Vizijski sustav za praćenje objekata prema obliku i boji

Vranić, Tibor

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Tibor Vranić

Zagreb, 2023.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

ZAVRŠNI RAD

Mentori:

Prof. dr. sc. Mladen Crneković, dipl. ing.

Student:

Tibor Vranić

Zagreb, 2023.

Izjavljujem da sam ovaj rad izradio samostalno koristeći znanja stečena tijekom studija i navedenu literaturu.

Zahvaljujem se sestri i roditeljima na neizmjernom strpljenju i podršci tijekom obrazovanja i pisanja rada, te svim kolegama i prijateljima na pruženoj pomoći. Posebice zahvaljujem Viti Gajdek na konstantnoj motivaciji i prijateljstvu tijekom studija.

Također zahvaljujem se mentoru prof. dr. sc. Mladenu Crnekoviću na ukazanom povjerenju, vodstvu i stručnosti koju ste dijelili sa mnom tokom izrade rada i preddiplomskog studija.

Tibor Vranić

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za završne i diplomske ispite studija strojarstva za smjerove:
proizvodno inženjerstvo, računalno inženjerstvo, industrijsko inženjerstvo i menadžment, inženjerstvo materijala i mehatronika i robotika

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa: **602 – 04 / 23 – 6 / 1**	
Ur.broj: **15 - 1703 - 23 -**	

ZAVRŠNI ZADATAK

Student: **Tibor Vranić** JMBAG: **0035227600**

Naslov rada na hrvatskom jeziku: **Vizijski sustav za praćenje objekata prema obliku i boji**

Naslov rada na engleskom jeziku: **A vision system for tracking objects by shape and color**

Opis zadatka:

Osnovni preduvjet za inteligentno ponašanje mobilnih robota je spoznaja o prostoru u kojem se kreću, klasifikacija situacije u kojoj se nalaze te posjedovanje strategija izvršavanja zadataka. Programiranje takvog sustava moguće je samo u programskim jezicima koji su orijentirani zadatku, a najbolje informacije takvim programskim jezicima daju vizijski sustavi.

Za potrebe vođenja mobilnih robota potrebno je vizijskim sustavom pratiti objekte prema obliku i boji, a podatke o prepoznatome serijskom komunikacijom slati na drugo računalo ili drugom programu.

U radu je potrebno:

- ostvariti funkciju istovremenog prepoznavanja i praćenja tri oblika i tri boje,
- informaciju o prepoznatom (oblik, boja, položaj i veličina) slati drugoj aplikaciji,
- druga aplikacija treba grafički prikazati rezultate prepoznavanja u realnom vremenu.

U radu je potrebno navesti korištenu literaturu i eventualno dobivenu pomoć.

Zadatak zadan:
30. 11. 2022.

Zadatak zadao:
Prof. dr. sc. Mladen Crneković

Datum predaje rada:
1. rok: 20. 2. 2023.
2. rok (izvanredni): 10. 7. 2023.
3. rok: 18. 9. 2023.

Predviđeni datumi obrane:
1. rok: 27. 2. – 3. 3. 2023.
2. rok (izvanredni): 14. 7. 2023.
3. rok: 25. 9. – 29. 9. 2023.

Predsjednik Povjerenstva:
Prof. dr. sc. Branko Bauer

SADRŽAJ

POPIS SLIKA

POPIS TABLICA

SAŽETAK

Osnovni uvjet inteligentnog ponašanja mobilnih robota je spoznaja o prostoru u kojem se kreću te sposobnost klasifikacije situacije u svrhu izvršavanja zadatka. Fokus rada će stoga bit postavljen na konkretnu implementaciju računalnog i strojnog vida u Pythonu.

Korištenjem bogatih biblioteka OpenCV i NumPy, Python nam omogućuje da emuliramo ljudski vizualni sustav. Ovime se otvara mogućnost pretvaranja vizualnih informacija u korisne uvide i unapređenja u područjima poput robotike, medicine i automatizacije.

Uz to, biti će objašnjen način na koji Python olakšava izradu korisničkog sučelja pomoću Tkinter biblioteke, omogućujući čak i manje iskusnim programerima da razviju intuitivna rješenja u svojim aplikacijama.

Koristeći prikladne metode i algoritme iz područja strojnog vida izrađena je programska aplikacija za detekciju objekata temeljem boje i oblika. Također je analizirana funkcionalnost ovakvog sustava u različitim uvjetima te ponuđeno rješenje na temelju kalibracije boja.

Ključne riječi: računalni vid, strojni vid, robotika, Python, Tkinter, NumPy, OpenCV

SUMMARY

The fundamental requirement for the intelligent behavior of mobile robots is their understanding of the space they move in and their ability to classify situations for the purpose of task execution. Therefore, the focus of this work will be on the concrete implementation of computer and machine vision in Python.

By using rich libraries like OpenCV and NumPy, Python allows us to emulate the human visual system. This opens up the possibility of transforming visual information into valuable insights and improvements in areas such as robotics, medicine, and automation.

Furthermore, the way Python facilitates the creation of user interfaces through the Tkinter library will be explained. This enables even less experienced programmers to develop intuitive solutions in their applications.

Using appropriate methods and algorithms from the field of machine vision, a software application for object detection based on color and shape has been developed. The functionality of such a system in different conditions has also been analyzed, and a solution based on color calibration has been proposed.

Key words: computer vision, machine vision, robotics, Python, Tkinter, NumPy, OpenCV

1. UVOD

U suvremenom dobu, domene računalnog vida, obrade slika i programiranja, ujedinile su se kako bi preoblikovale način na koji percipiramo i komuniciramo sa okolinom. Mobilni roboti, kao i ostali autonomni sustavi, zahtijevaju informaciju o svom okruženju kako bi uspješno klasificirali svoju situaciju i ponudili zadovoljavajuće rješenje na problem.

U tom kontekstu, koncepti poput modela boja, prostora boja te tehnika računalnog i strojnog vida su temelji za razvoj aplikacija prepoznavanja boja i oblika. Korištenjem alata za poboljšanje kvalitete slika, izvlačenje važnih informacija te isticanje bitnih karakteristika, postiže se efekt sličan našem vizualnom osjetilu.

A već postojeće i duboko razvijene biblioteke programa Python daju podlogu za povezivanje kompleksnih tehnika obrade slika i korisnika, omogućujući veću dostupnost i primjenu ovih tehnika u različitim domenama.

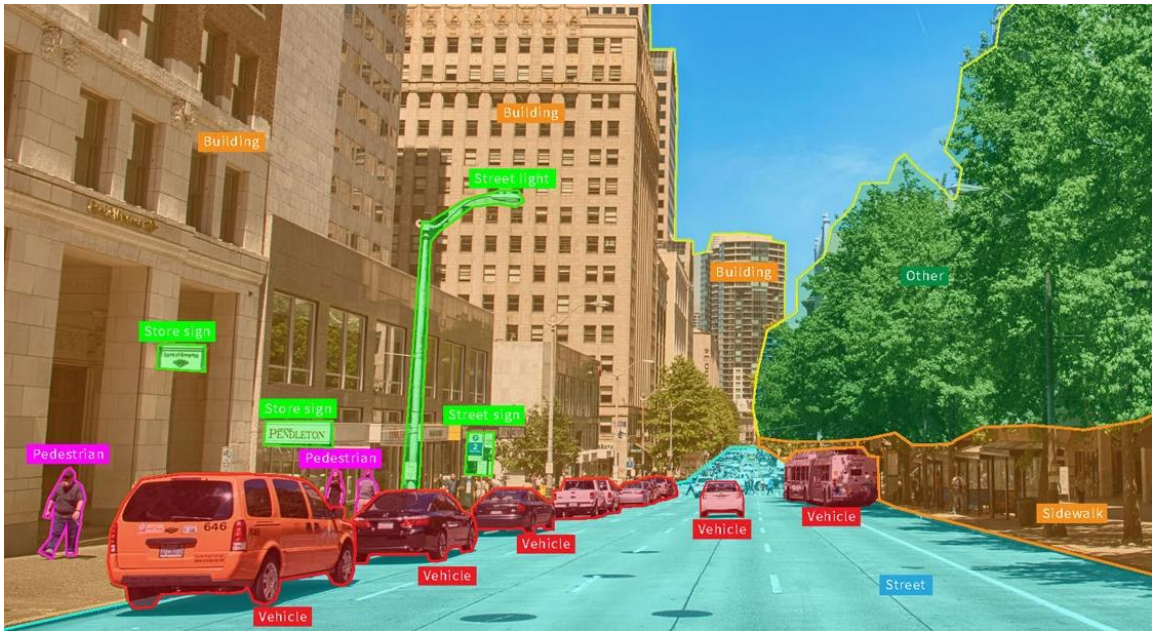
1.1. Strojni i računalni vid

Iako se često poistovjećuju, računalni vid (CV) i strojni vid (MV) su dva različita pojma koji opisuju slične tehnologije. Ključna razlika između njih leži u njihovom fokusu i primjeni.

Strojni vid se odnosi na sposobnost stroja ili računala da percipira i razumije svoju okolinu putem uređaja kao što su kamera, LIDAR i ultrazvučni senzori. Podrazumijeva korištenje računalnog vida za izvršavanje zadataka vizualnog sustava, a može biti usmjeren na navigaciju, praćenje kretanja i interakciju s okolinom u kontekstu autonomnih vozila, industrijske automatizacije, robotske percepcije itd. To se postiže korištenjem algoritmima za obradu podataka nad podatcima kao što su slike, videozapisi, 3D skenovi i informacije o dubini i zvuku.

Računalni vid je multidisciplinarno područje koje se fokusira na razvoj algoritama i tehnika za obradu i analizu slika i videozapisa kako bi se razumjelo vizualno okruženje. Ovaj koncept se temelji na ideji da računala mogu "vidjeti" svijet oko sebe na sličan način kao ljudski vid, koristeći tri ključne procesne komponente; akvizicija slike (*eng. Image acquisition*), obrada slike (*eng. Image processing*) te analiza i razumijevanje slike (*eng. Image analysis and understanding*). Koristi segmentaciju objekata, prepoznavanje oblika, boja, tekstura i još mnogo toga za primjenu u domenama prepoznavanja objekata, identifikacije lica, analize medicinskih snimaka, autonomne vožnje te industrijske inspekcije.

Računalni vid se može koristiti samostalno, bez potrebe da bude dio većeg strojnog sustava, dok sustav strojnog vida ne radi bez računala i specifičnog softvera u svojoj srži.[8]

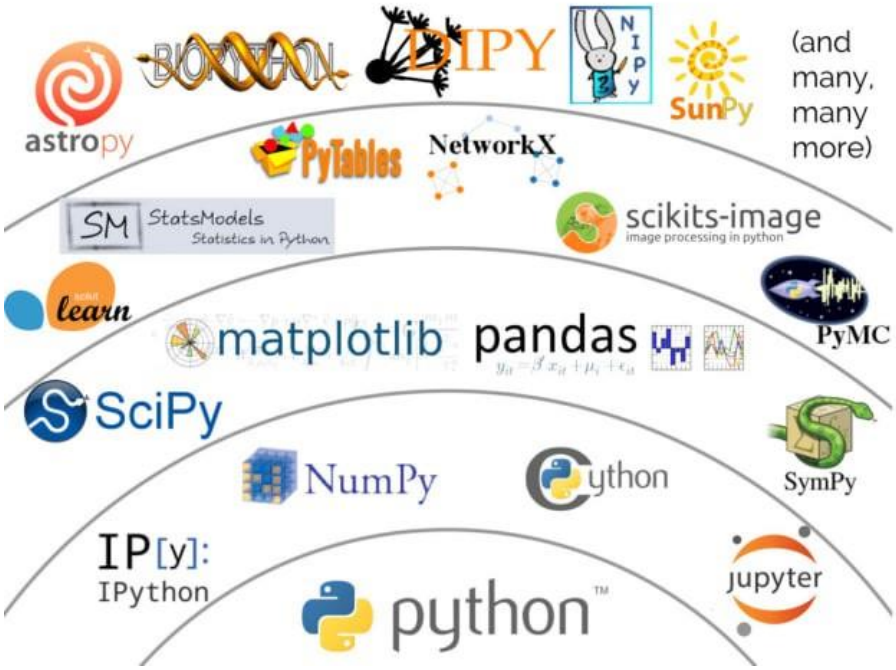


Slika 1. Računalni vid

1.2. Python

Python je objektno orijentirani programski jezik visoke razine koji se ističe svojom jednostavnom i čitljivom sintaksom, stoga je idealan za programere svih iskustvenih razina, od početnika do iskusnih profesionalaca. Jedna od najvećih prednosti Pythona je ogromna zajednica korisnika i entuzijasta za razvoj softvera što rezultira obiljem resursa, objašnjenja, vodiča i biblioteka za različite svrhe. Sve to ga čini izrazito korisnim i sposobnim u domenama web razvoja, desktop aplikacija, skripti, znanstvenih istraživanja, analize podataka, automatizacije, obrade prirodnog jezika (NLP), strojnog učenja i umjetne inteligencije.

Za lakše upravljanje okolinama (*eng. Enviroment*) i bibliotekama, pri izradi ovog rada je korišten Anaconda softverski paket . To je sveobuhvatna i integrirana okolina koja omogućuje programerima i znanstvenicima da brzo započnu raditi na svojim projektima bez potrebe za traženjem, instalacijom i upravljanjem pojedinačnim komponentama. Uključuje Python programski jezik, Jupyter Notebook i Spyder okolinu te niz popularnih biblioteka i alata za analizu podataka, znanstveno istraživanje i razvoj softvera.[9]



Slika 2. Python biblioteke

1.2.1. Tkinter

Tkinter je standardna biblioteka za izradu grafičkog korisničkog sučelja (GUI) u Pythonu. Ona pruža različite elemente poput gumba, polja za unos teksta, prozora, izbornika i ostalih značajki za komunikaciju između korisnika i aplikacije te povezivanje funkcionalnost aplikacije s grafičkim elementima. Važna je značajka u primjenama strojnog i računalnog vida, gdje je obavezna interakcija korisnika s aplikacijom kako bi se analizirale ili manipulirale slike i videozapisi.

1.2.2. OpenCV

OpenCV (*eng. Open Source Computer Vision Library*) je najveća i najpopularnija biblioteka za obradu slika i računalni vid. Nudi širok spektar funkcionalnosti kao što su učitavanje i spremanje slika, manipulacija pikselima, detekcija rubova i oblika, prepoznavanje objekata, praćenje pokreta, kalibracija kamera, analiza slika u stvarnom vremenu i još mnogo toga. Savršen je alat za prepoznavanje lica, praćenje objekata i pokreta, analizu medicinskih snimaka, autonomnu vožnju, analizu sigurnosnih kamera, industrijsku inspekciju i još mnogo toga. OpenCV biblioteka je izdana pod BSD licencom i stoga je besplatna za akademsku i komercijalnu upotrebu.[4]

1.2.3. NumPy

NumPy biblioteka služi za numeričku analizu, omogućuje skalabilnost i učinkovitost obrade podataka, njena sposobnost brze manipulacije matricama i vektorima ključna je u analizi slika. Također olakšava reprezentaciju boja i oblika kroz matematičke strukture, čime se stvara temelj za daljnje prepoznavanje.

