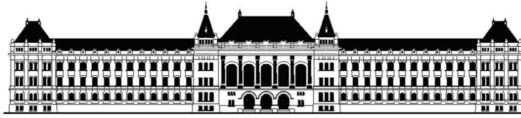


Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Távközlési és Médiainformatikai Tanszék

Grafikus processzorral gyorsított képszegmentálási algoritmus implementálása Android platformon

Szerző:
OPRA ISTVÁN BALÁZS

Konzulens:
DR. VAJDA FERENC

2013. OKTÓBER 25.

Tartalomjegyzék

1. Bevezetés **3**
1.1. A célkitűzés 3
1.2. A dokumentum felépítése 4

2. Elméleti háttér **5**
2.1. Képfeldolgozási eszköztár 5
2.1.1. Színterek 5
2.1.1.1. Szürkeárnyalatos képek 5
2.1.1.2. RGB és HSV 5
2.1.1.3. CIE L*a*b* 7
2.1.2. Küszöbözés 7
2.1.3. Morfológiai műveletek 8
2.1.4. Kép nyomatékok 9
2.2. Színszegmentálás 10
2.2.1. Az alapötlet kiválasztása 10
2.2.2. A távolság meghatározása 11
2.2.2.1. Mahalanobis-távolság 11
2.2.2.2. A megfelelő színreprezentáció 12
2.2.2.3. Küszöbérték választás 15
2.3. Android platform 15
2.4. Android hardver 16
2.5. Grafikus feldolgozó egységek 17
2.5.1. OpenGL ES 18
2.6. OpenCV függvénykönyvtár 19
2.7. Ellenállás kódolás 19

3. Az algoritmus Android implementációja **21**
3.1. Felhasználói interakció 21
3.2. Az ellenállás testének azonosítása 21
3.2.1. Detektálás indítása, kép vágása 22
3.2.2. Éldetektálás és küszöbözés 23

3.2.2.1. S csatorna küszöbözés 23
3.2.2.2. Éldetektálás 24
3.2.2.3. Morfológiai utófeldolgozás 25
3.2.3. A vezetékek leválasztása 25
3.3. A színsávok detektálása 27
3.3.1. Méret normalizálás 27
3.3.2. OpenCV implementáció 28
3.3.3. OpenGL implementáció 29
3.3.3.1. Android jellegzetességek 29
3.3.3.2. Az OpenGL program 30
3.4. A dekódolás 33
3.4.1. Morfológiai előkészítés 33
3.4.2. Kontúrok és nyomatékok meghatározása 34
3.4.3. Pontszerű hibák kiszűrése 34
3.4.4. A tok orientációjának számítása 34
3.4.5. Régió összevonás 36
3.4.6. Az ellenállás érték meghatározása 37
3.5. A kép előzetes kondicionálása 37
3.5.1. Fehéregyensúly 38
3.5.1.1. Súlytényezők meghatározása 39
3.5.1.2. OpenCV előnézet használata 40
3.5.1.3. OpenGL előnézet használata 40

4. Tesztelés **42**

5. Eredmények, továbblépési lehetőségek **44**

1. fejezet

Bevezetés

1.1. A célkitűzés

A képszegmentálás klasszikus képfeldolgozási probléma. Fő célja, hogy egy képet különálló szegmensekre bontson, amik jól megfeleltethetők való világbeli objektumoknak. A képszegmentálási algoritmusok közül a szín alapú szegmentálás népszerű kutatási területté vált a hardvereszközök teljesítményének növekedésével és a színes képek alkalmazásának elterjedésével.

Napjainkban az átlagos mobilkészülékekben már alapfelszereltség a színes kamera szenzor, és az ilyen eszközök teljesítménye vetekszik a néhány évvel ezelőtti asztali számítógépekével. Az ilyen hordozható készülékek piaca folyamatosan nő. Felmérések szerint [6] 6,8 milliárdra tehető az aktív mobiltelefon előfizetések száma. Az előfizetések közel egyharmada pedig mobil szélessávot is tartalmaz, azaz egy internetböngészésre alkalmas, fejlettebb készülék – divatos kifejezéssel „okostelefon” – társul hozzá.

Fontos cél tehát, hogy az ezekben a készülékekben rejlő lehetőségeket minél inkább kihasználjuk. A színes kamera és megfelelő feldolgozóképesség lehetővé teszi, hogy képfeldolgozási feladatokat hajtsunk végre velük – számtalan konkrét alkalmazás lehetséges. A gépi látás alkalmazása segíthet a tájékozódásban (augmented reality), vásárlásban (vonalkód leolvasás), vagy akár laboratóriumi munkában is.

Központi feladatul azt tűztem ki, hogy egy, a mobileszközök képességeihez igazodó, lehetőleg minél gyorsabb szín alapú felismerési eljárást keressek, és egy konkrét alkalmazásban, mobil hardveren teszteljek. Ilyen tesztelő alkalmazásnak a furatszerelt ellenállások színkódjának felismerését választottam.

1.2. A dokumentum felépítése

A második fejezetben bemutatom a kiválasztott algoritmust, és az implementációban használt eljárásokat. Továbbá rövid áttekintést adok a feladathoz köthető témakörökhöz is.

A harmadik fejezetben írom le az Android alkalmazás elkészítésének munkáját, és ezen keresztül bemutatom a ténylegesen implementált algoritmust.

A negyedik fejezetben a tesztelési folyamatról írok.

Végül az ötödik fejezetben összefoglalom az elért eredményeket, és továbblépési lehetőségekre teszek javaslatot.

2. fejezet

Elméleti háttér

2.1. Képfeldolgozási eszköztár

A következőkben áttekintek néhány alapvető módszert a képfeldolgozás eszközei közül, amiket felhasználtam az implementáció során.

2.1.1. Színterek

A gépi látási feladatokat dominánsan digitális képeken végezzük. A mobil eszközök kamera hardvere is ilyen kimenetet produkál. A digitális kódolás alapvető módszere a szürkeárnyalatos képek ábrázolásán mutatható be.

2.1.1.1. Szürkeárnyalatos képek

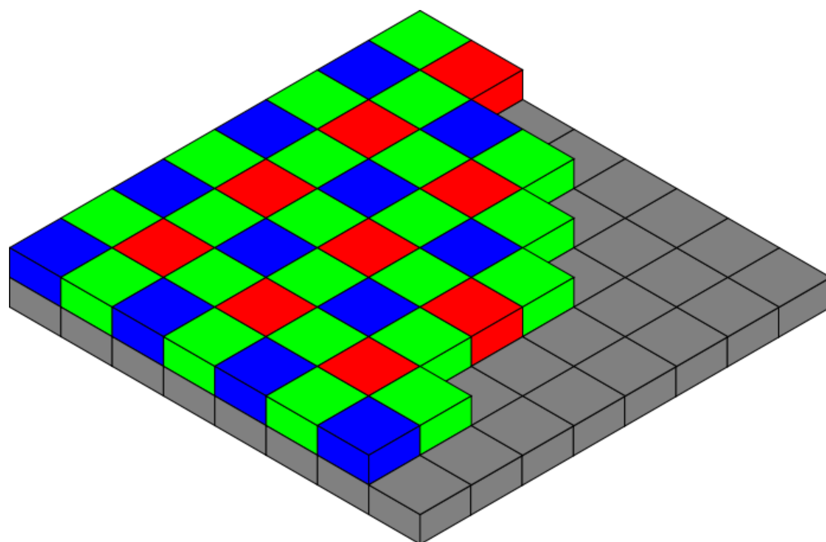
Az ilyen, monokróm képek – mint minden digitalizált kép – képpontokból állnak. Szürkeárnyalatos esetben egy képpontot egyetlen fényerő érték ír le. Ha a digitális kamerák kimenetét nem vetjük alá méretcsökkentő tömörítési eljárásoknak, akkor az eredmény kép egy pixele a kamera hardver egy fényerő szenzorának feleltethető meg. Az ilyen kódolásra mondjuk, hogy egy csatornás. A képnek tehát beszélhetünk a térbeli felbontásáról, ami megadja, hogy vízszintes és függőleges irányban hány képpontból áll, illetve az árnyalat-felbontásáról, azaz hogy hány különféle fényesség értéket vehetnek fel a pixelek.

2.1.1.2. RGB és HSV

A színes képek nyilvánvalóan több információt hordoznak, így több memóriát is foglalnak. Egy színes képpontot általában három csatorna értéke ír le. A digitális kamerákban színes képek készítésére színszűrő lencséket tesznek az egyes

fényerő szenzorok elé. A leggyakrabban használt elrendezés a Bayer szűrőmozaik, feltalálójáról, Dr. Bryce E. Bayer-ről kapta nevét. Az elrendezés a 2.1 ábrán látható.

2.1. ábra: A Bayer színszűrő elrendezés



A rácsban két zöld szenzor jut egy-egy piros és kék érzékelőre, ez az emberi szem zöld színre való nagyobb érzékenységét próbálja közelíteni. Egy három csatornás kimeneti kép használatakor egy szenzor önmagában nem képes egy pixel teljes leírására, a három csatorna értékét valamilyen interpolációs eljárással kell számolni a környező szenzorok értékeiből.

A Bayer szűrős kamerák úgynevezett nyers, interpolálatlan kimenete zöld, kék és piros pixeleket tartalmaz, a legkézenfekvőbb interpolációs eljárás tehát egy piros, kék és zöld csatornákból álló képet eredményez. A három csatorna intenzitás értékei függetlenek, összeadhatók, így alkotható belőlük egy három dimenziós Descartes-tér: ez az RGB színtér.

Ugyanakkor ezt a teret különféle transzformációk segítségével átalakíthatjuk igényeinknek megfelelően. Egy ilyen változat az YUV színtér, ami a műsorszórás igényeinek megfelelően alakítja a színeket. Az Y csatorna az RGB csatornákból az emberi szem érzékenységének megfelelő arányban súlyozott fényerő értéket ad. Az U és V pedig a műsorszóró hardverrel való kompatibilitásnak megfelelően számítódnak.

A fent említett két színtér a hardver megkötéseit és sajátosságait követi – pontok színtérbeli helyzete és távolsága nem arányos az emberi szem által érzékelt különbséggel ezekben a terekben. Ez képfeldolgozás szempontjából nem igazán előnyös. Léteznek ugyanakkor más, az emberi látás jellegzetességeihez jobban

igazodó kódolási formák is.

Az egyik ilyen megközelítés a HSV színtér. Itt is három csatorna állítja elő a színeket, ám mindháromnak az ember számára intuitív jelentése van. A H – hue, színárnyalat – csatorna kódolja a tulajdonságot, amit legtöbben a szín alatt értenek, például ami alapján a pirosat meg lehet különböztetni a sárgától. Az S – saturation, telítettség – mutatja meg, hogy mennyi szín van jelen, például ez különbözteti meg a halvány pirosat a vöröstől. Végül a V – value, érték – csatorna a fényerőt kódolja, például a világoskék és sötétkék közötti különbséget. Az implementációmban ezt a színteret is felhasználom az ellenállás képen való pozíciójának azonosítására, mivel a telítettség (S) csatornán jól elválasztható a semleges színű háttér a tarka ellenálláshoz képest (lásd a 3.2.2.1 fejezetet).

2.1.1.3. CIE L*a*b*

A HSV színtérrel rokonságban áll a CIE L*a*b*. Ez a CIE (Commission internationale de l'éclairage) által 1931-ben kidolgozott XYZ színtérre alapul, ami az első matematikailag definiált színkódolási eljárás. Az emberi látáson végzett kísérletek alapján dolgozták ki. A szemben három fajta színérzékeny sejt található, egy rövidebb, egy közepes, egy pedig nagyobb hullámhosszakra reagál inkább (innen származik a piros-zöld-kék alapszínek gondolata). A kísérletek azt mutatták, hogy a látás e három sejtcsoport által adott jelek különbségét dolgozza fel. Megfogalmazták, hogy három különbség-csatorna létezik, a piros-zöld, kék-sárga és fekete-fehér. Az ilyen színek intenzitása látásunkban egymás ellen dolgozik, ezért nem látunk például pirosas zöldnek semmit.

Az L*a*b*-ben a fekete-fehér csatorna az L (lightness, fényesség), a másik kettő pedig az a és b. A HSV-hez tehát annyiban hasonlít, hogy itt is van egy külön fényerőt leíró adat, amitől elválik a többi színjellemző. Az L csatorna értékkészlete $[0, 100]$, 100 a maximális fényerő, a és b pedig $[-127, 127]$ tartományban mozoghat.

A színtér nagy előnye, hogy ha 3D Descartes-koordináta rendszerként tekintünk a csatornákra, akkor két szín euklidészi távolsága nagy mértékben arányos az emberi szem által érzékelt érzetkülönbséggel.

2.1.2. Küszöbözés

A képszegmentálás egyik hagyományos, egyszerű módja a küszöbözés [9]. Ennek során meghatározunk egy fényesség tartományt az eredeti képen, és az ebbe eső képpontokat az előtérbe tartozónak, a többi képpontot pedig háttérnek osztályozzuk. Eredményül egy bináris, vagy kétszintű képet kapunk. Az előteret és hátteret tetszőleges módon jelölhetjük az adott kódolásban, elterjedt megoldás, hogy az előteret 1-gyel, a hátteret 0-val jelöljük.

A módszer könnyen általánosítható, ha nem egyszerűen pixel intenzitásokat hasonlítunk össze egy küszöbértékkel, hanem valamilyen összetettebb módszerrel származtatunk egy-egy számértéket a képpontokhoz.

2.1.3. Morfológiai műveletek

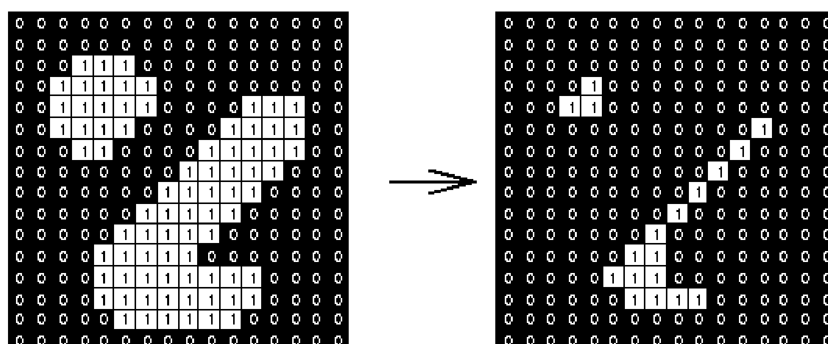
A morfológiai operátorok használata széles körben elterjedt a képfeldolgozás sok területén. Gyakran előálló helyzet, hogy bináris képen kell dolgoznunk (például küszöbözés után, lásd előző rész), ami előfeldolgozás nélkül még egy igen összetett szegmentáló algoritmus alkalmazása esetén is tartalmaz hibásan háttérnek vagy előtérnek meghatározott pixeleket. Ezek kijavítására egyszerű és megbízható eszközök a morfológiai műveletek (áttekintő összefoglalásért lásd [3]). Mivel szegmentálási feladatot hajtok végre, így én is többször alkalmazom őket.

