Szegedi Tudományegyetem

Informatikai Intézet

SZAKDOLGOZAT

Tömördi Péter

2017

Szegedi Tudományegyetem
Informatikai Intézet

Színvakság szimulátor mobil eszközön

Szakdolgozat

Készítette:
Tömördi Péter
gazdaságinformatika szakos hallgató

Témavezető:
Dr. Tanács Attila
egyetemi adjunktus

Szeged
2017

Feladatkiírás

A világhálón számtalan olyan oldalt találunk, amely a színtévesztéssel, annak különféle eseteivel foglalkozik. Színreprezentációk közötti transzformációk segítségével ezek nagy részének a szimulációja kidolgozott. Nem találtunk viszont olyan Android rendszerre készült alkalmazást, amely az ilyen átalakításokat a kamera élőképen, valós időben tudja végezni és stabilan működik. A hallgató feladata egy ilyen mobil alkalmazás készítése.

Az átalakítás két irányban is működjön. Egyrészt empátia szimulátorként lehetőséget adjon a normál látásúaknak, hogy átérezhessék, mit látnak a környező világból a különféle színtévesztők. Másrészt a színtévesztők számára megkülönböztethetetlen, vagy nehezen elkülöníthető színek megfelelő cseréjével segítse a látásukat.

A megoldás opcionálisan bővíthető egyéni ötletekkel.

Tartalmi összefoglaló

- ***A téma megnevezése:***

Színvakság szimulátor mobil eszközön

- ***A megadott feladat megfogalmazása:***

A feladat egy olyan Android alkalmazás elkészítése, amely az eszköz kamerájának élőképét felhasználva szimulálja, hogyan lát egy színtévesztő, illetve olyan módosításokat végez a képen, aminek eredményeként a színtévesztők is meg tudják különböztetni azokat a színeket, amiket alapból nem tudnának.

- ***A megoldási mód:***

A képmanipulációs algoritmusok által adott lépések implementálása úgy, hogy azok egyaránt használhatók legyenek mozgóképek és állóképek esetén is. Ishihara teszt generálása forrásképek alapján, futásidőben.

- ***Alkalmazott eszközök, módszerek:***

A fejlesztés Android Studio-val történt, Java és C nyelveken. A kép átalakítását végző algoritmusok Renderscript használatával lettek implementálva. A tesztelés több emulátoron, egy erősebb és egy gyengébb fizikai eszközön volt elvégezve.

- ***Elért eredmények:***

Az alkalmazást a Google Play-en publikáltam, így sok emberhez el tudott jutni. A tesztek alapján a program fő funkciói az elérhető Android alapú eszközök többségén hiba nélkül működtek.

- ***Kulcsszavak:***

színtévesztés, android, élőkép, ishihara, mobil képelemzés

Tartalom

Feladatkiírás ... 1
Tartalmi összefoglaló ... 2
Bevezetés ... 4
1. A színtévesztésről ... 6
1.1. Mit jelent a színtévesztés? ... 6
1.2. Hogyan alakulhat ki a színtévesztés, mi okozza? ... 7
1.3. A színtévesztés főbb típusai, jellemzői ... 8
1.4. Módszerek a színtévesztés diagnosztizálására ... 9
2. Képkezelés az Androidban ... 12
2.1. Kamera kezelése ... 12
2.2. Bitmap kezelés ... 13
3. Az alkalmazás tervezett felépítése, működése ... 16
3.1. Fejlesztési és tesztelési környezet ... 16
3.2. Az alkalmazás tervezett megjelenése ... 16
3.3. Az alkalmazás tervezett felépítése ... 19
4. Szűrők használata a programban ... 21
4.1. Daltonizáló szűrő ... 21
4.2. A szimuláció ... 23
4.3. Szűrők implementálása, alkalmazása ... 23
4.3.1. A szűrők Java-ban ... 23
4.3.2. A szűrők Renderscriptben ... 24
4.3.3. Az állóképes daltonizálás ... 25
4.3.4. A szűrők alkalmazása élőképen ... 30
5. Ishihara teszt megvalósítása ... 37
5.1. A teszt ... 37
5.2. A teszteredmény megtekintése ... 41
6. Firebase ... 43
6.1. Realtime Database ... 43
6.2. Crash Reporting ... 45
7. A program használata ... 46
7.1. Élőképes mód ... 46
7.2. Állóképes mód ... 48
7.3. Teszt ... 50
8. Teszteredmények ... 53
Irodalomjegyzék ... 57
Nyilatkozat ... 58

Bevezetés

Szakdolgozatomban olyan program elkészítése volt a cél, amely később hasznossá válhat, és segítséget nyújthat az embereknek. A képfeldolgozás mindig is érdekelt, így merült fel a színtévesztés, a szimuláció, és a daltonizálás, mint téma. Első lépésként tájékozódtam, hogy a témában milyen mennyiségű, és minőségű szoftver érhető el, ezeket milyen eszközök támogatják, mennyire könnyű őket használni, és természetesen hogyan vélekednek róluk az érintettek. Több remek nyílt forráskódú kezdeményezéssel találkoztam, amelyből ötletet is merítettem. Ezek után folytattam a keresést az Android Play áruházban. Arra a kérdésre kerestem a választ, hogy érdemes-e színtévesztés szimulálásával, daltonizálásával foglalkozó alkalmazást készíteni androidos eszközre, nincs-e túl sok már belőle. Színtévesztéssel, teszteléssel foglalkozó alkalmazás akad egész sok, de kamera élőképes szimulálással foglalkozó megoldásból kevés van, ráadásul ezek többsége nem túl stabil, illetve régóta nem fejlesztik őket. Olyat nem találtam, ami élőképen daltonizálást végzett volna.

A feladat Android alapú mobil eszközökön a színtévesztés szimulálása, illetve a színtér olyan módosítása, amelynek eredményeképp a színtévesztők különbséget tudnak tenni olyan színek között, amelyre alapesetben nem lennének képesek (daltonizálás). A színtévesztés 3 fő típusának szimulálására, és daltonizálására már léteznek függvények, én ezeket a már létező megoldásokat használtam fel a programomban. Az aktuálisan létező, elérhető, és felhasználható megoldásokat kipróbáltam, az eredményeket a lehető legtöbb mintaképpel vetettem össze. Az így összegyűjtött tapasztalatok alapján választottam, és a kiválasztott megoldást androidos környezetben felhasználtam.

Fő cél, hogy a fenti kétféle színmódosítás a mobil eszköz kamerája által nyújtott élőképen működjön, így az érdeklődők könnyen és gyorsan tudják a tárgyakat, vagy a környezetet vizsgálni. Ebben a módban lehetőség van a 3 fő típus közötti váltásra, illetve a szűrő be és kikapcsolására, így azonnal látható az egyes szűrők közötti különbség. Lehetőség lesz továbbá ebben a nézetben az éppen látott képnek az elmentése is.

A daltonizáló szűrőhöz készült egy külön állóképes modul is, ahol vannak példaképek is az egyes szűrőkhöz, illetve lehetőség van egyedi képek megnyitására, és ezen képekre is lehet alkalmazni a szűrőket. Képet meg lehet nyitni a telefon háttértárolójáról, illetve az Androidba épített kamera alkalmazással közvetlenül is lehet fényképet készíteni, és ezt megnyitni. A módosított képet meg is lehet osztani.

Az alkalmazás részét képezi egy Ishihara teszt is. 30 darab ábrát generál a program különböző színkombinációkkal. Az ábrák főleg alakzatokból állnak, és mindig majd a felajánlott eredeti alakzatok közül lehet választani, hogy éppen mit lát az illető. Az alakzatok sorrendje, az alkalmazott előre meghatározott színkombinációk, és maga a generálás futásidőben történik. A teszt végén a kitöltő kap egy százalékos eredményt, ami azt mutatja, hogy mennyi ábrát látott helyesen. Ha a felhasználó a kitöltés előtt hozzájárult, akkor statisztikai célból az eredménye névtelenül fel töltődik egy adatbázisba. Ha a kitöltőt az eredménye után érdekli, hogy mely ábrákra adott rossz választ, akkor lehetősége nyílik a hibáinak a megtekintésére. Ez olyan formában valósult meg, hogy a felhasználó megnézheti az ábrát, az általa adott választ, a helyes választ, és az ábra daltonizált változatát, ahol választhat a 3 féle szűrő közül. Így az is kiderülhet, hogy az adott személynek segít-e a daltonizálás, vagy sem, mert így lehetősége nyílik az ábrák, és a helyes válasz összevetésére.

1. A színtévesztésről

1.1. Mit jelent a színtévesztés?

Színtévesztésről akkor beszélünk, amikor az egyén nem képes olyan színeket egyértelműen megkülönböztetni egymástól, amiket ugyanolyan körülmények között más egészséges látású emberek el tudnak különíteni. Gyakran hibásan színvakságként emlegetik, pedig annak eltérő jelentése van. Aki színvak, az egyáltalán nem képes két szín közötti különbség érzékelésére, mondhatni, hogy szürkeárnyalatosan, vagy egy színben látja a világot. A színtévesztők látják a színeket, de vannak olyan színárnyalatok, amiket közel egyformának látnak.

Az elsők között Mr. John Dalton, angol fizikus és kémikus foglalkozott komolyabban a színtévesztés problémájával. Mivel ő maga is érintett volt, így megfigyeléseit kellő részletességgel le tudta írni [1]. Különböző fényviszonyok (napfény, gyertya fénye) között vizsgálta, hogy ő milyennek látja az adott tárgyak színeit, és más emberek miként látják azokat. Olyan általános dolgokat vizsgált, mint a fű, vagy a rózsaszínű rózsa. Mint kiderült, az ő szemszögéből a kék, a rózsaszín, a lila, és a karmazsinvörös mind kéknek tűnik. Ennél példát is hoz, a muskátlit főleg kéknek látta, ám ha ugyanazt a virágot gyertyafénynél nézte, akkor más színűnek érzékelte.

1.1 ábra: Dalton feltételezett látása

Az 1.1 ábra mutatja be, hogy feltételezhetően hogyan látott Dalton természetes fényben. A felső kép az eredeti kép, az alsó pedig az átalakított. A felső és alsó képek között főként a lila, rózsaszín, és piros árnyalatok térnek el jelentősen, ez derült ki Dalton leírásából is. Egyéb színek esetén, például a kék, és a zöld, csupán kisebb eltéréseket tapasztalt. Más esetekben, például a barna különböző árnyalatainál sokkal nagyobb eltéréseket érzékelt,

mint az egészséges látásúak. Bizonyos barnákat pirosnak érzett, más sötétebb barnákat pedig inkább feketének.

1.2. Hogyan alakulhat ki a színtévesztés, mi okozza?

Dalton azt feltételezte, hogy hibás színlátását az okozza, hogy a szemében lévő folyadék elszíneződött. Emiatt külön kérte is, hogy halála után vizsgálják meg a szemét, hogy feltételezése igaz-e vagy sem. Mint kiderült, nem ez áll a színtévesztés jelensége mögött.

A megfelelő színek érzékeléséért a retinában található csapok a felelősek [2-3]. Minden csap egy adott színkomponensre érzékeny. Ennek megfelelően vannak pirosra, zöldre, és kékre érzékeny csapok, vagyis a hosszú, a közepes, és a rövid hullámok feldolgozásáért felelősek. Ezek alapján nevezik őket L, M, és S csapoknak. Ha ezek nem megfelelően működnek, akkor hibás lesz a színérzetünk, hiszen nélkülük nem tudjuk érzékelni az adott színtartomány(oka)t.

A színtévesztés főként örökletes [4]. A gyerekek az X kromoszóma által örökölhetik, emiatt még a születésük előtt, a fejlődési szakaszban hibásan fognak felépülni a csapok. Mivel az X kromoszóma érintett, így főként a férfiaknál jelentkezik a probléma, a nők főleg csak hordozók szoktak lenni, az ő esetükben lényegesen ritkábban tapasztalható színtévesztés. Ilyenkor a probléma legtöbbször már a születéskor jelentkezik.

Egyéb betegségek szövődményeként is jelentkezhet színtévesztés, ilyen lehet például a cukorbetegség [5]. Részletesen foglalkoztak ezzel, vizsgálták, hogy a cukorbetegség során pontosan mi is válthatja ki a színtévesztést. Egy összetett eljárással vizsgálták egyészséges, és cukorbeteg emberek színlátását, és ebből vontak le következtetéseket.

Másik példa lehet a makuladegeneráció [6]. A makula a retina egy jól meghatározott területe, ennek a területnek a kóros átalakulását jelenti a megnevezés. Ekkor is jelentkezhet a színtévesztés, de emellett egyéb problémákat is tapasztalhat az, akinél ez kialakul. Itt már gond lehet az éleslátással, és összességében a fényérzékeléssel is.

Hasonlóan a halláskárosodáshoz, a színtévesztés sem gyógyítható. Nincs lehetőség a hibás csapok helyreállítására, javítására, azonban különböző módszerekkel a színérzet javítható. Készülnek különböző típusú színtévesztésekhez speciális szemüvegek, amelyek színszűrővel vannak ellátva, ezáltal könnyebbé válik a színtévesztőknek olyan színek megkülönböztetése is, amelyek eddig gondot okoztak nekik. Az informatikában is

léteznek már olyan algoritmusok, amelyek ezt a célt szolgálják. Többek között az én programomban is találhatóak ilyenek, ezzel segítve a színtévesztőket.

1.3. A színtévesztés főbb típusai, jellemzői

A különböző színtévesztéseket a csapok rendellenes működése alapján szokták csoportosítani [7].

Monokromáziáról beszélünk akkor, amikor egyik csaptípus sem működik, vagy esetleg egy működik, és a másik kettő típus működésképtelen. Ebben az esetben az érintett vagy szürkeárnyalatosan lát, vagy éppen a működő csaptípusának megfelelő szín árnyalatait látja, de a többit nem, vagyis az általa érzékelt színspektrum egyetlen szín árnyalataira korlátozódik (1.2 ábra). Őket szokták színvakként emlegetni.

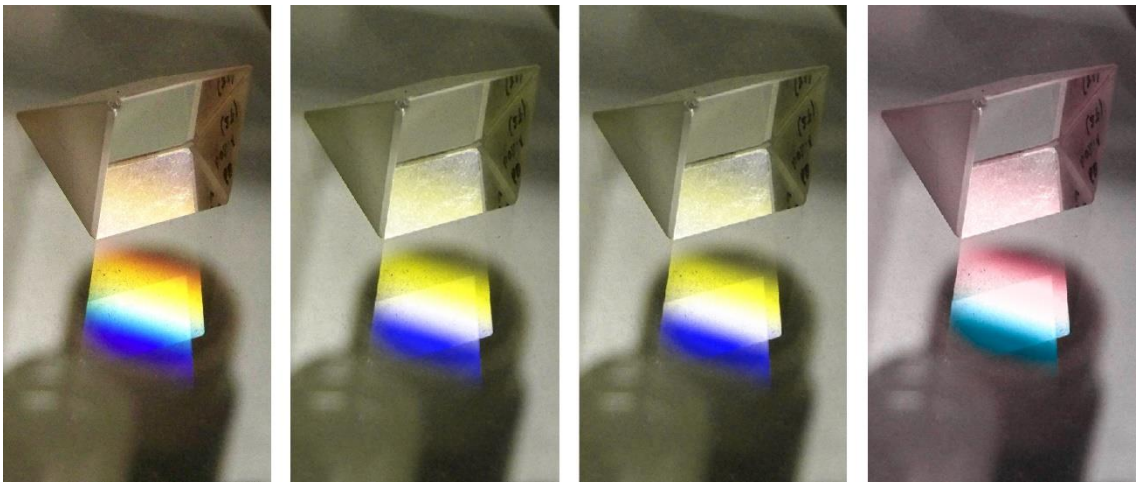
1.2 ábra: a monokromázia egy lehetséges változata

A dikromázia esetében 3 fő típusról beszélhetünk. A dikromázia lényege, hogy 2 csaptípus megfelelően működik, a 3. csaptípus állapotának megfelelően pedig két esetről beszélünk. Az egyik, amikor a 3. csaptípus egyáltalán nem működik, a másik, amikor a 3. csaptípus csak részlegesen működik. Attól függően, hogy melyik csaptípus működik hibásan, vagy egyáltalán nem működik, 3 kategóriát állapítottak meg. Az egyik a protanópia, ami az L-csapok hibás működésére utal, vagyis a piros színnek az érzékelése problémás, hiányos. Az érintettek nehezen, vagy egyáltalán nem képesek megkülönböztetni a kéket és a zöldet, illetve a pirosat és a zöldet. Az M-csapok hibás működését, működésképtelenségét jelenti a deuterapópia, ebben az esetben a zöld szín érzékelésével vannak gondok. Az érintettek ebben az esetben nehezen, vagy egyáltalán nem tudják megkülönböztetni a pirosat a zöldtől. Tritanópia esetén az S-csapok

problémásak, vagyis a kék szín érzékelése hiányos. Ebben az esetben a zöld és a kék megkülönböztetése okoz nehézségeket.