Integracija navedenih biblioteka u sustav prepoznavanja boja i oblika omogućuje razvoj sofisticiranih aplikacija za analizu vizualnih podataka na precizan, efikasan i interaktivan način. Takav sustav se može primijeniti u industrijskoj inspekciji, medicinskoj dijagnostici, računalnim vizualizacijama, automatizaciji, te je temelj aplikacije razvijene u svrhu ovog rada.

1.2.4. Ostale biblioteke

Ove biblioteke nisu korištene u istoj mjeri kao i prethodno opisane, no krucijalne su za funkcionalnost i robustnost sustava. Biblioteka Pygrabber je bila ključna u dobavi naziva i provjeri dostupnosti vizualnog senzora (kamere). Python Imaging Library, često skraćeno Pillow, pruža skup alata za rad s slikama, a konkretno je korišten za pretvaranje slika iz OpenCV formata u format koji je pogodan za prikaz u korisničkom sučelju Tkinter aplikacije. Modul Time se koristi za mjerenje vremena i kontrolu nad ažuriranjem informacija kako bi spriječilo prečesto ažuriranje informacija i time nepotrebno opterećenje računala ili grafičkog sučelja.

2. DETEKCIJA BOJA

2.1. Model boja

Model boja je apstraktni koncept koji omogućuje reprezentaciju boja koristeći matematičke parametre, vizualizirajući spektra boja kao višedimenzionalni model. Modeli definiraju osnovne gradivne blokove koji se miješaju kako bi se stvorile različite boje, time omogućuju precizno definiranje i manipulaciju boje kako bi se postigao željeni vizualni učinak.

Većina modela boja ima 3 dimenzije pa se mogu prikazati kao 3D oblici, no mogu imat i više dimenzija kao što ima CMYK. RGB, HSV i HSL modeli boja svi koriste iste osnovne boje RGB-a, što znači da modeli boja mogu vizualizirati isti spektar boja u široko različitim dimenzijama. Dijelimo ih na modele koji su sklopovski orijentirani (RGB, CMY i CMYK) i korisnički orijentirani (HSV, HLS i HVC)

Model boje bez pripadajuće preslikavajuće funkcije prema apsolutnom prostoru boje je samo proizvoljan sustav boja bez veze s bilo kojim globalno razumljivim sustavom tumačenja boja. Tek kada se doda specifična preslikavajuća funkcije između modela boje i referentnog prostora boje, u referentnom prostoru boje uspostavlja se određeni raspon boja, i za određeni model boje to definira prostor boje. [1][2]

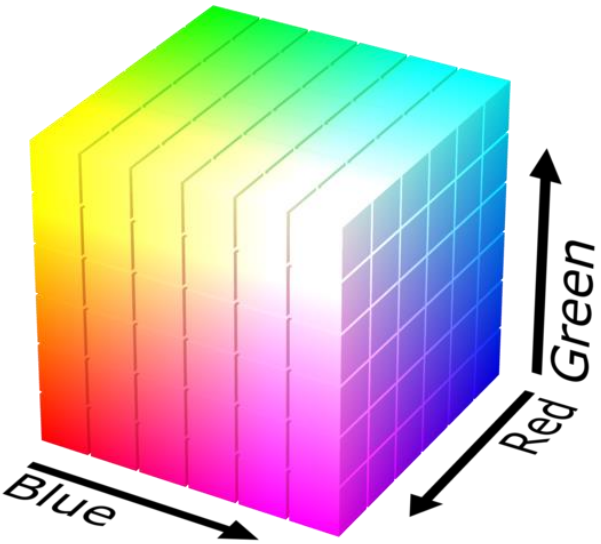
2.1.1. RGB

RGB aditivni svjetlosni model se sastoji od tri primarne boje; crvene (*eng.* ***R****ed*), zelene (*eng.* ***G****reen*) i plave (*eng.* ***B****lue*), a miješanjem njih dobivamo sve ostale, sekundarne i tercijarne. Korištenje drugih primarnih boja načelno je moguće, ali kombinacijom crvene, zelene i plave pokriven je najširi raspon ljudskog vidnog spektra.

U kartezijevom sustavu svaka koordinatna os predstavlja nijanse primarne boje, maksimalna vrijednost može biti označena postocima, decimalnim brojevima i cijelim brojevima. U modelu prikazanom pomoću kocke, gdje svaki parametar definira intenzitet boje kao cijeli broj između 0 i 255, izvorna točka (0, 0, 0) predstavlja nedostatak svjetla i crnu boju, a dijagonalno nasuprotni vrh je mješavina svih boja, odnosno bijela. Prostorna dijagonala koja ih povezuje predstavlja razine sive (R = G = B). Za piksel koji sadrži vrijednost 3 primarne boje od 0 do 255 (8 bitova) dobivamo spektar od:

$$2^8 * 2^8 * 2^8 = 2^{24} = 16777216 \text{ nijansi} \quad (1)$$

RGB model najčešće se koristi u elektroničkim sustavima kao što su digitalne i video kamere, pametni telefoni, skeneri, monitori i slično.



Slika 3. RGB model boja

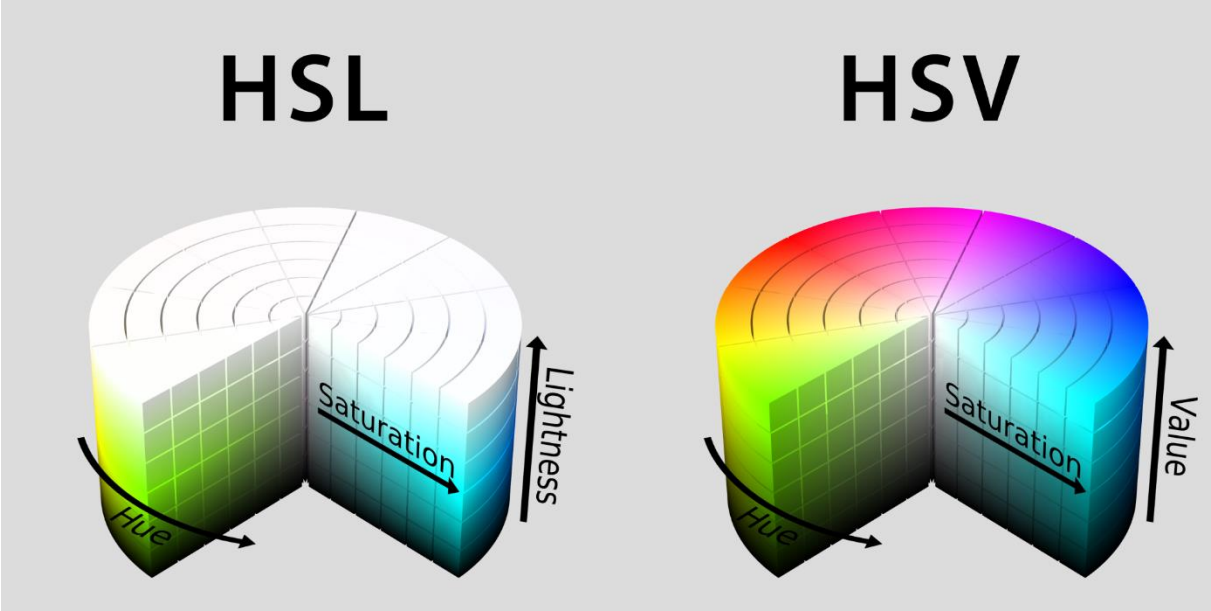
2.1.2. HSV i HSL

HSV model je razvijen kako bi se sustav više približio načinu na koji čovjek vidi boje. Intuitivniji je od RGB jer se komplementarne boje nalaze jedna nasuprot dugoj u cilindričnom koordinatnom sustavu i svaka je boja određena sa 3 komponente. Nijansa (*eng. hue*) je određena valnom duljinom spektra te ima vrijednosti 6 boja (crvene, žute, zelene, cijan, plave i magente) u krugu 0 do 360°. Boje omeđuje izvrnuta šesterostrana piramida koja se izobličuje u valjak što rezultira prostorom boja u kojem su najsvjetliji tonovi na vrhu a najtamniji na dnu. Zasićenost (*eng. saturation*) određuje čistoću boje prema udaljenosti od vertikale, od 0 (nema boje) do 100% (zasićena boja) i odnosi se na postotak bijele svjetlosti u boji. Kada je zasićenje nula, vrijednosti tona su nebitne. Vrijednost svjetline (*eng. value*) je relativna količina svjetlosti koju boja prirodno emitira od 0 (crna) do 100% (čista boja) po vertikalnoj osi. Osim što je u HSV modelu lakše i intuitivnije postaviti boje u nego u BGR modelu, on je i otporniji na promjene osvjetljenje pa ćemo upravo taj i koristiti u radu.

Tablica 1. Rasponi boja u HSV modelu

Boja	Nijansa (°)	Zasićenost (%)	Vrijednost svjetline (%)
Crvena	0-30	50-100	50-100
Narančasta	30-50	50-100	50-100
Žuta	50-80	50-100	50-100
Zelena	80-150	50-100	50-100
Cijan	150-210	50-100	50-100
Plava	210-270	50-100	50-100
Magenta	270-330	0-50	50-100
Roza	330-345	0-50	50-100
Crvena	330-360	50-100	50-100
Smeđa	0-30	50-100	0-50
Siva	0	0-50	0-50
Bijela	0	0	50-100
Crna	0	0-100	0

HSL model je vrlo sličan HSV-u, no on je definiran sa 2 spojena stošca gdje vrhovi definiraju crnu i bijelu boju. Umjesto vrijednosti svjetline, koristi se svjetlina (*eng. lightness*) koja određuje koliko je boja svijetla ili tamna, također u postocima. Bijela boja u HSL modelu ima vrijednost svjetlosti 100%, što znači da je bijela potpuno svijetla i ne sadrži nikakvu nijansu boje ili zasićenje. Bez obzira na kut nijanse koji se postavi, bijela boja će uvijek ostati bijela. Crna boja stoga ima vrijednost svjetlosti 0%, potpuno je tamna i bez zasićenja. Problem kod tog modela u računalnoj grafici se javlja zato što je raspon vrijednosti zasićenosti boje sve manji prema vrhovima stošca pa nemaju sve palete jednaki promjer.



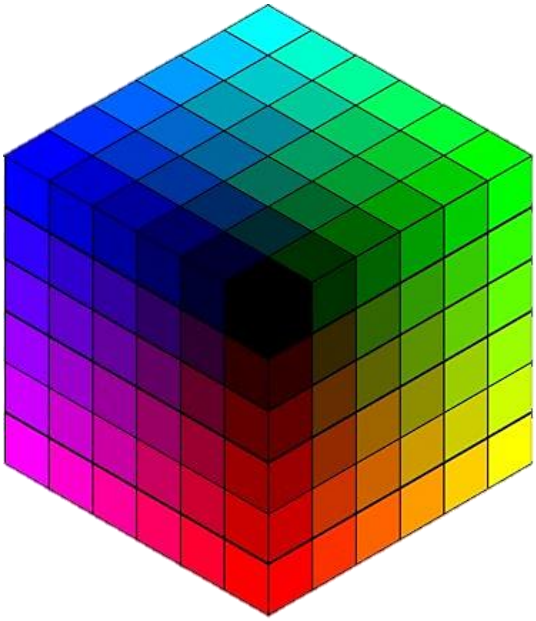
Slika 4. HSL i HSV model boja

2.1.3. CMYK

CMYK model boja igra ključnu ulogu u tiskarskoj industriji i tiskarskim procesima. Akronim označava četiri osnovne boje koje se koriste za stvaranje različitih nijansi i tonova pri reprodukciji slika i teksta. Ovaj modela koristi subtraktivnu sintezu, što znači da se boje oduzimaju od bijele svjetlosti umjesto da se dodaju kako bi se njihovom kombinacijom stvorila određena boja. Stoga se za svaku primarnu boju CMY modela može se reći da je proizašla iz oduzimanja određene boje iz bijele svjetlosti.

Cijan se dobiva oduzimanjem crvene svjetlosti iz bijele, magenta oduzimanjem zelene, te žuta oduzimanjem plave. Stoga se može reći da su redom cijan, magenta i žuta komplementarne boje crvenoj, zelenoj te plavoj. Cijan, magenta i žuta, se miješaju kako bi se stvorile različite nijanse. Rezultat spajanja sva tri elementa CMY modela je crna boja, zbog apsorpcije cjelokupne svjetlosti. No na taj način se ne dobije pravilan izgled crne boje već nijansa sive ili smeđe. Stoga se modelu pridodaje četvrti element u obliku crne boje. Iz razloga što se slovo B već koristi za plavu boju, dogovoreno je da se za crnu boju koristi slovo K odakle proizlazi naziv CMYK. Iako se može činiti kontradiktorno, dodavanje crne boje ovom modelu pomaže u poboljšanju kvalitete i preciznosti otiska. Prisustvo sve 4 temeljne boje na najnižoj razini (0, 0, 0, 0) tvori najbližu reprezentaciju bijele moguće za ovaj model.

Kroz ovu kombinaciju boja, moguće je postići široku paletu boja potrebnih za tiskanje različitih elemenata, no prostor je manji je od RGB prostora što znači da tiskarski stroj ne može reproducirati sve boje koje može prikazati monitor.



Slika 5. CMYK model boja

3. IMPLEMENTACIJA DETEKCIJE BOJA

3.1. Priprema slike i raspona boja

Zadani način rada OpenCV biblioteke koristi BGR (RGB) prostor boja, koji se često ne preporučuje za implementaciju sustava prepoznavanja boja. Kako bismo poboljšali točnost prepoznavanja boja, prvi korak je pretvoriti sliku (frame) koja se dobiva od kamere u HSV prostor boja koristeći ugrađenu funkciju OpenCV-a *cv2.COLOR_BGR2HSV*.

Sljedeći korak je definiranje raspona za svaku od 3 boje koje želimo prepoznati. Rasponi se definiraju kao NumPy nizovi od 3 komponente: nijansa, zasićenost i vrijednost u 8-bitnom zapisu. Ovaj zapis je optimalan jer štedi memoriju, kompatibilan je s većinom formata slika (JPEG, PNG), nudi dovoljnu preciznost za ljudsko oko te omogućava brzu obradu i prijenos, što je ključno za detekciju u stvarnom vremenu.

```
self.preset_ranges = {
    "CRVEN": ([0, 50, 100], [25, 255, 255]),
    "ZELEN": ([35, 50, 100], [85, 255, 255]),
    "PLAV": ([100, 50, 100], [130, 255, 255])
}
```

Slika 6. Definiranje HSV raspona boja

3.2. Kalibracija boje

Kalibracija kao proces definiranja stvarnih raspona boja koje će sustav pratiti i prepoznavati igra ključnu ulogu u ovom sustavu. Osvjetljenje može značajno utjecati na percepciju boja, a kalibracija omogućuje sustavu da prilagodi raspon boja kako bi se kompenzirale promjene osvjetljenja i osigurala dosljedna detekcija. Na ovaj način se prilagođavamo različitim uvjetima i okruženjima, proširujući spektar primjene aplikacije.

