

# SZAKDOLGOZAT

**Snooker billiardjáték elemzése**

**Lengyel Márk**

Mérnökinformatikus BSc  
Mérnökinformatikus Szakirány

**2022**

# Tartalomjegyzék

<b>1. Bevezetés</b>	<b>2</b>
1.1. A projekt célja . . . . .	2
1.2. A Snooker játék . . . . .	2
1.2.1. Általánosságban a snooker játékról . . . . .	2
1.2.2. Eszközök . . . . .	2
1.2.3. Pontszerzés . . . . .	3
1.3. Az OpenCV képfeldolgozási könyvtár . . . . .	3
1.4. Tensorflow a neurális hálózatokhoz . . . . .	3
<b>2. A golyók pozíciójának felismerése</b>	<b>4</b>
2.1. A probléma ismertetése . . . . .	4
2.2. Az asztal felismerése . . . . .	4
2.3. A golyók azonosítása . . . . .	7
2.3.1. Azonosítás mintaillesztéssel . . . . .	7
2.3.2. Azonosítás körkeresés és mintaillesztéssel . . . . .	9
2.3.3. Azonosítás körkeresés és gépi tanulás segítségével . . . . .	9
<b>3. Analizálás</b>	<b>12</b>
<b>4. Megvalósítás</b>	<b>13</b>
4.1. Alkalmazott módszerek . . . . .	13
4.2. A golyók pozíciója . . . . .	13
4.2.1. A szükséges könyvtárak importálása . . . . .	13
4.2.2. Az asztal kontúrjának megkeresése . . . . .	13
4.2.3. Az asztal kivágása és torzítása . . . . .	17
4.2.4. Körkeresés és azonosítás . . . . .	19
<b>Ábrák jegyzéke</b>	<b>20</b>
<b>Irodalomjegyzék</b>	<b>22</b>

# 1. fejezet

## Bevezetés

### 1.1. A projekt célja

A dokumentumban szereplő projekt célja snooker billiardjáték felismerése, és analizálása fénykép/képernyőfelvétel alapján. A felismerés az asztalon elhelyezkedő különböző színű golyók pozíciójának meghatározásából áll. A felismerés megvalósításához különféle képfeldolgozási eszközöket, neurális hálózat alapú kép osztályozást használok, amelyeket **Python** programozási nyelven valósítok meg főként **OpenCV** és **Tensorflow** könyvtárak használatával. A bevezetés későbbi részeiben ismertetem a snooker billiardjátékot, továbbá a projekt elkészítéséhez használt programozási nyelvet és fejlesztési könyvtárakat.

### 1.2. A Snooker játék

#### 1.2.1. Általánosságban a snooker játékról

A snooker a billiardjátékok egy bizonyos fajtája, amelyet egy zöld színű posztóval bevont asztalon játszanak, amelynek mérete általában 12 x 6 láb (365,8 cm x 182,9 cm)[24]. Az asztal négy sarkában és a két hosszabb oldal felénél ún. **zsebek** helyezkednek el. A játék célja a színes golyók belökése a fehér golyó segítségével a fent említett zsebekbe.

#### 1.2.2. Eszközök

A játékot két fél játssza egymás ellen. A felek a lökéseiket egy hosszúkás, fából készült eszköz segítségével végzik. Ezt az eszközt **dákónak** nevezik. A dákó vége, amellyel a golyó elütésre kerül, bőrrel van bevonva, amely a golyóval való kapcsolatot javítja. A dákón kívül a golyó elütéséhez a játékosok használhatnak segédeszközöket.

A tartozékok részei továbbá a már eddig is szóba került golyók. A játékhoz **22 db színes golyó** tartozik amelyek átmérője 52,5 mm.[24]

Az egyes golyók különböző pontértékekkel rendelkeznek:[24]

- 1 db Fehér
- 15 db Piros - 1 pont
- 1 db Sárga - 2 pont
- 1 db Zöld - 3 pont
- 1 db Barna - 4 pont
- 1 db Kék - 5 pont
- 1 db Rózsaszín - 6 pont
- 1 db Fekete - 7 pont

A fehér golyó nem rendelkezik pontértékkal, mivel a játékosok ezt a golyót használják lökéseikhez.

A játék egy menetét **frémnek** nevezik, amely a kezdő lökéstől a fekete golyó elhelyezéséig tart.[24]



**1.1. ábra.** A golyók kezdeti pozíciója.

### 1.2.3. Pontszerzés

A játékosok a pontjaikat a golyók bizonyos sorrendben való zsebbe helyezésével szerzik. Az egymás után hiba nélkül szerzett pontok összegét **törésnek** nevezzük. Egy játékos például szerzhet 9 pontos törést a következő golyók egymás utáni elhelyezésével *piros -> zöld -> piros -> barna*.[18]

Egy játékos büntetőpontokat kap hibák elkövetése esetén. Hibát elkövetni lehet például a fehér golyó zsebbe helyezésével, nem megfelelő színű golyó elütésével. Az elkövetett hibáért minimum 4, maximum 7 pontlevonás jár, attól függően, hogy milyen színű golyók mozdulnak a hiba elkövetésekor (pl.: Ha a cél a piros golyó lelökése, de a lövő a feketét találja el, akkor 7 hibapont jár). A hibát elkövető játékos a törésének pontjait megkapja a hibát elkövetett lövés közben elhelyezett golyók pontjainak kivételével.[24]

## 1.3. Az OpenCV képfeldolgozási könyvtár

Az OpenCV egy főként **valós idejű képfeldolgozáshoz** használt programozási függvénykönyvtár. A könyvtár többféle programozási nyelvekhez készült implementációval létezik (pl.: C++, Python, Java stb. )[7], ezek közül ebben a projektben Python programozási nyelven keresztül fogom használni.

A könyvtárból használt függvények segítségével kerülnek megnyitásra a képek, továbbá a képeken való műveletek (pl.: szürkeárnyalatolás, élkeresés) is a könyvtár segítségével lesznek végrehajtva. A későbbiekben lesz szó a könyvtárból használt függvényekről, azok működéséről nagyobb részletességgel.

## 1.4. Tensorflow a neurális hálózatokhoz

A Tensorflow az OpenCV -hez hasonlóan egy függvénykönyvtár, azzal a különbséggel, hogy a könyvtár a **neurális hálózatok elkészítését és betanítását** teszi lehetővé.[22] A neurális hálózatok közül itt főként neurális hálózatokat (Neural Network) fogok használni, amelyek a képfeldolgozás, kép osztályozás területén teljesítenek kiemelkedően. A könyvtár eszközeiről szintén részletesebben beszélek majd a későbbi fejezetekben.

## 2. fejezet

# A golyók pozíciójának felismerése

### 2.1. A probléma ismertetése

Ebben a fejezetben a címből is láthatóan az asztal és azon elhelyezkedő **golyók pozíciójának felismeréséről** lesz szó. A program ezen része bemenetként fog fogadni egy képet. Ez a kép származhat **videófelvételből** (videófelvétel egy képkockája), **valós idejű felvételből** (szintén egy képkocka), vagy szimplán egy **képfájlból**. A program készítése közben egy valósághű internetes snooker játékról[5] készített felvételeket fogok használni a felismeréshez. A videójátékból származó felvétel azonban bizonyos szempontokból eltér a valóságtól, ezekről később lesz szó, de lényegében jól használható a felismerés elkészítéséhez, továbbá megkönnyíti a tesztelést és a felvételek beszerzését. A bemenő adatot ezután képfelismerési folyamatok után a 2.1 táblázatban látható koordinátákra alakítjuk.



2.1. ábra. Egy bemeneti kép az asztal felismerő lépés után.

A 2.1 képen már az asztal felismerés után való képet mutatom, a láthatóság megkönnyítése érdekében. A fejezet következő részeiben a bemeneti nyers képről koordinátákra való átalakítás lehetséges módszereit ismertetem lépésekre bontva.