Ezek alapja az erózió és dilatáció operátorok. Bináris képről pixeleket távolítanak el (erózió) illetve adnak hozzá (dilatáció) adott szabályok szerint. A műveletek a kiindulási kép minden pixelén végrehajtódnak, és csak a kiindulási pixel elrendezéstől függ eredményük. Megjegyzendő még, hogy ezen operátorok nem reverzibilisek, a végeredmény képből nem lehetséges az eredeti kép visszaállítása.

Az általam használt operátorok egy rögzített pixelelrendezést használnak (kernel), aminek definiált középpontja van. Ezt a középpontot ráhelyezik a kiindulási kép minden pixelére, és vizsgálják, hogy egyezik-e a kép a kernellel. Dilatáció végzésekor egyetlen egyezés esetén már aktív állapotú lesz az eredmény kép pixele, míg ehhez erózió esetén teljes egyezés kell a kiindulási kép és kernel között.

A kernel lehet tetszőleges elrendezésű. Én diszk alakúakat használtam, amik a legtöbb célra megfelelnek. Az alábbi ábrán látható az erózió hatása.

2.2. ábra: Morfológiai erózió hatása 3x3-as diszk kernel alkalmazása esetén



Mint látható, az erózió karcsúsítja a nagyobb összefüggő komponenseket, míg a kisebbeket teljesen eltávolítja. Ebből fakadóan kiválóan alkalmas só és bors típusú, pontszerű zaj kiszűrésére. A dilatáció ezzel ellentétes hatású, új aktív

pontokat vesz hozzá ez előtér régiók széléhez. Segítségével „hézagos” régiókat kitölthetünk, közeli blobokat pedig összeköthetünk.

A morfológiai operátorok, mint említettem roncsoló hatásúak. Ha a zajt eltávolítottuk, vagy a réseket kitömtük, a megmaradó objektumok mérete is változott. Ezt a hatást elkerülhetjük, ha a dilatációt és az eróziót egymás után alkalmazzuk. Ezek a műveletek külön néven szerepelnek: a morfológiai nyitás és zárás.

A nyitásnál először több eróziót hajtunk végre, majd ugyanennyi dilatációt. A kisebb méretű előtérbeli objektumok eltűnnek, a nagyobbak pedig közelítően megőrzik eredeti méretüket és alakjukat. A nyitásban fordított a beavatkozások sorrendje. Kitölti a réseket, és összeköt közeli régiókat, míg azok kiterjedésén nem változtat.

Ezek az operátorok mind erősen függnek a kép felbontásától és a használt kernel méretétől, ezért használatuk odafigyelést igényel.

2.1.4. Kép nyomatékok

Amennyiben már bináris képünk van, és ezen összefüggő régiókat vizsgálunk, akkor sok hasznos információt nyerhetünk ezekről a kép nyomatékok használatával. Az úgynevezett térbeli nyomatékok a következőképp kaphatók:

$$m_{ij} = \sum_{x,y} I(x,y) x^i y^j \tag{2.1}$$

Itt $I(x, y)$ a bináris kép adott (x, y) koordinátájához tartozó értékét jelenti. A $\max i, j$ kifejezés adja a nyomaték rendjét. A 0-ad rendű nyomaték (m_{00}) az előtér pixelek száma.

Elsőrendű nyomatékok segítségével a vizsgált régió (x_c, y_c) tömegközéppontját is megkaphatjuk:

$$x_c = \frac{m_{10}}{m_{00}}, \qquad y_c = \frac{m_{01}}{m_{00}} \tag{2.2}$$

A térbeli nyomatékokon kívül gyakran használnak még úgynevezett centrális nyomatékokat. Ezek az eddigi jelölésekkel így számíthatók:

$$\mu_{ij} = \sum_{x,y} I(x,y)(x - x_c)^i (y - y_c)^j \tag{2.3}$$

A kép nyomatékok használatával megkaphatjuk például a bináris kép pixeleire illeszthető ellipszis orientációját, vagy főtengelyeinek hosszát. Ezeket az implementáció során én is kihasználom.

2.2. Színszegmentálás

Munkám elsődleges célja a színszegmentálási algoritmus kidolgozása, implementálása és tesztelése volt. A továbbiakban bemutatom a kiválasztás folyamatát és működés elvét. Az ebben a részben dokumentált munkához a számításokat és teszteket Matlab környezetben végeztem az Image Processing Toolbox használatával. A kapott eredmények alapján készítettem az Androidos alkalmazást, amit a 3. fejezetben írok le.

2.2.1. Az alapötlet kiválasztása

Egy olyan módszert szerettem volna találni, ami a mobil készülékek relatíve kisebb számítási kapacitása mellett is igen nagy sebességgel képes adott színű objektumok azonosítására, így minimális járulékos terhelést jelent egy komplexebb képértelmezési feladat alapjaként. További célom volt, hogy valósidejű alkalmazásokban is jól alkalmazható algoritmust találjak, így a módszer futási idejének felső korlátja determinisztikusan számolható kell legyen.

Első lépésként megvizsgáltam néhány megoldást a terület irodalmából. Egy lehetséges módszert kínál a régió növesztésen alapuló megközelítés. Alapja az, hogy a képről kiválasztunk úgynevezett mag-képpontokat, és ezek környezetét vizsgáljuk. Specifikálnunk kell homogenitási kritériumokat, amik segítségével eldönthetjük, hogy a magok szomszédságában levő képpontok bevehetők-e az adott régióba. Egy ilyen döntés egy iterációs lépés, az iterációt addig folytatnunk kell, amíg minden képpont be nem került egy-egy régióba.

A módszer finomítható régió összevonás megengedésével, és számtalan egyéb módon. Egy példa a színszegmentálás területéről a J. Tang által javasol megoldás, ami a mag-pontok kiválasztására az úgynevezett vízválasztó algoritmust használja [10].

A régiónövesztés módszer jó eredményeket képes elérni egyenetlen felületű tárgyak és ideálistól eltérő megvilágítás esetén is. Ugyanakkor implementáláskor komoly figyelmet kell fordítani a valós idejűség korlátozásainak betartására. Ehhez olyan homogenitási kritériumot kell választanunk, ami garantálja, hogy adott lépés elvégzése után statikus állapotot érünk el. A futási idő felső korlátjának számítása azonban még ilyen esetben is meglehetősen összetett feladat.

Az egyszerűbb, képpont-alapú módszerek ilyen szempontból jobban megfelelnek az általam előírt kritériumoknak. Amennyiben egy-egy pixel besorolása valamilyen determinisztikusan kapható leíró szerinti küszöbözésen alapul, akkor a futási idő egyenesen arányos a feldolgozandó pixelek számával.

Emellett az is a pixel szintű szegmentálás mellett szól, hogy a feladat nagy mértékben párhuzamosítható – a pixelekhez tartozó leíró értékek egymástól füg-

getlenül számolhatók –, így a mobilkészülékekben található grafikus segédprocesszor használatával várhatóan nagy sebességnövekedés érhető el.

Ezen megfontolások alapján a képpont-alapú módszerek részletesebb vizsgálata mellett döntöttem. Itt a legfontosabb kérdések a pixelre jellemző leíró előállítási módja és a küszöbözési eljárás.

Phung, S.L. et al. cikkükben [8] több pixel-alapú besorolási módszert megvizsgálnak a bőrszín felismerés kontextusában. Az egyik vizsgált eljárásban (2.2.3-as szekció a cikkben) normál eloszlások kombinációjaként közelítik a detektálandó pixelhalmazt. A távolság meghatározását pedig ezt a feltételezést felhasználva egy speciális módon, a Mahalanobis-távolság felhasználásával végzik. Az általam vizsgált probléma – mesterségesen színezett egyszínű objektumok felismerése – lényegesen könnyebbnek tűnik, ezért abból a feltételezésből indultam ki, hogy a cikkben mutatott eljárás bizonyos szempontok szerinti egyszerűsítése is jó eredményt adhat a sebesség növekedése mellett.

2.2.2. A távolság meghatározása

A következő leírásban egyelőre nem veszem figyelembe az esetenként változó megvilágítás okozta nehézségeket (ezekről a 3.5 fejezetben írok). Most minden képnél és lépésnél állandó megvilágítást tételezek fel.

Elsőként egy kézenfekvő egyszerűsített megoldást teszteltem. Itt a keresett színre jellemző pont egy, az adott színsávot tartalmazó ellenállásról készített kép alapján nyerendő, úgy, hogy a színsávhoz tartozó pixelek értékének számtani közepét vesszük. A távolságszámítás a közönséges euklidészi normával történik.

A globális küszöbértéket a színsáv pixelhalmazának szórására alapozva próbáltam meghatározni, de nem sikerült egyetlen olyan thresholdot kapni, amivel jól elkülönülnek az adott színsáv pixelei. Az Android környezetben a kamerából legkönnyebben kinyerhető kép RGB formátumú, így először ezt a megoldást vizsgáltam meg, siker nélkül. Majd az Lab reprezentációval is kísérletet tettem, de az eredmény hasonlóan rossz volt. Az okot abban találtam, hogy a színsávban előforduló színpontok színtérbeli eloszlása egyik reprezentációban sem közelíthető jól gömbbel. Így a Mahalanobis-távolság használatára tértem át.

2.2.2.1. Mahalanobis-távolság

A Mahalanobis-távolság nem egy definíció szerinti metrika, hanem egy statisztikai jellemző, ami arról ad információt, hogy egy adatpont mennyire tartozik egy rögzített pontokból álló halmazba.

A gömbszerű elhelyezkedésnél pontosabb modellt állít fel, egy ellipszissel (3D térben ellipszoiddal) közelíti az adatpontok eloszlását.

Az illeszkedő ellipszoid egy jellemző reprezentációja az adathalmazból számított Σ kovariancia mátrix. Ha az adathalmazunk n dimenziós, és $\mathbf{x} = \begin{bmatrix} x_1 \\ \vdots \\ x_n \end{bmatrix}$ alakú adatpontokból áll ahol x_i egy valószínűségi változó véges szórással, akkor Σ elemei a következőképp számíthatók:

$$\Sigma_{ij} = E\left[\left(x_i - \mu_i\right)\left(x_j - \mu_j\right)\right] \tag{2.4}$$

, ahol $\mu_i = E(x_i)$, azaz az i-edik változó várható értéke.

E mátrix sajátvektorai egy ellipszoid főátlóinak orientációit adják, a hozzájuk tartozó sajátértékek pedig jellemzőek azok hosszára. A sajátértékek ugyanis $\mathbf{x}$ x_i változói varianciájának felelnek meg.

A Mahalanobis-távolság számítása Σ és $\mu = E(\mathbf{x})$ ismeretében egy tetszőleges $\mathbf{x_r}$ pontra következőképp végezhető:

$$D_M(\mathbf{x_r}) = \sqrt{(\mathbf{x_r} - \mu)^T \Sigma^{-1} (\mathbf{x_r} - \mu)} \tag{2.5}$$

Az euklidészi távolsághoz képesti újdonság a Σ inverzének gyökével való szorzás. Ez szemléletesen annak felel meg, hogy az adathalmaz középpontjától vett távolságot leosztjuk az ellipszoid a vizsgált pont irányában való kiterjedésével – az adott iránybeli szórással.

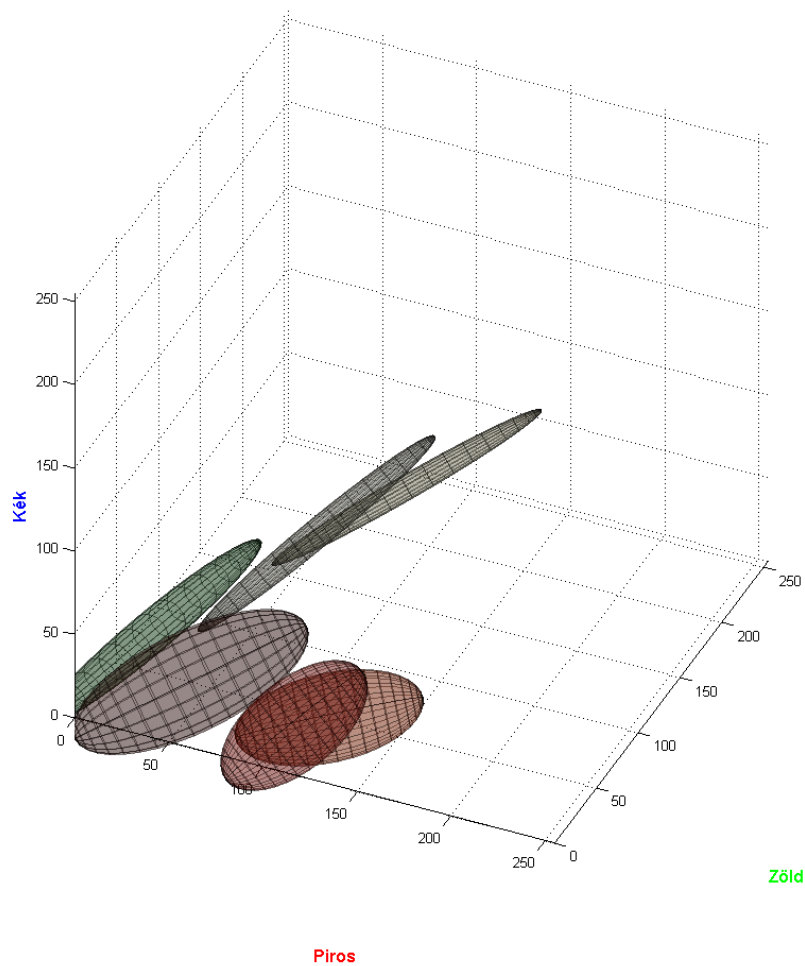
A Mahalanobis-távolság tehát jobb eredményt ad, ha az adathalmazra jól illeszthető egy ellipszoid. Ilyen eset például, amikor közelítően egy többváltozós normál eloszlás szerinti elrendezésűek az adatpontok. A mostani problémában keresett színsávok színpontjairól ugyanakkor intuitíven nem tudjuk megmondani, hogy teljesítik-e ezt a követelményt. Az elhelyezkedésük ráadásul a választott színreprezentációtól is függ. Vizsgálatokat végeztem tehát, hogy eldöntsem, ez a távolságmérték alkalmas-e már a színsávok elkülönítésére.

