Önálló szakmai projekt II.

Ki nevet a végén játékról készült videó alapján a játékmenet követése

Domán Dániel
témavezető: Pataki Péter (ARH Zrt.) és Tamaga István (ARH Zrt.)

A feladat erre a félévre a dobókocka megtalálása és leolvasása fénykép alapján elemi képalkotó eszközökkel és gépi tanulással.
Programnyelv: C++
Képfeldolgozó szoftverkönyvtár: OpenCV
Fejlesztői környezet: Qt Creator
Detektor: Haar Cascade

1. Dobókocka megtalálása és leolvasása elemi képfeldolgozó eszközökkel

A dobókockát vázlatosan az alábbi gondolatmenet alapján szeretnénk azonosítani. Detektálunk éleket a képről, és keresünk olyanokat, amik egy zárt konvex töröttvonalat alkotnak, a töréspontok száma (legalább) 4, az átmérőjének nagysága csak meghatározott értékek között mozog, a területe nagyjából megegyezik az átmérője négyzetének felével. Ezután keresünk benne pöttyöket, ha találtunk benne 1-6 fekete pöttyöt, akkor megtaláltuk a dobókockát. Ennek a gyakorlatban történő meghatározását részletezzük a következőkben.

Az detektált éleket töröttvonalként értelmezzük, a töréspontokat pedig egy vektorban tároljuk.

1.1 Az algoritmus gyakorlati megvalósítása, függvények

Az alábbi egyszerű segédfüggvényeket hoztuk létre:

- ***float dist***
  - INPUT: p_1, p_2 síkbeli pontok
  - OUTPUT: a pontok távolsága
- ***float atmero***
  - INPUT: *alakzat* vektor, melyben pontok
  - OUTPUT: a két legtávolabbi pont távolsága
- ***Point sulypont***
  - INPUT: *alakzat* vektor, pontokból áll
  - OUTPUT: a pontok súlypontja
- ***bool isFekete***
  - INPUT: *p* pont, *image*
  - OUTPUT: *IGAZ*, ha a *p* színe az *image* képen fekete (konkrétan a pont színének mindhárom értéke – *RGB* – legfeljebb 50), különben HAMIS

A továbbiak bonyolultabbak, azokat akkor mutatjuk be, amikor használjuk őket. Előfordulnak olyan segédfüggvények, amiket használunk, de nem mutatunk be (pl. terület, konvex burok). Ezek az OpenCV beépített, előre megírt függvényei.

A következőkben a ***findKocka*** függvényünket fogjuk részletezni, típusa *void*, inputja egy kép és pontokat tartalmazó vektorokból álló vektor, ehhez fogjuk hozzáadni a jelölteket.

A kocka megtalálásához szükségünk lesz először egy négyzetdetektáló algoritmusra (valójában téglalapokat detektál). A *findKocka* függvényünkhöz az alábbi címen található kódot tekintjük

alapnak, az OpenCV hivatalos honlapján szereplő mintakód segítségével tudunk négyzetet detektálni.

https://docs.opencv.org/3.4/db/d00/samples_2cpp_2squares_8cpp-example.html

Azonban ahhoz, hogy pontosan a dobókockát találja meg, szükségünk van a kód módosítására.

A *tresh* és *N* paramétereket 5000-re, illetve 40-re állítjuk. A tapasztalat azt mutatja, hogy ezekkel a paraméterekkel (és ha a nem kívánt szűrőfeltételeket elhagyjuk) a dobókockát mindig sikerül élként detektálni.

A mintakód szűrőfeltételeket alkalmaz, és csak azok az élek kerülnek rögzítésre, amik mindegyiken átmennek. Ezen feltételek helyenként túl szigorúak, mivel a detektálás minősége sok mindentől függ, és sokszor nem sikerül tökéletesen a kockát megtalálni. Máshol erősíteni kell a szűrőfeltételeket, mivel mi nem csak egyszerűen téglalapokat akarunk megtalálni.

Elhagyjuk a feltételek közül azt, hogy csak a 4 pontú alakzatokat engedjük tovább, hiszen a dobókocka felülnézetből valójában egy kerekített sarkú négyzet, ezt módosítjuk legalább 4-re. Emiatt értelemszerűen azt a feltételt is el kell hagynunk, hogy a szögek legyenek nagyjából 90 fokosak. Ezek helyett a négyzetnek egy olyan tulajdonságát kell megfognunk, ami a kerekített sarkú négyzetre is igaz. (Az oldalhosszok egyenlőségére nem is vonatkozott feltétel, tehát a mintakód valójában eredetileg téglalapokat detektált, ezt is meg kell javítanunk.) A tulajdonság pedig az átmérő és a terület közötti összefüggés: a négyzet területe megegyezik az átmérő hossznégyzetének felével. (Egy alakzat átmérője alatt a két legtávolabbi pontjának távolságát értjük). Ezt követeljük meg, kis hibát engedélyezünk (a hányadosnak 0,35 és 0,6 közé kell esnie). Ez természetesen gyengébb feltétel, ezért szükség lesz többre is.

