

SZAKDOLGOZAT

Snooker billiardjáték elemzése

Lengyel Márk

Mérnökinformatikus BSc
Mérnökinformatikus Szakirány

2022

Tartalomjegyzék

Ábrák jegyzéke	1
1. Bevezetés	2
2. A Snooker játék	3
2.1. Általánosságban a snooker játékról	3
2.2. Eszközök	3
2.3. Pontszerzés	3
3. Felhasznált szoftverek	5
3.1. Az OpenCV képfeldolgozási könyvtár	5
3.2. Tensorflow gépi tanulási könyvtár	5
4. A program megtervezése	7
4.1. Az asztal felismerése	7
4.2. A golyók azonosítása	9
4.2.1. Azonosítás mintaillesztéssel	10
4.2.2. Azonosítás körkeresés és mintaillesztéssel	11
4.2.3. Azonosítás körkeresés és gépi tanulás segítségével	12
4.3. Analizálás	13
5. A felismerő megvalósítása	14
5.1. Alkalmazott módszerek	14
5.2. A szükséges könyvtárak importálása	14
5.3. Az asztal kontúrjának megkeresése	14
5.4. Az asztal kivágása és torzítása	19
5.5. Körkeresés	20
5.6. A detektált körök osztályozása	21
5.6.1. Osztályozás mintaillesztéssel	21
5.6.2. Osztályozás neurális hálózattal	21
Irodalomjegyzék	23

Ábrák jegyzéke

2.1. A golyók kezdeti pozíciója.	4
4.1. Egy nyers bemeneti kép a felvételről.	7
4.2. A 4.1 kép HSV re alakított verziója RGB reprezentációban.	8
4.3. A 4.1 kép a maszkolás után.	8
4.4. Az eredeti képből kinyert játékterület.	9
4.5. A mintaillesztéshez használt minta.	10
4.6. A zöld golyó mintaillesztésének hibája (felül) és annak orvoslása HSV konverzióval (alul).	10
4.7. A barna golyó mintaillesztésének hibája (felül) és annak orvoslása a piros golyók levonásával (alul).	11
4.8. A Hough transzformáció lefutása után kapott körök.	12
4.9. Az adatkészlet elemei.	13
4.10. A neurális hálózattal való golyófelismerés kimenete.	13
5.1. A Canny éldetektálás után kapott kép.	17
5.2. A felismert asztal kontúra a bináris képen, piros körvonallal keretezve. . . .	18
5.3. A felismert asztal négy pontból álló körvonala a bináris képen, piros körvonallal keretezve.	19

1. fejezet

Bevezetés

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 eszközeinek használatával.

A bevezetés későbbi részeiben ismertetem a snooker billiardjátékot, továbbá a projekt elkészítéséhez használt programozási nyelvet és fejlesztési könyvtárakat.

2. fejezet

A Snooker játék

2.1. Általánosságban a snooker játékról

A snooker a billiárdjátékok 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)[1]. 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.

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.[1]

Az egyes golyók különböző pontértékkal rendelkeznek:[1]

- 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 pontértékkel, mivel a játékosok ezt a golyót használják lökéseikhez.

A játék egy menetét **frémnek** nevezik, amely a kezdő lökéstől a fekete golyó elhelyezéséig tart.[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** nevezik. Egy játékos például szerzhet 9 pontos törést a következő golyók egymás utáni elhelyezésével *piros -> zöld -> piros -> barna*.[2]



2.1. ábra. A golyók kezdeti pozíciója.

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.[1]

3. fejezet

Felhasznált szoftverek

3.1. Az OpenCV képfeldolgozási könyvtár

Az OpenCV (Open Source Computer Vision Library) egy főként **valós idejű képfeldolgozáshoz** használt programozási függvénykönyvtár. A könyvtár többféle programozási nyelvhez készült implementációval létezik (pl.: C++, Python, Java stb.)[3], amelyek közül ebben a projektben a Python programozási nyelvhez készült verziót fogom használni. A szoftver szabadon használható az *Apache License 2.0* alatt.

A függvénykönyvtár fejlesztését az Intel Research kezdeményezte 1999-ben, a CPU intenzív alkalmazások fejlődése érdekében. A projekt lefőbb hozzájárulói az Intel Russia optimalizálással foglalkozó szakemberei, továbbá az Intel Performance Library csapata. [4]

Kezdetben az OpenCV létrehozásának célja volt, hogy **nyílt, optimalizált** kódot képezzent gépi látáshoz, valamint, hogy egy **egységes infrastruktúrát** biztosítson a fejlesztőknek a területen, ezzel megkönnyítve a programkódok olvashatóságát és terjesztőségét. Cél volt még, hogy fejlesszék a gépi látásra alapuló kereskedelmi felhasználást, hordozható, teljesítményorientált programkód létrehozásával.[5]

Az OpenCV-t sokféle területen használják, ezek közül néhány:

- arcfelismerő rendszerek,
- gesztusok felismerése,
- objektumok felismerése,
- szegmentálás és felismerés,
- mozgás felismerés,
- kiterjesztett valóság.

A dolgozat keretében 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 részletességen.

3.2. Tensorflow gépi tanulási könyvtár

A Tensorflow az OpenCV -hez hasonlóan egy függvénykönyvtár, amely a **neurális hálózatok elkészítését és betanítását** teszi lehetővé. A Tensorflow egy nyílt forráskódú szoftver könyvtár, használható különböző feladatok elvégzésére is, de főként mély neurális hálózatok betanítására és azokkal való következtetésekre, becslésekre használható. [6]

A Tensorflow a Google Brain csapat által lett kifejlesztve a Google saját kutatásaihoz. Az első verzió az *Apache License 2.0* szoftverlicenz alatt jelent meg, majd később 2019

szeptemberében megjelent a Tensorflow frissített verziója, amelyet Tensorflow 2.0nak neveztek el.[6]

A könyvtár használható különböző programozási nyelvekben is (Python, C++, Javascript, stb.) amelynek köszönhetően flexibilisen használható különféle alkalmazásokban. A Tensorflow sokfélé funkcióval rendelkezik, ezek közül a következőkben néhányat részletesen megemlítek.

