ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

NEUMANN JÁNOS
INFORMATIKAI KAR

SZAKDOLGOZAT

OE-NIK	Hallgató neve:	**Horváth Ádám**
2022	Hallgató törzskönyvi száma:	**T/006226/FI12904/N**

Óbudai Egyetem
Neumann János Informatikai Kar
Szoftvertervezés és -fejlesztés Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: **Horváth Ádám**
Törzskönyvi száma: T/006226/FI12904/N
Neptun kódja: [redacted]

A dolgozat címe:

Testrész-felismerési algoritmusok javítása stereo látási rendszerek segítségével
Improving Bodypart-recognition algorithms using stereo vision systems

Intézményi konzulens: Kovács András
Külső konzulens:

Beadási határidő: 2022. május 15.

A záróvizsga tárgyai: Számítógép architektúrák
Szoftvertervezés és –fejlesztés specializáció

A feladat

Tervezzen meg és készítsen el egy olyan rendszert, amely képes egy sztereó kamera által készített bemeneti kép alapján rekonstruálni a jeleneten szereplő személyek testtartását. Az elkészítendő alkalmazás legyen képes a sztereó kamerás rendszer kalibrálására, valamint mélységbecslésre. Nézze át a számítógépes sztereó látáshoz, valamint pózbecsléshez kapcsolódó irodalmakat. Mutassa be a hasonló rendszerek jellemzőit, térjen ki, hogy ezek a rendszerek milyen módszereket, technológiákat alkalmaznak és ezekkel milyen eredményeket értek el. Az irodalomkutatás alapján tervezze meg rendszerének felépítését, specifikálja a be- és kimeneteket, majd fejlessze le azt! Alkalmazását tesztelje és eredményeit értékelje a hasonló rendszerekhez hasonlítva, ezek ismeretében pedig vonja le a szükséges következtetéseket.

A dolgozatnak tartalmaznia kell:

- a szakirodalom részletes áttekintését és értékelését,
- a megoldandó feladat pontos leírását és részletes elemzését,
- a meglévő rendszerek elemzését,
- az alkalmazni kívánt technológiák elemzését, döntések indoklását,
- a rendszertervet,
- a megvalósítás pontos leírását,
- a tesztelési tervet, teszteredményeket, az eredmények és a fejlesztés során felmerült problémák elemzését,
- az elkészült rendszer felhasználási és továbbfejlesztési lehetőségeit.

ÓBUDAI EGYETEM Neumann János Informatikai Kar 1. Budapest

PH.

.....
Dr. Vámossy Zoltán
intézetigazgató

A szakdolgozat elévülésének határideje: **2024. május 15.**
(OE TVSz 55.§ szerint)

A dolgozatot beadásra alkalmasnak tartom:

.....
külső konzulens

.....
intézményi konzulens

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozatomban található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja.

Budapest, 2022. Május 11.

Horváth Ádám
hallgató aláírása

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

9. sz. melléklet

KONZULTÁCIÓS NAPLÓ

Hallgató neve: Neptun kód: Tagozat:

Horváth Ádám ..... [redacted] ..... Nappali .....

Telefon: Levelezési cím (pl: lakcím):

[redacted]

Szakdolgozat címe magyarul:

Testrész-felismerési algoritmusok javítása stereo látási rendszerek segítségével.....

Szakdolgozat címe angolul:

Improving Bodypart-recognition algorithms using stereo vision systems .....

Intézményi konzulens: Külső konzulens:

Szabó-Resch Miklós Zsolt..... .....

Kérjük, hogy az adatokat nyomtatott nagybetűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	**2021.02.12.**	**Téma konkretizálása, ütemterv megbeszélése**	Szabó-Resch Zsolt
2.	**2021.03.17.**	**Eddigi haladás áttekintése, véleményezése**	Szabó-Resch Zsolt
3.	**2021.04.14.**	**Eddigi haladás áttekintése, véleményezése**	Szabó-Resch Zsolt
4.	**2021.05.05.**	**Szakdolgozat áttekintése, véleményezése, prezentációval kapcsolatos tudnivalók**	Szabó-Resch Zsolt

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat I. tantárgy követelményét teljesítette, beszámolóra bocsátható.

Budapest, 2021.05.13.....

Szabó-Resch Zsolt
.....
intézményi konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Neumann János Informatikai Kar

KONZULTÁCIÓS NAPLÓ

Hallgató neve:	Neptun Kód:	Tagozat:
HORVÁTH ÁDÁM	[redacted]	NAPPALI

Telefon: Levelezési cím (pl.: lakcím):

[redacted]

Szakdolgozat címe magyarul:

TESTRÉSZ-FELISMERÉSI ALGORITMUSOK JAVÍTÁSA STEREO LÁTÁSI RENDSZEREK SEGÍTSÉGÉVEL

Szakdolgozat címe angolul:

IMPROVING BODYPART-RECOGNITION ALGORITHMS USING STEREO VISION SYSTEMS

Intézményi konzulens: Külső konzulens:

KOVÁCS ANDRÁS

Kérjük, hogy az adatokat nyomtatott nagy betűkkel írja!

Alk.	Dátum	Tartalom	Aláírás
1.	2022.02.15.	IMPLEMENTÁCIÓ TERVEZÉS	
2.	2022.03.25.	IMPLEMENTÁCIÓ ÁTTEKINTÉSE	
3.	2022.04.10.	TESZTELÉSEK KIÉRTÉKELÉSE	
4.	2022.05.10.	VÉGSŐ ELLENŐRZÉS, LEZÁRÁS	

A Konzultációs naplót összesen 4 alkalommal, az egyes konzultációk alkalmával kell láttamoztatni bármelyik konzulenssel.

A hallgató a Szakdolgozat II. tantárgy követelményét teljesítette, védésre bocsátható.

.....

Intézményi konzulens

Budapest, 2022. május 10.

Tartalomjegyzék

ABSZTRAKT

Minden sportrajongóban valószínűleg felmerültek már azok a gondolatok, kérdések, hogy a közvetítők hogyan képesek ilyen gyorsan információt szolgáltatni a nézők felé. Miként képesek a pályára vetíteni különféle grafikai elemeket anélkül, hogy kitakarnák a játékosokat. Ezek a gondolatok bennem is megfogalmazódtak és ahogy egyre többet tanultam, egyre inkább szakmai kíváncsisággá kezdtek válni.
Témaválasztásnál egyből tudtam, hogy a gépi látás témakörével szeretnék foglalkozni, viszont ez önmagában nem volt egy jól fókuszált ötlet, ezért le kellett szűkítenem egy konkrét feladatkörre. Ez nem volt más, mint a jelenet újraépítése egy három dimenziós térben. Viszont a sport témakörét sem szerettem volna teljesen elvetni, ezért a feladatot pontosítva, a jelenetet úgy szeretném újraépíteni, hogy azon az emberek testtartása és egymáshoz viszonyított elhelyezkedése látszódjon.
A kutatást meg lehetett volna közelíteni többféleképpen. A mai modern rendszerekben a mélységbecslést mély tanulási módszerek segítségével számítják ki vagy pedig lézer alapú távérzékeléssel határozzák meg. Ezzel szemben én egy hagyományosabb megközelítés felé szerettem volna elindulni, ez pedig nem volt más, mint a sztereó látás témaköre.

A kutatás során kitérek a digitális képalkotás, képfeldolgozás és a gépi látás alapjaira, amik kulcsfontosságúak ahhoz, hogy a fő problémakört megfelelően megértsük. A sztereó látás témaköre több főbb lépést foglal magában. A kutatás során mindegyiket érinteni fogjuk így egy átfogó képet kaphatunk a folyamatról. Az irodalomkutatás második nagyobb témaköre a testrész-felismerési algoritmusok, amik segítségével képesek leszünk reprezentálni a jeleneten szereplő személyeket. Ki fogok térni az osztályozás, lokalizálás és szegmentálás fogalmára, valamint a tárgyfelismerési modellek általános működésére. Ezt követően a pózbecslést témakörének járok utána.

ABSTRACT

Every sports fan has probably already had thoughts and questions about how broadcasters can provide information to viewers so quickly. How they are able to project various graphic elements onto the field without obscuring the players. These thoughts also expressed themselves in me and as I learned more and more they have become a professional curiosity.
When choosing a topic I knew right away that I wanted to deal with the topic of machine vision but that in itself was not a well-focused idea so I had to narrow it down to a specific area. My original idea was to recreating a given scene in a three-dimensional space. However, I did not want to completely discard the topic of sports either so I would like to recreate the scene in such a way that the posture and relative position of the people can be seen.
The research could have been approached in a number of ways. In today's modern systems, the depth estimate is calculated using deep learning methods or determined by laser-based remote sensing. In contrast, I wanted to move towards a more traditional approach, and that was the topic of stereo vision.

In my research I will cover the fundamentals of digital imaging, image processing and machine vision which are key to a proper understanding of the main problems. The topic of stereo vision involves several main steps. We will touch on each of them during the research so we can get a comprehensive picture of the process. The second major topic of literature search is the body part recognition algorithms, which will enable us to represent the individuals in the scene. I will cover the concepts of classification, localization, and segmentation as well as the general operation of object recognition models. I will then follow up on the topic of pose estimation.

1. BEVEZETÉS

Szakdolgozatomban egy olyan rendszert szeretnék elkészíteni, ami képes emberek testrészeit detektálni és lokalizálni. A lokalizáció során nem csak kétdimenziós információkat szeretnék kinyerni a képből, hanem egy sztereó látási rendszer segítségével háromdimenziós mélységi adatokkal szeretném tovább javítani a pózbecslést végző algoritmust. Ez pedig lehetőséget ad arra, hogy egy háromdimenziós térben újra tudjam építeni a jelenetet.

A szakdolgozat témaválasztásánál egyértelmű volt számomra, hogy milyen irányba szeretném mélyíteni a tudásomat, ez pedig nem volt más, mint a képfeldolgozás és gépi látás témaköre. A téma iránti érdeklődésemet elsősorban a televíziós sportközvetítések váltották ki, azon belül is a különböző grafikus elemek megjelenítése a pályán vagy épp a játékosok körül. Az a kérdés is foglalkoztatott, hogy hogyan képesek szinte azonnali információkat leszűrni egy játékos helyzetéről vagy teljesítményéről, mint például a megtett távolság, átlagsebesség és az ezekhez hasonló adatok. A sporton kívül még más helyen is találkoztam közvetett módon ezekkel a technológiákkal, mégpedig az önvezető autók témakörénél. A témában kevésbé jártas emberekben megfogalmazódnak azok a kérdések, hogy hogyan képesek eligazodni az utakon, hogyan látnak, azon belül is hogyan tudják megkülönböztetni a tárgyakat az emberektől. Ezek mind-mind olyan dolgok voltak, amik növelték a szakmai kíváncsiságomat és végül ez a kíváncsiság vezetett a témaválasztásomhoz.

Célom, hogy az irodalomkutatás alapján egy olyan alkalmazás fejlesszek le, ami végig vezet a sztereó látási rendszer főbb lépésein és aminek végeredményeként egy három dimenzióban újraépített jelenetet kapunk.

2. IRODALOMKUTATÁS

Ahhoz, hogy el tudjuk kezdeni a rendszer tervezését, szükséges kitekintést tegyünk arra, hogy más rendszerek, milyen megközelítésekkel oldják meg ezt a problémát. A témakör átfogó megértése érdekében utánajárok az alapvető fogalmakra, amik később kapcsolódni fognak a tárgyalni kívánt algoritmusokhoz, technológiákhoz és az esetlegesen felmerülő problémákhoz, mellékhatásokhoz.

2.1. Hasonló rendszerek

A többkamerás rendszerek használata igen sokszínű, kezdve az autóipari fejlett vezetéstámogató rendszereken keresztül - amik kényelmesebbé és biztonságosabbá tehetik a vezetést -, egészen a kereskedelmi üzletekben a vásárlók mozgásának elemzéséig, amely segítségével javítható az üzleten belüli termékek elrendezése. Ebben a fejezetben két rendszert szeretnék tárgyalni, amik alapötletükben közel azonosak az általam választott témával és a mai modern technológiák jelenik meg bennük, mint például a felhő vagy pedig a neurális hálózatok.

2.1.1. Hawk-Eye

A Sony tulajdonában lévő Hawk-Eye [1] egy számítógépes látórendszer, amit 2001-ben fejlesztettek ki a különféle labdasport mérkőzések követésére, mint például krikett, tenisz, tollaslabda, labdarúgás vagy épp egy röplabda meccs, ahol a labda pontos pozícióját szerették volna meghatározni és ezekből akár következtetéseket levonni. Maga a rendszert hat nagy felbontású kamera alkotja, amik különböző helyeken és szögekből veszik a pályát és a rajta elhelyezkedő labdát. A videofelvételeket jelenetenként dolgozzák fel, minden kamera képén a rendszer azonosítja a pixelek azon csoportját, ahol feltehetőleg a labda elhelyezkedik, ez után összehasonlítja a kinyert adatokat az egyes kamerák között és háromszögelés segítségével határozza meg a labda adott képkockán elhelyezkedő fizikai helyzetét. Ezek a feldolgozott jelenetek végül felépítenek egy utat, amelyen a labda végig haladt. Ezen felül a Hawk-Eye képes megjósolni a labda jövőbeli repülési útvonalát is, illetve a várható leérkezésének helyét. A rendszer maga nem tökéletes, de 3,6 milliméteres pontossággal hirdetik [2] és sokan a bírók utáni második véleményként bíznak a működésében. A rendszert például a 2013-14-es Angol első osztályú labdarúgó-bajnokságban [4] használták, illetve a 2015-16-os Bundesligában [3].

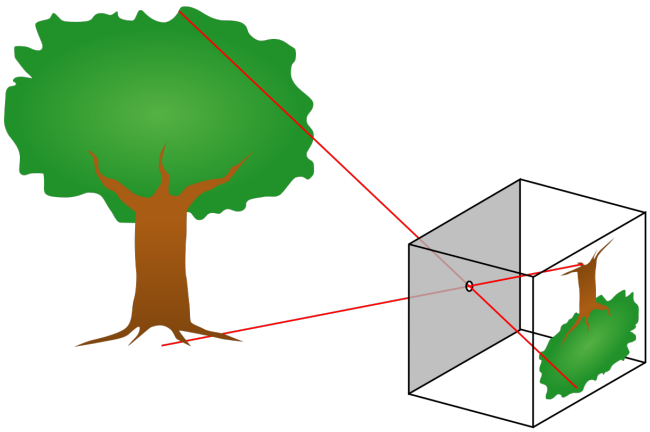
2.1.2. TRACAB

A ChyronHego [5] által kifejlesztett TRACAB [6] első generációját 2005-ben jelentették be, azóta számozásnál jelenleg a 2019-ben megjelent ötödik generációnál tartanak. A TRACAB egy optikai nyomkövető rendszer, amit elsősorban sportvilágban használnak. A legújabb generáció egy ultra-nagyfelbontású, elosztott kamera rendszert használ, ami a játéktér teljes részét lefedi, illetve egyes részek több szögből is látszódnak. A TRACAB rendkívül pontos adatokat képes szolgáltatni a pályán lévő játékosokról, bírókról, illetve a labda helyzetéről. A rendszer által rögzített adatokat felhőben tárolja, majd a feldolgozott adatot valós időben - a cég állítása szerint 0,15 másodpercen belül - szolgáltatja, így élő közvetítéseknél, grafikus elemek és statisztikák megjelenítésére alkalmazható. Ezenkívül az edzőknek és a menedzsereknek is rendkívül hasznos, mivel így adatok és statisztikák által személyre szabott elemzésekkel javíthatják a játéktechnikai tudás elsajátítását. Ezeken felül a rendszer mély konvolúciós hálózatokat is használ, ezáltal képes az egyes csapatok, játékosok, mezszámok felismerésére, illetve képes a játékosok testrészeinek nyomkövetesére. A ChyronHego rendszere már több száz stadionban bizonyított eredményességéről, mint például a 2014-ben átadott budapesti Groupama Arénában. [7] [8]

2.2. A digitális kép

A szakdolgozatomhoz, ha nem is közvetlen, de közvetetten tartozik a képalkotás témaköre, mivel kamerákkal dolgozunk. Később látni fogjuk, hogy a kamerák fizikai felépítéséből adódóan felbukkannak olyan kérdések, mint például, hogyan lehetséges az, hogy az elkészült képünk eltér a jelenettől amiről készült. Továbbá a kamera kalibráció és mélységbecslés témakörénél előkerülnek olyan fogalmak, amik szorosan a kamerák fizikai felépítéséből adódnak. Ebben a fejezetben a technikai részletek helyett csak dióhéjban szeretném bemutatni a témát az olvasónak.

Egyből talán nem is gondolnánk rá, de a képalkotás alapja a fény, azon belül is maguk a fényrészecskék, amiket fotonoknak hívunk. Ezek a fotonok amikor kölcsönhatásba lépnek egy tárgyal, akkor saját frekvenciájukat némileg megváltoztatva reagálnak többek között az adott tárgy felületének színére. Ha ezeket a visszapattanó fotonokat átengedjük egy kis nyíláson, akkor a nyílás mögött egy vízszintesen megtükrözött képet fognak alkotni arról a jelenetről, ami a nyílás előtt található. Ennek a leírására szolgál az úgynevezett lyukkamera modell (2.1



2.1. ábra.
Lyukkamera modell [9]

ábra), ami a háromdimenziós térben lévő pont koordinátái és a kamera képsíkjára levetített koordináták között teremt matematikai kapcsolatot. A valóságban egy digitális kamera nyílásánál egy (, de akár több) lencse- és egy rekesz helyezkedik el. A lencse feladata, hogy összpontosítsa adott távolságra a nyílásba beérkező fény fotonokat, ezzel egy tisztább, élesebb képet vetítve a kép síkjára. A rekesz alapértelmezetten zárt állapotban van, de amikor megnyomjuk a fényképezőgép gombját, akkor elhúzódik és ezzel lehetővé teszi, hogy fény jusson a lencsébe. Ez a folyamat hasonló ahhoz, ahogy a szemünk működik.

Az eddigiekben csak a jelenet képét juttattuk be a kamerába, de ezt a képet valahogy rögzíteni is tudnunk kell. Erre a célra szolgál a kamera szenzora, ami a lencse által levetített képet dolgozza fel oly módon, hogy az egyes fotonokat töltött elektronként rögzíti szilíciumban. Ez a szilícium egy rács pixeltömb, aminek elemei az úgynevezett pixelek. Ezek a pixelek fényérzékenyek, vagyis képesek érzékelni azt, hogy hány foton érkezett be rájuk. Különböző színek rögzítését más-más színű szűrőkkel oldják meg.

Ekkor a folyamat még nem ért véget, mivel ezeket a töltött elektronokat még nem képes értelmezni a számítógépünk, így ezeket a töltött elektronokat először feszültséggé alakítjuk át kondenzátorok és erősítők segítségével, ezzel egy analóg jelet kapva. Az ezt követő lépésben a jelet digitalizálnunk szükséges. A digitalizálás általában két műveletből áll. A mintavételezésből és a kvantálásból. A mintavételezés során az analóg jelből időközönként mintákat veszünk, ezzel diszkrét mennyiségeket kapva. A kvantálás során ezeket a mintaértékeket soroljuk

egy-egy értéktartományba, vagyis az amplitúdó nagyságokat diszkretizáljuk. Végeredményül pedig bináris számokat kapunk, ami már a számítógépek számára is értelmezhető.
A digitális kép tehát egy kétdimenziós tömbként is ábrázolható, ahol az egyes cellák diszkrét értékeket tartalmaznak, amik megfelelnek az adott pixel intenzitásának.

2.3. Képfeldolgozás és a gépi látás

Hagyományos megközelítésben a jelfeldolgozási technikákat értjük képfeldolgozásnak. A bemenet valamilyen folytonos kép és kezdeti célja a képek javítása, fokozása, hogy az emberek vagy algoritmusok számára értelmezhetőbb legyen. Modern megközelítésben a képhez valamilyen leírókat, jellemzőket próbálunk hozzárendelni, például a hasonló részeket megjelöljük, majd osztályozzuk *(milyen objektumok vannak?)*, kielemezzük *(hogy kapcsolódnak az objektumok egymáshoz?)*, legvégül pedig megpróbáljuk megérteni, hogy mit látunk *(miért?)*.
Három feldolgozási szintet különböztetünk meg. Alacsony szintről vagy szűkebb értelmezésű képfeldolgozásról, akkor beszélünk, hogy ha mind a bemenet és a kimenet is valamilyen egyszerű kép. Középszinten, a kimenet nem csak valamilyen módosított kép, hanem további leírókat nyerünk ki az eredeti képből, például, hogy hol helyezkednek el élek. Magas szinten, ezekből a leírókból következtetünk *(osztályozás, elemzés, megértés)*.

A digitális képfeldolgozás nem más, mint a digitális képek feldolgozása digitális számítógépekkel. Célja a képek fokozása, módosítása. Képjavítás, képfokozás során a bemenet és a kimenet is egy kétdimenziós kép.
A számítógépes gépi látás egy más terület, mint a digitális képfeldolgozás, de sok közös van a kettőben. Ide tartozik az osztályozás, elemzés, valami a megértés. Itt jutottunk el a szakdolgozat témájához, ahol a bemenet egy háromdimenziós környezetben lévő munkatér és ebből a munkatérből készítünk 2D képeket. A célunk, hogy az eredeti tartalomra visszakövetkeztessünk és/vagy valahogy értelmezzük a képet. Ez a folyamat történhet valós időben többkamerás rendszerek segítségével, valamint mesterséges intelligenciával is párosulhat. A gépi látást elég sok tényező tudja bonyolítani. Mivel a háromdimenziós világot képezzük le két dimenzióra, ezért csak redukált információ érkezik a környezetből. Az értelmezés lépései sem egyértelműek és ezen felül a kompozíció tartalmazhat sok oda nem illő elemet, zajt és kitakarásokat,

valamint nagyon sok adat feldolgozásához nagy számítási teljesítmény szükséges. Az esetünkben ilyen tényezők lehetnek egy sportesemény nézői vagy ha a játékosok egymást takarják ki, valamint az adatfeldolgozás teljesítménye túlnyomórészt a kamerák minősége (vagyis a felbontás és hogy hány képkockát képesek rögzíteni másodpercenként) fogja meghatározni. Az első tényezőre egy lehetséges megoldást nyújthat, ha a detektált objektumokat szűrjük távolság alapján. A második felvetésre pedig a kamerák száma, illetve a lehető legideálisabb elhelyezése adhat megoldást.

2.4. Jellemző pontok detektálása és illesztése

Ebben a fejezetben egy olyan fogalmat szeretnék ismertetni, ami elengedhetetlenek egyes algoritmusok - esetünkben elsősorban a kamera kalibráció - működéséhez.

2.4.1. Jellemzők

Jellemző pontok a kép megkülönböztethető struktúrái, mint például élek, sarkok vagy foltok. Invariáns lokális jellemzőknek nevezzük azokat a pontokat, pixeleket, amik egy átméretezett, elforgatott, eltolt képen vagy más fényviszonyok mellett is ugyan azt az értéket generálják. Célunk olyan jellemzők keresése, amik invariánsok az előbb felsorolt jellemzőkre és ezen felül jól meghatározott helyük van a képen, ezáltal az észlelés hatékonysága nő.

2.4.2. Jellemző pontok detektálása

A jellemzőpont detektálás legegyszerűbb modellje, ha kis ablakokat helyezünk a kép minden területére, majd az ablakot kis mértékben elmozdítjuk, lehetőleg minden irányba és megvizsgáljuk, hogy mekkora volt a változás mértéke. Ha homogén részen vagyunk vagy él irányában, akkor nem lesz változás, viszont, ha sarok ponton vagyunk akkor jelentősebb változást fogunk kapni minden irányban.

2.4.3. Jellemzőpont leírók

A jellemzőpont leírók olyan algoritmusok, amik bemenete maga a kép és kimenetük a képből kinyert jellemzőpont leírók/vektorok. A leírók olyan érdekes információkat kódolnak bele egy számsorba, amik egy numerikus ujjlenyomatként viselkednek, vagyis általa képesek leszünk megkülönböztetni az egyik jellemző pontot a másiktól. Kétféle típusú leíró létezik. Az első az úgynevezett lokális leíró,

ami tömören ábrázolja az adott pont helyi környezetét. Ábrázolás alatt azt értjük, hogy a pont szomszédságának alakjára, megjelenésére próbál információt adni. Ez a megközelítés kifejezetten alkalmas különféle illesztési feladatoknál. A másik típus, a globális leíró, a teljes képre vonatkozóan próbál információt adni.

