ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Alba Regia Műszaki Kar
Természettudományi és Szoftvertechnológiai Intézet

SZAKDOLGOZAT

Markerdetektálás alapú holografikus megjelenítés kevert valóságban

OE-AMK

2024

Hallgató neve: Kovács Mátyás

Hallgató törzskönyvi száma: T001032/FI12904/A-M

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Óbudai Egyetem
Alba Regia Műszaki Kar
Természettudományi és Szoftvertechnológiai Intézet

SZAKDOLGOZAT FELADATLAP

Hallgató neve: Kovács Mátyás
Szakdolgozat száma: SZD2301230932238803[redacted]
Törzskönyvi száma: T001032/FI12904/A-M **Neptun kódja:** [redacted]

Szak: mérnökinformatikus
Specializáció: Felhő szolgáltatási technológiák és IT biztonság specializáció

A dolgozat címe: Markerdetektálás alapú holografikus megjelenítés kevert valóságban
A dolgozat címe angolul: Marker Detection Based Holographic Projection in Mixed Reality
A feladat részletezése: A robotkarbantartásban hasznos segítség lenne egy kevert valóságú szemüveg alkalmazása a kisegítő információk megjelenítésére. Egy ilyen alkalmazás fejlesztése nehéz informatikai feladat, mert annak ellenére, hogy létező függvénykönyvtárak alkalmazásával a feladat megoldható, a különböző eszközök közötti átjárás kompatibilitási problémákkal jár, ezen kívül a különböző típusú és irányúltságú koordinátarendszerek közötti váltásokat is kezelni kell. A hallgató feladata mintaalkalmazást fejleszteni egy ilyen eszközre, amely a későbbiekben egy újrafelhasználható csomagként felhasználható lesz. A dolgozat tartalmazza a szükséges számítógépes grafikai alapok összefoglalását, a kész szoftver dokumentációját, tesztelésének módját és eredményét.

Intézményi konzulens neve: Nagyné Dr. Hajnal Éva

A kiadott téma elévülési határideje:2026. 02. 10.
Beadási határidő: 2023. 12. 15.

A szakdolgozat: Nem titkos.

ÓBUDAI EGYETEM Alba Regia Műszaki Kar Természettudományi és Szoftvertechnológiai Intézet Székesfehérvár

Kiadva: Székesfehérvár, 2023. 10. 17. P.H. .....
Intézetigazgató

A dolgozatot beadásra alkalmasnak találom: 2023. 12. 15.

.....
belső konzulens

.....
külső konzulens

ÓBUDAI EGYETEM
ÓBUDA UNIVERSITY

Alba Regia Műszaki Kar
Természettudományi és Szoftvertechnológiai Intézet

HALLGATÓI NYILATKOZAT

Alulírott hallgató kijelentem, hogy a szakdolgozat/diplomamunka saját munkám eredménye, a felhasznált szakirodalmat és eszközöket azonosíthatóan közöltem. Az elkészült szakdolgozat/diplomamunkában található eredményeket az egyetem és a feladatot kiíró intézmény saját céljára térítés nélkül felhasználhatja, a titkosításra vonatkozó esetleges megkötések mellett.

Kelt: 2024.05.13

Kovács Mátyás
hallgató aláírása

Tartalomjegyzék

1. RÖVIDÍTÉSEK JEGYZÉKE

ABI Application Binary Interface 16
API Application Programming Interface 16, 17, 18, 23, 33, 36
CISC Complex Instruction Set Computing 13
CMOS Complementary Metal-Oxide Semiconductor 4
COM Component Object Model 16, 17
DLL Dynamic Link Library 17, 18, 27, 29
DSP Digital Signal Processor 4
EXE Executable file 17, 18
FPS Frames Per Second 12
HL2 HoloLens 2 ..... 21, 31, 32, 33, 34, 40, 42, 44, 46
IDL Microsoft Interface Definition Language 21, 22
IMU Inertial Measurement Unit 10, 12
LBS Laser Beam Scanning 11
MRTK Mixed Reality Toolkit 30, 31
OS Operating System 13
PV Photo Video 12, 33, 39, 44
Repo Repository 25, 26
RISC Reduced Instruction Set Computing 13
SDK Software Development Kit 13, 17
UWP Universal Windows Platform 13, 17, 31
viSLAM Visual Inertial Simultaneous Localization And Mapping 12
WRL Windows Runtime Template Library 17, 21

2. BEVEZETÉS

A kevert valóság technológia a virtuális és valós világ összekapcsolásával foglalkozik, azzal a céllal, hogy a felhasználó számára egy minél valósághűbb élményt biztosítsanak anélkül, hogy a valós világot egy teljesen virtuálissal helyettesítenék. A felhasználó ebben az esetben egyszerre tapasztalhatja meg a valós és virtuális világot, ami új lehetőségeket teremt rengeteg felhasználási terület számára.

A robotiparban használt rendszerek tervezése és fejlesztése egy összetett és költséges feladat. A kevert valóságnak az alkalmazása ebben a szektorban jelentősen megkönnyíthetné a fejlesztők munkáját. Több gyártó is kínál ehhez a technológiához eszközöket, melyek speciális szenzorokkal vannak felszerelve a környezetük feltérképezése és megértése érdekében.

Egy ilyen eszköz a valós térben jeleníthetné meg a virtuális terveket, gyártósorok modelljeit, szimulációkat, megkönnyíthetné a robotikai rendszerek optimalizálását és tesztelését. Ez kevesebb fizikai prototípus építéséhez vezetne és lehetővé tenné a cégek számára, hogy jelentősen csökkentsék a költségeket, valamint értékes időt takaríthatnának meg.

Az ilyen komplex feladatot megvalósító szoftveres megoldás általában több kisebb rész komponensből áll. A szakdolgozat az egyik ilyen komponens fejlesztését mutatja be, ami egy különálló szoftvercsomagként egy speciális mobil eszközön futtatva képes a felhasználó által a valós világban megjelölt pontokra a virtuális világban létező objektumok megjelenítésére. Bemutatásra kerülnek a különböző keretrendszerek és szoftver könyvtárak, amelyek lehetővé teszik a valós világban lévő objektumok, pontok speciális markerekkel való megjelölését, ezeknek a markereknek a generálását, felismerését, valamint az eszközbe épített szenzorok által begyűjtött adatok közel valós időben való feldolgozását és kiértékelését. Részletesen ismertetésre kerül az eszköz, amelyre a szoftvercsomag készült, az azon található szenzorok paraméterei, a kalibráció módszertana és folyamata, valamint a fő alkalmazás, ami megvalósítja a markerek felismerését és a virtuális objektumok megjelenítését.

3. DIGITÁLIS KAMERÁK ÉS PARAMÉTEREIK

A digitális kamera egy komplex eszköz, amely átalakítja a fényt képekké és azokat bináris formában továbbítja tárolásra vagy megjelenítésre.

3.1 Kamera lencsék

Minden kamera fontos részét képezi egy vagy több konvex, vagy konkáv általában üvegből készült lencse melyek a kamera szenzor elött helyezkednek el és feladatuk a beáramló fény összegyűjtése és egy pontba való fókuszálása [1].

3.1.1 Fókusztávolság

A fókusztávolság határozza meg a távolságot a kamera szenzor és a lencse optikai közepe között. Minden lencsének más és más a fókusztávolsága, amit milliméterben (mm) adnak meg. Minél nagyobb a fókusztávolság, annál erősebb a nagyítás, de kisebb a látószög, minél alacsonyabb, annál gyengébb a nagyítás és nagyobb a látószög.

3.1.2 Apertúra (F-érték)

A lencse apertúrája azt a nyílást jelöli, amelyen keresztül haladva a fény eléri a kamera szenzort. Ez egy változtatható érték, ami az emberi szem íriszeihez hasonlóan működik. A kamera lencséknél az apertúrát F-értékként adják meg, általában minél alacsonyabb az F-érték annál nagyobb az apertúra nyílása. Az apertúra hatással van a mélységélességre és az alacsony fényviszonyok közötti működésre. Kisebb F-értékű apertúra esetén több fény fog áthaladni a lencsén.

3.2 Kamera szenzorok

A kamera szenzor egy félvezető alapú eszköz, ami képes elnyelni a fény részecskéit (fotonokat) több millió fény érzékeny pixelen keresztül, hogy azután átalakítsa őket elektromos jelekké. Ezeket a jeleket egy nagysebességű feldolgozó egység utána

értelmezi és átalakítja digitális képekké. Többféle kamera szenzor létezik, de a legelterjedtebb mind közül a CMOS alapú, ami minden modern digitális kamerában megtalálható [2].

3.2.1 CMOS szenzor működése

Egy ilyen szenzor sok millió apró pixel rácsba kötéséből van felépítve. Minden pixel egy a többitől független fény gyűjtő. Amikor a fotonok bekerülnek a fény gyűjtőbe, először áthaladnak egy mikro lencsén, amely megakadályozza a fotonok leugrását a szenzorról, majd egy színszűrő következik. A színszűrő egy előre meghatározott mintát tartalmaz piros, zöld és kék színekből melyek rácsba vannak rendezve. (Bayer filter) Ennek a célja, hogy egy pixel csak vagy piros vagy zöld vagy kék fénynek legyen kitéve. Vannak olyan szenzorok, amik nem tartalmaznak színszűrőt, ezek csak monokróm képek rögzítésére alkalmasak. A színszűrő után a fotonok nekicsapódnak egy fény érzékeny félvezető diódának, ami átkonvertálja őket elektromos jellé. Ez az elektromos jel pixel szinten van erősítve majd egy analóg digitális konverter átalakítja digitális formába és továbbítja a digitális jelfeldolgozó egységnek. A DSP értelmezni tudja ezeket a jeleket és képes átfordítani őket egy képpé mivel minden pixelhez egy egyedi érték tartozik, ami a mért fény erejétől függ [3].

3.3 Kamera paraméterek

3.3.1 Belső paraméterek

A belső paraméterek ismerete lehetővé teszi a képkocka pixel koordinátái és a kamera koordinátái közötti leképezés megvalósítását, tehát az ezekből a paraméterekből felépített mátrix segítségével a 3D kamera koordinátákat 2D homogén képkocka koordinátákká lehet leképezni [4]. A belső paramétereket ábrázoló mátrix az 1. egyenlet szerint néz ki:

$$K = \begin{pmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} \tag{1}$$

amelyből f_x, f_y a fókusztávolság, x_0, y_0 a kamera főpontjában való eltolás és s pedig a tengelyferdeség.

Minden egyes paraméter leírja a kamera egy-egy geometriai tulajdonságát. Az ideális kameránál f_x, f_y egyenlő, de a gyakorlatban ezek eltérőek az alábbi okok miatt:

- A kamera szenzor hibái
- A képek utólagos feldolgozás során nem megfelelően lettek méretezve
- A kamera lencséi torzulást okoznak
- Kamera kalibrációs hibák

Aminek az eredménye, hogy a kép pixelei nem teljesen szögletesek. A kamera főtengelye a képsíkra merőleges vonal, ami áthalad a lencse optikai közepén. A képsíkkal való metszéspontot főpontnak nevezik. A főpontban való eltolás a főpont helyzete a szenzorhoz képest. A tengelyferdeség nyírási torzulást okoz a képen viszont ez általában 0 és emiatt sokszor nem releváns. A mátrixban használt pixel egységek átválthatóak a való világban használt mértékegységekre, ha ismerjük a kamera egyik dimenzióját a valós világban. Ha tudjuk, hogy a szenzornak van egy W szélessége mm-ben és a kép szélessége pixelben kifejezve w akkor a fókusztávolságot f_x át lehet konvertálni világ mértékegységbe a 2. egyenlet segítségével:

$$F_x = f_x \frac{W}{w} \tag{2}$$

Ugyanez a módszer érvényes többi f_y, x_0, y_0 paraméterre is:

$$F_y = f_y \frac{H}{h} \quad X_0 = x_0 \frac{W}{w} \quad Y_0 = y_0 \frac{H}{h} \tag{3}$$

A belső paraméterek mátrixszát felbonthatjuk nyírási, skálázási és eltolási transzformációk sorozatára, amelyek megfelelnek a tengelyferdeségnek, a fókusztávolságnak és a főponteltolódásnak, ezeket a 4. egyenlet szemlélteti:

$$K = \begin{pmatrix} f_x & s & x_0 \\ 0 & f_y & y_0 \\ 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & x_0 \\ 0 & 1 & y_0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} f_x & 0 & 0 \\ 0 & f_y & 0 \\ 0 & 0 & 1 \end{pmatrix} * \begin{pmatrix} 1 & {}^{s}/_{f_x} & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{4}$$

3.3.2 Külső paraméterek

A külső paraméterek leírják a kamera helyét és azt, hogy milyen irányba mutat a valós világban [5]. Az ezekből a paraméterekből felépített transzformációs mátrixnak két összetevője van: egy R forgatási mátrix és egy t eltolási vektor. A forgatást egy 3x3-as mátrix, az eltolást pedig egy 3x1-es oszlopvektor valósítja meg:

$$[R|t] = \begin{bmatrix} r_{1,1} & r_{1,2} & r_{1,3} & t_1 \\ r_{2,1} & r_{2,2} & r_{2,3} & t_2 \\ r_{3,1} & r_{3,2} & r_{3,3} & t_3 \end{bmatrix} \tag{1}$$

Ebből a mátrixból létezik egy olyan változat, ami egy plusz sorral rendelkezik (0, 0, 0, 1) az alján, amitől szögletes lesz és így tovább bontható forgatásra és eltolásra:

$$\begin{aligned} \left[\begin{array}{c|c} R & \boldsymbol{t} \\ \hline \mathbf{0} & 1 \end{array}\right] &= \left[\begin{array}{c|c} I & \boldsymbol{t} \\ \hline \mathbf{0} & 1 \end{array}\right] \times \left[\begin{array}{c|c} R & \mathbf{0} \\ \hline \mathbf{0} & 1 \end{array}\right] \\ &= \left[\begin{array}{ccc|c} 1 & 0 & 0 & t_1 \\ 0 & 1 & 0 & t_2 \\ 0 & 0 & 1 & t_3 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \times \left[\begin{array}{ccc|c} r_{1,1} & r_{1,2} & r_{1,3} & 0 \\ r_{2,1} & r_{2,2} & r_{2,3} & 0 \\ r_{3,1} & r_{3,2} & r_{3,3} & 0 \\ \hline 0 & 0 & 0 & 1 \end{array}\right] \end{aligned} \tag{2}$$

Ilyen formában leírja, hogyan kell a valós világ koordináta rendszerében lévő pontokat a kamera koordináta rendszerében lévő pontokká átalakítani. A t eltolási vektor a világ origójának pozíciójaként értelmezhető a kamera koordináta rendszerében, az R forgatási mátrix oszlopai pedig a világ koordináta rendszer tengelyeinek az irányait jelölik szintén a kamera koordináta rendszerében. Fontos megjegyezni, hogy ez a mátrix azt írja le, hogy hogyan van transzformálva a világ a kamerához képest és nem fordítva [5].

4. TÁRGYAK A VIRTUÁLIS ÉS VALÓS VILÁGBAN

4.1 Virtuális, kiterjesztett és kevert valóság

A virtuális, kiterjesztett és kevert valósághoz számos gyártó biztosít eszközöket, és az elmúlt évek során elképesztő érdeklődés övezte körül ezeket a technológiákat, mind a felhasználók, mind pedig a gyártók és a fejlesztők részéről. Fontos azonban különbséget tenni az említett technológiák között [6].

4.1.1 VR (Virtual Reality)

A virtuális valóság egy szimulált digitális környezetbe nyújt betekintést a felhasználó számára. Jelenleg a legfejlettebb VR eszközök szabad mozgást biztosítanak a digitális térben. A különleges kézi vezérlőknek köszönhetően a felhasználók képesek interaktálni a virtuális térben lévő objektumokkal is. A folyamatos fejlesztéseknek köszönhetően egyre valósághűbb a grafika, a hangtartalomnak, valamint az interaktív funkcióknak köszönhetően egyre hihetőbbé válik az élmény felhasználó számára. Ehhez az élményhez szükséges, hogy a felhasználó egy VR szemüveget viseljen. A legtöbb ilyen eszköz a számítógéphez vagy játék konzolhoz csatlakozik és feldolgozás azokon az eszközökön történik. Léteznek mobil VR szemüvegek is melyekbe beépített feldolgozó egységek vannak és ezeknél nincs szükség külső eszközre. A VR szemüvegekben általában két kijelző van a felhasználó szeméhez igen közel, és az ezeken megjelenített képek segítségével hiteti el, hogy a virtuális környezet valóságos. Eleinte a virtuális valóság technológiát a szórakoztató ipar alkalmazta, viszont mára már a VR alkalmazásokat számos szervezet és iparág széles körben használja, mint pl.: hadsereg, divat ipar, oktatás, egészségügy, építőipar, autóipar [7].

4.1.2 AR (Augmented Reality)

A kiterjesztett valóságban a felhasználók a valós világot látják és interakcióba lépnek vele miközben digitális tartalmat adnak hozzá. A legismertebb ilyen alkalmazás, ami az egész világon elterjedt az a Pokémon Go melyet használva emberek milliói járták az utcákat okostelefonjukkal virtuális lények után kutatva. Ennek a technológiának előnye, hogy

nem szükséges hozzá külön erre a célra készített eszköz, manapság rengeteg embernél van okostelefon és percek alatt letölthető rá egy alkalmazás, amivel ki próbálható a kiterjesztett valóság.

4.1.3 MR (Mixed Reality)

A kevert valóságos környezetben a valós és virtuális elemek egymással kölcsönhatásba lépnek és ennek köszönhetően túlmutatnak a virtuális és kiterjesztett valóságon. E technológiánál a virtuális tárgyak egymás mellett létezhetnek és valós időben lehet velük interakcióba lépni a fizikai tárgyakhoz hasonlóan. Amikor egy MR (Mixed Reality) szemüveget visel a felhasználó, a fizikai térben megjelenített 3D virtuális tartalmak távolabbinak vagy közelebbinek tűnhetnek, különböző perspektívákban jelenhetnek meg objektumok a felhasználó és a tárgy helyzete függvényében. Ugyan sokszor kevésbé magával ragadó élményt nyújt, mint a virtuális valóság, viszont a virtuális és kiterjesztett valóságot kapcsolja össze annak érdekében, hogy egy hihető és hatékony interakciót hozzon létre. Számos gyártó foglalkozik ilyen eszközök fejlesztésével, a legismertebb ilyen eszközök a Microsoft HoloLens, Acer Windows Mixed Reality, Lenovo Explorer és a Samsung Odyssey [6].

