

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	4
3. Felhasznált szoftverek	5
3.1. Az OpenCV képfeldolgozási könyvtár	5
3.2. Tensorflow gépi tanulási könyvtár	6
4. A program megtervezése	8
4.1. Az asztal felismerése	8
4.2. A golyók azonosítása	10
4.2.1. Azonosítás mintaillesztéssel	11
4.2.2. Azonosítás körkeresés és mintaillesztéssel	13
4.2.3. Azonosítás körkeresés és gépi tanulás segítségével	14
4.3. A játékmenet elemzése	16
4.3.1. A piros labdák felcímkézése	16
4.3.2. Megtett távolság és útvonal	17
4.3.3. A golyó sebessége	17
4.3.4. Zsebbe helyezés megállapítása	18
5. A felismerő megvalósítása	20
5.1. Alkalmazott módszerek	20
5.2. A szükséges könyvtárak importálása	20
5.3. Egyénileg készített osztályok és struktúrák	21
5.4. Az asztal kontúrjának megkeresése	22
5.5. Az asztal kivágása és torzítása	29
5.6. Körkeresés	31

5.7.	A detektált körök osztályozása	32
5.7.1.	Osztályozás mintaillesztéssel	32
5.7.2.	Osztályozás neurális hálózattal	34
5.8.	Játékmenet vizsgálati szempontok	36
5.8.1.	A piros labdák megkülönböztetése	36
5.8.2.	Az elemzési szempontok kiszámolása	38
	Irodalomjegyzék	43

Ábrák jegyzéke

2.1.	A golyók kezdeti pozíciója.	4
4.1.	Egy nyers bemeneti kép a felvételről.	9
4.2.	A 4.1 kép HSV re alakított verziója RGB reprezentációban.	9
4.3.	A 4.1 kép a maszkolás után.	10
4.4.	Az eredeti képből kinyert játékterület.	10
4.5.	A mintaillesztéshez használt minta.	11
4.6.	A zöld golyó mintaillesztésének hibája (felül) és annak orvoslása HSV konverzióval (alul).	12
4.7.	A barna golyó mintaillesztésének hibája (felül) és annak orvoslása a piros golyók levonásával (alul).	13
4.8.	A Hough transzformáció lefutása után kapott körök.	14
4.9.	Az adatkészlet elemei.	15
4.10.	A neurális hálózattal való golyófelismerés kimenete.	16
4.11.	A fehér golyó útvonala a felismert képen kék vonallal jelölve.	18
5.1.	A BallLabel enum felépítése.	21
5.2.	A Ball osztály felépítése.	22
5.3.	A BallMovement struktúra felépítése.	22
5.4.	A BallData osztály felépítése.	22
5.5.	A Canny éldetektálás után kapott kép.	26
5.6.	A felismert asztal kontúrja a bináris képen, piros körvonallal keretezve.	27
5.7.	A felismert asztal négy pontból álló körvonala a bináris képen, piros körvonallal keretezve.	29
5.8.	Egy sárga golyó kivágott képére elvégzett mintaillesztések eredményei.	33
5.9.	Egy sárga golyó kivágott képén neurális hálózattal történt becslés eredményei.	35

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 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)[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** nevezzük. 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 pontértékekkel 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 pontértékkal, mivel a játékosok ezt a golyót használják lökéseikhez.

A játék egy menetét **frémnek** nevezik, amely a kezdő lökéstől a fekete golyó elhelyezéséig tart.[1]



2.1. ábra. A golyók kezdeti pozíciója.

2.3. Pontszerzés

A játékosok a pontjaikat a golyók bonyos 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*.[2]

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 nyelvekhez 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 képezzen gépi látáshoz, valamint, hogy egy **egységes infrastruktúrát** biztosítson a fejlesztőknek a területen, ezzel megkönnyítve a programkódok olvashatóságát és terjeszthetőségét. 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 alkalmaságokban. A Tensorflow sokféle funkcióval rendelkezik, ezek közül a következőkben néhányat részletesen meglemlíték.

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) modellekben, 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 "hibát", 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 végezhetőek 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 felhasznállok.

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 játékterület 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.

Az asztalfelismerő lépés kicsit elkülönül a többi lépéstől, hiszen a további lépések különféle bemenetszerzési módszerektől függetlenül, megegyezően hajtódnak végre. Ahhoz hogy a további lépések pontosak legyenek és megfeleő teljesítménnyel működjenekek, az asztalt mindenkor mindenkorának megfelelően kell felismerni, így mindenkorának megfelelően kell elindítani a felismerést.

Ebben a módszerben az elforgatást leszámítva a detektálás, átméretezés 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 módszeréről, a 4.1 képen látható.

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

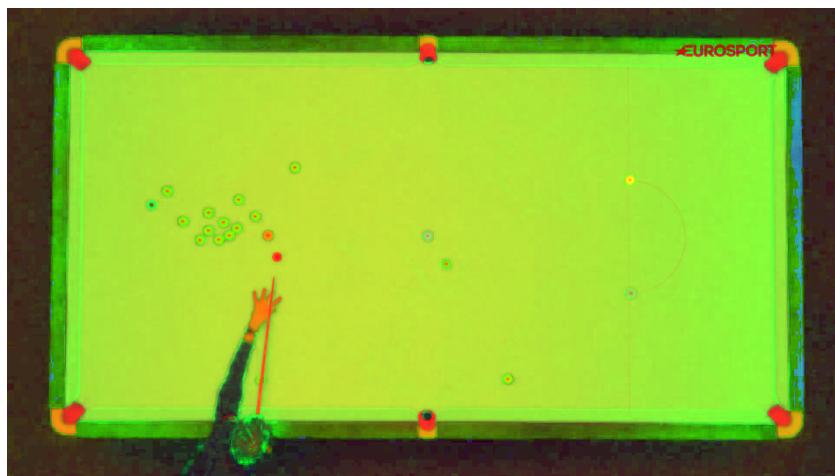
Ez az átalakítás azért fontos, mert HSV formátumban könnyebben intervallumok közé lehet szorítani a játékterület zöld színét. A HSV konverzió belső működéséről részletesebben a megvalósítás részben lesz szó.

A HSV konverzió során kapott intervallumok:

- árnyalat (Hue),
- telítettség (Saturation),



4.1. ábra. Egy nyers bemeneti kép a felvételről.

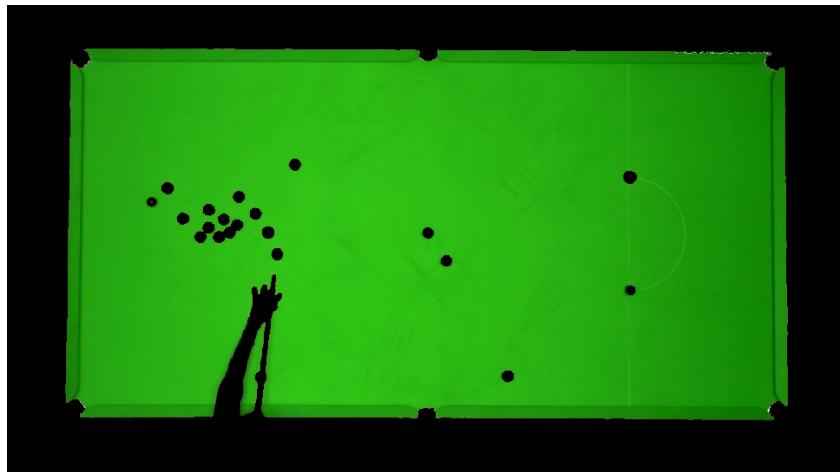


4.2. ábra. A 4.1 kép HSV re alakított verziója RGB reprezentációban.

- 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.

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 kontúrja. 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.



4.3. ábra. A 4.1 kép a maszkolás után.



4.4. ábra. Az eredeti képből kinyert játékterület.

Fent említettem, hogy feltételezhető, hogy a játékterület kontúrja a kontúrkere-sés 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 sebes-ségben és pontosságban. A folyamatok bemenete az előzőekben megismert asztal felismerés kimenete lesz, kimenetük pedig a 4.1 táblázatban látható x és y pozí-ciók, adott golyók színe szerint. A folyamat belső működése módszerenként eltér

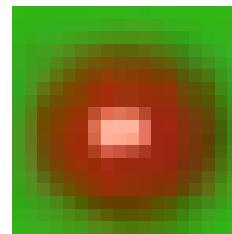
egymástól, ezeket a módszereket az implementáció egyes iterációjának részeként használtam, majd változtattam meg az elért teljesítmény növeléséhez.