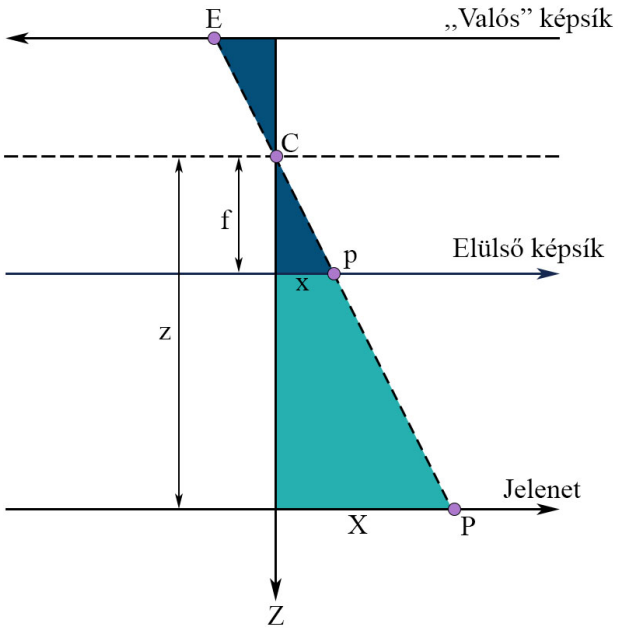
2.4.4. Jellemzőpontok illesztése

Ha illesztési feladatot szeretnénk végezni, akkor feltételezzük, hogy van két képünk, amiken már detektáltuk külön-külön a jellemző pontokat. A jellemzőpont illesztő algoritmusok fő feladata, hogy ezeket a jellemzőpontokat párosítsa össze egymással. Ez úgy lehetséges, ha a két kép leíróit összehasonlítjuk és azonosítjuk a hasonló jellemzőket. Egy összeillesztett pár $((x_1, y_1), (x_2, y_2))$, ahol az első képen lévő (x_1, y_1) jellemző pont párja a második képen lévő (x_2, y_2) pont. Az illesztés hatékonysága túlnyomórészt azon múlik, hogy milyen jellemzőpont detektáló és leíró algoritmust használtunk. Ha a problémának nem megfelelő algoritmussokkal álltunk neki a feladatnak, akkor az illesztés teljesítménye romlani fog. Ha a feladatunk egy sakktábláról készült két kép összeillesztése, akkor a jellemzőpont detektálást és leírást ne olyan algoritmussal végezzük, amit főként foltok, hanem olyannal, amit élek detektálására találtak ki.

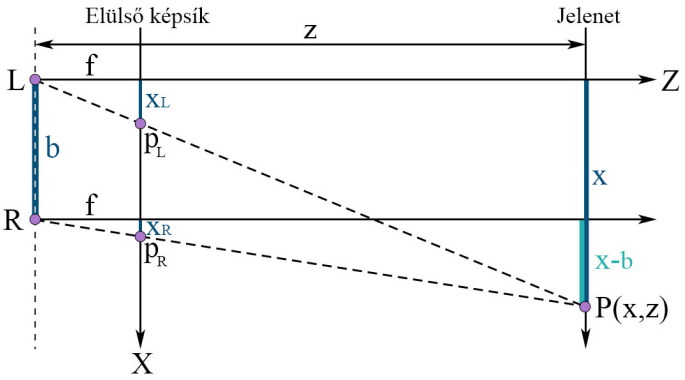
2.5. A sztereó kamera

A 2.2 fejezetben tettem egy olyan kijelentést, hogy a digitális kamera némileg hasonlóképpen működik, mint a szemünk. A sztereó kamerás rendszerek megértését is ezzel a megközelítéssel kezdeném. Azért vagyunk képesek a mélység érzékelésére, mert két közel tökéletesen egymáshoz igazított frontálisan-párhuzamos szemünk van, amelyek kissé eltérő helyzetből látják a világot. A két perspektíva közti eltérést nagyon könnyen megfigyelhetjük akkor, ha megnézünk közelről egy tárgyat először az egyik, majd a másik szemünkkel. Ezt a különbséget az agyunk automatikusan feldolgozza, így képesek vagyunk a háromdimenziós információk érzékelésére.
A sztereó kamerás rendszerek hasonlóképpen működnek. Ha van két idealizált kameránk, amik közt van legalább egy kis távolság és a két kamera képén a közös jellemző pontok egy síkba helyezkednek el, akkor képesek vagyunk a háromdimenziós jelenet szerkezetének visszanyerésére. A szakirodalomban a korábban említett eltérést diszparitásnak nevezzük, ami akkor alakul ki, amikor egy háromdimenziós pontot perspektivikusan levetítünk két különböző kamerára. Ha a jelenet minden

pontjára meghatározzuk ezt a diszparitást, akkor egy diszparitás térképet kapunk, ami már megjeleníthető, mint egy kép. Ez a diszparitás térkép később átalakítható a jelenet háromdimenziós térképévé, feltéve, ha a sztereó geometria ismert. A 2.2 ábrán látható egy háromdimenziós objektum tetszőleges pontjának kamera által levetített képe. Ennek a pontnak a háromdimenziós helyzete egyetlen egyenesre korlátozódik le, ami a kamera fókuszpontján **(C)** és a háromdimenziós objektum adott **(P)** pontján halad át.



2.2. ábra.
Perspektivikus vetület

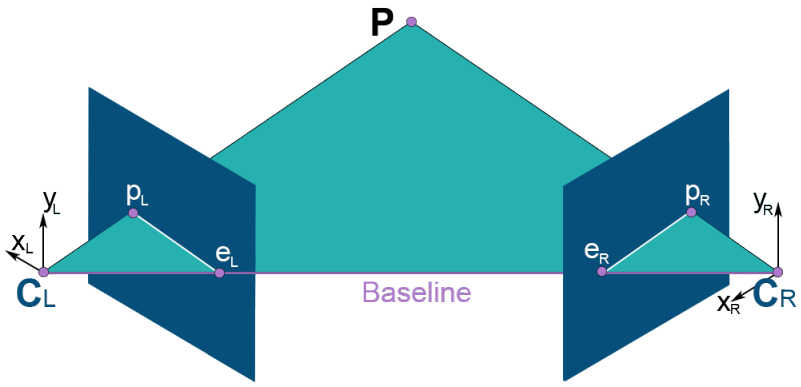


2.3. ábra.
Sztereó képek vetületei

Tegyük fel, hogy **P** pont mélységét szeretnénk meghatározni. Ehhez egy sztereó kamerás rendszert kell lemodellezzünk, ami a 2.3 ábrán látható, ahol a bal oldali kamera **(L)**, a jobb oldali kamera pedig **(R)** megjelöléssel szerepel. A háromdimenziós objektum vetületei a két kamera elülső képsíkjain p_l és p_r megjelöléssel szerepelnek. Vegyünk egy egyszerű esetet, amikor a két idealizált kamera optikai tengelye párhuzamos egymással és az Y tengely merőleges az oldalra. Ekkor a diszparitás **(d)** kiszámítható a következőképpen:

$$d = (p_l - f) - (p_r - f) = x_l - x_r \tag{2.1}$$

Egy sztereó látási rendszert alapvetően két probléma alkotja. Az első az egyeztetés problémája, vagyis meg kell találjuk az összes $(x_l, y_l), (x_r, y_r)$ jellemzőpont párt úgy, hogy az egyes párok ugyan arra a háromdimenziós pontra mutassanak. Ezeknek a pontoknak a helyes párosítása döntő fontosságú a végeredményre nézve, mivel nem kívánatos többértelmű megfeleltetések jöhetnek létre. Ennek a problémának megoldására szolgálnak a 2.4.4 tárgyalt jellemzőpont detektáló és illesztő algoritmusok, amik segítségével a $(x_l, y_l), (x_r, y_r)$ párokat képesek vagyunk meghatározni. A sztereó látási rendszerre, konkrétabban a $(x_l, y_l), (x_r, y_r)$ párok egyezőségére fenn áll egy szigorú megkötés, mégpedig az, hogy a két összepárosított pontnak ugyan arra az epipoláris vonalra kell esnie. Ezt a megkötést epipoláris kényszernek nevezzük és segítséget nyújt abban, hogy x_l, y_l párját ne kelljen az egész képen keresnünk, csak az epipoláris vonala mentén.



2.4. ábra.
Epipoláris sík és vonal

Ahhoz, hogy az epipoláris vonal fogalmát megértsük, meg kell ismerjük, hogy mi az az epipoláris sík. Az epipoláris sík (2.4 ábra), egy sík, ami a két kamera fókuszpontját

és a **P** pontot köti össze. Ez az epipoláris sík, a két képsíkot átvágva létrehoz egy-egy epipoláris vonalat. Így két pont illesztésénél ezen a vonalon kell keresnünk az egyezőségeket. Ez a megkötés akkor is fennáll, ha a két kamera optikai tengelye nem párhuzamos egymással, de ebben az esetben az epipoláris vonalak lehet már nem vízszintesen helyezkednek majd el. A sztereó látási rendszer második problémája az újjáépítéssel kapcsolatos, amikor az összepárosított pontok alapján számítjuk ki a kép mélységtérképet.

2.5.1. Sztereó rektifikáció

Az előző fejezetben láthattuk, hogy az epipoláris kényszer akkor is fennáll, ha az optikai tengelyek nem párhuzamosak. Erre ad megoldást a sztereó rektifikáció, ami egy korrekciós lépésként is tekinthető. Akkor nevezünk egy képet rektifikáltnak, amikor minden epipoláris vonal párhuzamos a kép vízszintes tengelyével, illetve minden egymásnak megfeleltethető pontnak a két képen azonos a függőleges koordinátája. Ezt legegyszerűbben úgy tehetjük meg, hogy úgy forgatjuk meg mind a két képet, hogy a két kamera fókuszpontját összekötő egyenes párhuzamosnak tűnjön. Ezt követően csavarjuk meg úgy az optikai tengelyeket, hogy a vízszintes tengelye minden képpontnak párhuzamos legyen a másik kamera fókuszpontjával. Legvégül pedig a kisebb képet méretezzük úgy át, hogy a két vonal egymásba essen.

2.6. Kamera kalibráció

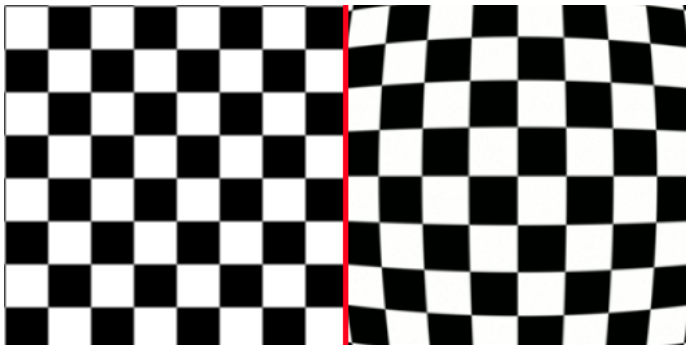
2.6.1. Homogén koordináták és transzformációk

A homogén koordináták egy nagyon hasznos és alapvető gondolata a számítógépes grafikának, illetve számítógépes látásnak. A homogén koordináták az n dimenziós tér egy pontjának helyzetét n+1 koordináta segítségével írják le. Ha van egy $P = (x, y)$ pontunk a derékszögű koordináta rendszerben, akkor annak a homogén koordináta rendszerében található megfelelője a $P_{hom} = (x \cdot w, y \cdot w, w)$ pont, ahol a **w** egy nem nulla tetszőleges szám. Mivel **w**-nek egy tetszőleges számot választhatunk, ezért **P** pontnak számtalan sok homogén koordinátája lehet, ezért szokás **w**-nek egy állandó értéket választani, esetünkben az 1-et, mivel így a homogén és az n dimenziós tér koordinátái megegyeznek. Ekkor $P_{hom} = (x, y, 1)$ a **P** pont normalizált homogén koordinátája. A homogén koordinátás leírás által, minden geometriai transzformációt mátrix műveletek segítségével hajthassunk végre, több egymás után végrehajtandó transzformáció eredőjét egy transzformációs mátrixba foglalhatjuk

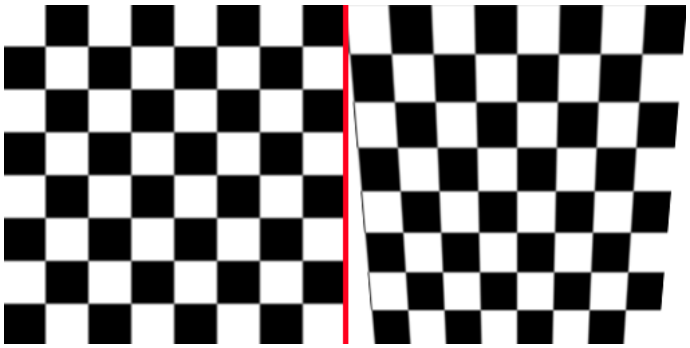
össze. Azért fontos számunkra a homogén koordináták ismerete, mivel a kamera kalibrálásnál is jelen van, ahol egy olyan transzformációt akarunk majd meghatározása, amely tetszőleges pontot transzformál az egyik képről a másikra, illetve a kamera belső paramétereinél is megjelenik perspektív transzformáció formájában, amikor három dimenzióról képzünk le két dimenzióra.

2.6.2. Torzítás

A 2.2 fejezetben láthattuk, hogy hogyan működik nagy vonalakban egy digitális kamera. Az elkészült képet nagy mértékben befolyásolhatja a kamerában elhelyezkedő lencse/lencsék minősége. Ahhoz, hogy reprezentálni tudjunk egy valódi kamerát, a modellünket ki kell bővítsük egy sugárirányú és egy érintőleges lencse torzítással.



2.5. ábra.
Sugárirányú lencse torzítás



2.6. ábra.
Érintőleges lencse torzítás

Sugárirányú lencse torzítás (2.5 ábra) akkor jön létre, amikor a lencse formája tökéletlen, vagyis a lencse külső élei kifelé (de akár befelé) görbülnek és ennek az lesz a következménye, hogy a kamera egy halszem-képet hoz létre. Az érintőleges lencse torzítás (2.6 ábra) akkor következik be, amikor a lencse nem párhuzamos

a képi síkkal, vagyis, amikor a fényérzékelő- és a lencse középpontja nem igazodik egymáshoz tökéletesen, ezzel egy kissé ferde nézetet hozva létre. Ezeket a torzításokat együtthatókkal tudjuk megszüntetni, amiket a kamera kalibrálás során határozunk majd meg.

2.6.3. Kamera paraméterek

Kamera paraméterek segítségével vagyunk képesek a jelenet háromdimenziós szerkezetének visszaállítására. Paraméterek típusát tekintve két csoportra bonthatjuk őket, külső- és belső paraméterekre.
A külső paraméterek határozzák meg a kamera koordináta rendszerének elhelyezkedését és irányultságát az ismert háromdimenziós világ koordináta rendszeréhez viszonyítva. A külső paramétereket két fő komponens alkotja. Az egyik a forgatási mátrix **(R)**, ami a kamera koordináta rendszerének elforgatását írja le a világ koordináta rendszeréhez képest. A másik az eltolásvektor **(t)**, ami a kamera koordináta rendszerének eltolását írja le a világ koordináta rendszeréhez képest.
A belső paraméterek teremtenek kapcsolatot a kép koordináta rendszere és a kamera koordináta rendszere között. Ahhoz, hogy ezeket a belső paramétereket megkapjuk egy pár lépésben le kell vezessük ezt a projektív transzformációt. Egy kamera koordináta rendszerben található $P = (X, Y, Z)$ pontot szeretnénk átalakítani a kép koordináta rendszerében található $P' = (X', Y')$ megfelelőjévé. Legyen az X és Y irányokban a fókusztávolság F_x és F_y. A fókusztávolságot általában milliméterben, centiméterben vagy hüvelykben adják meg, de használhatunk egy torzítási együtthatót **(s)** is, amely az egységnyi hosszra eső pixelek számát jelenti. Ezeket felhasználva legyen $f_x = s \cdot F_x$ és $f_y = s \cdot F_y$.

$$\text{Ekkor:} \quad \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{f_x \cdot X}{Z} \\ \frac{f_y \cdot Y}{Z} \end{bmatrix} \tag{2.2}$$

Az érintőleges torzítást figyelembe véve, a kép középpontja kismértékben el van tolva a kamera koordináta rendszerének origójához képest. Továbbá a kép origója a kép koordináta rendszerben a bal felső sarokban helyezkedik el, ezért át kell transzformáljuk a képet egy $(c_x, c_y)^T$ vektor segítségével.

$$\text{Ekkor:} \quad \begin{bmatrix} x \\ y \end{bmatrix} = \begin{bmatrix} \frac{f_x \cdot X}{Z + c_x} \\ \frac{f_y \cdot Y}{Z + c_y} \end{bmatrix} \tag{2.3}$$

Ha minden koordinátát átalakítunk homogén koordinátákká, akkor megkapjuk az alábbi egyenletet.

$$\text{Ekkor:} \quad p = \begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix} = K \cdot P \tag{2.4}$$

Ahol **K** mátrixot a kamera belső mátrixának hívják és f_x, f_y, c_x, c_y a kamera belső paraméterei.

2.6.4. Kalibráció

Ideális kamera esetén nem lenne szükségünk kalibrációra, mivel a jelenet és az ideális kamera által rögzített kép azonos lenne. Egy valódi kameránál láttuk, hogy sokkal több tényező befolyásolja a végeredményt, amiket egy lyukkamerás modell nem tartalmaz. Ilyen tényező lehet például a lencse és az emiatt létrejövő 2.6.2 ismertetett torzítások, vagy a kamerába szerelt eszközök minősége, mint például a szenzor. Ahhoz, hogy megközelíthessük ezt az ideális képet át kell transzformálnunk a nyers képet a kamera belső paraméterei és a torzítási paraméterek segítségével.
A kamera kalibráció az a folyamat, amikor a kamera belső paramétereit számítjuk ki. Ahhoz, hogy ki tudjuk számolni a belső paramétereket, szükségünk van a torzítási paraméterekre. Ámde ahhoz, hogy ki tudjuk számítani a torzítási paramétereket, szükségünk van a kamera belső paramétereire. Az egyik megoldás a kalibrálásra, ha a torzítási paramétereket kezdetben nullának tekintjük, majd kiszámítjuk a belső paramétereket és ezek segítségével kiszámítjuk a torzítási paraméterek értékét, majd ezt a folyamatot többször megismételjük.
A kalibrációs folyamat bemutatására Zhang [10] által javasolt technikát fogom bemutatni, amit az OpenCV-ben [13] is használnak. A módszer alapelve, hogy szükségünk van egy kalibrációs objektumra, amelynek jellemzőpontjainak koordinátái ismertek. A legnépszerűbb ilyen objektum egy sakktábla minta, amit egy síkfelületre rögzítenek és a sakktáblán található belső négyzetek sarkai jelentik a jellemzőpontokat. Hiába vagyunk térben, mivel a sakktábla sarokpontjai egy x-y síkra esnek, ezért a z koordináta értéke mindig nulla. Ezzel pedig definiálni tudjuk a véges számú sarokpontok koordinátáit, mint $(1,1,0)^t$, $(1,2,0)^t$, $(1,3,0)^t$, $(1,4,0)^t$ stb. Ezután azzal a kamerával, aminek ki szeretnénk számolni a belső paramétereit, egy tetszőleges nézetből készítünk egy képet erről a sakktábláról. Ekkor három koordináta-rendszerünk lesz, egy a sakktáblának, egy a kamerának

és egy a képnek. Ezen a nyers képen elvégzünk egy jellemzőpont detektálást, vagyis meghatározzuk a nyers képen a belső sarokpontok helyzetét, majd ezeket a detektált koordinátákat össze tudjuk vetni a sakktábla valós koordinátáival. Összehasonlítás alatt egy úgynevezett projektív megfeleltetést értünk, amikor azt a transzformációt szeretnénk meghatározni, ami a nyers és a valós koordináták között fellelhető. Ezt a transzformációs mátrixot hívjuk homográfiának **(H)**. A 2.6.3 alfejezetbe már láthattuk, hogy a kamera belső és külső paraméterei milyen formában teremtenek kapcsolatot a koordináta rendszereink közt. Ezt a tudást felhasználva pedig a homográfia és a kamera belső mátrixa között egy kapcsolatot tudunk meghatározni. Amikor tetszőleges nézetből készítünk erről a sakktábláról egy képet, akkor különféle megkötéseket, ismeretleneket tudunk kinyerni. Lesznek olyan ismeretlenek, amik állandóak attól függetlenül, hogy milyen nézetből készítettük a képet, ezek az ismeretlenek lesznek a kamera belső paraméterei. Ezért is van arra szükség, hogy több nézetből készítsünk a sakktábláról képeket, mert több egyenletrendszert kell tudnunk felírni, mint amennyi ismeretlenünk van. Miután meghatároztuk a kamera belső paramétereit, már képesek leszünk kiszámítani a torzítási paramétereket is.
Természetesen az eredeti célunk változatlan maradt, vagyis a nyers képet úgy feljavítani, hogy jobban hasonlítson arra képre, mint amit egy idealizált kamerával készítettünk volna. Mivel minden paramétert sikeresen meghatároztunk, ezért ezeket felhasználva a nyers képet torzítatlanná tehetjük. [11] [12]

2.7. OpenCV

Amikor képfeldolgozás témakörben tevékenykedünk talán a legelső kifejezés, amivel szembe kerülünk az az OpenCV. Az OpenCV egy hatalmas nyílt-forráskódú könyvtár, amit gépi látásra, gépi tanulásra és képfeldolgozási feladatokra használnak fel. A sztereó kamerával kapcsolatos komponensekben én is ezt a könyvtárat fogom felhasználni, megoldható lenne, hogy mindent saját kezűleg fejlesszek le, de az OpenCV által nyújtott függvények a több éven át tartó optimalizálásoknak köszönhetően nagyon jó teljesítménnyel bírnak, így a fejlesztést felgyorsítása és az alkalmazás teljesítménye érdekében az OpenCV által nyújtott funkcionalitásokat fogom alkalmazni.

2.7.1. Használható programozási nyelvek

OpenCV különféle programozási nyelveket támogat, a legismertebbek ezek közül a MATLAB, C++, C# és a Python [25].
A MATLAB esetében számunkra a hátrányok nehezebb súllyal bírnak, mint az előnyei, mivel hiába van szinte minden feladatra egy-egy eszköztára, de maga a szoftver és valamennyi komponense fizetős. Ezeken felül a nyelv szintaxisa merőben eltér az előzőleg felsorolt nyelvektől, így tanulása nehezebb és ha nem vagyunk teljesen tisztában a nyelvvel, akkor nem meglepő, hogy a végeredmény egy elég lassú kód lesz, annak ellenére, hogy a beépített eszköztárak általában jó teljesítménnyel bírnak.
C++ nyelv használata és az OpenCV jelentős része is ingyenes, valamint hatalmas könyvtár készlettel rendelkezik, amik nagyon jól optimalizáltak és a nagy fejlesztői közösségnek köszönhetően nagyon sok segítség érhető el az interneten a nyelvvel és annak OpenCV részének használatával kapcsolatban. További előnye, hogy készíthetünk vele asztali alkalmazásokat és webes alkalmazások backend-jében is használható, valamint telefonos applikációkban is működőképes lehet. A nyelvből kifolyólag, a fejlesztés folyamata lassabb és azoknak, akiknek nincs kellő tapasztalata a nyelvvel, nem csak lassabb, de nehezebb is lesz a fejlesztés, valamint a gépi tanulást csak minimálisan támogatja.
Python talán legnagyobb előnye, hogy könnyen használható és könnyen elsajátítható, valamint a fejlesztés vele nagyon gyors és gördülékeny. Köszönhetően az erőteljes támogatásnak a gépi látás és gépi tanulás terén, a Python vált a tudományos számítások nyelvévé. Egyik hátránya, hogy rosszabb teljesítményt tudunk elérni, mint egy C++ alkalmazás esetén.
C# eseténben OpenCV helyett EmguCV-ről beszélünk, ami egy cross platform .NET wrapper az OpenCV könyvtárnak, így bármilyen .NET kompatibilis nyelvből elérhetőek lesznek az OpenCV osztályai és algoritmusai, mint például Unity-ben.
A felsorolt nyelveknek mind megvannak a maga használati esetei. MATLAB lehetőségét a nyelvbeli eltérés és a licence költségek miatt egyből elvetem, valamint hiába az elérhető legjobb teljesítményt a C++ adná, de a fejlesztés felgyorsítása miatt erről a nyelvről is lemondok, így több idő jut más technológiák körbejárására és a meglévőknél a legjobb végeredmény elérésére. C# és Python előnyei és hátrányai szinte azonosak, közössége és dokumentációja szintén közel hasonlóak. Egy nagyon erős érv szól a Python mellett, mégpedig, hogy sokkal jobban támogatja a gépi

látás és gépi tanulást, ami a feladatom pózbecslési komponensénél elengedhetetlen lesz, így a továbbiakban a Python OpenCV könyvtárát fogom tárgyalni. [14]

2.7.2. OpenCV verziók

OpenCV-be jelenleg három fő verzió érhető el: 2.X, 3.X és 4.X. Ezeken belül természetes további al-verziók is érhetőek, de csak nagy vonalakban szeretnék a lényegesebb különbségekről beszélni.
A 2.X és 3.X verziók között a főbb különbségek, ami számunkra is döntésértékű lehetnek, hogy 3.X-be már nem elérhetőek olyan algoritmusok, mint pl. a FREAK, BRIEF, SIRF, SURF. OpenCV-ben jelenleg a fejlesztések a 4.X-be kerülnek, így többek között támogatja például a G-API-t vagy a cuDNN-t támogatás és alapvetően is több C++ 11 funkciót támogat. Ezzel szemben a 3.X egy stabil kódág, ahova csak hibajavítások kerülnek, így a többéves optimalizálásnak köszönhetőek egy kicsivel jobb teljesítmény érhető el vele, mint a 4.X-el. Mivel számunkra egyelőre nem látom szükségességét az új funkcióknak és a jellemzőpont detektálásra 3.X-be a többi algoritmus helyére bekerült az ORB, így 3.X verziót teljes értékűnek érzem és a fejlesztést is ebben fogom végezni.

