan lehet ugyanezt az eredményt elérni. A kód elején szükség van beimportálni a cv2 könyvtárat, majd létre kell hozni egy VideoCapture objektumot, melynek forrásaként a GStreamer pipeline fog szolgálni. Ezt követően lehetőségünk van megjeleníteni az OpenCV imshow függvényével a beérkező stream képet. A 3.3.4d. ábrán látható, hogy a kód futtatását követően az UDP Stream ablakban megjelent a RealSense kamera képe.



3.3.4d. ábra – RealSense kamera képe Python szkripttel

Ezen lépésekkel és beállításokkal valósítottam meg a kamera kép hálózaton kereszttüli átvitelét.

4 Fejezet

Képfeldolgozás és színezés

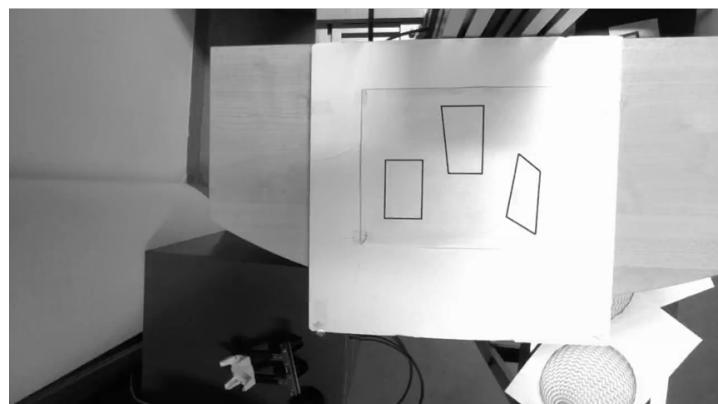
Ebben a fejezetben kerül bemutatásra a kamera által látott kép feldolgozása, valamint a színezés folyamata. Ismertetem, hogy a kódom milyen algoritmus alapján ismeri fel a papíron lévő négyzeteket, szó lesz a koordináták problémáiról, illetve részletesen tárgyalom a színezés koncepcióját és a robot mozgatásának megvalósítását is.

4.1 Négyzetek felismerése

A képfelismerés megvalósításához az OpenCV által nyújtott eszközöket és függvényeket használtam, melyet több egymásra épülő képfeldolgozási és számítógépes látás algoritmus kombinációjával oldottam meg [14].

Az OpenCV használata esetén általában az objektumfelismerést mintaillesztéssel valósítják meg, azonban én nem ezt az utat választottam. Az algoritmusom alapja, hogy a program először úgynevezett kontúrokat keres a képen, majd később ezeket felelteti meg különböző kritériumoknak. Ha az összes feltételnek eleget tesz egy kontúr, a program a továbbiakban egy négyzöként fogja azt kezelni.

Mielőtt a képfelismerés megvalósulna, vannak bizonyos előfeldolgozási lépések, amelyek szükségesek a hatékony és pontos működés érdekében. Ide tartozik a beérkező stream binarizálása, ami annyit tesz, hogy a beérkező színes képkockákat átkonvertálja fekete-fehér pixelekre (4.1a. ábra). Erre a lépésre azért van szükség, mert csökkenti a feldolgozandó adatmennyiséget, ezáltal növeli a feldolgozási sebességet.



4.1a. ábra – Binarizált kép

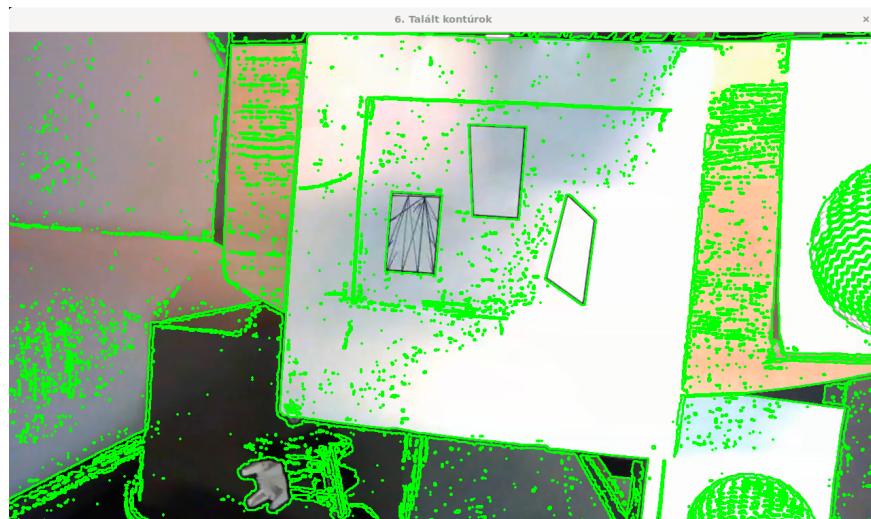
Második elengedhetetlen lépés a zajszűrés. Én ezt Gauss-szűrő segítségével valósítottam meg, melynek lényege, hogy minden képpontot és annak környezetét egy kétdimenziós Gauss-függvény szerint súlyoz. Ez azért lényeges, mert a képeken jelenlévő zaj és apró részletek zavarhatják a kontúrdetekciót, ezáltal akár az algoritmus félre detektálhatja a felismert alakzatokat.

