



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Távközlési és Médiainformaticai Tanszék

Nézőponthelyreállítás több kameraképből

DIPLOMATERV

Készítette
Kriván Bálint

Konzulens
dr. Kovács Gábor

2015. május 20.

HALLGATÓI NYILATKOZAT

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

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

Budapest, 2015. július 31.

Kriván Bálint
hallgató

Kivonat

A gépi látás területén manapság népszerű kutatási téma a különböző pozícióba helyezett kamerák képei alapján történő háromdimenziós jelenet visszaállítás. A diplomaterv egy, a szakterület újabb eredményeire alapozó eljárást ír le, amely képes két rögzített kamera videojelfolyamát felhasználva mozgó objektumok háromdimenziós rekonstrukciójára a kamerák nézőpontja közötti tetszőleges irányból. A módszer hatékonyságát egy erre a célra készített szimulációs szoftverben végzett mérésekkel támasztja alá, továbbá lehetőséges továbbfejlesztési irányokat vázol.

Abstract

One of the popular topics in machine vision nowadays is the three dimensional scene reconstruction using cameras with different placement. This paper presents a method, which is able to reconstruct three dimensional objects based on video streams of two fixed cameras from any viewpoint between these cameras utilizing recent results in the field. It confirms the efficiency of the procedure by measurements performed in a simulation software created for this purpose, and provides directions for further improvement.

Tartalomjegyzék

Kivonat	2
Abstract	3
1. Bevezető	6
2. Elméleti bevezetés	8
2.1. Feladat értelmezése	8
2.2. Lineáris transzformációk	9
2.3. Homogén koordináták	9
2.4. Affin transzformációk	10
2.5. A háromdimenziós tér alapvető transzformációi	10
2.5.1. Eltolás	11
2.5.2. Forgalás	11
2.5.3. Skálázás	11
2.5.4. Nyírás	11
2.5.5. Transzformációk egymásutánja	12
2.6. Perspektivikus projekció – A lyukkamera-modell	12
2.7. Összefoglaló	15
3. Több kamerából álló kamera-rendszerek	16
3.1. Kamerák szinkronizációja	16
3.2. Epipoláris geometria	16
3.3. A háromdimenziós tér rekonstrukciója	18
3.3.1. Sztereó-látás, sztereó-kalibráció	18
3.3.2. Optikai folyam	20
3.3.3. Háromszögelés	24
3.4. Objektum-detektálás	26
3.4.1. Mozgó objektumok kiválasztása	26
3.4.2. Objektumok meghatározása a kamerák képein	27
3.5. Összefoglaló	27
4. Tervezés	28
4.1. Specifikáció kidolgozása	28

4.2. Keretrendszer	29
4.3. A tervezett rendszer működése	29
4.4. Összefoglaló	30
5. Megvalósítás	33
5.1. Alapok	33
5.2. Kalibráció	34
5.2.1. Kamerák pozíciójának meghatározása világkoordinátákban	35
5.3. Objektum detektálás	36
5.3.1. Előtér maszk meghatározása	36
5.3.2. Egyetlen objektum detektálása	39
5.3.3. Több objektum detektálása, párosítás a kamerák képein	40
5.4. Optikai folyam meghatározása	42
5.5. Háromszögelés	46
5.5.1. OpenCV-s függvényekkel	47
5.5.2. Iteratív lineáris legkisebb négyzetek (<i>Iterative-LS</i>)	48
5.6. Változtatható nézőpont	49
5.7. Teljes folyamat együtt, implementációk kiválasztása	50
5.8. Összefoglaló	51
6. Helyreállítás tesztelése	54
6.1. Első jelenet	54
6.2. Második jelenet	55
6.3. Harmadik jelenet	57
6.4. Összefoglaló	59
7. Gyorsítási lehetőségek	60
7.1. Első mérések	60
7.2. Párhuzamosítás, újra-kalkulált eredmények	61
7.3. Optikai folyam számolása GPU-n	63
7.4. Valós idejű helyreállíthatóság vizsgálata	64
7.5. Összefoglaló	65
8. Eredmények	67
8.1. Továbbfejlesztési lehetőségek	68
9. Összefoglalás	69
Köszönetnyilvánítás	70
Irodalomjegyzék	74
Függelék	75
F.1. Az alkalmazás elérhetősége	75

1. fejezet

Bevezető

Az informatika, ezen belül pedig a gépi látással foglalkozó terület, valamint az ehhez szükséges számítási kapacitás és célhardverek rohamos fejlődésével mind újabb, hatékonnyabb valamint pontosabb megoldások születtek és születnek a felmerült problémákra.

Míg egy-két évtizeddel ezelőtt a gépi látáshoz kapcsolódó kutatások jelentős részét főként a robotika, a katonság (pl. drónok), valamint az űrkutatás adta, manapság már a minden nap élet gyökeres részévé vált. Vegyük például a közlekedést: a középf-felső kategóriás autóknál már szériatartozéknak tekinthető az elülső és hátsó tolató radar. Ugyanígy a kereskedelmi forgalomban kapható robot-porszívók is rendelkeznek beépített kamera/radar-rendszerrel, amely a beltéri navigációt segíti. A napjainkban kapható játékkonzolokhoz is vásárolható kiegészítő kamera rendszer, mely a játékos mozgását és pozícióját figyeli, lényegében a játékos a saját testét használja vezérlőként. Szórakozást tekintve a manapság egyre nagyobb teret kapó quadcoptereket [1] is említeni kell, ezek is rendelkeznek kamerával (vagy rájuk szerelhető), és már folynak kutatások, amelyek ezek akár autonóm [2], akár tömeges [3] – rajban történő – repülését vizsgálja.

A dolgozat motivációját a következő feltevésekhez hasonló problémák adták:

- Egy izgalmas gólhelyzet során mit láthatott a kapus a kapuban állva?
- Egy bankrablási szituációban mit láthatott az elkövető, és mit a biztonsági őr?
- Egy vizsga során, a gyanús egyetemista láthatta-e az előtte ülő dolgozatát?
- Egy közlekedési balesetben mit láthatott a biciklis, és mit a buszsofőr?

Ezekre és ehhez hasonló kérdésekre részben választ nyújthat a kitűzött feladat megoldása, miszerint több kamerával megfigyelt térrészt egy választott nézőpontból rekonstruálunk, mivel a fenti esetekben nem oldható meg, hogy a választott személy nézőpontjába valódi kamerákat állítsunk.

A diplomamunka részletesen leírja azt az eljárást, amely két rögzített kamera videofolyamai alapján előállítja a mozgó objektumok rekonstrukcióját egy választott nézőpontból. Az egyszeri kalibrációt követően a videofolyamok képkockáit több lépésben dolgozza fel. Először a mozgó objektumokat azonosítja két fázisban. Elsőként kijelöli azon képrészleteket, melyek az előtérhez tartoznak, majd utána ezeket párosítja a kamerák képein jellegzetes pontok segítségével. Az így adódó képrészlet-párosítások adják az ugyanazon valódi objektumhoz tartozó képrészleteket. Ezt követően az optikai folyamok segítségével az objektumok minden kamera képén látszódó pontjai között sűrű pont-pont megfeleltetést számol. A kamerák helyzeteit felhasználva a pontpárokból háromszögeléssel meghatározza a pontok háromdimenziós koordinátáit, melyek alapján végül rekonstruálja a választott nézőpontból látható képet.

A 2. fejezetben a feladat értelmezését követően a megoldásához szükséges elméleti háttéről lesz szó, mely tartalmazza a háromdimenziós tér alapvető transzformációit, valamint a dolgozat során használt kameramodellt. A 3. fejezet a több kamerából álló kamerarendszereknél felmerülő problémákat tárgyalja, illetve, hogy ezekre milyen, a napjainkban is használt megoldások léteznek, kitérve a feladat során szükséges részproblémák megoldására is. A 4. fejezet bemutatja az előző fejezetek által leírt információk és algoritmusok alapján egy, a kitűzött feladatra megoldást adó rendszer vázát, megvalósítási tervét. Az 5. fejezetben a rendszer megvalósítáról, és az aközben hozott tervezői döntésekéről lesz szó. Végigvezet az egyes fázisoknál elkészült implementációkon, és végül bemutatja a választott eljárásokat. A 6. fejezet az elkészült alkalmazás segítségével három különböző helyzetben telepített kamerákkal felvett jeleneteket rekonstruál. A 7. fejezet a gyorsítási lehetőségeket vizsgálja meg, kitér a párhuzamosítás lehetőségeire, valamint a GPU-n történő futtatásra, és az így elérhető sebességnövekedésre. Végül megvizsgálja a valós idejű helyreállíthatóság korlátait, melyeket konkrét számokkal támaszt alá. Végezetül a 8. fejezet a gyűjtött eredményeket foglalja össze, és értékeli az elkészült alkalmazást.

2. fejezet

Elméleti bevezetés

Ezt a fejezetet a feladat értelmezésével kezdem, majd az alkalmazott matematikai hátteret mutatom be. Rövid lineáris algebrai bevezető után a háromdimenziós tér alapvető transzformációit tárgyalom. A fejezetet a lyukkamera-modell leírásával zárom.

2.1. Feladat értelmezése

A feladatom egy legalább két fix telepítésű kamerából álló kamerarendszer által megfigyelt térrészben egy tetszőleges pontban és irányban látható kép valós idejű helyreállíthatóságának vizsgálata, a probléma megoldhatósági korlátainak meghatározása, és egy megvalósítási terv készítése volt.

A dolgozat során két eltérő módszert mutatok be: a sztereó-kamerák valamint az optikai folyamok módszerét. Ezek közül az utóbbit valósítom meg, tesztelem és végül értékelém. A probléma megoldása során a könnyebb objektum-detektálás érdekében felfesztem, hogy a rekonstruálálandó objektumok mozognak a kamerák képein. Ez a kényszer igazodik a feladat kitűzésem második pontjához, és így ennek a tervezett megoldásom eleget tud tenni. A látott kép helyreállítása alatt az adott nézőpontból látható mozgó objektumok felismerését és azok kontúrjainak kirajzolását értem.

A valós idejű helyreállítóság vizsgálatát és az eredmények dokumentációját a következőknek megfelelően végzem el. Először egy darab mozgó objektumot rekonstruálok, közben a nézőpontot beviteli vezérlők segítségével a két kamera közötti vonalon szabadon változtatom. Ezt követően két darab, lényegesen eltérő textúrával rendelkező mozgó objektumot helyezek a térrészbe, és vetem alá a helyreállításnak. Ezt két különböző kameratelepítéssel is megvizsgálom, egyik esetben az optikai tengelyek párhuzamosak, míg másik esetben hegyes szöget zárnak be. Végül egy szoba két sarkába helyezem a két kamerát és a szoba közepén mozgó objektumot állítok helyre. Rekonstrukció során a feldolgozás sebességét az egységes idő alatt rekonstruált és kirajzolt képkockák számában

határozom meg (FPS - *frames per second*). Az eredmény minőségét, azaz a helyreállítás helyességét – referencia rekonstrukció hiányában – szubjektíven fogom értékelni.

A fejezet hátralévő részében a matematikai hátteret és a lyukkamera-modellt mutatom be.

2.2. Lineáris transzformációk

Legyen $B = \{\mathbf{b}_1, \mathbf{b}_2, \dots, \mathbf{b}_n\}$ bázis \mathbb{R}^n -ben. Ekkor egy teszőleges $\mathbf{v} \in \mathbb{R}^n$ egyértelműen felírható a bázisvektorok lineáris kombinációjaként:

$$\mathbf{v} = \lambda_1 \mathbf{b}_1 + \lambda_2 \mathbf{b}_2 + \dots + \lambda_n \mathbf{b}_n \quad \text{ahol } \lambda_i \in \mathbb{R}$$

Ekkor \mathbf{v} koordinátái a B bázisban:

$$[\mathbf{v}]_B = \begin{pmatrix} \lambda_1 \\ \lambda_2 \\ \vdots \\ \lambda_n \end{pmatrix}$$

$\mathcal{F} : \mathbb{R}^n \rightarrow \mathbb{R}^n$ lineáris transzformáció, amennyiben tetszőleges $\mathbf{v}, \mathbf{w} \in \mathbb{R}^n$ és $\lambda \in \mathbb{R}$ -re teljesül az alábbi:

$$\mathcal{F}(\mathbf{v} + \mathbf{w}) = \mathcal{F}(\mathbf{v}) + \mathcal{F}(\mathbf{w}) \quad \text{és} \quad \mathcal{F}(\lambda \mathbf{v}) = \lambda \mathcal{F}(\mathbf{v})$$

Az \mathbf{M} mátrixot az \mathcal{F} lineáris transzformáció mátrixának hívjuk B -ben, ha minden $\mathbf{v} \in \mathbb{R}^n$ -re teljesül, hogy:

$$[\mathcal{F}(\mathbf{v})]_B = \mathbf{M} \cdot [\mathbf{v}]_B$$

2.3. Homogén koordináták

Az euklideszi sík, illetve tér projektív lezárása ezek ideális pontokkal való bővítése: minden egymással párhuzamos egyenest ugyanazon ideális ponttal bővítünk. Ezek síkgeometria esetén egy ideális egyenest, térben pedig egy ideális síkot határoznak meg. Algebrai módszerekkel történő kezelésükhez Descartes-féle koordinátákról át kell térünk homogén koordinátáakra.

A homogén koordináták csak egy konstans szorzó erejéig vannak meghatározva, vagyis:

$$\begin{bmatrix} u_1 \\ u_2 \\ u_3 \end{bmatrix} = \begin{bmatrix} \lambda u_1 \\ \lambda u_2 \\ \lambda u_3 \end{bmatrix} \quad \lambda \neq 0$$

Egy euklideszi tér $(x, y, z)^T$ pontjának a $[x, y, z, 1]^T$ homogén koordinátákat és ezzel arányos számnégyeseket feleltetjük meg. Fordítva, ha $[x_1, x_2, x_3, x_4]^T$ homogén koordináták, és $x_4 \neq 0$, akkor ez egy közönséges pont, melynek Descartes-koordinátái: $\left(\frac{x_1}{x_4}, \frac{x_2}{x_4}, \frac{x_3}{x_4}\right)^T$. Amennyiben $x_4 = 0$, akkor a kérdéses pont egy ideális pont (és az ehhez tartozó párhuzamos egyenesek irányvektora $[x_1, x_2, x_3]^T$).

A síkban egy egyenes $ax + by + c = 0$ alakban írható fel, melyet az $[a, b, c]^T$ homogén koordinátákkal reprezentálhatunk. Egy $(x, y)^T$ pont rajta van ezen az egyenesen, ha kielégíti az egyenes egyenletét. Tekintsünk a P pontot $\mathbf{p} = [x, y, 1]^T$ homogén koordinátákkal és az e egyenest az $\mathbf{e} = [a, b, c]^T$ vektorral, ekkor P rajta van e -n, ha:

$$\mathbf{p}^T \mathbf{e} = 0$$

2.4. Affin transzformációk

Az σ síkon vett affin transzformációt egy olyan $\phi : \sigma \rightarrow \sigma$ bijekciót értünk, amely minden $e \in \sigma$ egyenest $e' \in \sigma$ egyenesbe képezi le. A lineáris algebra eszközeinek felhasználásával, egy tetszőleges $P(x, y)$ pont $P'(x', y')$ affin képe felírható az alábbi mátrix-egyenlettel:

$$\begin{pmatrix} x' \\ y' \end{pmatrix} = \begin{pmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{pmatrix} \cdot \begin{pmatrix} x \\ y \end{pmatrix} + \begin{pmatrix} b_1 \\ b_2 \end{pmatrix} \quad \text{ahol} \quad \begin{vmatrix} a_{11} & a_{12} \\ a_{21} & a_{22} \end{vmatrix} \neq 0$$

Vagyis egy affin transzformáció leírható egy lineáris transzformáció és egy eltolás egymásutánjával. Homogenizált alakban – azaz a pontokat homogén koordinátarendszerben felírva – az alábbi formát ölti:

$$\begin{bmatrix} x' \\ y' \\ 1 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & b_1 \\ a_{21} & a_{22} & b_2 \\ 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} x \\ y \\ 1 \end{bmatrix}$$

2.5. A háromdimenziós tér alapvető transzformációi

A következőkben négy alapvető affin transzformáció kerül bemutatásra a térben a homogén koordináták segítségével, ezek sorrendben: eltolás, forgatás, skálázás és nyírás.

2.5.1. Eltolás

Legyen $\mathbf{v} = (v_x, v_y, v_z)^T, \mathbf{t} = (t_x, t_y, t_z)^T \in \mathbb{R}^3$, ekkor \mathbf{v} vektor \mathbf{t} eltoltja:

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & t_x \\ 0 & 1 & 0 & t_y \\ 0 & 0 & 1 & t_z \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix}$$

2.5.2. Forgatás

Legyen $\mathbf{v} = (v_x, v_y, v_z)^T \in \mathbb{R}^3$ és $\theta \in \mathbb{R}$, ekkor \mathbf{v} vektor y tengely körüli θ szöggel való (jobb-kéz szabályt használva) elforgatottja:

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \\ 1 \end{bmatrix} = \begin{bmatrix} \cos(\theta) & 0 & \sin(\theta) & 0 \\ 0 & 1 & 0 & 0 \\ -\sin(\theta) & 0 & \cos(\theta) & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix}$$

2.5.3. Skálázás

Legyen $\mathbf{v} = (v_x, v_y, v_z)^T, \mathbf{s} = (s_x, s_y, s_z)^T \in \mathbb{R}^3$, ekkor \mathbf{v} vektor origóból történő skálázása \mathbf{s} -sel:

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \\ 1 \end{bmatrix} = \begin{bmatrix} s_x & 0 & 0 & 0 \\ 0 & s_y & 0 & 0 \\ 0 & 0 & s_z & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix}$$

2.5.4. Nyírás

A térbeli nyírás a tér P pontjainak egy fix síkkal történő párhuzamos csúsztatása. Legyen $\mathbf{v} = (v_x, v_y, v_z)^T \in \mathbb{R}^3$ és $\lambda \in \mathbb{R}$, ekkor a \mathbf{v} vektor x tengely irányában az $x - y$ sík mentén vett nyírása:

$$\begin{bmatrix} v'_x \\ v'_y \\ v'_z \\ 1 \end{bmatrix} = \begin{bmatrix} 1 & \lambda & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \cdot \begin{bmatrix} v_x \\ v_y \\ v_z \\ 1 \end{bmatrix} = \begin{bmatrix} v_x + \lambda \cdot v_y \\ v_y \\ v_z \\ 1 \end{bmatrix}$$

2.5.5. Transzformációk egymásutánja

Ha egy $\mathcal{A}_1, \mathcal{A}_2, \dots, \mathcal{A}_n$ transzformáció sorozatnak megfelelő transzformációs mátrixok rendre $\mathbf{T}_1, \mathbf{T}_2, \dots, \mathbf{T}_n$, akkor \mathbf{v} transzformáltja:

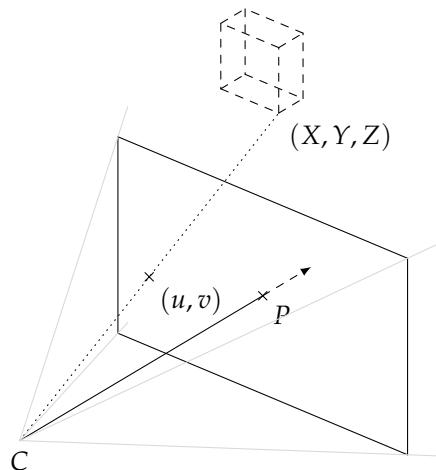
$$\mathbf{v}' = \mathcal{A}_n(\mathcal{A}_{n-1}(\dots \mathcal{A}_2(\mathcal{A}_1(\mathbf{v}))) \dots) = \mathbf{T}_n \cdot \mathbf{T}_{n-1} \cdot \dots \cdot \mathbf{T}_2 \cdot \mathbf{T}_1 \cdot \mathbf{v}$$

Így az eredő transzformációt lényegében úgy kapjuk, hogy a transzformációs mátrixok szorzatával, azaz az „eredő transzformációs mátrixszal” szorozzuk be a vektort:

$$\mathbf{v}' = \mathbf{T} \cdot \mathbf{v} \quad \text{ahol } \mathbf{T} = \prod_{i=1}^n \mathbf{T}_i$$

2.6. Perspektivikus projekció – A lyukkamera-modell

A kamerák a valós világot képezik le egy adott nézőpontból. A lyukkamera-modell alkalmazása során ezt a leképezést egy perspektivikus projekciónak tekintjük [4, 2.2. fejezet] (lásd a 2.1. ábrát). A projekció középpontját *optikai középpontnak* (C) vagy *kamera középpontnak*, az egyenest, ami merőleges a képsíkra, és átmegy az optikai középponton *optikai tengelynek*, magát a metszéspontot (P) pedig *főpontnak* nevezik.



2.1. ábra. Lyukkamera-modell, (X, Y, Z) pont képe a képsíkon (u, v)

Tegyük fel, hogy az optikai tengely párhuzamos a z tengellyel, és az optikai középpont a 3D-s koordinátarendszer origójában van. Ekkor az alábbi egyenlettel írható fel a projekció:

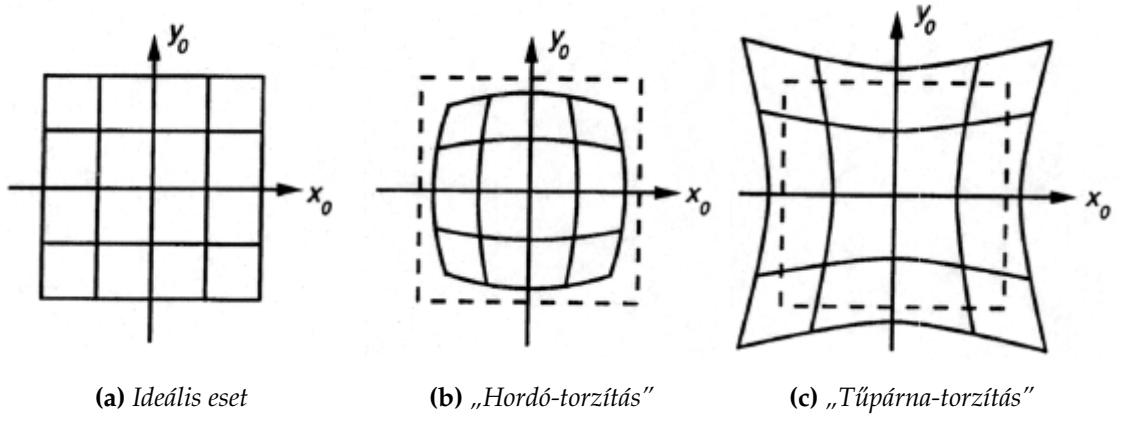
$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & 0 & 0 \\ 0 & f & 0 & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

ahol f a fókusztávolság (C és a képsík távolsága), $s = Z$ pedig a skálázási tényező.

A manapság használt képalkotó rendszerek a képsík origójának nem a főpontot, hanem a kép balfelső sarkát tekintik, így a fenti egyenlet a következő alakot ölti, amennyiben a főpont koordinátája (o_x, o_y) :

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & o_x & 0 \\ 0 & f & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

Valódi kamerák használata esetén számolni kell azzal, hogy a lencséknek van radiális (lásd a 2.2. ábra) és tangenciális torzításuk (ami abból adódik, hogy a lencse és a képalkotó sík nem párhuzamos). Mivel ezek csak a konkrét kamerától függnek, ezért mértékük kalibrációval meghatározható és hatásuk kiküszöbölhető.



2.2. ábra. Lencsék radiális torzítása [5]

A gyakorlatban gyakran használt modell, hogy a radiális torzítás esetében a torzított és a javított pontok között egy polinomfüggvény teremt kapcsolatot:

$$\begin{pmatrix} x_{\text{jav}} \\ y_{\text{jav}} \end{pmatrix} = (1 + k_1 r^2 + k_2 r^4 + k_3 r^6) \begin{pmatrix} x \\ y \end{pmatrix}$$

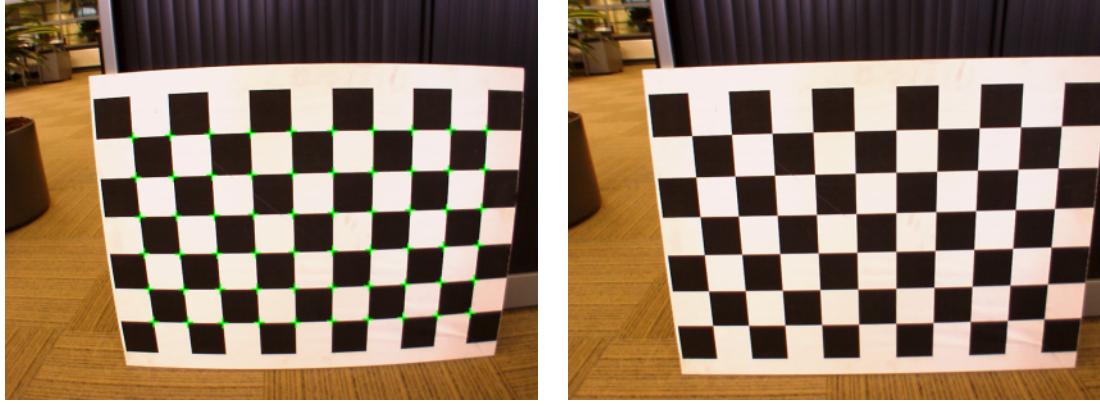
ahol (x, y) volt az eredeti kép egy pixelének koordinátája, $r^2 = x^2 + y^2$ és $(x_{\text{jav}}, y_{\text{jav}})$ pedig az új koordináta a korrigált képen. Tangenciális torzítás pedig a következő egyenletek segítségével javítható:

$$x_{\text{jav}} = x + (2p_1 xy + p_2(r^2 + 2x^2))$$

$$y_{\text{jav}} = y + (p_1(r^2 + 2y^2) + 2p_2xy)$$

Ekkor k_1, k_2 és k_3 a radiális, p_1 és p_2 pedig a tangenciális torzítás együtthatói.

Kalibrálás során ezeket az együtthatókat határozzuk meg úgy, hogy az egyenesek képeinek görbületét, mint költség-függvényt minimalizáljuk. A gyakorlatban ez úgy törtenik, hogy egy előre ismert objektumot – például egy sakktáblát – detektálunk a képeken (lásd a 2.3. ábra), és a sarokpontokat összekötő egyeneseket vizsgáljuk.



(a) Kamera képe, sarokpontokkal

(b) Javított kép

2.3. ábra. Radiális torzítás meghatározása [4]

Az előzőekben a kamera *belső paramétereit* tekintettük át, melyeket a kamera lencséi és fókusztávolsága határoznak meg. Azonban szükséges beszélnünk a *külső paramétereiről* is, amely a kamera pozícióját és irányát mutatja a világ-koordinátához képest.