4.2 Tárgyak helyzetének meghatározása a valós világban

Ha meg akarjuk határozni egy objektum helyzetét a valós világban kamerák segítségével akkor szükség van a kamera valós világban lévő aktuális pozíciójára, orientációjára és a belső a paramétereire, valamint a képen található keresett objektum pontjainak koordinátáira. Ez után kiszámíthatóak a kivetülések a valós és virtuális térben lévő ismert pontok között. A valós térben az objektumok pontjait gyakran speciális referencia pontokon alapuló markerek segítségével jelölik meg, amiknek a pozícióját és orientációját utána az erre a célra kidolgozott matematikai algoritmusok segítségével határozzák meg. A probléma megoldására már léteznek kész szoftver könyvtárak melyek tartalmazzák a markerek generálásához, felismeréséhez és helyzetüknek meghatározásához, valamint az optikai szenzor (kamera) kalibrálásához szükséges algoritmusokat. A legelterjedtebb referencia pontokon alapuló markerek szögletesek és

laposak, valamint egy binárisan kódolt négyzetes területet tartalmaznak, ami fekete szegéllyel van körülvéve. A fekete részek bináris 1-nek, a fehérek pedig bináris 0-nak felelnek meg, ezekből felépítve egy mátrixot nullákból és egyesekből. A marker négy sarkából kiszámítható a pozíciója, a közepében lévő binárisan kódolt mezők pedig a marker azonosítóját, illetve a hibák detektáláshoz szükséges biteket tartalmazzák [8].

4.2.1 Gépi Látás (Computer Vision)

A gépi látás technológiának célja, hogy imitálja az emberi látást, értelmezni és felismerni tudja a digitális képeken, videókban szereplő tárgyakat, jeleneteket. Ezt olyan hardver és szoftver kombinációval érik el, ami az emberi látórendszert próbálja utánozni. Gyakran alkalmaznak minta felismerést, jellemzőkinyerést és képfeldolgozást. A gépi látás azokat a technikákat foglalja magába, amelyek segítségével a gépek képesek a vizuális információkat felismerni és megérteni. A számítógép ilyenkor komoly számítástechnikai feladatokat hajt végre és kifinomult algoritmusokat alkalmaz az optikai szenzorok (általában kamerák) által szolgáltatott képek vagy videók elemzésére és kiértékelésére [9].

4.2.2 Az ArUco marker

A közelmúltban rengeteg szögletes marker rendszert fejlesztettek, viszont a legtöbbnél komoly gondokat okozhat a fényvisszaverődés, tükröződés a markerek felületéről, a kamera zaj és az esetleges alacsony távolság több marker között [8]. Ezek miatt a rendszer akár több markert egynek azonosíthat vagy figyelmen kívül hagyhat. Az ArUco modul az OpenCV szoftverkönyvtár része, és az évek során ezekkel a problémákkal szemben rendkívül ellenállónak és hibatűrőnek bizonyult. Rengeteg dokumentáció érhető el hozzá, nagy felhasználóbázisa van mind kutatási mind ipari szinten, forráskódja publikus és meglehetősen egyszerű vele dolgozni. Főként ezen előnyei miatt esett rá a választás a projektnél.

4.2.3 ArUco markerek generálása

Amikor generáljuk a markereket, egy úgynevezett szótárt (dictionary) adunk meg bemenetként az algoritmus számára, amiben specifikáljuk a markerek és a bitek számát. Minél nagyobb a bitátmenet a markerek között annál kisebb az esélye, hogy a felismerő algoritmus összetéveszti a markert a környezetében lévő tárgyal vagy egy másik markerrel, de ez növeli a feldolgozási időt is [8]. Az algoritmus ezután a kiválasztott szótárból létrehozza a kívánt markert, ami utána kiírható egy fájlba. Ha több markert akarunk generálni, akkor az algoritmust futtathatjuk egy ciklus részeként is.

4.2.4 ArUco markerek felismerése és pozíciójának meghatározása

Az ArUco markerek felismerése és pozíciójának meghatározása több lépésből áll. Először be kell állítani az adott szótárt, amiből a keresett marker származik, utána fel kell venni a kamerának a belső paramétereit és a torzítási együtthatóinak modelljét. Ezek után szükség van a marker sarkainak a pontjaira, ezeket a marker egy oldalának hosszából számolhatjuk ki. Ezt muszáj manuálisan lemérni a markeren és az eredményt beállítani az algoritmus számára. Ha ez megvan akkor futtatható a marker felismerő algoritmus, ami csak a megadott szótárból származó markereket fogja elfogadni, és ha olyat talál akkor azoknak a sarkait és azonosítóit visszaadja. Ezt követően a beállított marker pontjai, a talált sarkok, a kamera mátrix, és a torzítási együtthatók alapján futtatható a marker helyzetét meghatározó algoritmus, ami kiszámítja a valós világban lévő markerek közepének koordinátáit a kamerához viszonyítva [8]. Ahhoz, hogy ezt egy Mixed Reality eszköznél is alkalmazni lehessen, szükség van az eszközön található fedélzeti szenzorok által mért adatokra is, melyeket a marker helyzetét meghatározó algoritmus kimenetével transzformálva kiszámítható a marker valós térben lévő helyzetének megfelelő virtuális térben lévő helyzete. (pozíció, orientáció) A fedélzeti szenzorok közül általában az eszközön található IMU-ban lévő gyorsulásmérőre, giroszkópra és magnetométerre van szükség, melyeknek segítségével meghatározható a kamera valós világban lévő helyzete.

5. A HARDVER

5.1. ábra Microsoft HoloLens 2 [https://www.microsoft.com/en-us/hololens]

5.1 Microsoft HoloLens 2

A HoloLens 2 egy újra gondolt verziója az első generációs HoloLens-nek, elődjénél könnyebb, ergonomikusabb és erősebb hardverrel rendelkezik. A dizájn középpontjában a felhasználói kényelem és funkcionalitás áll. Az eszköz gyakorlatilag egy szemüveg, ami egy állítható pántra van szerelve mely a felhasználó feje köré illeszkedik. Maga a szemüveg része felhajtható, így a felhasználó könnyedén válthat a valós világ és a kevert valóság között. Az eszköz fejpántjába hangszórók vannak építve melyek biztosítják a térbeli hangzást, de nem nyomják el a valós világot [10].

5.1.1 Megjelenítő rendszer

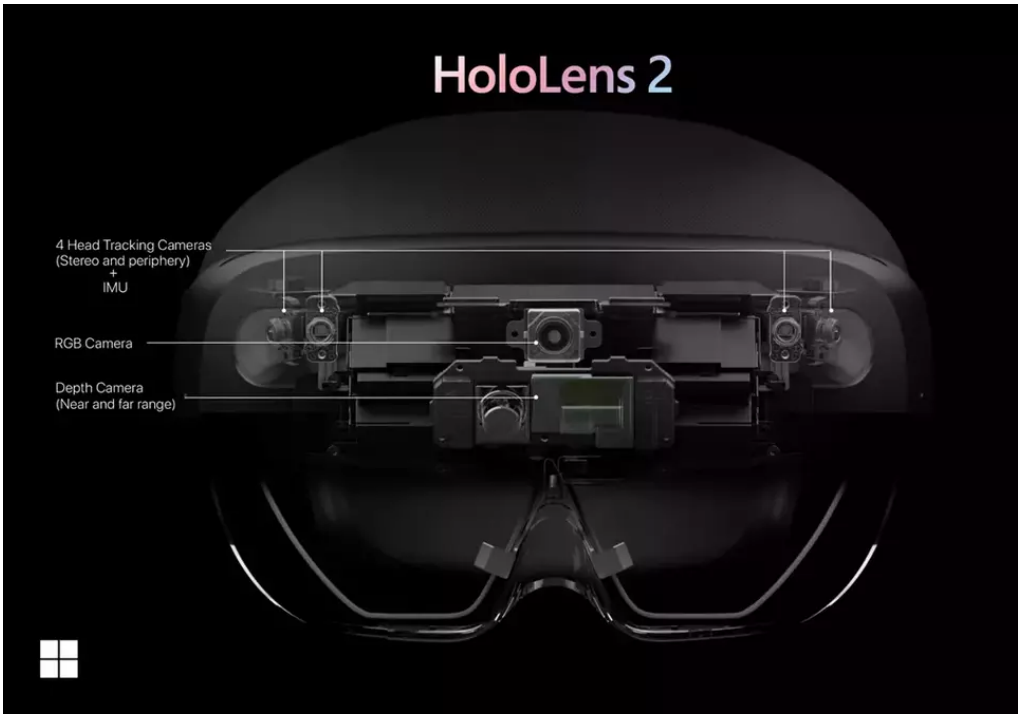
A HoloLens 2 két darab 1440x936 pixeles felbontású 3:2-es képarányú holografikus lencsével van felszerelve melyek 60Hz-es képfrissítéssel működnek. A headset vízszintesen 43 fokos, függőlegesen 29 fokos látómezőt biztosít. A megjelenítő rendszer LBS technológiát használ, hullámvezetők és fényprojektorok kombinációjából áll [11]. A felhasználó az eszköz viselése közben a hullámvezetőkön (holografikus lencséken) néz keresztül melyeken a kép is megjelenik, így egyszerre láthatja a valós és virtuális világot [12].

5.1.2 Fedélzeti szenzorok

A szemüveg számos érzékelővel rendelkezik, többek között egy IMU-val, ami tartalmazza a gyorsulásmérőt, a giroszkópot és a magnetométert [13]. Az eszköz elején található a mélység érzékelő kamera, ami két különböző üzemmódban tud működni:

- AHAT: nagy sebességű (45 FPS) közeli mélység érzékelésre alkalmas, ezt használja az eszköz a kezek követésére [13].
- Long Throw: alacsony sebességű (1-5 FPS) mód távoli mélység érzékelésére, ezt használja a környezet feltérképezésére [13].

Mindkettő üzemmódban infravörös fénnyel világítja a környezetét az eszköz, ami miatt a környezeti fényviszonyoktól függetlenül is képes a kezek követésére és a fizikai világ feltérképezésére [13]. Az headset elején jobb és bal oldalt helyet kapott négy darab fekete-fehér kamera egyenként 640x480-as felbontással (30 FPS), melyeket a rendszer a valós idejű viSLAM-hez használ [13]. Az előlap közepén, közvetlenül a mélység kamera fölött helyezkedik el a PV kamera 1080p felbontással (30 FPS) ami egyszerűen hozzáférhető különféle multimédiás alkalmazások számára.



5.2 ábra Microsoft HoloLens 2 fedélzeti szenzorok [https://learn.microsoft.com/en-us/hololens/hololens2-hardware]

5.1.3 Feldolgozó egységek

A HoloLens 2 középpontjában a Qualcomm Snapdragon 850-es Compute Platform áll, amihez a Microsoft egy saját fejlesztésű holografikus feldolgozó egységet (HPU) tervezett [12]. Az eszköz CPU architektúrája eltér az asztali és mobil számítógépekben megszokottaktól, ennél egy 64-bites RISC (ARM) architektúrájú processzort használtak ellentétben az előző generációs HoloLens-el, ami CISC x86 volt. Ezen módosítások miatt minden szoftvernek, vagy komponensnek, amit a szoftver használ ARM kompatibilisnek kell lennie, hogy futtatni lehessen rajta. Az eszköz egy külön kiadását használja a Windows 10-nek, amit Windows Holographic OS-nek neveznek. Fontos megjegyezni, hogy a HoloLens 2-re fejlesztett alkalmazások UWP Appok, amelyek létrehozásához Visual Stúdióra és Windows SDK-re van szükség és jelenleg nem lehet rá Windows 10 vagy Windows 11-en kívül más operációs rendszeren alkalmazást fejleszteni.

6. A SZOFTVER

6.1 Szükséges technológiák, komponensek áttekintése

Ebben a részben bemutatásra kerülnek azok a szoftverek, könyvtárak, technológiák, komponensek, amik a projekt fejlesztése és működése során elengedhetetlenek.

6.1.1 CMake

A CMake egy nyílt forráskódú build rendszer és konfigurációs eszköz melyet a kitware fejleszt. Kifejezetten komplex szoftver projektekhez készítették, ahol több fejlesztő dolgozik különböző platformokon. Lehetővé teszi, hogy platformfüggetlen és hatékony módon határozzák meg a build folyamatot konfigurációs fájlok (CMakeLists.txt) létrehozásával, amelyek leírják hogyan kell a projektet generálni és összerakni [14].

6.1.2 OpenCV

Az OpenCV eleinte az Intel Research Labs kezdeményezésének a részeként jött létre CPU intenzív alkalmazások fejlesztésére. Az Intelnek dolgozó egyik fejlesztő észrevette, hogy nagy egyetemeknél pl.: MIT Media Lab, jól kidolgozott és az egyetemeken belül szabadon elérhető gépi tanulásra használt szoftver könyvtárakat használnak a fejlesztés felgyorsítására. A kódot a hallgatók megosztották egymás között és jelentős előnyt biztosított a gépi látást (CV), gépi tanulást (ML) kiaknázó alkalmazások kifejlesztésénél mivel az alap függvényeket nem kellett mindig újra megírniuk. Az Intelnél az OpenCV fejlesztőcsapatának az elsődleges célja az volt, hogy a kereskedelmi gépi látás, tanulás alkalmazások fejlesztését felgyorsítsák azáltal, hogy közös infrastruktúrát biztosítanak, amelyre mindenki építkezhet. A gépi látás és tanulás alapú alkalmazások, valamint a hordozható kód ingyenessé és elérhetővé tétele növelte a gyors processzorok iránti igényt. Ez a cég fő üzletága és a gyorsabb processzorokra való átállás több bevételt hozott. Mivel nyílt forráskódúvá tették a szoftver könyvtárat, így rengeteg ember hozzáférhetett és módosíthatta. Később a központi fejlesztés az Intelen kívülre helyeződött, majd 2012 augusztusában a projekt az OpenCV nonprofit alapítványhoz került. Az OpenCV-t C és C++ programnyelven írták és elérhetővé tették minden elterjedtebb operációs rendszerre

(Android, iOS, Windows Linux, MacOS). Több mint 2500 algoritmust tartalmaz, nagyon átfogó dokumentációval, forrás kóddal, és példa programokkal a valós idejű gépi látás kipróbálásához. Az első megjelent verziója óta (2000, BSD licensz) a nyíltforráskódú szoftver keretrendszer nagyon sok kivételes alkalmazásban, termékben, kutatásban játszott kulcsfontosságú szerepet. Ezen alkalmazások között találunk műhold kamerák képeinek összeillesztését, webes alapú térképeket, orvosi berendezések felvételén keletkezett zajcsökkentést, tárgy felismerést, biztonságtechnikai berendezéseket, gyártást felügyelő rendszereket, kamera kalibrációt, katonai védelmi rendszereket, pilóta nélküli légi, földi és vízalatti járműveket. Az OpenCV rendkívül hatékony és nagy teljesítményű képfeldolgozást tesz lehetővé, valamint már 2011 óta támogatja a GPU gyorsítókat és az NVIDIA CUDA-t. Bemeneti adatként nemcsak képeket, hanem videó forrást is képes kezelni, vagy akár több kamera kimenetét is kombinálni tudja. Mivel a gépi tanulás elengedhetetlen a gépi látáshoz, az OpenCV tartalmaz egy teljeskörű gépi tanulást (ML) megvalósító könyvtárat is, ami statisztikai minta felismerésen és klaszterezésen alapul. Rengeteg opcionális modullal rendelkezik melyek szintén nyíltforráskódúak és tetszés szerint hozzáadhatóak a szoftver könyvtárhoz [9].

6.1.3 Unity

A Unity egy 2D és 3D játékmotor (game engine) amit a Unity Technologies fejleszt 2005 óta azzal a céllal, hogy egy komplett játék fejlesztéséhez szükséges környezetet biztosítson. Hosszú élete során a játékmotor drámai változásokon és bővítéseken esett át, hogy sikeresen lépést tudjon tartani a játékipar rohamos fejlődésével. A cég célkitűzése napjainkban sem változott, a lehető legstabilabb eszközkészletet kívánja nyújtani elsősorban a játékipar számára és mindezt egyszerű megoldások és átlátható kezelő felület mellett azért, hogy a kezdő fejlesztők is könnyen bele tanulhassanak. A játékiparon kívül az oktatásban, filmiparban és az autóiparban is alkalmazzák, emiatt a cég nagy hangsúlyt fektet a valós idejű 3D technológia fejlesztésére is. Nagyon átlátható módszert kínál a játék architektúrájának az összeállításához. Egy Unity projektben minden szint egy szcenárióra (scene) van osztva és minden szcenárió tartalmazza az összes játék objektumot melyek a játékos számára szükségesek a szint használatához legyen az a pálya, a karakter, ellenség, lövedék... A Unity lehetővé teszi, hogy a hierarchiában lévő

objektumok között szülő-gyermek kapcsolat legyen, így nagyon egyszerűen lehet több objektumot (ruhát, fegyvert, érzékelőket, vezérlőket) rendelni egy szülő játékos objektumhoz, amelyből a játékost leszármaztathatjuk. Egy külön panelen keresztül (inspector) gyors hozzáférést biztosít az objektumok tulajdonságaihoz és így azokat a játék futása közben is bármikor meg lehet változtatni anélkül, hogy a forráskódban kellene módosításokat végezni. Mindezek mellett fejlett Scripting API-al rendelkezik, amivel hozzá lehet férni a leggyakrabban használt függvényekhez és ezekhez részletes dokumentációt is nyújt melyek együttese lényegesen megkönnyíti a fejlesztő dolgát. Lényegében, amit az inspectorból be lehet állítani, azt a Scripting API kódból is elérhetővé teszi. A játékmotor több platformra képes futtatható állományt előállítani (Cross-Platform Build Suppor), a fejlesztőnek csak le kell töltenie a kívánt platformhoz szükséges környezetet (development kit) és máris exportálhatja többek között Windows, Linux, MacOS, Android, iOS, PS4, PS5, Xbox One vagy akár Microsoft HoloLens-re is a játékot. Még a HTML5-t is támogatja, ami azt jelenti, hogy webes játékok készítésére is alkalmas. Amikor VR-ról és AR-ról vagy MR-ról van szó, amelyek jelenleg a legújabb technológiák, a Unity az egyik legnagyobb támogatója ezeknek. VR esetében számos olyan kiegészítő csomag áll rendelkezésre a játékmotorhoz, ami támogatja szinte az összes elérhető VR headsetet és ezeket folyamatosan frissítik, hogy lépést tartsanak a technológia rohamos fejlődésével. AR esetében a Unity készített egy AR Foundation kiegészítőt is, amivel a fejlesztők egyszerre készíthetnek AR alkalmazásokat Android és iOS platformokra és nincs szükség különálló projektekre. Később a kettőt ötvözték és létrejött az XR Interaction Toolkit ami még jobban leegyszerűsítette a VR és AR játékok fejlesztését [15].