2.2.2.2. A megfelelő színreprezentáció

Itt is először RGB színtérrel próbálkoztam. Nem számítottam kiváló eredményre, mivel, mint említettem, az RGB színtér nem az emberi látás tulajdonságaihoz illeszkedik (lásd a 2.1.1 fejezetet). Egy-egy színsáv az ellenállás gyártása során homogén színt kap, egy valós képen ezt a homogenitást az ellenállás tokjának görbülete rontja el. Általános megvilágítási környezetben a kerület mentén a színnek leginkább az a tulajdonsága változik, amit az emberek általában világosságnak hívnak. A kamera irányából érkező megvilágítás esetén a tok hossztengelyénél világosak a színek, a tengelytől eltávolodva sötétednek. Ilyen „világosságra”, intenzitásra jellemző információt az RGB reprezentáció explicit módon nem tartalmaz.

A már mintavételezett színsávok adathalmazát ábrázoltam RGB színtartományban, és megvizsgáltam, hogy milyen jellegű ellipszoidot kapok. Az alábbi ábrákon az ellipszoidok mérete úgy adódik, hogy a főtengelyek irányában normál eloszlást feltételezek, a tengelyek hossza pedig 2σ konfidencia intervallumnak felel meg. A színmélység csatornánként 8 bit, a piros, narancs, barna, fehér, szürke és zöld színeket ábrázoltam. Az ellipszoidok színe az adott pixelhalmaz középpontja RGB-ben.

2.3. ábra: Kovariancia ellipszoidok RGB térben



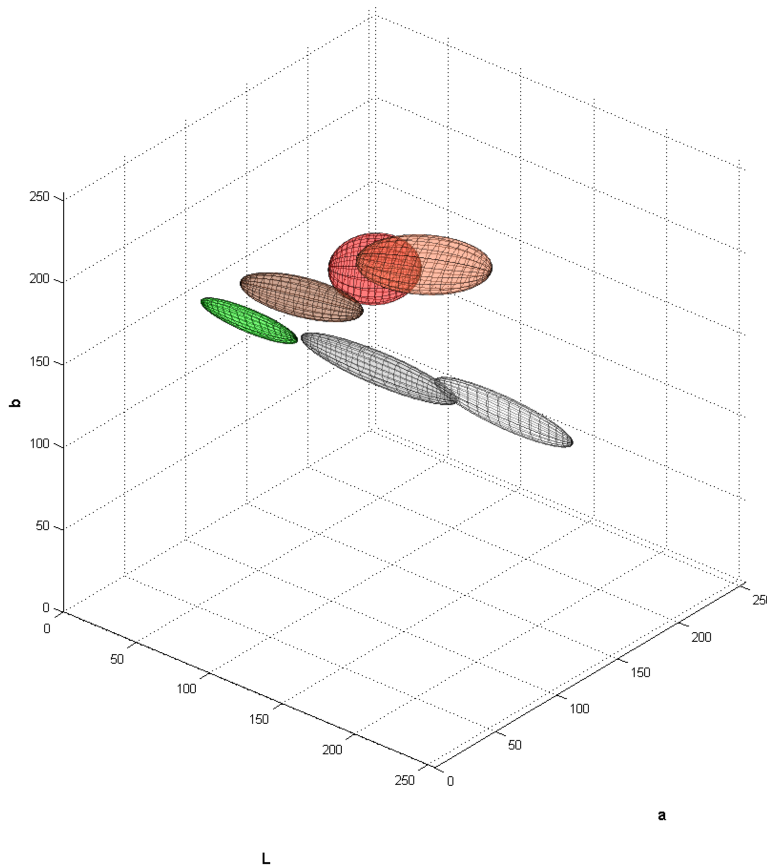
Összehasonlítás céljából ugyanezen színsávok pixelhalmazát CIE L*a*b* térbe transzformáltam, és ott ábrázoltam a pontfelhőt, illetve az illesztett ellipszoidokat. A két színtér értékkészlete eltérő (lásd a 2.1.1 fejezetet), annak érdekében, hogy jobban összehasonlítható ábrákat kapjak, a következő – 8 bites színmélységnél

gyakran használt – átalakítást végeztem még az L*a*b* csatornákon:

$$L \leftarrow L * \frac{255}{100}, \qquad a \leftarrow a + 128, \qquad b \leftarrow b + 128 \tag{2.6}$$

A **??** ábrán az ellipszoidok színezését kézzel végeztem, hiszen az L, a és b értékek RGB színként való ábrázolása nem szemléletes.

2.4. ábra: Kovariancia ellipszoidok L*a*b* térben



Azt tapasztaltam, hogy az RGB színtérben az ellipszoidok főtengelyeinek orientációjában nincs korreláció és kiterjedésük is nagyobb skálán mozog. A nagyobb méret itt nagyobb szórásra utal, azaz az adatfelhők ilyen közelítése RGB térbe kevésbé ideális.

Az L*a*b* térben a hossztengelyek az L („lightness”, fényesség) tengellyel párhuzamosan orientálódnak. Ez az vártnak megfelelő, hiszen, mint már említettem, a színek a görbület mentén főként fényességükben változnak, ezt pedig az L csatorna explicit kódolja.

Ezek alapján az L*a*b* színtér használata mellett döntöttem.

2.2.2.3. Küszöbérték választás

A thresholdot ismert és állandó megvilágítási környezetben kézi állítással is meg lehet kapni. Ennél azonban egy némileg kifinomultabb eljárással dolgoztam.

Adott megvilágításban egy a keresett színt tartalmazó ellenállásról egy képet készítettem, ezen végeztem a kalibrációt. A küszöbértéket 0-ról növelve a kapott bináris képen figyeltem a legnagyobb összefüggő régió méretét. Ez általában egyenletesen nő egy ideig, ezt egy rövid plató fázis követi, majd tovább kezd növekedni. A plató fázis jelenti azt a küszöbérték-tartományt, amikor a vizsgált színhez tartozó színsáv legtöbb pixele már megjelenik az eredmény képen, más sávok és a háttér képpontjai még nem. Ebből a tartományból választom tehát a threshold első közelítését, és megnézem az azzal nyerhető bináris képet. Ritka esetben ezt még kézzel tovább hangolom.

2.3. Android platform

Az Android egy főként érintőképernyős készülékekre szánt operációs rendszer. Az Android Inc. kezdte a fejlesztést, a Google kezdetben anyagi támogatást nyújtott, majd 2005-ben meg is vásárolta a céget. A publikum 2007-ben kapta meg az első információkat az új mobil platformról. Ezzel egybekötve jelentette be a Google az Open Handset Alliance csoport megalakulását is, aminek tagjai között nagy hardver, szoftver és telekommunikációs cégek találhatók, mint a Samsung Electronics, Texas Instruments, vagy a T-Mobile [7]. A csoport fő céljául az Android alapú készülékek gyártását, illetve az operációs rendszer, és hozzá tartozó alkalmazások fejlesztését és népszerűsítését tűzték ki.

Az operációs rendszer új verzióit a Google fejleszti, és a végső változat elkészülése után a forráskódot bárki számára elérhetővé teszi az Apache licensz [2] alatt. A nyílt forráskód lehetővé teszi a gyártók és mobilszolgáltatók számára, hogy ahelyett, hogy saját maguk költséges fejlesztésbe kezdenének, egy ingyenesen hozzáférhető, kész rendszert használhassanak, amit saját igényeik szerint alakíthatnak.

Emellett a platformra ingyenesen hozzáférhető a hivatalos fejlesztői környezet, aminek segítségével minden érdeklődő nekiláthat a fejlesztésnek [1]. A készülő alkalmazásokat Android készülékeken, vagy akár egy emulátor segítségével is tesztelni lehet. Egy kisebb összeg befizetése után pedig bárki regisztrált Android fejlesztővé válhat, és alkalmazásait pénzért árulhatja a Google által üzemeltetett Play Store online alkalmazás bolt környezetben. Mára több, mint 700 ezer alkalmazás érhető el, és az Android a legnépszerűbb platform a fejlesztők körében.

A fent ismertetett tényezők vezethettek ahhoz, hogy mára a legszélesebb kör-

ben használt mobil platformmá vált. A fejlesztés leggyakrabban a Java nyelv a platformhoz igazított változatán folyik, így sokak számára már ismertek az alapok, illetve beletanulni is gyorsabb. Ez, és az ingyenes fejlesztői környezet a tudományos világban is népszerűvé tette az Androidot – ezért választottam én is.

2.4. Android hardver

A készülék, amin az alkalmazás fejlesztésének java részét végeztem a Samsung Galaxy Tab 10.1 nevű táblagép volt. Ez az eszköz 2011-ben jelent meg, így, figyelembe véve a mobil készülékek piacán tapasztalható nagy ütemű hardveres fejlődést, a hardver jellemzői számadatai nem reprezentatívak a legaktuálisabb készülékek körében. A tesztelést egy újabb, 2012-es tableten, a Galaxy Tab 2 7.0-n végeztem. Röviden bemutatom az eszközök specifikációit, hogy a későbbiekben prezentált sebesség adatokat a hardver számítási kapacitásának kontextusába lehessen helyezni.

A Galaxy Tab 10.1-ben az Nvidia által gyártott Tegra 2 chip a központi egység. Ez egy system on a chip rendszer, ami processzort, RAM-ot, és grafikus feldolgozó egységet egy lapkán tartalmaz. A processzor a mobil eszközökben leggyakrabban használt ARM Cortex sorozat A9-es modellje, egy RISC architektúrájú, több magos, 32 bites egység. Az ARM cég nem gyárt IC-ket, az áramkör terveiket más gyártóknak licenszeli, és a gyártók implementációi között lehetnek kisebb eltérések. Az Nvidia Tegra 2-es implementációja két magos, 1GHz órajelen fut.

A grafikus chip az Nvidia saját ULP (Ultra-Low-Power) GeForce architektúrájú egysége. 333MHz órajelen fut, 8 feldolgozó egységgel rendelkezik. A grafikus csővezeték teljesen programozható az OpenGL ES 2.0 szabványnak megfelel. Utóbbiról a 2.5.1 szekcióban írok részletesen.

A Tegra 2 chip vagy 300MHz-es LPDDR2, vagy 333MHz DDR2 típusú RAM-mal kapható, 1GB felső méretig - Samsung nem hozta nyilvánosságra a pontos típust.

A Galaxy Tab 2 7.0-ban a központi egység a Texas Instruments által gyártott OMAP 4430 system on a chip. A CPU az előző modellel megegyezően az ARM Cortex A9 két magos, 1GHz-es változata.

A grafikus egység a PowerVR SGX540. A PowerVR az Imagination Technologies cég egy részlege, jelenlegi profilja SoC-be integrálható GPU egységek tervezése. Az ARM-hez hasonlóan ők sem gyártanak, hanem licenszelik hardverterveiket. A PowerVR a piac legtöbb másik szereplőjétől eltérően a TBDR (tile-based deferred rendering) technológiát alkalmazza, ami korlátozott renderelő hardver kapacitás esetén jelentősen javítja a teljesítményt – a következő szekcióban (2.5) részletesebben kitérek erre is. Az SGX5-ös széria csővezetéke prog-

ramozható pixel, vertex és geometria árnyalókat is tartalmaz, illetve támogatja az OpenGL ES 2.0-t.

Az OMAP 4430 1GB LPDDR2 RAM-ot tartalmaz, a pontos órajelet itt sem publikálta a Samsung.

A fenti összehasonlításból látszik, hogy a két készülékben számítási kapacitás szempontjából lényegi eltérés csak a grafikus feldolgozó egységben van. Ott, az alapvetően eltérő renderelési eljárás miatt nehéz megjósolni, hogy melyik lesz a győztes – ez ráadásul az adott alkalmazástól is erősen függ.

A következő részben rövid áttekintést adok a grafikus feldolgozó egységek általános felépítéséről, illetve szót ejtek a GPU-k alkalmazásának előnyeiről a képfeldolgozás területén.

2.5. Grafikus feldolgozó egységek

A GPU (graphics processing unit) egy erősen párhuzamos struktúrával rendelkező áramkör, aminek fő feladata a számítógépes grafikában végzett számítások gyorsítása. Klasszikus grafikai feladatok objektumok a három-dimenziós térben való megjelenítése, ezek megvilágítása, árnyalása, és színezése.

A virtuális tér objektumai csúcspontok (vertexek) halmazaként írhatók le. Ha a dolgokat mozgatni szeretnénk a megfigyelő szemszögéhez képest, akkor a vertexeken kell forgatás és eltolás transzformációkat végrehajtani. A mai grafikus alkalmazásokban továbbá alapvető elvárás, hogy a tárgyakat a valószerűbb hatás elérése érdekében részletgazdag textúrákkal lássák el. Ezen alapvető feladatok mellett még sok mást is végeznek a modern grafikában, mint például az egész képeken futtatott simító szűrés az alacsony képfelbontás miatt keletkező esetleges ugráló élek elmosására.

Ezen eljárások szoftveres megvalósításában igen sokszor fordulnak elő mátrixokon és vektorokon végzett műveletek, interpolációs feladatok illetve képek adott pontjainak memóriából történő olvasása. Így a grafikus processzorok úgy fejlődtek, hogy az ilyen műveleteket minél nagyobb sebességgel tudják végezni. Hardveresen huzalozott bennük sok mátrixalgebrai és egyéb matematikai, illetve textúra mintavételezési művelet.

Továbbá, mivel a grafikában végeredményül egy képet kell előállítani, és a folyamat nagy része képpontonként végzendő, a lehető legnagyobb sebesség elérésére érdekében fontos, hogy egyszerre minél több képponton történjen munka. Ezért a GPU-kban általában nagy számú feldolgozó egység található, amik egymással megegyező számítási teljesítménnyel rendelkeznek. Ezek csoportokra oszlanak, egy csoport ugyanazon program futtatására képes egységenként eltérő bemeneteken – ez a SIMD (single instruction multiple data) adatfolyam feldolgozási

struktúra. A programokat, amiket egy-egy csoport futtat, árnyalóknak (shader) nevezik.

A grafikus feldolgozó egységek működési elve az előbbiek alapján tehát olyan, ami jól adaptálhatóvá teszi őket sok képfeldolgozási feladat elvégzésére. A képfeldolgozásban használt paraméterek átadhatók a GPU-nak, mint vertex, a kép maga pedig, mint textúra. Ezeken a bemeneteken pedig legtöbbször a gépi látásban is vektor- és mátrixműveletekre épülő beavatkozásokat kell végrehajtani.