2.8. Sztereólátási függvények OpenCV-ben

Ebben a fejezetben be szeretnék mutatni néhány fontosabb OpenCV függvényt, amiket a közeljövőben fel fogok használni főként a sztereó kamerákkal kapcsolatos komponenseknél. A függvényeket az OpenCV dokumentációja [15] alapján fejtem ki.

2.8.1. VideoCapture

Ezzel a metódussal fogom rögzíteni a kalibrációhoz szükséges képeket/videókat bal-, jobb- és sztereókamera esetében egyaránt. A metódus bemeneti paramétere egy *index*, ami a gépre kötött eszközök azonosítására szolgál. Ezzel a paraméterrel tudjuk szabályozni, hogy melyik oldalhoz, melyik kamera tartozik.

2.8.2. findChessboardCorners

Ezt a metódust kamera kalibrációjánál fogjuk használni, ahol, majd a bemeneti képen próbálja meghatározni a sakktáblán szereplő belső sarokpontok helyzetét. Ha megtalált minden ilyen belső sarokpontot és sorba tudta rakni őket, akkor a

kimenete egy nem negatív szám. A sorba rendezést a bemenetként megadott minta alapján próbálja végrehajtani minden sorban, balról-jobbra haladva. Ha nem talált sarokpontokat vagy nem tudta összerendezni őket, akkor a visszatérési értéke nulla. A metódus bemeneti paramétere egy kép, a keresendő minta, valamint egy *flag*, ami a metódus működésén változtat, például, hogy adaptív küszöbölést használjon vagy normalizálja a képet a küszöbölés előtt. Kimenete a lokalizált sarokpontok helyzetét tartalmazza.

2.8.3. calibrateCamera

Ez a metódus határozza meg a kamera belső- és külső paramétereit. A metódus a 2.6.4 alfejezetben tárgyalt, Zhengyou Zhang által javasolt technikát valósítja meg.

Bemeneti paraméterei a következők:

- Az objektum pontokból álló rendezett lista, ami a sakktábla belső sarokpontjait írja le a világ koordináta rendszerében. Ezek a pontok állandók, előre le kell generáljuk őket, valamint mivel egy síkban helyezkednek el, ezért a Z koordináta nullának vehető.
- Egy rendezett lista, ami a kép koordináta rendszerében lokalizált belső sarokpontok helyzetét tartalmazza.
- A bemeneti kép szélessége és magassága.
- Egy kritérium érték, amit, ha elér az iteratív optimalizálási algoritmus, akkor leáll. Az optimalizáció során az újravetítési hibát próbálja minimalizálni.

Kimeneti paraméterei az alábbiak:

- A kamera belső paramétereit tartalmazó 3x3-as mátrix. Ez nem csak kimenetként szolgálhat, hanem valamennyi *flag* paraméter megköveteli a mátrixon belül található néhány/összes érték megadását.
- A torzítási együtthatókat tartalmazó (4, 5, 8, 12, vagy 14 elemből álló) vektor. Ez szintén szerepelhet bemenetként a *flag* paraméter alapján.
- A kamera külső paramétereit alkotó forgatási-, illetve eltolási vektorok.
- Egy vektor, ami az egyes nézetekre vonatkozó újravetítési hibákat tartalmazza. Az újravetítési hiba, a lokalizált belső sarokpont koordináták, illetve a nekik megfelelő objektum pontok közti négyzetes távolságok összege.
- Egy *flag* paraméter, aminek értéke lehet nulla, vagy a lehetséges *flag* paraméterek kombinációja.

2.8.4. stereoCalibrate

Ezzel a metódussal fogjuk kalibrálni a sztereó kamera rendszert. A metódus meghatározza kamerák belső paramétereit, illetve a kamerák közti külső paramétereket. Valamennyi bemeneti paramétere megegyezik a *calibrateCamera* metódus során tárgyalt kimeneti paraméterrel, viszont ezeket mind a két kamerára vonatkozóan meg kell tudjuk adni.

Kimeneti paraméterei az alábbiak:

- Egy forgásmátrix, illetve egy eltolásvektor. Ezek közösen letudják képezni az első kamera koordináta rendszerében szereplő pontokat a második kamera koordináta rendszerében szereplő pontokra.
- Egy fundamentális mátrix, ami az epipoláris geometria leírására, reprezentálására szolgál.
- Egy esszenciális mátrix, ami a sztereó képek megfelelő pontjait kapcsolja össze.

2.8.5. stereoRectify

Ez a metódus számítja ki a sztereó kamera képeinek helyreigazításhoz szükséges paramétereket. Mivel szorosan összefüggő lépések, ezért ennek a metódusnak a bemenete szintén építkezik az előzőleg tárgyalt metódus valamennyi kimenetére.

További bemeneti paraméterei:

- Egy alfa paraméter, ami egy nulla és egy közti szám. A nulla jelenti azt, hogy a korrigált képet felnagyítja és eltolja oly módon, hogy csak érvényes képpontok maradjanak a képen. Ez a gyakorlatban azt jelenti, hogy a korrigálás során létrejött fekete keretet levágja a képről. Ha értéke egy, akkor megtart minden képpontot, valamint, ha nem adjuk meg vagy pedig mínusz egyre állítjuk, akkor egy alapértelmezett méretezést fog használni.
- Megadható a korrigálás utáni kép felbontása is, aminek, ha egy nagyobb értéket választunk, akkor segít megőrizni a részleteket az eredeti képen, pláne, hogyha a szimmetrikus lencse torzítás jelentős.

Kimeneti paraméterei a következők:

- Az első-, illetve második kamerára korrigálására szolgáló forgásmátrixok. Ezek a mátrixok a korrigálatlan (torzított) és korrigált koordináta rendszerek közti transzformációt írják le.
- Egy mátrix, ami segítségével a diszparitást tudjuk leképezni mélységgé.

Q egy 4x4-es perspektíva transzformációs mátrix, **flag** bekapcsolásával a két kép főbb pontjai egy koordinátára fognak esni, az **alpha** pedig egy méretezési paraméter, ha nincs megadva vagy -1, akkor alapértelmezett méretezést használ. 0 és 1 közötti értéket tudunk megadni, 0 esetében csak érvényes pixelek láthatóak, nincsenek fekete részek, ha pedig 1, akkor minden pixel az eredeti képről megmarad.

2.8.6. StereoSGBM

Ennek az osztálynak a felparaméterezésével fogjuk tudni kiszámítani a különbségtérképünket. Heiko Hirschmuller [16] 2005-ben bemutatott ötletén alapul, de néhány eltéréssel valósítja meg. Ilyen eltérések például, hogy tartalmaz elő és utófeldolgozó lépéseket, illetve az illesztés sem pixelenként történik, hanem blokkokban.

Az osztály létrehozásánál megadható paraméterek:

- A megengedett legkisebb eltérés, amit még keres. Olyan esetben, amikor a tárgyak távolabb helyezkednek el a kameráktól, akkor ezt érdemes kisebbre állítani, amikor pedig közel, akkor nagyobbra. Negatív értékeket, akkor érdemes használni, amikor a kamerák keresztbe vannak állítva. Alapértelmezett értéke a nulla.
- A tartomány, amiben az eltéréseket kell keresnie. Ez gyakorlatilag a jelenet mélységélessége, vagyis egy távolság a legközelebbi és a legtávolabbi tárgy között. A jelenlegi implementációban ennek az értéknek tizenhattal osztható nak kell lennie.
- Az illesztés során használt blokk mérete. Ennek egy páratlan számnak kell lennie.
- A többi paraméter az elő és utófeldolgozást szabályozza. Csökkentik a zajt a bemeneti, illetve a már kiszámolt különbségtérképben.
- Egy *mode* paraméter segítségével szabályozhatjuk, hogy milyen algoritmust használjon a különbségtérkép meghatározására.

2.9. Tárgyak felismerése és szegmentálása

Itt elérkeztünk a szakdolgozat másik nagyobb komponenséhez, amikor a sztereó képeinken detektálni szeretnénk az embereket. Annak érdekében, hogy erről a témáról is egy átfogóbb képet kapjunk többféle módszert, technológiát szeretnék bemutatni.

Erre a problémára egy lehetséges megoldást az osztályozás nyújtja, aminek bemenete egy kép és kimenete a felismert tárgy címkéje. Ez egy adat vezérelt megközelítés, vagyis begyűjtünk egy adatkészletet, amik képekből és címkékből állnak és gépi tanulás segítségével feltanítunk egy modellt erre az adatkészletre. Ezt követően teszteljük ill. kiértékeljük a modellünket. A gyakorlatban egy jó osztályozónak a tanítása lehet egy lassú és erőforrás igényes folyamat, de a használatának gyorsnak kell lennie kis erőforrásigénnyel, így akár a telefonunkon vagy böngészőnkön is képesek leszünk alkalmazni.

Ha ezen kérdésen túl azt is meg szeretnénk becsülni, hogy ez az osztályozott tárgy a képen belül hol található, akkor már osztályozás + lokalizálás módszeréről beszélünk. Ekkor egy osztályozó döntést kell meghoznunk, illetve a lokalizáció során egy határoló dobozzal szeretnénk meghatározni az objektum helyét a képen. Ennél a módszernél egy fontos megkötés, hogy pontosan egy objektum esetén ad teljes értékű megoldást, a többit figyelmen kívül hagyja.

Ha ezt az osztályozást és lokalizálást több objektumra szeretnénk elvégezni, akkor erre a célra különféle tárgyfelismerési modelleket tudunk alkalmazni. Itt is vannak előre meghatározott címkéink és ha ezek közül a kategóriák közül valamelyik feltűnik a képen, akkor egy dobozt rajzolunk köré, illetve ellátjuk a saját címkéjével. Az osztályozás+lokalizálásnál láttuk, hogy csak egyetlen objektumra működik, ez a módszer azon felül, hogy több objektum esetében is teljes értékű megoldást ad, akkor is használható, ha nem tudom, hogy hány objektum lesz a képen.

A következő továbbfejlesztés az lehetne, hogy azon felül, hogy meghatározom a címkéket és a határoló dobozokat, azt is megszeretném kapni, hogy ezen határoló dobozon belül melyik pixel tartozik az objektumhoz. Ezt a problémát szemantikus szegmentálásnak nevezik, ekkor pixel szerinti osztályozást és címkézést végzünk, viszont a kimenet nem tesz különbséget az egyének között, vagyis, ha a képen van két ölelkező emberünk, akkor a módszer nem tesz különbséget az egyének között, hanem egy címke alá fogja vonni a kettőt és egy összetartozó pixel halmazt kapunk. Ha az azonos egyének között is különbséget szeretnénk tenni, akkor már

példányszegmentálásról beszélünk. Az előzőben definiált módszert egy szinttel továbbfejleszti, ez is pixel szerinti osztályozást végez, de itt egyénenként próbáljuk meghatározni az objektumokat. Ezt úgy is ellehet képzelni, mint a tárgyfelismerés és szemantikus szegmentáció módszerek ötvözete, vagyis több objektumot tudunk kezelni, eltudjuk különíteni az egyéneket és mindezt képpontonként definiálhatjuk.

2.9.1. A modellek általános működése

A tárgyfelismerés korszerű megközelítésévé a mély tanulási módszerek váltak. A fő ok arra, hogy miért nem lehet a problémát megközelíteni egy szabványos konvolúciós hálózattal az-az, hogy a kimeneti réteg hossza változó, nem konstans. Ez azért van, mert a minket érdeklő objektumok előfordulásának száma nem rögzített.
A problémára egy naiv megoldás lenne, hogyha vennénk a képről különféle régiókat és arra használnánk fel a modellt, hogy ezeken a régiókon belül végezzen osztályozást. Ez a megközelítés azért nem célszerű, mert a különböző objektumok térbeli elhelyezkedése és méretaránya nem azonos, ennélfogva rengetek régiót kellene kiválasztani, ami számításigény szempontjából jelentene akadályt.
Ezt a brute force módszert cserélte le az R-CNN-ben [22] található szelektív keresés, aminek eredményeként 2000 régiót nyerünk ki, amit régió javaslatoknak neveztek el a szakirodalomban, viszont ez a szelektív keresés még mindig egy lassú és időigényes folyamat, ennek kiváltására a Faster R-CNN-ben [23] adtak megoldást, ahol elhagyták a szelektív keresési algoritmust és a régió javaslatok azonosítására egy különálló hálózatot hoztak létre, aminek segítségével a modell képes megtanulni a régió javaslatokat. Az így kapott algoritmus pedig már valós időben végzendő feladatokra is használható.
Egy alternatív megoldás, ami egy aranyközépút az előző két módszer között az az úgynevezett egylövéses detektálás, amikor ahelyett, hogy egy külön hálózatot használnánk a régió javaslatok azonosítására, javaslunk pár előre meghatározott határoló dobozt, amik egy pont körül helyezkednek el és ezeken a határoló dobozokon belül becsüljük meg annak a valószínűségét, hogy ott található e objektum vagy sem és eszerint módosítjuk a határoló dobozok helyzetét és méretét. Előfordulhat, hogy egy lehetséges detektálásra több közel azonos eredményt kapunk, mivel egy horgonypont körül több határoló dobozunk van, valamint ezek a horgonypontok túl közel is kerülhetnek egymáshoz. Ezért egy utófeldolgozásra van szükségünk, ahol a lehetséges eredményekből a legjobbat választjuk ki. Erre a legnépszerűbb módszer a nem maximumok elnyomása. Ilyen megközelítést használ például a YOLO. [17]

[18]

2.10. Pózbecslés

Pózbecslés során személyek testtartásaira próbálunk következtetéseket adni. Ezt a testtartást úgy tudjuk kinyerni, hogy különféle kulcspontokat definiálunk az emberi testen és ezeket a kulcspontokat próbáljuk meg detektálni. Kulcspont lehetnek a lábfejek, bokák, térdek, csípő jobb- és bal része, törzs alsó-, középső-, felső része, vállak, könyökök, csuklók, kézfejek, nyak töve, fej közepe, szemek, fülek vagy az orr. Ha ezeket a kulcspontokat összekötjük, akkor egy csontvázat kapunk meg, az adott jelenet, adott személyének testtartásáról. A pózbecslésre kétféle megközelítést alkalmaznak, de mindkettőre igaz, hogy mély tanulási architektúrát használ, ami különféle konvolúciós ideghálózatokból épül fel.
Az első megoldás egy alulról-felfelé történő megközelítés, ahol a modell először a kulcspontokat keresi meg, majd ezeket megpróbálja összecsoportosítani.
A második megoldás az első ellentéte, a felülről-lefelé történő megközelítés, amikor a hálózat először detektálja és lokalizálja az embereket határoló dobozokkal, majd ezen régiókon belül keres kulcspontok után.

2.10.1. A modellek általános működése

A pózbecslés segítségével képesek vagyunk nyomon követni egy vagy több ember helyzetét a kép koordináta rendszerében, de léteznek olyan modellek is, amik a világ koordináta rendszerében is el tudják helyezni a detektált kulcspontokat, de ez egy jóval nagyobb kihívást jelentő probléma. Különféle modellek különféle megközelítésekkel állnak a problémához, de általánosságban mindegyiknek az első lépése, hogy a bemeneti képen jellemző pontokat keresnek. Az ezt követő lépes az adott modelltől függ.
A legegyszerűbb módszer, ha regresszió segítségével megpróbálja megbecsülni a kulcspontok helyzetét. Ekkor a modell kimenete a detektált kulcspontok koordinátája.
Egy kicsivel nehezebb megközelítés, ha a modell egy kódoló-dekódoló architektúrát használ. Ahelyett, hogy azonnal megpróbálná megbecsülni a kulcspontok koordinátáit, a modell egy hőtérképet hoz létre. A hőtérképen belüli értékek annak valószínűségét írják le, hogy az adott régióra mennyire valószínű, hogy kulcspontot tartalmaz. Ezt a lépést egy utófeldolgozás követi, amikor a modell a hőtérkép

legnagyobb valószínűségeit tartalmazó régióit választja ki. Ha a hőtérkép több magas valószínűségű régiót tartalmaz, azaz például több jobb kezet talált, akkor egy további utófeldolgozó lépést kell bevezetni, ahol az egyes kulcspontok csoportosításra kerülnek attól függően, hogy melyik személyhez tartoznak. A fentről-lefelé történő megközelítésnél nem a teljes képre készül hőtérkép, hanem az egyes határoló dobozokon belül, így könnyebbé válhat a feladatnak azon része, amikor hozzá kell rendelni a kulcspontokat az emberekhez.
Ismertebb pózbecslési modellek például a Mask R-CNN [24], ami a kódoló-dekódoló architektúrát használja és képes több ember testtartásának detektálására. A modell egyik előnye, hogy nem csak határoló dobozokkal közelíti az embereket, hanem pixel pontosan próbálja meg szegmentálni az egyéneket egymástól. Ezen kívül például az OpenPose és PoseNet is a kódoló-dekódoló architektúrát használja, viszont a hőtérképen kívül rövid, közepes és nagy hatótávolságú eltolásokat határoz meg, amik segítségével a hőtérkép egyes régióin belül a detektálás pontossága javítható. [19] [19]

2.11. Összegzés

Az irodalomkutatás végeztével összefoglalnám a tárgyalt témaköröket. Láthattuk, hogy a kamerák fizikai felépítéséből adódóan milyen problémák keletkezhetnek a képek készítésénél. Megismertük azokat a fogalmakat, amik a kamerák belső és külső paramétereinek leírására szolgálnak. Tárgyaltunk egy lehetséges megoldást ezen paramétereknek a meghatározására, továbbá láthattuk, hogy ezek hogyan használhatók fel arra, hogy végeredményként olyan képeket kaphassunk, amik már nem tartalmazzák a fizikai felépítésből adódó torzulásokat. A kutatásban kitértünk az epipoláris kényszer fogalmára, ami a rektifikációval karöltve lehetőséget adott arra, hogy a sztereó pontok egyezőségének keresését egyetlen függőleges koordinátán végigmenve tudjunk kivitelezni. Kitekintettünk arra, hogy milyen tárgyfelismerési, illetve pózbecslési algoritmusok léteznek, továbbá, hogy ezek a modellek, hogyan működnek.
A kutatás segítségével a megvalósítani kívánt rendszert el tudom kezdeni megtervezni, valamint később a fejlesztést is el tudom kezdeni oly módon, hogy már rendelkezésemre állnak a szükséges ismeretek.
Ami a technológia szempontokat illeti, a TRACAB és Hawk-Eye rendszereknél láthattuk, hogy a lehető legpontosabb számítások érdekében kettőnél jóval több

kamerára lenne szükségünk, valamint a kamerák teljesítménye sem egy elhanyagolható tényező.
A fejlesztés során szükségem lesz az OpenCV könyvtárra, valamint pózbecslésnél a Mask R-CNN-re. Azért erre a modellre esett a választásom, mert a többi pózbecslési technológiával szemben példányszintű szegmentálást is végez. Később ezeket a maszkokat fel tudom használni az esetleges továbbfejlesztési lehetőségeknél.
Programozási nyelveket tekintve a számításokért felelős komponenseket Pythonban fogom megírni, mivel az OpenCV itt nyújtja a legszélesebb támogatást, valamint a konvolúciós neurális hálózatok túlnyomó része is ezt a nyelvet támogatja.
A felhasználói felületet pedig C#-ban fogom lefejleszteni egy WPF applikáció formájában.

3. KÖVETELMÉNY SPECIFIKÁCIÓ

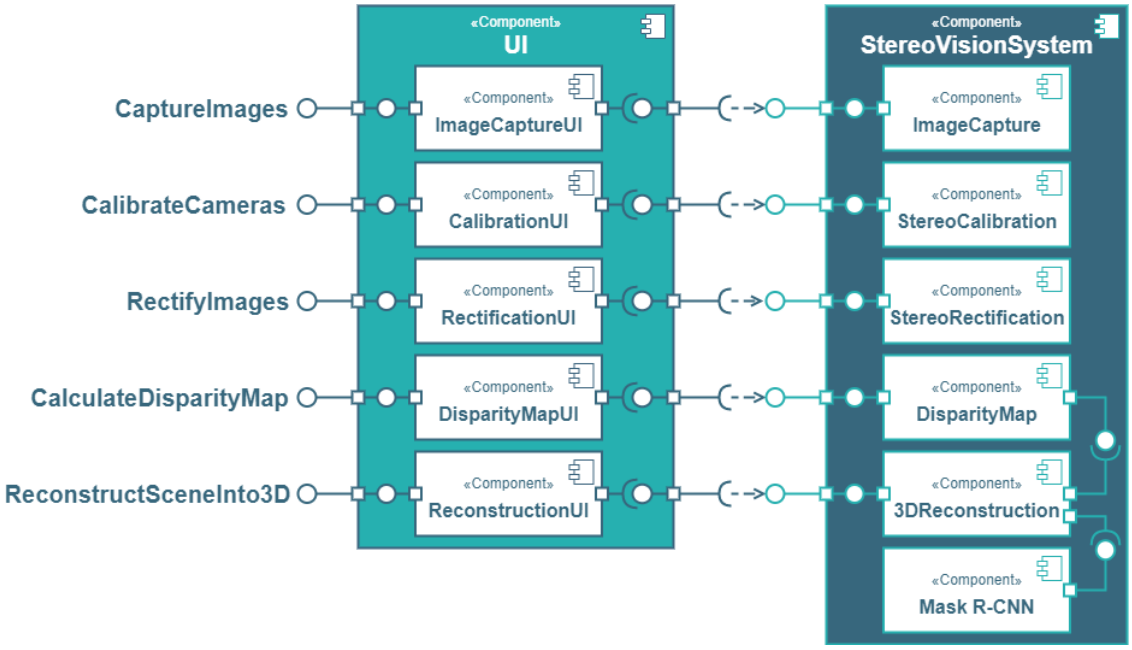
Ebben a fejezeteben a lefejlesztendő rendszerrel szemben állított funkcionális, illetve nem funkcionális követelményekre szeretnék kitérni.
Funkcionális követelményeket tekintve - mivel a rendszer fő célja, hogy a háromdimenziós jelenet újraépítéséhez vezető minden fontosabb lépést nyomon tudjon a felhasználó követni, esetlegesen az egyes lépéseket külön-külön szabályozni - a következő funkcionalitásokat kell tartalmaznia az elkészítendő alkalmazásnak. Rendelkeznie kell egy felhasználói felülettel, amiben a felhasználó képes navigálni az egyes főbb lépéseknek szánt menüpontok között. Az első ilyen menüpontban legyen lehetőség a gépre csatlakoztatott kamerák segítségével képeket, illetve videót készíteni. Ezzel a funkcióval szemben támasztott további követelmény, hogy legyen lehetőség a gépre csatlakoztatott eszközök közül válogatni, illetve az egyes eszközökhöz hozzá lehessen rendelni azt, hogy a sztereó kamerás rendszerben, a jobb, vagy pedig a bal oldalon helyezkednek el. Egy menüpontban legyen lehetőség ezeknek a kalibrációs képeknek a megadásával kalibrálni a sztereó rendszert. Kimenetként meg kell kapjuk a bal, jobb és a sztereó kamerákkal kapcsolatos mátrixokat, illetve együtthatókat. Az egyes kamerák kalibrációjának sikerességét számszerűsíteni szükséges. A kalibrációt végző komponensnek elsősorban olyan képekre kell működnie, amiken a standard sakktábla kalibrációs mintája található. Az alkalmazás rendelkezzen egy olyan komponenssel, ami a kalibráció során kiszámolt mátrixok és együtthatók segítségével egy, a sztereó kamerás rendszer által készített tetszőleges képet, torzítatlanná és rektifikálttá képes tenni. Továbbá legyen az alkalmazás része egy olyan funkcionalitás is, ami egy tetszőleges, rektifikált és torzítatlan sztereó képre képes legyen kiszámítani a hozzá tartozó különbségtérképet. Erre legyen kiépítve egy felület, ahol a felhasználó képes szabályozni a számítást befolyásoló paramétereket. Végezetül a lefejlesztett rendszernek tartalmaznia kell egy olyan komponenst, ami képes a jeleneten szereplő személyek testtartásának detektálására és a különbségtérkép segítségével egy három dimenziós térben el tudja helyezni őket.
Nem funkcionális követelmények közül, az alkalmazásnak Windows 10 operációs rendszerrel kompatibilisnek kell lennie, valamint lehetőség szerint minél kevesebb külső függőséggel kell rendelkeznie a könnyebb kitelepítés érdekében. Hogy minél több rendszeren futtatható legyen az alkalmazás, a pózbecslést végző komponens számításait videókártya helyett, a processzoron kell végezni. További köve-

telmény, hogy az alkalmazás gerincét adó komponenseit dokumentálni szükséges, így más fejlesztők is könnyebben megérthetik a működését, illetve a bemeneti argumentumokról is egy rövid leírást kell szolgáltatni.