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 pozícióból 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.

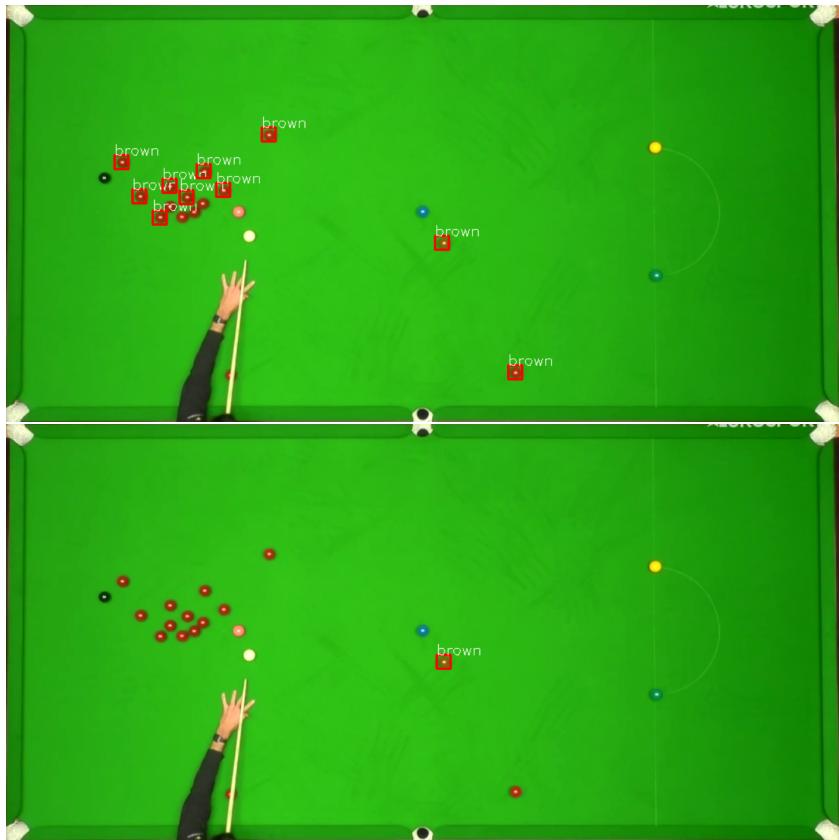


4.6. ábra. A zöld golyó mintaillesztésének hibája (felül) és annak orvoslása HSV konverzióval (alul).

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 megkülönböztetésekor is. Ez a 4.7 képen látható.

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 konvertálás 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 megkönnyíti.



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

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 kördetektáló 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 nagymértekben 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.

Ez a módszer körök megtalálására jól használható, a mintaillesztéshez szükséges képek könnyedén kivághatók az eredeti képből a körök paraméterei alapján. A



4.8. ábra. A Hough transzformáció lefutása után kapott körökök.

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

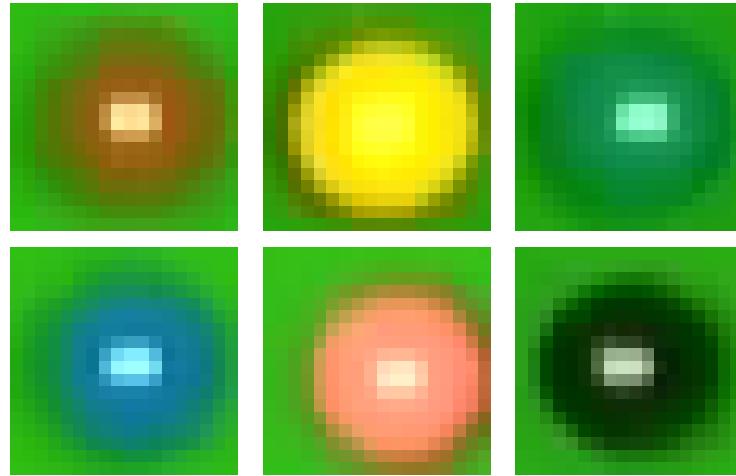
A mintaillesztés hibáinak kiküszöböléséhez, az osztályozás elvégezhető neurális hálózat segítségével. Ez a játékterület egy kerettel való képpontonkénti végigpásztázásával szintén megoldható, azonban körkeresésnél megismert Hough transzformációval jobb teljesítmény érhető el.

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 számkként. 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.



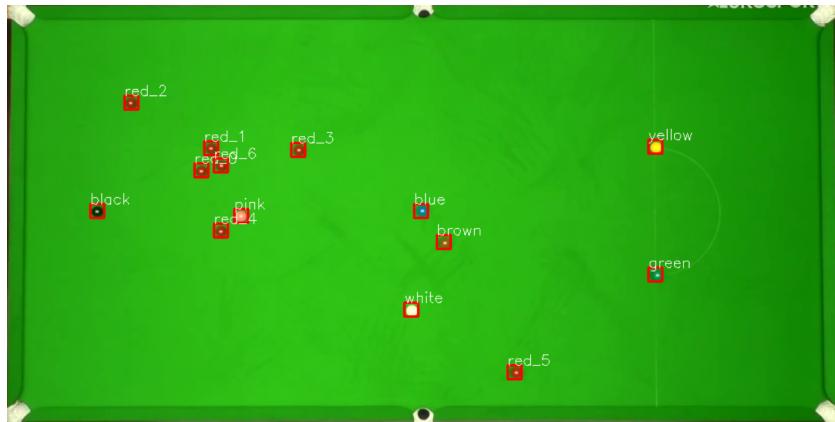
4.9. ábra. Az adatkészlet elemei.

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.

A módszer eredményességének köszönhetően későbbiekben részletesebben ismertetem a megvalósítás fejezetben.



4.10. ábra. A neurális hálózattal való golyófelismerés kimenete.

4.3. A játékmenet elemzése

A golyók felismert pozíciója alapján a játékmenet többféle szempontból vizsgálható, amelyeknek köszönhetően érdekes statisztikai adatokhoz lehet jutni. A vizsgálati szempontok közül ebben a munkában négyfélét fogok megfigyelni, ezek az egyes golyók által **megtett távolság**, adott golyó **pillanatnyi sebessége**, a golyók **megtett útvonala** és a zsebekben elhelyezett golyók **megállapítása**.

Az említett szempontok alapján más komplexebb tényezőket is lehet vizsgálni (pl.: átlagos sebesség, golyók lepattanásának száma stb.), itt viszont az egyszerűség kedvéért csak az alapvetőbb szempontokat vizsgálom.

Ahhoz, hogy megtett utakat és sebességeket lehessen megállapítani, egy folytonos videófelvételre van szükség, a videófelvétel minősége és képkockasebessége is befolyásoló tényezők az adatok pontosságához és megállapításához.

A következő alfejezetekben az egyes vizsgálati szempontokat részletezem, viszont mindenek előtt még egy fontos problémának a megoldását írom le a következő fejezetben.

4.3.1. A piros labdák felcímkézése

A felismert golyók pozíciója és színe nagyszerű kiindulópont az egyes színű golyók útvonalának megállapításához, azonban a piros golyók sajátossága, hogy több is szerepel belőlük a játékterületen. Ez a tulajdonság ahhoz vezet, hogy az egyes golyókat meg kell különböztetni valahogyan egymástól, hogy például azok útvonalát el tudjuk tárolni.

A probléma megoldására létezik egy elméletben viszonylag egyszerű megoldás, amely a következő lépésekkel áll:

- 1. Első felismerés során eltároljuk az egyes piros golyók címkéjét tetszés szerint.

- 2. A következő felismerési folyamat során növekvő sorrendbe helyezzük az előzőleg és jelenleg felismert piros golyók egymástól viszonyított távolságát.
- 3. Az előző lépésben kiszámolt listán sorban végighaladva beállítjuk a jelenlegi golyókhöz a címkét az előző golyók közül a hozzájuk legközelebb esővel.

