Razvoj sustava za detekciju i praćenje pikado igre primjenom vizijskih tehnologija

Cvetić, Jurica

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Jurica Cvetić

Zagreb, 2024.

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE

DIPLOMSKI RAD

Mentor:

Doc. dr. sc. Filip Šuligoj, mag.ing.mech.

Student:

Jurica Cvetić

Zagreb, 2024.

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

Veliko hvala mentoru doc.dr.sc. Filipu Šuligoju na dostupnosti, korisnim savjetima, motivaciji i uputama prilikom izrade rada.

Zahvaljujem svim kolegama u CRTA-i na motivaciji i pružanoj pomoći prilikom sklapanja fizičkog postava.

Od srca zahvaljujem svim prijateljima na velikoj podršci i motivaciji tijekom cijelog studija.

Enormno hvala obitelji i djevojci koji su me uvijek poticali i pružali sve kako bi mi obrazovanje bilo ljepše i bezbolnije.

Hvala Ti!

Jurica Cvetić

SVEUČILIŠTE U ZAGREBU
FAKULTET STROJARSTVA I BRODOGRADNJE
Središnje povjerenstvo za završne i diplomske ispite
Povjerenstvo za diplomske ispite studija strojarstva za smjerove:
Proizvodno inženjerstvo, inženjerstvo materijala, industrijsko inženjerstvo i menadžment, mehatronika i robotika, autonomni sustavi i računalna inteligencija

Sveučilište u Zagrebu Fakultet strojarstva i brodogradnje	
Datum	Prilog
Klasa: **602 - 04 / 24 - 06 / 1**	
Ur.broj: **15 - 24 -**	

DIPLOMSKI ZADATAK

Student: **Jurica Cvetić** JMBAG: 35219238

Naslov rada na hrvatskom jeziku: **Razvoj sustava za detekciju i praćenje pikado igre primjenom vizijskih tehnologija**

Naslov rada na engleskom jeziku: **Development of a dart game detection and tracking system using vision technologies**

Opis zadatka:

S obzirom na primjenu vizijskih sustava u različitim područjima ljudske aktivnosti, ovaj rad fokusira se na razvoj naprednih tehnologija u rekreacijskim i sportskim aktivnostima, konkretnije u igri pikada. Cilj rada je razviti sustav koji koristi strojni vid i algoritme strojnog učenja za detekciju i aktivno praćenje igre pikada, što uključuje detekciju ulazne točke pikado strelice i praćenje rezultata. Sustav će se oslanjati na tri 2D kamere postavljene tako da pokrivaju cijelu pikado metu iz različitih kutova, omogućujući precizno praćenje rezultata igre. Specifični zadaci rada obuhvaćaju:

- Projektiranje sustava i odabir komponenti. Prije implementacije, ključno je projektirati cjelokupni sustav za detekciju i praćenje pikado igre, uključujući odabir i postavljanje kamera, nosača, rasvjete i markera. Ovaj korak uključuje analizu prostornih zahtjeva i odabir tehnoloških komponenti koji će osigurati sve tražene funkcionalnosti.
- Intrinzična kalibracija tri 2D kamere. Prvi korak uključuje kalibraciju kamera kako bi se osigurala korekcija distorzije slike te visoka preciznost detekcije pozicije pikado strelice. Kalibracija će se provesti korištenjem standardnih metoda kalibracije, uključujući upotrebu kalibracijskih uzoraka.
- Detekcija markera. Za precizno praćenje igre, potrebno je ispraviti perspektivu slike mete. Ovo će se postići detekcijom markera postavljenih oko mete, omogućujući homografsko ispravljanje slike.
- Detekcija polja mete. Algoritmi strojnog vida koristit će se za detekciju različitih polja na pikado meti, omogućujući sustavu da precizno odredi gdje je strelica pogodila.
- Detekcija ulazne točke pikado strelice. Sustav će koristiti napredne algoritme detekcije za identifikaciju točke ulaska pikado strelice u metu. Ovo uključuje analizu slike za detekciju promjena na ploči uzrokovanih ulaskom strelice.
- Validacija postotka uspješnosti i praćenje rezultata. Konačni korak uključuje evaluaciju performansi sustava kroz validaciju postotka uspješnosti detekcije i praćenja.

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

Zadatak zadan:	Datum predaje rada:	Predviđeni datumi obrane:
7. ožujka 2024.	9. svibnja 2024.	13. – 17. svibnja 2024.

Zadatak zadao: Predsjednik Povjerenstva:

Doc. dr. sc. Filip Šuligoj

Prof. dr. sc. Ivica Garašić

SADRŽAJ

POPIS SLIKA

POPIS TABLICA

POPIS OZNAKA

Oznaka	Jedinica	Opis
$\hat{x}_w$	-	*x*-os *world* koordinatnog sustava
$\hat{y}_w$	-	*y*-os *world* koordinatnog sustava
$\hat{z}_w$	-	*z*-os *world koordinatnog sustava*
$\boldsymbol{x}_w$	-	vektor položaja točke P u *world* koordinatnom sustavu
$\hat{x}_c$	-	*x*-os koordinatnog sustava kamere
$\hat{y}_c$	-	*y*-os koordinatnog sustava kamere
$\hat{z}_c$	-	*z*-os koordinatnog sustava kamere
$\boldsymbol{x}_c$	-	vektor položaja točke P u koordinatnom sustavu kamere
f	mm	fokalna (žarišna) duljina
$\hat{x}_i$	-	*x*-os koordinatnog sustava ravnine slike
$\hat{y}_i$	-	*y*-os koordinatnog sustava ravnine slike
$\boldsymbol{x}_i$	-	vektor položaja točke P u koordinatnom sustavu ravnine slike
m_x	pikseli/mm	gustoća piksela u smjeru osi *x*
m_y	pikseli/mm	gustoća piksela u smjeru osi *y*
x_{dist}	mm	*x* koordinata distorziranog piksela
y_{dist}	mm	*y* koordinata distorziranog piksela
k_1	-	koeficijent radijalne distorzije
k_2	-	koeficijent radijalne distorzije
k_3	-	koeficijent radijalne distorzije
p_1	-	koeficijent tangencijalne distorzije
p_2	-	koeficijent tangencijalne distorzije
φ	°	kut koji zatvara pozicija točke P_1 s horizontalnom osi *x*
θ	°	kut koji zatvara pozicija točke P_1 s pozicijom točke P_2
a_{ij}	-	parametri afine transformacijske matrice
h_{ij}	-	parametri projektivne transformacijske matrice
x_o	piksel	*x* koordinata točke u odredišnoj slici
y_o	piksel	*y* koordinata točke u odredišnoj slici
x_i	piksel	*x* koordinata točke u izvornoj slici
y_i	piksel	*y* koordinata točke u izvornoj slici

POPIS KRATICA

Oznaka	**Opis**
OpenCV	eng. *Open Source Computer Vision Library*
SBC	eng. *Single Board Computer*
LED	eng. *Light-emmiting diode*
FPS	eng. *Frames per second*
CAD	eng. *Computer Aided Design*
DIN	DIN-Norm; njem. *Deutsches Institut für Normung*
FOV	eng. *Field of View*
USB	eng. *Universal serial bus*
CMOS	eng. *Complementary Metal-Oxide-Semiconductor*
AR	eng. *Augumented Reality*

SAŽETAK

U današnjem je svijetu naglasak na konstantan napredak različitih segmenata društva. Napredak društva uvjetuje manjak dostupnog slobodnog vremena koje je potrebno svakom pojedincu radi odmora od napornog tempa života. Shodno tome, potreba za poboljšanjem te optimiranjem rekreacijskih aktivnosti sve više raste. Jedan od načina optimiranja je mogućnost automatizacije aktivnosti kako bi se uštedjelo na vremenu i omogućila bezbrižnost pojedinaca. U tom je kontekstu razvijen vizijski sustav za jedan od popularnijih sportova današnjice – pikado. Cilj rada je korištenje vizijskog sustava za automatsko bodovanje bačene strelice. Za potrebe rada vizijskog sustava razvijen je fizički postav koji omogućuje stabilnost i robusnost u radu algoritma. Fizički postav sadrži 3 kamere koje omogućuju razne kadrove pikado mete, prikladno osvjetljenje koje osigurava konstantne uvjete važne za rad algoritma te *Jetson Nano* računalo. Kako bi algoritam mogao ispravno funkcionirati, izvršena je intrinzična kalibracija kamera uz obuhvaćanje distorzijskih koeficijenata. Zatim je razvijen algoritam koji omogućuje izračun matrica homografija za svaku kameru. Tako je osigurano pravilno pronalaženje pozicije vrha strelice te prikladno bodovanje s obzirom na pronađenu poziciju. Za bodovanje je razvijen algoritam koji u polarnim koordinatama definira udaljenost i kut točke za precizno određivanje bodova. Kako bi prethodni algoritmi funkcionirali, razvijen je ključan algoritam pronalaska vrha na temelju detekcije i obrade konture strelice u kadru kamera. U konačnici je napisan algoritam koji objedinjuje sve procese i omogućava paralelan rad triju kamera koje glasovanjem odlučuju o postignutom rezultatu. Na kraju je izvršena evaluacija rezultata koja daje pojam o točnosti i nedostacima razvijenog sustava.

Ključne riječi: vizijski sustav, intrinzična kalibracija, matrica homografije, ispravljanje perspektive kamere, detekcija konture, obrada slike, paralelno procesiranje

SUMMARY

Today's world requires the constant advancement of various segments of society. This progress leads to a lack of leisure time, which is necessary for each individual's mental health. Hence, the need for recreational activity optimization keeps rising. One way of optimizing is by using automation techniques in order to save time and enable the careless engagement of individuals during these activities. Hence, a vision system has been developed for a very popular sport - darts. The goal of this master thesis is to use a vision system for automatic determination of scored points. In order to achieve that, a physical setup has been developed to provide stability and robustness of the algorithm. The physical setup consists of three cameras placed around the dartboard, offering different fields of view. Suitable lighting has also been secured in order to ensure constant conditions. Finally, *Jetson Nano* provides the necessary integration of all process participants. The next key step was the intrinsic camera calibration in order to ensure correct algorithmic operation. An algorithm for each camera's homography matrix has been developed in order to precisely detect the dart's tip position. Furthermore, an algorithm for scoring that uses polar coordinates to describe the point position has been developed. Finally, an algorithm for detecting the dart's tip, which consists of different image processing techniques, has also been developed. In order to enable parallel execution of the mentioned algorithms, a main algorithm using multiprocessing techniques integrates all algorithms and offers real-time score tracking. Finally, an evaluation of these algorithms has been conducted to assess the quality and drawbacks of this vision system.

Key words: vision system, intrinsic calibration, homography matrix, camera perspective correction, contour detection, image processing, multiprocessing

1. UVOD

Računalni vid je grana umjetne inteligencije koja omogućuje računalima da interpretiraju vizualne informacije dobivene iz slika, videozapisa i sl. Razvitak ovog područja počinje 1956. godine na *Dartmouth* konferenciji. Tamo se prvi puta spominje termin umjetne inteligencije čiji je razvitak nužan za razvoj računalnog vida. Sljedeće se godine pojavljuje razvijeni projekt *Perceptrona*, algoritma koji predstavlja neuronsku mrežu dizajniranu za prepoznavanje i klasificiranje jednostavnih vizualnih uzoraka. Područje računalnog vida se tako nastavlja razvijati sve do današnjih projekata koji uključuju razne primijene u područjima autonomnih vozila, poljoprivrede, medicine, proizvodnje i sl. Ova se grana temelji na principu sličnom onom koji je prisutan kod ljudi; korištenje očiju za prikupljanje slika te mozga koji interpretira vizualne ulaze stvarajući informacije na temelju kojih ljudi vrše određene akcije. Kako bi ovaj princip funkcionirao u računalnom svijetu, bilo je potrebno razviti razne algoritme i tehnike za procesiranje i analizu vizualnih ulaza (slika, videozapisa). Tako je danas omogućeno korištenje računalnog vida u razne svrhe. U ovom je radu potrebno iskoristiti područje računalnog vida u više različitih segmenata procesa računanja bodova. Prije svega, potrebno je osmisliti stabilan i robustan fizički postav koji će osigurati nepomičnost svih elemenata. Osim glavnog nosećeg postava, potrebno je modelirati nosače kamera te osigurati stabilno osvjetljenje u kojem algoritam može neometano raditi. Nakon kreiranja dobrog fizičkog postava koji je nužan preduvjet za mogućnost provođenja daljnjih koraka, slijedi problematika integracije elemenata. Dakle, potrebno je omogućiti pokretanje i zajedničko međudjelovanje kamera. Nadalje je potrebno provesti proces kalibracije kamera kako bi se uklonile smetnje poput distorzija koje negativno utječu na točnost algoritma. Zasebno podešavanje svih elemenata omogućuje daljnju provedbu softverskog dijela zadatka. Jedan od problema tog dijela zadatka je pronaći vrh strelice. Postoje razne metode obrade slike, a neke od njih se temelje na razlikama objekata u bojama, obliku, veličini i sl. Nužno je iskoristiti neka od prethodno navedenih svojstava kako bi se pronašao prethodno spomenuti vrh. Nakon uspješnog pronalaska vrha strelice, potrebno je osmisliti način određivanja bodova za bačenu strelicu. U tom segmentu također postoji više različitih pristupa koji uključuju obradu slike utemeljenu na raznim karakteristikama objekata, matematičke pristupe i sl. U konačnici je potrebno objediniti paralelan rad svih kamera pri izvršavanju prethodno spomenutih algoritama.

1.1. ELEMENTI SUSTAVA

Kako bi se započelo s razvitkom cjelokupnog vizijskog sustava, potrebno je odabrati elemente koji sačinjavaju sustav te koji zadovoljavaju kriterije za uspješno izvršavanje zadaće prilikom igranja pikada. Prvo je potrebno odabrati kamere koje će biti dostatne za motrenje ploče iz različitih kuteva kako bi se precizno odredila pozicija strelice za bilježenje rezultata. Za odabir kamere korištene su smjernice koje podrazumijevaju čim manju cijenu uz dovoljno dobru kvalitetu slike kako bi projekt bio provodljiv. Osim toga, poželjno je odabrati kameru s manualnim fokusom i podešavanjem parametara poput zasićenja, kontrasta, ekspozicije i sl. Pošto je okolina prilično statična, nije potrebno tražiti kameru s većim *framerate*-om. Vodeći se ovim smjernicama odabrana je *web* kamera *White Shark GWC-004 Owl.* Uz prethodno spomenute smjernice, pogodna je okolnost što je vrlo lako izvaditi kameru iz njenog držača i montirati je na vlastiti držač. Prikaz kamere i općenite karakteristike dane su u nastavku.

Slika 1.1. Odabrana kamera - *White Shark GWC-004 Owl* [1]

Tablica 1.1. Karakteristike odabrane kamere [1]

Rezolucija	1920x1080, 2 MP
Fokus	manualni
Framerate	30 FPS
Senzor	CMOS 2.0 Mpix SONIX 2279 + 2053
Povezivost	USB 2.0

Nakon odabira kamere, slijedi odabir sustava koji će povezati 3 kamere u cjelinu i izvršavati algoritam koji će omogućiti iskorištavanje informacija s prikupljenih slika u svrhu izračuna pogođenog rezultata. Pogodnim se izborom pokazao *Jetson Nano Developer Kit*. Predstavlja maleno i moćno računalo koje je razvila tvrtka NVIDIA. Ovaj je SBC pogodan za programere i entuzijaste koje zanima rad u području umjetne inteligencije, odnosno računalnog vida jer omogućuje brzo izvršavanje zahtjevnih algoritama koji se koriste u ovim granama. Koristi se za real-time analizu slika, detekciju objekata, segmentaciju i sl. Osim ovog računala NVIDIA je razvila i druga koja se koriste ovisno o zadaći koju moraju izvršavati. Od velike je koristi činjenica da postoji dobra podrška korisnicima koji mogu koristiti *open-source* projekte koje svakodnevno stvara aktivna zajednica pojedinaca. Neki od kriterija za odabir upravo ovog SBC-a su svakako njegova veličina, mogućnost povezivanja triju kamera i dovoljna snaga za obradu podataka u ovom projektu.[2] Prikaz *Jetson Nano Developer Kit*-a i njegovih karakteristika slijedi u nastavku.