Ezt követően én még alkalmaztam két előfeldolgozási lépést, melyek nem létfontosságúak, azonban a pontosabb detekció miatt hasznosak. Az egyik az úgynevezett adaptív küszöbölés. Ennek a módszernek a segítségével elérhetjük azt, hogy a program alkalmazkodjon változó képviszonyokhoz. Ez a probléma nálam akkor jött elő, amikor különböző napszakokban dolgoztam a kamerával, és erősebb megvilágítás esetén tökéletesen felismerte a négyzeteket a képen, azonban ha kicsit árnyékosabb, sötétebb volt a környezet, akkor már hibás volt a működése. A másik hasznos funkció a dilatáció. Vékonyabb vonalak esetén előfordult, hogy az algoritmus nem ismerte fel a kontúrokat, ezáltal volt olyan négyzet, amelyet nem tudott detektálni. Erre egy jó megoldásként a dilatáció alkalmazása szolgált, ugyanis működésének lényege, hogy a küszöbölés utáni apróbb szakadásokat javítja, mégpedig úgy, hogy összeköti a közel eső képpontokat és kitölți a kis réseket.

Az előfeldolgozási lépések követően jöhét maga a kontúrdetektálás, és annak eldöntése, hogy ezek közül melyek számítanak négyzetnek. Miután a potenciális négyzeteket felismeri a program, ezt követően az alábbi szűréseket végzi:

- terület alapú szűrés: ennek célja, hogy kiszűrje a túl kicsi vagy túl nagy objektumokat, amelyek valószínűleg nem releváns négyzetek. Ilyenek lehetnek például a zajból eredő kis objektumok, illetve a félredetektált nagy méretű négyzetek.
- oldalarány vizsgálat: ez a lépés szintén a félredetektált objektumokat igyekszik kiszűrni, mégpedig úgy, hogy kiszedi a túl keskeny vagy túl széles objektumokat.
- geometriai tulajdonságok ellenőrzése: a program meghatározza a valódi kontúrokat és ellenőrzi, hogy 4 sarokpontból állnak-e a felismert objektumok.

A 4.1b ábrán látható az összes detektált kontúr, mielőtt az előbb felsorolt szűrések el lennének rajtuk végezve. Látható, hogy rengeteg pontot detektál az algoritmus, amelyből a tényleges négyzeteket ki kell szűrni.

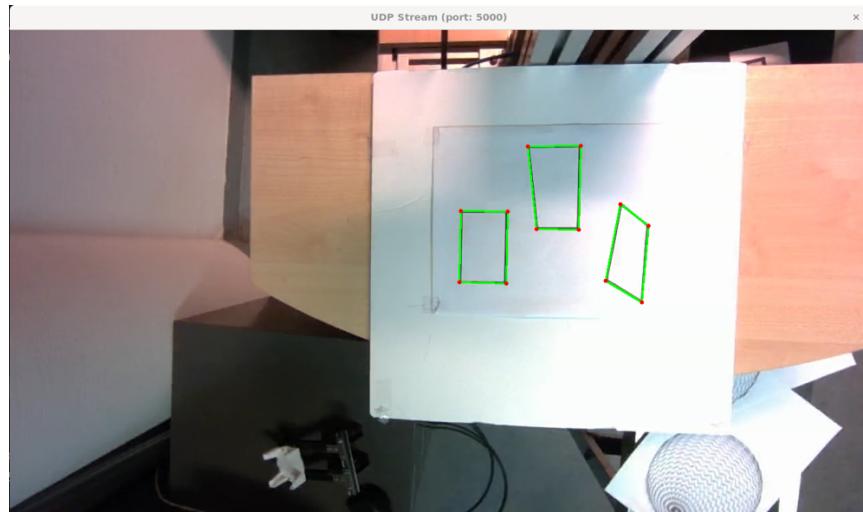


4.1b ábra – Az összes detektált kontúr

A fent említett lépésekkel az általam készített A4-es papíron szinte minden futtatásnál helyesen felismerte a program a képen szereplő négyzeteket, és nagyon ritkán detektált félre az algoritmus. Felmerült problémaként még az a jelenség, hogy minden négyzetet duplán detektált, így ha a képen 3 objektum szerepelt, akkor az algoritmus 6 helyes detekcióval tért vissza. Ennek a jelenségnek az oka valószínűleg az volt, hogy a papíron szereplő négyzetek vastag vonallal lettek megrajzolva, melyeknek a külső és a belső kontúrját is felismerte az algoritmus. Ezen hiba megoldásához végeztem még egy szűrést, amely a négyzetek középpontjai között távolságmérést végez, és ha két pont nagyon közel van egymáshoz, akkor az egyiket közülük eldobja, így a felismert négyzetek halmazában nem lesz duplikáció.