6.1.4 Windows Runtime Komponensek

A Microsoft Windows rendszerek egy alkalmazásprogramozási felületet (API) biztosítanak a programok számára, amely segítségével a programnak nem feltétlenül kell ismernie a hardvert, hogy el tudja végezni a feladatát. Ezt a felületet Win32 API-nak is nevezik, funkciókból és adatszerkezetekből áll, amik több DLL által kerülnek exportálásra [16]. Amikor a Win32 API-t tervezték, a legtöbb program még vagy C vagy C++ programnyelveken íródott. Létrehoztak egy COM felületet annak érdekében, hogy a

futás idejű verziókezelést ugyanazon alapvető ABI segítségével lehessen kezelni. Ezt már C++ nyelvből sokkal könnyebb volt programozni, viszont technikailag még mindig sok helyen C-t használt makrókon keresztül. Később megjelentek a menedzselt programnyelvek (.NET és hozzá hasonlóak) és ezek már más hívási mechanizmusokat alkalmaztak. El lehetett érni a COM-ot .NET programnyelvekből is viszont ez nem volt túl egyszerű, egyre több wrapper-t hoztak létre a C/C++ alkalmazásprogramozási felületek és típusok eléréséhez. A C++/CX, C++/WinRT, WRL célja alapvetően ugyanaz, mégpedig mechanizmust biztosítani a Windows Runtime stílusú API-ok és típusok C++ programnyelvből történő hívásához és a Windows Runtime stílusú API-ok és típusok írásához [17]. A C++/WinRT-vel a Microsoft ötvözte a COM jellemzőit és a .NET metaadat szerű megközelítését. Ennek előnye, hogy egy API-t egyszer kell megírni C++/WinRT-ben és utána azt az összes Windows Runtime programnyelvű alkalmazásból lehet használni [18]. A Windows Runtime Komponens egy olyan önálló szoftvermodul, ami bármely Windows Runtime programnyelven (C#, C++/WinRT, C++/CX, Visual Basic, Javascript) írt alkalmazásból elérhető és használható [19]. A Microsoft által gyártott fejlesztői környezett (Visual Stúdió) segítségével lehet létre hozni ilyen modulokat, amiket utána vagy Windows App SDK-t használó alkalmazások, vagy UWP alkalmazások használhatnak.

6.1.5 Dinamikus Csatolású Könyvtárak (Dynamic Link Library)

Szintén Microsoft Windows specifikus megoldás, olyanok, mint az EXE fájlok, de nem közvetlenül futtathatóak. A DLL-ek a megosztott program könyvtárak Microsoft általi implementációi. Annyira hasonlítanak az EXE fájlokhoz, hogy a formátumuk ugyanaz (Portable Executable). Általában COM komponenseket, .NET könyvtárakat tartalmaznak, de olyan függvényeket, osztályokat, változókat, esetleg felhasználói felületeket, erőforrásokat is tartalmazhatnak, amelyeket egy EXE vagy egy másik DLL használ. Minden operációs rendszeren kétfajta könyvtár létezik, statikus és dinamikus. Windows-nál a statikus „.lib”, a dinamikus „.dll” kiterjesztést használ. A legfontosabb különbség, hogy a statikus könyvtárak a program fordításakor kapcsolódnak a futtatható állományhoz (EXE) míg a dinamikusan csatolt könyvtárak csak futáskor. A statikus könyvtárakat általában nem látjuk a számítógépen mert azok közvetlenül egy EXE-be

vagy DLL-be vannak beágyazva. A DLL (dinamikus könyvtár) egy önálló az EXE fájloktól független fájl, bármikor módosítható és csak akkor töltődik be futáskor, ha egy EXE betölti. Ennek előnye, hogy a DLL-ben lévő kód egyszerre több futó programban is felhasználható, ami növeli a memória hatékonyságot mert ha például 10 alkalmazás használja ugyanazt a DLL-t egyidőben, akkor annak a DLL-nek csak egy példánya lesz betöltve a memóriába. Egy statikus könyvtárat (.lib) nem lehet megváltoztatni, ha már lefordították az EXE-ben. A DLL fájlok külön-külön frissíthetőek az EXE fájl frissítése nélkül. Amikor indul egy program, akkor a DLL-t a Win32 API a LoadLibrary hívásával tölti be [20].

6.2 Koncepció, Működési elv

Az projekt alapjául a Unity-t választottam, többek között az egyszerűsége és kifinomultsága miatt, valamint mert támogatja a Microsoft HoloLens 2 eszközökre való alkalmazás fejlesztést. A szoftvercsomag részét képezi a marker felismeréshez és a kamera kalibrálásához szükséges Unity projekt, valamint a különféle cli segéd programok, amik a HL2-től jövő információk egy másik számítógépen való megjelenítéséhez, feldolgozásához és a nyomtatható ArUco markerek generálásához biztosítanak megoldásokat. A Unity projekt 2 szcenárióból áll, az egyik a markerek felismerését és követését valósítja meg, a másik pedig a kamera kalibrációs folyamathoz szükséges. A marker felismerés megvalósításához egy egyedi OpenCV 4.8 buildet használtam az ArUco kiegészítő modullal. A Unity és az OpenCV közötti kommunikációt egy külön Windows Runtime Komponens biztosítja (OpenCVBridge).

6.2.1 ArUcoTracking szcenárió

Ez a szcenárió felelős az ArUco markerek felismeréséért, helyzeteinek meghatározásáért és azok pozícióján a felhasználó által kiválasztott virtuális objektumok megjelenítéséért. Az inspector menüből az aktuális szcenárióhoz társított scriptek által exportált paraméterek állíthatóak be melyek jelen esetben:

1. ArUco Szótár (Dictionary): ebből generálták a markert, formátuma a következő: „DICT_6X6_100”, ahol a „6X6” jelenti a bitátmenet számát, a 100 pedig a szótár

méretét, ennyi marker van benne, amit maximum generálni lehet. Fontos, hogy minél nagyobb a bitátmenet, annál jobban azonosítja az algoritmus, viszont ez növeli a feldolgozási időt is a felismerésnél.

2. Marker Go: ez adja meg milyen virtuális objektum (3D modell) jelenjen meg az észlelt markeren.
3. Media Capture Profile: a HoloLens-en különböző profilok alapján lehet konfigurálni a PV kamerát. A videó profilok közül néhányat kiválasztottam és a felbontásaik alapján csoportosítottam. Fontos, hogy minél nagyobb a felbontás annál több pixelt kell feldolgoznia majd az OpenCV-nek ami lassítja a markerek felismerését és növeli a CPU terhelését, ez kihatással lesz az alkalmazás futási teljesítményére.
4. Auto Release Marker Gos: ez egy kapcsoló, ha engedélyezve van akkor amint egy markert letakarunk a kamera elől, az ahhoz tartozó virtuális objektum példányát törli és az eltűnik. Ha nincs engedélyezve akkor a letakart markert jelölő virtuális objektum az utolsó kiszámított helyzetben (pozíció, orientáció) marad mozdulatlanul.
5. Send Detected ArUco Data via TCP: kapcsoló, ha engedélyezve van és az inspectorban be van állítva a számítógép IP címe és Port-ja, amin a TCPServer.py program fut, akkor továbbítja az észlelt markerek adatait (azonosító, pozíció, orientáció) a cli programnak, ami azt kiírja. Ez további feldolgozást és adatrögzítést tesz lehetővé.
6. Use custom camera intrinsics: szintén egy kapcsoló, ha engedélyezzük akkor a kamera kalibrációs szcenárióból nyert képeken lefuttatott „Calibrate_PhotoVideo.py” program kimeneti értékeit beállítva az algoritmus pontosabban fogja meghatározni a markerek helyzetét. Ha nem akkor az alkalmazás megpróbálja API-okon keresztül elérni az aktuális értékeket az adott kamera képkockához.
7. Enable DBG Display: ez a kapcsoló engedélyezi a valós idejű szöveges kimenet megjelenítését, ilyenkor a felhasználó láthatja, hogy az alkalmazás éppen mit csinált, melyik folyamatnál jár. Elsősorban hibakeresésre szolgál.

A program indulásakor készít egy referencia koordináta rendszert, amit eltárol egy változóban, később ehhez képest lesznek majd a felismert ArUco markerek transzformálva. Fontos megjegyezni, hogy ez körülbelül egy $2m^2$-es területen belül fog működni, ha a felhasználó ezen kívülre sétál akkor a hologramok elcsúsznak és az eszköznek új referencia koordináta rendszert kell felvennie. (újra kell indítani az alkalmazást) Következő lépésben elindítja a ArUco követés szcenárió inspectorában beállított paraméterekkel az eszköz kameráját. Ha a szenzor sikeresen elindult, akkor az adatfolyam továbbításra kerül egy híd projektnek (OpenCVBridge) ami arra szolgál, hogy az OpenCV szoftverkönyvtár és a Unity alkalmazás (HoloLens2CVUnity) között lebonyolítsa a kommunikációt. Az OpenCV markerfelismerő algoritmusa minden képkockán végig megy és ha talál markereket akkor lefut a pozíció meghatározó algoritmus is, ami egyesével meghatározza az aktuális képkockán található markerek helyzetét. Ezek után a híd projekt visszaad egy listát a Unity alkalmazásnak, ami tartalmazza az összes észlelt markert és azok helyzeteit. A Unity alkalmazás ezt követően a különböző típusok és koordináta rendszerek közötti konverziók, mátrix transzformációk után egyesével megjeleníti a markerek pozícióján az inspectorban kiválasztott virtuális objektumot.

6.2.2 CameraCalibration szcenárió

Ez a szcenárió a kamera kalibrációs folyamat egyik fontos részét képezi. Valós idejű képet jelenít meg a felhasználó számára a kameráról, mely segítségével a kalibrációs mintáról könnyedén lehet képeket készíteni különféle nézőpontokból. Ezeket a képeket továbbítja egy másik számítógépre, ahol a kamera paraméterek a mellékelt cli programok segítségével kiszámításra kerülnek. Az inspector menüből az alábbi paraméterek állíthatóak:

1. Media Capture Profile: Akárcsak az ArUco követés szcenárióná, itt is különböző profilok alapján lehet konfigurálni a PV kamerát, ezeket felbontás alapján csoportosítottam.
2. Server IP, Port: itt kell beállítani a számítógép IP címét és Port-ját, amin a TCPServer.py cli program fut.

Ennél a szcenáriónál nincs szükség a referencia koordináta rendszer felvételére indításkor, mivel nem kell semminek meghatározni a helyzetét, csak a szenzortól jövő adatfolyamot kell valós időben megjeleníteni a felhasználó szemei előtt. A program indulása után elindítja az inspectorban beállított paraméterekkel a kamerát és a hálózati klienst. Ha sikeresen elindult a szenzor, akkor az adatfolyamot megjeleníti a felhasználó szeme előtt és kiépül a kapcsolat az azonos alhálózaton lévő számítógéppel, amin a TCPServer.py cli program fut. A HL2-vel a kalibrációs táblát nézve többféle szögből kell képeket készíteni. Amikor képet készítünk akkor azt először szétbontja majd továbbítja a TCPServer.py programnak, ami újra összerakja és a számítógépre menti azt. Ajánlott legalább 30-40 darab képet készíteni az algoritmus optimális működése érdekében. Ha ez megvan akkor futtatható a „Calibrate_PhotoVideo.py” script, ami kiszámítja a kamera paramétereket és kiírja egy fájlba. Ezeket a paramétereket beállítva az ArUcoTracking szcenárió inspectorában a marker pozíció meghatározó algoritmus pontossága növelhető.

6.3 OpenCVBridge Projekt

A Unity alkalmazás és az OpenCV szoftverkönyvtár között szükség van egy komponensre, ami biztosítja a kommunikációt és tartalmazza a kiegészítő függvényeket, amiken keresztül a Unity alkalmazás eléri az OpenCV szoftver könyvtárat. A projekt Visual Studio 2022-vel készült, típusa Windows Runtime Komponens (C++/WinRT), és kimenete egy dinamikusan csatolt könyvtár („.dll”) a hozzá tartozó metaadat fájlal („.winmd”). A projektbe NuGet csomagként telepítve van az OpenCV 4.8 egyedileg buildelt verziója, ami tartalmazza az ArUco kiegészítő modult. Windows Runtime Komponenseknél a Visual Stúdióban úgynevezett IDL („.idl”) fájlok alá vannak csoportosítva a szokásos C++ header és implementációs fájlok. A Microsoft Interface Definition Language egy egyszerűsített, modern szintaxis Windows Runtime típusok definiálására az IDL fájlokban. Az IDL-ben definiált típusokat utána többféle, a WRL által támogatott programnyelveken lehet implementálni [21].

6.3.1 DetectedMarker.idl

Ez az adatstruktúra, ami az ArUco markert reprezentálja. Tartalmazza a marker azonosítóját és helyzetét, valamint a változók írásához és lekérdezéséhez szükséges függvények definícióját. Mivel ezt majd használni fogjuk egy másik programból, (Unity alkalmazás) így ez egy „runtimeclass" is egyben. Fontos, hogy az IDL-ben a C++ típusok WinRT megfelelőjét kell definiálni, például a marker helyzete a C++ forrásfájlokban Float3-ként van deklarálva, de az IDL-ben Vector3-at kell használni helyette, annak ellenére, hogy mindkettő egy 3D vektort jellemez. Ennek az az oka, hogy eltérő névtérből vannak és az IDL-ben csak WinRT által támogatott névtérből származó típusok lehetnek [22]. Az IDL fájl alá tartozó DetectedMarker.h fájl tartalmazza a változók és függvények definícióit, a DetectedMarker.cpp fájl pedig ezek implementációit.

6.3.2 OpenCVHelper.idl

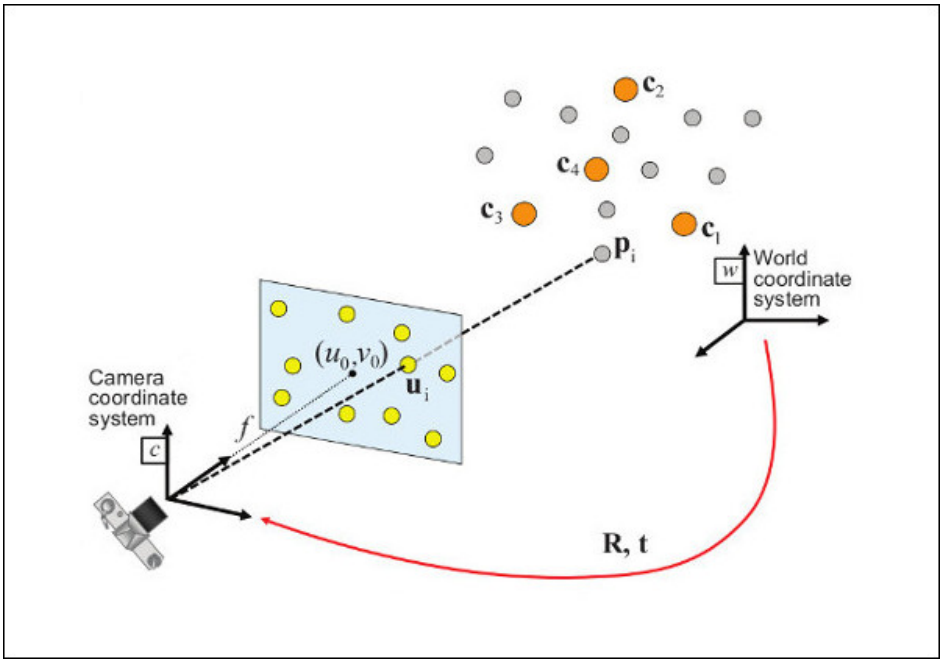
A projekt legalapvetőbb része és szintén „runtimeclass", ez tartalmazza a fő függvény definícióját, ami bemeneti változóként megkapja a kamera képet, a kamera paramétereket, a keresett markerek szótár (dictionary) azonosítóját, és a nyomtatott marker oldalainak a hosszát majd visszaad egy listát a talált markerekről és azok adatiról. A függvény visszatérési értéke egy DetectedMarker típusú IVector, ami lényegében egyfajta WinRT kollekció (lista) [23]. Fontos, hogy az IDL fájl elején importálni kell a DetectedMarker.idl-t azért, hogy azt használni lehessen a kollekció típusaként. Az IDL fájl alá tartozó OpenCVHelper.h és OpenCVHelper.cpp fájlokban vannak a függvények definíciói és implementálációi C++ programozási nyelven. Összesen 3 függvény van definiálva és deklarálva, viszont csak egyet kell külső alkalmazásból elérni, így csak annak az egynek szerepel a definíciója az IDL fájlban (ProcessWithArUco()), a másik kettő a kamera kép konvertálásban játszik kulcs fontosságú szerepet és csak a Windows Runtime Komponensen belül szükséges.

A ProcessWithArUco() függvény először a kapott kamera paraméterekből felépíti a kamera mátrixot majd a torzítási együtthatók mátrixát. A kamera mátrix egy 3x3-as mátrix jelen esetben, ami tartalmazza a fókusztávolságot és a kamera középponti koordinátáit. A torzítási együtthatók mátrixa 1x5-ös (5 vektort tartalmaz), és a kamera

által okozott torzulást modellezi. Ezek ismerete elengedhetetlen az ArUco markerek helyzetének a meghatározásához. Ha ez megvan akkor felveszi egy mátrixba a marker 4 sarkának pontjait, ez lesz a marker koordináta rendszerének mátrixa, melyet a bemeneti paraméterként kapott markerhosszból számít ki. (manuálisan kell lemérni a nyomtatott markeren és beállítani a Unity projekt inspectorában)

A következő lépésben a kapott kamera kép konvertálása történik, erre azért van szükség, mert a HoloLens által biztosított API egy tömörítetlen bittérképet (SoftwareBitmap) ad vissza, ami az OpenCV-nek nem megfelelő. Két kiegészítő funkcióhoz kerül a bittérkép. Először pointereket állít be a pixelekhez és betölti a bittérkép méretét egy változóba, utána zárja a puffer és a memóriában lévő adatról csinál egy hivatkozást. A bemenetként kapott bittérkép BGRA8-as formátumú képet tartalmaz, ami azt jelenti, hogy színek sorrendje kék, zöld, piros, és van egy alpha érték, ami a pixelek átlátszóságát jelöli. A 8-as szám a bitmélységet adja meg. Minden egyes szín 1 bájtot tölt ki, ehhez jön még az alpha érték, ami szintén 1 bájt, így összesen pixelenként 4 bájtra (32bit) van szükség [24]. A puffer tartalmát betölti egy speciális mátrixba ami az OpenCV szoftverkönyvtár része (cv::mat). A mátrix típusneve „CV_8UC4” ahol 8 a pixelérték, U az elem típusa (unsigned integer), C4 pedig a 4 csatorna, tehát összesen 32 bit pixelenként ami képes lesz tárolni a BGRA8-as képet [25]. Az ArUco markereket felismerő algoritmus nem igényel színes képeket, de a bemeneti kép (mátrix) jelenleg még tartalmazza a szín információkat így azt egy beépített függvénnyel (cv::cvtColor()) egy másik mátrixba átkonvertálja, hogy csak fekete-fehér (1 csatornás) képet tartalmazzon. A marker felismerő algoritmus alapbeállításokkal és a bemeneti változóként kapott ArUco szótár azonosítója alapján megkeresett szótárral kerül inicializálásra és utána elindul a markerek keresésének folyamata (DetectMarkers()). Ha az algoritmus markert talál akkor visszaadja a négy sarkának pozícióját és az azonosítóját. Első lépésben szögletes formákat keres, amik markerek is lehetnek. Ezt szegmentálással éri el, egy adaptív küszöbérték tartományt határoz meg majd az ezzel az érték tartománnyal ellátott képből kontúrokat von ki és azokat a formákat, amik nem konvexek vagy nem közelítenek egy négyzet alakzathoz eldobja [26]. A folyamat a maradék forma belső kódolásának az elemzésével folytatódik, ebből kiderül, hogy valóban markerekről van-e szó. Ezután készít egy perspektivikus transzformációt, hogy a markert kanonikus formába hozza, majd küszöbértékkel látja el,

hogy a fekete és fehér biteket szét lehessen választani. A képet ezután felosztja cellákra a marker mérete és a szegély mérete alapján. Az itt kapott cellákban lévő fekete vagy fehér pixeleket megszámolja és a mennyiségek alapján meghatározza, hogy fehér vagy fekete bitről van-e szó. A bitek elemzése után megállapítja, hogy az adott szótárba (dictionary) tartozik-e a marker, ha igen akkor eltárolja a talált sarkokat és a marker azonosítóját [27]. Ha több markert talált mint 0 akkor lefut minden egyes markerre a helyzet meghatározó algoritmus (SolvePnP) ami bemeneti paraméterként megkapja a marker koordináta rendszer mátrixát, a marker felismerő által talált sarkokat, a kamera mátrixot, a torzítási együtthatók mátrixát, végül kimenetként visszaadja az adott markerhez tartozó eltolási és forgatási vektorokat, amelyek jelen esetben az objektum (valós világ) koordináta rendszerében kifejezett 3D pontot a kamera koordináta rendszerbe transzformálják [28].