Slika 1.2. *Jetson Nano Developer Kit* [2]

Tablica 1.2. Karakteristike *Jetson Nano Developer Kit*-a [2]

GPU	128-jezgri, Maxwell
CPU	četvero-jezgreni ARM A57 @ 1.43 GHz
RADNA MEMORIJA	4 GB 64-bit LPDDR4 25.6 GB/s
POHRANA	microSD
USB	4x USB 3.0, USB 2.0 Micro-B
DIMENZIJE	69.6 mm x 45 mm

Vrijedi napomenuti kako se, uslijed manjka USB ulaza na *Jetson Nanou*, kamere prvotno spajaju na USB *hub*, a zatim se *hub* spaja na jedan od USB ulaza na SBC-u.

1.2. CJELOKUPNI POSTAV

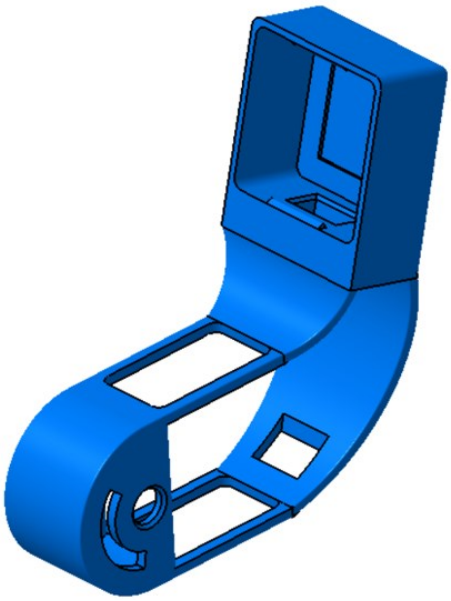
Korištena su dva različita postava. Prilikom razvijanja cjelokupnog sustava, korišten je postojeći kućni postav uz dodatne nadogradnje (montiranje kamera i nosača SBC-a). U ovom se slučaju koristi pikado ploča *Gladiator* 3 proizvođača *One*80. Radi se o jednoj od najkvalitetnijih ploča na tržištu; izrađena je od sisala koji predstavlja otporan i dugotrajan materijal. Sadrži okvir s brojevima koji ne stvara odsjaj te ožičenje koje je vrlo tanko i kvalitetno izrađeno čime se osigurava lakši upad strelice u ploču. Nadalje, sadrži *Rotafix* nosač koji se lagano montira i omogućuje zakretanje ploče. Naime, s vremena na vrijeme je potrebno zakrenuti ploču za par polja kako bi se vršilo jednako trošenje svih segmenata (polja) na ploči. U pravilu korisnik mora skinuti ploču i ponovno je montirati na zid u drugačijoj orijentaciji, ali ovaj nosač omogućuje zakretanje ploče bez skidanja iste.[3]

Slika 1.3. Pikado ploča *Gladiator* 3 proizvođača *One80* [3]

U prvotnom je postavu ploča montirana na šperploču dimenzija 90x80 cm, dok je šperploča montirana na držače osigurane tiplama u zidu. Radi ljepšeg izgleda postava zalijepljeni su pluto čepovi. Nadalje je potrebno razraditi problem osvjetljenja pikado ploče. Prednosti dobrog osvjetljenja su uklanjanje bilo kakve sjene od strelice koja je upiknuta u ploču te dosta poboljšana vidljivost. Prva je prednost prilično važna za potrebe dobrog rada vizijskog sustava jer sjene mogu predstavljati velike smetnje prilikom lociranja strelica. Kako bi se sjena u potpunosti uklonila, potrebno je imati kružno osvjetljenje (360 stupnjeva). Za te je potrebe 3D printan LED prsten koji je dostupan na internetu.[4] Također je kupljena LED traka koja odgovara opsegu isprintanog prstena te zalijepljena na isti. Nakon rješavanja problema osvjetljenja slijedi montaža kamera na šperploču. Vrlo je važno pripaziti na sve aspekte montaže kamera:

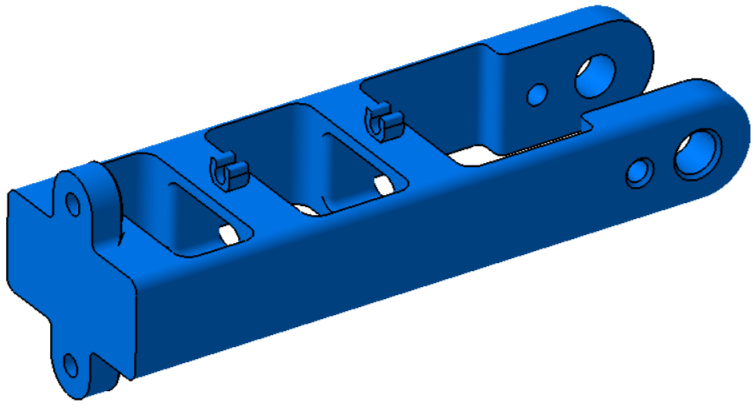
- međusobno jednako udaljene u rasponu 360 stupnjeva
- osiguravanje dobrog pogleda na pikado ploču
- osiguravanje nepomičnosti kamera
- smještanje USB kablova

Kako bi svi aspekti bili zadovoljeni, izrađen je CAD model držača kamere koji je kasnije 3D printan. CAD modeli stvaraju se u softveru *CATIA V5R21.* Zbog dimenzija i mogućnosti printera, nosač je izrađen iz tri dijela koja se mogu spojiti imbus vijcima norme DIN 912. Prvi dio nosača predstavlja držač (kućište) kamere koji je prikazan na slici ispod.



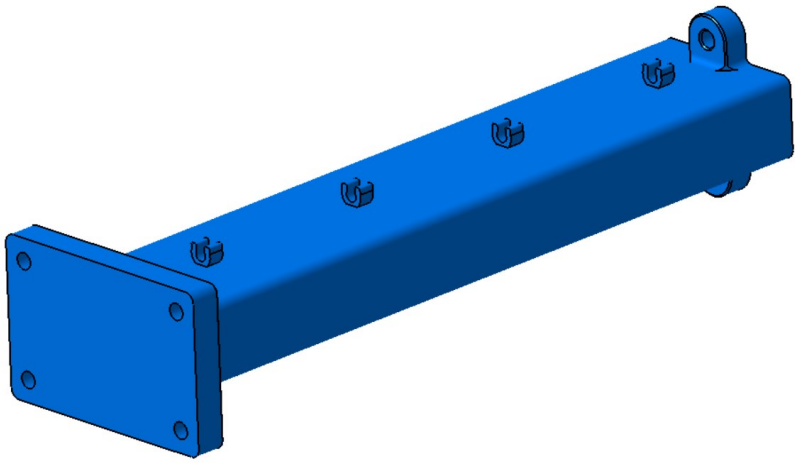
Slika 1.4. CAD model prvog dijela nosača kamere

Unutrašnjost nosača sadrži utor u koji sjeda dodatan nosač na kojem se nalazi kamera. Maleni izdanci pri samom vrhu nosača služe za *snap fit* odnos između nosača i poklopca koji će biti prikazan u nastavku. U unutrašnjosti nosača napravljen je provrt koji izlazi na stražnju stranu nosača kako bi se USB kabel mogao elegantno provesti tako da ne smeta u montaži. U donjem lijevom kutu vidljiv je provrt te dodatan provrt u obliku kružnog luka. Cilindrični provrt omogućuje osiguranje pozicije prvog dijela nosača kamere na drugom dijelu koji će biti prikazan u nastavku. Kružni luk omogućuje zakret nosača kamere radi namještanja FOV-a kamera. U nastavku je prikazan prethodno spomenuti drugi dio nosača kamere.



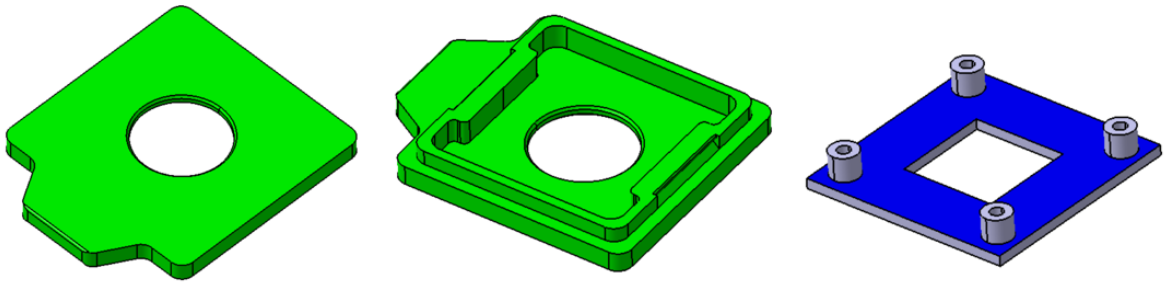
Slika 1.5. CAD model drugog dijela nosača kamere

Na drugom dijelu nosača kamere, na desnom kraju, vidljive su dvije prolazne rupe koje služe za fizičku vezu s prvim dijelom nosača kamere. U sredini su vidljive vodilice koje služe za osiguranje pozicije USB kabela kamere, dok se na lijevom kraju vide dvije prolazne rupe za vezivanje drugog dijela s trećim dijelom nosača kamere. Isti je prikazan na slici ispod.



Slika 1.6. CAD model trećeg dijela nosača kamera

Na gornjem dijelu trećeg komada nosača napravljen je oblik koji prianja na drugi dio nosača. Valja napomenuti kako krajnji dio nosača sadrži 4 provrta s upustom koji služe za montažu nosača na šperploču. Nakon izrade nosača kamera, potrebno je izraditi dodatan nosač na kojem se nalazi kamera. Razlog korištenja dodatnog nosača je praktične prirode; uvelike olakšana montaža kamere s pomoću jakog ljepila koje osigurava nepomičnost kamere. Za montažu kamere na dodatan nosač koriste se vijci norme DIN 84. U konačnici je preostalo izraditi poklopac držača koji sadrži izdanak za lakše uklanjanje s držača. Osim toga sadrži i središnji provrt za leću kamere. Prethodno spomenute stavke su prikazane na slici ispod.



a) b)

Slika 1.7. a) poklopac nosača, b) dodatan nosač kamere

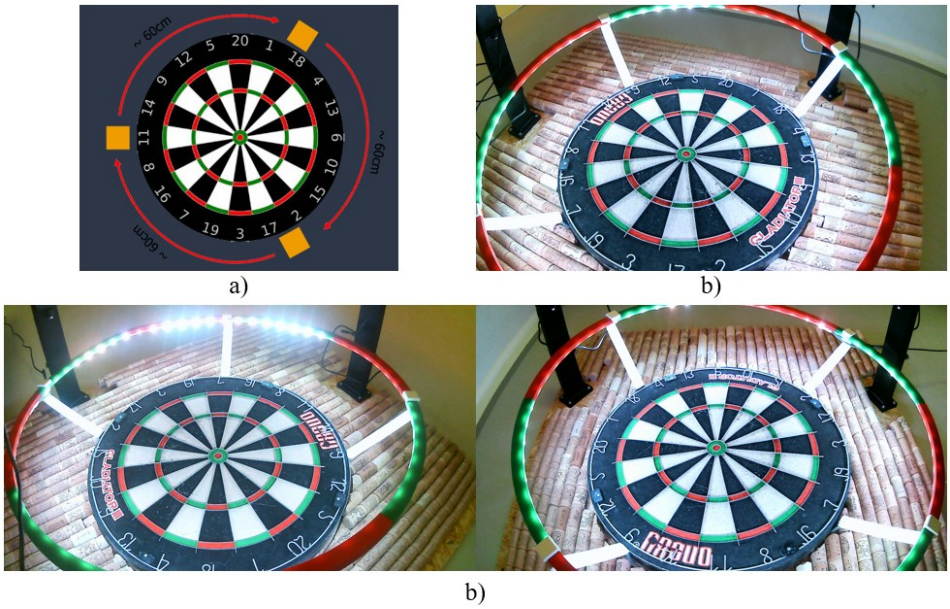
Konačan prikaz unutrašnjosti te cjelokupno spojenog nosača s montiranom kamerom dan je na slici ispod. Potrebno je napomenuti kako slika ispod prikazuje prvotni oblik nosača koji je kasnije promijenjen radi osiguravanja dodatnih mogućnosti (promjena kuta kamera).

a) b)

Slika 1.8. a) kućište nosača bez i s poklopcem, b) montirani cjelokupni nosač

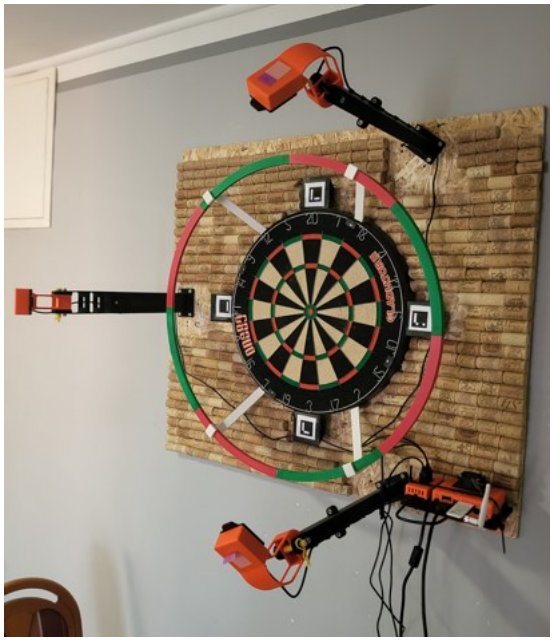
Nakon osiguravanja kamera, potrebno je smjestiti *Jetson Nano* u prikladno kućište. Kućište koje se koristi u ovom radu preuzeto je sa stranice dane u literaturi.[5]

Izradom i sklapanjem svih potrebnih dijelova za rad sustava slijedi montaža nosača na šperploču. Ovo predstavlja vrlo važan korak u kojem je nužno osigurati približno jednaku međusobnu udaljenost nosača u rasponu 360 stupnjeva. Posljedica toga je dobra pokrivenost cijele ploče čime se osigurava precizno lociranje strelice na ploči. Način montaže te rezultati iste dani su na slikama ispod.



Slika 1.9. a) pozicioniranje nosača kamera na šperploči [6], b) *FOV* triju kamera

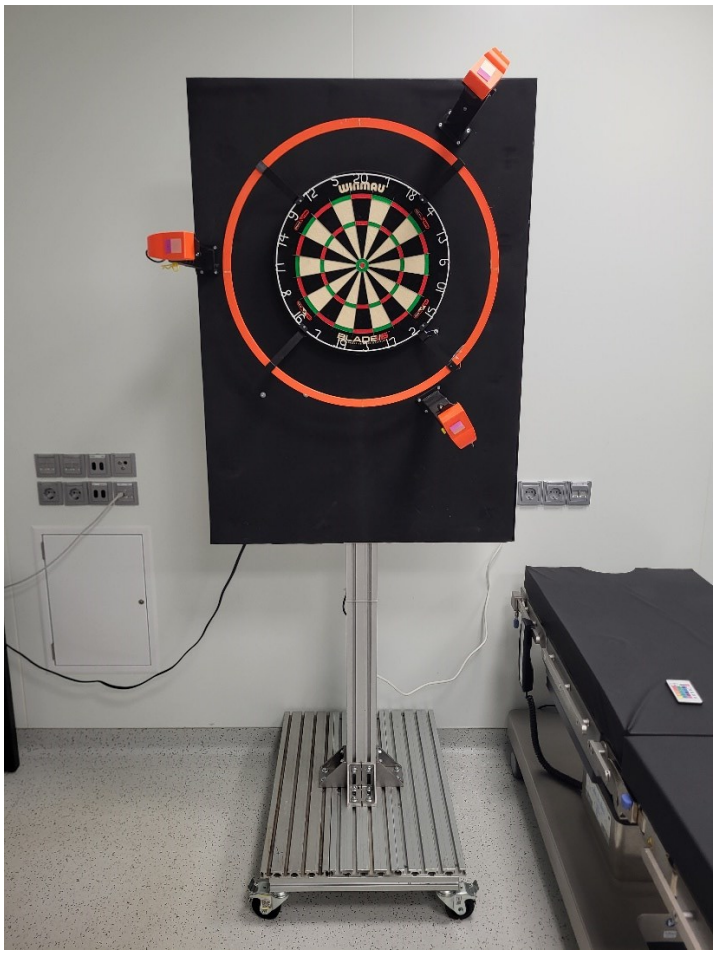
Cjelokupni postav prikazan je na slici ispod.



