UNIVERZITET U BEOGRADU

ELEKTROTEHNIČKI FAKULTET

OBRADA SLIKA NA ANDROID OPERATIVNOM SISTEMU

Mentor:

dr Irini Reljin, prof.

Studenti:

Sofija Purić, dipl. ing.

Milan Mladenović, dipl. ing.

Beograd, jul 2014.

Sadržaj

Spisak slika

1. Uvod

Obrada slike predstavlja postupak promene sadržaja ili detalja neke slike po određenim definisanim i kontrolisanim koracima. Sam proces obrade slike je stariji od samih računara i počeo je prostim operacijama kao što su retuširanje, upotreba spreja sa bojom, ili upotrebom tradicionalnih umetničkih alata kao što su četkice, mastilo, grafit, boje, ulje itd. Pojavom računara pojavljuju se i računarski programi – alati čiji je zadatak da vrše digitalnu obradu slika. Postoje dva tipa alata za digitalnu obradu slika. Alati za rad sa rasterskom grafikom Paint, Photoshop, itd.. i alati sa vektorskom grafikom primer Corel draw itd.

Obrada slika na mobilnim uređajima krenula je sa razvojem kamera i uređaja u boji, a svoj vrhunac dostiže razvojem *smartphonova* tj. pametnih telefona, različitih proizvođača i nad različitim platformama. Android platforma jedna je od najzastupljenijih u svetu. Sam operativni sistem Android u zavisnosti od verzije može imati ugrađene neke od alata za manipulaciju i obradu slika, ali i ne mora. Starije verzije ovog operativnog sistema poseduju samo najprostije ili skoro nikakve alate za obradu. Aplikacija Obrada slika na Android operativnom sistemu napravljena je sa ciljem davanja funkcionalnosti ili proširenja mogućnosti obrade slika, i to

dodavanjem novih filtera i operacija kao i testiranja performansi Java virtualne mašine prilikom ovakvih obrada.

Rad se sastoji iz tri veće celine. Prvi deo predstavlja aplikativnu obradu slika na osnovu filtera za njihovu obradu. Predstavljene su tri osnovne grupe filtera za obradu slika i to: Unarni filteri, koji se primenjuju nad jednom slikom i vrše kvalitativnu izmenu slike na osnovu odgovarajućih unetih parametara za promenu, zatim binarni filteri ili operacije koji koriste dve učitane slike, pri čemu ih pre primene logički rasporede po tzv. *slojevima*, a zatim nad njima vrše kvalitativnu obradu pri kojoj nastaje treća slika kao rezultat primene funkcije nad ta dva sloja, a na kraju tu su konvolucioni filteri koji se, po principu sličnom matematičkoj konvoluciji, izvršavaju tako što nad svakim pikselom primene obradu koju definiše odgovarajuća konvoluciona matrica ili kernel. Ovi filteri su takođe unarnog tipa.

Drugi deo rada predstavlja implementaciju algoritma i prikaz histograma i to: prikaz histograma nad određenom slikom koja se učita ili dobije primenom nekih od filtera, prikaz histograma u relanom vremenu tj. nad pokrenutom aplikacijom za kameru u datom trenutku.

Treći deo rada predstavlja analizu performansi svakog od filtera, grafičku statistiku i procenu vremena izvršenja u zavisnosti od parametra potrebnog za izvršenje tog filtera. Podela je izvršena na tri dela i to: performanse unarnih filtera, performanse binarnih filtera kao i performanse konvolucionih filtera.

2. Postavka problema

Potrebno je implementirati algoritme za obradu slike nad Android operativnim sistemom, što podrazumeva izvršavanje složenosti od $O(n^2)$ do $O(n^4)$ u realnom vremenu, u zavisnosti od algoritma koji se bira.

S obzirom na kvalitet novih fotoaparata na mobilnim uređajima koji pružaju fotografije visoke rezolucije ali i na mogućnost učitavanja slika iz memorije isto sa visokim rezolucijama, postoji problem veličine učitanih slika, kao i njihovo složeno procesiranje, koje može dovesti do prekoračenja dozvoljene memorije za učitanu sliku. Ova greška poznata je pod nazivom *Memory overflow* odnosno prekoračenje dozvoljene memorije, pa je zato potrebno prethodno izvršiti optimalno skaliranje i prilagođavanje fotografije uslovima za dalju obradu.

Sama implementacija algoritama može se raditi na dva načina i to:

Korišćenjem Java programskog jezika, pri čemu bi se algoritmi obrade prvo prevodili na Java *bytecode*, koji predstavlja međukod između Java koda i binarnog zapisa na svakom od računara. Svaka Java virtualna mašina prilikom izvršavanja Java koda radi proceduru prevođenja na *bytecode* pa zatim taj *bytecode* prevodi na binarni. Razlog za uvođenje ovakvog načina

izvršavanja je nezavisnost programskog koda u odnosu na platformu na kome se on izvršava, odnosno unifikacija uređaja različitih proizvođača nad kojim se sam kod izvršava. Ovaj način implementacije je možda brži i lakši, pa čak i bliži običnom posmatraču, ali uz moguću pojavu problema vezanih za performanse izvršavanja zbog dvostrukog prevođenja koda, dakle iz Jave u *bytecode* pa u binarni kod.

Drugi način implementacije bio bi korišćenje *native* ključne reči nad potpisima metoda koji predstavljaju obradu slika. Na ovaj način sama implementacija filtera bila bi nad izvornim programskim jezicinma C ili C++ koji dozvoljavaju direktan pristup i rad sa registrima i memorijom i ne prevode se na Java bytecode već direktno na binarni kod platforme na kojoj se izvršavaju. A Java bi bila zadužena samo za pozivanje ovih metoda. Ovaj metod razvoja je hardverski efikasniji nego prethodni, ali je dosta teži za implementaciju jer koristi naprednije tehnike Java programiranja, odnosno integraciju Jave i programskih jezika C/C++ poznavanje svih navedenih jezika.

Osim prethodno navedenih problema postoje i problemi vezani za implementaciju nekih od filtera, primer *Gaussian blur* ili drugi konvolucioni filteri, čije performanse prilično zavise od učitanih parametara koji su potrebni za uspešno izvršenje algoritma. Tako se sa velikim povećanjem tih ulaznih faktora algoritam usporava jer se njegova složenost povećava pa performanse nisu više $O(n^4)$ već imaju neki faktor a koji predstavlja pozitivan realan broj veći od 1 i utiče na performansu pa složenost postaje $O(an^4)$.

Zbog gore navedenih problema sama Aplikacija obrade slika na Android operativnom sistemu, nameće nam, osim logičkih i implementacionih problema filter algoritama, novi zadatak, koji predstavlja ispitivanje performansi filtera u zavisnosti od hardverskog okruženja, ali i od *scale* parametara različitih opsega koji se koriste za pokretanje određenih filtera.

3. Karakteristike obrađenih filtera

Primena filtera nad slikom predstavlja proces gde se nad informacijom koju nosi piksel, dakle *RGB* vrednosti, primenjuje matematička funkcija da bi se proizvela odgovarajuća alternativna verzija slike. Dakle, filteri predstavljaju programske operacije koje nam omogućuju primenu različitih efekata nad slikama.

Slika je funkcija predstavljena formulom 3.1:

$$I : \{0,...,N\} x \{0,...,M\} \rightarrow \{0,...,255\}^3 \tag{3.1}$$

N i M su iz skupa prirodnih brojeva, pri čemu su N i M dimenzije slike u pikselima. Svaki piksel predstavljen je sa tri komponente: crvena (*R*), zelena (*G*) i plava (*B*), i to u opsegu od *0...255* za svaku komponentu. Neka je skup slika označen sa *I*. Filter predstavlja funkciju predstavljenu formulom 3.2:

$$f : I \rightarrow I \tag{3.2}$$

koja preslikava postojeću sliku u novu sliku uz promenu (*RGB*) vrednosti svakog piksela po odgovarajućem algoritmu. Ako je skup svih filtera označen sa *F*, onda simbol *f(I)* predstavlja primenu nekog od filtera iz skupa *F* nad nekom od slika iz skupa slika *I*.

U ovom poglavlju biće dat pregled svih obrađenih filtera nad izabranom slikom, koji su podeljeni u tri kategorije: unarni, binarni i konvolucioni filteri. Biće opisan način rada samog filtera, kao i način na koji je filter implementiran u aplikaciji uz prikaz delova koda. Osim toga biće prikazan i rad tih filtera uz slikovit prikaz delova aplikacije.

3.1. Primena unarnih filtera nad slikom

Pod pojmom unarnih filtera podrazumevamo one koji kao ulazni parametar zahtevaju samo jednu sliku nad kojom se primenjuje taj filter i kao rezultat vraćaju novu sliku nakon primene tog filtera. Pored toga unarni filteri mogu imati i argumente koji mogu predstavljati intenzitet primene ili jačinu neke od *RGB* komponenti koje su potrebne za primenu tog filtera nad slikom.

Filteri koji će biti opisani u nastavku rada su sledeći: invert, black&white, brightness, contrast, flip vertical, flip horizontal, grayscale, gamma correction, color filter, saturation, hue i shading filter.

3.1.1 Invert filter

Invert filter, poznat još pod imenom negativ, filter je koji menja originalnu vrednost piksela u njegovu inverznu vrednost, odnosno invertuje vrednost svakog piksela. Svaki piksel na slici čuva informacije o četiri kanala: alpha, red, green i blue. Međutim, alpha ne odražava vizuelno boju slike, tako da se inverzija primenjuje na ostala tri kanala, po formuli:

$$nova\_vrednost\ (R, G, B) = 0xFF - trenutna\_vrednost(R, G, B) \tag{3.3}$$

Najvažniji deo koda koji obavlja invertovanje piksela prikazan je u listingu 3.1:

Listing 3.1: Invert filter nad slikom

```
for (int x=0; x<width; x++) {
                  for (int y=0; y<height; y++) {
                        r=255-Color.red(colorArray[y*width+x]);
                        g=255-Color.green(colorArray[y*width+x]);
                        b=255-Color.blue(colorArray[y*width+x]);

                        colorArray[y*width+x]=Color.rgb(r,g,b);
                        returnBitmap.setPixel(x,y,colorArray[y*width+x]);
                  }
            }
```

Pri tome, width označava širinu slike, height visinu, a colorArray je niz celobrojnih vrednosti u kome pamtimo vrednosti piksela (ima width*height elemenata). Vrednosti komponenata piksela

(alpha, red, green, blue) su uskladištene u vrednost piksela na Android operativnom sistemu na sledeći način [1]:

$$pixel = (alpha << 24) \mid (red << 16) \mid (green << 8) \mid blue \tag{3.4}$$

Metode Color.red, Color.green i Color.blue vraćaju crvenu, zelenu i plavu komponentu odgovarajuće vrednosti piksela koja se prosleđuje kao argument metode, respektivno. Zbog poznatog načina skladištenja pojedinačnih komponenti piksela, te vrednosti smo mogli dobiti i na sledeći način:

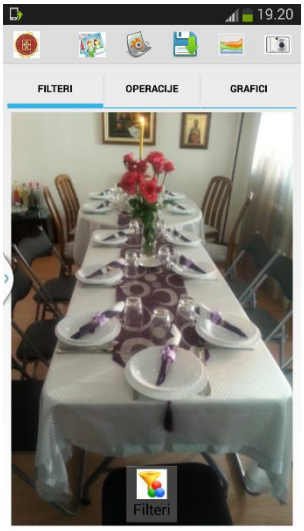
$$red = (color >> 16) \ \& \ 0xFF \tag{3.5}$$

$$green = (color >> 8) \ \& \ 0xFF \tag{3.6}$$

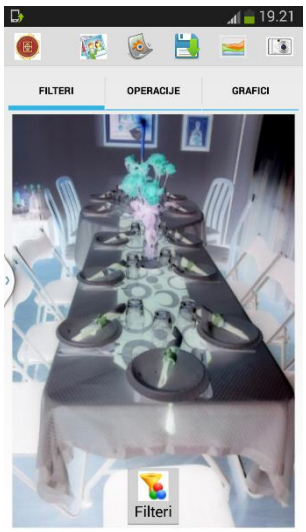
$$blue = color \ \& \ 0xFF \tag{3.7}$$

Metoda Color.rgb(r, g, b) vraća celobrojnu vrednost piksela na osnovu vrednosti pojedinačnih komponenti koje se prosleđuju kao argumenti poziva metode, a koje treba da budu u opsegu [0..255]. Vrednost alpha komponente je implicitno 255, što označava potpunu neprozračnost (neprovidnost). Metoda returnBitmap.setPixel(x, y, colorArray[y*width+x]) upisuje odgovarajuću vrednost piksela na odgovarajuće mesto u slici (na specificiranu x i y koordinatu piksela). Slika je na Android operativnom sistemu predstavljena preko objekta Bitmap.

Na slikama 3.1 i 3.2 prikazane su originalna slika i slika nakon primene invert filtera:



3.1 Originalna slika pre invert filtera



3.2 Slika nakon invert filtera

3.1.2 Black&White filter

Black and white predstavlja filter koji za postojeću sliku koja se zadaje, kao i *scale* parametar koji predstavlja intenzitet, vraća sliku čiji pikseli mogu imati samo crnu ili belu boju. tj. vrednosti komponenti *RGB* za sve piksele mogu biti ili *0,0,0* ili *255,255,255*. Filter prihvata sliku kao prvi argument, i parametar *scale* koji predstavlja realan broj od *0* do *100*. Filter zatim pravi lokalnu kopiju prosleđene slike, i računa prag na osnovu *scale* parametra po formuli:

$$t = (scale / 100) * 255 \tag{3.8}$$

Zatim se vrši obilazak matrice i za svaki piksel određuje se njegovo osvetljenje po formuli:

$$newValue = 0.2126 * x[i][j].red + 0.7152 * x[i][j].green + 0.0722 * x[i][j].blue \tag{3.9}$$

pri čemu je *x[i][j]* piksel na poziciji *i,j*, a anotacija *.red, .green, .blue,* označava da se uzima crvena (*R*), zelena (*G*), plava (*B*) komponenta boje tog piksela. Nakon toga se na osnovu osvetljenosti piksela i praga računa njegova nova vrednost i to ako je *newValue* > *t* uzima se 255, ako je *newValue* < *t* uzima se *0*, ako je *t* > *127.5* uzima se *255*, ako je *t* < *127.5* uzima se *0* i na kraju ako nije ni jedan od prethodnih uslova uzima se *255*. Zatim se nova vrednost postavlja kao parametar za sve tri komponente (*RGB*) piksela.

Na slikama 3.3 i 3.4 prikazane su originalna slika i slika nakon primene black&white filtera:

3.3 Originalna slika pre black&white filtera

3.4 Slika nakon primene black and white fitera sa faktorom 50

Najvažniji deo koda koji obavlja black&white filter prikazan je u listingu 3.2:

Listing 3.2: Black&white filter nad slikom

```
scale = (scale/100)*255;

for(int i = 0; i < width; i++) {

      for(int j = 0; j < height; j++) {
            r = Color.red(colorArray[j * width + i]);

            g = Color.green(colorArray[j * width + i]);

            b = Color.blue(colorArray[j * width + i]);

            newValue=0.2126*r+0.7152*g+0.0722*b;
            if (newValue>scale) newValue=255;
            else if (newValue<scale) newValue=0;
            else if (scale>127.5) newValue=255;
            else if (scale<127.5) newValue=0;

            else newValue=255;

            colorArray[j * width + i] = Color.rgb((int)newValue,
            (int)newValue, (int)newValue);

            returnBitmap.setPixel(i, j, colorArray[j * width + i]);
      }

}
```

3.1.3 Brightness

Brightness predstavlja filter koji postojeću zadatu sliku osvetljuje ili potamnjuje na osnovu zadatog *scale* parametra koji predstavlja intenzitet. Svaka od vrednosti komponenti *RGB* povećava se za navedeni *scale* faktor i dobija se nova vrednosti komponente. Bitno je napomenuti da se svaka komponenta povećava za isti *scale* faktor pa odnos *RGB* komponenti ostaje isti što dovodi do efekta zatamnjivanja ili osvetljivanja slike dok se same boje ne menjaju. Filter prihvata sliku kao prvi argument, i parametar *scale* koji predstavlja ceo broj od *-255* do *255*. Filter zatim pravi lokalnu kopiju prosleđene slike, a zatim se vrši obilazak matrice i za svaki piksel određuje njegova nova vrednost po formulama:

$$newValue = x[i][j].red + scale \tag{3.10}$$

$$newValue = x[i][j].green + scale \tag{3.11}$$

$$newValue = x[i][j].blue + scale \tag{3.12}$$

pri čemu je *x[i][j]* piksel na poziciji *i,j*, a anotacija *.red, .green, .blue,* označava da se uzima crvena (*R*), zelena (*G*), plava (*B*) komponenta boje tog piksela. Naravno u slučaju prekoračenja opsega *0..255* uzimaju se maksimalne vrednosti *0* i *255*. Dakle ako je *newValue > 255* uzima se da je *x[i][j].red = 255*, ako je *newValue < 0* uzima se da je *x[i][j].red = 0* u suprotnom uzima se *x[i][j].red = newValue*. Analogno se računaju i ostale dve komponente.