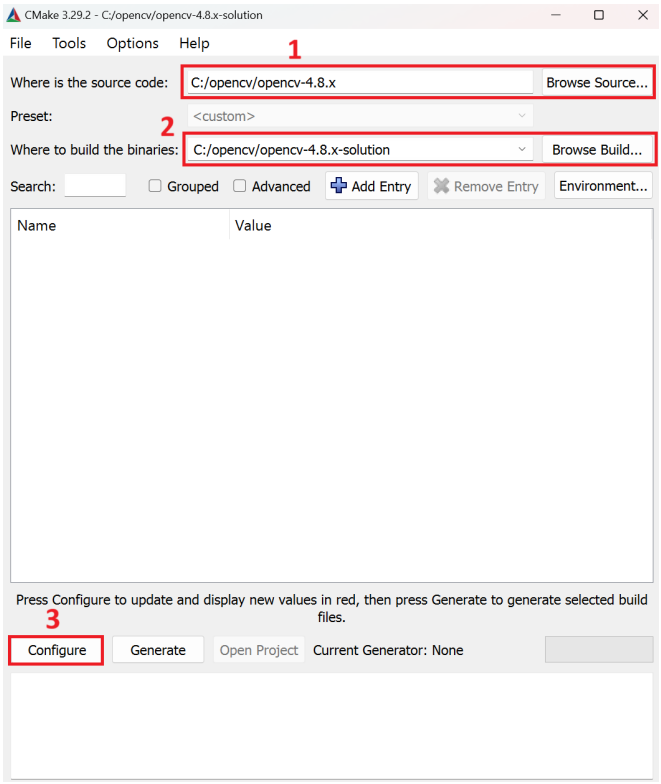
6.1. ábra Perspective-n-Point (PnP) pose computation [https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html]

Ebből következik, hogy a markerhez viszonyított kamera aktuális helyzete egy 3D transzformáció az objektum koordinátarendszeréből a kamera koordinátarendszerébe, melyet az eltolási és forgatási (rvec, tvec) vektorok határoznak meg. Ezeket minden egyes

markerhez kiszámolja az algoritmus. Utána végig iterál a talált marker azonosítókon és minden egyes azonosítóhoz tartozó markert a DetectedMarker osztály egy új példányához rendeli, amibe betölti az adott marker azonosítóját, a hozzátartozó eltolási vektort és a forgatási vektort, majd hozzáadja az IVector-hoz, ami a függvény visszatérési értéke lesz.

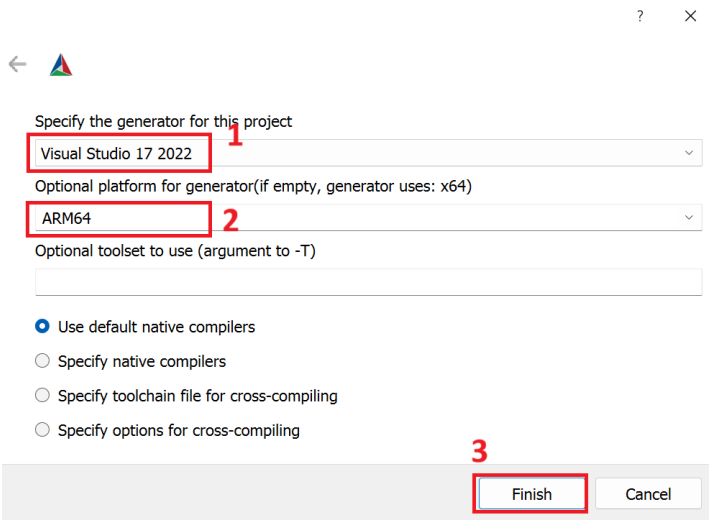
6.3.3 OpenCV build folyamata és telepítése a projektbe

Az OpenCVBridge projektbe az OpenCV 4.8-as verziója van NuGet csomagból telepítve, így az összes speciális névtér és függvény, amit tartalmaz elérhető a Windows Runtime Komponensen belül. A NuGet a Microsoft Visual Stúdióban használt csomag kezelő rendszer, ami lehetővé teszi a fejlesztők számára, hogy előre megírt szoftver könyvtárakat osszanak meg vagy importáljanak a saját alkalmazásaikba. Miután letöltöttük az OpenCV forráskódját GitHub-ról, a CMake program segítségével tudunk belőle Visual Stúdió projektet generálni [29]. A HL2-őn futó alkalmazás az ArUco modult is használja, ami az OpenCV contribution moduljait tartalmazó Repo-ban van [30].



6.2 ábra CMake könyvtárak beállítása

Ha ezeket letöltöttük, akkor először a CMake programban be kell állítani a forráskódot tartalmazó mappát és a generálandó projekt kimeneti mappáját. Ezután a „Configure” gombra kattintva felugrik egy ablak, amiben ki kell választani a Visual Stúdió verzióját, amihez a projekt készül és a platform architektúráját, amin az OpenCV-t használni akarjuk. Jelen esetben ez a Visual Stúdió 2022 és az architektúra pedig ARM64.

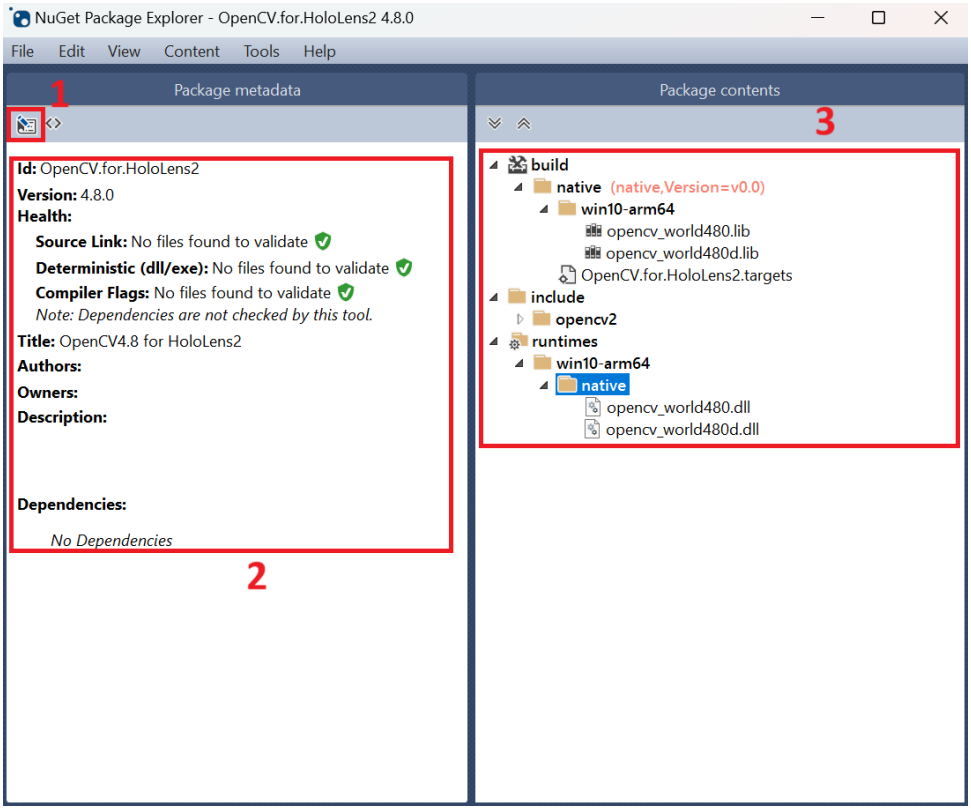


6.3 ábra CMake platform és generátor alkalmazás kiválasztása

Utána az „extra” kulcsszóra rákeresve beállíthatjuk az opcionális modulokat tartalmazó könyvtár elérési útvonalát. Fontos, hogy itt ne a Repo, hanem az azon belüli „modules” könyvtár elérési útvonalát adjuk meg. A „Configure”-ra kattintva most már a beállítások listájába bekerültek az opcionális modulok is. A projektben messze nincs szükség az OpenCV összes funkciójára és moduljára, ezek kikapcsolgatásával lényegesen csökkenthető a szoftver könyvtár mérete és a buildelési idő is. A szakdolgozat mellékletében található egy lista a projekthez használt CMake beállításokról. Ha megvagyunk a beállításokkal akkor a „Generate” gombra kattintva a CMake létrehozza a beállított kimeneti könyvtárba a Visual Stúdió projektet. Ezt megnyitva először be kell állítani a Solution Configuration-t Debug-ra és az Architecture-t ARM64-re, utána a jobb

oldalon a Solution Explorer-ben a CMakeTargets mappában az ALL_BUILD-re jobb egér gombbal kattintva a Build gombot kiválasztva elindítható a szoftver könyvtár moduljainak a buildelése.

Ha ez végzett akkor ugyanitt az INSTALL-ra jobb egér gombbal kattintva és a Build gombot választva indítható a szoftver könyvtár DLL fájlba való csomagolása. Ezeket a lépéseket ismételjük meg úgy is, hogy a Solution Configuration > Release módban van. Ha a folyamatok hiba nélkül végbe mentek, akkor létrehozhatjuk a NuGet csomagot, amibe majd az OpenCV-t beletesszük. Ezt többféleképpen is meg tehetjük, de a legegyszerűbb, ha a NuGet Package Explorer alkalmazást használjuk erre a célra. Miután elindítottuk, válasszuk a „Create a new package” opciót. Állítsuk be a csomag metaadatait, majd a jobb oldalán hozzuk létre a 6.4-es ábrán látható mappa struktúrát:

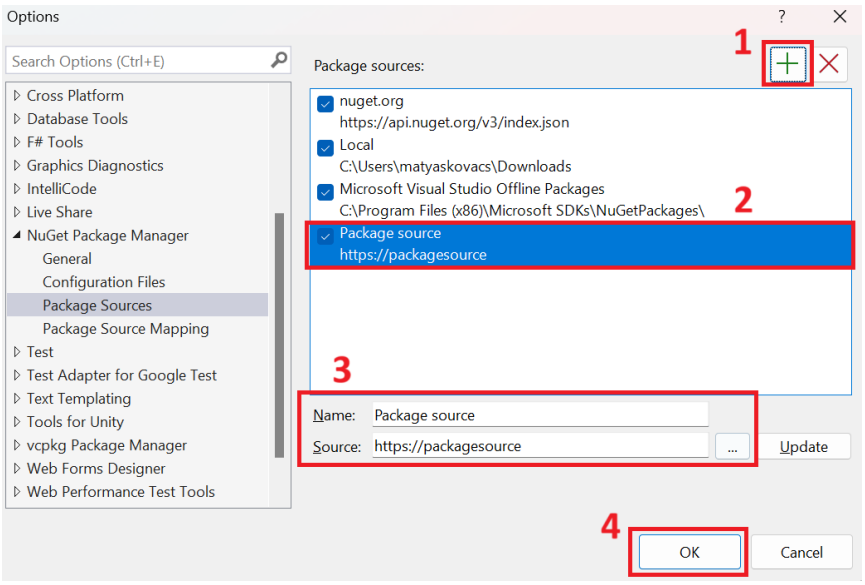


6.4 ábra NuGet Package Explorer új csomag létrehozása

Ha ezek megvannak akkor az ábrán látható mappákba be kell másolni az OpenCV Visual Stúdió projekt által létrehozott fájlok közül néhányat. Az OpenCV DLL fájljai a Visual Stúdió projekt mappában lévő ../install/ARM64/vc17/bin könyvtárban vannak. A statikus könyvtárak az ../install/ARM64/vc17/lib mappában találhatóak. Az include mappa tartalma pedig az ../install/include/opencv2 kell legyen. Ha ezeket bemásoltuk a NuGet csomagba, akkor létre kell hozni az ábrán látható „.targets” fájlt a jobb egér gomb és az „Add New File” kiválasztásával. A fájl tartalma a szakdolgozat mellékletében található. Ezután a File > Export gombot megnyomva létrehozható a NuGet csomag, amely tartalmazza az OpenCV-t.

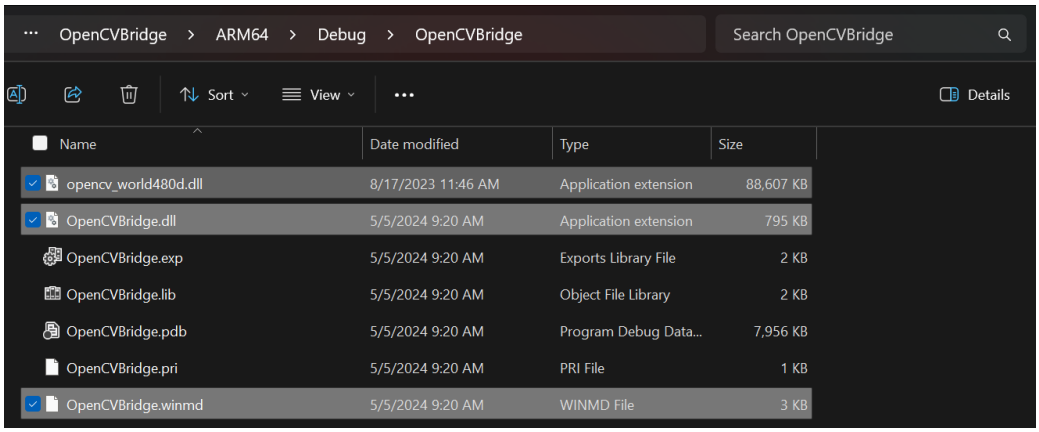
A következő lépésben telepíteni kell a NuGet csomagot az OpenCVBridge projektben. Ha megvan nyitva az OpenCVBridge projekt, jobb oldalon a Solution Explorerben a projektre jobb egérgombbal kattintva a Manage NuGet Packages menüben el kell elvégezni a következő beállításokat:

1. Jobb oldalon a Package Source: nuget.org mellett a beállítások ablak megnyitása
2. + gombbal új Package Source hozzáadása
3. A Package Source nevének és elérési útvonalának beállítása. (azt a könyvtárat kell megadni, ahol a létrehozott NuGet csomag van)
4. Változtatások mentése az OK gombbal



6.5 ábra Visual Studio 2022 NuGet csomag forrás útvonal beállítása

Ha átváltjuk a Package source-t nuget.org-ról az általunk beállítottra akkor bal oldalon meg fog jelenni a frissen létrehozott NuGet csomag, ami az OpenCV-t tartalmazza. A csomagra kattintva jobb oldalon elérhetővé válik a telepítés gomb, ezt kiválasztva elindítható a projektbe való telepítés. Ezek után minden függvény és névtér, amit az adott OpenCV build tartalmaz elérhető a projektből, valamint az OpenCV szoftverkönyvtár DLL fájlja automatikusan bekerül majd a projekt kimeneti könyvtárába buildelésnél, amit az alkalmazás be tud majd tölteni.



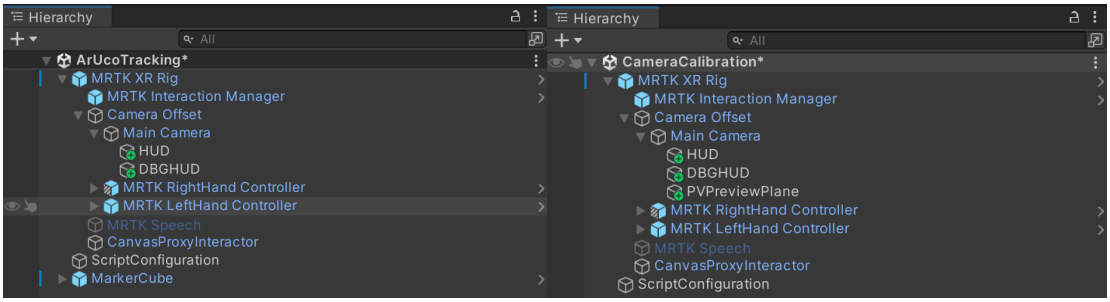
6.6 ábra OpenCVBridge projekt kimeneti fájlok

6.4 HoloLens2CVUnity Projekt

Ebben a részben ismertetésre kerül a fő alkalmazás logikai felépítése, az osztályok, scriptek leírása és működésük bemutatása, valamint megemlítésre kerülnek a projekthez mellékelt kiegészítő programok is.

A projekt fejlesztéséhez Unity 2022.3.3f1-es verziót használtam, Microsoft Visual Stúdió 2022-vel. A Unity C# programnyelvet használ scriptelésre, így az alkalmazás kódja is C#-ban van írva. A projekt, 2 szcenárióból áll:

1. ArUcoTracking
2. CameraCalibration



6.7 ábra Unity projekt szcenáriók

Mindkét szcenárió tartalmazza az MRTK XR Rig prefabot, és egy ScriptConfiguration nevű objektumot. Az XR Rig prefab biztosítja a HoloLens-el való integrációt, ebben található a fő kamera objektum, ami a headset-nek felel meg a szcenárióban, mozgása leképezi a valós térben történő mozgást. Tartalmaz egy bal és jobb kezes vezérlő objektumot is, amik a kezek követésére és mozdulatokra való reagálásra alkalmasak. A fő kamera alatt helyet kapott két szöveges doboz, amik az alkalmazás aktuális folyamatairól nyújtanak információt a felhasználó számára. Ezek a fő kamerával együtt mozognak, mint egy virtuális kijelző. A ScriptConfiguration objektum egy helyfoglaló, ami az adott szcenárió által használt scripteknek biztosít helyet magában a szcenárióban és az inspectorban egyaránt. Az ArUcoTracking szcenárióban van egy plusz objektum, a MarkerCube ami egy 3D kocka modellje és működés közben a felismert markerek tetején jelenik meg. A virtuális kocka oldalának hossza megegyezik a nyomtatott ArUco marker oldalának a hosszával miután azt az inspectorban (ScriptConfiguration) beállítottuk. A kamera kalibrációs szcenárióban lévő PVPreviewPlane objektumra vannak kivetítve a kamera képek, és ez szintén a fő kamerával együtt mozog. A project „Assets” nevű mappájában található az alkalmazás összes erőforrása, mely jelen esetben az alábbiak szerint néz ki:

1. Editor: Ide kerülnek a Unity szerkesztő (Editor) funkcionalitását bővítő scriptek. Ezek csak a szerkesztőben futnak, az alkalmazásba nem kerülnek bele.
2. Materials: Unity-ben amikor valamit meg akarunk jeleníteni akkor információt kell biztosítani a játékmotornak, hogy az adott modell formáját és a felületét hogyan jelenítse meg. A formák leírásához hálókat (mesh) használnak, a

felületekhez pedig anyagokat (material) és az utóbbiakat leíró fájlok kerülnek ebbe a mappába.