A gyakorlatban a kamera helyét és irányát egy $\mathbf{c} = (c_1, c_2, c_3)^T$ vektorral és egy \mathbf{R} forgatási mátrixszal adjuk meg. Így ahhoz, hogy egy 3D-s pont képét megkapjuk, először a kamerát el kell tolni a világkoordináta-rendszer origójába, majd forgatni, vagyis:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & o_x & 0 \\ 0 & f & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} r_{11} & r_{12} & r_{13} & 0 \\ r_{21} & r_{22} & r_{23} & 0 \\ r_{31} & r_{32} & r_{33} & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 & 0 & -c_1 \\ 0 & 1 & 0 & -c_2 \\ 0 & 0 & 1 & -c_3 \\ 0 & 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

A két transzformáció összevonása után:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \begin{bmatrix} f & 0 & o_x & 0 \\ 0 & f & o_y & 0 \\ 0 & 0 & 1 & 0 \end{bmatrix} \begin{bmatrix} \mathbf{R} & -\mathbf{R}\mathbf{c} \\ \mathbf{0} & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

$\mathbf{t} = -\mathbf{R}\mathbf{c}$ helyettesítéssel, valamint a fókusztávolságot az effektív pixelméretekkel beszorozva kapjuk [6]-ban is található egyenletet:

$$s \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix}}_{\mathbf{A}} \underbrace{\begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix}}_{\mathbf{R}} \begin{bmatrix} X \\ Y \\ Z \\ 1 \end{bmatrix}$$

ahol:

- s a homogén skálázási tényező
- \mathbf{A} a kamera-mátrix (*belő paraméterek mátrixa*)
- (o_x, o_y) a főpont képsíkbeli koordinátája, amely általában a kép közepe
- (f_x, f_y) a fókusztávolságok pixelekben kifejezve
- \mathbf{R} pedig a forgatási mátrix és $\mathbf{t} = (t_1, t_2, t_3)^T$ az eltolás-vektor – együttesen $(\mathbf{R} | \mathbf{t})$ a forgatás-eltolás mátrix, mely a *külső paraméterek mátrixa*; amely megadja a világ-koordinátarendszer és a kamera-koordinátarendszer közötti kapcsolatot.

A kamera külső paramétereit, a belső paraméterekhez hasonlóan kalibrálással kapjuk meg. Ehhez egy ismert objektumot használunk fel, pl. egy sakktáblát: a sarokpontokat a világ-koordinátarendszerben koordinátázzuk például úgy, hogy azok az x - y síkba esnek, 3. koordinátájukat pedig fixen zérusnak választjuk. Ezeket a világbeli koordinátákat a képpontoknak megfeleltetve meghatározható \mathbf{R} és \mathbf{t} [6], feltéve, hogy a kamera képét már az előbbiek alapján rektifikáltuk.

2.7. Összefoglaló

Ebben a fejezetben a dolgozat megértéséhez szükséges és a feladat megoldása során is felhasznált matematikai hátteret mutattam be. Kitértem a háromdimenziós tér alapvető transzformációira, illetve az alkalmazott lyukkamera-modellre, különös figyelmet fordítva a kamerák képein megfigyelhető torzításokra, és azok kalibrációval történő kiküszöbölésére.

3. fejezet

Több kamerából álló kamera-rendszerek

A kitűzött feladat szempontjából két fontos részproblémát kell több kamerából álló rendszerek esetén megoldanom. Az első a kamerák szinkronizációjának problémája, ami akkor merül fel, ha a kamerák nem egy közös feldolgozó egységhez kapcsolódnak, hanem például a kamerák képeit egy videofolyamként kapjuk. A második pedig, hogy a kamerák képei alapján a háromdimenziós teret, vagy annak egy részét a lehető legjobban rekonstruáljam. Ez a fejezet ezen két problémát járja körbe, valamint a mozgó objektumok detektálását mutatja be.

3.1. Kamerák szinkronizációja

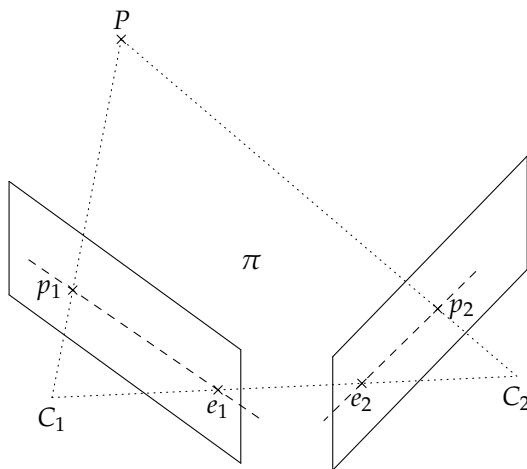
Amennyiben a videofolyamokba időpecséteket kódolunk, a kamerák szinkronizációja könnyen megoldható. A képsorokat a kódolt információ alapján egymáshoz tudjuk időzíteni, feltéve, hogy a kamerák óráit már előtte szinkronizáltuk.

Amennyiben ilyen adat nem áll rendelkezésre a videofolyamban, egyéb módszert kell keresnünk. Megoldás lehet, ha a kamerák képsorozatain egy közös trigger-eseményt keresünk. Egy korábbi munkám során [7] egy lézerpontot kerestem mindenek képen, és a felvillanás időpontját tekintettem referenciának. Ehhez nagyon hasonló a filmiparban használt *clapperboard*, aminek összeüttését használják szinkronizációra, de akár egy egyszerű taps is megfelelhet, amennyiben a kamerák hangot is rögzítenek.

3.2. Epipoláris geometria

Az epipoláris geometria adja meg a kapcsolatot két kamera képe között [8, 9. fejezet]. Tekintsük a 3.1. ábrát. Legyenek C_1 és C_2 pontok a kamerák középpontjai, és P egy pont

a térben, mely az első kamera I_1 képén p_1 -be, a második kamera I_2 képén p_2 -be képződik le. p_1 , p_2 és P egy síkot határoz meg, jelöljük ezt π -vel. Ezen a síkon vannak rajta a kamerák középpontjaiból a képpontokon áthaladó félegyenesei is melyek P -ben metszik egymást. π -t a P -hez tartozó *epipoláris síknak*, a két kameraközéppontot összekötő egyenest *bázis egyenesnek*, ennek metszéspontjait a képsíkokkal (e_1 és e_2) *epipólusoknak* nevezik. Gondoljuk meg, hogy ha p_1 -et ismerjük és p_2 -t szeretnénk meghatározni, akkor azt az epipoláris sík és az I_2 képsík metszésvonalán kell keresnünk, ezt a p_1 -hez tartozó *epipoláris egyenesnek* hívjuk. Így egy képpont pájját a másik képen nem az egész síkon, hanem csak egy egyenesen kell keresnünk.



3.1. ábra. Epipoláris geometria szemléltetése

Az \mathbf{F} fundamentális mátrix az epipoláris geometria algebrai megfeleltetése, mely az előbb említett képpontból epipoláris egyenesre történő leképezést adja meg. Legyen \mathbf{x} egy X pont képe az egyik képsíkon, \mathbf{l}' pedig a hozzá tartozó epipoláris egyenes a másik képsíkon, ekkor tehát

$$\mathbf{l}' = \mathbf{F}\mathbf{x}.$$

Amennyiben \mathbf{x}' -vel jelöljük X képét a másik képsíkon, akkor $\mathbf{x}'^T \mathbf{l}' = 0$, hiszen \mathbf{x}' rajta van \mathbf{l}' -n. Felhasználva az előző egyenletet kapjuk tehát, hogy egy X pont két képére

$$\mathbf{x}'^T \mathbf{F}\mathbf{x} = 0.$$

Ezen mátrixegyenletből kiindulva, egymásnak megfelelő képpontpárokból kiszámolhatjuk \mathbf{F} -et, 8 pontpárból egy skála erejéig, 8-nál több esetén pedig legkisebb négyzetek módszerével (7 esetén pedig létezik nem-lineáris megoldás) [8, 10.1 alfejezet].

3.3. A háromdimenziós tér rekonstrukciója

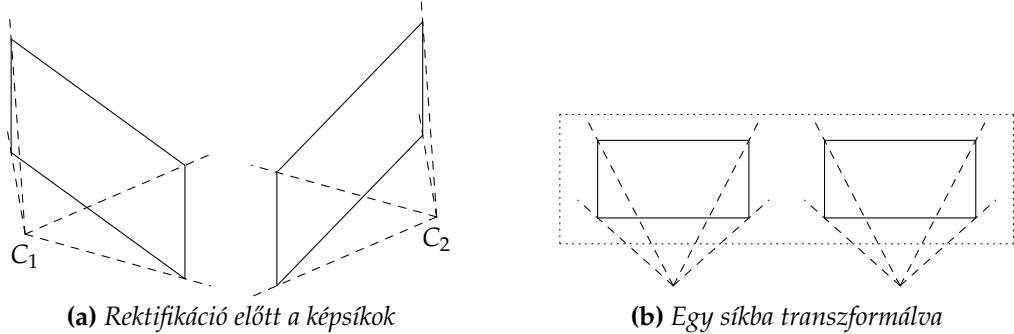
A dolgozat során két, alapvetően eltérő módszerre térek ki. Az első a sztereó-látás elvén alapszik, azaz a kamerákat párosával használva, azokat sztereó-kalibrálva alkotunk mélysékgépet, mely segítségével a képpontokat mélységinformációval ruházzuk fel. A második eljárás az optikai folyamok segítségével a kamerák képein meghatározza az egymásnak megfelelő pontokat, és ezekből a párosításokból háromszögelés után kiszámolja a pontok világbeli koordinátáját.

3.3.1. Sztereó-látás, sztereó-kalibráció

Sztereó-kalibráció [9] során egy már ismert háromdimenziós objektumot használunk, és az alapján, hogy az objektum ismert pontjait a két kamera képén hol látjuk, kiszámoljuk a kamerák belső és külső paramétereit (lásd előző fejezet), valamint a kamerák egymáshoz való viszonyát (eltolás, forgatás).

Amennyiben a két kamera képsíkja egybeesik, akkor a bázisegyenek a képsíkokkal a végzetlenben metszik egymást, ebből adódóan az epipoláris egyenek párhuzamosak. Ha a két kamera közötti transzformáció csak tisztán egy x tengellyel párhuzamos eltolás (standard sztereókamerák), akkor az epipoláris egyenek az x tengellyel is párhuzamosak, tehát a pontpárok keresése pusztán a képkockák egy adott sorára korlátozódik.

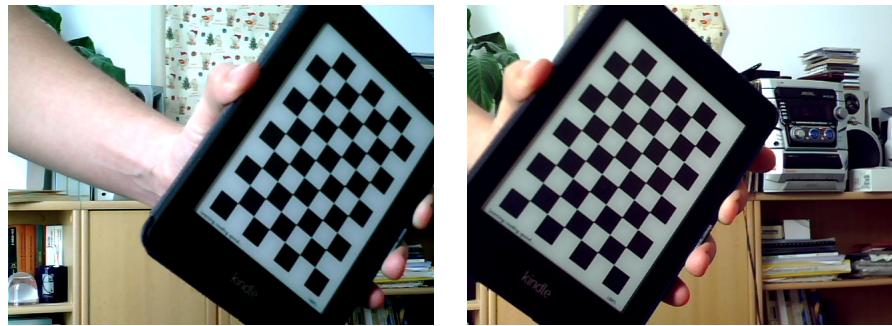
Mivel a gyakorlatban nehéz pontosan úgy elhelyezni a kamerákat, hogy képsíkjuk egybeessen és csak vízszintes irányú eltolás legyen köztük, ezért a kamerák képeit célszerű *rektifikálni*, hogy a pontok párait továbbra is csak egy-egy képsorban legyen szükséges keresni. Ekkor a kamerák képeit úgy transzformáljuk, hogy a képsíkok egy képzeletbeli közös síkba essenek, lásd a 3.2. ábrát.



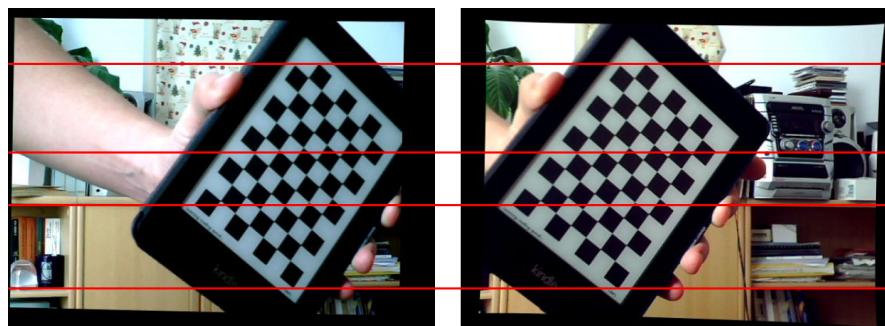
3.2. ábra. Rektifikáció szemléltetése

A 3.3. és a 3.4. ábrán látható egy sakktábla-minta a kamerák képein a rektifikáció [9] előtt és után. A piros segédegyenek segítségével az is jó látható, hogy a rektifikáció után az egymásnak megfelelő rácspontok valóban egy vízszintes egyenesre esnek.

A rektifikált képpárokra pedig már léteznek algoritmusok [10, 11], melyek megtalálják a képeken az egymásnak megfelelő pontokat, és az elmozdulásuk alapján megadják a

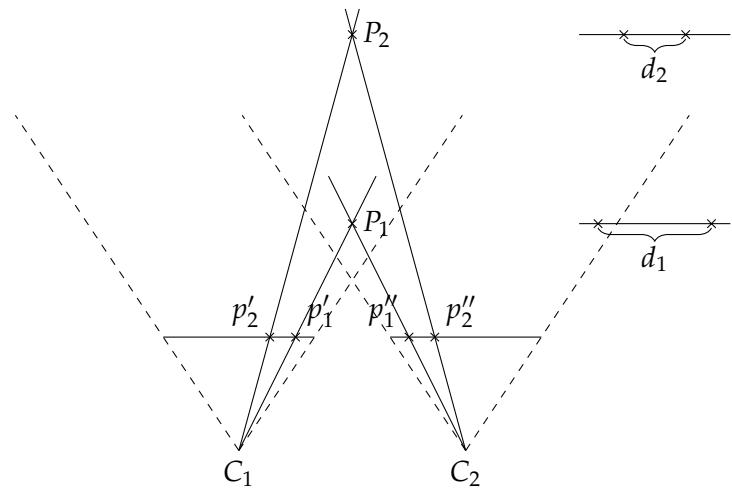


3.3. ábra. A kalibrációhoz felhasznált egyik képpár



3.4. ábra. Rektifikáció után a két kamera képe, pirossal jelölve néhány epipoláris egyenes

pontok mélységeit. A 3.5. ábra segítségével könnyű látni, hogy a nagy elmozdulást mutató pontok közelebb, a kisebbek távolabb találhatóak a kamerától.



3.5. ábra. Elmozdulás és mélység kapcsolata; a kameráktól távolabbi P_2 pont képeinek d_2 távolsága kisebb, mint a közelebbi P_1 pont képeinek d_1 távolsága

Fontos megemlíteni, hogy gyakorlatban a kameráknak már eleve nagyon közel kell lenniük az ideális elhelyezkedéshez, azaz egymás mellett (vagy egymás felett, attól fügően, hogy horizontális, vagy vertikális sztereó-kamerákat használunk) legyenek, úgy hogy képsíkjuk közel egybeessenek. Így érhető el az, hogy a rektifikáció sikeres legyen, valamint, hogy utána a kép legnagyobb része „használható” legyen, lásd a 3.4. ábrán a képszéleken lévő fekete részeket.

3.3.2. Optikai folyam

Az előzőekben megismert eljárás azt feltételezi, hogy a kamerák párosával sztereó-kalibráltak, és ezt kihasználva határozzuk meg a mélysékgépet a kalibráció során alkalmazott segéd-objektum alapján.

A most ismertetésre kerülő eljárás, az *optikai folyamokat* [12] hívja segítségül. Vagyunk egymás utáni képeket egy mozgó objektumról. Az objektum mindegyik pontja egy háromdimenziós $\mathbf{P}(t)$ útvonalon mozog, mely a kamerák képén egy $\mathbf{p}(t) = (x(t), y(t))$ kétdimenziós útvonalnak felelhető meg. minden pontra nézve az elmozdulás $d\mathbf{p}(t)/dt$ irányát egy vektormezőt kapunk, ezt hívjuk optikai folyamnak.

Jelöljük $I(x, y, t)$ -vel a kép (x, y) pontjának intenzitását a t időpillanatban. Feltehető, hogy két egymás utáni képkockán ugyanazon pont képpontjainak intenzitása konstans (világosság állandóság, az angol *brightness constancy* kifejezésből), vagyis

$$I(x, y, t) = I(x + dx, y + dy, t + dt),$$

ahol az (x, y) képpont dt idő alatt (dx, dy) -nal mozdult el. A jobb oldal jól közelíthető az első-rendű Taylor-sor kifejtésével:

$$I(x + dx, y + dy, t + dt) \approx I(x, y, t) + \frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt.$$

Felhasználva a világosság állandóság kényszert kapjuk, hogy

$$\frac{\partial I}{\partial x}dx + \frac{\partial I}{\partial y}dy + \frac{\partial I}{\partial t}dt = 0.$$

Mindkét oldalt dt -vel osztva:

$$\frac{\partial I}{\partial x} \frac{dx}{dt} + \frac{\partial I}{\partial y} \frac{dy}{dt} + \frac{\partial I}{\partial t} = 0$$

Végül bevezetve a $\nabla \mathbf{I} = \left(\frac{\partial I}{\partial x}, \frac{\partial I}{\partial y} \right)$ és $\mathbf{V} = \left(\frac{dx}{dt}, \frac{dy}{dt} \right)$ jelölésekét, az alábbi egyszerű alakot kapjuk:

$$\nabla \mathbf{I} \cdot \mathbf{V} = -\frac{\partial I}{\partial t}.$$

Ezt nevezzük az optikai folyam feltételi egyenletének [13]. Mivel $\mathbf{V} \left(\frac{dx}{dt}, \frac{dy}{dt} \right)$ nem ismert, ezért egy egyenletben két ismeretlenünk van, így az optikai folyam általános esetben nem oldható meg.

Lucas-Kanade módszer [14]

Ez a módszer ritka vektor-mezőt generál a jellegzetes pontok elmozdulására. Lényege, hogy a vizsgált pontok és azok környezetében az elmozdulást azonosnak tekinti [15], vagyis:

$$\nabla \mathbf{I}_{\mathbf{q}_1} \cdot \mathbf{V} = -\frac{\partial I}{\partial t}(\mathbf{q}_1, t)$$

$$\nabla \mathbf{I}_{\mathbf{q}_2} \cdot \mathbf{V} = -\frac{\partial I}{\partial t}(\mathbf{q}_2, t)$$

⋮

$$\nabla \mathbf{I}_{\mathbf{q}_n} \cdot \mathbf{V} = -\frac{\partial I}{\partial t}(\mathbf{q}_n, t),$$

ahol $\mathbf{q}_1, \mathbf{q}_2, \dots, \mathbf{q}_n$ a vizsgált pont környezetében lévő pontok és $\nabla \mathbf{I}_{\mathbf{q}_i} = \left(\frac{\partial I}{\partial x}(\mathbf{q}_i), \frac{\partial I}{\partial y}(\mathbf{q}_i) \right)$. Ez így viszont már egy túlhatározott egyenletrendszer, melyhez közelítő megoldás a legkisebb-négyzetek módszerével kereshető [14, 15]. A gyakorlatban a követendő pontokra a Shi-Tomasi módszer [16] által adott sarokpontokat szokták használni (lásd a 3.6. ábra).



3.6. ábra. Minta a sarokpontok követésére [17]

Gunner Farnebäck módszer [18]

A módszer lényege, hogy a kép pixeleinek intenzitását egy polinommal közelíti:

$$f(\mathbf{x}) \sim \mathbf{x}^T \mathbf{A} \mathbf{x} + \mathbf{b}^T \mathbf{x} + c$$

ahol \mathbf{A} egy szimmetrikus mátrix, \mathbf{b} vektor és c skalár.

Ezeket az együtthatókat a legkisebb-négyzetek módszerével közelítően meghatározhatjuk. Tegyük fel, hogy minden kép összes pixelének intenzitása egyetlen polinommal felírható, és a két képkockán az egész polinom ideálisan \mathbf{d} -vel mozdul el, vagyis

$$f_1(\mathbf{x} - \mathbf{d}) = f_2(\mathbf{x}).$$

A fenti képlet jelenti a világosság állandóság kényszer teljesülését. Bal oldalt kifejtve, jobb oldalt pedig átírva a polinom-alakba [18] kapjuk, hogy

$$\mathbf{x}^T \mathbf{A}_1 \mathbf{x} + (\mathbf{b}_1 - 2\mathbf{A}_1 \mathbf{d})^T \mathbf{x} + \mathbf{d}^T \mathbf{A}_1 \mathbf{d} - \mathbf{b}_1^T + c_1 = \mathbf{x}^T \mathbf{A}_2 \mathbf{x} + \mathbf{b}_2^T \mathbf{x} + c_2.$$

Az együtthatókat egyenlővé téve, kapjuk a következő egyenleteket:

$$\begin{aligned} \mathbf{A}_2 &= \mathbf{A}_1 \\ \mathbf{b}_2 &= \mathbf{b}_1 - 2\mathbf{A}_1 \mathbf{d} \\ c_2 &= \mathbf{d}^T \mathbf{A}_1 \mathbf{d} - \mathbf{b}_1^T + c_1 \end{aligned}$$

Jól látható, hogy \mathbf{d} meghatározhatóságának szükséges feltétele, hogy \mathbf{A}_1 determinánsa nem nulla. Általános esetben természetesen nem élhetünk az előbbi feltételekkel, miszerint minden kép leírható egyetlen polinommal és ideálisan minden pont \mathbf{d} -vel mozdult el. Ezért lokális polinomokkal dolgozunk, melyek a pontok egy környezetében érvényesek, vagyis $\mathbf{A}_1(\mathbf{x}), \mathbf{b}_1(\mathbf{x}), c_1(\mathbf{x})$ és $\mathbf{A}_2(\mathbf{x}), \mathbf{b}_2(\mathbf{x}), c_2(\mathbf{x})$ együtthatókkal kell számolunk. Ekkor az alábbi közelítéseket téve:

$$\mathbf{A}(\mathbf{x}) = \frac{\mathbf{A}_1(\mathbf{x}) + \mathbf{A}_2(\mathbf{x})}{2}$$

$$\Delta \mathbf{b}(\mathbf{x}) = -\frac{1}{2} (\mathbf{b}_2(\mathbf{x}) - \mathbf{b}_1(\mathbf{x}))$$

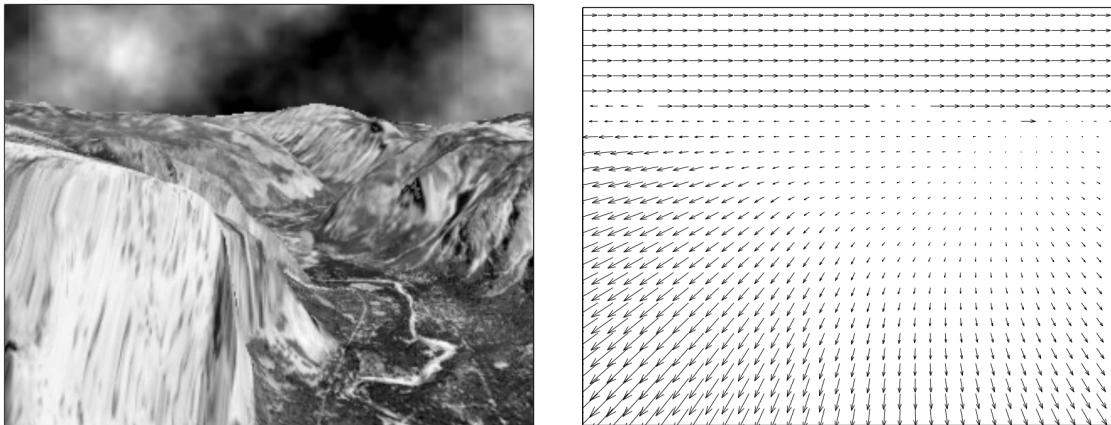
Kapjuk az előbbi egyenletekkel összevetve a következőt:

$$\mathbf{A}(\mathbf{x}) \mathbf{d}(\mathbf{x}) = \Delta \mathbf{b}(\mathbf{x})$$

Ekkor $\mathbf{d}(\mathbf{x})$ szerepében megkaptuk a vektor-mezőt. Ez az egyenlet pontonként felírva megoldható, de a gyakorlatban zajos eredményt ad. Ezt elkerülendő azzal a feltevéssel

élünk, hogy a pontok egy környezetében az elmozdulás vektorok csak kicsit változnak, és így a fenti megoldása egy költség-minimalizálási problémává változik [18].

Ez utóbbi algoritmust [17] választottam a feladat megoldása során, mert sűrű optikai folyamokat szolgáltat, és nem kell sarokpontokat keresni a kiinduló képen. A 3.7. ábrán látható egy példa, ami egy kanyon felett elrepülő helikopter videójának két egymás utáni képkockájából meghatározott sűrű optikai folyamot mutatja. Fontos megemlíteni, hogy minden két eljárás önmagában csak kis mozgásokat tud követni, ezért nagy elmozdulások esetén a piramis-módszert használjuk: a képeket több lépcsőben kicsinyítjük, és ezeken is lefuttatjuk az algoritmusokat, így a kis mozgások eltűnnek, a nagyok pedig kisebbeké, ezáltal detektálhatóvá válnak.

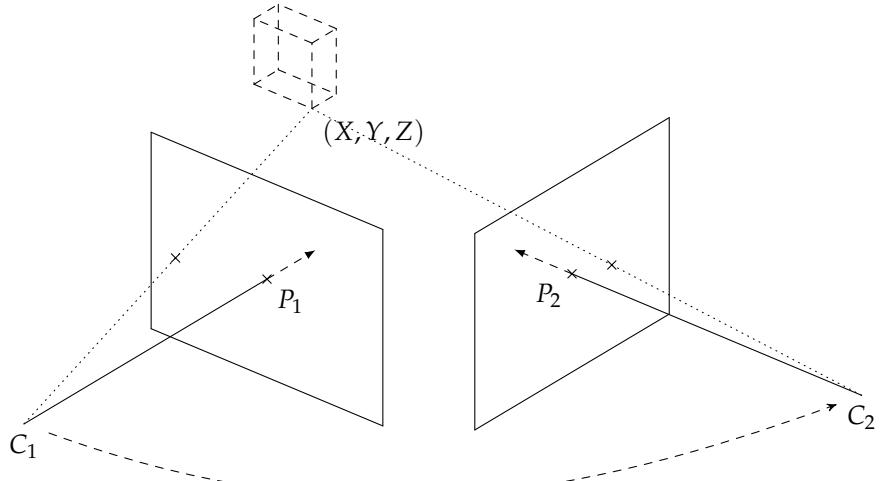


3.7. ábra. Minta sűrű optikai folyam meghatározásra [18]

Az optikai folyamokat felhasználhatjuk statikus objektumok (pl. szobrok, kastélyok) modellezésére, ezt a problémát *structure-from-motion*-nek hívjuk [19]. Tekintsünk egy adott objektum felé néző, de körülötte mozgó kamera két helyzetében (C_1 és C_2) láttott képet (lásd a 3.8. ábrát). Az objektumról a kamera mozgása során több képet készítve, és ezeken az egymásnak megfelelő pontokat az optikai folyamok segítségével meghatározva a már említett háromszögelés módszerével meghatározhatóak a pontok háromdimenziós koordinátái, valamint a nézőpontok közötti transzformációk is [20, 4. fejezet].