4. TERVEZÉS

Az irodalomkutatás alapján ebben a fejezetben fogom megtervezni az alkalmazást. Ki fogok térni arra, hogy a rendszert távolabbról nézve, milyen fő komponensek rajzolódnak ki, valamint ezek hogyan kapcsolódnak össze. Be fogom mutatni a rendszer működését a felhasználó szemszögéből, illetve néhány funkció működésére is kitérek a benne használt objektumok közti interakciók segítségével. Összegyűjtöm a rendszerben található funkciókat, illetve meghatározom, hogy a fejlesztést milyen ütemterv mentén fogom végrehajtani. A mai alkalmazások túlnyomó része interaktív működésű, tehát a felhasználói felület kinézetéről és elrendezéséről is vázlatos tervet fogok készíteni.

4.1. Komponens diagram



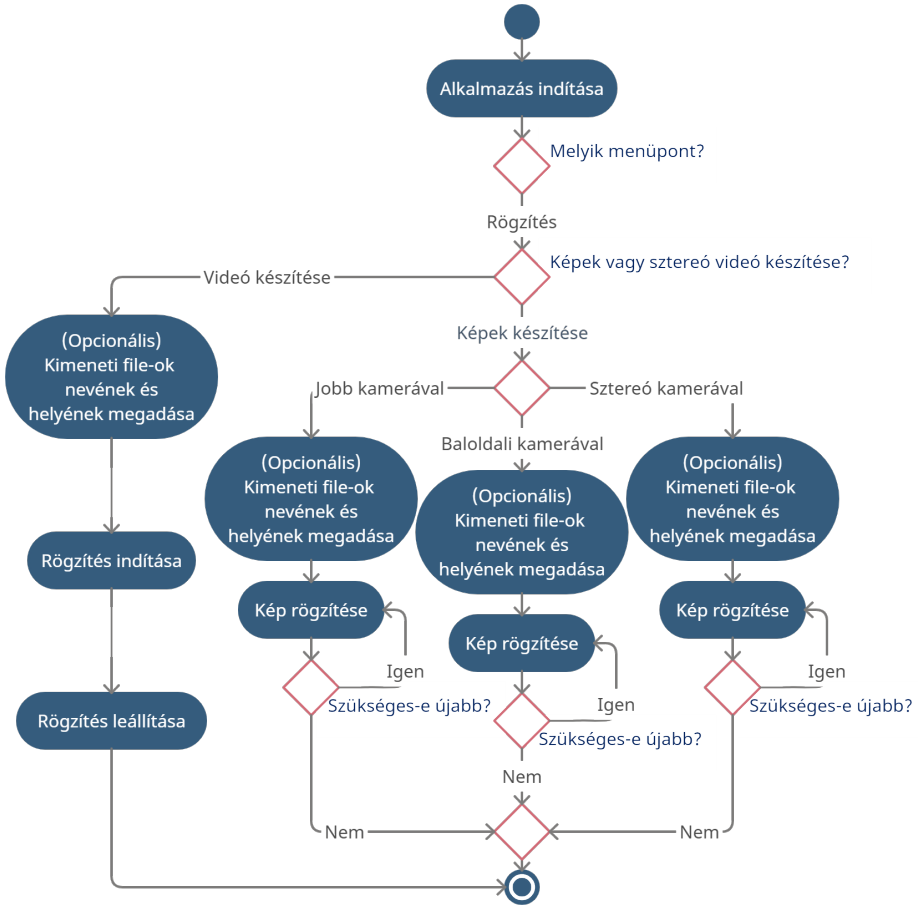
4.1. ábra.
Komponens diagram

A 4.1 ábrán a rendszer komponens diagramja látható. Két fő komponensből épül fel. Az első a számításokért felelős **StereoVisionSystem**, a második pedig a felhasználói felületet nyújtó **UI** komponens.
A *StereoVisionSystem* komponens több al-komponensből áll, amelyeknek jól elhatárolható feladatokat látnak el, mint például a rögzítést vagy kalibrációt.

Ezeket a funkciókat a megjelenítésért, valamint a felhasználói interakciók kezeléséért felelős *UI* komponens fogja használni. Az egyes komponensek jól meghatározott interfészeket nyújtanak az őket használóknak, ezzel kikényszerítve a helyes, egyszerű és egyértelmű használatot.
A *UI*-t is több al-komponens alkotja, amik valójában az alkalmazás egyes menüpontjait fogják alkotni. A komponens fő feladata, hogy egy felületet adjon a felhasználóknak arra, hogy az egyes al-komponenseket fel tudja paraméterezni, el tudja indítani, illetve a kimenetüket megjelenítve prezentálja.

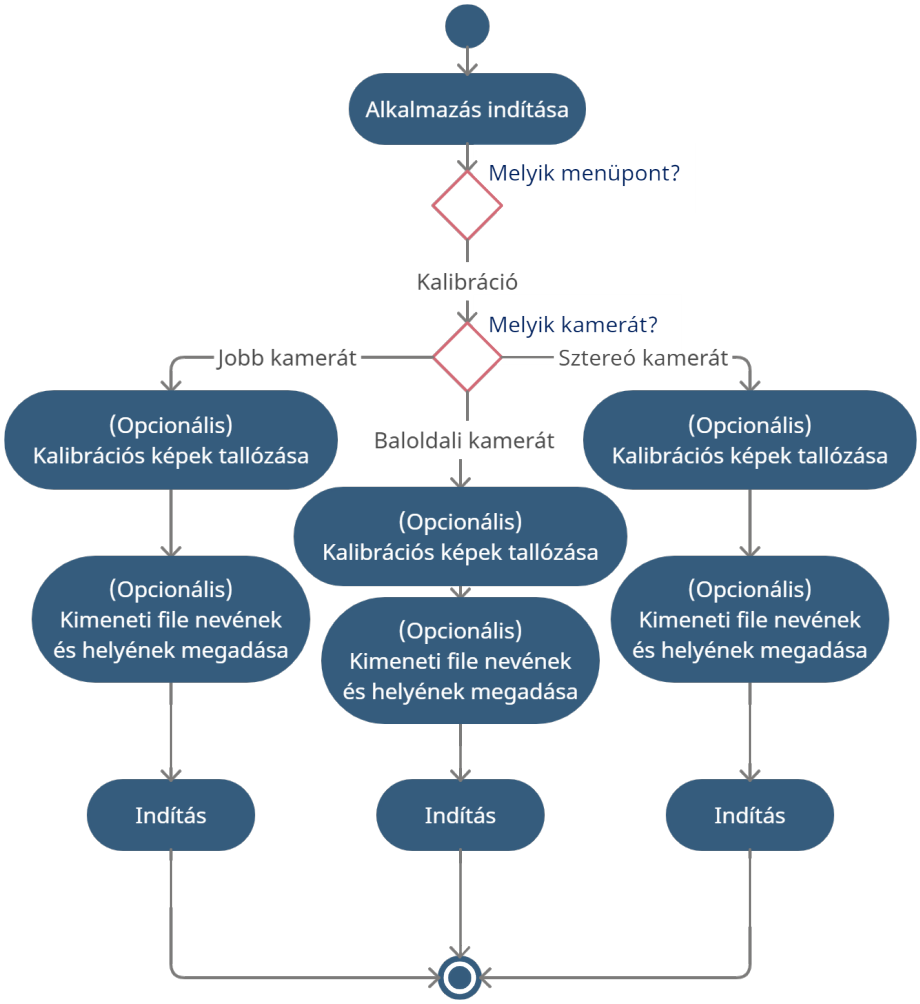
4.2. Aktivitás diagram

Ebben a fejezetben a rendszer működését szeretném bemutatni a felhasználó szemszögéből.



4.2. ábra.
Aktivitás diagram - Rögzítés

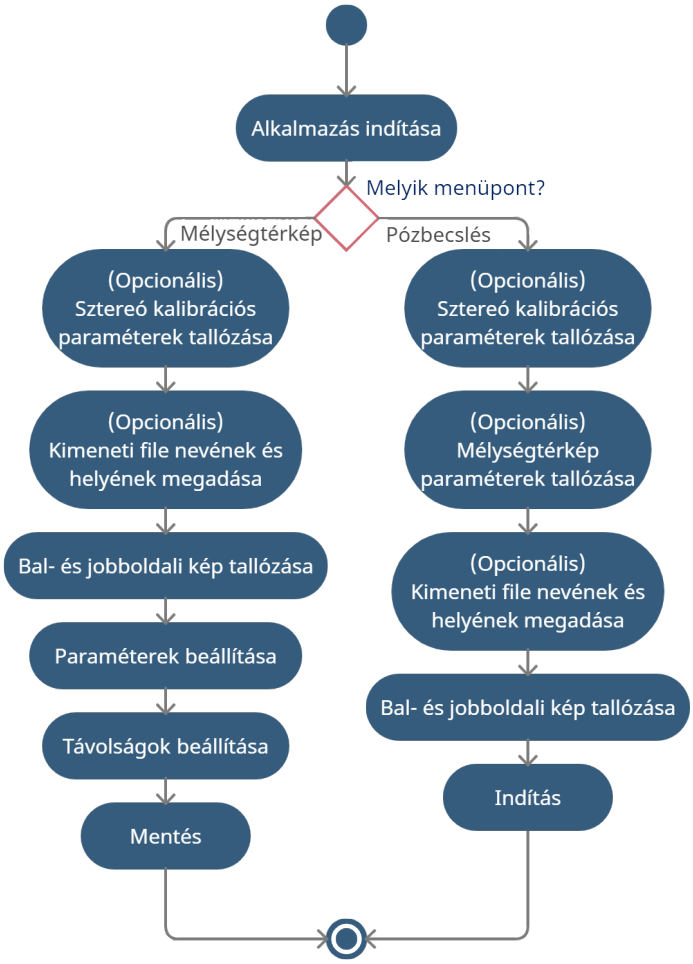
Kép/videó rögzítésének folyamatát a 4.2 diagram mutatja be. A menüponton belül ki lehet választani, hogy képet vagy videót szeretnénk rögzíteni. Ha a felhasználó képet szeretne készíteni, akkor felkínáljuk neki a lehetőséget, hogy bal-, jobb- vagy sztereó kamerával akarja e. Miután megnyílt a kamera/kamerák képe egy gomb lenyomásával menti az adott jelenetet. Természetesen annyi képet tud készíteni, amennyit szeretne. Videó rögzítése esetén a felhasználó egy indítás-, illetve leállítás gombbal tud interakcióba lépni. Függetlenül attól, hogy képet/videót rögzítünk, az alkalmazás felkínálja a lehetőséget a kimeneti állomány nevének és helyének meghatározására, ha ezt a lépést kihagyjuk, akkor egy alapértelmezett helyre és néven mentődik el.



4.3. ábra.
Aktivitás diagram - Kalibráció

A 4.3 diagram a **Kalibráció** menüpont felhasználását mutatja be. A felhasználó

ki tudja választani, hogy bal-, jobb- vagy sztereó kamerára szeretné e elvégezni a műveletet. Ezt követően egy gomb lenyomása elindítja az adott kamera kalibrálását. A menüponton belül két opcionális paramétert is megadhatunk. Az egyik, hogy honnan tallózza a kalibrációs képeket, a másik, hogy a kimeneti állományt hova és milyen néven mentse.



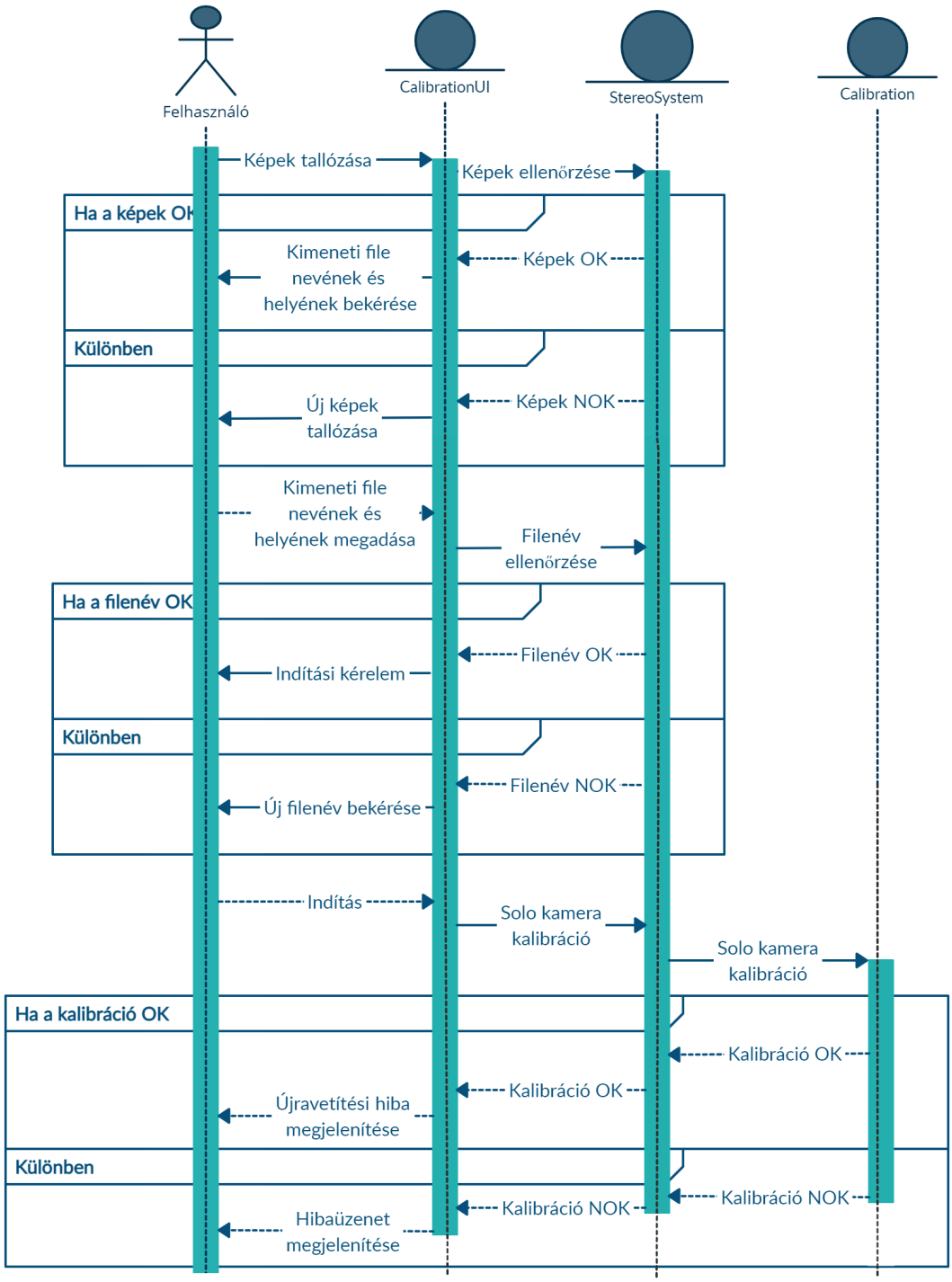
4.4. ábra.
Aktivitás diagramok

A 4.4 diagram két menüpont felhasználását ábrázolja. Az első menüpontban a felhasználó, a diszparitástérkép számolásához szükséges adatokat tudja megadni. Az egyik ilyen paraméter, hogy honnan tallózza a sztereó kalibrációs paramétereket. A másik, hogy a kimeneti állományt hova és milyen néven mentse. Továbbá minden 2.8 alfejezetben tárgyalt paramétert képes szabályozni. Miután megad a felhasználó egy sztereó képet, megjelenik a hozzá tartozó különbségtérkép. A második menüpont

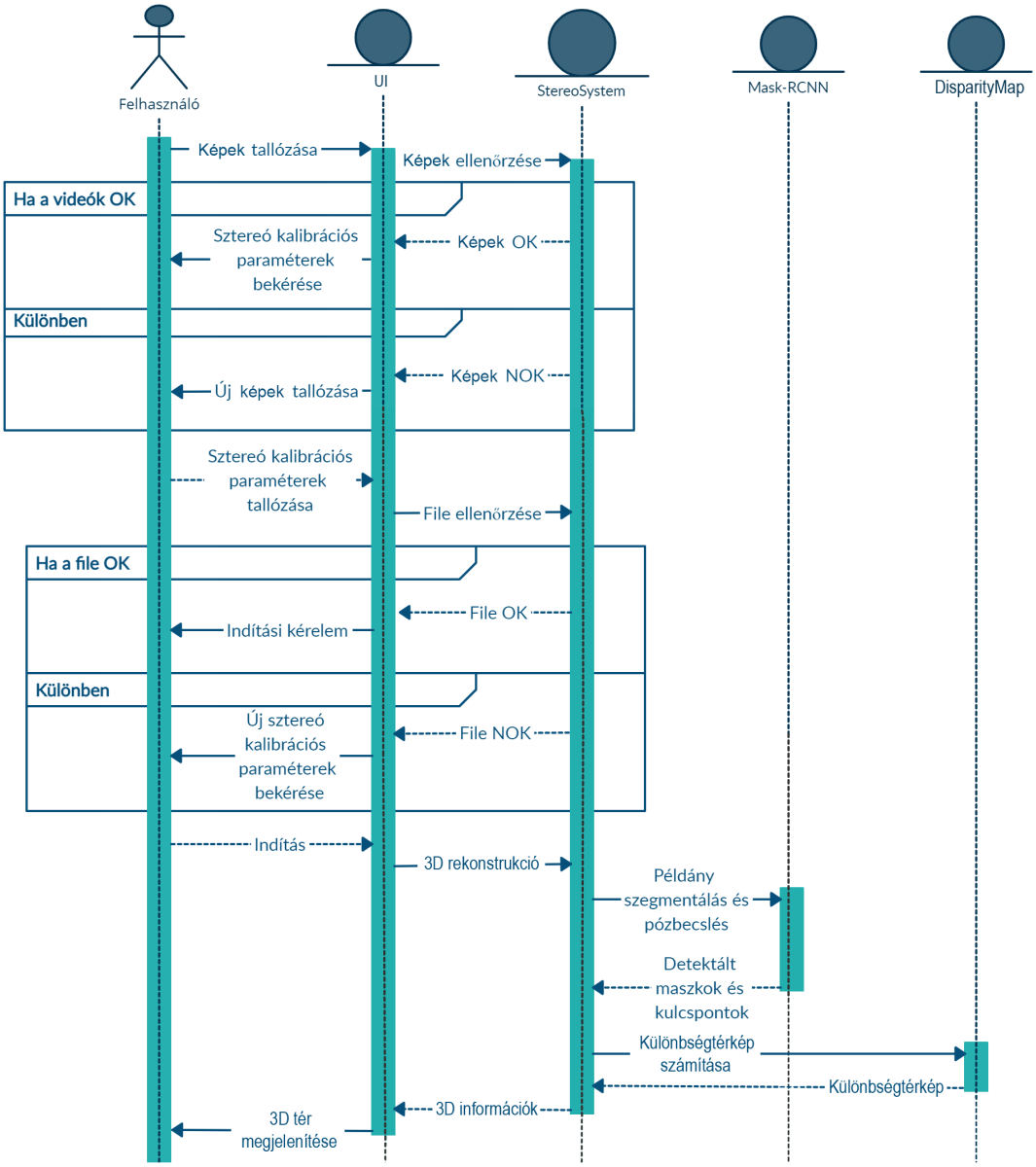
a háromdimenziós újraépítéssel kapcsolatos. Itt a *Diszparitástérkép* menüpontban definiált két opcionális paraméter kiegészül egy harmadikkal, vagyis, hogy honnan tallózza be a diszparitástérkép paramétereit. Miután a felhasználó megadott egy sztereó képet, egy gomb lenyomásával el tudja indítani az algoritmust.

4.3. Szekvencia diagram

Ebben a fejezetben a rendszer két fontosabb funkciójának belső működését szeretném bemutatni szekvencia diagram segítségével.
A 4.5 diagramon a kamera kalibrációjának művelete látható. Nulladik lépés a kamera kiválasztása lenne, de az egyszerűség érdekében, most feltételezem, törekedve, hogy bal- vagy jobboldali kamerát választott ki. Az opcionális paraméterek miatt többféle úton érhetünk el a kalibráció végéhez, ezek közül egy szituációt fogok végigvezetni. Megadjuk a kalibrációhoz szükséges képek elérési útját, ami először a *UI* kalibrációért felelős al-komponenséhez kerül. Ezt követően meghívja a *StereoVisionSystem* képek validációjáért felelős metódusát. Ha a képek megfelelőek, akkor a felhasználó továbbléphet, különben új képeket kell megadnia. Az opcionális paraméterek közül kitöltjük a kimeneti fájl nevét és hogy hova szeretnénk menteni. Ennek a bemenetnek a helyességét a *UI* al-komponens ellenőrzi. Az indítás gombra kattintva, a *UI* al-komponense az előzőleg megadott paramétereket átadva meghívja a *StereoVisionSystem* egyetlen -bal, vagy jobb - kamera kalibrálásáért felelős metódusát. A kérés továbbításra kerül a *Calibration* al-komponense felé. Ha a művelet sikeresen lezajlott, megjelenik a felhasználói felületen az újra vetítési hiba, ellenkezőleg pedig egy hibaüzenet.
A 4.6 diagramon a jelenet háromdimenziós újraépítésének menete látható sztereó képekre. Az opcionális paraméterek miatt, itt is egy lehetséges folyamatot mutatok be. A felhasználó megadja a sztereó kamerás rendszer kalibrációs paraméterek, illetve a sztereó képek elérési útját. Mindkét bemenetet ellenőrizzük, annak érdekében, hogy minél korábban felfedjük az esetleges hibákat. Az indítás gombra rákattintva el elindul a számítást. A *UI* al-komponense meghívja a *StereoVisionSystem* háromdimenziós újjáépítéséért felelős metódusát. Először a *Mask R-CNN* segítségével meghatározza a maszkokat, illetve kulcspontokat. Ezt követően kiszámítja a sztereó kép diszparitástérképét. Miután a különbségtérkép, a maszkok és a kulcspontok rendelkezésre állnak, a *StereoVisionSystem* feldolgozza őket és a kimenetet megjeleníti egy három dimenziós térben.



4.5. ábra.
Szekvencia diagram - Kalibráció



4.6. ábra.
Szekvencia diagram - 3D újraépítés

4.4. A rendszer funkciólistája

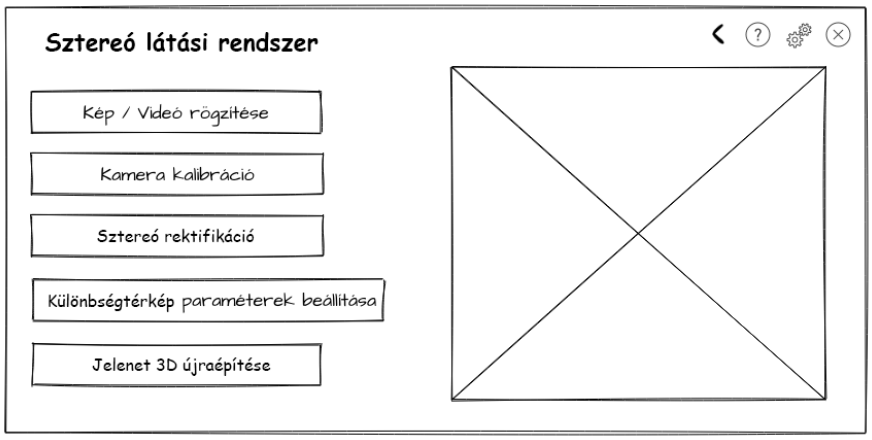
Ebben a fejezetben az alkalmazásba tervezett menüpontokat és a bennük lévő funkciókat szeretném felsorolni.

- **Rögzítés**:
  - Bal-, jobb-, sztereó kamera által képek, illetve videók rögzítése.