### 2.2. Az asztal felismerése

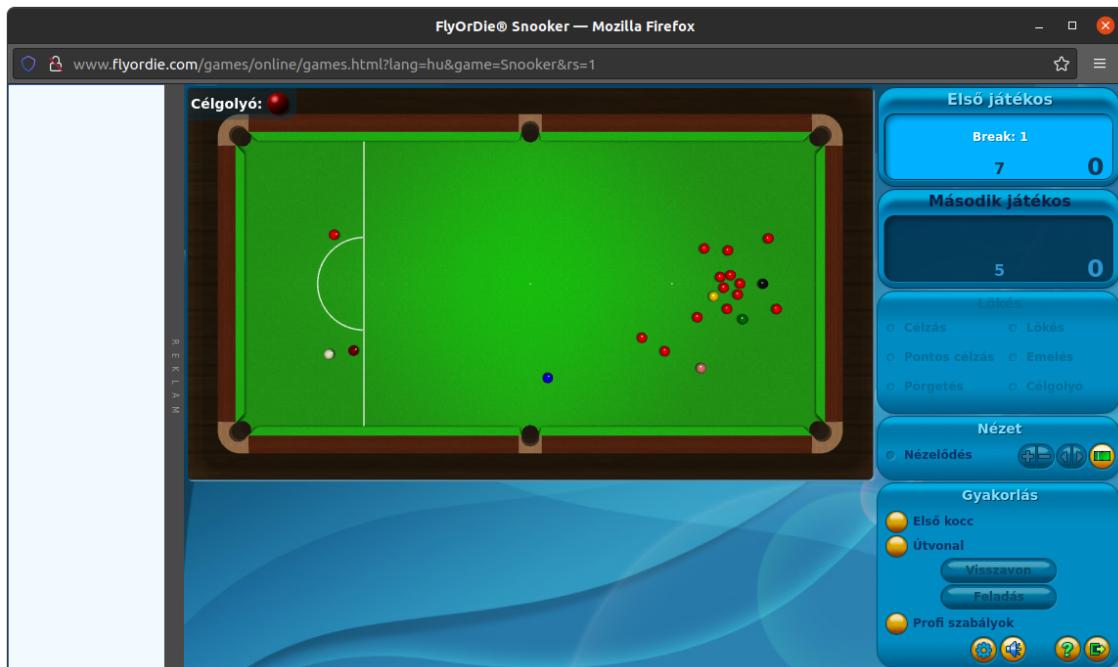
Ez a lépés kicsit elkülönül a többi lépéstől, hiszen a további lépések különféle módszerektől függetlenül, megegyezően ezen a lépésen alapszanak. Ahhoz hogy a további lépések pontosak legyenek és megfeleő teljesitménnyel működjenek, az asztalt mindenkorán azonos méretben, elforgatásban és torzításban kell megkapniuk.

Ebben a módszerben az elforgatáson kívül a detektálás, átméretezés és a torzítás lesz középpontban. Az elforgatást nem veszem figyelembe, hiszen a bemenetről elvárt, hogy

**2.1. táblázat.** A golyó felismerés kimeneti adatai a 2.1 kép alapján.

golyó színe	x pozíció (0 - 1024)	y pozíció (0 - 512)
yellow	156	176
brown	224	254
green	226	178
red	460	480
pink	464	344
blue	514	256
white	622	472
red	698	324
red	724	394
red	896	250
black	918	256

bizonyos orientációban kapjuk meg. Tehát amit valójában kapunk függetlenül a bemenet megszerzésének módszerétől, a 2.2 képen látható.



**2.2. ábra.** Egy nyers bemeneti kép a játék ablakáról.

Ezen a képen kell megtalálnunk a játékterület zöld részét. Ezt megtehetjük, ha először is a képünket ún. HSV (Hue Saturation Value) formátumra alakítjuk. Ennek a konverziónak a kimenete a 2.3 képen látható.

Ez az átalakítás azért fontos, mert HSV formátumban könnyebben intervallumok közé lehet szorítani a játékterület zöld színét.

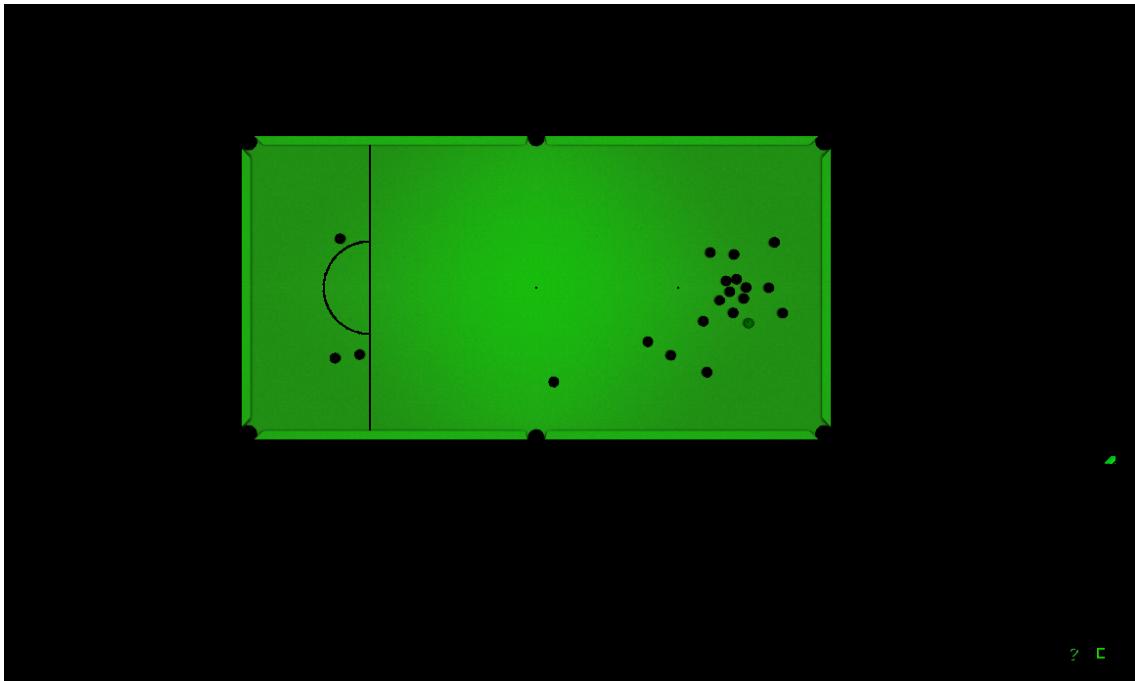
Ezek az intervallumok:

- Árnyalat (Hue)
- Telítettség (Saturation)
- Érték (Value)

Az adott intervallum értékeken kívül helyezkedő értékeket maszkoljuk, és eredményül a kívánt zöld területeket kapjuk. Ezt a képet a maszkolás után visszaalakítjuk RGB formátumra. A folyamat után kapott kép a 2.4 képen látható.



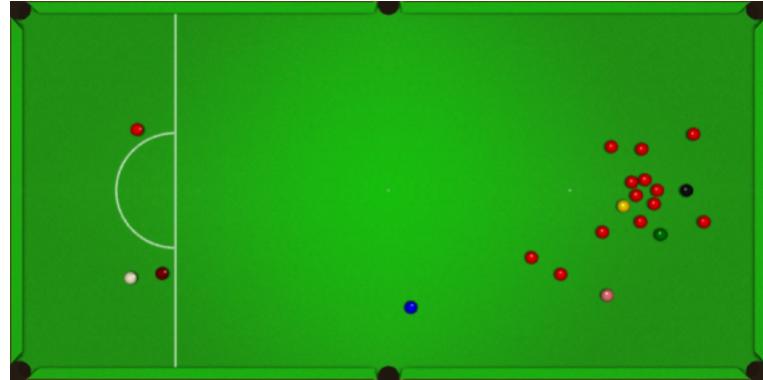
**2.3. ábra.** A 2.2 kép HSV re alakított verziója RGB reprezentációban.