Ettől lényegesen eltérő felhasználási mód, ha az optikai folyamokat nem egy mozgó kamera két képkockáján, hanem két különböző kamera egy időponthoz tartozó két képkockáján határozzuk meg, melyek megközelítőleg ugyanazon térrészt veszik, de kissé eltérő pontból és szögből. Ekkor az optikai folyam lényegében a két kamera képén egy megfeleltetést tesz lehetővé. Feltéve, hogy előzetesen a kamerák pozícióját meghatároztuk (külső paraméterek), valamint ismerjük a projekciójukat leíró mátrixokat (beli paraméterek), akkor az egyező képpontokat felhasználva háromszögeléssel kaphatjuk a minden két kamera által látott pontok világbeli koordinátáit.

Az előbbi módszer jól használható nagy felbontású képeknél, ahol az objektum a kép nagy részét teszi ki, és így sok hasznos képpontot gyűjthetünk. A feladatom során



3.8. ábra. Mozgó kamera két állapotban

éppen ellenkezőleg, a képeket csak egy részén megjelenő és ott mozgó objektumokat kellett rekonstruálni, így az utóbbi megközelítést valósítottam meg.

3.3.3. Háromszögelés

Legyen adott két kamera a belső és külső paramétereivel együtt, tehát ismerjük a kamerák \mathbf{P} és \mathbf{Q} projekciós mátrixát. Amennyiben vesszük egy P pont vetületeit a kamerák képein, akkor háromszögeléssel meghatározhatjuk ezen pont valóvilágbeli koordinátáit.

Legyen a két egymásnak megfeleltetett képpont \mathbf{u} és \mathbf{u}' . A megfeleltetés zajos jellege miatt, ezek általánosságban nem elégítik ki az epipoláris $\mathbf{u}'^T \mathbf{F} \mathbf{u} = 0$ kényszert, ahol \mathbf{F} a két kamera fundamentális mátrixa. Ilyenkor ezek helyett olyan $\hat{\mathbf{u}} \leftrightarrow \hat{\mathbf{u}'}$ megfeleltetést keresünk, mely minimalizálja a

$$d(\mathbf{u}, \hat{\mathbf{u}})^2 + d(\mathbf{u}', \hat{\mathbf{u}'})^2$$

költségfüggvényt, ahol $d(., .)$ két pont euklideszi távolsága, és kielégíti az

$$\hat{\mathbf{u}}'^T \mathbf{F} \hat{\mathbf{u}} = 0$$

epipoláris kényszert [21]. Amikor megvan $\hat{\mathbf{u}}$ és $\hat{\mathbf{u}'}$, már meghatározhatjuk a kamerák középpontjából induló ezen pontokon átmenő sugarak metszéspontjait, hiszen biztosan létezni fog. Hartley és Sturm [21] megad egy optimális polinomiális megoldást, mely a fenti költségfüggvényt minimalizálja Gauss-elasztikus zaj feltételezése esetén.

Ennek ellenére a gyakorlatban inkább lineáris háromszögelést szoktak alkalmazni [21, 5.1 szekció]. Legyen \mathbf{x} egy pont és ennek vetülete az első kamera képén \mathbf{u} , vagyis $\mathbf{u} = \mathbf{Px}$. Legyen \mathbf{u} homogén koordinátákkal $\mathbf{u} = w_u [x_u, y_u, 1]^T$, tehát

$$w_u \begin{bmatrix} x_u \\ y_u \\ 1 \end{bmatrix} = \begin{bmatrix} \mathbf{p}_1^T \\ \mathbf{p}_2^T \\ \mathbf{p}_3^T \end{bmatrix} \mathbf{x},$$

ahol \mathbf{p}_i^T jelöli a \mathbf{P} mátrix i . sorvektorát. Így összesen 3 egyenletet kapunk:

$$w_u x_u = \mathbf{p}_1^T \mathbf{x} \quad w_u y_u = \mathbf{p}_2^T \mathbf{x} \quad w_u = \mathbf{p}_3^T \mathbf{x}$$

Harmadik egyenletet felhasználva az első kettőnél, kapjuk, hogy:

$$x_u \mathbf{p}_3^T \mathbf{x} = \mathbf{p}_1^T \mathbf{x}$$

$$y_u \mathbf{p}_3^T \mathbf{x} = \mathbf{p}_2^T \mathbf{x}$$

Ugyanezt levezetve $\mathbf{v} = w_v [x_v, y_v, 1]^T = \mathbf{Qx}$ képpontra:

$$x_v \mathbf{q}_3^T \mathbf{x} = \mathbf{q}_1^T \mathbf{x}$$

$$y_v \mathbf{q}_3^T \mathbf{x} = \mathbf{q}_2^T \mathbf{x}$$

A fenti négy egyenletet rendezve:

$$\begin{pmatrix} x_u \mathbf{p}_3^T - \mathbf{p}_1^T \\ y_u \mathbf{p}_3^T - \mathbf{p}_2^T \\ x_v \mathbf{q}_3^T - \mathbf{q}_1^T \\ y_v \mathbf{q}_3^T - \mathbf{q}_2^T \end{pmatrix} \mathbf{x} = 0.$$

Ez meghatározza \mathbf{x} -et egy faktor erejéig, és ennek nem-zérus megoldását keressük. Zajos adatokat feltételezve nem feltétlen találunk pontos megoldást, így közelítenünk kell. Egy lehetséges megközelítés a *Linear-LS* (lineáris, legkisebb négyzetek) módszer, mely $\mathbf{x} = [x, y, z, 1]^T$ alakú megoldások között keresi az egyenletrendszer legkisebb négyzetek megoldását (pl. szinguláris érték szerinti felbontással (SVD) [22]).

Általános esetben a talált \mathbf{x} nem pontos megoldása a fenti egyenleteknek, azaz egy $\varepsilon = x_u \mathbf{p}_3^T \mathbf{x} - \mathbf{p}_1^T \mathbf{x}$ hibával számolhatunk az egyenletrendszer első egyenleténél. Mi nem ezt, hanem x_u és \mathbf{x} vetületének első koordinátája közötti eltérését szeretnénk minimalizál-

ni, azaz $\epsilon' = \frac{\epsilon}{\mathbf{p}_3^T \mathbf{x}} = x_u - \frac{\mathbf{p}_1^T \mathbf{x}}{\mathbf{p}_3^T \mathbf{x}}$. Tehát, ha az egyenletet $1/\mathbf{p}_3^T \mathbf{x}$ -szel súlyoztuk volna, akkor az úgy kapott hiba pont az általunk minimalizálálandó hiba lett volna. Természetesen ezt nem tudjuk megtenni, hiszen \mathbf{x} még nem ismert, csak miután megoldottuk az egyenletrendszer, de egy iteratív megközelítést tudunk alkalmazni. minden iteráció végén újrasúlyozzuk a megfelelő sorokat $1/\mathbf{p}_3^T \mathbf{x}_i$ -vel, valamint $1/\mathbf{q}_3^T \mathbf{x}_i$ -vel, ahol \mathbf{x}_i az i . iteráció végén kapott megoldás. Ha a lépések nél az aktuális megoldást az előbbi *Linear-LS*-sel keressük, akkor az így kapott eljárást *Iterative-LS*-nek hívjuk. Hartley és Sturm [21, 5.2 szekció] munkája alapján az esetek 95%-ban a megoldás konvergens (epipólus melletti pontokra volt divergens), és amikor az új súly már csak kis mértékben változik az előző-höz képest, megszakíthatjuk az iterációt.

3.4. Objektum-detektálás

Objektumok detektálására a szakirodalomban több eltérő módszert is találhatunk. Az egyik legegyszerűbb ezek közül a minta alapú illesztés (*template matching*) [23]. Ekkor lényegében egy minta képet illesztünk egy forrás képre csúszóablakos módszerrel, és valamely metrika mentén minden pozícióban egy illeszkedési értéket számolunk. Ezt az egész képre meghatározva, majd ennek ennek a maximumát véve kapjuk az illeszkedési pontot, ahol a keresett objektumot megtaláltuk.

Egy másik nagyon elterjedt módszer a kaszkád osztályozó (*cascade classifier*), ami gépi tanuláson alapszik [24]. Ennek segítségével egy előzetes tanulási szakasz után viszonylag komplex objektumok felismerése is nagyon gyorsan lehetővé válik.

A harmadik módszer, amit kiemelnék a jellegzetes pontok megfeleltetésén (*feature matching*) alapszik. A keresendő objektum képén és a forrás képen különböző algoritmusok segítségével jellegzetes pontokat emelünk ki, majd ezek tulajdonságait (pixel színe, annak környezete stb.) vektorokkal írjuk le. Ezeket a vektorokat úgy párosítjuk, hogy valamely – a vektorok típusától függő – értelemben vett távolságuk a lehető legkisebb legyen. Az így adódó megfeleltetésekkel kapjuk a keresett objektum helyét a forrás képen.

3.4.1. Mozgó objektumok kiválasztása

Mivel a feladat megoldása során mozgó objektumokat kell követnem, azokat modellez-nem, így szükséges a mozgó objektumok statikus háttér-től való elkülönítése.

Egy lehetőségünk például az előző részben leírt Farnebäck-módszer használata. Egy kamera egymás utáni képkockáira alkalmazva, becslést kaphatunk arra vonatkozóan, hogy hol lehetnek a mozgó tárgyak. A kamerák statikus helyzete révén, a háttér képpontjaihoz tartozóan közel zéró elmozdulást várunk, míg a mozgó részeken ettől eltérőt.

Másik megoldás lehet például egy háttér-előter szegmentálási algoritmus (MOG) [25], mely az egymás utáni képkockákból egy háttér-modellt épít. Az aktuális képkocká-

ból a kapott háttér kivonva, megkapjuk az éppen aktuális előtérhez tartozó maszkot, lásd a 3.9. ábrát. Ezen algoritmus előnye, hogy az árnyékokat nagy valószínűséggel kiszűri, és azokat nem tekinti az objektum részének, valamint a folyamatos modellépítésnek köszönhetően, nem csak egy fix tanulási időszak alatt gyűjtött információkból határozza meg a háttteret.



3.9. ábra. Minta a háttér-előtér elválasztási algoritmus eredményére [26]

3.4.2. Objektumok meghatározása a kamerák képein

Miután meghatároztam a mozgó objektumokhoz tartozó maszkokat, az ezek által kijelölt képrészleteket kell úgy párosítanom a kamerák képein, hogy azok egy valódi objektumhoz tartozzanak.

Erre a feladatra a kaszkád osztályozót a szükséges előzetes tanítás miatt elvetettem. A minta alapú illesztést alkalmazhattam volna úgy, hogy az egyik képen meghatározott képrészleteket egyenként használom a másik kép számára mintaként. Ahogy majd később látni fogjuk az optikai folyam alapú sűrű pont-pont megfeleltetés robosztus meghatározásához szükségünk lesz már egy előzetes megfeleltetésre, ezért az előző fejezetben leírt objektumok detektálásához használható eljárások közül a jellegzetes pont alapú megfeleltetést választottam. Ennek segítségével párosítottam az előtérhez tartozó képrészleteket a kamerák képein, így sikeresen detektáltam a mozgó objektumokat a kamerák képein.

3.5. Összefoglaló

Ebben a fejezetben a több kamerából álló kamera-rendszerek során felmerülő és a dolgozat során is megoldandó problémákat, valamint ezek lehetséges megoldásait mutattam be. Kitertem a kamerák szinkronizációjára, az epipoláris geometriára, mely kapcsolatot teremt a kamerák képei között, és leírtam két lényegében eltérő rekonstrukciós eljárást, és azok mögött meghúzódó elméleti háttteret. Bemutattam a háromszögelés elméleti módszerét, végül a fejezetet az objektum-detektálás problémakörével zártam.

4. fejezet

Tervezés

Az elméleti megfontolások, alapelvezek bemutatását követően ebben a fejezetben rátérek a feladat megoldását adó rendszer struktúrájának, főbb komponenseinek tárgyalására.

4.1. Specifikáció kidolgozása

A diplomamunka során elkészített rendszernek képesnek kellett lennie egy zárt térrész tetszőlegesen választott pontjában és irányában látható, a mozgó objektumokat tartalmazó kép helyreállítására fix telepítésű kamerák valós idejű videofolyamai alapján. A kamerák fix helyzete egyszerűsíti a problémát, mert ezzel a kamera koordinátarendszert kalibráció segítségével előre össze lehetett hangolni a világ-koordinátarendszerrel. A helyreállítást akkor tekintettem sikeresnek, ha a mozgó objektumokat detektáltam és választott nézőpontból azok megközelítő kontúrjait sikeresen meghatároztam. A tényleges objektum struktúrájának illetve textúrájának meghatározása nem esett a dolgozat hatáskörébe. A választott nézőpontra is voltak korlátaink, érdemi rekonstrukciót csak a kamerákat összekötő képzeletbeli szakaszon és annak környezetében várhatunk, így a tesztelést is így végeztem el.

A rendszer tervezése során figyelembe vetttem, hogy lényegében tetszőleges számú kamerát is felhasználhattam a probléma megoldásához, de az elérhető eszközök korlátozott száma miatt csak két kamerát felhasználva készült el a konkrét implementáció.

Egyre több kamera bevonásával a rekonstrukció minősége, valamint a helyesen rekonstruálható nézőpontok száma növekszik, de ezzel együtt a rendszer teljesítménye a megnövekedő számítási szükséglet miatt csökken. Fontos megemlíteni, hogy a feladat és a megoldás jellegéből adódóan a független kamerákból meghatározott rekonstrukciók párhuzamosíthatóak, így ezek megfelelő feldolgozó egységet feltételezve egy időben elvégezhetők. Ez azt jelenti, hogy a kamerák számának növelése valóban megoldás lehet a robusztusabb eredmény eléréséhez.

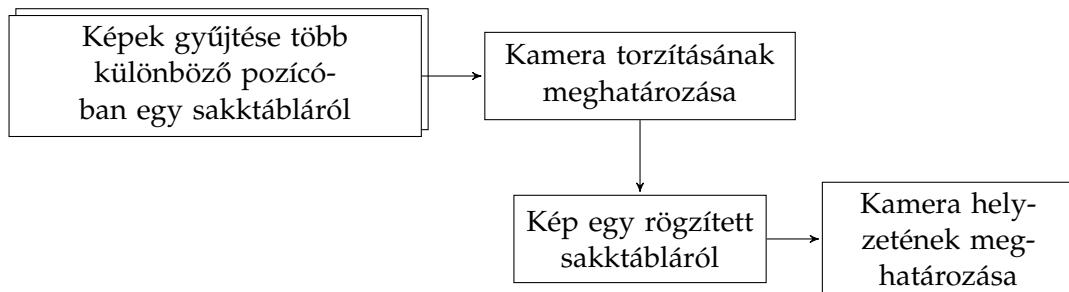
4.2. Keretrendszer

A diplomamunka készítése során az OpenCV [27] keretrendszert használtam, melynek célja, hogy a fejlesztőknek egy szabadon és ingyenes elérhető alkalmazás-könyvtárat biztosítson a gépi látás területén elterjedt és gyakran használt algoritmusokhoz. Több nyelvhez is biztosít API-t, én ezek közül a C++-os interfészét alkalmaztam, a lehető legnagyobb teljesítmény elérése érdekében. Az OpenCV-t már az előző félévekben megismertem, így a diplomamunka során már eredményesen tudtam építkezni az általa nyújtott funkciókra.

4.3. A tervezett rendszer működése

A feladat megoldásához a 3.3.2. részben leírt optikai folyam alapú eljárást használtam fel. A rendszer tervét, átfogó képét a következőkben mutatom be.

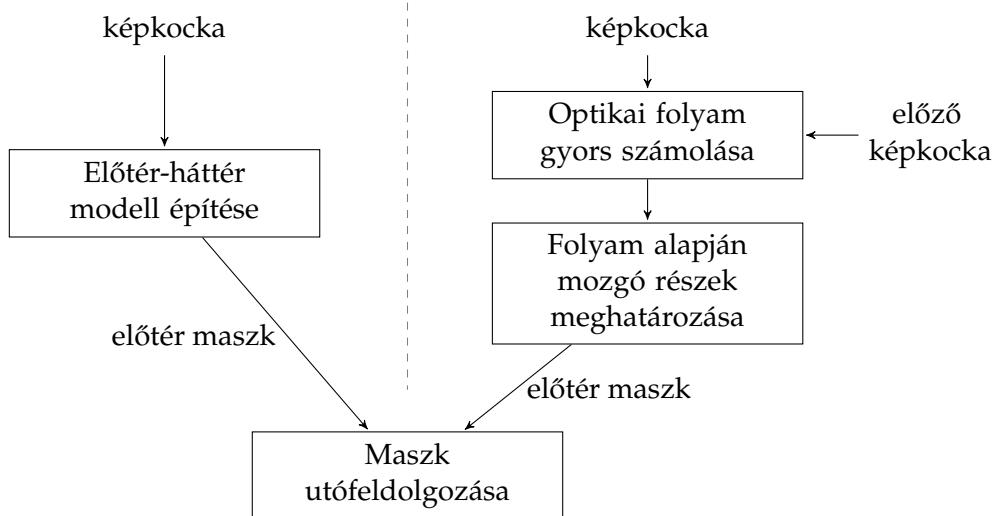
A kamerákat két lépésben kalibráltam; először, hogy a torzításukat minimalizáljam, majd azért, hogy meghatározhassam az elhelyezkedésüket egy rögzített koordinátarendszerben, melyet egy sakktábla segítségével definiáltam. Ezt a 4.1. ábra szemlélteti.



4.1. ábra. Kamerák kalibrációja

Ezt követően a kamerák képeiből meghatároztam az előtér maszkot. Erre kétféle megközelítést is implementáltam; az egyik az optikai folyamokat használja a mozgás érzékeléséhez, ami kijelöli az előteret, a másik pedig egy előtér-háttér szegmentációs eljárás. Ezeket a 4.2. ábra mutatja be, közülük összehasonlítás után választottam, lásd a következő fejezetet.

Miután a vélt előtér objektumok maszkjait meghatároztam, ezek által kijelölt képrészleteket (*blobok*) a kamerák képein fel kellett ismernem és egymással párosítanom. Az első egy egyszerű „legnagyobb-területű” kiválasztás, amely minden képen a legnagyobb területű *blobot* keresi, és ezeket párosítja egymással. Természetesen ezzel csak egy objektumot rekonstruálhatunk, de a további lépésekkel jól lehet rajta tesztelni. A másik eljárás pedig, hogy a képrészleteken jellegzetes pontokat kerestem, ezeket egymással párosítottam, majd többségi döntést alkalmazva meghatároztam, hogy az egyik kép blobja melyik másik blobnak felel meg. Így megkaptam az ugyanazon objektumokhoz tartozó képrészleteket a kamerák képein. Ezt a két folyamatot a 4.3. ábra mutatja be.



4.2. ábra. Előtér maszk meghatározásának két módja

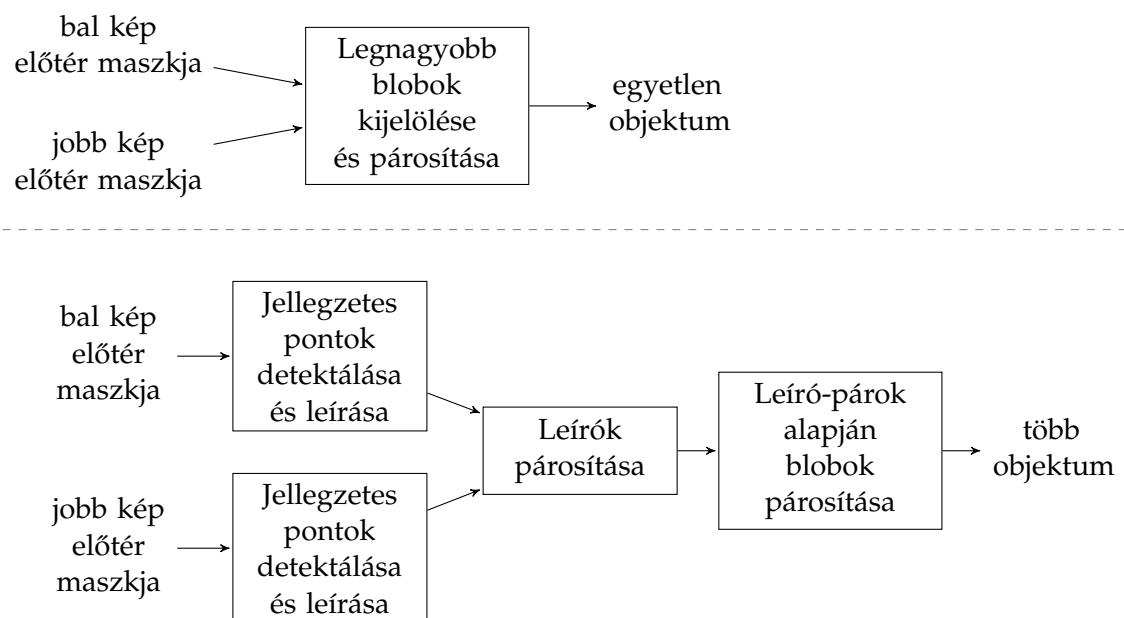
Minden objektumhoz külön-külön optikai folyamokkal sűrű pont-pont megfeleltést számoltam a két képen. Felhasználva, hogy a kamerák helyzetét ismertem, háromszögeléssel megkaptam az adott objektum minden két képen látható pontjainak világbeli koordinátáit.

Végül a választott nézőpontból projekció segítségével meghatároztam az onnan látható becsült képet (felhasználva az eredeti képből kinyerhető színinformációkat), valamint az objektumokhoz tartozó kontúrokat.

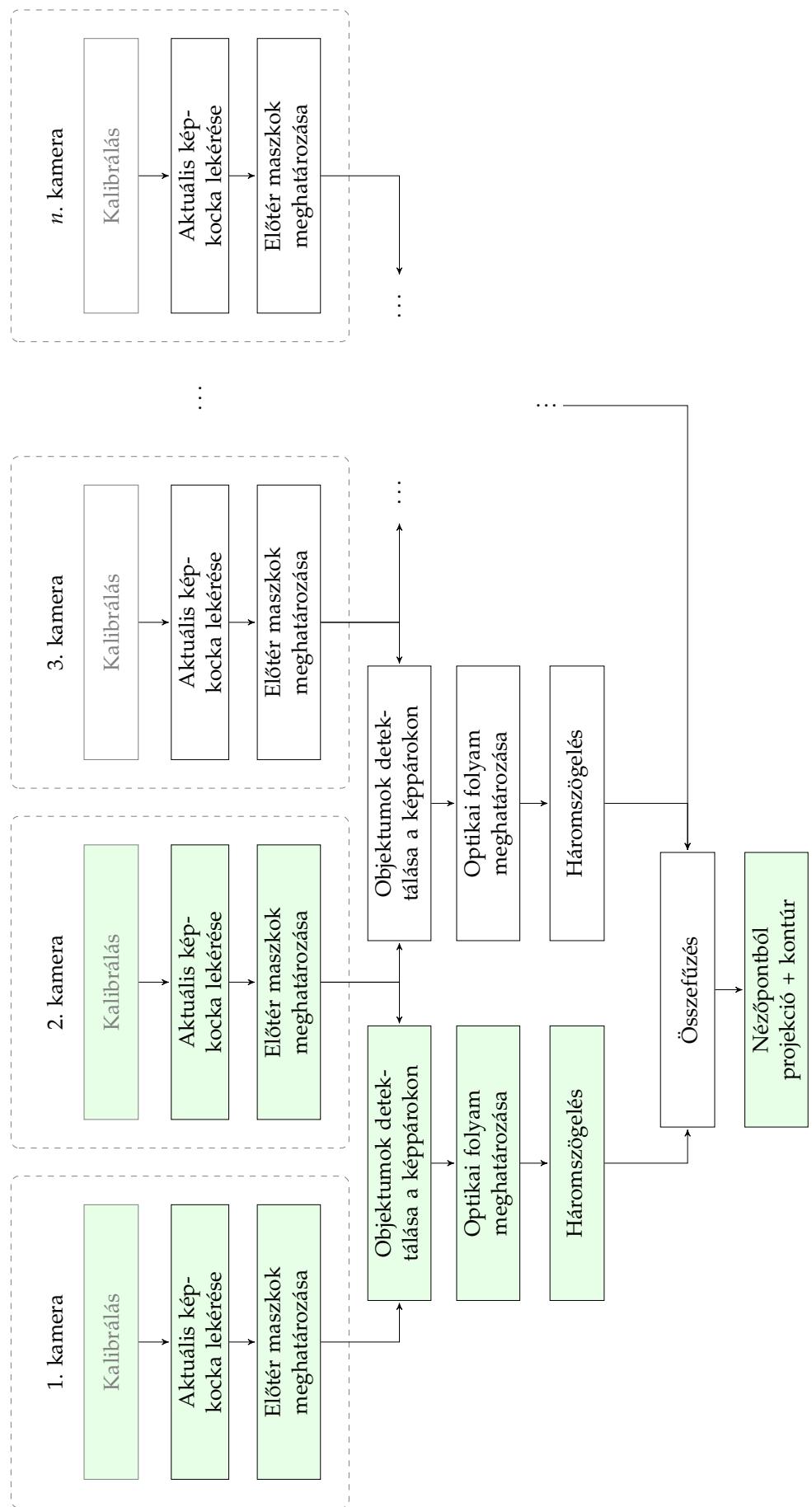
A fenti eljárás tetszőleges számú kamera-párra (amelyek nem feltétlen diszjunktak) általánosítható, és így kapott pontfelhők – felhasználva, hogy az adatokat az előzetes kalibrációnak köszönhetően egy közös világ-koordinátarendszerben kapjuk meg – könnyen egyesíthetők. A diplomaterv keretében a 4.4. ábrán (32. oldal) látható színezett hátterű lépések készültek el, a több kamerára vonatkozó együttes kezelés nem.

4.4. Összefoglaló

Ebben a fejezetben bemutattam az alkalmazott keretrendszeret, melyre a megoldásomat építettem. Kidolgoztam a specifikációt, miszerint két rögzített kamera által megfigyelt térrészben rekonstruálók egy illetve két mozgó objektumot egy a két kamera közötti szakaszon választott nézőpontból. Rögzítettem a sikerkritériumot, hogy a helyreállítást akkor tekintem sikeresnek, ha a mozgó objektumok megközelítő kontúrjait sikeresen meghatározzom. Végül leírtam az implementálásra kerülő rendszer lépésekre bontott logikáját.