- **Kalibráció**:
  - Bal-, jobb-, sztereó kamera belső- és külső paramétereinek kiszámítása.
  - A kiszámított paraméterek fájlba mentése.
- **Sztereó rektifikáció**:
  - Egy, a rendszerrel készített sztereó kép korrigálása oly módon, hogy torzítatlanná és rektifikálttá váljon.
- **Diszparitástérkép**:
  - A különbségtérkép kiszámításához szükséges paraméterek beállítása.
  - A paraméterek alapján a diszparitástérkép kiszámítása.
  - A paraméterek fájlba mentése.
- **3D újraépítés**:
  - Jeleneten szereplő emberek kulcspontjainak lokalizálása.
  - A lokalizált kulcspontok koordinátáinak kiegészítése a különbségtérkép értékeivel.
  - A kulcspontok összekötése, hogy emberi vázakat alkossanak.
  - A jelenet megjelenítése egy háromdimenziós térben.

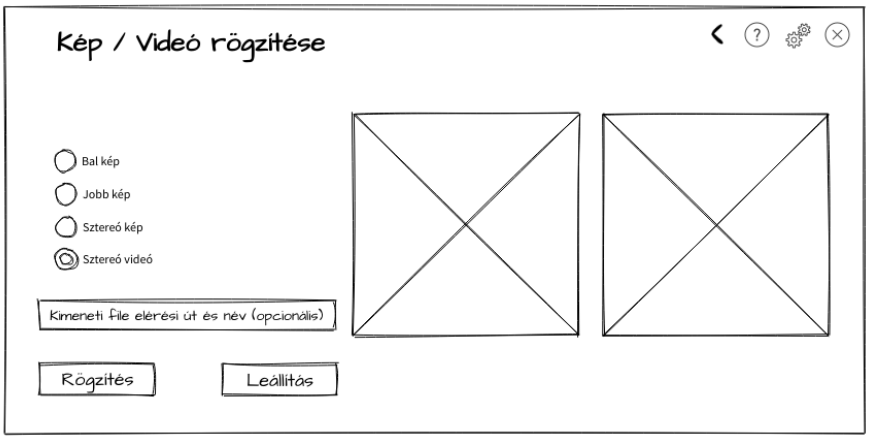
4.5. Drótváz

Ebben az alfejezetben az egyes menüpontok drótvázát terveztem meg. Ez lehetőséget ad arra, hogy a felhasználói felület fejlesztésénél a végleges kialakítást, egy, már előre megtervezett váz alapján tudjam megvalósítani.



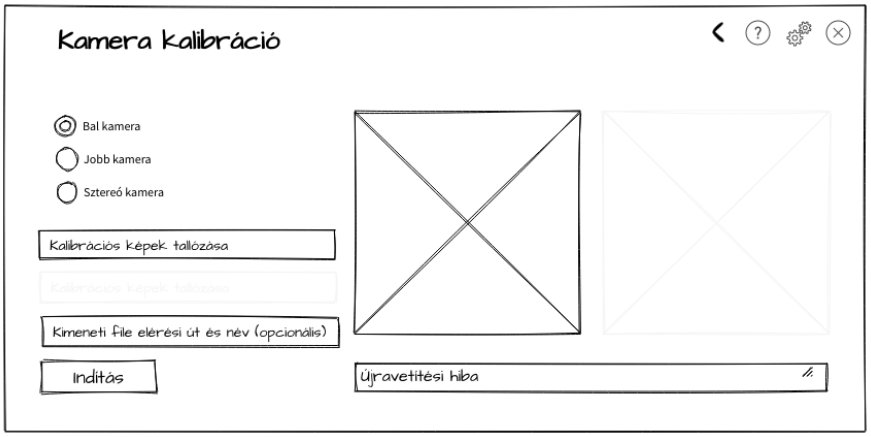
4.7. ábra.
Drótváz - Főmenü

A 4.7 ábrán látható főmenü, az egyes menüpontok közti navigációt teszi lehetővé. Ez az ablak fogadja felhasználót az alkalmazás megnyitásánál.



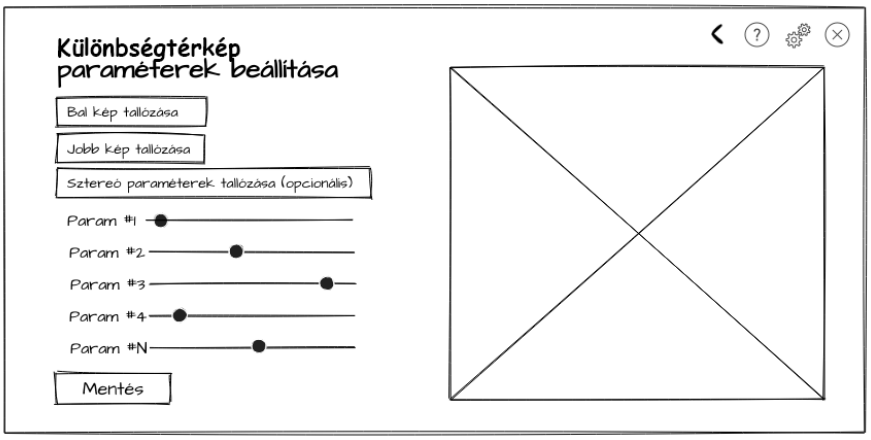
4.8. ábra.
Drótváz - Rögzítés

A 4.8 ábrán látható menüpontban képeket vagy videókat tud a felhasználó rögzíteni. Itt rádió gombok segítségével tudja azt kiválasztani, hogy mit szeretne, illetve különféle gombokkal tudja a rögzítést megtenni (például képek rögzítésénél Rögzít/Leállít). Az ablak jobb oldalán, pedig a kamera képei jelennek meg.



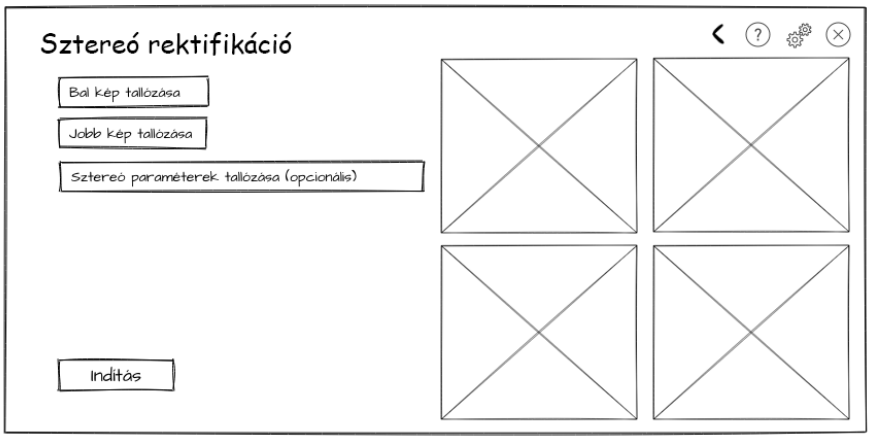
4.9. ábra.
Drótváz - Kalibráció

A 4.9 ábrán látható menüpontban a felhasználó kalibrálni tudja a sztereó rendszert. Itt szintén rádió gombok segítségével tudja kiválasztani, hogy melyik oldalt szeretné kalibrálni. A jobb oldalon a kalibrációs képek, illetve az újravetítési hiba fog megjelenni.



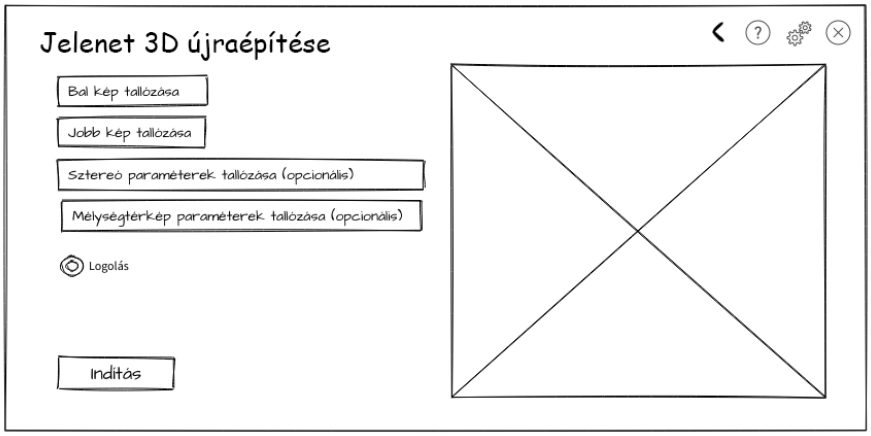
4.10. ábra.
Drótváz - Diszparitástérkép

A 4.10 ábrán látható menüpontban a diszparitás térkép számításához szükséges paramétereket lehet beállítani. Ezeket a paramétereket főként csúszkák segítségével lehet szabályozni. Jobb oldalon pedig a különbségtérkép fog megjelenni.



4.11. ábra.
Drótváz - Sztereó rektifikáció

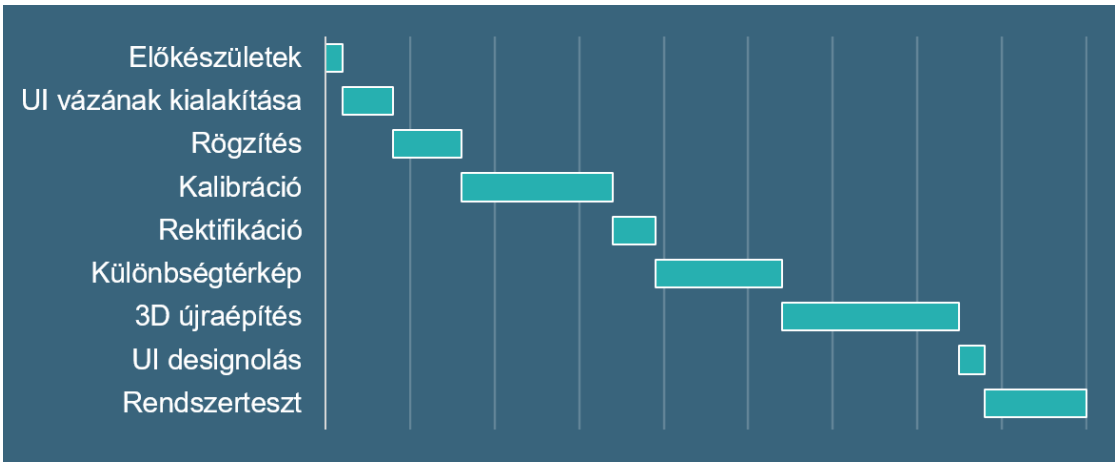
A 4.11 ábrán látható menüpontban a sztereó képek korrigálása történik meg. A felhasználó, dialógus ablakok segítségével tudja kiválasztani a sztereó képpárt. Jobb oldalon megjelennek a bemeneti-, illetve a kimeneti torzítatlan és rektifikált képek.



4.12. ábra.
Drótváz - 3D újraépítés

A 4.12 ábrán a háromdimenziós újraépítés menüpontja látható. Itt a szükséges adatok megadása után a jobb oldalon jelenik meg a háromdimenziós tér, a jelenetben szereplő emberek csontvázaival.

4.6. Fejlesztés tervezett menete



4.13. ábra.
Gantt diagram

Ebben a fejezetben a 4.13 diagram segítségével fogom bemutatni, hogy milyen fejlesztési feladatokat definiáltam, milyen hosszokkal, illetve azt, hogy ezek között milyen függőségek figyelhetőek meg. Fontos kiemelni, hogy az olvashatóság érdekében a diagramon belül nem szerepelnek az egyes komponensek fejlesztését

követő egységtesztelések, integrációk, valamint integrációs tesztelési lépések, de ezek a fejlesztés során jelen lesznek.
Előkészületek alatt értek minden olyan tevékenységet, ami szükséges a fejlesztés elkezdéséhez. Ez a lépés elsősorban a kamerák beszerzését, illetve a fejlesztési környezetek beüzemelését fogja jelenteni.
A felhasználói felület felépítésének kialakítása azért a fejlesztés első szakasza, mivel így minden egyes funkciót, egy-egy új menüpontként tudok kezelni. Bővebben, ha már létezik egy keret, akkor abba az egyes funkcionalitások tudnak majd építkezni.
A felhasználó felület felépítésénél a legfontosabb dolog az infrastruktúra kiépítése. Ez túlnyomórészt attól függ, hogy milyen architekturális tervezési minta mellett döntünk. Ebben a lépésben szükséges megvalósítsuk a főmenü-t, ami lehetőséget ad a navigációra az egyes menüponttok között.
Minden további fejlesztési lépés azon felül, hogy magába foglalja a funkciónak az implementálását, együtt jár egy:

- egységteszteléssel, ami a funkcionalitás helyességét fogja ellenőrizni,
- integrációval, ami az adott menüpont kialakítását jelenti, oly módon, hogy a felhasználó már tudja használni a funkciót,
- integrációs teszteléssel, ami azt ellenőrzi, hogy integráció után is helyesen működik együtt a felhasználói felület, illetve az alatta lévő üzleti logika.

A fejlesztés ütemezése innentől természetes módon adja magát, mivel minden egyes lépés függ egy korábbi lépéstől.
Rögzítés alatt értem a bal-, jobb-, sztereó kamerák általi kalibrációs képek, illetve videók készítését.
Az imént említett függőségek itt mutatkoznak meg először, mivel kalibrációs képek nélkül, magát a kalibrációt sem tudjuk elkezdeni. A kalibráció lépés alatt értem a bal-, jobb-, sztereó kamerák kalibrálását, valamint a kiszámított paraméterek fájlba mentését, illetve olvasását. Ez az első fontosabb al-komponens, így ennek a fejlesztési időtartamán ennek megfelelően határoztam meg.
A következő lépés szintén adja magát, mivel torzított, vagy nem rektifikált képekre nem tudnánk különbségtérképet számolni, ezért a rektifikációért felelős al-komponens következik. Mivel az irodalomkutatás során kiderült, hogy erre létezik az OpenCV-nek egy beépített függvénye, ezért ennek a fejlesztési lépésnek a tervezett időtartamát rövidebbre vettem.
Az imént említett különbségtérkép számításáért felelős al-komponens fejlesztése a következő. Ez egy kulcsfontosságú fejlesztési lépés, mivel, ha ez a komponens

nem működik megfelelően, akkor a háromdimenziós újraépítés során sem fogunk jó eredményeket kapni. Itt a rengeteg szabályozható paraméter miatt a felhasználói felület kiépítése is több időt vehet igénybe.
Az utolsó lefejlesztendő al-komponens a jelenet háromdimenziós újraépítéséért felel. Ezen belül több al lépést tudunk definiálni. Először is, képesnek kell legyünk a kulcspontok lokalizálására, majd összekötésére. Az összekötött kulcspontok, végtagokat alkotnak, amik együttesen egy emberi csontvázat fognak reprezentálni. Másodszor, ezeket a koordinátákat kombinálnunk kell tudni a különbségtérkép értékeivel. Harmadszor, ezek az adatokat képesnek kell legyünk megjeleníteni egy három dimenziós térben. A tervezett menetrendben eszerint választottam egy hosszabb időt ennek a lépésnek.
Mivel az alkalmazást egy kellemes-, és egységes kinézettel együtt szeretném leszállítani, ezért ennek a megvalósítására is tervezek időt.
A fejlesztés során minden komponens tesztelésre kerül, de egy rendszertesztnek is szánok fejlesztési időt, az esetlegesen előfordulható hibák feltárásához.

5. FEJLESZTÉS

5.1. Előkészületek

A fejlesztés megkezdésekor egy nagyon alapvető problémába ütköztem, mégpedig, hogy nem volt egy elérhető kamera a közelben, nem hogy egyszerre kettő. Így első feladatom volt, hogy beszerezzek két azonos típusú, Full-HD-s webkamerát. A választásom a Xiaomi IMILAB W88S-re esett, mivel ez volt ár/érték arányban a legkedvezőbb. Természetesen ezek már eleve egy szűk keresztmetszetet jelentenek a 2.1 alfejezetben tárgyalt technológiákhoz képest. Ezek után már nekiláthattam a fejlesztési környezet beüzemelésének. A Visual Studio [27] már telepítve volt a gépemen, viszont a Visual Studio Code [28] még nem, így ezt letöltöttem és beüzemeltem a Python-ban való fejlesztésre. Ahhoz, hogy menet közben többféle megoldást ki tudjak próbálni - amik esetlegesen más függőségeket tartalmaznak vagy éppen ugyan azokat, de más verzióval -, szükségem volt virtuális környezetekre, erre pedig az Anaconda [26] nyújtotta a számomra legegyszerűbb megoldást. Továbbá a fejlesztés során verzió követni is fogom az alkalmazást, ennek érdekében létrehoztam egy Gitlab-os távoli repository-t. Mivel a fejlesztést csak én fogom végezni, ezért egy egyszerű GitHub Flow branching stratégiát választottam, ahol van egy stabil főágam, amiről az egyes funkciók fejlesztése esetén leágazok.

5.2. Felhasználói felület felépítésének kialakítása

Mivel a felhasználói felület tervezés és kivitelezés közel áll a szívemhez, ezért a WPF alkalmazás kiépítésével kezdtem. Mivel egy teljesen új és friss rendszerről van szó, amihez nincsenek külső függőségeim - a UI réteget tekintve -, ezért a legfrissebb .NET 6.0 keretrendszerben hoztam létre a projektet. C# oldalról a fő célom az volt, hogy egy jól felépített rendszert hozzak létre. Szerencsére a WPF alkalmazások esetében ez már egy szinte előre eldöntött architekturális mintát jelent, ez pedig nem más, mint a Modell-Nézet-Nézetmodell (MVVM) tervezési minta, aminek talán a legfőbb aspektusa, hogy adatkötéseken alapul. Az adatkötések gyakorlatban történő felhasználására különféle implementációk nyújthattak volna segítséget, mint például a Prism [29], vagy az MvvmLightLibs [30], de úgy döntöttem, hogy mivel egyenlőre nincs szükségem a teljes funkcionalitásaikra, ezért saját magam készítem el a szükséges osztályokat.
Az első ilyen osztály a **ViewModelBase**, ami megvalósítja az *INotifyPropertyC-*

hanged interfészt. Erre azért van szükség, hogy az egy- vagy kétirányú adatkötések megfelelően működhessenek. Enélkül, ha kétirányú adatkötésről beszélünk, a nézet nem értesülne a nézetmodell változásairól és ez fordítva is igaz lenne. A *ViewModelBase* legfontosabb metódusa a *Set*, ami bemenetként egy generikus mezőt vár referenciaként és egy, a mező típusának megfelelő értéket. Egy paraméterre van még szükségem, mégpedig a hívó tulajdonság nevére, de ezt egy attribútum segítségével automatikusan le tudom kérdezni. A metódus egy dolgot ellenőriz, mégpedig, hogy a bemenetként kapott mező értéke megegyezik-e az új értékkel, ha nem, akkor elsüti a *PropertyChanged* eseményt. Erre az ellenőrzésre azért van szükség, hogy elkerüljük az úgynevezett eseménykezelő mérgezést vagyis, ha a mező értéke nem változott, akkor ne hívódjon meg a *PropertyChanged* esemény újra és újra feleslegesen. Miután az adott nézet konstruktorában a *DataContext*-et beállítjuk az adott nézetmodell példányra, akkor már működőképesek lesznek az adatkötések, ha a nézeten lévő UI vezérlők tulajdonságait hozzákötjük a nézetmodellben szereplő megfelelő tulajdonsággal, valamint a nézetmodell tulajdonságán belül a *set* kódblokkban az egyszerű értékadás helyett az ős, vagyis a *ViewModelBase Set* metódusát hívjuk meg az új értékkel.

A második osztály, amit az MVVM minta megvalósítása érdekében implementálnom kellett az a **DelegateCommand** volt. Erre azért van szükség, mert eddig csak a tulajdonságokat tudtuk adat kötni, de mi van akkor, ha a felhasználó rákattint egy gombra? Alap esetben a nézet kód hátterében lenne megadva egy kattintás metódus, ami meghívódna, ha a gomb kattintás eseménye elsülne. Hogy adat kötni tudjuk a nézetmodell metódusát egy nézet vezérlő eseményére, ahhoz implementálnunk kell az *ICommand* interfészt, aminek van egy *Execute* metódusa. Ez akkor hívódik meg, ha a gombra rákattintunk. Ezt az osztályt úgy tudjuk majd felhasználni, hogy tulajdonságként megadjuk a *ViewModelBase*-ből leszármazó osztályban és a konstruktorban pedig létrehozunk egy új *DelegateCommand* példányt, paraméterként pedig egy delegáltat adunk meg, ami majd az egyedi metódusunk lesz.

Az alkalmazáson felül magát a projektet is próbáltam minél strukturáltabban kialakítani, így az utóbb említett osztályok az *Infrastructure* névtérbe kerültek, mivel a UI réteg alapvető működéséhez szükségesek.

Most, hogy már minden szükséges osztály rendelkezésemre állt, így már az automatikusan legenerált *MainWindow* nevű nézethez létre tudtam hozni egy **MainViewModel** nézetmodellt, ami természetesen az előbb tárgyalt *ViewMo-*

delBase-ből származik le. Ez a nézetmodell tartalmazni fog minden navigációval kapcsolatos funkcionalitást. Ahhoz, hogy a *MainWindow*-ban az éppen kiválasztott menüpont nézetét dinamikusan cserélgetni tudjuk, a nézetmodellbe szükségünk van egy *ViewModelBase* típusú tulajdonságra, amit adat kötünk a *MainWindow*-on lévő *ContentPresenter* vezérlő [Content] tulajdonságával. Amikor a felhasználó rákattint egy menügombra, akkor meghívódik a nézetmodell adott metódusa, ami kicseréli ezt a **ViewModelBase** típusú tulajdonságot az újonnan kiválasztott menüpontnak megfelelő példányra. Felmerülhet a kérdés, hogy egy nézetmodell típus alapján, honnan tudja a keretrendszer, hogy milyen nézetet jelenítsen meg. A válasz, hogy sehogy, ezt nekünk kell megadni a *App.xaml*-ben, mint *DataTemplate*, ahol megadjuk, hogy adott nézetmodellhez, melyik nézet tartozik.
A következő lépés magának a nézet elrendezésének a kialakítása volt. Mivel az alkalmazás összesen hat menüpontból fog állni almenük nélkül, ezért egy egyszerű függőleges menü elrendezést terveztem meg egy extra funkcionalitással, mégpedig, hogy összecsukható legyen. Ezt a funkciót azért tartottam szükségesnek, mivel sztereó képeket fogunk megjeleníteni egymás mellett, ezért minél nagyobb helyet akartam hagyni az egyes menük tartalmának. Ezt a funkcionalitást egy *ToggleButton* vezérlő segítésével értem el, amire, ha rákattint a felhasználó, akkor a külső *Grid* - ami a vertikális menü vázát adja - szélessége összeszűkül vagy kiszélesedik.
Egy mai modern felhasználói felület nem lenne teljes egy téma nélkül, így minden felhasznált UI vezérlő kinézetét személyre szabtam, valamint minden felhasznált szín-t egy *ResourceDictionary*-ben tároltam el, így akár a későbbiekben egy Sötét/Világos téma váltót is könnyen lehetne implementálni, mivel a fő színek nincsenek a nézetekre égetve. Valamint, hogy a projekt struktúrája ne váljon hamar átláthatatlanná, a stílusok, ikonok és erőforrások is külön mappákba kerültek. Természetesen, hogy az imént említett dolgok felhasználhatóak legyenek, ezeket is fel kell venni az alkalmazás erőforrásai közé az *App.xaml*-ben.
Ahhoz, hogy az egyes osztályok függőségeinek feloldását egy helyen tudjam kezelni, hozzáadtam a projekthez az Autofac [31] nevű NuGet csomagot. Ez egy IoC konténer, aminek a segítségével képes vagyok beregisztrálni nem csak komponenseket, hanem teljes modulokat is és automatikusan képes kezelni a konstruktor paraméterek injektálását.

5.3. Python szkriptek futtatása és megjelenítése

A rendszerrel szemben állított követelmények között szerepel, hogy az egész folyamatot végig tudja követni a felhasználó, menüpontról menüpontra haladva. Mivel az egyes főbb funkcionalitások külön-külön felparaméterezhető Python szkriptekbe lesznek szervezve, ezért ez a követelmény könnyen kivitelezhető. Első lépésként lehetőséget kell adni a felhasználónak arra, hogy az egyes szkriptek argumentumait megtudja adni az alkalmazás felületén. Ha minden paraméter rendelkezésre áll, akkor a Python interpreter segítségével külön folyamatként lehet futtatni a felparaméterezett szkriptet. Ezzel csak egy problémám volt, hogy a szkript éppen megjelenő kimenete egy külön ablakban jelenik meg, nem pedig a felhasználói felületen belül. Valószínűleg sokan az előbb említett megoldással is ki tudtak volna békülni, de személy szerint én nem és már csak szakmai kíváncsiságból is ki akartam próbálni, hogy képes vagyok e valahogy az ablakon belülre helyezni a szkriptek kimenetét. Ez nem volt egy egyszerű feladat, valószínűleg több idő ment el rá, mint amennyivel számoltam, de végeredményként előállt a működő megoldás. Létrehoztam egy **PythonProcessWrapper** nevű szerelvényt, ami logikailag a felhasználó felületet nyújtó réteg alatt helyezkedik el.