Način na koji kalibracija funkcionira u ovom kodu se temelji na mogućnosti korisnika da klikne na sliku prijenosa kamere. No prvo korisnik bira boju koju želi kalibrirati, klikom na odgovarajući gumb *Kalibriraj {boja}*, primjerice crvenu, zelenu, plavu. Nakon što je boja odabrana, korisnik klikom miša na video prikaz odabire područje koje sadrži željenu boju, a sustav zatim bilježi boju s tog područja pomoću naredbe *self.video_label.bind('<Button-1>', self.capture_color)*. Time se veže događaj *<Button-1>* za funkciju *self.capture_color*, lijevim klikom pozvat će se funkcija *self.capture_color(event)*. Dohvaćaju se koordinate klika te se mapiraju na dimenzije video prikaza kako bi se dobile ispravne koordinate bez obzira na veličinu prozora na kojem se prikazuje.

Raspon boje se onda određuje tako da se uzme prosječna vrijednost iz područja definiranog pomoću radijusa oko piksela na koji smo kliknuli, i dopusti se neka varijacija kako bi se uhvatili različiti tonovi iste boje. Stoga postavljamo radijus područja od 10 piksela oko klika i funkcijom *np.mean(cropped_frame, axis=(0, 1))* računamo prosječnu boju.

```
def capture_color(self, event):
    if self.calibrating_color:
        x, y = event.x, event.y
        x_mapped = x * self.video_width / self.video_label.winfo_width()
        y_mapped = y * self.video_height / self.video_label.winfo_height()
        radius = 10
        x1, y1 = max(0, int(x_mapped - radius)), max(0, int(y_mapped - radius))
        x2, y2 = min(self.video_width, int(x_mapped + radius)),
        min(self.video_height, int(y_mapped + radius))
        cropped_frame = self.calibration_frame[y1:y2, x1:x2]
        average_color = np.mean(cropped_frame, axis=(0, 1))
```

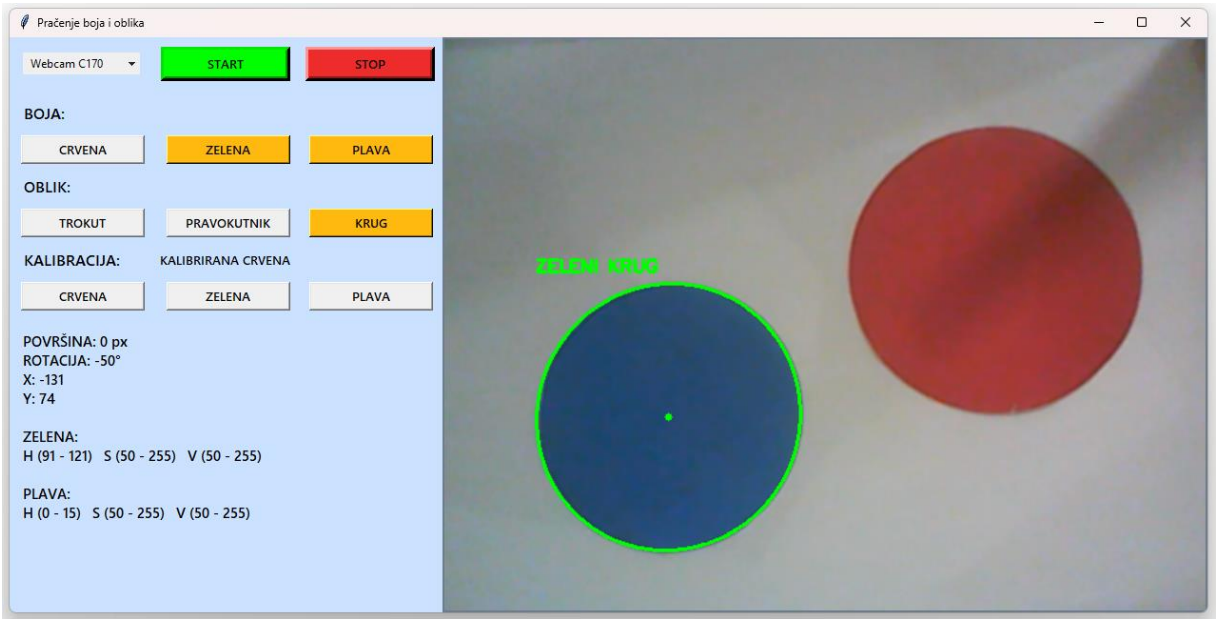
Slika 7. Kalibracija boje

Također je potrebno proširiti raspon nijanse oko dobivene boje, u ovom slučaju za ± 15, te se pobrinuti da time raspon nijanse ne pada ispod 0 ili prelazi iznad 180, jer tu nije definiran.

```
def get_color_range_from_hsv(self, hsv_color):
    hue = hsv_color[0]
    hue_tolerance = 15
    lower_bound = np.array([hue - hue_tolerance, 100, 100])
    upper_bound = np.array([hue + hue_tolerance, 255, 255])
    if lower_bound[0] < 0:
        lower_bound[0] = 0
    if upper_bound[0] > 180:
        upper_bound[0] = 180
    return lower_bound, upper_bound
```

Slika 8. Proširivanje raspona kalibrirane boje

Nakon kalibracije, više se ne koristi unaprijed zadani raspon već kalibrirani. Ovakav pristup kalibraciji zapravo omogućuje korisniku da kalibrira crvenu boju kao zelenu, plavu kao žutu itd.



Slika 9. Primjer „pogrešnog“ kalibriranja zelene boje

4. DETEKCIJA OBLIKA

Osnova detekcije oblika objekta je jasno izdvajanje elemenata koje želimo identificirati od okolnih elemenata, tj. segmentacija. Fokusirat ćemo se na tehnike analize oblika kako bi dobili željen rezultat - konture. Upareno sa detekcijom i kalibracijom boja razvijamo robustan sustav s bogatim razumijevanje okoline i objekata. Dodatni stupanj raspoznavanja oblikom također nadoknađuje manu kod prepoznavanja boja, jer iako se objekti pojavljuju u različitim bojama ili u okruženjima s promjenjivim svjetlosnim uvjetima, struktura i geometrija objekata manje variraju. To znači da se oblici mogu prepoznavati čak i ako su boje bliske pozadini ili ako su objekti djelomično prekriveni.

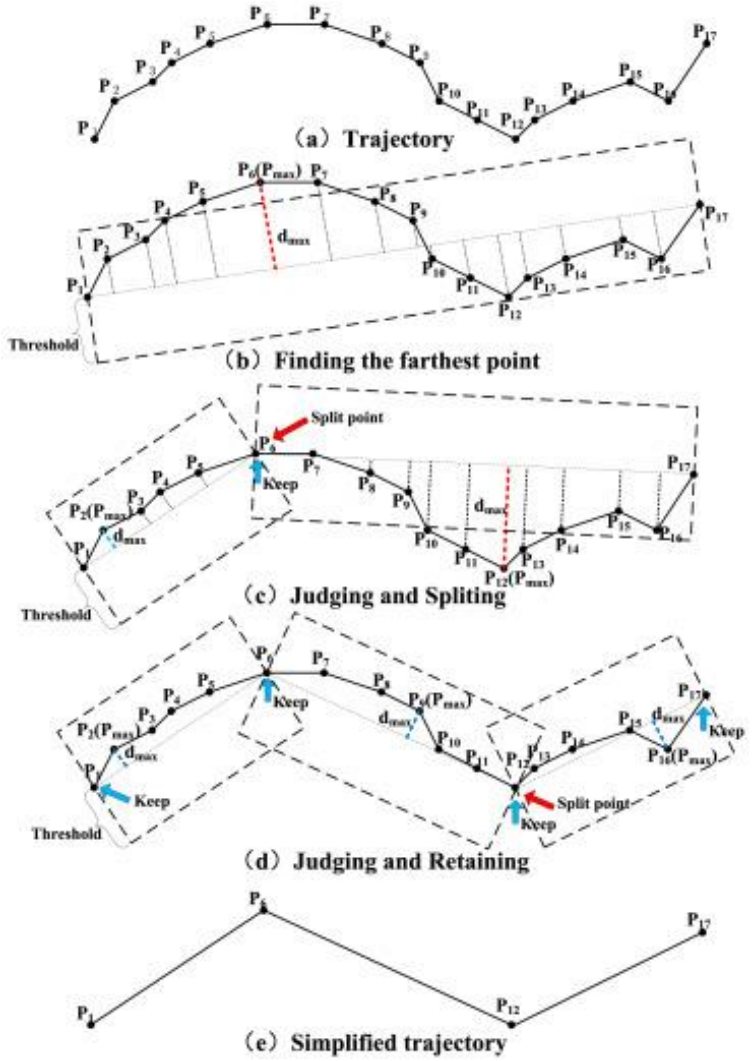
4.1. Aproksimacija kontura

Aproksimacija (pojednostavljenje) kontura je tehnika koja se koristi za aproksimiranje krivulja ili kontura jednostavnijim poligonalnim oblicima. Ova metoda pomaže smanjiti broj vrhova u krivulji, čime se olakšava njezina analiza i obrada. Kontura predstavlja vanjski rub ili granicu objekta na slici,mogu biti složene i sadržavati mnogo točaka, stoga aproksimacijom smanjujemo tu kompleksnost i zadržavamo samo ključne točke ili segmente koji dobro reprezentiraju oblik objekta. Ovime daljnju analizu i obradu činimo bržom i učinkovitijom.

Douglas-Peucker algoritam je popularan postupak za aproksimaciju krivulja ili kontura. Algoritam radi tako da zadržava početnu točku i krajnju točku konture te zatim pronalazi točku na konturi koja je najudaljenija od pravca koji povezuje početnu i krajnju točku. Ako je ta udaljenost manja od unaprijed zadane praga (tolerancije), to znači da se nema potrebe za dodatnim točkama između te dvije točke. U suprotnom, algoritam odabire točku koja je najudaljenija i koristi je kao dodatnu točku za aproksimaciju.[6]

Ključna ulazna vrijednost algoritma je tolerancija (epsilon). Ona određuje koliko blizu nove točke treba biti pravcu između početne i krajnje točke da bi se smatrala suvišnom. Veća vrijednost epsilona rezultira većim pojednostavljenjem, dok manja vrijednost očuva više detalja.

Algoritam može naići na poteškoće kod pojednostavljenja krivulja s oštrim zavojima ili višestrukim prekidima, a odabir odgovarajuće vrijednosti epsilona često zahtijeva pažljivo prilagođavanje specifičnoj primjeni.[4]



Slika 10. Proces aproksimacije krivulje

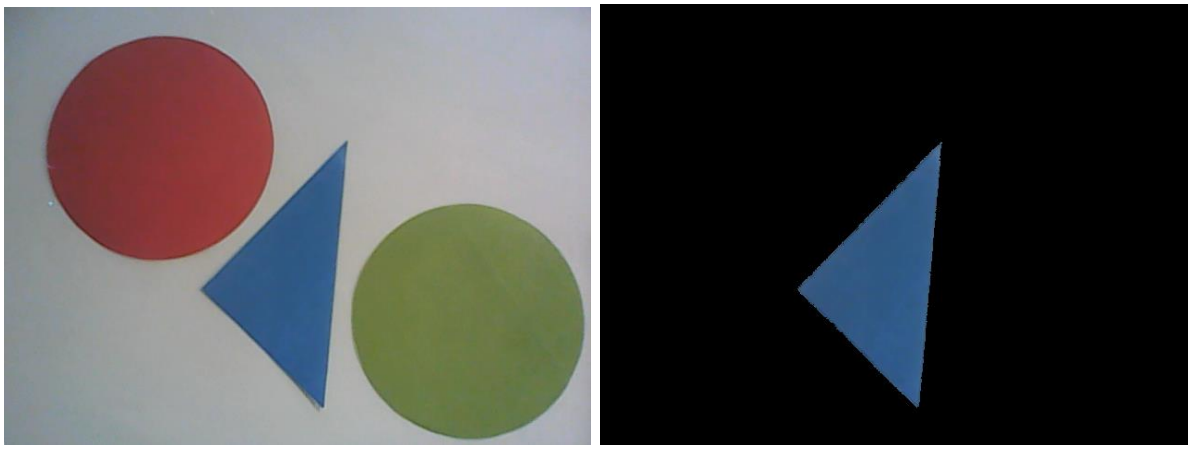
4.2. Maskiranje

Maskiranje se temelji na izdvajanju dijela prikaza koji odgovara zadanim parametrima. U slučaju ovog rada uvjeti izdvajanja dani su preko određene nijanse tražene boje. Drugim riječima, na slici će biti označena samo područja koja se podudaraju s traženom bojom, čime se fokusira pažnja na objekte koji nas zanimaju i ignoriramo dijelove slike koji nam nisu relevantni. Maskiranje je posebno korisno u situacijama gdje je pozadina jednobojna i razlikuje se od objekata koji su nam zanimljivi.

Kako bismo primijenili ovu tehniku, prvo je potrebno pretvoriti sliku iz BGR modela boja u HSV model boje. Zatim koristeći donji i gornji raspon boja precizno odabiremo piksele na slici koje želimo zadržati.

Nakon definiranja raspona boja, primjenjujemo masku na ulaznu sliku, a rezultat ove operacije je slika koja sadrži samo one piksele koji se nalaze unutar definiranog raspona boja, dok svi ostali pikseli postaju crni ili im se dodjeljuje vrijednost nula.

Važno je napomenuti da ovaj pristup može biti osjetljiv na svjetlosne uvjete, ali će pružiti zadovoljavajuće rezultate sve dok je izvor svjetlosti relativno konstantan.[4]

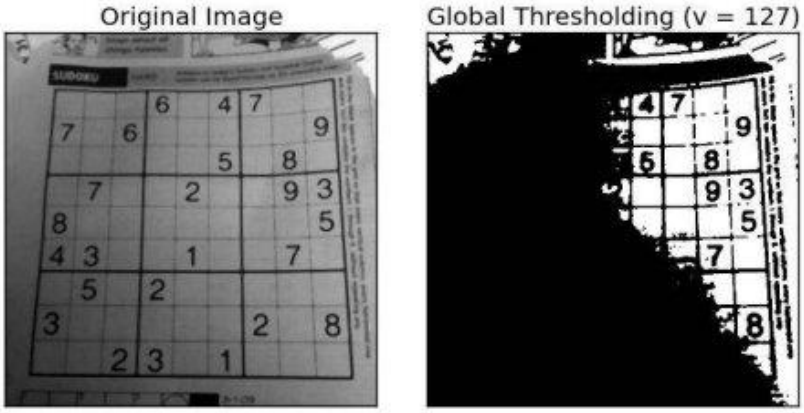


Slika 11. Rezultat maskiranja

4.3. Thresholding

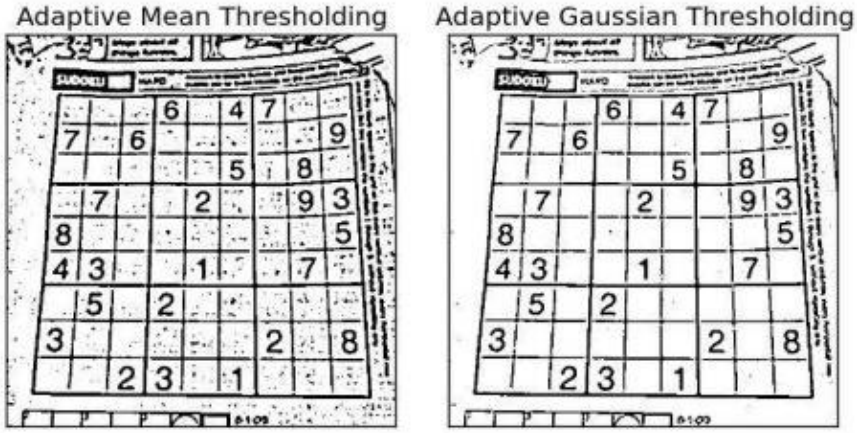
Tresholding je najjednostavniji segmentacijski proces koji omogućava pretvaranje slike u binarnu formu, gdje svaki piksel na slici pripada jednoj od dvije kategorije - bijeloj ili crnoj, ovisno o tome prelazi li piksel zadani prag (*eng. threshold*) ili ne. Mnogi objekti ili regije slike imaju karakterističnu konstantnu refleksiju ili apsorpciju svjetla na njihovim površinama što nam omogućuje da odredimo neki konstantni prag koji će razdvajati objekte od pozadine.