A fenti módszer úgymond 'mohó' algoritmus, ezzel a számára legkedvezőbb feltételek választja minden esetben, feltételezve, hogy a golyók között a lehető legkisebb az összes elmozdulás. A módszer bizonytalannak tűnhet, azonban megfelelő minőségű videófelvétel mellett elfogadhatóan pontos eredményeket ad. A 4.10 ábrán látható felismert golyók ezen módszer segítségével lettek felcímekézve, és az alkalmazás megvalósításánál is a jelenlegi módszert alkalmazom.

4.3.2. Megtett távolság és útvonal

Ezt a két szempontot egy alfejezetben ismertetem, mert mindenkor megállapítása azonos és egyetlen alapon nyugszik, ez pedig nem más mint szimplán az adott golyók pozíciója.

A teljes megtett távolság könnyen megállapítható a golyók két képkocka közti elmozdulásainak a felhalmozásával. Ahhoz viszont, hogy egy számunkra hétköznapokban is használt mértékegység formájában mutatkozzanak a távolságok, egy apró trükkre lesz szükség. A golyók elmozdulásai kezdetben megállapításkor képpont formájában szerepelnek, a 4.1 alfejezet tábla felismerés részénél tudjuk, hogy a felismeréshez használt asztal egy fix szélességre és magasságra van transzformálva. Ennek következtében megállapítható, hogy ha a transzformációs szélesség 1024 képpont és a szabvány snooker asztal mérete a 2.1 rész alapján 12 láb, avagy 365,8 cm, akkor a valóságban $1 \text{ cm} = \frac{1024}{365,8} \approx 2.8$ képpont formájában mérhető a transzformált képen.

A megtett útvonal megállapításánál nincs szükség mértékegység átváltásokra, hiszen az útvonal csak a felismerés közben a transzformált asztalon kerül kirajzolásra. Az útvonal a egyes képkockákon, adott golyók pozíciójának folytonos listában való eltárolásával lehetséges. A tárolt listák közül egyet a 4.11 ábrán látható módon lehet megjeleníteni a transzformált képen.

4.3.3. A golyó sebessége

Ahhoz, hogy a golyó sebességét meg lehessen állapítani, két tényezőre van szükség, ezek a megtett távolság és az ezen távolság megtételéhez szükséges idő. A megtett távolságot már az előző alfejezetben leírtak alapján megállapítható, tehát csak az eltelt időt kell kiszámolni a sebesség megállapításához. Az eltelt időt a felvétel egy paramétere, a képkockasebesség alapján fogom kiszámolni.



4.11. ábra. A fehér golyó útvonala a felismert képen kék vonallal jelölve.

Ha a videófelvételről tudjuk, hogy a képkockasebessége 30 fps (másodpercenként 30 képkocka), vagy másnéven, két képkocka között eltelt idő $\frac{1}{30} \approx 0.03$ másodperc, és tudjuk, hogy két képkocka között egy adott golyó elmozdulása például 3 képpont, vagy centiméterre átszámolva $1,07 \text{ cm}$, akkor a golyó sebessége $\frac{1,07}{0,03} \approx 35.67 \text{ cm/s}$.

A sebességek értékét egy listában tárolva, azokat a felismerési folyamat végén összegezve, majd leosztva a lista hosszával, kiszámolható a felismerés időtartama alatt egy adott golyóhoz tartozó átlagos sebesség.

4.3.4. Zsebbe helyezés megállapítása

A zsebbe helyezett golyó megállapítása ugyancsak a golyó pozíciójának a segítségével állapítható meg, azonban közre játszik a feilsmerít golyó beazonosításának megszakadása, megszűnése is. A golyó azonosítása akkor szűnik meg, amikor a golyó olyan takarásba vagy árnyékba kerül, hogy annak a felismerését nem lehet többé végrehajtani. Amikor egy golyó felismerése megszűnik, két képkocka között a felismert golyók száma közt eltérés lép fel, ez visszavezethető egy adott golyóra, és annak az utolsó ismert pozíciójára.

Az eltűnt golyó utolsó pozíciójának ismeretében megvizsgálható annak a távolsága a zsebek pozíciójához viszonyítva. A zsebek pozíciója kiszámolható az előzőleg már említett transzformált kép tárolt fix szélesség és magasság értékeivel, hiszen a zsebek a négy sarkon és a hosszanti oldalak felénél helyezkednek el. A zsebek elhelyezkedését a 4.3 táblázat szemlélteti.

Amennyiben a golyó utolsó ismert pozíciója elegendően közel van egy zseb pozíciójához, feltételezhetjük, hogy a golyó az adott zsebben lett elhelyezve.

4.3. táblázat. A zsebek elhelyezkedése szélesség és magasság alapján.

(x, y)	Bal	Közép	Jobb
Alul	(0, 0)	$(\frac{\text{szélesség}}{2}, 0)$	(szélesség, 0)
Felül	(0, magasság)	$(\frac{\text{szélesség}}{2}, \text{magasság})$	(szélesség, magasság)

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>
```

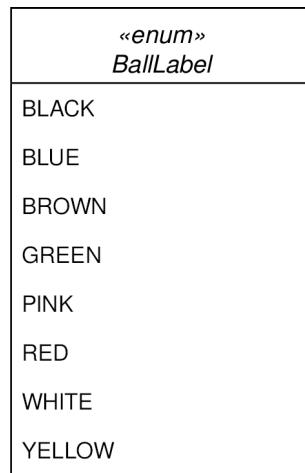
5.1. kódrészlet. Könyvtárak importálása.

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. Egyénileg készített osztályok és struktúrák

Ebben a részletben szeretném ismertetni a további részekben szereplő kódrészletben felbukkanó osztályokat és struktúrákat. Ezeket az objektumokat könnyebb felhasználás és átláthatóság szempontjából készítettem, hogy felgyorsítsam és megkönnyítsem a program olvashatóságát és megérthetőségét.

Az első ilyen objektum egy enum **BallLabel** névvel, ez az enum a golyók lehetséges színét tárolja. Enum formájában megelőzhető sztringek és szimpla egész számok használata, amelyek hosszú távon hibákhoz és bonyodalmakhoz vezethetnek. Az enum felépítése a 5.1 ábrán látható.

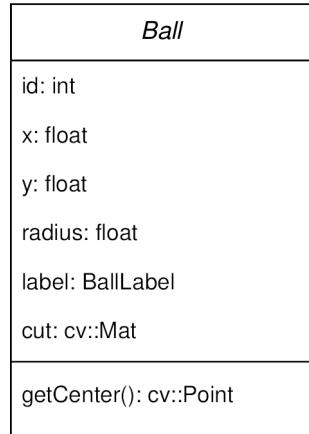


5.1. ábra. A **BallLabel** enum felépítése.

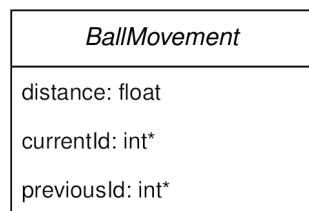
A következő osztály a játék során használt golyók reprezentálására szolgál, ez az osztály a **Ball** nevet kapta. Az osztály tárolja egy adott golyó elhelyezkedését (`x`, `y`), sugarát (`radius`), a címkéjét és azonosítóját (`label`, `id`), továbbá cut néven egy képet amely a golyó asztalról kivágott képét tárolja. A `getCenter` függvény a golyó pozícióját adja vissza egy `cv::Point` típus[8] formájában, amely egy koordináta pontot ad meg. Az osztály felépítése a 5.2 ábrán látható.

Kifejezetten a piros golyók felcímkézéséhez készült az alábbi 5.3 ábrán látható **BallMovement** struktúra, ez tárolja a piros golyók összehasonlításához szükséges távolságot (`distance`) és azonosítókat

A **BallMovement** osztályhoz hasonlóan létezik egy ún. **BallData** osztály, amely arra szolgál, hogy a golyókon megvizsgált vizsgálati szempontokat eltárolja és kiszámolja. Ilyen szempontok a 4.3 részben említett elemzési szempontok, ezek közül a teljes megtett távolságot a `totalDistance`, az útvonalat a `path` és a pillanatnyi sebességeket a `speed` változók tárolják. Az osztály része még a `previousUpdateFrame` változó, ez eltárolja az adott golyóhoz a legutóbbi frissítés időpontját képkocka formájában, amelyet az `addPosition` függvény felhasznál a két frissítés között eltelt idő

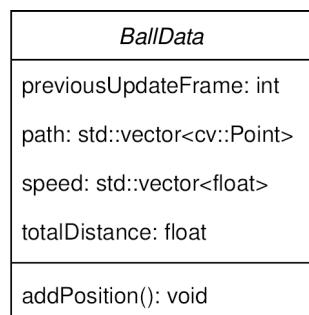


5.2. ábra. A **Ball** osztály felépítése.



5.3. ábra. A **BallMovement** struktúra felépítése.

megállapítására, továbbá az előzőekben említett változókat is beállítja vagy frissíti. A **BallData** osztály felépítése a 5.4 ábrán látható.



5.4. ábra. A **BallData** osztály felépítése.

5.4. Az asztal kontúrjának megkeresése

A nyers képből a játékterület megszerzéséhez azt először be kell tölteni egy több-dimenziós 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 cv::cvtColor(image, hsv, cv::COLOR_BGR2HSV);
2
3 cv::Scalar lowerGreen = {50, 50, 70};
4 cv::Scalar upperGreen = {65, 255, 255j};
5
6 cv::inRange(hsv, lowerGreen, upperGreen, mask);
7 cv::bitwise_and(image, image, result, mask);