Fő osztálya a **PythonProcessHost**, ami elsődlegesen azért felelős, hogy elindítsa a Python szkripteket és úgymond önmagába zárja a szkript által megjelenített ablakot. Külön érdekesség ezzel az osztállyal kapcsolatban, hogy a felhasználói felületre egyből felhelyezhető, mivel *UIElement* típusként funkcionál. Ahhoz, hogy ez a megoldás működhessen fel kellett használnom a Win32 API-t, amiből szükségem volt pár natív osztályra és metódusra. Ennél a megoldásnál viszont le kellett mondjak arról a lehetőségről, hogy a szkripteket a Python interpreteren keresztül futtassam, mivel nem várt eredményeket hozott. Ezért a rendszer üzembe helyezése kibővült egy extra feladattal, mégpedig, hogy a szkriptekből a PyInstaller [32] segítségével futtatható állományt kell készíteni. Ez egyrészt előnyt is jelent, mivel a Python szkriptet és annak összes függőségét egyetlen futtatható állományba köti össze, így a felhasználónak nincs szüksége Python interpreterre vagy a szkript függőségeire, ahhoz, hogy futtatni tudja az alkalmazást. Másrészt ez egy hátrányt jelent abból a szempontból, hogy ha a szkript változik, akkor minden egyes alkalommal újra le kell futtatni a telepítőt, valamint nagyobb méretet jelent, mivel minden futtatható Python alkalmazás külön-külön tárolja a függőségeit. A mai tárhely kapacitásokat figyelembe véve, úgy gondolom, hogy az utóbbi egy

elhanyagolható tényező.
Korábban már említettem, hogy minden funkcionalitás külön szkriptbe lesz szervezve, így egy megoldást kellett találjak arra, hogy hogyan tudom egységes módon elindítani és megjeleníteni a Python alkalmazásokat. Ennek megoldására létrehoztam egy **PythonProcessInfo** nevű adattároló osztályt, amit a UI rétegen kötelező teljesen feltölteni ahhoz, hogy eltudjuk indítani a Python alkalmazást. Ez az osztály tartalmazza a futtatható állomány nevét és elérési útvonalát, argumentum listáját és, ha az adott alkalmazás megjelenít egy ablakot, akkor annak a nevét. Az utóbbi azért fontos, mert próbálkoztam többféle megoldással is, mint például, hogy a futó folyamatok közül adott ablak osztályra szűrök rá, de végül az ablak nevére való szűrés nyújtotta a legkonzisztensebb végeredmény. Ez az ablak név szintén argumentumként adódik át a Python alkalmazás felé és, ha alkalmazáson belül megakarunk jeleníteni egy ablakot, például rögzítés során a webkamera képének, akkor ezen a néven keresztül tudjuk megtenni és így az ablak megjeleníthető lesz a felhasználói felületen belül.
A **PythonProcessHost** ezeken a funkciókon kívül még képes a folyamatot megállítani, képes visszajelzést adni arról, hogy a folyamat fut e még és minimális interakcióra is képes, mint például gombleütést továbbítani.
A szerelvény lefejlesztésének végén szembesültem azzal, hogy mennyi rejtett függőség került a kódba, mint például a statikus *File* osztály, így elkezdtem kiszervezni őket külön szerviz osztályokba, amik mind egy nekik megfelelő absztrakcióra épülnek, vagyis egy interfészre.

5.4. Kalibrációs képek rögzítése

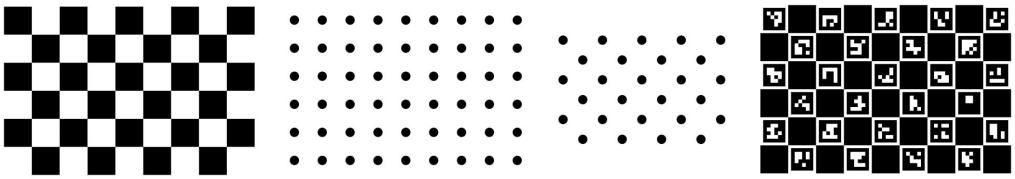
A kalibrációs képek rögzítésére vonatkozó követelmények alapján kezdtem el megtervezni a Python szkriptet. Argumentumként várom azt a paramétert, hogy melyik kamerának a képét szeretném rögzíteni, ez lehet a jobb-, bal vagy a sztereó kamera, valamint a hozzájuk tartozó indexelés, olyan formában, ahogy az operációs rendszeren is megjelennek ezek az eszközök. Az utóbbi esetében nem kell komplex dologra gondolni, ha egy eszköz (pl. webkamera) van a gépre csatlakoztatva, akkor annak az indexe egy nullás értéket vesz fel, a következő csatlakoztatott eszköz pedig az egyest, de ezzel a felhasználónak nem kell foglalkoznia. A 5.3 fejezetben leírtak alapján, mivel esetlegesen egyszerre két kamera képe is megjelenhet, ezért szükségünk van két ablak névre is. Ezek a nevek lehetnek tetszőlegesek vagy véletlenszerűek is akár,

a lényeg, hogy minden rétegben ugyan azt az értéket hordozzák, ezért kell kívülről szolgáltassuk. Továbbá szükségünk van egy kimeneti könyvtár útvonalra, ahova a képek mentése fog megtörténni. Ez alapértelmezetten a *Output* mappa. Valamint opcionális paraméterként meg lehet adni egy felbontást, ami értékre be lesz állítva az OpenCV beépített rögzítőjének felbontása, ha esetlegesen nem felelne meg az alapértelmezett, ami a mi esetünkben a 1920x1080 felbontást jelenti. Működését tekintve először elindítom a kamerákat és ha gombnyomás történik, akkor kiolvasom a videofolyamból az éppen aktuális képkockát és elmentem az argumentumként megadott kimeneti könyvtárba. Az elmentett fájl neve tartalmazza, hogy melyik oldalról készült, valamint egy indexet. Külön érdekesség, hogy sztereó képek rögzítésénél, nem az OpenCV *Read* metódusát használom, hanem először a *Grab* metódust, ami csak egy igaz/hamis értékkel tér vissza, hogy az éppen aktuális képkocka megszerzése sikeres volt e vagy sem és csak ezután hívom a *Read* metódust. Erre azért van szükség, mert sztereó képek rögzítésénél nagyon fontos, hogy a két jelenet szinkronba legyen egymással és mivel a *Grab* metódus kevesebb időt vesz igénybe, mint a *Read*, ezért ebben a formában érdemes használni őket. Kicsit előre gondolkozva, úgy döntöttem, hogy nem csak képek, hanem videók rögzítésére is lehetőséget adok. Egyetlen egy argumentummal egészítettem ki az alkalmazást, ami a rögzítés módjára utal, vagyis, hogy képeket, vagy videókat szeretnénk rögzíteni. Miután megbizonyosodtam, hogy minden funkció, minden argumentum kombinációval működik, a 5.3 fejezetben említett PyInstaller segítségével elkészítettem a futtatható állományt.
A felhasználói felület kialakítását azzal kezdtem, hogy felvettem egy új nézetet a hozzá tartozó nézetmodellel, ami a 5.2 fejezetben tárgyalt *ViewModelBase*-ből származik le. Ezután elkezdtem felvenni a szükséges vezérlőket a nézetre, de hamar rájöttem, hogy a paramétereknek a megadása nagyon sok helyet foglal el a nézeten, így a vertikális menühöz hasonló funkcionalitást vezettem be, vagyis egy *ToggleButton* segítségével változtattam a panel méretét, ezzel elrejtve vagy éppen megjelenítve a beállításokat. Ami megemlítendő, hogy a felhasználói felületet próbáltam minél inkább úgy kialakítani, hogy nehezebben lehessen hibázni. Például, ameddig nincs kiválasztva oldalnak a sztereó lehetőség, addig csak az egyik *ComboBox* látszik, de ahogy kiválasztjuk a sztereó módot, akkor megjelenik a másik is. Valamint, próbáltam minél intuitívabb megoldásokkal előállni, mint például, ha a *ComboBox*-ban nincs elérhető elem, akkor ne lehessen rákattintani, hogy legördüljön egy üres lista, hanem már eleve legyen kikapcsolt állapotban és legyen beleírva, hogy nincs elérhető kamera a gépre kötve. A könnyebb használhatóság érdekében a kimeneti

könyvtárat egy könyvtár választó dialógus ablakban lehet megadni.
Mivel nem csak képeket, hanem videót is lehet rögzíteni ezért a rögzítés nevű gombon kívül kellett egy indítás és egy megállítás is, de hogy a felület egyszerűen átlátható maradjon, ezért ezeknek a láthatóságát is folyamatosan változtattam. Ha videó mód van kiválasztva, akkor csak a start és a stop látszik, ha pedig kép mód van kiválasztva, akkor csak a rögzítés látszik. Ezeknek az állítgatására egyetlen tulajdonságot használtam fel, viszont adatkötés során átalakítókat használtam, amik egy *Boolean* értéket alakítottak át *Visibility* típussá. Az egyik gombra egy olyan konvertert csináltam, ami igaz értékre ad látható értéket, a másik gombra pedig egy olyat, ami épp ellenkezőleg viselkedik, vagyis igaz értékre ad összecsukott állapotot.

5.5. Kamerák kalibrálása

A 2.6.4 alfejezetben ismertetett irodalom alapján első feladatom az volt, hogy készítsek egy kalibrációs objektumot. Azt hinnénk, hogy a kalibrációs minta kiválasztása egy egyértelmű feladat, de valójában több lehetőségünk van és pár dologra érdemes odafigyelni.



5.1. ábra.
Kalibrációs minták [33]

A 5.1 ábrán láthatjuk, hogy a standard sakktábla minta helyett választhatnánk akár körökből álló mintát is. Ebben a témában megoszló véleményeket olvastam, mivel sokak szerint köröket könnyebb lokalizálni, mint a belső sarokpontokat, de valaki meg épp ellenkezőleg, a sakktábla mintára esküszik meg. A fejlesztés során a sakktábla mintára esett a választásom, mivel az OpenCV is inkább ezt támogatja. A kalibrációs mintára vonatkozóan még van pár dolog, amit érdemes szem előtt tartani.

- Több négyzet, több belső sarokpontot jelent, ami pedig lehetőséget ad számunkra, hogy több 3D-2D pontmegfelelést kapjunk.
- A kalibrációs minta körül ajánlott egy szélesebb szegélyt hagyni.

- Ha a sztereó kamerák közti alapvonal nagy, akkor érdemes egy méretben nagyobb kalibrációs objektumot választani, mivel így a sztereó átfedés kisebb lesz.
- A kalibrációt segítő eszköztárnál (pl. OpenCV) érdemes a dokumentációban utánaolvasni, hogy támogat e tetszőleges sakktábla méretet, mivel léteznek olyanok, amik csak a [páratlan szám x páros szám] vagy a [páros szám x páratlan szám] formátumot támogatják.

A fent tárgyalt szempontok alapján egy A4-es lapra kinyomtattam a 6x9-es standard sakktábla mintát, amit az egyik polcomra ragasztottam fel és lefedtem egy átlátszó műanyaglappal. A 6x9-es méret nem a négyzetek számát jelöli, hanem a belső sarokpontjainak a számát. A polcra való rögzítést azért választottam, hogy egy stabil síkot adjon a kalibrációs mintának, valamint azért fedtem le egy átlátszó műanyag lappal, hogy:

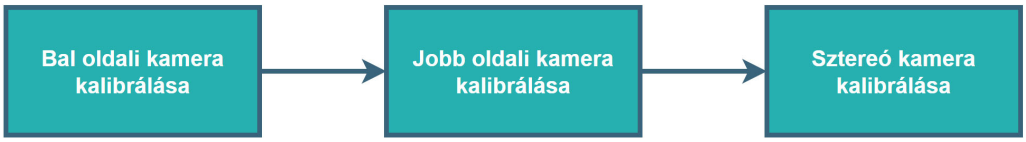
- ne sérüljön a papír felülete,
- a változó páratartalom ne legyen hatással a papírra,
- minél jobban rásimuljon a papír a felületre, így elkerülve az esetleges felgyűrődéseket, hullámokat, amik rontanák a kalibráció eredményét.

A következő lépés a kalibrációs képek rögzítése volt. Ez egy egyszerű feladatnak bizonyult, mivel ez a funkcionalitás már be lett építve a rendszerbe, aminek a lefejlesztéséről a 5.4 alfejezetben írtam. Azt gondolnánk, hogy a kalibrációs képek készítése egy nagyon triviális lépés, hiszen egy naiv megközelítés az lenne, ha csak elkezdenénk véletlenszerű képeket készíteni erről a kalibrációs objektumról. Ez természetesen kivitelezhető, de nem feltétlen fogunk jó eredményt kapni végeredményként, ezért összegyűjtöttem pár tanácsot, hogy mire kell figyelni, amikor egy sztereó kalibrációhoz szeretnénk képeket rögzíteni.

- A kamerák fókusza legyen rögzített, állandó.
- Különböző nézőpontokból és szögekből készítsünk képeket az objektumról, mivel így kaphatunk a fókusztávolságra és a torzítási paraméterekre egy jó eredményt.
- Az kalibrációs objektum távolsága lehet tetszőleges, de a kép legalább felét a mintának kell alkotnia.

- A kalibrációs objektumot úgy kell mozgatni, hogy lefedje a kamera teljes látómezőjét. A kép sarkai felé mozgatva segít jobban meghatározni a torzítási együtthatókat.
- Fordítsuk el több irányba a kalibrációs objektumot, ezzel a fókusztávolságot jobban meg tudjuk határozni.
- A fényviszonyok legyenek megfelelőek.

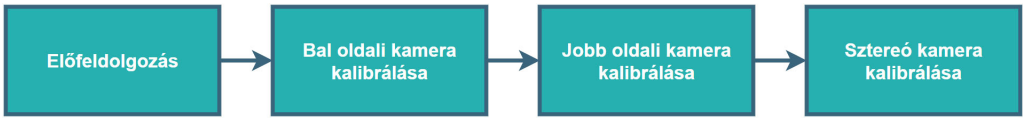
Csak ezután kezdődhetett meg a fejlesztés, ami a Python szkripttel indult. Ez felel azért, hogy a kalibrációs képek segítségével kiszámítsa a kamera belső és külső paramétereit. Mivel a következőkben mindenképpen a sztereó paraméterekre van szükségünk, ezért nem bontottam szét a funkcionalitást az egyes kamerákra. A szkript bemeneti argumentum listája tartalmazza az elérési utat a kalibrációs képekhez, a kalibrációs minta méretét, a kalibrációs mintát alkotó négyzetek oldalhossza milliméterben, valamint a kimeneti könyvtárat, ahova az egyes kamerák és a sztereó kamera paraméterei mentésre fognak kerülni.



5.2. ábra.
Kalibráció lépései

A 5.2 ábrán látható a kalibráció fő menete. Minden esetben először ki kell számolni a objektum pontokat, amik három dimenziós pontok a térben és a 2.6.4 alfejezetben tárgyaltak alapján, mivel a pontok egy síkon helyezkednek el, esetünkben a polcon, ezért a Z koordinátát nullának vettem. Ezután minden kalibrációs képet szürke skálás képpé alakítok és a jobb eredmény érdekében hisztogram kiegyenlítést végzek rajtuk, majd a 2.8 alfejezetben tárgyalt *findChessboardCorners* metódust felhasználva lokalizálom a sakktábla belső sarkait, majd egy utófeldolgozási lépés segítségével, finomítom azokat. Miután minden kalibrációs képet feldolgoztam és feltöltődött a két lista, ami tartalmazza a 3D objektum pontokat és a hozzájuk tartozó 2D képpontokat, elkezdhettem a kamera paramétereinek meghatározását. Erre szintén a 2.8 alfejezetben tárgyalt *calibrateCamera* metódust használtam fel, aminek eredményeképpen megkaptam a kamera belső és külső paramétereit. Ahhoz, hogy megtudjam határozni a kalibráció sikerességét, kiszámoltam az újravetítési hibát, ami a vetített és a mért pont közötti képtávolságnak megfelelő geometriai

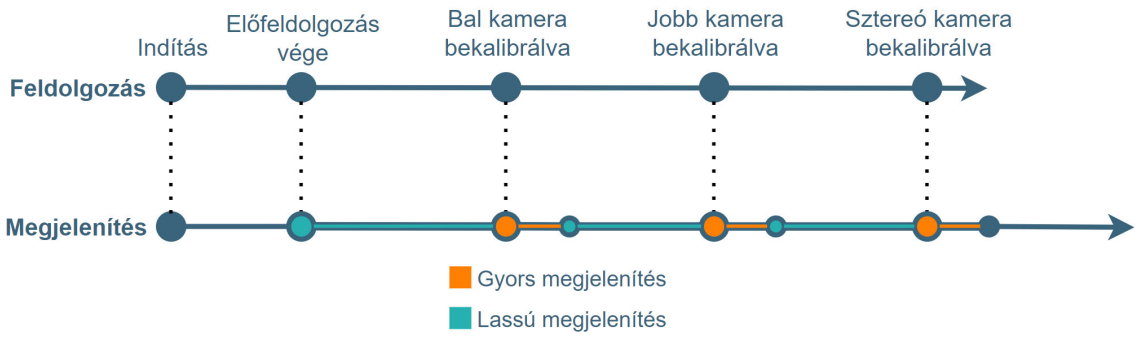
hiba. [34] Ezeket a műveleteket a bal, jobb és sztereó képek esetén is elvégeztem, az eredményeket pedig fájl-ba mentettem. Mivel ez egy elég hosszú folyamat, ezért a könnyebb nyomon követés érdekében beüzemeltem egy naplózót. Erre a feladatra a Python naplózó modulját használtam fel.
Hogy a folyamat minél látványosabb legyen, meg szerettem volna jeleníteni a kalibrációs képeket a lokalizált belső sarokpontokkal. Ez elsőre nem tűnt egy bonyolult feladatnak, mivel az OpenCV-nek van egy beépített függvénye arra, hogy ezeket a sarokpontokat fel tudjam rajzolni a képre. Mint a webkamera képek esetén, itt is az volt az elképzelésem, hogy egy ablakban megjelenítem a képeket, és ezt az ablakot húzom be a felhasználó felületre. Ez működött is a 5.3 alfejezetben tárgyalt modul segítségével, viszont, amikor a folyamat elért a kamera külső és belső paramétereinek meghatározásához, akkor a teljes felhasználói felület lefagyott, mivel szinkron módon működött, ezzel teljesen megakasztva a fő, UI szálat. Mivel a számítás sebessége túlnyomó részt a kalibrációs képek felbontásától és mennyiségétől függ, ezért ez a lépés több percig is eltarthat.



5.3. ábra.
Kalibráció lépései (módosított)

A lefagyás elkerülése érdekében át kellett szervezzem a modul főbb lépéseinek sorrendjét, ez a 5.3 ábrán látható. A 3D és 2D pontok kiszámítása ki lett szervezve egy előfeldolgozó lépésbe, így a számításigényes kalibrációk, csak ezután következnek. Az előfeldolgozás során rajzolom fel a képekre a lokalizált belső sarokpontokat, viszont nem megjelenítem őket közvetlenül, hanem fájlba mentem őket. Ez lehetőséget ad arra, hogy a megjelenítés teljesen elkülönüljön a feldolgozástól, így a felhasználói felület lefagyása kiküszöbölhető némi tárhely kapacitás kárára.
A felhasználói felületet a rögzítés funkcióhoz hasonló módon építettem fel, vagyis van egy kinyitható menüpanel a beállításoknak, ahol a felhasználó meg tudja adni a sakktábla méreteire vonatkozó adatokat, valamint egy fájlkezelő segítségével meg tudja adni a kalibrációs képek elérési útvonalát, illetve a kimeneti könyvtárat. Ezután rá tud menni az indítás gombra, ami elindítja a folyamatot. Mivel a kalibráció, akár percekig eltarthat, szerettem volna visszajelzést adni arról a felhasználónak, hogy éppen hol tart a program futása. Ezt úgy tudtam megtenni, hogy már korábban

beüzemeltem egy naplózó funkciót, így már csak egy szervizt kellett létrehozzak C# oldalon, ami futásidőben a legfrissebb napló állományt feldolgozza úgy, hogy közben nem blokkolja az adott napló fájlt, mert ha blokkolná, akkor közben a Python szkript nem tudná írni. Ez a szerviz aszinkron módon működik és egy **LogProcessor** nevű absztrakt osztályból származik le, ami szintjük szerint (kritikus hiba, figyelmeztetés, információ) tárolja a különféle napló bejegyzéseket, valamint tartalmaz egy delegáltat, amire a szervizt használók fel tudnak iratkozni. Amikor egy fontosabb bejegyzés kerül a naplóba, akkor a napló feldolgozó értesíti a delegált segítségével a feliratkozókat, (esetünkben a nézetmodell-t) és az értesítésben szerepel az értesítés típusára vonatkozó információ is, amit egy *LogNotification* nevű felsorolt típus segítségével valósítottam meg. Végeredményként pedig a felületen, státusz ikonok formájában jelenik meg ez az információ. A felhasználó láthatja, hogy éppen mely folyamat fut, melyek végeztek vagy várnak, illetve, hogy esetlegesen melyek futottak hibára. A felületen továbbá megjelennek az újravetítési hibák is az egyes kameráknál.



5.4. ábra.
Kalibrációs képek megjelenítése

Korábban írtam arról, hogy a megjelenítést külön tudtam választani a feldolgozástól oly módon, hogy az ne fagyassza meg a felületet. Ezt is egy szinttel tovább akartam gondolni. Ha beégetett időközönként jelenítem meg a képeket előfordul, hogy sokkal hamarabb végez a megjelenítés, mint a feldolgozás és ez fordított esetben is igaz, nem akarom feltartani a felhasználót a megjelenítéssel, ha már a feldolgozás végzett. Mivel a napló feldolgozó szerviz segítségével pontosan tudom, hogy éppen hol tart a feldolgozás, ezért képes vagyok szabályozni a megjelenítés sebességét. A 5.4 ábrán látható, hogy a két folyamat elkülönülve dolgozik. Az előfeldolgozás végén már biztosan rendelkezésünkre állnak a képek, így a megjelenítést megtudjuk kezdeni. Ameddig az adott kamera kalibrációja zajlik, addig a megjelenítés is lassú, viszont,

amint a feldolgozás végzett, a képek megjelenítése felgyorsul. Ezzel a működéssel tudom biztosítani azt, hogy egymáshoz képest a két folyamat ne érjen túl nagy eltéréssel véget.

5.6. Képek rektifikálása

Ez a modul felel azért, hogy egy, a már bekalibrált sztereó rendszerünkkel készített képet, rektifikálni és torzítás mentesíteni tudjunk. A Python szkript bemenete a sztereó kalibráció során kiszámolt paramétereket és együtthatókat tartalmazó fájl elérési útvonala, a sztereó rendszerrel készített sztereó kép, valamint a kimeneti könyvtár, ahová a torzítatlan és rektifikált képek mentésre fognak kerülni. Magáról a szkript működéséről nem lehet nagy terjedelmekben beszélni, hiszem már minden információ a rendelkezésünkre áll, már csak be kell olvasni az argumentumként megadott adatokat és a 2.8 alfejezetben tárgyalt *initUndistortRectifyMap* metódusnak ezeket átadva, megkapunk egy olyan transzformációs térkép, ami segítségével ki tudjuk korrigálni a sztereó képeket.
A felhasználói felületen meg lehet adni egy fájlkereső segítségével a sztereó paramétereket-, sztereó képet tartalmazó fájl-, valamint a kimeneti könyvtár elérési útvonalát. A *Rectify* gombra nyomva pedig megjelennek a nyers bal- és jobb oldali, valamint a rektifikált és torzítatlan bal- és jobb oldali képek.
Hogy minél jobban lehessen szemléltetni a rektifikáció sikerességét, megjelenítés során felrajzolok a képek fölé néhány horizontális vonalat, amik segítségével láthatjuk, hogy a két képen, az egymásnak megegyező képpontok valóban egy függőleges koordinátára esnek e.