Na slikama 3.5, 3.6 i 3.7 prikazane su orginalna slika pre primene brightness filtera, kao i slike nakon primene brightness filtera sa scale faktorom 85 i -114, respektivno.

3.5 Originalna slika pre primene brightness filtera

3.6 Slika nakon primene brightness filtera sa scale faktorom 85

3.7 Slika nakon primene brightness filtera sa scale faktorom -114

Najvažniji deo koda koji obavlja brightness filter prikazan je u listingu 3.3:

Listing 3.3: Brightness filter nad slikom

```
if (scale <= 255 && scale >= -255) {
      for(int i=0; i < width; i++) {
      for(int j=0; j < height; j++) {
            r = Color.red(colorArray[j * width + i]);
            g = Color.green(colorArray[j * width + i]);
            b = Color.blue(colorArray[j * width + i]);

            newValue = r + scale;
            if (newValue > 255) r = 255;
            else if (newValue < 0) r = 0;
            else r = newValue;

            newValue = g + scale;
            if (newValue > 255) g = 255;
            else if (newValue < 0) g = 0;
            else g = newValue;

            newValue = b + scale;
            if (newValue > 255) b = 255;
            else if (newValue < 0) b = 0;
            else b = newValue;

      colorArray[j * width + i] = Color.rgb(r, g, b);
      returnBitmap.setPixel(i, j, colorArray[j * width + i]);
            } } }
```

3.1.4 Contrast

Reč kontrast, po široj definiciji predstavlja *suprotnost osobina*. Definisano je pet različitih vrsta kontrasta:

1. Na subjektu koji se fotografiše, to je odnos osvetljenja najtamnijeg i najsvetlijeg područja.
2. Kod rasvete, to je odnos najvećeg i najmanjeg intenziteta svetla koje dopire do različitih delova scene.
3. U slici, to je odnos svetla koje propuštaju ili reflektuju najprozirniji i najneprozirniji deo negativa ili pozitiva. Logaritam tog odnosa je gustina.
4. Kod foto materjala, kvalitativni pokazatelj mogućnosti stvaranja slike određene vrste kontrasta pri nekim uslovima.
5. Prilikom gledanja, razlika u izgledu pojedinih područja unutar vidnog polja.

Kontrast u fotografiji predstavlja razliku između tamnih i svetlih nijansi slike. Smanjivanjem kontrast faktora (*scale*) svetlije nijanse slike postaju tamne, dok tamnije nijanse slike postaju svetlije. Pojačavanje kontrast faktora radi obrnutu stvar, dakle tamnije nijanse postaju još tamnije, dok svetlije postaju još svetlije. Maksimalnim pojačanjem kontrast faktora slika postaje crno bela.

Filter prihvata sliku kao prvi argument i *scale* faktor koji predstavlja realan broj između -*100* i *100*. Filter zatim pravi lokalnu kopiju prosleđene slike, i računa novu vrednost *scale* faktora. Zatim se vrši obilazak matrice i za svaki piksel određuje njegova nova vrednost tako što se prvo za svaku komponentu *RGB* posebno oduzme *126*, pomnoži sa *scale* faktorom i onda doda 126. Novodobijena vrednost ukoliko pređe dozvoljene granice od *0* i *255* se postavlja na *0* ili *255* respektivno.

Na slikama 3.8 i 3.9 prikazane su originalna slika i slika nakon primene kontrast filtera:

3.8 Originalna slika pre kontrast filtera

3.9 Slika nakon primene kontrast filtera sa scale faktorom 50

Najvažniji deo koda koji obavlja kontrast filter prikazan je u listingu 3.4:

Listing 3.4: Contrast filter nad slikom

```
if (scale >=- 100 && scale <= 100) {
      scale = (scale+100)/100;
      scale = scale*scale;

      for(int i = 0; i < width; i++) {
      for(int j = 0; j < height; j++){

            r = Color.red(colorArray[j * width + i]);
            g = Color.green(colorArray[j * width + i]);
            b = Color.blue(colorArray[j * width + i]);
            a = Color.alpha(colorArray[j * width + i]);

            newValue = r;
            newValue -= 126;
            newValue *= scale;
            newValue += 126;
            if (newValue <= 0) r = 1;
            else if (newValue >= 255) r = 254;
            else r = newValue;

            newValue = g;
            newValue -= 126;
            newValue *= scale;
            newValue += 126;
            if (newValue <= 0) g = 1;
            else if (newValue >= 255) g = 254;
            else g = newValue;

            newValue = b;
            newValue -= 126;
            newValue *= scale;
            newValue += 126;
            if (newValue <= 0) b = 1;
            else if (newValue >= 255) b = 254;
            else b = newValue;

            colorArray[j * width + i] = Color.argb(a, r, g, b);
            returnBitmap.setPixel(i, j, colorArray[j * width + i]);

            }
      }

}
```

3.1.5 Flip vertical

Flip vertical predstavlja filter koji postojeću zadatu sliku rotira oko *x* ose za 180 stepeni. Tako dobijena slika je što se intenziteta komponenti *RGB* tiče nepromenjena, ali je okrenuta naopako. Filter prihvata sliku kao prvi i jedini argument. Filter zatim pravi lokalnu kopiju prosleđene slike, a zatim se vrši obilazak matrice i za svaki piksel određuje njegova nova vrednost tako što se zameni sa odgovarajućim pikselom iz njegove kolone, tj. svaki piksel *x[i][j]* menja se *x[i][height-j-1]* pri čemu je *height* visina slike. Najvažniji deo koda koji obavlja flip vertical filter prikazan je u listingu 3.5:

Listing 3.5: Flip vertical filter nad slikom

```
for (int i = 0; i < width; i++) {
for(int j=0; j < height/2; j++) {
      int rtemp = Color.red(colorArray[j * width + i]);
      int gtemp = Color.green(colorArray[j * width + i]);
      int btemp = Color.blue(colorArray[j * width + i]);

      r = Color.red(colorArray[(height-j-1) * width + i]);
      g = Color.green(colorArray[(height-j-1) * width + i]);
      b = Color.blue(colorArray[(height-j-1) * width + i]);

      colorArray[j * width + i] = Color.rgb(r, g, b);
      returnBitmap.setPixel(i, j, colorArray[j * width + i]);

      colorArray[(height-j-1) * width + i] = Color.rgb(rtemp, gtemp,
      btemp);
```

```
            returnBitmap.setPixel(i, (height-j-1), colorArray[(height-j-1)
            * width + i]);

            }
    }
```

Na slikama 3.10 i 3.11 prikazane su originalna slika i slika nakon primene flip vertical filtera.

3.10 Originalna slika pre flip vertical filtera

3.11 Slika nakon primene flip vertical filtera

3.1.6 Flip horizontal

Flip horizontal predstavlja filter koji postojeću zadatu sliku rotira oko *y* ose za 180 stepeni. Tako dobijena slika je što se intenziteta komponenti *RGB* tiče nepromenjena, ali je dobijen efekat ogledala. Filter prihvata sliku kao prvi i jedini argument. Filter zatim pravi lokalnu kopiju prosleđene slike, a zatim se vrši obilazak matrice i za svaki piksel određuje njegova nova vrednost tako što se zameni sa odgovarajućim pikselom iz njegove vrste, tj. svaki piksel *x[i][j]* menja se *x[width-i-1][j]* pri čemu je *width* širina slike. Najvažniji deo koda koji obavlja flip horizontal filter prikazan je u listingu 3.6:

Listing 3.6: Flip horizontal filter nad slikom

```
for (int i=0; i < width/2; i++) {
for (int j = 0; j < height; j++) {
      int rtemp = Color.red(colorArray[j * width + i]);
      int gtemp = Color.green(colorArray[j * width + i]);
      int btemp = Color.blue(colorArray[j * width + i]);

      r = Color.red(colorArray[j * width + (width-i-1)]);
      g = Color.green(colorArray[j * width + (width-i-1)]);
      b = Color.blue(colorArray[j * width + (width-i-1)]);

      colorArray[j * width + i] = Color.rgb(r, g, b);
      returnBitmap.setPixel(i, j, colorArray[j * width + i]);
```

```
        colorArray[j * width + (width-i-1)] = Color.rgb(rtemp, gtemp, btemp);
        returnBitmap.setPixel((width-i-1), j,
        colorArray[j * width + (width-i-1)]);

        }
}
```

Na slikama 3.12 i 3.13 prikazane su originalna slika i slika nakon primene flip horizontal filtera.

3.12 Originalna slika pre flip horizontal filtera

3.13 Slika nakon primene flip horizontal filtera

3.1.7 Grayscale

U fotografiji, pojam *grayscale* ili *grayscale* digitalna slika predstavlja sliku kod koje važi da za svaki piksel vrednosti komponente *RGB* budu međusobno jednake, dakle *R = G = B*. Zbog toga svaki piksel sa sobom nosi jedino informaciju o intenzitetu tih komponenti *RGB* koji mogu biti u opsegu od *0* do *255*. Na taj način se postiže efekat da se *grayscale* slike sastoje samo iz različitih nijansi sive boje, počevši od najtamnije nijanse tj. crne pa sve do najsvetlije tj. bele.

Grayscale slike treba razlikovati od *black and white* slika dobjenih primenom filtera *black&white* iz poglavlja 3.1.2 koje se sastoje samo iz dve boje, i to crne i bele bez prelaznih nijansi između njih. Dakle *grayscale* slike imaju različite nijanse sive između najsvetlije bele *(255,255,255)* i najtamnije crne *(0,0,0)*.

Filter za *grayscale* prihvata sliku kao prvi i jedini argument. Filter zatim pravi lokalnu kopiju prosleđene slike, a zatim se vrši obilazak matrice i za svaki piksel određuje njegova osvetljenost po formuli:

$$newValue = 0.2126 * x[i][j].red + 0.7152 * x[i][j].green + 0.0722 * x[i][j].blue \quad (3.13)$$

pri čemu je *x[i][j]* piksel na poziciji *i,j*, a anotacija *.red, .green, .blue,* označava da se uzima crvena (*R*), zelena (*G*), plava (*B*) komponenta boje tog piksela. Zatim se ta nova vrednost postavlja za svaku od komponenti *RGB* piksela i na taj način dobija efekat *grayscale*-a. Najvažniji deo koda koji obavlja grayscale filter prikazan je u listingu 3.7:

Listing 3.7: Grayscale filter nad slikom

```
for (int x = 0; x < width; x++) {
for (int y = 0; y < height; y++) {
      r = Color.red(colorArray[y * width + x]);
            g = Color.green(colorArray[y * width + x]);
                  b = Color.blue(colorArray[y * width + x]);

      newValue = 0.2126*r+0.7152*g+0.0722*b;
      colorArray[y * width + x] =
      Color.rgb((int)newValue, (int)newValue, (int)newValue);

      returnBitmap.setPixel(x, y, colorArray[y * width + x]);

      }
}
```

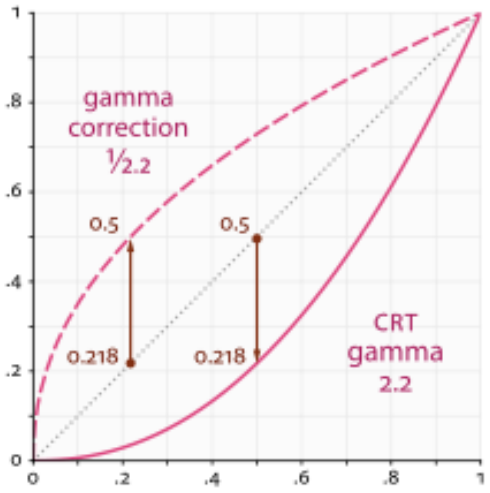
Na slikama 3.14 i 3.15 prikazane su originalna slika i slika nakon primene *grayscale* filtera.

3.14 Originalna slika pre primene grayscale filtera

3.15 Slika nakon primene grayscale filtera

3.1.8 Gama korekcija

Pojam gamma potiče još iz doba fotografije i opisuje nelinearnost prikaza slike na fotografskom papiru u odnosu na stvarnu sliku. Pojam korekcije boja kod monitora nasleđen je iz doba nastanka TV sistema. Osobina CRT (cathode ray tube) sistema je da odnos između upravljačkog signala katodne cevi (signala slike) i intenziteta svetla koji daje piksel nije linearan. Naime, neka 100% vrednosti signala na CRT daje 100% dobijenog intenziteta svetla i neka su to maksimalne normalizovane vrednosti u koordinatnom sistemu. U idealnom sistemu pad signala na 50% vrednosti ima kao rezultat pad intenziteta svetla na 50%, tj. odnos signala i intenziteta svetla je linearan. U praksi nije tako, naime pad signala od 50% daje intenzitet svetlosti koji je dosta manji od 50% u odnosu na maksimalnu vrednost. Na slici 3.16 prikazan je primer CRT gama korekcije – linija sa tačkicama predstavlja linearnu funkciju odnosa signala slike (Y osa) i intenziteta svetla (X osa), pri čemu gama faktor iznosi 1. Puna linija prikazuje tipično ponašanje CRT sistema, dok isprekidana linija predstavlja inverznu funkciju, tako da ukupni rezultat daje linearnu zavisnost i sliku prihvatljivu za oko [2].



3.16 Primer CRT gama korekcije

Gama korekcija je nelinearna operacija koja u svojoj suštini predstavlja posvetljavanje slike u zavisnosti od gama faktora koji predstavlja stepen posvetljavanja slike. Gama korekcija je definisana sledećim izrazom [3]:

$$Vout = A*Vin^{¥} \tag{3.14}$$

A je konstanta, *Vin* i *Vout* su ulazna i izlazna vrednost koje su nenegativni realni brojevi, a predstavljaju intenzitet vrednosti piksela pre i posle gama korekcije, dok ¥ predstavlja gama faktor koji ukoliko je veći od 1, ima efekat potamnjivanja senki na originalnoj slici, dok ukoliko je u opsegu [0..1] ima efekat posvetljavanja tamnih delova slike.

Prilikom implementacije algoritma za gama korekciju, gama faktor se unosi od strane korisnika, a može biti u opsegu [0..5], pri čemu ukoliko se odabere vrednost 1, originalna slika neće biti izmenjena. Formula na kojoj se bazira implementacija algoritma je sledeća:

$$Vout = Wmax * ((Vin - Wmin) / (Wmax - Wmin)) \wedge ¥ + Wmin \tag{3.15}$$

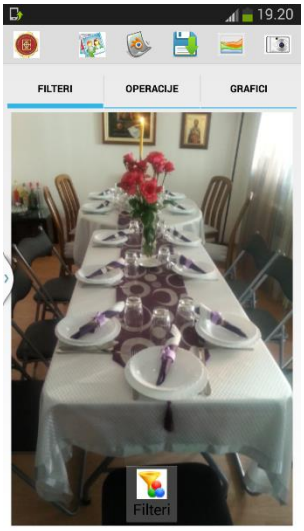
gde su *Vin* i *Vout* ulazna i izlazna vrednost piksela, ¥ gama faktor, a *Wmin* i *Wmax* minimalna i maksimalna vrednost intenziteta piksela na slici. Uzeto je da su te vrednosti 0 i 255, respektivno. Najvažniji deo koda koji obavlja gama korekciju nad pikselima prikazan je u listingu 3.8:

Listing 3.8: Gama korekcija nad slikom

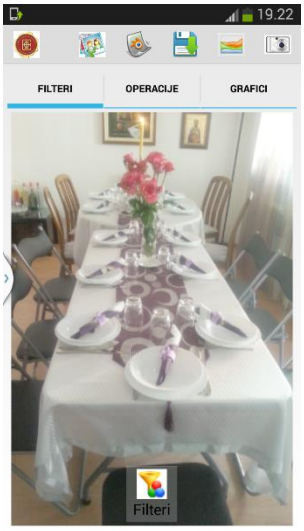
```
final int    MAX_SIZE = 256;
final double MAX_VALUE_DBL = 255.0;
final int    MAX_VALUE_INT = 255;
int[] gamma = new int[MAX_SIZE];
if (gammaFactor > 0) {
          for(int i = 0; i < MAX_SIZE; ++i)
              gamma[i] = (int)Math.min(MAX_VALUE_INT,
```

```
                (int)((MAX_VALUE_DBL * Math.pow(i / MAX_VALUE_DBL,
                 gammaFactor))));
            for(int x = 0; x < width; ++x) {
                for(int y = 0; y < height; ++y) {
                    pixel = src.getPixel(x, y);
                    A = Color.alpha(pixel);
                    R = gamma[Color.red(pixel)];
                    G = gamma[Color.green(pixel)];
                    B = gamma[Color.blue(pixel)];
                    bmOut.setPixel(x, y, Color.argb(A, R, G, B));
                }
            }
        }
return bmOut;
```

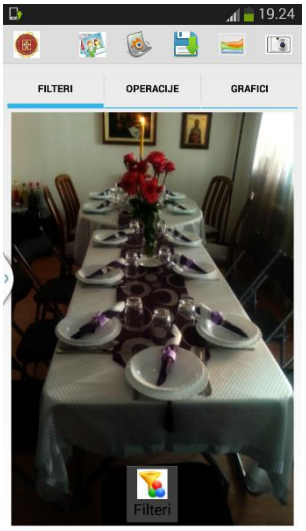
Metoda Color.argb(a, r, g, b) vraća celobrojnu vrednost piksela na osnovu vrednosti pojedinačnih komponenti (alfa komponente, crvene, zelene i plave komponente) koje se prosleđuju kao argumenti poziva metode, a koje treba da budu u opsegu [0..255]. Na slikama 3.17, 3.18, 3.19 i 3.20 predstavljene su originalna slika i slike nakon primene gama korekcije sa gama faktorom od 0.5, 2 i 4, respektivno.