3. Plugins: Ebbe kerülnek a külső szoftver könyvtárak, amiket az alkalmazásból szeretnénk használni. Jelen esetben ez tartalmazza az OpenCVBridge projekt DLL és WINMD fájljait.
4. Prefabs: Ide kerülnek a modellek formáit leíró fájlok.
5. Scenes: Ez tartalmazza a Unity számára a szcenário fájlokat, jelen esetben az ArUcoTracking.unity-t és a CameraCalibration.unity-t.
6. Scripts: Itt kapnak helyet az alkalmazás működéséhez szükséges scriptek, amik a kódot tartalmazzák.
7. Packages: Itt találhatóak a projekt számára szükséges szoftver csomagok. Ezek közül rengeteget a Unity előre telepít, viszont van néhány, amit a HL2-őn való működéshez saját magunknak kell telepíteni a projektbe.

6.4.1 Build folyamat és telepítés

Az alkalmazás Microsoft HoloLens 2 eszközökre való exportálásához szükség van néhány kiegészítő szoftver csomag telepítésére és beállítás elvégzésére. Egy Unity projektbe rengeteg kiegészítőt telepíthetünk, viszont itt olyanok is kellenek, amiket a Microsoft Mixed Reality Feature Tool biztosít és ezek közül az alkalmazásnak alábbiakra mindenképp szüksége van:

- MRTK Core Definitions
- MRTK Input
- MRTK UX Components
- MRTK UX Components (Non-Canvas)
- Mixed Reality OpenXR Plugin
- Microsoft Spatializer

A szükséges beállításokat pedig a Unity szerkesztőből (Editor) kell elvégezni. A File > Build Settings menüben Platformnak az UWP-t (Universal Windows Platform) kell kiválasztani, jobb oldalon az Architechture pedig legyen ARM 64-bit. Az Edit > Project Settings menüből az XR Plug-in Management fülön a Plugin Providers alatt az OpenXR-

nek és a Microsoft HoloLens Feature Group-nak kell bekapcsolva lennie. Az XR-Plug-in Management alatt az OpenXR-t kiválasztva a következő menüben 3 interakciós profilra van szükség melyeket a + gombbal tudjuk hozzáadni:

1. Microsoft Motion Controller Profile
2. Microsoft Hand Interaction Profile
3. Eye Gaze Interaction Profile

Szükség van továbbá az alap MRTK profil hozzáadására, amit szintén a Project Settings menüből tudunk megtenni az MRTK3 fül alatt. Mivel külső szoftver könyvtárat is használ a projekt, a Project Settings menüben a Player > Other Settings > Allow 'unsafe' code opciót be kell pipálni. A Unity build folyamat kimenete egy Visual Stúdió projekt, ami tartalmazza a teljes alkalmazás HL2-őn való futtatható állományának létrehozásához szükséges fájlokat.

Fontos, hogy amikor először futtatnánk az alkalmazást a HL2-őn, alapból nincsenek telepítve az OpenCV működéséhez elengedhetetlen C++ kiegészítők. Ezért mielőtt telepítjük az alkalmazást, a Visual Stúdióban jobb oldalon a projektre jobb egérgombbal kattintva Add > Reference > Universal Windows > Extensions menüből a Visual C++ 2015-2019 UWP Desktop Runtime kiegészítőt be kell pipálni.

Az alkalmazást telepítése Visual Stúdióból a HL2-re az alábbi lépésekből áll:

1. Solution Platform ARM64-re állítása
2. A HoloLens2CVUnity (Universal Windows) Project Properties menüjében a Debugging fülön a Machine Name mezőbe a HoloLens 2 IP címének megadása
3. A Remote Machine gombal a folyamat elindítása

Ha az alkalmazás feltelepült, automatikusan el fog indulni és a Unity Build Settings menüben kiválasztott szcenárió fog betöltődni.

6.4.2 Szinkron, aszinkron műveletek a Unity-ben

A szoftverfejlesztésben a szinkron műveletek olyan feladatok végrehajtására utalnak melyek végrehajtása szekvenciális, blokkoló módon történik. Minden művelet meg várja az előző befejezését, a program végrehajtása lineáris, a feladatok egymás után kerülnek

feldolgozásra. A Unity alkalmazások elsősorban egy szálon futnak, amit fő szálnak (game loop, main thread) neveznek. Ez a szál felelős a kritikus műveletekért (fizikai szimulációk, renderelés) és ciklikusan folytatja ezeket a lépéseket fenntartva a játék valós idejű jellegét. Ennek van egy nagy hátránya, ha egy feladat végrehajtása túl sok időt vesz igénybe, akkor amíg arra vár, addig az alkalmazás befagyhat, esetleg összeomolhat. Az aszinkron műveletek lehetővé teszik, hogy ezeket az időigényes feladatokat a háttérben másik szálakon hajtsák végre, biztosítva azt, hogy a fő szál szabadon marad más kritikus feladatok futtatására, mint a renderelés, bemeneti információk feldolgozása és a játék logika kezelése. Az aszinkron műveletek gyakran jobb hibakezelést tesznek lehetővé, míg a szinkron feldolgozásnál egy hiba megzavarhatja az egész folyamatot, addig az aszinkron feldolgozásnál fellépő hibákat „el lehet kapni” és le lehet kezelni különböző erre kidolgozott megoldásokkal [31]. A képfeldolgozás és a Unity alkalmazás futtatása erőforrás igényes feladat, így ahhoz, hogy ez optimálisan működjön, célszerű a feladatokat több szálnak kiadni, hogy a terhelés lehetőleg ne egy szálra (fő szál) történjen.

6.4.3 MediaCapturer script

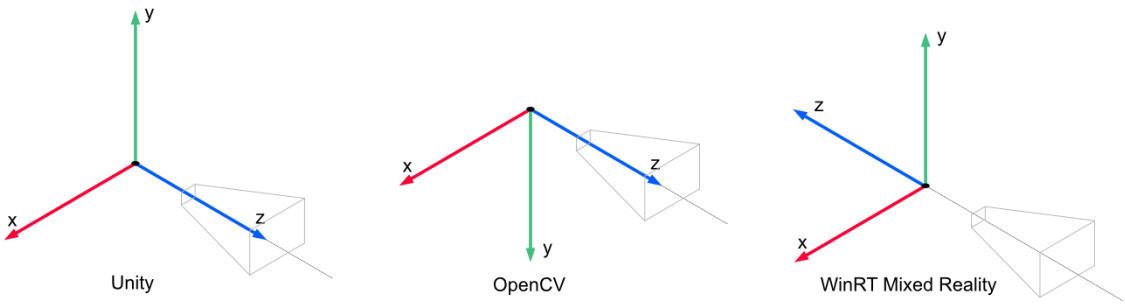
A MediaCapturer script felelős a HL2 PV kamerájának az inicializálásáért és az adatfolyam fenntartásáért. Egy publikus osztályt tartalmaz, mert több scriptben is objektumot kell majd létrehozni belőle. Mindkettő szcenárió vezérlő scriptje az ebben lévő osztályból létrehozott objektumon keresztül kapja meg a kamerától érkező képeket. Az osztály két aszinkron feladatot (Task) tartalmaz, melyek a kamera és az adatfolyam elindításáért és leállításáért felelősek. A HL2 kamerájának elérése Windows Runtime API-okon keresztül történik, és azokat a kódrészeket a Unity scriptekben, amik WinRT API-okat kezelnek, #if ENABLE_WINMD_SUPPORT #endif direktívák közé kell tenni, mert a Unity szerkesztőben ezek a kódrészek nem tudnak futni és csak hibát dobnának [32]. A kamera és adatfolyam elindításáért felelős feladat (StartCapture) bemeneti paraméterként megkapja a videó szélességét, magasságát és képkocka sebességét beállító változókat. A HoloLens-en a kamera több profilt támogat melyek mindegyike különböző üzemmódokat konfigurál. Innentől aszinkron API hívásokon keresztül, először megkeresi az összes elérhető profilt, majd kiválasztja a bemeneti paraméterekkel megegyezőt és azzal inicializálja a MediaCapture objektumot, ami a kamera adatfolyam rögzítésére

biztosít függvényeket. Ezután elindítja a videó adatfolyamot majd a MediaFrameReader objektum segítségével elkezdi olvasni azt. A MediaFrameReader objektum hozzáférést ad a képkockákhoz, és eseményt is képes hívni, ha egy új képkocka érkezett. A képkockákról egy a scriptben implementált másik függvény (GetLatestFrameRef()) segítségével lehet egyesével hivatkozást (MediaFrameReference) készíteni, amiből azután kinyerhető a kép (SoftwareBitmap). A SoftwareBitmap egy tömörítetlen bittérképet ábrázol a kamera képről [33].

6.4.4 ArUcoTracking script

Ez a script felelős az ArUco markerek követését bemutató szcenárió vezérléséért, és ha ez a szcenárió fut akkor a fő szál feladatai is az ebben implementáltak lesznek. Amikor elindul az alkalmazás és betöltődik az ArUcoTracking script, Windows Runtime API-okon keresztül készít egy referencia koordináta rendszert a HL2 aktuális szenzor adatainak alapján, amihez később az ArUco markerek lesznek transzformálva. Ha az alkalmazás betöltött, először szekvenciálisan elkezdi az indító függvény végrehajtását. (Start) Ebben a függvényben kerülnek betöltésre az inspectorban beállított változók értékei a megfelelő objektumokba, majd aszinkron módon megpróbálja elindítani a kamerát a MediaCapturer-ből létrehozott objektumon keresztül. Ha kamera sikeresen elindult, akkor egy másik szálon elindul egy ciklus, ami a képek feldolgozását végző HandleArUcoTracking() függvényt hajtja végre egészen addig amíg le nem állítjuk. A HandleArUcoTracking() függvényben először lekéri a legfrissebb képkockát a GetLatestFrameRef()-el, majd egy új változóba felveszi a bittérképet (SoftwareBitmap). Ha a bittérkép nem 0, tehát rendelkezésre áll akkor felveszi egy változóba a képkockából az aktuális koordináta rendszert, amiben a kép készült majd meghívja a DetectMarkers() függvényt, ami az ArUco markerek felismerését, pozíciójának meghatározását és a virtuális objektum (3D kocka) renderelését végzi. Innen a bittérkép (SoftwareBitmap) az OpenCVBridge Windows Runtime Komponenshez kerül, ahol az OpenCV és az ArUco modul feldolgozza, majd visszaadja a talált markereket tartalmazó listát. Ha a lista nem üres, akkor kezdődik a markerek egyesével való feldolgozása. A WinRT, az OpenCV és a Unity nemcsak különböző típusokat, de különböző koordináta rendszereket is használ,

emiatt konverziókat kell végrehajtani a kapott adatokon mielőtt azokat használni lehetne [34], [35], [36].



6.8 ábra Unity, OpenCV, WMR koordináta rendszerek

A WinRT System.Numerics névterű változókat ad vissza, ezeket először UnityEngine névterűre kell hozni, hogy azok Unity-ből is használhatóak lehessenek. A konverziókat végző függvények az ArUcoUtils osztályban vannak implementálva. A marker eltolási vektorja és forgatási vektorja két új UnityEngine.Vector3 típusú változóba kerül. Egy Unity szcenárióban minden objektumnak van egy transzformációja. Ez szolgál az objektum pozíciójának, forgásának és méretarányának a tárolására. A forgatásokra a Unity Kvaterniókat használ [37]. (Quaternion)

Az OpenCV egy forgatási vektort ad vissza, ami a legkompaktabb implementációja a forgatási mátrixnak. Ezt Rodrigues paraméterekként is nevezik, négy szám alkotja $[\theta, x, y, z]$, amelyek megadják, hogy a $v = [x, y, z]$ egység vektor által leírt tengely körül theta (θ) szöggel történik a forgatás [38]. Az OpenCV ennek a jelölésnek a kompakt változatát használja, ilyenkor a 3 elemű forgatási vektor (rvec) hossza lesz a theta (θ), a v forgástengely pedig a normalizált bementi vektor:

$$rvec = [a, b, c] \quad \theta = \sqrt{(a^2 + b^2 + c^2)} \quad v = \frac{rvec}{\theta} = \left[\frac{a}{\theta}, \frac{b}{\theta}, \frac{c}{\theta}\right] \tag{1}$$

Miután megvan a forgástengely és a forgatási szög, át lehet konvertálni kvaternióba, hogy Unity-ben is használható legyen [39]. Ez egy beépített függvény, a Quaternion.AngleAxis() segítségével könnyen megoldható, ami lényegében 2-es. és 3-as számú egyenletek alapján működik:

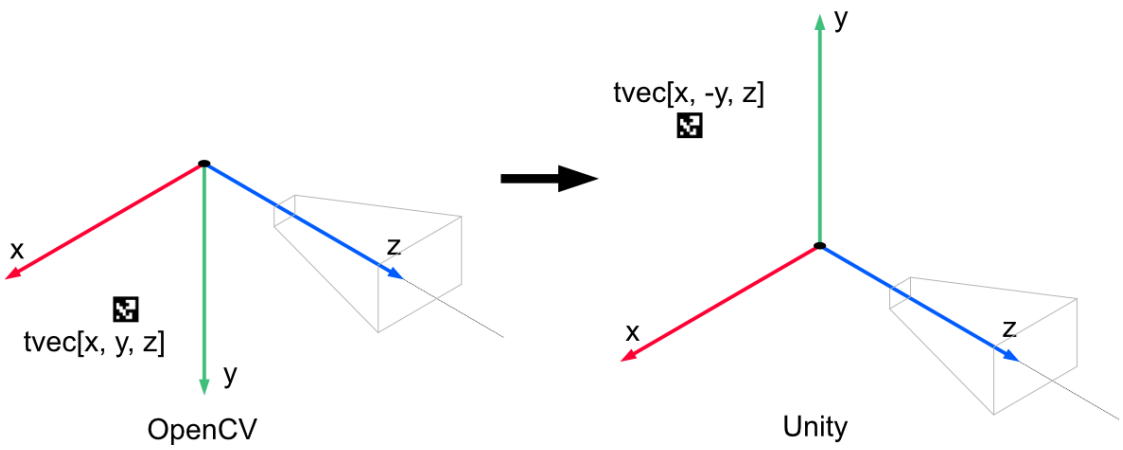
$$q_0 = \cos\left(\frac{\theta}{2}\right)\ q_1 = \hat{x}\sin\left(\frac{\theta}{2}\right)\ q_2 = \hat{y}\sin\left(\frac{\theta}{2}\right)\ q_3 = \hat{z}\sin\left(\frac{\theta}{2}\right) \quad (2)$$

$$q = (q_0, q_1, q_2, q_3) \quad (3)$$

Fontos, hogy itt minden egyenletnél fokban és nem radiánban vannak a szögek.

A Unity-ben a koordináták az úgynevezett „bal-kezes” koordináta rendszer szerint vannak reprezentálva, ezzel szemben az OpenCV a „jobb-kezes” koordináta rendszert használja. Emiatt a kiszámított kvaternió forgástengelyén az x és z koordinátát 180 fokkal el kell forgatni, hogy a Unity koordináta rendszerében is a megfelelő helyen legyenek.

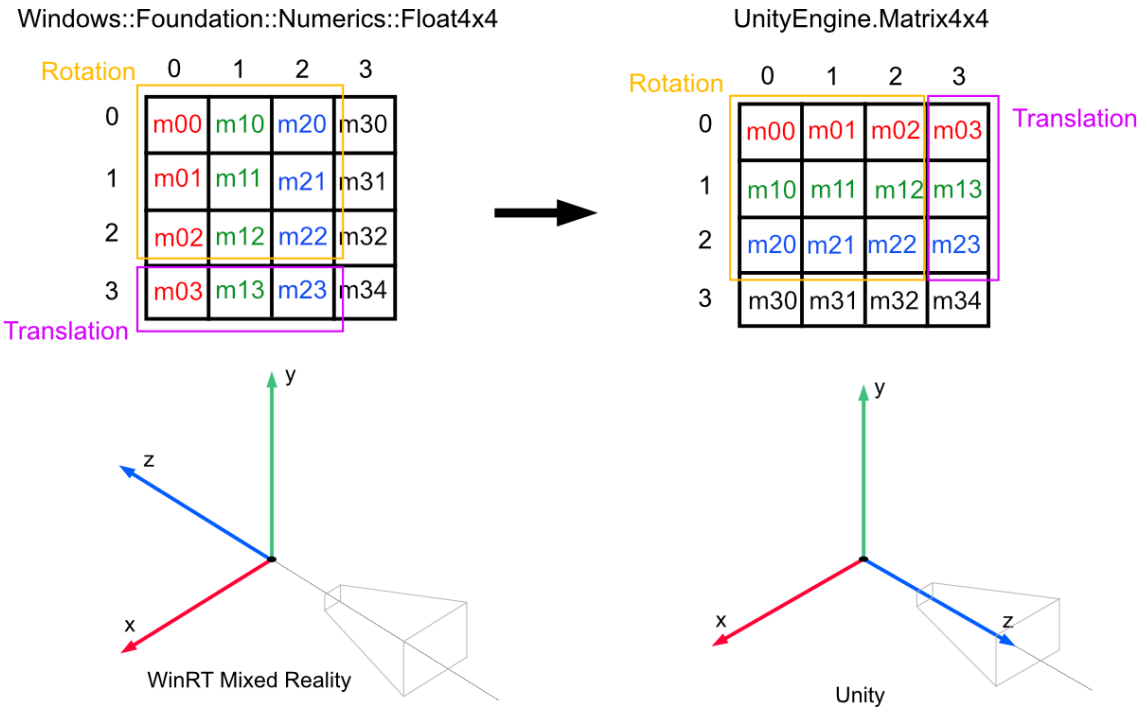
Az OpenCV által visszaadott eltolási vektorban szintén „jobb-kezes” koordináta rendszer szerint vannak a koordináták, de mivel itt nem forgatásról, hanem pozícióról van szó, az y tengelyt kell 180 fokkal elforgatni, hogy a koordináták a megfelelő helyre kerüljenek. (-1 el meg kell szorozni az y koordinátákat)



6.9 ábra OpenCV-Unity eltolás koordináta konverzió

Mostanra megvan az eltolási vektor és a forgatási kvaternió, és mindkettő UnityEngine típusú változókban. Következő lépésben szükség van az objektum koordináta rendszerében lévő pontok a Unity kamera koordináta rendszerébe való transzformálására. A függvény felépíti a eltolási vektor és a forgatási kvaternió transzformációját a Matrix4x4.TRS() meghívásával. Az itt kapott mátrix az objektumokat a bemeneti eltolási vektor pozíciójára, a bemeneti forgatás (kvaternió) orientációjába helyezi és a 3. bemeneti

paraméter szerint skálázza, ami egy egységvektor. (Vector3.one) Ezután szükség van a képkocka által visszaadott kamera koordináta rendszer és a Unity koordináta rendszer (indításkor felvett referencia koordináta rendszer) közötti transzformációra. Először egy Windows Runtime API (Mixed Reality) függvénnyel (TryGetTransformTo()) meghatározza a képkocka koordináta rendszer transzformációs mátrixát a Unity koordináta rendszerhez képest, majd ennek egy beépített identitás mátrixszal (Matrix4x4.Identity) való megszorzása után megkapja a WinRT-Unity transzformációs mátrixot. A WMR API is „jobb kezes" koordináta rendszert használ, viszont itt a z tengely van ellentétes irányban a Unity koordináta rendszerében lévővel. A WMR mátrixa (Float4x4) sorvektor elrendezés szerint működik a Unity (Matrix4x4) oszlopvektor elrendezésével szemben [40], [41], [42].



