



MISKOLCI EGYETEM
GÉPÉSZMÉRNÖKI ÉS INFORMATIKAI KAR



TUDOMÁNYOS DIÁKKÖRI DOLGOZAT

Dolgozat címe:

AZ ESZTÉTIKUS MARKER FELISMERÉSE DIGITÁLIS VIDEO FOLYAMBÓL

Készítette:

BARTA JÁNOS

Mérnök informatikus hallgató

KIS KRISZTIÁN

Mérnök informatikus hallgató

2013/2014 TANÉV, 2. FÉLÉV

Konzulens:

DR. VINCZE DÁVID

Egyetemi tanársegéd

Miskolci Egyetem Általános Informatikai Intézeti Tanszék



Tartalomjegyzék

I. Bevezetés	2
II. Az esztétikus marker	3
2.1. Az esztétikus marker bemutatása	3
2.2. A mobil robot	4
III. Az esztétikus marker szoftveres felismerése	5
3.1. Az esztétikus marker színekre bontása	5
3.2. A színszűrés megvalósítása	6
3.3. Kontúrkeresés a kiszűrt képeken	11
3.4. Az esztétikus marker behatárolása	22
3.5. Egyenes illesztés a marker oldalaira	29
3.6. A marker sarokpontjainak megtalálása	36
IV. A felismert esztétikus marker értelmezése	38
4.1. A hasznos rész kivágása, illetve nézőpont korrekciója.....	38
4.2. A kivágott marker forgatása.....	41
4.3. A marker hálójának berajzolása	43
4.4. A háromszögeket alkotó pixelek vizsgálata.....	46
V. Összefoglalás.....	47
Summary	48
Irodalomjegyzék	49

I. BEVEZETÉS

Különböző feltételezések szerint a mobil robotok a közeljövőben már nem csak ipari környezetben, hanem otthonainkban is meg fognak jelenni. Ennek egyik feltétele, hogy a különböző robotok rendelkezzenek az önálló helymeghatározás és navigáció képességeivel.

A helymeghatározás, illetve pályatervezés egy lehetséges módja, hogy a robotok különböző, a térben elhelyezett jelrendszerekre támaszkodnak. Ezek a jelzések (markerek) olyan információkat tartalmaznak, amik megfelelő viszonyítási pontot jelentenek, amelyekből a saját pozíciójuk adott pontossággal meghatározható.

A dolgozatban egy mesterséges esztétikus marker felismerésére kidolgozott algoritmus kerül bemutatásra. Az ipari markerekkel szemben az esztétikus markerek előnye, hogy esztétikai szempontból harmonizálnak a környezetükkel.

Jelen dolgozatban egy otthoni vagy vállalati környezetben elhelyezett esztétikus marker behatárolásával és felismerésével foglalkozunk.

A felismerés folyamata egy hagyományos digitális kameraképből történik. Az elkészített program célja az esztétikus marker behatárolása és információ tartalmának dekódolása. A dolgozatban lépésről lépésre bemutatásra kerül az újonnan kifejlesztett algoritmus, illetve annak működő implementációja a kezdeti színszűréstől a marker behatárolásán át annak végső nézőpont korrekciójáig. Továbbá magának az esztétikus markerben hordozott információnak az értelmezési módszerébe is betekintést nyerhet az olvasó.

A detektálás megvalósításához a Python programozási nyelvet használjuk Eclipse fejlesztői környezetben, valamint a Python nyelven belül az OpenCV képfeldolgozási függvénykönyvtárat. Ez az implementáció akár már jelenlegi formájában is használható lehet mobil robotokon tájékoztató feladatok ellátására.

II. AZ ESZTÉTIKUS MARKER

2.1. AZ ESZTÉTIKUS MARKER BEMUTATÁSA

Az általunk felismerni kívánt esztétikus markert a BME Gép és Terméktervezési Tanszékének hallgatói készítették él és be is mutatták egy TDK keretében [1] [2]. Az 1. ábrán látható az esztétikus marker egy minta példája:



1. ábra: Az esztétikus marker

Tehát a mobil robotnak ezt a falra felhelyezett markert kell eltérő távolságokból és látószögekből felismernie. Ahogy az látható öt különböző szín található meg benne: a sárga, a narancssárga, a kék, a világos lila és a sötétlila. A vízszintesen futó sávok nem rejtenek magunkban információt a robot számára, csupán csak díszítő elemként szolgálnak. Ebből következik, hogy a marker keresése során a belső 9 kis négyzetre és a benne lévő háromszögek felismerésére fog fókuszálni a megírt programkód.

A háromszögek négyzeten belüli elhelyezése, valamint az eltérő színek hordozzák magukban az információt. Attól függően, hogy az egyes háromszögek milyen módon vannak elforgatva a négyzeteken belül, más-más információt jelenthetnek a robot számára.

Szintén növelheti a tárolt információk mennyiségét az, ha egy újabb szint adunk a már meglévők közé.

A marker rendelkezik egy úgynevezett referencia ponttal vagy más néven bázisponttal. Ez a képen látható nagy négyzet jobb alsó sarkában lévő nagy sötétlila háromszög. A referencia pontnak a betájolás során van fontos szerepe, hiszen ez a referencia pont mindig a marker jobb alsó sarkában található.

Emellett meg kell említeni azt, hogy a robotra való tekintettel ügyelni kell arra, hogy viszonylag alacsonyan helyezzük el a markert a falon. Ez azért szükséges, mert a mobil robotra szerelt Kinect – ami a valós idejű képfeldolgozást végzi – körülbelül 1,2m magasan található. A képfeldolgozás szempontjából így lesz a legnagyobb esély arra, hogy az esetleges markereket fel tudjuk majd ismerni.

2.2. A MOBIL ROBOT

A mobil robotot [3] [4] [5] a BME bocsájtotta a Miskolci Egyetem rendelkezésére abból a célból, hogy az általuk épített mobil robot számára segítsenek a vezérlésért felelős programot elkészíteni. A mobil robot megalkotásánál fő szempont volt a hasznosság, törekedtek arra, hogy a véglegesen elkészült robot minél inkább tudja majd segíteni az emberek munkáját. A robotot úgy építették, hogy különböző dolgok szállítására legyen alkalmas, illetve a rászerezhető kar segítségével képes legyen mágneskártyás ajtók kinyitására. A felépítését tekintve két fő részből áll: az alapról és a rászerezhető felépítményből. A robot mozgását a holonomikus hajtás végzi, amely lehetővé teszi a síkon mindhárom szabadságfok menti elmozdulást.

Ezen a roboton fog futni az esztétikus markert felismerő program, azonban a program univerzálisan is használható más mobil robotok esetében is.



III. AZ ESZTÉTIKUS MARKER SZOFTVERES FELISMERÉSE

3.1. AZ ESZTÉTIKUS MARKER SZÍNEKRE BONTÁSA

Ahhoz, hogy a robot a Kinect segítségével érzékelni tudja az esztétikus markert szükség volt egy olyan megoldásra, ami kiemeli a markert a környezetéből. Legyen ez egy irodaház folyosója vagy akár egy lakóház szobája. Ennek megvalósítására a markert színösszetevőire bontottuk, megadva ezzel azt, hogy a feldolgozás során a robotnak milyen színű formák után kell majd kutatnia. A kék, a sárga, a narancssárga, a világos lila és a sötétlila színekre történő szűréssel az ezektől eltérő színeket figyelmen kívül hagyjuk. Persze ez önmagában még távol áll a marker végső felismerésétől, hiszen ha belegondolunk, ez az öt szín viszonylag gyakran megfordul a környezetünkben, akárhol is tartózkodjunk. További lépésként – miután a színekre történő szűrés megtörtént – célszerű az általunk használt markerre jellemző síkidomokat keresnünk. Ilyen például a háromszög, illetve esetlegesen a négyzet. Ezáltal a marker információt nem tartalmazó részeit is ki tudjuk szűrni.

3.2. A SZÍNSZŰRÉS MEGVALÓSÍTÁSA

A mobil robotnak 'real-time' módon kell majd az esztétikus markert kiszűrnie a környezetéből. Az esztétikus marker detektálása során a Kinect által készített több állókép segítségével teszteltük azt, hogy a marker felismeréséhez szükséges egyes programrészek megfelelően működnek-e. Tehát nem közvetlen videofolyamból történt a marker felismerés.

A Kinect-el készített, egy ilyen teszt kép az 2. ábrán látható.



2. ábra: A Kinect-el készített próba kép

Ahogy az a képen is látható, az információt tartalmazó marker mellett lévő díszítő sávok nem szerepeltek a tesztelésben. A középső négyzet kivágásra került és csak ennek behatárolására volt szükség.

A többi tesztképen a marker más-más szögben és távolságban helyezkedik el. A több tesztelési kép használata segítséget nyújtott abban, hogy ellenőrizzük a használt algoritmusok, függvények helyes működését.

Mielőtt a kamera képét szűrnénk a különböző színek alapján célszerű elvégezni némi elő műveletet, pontosabban zajszűrést. Zajszűrés azért van szükség, mert a felvett kép szemcsés, zavart lehet és nehezkesé teszi a pontos formák kiszűrését. Illetve egy tiszta, elsimított képen hatékonyabban végezhető el a színekre bontás művelete.

A kamera képen előműveletként kétszeres zajszűrést végeztünk. Szűrést külön függvénybe szerveztem, amely a Kinect által készített eredeti képet kapja meg bemenő paraméterként. A zajszűréshez az OpenCV Smooth függvényét használtam. Először *MEDIAN*, aztán pedig *BLUR* szűrést alkalmaztunk az alábbi módon:

```
cv.Smooth( in_image, out, cv.CV_MEDIAN, 3, 3);  
cv.Smooth( out, out, cv.CV_BLUR, 1, 1);
```

A szűrést követően a változás a 3. ábrán látható.



3. ábra: A zajszűrt kamerakép

A zajsűrűséget követően elvégezhetővé vált a rögzített kép színekre bontása, felhasználva az esztétikus markert alkotó színeket. A felbontáshoz meg kellett határozni, hogy az öt szín – amelyek a markert alkotják - milyen tartományon belül mozog a HSV színrendszerben. Maximum, illetve minimum értéket kellett meghatározni minden szín esetében.

A színek angol elnevezéséhez rendeltük hozzá a tartományokat a *cv.Scalar* függvény felhasználásával:

```
YELLOW_MIN = cv.Scalar (20, 80, 80)
YELLOW_MAX = cv.Scalar(55, 255, 255)

BLUE_MIN = cv.Scalar (70, 50, 0)
BLUE_MAX = cv.Scalar(130, 255,255)

ORANGE_MIN = cv.Scalar (0, 100, 100)
ORANGE_MAX = cv.Scalar(15, 230, 230)

PURPLE_MIN = cv.Scalar (135, 90, 40)
PURPLE_MAX = cv.Scalar(160, 170, 100)

PURPLE_L_MIN = cv.Scalar(140, 100, 90)
PURPLE_L_MAX = cv.Scalar(160, 140, 155)

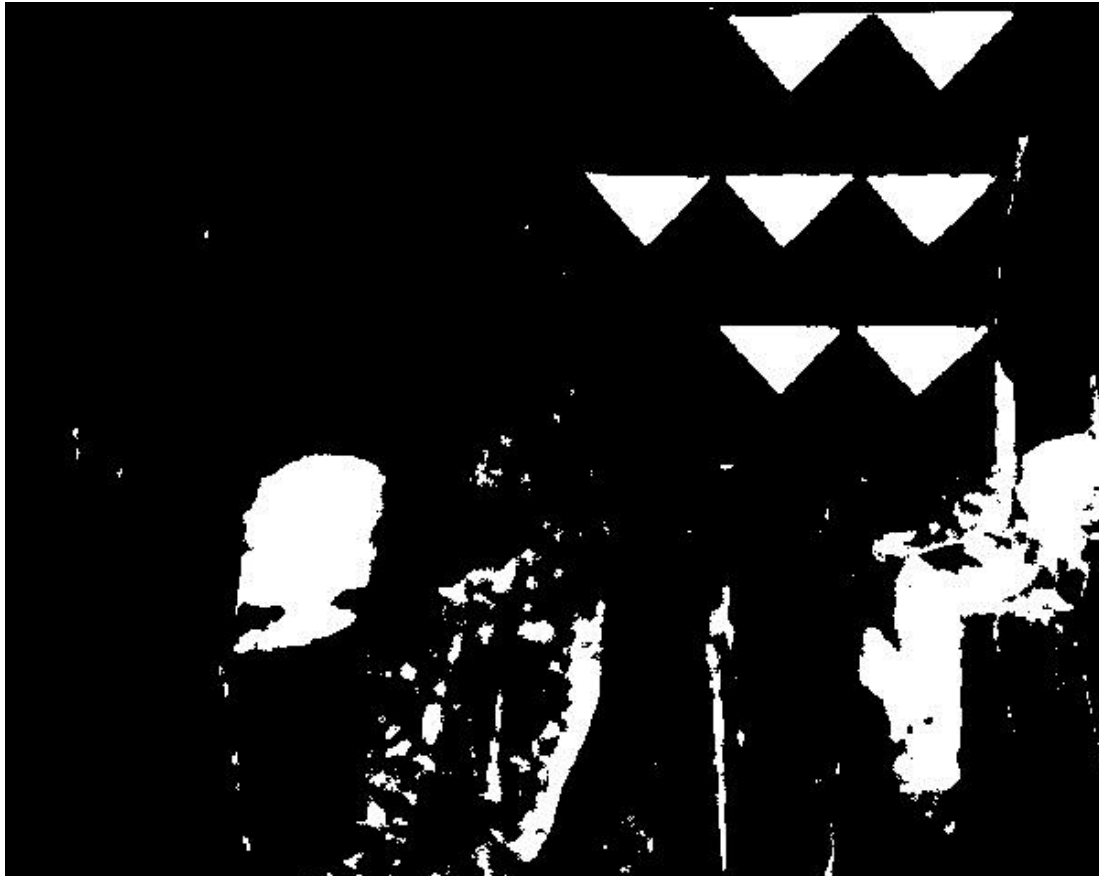
PURPLE_RED_MIN = cv.Scalar(0, 90, 100)
PURPLE_RED_MAX = cv.Scalar(8, 120, 128)
```

A zárójelben megadott számértékek ebben a sorrendben a HSV színtér színárnyalat (H), telítettség (S) és világosság (V) értékeit jelentik.