A grafikus processzorok programozására több elterjedt megoldást találunk a mai piacon. Ilyen a Microsoft által fejlesztett DirectX, és a hozzá tartozó HLSL (high level shading language), amit csak Microsoft környezetekben használhatunk (Windows, XBox, Windows Phone), illetve a platformfüggetlenségre törekvő OpenGL és árnyaló nyelve, a GLSL.

2.5.1. OpenGL ES

Az OpenGL-t a Silicon Graphics cég kezdte fejleszteni 1991-ben, mára széles körben elterjedt. Egy több nyelvet és platformot támogató API (application programming interface), ami képes grafikus feldolgozó egységek vezérlésére. Fő célja a két- és háromdimenziós renderelés támogatása. Hozzá tartozik a GLSL árnyaló nyelv, amiben a GPU-n futtatandó shadereket írhatjuk meg. A GPU programokat a gazdaalkalmazásban végzett OpenGL hívásokkal fordíthatjuk, tölthetjük fel a grafikus chipre, illetve kommunikálhatunk velük.

Android platformon jelenleg az egyetlen mód a készülékek grafikus processzorának közvetlen programozására az OpenGL ES (for Embedded Systems) API használata. Az OpenGL ES a nagy testvér OpenGL API beágyazott rendszerekre szánt változata, bizonyos megkötéseket, korlátozásokat tartalmaz ahhoz képest. 2.0-s változata már támogatja a felhasználói árnyaló programok fordítását és futtatását, ez pedig az Android a 3.0-s változata óta tartalmazza. Megjegyzem, hogy a Google dolgozik egy másik, általános számítási célokra használható Android API, a RenderScript olyan irányú fejlesztésén, hogy az majd a GPU-t is kihasználva működjön, azonban egyelőre ez még nem alternatíva.

Egy OpenGL ES program legalább két árnyalót kell tartalmazzon, a vertex és a fragment shadert. Az első tradicionális feladata a megjelenítendő objektumok megfelelő helyre mozgatása, forgatása, és a kamera szemszögébe való vetítése. Bemenetei tehát főként vertexek, kimenetként pedig ezek vetületét adja meg.

A fragmens árnyaló a vertex shader által kapott vertex pozíciókat használja fel, hogy az objektumokat ellássa textúrákkal, és meghatározza a kimeneti kép egy-egy pixelének színét. Mivel ez állítja elő a kimenő képpontokat, képfeldolgozási alkalmazás esetén a feladatok nagy részét ebben az árnyalóban végezzük.

2.6. OpenCV függvénykönyvtár

Képfeldolgozási területen manapság szinte elkerülhetetlen, hogy ne találkozzunk az OpenCV (Open Source Computer Vision Library) névvel. Egy programozási függvénykönyvtárról van szó, fő célja a valós idejű gépi látás alkalmazások támogatása. Fejlesztését az Intel kezdte 1999-ben, amibe a Willow Garage nevű robotikával foglalkozó cég is becsatlakozott. A könyvtár szabadon felhasználható, nyílt forráskódú. Magja C++-ban készül, és a legkényelmesebben ezen a nyelven használható. Ugyanakkor elérhető már Python, Java, és Matlab nyelveken is használható interfész hozzá. Továbbá kifejezetten az Android platfomra is készült egy változata, itt Android natív alkalmazás esetén a C++ metódusokat lehet használni, Java fejlesztéskor pedig a Java interfészt.

A könyvtár tartalmaz függvényeket a képfeldolgozás klasszikus feladatainak ellátására, illetve kész megoldást kínál a legújabb eljárások (pl. feature pontok keresése ORB algoritmussal) implementálására is. Ezért az Android alapú fejlesztés során gyakran támaszkodtam rá.

2.7. Ellenállás kódolás

Mint ismeretes, a furatszerelt típusú ellenállások névleges értékéről, toleranciájáról, illetve bizonyos esetekben még a hőmérsékleti együtthatójáról és megbízhatóságáról az ellenállások borítására felvitt színes gyűrűk adnak információt. A gyűrűk színe és elhelyezkedése együttesen egy kódot alkot, amit visszafejtve kaphatjuk meg a névleges értékeket. A jelöléseket általánosatt elfogadott előírások rögzítik, amiket még az 1920-as években dolgoztak ki. A jelenlegi nemzetközi szabvány az IEC 60062 [5] [1]. A szabvány az ellenállásokról való rendelkezéseit röviden összefoglalom.

Szabványosan 12 különböző szín szerepelhet a kódolásban. Ezek közül 10 homogén színréteg, mellettük pedig előfordulhat ezüst és arany gyűrű is. Az utóbbi, fémes hatást keltő színeket általában úgy valósítják meg, hogy apró fémszemcséket tartalmazó festéket használnak. Mindegyik szín többféle jelentéssel bírhat, a többi gyűrűhöz képest való elhelyezkedésétől függően. A kód csak egyik irányból leolvasva értelmezhető, azt, hogy melyik gyűrűnél kell kezdeni, a gyűrűk között levő távolság alapján lehet eldönteni. Az „utolsó” gyűrű az, amelyik távolabb van szomszédjától, mint az „első”. A színkód állhat négy, öt vagy hat gyűrűből, a több gyűrű értelemszerűen több információt kódol.

A gyakorlatban leggyakrabban felmerülő kérdés az ellenállás névleges értéke. Négy gyűrűs kód esetén az első két gyűrűt számértéken kell venni, egymás után

[1]A furatszerelt kapacitások is hasonló elven működő kódolást használnak, feladatomban csak ellenállásokkal foglalkozom.

írva őket egy 10 és 99 közötti v számot adnak ki. A harmadik gyűrű m értéke egy szorzót jelent. A névleges ellenállás érték Ohm-ban a v^m képlettel kapható meg. Öt- és hatgyűrűs kódolás esetén v-t az első három gyűrű adja (így 100 és 999 között vehet fel értéket), m pedig a negyedik gyűrű.

Négy- és ötsávos esetben az utolsó szín kódolja a toleranciát. Hat gyűrű esetén a hatodik szín ráadásként a hőmérsékleti együtthatót, vagy ritkább esetben (főként katonai alkalmazásoknál) a megbízhatóságot.

Az aranyat és ezüstöt kivéve a festékek homogének, így ideális esetben, szórt fénnyel megvilágítva a gyűrűk teljesen egyszínűnek látszanak. Gyakorlatban a fény gyakran tartalmaz egy erősen irányított komponenst, ami, mivel az ellenállások tokja henger alakú, elrontja a teljes homogenitást, a henger kerülete mentén haladva a szín változik, illetve megjelenhetnek becsillanások a tokozás enyhén tükröződő anyaga miatt.

Ezekben az esetekben a színgyűrűk helyes detektálása némileg összetettebb feladat. A probléma megfelelő arra, hogy rajta keresztül megvizsgáljuk, mennyire alkalmas egy színszegmentálási algoritmus valós körülmények között mesterségesen színezett objektumok elkülönítésére. Ilyen objektum lehet a mobilis robotikában, és számtalan más területen használt marker, amit előre elhelyeznek a munkatérben, vagy egy mozgó objektumon (pl. robot).

Továbbá a furatszerelt ellenállások felismerése klasszikus feladat a villamosmérnöki gyakorlatban. A színkód elsajátítása némi gyakorlatot igényel, így a területen kisebb tapasztalattal rendelkező mérnökök, hallgatók számára segítséget nyújthat egy képfeldolgozás-alapú detektálás. Megjegyzendő, hogy a furatszerelt ellenállások használata mára jellemzően már csak az oktatásban, illetve hobbi projektekben fordul elő, ahol a felhasználók általában pont a fenti kategóriák egyikébe esnek. Emellett a színtévesztők számára az egyetlen alternatíva az elektronikus mérés, ami egy áramkörbe szerelt ellenállás esetén nem mindig lehetséges.

A fenti megfontolások alapján választottam a színkód felismerését a tesztelés alapjául.

3. fejezet

Az algoritmus Android implementációja

Ebben a fejezetben lépésről lépésre bemutatom a megvalósított Android alkalmazás működését, és ezen keresztül az algoritmus lépéseit. Esetenként prezentálok alternatívákat, és indoklást, hogy végül miért nem azok mellett döntöttem. A legtöbb művelet végzésekor a nagy sebesség volt elsődleges szempontom, így sebességre jellemző adatokkal is szolgálok.

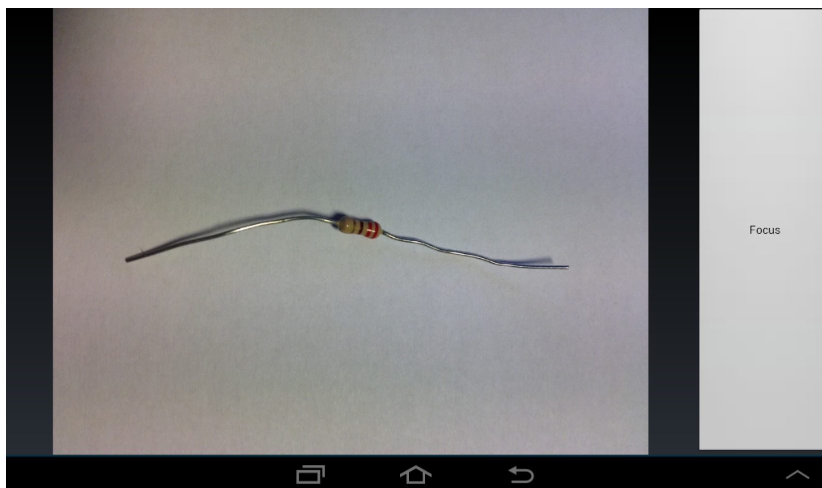
3.1. Felhasználói interakció

Az alkalmazás elindítása után egy egyszerű felület fogad minket, aminek nagy részét a készülék hátoldali kamerájából származó képfolyam tölti ki. Emellett található egy gomb, ami segítségével automatikus fókuszbeállítást kérhet a felhasználó az éppen mutatott képre. A detektálás megkezdése előtt az ellenállást a kamera látóterébe kell hozni, az élességet az autófókusz segítségével beállítani, végül pedig meg kell érinteni az ellenállást a képernyőn. A 3.1 ábra mutatja ezt a képernyőt. Ez után az azonosítás lefut, és az alkalmazás prezentálja az eredményeket, esetleges hibákat. A felület és működés ilyen kialakítását a hardver jellegzetességek, illetve algoritmikus megfontolások miatt választottam. Ezeket mutatom be a következőkben.

3.2. Az ellenállás testének azonosítása

A programban implementált színszegmentálási eljárás akár a nyers erő módszerével az egész kameraképre futtatható, és eredményes, amennyiben a lehetséges kódszínek csak ténylegesen az ellenálláson fordulnak elő. Ez egy olyan alkalmazásban, ahol a környezet színeitől élesen elütő színű markerek felismerése a

3.1. ábra: A felhasználói felület



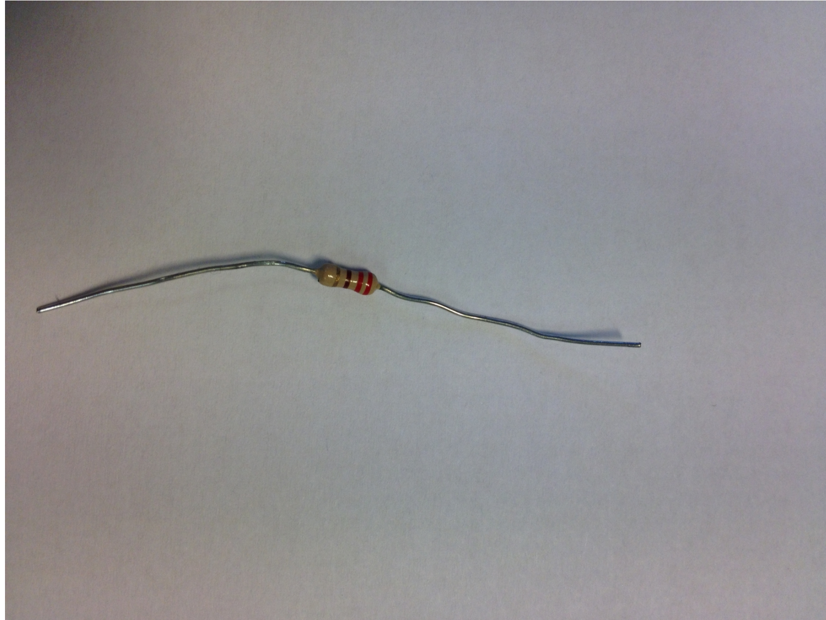
cél, megfelelő. Ugyanakkor ezt a megkötést túlságosan korlátozónak találtam a mostani alkalmazásban – elképzelhető, hogy más ellenállások, vagy hasonló színkóddal rendelkező áramköri elemek is jelen vannak, – illetve, mivel pixel-alapú a színszegmentálás, a felbontással arányos a futási idő. Ezen megfontolások alapján úgy döntöttem, hogy szükséges egy előfeldolgozó lépés, ami kijelöli az ellenállás testét a képen, hogy a következő lépések már csak a hasznos, lényegesen kisebb pixelmennyiséget jelentő területen dolgozzanak.

3.2.1. Detektálás indítása, kép vágása

Mivel az ellenállás elkülönítése a képen nem kapcsolódik szorosan a színszegmentáláshoz, így bizonyos előfeltételek teljesülését megkívánó, egyszerűbb megoldást választottam. A képről tehát feltételezem, hogy az ellenállás környezetében homogén, lehetőleg semleges (alacsony telítettség) színű. Ezek a követelmények tökéletesen teljesülnek abban az esetben, ha egy fehér papírlapra helyezzük az ellenállást. Ilyen elrendezést használtam a fejlesztés során, és a tesztelés nagy részét is így végeztem. Ugyanakkor ha a simaság és alacsony szaturáció követelmények teljesülnek a háttérre, nem szükséges a teljesen fehér háttér. A 3.2 ábrán egy, a teszteléshez használt környezetben készült kiindulási fotó látható, a további lépéseknél mutatott az ezen végzett műveletek eredményét mutatják.

Mivel az algoritmus indítása az alkalmazásban felhasználói beavatkozáshoz kötött, a beavatkozásból származó információt is kihasználom. A tapintás pillanatában a program egy képet készít, a továbbiakban ezen dolgozik. Mielőtt bármilyen feldolgozás elindulna, a kép területének $1/8$-ára vág a program. A kivágott kép arányai megegyeznek az eredetiével, szélessége és magassága az eredeti

3.2. ábra: Egy kiindulási fotó



negyede, a középpontja pedig ott van, ahova a felhasználó tapintott. A vágási arányokat empirikus úton kaptam. Az átlagos mobil készülékek ha még kellően éles (nem túl közeli), ugyanakkor még elegendően sok pixelből álló (nem túl távoli) képet adnak egy ellenállásról, akkor az minden általam próbált készülék esetében belefér a fotó egynyolcadába. (Ilyen készülékek voltak az Apple iPhone 4, 4S, Samsung Galaxy Tab 10.1, Galaxy Tab 2 7.0).

