Gépi látás beadandó

GKNB_INTM038

Révész Zsolt

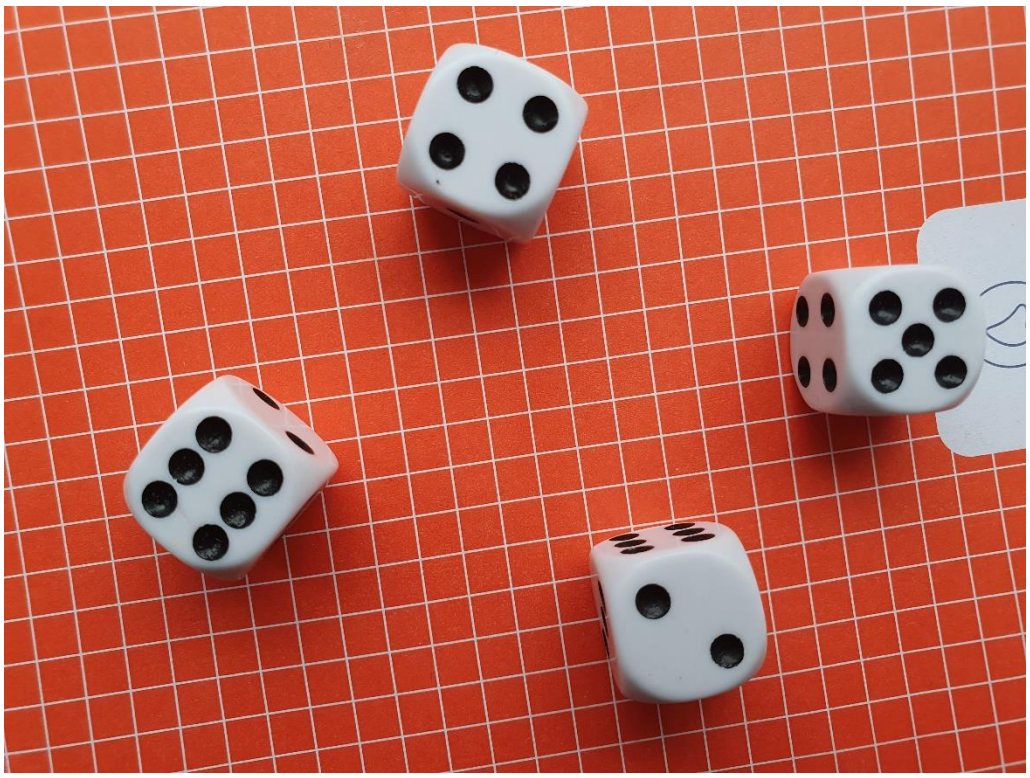
O4MBFN

2022.12.16.

Tartalomjegyzék

1. Bevezetés

Olyan program tervezése és megvalósítása a cél, mely képes egy bemeneti képről a szabályos hat oldalú dobókockák számát és a rajtuk lévő pontok összértékét meghatározni. A program maximum 4 dobókockára legyen optimalizálva, de előnyös, ha többre is működik. A fénykép készítésénél ne kelljen a háttérre figyelni, ezért olyan algoritmus készítése a cél, amit nem zavar össze, ha a dobókockák háttere nem egyforma vagy nem homogén. Bemeneti képre példát az 1. ábra mutat.



1. ábra: Példa bemeneti képre

A 1. ábrán lévő bemeneti képre a felhasználó azt várja el a programtól, hogy a képre írva visszaadja, hogy 4 darab kocka van a képen. Az ezeken található pontok összértéke 17.

2. Elméleti háttér

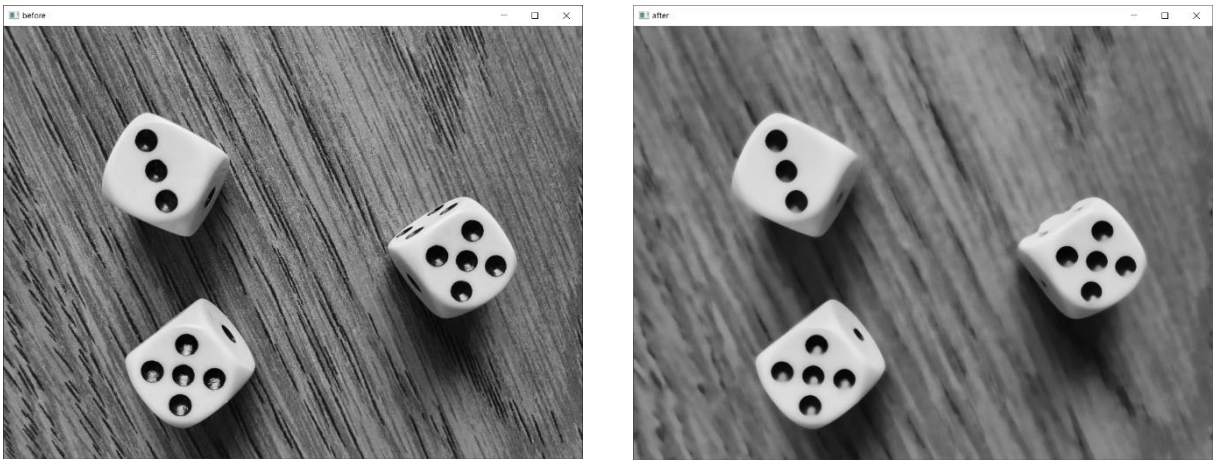
A feladat megvalósításához szükség van a kockák és a rajtuk lévő pontok detektálására. Az alakzatok felismerésénél fontos kritérium, hogy az azok jól elkülönüljenek a háttértől. Ezek felismerése közben nagy esély van a hibák felbukkanására. A hibákat két típusba lehet sorolni. Az egyik, amikor egy alakzatot nem talál meg az algoritmus, a másik eset, amikor egy olyan objektumot sorol be az alakzatok közé, amelyik az emberi szem számára nyilvánvaló, hogy nem tartozik a feladathoz fontos objektumokhoz, de az algoritmus nem tudja. Utóbbira példa, ha a háttéren megjelenő, körre hasonlító alakzatot is dobókockán lévő pontnak érzékel. Ehhez hasonló esetek elkerülése érdekében a megtalált objektumokat valamilyen paraméter alapján meg kell vizsgálni és eldönteni, hogy melyik helyes és melyik hibás detektálás.

2.1. A kép előfeldolgozása

A bemeneti kép bármilyen méretű lehet, ezért vagy olyan algoritmust kell választani, ami erre nem érzékeny vagy valamilyen módon hasonló méretűvé kell alakítani a képeket. Előbbi előnye, hogy a technológia fejlődése nem fogja elavulttá tenni a programot, viszont nehezebb a megfelelő algoritmus megvalósítása és tesztelése. Utóbbi megoldáshoz könnyebb az alakzat felismerő függvény paramétereit beállítani, viszont nagyobb méretű képek kicsinyítésénél előfordulhat olyan mennyiségű információ vesztés, amit az algoritmus már nem képes áthidalni és hibás detektáláshoz vagy még valószínűbb, hogy a detektálás hiányához vezethet. Egy másik megoldás lehet, ha megvizsgálja az algoritmus, mekkora objektumok vannak a képen és azoknak megfelelő arányban alakítja a további algoritmusok paramétereit.