A lista végén szereplő, hatodik színre az úgynevezett túlsordulás miatt volt szükség. A 0-8 tartományban szereplő szín része a lilának, így szükségszerűen ezt a színárnyalatot is ki kellett szűrni a minél jobb eredmény érdekében. A pluszban kiszűrt színt hozzárendeltük a világos lilához a *cv.AddWeighted* függvény segítségével, így a későbbiekben csak 5 színnel kellett dolgoznunk.

Minden kiszűrt szín külön keretbe került lementésre fekete és fehér színeket használva. A fehér szín jelöli a kimentett képeken az adott színnek megfelelő részt.

A kiszűrt kék színt a 4. ábra mutatja.



4. ábra: A kék szín kiszűrése

Ebben az esetben az esztétikus markeren belül lévő kék háromszögeket nagy pontossággal tudtuk kiszűrni. Illetve látható, hogy a környezetben lévő azonos színek is megjelenítésre kerülnek. A későbbiek folyamán azt kellett megoldanunk, hogy a hét háromszögön kívül észlelt egyéb kék foltokat, formákat a robot figyelmen kívül hagyja.

A kék szín kiszűrése ritkának mondható, szinte teljesen optimális esetet eredményezett. A szűrések nagy többségében viszont olyan eredményeket kapunk, amelyeknél a marker háromszögei nem vehetőek ki teljes egészében. Hiányosak lehetnek köszönhetően annak, hogy a markert érő fény befolyásolja az érzékelt színt. A fény hatására az érzékelt szín az adott tartományon kívül esik, így nem kerül kiszűrésre.

Erre az eshetőségre szolgál példaként az 5. ábra, ahol a narancssárga szín került kiszűrésre.



5. ábra: A narancssárga szín szűrési eredménye

A hátralévő 3 szín a fentiekhez hasonló módon került detektálásra. A megtalált háromszögek pontossága, minősége eltérő lehet, attól függően melyik tesztképen végeztük el a szűrést, hiszen a változó körülmények hatással vannak az eredményekre.

3.3. KONTÚRKERESÉS A KISZŰRT KÉPEKEN

Miután sikeresen megtörtént az öt szín kiszűrése a már zajszűrt képről, a programkód további részében a kiszűrt képnek képeztük az inverzét, azaz az eddig fekete részek fehérek, az eddig fehér részek pedig feketék lettek. Ezt a műveletet a `cv2.threshold` OpenCv függvénnyel végeztük a következő módon:

```
ret1,thresh_yellow = cv2.threshold(yellow,127,255,1)
ret2,thresh_blue = cv2.threshold(blue,127,255,1)
ret3,thresh_orange = cv2.threshold(orange,127,255,1)
ret4,thresh_purple_dark = cv2.threshold(purple_dark,127,255,1)
ret5,thresh_purple_light = cv2.threshold(purple_light,127,255,1)
```

Minden esetben a zárójelben lévő első paraméter (yellow, blue, orange, purple_dark, purple_light) jelenti a bemenő képet, aminek fekete-fehér képnek kell lennie. A 127 az úgynevezett 'threshold' érték, a 255 a lehetséges maximális használható érték, az 1 pedig azt határozza meg, hogy a kimenet binárisan invertált kép lesz.

Az előállított kép a következőképpen néz ki a kék szín esetében a 6. ábrán.



6. ábra: A szűrt kék szín inverze

Természetesen a többi szín esetében is hasonló jellegű képet kapunk eredményként. Azzal, hogy előállt az öt inverz kép egy újabb lépés következik, mégpedig a kontúrvonalak megkeresése a kapott képeken.

A kontúrkeresést a `cv2.findContours` függvénnyel végeztük felhasználva az invertált képeket. A megtalált kontúrvonalakat `contours_yellow`, `contours_blue`, `contours_orange`, `contours_purple_dark`, `contours_purple_light` elnevezésű tömbökben tároltuk.

```
contours_yellow,h1 = cv2.findContours(thresh_yellow,1,2)
contours_blue,h2 = cv2.findContours(thresh_blue,1,2)
contours_orange,h3 = cv2.findContours(thresh_orange,1,2)
contours_purple_dark,h4 = cv2.findContours(thresh_purple_dark,1,2)
contours_purple_light,h5 = cv2.findContours(thresh_purple_light,1,2)
```

Az eltárolt kontúrvonalakat felhasználva a megtalált formák kirajzolása elvégezhetővé vált. Azonban a marker detektálása szempontjából számunkra csak azok a formák és alakzatok a fontosak, amelyek három csúcsponttal rendelkeznek, azaz a háromszögek. Éppen ezért minden színnél ki kell gyűjteni a háromszöget formáló kontúrvonalakat. Ennek megvalósításához a `cv2.approxPolyDP` nevű függvény tökéletes segítséget nyújtott, hiszen eltárolta a kontúron belül található csúcspontokat. Egy egyszerű IF elágazással eldönthető, hogy a csúcspontokat tartalmazó `approx` változóban három csúcspont szerepel, avagy sem.

Ezt a csúcspontokat rögzítő függvényt egy ciklusba kellett szerveznünk. Mégpedig egy olyan ciklusba, amely az öt különböző szín kontúrjain megy végig. Ezeket pedig a `contours_yellow`, `contours_blue`, `contours_orange`, `contours_purple_dark` és `contours_purple_light` változók tárolják.

Ellenőrzésképpen, hogy valóban kiválogatásra kerülnek-e a markert alkotó háromszögek megjelenítettük őket a zajszűrt képen.

A megjelenítés során, hogy megkülönböztethetőek legyenek az eltérő színű háromszögek, a háromszögek színének megfelelő BGR színekkel színeztük ki azokat.

A használt BGR szín értékek:

Sárga: (0, 255, 255)

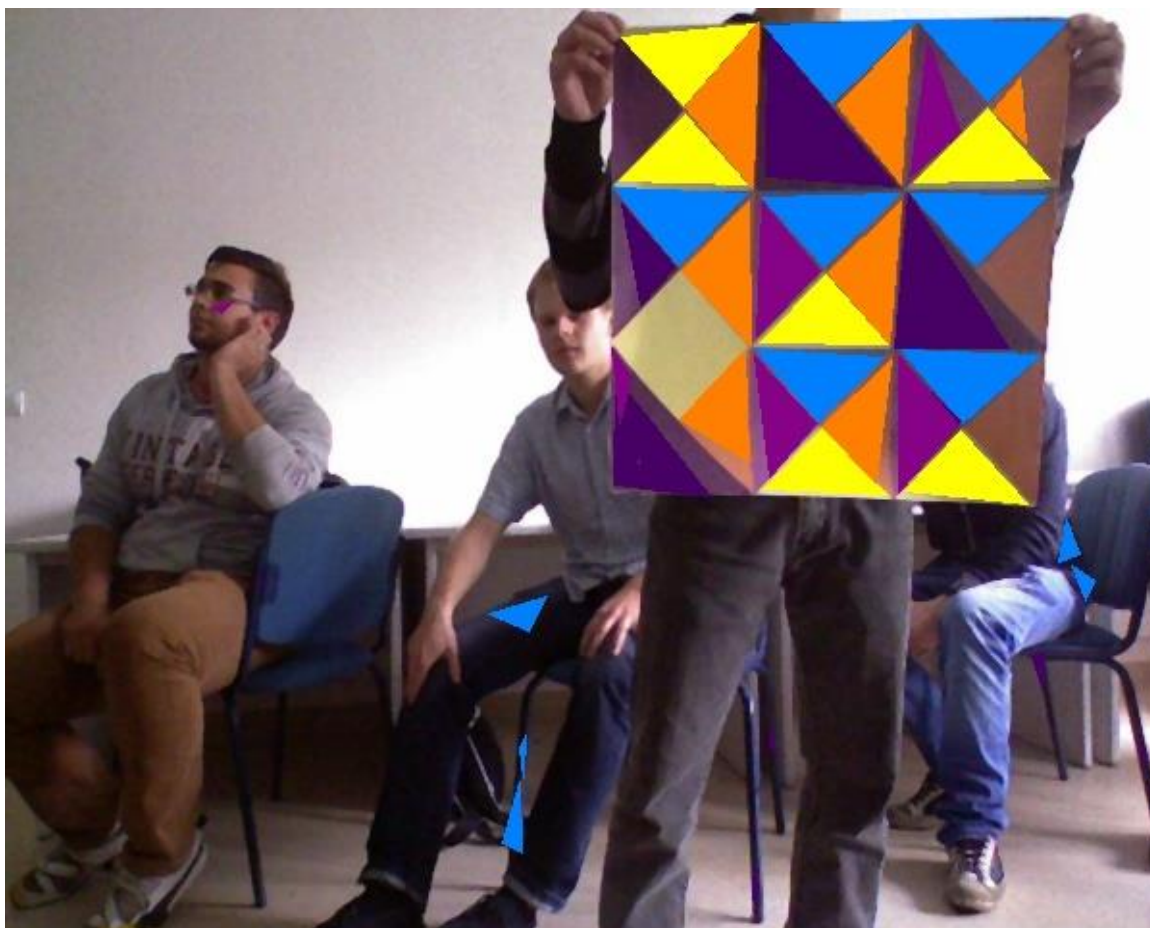
Kék: (255, 128, 0)

Narancs: (0, 128, 255)

Sötétlila: (100, 0, 71)

Világos lila: (134, 2, 134)

A kirajzolás utáni eredményt mutatja a 7. ábra.



7. ábra: A megtalált háromszögek

Látható, hogy a program valóban háromszögeket talál meg, azonban észrevehető, hogy a helyesen felismert markert alkotó háromszögek mellett más, fals háromszögeket is megjelenít. Éppen ezért az algoritmus további finomítást igényel.

Kezdetben arra törekedtünk, hogy az apró háromszögek ne kerüljenek megjelenítésre. A `cv2.contourArea` függvény kiszámítja a kontúrvonal által befoglalt terület nagyságát, így egy felső korlátot megadva kiszűrhetőek a túlságosan kicsi háromszögek, legyen az akármilyen színű.

A továbbiakban arra kellett törekedni, hogy a lefoglalt terület szempontjából megfelelő, de nem a markerhez tartozó háromszögek ne jelenjenek meg. A 8. ábrán látható, hogy az apró háromszögeket sikerült eltüntetni a korábban említett függvény segítségével.



8. ábra: Az apró háromszögek kiszűrése

A nagyobb méretű fals háromszögek kiszűréséhez három további eljárást használtunk. Elsőként az oldalak méretét vettük figyelembe. A cikluson belül a háromszög három oldala folyamatosan tárolásra került az alábbi módon:

```
elsooldal = math.sqrt(((approx[1][0][0]-approx[0][0][0])**2) +  
((approx[1][0][1] - approx[0][0][1])**2))  
  
masodikoldal = math.sqrt(((approx[2][0][0]-approx[1][0][0])**2) +  
((approx[2][0][1]-approx[1][0][1])**2))  
  
harmadikoldal = math.sqrt(((approx[2][0][0]- approx[0][0][0])**2) +  
((approx[2][0][1] - approx[0][0][1])**2))
```

Az *approx* tömb a háromszögek csúcspontjainak koordinátáit tárolja és ezekből kiszámítható az adott oldal hossza. A `***` pedig a Python nyelvben a hatványozást jelenti.