Kod običnog thresholdinga se koristi fiksna vrijednost praga kako bi se slika binarizirala. Primjenom na masku, sliku pojednostavljujemo za daljnju analizu jer se jasno razdvajaju objekti od pozadine i poboljšava se kontrast. Samo određivanje praga je računski nezahtjevna i jednostavna metoda pa se vrlo jednostavno izvodi u realnom vremenu.[4]



Slika 12. Rezultat thresholdanja

Adaptivno thresholdanje je naprednija tehnika koja se prilagođava lokalnim karakteristikama slike. Umjesto fiksnog praga, koristi različite pragove za različite dijelove slike. Svaki prag se računa na temelju prosjeka ili medijana vrijednosti piksela u okolnom području svakog piksela. Ova tehnika je izuzetno korisna kada je osvjetljenje na slici neujednačeno ili se mijenja, no zato zahtijeva složeniju obradu slike i može biti sporije u usporedbi s običnim thresholdanjem. [4]



Slika 13. Rezultat adaptivnog thresholdanja

4.4. Erozija i Dilatacija

Erozija i dilatacija su osnovne morfološke operacije u obradi slike koje se koriste za promjenu oblika i veličine objekata na slici. Ove operacije omogućavaju poboljšanje kvalitete binarnih maski, izoštravanje kontura objekata te uklanjanje sitnog šuma ili oštećenja na slici.

Erozija se primjenjuje na binarnu masku kako bi se smanjile konture objekata. Funkcionira tako da se za svaki piksel u slici pregledava njegova okolina definirana strukturnim elementom (*eng. kernel*).

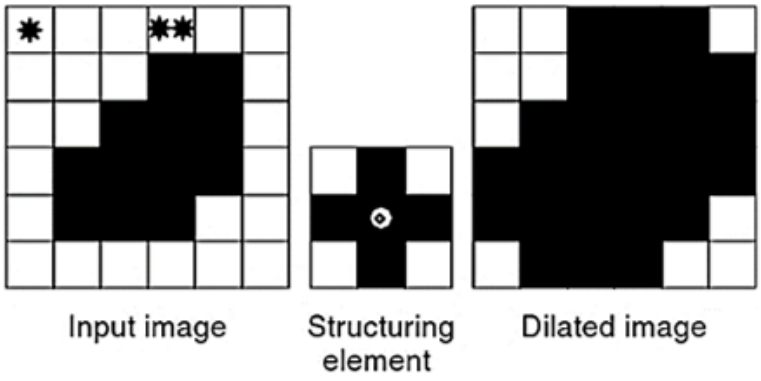
Ako svi pikseli u okolini (pod utjecajem kernela) imaju vrijednost jedan (bijela), tada će i centralni piksel ostati bijel. Inače, centralni piksel će postati crn.

Ovo smanjenje objekata pomaže u uklanjanju malih šumova i oštećenja, ali također može smanjiti veličinu i debljinu objekata. U kontekstu detekcije oblika, erodiranje može pomoći u ravnanju rubova objekata kako bi se dobile preciznije konture.

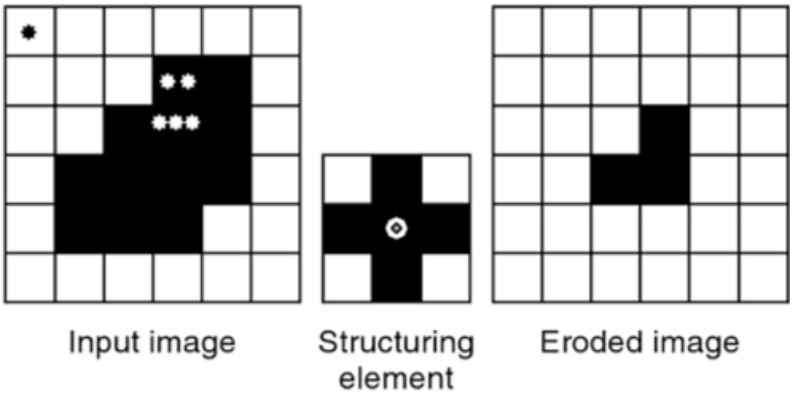
Dilatacija je obratna operacija erodiranju i služi za povećanje objekata na slici. Također se primjenjuje pomoću strukturnog elementa (kernela) koji se koristi za definiranje okoline svakog piksela na slici. Ako barem jedan piksel unutar te okoline ima vrijednost jedan (bijela), centralni piksel će postati bijel, inače ostaje crn.

Dilatacija pomaže u povećanju debljine i površine objekata, ističući ih i povećavajući njihovu vidljivost, vrlo korisno kada su objekti slabo definirani ili su konture neujednačene.

Njihova kombinacija otvaranjem (erozija s dilatacijom) može ukloniti šum i tanje dijelove objekata, ostavljajući samo glavne dijelove objekata, dok zatvaranjem (dilatacija s erozijom) može povećati objekte i zatvoriti rupe unutar njih.



Slika 14. Rezultat dilatacije



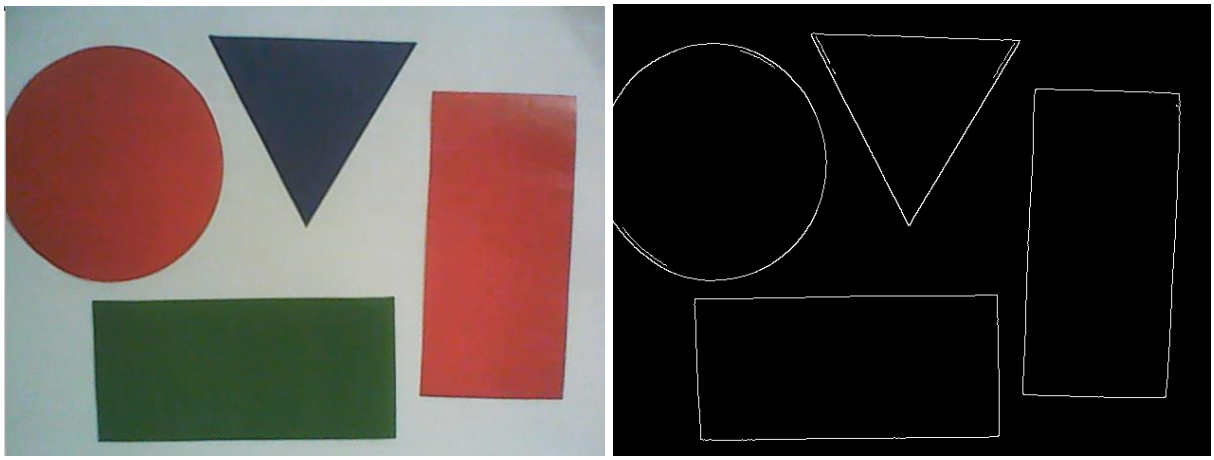
Slika 15. Rezultat erozije

4.5. Canny

Nakon prethodnih operacija maskiranja boje, thresholdanja, te erozije i dilatacije, primjenjuje se Cannyjev algoritam za precizno identificiranje rubova objekata na slici, tj. promjene u intenzitetu piksela. Razvio ga je John F. Canny 1986. godine i ostao je jedan od najčešće korištenih algoritama za ovu svrhu zbog svoje učinkovitosti i sposobnosti otkrivanja rubova uz minimiziranje lažnih detekcija. [4]

Algoritam se sastoji od 4 koraka:

1. Uklanjanje šuma i omekšavanje prijelaza između piksela Gaussovim filterom
2. Računanje gradijenta intenziteta piksela - brze promjene intenziteta i potencijalne lokacije rubova
3. Potiskivanje piksela
4. Podvrgavanje rezultata gradijenta intenziteta tresholdanju - postavljanje praga



Slika 16. Rezultat primjene Canny algoritma

5. IMPEMENTACIJA DETEKCIJE OBLIKA

Prvi korak je stvaranje maske boje pomoću funkcije *cv2.inRange* i vrijednosti *lower_color* i *upper_color*. Sljedeće koristimo adaptivno thresholdanje *funkcijom cv2.adaptiveThreshold*, kako bi se stvorila binarna slika.

Funkcija *cv2.ADAPTIVE_THRESH_MEAN_C* znači da će prag biti postavljen na temelju srednje vrijednosti okoline piksela. Postavljamo maksimalnu vrijednost koju će pikseli imati na izlazu na 255, dok će crna boja ostati 0. Argument *11* je veličina okolnog područja koje će se uzeti u obzir pri izračunu srednje vrijednosti za svaki piksel (kvadrat veličine 11x11 piksela). *2* predstavlja konstantu koja se oduzima od izračunate srednje vrijednosti kako bi se dobio prag.

Definiramo jezgru (kernel) kao kvadratnu matricu veličine 5x5 s vrijednostima 1 (bijela boja) i služit će za eroziju i dilataciju.

Slijedi erozija (*cv2.erode*) i dilatacija (*cv2.dilate*), obje koriste jezgru 5x5 te argument *iterations=2* koji označava koliko puta će se primijeniti metoda.

U funkciji *cv2.Canny*, argumenti 100 i 200 predstavljaju donji i gornji prag za detekciju rubova. Svi pikseli s vrijednostima između tih pragova bit će označeni kao rubovi.

Na kraju se koristi *cv2.findContours*. *cv2.RETR_EXTERNAL* označava da će se pronaći samo vanjske konture, dok *cv2.CHAIN_APPROX_SIMPLE* predstavlja jednostavnu kompresiju kontura radi uštede memorije. Rezultat je lista kontura.

```
def improve_shape_detection(self, frame, lower_color, upper_color):
    color_mask = cv2.inRange(frame, lower_color, upper_color)
    # Koristimo adaptivno thresholdiranje umjesto fiksne vrijednosti praga
    adaptive_thresh = cv2.adaptiveThreshold(color_mask, 255,
        cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 11, 2)
    kernel = np.ones((5, 5), np.uint8)
    adaptive_thresh = cv2.erode(adaptive_thresh, kernel, iterations=2)
    adaptive_thresh = cv2.dilate(adaptive_thresh, kernel, iterations=2)
    edges = cv2.Canny(adaptive_thresh, 100, 200)
    contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
    return contours
```

Slika 17. Implementacija algoritama i metoda za detekciju kontura

5.1. Detekcija krugova

Algoritam izračunava epsilon (ε) kao 4% duljine konture, pomnožene s *cv2.arcLength* funkcijom. Epsilon je vrijednost koja se koristi za aproksimaciju konture s manje točaka. Veća vrijednost ε rezultira aproksimacijom s manje točaka, čime se pojednostavljuje oblik.

Nakon izračunavanja epsilon-a, funkcija *cv2.approxPolyDP* koristi ga za aproksimaciju konture, pokušava zamijeniti originalnu konturu s manjim brojem točaka.

Algoritam provjerava koliko točaka ima nakon aproksimacije konture. Ako je broj točaka 5 ili algoritam izračunava površinu konture koristeći *cv2.contourArea* funkciju. Također, izračunava površinu najmanjeg opisnog kruga oko konture koristeći *cv2.minEnclosingCircle* funkciju.

Provjerava se omjer površine konture i površine najmanjeg opisnog kruga. Razlik između tih dviju površina je postavljena da mora biti manja od 0.4 kako bi se kontura prepoznala kao krug.

Također se provjerava cirkularnost konture kako bi se osiguralo da je oblik blizak krugu.

$$circularity = \frac{4 * \pi * Area}{cv2.arcLength\,(contour, True)^2} \quad (2)$$

Kontura se prepoznaje kao krug ako je cirkularnost veća od 0.8.

Ako svi ovi uvjeti za broj točaka, omjer površina i cirkularnost budu zadovoljeni, kontura se prepoznaje kao krug. Ova kombinacija uvjeta osigurava da se prepoznaju samo konture koje su bliske krugovima (uključujući elipse) i ispunjavaju određene kriterije za površinu i cirkularnost.

```
def detect_shape(self, contour):
    epsilon = 0.04 * cv2.arcLength(contour, True)
    approx = cv2.approxPolyDP(contour, epsilon, True)
    if len(approx) >= 5:
        area = cv2.contourArea(contour)
        circularity = (4 * 3.14159265359 * area) / (cv2.arcLength(contour, True) ** 2)
        min_circle_circularity = 0.8
        (x, y), radius = cv2.minEnclosingCircle(contour)
        circle_area = np.pi * (radius ** 2)
        area_ratio = area / circle_area
        if abs(1 - area_ratio) < 0.4:
            if circularity >= min_circle_circularity:
                return "KRUG"
```

Slika 18. Detekcija krugova

5.2. Trokut

Prvo se provjerava broj točaka nakon što je kontura aproksimirana, ako taj broj iznosi 3, tada je moguće da je kontura trokut. Nakon što se utvrdi da kontura ima 3 točke, izračunavaju se kutovi između svake kombinacije susjednih točaka.

Kako bismo provjerili je li kontura trokut, uspoređujemo izračunate kutove s kutovima jednakokračnog trokuta. Kako bi se uzela u obzir moguća odstupanja u izračunima i malene nepreciznosti, dopuštena je mala margina greške za kutove u iznosu ±10 stupnjeva. To znači da

će se kontura prepoznati kao trokut ako su svi izračunati kutovi unutar raspona od 50 do 70 stupnjeva (60 ± 10 stupnjeva).