3.2.2. Éldetektálás és küszöbözés

Miután elkészült a levágott kép, azt feltételezem, hogy azon csak egy ellenállás, és a homogén háttér szerepel. Így a következő eljárást alkalmazom.

3.2.2.1. S csatorna küszöbözés

Elsőként az RGB képből előállítok egy HSV színtérbeli másolatot. A homogén, semleges színű háttér kritérium miatt az S (szín telítettség) csatornán jól elkülöníthető az ellenállás a háttértől. Az S csatornát így küszöbözöm egy előre, empirikusan meghatározott küszöbérték segítségével, így előáll egy bináris maszk.

Ugyanakkor sokféle zavaró tényező miatt elképzelhető, hogy a szaturáció bizonyos helyeken a háttérben is magasabb lesz. A legtöbb gondot a megvilágítás változása jelentette. A jó kép készítéséhez a készüléket gyakran igen közel kellett vinni a céltárgyhoz, ez pedig általában nagy árnyékfoltok megjelenését ered-

ményezte. Ezek a foltok esetenként barnás, vöröses színűek, és megközelítik az ellenállás szaturáció értékeit.

3.2.2.2. Éldetektálás

Ahhoz, hogy az S csatorna küszöbözést robusztusabbá tegyem, éldetektálást is végrehajtok a képen. Az RGB képet szürkeárnyalatosítom, majd X és Y irányú Sobel kernellel szűrök, és a két eredmény 2-es normáját veszem (a végeredmény egy 2 dimenziós Sobel szűrés). Az ilyen módon szűrt képen előre meghatározott értékkel küszöbözést hajtok végre, ami eredményül egy másik bináris maszkot ad. Ezen a maszkon az árnyékokból származó, lágyabb körvonalú foltok nem jelennek meg.

Az élkeresés és küszöbözés után elkészült képen, ha a háttér megfelelően sima, csak az ellenállás test és a vezetékek körvonalai, illetve néha a színsávok széle és egyéb, megvilágítás miatt előálló hibák látszanak. Ez a kép ilyen formában még nem alkalmas a bináris maszkként való felhasználásra, hiszen az ellenállás testének egészét nem fedi le.

Ennek megoldására fontolóra vettem valamilyen kontúrkereső algoritmust. A legegyszerűbb ilyen módszerek a képen levő zárt görbéket megkeresik és kitöltik. Ha ezek rendelkezésre állnának, egyszerű lenne a kontúrok által befoglalt területet kitölteni, így jól használható maszkot kapnánk.

Az OpenCV-ben rendelkezésünkre áll egy egyszerű kontúrkereső eljárás a `findContours` metódus formájában. Ez összefüggő zárt görbéket keres, és visszaadja azok külső kontúrját. A jelen esetben ez nem mindig elegendő, mert az esetek túlnyomó részében éldetektálás után az ellenállás kontúrja nem folytonos, a külső kontúrok hiányoznak, a színsávok peremei dominánsabbak inkább. Ráadásul a Sobel operátor a háttérben levő apró zajok, egyenetlenségek miatt sok kisebb, só és bors jellegű artifaktot eredményez, aminek kiszűrése további feladatot jelent kontúrozás esetén.

A háttérben tévesen detektált éleket nagyban csökkenthetjük, ha az összetettebb Canny élkereső algoritmust alkalmazzuk. Ugyanakkor ennek futási ideje némileg magasabb, és a két küszöbérték hangolása új járulékos problémát jelent. Az ellenállás kontúrjának folytonosságát hosszasabb kézi hangolás ellenére ez az eljárás sem tudta garantálni.

A kontúrkereső eljárások egyszerű implementációja tehát előfeldolgozás nélküli éldetektált képen nem használható eredményt ad. A morfológiai operátorok kínálnak egy egyszerűbben implementálható, végeredményben lényegesen jobb megoldást.

3.2.2.3. Morfológiai utófeldolgozás

A morfológiai zárás művelete (dilatációkat követő eróziók alkalmazása, lásd a 2.1.3 fejezet) pontosan ilyen helyzetekben alkalmazandó, amikor a nagyobb összefüggő objektumokban kisebb kitöltetlen régiók vannak, illetve kontúrjuk nem folytonos.

A morfológiai műveletek hatékonyságát alapvetően meghatározza, hogy mekkora a kép felbontása, hány pixelből állnak az összefüggő régiók, és hogy mekkora, illetve milyen elrendezésű kernelt használunk. Egy bizonyos felbontásnál adott méretű kernellel történő erózió eltüntet kisebb zajokat, míg a nagyobb blobokat (összefüggő objektumokat) csak karcsúsítja. Ha a kép felbontását csökkentjük, ugyanakkora kernel már a nagyobb, számunkra értékes információt hordozó blobokat is eltörölheti.

Ezt a problémát ki lehet küszöbölni a kernelméret képfelbontástól függő dinamikus megválasztásával, vagy a képfelbontás rögzítésével. Az utóbbi megoldás más előnyökkel is jár a jelenlegi feladatban. Ugyan az általam használt készülékek kamerája 3 megapixeles, más eszközök képesek ennél lényegesen nagyobb adatsűrűségű, 8 megapixeles képek készítésére is. Amennyiben csak a kép negyedére vágok, 8 megapixeles esetben a számítási idő – pixel/régió alapú algoritmusokról lévén szó – durva közelítéssel $8/3 = 2\frac{2}{3}$-ára nő. A fix felbontás beállítása így az algoritmus ezen részének futási idejét is determinisztikussá teszi.

Tesztelés során meghatároztam tehát egy olyan felbontást, ami minimális méretű, ám még elég pixelt tartalmaz ahhoz, hogy a fent említett műveletek segítségével az ellenállás testét be lehessen azonosítani. A méret normalizálást a küszöbözési műveletek előtt elvégzem, de megtartom az eredeti felbontású képet is. A felbontáshoz a morfológiai kernelméretet is rögzítettem, az értékeket tapasztalati úton kaptam.

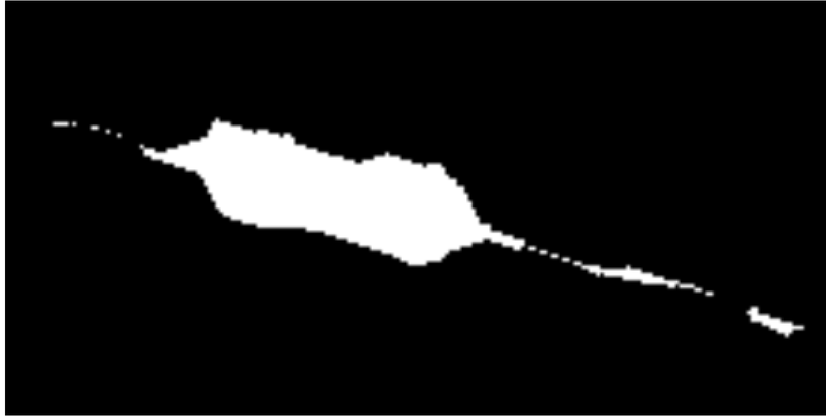
A méret normalizálás után tehát biztonsággal használhatok morfológiai zárást. Ezt mind a szaturáció-küszöbözött, mind az éldetektált-küszöbözött képre elvégzem.

A maszkok az esetek nagy részében csak olyan hibákat tartalmaznak ekkor, amik csak az egyikben fordulnak elő. Erre alapozva az ellenállást lefedő bináris képek a két maszk pixelenkénti logikai **ÉS** műveletével kapom. Ennek eredménye a 3.3 ábrán látható.

3.2.3. A vezetékek leválasztása

Az előző részben ismertetett lépések az előkövetelményeknek megfelelő képekre egy kétszintű képet adnak, ahol az aktív pixelek lefedik az ellenállás testét, és a fém kivezetéseket. A feladat szempontjából kitüntetett régió nem tartalmazza a vezetékeket, így azokat valamilyen úton el kell távolítani.

3.3. ábra: A maszkok kombinációja



A furatszerelt ellenállásokról általánosan elmondható, hogy a test átmérője lényegesen nagyobb, mint a huzaloké, ami a bináris maszkon is megjelenik.

E megfigyelésre alapozva a vezetékek eltávolítására végső erózió (ultimate erosion) alkalmazása mellett döntöttem. Az algoritmus egy FIFO pufferben tárolja az egymás után végzett eróziók eredményeit. Amikor egy, esetenként változó p lépés alatt az összes aktív pixel eltűnik a képről, a program elővesz a FIFO puffer m-edik elemét. Ezen az elemen $p-m$ dilatációt hajt az erózióknál használt kernellel.

Ennél az eljárásnál is kiemelten fontos, hogy a kernelméret és képfelbontás rögzített legyen. Az m konstanst is ezekhez határoztam meg.

Az eddigi lépések eredménye egy bináris maszk, ami az ellenállás testét fedi le (a 3.4 ábrán látható). Ezt visszaskálázom a méretcsökkentés előtti képfelbontásra, hogy az eredeti kép és a kétszintű kép pixelei egy-az-egyhez megfeleltethetők lehessenek. Ez után a korábban említett `findContours` (lásd a 3.2.2.2 szekciót) utasítással könnyen kinyerhető a test kontúrja, és a kontúrt befoglaló legkisebb téglalap, az úgynevezett bounding box.

Mind az eredeti, mind a bináris képet az így kapott bounding box méreteire vágom, majd a színes kép pixeleit csatornánként megszorzom a bináris kép pixeleivel. Az implementáció olyan, hogy a maszkban az aktív pixelek intenzitás értéke 1, míg a háttéré 0, így az eredmény egy olyan kép, ahol a nem az ellenállás testéhez tartozó képpontok $\{0, 0, 0\}$ RGB értékűek, a testhez tartozók pedig megőrzik eredeti színüket.

3.4. ábra: A végső erózió utáni maszk

3.3. A színsávok detektálása

Miután rendelkezésre áll az előzőekben megkapott, csak a tokozást tartalmazó kép, a következő lépés a a színsávok felismerése. Ez a rész szolgál tehát a 2.2 fejezetben kiválasztott színszegmentálási algoritmus próbájaként.

Ennek az algoritmusnak a legfontosabb lépése a Mahalanobis-távolság számítása – két megoldást teszteltem.

3.3.1. Méret normalizálás

Ahhoz, hogy determinisztikusan megkapható legyen a futási idő, ennél a lépésnél is fontos, hogy fix mennyiségű pixelen kelljen az algoritmust futtatni (hiszen pixel-alapú a feldolgozás). Az ellenállás tokját tartalmazó kép képaránya tetszőleges lehet a tok képen való orientációjának, és tényleges méretének függvényében. Ezért nem egy fix felbontásra normalizálok, hanem pixelszámra. Ezt az N pixelszámot empirikus módon határoztam meg.

A skálázás úgy zajlik, hogy kiszámolom az $r = \frac{N}{M}$ arányt, ahol M a maszkolt ellenállás kép pixeleinek száma. Majd $\sqrt{r}$-rel szorzom mind a függőleges, mind a vízszintes felbontását. A végeredmény egy olyan kép, ami legfeljebb az új függőleges és vízszintes felbontás maximumával tér el az előírt pixelszámtól – a kerekítési hibák miatt.

3.3.2. OpenCV implementáció

Az OpenCV tartalmaz egy Mahalanobis-távolság számítását elvégző függvényt (`cvMahal`), ami az $\mathbf{x}$ adatpontot, az adathalmaz μ átlagát, és a Σ^{-1}-et, a kovariancia mátrix inverzét várja bemenő paraméterül. A színsávok leíró adatait, mint említettem, offline módon, Matalab környezetben állítom elő, ezeket az Android alkalmazás szöveges erőforrásként (resource) kapja meg. A leírókat előre olyan alakra lehet hozni, ami az online futást a leggyorsabbá teszi. Tehát a kovariancia mátrix invertálását már Matlab alatt elvégzem.

Ez az implementáció a `cvMahal` függvényt a maszkolt ellenállás tokot tartalmazó kép minden pixelére 10-szer le kell futtassa, hogy mindegyik lehetséges homogén színről bináris maszkot készítsen. Ez az OpenCV Androidos implementációjában csak szekvenciális futtatással lehet, a kód a CPU-n fog futni.

Egyetlen színsávra való futtatást teszteltem először. A futási időt az Android platform `SytemClock` beépített osztályának `uptimeMillis()` függvényének segítségével mértem. Ez milliszekundumban visszaadja a rendszer indítása óta eltelt időt. A számítások megkezdése előtt lekérem ezt a t értéket, befejezés után pedig egy új, t_{finish} számot, és a t_{run} futási időt $t_{run} = t_{finish} - t$ módon számolom.

A Mahalanobis-távolság OpenCV-s számítása igényel néhány előkészítő lépést, mint a paraméterek és a kép `Mat` osztályú segédváltozókba való betöltését (lásd **??**, ezeket azonban nem vettem bele a számítási időbe. Továbbá minden háttérben futó alkalmazást leállítottam. A tesztet az Android 4.1.1-es verzióján végeztem.

Azt tapasztaltam, hogy egy színsávra a távolság számítása átlagosan 3 másodpercet vesz igénybe. Tíz mérés alapján 2938ms átlagértéket kaptam mindössze 35ms szórással. A szórás jelenlétét okozhatja a korábban említett nem maximálisan pontos méret normalizálás, illetve az Android feladatütemezőjének nem determinisztikus működése.

Több lehetőség is kínálkozik a futási idő csökkentésére. A legkézenfekvőbb módszer a pixelszám csökkentése, hiszen egy minimális overheadet leszámítva a futás egyenesen arányos a pixelek számával. Továbbá az OpenCV Java API-ját használom, ez minden OpenCV függvényhívásnál további járulékos terhelést jelent, amíg a Java osztályokból különböző segédfüggvények átalakítják a bemenő paramétereket natív C++ kompatibilis adatokká.

Az általam kitűzött sebesség cél azonban ennél több, mint egy nagyságrenddel kisebb. A jelenlegi, ellenállás felismerési alkalmazásban azt szeretném, ha egy hagyományos elektronikus mérés sebességét legalább is elérje az alkalmazás, általános esetben pedig milliszekundum nagyságrendű futási idő lenne kedvező, hogy az algoritmusra épülő magasabb szintű feldolgozásnak minél több idő jusson.

Ezen megfontolások alapján egy alapvetően más implementációs megoldást próbáltam ki, az OpenCV irányban nem tettem további lépéseket.