Ezt felhasználva egy elágazásban – szintén a cikluson belül – definiáltuk azt, hogy a háromszög egyik oldala sem lehet nagyobb, mint a másik két oldal 1.7 –szerese. Az 1.7-szeres szorzó kiválasztásánál figyelembe vettük, hogy ez a korlát ne legyen se túl szűk, se túl tág a háromszögek megtalálásánál. Más értékekkel is kipróbáltuk több kép esetében is, de ez az 1.7 bizonyult optimálisnak.

```
if ((elsooldal < masodikoldal * 1.7 and elsooldal < harmadikoldal *  
1.7) and ...
```

Az elágazása folytatódik tovább a második, illetve a harmadik oldallal is.

Ez a megoldás kiküszöbölheti az előző oldalon látott szabálytalanabb alakú narancssárga háromszöget. Viszont ez mégsem elegendő ahhoz, hogy minden szükségtelen háromszöget kiszűrjünk, hiszen más képek esetében előfordulhatnak olyan háromszögek, amelyek a véletlennek köszönhetően jóval szabályosabbak, éppen ezért átmennek ezen a szűrési feltételen.

A háromszögek oldal arányai mellett, egy másik ehhez hasonló módszert is alkalmaztunk. Szintén figyelembe vettem a megtalált háromszögek oldalait, de most ahelyett, hogy egymáshoz viszonyítottuk volna őket képeztük az oldalak átlagos hosszát. A folyamatosan futó ciklusban hozzáadtuk az éppen aktuális oldalértékeket megfelelő változókhoz. Így a három oldal értékei összeadódtak a más-más oldalakat tároló változóban.

Emellett egy külön változó segítségével számoltuk, hogy összesen hányszor fordulnak elő az egyes oldalak. Végül a megkapott számértékekkel elosztottuk azt a három változót, amely az összeadott oldalhosszakat tartalmazta és így megkaptuk a három oldal átlagos hosszúságát.

Az átlagos oldalhosszúságokat szintén egy elágazáson belül használtuk fel. Korlátként azt határoztuk meg, hogy az éppen aktuálisan megkapott és vizsgált oldal (legyen az első, második vagy harmadik) csak abban az esetben legyen elfogatható, ha a vizsgált oldal az átlagos oldalhossztól nem tér el nagyobb mértékben, mint az átlag 40%-a. Se negatív, se pozitív irányba. Ez a fajta szűrés további hibásan észlelt háromszögeket tüntetett el.

```
ossz_elsőoldal = ossz_elsőoldal + elsőoldal
ossz_masodikoldal = ossz_masodikoldal + masodikoldal
ossz_harmadikoldal = ossz_harmadikoldal + harmadikoldal
k=k+1
avg_elsőoldal=ossz_elsőoldal/k
avg_masodikoldal=ossz_masodikoldal/k
avg_harmadikoldal=ossz_harmadikoldal/k
if(((elsőoldal>avg_elsőoldal*0.4) and (elsőoldal < avg_elsőoldal*1.6)) and
...
```

Az elágazás ebben az esetben is folytatódik tovább a második és harmadik oldallal.

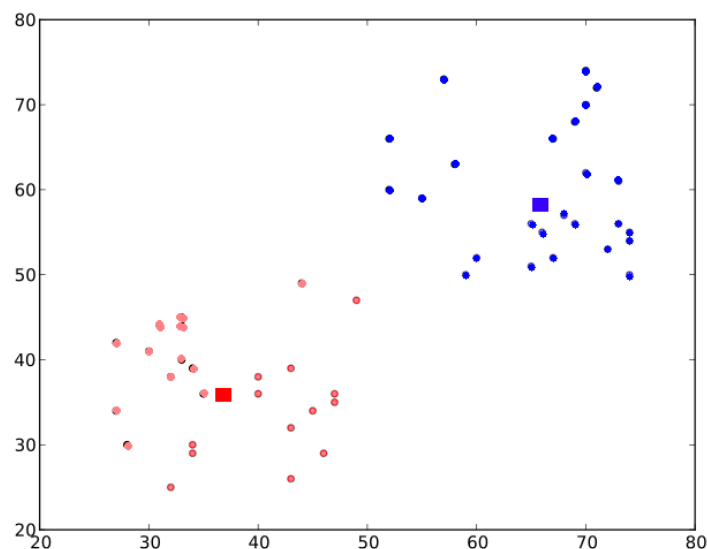
Az eddig említett három módszer, amely a hibás háromszögek eltávolítására irányult egészen jó eredménnyel szolgált. Azonban előfordult, illetve a későbbiek folyamán előfordulhat olyan szituáció, amikor a megtalált fals háromszög formailag szinte teljesen megegyezik a markert alkotó, információt hordozó háromszögekkel. Emiatt az átlagtól való eltéréssel és az oldalak egymáshoz viszonyított arányával nem szűrhetőek ki.

Először ezeket a megmaradt háromszögeket úgynevezett klaszterezéssel próbáltuk meg eltávolítani. Az OpenCV-n belül elérhető a K-Means klaszterezési módszer [6] [7] a `cv2.kmeans` függvénnyel.

A K-Means algoritmus nagyszámú adatot képes megadott számú csoportba, klaszterbe helyezni. Egy-egy klaszter a klaszterközépponttal (centroidtal) kerül azonosítása. Egy adott elem ahhoz a klaszterhez fog tartozni, amelyik középpontja legközelebb esik az elemhez.

Két klaszter kijelölése esetén a két klaszterközéppont először tetszőlegesen kerül kiválasztása. Ezt követően minden egyes elemet - amit a klaszterekbe akarunk szervezni – hozzávetünk mindkét középponthoz és abba a klaszterbe soroljuk be, amelyikhez közelebb található. Majd kiszámoljuk a két klaszteren belüli pontok egymáshoz viszonyított távolságát és annak megfelelően fog változni a klaszterközéppont.

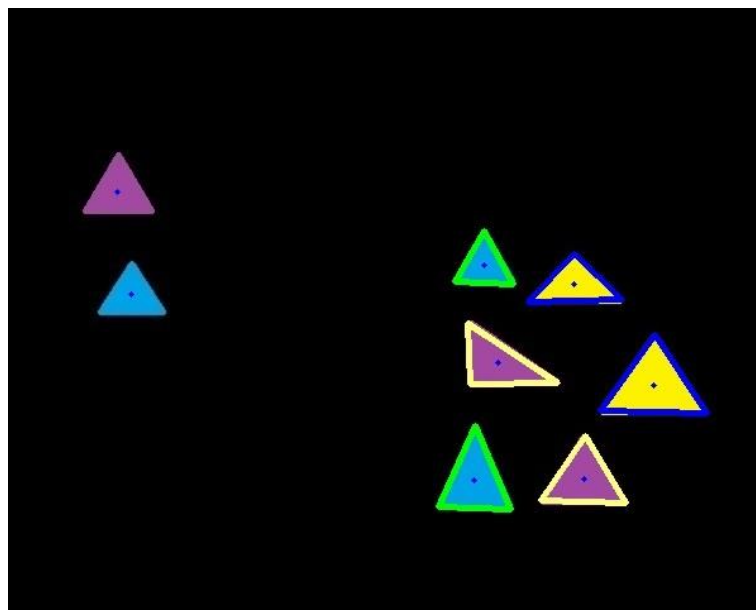
A két klaszterbe sorolt elemeket lenti ábra mutatja. A kék és a piros szín jelöli az egyes klasztereket. Az adott színnek megfelelő négyzetek a klaszter középpontjai. A 9. ábra a rendezés végső állapotát mutatja be.



9. ábra: Az elemek két klaszterbe rendezése

A K-Means klaszter kipróbálásához készítettünk egy saját képet, ahol fekete háttér mellett különböző színű háromszögeket rendeztünk el két csoportba. A beépített függvény segítségével azt szeretnénk volna elérni, hogy a több háromszöget tartalmazó csoport háromszögeit eltérő színekkel kerüljenek kijelölésre, így elkülönítve azokat.

A megírt kód sikeresen működött, az eredmény 10. ábrán látható.



10. ábra: A K-Means tesztelése

A felhasznált algoritmust a markernél megjelenő hamis háromszögek esetében is alkalmazni akartuk, azonban a próbálkozás nem hozott pozitív eredményt. A problémát az eljárás egyik alapvető korlátja okozta. Mégpedig az, hogy előre kell definiálnunk azt, hogy hány klaszterbe szeretnénk az adatokat sorolni.

Ha például két klasztert adunk meg akkor csak abban az esetben kapunk jó eredményt a kiszűrésre, ha a fals háromszögek egymáshoz közel és a markertől pedig távol helyezkedtek el. Ha a hibás háromszögek egymástól távol, több irányban vannak elszórva vagy egyáltalán nincs is fals háromszög a képen, akkor a kettőre beállított klaszter a markeren belül helyesen megtalált háromszögekből is vesz el. Ennek oka az a beállítás miszerint a legtöbb háromszöggel rendelkező klaszter kerül megjelenítésre. Tehát a klaszter alkalmazása nem vezet jó eredményre ebben az esetben. Hiszen minden egyes szituációnál más-más klaszterszám ad megfelelő eredményt. Azt azonban nem tudjuk, hogy mikor mennyire van szükség.

A távolabb eső, hibás háromszögek kiszűrése érdekében egy másféle megközelítésre volt szükség. A megoldáshoz a megtalált háromszögek súlypontjainak felhasználása vezetett. A háromszögek súlypontjának x és y koordinátáit a kontúrkeresésénél használt *approx* változó segítségével számítottuk ki:

$$S_x=((approx[0][0][0]+approx[1][0][0]+approx[2][0][0])/3);$$
$$S_y=((approx[0][0][1]+approx[1][0][1]+approx[2][0][1])/3);$$

Minden megtalált háromszög súlypontjának kiszámítása után itt is átlagot képeztünk, mégpedig a háromszög x és y koordinátáinak átlagát, így megkapva az átlagos súlypont elhelyezkedésének koordinátáit. Fontos megjegyezni, hogy ez a koordinátpont nem egyezik meg egyik háromszög súlypontjával sem, illetve ez a pont a négyzet alakú esztétikus marker középső részén található meg. Nem a középponttal azonos, de a közelében található.

Az átlagos súlypont megtalálását követően megvizsgáltuk, hogy minden felhasznált súlypont milyen távol esik az átlagostól. A háromszögek súlypontja eltérő távolságokra voltak a kiszámított átlagos súlyponttól. Ezeket a távolságokat is átlagoltuk.

A meglévő átlagos súlypontból és az ettől a ponttól számított többi súlypont átlagos távolságából létrehozhatóvá vált egy újabb szűrési feltétel. Mégpedig azáltal, hogy egy képzeletbeli kör húzható az átlagos súlypont köré úgy, hogy a kör sugara az átlagos súlyponttól mért átlagos távolság lesz. Ennek köszönhetően a markertől távolabbi, hibás háromszögek kiküszöbölhetőek.

A kiszűrés folyamán az átlagos távolságtól való eltérésnél bizonyos tűrést kellett használnunk, mert ha csak önmagában, az azon a távolságon belüli súlypontokkal rendelkező háromszögeket vettem volna figyelembe akkor többi olyan háromszög sem került volna megjelenítésre, amelyek a marker részét képezik. Többszöri tesztelés után végül az átlagos távolság kétszeresénél kell kisebbnek lennie a vizsgált háromszög súlypontja és az átlagos súlypont által meghatározott távolságnak.

A viszonylag nagynak tűnő ráhagyás annak is köszönhető, hogy a végleges átlagos súlypont nem azonnal áll elő, hanem a cikluson belül többször változik mindaddig, amíg a végső értéket el nem nyeri. Az alábbi programrészlet végzi el a fent említett műveletek nagy részét:

```
for i in range(len(ossz_tav_list)):
    if (avg_S_tav * 2 > ossz_tav_list[i]):
        approx_x=[tri[i][0][0][0],tri[i][1][0][0],tri[i][2][0][0]];
        approx_y=[tri[i][0][0][1],tri[i][1][0][1],tri[i][2][0][1]];
        ossz_app_x.append(approx_x)
        ossz_app_y.append(approx_y)
        cv2.circle(imgfull, (avg_S_x, avg_S_y), 5, (0,255,0), -1)
        cv2.drawContours(imgfull,[tri[i]],0,fillcolors[con[i]],-1)
```

A későbbiekben a jobb láthatóság kedvéért egy zöld ponttal jelöltük meg az átlagos súlypont koordinátáját. Ezt az előző oldalán ismertetett kódrészletben a `cv2.circle` függvény végzi el. Az első paramétere jelenti azt a képet, amire rajzolni szeretnénk. A második az x és az y koordináta. A harmadik a pont nagysága, a negyedik a szín. Az ötödik paraméterként megadott -1 pedig azt adja meg, hogy a pont tömör legyen.

