OpenCV biblioteka i primjene

Franić, Dino

SVEUČILIŠTE U ZAGREBU

PRIRODOSLOVNO–MATEMATIČKI FAKULTET

MATEMATIČKI ODSJEK

Dino Franić

OPENCV BIBLIOTEKA I PRIMJENE

Diplomski rad

Voditelj rada:
dr. sc. Goran Igaly

Zagreb, rujan, 2016

Ovaj diplomski rad obranjen je dana _____ pred ispitnim povjerenstvom u sastavu:

1. _____, predsjednik
2. _____, član
3. _____, član

Povjerenstvo je rad ocijenilo ocjenom _____.

Potpisi članova povjerenstva:

1. _____
2. _____
3. _____

Ovaj rad je plod dugotrajnog truda i putovanja prostranstvima računalnog vida. Posebno zahvaljujem dr. sc. Goranu Igalyju na pruženoj pomoći i podršci u radu. Zahvaljujem svojoj obitelji i djevojci na podršci i razumijevanju.

Sadržaj

Uvod

Open Source Computer Vision biblioteka predstavlja široki skup znanja o računalnom vidu u obliku implementiranih algoritama. Zapravo je rezultat konstantnog rada developera te se svakodnevno unaprjeđuje.
Računalni vid pronalazi široku primjenu, a napretkom tehnologija i informatizacijom postaje neizostavan dio u raznim područjima.

Kako bismo razumjeli računalni vid, prvo ćemo objasniti temeljne pojmove kao što su računalni prikaz slike, prostore boja i nastajanje same slike u kameri. Prikazat ćemo sadržaj OpenCV biblioteke te istražiti nekoliko područja. Pokušat ćemo prikazati temeljne matematičke ideje korištene za rješavanje raznih problema. Više pažnje ćemo posvetiti dijelu o prepoznavanju objekata jer ćemo se time poslužiti u praktičnom dijelu rada.

Koristeći mogućnosti biblioteke konstruirat i implementirat ćemo algoritam za praćenje otvorenog dlana i brojanje prstiju. Za tu potrebu ćemo trenirati nekoliko kaskadnih klasifikatora te prikazati njihove performanse.

Poglavlje 1

Pogled u problem

1.1 Ljudski vid

Prilikom percepcije okoline, kod ljudi je najzastupljenije osjetilo vida. Vidimo u vidljivom elektromagnetskom spektru[1]. U svakoj sekundi vidimo otprilike 60 novih slika i prema najnovijim istraživanjima, mozak je sposoban identificirati sadržaj slike za manje od 100 milisekundi. Slika kakvu doživljavamo nastaje iz dva izvora (oka) te tako nastaje stereoskopski vid, odnosno percepcija dubine. Uzmemo li u obzir rezoluciju ljudskog oka od 576 megapiksela[2][5] očito je da mozak mora obraditi veliku količinu podataka. Zanimljivo je, da pri tom, većinu vremena ne ulažemo svjestan napor i poimamo vid intuitivno.

Naizgled lagan problem, nama posve intuitivan, a zapravo vrlo složen. Samo funkcioniranje i anatomija oka su dobro poznati, no što se dalje događa u vizualnom korteksu nije posve jasno. Kognitivni procesi koji se odvijaju prilikom vizualne percepcije bivaju istraživani od strane mnogih stručnjaka, no holistički model još uvijek nije poznat. Iz rezultata dosadašnjih istraživanja na čimpanzama, zaključeno je da se obrada slike izvodi na više nivoa. Naime, ulazna slika (iz oka) prolazi brojne transformacije prije samog prepoznavanja sadržaja. Istraživanja su također pokazala da u ljudskom vizualnom korteksu postoje stanice koje vrše svojevrsnu detekciju rubova. Time je zadatak računalnog vida i razumijevanje procesa koji se odvijaju u mozgu. Jer, ako bismo razvili algoritam koji ima približne performanse kao ljudski mozak, naš problem bio bi riješen.

[1] Otprilike od 400 *nm* do 700 *nm*.

[2] Jednog oka pri veličini kuta vidnog polja od 120 stupnjeva.

1.2 Računala

Kako bismo postigli što veću funkcionalnost računala, nastojimo im omogućiti obavljanje poslova koji iziskuju vizualnu percepciju okoline. Prva istraživanja započela su 60-tih i 70-tih godina 20. stoljeća. Analogijom bismo mogli usporediti oko s kamerom, kao ulaznom jedinicom računala, a algoritme za obradu i analizu slike, s dijelovima mozga zaduženim za vizualnu percepciju. Time dolazimo do područja računalnog vida, pa ćemo navesti karakterizaciju tog pojma. Računalni vid je automatska analiza slike ili video sadržaja koju izvodi računalo kako bi postiglo razumijevanje nekog dijela okoline.[6] Na primjer želimo brojati promet biciklističkom stazom sa stacinoarnom kamerom. Dakle, računalo treba analizirati scenu (oduzeti pozadinu, uočiti objekte koji se gibaju. . .), izdvojiti dijelove slike na kojima postoji mogućnost da se nalazi biciklist (npr. objekt koji nije na pozadini), a zatim odrediti da li izdvojeno područje prikazuje biciklista.

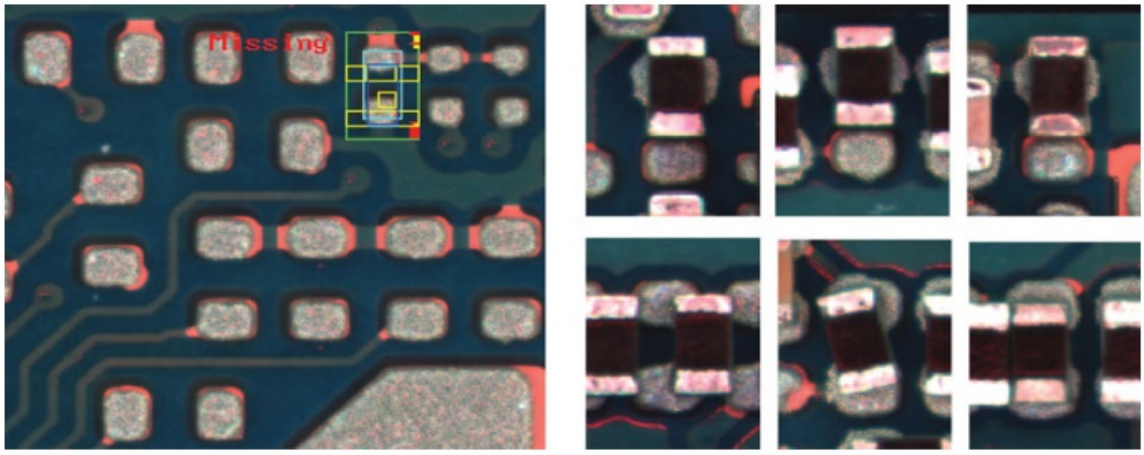
Računalo je automatski stroj koji vrši operacije nad podacima. Da bismo podatke unijeli u memoriju koristimo ulaznu jedinicu, odnosno više njih. Podatke često moramo na neki način smisleno obraditi da bismo ih spremili u memoriju računala i koristili za daljnju obradu. U najjednostavnijem slučaju moramo prepisati podatke, na primjer rezultate ispita. Uzmimo za primjer aplikaciju Photomath. Slikamo matematički problem, a aplikacija ponudi rješenje. Podatke, odnosno zadatak ne moramo prepisati, već ih u izvornom obliku unosimo u računalo putem kamere. Dakle, računalni vid nam pruža mogućnost da pojednostavimo prijenos informacija.

1.2.1 Primjena računalnog vida

U početku se zbog tehničkih ograničenja, računalni vid koristio primarno u industriji.

- Provjera razine tekućine u boci. Slika 1.1.
- Ispitivanje kvalitete naljepnice. Slika 1.1.
- Ispitivanje kvalitete lema na tiskanoj ploči. Slika 1.2
- Navođenje industrijskih robota.

Slika 1.1: Očitanje razine tekućine u boci (lijevo), provjera kvalitete ispisa roka trajanja (desno).

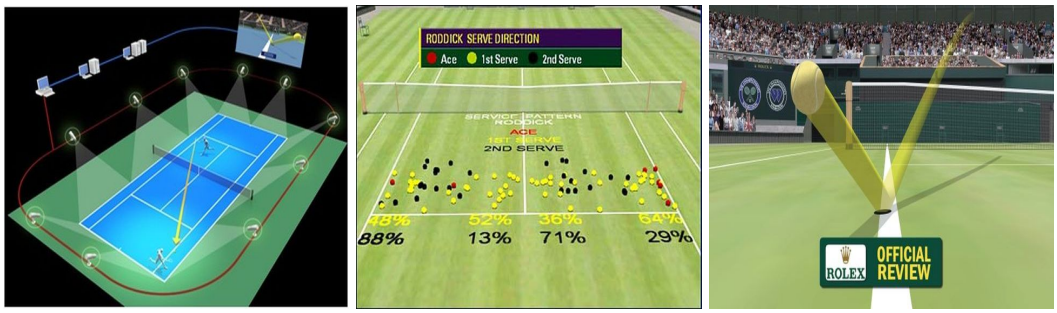


Slika 1.2: Provjera kvalitete lema.

Uočimo da je prethodnim primjerima zajedničko kontrolirana okolina tj. konstantno osvjetljenje i pozadina. Razvojem računalne tehnologije računalni vid pronalazi primjene i u mnogim drugim aspektima.

- Prepoznavanje znakova tijekom vožnje te upozorenje ukoliko automobil prilazi prepreci velikom brzinom (u odnosu na zaustavni put).
- Biometrijski sigurnosni sustavi. Na primjer očitavanje zjenice oka ili prepoznavanje cijelog lica.

- Predoperativni 3D modeli.
- Upravljanje gestama.
- Očitavanje registarskih oznaka na rampama.
- Potpomognuta stvarnost, virtualna stvarnost. Na primjer 3D rekonstrukcija objekta na temelju slika.
- Analiza sportskog prijenosa. Praćenje igrača (pretrčani kilometri, područje gibanja, dodiri s loptom), rekonstrukcija putanje lopte. . . .



Slika 1.3: Na teniskim terenima opremljenim Hawk-Eye sustavom igrači mogu zatražiti provjeru sudačke odluke. Algoritam analizom snimki generira virtualnu putanju lopte i pokazuje mjesto dodira s terenom. Također analizira druge aspekte igre i daje statističku analizu.

U relativno jednostavnim uvjetima, odnosno kada imamo zadan dio parametara, obrada računalnim vidom nije zahtjevna. Konstantna pozadina, osvjetljenje ili uzorak koji se konstantno ponavlja (kao tiskana ploča, razina tekućine u boci), smanjuju broj mogućih stanja i uvelike olakšavaju konstrukciju algoritma. Pogledajmo na primjeru provjere kvalitete lema na tiskanoj ploči.
Kamera je stacinarna, razina osvjetljenja je konstantna, a svaka tiskana ploča zauzima istu dio kadra. Boja lema i boja ploče su u pogledu spektra, dosta udaljene, pa vrlo lako možemo razlikovati pozadinu (tiskanu ploču) od lema. Oblik i veličina lema su također predefinirani, pa ih bez smanjenja općenitosti možemo promatrati kao pravokutnike. Stranice tih pravokutnika su paralelne s jednim rubom, odnosno okomite na drugi rub slike. Preostaje nam provjeriti da li je neki od lemova nepravilnog (nije pravokutnik) oblika, a zatim ukoliko zadovoljava oblik, provjeriti njegovu orijentaciju (stranice u odnosu na rubove slike). Ako znamo gdje bi se trebali nalaziti lemovi, jednostavnim postupkom možemo i to

provjeriti.

Prethodni algoritam bi za zadane uvjete postizao vrlo visoku točnost, no ukoliko bi došlo do odstupanja od zadanog formata, vrlo vjerojatno bi bio neučinkovit. Računalni vid u promjenjivoj okolini zahtijeva složeniji pristup. Da bismo ilustrirali, pogledajmo problem prebrojavanja biciklista sa stacionarnom kamerom.

Analogno kao kod provjere lemova, pozadinu je relativno jednostavno ukloniti. No što s objektima koji nisu pozadina? Kako ćemo zaključiti da li se radi o biciklistu? Oblik bicikla je relativno jednostavno prepoznatljiv, mogli bismo reći da se sastoji od dvije povezane kružnice. Možemo ga na taj način prepoznati, no što ako je u kadru više biciklista koji se djelomično preklapaju? Također moramo kroz cijeli kadar pratiti biciklista kako ga ne bismo više puta brojali. Ako bismo išli tim pristupom, odnosno da na temelju oblika identificiramo bicikl, morali bismo uzeti u obzir cijeli niz parametara. Kotači različitih bicikala ne moraju biti jednakih promjera, a ovise i o udaljenosti bicikliste od kamere, bicikli su različitih boja... Takav pristup bi bio vrlo nepraktičan, a funkcionalnost kao i točnost, upitna.

Računalni vid koristeći strojno učenje i umjetnu inteligenciju, doskače tom problemu. Ako želimo primijeniti računani vid na rješavanje problema u promjenjivoj okolini, strojno učenje postaje neizostavan dio. Iako predstavlja drugu granu računalstva kod računalnog vida predstavlja neizostavan alat za rješavanje općenitijih problema. Neke od algoritama ćemo prvo objasniti, a zatim i koristiti u poglavlju.

1.2.2 Budućnost računalnog vida

Zajednica developera konstantno pomiče granice mogućeg[6]. Napredak i rast performansi računala svakako doprinosi razvoju računalnog vida. Također su dostupni pametni telefoni, tableti i televizori čije performanse nadilaze performanse računala od prije desetak godina. Neki od njih nativno održavaju upravljanje gestama, kao što je listanje stranica pokretom dlana ili pomicanje po web stranici praćenjem zjenice. Razvijeni su prototipovi autonomnih automobila[16], koji su sposobni s velikom preciznošću navigirati i razumjeti okolinu. Javni prijevoz s autonomnim autobusima. Razvijene su razne primjene računalnog vida u medicini[4] te se nastoje razviti nove dijagnostičke metode. Virtualna i potpomognuta stvarnost nude nove mogućnosti u obučavanju ljudi, bez izlaganja opasnosti.
Slijedom vremena, računalni vid nastoji riješiti sve složenije probleme, odnosno one s općenitijim okolinama. Pojavljuju se robusnija rješenja i time omogućuju širu primjenu. Neke od očekivanih primjena su:[6]

- Postati integralni dio općih računalnih sučelja.

- Pružiti veću sigurnosnu razinu uz pomoć složenije biometrijske analize.
- Koristeći medicinske dijagnostičke snimke i medicinske kartone dati pouzdanu dijagnozu stanja.
- Omogućiti autonomna vozila.
- Automatski utvrditi identitet osobe sa slike ili video zapisa.
- Računalni vid u robotici.
- Primjena virtualne stvarnosti u edukacijske svrhe.

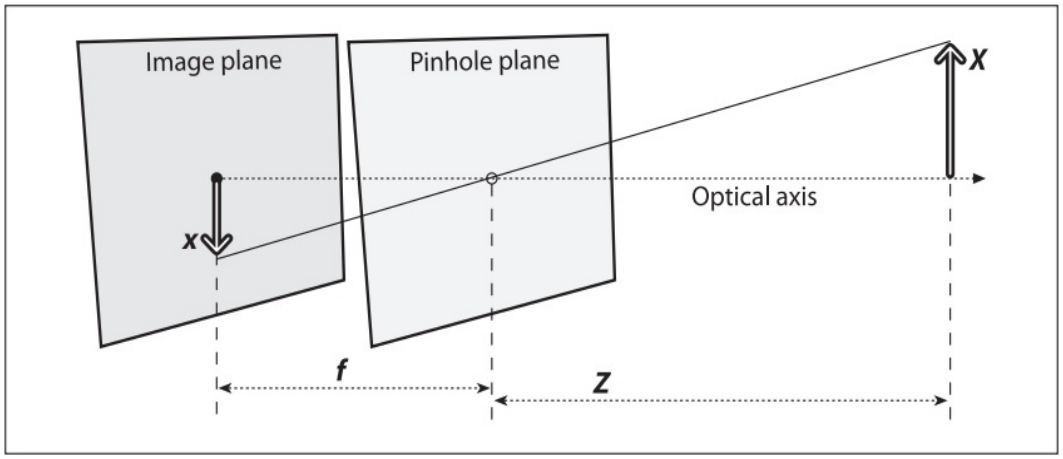
Poglavlje 2

Računalni prikaz slike

Svijet kojeg prikazujemo na slikama je najčešće trodimenzionalan - 3D, a slika je zapravo dvodimenzionalna - 2D ravnina. Predmete hipotetski možemo raščlaniti do atomske razine, no na slici ih moramo prikazati disjunktnim točkama. Kako uspijevamo prethodno, prikazat ćemo u sljedećem poglavlju

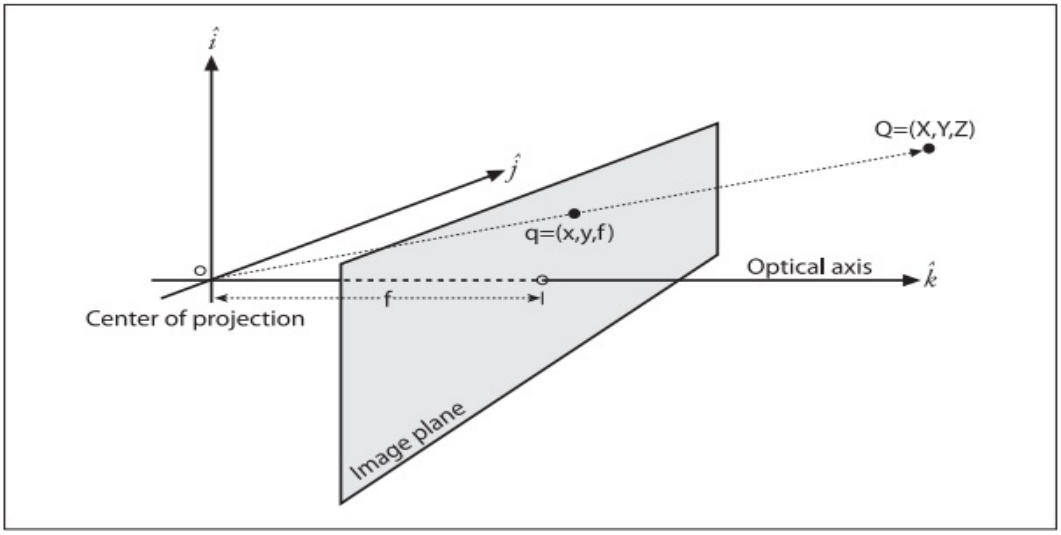
2.1 Kamera

Kamera se sastoji od fotoosjetljivog područja, leće na kućištu koja dopušta da određena svjetlost dopre do fotoosjetljivog područja i kućišta koje sprječava okolnu svjetlost. Leća fokusira ulaznu svjetlost na fotoosjetljivu površinu koju još nazivamo ravninom slike. Poslužit ćemo se jednostavnim modelom kamere, takozvanim *pinhole* modelom, kako bi prikazati nastajanje 2D slike iz 3D svijeta.



Slika 2.1: Prikaz *pinhole* modela. Zraka svjetlosti se odbija od objekta X, prolazi otvorom gdje siječe optičku os i projicira u ravnini slike. Orijentacija objekta i njegove projekcije je obrnuta, a njihov odnos veličina je dan žarišnom duljinom f.[2]

Pinhole model možemo interpretirati na način prikazan na slici 2.2.



Slika 2.2: Sjecište ravnine slike i zrake koja prolazi centrom projekcije i točkom Q, je projekcija točke Q. Ravnina slike je udaljena od centra projekcije za žarišnu duljinu f i okomita je na optičku os.[2]

Za točku $Q = (X, Y, Z)$, njenu projekciju $q = (x, y, 1)$ zadanu u homogenim koordina-

tama i žarišnu dužinu f vrijedi

$$x = f\frac{X}{Z}, y = f\frac{Y}{Z}. \tag{2.1}$$

Zbog nepreciznosti u proizvodnji, središte fotoosjetljivog senzora kamere nije na optičkoj osi. Koristeći parametre c_x i c_y modeliramo moguće odstupanje središta senzora od optičke osi.[2] Sada za koordinate x_{zaslon} i y_{zaslon} na zaslonu vrijedi:

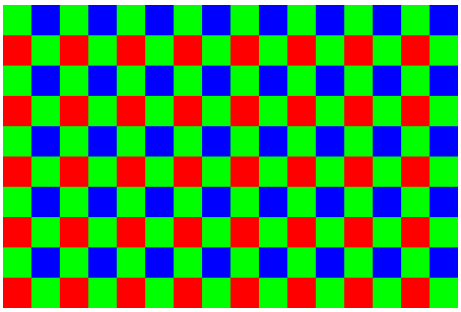
$$x_{zaslon} = f_x\frac{X}{Z} + c_x, y_{zaslon} = f_y\frac{Y}{Z} + c_y. \tag{2.2}$$

f_x i f_y su kombinacije fizičke žarišne duljine F i veličine piksela na senzoru u smjeru x i y.

Projekciju točke $Q = (X, Y, Z)$ u 3D svijetu u točku $q = (x, y, w)$ na slici možemo prikazati na sljedeći način:

$$\begin{bmatrix} x \\ y \\ w \end{bmatrix} = \begin{bmatrix} f_x & 0 & c_x \\ 0 & f_y & c_y \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} X \\ Y \\ Z \end{bmatrix}. \tag{2.3}$$

Koeficijent w nazivamo i skalirajućim faktorom. Završna slika se dobiva nakon potpune kalibracije s obzirom na tehničke nesvršenosti kao nepravilnosti u leći.[8]



Slika 2.3: Senzor kamere je najčešće konstruiran kao na slici. Naziva se Bayerov uzorak. Svaki piksel slike nema sve tri vrste fotoosjetljivih područja. Boje piksela koje nedostaju dobivaju se interpolacijom uz pomoć vrijednosti susjednih piksela.[14]

Uočimo da ima najviše senzora osjetljivih na zeleni spektar. Zašto je tako, vidjet ćemo u dijelu o prostorima boja.

2.2 Pikseli

Slika je zapravo neprekidna funkcija dviju varijabli, odnosno koordinata (x, y) u ravnini slike. S obzirom da sliku prikazujemo u memoriji računala, moramo ju razložiti na dis-

kretne elemente. Prikazujemo ju kao matricu s m stupaca i n redaka. Elemente matrice nazivamo pikselima.

Svaki piksel sadrži informaciju o akromatskoj svjetlini te točke na slici, također može sadržavati i informaciju o kromatskoj svjetlini. Akromatsku svjetlost doživljavamo kao crnu, bijelu i svu, odnosno nijanse od bijele do crne. Vrijednost piksela akromatske slike zapravo predstavlja količinu svjetlosti u toj točki. Tu vrijednost često nazivamo i sjajnost[1].

Količinu piksela na slici nazivamo razlučivost slike. Zasloni na kojima promatramo, a i same slike su pravokutnog oblika, pa ukupan broj piksela definiramo kao umnožak broja redaka i stupaca, na zaslonu, odnosno slici. Svaki piksel predstavlja jednu diskretnu točku, pa više piksela daje bolju razlučivost slike, odnosno više detalja. Veća razlučivost povlači veću količinu podataka što rezultira duljim vremenom potrebnim za obradu slike. Mogli bismo reći da broj piksela ograničava broj objekata koje možemo razlikovati. Kod računalnog vida cilj nam je kraće vrijeme obrade slike, tako ćemo u većini slučajeva razlučivost slike postaviti na najmanju moguću vrijednost pazeći da ne izgubimo potrebne detalje.

Slika 2.4: S lijeve strane je slika razlučivosti $1920x1080$ te izdvojeni dio slike. Desno je slika razlučivosti $640x400$ i izdvojeni dio te slike. Pri nižoj razlučivosti, natpis na zgradi je skoro nečitljiv, dok je pri višoj razlučivosti posve jasan.

Uočimo kod slike 2.4, da je pri manjoj razlučivosti slika dovoljno oštra,,da možemo izdvojiti skoro sve objekte, čak i veliki natpis na vratima. Prema tome, niža rezolucija nam

[1]Na engleskom jeziku se koristi riječ *brightness*

je dovoljna ukoliko bismo htjeli izdvojiti konture zgrada, no ako bismo htjeli čitati natpise sa zgrada tada bi viša razlučivost bila povoljnija.

Piksel (u memoriji računala) predstavlja strukturu podataka koja opisuje njegov prikaz na zaslonu. Implementacija u memoriji računala ovisi o potrebama, no postoji nekoliko čestih standarda.

2.3 Akromatska slika

Prisjetimo se da kod akromatske slike svaki piksel sadrži samo informaciju o sjajnosti točke na slici. Ovisno o broju bitova koje koristimo za zapis piksela u memoriji, moći ćemo postići određeni broj nijansi. Najčešće koristimo (i koristit ćemo u radu s OpenCV bibliotekom) 8-bitni zapis koji nam daje mogućnost prikaza u 256 različitih nijansi[2], odnosno intenziteta.

Slika 2.5: Izvorna slika (desno), nakon transformacije u sivu sliku (lijevo).

Takvu sliku često nazivamo i siva slika. Često ćemo slike iz drugih prostora boja transformirati u sivu sliku, a to radimo iz sljedećih razloga[6]

- Ljudi razumiju sivu sliku.
- Siva slika zauzima manje memorije i jednostavnija je za rad.

2.4 Slika u boji

Boje sadrže određenu informaciju i mogu pomoći u rješavanju određenih problema. Njihova uporaba će ovisiti o problemu kojeg želimo riješiti, prisjetimo se provjere kvalitete

[2] $2^8 = 256$

lema i prebrojavanja biciklista. Kod prvog boja nam omogućuje segmentaciju slike, odnosno odvajanje bitnog (lem) od nebitnog (pozadina), dok kod drugog problema nema nikakav utjecaj.

2.4.1 Red-Green-Blue (RGB)

RGB je jedan od najraširenijih standarda. Određena boja se stvara dodavanjem primarnih boja crvene (R), zelene (G) i plave (B). Koristi za prikaz piksela na zaslonu[3] i u rasterskoj grafici. Svaka komponenta, odnosno boja je zadana s 8 bitova, pa je ukupan broj mogućih boja $256 * 256 * 256 \approx 16.8$ miliona. Bijela boja nastaje dodavanjem maksimalnih vrijednosti za sve tri komponente, pa se ovakav model naziva i aditivnim modelom.

Pretvorbu iz RGB modela u sivu sliku vršimo na sljedeći način: Neka su $r_{(i,j)}, g_{(i,j)}, b_{(i,j)}$ vrijednosti, redom R, G, B komponenti piksela (i, j), tada je vrijednost piksela $s_{(i,j)}$ na sivoj slici zadana s jednakošću

$$s_{(i,j)} = 0.299 * r_{(i,j)} + 0.587 * g_{(i,j)} + 0.114 * b_{(i,j)}. \tag{2.4}$$

Prisjetimo se da je vrijednost piksela kod sive slike zapravo količina sjajnosti, a iz RGB slike smo ju dobili prema jednakosti 2.4. Uočimo da pri pretvorbi R, G, B komponente pridonose ukupnoj sjajnosti množenjem sa konstantnim koeficijentom. Prema tome, sjajnost piksela je dijelom sadržana u svakoj komponenti RGB modela. Takva raspodjela sjajnosti nam otežava segmentaciju slike jer su boje vrlo podložne promjenama u osvjetljenju. Količina određenih senzora u Bayerovom uzorku je proporcionalna doprinosu određene boje ukupnoj sjajnosti.