5.7. Diszparitás térkép kiszámolása

Ez a modul felel azért, hogy meghatározza a diszparitás térképet a sztereó képekre. A funkcionalitás fejlesztése során a fő célom az volt, hogy a felhasználó a teljes folyamatot képes legyen vezérelni, vagyis, minden számítás során használt paraméter kívülről beállítható legyen, ne pedig beégetett értékekkel működjön. Argumentumként megadható a rektifikált sztereó kép és a kimeneti könyvtár, ahová a diszparitás térkép le lesz mentve. A feldolgozást befolyásoló argumentumként megadható paraméterek:

- minDisparity
- numDisparities
- blockSize
- windowSize
- disp12MaxDiff
- preFilterCap
- uniquenessRatio
- speckleWindowSize
- speckleRange
- mode
- lambda
- sigma

Az egyes paraméterek jelentését a 2.8 alfejezetben tárgyaltam. A diszparitás térkép számítására két osztályt nyújt az OpenCV, a blokkillesztési algoritmusra épülő *StereoBM*-t és a félig-globális illesztés algoritmusát megvalósító *StereoSGBM*-t. Az előbbivel szemben, a *StereoSGBM* egy teljesebb diszparitás térképet hoz létre, való igaz, hogy cserében számításigényesebb, de jelen célkitűzéseink nem térnek ki valós idejű működésre, így ezt az algoritmust választottam. A képek beolvasása után lehetőség van leskálázni a méretüket, ezzel is felgyorsítva az illesztést, de mivel én a lehető legjobb minőséget szeretném elérni, ezért ezt a lépést kihagytam. Ezt követően már egyből ki is tudnánk számolni a diszparitás térképet, viszont a jobb eredmény érdekében egy utószűréssel egészítettem ki. Szűrőnek a súlyozott legkisebb négyzetek módszerét választottam, ami egy nem lineáris, élvédő, simító szűrő. Az argumentumlistában szereplő, eddig még nem említett két paraméter, lambda és szigma, ehhez az algoritmushoz tartozik. A szűrés során a lambda határozza meg a regularizáció mértékét. Nagyobb értékek arra kényszerítik a diszparitás térkép éleit, hogy egyre jobban illeszkedjenek az eredeti kép éleihez. A szigma pedig azt határozza meg, hogy a szűrési folyamat mennyire érzékeny a forráskép éleire. Ahhoz, hogy a szűrő segítségével megkaphassam a kívánt diszparitás térképet, létre kellett hozni egy bal és egy jobboldali diszparitás térképet, amikhez egy bal és egy jobb oldali illesztőre volt szükségem. A bal oldali illesztőt a korábban említett *StereoSGBM* segítségével hoztam létre, itt állítom be az argumentumként átadott paramétereket. A diszparitás térkép egy normalizáláson is átmegy, hogy esetlegesen kompatibilis legyen más metódusokkal is, valamint, hogy a felhasználó számára jobban látható értékek jelenjenek meg, ezért még hisztogram kiegyenlítést is végrehajtottam rajta. A kimeneti képek ezután pedig a kimeneti könyvtárba kerülnek mentésre.
A felhasználói felületen ennél a funkciónál érződik a legjobban, hogy miért is volt hasznos lefejleszteni a kinyitható panelt, mivel rengeteg bemeneti vezérlőnk van és ez elfoglalná alap esetben a nézet túlnyomó részét. A felület kialakításánál a leg-

nagyobb problémát az jelentette, hogy a különféle paraméterhez a szakirodalomban elég nehezen találni konkrét ajánlásokat, hogy az egyes paraméterek értékei milyen tartományban mozogjanak, illetve, hogy milyen léptékkel. Ezeket egy elég hosszadalmas keresgélés eredményeként tudtam csak beállítani.

5.8. 3D rekonstrukció

Ennél a modulnál ér össze a sztereó látás a pózbecsléssel, vagyis itt kell tudjuk kombinálni a detektált kulcspontok koordinátáit a diszparitás térkép értékeivel. Mivel a pózbecslési algoritmus a kép koordináta rendszerében lokalizálja az emberek kulcspontjait, így egy listányi (x,y) koordinátapárost fogunk visszakapni tőle eredményként. Ezeket a koordinátákat kell kiegészítsük a diszparitás térkép értékei segítségével, ezzel egy z koordinátát megkapva.
Az irodalomkutatás során kitértem arra, hogy végezetül pózbecslésre a Mask R-CNN-t választottam, mivel ebben látom a legnagyobb lehetőséget a későbbi továbbfejlesztési lehetőségek esetében. Viszont, mivel a különböző pózbecslési modellek által kinyert kulcspontok formátuma közel azonos, ezért itt is arra próbáltam törekedni, hogy egy általánosan használható megoldást adjak és akár legyen lehetőség arra, hogy a közeljövőben ezeket a modelleket cserélgetni tudjam úgy, hogy a kód többi részét ne érintse ez a változás.
Az első nehézségeket a függőségek feloldása okozta, mivel hiába fekete dobozként kezelem a modell-t, attól még az Anaconda virtuális környezetében minden általa használt függőségnek szerepelnie kellett, ami első körben egy egyszerű feladatnak hangzik, viszont a modell verziója nem követte a függőségek verzióját, így hiába telepítettem fel a legfrissebb csomagokat, ha olyan funkciók kerültek ki belőlük, amiket a modell-még használ. Minden hibaüzenet után próbáltam megfejteni, hogy a függvény, ami a hibát dobta, melyik verziójú csomagban érhető el. Voltak olyan függőségek, amik függőségeit szintén egy korábbi verzióra kellett visszaállítanom. Ez egy lassú folyamat volt, de úgy gondolom, hogy ez is a tapasztalataim tárházát bővítette. Első körben tesztképek segítségével néztem meg a működését. Hiába tanultam már a témáról, viszont még soha nem használtam ilyen modelleket a gyakorlatban és el kell mondjam, hogy még mindig lenyűgöznek.
A következő feladatom volt a kulcspontok és végtagok megjelenítése. Alapesetben ez nem egy bonyolult folyamat, mivel a Mask R-CNN által nyújtott demo alkalmazás tartalmazta, hogy hogyan kell megjeleníteni őket a bemeneti képen,

viszont én térben szerettem volna ugyan ezt megtenni. Kis kutakodás után erre két megoldást találtam. Az első a Matplotlib [35] modul segítségével, amit elsősorban kétdimenziós ábrázolásra terveztek, de későbbi verzióiban megjelent néhány háromdimenziós ábrázoló segédprogram. A második lehetőséget a PyQtGraph [37] 3D grafikai rendszere adta, ami az OpenGL [38] fix-funkciós moduljára épül. Azt hinnénk, hogy mivel sokkal fókuszáltabb funkcionalitást nyújt a Matplotlib, ezért gyorsabbnak kell lennie, viszont mivel a PyQtGraph a NumPy alapjaira építkezik, ezért számunkra sebességben lényegében nincs számottevő különbség. Végül a PyQtGraph mellett döntöttem a továbbfejlesztési lehetőségeket szem előtt tartva, így neki is láttam a beüzemelésének.

5.5. ábra.
3D rekonstrukció folyamata

A 5.5 ábrán látható a Python szkript működésének folyamata. Argumentumként meg kell adni az ablak nevét, a rektifikált kép, valamint a diszparitás térkép elérési útvonalát. Az első lépések magában foglalják a megjelenített ablakkal-, 3D térrel és a modellel kapcsolatos adatok beállítását, ezt követően pedig az argumentumként beadott képen detektálom az emberek kulcspontjait. A 2D koordináták alapján kiolvasom a diszparitás térkép adott pozícióján szereplő értéket, amit egy tényező segítségével hatványozok, arra az esetre, ha jobban szét akarnánk húzni az értékeket a Z tengely mentén. Mivel már rendelkezésünkre állnak a 3D koordináták, már csak össze kell kössük őket, ezt pedig egy váz szerkezet alapján tudjuk megtenni, amit előre definiálnunk kell. Ezeket a kapcsolatokat először a modellhez mellékelt példa programból néztem ki, viszont nem voltam teljesen megelégedve az eredménnyel, így némileg személyre szabtam, kiegészítettem új pontokkal, például a vállak közé raktam egy új pontot, amit úgy kaptam meg, hogy a két váll koordinátáját összeadtam és elosztottam kettővel. Ezután már csak a térhez hozzá kellett adnom a kulcspon-

tokat és a végtagokat reprezentáló vonalakat, majd megjelenítettem az ablakot. Naiv módon azt hittem, hogy a felhasználói felületbe való integrálás könnyű feladat lesz, hiszen többször bebizonyosodott, hogy az eddig ismertetett módszer működik, értem ez alatt a 5.3 alfejezetben tárgyalt *PythonProcessHost* osztályt. A futtatható állományt a megszokott módon indítottam, viszont amikor a program elérkezett arra a pontra, hogy kész felhelyezni az ablakot a felhasználói felületre, akkor egy nem várt hibával leállt a program futása. Az alapvető problémát az adja, hogy a Python folyamatból származó ablakot próbálom beágyazni a WPF folyamatból származó ablak gyermekeként, viszont a Python által megjelenített ablak nem tud arról, hogy részt vesz ebben a folyamatokon átívelő szülő/gyermek kapcsolatban. Ez az ablakok közti együttműködés egyszerű OpenCV által megjelenített képeknél még támogatva volt, viszont a PyQtGraph működéséből adódóan ez már nincs garantálva és lényegében nulla esélyem van arra, hogy ezt befolyásolni tudjam, így sajnos egyenlőre ez a funkció külön ablakban jelenik meg.

6. TESZTELÉS

Ebben a fejezetben az egyes komponensekre vonatkozó teszteseteket fogom leírni táblázatos formában. Fontos kiemelni, hogy annak ellenére, hogy a fejlesztés során külön törekedtem arra, hogy az alkalmazás egység tesztelhető legyen (pl. C# oldalról a statikus függőségeket próbáltam interfész mögé elrejteni), még így is nagyon kevés olyan osztály vagy metódus van, amire lehetne automatizált tesztek írni. Ezt azzal lehet magyarázni, hogy C# oldalon viszonylag kevés üzleti logika van, hiszen nagyrészt csak arra használom, hogy egy keretet adjon a szkriptek be- és kimenetének. Python oldalról pedig azért nehéz automatizált teszteket írni, mivel a működés során nagyon sok külső modul-t felhasználok, többek között az OpenCV-t. Továbbá a metódusok vagy függvények túlnyomó része függ a bemenetként szolgáltatott képektől, amik eredményét nem tudjuk összevetni már létező referencia értékkel, mivel nem áll rendelkezésünkre ilyen. Ezért a tesztelés nagy része manuálisan, smoke teszt formájában fog történni, vagyis a tesztesetek a UI szinten, a felhasználói interakciók köré fognak íródni.

6.1. Felhasználói felület tesztelése

A felhasználói felülettel kapcsolatban egy alapvető funkcionalitást kell teszteljünk, mégpedig, hogy a vertikális menü megfelelően működik e.

Előfeltétel	Teszteset leírása	Elvárt viselkedés	Teljesítés
A főmenü nyitott állapotban van	Rákattintunk a hamburger gombra	A főmenü csukott állapotba kerül	✔
A főmenü csukott állapotban van	Rákattintunk a hamburger gombra	A főmenü nyitott állapotba kerül	✔
	Rákattintunk a „Otthon” menüre	Megjelenik a „Otthon” menüpont	✔
	Rákattintunk a „Rögzítés” menüre	Megjelenik a „Rögzítés” menüpont	✖
	Rákattintunk a „Kalibráció” menüre	Megjelenik a „Kalibráció” menüpont	✔
	Rákattintunk a „Rektifikáció” menüre	Megjelenik a „Rektifikáció” menüpont	✔
	Rákattintunk a „Diszparitás térkép” menüre	Megjelenik a „Diszparitás térkép” menüpont	✔
	Rákattintunk a „3D Rekonstruálás” menüre	Megjelenik a „3D Rekonstruálás” menüpont	✔

6.1. táblázat.
Főmenü tesztelése

Egy tesztesetet leszámítva az összes funkció megfelelően működik, kivéve a Rögzítés nézetre való navigáció, mivel úgy néz ki, hogy a fejlesztés során véletlen kitöröltem azt a sort, ami összeköti a nézetmodell típust a neki megfelelő nézettel. Ezt a hibát javítottam.

6.2. Képeket rögzítő komponens tesztelése

Ennek a komponensnek a tesztelése különleges abból a szempontból, hogy bemeneti eszközökön, a számítógépre csatlakoztatott kamerákon, webkamerákon keresztül nyújtjuk a funkcionalitást, így szükséges a külső behatásokat is tesztelnünk. Továbbá, mivel itt jelenik meg először a beállításokat magába záró panel, ezért

annak működését is tesztelnünk kell.

Előfeltétel	**Teszteset leírása**	**Elvárt viselkedés**	**Teljesítés**
A beállítások panel csukott állapotban van	Rákattintunk a fogaskerék gombra	A beállítások panel nyitott állapotba kerül	✔
A beállítások panel csukott állapotban van	Rákattintunk a fogaskerék gombra	A beállítások panel nyitott állapotba kerül	✔
Nincsen a gépre csatlakoztatott kamera eszköz	Rákattintunk a megnyitás gombra	Felugró ablakban hibaüzenet jelenik meg	✔
Bal/Jobb opció van kiválasztva	Rákattintunk a Sztereó opcióra	Megjelenik mindkét legördülő lista	✔
Sztereó opció van kiválasztva	Rákattintunk a Bal/Jobb opcióra	Csak az adott oldalhoz tartozó legördülő lista jelenik meg	✔
	Rákattintunk a böngészés gombra	Megnyílik egy könyvtár böngésző ablak	✔
Vannak a gépre csatlakoztatott kamera eszközök	Rákattintunk a legördülő listára	Megjelenik minden elérhető kamera eszköz	✔

6.2. táblázat.
Rögzítés felületének tesztelése

Előfeltétel	Teszteset leírása	Elvárt viselkedés	Teljesítés
Vannak a gépre csatlakoztatott kamera eszközök és megadtunk minden szükséges adatot	Rákattintunk a megnyitás gombra	Megjelennek a kamera/kamerák adása a felület belsejében	✔
Képek készítése opció van kiválasztva és megy a kamera	Rákattintunk a rögzítés gombra	A kimeneti könyvtárba megjelenik a kép	✔
Videó készítése opció van kiválasztva és megy a kamera	Rákattintunk az indítás gombra, majd kis idő múlva a leállításra	A kimeneti könyvtárba megjelenik a videó	✔
Megy a kamera	Kihúzzuk a gépből az eszközt	Nem törik össze az alkalmazás	✖

6.3. táblázat.
Rögzítés funkciójának tesztelése

A tesztelés során kiderült, hogy a külső behatások kezelését a fejlesztés során kifelejtettem. Ha megszakad a kapcsolat a kamerával, akkor nem tudok egy alapértelmezett értékre állni, mivel a működéshez elengedhetetlen. Valószínűleg az sem segítene, ha bizonyos időközönként megpróbálnánk újra csatlakozni rá, mivel lehet, hogy az adott indexen már más eszköz helyezkedik el. Így, ha kivétel keletkezik, akkor egy rendszer hiba kóddal leállítom a folyamatot, így nem kezeletlen a kivétel és nem törik össze az alkalmazás.
Hiába összetett a nézetmodell működése, de nem tartalmaz egység tesztelhető kódot. Ez azzal magyarázható, hogy a funkcionalitás jelentős része a *PythonProcessHost* osztályt használja, ami hiába egy interfész mögött van, de nem tudnánk lecserélni egy mockra/stubra. Ha úgy döntenénk, hogy a *PythonProcessHost* osztállyal szeretnénk integrációs tesztet írni, akkor is előjönne az a probléma, hogy a kameráktól még mindig függünk. A funkcionalitás másik részében viszont szimpla adatmódosításokat végzünk a rengeteg tulajdonságon, amikre a felhasználói felület működéséhez szükségünk van, ezekre lehetne egységteszteket írni, viszont mivel ezek

mindegyike privát metódus, ezért felmerülne az a kérdés, hogy megérné e ebben az esetben szembe menni az objektum orientált elvekkel. Azokat a metódusokat tudtam egység tesztelni, amit a *string* típus kiterjesztéseként írtam. Az NUnit [39] nevű egység tesztelési keretrendszert használtam és mivel nincs külső függőségem, ezért ezen kívül másra nem volt szükségem.
Az első ilyen *string* kiterjesztési metódus a *FirstCharToUpper*, aminek nincs külön bemeneti paramétere, csak maga a szöveg, amire hívják.

Szöveg	**Elvárt**	**Teljesíti**
„szöveg”	„Szöveg”	✔
„c”	„C”	✔
„cC”	„CC”	✔
null	ArgumentNullException	✔
„”	ArgumentException	✔

6.4. táblázat.
FirstCharToUpper metódus egység tesztelése

A másik kiterjesztési metódus a *GetTextAfterKeyword*, aminek már van bemeneti paramétere, mégpedig egy kulcsszó.

Szöveg	**Kulcsszó**	**Elvárt**	**Teljesíti**
„Egy hosszú szöveg”	„hosszú”	„ szöveg”	✔
„Egy szöveg”	„alma”	„Egy szöveg”	✔
„”	„alma”	„”	✔
„Egy szöveg”	„”	„Egy szöveg”	✔
null	„Alma”	ArgumentNullException	✔
„Alma”	null	ArgumentNullException	✔

6.5. táblázat.
GetTextAfterKeyword metódus egység tesztelése

6.3. Kalibrációs komponens tesztelése

Ennél a komponensnél szerepelnek olyan tesztesetek, amiket már az előző alfejezetben lefedtem, gondolok itt például a böngészés gombra való kattintásra, ahol azt

nézem, hogy megnyílik e a könyvtár kereső ablak. Természetesen itt is fontos ezeket tesztelni, de az olvashatóság érdekében nem fogalom őket újra táblázatba.

Előfeltétel	**Teszteset leírása**	**Elvárt viselkedés**	**Teljesítés**
	Valamely bemeneti mezőbe beírunk egy kétszámjegyű értéket	A mezőt elhagyva az érték megmarad	✔
	Valamely bemeneti mezőbe beírunk egy hibás értéket	A mezőt elhagyva az eredeti érték marad meg	✔
Minden paramétert beállítottunk	Rákattintunk a megnyitás gombra	Elindul a kalibráció és megjelennek a kalibrációs képek	✔
Fut a kalibráció, valamint az egyik kalibráció végzett		A képek felgyorsulva jelennek meg	✔
Fut a kalibráció		A státusz ikonok megfelelő sorrendben váltják egymást	✔
Fut a kalibráció	Megnyomjuk a leállítás gombot	A végzett státuszon kívül minden ikon hibát jelez	✔

6.6. táblázat.
Kalibráció felületének tesztelése

Előfeltétel	**Teszteset leírása**	**Elvárt viselkedés**	**Teljesítés**
Minden paramétert beállítottunk	Rákattintunk a megnyitás gombra	Elindul a kalibráció, aminek végén, a kimeneti könyvtárba megjelennek a fájlok	✔

6.7. táblázat.
Kalibráció funkciójának tesztelése

A nézetmodellben több metódus egység tesztelésére is lehetőség volt. Már korábban említettem, hogy minden nézetmodell függősége a *PythonProcessHost* osztályt, hiszen elsősorban ezzel vagyunk képesek megjeleníteni és elindítani a Python alkalmazásokat. Ezen felül van még egy függősége a nézetmodellnek, amit azért hoztam létre, hogy kiszervezzem a statikus függőségeket a kódból, így lehetőséget adva az egység tesztek írására. Ennek az osztálynak az implementációját elrejtettem egy interfész mögé. Erre azért volt szükség, mivel így a Moq [40] nevű keretrendszer segítségével az egyes tesztesetekre saját viselkedést írhatok, valamint ellenőrizhetem, hogy az interfészen keresztül, milyen hívások történtek.
A felhasználó által bevitt érték ellenőrzését a *IsValidNumberInput* nevű metódus segítségével végzem, aminek bemenete egy szöveg, és egy reguláris kifejezés által leellenőrzöm, hogy egy érvényes egy/két számjegyű érték e vagy sem.

Szöveg	**Elvárt**	**Teljesíti**
„1”	Igaz	✔
„9”	Igaz	✔
„10”	Igaz	✔
„99”	Igaz	✔
„100”	Hamis	✔
„”	Igaz	✔
null	ArgumentNullException	✖

6.8. táblázat.
IsValidNumberInput metódus egység tesztelése

Az egység tesztelés során kiderült, hogy nem figyeltem arra, hogy a reguláris kifejezés illesztésénél a null bemeneti érték *ArgumentNullException* hibát vált ki. Ezt javítottam, most már null bemenet esetén hamis értéket ad vissza a metódus.
A következő metódus a *GetSubstringIfPossible*, ami a szöveg adott indexétől számítva megpróbál visszaadni egy X hosszú szöveget. Ha esetlegesen túlindexelés történne, akkor vissza kéne adnia az eredeti szöveget.

Szöveg	Index	Hossz	Elvárt	Teljesíti
„0123456789”	0	4	„0123”	✔
„0123456789”	0	0	„”	✔
„0123456789”	10	0	„”	✖
„0123456789”	4	2	„45”	✔
„0123456789”	10	1	„0123456789”	✔

6.9. táblázat.

GetSubstringIfPossible metódus egység tesztelése

A hiba azért keletkezett, mivel az egyenlőtlenség vizsgálat nem volt jól megadva. A hiba javításra került.

Az alfejezetben említettem a függőségeket és hogy miért szükséges a Moq használata. A soron következő *IsValidFileCount* metódusban eleinte szerepelt egy .Net keretrendszerben található statikus osztály statikus metódusa. A fejlesztés végén, a refaktorálás során kiszerveztem egy interfész mögé, így mock objektumot tudok belőle csinálni és minden egység teszt elején be tudom üzemelni az adott tesztesetnek megfelelően. Maga a tesztelendő metódus bemenetként kötelezően vár egy elérési útvonalat, valamint opcionálisan egy másik elérési útvonalat, illetve egy számot. Ezt szintén validációra használom, hogy megvizsgáljam, hogy megfelelő számú kalibrációs képek állnak e rendelkezésre a megadott útvonalakon, illetve sztereó képek esetében képpárokról beszélünk e vagy sem. A táblázat olvashatósága érdekében a könyvtár elérési útvonalakat nem tüntetem fel, azoknak létezését egy másik metódus segítségével ellenőrzöm.

Fájlok száma #1	Fájlok száma #2	Min. érték	Elvárt	Teljesíti
10	10	10	Igaz	✔
11	10	10	Igaz	✔
10	11	10	Igaz	✔
9	9	10	Hamis	✔
10	9	10	Hamis	✔
9	10	10	Hamis	✔
-1	1	0	Hamis	✔
1	-1	0	Hamis	✔

6.10. táblázat.
IsValidFileCount metódus egység tesztelése

6.4. Rektifikáló komponens tesztelése

A program hátra lévő komponenseinket automatizált tesztelése nem megoldható, mivel itt már nagyon hangsúlyos a képtől és az OpenCV-től való függés, ezért itt is smoke teszt jelleggel történik a tesztelés.

Előfeltétel	Teszteset leírása	Elvárt viselkedés	Teljesítés
Minden paramétert beállítottunk	Rákattintunk a megnyitás gombra	Megjelennek a bemeneti-, valamint a rektifikált képek a vízszintes vonalakkal	✔

6.11. táblázat.
Rektifikáció felületének tesztelése

Előfeltétel	Teszteset leírása	Elvárt viselkedés	Teljesítés
Minden paramétert beállítottunk	Rákattintunk a megnyitás gombra	A kimeneti könyvtárba, megjelennek a rektifikált képek	✔

6.12. táblázat.
Rektifikáció funkciójának tesztelése

6.5. Diszparitás térkép komponens tesztelése

A fejlesztés során azért döntöttem a csúszka vezérlők használata mellett, hogy elkerüljem az esetleges felhasználói hibákat, mivel a vezérlő maga kikényszeríti a megfelelő használatát. Olyan megkötéseket ad, mint a minimum és maximum érték, valamint a lépték. Ez továbbá azért is előnyös, mivel külön kódban sem kellett validációt implementáljak és mivel rengeteg különféle érték van, ennek az ellenőrzése hibák előfordulására adhatott volna lehetőséget.

Előfeltétel	**Teszteset leírása**	**Elvárt viselkedés**	**Teljesítés**
Minden paramétert beállítottunk	Rákattintunk a számolás gombra	A felületen megjelenik a diszparitás térkép	✔
Már meg van jelenítve egy diszparitás térkép. Módosítunk a paramétereken	Rákattintunk a számolás gombra	A felületen megjelenik a diszparitás térkép, és látszik eltérés az előzőhöz képest	✔

6.13. táblázat.
Diszparitás térkép felületének tesztelése

Előfeltétel	**Teszteset leírása**	**Elvárt viselkedés**	**Teljesítés**
Minden paramétert beállítottunk	Rákattintunk a számolás gombra	A kimeneti könyvtárba megjelenik a diszparitás térkép	✔

6.14. táblázat.
Diszparitás térkép funkciójának tesztelése

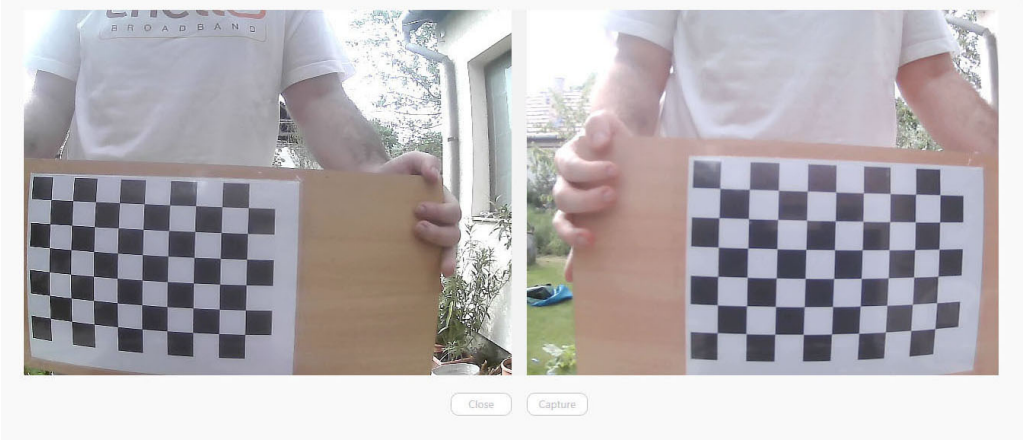
6.6. 3D rekonstrukciós komponens tesztelése

Ennél a komponensnél nem csak az OpenCV-től való függést, de a Mask R-CNN-től való függést sem tudjuk áthidalni, hogy esetlegesen automatizált teszteket írhassunk.