**2.4. ábra.** A 2.2 kép a maszkolás után.

Az előző folyamat után már jól látható a játékterület, azonban vannak apró foltok amik nem kerületk maszkolásra. Ezek a foltok hasonló HSV értékekkel rendelkeznek, mint a játékterület, kiszűrésük megoldható a kontúrok megkeresésével, majd feltételezve, hogy a legnagyobb folt a maszkolt képen a játékterület, annak kiválasztásával. Ezzel az eljárással már megadható a játékterület kontúrja. Feltételezve, hogy ez a négy sarokpontból áll,

és egy téglalap pontosan határolja, a játékterület képe 2.5 megkapható a kontúr eredeti képből való kivágásával és átméretezésével.



**2.5. ábra.** Az eredeti képből kinyert játékterület.

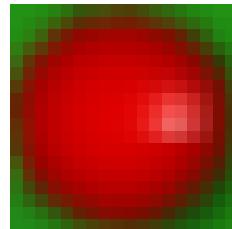
Fent említettem, hogy feltételezhetjük, hogy a játékterület kontúrja a kontúrkeresés után téglalap alakú, és a kontúrt alkotó pontok száma 4. Viszont valóságban ez nehezen fordul elő, ezért szükséges a pontok számának leszűkítése és a négy pontot határoló alakzatban lévő kép torzítása 2:1 oldalarányú téglalapra. Az oldalarány a szabvány snookerstal 12 x 6 láb (365,8 cm x 182,9 cm)[24] méretéből következik. A pontok szűkítéséről és a torzításról a megvalósítás fejezetben lesz részletesebben szó.

## 2.3. A golyók azonosítása

A golyók azonosítását különféle módszerekkel végezhetjük el, ezek eltérnek sebességen és pontosságban. A folyamatok bemenete az előzőekben megismert asztal felismerés kimenete lesz, kimenetük pedig a 2.1 táblázatban látható x és y pozíciók, adott golyók színe szerint. A folyamat belső működése módszerenként eltér egymástól, ezeket a módszereket az implementáció egyes iterációjának részeként használtam, majd változtattam meg az elért teljesítmény növeléséhez.

### 2.3.1. Azonosítás mintaillesztéssel

Ennek a módszernek az alapja tisztán mintaillesztéssel működik. A mintaillesztés egy arányos méretű képet illeszt rá a játékterület képére, majd amennyiben az illeszkedés mértéke meghalad egy bizonyos küszöbértéket, a mintaillesztés iterációjának a pozíciója mentésre kerül. Ebből a pozícióból meghatározható a golyó helyzete. A mintaillesztéshez használt minta a 2.6 ábrán látható.



**2.6. ábra.** A mintaillesztéshez használt minta.

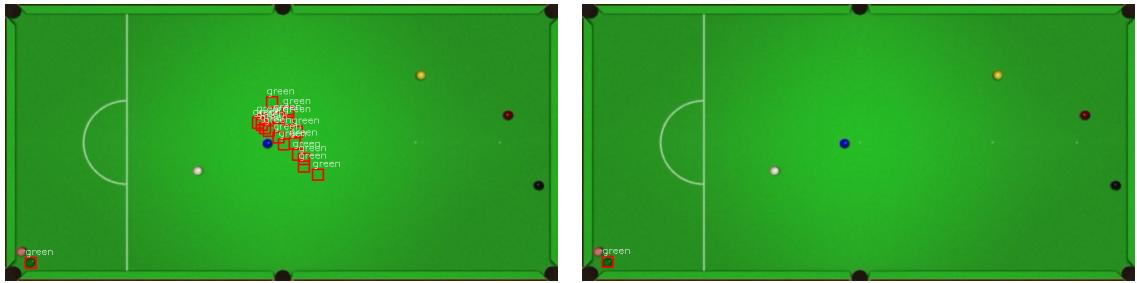
A 2.6 minta felbontása láthatóan alacsony, azonban túl nagy felbontás esetén a folyamat meglehetősen lassabban megy végbe, továbbá a minta méretét még a felismert játékterület mérete is meghatározza.

A mintaillesztés ún. Kereszt Korrelációval (Normed Cross Correlation) megy végbe, ezt a 2.1 képletben láthatjuk[16].

$$R(x, y) = \frac{\sum_{x',y'} (T(x', y') \cdot I(x + x', y + y'))}{\sqrt{\sum_{x',y'} T(x', y')^2 \cdot \sum_{x',y'} I(x + x', y + y')^2}} \quad (2.1)$$

A képletben az  $x$  és  $y$  az eredeti képen vizsgált terület bal felső sarkát,  $x'$  és  $y'$  a minta képének az adott képpontját,  $T$  a minta képet és  $I$  az eredeti képet jelöli.

A mintaillesztés problémái közé tartozik, hogy a zöld golyó mintaillesztésénél a küszöbérték et beállítani nehéz, és az eredmény pontatlan, lásd 2.7.



**2.7. ábra.** A zöld golyó mintaillesztésének hibája (bal) és annak orvoslása HSV konverzióval (jobb).

A 2.7 ábrán látható hiba valamelyes orvosolható a kép és minta HSV -re való konvertálásával. Ez a konverzió jó eredményeket ad, azonban nagyon minimálisan csökkenti a teljesítményt. A mintaillesztés sajnos problémákba ütközik a piros és barna golyók megkülönböztetésekor is. Ez a 2.8 képen látható.



**2.8. ábra.** A barna golyó mintaillesztésének hibája (bal) és annak orvoslása a piros golyók levonásával (jobb).

Itt a színek közelisége miatt nehéz megkülönböztetni a golyókat, ezért a piros és barna golyók nagyon minimálisan térnek el a 2.1 függvény miatt. Ez a probléma HSV konvertálás után is fennáll. Erre a megoldás, hogy az érzékeltek piros golyók kivonásra kerülnek a barna golyók listájából. Ahhoz, hogy pontos legyen az eredmény, viszont szükséges, hogy a piros golyók megfelelően legyenek érzékelve, amely nem minden esetben biztosítható, ezért ez a módszer nem túl pontos bizonyos bemenetekre. Hasonló problémák merülhetnek fel a piros és rózsaszín, továbbá a fehér és rózsaszín golyók felismerésekor is.

Annak ellenére, hogy a módszer nem túl optimális, jól használható adatkészletek készítésére, hiszen a felsmerést nagyrészt helyesen megoldja és a problémák ismeretében a kézzel válogatást nagymértékben megkönnnyíti.

### 2.3.2. Azonosítás körkeresés és mintaillesztéssel

Az előző módszerhez hasonlóan a golyó színek szerinti osztályozása itt is mintaillesztéssel történik, azonban a sebesség növelése érdekében itt először a golyókat kördetektálással azonosítjuk, majd ennek eredményét adjuk át a mintaillesztő algoritmusnak. Ez az egész képen való pástázás és mintaillesztéshez képest a teljesítményt a minták leszűkített mennyiségevel nagymértékben növeli.