[3] Svaki piksel zaslona sadrži crvenu, zelenu i plavu komponentu.

Slika 2.6: Slika (gore) prikazana u RGB prostoru boja.

Problem takve segmentacije rješava sljedeći model.

2.4.2 Hue-Luminance-Saturation (HLS)

Ovaj model se vrlo često koristi u radu s računalnim vidom. Omogućuje razdvajanje akromatske od kromatske svjetlosti s kojima možemo točnije opisati traženu boju. Sjajnost je zadana s komponentom *Luminance*, a boja je određena s komponentama *Hue* i *Saturation*. *Hue* opisuje boju, a *Saturation zasićenost*, odnosno čistoću boje. Postoji i nekoliko sličnih modela HSB (*Hue, Saturation, Brightness*), HSV (*Hue, Saturation, Value*). Sama implementacija nije standardizirana i ovisi o programskom okruženju.

Slika 2.7: Komponente HLS spektra prikazujemo intenzitetom. Lijevo je komponenta H, u sredini L i desno S.

Ako normaliziramo (između 0.0 i 1.0) vrijednosti komponenti RGB modela, tada pretvorbu iz RGB u HLS model provodimo po formulama:

$$L = \frac{Max(R,G,B) + Min(R,G,B)}{2} \tag{2.5}$$

$$S = \begin{cases} \dfrac{Max(R,G,B) - Min(R,G,B)}{Max(R,G,B) + Min(R,G,B)}, & \text{za } L < 0.5 \\ \dfrac{Max(R,G,B) - Min(R,G,B)}{2 - (Max(R,G,B) + Min(R,G,B))}, & \text{za } L \geq 0.5 \end{cases} \tag{2.6}$$

$$H = \begin{cases} \dfrac{60 * (G - B)}{S}, & \text{ako je } R = Max(R, G, B) \\ \dfrac{120 + 60 * (B - R)}{S}, & \text{ako je } G = Max(R, G, B) \\ \dfrac{240 + 60 * (R - G)}{S}, & \text{ako je } B = Max(R, G, B) \end{cases} \tag{2.7}$$

ako je $H < 0$ tada uzimamo $H = H + 360$ zbog periodičnosti.
Lako vidimo da su vrijednosti parametara:

$$0 \leq V \leq 1.0,$$
$$0 \leq S \leq 1.0,$$
$$0 \leq H \leq 360.$$

2.4.3 Cyan-Magenta-Yellow

Temelji se na sekundarnim bojama cijan - C, magenta - M i žuta - Y. Od RGB modela se razlikuje i po načinu dobivanja bijele boje. Komponente se zapravo oduzimaju od bijele boje kako bi se dobila željena boja. Takav model je pogodan za ispisivanje (zbog bijele boje papira), pa se koristi kod pisača. Transformacija iz RGB modela zadana je s:

$$H = \begin{cases} C = 255 - R, \\ M = 255 - G, \\ Y = 255 - B \end{cases} \tag{2.8}$$

2.4.4 Binarna slika

U računalnom vidu često provjeravamo samo prisustvo nekog svojstva ili objekta. U tom slučaju nam čak nije niti bitna svjetlina točke, pa sliku možemo reprezentirati kao binarnu sliku[4]. Svi pikseli slike imaju vrijednost 0 ili 1. Najjednostavniji primjer je ako uzmemo sivu sliku X i neku graničnu vrijednost g te tražimo sve piksele čija je vrijednost veća od g. Binarna slika B je tada zadana s:

$$\forall (i, j) \in X, B(i, j) = \begin{cases} 1 & \text{za} \quad X(i, j) < g \\ 0 & \text{za} \quad X(i, j) \geq g \end{cases} . \tag{2.9}$$

Prethodna metoda se naziva *tresholding*. Binarna slika se koristi i za prikaz rubova na slici, svaki piksel slike koji je dio ruba ima vrijednost 1, a pikseli koji nisu na rubu imaju vrijednost 0.

Koristeći binarnu sliku, raznim tehnikama možemo manipulirati pikselima kako bismo izdvojili određena svojstva.

[4]Takvu sliku možemo smatrati crno-bijelom.

Slika 2.8: Siva slika (desno) i binarna slika (lijevo) nakon *tresholding*-a s graničnom vrijednošću 150.

Poglavlje 3

OpenCV biblioteka

OpenCV - *Open Source Computer Vision Library* je biblioteka otvorenog koda s BSD licencom, odnosno besplatna je za akademske (obrazovne) i komercijalne svrhe. Sadrži sučelja za rad u programskim okruženjima C, C++, Python, Java i MATLAB, te podržava Windows, OS X, Linux, iOS, Androd i BlackBerry operacijske sustave. Sadrži optimizirane implementacije 2500 algoritama namijenjenih računalnom vidu, što uključuje i algoritme strojnog učenja. Dizajnirana je s naglaskom na računarsku učinkovitost i rad u realnom vremenu. Napisana je i optimizirana u C/C++ okruženju te uz pomoć OpenCL-a omogućuje rad na heterogenim sustavima.[15] Glavni moduli biblioteke su prikazani u tablici 3.1, a sadrži i pomoćne module poput FLANN-a[1].

[1] *Fast Library for Approximative Nearest Neighbors*. Služi za brže pretraživanje višedimenzionalnih prostora značajki.

Tablica 3.1: Moduli sadržani u OpenCV biblioteci

Modul	Sadržaj
Core	Osnovne strukture i tipovi podataka, te upravljanje memorijom
Calib3d	Kalibracija kamere i 3D rekonstrukcija iz više gledišta
Highgui	Grafičko sučelje te učitavanje i spremanje slika i video zapisa
Imgproc	Filtriranje slika, geometrijske transformacije slike i analiza oblika
Video	Analiza pokreta i praćenje objekta
Features2d	Rad sa značajkama
Objdetect	Kaskadni klasifikator
GPU	Paralelizacija određenih algoritama u svrhu bržeg izvođenja na GPU jedinici
Ml	Algoritmi za strojno učenje kao na primjer SVM i neuralne mreže
Nonfree	Algoritmi SIFT i SURF
Ocl	Modul za paralelizaciju na uređajima koji podržavaju OpenCL
Flann	Optimizirani algoritmi za pretrage u višedimenzionalnim prostorima

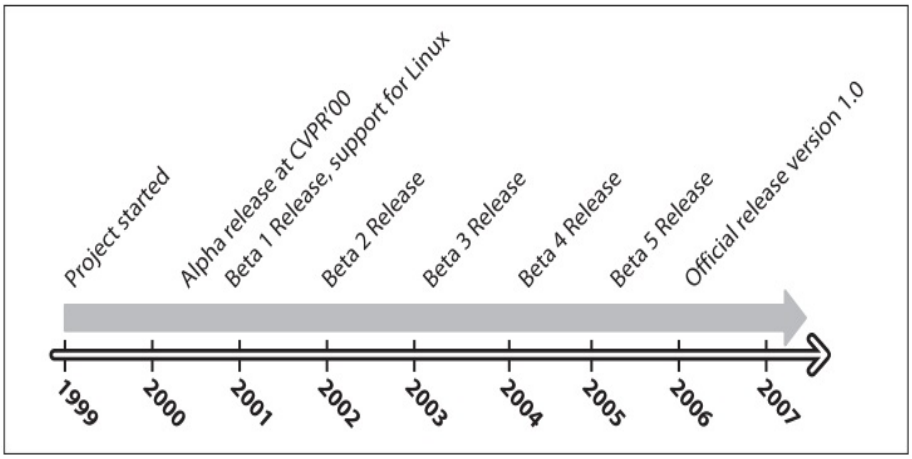
3.1 Povijest

OpenCV projekt izvorno je razvijen u Intelovom istraživačkom centru u Nižnji Novgorodu (Rusija). Prva alpha verzije izdana je 2000. godine, a u razdoblju do 2005. godine izlazi pet beta verzija. Verzija 1.0 izdana je 2006. godine, a tri godine kasnije OpenCV2 s mnogim poboljšanjima. Polovicom 2015. godine je izdana aktualna verzija 3.0.

Od 2012. godine podršku za OpenCV preuzima neprofitna organizacija OpenCV.org, a službena izdanja izdaje neovisni ruski tim u suradanji sa širokom zajednicom developera.[17]

3.2 Uvod u OpenCV

U ovom odjeljku ćemo prikazati osnovne objekte, te operacije s njima. Također ćemo se pozabaviti sučeljem, učitavanjem slike/video zapisa i pristupom kameri. Korištena verzija biblioteke je 2.4.9.



Slika 3.1: Vremenska crta prikazuje razvoj OpenCV biblioteke.

3.2.1 Klasa Mat

Već ranije smo opisali računalni prikaz slike, a sad ćemo vidjeti implementaciju u OpenCV biblioteci.
Za prikaz slike u memoriji koristimo klasu `Mat`, koji se sastoji od headera i podataka. S obzirom da je to osnovna klasa, posvetit ćemo joj nešto više prostora. Evo nekoliko načina na koje možemo stvoriti instancu klase `Mat`.

- `Mat()`
- `Mat(int rows, int cols, int type)`
- `Mat(int rows, int cols, int type, const Scalar& s)`

Parametri `rows` i `cols` određuju broj redaka, odnosno stupaca i možemo ih zadati u obliku jednog parametra `Size size(rows, cols)`. Vrijednosti parametra `int type` određuje količinu bitova i vrstu zapisa za kanal, te broj kanala. Zapisuje se u obliku `CV_{bitovi}{vrsta}{kanali}`. Broj bitova može biti 8, 16, 32 ili 64, a vrsta zapisa U-`unsigned integer`, S-`signed integer` i F-`floating-point number`. Broj kanala je proizvoljan i zadaje se u obliku `C{broj_kanala}`.
`s` je adresa memorijske lokacije strukture `Scalar`[2] čije će vrijednosti elemenata redom popuniti kanale matrice. Na primjer

```
1 Mat M(15, 10, CV_32FC3, Scalar(4,5,7))
```

[2] `Scalar` je zapravo restrikcija strukture `Vec` na maksimalno četiri elementa.

će stvoriti matricu 15 x 10 sa tri 32-bitna float kanala te će svi elementi prvog kanala biti ispunjeni sa 4, drugi sa 5, a treći sa 7.

Elementima matrice pristupamo na način prikazan u tablici 3.2.

Tablica 3.2: Pristup elementima strukture `Mat`

Sintaksa	**Opis**
`M.at<double>(i,j)`	Pristup elementu u i-tom retku j-tom stupcu. Brojanje počinje od 0
M.row(i)	Pristup i-tom retku
M.col(j)	Pristup j-tom retku
M.rowRange(i_1,i_2).colRange(j_1,j_2)	Pristup redcima od i_1 do i_2 i stupcima j_1 do j_2
	Možemo korstiti M.rowRange(i_1,i_2) ili M.colRange(j_1,j_2) za pristup samo redcima, odnosno stupcima

3.2.2 Slika, video, kamera

Slika

OpenCV nudi vrlo lagan i intuitivan način čitanja i pisanja slika, odnosno video zapisa. Podržava veliki broj formata kao što su BMP, JPEG, PNG, TIFF, AVI.
Slike učitavamo pomoću funkcije `imread(const string& filename, int flags=1)`. Ako je učitavanje uspješno, funkcija vraća objekt `Mat` koji sadrži sliku, a u suprotnom prazni objekt tipa `Mat`. Parametar `filename` predstavlja lokaciju i puni naziv slike, uključujući format. Drugim parametrom `flags` određujemo željeni način učitavanja slike, prikazanu tablici 3.3

Tablica 3.3: Parametri za učitavanje slike

Sintaksa	Opis
`CV_LOAD_IMAGE_ANYDEPTH`	Ako je slika 16-bitna/32-bitna, učitava ju u izvornoj dubini, inače pretvara u 8-bitnu
`CV_LOAD_IMAGE_COLOR`	Pretvara ulaznu sliku u sliku u boji
`CV_LOAD_IMAGE_GRAYSCALE`	Pretvara ulaznu sliku u sivu
> 0	Pretvara ulaznu sliku trokanalnu BGR sliku
$= 0$	Pretvara u sivu sliku
< 0	Učitava u izvornom obliku

Kod RGB slika, kanali su poredani u obrnutom redoslijedu, odnosno BGR.

Video, kamera i prozori

Slike učitavamo funkcijom, a video zapis pomoću objekta `VideoCapture`. Zbog nekoliko nužnih napomena, najbolje da pogledamo na primjeru pokretanja kamere.

```
#include "opencv2/opencv.hpp"

using namespace cv;

int main(int, char**)
{
  VideoCapture cap(0); // pokrece zadanu kameru
  if (!cap.isOpened())  // provjera da li je uspjesno pokrenuta kamera
    return -1;


  while (1) //petlja bez terminalnog uvjeta kako bismo mogli dohvacati nove slike kamere
  {
    Mat slika;
    cap >> slika; // uzmi novu sliku s kamere
    imshow("izlaz", slika); // na prozoru izlaz prikazi slika
    if (waitKey(1) == 27) break; //
  }

  return 0;
}
```

`waitKey()` je funkcija koja čeka ulazni prekid, potrebna je kako bi se ispravno inicijalizirala instanca klase VideoCapture. U ovom slučaju će se petlja prekinuti ako je pritisnuta tipka `escape`. Ako bismo htjeli učitati video zapis, umjesto argumenta `0` kod instance `cap`, klase `VideoCapture`, navodimo apsolutni put video zapisa. Na primjer `cap(''c:/video/zapis.avi'')`. Klasi `VideoCapture` možemo podešavati niz parametara, kao što su broj sličica u sekundi[3], širinu i visinu slike, zasićenost i mnoge druge.
Prozore stvaramo uz pomoć funkcije `nammedWindow` čiji su ulazni argumenti ime prozora i opcionalni argument dimenzije i omjera stranica prozora. Također postoji grafičko sučelje s klizačima i zadanim *callback* funkcijama.

[3] FPS

Poglavlje 4

Osnovna obrada slike

U ovom poglavlju ćemo obraditi osnovne manipulacije slikom. Prikazat ćemo kako ukloniti šum na slici i otkriti rubove na slici. Prisjetimo se da je slika funkcija dviju varijabli i, j, a reprezentiramo ju matricom. U računalnom vidu pri obradi slike koristimo konvoluciju matrice slike s konvolucijskom matricom operatora. Nazivamo ju konvolucijska maska.

4.1 Smetnje na slici

Poznatije kao šum na slici, najčešće se javljaju zbog ograničenja senzora kamere. Razlikujemo nekoliko vrsta šumova, a najčešći su Gaussov šum i zrnati šum[1].
Gaussov šum je zapravo dobra aproksimacija većine realnih šumova[6], dok je zrnati šum vrsta impulsnog šuma koji se očituje kao pikseli s velikim razlikama u svjetlini.

[1] Eng. *Salt and Pepper Noise*.

Slika 4.1: Izvorna slika (lijevo), Gaussov šum (desno) i zrnati šum (dolje).[6]

Šum nam uvelike otežava rad. Na primjer, želimo li dobiti rubove na slici, šum će uzrokovati lažni odaziv detektora rubova što će rezultirati nepreciznim i razlomljenim rubovima. Da bismo uklonili šum, koristimo nekoliko metoda zaglađivanja, ovisno o vrsti šuma.

4.1.1 Zaglađivanje prosječnom vrijednošću uzastopnih slika

Ovu metodu možemo koristiti ako promatramo istu scenu. Tada uzimamo težinski prosjek slika kako bismo eliminirali šum. Za n slika gdje je $f_k(i, j)$ vrijednost piksela (i, j) na slici f_k, vrijednost rezultirajućeg piksela $f(i, j)$ je dana s:

$$f(i, j) = \frac{1}{n} \sum_{k=1...n} f_k(i, j). \tag{4.1}$$

Vrijednost piksela $f_k(i, j)$ je zadana kao suma vrijednosti piksela $g_k(i, j)$ i šuma $v_k(i, j)$. Da bi ova metoda bila efektivna mora postojati statistička neovisnost šumova $v_k(i, j)$ i svaki od tih šumova mora imati Gaussovu razdiobu sa srednjom vrijednošću 0 i standardnom

devijacijom σ. U tom slučaju, ovom metodom se smanjuje standardna devijacija šuma za faktor $\sqrt{n}$. Zbog velike razlike u svjetlini od okolnih piksela, zrnati šum nije pogodan za ovu metodu.

Slika 4.2: Izvorna slika (lijevo), dodan Gaussov šum sa srednjom vrijednošću 0 i standardnom devijacijom 30 (sredina). Desno je slika nastala metodom prosječne vrijednosti na 8 slika sa dodanim Gaussovim šumom kao na slici u sredini.[6]

4.1.2 Lokalno zaglađivanje

Za dinamičnu scenu možemo koristiti lokalne metode eliminacije šuma. Za svaki piksel slike gledamo njegovu okolinu, odnosno susjedne piksele. Kako smo ranije naveli, postoje linearne i nelinearne metode zaglađivanja.

Linearno zaglađivanje

Dimenzija konvolucijske maske određuje veličinu okoline, a vrijednosti elemenata maske određuju težinske omjere pri sumiranju. Nova vrijednost piksela se dobiva na sljedeći način:

- za svaki piksel (i, j) izvorne slike:
- postavi konvolucijsku masku na sliku tako da je središnji element (i, j)
- pomnoži vrijednosti konvolucijske maske s vrijednostima piksela izvorne slike, te zatim zbroji umnoške
- nova vrijednost piksela (i, j) izvorne slike je jednaka sumi podijeljenoj sa zbrojem elemenata konvolucijske maske

Ako su elementi konvolucijske maske 1, tada govorimo o najjednostavnijem zaglađivanju kojeg često nazivamo i zamućivanje[2]. U OpenCV-u koristimo funkcije `blur()`. Na primjer:

```
1 blur(slika, zagladena_slika, Size(3,3));
```

će sliku `slika` zagladiti konvolucijskom maskom dimenzije 3x3 i spremiti kao `zagladena_slika`.

Jedano od najpoznatijih linernih zaglađivanja je Gaussovo zaglađivanje. Temelji se na tome da pikseli bliži promatranom pikselu doprinose većom težinom prilikom sumiranja. Koristi se diskretna aproksimacija Gaussove funkcije te se reprezentira maskom s cjelobrojnim vrijednostima.

$$G(x, y, \sigma) = \frac{1}{2\pi\sigma^2} e^{\frac{-(x^2-y^2)}{2\sigma^2}} \tag{4.2}$$

Slika 4.3: Dvodimenzinalni analogon Gaussove razdiobe sa varijancom σ^2.

Dimenziju maske određuje standardna devijacija, najčešće se gleda okolina do 3 standardne devijacije, no može se i zadati dimenzijom maske.

$$\frac{1}{273}\begin{bmatrix} 1 & 4 & 7 & 4 & 1 \\ 4 & 16 & 26 & 16 & 4 \\ 7 & 26 & 41 & 26 & 7 \\ 4 & 16 & 26 & 16 & 4 \\ 1 & 4 & 7 & 4 & 1 \end{bmatrix}$$

Slika 4.4: Konvolucijska matrica Gaussovog zaglađivanja dimenzije 5x5.

Napomenimo još da se prilikom konvolucije rubni pikseli promatraju kao specijalni slučajevi. U Opencv-u koristimo funkciju `GaussianBlur`. Naprimjer:

```
1 GaussianBlur(slika, zagladena_slika, Size(7,7),1.5);
```

gdje je zadnji parametar vrijednost σ Gaussove razdiobe. Gaussovo zaglađivanje također ne bi bilo pogodno za eliminaciju zrnatog šuma. Kako bismo riješili taj problem, koristimo nelinearno zaglađivanje

[2]Eng. *blurring*.

Nelinearno zaglađivanje

Jedan od najpoznatijih algoritama nelinearnog zaglađivanja, i dostupan u OpenCV-u, je temeljen na medijanu. Za svaki piksel se gleda okolina zadane veličine i traži srednja vrijednost, nakon toga promatrani piksel zamjenjujemo medijanom. Ovakvo zaglađivanje je pogodno za zrnati šum jer će nova vrijednost biti realna vrijednost nekog od susjednih piksela, za razliku od linearnih zaglađivanja gdje je vrijednost dobivena sumiranjem, između ostalog i piksela šuma s ekstremnom vrijednošću. Pogledajmo izdvojeni dio slike dimenzije 3x3 piksela.

18	16	13
37	69	66
92	99	88

Pretpostavimo da se dogodio zrnati šum na središnjem pikselu. Uzmimo da je vrijednost piksela šuma 255.

18	16	13
37	255	66
92	99	88

Koristeći medijansko zaglađivanje s maskom dimenzije 3x3 srednja vrijednost je 66. Gaussovo zaglađivanje dimezije maske 3x3 zadano je s :

$$\frac{1}{16}\begin{bmatrix}1 & 2 & 1\\ 2 & 4 & 2\\ 1 & 2 & 1\end{bmatrix}$$

zamjenjuje vrijednost središnjeg piksela sa 104. Dakle, odstupanje izvorne vrijednosti od rezultata medijanskim zaglađivanjem je 3, dok je kod Gaussovog zaglađivanja razlika 35. U OpenCV-u medijansko zaglađivanje pozivamo na sljedeći način:

```
medianBlur(slika, zagladena_slika, 3);
```

Maska je kvadratnog oblika pa se dimenzija zadaje jednim brojem, u ovom slučaju 3. Medijansko zaglađivanje ima i nepoželjne posljedice, oštećuje tanke linije i kuteve[6].Da bismo smanjili oštećivanje rubova možemo koristiti bilateralno zaglađivanje. Analogno kao Gaussov zaglađivanje, zadaje težine susjednim pikselima od promatranog, no težine su zadane s dvije komponente. Prva komponenta su težine s Gaussovom razdiobom, a druga komponenta uzima u obzir razliku u intenzitetu između središnjeg i okolnih piksela. U OpenCV-u koristimo funkciju:

```
bilateralFilter ( slika, slika_zagladena, d, sigma_b, sigma_k );
```

gdje je `d` promjer okoline svakog piksela, `sigma_b` standardna devijacija u prostoru boja, `sigma_k` standardna devijacija u prostoru koordinata piksela. Ukoliko je parametar `d` > 0

uzima se okolina zadana s tim parametrom, u suprotnom se promjer okoline piksela računa iz `sigma_k`.

4.1.3 Matematička morfologija

Morfološke operacije najčešće opisuju nelinearne operacije na slici. Provodimo ih uz pomoć nelinearnih maski. Primjenjujemo ju najčešće na binarnim slikama, pa se zapravo radi o binarnoj morfologiji. Opisat ćemo morfološke operacije dilatacije, erozije, morfološkog otvaranja i zatvaranja.

Definicija 4.1.1. *Za* $A, B \subset R^n$ *definiramo dilataciju kao* $A \oplus B = \{c \in R^n | c = a + b, a \in A, b \in B\}$.

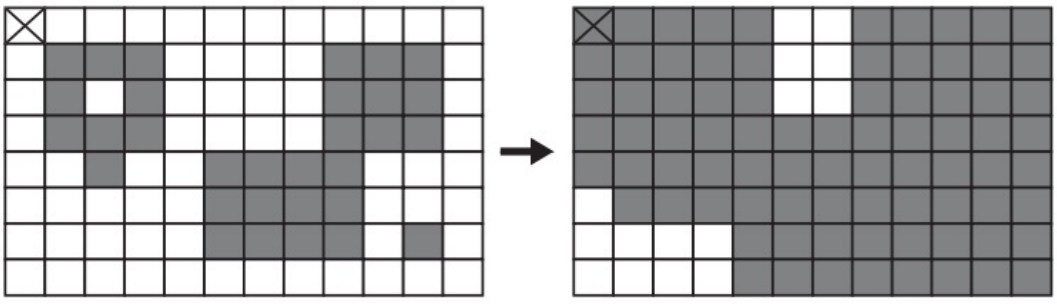
Definicija 4.1.2. *Za* $A, B \subset R^n$ *definiramo eroziju kao* $A \ominus B = \{c \in R^n | c + b \in A, \forall b \in B\}$.

U računalnom vidu A predstavlja dvodimenzionalnu binarnu sliku sa konačnim brojem piksela, a B strukturni element odnosno masku. Posljedica dilataciije je širenje broja piksela objekata, te nam tako pomaže u uklanjanju šupljina.

Da bismo u OpenCV-u izvršili dilataciju potreban nam je strukturni element, koji je zapravo konvolucijska maska. Predstavljamo ju strukturom `Mat`, a dobivamo ju kao povratnu vrijednost funkcije `getStructuringElement()`. Oblik može bit pravokutan, eliptični i u obliku križa. Na primjer:

```
1 Mat strukturni_element = getStructuringElement(MORP_RECT, Size(7,7),
    Point(-1,-1));
```

Stvara pravokutni strukturni element dimenzije 7x7 sa svim vrijednostima jednakim 1 i sidrišnom točkom u centru. `Point(-1,-1)` označava zadanu vrijednost prema kojoj je sidrišna točka u centru elementa.



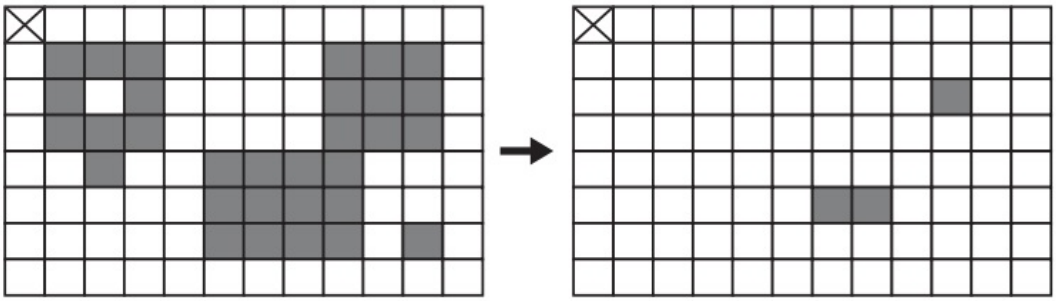
Slika 4.5: Dilatacija kvadratnim strukturnim elementom dimenzije 3x3.[6]

Funkcija dilatacije se poziva na sljedeći način:

```
dilate(binarna_slika, dilatirana_slika, strukturni_elment);
```

Ukoliko umjesto strukturnog elementa stavimo `Mat()` tada se stvara kvadratni strukturni element dimenzije 3x3.

Erozija uklanja manje nakupine piksela i smanjuje broj piksela objekata.



Slika 4.6: Erozija kvadratnim strukturnim elementom dimenzije 3x3.

Funkciju erozije pozivamo analogno kao funkciju dilatacije. Na primjer:

```
erode(binarna_slika, erodirana_slika, strukturni_elment);
```

Kombinirajući ove dvije operacije nastaju morfološko otvaranje i morfološko zatvaranje.

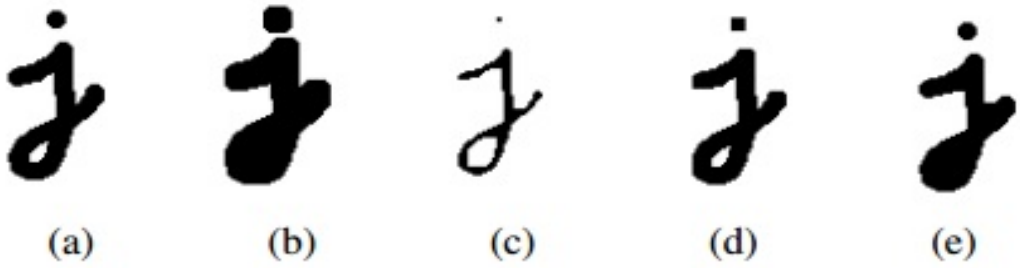
Definicija 4.1.3. *Morfološko otvaranje slike A strukturnim elementom B definiramo s* $A \circ B = (X \ominus B) \oplus B$.