Dikromázia:

- A 3. csaptípus részlegesen működik:
  - protanormál
  - deuteranormál
  - tritanormál
- A 3. csaptípus egyáltalán nem működik:
  - protanópia
  - deuteranópia
  - tritanópia



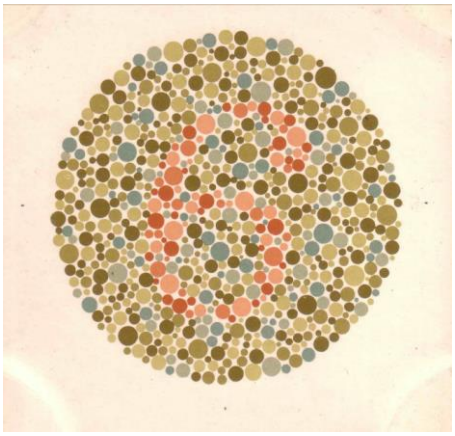
1.3 ábra: Lehetséges dikromáziák, eredeti, deuteranóp, protanóp, tritanóp

Az egészséges látásúak esetén beszélünk trikromáziáról. Azonban ennek is van egy rendellenes formája, ekkor a probléma kifejezetten összetett is lehet. Ilyenkor az egyik csaptípus nem a neki megfelelő színre érzékeny, hanem érzékenysége eltolódik valamelyik irányba. Ez sokféle hibás színérzetet okozhat attól függően, hogy mely csap érintett, milyen irányú, és milyen mértékű az eltolódás.

1.4. Módszerek a színtévesztés diagnosztizálására

Gyakori módszer az Ishihara teszt [9], amely alkotójáról, Dr. Shinobu Ishiharáról, a tokiói egyetem professzoráról kapta nevét. Ennek lényege, hogy általában 38 képet mutatnak a vizsgált személynek, akinek meg kell mondania, hogy mit is lát. Minden kép egy körben tartalmaz pöttyöket, amiknek elhelyezkedése, és mérete véletlenszerű, de megadott

tartományokon belül mozog. Ezen kell általában számot, de olykor más alakzatot felismerni. A háttérnek, és magának az alakzat színeinek az összetétele, és a színek eloszlása meghatározott. A teszt a protanóp és deuteranóp színtévesztés diagnosztizálására alkalmas, tritanóp esetben mást kell alkalmazni. A teszt csak akkor adhat pontos eredményt, ha megadott szabályok szerint végzik. Ezek alapján nappal kell elvégezni a tesztet, vagy olyan fényeket kell biztosítani, ami nappali fényviszonyokat teremt. A vizsgált személynek 75 cm-ről kell néznie a képeket, és adott minták esetén fix maximális gondolkodási idő biztosított. Mivel nem igényel ezeken kívül egyéb speciális körülményeket, ezért találkozhatunk gyakran ezzel a módszerrel. Például a jogosítvány megszerzéséhez szükséges alkalmassági teszten is többnyire ezzel a módszerrel vizsgálják az emberek színlátását.



1.4 ábra: egy Ishihara kép

Egy precízebb tesztelési módszer a Farnsworth-Munsell teszt [8]. Itt azt vizsgálják, hogy mennyi színárnyalatot tudunk megkülönböztetni. Maga a teszt abból áll, hogy a tesztelendő személy kap több sornyi színt. A sorokban négyzet formájában vannak színárnyalatok, a sor első színe, és az utolsó színe rögzített. Ezek között összekeverve vannak a kettő közötti árnyalatok, és ezeket kell sorba rendezni. A tesztnek többféle változata is létezik, valahol 22 db színárnyalat van egy sorban, máshol csak 10. Minél több a színárnyalat, annál pontosabban végezhető el a teszt, de annál nehezebb is megoldani a tesztet. A tesztet ma már megfelelően kalibrált kijelzővel ellátott számítógépen szokták elvégezni, emellett itt is biztosítani kell a megfelelő fényviszonyokat.

1.5 ábra: Farnsworth-Munsell teszt példa, összekevert árnyalatokkal

Ezek mellett természetesen számos egyéb módszer is létezik, amiket attól függően alkalmaznak, hogy mennyire alapos diagnózisra van szükség, milyen technikai felszerelés érhető el, és mennyi időt szeretnének magára a tesztre rászánni. Sajnos a színtévesztés, illetve a színlátással kapcsolatos problémák rendkívül sokfélék lehetnek, ezért a ritkább esetek diagnosztizálása nehéz lehet, ilyenkor van szükség összetettebb, precízebb tesztekre is.

2. Képkezelés az Androidban

2.1. Kamera kezelése

Androidban régen a `Camera` osztály reprezentálta a kamerát, amely még az újabb API-kban is fellelhető, de elavult. API 21-től jelent meg az új, `Camera2` osztály, mely ugyanezt a célt szolgálja. Új alkalmazás fejlesztésekor érdemes már ezt használni, viszont az alkalmazás ekkor a régi eszközökön nem fog működni. Én a 4-es Android verziókat is szerettem volna támogatni, amik API 15-től kezdődnek, így én a sima `Camera` osztályt használtam.

A kamerából kétféleképpen lehet képi adatot szerezni. Az egyszerűbb megoldás az, amikor az alkalmazásomból meghívok egy másik kamera kezelő alkalmazást, ezzel a felhasználó elvégezi a fotózást, majd a képi információ eljut az alkalmazásomhoz, ahol ezt fel tudom dolgozni. Ez azért könnyebb, mert itt nem kell foglalkozni magának a kamerának a kezelésével, elég csak az elkészült képre koncentrálni. Annyira azért nem egyszerű itt se a helyzet, hiszen a képi adatok továbbítása API szintenként, és készülék típusonként is eltérhet. Ez abból ered, hogy az egyes eszközökön más beépített kamerakezelő szoftver van, és maga a rendszer is másképp viselkedhet. A külső szoftver hívás így történik:

```
takePictureIntent =
new Intent(MediaStore.ACTION_IMAGE_CAPTURE);
startActivityForResult(takePictureIntent, 1);
```

Hivatalosan [10] a képrögzítés ebben az esetben úgy történik, hogy a kamera kezelő alkalmazás, miután befejezte a kép rögzítését, meghívja az alkalmazásom `onActivityResult` függvényét, amelyben átad egy `Bundle` objektumot, amely többek között a kép egy kis felbontású változatát tartalmazza. Ha a teljes méretű képre is szükség van, akkor a külső alkalmazás hívása során meg kell neki azt mondani, hogy hova rögzítse a képet, egy konkrét, létező fájlra van szüksége, elérési úttal. Ezt előbb nekünk létre kell hoznunk, amihez jogosultság kell, aminek meglétét ellenőrizni kell. Erre a folyamatra azért van szükség, mert a `Bundle` objektumok mérete rendkívül korlátozott, nem férne el benne egy teljes felbontású kép.

A bonyolultabb eljárás az, amikor alkalmazáson belül kerül lekezelésre a kamera. Ekkor sokkal több a tennivaló, de sokkal több a lehetőség is. Elsőként külön engedélyre lesz szükség, hogy a kamerát az alkalmazás használhassa, ezt mindig ellenőrizni kell,

engedély hiányában kérni kell azt. Az Androidban a kamera egy erőforrás, amit le kell foglalni, és használat után el kell engedni, fel kell szabadítani. A kamera elkérése:

```
Camera c = null;
try {
    c = Camera.open();
}
catch (Exception e){
    e.printStackTrace();
}
```

Egyáltalán nem biztos az, hogy amikor az alkalmazásnak szüksége van a kamerára, akkor meg is kapja azt, hiszen előfordulhat, hogy épp használatban van, erre fel kell készülni. Miután sikerült megszerezni a kamerát, akkor a `Camera` osztály függvényei segítségével paraméterezhető, irányítható a kamera. Számos információ kérdezhető le a kameráról, például támogatott felbontások, előnézeti képen az FPS szám, támogatja-e az autófókuszt, stb. Ezeknek a többségét be is lehet állítani neki, tehát meghatározható a felbontás, legyen-e autófókusz, stb.

Az egyik legfontosabb teendő, hogy a kamera által szolgáltatott előnézeti élőképet meg kell jeleníteni. Ehhez a `SurfaceView` osztályból kell származtatni egy saját osztályt. Ezen jelenhet meg maga az élőkép, ezen kívül persze szükség lesz egy másik saját nézetre is, amelyen elhelyezzük a szükséges gombokat, feliratokat, és egyebeket. A megjelenő élőkép ezek után már tetszés szerint manipulálható.

2.2. Bitmap kezelés

Android alatt a képkezelés központi osztálya a `Bitmap`. Ha bármilyen képet szeretnénk megjeleníteni, legyen az erőforrás, fájl, vagy kamerából érkező kép, az adatok kezeléséhez egy `Bitmap` objektumra lesz szükség.

Fájból való olvasás és konvertálás csak jpg, png, gif, és webp képek esetén lehetséges. Fájlból képbeolvasásra példa:

```
File imagePath =
new File(getApplicationContext().getCacheDir(), "images");
File newFile = new File(imagePath, "test.png");
Bitmap newBitmap =
BitmapFactory.decodeFile(newFile.getPath());
```

Amikor létrehozunk egy `Bitmap` objektumot, akkor lehetőségünk van többek között meghatározni, hogy milyen színteret használjon. Bitmap létrehozásra példa:

```
Bitmap newBitmap = Bitmap.createBitmap(width, height,
Bitmap.Config.ARGB_8888);
```

Leggyakrabban az `ARGB_8888` a használatos, ekkor minden pixel 32 biten kerül eltárolásra, ahol az első 3 bájt a piros, zöld, és kék színek, a 4. bájt pedig az alfa csatorna. Másik lehetőség az `RGB_565`, ekkor 16 bitet foglal el egy pixel, itt a piros és a kék 5 bites, a zöld pedig 6 bites. Ha színinformációkra nincs szükség, akkor az `ALPHA_8` használatos, ahol színinformáció nem kerül tárolásra. Ezekből adódik, hogy a kép a memóriában tömörítetlenül van tárolva, ezért a szükséges memóriamennyiséget a felbontás, és a használt színtér fogja megszabni, tehát teljesen mindegy, hogy jpg, vagy png volt-e a forrás. Nagyfelbontású képet csupán megjelenítési célból ezért felesleges is beolvasni, hiszen értelmetlenül fog nagy memóriát igényelni, és úgyis csak a kép felbontásához képest kicsi kijelzőn fogjuk megjeleníteni. Éppen ezért lehetőség is van csökkentett felbontású kép létrehozására, viszont itt nem árt tudni, milyen volt az eredeti kép felbontása. Ezért a képbeolvasásnál meglehet adni opcióként az `inJustDecodeBounds` logikai értéket igaznak, így nem olvassuk be rögtön a teljes képet, de a méretét már megtudhatjuk. További helymegtakarítással jár az, ha nem olvasunk be minden pixelt, hanem minden másodikat, vagy negyediket. Emellett még megadható egy denzitás érték is, ami adott számú pixel helyett 1 db átlagolt pixelt fog eredményezni.

Egy `Bitmap` minden pixelét el tudjuk érni, le tudjuk kérni. Módosítani is lehet pixelenként, de ahhoz módosíthatónak kell lennie. A fájlból beolvasott képek alapértelmezés szerint nem módosíthatók. X és Y koordinátákkal lehet hivatkozni egy pixelre, ahol a 0,0 érték a kép bal felső sarkát jelenti. A különböző szűrők működéséhez is erre van szükség, tudni kell az adott pixel értékét, és ez alapján kell felülírni egy adott másik értékkel, hogy a végén a megfelelő képet kapjuk.

Ha rajzolásra is szükség van, akkor ez a pixelenkénti módszer eléggé nehézkes lehet. Az Ishihara teszt készítéshez különböző méretű és színű köröket kell rajzolni. Ehhez már `Canvas`-t érdemes használni. Segítségével különböző primitíveket rajzolhatunk egy előre megadott `Bitmap`-re, amit aztán ki lehet tenni egy `ImageView`-ra, vagy el lehet menteni fájlként. `Canvas` segítségével lehet vonalat, kört, ellipszist, stb. rajzolni, a használt színt pedig `Paint` objektumokkal tudjuk beállítani. A rajzolási folyamat a futás egy jól meghatározott pontjában történik, amit az Android hív meg. Két lehetőség van, az egyik, hogy egy saját `View onDraw` metódusában történik a `Canvas`-re rajzolás, a

másik, amikor `SurfaceView` használatával történik rajzolás. A kettő közötti lényeges eltérés, hogy saját `View` esetén a számítások a UI szálon futnak, és ahhoz, hogy újra hívódhasson az `onDraw`, szükség van egy `invalidate()` hívásra. Az `onDraw` máskor is hívódhat, amikor az Android azt szükségesnek tartja, például orientációváltáskor. A `SurfaceView`-s megoldásnak ezzel szemben az előnye, hogy a számítások külön szálon futnak, és nincs szükség `invalidte()`-re, ha újra akarjuk rajzolni a képet.

3. Az alkalmazás tervezett felépítése, működése

3.1. Fejlesztési és tesztelési környezet

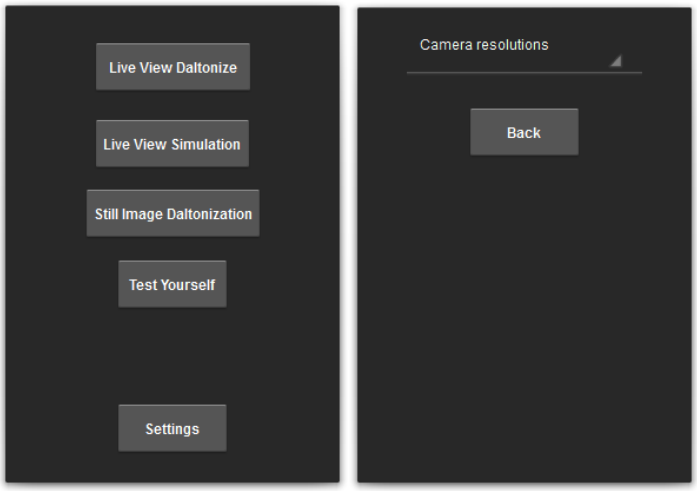
Fejlesztési eszközként több alternatíva is rendelkezésre áll. Régebben az Eclipse kifejezetten népszerű volt ezen a téren, de a Google ennek támogatását megszüntette, miután saját IDE-vel állt elő, ez pedig az Android Studio. IntelliJ alapokon nyugszik, így a fizetős IntelliJ IDEA alatt is lehet Androidra fejleszteni, viszont az Android Studio ezzel szemben ingyenes, és hivatalos is, ebből adódóan rengeteg hasznos dokumentáció segíti a kezdőket és a haladókat is. Ezek miatt én is emellett döntöttem. Az első project megnyitásakor az Android Studio az 1.4-es verziónál tartott, frissítések viszonylag gyakran érkeznek hozzá. A Studio szerves részét képezi az Android SDK. Ez mindent tartalmaz, ami szükséges a fejlesztéshez, fordításhoz. Ehhez tartozik egy SDK Manager is, amivel BuildTools-t, SupportLibrary-t, System Image-eket, Drivereket és egyebeket lehet letölteni és frissíteni. Az Android Studio-t főleg Windows alatt használtam.

A tesztelés első körben két fizikai eszközön történt. Az egyik készülék egy szerényebb hardverrel rendelkező Alcatel OneTouch, melyen 16-os API szintű Android futott. Hardver szempontjából ez a készülék képezte a szűk keresztmetszetet, vagyis ez a leggyengébb eszköz, amin az alkalmazás használhatóan tud még futni. A másik eszköz egy HTC One Silver, lényegesen erősebb hardverrel, és 21-es API szintű Androiddal. Ezek mellett, ha ritkábban is, de teszteltem az alkalmazást virtuális eszközökön is. Erre lehetőséget kínál az Android SDK is, egészen 10-es API szintig lehet letölteni System Image-eket, amelyekből aztán az AVD Managerrel virtuális eszközök készíthetők, és utána futtathatók. Ezen a módon lehetőségem volt megnézni, hogy újabb API szinteken hogyan viselkedik az alkalmazás. Emellett a Firebase is rendelkezik Test Lab nevű területtel, ahova apk-t lehet feltölteni, és limitált számú eszközön (fizikain és virtuálison is) lehet robo test-et, vagy espresso tesztet futtatni ingyenesen. Erről adott esetben veremlenyomat, képernyőkép, és videó is érkezik.