Miután futtattam az egész algoritmust, a 4.1c. ábrán látható kimenetet kaptam. Látható, hogy a kamera elé helyezett papíron megrajzolt négyzeteket helyesen felismeri a program, és félredetektálás sem található rajta.

Az OpenCV képes a felismert négyzetek sarokpontjainak koordinátákban való meghatározására. Számomra ez azért különösen fontos, ugyanis a robotmozgatásnak ezen (x, y) párok lesznek az alappillérei.



4.1c. ábra – Helyesen felismert négyzetek

A négyzet detektciót több példára is kipróbáltam, hogy az algoritmus az esetek többségében helyesen működik-e. Néha belebotlottam olyan jelenségebe, hogy esetleg a lap széleit, vagy a robotkar előtt lévő asztal peremét is négyzögnek vélte, azonban ez ritkán fordult elő.

4.2 Színezés megvalósítása

Ebben az alfejezetben bemutatom, hogy a színezést milyen elképzélés alapján hajtottam végre, és hogy ehhez milyen lépésekre volt szükség.

4.2.1 Színezés koncepciója és megvalósítása

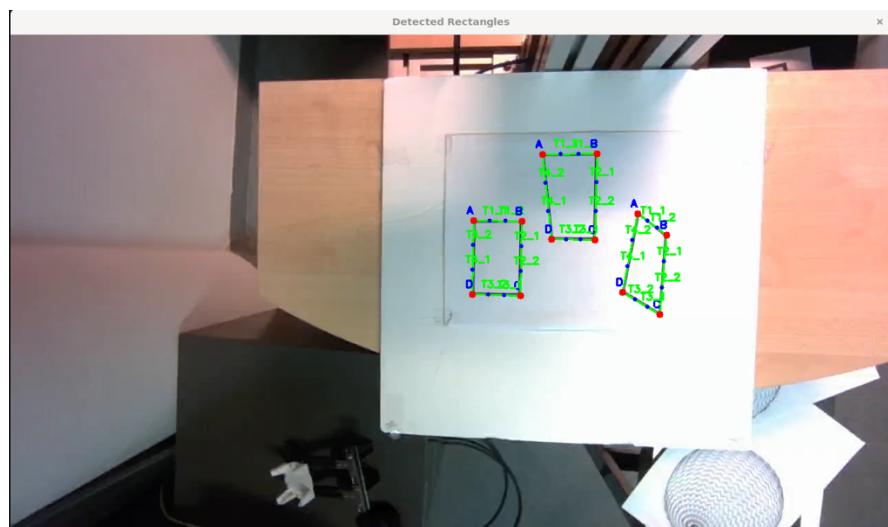
A színezés megvalósításához a négyzetek oldalait három részre osztottam a harmadolópontok segítségével, majd az így kapott pontokat használtam fel úgy, hogy az összes harmadoló- és sarokpontot összekötöttem az összes többivel. Így a négyzetek belsejében egy sűrű hálózatot kaptam.

Számos lehetséges megoldás létezett volna a színezés megvalósítására, azonban számomra ez bizonyult a legegyszerűbben megvalósítható útnak. Próbálkoztam azzal a módszerrel, hogy két szemközti oldalt felosztok 20 pontra, és ezeket kötöm össze, ez viszont megvalósítását tekintve nem sikerült. Ennek oka az volt, hogy próbáltam minden a képernyő bal oldalához legközelebbi és a vele szemben lévő oldalakat felosztani, viszont, ha a négyzet úgy helyezkedett el, hogy az egyik sarokpontja volt a legbaloldalibb pontja, akkor minden a négy oldalt fel kellett volna osztanom, hogy a négyzet ki legyen színezve. Mivel az OpenCV-vel nem tudtam megvalósítani, hogy az ilyen négyzeteket felismerje, és máshogyan kezelje, ezért kitaláltam a harmadolópontos

megoldást. Ennek elkészítése során rájöttem, hogyan tudtam volna az előbb említett módszert megvalósítani, azonban mivel kész lett a harmadolópontos algoritmus, ezért azt már nem módosítottam.