A Tensorflow funkciói közé tartozik, hogy támogatja az **automatikus differenciálás** (Automatic differentiation) folyamatát, amellyel automatikusan kiszámolható a gradiens vektor egy modelhez, annak paramétereit figyelembe véve. Ez a folyamat különösen hasznos a visszaterjesztéses (Backpropagation) modelleknek, ahol gradiensekre van szükség az optimalizáláshoz. Ahhoz, hogy ez megvalósítható legyen a keretrendszer számon tartja a modell bemenetére végzett műveleteket, majd a modell paramétereitől függően kiszámolja a gradienseket. [6, 7]

A könyvtár lehetővé teszi továbbá, a számítások **elosztását** különböző hardver eszközök között, ezzel a tanítás és kiértékelés folyamatok nagymértékben felgyorsíthatóak főként komplex, több rétegű modellek esetén.[6]

A modellek tanításához nélkülözhetetlen a **költségfüggvények használata**, ezek rendelkezésre állnak a könyvtárban. A költségfüggvények szerepe, hogy kiszámolják a "hiba", vagy másnéven eltérést a modell kimenete, és annak az adott bemenethez tartozó elvárt kimenete között, amely érték segítségével a modell hangolni tudja paramétereit.

A Tensorflow könyvtárban megtalálható egy ún. `tf.nn` modul, amelynek segítségével egyszerű műveletek végezhetők neurális hálózatokon. Ilyen műveletek lehetnek konvolúciók képfelismeréshez, aktivációs függvények (Softmax, RELU, Sigmoid, stb.) és egyéb primitív műveletek.

A könyvtár részét képezik különböző **optimalizálók**, amelyek a modellek betanításában játszanak szerepet, különböző optimalizálók lehetővé teszik a paraméterek különféle módon való hangolását, ezzel kihatva a modell teljesítményére. Az ilyen optimalizálók közül az egyik legismertebb az *ADAM Optimizer*, amelyet ebben a munkában is felhasználunk.

Neurális hálózatok közül főként **konvolúciós neurális hálózatokat** (Convolutional Neural Network) fogok használni a későbbiekben, amelyek a képfeldolgozás, kép osztályozás területén teljesítenek kiemelkedően. A könyvtár egyes eszközeiről szintén részletesebben beszélek majd a megvalósítással kapcsolatos fejezetben.

4. fejezet

A program megtervezése

4.1. Az asztal felismerése

Ebben az alfejezetben a program első folyamatáról lesz szó, ami nem más mint az asztal, vagy más néven játékerület felismerése. 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 bemutatásához egy felülnézetből készített snooker játszma felvételét fogom használni.

Ebben a módszerben az elforgatást leszámítva a detektálás, átmérétezés és a torzítás lesz középpontban. Az elforgatást nem veszem figyelembe, hiszen a bemenetről feltételezhető, hogy bizonyos orientációban áll rendelkezésre.

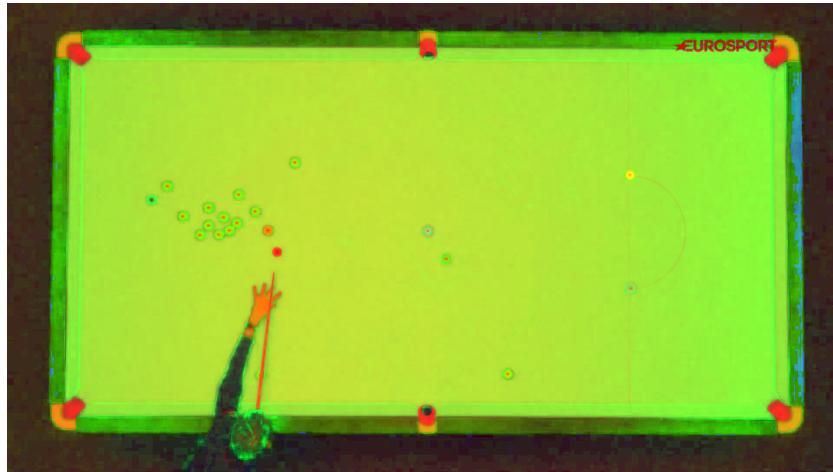
A valódi bemeneti kép, függetlenül annak előállítási módszerétől, a 4.1 képen látható.



4.1. ábra. Egy nyers bemeneti kép a felvételről.

Ezen a képen kell megtalálni a játékterület zöld részét. Ez könnyen megtehető a kép ún. HSV (Hue Saturation Value) formátumra való átalakításával. Ennek a konverziónak a kimenete a 4.2 képen látható.

Ez az átalakítás azért fontos, mert HSV formátumban könnyebben intervallumok közé lehet szorítani a játékterület zöld színét. A HSV konverzió belső működéséről részletesebben:



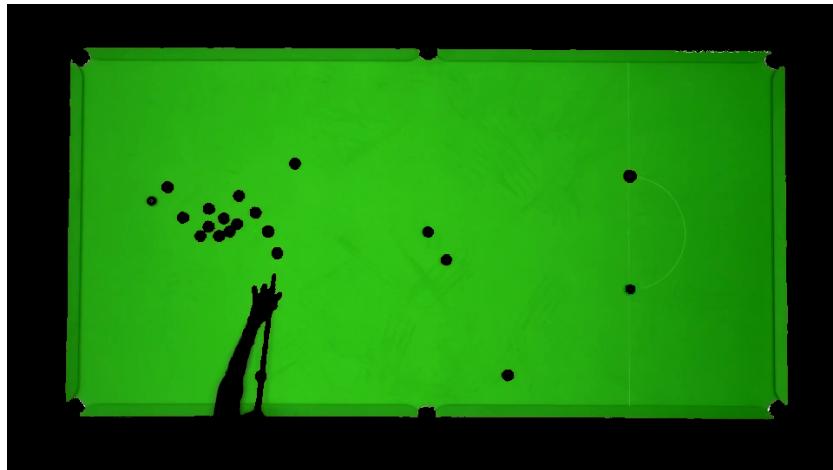
4.2. ábra. A 4.1 kép HSV re alakított verziója RGB reprezentációban.

ben a megvalósítás részben lesz szó.

A HSV konverzió során kapott intervallumok:

- árnyalat (Hue),
- telítettség (Saturation),
- érték (Value).

A HSV konverzió után az adott specifikus intervallumon kívül helyezkedő értékek maszkolásra kerülnek, hátrahagyva a kívánt játékterület képpontjait. A kép a maszkolás után visszaalakításra kerül az eredeti RGB formátumára. A folyamat után kapott kép a 4.3 képen látható.



4.3. ábra. A 4.1 kép a maszkolás után.

Az előző folyamat után már jól látható a játékterület, azonban vannak apró foltok amik nem kerültek 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 kontúra. Szintén feltételezve, hogy ez négy sarokpontból áll, és egy téglalap pontosan határolja, a játékterület képe 4.4 megkapható a kontúr eredeti képből való kivágásával és átméretezésével.