Ako niti jedan kut nije unutar dopuštenog raspona od ±10 stupnjeva jednakokračnog trokuta, kontura se ne smatra trokutom i prelazi se na sljedeći oblik.

```
if len(approx) == 3:
    angles = []
    for i in range(3):
        pt1 = approx[i][0]
        pt2 = approx[(i + 1) % 3][0]
        pt3 = approx[(i + 2) % 3][0]
        vector1 = pt1 - pt2
        vector2 = pt3 - pt2
        cos_angle = np.dot(vector1, vector2) / (np.linalg.norm(vector1) * np.linalg.norm(vector2))
        angle = np.arccos(np.clip(cos_angle, -1, 1))
        angle_degrees = np.degrees(angle)
        angles.append(angle_degrees)
    angle_threshold = 10
    if (all(abs(angle - angles[0]) < angle_threshold for angle in angles)
        or any(abs(90 - angle) < angle_threshold for angle in angles)):
        return "TROKUT"
return "Nepoznato"
```

Slika 19. Detekcija trokuta

5.3. Pravokutnik

Provjerava se duljina konture kako bi se utvrdilo ima li točno 4 točke. Ako kontura nema 4 točke, oblik se ne smatra pravokutnikom.

Za svaki od četiri vrha aproksimiranog poligona izračunavaju se kutovi između dva vektora koji se protežu od trenutnog vrha do dvaju susjednih vrhova, koristeći trigonometrijsku funkciju arkussinus.

Argument *angle_threshold* postavljen je na 10 stupnjeva, samo kutovi koji su unutar 10 stupnjeva od 90 stupnjeva (pravog kuta) su prihvatljivi kao kutovi pravokutnika.

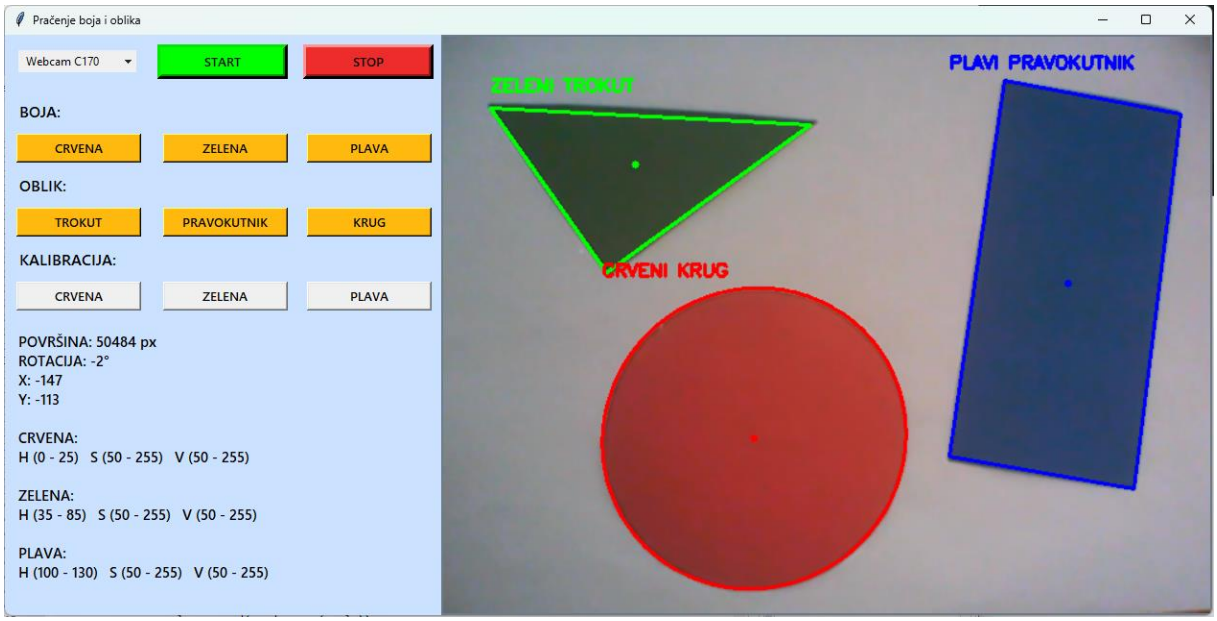
Da bi se dodatno potvrdilo da je oblik pravokutnik, provjerava se njegov omjer širine i visine. Računa se kao w / h, gdje su w *i* h širina i visina uređujućeg pravokutnika oko konture. Oblik će se prepoznati kao pravokutnik samo ako je omjer izvan raspona od 0.8 do 1.2.

Osim prepoznavanja pravokutnika, kod također provjerava je li oblik kvadrat (omjer vrlo blizu 1). Ovaj korak je nužan zbog česte greške prepoznavanja kvadra unutar kruga.

```
if len(approx) == 4:
    angles = []
    for i in range(4):
        pt1 = approx[i][0]
        pt2 = approx[(i + 1) % 4][0]
        pt3 = approx[(i + 2) % 4][0]
        vector1 = pt1 - pt2
        vector2 = pt3 - pt2
        angle = np.arccos(np.dot(vector1, vector2) / (np.linalg.norm(vector1) * np.linalg.norm(vector2)))
        angles.append(np.degrees(angle))
    angle_threshold = 10
    if (
        abs(angles[0] - angles[2]) < angle_threshold
        and abs(angles[1] - angles[3]) < angle_threshold
    ):
        w, h = cv2.boundingRect(contour)[2:]
        aspect_ratio = float(w) / h
        if not (0.8 <= aspect_ratio <= 1.2):
            return "PRAVOKUTNIK"
        else:
            return "Nepoznato"
```

Slika 20. Detekcija pravokutnika

Nakon uspješne implementacije sustava za prepoznavanje oblika i boja, napokon dobivamo zadovoljavajući rezultat detekcije.



Slika 21. Detekcija boja i oblika

6. KORISNIČKO SUČELJE

Glavni alat za uspostavljanje kontakta između korisnika i koda je Tkinter. On omogućuje stvaranje prozora, gumba, polja za unos i drugih korisničkih elemenata koji su potrebni za interakciju s korisnikom. Za sve elemente je moguće odredit definiramo stupac (*column*), raspon stupaca (*columnspan*), redak (*row*), raspon redaka (*rowspan*), poziciju (*sticky*), širinu (*width*), razmak (*padx*, *pady*), itd. Također preko njihovih stanja (0/1, utpušteno/stisnuto) možemo upravljati ostalim funkcijama aplikacije.[10]

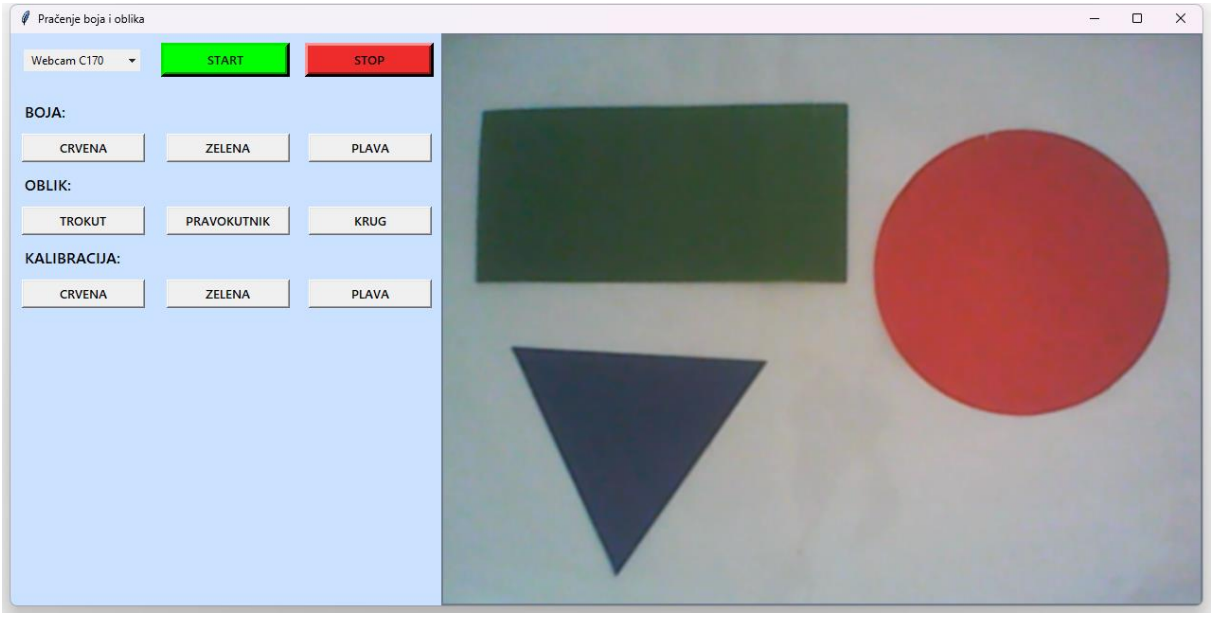
Najosnovnija funkcija je postavljanje naziva prozora u "*Pračenje boja i oblika*", te zadavanje boje pozadine u *lightsteelblue1*, gdje *root* argument predstavlja glavni prozor. Kako bi zadržali pravilan oblik sučelja, određujemo minimalnu širinu stupaca na 150 piksela.

Za odabir kamere je napravljen spuštajući izbornik, koji izlistava sve dostupne kamere. Kako bi se dobili sustavski nazivi kamera, koristi se biblioteka *pygrabber.dshow_graph* i njena funkcija *FilterGraph*.

Značajka koja je najviše korištena u ovom kodu je stvaranje i korištenje gumbova, za pokretanje i zaustavljanje aplikacije, odabir oblika i boja za praćenje te kalibraciju istih. Na primjer, gumb *START* u prozoru root, širine 15, pozadinske boje svjetlo zelena , font slova *Segoe UI Semibold*, 10 boldano, boja teksta postavljena na crnu i obrub gumba debljine 5. Također smo smjestili taj gumb u 0-ti red prvog stupca, te dodali razmake od ostalih objekata u smjeru x za 0 piksela i u smjeru y za 10 piksela. Slične postavke su postavljene i na ostale gumbe koji služe za kalibraciju i detekciju boja. Tek odabirom kamere i klikom na start se prikazuju gumbi za detekciju, kalibraciju, prijenos kamere te informacije o objektu.

Isto tako detekcija boja i oblika započinje tek kada odaberemo bar jednu boju i oblik, kalibracija odabirom boje, a cijeli program možemo zatvoriti klikom na *STOP*

Slika 22. Početni zaslon



Slika 23. Korisničko sučelje

6.1. Označivanje detektiranih objekata

U svrhu jasnog prikaza rezultata detekcije na zaslonu prijenosa kamere, detektirani oblici su označeni. Funkcija *cv2.putText* se koristi za ispisivanje boje i oblika objekta iznad njega samog.

Metoda draw_rectangle koristi funkciju *cv2.minAreaRect(contour)* za pronalaženje najmanjeg okvira (eng. bounding box) koji obuhvaća konturu. Koordinate četiri ugla pravokutnika se pretvaraju u cijele brojeve pomoću *np.intp* i onda se pomoću *cv2.approxPolyDP* se aproksimira kontura kako bi se dobile grube koordinate pravokutnika. Na kraju se iscrtava se pravokutnik oko konture pomoću *cv2.drawContours*.

Metodom *draw_circle* se crta krug, odnosno elipsa za slučaj malog zakreta kruga, oko objekta. Funkcija *cv2.fitEllipse(contour)* služi za pronalaženje elipse koja najbolje odgovara konturi, te se izračunavaju koordinate težišta, poluosi i orijentacija kako bi se mogla nacrtati elipsa.

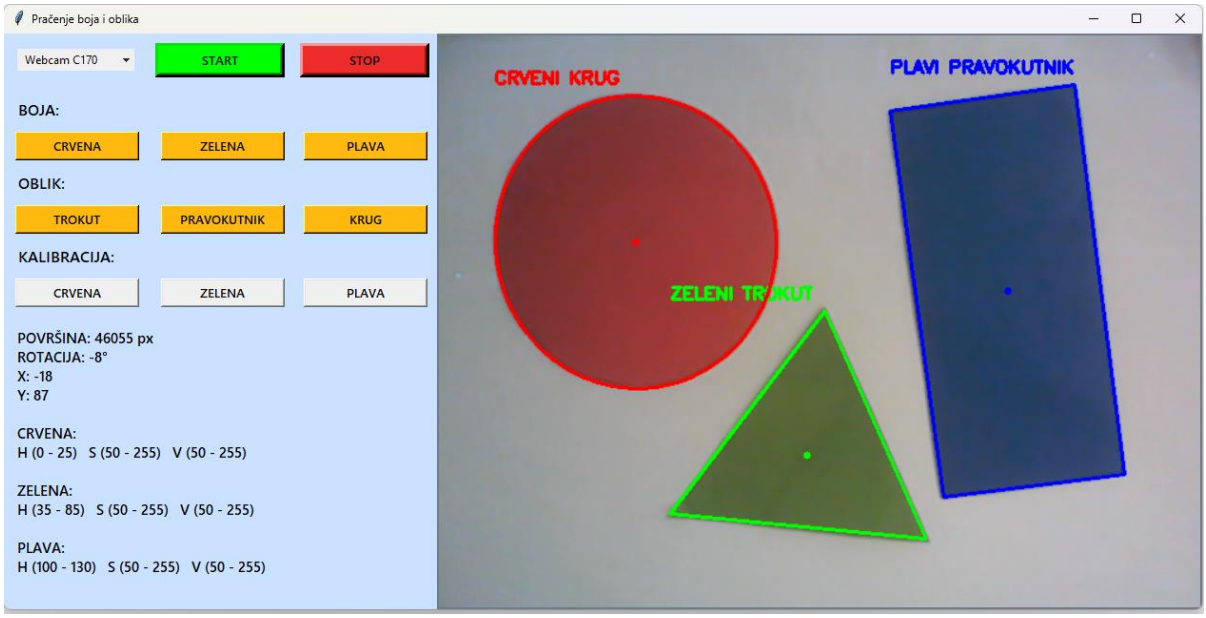
Draw_triangle koristi *cv2.approxPolyDP* za aproksimaciju konture kako bi se dobile grube koordinate trokuta, te se iscrtavaju, a koordinate težišta se računaju preko momenta pomoću *cv2.moments*. U sredini svih objekata se pomoću *cv2.circle* crta točka u težištu.

```
def draw_rectangle(self, frame, contour, color):
    rect = cv2.minAreaRect(contour)
    box = cv2.boxPoints(rect)
    box = np.intp(box)
    center_x = int(rect[0][0])
    center_y = int(rect[0][1])
    cv2.circle(frame, (center_x, center_y), 3, color, -1)
    epsilon = 0.04 * cv2.arcLength(contour, True)
    approx = cv2.approxPolyDP(contour, epsilon, True)
    cv2.drawContours(frame, [approx], 0, color, 2)

def draw_circle(self, frame, contour, color):
    (x, y), (major_axis, minor_axis), rotation = cv2.fitEllipse(contour)
    center = (int(x), int(y))
    axes = (int(major_axis / 2), int(minor_axis / 2))
    angle = int(rotation)
    cv2.ellipse(frame, center, axes, angle, 0, 360, color, 2)
    cv2.circle(frame, center, 3, color, -1)

def draw_triangle(self, frame, contour, color):
    epsilon = 0.04 * cv2.arcLength(contour, True)
    approx = cv2.approxPolyDP(contour, epsilon, True)
    cv2.drawContours(frame, [approx], 0, color, 2)
    M = cv2.moments(contour)
    center_x = int(M["m10"] / M["m00"])
    center_y = int(M["m01"] / M["m00"])
    cv2.circle(frame, (center_x, center_y), 3, color, -1)
```

Slika 24. Crtanje kontura objekta

7. REZULTATI DETEKCIJE

Detektirani objekt je omeđen linijama koje se poklapaju sa konturm objekta te je iznad toga ispisana i boja te oblik objekta. Kako bi se ovakva aplikacija stvarno mogla primijeniti u praksi, osim detekcije boje i oblika trebamo raspolagati i sa informacijama o položaju, orijentaciji te veličini predmeta. Ove informacije omogućuju dublje razumijevanje okoline i objekata te omogućuju preciznije i inteligentnije reakcije na događaje. Praćenje oblika je ključno u industrijama kao što su kvaliteta proizvoda, gdje se provjerava da li je proizvod pravilnog oblika. Položaj objekta u odnosu na naš sustav nam daje informaciju o našem okruženju što je osnova robotima, autonomnim vozilima i drugim uređajima da se precizno pozicioniraju i navigiraju kroz prostor. Površina objekta pruža informacije o veličini objekta i može biti korisna u kvantifikaciji kvalitete ili količine. Na primjer, u poljoprivredi se može pratiti površina povrća ili plodova na temelju koje se donose odluke o njihovom uzgoju ili berbi. To je također važno pri odabiru predmeta zanimanja jer ga možemo izolirati od ostalih smetnji. Rotacija predmeta je korisna za određivanje rotacije samog sustava u odnosu na okolinu ili na primjer određivanje potrebne rotacije prihvatnice u odnosu na objekt koji želimo podići. Također su ispisani i rasponi boja koje se detektiraju kako bi se vidjela promjena nakon kalibracije.