Slika 1.10. Cjelokupni kućni postav

Važno je napomenuti kako su na slici vidljivi i *AruCo* markeri koji se koriste pri kreiranju matrice homografije za ispravljanje perspektive slike. Kasnije se ti markeri neće koristiti radi smanjenja nepotrebne opreme za osiguravanje pravilnog rada sustava što će kasnije biti detaljnije objašnjeno. Kompletan algoritam je izrađen i testiran s obzirom na prethodno prikazani postav. Nakon potpune izrade, ekvivalentan postav je repliciran u CRTA-i uz određene nadogradnje i promjene:

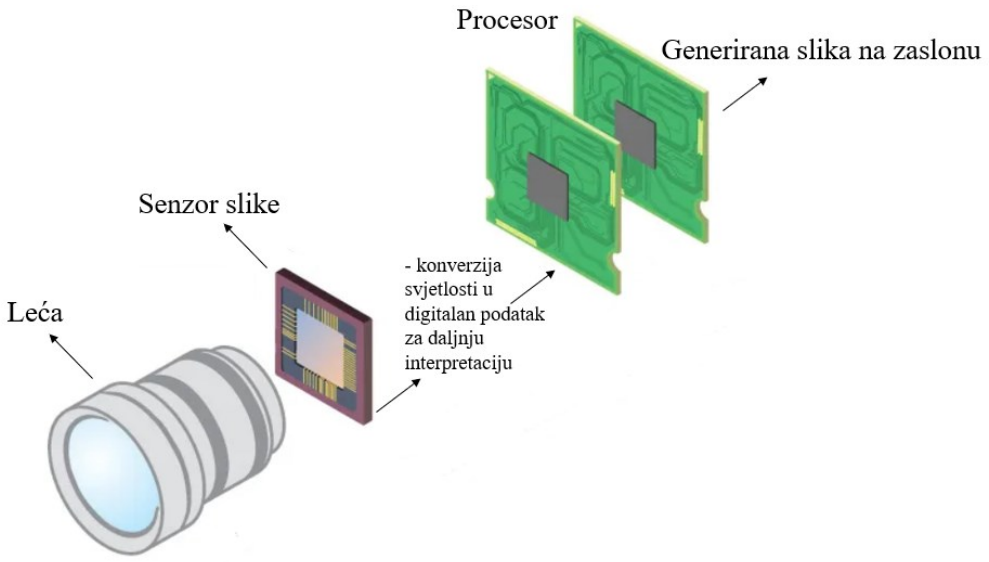
- izrađen je mobilni postav – 4 kotača, jedan profil 90x90x200 cm na koji je montiran nosač postava (pleksiglas)
- koristi se pikado meta drugog proizvođača
- *Jetson Nano* i potrebno napajanje elemenata je montirano sa stražnje strane pleksiglasa radi zaštite istih
- dodatna zaštita (mekana podloga) radi umanjenja oštećenja/padanja strelica, uniformna pozadina



Slika 1.11. Laboratorijski pikado postav

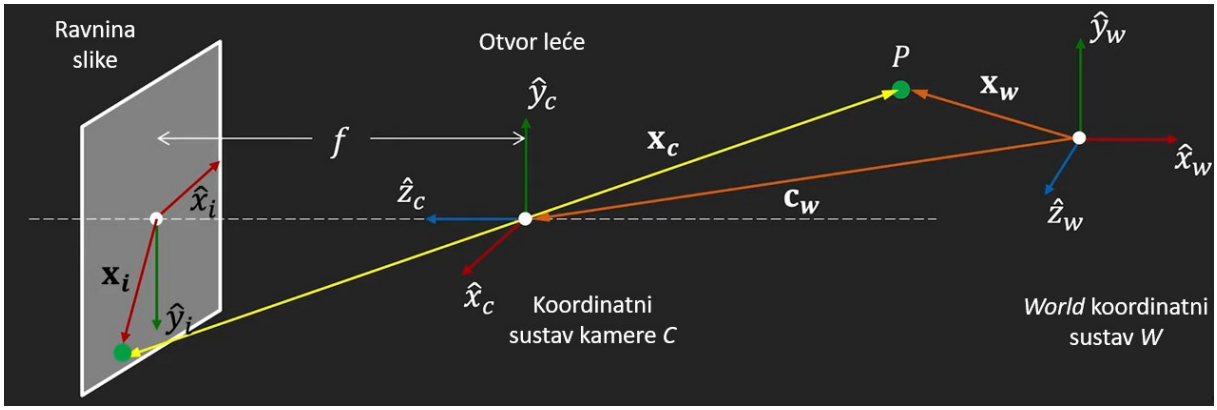
2. KALIBRACIJA KAMERA

Nakon uspješno izrađenog fizičkog postava te osiguravanja svih potrebnih uvjeta za pravilan rad algoritama slijedi provedba kalibracije kamera bez koje sustav ne bi pravilno funkcionirao. Kamere su samo jedan u nizu ljudskih proizvoda koji ima svoja ograničenja i nepravilnosti koje je nužno uzeti u obzir. Prije nepravilnosti je nužno razumjeti što čini jednu kameru. U ovom se radu koriste *web* kamere koje su klasično sačinjene od neke vrste senzora slike, leće i procesora. Senzor slike je vrlo važna komponenta svakog sustava koji se danas najčešće proizvodi od silicija. Razlog korištenja silicija krije se u činjenici da svaki njegov atom na foton dovoljne energije reagira otpuštanjem elektrona. Dakle, usmjeravanje snopa fotona na velik broj atoma silicija rezultira stvaranjem snopa elektrona koji nose informaciju o generiranom naboju. Taj se električni signal interpretira za stvaranje slike. Što je veći broj piksela (elementi osjetljivi na svjetlost), to je veća rezolucija slike što rezultira mogućnošću obuhvaćanja finijih detalja slike. U prisutnim *web* kamerama koristi se CMOS tehnologija. U CMOS senzorima svjetlost pada na piksele senzora čime se generira prethodno spomenuti električni naboj. Zatim se taj naboj procesira u podatak koji predstavlja intenzitet svjetlosti prisutan na tom pikselu. Nakon toga slijedi generiranje slike s pomoću procesora koji obuhvaća informacije s mnoštva piksela i vrši obradu istih u svrhu stvaranja smislene informacije (cjelokupne slike).[7]



Slika 2.1. Proces generiranja slike *web* kamera [8]

Svjetlo koje se projicira na senzor slike prvo prolazi kroz svojevrsni filter, odnosno leću. Leća je odgovorna za filtriranje snopa svjetlosti te usmjeravanje istog na senzor slike gdje dolazi do prethodno objašnjenog procesa. Izrađene su od staklenih ploča koje su konveksne ili konkavne. Može se zaključiti kako već pri izradi staklenih ploča može doći do generiranja pogrešaka prilikom usmjeravanja svjetlosti. Upravo se takve pogreške nesavršenosti izrade konveksnih i konkavnih oblika mogu ispraviti kalibracijom kamera. Kalibracija kamera predstavlja proces estimacije parametara leće i senzora slike. Ti parametri definiraju način na koji se 3D točka u stvarnom svijetu preslikava na prethodno spomenuti 2D senzor slike. Radi boljeg uviđaja i objašnjenja parametra, vrijedi definirati sve parametre i objasniti proces formiranja slike prema tzv. *pinhole* modelu kamere.



Slika 2.2. Pinhole model kamere [9]

Na desnoj je strani slike vidljiv definiran *world* koordinatni sustav u odnosu na koji se određuje pozicija 3D točke P u stvarnom svijetu. U sredini je slike vidljiv definiran koordinatni sustav kamere čija z-os leži na optičkoj osi kamere. Optička os kamere predstavlja onu os koja je okomita na ravninu slike te koja prolazi optičkim centrom leće. Osim optičke osi, potrebno je definirati i fokalnu (žarišnu) duljinu *f.* Ona predstavlja udaljenost između ravnine slike i optičkog centra leće kamere. Na lijevom je kraju slike definiran koordinatni sustav ravnine slike. Time su definirani svi potrebni elementi potrebni za projekciju 3D točke *world* koordinatnog sustava u 2D točku koordinatnog sustava ravnine slike. Važno je napomenuti kako pojam *pinhole* predstavlja leću s malim otvorom koji usmjerava zrake na senzor slike. Bez leće takve izvedbe, bilo bi nemoguće dobiti projekciju slike pošto bi snopovi svjetlosnih zraka 3D točaka bili projicirani u mnoštvo smjerova na senzor slike. Time bi se praktički dobila čista bijela slika, što ujedno predstavlja mnoštvo izgubljenih informacija.[9]

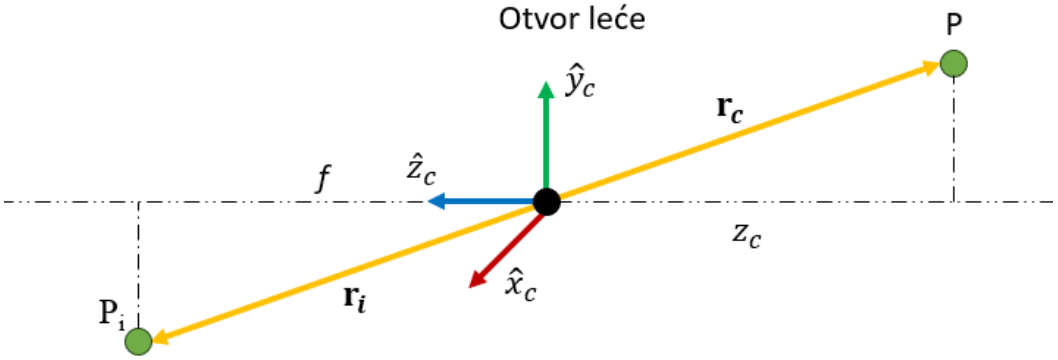
Za određivanje transformacije 3D točke P u 2D točku P_i na senzoru slike, potrebno je izvršiti dvije transformacije:

1) poznat položaj točke P u *world* koordinatnom sustavu ($\mathbf{x_w}$) potrebno je transformirati u koordinatni sustav kamere ($\mathbf{x_c}$) – transformacija 3D u 3D koordinate – ekstrinzični parametri kamere
2) poznat položaj točke P u koordinatnom sustavu kamere ($\mathbf{x_c}$) potrebno je transformirati u koordinatni sustav ravnine slike ($\mathbf{x_i}$) – transformacija 3D u 2D koordinate – intrinzični parametri kamere

U sklopu ovog rada potrebno je odrediti intrinzične parametre kamere, stoga je poželjno poznavati jednadžbe koje određuju te parametre.

2.1. PERSPEKTIVNA PROJEKCIJA (INTRINZIČNI PARAMETRI)

Za određivanje intrinzičnih parametara te definiranje intrinzične matrice kamere, potrebno je odrediti jednadžbe perspektivne projekcije. Na temelju slike [2.2.], vidljiva su sljedeća dva slična trokuta:



Slika 2.3. Određivanje jednadžbi perspektivne projekcije

Ako se vektori $\mathbf{r_c}$ i $\mathbf{r_i}$ definiraju sljedećim redoslijedom: (x_c, y_c, z_c) i (x_i, y_i, z_i), tada se jednadžbe perspektivne projekcije mogu pisati na sljedeći način:

$$\frac{\mathbf{r_i}}{f} = \frac{\mathbf{r_c}}{z_c},$$

odnosno:

$$\frac{x_i}{f} = \frac{x_c}{z_c}\ ;\ \frac{y_i}{f} = \frac{y_c}{z_c}. \tag{1}$$

Prethodno dobiveni izrazi u jednadžbi (1) predstavljaju jednadžbe perspektivne projekcije. Važno je napomenuti kako se u ovom slučaju pretpostavlja da je točka P definirana u koordinatnom sustavu kamere. Ako se dodatno uredi prethodno dobivena jednadžba, jednostavno se može doći do pozicije točke P u ravnini slike:

$$x_i = f\frac{x_c}{z_c}\ ;\ y_i = f\frac{y_c}{z_c}. \tag{2}$$

Prethodno dobivena jednadžba podrazumijeva da su koordinate izražene u [mm], odnosno u jednakoj mjernoj jedinici kojom je definirana ta ista točka u koordinatnom sustavu kamere. U stvarnosti zapravo imamo senzor slike čija je mjerna jedinica izražena u pikselima. Dakle, dodatno je potrebno odrediti transformaciju iz [mm] u piksele. Prethodno spomenuta ravnina slike može se povezati sa senzorom slike preko gustoće piksela.[9] Dakle, ako kažemo da m_x i m_y predstavljaju gustoću piksela [pikseli/mm] u smjerovima osi *x* i *y,* tada su koordinate točke P u ravnini senzora slike jednake:

$$u = m_x * x_i\ ; v = m_y * y_i. \tag{3}$$

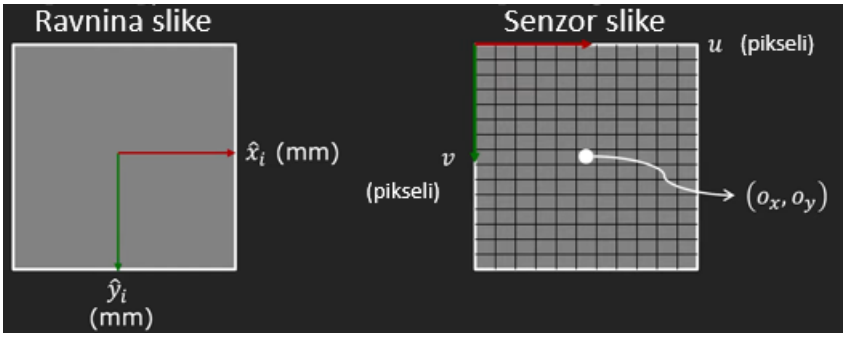
Prethodna jednadžba izvedena je pod uvjetom da se centri ravnine slike i senzora slike poklapaju, odnosno da su oba koordinatna sustava definirana u mjestu gdje optička os probada ravninu slike. No, koordinatni sustav senzora slike ne mora nužno biti u tom istom centru koji se inače naziva principijelnom točkom. Stoga se koordinatni sustav senzora slike može definirati u jednom od kuteva ravnine senzora slike, recimo u gornjem lijevom kutu. Ako se sada ponovno definira jednadžba (3), dobije se sljedeći set jednadžbi:

$$u = m_x * x_i + o_x\ ; v = m_y * y_i + o_y, \tag{4}$$

pri čemu (o_x, o_y) predstavlja poziciju principijelne točke. Važno je napomenuti kako je u prethodnoj jednadžbi nepoznata gustoća piksela, fokalna duljina i principijelna točka. Dodatnim uređenjem jednadžbe (4) može se napisati:

$$u = f_x * \frac{x_c}{z_c} + o_x\ ; v = f_y * \frac{y_c}{z_c} + o_y, \tag{5}$$

pri čemu je $f_x = m_x * f$ i $f_y = m_y * f$. Ta dva izraza predstavljaju fokalnu duljinu izraženu u pikselima u smjeru osi *x* i *y*. Najčešće je kod kamera prisutna jedna fokalna duljina, no u slučaju različite razdiobe gustoće piksela u smjeru pojedine osi (ovisno o obliku piksela) dobro je definirati fokalne duljine u tim pojedinim osima.[9] Prethodno prikazane jednadžbe temelje se na sljedećoj slici.