Definicija 4.1.4. *Morfološko zatvaranje slike A strukturnim elementom B definiramo s* $A \bullet B = (X \oplus B) \ominus B$.

Otvaranje prvo eliminira elemente manje od strukturnog elementa, te izgladi rubove objekata, a zatim dilatacijom poveća površinu objekata tako da otprilike odgovara stvarnom objektu. Morfološko zatvaranje prvo popuni rupe i praznine manje od strukturnog elementa, a zatim erozijom ukloni manje nakupine piksela. Otvaranje i zatvaranje se pozivaju istom funkcijom `morphologyEx()`. Na primjer:

```
morphologyEx(binarna_slika, slika_otvaranje, MORPH_OPEN,
    struktrurni_element);
morphologyEx(binarna_slika, slika_zatvaranje, MORPH_CLOSE,
    struktrurni_element);
```

Često se prvo provodi morfološko zatvaranje kako bi se uklonile manje nepravilnosti, a zatim morfološko otvaranje.



Slika 4.7: Prikaz morfoloških operacija na ulaznoj slici s maskom dimenzije 5x5. (a) izvorna slika, (b) dilatacija, (c) erozija, (d) otvaranje, (e) zatvaranje.[14]

4.2 Rubovi i oblici

Što je zapravo rub? Promatramo li pažljivo sliku uočit ćemo da rub predstavlja promjenu intenziteta piksela. Slika je dvodimenzionalna pa promatramo promjenu u dva ortogonalna smjera i i j. Promjenu intenziteta piksela nazivamo gradijent, stoga računamo gradijent u smjeru i odnosno j. Slika je diferencijabilna funkcija dviju varijabli vrijedi:

$$\nabla f(i,j) = \left(\frac{\partial f(i,j)}{\partial i}, \frac{\partial f(i,j)}{\partial j}\right) = (f_i(i,j), f_j(i,j)). \tag{4.3}$$

Sada gradijent točke promatramo kao vektor, njegova duljina određuje magnitudu gradijenta, a orijentacija smjer gradijenta. Magnituda gradijenta u točki (i_0, j_0) na slici f zadana je s:

$$\nabla f(i_0,j_0) = \sqrt{f_i(i_0,j_0)^2 + f_j(i_0,j_0)^2}. \tag{4.4}$$

Ponekad se uzima samo zbroj apsolutnih vrijednosti parcijalnih derivacija zbog manjeg broja računalnih operacija. Orijentacija za se računa formulom 4.5

$$\phi(i_0,j_0) = \arctan\left(\frac{f_j(i_0,j_0)}{f_i(i_0,j_0)}\right) \tag{4.5}$$

Slika je reprezentirana matricom, pa za računanje parcijalnih derivacija koristimo konvolucijske maske.
Računajući gradijente za svaki piksel zapravo računamo prvu derivaciju slike. Detektore rubova koji interpretiraju maksimume prve derivacije slike kao rubove nazivamo detektorima rubova temeljenim na prvoj derivaciji.

4.2.1 Detektori temeljeni na prvoj derivaciji

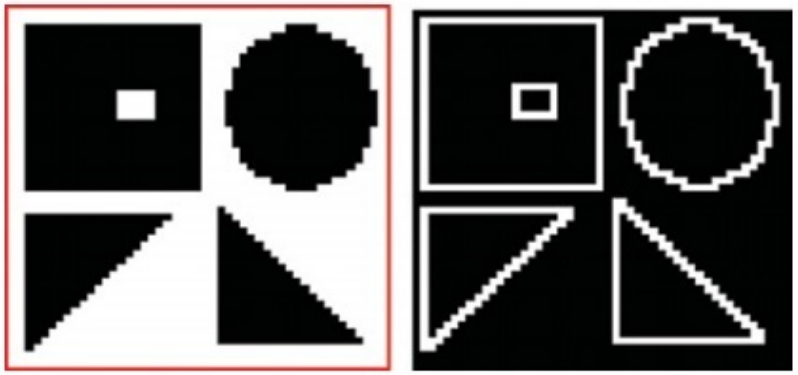
Spomenut ćemo Robertov detektor rubova i kompasne detektore rubova. Robertov detektor rubova za sliku f s vrijednostima piksela $f(i, j)$ se temelji na parcijalnim derivacijama

$$\partial_1(i, j) = f(i, j) - f(i + 1, j + 1) \quad \partial_2(i, j) = f(i, j) - f(i + 1, j - 1). \tag{4.6}$$

Prethodno možemo prikazati kao konvolucijske maske:

$$\partial_1(i, j) = \begin{bmatrix} 1 & 0 \\ 0 & -1 \end{bmatrix} \quad \partial_2(i, j) = \begin{bmatrix} 0 & 1 \\ -1 & 0 \end{bmatrix}. \tag{4.7}$$

Robertov detektor daje dobre rezultate na binarnoj slici, te je jedini od detektora temeljenih na prvoj derivaciji koji bi se trebao koristiti na binarnoj slici.[6]Robertov detektor uzima u obzir vrijednost dvaju piksela, pa zbog ima dosta lažno pozitivnog odaziva, a podložan je i šumu na slici.



(a)

(b)

Slika 4.8: Prikaz djelovanja Robertovog detektora rubova na binarnoj slici (a) i sivoj slici (b).

Ovaj detektor rubova uzrokuje pomak rubova od $\frac{1}{2}$ piksela.
Prewittov detektor i Sobelov detektor su najpoznatiji primjeri kompasnih detektora rubova. Svaki od njih ima osam parcijalnih derivacija, od kojih ćemo pokazati par ortogonalnih maski.

$$\partial_1(i,j) = \begin{bmatrix} 1 & 1 & 1 \\ 0 & 0 & 0 \\ -1 & -1 & -1 \end{bmatrix} \quad \partial_2(i,j) = \begin{bmatrix} -1 & 0 & 1 \\ -1 & 0 & 1 \\ -1 & 0 & 1 \end{bmatrix} \tag{4.8}$$

Slika 4.9: Prikaz maski ortogonalnih parcijalnih derivacija za Prewittow detektor rubova.

$$\partial_1(i,j) = \begin{bmatrix} 1 & 2 & 1 \\ 0 & 0 & 0 \\ -1 & -2 & -1 \end{bmatrix} \quad \partial_2(i,j) = \begin{bmatrix} -1 & 0 & 1 \\ -2 & 0 & 2 \\ -1 & 0 & 1 \end{bmatrix} \tag{4.9}$$

Slika 4.10: Prikaz maski ortogonalnih parcijalnih derivacija za Sobelov detektor.

Glavne razlike naspram Robertovog detektora su da kompasni detektori uzimaju veću okolinu, odnosno točke koje se testiraju su odvojene, te se kompasni detektori centriraju na točku, pa nema pomaka. Zbog šire okoline piksela, manje su osjetljivi na šum i imaju precizniji odaziv na sivoj slici.

U Opencv-u na sljedeći način možemo pomoću Sobelovog detektora rubova izračunati orijentaciju i gradijent kao normu, slike `siva_slika`:

```
Mat derivacija_x, derivacija_y, gradijent, orjentacija;

sobel(siva_slika, derivacija_x, CV_32F, 1, 0);
sobel(siva_slika, derivacija_y, CV_32F, 0, 1);

cartToPolar(derivacija_x, derivacija_y, gradijent, orijentacija)
```

Dobiveni gradijenti su raznih magnituda i rubovi su često prikazani s nekoliko piksela različitih magnituda. Cilj nam je što preciznije odrediti rub, kako bismo to postigli koristimo tehniku *non-maximum suppression*. Glavna ideja je da za svaki piksel pomoću orijentacije odredimo dva susjedna piksela sa istom orijentacijom i ukoliko je gradijent promatranog piksela manji od dva susjedna tada na njegovo mjesto upisujemo 0. Na kraju će ostati samo lokalni maksimumi, a rubovi će biti precizniji. *Non-maximum suppression* nije implementiran u OpenCV-u.

Slika 4.11: Slika u sivom području (lijevo), nakon detekcije svih rubova (sredina), poslije primjene non-maximum suppression.[6]

4.2.2 Detektori rubova temeljeni na drugoj derivaciji

Takvi detektori promatraju brzinu promjene magnitude gradijenta, odnosno drugu derivaciju slike, te traže gdje ona mijenja predznak. Neusmjerena druga derivacija slike f jednaka je:

$$\nabla^2 f = \frac{\partial^2 f}{\partial i^2} + \frac{\partial^2 f}{\partial j^2} \tag{4.10}$$

nazivamo ju i Laplaceovim operatorom. Diskretna aproksimacija konvolucijske maske dana je s:

$$h(i, j) = \begin{bmatrix} 0 & 1 & 0 \\ 1 & -4 & 1 \\ 0 & 1 & 0 \end{bmatrix}. \tag{4.11}$$

Učimo da je težina na središnjem pikselu nekoliko puta veća od susjednih, stoga je osjetljiv na šum na slici. Marr i Hildreth[3] su uz pomoć Gaussovog zaglađivanja i Laplaceove maske razvili algoritam za detekciju rubova poznat kao *Laplacian of Gaussian* ili *LoG*. Motivacija su pronašli u neuro-psihološkim eksperimentima koji su pokazali da ljudski vid vrši vrlo slične operacije.
Konvoluciju slike S Gaussovom maskom g možemo zapisati:

$$G = g * S \tag{4.12}$$

gdje je G zaglađena slika, a $*$ označava konvoluciju. Laplaceov operator označavamo s $\triangle^2$, pa možemo zapisati:

$$\triangle^2 G = \triangle^2 (g * S), \tag{4.13}$$

također vrijedi

$$\triangle^2 G = (\triangle^2 g) * S. \tag{4.14}$$

Možemo prvo Laplaceovim operatorom djelovati na Gaussovu masku i dobivenom maskom konvoluirati sliku, što smanjuje broj računalnih operacija. Također, Gaussova dvodimenzionalna maska može se razdvojiti na dvije jednodimenzionalne konvolucije.
Na dobivenoj slici traže se okoline oblika:

- $\{+, -\}$
- $\{+, 0, -\}$
- $\{-, +\}$
- $\{-, 0, +\}$

[3]David Marr - Britanski neuroznanstvenik i psihijatar. Ellen C. Hildreth - Američka profesorica računalne znanosti.

te okoline predstavljaju rubove. Za svaku okolinu s vrijednostima na $a, -b$ ili $-a, b$ se računa nagib po apsolutnoj vrijednosti te se rubovi filtriraju prema veličini nagiba. U OpenCV-u računamo *LoG* slike `siva_slika` na sljedeći način:

```
Mat laplac, gauss;
GaussianBlur(siva_slika, gauss, Size(7,7));
Laplacian(gauss, laplace, CV_32F, 3); //broj 3 oznacava velicinu maske
```

Marr i Hildreth su također predložili da bismo sliku mogli konvoluirati s Gaussovim maskama različitih σ vrijednostima[6] što je kasnije iskorišteno za stvaranje skala. Prednost *LoG*-a je da uzima širu okolinu, no ponekad zbog zaglađivanja gubi oštrinu rubova i kutova.

Cannyjev detektor rubova

Cannyjev detektor rubova koristi je dizajniran je da optimizira tri svojstva:

- Detekciju - da minimizira lažno pozitivnu i lažno negativnu detekciju.
- Lokalizaciju - udaljenost između detektiranog i stvarnog ruba treba biti što manja.
- Jedinstveni odaziv - za svaku točku stvarnog ruba detektor treba detektirati samo jednu točku.

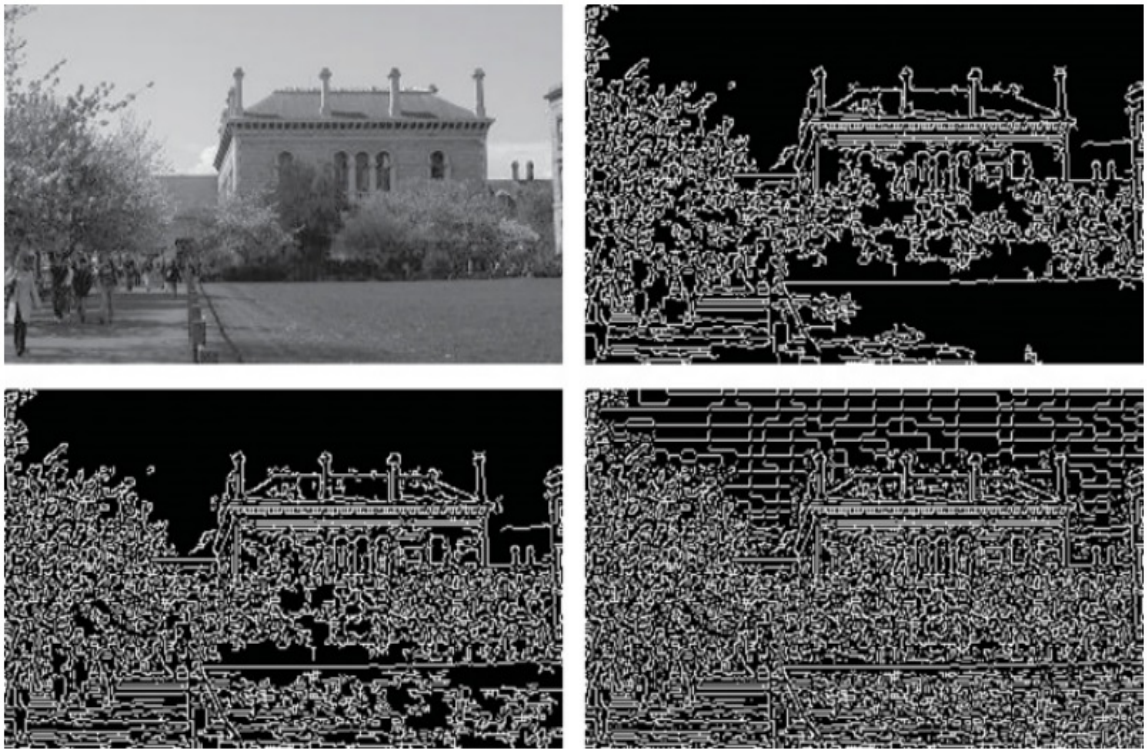
Algoritam se satoji od sljedećih koraka:

- Zagladi sliku Gaussovim zaglađivanjem.
- Odredi prvu derivaciju zaglađene slike.
- Na dobivenoj slici odredi orijentaciju i magnitudu gradijenta za svaku točku na slici.
- *non maximum suppression* za filtriranje rubova.
- Primjeni *hysteresis*. Koriste se dvije granične vrijednosti G i g za filtraciju točaka. Ako je magnituda gradijenta promatranog piksela veća od G tada taj piksel sigurno predstavlja rub. Pikseli čija magnituda gradijenta je manja od g se odbacuju. Ukoliko je magnituda gradijenta veća od g, a manja od G, piksel je dio ruba samo ako je povezan sa nekim pikselom čija je magnituda gradijenta veća od G. Time se ne gube dijelovi ruba s manjim intenzitetom.

Rezultat je binarna slika rubova. Cannyjev detektor rubova u OpenCV-u pozivamo na sljedeći način:

```
Mat rubovi;
int donja_granica =100;
int gornja_granica =200;
Canny(siva_slika, rubovi, donja_granica, gornja_granica);
```

.



Slika 4.12: Prikaz djelovanja Canny-jevog detektora rubova za razne granične vrijednosti. Gore desno - $g = 7$, $G = 236$, dolje lijevo - $g = 7$, $G = 100$, dolje desno - $g = 1$, $G = 1$

4.2.3 Konture

Nakon detektiranih rubova želimo odrediti konture na slici, kako bismo mogli segmentirati objekte. Rubovi su reprezentirani binarnom slikom, pa se konture pronalaze gledajući povezanost piksela. Kontura je reprezentirana vektorom točaka, a sve konture vektorom čiji elementi su vektori točaka. Hijerarhija kontura je zadana vektorom struktura `Vec4i`. Za svaku konturu može sadržavati informaciju o ugniježđenoj konturi, roditeljskoj konturi, te o idućoj i prethodnoj konturi na istom hijerarhijskom nivou. U OpenCV-u se za pronalazak kontura koristi funkcija `findContours()`. Postoji nekoliko tehnika za reprezentaciju kontura:

- `CV_RETR_EXTERNAL` prikazuje samo vanjske konture.
- `CV_RETR_LIST` prikazuje sve konture bez uspostave hijerarhije.
- `CV_RETR_CCOMP` prikazuje sve konture u dva hijerarhijska nivoa. Vanjske konture i konture unutar njih.
- `CV_RETR_TREE` prikazuje sve konture u potpunoj hijerarhiji.

Primjer traženja kontura za binarnu sliku rubova `bin_rubovi` u OpenCV-u:

```
1 vector<vector<Point>> konture;
2 vector<Vec4i> hijerarhija;
3 findContours(bin_rubovi, konture, hijerarhija, CV_RETR_TREE,
    CV_CHAIN_APPROX_NONE);
```

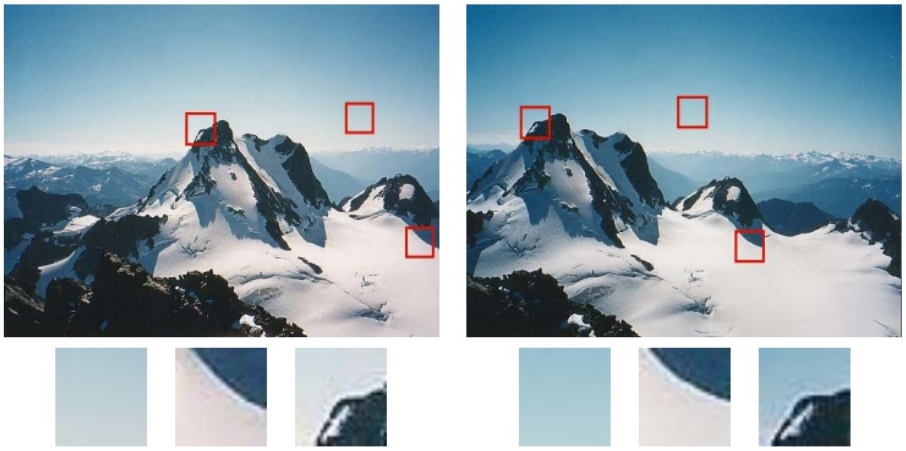
Dostupne su razne funkcije za analizu kontura prikazane u tablici 4.1.

Funkcija	**Opis**
`ArcLength()`	Pronalazi duljinu konture
`ContourArea()`	Izračunava površinu konture
`BoundingRect()`	Pronalazi najmanji uspravni pravokutnik koji sadrži konturu
`ConvexHull()`	Izračunava konveksnu ljusku konture
`IsContourConvex()`	Ispituje da li je kontura konveksna
`MinAreaRect()`	Pronalazi najmanji rotirani pravokutnik koji sadrži konturu
`MinEnclosingCircle()`	Pronalazi krug opisan konturi najmanjeg promjera
`FitLine()`	Pronalazi liniju koja aproksimira konturu

Tablica 4.1: Funkcije za analizu kontura.

4.2.4 Lokalizacija kutova

Želimo li u dvije uzastopne slike odrediti pomak ravnog ruba nije jednoznačno određeno koju točku ruba na prvoj slici ćemo povezati s točkom ruba na drugoj slici. Rješenje problema je da promatramo bolje lokalizirane točke takozvane točke interesa, na primjer kutove. Točke interesa mogu zapravo predstavljati bilo koje točke na slici koje se daju robusno lokalizirati. Nešto više o njima ćemo reći u idućem poglavlju.



Slika 4.13: Dijelovi slike s većim kontrastom u više smjerova daju se bolje lokalizirati. [14]

Kut ima svojstvo da je gradijent u lokalnoj okolini različitih smjerova ali intenzivne magnitude. Detektori kutova raznim tehnikama pronalaze takva mjesta na slikama. U OpenCV-u za detekciju kutova koristimo Harrisov detektor i `FAST` detektor.

Slika 4.14: Prikaz kutova dobivenih sa Harrisovim detektorom (desno) i FAST detektorom (lijevo).

Prikazat ćemo pozive detektora u OpenCV-u.

```
1 Mat siva_slika, izlaz;
```

```
2 vector<KeyPoint> kutovi; // vektor strukture KeyPoint u kojoj se
      pohranjuju pronadene tocke i informacije
3 cvtColor(slika, siva_slika, CV_BGR2GRAY);
4
5 FastFeatureDetector fast_detektor(70, true); // FAST detektor
6 fast_detektor.detect(siva_slika, kutovi);
7
8 /*GoodFeaturesToTrackDetector harris_detektor(1000, 0.01, 10, 3, true);
9 /harris_detector.detect(siva_slika, kutovi);*/  //za Harrisov detektor
10 drawKeypoints(slika, kutovi, izlaz, Scalar(0, 0, 255)); // kljucne tocke
      , a izmedu ostalog i kuteve, mozemo nacrtati
```

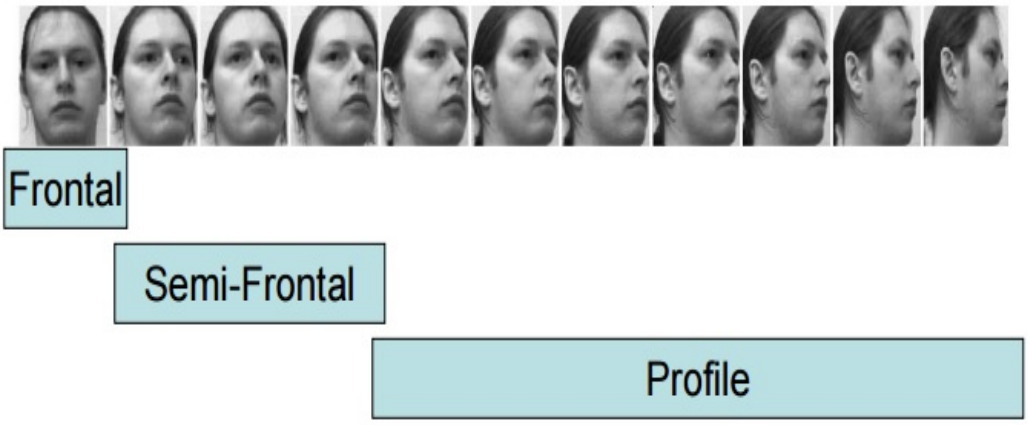
Poglavlje 5

Kako prepoznati objekt

Osnovni problem računalnog vida u današnje vrijeme je kako prepoznati traženi objekt (klasu objekata) na slici. Što zapravo znači prepoznati objekt?

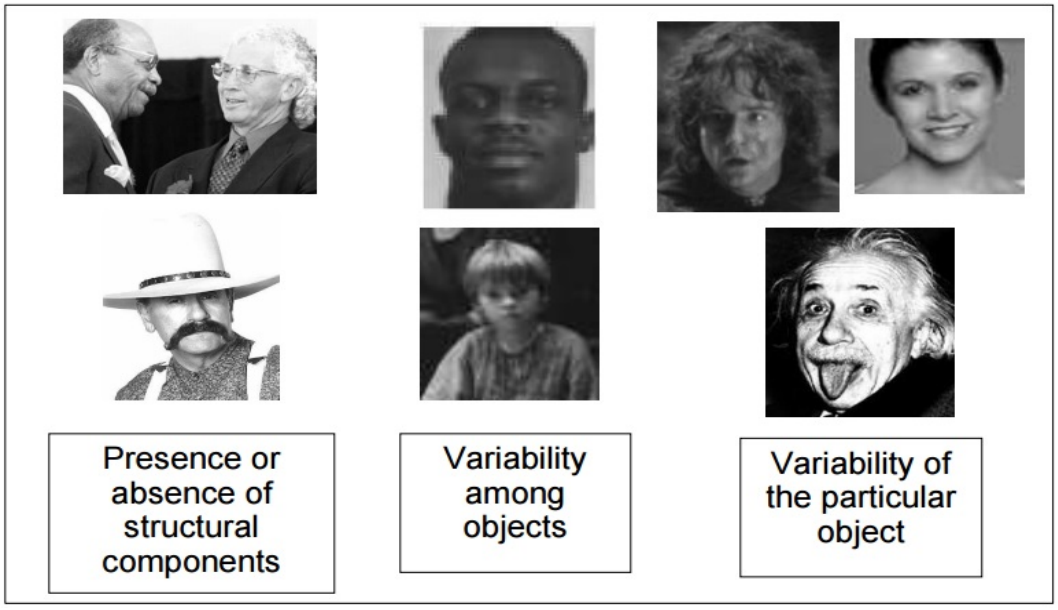
Definicija 5.0.1. *Kažemo da smo prepoznali zadani objekt (ako postoji) na zadanoj slici, ako smo mu odredili poziciju i skalu na toj slici.[12]*

Glavni problem prilikom prepoznavanja objekta je varijabilnost izgleda kako samog objekta tako i okoline.[1] Tako se u nekim slučajevima drugi kut gledanja (poza objekta) može razmatrati kao zaseban objekt (klasa).



Slika 5.1: Varijabilnost lica s obzirom na rotaciju izvan ravnine daje zapravo 3 klase objekata: frontalo lice, polu-frontalno lice i profilno lice [12]

[1] Uz pretpostavku da se može izvoditi u realnom vremenu.



Slika 5.2: Intrinzična varijabilnost lica [12]

Osim intrinzične varijabilnosti objekta, detekciju objekta nam otežava i varijabilnost okoline kao što je osvjetljenje, zaklonjenost objekta, skala objekta, kvaliteta kamere. Jasno je da prepoznavanje objekata predstavlja vrlo zahtjevan izazov. Da bi se doskočilo tom problemu, koriste se razne tehnike poput statističke klasifikacije i strojnog učenja. OpenCV nudi mogućnost rada s kaskadnim klasifikatorima i ključnim točkama, pa ćemo te dvije tehnike objasniti.

5.1 Klasifikator

Klasifikatori na temelju trening podataka određuju kojoj klasi pripada promatrani objekt, pa tako u prepoznavanju objekata čine temeljnu ulogu. Često se temelje na strojnom učenju.

5.1.1 Statistička klasifikacija

Sastavljajući slagalice, ljudi uzimaju dio i uspoređujući ga sa cijelom slikom, pronalaze moguće mjesto. Često se i dok tražimo neki nepoznati objekt poslužimo njegovom fotografijom kako bismo ga lakše pronašli. Takav način prepoznavanja, odnosno traženja

objekta nazivamo uzorkovanje[2] i koristi se u računalnom vidu kao najjednostavniji oblik prepoznavanja.

Algoritam uzorkovanja[6] ima sljedeće korake:

- Za svako moguće mjesto (i, j) uzorka u izvornoj slici:

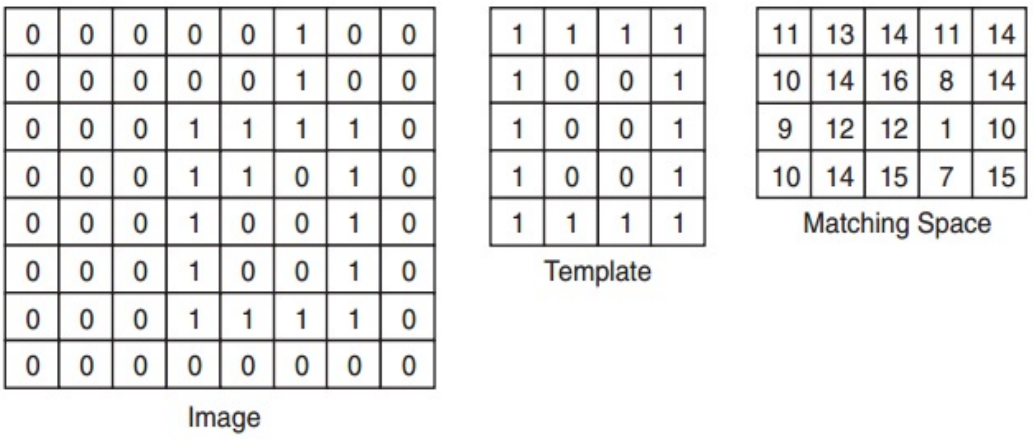
  Izračunaj podudarnost i spremi vrijednost u prostor podudaranja[3].