A körök detektálásához az ún. Hough transzformációt (Hough Transformation) fogom használni, ez a H.K. Yuen, J. Princen, J. Illingworth és J. Kittler et. al. 1990 [25] szerint abban az esetben, ha egy kör a következő 2.2 függvénytel írható le,

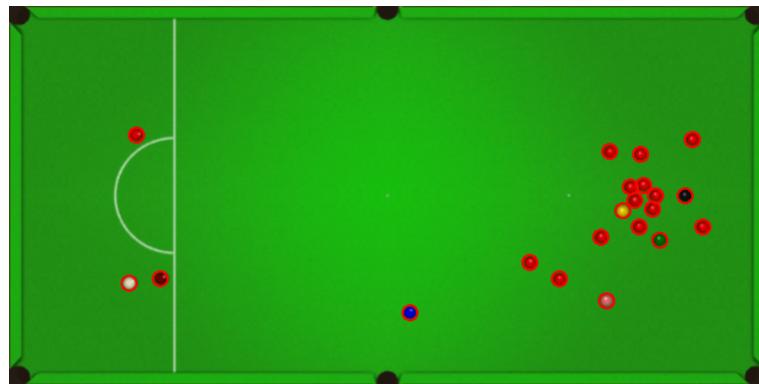
$$(x - a)^2 + (y - b)^2 = r^2 \quad (2.2)$$

ahol  $a$  és  $b$  a kör középpontjának koordinátái és  $r$  a sugár, akkor a körvonal élének egy tetszőleges  $x_i, y_i$  pontja átalakításra kerül egy  $a, b, r$  paraméterek által meghatározott térben elhelyezkedő egyenes kör alapú kúppá.[4, 25] Amennyiben az adott pontok egy körvonalon helyezkednek el, a kúpok metszeni fogják egymást a kör  $a, b, r$  pontjainak megfelelően.[25]

A körök megtalálásához az algoritmusnak meg kell adni néhány paramétert, ezek közé tartozhat:

- A keresett körök minimális és maximális sugara
- A keresett körök közti minimális távolság, duplikációk szűréséhez
- Az ellenőrzött alakzatok körrel való hasonlóságának küszöbértéke

Az algoritmus lefutása után megkapjuk a bemeneti játékterületen talált kör alakú kontúrokat. Ezt a 2.9 ábra szemlélteti.



**2.9. ábra.** A Hough transzformáció lefutása után kapott körök.

Ez a módszer körök megtalálására jól használható, a mintaillesztéshez szükséges képek könnyedén kivághatók az eredeti képből a körök paraméterei alapján. A probléma szintén a mintaillesztéssel van, hiszen a kivágott körök az előzőleg megismert mintaillesztéssel kerülnek beazonosításra. Ez sajnos az eddig megismert hibákat vonja maga után, annak ellenére, hogy a sebesség javul. Viszont akárcsak a szimpla mintaillesztéses módszer ez a módszer is alkalmas adatkészletek elkészítésére, és a kapott adatok kézzel ellenőrzését nagymértékben megkönyíti.

### 2.3.3. Azonosítás körkeresés és gépi tanulás segítségével

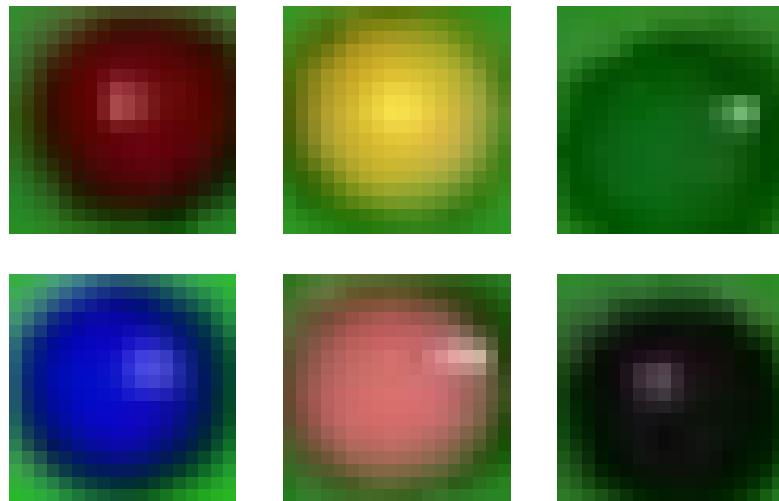
Ahhoz, hogy kiküszöböljük a mintaillesztés hibáit, az osztályozást elvégezhetjük neurális hálózat segítségével. Ezt megtehetjük ugyancsak, ha egy kerettel végigpásztázunk a játkterület képén, de a körkereséses módszernél megismert Hough transzformációval jobb

eredményt is elérhetünk. A következőkben a körkeresés eredményeképp kapott, a bemeneti játékterület képéből kivágott golyók osztályozásáról lesz szó neurális hálózat segítségével. A kivágott képek fogják a bemenetet képezni, majd a neurális hálózat azt osztályozza egy 0-tól 7-ig terjedő egész számként. Ezek a számok a golyók színeit reprezentálják, lásd 2.2 táblázat.

**2.2. táblázat.** A golyók szín szerint, és azok azonosítói.

Golyó színe	Azonosító
fekete	0
kék	1
barna	2
zöld	3
rózsaszín	4
piros	5
fehér	6
sárga	7

A neurális hálózat betanításához szükség van betanítási adatkészletre, viszont az előzőekben megsimert módszereknél szóba került, hogy viszonylag kevés kézi szortírozással könnyedén lehet velül előállítani adatkészletet, amely tökéletes a neurális hálózat betanításához és teszteléséhez. Az adatkészlet néhány eleme látható a 2.10 ábrán.



**2.10. ábra.** Az adatkészlet elemei.

A betöltött adatokat megfelelően azonosítva címkéjük szerint, átadjuk a neurális hálózatnak. Ahhoz, hogy a tanítás jó eredményeket hozzon, gondoskodni kell arról, hogy a betanítási adatok között egyenlő arányban szerepelnek az egyes golyók színek szerint, továbbá, hogy az adatkészlet elemei megfelelően össze vannak keverve. Az előkészített adatkészlet egy kis részét (10% - 30%) elkülönítjük, majd ezt használjuk a neurális hálózat pontosságának tesztelésére.

A betanítási folyamatok után elmentjük a neurális hálózatot, majd amikor használni szeretnénk, csak be kell tölteni. A tanítás és felhasználás külön programfájlokban könnyedén elvégezhető.

A körfelismerés módszerrel ötvözve, a neurális hálózattal való osztályozás gyors és pontos eredményeket biztosít. A felismerés egy kimenetele a 2.11 képen látható.



**2.11. ábra.** A neurális hálózattal való golyófelismerés kimenete.

A módszer eredményességének köszönhetően későbbiekben részletesebben ismertetem a megvalósítás fejezetben.

### **3. fejezet**

## **Analizálás**

## 4. fejezet

# Megvalósítás

### 4.1. Alkalmazott módszerek

A megvalósítás során az eddig megismert módszereket fogom felhasználni, azokat Python programozási nyelven fogom elkészíteni és ismertetni. Az egyes algoritmusokat függvények formájában készítem el, ezeket a függvényeket pedig több Python szkriptben is felhasználom.

Az eddig megismert módszerek közül elsősorban a nyers bemenetből emelem ki a játékterületet, ezt követően a golyók pozíciójának felismeréséhez kör detektálást és egy neurális hálózatot fogok használni. A neurális hálózat betanításához az adatkészletet kör detektálással, mintaillesztéssel és kézi válogatással fogom elkészíteni. A következőkben az egyes függvények működését, azokban felhasznált külső könyvtárak eszközeit ismertetem részleteiben.

A fejezetek felosztása az eddig megismert lépések szerint kerül rendezésre.

### 4.2. A golyók pozíciója

#### 4.2.1. A szükséges könyvtárak importálása

Ahhoz hogy a függvények megfeleően működjenek, meg kell mondanunk a programnak, hogy használja a külső könyvtárakat.

Ezt a 4.1 kódSOROK alapján tehetjük meg.

```
1 import math  
2 import numpy as np  
3 import cv2
```

#### 4.1. kódRészlet. Könyvtárak importálása.

A `math` könyvtár segítségével matematikai műveleteket (gyökvonás, szinusz, koszinusz) tudunk végezni, a `numpy` könyvtár a tömbök, mátrixok kezelését, azokkal való műveleteket segíti és gyorsítja, a `cv2` pedig az OpenCV eszközeit teszi elérhetővé.

#### 4.2.2. Az asztal kontúrjának megkeresése

Annak érdekében, hogy a nyers képből kinyerjük a játékterületet, azt először be kell tölteni egy többdimenziós tömbbe. A kép betöltése többféleképp végbemehet, ezért ezt konkrétan nem részletezem.