4.4. ábra. Az eredeti képből kinyert játékterület.

Fent említettem, hogy feltételezhető, 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 snookerasztal 12 x 6 láb (365,8 cm x 182,9 cm)[1] méretéből következik. A pontok szűkítéséről és a torzításról a megvalósítás fejezetben lesz részletesebben szó.

4.2. A golyók azonosítása

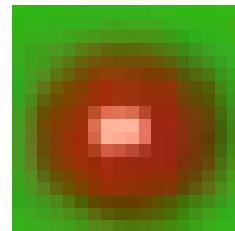
A golyók azonosítását különféle módszerekkel lehet elvégezni, ezek eltérnek sebességben és pontosságban. A folyamatok bemenete az előzőekben megismert asztal felismerés kimenete lesz, kimenetük pedig a 4.1 táblázatban látható x és y pozíciók, adott golyók színe szerint. A folyamat belső működése módszerenként eltér egymástól, ezeket a módszereket az implementáció egyes iterációjának részeként használtam, majd változtattam meg az elért teljesítmény növeléséhez.

4.1. táblázat. A golyó felismerés kimeneti adatai a 4.4 kép alapján.

golyó színe	x pozíció [0, 1023]	y pozíció [0, 511]
fehér	298.5	283.5
piros 0	244.5	204.5
piros 1	242.5	243.5
piros 2	323.5	159.5
piros 3	190.5	260.5
piros 4	202.5	222.5
piros 5	216.5	259.5
piros 6	268.5	227.5
piros 7	144.5	192.5
piros 8	625.5	451.5
piros 9	166.5	234.5
piros 10	202.5	247.5
piros 11	231.5	254.5
piros 12	223.5	234.5
sárga	797.5	174.5
zöld	797.5	331.5
barna	536.5	291.5
kék	512.5	252.5
rózsaszín	285.5	253.5
fekete	120.5	211.5

4.2.1. Azonosítás mintaillesztéssel

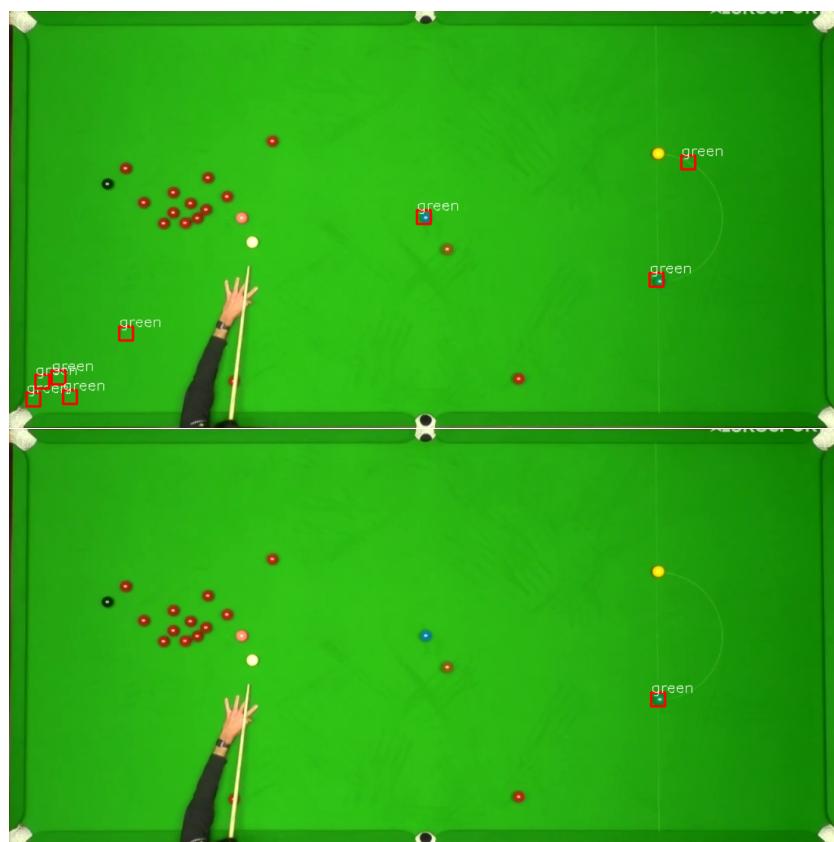
Ennek a módszernek az alapja tisztán mintaillesztéssel működik. A mintaillesztés egy arányos méretű képet illeszt rá a játékterület képére, majd amennyiben az illeszkedés mértéke meghalad egy bizonyos küszöbértéket, a mintaillesztés iterációjának a pozíciója mentésre kerül. Ebből a pozíóból meghatározható a golyó helyzete. A mintaillesztéshez használt minta a 4.5 ábrán látható.



4.5. ábra. A mintaillesztéshez használt minta.

A 4.5 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 problémái közé tartozik, hogy a zöld golyó mintaillesztésénél a küszöbértéket beállítani nehéz, és az eredmény pontatlan, lásd 4.6.



4.6. ábra. A zöld golyó mintaillesztésének hibája (felül) és annak orvoslása HSV konverzióval (alul).

A 4.6 ábrán látható hiba valamelyest 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 megkü-lönböztetésekor is. Ez a 4.7 képen látható.



4.7. ábra. A barna golyó mintaillesztésének hibája (felül) és annak orvoslása a piros golyók levonásával (alul).

Itt a színek közelisége miatt nehéz megkülönböztetni a golyókat, ezért a piros és barna golyók nagyon minimálisan térnek el a 5.12 függvény miatt. Ez a probléma HSV konver-tálás után is fennáll. Erre a megoldás, hogy az érzékelt 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 megkönníti.

4.2.2. Azonosítás körkeresés és mintaillesztéssel

Az előző módszerhez hasonlóan a golyó színek szerinti osztályozása itt is mintaillesztéssel történik, azonban a sebesség növelése érdekében először a golyók helyzetét egy körde-tektáló algoritmus határozza meg, majd a kapott értékek jutnak tovább a mintaillesztő algoritmushoz. Ez az egész képen való pásztázás és mintaillesztéshez képest a teljesítményt a minták leszűkített mennyiségének köszönhetően nagymértékben növeli.

A körök megtalálásához a körkereső algoritmusnak meg kell adni néhány paramétert, ezek közé tartoznak:

- a keresett körök minimális és maximális sugara,

- a keresett körök közti minimális távolság, duplikációk szűréséhez,
- az ellenőrzött alakzatok körrel való hasonlóságának küszöbértéke.