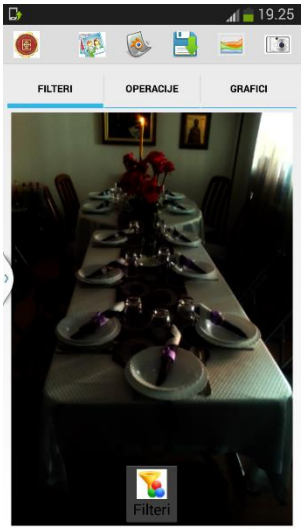
3.17 Originalna slika pre gama korekcije



3.18 Slika sa gama faktorom 0.5



3.19 Slika sa gama faktorom 2



3.20 Slika sa gama faktorom 4

3.1.9 Color filter

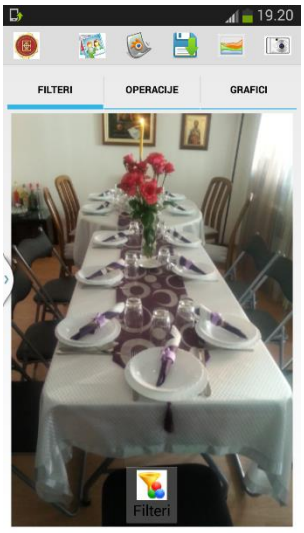
Color filter nam omogućava filtriranje boja na slici, odnosno prikazivanje slike preko samo jednog kanala boje (crvenog, zelenog ili plavog) ili kombinacijom sva tri kanala boje. Parametri sa kojima se množe pojedinačne komponente piksela (crvena, zelena, plava) unose se od strane korisnika i u opsegu su [0..1]. Najvažniji deo koda koji obavlja filtriranje boja na slici prikazan je u listingu 3.9:

Listing 3.9: Color filter nad slikom

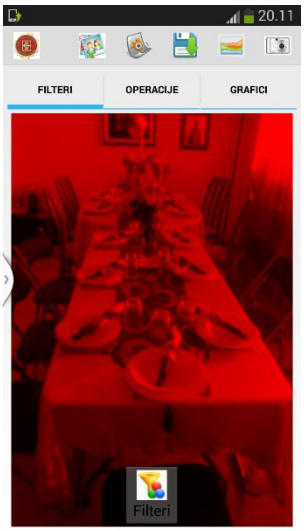
```
for(int x = 0; x < width; ++x) {
            for(int y = 0; y < height; ++y) {
                pixel = src.getPixel(x, y);
                A = Color.alpha(pixel);
                R = (int)(Color.red(pixel) * red);
                G = (int)(Color.green(pixel) * green);
                B = (int)(Color.blue(pixel) * blue);
                bmOut.setPixel(x, y, Color.argb(A, R, G, B));
            }
        }
```

Prikazivanje originalne slike prikazane na slici 3.21 samo preko crvene komponente omogućeno je odabirom RGB parametara koji su (1, 0, 0) kao što je prikazano na slici 3.22; samo preko zelene komponente omogućeno je odabirom RGB parametara koji su (0, 1, 0) kao što je prikazano na slici 3.23; samo preko plave komponente omogućeno je odabirom RGB

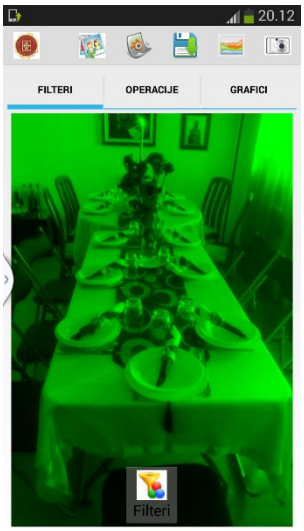
parametara koji su (0, 0, 1) kao što je prikazano na slici 3.24; preko svih triju komponenti koje su ravnomerno rasporedjene omogućeno je odabirom RGB parametara koji su (0.5, 0.5, 0.5) kao što je prikazano na slici 3.25.



3.21 Originalna slika pre color filtera

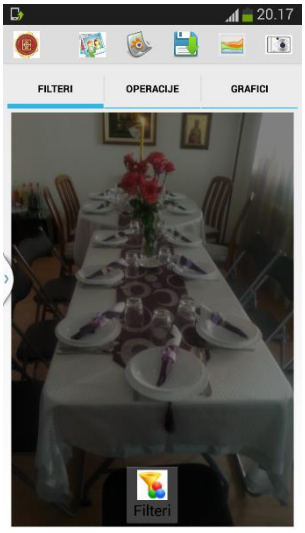


3.22 Slika sa primenom R,G,B = (1, 0, 0)



3.23 Slika sa primenom R,G,B = (0, 1, 0)

3.24 Slika sa primenom R,G,B = (0, 0, 1)



3.25 Slika sa primenom R,G,B = (0.5, 0.5, 0.5)

3.1.10 Shading filter

Shading filter je tehnika koja koristi bitski I (AND) operator izmedju originalne boje piksela i odabrane boje kojom se vrši senčenje slike. Preko dijaloga za odabir boje za senčenje (slika 3.26) korisnik odabira željenu boju kojom će se vršiti senčenje slike. Najvažniji deo koda koji obavlja senčenje slike prikazan je u listingu 3.10:

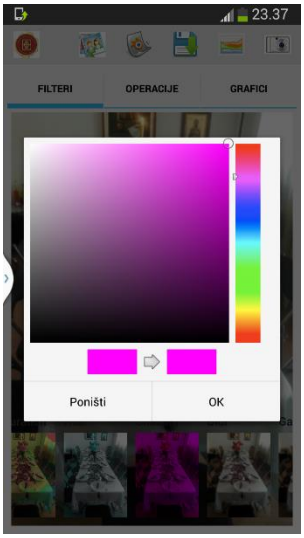
Listing 3.10: Shading filter nad slikom

```
source.getPixels(pixels, 0, width, 0, 0, width, height);
int index = 0;
for(int y = 0; y < height; ++y) {
      for(int x = 0; x < width; ++x) {
                  index = y * width + x;
                  pixels[index] &= shadingColor;
               }
          }
Bitmap bmOut = Bitmap.createBitmap(width, height, Bitmap.Config.ARGB_8888);
bmOut.setPixels(pixels, 0, width, 0, 0, width, height);
return bmOut;
```

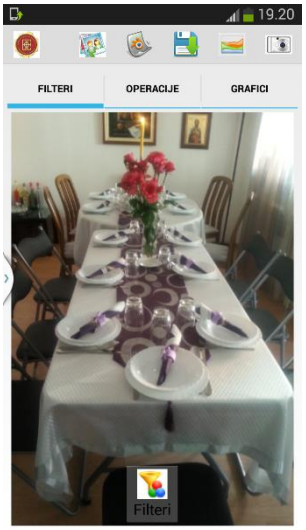
Pri tome, width predstavlja širinu slike, height dužinu tj. visinu, a pixels je niz width*height celih brojeva. Metoda getPixels koja se poziva nad objektom Bitmap-e u nizu pixels vraća kopiju podataka iz bitmap-e. Za svaki piksel računamo poziciju u dvodimenzionalnoj matrici (index) i nad tom vrednošću piksela radimo bitsku operaciju AND sa celobrojnom vrednošću boje kojom senčimo sliku. Format slika ARGB_8888 označava da se svaki piksel slike skladišti pomoću 4B pri čemu svaka od komponenti piksela zauzima po 8

bitova (R, G i B komponenta i alpha komponenta). Metoda setPixels vrši zamenu svih piksela u bitmap-i bmOut vrednostima piksela uskladištenim u nizu pixels. Na taj način je postignuto da napravimo novu sliku koja će se sastojati od izmenjenih vrednosti piksela originalne slike.

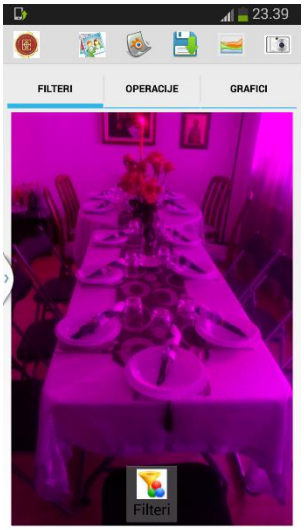
Originalna slika prikazana je na slici 3.27, a senčenje originalne slike izabranom bojom prikazano je na slici 3.28.



3.26 Color Picker dijalog



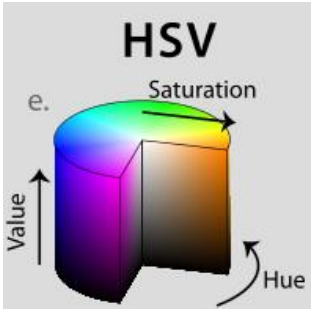
3.27 Originalna slika pre shading filtera



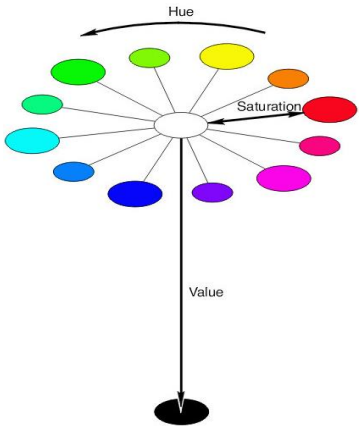
3.28 Slika sa shading filterom

3.1.11 Saturation filter

HSV je jedan od dva najčešća cilindrična modela predstavljanja boja u RGB sistemu. Ovaj model preuredjuje geometriju RGB modela boja kako bi bila intuitivnija od reprezentacije boja u Dekartovom koordinatnom sistemu i to čini mapiranjem vrednosti boja u cilindar, kao što je prikazano na slici 3.29. Ugao oko centralne vertikalne ose predstavlja dimenziju Hue, a udaljenost od ose predstavlja dimenziju Saturation. Visina cilindra odgovara trećoj dimenziji u HSV modelu, a to je Value. Dimenzija Saturation se odnosi na dominaciju nijanse u nekoj boji. [4] Na spoljnoj ivici „točka“ boja prikazanom na slici 3.30 nalaze se tzv. čiste nijanse boja. Kada se pomeramo ka centru „točka“ nijanse koje koristimo za opisivanje boja su sve manje i manje dominantne. Te boje direktno na centralnoj osi se smatraju nezasićenim (za šta se još koristi termin desaturacija) – one predstavljaju sive tonove, krećući se od bele ka crnoj, sa svim nijansama sive izmedju. Saturation (zasićenje) je dakle dimenzija koja se kreće od spoljne ivice „točka“ boja (kada je boja u potpunosti zasićena) ka centru „točka“ (kada je boja potpuno nezasićena). [5]



3.29 HSV cilindrični model



3.30 HSV model sa tri dimenzije

Dimenzija Hue je ono što većina zove bojom, a saturation označava koliko je boja (hue) čista, odnosno jasna. Boja koja uopšte nema zasićenje (saturation) biće siva. Boja koja ima maksimalno zasićenje (saturation) je u svojoj najintenzivnijoj formi.

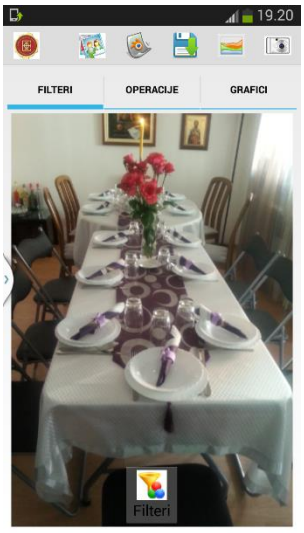
Algoritam za primenu saturation filter-a implementiran je tako da korisnik unosi željenu vrednost za dimenziju zasićenja, odnosno saturation, koja mora biti u opsegu [0..1], nakon toga se boja svakog pojedinačnog piksela na slici konvertuje u HSV komponente, promeni se vrednost saturation dimenzije u onu koju je korisnik odabrao i na kraju se takve promenjene HSV komponente konvertuju nazad u ARGB model boje pojedinačnog piksela. Najvažniji deo koda koji obavlja primenu saturation filtera prikazan je u listingu 3.11:

Listing 3.11: Saturation filter nad slikom

```
float[] HSV = new float[3];
      for(int y = 0; y < height; ++y) {
              for(int x = 0; x < width; ++x) {
                  index = y * width + x;
                  Color.colorToHSV(pixels[index], HSV);
                  HSV[1] = level;
                  HSV[1] = (float) Math.max(0.0, Math.min(HSV[1], 1.0));
                  pixels[index] = Color.HSVToColor(HSV);
              }
          }
```

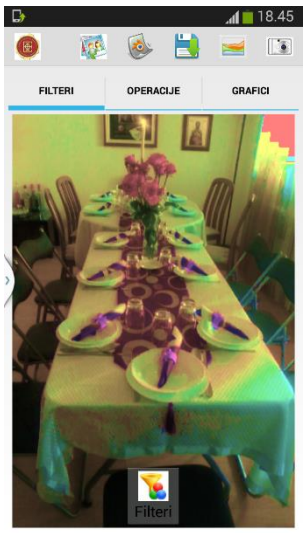
U nizu realnih brojeva HSV čuvamo vrednosti hue, saturation i value komponenti pojedinačnog piksela. Metoda Color.colorToHSV vrši konverziju boje iz ARGB modela boja u njene HSV komponente, pri čemu je HSV[0] hue komponenta u opsegu [0..360), HSV[1] je saturation komponenta u opsegu [0..1], a HSV[2] je value komponenta u opsegu [0..1]. Metoda Color.HSVToColor vrši konverziju HSV komponenti u ARGB model boje, a argument koji se prosledjuje ovoj metodi je niz HSV koji sadrži izmenjene vrednosti HSV komponenti.

Originalna slika prikazana je na slici 3.31, dok su slike nakon primene saturation filtera sa vrednostima 0, 0.5 i 1 prikazane na slikama 3.32, 3.33 i 3.34, respektivno.