A harmadolópontos színezés megvalósítása érdekében kiegészítettem a képfelismerést egy új függvénnyel, amely a négyzetek sarokpontjainak koordinátáiból kiszámítja azok harmadolópontjait. Az algoritmus két sarokpont közötti távolságot osztott el három egyenlő részre, majd hozzáadta egy listához a meghatározott új koordinátákat, illetve a hozzájuk tartozó sarokpontokat is. Így létrejött egy lista (all_points), amelyben az összes, továbbiakban szükséges (x, y) koordináta pár belekerült.

Ellenőrzésképp megjelenítettem a képet, amely a 4.2.1-es ábrán látható. Bejelöltettem rajta a meghatározott pontokat, a konzolra pedig kiírtam a hozzájuk tartozó koordinátákat.



4.2.1. ábra – Felismert négyzetek a harmadolópontokkal

4.2.2 Koordináták átszámítása

A színezés megvalósításának egyik alapfeltétele a megfelelő koordináták kiszámítása. Mivel az OpenCV és a robot koordináta rendszere különbözik, ezért szükséges az összes meghatározott harmadoló- és sarokpontot egy algoritmus segítségével átszámolni.

Ehhez az úgynevezett projektív transzformációt használtam, amely egy olyan transzformáció, ami egy sík pontjait képezi le egy másik sík pontjaira, homogén koordináták segítségével. Legfontosabb jellemzője, hogy az egyeneseket megtartja, azonban a szögeket és az arányokat nem feltétlenül. Azért erre esett a választásom, mert

a kamera képe torzított perspektívát ad, amely valószínűleg ferde szögből látja a munkateret. Ebből kifolyólag egy egyszerű lineáris leképezés nem elég, mert a perspektívából adódó torzítást nem tudja kezelni, így a koordinátákat nem tudja helyesen kiszámolni.

A megvalósításhoz kiválasztottam hat referencia pontot a felismert sarokpontok közül. Itt fontos megemlíteni, hogy kevesebb pont megadása is lehetséges, nem feltétlen szükséges hozzá hat darab. Azonban a helyes számítás érdekében minél több referencia pont van megadva, annál pontosabban számol az algoritmus.

Miután meghatároztam a pontok koordinátáit OpenCV segítségével, a robotkart filccel a kezében odamozdítottam a Teach Pendant segítségével ezekhez a pontokhoz, majd a getTCPPose() függvényel lekérdeztem az azokhoz tartozó robotkoordinátákat, és feljegyeztem magamnak.

Például a 4.2-es ábrán látható négyzetek közül a bal oldali „A” pontjának a koordinátái: (634, 237). Az ehhez tartozó robotkoordináták pedig a következők: (-0.16573431995682603, -0.23870808023642998).

Miután megvolt a hat referencia pont, létrehoztam egy függvényt, amely az átszámítást elvégzi minden paraméterül megadott pontra. Az alábbiakban látható az implementáció:

```
def transform_coordinates(pixel_points: List[Tuple[int, int]]) ->
List[Tuple[float, float]]:
    ref_pixels = np.array([
        [628, 307, 1],
        [718, 335, 1],
        [711, 426, 1],
        [621, 397, 1],
        [721, 309, 1],
        [799, 314, 1],
        [787, 437, 1]
    ])
    ref_robot = np.array([
        [-0.17096802216745088, -0.2822605130761222],
        [-0.11107599806005546, -0.29851333904563787],
        [-0.11255847019751782, -0.35755271462652743],
        [-0.17275235823745647, -0.34153859191249136],
        [-0.11244621412764004, -0.2760906924091822],
        [-0.06313047631880779, -0.27571414328246074],
        [-0.06391195432021181, -0.3532935132486609]
    ])
    A = np.linalg.lstsq(ref_pixels, ref_robot, rcond=None)[0]
```

```

points_array = np.array(pixel_points)
pixel_points_homog = np.hstack([points_array, np.ones((len(pixel_points),
1))])

robot_points = pixel_points_homog @ A

return list(map(tuple, robot_points))

```

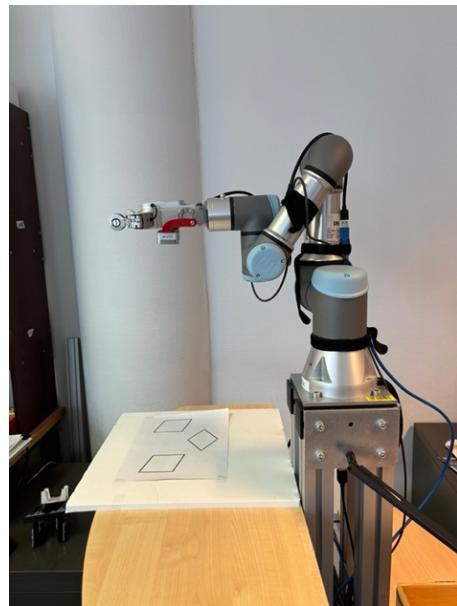
A transzformációs mátrix meghatározásához a NumPy könyvtár linalg.lstsq beépített függvényét alkalmaztam. A módszer homogén koordinátarendszerben dolgozik, ezért a referencia pixelkoordinátákat kiegészítettem egy egységnyi harmadik komponenssel ($Z=1$), létrehozva így a háromdimenziós homogén koordinátákat. A függvény ezt követően minden bemenő pontot automatikusan homogén koordinátákká alakít, majd végrehajtja rajtuk a transzformációt.