3.3.3. OpenGL implementáció

Az Android készülékek jelenleg elérhető kínálatában nem található olyan, amelyik ne tartalmazna grafikus segédprocesszort. Ez általánosságban is elmondható a mobilkészülékek terén, a többi jelentős platform, mint az Apple iOS-alapú készülékei, vagy a Windows Phone is mind alapfelszereltség szinten rendelkezik hardveres grafikai egységgel.

Az OpenGL ES 2.0 lehetőséget nyújt a GPU-k általános célú kihasználására saját árnyaló programok írásával, és mind Android, mind Apple környezetben több éve támogatott. (Bővebben lásd a 2.5.1 fejezetet.)

A pixel alapú algoritmusok alapvető jellegük miatt kiváló jelöltek a párhuzamosításra. Továbbá a Mahalanobis-távolság számítása vektor és mátrixszorzásokból épül fel, amik a tradicionális számítógépes grafikai feladatok alapkövei, így a GPU-k ezek elvégzését huzalozott célhardver modulokkal gyorsítják.

E megfontolások alapján elkészítettem egy, a Mahalanobis-távolság számítását és küszöbözést is elvégző OpenGL ES programot.

3.3.3.1. Android jellegzetességek

Az Android az OpenGL használatát egy külön nézetosztályhoz köti, a `GLSurfaceView`-hez. Ez egy, a felhasználói felületbe építhető elem, amin alapértelmezett esetben a GPU-s renderelés eredménye megjelenik. Ez az osztály továbbá automatikusan létrehozza és kezeli az OpenGL kontextust, ahol az OpenGL-nek adhatók parancsok egy Java API-on keresztül. Az osztály működése legkönnyebben úgy szabható testre, ha egy leszármazott osztályt írunk, és felüldefiniáljuk az örökölt metódusokat – így jártam el én is.

Társítani kell hozzá egy `GLSurfaceView.Renderer` osztályt (vagy leszármazottját), ami különböző callback metódusok segítségével előkészíti az OpenGL környezetet, és vezérli a program végrehajtását. Az `onDrawFrame` callback függvényben kell megvalósítani a megjelenítő program futtatását. Ez történhet folyamatosan, ilyenkor ez a metódus visszatérte után azonnal újra meghívódik. A másik lehetőség a manuális meghívás, ilyenkor a `GLSurfaceView` osztályból (vagy származtatottjából) kell indítani egy render passt a `requestRender()` hívással.

Ez a keretrendszer hagyományos grafikai alkalmazásokhoz illeszkedik inkább, azt feltételezi, hogy a renderelés eredményét a felhasználó számára meg akarjuk jeleníteni. Több olyan hívást, memóriafoglalást tartalmaz a környezet felállítása,

ami általános célú GPU felhasználás esetében szükségtelen, felesleges többletterhelést jelent. Ez úgy hidalható át, ha kézi úton kezeljük az OpenGL kontextust, nem támaszkodunk a `GLSurfaceView` osztály által nyújtott kényelmi funkciókra.

Ilyen irányú optimalizálással a végső megoldásban nem foglalkoztam, a többletterhelés tehát jelen van a programban. Ugyanakkor az inicializálás lehető legtöbb részét igyekeztem az alkalmazás indulásakor elvégezni, így az overhead túlnyomó része csak egyszer jelentkezik ott, ahol a felhasználók hagyományosan hozzá vannak szokva egy hosszabb idejű várakozáshoz. Ez az időtöbblet egyébként igen alacsony, 150-200ms alatt befejeződik a `GLSurfaceView` leszármazott osztálynak és járulékos osztályainak adott képtől független inicializálása.

3.3.3.2. Az OpenGL program

Amint tehát előáll a normalizált méretű maszkolt kép az ellenállás tokjáról, a program futása átvált az OpenGL kontextusra. Az addig az OpenCV saját `Mat` osztályú objektumában tárolt képet az Android általános `Bitmap` osztályába alakítom, majd ezt feltöltöm a GPU memóriába egy `GL_TEXTURE_2D` textúra objektumba. Itt megjegyzem, hogy amennyiben a betöltött textúra vízszintes vagy függőleges mérete nem páros számú pixel volt, akkor a kimenetül kapott kép torz lett a Galaxy Tab 10.1 készüléken. A képpont mennyiség normalizálásakor így erre is figyelnem kell, szükség esetén egy-egy pixellel megtoldom a méreteket.

Mivel nem akarom megjeleníteni a renderelés eredményét – a bináris kép nem hordoz hasznos információt a felhasználónak, aki csak az ellenállás értékét szeretné tudni – létrehozok és beállítok egy framebuffer objektumot is. Ehhez kötök egy `GL_TEXTURE_2D` objektumot, amibe a számítás eredménye fog íródni.

3.3.3.2.1. Vertex árnyaló. Az árnyaló programban a vertex és fragment shadereket használom. A vertex árnyaló bemenő vertex attribútumaként beadok egy négy elemből álló háromszög strip-et, ami lefedi a teljes látóteret. A betáplált vertexek koordinátái tehát a $\{-1;-1\}, \{-1;1\}, \{1;-1\}$ és $\{1;1\}$. Ugyanezeket a koordinátákat textúra koordinátaként is átadom a vertex shadernek, amiket a program hardveres interpoláció segítségével számít át a fragmenseknek megfelelő értékekre. A vertex árnyaló hagyományos feladata a térbeli transzformációk elvégzése: a megjelenítendő modell világkoordinátákhoz képesti pozíciójának és orientációjának beállítása, továbbá a nézőpont és leképezés specifikálása. Mivel egy kétdimenziós képet szeretnék megjeleníteni, a modell-nézet-projekció transzformációm konstans identitás mátrix. Az ezzel való szorzástól eltekintek, a bemenő paramétereket egyszerűen továbbadom interpolálásra.

3.3.3.2.2. Fragmens árnyaló. A tényleges távolság számítást és küszöbözést a fragmens árnyalóban végzem. Tíz féle színre kell előállítanom bináris képet, ezt célszerű lenne minél kevesebb render pass alatt elvégezni. Az OpenGL ES 2.0 implementációjában készülékről készülékre lehetnek bizonyos eltérések, az azonban minden esetben szabvány szinten elvárt, hogy a framebuffer `GL_COLOR_ATTACHMENT0` típusú kötési pontján levő textúra objektumból visszaolvashatók kell legyenek az adatok. Más kötési pontok (stencil és mélység) hozzáférése az ES implementációban limitált, így ezekkel nem próbálkoztam.

Ez azt jelenti, hogy fragmens shaderemben a szabványos `gl_FragColor` kimeneten keresztül négy lebegőpontos számot tudok beírni a framebufferhez kötött textúra megfelelő koordinátájába – ezek hagyományosan egy $RGBA$ (itt az A az alpha, „áttetszőség” csatorna) kép csatornáinak értékét adják. Tehát ha 10 bináris képre van szükségem, akkor ezek három pass alatt előállíthatók, kétszer mind a négy kimenet, egyszer pedig csak kettő felhasználásával.

A három futtatás között a bemenő paramétereknek változni kell. Ugyanakkor OpenGL-ben egy új program fordítása és betöltése overheadet jelent, így ezt kerültem a megoldásomban. A futtatásonként változó paraméterek betöltésére kézenfekvő megoldást nyújtanak az `uniform` típusú bemenő változók. Ilyen bemeneten keresztül adom át az aktuálisan keresett 2-4 színre jellemző μ átlagértéket és Σ^{-1} inverz kovariancia mátrixot és a színekhez tartozó küszöbértékeket is.

A feldolgozandó kép egy `sampler2D` típusú uniform bemeneten érkezik, amit az inicializáláskor létrehozott textúra objektumra állítok. Ez futásonként értelemszerűen nem változik. Az adott fragmensben feldolgozandó $\mathbf{x}$ pixel értéket a `texture2D()` OpenGL függvénnyel kapom, aminek egyik bemenete a textúra objektum, másik pedig a vertex shaderből érkező interpolált textúra koordináta.

Mielőtt az árnyaló elvégezné a távolság számítását, megvizsgálja, hogy az adott pixel nem teljesen fekete-e (azaz nem $\{0; 0; 0\}$ értékű). Ez ugyanis korábban ismertetett maszkolás (3.2.3 fejezet) mellett azt jelenti, hogy az a pixel nem tartozik az ellenállás tokjához. Valós képen ilyen értékű képpont a legritkább esetben fordulhat elő, a gyakorlatban ez a módszer tehát nem okoz tévesztést. A távolságszámítás csak akkor fut le, ha a pixel az ellenálláshoz tartozik, egyébként a kimenet automatikusan inaktív értékű. Megjegyzem, hogy a GPU program párhuzamos futása miatt ez az ellenőrzés nem okoz sebességnövekedést. Azok a szálak, amik maszkolt pixelekkel foglalkoznak ugyan hamarabb lefutnak, de a futási idő a leglassabb szál sebességétől függ, ami nyilvánvalóan egy távolságszámítást végző thread lesz.

A Mahalanobis-távolság meghatározását három részre bontottam fel (emlékeztetőként a 2.5 egyenlet tartalmazza az implementálandó kifejezést). Első lépésben kiszámítom a $\delta = \mathbf{x} - \mu$ értéket, a kifejezés mindhárom tagja három elemű vektor. Ez után elvégzem a $k = \delta\Sigma^{-1}$ szorzást, ahol k szintén három elemű.

A végeredmény pedig $k \cdot \delta$ módon adódik, $\cdot$ a skaláris szorzatot jelenti. Ezek a számítási lépések – vektor-mátrix szorzat, vektor különbség képzés, skaláris szorzat – a GPU-ban hardveresen támogatottak, az utóbbi kettő egyetlen órajel alatt fut. Megjegyzem, hogy ez nem pontosan a 2.5 egyenletet valósítja meg, mivel hiányzik a gyökvonás. Ahelyett ugyanis, hogy az online futást terhelném, az előre meghatározott küszöbérték négyzetét égetem be a programba, így a négyzetgyök megspórolható.

A megkapott távolságokat összehasonlítom a küszöbértékkel, amennyiben kisebbek annál, a megfelelő kimeneti csatornát aktív értékre állítom.

3.3.3.2.3. Sebesség. A fő ok, amiért a GPU alapú implementációt elvégeztem a nagyobb futási sebesség elérése volt. A sebességmérést az OpenCV-nél ismertetett módszerhez hasonlóan végzem, egy kiegészítéssel. Az OpenGL irányába történő API hívások az adatok átadása után azonnal visszatérnek, így egy renderelési hívás futási idejét mérve nem a valódi sebességet kapjuk. Az OpenGL azonban rendelkezik egy barrier-jellegű utasítással, ez `glFinish()`. Csak akkor tér vissza, amikor az összes elindított OpenGL hívás lefutott.

A mérések alapján a három pass eltérő sebességgel hajtódik végre. Az első futtatás átlagosan 21ms alatt lezajlik, 6ms szórással. A második kettőre pedig 0-2ms közötti értékeket kaptam. Ezek tehát olyan gyorsak, hogy a mérési eljárás határait feszegetik. Az első futtatás relatív lassúsága valószínűleg annak köszönhető, hogy a pipeline először találkozik a textúrával, a memóriaolvasások ekkor még optimalizálatlanok.

A három render pass futási ideje legrosszabb esetben sem érte el a 30 ms értéket. Ez alapján kijelenthető, hogy az OpenGL implementáció legalább 100-szoros gyorsulást ér el az OpenCV-hez – szekvenciális végrehajtáshoz – képest.

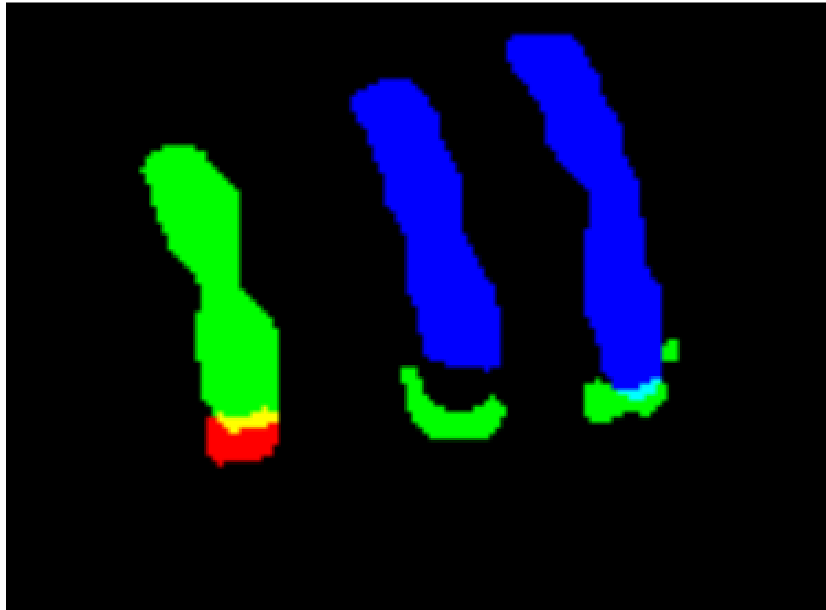
3.3.3.2.4. Az eredmény kiolvasása. A fent ismertetett program egy-egy futtatása után a framebufferből ki kell olvasni az elkészült bináris képeket, amit az OpenGL ES `glReadPixels()` függvényével végzek.

Ez a metódus egy Java standard `ByteBuffer`-be tölti a kimenet értékeit. Egy bemenő paraméterében megadható, hogy milyen formátumban szeretnénk megkapni az adatokat, jelen esetben a négy csatornás, csatornánként 8 bites konfiguráció megfelelő (még pazarló is, hiszen bináris képhez elég lenne egy egy bites felbontás, de az nem támogatott). Az OpenGL program kimenetei, mint minden belső változója, $0 - 1$ intervallumra normalizált lebegőpontos szám. A `glReadPixels()` automatikusan elvégzi a kívánt formátumra való alakítást még a GPU-n. Az végeredmény a tíz keresett színhez egy-egy bináris kép.

A 3.5 ábrán látható egy barna-piros-piros ellenállás küszöbözése után közvetlenül a GPU-ból kiolvasott kép. Az R csatorna a fekete, a G a barna, a B pedig

a piros színt jelöli. Látható, hogy a barna sávok szélein hibásan kis piros régiók is keletkeztek, illetve helyenként olyan sötét a szín, hogy a fekete küszöbözés is pozitív eredményt ad. A valós színsávok azonban dominálnak a hibák fölött.

3.5. ábra: Egy küszöbözés eredmény



Az ellenállások értékének meghatározásához további feldolgozás szükséges. Ez már feladatspecifikus rész, így nem törekedtem maximális sebességre, a következő lépéseket mind az OpenCV által nyújtott szolgáltatások segítségével végeztem.