3.2. Az alkalmazás tervezett megjelenése

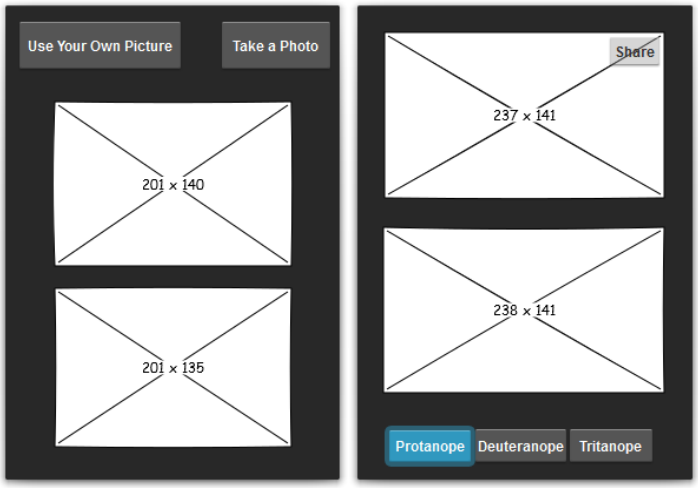
A program minden esetben rögzített orientációval megjelenik meg. Az élőképes részek fekvő tájolásúak, abból a megfontolásból, hogy többnyire a kamera alkalmazások is fekvő tájolásúak szoktak lenni. A program összes többi része álló tájolású. A 3.1 ábra első képe a fő menü, ez fogadja a felhasználót a program indításakor. Innen tud eljutni a menüpontokon keresztül az összes lehetőséghez. A második kép a beállítások, ide került

a kamera élőkép esetén alkalmazott felbontás kiválasztása. A kamera által támogatott felbontások közül lehet itt választani, választás után közvetlenül vissza lehet jutni a fő menübe.



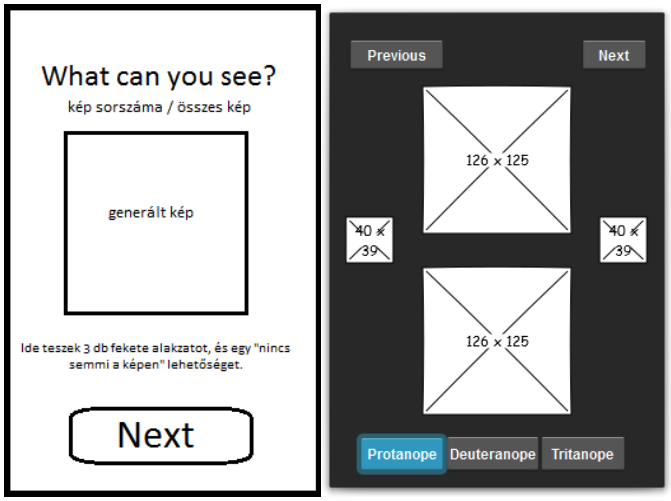
3.1 ábra: Fő menü és beállítások

A 3.2 ábra az állóképes Daltonizálást mutatja. Amikor a felhasználó rányom az ennek megfelelő gombra, akkor először egy példaoldalt fog kapni, ahol néhány színes példaképen keresztül lesz alkalmazva a daltonizáló szűrő, hogy látni lehessen, miről is van szó. Innen majd tovább is lehet menni, és saját képre is lehet alkalmazni a szűrőket. A bal oldali kép a példaoldalt mutatja, lent láthatók a példaképek, a jobb oldali pedig azt az oldalt mutatja, ahova a felhasználó kerül, miután képet választott. Lent három gomb van, amik a szűrőknek felelnek meg, innen tud a felhasználó választani, hogy melyiknek a hatására kíváncsi. A gomb kiválasztását követően rögtön látható lesz annak hatása. Az alsó kép az eredeti, a felső pedig a szűrt. Ezt a képet lehet megosztani a kép jobb felső sarkánál lévő gombbal. Ha a felhasználó ezt választja, akkor lehetősége lesz kiválasztani, hogy mivel szeretné a képet megosztani. Listát kap az ilyen jellegű megosztást támogató, jelenleg telepített alkalmazásairól. Ilyen lehet a GMail, vagy a Messenger.



3.2 ábra: Állóképes Daltonizálás

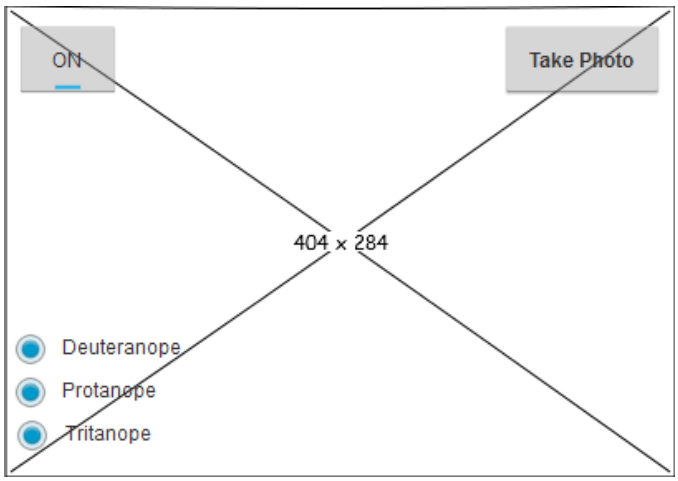
A 3.3 ábra mutatja az Ishihara teszt tervezett megjelenését. A teszt indítása előtt van egy rövid magyarázat, és egy jelölőnégyzetes mondat, ami az anonim eredményfeltöltésről szól. Ha a felhasználó bent hagyja a pipát, akkor hozzájárul az eredmény feltöltéséhez. Ezután fogja látni a bal oldali szerkezetet, középen lesz a generált kép, alatta 3 forráskép, ezeket kell összehasonlítani, és felismerni a generált képen. A teszt kitöltése után kap egy százalékos eredményt. Ha ezek után kíváncsi, hogy mit rontott el, akkor kapja meg a bal oldali szerkezetet. A felső kép az a generált kép, amit kérdésként kapott, tőle balra a válasza, jobbra a helyes válasz, alatta pedig ugyanaz a kép, csak lehetősége lesz daltonizáló szűrőt alkalmazni rá.



3.3 ábra: Ishihara teszt

A 3.4 ábrán látható az élőképes módok tervezett szerkezete. Teljes területen látható az élőkép úgy, hogy a kiválasztott felbontás arányát megtartja. Itt futás közben lehet ki-be

kapcsolni a szűrőt, és a rádiógombokkal lehet szűrőt választani. Az éppen látott képet el is lehet menteni, ekkor az elkészült kép az alapértelmezett képek mappába fog kerülni.



3.4 ábra: Az élőképes nézet terve

3.3. Az alkalmazás tervezett felépítése

Az alkalmazás építése során törekedtem az androidos konvenciók követésére. Az alkalmazás részei főleg androidos komponensek osztályaiból származnak le. Grafikus megjelenési szinten, és kód szinten is az `Activity`-k álltak a középpontban, köréjük szerveztem a kódot. Ennek fő oka, hogy a program által használandó API függvények többsége csak `Activity`-ből, esetleg `Fragment`-ből érhető el. Készítettem teljesen egyedi osztályokat is, ezek főleg olyan részfeladatokat megvalósítanak, amik nem igénylik az androidos osztályból való leszármazást. Emellett olyan funkciókat is kiszerveztem ilyenekbe, amelyekre több helyen is szükség volt, például a jogosultság ellenőrzés, és elkérés.

Főként az Android rendszer beépített GUI elemeit használtam. Maga a GUI felépíthető lett volna csak Java kód használatával, de ezt kerültem. A Java kódból felépített GUI nehezebben áttekinthető, emiatt nem ajánlják. Helyette inkább az xml alapú GUI felépítést követtem, mely sokkal rugalmasabb és áttekinthetőbb, később pedig könnyebben módosítható. Szükségem volt egy három állapotú gombra a három féle szűrő közötti váltáshoz. Ilyennel az Android beépítetten nem rendelkezik, ezt az elemet egyéb forrásból emeltem be.

Két külső könyvtárat használtam. Az egyik a Glide, amely a képek kezelésében nyújt jelentős segítséget. Bár `Activity`-k többégében nincs túl sok kép egyszerre betöltve,

így azokban nem kellett használnom, de volt egy pont, ahol feltétlenül szükségem volt rá, ez pedig az állóképes daltonizáló rész, ahol több kép is megjelenik egyszerre. Itt fontosabb, hogy optimális legyen a memóriafelhasználás, mint ott, ahol csak 1-2 kép jelenik meg. A másik könyvtár pedig valójában nem egy könyvtár, hanem ezek egy csoportja: Firebase. Itt szükségem volt 3 könyvtárra is, az egyik az adatbázisnál a bejelentkezésben segít, a másik magát az adatbázis kezelést végzi, a harmadik pedig egyezektől független könyvtár, az esetleges alkalmazás összeomlásokat gyűjti össze, és tölti fel a Firebase Crash Reporting felületére.

4. Szűrők használata a programban

4.1. Daltonizáló szűrő

Ez a szűrőtípus az eredeti színes képet olyan formára képes hozni, hogy azon adott színtévesztéssel rendelkező személy is meg tudja különböztetni azokat a színeket, amelyeket az eredeti képen nem tudna megtenni. Az ötlet nem új, többen foglalkoztak már ilyen szűrők készítésével, így már meglévő megoldásokat kerestem, és néztem meg, hogy milyen végeredményt szolgáltatnak. Mivel nem vagyok személyesen érintett a színtévesztés terén, ezért igyekeztem mások által előre elkészített mintaképeken kipróbálni az egyes megoldásokat. Kiindulópontként a Vischeck daltonizálással foglalkozó oldala [11] szolgált. Itt több színes eredeti és daltonizált kép van, így itt jól megfigyelhető, hogy milyen változtatásokat kell elvégezni a képen. Az oldal szerint kétféleképpen lehet jól átalakítani a képet. Az első módszer szerint az érintett színek közötti kontrasztot kell növelni, a másik módszer szerint meg kell vizsgálni a képet, keresni kell az adott problémás színek által érintett területeket. Ezeket kell aztán a színtévesztők számára jól elkülöníthető színekre cserélni.

A Daltonize.org oldalon található több nyelven már elkészített daltozináló szűrő[1]. Én az itt található javascriptes megoldást veszem alapul. Ez a jelenleg elfogadott algoritmust használja. Ennek első lépése, hogy a forrásként kapott eredeti RGB színterű képet az ennek megfelelő dikromatikus változatra alakítja. Ehhez az LMS transzformációt alkalmazza, aminek lényege, hogy a színeket a csapok érzékenysége, ingerlékenysége szerint bontja szét, innen ered az LMS rövidítés is (Longwave, Middlewave, Shortwave). Egészséges látásúaknál szükség van mindhárom komponensre, hogy egy színt leírjunk, azonban azoknál, akiknél valamilyen gond van valamelyik csappal, csupán két komponens áll rendelkezésre, hogy leírjunk egy színt. Ehhez az alábbi mátrix szorzás elvégzésére van szükség:

$$\begin{bmatrix} L \\ M \\ S \end{bmatrix} = \begin{bmatrix} 17.8824 & 43.5161 & 4.1193 \\ 3.4557 & 27.1554 & 3.8671 \\ 0.02996 & 0.18431 & 1.4670 \end{bmatrix} \begin{bmatrix} R \\ G \\ B \end{bmatrix}$$

[1] http://www.daltonize.org/2010/05/lms-daltonization-algorithm.html

Ezzel a művelettel kapjuk meg RGB-ből az LMS értékeket, ezt követően kell a színteret lecsökkenteni az adott színtévesztéstípusnak megfelelően. Protanóp esetben ez az alábbi módon történik:

$$\begin{bmatrix} Lp \\ Mp \\ Sp \end{bmatrix} = \begin{bmatrix} 0 & 2.02344 & -2.52581 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} L \\ M \\ S \end{bmatrix}$$

Itt LpMpSp lesz a csökkentett színterű LMS kép. A módszer deuteranóp és tritanóp esetben is így működik, de ott mások lesznek a mátrix értékei. Ezt követően vissza kell térni RGB alakra:

$$\begin{bmatrix} Rp \\ Gp \\ Bp \end{bmatrix} = \begin{bmatrix} 0.0809 & -0.1305 & 0.1167 \\ -0.0102 & 0.0540 & -0.1136 \\ -0.0003 & -0.0041 & 0.6935 \end{bmatrix} \begin{bmatrix} Lp \\ Mp \\ Sp \end{bmatrix}$$

A daltonizált kép előállításához szükség van egy hiba mátrixra. Ezzel tudjuk elkülöníteni azokat a színeket, amiket az adott színtévesztő nem lát helyesen. Az így megkapott kép már nevezhető daltonizáltnak, azonban ebben az esetben a módosítás minden színt érint. Alapvetően elég lenne csak azokat a színeket módosítani, amik problémát okoznak az adott színtévesztőnek, a többit pedig érintetlenül lehetne hagyni. Ennek megvalósításához alkalmaznak még egy korrekciós mátrixot is, ami a legtöbb esetben így néz ki:

$$\begin{bmatrix} 0 & 0 & 0 \\ 0.7 & 1 & 0 \\ 0.7 & 0 & 1 \end{bmatrix}$$

Ez a korrekciós mátrix már nem feltétlenül egységes, van olyan tanulmány, ahol más értéteket alkalmaznak. A Bielefeld egyetemen is foglalkoztak ezzel [12], ott konkrét színértékeken is vizsgálták a módszert, és ők szabályban azt határozták meg, hogy a korrekciós mátrix értékeinek összege 3 kell, hogy legyen. Ebben a tanulmányban az általam alkalmazottnál összetettebb eljárást alkalmaznak a daltonizálásra.

Ellenőrzésképp megvizsgáltam, hogy a javascriptes program által végzett módosítások mennyire vannak összhangban egyéb daltonizált képpel. Főleg a Vischeck oldalán lévő képeket vizsgáltam így, ahol tapasztaltam némi eltéréseket. Feltehetően ott összetettebb eljárásokat alkalmaznak. Már elérhető, és működő eszközt is vizsgáltam, ilyen volt a Chrome Daltonize[2], aminek szűrője szinte teljesen megegyezik azzal, amit én is

[2] https://chrome.google.com/webstore/detail/chrome-daltonize/efeladnkafmoofnbagdbfaieabmejfcf?hl=en

kiválasztottam, ugyanis az általa szolgáltatott végeredmények kifejezetten hasonlóak voltak ahhoz, amit a javascriptes megoldás is kínált. Ez számomra azért is hasznos, mert az ott lévő visszajelzések alapján már képet kaphatok arról, mennyire hasznos a szűrőnek ez a változata. A jelenlegi 4-es értékelés mindenképp pozitív, és olvasva a visszajelzéseket, több színtévesztő is pozitívan írt az alkalmazásról, bár természetesen voltak olyanok is, akiknek nem használt a szűrő.

4.2. A szimuláció

A fenti megoldás alkalmazásakor már megkaptam a szimulált képet is. Ha az eljárást megszakítom, mielőtt a hiba mátrix alkalmazásra került volna, akkor az ott előálló RGB értékek már egy szimulált képet adnának. Ennek ellenére nem az itt létrejövő képet használom fel a szimuláláshoz, hanem egy másik eljárást alkalmazok, amely a ColorOracle[3] nevű programban található meg.

Itt a protanóp és deuteranóp szimuláció előállítása hasonlóan történik. Első lépésként 256 darab gamma korrigált lineáris RGB értéket állít elő a 0-32768 egész tartományra. Ezeket használja fel a további számításokhoz. 3-3 értéket használ, amelyeket egész számokon végez el, majd a végén ezeket skálázza le a 0-255 tartományra.

A tritanóp szimulálás ebben a programban a GIMP 2.2-es verziójában lévő megoldással egyezik. Itt már kisebb az eltérés a javascriptes megoldáshoz viszonyítva, használ LMS transzformációt, de itt is végez gamma korrekciót, és skálázást.

Ezt a megoldást is összevetettem a Vischeck példáival, és elég pontos egyezést tapasztaltam. Szimulált képekből jóval több található online, így több mintán is meg tudtam nézni a szimulálás hatását. Az online elérhető képek többsége egyezett a ColorOracle szűrője által szolgáltatottal.

4.3. Szűrők implementálása, alkalmazása

4.3.1. A szűrők Java-ban

A megfelelő szűrőket egy paraméterezhető függvényben helyeztem el. Ennek a függvénynek adom át az eredeti képet reprezentáló `Bitmap` objektumot. Mivel az eredetileg betöltött `Bitmap` nem volt módosítható alapból, így a függvényen belül szükség volt egy másolatra, amely már módosítható lehet, így a `setPixel` segítségével

[3] http://colororacle.org

már tudom módosítani a `Bitmap` tartalmát. A megfelelő mátrixokat kétdimenziós `double` típusú tömbben tárolom. A teljes képen két egymásba ágyazott `for` ciklussal megyek végig, ahol minden egyes iterációban kikérek egy pixelt a `getPixel`-el, ezt követően következnek az algoritmusok által meghatározott lépések, végül pedig az információt visszaírom a `Bitmap` objektumba, és legvégül ezzel tér vissza függvény.