```

5.2. kódrészlet. A játékterület maszkolása.

A 5.2 kódrészletben az `image` a bemeneti kép, amelyet a `cv::cvtColor` függvényel [8] konvertálok át HSV formátumra. Ennek a függvénynek az első paramétere a bemeneti kép, a második a kimeneti kép változója, a harmadik pedig a konverziótípusa, 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 színkomponenst, $\min(R, G, B)$ 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 `lowerGreen` és `upperGreen` változók az alsó és felső intenzitási határokat jelölik sorrendnek megfelelően. A maszk elkészítését a `cv::inRange` függvényel [8] végezzük el, itt a paraméterek sorban a HSV-re konvertált kép, az alsó és felső intenzitás értékek, valamint a kimeneti maszk változója. 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 mindenhol 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 `cv::bitwise_and` függvény [8] segítségével. Itt a paraméterek a bejövő eredeti kép `image` kétszer, a kimeneti kép és a maszk `mask`.

A folyamat során a metódus a 5.5 egyenlet szerint jár el,

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

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 közt 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 közt 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 cv::cvtColor(result, imageGray, cv::COLOR_BGR2GRAY);
2
3 cv::Canny(imageGray, edges, 200, 100);
```

5.3. kódrészlet. Szürkeárnyalatolás és éldetektálás.

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

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

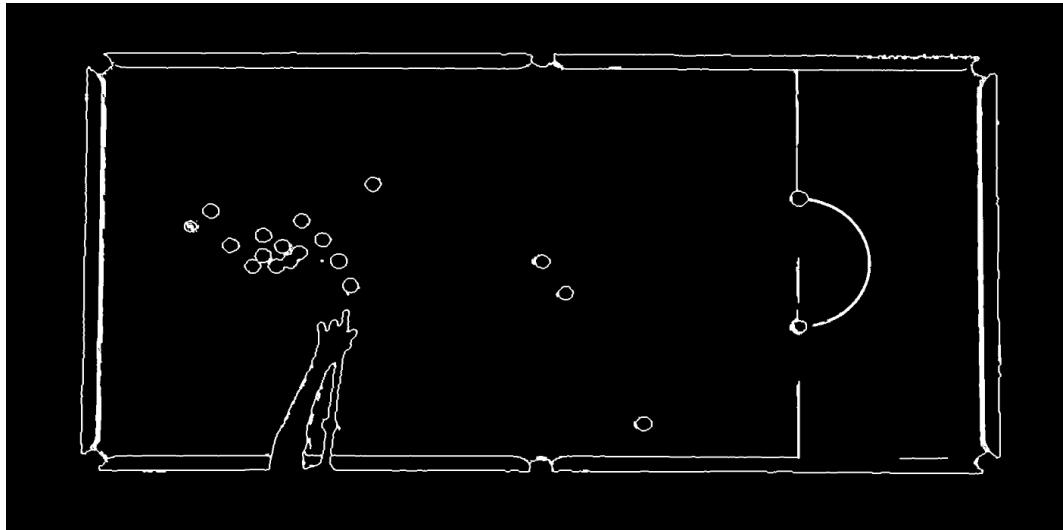
- homályosítás Gauss szűrővel [10] a zajcsökkentés érdekében,
- élek helyének és irányának megállapítása intenzitás-gradiensből,
- nem-maximum vágás merőleges élek szűréshéhez,
- kettős küszöbölés élek szűréséhez.

Az éldetektálásnál meg kell adni a függvénynek a szürkeárnyalatos képet (`imageGray`), a kimeneti képet, továbbá két küszöbértéket, 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 5.5 ábrán látható.

A következő lépésben a bináris képen lefuttatásra kerül egy kontúrkereső algoritmus [11], majd a kapott kontúroknak vesszem a konvex körvonalaát, azok egyszerűsítése, esetleges konkáv alakzatok megszüntetése érdekében. Ezek után feltételezve, hogy a kontúrok közül a legnagyobb a játkerület, az kiválasztható a körvonalaik közül.

```
1 cv::findContours(edges, contours, cv::RETR_LIST, cv::CHAIN_APPROX_SIMPLE);
2
3 for(auto& contour : contours) {
4     std::vector<cv::Point> hull;
5     cv::convexHull(contour, hull);
6     contour = hull;
7 }
8
9 std::sort(contours.begin(), contours.end(), areaComparator);
```

5.4. kódrészlet. Kontúrok keresése.



5.5. ábra. A Canny éldetektálás után kapott kép.