A betöltött kép tömbjének alakja megegyezik a kép szélességével és magasságával, továbbá az intenzitási értékekkel, tehát ha betöltünk egy 1024 x 512 méretű RGB képet,

annak tömbjének az első és második dimenziója 1024 és 512, a harmadik pedig az RGB (Piros, Zöld, Kék) intenzitásoknak megfelelően 3 méretű.

Fontos megjegyezni, hogy az OpenCV a képeket betöltéskor BGR formátumban tölti be, ez az elnevezésből adódóan annyiban tér el az RGB formátumtól, hogy a piros (R) és kék (B) színcsatornák fel vannak cserélve.

A nyers bemeneti kép megszerzése után készen állunk az asztal kontúrjának megkeresésére. Első lépésként a képet átalakítjuk HSV formátumra, majd megadjuk az alsó és felső intenzitási értékhatarokat, amelyből elkészítjük a maszkot. Ezután maszkoljuk az eredeti képet a maszk segítségével.

A fentieket a 4.2 kódrészlettel végezhetjük el.

```

1 hsv = cv2.cvtColor(image, cv2.COLOR_BGR2HSV)
2
3 lower_green = np.array([40, 190, 50])
4 upper_green = np.array([65, 255, 225])
5
6 mask = cv2.inRange(hsv, lower_green, upper_green)
7
8 result = cv2.bitwise_and(image, image, mask = mask)

```

#### 4.2. kódrészlet. A játékterület maszkolása.

A 4.2 kódrészletben az `image` a bemeneti képünk, amelyet a `cv2.cvtColor` függvénykel [10] konvertálunk át HSV formátumra. Ennek a függvénynek az első paramétere a bemeneti képünk, a második pedig a konverzió típusa, amely ebben az esetben  $BGR \rightarrow HSV$ . A `lower_green` és `upper_green` változók az alsó és felső intenzitási határokat jelölik sorrendben megfelelően. A maszk elkészítését a `cv2.inRange` függvénykel [15] végezhetjük el itt a paraméterek sorban a HSV re konvertált képünk, valamint az alsó és felső intenzitás értékek. A függvény a következők alapján dönti el, a maszk intenzitását,

$$M(I) = L(I) \leq S(I) \leq U(I) \quad (4.1)$$

ahol  $M$  a maszk,  $L$  az alsó,  $U$  a felső és  $S$  a bemeneti HSV képet jelöli. A 4.1 függvény mindenáron intenzitásra alkalmazásra kerül, a maszkban az intervallumon belüli intenzitások 255, a kívüliek pedig 0 értéket kapnak. A maszk elkészítése után azt alkalmazzuk az eredeti bemenő képre a `cv2.bitwise_and` függvény [8] segítségével. Itt a paraméterek a bejövő eredeti kép `image` kétszer és a maszk `mask`.

A folyamat során a metódus a következőképp jár el,

$$R(I) = S_1(I) \wedge S_2(I), \text{ha } M(I) \neq 0 \quad (4.2)$$

ahol  $R$  a kimenő maszkolt kép (`result`)  $S_1$  és  $S_2$  a két bemeneti kép paraméter, és  $M$  a maszk. A bemenetben a kép azért szerepel kétszer egymás után, mert a 4.2 függvényben láthatóan a két bemenő paraméter között egy bit szintű 'és' művelet történik, amennyiben a maszk nem nulla. Ez azt teszi lehetővé, hogy az eredeti képet kapjuk a maszkolt elemek kivételével, ami azért történik, mert bit szinten ha két megegyező elem között történik 'és' művelet, akkor az eredmény szintén megegyezik a két elemmel. Ennek a folyamatnak a kimenetele látható a már előzőleg tárgyalta 2.4 ábrán.

A maszkolt kép megszerzése után elvégezhetjük az éldetektálást, amelyet megelőz egy szürkeárnyalatolás.

```

1 image_gray = cv2.cvtColor(result, cv2.COLOR_BGR2GRAY)
2
3 edges = cv2.Canny(image_gray, 100, 200)

```

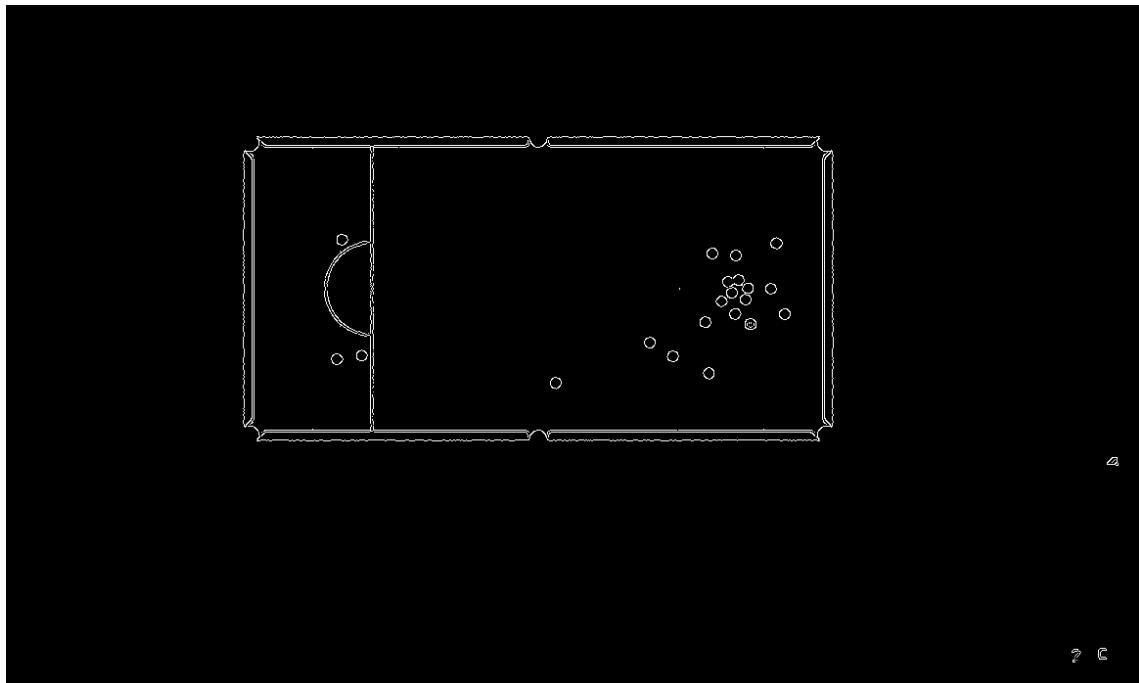
#### 4.3. kódrészlet. Szürkeárnyalatolás és éldetektálás.

A szürkeárnyalati konverziót a már megismert `cv2.cvtColor` függvényel [10] végezzük el a 4.3 kódrészlet alapján, majd ezután megkeressük az éleket a képen Canny éldetektálás [9, 3] (`cv2.Canny`) segítségével.

A Canny éldetektálás általában több lépésre bontható szét, ezek lehetnek:

- Homályosítás Gauss szűrővel [20] a zajcsökkentés érdekében
- Élek helyének és irányának megállapítása intenzitás-gradiensből
- Nem-Maximum vágás merőleges élek szűréshéhez
- Kettős küszöbölés élek szűréséhez

Az éldetektálásnál meg kell adnunk a függvénynek a szürkeárnyalatos képünket (`image_gray`), továbbá két küszöbértéket, amelyet a Canny detektálás a kettős küszöbölés folyamat során fog felhasználni. Itt, ha a felső küszöb felett van egy potenciális él, azt felvesszük az élek közé, ha az alsó küszöb alatt van eldobjuk és ha a felső és alsó küszöbök közt helyezkedik el, akkor a szomszédos pixelek alapján vesszük fel élnek. Az éldetektálással kapott kép (`edges`) a 4.1 ábrán látható.