Állóképeken néztem meg először a függvény működésének eredményét. A szűrők a várt módon működtek, azonban a sebességgel gondok voltak, érezhetően lassan futott gyengébb és erősebb eszközökön egyaránt. Gyorsításként azzal próbálkoztam, hogy a függvényben nem készítettem másolatot az eredeti képről, hanem egy új üresbe írtam bele a pixeleket. Számottevő változást nem lehetett látni ezután se. Ezek után döntöttem a Renderscript használata mellett.

4.3.2. A szűrők Renderscriptben

A Renderscript egy Androidhoz készült keretrendszer számításigényes feladatok végrehajtásához. Képfeldolgozással kapcsolatos műveletekhez kifejezetten ajánlják. A Renderscript az Android NDK (Native Developer Kit) része. Legfőbb előnye, hogy képes kihasználni a párhuzamosítás előnyeit, a számítási feladatokat automatikusan osztja szét futásidőben a rendelkezésre álló CPU-k és a GPU között. Ezen a szinten már nem Java, hanem C nyelv a használatos. Az NDK a C++ nyelvet használja, a Renderscript viszont a C99-et veszi alapul.

A Renderscriptben való implementálás túl sokban nem tér el a Java-s megoldástól. Itt Java oldalon lehetőségem van a Renderscript API használatára, így a kommunikáció megoldott. Ahol használni szeretném, ott első lépésként inicializálnom kell egy `Renderscript Context`-et. Ezt követően létre kell hozni egy bemeneti és egy kimeneti `Allocation`-t. Bemenetre kerül az eredeti `Bitmap`, kimenetre pedig a módosított. Ezek után már nem kell mást tenni, mint „példányosítani” a szkriptet, ahol a típus mindig `ScriptC_` előtaggal indul, és ezt követi a szkript neve.

```
ScriptC_protanope script = new ScriptC_protanope(rs,
                           getResources(), R.raw.protanope);

script.forEach_calculate(mInPixelsAllocation,
                         mOutPixelsAllocation);

mOutPixelsAllocation.copyTo(this.daltonizedBitmap);
```

A konstruktor 3 paramétert vár, első a `RenderSrcipt` objektum, a második a `View` oldali erőforrás, a harmadik pedig a szkript id-je. Ezután tudom hívni a szkriptben lévő kernelt. A kernel olyan függvény a szkripten belül, ami közvetlenül átadható a `RenderScript` futtatókörnyezetnek, vagyis azok a számítások, amik ezen belül vannak, optimálisan, lehetőleg párhuzamosítva fognak lefutni. Két kernel típus létezik, az amit én használok, a `mapping`, vagy `forEach` kernel. Ennek lényege, hogy párhuzamosan hajtja végre a feladatát, automatikusan végigiterálva a neki adott `Allocation`-ön. A fenti második sorban a hívás ezért indul a `forEach_` előtaggal, amit a kernel neve követ. Minden ilyen kernelen belül elhelyeztem a megfelelő szűrő algoritmusa által meghatározott lépéseket. Esetemben a `calculate` kernel két paramétert vár, a bemeneti és kimeneti `Allocation` egy egységét, ez jelenleg a pixel. Itt a mátrixokat 3 elemű vektorokként kezeltem, ugyanis Renderscriptben nem lehet tömböket használni, csak 2-3-4 elemű vektorokat, de ez a céljaimnak megfelel. Az algoritmus végén a kapott `out` paraméterbe kell elhelyeznem az aktuálisan kiszámolt pixel értékeit. Az eredmény így az `mOutPixelsAllocation`-be fog kerülni, ahonnan kimásolom végleges helyére, a `daltonizedBitmap`-ba.

Ezt a megoldást is ellenőriztem, főként a sebesség érdekelt. A különbség nagy mértékben érezhető volt. A Java és Renderscript közötti különbség vizsgálata végett mindkét megoldást ellenőriztem kamera élőképen is. Míg VGA közeli felbontáson Java esetén durván szakadozó képet adott még egy erősebb készülék is, addig ugyanez a Renderscriptes megoldás során folyamatosabbnak bizonyult.

4.3.3. Az állóképes daltonizálás

Az állóképes módot a fő menühöz tartozó `MainActivity` osztály `startStillImageActivity` függvénye indítja. Ezen belül egy sima `Intent` objektummal dolgozom, nincs szükség semmilyen plusz adat átadásra, vagy speciális indításra.

A fenti `Intent` objektum a `StillImageExampleActivity`-t indítja. Ennek `onCreate` függvényében létrejönnek további `Intent` objektumok, gombok, és képek. Két `Intent` van, amik külső alkalmazást hívnak meg, egyik egy képkezelő alkalmazást, a másik egy kamera kezelőt. Ezek ugyanúgy `Intent` objektumok, de itt az `Intent` konstruktorának több paramétert is át kell adni. Szeretnék képeket megnyitni a telefon

háttértárolójáról úgy, hogy a felhasználó kényelmesen kiválaszthassa, melyik képet szeretné megnyitni, ennek megfelelően paraméterezem az `Intent`-et.

Ehhez tartozik egy gomb is, amelyet megnyomva megjelenik egy lista, ahonnan a felhasználó kiválaszthatja, hogy melyik alkalmazással szeretné a képkiválasztási műveletet elvégezni. A gomb az `Activity` megjelenését leíró xml fájlban van létrehozva.

A gombon megjelenő szövegre itt erőforrásként hivatkozok, de közvetlenül is beírhatnám ide a szöveget. Itt az erőforrás a `strings.xml`. Minden nyelvhez tartozhat egy, így könnyű többnyelvű alkalmazást írni, hiszen az Android mindig tudni fogja az elérési útból, hogy az adott nyelvez melyik xml-t kell erőforrásként használnia. Mivel jelenleg az alkalmazásom egynyelvű, így csak egy ilyen állományom van, amely a `res/layout/values/strings.xml` állomány. Ha később szeretném a jelenlegi angol mellett támogatni például a magyart is, akkor készítenem kell majd egy `res/layout/values-hu/strings.xml`-t is.

A gombnyomásra az alábbi műveletek hajtódnak végre:

```
StillImageExampleActivity.this.openGallery = true;
HelperMethods.checkAndAquireStoragePermission(
StillImageExampleActivity.this);

if (HelperMethods.storageOk) {
	startActivityForResult(galleryIntent,
RESULT_LOAD_IMG);
	StillImageExampleActivity.this.openGallery = false;
}
```

Alapesetben elég lenne egy `startActivityForResult` hívás, azonban ellenőriznem kell, hogy a megfelelő jogosultság megvan-e adva. Egy képfájl olvasásához külön engedély kell. A `HelperMethods` egy saját statikus osztály, melyben elhelyeztem egy ilyen engedély ellenőrző, és szükség esetén kérő függvényt. Ha a futás során ez a függvény már hívódott, és a felhasználó megadta az engedélyt, vagy már eleve meg volt adva az engedély, akkor a `storageOk` statikus logikai változó már igaz lesz, így indulhat a művelet. Ha nincs meg az engedély, akkor kérni kell. Ehhez egy ablak fog megjelenni, ahol lehetőség lesz megadni az engedélyt. Az erre adott válasz eredményét az ugyanezen `Activity`-n belül található `onRequestPermissionResult` függvényben kapom meg. Mivel kétféle engedély kérésére is szükség lehet itt, ezért a

gomboknál beállításra kerül a megfelelő logikai változó, hogy tudni lehessen, mire jött engedély megadó, vagy megtagadó válasz. Bár a függvényben átadott `permissions` tömb tartalmazza szöveges formában, hogy pontosan mire történt az engedélykérés, nem találtam olyan összehasonlítási alapot, amely minden Android verziónál jó eredményt adott volna. Például ezen tömb első eleme API 21-nél „`android.permission.WRITE_EXTERNAL_STORAGE`” string-et tartalmazta, de más API szintnél ez eltérő volt. Emiatt használok most két saját változót erre a célra. Ha a felhasználó megadta az engedélyt, akkor a `startActivityForResult` az `onRequestPermissionResult` függvényen belül fog meghívódni.

A fénykép készítésének megfelelő gomb ugyanilyen elven van felépítve.

Mindkét esetben az eredményt az `onActivityResult` függvényben kapom meg. Eredetileg a HTC eszközén tesztelve mindkét módnál ugyanolyan formában kaptam meg az `Uri`-t, amely megadja, hol van a kép, amit beolvashatok. Az eszközök többségénél, mint később felfedeztem, nem így van. Képkiválasztás esetén a `data` nevű `Intent`-ből az eddig tesztelt összes fizikai és virtuális eszköznél gond nélkül kapok megfelelő `Uri`-t, azonban fényképkészítésnél nem. Itt használhattam volna `ContentProvider`-t, és utána skálázást a jobb minőségű kép megjelenítésért, de maradtam csak a kisméretű kép átvételénél. Így abban az esetben érkezik `nullPointerException`, ha az adott eszköznél eltérő módon kell a képet átvenni. Siker esetén a beolvasott kép egy statikus változóba kerül, és létrejön egy `Intent` a `PicActivity`-nek, amelyben a szűrőket lehet kipróbálni. `Bundle` objektumként nem tudtam volna átadni az új `Intent`-nek, mert túl nagy, legfeljebb fájlba tudtam volna kiírni, és ennek útvonalát átadni, hogy majd ő is betudja olvasni, de ez felesleges plusz művelet.

A példaképeket egy `scrollView`-ban helyeztem el. Használhattam volna egyéb `Adapter`-es megoldást, viszont ez csak nagy elemszám kezelésénél nyújt előnyt, néhány képért felesleges lett volna. Ezen belül `TextView`-k, és `ImageView`-k vannak. Ezek is a nézethez tartozó xml fájlban vannak létrehozva.

Minden képhez tartozik egy `TextView` és egy `ImageView` is. Java-ban létrehozom a megfelelő `ImageView` objektumokat, amelyeket a Glide-al töltök fel.

Ha sikerült egyéni képet kiválasztani, akkor a `PicActivity` nyílik meg. Ezen két `ImageView` található, egyik a szűrővel ellátott kép, a másik pedig az eredeti. Található

még itt egy Button, amivel meg lehet osztani a módosított képet, és egy három állapotú gomb is, amivel az aktív szűrőt lehet kiválasztani.

Itt Jose Luis Honorato által elkészített három állapotú gombot használtam fel[4]. A gombot az alábbiak szerint használom:

```
MultiStateToggleButton filterButton =
(MultiStateToggleButton)findViewById(R.id.mstb_multi_id);
filterButton.setOnValueChangedListener(
new ToggleButton.OnValueChangedListener() {
    @Override
    public void onValueChanged(int value) {
        generatePic(value);
        Pic_Activity.this.generationType = value;
    }
});
filterButton.setButtonState(
filterButton.getChildAt(0),true);
```

A gomb példányosítása után közvetlenül kap egy OnValueChangeListener-t. Ezen belül az onValueChanged függvény paraméterében megkapja, hogy a gomb hányadik eleme lett megnyomva, a számozás 0-tól kezdődik. Ezt az értéket használom a generatePic függvényemben is, ahol a konkrét szűrő alkalmazása történik meg a képen. Az osztály egy tagjába is elmentem az értéket. Amikor megnyomnak egy gombot, akkor hívódik az onValueChanged, és a megfelelő paraméterrel hívom a generatePic függvényt is, így rögtön látható lesz az adott típusú szűrő hatása. Ezek után a setButtonState függvénnyel beállítom, hogy alapértelmezetten, amikor az Activity elindul, akkor a gomb melyik része legyen benyomott. Itt elkérem a 0. gombot, és benyomottságát igazra állítom.

A PicActivity onCreate függvényében átveszem az egyéni képet reprezentáló Bitmap objektumot. Ellenőrzöm a felbontását, ugyanis 4096x4096 az a maximális felbontás, amit ImageView-ra ki lehet tenni. Ha ennél nagyobb a felbontás, akkor leskálázom, az arányokat megtartva. Vagy skálázva, vagy anélkül, de készül egy szerkeszthető másolat a Bitmap-ról, amivel a generatePic függvény dolgozhat. Ez a már említett Renderscriptes függvényeket hívja a kapott paraméternek megfelelően.

[4] Jose Luis Honorato: Multistate toggle button
https://github.com/jlhonora/multistatetogglebutton

Amikor befejezte tevékenységét, akkor a végeredményt elhelyezi a megfelelő `ImageView`-ban.

A generált kép megosztható. Ehhez külön gomb tartozik, megnyomásra még mielőtt a megosztási folyamatért felelős műveletek lefutnának, ellenőrzöm a kép felbontását, ugyanis ha ez túl nagy, akkor sokáig tarthat a folyamat, erről egy felugró ablakban tájékoztatom a felhasználót.

A tesztek során előfordult, hogy bizonyos esetekben a szűrt képnek megfelelő `Bitmap` eltűnt a memóriából. Ez akkor fordulhat elő, ha egy gyengébb hardverrel rendelkező eszközön kis méretre helyezik az alkalmazást, majd egy másik, erőforrás igényes alkalmazást indítanak. Ilyenkor az Android a háttérben lévő alkalmazás által lefoglalt memória egy részét, vagy egészét felszabadítja annak érdekében, hogy a másik, előtérben lévő alkalmazás rendesen futni tudjon. Egy ilyen esemény után is, a háttérbe helyezett alkalmazás újból előtérbe hozható, amely elvileg ugyanabban az állapotban fogja várni a felhasználót, ahogy az a kis méretre helyezése előtt közvetlenül volt. Emiatt van szükség arra, hogy megnézzem, hogy a `daltonizedBitmap`, amiben az átalakított kép van, valóban létezik-e. Ha nem létezik, újra generálom, ha létezik, akkor eleve minden rendben volt.

Ha minden rendben volt, akkor indulhat a megosztási folyamat, amit `shareImage` függvény végez. A `shareImage` függvényben ahhoz, hogy a képmegosztás működjön, fizikailag is létre kell hoznom egy fájlt, amit átadok más alkalmazásnak. Jelenleg az alkalmazásom saját ideiglenes könyvtárában lévő `images` mappát használom, ha ilyen nincs, akkor létrehozom. Ezen belül készül egy `default_image.png` állomány, ami itt még üres. `FileOutputStream`-re van szükségem ahhoz, hogy a `Bitmap`-ban lévő bájtfolyamot a png állományba tudjam írni. Létrehozok egy ilyen objektumot, ami az előbbi png állományra fog mutatni. A `Bitmap` osztály rendelkezik egy `compress` függvénnyel, amely a `FileOutputStream`-re tudja küldeni a képi információt. Itt a png mellett akár jpg formátumot is használhatnék, eltérő minőségi beállításokkal, de itt célom volt a jobb minőség megtartása. Az információ kiírása után zárom a `FileOutputStream`-et.

Ezután létrehozom a fájlhoz tartozó elérési utat, amihez szükségem van a `ContentProvider` egy alosztályára, a `FileProvider`-re. Ennek segítségével fogom tudni átadni az alkalmazásom saját ideiglenes mappájában lévő állományt egy

külső alkalmazásnak. A FileProvider-t az AndroidManifest.xml-ben kell létrehozni. Itt meta-data-ként meg kell neki adni, hogy pontosan milyen elérési utat szeretnénk átadni, amelyet szintén egy xml-ben kell elkészíteni. Ezen belül kell megadnom az általam, még a Java oldalon megadott images mappát. A FileProvider segítségével létrehozott ContentUri, amely az elérési utat tartalmazza, átadásra kerül egy új Intent-nek. Ez az Intent egy kiválasztó felület lesz. Ebben setAction-el megadom, hogy küldeni szeretnék valamilyen tartalmat, utána megadom Extra-ként, hogy mit is szeretnék átadni, majd a setType-al megmondom, hogy milyen típusú adatról van szó. Így az Intent-ben olyan már feltelepített alkalmazások jelennek meg, amelyek képesek fogadni a megadott típusú állományt. Végül még a fejlécébe megadok egy megjelenítendő szöveget.

4.3.4. A szűrők alkalmazása élőképen

Az élőképes módért a LiveViewActivity felel. Indítása az állóképes módhoz hasonlóan történik. Működését nagyban befolyásolja a fő menüből elérhető beállítások, ugyanis itt lehet kiválasztani, hogy milyen felbontású élőképet szeretnénk.

A beállítások a Fragment osztály leszármazottja, a PreferenceFragment, mely kifejezetten beállítások megjelenítésére, kezelésére szolgál, ő a PrefFragment osztályom. Működése hasonló, mint az Activity-ké, vagy egyéb Fragment-eké, azonban itt a nézet felépítéséért felelős xml kissé eltérő szerkezetű, másképp épül fel ekkor a GUI. A PreferenceFragment-hez egy preferences.xml tartozik, ez határozza meg a PreferenceScreen-t.