Az átméretezést követően érdemes a bemeneti színes képet szürkeárnyalatossá alakítani, ezzel csökkentve a feladat komplexitását, a vizsgálandó kép méretét, az algoritmus teljesítmény-, és időigényét.

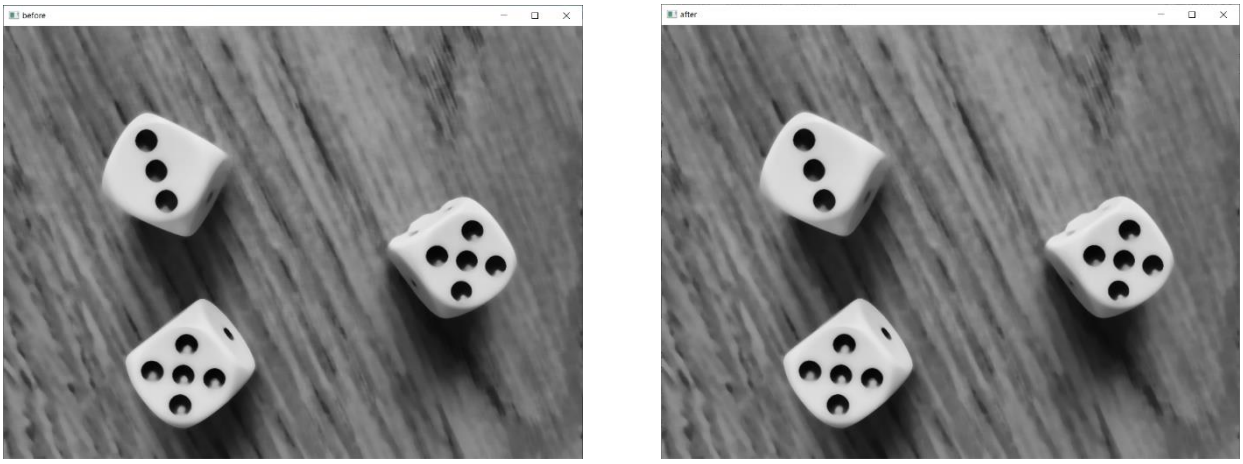
A képen a medián szűrő alkalmazása képes a nem kívánt zajok kiszűrésére. Használatával a háttéren lévő minták elmosódnak, a kockák tetején lévő pontok könnyebben felismerhetővé válnak. A szűrő használata egy másik problémára is megoldást nyújt. Ha több kockát viszonylag közelről fényképezünk, akkor elkerülhetetlen, hogy a kockák oldala is valamilyen mértékben látszódjon a képen, amin szintén vannak pontok. Ezek a pontok a dobott értékbe értelemszerűen nem tartoznak bele, ezért azokat ki kell valamilyen módon szűrni. A medián szűrő ezeket a pontokat is képes elég nagy hatékonysággal elhomályosítani, a háttérbe belemosni.



2. ábra: A medián szűrő használat előtti és utáni állapot

A háttér további homályosítását el lehet érni morfológia használatával. A technológiát bináris képeken lehet alkalmazni. Morfológia használatához kétdimenziós kép esetén létre kell hozni egy kétdimenziós négyzetes kernelt, melyet folyamatosan összehasonlít az algoritmus a kép bizonyos részeivel. A bemeneti képből úgy vesz ki részeket, hogy egy struktúráló elemet végig csúsztat a bemeneti kép minden pontján. Amennyiben a kernel ráillik a kép vizsgált szegletére, akkor az alapján módosítani fogja a kép adott pontját. Ez a módosítás a morfológia két alapműveletének egyikét fogja végrehajtani, az eróziót vagy a dilatációt. Előbbi a nem odaillő magas intenzitású helyeket lecseréli alacsony intenzitásúra, az utóbbi pedig a fordítottját végzi. Ezt a két műveletet gyakran egymás után szokás alkalmazni. Elnevezésük a műveletek sorrendjétől függően morfológiai zárás, illetve nyitás. A nyitást általában objektumok

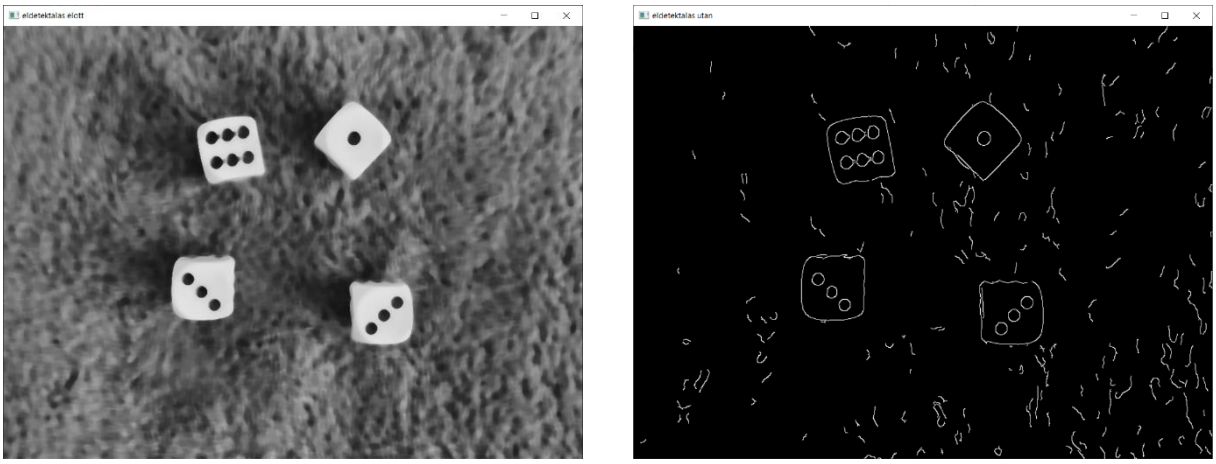
szeparálására használják, míg a zárást objektumok közötti rések betömésére. A dokumentumban tárgyalt programnak a morfológiai zárásra van szüksége. Ha a kernel csak egyeseket tartalmaz, akkor így el lehet érni, hogy a kis méretű, alacsony intenzitású helyeket a morfológia feltöltse, így a zaj mennyisége csökken, ami fontos kritérium a további vizsgáltok eredményeinek pontossága szempontjából.