```

1 bool areaComparator(const std::vector<cv::Point>& lhs, const std::vector<cv::Point>& rhs) {
2     return cv::contourArea(lhs) > cv::contourArea(rhs);
3 }
```

5.5. kódrészlet.

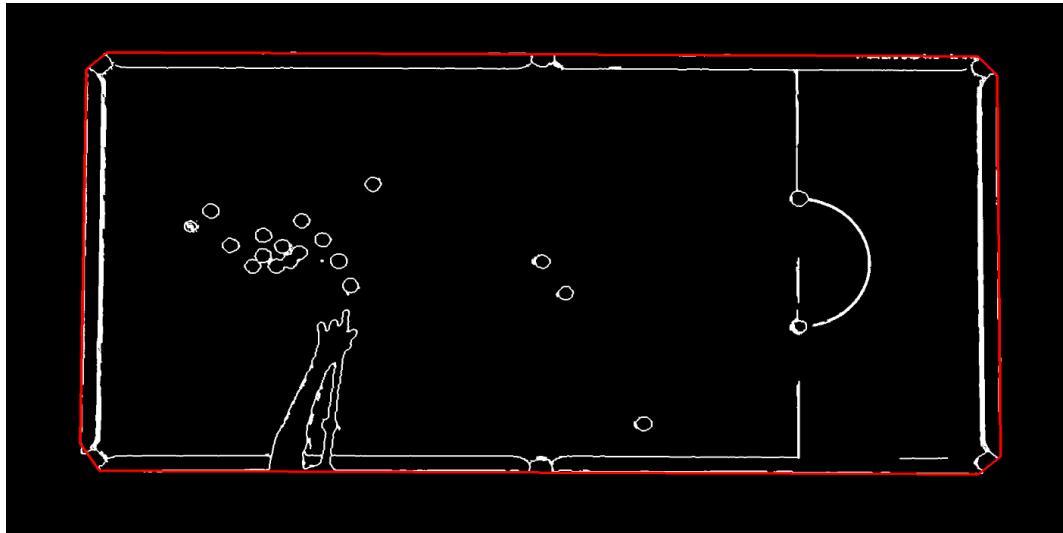
Sorba rendezéshez használt segédfüggvény.

A 5.4 kódrészletben található `cv::findContours` függvény [8, 11] egy határkövetéses algoritmussal kigyűjti a kontúrokat. Ezek a kontúrok a képen található képpont koordináták láncolatából állnak össze. A kontúrok körvonala a `cv::convexHull` függvény [8, 12] segítségével kapható meg. Ez az algoritmus a kontúrok koordinátáinak láncolatát használja, majd a kontúrt egy konvex körönallal határolja, ugyancsak koordináták láncolatai formájában reprezentálva.

A fenti művelet elsőre feleslegesnek tűnhet, hiszen a keresett asztal kontúrja előreláthatólag nem konkáv, a művelet elvégzése mégis fontos, hiszen így egyszerűsíthető az alakzat (kontúr koordináta láncolat pontjainak csökkentése), ezzel a folytatónagyságos műveleteket felgyorsítva.

A legnagyobb kontúr kiválasztásához tudni kell az egyes kontúrok területét. A területet a `cv::contourArea` függvény [8] lehet kiszámolni. Ez megtehető minden eddigi kontúr esetében függvénynek való paraméterkénti átadással. A kiszámolt területek közül a legnagyobbat kiválasztva, annak kontúr koordináta láncolata eltárolásra kerül. A sorba rendezés a 5.4 kódrészlet utolsó sorában látható függvényel történik, itt az `areaComparator` egy segédfüggvény, amely az előzőleg említett `cv::contourArea` metódust [8] használja a területek összehasonlításához és rendezéséhez. A segédfüggvény a 5.5 kódrészletben látható. A kapott kontúr kirajzolva a 5.6 ábrán látható.

A `cv::contourArea` függvény a Surveyor's Area algoritmust [13] használja az alakzatok területének számolásához. Ez az algoritmus a Green-tétel egy speciális



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

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 5.6 ábrán látható, hogy a kontúr téglalaphoz hasonló alajkának 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 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 5.7 képlet alapján megvégbe,

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

ahol D a kiszámolt pontok közti távolság, (x_a, y_a) és (x_b, y_b) pedig a két koordináta, amelyek között 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 négyzet kontúr később hibás lehet.

A metszéspontok kiszámolásához a következő képleteket [14] használtam,

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

$$P_x = \frac{(x_1y_2 - y_1x_2)(x_3 - x_4) - (x_3y_4 - y_3x_4)(x_1 - x_2)}{D} \quad (5.9)$$

$$P_y = \frac{(x_1y_2 - y_1x_2)(y_3 - y_4) - (x_3y_4 - y_3x_4)(y_1 - y_2)}{D} \quad (5.10)$$

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 5.6 kódrészletben látható,

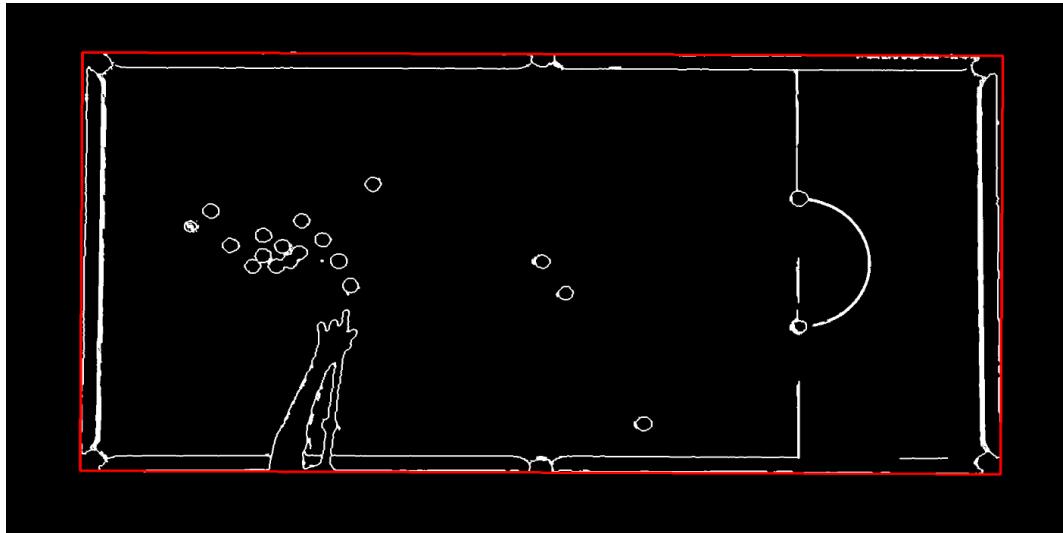
```

1  bool parallel = false;
2  cv::Point p1, p2, p3, p4, intersection;
3  float d, t1, t2;
4
5  d = (p1.x - p2.x) * (p3.y - p4.y) - (p1.y - p2.y) * (p3.x - p4.x);
6
7  if (std::abs(d) < 1e-8) {
8      parallel = true;
9  }
10
11 if (!parallel) {
12     t1 = (p1.x*p2.y - p1.y*p2.x) * (p3.x - p4.x) - (p3.x*p4.y - p3.y*p4.x) * (p1.x - p2.x);
13     t2 = (p1.x*p2.y - p1.y*p2.x) * (p3.y - p4.y) - (p3.x*p4.y - p3.y*p4.x) * (p1.y - p2.y);
14
15     intersection = cv::Point(t1 / d, t2 / d);
16 }
```

5.6. kódrészlet. Metszéspont kereső algoritmus.

ahol $p1, p2, p3, p4$ a fent megismert négy koordináta, d a kiszámolt nevező, $t1$ és $t2$ pedig segédváltozók a számlálók tárolásához. A kódrészlet 7. - 9. soraiban látható, hogy abban az esetben ha d nagyon kicsi, a metszéspont nem lesz kiszámolva. Ez azért van, mert a 5.8 függvényben kiszámolt nevező, $D = 0$ esetén a két egyenes párhuzamos, és ilyenkor nincs metszéspont.

A folyamat végeredményképp kapott kép a 5.7 ábrán látható.



5.7. ábra. A felismert asztal négy pontból álló körvonala a bináris képen, piros körvonallal keretezve.

5.5. 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 beszélni.

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 5.7 kódot használom,

```

1 std::vector<int> sums, diffs;
2
3 for (auto& point : quad) {
4     sums.push_back(point.x + point.y);
5     diffs.push_back(point.y - point.x);
6 }
7
8 std::vector<cv::Point2f> src;
9 src.push_back(quad[std::min_element(sums.begin(), sums.end()) - sums.begin()]);
10 src.push_back(quad[std::min_element(diffs.begin(), diffs.end()) - diffs.begin()]);
11 src.push_back(quad[std::max_element(sums.begin(), sums.end()) - sums.begin()]);
12 src.push_back(quad[std::max_element(diffs.begin(), diffs.end()) - diffs.begin()]);

```

5.7. kódrészlet. Átrendező algoritmus.

ahol az előzőleg kiszámolt négy metszéspontot felhasználva, ahhoz hogy meg tudjam állapítani a azok 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 9. - 11. soraiban láthatóak.

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

A célértékek a 5.8 kód soraival adhatóak meg,

```
1 std::vector<cv::Point2f> dst;
2
3 int width = 1024 - 1;
4 int height = 512 - 1;
5
6 dst.push_back(cv::Point(0, 0));
7 dst.push_back(cv::Point(width, 0));
8 dst.push_back(cv::Point(width, height));
9 dst.push_back(cv::Point(0, height));
```

5.8. kódrészlet. A kimeneti értékek megadása.

itt a cél kép méretei egy érték páros formájában szerepelnek 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 mindenek után már csak a transzformációs mátrix elkészítésére van szükség, majd a transzformáció végrehajtására.

Ezek a 5.9 kódrészlettel hajthatóak végre,

```
1 cv::Mat M = cv::getPerspectiveTransform(src, dst);
2 cv::warpPerspective(image, warp, M, cv::Size(width, height));
```

5.9. kódrészlet. A transzformáció végrehajtása.

itt `M` a transzformációs mátrix, amely a `cv::getPerspectiveTransform` függvénytel [8] kapható meg a forrás és célértékek megadásával. A függvény Gauss-elimináció [15] segítségével számol ki egy 3×3 méretű mátrixot, amelyet a `cv::warpPerspective` függvénytel [8] alkalmazok az `image` változóban tárolt képre az `M` mátrix és `cv::Size(width, height)` méret megadásával. A transzformáló függvény lineáris interpolációt [16] használ alapértelmezett esetben az intenzitás értékek meghatározásához, a torzított kép a `warp` változóban kerül tárolásra.

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

5.6. 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 körvonal élének egy tetszőleges x_i, y_i pontja átalakításra kerül egy a, b, r paraméterek által meghatározott térben 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 5.10 kódrészletben látható.