Összegzésképpen a 11. ábrán látható az eredeti zajszűrt kép, amikor még a négy szűrő eljárás nem volt használva.



11. ábra: Az eredeti zajszűrt kép a fals háromszögekkel

A 12. ábrán pedig már a négy szűrő eljárás használata után készült. Látható, hogy a szükségtelen, zavaró háromszögek eltűntek és a marker háromszögei is többségben detektálásra kerültek. Az nem probléma, hogy nem sikerült megtalálni az összes háromszöget, mert anélkül is meghatározható az esztétikus marker pozíciója.

A zöld pont is észrevehető, amely az átlagos súlypontot jelöli. A többi próbakép esetében is a marker középső részén található meg.



12. ábra: A hibás háromszögek eltávolítása

3.4. AZ ESZTÉTIKUS MARKER BEHATÁROLÁSA

Azzal, hogy sikerült megtalálni a markert felépítő háromszögek túlnyomó részét és elkülöníteni őket a környezetében észlelt egyéb, problémát jelentő háromszögektől, az esztétikus marker felismerésének művelete tovább folytatódhat.

A hibás háromszögek kiszűrése kulcsfontosságú szerepet játszik a marker végső felismerésében, hiszen a további algoritmusok a már meglévőkre épülnek. Feltételezve azok helyes működését. Amennyiben mégis kirajzolásra kerülne olyan hibás háromszög vagy háromszögek, amik nem részei a markernek, akkor abban az esetben a marker szélei nem lesznek jól behatárolva.

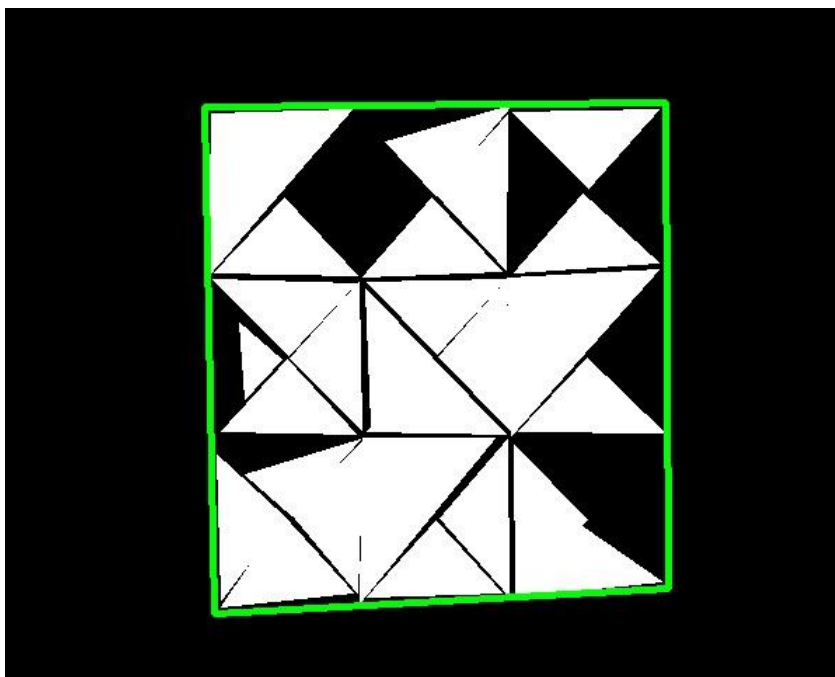
A Kinect által rögzített képen az esztétikus marker eltérő pozícióban jelenhet meg a képen, attól függően, hogy a robot milyen távolságból és látószögből rögzíti azt. Tehát a marker mérete annak függvényében fog változni, hogy milyen közelről vagy távolról rögzíti a Kinect.

A következő feladatunk az volt a háromszögek megtalálását követően, hogy minél pontosabban próbáljuk behatárolni a négyzet alakú marker széleit, a megtalált háromszögekből kiindulva. A megoldandó problémát az jelentette, hogy a markert alkotó 33 háromszögből jó pár darabot nem sikerült felismerni. A markert csak abban az esetben vizsgáljuk tovább, ha elegendő számú háromszöget sikerült megtalálni. A markerben található háromszögek száma miatt, ezt a korlátot 15-re állítottuk be. 15-nél kevesebb esetén eldobja a vizsgált frame-et és újat kér be. A tesztelés során minden megvizsgált kép esetében 20-nál több háromszöget sikerült megtalálni. Így folytathatóvá vált a marker felismerése a marker oldalainak behatárolásával, ez nagyobb gondot okozott, ha hiányoztak a sarokpontok.

Először egy olyan módszerrel próbálkoztunk, ahol a megtalált különböző színű háromszögeket egységesen fehér színűre színeztem és a háttért pedig feketére állítottam. A marker körvonalát a beépített `cv2.drawContours` függvénnyel próbáltuk megtalálni, úgy hogy csak akkor rajzoljon kontúr vonalat, ha a csúcspontok száma egyenlő néggyel.

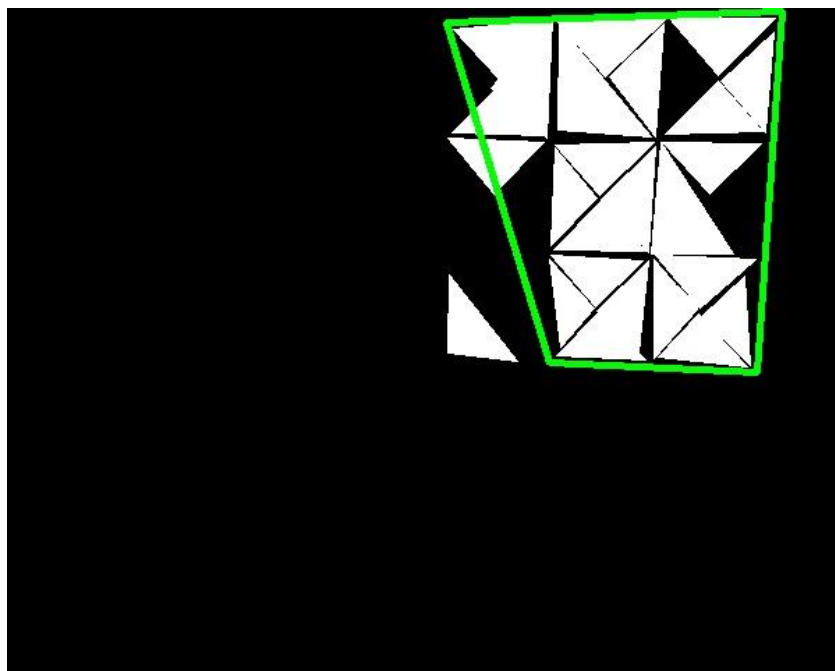
A tesztelés során ez a módszer viszonylag használható eredményeket adott, abban az esetben, amikor sok háromszöget sikerült detektálni, illetve ezek egymáshoz közel helyezkedtek el és összefüggő csoportot alkottak.

A 13. ábra mutatja a sok háromszögnél helyesen működő körbehatárolást.



13. ábra: A helyes körvonal berajzolás

Amikor a háromszögek nem helyezkedtek egymáshoz közel a körvonal megrajzolás hibásan működött. Ez az eset a 14. ábrán látható.



14. ábra: A marker hibás körvonalazása

Ez a módszer tehát nem bizonyult elég megbízhatónak a kezdeti sikerek ellenére. A későbbiekben nem sikerült orvosolnunk ezt a problémát, éppen ezért egy új megoldást kellett keresnünk.

A másik módszernél az érzékelt marker oldalainak kiszűréshez szükségszerűvé vált megtalálni minden oldal legszélső pontját. Korábban kigyűjtésre került a megtalált háromszögeket alkotó pontok x és y koordinátái külön tömbökbe. Ezekben a tömbökben az azonos indexű elemek alkotják a háromszögek egyes csúcspontjait.

A marker jobb szélének megtalálása esetén a háromszögek x koordinátáit tartalmazó tömbből maximum keresés segítségével kiválasztottuk a legnagyobb érték indexét. Ezt az indexet felhasználva megkerestük az y koordinátákat tároló tömbben azt a y koordinátát, amelyik az x -hez tartozik. Hiszen a korábban említett módon az y koordináta ugyanazzal az indexszel rendelkezik a saját tömbjén belül, mint a x koordináta.

Azáltal, hogy a jobbszélső pont elhelyezkedését így meghatároztuk a továbbiakban úgy folytattuk a műveletet, hogy a koordináta pont x értékére egy bizonyos tűrést állítottunk be. Optimálisnak az 5%-os tűrés bizonyult. A tűrést a képernyő bal oldala felé kell értelmezni, mivel a jobb oldal legszélső pontját találtuk meg és annak az x koordinátája a legnagyobb. A tűrés megadásával további olyan x koordinátákat találtunk, amelyek a legszélső pont x koordinátájával függőlegesen közel egy vonalba esnek.

Ha a jobbszélső x koordináta értéke például 560, akkor az 5%-os korlát hatására olyan x értékeket fog találni (amennyiben léteznek), amelyek 532 és 560 közé esnek majd. Ideális esetben 2-3 ilyen x érték is létezni fog és miután megvannak az indexüket felhasználva meghatározható a hozzájuk tartozó y érték is.

Az összetartozó értékek megtalálása után azt számítottuk ki, hogy a szűrési tartományon belül lévő pontok közül az adott pont milyen távol esik a legszélső, maximális x értékkel rendelkező ponttól.

A legnagyobb távolságra lévő pont kiderítésére azért van szükség, mert a marker oldalának érzékeléséhez az aktuális oldal két pontját kötöttük össze a *cv2.line* függvénnyel. Ha belegondolunk az oldal behatárolása sokkal pontosabb lehet, ha az oldal két egymástól távolabbi pontját kötjük össze. Egymáshoz viszonylag közelebb található pontoknál a problémát az okozhatja, ha azokat a háromszögeket, amelyek a marker szélén találhatóak nem sikerül teljes egészében felismerni, azaz ha az egyes részek levágásra kerülnek. Ezáltal a közeli pontok nagy valószínűséggel nem lesznek függőlegesen egy síkban és a két pont összekötésének meghosszabbítása nem a marker szélén fog végigfutni.

Az esztétikus marker bal oldalának megrajzolása hasonló módon történt, mint a jobb oldalé. A különbség a kettő között az volt, hogy a baloldal esetében az x értékeket tartalmazó tömbből a legkisebb értéket kellett kiválasztani. Az jelentette a marker legszélső bal pontjának x koordinátáját. Az y koordinátáját szintén az x érték indexe alapján határoztuk meg.

A x érték tűrésénél – ahhoz, hogy a x értékéhez közeli egyéb x-eket is megtaláljunk – egy újabb megoldandó probléma merült fel. A problémát az idézte elő, hogy azokban a szituációkban, amikor a marker nagyon közel van a Kinect-hez (azaz a rögzített frame-et szinte teljesen magába foglalja), akkor a marker bal széle a kép bal szélétől nagyon kicsi távolságra kerül. Ilyenkor a legkisebb x érték akár 10 vagy annál kisebb is lehet, ekkor az 5 %-os tűrés nem vezet pozitív eredményre, hisz nagy valószínűséggel nem fog találni a legszélső pont mellett más pontokat, hacsak nincs függőlegesen szinte teljesen egyvonalban a minimum x értékkel.

A minimális határérték meghatározását az alábbi módon sikerült kivitelezni:

$$\text{min_hatar} = 640 - (640 - \text{MIN_X1}[0]) * 0.95$$

A *MIN_X1[0]* érték a legszélső bal oldali pont x koordinátát jelenti. Mivel ez a szám túl kicsi volt, hogy a 105 %-át vegyem, ezért kivontuk a 640-ből, ami a Kinect által rögzített kép szélességével egyezik meg. Ennek a nagyobb számnak már tudtuk venni a 95%-át, de mivel a minimum értéktől kell nagyobb 5%-al ezért újra ki kell vonni a 640-ből.

A marker felső és alsó oldalát a jobb és a balhoz hasonlóan határoztuk meg. Az eltérés abban nyilvánult meg a korábbiakhoz képest, hogy most első lépésként az y értékeket tartalmazó tömbben kellett keresnünk. A felső oldal legtávolabbi pontjának meghatározásánál a maximális y értéket kellett kiválasztani és annak index alapján meghatározni a hozzá kapcsolódó x koordinátát. Az alsó oldal esetében pedig a minimális y koordinátájú pontot kellett megkeresni.