3. ábra: A morfológiai zárás előtti és utáni állapot

2.2. Pontok felismerése

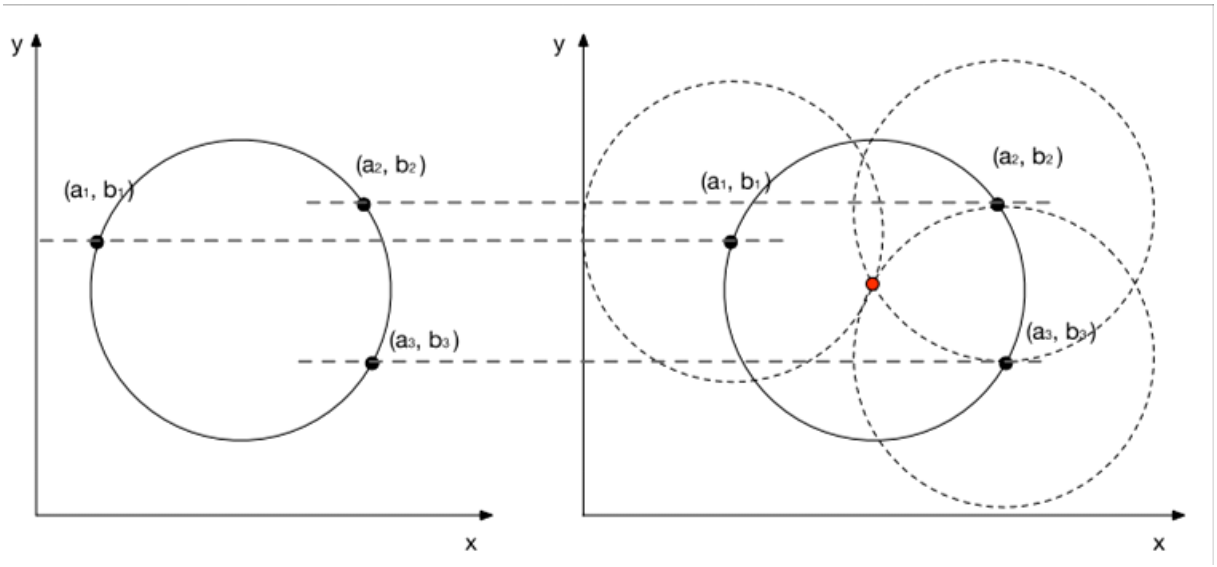
A dobókockák pontjainak felismerését a Hough transzformáció végzi, mely képes körszerű alakzatok felismerésére. A transzformáció a bemeneti képen először egy élfelismerő algoritmust futtat át, ami arra a feltevésre alapszik, hogy azokon a helyeken, ahol nagy mértékű, hirtelen intenzitás változás van, ott feltehetően él van. Így nem egész pontokat kell felismernie a Hough transzformációnak, hanem köröket. A folyamat közben azonban fellép egy probléma. Nem csak az objektumok határán találhatóak élek, hanem előfordulhatnak egy zajos háttéren is. Amíg az algoritmus bemeneti képét nézzük, a háttéren lévő zajok nem valószínű, hogy körre hasonlítanak. Az éldetektálást követően olyan élek is kirajzolódhatnak, amik eddig nem voltak láthatóak és ezeknél nem elhanyagolható esély van arra, hogy azok körre hasonlítanak. Ennek kiküszöbölését a 2.3. bekezdés tárgyalja.



4. ábra: Az éldetektálás után a háttéren lévő zajok körszerű alakzatokat formálhatnak

A Hough transzformáció kimenetként a detektált köröknek három paraméterét adja meg. A 2 dimenziós térben elfoglalt pozíciójának x és y koordinátáját, valamint a kör sugarának méretét. A Hough transzformációnak két verziója van. Egyik esetben ismert méretű köröket kell keresni,

a másik esetben viszont ez nincs megkötve. Előbbi esethez kevesebb erőforrás szükséges, de a képek előfeldolgozására nagyobb hangsúlyt kell fordítani. Jelen program esetén be van állítva egy tartomány, hogy mekkora minimum és maximum sugarú köröket vegyen figyelembe a felismerés során. A bemeneti kép alapján egy másik, úgynevezett Hough-térben végzi a körök keresését. A keresés folyamatának lényege, hogy a bemeneti képen talált élek pontjaira létrehoz a Hough-térben a megfelelő koordinátára egy olyan átmérőjű kört, mely azonos a keresendő kör méretével.

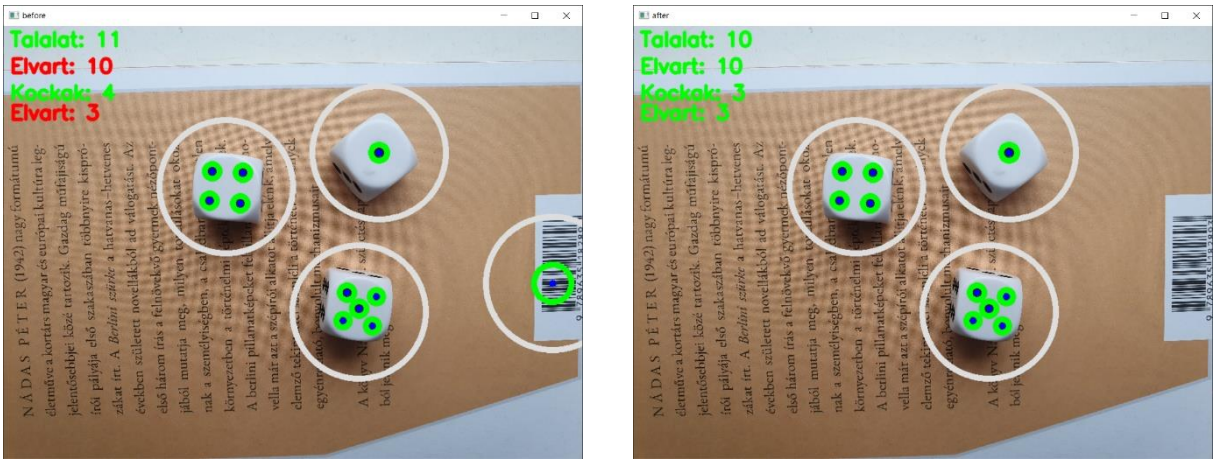


5. ábra: Kör detektálása Hough-tér segítségével

A Hough-térben létrejött körök metszéspontjaiból a Hough transzformáció képes kiszűrni, hol található az bemeneti képen kör. Ezt a 3. ábrán a piros színű pont jelzi.