A PrefFragment erőforrásként használja ezt az xml-t, ami egy listaszerű nézetet biztosít. A lista egyik eleme a ListPreference, amelyet kiválasztva újabb, felugró listát kapok. Ezt kérem el a PrefFragment onCreate függvényében, ezt a listát töltöm fel a kamera által támogatott felbontásokkal.

```
int paramsLength =
this.getArguments().getInt("ParamsLenght");
params = new String[paramsLength];
params = this.getArguments().getStringArray("CamParams");
String[] entryValues = new String[paramsLength];
for (int i = 0; i < params.length; i++) {
    entryValues[i] = Integer.toString(i);
}
```

```
list.setEntries(params);
list.setEntryValues(entryValues);
```

A `PrefFragment` itt már kész adatokat kap, amiket meg kell jelenítenie. Az adatokat a `MainActivity`-ből kapja egy `Bundle` objektumon keresztül, a `startSettingsFragment` függvényben. Itt a `getCamParams` függvény az, amelyik elkéri, és olvasható formában átadja a támogatott felbontásokat, a lényegi rész:

```
this.cameraParameters = camera.getParameters();
this.supportedPreviewSizes =
cameraParameters.getSupportedPreviewSizes();
final String[] s =
new String[supportedPreviewSizes.size()];
int i = 0;
for (Camera.Size size : supportedPreviewSizes) {
    s[i] = size.width + "x" + size.height;
    i++;
}
```

A tesztelt eszközök egy részén a `supportedPreviewSizes` növekvő sorrendben tartalmazta a támogatott felbontásokat, ezért választottam a fenti `ListPreference`-ben `defaultValue` attribútumként a nullát, mert így gyengébb hardveren is mindig a legkisebb felbontás indulhat automatikusan, nem fog akadni. Ha a felhasználó úgy érzi, jobb felbontást is elbír a készüléke, akkor nagyobb felbontást is beállíthat. A későbbi tesztek során azonban kiderült, nem minden eszköz adja vissza ilyen sorrendben a felbontásokat, volt olyan készülék, amely nulladikként a legnagyobb felbontásértéket adta vissza.

Az így megkapott adatokat veszi át a `PrefFragment`, amely tovább adja a `ListPreference`-nek, és a nekik megfelelő értékeket is előállítja, tovább adja.

Ennek eredményeképp kapok egy bejegyzést, amit kiválasztva egy listát kapok a felbontásokról, amelyek közül egyet kiválaszthatok. Már csak fel kell iratkozni (illetve adott esetben leiratkozni) a kiválasztás eseményére.

Ennek hatására a kiválasztási eseményről az `onSharedPreferenceChanged` függvényben értesülök, ahol jelzem a változást, és visszatérek a fő menübe. Ennek paraméterben kapok egy `key`-t, ami az adott `ListPreference` attribútumában megadott `key`, innen lehet tudni, hogy mely beállítás lett módosítva. Én most ezt nem figyelem, mert csak egy beállítási bejegyzésem van, de ha bővíteni szeretném a beállításokat a későbbiekben, akkor ezt is néznem kell majd. Esetemben most

lényegesebb a sharedPreferences paraméter, amelyben már elmentve megkapom, hogy milyen index lett kiválasztva. A sharedPreferences a program saját tárterületén tárolódik, csak ő férhet hozzá. Futás után is megmaradnak az itt tárolt értékek, ezért a beállítások tárolására kiválóan alkalmas. Természetesen a felhasználó ezt bármikor törölheti az alkalmazások beállításainál, ezért mindig fel kell készülni arra, hogy ha üres. Ebben a függvényben sose lesz üres, ugyanis a rendszer elmenti bele a szükséges értéket, és közvetlenül utána hívja meg a függvényt. Ezek után egy Toast üzenetet jelenítek meg, így a felhasználó tudja, hogy választása elmentésre került, és visszairányítom őt a fő menübe. Elhelyeztem még egy gombot is, amellyel vissza tud jutni a felhasználó a fő menübe úgy, hogy nem módosított a beállításokon. Ehhez tartozik a PrefFragmentUi, amely most csak ezt a gombot tartalmazza, illetve a hozzá tartozó függvényt.

A LiveViewActivity nézetének egy részét a live_view_layout.xml határozza meg. A látható gombokat és feliratokat tartalmazza. A megjelenő rádiógombokhoz saját megjelenést rajzoltam, ugyanis a téma szerinti megjelenés sok esetben nem látszódott, ezt mindig az élőkép tartalma határozta meg. Így olyan színkombinációjú gombot rajzoltam, ami a lehető legtöbb esetben jól látszik. Az onCreate függvényben példányosodnak a gombok, szövegek.

A lényegesebb eseményeket itt az onResume függvényben helyeztem el. Ezeknek ugyanis a program futásának esetleges megszakadása, majd újra indulása esetén is le kell futniuk. Első lépésként elkérem a beállított felbontás indexét:

```
SharedPreferences sharedPref =
PreferenceManager.getDefaultSharedPreferences(this);
mode = Integer.parseInt(sharedPref.getString("ChosenRes",
"0"));
```

A SharedPreference-ből lekérő függvények úgy működnek, hogy meg kell nekik adni egy alapértelmezett értéket. Ezzel fognak visszatérni, ha a lekérdendő kulcs mögött nincs érték eltárolva. Én a 0-t adtam itt meg, így ha a mode értéke 0 lenne ezek után, akkor elképzelhető, hogy nem volt felbontás állítás, így alapértelmezettként az első támogatott felbontással fog megnyílni az élőkép.

A következőkben elkérem a kamerát, és ha sikerült, akkor beállítom az autófókusz módot, ha van ilyen, illetve a megfelelő felbontás értékeket átadom neki, amik az élőkép

felbontását fogják meghatározni. Itt mindenképp pontosan olyan méretet kell neki megadni, amit támogat is, másképp hiba keletkezik.

```
this.cameraParameters = camera.getParameters();
this.cameraParameters.setFocusMode(
Camera.Parameters.FOCUS_MODE_CONTINUOUS_PICTURE);
this.supportedPreviewSizes =
cameraParameters.getSupportedPreviewSizes();
cameraParameters.setPreviewSize(
supportedPreviewSizes.get(this.mode).width,
supportedPreviewSizes.get(this.mode).height);
this.camera.setParameters(cameraParameters);
```

A kamerának úgy lehet paramétereket átadni, hogy először elkérjük tőle a paramétereit, majd az így kapott objektum függvényeit használjuk fel. Miután a kamera felbontását beállítottam, be kell állítsam az élőkép megjelenítésére szolgáló `FrameLayout` méretét is. Közvetlenül nem adhatom meg neki ugyanazokat a felbontás értékeket, amiket a kamerának is átadtam, hiszen ott lehetnek lényegesen nagyobb, és kisebb felbontások is a kijelző felbontásához mérten. Az élőképet viszont úgy szeretném megjeleníteni, hogy az a legjobban kitöltse a rendelkezésre álló területet úgy, hogy a kamera által szolgáltatott kép arányát is megtartsa. Ehhez először is tudom kell a jelenlegi képernyő felbontását. Ezt követően elkérem a megjelenítésre szolgáló `FrameLayout`-ot és paraméterét. Ezt a kamerához hasonlóan tudom paraméterezni.

```
preview = (FrameLayout) findViewById(R.id.camera_preview);
ViewGroup.LayoutParams layoutParameters =
preview.getLayoutParams();
```

Meg kell határozni a képernyő és a kamera képének képarányát, és ennek megfelelően történik a `FrameLayout` méretezése:

```
float previewRatio = ((float)
supportedPreviewSizes.get(this.mode).width) / ((float)
supportedPreviewSizes.get(this.mode).height);

float screenRatio = ((float) screenWidth) / ((float)
screenHeight);
float ratio;
if (previewRatio > screenRatio) {
    ratio = ((float) screenWidth) / ((float)
supportedPreviewSizes.get(this.mode).width);
} else {
    ratio = ((float) screenHeight) / ((float)
supportedPreviewSizes.get(this.mode).height);
}
```

```
layoutParameters.width = (int) (((float)
supportedPreviewSizes.get(this.mode).width) * ratio);
layoutParameters.height = (int) (((float)
supportedPreviewSizes.get(this.mode).height) * ratio);
preview.setLayoutParams(layoutParameters);
```

Miután a megfelelő méretek beállításra kerültek, példányosodik az `OverlayView`. Ennek az osztálynak a feladata a képi információ fogadása, alakítása. Ő szorosan együttműködik a `CameraPreview` osztállyal.

A `CameraPreview` paraméterként kapja meg az `OverlayView` példányát, és a megfelelő esemény bekövetkezésekor a megfelelő függvényét hívja meg. Létrehoz egy `SurfaceHolder`-t, amire a kamera kimenetét ráirányítja, és aminek eseményeire fel is iratkozik. Így a `SurfaceHolder` létrejöttekor hívódik a `surfaceCreated`, ahol a kamera kimenet beállítása történik. Másik fontos függvény a `surfaceChanged`, ez közvetlenül a `surfaceCreated` után hívódik, illetve olyan események után is, mint például az orientációváltás. Ezen a függvényen belül átadok a kamerának egy saját `PreviewCallback`-et. Itt egy `oneShotPreviewCallback` kerül beállításra, ennek lényege, hogy csak az első képkocka megérkezésekor hívódik a benne lévő `onPreviewFrame`. Ezen belül történik egy inicializálás szükség esetén, amit az `OverlayView` megfelelő függvénye végez, és egy újabb ugyanilyen feliratkozás, itt viszont tényleges `PreviewCallback` kerül beállításra, ami azt jelenti, hogy a benne lévő `onPreviewFrame` minden egyes érkező képkocka esetén hívódni fog.

Ezen belül adódik át az `OverlayView` objektumának a képi adat egy dimenziós bájt tömbként. Itt a kép YUV színtérben van kódolva. Ezt először át kell alakítani RGB alakra, aminek végeredményeként ugyanúgy egy dimenziós bájt tömböt kapok, csak itt már RGB formában. Ebből készül egy `Bitmap`, amivel már lehet dolgozni. Innentől megfelelő paraméterek alapján történik a szűrők alkalmazása (szimulálás, vagy daltonizálás, és a 3 típus közül melyik). A szűrők itteni használata lényegében megegyezik az állóképes módnál bemutatottal. Egy esetben van eltérés. A szimulációnál készítettem olyan szkriptet, ami plusz paramétert is vár.

Ehhez a szkriptet egy kicsit másképp kell előkészíteni. Protanóp és deutaranóp között csupán az előre elkészített paraméterekben van különbség, így a két szimulálást ugyanaz a szkript végzi, csak más értékekkel. A paraméter átadást úgy oldottam meg, hogy az összes paramétert elhelyeztem egy `int` tömbben. Itt az első három paraméter a

deutaranóp szűréshez kell, a második három a protanóphoz. Az utolsó paraméter fogja azt meghatározni, hogy melyik hármat kell felhasználni. Java oldalon deuteranóp esetben így történik a szkript használat:

```
inputData[6] = 0;
ScriptC_red_green_filter script =
new ScriptC_red_green_filter(
rs, getResources(), R.raw.red_green_filter);
Type t = new Type.Builder(rs,
Element.I32(rs)).setX(7).create();
param1 = Allocation.createTyped(rs, t);
param1.copyFrom(inputData);
script.set_inputData(param1);
script.forEach_calculate(mInPixelsAllocation,
mOutPixelsAllocation);
mOutPixelsAllocation.copyTo(mBitmap);
```

Első sornál a 0 alapján fogja tudni a szkript, hogy az `inputData` első három értéke fog kelleni. `Type` segítségével foglalok helyet az `inputData`-nak, hét darab 32 bites integer értéknek megfelelő hely kell. A `param1` is egy `Allocation` objektum, amelybe így bekerül az `inputData`, és átadom ezt is a szkriptnek. Szkript oldalon a paraméterek feldolgozása így történik:

```
int k1,k2,k3;

if (rsGetElementAt_int(inputData, 6) == 0) {
    k1 = rsGetElementAt_int(inputData, 0);
    k2 = rsGetElementAt_int(inputData, 1);
    k3 = rsGetElementAt_int(inputData, 2);
} else {
    k1 = rsGetElementAt_int(inputData, 3);
    k2 = rsGetElementAt_int(inputData, 4);
    k3 = rsGetElementAt_int(inputData, 5);
}
```

Mindez az `OverlayView onDraw` függvényében zajlik. Amikor egy képkocka feldolgozása megtörtént, az kirajzolódik a `Canvas`-re. Minden egyes képkocka érkezésekor a `CameraPreview` hívni fogja az `invalidate` függvényt, ami az `onDraw` újbóli hívásához vezet. Előfordulhat, hogy a kamera gyorsabban szolgáltat képet, mint ahogy az `onDraw`-beli feladatok be tudnának fejeződni. Ezért ott használok egy `isProcessing` nevű logikai változót, ami a függvény elején igaz lesz, a végén hamis, és új feldolgozás csak hamis érték esetén fog indulni.

A megfelelő szűrő kiválasztásáért a rádiógombok és a ki/be kapcsoló gomb függvényei felelnek, illetve szimulálás/daltonizálás közötti kapcsolást a `MainActivity`-ből kapott `Bundle` objektumban lévő logikai változó határozza meg, ami a `LiveViewActivity` `onResume` függvényében kerül beolvasásra.

Ebben a módban is lehetőség van kép készítésére. Ekkor az aktuálisan látható kép a megadott szűrővel és a megadott felbontással kerül mentésre az alapértelmezett képek mappába. Az ehhez tartozó függvény a `snapImage`, ami a jogosultság ellenőrzés/kérés után hívódik meg, ha minden rendben. A fájl létrehozás az állóképes módnál történő fájlátadásnál bemutatott módon történik annyi eltéréssel, hogy itt más az elérési út. Fájlnévként törekedtem mindig automatikusan egyedit megadni, a készítés dátuma ebben az esetben megfelelőnek bizonyult.

A kamerát, mint erőforrást a futtatás megszakadásakor el kell engednem. Ezt az `onPause` függvényben teszem meg. A kamera elengedése előtt le kell iratkozni az összes eseményről.

5. Ishihara teszt megvalósítása

A megfelelő menüpont kiválasztásakor először a `BlindTestStarter Activity` indul el. Itt egy szöveg írja le a funkció működését, és azt, hogy mi a feladata a felhasználónak. Ezen kívül itt lehet még kikapcsolni egy pipával az eredmény anonim feltöltését. Ez a kiválasztás megjegyzésre kerül `SharedPreferences` segítségével, alapértelmezett értéke igaz. Ennek megfelelően kell mindig a pipát beállítanom.

5.1. A teszt

Ezután jut a felhasználó el a `BlindTestActivity`-re. Itt egy saját `View`-ban jelenik meg maga a generált ábra, az ezért felelős osztályom a `TestImageView`. Felette jelenik meg maga a kérdés, alatta pedig rádiógombok és `ImageView`-k formájában a válaszlehetőségek, ezek alatt pedig egy gomb, amivel tovább lehet lépni.

Az ábráknak 10 féle színkombinációt gyűjtöttem össze, ebből 7 a hivatalos színkombináció, itt Ishihara könyvében lévő ábrákat vettem alapul [9]. Ezek nem 100%-osan pontosak, a digitális anyagból 1-1 kör átlagolt színértékét használtam. Ezeket az értékeket hexadecimális formában `String` tömbökben tároltam el, külön a háttér színeit (`offColors`), és külön az ábra színeit (`onColors`).

30 darab ábrát szeretnék megjeleníteni egymás után, véletlen szerű sorrendben és színkombinációkkal. A generálás alapjául szolgáló forrás képek 1 bites png állományok, amelyek a `res/drawable` mappában vannak. Létrehozok két 30 elemű `Integer` tömböt a konstruktorban, amelyek a képek sorrendjét és a színkombinációk sorrendjét határozzák meg.

Az ábra generálása az `onDraw` függvényben történik. Több eszközön tapasztaltam, hogy már az `Activity` első betöltése során kétszer hívódik az `onDraw`, illetve más esetekben is hívódhat, ami jelen esetben felesleges. A kép szükségtelen újra generálását úgy szüntettem meg, hogy a generált `Bitmap` objektumot eltároltam, és hívódáskor ha nem szükséges, akkor változatlanul rajzolódik ki az `onDraw` paraméterében kapott `Canvas` objektumra. Éppen emiatt bevezettem egy saját `Canvas` objektumot `canvasToDraw` néven, ennek a segítségével rajzolok. Ha új ábrára van szükség, akkor ezt törlöm, innen tudja majd a függvény, hogy új generálás kell, és nem a már meglévő `Bitmap`-et kell újra kirajzolni a `Canvas`-re.