Slika 2.4. Ostvarivanje veze koordinatnog sustava ravnine slike i koordinatnog sustava senzora slike [9]

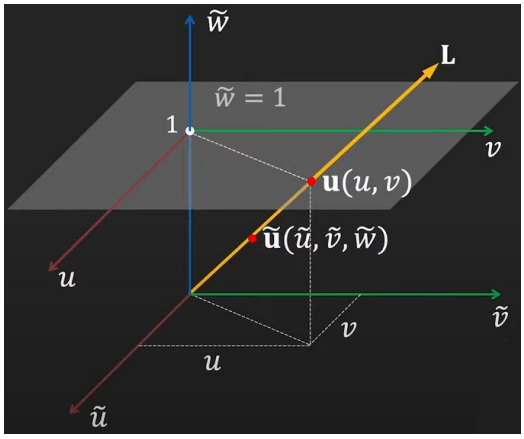
Kao što je prethodno rečeno, u jednadžbi (5) postoje 4 nepoznata člana – (f_x, f_y, o_x, o_y). Upravo ta 4 člana predstavljaju intrinzične parametre kamere. Fokalna duljina određuje vidno polje kamere, dok principijelna točka služi kao referenca za određivanje duljina i kuteva u slici. Set jednadžbi (5) predstavlja nelinearni model perspektivne projekcije koji je poželjno pretvoriti u linearni model kako bi cjelokupan proces bio lakše izvediv. Za te je potrebe potrebno uvesti homogene koordinate.

2.1.1. Homogene koordinate

Homogene se koordinate u polju računalnog vida koriste u svrhu omogućavanja stvaranja linearnog modela perspektivne projekcije, odnosno olakšanja izračuna translacija, rotacija te skaliranja prilikom matričnih množenja. Zapis točaka u polju homogenih koordinata podrazumijeva uvođenje dodatne fiktivne koordinate koja se u pravilu koristi za normalizaciju postojećih koordinata.[9] Primjerice, 2D točka definirana kao $\boldsymbol{u} = (u, v)$ u toj varijanti postaje 3D točka $\tilde{\boldsymbol{u}} = (\tilde{u}, \tilde{v}, \tilde{w})$. Koordinata $\tilde{w} \neq 0$ je fiktivna pri čemu vrijedi:

$$u = \frac{\tilde{u}}{\tilde{w}} \text{ i } v = \frac{\tilde{v}}{\tilde{w}}. \tag{6}$$

Na sljedećoj su slici prikazana dva koordinatna sustava, pri čemu je osnovni koordinatni sustav definiran tako da se njegova *xy* ravnina nalazi na fiktivnoj koordinati homogenog koordinatnog sustava vrijednosti 1.



Slika 2.5. Prikaz homogenih koordinata [9]

Sve točke (osim ishodišta) koje leže na pravcu **L** koji prolazi točkom **u** su međusobno ekvivalentne i jednake točki **u**. Također se može reći da svaka točka na tom pravcu predstavlja homogenu koordinatu točke **u**.[9] Prema tome, vrijedi sljedeći zapis:

$$\mathbf{u} \equiv \begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \tilde{w}u \\ \tilde{w}v \\ \tilde{w} \end{bmatrix} \equiv \begin{bmatrix} \tilde{u} \\ \tilde{v} \\ \tilde{w} \end{bmatrix} \equiv \tilde{\mathbf{u}}. \tag{7}$$

Točka **u** u homogenom koordinatnom sustavu se može reprezentirati prvom ekvivalentnošću – nalazi se na poziciji (u,v) na ravnini fiktivne koordinate vrijednosti 1. To je ekvivalentno skaliranjem bilo kojom vrijednošću $\tilde{w} \neq 0$. Ekvivalentna pravila se primjenjuju na 3D točku, primjerice onu u koordinatnom sustavu kamere ili *world* koordinatnom sustavu. U toj se varijanti 3D točka $\mathbf{x} = (x,y,z)$ u homogenim koordinatama može zapisati kao 4D točka na sljedeći način:

$$\mathbf{x} \equiv \begin{bmatrix} x \\ y \\ z \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \tilde{w}x \\ \tilde{w}y \\ \tilde{w}z \\ \tilde{w} \end{bmatrix} \equiv \begin{bmatrix} \tilde{x} \\ \tilde{y} \\ \tilde{z} \\ \tilde{w} \end{bmatrix} = \tilde{\mathbf{x}}. \tag{8}$$

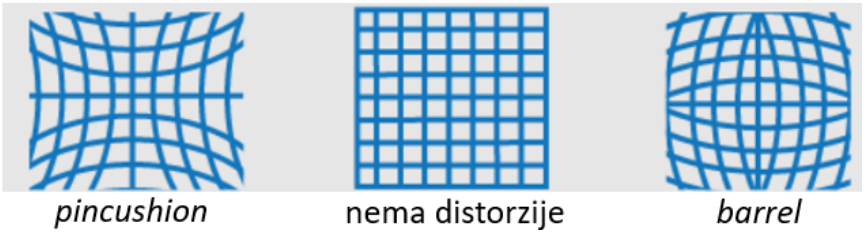
Dakle, prema istim pravilima vrijede i isti zaključci - u homogenim koordinatama točka **x** je ekvivalentna točki $\tilde{\mathbf{x}}$. Nakon uvođenja homogenih koordinata potrebno je ponovno svrnuti pažnju na set jednadžbi (5). Ako se točka $\mathbf{u} = (u,v)$ zapiše u homogenim koordinatama i pomnoži sa skalarom z_c, dobiva se sljedeći izraz:

$$\begin{bmatrix} u \\ v \\ 1 \end{bmatrix} \equiv \begin{bmatrix} z_c u \\ z_c v \\ z_c \end{bmatrix} \equiv \begin{bmatrix} f_x x_c + z_c o_x \\ f_y y_c + z_c o_y \\ z_c \end{bmatrix} \equiv \begin{bmatrix} f_x & 0 & o_x & 0 \\ 0 & f_y & o_y & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix} \begin{bmatrix} x_c \\ y_c \\ z_c \\ 1 \end{bmatrix}. \tag{9}$$

Iz prethodne je jednadžbe jasno vidljivo kako je dobiven linearan model perspektivne projekcije kojem je omogućen matrični zapis. Zadnji izraz predstavlja intrinzičnu matricu množenu s homogenim koordinatama 3D točke definirane u koordinatnom sustavu kamere. Upravo tu matricu je potrebno odrediti kako bi se izvršila kalibracija kamere. Važno je napomenuti kako u tu matricu nisu uključene distorzije. Razlog tome je što se izvedeni izrazi temelje na *pinhole* modelu kamere koji ne sadrži leću, a samim time ne sadrži ni imperfekcije u proizvodnji istih koje uzrokuju distorziju.

2.1.2. Distorzija

Osim intrinzičnih parametara kamere, potrebno je odrediti distorzijske koeficijente kako bi se distorzija mogla eliminirati u svrhu osiguranja mjerodavne slike prostora. Postoje dvije vrste distorzija: radijalna i tangencijalna. Radijalna se distorzija pojavljuje uslijed nesavršenosti pri izradi leće kamere. One uzrokuju da snopovi svjetlosti upadaju na senzor slike pod različitim kutevima, što je posebno primjetljivo na rubovima slike. Čim je leća manja, to je radijalna distorzija veća.[10] Na slici ispod su prikazane dvije vrste distorzije.



Slika 2.6. Vrste radijalne distorzije [10]

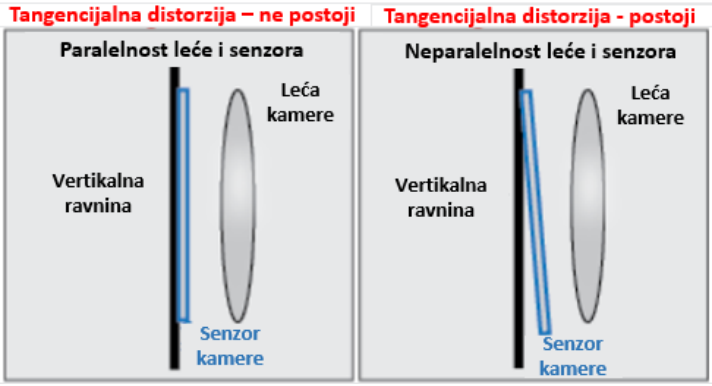
Vidljivo je kako se u varijanti *pincushion* distorzije ravne linije deformiraju prema rubu slike, dok se u *barrel* varijanti ravne linije deformiraju prema sredini slike. *Pincushion* distorzija rezultira negativnim distorzijskim koeficijentima, dok *barrel* distorzija rezultira pozitivnim distorzijskim koeficijentima. Radijalnu je distorziju moguće prikazati sljedećim jednadžbama:

$$x_{dist} = x(1 + k_1 r^2 + k_2 r^4 + k_3 r^6),$$
$$y_{dist} = y(1 + k_1 r^2 + k_2 r^4 + k_3 r^6), \quad (10)$$

pri čemu su:

- x_{dist}, y_{dist} – x i y koordinata distorziranog piksela,
- x, y – x i y koordinata nedistorziranog piksela,
- k_1, k_2, k_3 – radijalni distorzijski koeficijenti leće,
- $r^2 = x^2 + y^2$.

Potrebno je naglasiti kako su x i y koordinate normalizirane koordinate slike koje se izračunavaju tako da se koordinate točke pomiču u optički centar kamere kako bi se cijeli distorzijski postupak odvio oko principijelne točke kamere. Nakon toga se još dijele sa žarišnom duljinom u pikselima kako bi distorzija bila neovisna o rezoluciji i sl. Tako su x i y bezdimenzionalni. S druge strane je moguća i tangencijalna distorzija. Ona se javlja kao posljedica neparalelnosti leće i senzora kamere[10]:



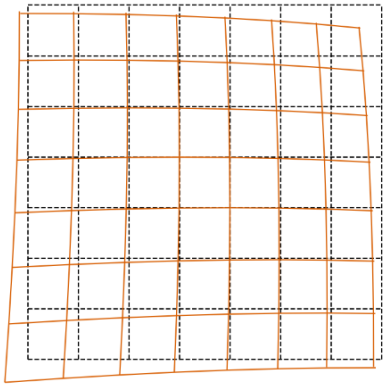
Slika 2.7. Uzrok pojave tangencijalne distorzije [10]

Tangencijalnu je distorziju moguće modelirati prema sljedećim jednadžbama:

$$x_{dist} = x + [2p_1xy + p_2(r^2 + 2x^2)],$$
$$y_{dist} = y + [p_1(r^2 + 2y^2) + 2p_2xy], \quad (11)$$

pri čemu su:

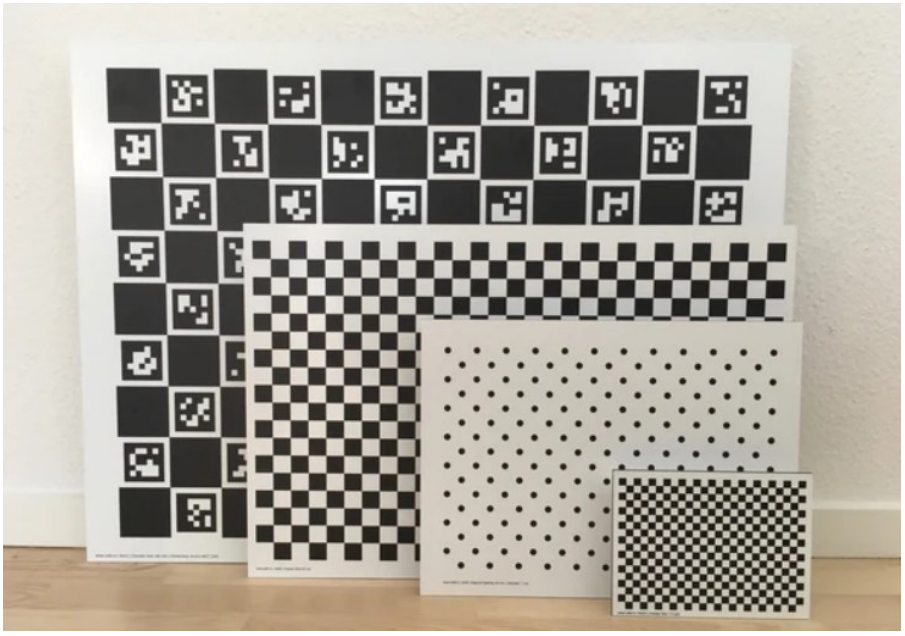
- x_{dist}, y_{dist} – x i y koordinata distorziranog piksela,
- x, y – x i y koordinata nedistorziranog piksela,
- p_1, p_2 – tangencijalni distorzijski koeficijenti leće,
- $r^2 = x^2 + y^2$.



Slika 2.8. Primjer tangencijalne distorzije [11]

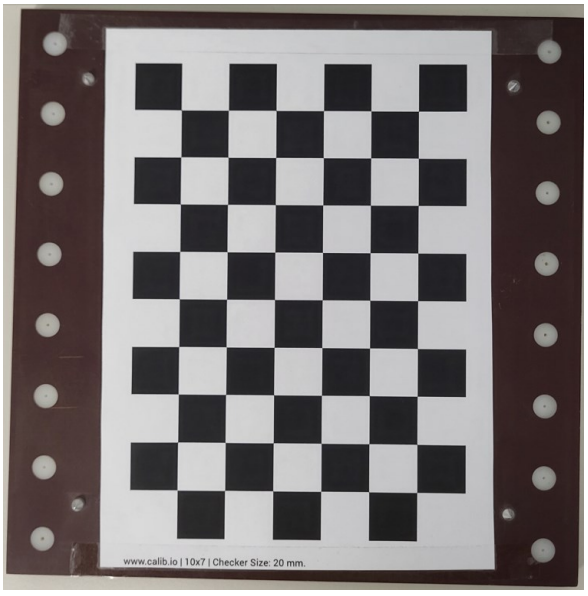
2.2. POSTUPAK KALIBRACIJE KAMERA

Kalibracija kamera se provodi u svrhu kompenzacije pogrešaka pri izradi kamera, odnosno njihovih dijelova. Pogreške pri izradi uzrokuju prethodno spomenute distorzije koje uvelike utječu na kvalitetu i mjerodavnost slike. Zbog toga je nužno provesti kalibraciju koja te distorzije uklanja i tako daje vjernu sliku. U tu je svrhu potrebno proći kroz proces kalibracije kako bi se mogla dobiti intrinzična matrica. Ista se zatim može, s pomoću spomenute biblioteke *OpenCV*, iskoristiti za primjenu na sliku/video. Nužno je odabrati način kalibracije o kojem ovisi cjelokupan proces. U ovom je radu odabrana kalibracija s pomoću kalibracijskog uzorka poznatih dimenzija. Kalibracijskih uzoraka ima mnogo, a neki od njih su prikazani na slici ispod.