```
1 cv::cvtColor(image, gray, cv::COLOR_BGR2GRAY);
2
3 float minRadius = 5;
4 float maxRadius = 12;
5 float minDistance = 12;
6 int circleThreshold = 40;
7 int circlePerfectness = 10;
8
9 std::vector<cv::Vec3f> circles;
10 cv::HoughCircles(
11     gray,
12     circles,
13     cv::HOUGH_GRADIENT,
14     1,
15     minDistance,
16     circleThreshold,
17     circlePerfectness,
18     minRadius,
19     maxRadius);
```

5.10. kódrészlet. A körkereső algoritmus.

A kódrészletben a Hough transzformációt a `cv::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 `cvtColor` függvényel[8] megy végbe.

A második paraméter a megtalált körök listáját tartalmazza, a harmadik az előzőleg megismert Hough transzformációs módszert[8, 18, 17] adja meg a `cv::HOUGH_GRADIENT` kulcsszóval, továbbá a negyedik paraméter a folyamathoz felhasznált képet skálázza. A skálázás az eredeti kép felbontást 1 értékkel nem változtatja,

2 értékkal 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` (minimális távolság) paraméterek megadják a keresett körök tulajdonságait, így növelhető az algoritmus teljesítménye és csökkenthető a duplikációk előfordulása.

A `circleThreshold` paraméter az algoritmus kezdeti feldolgozó lépéseként végrehajtott Canny éldetektálás felső paraméterét adja meg, az alsó paraméter ennek felével lesz egyenlő. Ez a Canny éldetektálás a 5.4 alfejezetben megismert módszerrel megegyezően hajtódiák végre. Végül pedig, a `circlePerfectness` névvel ellátott paraméter a körkeresés pontosságát szabályozza, minél kisebb az érték, annál kisebb a pontosság, amely több hamisan felismert kört eredményezhet.

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

5.7. A detektált körök osztályozása

5.7.1. Osztályozás mintaillesztéssel

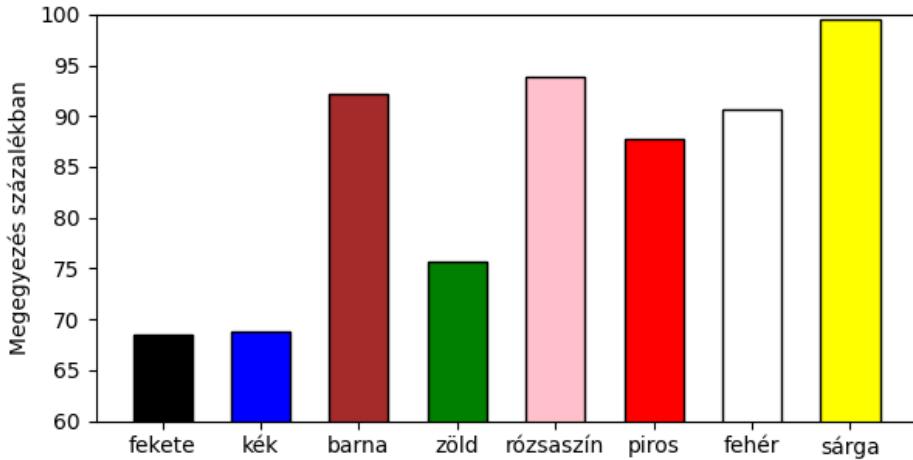
A mintaillesztés ún. Kereszt Korrelációval (Normed Cross Correlation) megy végbe, ennek a működése a 5.12 képleten alapul[4, 8].

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

A képleben 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. Ez a folyamat egy kernelhez hasonlóan végigpásztázza a képet, majd a kapott érték mátrixból eldönthető, hogy mely pontokon volt a legnagyobb egyezés a mintával.

Ebben az esetben viszont nem a teljes asztal képén vizsgálom a minta egyezését, hanem csak egy adott felismert körhöz tartozó metszeten. Ilyenkor a minta és a kép mérete megegyezik, ezért a kapott mártrixból a legnagyobb érték fogja reprezentálni az egyezőség mértékét. Ahhoz, hogy egy adott körnek meg lehessen állapítani a színét, az összes mintával hasonítani kell, majd a művelet végén a legnagyobb értékkel rendelkező illesztéshez tartozó minta színe fogja megadni a felismert kör színét. A 4.9 ábrán látható sárga golyóhoz hasonló adatra elvégzett mintaillesztés eredményeit a 5.8 ábra mutatja.

Az ábrán látható, hogy a mintaillesztés nehezen különbözteti meg az egyes színeket, szinte minden színhez 65% -nál nagyobb megegyezési értékekkel ad. Ez



5.8. ábra. Egy sárga golyó kivágott képére elvégzett mintaillesztések eredményei.

jól szemlélteti, hogy miért nem kerül közvetlen felhasználásra az alkalmazásban, azomban a adatkészlet készítéséhez felhasználható.

A mintaillesztés az alkalmazásban a 5.11 kód részlet formájában kerül felhasználásra.

```

1 if (hsvMode) {
2     cv::Mat templateHSV;
3     cv::cvtColor(templateBGR, templateHSV, cv::COLOR_BGR2HSV);
4     cv::matchTemplate(cutImageHSV, templateHSV, result, cv::TM_CCORR_NORMED);
5 }
6 else {
7     cv::matchTemplate(cutImageBGR, templateBGR, result, cv::TM_CCORR_NORMED);
8 }
9
10 double maxValue;
11 cv::minMaxLoc(result, NULL, &maxValue);

```

5.11. kód részlet. A mintaillesztés menete.

Itt a hsvMode logikai változó megadja, hogy a mintaillesztést BGR vagy HSV színinformátumban végezze el a kód. A HSV módban történő illesztésnél szimplán átkonvertálom a mintát HSV formátumra a már jól ismert cv::cvtColor függvénytel[8], majd minden a két módban úgyanúgy végrehajtom a mintaillesztést a cv::matchTemplate függvénytel[4, 8]. A függvény a 5.12 egyenlet alapján számolja ki a minta egyezőségét egy adott pontban. Paraméterként meg kell adni neki a képet amelyre a mintát szeretnénk illeszteni (cutImageBGR vagy cutImageHSV), a mintát (templateHSV vagy templateBGR), a kimeneti változónkat, amely ebben az esetben a result mátrix, továbbá meg kell adnunk a mintaillesztés algoritmust, ezt a cv::TM_CCORR_NORMED konstans teszi meg.

A kapott mátrixból az egyezőséget reprezentáló értéket a `cv::minMaxLoc` függvény[8] adja meg, ez a függvény kiszámolja, hogy a bemeneti mátrixon (`result`) hol található és mennyi a minimum és maximum érték. Az egyezőség megállapításához ezesetben csak a maximum értékre van szükség, ezt a függvény harmadik paramétere adja meg `maxValue` néven.

A fenti folyamat egy körre írja le a minta illesztését, ezt ciklusban az alkalmazás végrehajtja minden potenciális goylót leíró körre.

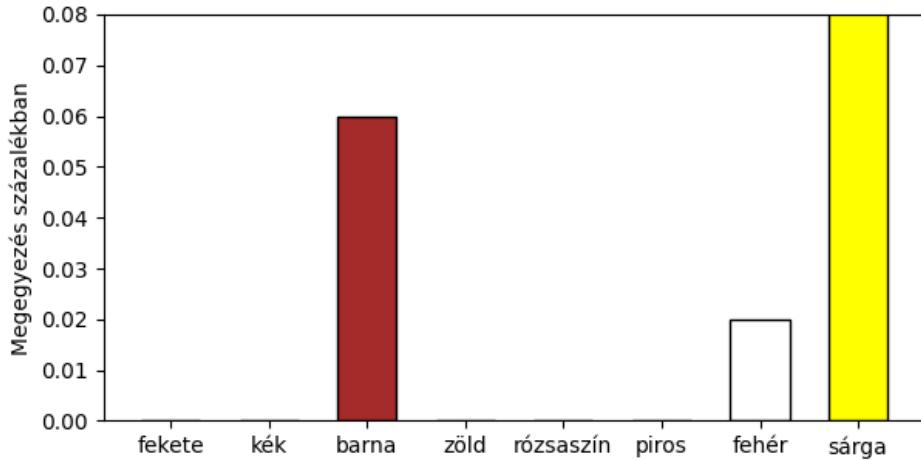
5.7.2. Osztályozás neurális hálózattal

A neurális hálózattal való osztályozás hasonlóan megy végbe, mint az előző alfejezetben megismert mintaillesztes. Az osztályozáshoz szükség van egy betanított neurális hálózatra, amit be kell tölteni az alkalmazásba, majd továbbítani neki a kör kivágott képet. A kivágott képet átadás előtt viszont módosítani kell, mégpedig úgy, hogy megegyezzen a neurális háló kívánt bemenetével.

Az általam használt modell bemenete egy olyan képet vár, amelynek hat intenzitásértéke van, azonban egy átlagos kivágott kép csupán három ilyen értékkel rendelkezik, ezek a BGR által reprezentált kék (blue), zöld (green) és piros (red) intenzitások, a másik három szükséges intenzitást a kép HSV értékei adják meg, ezek az árnyalat (hue), telítettség (saturation) és érték (value).

A kép intenzitásait összefűzve, majd azt továbbítva a hálózatnak, a kapott érték egy normalizált értékek listája, ami megadja, hogy egyes színek mennyire jellemzik a bemeneti képet. A 5.8 ábrához hasonlóan látható egy sárga golyó neurális hálózatos osztályozásának eredménye a 5.9 ábrán. Az ábrán látható, hogy míg a sárga érték nem mutatható teljesen az ábra intervallumán, hisz megegyezése közel 100% -os (99,46%), addig a többi színhez tartozó megegyezési értékek 0.1% alá esnek. Ez is mutatja, hogy a módszer rendkívül pontosan meg tudja jósolni egy potenciálisan felismert golyó színét.

A felismerés végrehajtásához használt kód a 5.12 kódrészletben látható.



5.9. ábra. Egy sárga golyó kivágott képén neurális hálózattal történt becslés eredményei.