Az algoritmus lefutás után megadja a bemeneti játékterületen talált kör alakú kontúrokat, ezeket a 4.8 ábra szemlélteti. A körkereső algoritmusról a megvalósítás fejezetben írok részletesebben.



4.8. 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 megkönnyíti.

4.2.3. Azonosítás körkeresés és gépi tanulás segítségével

A mintaillesztés hibáinak kiküszöböléséhez, az osztályozás elvégezhető neurális hálózat segítségével. Ez a játékterület egy kerettel való képpontonkénti végigpásztázásával szintén megoldható, azonban körkeresésnél megismert Hough transzformációval jobb teljesítmény érhető el.

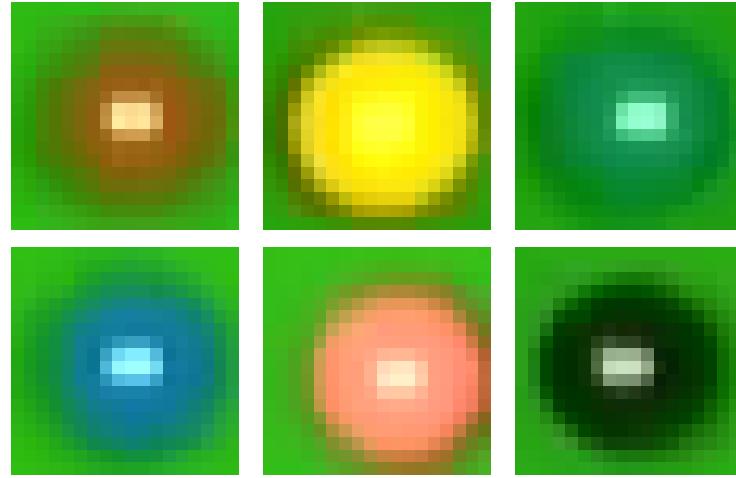
A következőkben a körkeresés eredményeképp kapott, a bemeneti játékterület képéből kivágott golyók osztályozásáról lesz szó neurális hálózat segítségével. A kivágott képek fogják a bemenetet képezni, majd a neurális hálózat azt osztályozza egy 0-tól 7-ig terjedő egész számkkal. Ezek a számok a golyók színeit reprezentálják, lásd 4.2 táblázat.

4.2. 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 velük előállítani adatkészletet, amely tökéletes a neurális hálózat betanításához és teszteléséhez. Az adatkészlet néhány eleme a 4.9 ábrán látható.

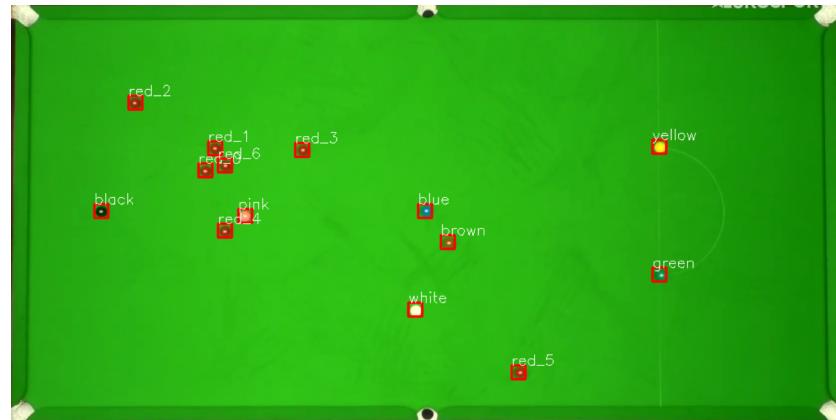


4.9. ábra. Az adatkészlet elemei.

A betöltött adatok címkével megfelelően azonosítva átadásra kerülnek a neurális hálózatnak. Ahhoz, hogy a tanítás jó eredményeket hozzon, gondoskodni kell arról, hogy a betanítási adatok közt egyenlő arányban szerepelnek az egyes golyók színek szerint, továbbá, hogy az adatkészlet elemei megfelelően össze vannak keverve. Az előkészített adatkészlet egy kis része (10% - 30%) elkülönítésre kerül, amely a neurális hálózat pontosságának tesztelésére fog szolgálni.

A betanítási folyamatok után a neurális hálózat mentésre kerül, így az későbbi kívánt használatba vétel esetén egyszerűen betölthető, 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 4.10 képen látható.



4.10. ábra. A neurális hálózattal való golyófelismerés kimenete.

A módszer eredményességének köszönhetően későbbiekbén részletesebben ismertetem a megvalósítás fejezetben.

4.3. Analizálás

5. fejezet

A felismerő megvalósítása

5.1. Alkalmazott módszerek

A felismerő program megvalósítása során az eddig megismert módszereket fogom felhasználni, azokat C++ programozási nyelven fogom elkészíteni és ismertetni. Az egyes algoritmusokat függvények formájában készítem el, ezeket a függvényeket pedig több helyen 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 következőkben az egyes függvények működését, azokban felhasznált külső könyvtárak eszközeit ismertetem részleteiben.

A fejezetek felosztása az eddig megismert lépések szerint kerül rendezésre.

5.2. A szükséges könyvtárak importálása

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

A legfontosabb könyvtárak importálása a 5.1 kódSOROK alapján lehető meg.

```
1 #include <opencv2/opencv.hpp>
2 #include <fdeep/fdeep.hpp>
```

5.1. kódrészlet. Könyvtárak importálása.

Az `opencv.hpp` könyvtár az OpenCV által biztosított képfeldolgozási függvényeket biztosítja, a `fdeep.hpp` könyvtár pedig a Tensorflow -al készített neurális hálózat betöltését teszi lehetővé.