A felső és alsó szélső pontokhoz szintén korlátok megadásával kerestük olyan pontokat, amelyek a megadott tűrési tartományon belül van, szóval a felső oldalnál a maximális y érték közelében, az alsó oldalnál pedig a minimális y érték közelében vannak.

Az alsó oldal behatárolásánál a baloldalnál használt módszert használtuk. Szükség volt rá, hiszen ha a korábban említett módon a markerről túl közeli kép készül, akkor nem csak a baloldaltól számított x érték, hanem a frame alsó oldalától számított y érték is túlságosan kicsi lehet.

A használt megoldás a következőképpen néz ki:

$$\text{min_hatar}=480-(480-\text{MIN_Y1}[1])*0.95$$

Látható, hogy ennél a megoldásnál a 640 helyett a 480-ból kell elvégezni a kivonásokat. Ez azért van, mert a vízszintes vonalak detektálásánál már a kép magasságát kell felhasználni a szélessége helyett, ami minden esetben 480 képpont.

A program felépítését tekintve az esztétikus marker behatárolásával foglalkozó programkód külön függvényben kapott helyet. Meghívása értelem szerűen azután történik, miután már sikerült a markert alkotó háromszögeket megtalálni és a szükségteleneket pedig kiszűrni. Három bemenő paraméterrel rendelkezik: az x értékeket tároló tömb, az y értéket tároló tömb, illetve egy frame amire a későbbi kirajzolás történik.

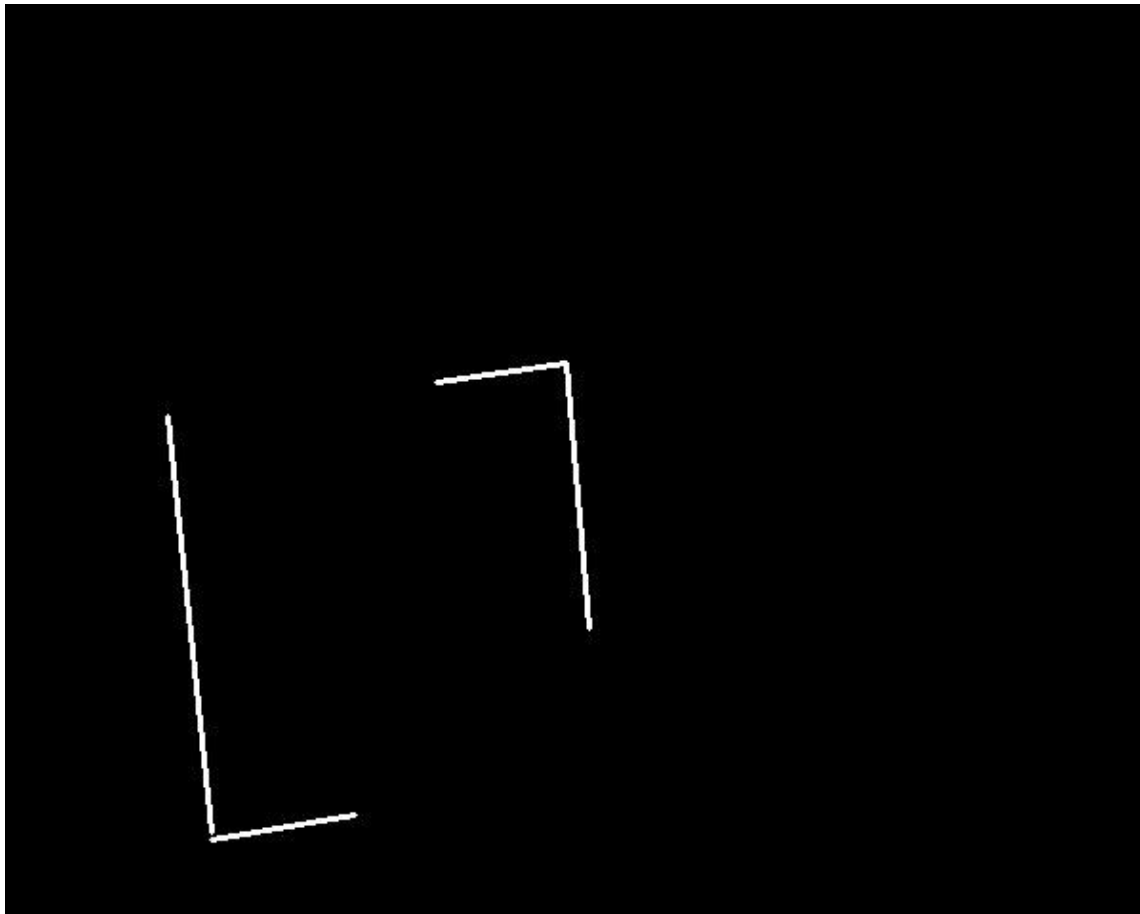
A 15. ábrán látható próba képen végeztük el az oldalak behatárolását. A képen elég háromszög felismerésre került, ezáltal gond nélkül elvégezhetővé vált az oldalak keresése. A 33 háromszögből megtalált 23 darab úgy helyezkedik el, hogy a négyzet alakú marker 3 csúcsát megfelelően kirajzolják. A jobb alsó sarokban lévő nagy sötétlila háromszög hiánya nem okozott gondok, mivel a marker jobb oldalát nézve lesz elég pont ahhoz, hogy a két legtávolabbit összekötve kirajzolják a marker szélét.



15. ábra: Az oldalak behatárolására használt kamerakép

A korábbiakban a jobb alsó sarokban lévő nagy lila háromszög melletti sárga „négyzetről” nem esett szó. Valójában nem is négyzetről van szó, hanem két sárga háromszögről, amelyeket egymás mellett helyeztek el. Ez a „négyzet” egy tesztképen sem került kiszűrésre, mivel a használt algoritmus csak háromszöget keres.

Az előző oldalon szereplő képen ha elvégezzük az oldalak behatárolását akkor a 16. ábrán látható eredményt kapjuk.



16. ábra: A marker behatárolt oldalai

A vonalak berajzoláshoz egy teljesen új képet hoztunk létre. Ha ráillesztenénk az eredeti képre a kirajzolt vonalakat, akkor nagy százalékban helyesen jelölné ki a marker széléit. A többi teszt képen is ehhez hasonló eredményeket kaptunk, tehát az algoritmus megfelelően működik.

A továbbiakban az volt a feladatunk, hogy a marker teljes kerülete legyen behatárolva. Mivel szükség lesz a későbbiekben a marker négy sarokpontjának behatárolására.

3.5. EGYENES ILLESZTÉS A MARKER OLDALAIRA

Az előző fejezetben ismertetett módszerrel - amikor a marker legszélső pontjait meghatároztuk és azokat összeköttöttünk az adott oldal egy másik pontjával – általános esetben nem kivitelezhető az, hogy a felismerni kívánt marker egyes oldalainak teljes hosszát kirajzoljuk. Ez azért nem lehetséges, mert az esetek nagy többségében mindig lesz olyan, a marker szélén található háromszög, amely nem kerül majd felismerésre. Ennek következtében a háromszögek alkotó pontok sem fognak szerepelni a x és y koordinátákat tároló tömbökben, így biztosan nem kerülhetnek detektálásra azok a szélső pontok, amelyekkel teljesen kirajzolhatóvá válna a négyzet alapú marker adott teljes oldalhossza.

Ezekből kifolyólag az ismertetett módszert tovább kellett bővítenünk. Az alapvető elgondolás az volt, hogy a lehetséges négy irányból, a kép szélétől indulva egyeneseket közelítünk a marker oldalai felé, majd amint elérik azokat, ráillesztjük őket.

Végül az egyenesek közelítésére nem volt szükség az oldalak irányából, köszönhetően annak, hogy az esztétikus marker mind a négy oldalának valamennyi szakaszát sikerült behatárolni. Még ha nem is teljesen.

A továbbiakban az tűnt kézenfekvő ötletnek, ha megpróbáljuk a megtalált szakaszokat kibővíteni, azaz egyeneseket illeszteni rájuk. Így várhatóan a meghosszabbított négy oldal valahol találkozni fog, létrehozva az egymással szomszédos oldalak metszéspontjait. Nem 100%-os bizonyossággal, de vélhetően a megtalált metszéspontok a marker négy sarokpontjának közvetlen közelében lesz.

Az egyenes illesztést megvalósító, külön függvény értelemszerűen az oldalak részleges szakaszait kirajzoló függvény eredményével fog dolgozni. A bemenő paraméternél megkapott képet először 8 bites, 1 csatornás képpé kell alakítani. Ennek megvalósítás az alábbi módon történt:

```
gray = cv2.cvtColor(frame3, cv2.COLOR_BGR2GRAY)
```

Az átalakítást azért kellett elvégezni, mert az egyenesek illesztéséhez szükség van a kontúrvonalak meghatározására. A kontúrkeresést a `cv2.findContours` végzi és csak olyan képeken tudja elvégezni a műveletet, amelyek vagy 8 bites 1 csatornás, vagy úgynevezett 32 bites 'signed single' csatornás képek.

Az egyenes illesztést a `cv2.fitLine` függvénnyel oldottuk meg. A függvény felhasználja a `cv2.findContours` által megtalált kontúrponthoz és kimenetként előállítja a vonalak paramétereit egy 4 elemű vektor formájában. A paraméterek meghatározása után a vonalak szélső értékeit kellett kiszámítani. Az alábbi kódrészlet a `fitLine` függvény, illetve a szélső értékek megoldását mutatja:

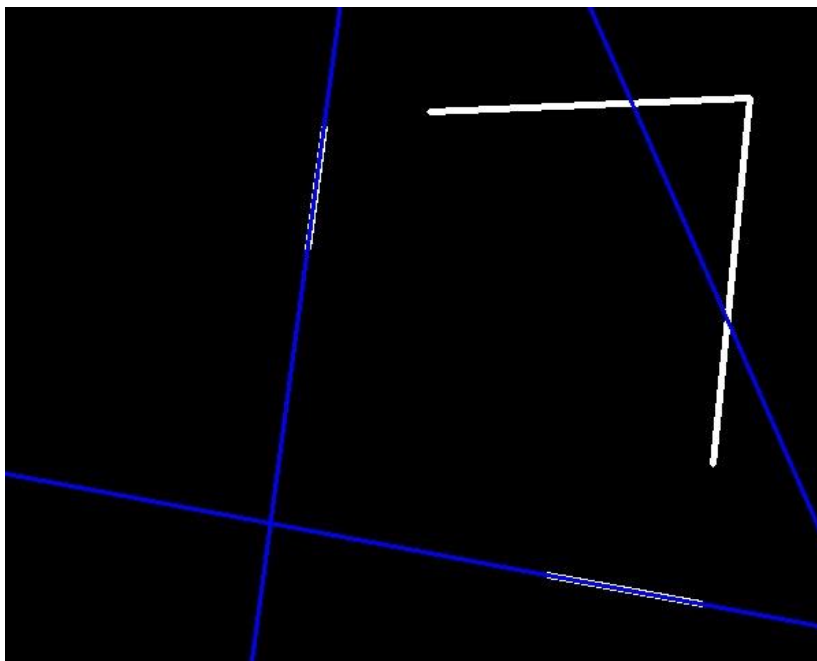
```
[vx,vy,x,y] = cv2.fitLine(cnt,cv2.cv.CV_DIST_L2,0,0.01,0.01)
lefty = int((-x*vy/vx) + y)
righty = int(((gray.shape[1]-x)*vy/vx)+y)
```

A `fitLine` függvénynél a kontúrponthoz megadása mellett (`cnt`) alapértelmezett függvény beállításokat használtunk, amelyek távolságokra vonatkoznak és bizonyos pontosságokat határoznak meg. A `vx` és `vy` a normalizált vektort jelentik, az `x` és `y` pedig egy pontot jelöl a szakaszon.