A fenti függvény megvalósítását követően indítottam egy ciklust, amely végigmegy a sarok- és harmadolópontokat tartalmazó all_points listán, átszámítja őket robot koordinátarendszerbe, majd egy másik tömböt (robot_coordinates) feltölt velük. Ez azért szükséges, mert a robot a színezés során ebből a listából fog dolgozni, így fogja tudni, hogy hova kell mozognia.

Itt még kitérnék arra, hogy a robotnak meghatároztam az elején egy konkrét helyzetet, ahonnan az egész munkalapot látja maga előtt. Ennek az értékeit feljegyeztem, és beállítottam a Python alkalmazásban, hogy amennyiben a képfelismerést el szeretném végezni, mindenkor ugyanebből a pozícióból induljon. Azért is volt fontos ez a lépés, mert a koordináták transzformációja az ebben a helyzetben meghatározott pixel koordinátákkal dolgozik, ezt tudja átszámítani. Ha a kamera helyzete változna, abban az esetben az átszámítás már torzulna, nem lenne megfelelő.

4.2.3 Működés

Ezek után minden eszköz, kapcsolat és program rendelkezésemre állt ahoz, hogy megvalósítsam a színezési folyamatot. Ahogy a 4.2.2-es fejezetben említettem, a robot kar egy előre meghatározott „home” pozícióból indul, ahonnan az egész papírlapot látja maga előtt. A 4.2.3a. ábrán látható, hogy kívülről ez hogyan néz ki. Az alkalmazás minden esetben úgy indul, hogy a robotot ebbe a kezdő pozícióba navigálja, hogy a képfelismerést helyesen el tudja végezni.



4.2.3a. ábra – Robot „home” pozíciója

A képfelismerés, a harmadolópontok meghatározása és a robotkoordináták átszámolását követően a robotkar filccel a kezében a papír fölé mozdul. Mivel a kamera a robotkar oldalán van elhelyezve, így a „Wirst 1” nevű szervomotornak 90 fokban kell elfordulnia ahhoz, hogy tudjon a papírra rajzolni. Itt is egy előre meghatározott pozícióba kerül, amely azért szükséges, ugyanis a robot azt ki tudja számolni, hogy az egyes koordinátákba hogyan kell eljuttatnia a TCP középpontot, azonban arra nem figyel, hogy közben ne akadjon bele saját magába. Ez a probléma akkor fordulhat elsősorban elő, ha a négyzet sarokpontjai a robot törzséhez közel helyezkednek el, és a kezdő pozícióból egyből ezen négyzet fölé szeretne a robot mozogni. Ez a 4.2.3b. ábrán látható.



4.2.3b. ábra – Robot kezdő pozíciója rajzolás előtt

Miután ez megtörtént, sorra veszi a robot_coordinates tömbben lévő koordinátákat, és egy dupla ciklus segítségével minden pontot összeköt minden másik ponttal. Ehhez egy külön függvényt írtam, melyet a képfelismerés elvégzését követően hív meg a program.

```

for i, start_point in enumerate(all_points):
    for j, end_points in enumerate(all_points):
        if i < j:
            if rtde_r.isConnect() == False:
                rtde_r.reconnect()
            rtde_c.moveL([start_point[0], start_point[1], magassag, rx, ry,
rz], 0.07, 0.07)
            time.sleep(1.0)

            if rtde_r.isConnect() == False:
                rtde_r.reconnect()
            rtde_c.moveL([start_point[0], start_point[1], rajzolomagassag, rx,
ry, rz], 0.07, 0.07)
            time.sleep(1.0)

            if rtde_r.isConnect() == False:
                rtde_r.reconnect()
            rtde_c.moveL([end_point[0], end_point[1], rajzolomagassag, rx, ry,
rz], 0.07, 0.07)
            time.sleep(1.0)

```

Az élek hatékonyabb megrajzolásához kihasználhatjuk a gráfelméleti szimmetriát. Amikor egy pontból kiindulva húzunk éleket, csak a még nem összekötött pontokkal kell foglalkoznunk, mivel a korábbi pontokkal való kapcsolatok már létrejöttek. Így minden él pontosan egyszer kerül megrajzolásra, elkerülve a felesleges duplikációt. Ezt a gyakorlatban úgy oldottam meg, hogy a rajzolás csak a start_point utáni értékekkel történt meg, ezzel is időt és erőforrást takarítva meg.