2.3. Hibásan detektált körök figyelmen kívül hagyása

A gondos előfeldolgozás és jól paraméterezett Hough transzformáció ellenére még mindig van esély hibás kördetektálásokra. Előfordulhatnak olyan hátterek, melyeken kör szerű formák találhatóak vagy olyan zajok, amik képesek megtéveszteni a Hough transzformációt. Ennek a problémának a kiküszöbölésére érdemes a detektált köröket valamilyen módon megvizsgálni és kiszűrni azokat, amik nem illeszkednek a többi kör közé. Az egyik ilyen szempont, amire alapozva hibás detektálást lehet feltételezni, hogy valamelyik kör mérete nagy mértékbe eltér a többitől. Ez normál esetben nem történhet meg, mert a dobókockák és a rajtuk található pontok nem különböznek egymástól és a képek nem extrém szögben készültek róluk. Ilyen esetben érdemes a körök méretét valamilyen módon átlagolni és azokat a köröket figyelmen kívül hagyni, amik ettől az értéktől túl nagy mértékben eltérnek. Ezt az értéket főleg tesztelések útján lehet optimálisan beállítani. Átlag megállapítására többféle módszer van. A számtani átlag egy egyértelmű és egyszerű megoldás lenne. Használatakor azonban az olyan hibás detektálások, amiknek nagy a méretkülönbsége a dobókockákon lévő pontoktól, képesek úgy eltolni az átlag értékét, hogy a helyes detektálások is kiesnének a helyesnek gondolt tartományból. Ilyen esetben célszerű a számtani átlag helyett a medián értéket választani, ugyanis ez a kiugró értékeket nem veszi figyelembe.



6. ábra: A medián értékkel való szűrés előtti és utáni állapota

A 4. számú ábrán látható, hogy a módszer használata előtt, a vonalkódot a Hough transzformáció körként detektálta, mert a vonalak pont úgy helyezkednek el, hogy a transzformáció számára körnek tűnjön, de egy ember számára könnyen felismerhető a hiba. Az ilyen esetekben kicsi az esélye annak, hogy pont akkora legyen a hibásan detektált kör mérete, mint a dobókockán láthatóké, így a medián értékkel való szűrés figyelmen kívül hagyta. A kis eséllyel, de nem nulla valószínűséggel felbukkanó hibák elkerülésére megoldás lehet, ha négyzet szerű formákat keres egy algoritmus és az eredményeit összehasonlítva az előzőekben leírt szerencsétlen szituációval, fel lehet ismerni az ilyen jellegű hibákat is.

2.4. A dobókockák megszámlálása

A dobókockák megszámlálásának az egyik talán legegyértelműbb módja, ha négyzethez hasonló formákat keres a képen egy algoritmus, de ez túl sok hibás detektáláshoz vezetne. Egy másik megközelítés, amit a jelenleg tárgyalt program algoritmusa is követ, hogy használja fel a már megtalált pontokat és azokból vonjon le a dobókockák darabszámára következtetést. A módszer problémája, hogy a hibásan dobókockán lévő pontnak vélt körök a dobókockák darabszámát is könnyedén tévútra képesek vezetni. Ilyen esetek elkerülése miatt is fontos a pontok felismerését követő vizsgálatok, hogy mire a kocka számláló algoritmusig eljutnak a pontok, addigra szűrve legyenek.

A kockák felismerését és koordinátáiknak visszaadását a következő algoritmus végzi. A pontok egymáshoz viszonyított távolságát megvizsgálva következtetni lehet arra, hogy azonos kockán lehetnek-e vagy sem. Ehhez rögzíteni kell egy határértéket, ami lehetőleg akkora legyen, amekkora maximális távolságra lehet egy kockán két pont. Mérési hibák elkerülése érdekében érdemes ennél egy kicsit nagyobb értéket venni. Két pont vizsgálatakor, ha a köztük lévő távolság túllépi ezt a határértéket, akkor azt két külön kockának vesszük. Ha az értéknél kisebb a távolság, akkor meg azonos kocka két külön pontjára lehet következtetni. Ez esetben a két közeli pont helyett az általuk meghatározott szakasznak a felező pontját véve redukálni lehet a vizsgálandó pontok számát. Így folytatva, az algoritmus el fog érni egy olyan pontra, amikor már nincs két olyan tetszőleges pont, amik közti távolság határértéken belül lenne. Ekkor már csak annyi vizsgálható pont lesz, ahány dobókocka és azok pozíciója pedig a kockák közepét fogja mutatni.

3. Megvalósítás terve és kivitelezés

3.1. Alapadatok

A program szerkesztéséhez a Visual Studio Code program volt használva. Az elkészült program Python programozásinyelven íródott és az OpenCV, a numpy és a math külső könyvtárakat használja.

Az OpenCV a Python és a C++ nyelveket támogatja. A Python nyelv mellet azért esett a döntés, mert egyszerűbb szintaxisa és könnyebb használata elősegítette a program egyszerűbb és átláthatóbb megalkotását és tesztelését.

3.2. A program indulása

A program elindításához és használatához kettő Python fájl szükséges, a teszteléshez pedig egy szöveges fájlal, valamint a tesztképeket tartalmazó mappával egészül ki. A main.py elnevezésű programot parancssorból lehet indítani, paraméterként pedig meg lehet adni, hogy milyen üzemmódban induljon el. Ha a parancssori paraméter egy létező fájl helyét mutatja és annak kép kiterjesztése van (png, jpg, jpeg), akkor arra futtatja le a felismerő algoritmust. Ha a fájl nincs a támogatott fájlkiterjesztések között, akkor erre egy egyértelű hibaüzenet hívja fel a felhasználó figyelmét. Ha nincsen paraméter vagy az olyan fájlra mutat, ami nem létezik, akkor a megfelelő hibaüzenet kiírását követően bezárul a program. Ha a -test paraméterrel indul a program, akkor az lefuttatja a tesztmappában található összes képre az algoritmust. A tesztelés folyamatát bővebben a 4. fejezet tárgyalja.

A main.py fájl elindítását követően a program megvizsgálja, hogy milyen parancssori paramétert kapott. Ha az egy létező fájlra mutat, akkor a szükséges környezet beállítását követően meghívja a dobokocka_azonosito.py fájl dicePointDetector elnevezésű függvényét. Az algoritmus bemenetként öt paramétert vár, de az utolsó négy csak a függvény tesztelésekor szükséges. Tehát olyankor, amikor a felhasználó csak egy fájl esetében kíváncsi az algoritmus eredményére, elég a fájl helyét megadni. A main.py más műveletet már nem végez ebben az üzemmódban.