A függvény először létrehozza a szükséges objektumokat, beállítja a megfelelő méreteket.

```
bitmap = this.createBitmap(imgIndex);
int width = getWidth();
int height = getHeight();
bitmapToDraw = Bitmap.createBitmap(width, height,
Bitmap.Config.ARGB_8888);
canvasToDraw = new Canvas(bitmapToDraw);
bitmap = Bitmap.createScaledBitmap(bitmap, width, height,
false);
```

Legelőször a `bitmap` nevű objektumomba betöltöm a generálás alapját képező ábrát. A `createBitmap` függvényem az `imgIndex` alapján betölti a megfelelő png képet. A `width` és `height` változóimba lekérem az `onDraw` paraméterébe átadott `canvas` méreteit. Ezt én határozom meg az `Activity`-hez tartozó xml fájlban, ahol a `layout_width` és `layout_height` attribútum szabja meg a méretet. Density pixelben adtam meg a méreteket, ennek előnye, hogy az így megadott méret felbontásfüggetlen tud lenni, látványra ugyanakkora helyet fog elfoglalni egy kicsi felbontású képernyőn, és egy nagyfelbontásún is. Viszont emiatt amikor elkérem ennek magasságát és szélességét, akkor más eredményeket fogok pixelben kapni az eltérő eszközökön. Ezzel számolnom kell.

Az ábrát felépítő köröknek meghatározok egy maximális sugarat, amely minden felbontáson a megfelelő érzetet kelti. Itt több eszközön is lefuttattam az algoritmust, közben az eredményt összevetettem egy hivatalos Ishihara ábrával, így született meg a 37,5-es érték, amit használok:

```
radius = (int) Math.round(getWidth() / 37.5);
```

Létrehozok még egy `Paint` objektumot is, ezzel rögtön be is színezem a `Canvas` hátterét fehérre, de később is ugyanezt az objektumot fogom színezére használni:

```
Paint paint = new Paint();
paint.setStyle(Paint.Style.FILL);
paint.setColor(Color.WHITE);
canvas.drawPaint(paint);
```

A könnyebb kezelhetőségért készítettem egy saját `Circle` osztályt, aminek minden egyes példánya 1-1 kis kört fog meghatározni. A középpont koordinátáját és a kör sugarát tárolom el. Mivel a generálás során tudnom kell, hogy hol van már letéve kör, vagyis hova tudok úgy kört letenni, hogy az ne fedjen le egy másikat, így a köröket egy 1000 elemű tömbben tárolom el. A lényegi generálásért egy `while` ciklus felel. Ezen belül

adott sugárérettel megpróbál köröket letenni, majd ha már nem fér el több, akkor csökkenti a méretet és tovább próbálkozik. Maximálisan egy adott sugárral 200-szor próbálkozhat, és természetesen van minimális sugárméret is. Minden egyes kis kör esetén véletlenszerűen meghatározásra kerülnek a középpont koordinátái. Ha az ezek által meghatározott pont az ábra által meghatározott körön belül helyezkedik el, és ebből a pontból a jelenlegi sugárérettel lehet úgy kört rajzolni, hogy az ne fedje a másikat, akkor az adott véletlenszerű kis kör mentésre kerül. Az átfedés vizsgálata:

```
boolean noOverlap = true;

for (int j = 0; j < i; j++) {
    if (circles[j] != null) {
        float dist = (float)    Math.sqrt(
        Math.pow((circles[j].x - x), 2) +
        Math.pow((circles[j].y - y), 2)) -
        circles[j].radius - radius;
        if (dist < 0) {
            noOverlap = false;
        }
    }
}
```

Itt az `i` változó mindig a `circles` tömb legutolsó objektumának indexe. Így nem is fordulhatna itt elő az, hogy a `circles[j] null` lenne, de azért a biztonság kedvéért ezt is ellenőrzőm. Minden már meglévő körre megnézem, hogy a kör középpontja, és a jelenlegi véletlenszerű pont távolságából az adott kör sugara és a jelenlegi sugárméret különbségét kivonva pozitív értéket kapok-e. Ha igen, akkor a jelenlegi középponttal és sugárral az új kis kör elfér a vizsgált mellett. Ha nem volt egyáltalán negatív érték, akkor a `noOverlap` logikai változó igaz marad, és a kis kör a jelenlegi adatokkal példányosodhat, ellenkező esetben új véletlenszerű x és y koordinátákra lesz szükség. Annyi korlátozást szabtam itt még meg, hogy 10-es sugár felett csak 20 darab kör keletkezhet, másképp túl kevés hely maradna a kisebb köröknek, és az elkészült kép nem adna ugyanolyan érzetet, mint az eredeti Ishihara ábrák. Ezt követően megnézem, hogy a jelenleg választott koordinátán a kép pixele fehér-e vagy fekete. Először a fekete színt keresem, mert az alkotja magát az ábrát:

```
Color.red(bitmap.getPixel(x, y)) == 0
```

Példányosítom a kört, színt váltasztok neki az előre megkevert `typeIndexes` tömb alapján, és rárajzolom a `Canvas`-re. Ezt követően ugyanez a folyamat zajlik le a fehér

színre is. Először nem választottam szét ilyen szinten a fekete és a fehér színek kezelését, először egy `else` ágban dőlt el, hogy milyen színű lesz a kör. Ezzel az volt a baj, hogy így elég kevés olyan kör volt, ami az ábrát alkotta. Így viszont maximalizálni tudom az ábrára kerülő körök számát, és a maradék helyre kerülhetnek a hátteret alkotó körök.

A kérdésre 4 válasz közül lehet választani, ezekből egyik az „egyik se”. Úgy döntöttem, hogy ez csak nehezítés legyen, így ez sose lesz jó válasz. Minden egyes ábra esetén máshol lesz a jó válasz, ez is az `onDraw`-ban dől el, miután az összes kis kör létrejött és kirajzolódott:

```
correctAnsNum = (int)Math.round(Math.random() * 2);
```

Ezután választok még kettő másik válaszlehetőséget, ami ettől mindenképp eltér. Miután a 3 lehetőség eldől, ezeket meg kell jelenítenem a megfelelő rádiógombok felett, a megfelelő `ImageView`-k megkapják a megfelelő `Bitmap`-eket. A jó válaszhoz már megvan az eredeti png kép a `bitmap` objektumban, a másik kettőt pedig a véletlenszerűen kiválasztott indexek alapján a `createBitmap` függvényemmel töltöm be.

A `BlindTestActivity`-ben történik a „tovább” gomb és a rádiógombok kezelése. A rádiógomboknál minden rádiógomb lenyomásakor beállítódnak a megfelelő logikai változók, amik alapján mindig tudni lehet, hogy melyik van kijelölve. A gombkijelölést, illetve annak levételét most kézzel kell megtennem. Alapból, ha xml-ben `RadioGroup`-ba teszem a rádiógombokat, akkor a rendszer alapból csak egy gombot enged egyszerre kijelölni, minden kijelölésnél a másik gomb kijelöltségét automatikusan leveszi. Ez a hatás sajnos megszűnt azzal, hogy `LinearLayout`-okba ágyaztam őket és az `ImageView`-kat, hogy a képek mindig pontosan a megfelelő gomb felett jelenjenek meg. Így viszont kézzel kell a kijelöléseket kezelnem.

A „tovább” gombhoz rendelt `onClick` függvénynek több feladata is van. Egyrészt mindig frissíti azt a `TextView`-t, ami az aktuálisan generált kép sorszámát mutatja, másrészt számolja a jó válaszokat, és menti a rossz válaszokat is. Először a `testImageView` `canvasToDraw` objektumát `null`-ra állítja, innen fogja tudni a `testImageView` `onDraw` függvénye, hogy új ábrát kell generálni. Ezek után megnézi, hogy melyik rádiógomb volt kijelölve, és ha ez megfelel a `testImageView` `correctAnsNum` változójának, akkor a `testImageView` helyes válaszok számát

tároló `correctAnsCount` változóját növeli. Rossz válasz esetén elmentem az alkalmazás ideiglenes könyvtárába png formában a generált képet, a felhasználó által adott rossz választ, és a jó választ is. Ezeket majd a `ReviewActivity` fogja felolvasni, ahol a felhasználó megnézheti a rossz válaszait. Az `onClick` függvény ezek után alapállapotba állítja a rádiógombokat, növel egyet a legenerálandó kép indexén, és a `testImageView`-nak meghívja az `invalidate` függvényét, ami az `onDraw` függvényének a meghívásához fog vezetni, és új ábra fog generálódni. Ha a képindex eléri a 30-at, akkor meghívja a `BlindTestResultActivity`-t.

5.2. A teszteredmény megtekintése

A `BlindTestResultActivity` indítása előtt az őt indító `Intent` megkapja a százalékszámításhoz szükséges jó válaszok számát „`percentage`” kulcs alatt, a rossz válaszok számát pedig `SharedPreferences`-be mentem, hogy máshonnan is elérhető legyen, ne csak a következő indítandó `Activity`-ben.

A `BlindTestResultActivity` az `onCreate` függvényében elkéri a jó válaszok számát, kiszámolja a százalékos eredményt, majd megjeleníti azt. Az anonim adatfeltöltés is itt történik, amennyiben a felhasználó ehhez hozzájárult. Ennek ellenőrzése az `onCreate`-en belül a megfelelő `SharedPreferences` változó lekérésével történik.

A megfelelő magyarázatokat tartalmazó `TextView`-k mellett található itt még egy gomb is, amivel a hibás válaszokat lehet megnézni, illetve azt, hogy mi lett volna a jó válasz, és hogy az adott ábra daltonizálva hogyan néz ki.

A hibás válaszokat csak akkor lehet megtekinteni, ha volt egyáltalán, és nem hibátlanra töltötte ki a felhasználó. Az x változó a százalékos eredményt tartalmazza, ha ez 100 volt, akkor hibátlan a teszt, nincs mit megmutatni. Ellenkező esetben indul a `ReviewActivity`.

Ezen összesen 4 darab `ImageView`, 2 darab gomb, 1 darab három állapotú gomb, és a megfelelő feliratok találhatóak. A gombokkal lapozni lehet, a három állapotú gombbal a daltonizálást lehet állítani, a képek pedig a kérdést, a jó és a rossz választ, illetve a kérdés daltonizált változatát tartalmazzák.

A képek beállításáért és feltöltéséért a `refreshView` függvény felel. A függvény a teszt kitöltése során az ideiglenes könyvtárba mentett képeket tölti be, és állítja be a megfelelő `ImageView`-nak. A `selectedImageIndex` mutatja, hogy melyik képet kell betölteni. Ennek beállítását, és a `refreshView` hívását a léptető gombok `onClick` függvényei végzik.

6. Firebase

A Firebase a Google által nyújtott szolgáltatáscsomag, melynek elérhető ingyenes és fizetős változata egyaránt. Egyrészt kibővíti a Google Play Console által nyújtott szolgáltatásokat, másrészt újakat is ad hozzá.

6.1. Realtime Database

Ezt a szolgáltatást használom a teszteredmények anonim gyűjtésére. Maga az adatbázis Json felépítésű, ahol kulcs-érték párokban tudom az adatokat eltárolni. Ennél magasabb szintű adatkapcsolatokra itt nincs is lehetőség, de nincs is rá szükségem. Esetemben a kulcs alapján lehet eldönteni, honnan és mikor érkezett a feltöltés, az érték pedig az eredményt fogja tartalmazni. A megfelelő műveletek végrehajtása után a feltöltési tranzakciók automatikusan lefutnak, amennyiben volt élő internetkapcsolat. Ennek hiányában viszont a dokumentáció szerint később se fognak lefutni a feltöltések, vagyis a tranzakciók nem kerülnek elmentésre, erről nekem kell gondoskodnom. Arra nincs lehetőség, hogy eldöntsem, sikeres volt-e a tranzakció, arra viszont van, hogy ellenőrizzem, van-e élő internetkapcsolat. Sajnos, az erre szolgáló Firebase hívás csak wifi kapcsolat esetén működött helyesen, mobilinternet esetén hiába volt internetkapcsolat, mindig azt jelezte, hogy nincs kapcsolat. Emiatt csak úgy tudtam volna ellenőrizni a kapcsolat, és a küldés sikerességét, ha feltöltés után rögtön meg is próbálom letölteni a tartalmat. A wifi és mobilkapcsolat bizonytalansága miatt viszont ekkor előfordulhat, hogy bár a feltöltés sikeres volt, a letöltés már nem sikerül, így hibásan azt feltételezné program, hogy a feltöltés nem sikerült, pedig sikeres volt. A Firebase API egyébként többször is megpróbálkozik a feltöltéssel, amennyiben úgy érezné, hogy a szükséges DNS kérés nem volt sikeres.

Az adatfeltöltést az `upload` nevű statikus függvényem végzi. Azért statikus, hogy bárhonnan könnyen hívható legyen.

A függvényben először létrehozok egy `FirebaseAuth` objektumot. Erre azért van szükség, mert azonosítással lehet csak hozzáférni az adatbázishoz. Itt lehetőség van Facebook, Google, és egyéb szolgáltatók által bejelentkezni, de esetemben erre egyáltalán nincs szükség, mivel én anonim módon gyűjtök adatot, nem szeretném beazonosítani a felhasználókat. Így én a Firebase oldalon az anonim hitelesítést engedélyeztem, amelyet a Realtime Database Rules részénél lehet megadni:

```
{
  "rules": {
    ".read": "auth != null",
    ".write": "auth != null"
  }
}
```

Itt lehetőségem lenne meghatározni, hogy ki írhat és olvashat, viszont ezzel a szabállyal nem határozok meg korlátozást erre nézve. Hibába állítottam ezt így be, szükség van ennek ellenére a `FirebaseAuth`-ra.

El kell kérnem még az adatbázisom referenciáját, amin keresztül majd írni és olvasni lehet. A kapcsolódási adatokat a `google-services.json` tartalmazza, amelyet a projekt `app` könyvtárában kell elhelyezni. Ezeket kézzel is meg lehet tenni, de az Anrdoid Studióba épített Firebase modul automatikusan is meg tudja csinálni. A referencia elkérésénél a `.info/connected` magadása nem kötelező, üresen is hívható a függvény, illetve más paramétert is elfogad. Ennek a megadásával viszont a `connectedRef` objektumtól meg lehet kérdezni, hogy van-e internetkapcsolat. Ezt a lehetőséget a fentebb leírt hibák miatt nem használom. Ehhez az objektumhoz most egy `ValueEventListener`-t csatolok. Elég, ha ez egyszer hívódik, ellenkező esetben minden írás és olvasás esetén meghívódna, amire most nincs szükség. Az `onDataChange` így jelenleg csak egyszer fog meghívódni, amin belül kísérletet teszek a bejelentkezésre. Itt ismét szükség lesz egy `Listener`-re, ami itt most a bejelentkezési folyamat végén fog aktivizálódni. A benne lévő `onComplete` függvényben információt kapok, hogy a feladat lefutása sikeres volt-e. Itt a sikert nem úgy kell értelmezni, hogy a bejelentkezés sikeres volt-e, hanem úgy, hogy futás idejű hiba nem történt. Sokszor ellenőriztem, de a `task.isComplete()` és `task.isSuccessful()` egyszer se adott hamis értéket, akkor se, ha nem volt internetkapcsolat.

Mivel több json is lehet az adatbázisban, így meg kell mondanom, melyikre lesz szükségem. Az se baj, ha nem létezőt adok meg, mert akkor első íráskor a rendszer létrehozza azt. Nekem összesen csak egy darabra lesz szükségem, aminek a gyökéreleme a „`message`”. Ezen belül érkezhetnek az eredmények, ahol a kulcs értékek az eszköz márkájából, modelljéből, az Android API szintjéből, és a küldés idejéből állnak össze. Így biztosan minden kulcs teljesen egyedi lesz. Természetesen az se lenne baj, ha ugyanaz a kulcs többször kerülne beküldésre, csak ekkor felülírás történne, amit nem szeretnék. A tényleges feltöltési folyamat a `setValue` hívás után kezdődik, és ha volt élő

internetkapcsolat, akkor pár másodpercen belül az információ meg is jelenik az adatbázisban.