Megfigyelhető, hogy a kódban minden robot utasítás előtt található két sor, amely a kapcsolat újonnani létrejöttéért felelős. Erre azért van szükség, mert vezeték nélküli kapcsolat esetén az RTDE interfész gyakran elveszíti a kapcsolatot a robottal. Ilyenkor a program futása megszakad, és előlről kell kezdeni az egész folyamatot. Azonban a reconnect() függvény segít, hogy futási időben újraépítse a kapcsolatot a robotkar és a vezérlést megvalósító eszköz között, így segítve a program teljes lefutását.

A robotmozgatáshoz és az RTDE interfész megfelelő működéséhez szükség van a time.sleep() függvény használatára is. Időt ad a robotnak a parancsok végrehajtására, illetve az RTDE interfésznek az adatok frissítésére, valamint megakadályozza a robotvezérlő túlerhelését túl gyakori parancsküldéssel, ami hibákhoz vagy kiszámíthatatlan működéshez vezethetne.

5 Fejezet

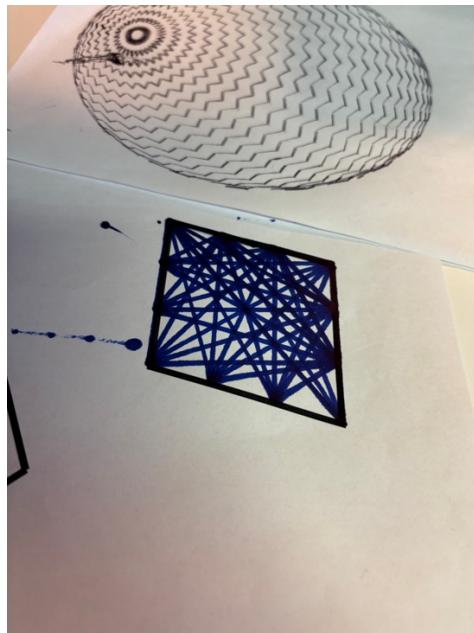
Tesztele és továbbfejlesztési lehetőségek

Ebben a fejezetben a megoldás teszteleét mutatom be, és leírom annak továbbfejlesztési lehetőségeit.

5.1 Tesztele

Amíg nem voltam biztos a működés helyességében, addig olyan magasságba állítottam a robotot, hogy még ne érje el a kezében lévő filc a papírt. Többször futtattam a programot, ellenőrizve, hogy a koordináták mennyire pontosak, hogy ha szükséges, akkor tudjak rajta módosítani.

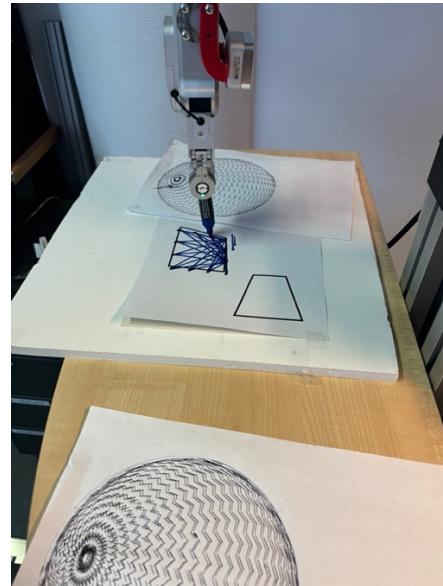
Miután az algoritmus jónak tűnt, megkerestem azt a magasságot a robotkarnak, ahol a filc eléri a papírlapot, de nem nyomja bele túlságosan a hegyét az asztalba. Ez azért is volt fontos, hogy a filc a nagy nyomástól ne ferdüljön el, és a hegye se menjen tönkre. Az első sikeres színezés az 5.1a. ábrán látható.



5.1a. ábra – Kiszínezett négyzet

A koordinátákat jól számolta az algoritmus, a négyzetet helyesen kiszínezte. Mivel ez volt az az alakzat, amelyen a megoldást igyekeztem tökéletesíteni, így szükséges volt más példákra is kipróbálnom a programot. Figyeltem arra, hogy másféle

négyszögeket kelljen kiszínezni, és a robotkar előtt lévő asztalon másoló helyezkedjenek el az alakzatok. Az 5.1b. ábrán látható a robotkar színezés közben egy másik példán. Itt belefutottam abba a hibába, hogy a filcet valószínűleg nem fogattam meg vele elég erősen, így munka közben elferdült. Ebből kifolyólag a vonalak kicsit kifutottak a négyzet oldalvonalaiból.