Kikötés kell a keresett alakzat méretére. Tapasztalat alapján erre jobb módszer, ha a területre teszünk kikötést, és nem az átmérőre. A jelenleg vizsgált képeken (1920x1080) ennek az értéknek 1800 és 2500 közé kell esni. Ha másfajta fotók / videók alapján akarunk majd kockát detektálni, akkor szükség lehet ennek a finomítására.

Előfordul, hogy a kocka közelében, vagy igazából bárhol a képen egynél több alakzatot is megtalálunk, amik lényegében megegyeznek, mégis két alakzatként érzékeli a detektáló. Ezen problémára az a módszer, hogy az egyiket meghagyjuk, és a többit, ami hasonló helyen hasonló méretű kitöröljük, nem mindig jó megoldás, mivel előfordulhat, hogy pont egy olyan alakzatot hagyunk meg, ami nem tartalmazza a kockát teljes egészében, esetleg a kocka árnyékát határolja. Sokkal jobb eredményt ad, ha összeolvasztjuk ezeket:

- ***void osszeolvaszt***
  - INPUT: *alakzatok* vektor, benne pontokból álló vektorok
  - OUTPUT: -
  - a függvény az alakzatokon egy bizonyos sorrendben két egymásba ágyazott *for* ciklussal végigmegy, ha talál kettőt, melyek súlypontjai közelebb vannak egymáshoz, mint az egyiknek az átmérője és az átmérők hányadosa ¼ és 4 közé esik, akkor a két alakzat vektorát kitöröljük és a töréspontjaik konvex burkával helyettesítjük az *alakzatok* vektorban. Az indexeket ezután helyre kell állítani.

Az *összeolvaszt* függvényt meghívjuk az eddigi szűrőkön átment alakzatokra. A megmaradt alakzatok lesznek a jelöltek.

A következő lépés, hogy a kockajelöltek közül azonosítsunk egyet, ezt pedig a rajta lévő pöttyök alapján fogjuk megtenni.

Létrehozzuk a ***cropKocka*** függvényt, a bemenete egy pontokat tartalmazó vektor és egy kép. A képből kivágunk egy kisebb téglalap alakú képet, a pontok közül meghatározzuk a legkisebb és legnagyobb *x* és *y* koordinátát, ezekből számoljuk ki, hogy hol kell a téglalap négy oldalát kivágni, illetve ha lehetséges, akkor még 40 képpontot ráhagyunk. Ha netán bármikor kimennénk az eredeti képből, akkor azon az oldalán nem vágunk belőle.

A következő lépésben végigmegyünk a *findKocka* által talált jelölteken sorban, amíg nem találtuk meg a kockát, addig mindegyiket először kivágjuk, majd a következőkben részletezett *findPotty* függvénnyel eldöntjük, hogy megtaláltuk-e a kockát.

A ***findPotty*** inputja egy kép, az outputja pedig egy nemnegatív egész. Ehhez is a feljebb belinkelt mintakódot tekintjük alapnak, csak ezt másként fogjuk módosítani. Lényegében itt is egy konvex, zárt töröttvonalat fogunk keresni, csak más feltételekkel. Azt nyilván nem követeljük meg, hogy 4 pontja legyen, se azt, hogy derékszögei. Egy alakzatot elfogadunk, ha a területe nagyobb 30 egységnél, az átmérője legfeljebb a kép szélességének a hatoda. A pöttyök valójában körök, egy körnek ha *d* az átlója, akkor a területe $d^2\pi/4$. Ezért kikötjük még feltételnek, hogy a terület legyen az átmérő négyzetének *0,6* és *0,8*-szorosa közt.

Itt is felmerül az egymást fedő pöttyök kérdése, itt jó megoldás lesz a törlés: az összeolvasztáshoz hasonlóan használom a dupla *for* ciklust. Ha két alakzat súlypontja közelebb van egymáshoz, mint az közülük az egyik átmérőjének a nagysága, és az átmérőik nagyságának hányadosa ½ és 2 közé esik, akkor az egyiket törlöm (nyilván ha az átmérőik közt nagyobb lenne a különbség, akkor is törölni kell az egyiket, hiszen a pöttyök nagyjából egyforma méretűek, csak ekkor nem egyértelmű, hogy melyiket, és a gyakorlatban eddig ez az eset nem fordult elő, szóval ezt a feltételt akár el is hagyhatnánk).

A függvény visszatérési értéke a megtalált pöttyök száma.