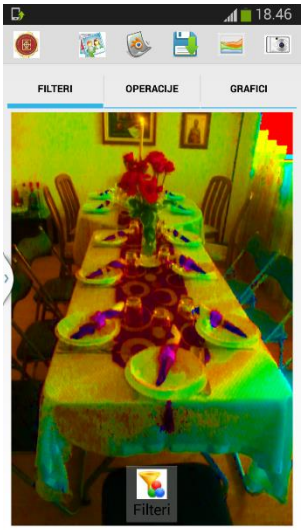


3.31 Originalna slika pre saturation filtera

3.32 Slika sa primenom saturation filtera = 0



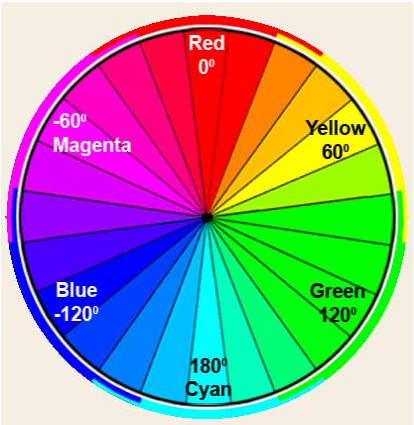
3.33 Slika sa primenom saturation filtera = 0.5



3.34 Slika sa primenom saturation filtera = 1

3.1.12 Hue filter

Dimenzija Hue predstavlja jednu od glavnih osobina boje koja je opisana kao „stepen do kojeg se stimulus može opisati kao sličan ili različit od stimulusa koji predstavlja crvenu, zelenu, plavu i žutu boju“ (koje su osnovne boje). Hue je, dakle, ono što mi podrazumevamo kada opisujemo boje – crvena, ljubičasta, plava itd, a konkretnije, hue predstavlja dominantnu talasnu dužinu boje. Kao što je već rečeno, HSV je jedan od cilindričnih modela za predstavljanje boja kod koga je Hue ugaona dimenzija, sa početkom u osnovnoj crvenoj boji na 0° koja prolazi kroz zelenu osnovnu boju na 120°, zatim plavu osnovnu boju na 240° (ili -60°) i na kraju se vraća nazad ka crvenoj osnovnoj boji na 360° (ili 0°). Hue dimenzija se slikovito može prikazati preko tzv. „točka“ boja, kao što se može videti na slici 3.35. [5]



3.35 Točak boja

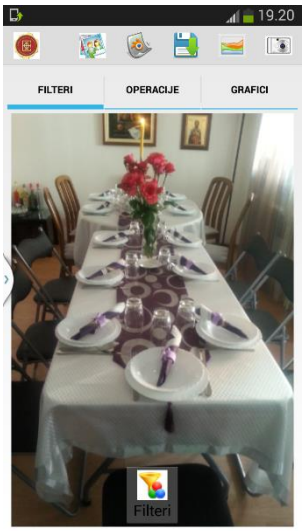
Kada menjamo dimenziju Hue, mi vršimo ponovno mapiranje postojećih opsega boja u nove opsege boja. Algoritam za primenu hue filter-a implementiran je tako da korisnik unosi željenu vrednost za dimenziju hue, koja mora biti u opsegu [0..360), nakon toga se boja svakog pojedinačnog piksela na slici konvertuje u HSV komponente, promeni se vrednost hue dimenzije u onu koju je korisnik odabrao i na kraju se takve promenjene HSV komponente konvertuju nazad u ARGB model boje pojedinačnog piksela. Najvažniji deo koda koji obavlja primenu hue filtera prikazan je u listingu 3.12:

Listing 3.12: Hue filter nad slikom

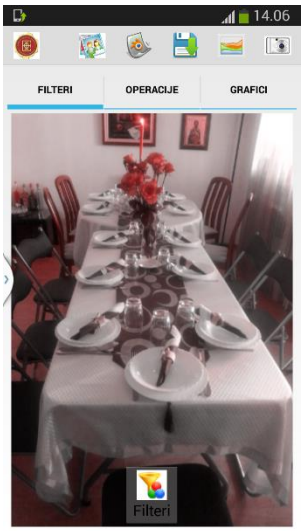
```
float[] HSV = new float[3];
      for(int y = 0; y < height; ++y) {
            for(int x = 0; x < width; ++x) {
                  index = y * width + x;
                  Color.colorToHSV(pixels[index], HSV);
                  HSV[0] = level;
                  HSV[0] = (float) Math.max(0.0, Math.min(HSV[0], 360.0));
                  pixels[index] = Color.HSVToColor(HSV);
              }
          }
```

U nizu realnih brojeva HSV čuvamo vrednosti hue, saturation i value komponenti pojedinačnog piksela. Metoda Color.colorToHSV vrši konverziju boje iz ARGB modela boja u njene HSV komponente, pri čemu je HSV[0] hue komponenta u opsegu [0..360), HSV[1] je saturation komponenta u opsegu [0..1], a HSV[2] je value komponenta u opsegu [0..1]. Metoda Color.HSVToColor vrši konverziju HSV komponenti u ARGB model boje, a argument koji se prosledjuje ovoj metodi je niz HSV koji sadrži izmenjene vrednosti HSV komponenti.

Originalna slika prikazana je na slici 3.36, dok su slike nakon primene hue filtera sa vrednostima 0, 120 i 240 prikazane na slikama 3.37, 3.38 i 3.39, respektivno.

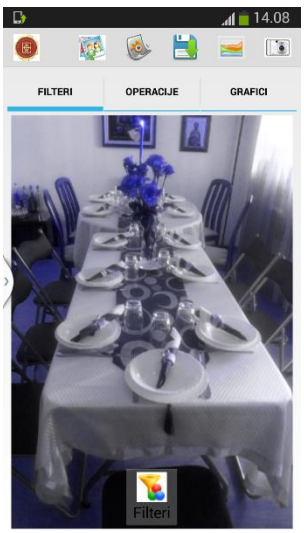


3.36 Originalna slika pre hue filtera



3.37 Slika sa primenom hue filtera = 0

3.38 Slika sa primenom hue filtera = 120



3.39 Slika sa primenom hue filtera = 240

3.2. Primena binarnih filtera nad slikama

Binarni filteri ili *blend* modovi u procesiranju slika predstavljaju operaciju koja prikazuje kako se dva sloja koji predstavljaju različite slike spajaju tj. stapaju u jedan. Klasičan *blend* mod ili predstavlja varjantu stapanja dve slike kod kojih gornji sloj ima zadatak da prosto prekrije sloj ispod. Međutim, s obzirom da su pikseli sami predstavljeni u *RGB* i imaju više komponenti nad kojima se može vršiti procesiranje, postoji više binarnih filtera, tj. više *blend* modova koji predstavljaju na koji način se dve slike mogu stopiti.

Pored normalnog blend moda, postoje još i dissolove koji uzima random piksel iz neke od slika i to u slučaju da je visoka providnost, većina piksela se uzima sa gornjeg sloja, u suprotnom većina piksela uzima se sa donjeg sloja. Zatim tu su još i *multiply, difference(subtract), lighter, darker*, kao i cela grupacija *dodge* i *burn* filtera koji menjaju osvetljenje slike na osnovu istoimene fotografske tehnike koja manipuliše ekspozicijom označenih oblasti na fotografskom papiru. U narednom delu poglavlja obrađeni su *normal blend filter, multiply, difference, lighter, darker*.

3.2.1 Normal blend mode

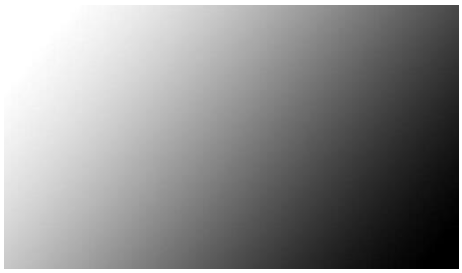
Normal blend mode je standardan blend mode, koji ima zadatak da gornjim slojem prekrije donji bez mešanja boja, i to povećavajući *alpha composition* faktor koji predstavlja stepen transparentnosti gornjeg sloja.

Da bi slika mogla da menja svoju transparentnost na osnovu *alpha* faktora, *Catmull* i *Smith* su uveli još jednu komponenta kao deo *RGB* standarda. Komponenta *A* tj *integralna alpha vrednost* predstavlja stepen providnosti *RGB* za dati piksel, i to u slučaju da je vrednost komponente *A 0%* piksel je potpuno transparentan, a u slučaju vrednosti *100%* piksel je neprovidan i jednak je pikselu iz običnog *RGB* kanala.

Filter za *normal blend mode* prihvata dve slike, gornji i donji sloj kao prvi i drugi argument, kao i *alpha* faktor između 0 i 1 koji predstavlja stepen stapanja slika. Filter zatim pravi lokalnu kopiju gornjeg sloja tj. prve argumentom prosleđene slike, a zatim se u slučaju ispravno unetog *alpha* faktora vrši obilazak matrice i za svaku poziciju piksela na gornjem sloju (gornjoj slici), se vrši promena i na donjem sloju (donjoj slici), i to tako što se svaka od komponenti *RGB* gornje slike množi *(1-alfa)* faktorom kako bi se povećala transparentnost.

Na slikama 3.40, 3.41 i 3.42 prikazane su originalna prva i druga slika (slika na prvom i drugom sloju), kao i rezultujuća slika koja se dobija primenom normal blend mode-a sa alpha faktorom od 0.67.

3.40 Slika na prvom sloju pre operacije blend



3.41 Slika na drugom sloju pre operacije blend

3.42 Slika nakon primene blenda između slojeva sa faktorom 0.67

Najbitniji deo koda koji obavlja operaciju normal blend mode prikazan je u listingu 3.13:

Listing 3.13: Normal blend mode nad slikama

```
if(alpha > 0 || alpha < 1) {

      for (int x = 0; x < width; x++)
      for (int y = 0; y < height; y++) {

            int red, green, blue;

            if (bitmap2.getHeight() > y && bitmap2.getWidth() > x) {

                  blue = (int)(Color.blue(bitmap1.getPixel(x, y))*(1-alpha)
                  + Color.blue(bitmap2.getPixel(x, y))*alpha);

                  green = (int)(Color.green(bitmap1.getPixel(x, y))*(1-
                  alpha) + Color.green(bitmap2.getPixel(x, y))*alpha);

                  red = (int)(Color.red(bitmap1.getPixel(x, y))*(1-alpha) +
                  Color.red(bitmap2.getPixel(x, y))*alpha);

            } else {

                  blue = (int)(Color.blue(bitmap1.getPixel(x, y)));
                  green = (int)(Color.green(bitmap1.getPixel(x, y)));
                  red = (int)(Color.red(bitmap1.getPixel(x, y)));

            }

            returnBitmap.setPixel(x, y, Color.rgb(red, green, blue));

      }
} else {
      return bitmap1;
}
```

3.2.2 Multiply blend mode

Multiply blend mode je blend mode između dva sloja, tj. dve slike u kome se za svaki piksel sa gornjeg sloja sve tri komponente *RGB* množe sa odgovarajućim pikselom donjeg sloja. Rezultat je tamnija slika. Formula za multiply blend mode je:

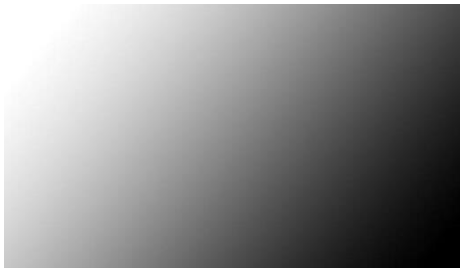
$$f(a, b) = ab \tag{3.16}$$

Gde je *a* osnovni tj. gornji sloj, a *b* predstavlja donji sloj. Filter je simetričan tako da zamenom mesta slojeva rezultat ostaje nepromenjen. U slučaju iste slike na oba sloja, ovaj mod je ekvivalentan primenom filtera gama korekcije sa faktorom $\gamma=2$. U slučaju da je jedan od slojeva homogena boja npr. siva, filter je ekvivalentan normal blend flteru sa donjim slojem koji je homogeno crne boje.

Filter za *multiply blend mode* prihvata dve slike, gornji i donji sloj kao prvi i drugi argument. Filter zatim pravi lokalnu kopiju gornjeg sloja tj. prve argumentom prosleđene slike, a zatim vrši obilazak matrice i vrši procesiranje, i to tako što se svaka od komponenti *RGB* gornje slike množi sa *RGB* komponentama odgovarajućeg piksela na donjoj slici.

Na slikama 3.43, 3.44 i 3.45 prikazane su originalna prva i druga slika (slika na prvom i drugom sloju), kao i rezultujuća slika koja se dobija primenom multiply blend mode-a.

3.43 Slika na prvom sloju pre operacije multiply



3.44 Slika na drugom sloju pre operacije multiply

3.45 Slika nakon primene operacije multiply

Najbitniji deo koda koji obavlja operaciju multiply blend mode prikazan je u listingu 3.14:

Listing 3.14: Multiply blend mode nad slikama

```
for (int x = 0; x < width; x++)
for (int y = 0; y < height; y++) {

      int red, green, blue;

      if (bitmap2.getHeight() > y && bitmap2.getWidth() > x) {

            blue = Color.blue(bitmap1.getPixel(x, y)) &
            Color.blue(bitmap2.getPixel(x, y));

            green = Color.green(bitmap1.getPixel(x, y)) &
            Color.green(bitmap2.getPixel(x, y));

            red = Color.red(bitmap1.getPixel(x, y)) &
            Color.red(bitmap2.getPixel(x, y));

      } else {

            blue = (int)(Color.blue(bitmap1.getPixel(x, y)));
            green = (int)(Color.green(bitmap1.getPixel(x, y)));
            red = (int)(Color.red(bitmap1.getPixel(x, y)));

      }

      returnBitmap.setPixel(x, y, Color.rgb(red, green, blue));

}

return returnBitmap;
```

3.2.3 Difference blend mode

Difference blend mode je blend mode između dva sloja, tj. dve slike u kome se za svaki piksel sa donjeg sloja sve tri komponente *RGB* oduzimaju od odgovarajućih komponenti *RGB* piksela gornjeg sloja. Rezultat je tamnija slika. Prilikom oduzimanja uzima se apsolutna vrednost razlike. Formula za subtract blend mode je:

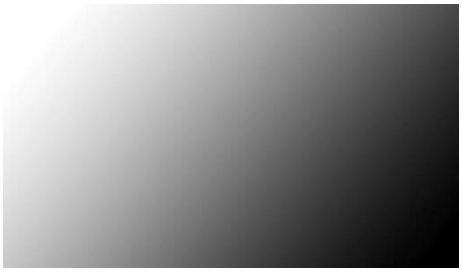
$$f(a, b) = |a - b| \tag{3.17}$$

gde je *a* osnovni tj. gornji sloj, a *b* predstavlja donji sloj.

Filter za *difference blend mode* prihvata dve slike, gornji i donji sloj kao prvi i drugi argument. Filter zatim pravi lokalnu kopiju gornjeg sloja tj. prve argumentom prosleđene slike, a zatim vrši obilazak matrice i vrši procesiranje, i to tako što se svaka od komponenti *RGB* donje slike oduzima od *RGB* komponente odgovarajućeg piksela na gornjoj slici. Prilikom oduzimanja koristi se funkcija *Math.abs* za računanje apsolutne vrednosti.

Na slikama 3.46, 3.47 i 3.48 prikazane su originalna prva i druga slika (slika na prvom i drugom sloju), kao i rezultujuća slika koja se dobija primenom difference blend mode-a.

3.46 Slika na prvom sloju pre operacije difference



3.47 Slika na drugom sloju pre operacije difference

3.48 Slika nakon primene operacije difference

Najbitniji deo koda koji obavlja operaciju difference blend mode prikazan je u listingu 3.15:

Listing 3.15: Difference blend mode nad slikama

```
for (int x = 0; x < width; x++)
for (int y = 0; y < height; y++) {

      int red, green, blue;

      if (bitmap2.getHeight() > y && bitmap2.getWidth() > x) {

            blue = (int)Math.abs((Color.blue(bitmap1.getPixel(x, y)) -
            Color.blue(bitmap2.getPixel(x, y))));

            green = (int)Math.abs((Color.green(bitmap1.getPixel(x, y)) -
            Color.green(bitmap2.getPixel(x, y))));

            red = (int)Math.abs((Color.red(bitmap1.getPixel(x, y)) -
            Color.red(bitmap2.getPixel(x, y))));

      } else {

            blue = (int)(Color.blue(bitmap1.getPixel(x, y)));
            green = (int)(Color.green(bitmap1.getPixel(x, y)));
            red = (int)(Color.red(bitmap1.getPixel(x, y)));

      }

      returnBitmap.setPixel(x, y, Color.rgb(red, green, blue));

}

return returnBitmap;
```

3.2.4 Lighter blend mode

Lighter blend mode je blend mode između dva sloja, tj. dve slike u kome se za svaki piksel sa gornjeg sloja *luminance* vrednost istog poredi sa odgovarajućom vrednošću *luminance* vrednosti piksela donjeg sloja. Uzima se uvek onaj piksel kod kojeg je *luminance* vrednost veća. Formula za *luminence* vrednost je:

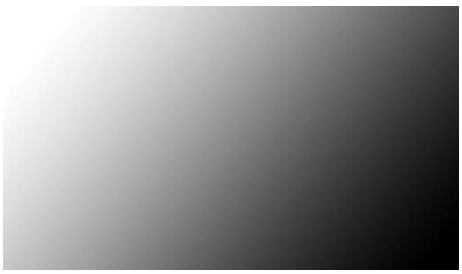
$$(0.2126 * Color.red(p)) + (0.7152 * Color.green(p)) + (0.0722 * Color.blue(p)) \quad (3.18)$$

gde je *p* piksel na nekoj poziciji *(x,y)*.

Filter za *lighter blend mode* prihvata dve slike, gornji i donji sloj kao prvi i drugi argument. Filter zatim pravi lokalnu kopiju gornjeg sloja tj. prve argumentom prosleđene slike, a zatim vrši obilazak matrice i vrši procesiranje, i to tako što se računa *luminance* za oba piksela na istoj poziciji gornjeg i donjeg sloja i uzima onaj sa većom vrednošću.

Na slikama 3.49, 3.50 i 3.51 prikazane su originalna prva i druga slika (slika na prvom i drugom sloju), kao i rezultujuća slika koja se dobija primenom lighter blend mode-a.