6.10 ábra WMR, Unity transzformációs mátrix elrendezése

Emiatt először transzponálni kell a kapott mátrixot, amitől az oszlopvektor formába kerül, majd a 3. sorban (az ábrán 2. indexű) lévő elemeket -1-el meg kell szorozni, hogy azok a Unity koordináta rendszere szerinti z tengelyen legyenek. Fontos, hogy a WinRT

mátrixnál a számozás 1-től, a Unity-nél pedig 0-tól indul, tehát például a WinRT m11 cellája Unity-ben m00-nak felel meg és az ábrán az utóbbi alapján vannak a jelölések.

Az előző lépések végrehajtása után megvan a marker transzformációs mátrixa és a kamera transzformációs mátrixa, ezeket összeszorozva a függvény kiszámolja a Unity világ transzformációt, amely tartalmazza a kamera állását a Unity, WinRT, és OpenCV viszonyában és a marker helyzetét. Ha ez megvan, akkor kibontásra kerülnek a létrehozott transzformációs mátrixból a marker Unity szcenárióban lévő koordinátái, amik megfelelnek a valós világban lévő marker koordinátáinak. Az eltolást egy 3D vektorban (Vector3), a forgatást pedig egy kvaternió (Quaternion) típusú változóban tárolja el. Ezek után következik a felhasználói felület frissítése.

A felhasználói felület frissítése a fő szálon történik, amit az InvodeOnAppThread() függvény hívásával hajt végre. Ebben frissíti a szövegdobozokon megjelenített aktuális állapotokat és ellenőrzi a listát a jelenleg megjelenített markerekről. Ha nincs benne a megjelenítendő marker azonosítója, akkor létrehoz egy új példányt a 3D kockából, ami megkapja a marker azonosítóját és utána beállítja a transzformációját a kiszámított értékekre (pozíció, orientáció). Ha benne van az azonosítója a listában, akkor frissíti a transzformációját. Ha az inspectorban engedélyezve volt a „Send Detected ArUco Data via TCP” opció akkor ellenőrzi, hogy a kapcsolat kiépült-e a TCPServer.py programmal, és elküldi a talált markerek adatait, amiket a másik számítógépen futó TCPServer.py program a konzolra kiír. (konverziók, transzformációk nélkül) A függvény a végén a SoftwareBitmap.Dispose() meghívásával eldobja bittérképet és befejeződik. A program ezt a függvényt ciklikusan ismétli addig, amíg le nem állítjuk.

6.4.5 CameraCalibration script

Ez a script felelős az kamera kalibrálásához szükséges képek elkészítését segítő szcenárió vezérléséért. Amikor a szcenárió fut akkor a fő szál feladatai is az ebben implementáltak lesznek. Az alkalmazás indulásával betöltődik a CameraCalibration script és az ArUcoTracking scripthez hasonlóan, először az inspectorban beállított változók értékeit betölti az objektumokba. Utána megpróbálja aszinkron módon elindítani a kamerát a MediaCapturer-ből létrehozott objektumon keresztül. A kamera kalibrációs szcenárióban

a felhasználói felület ugyanúgy tartalmazza a két szövegdobozt, amik az alkalmazás aktuális állapotáról nyújtanak tájékoztatást a felhasználó számára, de a felület kiegészül egy virtuális négyzetes síkkal, ami felhasználó elé van rögzítve. Ez fogja majd a kamera által közvetített adatfolyamot (képkockákat) megjeleníteni. Ha kamera sikeresen elindult, akkor egy másik szálon aszinkron módon elindít egy ciklust, ami a képek megjelenítését végző függvényt (HandlePreviewTextureUpdate()) hajtja végre egészen addig amíg le nem állítjuk. A HandlePreviewTextureUpdate() függvényben kibontásra kerül a MediaFrameReference és egy új változóba felveszi belőle a bittérképet (SoftwareBitmap). Ha a bittérkép nem 0, tehát rendelkezésre áll akkor először zárja a puffert a memóriában és megnyitja olvasó módban. Ezután szétválasztja a bittérképet és a bájt tömböt majd hivatkozást vesz fel a bájt tömbről. A kép tárolására egy bájt típusú tömb (byte[]) változót vesz fel aminek a mérete a kép szélességének, magasságának és a színcsatornák (4) számának a szorzata. Az új változóba a memória hivatkozás alapján egy .NET függvény (Marshal.Copy()) segítségével másolja át a képet. Ha a kép a tömbben van, akkor az ArUcoTracking scriptben használt megoldáshoz hasonlóan a felhasználói felület frissítése a fő szálon történik. A függvény frissíti a szövegdobozokon megjelenített aktuális állapotokat és a Unity LoadRawTextureData() függvényének hívásával betölti a képet a négyzetes felületre (PVMediaTexture) majd megjeleníti azt. A ciklus ezt addig ismétli amíg a programot meg nem állítjuk. Ebben a szcenárióban a felhasználó valós időben látja azt, amit a HL2 kamerája lát, és amikor a kamera kalibrációs folyamatot csinálja betudja pozícionálni a kalibrációs táblát, mielőtt képet készít.

A kép mentését végző feladat több módon is elindítható. Az alap konfiguráció szerint az alkalmazásnak szüksége van egy külső játékvezérlőre (Xbox One Controller) amely Bluetooth kapcsolaton keresztül kommunikál a HL2-vel. Ha a játékvezérlőn lenyomjuk a programban beállított gombot (A) akkor a bájt tömbből a képet megkapja a TCPClient script, ami a TCPServer cli programmal való kapcsolat bonyolításáért felel. Minden csomag, amit a TCPClient küld rendelkezik egy címkével (header) ami a csomag típusát jelöli és a csomag legelején helyezkedik el. A címke lehet „m” az ArUco marker adatok számára, ilyenkor szöveges adatot vár a cli program, vagy lehet „p”, ami a PV kamera képet jelöli.

Kamera kép küldése esetén a csomag elején a címke után következik a kép mérete, ezt követi a képhez csatolt időbélyegző, ami UNIX idő szerint adja meg a kép készítésének időpontját, majd utána jön a bájt tömb. A UNIX időbélyegző használata egy olyan módszer, ami másodpercek összegeként fejezi ki az eltelt időt egy bizonyos időponthoz (1970.01.01 0:00:00 UTC) képest, ez nem függ a különböző időzónáktól és megkönnyíti a számítógépes rendszerek számára az időpontok kezelését. A TCPServer cli program ezután kiírja a konzolra az időbélyegzőt, a kép típusát és az OpenCV segítségével létrehoz egy új fájlt az érkezett bájtokból, ami a képet tartalmazza. Ha elegendő képet készítettünk, akkor a Calibrate_PhotoVideo.py script használatával kiszámíthatóak a kamera paraméterek, amik azután felvihetőek az ArUco követés szcenárió inspectorában a megfelelő mezőkbe.

A kamera kalibrációs folyamat jelen esetben 3 részre bontható:

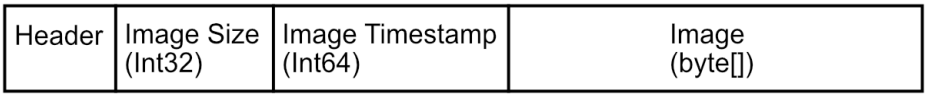
1. Képek készítése a kalibrációs tábláról a HL2 segítségével különböző nézőpontokból.
2. A képek feldolgozása OpenCV-vel egy másik számítógépen.
3. Kamera paraméterek bevitele az ArUcoTracking szcenárió ScriptConfiguration inspectorában a megfelelő helyekre.

6.5 Kiegészítő CLI programok és scriptek

A kiegészítő programok a HL2-től jövő adatok egy külön számítógépen való feldolgozását teszik lehetővé. Ezek segítségével lehet markereket generálni, a kamera paramétereket kiszámolni és a felismert markerek adatait valós időben kiíratni. A programok elkészítéséhez Python programnyelvet használtam, mert funkcióikat tekintve nem igénylik a magasabb szintű, komplexebb nyelvek által kínált megoldásokat és egyszerűségük miatt nincs szükség egy külön futtatható állomány létrehozására. A felhasználói felületük csak az információk kiírására alkalmas, a számítógép parancssorából futnak és az adatokat karakteresen jelenítik meg. Futtatásukhoz szükséges a Python 3 telepítése az opencv-python és opencv-python-contrib modulokkal együtt.

6.5.1 TCPServer.py

Ez a fő cli program, ehhez csatlakozik a HL2-őn futó alkalmazás és ennek továbbítja az adatokat. Egy egyszerű ciklusból áll, ami a program futtatásánál elindít egy WebSocket-et amit a kódban levő változók értékeként beállított IP cím és Port kombinációhoz rendel. Azért kell az IP címet külön beállítani, mert előfordulhat, hogy a HL2-re való fejlesztés során külön a fejlesztésre fenttartott hálózatot használunk, amin általában nincs internet elérés, csak a programok hibakeresését és a HL2-vel való kommunikációt szolgálja. Ilyenkor megadható a számítógép erre a hálózatra csatlakozó interfészének IP címe és a WebSocket csak azon keresztül lesz elérhető. Ha elindult a WebSocket, akkor egy végtelen ciklusban várja a bejövő kapcsolatokat amíg le nem állítjuk az ESC billentyűvel vagy ki nem épül egy kapcsolat. Ha kiépült a kapcsolat a HL2-őn futó alkalmazással, akkor az első ciklus véget ér és indul a második, amiben várja a bejövő csomagokat. Amikor beérkezik egy csomag, először szétbontja és dekódolja UTF-8 szerint az első 2 bájtot, ami a csomag címkéjét (header) tartalmazza. Ha megvan a címke akkor elágazások alapján eldönti, hogy a bejövő adat ArUco marker információ (header=m) vagy kamera kép (header=p). Ha kamera képről van szó, akkor kibontja a címke után érkezett bájt tömb méretét tartalmazó adatot (int32), ez 4 bájtot foglal el a csomagból, utána kibontja az időbélyeget, ami 8 bájt (int64) majd jön maga a bájt tömb, aminek a mérete a címke után érkezett és már ismert. A csomagban az adatok sorban vannak egymás után, ezért amikor a bájt tömbhöz ér akkor azt betölti egy mátrixba, aminek annyi sora van amekkora a kamera kép magassága, és annyi oszlopa amekkora a kamera kép szélessége. A kép szélessége és magassága a scriptben levő változókban van beállítva.



6.11 ábra TCP csomag a HL2-ről

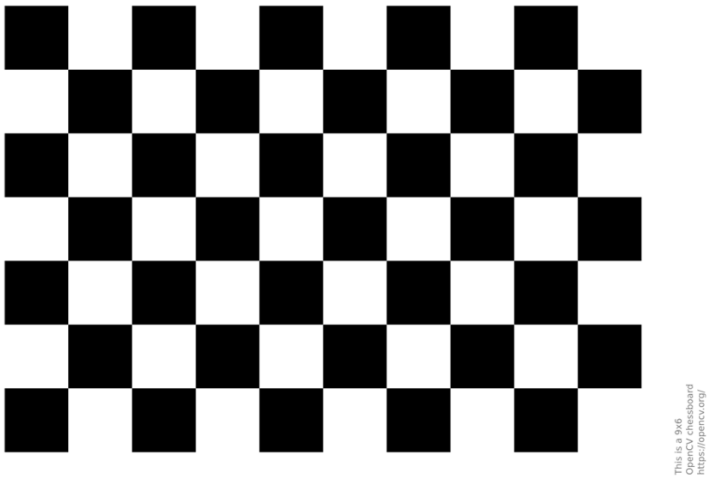
Mivel BGRA8-as formátumú képről van szó ezért a mátrix 4 csatornával rendelkezik. Ha megvan a mátrix akkor egy OpenCV függvénnyel (cv2.imwrite()) elmenti egy fájlba és a konzolon is kiírja az aktuális állapotot. Ha ArUco marker információkról van szó, akkor

azok szöveges formában érkeznek UTF-8 kódolásban, ilyenkor az egész csomagot ennek megfelelően dekódolja és kiírja a konzolra.

6.5.2 Calibrate_PhotoVideo.py

Ez a program felelős a kamera paraméterek kiszámításáért. A kamera kalibrálás célja, hogy megkapjuk a 3x3-as kamera mátrixot, a 3x3-as forgatási mátrixot és a 3x1-es eltolási vektort az ismert 3D pontok a világkoordináta rendszerben és a hozzájuk tartozó 2D kamera koordináta rendszerben lévő pontok segítségével (u, v).

Többféle kalibrálási folyamat létezik viszont az OpenCV-vel a legegyszerűbb a minta alapú kamera kalibrálás. Mivel a képalkotási folyamat felett teljes kontrollal rendelkezünk, így egy ismert méretű és dimenziójú mintáról különböző nézőpontokból több kép készítésével megoldható a folyamat.



6.12. ábra OpenCV kamera kalibrációs minta [https://github.com/opencv/opencv/blob/4.x/doc/pattern.png]

A program egyetlen függvényből áll, ami először a kalibrációs táblán lévő négyzetek oldalának a mérete és a sakktábla dimenzióinak alapján felveszi a minta pontjait. Ezt követően bemeneti adatként beolvassa a HL2-ről a TCPServer.py által lementett képeket, utána minden képen megkeresi a sakktábla mintát és meghatározza a sarkok koordinátáit (cv2.findChessboardCorners()), majd a már megtalált sarkok környezetét újra átnézi, de ez úttal többször és nagyobb pontossággal (cv2.cornerSubPix()). Ha ez megvan akkor

következik a kalibrálási folyamat utolsó része, melyben az algoritmus (cv2.calibrateCamera) megkapja a talált sarkokat, a 3D pontokat a világkoordináta rendszerben és azoknak a 2D kamera koordináta rendszerben lévő pontjait majd meghatározza kamera paramétereit [43]. Végül a program kiírja az eredményeket egy fájlba (PhotoVideo_intrinsics.yml), ahonnan azok felvihetőek az ArUco követés szcenárió ScriptConfiguration inspectorában lévő megfelelő mezőkbe.

6.5.3 GenerateArUcoMarkers.py

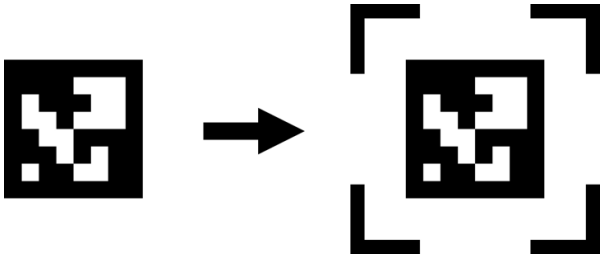
Ez a program a kinyomtatható ArUco markerek generálását segíti. Az OpenCV ArUco moduljának összes előre definiált szótáraiban elérhető markert létre tudja hozni. A program kimenete egy vagy több képfájl melyek a kinyomtatható ArUco markert tartalmazzák. A markerek mérete, mennyisége és az ArUco szótár, amiből a markereket generálja igény szerint beállítható a scriptben.

A program egy függvényből áll, ami egy véges ciklust futtat annyiszor, ahányat beállítottunk a generálandó markerek számának. A ciklusban az OpenCV ArUco moduljának generateImageMarker() függvénye fut le először ami létrehoz a script elején lévő változókba felvett szótárból és marker méretből egy markert, ami bekerül egy objektumba, és azt utána a cv2.imwrite() függvény hívásával kiírja a egy fájlba.

6.5.4 GenerateCanvas.py

Ez egy kiegészítő program a GenerateArUcoMarkers.py-hez. Amikor legeneráljuk az ArUco markereket, akkor a marker teljesen kitölti a képet és minden oldala pontosan annyi pixel lesz, mint amit a scriptben beállítottunk. Ez normál esetben amikor fehér papírra nyomtatunk (A4) nem jelent problémát, mert a felismerő algoritmus könnyedén képes lesz megtalálni a sarkokat. Amikor viszont a markert körbe vágják és nem hagynak fehér szegélyt körülötte, az negatív hatással van a felismerő algoritmus teljesítményére. Ez a program létrehoz egy plusz fehér területet a marker körül, ami megkönnyíti a körbevágását úgy, hogy a marker körüli fehér szegély is megmarad.

A program egy ciklusból áll, ami OpenCV-vel beolvassa (cv2.imread()) a markert tartalmazó képet és készít köré egy fehér szegélyt. Ha ez megvan akkor a fehér szegély 4 sarkára fekete téglalapokból sarkokat alakít ki, majd kiírja egy fájlba. A ciklus ezt addig ismétli, ameddig az összes bemeneti markeren végig nem ér.

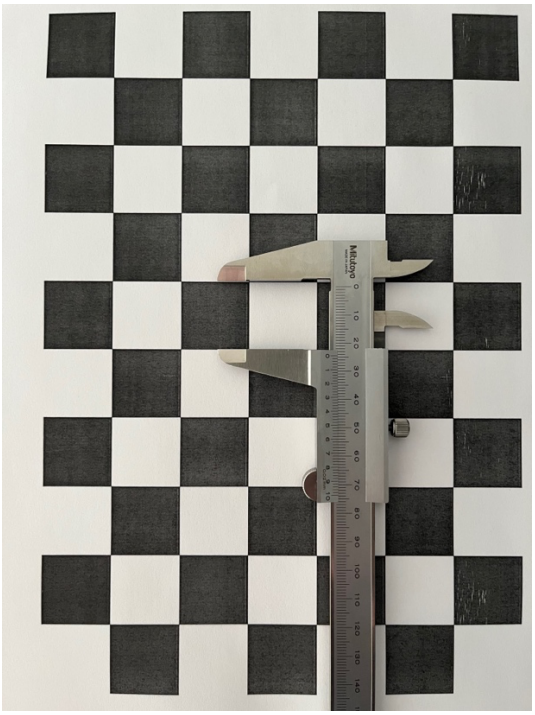


6.13 ábra 6x6-os ArUco marker szegéllyel kiegészítve

6.6 Tesztelés

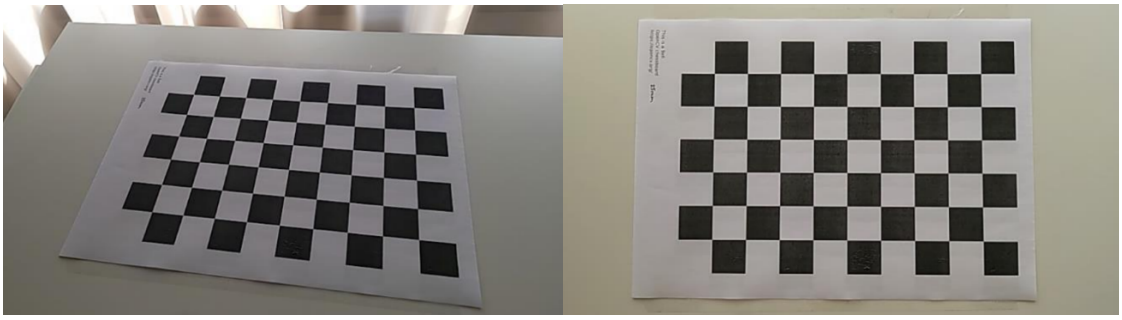
6.6.1 Kamera kalibráció

A szoftver csomag tesztelését a HL2 PV kamerájának kalibrálásával kezdtem. Először kinyomtattam az OpenCV kamera kalibrációs mintát egy A4-es fehér papírra. Ez egy sakktáblához hasonló effektíven 6 sorból és 9 oszlopból álló minta. A kinyomtatott papíron lemértem 3db különböző helyen lévő négyzet oldalának hosszát tolómérővel, majd ezek átlagát (24.6 mm) és a sakktábla dimenzióit (6, 9) beállítottam a Calibrate_PhotoVideo.py programban.