3.4. A dekódolás

Az ellenállás értékek meghatározásához nem elég tudnunk, hogy az tokozáson előfordul-e egy-egy színsáv. Ezek sorrendje és távolsága is szükséges a kód megfejtéséhez. Emiatt ez a tesztelési feladat egyúttal arról is információt ad, hogy mennyire jól képes az algoritmus lokalizálni a színes régiókat.

3.4.1. Morfológiai előkészítés

A kapott bináris képeken általában nem tökéletesen jelennek meg a színsávok, gyakran keletkeznek hézagok a sávokon belül. Itt ismét jó szolgálatot tehet a morfológiai zárás művelete a 3.2.2.3 fejezetben leírtakhoz hasonló módon. Itt kiemelkedően fontos, hogy ne roncsoljuk túlságosan a képet, csak relatíve kisebb

torzulást okozzunk, hiszen a további lépések számára fontos elhelyezkedés és méret információból minél többet meg kell őrizni.

A csatornánkénti bináris képek mérete (pixel száma) már rögzített, megegyezik a GPU program bemenetéül szolgáló képével, lásd a 3.3.1 fejezetet. Erre alapozva kézi módszerrel állítottam be a morfológiai kernel méretét, és a dilatációk/eróziók számát.

3.4.2. Kontúrok és nyomatékok meghatározása

A már megfelelően előkészített kétszintű képeken ezek után kontúrkeresést hajtok végre a már korábban ismertetett `cvFindContours()` utasítás segítségével.

Ezt meghívom mind a 10 képre, és megkapom az összes összefüggő régiót egy-egy `Mat` osztályú objektumban, amiket eltárolok egy listában. A kapott kontúrokon az OpenCV könyvtár számos további metódusával dolgozhatok. Egyik ilyen igen hasznos utasítás a `moments()`, ami visszaadja egy régió nyomatékait a harmadik rendig. (Kép nyomatékokról bővebben lásd a 2.1.4 számú részt). Ezt a függvényt minden kontúrra meghívom. A következőkben az így nyert momentumokat több célra is felhasználom.

3.4.3. Pontszerű hibák kiszűrése

Nem hajtok végre előzetes morfológiai zajszűrést, hogy minél kevesebb pozíció és méret információt veszítsek. Ugyanakkor fontos, hogy az esetleges apró méretű, a küszöbözés folyamán hibásan detektált pixelszigeteket ne vegyem figyelembe a dekódolásnál.

Ezt a következő módon igyekszem biztosítani. A 0-ad rendű, m_{00} nyomatékok adják az adott kontúr által kijelölt régió területét. Ezért a tíz képen talált összes kontúr közül megkeresem azt, amihez a legnagyobb, M értékű 0-ad rendű nyomaték tartozik. Ez után ismét végignézem a régiókat, és kitörlöm azokat a listából, amik területe kisebb $\alpha * M$-nél. Az α értéket rövid kézi hangolás után $0,2$-nek választottam.

E lépéseket követően már csak azokkal kontúrokkal dolgozom tovább, amik magas valószínűséggel tartoznak egy ténylegesen jelenlévő színsávhoz.

3.4.4. A tok orientációjának számítása

Ahhoz, hogy a kódot sikeresen megfejtsük, szükséges ismerni a színsávok egymáshoz képesti távolságát is. Kézenfekvő megoldásnak tűnik, hogy a meglévő kontúrok által határolt régiók tömegközéppontjainak távolságával becsüljük ezt az értéket.

Az eddigi műveletek eredményeként rendelkezésre álló adatokkal azonban ez az ötlet gyakorlatban nem vezet jó megoldásra. Ennek oka egyrészt az, hogy a régiók középpontja ritkán esik pontosan az ellenállás tokjának közepére. A megvilágítás irányától és erősségétől függően általában a detektált sávok középpontja el van csúszva az ellenállás főátlójától.

A másik ok az, hogy ritka esetekben az is előfordul, hogy egy sávot két, vagy több összefüggő régióként kapunk meg. Ez főként olyankor jelentkezik, ha a megvilágítás erős, és a tokozás valahol „becsillan”, igen magas intenzitás értékek jelentkeznek, amik kettévágják a detektált sávot. Ezeket a régiókat tehát először össze kellene vonni.

Általánosan elmondható, hogy a régiók középpontjait az általuk reprezentált valós színsáv középpontjával összekötő vektorok közel merőlegesek az ellenállás tokjának hossztengelyére. A fenti problémák megoldásához jó alapul szolgálna tehát, ha ismernénk az ellenállás tokjának orientációját. Ehhez rendelkezésre áll a 3.2.3 résznél előállt, a tok pixeleit tartalmazó kép. Ha erre is meghívjuk a `cvFindContours` függvényt, majd a `moments()` segítségével kinyerjük a nyomatékait, akkor a következő módon megkaphatjuk az orientációt.

Először számítjuk a következő μ'_{ij} értékeket a megfelelő centrális nyomatékokból:

$$\mu'_{11} = \frac{\mu_{11}}{\mu_{00}} \tag{3.1}$$

$$\mu'_{20} = \frac{\mu_{20}}{\mu_{00}} \tag{3.2}$$

$$\mu'_{02} = \frac{\mu_{02}}{\mu_{00}} \tag{3.3}$$

Innen pedig megkaphatjuk a θ_p orientációt:

$$\theta_p = \frac{1}{2} \arctan \left(\frac{2\mu'_{11}}{\mu'_{20} - \mu'_{02}} \right) \tag{3.4}$$

A θ_p szög viszont még nem teljes válasz. Az $\arctan$ függvény $-\pi/2$ és $\pi/2$ tartományban ad értéket, ez csak a fele a nekünk szükséges tartománynak. Szükség van még egy adatra, hogy egy, a $[-\pi, \pi]$ intervallumba eső megoldást kapjunk. A μ_{11} centrális nyomatékot használom fel. Ennek értéke ugyanis pozitív, ha az orientáció a függőlegeshez képest $[0, \pi/2)$ tartománybeli értékkel tér el (pozitív matematikai irányba, a függőlegeshez képest „balra”), $(0, -\pi/2]$ tartománybeli kitérés esetén pedig negatív.

Így μ_{11} előjelét vizsgálva a következő a számítási módszer a végső θ orientációra:

- $\mu_{11} > 0$:

- $\theta_p > 0: \quad \theta = -\theta_p$
- $\theta_p < 0: \quad \theta = \frac{\pi}{2} + \theta_p$

- $\mu_{11} < 0:$
  - $\theta_p > 0: \quad \theta = \frac{\pi}{2} - \theta_p$
  - $\theta_p < 0: \quad \theta = -\theta_p$

Mivel a régiók középpontjainak csak egymáshoz képesti távolsága fontos számunkra, ezért nem szükséges a pontos abszolút helyzetüket ismernünk. Azaz elég egy, a tokot tartalmazó kép koordináta rendszerének origóján áthaladó, a testátlóval párhuzamos egyenesre vetíteni a a középpontokat merőleges projekcióval. Ennek mátrixa az egyenes irányvektorának ismeretében megkapható. A $\mathbf{b}$ irányvektor a következőképp is megkapható a θ orientáció szögből:

$$\mathbf{b} = [1 \quad \tan\theta] \tag{3.5}$$

Innen a leképezés mátrixa:

$$\mathbf{P} = \frac{1}{\mathbf{b^T b}} \mathbf{b b^T} \tag{3.6}$$

A vetítést $\mathbf{P}$ mátrixszal balról szorzással végezhetjük el egy adott pontra.

Az így transzformált középpontok távolsága már jó közelítése a színgyűrűk távolságának.

3.4.5. Régió összevonás

Említettem, hogy előfordul bizonyos képeknél, hogy egy színsávot két, vagy több régióként detektálja a küszöböző algoritmus (és az azt követő morfológiai zárás). Ezzel rokon jellegű probléma, hogy néha egy színsávra nem csak a hozzá tartozó, hanem az adott reprezentációban (RGB vagy L*a*b*) az adott színhez hasonló, másik színhez tartozó bináris képen is megjelenik egy olyan méretű hibás régió, ami átmegy a 3.4.3 részben leírt szűrésen. Ez igen ritkán fordul elő, ha viszont megtörténik, teljesen elrontja a színkód megfejtését.

E két problémát a következő algoritmussal próbálom kiküszöbölni. Első lépésben az összes meglévő régiót a középpontjuk x koordinátája szerint sorba rendezem. A rendezett listán végighaladok, és kiszámítom a szomszédos régiók középpontjai közti távolságokat. Ezek közül megkeresem a legnagyobb D értéket. Ez után az egyes színekhez tartozó listákat külön-külön az összesített listához hasonló módon sorba rendezem, majd elkezdek végighaladni rajta. Vizsgálom a listákban a szomszédos pontok távolságát. Amennyiben olyan d távolságot találok, amire igaz, hogy $d < \alpha * D$, akkor a két régiót összevonom: az új középpont a régi centroidok mértani közepe lesz, és a területek összegződnek.

A megmaradt kontúrokat ismét összesítem egy rendezett listába, és távolságokat számolok. Ha egy szomszédnak kisebb a távolsága, mint $\alpha * D$, akkor ez most már csak úgy következhet be, ha különböző színűek. Ekkor úgy döntöm el, hogy melyik szín a helyes választás, hogy a kontúrokhoz tartozó területeket összehasonlítom, és a nagyobbat választom.

A fenti lépések befejeztével azt feltételezem, hogy olyan adatokat kapok, amik már szorosan korrelálnak a valós sáv elrendezéssel.

3.4.6. Az ellenállás érték meghatározása

Úgy, hogy rendelkezésemre állnak a tokon levő színgyűrűk, és relatív távolságuk, már egyszerű visszafejteni a kódot. Ugyanakkor megjegyzem, hogy a kidolgozott eljárás alapvető hiányossága, hogy a forgalomban levő ellenállások túlnyomó részén előforduló arany és ezüst színsávot nem képes detektálni. Ennek oka az, hogy ezek kromaticitás szempontjából nem homogén színként jelennek meg a képen, és a Mahalanobis-távolságban használt ellipszoid közelítés nem megfelelő.

Emiatt a tesztelésnél csak 5%-os toleranciájú ellenállásokat használtam, ezeken az utolsó sáv arany. A kódolás megfejtésénél tehát ezt előzetes ismeretnek tekintem. Azonban szükséges ismernem, hogy milyen irányból kell elkezdeni a színsávok leolvasását. Az ilyen, 5%-os ellenállásokra általánosan jellemző, hogy az utolsó homogén színű sáv és előző szomszédja közötti távolság nagyobb, mint a többi egyszínű sáv közötti távolságok. Ezt az ismeretet kihasználom az olvasási irány meghatározásához. Megnézem, hogy két szélső azonosított sáv közül melyiknek nagyobb a szomszédjához képesti távolsága – ez lesz az „utolsó” színgyűrű.

Ha már ezt is megállapítottam, már csak össze kell állítani az ellenállás értéket a 2.7 fejezetben ismertetett szabályok alapján.

Ezzel a felismerési algoritmus végéhez érkeztünk, azonban egy fontos kérdésre még nem tértem ki az implementáció ismertetése során. Ez a megvilágítás változásából származó hiba kezelése.

3.5. A kép előzetes kondicionálása

A 2.2.2 fejezetben, ahol az színszegmentálási algoritmus kiválasztásával foglalkozom, azt a megkötést teszem, hogy a megvilágítási környezet az összes készített képen állandó.

Ez azonban nyilván nem teljesül valós körülmények között. A fény intenzitása és színárnyalata mellett a megvilágítás iránya is változhat. Ha egy laboratóriumban elhúzzuk a függönyöket, és felkapcsoljuk a fénycsöves világítást, diffúz jellegű, sárgás háttérvilágítás helyett koncentráltabb, felülről érkező, más színű

fényt kapunk. Ez, ha a kamera szenzorunkból érkező képet semmilyen előfeldolgozási lépésnek nem vetjük alá, drasztikusan eltérő pixel intenzitási értékeket eredményez a képek között.

Mivel a színszegmentálás egy előre összeállított szín leíró adatbázis segítségével történik, ez értelemszerűen komoly gondot jelent. Ha az adatbázist úgy építem fel, hogy csak egy adott megvilágítási környezetben készült képeket használok, akkor a háttérfény változásával nagyban csökken a felismert pixelek száma. Ha pedig sokfajta megvilágítással készült képet használok fel az adatok összeállításához, akkor túlságosan pontatlan lesz az egy színhez tartozó leírás, sok lesz a tévesen valamilyen színsávnak felismert, egyébként háttérhez, vagy másik színhez tartozó képpont.

Az általam választott színszegmentálási módszer jellegéből fakadóan az előbbiek alapján tehát keresnem kell egy olyan eljárást, ami képes valamilyen szinten csökkenteni az egyes képek közötti megvilágításból származó különbségeket. A megoldásomat a fotográfiában rég óta bevált fehéregyensúly állításra alapoztam.

3.5.1. Fehéregyensúly

A világ objektumainak valódi színe alatt az általános felfogás – és a fényképészet – szerint azt a színt értjük, amit teljesen fehér fénnyel való megvilágítás esetén kapunk. Ettől általában eltérő színeket kapunk, mert például a megvilágítás nem tökéletesen fehér, az emberi agy azonban az esetek többségében képes felismerni a dolgok eredeti színét a háttérvilágítástól függetlenül. Ha a valóságot azonban egy kamerán keresztül mintavételezzük, akkor elveszítünk olyan információkat, amik az agyat segítenék a valódi szín meghatározásában, és ugyanígy elrontjuk a jelen munkában tárgyalt, az emberi látáshoz képest lényegesen szerényebb képességű algoritmus működését.

A képrögzítés ezen mellékhatásának kiküszöbölésére gyakran alkalmaznak fehéregyensúly (vagy színegyensúly) korrekciót. Fontos először leszögezni, hogy fehér alatt az olyan színeket értjük, amikben a látható hullámhosszak közül mindegyiken egyforma az intenzitás, azaz a frekvenciaspektrumuk a 700nm és 390nm hullámhosszakhoz tartozó frekvenciák között konstans. Ezt a kritériumot az RGB színtérbe szemléletesen át lehet fogalmazni: itt az a fehér (semleges) szín, aminek mindhárom csatornája azonos intenzitású.

Több eljárás létezik, ami automatikusan képes egy kamerával rögzített képet átalakítani úgy, hogy az emberi szem számára közelebb kerüljenek a kép színei a valódiakhoz. Ilyen módszer a Johannes von Kries ötlete alapján Herbert E. Ives által kidolgozott von Kries-Ives átalakítás [4]. Ez az emberi szem színérzékelő sejtjeinek frekvencia válasza alapján skálázza át a képpontokat.