3.49 Slika na prvom sloju pre operacije lighter



3.50 Slika na drugom sloju pre operacije lighter

3.51 Slika nakon primene operacije lighter

Najbitniji deo koda koji obavlja operaciju lighter blend mode prikazan je u listingu 3.16:

Listing 3.16: Lighter blend mode nad slikama

```
private static double luminance(int pixel) {

      return (0.2126 * Color.red(pixel)) + (0.7152 * Color.green(pixel)) +
      (0.0722 * Color.blue(pixel));

}

for (int x = 0; x < width; x++)
for (int y = 0; y < height; y++) {

      int red, green, blue;

      if (bitmap2.getHeight() > y && bitmap2.getWidth() > x) {

            if (luminance(bitmap1.getPixel(x, y)) >=
            luminance(bitmap2.getPixel(x, y))) {

                  blue = Color.blue(bitmap1.getPixel(x, y));
                  green = Color.green(bitmap1.getPixel(x, y));
                  red = Color.red(bitmap1.getPixel(x, y));

            } else {

                  blue = Color.blue(bitmap2.getPixel(x, y));
                  green = Color.green(bitmap2.getPixel(x, y));
                  red = Color.red(bitmap2.getPixel(x, y));

            }

      } else {

            blue = Color.blue(bitmap1.getPixel(x, y));
            green = Color.green(bitmap1.getPixel(x, y));
            red = Color.red(bitmap1.getPixel(x, y));

      }

      returnBitmap.setPixel(x, y, Color.rgb(red, green, blue));

}
```

3.2.5 Darker blend mode

Darker blend mode je blend mode između dva sloja, tj. dve slike suprotan od lighter blend mode filtera u kome se za svaki piksel sa gornjeg sloja *luminance* vrednost istog poredi sa odgovarajućom vrednošću *luminance* vrednosti piksela donjeg sloja. Uzima se uvek onaj piksel kod kojeg je *luminance* vrednost manja. Formula za *luminence* vrednost je:

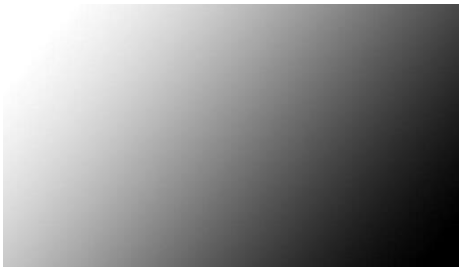
$$(0.2126 * Color.red(p)) + (0.7152 * Color.green(p)) + (0.0722 * Color.blue(p)) \quad (3.19)$$

gde je *p* piksel na nekoj poziciji *(x,y)*.

Filter za *darker blend mode* prihvata dve slike, gornji i donji sloj kao prvi i drugi argument. Filter zatim pravi lokalnu kopiju gornjeg sloja tj. prve argumentom prosleđene slike, a zatim vrši obilazak matrice i vrši procesiranje, i to tako što se računa *luminance* za oba piksela na istoj poziciji gornjeg i donjeg sloja i uzima onaj sa manjom vrednošću. Nakon uzastopne kombinovane primene lighter i darker filtera dobija se originalna slika.

Na slikama 3.52, 3.53, 3.54 i 3.55 prikazane su originalna prva i druga slika (slika na prvom i drugom sloju), rezultujuća slika koja se dobija primenom darker blend mode-a i slika koja se dobija nakon kombinovane primene lighter i darker blend mode-a, respektivno.

3.52 Slika na prvom sloju pre operacije darker



3.53 Slika na drugom sloju pre operacije darker

3.54 Slika nakon primene operacije darker

3.55 Slika nakon primene kombinacije operacija lighter pa darker

Najbitniji deo koda koji obavlja operaciju darker blend mode prikazan je u listingu 3.17:

Listing 3.17: Darker blend mode nad slikama

```
private static double luminance(int pixel) {

      return (0.2126 * Color.red(pixel)) + (0.7152 * Color.green(pixel)) +
      (0.0722 * Color.blue(pixel));

}

for (int x = 0; x < width; x++)
for (int y = 0; y < height; y++) {

      int red, green, blue;

      if (bitmap2.getHeight() > y && bitmap2.getWidth() > x) {

            if (luminance(bitmap1.getPixel(x, y)) <=
            luminance(bitmap2.getPixel(x, y))) {

                  blue = Color.blue(bitmap1.getPixel(x, y));
                  green = Color.green(bitmap1.getPixel(x, y));
                  red = Color.red(bitmap1.getPixel(x, y));

            } else {

                  blue = Color.blue(bitmap2.getPixel(x, y));
                  green = Color.green(bitmap2.getPixel(x, y));
                  red = Color.red(bitmap2.getPixel(x, y));

            }

      } else {

            blue = Color.blue(bitmap1.getPixel(x, y));
            green = Color.green(bitmap1.getPixel(x, y));
            red = Color.red(bitmap1.getPixel(x, y));

      }

      returnBitmap.setPixel(x, y, Color.rgb(red, green, blue));

}
```

3.3 Primena konvolucionih filtera nad slikom

Konvolucioni filteri su naziv dobili na osnovu matematičke konvolucije, koja predstavlja primenu operacije nad dve funkcije *f* i *g* čime se proizvodi nova fukcija. [6]

$$\phi(x) = \int_{-\infty}^{+\infty} f_1(x-y)\, f_2(y) dy \tag{3.20}$$

Kod procesiranja slika, termin kernel, konvoluciona matrica ili maska predstavlja matricu koja kada se primenom konvolucije nad slikom za odgovarajuće vrednosti matrice koju definiše filter, dobijaju različiti efekti. Sam algoritam primene konvolucionih filtera isti je za sve filtere ove vrste, jedina razlika je izbor kernela tj. njegove dimenzije i vrednosti kojima je on popunjen. U samom projektu prikazana je implementacija najčešće korišćenih filtera i to *Blur, Gaussian blur, Sharpen, Edge, Emboss, Engraving i Smoth.*

Implementacija navedenih konvolucionih matrica rađena je korišćenjem konvolucione matrice *3x3*. Sama konvoluciona matrica predstavlja matricu kod koje su dimenzije visine i širine iste i pri čemu moraju biti prirodan neparan broj veći od *1*. Primena konvolucione matrice dimenzije 1 ne bi imala nikakvog efekta nad slikom. Osim toga valja napomenuti da se povećanjem veličine konvolucione matrice dobija veći efekat filtera, ali su performanse sporije jer su sami algoritmi složenosti $O(n^4)$, što bi značilo logički 4 ugnježdena *for* ciklusa, pa bi se

povećanjem broja obrtaja najugnježdjenija dva ciklusa koja predstavljaju obilazak konvolucione matrice, znatno oborile performanse algoritama. Zbog toga svi navedeni filteri koriste matrice veličine *3x3* osim filtera *Gaussian blur* čiji se kernel računa po posebnoj formuli koja će biti naknadno definisana.

Pseudokod primene konvolucionih filtera izgleda ovako:

```
for y = 0 to ImageHeight do
   for x = 0 to ImageWidth do
      sum = 0
         for i= -h to h do
            for j = -w to w do
               sum = sum + k(j,i) * f(x - j, y - i)
            end for
         end for
      g(x y) = sum
   end for
end for
```

3.3.1 Blur

Blur je konvolucioni filter kojim se dobija efekat zamagljivanja slike, i jedan je od najkorišćenijih filtera pri obradi slike. [7] Ovaj filter dobija se primenom odgovarajuće konvolucione matrice 3x3 nad *RGB* komponentama svakog od piksela slike. U implementaciji blura korišćena je sledeća konvoluciona matrica.

$$\begin{matrix} \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \\ \frac{1}{9} & \frac{1}{9} & \frac{1}{9} \end{matrix}$$

Filter za *blur* prihvata sliku kao jedini argument, zatim pravi lokalnu kopiju slike, a onda se vrši obilazak matrice pri čemu se inicijalizuju za svaki piksel buduće vrednosti komponenti *RGB* na 0. Zatim se za svaki piksel vrši prolazak kroz konvolucionu matricu pri čemu je početni indeks (*width/2)*(-1)* a krajnji domet predstavlja (*width/2*) pri čemu *width* je neparan prirodan broj veći od *1* i predstavlja dimenziju konvolucione matrice. U toku prolaska kroz konvolucionu matricu za svaki piksel vrši se procesiranje nove vrednosti komponenti piksela i zatim se ta vrednost setuje kao nova vrednost piksela na toj poziciji.

Na slikama 3.56 i 3.57 prikazane su originalna slika i slika nakon 3 uzastopne primene konvolucionog filtera blur.

3.56 Originalna slika pre blur filtera

3.57 Slika nakon 3 uzastopna blur filtera

3.3.2 Sharpen

Sharpen je konvolucioni filter kojim se dobija efekat izoštravanja slike. [8] Ovaj filter dobija se primenom odgovarajuće konvolucione matrice nad *RGB* komponentama svakog od piksela slike. U implementaciji *sharpen* filtera korišćena je sledeća konvoluciona matrica.

$$\begin{matrix} 0 & -1 & 0 \\ -1 & 5 & -1 \\ 0 & -1 & 0 \end{matrix}$$

Filter za *sharpen* prihvata sliku kao jedini argument, zatim pravi lokalnu kopiju slike, a onda se vrši obilazak matrice pri čemu se inicijalizuju za svaki piksel buduće vrednosti komponenti *RGB* na 0. Zatim se za svaki piksel vrši prolazak kroz konvolucionu matricu pri čemu je početni indeks (*width/2)*(-1)* a krajnji domet predstavlja (*width/2*) pri čemu *width* je neparan prirodan broj veći od *1* i predstavlja dimenziju konvolucione matrice. U toku prolaska kroz konvolucionu matricu za svaki piksel vrši se procesiranje nove vrednosti komponenti piksela i zatim se ta vrednost setuje kao nova vrednost piksela na toj poziciji.

Na slikama 3.58 i 3.59 prikazane su originalna slika i slika nakon primene konvolucionog filtera sharpen.

3.58 Originalna slika pre sharpen filtera

3.59 Slika nakon primene sharpen filtera

3.3.3 *Edge*

Edge je konvolucioni filter kojim se detektuju ivice objekata neke slike. Ovaj filter dobija se primenom odgovarajuće konvolucione matrice nad *RGB* komponentama svakog od piksela slike. U implementaciji *edge* filtera korišćena je sledeća konvoluciona matrica.

$$\begin{matrix} -1 & -1 & -1 \\ -1 & 8 & -1 \\ -1 & -1 & -1 \end{matrix}$$

Filter za *edge* prihvata sliku kao jedini argument, zatim pravi lokalnu kopiju slike, a onda se vrši obilazak matrice pri čemu se inicijalizuju za svaki piksel buduće vrednosti komponenti *RGB* na 0. Zatim se za svaki piksel vrši prolazak kroz konvolucionu matricu pri čemu je početni indeks (*width/2)*(-1)* a krajnji domet predstavlja (*width/2*) pri čemu *width* je neparan prirodan broj veći od *1* i predstavlja dimenziju konvolucione matrice. U toku prolaska kroz konvolucionu matricu za svaki piksel vrši se procesiranje nove vrednosti komponenti piksela i zatim se ta vrednost setuje kao nova vrednost piksela na toj poziciji.

Na slikama 3.60 i 3.61 prikazane su originalna slika i slika nakon primene konvolucionog filtera edge.

3.60 Originalna slika pre edge filtera

3.61 Slika nakon primene edge filtera

3.3.4 Engrave

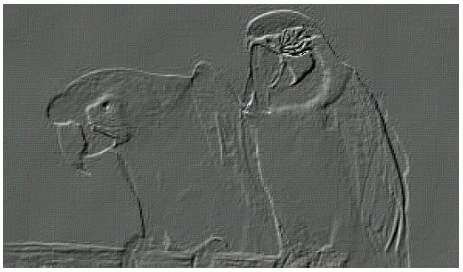
Engrave je konvolucioni filter kojim se dobija efekat gravura na slici. Ovaj filter dobija se primenom odgovarajuće konvolucione matrice nad *RGB* komponentama svakog od piksela slike. U implementaciji *engrave* filtera korišćena je sledeća konvoluciona matrica.

$$\begin{matrix} -2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 1 & 0 \end{matrix}$$

Filter za *engrave* prihvata sliku kao jedini argument, zatim pravi lokalnu kopiju slike, a onda se vrši obilazak matrice pri čemu se inicijalizuju za svaki piksel buduće vrednosti komponenti *RGB* na 0. Zatim se za svaki piksel vrši prolazak kroz konvolucionu matricu pri čemu je početni indeks (*width/2)*(-1)* a krajnji domet predstavlja (*width/2*) pri čemu *width* je neparan prirodan broj veći od *1* i predstavlja dimenziju konvolucione matrice. U toku prolaska kroz konvolucionu matricu za svaki piksel vrši se procesiranje nove vrednosti komponenti piksela i zatim se ta vrednost setuje kao nova vrednost piksela na toj poziciji.

Na slikama 3.62 i 3.63 prikazane su originalna slika i slika nakon primene konvolucionog filtera engrave.

3.62 Originalna slika pre engrave filtera



3.63 Slika nakon primene engrave filtera

3.3.5 *Emboss*

Emboss je konvolucioni filter kojim se dobija efekat urezbarene, reljefne slike. Svaki piksel slike zamenjen je efektom ispupčenja ili senke na osnovu svoje svetlije ili tamnije nijanse, dok su delovi slike koje imaju slabu kontrast zamenjeni sivom nijansom. Ovaj filter dobija se primenom odgovarajuće konvolucione matrice nad *RGB* komponentama svakog od piksela slike. U implementaciji *emboss* filtera korišćena je sledeća konvoluciona matrica.

$$\begin{matrix} -2 & -1 & 0 \\ -1 & 1 & 1 \\ 0 & 1 & 2 \end{matrix}$$

Filter za *emboss* prihvata sliku kao jedini argument, zatim pravi lokalnu kopiju slike, a onda se vrši obilazak matrice pri čemu se inicijalizuju za svaki piksel buduće vrednosti komponenti *RGB* na 0. Zatim se za svaki piksel vrši prolazak kroz konvolucionu matricu pri čemu je početni indeks (*width/2)*(-1)* a krajnji domet predstavlja (*width/2*) pri čemu *width* je neparan prirodan broj veći od *1* i predstavlja dimenziju konvolucione matrice. U toku prolaska kroz konvolucionu matricu za svaki piksel vrši se procesiranje nove vrednosti komponenti piksela i zatim se ta vrednost setuje kao nova vrednost piksela na toj poziciji.

Na slikama 3.64 i 3.65 prikazane su originalna slika i slika nakon primene konvolucionog filtera emboss.

3.64 Originalna slika pre emboss filtera

3.65 Slika nakon primene emboss filtera

3.3.6 Smooth

Smooth je konvolucioni filter kojim se dobija efekat glačanja slike. Ovaj filter dobija se primenom odgovarajuće konvolucione matrice nad RGB komponentama svakog od piksela slike. U implementaciji smooth filtera korišćena je sledeća konvoluciona matrica.

$$\begin{matrix} 1 & 1 & 1 \\ 1 & 5 & 1 \\ 1 & 1 & 1 \end{matrix}$$

Filter za *smooth* prihvata sliku kao jedini argument, zatim pravi lokalnu kopiju slike, a onda se vrši obilazak matrice pri čemu se inicijalizuju za svaki piksel buduće vrednosti komponenti *RGB* na 0. Zatim se za svaki piksel vrši prolazak kroz konvolucionu matricu pri čemu je početni indeks (*width/2)*(-1)* a krajnji domet predstavlja (*width/2*) pri čemu *width* je neparan prirodan broj veći od *1* i predstavlja dimenziju konvolucione matrice. U toku prolaska kroz konvolucionu matricu za svaki piksel vrši se procesiranje nove vrednosti komponenti piksela i zatim se ta vrednost setuje kao nova vrednost piksela na toj poziciji.

Na slikama 3.66 i 3.67 prikazane su originalna slika i slika nakon primene konvolucionog filtera smooth.

3.66 Originalna slika pre smooth filtera

3.67 Slika nakon primene smooth filtera

3.3.7 Gaussian blur

Gaussian blur predstavlja konvolucioni filter kojim se dobija efekat zamagljivanja slike. Ime je dobio po nemačkom matematičaru Karlu Fridrihu Gausu i njegovoj funkciji koja se koristi u statistici kod normalnih raspodela, u procesiranju signala, prilikom obrade slika itd. To je sledeća funkcija: [9]

$$f(x) = a \exp\left(-\frac{(x-b)^2}{2c^2}\right) + d \tag{3.21}$$

gde su a, b, c, d realne konstante.