- Nađi lokalne maksimume (ili minimume) u prostoru podudaranja čija je vrijednost veća od granične vrijednosti[4].

Za računanje podudarnosti možemo koristiti sumu kvadrata razlika.

$$D_{KvadratnaRazlika}(i, j) = \sum_{(m,n)} (M_{i+m,j+n} - T_{m,n})^2 \tag{5.1}$$

gdje je $M_{i,j}$ matrični prikaz izvorne slike, a $T_{m,n}$ matrični prikaz uzorka. Zapravo razmatramo Euklidsku udaljenost slika. Jasno je da manja vrijednost sume, pokazuje veću podudarnost.



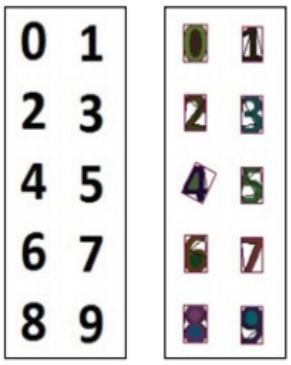
Slika 5.3: Izvorna slika i uzorak uspoređeni koristeći kvadrat razlike

Objekte možemo opisati po fizičkim svojstvima kao što su dužina, visina, površina, duljina ruba. Pomoću OpenCV-a možemo izračunati prethodna svojstva i još nekoliko njih Najčešće korištena svojstva su:

[2]Eng. *Template matching*
[3]Eng. *Matching Space*
[4]Eng. *Threshold*

- Konture i svojstva prikazana u tablici 4.1.
- Hu Moments - mjere distribuciju oblika uz pomoć težinskih prosjeka. Neki od njih su invarijantni na skaliranje i rotaciju.

U daljnjem tekstu ćemo umjesto izraza svojstva koristiti izraz značajke. Svojstvima često nazivamo osobine koje su primarno uočljive kao površina, konture i slično, no u daljnjim radovima postaju apstraktne pa ćemo koristiti izraz značajke. Računajući značajke za neki objekt, možemo ih zapisati kao uređenu n-torku $(x_1, x_2, ..., x_n)$. Takav zapis značajki razmatramo kao opisnik značajki[5] nekog objekta.
Prije nego krenemo dalje, uočimo da opisnik značajki možemo interpretirati kao vektor. Nadalje, možemo i konstruirati prostor značajki[6] kojeg razapinju vektori značajki. U dijelu o klasifikatorima ćemo spomenuti Haar i LBP značajke jer ih koristimo kod rada sa kaskadnim klasifikatorom u Opencv-u. Kasnije ćemo govoriti o drugim oblicima značajki.



Shape	Perimeter	Area	Bounding Box	Convex Hull	Hu Moments	Hole Areas
0	80	406	651	555	0.17 0.00 0.00	175
1	101	274	522	429	0.47 0.13 0.01	
2	111	347	570	528	0.42 0.07 0.00	
3	121	351	522	530	0.40 0.06 0.00	
4	85	347	592	442	0.20 0.01 0.00	60
5	122	351	570	504	0.43 0.06 0.00	
6	102	392	620	534	0.23 0.01 0.00	76
7	94	267	551	394	0.46 0.09 0.04	
8	65	438	651	571	0.20 0.01 0.00	45 62
9	101	389	589	524	0.23 0.01 0.00	74

Slika 5.4: Tablica prikazuje neke značajke za brojeve (lijevo).[6]

Značajke objekata i vjerojatnost pojave određenih značajki za neki objekt čine temelj statističkog prepoznavanja objekata[6]. Za klasu objekata W i svojstvo x računamo uvjetnu vjerojatnost, odnosno vjerojatnost da se uz značajku x pojavio objekt klase W. Zapravo želimo odrediti funkciju gustoće vjerojatnosti značajke x za klase W_i.

Zadan je skup $S = (x_i, y_i)_{i=1,...,m}$ primjera za vježbu s nepoznatom vjerojatnosnom distribucijom P. Uz pomoć tog skupa pronalazimo funkciju f koja predviđa vrijednost od y kao $f(x)$.

[5]Eng. *Feature descritor*
[6]Eng. *Feature space*

Cilj nam je pronaći funkciju f (klasifikator) takva da je

$$Vjer_{(x,y)\sim P}[f(x) \neq y]$$

vrlo malena.[12] Razlikujemo dvije vrste pogrešnog klasificiranja:

- Greška prilikom treninga $Trening\_greska = Vjer_{(x,y)\sim S}[f(x) \neq y]$. Zapravo krivo klasificirani testni primjer.

- Greška prilikom testa (rada) $Test\_greska = Vjer_{(x,y)\sim P}[f(x) \neq y]$. Krivo klasificirani općeniti objekt

Cilj nam je smanjiti vjerojatnost krivo klasificiranog objekta prilikom rada.

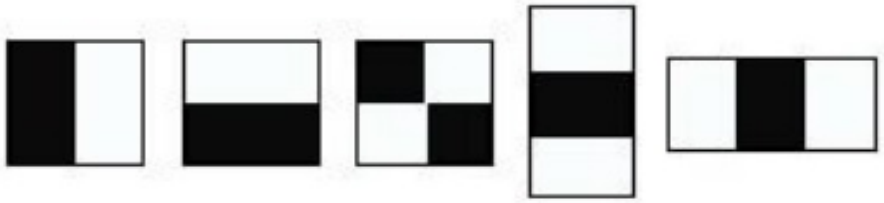
Kako bismo što preciznije odredili funkciju gustoće, potreban nam je kvalitetan skup objekata koji nam služe za trening. Loša konstrukcija [7] skupa za trening uzrokuje veće greške. Veličina skupa se ne može unaprijed odrediti, već proširujemo skup dok ne dobijemo funkciju sa željenom preciznošću. Osim objekata, ključnu ulogu čini algoritam za detekciju i opis značajki. Objekte želimo prepoznati u raznim scenama, položajima i skalama, pa značajke moraju biti robusne i jednoznačno određene. Također želimo primjenjivost algoritma na razne objekte, stoga bi bilo izrazito teško razlikovati objekte po fizičkim svojstvima kao što su površina, oblik, konveksna ljuska i slično. Razvijene su razne tehnike i modeli značajki, a neke ćemo proučiti u daljnjem tekstu.

5.1.2 Haar značajke

Haar značajke ili još Haar-slične značajke su iskoristili VIola i Jones[8] kako bi napravili kaskadni klasifikator lica. Osnovne značajke su prikazane na slici 5.6.

[7] Na primjer želimo prepoznati neki objekt iz više kutova gledanja, a trening skup se većinom sastoji od frontalnih slika.

[8] Paul Viola i Michael Jones.



Slika 5.5: Osnovne Haar značajke. Skaliranjem, rotacijom i translacijom nastaju ostale Haar značajke. Svaka značajka je konvolucijska maska, a vrijednost značajke se dobiva oduzimanjem sume vrijednosti piksela ispod bijelog područja od sume crnog područja pomnoženog s omjerom crne i bijele površine. Haar značajke su osjetljive na promjenu kontrasta pa ih možemo interpretirati kao značajke ruba ili linije.

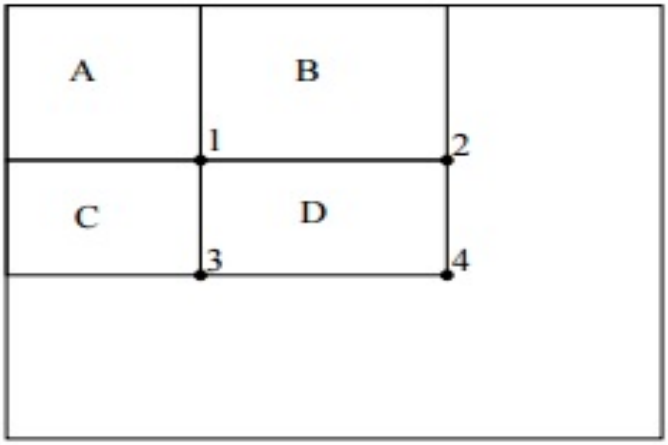
Prilikom treninga se na primjeru za vježbu računaju sve značajke. Značajki ima ukupno oko 180 000, stoga računanje zahtijeva mnogo vremena.

5.1.3 Integralna slika

Viola i Jones su korištenjem integralne slike značajno ubrzali računanje značajki.[**?**] Integralna slika je zapravo matrica $S_{m,n}$ gdje je svaki element zadan s:

$$s_{i,j} = \sum_{k \leq i} \sum_{l \leq j} f_{k,l}, i \leq m, j \leq n. \tag{5.2}$$

$f_{k,l}$ je vrijednost piksela izvorne slike dimenzije m, n. Svaki element $s_{i,j}$ matrice S je zapravo suma vrijednosti piksela lijevo i gore od piksela $f_{i,j}$ izvorne slike, uključujući vrijednost piksela $f_{i,j}$. Vidjeli smo da računanje Haar značajke zahtijevaju sumiranje elemenata unutar nekog područja. Pomoću integralne slike se to računanje svodi na nekoliko osnovnih računalnih operacija.



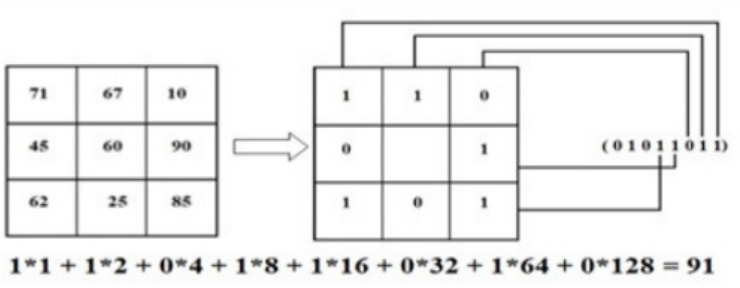
Slika 5.6: Želimo izračunati sumu vrijednosti piksela obuhvaćenih područjem D. Vrijednost Područja A je vrijednost integralne slike za element koji se nalazi na broju 1. Analogno, vrijednost kod broja 2 je $A + B$, kod 3 je $A + C$, kod 4 je $A + B + C + D$. Lako se vidi da je vrijednost područja $D = 4 + 1 - 2 - 3$,

Nakon inicijalnog računanja vrijednosti elemenata od S, računanje vrijednosti bilo kojeg područja svedeno je na operaciju zbrajanja i dvije operacije oduzimanja.

5.1.4 LBP značajke

Local binary pattern su bazirane na usporedbi piksela sa svojim najbližim susjedima. Time su više orijentirane na teksturu, a ne oblik. Izračun LBP značajke se provodi na sljedeći način:

- Dio slike koji promatrano podijelimo u blokove, na primjer 16x16.
- Usporedi vrijednost piksela sa osam najbližih susjeda. Piksel čija je vrijednost manja označi sa 0, a one s većom vrijednošću s 1. U smjeru kazaljke na satu ili obrnuto redom zapiši vrijednosti usporedbe sa susjednim pikselima. Dobiveni binarni zapis prebaci u dekadski sustav. Taj broj predstavlja binarni nivo piksela.
- Izračunaj histogram binarnih nivoa u bloku
- Normaliziraj dobiveni histogram
- Spoji sve histograme blokova. Tako dobiveni histogram je vektor značajke tog dijela slike.



Slika 5.7: Prikaz izračuna binarnog nivoa piksela.[10]

5.1.5 Kaskadni klasifikator

Kaskadni klasifikator je oblik strojnog učenja temelji na *AdaBoost* algoritmu,a detekciju objekata na Haar značajkama i 2-klasnoj klasifikaciji objekata, odnosno da li zadani objekt pripada klasi ili ne pripada. 2-klasna klasifikacija je klasifikacija uzoraka (primjera) u dvije prethodno zadane klase, gdje je jedna klasa negacija druge klase.[12]

Klasifikatorkoji s tek nešto većom vjerojatnošću ispravne klasifikacije od nasumičnog pogađanja nazivamo i slabim[9] klasifikatorom.[6]. Slabi klasifikator ima slabu korelacija sa ispravnom klasifikacijom, pa se pomoću *boosting* algoritma, najčešće AdaBoost, težinskim sumiranje stvara se jaki klasifikator. Opisat ćemo korake AdaBoost algoritma korištenog kod Viole i Jonesa, koji za zadani skup primjera i zadanim klasifikacijama stvara jaki klasifikator:
Zadano je n trening primjera $x_1, x_2, \ldots, x_n$ sa klasifikacijama $y_1, y_2, \ldots, y_n$ gdje je $y_i = 0$ ako je x_i negativan primjer, odnosno $y_i = 1$ ako je x_i pozitivan primjera. Prvo se inicijaliziraju težine primjera tako da je težina primjera x_i jednaka

$$w_{1,i} = \frac{1}{2(m(1 - y_i) + l * y_i)}$$

gdje je m broj negativnih primjera, a l broj pozitivnih primjera. Za t od 1 do T, gdje je T željeni broj *boost*-anja radi:

- Normaliziraj težine primjera tako da je za svaki x_i nova težina jednaka

$$w_{t,i} = \frac{w_{t,i}}{\sum_{j=1}^{n} w_{t,j}}.$$

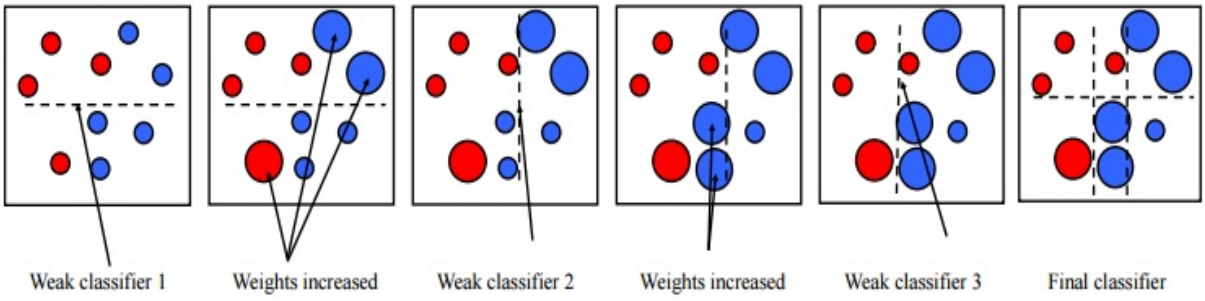
[9]Eng. *Weak Classifier*

- Za svaku značajku k odredi graničnu vrijednost koja minimizira pogrešnu klasifikaciju, odnosno stvori slabi klasifikator $h_k(x)$. Zatim za svaki slabi klasifikator $h_k(x)$ odredi pogrešku
$$\epsilon_k = \sum_{i=1}^{n} w_{t,i}|h_k(x_i) - y_i|.$$
- Odredi klasifikator $h_k(x)$ s najmanjom pogreškom ϵ_k i zapamti kao $c_t(x)$ s pogreškom E_t.
- Ponovno postavi težine primjerima tako da je za svaki primjer x_i nova težina jednaka
$$w_{t+1,i} = w_{t,i}\beta_t^{1-e_i}$$
gdje je $e_i = |c_t(x_i) - 1|$ i $\beta_t = \frac{E_t}{1-E_t}$.

Nakon T boostanja jaki klasifikator je jednak

$$h(x) = \begin{cases} 1, \text{ako je } \sum_{t=1}^{T} \alpha_t c_t(x) \geq \frac{1}{2} \sum_{i=1}^{n} \alpha_t \\ 0, \text{inače} \end{cases}$$

gdje je $\alpha_t = \log \frac{1}{\beta_t}$.



Slika 5.8: Nakon klasifikacije jednog slabog klasifikatora, pogrešno klasificirani uzorci dobivaju na težini, nadalje se traži slabi klasifikator koji bolje klasificira prethodno krivo označene. Linearnom kombinacijom slabih klasifikatora nastaje jaki klasifikator.[14]

Da bi se ubrzalo vrijeme klasifikacije, koristi se kaskadni klasifikator. Ideja je da se klasificiranje organizira u nivoe tako da se svaki od nivoa sastoji od jednog jakog klasifikatora. Prilikom treninga, negativno klasificirani pozitivni uzorci se ne prosljeđuju višim nivoima,

a parametri su podešeni tako da je broj lažno negativnih uzoraka vrlo malen. Klasifikatori na nižim nivoima su manje složeni od onih na višim pa vrijeme potrebno za treniranje također ovisi o željenom broju nivoa.
Slika na kojoj tražimo objekt se dijeli na potprozore. Mogućnost prepoznavanja na više skala, odnosno prepoznavanje neovisno o veličini objekta, se postiže skalairanjem[10] i računanjem značajki za svaku skalu. Umjesto skaliranja slike[11], povećava se veličina značajke[7]. Prilikom prepoznavanja potprozor traženja se pomiče za vrijednost od Δs gdje je s trenutna skala. Δs utječe na performanse klasifikatora. Viola i Jones su koristili $\Delta s = 1.0$, a promjenom vrijednosti $\Delta s = 1.5$ su postigli značajno ubrzanje uz nešto lošiju preciznost.[7]. Prilikom provjere, najviše potpodručja slike eliminira prvih nekoliko nivoa, a svako negativno klasificirano potpodručja se ne prosljeđuje višim nivoima, pa se potrebno vrijeme smanjuje. Potpodručje slike koje prođe kroz sve nivoe klasifikacije predstavlja traženi objekt. U praktičnom dijelu rada pokazat ćemo kako se trenira kaskadni klasifikator u OpenCV-u.

5.2 Ključne točke

Kada želimo zapamtiti neki objekt zapravo pamtimo njegove ključne karakteristike, analogno tome, možemo pomoću računala locirati ključne točke nekog objekta. Objekt želimo prepoznati bez obzira na njegovu rotaciju i skalu, a upravo nam tu pomažu ključne točke. Algoritmi za rad s njima su dizajnirani tako da pronalaze točke koje su sa svojom okolinom invarijantne na rotaciju i skaliranje. Opisat ćemo najpoznatiji algoritam SIFT. Prepoznavanje pomoću ključnih točaka možemo podijeliti u tri faze: otkrivanje ključnih točaka, stvaranje opisnika značajki i povezivanje značajki.

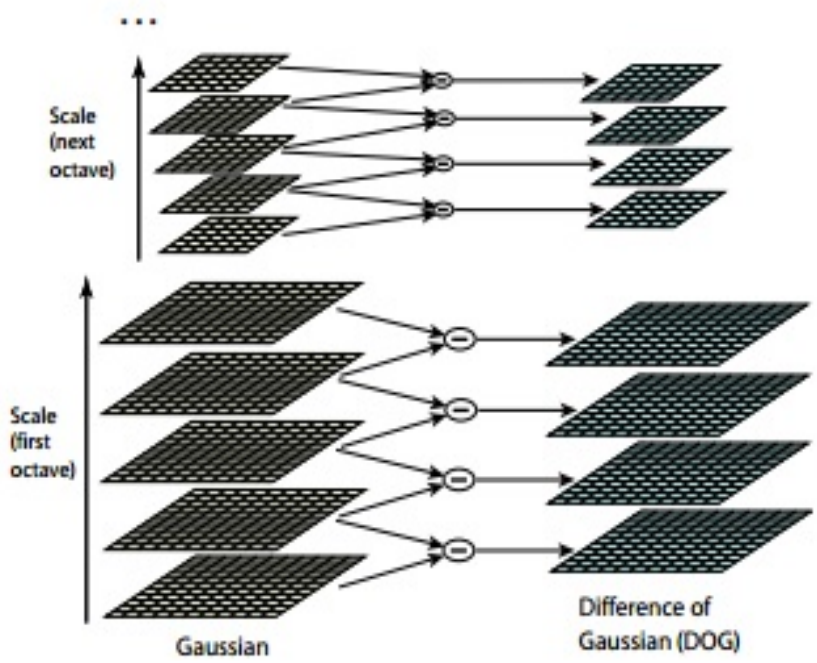
5.2.1 Scale-invariant Feature Transform

SIFT algoritam je razvio David Lowe 1999. godine. Prikazat ćemo na koji način algoritam traži ključne točke i stvara opisnike.

- Slika na kojoj tražimo ključne točke se konvoluira Gaussoviom maskama, povećavajući vrijednost σ za faktor k^n gdje je $n = 0, 1, 2, 3, 4$, $k = \sqrt{2}$. Slike se organiziraju u oktavu, nakon čega se razlučivost slike smanji za faktor 2 i ponavlja se proces do željenog broja oktava.

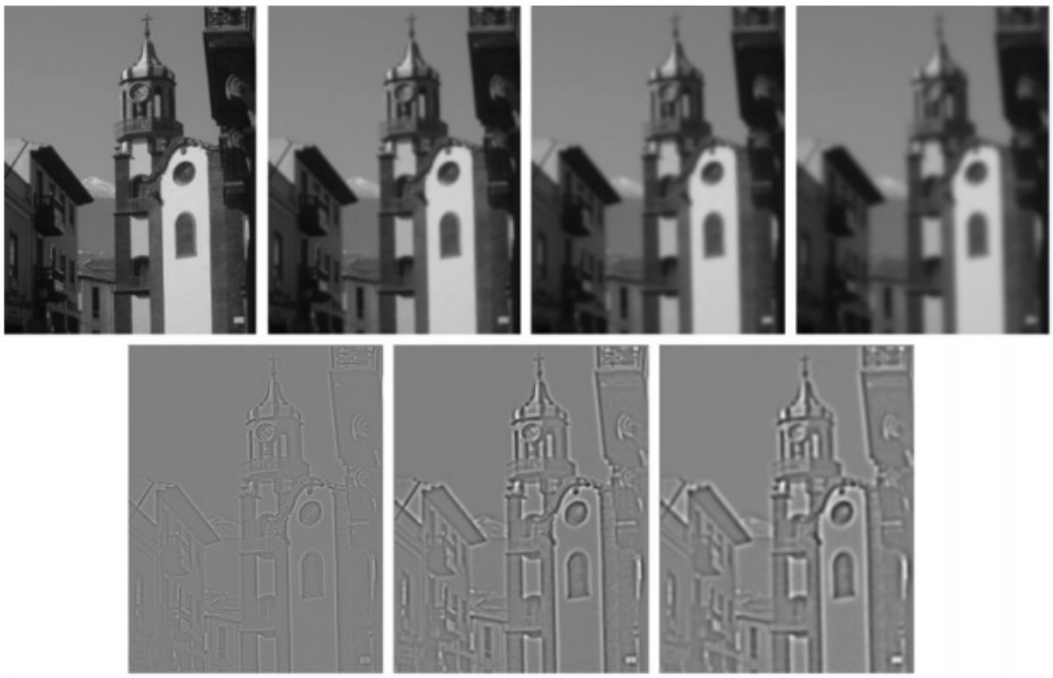
[10] Eng. *Multi scale*
[11] Prilikom treninga se također skaliraju značajke.

- Stvara se takozvana piramida skala. Oduzimanjem uzastopnih slika u oktavi dobiva se DoG[12] operator. Lowe je matematički pokazao da je rezultat primjene DoG-a direktno proporcionalan bliskoj aproksimaciji LoG-a[3].



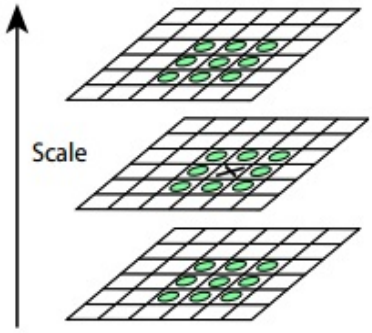
Slika 5.9: Skala s prikazom razlike Gaussa.[14]

[12] *Difference of Gaussians.*



Slika 5.10: U gornjem redu su slike konvulirane Gaussovim maskama različitih varijanci, a u donjem redu su razlike DoG.[6]

- Lokalni ekstremi se pronalaze usporedbom piksela sa svojih 26 susjeda u tri skale. Nazivamo ih točke interesa. Skala točke interesa se određuje kao kvadratni korijen varijance.[3]

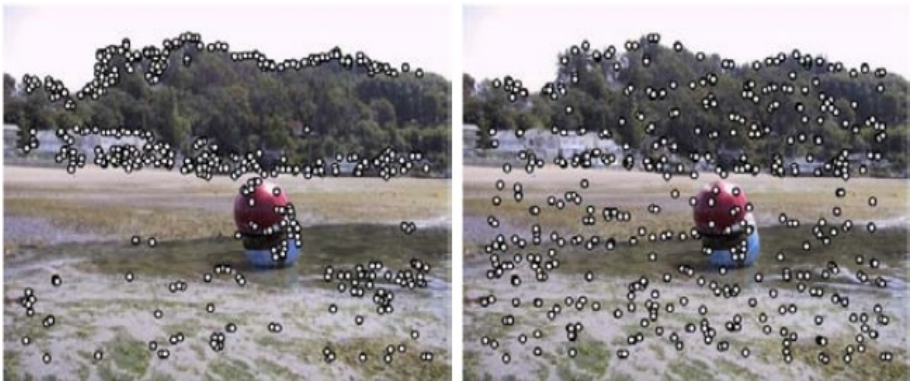


Slika 5.11: Uspoređuju se najbliži susjedi u tri skale.[14]

- S obzirom da se ključne točke oslanjaju na gradijent (kontrast) podložne su grupiranju u područjima jakog kontrasta. Da bi se doskočilo tom problemu koristi se traženje lokalnog maksimuma među točkama interesa u nekom zadanom radijusu. Također se postavlja granična vrijednost tako da lokalni maksimum mora imati dosta veći odaziv od ostalih točaka u tom području. Postupak se naziva *Adaptive non-maximal susppression* (ANMS).
  Također se eliminiraju točke lokalizirane na rubovima. Koristi se tehnika temeljena na reprezentaciji razlike Gaussa kao plohe i mjerenje zakrivljenosti uz pomoć Hessove matrice. Naime, ukoliko točka leži na rubu tada će jedna od glavnih zakrivljenosti (ploha reprezentirana razlikom Gaussa) biti dosta veća od druge. Za danu točku $x = (i, j)$ na slici, Hessova matrica $H(x, \sigma)$ u točki x i skali σ je jednaka

$$H(x, \sigma) = \begin{bmatrix} D_{ii}(x, \sigma) & D_{ij}(x, \sigma) \\ D_{ij}(x, \sigma) & D_{jj}(x, \sigma) \end{bmatrix} \tag{5.3}$$

  gdje je D_{ii}, D_{jj} i D_{ij} druge derivacije razlike Gaussa na skali σ.