Slika 25. Informacije o objektu

7.1. Rotacija

Određivanje rotacije predmeta je kompleksan problem jer zahtijeva informaciju o prvotnoj zakrenutosti te praćenju smjer u kojem se rotira u odnosu na taj položaj. Ovo je posebno problematično kod simetričnih oblika jer je otežano odredit referentnu ravninu u odnosu na koju mjerimo ili na primjer kod krugova, gdje ta ravnina niti ne postoji.

Način koji je rotacija određena u aplikaciji je tako da se računa kut između konture objekta i pozitivne osi x. Prvo se koristi funkcija *cv2.moments(contour)* za izračunavanje momenata konture. Momenti su matematički opisi oblika koji se koriste za izračunavanje različitih svojstava oblika, uključujući njegovu orijentaciju. Ta funkcija prima konturu kao ulaz i vraća vrijednosti momenata, kao što su *"m00", "mu20", "mu11", i "mu02",* za izračunavanje orijentacije konture.

if moments["m00"] == 0 provjerava je li moment *"m00"* jednak nuli. Moment *"m00"* predstavlja ukupnu površinu konture, pa ako je nula to znači da kontura nema površine, što bi onemogućilo izračunavanje kuta rotacije. U takvim slučajevima funkcija vraća 0 stupnjeva (nema rotacije)

covariance_matrix konstruira 2x2 matricu kovarijacije koristeći centralne momente "*mu20", "mu11", "mu11", i "mu02"*. Ova matrica se koristi za analizu raspodjele točaka unutar konture, prikazuje statističku vezu između dvije varijable te služi za dobivanje eigenvalues i eigenvectors.

Funkcijom *np.linalg.eig* se izračunavaju svojstvene vrijednosti i svojstveni vektori matrice kovarijacije. Svojstvene vrijednosti predstavljaju veličinu glavnih osi, a svojstveni vektori predstavljaju smjerove tih glavnih osi. U ovom kontekstu, svojstveni vektori će nam reći orijentaciju konture.

Sljedeće sa *np.argmax(eigenvalues)* pronalazimo indeks svojstvene vrijednosti s najvećim iznosom što odgovara glavnoj osi konture, a *major_axis* iz eigenvectors niza izdvaja svojstveni vektor koji odgovara glavnoj osi. Taj svojstveni vektor predstavlja smjer glavne osi konture.

Funkcija *np.arctan2* izračunava arkustangens omjera y - komponente i x - komponente svojstvenog vektora glavne osi što daje kut rotacije u radijanima. Konačno taj kut u radijanima pretvaramo u stupnjeve pomoću *np.degrees(rotation_radians).*

```
def calculate_rotation(self, contour):
    moments = cv2.moments(contour)
    if moments["m00"] == 0:
        return 0
    covariance_matrix = np.array([[moments["mu20"], moments["mu11"]],
                                  [moments["mu11"], moments["mu02"]]])
    eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
    major_axis_idx = np.argmax(eigenvalues)
    major_axis = eigenvectors[:, major_axis_idx]
    rotation_radians = np.arctan2(major_axis[1], major_axis[0])
    rotation_degrees = np.degrees(rotation_radians)
    return rotation_degrees
```

Slika 26. Računanje rotacije

7.2. Položaj

Funkcija *cv2.boundingRect(contour)* koristi konturu kako bi pronašla minimalni pravokutnik koji u potpunosti obuhvaća tu konturu. Vraća četiri vrijednosti:

- *x i y* su koordinate gornjeg lijevog ugla pravokutnika.
- *w* je širina pravokutnika.
- *h* je visina pravokutnika.

Varijable *center_x* i *center_y* predstavljaju položaj objekta u odnosu na sredinu slike, a *self.video_width* i *self.video_height* dimenzije prozora koji prikazuje prijenos kamere.

Prvo se računa x koordinata centra pravokutnika, položaj centra pravokutnika u odnosu na sredinu širine video okvira:

$$center\_x = \mathrm{x} + \frac{w}{2} - \frac{\mathrm{self.video\_width}}{2} \tag{3}$$

Položaj centra pravokutnika u odnosu na sredinu visine se računa na sličan način:

$$center\_y = \mathrm{y} + \frac{h}{2} - \frac{\mathrm{self.video\_height}}{2} \tag{4}$$

Ako su *center_x* i *center_y* pozitivni, objekt se nalazi desno i ispod središta slike, dok negativne vrijednosti znače da je objekt lijevo i iznad središta slike.

7.3. Površina

Površina objekta se jednostavno dobiva se analizom kontura i OpenCV funkcijom *cv2.contourArea(contour)*. Ova funkcija prima konturu kao ulaz i vraća površinu te konture.

Koristimo prag površine *self.min_contour_area* da bi smo odlučili da li je kontura dovoljno velika za daljnju analizu.

```
contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL, cv2.CHAIN_APPROX_SIMPLE)
for contour in contours:
    area = cv2.contourArea(contour)
    if area > self.min_contour_area:
        shape = self.detect_shape(contour)
```

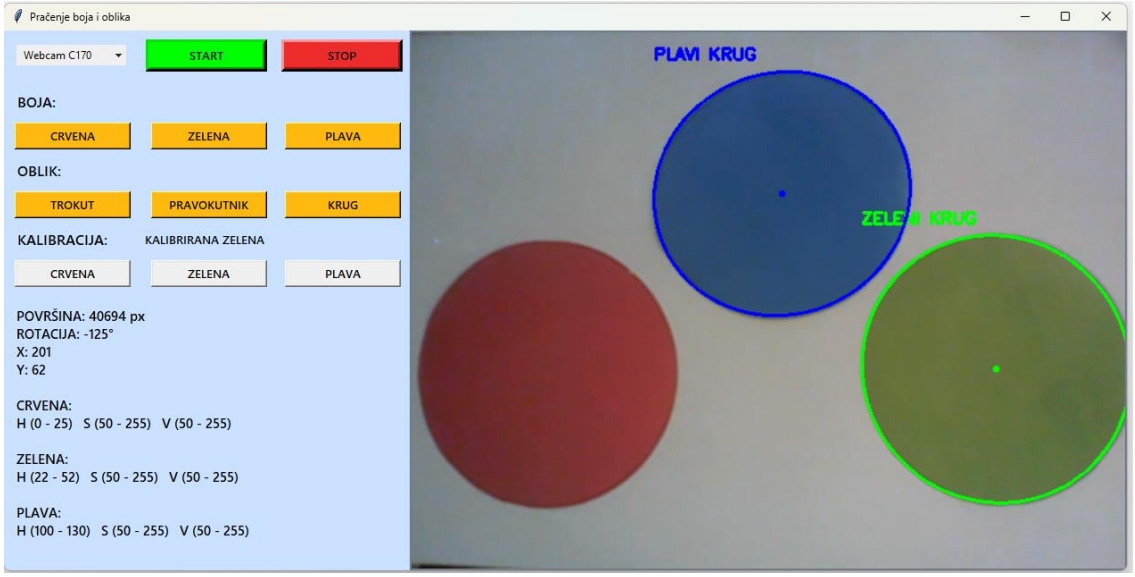
Slika 27. Računanje površine

8. ANALIZA REZULTATA

Korištenje Pythona i pripadajućih biblioteka u razvoju aplikacije za ovaj rad se pokazalo kao pogodno rješenje. Jedna od najvećih prednosti ovakvog sustava je intuitivnost i jednostavnost korištenja programskog jezika. Štoviše, Python je iznimno popularan i ima široku zajednicu korisnika diljem svijeta koji često dijele svoje znanje i iskustvo putem različitih foruma, što znači da za gotovo svaki problem ili grešku koja se može susrest u procesu programiranja, već postoji rješenje ili barem smjernica kako pristupiti problemu. A zahvaljujući rasprostranjenosti u svijetu, praktički za svaki dobiveni error je moguće pronaći rješenje jer se već netko sigurno susreo s istim problemom.

Naravno postoji i broj poteškoća koje nije uvijek jednostavno riješiti. Jedan od najvećih izazova je pravilno podešavanje parametara i pragova (thresholdova) kako bi se postigli željeni rezultati u detekciji objekata. To može biti prilično zahtjevan proces jer treba postići ravnotežu između detekcije željenih i isključivanja nepotrebnih značajki. Brzina rad i uključivanja aplikacije bi također mogla biti poboljšana koristeći naprednijih biblioteka poput Threading, boljim strukturiranjem i reduciranjem ovog povećeg koda.

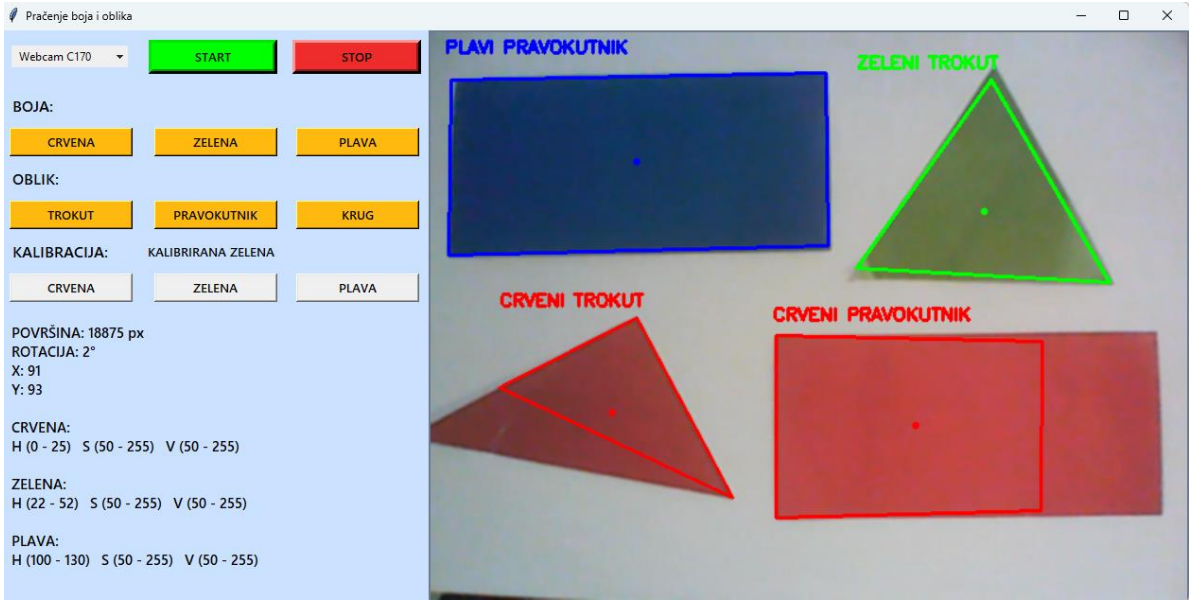
Na primjer, detekcija crvene boje može biti kompliciranа zbog položaja ove boje u modelu boja HSV. Crvena boja proteže se preko granica vrijednosti zasićenja, smještene između 330 i 360 stupnjeva te 0 i 30 stupnjeva. Stoga je potrebno uzeti oba ova raspona u obzir prilikom detekcije crvene boje. Osim toga, detekcija boja i oblika često ovisi o svjetlosnim uvjetima, a algoritmi za obradu slike mogu biti vrlo osjetljivi na te promjene.



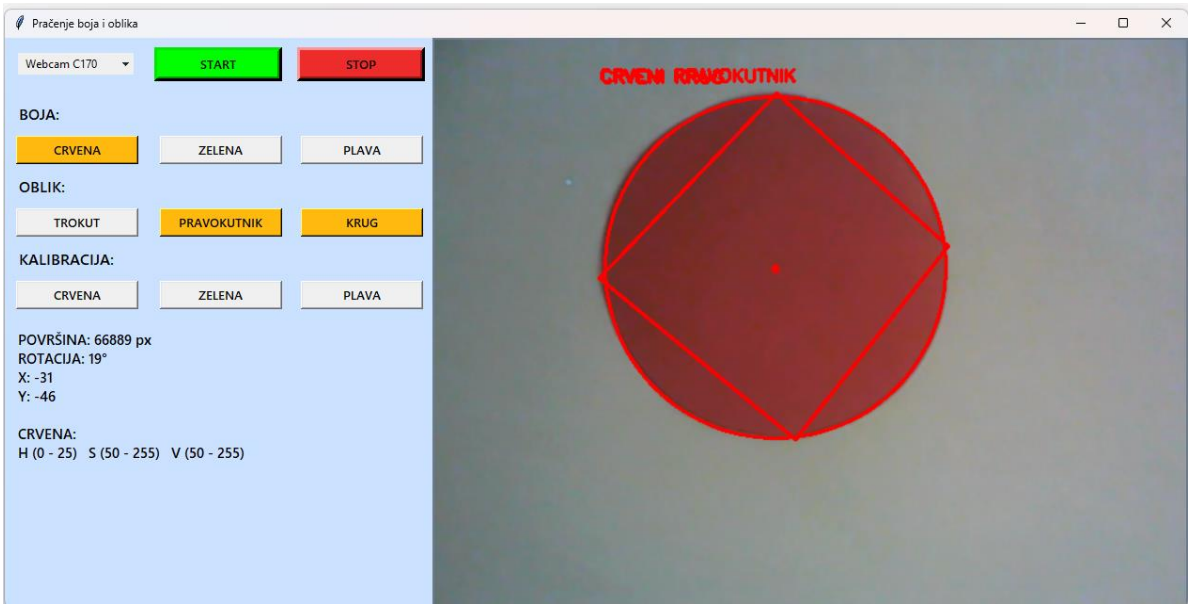
Slika 28. Problem pri detekciji crvene boje

Kalibracija može pomoći u rješavanju ovih problema. no da je ovaj sustav implementiran u mobilni robot koji se kreće po većem postrojenju sa različito osvjetljenim područjima u različita doba dana, kalibrirat bi se trebalo konstantno.