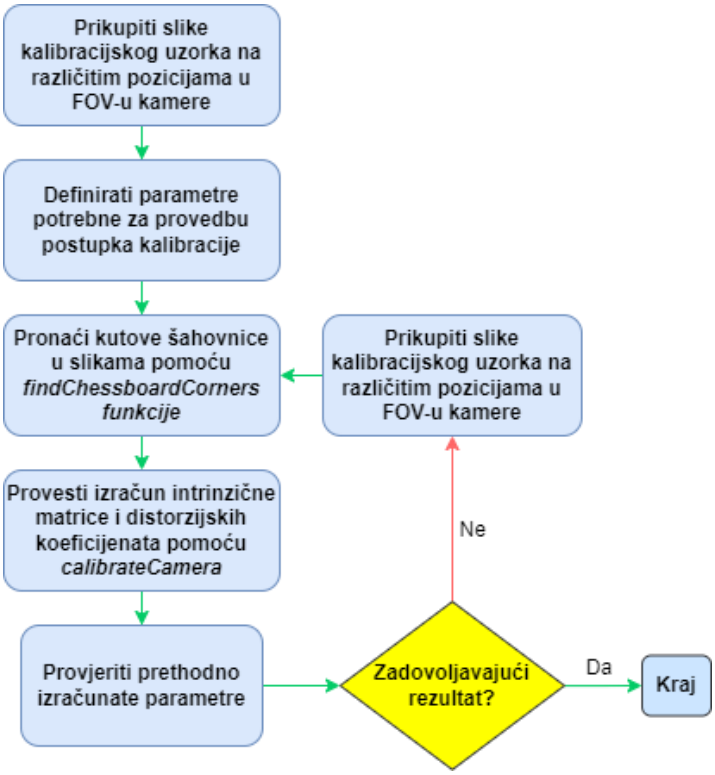
Slika 2.9. Razni kalibracijski uzorci [12]

U procesu odabira kalibracijskog uzorka potrebno je paziti na više faktora. Jedan od faktora je fizička veličina uzorka. Fizičku veličinu uzorka je potrebno odabrati s obzirom na FOV kamere. Najbolje je uzeti uzorak koji popunjava dobar dio slike koju generira *web* kamera. Sljedeći bitan uvjet je vrsta uzorka. Kao što je vidljivo na prethodnoj slici, postoji više različitih uzoraka. Jedan od najčešće korištenih uzoraka je šahovnica. Prednosti korištenja takvog uzorka su jednostavnost generiranja i printanja, robusnost, naglašen kontrast između crnih i bijelih polja. Vrlo je važno naglasiti kako je potrebno odabrati šahovnicu takvih dimenzija da broj redaka mora biti paran, a stupaca neparan. Isto vrijedi i u obrnutom slučaju. U slučaju da su oba parametra parne vrijednosti, dolazi do mogućnosti da algoritam u različitim slikama na različitim mjestima generira ishodište koordinatnog sustava. To je nužno spriječiti u svrhu osiguranja postojanja asimetričnosti u oba smjera i samim time točnosti izvedene kalibracije. Nadalje, postoje ChArUco kalibracijski uzorci koji, uz prethodno spomenutu šahovnicu, sadrže i ArUco markere u bijelim poljima šahovnice. Posljedica toga je mogućnost korištenja bijelih polja kao dodatne identifikacije uz jedinstvene oznake. Tako se osigurava mogućnost korištenja uzorka i u nepovoljnim uvjetima – nije potrebno imati cjelokupan uzorak u FOV-u kamere. Konačno, valja spomenuti kružne kalibracijske uzorke. Za korištenje istih je ponovno potrebno koristiti asimetrične uzorke kako bi se izbjegla prethodno spomenuta mogućnost greške pri izvođenju algoritma. Međutim, takvi se uzorci manje koriste u odnosu na prethodna dva tipa uzoraka pa samim time ima i manje dostupnih materijala za korištenje istih. S obzirom na to da je okružje u sklopu ovog rada relativno stabilno, odabran je prvi tip kalibracijskog uzorka.



Slika 2.10. Odabrani kalibracijski uzorak

Kao što je vidljivo na samom dnu prethodne slike, odabran je kalibracijski uzorak dimenzija 10x7 s veličinom polja od 20 mm. Vrlo je važno dobro odrediti prethodno spomenute dimenzije kako bi rezultat izvođenja algoritma bio validan. Isto tako je važno naglasiti da se kalibracijski uzorak mora nalaziti na ravnoj podlozi. Svaki nabori i neravnine mogu uzrokovati krive rezultate. Nakon odabira kalibracijskog uzorka, slijedi provedba postupka kalibracije. Radi lakšeg pregleda, dan je dijagram toka na slici ispod.



Slika 2.11. Dijagram toka provedbe kalibracije kamera

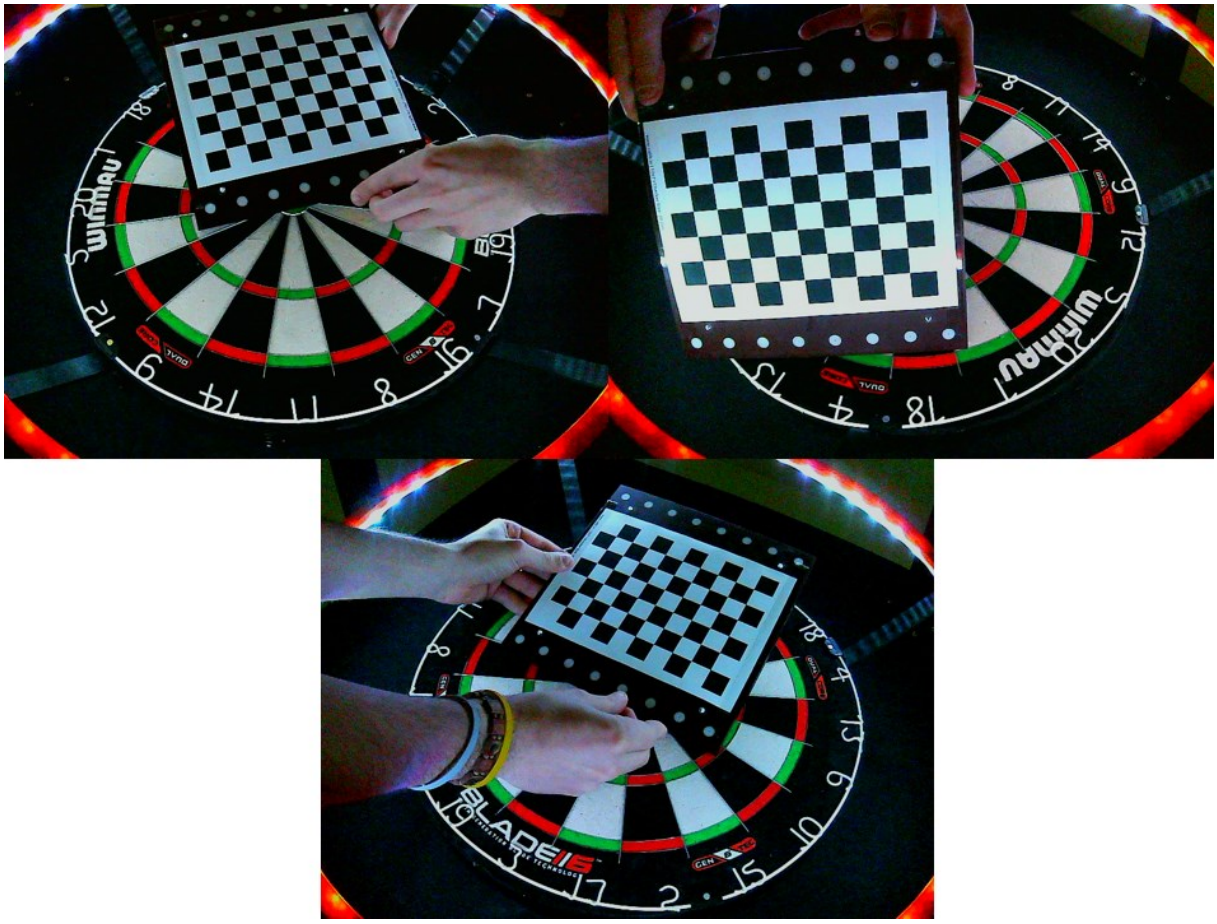
Prvi je korak prikupljanje slika na kojima će se izvršiti proces kalibracije kamera. Radi toga je napisan programski kod u *Python* programskom jeziku, a koji će se koristiti u cjelokupnom razvoju algoritma. Za potrebe pokretanja kamera koristi se prethodno spomenuta biblioteka *gstreamer*. S pomoću nje se mogu pokrenuti kamere sa specifičnim parametrima svjetline, oštrine, zasićenosti, kontrasta i sl. Pregled koda odgovornog za prvi korak vidljiv je u nastavku.

```
import cv2 as cv
width = 800 ; height = 600; framerate = "30/1"

cam1 = (
    "v4l2src device=/dev/video0 ! "
    "image/jpeg, width={}, height={}, framerate={} ! "
    "jpegdec ! videoconvert ! videobalance brightness=0.01 contrast=0.89 saturation=1.34 !
appsink").format(width, height, framerate)

cam2 = (
    "v4l2src device=/dev/video1 ! "
    "image/jpeg, width={}, height={}, framerate={} ! "
    "jpegdec ! videoconvert ! videobalance brightness=-0.01 contrast=0.87 saturation=1.35 !
appsink").format(width,height,framerate)

cam3 = (
    "v4l2src device=/dev/video2 ! "
    "image/jpeg, width={}, height={}, framerate={} ! "
    "jpegdec ! videoconvert ! videobalance brightness=0.01 contrast=0.87 saturation=1.35 !
appsink").format(width,height,framerate)

camlist = [cam1,cam2,cam3]
```

Na početku svakog koda nužno je inicijalizirati biblioteke koje se koriste u kodu, a u ovom slučaju je to *OpenCV*. Nakon toga su inicijalizirani podaci vezani za postavke rada kamere (rezolucija i FPS). U konačnici slijedi definiranje postavki svake kamere s pomoću biblioteke *gstreamer*. Bazira se na principu kreiranja *pipeline*-ova (eng. „cjevovoda“) uz pomoć raznih *plugin*-ova (eng. dodataka) koji su dostupni za kreiranje funkcionalne jedinice. Razni *plugin*-ovi pružaju razne funkcionalnosti, poput kodiranja i dekodiranja zvuka te videa. Nakon što započne video prijenos, potrebno je obraditi isti na način koji je korisniku prihvatljiv. U ovoj varijanti koristi se format *image/jpeg* sa specifičnim parametrima vezanim za definiranje slike. Razlog korištenja ovog *plugin*-a krije se u činjenici da je svaku sliku s nadolazećeg video prijenosa moguće obraditi upravo kao sliku uz pomoć kompresije u *jpeg* format. Na taj se način smanjuje veličina *frame*-ova dohvaćenog video prijenosa i samim time omogućuje reprodukcija većih rezolucija uz jedva primjetno kašnjenje videa na ekranu. Da je korišten neki drugi *plugin* za dohvaćanje video prijenosa, vrlo je izgledno kako bi prijenos dosta kasnio i shodno tome bio nepogodan za primjenu u ovom radu. *Videobalance plugin* koristi se u svrhu podešavanja parametara slike. Parametri vidljivi u kodu pokazali su se najboljima u ovome postavu. Tako su podešene sve tri kamere te stavljene u listu radi daljnjeg izvođenja algoritma.

```
for cam in range(len(camlist)):
    current_cam = camlist[cam]

    cap = cv.VideoCapture(current_cam,cv.CAP_GSTREAMER)
    num = 0

    while cap.isOpened():
        succes, img = cap.read()
        k = cv.waitKey(5)
        if k == 27 or num == 10:
            break
        elif k == ord('s'): # wait for 's' key to save and exit
            cv.imwrite('/home/fsb/Desktop/diplomski_opencv/intrinsic_calibration/images_dev
{}/img{}.png'.format(cam,num), img)
            print("image{} saved!".format(num))
            num += 1
        cv.imshow('Img',img)

    cap.release()
    cv.destroyAllWindows()
```

Drugi dio koda odnosi se na proces dohvaćanja slika. Koristi se *for* petlja prilikom čega se svaki prolazak kroz istu odnosi na pojedinu kameru. Pokreće se video prijenos te se slika cijelo vrijeme prikazuje na ekranu korisnika. *Frame*-ovi se konstantno dohvaćaju u *while* petlji, dok se s pomoću *k* varijable omogućuje spremanje pojedine slike s video prijenosa. Ako je pritisnuta tipka *ESC* ili je varijabla *num* dosegla vrijednost 10, *while* petlja se prekida i program završava s izvođenjem. Ako je pritisnuta tipka *s,* vrši se spremanje slike u pojedinu mapu. Dakle, nužno je pomicati kalibracijski uzorak u FOV-u kamere i spremati slike. Pri tome je važno poštivati određena pravila. Nije preporučljivo spremanje slika gdje se kalibracijski uzorak nalazi na istom mjestu. Važno je osigurati dobro osvjetljenje, a isto tako je poželjno slikati kalibracijski uzorak pod raznim kutevima i udaljenostima od leće kamere. Time se postiže raznolikost pozicija kalibracijskog uzorka što uzrokuje bolju točnost izračunatih parametara. Preporučljivo je uzeti 10-ak slika kako bi se osigurala raznolikost podataka. Primjer dohvaćenih slika dan je u nastavku.



Slika 2.12. Primjer generiranja slika s kalibracijskim uzorkom u FOV-u kamere

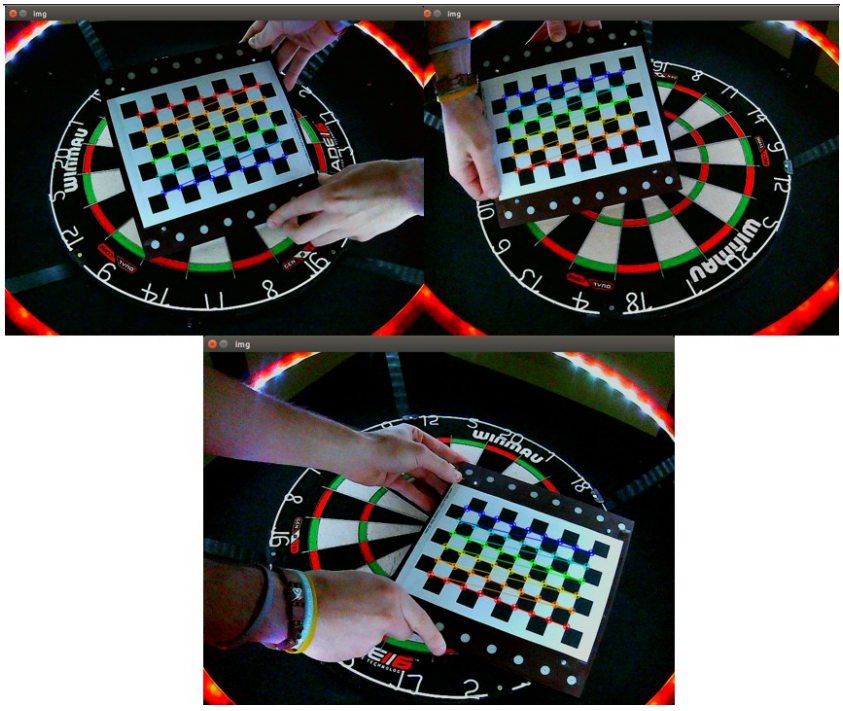
Na prethodnoj je slici vidljivo prisustvo distorzije koju je potrebno ukloniti. Kasnije će biti dana usporedba dviju slika, prije i nakon distorzije kako bi se dodatno naglasila važnost kalibracije kamera. Nakon što je generirano 10 slika za svaku kameru, slijedi drugi dio ranije prikazanog dijagrama toka.

```
import numpy as np
import cv2 as cv
import glob
import pickle
################ FIND CHESSBOARD CORNERS - OBJECT POINTS AND IMAGE POINTS
#############################
numcam = 3
chessboardSize = (9,6)
frameSize = (800,600)
# termination criteria
criteria = (cv.TERM_CRITERIA_EPS + cv.TERM_CRITERIA_MAX_ITER, 30, 0.001)
```

```
for cam in range(numcam):
    objpoints = [] # 3d points IRL
    imgpoints = [] # 2d points in image plane
    objp = []

    objp = np.zeros((chessboardSize[0] * chessboardSize[1], 3), np.float32) # type: ignore
    objp[:,:2] = np.mgrid[0:chessboardSize[0],0:chessboardSize[1]].T.reshape(-1,2)

    size_of_chessboard_squares_mm = 20
    objp = objp * size_of_chessboard_squares_mm
    images=glob.glob('/home/fsb/Desktop/diplomski_opencv/intrinsic_calibration/images_dev{}
/*.png'.format(cam))

    for image in images:
        img = cv.imread(image)
        gray = cv.cvtColor(img, cv.COLOR_BGR2GRAY)

        ret, corners = cv.findChessboardCorners(gray, chessboardSize, None)
        if ret == True:
            objpoints.append(objp)
            corners2 = cv.cornerSubPix(gray, corners, (11,11), (-1,-1), criteria)
            imgpoints.append(corners)
            # display corners
            cv.drawChessboardCorners(img, chessboardSize, corners2, ret)
            cv.imshow('img', img)
            cv.waitKey(500)
    cv.destroyAllWindows()
```

Radi lakšeg razumijevanja, cjelokupan je kod podijeljen na dva dijela. Prethodno prikazani prvi dio odnosi se na pozivanje svih biblioteka potrebnih za izvođenje programa, definiranje parametara veličine kalibracijskog uzorka, veličine slika i sl. te generiranje ulaznih podataka za funkciju *calibrateCamera*. Prije objašnjenja koda, vrijedi istaknuti generalnu ideju ovog cjelokupnog postupka. Cilj je kalibrirati kameru s obzirom na kalibracijski uzorak poznatih dimenzija, u ovom slučaju šahovnice dimenzija 10x7. Tako imamo jasno definirane pozicije svakog kuta šahovnice i ulazne podatke za daljnju obradu. Druge podatke dobivamo iz slike kamere, gdje se kutevi softverski pronalaze te spremaju njihove pozicije. Nakon dohvaćanja oba seta podataka, moguće je iste iskoristiti kao ulazne podatke u funkciju *calibrateCamera* čijim se izvođenjem dobivaju željene matrice. Što se tiče prvog dijela koda, varijabla *numcam* označava broj kamera. Ona se koristi za definiranje ponavljanja *for* petlje. Nadalje je potrebno definirati veličinu šahovnice kalibracijskog uzorka. Ta veličina predstavlja veličinu unutrašnjosti šahovnice, odnosno isključen je vanjski rub šahovnice. Prema tome, veličina šahovnice je 9x6. Naposljetku je potrebno definirati veličinu *frame*-a koja odgovara onoj veličini slika koje su zabilježene u prvom dijelu procesa.