Slika 5.12: Prikaz 500 ključnih točaka bez primjene ANMS-a (lijevo). Nakon primjene ANMS-a sa radijusom $r = 16$ (desno). [14]

- Na skali ključne točke se određuje orijentacija ključne točke. Uzima se područje slike ograničeno kružnicom sa centrom u ključnoj točki radijusa 1.5 puta većeg od skale slike. Orijentacija točke (i_0, j_0) se računa prema sljedećoj formuli:

$$\theta(i, j) = tg^{-1}\left(\frac{L(i, j+1) - L(i, j-1)}{L(i+1, j) - L(i-1, j)}\right) \tag{5.4}$$

gdje je L slika konvoluirana Gaussovom maskom na skali ključne točke. Sve orijentacije su reprezentirane sa 36 smjerova. Orijentacije promatranog područja slike se prikazuju

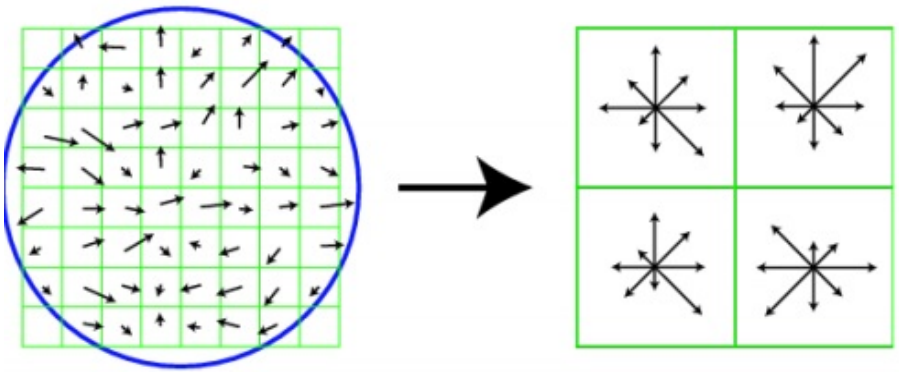
histogramom te se traži orjentacija sa najvišom frekvencijom. Ukoliko postoji nekoliko istaknutih orijentacija[13]stvara se nekoliko ključnih točaka na istom mjestu ali s različitim orijentacijama.

Slika 5.13: Lijevo su prikazani svi kandidati za ključne točke. U sredini su točke preostale nakon eliminacije na temelju kontrasta. Desno su su ključne točke preostale nakon eliminacije točaka koje su loše lokalizirane.[14]

Prethodnim postupkom smo odredili sve ključne točke. Time smo završili prvu fazu: otkrivanje ključnih točaka. Preostaje nam odrediti opisnike tih točaka. Svaka SIFT ključna točka ima i pripadni opisnik koji sadrži 128 elemenata. Kvadratna regija oko ključne točke se podijeli u 16 jednakih blokova. Za svaki blok se orijentacije gradijenata prikazuju histogramom sa osam različitih smjerova orijentacije. Frekvencija smjera na histogramu se prikazuju kao umnožak magnitude gradijenta i Gaussove konvolucijske maske[14] tako da je σ jednaka pola širine promatrane kvadratne regije, odnosno svaki smjer gradijenta doprinosi ukupnom smjeru ovisno o udaljenosti od ključne točke. Da bi se postigla invarijantnost na rotaciju, koordinate točaka i gradijenti te regije se rotiraju u smjeru orijentacije ključne točke.

[13]Do maksimalno 20% razlike od najfrekventnije orijentacije[3]

[14]Središnju element maske je na ključnoj točci.



Slika 5.14: Prikaz pojedinačnih gradijenata (lijevo) u kvadratnom području oko ključne točke i gradijenata opisnika ključne točke (desno). Prikazano kvadratno područje je dimenzije 8x8, a u stvarnom radu se koristi područje dimenzije 16x16.[14]

Nakon stvaranja opisnika koji sadrži 16 blokova od kojih svaki osam smjerova dobiva se vektor sa 128 elemenata. Normalizacijom tog vektora se stvara invarijantnost na osvjetljenje. Nakon ekstrakcije svih opisnika dobivamo prostor opisnika (značajki).
Preostaje nam još povezivanje točaka. Za sliku koju ispitujemo se analogno prethodnom, pronalaze i opisuju ključne točke. Da bi se pronašle ključne točke koje se podudaraju, koristi se euklidska udaljenost te se u prostoru značajki računaju udaljenosti među vektorima značajki. Često se koristi algoritam traženja najbližih susjeda. Dobivene udaljenosti se filtriraju tako da se uspoređuje druga i prva najmanja udaljenost od određene značajke. Ukoliko je omjer manje i veće udaljenosti iznad 0.8[15] promatrana točka se odbacuje.

Osim SIFT-a u OpenCV-u su dostupni sljedeći algoritmi za detekciju ključnih točaka:

- ”FAST” – FastFeatureDetector
- ”STAR” – StarFeatureDetector
- ”SIFT” – SIFT (nonfree module)
- ”SURF” – SURF (nonfree module)
- ”ORB” – ORB
- ”MSER” – MSER
- ”GFTT” – GoodFeaturesToTrackDetector

[15]Eksperimentalno je pokazano da se tako eliminira 90% lažnih i 5% točnih poklapanja.

- ”HARRIS” – GoodFeaturesToTrackDetector with Harris detector enabled
- ”Dense” – DenseFeatureDetector
- ”SimpleBlob” – SimpleBlobDetector

Za povezivanje opisnika značajki u OpenCV-u se koriste algoritmi[1]:

- Brute-force (`cv::BFMatcher`)
- Flann-based (`cv::FlannBasedMatcher`)

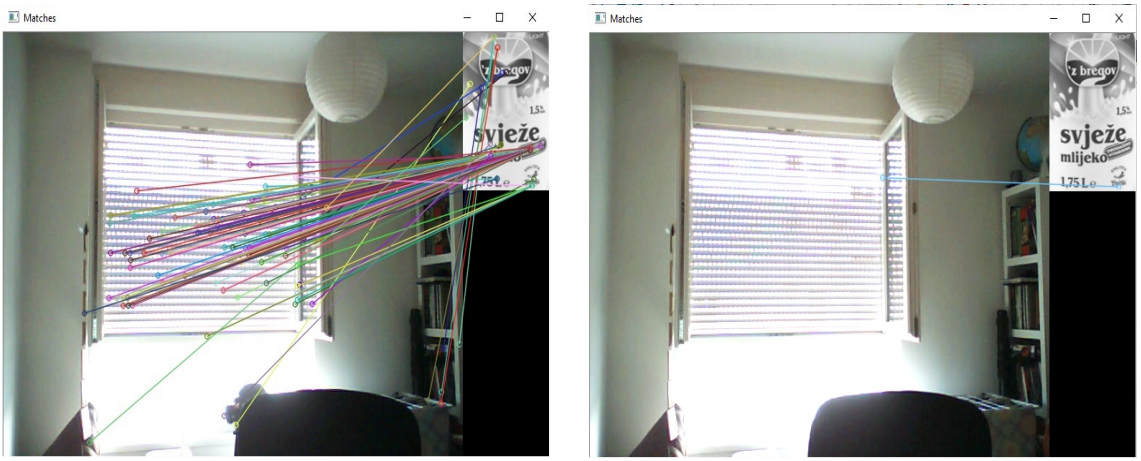
Brute-force algoritam traži najbolje podudaranje uspoređujući svaki opisnik iz zadanog skupa sa svakim opisnikom iz skupa koji se ispituje. *Flann-based* se temelji na optimiziranim algoritmima za traženje najbližih susjeda. Uz prethodne algoritme možemo koristiti metodu `.train()`. Brute-force povezivanje u svakom slučaju prolazi cijelim skupom opisnika, pa se pozivanjem prethodne metode samo pohranjuju opisnici skupa slika za trening. Kod *flann-based* povezivanja se metodom treninga stvaraju indeksna stabla opisnika, a time se povećava brzina povezivanja.

Pokazat ćemo jednostavan primjer prepoznavanja objekta pomoću SIFT algoritma.

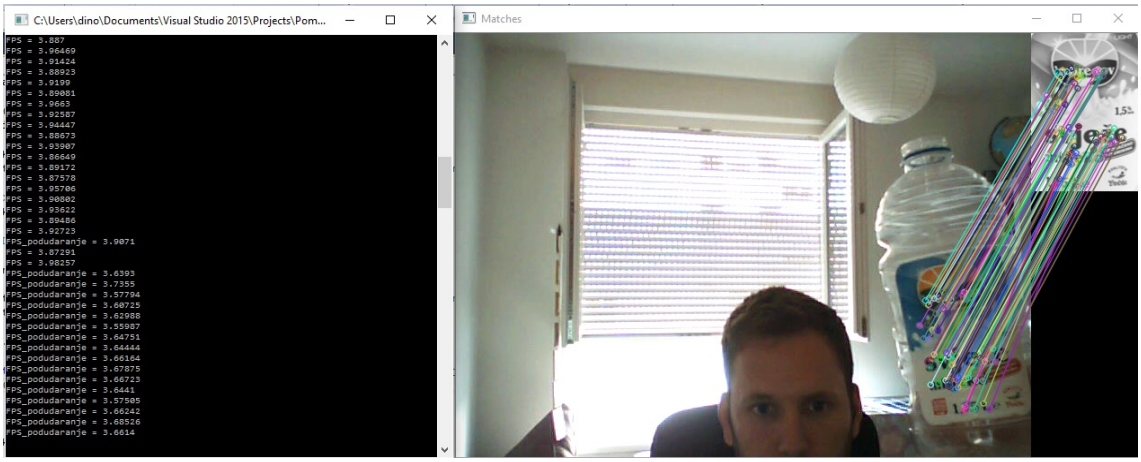
```
1  #include <stdio.h>
2  #include <iostream>
3  #include "opencv2/core/core.hpp"
4  #include "opencv2/features2d/features2d.hpp"
5  #include "opencv2/highgui/highgui.hpp"
6  #include "opencv2/nonfree/features2d.hpp"
7  #include <opencv2/nonfree/nonfree.hpp>
8  #include <opencv2/imgproc/imgproc.hpp>
9
10 using namespace cv;
11 using namespace std;
12
13 int main(int argc, char** argv)
14 {
15
16   Mat slika_kamera;
17   Mat opisnik_primjer, opisnik_kamera;
18   Mat siva_slika_kamera;
19   Mat slika_primjer = imread("C:/Users/dino/OneDrive/Dino Diplomski/
       slike/zbregov.jpg", CV_LOAD_IMAGE_GRAYSCALE); // ucitavanje slike
       uzorka i pretvaranje u sivu prilikom ucitavanja
20
21   if (!slika_primjer.data) // provjera da li je slika ucitana
22   {
23     return -1;
24   }
```

```
25
26  VideoCapture kamera("D:\MyVideo.avi");
27
28  SiftFeatureDetector sift_detector; // inicijalizacija detektora
     kljucnih tocaka
29  SiftDescriptorExtractor sift_extraktor; // inicijalizacija ekstraktora
     opisnika kljucnih tocaka
30
31  vector<KeyPoint> kljucne_tocke_primjer, kljucne_tocke_kamera;
32  vector<vector< DMatch> >podudaranja;
33
34  sift_detector.detect(slika_primjer, kljucne_tocke_primjer); //trazenje
     kljucnih tocaka uzorka
35  sift_extraktor.compute(slika_primjer, kljucne_tocke_primjer,
     opisnik_primjer); // stvaranje opisnika kljucnih tocaka uzorka
36
37  FlannBasedMatcher matcher; // inicijalizacija Flann-based pronalazaca
     podudaranja
38  vector<Mat> skup_opisnika(1, opisnik_primjer);// kako bismo mogli
     trenirati s vise slika, spremaju se u vektor
39  matcher.add(skup_opisnika);
40  matcher.train();
41
42  while (waitKey(1) != 27) {
43    double t_0 = cvGetTickCount(); // pomocna varijabla za mjerenje
     vremena izvrsavanja
44
45    kamera >> slika_kamera;
46    if (slika_kamera.empty()) return -1;
47
48    vector<DMatch> povoljna_podudaranja;
49
50    cvtColor(slika_kamera, siva_slika_kamera, CV_BGR2GRAY);
51
52    sift_detector.detect(siva_slika_kamera, kljucne_tocke_kamera);  //
     trazenje kljucnih tocaka slike kamere
53    sift_extraktor.compute(siva_slika_kamera, kljucne_tocke_kamera,
     opisnik_kamera); // stvaranje opisnika kljucnih slike kamere
54
55    matcher.knnMatch(opisnik_kamera, podudaranja, 2);           //
     trazenje podudaranja algoritmom knn
56
57    for (int i = 0; i < podudaranja.size(); i++) {             //
     eliminacija nepovoljnih podudaranja na temelju omjera udaljenosti
     kljucnih tocaka
58      if (podudaranja[i][0].distance < 0.6 * podudaranja[i][1].distance)
59        povoljna_podudaranja.push_back(podudaranja[i][0]);
```

```
60      }
61
62      Mat slika_podudaranja;
63
64      drawMatches(slika_kamera, kljucne_tocke_kamera, slika_primjer,
        kljucne_tocke_primjer, povoljna_podudaranja, slika_podudaranja,
        Scalar::all(-1), // crtanje podudaranja
65          Scalar::all(-1), vector<char>(), 2);
66
67      imshow("Matches", slika_podudaranja);
68
69      if (!povoljna_podudaranja.empty()) { cout << "FPS_podudaranje = " <<
        getTickFrequency() / (getTickCount() - t_0) << endl; } // prikaz
        vremena izvodenja
70      else cout << "FPS = " << getTickFrequency() / (getTickCount() - t_0)
        << endl;
71
72    }
73    return 0;
74
75
76
77  }
```



Slika 5.15: Lijevo su prikazana podudaranja koristeći omjer 0.8 najbližih podudaranja za eliminaciju nepovoljnih, desno je omjer postavljen 0.6



Slika 5.16: Brzina izvođenja je oko četiri sličice u sekundi, a ovisi broju ključnih točaka na slici kamere, no opet nije povoljna za izvršavanje u realnom vremenu. Vidimo da brzina lagano opadne kada se pojavi veći broj podudaranja.

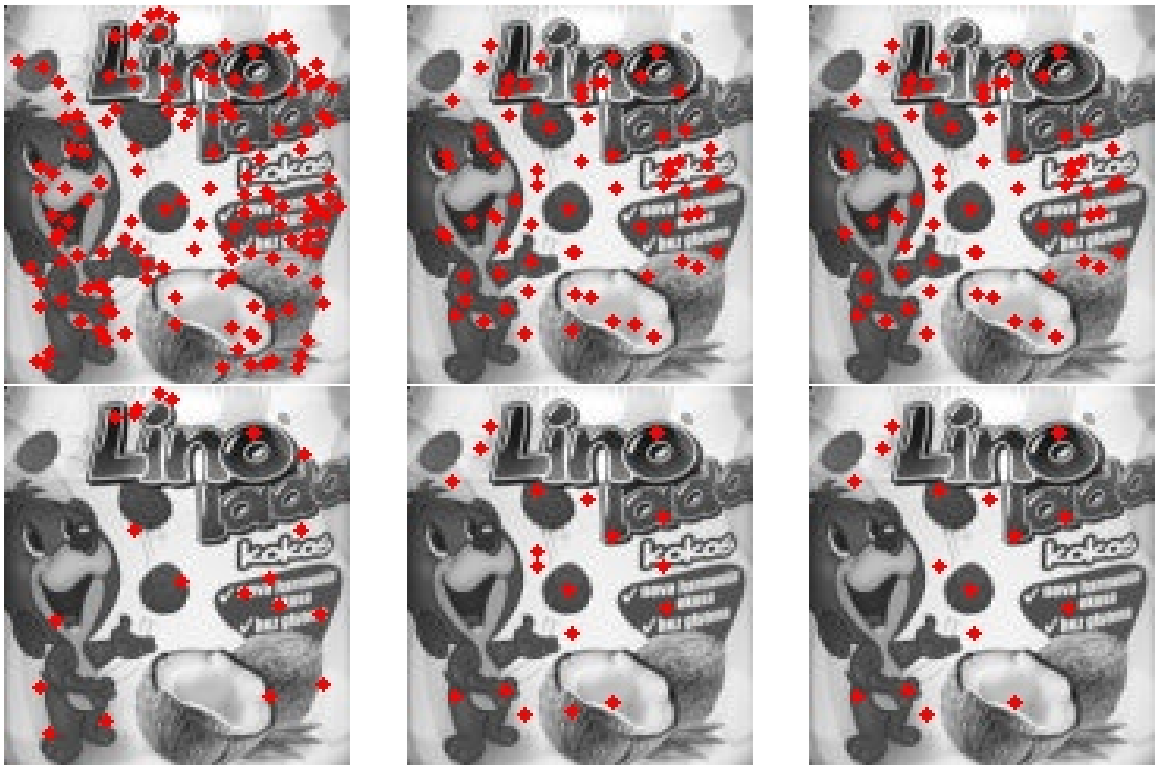
Brzina izvođenja je otprilike tri sličice u sekundi. Prilikom pronalaska podudaranja, brzina je nešto manja. Također ovisi i o broju ključnih točaka pronađenih na slici kamere. Usporedit ćemo brzinu izvođenja i broj podudaranja ključnih točaka za algoritme SIFT i SURF. Da bismo izmjerili brzinu i podudaranje ključnih točaka potreban nam je video zapis na kojem ćemo provesti testiranje. Uz pomoć programa `snimanje_video.cpp` ćemo snimiti neku pozadinu. Koristeći `ubacivanje_slike` ćemo nasumice dodati rotirane i skalirane slike traženog objekta u video zapis pozadine. Na dobivenom video zapisu ćemo mjeriti broj sličica u sekundi[16] pri radu s algoritmima SIFT i SURF, te broj lažno pozitivnih i istinito pozitivnih povezivanja ključnih točaka za svaki od algoritama. U tome će nam pomoći programi `mjerenje_vremena.cpp` i `mjerenje_pogodaka.cpp`.
video zapis je rezolucije 640x480 piksela, a rezolucija slike objekta je 100x125 piksela. Računalo: Intel Core i5 2500k[17], Nvidia GTX 970, 8GB DDR3, SSD 240GB. Prikazat ćemo točno povezane - TP i lažno pozitivne ključne - FP točke na zadanoj slici objekta.

[16]Eng. *Frames Per Second*
[17]Takt procesora je podignut na 4.5 Ghz.

Algoritam	FPS prisutan objekt	FPS nema objekta	TP	FP	TP / (TP +FP)
SURF(300)	18.573	22.275	12921	1787	0.879
SURF(500)	20.779	24.945	12438	1403	0.897
SIFT	4.927	5.567	35440	1875	0.950

Tablica 5.1: Mjerenja za SURF algoritam su prikazana u ovisnosti o graničnoj vrijednosti hessian. Hessian predstavlja jačinu odaziva ključne točke, odnosno umnožak svojstvenih vrijednosti Hessove matrice. Takve točke interpretiramo kao bolje lokalizirane točke.

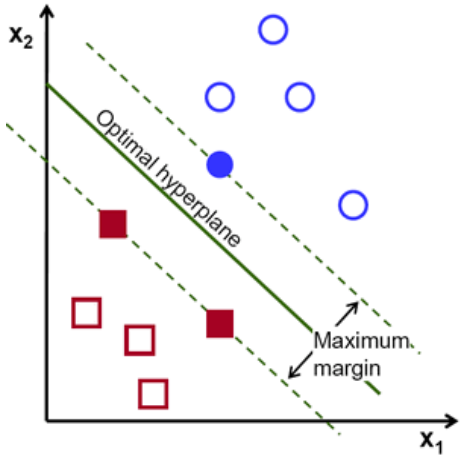
Algoritam SURF je preko četiri puta brži u izvođenju od SIFT algoritma, no broj točaka koje su pozitivno povezane je dosta veći kod SIFT algoritma. SURF algoritam ima najmanji broj lažno pozitivnih točaka, no ako pogledamo postotak istinito pozitivnih točaka tada SIFT ima bolje performanse.



Slika 5.17: Prikaz ispravno (gornji red) i neispravno (donji red) povezanih ključnih točaka. Lijevi stupac pokazuje rezultat za SIFT algoritam, u sredini je SURF(300) i desno SURF(500).

5.3 Support Vector Machine

Vidjeli smo da možemo relativno lako koristiti opisnike ključnih točaka iz trening primjera kako bismo tražili poklapanja na nekoj slici. Na takav način možemo prepoznati objekte koji su slični primjeru, no ako bismo htjeli prepoznati cijelu klasu objekata uz pomoć više trening primjera moramo koristiti složenije tehnike. Jedna od njih je *Support Vector Machine*, odnosno metoda potpornih vektora. SVM je vrsta klasifikatora. Temelji se na reprezentaciji značajki kao vektora. Vektori značajki čine prostor značajki, pa SVM u tom prostoru određuje hiperravninu koja klasificira vektore značajki s obzirom na položaj i udaljenost od hiperravnine. Ta ravnina je određena tako da ima najveću marginu između vektora klasa koji su blizu hiperravnine. Iz skupa primjera prilikom treninga se određuju takozvani potporni vektori koji su najbliži hiperravnini te služe prilikom klasifikacije primjera. SVM je pogodan za klasifikaciju značajki koje se temelje na ključnim točkama, jer su opisnici reprezentirani kao vektori. U OpenCV-u je SVM implementiran kao klasa `CvSVM`, a ulazne podatci su prikazani kao strukture `MAT`.



Slika 5.18: Prikaz rada SVM-a na jednostavnom primjeru.

Detaljan primjer kako trenirati SVM koristeći SURF značajke nalazi se u literaturi pod brojem [3].

Poglavlje 6

Praktični rad

U prethodnim poglavljima smo se bavili teorijskom pozadinom i prikazali uvod u rad s OpenCV-om. U idućem poglavlju pozabavit ćemo se praktičnim radom. Cilj nam je stvoriti osnovu za računalno sučelje temeljeno na prepoznavanju i praćenju dlana, te brojanju prstiju pomoću web kamere. Da bismo uspjeli, moramo riješiti dva glavna problema: kako prepoznati dlan i na njemu razlučiti prste. U radu je korištena web kamera logitech c310 razlučivosti 1280x720 piksela.

6.1 Trening kaskadnog klasifikatora

Prvo želimo izdvojiti dio slike koji sadrži otvoreni dlan, u daljnjem tekstu samo dlan. u tu svrhu ćemo trenirati kaskadni klasifikator. Za trening su potrebna dva skupa slika. Skup pozitivnih slika koji se sastoji od slika objekta kojeg želimo prepoznati, u našem slučaju su to slike dlana. Nazivamo ih pozitivnim primjerima, te skup slika koje ne sadrže traženi objekt, nazivamo ih negativnim primjerima. Pozitivne slike sadrže samo objekt, a kvaliteta skupa ovisi o količini pozitivnih slika, raznim pozama objekta i različitim pozadinama. Razlučivost pozitivnih primjera koje su koristili Viola i Jones[7] za detekciju lica je 24x24 piksela. Možemo ju promijeniti ali pod cijenu brzine treninga. Pozitivni primjeri ne moraju nužno biti iste razlučivosti , ali omjer visine i širine kod svih mora biti jednak. Razlučivost najmanjeg potprozora kojeg klasifikator može klasificirati je jednaka razlučivosti pozitivnih primjera. Negativni primjeri su veće razlučivosti od pozitivnih primjera, a u našem slučaju će biti razlučivosti kamere, odnosno 1280x720 piksela.
OpenCV nudi alat za trening kaskadnog klasifikatora `opencv_traincascade.exe`. Pokreće se iz komandne linije, a argumenti koje ćemo koristiti su:

- `-data put_do_mape` Mapa koja će služiti za spremanje rezultata tijekom treninga.
- `-vec ime_datoteke.vec` Datoteka tipa `.vec` koja sadrži pozitivne primjere.

- `-bg ime_datoteke.txt` Tekstualna datoteka koja sadrži put do svakog negativnih primjera.
- `-w int_w` Širina pozitivnih primjera u pikselima.
- `-h int_w` Visina pozitivnih primjera u pikselima.
- `-numPos int_pos` Broj pozitivnih primjera.
- `-numNeg int_neg` Broj negativnih primjera.
- `-numStages int_broj` Određuje broj nivoa prilikom treninga.
- `-minHitRate double_broj` Decimalni broj koji pomnožen s brojem pozitivnih primjera označava minimalan broj točno klasificiranih pozitivnih primjera za klasifikator na svakom nivou. Ukupan broj, za sve nivoe, dobije se kao `minHitRate^broj_nivoa`.
- `maxFalseAlarm double_broj` Određuje maksimalan broj lažno pozitivno ocijenjenih negatinih primjera za svaki nivo. Ukupan broj je jednak `maxFalseAlarm^broj_nivoa`.
- `-featureType vrsta` Vrsta značajki. Može biti HAAR ili LBP.

Za trenirane klasifikatore je broj nivoa jednak 11, `minHitRate` je postavljen na 0.999, a `maxFalseAlarm` na 0.4.
Potrebne su nam dvije tekstualne datoteke, jedna sadrži puteve do pozitivnih primjera, a druga putove do negativnih primjera. OpenCV također nudi alat `opencv_createsamples.exe` koji se pokreće preko komandne linije, a služi za stvaranje datoteka tipa `.vec`. Koristit ćemo sljedeće argumente:

- `-vec ime_datoteke.vec` Ime izlazne datoteke.
- `-info ime_datoteke_pos.txt` Tekstualna datoteka koja sadrži put do pozitivnog primjera, broj objekata na slici te njihovu poziciju i dimenzije u pikselima. Na primjer
  `C:\kaskadni_primjeri\pozitivni\pozitivna02.jpg 2 20 30 50 50  120 140 50 50 .`
- `-w` Širina pozitivnih primjera.
- `-h` Visina pozitivnih primjera.
- `-num` Broj pozitivnih primjera.

Uz pomoć ovog alata, moguće je iz jedne pozitivne slike i negativnih slika napraviti više pozitivnih primjera koristeći umetanje negativnog primjera kao pozadine i rotaciju pozitivne slike. Preporuča se korištenje stvarnih pozitivnih primjera.

6.1.1 Rezultati treninga

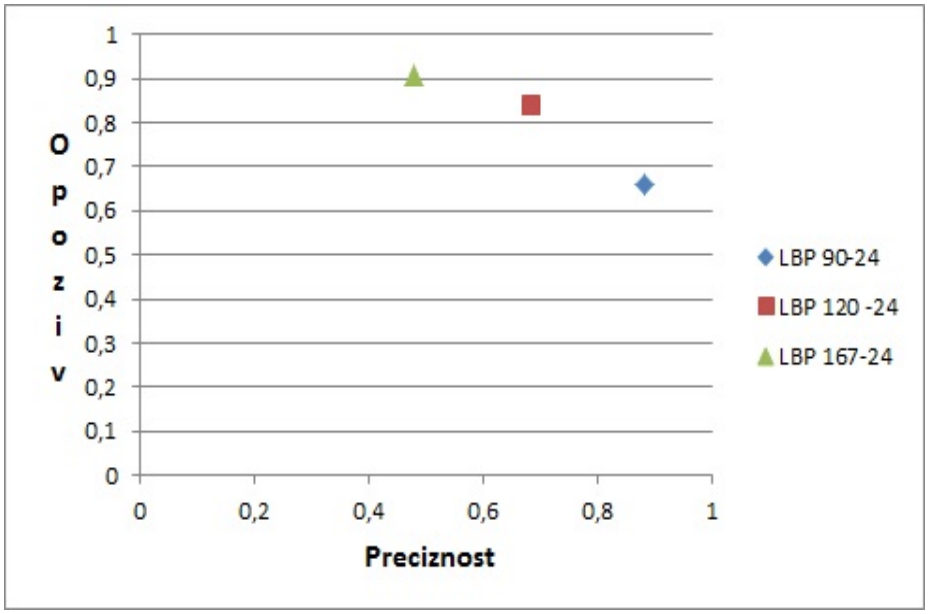
Slike korištene u treningu su pribavljene uz pomoć konstruiranog programa `crop.cpp`. Korišten je Gaussovo zaglađivanje za eliminaciju šuma i ekvalizacija histograma kako bi se smanjile varijacije u svjetlini. Dlan desne ruke je prikazan u frontalnoj pozi i rotiran na različitim pozadinama i uvjetima osvjetljenja.