4.3. ábra. Objektumok detektálásának két módja



4.4. ábra. Elkészített megoldás átfogó terve

5. fejezet

Megvalósítás

Ebben a fejezetben az előzőekben bemutatott rendszer megvalósítását mutatom be lépésekkel, melyek a 4.4. ábrában közölt sorrendet követik. A beszámoló során kitérek az alkalmazott jelentősebb OpenCV eljárásokra és a választott paraméterekre, amennyiben azok további magyarázatot igényelnek.

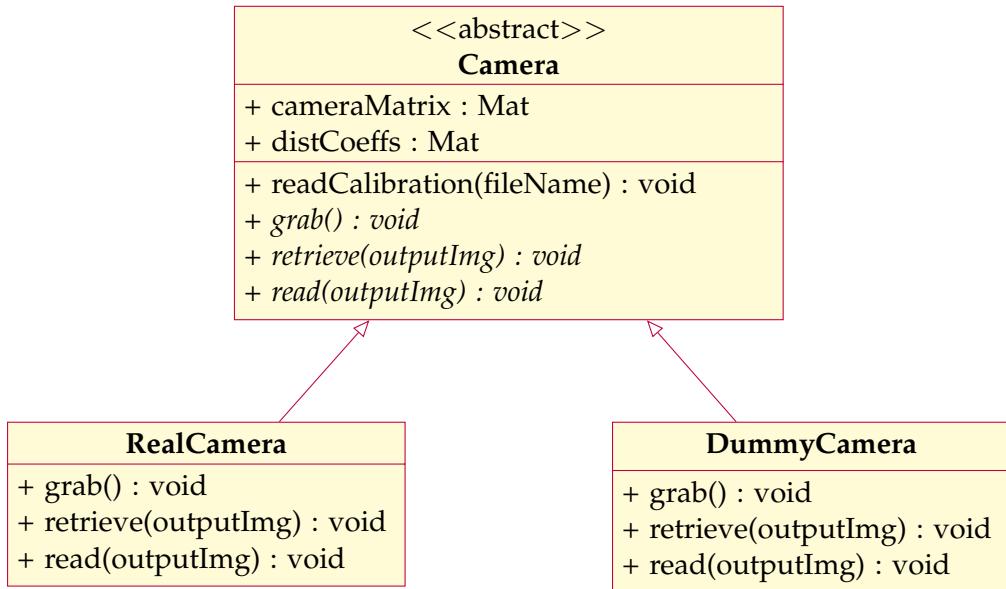
5.1. Alapok

Az OpenCV igen hasznos keretrendszer, rengeteg gyakran használt algoritmus implementációját tartalmazza, viszont felépítését tekintve procedurális. Jelen feladatom megoldása során is törekedtem az átlátható és jól struktúrált kód kialakítására, így a logikát jól körülhatárolt osztályokba szerveztem.

OpenCV-ben a legtöbb adatot egy mátrix (`cv::Mat`) adattípus reprezentál, ide értve a matematikai értelemben vett mátrixokat és a képeket is. Egy ilyen mátrix lényegében egy kétdimenziós tömb, melynek elemei lehetnek skalárok, de több-dimenziós vektorok is (több csatornás).

Elsőnek a `Camera` osztály, és annak konkrét implementációi készültek el, elfedve azt, hogy éppen a valódi kamerából kérünk le képkockákat, vagy – a tesztelés megkönnyítése céljából – fájlból olvassuk ki azokat. Ebből adódóan a `RealCamera` lényegében becsomagolja az OpenCV-s `VideoCapture` osztályt, és a `Camera` abszakt osztály közös interfészét nyújt a fájlból történő olvasáshoz is a `DummyCamera` számára. Utóbbi alkalmazása révén egy adott jelenetet elég volt egyszer felvennem, majd utána ezt többször használhattam bemenetként. Az osztálydiagram az 5.1. ábrán látható.

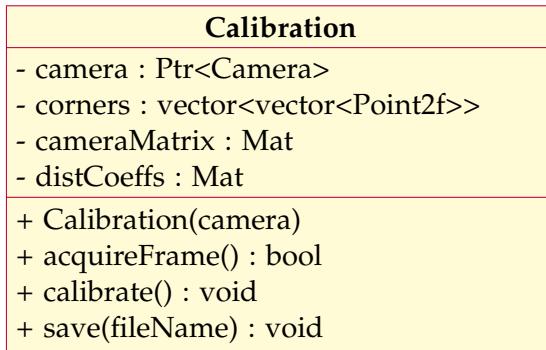
A `cameraMatrix` attribútum jelöli a *kamera-mátrixot* és a `distCoeffs` attribútum a torzítási együtthatókat (lásd a 2.6. szekciót). Mivel ezek egy kamerára nézve időben állandók, ezért csak egyszer kell őket meghatározni. A `readCalibration()` metódus szolgál ezek beolvasására egy külső fájlból.



5.1. ábra. Osztályok a kamerák kezeléséhez

5.2. Kalibráció

Elsőként a kamerák belső paramétereit határoztam meg. A módszert a 2.6. szekcióban mutattam be, a következőkben ennek megvalósítását tárgyalom. Ehhez készült egy segédosztály `Calibration` névvel, melynek feladata, hogy a szükséges információk alapján meghatározza a kamera-mátrixot és a torzítási együtthatókat, valamint kiírja ezeket egy fájlba, hogy azokat vissza lehessen olvasni.



5.2. ábra. *Calibration* osztály

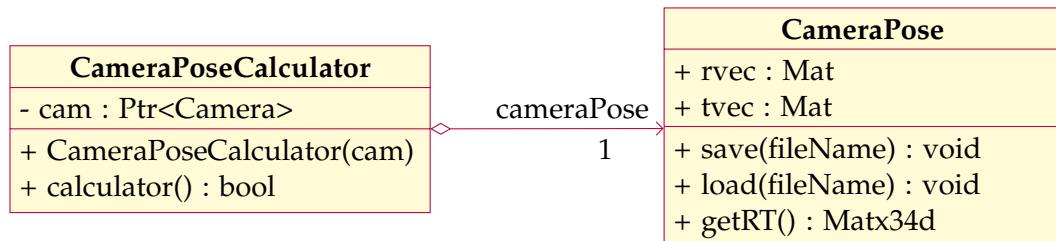
Konstruktőrben át kell neki adni a kamerára vonatkozó pointert, amitől a képeket kell majd lekérnie. Az `acquireFrame()` metódus szerepe, hogy a kamera aktuális képkockáját megszerezze, megkeresse a képen a sakktáblát, és a sakktábla sarokpontjaihoz tartozó koordinátákat a `corners` listához fűzze. Visszatérési értékben jelzi, hogy az adott képen sikeres volt-e a detekció. Belsőleg a `cv::findChessboardCorners()` OpenCV-s függvényt hívja, amelynek átadva egy fekete-fehér képet, megkapható a képen látható sakktábla sarokpontjainak képpontjai. Kellő számú (10-15) képkocka után a `calibrate()` metódus segítségével a kérdéses két mátrix (`cameraMatrix`, `distCoeffs`) kiszámolható.

Itt a `cv:::calibrateCamera()` függvényt hívtam segítségül, melynek két fontos bemeneti paraméterét emelem ki: a sarokpontok valóvilágbeli koordinátái, és a képeken detektált képpontjai sorfolytonosan (corners). A valóvilágbeli (x, y, z) koordinátákat az egyszerűség kedvéért úgy választottam, hogy $z \equiv 0$, és x valamint y egész számok úgy, hogy a sakktábla bal felső sarka $(0, 0, 0)$, jobb alsó sarka pedig -9×6 -os sakktáblát használva $- (9, 6, 0)$ koordinátákat kapta. A `save()` metódus a kiszámolt paramétereket menti el a megadott fájlnévvel olyan formátumban, amit a `Camera:::readCalibration()` vissza tud olvasni.

5.2.1. Kamerák pozíciójának meghatározása világkoordinátákban

Rögzített kamerák révén lehetőségem adódott, hogy előre meghatározzam a kamerák pozícióját és nézőpontjuk irányát. Ehhez szintén egy sakktáblát használtam kalibrációs objektumként. A sakktábla sarokpontjainak az előbbiekkel egyező módon vett koordinátázásával a világ koordinátarendszerét is rögzítettem.

Az OpenCV-ben a `solvePnP` függvény segítségével 3D-2D pont-összerendelésekkel kiszámolhatjuk a forgatási és eltolási vektort, amik együttesen megadják a transzformációt a világ-koordinátarendszerből a kamera koordinátarendszerébe. Ezen funkciót a `CameraPoseCalculator` osztályba ágyaztam, a két vektort pedig a `CameraPose` perzisztaálható osztályba csomagoltam, lásd az 5.3. ábra.

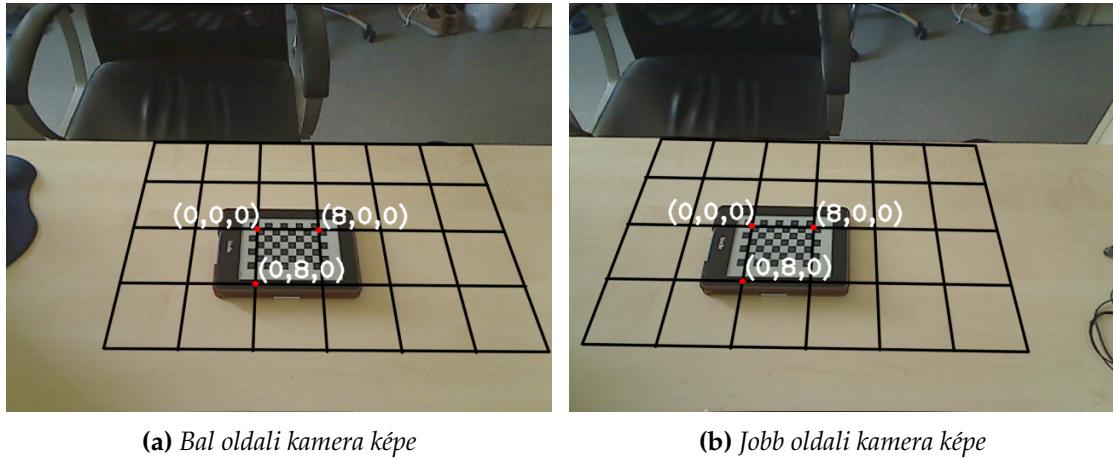


5.3. ábra. `CameraPoseCalculator` és `CameraPose` osztály

A `CameraPoseCalculator` osztály megkapja konstruktor argumentumaként annak a kamerára mutató pointerét, amelynek a külső paramétereit (forgatási és eltolási vektor, lásd 2.6. szekció vége) ki kell számolnia. A konkrét művelet végrehajtásáért a `calculator()` metódus felelős, amely a kamerától lekér egy képkockát, megkeresi rajta a sakktáblát, majd meghívja a `cv:::solvePnP()` függvényt. Visszatérési értékben jelzi, hogy sikeres volt-e a detekció, ha igen, akkor lekérhető tőle a `CameraPose` példány. Ez utóbbi a `save()` és `load()` metódusokkal elmenthető és visszatölthető, így ameddig a kamerát nem mozgatjuk el, ez újra felhasználható. A `getRT()` metódus a forgatási vektort a Rodrigues-féle forgatási formulával [9] forgatási mátrix-szá alakítja, és összefűzi azt az eltolási vektorral egy 3×4 -es $(R | t)$ forgatás-eltolás mátrixba.

A kamera külső paramétereinek meghatározása után már minden információ adott, hogy 3D-s pontok 2D-s vetületeit meg tudjam határozni a `cv:::projectPoints()` függ-

vény felhasználásával. Az 5.4. ábrán látható egy, a sakktábla síkjába rajzolt négyzetrács, melynek egyik jelölt pontja a világ-koordinátarendszer origója.



5.4. ábra. Világ-koordinátarendszer jelölése a képeken

5.3. Objektum detektálás

A következőkben az objektumok detektálásához kapcsolódó implementációs részleteket tárgyalom. Először bemutatok két megközelítést az előtér maszk meghatározásához, ami lényegében kijelöli a két képen látható mozgó részeket. Ezt követően a maszkokat szegmentálom különálló *blobokra* (egy objektumhoz tartozó egybefüggő rész a képen), és ezeket a két képen egymásnak megfeleltetem, hogy azonosítsam az egy objektumhoz tartozó képrészleteket.

5.3.1. Előtér maszk meghatározása

Előtér-háttér szegmentálás

A 3.4.1. szekcióban leírtam a mozgó objektumok detektálásának háttér modell építésével történő módszerét. A lényege, hogy a kamerák videofolyamai alapján háttér modellt építünk, ami alapján minden aktuálisan lekérhető az előtérhez tartozó maszk, ami kijelöli a mozgó objektumokat. OpenCV-ben ezt valósítja meg a `BackgroundSubtractorMOG2` osztály [28]. Példányosítás után a modell építése, és az aktuális maszk kinyerése az `apply` metódussal történik. Az 5.5. ábrán látható a kinyerhető maszkra egy példa.

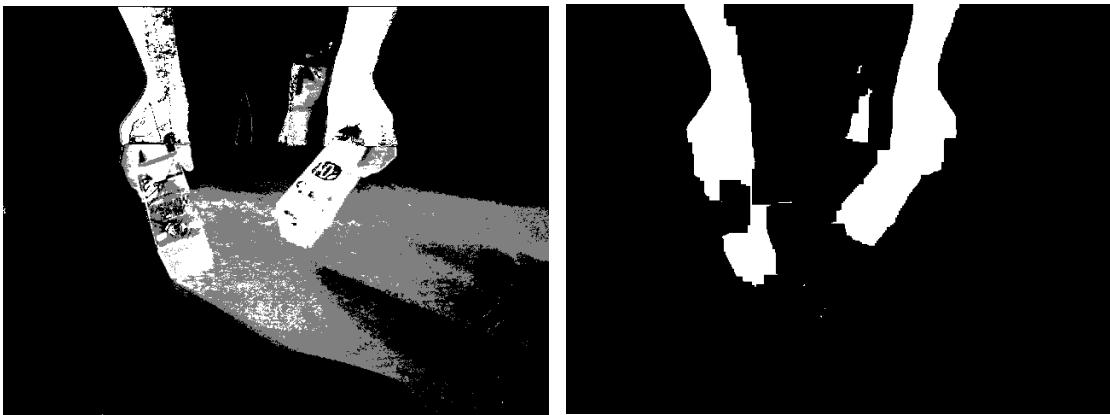
Megfigyelhető, hogy a maszk nem bináris, hanem háromértékű: azt is tartalmazza, amit az algoritmus árnyékként detektál (ezt az ábra szürke színnel jelöli). A zajt a maszkon az erózió-dilatáció morfológiai módszerek segítségével tudjuk csökkenteni. Előbbi a kisméretű zajokat tünteti el, utóbbi pedig a lyukakat szünteti meg. OpenCV-ben ezek implementációi a `dilate` és `erode` függvények. Előbbi az adott pixelt a környezetében (amit egy kernel ír le) lévő maximális, míg utóbbi a minimális értékkel helyettesít. Én egy



(a) *Statikus kép – „háttér”* (b) *Videofolyam egyik képkockája* (c) *Kapott előtér maszk*

5.5. ábra. Példa az előtér maszkra

erózió-dilatáció-erózió lépéssorozatot használtam, amely egy morfológiai nyitás és záras egymásutánja [29]. Ennek az eredményét az 5.6. ábra mutatja be.



(a) *Eredeti maszk* (b) *Árnyékok kivétele és zajcsökkentés után*

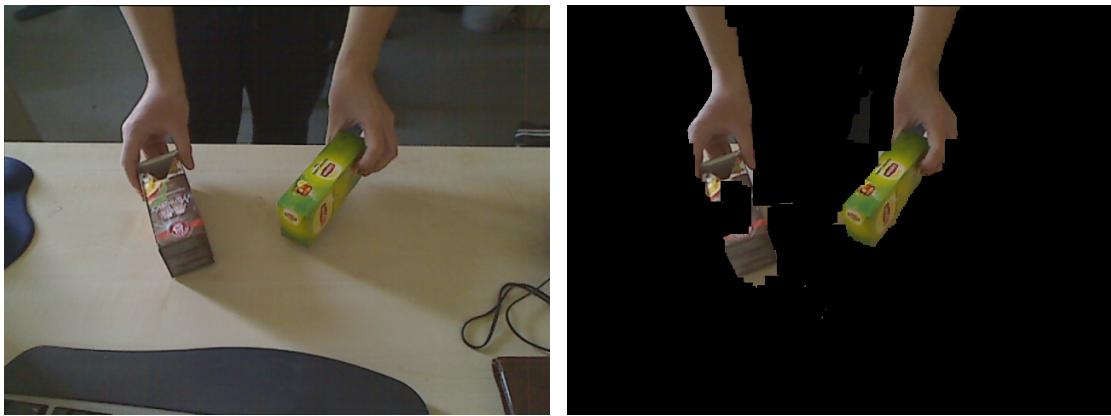
5.6. ábra. *Előtér maszk zajmentesítése*

Megfigyelhető, hogy a bal oldali dobozhoz tartozó maszk annyira zajos volt, hogy a nagy lyuk nem szűnt meg, míg a kevésbé zajos jobb oldali doboz rendben megmaradt. A maszkot alkalmazva az eredeti képre az 5.7. ábrán látható, hogy egy bizonyos hibahatáron belül sikeresnek tekinthető a mozgó részlet kijelölése.

Előtér meghatározása optikai folyamokkal

Másik megközelítésem során már az előtér meghatározásához is az optikai folyamokat hívtam segítségül.

A 3.3.2. részben bemutattam a sűrű optikai folyamok meghatározására Gunner Farnebäck módszerét. OpenCV-ben ezt a `cv::calcOpticalFlowFarneback()` [28] függvény valósítja meg. Az algoritmus szempontjából fontos paraméterei közül kiemelném a következőket:



(a) Eredeti kép

(b) A detektált előtér

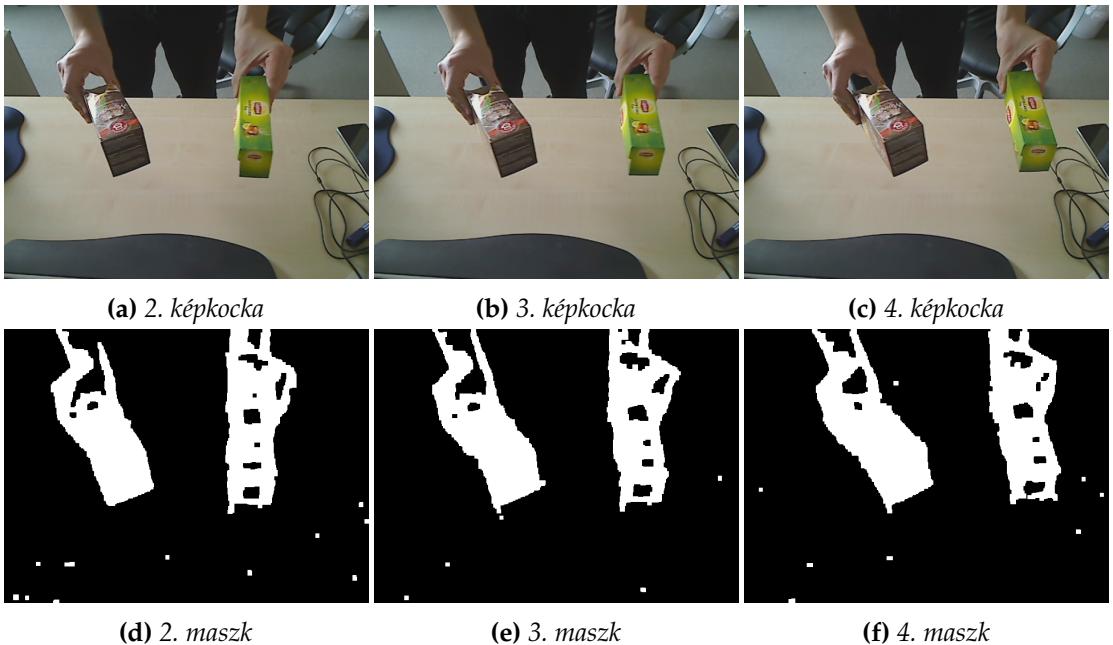
5.7. ábra. Előtér maszk alkalmazás az eredeti képre

1. `pyr_scale` – a már említett piramis-módszerhez kapcsolódó paraméter: azt definiálja, hogy rétegenként a következő réteg hányszorosa az előzőnek (ennek segítségével a nagy elmozdulások kisebbek lesznek a kisebb rétegeken)
2. `levels` – piramis rétegeinek a száma: amennyiben ez 1, akkor az optikai folyam meghatározásához csak az eredeti képet használja
3. `winsize` – az ablak méret, amit a mintavételezéshez használ
4. `iterations` – iterációk száma minden piramis rétegen

Tekintve, hogy nekünk csak az a fontos, hogy a lehető leggyorsabban számoljunk elmozdulásvektorokat, a paramétereket a következőknek megfelelően állítottam be: ne építsünk piramist (`levels = 1`, így `pyr_scale` beállítása lényegtelen), kicsi ablakmérettel dolgozzunk (3×3 , tehát `winsize = 3`), valamint rétegenként csak egy iteráció legyen (`iterations = 1`).

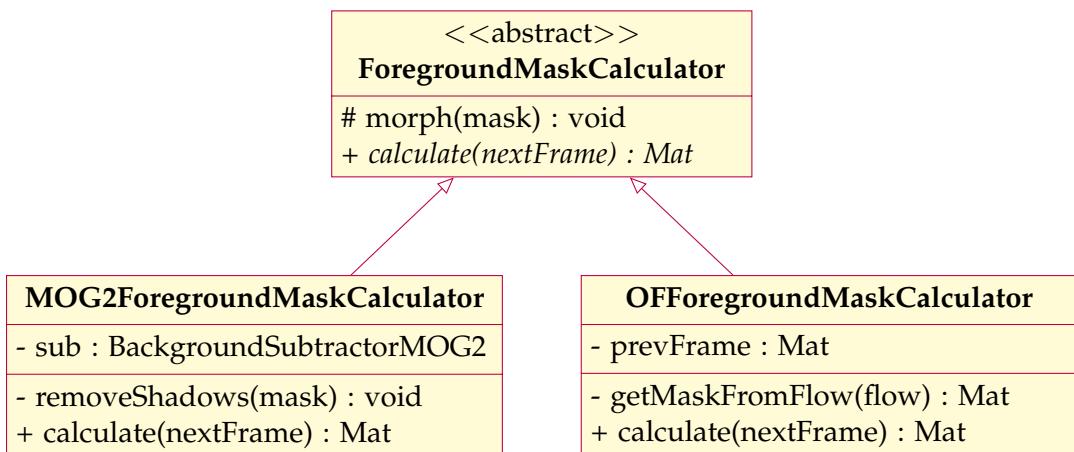
A kiszámolt vektormezőből a maszkokat úgy kaptam, hogy minden, egységnél hosszabb (tehát legalább 1 pixelnyi mozgást jelentő) vektor végpontját megjelöltem. Ezáltal lényegében azon részeket határoztam meg, ahova éppen a pontok mozdultak, ez pedig pontosan azon részei a képnek, ahol az adott képkockán az előtérben mozgó objektumokat várjuk. Az előzőekhez hasonlóan a maszkon itt is végeztem apró utófeldolgozást a dilatáció és erózió morfológiai műveletek segítségével. Az eredményt az 5.8. ábra mutatja be. Megfigyelhető, hogy a jól textúrázott részeken a maszk nagyon pontos, a textúrázatlanokon viszont a pontok statikus pontoknak tűnnek.

A bemutatott két eljáráshoz tartozó osztálydiagramot az 5.9. ábra mutatja be. `ForegroundMaskCalculator` osztály lényegében egy közös interfést nyújt csak, valamint egy `morph()` metódust tartalmaz, mely az említett morfológiai műveleteket végzi el a maszkon. A `MOG2ForegroundMaskCalculator` az OpenCV-s `BackgroundSubtractorMOG2` osztály egy példányát csomagolja be, ami minden új képkockát megkap, és visszaadja a maszkot, az algoritmus által árnyéknak jelölt részeket a `removeShadows()` távolítja el.



5.8. ábra. Egy jelenet 3 képkockája és ezekből az optikai folyamok felhasználásával kapott maszkok (már zajcsökkentés után)

OFForegroundMaskCalculator pedig eltárolja az előző képkockát, és ebből meg az éppen aktuális képkockából calcOpticalFlowFarneback() segítségével a lehető leggyorsabban meghatározza a képkockákra az optikai folyamot, melyből aztán az előbbiekben ismertetett módszerrel a getMaskFromFlow() adja vissza a maszkot.



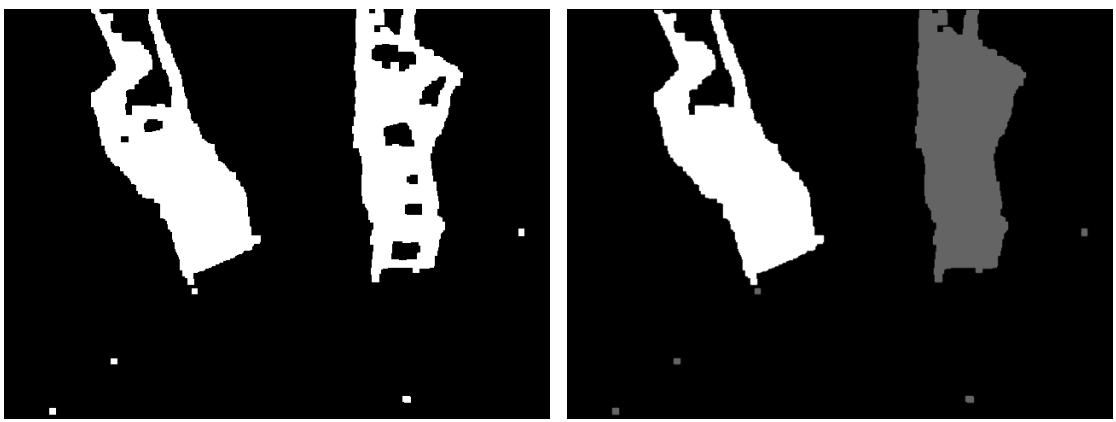
5.9. ábra. Előtér maszk meghatározására szolgáló implementációk

5.3.2. Egyetlen objektum detektálása

Kezdetben azzal az egyszerűsítéssel éltem, hogy a képen csak egyetlen mozgó objektumot detektálok, mégpedig azt, amelyik a legnagyobb részt foglalja el a képeken. Ez nagyban egyszerűsítette a problémát, mert ahogy majd a következő szekcióban bemutatom, az optikai folyam meghatározásához szükség van a képeken látható blobok párosítására a

kamerák képein. Mivel összesen egy blobot jelölök ki a képeken, ezek párosítása triviális, és nagy valószínűséggel ugyanazon objektumhoz tartoznak.