5.1b. ábra – Rajzolás elferdült filccel

Az 5.1c. ábrán direkt egy szabálytalan négyszöggel próbálkoztam, ahol arra voltam kíváncsi, hogy mennyire követi az oldalvonalakat az algoritmus.



5.1c. ábra – Szabálytalan négyszög színezése

Ebben az esetben is jól számolt a program, a robotkar oldalvonaltól oldalvonallig színezte ki a négyszöget. Összesen hat négyszög esetében próbáltam ki az alkalmazást, amely az esetek többségében jól működött. A koordináták számításánál előfordult pár milliméter eltérés, azonban a végeredményt ez nem befolyásolta.

5.2 Továbbfejlesztési lehetőségek

Az alkalmazás működésében három továbbfejlesztési lehetőséget is meg tudok említeni. Ezek közül az egyik a koordináták átszámítása robot koordináta rendszerbe. Itt van lehetőség az algoritmust finomítani, esetleg egy teljesen más megközelítést kitalálni, amely biztosítja, hogy az a pár milliméteres eltérés ne jelenjen meg. Ez nehéz feladat, ugyanis a robotkoordináták 9-10 tizedes jegy pontossággal állíthatóak be tökéletesen. Illetve a kamera nézőpontja is jelentősen befolyásolja az (x, y) koordináta párokat, hiszen az algoritmus egy előre fixált nézőpontból számolja a koordinátákat, melyet ha megváltoztatunk, az összes eddig helyes érték rossz lesz. Tesztelés során azt a jelenséget is megfigyeltem, hogy a kamera középpontjától távolodva egyre nagyobb a torzítás, egyre nagyobb az eltérés. Ezt én már nem tudtam megoldani, és valószínűleg más koncepcióra van szükség, akár úgy, hogy a robotkart beállítjuk, hogy a kamera pontosan vízszintesen nézzen le a papírra.

Másik lehetőség a színezés közbeni magasság állítása. Mivel a felület, amelyen dolgoztam, nem feltétlenül mindenhol egyenletes és a filc sem biztos, hogy pont merőlegesen áll a papírlaphoz képest, így előfordul, hogy valahol mélyebben belenyomja a robotkar azt az asztalba, valahol meg halványabb vonalat húz, mert kicsit magasabban áll. Ez akkor zavaró, ha az írőszköz ennek köszönhetően elfordul, és nem színezi ki tökéletesen a megtalált négyszöget. Valamilyen megoldást kellene találni arra, hogy a robotkar dinamikusan tudja figyelni ezt, és megfelelően tudja állítani a magasság értékét.

Természetesen a harmadik lehetőség bővíteni a felismerhető sokszögek számát. Itt arra gondolok, hogy ne csak négyszögek esetében működjön, hanem például háromszögeken és ötszögeken is el tudja végezni ugyanezt a feladatot. Ehhez nem csak a képfelismerés és képfeldolgozás részét kellene megvalósítani, de külön koncepcióra is szükség lenne ezen alakzatok kiszínezéséhez.

6 Fejezet

Összefoglalás

A robotkarral való ismerkedést a Témalabor keretein belül kezdtem el. Már akkor tetszett a működése, és érdektelt, hogy egy komplexebb feladatot hogyan lehetne megvalósítani a használatával. Így szakdolgozatomnak egy olyan témát választottam, amely lehetőséget nyújt a robot vezérlésének és működtetésének mélyebb megismerésére.

Összeségében egy komplex, és olykor nagy kihívásokat tartogató munka volt, mely során elmélyülhettem a robottal való munka világában. Az eredménnyel meg vagyok elégedve, a kitűzött célokot sikerült megvalósítanom. Természetesen finomítandó részek, illetve korlátosságok maradtak a megoldásomban, melyeket ki lehetne javítani, azonban az idő szűkössége erre nem adott lehetőséget.

A legnagyobb kihívás a kamera elő képének elérése volt hálózaton keresztül, melynek megvalósításával heteken át dolgoztam. Rengeteg módszert, programot és platformot kipróbáltam, végül hatalmas sikeres eredmény volt, mikor végre sikerült ezt megvalósítanom. A robotkoordináták kiszámításával foglalkoztam még rengeteget, kerestem azt a megvalósítást, amely akár 3-4 tizedes jegyre nagyjából pontos eredményt ad. Ezzel a résszel nem vagyok teljesen elégedett, valószínűleg sokkal szébb és hatékonyabb megoldást is találhattam volna rá, azonban a feladatom szempontjából az általam választott módszer is megfelelőnek bizonyult.

A ROS (Robot Operating System) megismerésére sajnálom, hogy nem volt lehetőségem az idő szűke miatt. Kapcsolat szempontjából lehetséges, hogy sokkal stabilabb megoldás születhetett volna, illetve valószínűleg szébb implementációt is tudtam volna írni a használatával.