Slika 6.1: Prikaz nekoliko korištenih pozitivnih primjera.

Tekstualna datoteka pozitivnih slika je napravljena uz pomoć datoteke `append_Pos.bat`. Za stvaranje datoteke `.vec` , korišten je alat `opencv_createsamples.exe`. Negativni primjeri su bilo koje slike na kojima se ne nalazi dlan. Prikupljeni su snimanjem okoline web kamerom i spremanjem svake sličice kao negativni primjer pomoću konstruiranog programa `snimanje_neg.cpp`. Nakon prikupljanja, uz pomoć datoteke `append_Neg.bat` stvorena je tekstualna datoteka koja sadrži put za svaki negativni primjer. Time smo prikupili sve potrebne podatke za trening kaskadnog klasifikatora. Prikazat ćemo nekoliko treniranih kaskadnih klasifikatora te izmjerene rezultate.

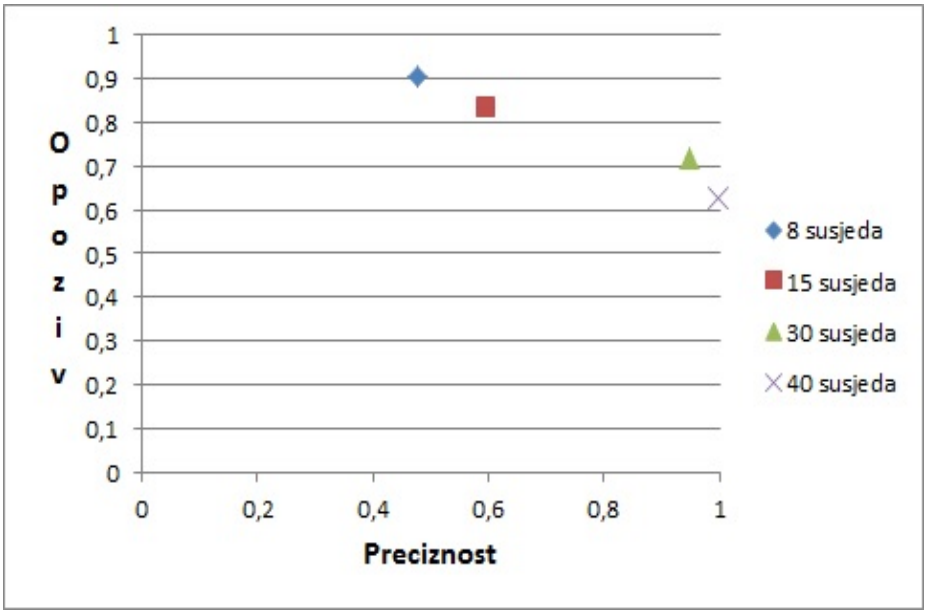
Za mjerenje rezultata su korištena četiri video zapisa snimljena na različitim pozadinama i uvjetima osvjetljenja. Snimana je ruka u pokretu s pokazanim dlanom. Na svakom zapisu je ručno uz pomoć konstruiranog programa `tag_video.cpp` zabilježena pozicija dlana u tekstualnu datoteku. Ukupan broj testnih pozitivnih potprozora je 2620. Zatim je konstruiranim programom `mjerenje_pogodaka_cascade.cpp` izmjeren broj istinito pozitivnih detekcija - TP, lažno negativnih - FN i lažno pozitivnih - FP, za svaki trenirani kaskadni klasifikator. Mjereni su opoziv i preciznost klasifikatora. Opoziv je jednak $\frac{TP}{TP+FN}$,

a predstavlja ukupan postotak detektiranih objekata. Preciznost je jednaka $\frac{TP}{TP+FP}$, odnosno postotak pozitivnih klasifikacija koje su točne.
Za svaki trenirani klasifikator korišten je skup od 10484 negativna primjera. Napravljena su tri skupa pozitivnih primjera koji redom sadrže 90, 120 i 167 slika, gdje je manji skup podskup većeg. Za svaki skup je treniran klasifikator sa LBP značajkama i dimenzijama pozitivnih primjera 24x24. Rezultati su prikazani sljedećim grafikonom.



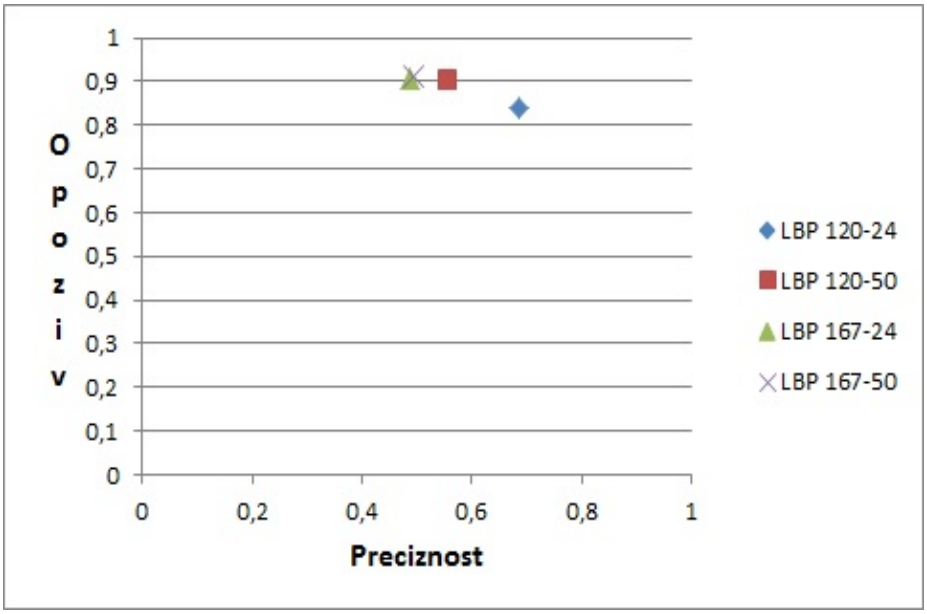
Trenirani klasifikator sa 167 pozitivnih primjera detektira dlan u 90% slučajeva, ali preciznost iznosi 0.47. U OpenCV-u metoda za detekciju, klase kaskadnog klasifikatora, ima parametar `minNeighbors` koji određuje minimalan broj susjednih potprozora[1] da bi se dio slike klasificirao kao dlan. U prethodnim testiranjima vrijednost je postavljena na 8. Povećanjem vrijednosti ćemo smanjiti broj lažno pozitivnih detekcija. Prikazat ćemo rezultate za klasfikator `LBP 167_24` s minimalnim brojem susjeda postavljenim na 15, 30 i 40.

[1]Na različitim skalama. Korišten je $\Delta s = 1.05$.



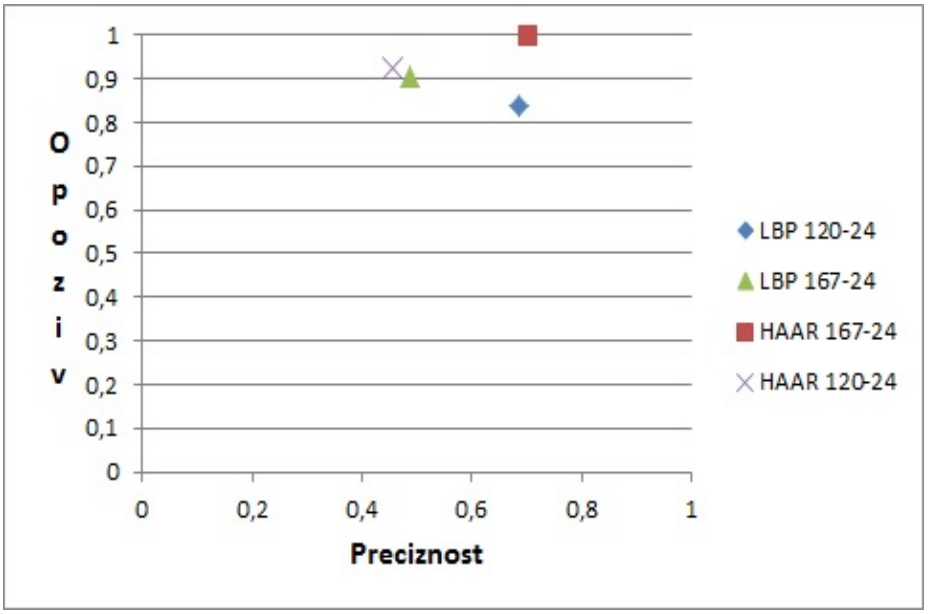
Slika 6.2: Rezultati povećanja minimalnog broja susjeda.

Regulacijom broja susjeda možemo uvelike smanjiti broj lažno pozitivnih detekcija, ali se prilikom toga smanjuje broj istinito pozitivnih detekcija. Za daljnju usporedbu uzet ćemo klasifikatore trenirane na 120 i 167 pozitivnih primjera te ih usporediti sa klasifikatorima treniranim na istim pozitivnim primjerima ali razlučivosti 50x50. Minimalan broj susjeda ćemo ostaviti na 8, jer ćemo prvo pronaći klasifikator sa najvećim opozivom, a zatim povećanjem minimalnog broja susjeda povećati preciznost.



Slika 6.3: Klasifikator treniran sa 120 pozitivnih primjera dimenzije 50x50 ima isti opoziv ali veću preciznost od klasifikatora treniranog na primjerima razlučivosti 24x24. Klasifikator treniran na 167 primjera razlučivosti 50x50 također ima nešto bolje performanse.

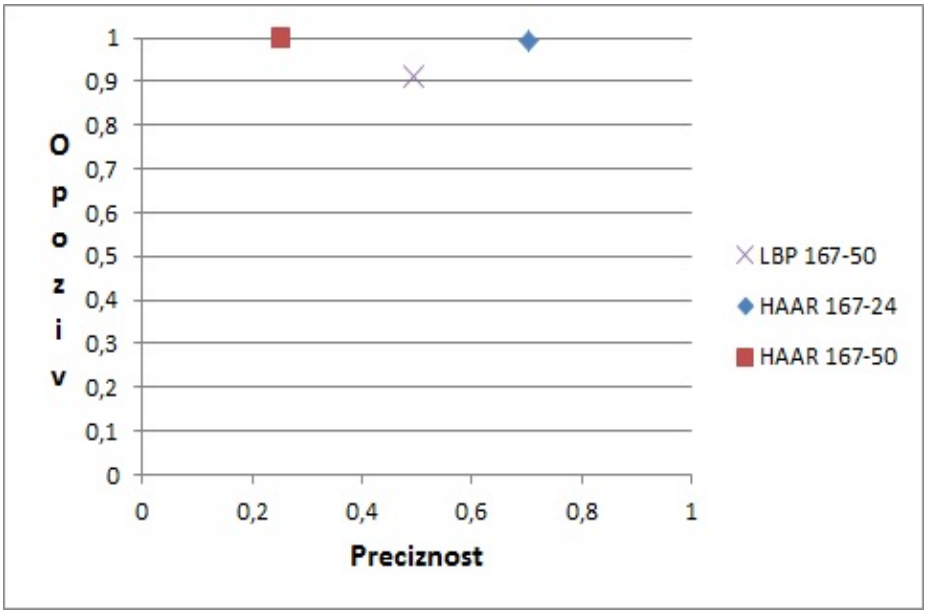
LBP značajke koriste cjelobrojne podatke, dok se za računanje HAAR značajke koriste realne tipove podataka, pa treniranje takvog klasifikatora zahtijeva daleko više vremena. Stoga smo prvo uspoređivali klasifikatore s LBP značajkama. Sada ćemo ih usporediti s klasifikatorima treniranim na 120 i 167 pozitivnih primjera razlučivosti 24x24 koristeći HAAR značajke .



Slika 6.4: Usporedba klasifikatora treniranih sa HAAR i LBP značajkama.

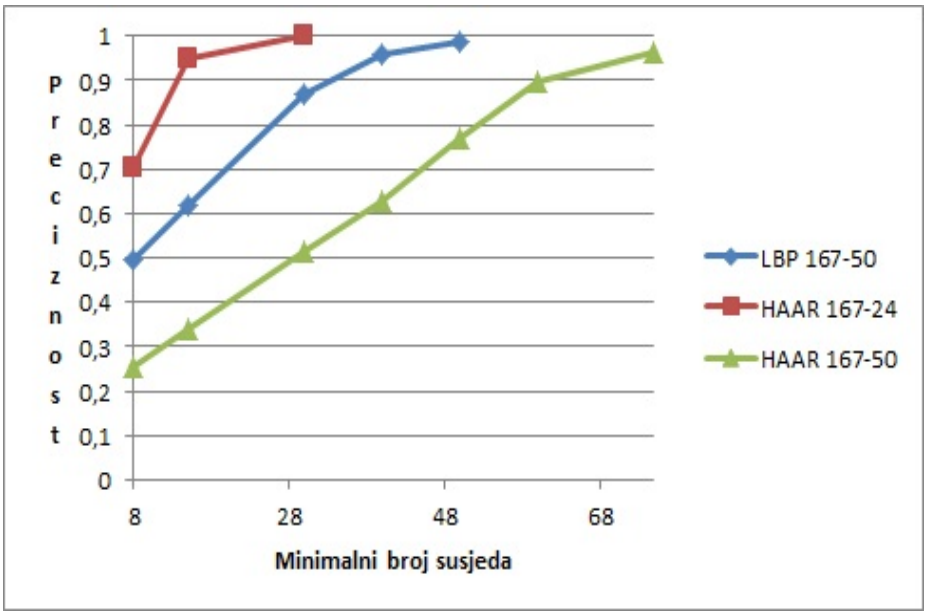
Klasifikatori koji koriste HAAR značajke u oba slučaja imaju veći opoziv od klasifikatora s LBP značajkama na istom skupu pozitivnih primjera. Preciznost je nešto manja na 120 primjera, a bolja kod treninga na 167 primjera. Klasifikator treniran na 167 pozitivnih primjera s HAAR značajkama postže opoziv od čak 0.994. Prethodna zapažanja daju motivaciju za treniranjem kaskadnog klasifikatora s HAAR značajkama i 167 pozitivnih primjera razlučivosti 50x50 piksela.

Trening klasifikatora HAAR 167-50 trajao je 99 sati, odnosno nešto duže od četiri dana dok je trening s LBP značajkama na istim primjerima trajao1 sat i 46 minuta. Za usporedbu ćemo uzeti dva klasifikatora s najvećim opozivom HAAR 167-24 i LBP 167-50.

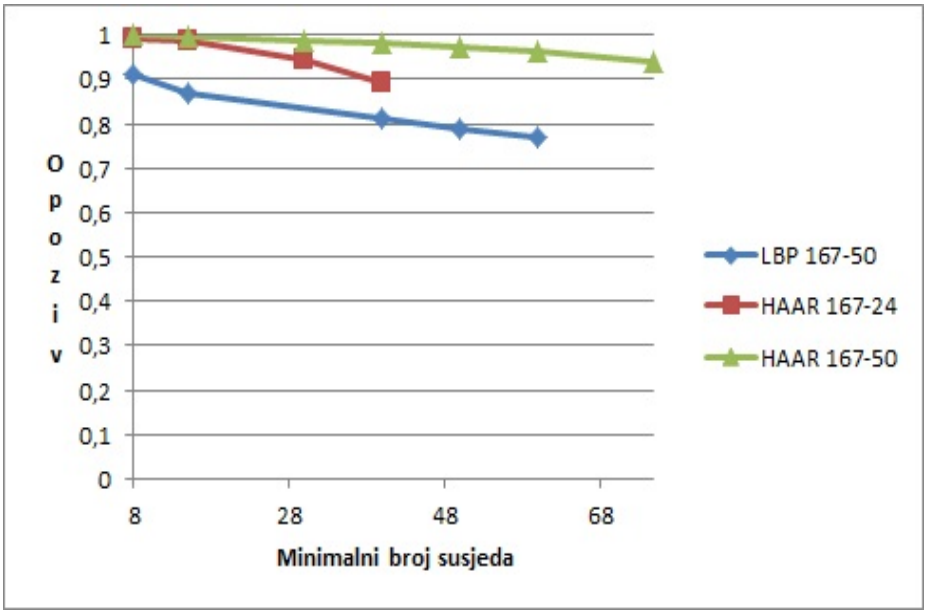


Slika 6.5: HAAR 167-50 postiže opoziv od 0.999, što je najbolji rezultat, ali preciznost je vrlo niska i iznosi 0.255.

Preciznost možemo povećati povećanjem minimalnog broja susjeda. Prikazat ćemo rezultate za tri najbolja klasifikatora.



Slika 6.6: Prikaz preciznosti ovisno o minimalnom broju susjeda.



Slika 6.7: Prikaz opoziva ovisno o minimalnom broju susjeda.

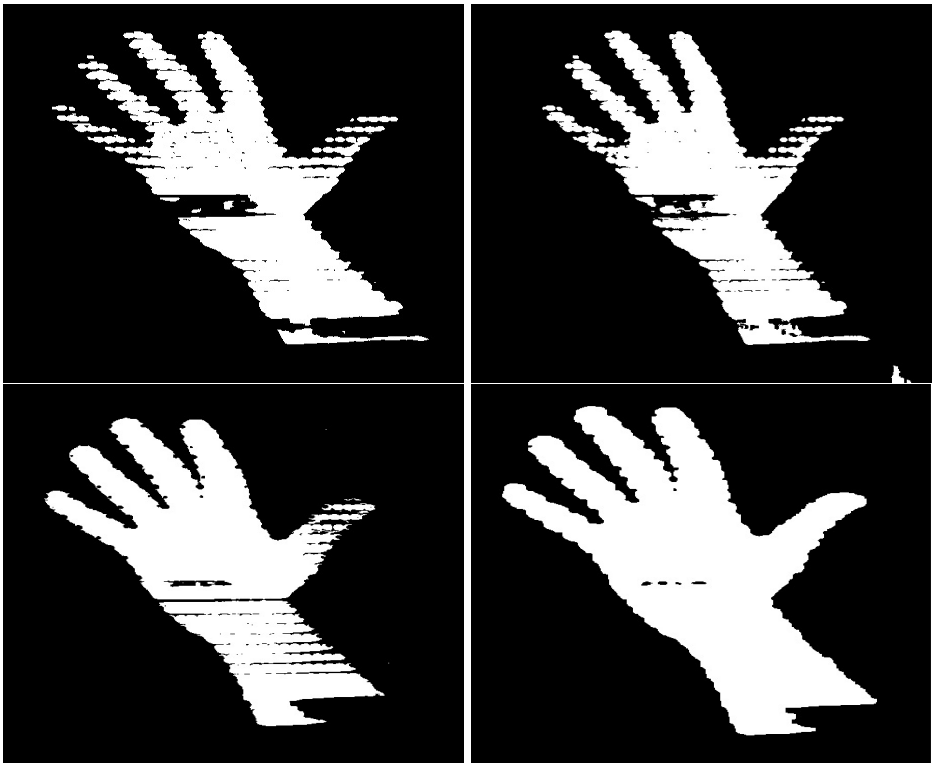
S minimalnim brojem susjeda 50, opoziv klasifikatora LBP 167-50 je pao na 0.771, a pritom je preciznost iznosila 0.987. Daljnje povećanje minimalnog broja susjeda za ovaj klasifikator nije ispitivano zbog niskog opoziva, čime bi izostao veliki postotak istinito pozitivnih detekcija. Klasifikator HAAR 167-24 za minimalan broj susjeda 40 ima postiže opoziv od 0.890 i preciznost 1. Klasifikator HAAR 167-50 postiže preciznost od 0.964 i opoziv od 0.942 za minimalan broj susjeda 75.

6.2 Algoritam

Opisat ćemo korake konstruiranog algoritma za detekciju i praćenje dlana, te brojanje prstiju.

- Pomoću prethodno treniranog kaskadnog klasifikatora detektiramo sve potprozore ulazne slike koji su klasificirani kao dlan. Potprozori su dani vektorom uspravnih pravokutnika `<vector<Rect>` gdje je svaki `Rect` određen koordinatama gornjeg ljevog kuta na ulaznoj slici i svojom širinom i dužinom u pikselima.

- Funkcija `pozadina()` uzima kao argument ulaznu sliku i vraća sliku pomaka na sceni. Za eliminaciju pozadine koristimo `MOG` algoritam koji je u OpenCV-u implementiran klasom `BackgroundSubtractorMOG`.



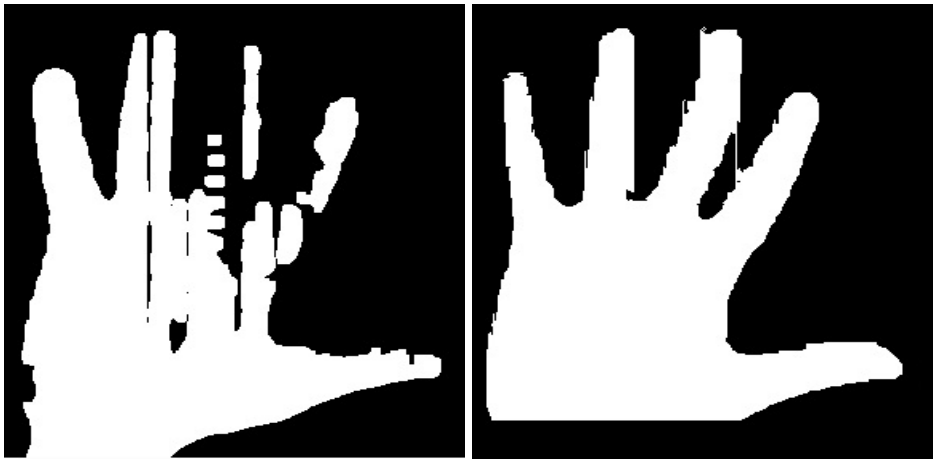
Slika 6.8: Pomak nakon oduzimanja pozadine od slike u boji (gore lijevo), nakon oduzimanja od sive slike (gore desno), nakon oduzimanja od sive slike sa ekvaliziranim histogramom (dolje lijevo), nakon oduzimanja i ekvalizacije histograma za svaki kanal RGB slike, te sumiranja u jednu sliku (dolje desno).

Da bismo što bolje eliminirali pozadinu, ulaznu sliku dijelimo u tri kanala i na svakome od njih provodimo oduzimanje pozadine. Na dobivenoj slici pomaka za svaki kanal provodimo morfološke operacije pomoću funkcije `morph()`. Slika pomaka se dobije sumiranjem svih triju kanala.

- Prisustvo dlana na nekom potprozoru podrazumijeva postojanje pomaka, jer pretpostavljamo da na početnoj pozadini nema dlana. Među kandidatima za dlan tražimo potprozor na kojem postoji najveći pomak. Gdje nema pomaka, vrijednost piksela je 0, pa gledamo potprozor u kojem ima najviše piksela većih od 0. Također primjenjujemo graničnu vrijednost te odbacujemo potprozor s najviše piksela većih od 0, ako je broj takvih piksela manji od granične vrijednosti. Preostali potprozor proglašavamo dlanom, ukoliko nije preostao niti jedan protprozor, algoritam kreće iz početka.

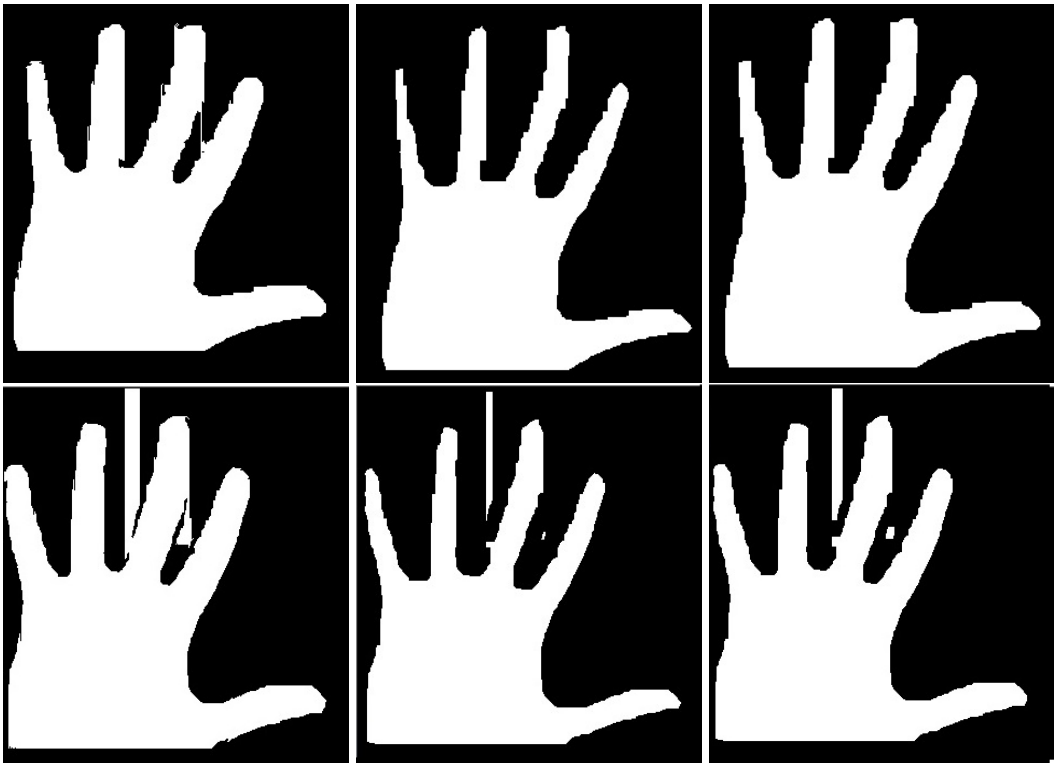
- Za potprozor koji sadrži dlan računamo prosječne koordinate i dimenzije za posljednje tri sličice kako bismo stabilizirali pravokutnik.

- Uzimamo dio slike za prosječni potprozor dlana te uz pomoć *flood-fil* tehnike pronalazimo binarno sliku dlana. Temelji se na razlici susjednih piksela u boji. Izabere se maksimalna i minimalna razlika u prostoru boja, vrsta povezanosti piksela[2] i sidrišne točke gdje započinje uspoređivanje. Funkcija u inicijaliziranu strukturu `Mat` napunjenu vrijednostima 0 i koja je za dva retka i stupca veća od potprozora, bilježi piksele povezane piksele. Dlan je širi u donjem i otprilike prsti čine polovinu duljine dlana, stoga za sidrišnu točku uzimamo središte pravokutnika pomaknuto ulijevo i prema dolje za četvrtinu širine i visine potprozora. Na dobivenoj slici provodimo morfološke opreacije uz pomoć funkcije `morph()`.



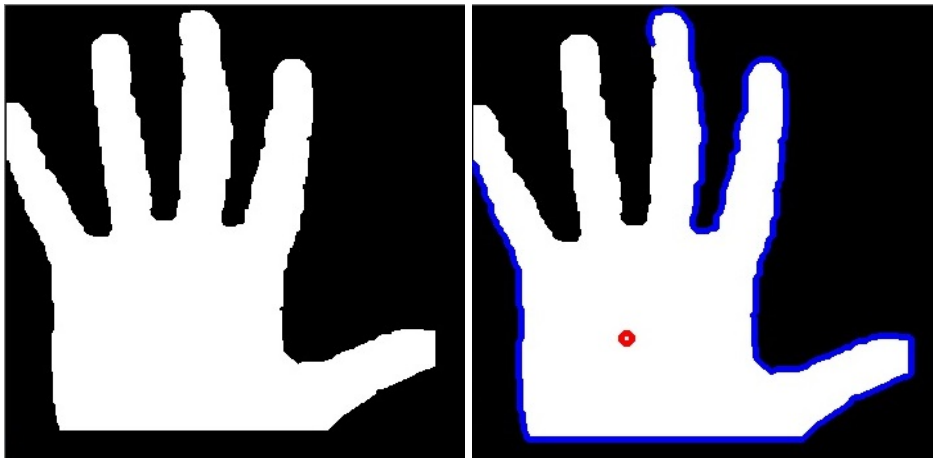
Slika 6.9: Binarna slika dlana oduzimanjem pozadine (lijevo) i binarna slika dlana nastala korištenjem *flood fill* tehnike. Ponekad oduzimanjem pozadine dobijemo sliku dlana kojoj nedostaju dijelovi iza kojih je pozadina slična dlanu, stoga koristimo *flood fill*.