Varijabla *criteria* se koristi u procesu pronalaženja kuteva šahovnice, a predstavlja kriterij s obzirom na koji proces staje u trenutku kad je isti postignut. Zatim slijedi inicijalizacija *objp* liste. Ta lista predstavlja poziciju (*x,y,z*) svakog kuta šahovnice. Pošto se kalibracijski uzorak nalazi na ravnoj podlozi, podrazumijeva se da je *z* koordinata za sve kuteve šahovnice jednaka 0. Prema tome, potrebno je odrediti samo *x* i *y* koordinate. One se određuju tako da prethodno generiranu dimenziju množimo s dimenzijom jednog polja šahovnice (u ovom slučaju 20 mm). Tako je definirana lista 3D točaka poznatih pozicija kuteva šahovnice. Slijedi učitavanje pojedine slike te pretvorba iste u *grayscale* format koji je potreban za korištenje funkcije *findChessboardCorners*. Iz poziva funkcije vidljivo je kako su ulazni podaci prethodno spomenuta *grayscale* slika te dimenzija kalibracijskog uzorka, dok su izlazni podaci *retval* (*bool* vrijednost koja je vrijednosti *true* ako su pronađeni kutovi šahovnice) i matrica pronađenih pozicija kuteva šahovnice koje su izražene u pikselima. Važno je napomenuti kako su pronađene pozicije kutova šahovnice aproksimirane te ih je potrebno dodatno preciznije odrediti. To se čini korištenjem dodatne funkcije *cornerSubPix*. Za dodatne informacije čitatelja se upućuje na službene stranice *OpenCV* biblioteke gdje se nalazi detaljniji opis funkcije. Prilikom svakog pronalaska kutova u svakoj slici, vrši se punjenje prethodno stvorenih lista kako bi se zabilježile informacije pozicija u slici. U konačnici se rezultati pronađenih kutova prikazuju pozivanjem *drawChessboardCorners* kako bi se dobiveni rezultat mogao validirati.



Slika 2.13. Pronalazak kutova kalibracijskog uzorka

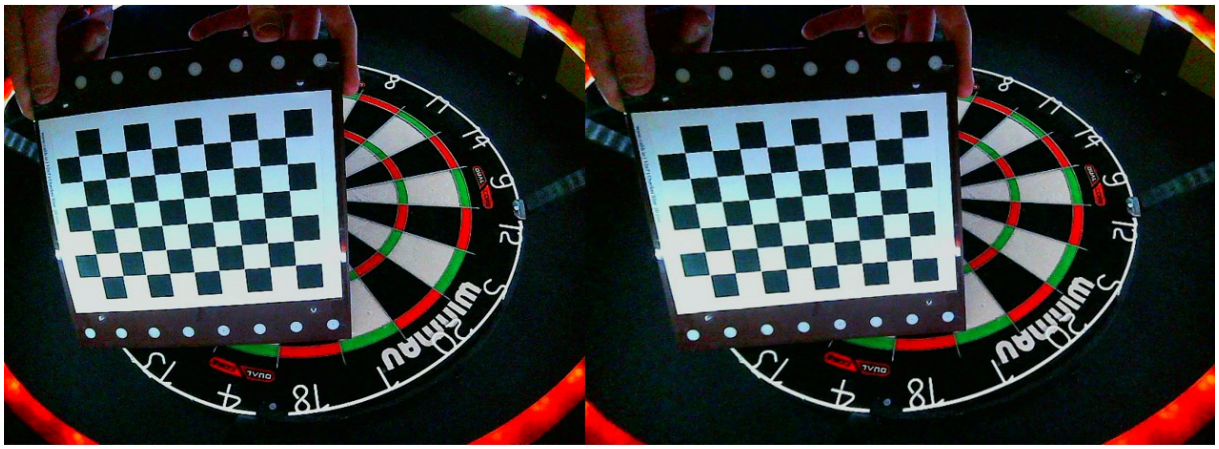
Vrlo je važno pratiti dobivene vidljive rezultate kako bi bili sigurni da je algoritam za svaku sliku pojedine kamere odabrao dobru orijentaciju te dobro ishodište. Ako se raspored pronađenih kutova određene kamere međusobno razlikuje, vrlo je vjerojatno da se dogodila pogreška koju je potrebno popraviti ponovnim bilježenjem slika. Isto tako, vjerojatno će pogreška biti reproducirana u provjeri rezultata koja će biti pojašnjena kasnije.

```
    ############## CALIBRATION ####################################################

    ret, cameraMatrix, dist, rvecs, tvecs = cv.calibrateCamera(objpoints, imgpoints,
frameSize, None, None)

    pickle.dump((cameraMatrix, dist, rvecs, tvecs), open(
"/home/fsb/Desktop/diplomski_opencv/intrinsic_calibration/calibration_cam{}.pkl".format(cam
), "wb"))
    pickle.dump(cameraMatrix, open(
"/home/fsb/Desktop/diplomski_opencv/intrinsic_calibration/cameraMatrix_cam{}.pkl".format(ca
m), "wb"))
    pickle.dump(dist, open(
"/home/fsb/Desktop/diplomski_opencv/intrinsic_calibration/dist_cam{}.pkl".format(cam),
"wb"))

    ############## UNDISTORTION ###################################################
    img=cv.imread("/home/fsb/Desktop/diplomski_opencv/intrinsic_calibration/images_dev{}/im
g5.png".format(cam))
    h,w = img.shape[:2]
    newCameraMatrix, roi = cv.getOptimalNewCameraMatrix(cameraMatrix, dist, (w,h), 1,
(w,h))
    dst = cv.undistort(img, cameraMatrix, dist, None, newCameraMatrix)
    # crop blank parts
    x, y, w, h = roi
    dst = dst[y:y+h, x:x+w]
    cv.imwrite('/home/fsb/Desktop/diplomski_opencv/intrinsic_calibration/caliResult_cam{}.p
ng'.format(cam), dst)

    # reprojection error
    mean_error = 0
    for i in range(len(objpoints)):
        imgpoints2, _ = cv.projectPoints(objpoints[i], rvecs[i], tvecs[i], cameraMatrix,
dist)
        error = cv.norm(imgpoints[i], imgpoints2, cv.NORM_L2)/len(imgpoints2)
        mean_error += error

    print( "total error_cam{}: {}".format(cam,mean_error/len(objpoints)) )
```

Nakon dohvaćanja stvarnih poznatih pozicija 3D točaka kutova šahovnice te pozicija 2D točaka kutova šahovnice u slici kamere, slijedi pozivanje funkcije *calibrateCamera*. Izlazni podaci pozivanja ove funkcije su prethodno spomenuta *bool retval* vrijednost, intrinzična matrica kamere, vektor distorzijskih koeficijenata, vektor rotacijskih te translacijskih vektora za svaki položaj kalibracijskog uzorka u pojedinoj slici. Od svih dobivenih podataka, najviše vrijedi istaknuti intrinzičnu matricu kamere te distorzijske koeficijente koji rješavaju problem perspektivne projekcije i ispravljaju sliku tako da daje mjerodavne informacije. Uz korištenje *pickle* biblioteke, prethodno se dobiveni podaci spremaju kako bi se kasnije mogli koristiti u razne svrhe. Konačno, slijedi provjera dobivenih matrica i vektora. Za svaku se kameru uzima jedna uslikana slika kalibracijskog uzorka u njenom FOV-u te se na istu primjenjuje uklanjanje distorzije. Prvo se pronalazi širina i visina slike, a zatim se s pomoću funkcije *getOptimalNewCameraMatrix* pronalazi nova optimirana matrica. Izračun se vrši s obzirom na prethodno dobivenu intrinzičnu matricu, distorzijske koeficijente, širinu i visinu slike te slobodni parametar *alpha* koji može poprimiti vrijednosti između 0 i 1:

- *alpha* = 0 – izrezuju se svi dijelovi slike koji bi sadržavali crne piksele uslijed uklanjanja distorzije, posebno vidljivo na samim rubovima slike
- *alpha* = 1 – zadržavaju se svi originalni pikseli slike koji uključuju i crne piksele uslijed uklanjanja distorzije – korisnik ručno uklanja dijelove slike koji su crni

U ovom je slučaju odabrana vrijednost parametra 1 kako bi se sačuvale sve informacije s originalne slike, dok se crni pikseli ručno uklanjaju. Nakon toga vrši se spremanje slika s uklonjenom distorzijom, a primjer usporedbe rezultata jedne od kamera dan je u nastavku.



a) b)

Slika 2.14. Usporedba slika: a) prije uklanjanja distorzije, b) nakon uklanjanja distorzije - kamera 3

U konačnici slijedi provjera dobivenih rezultata. Za svaki pronađeni kalibracijski uzorak uzima se lista stvarnih 3D točaka te dobiveni vektori i matrice s pomoću funkcije *calibrateCamera.* Navedeni parametri koriste se kao ulazni podaci za funkciju *projectPoints*. Tako se poznate pozicije 3D točaka kutova šahovnice projiciraju u ravninu slike i time se dobiva lista pozicija 2D točaka kutova šahovnice. Nakon toga se koristi funkcija *norm* u svrhu izračuna euklidske udaljenosti između 2 seta točaka – prethodno pronađenih 2D točaka s pomoću funkcije *findChessboardCorners* i projiciranih 2D točaka pronađenih s pomoću funkcije *projectPoints*. Pogreška se za svaki set podataka zbraja te u konačnici dijeli s brojem uslikanih uzoraka za pojedinu kameru. Čim je dobiveni rezultat bliži 0, to je bolje izvršena kalibracija.

```
0.153966545648
total error_cam0: 0.0204177631778
0.167335602117
total error_cam1: 0.0223445457774
0.147004457488
total error_cam2: 0.0197207611417
```

Slika 2.15. Provjera točnosti izvršene kalibracije kamera

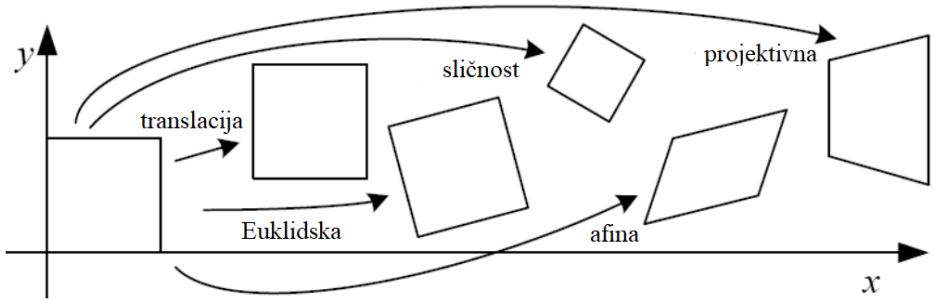
3. HOMOGRAFSKO ISPRAVLJANJE SLIKE

Nakon uspješne provedbe kalibracije kamera, slijedi postupak ispravljanja perspektive kamera stvaranjem homografske matrice. Homografija je vrlo bitan koncept u polju računalnog vida koja se koristi za opisivanje geometrijskih transformacija između dviju slika ili slike i stvarnog svijeta. Tako je matematički moguće povezati skup točaka jedne slike s točkama u drugoj slici uz očuvanje geometrijskih relacija među njima. Homografija nalazi primjenu u raznim spektrima računalnog vida. Koristi se u vidu spajanja dviju ili više preklapajućih slika čime se dobiva širi pogled, odnosno panorama. Svoju primjenu nalazi u registraciji slika gdje se poravnavaju slike iste scene uslikane s različitih mjesta u različito vrijeme. Također se koristi u polju proširene stvarnosti (AR), gdje se virtualni objekti projiciraju u okolinu koja nas okružuje.

Slika 3.1. Primjer korištenja homografije u proširenoj stvarnosti [13]

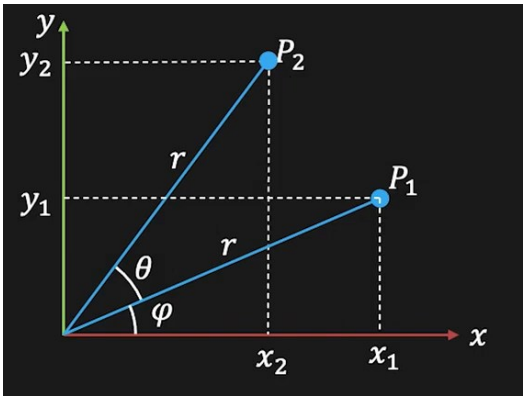
3.1. OSNOVNI KONCEPT

Kako bi se stvorila jasnija slika o postupku homografije, valja krenuti s osnovama transformacije slika. Postoji više geometrijskih transformacija slika koje su prikazane na slici u nastavku.



Slika 3.2. Vrste geometrijske transformacije slika [14]

Kao što je vidljivo na prethodnoj slici, moguće je izvršiti razne geometrijske transformacije nad određenom slikom. Pritom je važno obratiti pozornost na dobar postav transformacijske matrice kako bismo dobili željenu transformaciju. Moguće je koristiti transformacijske matrice dimenzija 2x2. Korištenjem takvih matrica moguće je ostvariti jednostavne operacije poput skaliranja, rotacija i skošenja slike. Kako bi se bolje razumjela kasnije korištena homografija, vrijedi vidjeti princip rada jedne od prethodno navedenih transformacija.