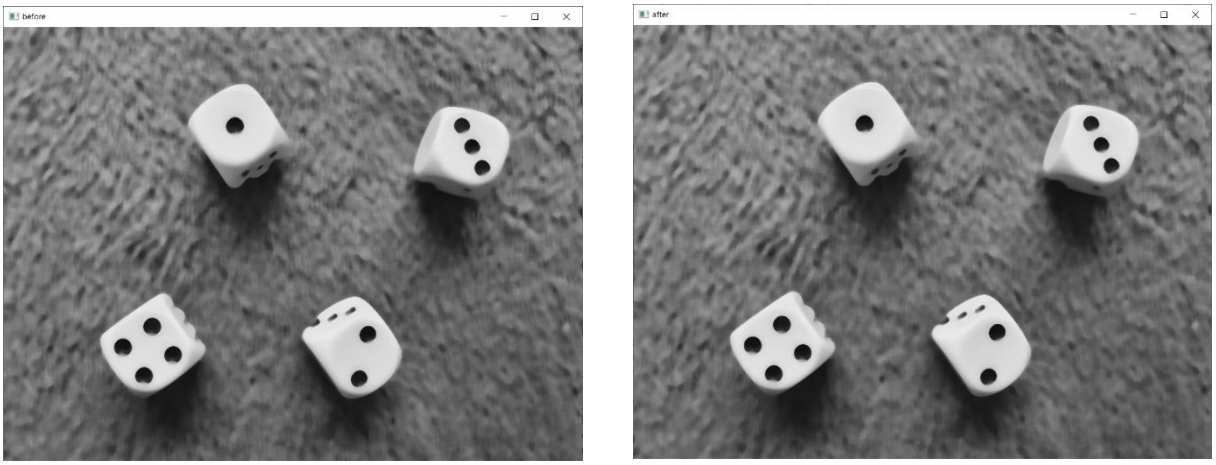
3.3. Előfeldolgozás

A dicePointDetector függvény első lépésként beolvassa a képet és átméretezi. Ennek segítségével a különböző bemeneti képek hasonló méretűre alakíthatóak. Nagy méretű képek esetén így csökken a vizsgálandó és mozgatandó objektum mérete, illetve a vizsgálatok paraméterei pontosabban beállíthatóak.

A következő lépésben a program a három színcsatornás (kék, zöld, piros) színes képet egy színcsatornássá csökkenti. Az így létrejövő szürkeárnyalatos kép használatával a feladat komplexitása tovább csökkenthető azzal, hogy a feladat szempontjából felesleges információk elhagyásra kerülnek.

A képek hátterén és a kockák felületén előfordulhatnak olyan nagy intenzitásváltozások, környezetükhöz viszonyítva kiugró értékek, melyek negatívan befolyásolhatják a különböző alakzat felismerő algoritmusok hatékonyságát. Ezeknek figyelmen kívül hagyásához és a háttér elhomályosításához egy medián szűrőt alkalmaz a program, aminek a paraméterével meg lehet

adni, mekkora méretű környezetet vegyen figyelembe az adott képpont vizsgálata és módosítása során. A technika alkalmazása még ahhoz is hozzájárul, hogy a kockák oldalán lévő pontokat is elhomályosítja, hiszen azok mérete sokkal kisebb és alakjuk szabálytalanabb, mint azoknak a pontoknak, amikre a kockák tetején közvetlenül rálát a fényképezőgép. A másik szerencsés tény, hogy a dobókockák fehér alapon fekete színűek. Ezt a tulajdonságot kihasználva az oldalsó kisebb méretű pontokat egy kellően nagy méretű medián szűrő el tudja tüntetni.



7. ábra: A morfológiai zárás előtti és utáni állapot

A 6. ábrán látható, hogy a morfológiai zárás alkalmazása előtt és után a képeken a változás eredménye nem túl szembetűnő, maximum annyi látható szabad szemmel, hogy a háttér minimális szinten simítva lett. Nagy számú minta esetén a tesztelés azonban bebizonyította, hogy szűkség van a használatára, mert így bizonyos esetekben a hibás detektálásokat képes elkerülni az algoritmus.

3.4. Pontok detektálása

A dobókocka pontjainak detektálása az OpenCV HoughCircles függvényével történik, ami a Hough transzformációt veszi alapul. Paramétereivel a függvény működését az adott problémára lehet szabni. Első paraméterként egy szürkeárnyalatos képet vár. A második paraméterével a detektálásra használt metódust lehet kiválasztani. A következő a Hough-tér eredeti képhez viszonyított arányát állítja. A programban ez 1 értékre van állítva, tehát a méretük megegyezik. Ez után a detektálható körök közötti minimum távolságot állítja be. Ennek a beállítására nem kell nagy figyelmet fordítani a jelenlegi probléma megoldás közben, más esetekben viszont jól jöhet. Segítségével különböző zajokat is át tud hidalni az algoritmus, például amikor két körszerű zaj kontúrja egymáshoz ér. A következő két paraméter a detektáló metódus paramétereit állítja. Ezeknek a finomhangolása sok tesztelést igényel és gyakran történik olyan, hogy az egyik képre tökéletesen beállított paraméter a másik képen semmit sem talál meg. Az utolsó két paraméter a detektálható körök minimum és maximum sugarát állítják be. Ezeknek a pontos beállításához volt szükség arra, hogy a különböző méretű bemeneti képeket hasonló nagyságúra méretezzünk.

A HoughCircles kimenetként egy olyan numpy listát ad, amiben a detektált körök kétdimenziós térben elfoglalat pozíciója és a hozzá tartozó sugár mérete van megadva.

```
rows = imgMorph.shape[0]
circles = cv2.HoughCircles(imgMorph, cv2.HOUGH_GRADIENT, 1, rows / 100,
                           param1=240, param2=23,
                           minRadius=1, maxRadius=50)
```

8. ábra: A HoughCircles függvény hívása a szükséges paraméterekkel

3.5. Hibásan detektált pontok kiszűrése és képre írás

A HoughCircles függvény paraméterei szándékosan egy kicsit lazára vannak állítva, hogy inkább találjon meg olyan köröket is, amik nem a kockán vannak, minthogy ne találja meg valamelyiket. Ugyanis az előbbit viszonylag könnyen ki lehet szűrni, de az utóbbival később már nem lehet mit kezdeni. A hibásan detektált körök figyelmen kívül hagyására a 2.3. bekezdéseben tárgyalt megoldás van használva. A megtalált körök listájából készítve van egy segédlista. Abban érték szerint rendezve vannak a sugarak és a lista középső értékét véve megvan a sugarak medián értéke. Ennek használatával törölve van az eredeti listában minden olyan kör, aminek a sugarának a mérete kisebb, mint a sugár medián - 5 vagy nagyobb, mint a sugár medián + 10. Így törlésre kerültek a háttéren lévő körök nagyrésze, illetve a kockákon lévő, átlagtól különböző méretű körök.

```
i=0
while i <= len(circles2[0])-1:
    if (circles2[0][i][2] > (radiusMedian+10) or circles2[0][i][2] < (radiusMedian-5)):
        toDelete = circles2[0][i]
        circles2[0].remove(toDelete)
    i += 1
```

9. ábra: A körök sugarainak mediánjától túl nagy mértékben eltérő sugarú körök törlése

Miután meglettek a dobókockákon lévő pontok, azokat az algoritmus a kimeneti képen bejelöli. A megtalált kör sugarával megegyező méretben egy zöld körrel megjelöli és a középpontjukat pedig egy kék ponttal jelzi. Ezek hozzájárulnak ahhoz, hogy a felhasználó tudja, hogy pontosan melyik pontokat számolta bele a végösszegbe az algoritmus, így egy esetleges hibára is azonnal rá lehet jönni.