[2] 4-connectednes, 8-connectedness



Slika 6.10: Desno su slike nastale *flood fill* tehnikom. Vidimo da postoje mostovi između prstiju i kod gornje slike vertikalna linija, nastali zbog sličnosti tih piksela sa pikselima dlana. Traženjem kontura na takvoj slici ne bismo dobili konturu ruke. Za uklanjanje takvih smetnji prvo dva puta erodiramo sliku s maskom veličine 5 piksela (slike u sredini), a zatim dilatiramo s maskom veličine 3 piksela.

- Za sliku popunjenu *flood fill* tehnikom tražimo rubove pomoću Cannyjevog detektora rubova. Prije traženja rubova u prethodno dobivenu sliku dodajemo po dva reda pikesla gore, dolje, lijevo i desno sa vrijednostima piksela 0. Zbog piksela dlana koji su uz rub potprozora ne dobijemo zatvorenu konturu, pa prethodnim postupkom rješavamo taj problem.



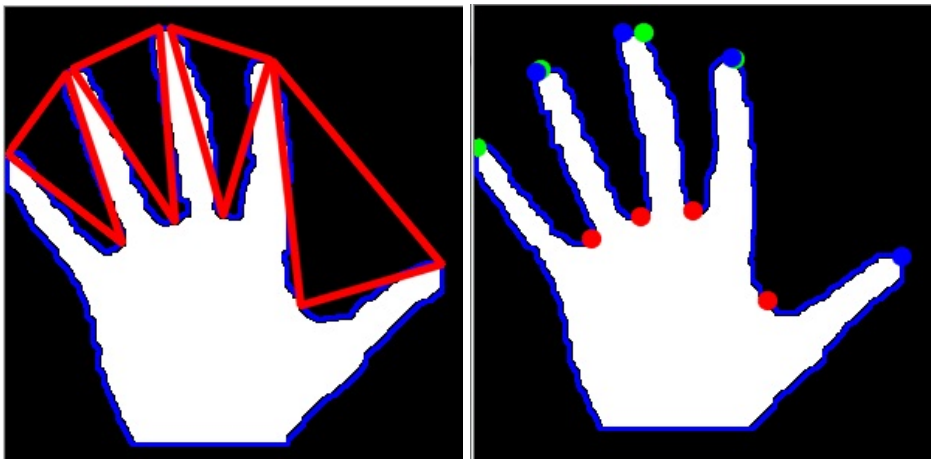
Slika 6.11: Primjer kada je prst uz rub potprozora. Desno je prikazana kontura plavom bojom. Crvena točka na dlanu je sidrišna točka za *flood fill*.

Iz dobivenih rubova određujemo konture OpenCV funkcijom `findContures()`. Pronalazimo najveću konturu tako da tražimo konturu s najviše točaka. Također provjeravamo da li je broj točaka veći od zadane granične vrijednosti, ako nema takvih kontura, algoritam se vraća na početak.

- Koristeći `HuMoments` i OpenCV funkciju `moments()` računamo momemte slike, te određujemo središte mase piksela na slici dobivenoj *flood fill* tehnikom.
- Za pronađenu konturu određujemo konveksnu ljusku OpenCV funkcijom `convexHull()`, te promatramo razlike između konture i ljuske.

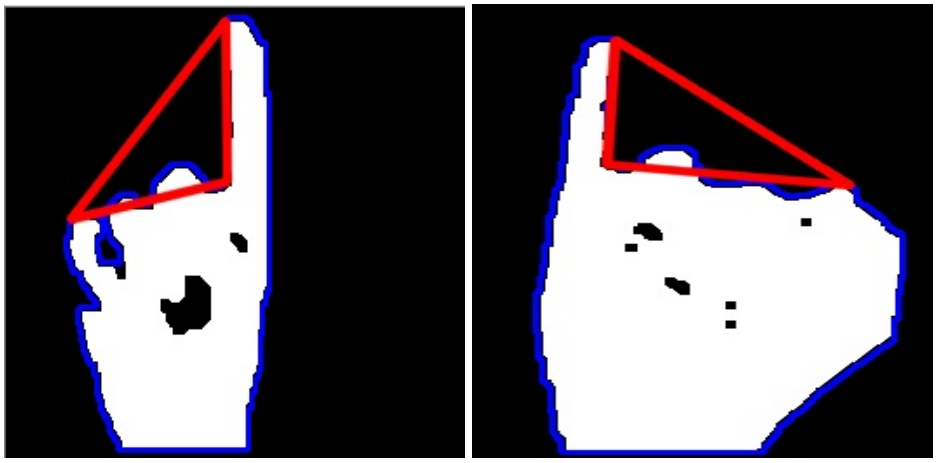
Slika 6.12: Konveksna ljuska prikazana zelenom bojom.

Razlike nazivamo defektima konveksnosti u daljnjem tekstu defekti. Promatrajući dlan, vidimo da su defekti među prstima relativno sličnog oblika što nam omogućuje da ih lakše odredimo. U OpenCV-u defekte tražimo funkcijom `convexityDefects()`. Nađeni defekti se pohranjuju kao vektor struktura `Vec4i`. Svaki defekt je određen s četiri elementa, redom: početna točka, završna točka, najdublja točka i dubina. Defekti između prstiju imaju najveću dubinu od defekata na dlanu, pa izdvajamo defekte čija je dubina veća od zadane granične vrijednosti. Također, defekti su oblikom veoma slični jednakokračnom trokutu pa ih eliminiramo i po omjeru udaljenosti $\frac{d(pocetna,najdublja)}{d(zavrsna,najdublja)}$.



Slika 6.13: Defekti preostali nakon filtriranja (desno), točke defekata početna - zelena, najdublja - crvena i završna -plava (lijevo).

Broj takvih defekata na dlanu je za jedan manji od broja prstiju, a vršci prstiju su određeni početnom i završnom točkom defekta. Ako postoje takvi defekti, pomoću strukture `tocke_dlana` pamtimo početne i završne točke defekata te broj prstiju koji je jednak broju defekata uvećanom za jedan. Defekti slični onima između prstiju, pojavljuju se i kada je prikazan jedan prst. Možemo ih razlikovati po dubini, no za konstantnu graničnu vrijednost, defekti jednog prsta na manjoj skali imaju veću dubinu od granične vrijednosti. Rješenje problema je množenje granične vrijednosti koeficijentom koji ovisi o razlučivosti potprozora detektiranog dlana.



Slika 6.14: Primjenom granične vrijednosti koja ovisi o razlučivosti potprozora dlana filtriramo slučajeve kada je prikazan jedan prst

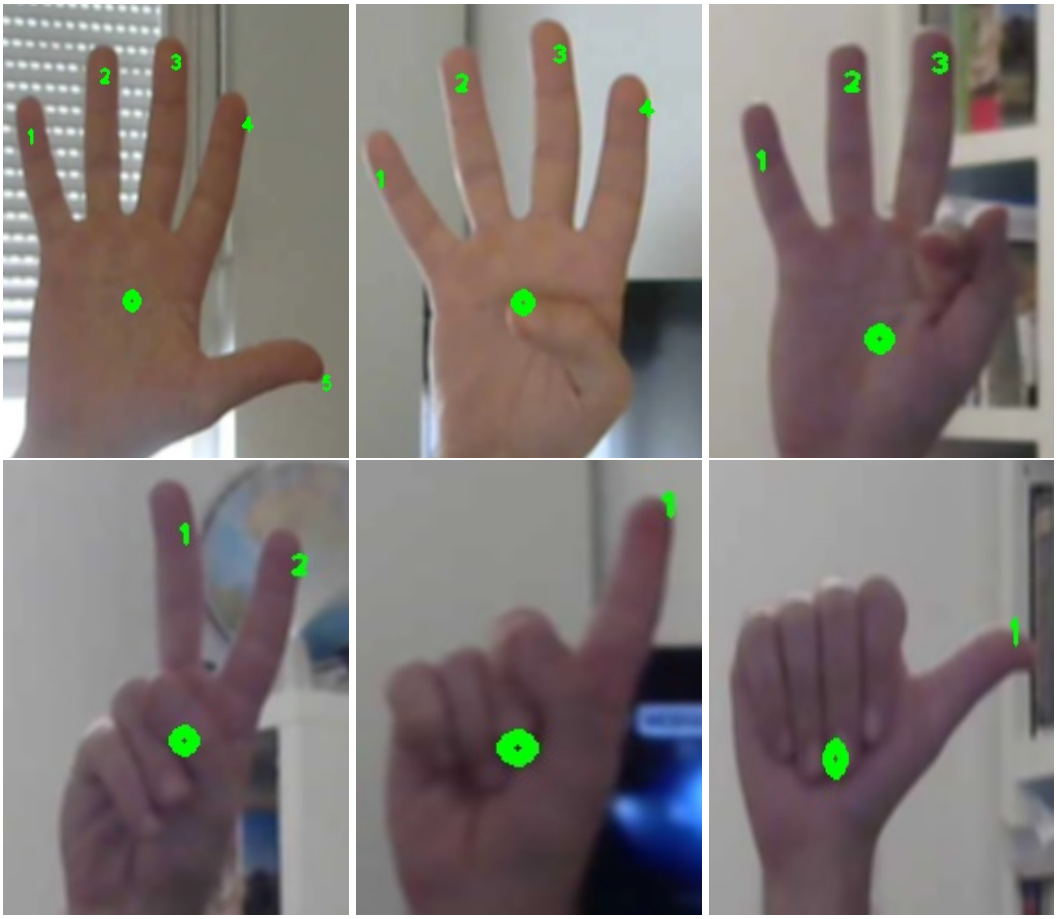
Specijalni slučajevi su kada niti jedan defekt nije zadovoljio uvijete. Postoje dvije mogućnosti, da je prikazan jedan prst ili ih uopće nema. Ispitujemo da li na konturi postoji neka točka koja koja je udaljena od centra mase za zadanu graničnu vrijednost. Ako postoji, tada je ta točka vrh prsta te ju pamtimo u strukturi `tocke_dlana.jedan_prst`, u suprotnom bilježimo da je broj prstiju 0 i izlazimo iz funkcije. Graničnu vrijednost udaljenosti od centra također množimo s koeficijentom.

- Slijedi označavanje prstiju sa brojevima. Provjeravamo koliko ima prstiju:

  - Ako je broj jednak 0 na centar pišemo 0.

  - Ako je broj jednak 1 tada pišemo 1 u točki `tocke_dlana.jedan_prst`.

  - Ako je broj prstiju veći od 1, moramo poredati defekte s lijeva na desno kako bismo mogli pravilno, uzastopnim brojevima, označiti prste. Poredamo ih s obzirom na x koordinatu završne točke defekta. Nakon toga, računamo koordinate točke koja je na polovištu udaljenosti početne točke defekta i završne točke idućeg defekta, ako nema idućeg defekta, tada se broj zapisuje u završnoj točci posljednjeg defekta. Brojevi se zapisuju na izlaznu sliku.



Slika 6.15: Primjeri pravilno detektiranih prstiju.

S obzirom da prepoznavanje vršimo za frontalni dlan, kako bismo brojali prste kada nije prikazan dlan, algoritam je podešen da nakon što detektirano dlan nestane sa scene, potprozor ostaje još 30 sličica. Ako je unutar tog vremena detektiran novi dlan, potprozor se pomiče na tu lokaciju.
Cijeli kod je dostupan u privitku. U njemu se nalaze korištene granične vrijednosti za korištena svojstava. Komentarima su označeni pojedini dijelovi. Korišteni su dodatni objekti za sučelje i funkcije crtanja korištene za prezentaciju rezultata. Dobivanje binarne slike dlana pomoću *flood fill* tehnike je vrlo uspješno u uvjetima dobrog osvjetljenja kao na primjer danje svijetlo. Prilikom slabog osvjetljenja pozadinski pikseli nemaju dovoljnu razliku u kontrastu od piksela dlana te se lažno detektiraju kao dio dlana. Jedan od razloga je i pikselizacija slike u lošim uvjetima osvjetljenja kao posljedica kvalitete senzora.

Vrijeme detekcije na video zapisu uvelike ovisi o razlučivosti pozitivnih primjera korištenih prilikom treninga. Klasifikatori trenirani na primjerima razlučivosti 24x24 su preko dva puta brži u detekcije od klasifikatora treniranih na pozitivnim primjerima razlučivosti 50x50. Detekcija s HAAR značajkama je između 10 i 18 posto sporija od detekcije s LBP značajki. SVi klasifikatori kao i cijeli program se izvode u realnom vremenu. Najmanji broj sličica u sekundi je 15 pri radu programa s kaskadnim klasifikatorom HAAR 167-50.

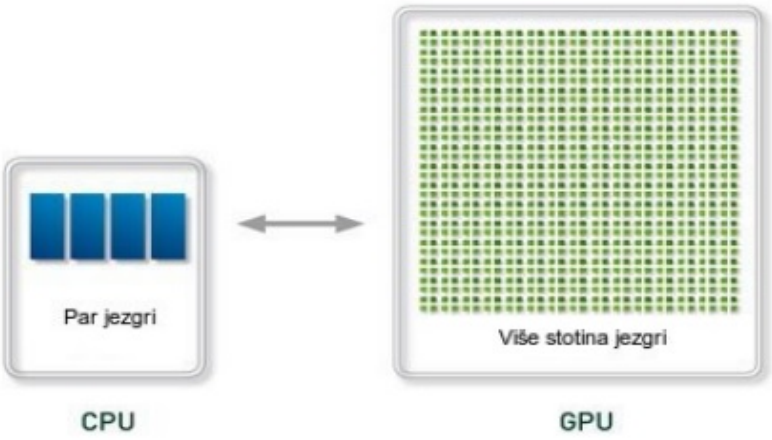
Navest ćemo neka od mogućih unaprijeđenja algoritma i klasifikatora:

- Skup negativnih primjera je dobiven snimanjem pomoću web kamere pa sadrži dosta dosta sličnih primjera. Izbacivanjem takvih primjera i dodavanjem novih bi se povećala kvaliteta skupa.
- Dobivanje binarne slike pomoću *flood fill* tehnike uvelike ovisi o osvjetljenju. Kalibracijom parametara kamere ili kamerom s kvalitetnijim senzorom mogli bismo poboljšati osjetljivost na osvjetljenje.
- Trenirati kaskadni klasifikator za dlan lijeve ruke zrcaljenjem pozitivnih primjera.
- Implementirati upravljanje kursorom pomoću praćenja pokreta dlana.
- Upravljanje gestama. Za svaku gestu koristiti više primjera te pratiti kretanje dlana i pamtiti pomoću opisnika. Zatim strojnim učenjem stvoriti klasifikator za svaku gestu.

Poglavlje 7

GPU ubrzanje

Obrada slike zahtijeva velik broj računalnih operacije. Operacije vršimo na pikselima neke slike, a rezultati operacija su često međusobno neovisni pa ih se može paralelizirati. Odvijaju u aritmetičko-logičkoj jedinici procesorske jedinke. Centralna procesorska jedinica - CPU je opće namjene te ima i upravljačku zadaću, pa sadrži manji broj aritmetičko-logičkih jedinica. Grafička procesorska jedinica - GPU je arhitekturom prilagođena obradi slike, stoga sadrži velik broj jednostavnijih procesorski jedinki.[13]



Slika 7.1: Usporedba arhitegture CPU i GPU.

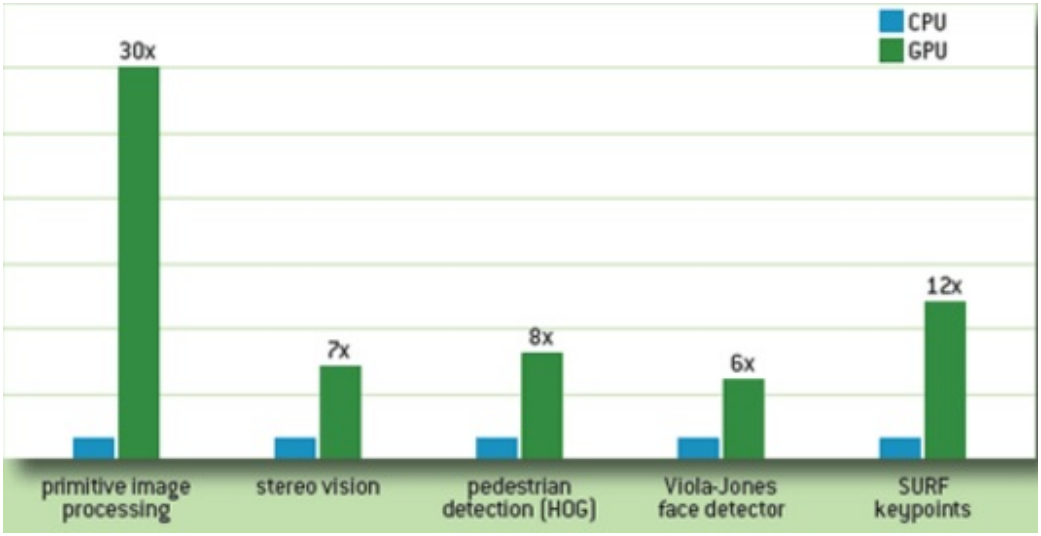
OpenCV biblioteka se izvršava na centralnoj procesorskoj jedinici ali ima dodatni GPU modul pomoću kojeg se može izvršavati na grafičkoj jedinici. Modul sadrži pokriva značajan dio funkcionalnosti OpenCV biblioteke i dalje se razvija.[11] Trenutno je implementiran za NVIDIA grafičke kartice koje imaju omogućen CUDA[1] Također je dostupna i

[1]Compute Unified Device Architecture.

OpenCL implementacija.
GPU modul sadrži posebnu klasu `GpuMat` koja služi pri obradi. Izvornu sliku reprezentiranu kao instanca klase `Mat` potrebno je reprezentirati kao instancu klase `GpuMat`. Ulaznu sliku `slika` reprezentiramo instancom `slika_gpu` klase na način `slika_gpu(slika)`. Klasa `GpuMat` se pohranjuje u memoriju grafičke jedinice te ne podržava direktan pristup podatcima.[11] Nakon obrade slike na GPU jedinici, potrebno ju je prebaciti u glavnu memoriju računala u obliku instance klase `Mat`, na primjer `slika(slika_gpu)`. Prikazat ćemo jednostavan primjer:

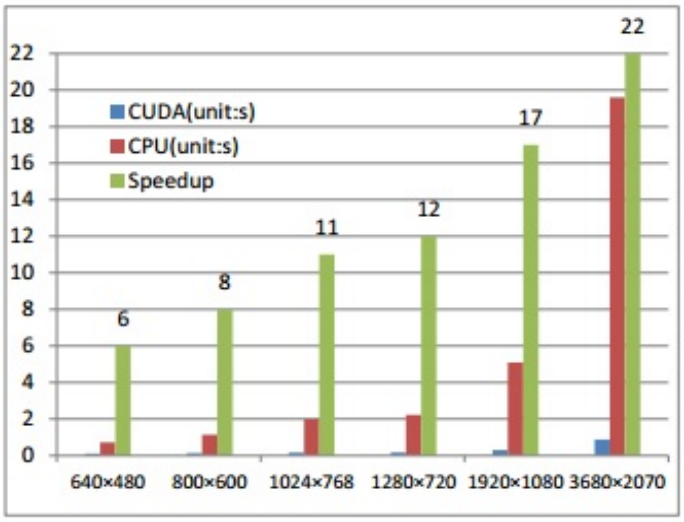
```
1 Mat slika = imread("slika.jpg");
2 gpu::GpuMat slika_gpu;
3 slika_gpu(slika);
4 gpu::GpuMat rezultat;
5 gpu::threshold(slika_gpu, rezultat, 128, CV_THRESH_BINARY);
6 slika(rezultat);
7 imshow("Prozor", slika);
8 waitKey();
```

Možemo izvršavati više funkcija na slici dok se nalazi u memoriji grafičke jedinice, odnosno ne moramo za svaku funkciju ponovno prebacivati sliku u memoriju. U novijim, 3.x verzijama OpenCV biblioteke, sintaksa je nešto drugačija. GPU modul je zamijenjen CUDA modulom.



Slika 7.2: Prikaz ubrzanja za pojedina područja.[11]

Ubrzanja su do čak 30 puta, ovisno o području. Za kaskadni klasifikator ubrzanje je i do 6 puta. Prema [9] ubrzanje za kaskadni klasifikator ovisi o razlučivosti ulazne slike te broju pozitivnih primjera korištenih pri treninku klasifikatora.



Slika 7.3: Vrijeme izvođenja na CPU i GPU jedinici pri različitoj razlučivosti ulazne slike. Zelenom bojom je prikazano ubrzanje, prema [9].

Bibliografija

[1] Daniel Lélis Baggio, *Mastering OpenCV with practical computer vision projects*, Packt Publishing Ltd, 2012.

[2] Gary Bradski i Adrian Kaehler, *Learning OpenCV: Computer vision with the OpenCV library*, " O'Reilly Media, Inc.", 2008.

[3] Samarth Brahmbhatt, *Practical OpenCV*, Apress, 2013.

[4] Chi hau Chen, *Computer vision in medical imaging*, sv. 2, World scientific, 2014.

[5] Roger N. Clark, *Notes on the Resolution and Other Details of the Human Eye*, 2014, `http://www.clarkvision.com/articles/eye-resolution.html`, posjećena 28.9.2016.

[6] Kenneth Dawson-Howe, *A practical introduction to computer vision with opencv*, John Wiley & Sons, 2014.

[7] P Jones, Paul Viola i Michael Jones, *Rapid object detection using a boosted cascade of simple features*, University of Rochester. Charles Rich, Citeseer, 2001.

[8] Marko Kosanović, *Metode kalibracija kamera*, Magistarska radnja, Fakultet elektrotehnike i računarstva, Sveučilište u Zagrebu, 2010.

[9] Ren Meng, Zhang Shengbing, Lei Yi i Zhang Meng, *CUDA-based real-time face recognition system*, Digital Information and Communication Technology and it's Applications (DICTAP), 2014 Fourth International Conference on, IEEE, 2014, str. 237–241.

[10] Sehgal Preeti, *Palm Recognition Using LBP and SVM*, International Journal of Information Technology & Systems **4** (2015), br. 1.

[11] Kari Pulli, Anatoly Baksheev, Kirill Kornyakov i Victor Eruhimov, *Real-time computer vision with OpenCV*, Communications of the ACM **55** (2012), br. 6, 61–69.

[12] Javier Ruiz-del Solar i Rodrigo Verschae, *Object detection using cascades of boosted classifiers*, 2006.

[13] Marijan Šuflaj, *Gesture recognition using GPU*, Disertacija, Sveučilište u Zagrebu Fakultet elektrotehnike i računarstva, Sveučilište u Zagrebu, 2014.

[14] Richard Szeliski, *Computer vision: algorithms and applications*, Springer Science & Business Media, 2010.

[15] OpenCV Developers Team, *OpenCV: About*, 2015, `http://opencv.org/about.html`, posjećena 15.3.2016.

[16] Wikipedia, *Autonomus Car*, 2015, `https://en.wikipedia.org/wiki/Autonomous_car`, posjećena 15.3.2015.

[17] Wikipedi, *OpenCV*, 2015, `https://en.wikipedia.org/wiki/OpenCVl`, posjećena 05.12.2015.

Sažetak

U ovom radu je dan uvid u problematiku računalnog vida. Objašnjen je računalni prikaz slike i osnove nastanka slike u kameri. Prikazan je sadržaj OpenCV biblioteke, dan kratki uvid u povijest razvoja i prikazane osnovne funkcije. Navedeni su algoritmi za otklanjanje šuma i detekciju rubova na slici, objašnjene su morfološke operacije te prikazan rad s konturama i detektorima kutova. Objašnjen je SIFT algoritam za rad s ključnim točkama, a zatim na primjeru uspoređen sa SURF algoritmom.
Prikazane su osnovne ideje klasifikacije objekata, a zatim je objašnjen algoritam kaskadnog klasifikatora. Objašnjene su, dostupne za kaskadni klasifikator u OpenCV-u, LBP i HAAR značajke. Također je naveden i SVM.

U praktičnom djelu je konstruiran i implementiran algoritam za prepoznavanje otvorenog dlana i brojanje prstiju. Objašnjen je trening korištenog kaskadnog klasifikatora uz pomoć OpenCV alata te prikazani rezultati treninga s LBP i HAAR značajkama na tri skupa s različitim brojem pozitivnih primjera te dvije razlučivosti 24x24 i 50x50 piksela. Iznesene su ideje za unaprjeđenje konstruiranog algoritma, a u zadnjem poglavlju je prikazano moguće ubrzanje putem GPU jedinice i dodatnog OpenCV modula.

Summary

This thesis presents an overview of computer vision. It gives basic insight of computer representation of image and formation of the image in the camera. Brief history of OpenCV along with contents of the library are shown. It also covers basic functions and structures. Algorithms for removing noise and edge detection in the image are shown and explanation of morphological operations with their application example is given. There are examples of contours extraction and corner detection also. SIFT key point detection and extraction is explained along with performance comparison to SURF detector.

The basic idea underlying classification of objects is shown along with explanation of cascade classifier algorithm. LBP and Haar features, which are available with OpenCV for cascade classifier detection, are presented. SVM is listed also.

Practical work covers construction and implemented of algorithm for open palm detection and finger counting. Training data preparation for cascade classifier training with OpenCV tool is explained. Classifiers are trained on three sets with different number of positive examples and two different resolutions. For every set of positive images, classifier is trained for both LBP and HAAR feature and performance of resulting classifiers is compared. Some ideas for improving constructed algorithm and cascade classifier are given and last chapter is shown the possible acceleration via GPU units and additional OpenCV modules.

Životopis

Moje ime je Dino Franić. Rođen sam 25.6.1989. godine u Metkoviću. Osnovno obrazovanje stekao sam u Osnovnoj školi Vladimir Nazor u Pločama, nakon toga se upisuje u Srednju školu fra Andrija Kačić Miošić, smjer opća gimnanzija. Po završetku srednje škole, 2008. godine upisujem se na Prirodoslovno-matematičk fakultet u Zagrebu. Dvije godine sam pohađao inženjerski smjer, nakon čega se prebacujem na edukacijski smjer. 2013. godine stečem zvanje prvostupnika i upisujem diplomski studij pri istom fakultetu, smjer edukacije matematike i informatike.