Ehhez először szükség van az összefüggő komponensek kiválasztására. Két módszer kínálkozik erre OpenCV-ben; az egyik a `findContours()` függvény, ami megkeresi a képen látható kontúrokat (paraméterezhető, hogy csak a legkülsőbbeket találja meg, a belső kontúrokat nem), a másik pedig a `connectedComponentsWithStats()`, mely a 3.0-s verziótól érhető el. Az első abban különbözik az utóbbitól, hogy a kapott kontúrt felhasználva megkaphatjuk a belső területet (lyukak nélkül), míg utóbbi lényegében a maszk pixeleit címkézi fel a komponenseknek megfelelően. Ezért én az előbbi használata mellett döntöttem. Miután meghatároztam a kontúrokat, ezek területeit a `cv::contourArea()` függvényel számoltam ki. Végül maximum kiválasztással a legnagyobb terüettel rendelkező maszk mellett döntöttem. Mindkét képre elvégezve ezt, megkaptam az egyik és másik képen a legnagyobb objektumhoz tartozó 1-1 maszkot. Az algoritmus eredménye az 5.10. ábrán látható.



(a) Eredeti maszk (b) Detektált blobok, fehérrel jelölve a legnagyobb

5.10. ábra. Legnagyobb területű blob kijelölése

5.3.3. Több objektum detektálása, párosítás a kamerák képein

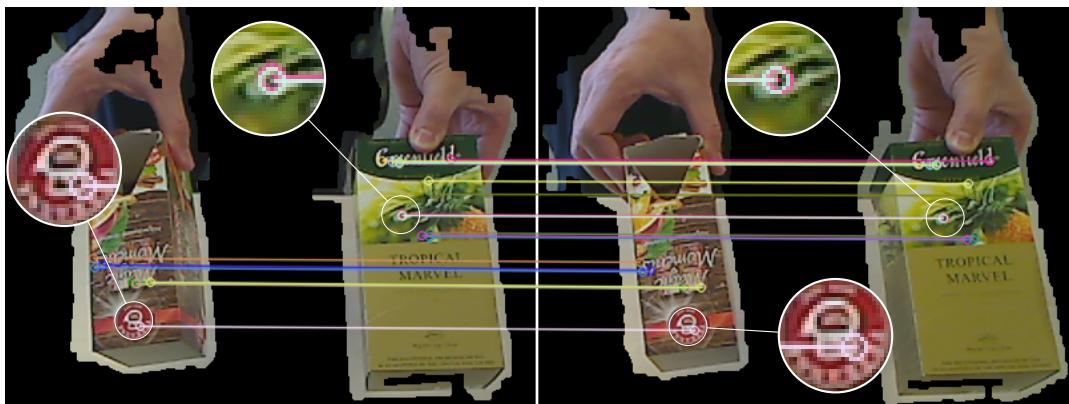
A következő feladat, amit meg kellett oldanom, az a több objektum együttes detektálása volt úgy, hogy a két képen az egymásnak megfelelő blobokat párosítom. Ehhez jellegzetes pontokat, és ezekhez tartozó leírókat kerestem, illetve számoltam ki a képi információk alapján.

A szakirodalomban több algoritmust is találhatunk, mely vagy a jellegzetes pontok (*feature*) detektálásában (pl. FAST [30], MSER [31]), vagy ezen pontokhoz tartozó leírók (*feature descriptor*) kinyerésében (pl. FREAK [32]), vagy mindenkorre (pl. ORB [33], SIFT [34] és SURF [35]) használhatóak. SIFT és a SURF a tradicionális leírók közé sorolhatóak abban a tekintetben, hogy vektor-alapú leírókat készítenek, ellenben az újabbakkal, amelyek bináris-füzéreket. Előbbi algoritmusok kiszámolása idő- és erőforrásigényes, valós idejű, valamint mobil eszközökön történő alkalmazásra kevésbé alkalmasak, utóbbiak vi-

szont igen, és a kinyert leírók összehasonlítása is gyorsabb (Hamming-távolság). Dolgozatomnak nem volt célja ezek mélyreható vizsgálata és rangsorolása, de Bekele és társai munkássága [36] alapján először a FREAK-öt próbáltam meg alkalmazni több különböző forgatás- és skálainvariáns detektor által visszaadott kulcsPontokra, sajnos kevés sikertől. Valószínűleg nem sikerült megfelelő paraméterezést találnom, mivel nem kaptam jobb végeredményt. Végül az ORB-ot próbáltam ki, amivel már kielégítő eredményt kaptam, hasonlóan a SURF-fel végzett kísérletekhez, ellenben jóval kevesebb idő alatt, így ennél maradtam.

Miután minden képen megkerestem a jellegzetes pontokat, és kinyertem a hozzájuk tartozó leírókat, ezeket párosítanom kellett. Ehhez használhatunk brute-force módszert (minden mindenkel összehasonlítva és kiválasztva a legközelebbi), vagy a FLANN (Fast Approximate Nearest Neighbor Search Library [37]) könyvtárat, melyhez OpenCV-ben is elkészült egy interfész [38]. Ez első sorban arra használható, hogy gyorsan tudunk több dimenziós vektortérben egy vektorhoz a hozzá legközelebbi vektort megkeresni, amely SIFT és SURF esetben ideális, de használható bináris leírókhoz is [39]. Mivel a halmaz mérete, ahol a párosítást keressük, nem olyan nagy, hogy a brute-force módszer hátránya kiütközzön, ezért ezt választottam, melynek OpenCV-ben az implementációját a cv::BFMatcher osztályban találjuk.

Végül a kapott párosításokat a fundamentális mátrix segítségével validáltam és megszűrtettem, vagyis megnéztem, hogy a már használt epipoláris ($\mathbf{u}'^T \mathbf{F} \mathbf{u} = 0$) kényszer egy adott hibahatáron belül teljesült-e. Megjegyzem, hogy ettől még maradhatott teljesen rossz párosítás is a halmazban, hiszen ilyenkor az adott ponthoz csak azt ellenőriztük, hogy a másik pont rajta van-e a hozzá tartozó epipoláris egyenesen. Az 5.11. ábrán látható két kamera két képének a blobok által kijelölt részén a legközelebbi 20 találat.



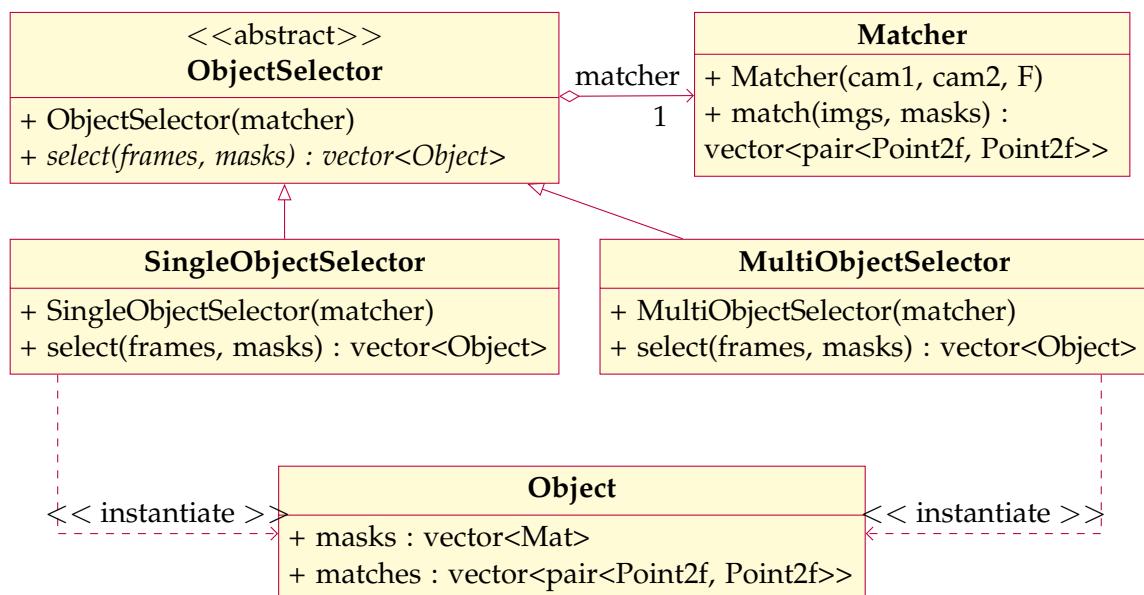
5.11. ábra. Blobok által kijelölt képrészleteken az egymásnak megfelelő pontok keresése.

A találatok szempontjából érdemi rész látható, a 20 legközelebbi találattal és ezekből 1-1 kiemelve közelről

Ezt követően a blobokat a következőképpen párosítottam. Vettem azt a képet, amelyiken több blob volt, mint a másikon (egyezség esetén a bal oldalit választottam), majd egyesével az itt lévő blobokhoz megkerestem, hogy a benne lévő párosításbeli pontok

párjainak többségét melyik – a másik képen lévő – blob tartalmazta. Így kaptam egy relációt a több elemű halmazból a kisebbe. Amelyik blobnak nem lett párja (nem tartalmazott *feature* pontot), azt a továbbiakban figyelmen kívül hagytam. Azokat viszont, amelyek ugyanazon blobhoz lettek társítva, egyesítettem. Így a végén a blobok egy részhalmazához egy kölcsönösen egyértelmű relációt, párosítást kaptam, ezek jelölték ugyanazon objektum két maszkját a két képen.

A következő lépések könnyebb tesztelése végett aktívan használtam a csak egy objektum detektálására képes algoritmust is, ezért a két megoldást egymással konform módon valósítottam meg. Mindkettőt egy közös *ObjectSelector* absztrakt osztályból származtattam, lásd 5.12. ábra. A továbbiakban fogjuk látni, hogy ennél a módszernél is jó, ha meghatározzuk az egymásnak megfelelő pontokat, így mindkettő megkappa konstruktorában a *Matcher* osztály egy példányát, melyet a *ObjectSelector* őriz. A *Matcher* két képkockából és az előtér maszkokból meghatározza az egymásnak megfelelő pontokat ORB-bal, melyeket pontpárok listájaként ad vissza. Konstruktorában azért, hogy a találatokat szűrni tudja, megkapja a kamera objektumokat és a fundamentális mátrixot. Mindkét konkrét *ObjectSelector* megvalósítás *Object* listával tér vissza, mely tartalmazza a két maszkot a két képen, valamint az összetartozó pontokat.



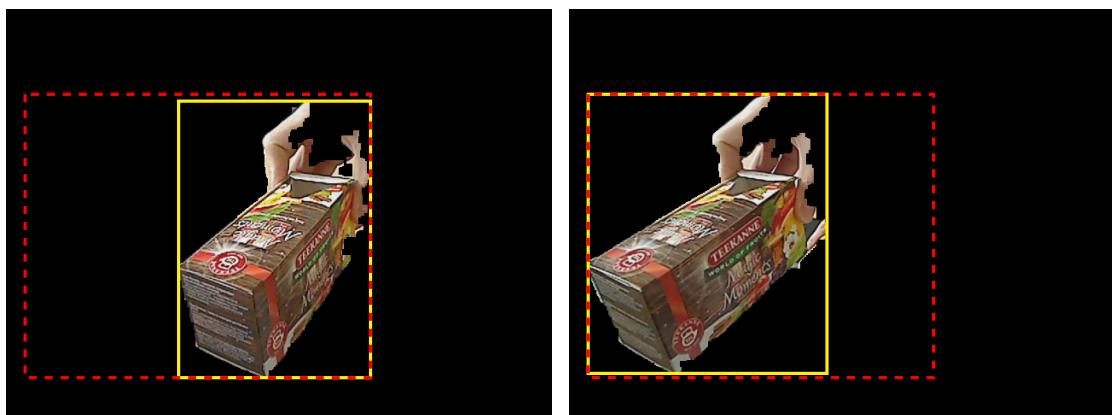
5.12. ábra. Objektumok kijelölését szolgáló osztályok

5.4. Optikai folyam meghatározása

Az 5.3.1. szekcióban bemutattam, hogyan lehet felhasználni a sűrű optika folyamokat előtér maszk meghatározására, a következőkben pedig azt írom le, hogyan lehet segítséggükkel a két kamera egy objektumhoz tartozó két képen sűrű pontmegfeleltetést számolni.

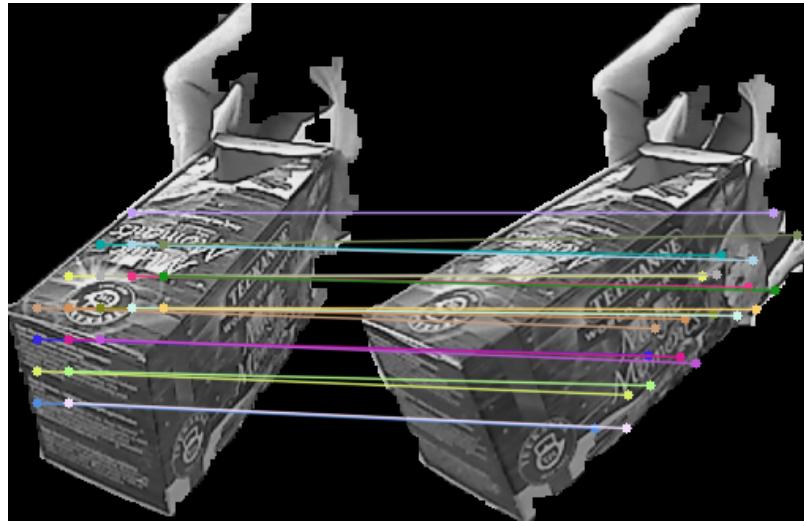
Az előzőekben már ismertettem a `cv::calcOpticalFlowFarneback()` függvényt, és annak néhány paraméterét, amelyeket úgy állítottam be, hogy a lehető leggyorsabban kapjak még használható becslést a mozgásról. Most viszont fontos, hogy a kapott eredmény pontosabb legyen, tehát ezeken kissé változtatnom kellett. Ezért 6 szintes piramist (`levels = 6`), és köztük 0,75-ös skálázást (`pyr_scale = 0.75`) állítottam be, hogy a nagyobb elmozdulásokat is detektálni tudjam. A Gauss-zaj feltételezése miatt bekapcsoltam, hogy egyszerű dobozsűrű helyett Gauss-szűrőt használjon az algoritmus, és az ehez kapcsolódó paramétereket a dokumentációban adott javaslatok alapján állítottam be, valamint emiatt az ablakméretet is növelteim (`winsize = 21`). A futási idő természetesen jelentősen függ a képek méretétől, melyekre az algoritmust futtatjuk, így fontos, hogy a már meghatározott objektum-maszkok segítségével a lehető legkisebbre vágjuk azokat.

A következőkben az 5.13. ábrán látható két képkocka lesz a kiinduló állapot, már az előbbiekben bemutatott objektum detektálás után, az előtér maszk által kijelölve. A képkockák két olyan kamera beállításal készültek, ahol a két kamera képsíkja nagyjából egybe estek (egy irányba néztek), és csak vízszintes irányban voltak egymáshoz képest eltolva.



5.13. ábra. Bal és jobb kamera által látott objektum kijelölve; a sárga keretek jelölik az objektumok befoglaló téglalapjait, a piros szaggatott pedig ezen téglalapokat tartalmazó legkisebb területű téglalapot

Első megközelítésem, hogy a két objektum befoglaló téglalapját tekintettem, és vettettem azt a téglalapot, mely a legkisebb területű azok közül, amely mindenkorral tartalmazza. Kivágva ezt a téglalapot a két képből, két egyforma méretű képrészletet kaptam, melyek külön-külön tartalmazták a teljes objektumot. Ez látható az 5.13. ábrán pirossal jelölve. Erre a két részletre számolva optikai folyamot, 7 646 darab vektort kaptam, melyek közül néhányat vizualizáltam az 5.14. ábrán. A vektorok kezdő és végpontjaiból alkottam pontpárokat, ezeket tekintettem egymásnak megfelelő pontoknak a két képen. Jól látható, hogy a kevés kirajzolt pontpárból egyik sem jó, mert a doboz különböző lapjaihoz tartoznak. Ez a nagy elmozdulás miatt van, hiába a piramis módszer, használhatatlan a végeredmény.



5.14. ábra. Első megközelítés (7 646 vektor)

A következő lépésem, hogy meghatároztam azt a vektort, amivel eltolva az egyik képet, az egymásnak megfelelő képpontok elmozdulásai a lehető legkisebbek lettek. Ehhez először szükségem volt néhány egymásnak megfelelő pontpárra.

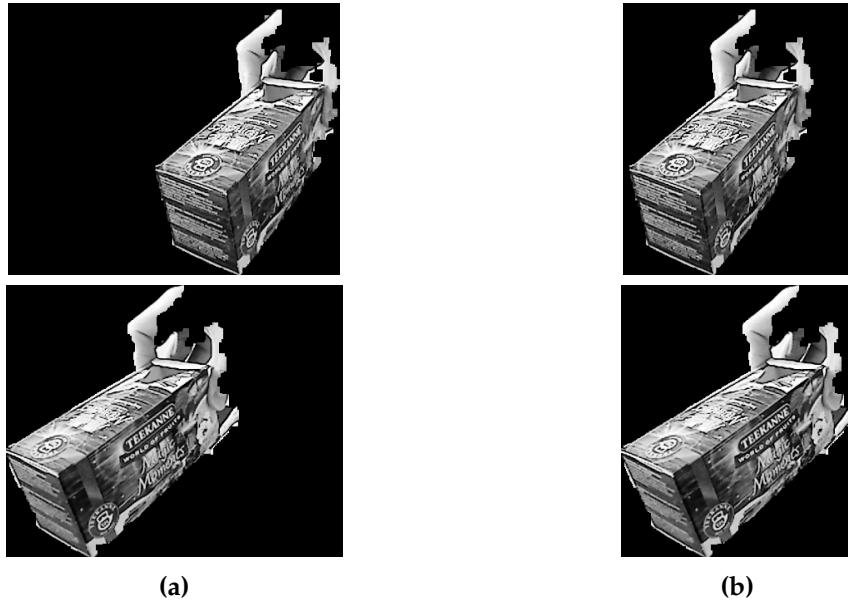
Az objektumok meghatározásánál (legyen az egy vagy több), már meghatároztam néhány egymásnak megfelelő pontot, így ezeket itt újra fel tudtam használni (`Object::matches`). A párosítások jelentette vektorokból kellett meghatároznom azt a vektort, amellyel az egyik képet eltolva az összetartozó pontok közötti távolság minimális. Azaz

$$\arg \min_{\mathbf{v}(x,y)} \sum_{i=1}^n |\mathbf{v} - \mathbf{d}_i|$$

ahol $\mathbf{d}_1(x_1, y_1), \mathbf{d}_2(x_2, y_2), \dots, \mathbf{d}_n(x_n, y_n)$ a párosításban szereplő pontok közti távolságvektorok. A problémát úgy is megfogalmazhatjuk, hogy a \mathbf{d}_i vektorokat pontoknak tekintjük és \mathbf{v} -t, a pontok geometriai mediánját keressük. Bizonyították [40], hogy ennek megoldásához nem létezik konkrét formula, vagy algoritmus csak aritmetikai műveletek és k. gyök felhasználásával. Ellenben konvex függvényről lévén szó, iteratíve közelíthetünk a megoldáshoz. Ezt elkerülendő, a távolság különbségek összege helyett közelítő megoldásként a távolság különbségek négyzetösszegét minimalizáltam, tehát:

$$\arg \min_{(x,y)} \sum_{i=1}^n \left((x - x_i)^2 + (y - y_i)^2 \right)$$

Ismeretes viszont, hogy a fenti a minimumhelyét az (x_i, y_i) pontok súlypontjában veszi fel. Az 5.15. ábrán látható az eltolás előtt és után az objektum helyzete a két képen.



5.15. ábra. Az objektum elhelyezkedése a kamerák képein (a) eredetileg és (b) a kiszámolt vektorral való eltolás után.

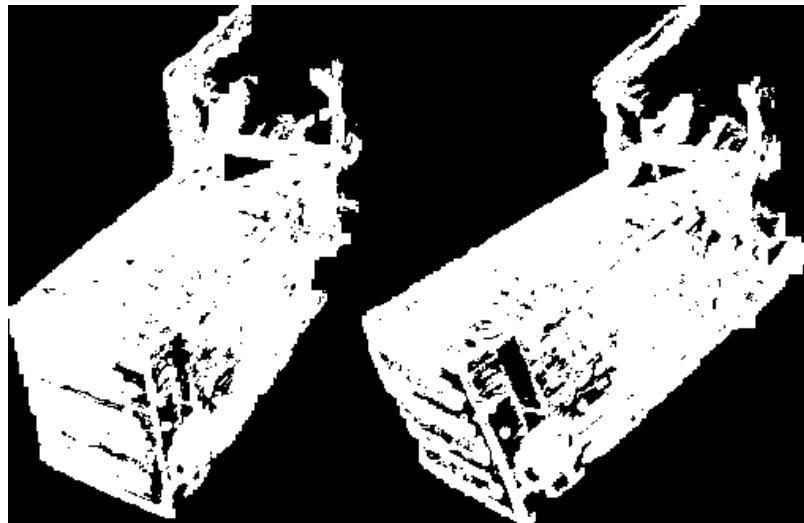
Optikai folyamok számítása esetén két jelenséget nem szabad figyelmen kívül hagynunk: egyik a textúrázatlanság, a másik pedig a kitakart pontok problémája. Ezekben az optikai folyam rektifikációja segíthet. You Yang és társai [41] egy bináris függvényt javasolnak a textúrázatlanság eldöntésére:

$$\zeta(\Omega_X) = \begin{cases} 0, & \text{ha } \sigma(I_{Y \in \Omega_X} - I_X) < \varepsilon_\Omega \\ 1, & \text{különben} \end{cases}$$

ahol Ω_X jelöli X képpont egy környezetét, I_X az X pont intenzitását, $\sigma(\cdot)$ a szórás-operátort egy halmazra nézve, valamint ε_Ω egy küszöbértéket, ami konstans. Úgy találták, hogy $\varepsilon_\Omega = 6$ választással jó eredményeket értek el, így én is ezt használtam. A mintaképek pontjaira kiszámolva ezt a függvényt, az 5.16. ábrán látható maszkokat kaptam.

A kitakart pontok kiszűrésére én Yang-ék megoldásától [41] eltérő megközelítést alkalmaztam. Legyen két képkocka K_1 és K_2 , valamint $F_{1,2} = \mathcal{F}(K_1, K_2)$ és $F_{2,1} = \mathcal{F}(K_2, K_1)$, ahol \mathcal{F} jelöli két képkocka közti optikai folyam operátort, melynek eredménye egy vektormező ($F_{1,2}, F_{2,1} : \mathbb{R}^2 \rightarrow \mathbb{R}^2$). Gondoljuk meg, hogy ha $x \in K_1$ és $x + F_{1,2}(x) = x' \in K_2$, akkor $x' + F_{2,1}(x') \approx x \in K_1$, vagyis ha egy x pont K_1 -ről K_2 -re az x' pontba mozog, akkor visszafelé nézve x' pontnak ideális esetben x pontba kell mozognia. Tehát oda-vissza számolva 1-1 optikai folyamot a kitakart pontokat kiszűrhetjük, hiszen a másik irányban nem fogjuk megtalálni a párosítást.

A fent leírtakat az `OpticalFlowCalculator` osztályban implementáltam (rövid áttekintő látható az 5.17. ábrán). Egy publikus metódusa van `calcDenseMatches()` néven,



5.16. ábra. ζ függvény alkalmazva az objektum összes pontjára

mely megkapja a két képkockát és egy objektumot, amelyre a sűrű pontmegfeleltetést szeretnénk számolni. Ez lényegében három dolgot csinál: meghatározza a textúrázott régiókat (calcTexturedRegion) a két képen, kiszámolja oda és vissza az optikai folyamokat (calcOpticalFlows), majd ezek alapján párosítja a pontokat (collectMatchingPoints).

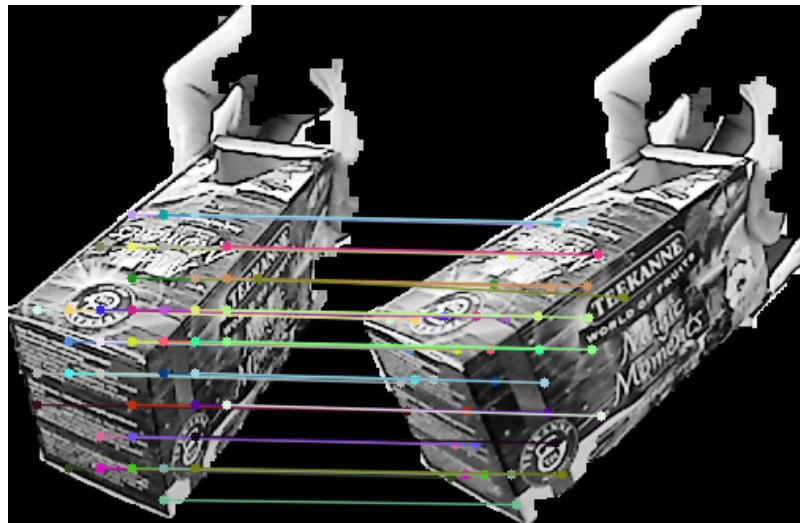
OpticalFlowCalculator
- frames : Mat[]
- masks : Mat[]
- texturedRegions : Mat[]
- calcTexturedRegion(frame, mask) : Mat
- calcOpticalFlows() : vector<Mat>
- collectMatchingPoints(flows) :
pair<vector<Point2f>, vector<Point2f>>
+ calcDenseMatches(frames, object) :
pair<vector<Point2f>, vector<Point2f>>

5.17. ábra. Sűrű pontmegfeleltetések optikai folyamok segítségével meghatározó *OpticalFlowCalculator* osztály

Az algoritmust lefuttatva egész pontos párosításokat kaptam, az 5.18. ábrán mutatok be ezek közül néhányat. Összesen 16 841 párt kaptam, tehát ennyit tudtam felhasználni a háromszögeléshez.

5.5. Háromszögelés

Ahogy a 3.3.3. alfejezetben bemutattam az elméleti hátteret, a következőkben ennek implementálását tárgyalom.