6.2. Crash Reporting

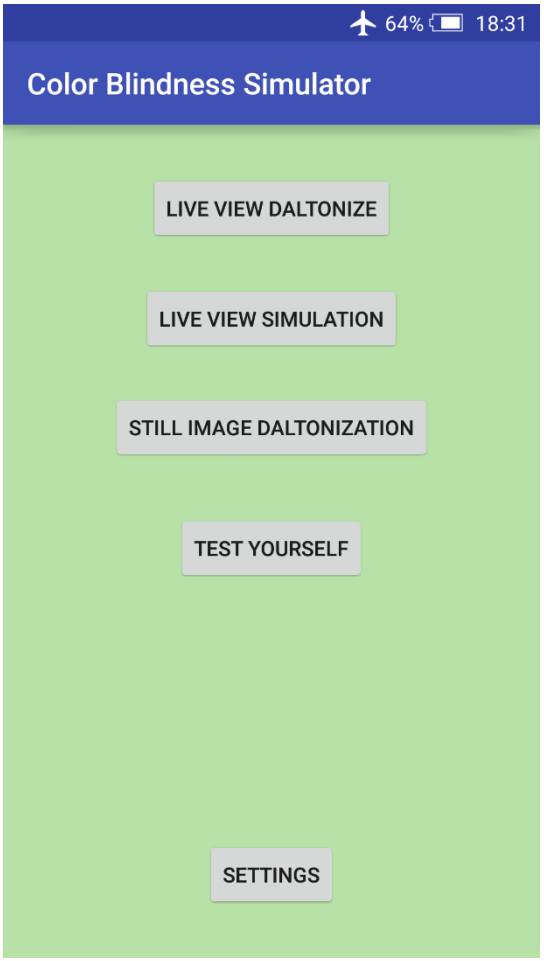
Hasonló szolgáltatással a `Google Developer Console` is rendelkezik. A lényeg, hogy ha az alkalmazás összeomlik akkor a veremlenyomat eljuthat hozzám, így tudomást szerzek az esetleges hibás működésről, és a veremlenyomat által a hiba forrását is megtudom. Amikor az alkalmazás lefagy, vagy összeomlik, akkor az Android megjelenít egy üzenetet a fagyás tényéről, és ugyanitt a felhasználó hibajelentést küldhet, amelynek része lesz a veremlenyomat is. Az így elküldött hibaüzenetek jelennek meg nálam a `Developer Console`-on.

A `Firebase Crash Reporting` ennél sokkal többet nyújt. Az alkalmazás összeomlásakor automatikusan küld hibajelentést, ahol a veremlenyomat mellett egyéb hasznos adatokat is megkapok. A `Developer Console`-on a készülék típusát és az Android verziószámát kaptam meg, a `Crash Reporting`-on kereszül ezek mellett megkapom még többek között a szabad RAM mennyiségét, az akkumulátor töltöttségi szintjét és az internetkapcsolat típusát (mobilinternet vagy wifi).

Saját `Exception` használatára is van lehetőség, ezek is automatikusan feltöltődnek a `Crash Reporting` felületére. Főleg hagyományos kivételek lekezelésénél használom ezeket, így értesülök arról, hogy az adott esemény milyen gyakran következik be, például így értesülök arról, hogy mennyire gyakran nem sikerül elkérni a kamerát a futások során.

7. A program használata

A program indításakor a felhasználót a fő menü fogadja, ez a terveknek megfelelően jelenik meg, és a fő funkciókat innen lehet elérni.

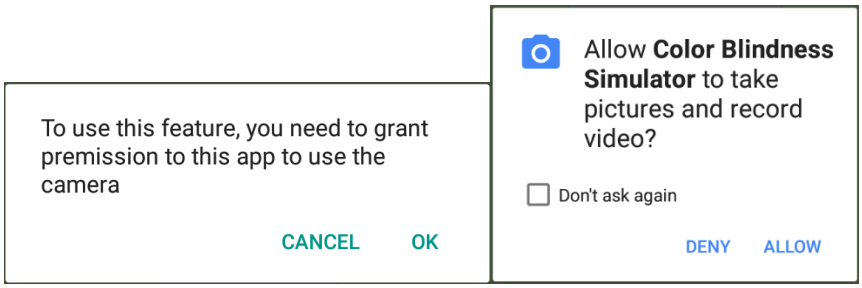


7.1 ábra, fő menü

7.1. Élőképes mód

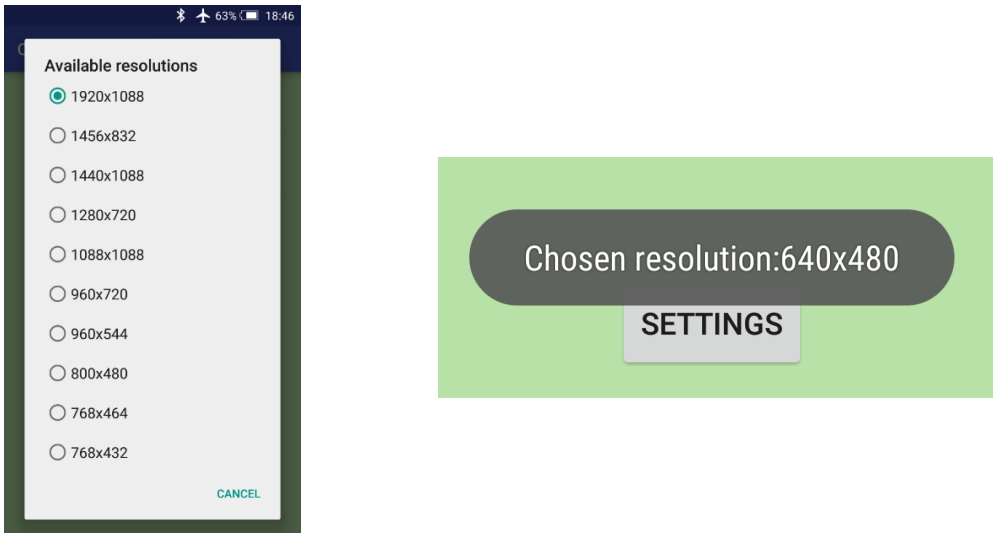
Az élőképes módok a fő menü első két menüpontjaihoz tartoznak. Felépítésük szempontjából pontosan ugyanaz a kettő, egyedül az alkalmazott szűrők terén különböznek.

Ha az Android 6-os, vagy magasabb verzióját használjuk, akkor valószínű, hogy a kamerához való hozzáférés alapból nem lesz megadva az alkalmazásnak, erről ilyenkor értesítés fog megjelenni (7.2 ábra)



7.2 ábra: engedélykérő üzenetek kamera használatra

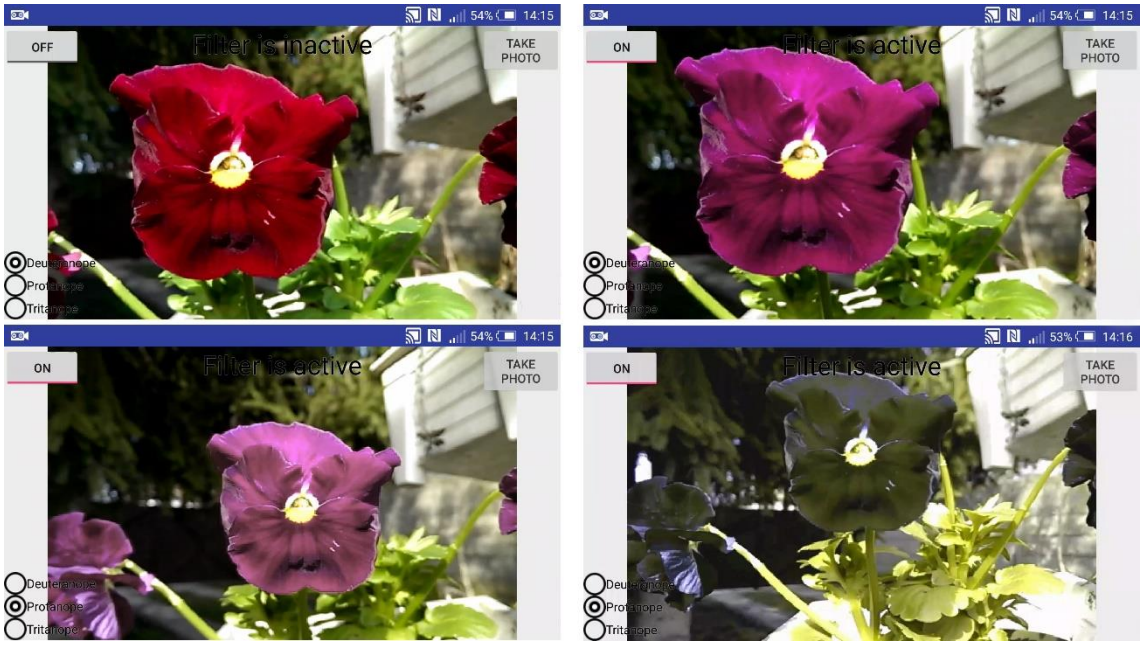
Indításuk előtt érdemes megnézni a beállításokat, ahol az élőkép felbontását lehet beállítani. A beállítások megnyomása után megjelenő felületen a „Choose a resolution” kiválasztásakor megjelenik egy rádiógombos lista a kamera által támogatott összes felbontással, amik közül egyet ki lehet választani. Kiválasztás után vissza kerülünk a fő menübe, és egy üzenet jelenik meg, hogy a felbontás ki lett választva (7.3 ábra).



7.3 ábra: beállításokban a támogatott kamera felbontások (egy része), és a kiválasztás után megjelenő üzenet.

Az élőképes módok egyikének kiválasztása után rögtön látható is lesz a kamera élőképe a beállításokban kiválasztott felbontásnak, és az ezáltal meghatározott aránynak megfelelően. A bal felső sarokban lévő gombbal lehet ki/bekapcsolni a szűrőt, ez alapból kikapcsolt állapotban van. Azt, hogy a szűrő be van-e kapcsolva, egyrészt a gomb állapota is jelzi, de a kép felső részén egy szöveg is jelzi ugyanezt. Bekapcsolt állapot esetén a bal alsó sarokban lévő rádiógombok segítségével lehet kiválasztani, hogy melyik szűrő legyen alkalmazva az élőképre. A jobb felső sarokban lévő gombbal lehet elmenteni az aktuálisan látott képet. Gombnyomás esetén minden esetben meg fog jelenni egy üzenet,

ami sikeres képmentés esetén kiírja, hogy hova tette a képet, sikertelen mentés esetén ennek tényéről tájékoztat, ha pedig nincs meg az engedély a fájl írásához, akkor a kamera engedélykérésénél látottakhoz hasonló jellegű engedélykérő üzenetet jelenít meg.



7.4 ábra: élőkép példák, bal felső eredeti, jobb felső deuteranóp daltonizálás, bal alsó protanóp daltonizálás, jobb alsó protanóp szimulálás

7.2. Állóképes mód

A fő menüben ez a harmadik menüpont. Itt lehetőség van különböző forrásból származó képeket daltonizálni és megosztani. A menüpont kiválasztásakor először egy demó oldal jelenik meg, ami két beépített képen mutatja be a szűrők alkalmazásának eredményét (7.5 ábra).

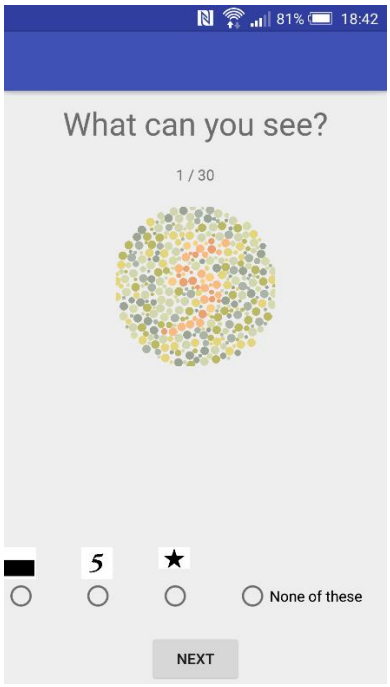
7.5 ábra: állóképes mód demó oldala

Itt a demó képek megtekintése mellett lehetőség van saját kép kiválasztására is. A bal felső gombbal már meglévő képet lehet kiválasztani, a jobb felsővel pedig új képet lehet készíteni. Mindkét esetben külső alkalmazás listát kapunk, amin az adott feladatot támogató már feltelepített alkalmazások fognak megjelenni. Itt lehet kiválasztani, melyikkel szeretnénk a feladatot elvégezni. Miután kiválasztottunk, vagy készítettünk egy képet, akkor a 7.6 ábra által mutatott felület fog megjelenni. A megjelenő két kép közül az alsó az eredeti kép, a felső pedig a daltonizált. A felső képet az alul található három állapotú gombbal lehet módosítani. Amit a gombon más lehetőséget választunk, annak hatása rögtön láthatóvá válik a felső képen. A felső kép megosztható, erre a kép jobb felső sarkában lévő gomb szolgál. Ha a kép felbontása túl nagy, akkor egy figyelmeztető üzenet jelenik meg erről, mert ilyenkor a megosztási folyamat tovább is eltarthat. Miután a folyamat befejeződött, egy lista jelenik meg azokkal a telepített alkalmazásokkal, amelyek képesek fogadni a képet.

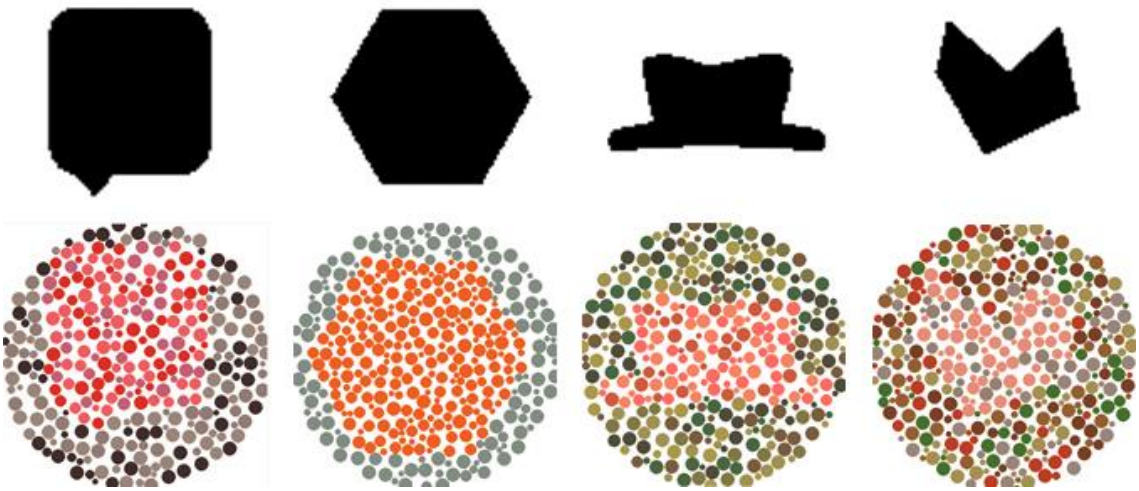
7.6 ábra: állóképes daltonizálás

7.3. Teszt

A teszt indítása után egy rövid magyarázat jelenik meg, mely leírja, hogy hogyan is kell kitölteni a tesztet. Itt van lehetőség az anonim eredményfeltöltés letiltására is, ami az indítógomb feletti szöveges jelölőnégyzetből a pipa eltávolításával lehetséges. Tovább lépés után a 7.7 ábra által mutatott megjelenés fogad. Középen jelenik meg a generált kép, amin minden esetben valamilyen alakzat szerepel, ezt kell felismerni. Lent a megfelelő kis kép alatt lévő rádiógombbal lehet rögzíteni a választ, az alattuk lévő gombbal pedig tovább lehet lépni. A generált kép felett lehet látni, hogy éppen hányadik kérdésnél járunk. A generált képek összesen tíz féle színkombinációban jelenhetnek meg, a sorrend, a színkombinációk sorrendje véletlenszerű. A színkombinációkról néhány példát mutat be a 7.8 ábra.

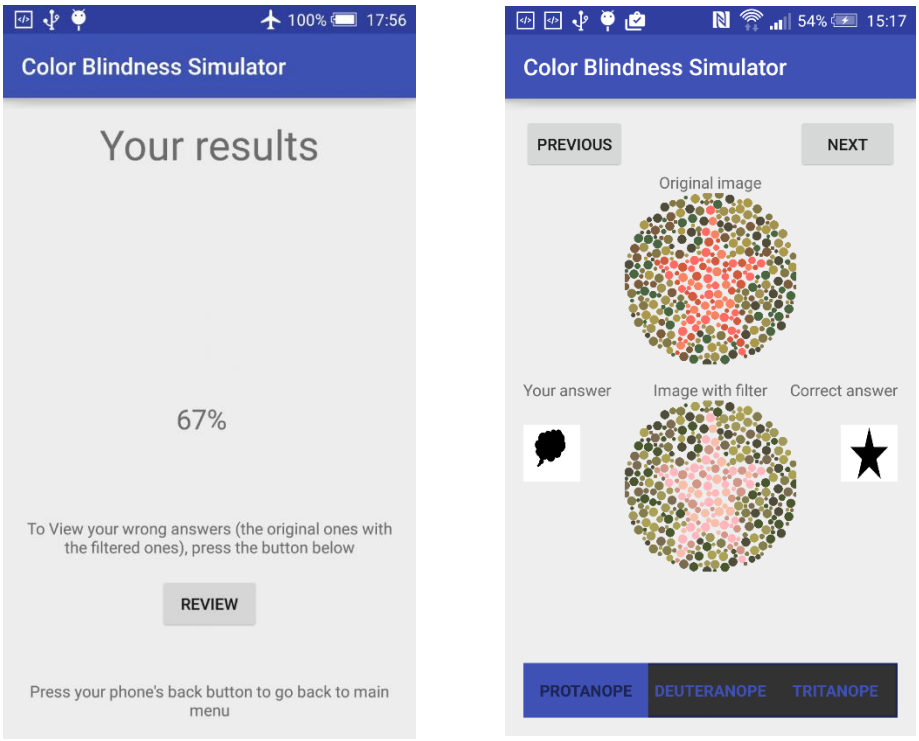


7.7 ábra: Ishihara teszt



7.8 ábra: forrás képek, és a belőlük létrejövő generált ábrák

A teszt kitöltése után megjelenik a teszt eredménye százalékos formában. Ha a felhasználó engedélyezte, akkor ennél a pontnál történik meg ennek a százalékos értéknek a feltöltése is. Egyetlen gomb található itt, amivel a hibákat lehet megtekinteni, amennyiben nem 100%-os lett az eredmény.