5.3. Az asztal kontúrjának megkeresése

A nyers képből a játékterület megszerzéséhez 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 egy 1024 x 512 méretű RGB képet betöltve, 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 következik az asztal kontúrjának megkeresése. Első lépésként a képet HSV formátumra kell alakítani, majd az alsó és felső intenzitási értékhatárok megadásával meghatározható a maszk, amely alkalmazható az eredeti képre. A fentieket a 5.2 kódrészlettel végzem 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)

```

5.2. kódrészlet. A játékterület maszkolása.

A 5.2 kódrészletben az `image` a bemeneti kép, amelyet a `cv2.cvtColor` függvénnyel [8] konvertálók át HSV formátumra. Ennek a függvénynek az első paramétere a bemeneti kép, a második pedig a konverzió típusa, amely ebben az esetben $\text{BGR} \rightarrow \text{HSV}$.

A BGR értékekből a HSV értékek kiszámolásához először az értéket (Value) kell kiszámolni, ez a 5.1 egyenlet[8] szerint lehetséges,

$$V \leftarrow \max(R, G, B) \quad (5.1)$$

ahol V az értéket (Value) jelöli, R , G és B pedig az adott képpont három színkomponensét (Piros, Zöld, Kék). Az egyenlet alapján az érték a három színkomponens közül a legnagyobbnak az értékét fogja felvenni. Az érték kiszámolásával megadható a telítettség (Saturation).

Ezt a 5.2 egyenlet[8] alapján lehet kiszámolni,

$$S \leftarrow \begin{cases} \frac{V - \min(R, G, B)}{V} & , \text{ha } V \neq 0 \\ 0 & , \text{különben} \end{cases} \quad (5.2)$$

itt S a telítettséget (Saturation) jelöli, és az előzőekhez hasonlóan V az értéket (Value), R , G és B pedig a színkomponenseket, továbbá $\min(R, G, B)$ a három színkomponens közül a legkisebbet adja meg.

Az árnyalatot (Hue) szintén az érték (Value) segítségével lehet kiszámolni, ezt a 5.3 egyenlet[8] adja meg,

$$H \leftarrow \begin{cases} \frac{60(G-B)}{V-\min(R,G,B)} & , \text{ha } V = R \\ \frac{120+60(B-R)}{V-\min(R,G,B)} & , \text{ha } V = G \\ \frac{240+60(R-G)}{V-\min(R,G,B)} & , \text{ha } V = B \\ 0 & , \text{ha } R = G = B \end{cases} \quad (5.3)$$

ahol H az árnyalatot (Hue), V az értéket (Value), R , G és B a három színkomponensem, $\min(R, G, B)$ pedig a színkomponensek közül a minimálisat jelöli.

Amennyiben H értéke kisebb, mint 0, annak értéke $H \leftarrow H + 360$ szerint alakul. A 8-bites és 16-bites színnel rendelkező képeknél R , G és B értéke kezdetben normalizálásra kerül a $[0, 1]$ intervallumba, ennek következtében a három értéknél $0 \leq V \leq 1$, $0 \leq S \leq 1$, $0 \leq H \leq 360$ tartományok jelentkeznek. Az értékek visszaállítása tartománynak megfelelően 8-bites képek esetében az értékek megszerzése után $V \leftarrow 255V$, $S \leftarrow 255S$ és $H \leftarrow H/2$ szerint megy végbe, ez hasonló 16-bites szín esetében is. 32-bites színnel rendelkező képeknél nincs kezdeti normalizálás, és ennek következtében visszaalakítás sem szükséges.[8]

Az 5.2 kódrészlethez visszatérve, a `lower_green` és `upper_green` változók az alsó és felső intenzitási határokat jelölik sorrendnek megfelelően. A maszk elkészítését a `cv2.inRange` függvénnyel [8] végzem el, itt a paraméterek sorban a HSV re konvertált kép, valamint az alsó és felső intenzitás értékek.

A függvény a 5.4 egyenlet alapján dönti el, a maszk intenzitását,

$$M(I) = L(I) \leq S(I) \leq U(I) \quad (5.4)$$

ahol M a maszk, L az alsó, U a felső és S a bemeneti HSV képet jelöli. A 5.4 függvény minden három intenzitásra alkalmazásra kerül, a maszkban az intervallumon belüli intenzitások 255, a kívüliek pedig 0 értéket kapnak. A maszk elkészítése után a maszkolás megtörténik 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 5.5 egyenlet szerint jár el,

$$R(I) = S_1(I) \wedge S_2(I), \text{ha } M(I) \neq 0 \quad (5.5)$$

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 5.5 függvényben láthatóan a két bemenő paraméter között egy bit szintű 'és' művelet történik, amennyiben a maszk nem nulla. Ennek eredményeképp az eredeti kép adódik vissza amelyen maszkolt képpontok feketével szerepelnek. Ez 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 tárgyalt 4.3 ábrán.

A maszkolt kép megszerzése után elvégezhető az éldetektálás, amelyet megelőz egy szürkeárnyalatolás.

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

5.3. kódrészlet. Szürkeárnyalatolás és éldetektálás.

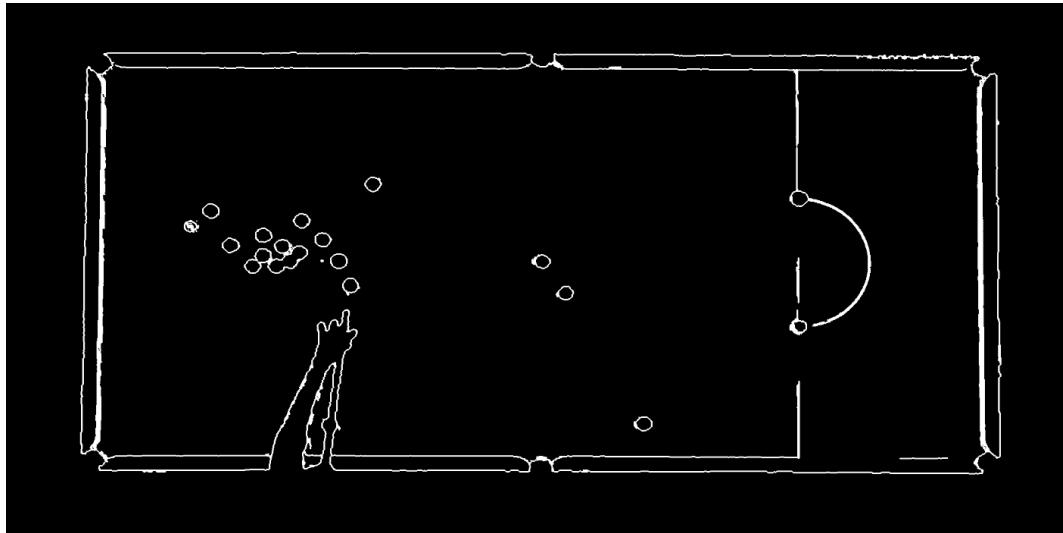
A szürkeárnyalati konverziót a már megismert `cv2.cvtColor` függvénnyel [8] végzem el a 5.3 kódrészlet alapján, majd ezután megkeressem az éleket a képen Canny éldetektálás [8, 9] (`cv2.Canny`) segítségével.

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

- homályosítás Gauss szűrővel [10] a zajcsökkentés érdekében,
- élek helyének és irányának megállapítása intenzitás-gradiensből,
- nem-maximum vágás merőleges élek szűréshéz,
- kettős küszöbölés élek szűréséhez.

Az éldetektálásnál meg kell adni a függvénynek a szürkeárnyalatos képet (`image_gray`), továbbá két küszöbertéket, amelyet a Canny detektálás a kettős küszöbölés folyamat során fog felhasználni. Itt, ha a felső küszöb felett van egy potenciális él, az hozzáadódik az élek közé, ha az alsó küszöb alatt van eldobódik és ha a felső és alsó küszöbök között helyezkedik el, akkor a szomszédos pixelek alapján kerül az élek közé. Az éldetektálással kapott kép (`edges`) a 5.1 ábrán látható.

A következő lépésekben a bináris képen lefuttatásra kerül egy kontúrkereső algoritmus [11], majd a kapott kontúrok vesszék a konvex körfonalát, azok egyszerűsítése, esetleges konkáv alakzatok megszüntetése érdekében. Ezek után feltételezve, hogy a kontúrok közül a legnagyobb a játékterület, az kiválasztható a körfonalak közül.



5.1. ábra. A Canny éldetektálás után kapott kép.