5.18. ábra. Végső párosítás az optikai folyamok segítségével (16 841 vektor)

5.5.1. OpenCV-s függvényekkel

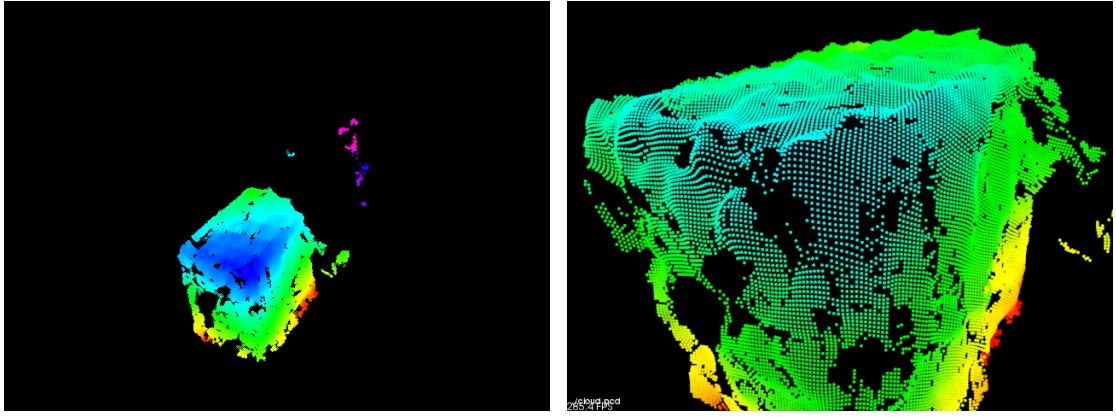
Az OpenCV `triangulatePoints` függvényében a *Linear-LS* (lineáris legkisebb négyzettek) módszer van implementálva. A dokumentáció [9] alapján sztereó-kalibráció során nyert projekciós mátrixokat vár a pontpárok mellett paraméterül. Az én esetben a kamerák nem voltak sztereó-kalibrálva, de a két projekciós mátrixszal rendelkeztem. Ennek ellenére a kimeneti 3D-s koordináták használhatatlanok voltak. Kis kutatás után a `cv::undistortPoints()` függvény meghívása jelentette a megoldást. Ez először normalizálta őket, vagyis:

$$\begin{bmatrix} u' \\ v' \\ 1 \end{bmatrix} = \underbrace{\begin{bmatrix} f_x & 0 & o_x \\ 0 & f_y & o_y \\ 0 & 0 & 1 \end{bmatrix}}_{\text{kamera-mátrix}}^{-1} \begin{bmatrix} u \\ v \\ 1 \end{bmatrix}$$

ahol (u', v') az (u, v) pont normalizáltja (kamera-mátrixtól független koordináták), majd a torzítási együtthatókat felhasználva az (u', v') pontoknak meghatározta a javított koordinátájukat. Ezután a `cv::triangulatePoints` függvénynek a projekciós mátrixok helyett csak az $(R | t)$ mátrixokat kellett átadnom. Fontos, hogy az előbbiekbén kialakított pontpárosítás azon osztályához tartozó pontokat, melyek az eltolt képhez tartoztak, az eltoláshoz használt vektor ellenetétjével vissza kellett tolni, hogy valós koordinátákat kapjunk.

Miután meghatároztam a pontok valóvilágbeli koordinátáikat, ezek pontosságát a képsíkokra történő visszavetítéssel vizsgáltam meg. A pontokat a `cv::projectPoints` függvénytel vettítettem a bal és jobb oldali kamera képére, majd ezek és az eredeti pontok távolságainak átlagát – átlagos visszavetítési hiba (*average reprojection error*) – vizsgáltam

képenként, majd vettet ezek maximumát. Ez az előzőekben mutatott bemenetre 8,687 pixel lett, amit elfogadható hibának vettet, hiszen azt jelenti, hogy minden pont átlagosan egy 3 pixel sugarú körön belül csúszott el. Az 5.19. ábrán látható az eredményeket bemutató két vizualizáció, melyeken a szín a pontok z koordinátáját jelzi.



(a) Bal oldali kamera nézőpontjából nézve **(b) PCL vizualizációs szoftverrel ráközelítve**

5.19. ábra. Háromszögelés Linear-LS-sel. Jól látható a közelí nézőpontból, hogy egy kissé hullámos lett a felület.

Az OpenCV keretrendszerben megtalálhatjuk a [21]-ben leírt optimális, de polinomiális algoritmust is implementálva `cv:::correctMatches()` néven, amely a fundamentális mátrix segítségével a 3.3.3. szakaszban említett módon javítja a pontpárokat, azaz $\mathbf{u} \leftrightarrow \mathbf{u}'$ összerendelések helyett olyan $\hat{\mathbf{u}} \leftrightarrow \hat{\mathbf{u}'}$ párokat ad, melyekre $d(\mathbf{u}, \hat{\mathbf{u}})^2 + d(\mathbf{u}', \hat{\mathbf{u}'})^2$ minimális, és teljesül, hogy $\hat{\mathbf{u}}'^T \mathbf{F} \hat{\mathbf{u}} = 0$. Ez az előbbi átlagos visszavetítési hibát 8,054 pixelre javította, de rohamos teljesítménycsökkenés mellett (0,06 másodperces futási idő helyett 0,64 másodperc lett a háromszögelés), mely az eredményeket szabad szemmel megnézve nem volt meggyőző.

5.5.2. Iteratív lineáris legkisebb négyzetek (*Iterative-LS*)

A 3.3.3. szakaszban leírt iteratív módszert nem tudjuk alkalmazni a fenti `cv::triangulatePoints` függvénytel, mert nem lehet az egyenleteket pontonként külön-külön súlyozni. Ennek megfelelően először a „szimpla” *Linear-LS* megközelítést kellett implementálnom, majd ezt már meghívhattam iteratívan különböző súlyokkal.

Ehhez én Roy Shilkrot online elérhető alkalmazás-könyvtárából [42] merítettem a kiindulási alapot, és azt alakítottam az általam használt adatszerkezetekhez. Ehhez is előtte minden pontot normalizálni, valamint a torzítási együtthatóknak megfelelően korrigálni kellett. Ezzel a megközelítéssel 8,21 pixelnyi átlagos visszavetítési hibát kaptam, de a sebesség harmadára csökkent (0,06 másodperc helyett 0,18 másodperc lett a futási idő).

A fentieket összegyűjtve a Triangulator osztályban (lásd 5.20. ábra) implementáltam, melynek két metódusa a fent említett két módszert (OpenCV-s triangulatePoints, valamint az Iterative-LS) valósítja meg. Mindkettő az egymásnak megfelelő pontpáro-

kat várja bemenetként, és harmadik paraméterében visszaadja CloudPoint-ok listájákat a pontfelhőt, valamint visszatérési értékként az átlagos visszavetítési hibát. Egy CloudPoint az aktuális koordinátákon kívül azt is tudja magáról, hogy mekkora a hozzá tartozó visszavetítési hiba, így később a megjelenítésnél ezt is figyelembe tudtam venni.

Triangulator	CloudPoint
<ul style="list-style-type: none"> + camera1 : Camera + camera2 : Camera + cameraPose1 : CameraPose + cameraPose2 : CameraPose <ul style="list-style-type: none"> + triangulateIteratively(points1, points2, cloudpoints) : double + triangulateCv(points1, points2, cloudpoints) : double 	<ul style="list-style-type: none"> + pt : Point3d + reprojErr : double

5.20. ábra. *Triangulator* osztály és a *CloudPoint* struktúra

5.6. Változtatható nézőpont

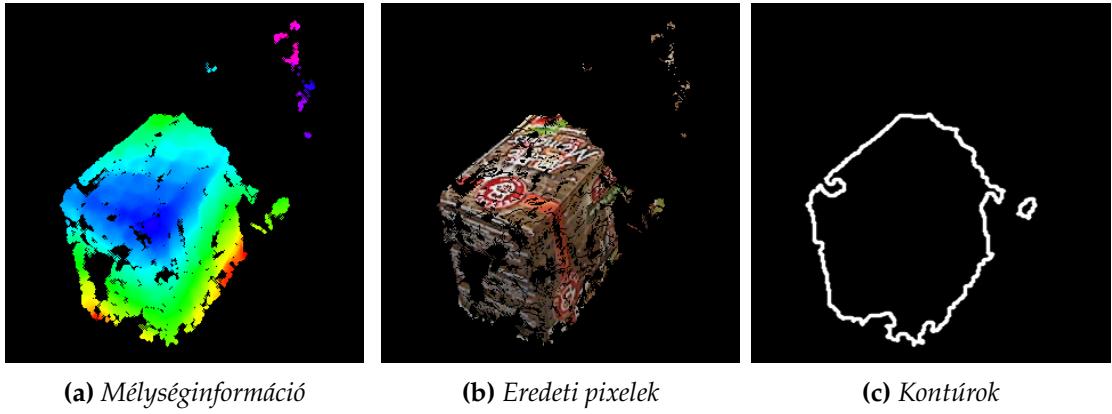
Ahogy az előbbiekben (lásd 5.19a. ábra) már említettem, a `cv::projectPoints` függvény segítségével lehet meghatározni, hogy adott kamerából fényképezve a valóvilágbeli pontoknak hova esnek a vetületei. Háromfélé megjelenítési technikát implementáltam; 1) z koordinátát jelölő színkód, 2) eredeti képpontok színeinek felhasználása, valamint a 3) kontúrok rajzolása.

Az első implementálását a HSV (hue, saturation, value) színtér segítségével valósítottam meg. *S* és *V* értékét fixen a maximumra állítottam, *H* értékét pedig egyenletesen elosztottam a különböző z koordináták mentén.

A másodikhoz a pontfelhő mellé még két információra volt szükség: a pontok koordinátáira a bal képen, valamint magára a bal képre, hogy a pixelek színeit ki tudjam nyerni. Ezek után az adott 3D-s pont vetített képpontjának a színét a bal képen lévő forrás pixel színére állítottam.

A harmadik megoldást három lépésben valósítottam meg. Először fehér pontokként levetítettem a pontokat a képsíkra. Ezt követően dilatáció-erózió kombinációval morfológiai zárást hajtottam végre a bináris képen, minek köszönhetően a kisebb lyukak és szakadások megszűntek. Végül az eredményen kontúrokat kerestem a `cv::findContours()` függvény segítségével, és ezeket kirajzoltam. Egy határérték (amit 100 pixel²-nek választottam kézi hangolás után) alatti területtel rendelkező kontúrokat elvetettem, mert ezek olyan kisebb foltokat jelentenek, melyeket nem tudunk egyértelműen egy nagyobb objektumhoz rendelni.

Ezen három vizualizáció eredménye a bal oldali kamera nézpontjából látható az 5.21. ábrán.



5.21. ábra. Különböző vizualizációk, ráközelítve a hasznos területet

A 4. fejezetben leírtam, hogy a választott nézőpont, ahonnan érdemes lehet rekonstruálni az objektumot, a két kamera nézőpont között helyezkedhet el. Tehát a két kamera között kellett interpolálni a virtuális kamera helyzetét és irányát, és innen elvégezni a vetítést. Fontos, hogy nem csak a kamera helyzetét (\mathbf{t}_v), a forgatási mátrixát (\mathbf{R}_v) is meg kellett határoznom. Jelölje $r \in [0; 1]$ a virtuális kamera pozícióját a pályáján, ahol $r = 0$, ha a bal oldali kamera, és $r = 1$, ha a jobb oldali kamera helyzetében van. Ekkor a virtuális kamera eltolási vektora $\mathbf{t}_v = (1 - r)\mathbf{t}_b + r\mathbf{t}_j$, ahol \mathbf{t}_b jelöli a bal oldali, \mathbf{t}_j a jobb oldali kamera eltolási vektorát. Ken Shoemake cikke [43] alapján célszerű a forgatási mátrixok (vagy akár az ebből nyerhető Euler-szögek) helyett kvaterniókra áttérni, és ezek között úgynevezett gömbi lineáris interpolációval (*slerp*) kiszámolni a kívánt köztes lépést. Ennek implementációját az Eigen [44] alkalmazáskönyvtárral valósítottam meg. Először az OpenCV-s `cv::Mat` mátrix objektumokat Eigen-es `Eigen::Matrix` objektumokká konvertáltam, majd ezeket már az API-t használva kvaterniókká alakítva a `slerp()` metódus segítségével egy köztes kvaterniót interpoláltam. Végül ezt visszaalakítottam az \mathbf{R}_v OpenCV-s mátrixszá.

Ezt a logikát a *Visualization* osztályban implementáltam, mely konstruktorában egy *CameraPose* példányt (eltolási és forgatási vektor – lásd Rodrigues-formula), valamint egy kamera-mátrixot vár. Négy további metódusa van, ebből három a fenti megjelenítési típusokat valósítja meg, az utolsó pedig visszaadja magát az elkészült képet. Az áttekintő osztálydiagramot az 5.22. ábra mutatja be. Az r arányszámot a felületről egy csúszka segítségével tudjuk változtatni, az 5.23. ábra (lásd 52. oldal) mutatja a két szélső és a középső helyzetben a vizualizáció eredményét.

5.7. Teljes folyamat együtt, implementációk kiválasztása

Az előző szekciókban felvázolt lépések nél három helyen készítettem két különböző implementációt, ezek között az alábbiak szerint döntöttem. Az előtér maszk meghatározásához végül az *OFForegroundMaskCalculator* választottam. Ugyan tovább tart, mint a párja, viszont sokkal pontosabb maszkot eredményez, ami jobb rekonstrukciót tesz lehe-

Visualization
- cameraPose : CameraPose
- cameraMatrix : Mat
- result : Mat
+ Visualization(cameraPose, cameraMatrix)
+ renderWithDepth(points) : void
+ renderWithContours(points) : void
+ renderWithColors(points, imgPoints, img) : void
+ getResult() : Mat

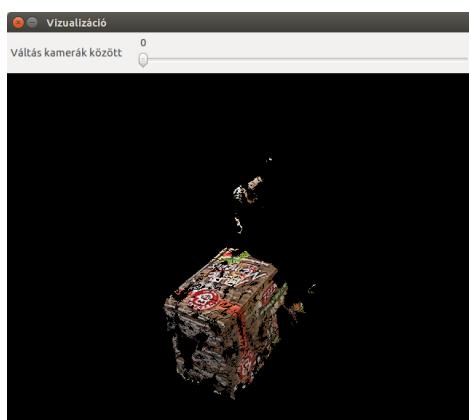
5.22. ábra. Vizualizációs osztály

tővé. A MOG2ForegroundMaskCalculator által kiszámolt előterek nagyon hiányosak voltak, és nem sikerült a paraméterezését úgy módosítanom, hogy ez javuljon. Az objektum-detektáláshoz természetesen a MultiObjectSelector-t választottam, hiszen a másik csak tesztelés miatt került implementálásra, hogy két képkockáról gyorsan kapjak egy objektumot, amire a további lépéseket végrehajthatom. A Triangulator osztály esetén pedig az OpenCV-s metódus mellett döntöttem (triangulateCv), mert csak kicsit pontatlanabb, ellenben természetéből adódóan jóval gyorsabb (átlagosan 5-ször), mint az iteratív megoldás.

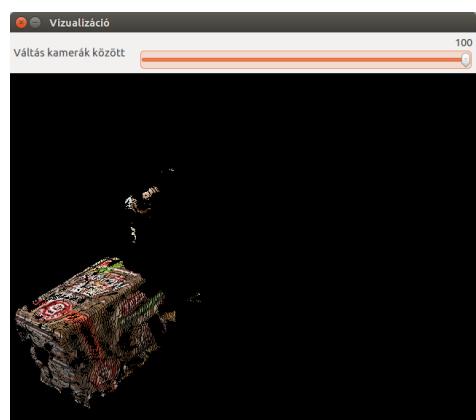
Egy képkockapár feldolgozásának a folyamata szekvencia-diagramként látható az 5.24. és az 5.25. ábrákon. Az első mutatja be a képkockák lekérésétől az objektumok meghatározásáig a folyamatot, a második pedig az objektumok rekonstruálását és vizualizációját.

5.8. Összefoglaló

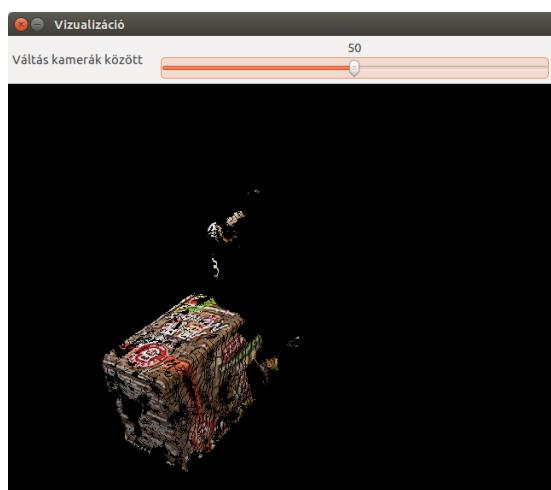
Ebben a fejezetben az elkészült alkalmazás implementációjának fejlesztését mutattam be. Az egyes fázisokat, részproblémák megoldásait példákon keresztül szemléltettem. Amennyiben egy problémára több megoldást is elkészítettem, indokoltam a választásom valamelyik megközelítés mellett. Végezetül a videofolyamok képkockáinak feldolgozását szekvencia diagramokon ábrázoltam.



(a) Bal oldali kamera képe és a helyreállítás
ugyanebből a nézőpontból

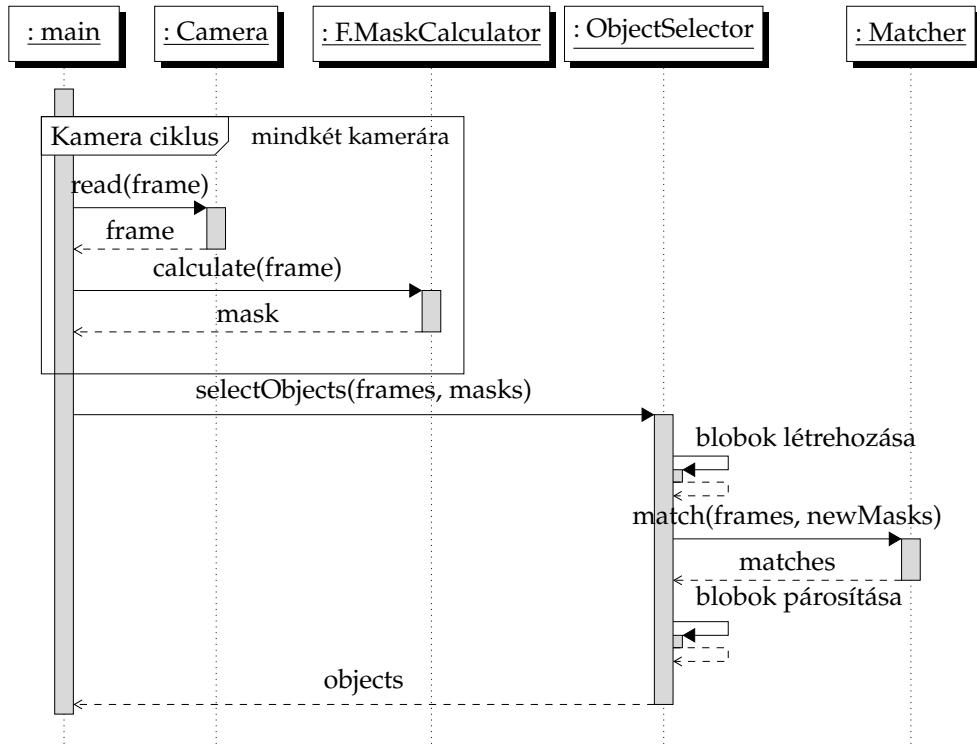


(b) Jobb oldali kamera képe és a helyreállítás
ugyanebből a nézőpontból

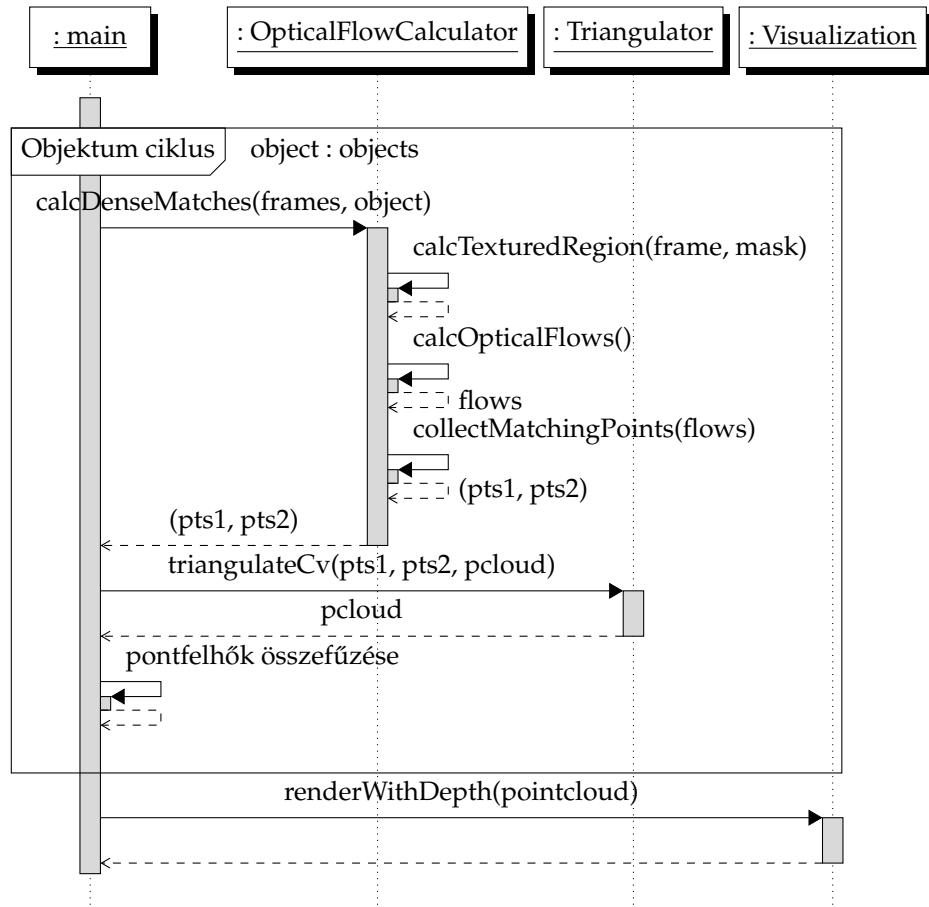


(c) Helyreállítás a két kamera közötti, interpolált nézőpontból

5.23. ábra. Választott nézőpont (a) bal kamera, (b) a jobb kamera és (c) a két kamera közötti pozícióban



5.24. ábra. Egy képkocka-pár esetén az objektumok kijelöléséhez szükséges lépések szekvencia diagramja



5.25. ábra. Objektumonkénti rekonstrukció és vizualizáció szekvencia diagramja

6. fejezet

Helyreállítás tesztelése

Ebben a fejezetben három jelenetben teszteltem a helyreállítás minőségét. Az elsőben a két kamerát egymás mellé helyezem úgy, hogy képsíkjaik nagyjából egybeesszenek, a második jelenetnél a két kamera optikai tengelye egy hegyes szöget zár be, de még elég közel vannak egymáshoz, míg a harmadik egy valóshoz közel körülményt szimulál, amikor a két kamera a szoba két sarkában van elhelyezve és a helyiség közepét veszik két oldalról.

A teszteléshez két darab Logitech QuickCam Pro 9000 típusú webkamerát használtam, melyektől VGA felbontású (640×480) képeket kértem le.

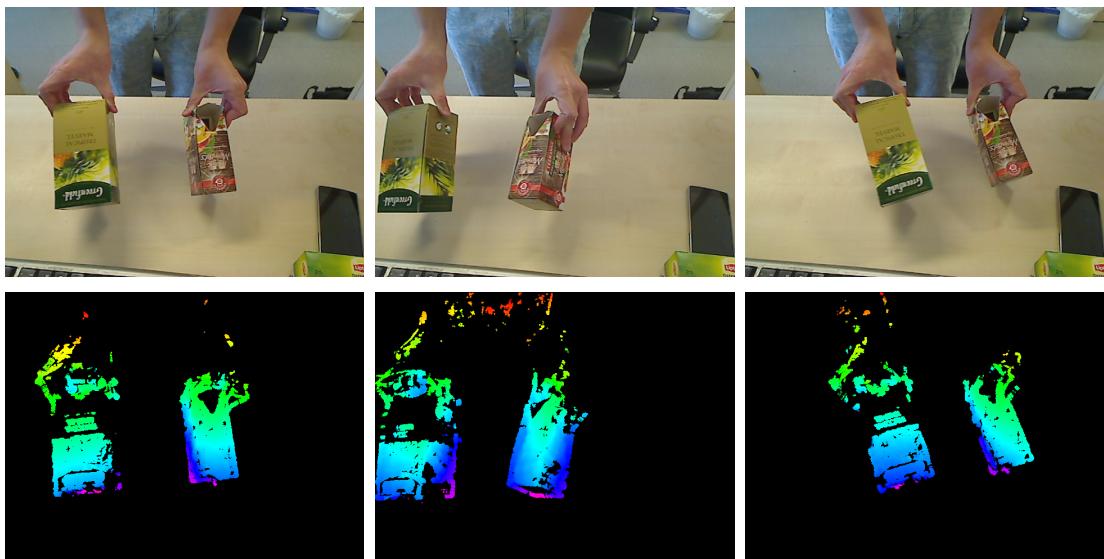
6.1. Első jelenet

Elsőként egy olyan jelenetben próbáltam ki az elkészült alkalmazást, amely esetén a két kamera egymás mellett van (lásd 6.1. ábra), egy irányba néznek, és két teásdobozt mozgatok előttük. A jelenet 178 képkockából állt (~6 másodperc), melyből mindegyik 640×480 -as felbontású, színes kép. A 6.2. ábrán látható a bal oldali kamerák által rögzített három képkocka és az ezekhez tartozó, – a könnyebb összehasonlíthatóság végett – ugyanezen nézőpontból vett helyreállítások. Megfigyelhető, hogy a bal oldali teásdoboz rekonstrukciója a textúrázatlan területeken várakozásainknak megfelelően hiányos, hiszen itt nem hagyatkozhattunk az optikai folyam által adódó elmozdulásokra. Ugyanezen okok miatt a szereplő kezeiből is csak néhány pontot lehetett helyreállítani.

A 178 képkockából 157-szer kaptam a 6.2. ábrához hasonló rekonstrukciókat, 13-szor csak 1 objektumot sikerült helyreállítani, illetve 8-szor volt értékelhetetlen a vég-eredmény (nagyon kicsi pontfelhő, néhány ponttal). A hibák akkor történtek, amikor a jelenetben a két doboz nem mozgott jelentősen, csak helyben forgott, minek következtében az előtér maszkok hiányosak lettek. A rekonstrukciók során az összes képkockára nézve az átlagos visszavetítési hiba 0,8 pixel lett, ami elhanyagolhatónak számít. A már bemutatott pillanatokhoz rajzolt kontúrok a 6.3. ábrán látható. Figyeljük meg, hogy a



6.1. ábra. Kamerák helyzete az első jelenetnél



6.2. ábra. A bal oldali kamera 3 képkockája (40., 100. és 170. képkocka) az első jelenetből, valamint a helyreállított képek a bal oldali kamera nézőpontjából

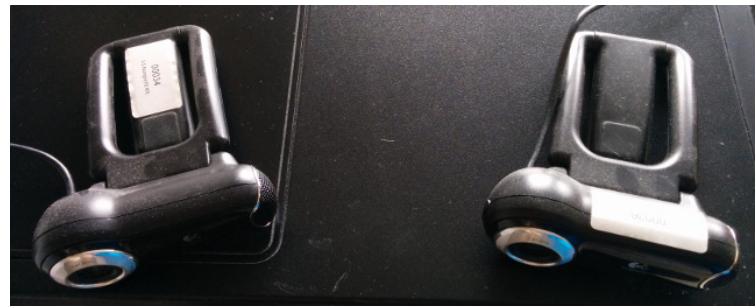


6.3. ábra. A 40., 100. és 170. képkockák alapján számolt kontúrok a bal oldali kamera nézőpontjából

jobb oldali dobozt jól meghatározza annak kontúrja, viszont a bal oldali doboz textúrázottságának hiánya jelentősen befolyásolja a kontúrjainak értelmezhetőségét.