3.6. Dobókockák megszámlálása

A dobókockák megszámlálása a 2.4. bekezdés elméleti hátterén alapul. A lényege, hogy megvizsgálja minden pont távolságát a többi ponthoz viszonyítva. Ha ez az érték két pont között kisebb, mint egy dobókocka mérete, akkor feltételezhetően azonos kockán vannak. Ilyen esetben a két pont helyett a továbbiakban csak a közéjük húzható szakasz felezőpontját veszi fegyelembe az algoritmus, a másik kettőt törli. Miután kellő mennyiségben lefutott a ciklusmag, a folyamat végén annyi pont fog maradni a listában, ahány dobókocka van a képen. A módszer hátránya, hogy kell egy segéd tömb, amin a műveleteket végre lehet hajtani és elemeket lehet belőle tetszőlegesen törölni és hozzáadni. A ciklus lefutás kellő mennyisége a következő szerint lett beállítva. Mivel minden pontot minden ponttal össze kell hasonlítani, így ezekhez két olyan ciklus kell, ami végigmegy az összes detektált ponton. (Valójában a külső ciklusnak elég az elemek száma mínusz egyig menni, mivel a belső ciklus úgyis meg fogja azt vizsgálni.) Mivel ezek összehasonlításakor a lista végére hozzáadásra kerülnek új pontok (a felezőpontok), ezért

azokat újra és újra meg kell vizsgálni. Hogy nehogy kimaradjon egy vizsgálat, ezt a két ciklust egy harmadikba kell csomagolni, ami hatszor fog lefutni, mivel egy dobókockán maximum hat pont lehet.

```
for z in range(0,6):
    for i in range(0, len(circles2[0])-1):
        for j in range(0, len(circles2[0])):
```

10. ábra: A dobókockák megszámlálásához szükséges ciklusok ciklus vezérlő feltétele

Három egymásba ágyazott ciklus sosem szerencsés, de mivel tudjuk, hogy ezek lefutásának száma rögzítve van egy felső határhoz, ezért nagy teljesítmény problémát nem tud okozni. A felső határ az 1. fejezetben, a probléma leírásában van rögzítve. Maximum négy szabványos dobókockára kell optimalizálni a programot. Tehát a maximum felismerhető pontok száma hatszor négy. A belső ciklus lefutása $O(3312)$. Ezt a futási számot csak abban az esetben éri el, ha mind a négy kockán hatos van dobva.

```
for z in range(0,6):
    for i in range(0, len(circles2[0])-1):
        for j in range(0, len(circles2[0])):
            if  i != j and i <= len(circles2[0])-1 and j <= len(circles2[0])-1 :
                if math.sqrt(float( ( (float(circles2[0][i][0])-float(circles2[0][j][0]))**2 + (float(circles2[0][i][1])-float(circles2[0][j][1]))**2 ))) <= 85:
                    x=(float(circles2[0][i][0])+float(circles2[0][j][0]))/2
                    y=(float(circles2[0][i][1])+float(circles2[0][j][1]))/2

                    if i<j:
                        toDelete=circles2[0][i]
                        circles2[0].remove(toDelete)
                        toDelete=circles2[0][j-1]
                        circles2[0].remove(toDelete)
                    else:
                        toDelete=circles2[0][i]
                        circles2[0].remove(toDelete)
                        toDelete=circles2[0][j]
                        circles2[0].remove(toDelete)
                    circles2[0].append([x,y, 15])


dices=len(circles2[0])
```

11. ábra: A dobókockák megszámlálására használt algoritmus

Az 11. ábrán látható az algoritmus implementálása. A belső ciklusmag első if függvénye először megvizsgálja, hogy az összehasonlítandó két pont különböző-e. Ha ez nem lenne, akkor egy kivételével az összes pontot törölné (hiszen minden pont pozíciója közel van a saját pozíciójához) és a végén hibásan az jönne ki, hogy egy dobókocka van a képen. A következő feltétel azért szükséges, hogy nehogy túlindexeljen az algoritmus. Normál működés közben a pontok száma folyton csökken a listában, hiszen két közeli pont megtalálása után csak a felezőpontjukat visszük tovább. A legbelső ciklus midig friss adatokkal rendelkezik a lista méretét illetően, hiszen az végzi a törléseket is, de a középső ciklus, mivel sokkal ritkábban fut le a legbelsőhöz képest, így nem ismeri mindig a lefrissebb állapotot.

A második if függvény végzi a pontok közötti távolság kiszámolását, illetve ennek az értéknek az összehasonlítását a 85 értékkel. Ez a 85 az az érték, ami körülbelül meghatározza, hogy maximum mekkora távolságra lehet egy dobókocka két tetszőleges pontja.

A pontok törlése és az új pont hozzáadása tetszőleges sorrendben lehet. Törléskor egy dologra kell odafigyelni. A középső és a belső ciklus melyik iterátora mutat kisebb indexű elemre, mivel mind a két ciklus ugyanannak a tömbnek az elemein megy végig. Ha először a nagyobb indexű elem kerül törlésre, az nem okoz indexelési problémákat, viszont fordított esetben, egy kisebb

indexű elem törlése a nagyobb indexű elem pozícióját is befolyásolja. Ezért kellett a törlést kétféle verzióban megvalósítani.

A dobókockák számának megállapítását követően a dicePointDetector függvény már csak a megtalált pontok és kockák számát írja rá a kép bal felső sarkára és az elkészült képet megjeleníti a következő billentyűleütésig.

4. Tesztelés

A program tesztelésére létre van hozva egy tesztképeket tartalmazó mappa, illetve ezekkel összhangban egy szöveges fájl. Ebben a fájlban a tesztképek nevei, a képeken lévő dobókockák pontjainak az összértéke, illetve a dobókockák számai vannak összegyűjtve. A tesztelés folyamán a program ezekkel az értékekkel hasonlítja össze az algoritmus által megtalált pontok és dobókockák számát, amellyel következtetni lehet az algoritmus pontosságára.

A tesztelés elindításához a main.py programot kell a -test paraméterrel elindítani. Ilyenkor a main.py fájlban három, funkciójukban elkülöníthető rész megy végbe. A szöveges fájl beolvasása, a dobókockát felismerő függvény futtatása képenként, majd az eredmények összevetése az elvárt eredménnyel.