Akkor döntünk úgy, hogy megtaláltuk a kockát, ha a *findPotty* visszatérési értéke legalább 1 és legfeljebb 6. Ha ez egyszer se sikerül, akkor sikertelennek tekintjük a kocka megtalálását.

1.2 A módszer hatékonysága

A tesztelt 37 képből 35-ön sikerült felismerni a dobókockát, ebből 32 esetben sikerült helyesen leolvasni a pöttyök számát. A sikertelen kockakeresések közül egyik esetben semmit se találtunk, a másikban pedig egy, a játéktáblára kerülő pöttyöt, és tévesen felismerte 1-es dobásnak.

A tanulásos módszerhez fontos, hogy a kocka helyét akkor is találjuk meg, ha nem helyes a leolvasás, ha pedig ez nem sikerül, akkor vegyük ezt észre, és ne adjunk vissza hamis eredményt. Ezt a következő módon biztosítjuk: két dobásértéket határozunk meg. Az egyiket úgy, ahogyan eddig, a *findKocka* által meghatározott jelölteket sorban addig vágjuk ki és keressük a pöttyöket, amíg nem kapunk egy 0-nál nagyobb és 7-nél kisebb eredményt. A másikat pedig úgy, hogy a jelölteken fordított sorrendben megyünk végig, és csak akkor fogadjuk el a kockát, ha a két érték megegyezik. Ennél még biztosabb módszer lehetne, ha a pöttyök koordinátáit is összehasonlítanánk. Ezzel kiküszöböljük azt, hogy bár megtalálnánk a kockát, az algoritmus egy hamis kocka miatt hamarabb leáll. Ilyen gyakorlatban is előfordult, amikor egy pötty (talán egy ruhadarabról leszakadt dolog) került a táblára, a környezetét pedig az algoritmus kockaként értelmezte.

Ezzel a módszerrel a 37-ből 34-szer sikerült megtalálni a kockát, azaz egy esetben fordult elő, hogy az algoritmus több kockát is talált, viszont a korábbi módszerrel először a helyeset adta vissza.

2. Dobókocka megtalálása gépi tanulással

A kockát megpróbáljuk gépi tanulás segítségével megtalálni. Ehhez a Haar Cascade algoritmust fogjuk használn.

2.1. Néhány szó a Haar Cascade algoritmusról

A Haar Cascade a tanulás során legelőször lebutítja a keresett objektumot egészen kicsi méretűre esetünkben 16x16 pixelesre. A Haar Cascade algoritmus úgynevezett gyenge klasszifikátor bevetésével próbálja meg beazonosítani a tárgyakat. Például ha arcfelismerésre kellene használni, akkor olyan megfigyeléseket tenne, hogy az emberek homloka világosabb, mint a szemöldöke. A működés elve, hogy kidobja azt, ami biztosan nem a keresett objektum. Ennek az az értelme, hogy villámgyorsan elhajítsuk a vizsgálandó kép területének 90-95%-át. A potenciális objektumok további szűrését már csak az első lépésben “reménykeltőnek” jelzett területeken kell lefuttatni ugyanúgy, mint az előbb, csak egyre finomabb szűrőkkel. Minden szinten további „nem-objektumokat” hajítunk ki. A mi esetünkben 15-20 szintet használunk.

2.2. Tanító adatok előállítása

Ahhoz, hogy ez az algoritmus működjön, szükségünk van nagy mennyiségű tanító adat előállítására, pozitív, illetve negatív képekre.

2.2.1. Pozitív adatok
A pozitív képek alatt azt értjük, hogy rajta van a dobókocka. Készítenünk kell róluk egy leírást, amiben megtalálhatóak a képfájlok elérési útvonalai. Továbbá meg kell adni a kockák darabszámát (nálunk tipikusan 1), és a koordinátáit. Ezt meg lehetne oldani úgy, hogy minden képen külön-külön bejelöljük, hogy hol van a kocka. A másik lehetőség, hogy felhasználjuk az 1-es fejezetben tárgyalt kockadetektáló algoritmusunkat. Mi az utóbbi lehetőséget követjük, mégpedig az erről szóló fejezet legvégén tárgyalt változatot, ahol vagy a jó eredményt adjuk vissza, vagy hibát, de rossz eredményt (remélhetőleg) sose. Ha egy kép hibás, azaz nem sikerült megtalálni a kockát, akkor nem kerül be a tanító adathalmazba.

2.2.2. Negatív adatok
Az úgynevezett negatív tanítóadatok közt olyan képeknek kell szerepelnie, amikkel a klasszifikátor találkozni fog, de nincs rajta a keresett objektum. Ez esetünkben a játéktáblát jelenti a dobókocka nélkül. Néhány kísérlet után a negatívokat gyarapítottuk azzal, hogy a hamisan megtalált kockákat is beleraktuk.