Još jedan problem koji se češće pojavljivao je bilo pogrešno detektiranje oblika. Sustav bi detektirao trokute unutar pravokutnika i krugova, ili kvadrate upisane unutar krugova. Stoga su poduzete mjere za strože detektiranje oblika.



Slika 29. Pogrešno detektiranje oblika



Slika 30. Pogrešno detektiranje kvadra unutar kruga

Aplikacija bi se mogla značajno poboljšati primjenom tehnologija dubokog učenja (*eng. deep learning*). To bi pridonijelo višoj razini preciznosti i učinkovitosti u detekciji i prepoznavanju objekata u stvarnom vremenu. Glavna prednosti bi bila preciznije prepoznavanja objekata na temelju složenih obrazaca i karakteristika. Umjesto tradicionalnih metoda detekcije objekata, aplikacija bi mogla koristiti duboke neuronske mreže poput YOLO (You Only Look Once) ili Faster R-CNN. Jedan od izazova koji bi duboko učenje moglo riješiti je detekcija crvene boje, koja može biti kompleksna zbog svojeg položaja u HSV modelu boja, a duboki modeli mogu naučiti prepoznavati ovu boju bez obzira na osvjetljenje i nijanse.

9. ZAKLJUČAK

Cilj rada je bio razviti aplikaciju koja dostojno može riješiti problem detekcije boja i oblika. Jasno je iskazana moć i svestranost programskog jezika Python i njegovih biblioteka. Također je dana teorijska podloga obavezna za razumijevanje rada detektiranja boja i oblika. Od računalnog i strojnog vida, modela i prostora boja, te različitih načina obrade slike i dobivanja kontura.

Izazovi podešavanja parametara i pragova za detekciju objekata su zahtijevali posebnu pažnju i strpljenje kako bi se dobio zadovoljavajući ishod. Problemi kod detekcije boja zbog osjetljivosti sustava na svjetlosne uvjet, pogotovo crvene zbog njezinog položaja u HSV modelu boja, su rezultirali obaveznim razvojem sustava kalibracije.

Nadalje, ova aplikacija uspješno izvršava svoju zadaću te ostavlja mogućnost nadogradnje i daljnjeg razvoja, što bi bilo i neophodno ako bi se upustili u domenu ozbiljnijeg rada i primjenu kod mobilnih robota i automatizacije.

LITERATURA

[1] https://en.wikipedia.org/wiki/Color_model

[2] https://www.colorsexplained.com/color-models/

[3] https://stackoverflow.com

[4] https://docs.opencv.org/4.x/index.html

[5] https://github.com

[6] https://en.wikipedia.org/wiki/Ramer–Douglas–Peucker_algorithm

[7] https://www.geeksforgeeks.org

[8] https://www.zebra.com/us/en/resource-library/faq/what-is-the-difference-between-machine-vision-computer-vision.html