```

1 std::vector<cv::Mat> channels = {cutImageBGR, cutImageHSV};
2 cv::Mat mixed;
3 cv::merge(channels, mixed);
4
5 fdeep::tensor input = fdeep::tensor_from_bytes(
6     mixed.ptr(),
7     static_cast<std::size_t>(mixed.rows),
8     static_cast<std::size_t>(mixed.cols),
9     static_cast<std::size_t>(mixed.channels()),
10    0.0f,
11    1.0f
12 );
13
14 std::vector<float> result = model.predict(input).to_vector();

```

5.12. kód részlet.

A neurális hálóval történő osztályozás menete.

A 5.12 kód részletben elsősorban a kép intenzitás érékeit fűzzük össze, ezt a `cv::merge` függvény[8] teszi meg, ami bemenetként kéri a két képet (BGR és HSV) egy tömb formájában, a kimenetet pedig a `mixed` változóba teszi bele.

A hat színcsatornával rendelkező `mixed` mátrixot ezekután egy ún. tensor típusra kell átalakítani, amelyet a neurális hálózat közvetlenül kezelni tud. Az átalakításhoz a `fdeep::tensor_from_bytes` függvény[19] szükséges, amelynek paraméterként át kell adni a mátrixunk memóriacímének mutatóját (`mixed.ptr()`), ez alapján tudja megállapítani a függvény az adatok helyzetét a memóriában, továbbá még át kell adni a mátrix szélességét (`mixed.cols`), magasságát (`mixed.rows`) és a színcsatornák számát (`mixed.channels`). Az utolsó két paraméter megadja, hogy az intenzitás értékek 0 és 1 értékek közé legyenek skálázva. Ez egy 0 és 255 értékek között mozgó intenzitásnál, például $\frac{130}{255} \approx 0.51$ értéket jelent.

A tensor-ra alakítás után a `model.predict` függvény[19] lehet a hálózat jóslását elvégezni, amely kimenete a `result` változóba kerül. A kimenet egy lebegőpontos értékek tömbje lesz, ennek elemei adják meg, hogy egyes színek mennyire jellemzik az osztályozott képet. A legnagyobb tömb elemhez tartozó index jelöli a jósolt színt, hasonlóan az előző 5.7.1 alfejezetben megismert mintaillesztéshez.

5.8. Játékmenet vizsgálati szempontok

5.8.1. A piros labdák megkülönböztetése

A piros labdák megkülönböztetéséhez azoknak a szín mellett adni kell egy másodlagos azonosító paramétert, ez az alkalmazásban azonosító számok formájában történik, a számok 0 és 14 (15db) között helyezkedhetnek el. A piros golyók beazonosításához a 4.3.1 részben ismertetett módszereket fogom használni, amelyhez először kiszámolom minden a frissen felismert és az előzőleg felismert piros golyók közt a távolságot. A távolság kiszámolásához használt kód a 5.13 kódrészletet használom.

```

1 std::vector<BallMovement> movements;
2
3 for (Ball& ball : balls) {
4     for (Ball& previousBall : previousBalls) {
5         if (ball.label == BallLabel::RED && previousBall.label == BallLabel::RED) {
6             float d = cv::norm(ball.getCenter() - previousBall.getCenter());
7             movements.push_back({d, &ball.id, &previousBall.id});
8         }
9     }
10 }
```

5.13. kódrészlet. Piros golyók közti távolság kiszámolása.

A kódrészletben a már előzőleg megismert egyenlet 5.7 alapján a `cv::norm` függvény[8] kiszámolja a két golyó középpontja közti távolságot, majd beleteszi azt a `d` változóba. A következő lépés hozzáadja egy `BallMovement` típusú `movements` néven használt tömb (bővebben 5.3 rész) elemeihez az imént kiszámolt távolságot, továbbá a jelenlegi (`ball.id`) és előző (`previousBall.id`) golyók azonosítóját memóriacímük szerint. A memóriacím szerinti átadás azért fontos, mert az érék megváltoztatásakor a változott értéket azonnal el tudjuk érni bárhonnan a memóriacím segítségével, hisz ilyenkor a memóriacím nem változik, csak a változó értéke.

A tömb elkészítése után az algoritmus további részei előtt sorba kell rendezni az értékeket távolságok szerint. A sorba rendezést a 5.14 kódrészlet mutatja, a kódrészletben használt összehasonlító segédfüggvényt pedig a 5.15 kódrészlet ábrázolja.

```
1 std::sort(movements.begin(), movements.end(), distanceComparator);
```

5.14. kódrészlet. Piros golyók közti távolságok sorba rendezése.

```

1 bool distanceComparator(const BallMovement& lhs, const BallMovement& rhs) {
2     return lhs.distance < rhs.distance;
3 }
```

5.15. kódrészlet.

A 5.14 kódrészletben használt összehasonlító segédfüggvény.

Az összehasonlító segédfüggvény a BallMovement osztály distance paramétert használja összehasonlítási alapnak.

A következő lépés a folyamat legkomplexebb része, ez a rész legfőképp egy ciklusból áll, amivel végighalad az algoritmus a sorba rendezett movements tömbön és megvizsgálja, hogy az adott elemhez tartozó azonosítók renezésre kerültek-e. A ciklus és vizsgálat rész a 5.16 kódrészletben látható.

```

1 std::vector<int*> setPrevious;
2 std::vector<int*> setCurrent;
3 for (const auto& movement : movements) {
4     if (std::count(setPrevious.begin(), setPrevious.end(), movement.previousId) <= 0 &&
5         std::count(setCurrent.begin(), setCurrent.end(), movement.currentId) <= 0) {
6         ...
7     }
8 }
```

5.16. kódrészlet.

A 5.14 kódrészletben használt összehasonlító segédfüggvény.

A ciklus kezdete előtt elhelyezkedő setPrevious és setCurrent változók tárolják azon golyók azonosítójának listáját amelyek már beállításra kerültek a folyamat során. A beállítottság állapotának vizsgálata az if állításban látható, itt az std::count függvény segítségével ellenőrizhető, hogy a ciklus adott eleméhez (movement) tartozó azonosítók közül valamelyik megtalálható-e a beállított azonosítók közt. Amennyiben nem található meg, lefut az algoritmus belső része, ez a 5.17 kódrészletben látható.