**4.1. ábra.** A Canny éldetektálás után kapott kép.

A következő lépésben a bináris képen lefuttatásra kerül egy kontúrkereső algoritmus [21], majd a kapott kontúroknak vesszük a konvex körvonalát, azok egyszerűsítése, esetleges konkáv alakzatok megszüntetése érdekében. Ezek után feltételezve, hogy a kontúrok közül a legnagyobb a játszterület, kiválasszuk a körvonalak közül.

```

1 contours, _ = cv2.findContours(edges, cv2.RETR_LIST, cv2.CHAIN_APPROX_SIMPLE)
2
3 contours = [cv2.convexHull(c) for c in contours]
4 contours = sorted(contours, key=lambda x : cv2.contourArea(x), reverse=True)[:1]

```

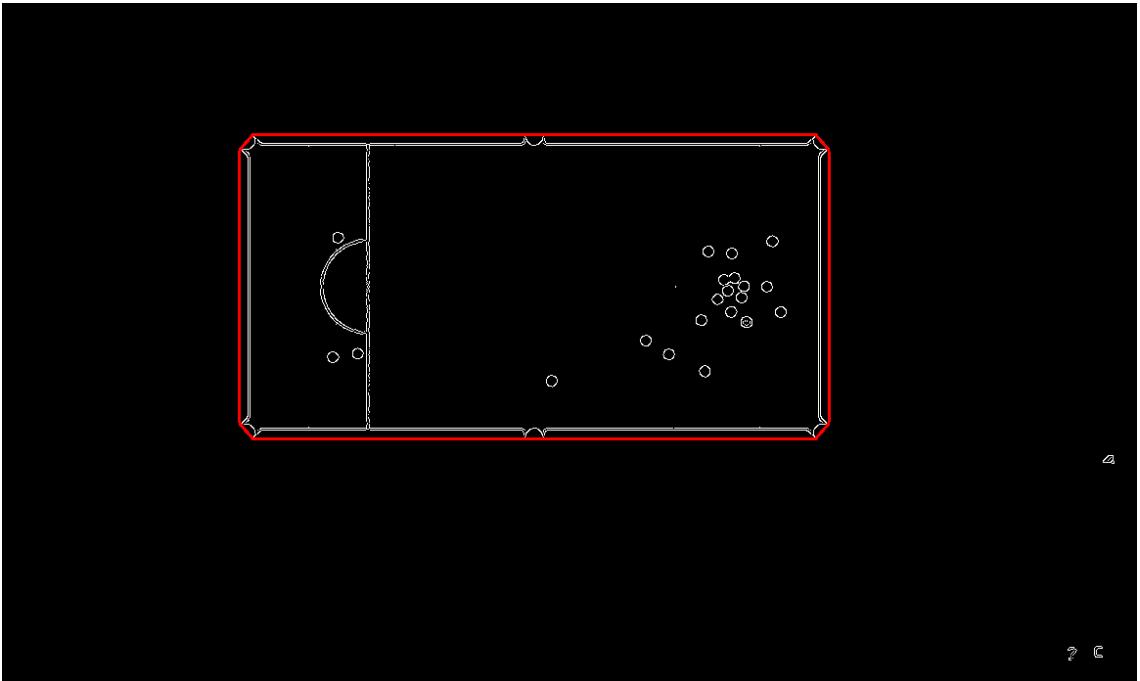
#### 4.4. kódrészlet. Kontúrok keresése.

A 4.4 kódrészletben található `cv2.findContours` függvény [13, 21] egy határkövetéses algoritmussal kigyűjt a kontúrokat. Ezek a kontúrok a képen található képpont koordináták láncolatából állnak össze. A kontúrok körvonalát a `cv2.convexHull` függvény [12, 19] segítségével kapjuk meg. Ez az algoritmus a kontúrok koordinátáinak láncolatát használja, majd

a kontúrt egy konvex körfonalal határolja, ugyancsak koordináták láncolatai formájában reprezentálva.

A fenti művelet elsőre feleslegesnek tűnhet, hiszen a keresett asztal kontúrja előre-láthatólag nem konkáv, a művelet elvégzése mégis fontos, hiszen így egyszerűíthetjük az alakzatunkat (pl.: kotúr koordináta láncolat pontjainak csökkentése), ezzel a következő műveleteket felgyorsítva.

A legnagyobb kontúr kiválasztásához tudnunk kell az egyes kontúrok területét. A területet a `cv2.contourArea` függvény [11] számolhatjuk ki. Ezt megtesszük minden eddigi kontúrral úgy, hogy szimplán paraméterként adjuk a függvénynek. A kiszámolt területek közül kiválasszuk a legnagyobbat, majd annak kontúr koordináta láncolatát eltároljuk. A kapott kontúr kirajzolva a 4.2 ábrán látható.



**4.2. ábra.** A felismert asztal kontúrja a bináris képen, piros körvonallal keretezve.

A `cv2.contourArea` függvény a Surveyor's Area algoritmust [2] használja az alakzatok területének számolásához. Ez az algoritmus a Green-tétel egy speciális esete, amely alkalmazható egyszerű sokszögekre.

Az algoritmus függvénye a következő,

$$A = \sum_{k=0}^n \frac{(x_{k+1} + x_k)(y_{k+1} - y_k)}{2} \quad (4.3)$$

ahol  $n$  az óramutató járásával ellentétesen rendezett kontúr koordináták száma,  $(x_k, y_k)$  a  $k$  adik koordináta  $x$  és  $y$  pozíciója, és feltételezhetjük, hogy a  $k = n + 1$  elem megegyezik a  $k = 0$  elemmel.

A 4.2 ábrán látható, hogy a kontúrunk téglalaphoz hasonló aljkának ellenére több, mint 4 pontból áll. Ahhoz hogy téglalap formájában ki tudjuk vágni a képet, meg kell keresnünk azt a négyzetet, amely a kontúrt határolja. Erre egy olyan algoritmust készítettem, amely megkeresi a kontúr koordináták segítségével a négy leghosszabb oldalt, majd kiszámolja ezek metszéspontját. A négy leghosszabb oldal használata feltételezi, hogy a

kép közel felső nézetből készült az asztalról, továbbá, hogy a sarkoknál jelenik meg több pont a kontúr keresés után.

Az oldalhosszok számolása a 4.4 képlet alapján megy végbe,

$$D = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2} \quad (4.4)$$

ahol  $D$  a kiszámolt pontok közti távolság,  $(x_a, y_a)$  és  $(x_b, y_b)$  pedig a két koordináta, amelyek között a távot számoljuk. Miután megkaptuk az oldalak hosszait, kiválasszuk a négy legnagyobbat, majd az ezekhez tartozó koordinátákat tároljuk. A kiválasztott pontokat fontos, hogy óramutató járásával ellentétes sorrendben rendezzük, amennyiben nem a négyzetöszög kontúrunk később hibás lehet.

A metszéspontok kiszámolásához a következő képleteket [23] használtam,

$$D = (x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4) \quad (4.5)$$

$$P_x = \frac{(x_1 y_2 - y_1 x_2)(x_3 - x_4) - (x_3 y_4 - y_3 x_4)(x_1 - x_2)}{D} \quad (4.6)$$

$$P_y = \frac{(x_1 y_2 - y_1 x_2)(y_3 - y_4) - (x_3 y_4 - y_3 x_4)(y_1 - y_2)}{D} \quad (4.7)$$

ahol a 4.5 képletben a  $D$  a 4.6 és 4.7 képletekben a nevező kiszámolásához biztosít könnyebb átláthatóságot,  $(P_x, P_y)$  a kiszámolt metszéspont,  $(x_1, y_1), (x_2, y_2), (x_3, y_3)$  és  $(x_4, y_4)$  pedig a négy pont, amelyek a két egyenest határozzák meg, itt ezek közül az első kettő az egyik, a második kettő a másik egyeneshez tartozik.