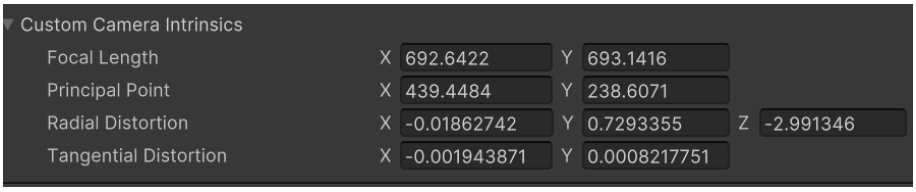
6.14 ábra A kinyomtatott OpenCV kamera kalibrációs minta mérése

A Unity projekt CameraCalibration szcenáriójával kipróbáltam több kamera beállítást, és a videó a 896x504-es felbontású profillal volt a legfolyamatosabb, így ezt használtam a kalibrálás és marker detektálás során. Beállítottam a TCPServer.py programot futtató számítógép IP címét és portját a Unity projekt inspectorában és lebuildeltem a programot. Az alkalmazást Visual Stúdióval telepítettem a HL2-re a 6.4.1-es részben leírtak alapján. Először a TCPServer.py programot indítottam el, majd utána a HL2-őn a Unity alkalmazást, ezt követően pedig különböző nézőpontokból képeket készítettem a headsettel amiket a TCPServer.py kép fájlokba kiírt a másik számítógépen.



6.15 ábra HL2 PV kamera képek 896x504-es videó profillal

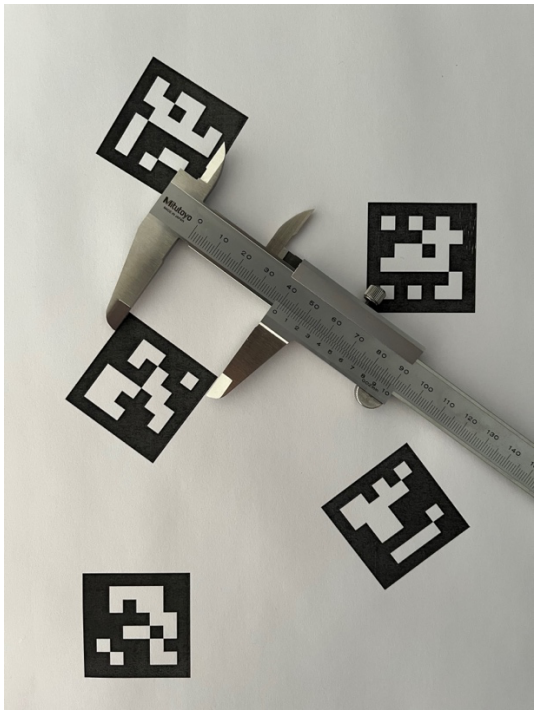
Körülbelül 40 kép után lefuttattam a Calibrate_PhotoVideo.py programot, ami kiszámolta a kamera paramétereket és azokat beállítottam a Unity projekt ArUcoTracking szcenárió inspectorában a megfelelő mezőkben.



6.16 ábra ArUcoTracking scene inspector kamera paraméterek beállítása

6.6.2 ArUco markerek generálása és felismerése a HL2-vel

Összesen 5db ArUco markert generáltam a GenerateArUcoMarkers.py programmal, ezekhez a „DICT_6X6_100”-as szótárt és 500 pixeles oldal méretet adtam meg. A markereket egy A4-es méretű képre különböző helyeken és irányokba forgatva helyeztem el egy képszerkesztő programmal (Affinity Designer), majd ezt kinyomtattam.

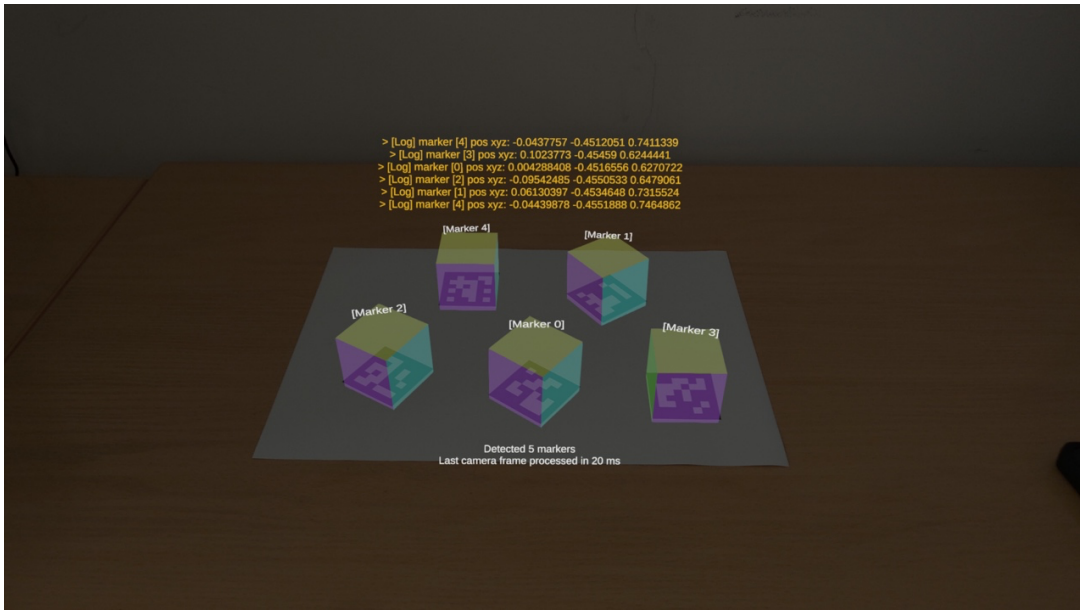


6.17 ábra A kinyomtatott ArUco marker oldal hosszának mérése

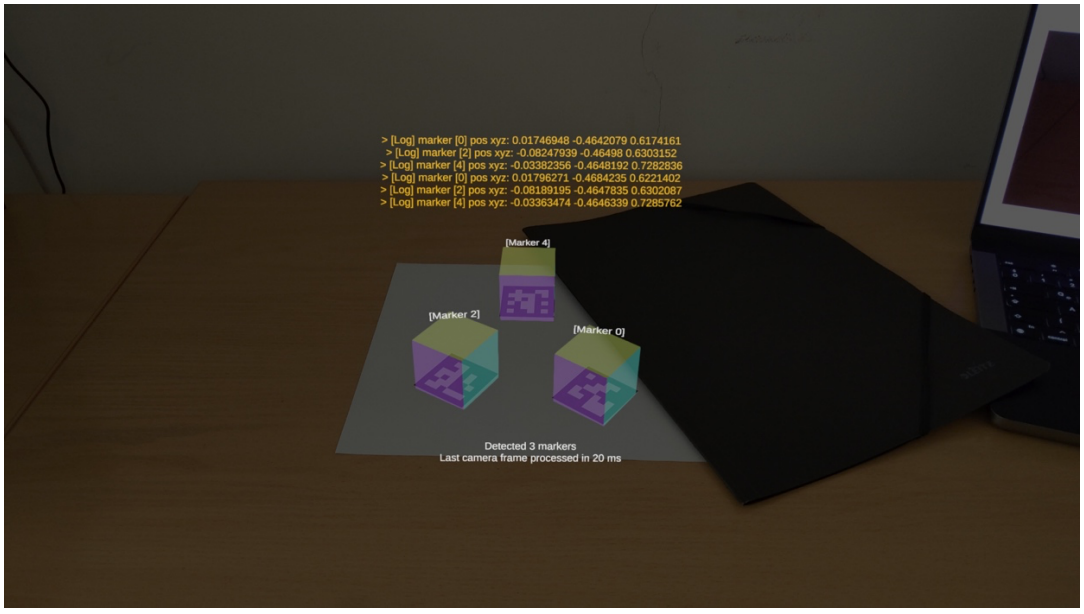
A nyomtatott markerek oldalát lemértem és a Unity projekt ArUcoTracking szcenárió inspectorában beállítottam az eredmények átlagát, a szótárt, valamint a videó profilt. Fontos, hogy az inspectorban a marker oldalának a hosszát méterben kell megadni, így a 43mm 0.043m ként lett beírva.

6.18 ábra ArUcoTracking scene inspector marker és videó profil beállítása

Ezt követően az ArUcoTracking szcenáriót kiválasztva buildeltem és Visual Stúdióval telepítettem a HL2-re a 6.4.1-es részben leírtak alapján az alkalmazást. A kinyomtatott markereket tartalmazó papírra körülbelül 40 cm-es távolságból néztem rá a headsettel, az alkalmazás észlelte és sikeresen meghatározta a papíron található összes marker helyzetét, majd megjelenítette rajtuk a virtuális kockákat.



6.19 ábra HoloLens2CVUnity program futás közben, az összes markert látja a kamera



6.20 ábra HoloLens2CVUnity program futás közben, a letakart markerekről lekerül a virtuális kocka

6.7 Tovább fejlesztési javaslatok

A tesztelések után megállapítható, hogy a szoftver sikeresen eleget tesz a feladatban megfogalmazott kritériumoknak, viszont közel sem tökéletes. Ebben a részben kifejtem az észrevételeimet és leírom az ötleteimet a program továbbfejlesztésével kapcsolatban,

valamint javaslatokat teszek a szoftverben alkalmazott megoldások módosítására, azok esetleges cseréjére.

6.7.1 Felhasználói felület

A HoloLens-en futó szoftver (Unity alkalmazás) felhasználói felülete minimális, csak a legalapvetőbb információkat jeleníti meg a felhasználó számára. A program működése lineáris, az indítást követően minden folyamat automatikusan megkezdődik, a felhasználónak nincs lehetősége ezek között váltani vagy leállítani az alkalmazásból való kilépésen kívül. Az alkalmazás mindig a Unity szerkesztőben kiválasztott szcenáriót fogja betölteni és nincs lehetőség váltani köztük a program újra buildelése nélkül. Be lehetne vezetni egy menürendszert, amivel a különböző szcenáriók között lehetne váltani, a folyamatokat tetszőlegesen elindítani, leállítani.

6.7.2 Referencia koordináta rendszer

A szoftver induláskor veszi fel a referencia koordináta rendszert, amihez később a markerek helyzetének megállapításához szükséges transzformációkat végzi. Ez a fizikai térből egy viszonylag kicsi, körülbelül $2m^2$ méretű területet képes megbízhatóan lefedni, ami azt jelenti, ha a felhasználó ebből kisétál akkor a hologramok elcsúsznak amíg vissza nem megy. A felhasználói felületen biztosítani lehetne erre egy külön kapcsolót, ami segítségével a referencia koordináta rendszer bármikor újból felvehető lenne, így az nem kötné a felhasználót egy bizonyos területre.

6.7.3 Kamera kalibráció

A kamera kalibráció a szoftver csomagban több bonyolult lépés elvégzésével teljesíthető, melyeken lehetne egyszerűsíteni. Mivel a HoloLens-en futó alkalmazás is rendelkezik OpenCV-vel ami jelenleg csak a képkocka feldolgozásért felel, a kamera kalibrációt automatizálni lehetne az eszközön és így nem lenne szükség külön számítógépre a képek feldolgozásához. Esetleg lehetne egy komplett interaktív kamera kalibrációs szcenáriót készíteni, ami valós időben mutathatná a mintát és az OpenCV által megtalált sarkokat.

Az OpenCVBridge projektben létre lehetne hozni a Calibrate_PhotoVideo.py programban használt függvényhez hasonlót, ami a HoloLens-en lehetővé tenné a kamera paraméterek meghatározását, majd azokat le is menthetné az eszköz tárhelyére és minden indításkor onnan vissza is tölthetné. Az ehhez szükséges menürendszer is létrehozható lenne a Unity projektben.

7. ÖSSZEFOGLALÓ

A szakdolgozat célja egy speciális mobil eszközre való szoftver csomag fejlesztése volt, amely segítségével a felhasználó képes lehet a valós világban különleges markerekkel megjelölt pontokra a virtuális világban létező objektumokat elhelyezni. A feladat megoldás során szükséges információk gyűjtésére az interneten elérhető dokumentációkat, publikációkat, fórumokat és blogokat használtam.

Először megvizsgáltam, hogy hogyan lehetne különféle szenzorok segítségével felmérni a valós világot és megjelölni benne a pontokat, amiket később a virtuális világban le lehet képezni. Ehhez meg kellett értenem a szenzorok működését, mérési értékeik kifejezésének rendszerét és a kalibrálásuk folyamatát. Megismerkedtem a virtuális, kiterjesztett és a kevert valóság fogalmával, valamint ezek alkalmazási területeivel. Az eszköz amire a szoftver csomag készült egy kevert valóság headset ami számos szenzorral rendelkezik melyek a projektben kulcsfontosságú szerepet töltenek be.

A hardver alapos tanulmányozása után kipróbáltam a gyártó által biztosított példa programokat és megismerkedtem az eszközre való szoftverfejlesztés menetével. Kiválasztottam a szükséges fejlesztői környezetet és megterveztem az alkalmazás architektúráját, majd létrehoztam a projekteket, amelyekből a szoftvercsomag felépül.

A fő alkalmazás két szcenárióból áll, egyik a markerek felismerését és a virtuális objektum elhelyezését valósítja meg, a másik pedig az eszköz kamerájának kalibrálásához nyújt segítséget. A markerek felismerését és helyzetének meghatározását egy külső szoftver könyvtárral valósítottam meg, aminek az elérése közvetlenül nem volt lehetséges a fő alkalmazásból így egy külön projektet kellett létrehoznom, ami biztosítja a kettő között a kommunikációt. Mivel a fő alkalmazás és a külső szoftver könyvtár, valamint az API, ami a szenzoroktól jövő adatokat biztosítja különböző típusokat és koordináta rendszereket használ, így meg kellett oldani a konverziókat ezek között, annak érdekében, hogy megfelelően együtt tudjanak működni.

A projekthez készítettem néhány kiegészítő programot, amik a speciális markerek generálását, az eszközről jövő adatok megjelenítését és a kamera paraméterek kiszámítását valósítják meg. A dolgozat egy átfogó dokumentációnak is tekinthető az alkalmazás felépítéséről és működéséről, tartalmazza a tesztelés menetét, valamint fejlesztési javaslatokat a szoftver jobbá tétele érdekében.

8. SUMMARY

The aim of the thesis was to develop a software package for a special mobile device which allows the user to place virtual objects on real world points marked with fiducial markers. To gather the information needed to solve the task, I used documentations, publications, forums and blog posts available on the Internet. First, I looked at how to use different sensors to measure the real world and mark points in it which could later be mapped in the virtual world. To do this, I needed to understand how the sensors work, what system they use to represent their measurements and what is the process of their calibration. I learned about the concepts and applications of virtual, augmented and mixed reality. The device which the software package was built for is a mixed reality headset with a number of sensors that play a key role in the project.

After a studying the hardware, I tried the example programs provided by the manufacturer and learned about the process of developing software for the device. I have chosen the development environment and designed the application architecture, then created the projects from which the software package consists of.

The main application is divided to two scenes, one to detect markers and place virtual objects on top of them, and one to help with calibration of the camera on the device. The marker detection and positioning was implemented using an external software library which could not be accessed directly from the main application and because of that I had to create a separate project to provide communication between these. Since the main application and the external software library and the API that provides the data from the sensors are useing different types and coordinate systems, I had to implement conversions between these in order to make them work together properly.

The software package also contains some additional programs that are meant to help the user generate special markers, display data from the device and calculate the camera parameters. The thesis can also be seen as a comprehensive documentation about the architecture and the functionality of the application, it also includes the testing process and some suggestions on how to improve the software.

9. IRODALOMJEGYZÉK

[1] “Camera Lenses Explained | Canon Australia.” Accessed: May 13, 2024. [Online]. Available: https://www.canon.com.au/get-inspired/camera-lenses-explained-jenn-cooper

[2] “Camera sensors explained - Canon Europe.” Accessed: May 11, 2024. [Online]. Available: https://www.canon-europe.com/pro/infobank/image-sensors-explained/

[3] “Camera Sensors: What Are They and How Do They Work? | FUJIFILM Exposure Center – USA.” Accessed: May 11, 2024. [Online]. Available: https://fujifilm-x.com/en-us/series/fundamentals-of-photography/camera-sensors-what-are-they-and-how-do-they-work/

[4] “Dissecting the Camera Matrix, Part 3: The Intrinsic Matrix ←.” Accessed: May 12, 2024. [Online]. Available: https://ksimek.github.io/2013/08/13/intrinsic/

[5] “Dissecting the Camera Matrix, Part 2: The Extrinsic Matrix ←.” Accessed: May 12, 2024. [Online]. Available: https://ksimek.github.io/2012/08/22/extrinsic/

[6] “The Important Difference Between Virtual Reality And Mixed Reality | Bernard Marr.” Accessed: May 11, 2024. [Online]. Available: https://bernardmarr.com/the-important-difference-between-virtual-reality-and-mixed-reality/

[7] “The Difference Between Virtual Reality, Augmented Reality And Mixed Reality.” Accessed: May 11, 2024. [Online]. Available: https://www.forbes.com/sites/quora/2018/02/02/the-difference-between-virtual-reality-augmented-reality-and-mixed-reality/?sh=489da7752d07

[8] S. Garrido-Jurado, R. Muñoz-Salinas, F. J. Madrid-Cuevas, and M. J. Marín-Jiménez, “Automatic generation and detection of highly reliable fiducial markers under occlusion,” *Pattern Recognit*, vol. 47, no. 6, pp. 2280–2292, Jun. 2014, doi: 10.1016/j.patcog.2014.01.005.