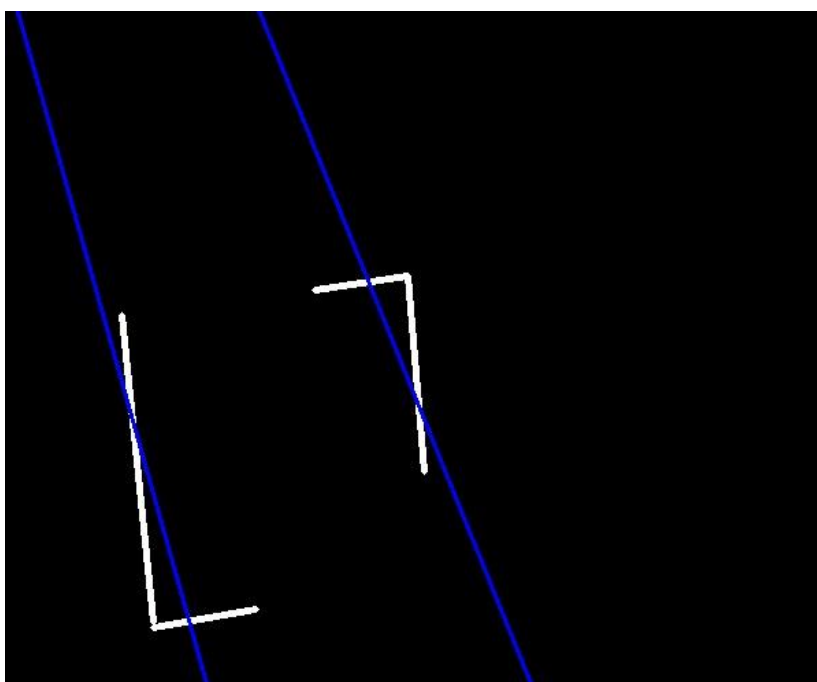
A `fitLine` függvényt ciklusba kellett szervezni, mivel minden oldal esetén meg kell vizsgálni az oldalszakaszok kontúrjait. Az oldalankénti szakaszok szélső értékeinek meghatározása után a `cv2.line` segítségével kirajzoltattuk a szakaszokra illesztett egyeneseket.

Az első próbálkozások során az algoritmus nem működött megfelelően. A futtatás eredményei a következő oldalon láthatóak.

Az egyenes illesztés korai eredményeit a 17. és 18. ábra mutatja. Kék színnel jelöltük az illesztett egyeneseket. A fehér szakaszok pedig a marker korábban behatárolt oldalait jelölik.



17.ábra: Az egyenes illesztés kezdeti eredménye



18.ábra: Az egyenes illesztés kezdeti eredménye egy másik beállításnál

Látható a két képen, hogy az egyenesek illesztése valóban nem jól valósul meg. Azonban az is megfigyelhető az első képen, hogy az esztétikus marker bal, illetve alsó oldalára a kék egyenes tökéletesen illesztésre kerül. Tehát a használt módszer bizonyos esetekben jól működik.

Már ezen a két képen is látható, hogy az alapvető problémát az okozza, ha a marker valamelyik két szomszédos oldalát behatároló szakasz találkozik egymással. Ezekben az esetekben történt az, hogy egyszerre a két egyenesre akart egy darab kék egyenest illeszteni, amely hibához vezetett. A teszt képek között előfordult az, hogy a marker behatárolásánál nem csak a két oldala ért össze egymással, hanem mind a négy. Ilyenkor tévesen egy egyenest próbált ráilleszteni a vonalakra a négy helyett.

Az egyenes illesztés jól hajtodik végre akkor, ha az adott szakasz nem érintkezik semmilyen másik szakasszal. Ezt az elméletet alátámasztva megvizsgáltuk azt, hogy az egyes szakaszoknál a kontúrkeresést követően hány csúcspontot (*approx*) talált a program.

Egy olyan kép során, ahol csak két oldal ér egymáshoz és a másik kettő különálló, ott arra jutottunk, hogy a különálló szakaszoknál két csúcspontot talál, míg az egymással érintkezőeknél viszont négyet. További tesztelés céljából ellenőriztük egy olyan képen is a megtalált csúcspontok számát, ahol mind a négy oldal találkozik. Itt egy nyolc csúcspontból álló kontúrvonalat talált. Ennek megfelelően, ha három oldal van összekapcsolódva és egy külön, akkor hat csúcspontból és egy két csúcspontból álló kontúrt talál meg.

Korábban említésre került az a megállapítás, miszerint az egyenes illesztés akkor sikeres, ha az adott oldalszakasz nem érintkezik további másikkal. Ennek feltétele pedig az, hogy a kontúrvonalak keresése során a szakasz két csúcsponttal rendelkezzen.

Tehát ahhoz, hogy az egyeneseket megfelelően tudjuk ráilleszteni az oldalakra szükség van a négy, a hat és a nyolc csúcspontból álló kontúrok szétbontására az illesztés során.

A szétadarabolás során két csúcspontból álló részekre kell a bontást elvégezni. Ennek érdekében megvizsgáltuk az egyenes illesztését végző függvényen belül azt, hogy hány csúcsponttal rendelkezik vagy rendelkeznek a megtalált kontúrok. Az eredmény kettő, négy, hat vagy nyolc lehetett. Külön elágazásba szerveztük a négy variációt.

Ha a csúcspontok száma kettővel egyezett meg, akkor változtatás nélkül elvégezhető volt az illesztés.

A másik három variációnál a csúcspontokat tároló *approx* tömböt két csúcspontból álló kisebb tömbökre osztottuk az alábbi módon. Itt a csúcspontok száma négygyel egyezett meg:

```
if len(approx) == 4:
    approx1 = np.array([approx[0],approx[1]])
    approx2 = np.array([approx[2],approx[3]])
    for app in [approx1, approx2]:
        [vx,vy,x,y] = cv2.fitLine(app,cv2.cv.CV_DIST_L2,0,0.01,0.01)
        lefty = int((-x*vy/vx) + y)
        righty = int(((gray.shape[1]-x)*vy/vx)+y)
        if (abs(lefty) < 1<<16L) and (abs(righty) < 1<<16L):
            cv2.line(imgfull,(gray.shape[1]-
            1,righty),(0,lefty),(255,255,255),2)
            cv2.line(frame3,(gray.shape[1]-
            1,righty),(0,lefty),(255,255,255),2)
```

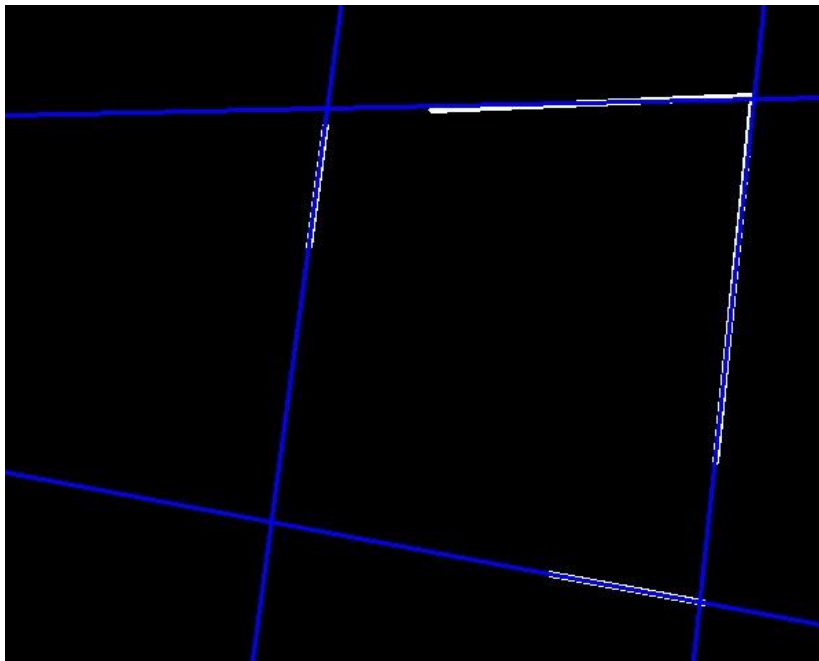
A két kisebb tömb közül az egyiket az *approx* tömb 0. és 1. eleme alkotja, amíg a másikat 2. és 3. eleme. A felbontást követően a két kisebb tömbön külön végezzük el a *cv2.fitline* függvényt és keressük meg a szélső értékeket. A kettébontásnak köszönhetően két külön szakaszt vizsgáltunk az összefüggő egy helyett.

Akkor, ha a csúcspontok száma hattal egyenlő, akkor azt három részre osztjuk fel, ahogy az látható:

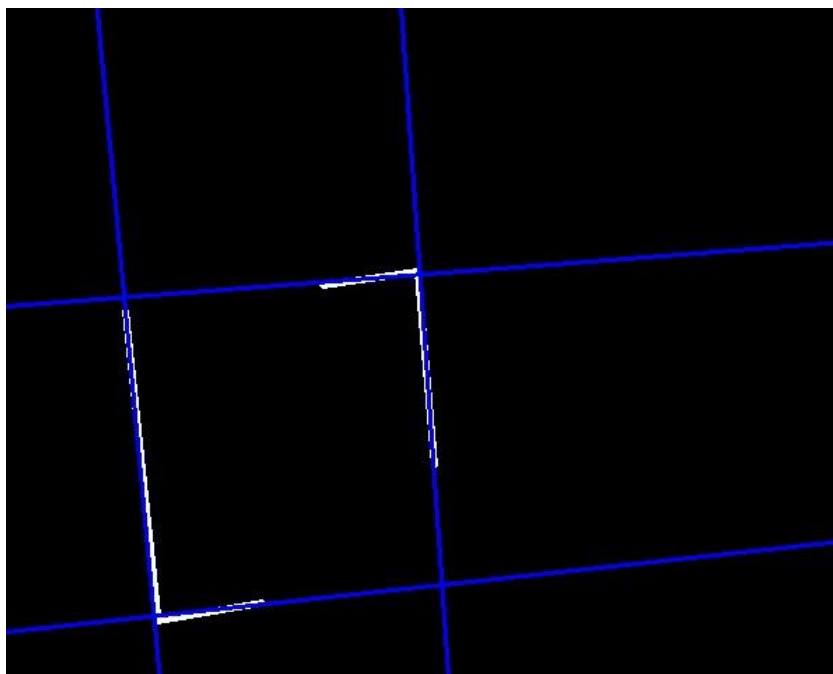
```
if len(approx) == 6:
    approx1 = np.array([approx[0],approx[1]])
    approx2 = np.array([approx[2],approx[3]])
    approx3 = np.array([approx[4],approx[5]])
    for app in [approx1, approx2, approx3]:
        ...
```

Nyolc esetében pedig négyre.

Ezekkel a kiegészítésekkel az egyeneseket sikerült megfelelően a négy oldalra illeszteni. Ezt mutatja a 19. és 20. ábra.



19. ábra: Sikeres egyenes illesztés az oldalakra

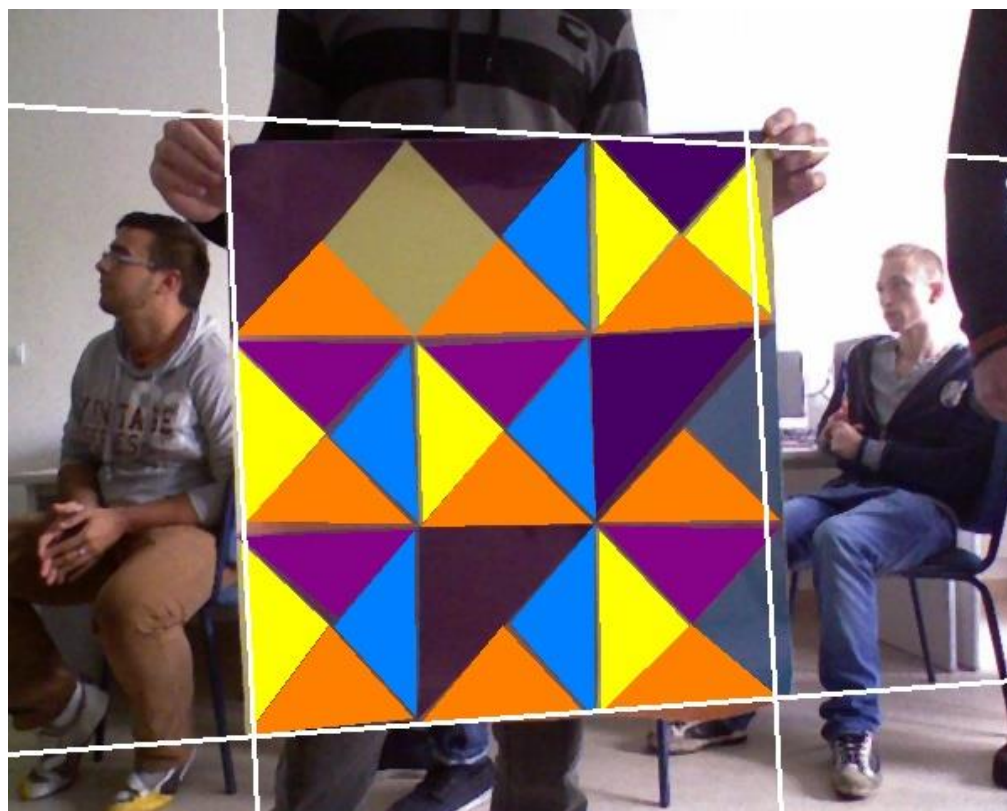


20. ábra: Sikeres illesztés a másik beállításnál is

Az egyenesek ráillesztésével a marker helyzete a Kinect által rögzített aktuális képen nagy pontossággal meghatározható. Látható az is, hogy nagyon fontos szerepet játszott a zavaró, hibás háromszögek teljes kiszűrése.