A feladat megvalósítása alatt a munkát nagyon élveztem, és jó volt látni a végén az elérte eredményt. Rengeteget tanultam, és sok tapasztalatot szereztem, mely az elsődleges céлом volt ezen feladat kiválasztása során.

Irodalomjegyzék

- [1] Universal Robots: Our History, <https://www.universal-robots.com/about-universal-robots/our-history/> (2024.11.17.)
- [2] Universal Robots: PolyScope, <https://www.universal-robots.com/products/polyscope/> (2024.11.17)
- [3] OnRobot RG2FT,
https://onrobot.com/sites/default/files/documents/Datasheet_RG2-FT_20191122.pdf (2024.11.17.)
- [4] Intel RealSense: D405, <https://www.intelrealsense.com/depth-camera-d405/> (2024.11.17.)
- [5] Wikipédia: OpenCV, <https://hu.wikipedia.org/wiki/OpenCV> (2024.11.26.)
- [6] UniversalRobots: Overview of Client Interfaces, <https://www.universal-robots.com/articles/ur/interface-communication/overview-of-client-interfaces/> (2024.11.26.)
- [7] ROS Home, <https://www.ros.org/> (2024.11.26.)
- [8] Universal Robots: Real-Time Data Exchange Guide, <https://www.universal-robots.com/articles/ur/interface-communication/real-time-data-exchange-rtde-guide/> (2024.12.05.)
- [9] GitHub: RTDE Python Client Library,
https://github.com/UniversalRobots/RTDE_Python_Client_Library (2024.11.26.)
- [10] SocketTest, <https://sockettest.sourceforge.net/> (2024.12.03.)
- [11] Intel RealSense SDK 2.0, <https://www.intelrealsense.com/sdk-2/> (2024.12.03.)
- [12] GStreamer: What is GStreamer?,
<https://gstreamer.freedesktop.org/documentation/application-development/introduction/gstutorial.html?gi-language=c> (2024.12.03.)
- [13] Compile OpenCV with GStreamer, <https://galaktyk.medium.com/how-to-build-opencv-with-gstreamer-b11668fa09c> (2024.12.03.)
- [14] GeeksForGeeks: Essetial OpenCV Functions To Get Started into Computer Visions, <https://www.geeksforgeeks.org/essential-opencv-functions-to-get-started-into-computer-vision/> (2024.12.03.)

Függelék

1.1. ábra - Laborban található Erik nevű robotkar: saját készítésű kép

1.1.2. ábra - PolyScope a Teach Pendant-on: <https://www.universal-robots.com/products/polyscope/>

1.1.4. ábra - Koordináta rendszerek: <https://www.stxim.com/tool-calibration-speeds-implementation-of-6-degrees-of-freedom-industrial-robots/>

1.2. ábra - Gripper, filccel az ujjai között: saját készítésű kép

1.3.1. ábra - RealSense D405 kamera: <https://www.framos.com/en/products/intel-realsense-depth-camera-d405-camera-only-26126>

2.1.3. ábra - ROS komponensek és az adatáramlás:

https://github.com/UniversalRobots/Universal_Robots_ROS2_Driver/blob/main/ur_robot_driver/doc/architecture_coarse.svg

2.2.3. ábra - Robotkar pingelése: saját készítésű kép

2.2.3.1. ábra - SocketTest használata: saját készítésű kép

3.1.2. ábra - Futtatott kód és a megjelenő kamera kép: saját készítésű kép

3.3.2. ábra - GStreamer verzió: saját készítésű kép

3.3.2.1. ábra - GStreamer sikeres integrálása OpenCV-be: saját készítésű kép

3.3.4a. ábra – Stream sikeres elindítása a szerveren: saját készítésű kép

3.3.4b. ábra – Wireshark a stream működése közben: saját készítésű kép

3.3.4c. ábra – Laptopomon megjelenő kamerakép: saját készítésű kép

3.3.4d. ábra – RealSense kamera képe Python szkripttel: saját készítésű kép

4.1a. ábra – Binarizált kép: saját készítésű kép

4.1b. ábra – Az összes detektált kontúr: saját készítésű kép

4.1c. ábra – Helyesen felismert négyzetek: saját készítésű kép

4.2.1. ábra – Felismert négyzetek a harmadolópontokkal: saját készítésű kép

4.2.3a. ábra – Robot „home” pozíciója: saját készítésű kép

4.2.3b ábra – Robot kezdő pozíciója rajzolás előtt: saját készítésű kép

5.1a. ábra – Kiszínezett négyszög: saját készítésű kép

5.1b. ábra – Rajzolás elfordult filccel: saját készítésű kép

5.1c. ábra – Szabálytalan négyszög színezése: saját készítésű kép