Slika 3.3. Pronalazak transformacijske matrice za 2D rotaciju [15]

Na prethodno je prikazanoj slici cilj pronaći transformacijsku matricu kako bi iz točke P_1 bilo moguće doći u ciljnu točku P_2. Pozicije prethodno spomenutih točaka mogu se opisati u polarnim koordinatama, preko udaljenosti od ishodišta *r* te kuteva φ i θ:

$$x_1 = r\cos(\varphi),$$
$$y_1 = r\sin(\varphi),$$
$$x_2 = r\cos(\varphi + \theta),$$
$$y_2 = r\sin(\varphi + \theta). \quad (12)$$

Primijenom adicijskih teorema, dio prethodnih jednadžbi moguće je dodatno raspisati na sljedeći način:

$$x_2 = rcos(\varphi)\cos(\theta) - rsin(\varphi)\sin(\theta),$$

$$y_2 = rcos(\varphi)\sin(\theta) + rsin(\varphi)\cos(\theta),$$

odnosno uz zamjenu pomoću dijela jednadžbi (12):

$$x_2 = x_1 cos(\theta) - y_1 sin(\theta),$$

$$y_2 = x_1 sin(\theta) + y_1 cos(\theta). \quad (13)$$

Iz prethodnog je izraza vidljivo kako se isti može zapisati u sljedećoj matričnoj formi:

$$\begin{bmatrix} x_2 \\ y_2 \end{bmatrix} = \begin{bmatrix} cos(\theta) & -sin(\theta) \\ sin(\theta) & cos(\theta) \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \end{bmatrix}, \quad (14)$$

pri čemu 2x2 matrica predstavlja transformacijsku matricu rotacije slike. Na isti je način moguće definirati transformacijske matrice za sve geometrijske promjene slike vidljive na slici 3.2. Vrijedi napomenuti kako 2x2 transformacijske matrice imaju sljedeća svojstva[15]:

- ishodište se transformira u ishodište
- linije se transformiraju u linije
- paralelne linije ostaju paralelne
- algebarski zatvoreno pod kompozicijom

Na isti je način moguće koristiti 3x3 transformacijske matrice, no uz dodatnu primjenu prethodno spomenutih homogenih koordinata. Bez njihove upotrebe, stvaranje transformacijskih matrica dimenzija 3x3 bi bilo nemoguće. To je najbolje vidljivo na primjeru jednostavne translacije slike za određeni pomak u smjerovima osi *x* i *y*.[16] Tako je moguće generirati 3x3 transformacijske matrice za translaciju, rotaciju, skošenje, skaliranje i sl. Kad bi korisnik htio izvršiti sve prethodno nabrojane geometrijske promjene, mogao bi to učiniti postepeno. No, jednostavnije je izvršiti kompoziciju svih prethodno nabrojanih geometrijskih transformacija čime se dobiva jedna 3x3 matrica koja predstavlja prethodno spomenutu afinu transformaciju. Svaka takva transformacija ima sljedeći oblik[16]:

$$\begin{bmatrix} x_2 \\ y_2 \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = \begin{bmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} \tilde{x}_1 \\ \tilde{y}_1 \\ \tilde{z}_1 \end{bmatrix}. \quad (15)$$

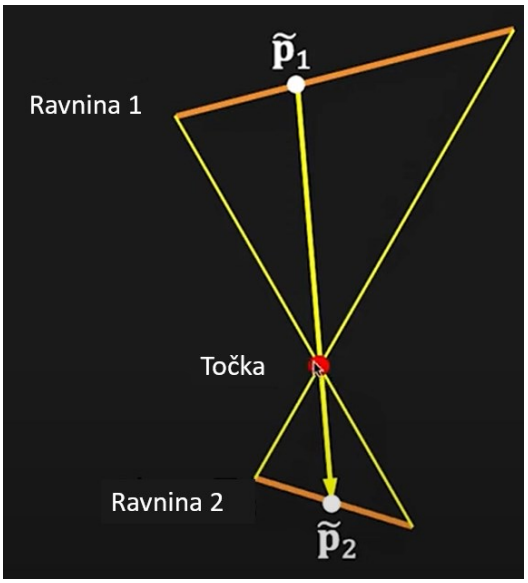
Iz prethodne je matrične forme vidljivo kako 3x3 transformacijska matrica sadrži uvijek isti zadnji redak (0-0-1) što ukazuje na upotrebu homogenih koordinata. Također je vidljivo kako ta matrica sadrži 6 slobodnih parametara koje je moguće računati. Što se tiče afine transformacije, ona ima sljedeće karakteristike[16]:

- ishodište se ne mora nužno transformirati u to isto ishodište
- linije se transformiraju u linije
- paralelne linije ostaju paralelne
- algebarski zatvoreno pod kompozicijom

Ako se zadnji redak afine transformacijske matrice zapiše kao skup dodatnih 3 parametra, dobiva se sljedeći zapis:

$$\begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} \tilde{x}_1 \\ \tilde{y}_1 \\ \tilde{z}_1 \end{bmatrix}. \tag{16}$$

U ovom slučaju 3x3 matrica predstavlja projektivnu transformacijsku matricu, također poznatu pod nazivom matrica homografije. Ta matrica transformira jednu ravninu na drugu ravninu kroz jednu točku.



Slika 3.4. Prikaz djelovanja projektivne transformacije [16]

U matričnom zapisu bi transformacija izgledala ovako:

$$\tilde{p}_2 = \mathbf{H}\tilde{p}_1. \tag{17}$$

Valja naglasiti bitno pravilo kod generiranja matrice homografije. Množenjem svih elemenata matrice skalarom ne mijenja ništa, odnosno i dalje se dobiva ista transformacija:

$$\begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} \tilde{x}_1 \\ \tilde{y}_1 \\ \tilde{z}_1 \end{bmatrix} \equiv \begin{bmatrix} \tilde{x}_2 \\ \tilde{y}_2 \\ \tilde{z}_2 \end{bmatrix} \equiv k \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} \begin{bmatrix} \tilde{x}_1 \\ \tilde{y}_1 \\ \tilde{z}_1 \end{bmatrix}. \tag{18}$$

Dakle, ako fiksiramo taj skalar na sljedeći način:

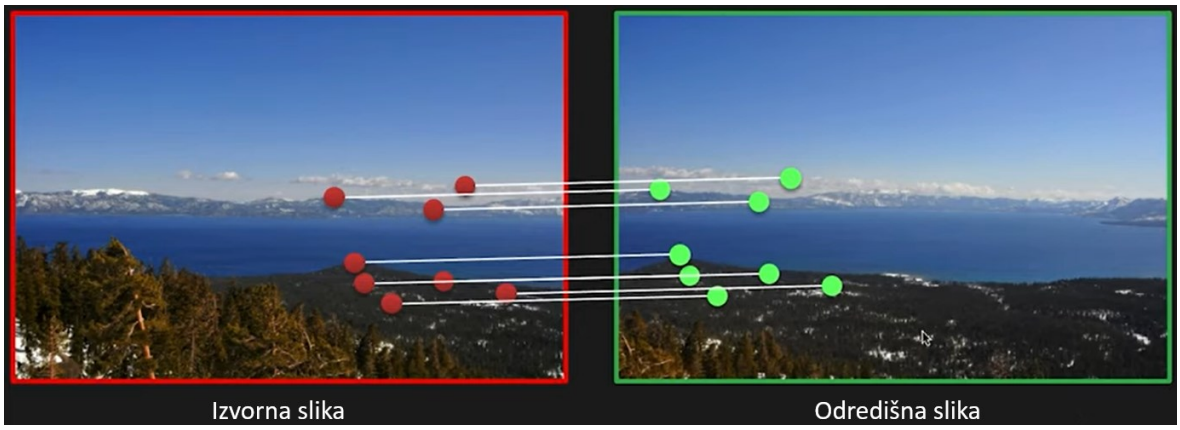
$$k = \sqrt{\sum \left(h_{ij}\right)^2} = 1, \tag{19}$$

tada se može reći kako matrica homografije ima 8 slobodnih parametara. S obzirom na to da je matrica homografije drugačije definirana u odnosu na afinu transformaciju, vrijedi istaknuti njena svojstva:

- ishodište se ne mora nužno transformirati u to isto ishodište
- linije se transformiraju u linije
- paralelne linije ne moraju nužno ostati paralelne
- algebarski zatvoreno pod kompozicijom

Također je važno istaknuti slučajeve u kojima se homografija između dviju slika može koristiti. Prvi je slučaj taj da se sve točke koje se koriste u homografiji moraju nalaziti na istoj ravnini. Drugi slučaj u kojem se homografija može koristiti je da točke koje ulaze u homografski izračun moraju biti dosta udaljene, tako da ne moraju nužno ležati u istoj ravnini. Treći slučaj predstavlja onaj u kojem su slike uzete s iste točke gledišta pri čemu se samo mijenja rotacija kamere.[17]

3.1.1. Teoretski izračun matrice homografije



Slika 3.5. Teoretski izračun matrice homografije s pomoću točaka dviju slika [17]

Na prethodnoj su slici prikazane izvorna i odredišna slika te točke koje se međusobno podudaraju na njima. Cilj je iskoristiti te točke kako bi se odredila matrica homografije koja se kasnije koristi za ispravljanje perspektive cjelokupne slike. Na temelju zapisa (16), problem s prethodne slike moguće je prikazati na sljedeći način:

$$\begin{bmatrix} x_o \\ y_o \\ 1 \end{bmatrix} \equiv \begin{bmatrix} \tilde{x}_o \\ \tilde{y}_o \\ \tilde{z}_o \end{bmatrix} = \begin{bmatrix} h_{11} & h_{12} & h_{13} \\ h_{21} & h_{22} & h_{23} \\ h_{31} & h_{32} & h_{33} \end{bmatrix} = \begin{bmatrix} x_i \\ y_i \\ 1 \end{bmatrix}. \tag{20}$$

S obzirom na to da matrica homografije ima 8 stupnjeva slobode, odnosno 8 neodređenih parametara, to znači da je potrebno naći minimalno 4 para točaka. Čim je veći broj ispravnih parova, to je matrica homografije preciznija. Svaki par čini jedna točka u izvornoj slici te njena pripadajuća pozicija u odredišnoj slici. Kad se matrični zapis (20) raspiše i dodatno uredi tako da se izrazi brojnik svakog izraza, dobivaju se sljedeće jednadžbe:

$$\begin{aligned} x_o^{(i)}\left(h_{31}x_s^{(i)} + h_{32}y_s^{(i)} + h_{33}\right) &= h_{11}x_s^{(i)} + h_{12}y_s^{(i)} + h_{13}, \\ y_o^{(i)}\left(h_{31}x_s^{(i)} + h_{32}y_s^{(i)} + h_{33}\right) &= h_{21}x_s^{(i)} + h_{22}y_s^{(i)} + h_{23}, \end{aligned} \tag{21}$$

pri čemu je:

- i – oznaka pojedinog para točaka.

Prebacivanjem desne strane izraza na lijevu te uklanjanjem postojećih zagrada, može se stvoriti sljedeći matrični zapis:

$$\begin{bmatrix} x_i^{(i)} & y_i^{(i)} & 1 & 0 & 0 & 0 & -x_o^{(i)}x_i^{(i)} & -x_o^{(i)}y_i^{(i)} & -x_o^{(i)} \\ 0 & 0 & 0 & x_i^{(i)} & y_i^{(i)} & 1 & -y_o^{(i)}x_i^{(i)} & -y_o^{(i)}y_i^{(i)} & -y_o^{(i)} \end{bmatrix} \begin{bmatrix} h_{11} \\ h_{12} \\ h_{13} \\ h_{21} \\ h_{22} \\ h_{23} \\ h_{31} \\ h_{32} \\ h_{33} \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix}. \tag{22}$$

Prethodni je zapis prikaz matričnog zapisa za jedan par točaka. Dakle, matrice se dodatno proširuju s brojem porasta parova točaka. Prema tome, ako se prethodni zapis zapiše na sljedeći način:

$$\mathbf{Ah} = 0, \tag{23}$$

tada se može konstituirati da je:

- **A** – matrica parova poznatih točaka dimenzija 2n x 9 - pri čemu je n broj parova točaka,
- **h** – vektor stupac parametara matrice homografije dimenzija 9 x 1,
- **0** – vektor stupac 0 dimenzija 2n x 1 - pri čemu je n broj parova točaka,
- dodatan uvjet: $\|\mathbf{h}\|^2 = 1$, kako ne bi postojalo trivijalno rješenje **h** = 0.

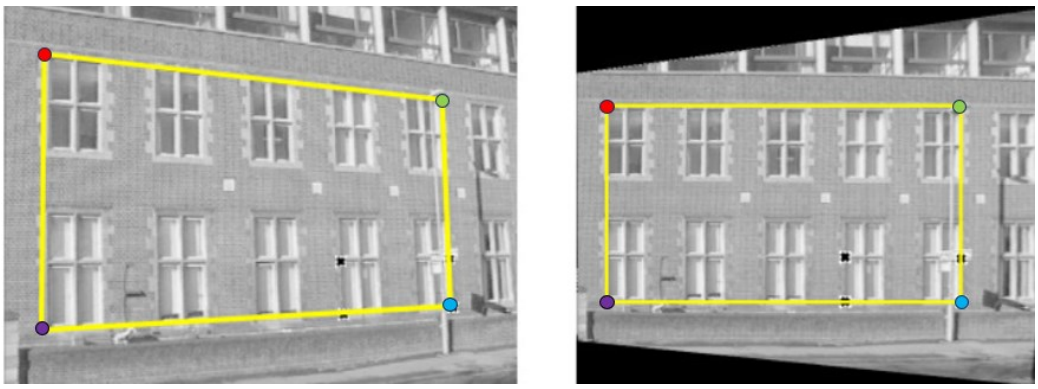
U jednadžbi (23) je lako zaključiti kako je jedina nepoznanica vektor stupac **h** koji je potrebno izračunati. Ovom se problemu pristupa metodom najmanjih kvadrata. Korištenje te metode je najviše korisno u slučajevima gdje ima više prisutnih jednadžbi nego nepoznanica. Prema tome, ovaj se problem može prikazati na sljedeći način:

$$\min\|\mathbf{Ah}\|^2, \tag{24}$$

uz prethodno spomenuti uvjet vezan za uklanjanje trivijalnog rješenja. Korištenjem metode najmanjih kvadrata može se doći do svojstvenog vektora **h** koji predstavlja rješenje problema, odnosno matricu homografije koja se traži. Matematički raspis ovog problema moguće je pronaći u literaturi [17].

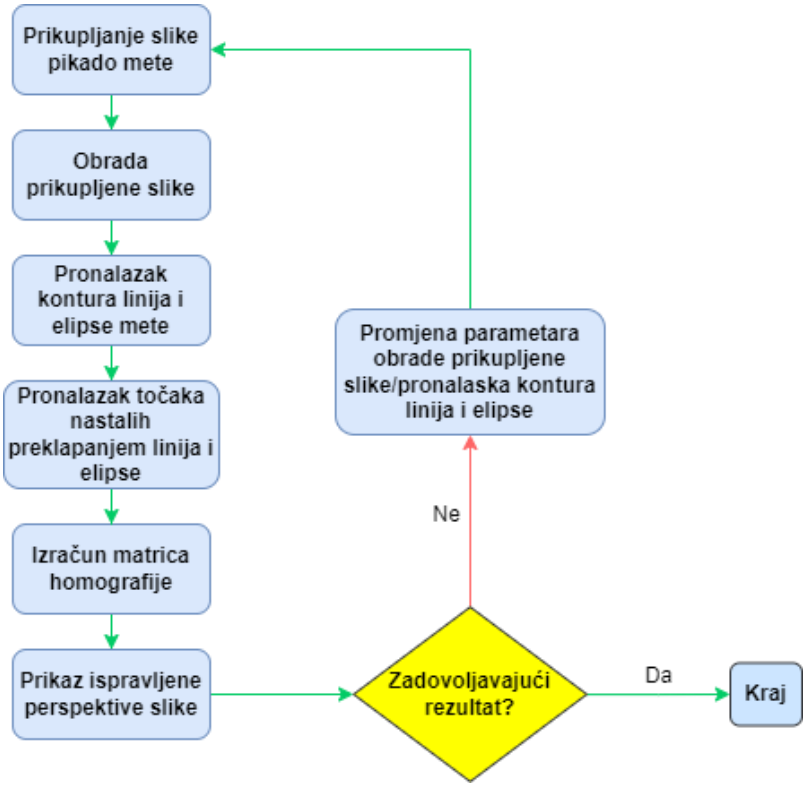
3.2. ODREĐIVANJE MATRICA HOMOGRAFIJE

Nakon detaljnijeg objašnjenja koncepta homografije, može se pristupiti praktičnoj primijeni prethodno prikazanog teoretskog dijela. Osnovna je ideja bila korištenje AruCo markera kako bi se pronalazile točke na izvornoj slici koje bi se kasnije koristile za izračun matrice homografije. Međutim, radi uštede prostora i nepotrebnosti korištenja markera, pribjeglo se rješenju koje ne uključuje dodatne elemente. Naime, primjenom metoda obrade slike s pomoću biblioteke *OpenCV* moguće je konstantno odabirati iste kvalitetne točke koje se mogu koristiti za izračun matrice homografije. Ova će metoda biti kasnije detaljnije objašnjena. Prije svega, važno je razumjeti koncept empirijske primjene ovog koncepta. U ovom je radu cilj pronaći prethodno spomenute 4 točke na izvornoj slici, dok se 4 destinacijske točke biraju samostalno. Razlog tome je to što je ovdje cilj ispraviti perspektivu slike na način vidljiv na sljedećoj slici. Važno je primijetiti kako se svaka obojena točka s izvorne slike preslikava u obojenu točku na destinacijskog slici.