Amennyiben tartalmazott volna fals háromszöget vagy háromszögeket a felvett kép, akkor már az oldalak behatárolása is hibásan történt volna meg és abból adódóan az oldalakra történő egyenes illesztése is.

A jobb szemléltetés végett az egyenesen illesztés megvalósításával foglalkozó függvény eredményét az eredeti képre helyeztük vissza. Ez látható a 21. ábrán.



21. ábra: Az eredeti képre visszahelyezett oldal egyenesek

3.6. A MARKER SAROKPONTJAINAK MEGTALÁLÁSA

Következő műveletként a már egyenesekkel behatárolt esztétikus marker sarokpontjait kerestük meg. A sarokpontok megtalálására azért lehet szükség, mert a továbbiakban a markert felismerésének folyamatát képezi majd az, hogy csak magát a markert elemezzük tovább. Ehhez azonban a rögzített képről ki kell, hogy vágjuk azt.

A marker kivágásának legegyszerűbb módja az lehet, hogy mind a négy lehetséges sarokpontot lokalizáljuk és az azokat összekötve a kirajzolt négyszög alakzatot távolítjuk el.

A sarokpontok megkeresését végző függvény az egyenes illesztés outputját használja bemenő paraméterként. A bemenő paraméternél megkapott képet itt is először 8 bites, 1 csatornás képpé kell alakítani. Ezt követően szintén kontúrkeresést végeztünk a *cv2.findContours* segítségével.

Az eltárolt kontúrvonalak közül azokat kellett kiválasztanunk, amelyek négy csúcsponttal rendelkeznek és ezt ebben az esetben is az *approx* változó hossza határozza meg.

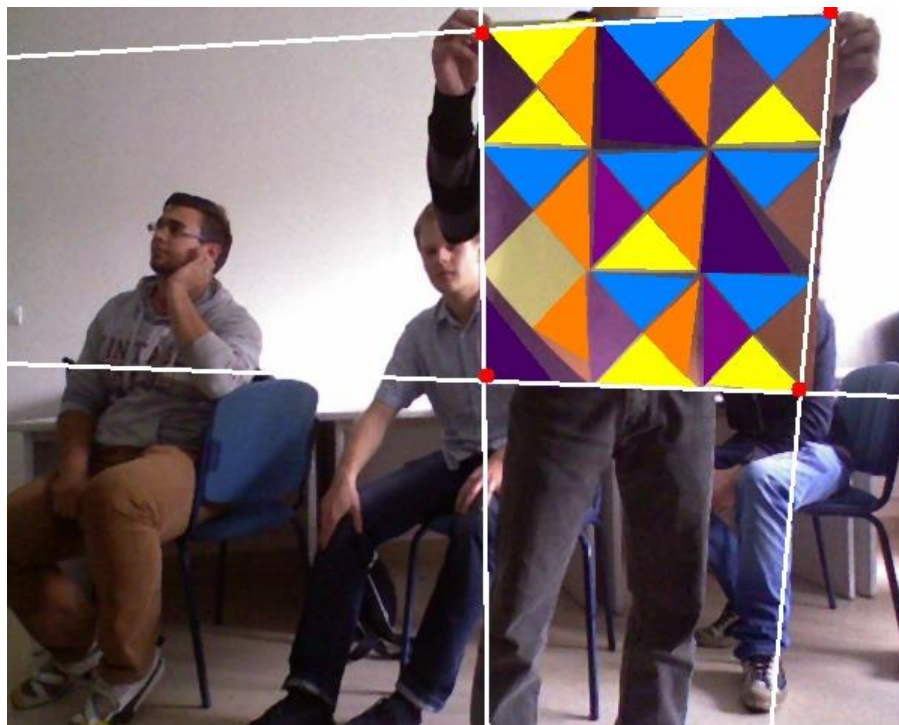
A megtalált négyszögre méretkorlátot is beállítottunk annak érdekében, hogy az esetleges túl kicsi négyszöget ne adja vissza a program lehetséges megoldásként.

A sarokpontok detektálásának megvalósítása:

```
def square_detect(frame3, imgfull):
    gray = cv2.cvtColor(frame3, cv2.COLOR_BGR2GRAY)
    ret, thresh = cv2.threshold(gray,127,255,cv2.THRESH_BINARY)
    contours,hier = cv2.findContours(thresh,cv2.RETR_LIST,cv2.CHAIN_APPROX_SIMPLE)

    for cnt in contours:
        approx = cv2.approxPolyDP(cnt,0.02*cv2.arcLength(cnt,True),True)
        if len(approx) == 4:
            if (cv2.contourArea(approx) > 500 ):
                for i in approx:
                    cv2.circle(imgfull, (i[0][0], i[0][1]), 5, (0,0,255), -1)
    return approx
```

A kódrészletben szereplő `cv2.circle` függvénnyel rajzoltuk ki a sarokpontok helyét. Piros ponttal jelöltem őket. A bejelölt sarkok a 22. és 23. ábrán láthatóak.



22. ábra: A marker sarokpontjainak detektálása



23. ábra: A marker kijelölt sarkai

IV. A FELISMERT ESZTÉTIKUS MARKER ÉRTELMEZÉSE

4.1. A HASZNOS RÉSZ KIVÁGÁSA, ILLETVE NÉZŐPONT KORREKCIÓJA

Az esztétikus marker sarokpontjainak behatárolását követően a következő lépés a robot számára értékes információt tartalmazó marker kivágása. Ennek kivitelezéséhez először is a megtalált sarokpontokat egy magadott sorrend szerint rendeznünk kell és el kell tárolnunk egy tömbben. A tárolás során a következő sorrendben kell a sarkokat megadni: bal felső sarok, jobb felső sarok, jobb alsó sarok, bal alsó sarok.

A sarkok rendezését egy különálló függvényben oldottuk meg. A függvény bemenő paramétere a négyzet csúcspontjait tartalmazó *approx* változó. Az *approx* változó megfelelő formára hozása után összegezzük egyenként a négy pont x és y koordinátáját. A bal felső saroknak lesz a legkisebb az összege, a jobb alsónak pedig a legnagyobb. Ezt követően különbséget számolunk: kivonjuk a sarokpontok y koordinátájából az x koordinátákat. Így meghatározhatjuk, hogy a sarkok közül melyik a jobb felső és melyik a bal alsó, mivel a kapott értékek közül a jobb felső a legkisebb és a bal alsó pedig a legnagyobb.

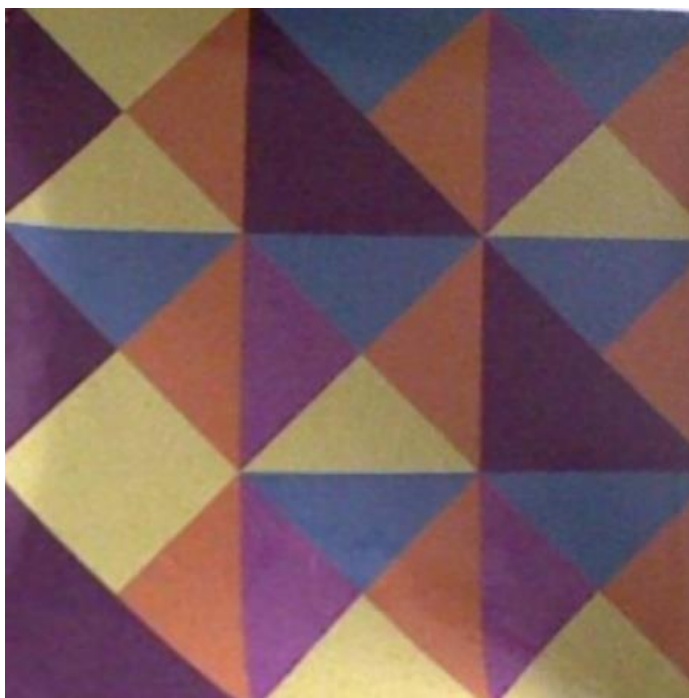
A megvalósítást az alábbi programrészlet mutatja:

```
def corner_sort(approx):  
    approx = approx.reshape((4,2))  
    approxnew = np.zeros((4,2),dtype = np.float32)  
  
    add = approx.sum(1)  
    approxnew[0] = approx[np.argmin(add)]  
    approxnew[2] = approx[np.argmax(add)]  
  
    diff = np.diff(approx,axis = 1)  
  
    approxnew[1] = approx[np.argmin(diff)]  
    approxnew[3] = approx[np.argmax(diff)]  
  
    return approxnew
```

A sarokpontok sorba rendezését követően létrehoztunk egy új képkeretet, amelynek szélessége és magassága is 450 pixel volt. Ebben az új képbe illesztettük be a kivágott markert, úgy, hogy marker sarokpontjain megegyeztek a kép megfelelő sarkaival. A művelet során trapézkorrekciót is kellett használnunk, hogy a kivágott kép megfelelően kerüljön beillesztésre a másik képen. A trapézkorrekció kivitelezéséhez az OpenCV beépített függvényeit használtuk. Ez a kettő név szerint a *cv2.getPerspectiveTransform* és a *cv2.warpPerspective*.

Ezek után a jobb szemléltetés érdekében, a program outputját két ablakban, két képen jelenítettük meg. A korábban mutatott 22. ábra reprezentálja azt a képet, amelyen a marker kijelölésre került.

A 24. ábrán pedig az előző oldalon lévő képről kivágott marker látható:



24. ábra: A kivágott marker

Látható, hogy ebben az esetben nagy pontossággal sikerült kivágni és áthelyezni a markert egy másik képre. Előfordult viszont olyan alkalom, amikor a marker széleit kijelölő egyenesek némileg letérnek a marker vonaláról.

Ilyenkor a kivágást és trapézkorrekciót követően a környezet egy része is láthatóvá válik a képen. De ez az apró pontatlanság nem jelent nagy problémát a későbbiek folyamán. Ezt illusztrálja a 25. ábra:



25. ábra: A nem tökéletesen kivágott marker

4.2. A KIVÁGOTT MARKER FORGATÁSA

A kivágás után meg kell vizsgálnunk, hogy a markert a megfelelő pozícióban kaptuk e meg, vagy szükség van a helyes irányba való forgatásra. Azt tudjuk biztosra, hogy a jobb alsó sarokban kell lennie fixen egy nagyobb sötétlila háromszögnek. Ezt használjuk fel a vizsgálat során.

Első lépésben ki kell szűrni a csak lila háromszögeket. A szűréshez fel kellett használnunk a lila színhez megadott színtartományt a HSV rendszerben. Ehhez segítséget nyújtott a *cv.Scalar* függvény.

```
PURPLE_MIN = cv.Scalar(135, 90, 45)
PURPLE_MAX = cv.Scalar(170, 210, 100)

PURPLE_L_MIN = cv.Scalar(140, 100, 90)
PURPLE_L_MAX = cv.Scalar(160, 140, 155)

PURPLE_RED_MIN = cv.Scalar(0, 90, 100)
PURPLE_RED_MAX = cv.Scalar(8, 120, 128)
```

A PURPLE_RED tartományhoz szintén túlcordulás miatt volt szükségünk. Ennek köszönhetően jobb eredményt kaptunk. A szűrés után ezt a három színt a *cv.AddWeighted()* függvénnyel hozzárendeltük egymáshoz. Így már egy képen volt az összes lila árnyalat.

A következő lépésben, csak a nagy háromszögeket kerestük meg. Az OpenCV *ContourArea* függvényének segítségével sikerült kiszűrni a csak nagy háromszögeket – terület alapján – és a kicsiket pedig eldobni.

A megmaradt nagy háromszögek közül keresnünk kell a sarokban elhelyezkedő háromszöget. Ezt úgy sikerült végrehajtanunk, hogy összevetettük a háromszög pontjainak a koordinátáit a kép sarokpont koordinátaival. Amelyik pont volt legközelebb a kép valamelyik sarokpontjához, az ahhoz tartozó háromszög volt a keresett háromszög.

A következő oldalon, a 26. ábra szemlélteti azt az esetet, amikor a kivágott markert nem a kívánt formában kaptuk meg: referencia pontot jelentő lila háromszög a balsarokban helyezkedik el.