7.9 ábra: a teszt eredménye, és a hibák megtekintése

A hiba megtekintő oldalon megkapjuk az eredeti kérdésben szereplő ábrát fent, és alatta ugyanazt daltonizálva, amin az állóképes módhoz hasonlóan lehet állítani a lenti három állapotú gombbal. A bal oldali kis kép mutatja a mi hibás válaszunkat, ez itt üres, ha az „egyik se” választ adtuk. A jobb oldali kis kép tartalmazza a helyes választ. A fenti gombokkal lehet a hibás válaszok között lépkedni.

8. Teszteredmények

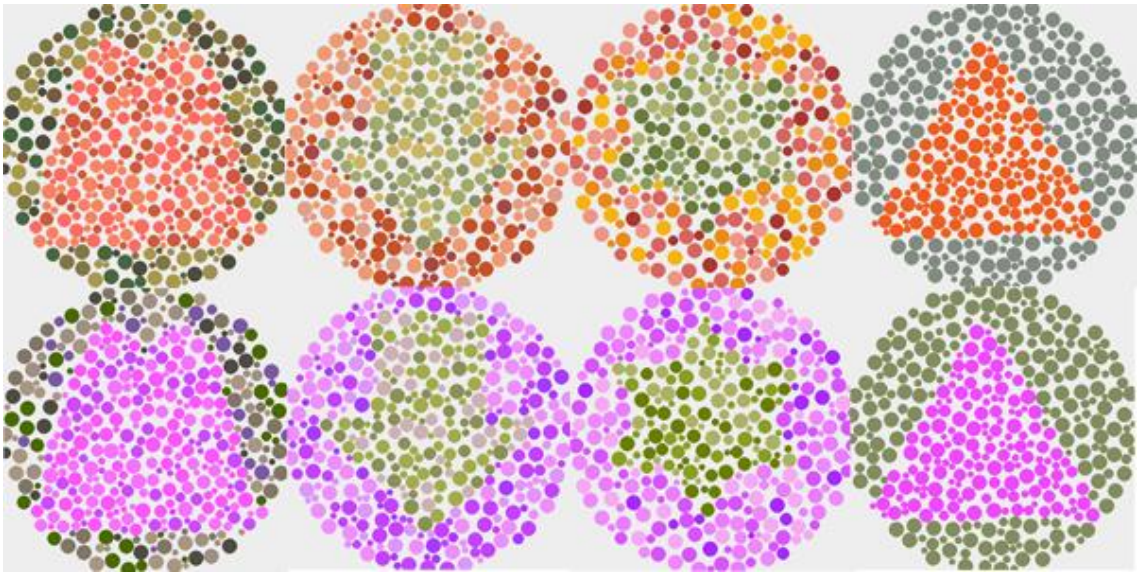
Lehetőségem nyílt több, mint 100 emberrel tesztelni az alkalmazást, több esetben személyesen, más esetekben a Firebase-be érkezett adatok alapján tudtam következtetni. A tesztelés során főleg azt vizsgáltam, hogy az alkalmazásban lévő teszt után kapott százalékos eredmény mennyire lehet összhangban az adott személy színlátásának helyességével.

Körülbelül 10-20 olyan személlyel is kitöltöttem a tesztet, aki egészséges színlátásúnak vallotta magát. Ők kivétel nélkül 90% feletti eredményt értek el, többen 100%-osra töltötték ki a tesztet. Akik hibáztak, azok főként elbizonytalanodtak, és úgy gondolták, hogy egyik ábrát sem látják a felsorolt lehetőségek között, így az „egyik se” lehetőséget választották. A feltételezett egészséges látásúak esetében csupán 1-2 esetben fordult elő olyan, hogy rosszul ismerték volna fel a látott ábrát, ez az 1-2 eset is akkor fordult elő, amikor a generált ábrán már elég sok szín jelent meg, ami kellően zavaró lehet az összhatásra való tekintettel, vagyis a legnehezebben felismerhető ábrák esetén (8.1 ábra).

8.1 ábra: nehezebb ábrák

5 olyan emberrel töltöttem ki a tesztet, aki enyhén színtévesztőnek tartotta magát. Esetükben nem lehetett egyértelműen kijelenteni, hogy adott típusú színtévesztés lenne jellemző rájuk, ugyanis három egyforma színösszeállítású ábrából volt, hogy kettőt helyesen felismertek, de a harmadiknál rossz ábrát láttak, másik három ábra esetén, ami nagyon hasonló színekből állt, mindháromra helyesen válaszoltak. Esetükben jellemző volt továbbá, hogy több ideig gondolkodtak a válaszon, alaposabban megnézték az ábrát és a válaszlehetőségeket is. Többnyire 50-80%-os eredményt értek el a teszten. Kíváncsi voltam arra, hogy a teszt után a hibák megtekintésénél segít-e valamit az ábra daltonizált változata, vagy sem. Itt személyenként és ábránként is eltérő volt a visszajelzés, volt aki nem látta jobban a daltonizált változatokon az ábrát, másoknak kifejezetten jobban látható volt néhány ábra. Azoknál az ábratípusoknál, ahol könnyebben elkülöníthető az ábra a

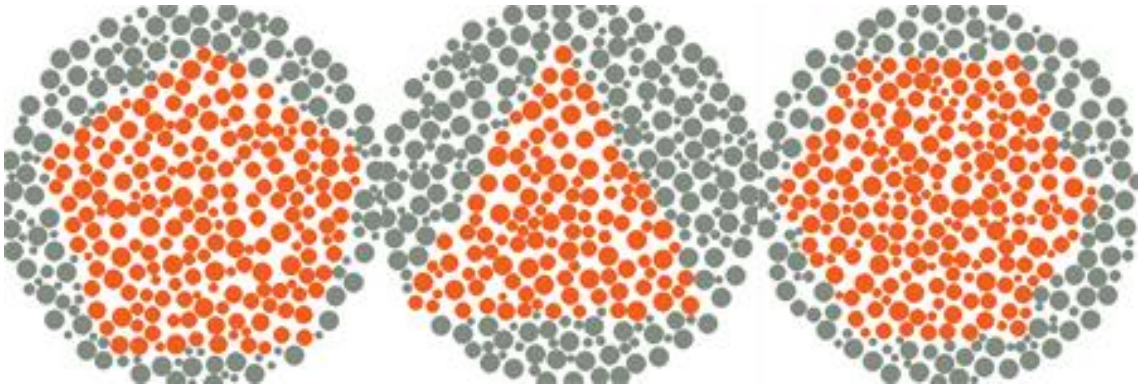
háttértől, azaz kevesebb eltérő színből állnak, ott többen hasznosabbnak ítélték az ábra daltonizált változatai közül a tritanóp daltonizálást (8.2 ábra).



8.2 ábra: generált képek (fent) és tritanóp daltonizált változataik (lent)

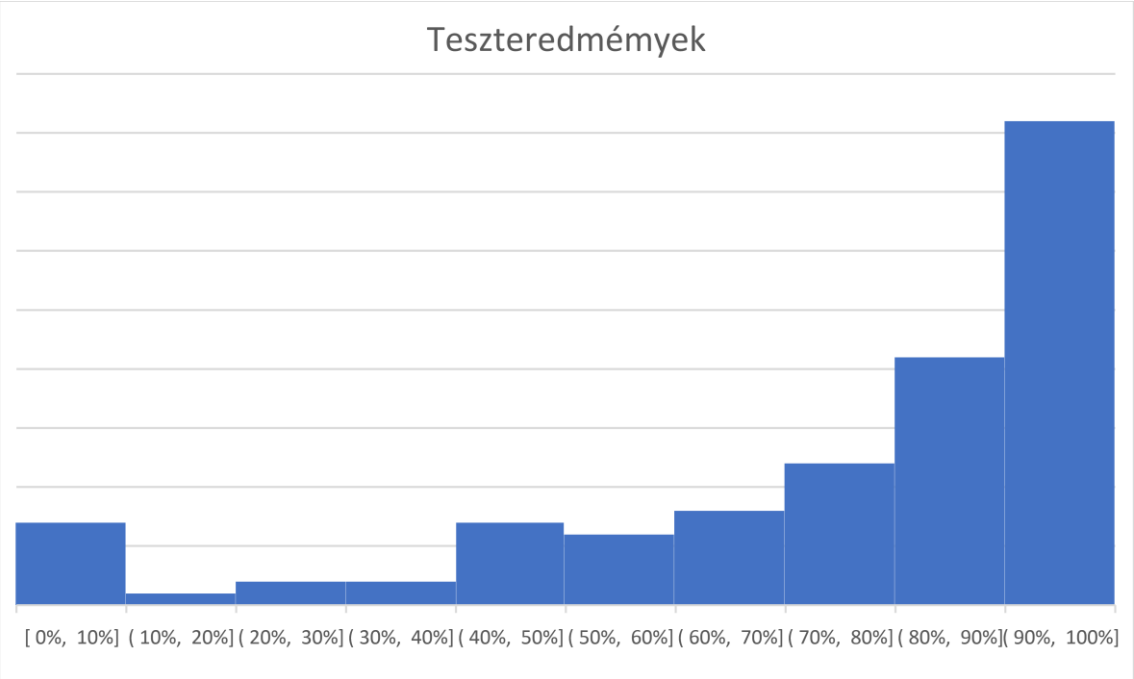
A tesztet elvégeztem egy cukorbeteg személlyel is. Az ő eredménye 67% lett. A könnyebb ábrákat szinte gondolkodás nélkül helyesen felismerte, gondot főleg a nehezebb ábrák okoztak, ott viszont egyértelműen nem ismert fel helyesen egy ábrát se. A daltonizált ábrák közül ő is a tritanóp daltonizált ábrák esetén érezte úgy, hogy azok jobban láthatóak a többi esetnél.

Egy ritkább színtévesztéssel rendelkező személlyel is el tudtam végezni a tesztet. Elmondása szerint gondot okozott számára a piros és fekete színek közötti különbség érzékelése. Ez a színtévesztés nem illeszthető be egyik nagyobb és gyakoribb színtévesztés típusba se, ezért is külön érdekes, hogy az Ishihara teszten akad-e olyan, amit felismer helyesen. A 30 generált ábra közül egyet se tudott felismerni, egyáltalán nem látott különbséget az alakzat színei és háttér színei között egyik esetben sem, továbbá a daltonizálás se segített semmit az ő esetében, egyáltalán nem érzékelte az alakzatot egyik daltonizált ábrán se, még a legkönnyebb (8.3 ábra) ábrák esetében se.



8.3 ábra: legkönnyebben felismerhető ábrák

A Firebase-be érkezett teszteredményeket a 8.4 ábra mutatja. Az ábra azt szemlélteti, hogy az adott százalékos tartományokba milyen sok teszt eredmény esett bele. Fontos, hogy az adatgyűjtés teljesen anonim, így csak annyit lehet látni, hogy milyen készülékről milyen eredmény mikor érkezett. A készülékekről se tárolok semmilyen egyedi azonosítót, csupán márkát és Android API számot. Emiatt az is előfordulhat, hogy két azonos készülékbejegyzés valójában kettő különböző eszközt takar. Így mivel nem azonosítom a felhasználókat és az eszközöket egyértelműen, az is előfordulhat, egy személy többször is kitölti a tesztet, és így több eltérő eredményt is elérhet, illetve olyan eset is előfordulhat, hogy valaki letölti az alkalmazást a saját eszközére, és több ismerősével is kitölteti a tesztet, így arról az eszközről több különböző személy eredménye is megjelenhet. Két eredmény bejegyzést így a bejegyzés időbéjege tud egyértelműen megkülönböztetni. Jelen esetben attól eltekintek hogy külön kezeljem azt az esetet, amikor pontosan ugyanolyan eszközről, ugyanolyan Android API szinttel, pontosan ezredmásodpercre megegyezően érkezik eredmény. Ha véletlenül ez megtörténne, akkor az egyik eredmény felülírná a másikat. Ezek miatt a feltételek miatt nem szabad azt feltételezni, hogy minden eredmény külön személyhez kötődik, ezt akkor se lehetne biztosra venni, ha lenne felhasználó azonosítás. Ezen megkötések ellenére jól kivehető, hogy sok olyan személy is letöltötte az alkalmazást, és kitöltötte a tesztet, akinek nincs sok gondja a színlátással csak érdeklődnek a téma iránt, de bőven akadtak olyanok is, akik ez alapján érintettek lehetnek, és így egy hozzávetőleges százalékos jellemzést kaphattak a színlátásukról. A rosszabb eredmények között több olyan is szerepelt, ahol nem válaszoltak az összes kérdésre, például válaszoltak az első 12 kérdésre, aztán abba hagyták a kitöltést valami miatt. Az ilyen tesztek természetesen lényegesen rosszabb százalékos eredményt érnek.



8.4 ábra: Firebase-be érkezett eredmények, 107 kitöltés (2017.04.22.-ei állapot)

Irodalomjegyzék

[1] John Dalton: Extraordinary facts relating to the vision of colours: with observations, Cadell and Davins, 1794

[2] Susan A. Randolph: Workplace Health & Safety, 2013, http://journals.sagepub.com/doi/pdf/10.1177/216507991306100608

[3] Taryn Biggs, Susan McPhail, Kurt Nassau, Hemant Patankar, Margaret Stenerson, Firman Maulana, Michael Douma, Sally E. Smith: Causes and Incidence of Colorblindness, http://www.webexhibits.org/causesofcolor/2C.html

[4] Colour Blind Awareness, http://www.colourblindawareness.org/colour-blindness/inherited-colour-vision-deficiency

[5] Michael L. Daley, PhD, Robert C. Watzke, MD, and Matthew C. Riddle, MD: Early Loss of Blue-Sensitive Color Vision in Patients With Type I Diabetes, DIABETES CARE, VOL. 10 NO. 6, NOVEMBER-DECEMBER 1987

[6] Afsaneh Khetrapal, BSc: What is Macular Degeneration? http://www.news-medical.net/health/What-is-Macular-Degeneration.aspx

[7] Daniel from Colblindor: Color Blind Essentials

[8] Dean Farnsworth: The Farnsworth-Munsell 100-Hue and Dichotomous Tests for Color Vision, Journal of the Optical Society of America, Vol. 33, Issue 10, pp. 568-578 (1943)

[9] Dr. Shinobu Ishihara: The Series of Plates Designed as a Test for Colour-Blindness, Kanehara Shuppan Co, 1972

[10] Hivalatos Android dokumentáció, https://developer.android.com/training/camera/photobasics.html

[11] Bob Dougherty, PhD, Alex Wade, PhD: Color blind image correction, http://www.vischeck.com/daltonize/

[12] Christos-Nikolaos Anagnostopoulos, George Tsekouras, Ioannis Anagnostopoulos Christos Kalloniatis: Intelligent modification for the daltonization process of digitized paintings, 5th International Conference on Computer Vision Systems http://biecoll.ub.uni-bielefeld.de/volltexte/2007/52/pdf/ICVS2007-6.pdf

Nyilatkozat

Alulírott Tömördi Péter, Gazdaságinformatikus szakos hallgató, kijelentem, hogy a dolgozatomat a Szegedi Tudományegyetem, Informatikai Intézet Képfeldolgozás és Számítógépes Grafika Tanszékén készítettem, Gazdaságinformatikus BSC diploma megszerzése érdekében.

Kijelentem, hogy a dolgozatot más szakon korábban nem védtem meg, saját munkám eredménye, és csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.

Tudomásul veszem, hogy szakdolgozatomat a Szegedi Tudományegyetem Informatikai Intézet könyvtárában, a helyben olvasható könyvek között helyezik el.

Dátum 2017.05.21

Aláírás