```

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]

```

5.4. kódrészlet. Kontúrok keresése.

A 5.4 kódrészletben található `cv2.findContours` függvény [8, 11] egy határkövetéses algoritmust kigyűjti a kontúrokat. Ezek a kontúrok a képen található képpont koordináták láncolatából állnak össze. A kontúrok körfelülete a `cv2.contourArea` függvény [8, 12] segítségével kapható meg. Ez az algoritmus a kontúrok koordinátáinak láncolatát használja, majd a kontúrt egy konvex körfelülettel 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 egyszerűsíthető az alakzat (kontúr koordináta láncolat pontjainak csökkentése), ezzel a folytatónak műveleteket felgyorsítva.

A legnagyobb kontúr kiválasztásához tudni kell az egyes kontúrok területét. A területet a `cv2.contourArea` függvénytel [8] lehet kiszámolni. Ez megtehető minden eddig kontúr esetében függvények való paraméterkénti átadással. A kiszámolt területek közül a legnagyobbat kiválasztva, annak kontúr koordináta láncolata eltárolásra kerül. A kapott kontúr kirajzolva a 5.2 ábrán látható.

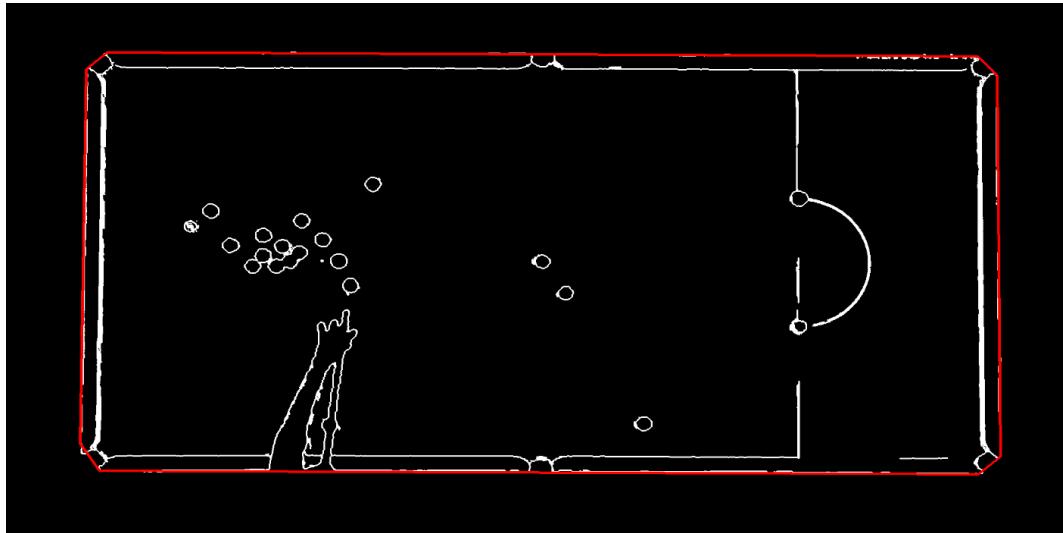
A `cv2.contourArea` függvény a Surveyor's Area algoritmust [13] használja az alakzatok területének számolásához. Ez az algoritmus a Green-tétel egy speciális esete, amely alkalmazható egyszerű sokszögekre.

Az algoritmus a 5.6 egyenletben látható,

$$A = \sum_{k=0}^n \frac{(x_{k+1} + x_k)(y_{k+1} - y_k)}{2} \quad (5.6)$$

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

A 5.2 ábrán látható, hogy a kontúr téglalaphoz hasonló alakjának ellenére több, mint 4 pontból áll. Ahhoz hogy téglalap formájában legyen kivágva a kép, meg kell keresni azt a négyzetet, amely a kontúrt határolja. Erre egy olyan algoritmust készítettem, amely



5.2. ábra. A felismert asztal kontúrja a bináris képen, piros körvonalal keretezve.

megkeresi a kontúr koordináták segítségével a négy leghosszabb oldalt, majd kiszámolja ezek metszéspontját. A négy leghosszabb oldal használata feltételezi, hogy a kép közeli felső nézetből készült az asztalról, továbbá, hogy a sarkoknál jelenik meg több pont a kontúr keresés után.

Az oldalhosszok számolása a 5.7 képlet alapján megy végbe,

$$D = \sqrt{(x_a - x_b)^2 + (y_a - y_b)^2} \quad (5.7)$$

ahol D a kiszámolt pontok közti távolság, (x_a, y_a) és (x_b, y_b) pedig a két koordináta, amelyek közt a táv számolandó. Miután megvannak az oldalak hosszai, eltárolásra kerül a négy legnagyobb oldalhoz tartozó koordináta. A kiválasztott pontoknál fontos, hogy óramutató járásával ellentétes sorrendben legyenek rendezve, amennyiben nem, a négyzetkör kontúr később hibás lehet.

A metszéspontok kiszámolásához a következő képleteket [14] használtam,

$$D = (x_1 - x_2)(y_3 - y_4) - (y_1 - y_2)(x_3 - x_4) \quad (5.8)$$

$$P_x = \frac{(x_1 y_2 - y_1 x_2)(x_3 - x_4) - (x_3 y_4 - y_3 x_4)(x_1 - x_2)}{D} \quad (5.9)$$

$$P_y = \frac{(x_1 y_2 - y_1 x_2)(y_3 - y_4) - (x_3 y_4 - y_3 x_4)(y_1 - y_2)}{D} \quad (5.10)$$

ahol a 5.8 képletben a D a 5.9 és 5.10 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 5.5 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]

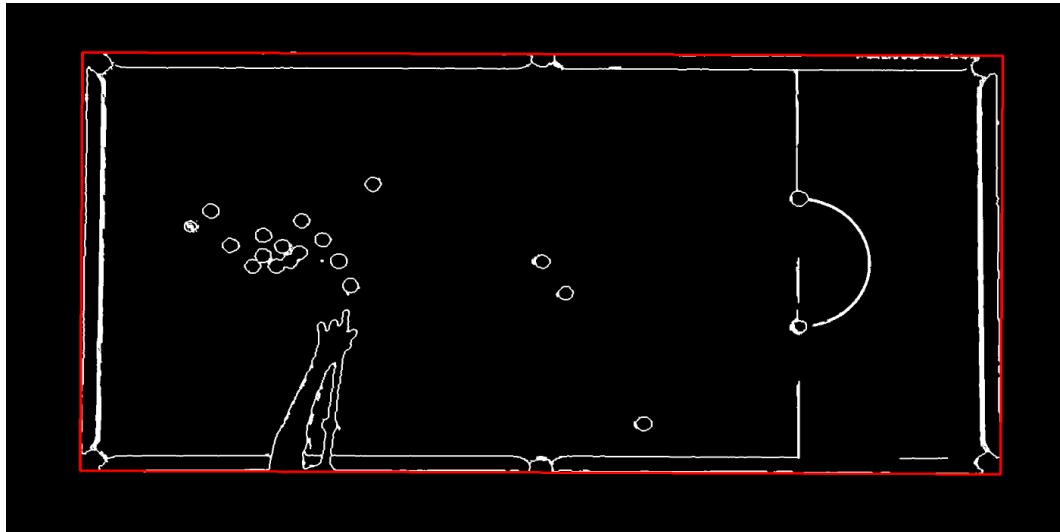
```