```

1 for (int i = 0; i < movements.size(); i++) {
2     if (*movements[i].currentId == *movement.previousId) {
3         *movements[i].currentId = *movement.currentId;
4     }
5 }
6
7 *movement.currentId = *movement.previousId;
8
9 setPrevious.push_back(movement.previousId);
10 setCurrent.push_back(movement.currentId);
```

5.17. kódrészlet.

A piros golyók azonosítójának beállítása.

Itt egyértelműnek tűnhet, mivel feltételezhetően az elmozdulás ugyanazon golyót írja le két különböző időpillanatban, hogy szimplán át kell állítani a jelenlegi azonosítót (movement.currentId) az előző azonosítóra (movement.previousId), viszont az elmozdulások listájában szerepelhetnek még olyan elemek, amelyeknél a jelenlegi azonosító (movements[i].currentId) megegyezik az általunk beállítani

kívánt azonosítóval (`movement.previousId`), ezért le kell ellenőrizni, hogy az elmozdulások közt létezik-e ilyen elem, majd annak azonosítóját kicserélni a vizsgált elemünk jelenlegi azonosítójával. Ez az ellenőrzés a kód 1. - 5. soraiban történik egy ciklus formájában, majd a 7. sorban kerül beállításra az adott elem jelenlegi azonosítója.

Az előző lépés bonyolultnak tűnhet, viszont lényegében egy cserét hajt végre az algoritmus az előzőleg és a jelenleg meghatározott azonosítóval rendelkező elemek között. A folyamat végeztével mind a jelenlegi, mind az előző azonosító bekerül a számukra megfelelő ellenőrzött elemek listájába a 9. és 10. sorban látható módon.

5.8.2. Az elemzési szempontok kiszámolása

A távolságok, útvonalak és pillanatnyi sebesség kiszámolását a 5.3 részben ismertetett BallLabel osztály `addPosition` függvénye (bővebben 5.3 rész) végzi el, az útvonalakat a 5.18 kódrészletben látható módon számolja ki.

```

1 void addPosition(const cv::Point& newPosition, const int& framePosition) {
2     if (path.empty() || speed.empty()) {
3         speed.push_back(0);
4         path.push_back(newPosition);
5     }
6     else {
7         if (framePosition <= previousUpdateFrame) {
8             return;
9         }
10        float distance = cv::norm(newPosition - path.back());
11        float time = framePosition - previousUpdateFrame;
12
13        totalDistance += distance;
14        speed.push_back(distance / time);
15        path.push_back(newPosition);
16    }
17
18    previousUpdateFrame = framePosition;
19}

```

5.18. kódrészlet. Egy golyó megtett távolságának, útvonalának és sebességének kiszámolása.

Az `addPosition` függvény bemenetként megkapja a frissített golyó pozíóját (`newPosition`) és a feldolgozott videó pillanatnyi képkockáját (`framePosition`). A belső részben található egy `if` állítás, itt ellenőrzésre kerül, hogy az adott golyóhoz tartozó adatok rendelkeznek-e értékkal, amennyiben nem, kezdő értékek lesznek hozzáadva minden a `speed`, minden a `path` tömbökhöz, ez a 3. és 4. sorokban látható. Abban az esetben amikor már léteznek a kezdeti elemek, az `else` ágban első sorokban (7. - 9.) ellenőrzésre kerül, hogy a jelenlegi képkocka megegyezik-e a következő képkocká-

val, ha megegyezik, akkor a függvény automatikusan visszatér új elemek hozzáadása nélkül, ha nem egyezik meg, akkor elkezdődik az egyes szempontok kiszámolása.

Első sorban a távolság kerül kiszámolásra, ez már az előzőekben megismert cv ::norm függvényel[8] történik meg, amely a 5.7 egyenlet alapján kiszámolja a két pont távolságát, ezek a newPosition és path.back() amelyek sorra az új hozzáadott pozíció és a path lista utolsóként beállított pozíóját adják meg. A kiszámolt távolság a distance változóba kerül tárolásra, amely értékkel később növelődik a totalDistance változó, ennek értéke megadja a teljes megtett távolságot képpont formájában.

A distance értékének kiszámolása után az eltelt idő nagysága az eddig tárolt (previousUpdateFrame) és a jelenlegi (framePosition) képpontokat tároló változók különbségéből adódik. A time változó ezt az értéket képpontok formájában tárolja, majd később a sebesség kiszámolásához a távolság osztásra kerül az eltelt idővel, amelynek eredménye a speed tömbhöz hozzáadódik. Ez a művelet a kódrészlet 14. sorában látható. A sebességek tömbjét növelő sor után a kódban következik az eltárolt útvonalak növelése, amely a 15. sorban a path tömb newPosition értékkel való növelését teszi meg.

A **lelökött golyók észlelése** a kód egy másik részében található, ahol egy ciklusban végighalad az algoritmus az előző képkockán felismert golyókon, ha egy adott golyót nem talál meg a jelenlegi képkockán, feltételezi, hogy az egy zsebbe került. Az eltűnt golyók ellenőrzéséért felelős kód a 5.19 kódrészletben található.

```
1 for (const Ball& previousBall : previousBalls) {
2     bool possiblyScored = true;
3
4     for (const Ball& ball : balls) {
5         if ((ball.id == previousBall.id) && (ball.label == previousBall.label)) {
6             possiblyScored = false;
7             break;
8         }
9     }
10    ...
11 }
```

5.19. kódrészlet. Golyók potenciális lelökésének vizsgálata.

Itt az előzőleg felismert golyók a previousBalls a jelenleg felismert golyók a balls tömbben vannak, a folyamat a possiblyScored változó beállításával törénik, ez a változó jelzi, hogy olyan golyóról beszélhetünk-e amely potenciálisan lett lökve. Az imént említett logikai változó kezdetben igaz értéket vesz fel, majd amennyiben egy előzőleg vizsgált golyó (previousBall) azonosítója és címkeje létezik a frissen felismert golyók (ball) közt, akkor feltehetőleg nem került zsebbe, és ezért hamisra állítódik a possiblyScored változó.

Abban az esetben amikor a possiblyScored változó nem lesz hamisra állítva, további lépésekben meg kell vizsgálni a golyót, hogy melyik zsebbe lett elhelyezve, ugyanis ha minden zsebtől távol helyezkedik el a golyó utolsó felismert pontja, akkor az algoritmus nem számol pontot a golyó elhelyezéséért. A zsebeket ellenőrző algoritmus kódja a 5.20 kódrészletben látható.

```

1 if (possiblyScored) {
2     std::vector<cv::Point> pockets = {
3         {0, 0},           {0, width / 2},           {0, width},
4         {height, 0},      {height, width / 2},      {height, width}
5     };
6
7     for (const cv::Point& pocket : pockets) {
8         float distance = cv::norm(pocket - previousBall.getCenter());
9         if (distance < 36) {
10             std::cerr << "scored " << previousBall.label << '\n';
11         }
12     }
13 }
```

5.20. kódrészlet.

Potenciálisan lelökött golyó elhelyezett zsebének vizsgálata.

Ez a kódrészlet az előző 5.19 kódrészletnek a 11. sorában jelzett üres helyen hajtódkik végre, itt először a 4.3 résznél ismertetett 4.3 táblázat alapján készül egy lista a zsebek helyzetéről a szélesség (width) és magasság (height) értékek segítségével, ez a pockets változóban kerül tárolásra. A zsebek helyzetén végigiterálva, amennyiben a golyó és a zseb pozícióinak távolsága egy bizonyos szint alá esik, feltételezhetjük, hogy a golyó el lett helyezve a zsebbe. A két pont távolsága a már jól ismert cv::norm függvény[8] segítségével számolható ki, a távolság értéke a distance változóban kerül tárolásra az ellenőrzéshez, amely a 9. sorban látható vizsgálattal történik meg.

A golyó lelökésekor a fenti algoritmus egy üzenetet ír ki a konzolablakba ami megadja, hogy milyen színű golyó lett lelökve, ez a kód 10. sorában látható.

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, 2022. URL <https://www.tensorflow.org/guide/autodiff>.

- [8] OpenCV Documentation. Opencv dokumentáció, 2020. 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.
- [10] George Stockman and Linda G. Shapiro. *Computer Vision*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2001. ISBN 0130307963.
- [11] Satoshi Suzuki and KeiichiA 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: [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, 2002. 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>.

- [19] Tobias Hermann. frugally-deep. <https://github.com/Dobiasd/frugally-deep>, 2016.