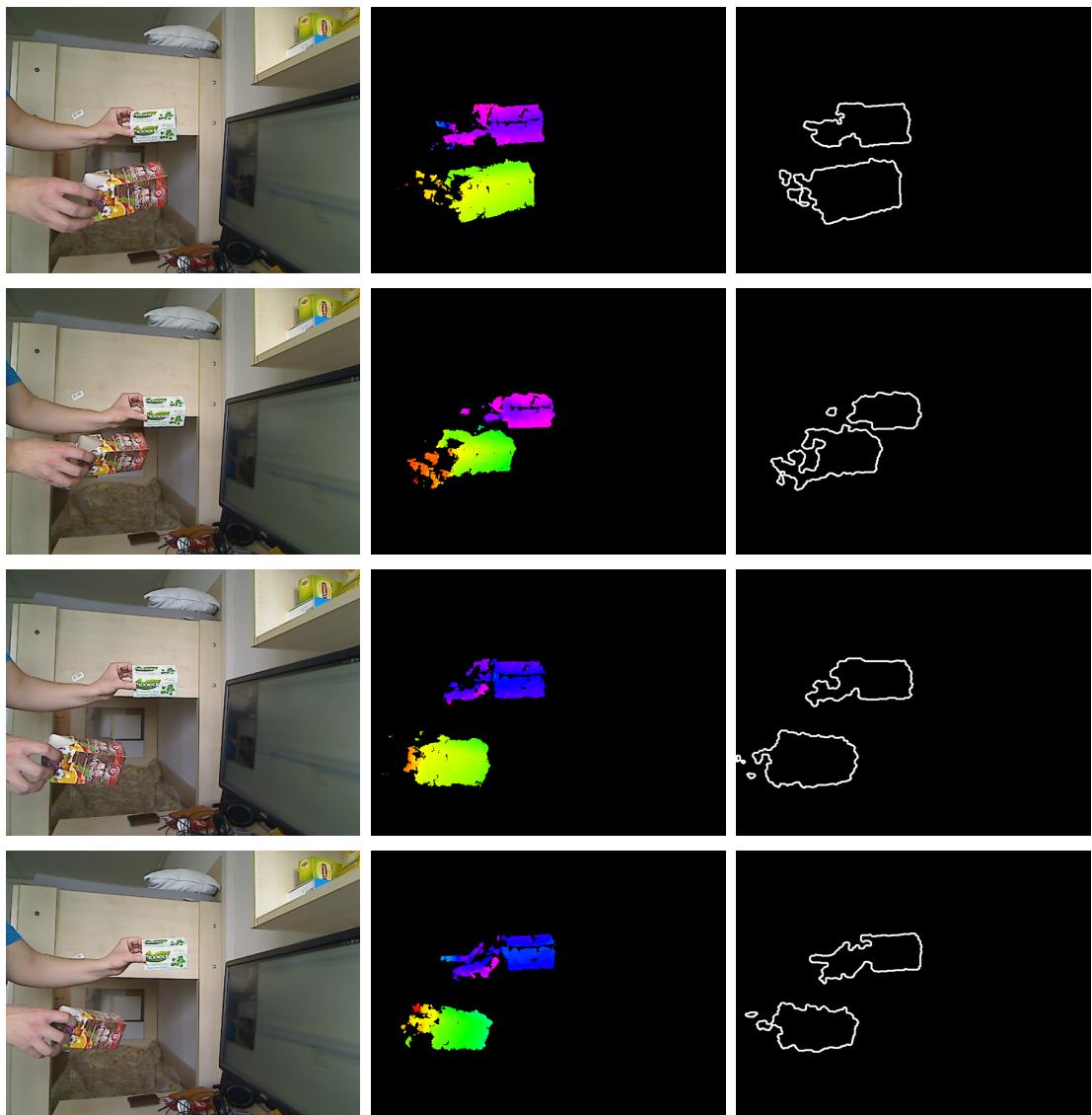
6.2. Második jelenet

Második jelenet két olyan kamera által került rögzítésre, amelyek optikai tengelyei egy hegyes szöveg zártak be (lásd 6.4. ábra). Ebben az esetben is két teásdobozt mozgattam.

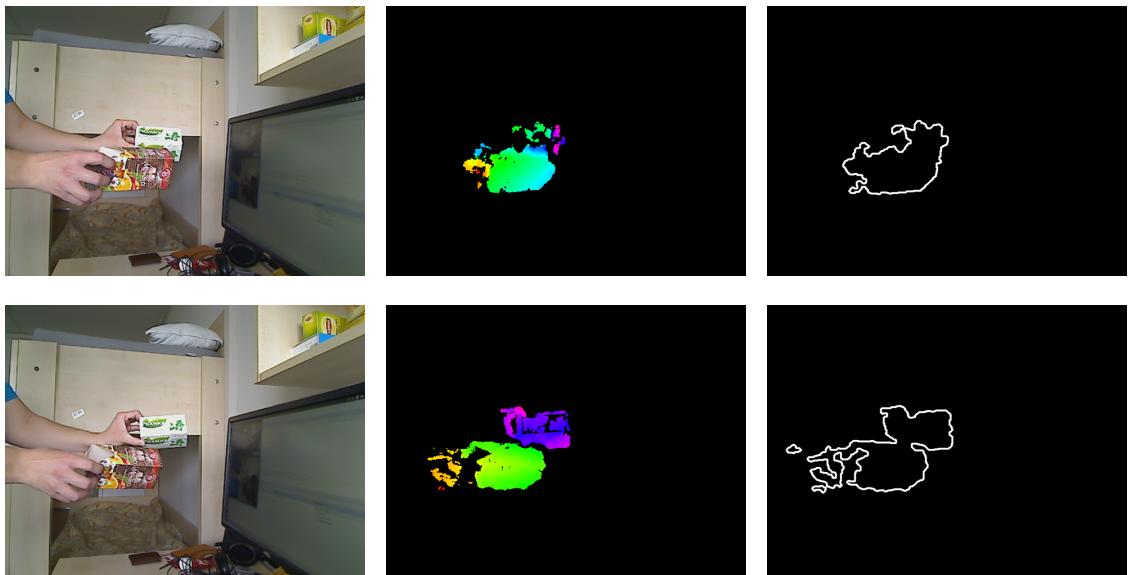


6.4. ábra. Kamerák helyzete a második jelenetnél

A jelenet 360 képkockából állt (~12 másodperc), melyből mindegyik 640×480 -as felbontású, színes kép. A 6.5. ábrán látható a bal oldali kamerák által rögzített 4 képkocka és ugyanezen nézőpontból vett helyreállítások vizualizációi.



6.5. ábra. A második jelenethez tartozó 45., 130., 215. és 339. képkockák bal oldali képei és azok vizualizációi



6.6. ábra. Amikor a két doboz közel van egymáshoz, kiemelve a részleteket

A képek jól mutatják, hogy elég sok pontot sikerült helyreállítani köszönhetően a jó textúrázott dobozoknak, a lassú mozgásnak és a megfelelő fényviszonyoknak. Az átlagos visszavetítési hiba ennél a jelenetnél 1,5 pixel lett.

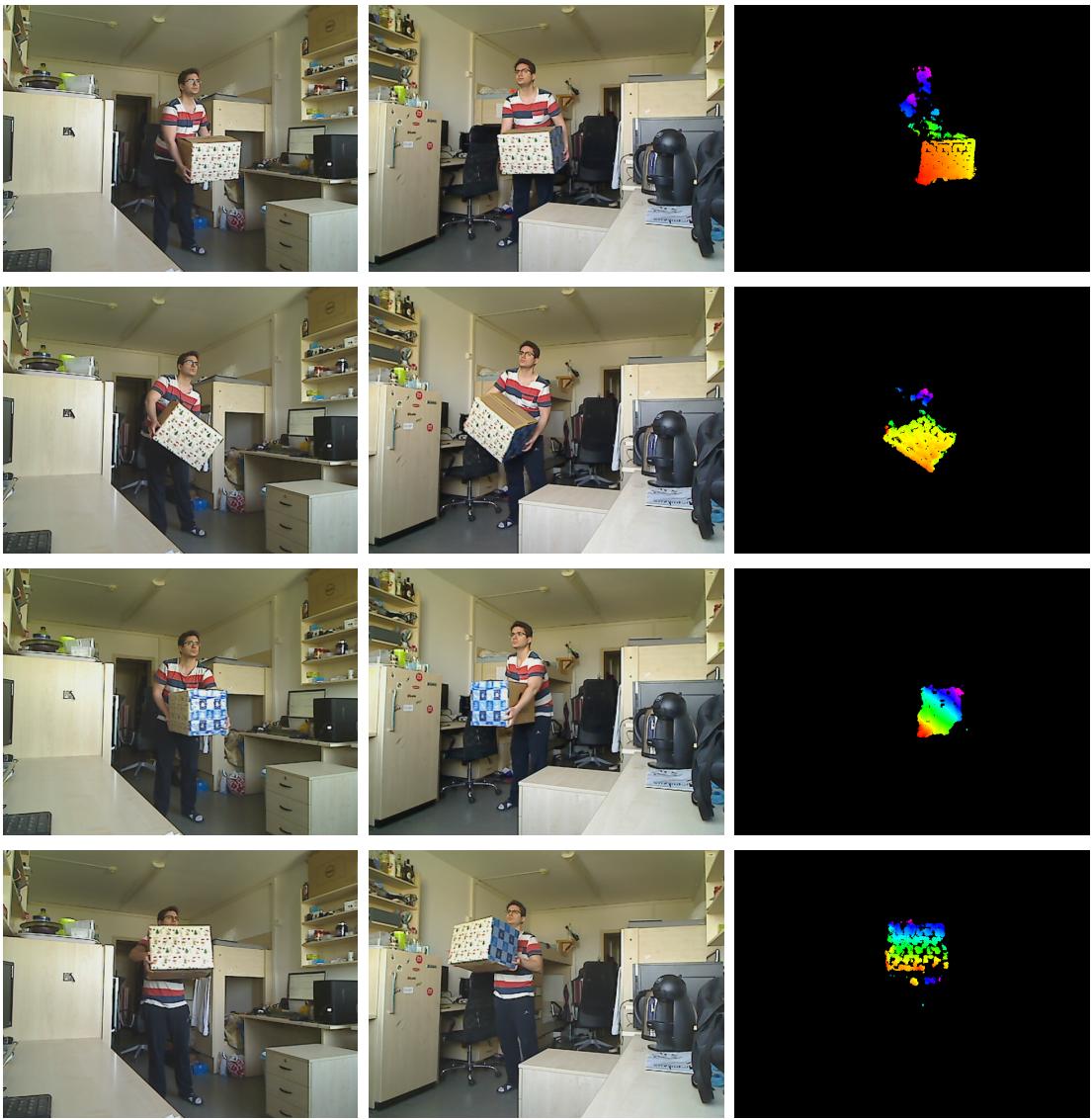
A 6.6. ábrán látható két olyan helyzet, amikor a két doboz a képeken nagyon közel volt egymáshoz. Az első esetben a hátul lévő dobozt szinte teljesen elveszítettük, míg a második esetben jól elkülönült térben az eredmény. Ekkor viszont a képen látható közelég miatt az előtér maszkok összemosódtak, így logikailag egy objektumként értelmezük őket, amit a kontúrrajzon is megfigyelhetünk.

6.3. Harmadik jelenet

A harmadik jeleneten egy valós helyzethez igen közeli kamerabeállítást használtam (lásd 6.7. ábra), amikor egy megfigyelt terület két sarkába helyeztem a (megfigyelő) kamerákat. A jelenet során egy nagyobb textúrázott dobozt mozgattam és 500 képkockából ál-



6.7. ábra. A harmadik jelenetnél használt kamera elrendezés (bal és jobb kamera a párkány szélén)



6.8. ábra. A második jelenethez tartozó 52., 96., 276. és 368. képkockák bal oldali képei és azok vizualizációi

ló videót készítettem (~16,5 másodperc), melyből az eddigiekhez hasonlóan mindegyik 640×480 -as felbontású, színes kép volt. A 6.8. ábrán látható a két kamera által rögzített kép és a bal oldali kamera nézőpontjából helyreállított kép vizualizációja.

Ennél a jelenetnél is számos alkalommal (223 képkocka esetén) jól használható rekonstrukciókat kaptam, kivéve amikor megcsillant a napfény a fényes felületen, a mozgás túl hirtelen volt, így a kép elmosódott, vagy a doboz textúrázott oldala az egyik képen már túl hegyes szögben látszódott. Az átlagos visszavetítési hibára ezen képkockák esetén 2,56 pixel adódott.

6.4. Összefoglaló

Ebben a fejezetben három jelenet segítségével teszteltem a helyreállítást. Az eredmények alapján a tesztelést sikeresnek tekintettem. A tapasztalatok alapján, amennyiben eleget teszünk azon követelményeknek, miszerint az objektumok mozogjanak a képeken és a felületük textúrája ne legyen nagy felületeken egybefüggően homogén, akkor igen pontos és jól felismerhető rekonstrukciókat kapunk az elkészült megoldás segítségével. Az utolsó jelenet eredményei azt is igazolják, hogy egy szoba közepén játszódó eseménysorozat, annak két sarkából rögzített videófolyamok alapján a két nézőpont között szintén jó minőségben helyreállítható a dolgozatban leírt és megvalósított módszerrel.

7. fejezet

Gyorsítási lehetőségek

Ebben a fejezetben az előzőekben bemutatott első jelenet képsorait fogom felhasználni egy kezdeti méréshez, majd utána megvizsgálom azon lehetőségeket, amelyekkel gyorsítni lehet az elkészült implementáció működését. Először a párhuzamosítás lehetőségeit vizsgálom meg, majd kitérek a GPU-n történő számítási módszerekre, végül a bemeneti képek méreteinek változtatásának hatását vizsgálom meg a teljesítményre.

7.1. Első mérések

A teszteket egy olyan számítógépen végeztem, amiben egy Intel® Core™ i5-4440 processzor, NVIDIA GeForce GTX 750 Ti típusú videókártya és 8GB memória volt, a kódot pedig ezen futó Linux operációs rendszerre fordítottam, majd futtattam.

A következő mérésekhez az előző fejezetben bemutatott első jelenetet (tehát, amikor a kamerák képsíkjai nagyjából egybeesnek) használtam fel. A jelenethez tartozó, már említett fontosabb adatok: 178 képkocka (~6 másodperc), mindegyik 640 × 480-as felbontású, színes kép, ezeken pedig két mozgó teásdoboz. Az alkalmazást lefuttatva erre a jelenetre a 7.1. táblázaiban látható időket kaptam, lebontva a fontosabb lépésekre sorrendben, majd a teljes folyamatra is nézve. A legtöbb időt az optikai folyam meghatározása vitte el, utána az előtér maszkok, majd a textúrázottság meghatározása következett. Az átlagos visszavetítési hiba 0,806 pixelnyi lett az összes képkockára nézve. A teljes folyamat átlaga alapján 1 képkockapár rekonstrukciója majdnem 1 másodperc volt, azaz kb. 1,1 FPS sebességgel lehet rekonstruálni ezzel a megközelítéssel, ezen a hardveren, ilyen jellegű jelenet esetén.

Tevékenység képkockánként	Eltöltött idő (s)			
	min	max	átlag	szórás
Előtér maszk (2 kamerára)	0,039	0,229	0,178	0,0122
Pontpárosítások ORB-bal	0,0194	0,0629	0,0235	0,00474
Objektumok párosítása	0,0362	0,146	0,0583	0,0101
Textúrázottság meghatározása	0,0391	0,251	0,163	0,043
Optikai folyam oda-vissza	0,149	0,6	0,427	0,103
Sűrű pontmegfeleltetések	0,00222	0,0173	0,0119	0,00282
Háromszögelés	0,0008	0,0855	0,0587	0,0147
Vizualizáció	0,00226	0,0162	0,00845	0,00165
Teljes folyamat	0,0695	1,16	0,908	0,17

7.1. táblázat. Első jelenet esetén az egyes lépések futási idejükhez kapcsolódó statisztikái (178 képkocka)

7.2. Párhuzamosítás, újra-kalkulált eredmények

Az előző szekcióban leírtak alapján látható, hogy szekvenciális végrehajtás esetén mekkora sebességet várhatunk az említett hardveren. A következőkben a párhuzamosításban rejlő lehetőségeket vizsgáltam meg.

Különböző platformokon különböző megoldásokat találhatunk többszálú feladatvégrehajtásra. C++-ban az egyik legelterjedtebb az OpenMP API [45, 46]. Felhasználásával platform-független, megosztott-memóriás párhuzamos programokat írhatunk. Nagy előnye, hogy ha a cél nem támogatja (legyen az a fordító, vagy a platform), akkor az API megfelelő alkalmazása esetén az alkalmazás ugyanúgy fordítható és futtatható, azzal a különbséggel, hogy az elkészült bináris csak egy szálon hajtja végre az utasításokat. A következőkben röviden áttekintem az általam használt OpenMP funkciókat.

A többszálúság kezelése deklaratív alapon történik a `#pragma omp` utasításokkal. Az első `#pragma omp parallel` direktívánál egy fix számú szál-csapat jön létre (ami általában a CPU magjainak számától függ, de `num_threads(N)` attribútummal ez kézzel is állítható), amely a program futása alatt konstans méretű. Ennek az az indoka, hogy a szálak létrehozása költséges folyamat, míg munkába állításuk nem.

A legegyszerűbb párhuzamosítási lehetőség, hogy az egymástól független lépések, amelyek ugyanazon algoritmusokat hajtják végre eltérő bemeneteken (pl. két kamera két képéből külön-külön egy előtér meghatározása, textúrázottság meghatározása) `for`-ciklusokba szervezzük, és ezek iterációit külön szálakon hajtjuk végre a `#pragma omp for` utasítással.

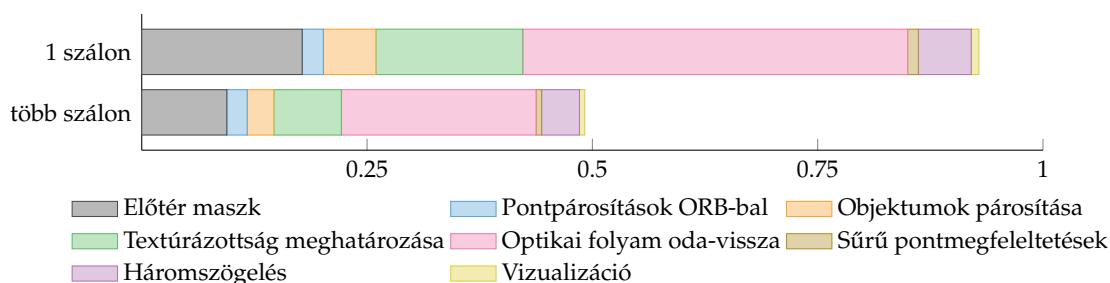
Az OpenCV (illetve az általa használt hardverillesztő) a kamerák képeit buffereli, így ha az adott képkockákat nem tudjuk elég gyorsan lekérni, akkor azok „beragadnak”, a későbbiek (amik nem férnek be a várakozási sorba) pedig elvesznek. Ha ~30 FPS sebességgel rögzíti a képeket a kamera, akkor nagyjából 33ms időnk van egy képkocka feldolgozására. Mivel ezt nem sikerült elérni, így a következő ötletet alkalmaztam. Annak érdekében, hogy mindenkor a képkockát dolgozzam fel, amit éppen aktuálisan a kamera

rögzít, a képek lekérését egy szálon folyamatosan végzem, és amikor éppen nem fut egy feldolgozás, akkor indítok egyet az aktuális képkockával. Ennek köszönhetően ugyan a feldolgozás nem folytonos, kimaradnak képkockák, viszont az adott rekonstrukció a jelenlegi időponthoz nagyon közelí. A `#pragma omp task` direktíva segítségével szerveztem a rekonstrukciót OpenMP feladattá, amely tulajdonképpen egy aszinkron végrehajtást jelent.

Ezekkel az apró javításokkal ellátva az alkalmazás kódját, a 7.2. táblázatban látható futási időket mértem az első jelenetre (itt még offline történt a feldolgozás, nem maradt ki egy képkocka sem). Az egyszálú végrehajtással összehasonlított átlagos idők a 7.1. ábrán láthatóak. Megfigyelhető, hogy azon műveletek, amiket lehetett párhuzamosítani (pl. két képen ugyanazon művelet, két optikai folyam számolás), nagyjából kétszer gyorsabban fejeződtek be, a teljes folyamat pedig több, mint kétszeresére gyorsult. Ez annak volt köszönhető, hogy az objektumok rekonstrukcióját is párhuzamosítottam, valamint ebből következett az is, hogy egy teljes képkocka átlagos rekonstrukciója rövidebb ideig tartott, mint a részfeladatok átlagainak szummája. Természetesen nem lehet a párhuzamosítással a végletekig minden gyorsítani, a rendelkezésre álló feldolgozóegységek korlátot szabnak az ésszerűen egymás mellett futtatható szálak számára, amely az én esetben négy szál volt (ez pont ideális átlagosan két objektum/jelenet esetén).

Tevékenység képkockánként	Eltöltött idő (s)			
	min	max	átlag	szórás
Előtér maszk (2 kamerára)	0,0875	0,103	0,0944	0,00261
Pontpárosítások ORB-bal	0,0122	0,0368	0,0226	0,00347
Objektumok párosítása	0,0206	0,0441	0,0296	0,00344
Textúrázottság meghatározása	0,0208	0,11	0,0749	0,0153
Optikai folyam oda-vissza	0,0731	0,312	0,216	0,0489
Sűrű pontmegfeleltetések	0,000751	0,0122	0,00618	0,00147
Háromszögelés	0,000567	0,058	0,042	0,0103
Vizualizáció	0,00065	0,00838	0,0059	0,00138
Teljes folyamat	0,109	0,574	0,391	0,105

7.2. táblázat. Többszálú végrehajtás esetén az első jelenet feldolgozása során az egyes lépések futási idejükhez kapcsolódó statisztikái (178 képkocka)



7.1. ábra. Első jelenet esetén az egyes lépések képkockánti átlagos időigénye másodpercben

7.3. Optikai folyam számolása GPU-n

OpenCV keretrendszerben néhány általam is használt eljárást a CUDA [47] platformra is implementálták, melyeket egy külön `cv::cuda` névteren belül találhatunk. Ezek segítségével, ha a videókártya támogatja, az algoritmusok implementációit a grafikus kártyán hatékonyabban futtathatjuk így növelve a teljesítményt.

A 7.2. táblázatban láthatjuk, hogy az optikai folyam számolások, mind a sűrű pontmegfeleltetések számolásához, mind az előtér maszkhoz sok időt emésztenek fel. Miután feltelepítettem a CUDA Toolkit-et az NVIDIA honlapjáról [48], újrafordítottam az OpenCV-t forrásból engedélyezve a CUDA modulokat. A GPU-s Färneback optikai folyam implementációját a `cv::cuda::FarnebackOpticalFlow` osztályban találtam meg. Ezzel a módosítással további jelentős sebességnövekedést értem el, mely statisztikáit a 7.3. táblázat mutatja. Ezzel a megoldással az első jeleneten kb. 4 FPS-es feldolgozási sebességet értem el.

Az ORB-bal történő jellegzetes pontok detektálása és leírók számolása is történhet a GPU-n, viszont tesztelések alapján úgy tűnt, hogy ezek a műveletek nem szálbiztosak. A dokumentáció [49] is említi, hogy némelyik függvény konstans GPU buffert használ, így ugyanazon műveletek egymás utáni többszöri meghívása okozhat problémákat (az előző optikai folyam számolása úgy tűnt nem ilyen – konkrét dokumentációt erről nem találtam). Sebességen ez kb. 0,01 másodpercent jelentett volna képkockánként, annak kárára, hogy ezt a részt szálbiztosá teszem, így ezt az ötletet elvetettem.

A 7.3. táblázatban jól látható, hogy jelenleg a legtöbb időt a textúrázottság meghatározása, valamint maga az optikai folyam kiszámolása viszi el. Utóbbin már nagyon nem lehet javítani, esetleg egy másik algoritmus választásával, előbbin pedig GPU-n történő számolással. Ezeket érdemes lenne a jövőben megvizsgálni.

Tevékenység képkockánként	Eltöltött idő (s)			
	min	max	átlag	szórás
Előtér maszk (2 kamerára)	0,0304	0,0384	0,0337	0,00148
Pontpárosítások ORB-bal	0,0176	0,0351	0,0262	0,00321
Objektumok párosítása	0,0298	0,0477	0,0395	0,00328
Textúrázottság meghatározása	0,0252	0,131	0,0886	0,0188
Optikai folyam oda-vissza	0,0404	0,226	0,124	0,0405
Sűrű pontmegfeleltetések	0,00101	0,0166	0,00773	0,00256
Háromszögelés	0,000221	0,0561	0,0348	0,012
Vizualizáció	0,00252	0,0106	0,00665	0,0016
Teljes folyamat	0,137	0,338	0,259	0,0406

7.3. táblázat. Többszálú végrehajtás és GPU-n történő optikai folyam számolás esetén az első jelenet feldolgozási statisztikája

Megjegyzem, hogy az OpenCV-nek a még ki nem adott 3.0-s verziójától kezdve az OpenCL keretrendszerrel is jobb az integrációja [50], és a dokumentáció alapján kicsiny

befektetés mellett, lényegében transzparens módon érhető el, hogy minden, amit felkészítettek rá az OpenCL kernelek segítségével párhuzamosan fusson heterogén rendszerekben. Amint láthattuk a CUDA-s megvalósítás nem ilyen, ott explicit függvényhívásokkal kell operálnunk a megnövekedett teljesítmény eléréséhez. Az OpenCL egyik előnye az lenne, hogy a videókártya típusától függetlenül tudnánk kihasználni az ebben rejlő lehetőségeket. Nekem sajnos nem sikerült a gyakorlatban kipróbálnom az OpenCL-es újításokat, és a kódbazis még annyira friss, hogy a dokumentációban és az interneten sem találtam ehhez segítséget.