Gaussian blur ili Gausov blur dakle predstavlja konvolucioni filter koji koristi Gausovu funkciju da bi se izračunala transformacija koja je potreba da se primeni na svaki piksel slike. Formule za Gausovu funkciju su sledeće:

$$G(x) = \frac{1}{\sqrt{2\pi\sigma^2}} e^{-\frac{x^2}{2\sigma^2}} \tag{3.22}$$

Za jednu dimenziju, odnosno

$$G(x) = \frac{1}{2\pi\sigma^2} e^{-\frac{x^2+y^2}{2\sigma^2}} \tag{3.23}$$

gde su *x* i *y* koordinate, a σ predstavlja standardnu devijaciju Gausove raspodele. Kada se primeni sa dve dimenzije ova formula proizvodi blur efekat u obluku koncentričnih krugova. Vrednosti raspodele koriste se za dobijanje konvolucione matrice koja se primenjuje nad svaki od piksela slike.

Filter za *gaussian blur* prihvata sliku kao prvi argument i parametar sigma tipa relan broj koji predstavlja standardnu devijaciju. Zatim se na osnovu unete standardne devijacije računa veličina kernela – *ksize* po formuli *(3sigma+1)*. A zatim se vrši inicijalizacija kernela, tako što se prolazi kroz matricu veličine *ksize* i računa vrednost na poziciji *(x,y)* po formuli 3.23, nakon toga vrši se normalizacija dobijenog *Gaussian blur* kernela tako što se na svakoj poziciji vrednost deli sa sumom svih vrednosti. Nakon dobijenog kernela postupak je isti kao kod ostalih konvolucionih filtera i to prvo se pravi lokalna kopiju slike, a onda se vrši obilazak matrice pri

čemu se inicijalizuju za svaki piksel buduće vrednosti komponenti *RGB* na 0. Zatim se za svaki piksel vrši prolazak kroz konvolucionu matricu pri čemu je početni indeks (*width/2)*(-1)* a krajnji domet predstavlja (*width/2*) pri čemu *width* je neparan prirodan broj veći od *1* i predstavlja dimenziju konvolucione matrice. U toku prolaska kroz konvolucionu matricu za svaki piksel vrši se procesiranje nove vrednosti komponenti piksela i zatim se ta vrednost postavlja kao nova vrednost piksela na toj poziciji.

Na slikama 3.68 i 3.69 prikazane su originalna slika i slika nakon primene konvolucionog filtera gaussian blur.

3.68 Originalna slika pre gaussian blur filtera

3.69 Slika nakon primene gaussian blur filtera sa standardnom devijacijom sigma = 2

4 Prikaz histograma

U opštem slučaju, histogram je definisan kao način prikazivanja podataka rasporedjenih u odredjene kategorije ili grupe. Histogram ima široku primenu, a posebno je pogodan za prikaz rezultata ispitivanja sprovedenog na velikom broju uzoraka, odnosno kada nije pogodno prikazivati vrednost svakog pojedinačnog ispitivanog uzorka (npr. razna statistička ispitivanja itd). [10]

Jedna od najvažnijih primena histograma je kod digitalnih slika. Na digitalnoj slici osvetljenje pojedinačnog piksela predstavljeno je celobrojnom vrednošću u opsegu [0..255], pri čemu 0 označava najtamniju nijansu odredjene boje, a 255 najsvetliju odredjene boje. Histogram posmatra sliku u celini i odredjuje koliki broj piksela imaju odredjeno (isto) osvetljenje. Dakle, na x-osi su vrednosti osvetljenja od 0 do 255, a na y-osi su brojevi piksela koji imaju odredjeno (isto) osvetljenje. Leva strana histograma predstavlja najtamnije nijanse odredjene boje, dok desna strana histograma predstavlja najsvetlije nijanse odredjene boje. Deo histograma gde je smešten najveći broj vrednosti osvetljenja naziva se raspon tonova. [11]

Osim glavnog RGB histograma, koji je najčešće korišćeni histogram i koji prikazuje osvetljenje piksela na slici, postoje i histogrami po pojedinačnim bojama za crvene, zelene i plave kanale. RGB histogram u tom slučaju predstavlja kombinaciju sva tri kanala, ali na osnovu pojedinačnog kanala boje može da se utvrdi koje boje su najzastupljenije na slici.

U nastavku rada biće opisana implementacija prikaza histograma po pojedinačnim bojama za crvene, zelene i plave kanale za odabranu sliku, kao i prikaza histograma u realnom vremenu, dok je kamera telefona uključena.

4.1 Prikaz histograma za odabranu sliku

Glavne strukture podataka koje se koriste u klasi za prikaz histograma jesu sledeće: mRGBData – niz od width*height celobrojnih vrednosti koje predstavljaju vrednosti piksela slike, pri čemu je width širina, a height visina slike; mBitmap – objekat slike nad kojom se računa i prikazuje histogram; mRedHistogram, mGreenHistogram, mBlueHistogram – nizovi od 256 celobrojnih vrednosti u kome se čuvaju brojevi piksela koji imaju odgovarajuće (isto) osvetljenje (osvetljenje odgovarajuće boje je u opsegu [0..255] pri čemu 0 predstavlja najtamniju nijansu odredjene boje, a 255 najsvetliju nijansu). Najvažniji deo koda koji poziva metodu za računanje histograma i metoda koja obavlja sumiranje broja piksela koji imaju odredjeno (isto) osvetljenje i inicijalizaciju odgovarajućih nizova (mRedHistogram, mGreenHistoram, mBlueHistogram) prikazan je u listingu 4.1:

Listing 4.1: Metoda za izračunavanje histograma

```
calculateIntensityHistogram(mRGBData, mRedHistogram, mImageWidth,
mImageHeight, 0);
calculateIntensityHistogram(mRGBData, mGreenHistogram, mImageWidth,
mImageHeight, 1);
calculateIntensityHistogram(mRGBData, mBlueHistogram, mImageWidth,
mImageHeight, 2);

static public void calculateIntensityHistogram(int[] rgb, int[] histogram,
                  int width, int height, int component) {
            for (int bin = 0; bin < 256; bin++) {
                  histogram[bin] = 0;
            }
            if (component == 0) // red
            {
                  for (int pix = 0; pix < width * height; pix++) {
                        int pixVal = (rgb[pix] >> 16) & 0xff;
                        histogram[pixVal]++;
                  }
            } else if (component == 1) // green
            {
                  for (int pix = 0; pix < width * height; pix++) {
                        int pixVal = (rgb[pix] >> 8) & 0xff;
                        histogram[pixVal]++;
                  }
            } else // blue
            {
                  for (int pix = 0; pix < width * height; pix++) {
                        int pixVal = rgb[pix] & 0xff;
                        histogram[pixVal]++;
                  }
            }
      }
```

Metoda calculateIntensityHistogram prima 5 argumenata, a to su: niz vrednosti piksela (koji ima width*height elemenata), niz u kome se smeštaju brojevi piksela koji imaju odgovarajuće (isto) osvetljenje, a koji je različit za svaki od 3 kanala boje (mRedHistogram, mGreenHistogram, mBlueHistogram), širina i visina slike, kao i flag koji odredjuje za koji kanal boje se histogram izračunava (vrednost 0 označava da se preračunava histogram za crveni kanal boje, vrednost 1 označava da se preračunava histogram za zeleni kanal boje, dok vrednost 2 označava da se preračunava histogram za plavi kanal boje). Prikazana metoda najpre invaliduje sve elemente niza u kome se čuvaju brojevi piksela koji imaju odgovarajuće (isto) osvetljenje, a zatim, u zavisnosti od flag-a koji je prosledjen, iterira kroz niz piksela slike (kojih ima width*height) i dohvata odgovarajuću komponentu boje piksela (crvenu, zelenu ili plavu) na osnovu poznatog načina skladištenja pojedinačnih komponenti piksela (formula 3.2). Pojedinačna komponenta piksela (crvena, zelena, plava) dobija se po formulama 3.5, 3.6 i 3.7, što je iskorišćeno u datom kodu, a mogle su se koristiti i gotove metode Android operativnog sistema Color.red, Color.green i Color.blue. Na taj način dobijamo vrednost pixVal koja

predstavlja vrednost osvetljenja odgovarajuće boje (crvene, zelene ili plave) i u opsegu je [0..255]. Ostaje još da broj elemenata niza koji imaju taj indeks uvećamo za jedan čime, u stvari, sumiramo broj piksela koji imaju odgovarajuće (isto) osvetljenje.

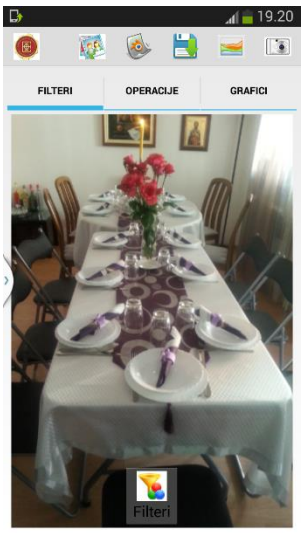
Osim samog izračunavanja histograma za pojedinačni kanal boje, vršeno je izračunavanje i srednjih vrednosti osvetljenja za pojedinačni kanal boje i deo koda koji to obavlja prikazan je u listingu 4.2:

Listing 4.2: Izračunavanje srednjih vrednosti osvetljenja za pojedinačni kanal boje

```
double imageRedMean = 0, imageGreenMean = 0, imageBlueMean = 0;
for (int bin = 0; bin < 256; bin++) {
      imageRedMean += mRedHistogram[bin] * bin;
      redHistogramSum += mRedHistogram[bin];
      imageGreenMean += mGreenHistogram[bin] * bin;
      greenHistogramSum += mGreenHistogram[bin];
      imageBlueMean += mBlueHistogram[bin] * bin;
      blueHistogramSum += mBlueHistogram[bin];
      }
imageRedMean /= redHistogramSum;
imageGreenMean /= greenHistogramSum;
imageBlueMean /= blueHistogramSum;
```

Najpre se vrši sumiranje vrednosti osvetljenja pojedinačnih komponenti (crvene, zelene, plave) svih piksela, i te vrednosti se čuvaju u promenljivama imageRedMean, imageGreenMean i imageBlueMean, dok se u promenljivama redHistogramSum, greenHistogramSum i blueHistogramSum čuva ukupan broj piksela. Deljenjem odgovarajućih vrednosti dobijaju se srednje vrednosti osvetljenja za pojedinačni kanal boje.

Pored ovoga, implementirano je i da kada se korisniku prikažu 3 histograma (za svaki kanal boje) on može da klikom na željeni histogram (neki njegov deo) dobije informaciju o tome koliko piksela imaju odgovarajuće osvetljenje koje je odabrao. Originalna slika prikazana je na slici 4.1, histogram originalne slike (po jedan za svaki kanal boje) prikazan je na slici 4.2, a prikaz broja piksela sa odgovarajućim osvetljenjem pojedinačnog kanala boje prikazan je na slici 4.3.



4.1 Originalna slika

4.2 Histogram nad originalnom slikom

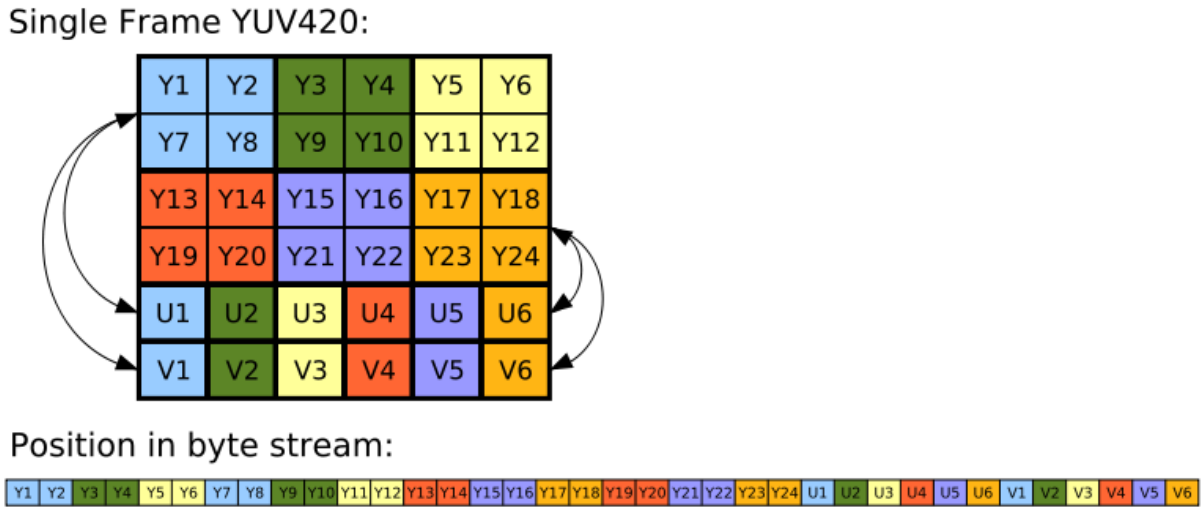
4.3 Prikaz informacija o histogramu

4.2 Prikaz histograma u realnom vremenu

Da bismo mogli prikazati histogram u realnom vremenu, dok je kamera telefona uključena, neophodno je iskoristiti već opisanu implementaciju prikaza histograma za svaki pojedinačni kanal boje (crveni, zeleni, plavi), a pored toga je neophodno implementirati još nekoliko stvari. Jedna od tih je pomoćna klasa koja proširuje Android klasu SurfaceView i implemetira Android interfejs SurfaceHolder.Callback. Izvodjenje iz klase SurfaceView neophodno je iz razloga što se slika (a samim tim i izgled histograma za sva tri kanala) na ekranu menja jako brzo, jer korisnik može proizvoljnom brzinom pomerati telefon i u svakom trenutku treba prikazati ažurne informacije na ekranu, analogno snimanju video zapisa. [12] Implementacija interfejsa SurfaceHolder.Callback neophodna je da bismo mogli da primimo informacije o promeni našeg surface-a, odnosno promeni slike koja je u tom trenutku vidljiva preko kamere telefona. Da bi to bilo omogućeno, neophodno je implementirati 3 najvažnije metode tog interfejsa, a to su: surfaceCreated – poziva se odmah nakon što se surface kreira prvi put; surfaceChanged – poziva se odmah nakon što dodje do bilo kakve strukturne izmene nad surface-om; surfaceDestroyed – poziva se malo pre nego što se surface uništi. [13] U svakoj od tih metoda, vrše se odgovarajuće akcije kako bi se ceo postupak prikaza histograma u realnom vremenu uspešno sproveo: preko instanciranja objekta kamere i dohvatanja njenih parametara (visina, širina), kreiranja odgovarajućih objekata Bitmap-e, kao i nizova u kojima će se čuvati vrednosti piksela tekuće Bitmap-e, postavljanja dimenzija za slike koje će se dohvatati preko kamere, postavljanja tzv. scene i focus načina rada, pa do uništavanja objekta kamere.

U okviru metode surfaceCreated neophodno je setovati povratni poziv (callback) objektu kamere koji će biti pozivan za svaki frame prikazan na ekranu preko metode setPreviewCallback. Taj callback će biti periodično pozivan sve dok je preview aktivan. Kao argument ovoj metodi prosledjujemo novokreirani objekat tipa PreviewCallback kome moramo implementirati metodu onPreviewFrame koja se poziva prilikom prikazivanja frame-a i kao argumente prima objekat kamere i niz bajtova koji predstavljaju sadržaj frame-a koji se prikazuje na ekranu u formatu koji je podrazumevano YUV420p format. Da bismo mogli iskoristiti već opisane metode za računanje histograma (koji računanje vrši na osnovu celobrojnih vrednosti piksela), niz bajtova u YUV420p formatu moramo konvertovati u niz celobrojnih vrednosti u ARGB formatu.