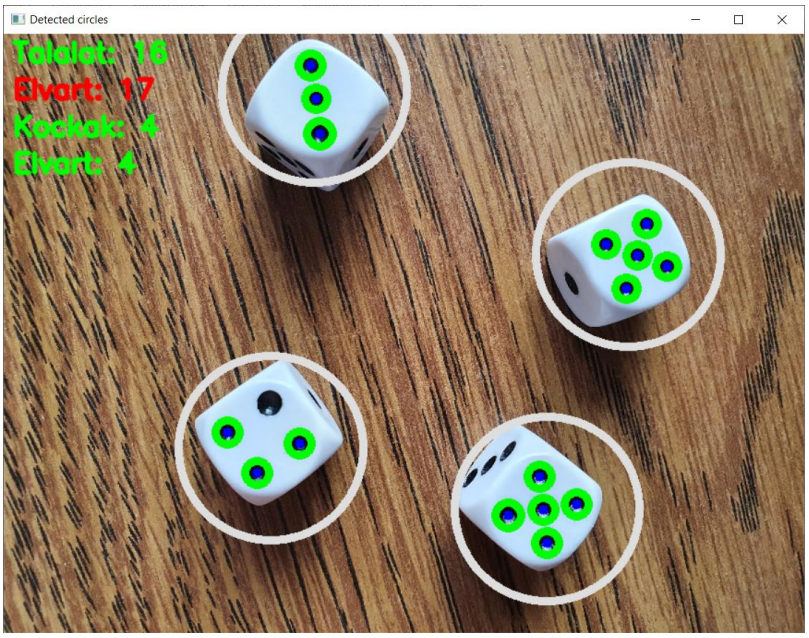
Miután a program megbizonyosodott a szöveges fájl létezéséről, abból elkezdi beolvasni és eltárolni az adatokat. Ilyen adat a kép fájlok nevei és a képen található pontok és dobókockák darabszáma. Az adatokat az előre meghatározott formázás alapját gyűjti be. A követelmény, hogy az adatok vesszővel legyenek elválasztva egymástól és az agymáshoz tartozó adatok azonos sorban legyenek, a különbözőek pedig különbözőben. Az üres sorokra nem érzékeny a beolvasó függvény. Ilyen esetben automatikusan a következő sorra ugrik.

```
kepek/1.jpg,16,4
kepek/2.jpg,16,4
kepek/3.jpg,11,4
kepek/4.jpg,11,4
kepek/5.jpg,10,4
kepek/6.jpg,10,4
kepek/7.jpg,18,4
kepek/8.jpg,18,4
kepek/9.jpg,15,4
```

12. ábra: A teszteléshez szükséges szöveges fájl felépítése

A beolvasott adatok felhasználásával a main.py meghívja a dobokocka_azonosito.py fájl dicePointDetector elnevezésű függvényét. Az algoritmus bemenetként öt paramétert vár, amelyek rendre a következők. A feldolgozandó kép neve és mappákban elfoglalt pozíciója, az elvárt pontok darabszáma, a találatra elvárt kockák darabszáma. Az utolsó kettő argumentum pedig két lista, melybe az algoritmus hozzáfűzi a megtalált pontok és dobókockák darabszámát. Ezeket a main.py az eredmények kiértékelésénél és a hibák számának meghatározásánál fogja használni.

A pont és a dobókocka felismerő algoritmus működése megegyezik azzal az esettel, amikor nem teszt üzemmódban indul a program, egy eltérést leszámítva. Amikor az algoritmus lefut, az eredményeket mindkét üzemmódban a kép bal felső sarkába írja. Teszt üzemmódban ez még kiegészül azokkal az adattokkal, amik valójában a képen található adatok. Erre példát a 13. számú ábra mutat. Tehát a képen lévő szövegek rendre az alábbiak: detektált pontok száma, dobókockán lévő pontok száma, detektált dobókockák száma, dobókockák száma. Tehát minden második adat a szövegfájlból van kiszedve és a felhasználó összehasonlíthatja a kapott eredménnyel.



13. ábra: A dobott érték és a dobókockák száma a képre írva

A 13. ábra egy olyan esetet mutat, amikor az algoritmus nem tudja az összes pontot megtalálni a képen. Ilyenkor a kiírás színe pirosra vált, hogy ezzel felhívja a tesztelő figyelmét a hibára. Mivel az összes megtalált pontot jól láthatóan bejelöli az algoritmus zöld színnel, így az emberi szem számára hamar nyilvánvalóvá válik, hogy melyik pont megtalálása nem sikerült. A kód felépítésének köszönhetően a hiba forrása két helyre szűkíthető le. Vagy a képek előfeldolgozása közben történt nemvárt esemény, esetleg nem lettek eléggé kiszűrve a zajok, vagy a Hough transzformáció paraméterei nem lettek kellő odafigyeléssel megválasztva, aminek köszönhetően az algoritmus elsiklott ennek a pontnak a detektálása felett. Utóbbi lehetséges problémára a megoldás megtalálásának módja nem csak egyszerű paraméter értékek finomítása, mivel, ha egy értékkel is megváltoznak azok, akkor már a többi képnél léphetnek fel hibák. Ebből kifolyólag a probléma forrását az előfeldolgozásnál kell keresni vagy a kikerülését ott kell elkezdeni.

Amikor a dobókocka felismerő függvény véget ér és a program visszalép a main.py fájlba, akkor a kapott eredmények és az elvárt eredmények alapján összehasonlítja az adatokat és tömören összegzi azok eredményét egy konzol üzenet formájában. A dobókockák és a pontok felismerésére vonatkozóan két-két adatot ír ki. Hány képnél található eltérés az elvárt és megtalált eredmények között. Illetve képek darabszámától függetlenül hány detektálás volt hibás.

```
Elteresek szama megtalalt pontok es az elvart eredmeny kozott (ennyi kepnel talalhato hiba): 9
Elteresek szama megtalalt pontok es az elvart eredmeny kozott (osszes hibaszam): 11
Elteresek szama elvart es megtalalat kockak kozott (ennyi kepnel talalhato hiba): 0
Elteresek szama elvart es megtalalat kockak kozott (osszes hibaszam): 0
```

14. ábra: A tesztelés után kiírt konzol üzenet

Ha a fejlesztő nem rendszer szintű tesztelést akar, hanem csak a dicePointDetector függvényt szeretné tesztelni, akkor a dobokocka_azonosito.py fájl alsó sorában található kikommentezett sor segítségével könnyedén kipróbálhatja a működését. Ilyenkor manuálisan kell megadni, hogy hány pont és hány dobókocka van a képe, különben hibásnak fogja jelezni a detektálást. Ez nem befolyásolja annak helyességét, de a felhasználó számára aggodalomra adhat okot.

```
# A függvény teszteléséhez a következő három sorra van szükség

detectedPoints = []
detectedDices = []
dicePointDetector("kepek/61.jpg", "13","3", detectedPoints, detectedDices)
```

15. ábra: A függvény teszteléséhez szükséges programkód

A tesztelés több, mint 100 képet vett figyelembe és a következő következtetéseket lehet belőlük leszűrni. Ha a Hough transzformáció eredményét utólagos ellenőrzés és szűrés nélkül továbbítjuk a dobókockák megszámlálásáért felelős programrésznek, akkor a dobókockák száma nagy hibamennyiséget eredményezhet. Ennek kiküszöbölése érdekében a körök sugarának medián értékével való szűrés a tesztképek mindegyikén képes volt a hibásan körnek detektált alakzatok kiszűrését elvégezni. Ez azt is jelenti, hogy a tesztképeken való futtatás közben egy képen sincs olyan hibás kördetektálás, ami valójában nem kör vagy nem a dobókockák valamelyikén van.