[9] “What is OpenCV? The Complete Guide (2024) - viso.ai.” Accessed: May 12, 2024. [Online]. Available: https://viso.ai/computer-vision/opencv/

[10] “HoloLens 2—Overview, Features, and Specs | Microsoft HoloLens.” Accessed: May 13, 2024. [Online]. Available: https://www.microsoft.com/en-us/hololens/hardware#document-experiences

[11] “Teardown Shows Microvision Inside Hololens 2 – KGOnTech.” Accessed: May 11, 2024. [Online]. Available: https://kguttag.com/2020/05/18/teardown-shows-microvision-inside-hololens-2/

[12] “HoloLens 2 hardware | Microsoft Learn.” Accessed: May 11, 2024. [Online]. Available: https://learn.microsoft.com/en-us/hololens/hololens2-hardware

[13] D. Ungureanu *et al.*, “HoloLens 2 Research Mode as a Tool for Computer Vision Research,” Aug. 2020, [Online]. Available: http://arxiv.org/abs/2008.11239

[14] “Introduction to CMake - DEV Community.” Accessed: May 12, 2024. [Online]. Available: https://dev.to/iamdeb25/introduction-to-cmake-26nk

[15] “What Is Unity? - A Top Game Engine For Video Games - GameDev Academy.” Accessed: May 13, 2024. [Online]. Available: https://gamedevacademy.org/what-is-unity/

[16] “winapi - What is a Windows API? - Stack Overflow.” Accessed: May 12, 2024. [Online]. Available: https://stackoverflow.com/questions/993470/what-is-a-windows-api

[17] “Windows Runtime C++ Template Library (WRL) | Microsoft Learn.” Accessed: May 12, 2024. [Online]. Available: https://learn.microsoft.com/en-us/cpp/cppcx/wrl/windows-runtime-cpp-template-library-wrl?view=msvc-170

[18] “Windows Runtime components with C++/WinRT - UWP applications | Microsoft Learn.” Accessed: May 12, 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows/uwp/winrt-components/create-a-windows-runtime-component-in-cppwinrt

[19] “Windows Runtime components - UWP applications | Microsoft Learn.” Accessed: May 12, 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows/uwp/winrt-components/

[20] “Dynamic link library (DLL) - Windows Client | Microsoft Learn.” Accessed: May 12, 2024. [Online]. Available: https://learn.microsoft.com/en-us/troubleshoot/windows-client/setup-upgrade-and-drivers/dynamic-link-library

[21] “Introduction to Microsoft Interface Definition Language 3.0 - Windows UWP applications | Microsoft Learn.” Accessed: May 12, 2024. [Online]. Available: https://learn.microsoft.com/en-us/uwp/midl-3/intro

[22] “The Windows Runtime (WinRT) type system - Windows UWP applications | Microsoft Learn.” Accessed: May 12, 2024. [Online]. Available: https://learn.microsoft.com/en-us/uwp/winrt-cref/winrt-type-system

[23] “Collections with C++/WinRT - UWP applications | Microsoft Learn.” Accessed: May 12, 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows/uwp/cpp-and-winrt-apis/collections

[24] “Color pixel formats.” Accessed: May 11, 2024. [Online]. Available: https://www.1stvision.com/cameras/IDS/IDS-manuals/en/basics-color-pixel-formats.html

[25] “Difference of OpenCV Mat types - Stack Overflow.” Accessed: May 12, 2024. [Online]. Available: https://stackoverflow.com/questions/19248926/difference-of-opencv-mat-types

[26] “Otsu’s Thresholding Technique | LearnOpenCV.” Accessed: May 12, 2024. [Online]. Available: https://learnopencv.com/otsu-thresholding-with-opencv/

[27] “OpenCV: Detection of ArUco Markers.” Accessed: May 12, 2024. [Online]. Available: https://docs.opencv.org/4.x/d5/dae/tutorial_aruco_detection.html

[28] “OpenCV: Perspective-n-Point (PnP) pose computation.” Accessed: May 11, 2024. [Online]. Available: https://docs.opencv.org/4.x/d5/d1f/calib3d_solvePnP.html

[29] “GitHub - opencv/opencv: Open Source Computer Vision Library.” Accessed: May 12, 2024. [Online]. Available: https://github.com/opencv/opencv

[30] “GitHub - opencv/opencv_contrib: Repository for OpenCV’s extra modules.” Accessed: May 12, 2024. [Online]. Available: https://github.com/opencv/opencv_contrib

[31] “Async/Await in C# Unity explained in easy words | by santosh parihar | Medium.” Accessed: May 12, 2024. [Online]. Available: https://medium.com/@sonusprocks/async-await-in-c-unity-explained-in-easy-words-571ebb6a9369

[32] “Unity - Manual: Windows Runtime support.” Accessed: May 12, 2024. [Online]. Available: https://docs.unity3d.com/Manual/IL2CPP-WindowsRuntimeSupport.html

[33] “SoftwareBitmap Class (Windows.Graphics.Imaging) - Windows UWP applications | Microsoft Learn.” Accessed: May 12, 2024. [Online]. Available: https://learn.microsoft.com/en-us/uwp/api/windows.graphics.imaging.softwarebitmap?view=winrt-22621

[34] “Coordinate systems in DirectX - Mixed Reality | Microsoft Learn.” Accessed: May 12, 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows/mixed-reality/develop/native/coordinate-systems-in-directx

[35] “Converting OpenCV cameras to OpenGL cameras. · Amy Tabb.” Accessed: May 12, 2024. [Online]. Available: https://amytabb.com/tips/tutorials/2019/06/28/OpenCV-to-OpenGL-tutorial-essentials/

[36] “Unity - Manual: Rotation and orientation in Unity.” Accessed: May 12, 2024. [Online]. Available: https://docs.unity3d.com/Manual/QuaternionAndEulerRotationsInUnity.html

[37] “Unity - Scripting API: Transform.” Accessed: May 12, 2024. [Online]. Available: https://docs.unity3d.com/ScriptReference/Transform.html

[38] “c++ - Rodrigues into Eulerangles and vice versa - Stack Overflow.” Accessed: May 12, 2024. [Online]. Available:

https://stackoverflow.com/questions/12933284/rodrigues-into-eulerangles-and-vice-versa

[39] “Quaternions.” Accessed: May 11, 2024. [Online]. Available: https://danceswithcode.net/engineeringnotes/quaternions/quaternions.html

[40] “c++ - HLSL mul() variables clarification - Stack Overflow.” Accessed: May 12, 2024. [Online]. Available: https://stackoverflow.com/questions/16578765/hlsl-mul-variables-clarification

[41] “float4x4 structure (Windowsnumerics.h) - Win32 apps | Microsoft Learn.” Accessed: May 12, 2024. [Online]. Available: https://learn.microsoft.com/en-us/windows/win32/numerics_h/float4x4-structure

[42] “Unity - Scripting API: Matrix4x4.” Accessed: May 12, 2024. [Online]. Available: https://docs.unity3d.com/ScriptReference/Matrix4x4.html

[43] Y. Zhao, Y. Wang, and Q. Cui, “Calculating intrinsic and extrinsic camera parameters based on the pnp problem,” *Journal of Engineering and Technological Sciences*, vol. 46, no. 3, pp. 258–270, 2014, doi: 10.5614/j.eng.technol.sci.2014.46.3.2.

10. MELLÉKLET

10.1 Vektorok

A vektor egy hosszúságot és egy irányt ír le. Általában egy nyíllal ábrázolják. Két vektor akkor egyenlő, ha azonos hosszúságúak és irányúak, valamint nem számít, ha különböző helyeken vannak. A vektorokra nyílként célszerű gondolni, nem koordinátákként vagy számokként. A programokban előfordul, hogy számokként kell ábrázolni őket, de a kódban objektumokként vannak kezelve, és azok közül csak az alacsony szintű vektor műveleteket végzőknek kell tudniuk a vektorok numerikus reprezentációikról. Az egységvektor minden olyan vektort magába foglal, amelynek a hossza egy. A nullvektor az a vektor, aminek a hossza nulla, és az iránya meghatározhatatlan. A vektorokat sokféle dolog ábrázolására lehet használni, ezek közül a legelterjedtebb az eltolás, és a helyzet tárolása. A helyzetet ábrázolhatjuk egy másik helytől való elmozdulásként. Általában van valamilyen pont (origó), amelytől az összes többi hely eltolásként értelmezhető. Egy hely nem egy eltolás, hanem egy eltolás egy adott origótól. Az eltolás pedig önmagában nem egy hely.

10.2 Mátrixok

A mátrix egy tömb, amelynek lehet több sora és oszlopa, valamint megfelel bizonyos aritmetikai szabályoknak. A számítógépes grafikában gyakran használt adatstruktúra mivel kiválóan lehet vele reprezentálni képeket, vektorokat (2D, 3D) és ezeken előre definiált függvények segítségével műveleteket végezni. A továbbiakban a dolgozathoz használt mátrixokat és műveleteket nézzük meg.

10.2.1 Mátrix aritmetika

Mátrix konstans számmal való szorzata alatt azt a műveletet értjük, amikor az adott mátrix minden egyes elemét megszorozzuk a konstanssal. A mátrixhoz hozzátudunk adni egy másik mátrixot abban az esetben, ha a két mátrix azonos rendű (ugyanannyi sora és oszlopa van), ilyenkor a megfelelő indexű elemeket kell összeadni és azok alkotják az új mátrixot. Mátrixok szorzása esetén szimplán összeszorozzuk az első mátrix sorait a

második mátrix oszlopaival, viszont ezt csak abban az esetben tehetjük meg ha bal oldali mátrix oszlopainak száma megegyezik a jobb oldalon lévő mátrix sorainak a számával.

10.2.2 Műveletek mátrixokon

Az A^{-1} inverz mátrixa A mátrixnak mert teljesíti, hogy AA-1=I, tehát:

$$\begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}^{-1} = \begin{bmatrix}-2.0 & 1.0\\1.5 & -0.5\end{bmatrix} \quad \begin{bmatrix}1 & 2\\3 & 4\end{bmatrix}\begin{bmatrix}-2.0 & 1.0\\1.5 & -0.5\end{bmatrix} = \begin{bmatrix}1 & 0\\0 & 1\end{bmatrix}$$

Ha pedig vissza akarjuk kapni az eredeti mátrixot, akkor az inverz mátrix inverze lesz az.

Egy mátrix transzponáltja az a mátrix, amelyben ugyanazok a számok vannak viszont a sorok kivannak cserélve az oszlopokkal:

$$\begin{bmatrix}1 & 2\\3 & 4\\5 & 6\end{bmatrix}^{\mathrm{T}} = \begin{bmatrix}1 & 3 & 5\\2 & 4 & 6\end{bmatrix}$$

10.2.3 Vektor műveletek mátrix formában

A számítógépes grafikában gyakran használnak négyzetes mátrixokat vektorok átalakítására. Például.: ha van egy 2D vektorunk és el akarjuk forgat 90 fokkal az origó körül, ennek megoldására használhatjuk egy 2x2-es és egy 2x1-es mátrix szorzatát, amit oszlop vektornak is neveznek. Általában ez egyszerűbben megoldható transzponálással is. Használhatunk mátrixokat vektorokon végzett műveletek kódolására is. Sokszor hasznosabb a mátrix szorzásokra vektor műveletekként tekinteni, így egy 3x3-as mátrixot leírhatunk egy tömbbel, ami vagy 3db oszlopvektort tartalmaz egymás mellett, vagy 3db sorvektort egymás fölött. (a vektorok 3D-sek ebben az esetben)

10.2.4 Identitás mátrix

Ez egy különleges típusa a diagonális mátrixoknak, ahol az átlóban olyan elemek fordulnak elő melyek értéke nem 0. Az átló olyan elemekből áll melyek oszlop indexe

egyenlő a sor indexével a mátrix bal felső sarkától számolva. Az identitás mátrix azzal a tulajdonsággal is rendelkezik, hogy ugyanaz, mint a transzponáltja és ebből következik, hogy szimmetrikus. Nevezhető ortogonális mátrixnak is mivel minden oszlopának hosszára vektorként tekintve 1 egységet kapunk és az oszlopok ortogonálisok egymásra, valamint ugyanez igaz a soraira is. Minden ortogonális mátrix determinánsa vagy +1 vagy -1.

10.3 OpenCV CMake beállítások és NuGet .targets fájl

```
Commandline options:
-DBUILD_opencv_rapid:BOOL="0" -DBUILD_JAVA:BOOL="0" -DWITH_OBSENSOR:BOOL="0" -DBUILD_opencv_fuzzy:BOOL="0"
 -Dthunder:BOOL="0" -DBUILD_opencv_world:BOOL="1" -DWITH_WIN32UI:BOOL="0" -DBUILD_opencv_ml:BOOL="1"
 -DOPENCV_DNN_TFLITE:BOOL="0" -DBUILD_opencv_intensity_transform:BOOL="0" -DBUILD_ZLIB:BOOL="0"
 -DBUILD_opencv_superres:BOOL="0" -DWITH_MSMF:BOOL="0" -DOPENCL_FOUND:BOOL="0" -DBUILD_opencv_gapi:BOOL="0"
 -DWITH_IMGCODEC_SUNRASTER:BOOL="0" -DWITH_IMGCODEC_PFM:BOOL="0" -DBUILD_opencv_stitching:BOOL="1"
 -DBUILD_opencv_java_bindings_generator:BOOL="0" -DBUILD_opencv_ximgproc:BOOL="0" -DBUILD_opencv_phase_unwrapping:BOOL="0"
 -DWITH_FLATBUFFERS:BOOL="0" -DBUILD_opencv_objc_bindings_generator:BOOL="0" -DBUILD_opencv_bgsegm:BOOL="0"
 -DBUILD_opencv_shape:BOOL="0" -DBUILD_opencv_js_bindings_generator:BOOL="0" -DBUILD_opencv_dpm:BOOL="0"
 -DBUILD_opencv_plot:BOOL="0" -DWITH_PNG:BOOL="0" -DBUILD_OPENJPEG:BOOL="0" -DBUILD_opencv_apps:BOOL="0"
 -DWITH_IMGCODEC_HDR:BOOL="0" -DWITH_DSHOW:BOOL="0" -DWITH_DIRECTX:BOOL="0" -DBUILD_opencv_ts:BOOL="0"
 -DINSTALL_PDB:BOOL="0" -DBUILD_opencv_reg:BOOL="0" -DBUILD_opencv_bioinspired:BOOL="0" -DWITH_ADE:BOOL="0"
 -DWITH_GSTREAMER:BOOL="0" -DWITH_OPENEXR:BOOL="0" -DWITH_OPENCLAMDFFT:BOOL="0" -DWITH_WEBP:BOOL="0"
 -DCV_TRACE:BOOL="0" -DBUILD_opencv_xphoto:BOOL="0" -DBUILD_opencv_face:BOOL="0" -DBUILD_opencv_surface_matching:BOOL="0"
 -DBUILD_SHARED_LIBS:BOOL="1" -DANT_EXECUTABLE:FILEPATH="ANT_EXECUTABLE-NOTFOUND" -DWITH_TIFF:BOOL="0"
 -DOPENCV_EXTRA_MODULES_PATH:PATH="C:/opencv/opencv_contrib-4.8.x/modules" -DBUILD_WEBP:BOOL="0"
 -DBUILD_opencv_videostab:BOOL="0" -DWITH_1394:BOOL="0" -DWITH_JPEG:BOOL="0" -DBUILD_opencv_mcc:BOOL="0"
 -DBUILD_TIFF:BOOL="0" -DBUILD_JPEG:BOOL="0" -DWITH_JASPER:BOOL="0" -DBUILD_ITT:BOOL="0" -DBUILD_IPP_IW:BOOL="0"
 -DBUILD_opencv_hfs:BOOL="0" -DCMAKE_BUILD_TYPE:STRING="Release" -DBUILD_opencv_xfeatures2d:BOOL="0"
 -DWITH_TESSERACT:BOOL="0" -DBUILD_PACKAGE:BOOL="0" -DWITH_EIGEN:BOOL="0" -DBUILD_opencv_python_bindings_generator:BOOL="0"
 -DWITH_FFMPEG:BOOL="0" -DBUILD_opencv_xobjdetect:BOOL="0" -DWITH_VTK:BOOL="0" -DWITH_OPENCLAMDBLAS:BOOL="0"
 -DINSTALL_PDB_COMPONENT_EXCLUDE_FROM_ALL:BOOL="0" -DWITH_QUIRC:BOOL="0" -DBUILD_opencv_structured_light:BOOL="0"
 -DBUILD_WITH_STATIC_CRT:BOOL="0" -DWITH_PROTOBUF:BOOL="0" -DBUILD_opencv_datasets:BOOL="0"
 -DOPENCV_TEST_DNN_TFLITE:BOOL="0" -DBUILD_opencv_wechat_qrcode:BOOL="0" -DWITH_IMGCODEC_PXM:BOOL="0"
 -DBUILD_opencv_rgbd:BOOL="0" -DBUILD_opencv_text:BOOL="0" -DWITH_OPENCL:BOOL="0" -DBUILD_TESTS:BOOL="0"
 -DOPENCV_DNN_OPENCL:BOOL="0" -DWITH_LAPACK:BOOL="0" -DCV_ENABLE_INTRINSICS:BOOL="0" -DWITH_IPP:BOOL="0"
 -DWITH_ITT:BOOL="0" -DBUILD_JASPER:BOOL="0" -DBUILD_opencv_python_tests:BOOL="0" -DBUILD_opencv_saliency:BOOL="0"
 -DWITH_MSMF_DXVA:BOOL="0" -DWITH_OPENCL_D3D11_NV:BOOL="0" -DBUILD_PERF_TESTS:BOOL="0" -DWITH_OPENJPEG:BOOL="0"
 -DBUILD_opencv_highgui:BOOL="0" -DBUILD_PROTOBUF:BOOL="0" -DBUILD_opencv_dnn_superres:BOOL="0"
 -DBUILD_opencv_quality:BOOL="0" -DBUILD_opencv_optflow:BOOL="0" -DBUILD_PNG:BOOL="0"
```

```
<?xml version="1.0" encoding="utf-8"?>
<Project ToolsVersion="4.0" xmlns="http://schemas.microsoft.com/developer/msbuild/2003">
<PropertyGroup>
        <OpenCVforHoloLens2-Platform Condition="'$(Platform)' == 'ARM64'">arm64</OpenCVforHoloLens2-Platform>
    </PropertyGroup>

    <ItemGroup Condition="'$(TargetPlatformIdentifier)' == 'UAP'">
        <ReferenceCopyLocalPaths Condition="'$(Configuration)' == 'Release'" Include="$(MSBuildThisFileDirectory)..\..\runtimes\win10-$(OpenCVforHoloLens2-Platform)\native\opencv_world480.dll" />
        <ReferenceCopyLocalPaths Condition="'$(Configuration)' == 'Debug'" Include="$(MSBuildThisFileDirectory)..\..\runtimes\win10-$(OpenCVforHoloLens2-Platform)\native\opencv_world480d.dll" />
    </ItemGroup>

    <ItemDefinitionGroup Condition="'$(TargetPlatformIdentifier)' == 'UAP'">
        <ClCompile>
            <AdditionalIncludeDirectories>$(MSBuildThisFileDirectory)..\..\include;%(AdditionalIncludeDirectories)</AdditionalIncludeDirectories>
        </ClCompile>
        <Link>
            <AdditionalLibraryDirectories>$(MSBuildThisFileDirectory)win10-$(OpenCVforHoloLens2-Platform);%(AdditionalLibraryDirectories)</AdditionalLibraryDirectories>
            <AdditionalDependencies Condition="'$(Configuration)' == 'Release'">opencv_world480.lib; %(AdditionalDependencies)</AdditionalDependencies>
            <AdditionalDependencies Condition="'$(Configuration)' == 'Debug'">opencv_world480d.lib; %(AdditionalDependencies)</AdditionalDependencies>
        </Link>
    </ItemDefinitionGroup>
</Project>
```

10.4 Projektek, forráskód

A teljes szoftver csomagot tartalmazó GitHub Repo:

https://github.com/nooway077/HoloLens2CVExperiments