A fenti egyenletek megvalósítása a 4.5 kód részletben látható,

```

1 def intersection(p1, p2, p3, p4):
2     d = (p1[0] - p2[0]) * (p3[1] - p4[1]) - (p1[1] - p2[1]) * (p3[0] - p4[0])
3     if (abs(d) < 1e-8):
4         return False
5
6     t1 = (p1[0]*p2[1] - p1[1]*p2[0]) * (p3[0] - p4[0]) - (p3[0]*p4[1] - p3[1]*p4[0]) * (p1[0] - p2[0])
7     t2 = (p1[0]*p2[1] - p1[1]*p2[0]) * (p3[1] - p4[1]) - (p3[0]*p4[1] - p3[1]*p4[0]) * (p1[1] - p2[1])
8     return [t1 / d, t2 / d]

```

#### 4.5. kód részlet. Metszéspont kereső algoritmus.

ahol  $p_1, p_2, p_3, p_4$  a fent megismert négy koordináta,  $d$  a kiszámolt nevező,  $t_1$  és  $t_2$  pedig segédváltozók a számlálók tárolásához. A kód részlet 3. és 4. sorában látható, hogy abban az esetben ha  $d$  nagyon kicsi, a függvény `False` értéket ad vissza. Ez azért van, mert a 4.5 függvényben kiszámolt nevező,  $D = 0$  esetén a két egyenes párhuzamos.

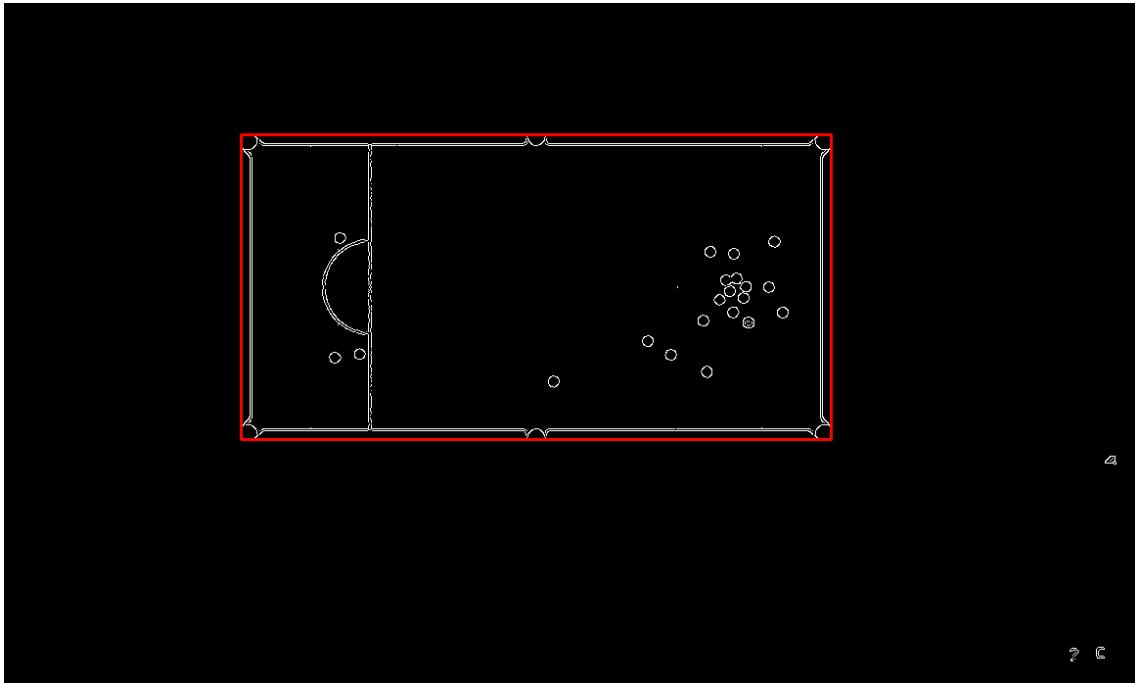
A folyamat végeredményeképp kapott kép a 4.3 ábrán látható.

#### 4.2.3. Az asztal kivágása és torzítása

Ahhoz, hogy az asztalt a kontúr segítségével kivágjuk a képből, szükségünk lesz egy téglalapra, amely alapján a kivágást elvégezzük. Ebben a részben ennek a folyamatnak a működéséről fogok beszélni.

A folyamat első része abból áll, hogy a kapott négy koordinátából álló kontúr pontjait rendezzük. A pontokat bal felső, jobb felső, jobb alsó és bal alsó sorrendben kell átrendeznünk.

Az átrendezéshez a 4.6 kód részletet használom,



**4.3. ábra.** A felismert asztal négy pontból álló körvonal a bináris képen, piros körvonallal keretezve.

```

1 pts = contour.reshape(4, 2)
2 src = np.zeros((4, 2), "float32")
3
4 sums = pts.sum(axis=1)
5 diffs = np.diff(pts, axis=1).flatten()
6
7 src[0] = pts[np.argmin(sums)]
8 src[1] = pts[np.argmin(diffs)]
9 src[2] = pts[np.argmax(sums)]
10 src[3] = pts[np.argmax(diffs)]

```

#### 4.6. kódrészlet. Átrendező algoritmus.

ahol először az első két sorban rendezem a koordináta pontokat a `pts` változóba és készítek egy `src` tömböt a rendezett adatok tárolásához. A következő lépésekben, ahhoz, hogy meg tudjam állapítani a pontok relatív helyzetét, készítek az egyes pontokból összegeket (`sums`) és különbségeket (`diffs`), amelyek az egyes koordináták  $x$  és  $y$  összetevőinek összegeiből vagy különbségeiből állnak. Ezekből az összegek és különbségekből megállapítható a pontok helyzete, tehát például a bal felső koordinátát az összegek közül a legkisebb érték, a bal alsót a különbségek közül a legnagyobb érték határozza meg, és így a többi koordinátát is. A fent említett műveletek a kódrészlet 7 - 10 soraiban láthatóak.

Az előző művelet után a sorba rendezett koordináták meghatározzák a transzformációhoz szükséges mátrix kiszámításához a forrás (`src`) értékeit. A transzformációhoz szükségünk van még a célértékekre is.

A célértékek a 4.7 kód soraival adhatóak meg,

```

1 width, height = (size[0] - 1, size[1] - 1)
2
3 dst = np.array([
4     [0, 0],
5     [width, 0],
6     [width, height],
7     [0, height]],
8     dtype="float32")

```

#### 4.7. kódrészlet. A kimeneti értékek megadása.

itt a cél képünk méretei egy érték páros formájában szerepelnek a `size` változóban, ezt szélesség és magasság elemekre bontjuk a `width` és `height` változókban. Az értékekből egyet való levonás az indexelés végett szükséges. A szélesség és magasság értékekkel ezután meg tudjuk adni a célértékeket a transzformációs mátrix elkészítéséhez, ezek a `dst` változóba kerülnek.

A transzformáció végrehajtásához mindezek után már csak a transzformációs mátrix elkészítésére van szükség, majd a transzformáció végrehajtására.

Ezeket a 4.8 kódrészlettel hajthatjuk végre,

```

1 M = cv2.getPerspectiveTransform(src, dst)
2 warp = cv2.warpPerspective(image, M, size)

```

#### 4.8. kódrészlet. A transzformáció végrehajtása.

itt `M` a transzformációs mátrixunk, amelyet a `cv2.getPerspectiveTransform` függvénytel [14] kapunk meg a forrás és célértékek megadásával. A függvény Gauss-elimináció [6] segítségevel számol ki egy  $3 \times 3$  méretű mátrixot, amelyet a `cv2.warpPerspective` függvénytel [17] alkalmazunk az `image` változóban tárolt képünkre a `M` mátrix és `size` méret megadásával. A transzformáló függvény lineáris interpolációt [1] használ alapértelmezett esetben az intenzitás értékek meghatározásához.