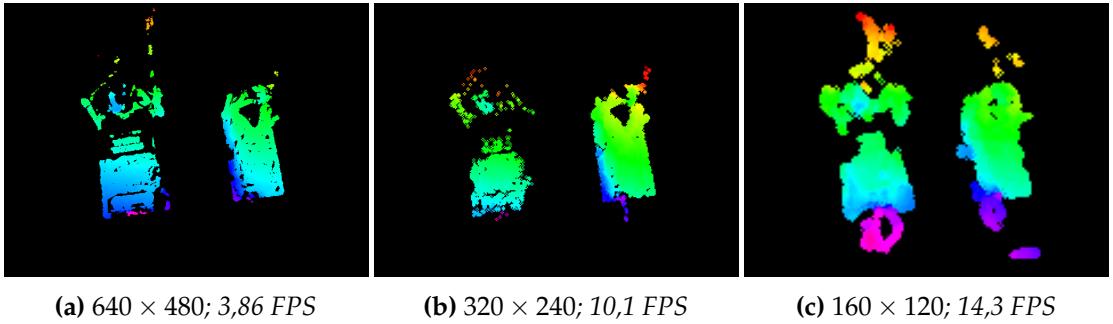
7.4. Valós idejű helyreállíthatóság vizsgálata

A következőkben megvizsgálom, hogy milyen feltételek mellett érhető el a megvalósított megoldással valós idejű helyreállítás az átalam használt számítógépen. Az alkalmazott kamerákból kinyert videofolyamok átlagosan 30 képkockát tartalmaznak másodpercenként. Mivel a televíziózásban és a mozi világában még mindig igen elterjedt a 24 FPS használata, így én is akkor tekintem valós idejűnek a feldolgozást, ha legalább 24 képkockát sikerül rekonstruálnom másodpercenként. Amennyiben meghaladom a 12 képkockát másodpercenként, akkor azt közel-valós idejűnek tekintem. A következőkben megvizsgálom, hogy mekkora felbontás, valamint képen látható objektum-méret esetén milyen sebesség érhető el.

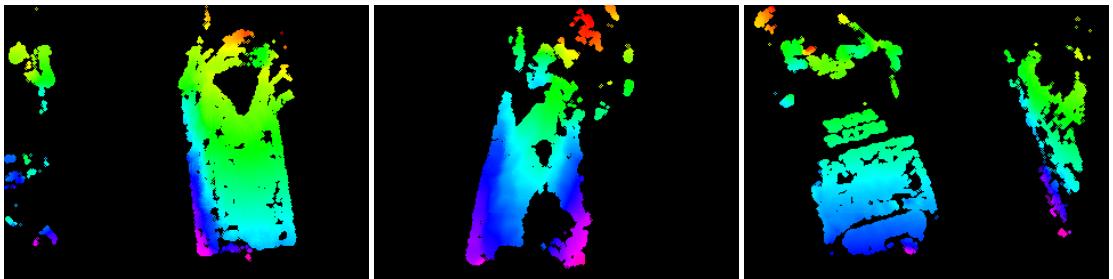
Először a képkockák méretét próbáltam csökkenteni a `cv::resize()` függvényel két lépésben, először 320×240 -es, majd 160×120 -as felbontásra. A sebességre jótékony hatással volt a képméretek csökkenése, viszont a rekonstrukció minőségére éppen ellenkezőleg. Azokon a képkockákon, ahol az objektumok helyben forogtak, de vízszintes irányban nem sokat mozdultak értékelhetetlen lett a helyreállítás. Ez annak a következménye, hogy sok párosítás szempontjából fontos pont a kicsinyítés miatt eltűnt, illetve a képpontok eredeti mozgása a kisebb képeken elhanyagolható lett. Azonos képkockához tartozó rekonstrukciók eredményei és azok sebességei (csak azon képkockák feldolgozásai idejeit számolva, ahol ténylegesen sikerült helyreállítani legalább 1 objektumot) láthatóak a 7.2. ábrán. Ezek alapján sajnos nem sikerült elérni a valós idejű feldolgozást, csak a közel-valós idejűt jelentős minőségcsökkenés és információveszteség mellett.

Ezt követően, a képkockák méretét nem kicsinyítéssel, hanem levágással csökkentettem. Az első jelenet esetében a képek érdemi része középen található, így először nagyból középső 320×240 -es felbontású részt vettem figyelembe (mindegyik kamerán azt a részt, amelyen nagyból ugyanaz a térrész látható). Ekkor átlagosan 7,35 FPS-es sebességet értem el, mely főleg annak köszönhető, hogy a kisebb méret miatt általában csak 1 objektum látszódott, így csak egyet kellett rekonstruálni. A 7.3. ábrán látható ugyanazon képkockáknak a helyreállított vizualizációi, mint a 6.2. ábrán (55. oldal).

Ebből kiindulva az objektum detektálás implementációját lecseréltem a `SingleObjectSelector`-ra, valamint az objektumot a kamerától távolabbi mozgattam,



7.2. ábra. Egyre kisebb képkockák esetén a helyreállítás eredménye a bal oldali kamera nézőpontjából



7.3. ábra. Az első jelenet 40., 100. és 170. képkockájának a középső 320 x 240-es képrészleteinek helyreállításai a bal oldali kamera nézőpontjából

hogy kisebb területet foglaljon el, és a képeken az objektumot tartalmazó 160 x 140-as területeket használtam a rekonstruáláshoz. Nagyjából minden második képkockánál volt értékelhető az eredmény, a kis objektum méret és a rendelkezésre álló területen történő kicsiny mozgás miatt. Időnként a maszk kijelölés nem sikerült jól, máskor pedig az előzetes ORB alapú párosítás volt helytelen, ami rossz optikai folyamot eredményezett. A 7.4. ábrán látható néhány egészen pontos rekonstrukció a két kamera képével és a két kamera közötti nézőpontból rekonstruált nézettel.

Ilyen korlátok mellett éppen sikerült elérni a másodpercenkénti 12 képkocka rekonstrukcióját, ami az általam közel-valós idejű feldolgozásnak tekintett sebesség alsó határa. A 7.4. táblázat mutatja az adott lépések hosszát. Átlagosan 5 pixel lett képkockánként az átlagos visszavetítési hiba, ami szintén a távoli objektum következtében jelentkező információhiánynak tudható be. A továbbiakban nem próbálkoztam még kisebb képeken az objektumok helyreállításával, mert már ezen utolsó feldolgozás hasznossága az alsó küszöböt érinti. Ennél kisebb információtartalommal rendelkező rekonstrukcióknak nincs gyakorlati haszna, még valós időben sem.

7.5. Összefoglaló

Ebben a fejezetben bemutattam, hogy az implementáció sebességét milyen megközelítések mentén tudtam javítani. Kitértem az OpenMP segítségével megvalósított párhuzamosításra, valamint az OpenCV-ben támogatott CUDA platform használatára. A fejezet



(a)



(b)

7.4. ábra. 160×140 -es felbontású képek és azok rekonstrukciói. A rekonstrukció a két kamera közötti képzeletbeli szakasz felezőpontjában vett nézőpontból készült

végén pedig a valós idejű helyreállíthatóságot és ennek korlátait vizsgáltam meg. Ennek során arra jutottam, hogy az elkészült megoldással szigorúan vett valós idejű rekonstrukció nem valósítható meg az általam használt hardveren, ellenben kvázi-valós idejű igen, amennyiben kis méretű ($\sim 150 \times 150$) objektumot kell detektálni és helyreállítani a videofolyamok alapján.

Tevékenység képkockánként	Eltöltött idő (s)			
	min	max	átlag	szórás
Előtér maszk (2 kamerára)	0,0127	0,0425	0,0189	0,0055
Pontpárosítások ORB-bal	0,00229	0,0216	0,0066	0,0031
Objektumok párosítása	0,00555	0,129	0,0188	0,0112
Textúrázottság meghatározása	0,00117	0,0352	0,0158	0,00562
Optikai folyam oda-vissza	0,00239	0,0485	0,0222	0,00548
Sűrű pontmegfeleltetések	0,000649	0,0161	0,00284	0,00292
Háromszögelés	0,00001	0,0188	0,00434	0,00427
Vizualizáció	0,000257	0,00776	0,00106	0,000901
Teljes folyamat	0,0184	0,206	0,083	0,0263

7.4. táblázat. 160×140 -es felbontású képek rekonstrukciói során az egyes lépések átlagos hosszai (230 képkocka)

8. fejezet

Eredmények

Ebben a fejezetben az elért eredményeket összesítem, és értékelem a megoldásom.

A 6. fejezet elején bemutattam az első jelenetet, amit később a gyorsítási lehetőségek vizsgálatánál is felhasználtam. Ez 178 darab 640×480 -as felbontású képből állt, melyeken két teásdobozt mozgattam, ezek mérete összesen kb. a képek területének 15%-át tettek ki. A bemutatott képeken jól látszódott, hogy a helyreállítás minősége a mélységet bemutató színábrázolással szemléltetve nagyon jónak értékelhető, melyet az átlagos visszavetítési hibák (melyek átlaga 0,806 pixel lett) is jól alátámasztottak.

Ahogy a korábbi fejezetekben említésre került, a választott megoldáshoz szükséges, hogy az objektumok jól textúrázottak legyenek, valamint azért, hogy a különböző objektumokat elkülöníthessem, egymáshoz képest is eltérő leírókkal (alak, szín, textúra jellege) kell rendelkezniük. Ezek az én esetben nagyjából teljesültek, egyedül az egyik teásdoboz volt néhol túl homogén, de ez jól bemutatta a megoldás ezen korlátját.

A kezdeti 1 FPS-es sebességből párhuzamosítás és GPU-n történő gyorsítás után egészen 3,86 FPS-re sikerült gyorsítani ugyanazon jeleneten a feldolgozás sebességét, amely jelentős sebességnövekedést jelentett. Megvizsgáltam azt is, hogy kisebb képek esetén, mennyire közelíthető meg a kívánt valós idejű feldolgozás. A tesztek alapján úgy találtam, hogy az általam rendelkezésre álló hardveren a készített megoldás még az értékelhető 160×140 -es felbontás körüli képrészletekből jó minőségűnek mondható helyreállítást készített (ámbátor már jelentősebb, 5 pixeles átlagos visszavetítési hibával), és elérte a 12 FPS-es sebességet. Az elérni kívánt másodpercenkénti ~24 képkocka feldolgozását viszont nem sikerült megközelíteni, ennél kisebb képeken pedig már nem volt gyakorlati haszna kísérletezni. Ezen eredményeket összefoglalóan a 8.1. táblázat mutatja be, melyen a 6. fejezet többi jelenetének sebessége is látható a végső koncepcióval (többszálon futtatva, GPU-n is számolva).

Fontos megemlíteni, hogy a sebesség különbség az utolsó 2-2 esetben abból adódik, hogy amikor az eredeti kép egy részletét vizsgáltam, akkor a kép legnagyobb részét maga az objektum tette ki, míg a legkicsinyített verzióban csak egy kis részét.

Módszer	Felbontás	Átlagos sebesség
1. jelenet egyszálon, CPU-n	640×480	1,1 FPS
1. jelenet többszálon, CPU-n	640×480	2,56 FPS
1. jelenet többszálon, GPU-n is	640×480	3,86 FPS
2. jelenet többszálon, GPU-n is	640×480	4,48 FPS
3. jelenet többszálon, GPU-n is	640×480	6,06 FPS
1. jelenet, eredeti kép kicsinyítve	320×240	10,1 FPS
1. jelenet, eredeti kép kicsinyítve	160×120	14,3 FPS
1. jelenet, eredeti kép egy részlete	320×240	7,7 FPS
1. jelenet, eredeti kép egy részlete	160×140	12 FPS

8.1. táblázat. Egyes módszerek és kiülönböző felbontás esetén az átlagosan elért sebességek

A fentiek tükrében a kitűzött feladat megoldását sikeresnek tekintem, az elért sebesség gyakorlati körülmények között is jól használható. A következőkben a lehetséges továbbfejlesztési lehetőségekre térek ki.

8.1. Továbbfejlesztési lehetőségek

A 7.3. táblázatban (63. oldal) jól látszódik, hogy a textúrázottság meghatározása sok időt emész fel. Ezt GPU-ra megírva (CUDA vagy OpenCL kernel segítségével) jelentős sebességnövekedést várnék, amit érdemes lenne kivizsgálni. Ehhez viszont jobban meg kéne ismerni a heterogén párhuzamos programozás adta lehetőségeket.

Az előtér maszk meghatározása is jelentős időbe telik. Sajnos a gyorsabb MOG2 algoritmus rossz minőségű maszkokat eredményezett, bármilyen paraméterezéssel is próbálkoztam. Egy mélyrehatóbb vizsgálat lenne szükséges, hogy milyen egyéb megoldásokat lehetne itt alkalmazni, amely a teljes feldolgozás sebességét javítaná.

Érdemes lenne kipróbálni egyéb sűrű optikai folyamokat adó algoritmusokat is a meglévő mellett, hátha van olyan, ami hasonló eredményeket produkál ugyanazon hardveren kevesebb idő alatt. Erre Christopher Zach és társai [51] munkája egy jó alternatívának tűnik.

Az OpenCV új verziójában átstruktúrált OpenCL-es implementációk [50] alkalmazása szintén kedvező hatással lehet a performanciára, ugyan a dolgozatom során megpróbálkoztam ezzel, de sajnos sikertelenül. Elképzelhető, hogy ez a kész 3.0-s (ami jelenleg RC1 fázisban van) verzióban már gond nélkül fog működni.

9. fejezet

Összefoglalás

Diplomamunkámban két rögzített kamera videofolyamai alapján előállítottam a megfigyelt térrészben mozgó objektumok rekonstrukcióját egy, a két kamera között választott nézőpontból. Áttekintettem a gépi látás szakirodalmában publikált és a feladatomhoz kapcsolódó legfontosabb eredményeket, különös tekintettel azok elméleti hátterére és az alkalmazható algoritmusokra. Megterveztem és elkészítettem egy szimulációs alkalmazást, aminek segítségével az eljárást három jelenetben teszteltem. A szoftver segítségével a használt kamerákat bekalibrálhatjuk, és azok videofolyamai alapján folyamatos rekonstrukciót végezhetünk, melyhez tartozó nézőpontot egy csúszka segítségével változtathatjuk a két kamera között. A valós idejű feldolgozás korlátait megvizsgáltam, az eredmények alapján a megoldást a gyakorlatban is alkalmazhatónak találtam.

Az alkalmazott kamerák számának további növelésével a nézőpont helyzete, ahonnan megfelelő rekonstrukcióra számíthatunk, nagyobb szabadságfokkal választható. Például egy kellő számú kamerával felszerelt teremben mozgó személyek helyzetei a valós idejű videofolyamok alapján nagyon jó közelítéssel meghatározhatóak, amely egy biztonsági megoldás alapjául szolgálhat.

A továbbfejlesztés egy másik lehetősége a feldolgozási teljesítmény növelése, mely hatékonyabb algoritmusokkal és azok célhardveren történő futtatásával érhető el.

Köszönetnyilvánítás

Végezetül szeretnék köszönetet mondani konzulensemnek, dr. Kovács Gábornak, aki folyamatosan támogatott, szakmai észrevételeivel és tanácsaival segítette munkámat. Sokat köszönhetek édesapámnak és barátaimnak, visszajelzésekkel és javaslataikkal nagyban hozzájárultak a dolgozat elkészítéséhez.

Irodalomjegyzék

- [1] Andrew Nolan et al. "Obstacle mapping module for quadrotors on outdoor Search and Rescue operations". In: *International Micro Air Vehicle Conference and Flight Competition (IMAV2013), Toulouse, France*. 2013 (cit. on p. 6).
- [2] Tomáš Krajník et al. "AR-drone as a platform for robotic research and education". In: *Research and Education in Robotics-EUROBOT 2011*. Springer, 2011, pp. 172–186 (cit. on p. 6).
- [3] M Anwar Ma'sum et al. "Autonomous quadcopter swarm robots for object localization and tracking". In: *Micro-NanoMechatronics and Human Science (MHS), 2013 International Symposium on*. IEEE. 2013, pp. 1–6 (cit. on p. 6).
- [4] Yanninck Morvan. "Acquisition, compression and rendering of depth and texture for multi-view video". PhD thesis. Technische Universiteit Eindhoven, 2009 (cit. on pp. 12, 14).
- [5] The Unkelbach Valley Software Works. *Radial Distortion Correction*. 2014. URL: http://www.uni-koeln.de/~a1001/radcor_files/hs100.htm (cit. on p. 13).
- [6] OpenCV dev team. *Camera calibration With OpenCV – OpenCV 2.4.9.0 documentation*. 2014. URL: http://docs.opencv.org/doc/tutorials/calib3d/camera_calibration/camera_calibration.html (cit. on pp. 14, 15).
- [7] Kriván Bálint. "Kép- és videófeldolgozás". Önálló laboratórium beszámoló. Budapesti Műszaki és Gazdaságtudományi Egyetem, 2013 (cit. on p. 16).
- [8] Richard Hartley és Andrew Zisserman. *Multiple view geometry in computer vision*. Second. Cambridge university press, 2004 (cit. on pp. 16, 17).
- [9] OpenCV dev team. *Camera Calibration and 3D Reconstruction – OpenCV 2.4.11.0 documentation*. 2015. URL: http://docs.opencv.org/modules/calib3d/doc/camera_calibration_and_3d_reconstruction.html (cit. on pp. 18, 35, 47).
- [10] Heiko Hirschmuller. "Stereo processing by semiglobal matching and mutual information". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 30.2 (2008), pp. 328–341 (cit. on p. 18).
- [11] Sergey Kosov. "Multi-View 3D Reconstruction with Variational Method". MA thesis. Department of Computer Science, Saarland University, 2008 (cit. on p. 18).

- [12] David Fleet és Yair Weiss. "Optical flow estimation". In: *Handbook of Mathematical Models in Computer Vision*. Springer, 2006, pp. 237–257 (cit. on p. 20).
- [13] Vámossy Zoltán. "Mobil robotok navigációja PAL-optikára alapozott gépi látással". PhD thesis. Budapesti Műszaki és Gazdaságtudományi Egyetem (cit. on p. 21).
- [14] Jean-Yves Bouguet. "Pyramidal implementation of the affine lucas kanade feature tracker description of the algorithm". In: *Intel Corporation 5* (2001), pp. 1–10 (cit. on p. 21).
- [15] The Wikipedia. *Lucas–Kanade method - Wikipedia, the free encyclopedia*. 2014. URL: http://en.wikipedia.org/wiki/Lucas–Kanade_method (cit. on p. 21).
- [16] Jianbo Shi és Carlo Tomasi. "Good features to track". In: *Computer Vision and Pattern Recognition, 1994. Proceedings CVPR'94., 1994 IEEE Computer Society Conference on*. IEEE. 1994, pp. 593–600 (cit. on p. 21).
- [17] OpenCV dev team. *Optical Flow – OpenCV 3.0.0-dev documentation*. 2014. URL: http://docs.opencv.org/master/doc/py_tutorials/py_video/py_lucas_kanade/py_lucas_kanade.html (cit. on pp. 21, 23).
- [18] Gunnar Farnebäck. "Two-frame motion estimation based on polynomial expansion". In: *Image Analysis*. Springer, 2003, pp. 363–370 (cit. on pp. 21–23).
- [19] Tony Jebara, Ali Azarbajayani, és Alex Pentland. "3D structure from 2D motion". In: *Signal Processing Magazine, IEEE* 16.3 (1999), pp. 66–84 (cit. on p. 23).
- [20] Daniel Lélis Baggio et al. *Mastering OpenCV with Practical Computer Vision Projects*. Packt Publishing, 2012 (cit. on p. 23).
- [21] Richard I Hartley és Peter Sturm. "Triangulation". In: *Computer vision and image understanding* 68.2 (1997), pp. 146–157 (cit. on pp. 24–26, 48).
- [22] Fogaras Dániel. *Szinguláris felbontás*. URL: http://www.cs.bme.hu/~fd/papers/fog_svd02.pdf (cit. on p. 25).
- [23] Roberto Brunelli és T Poggio. "Template matching: Matched spatial filters and beyond". In: *Pattern Recognition* 30.5 (1997), pp. 751–768 (cit. on p. 26).
- [24] Paul Viola és Michael Jones. "Rapid object detection using a boosted cascade of simple features". In: *Computer Vision and Pattern Recognition, 2001. CVPR 2001. Proceedings of the 2001 IEEE Computer Society Conference on*. Vol. 1. IEEE. 2001, pp. I–511 (cit. on p. 26).
- [25] Pakorn KaewTraKulPong és Richard Bowden. "An improved adaptive background mixture model for real-time tracking with shadow detection". In: *Video-based surveillance systems*. Springer, 2002, pp. 135–144 (cit. on p. 26).
- [26] OpenCV dev team. *How to Use Background Subtraction Methods – OpenCV 3.0.0-dev documentation*. 2014. URL: http://docs.opencv.org/trunk/doc/tutorials/video/background_subtraction/background_subtraction.html (cit. on p. 27).
- [27] Itseez. *OpenCV | OpenCV*. [Online; hozzáférve 2014. december 1-jén]. 2014. URL: <http://opencv.org/> (cit. on p. 29).

- [28] OpenCV dev team. *Motion Analysis and Object Tracking – OpenCV 2.4.11.0 documentation*. 2015. URL: http://docs.opencv.org/modules/video/doc/motion_analysis_and_object_tracking.html#backgroundsubtractorMog2 (cit. on pp. 36, 37).
- [29] Fazekas Gábor és Hajdu András. *Képfeldolgozási módszerek*. Tech. rep. Debreceni Egyetem, Informatikai Intézet, 2004. URL: http://www.inf.unideb.hu/~hajdua/km_main.pdf (cit. on p. 37).
- [30] Edward Rosten és Tom Drummond. “Machine learning for high-speed corner detection”. In: *Computer Vision–ECCV 2006*. Springer, 2006, pp. 430–443 (cit. on p. 40).
- [31] Jiri Matas et al. “Robust wide-baseline stereo from maximally stable extremal regions”. In: *Image and vision computing* 22.10 (2004), pp. 761–767 (cit. on p. 40).
- [32] Alexandre Alahi, Raphael Ortiz, és Pierre Vandergheynst. “Freak: Fast retina key-point”. In: *Computer Vision and Pattern Recognition (CVPR), 2012 IEEE Conference on*. IEEE. 2012, pp. 510–517 (cit. on p. 40).
- [33] Ethan Rublee et al. “ORB: an efficient alternative to SIFT or SURF”. In: *Computer Vision (ICCV), 2011 IEEE International Conference on*. IEEE. 2011, pp. 2564–2571 (cit. on p. 40).
- [34] David G Lowe. “Distinctive image features from scale-invariant keypoints”. In: *International journal of computer vision* 60.2 (2004), pp. 91–110 (cit. on p. 40).
- [35] Herbert Bay, Tinne Tuytelaars, és Luc Van Gool. “Surf: Speeded up robust features”. In: *Computer vision–ECCV 2006*. Springer, 2006, pp. 404–417 (cit. on p. 40).
- [36] Daghmawi Bekele, Michael Teutsch, és Tobias Schuchert. “Evaluation of binary key-point descriptors”. In: *Image Processing (ICIP), 2013 20th IEEE International Conference on*. IEEE. 2013, pp. 3652–3656 (cit. on p. 41).
- [37] Marius Muja és David G. Lowe. “Scalable Nearest Neighbor Algorithms for High Dimensional Data”. In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 36 (2014) (cit. on p. 41).
- [38] OpenCV dev team. *Fast Approximate Nearest Neighbor Search*. 2015. URL: http://docs.opencv.org/modules/flann/doc/flann_fast_approximate_nearest_neighbor_search.html (cit. on p. 41).
- [39] Marius Muja és David G Lowe. “Fast matching of binary features”. In: *Computer and Robot Vision (CRV), 2012 Ninth Conference on*. IEEE. 2012, pp. 404–410 (cit. on p. 41).
- [40] Chanderjit Bajaj. “Proving geometric algorithm non-solvability: An application of factoring polynomials”. In: *Journal of Symbolic Computation* 2.1 (1986), pp. 99–102 (cit. on p. 44).
- [41] You Yang et al. “Dynamic 3D scene depth reconstruction via optical flow field rectification”. In: *PloS one* 7.11 (2012), e47041 (cit. on p. 45).

- [42] Roy Shilkrot. *A toy library for Structure from Motion using OpenCV 2.3+*. 2015. URL: <https://github.com/royshil/SfM-Toy-Library> (cit. on p. 48).
- [43] Ken Shoemake. “Animating rotation with quaternion curves”. In: *ACM SIGGRAPH computer graphics*. Vol. 19. 3. ACM. 1985, pp. 245–254 (cit. on p. 50).
- [44] Gaël Guennebaud, Benoît Jacob, et al. *Eigen v3*. <http://eigen.tuxfamily.org>. 2010 (cit. on p. 50).
- [45] Leonardo Dagum és Ramesh Menon. “OpenMP: an industry standard API for shared-memory programming”. In: *Computational Science & Engineering, IEEE* 5.1 (1998), pp. 46–55 (cit. on p. 61).
- [46] OpenMP Architecture Review Board. *OpenMP Application Program Interface Version 3.1*. July 2011. URL: <http://www.openmp.org/mp-documents/OpenMP3.1.pdf> (cit. on p. 61).
- [47] John Nickolls et al. “Scalable parallel programming with CUDA”. In: *Queue* 6.2 (2008), pp. 40–53 (cit. on p. 63).
- [48] NVIDIA Corporation. *CUDA Toolkit*. 2015. URL: <https://developer.nvidia.com/cuda-toolkit> (cit. on p. 63).
- [49] OpenCV dev team. *OpenCV: cv::cuda::Stream Class Reference*. 2015. URL: http://docs.opencv.org/master/d9/df3/classcv_1_1cuda_1_1Stream.html (cit. on p. 63).
- [50] Harris Gasparakis. *OpenCV 3.0 – The Transparent API and OpenCL™ Acceleration*. Oct. 2014. URL: <http://developer.amd.com/community/blog/2014/10/15/opencv-3-0-transparent-api-opencl-acceleration/> (cit. on pp. 63, 68).
- [51] Christopher Zach, Thomas Pock, és Horst Bischof. “A duality based approach for realtime TV-L1 optical flow”. In: *Pattern Recognition*. Springer, 2007, pp. 214–223 (cit. on p. 68).

Függelék

F.1. Az alkalmazás elérhetősége

Az elkészült alkalmazás Git tárolója elérhető az alábbi címen:

```
https://github.com/messzo/diploma
```

Az alkalmazás C++ nyelven íródott, a platform független támogatást és a függőségek kezelését a CMake teszi lehetővé. Linuxon az alkalmazás a következő parancsok kiadásával fordítható:

```
$ mkdir build && cd build
$ cmake ..
$ make -j4
```

Az alkalmazás rövid használati útmutatója szintén a fenti tárolóban elérhető.