5.5. kódrészlet.

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 kódrészlet 3. és 4. sorában látható, hogy abban az esetben ha d nagyon kicsi, a függvény `False` értéket ad vissza. Ez azért van, mert a 5.8 függvényben kiszámolt nevező, $D = 0$ esetén a két egyenes párhuzamos.

A folyamat végeredményeképp kapott kép a 5.3 ábrán látható.



5.3. ábra. A felismert asztal négy pontból álló körfonal a bináris képen, piros körfonalall keretezve.

5.4. Az asztal kivágása és torzítása

Ahhoz, hogy az asztal a kontúr segítségével kivágható legyen a képből, szükség lesz egy téglalapra, amely alapján a kivágás elvégezhető. Ebben a részben ennek a folyamatnak a működéséről fogok beszálni.

A folyamat első részeként a kapott, négy koordinátából álló kontúr pontjait rendezni kell. A pontokat bal felső, jobb felső, jobb alsó és bal alsó pontok szerint kell sorba rendezni. Az átrendezéshez a 5.6 kódot használom,

```

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)]

```

5.6. 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 pontkból 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 kód részlet 7 - 10 soraiban láthatóak.

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

A célértékek a 5.7 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")

```

5.7. kód részlet. A kimeneti értékek megadása.

itt a cél kép méretei egy érték páros formájában szerepelnek a `size` változóban, ezek szélesség és magasság elemekre bomlanak a `width` és `height` változókban. Az értékekből egyet való levonás az indexelés végett szükséges. A szélesség és magasság értékekkel ezután meg lehet 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.

Ezek a 5.8 kód részlettel hajthatóak végre,

```

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

```

5.8. kód részlet. A transzformáció végrehajtása.

itt `M` a transzformációs mátrix, amely a `cv2.getPerspectiveTransform` függvényvel [8] kaptható meg a forrás és célértékek megadásával. A függvény Gauss-elimináció [15] segítségével számol ki egy 3×3 méretű mátrixot, amelyet a `cv2.warpPerspective` függvényel [8] alkalmaznak az `image` változóban tárolt képre az `M` mátrix és `size` méret megadásával. A transzformáló függvény lineáris interpolációt [16] használ alapértelmezett esetben az intenzitás értékek meghatározásához.

A kivágott és torzított kép a már megismert 4.4 ábrán látható.

5.5. Körkeresés

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 [17] szerint abban az esetben, ha egy kör a kövekező 5.11 függvényel írható le,

$$(x - a)^2 + (y - b)^2 = r^2 \quad (5.11)$$

ahol a és b a kör középpontjának koordinátái és r a sugár, akkor a kör vonal élénk egy tetszőleges x_i, y_i pontja átalakításra kerül egy a, b, r paraméterek által meghatározott téren elhelyezkedő egyenes kör alapú kúppá.[18, 17] Amennyiben az adott pontok egy

körvonalon helyezkednek el, a kúpok metszeni fogják egymást a kör a , b , r pontjainak megfelelően.[17]

Az algoritmus lefutása után a metszéspontok megadják az egyes körök pozícióját, amelyeket könnyedén tárolni lehet egy listában.

A körkereső algoritmus a programkód formájában a 5.9 kódrészletben látható.