YUV je prostor boja koji se tipično koristi kao deo pipeline-a kolor slike. Ovaj format enkoduje sliku u boji ili video uzimajući u obzir ljudsku percepciju omogućavajući smanjenu propusnost za hrominentne komponente (chrominance) tipično na taj način omogućavajući da se greške u prenosu ili nus-efekti kompresije efikasnije maskiraju više ljudskom percepcijom nego klasičnim RGB predstavljanjem. YUV model definiše prostor boja koji se sastoji od jedne luma komponente (Y) i dve chroma komponente (UV). Luma predstavlja osvetljenost u slici, odnosno ahromatski deo slike, dok se chroma koristi da prenese informacije o boji slike, odvojene od pratećeg luma signala. YUV420p je ravan format što znači da su Y, U i V vrednosti grupisane zajedno umesto da su umetnute jedna pored druge. Razlog za to je što se grupisanjem U i V vrednosti zajedno dobija da slika postaje mnogo pogodnija za kompresiju. Kada dobijemo niz bajtova koji predstavljaju sliku u YUV420p formatu, sve Y vrednosti dolaze na početku tog niza praćene svim U vrednostima, koje su konačno praćene svim V vrednostima do kraja niza. Kao i kod većine YUV formata, i kod YUV420p formata postoji onoliko Y vrednosti koliko ima piksela na slici. Ukoliko nam je size = width*height, gde su width i height širina i visina slike, respektivno, prvih size vrednosti u nizu predstavljaju Y vrednosti koje odgovaraju svakom pojedinačnom pikselu. Medjutim, postoji samo size/4 U i V vrednosti – U i V vrednosti odgovaraju svakom bloku 2x2 na slici, odnosno svaka U i V vrednost se odnosi na po 4 piksela. Nakon Y vrednosti, narednih size/4 vrednosti u nizu predstavljaju U vrednosti (po jedna U vrednost za svaki blok veličine 2x2) i poslednjih size/4 vrednosti u nizu predstavljaju V vrednosti (po jedna V vrednost za svaki blok veličine 2x2). Opisani način skladištenja Y, U i V vrednosti prikazan je na slici 4.4. [14]



4.4 Skladištenje komponenti u YUV420

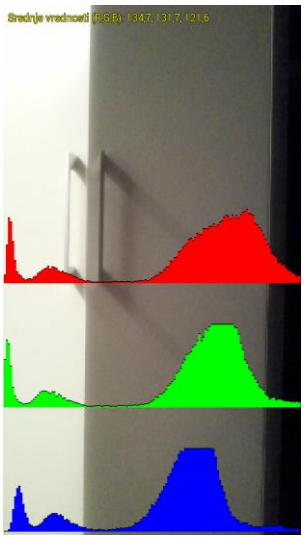
Kao što je prikazano na slici, Y, U i V komponente se smeštaju odvojeno u sekvencijalne blokove – Y vrednost se čuva za svaki piksel, dok se U i V vrednosti čuvaju za svaki kvadratni blok od 2x2 piksela. Odgovarajuće Y, U i V komponente su prikazane istom bojom na slici. Y blok u nizu bajtova koji se dobija od uredjaja (u našem slučaju sa kamere telefona) počinje od pozicije 0, U blok od pozicije size, a V blok od pozicije size + size/4. NV21 je standardni format za slike koji Android kamera koristi, koji u stvari predstavlja ravan YUV 4:2:0 format. Deo koda koji vrši konverziju niza bajtova iz YUV420p formata u niz celobrojnih vrednosti u ARGB formatu prikazan je u listingu 4.3:

Listing 4.3: Konverzija niza bajtova iz YUV420p formata u ARGB format

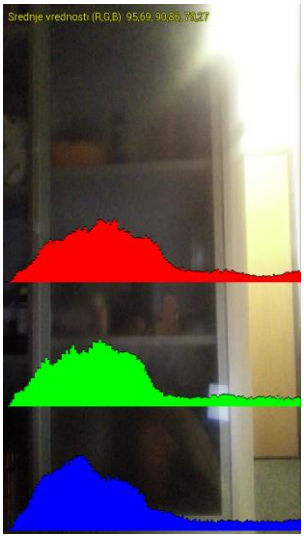
```
public static void convertYUV420_NV21toRGB8888(int [] rgb, byte [] data, int
width, int height) {
      int size = width*height, offset = size, u, v, y1, y2, y3, y4;
      for(int i=0, k=0; i < size; i+=2, k+=2) {
            y1 = data[i] & 0xff; y2 = data[i+1] & 0xff;
            y3 = data[width+i] & 0xff; y4 = data[width+i+1] & 0xff;
            u = data[offset+k] & 0xff; v = data[offset+k+1] & 0xff;
            u = u-128; v = v-128;
            rgb[i] = convertYUVtoRGB(y1, u, v);
            rgb[i+1] = convertYUVtoRGB(y2, u, v);
            rgb[width+i] = convertYUVtoRGB(y3, u, v);
            rgb[width+i+1] = convertYUVtoRGB(y4, u, v);
            if (i!=0 && (i+2)%width==0)
                i+=width;
        }
    }


    private static int convertYUVtoRGB(int y, int u, int v) {
        int r,g,b;
        r = y + (int)1.402f*v;
        g = y - (int)(0.344f*u +0.714f*v);
        b = y + (int)1.772f*u;
        r = r>255? 255 : r<0 ? 0 : r;
        g = g>255? 255 : g<0 ? 0 : g;
        b = b>255? 255 : b<0 ? 0 : b;
        return 0xff000000 | (b<<16) | (g<<8) | r;
    }
```

Metoda convertYUV420_NV21toRGB8888 prima četiri argumenta, a to su: niz celobrojnih vrednosti koji će biti inicijalizovan odgovarajućim vrednostima piksela nakon konverzije, niz bajtova u YUV420 formatu koje treba konvertovati, širina i visina slike respektivno. U svakoj iteraciji petlje preračunava se po 4 Y vrednosti uzimajući bajtove sa odgovarajućih pozicija (operacija data[...] & 0xff izdvaja tačno 8 bitova koliko nam treba) i po jedna U i V vrednost. Od U i V vrednosti oduzimamo 128 jer te komponente mogu biti pozitivne ili negativne, ali su one uskladištene kao unsigned (neoznačene) vrednosti, pa je konstanta 128 bila dodata na njihovu vrednost prilikom njenog generisanja da bi se te komponente zadržale kao pozitivne, pa mi sada moramo oduzeti konstantu 128 da bi se vrednost restaurirala na pravilan

način. Kombinacijom odgovarajućih Y, U i V vrednosti i njihovom konverzijom u ARGB format dobija se celobrojna vrednost piksela koja se upisuje u niz rgb na odgovarajuće mesto (indeks rezultujućeg piksela odgovara indeksu njegove Y komponente). Ukoliko smo obradili widht bajtova iz ulaznog niza, prelazimo u naredni red. Konverzija Y, U i V komponente u celobrojnu vrednost piksela prikazana je u metodi convertYUVtoRGB koja prima tri celobrojne vrednosti, setuje r, g i b komponente piksela i na osnovu poznatog načina skladištenja pojedinačnih komponenti piksela (formula 3.4) vraća jedinstvenu celobrojnu vrednost odgovarajućeg piksela. Histogram u realnom vremenu prikazan je na slikama 4.5 i 4.6.



4.5 Histogram u realnom vremenu 1



4.6 Histogram u realnom vremenu 2

5 Pregled perfmormansi filtera

Poslednji modul koji priloženi projekat obuhvata je pregled performansi filtera, koji se sastoji od tri podmodula, a to su pregled performansi unarnih filtera, pregled performansi binarnih operacija i pregled performansi gausian blur filtera u zavisnosti od parametra unetog od strane korisnika. Vremena izvršavanja odgovarajućih filtera prikazana su grafički, preko tzv. bar chart-a, a implementiran je i prikaz statističkih podataka o vremenima izvršavanja i poredjenjima performansi filtera. Sva testiranja vršena su na Android operativnom sistemu, konkretno na smartphone-ovima marke Samsung Galaxy S3 i Samsung Galaxy S4. Neophodno je još napomenuti da zbog ograničene RAM memorije, kao i karakteristika procesora, performanse su lošije nego da je testiranje vršeno na jačim Android uredjajima novije generacije, ali s druge strane, znatno su bolje nego da je isto vršeno na slabijim Android uredjajima starijih generacija, u pogledu brzine procesora i količine memorije. Te dve primarne karakteristike na pomenutim telefonima iznose:

- Samsung Galaxy S3: quad-core (4-jezgarni) CPU, brzine 1.4GHz; 1GB RAM

- Samsung Galaxy S4: quad-core (4-jezgarni) CPU, brzine 1.6GHz; 2GB RAM

Takodje, treba još napomenuti da bi implementacija filtera preko native Android funkcija napisanih u C-u ili C++u dala još bolje performanse iz razloga što veliki deo aplikacije predstavlja izvršavanje vremenski zahtevnih operacija nad svakim pojedinačnim pikselom slike, koje zahtevaju veliko procesorsko vreme.

5.1 Pregled performansi unarnih filtera

Da bi se vremena izvršavanja unarnih filtera verodostojno prikazala, odabrano je da se upis imena filtera i vreme njegovog izvršavanja čuva u CSV (comma-separated values) fajlu koji je uskladišten u memoriji telefona i koji se ažurira svaki put kada se zahteva izvršavanje nekog od filtera nad odabranom slikom. Radi lakšeg čitanja iz CSV fajla i upisa u isti, kao pomoćna bibiloteka korišćena je opencsv jar, verzija 2.1. [15] Ono što je još bitno napomenuti je to da nekoliko unarnih filtera zahtevaju unos jednog ili više parametara od strane korisnika koji su neophodni za izvršavanje tog filtera nad slikom, a to su sledeći: black&white, brightness, contrast, gausianBlur, gammaCorrection, colorFilter, saturationFilter, hueFilter i shadingFilter. Medjutim, primećeno je da su svi ovi filteri (osim gausianBlur filtera koji predstavlja konvolucioni filter) osnovni unarni filteri, koji ne zahtevaju mnogo procesorkog vremena i koji nad tim ulaznim parametrima izvršavaju sledeće operacije:

- black&white: operacije množenja, deljenja i poredjenja ulaznog paramtera na vrednost

- brightness: operacije sabiranja ulaznog parametra sa r, g i b komponentama piksela

- contrast: operacije sabiranja, množenja i deljenja sa ulaznim parametrom

- gammaCorrection: operacija eksponentizacije sa ulaznim parametrom

- colorFilter: operacije množenja sa ulaznim parametrima

- saturationFilter: operacija čitanja ulaznog parametra

- hueFilter: operacija čitanja ulaznog paramtra

- shadingFilter: bitska operacija I (AND) sa ulaznim parametrom

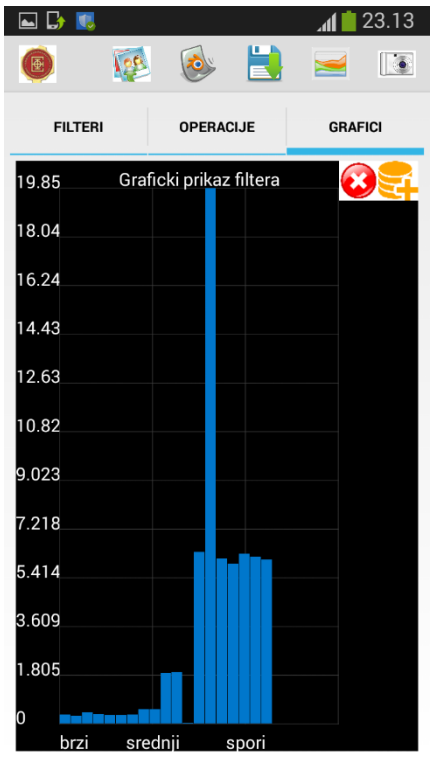
Na osnovu toga je zaključeno da promena ulaznog parametra (ili ulaznih parametara) ne utiče na vreme izvršavanja kod osnovnih unarnih filtera, pa implementacija grafika zavisnosti vremena izvršavanja od vrednosti ulaznog parametra nije bila od interesa u aplikaciji. Za razliku od pomenutih filtera, vreme izvršavanja gausian blur filtera u najvećoj meri zavisi od vrednosti ulaznog parametra sigma, jer se radi o konvolucionom filteru, koji koristi konvolucionu matricu za izračunavanje, a dimenzija te matrice zavisi od sigma ulaznog parametra. Iz tog razloga je implementiran poseban grafik zavisnosti vremena izvršavanja gausian blur filtera od ulaznog parametra sigma o kome će biti reči u nastavku rada.

Već je pomenuto da se parovi filter – vreme izvršavanja čuvaju u CSV fajlu, pri čemu se prilikom zahteva korisnika za prikazom odgovarajućeg grafika iterira kroz sve parove u CSV fajlu, pronalaze se vremena izvršavanja pojedinačnog filtera, računa srednje vreme izvršavanja, koje se i prikazuje na grafiku. Za samo iscrtavanje grafika korišćena je gotova biblioteka graphview.jar, verzija 3.1. [16] Deo sadržaja CSV fajla koji se odnosi na performanse unarnih filtera prikazan je na slici 5.1.

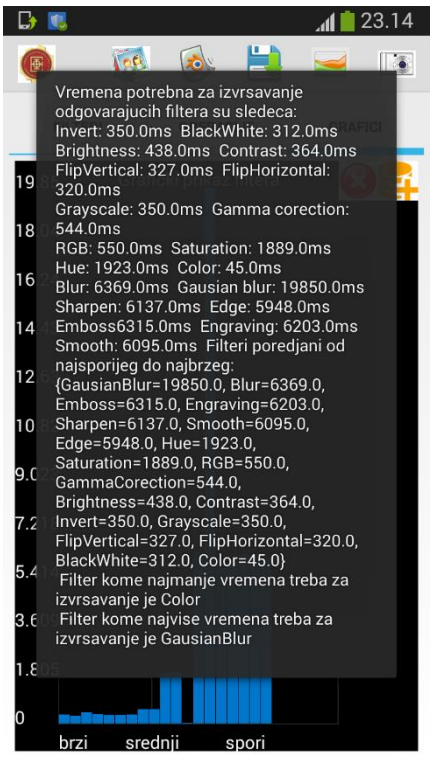
```
GausianBlurFilter,"5719","0.2"
GausianBlurFilter,"15033","0.9"
GausianBlurFilter,"28878","1.4"
GausianBlurFilter,"29770","2.0"
InvertFilter,"350"
BlackWhiteFilter,"312"
BrightnessFilter,"438"
ContrastFilter,"364"
FlipVerticalFilter,"327"
FlipHorizontalFilter,"320"
GrayscaleFilter,"350"
GammaCorectionFilter,"544"
RGBFilter,"550"
SaturationFilter,"1889"
HueFilter,"1923"
ColorFilter,"45"
BlurFilter,"6369"
SharpenFilter,"6137"
EdgeFilter,"5948"
EmbossFilter,"6315"
EngravingFilter,"6203"
SmoothFilter,"6095"
```

5.1 Sadržaj CSV fajla 1

Sa slike se može videti da se za sve filtere osim gausian blur filtera čuvaju dva podatka – ime filtera i vreme izvršavanja filtera u milisekundama, dok se za gausian blur filter čuvaju tri podatka – ime, vreme izvršavanja i vrednost ulaznog parametra sigma, pri čemu će nam treći parametar biti od interesa prilikom prikaza grafika zavisnosti vremena izvršavanja gausian blur filtera od vrednosti ulaznog parametra sigma. Grafički prikaz performansi unarnih filtera i prikaz statističkih podataka prikazani su na slikama 5.2 i 5.3.



5.2 Grafički prikaz performansi unarnih filtera



5.3 Prikaz statističkih podataka unarnih filtera

Sa slika 5.2 i 5.3 može se videti da je filter koji zahteva najmanje procesorskog vremena shading filter (nazvan color filter kod prikaza statističkih podataka) za čije izvršavanje je u proseku potrebno 45 milisekundi. Razlog tome je što je jedina operacija koje se izvršava u okviru algoritma za primenu shading filtera bitska operacija I (AND) izmedju odgovarajućeg piksela i boje kojom se slika senči, a sve bitske operacije (uključujući i bitsko I) su jako brze, primitivne operacije direktno podržane u procesoru i višestruko su brže od operacija množenja, deljenja i sabiranja. Nakon toga, slede filteri koji imaju približno srednje vreme izvršavanja, a to su sledeći, poredjani od najbržeg ka najsporijem: black&white, flip horizontal, flip vertical, grayscale, invert, contrast, brightness, gamma correction, RGB (color) filter. Nakon toga, slede filteri saturation i hue koji su nešto sporiji od do sada opisanih iz razloga što se za njihovu implementaciju za svaki piksel vrši konverzija vrednosti piksela iz ARGB modela u HSV model, promena odgovarajuće vrednosti saturation, odnosno hue komponente i opet se vrši konverzija vrednosti piksela sada iz HSV modela u ARGB model. Nakon toga, slede zahtevniji filteri za primenu nad odabanom slikom u pogledu procesorskog vremena neophodnog za izvršavanje, a to su konvolucioni filteri koji za svoje izvršavanje osim matrice piksela slike, koriste i konvolucionu matricu dimenzija 3x3 i vrednost svakog piksela se dobija tako što se množe vrednosti tog piksela i njemu susednih 8 piksela (uzima se blok dimenzija 3x3) sa vrednostima iz konvolucione matrice, ti medjurezultati se sabiraju i na taj način se dobija konačna, izmenjena vrednost piksela. Algoritmi za implementaciju konvolucionih filtera nad slikom iteriraju kroz 4 ugneždjene for petlje, pa je i logično što dolazi do naglog povećanja vremena neophodnog za

njihovo izvršenje. Reč je o filterima blur, sharpen, engraving, smooth, emboss, edge i gausian blur. Posebno treba obratiti pažnju na gausian blur filter koji je drastično zahtevniji za primenu od ostalih, što običnih, što konvolucionih filtera, iz razloga što njegova konvoluciona matrica nije fiksnih dimenzija 3x3 i nema predefinisane vrednosti, već se dimenzija matrice i njene vrednosti računaju na osnovu ulaznog parametra sigma. U projektu je definisano da se gausian blur filter može primenjivati sa konvolucionom matricom dimenzija 3x3, 5x5 i 7x7 (teorijski bi ove dimenzije mogle biti i veće, ali je ovde uzeto ovakvo ograničenje da se procesor Android telefona i memorija neophodna za izvršenje filtera ne bi preopteretili), pa se samim tim povećava broj iteracija for petlji, kao i vreme izvršenja gausian blur filtera sa povećanjem sigma ulaznog parametra. Kako je ovde vršeno usrednjavanje vremena potrebnih za izvršenje odgovarajućeg filtera, očekivano je bilo da tako dobijeno vreme za gausian blur filter bude veće od svih ostalih srednjih vremena i u proseku iznosi 19850 milisekundi, odnosno 19.85 sekundi.