Dodatak A

glavni program

Program korišten za detekciju dlana i brojanje prstiju.

```
#include <opencv2/opencv.hpp>
#include <stdio.h>
#include <math.h>

using namespace std;
using namespace cv;

////granicne vrijednosti i pomocne varijable/////
int tresh_broj_piksela = 3000;          //minimalan broj piksela > 0 na slici pomaka
int rubovi_g = 10; int rubovi_G = 30;     //Canny treshold rubova, svejedno jer je binarna slika
int min_duljina_tresh = 120;          //minimalni broj tocaka u konturi
float treshol_omjer_stranica_max = 1.5;     //omjer stranica defekta
float treshol_omjer_stranica_min = 0.5;
float osnovni_treshold_za_dubinu_min = 19000.0, treshold_za_dubinu_min; //minimalna dubina
float osnovni_treshold_centar = 120, treshold_centar;          //minimalna udaljenost od centra
int hi_diff = 2, lo_diff = 1;                       //flood-fill razlika

int FPS = 0;
int delay = 30;

vector<Rect> dlan_kandidati;
vector<vector<Point>> konture;
vector<Vec4i> hierarchy;
Ptr<BackgroundSubtractor> pMOG_B, pMOG_G, pMOG_R;

///struktura za spremanje dlana
struct tocke_dlana {
```

```
28   Rect okvir_dlan;
29   int broj_prstiju;
30   int prazno = 0;
31   Point centar;
32   Point jedan_prst;
33   vector<vector<Point>> prsti;
34 };
35 tocke_dlana tocke;
36
37 ///potrebno za racunanje prosjecnog potprozora
38 vector<int> prosjek_lista__rect_x(3), prosjek_lista__rect_y(3);
39 vector<int> prosjek_lista__rect_w(3), prosjek_lista__rect_h(3);
40 vector<Point> prosjek_lista__centar;
41 int brojac_prosjek = 0;
42
43 ///funkcije
44 void nadi_tocke(Mat ulazna_slika, vector<Rect> dlan_kandidati, bool
       crtanje_ljuske, bool crtanje_defekata, bool crtanje_tocaka);
45 vector<Rect> pronadi_kandidate(Mat ulazna_slika);
46 Mat flod_fill(Mat ulazna_slika);
47 void broji_prste(Mat dlan, tocke_dlana tocke);
48 void morph(Mat morph_ulaz, bool redoslijed, int maska);
49 Rect average_rect(Rect roi_dlan);
50 int prosjek_int(vector<int> ulaz);
51
52 String cascade_dlan = "cascade_dlan.xml";
53 CascadeClassifier trazenje_dlan;
54
55 ///udaljenost tocaka
56 float euklid_udalj(Point& p, Point& q) {
57   Point diff = p - q;
58   return cv::sqrt(diff.x*diff.x + diff.y*diff.y);
59 }
60
61 ///za sortiranje defekata s lijeva na desno
62 struct slijedom {
63   inline bool operator()(const vector<Point>& toc1, const vector<Point>&
       toc2) { return (toc1[1].x < toc2[1].x); }
64
65 };
66
67
68 ////////// sucelje ////////////
69 class interface {
70   struct gumb {
71     int redni_broj;
72     int kliknuto = 0;
```

```
73      string natpis;
74      Rect pomocni_rect;
75      bool* stanje;
76    };
77    vector<gumb> svi_gumbovi;
78    int pozicija_x, broj_rect;
79    int pozicija_y;
80    Mat background;
81    string window;
82  public:
83    interface(int, int, string, bool);
84    void dodaj_rect(bool* &on_off, string tekst);
85    void update_tekst(gumb za_update);
86    void klik(int x, int y);
87  };
88
89  interface::interface(int x, int y, string name, bool move) {
90    pozicija_x = x;
91    pozicija_y = y;
92    window = name;
93    broj_rect = 0;
94    background = Mat(250, 250, CV_8UC3, Scalar::all(255));
95    if (move) moveWindow(window, pozicija_x, pozicija_y);
96    resizeWindow(window, 250, 250);
97    cout << background.size() << endl;
98  }
99
100 void interface::update_tekst(gumb za_update) {
101   int zapis_x = 80 * (za_update.redni_broj % 3) + 10;
102   int zapis_y = 30;
103   if (za_update.redni_broj >= 3) zapis_y = 110;
104   cout << "redni broj " << za_update.redni_broj << endl;
105   if (za_update.kliknuto == 1) {
106     //*za_update.stanje = *za_update.stanje == true ? "false" : "true";
107     if (*za_update.stanje == false) {
108       *za_update.stanje = true;
109     }
110     else {
111
112       *za_update.stanje = false;
113     }
114   }
115
116   string off_on;
117   off_on = *za_update.stanje == true ? "true" : "false";
118   int baseline = 0;
```

```
119   rectangle(background, za_update.pomocni_rect, Scalar(0, 0, 0),
      CV_FILLED, 8);
120   putText(background, za_update.natpis, Point(zapis_x, zapis_y),
      FONT_HERSHEY_PLAIN, 0.8, CV_RGB(255, 255, 255), 1, 8);
121   putText(background, off_on, Point(zapis_x + 10, zapis_y + 30),
      FONT_HERSHEY_PLAIN, 0.8, CV_RGB(255, 255, 255), 1, 8);
122   imshow(window, background);
123   za_update.kliknuto = 1;
124 }
125
126 void interface::dodaj_rect(bool* &on_off, string tekst) {
127   int pozicija_rect_y = 10;
128   if (broj_rect > 6) {
129     cout << "Nema mjesta" << endl;
130     return;
131   }
132   int pozicija_rect_x = 80 * (broj_rect % 3) + 10;
133   if (broj_rect >= 3) pozicija_rect_y = 90;
134   Rect pomocni = Rect(pozicija_rect_x, pozicija_rect_y, 70, 70);
135   rectangle(background, pomocni, Scalar(0, 0, 0), CV_FILLED, 8);
136   gumb pomocni_gumb;
137   pomocni_gumb.natpis = tekst;
138   pomocni_gumb.stanje = on_off;
139   pomocni_gumb.redni_broj = broj_rect;
140   pomocni_gumb.pomocni_rect = pomocni;
141   svi_gumbovi.push_back(pomocni_gumb);
142   update_tekst(svi_gumbovi[broj_rect]);
143   broj_rect++;
144 }
145
146 void interface::klik(int x, int y) {
147   for (int i = 1; i <= broj_rect; i++) {
148     bool klik_y;
149     bool klik_x = (((x >= (80 * ((i - 1) % 3) + 10))) && ((x <= (80 *
      (((i - 1) % 3) + 1) + 10))));
150     if (i < 4) klik_y = (y <= 90) && (y >= 10);
151     else klik_y = (y >= 90) && (y <= 160);
152     int a = (80 * (i % 3) + 10);
153     if (klik_x && klik_y) {
154       svi_gumbovi[i - 1].kliknuto = 1;
155       update_tekst(svi_gumbovi[i - 1]);
156     }
157   }
158
159 }
160 interface sucelje(1200, 700, "GUI", true);
161 ////kraj sucelja/////
```

```
162
163 ////call back funkcija za sucelje/////
164 void CallBackFunc(int event, int x, int y, int flags, void* userdata)
165 {
166   if (event == EVENT_LBUTTONDOWN)
167   {
168     sucelje.klik(x, y);
169   }
170 }
171
172 /////// varijable za sucelje /////////
173 bool crtanje_ljuske_g = false;
174 bool crtanje_defekata_g = false;
175 bool crtanje_tocaka_g = false;
176 bool update_background = false;
177
178 bool* ptr_backg = &update_background;
179 bool* ptr_c_lj = &crtanje_ljuske_g;
180 bool* ptr_c_d = &crtanje_defekata_g;
181 bool* ptr_c_t = &crtanje_tocaka_g;
182
183 /////////main////////////
184 int main() {
185   if (!trazenje_dlan.load(cascade_dlan)) { //ucitavanje trenirane
      kaskade
186     cout << "Greska pri ucitavanju kaskade\n" << endl;
187     waitKey();
188     return -1;
189   }
190   namedWindow("Izlaz");
191   moveWindow("izlaz", 100, 100);
192   namedWindow("GUI");
193   moveWindow("GUI", 1200, 700);
194
195   //sucelje
196   setMouseCallback("GUI", CallBackFunc, NULL);
197   sucelje.dodaj_rect(ptr_c_lj, "K ljuska");
198   sucelje.dodaj_rect(ptr_c_d, "Defekti");
199   sucelje.dodaj_rect(ptr_c_t, "Tocke");
200   sucelje.dodaj_rect(ptr_backg, "Background");
201
202   //background subtractor za svaki kanal
203   pMOG_B = new BackgroundSubtractorMOG(10, 5, 0.1);
204   pMOG_G = new BackgroundSubtractorMOG(10, 5, 0.1);
205   pMOG_R = new BackgroundSubtractorMOG(10, 5, 0.1);
206
```

```
207   VideoCapture kamera(0); // "videodlan2.avi");//0); //inicijalizacija
      video ulaza
208     kamera.set(CV_CAP_PROP_FRAME_WIDTH, 1280);
209     kamera.set(CV_CAP_PROP_FRAME_HEIGHT, 720);
210
211     if (!kamera.isOpened()) {
212       cout << "Greska pri pokretanju videa" << endl;
213       waitKey();
214       return -1;
215     }
216
217     Mat ulazna_slika;
218
219     while (kamera.isOpened() && (waitKey(1) != 27)) {
220       kamera >> ulazna_slika;
221       if (!ulazna_slika.empty()) {
222         vector<Rect> dlan_kandidati;
223         GaussianBlur(ulazna_slika, ulazna_slika, Size(5, 5), 2);
224         dlan_kandidati = pronadi_kandidate(ulazna_slika);
225         nadi_tocke(ulazna_slika, dlan_kandidati, crtanje_ljuske_g,
      crtanje_defekata_g, crtanje_tocaka_g);
226         broji_prste(ulazna_slika(tocke.okvir_dlan), tocke);
227         imshow("Izlaz", ulazna_slika);
228       }
229     }
230       cout << "Kraj " << endl;
231     return 0;
232 }
233
234
235 Mat pozadina(Mat ulazna_slika) {  //funkcija za oduzimanje pozadine
236   Mat chB, chG, chR, back_chB, back_chG, back_chR, foreground;
237   vector<Mat> kanali(3);
238
239   split(ulazna_slika, kanali);
240   chB = kanali[0];
241   chG = kanali[1];
242   chR = kanali[2];
243
244   equalizeHist(chB, chB);
245   equalizeHist(chG, chG);
246   equalizeHist(chR, chR);
247
248   pMOG_B -> operator()(chB, back_chB, 0.0);
249   pMOG_G -> operator()(chG, back_chG, 0.0);
250   pMOG_R -> operator()(chR, back_chR, 0.0);
251
```

```
252   morph(back_chB, 3, 3);
253   morph(back_chG, 3, 3);
254   morph(back_chR, 3, 3);
255
256   foreground = back_chB + back_chG + back_chR;
257
258   if (update_background) {  //ako je pritisnuto na sucelju
259     pMOG_B = new BackgroundSubtractorMOG(10, 5, 0.04);
260     pMOG_G = new BackgroundSubtractorMOG(10, 5, 0.04);
261     pMOG_R = new BackgroundSubtractorMOG(10, 5, 0.04);
262     update_background = false;
263   }
264   return foreground;
265 }
266
267 vector<Rect> pronadi_kandidate(Mat ulazna_slika) {     //detekcija
      potprozora klasifikatorom
268   Mat siva_slika;
269   dlan_kandidati.clear();
270   cvtColor(ulazna_slika, siva_slika, CV_BGR2GRAY);
271   equalizeHist(siva_slika, siva_slika);
272   trazenje_dlan.detectMultiScale(siva_slika, dlan_kandidati, 1.05, 40,
      0, Size(120, 120), Size(600, 600));  //metoda za detekciju
273   return dlan_kandidati;
274 }
275
276
277 void nadi_tocke(Mat slika_ulaz, vector<Rect> dlan_kandidati, bool
      crtanje_ljuske, bool crtanje_defekata, bool crtanje_tocaka) {
278   tocke.prsti.clear();
279   Mat rubovi;
280   int indeks_najvece = 0;
281   bool postojeci_pogodni;
282   Mat foreground = pozadina(slika_ulaz); //slika pomaka za potprozor
283   vector<Vec4i> pogodni_defekti;
284
285   /////trazenje pravokutnika koji sadrzi dlan na temelju kolicine
      piksela/////
286   int broj_piksela, max_broj_piksela = 0, ind_najbolji_kandidat = 0,
      brojac = 0;
287   for (brojac; brojac < dlan_kandidati.size(); brojac++) {
288     broj_piksela = countNonZero(foreground(dlan_kandidati[brojac]));
289     if (broj_piksela > max_broj_piksela) {
290       max_broj_piksela = broj_piksela;
291       ind_najbolji_kandidat = brojac;
292     }
293   }
```

```
294   if (max_broj_piksela < tresh_broj_piksela) {
295     if (!(FPS == 0 || FPS % delay == 0) && dlan_kandidati.size() == 0) {
296       FPS++;
297       FPS = FPS % delay;
298       cout << "delay " << endl;
299     }
300     else {
301       tocke.broj_prstiju = 0;
302       tocke.prazno = 1;
303       return;
304     }
305   }
306   else {
307     FPS = 1;
308     tocke.okvir_dlan = dlan_kandidati[ind_najbolji_kandidat];
309   }
310   //rectangle(slika_ulaz, tocke.okvir_dlan, Scalar(0, 255, 0), 2);
311   tocke.okvir_dlan = average_rect(tocke.okvir_dlan); // radimo
        stabilizaciju potprozora
312
313   //floood fill
314   Mat flood, nacrtano;
315   flood = flod_fill(slika_ulaz(tocke.okvir_dlan));
316   morph(flood, 0, 5);
317   morph(flood, 1, 3);
318
319   //////////// HuMomments ////////////////
320   Moments mu;
321   mu = moments(flood, true);
322   Point centar;
323   centar = Point(mu.m10 / mu.m00, mu.m01 / mu.m00);
324   tocke.centar = centar;
325
326   ////uklanjanje rubnih piksela/////
327   Mat prazni_red(2, flood.cols, CV_8UC1, Scalar::all(0)), prazni_stupac(
        flood.rows, 2, CV_8UC1, Scalar::all(0));
328   flood.rowRange(0,1) = prazni_red;
329   flood.rowRange(flood.rows - 2, flood.rows - 1) = prazni_red;
330   flood.colRange(0,1) = prazni_stupac;
331   flood.colRange(flood.cols - 2, flood.cols - 1) = prazni_stupac;
332
333   Canny(flood, rubovi, rubovi_g, rubovi_G);    //rubovi
334   morph(rubovi, 1, 3);
335
336   findContours(rubovi, konture, hierarchy, CV_RETR_EXTERNAL,
        CV_CHAIN_APPROX_SIMPLE); //konture na slici rubova
337
```

```
338
339
340   ////trazenje pogodne konture/////
341   if (konture.size() == 0) {
342     tocke.broj_prstiju = 0;
343     tocke.prazno = 1;
344     return; //ako je premalo kontura
345   }
346   else {
347     double duljina_max = 0.0;
348     for (int k = 0; k < konture.size(); k++){
349       int duljina = konture[k].size();
350       if (duljina_max < duljina) {
351         duljina_max = duljina;
352         indeks_najvece = k;
353       }
354     }
355     if (duljina_max < min_duljina_tresh) {
356       tocke.broj_prstiju = 0;
357       tocke.prazno = 1;
358       return;
359     }
360   }
361
362   ////konveksna ljuska/////
363   vector<Point> ljuska;
364   convexHull(konture[indeks_najvece], ljuska, false);
365
366   ////trazenje defekata/////
367   vector<int> indeksi_ljuske;//indeksi zbog potrebe convexityDefects
368   convexHull(konture[indeks_najvece], indeksi_ljuske, false);//punjenje
       indeksa
369   bool postoje_defekti = (indeksi_ljuske.size() > 3); // ako je manje od
       3 nema defekata
370   if (postoje_defekti) {
371     vector<Vec4i> defekti;
372     vector<Vec4i> defekti_po_dubini;
373     convexityDefects(konture[indeks_najvece], indeksi_ljuske, defekti);
374
375     //eliminacija po dubini
376     int dubina;
377     for (int i = 0; i < defekti.size(); i++) {
378       dubina = defekti[(i)][3];
379       if ((dubina > treshold_za_dubinu_min) && (dubina <
      treshold_za_dubinu_min*1.5)) {
380         defekti_po_dubini.push_back(defekti[i]);
381         cout << "dubina " << dubina << endl;
```

```
382         cout << "rect size " << tocke.okvir_dlan.width << tocke.
    okvir_dlan.height << endl;
383       }
384     }
385
386     //eliminacija po omjeru stranica
387     for (int i = 0; i < defekti_po_dubini.size(); i++) {
388       int prva_t = defekti_po_dubini[i][0];        //indeksi tocaka
    defekta u konturi
389       int zadnja_t = defekti_po_dubini[i][1];
390       int najdublja_t = defekti_po_dubini[i][2];
391       bool sukladne = (((euklid_udalj(konture[indeks_najvece][
    najdublja_t], konture[indeks_najvece][prva_t]) / euklid_udalj(
    konture[indeks_najvece][najdublja_t], konture[indeks_najvece][
    zadnja_t])) < treshol_omjer_stranica_max)
392         && ((euklid_udalj(konture[indeks_najvece][najdublja_t], konture[
    indeks_najvece][prva_t]) / euklid_udalj(konture[indeks_najvece][
    najdublja_t], konture[indeks_najvece][zadnja_t])) >
    treshol_omjer_stranica_min)
393         );
394       if (sukladne) {
395         pogodni_defekti.push_back(defekti_po_dubini[i]);
396         vector<Point> tocka;
397         tocka.push_back(konture[indeks_najvece][zadnja_t]);
398         tocka.push_back(konture[indeks_najvece][prva_t]);
399         tocke.prsti.push_back(tocka);
400         tocke.broj_prstiju = tocke.prsti.size() + 1;
401         tocke.prazno = 0;
402       }
403     }
404
405     ////// ako nema pogodnih defekata
406     bool postoji_jedan_prst;
407     if (pogodni_defekti.size() == 0) {
408       float pom_udaljenost, pom_max = 0;
409       int indeks;
410       for (size_t i = 0; i < konture[indeks_najvece].size(); i++) {
411         pom_udaljenost = euklid_udalj(centar, konture[indeks_najvece][i
    ]);
412         if (pom_udaljenost > pom_max) {
413           pom_max = pom_udaljenost;
414           indeks = i;
415         }
416       }
417       if (pom_max > treshold_centar) {
418         tocke.jedan_prst = konture[indeks_najvece][indeks];
419         tocke.broj_prstiju = 1;
```

```
420         tocke.prazno = 0;
421       }
422       else {
423         tocke.broj_prstiju = 0;
424         tocke.prazno = 1;
425         return;
426       }
427     }
428
429     //////// crtanje ///////////////7
430     cvtColor(flood, nacrtano, CV_GRAY2BGR);      //da bi mogli crtati u
    boji
431                               //drawContours(nacrtano, konture,
    indeks_najvece, Scalar(255, 0, 0), 2, 8); //crtanje konture
432                               // crtanje pogodnih (tocaka) defekata
433     if (crtanje_tocaka) {
434       for (int i = 0; i < pogodni_defekti.size(); i++) {
435         circle(nacrtano, konture[indeks_najvece][pogodni_defekti[i][2]],
     3, Scalar(0, 0, 255), 5, CV_AA);
436         circle(nacrtano, konture[indeks_najvece][pogodni_defekti[i][1]],
     3, Scalar(0, 255, 0), 5, CV_AA);
437         circle(nacrtano, konture[indeks_najvece][pogodni_defekti[i][0]],
     3, Scalar(255, 0, 0), 5, CV_AA);
438       }
439     }
440
441     //crtanje defekata
442     Point pt, pt0;
443     if (crtanje_defekata) {
444       for (int i = 0; i < pogodni_defekti.size(); i++) {
445         pt0 = konture[indeks_najvece][pogodni_defekti[i][0]];
446         for (int j = 2; 0 <= j; j--)
447         {
448           pt = konture[indeks_najvece][pogodni_defekti[i][j]];
449           line(nacrtano, pt0, pt, Scalar(0, 0, 255), 3, CV_AA);
450           pt0 = pt;
451         }
452       }
453
454     }
455
456     //crtanje ljuske
457     if (crtanje_ljuske) {
458       pt0 = ljuska[ljuska.size() - 1];
459       for (int i = 0; i < ljuska.size(); i++)
460       {
461         pt = ljuska[i];
```

```
462         line(nacrtano, pt0, pt, Scalar(0, 255, 0), 3, CV_AA);
463         pt0 = pt;
464       }
465     }
466
467     //imshow("Nacrtano", nacrtano); //prikaz nacrtanoga
468   }
469   return;
470 }
471
472 /////brojevi prstiju/////
473 void broji_prste(Mat dlan, tocke_dlana tocke) {
474   if (tocke.prazno == 1) return;
475   circle(dlan, tocke.centar, 5, Scalar(0, 255, 0), 5, 8, 0);
476   int zadnji_el = tocke.prsti.size() - 1;
477   if (zadnji_el >= 1) sort(tocke.prsti.begin(), tocke.prsti.end(),
      slijedom());
478   if (tocke.prsti.size() != 0) {
479     putText(dlan, "1", Point(tocke.prsti[0][0].x, tocke.prsti[0][0].y +
      30), FONT_HERSHEY_PLAIN, 1.0, CV_RGB(0, 255, 0), 2.0);
480     for (int i = 0; i < zadnji_el; i++) {
481       if (euklid_udalj(tocke.prsti[i][1], tocke.prsti[i + 1][0]) < 50.0)
      {
482         stringstream broj_out;
483         broj_out << (i + 2);
484         Point mjesto = Point(tocke.prsti[i][1].x + (abs(tocke.prsti[i +
      1][0].x - tocke.prsti[i][1].x) / 2), tocke.prsti[i][1].y + (abs(
      tocke.prsti[i + 1][0].y - tocke.prsti[i][1].y) / 2) + 20);
485         putText(dlan, broj_out.str(), mjesto, FONT_HERSHEY_PLAIN, 1.0,
      CV_RGB(0, 255, 0), 2.0);
486       }
487     }
488     stringstream broj_out;
489     broj_out << (tocke.prsti.size() + 1);
490     putText(dlan, broj_out.str(), Point(tocke.prsti[zadnji_el][1].x,
      tocke.prsti[zadnji_el][1].y + 20), FONT_HERSHEY_PLAIN, 1.0, CV_RGB
      (0, 255, 0), 2.0);
491   }
492   else if (tocke.jedan_prst != Point(-1, -1)) {
493     putText(dlan, "1", tocke.jedan_prst, FONT_HERSHEY_PLAIN, 1.0, CV_RGB
      (0, 255, 0), 2.0);
494
495   }
496   else putText(dlan, "0", Point(30, 30), FONT_HERSHEY_PLAIN, 1.0, CV_RGB
      (0, 255, 0), 2.0);
497 }
498
```

```
499
500 ////Prosijek za potprozor/////
501 Rect average_rect(Rect roi_dlan) {
502   //ako je razlika starog i novog potprozora veca od 3 piksela
503   int razlika_x, razlika_y, razlika_w, razlika_h;
504   razlika_x = abs(roi_dlan.x - prosjek_int(prosjek_lista__rect_x));
505   razlika_y = abs(roi_dlan.y - prosjek_int(prosjek_lista__rect_y));
506     razlika_w = abs(roi_dlan.width - prosjek_int(prosjek_lista__rect_w))
      ;
507   razlika_h = abs(roi_dlan.height - prosjek_int(prosjek_lista__rect_h));
508   bool _pomak = (razlika_x > 3 && razlika_y > 3);
509   if (_pomak) {
510     prosjek_lista__rect_w[brojac_prosjek] = roi_dlan.width;
511     prosjek_lista__rect_h[brojac_prosjek] = roi_dlan.height;
512     prosjek_lista__rect_x[brojac_prosjek] = roi_dlan.x;
513     prosjek_lista__rect_y[brojac_prosjek] = roi_dlan.y;
514     brojac_prosjek++;
515     brojac_prosjek = brojac_prosjek % 3;
516   }
517   Rect izlazni_rect = Rect(prosjek_int(prosjek_lista__rect_x),
      prosjek_int(prosjek_lista__rect_y), int(float(prosjek_int(
      prosjek_lista__rect_w))*1.15 ), int(float(prosjek_int(
      prosjek_lista__rect_h))));
518
519   float koeficijent = float(izlazni_rect.width)/250;          //
      koeficijent
520   treshold_za_dubinu_min = osnovni_treshold_za_dubinu_min * koeficijent;
521   treshold_centar =osnovni_treshold_centar * koeficijent;
522   if (izlazni_rect.x + izlazni_rect.width < 1280 && izlazni_rect.y +
      izlazni_rect.height < 720 && izlazni_rect.y > 0) //provjera ako je
      greska u dimenziji
523     return izlazni_rect;
524   else
525     return roi_dlan;
526 }
527
528
529 ////funkcija za prosijek/////
530 int prosjek_int(vector<int> ulaz) {
531   int sum = 0;
532   int izlaz;
533   for (int i = 0; i < 3; i++) {
534     sum = sum + ulaz[i];
535   }
536   izlaz = sum / 3;
537   return izlaz;
538 }
```

```
539
540 //////funkcija za morfoloske operacije/////
541 void morph(Mat morph_ulaz, bool redoslijed, int maska) {
542   switch (redoslijed) {
543   case 0:
544     erode(morph_ulaz, morph_ulaz, getStructuringElement(MORPH_RECT, Size
      (maska, maska)));
545     erode(morph_ulaz, morph_ulaz, getStructuringElement(MORPH_RECT, Size
      (maska, maska)));
546     break;
547   case 1:
548     dilate(morph_ulaz, morph_ulaz, getStructuringElement(MORPH_RECT,
      Size(maska, maska)));
549     dilate(morph_ulaz, morph_ulaz, getStructuringElement(MORPH_RECT,
      Size(maska, maska)));
550     break;
551   case 3:
552     erode(morph_ulaz, morph_ulaz, getStructuringElement(MORPH_RECT, Size
      (maska, maska)));
553     erode(morph_ulaz, morph_ulaz, getStructuringElement(MORPH_RECT, Size
      (maska, maska)));
554     dilate(morph_ulaz, morph_ulaz, getStructuringElement(MORPH_RECT,
      Size(maska, maska)));
555     dilate(morph_ulaz, morph_ulaz, getStructuringElement(MORPH_RECT,
      Size(maska, maska)));
556     break;
557   case 4:
558     dilate(morph_ulaz, morph_ulaz, getStructuringElement(MORPH_RECT,
      Size(maska, maska)));
559     dilate(morph_ulaz, morph_ulaz, getStructuringElement(MORPH_RECT,
      Size(maska, maska)));
560     erode(morph_ulaz, morph_ulaz, getStructuringElement(MORPH_RECT, Size
      (maska, maska)));
561     erode(morph_ulaz, morph_ulaz, getStructuringElement(MORPH_RECT, Size
      (maska, maska)));
562     break;
563   }
564 }
565
566 //////flood-fill funkcija/////
567 Mat flod_fill(Mat ulazna_slika) {
568   Mat hsl;
569   ulazna_slika.copyTo(hsl);
570
571   GaussianBlur(hsl, hsl, Size(3, 3), 2);
572   imshow("HSL ", hsl);
573   Mat maska(Size(hsl.cols + 2, hsl.rows + 2), CV_8UC1, Scalar::all(0));
```

```
574   Point seed_tocka = Point((hsl.cols / 2) * 0.6, (hsl.rows / 2) * 1.40);
575   floodFill(hsl, maska, seed_tocka, Scalar(0, 0, 0), 0, Scalar(lo_diff,
        lo_diff, lo_diff), Scalar(hi_diff, hi_diff, hi_diff), 8 + (255 << 8)
        + CV_FLOODFILL_MASK_ONLY);
576   return maska;
577 }
```