26. ábra: A nem megfelelően detektált marker

A megtalált lila háromszöget következő lépésben meg kell vizsgálni, hogy jó helyen van-e, vagy ha nincs, mennyit kell forgatni a képen.

A legszélső sarokpont indexe alapján döntjük el a háromszög helyét. Ha rossz helyen van meghívjuk a `rotateImage()` függvényünket, melynek átadjuk a képet, és az elforgatás mértékét fokban.

```
if coordindex == 0:  
    imgfull = rotateImage(imgfull,180)  
if coordindex == 1:  
    imgfull = rotateImage(imgfull,-90)  
if coordindex == 3:  
    imgfull = rotateImage(imgfull,90)
```

A kettes indexű sarokpont azért nincs benne, mert ha a pont indexe 2, akkor a háromszög jó helyen van.

A `rotateImage()` függvényben először meghatározzuk a kép középpontját – a pont körül fogja forgatni a képet. Következő lépésben a `cv2.getRotationMatrix2D()` függvény létrehozza az affín mátrixot a kép középpontjából, és a szögből (amennyivel el szeretnénk forgatni a képet). A létrejött mátrix alapján pedig a `cv2.warpAffine()` függvény végrehajtja a forgatást a képen.

```
rotate = cv2.getRotationMatrix2D(image_center,angle,1.0)  
rot_img = cv2.warpAffine(image, rotate, image.shape[0:2],flags=cv2.INTER_LINEAR)
```

Az elvégzett művelet eredményét mutatja 27.ábra.



27. ábra: A sikeresen beforgatott marker

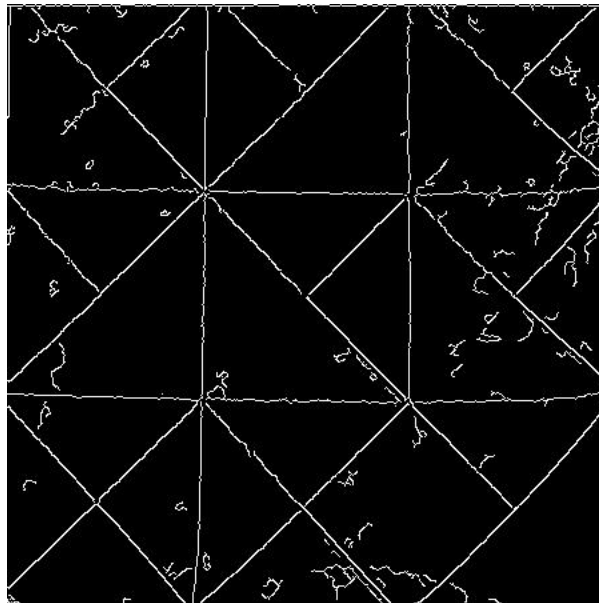
4.3. A MARKER HÁLÓJÁNAK BERAJZOLÁSA

A helyesen kivágott, vagy megfelelően beforgatott marker hálóját kell elkészítenünk a továbbiakban. Ez a háló elválasztja majd az egymástól eltérő színű háromszögeket. A háló készítését egy *cv2.Canny* algoritmussal kezdtük. Ez egy él detektáló függvény az OpenCV függvénykönyvtáron belül.

Ahhoz, hogy az él detektálás működjön szíkonverzióra van szükségünk. A képből egy fekete fehér egy csatornás, 8 bites képet hozunk létre. Ezen a képen fog történni az él keresése. Ezt a képet kapja meg a függvény első paraméternek. Második lesz az első küszöbérték a hiszterézis eljárás számára, a harmadik pedig a második küszöbérték.

```
edges = cv2.Canny(gray,10,35)
```

A használt függvény eredményét mutatja a 28. ábra.



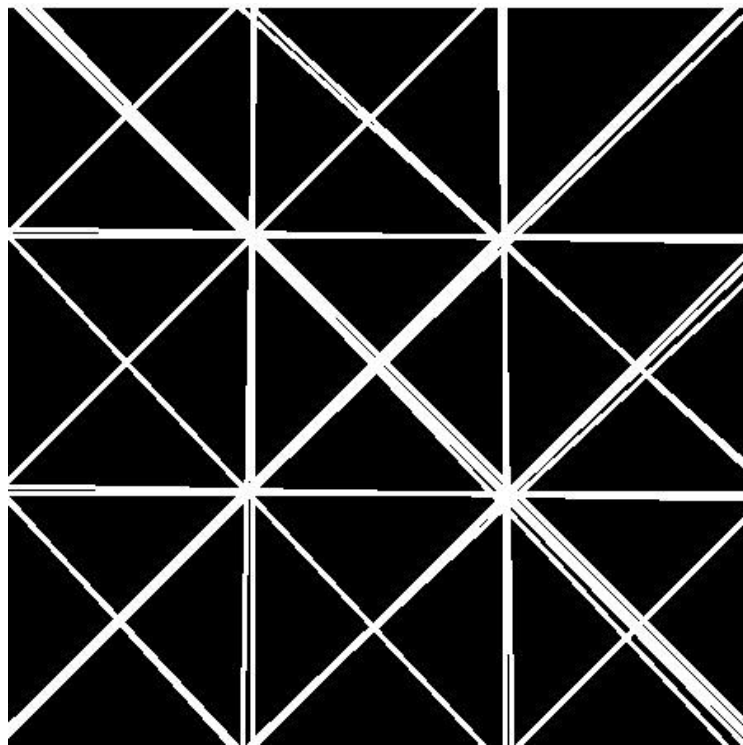
28. ábra: A fekete-fehér kép megtalált élei

Ahogy a képen is látható a marker jelentős számú éleit sikerült detektálnunk. Azonban az is megfigyelhető, hogy számunkra szükségtelen hosszabb-rövidebb görbe vonalakat is megtalált a függvény. A megtalált vonalakat eltávolítjuk egy edges változóban, mivel a továbbiakban szükségünk le rájuk.

Következő lépésben a megtalált éleket használja fel a `cv2.Houghlines` függvény. Segítségével még jobban továbbszűrhetjük az éleket. Eldobja a görbe, túl rövid és használhatatlan vonalakat. Csak az egyeneseket hagyja meg. Eközben elmenti a vonalak hosszát és meredekségét egy tömbbe. Ezen a tömbön végigmegyünk egy for ciklussal. A rho ciklusváltozó veszi fel a vonal hosszát, a theta pedig a meredekségét. Ennek a két adat segítségével ki tudjuk számolni a vonal folytatását, és azok végpontjait. Miután a meghosszabbított vonal végpontjait megtudtuk, már csak annyi a teendőnk, hogy a `cv2.lines` függvényt használjuk és összekötjük a kezdő és végpontokat.

```
for rho, theta in lines[0]:
    a = np.cos(theta)
    b = np.sin(theta)
    x0 = a*rho
    y0 = b*rho
    x1 = int(x0 + 1000*(-b))
    y1 = int(y0 + 1000*(a))
    x2 = int(x0 - 1000*(-b))
    y2 = int(y0 - 1000*(a))
    cv2.line(image, (x1,y1), (x2,y2), (255,255,255), 2)
```

A 29. ábra mutatja a használt algoritmus eredményét, amikor a marker háromszögeinek megtalált éleire egyeneseket illesztettünk. Az illesztett egyenesek nem jelölnek ki minden egyes különálló háromszöget, de meglévők is elegendőek lehetnek a keresett marker felismeréséhez.



29. ábra: A marker hálójá

4.4. A HÁROMSZÖGEKET ALKOTÓ PIXELEK VIZSGÁLATA

Azzal, hogy az előző lépésként a marker hálóját elkészítettük lehetőség adódik arra, hogy hálóval kijelölt háromszögek pixeleit megvizsgáljuk. A berajzolt háló a különböző színű háromszögek határvonalaként szolgál majd a pixelek elemzése során.

Az egyes háromszögek súlypontjainak közelében lévő néhány pixel vizsgálata is elegendő információt biztosít az adott háromszög színének meghatározásához. A pixelek színértékeinek kigyűjtése után többségi szavazással eldöntjük a háromszög tényleges színét. A többségi szavazásra azért van szükség, mert az esteleges képhibák, szemcsék miatt nem biztos, hogy minden pixel színe a háromszög eredeti színével azonos. De a nagy többségnél a helyes színt érzékeljük ezért néhány hibás pixel nincs negatív hatással a végső szín meghatározásánál.

Miután egyesével megvizsgáltuk minden háromszöget és meghatároztuk a hozzá tartozó színt egy mátrixba rendezzük a kapott értékeket, amelyből kinyerhetőek lesznek a háromszögekhez tartozó színek.

V. ÖSSZEFOGLALÁS

A technológia nagymértékű fejlődésének köszönhetően a robotika egyre inkább a figyelem középpontjába kerül. Újabbnál újabb robotok kerülnek megalkotásra és a mérnökök számára kihívást jelentő feladat a gépek önálló helyzet meghatározása és koordinációja. Ennek megvalósítására különböző ötletekkel és módszerekkel állnak elő.

Dolgozatunkban lépésekre bontva mutattuk be egy lehetséges megoldást arra, hogy hogyan végezhető el a Kinect-tel rögzített képen a marker behatárolása, illetve hálójának kijelölése. Kitértünk a markert felépítő, jellegzetes háromszögek felismerésére, a marker oldalainak és sarokpontjainak behatárolására, az információt hordozó marker kivágására és arra, hogy hogyan határozható meg a marker hálója.

Az esztétikus marker lokalizálásához elkészített program segítséget nyújthat a mobil robotok tájékozódásában. Ötleként szolgálhat más vezérlő programok elkészítéséhez. Azzal, hogy a program a robot által rögzített képen képes az esztétikus markert detektálni még nem jelenthetjük ki azt, hogy a marker alapú tájékozódás teljesen megoldott lenne. A munka folytatódik tovább, a végső műveletben kivágott marker további feldolgozáson megy át, hogy a marker információ tartalmát sikeresen kinyerjük. Hiszen a robotok számára a markerben tárolt információ a lényeges.

Emellett az általunk készített program esetlegesen tovább optimalizálható a gyorsabb számítás érdekében. Továbbá átírható C vagy C++ nyelvre, mivel a robotok vezérlésénél főként ezt a két nyelvet alkalmazzák.

A programkódok az alábbi címen érhetőek el:

<http://users.iit.uni-miskolc.hu/~barta7/>

SUMMARY

Thanks to the considerable developments in robotics, robots and intelligent devices are in the focus of attention for a long time now. Newer and newer robots are made and it is a challenging task for the engineers to make robots with self-determination and coordination. They bring out new ideas and possible methods to achieve the challenging tasks.

In this thesis, We introduced a possible solution for the localization of the aesthetic marker by using a simple camera attached to a PC. We present a method for the identification of the characteristic triangles found on the marker itself and for the characterization of the sides and corners of the marker. We also give a possible solution for extracting the marker from the original picture, which then can be used for further processing to gather the information represented by the marker.

The created program for localization of the aesthetic maker may assist in the orientation of mobile robots. Also it could serve as an inspiration for creating other programs. The completed program can be further expanded with extracting useful information from the marker. The information stored in the markers is the important thing for the robot.

In addition, the program made by us can be further optimized for faster calculation and possibly it can be transformed into C or C++ language, or any other language which directly runs on the hardware, because currently it is implemented in Python, which is an interpreted programming language.

Furthermore, this work serves as a starting point for a project, which extracts the contained information from the recognized and pre-processed marker image.



IRODALOMJEGYZÉK

- [1] Haláchy Nóra, Schmidt Dorottya: Esztétikus markerek robot lokalizációhoz, BME TDK konferencia, 2012
- [2] Farkas, Z.V., Korondi, P., Illy, D., Fodor, L.: Aesthetic marker design for home robot localization, IECON 2012 - 38th Annual Conference on IEEE Industrial Electronics Society, 25-28 Oct. 2012, pp. 5510-5515.
- [3] Toronyi Zsuzsanna Szilvia: Mozgáskorlátozottak életkörülményeit segítő robot termékfejlesztése, BME Gépészmérnöki Kar, 2013
- [4] Kovács Róbert: Portás és takarító robot termékfejlesztése, BME Gépészmérnöki Kar, 2013
- [5] Navratil Péter: Kilincsnyitó mechatronikai modul tervezése mobil robotba, BME Gépészmérnöki Kar, 2013
- [6] Abid Rahman: OpenCV Python Tutorials, K-Means Clustering
<http://opencvpython.blogspot.in/2012/12/k-means-clustering-1-basic-understanding.html>
(December, 2013)
- [7] Gulyás Gergely: K-Means, X-Means, FCM klaszterezők http://www.inf.u-szeged.hu/~gulyasg/mi2/pdf/gyak_5_K-Means.pdf (December, 2013)