Mivel a jelen alkalmazásban egy gépi látási algoritmus számára kell konstans környezetet biztosítani, egyszerűbb eljárást is használhatunk.

A fehéregyensúly javítás tehát elvégezhető, ha van egy specifikus körülmények között készült (megvilágítás, kamera szenzor) képből egy színértékünk, amiről biztosan tudjuk, hogy egy, a való világban fehér színű objektumhoz tartozik. Ha RGB formátumban van a képünk, akkor az ismert, fehérre jellemző értékekből lehet számolni arányszámokat, amik jellemzik hogy mennyire térnek el a semleges fehér színtől.

Ezt a fehér színhez tartozó színadatot tehát valamiképp meg kell kapnunk. Én a fényképészetben elterjedt megoldást választottam: mutatni kell a kamerának valami fehér tárgyat. A pontos módszert a következőkben mutatom be.

3.5.1.1. Súlytényezők meghatározása

A készülék kameráját úgy kell irányítani, hogy az előnézetben egy túlnyomó részt fehér színt tartalmazó képet lássunk. Ezután az előnézetre egy hosszú tapintás (long tap) gesztussal lehet tudatni a programmal, hogy fehéregyensúly számításhoz vegyen mintát. A program ez után készít egy képet, és azon OpenCV függvényekkel dolgozik.

Az így készített képből RGB hisztogramot építek. Ez után a legtöbb elemet tartalmazó tárolókhoz (binekhez) tartozó R, G és B értékeket választom ki, mint fehéret, ezeket R_w, G_w és B_w-vel jelölöm. Ez az eljárás gyakorlatilag a képen levő színcsatornák móduszát választja ki. Ezzel már végezhető a fehéregyensúly számítás.

Azonban az, hogy kiegyenlítjük az eredetileg fehér színek RGB értékeit, még nem elegendő a mostani felhasználásban, hiszen az összehasonlítás pixel intenzitás értékek alapján történik. A fényerőre jellemző tulajdonságot is normalizálni kell. Ezért a következőképp számítom a színcsatornákat súlyozó tényezőket:

$$w_R = \frac{I_w}{R_w}, \qquad w_G = \frac{I_w}{G_w}, \qquad w_B = \frac{I_w}{B_w} \tag{3.7}$$

Az I_w értéknek egy magas fényerejű konstanst választottam. Ezzel az adott környezetben levő fehér objektum intenzitás értéke alapján az egész kép világosságát megkísérlem egy előre meghatározott értékre hozni.

E megkapott súlytényezőkkel szorzom egyrészt a referencia képek minden pixelét, amikből utána kigyűjtöm a színekre jellemző pixelhalmazokat. Másrészt pedig alkalmazásuk minden ellenállás felismerési algoritmus futtatásának első lépése kell legyen. Kiszámolásuk a fent leírt módszerrel néhány száz ms alatt lefut, és utána a felhasználó egy felvillanó szöveges üzenetben kap értesítést az számítás sikerességéről. Úgy éreztem azonban, hogy hasznos, ha a felhasználó látja a számítás hatását az elkészülendő képekre. Amellett döntöttem tehát, hogy a kamera előnézet folyam képein is elvégzem a fehéregyensúly korrekciót. Erre könnyen implementálható lehetőséget kínál az OpenCV Androidos keretrendszere.

3.5.1.2. OpenCV előnézet használata

Az OpenCV rendelkezik saját osztállyal, amik az Android készülék kamera hardveréből érkező adatfolyamot képesek megjeleníteni a felhasználó felé, illetve a megjelenítendő képkockákat a programozó rendelkezésére bocsátja a OpenCV standart `Mat` formátumban. Ez a `JavaCamera` osztály, ebből való származtatással próbálkoztam először.

Az előnézet felállítása meglehetősen egyszerű volt, így könnyen nekiláthattam a fehéregyensúly OpenCV-s implementációjának. Az egyik módszer, amit fontolóra vettem, az volt, hogy a megkapott kép minden képkockáját egyesével összeszorzom a súlytényezőkkel egy Java ciklusban. Végül azonban az OpenCV `filter2D()` utasítása mellett döntöttem. Ez vár egy képet, és egy kernelt, amivel a képen végighalad és korreláció műveletet hajt végre. Amennyiben a kernel egy pixel méretű, és a súlytényezőket tartalmazza, a szűrés megegyezik a pixelenkénti szorzással.

A módszert kipróbálva azt tapasztaltam, hogy a folyamatos vizuális visszacsatolás valóban jobb felhasználói élményt nyújt. Az OpenCV-s megvalósítás azonban igen lassú. A fehéregyensúly korrekció nélkül is jelentősen alacsonyabb a képfrissítési sebessége, mint a natív Androidos `SurfaceView`-en alapuló megoldásnak. Azzal együtt pedig 5 FPS-re (frame per second, másodpercenkénti képkockák száma) lassul. Ez pedig igen kényelmetlenné és nehézkessé teszi a jó minőségű, megfelelően fókuszált képek készítését a felhasználó számára. Ezért implementáltam egy másik megoldást is.

3.5.1.3. OpenGL előnézet használata

Az Android a 3.0-s verzió óta támogatja az OpenGL ES 2.0-t. Ebben a verzióban egyúttal egy olyan új kamera előnézet megoldást is bevezettek, ami az OpenGL-re és GPU-ra épít. A kamerából származó előnézet-képkockákat elkészültük után egyből egy speciális textúra objektumba tölthetjük fel, és utána rajtuk tetszőleges árnyaló programot futtathatunk. Ez a jelen alkalmazásban igen jó eredményekkel kecsegtet, hiszen a fehéregyensúly számítás szintén képpont alapú módszer, és egyetlen vektorszorzást igényel, ami pedig egy órajel alatt fut a grafikus processzorokon.

Az OpenGL alapú előnézet felállítása némileg több időt vett igénybe, mert itt is a már korábban (3.3.3.1 fejezet) említett `GLSurfaceView` osztályból kellett leszármaztatni, és ahhoz `GLRenderer`-t társítani, a megfelelő metódusokat felüldefiniálni. Továbbá a szükséges vertex és fragmens árnyalókat is meg kellett írni.

A vertex árnyalónak – a Mahalanobis programhoz hasonlóan – csak az a dolga, hogy átadja az interpolált textúra koordinátát és pozíciót a fragmensnek. Itt is ter-

mészetesen be kell adni egy háromszög stripet, ami kitölti a látóteret (l. a 3.3.3.2.1 fejezetet).

A fragment shaderben mintavételezem a textúrát, és a kapott RGB értékeket összeszorzom a súlytényezőkkel. A súlyokat uniform változóban adom át a programnak. A program indulásakor $\{1; 1; 1\}$ értékűek, ha a felhasználó ki-, vagy újraszámíttatja őket, akkor ezt a változót frissítem.

Az OpenGL megoldás igen gyors, képfrissítési sebessége megegyezik a `GLSurfaceView`-ével. Ebből arra következtetek, hogy a kamera hardver frissítési sebessége a korlát, nem a GPU program futási ideje. Ezzel együtt teljesen sima, képkészítésre kiválóan alkalmas előnézetet kapunk.

A fehéregyensúly korrekció az előzetes tesztelés során jelentősen javította a küszöbözés eredményét, a detektált sávok sokkal jobban fedték a valóságos színgyűrűket. Illetve több esetben is használata nélkül sikertelen volt a kód megfejtése, a súlyozó tényezők számíttatása után pedig már előállt a jó eredmény.

4. fejezet

Tesztelés

Az implementáció tesztelésére összegyűjtöttem az E24-es szériájú (5%-os toleranciájú) furatszerelt ellenállások közül az összeset az $[1k\Omega, 10k\Omega]$ érték-intervallumban. Ez 23 darab, köztük mind a tíz lehetséges szín előfordul.

A színeket leíró kovariancia mátrixokat színenként két-két képről állítottam össze. Mindkét képen ugyanaz az ellenállás szerepelt, az egyik ambiens jellegű, melegebb megvilágításban, másikon pedig erősebb, kékesebb neonfényben. A képek elkészítéséhez magát az alkalmazást használtam: kiegészítettem úgy, hogy képes legyen a fotókat a készülék belső flash memóriájára menteni. Mentés előtt minden esetben lefuttattam a fehéregyensúly tényezők számítását. A program ugyanis úgy végezi a tárolást, hogy előtte az RGB képen végrehajtja a színkorrekciót az aktuális súlyokkal, utána pedig CIE L*a*b* színtérbe transzformálja azt a 2.6 egyenletben leírt átalakítással együtt.

A két képről Matlab környezetben mintavételeztem ugyanazt a színsávot (RGB képként jelenítettem meg őket, a 2.6 átalakítás miatt így is felismerhetők az ellenállások és a sávok), a kapott pixelekből pedig elkészítettem a színt leíró Σ^{-1} inverz kovariancia mátrixot, és a μ átlagot. Mivel az OpenGL-ben a színintenzitások a $[0, 1]$ intervallumban mozognak, ügyeltem, hogy a 8 bit színmélységű képekből kinyert pixeleket a számítások előtt 255-tel való osztással normáljam.

A küszöbértékeket a Matlab-ban hangoltam színenként a 2.2.2.3 számú részben vázolt eljárás segítségével.

Az Android alkalmazás a színleíró adatokat (átlag, kovariancia inverz, küszöb) szöveges fájlokból olvassa be, így a megkapott színjellemzőket a Matlab-ból a megfelelő belső formátumban `txt` fájlokba exportáltam (egy színhez egy file) és felmásoltam a készülékre.

Ez után összesen 46 darab teszt-futtatást végeztem az ellenállásokon, mindegyiken kettőt a kétféle megvilágításban. 39 alkalommal helyes ellenállás-értéket adott a program, ez közel 85%-os pontosságot jelent.

Minden esetben megjelenítettem a küszöbözés után nyert bináris képeket és

a maszkolt színes képet is az algoritmus számszerű végeredménye mellett. A sikertelen esetek közül hat alkalommal az ellenállás testének felismerését végző rész már rossz eredményt adott, a maszkolt képen a tokozás csak egy kis darabja, vagy egy vezeték-rész volt látható. Egy alkalommal pedig a narancssárga színsáv egy nagy régiója pirosként is detektálva lett, és ez esetben ezt az utófeldolgozás nem tudta kiszűrni, így rossz ellenállás érték született.

5. fejezet

Eredmények, továbblépési lehetőségek

A tesztek alapján tehát a színszegmentálási rész megfelelőnek tűnik a színsávok elkülönítésére, azonban a hozzá társuló előkészítési fázis és utólagos logikai dekódolás nem elég robusztus.

Ugyanakkor egy tetszőleges magas szintű képértelmezési feladatnál ezek az elemek esetről esetre szükségszerűen mások lesznek. A Mahalanobis-távolság alapú küszöbözés bizonyítottan jó alapszintű megoldás, amire a további feldolgozási lépések biztonsággal építhetnek.

A megbízhatóságon kívül célom volt még a valós idejűség időbeli megkötéseinek megfelelni. Ezen a területen is sikeresnek mondható a választott módszer, hiszen 10 különböző szín detektálására 30ms felső korlát adható.

Az ellenállás azonosítás területére koncentrálva jövőbeli cél a jelenlegi tokazonosítási, illetve utófeldolgozási algoritmus részek hatékonyságának javítása. Ehhez egy lehetséges haladási irányt kínál a jelenlegi implementáció továbbgondolása, új szabályok, megkötések beillesztése. Egy másik út lehet a teljesen különböző elvű módszerek kipróbálása, pl. a kulcspontokon alapuló mintaillesztés.

Az színszegmentálás szemszögéből nézve fejleszthető még a mostani, Android-OpenGL alapú megoldás hatékonysága. A 3.3.3.1 fejezetben leírt megfontolások alapján lehetne optimalizálni az Androidos OpenGL környezet felállításának módját, illetve a GLSL árnyaló kódot.

Android platformon maradva megvizsgálandó lehetőség a RenderScript API használata is (l. a 2.5.1 fejezetet), hiszen a közeljövőben ez egy új, rendszer szinten optimalizált alternatívát fog nyújtani a grafikus processzorok használatára. A mobilkészülékek területén pedig megcélozhatók más platformok is, mint a piaci részesedésben második helyen levő Apple készülékek.

Ezek mellett pedig jó továbblépési lehetőséget kínál az eddigi munka során megtalált módszerre való építkezés, azaz egy másik, mobil eszköz alapú magas

szintű feldolgozási feladat megvalósítása. Ilyen lehet egy mesterséges markeres robotikai alkalmazás, pl. Visual Servoing [11].

Irodalomjegyzék

[1] *Android Software Development Kit*. 2013. URL: http://developer.android.com/sdk/index.html.

[2] *Apache License 2.0*. 2013. URL: http://www.apache.org/licenses/LICENSE-2.0.

[3] G. J F Banon. "Characterization of linear and morphological operators". In: *Computer Graphics and Image Processing, 1999. Proceedings. XII Brazilian Symposium on*. 1999, pp. 245–. DOI: 10.1109/SIBGRA.1999.805731.

[4] Michael H. Brill. "The relation between the color of the illuminant and the color of the illuminated object". In: *Color Research & Application* 20.1 (1995), pp. 70–76. ISSN: 1520-6378. DOI: 10.1002/col.5080200112. URL: http://dx.doi.org/10.1002/col.5080200112.

[5] International Electrotechnical Commission. *Marking codes for resistors and capacitors. IEC 60062 ed5.0*. 2004.

[6] mobiThinking. *Global mobile statistics 2013*. URL: http://mobithinking.com/mobile-marketing-tools/latest-mobile-stats/a#subscribers.

[7] *Open Handset Alliance*. URL: http://www.openhandsetalliance.com/android_overview.html.

[8] S.L. Phung, A. Bouzerdoum, and Sr. Chai D. "Skin segmentation using color pixel classification: analysis and comparison". In: *Pattern Analysis and Machine Intelligence, IEEE Transactions on* 27.1 (2005), pp. 148–154. ISSN: 0162-8828. DOI: 10.1109/TPAMI.2005.17.

[9] John C. Russ. "Thresholding". In: *The Image Processing Handbook*. Chap. Segmentation and Thresholding. ISBN: 978-1-4398-4045-0.

[10] Jun Tang. "A color image segmentation algorithm based on region growing". In: *Computer Engineering and Technology (ICCET), 2010 2nd International Conference on*. Vol. 6. 2010, pages. DOI: 10.1109/ICCET.2010.5486012.

[11] Wikipédia. *Visual Servoing*. 2013. URL: http://en.wikipedia.org/wiki/Visual_Servoing#cite_note-Weiss83-5.