[9] https://www.python.org

[10] https://www.askpython.com/python-modules/tkinter/tkinter-padding-tutorial

PRILOZI

I. CD-R disc

II. Tehnička dokumentacija

TEHNIČKA DOKUMENTACIJA

```
import tkinter as tk
from tkinter import ttk
from tkinter import StringVar
import NumPy as np
import cv2
import time
from pygrabber.dshow_graph import FilterGraph
from PIL import Image, ImageTk

class ColorShapeDetectionApp:
    def __init__(self, root):
        self.root = root
        self.root.configure(bg='lightsteelblue1')
        self.root.title("Pračenje boja i oblika")
        root.columnconfigure(0, minsize=150)
        root.columnconfigure(1, minsize=150)
        root.columnconfigure(2, minsize=150)
        self.camera_index = None
        self.cap = None
        self.active_camera_index = None
        self.preset_ranges = {
            "CRVEN": ([0, 50, 50], [25, 255, 255]),
            "ZELEN": ([35, 50, 50], [85, 255, 255]),
            "PLAV": ([100, 50, 50], [130, 255, 255])
        }
        self.calibration_button_states = {
            "CRVEN": False,
            "ZELEN": False,
            "PLAV": False
        }
        self.calibration_button_colors = {
```

```
            "CRVEN": "SystemButtonFace",
            "ZELEN": "SystemButtonFace",
            "PLAV": "SystemButtonFace"
        }
        self.selected_colors = []
        self.selected_shapes = []
        self.tracking_frame = tk.Frame(self.root, bg='lightsteelblue1')
        self.tracking_frame.grid(row=2, column=0, padx=0, pady=0, columnspan=3, rowspan=1,
sticky='ew')
        self.tracking_frame.columnconfigure(0, minsize=150)
        self.tracking_frame.columnconfigure(1, minsize=150)
        self.tracking_frame.columnconfigure(2, minsize=150)
        self.width_ratio = 1.0
        self.height_ratio = 1.0
        self.video_label = tk.Label(self.root, bg='lightsteelblue1')
        self.video_label.grid(row=0, column=3, padx=0, pady=0, rowspan=200, columnspan=3,
sticky="nsew")
        self.video_label.max_size = (800, 600)
        self.video_update_id = None
        self.calibrating_color = None
        self.calibration_color_ranges = {}
        self.calibration_widgets_visible = False
        self.active_camera_index = None
        self.camera_options = []
        self.camera_selection_var = StringVar(self.root)
        self.camera_selection_menu = ttk.OptionMenu(self.root, self.camera_selection_var,
"KAMERA", *self.get_available_cameras())
        self.camera_selection_menu.config(width=15)
        self.camera_selection_menu.grid(row=0, column=0, padx=5, pady=10)
        self.refresh_camera_options()
        self.start_button = tk.Button(self.root, text="START",
command=self.start_selected_camera, width=15, bg='lime', font=("Segoe UI Semibold", 10,
"bold"), fg='black', bd=5)
        self.start_button.grid(row=0, column=1, padx=0, pady=10)
```

```
        self.last_info_update_time = 0
        self.stop_button = tk.Button(self.root, text="STOP", command=self.stop_detection,
width=15, bg='firebrick2', font=("Segoe UI Semibold", 10, "bold"), fg='black', bd=5)
        self.stop_button.grid(row=0, column=2, padx=0, pady=10)
        self.active_camera_index = None
        self.selected_color_button = None
        self.selected_shape_button = None
        self.color_names = ['CRVEN', 'ZELEN', 'PLAV']
        self.calibration_widgets_visible = False
        self.calibrated_colors = {}
        self.calibration_success_label = None
        self.video_width = 0
        self.video_height = 0
        self.last_frame = None
        self.color_to_track = None
        self.shape_to_track = None
        self.min_contour_area = 300

    def get_camera_backend_name(self, idx):
        cap = cv2.VideoCapture(idx)
        backend_name = FilterGraph().get_input_devices()
        cap.release()
        return backend_name

    def get_available_cameras(self):
        available_cameras = []
        devices = FilterGraph().get_input_devices()
        for idx, device_name in enumerate(devices):
            available_cameras.append((device_name, idx))
        return available_cameras

    def change_camera(self):
        if self.cap:
            self.cap.release()
```

```
        selected_camera = self.camera_selection_var.get()
        try:
            new_camera_index = selected_camera[1]
        except (ValueError, IndexError):
            print(f"Invalid camera option format: {selected_camera}")
            return
        self.cap = cv2.VideoCapture(new_camera_index)
        self.update_video_label()

    def refresh_camera_options(self):
        menu = self.camera_selection_menu["menu"]
        menu.delete(0, "end")
        cameras = self.get_available_cameras()
        for camera_option in cameras:
            menu.add_command(label=camera_option[0], command=lambda
value=camera_option[0]: self.camera_selection_var.set(value))

    def is_camera_available(self, idx):
        cap = cv2.VideoCapture(idx)
        available = cap.isOpened()
        cap.release()
        return available

    def start_selected_camera(self):
        selected_camera = self.camera_selection_var.get()
        camera_index = self.find_camera_index_by_name(selected_camera)
        if camera_index is not None and self.is_camera_available(camera_index):
            self.start_detection(camera_index)
            self.video_label.config(bg='lightsteelblue4')
            self.info_label = tk.Label(self.tracking_frame, text="", font=("Segoe UI Semibold", 11,
"bold"), width=35, anchor='w', justify='left', bg='lightsteelblue1')
            self.info_label.grid(row=7, column=0, padx=0, pady=0, columnspan=3, sticky='w')

    def find_camera_index_by_name(self, camera_name):
```

```
        for index, name in enumerate(self.get_available_cameras()):
            if camera_name in name:
                return index
        return None

    def start_detection(self, camera_index):
        if self.cap:
            self.cap.release()
        if self.video_update_id:
            self.root.after_cancel(self.video_update_id)
        self.camera_index = camera_index
        self.cap = cv2.VideoCapture(self.camera_index)
        if not self.cap.isOpened():
            print(f"Neuspjelo otvaranje kamere {self.camera_index}")
            return
        self.video_width = int(self.cap.get(cv2.CAP_PROP_FRAME_WIDTH))
        self.video_height = int(self.cap.get(cv2.CAP_PROP_FRAME_HEIGHT))
        self.create_tracking_buttons()
        self.create_color_calibration_buttons()
        self.update_video_label()
        self.refresh_camera_options()

    def stop_detection(self):
        if self.cap:
            self.cap.release()
        if self.video_update_id:
            self.root.after_cancel(self.video_update_id)
        if hasattr(self, 'calibration_window'):
            self.calibration_window.destroy()
        if hasattr(self, 'video_label'):
            self.video_label.config(image=None)
        self.root.destroy()

    def update_video_label(self):
```

```
        ret, frame = self.cap.read()
        if ret:
            self.last_frame = frame
            frame_with_shapes = self.detect_colors_and_shapes(frame)
            resized_frame, self.width_ratio, self.height_ratio = self.resize_frame(frame_with_shapes)
            self.display_frame(resized_frame)
            self.video_update_id = self.root.after(10, self.update_video_label)
        else:
            self.stop_detection()

    def resize_frame(self, frame):
        max_width, max_height = self.video_label.max_size
        height, width, _ = frame.shape
        width_ratio = max_width / width
        height_ratio = max_height / height
        ratio = min(width_ratio, height_ratio)
        new_width = int(width * ratio)
        new_height = int(height * ratio)
        resized_frame = cv2.resize(frame, (new_width, new_height))
        return resized_frame, width_ratio, height_ratio

    def display_frame(self, frame):
        if frame is not None and frame.size > 0:
            frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
            img = Image.fromarray(frame)
            img = ImageTk.PhotoImage(img)
            if hasattr(self, 'video_label'):
                self.video_label.img = img
                self.video_label.config(image=img)
            else:
                self.video_label = tk.Label(self.root, image=img)
                self.video_label.grid(row=0, column=0, columnspan=3, padx=10, pady=10)
        else:
```

```
        if hasattr(self, 'video_label'):
            self.video_label.config(image=None)

    def improve_shape_detection(self, frame, lower_color, upper_color):
        color_mask = cv2.inRange(frame, lower_color, upper_color)
        adaptive_thresh = cv2.adaptiveThreshold(color_mask, 255,
            cv2.ADAPTIVE_THRESH_MEAN_C, cv2.THRESH_BINARY, 11, 2)
        kernel = np.ones((5, 5), np.uint8)
        adaptive_thresh = cv2.erode(adaptive_thresh, kernel, iterations=2)
        adaptive_thresh = cv2.dilate(adaptive_thresh, kernel, iterations=2)
        edges = cv2.Canny(adaptive_thresh, 100, 200)
        contours, _ = cv2.findContours(edges, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
        return contours

    def detect_colors_and_shapes(self, frame, detect_green=True, detect_red=True):
        frame = cv2.GaussianBlur(frame, (5, 5), 0)
        hsv_frame = cv2.cvtColor(frame, cv2.COLOR_BGR2HSV)
        tracked_objects = []
        for color in self.selected_colors:
            if color in self.calibrated_colors:
                lower_color, upper_color = self.calibration_color_ranges[color]
                contours = self.improve_shape_detection(hsv_frame, lower_color, upper_color)
            else:
                preset_range = self.preset_ranges.get(color)
                if preset_range:
                    lower_color, upper_color = np.array(preset_range[0]), np.array(preset_range[1])
                else:
                    continue
            mask = cv2.inRange(hsv_frame, lower_color, upper_color)

            contours, _ = cv2.findContours(mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
            for contour in contours:
```

```
            area = cv2.contourArea(contour)
            if area > self.min_contour_area:
                shape = self.detect_shape(contour)

                if shape in self.selected_shapes:
                    tracked_objects.append((color, shape, contour))
        if detect_green and 'ZELEN' in self.selected_colors:
            green_ranges = [
                (np.array([35, 100, 100]), np.array([85, 255, 255]))
            ]
            if not self.calibrated_colors.get('ZELEN'):
                green_mask = np.zeros(hsv_frame.shape[:2], dtype=np.uint8)
                for lower_green, upper_green in green_ranges:
                    green_mask1 = cv2.inRange(hsv_frame, lower_green, upper_green)
                    green_mask = cv2.bitwise_or(green_mask, green_mask1)
                green_mask = cv2.GaussianBlur(green_mask, (5, 5), 0)
                _, green_mask = cv2.threshold(green_mask, 50, 255, cv2.THRESH_BINARY)
                green_contours, _ = cv2.findContours(green_mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
                for contour in green_contours:
                    area = cv2.contourArea(contour)
                    if area > self.min_contour_area:
                        shape = self.detect_shape(contour)
                        if shape in self.selected_shapes:
                            tracked_objects.append(("ZELEN", shape, contour))
        if detect_red and 'CRVEN' in self.selected_colors:
            red_ranges = [
                (np.array([0, 100, 100]), np.array([10, 255, 255])),
                (np.array([160, 100, 100]), np.array([179, 255, 255]))
            ]
            if not self.calibrated_colors.get('CRVEN'):
                red_mask = np.zeros(hsv_frame.shape[:2], dtype=np.uint8)
                for lower_red, upper_red in red_ranges:
                    red_mask1 = cv2.inRange(hsv_frame, lower_red, upper_red)
```

```
            red_mask = cv2.bitwise_or(red_mask, red_mask1)
        red_mask = cv2.GaussianBlur(red_mask, (5, 5), 0)
        _, red_mask = cv2.threshold(red_mask, 50, 255, cv2.THRESH_BINARY)
        red_contours, _ = cv2.findContours(red_mask, cv2.RETR_EXTERNAL,
cv2.CHAIN_APPROX_SIMPLE)
        for contour in red_contours:
            area = cv2.contourArea(contour)
            if area > self.min_contour_area:
                shape = self.detect_shape(contour)
                if shape in self.selected_shapes:
                    tracked_objects.append(("CRVEN", shape, contour))
    drawn_shapes = {}
    for tracked_object in tracked_objects:
        color, shape, contour = tracked_object
        if color in drawn_shapes:
            if shape in drawn_shapes[color]:
                continue
        x, y, w, h = cv2.boundingRect(contour)
        color_rgb = (0, 255, 0)
        if color == "CRVEN":
            color_rgb = (0, 0, 255)
        elif color == "PLAV":
            color_rgb = (255, 0, 0)
        if shape == "TROKUT":
            self.draw_triangle(frame, contour, color_rgb)
        elif shape == "KRUG":
            self.draw_circle(frame, contour, color_rgb)
        elif shape == "PRAVOKUTNIK":
            self.draw_rectangle(frame, contour, color_rgb)
        if color not in drawn_shapes:
            drawn_shapes[color] = [shape]
        else:
            drawn_shapes[color].append(shape)
        info_text = f"{color}I {shape}"
```

```
            cv2.putText(frame, info_text, (x, y - 10), cv2.FONT_HERSHEY_SIMPLEX, 0.5, color_rgb, 2)
            rotation = self.calculate_rotation(contour)
            center_x = x + w / 2 - self.video_width / 2
            center_y = y + h / 2 - self.video_height / 2
            self.update_info_label(area, center_x, center_y, rotation)
        return frame

    def draw_rectangle(self, frame, contour, color):
        rect = cv2.minAreaRect(contour)
        box = cv2.boxPoints(rect)
        box = np.intp(box)
        center_x = int(rect[0][0])
        center_y = int(rect[0][1])
        cv2.circle(frame, (center_x, center_y), 3, color, -1)
        epsilon = 0.04 * cv2.arcLength(contour, True)
        approx = cv2.approxPolyDP(contour, epsilon, True)
        cv2.drawContours(frame, [approx], 0, color, 2)

    def draw_circle(self, frame, contour, color):
        (x, y), (major_axis, minor_axis), rotation = cv2.fitEllipse(contour)
        center = (int(x), int(y))
        axes = (int(major_axis / 2), int(minor_axis / 2))
        angle = int(rotation)
        cv2.ellipse(frame, center, axes, angle, 0, 360, color, 2)
        cv2.circle(frame, center, 3, color, -1)

    def draw_triangle(self, frame, contour, color):
        epsilon = 0.04 * cv2.arcLength(contour, True)
        approx = cv2.approxPolyDP(contour, epsilon, True)
        cv2.drawContours(frame, [approx], 0, color, 2)
        M = cv2.moments(contour)
        center_x = int(M["m10"] / M["m00"])
        center_y = int(M["m01"] / M["m00"])
```

```
        cv2.circle(frame, (center_x, center_y), 3, color, -1)

    def detect_shape(self, contour):
        epsilon = 0.04 * cv2.arcLength(contour, True)
        approx = cv2.approxPolyDP(contour, epsilon, True)
        if len(approx) >= 5:
            area = cv2.contourArea(contour)
            circularity = (4 * 3.14159265359 * area) / (cv2.arcLength(contour, True) ** 2)
            min_circle_circularity = 0.8
            (x, y), radius = cv2.minEnclosingCircle(contour)
            circle_area = np.pi * (radius ** 2)
            area_ratio = area / circle_area
            if abs(1 - area_ratio) < 0.8:
                if circularity >= min_circle_circularity:
                    return "KRUG"

        if len(approx) == 4:
            angles = []
            for i in range(4):
                pt1 = approx[i][0]
                pt2 = approx[(i + 1) % 4][0]
                pt3 = approx[(i + 2) % 4][0]
                vector1 = pt1 - pt2
                vector2 = pt3 - pt2
                angle = np.arccos(np.dot(vector1, vector2) / (np.linalg.norm(vector1) *
np.linalg.norm(vector2)))
                angles.append(np.degrees(angle))
            angle_threshold = 10
            if (
                abs(angles[0] - angles[2]) < angle_threshold
                and abs(angles[1] - angles[3]) < angle_threshold
            ):
                w, h = cv2.boundingRect(contour)[2:]
                aspect_ratio = float(w) / h
```

```
            if not (0.8 <= aspect_ratio <= 1.2):
                return "PRAVOKUTNIK"
            else:
                return "Nepoznato"

        if len(approx) == 3:
            angles = []
            for i in range(3):
                pt1 = approx[i][0]
                pt2 = approx[(i + 1) % 3][0]
                pt3 = approx[(i + 2) % 3][0]
                vector1 = pt1 - pt2
                vector2 = pt3 - pt2
                cos_angle = np.dot(vector1, vector2) / (np.linalg.norm(vector1) *
np.linalg.norm(vector2))
                angle = np.arccos(np.clip(cos_angle, -1, 1))
                angle_degrees = np.degrees(angle)
                angles.append(angle_degrees)
            angle_threshold = 10
            if (all(abs(angle - angles[0]) < angle_threshold for angle in angles)
                or any(abs(90 - angle) < angle_threshold for angle in angles)):
                return "TROKUT"
        return "Nepoznato"


    def create_color_calibration_buttons(self):
        self.color_calibration_buttons = tk.Frame(self.tracking_frame)
        self.color_calibration_buttons.grid(row=5, column=0, padx=0, pady=0, columnspan=1,
rowspan=1)
        self.odabir_kamere = tk.Label(self.tracking_frame, text="  KALIBRACIJA:  ",
font=("Segoe UI Semibold", 12, "bold"), bg='lightsteelblue1')
        self.odabir_kamere.grid(row=5, column=0, columnspan=1, padx=0, pady=10, sticky='w')
        color_names = ['CRVEN', 'ZELEN', 'PLAV']
        for idx, color in enumerate(color_names):
            button_name = f'{color.lower()}'
```

```
            button = tk.Button(self.tracking_frame, text=f"{color}A", command=lambda c=color:
self.capture_calibration_color(c), width=15, font=("Segoe UI Semibold", 10, "bold"))
            setattr(self, button_name, button)
            button.grid(row=6, column=idx, padx=0, pady=0)
        self.update_calibration_button_colors()
        self.video_label.bind('<Button-1>', self.capture_color)

    def bind_capture_color(self):
        self.calibration_video_label.bind('<Button-1>', self.capture_color)

    def capture_color(self, event):
        if self.calibrating_color:
            x, y = event.x, event.y
            x_mapped = x * self.video_width / self.video_label.winfo_width()
            y_mapped = y * self.video_height / self.video_label.winfo_height()
            radius = 10
            x1, y1 = max(0, int(x_mapped - radius)), max(0, int(y_mapped - radius))
            x2, y2 = min(self.video_width, int(x_mapped + radius)), min(self.video_height,
int(y_mapped + radius))
            cropped_frame = self.calibration_frame[y1:y2, x1:x2]
            average_color = np.mean(cropped_frame, axis=(0, 1))
            if np.isnan(average_color).any():
                self.calibration_success_label = tk.Label(self.tracking_frame, text="KALIBRACIJA
NEUSPJEĹ NA", font=("Segoe UI Semibold", 10, "bold"), bg='lightsteelblue1')
                self.calibration_success_label.grid(row=5, column=1, columnspan=3, padx=0,
pady=0)
                return
            hsv_color = cv2.cvtColor(np.uint8([[average_color]]), cv2.COLOR_BGR2HSV)[0][0]
            self.calibrated_colors[self.calibrating_color] = tuple(hsv_color)
            self.calibration_color_ranges[self.calibrating_color]                    =
self.get_color_range_from_hsv(tuple(hsv_color))
            if self.calibration_success_label:
                self.calibration_success_label.destroy()
```

```
            self.calibration_success_label = tk.Label(self.tracking_frame, text=f"KALIBRIRANA {self.calibrating_color}A", font=("Segoe UI Semibold", 10, "bold"), bg='lightsteelblue1')
            self.calibration_success_label.grid(row=5, column=1, columnspan=3, padx=0, pady=0, sticky='w')
            self.calibration_button_states[self.calibrating_color] = not self.calibration_button_states[self.calibrating_color]
            self.update_calibration_button_colors()
            self.calibrating_color = None
            self.video_label.bind('<Button-1>', self.capture_color)

    def set_calibrating_color(self, color):
        self.calibrating_color = color
        self.calibration_frame = self.last_frame
        self.video_label.bind('<Button-1>', self.capture_color)

    def get_color_range_from_hsv(self, hsv_color):
        hue = hsv_color[0]
        hue_tolerance = 15
        lower_bound = np.array([hue - hue_tolerance, 50, 50])
        upper_bound = np.array([hue + hue_tolerance, 255, 255])
        if lower_bound[0] < 0:
            lower_bound[0] = 0
        if upper_bound[0] > 180:
            upper_bound[0] = 180
        return lower_bound, upper_bound

    def update_calibration_button_colors(self):
        for color, state in self.calibration_button_states.items():
            if state:
                getattr(self, f'{color.lower()}').config(bg=self.calibration_button_colors[color])
            else:
                getattr(self, f'{color.lower()}').config(bg="SystemButtonFace")

    def end_calibration(self):
```

```
        if self.calibration_success_label:
            self.calibration_success_label.destroy()
        for color in ['CRVEN', 'ZELEN', 'PLAV']:
            self.calibration_button_states[color] = False
            self.calibration_button_colors[color] = "SystemButtonFace"
        self.update_calibration_button_colors()

    def create_tracking_buttons(self):
        self.create_color_buttons()
        self.create_shape_buttons()

    def set_color_to_track(self, color):
        if color in self.selected_colors:
            self.selected_colors.remove(color)
            self.color_buttons[color].config(bg="SystemButtonFace")
        else:
            self.selected_colors.append(color)
            self.color_buttons[color].config(bg="darkgoldenrod1")

    def set_shape_to_track(self, shape):
        if shape in self.selected_shapes:
            self.selected_shapes.remove(shape)
            self.shape_buttons[shape].config(bg="SystemButtonFace")
        else:
            self.selected_shapes.append(shape)
            self.shape_buttons[shape].config(bg="darkgoldenrod1")

    def update_info_label(self, contour_area=None, center_x=None, center_y=None,
rotation=None):
        current_time = time.time()
        if current_time - self.last_info_update_time < 0.2:
            return
        info_text = "\n"
        if contour_area is not None:
```

```
            contour_area = int(contour_area)
            info_text += f"   POVRŠINA: {contour_area} px\n"
        else:
            info_text += "   POVRšINA: x\n"
        if rotation is not None:
            rotation = int(rotation)
            info_text += f"   ROTACIJA: {rotation}°\n"
        else:
            info_text += "   ROTACIJA: x\n"
        if center_x is not None and center_y is not None:
            center_x = int(center_x)
            center_y = int(center_y)
            info_text += f"   X: {center_x}\n"
            info_text += f"   Y: {center_y}"
            info_text += "\n"
        else:
            info_text += "   X: x\n"
            info_text += "   Y: x"
        for color in self.selected_colors:
            if color in self.calibrated_colors:
                min_values, max_values = self.calibration_color_ranges[color]
            else:
                min_values, max_values = self.preset_ranges[color]
            info_text += f"\n   {color}A:\n"
            info_text += f"   H ({min_values[0]} - {max_values[0]})"
            info_text += f"   S ({min_values[1]} - {max_values[1]})"
            info_text += f"   V ({min_values[2]} - {max_values[2]})\n"
        self.info_label.config(text=info_text, padx=0, pady=0, anchor='w')
        self.last_info_update_time = current_time

    def create_color_buttons(self):
        color_names = ['CRVEN', 'ZELEN', 'PLAV']
        self.odabir_kamere = tk.Label(self.tracking_frame, text="    BOJA:", font=("Segoe UI
Semibold", 12, "bold"), bg='lightsteelblue1')
```

```
        self.odabir_kamere.grid(row=0, column=0, columnspan=1, padx=0, pady=10, sticky='w')
        self.color_buttons = {}
        for idx, color in enumerate(color_names):
            button = tk.Button(self.tracking_frame, text=f"{color}A", command=lambda c=color:
self.set_color_to_track(c), width=15, font=("Segoe UI Semibold", 10, "bold"))
            button.grid(row=1, column=idx, padx=0, pady=0, columnspan=1)
            self.color_buttons[color] = button

    def create_shape_buttons(self):
        shape_names = ['TROKUT', 'PRAVOKUTNIK', 'KRUG']
        self.odabir_kamere = tk.Label(self.tracking_frame, text="   OBLIK:", font=("Segoe UI
Semibold", 12, "bold"), bg='lightsteelblue1')
        self.odabir_kamere.grid(row=2, column=0, columnspan=1, padx=0, pady=10, sticky='w')
        self.shape_buttons = {}
        for idx, shape in enumerate(shape_names):
            button = tk.Button(self.tracking_frame, text=f"{shape}", command=lambda s=shape:
self.set_shape_to_track(s), width=15, font=("Segoe UI Semibold", 10, "bold"))
            button.grid(row=3, column=idx, padx=0, pady=0, columnspan=1)
            self.shape_buttons[shape] = button

    def display_calibration_frame(self, frame):
        frame = cv2.cvtColor(frame, cv2.COLOR_BGR2RGB)
        img = Image.fromarray(frame)
        img = ImageTk.PhotoImage(img)
        if hasattr(self, 'calibration_video_label'):
            self.calibration_video_label.img = img
            self.calibration_video_label.config(image=img)
        else:
            self.calibration_video_label = tk.Label(self.calibration_window, image=img)
            self.calibration_video_label.grid(row=5, column=5, padx=0, pady=0)
        self.bind_capture_color()

    def capture_calibration_color(self, color):
        if self.calibrating_color == color:
```

```
            self.calibrating_color = None
            self.calibration_button_states[color] = False
            self.calibration_button_colors[color] = "SystemButtonFace"
            self.update_calibration_button_colors()
            if hasattr(self, 'calibration_video_label'):
                self.calibration_video_label.grid_forget()
        else:
            if self.calibrating_color:
                prev_color = self.calibrating_color
                self.calibration_button_states[prev_color] = False
                self.calibration_button_colors[prev_color] = "SystemButtonFace"
                self.update_calibration_button_colors()
            self.calibration_button_states[color] = not self.calibration_button_states[color]
            self.calibration_button_colors[color] = "darkgoldenrod1" if
self.calibration_button_states[color] else "SystemButtonFace"
            self.calibrating_color = color
            self.update_calibration_button_colors()
            if self.calibration_button_states[color]:
                ret, frame = self.cap.read()
                self.calibration_frame = frame

    def clear_calibration_success_label(self):
        if self.calibration_success_label:
            self.calibration_success_label.destroy()

    def calculate_rotation(self, contour):
        moments = cv2.moments(contour)
        if moments["m00"] == 0:
            return 0
        covariance_matrix = np.array([[moments["mu20"], moments["mu11"]],
                                      [moments["mu11"], moments["mu02"]]])
        eigenvalues, eigenvectors = np.linalg.eig(covariance_matrix)
        major_axis_idx = np.argmax(eigenvalues)
        major_axis = eigenvectors[:, major_axis_idx]
```

```
        rotation_radians = np.arctan2(major_axis[1],  major_axis[0])
        rotation_degrees = np.degrees(rotation_radians)
        return rotation_degrees

if __name__ == "__main__":
    root = tk.Tk()
    app = ColorShapeDetectionApp(root)
    root.protocol("WM_DELETE_WINDOW", app.stop_detection)
root.mainloop()
```