Előfeltétel	**Teszteset leírása**	**Elvárt viselkedés**	**Teljesítés**
Olyan képeket állítottunk be, amiken található személy	Rákattintunk a számolás gombra	Egy külön ablakban megjelenik a 3D rekonstruált jelenet a csontvázakkal	✔
Olyan képeket állítottunk be, amiknek eltér a mérete	Rákattintunk a számolás gombra	Egy külön ablakban hibaüzenetet kapunk	✔
Olyan képeket állítottunk be, amiken nem található személy	Rákattintunk a számolás gombra	Az ablak hiba nélkül jelenik meg, viszont nem tartalmaz csontvázakat	✔
Helyes képeket állítottunk be	Rákattintunk a számolás gombra	A csontvázakon megfigyelhető Z tengely menti eltérés	✔

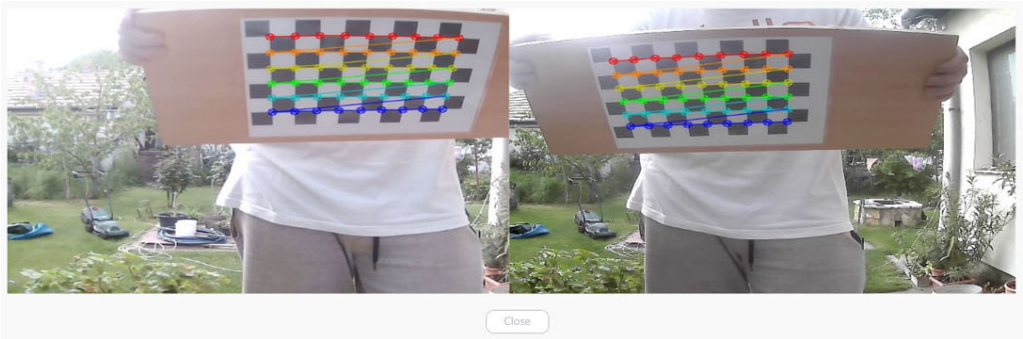
6.15. táblázat.
3D rekonstrukció funkciójának tesztelése

7. ALKALMAZÁS BEMUTATÁSA



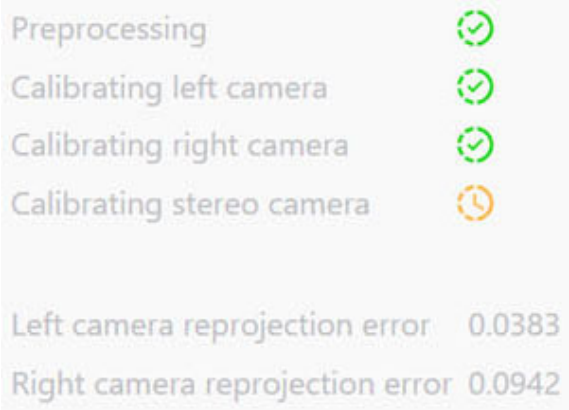
7.1. ábra.
Bemutatás - Rögzítés

A *Rögzítés* menüpontban, miután megadtuk, hogy sztereó kamerára szeretnénk képeket rögzíteni a 7.1 ábrának megfelelően megjelenik a sztereó kamera képe. Alul, a *Capture* gombra kattintva kerülnek mentésre a képek.



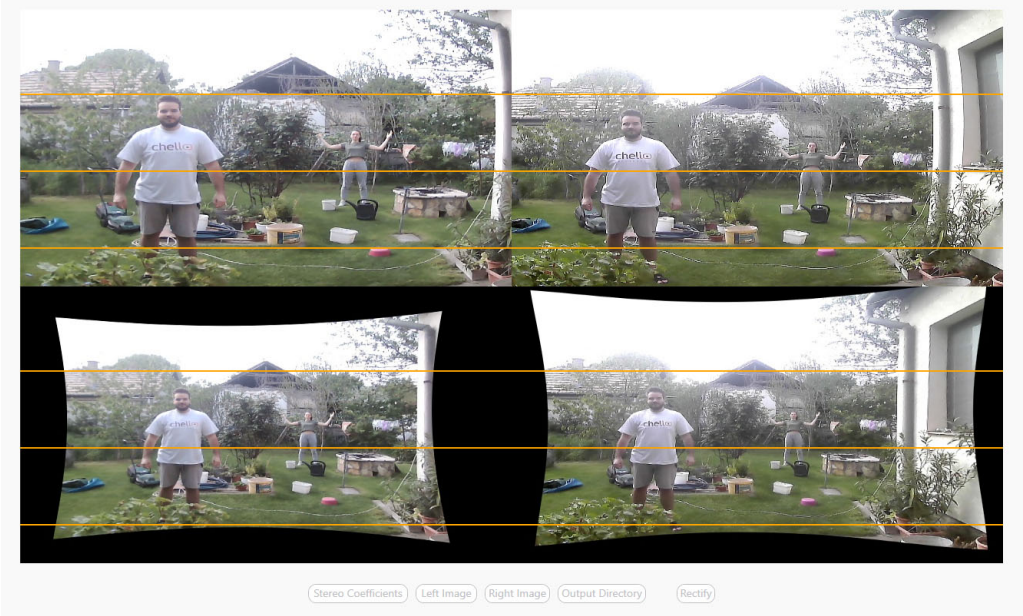
7.2. ábra.
Bemutatás - Kalibráció

A kalibráció folyamata a 7.2 ábrán látható. Jól mutatja, hogy az aktuális jeleneten sikeresen tudtuk lokalizálni a sakktábla belső sarokpontjait.



7.3. ábra.
Bemutatás - Kalibráció információi

A 7.3 ábrán a kalibráció fő lépései láthatóak. Jelen esetben már túl vagyunk az előfeldolgozáson, illetve az egyes kamerák kalibrálásán. Az újravetítési hibák is megjelentek már.

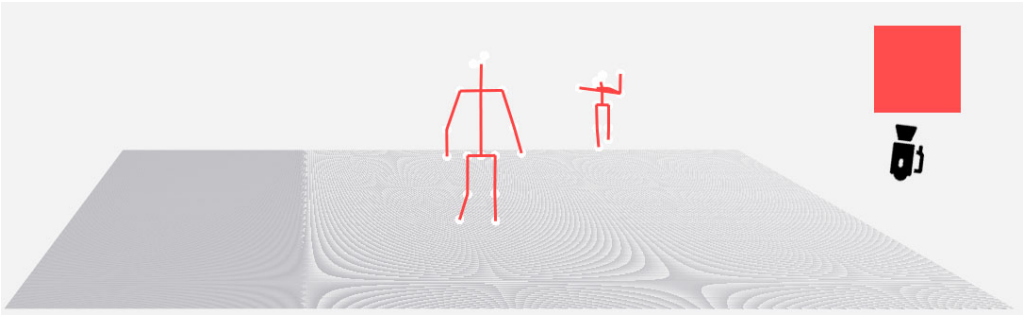


7.4. ábra.
Bemutatás - Sztereó rektifikáció

A 7.4 ábrán a *Sztereó rektifikáció* menüpont látható. A fenti, rektifikáció előtti képeken, az egymással megegyező jellemző pontok teljesen más függőleges koordinátán helyezkednek el. Megfigyelhető, hogy a sztereó rektifikálás után ezek már egy epipoláris vonalra esnek. Továbbá az alsó képeken, a fekete részek jól mutatják, hogy mennyit kellett a képeken korrigálni.

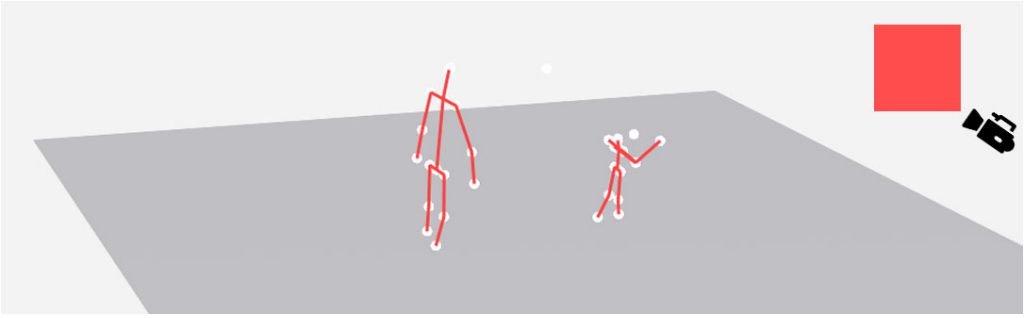
7.5. ábra.
Bemutatás - Diszparitástérkép

A 7.5 ábrán a *Diszparitástérkép* menüpont belseje látszódik. A tömérdek paraméter beállítása után a különbségtérkép megjelent. Látható, hogy a közelebbi objektumok világos-, míg a távolabbiak egyre sötétebb színnel jelennek meg.



7.6. ábra.
Bemutatás - 3D újraépítés (szemből)

A 7.6 ábrán a *3D újraépítés* menüpont által megjelenített háromdimenziós tér látható, miután megadtuk az alkalmazásnak a rektifikált baloldali képet, illetve a baloldali képre illesztett különbségtérképet. A jobb felső sarokban szereplő ábra utólag lett rászerkesztve a képre, hogy segítse a megértést.



7.7. ábra.
Bemutatás - 3D újraépítés (oldalról)

A 7.7 ábrán látható, hogy valóban sikerült megjelenítsük a képen szereplő személyek egymáshoz viszonyított helyzetét.

8. ÉRTÉKELÉS

Legvégül pedig az egyes funkcionalitásokat, módszereket, illetve megoldásokat szeretném értékelni a követelményspecifikáció alapján.

Funkcionalitás	**Teljesítés**
Könnyen használható felhasználó felület	✔
Képek és videó rögzítése	✔
Kamerák kalibrációja	✔
Kalibráció sikerességének számszerűsítése	✔
Képek korrigálása	✔
Különbségtérkép számítása	✔
Csontvázak lokalizálása	✔
Különbségtérkép illesztése a kulcspontokra	✔
3D vizualizáció	✔

8.1. táblázat.
Kiértékelés - Funkcionális követelmények

Funkcionalitás	**Teljesítés**
Windows kompatibilitás	✔
Kód dokumentációja	✔
Processzoron történő számítások	✔
Függőségek minimalizálása	✖

8.2. táblázat.
Kiértékelés - Nem funkcionális követelmények

Az irodalom kutatás alapján történő programozási nyelv kiválasztása utólag sikeresnek értékelhető, a Python nagyon hatékonynak bizonyult a feladatokra.
A felhasználói felület által könnyen el tudjuk érni az egyes funkciókat. Technikai oldalról nézve a WPF-ben való fejlesztés nagyon gördülékeny volt, viszont a problémák akkor kezdtek előjönni, amikor a Python komponensek integrációját végeztem. Új folyamatok indítása egy költséges lépés, ami megkerülhető lett volna, viszont akkor le kellett volna mondjak arról, hogy az egyes funkciók kimenete az ablakon belül jelenjen meg.

Sztereó képek, illetve videók rögzítésére egy hatékony, valamint felhasználói szemszögből nagyon könnyen használható megoldást sikerült találjak.
Már előre rögzített kalibrációs képek segítségével a kamerák kalibrációjára lehetőség van. A kimeneten megjelennek az egyes kamerák, illetve a sztereó kamera külső és belső paraméterei, amit bármikor vissza is tudunk olvasni. Az újravetítési hibák, amiket a kalibráció végén kaptam az elvárható értékhatáron belül voltak, figyelembe véve, hogy széles látószögű kamerák esetén ez a hibahatár kitolódik. A sakktáblás kalibrációs minta az eredményekre való tekintettel egy megfelelő választás volt, idő hiányában a többi módszer kipróbálására már nem volt lehetőségem.
A kamerák külső és belső paraméterei segítségével képes voltam torzítatlanná és rektifikálttá tenni a képeket. A rektifikáció sikerességét számszerűsítve nem lenne lehetetlen értékelni, viszont egy új komponens lefejlesztését igényelné, amire a fejlesztési idő korlátai miatt nem volt már lehetőségem.
Diszparitás térkép számítására lehetőség van, ezt szintén a felhasználói felületről tudjuk indítani, valamint az egyes paramétereket könnyedén tudjuk állítani. A komponens adott kamerarendszerre történő felparaméterezése esetén az olyan objektumok, amik rendelkeznek kellő mennyiségű jellemzőponttal, azok megfelelőnek tekinthető értékkel rajzolódnak ki. Ellenben, ahol már nem áll rendelkezésre kellő számú jellemzőpont, például a homogén felületeknél, vagy a fényviszonyoknak nem megfelelő területeknél, ott a diszparitás értékek helytelenül jelennek meg. Ennél a komponensnél kipróbáltam többféle módszert, algoritmust, próbáltam előfeldolgozással javítani a képet és utófeldolgozással a diszparitás térképet, próbáltam a kamera rendszer fizikai elhelyezkedésénél változtatni a fényviszonyokon, de ezek vagy rontottak az eredményen, vagy nagyon keveset segítettek. Minden próbálkozásomnál egy állandó volt, mégpedig a kamerák. Ezeknek a minősége közel sem ideális, hiszen belépő szintű webkamerákról van szó. Teljesen átfogó értékelést csak akkor tudnék adni, ha jobb minőségű kamerákkal dolgoztam volna, de sajnos erre nem volt lehetőségem.
A jelenet három dimenziós újraépítése során használt Mask R-CNN hatékonynak bizonyult a kulcspontok detektálásánál és a diszparitás térképpel megfelelő módon tudtam kombinálni. A PyQtGraph segítségével a kulcspontokat és a végtagokat könnyedén ki lehetett rajzolni, viszont a modul által megjelenített ablakot a fejlesztés során tárgyalt okok miatt nem tudtam a felhasználói felületen belül megjeleníteni.
A hasonló rendszerekkel összevetve a sztereó kamerák által elért mélységbecslés

nagyon alulmarad a mély tanulási módszerek által elérhető eredményektől, valamint az egyes paraméterek beállítása is egy nagyon időigényes folyamatnak bizonyult. Fontos kiemelni, hogy mindezek ellenére, rövid távolságokban ez a módszer is működőképes eredményt hozhat egy megfelelően felállított kamerarendszerrel.

9. TOVÁBBFEJLESZTÉSI LEHETŐSÉGEK

- Egy olyan komponens lefejlesztése, ami a rektifikáció jóságára ad egy számszerűsített értéket. Itt elképzelésem szerint jellemzőpontokat kellene detektáljunk és egymásra illesszünk, majd pedig le kéne ellenőrizni, hogy az illesztett párok hány százaléka rendelkezik azonos függőleges koordinátákkal.

- A számítást végző komponensek felhőben történő működtetésével áthidalható lenne a teljesítménybeli problémák egy része, valamint automatikusan tudnánk skálázni szükség szerint. Ennél a megközelítésnél IP kamerákat használnék, valamint ki kellene építeni egy kliens-szerver architektúrát.

- Különbségtérkép felhasználása, hogy valódi távolságot tudjunk számolni egy háromszögelés segítségével. Ennek a fejlesztését akkor lenne célszerű elkezdeni, amikor már a diszparitás térképre egy kimagaslóan jó eredményt kapunk.

- Mask R-CNN által kinyert maszkok felhasználása és ráillesztés pont felhő szerűen a térben szereplő csontvázakra.

- A jelenet három dimenziós újraépítése nem csak álló képekre, hanem videóra is.

- A meglévő komponensek gyorsítása addig, hogy egy valós időben működő megoldást kapjunk. Ebben az esetben a Mask R-CNN-t le kellene cserélnünk egy másik kulcspont detektáló modellre.

IRODALOMJEGYZÉK

[1] Hawk-Eye Innovations Ltd., Hawk-Eye, (https://www.hawkeyeinnovations.com/), utoljára megtekintve: 2021.05.06.

[2] ESPN, Two British scientists call into question Hawk-Eye's accuracy, (https://www.espn.com/sports/tennis/wimbledon08/news/story?id=3452293), publikálás dátuma: 2008.06.19., utoljára megtekintve: 2021.02.24.

[3] The Observer, Bundesliga approves Hawk-Eye goal-line technology for new season, (https://www.carlyleobserver.com/sports/bundesliga-approves-hawk-eye-goal-line-technology-for-new-season-1.1650074), publikálás dátuma: 2014.12.04. , utoljára megtekintve: 2021.02.24.

[4] Owen Gibson, The Guardian, Premier League clubs choose Hawk-Eye to provide new goalline technology, (https://www.theguardian.com/football/2013/apr/11/premier-league-hawkeye-goalline-technology), publikálás dátuma: 2013.04.11., utoljára megtekintve: 2021.02.25.

[5] ChyronHego Corporation, (https://chyronhego.com/), utoljára megtekintve: 2021.02.27.

[6] ChyronHego Corporation, TRACAB Technologies, (https://tracab.com/products/tracab-technologies/), utoljára megtekintve: 2021.02.27.

[7] Dennis Scimeca, Machine vision cameras power multi-object tracking system, (https://www.vision-systems.com/cameras-accessories/article/14178534/tracab-optical-sports-tracking-system-gen-5-uses-teledyne-dalsa-genie-nano-gige-cameras), publikálás dátuma: 2020.09.11., utoljára megtekintve: 2021.02.27.

[8] Sports Video Group Staff, ChyronHego Generates Latest Version of Tracking With TRACAB Gen5, (https://www.sportsvideo.org/2019/05/15/chyronhego-generates-latest-version-of-tracking-with-tracab-gen5/), publikálás dátuma: 2019.05.15., utoljára megtekintve: 2021.02.27.

[9] A diagram of a pinhole camera, (https://en.wikipedia.org/wiki/Pinhole_camera_model#/media/File:Pinhole-camera.svg), utoljára megtekintve: 2021.03.01.

[10] Zhengyou Zhang, IEEE, A flexible new technique for camera calibration, (https://ieeexplore.ieee.org/abstract/document/888718), publikálás dátuma: 2000. november, utoljára megtekintve: 2021.03.10.

[11] Vasista Ayyagari, Medium, Camera Calibration — Theory and Implementation, (https://medium.com/analytics-vidhya/camera-calibration-theory-and-implementation-b253dad449fb), publikálás dátuma: 2020.09.14., utoljára megtekintve: 2021.03.10.

[12] The MathWorks, Inc., What Is Camera Calibration?, (https://www.mathworks.com/help/vision/ug/camera-calibration.html), utoljára megtekintve: 2021.03.10.

[13] OpenCV, Medium, (https://opencv.org/), utoljára megtekintve: 2021.03.10.

[14] Satya Mallick, OpenCV (C++ vs Python) vs MATLAB for Computer Vision, (https://learnopencv.com/opencv-c-vs-python-vs-matlab-for-computer-vision/), publikálás dátuma: 2015.10.30., utoljára megtekintve: 2021.03.15.

[15] Camera Calibration and 3D Reconstruction, (https://docs.opencv.org/3.4/d9/d0c/group__calib3d.html), utoljára megtekintve: 2021.03.16.

[16] Heiko Hirschmuller, Stereo processing by semiglobal matching and mutual information, Pattern Analysis and Machine Intelligence, IEEE, publikálás éve: 2008, utoljára megtekintve: 2021.03.18.

[17] Rohith Gandhi, Medium, R-CNN, Fast R-CNN, Faster R-CNN, YOLO — Object Detection Algorithms, (https://towardsdatascience.com/r-cnn-fast-r-cnn-faster-r-cnn-yolo-object-detection-algorithms-36d53571365e), publikálás dátuma: 2018.07.09., utoljára megtekintve: 2021.03.21.

[18] FRITZ AI, FRITZ LABS INCORPORATED., Object Detection Guide, (https://www.fritz.ai/object-detection/), utoljára megtekintve: 2021.03.21.

[19] TensorFlow, Pose estimation, (https://www.tensorflow.org/lite/examples/pose_estimation/overview), publikálás dátuma: 2021.02.22., utoljára megtekintve: 2021.03.25.

[20] FRITZ AI, FRITZ LABS INCORPORATED., Pose Estimation Guide, (https://www.fritz.ai/pose-estimation/), utoljára megtekintve: 2021.03.25.

[21] Girshick, R.: Fast R-CNN. *arXiv*, 2015

[22] Girshick, R., Donahue, J., Darrell, T., Malik, J.: Rich feature hierarchies for accurate object detection and semantic segmentation. *arXiv*, 2014

[23] Ren, S., He, K., Girshick, R., Sun, J.: Faster R-CNN: Towards Real-Time Object Detection with Region Proposal Networks. *arXiv*, 2016

[24] He, K., Gkioxari, G., Dollár, P., Girshick, R.: Mask R-CNN. *arXiv*, 2018

[25] Python, Python Software Foundation, (https://www.python.org/), utoljára megtekintve: 2022.04.26.

[26] Anaconda Inc., Anaconda Distribution, (https://www.anaconda.com/), utoljára megtekintve: 2022.04.26.

[27] Microsoft, Visual Studio, (https://visualstudio.microsoft.com/), utoljára megtekintve: 2022.04.26.

[28] Microsoft, Visual Studio Code, (https://code.visualstudio.com/), utoljára megtekintve: 2022.04.26.

[29] PrismCore, Prism Library, (https://www.nuget.org/packages/Prism.Core/), utoljára megtekintve: 2022.04.26.

[30] MvvmLightLibs, MVVM Light Toolkit, (https://www.nuget.org/packages/MvvmLightLibs/), utoljára megtekintve: 2022.04.26.

[31] Autofac, Inversion of Control container, (https://www.nuget.org/packages/Autofac/), utoljára megtekintve: 2022.04.26.

[32] PyInstaller, (https://pyinstaller.org/), utoljára megtekintve: 2022.04.27.

[33] Calib.io, Camera Calibration, (https://calib.io/pages/camera-calibration-pattern-generator), utoljára megtekintve: 2022.04.30.

[34] Richard Hartley, Andrew Zisserman, Cambridge University Press, Cambridge, Multiple View Geometry in Computer Vision, publikálás éve: 2003, utoljára megtekintve: 2022.04.30.

[35] Matplotlib, Visualization with Python, (https://matplotlib.org/), utoljára megtekintve: 2022.05.01.

[36] PyQtGraph, Scientific Graphics and GUI Library for Python, (https://www.pyqtgraph.org/), utoljára megtekintve: 2022.05.01.

[37] PyQtGraph, Scientific Graphics and GUI Library for Python, (https://www.pyqtgraph.org/), utoljára megtekintve: 2022.05.01.

[38] OpenGL, Silicon Graphics, (https://www.opengl.org/), utoljára megtekintve: 2022.05.01.

[39] NUnit, (https://nunit.org/), utoljára megtekintve: 2022.05.02.

[40] Moq, (https://www.nuget.org/packages/Moq/), utoljára megtekintve: 2022.05.02.

ÁBRAJEGYZÉK

Táblázatok jegyzéke

MELLÉKLETEK

A program futtatásához szükséges eszközök:

- 64 bites Windows
- Legalább 8 GB RAM

Forráskódtól a program használatáig szükséges lépések:

- Python 3.6 telepítése
- .NET 6.0 keretrendszer telepítése
- Pip csomagkezelő telepítése
- Szükséges pip függőségek:
  - matplotlib
  - numpy
  - opencv-contrib-python
  - opencv-python
  - pyinstaller
  - pycocotools
  - tensorflow==1.13.1
  - keras==2.0.8
  - h5py==2.10.0
  - scipy==1.2.2
  - passlib==1.7.2
  - Pillow
  - cython
  - scikit-image
  - imgaug
  - pyqtgraph==0.11.0
- Fordítható állományok készítése **PyInstaller**-el, parancssorból:
  - pyinstaller –onefile –noconsole {Python szkript neve}.py
- A fordítható állomány átmozgatása, a szkripttel megegyező mappába.