2.2.3. Adatok sokszorozása
Ahhoz, hogy a tanulás eredményes legyen, nem elég 30-50 kép, szükség lenne legalább ezerre. Azonban van más megoldás is, mint ezer fotó elkészítése. A képeken apró módosításokat végrehajtva, igen sok példányt lehet generálni. Az egyik módosítási lehetőség a fényerő változtatása. Erre egyébként is szükség lenne, hiszen a végcél a videós elemzés, ahol közel sem lehet mindig olyan fényviszonyokat előállítani, mint amilyeneket egy fotóhoz. A másik, pedig az, hogy a képeken valamilyen lineáris transzformációt hajtunk végre. Ilyen például a tükrözés és a forgatás. Ennél bonyolultabb, hogy a téglalap alakú képet minimálisan trapézzá „összenyomjuk”. Ez azért is bonyolultabb, mert a kockát egy deformált képen rosszabb eséllyel talál meg a korábbi algoritmus. Erre a megoldás az, hogy az eredeti képen megtalált kocka helyét a képpel együtt transzformálni kell.

Másik lehetőség sokszorozásra, hogy videóból vágjunk ki képeket. A későbbiekben még lehetséges, hogy ezt a módszert is be fogjuk vetni.

2.3. Gyakorlati megvalósítása

2.3.1. A tanítás

Mindenek előtt a pozitív és negatív képek közül is néhányat félreteszünk, később ezekkel tesztelünk.

A tanításhoz a pozitív és negatív képeket két külön könyvtárba kell tennünk: *„positives”*, *„negatives”* és ezeket berakni egy közös könyvtárba: *„ImageSamples2”*. A terminál seítségével belenavigálunk az *ImageSamples2* könyvtárba és a következő parancsot adjuk:

opencv_createsamples -info positive.txt -num 1368 -w 16 -h 16 -vec kocka.vec

1368 pozitív képet használunk, melyeknek a leírása a *pozitive.txt*-ben található, a korábban említett 24x24 helyett mi 16x16 pixelesre csökkentjük a kockák méretét, a 24x24-es szám eredetileg arcfelismerőhöz tartozott, egy dobókockán kevesebb információ van. Ezzel a paranccsal hozzuk létre a *kocka.vec* fájlt, amire szükség van a tanításhoz. Létre kell még hoznunk ez *data* nevű üres könyvtárat. Maga a tanítás a következő paranccsal hajtható végre:

opencv_traincascade -data data -vec kocka.vec -bg negative.txt -numPos 1100 -numNeg 875 -numStages 15 -w 16 -h 16

Érdemes megemlíteni, hogy a tréninghez kicsivel kevesebb pozitív képet használhatunk fel, mint amennyit a *vec* fájl elkészítéséhez. A maradék lesz az úgynevezett tesztadat. A *negative.txt*-ben található a negatívok leírása, ami esetükben csak a képfájlok elérési útvonalát tartalmazza, itt épp 875 negatív képpel dolgoztunk. A legutóbbi futtatáskor 15 szintet használtunk. Ez adta a legjobb eredményt, nagyobb szinten többször is kidobta a kockát, alacsonyabb szinten pedig sok hamis kockát talált.

A tanítás során végül egy *xml* kiterjesztésű fájlt generálunk, amit a *data* mappában találunk.

2.3.2. Tanítás után

Az elkészült *xml* fájlt betöltjük egy *CascadeClassifier* típusú változóba. Ennek a segítségével detektáljuk a kockát. Szűrési lehetőség, hogy méretet állítsunk be, azaz egy adott méretnél nagyobb vagy kisebb találatokat ne vegyünk figyelemben. Lényegében a tanító módszer a *findKocka* függvényt tudja kiváltani.

2.4. Eredmények

A legutolsó futtatáskor a letesztelt 249 képből 205 helyen találta meg a kockát, és 188 alkalommal sikerült helyesen leolvasni. A leolvasást a korábbi *findPotty* segítségével csináltuk.

Ez az arány (egyelőre) kicsivel rosszabb, mint a tanulás nélküli módszerrel, bár az is tény, hogy azzal defromált táblát nem is vizsgáltunk. Ezen kívül a tanulásos módszer gyorsabb, ami a videóelemzésnél hasznos lehet. A másik előnye a tanulást nem alkalmazó módszerhez képest, hogy ha a gyakorlat azt mutatja, hogy ez a hatékonyság kevés, akkor sokkal jobb esélyünk van tovább fejleszteni, és jobb találati arányt elérni.

Leteszteltük azon a 37 képen is, amelyiken kezdetben dolgoztunk, ezeknek egy része benne volt a tanítóadatok közt, de nem mindegyik. A 37-ből 36 esetben sikerült megtalálni a kockát, tehát itt már sikerült jobb hatékonyságot elérni.