A kivágott és torzított kép a már megismert 2.5 ábrán látható.

#### 4.2.4. Körkeresés és azonosítás

# Ábrák jegyzéke

1.1.	A golyók kezdeti pozíciója . . . . .	3
2.1.	Egy bemeneti kép az asztal felismerő lépés után. . . . .	4
2.2.	Egy nyers bemeneti kép a játék ablakáról. . . . .	5
2.3.	A 2.2 kép HSV re alakított verziója RGB reprezentációban. . . . .	6
2.4.	A 2.2 kép a maszkolás után. . . . .	6
2.5.	Az eredeti képből kinyert játékterület. . . . .	7
2.6.	A mintaillesztéshez használt minta. . . . .	7
2.7.	A zöld golyó mintaillesztésének hibája (bal) és annak orvoslása HSV konverzióval (jobb). . . . .	8
2.8.	A barna golyó mintaillesztésének hibája (bal) és annak orvoslása a piros golyók levonásával (jobb). . . . .	8
2.9.	A Hough transzformáció lefutása után kapott körök. . . . .	9
2.10.	Az adatkészlet elemei. . . . .	10
2.11.	A neurális hálózattal való golyófelismerés kimenete. . . . .	11
4.1.	A Canny éldetektálás után kapott kép. . . . .	15
4.2.	A felismert asztal kontúrja a bináris képen, piros körvonallal keretezve. . . .	16
4.3.	A felismert asztal négy pontból álló körvonala a bináris képen, piros körvonallal keretezve. . . . .	18

# Irodalomjegyzék

- [1] T. Blu – P. Thevenaz – M. Unser: Linear interpolation revitalized. *IEEE Transactions on Image Processing*, 13. évf. (2004) 5. sz., 710–719. p.
- [2] Bart Braden: The surveyor’s area formula. *The College Mathematics Journal*, 17. évf. (1986) 4. sz., 326–337. p.
- [3] John Canny: A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8. évf. (1986) 6. sz., 679–698. p.
- [4] Richard O. Duda – Peter E. Hart: Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15. évf. (1972. jan) 1. sz., 11–15. p. ISSN 0001-0782. URL <https://doi.org/10.1145/361237.361242>. 5 p.
- [5] Snooker játék. URL <http://www.flyordie.hu/snooker/>.
- [6] Joseph F Grcar: Mathematicians of gaussian elimination. *Notices of the AMS*, 58. évf. (2011) 6. sz., 782–792. p.
- [7] OpenCV about, 2020. Nov. URL <https://opencv.org/about/>.
- [8] OpenCV bitwise and. URL [https://docs.opencv.org/3.4/d2/de8/group\\_\\_core\\_\\_array.html#ga60b4d04b251ba5eb1392c34425497e14](https://docs.opencv.org/3.4/d2/de8/group__core__array.html#ga60b4d04b251ba5eb1392c34425497e14).
- [9] OpenCV canny edge detection.  
URL [https://docs.opencv.org/3.4/dd/d1a/group\\_\\_imgproc\\_\\_feature.html#ga04723e007ed888ddf11d9ba04e2232de](https://docs.opencv.org/3.4/dd/d1a/group__imgproc__feature.html#ga04723e007ed888ddf11d9ba04e2232de).
- [10] OpenCV color conversion. URL [https://docs.opencv.org/3.4/d8/d01/group\\_\\_imgproc\\_\\_color\\_\\_conversions.html#ga397ae87e1288a81d2363b61574eb8cab](https://docs.opencv.org/3.4/d8/d01/group__imgproc__color__conversions.html#ga397ae87e1288a81d2363b61574eb8cab).
- [11] OpenCV contour area. URL [https://docs.opencv.org/3.4/d3/dc0/group\\_\\_imgproc\\_\\_shape.html#ga2c759ed9f497d4a618048a2f56dc97f1](https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#ga2c759ed9f497d4a618048a2f56dc97f1).
- [12] OpenCV convex hull. URL [https://docs.opencv.org/3.4/d3/dc0/group\\_\\_imgproc\\_\\_shape.html#ga014b28e56cb8854c0de4a211cb2be656](https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#ga014b28e56cb8854c0de4a211cb2be656).
- [13] OpenCV find contours. URL [https://docs.opencv.org/3.4/d3/dc0/group\\_\\_imgproc\\_\\_shape.html#ga17ed9f5d79ae97bd4c7cf18403e1689a](https://docs.opencv.org/3.4/d3/dc0/group__imgproc__shape.html#ga17ed9f5d79ae97bd4c7cf18403e1689a).
- [14] OpenCV calculate perspective transform matrix.  
URL [https://docs.opencv.org/4.x/da/d54/group\\_\\_imgproc\\_\\_transform.html#ga20f62aa3235d869c9956436c870893ae](https://docs.opencv.org/4.x/da/d54/group__imgproc__transform.html#ga20f62aa3235d869c9956436c870893ae).
- [15] OpenCV masking. URL [https://docs.opencv.org/3.4/d2/de8/group\\_\\_core\\_\\_array.html#ga48af0ab51e36436c5d04340e036ce981](https://docs.opencv.org/3.4/d2/de8/group__core__array.html#ga48af0ab51e36436c5d04340e036ce981).

- [16] OpenCV correlation template matching (CCORR).  
URL [https://docs.opencv.org/3.4/df/dfb/group\\_\\_imgproc\\_\\_object.html#gga3a7850640f1fe1f58fe91a2d7583695daf9c3ab9296f597ea71f056399a5831da](https://docs.opencv.org/3.4/df/dfb/group__imgproc__object.html#gga3a7850640f1fe1f58fe91a2d7583695daf9c3ab9296f597ea71f056399a5831da).
- [17] OpenCV apply perspective transform matrix.  
URL [https://docs.opencv.org/4.x/da/d54/group\\_\\_imgproc\\_\\_transform.html#gaf73673a7e8e18ec6963e3774e6a94b87](https://docs.opencv.org/4.x/da/d54/group__imgproc__transform.html#gaf73673a7e8e18ec6963e3774e6a94b87).
- [18] M.I. Shamos: *The New Illustrated Encyclopedia of Billiards*. G - Reference, Information and Interdisciplinary Subjects Series sorozat. 2002, Lyons Press. ISBN 9781585746859. URL <https://books.google.hu/books?id=B02BAAAAMAAJ>.
- [19] Jack Sklansky: Finding the convex hull of a simple polygon. *Pattern Recognition Letters*, 1. évf. (1982) 2. sz., 79–83. p. ISSN 0167-8655. URL <https://www.sciencedirect.com/science/article/pii/0167865582900162>.
- [20] George Stockman – Linda G. Shapiro: *Computer Vision*. Upper Saddle River, NJ, USA, 2001, Prentice Hall PTR, 166. p. ISBN 0130307963.
- [21] Satoshi Suzuki – Keiichi A be: Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30. évf. (1985) 1. sz., 32–46. p. ISSN 0734-189X. URL <https://www.sciencedirect.com/science/article/pii/0734189X85900167>.
- [22] TensorFlow, 2020. Nov. URL <https://www.tensorflow.org/>.
- [23] Weisstein & Eric W.: Line-line intersection.  
URL <https://mathworld.wolfram.com/Line-LineIntersection.html>.
- [24] WPBSA: *Official Rules of the Games of Snooker and English Billiards*. 2019, The World Professional Billiards & Snooker Association Limited. URL <https://wpbsa.com/wp-content/uploads/WPBSA-Official-Rules-of-the-Games-of-Snooker-and-Billiards-2020.pdf>.
- [25] HK Yuen – J Princen – J Illingworth – J Kittler: Comparative study of hough transform methods for circle finding. *Image and Vision Computing*, 8. évf. (1990) 1. sz., 71–77. p. ISSN 0262-8856. URL <https://www.sciencedirect.com/science/article/pii/026288569090059E>.