A detektálás másik lehetséges hibájának a tökéletes elkerülése már nem mondható el a programról. Olyan esetek előfordultak, hogy kockán lévő kört a Hough transzformáció nem tekintett körnek. Ilyen esetre mutatott példát a 13. ábra. Összességében a tesztképek futtatása utána az alábbi következtetések vonhatók le. 101 kép közül 9 képen található detektálás hiányából eredő hiba és összesen pedig, az 1012 képeken lévő pontok közül 11 pontot nem talált meg az algoritmus.

5. Felhasználó leírás

A dobókocka számláló alkalmazás futtatására olyan parancssor szükséges, mely képes Python fájlok futtatására, valamint rá van telepítve az OpenCV nevű könyvtár.

Windows PowerShell-en való futtatáshoz az alábbi módon lehet felkészíteni a futtató környezetet:

Először telepítsük a rendszerre a Python programozásinyelv bármelyik verzióját. Érdemes a legújabbat. Ehhez a Python hivatalos weboldalán található útmutató. (https://www.python.org/downloads/windows/) Ezt követően érdemes meggyőződni a telepítés sikerességéről úgy, hogy kiiratjuk a telepített verzió számát. Ehhez a Windows PowerShell-be a következő kódot kell beírni és futtatni: python --version

Ezután az OpenCV könyvtár telepítése következik, melyhez útmutató az alábbi oldalon érhető el: https://pypi.org/project/opencv-python/

Röviden a pip nevű programot kell telepíteni, majd frissíteni. Ez egy Python nyelvhez írt könyvtárak kezelésére és telepítésére szolgáló program. Segítségével az OpenCV könyvtár Python programnyelvvel kompatibilis verzióját kell telepíteni.

Ezzel a futtató környezet fel lett készítve a program futtatására.

Windows 10 és 11 operációs rendszereken lehetőség van Linux-os környezet létrehozására a Windows Subsystem for Linux programmal, azonban ezzel nem lehet futtatni az alkalmazást, mert a grafikus részek megjelenítésére még nem alkalmas.

A dobókocka számláló alkalmazást az alábbi oldalról lehet tömörítve letölteni:

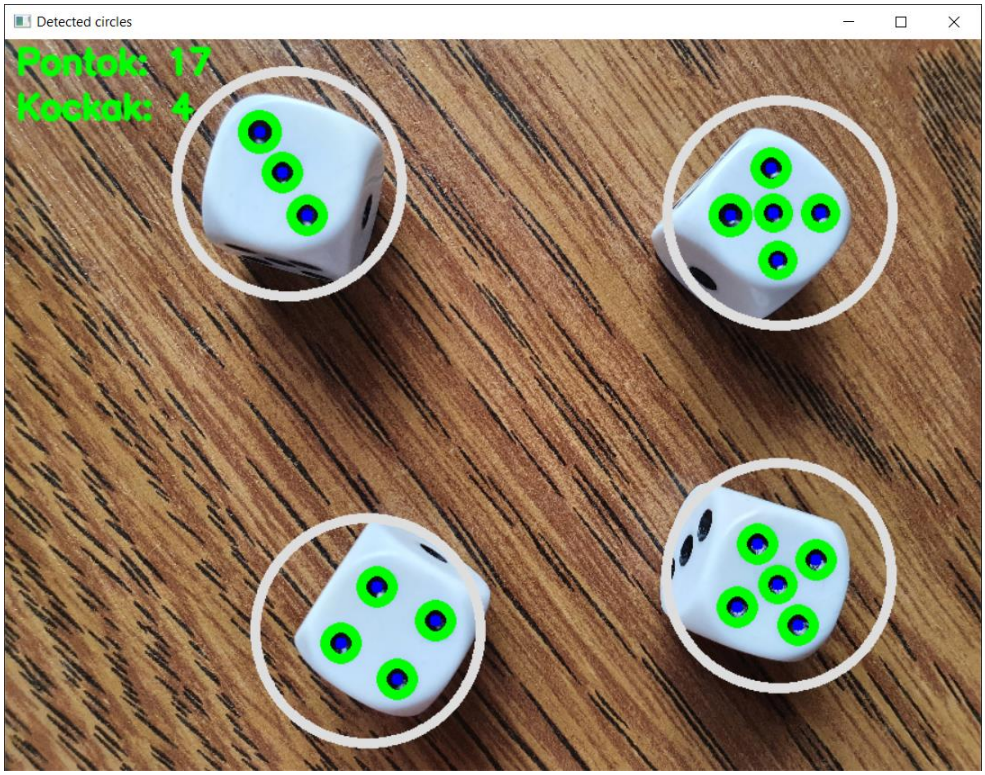
https://github.com/rzsolt29/Gepi_latas

A tömörített állomány kicsomagolását követően, a Windows PowerShell segítségével a kicsomagolás helyére navigálunk.

A program indítására az alábbi kód képes:

```
python .\main.py {a kép neve}
```

A felismerendő képet előzetesen be kell tenni a programot tartalmazó mappába. A kép nevét meg kell adni parancssori argumentumként.



16. ábra: A program futását követően hasonló felépítésű képet kell látni

A 16. ábra által mutatott eredmény a tesztképek használatával a következő parancs futtatását követően lehet elérni.

```
python .\main.py kepek/58.jpg
```

Amennyiben a program tesztelése a cél, akkor a kepek mappában lévő tesztképek alkalmasak erre. A teszt futtatását a már említett parancs segítségével lehet elindítani:

```
python .\main.py {a kép neve}
```

A kép nevének meg lehet adni bármelyik kepek mappában szereplő fájl nevét, elérési ótvonallal együtt. Például:

```
python .\main.py kepek/1.jpg
```

Ha mindegyik képpen le szeretnénk futtatni a programot, akkor a képeket nem mutatja meg minden egyes alkalommal, de kiírja a teszt végén, hogy a program mennyi hibát vétett a tesztképek detektálása közben. Ehhez a parancs:

```
python .\main.py -test
```

Irodalomjegyzék

https://vik.wiki/images/f/fc/IKM_2014_morfologia.pdf

https://docs.opencv.org/4.x/dd/d1a/group__imgproc__feature.html#ga47849c3be0d0406ad3ca45db65a25d2d

Ábrajegyzék