```

1 gray = cv2.cvtColor(image, cv2.COLOR_BGR2GRAY)
2
3 minRadius = 5
4 maxRadius = 11
5 minDistance = 15
6
7 circles = cv2.HoughCircles(
8     gray,
9     cv2.HOUGH_GRADIENT,
10    1,
11    minDistance,
12    param1=80,
13    param2=15,
14    minRadius=minRadius,
15    maxRadius=maxRadius)
```

5.9. kódrészlet. A körkereső algoritmus.

A kódrészletben a Hough transzformációt a `cv2.HoughCircles` függvény[8] végzi el, ennek első paramétere egy szürkeárnyalatos kép, amely a kivágott asztal konvertálásával kerül bele a `gray` változóba, a konvertálás a már megismert `cv2.cvtColor` függvénnyel[8] megy végbe.

A második paraméter az előzőleg megismert Hough transzformációs módszert[8, 18, 17] adja meg a `cv2.HOUGH_GRADIENT` kulcsszóval, továbbá a harmadik paraméter a folyamathoz felhasznált képet skálázza. A skálázás az eredeti kép felbontást 1 értékkel nem változtatja, 2 értékkel felére csökkenti azt, fordított arányosságnak megfelelően[8]. Ez a paraméter a folyamat lefutásának gyorsítását teszi lehetővé, azonban, akárcsak a jelenlegi esetben, kisebb körök detektálásához jobb az eredeti felbontás megtartása.

A `minRadius` (minimális sugár), `maxRadius` (maximális sugár) és `minDistance` (minimálistávolság) paraméterek megadják a keresett körök tulajdonságait, így növelhető az algoritmus teljesítménye és csökkenthető a duplikációk előfordulása. A `param1` és `param2` névvel ellátott paraméterek a `cv2.HOUGH_GRADIENT` használatakor az algoritmus kezdeti feldolgozó lépéseként végrehajtott Canny éldetektálás felső (`param1`) és alsó (`param2`) küszöbértékét adják meg[8]. Ez a Canny éldetektálás a 5.3 alfejezetben megismert módszerrel megegyezően hajtódi kitér.

A folyamat lefutása után a körök pozíciója és mérete ismeretében jelölni tudjuk őket a kivágott képen. Ezt szemlélteti a 4 fejezet 4.8 ábrája.

5.6. A detektált körök osztályozása

5.6.1. Osztályozás mintaillesztéssel

A mintaillesztés ún. Kereszt Korrelációval (Normed Cross Correlation) megy végbe, ez a 5.12 képletben látható[4, 8].

$$R(x, y) = \frac{\sum_{x',y'}(T(x', y') \cdot I(x + x', y + y'))}{\sqrt{\sum_{x',y'} T(x', y')^2 \cdot \sum_{x',y'} I(x + x', y + y')^2}} \quad (5.12)$$

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

5.6.2. Osztályozás neurális hálózattal

Irodalomjegyzék

- [1] WPBSA. *Official Rules of the Games of Snooker and English Billiards*, 2019. URL <https://wpbsa.com/wp-content/uploads/WPBSA-Official-Rules-of-the-Games-of-Snooker-and-Billiards-2020.pdf>.
- [2] M.I. Shamos. *The New Illustrated Encyclopedia of Billiards*. G - Reference, Information and Interdisciplinary Subjects Series. Lyons Press, 2002. ISBN 9781585746859. URL <https://books.google.hu/books?id=B02BAAAAMAAJ>.
- [3] G. Bradski. The OpenCV Library. *Dr. Dobb's Journal of Software Tools*, 2000.
- [4] A. Kaehler and G. Bradski. *Learning OpenCV 3: Computer Vision in C++ with the OpenCV Library*. O'Reilly Media, 2016. ISBN 9781491938003. URL <https://books.google.hu/books?id=SKy3DQAAQBAJ>.
- [5] G. Bradski and A. Kaehler. *Learning OpenCV: Computer Vision with the OpenCV Library*. O'Reilly Media, 2008. ISBN 9780596554040. URL <https://books.google.hu/books?id=seAgi0fu2EIC>.
- [6] Martín Abadi, Ashish Agarwal, Paul Barham, Eugene Brevdo, Zhifeng Chen, Craig Citro, Greg S. Corrado, Andy Davis, Jeffrey Dean, Matthieu Devin, Sanjay Ghemawat, Ian Goodfellow, Andrew Harp, Geoffrey Irving, Michael Isard, Yangqing Jia, Rafal Jozefowicz, Lukasz Kaiser, Manjunath Kudlur, Josh Levenberg, Dandelion Mané, Rajat Monga, Sherry Moore, Derek Murray, Chris Olah, Mike Schuster, Jonathon Shlens, Benoit Steiner, Ilya Sutskever, Kunal Talwar, Paul Tucker, Vincent Vanhoucke, Vijay Vasudevan, Fernanda Viégas, Oriol Vinyals, Pete Warden, Martin Wattenberg, Martin Wicke, Yuan Yu, and Xiaoqiang Zheng. TensorFlow: Large-scale machine learning on heterogeneous systems, 2015. URL <https://www.tensorflow.org/>. Software available from tensorflow.org.
- [7] Introduction to gradients and automatic differentiation. Introduction to gradients and automatic differentiation. URL <https://www.tensorflow.org/guide/autodiff>.
- [8] OpenCV Documentation. Opencv dokumentáció. URL <https://docs.opencv.org/4.5.0/>.
- [9] John Canny. A computational approach to edge detection. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, PAMI-8(6):679–698, 1986. DOI: [10.1109/TPAMI.1986.4767851](https://doi.org/10.1109/TPAMI.1986.4767851).
- [10] George Stockman and Linda G. Shapiro. *Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. ISBN 0130307963.
- [11] Satoshi Suzuki and KeichiA be. Topological structural analysis of digitized binary images by border following. *Computer Vision, Graphics, and Image Processing*, 30(1):32–46, 1985. ISSN 0734-189X. DOI: [10.1016/S0734-189X\(85\)80005-7](https://doi.org/10.1016/S0734-189X(85)80005-7).

- [https://doi.org/10.1016/0734-189X\(85\)90016-7](https://doi.org/10.1016/0734-189X(85)90016-7). URL <https://www.sciencedirect.com/science/article/pii/0734189X85900167>.
- [12] Jack Sklansky. Finding the convex hull of a simple polygon. *Pattern Recognition Letters*, 1(2):79–83, 1982. ISSN 0167-8655. DOI: [https://doi.org/10.1016/0167-8655\(82\)90016-2](https://doi.org/10.1016/0167-8655(82)90016-2). URL <https://www.sciencedirect.com/science/article/pii/0167865582900162>.
 - [13] Bart Braden. The surveyor’s area formula. *The College Mathematics Journal*, 17(4):326–337, 1986.
 - [14] Weisstein & Eric W. Line-line intersection. URL <https://mathworld.wolfram.com/Line-LineIntersection.html>.
 - [15] Joseph F Grcar. Mathematicians of gaussian elimination. *Notices of the AMS*, 58(6):782–792, 2011.
 - [16] T. Blu, P. Thevenaz, and M. Unser. Linear interpolation revitalized. *IEEE Transactions on Image Processing*, 13(5):710–719, 2004. DOI: 10.1109/TIP.2004.826093.
 - [17] HK Yuen, J Princen, J Illingworth, and J Kittler. Comparative study of hough transform methods for circle finding. *Image and Vision Computing*, 8(1):71–77, 1990. ISSN 0262-8856. DOI: [https://doi.org/10.1016/0262-8856\(90\)90059-E](https://doi.org/10.1016/0262-8856(90)90059-E). URL <https://www.sciencedirect.com/science/article/pii/026288569090059E>.
 - [18] Richard O. Duda and Peter E. Hart. Use of the hough transformation to detect lines and curves in pictures. *Commun. ACM*, 15(1):11–15, jan 1972. ISSN 0001-0782. DOI: 10.1145/361237.361242. URL <https://doi.org/10.1145/361237.361242>.