5.2 Pregled performansi Gausian Blur filtera

Kao što je već rečeno, za gausian blur filter u CSV fajlu čuvamo tri podatka, ime filtera – na osnovu imena vršimo usrednjavanje odgovarajućih vremena izvršavanja, vreme izvršavanja filtera i vrednost ulaznog parametra sigma za koju se filter toliko vremena izvršavao. Kod gausian blur filtera nam je od interesa i vrednost ulaznog parametra sigma, za koju je odlučeno da je korisnik odabira, ali mora biti u opsegu [0..2], s obzirom da se na osnovu tog parametra računa dimenzija konvolucione matrice po sledećoj formuli:

$$ksize = sigma * 3 + 1 \tag{5.1}$$

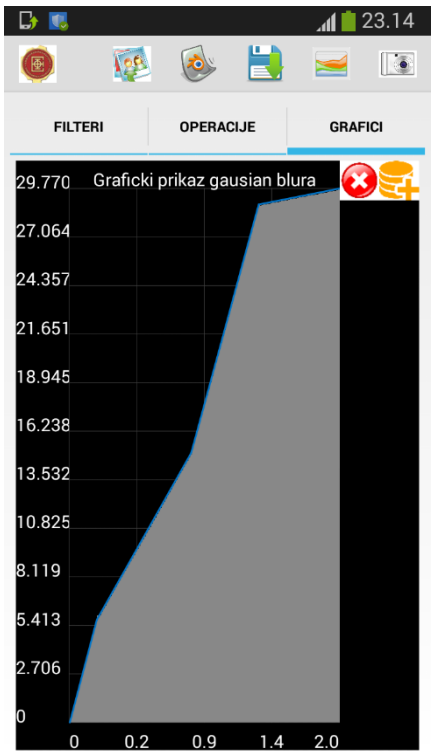
Treba napomenuti da ukoliko korisnik unese jako malu vrednost za parametar sigma, takvu da ksize bude 1, nema smisla primenjivati gausian blur nad slikom, jer bi se u tom slučaju formirala konvoluciona matrica 1x1 koja bi obuhvatala samo jedan piksel koji se u tom trenutku obradjuje, a to nije ideja ovog algoritma, već se za konačnu vrednost piksela uzima u obzir i vrednosti svih suseda tekućeg piksela. Takodje, dimenzija konvolucione matrice treba da bude neparan broj, stoga ako se za ksize dobije paran broj, on će biti uvećan za 1. Zbog opsega za ulazni parameter sigma koji je od 0 do 2, postignuto je algoritam za primenu gausian blur filtera radi sa konvolucionim matricama dimenzija 3x3, 5x5 ili 7x7, u zavisnosti od unetog sigma parametra. Teorijski bi dimenzija konvolucione matrice mogla da bude i veća, ali je ovde odlučeno da se ne ide preko vrednosti 7, kako se procesorsko vreme ne bi višestruko povećalo i

uredjaj preopteretio. Deo sadržaja CSV fajla koji se odnosi na performanse gausian blur filtera prikazan je na slici 5.4.

```
GausianBlurFilter,"5719","0.2"
GausianBlurFilter,"15033","0.9"
GausianBlurFilter,"28878","1.4"
GausianBlurFilter,"29770","2.0"
```

5.4 Sadržaj CSV fajla 2

Kao što se može videti na osnovu sadržaja CSV fajla, za porast ulaznog parametra sigma, povećava se vreme izvršavanja gausian blur filtera, mada ne mora to uvek biti slučaj, naime ukoliko se odabere približna vrednost sigma parametra u odnosu na neku prethodnu, koja će po formuli 5.1 dati istu vrednost dimenzije konvolucione matrice (ksize), vreme izvršavanja filtera može biti približno isto ili manje, iako je vrednost sigma parametra neznatno povećana. To se dešava iz razloga što na vreme izvršavanja gausian blur filtera direktno utiče dimenzija konvolucione matrice, a za veliki broj različitih vrednosti parametara sigma se može dobiti ista vrednost dimenzije konvolucione matrice. Grafički prikaz performansi gausian blur filtera i prikaz statističkih podataka prikazani su na slikama 5.5 i 5.6.



5.5 Graficki prikaz performansi gausian blur filtera

5.6 Prikaz statističkih podataka gausian blur filtera

5.3 Pregled performansi binarnih operacija

Da bi se vremena izvršavanja binarnih operacija nad slikama verodostojno prikazala, odabrano je da se upis imena operacije i vreme njegovog izvršavanja čuva u CSV (comma-separated values) fajlu koji je uskladišten u memoriji telefona i koji se ažurira svaki put kada se zahteva izvršavanje neke od operacija nad odabranim slikama. Ono što je još bitno napomenuti je to da binarna operacija blend zahteva unos parametara alpha od strane korisnika koji je neophodan za izvršavanje te operacije nad slikama. Medjutim, primećeno je da ova operacija nad tim ulaznim parametrom alpha izvršava operacije oduzimanja vrednosti alpha od konstante 1 i množenje tako dobijene vrednosti sa pojedinačnim komponentama piksela jedne slike, kao i množenje parametra alpha sa pojedinačnim komponentama (red, green, blue) piksela druge slike, pa te dve vrednosti sabira u rezultujuću komponentu piksela. Na osnovu toga je zaključeno da promena ulaznog parametra alpha ne utiče na vreme izvršavanja binarne operacije blend, pa implementacija grafika zavisnosti vremena izvršavanja od vrednosti ulaznog parametra nije bila od interesa u aplikaciji. Stoga je implentiran prikaz vremena izvršavanja pojedinačnih binarnih operacija medjusobno ih uporedjujući po tom kriterijumu.

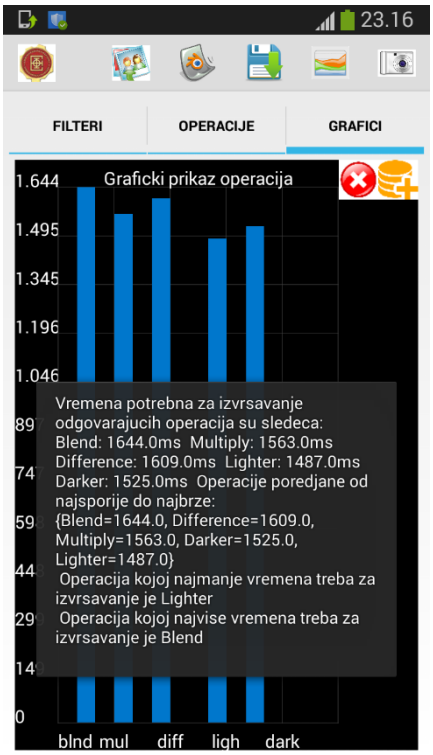
Već je pomenuto da se parovi operacija – vreme izvršavanja čuvaju u CSV fajlu, pri čemu se prilikom zahteva korisnika za prikazom odgovarajućeg grafika iterira kroz sve parove u CSV fajlu, pronalaze se vremena izvršavanja pojedinačne operacije, računa srednje vreme izvršavanja, koje se i prikazuje na grafiku. Deo sadržaja CSV fajla koji se odnosi na performanse binarnih operacija prikazan je na slici 5.7.

BlendOperation,"1773"
BlendOperation,"1657"
MultiplyOperation,"1569"
DifferenceOperation,"1604"
LighterOperation,"1499"
DarkerOperation,"1466"
BlendOperation,"1546"
MultiplyOperation,"1566"
DifferenceOperation,"1534"
LighterOperation,"1461"
DarkerOperation,"1477"
BlendOperation,"1603"
MultiplyOperation,"1554"
DifferenceOperation,"1689"
LighterOperation,"1501"
DarkerOperation,"1632"

5.7 Sadržaj CSV fajla 3

Grafički prikaz performansi binarnih operacija i prikaz statističkih podataka prikazani su na slikama 5.8 i 5.9.

5.8 Grafički prikaz performansi binarnih operacija



5.9 Prikaz statističkih podataka binarnih operacija

Sa slika se vidi da su binarne operacije kojima najmanje treba procesorskog vremena za izvršavanje operacije Lighter i Darker čije je prosečno vreme izvršavanja 1487, odnosno 1525 milisekundi. Za izvršavanje ovih operacija neophodno je najpre za svaki piksel polaznih slika izračunati vrednost luminance, odnosno osvetljenost, koja se računa po sledećoj formuli:

$$0.2126 * Color.red(pixel)) + (0.7152 * Color.green(pixel)) + (0.0722 * Color.blue(pixel) \quad (5.2)$$

Nakon toga, kod operacije Lighter se, ukoliko je vrednost luminance odgovarajućeg piksela prve slike veća od vrednosti luminance odgovarajućeg piksela druge slike, uzima piksel prve slike, a u suprotnom, piksel druge slike. Kod operacije Darker se radi obrnuto. Dakle, ove dve operacije jedino obavljaju aritmetičke operacije množenja, sabiranja i poredjenja, te je i logično da se brže izvršavaju od ostalih operacija. Nakon njih, sledeća po vremenu izvršavanja je operacija Multiply koja nad pikselima polaznih slika koji se nalaze na istim pozicijima u nizu piksela vrši bitsku operaciju I (AND) za koju je već rečeno da je jako brza, direktno podržana u procesoru, te je i bilo očekivano da ona bude približna po vremenu izvršavanja sa operacijama Lighter i Darker. Dve operacije kojima najviše vremena treba za izvršavanje su Difference i Blend, pri čemu operacija Difference troši manje procesorskog vremena od operacije Blend. Operacija Difference vrši oduzimanje odgovarajućih komponenti piskela koji se nalaze na istim pozicijama u ulaznim slikama i uzima apsolutnu vrednost od tog rezultata. Operacija Blend, kao što je već rečeno, uzima vrednost ulaznog parametra alpha i izvršava operacije oduzimanja vrednosti alpha od konstante 1 i množenje tako dobijene vrednosti sa pojedinačnim komponentama piksela jedne slike, kao i množenje parametra alpha sa pojedinačnim komponentama (red, green, blue) piksela druge slike, te sabira dobijene vrednosti u rezultujuću komponentu piksela. Operacija Blend je zahtevnija za izvršavanje od svih ostalih operacija implementiranih u ovom projektu i njeno prosečno vreme izvršavanja iznosi 1644 milisekundi.

Ipak, treba napomenuti da su vremena izvršavanja binarnih operacija mnogo ujednačenija nego što je to bio slučaj kod vremena izvršavanja filtera. Razlog tome je što sve implementirane operacije koriste manje-više slične aritmetičke i bitske operatore u svojim algoritmima, te je i njihovo vreme izvršavanja dosta slično. Za razliku od njih, kod implementacije filtera smo imali slučaj da su pojedini filteri zahtevali konverziju vrednosti piksela iz jednog formata u drugi (Saturation i Hue), dok su konvolucioni filteri koristili vrednosti iz svojih konvolucionih matrica i vrednosti okružujućih piksela za izračunavanje vrednosti svakog pojedinačnog piksela, što je, videli smo, najzahtevnije za izvršavanje po pitanju utrošenog procesorskog vremena.

6 *Zaključak*

Aplikacija Obrada slika na Android operativnom sistemu radjena je u okviru predmeta Multimedijalni sistemi na Elektrotehničkom fakultetu u Beogradu i rezultirala je sledećim zaključcima i naučnim doprinosima:

- primene različitih filtera, pogotovo konvolucionih, procesorki su jako zahtevne operacije u pogledu potrošnje procesorskog vremena

- dobijene perfomanse i rezultati testiranja dosta zavise od hardverske platforme na kojoj je testiranje radjeno, tačnije od karakteristika procesora i RAM memorije uredjaja

- testiranja su vršena na uredjajima relativno novijih generacija, te su rezultati prihvatljivi i izvršenja različitih funkcionlosti prilikom korišćenja aplikacije su vremenski zadovoljavajuća jer ne zahtevaju previše čekanja od strane korisnika na njihovo izvšenje

- procesorski su posebno zahtevni konvolucioni filteri, pogotovo Gausian Blur filter, koji se može izvršavati koristeći konvolucionu matricu (kernel) dimenzija većih od standardnih 3x3, pri čemu se sa povećanjem dimenzija matice višestruko povećava vreme neophodno za izvršenje filtera

- ostali unarni filteri, osim konvolucionih, kao i binarne operacije imaju slične performanse u pogledu vremena izvršavanja

- prikaz histograma, kako nad pojedinačnom slikom, tako i u realnom vremenu, zadovoljavajuće je brzine i okom korisnika se ne primećuje čekanje neophodno za izračunavanje i prikazivanje istog, a koje se obavlja u pozadini (u realnom vremenu kako se menja slika zahvaćena kamerom telefona)

- dobijene performanse ne zavise od veličine, odnosno broja bajtova koje slika zauzima u memoriji telefona, kao i dimenzija slike s obzirom da je pre korišćenja svih funkcionalnosti izvršavano sklaliranje slika kako ne bi došlo do prekoračenja dozvoljene memorije za učitanu sliku

- dobijene performanse bi bile bolje da je testiranje vršeno na procesorki i memorijski jačim Android uredjajima novijih generacija u odnosu na Samsung Galaxy S3 i Samsung Galaxy S4

- dobijene performanse bi bile bolje da su implementirane *native* metode za obradu slika koje bi dozvolile direktan pristup memoriji i rad sa registrima

- dobijene performanse bi bile bolje kada bi se implementacija vršila na multiprocesorskim sistemima koji su pogodni za ovakvu vrstu algoritama i kod kojih bi celokupan „posao“ koji treba da se izvrši u toku rada algoritma bilo delegiran na veći broj procesora koji bi paralelno radili

Moguć nastavak ovog istraživanja i nadogradnja priložene aplikacije sastojale bi se u pokušaju prevazilaženja problema vezanog za prekoračenje dozvoljene memorije za učitanu sliku (*Memory Overflow*) čijim rešavanjem bi se moglo utvrditi i kako veličina, dimenzije i ostale karakteristike pojedinačne slike utiču na vreme izvršavanja filtera nad istom. Takodje, implementacija *native* metoda za obradu slika i/ili implementacija na multiprocesorskom sistemu korišćenjem tehnologija novije generacije poput MPI-a (Message Passing Interface), OpenMP tehnologije ili pak paralelne kompjuterske platforme – CUDA (Compute Unified Device Architecture) su nešto što je planirano za budućnost i produbljivanje ovog istraživanja, koje bi bilo zasnovano na predloženom rešenju i zaključcima do kojih se u ovom radu došlo.

7 Literatura

[1] http://developer.android.com/reference/android/graphics/Color.html April, 2014

[2] http://www.cambridgeincolour.com/tutorials/gamma-correction.htm June, 2014

[3] http://en.wikipedia.org/wiki/Gamma_correction, June, 2014

[4] http://en.wikipedia.org/wiki/HSL_and_HSV, May, 2014

[5] http://www.ncsu.edu/scivis/lessons/colormodels/color_models2.html, June, 2014

[6] http://en.wikipedia.org/wiki/Convolution, May, 2014

[7] http://www.jhlabs.com/ip/blurring.html, June, 2014

[8] http://docs.gimp.org/en/plug-in-convmatrix.html, May, 2014

[9] http://www.pixelstech.net/article/index.php?id=1353768112, June, 2014

[10] http://www.raf.edu.rs/Multimedija/Histogram_u_rasterskoj_grafici.htm, May, 2014

[11] http://en.wikipedia.org/wiki/Image_histogram, May, 2014

[12] http://developer.android.com/reference/android/view/SurfaceView.html, June, 2014

[13] http://developer.android.com/reference/android/view/SurfaceHolder.Callback.html, June, 2014

[14] http://en.wikipedia.org/wiki/YUV, June, 2014

[15] http://grepcode.com/snapshot/repo1.maven.org/maven2/net.sf.opencsv/opencsv/2.1, May, 2014

[16] http://android-graphview.org/, May, 2014