Slika 3.6. Primjer empirijske primijene matrice homografije [13]

Dakle, odredišne točke korisnik samostalno bira kako bi se ispravljena slika pozicionirala na ekranu na način koji odgovara korisniku. Nakon uspješnog ispravljanja perspektive slike, potrebno je zabilježiti matricu homografije koja se kasnije koristi za ispravljanje pozicije točke vrha strelice, odnosno kasnije traženje postignutih bodova. Kako bi kod za dohvaćanje matrica homografije bio jasniji, ponovno se koristi predočenje s pomoću dijagrama toka.



Slika 3.7. Dijagram toka izračuna matrica homografije

```
import cv2 as cv
import numpy as np
import math, pickle, subprocess

anglenum = 0
width1 = 800; height1 = 600; framerate = "30/1"
subprocess.call(["v4l2-ctl", "-d","/dev/video0", "--set-ctrl=exposure_auto=1"])
subprocess.call(["v4l2-ctl", "-d","/dev/video0", "--set-ctrl=exposure_absolute=28"]) # --
mijenjati ekspoziciju ovisno o vanjskom svijetlu
subprocess.call(["v4l2-ctl", "-d", "/dev/video0", "--set-ctrl=gamma=112"])
subprocess.call(["v4l2-ctl", "-d", "/dev/video0", "--set-ctrl=sharpness=4"])

subprocess.call(["v4l2-ctl", "-d","/dev/video1", "--set-ctrl=exposure_auto=1"])
subprocess.call(["v4l2-ctl", "-d","/dev/video1", "--set-ctrl=exposure_absolute=28"]) #
subprocess.call(["v4l2-ctl", "-d", "/dev/video1", "--set-ctrl=gamma=111"])
subprocess.call(["v4l2-ctl", "-d", "/dev/video1", "--set-ctrl=sharpness=4"])

subprocess.call(["v4l2-ctl", "-d","/dev/video2", "--set-ctrl=exposure_auto=1"])
subprocess.call(["v4l2-ctl", "-d","/dev/video2", "--set-ctrl=exposure_absolute=28"])
subprocess.call(["v4l2-ctl", "-d", "/dev/video2", "--set-ctrl=gamma=112"])
subprocess.call(["v4l2-ctl", "-d", "/dev/video2", "--set-ctrl=sharpness=4"])

cam1 = (
    "v4l2src device=/dev/video0 ! "
    "image/jpeg, width={}, height={}, framerate={} ! "
    "jpegdec ! videoconvert ! videobalance brightness=0.01 contrast=0.89 saturation=1.34 !
appsink").format(width1, height1, framerate)

cam2 = (
    "v4l2src device=/dev/video1 ! "
    "image/jpeg, width={}, height={}, framerate={} ! "
    "jpegdec ! videoconvert ! videobalance brightness=-0.01 contrast=0.87 saturation=1.35 !
appsink").format(width1,height1,framerate)

cam3 = (
    "v4l2src device=/dev/video2 ! "
    "image/jpeg, width={}, height={}, framerate={} ! "
    "jpegdec ! videoconvert ! videobalance brightness=0.01 contrast=0.87 saturation=1.35 !
appsink").format(width1,height1,framerate)

camlist = [cam1,cam2,cam3]
for camera in range(len(camlist)):
    curcam = camlist[camera]
    cap = cv.VideoCapture(curcam,cv.CAP_GSTREAMER)
    while True:
        ret, frame = cap.read()
```

```
        if ret:
            cv.imshow("frame", frame)
            k = cv.waitKey(10)
            if k == ord('s'):
                dartboard = frame
                break
            else:
                pass
    cap.release()
    cv.destroyAllWindows()
```

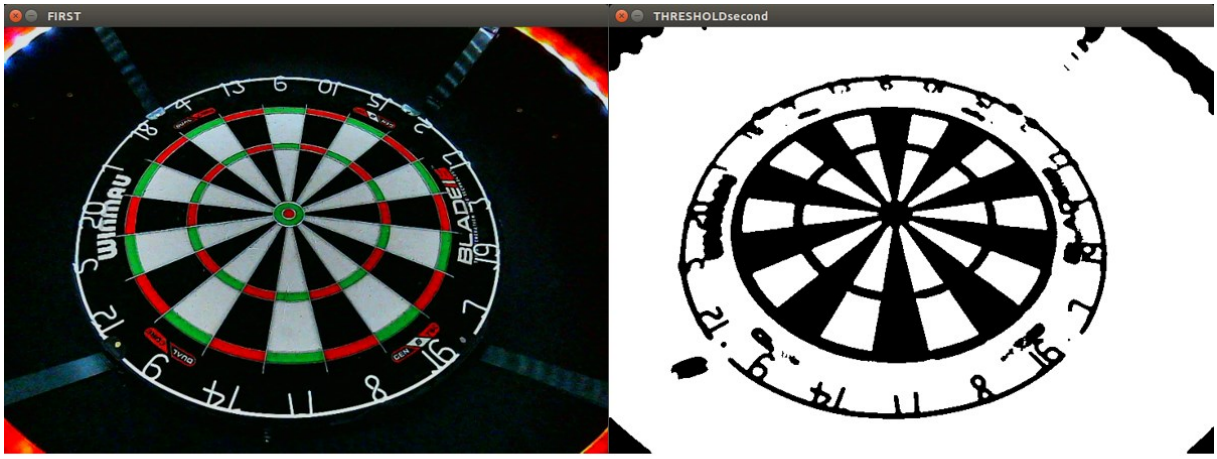
Kako bi se započelo s izračunom matrica homografije za pojedinu kameru, potrebno je inicijalizirati biblioteke koje se koriste u ovom dijelu koda. Biblioteke koje se dosad nisu koristile su *numpy*, *math, pickle* i *subprocess. Numpy* biblioteka je vrlo korisna za numeričke izračune, a prvenstveno se ističe pri matričnim operacijama, stvaranjem lista i sl. *Math* biblioteka je korisna u matematičkom području pošto sadrži brojne funkcije i matematičke konstante koje se često koriste u rješavanju problema. *Pickle* biblioteka omogućuje operacije spremanja *python* lista/matrica u datoteke na određeno lokalno mjesto u računalu te kasnije učitavanje tih datoteka u neki drugi *python* program. *Subprocess* biblioteka se koristi za izvršavanje vanjskih naredbi (*terminal*) unutar *python* skripte. Ponovno je potrebno dobro postaviti parametre slike, a zatim se izvršavaju vanjske naredbe kako bi se ručno mogla postaviti ekspozicija, oštrina i gamma parametar kamera. Od prethodnih je parametara najvažnija mogućnost manualnog postavljanja ekspozicije. Ona predstavlja povećanje/smanjenje otvora leće što u konačnici rezultira količinom svjetlosti koja dolazi do senzora kamere. Ekspoziciju je potrebno postaviti u odnosu na količinu svjetla koja je dostupna u okruženju kamere. Ako je svjetla dovoljno, ekspoziciju je potrebno postaviti na niže vrijednosti, dok ju je u obrnutom slučaju potrebno povećati. U konačnici se ponovno postavljaju postavke kamera korištenjem *gstreamer* biblioteke te se iste stavljaju u listu koju će se u nastavku koristiti za iteracije unutar programa. Nakon toga slijedi provedba postupka za svaku kameru. Prvo je potrebno koristiti *OpenCV* biblioteku za dohvaćanje slike kamere koja se sprema u varijablu *dartboard*.

```
    # intrinsic matrix and distortion parameters
    file=open("/home/fsb/Desktop/diplomski_opencv/intrinsic_calibration/calibration_cam{}.p
kl".format(camera), "rb")
    cammatrix = pickle.load(file)
    file.close()
    intrmatrix = cammatrix[0]; distcoefs = cammatrix[1]
```

```
    ############## UNDISTORTION ####################################################
    h,w = dartboard.shape[:2]
    newintrmatrix, roi = cv.getOptimalNewCameraMatrix(intrmatrix, distcoefs, (w,h), 1,
(w,h))

    # Undistort
    correctedimage = cv.undistort(dartboard, intrmatrix, distcoefs, None, newintrmatrix)

    # crop the image
    x, y, w, h = roi
    correctedimage = correctedimage[y:y+h, x:x+w]

    dartboard = correctedimage
    dartboard2 = correctedimage.copy()
    dartboard3 = correctedimage.copy()
    blackcont = np.zeros_like(correctedimage)
    blackline = np.zeros_like(correctedimage)

    dartboard = cv.cvtColor(dartboard, cv.COLOR_BGR2HSV)
    kernel = np.ones((7,7), np.float32) / 49 # type:ignore
    dartboard = cv.filter2D(dartboard,-1,kernel)
    h,s,v = cv.split(dartboard)
    _, dartboard = cv.threshold(v, 255/3, 255, cv.THRESH_BINARY_INV|cv.THRESH_OTSU)
```

Idući je korak provedba postupka obrade prikupljene slike. Prvi je korak učitavanje intrinzične matrice i distorzijskih koeficijenata za pojedinu kameru. To je vrlo važan korak koji osigurava uklanjanje distorzije te daljnju ispravnu provedbu obrade slike. Postupak uklanjanja distorzije te pripadajući rezultati identični su onima u procesu provjere izvršene kalibracije. Nakon toga se vrši inicijalizacija varijabli koje će kasnije biti korištene u procesu. Od toga valja istaknuti varijablu *blackcont* koja se koristi za iscrtavanje konture elipse mete, te *blackline* koja se koristi za iscrtavanje filtriranih linija mete. Inicijaliziraju se naredbom *np.zeros_like* kako bi se stvorila crna slika istih dimenzija kao i slika mete nakon uklanjanja distorzije. Pošto su na pikado meti prisutne razne dominantne boje (crna, bijela, crvena, zelena), slika pikado mete pretvara se u drugi *colorspace* korištenjem naredbe *cv.cvtColor*. Taj je postupak gotovo neizostavan u svakoj obradi slike pošto je njime moguće izraziti određene aspekte slike, a isto tako je često i preduvjet za korištenje nekih drugih funkcija biblioteke *OpenCV*. Matricu *kernel* je potrebno inicijalizirati za potrebe korištenja funkcije *cv.filter2D*. Ta funkcija omogućuje primjenu filtera na sliku koji vrši operacije izoštravanja, zamućenja, naglašavanja rubova objekata i sl. Ovisno o dimenziji i članovima formirane matrice *kernel,* mogu se postići različite prethodno navedene funkcije.

U ovom se slučaju koristi matrica koja omogućuje primjenu zamućenja slike kako bi se uklonili šumovi u slici, odnosno kako bi se istakle one karakteristike bitne korisniku. Nakon primjene filtera, slijedi korištenje funkcije *cv.split*. Svaka slika u boji sadrži 3 različita kanala, ovisno o tome u kojem je *colorspace*-u slika izražena. U ovom slučaju se koristi *HSV* kanal, odnosno svaka slika sadrži *hue, saturation* i *value* kanal. Kako bi se mogla koristiti *cv.threshold* funkcija, potrebno je stvoriti jedno-kanalnu sliku, odnosno potrebno je izuzeti svaki kanal *HSV* slike u zasebnu sliku. Upravo tome služi *cv.split* funkcija. Nakon razdvajanja slike, odabire se ona koja je pogodna za daljnje prosljeđivanje u *cv.threshold* funkciju. Ta funkcija služi kao svojevrstan filter svih piksela slike. Svaki se piksel slike uspoređuje s graničnom vrijednošću koju postavlja korisnik. Nakon toga, ovisno o odabranom načinu *threshold*-a, vrši se postavljanje vrijednosti tog piksela na specifičnu vrijednost. U ovom se slučaju *thresh* vrijednost postavlja na vrijednost 255/3, dok se maksimalna vrijednost piksela postavlja na 255. U slučaju korištenja *THRESH*_BINARY metode, to znači da se svi pikseli vrijednosti manje od *thresh* vrijednosti postavljaju u vrijednost 0, dok se svi ostali postavljaju na vrijednost 255. No, pošto se koristi metoda *THRESH_BINARY_INV*, događa se upravo suprotno od prethodno navedenog scenarija. Dodatno se koristi *THRESH_OTSU* metoda koja omogućuje optimalno traženje *thresh* vrijednosti u pojedinim područjima slike, time dajući bolje rezultate. Nakon prethodno objašnjenih obrada, dobiva se rezultat u nastavku. Važno je napomenuti kako će u nastavku biti prikazivani rezultati za jednu kameru radi boljeg pregleda. Iste se operacije primjenjuju na sve tri kamere.



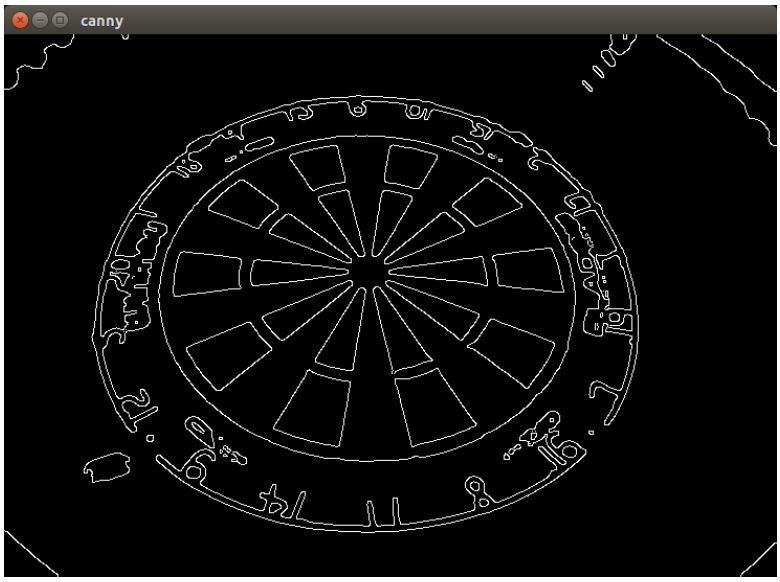
a) b)

Slika 3.8. a) početna slika pikado mete, b) obrađena slika pikado mete

```
    cont, _ = cv.findContours(dartboard, cv.RETR_LIST, cv.CHAIN_APPROX_NONE)
    for c in cont:
        if 50000 < cv.contourArea(c) < 110000:
            ellipse = cv.fitEllipse(c)
            cv.ellipse(blackcont,ellipse,color = (255,255,255), thickness=2) # type: ignore

    canny = cv.cvtColor(dartboard3, cv.COLOR_BGR2HSV)
    kernel = np.ones((5,5), np.float32) / 25 # type:ignore
    canny = cv.filter2D(canny, -1, kernel)
    h,s,canny = cv.split(canny)
    _, canny = cv.threshold(canny, 255/2,255,cv.THRESH_BINARY|cv.THRESH_OTSU)
    canny = cv.Canny(canny,150/2,150, apertureSize=5)
    linesparam = [(110,100),(80,90),(100,100)]
    lines = cv.HoughLines(canny, 1, np.pi/linesparam[camera][0], linesparam[camera][1]) #
type: ignore

    anglelist = [(5,8),(-81,-78),(33,35),(-44,-42),(51,54),(-26,-23)]
    for i in range(0, len(lines)):
        rho = lines[i][0][0]
        theta = lines[i][0][1]
        a = math.cos(theta)
        b = math.sin(theta)
        x0 = a * rho
        y0 = b * rho
        pt1 = (int(x0 + 1000*(-b)), int(y0 + 1000*(a)))
        pt2 = (int(x0 - 1000*(-b)), int(y0 - 1000*(a)))
        angle = (math.atan2(pt2[1]-pt1[1], pt2[0]-pt1[0]) * 180.0) / np.pi # type: ignore

        if anglelist[anglenum][0] <= angle <= anglelist[anglenum][1] or
anglelist[anglenum+1][0] <= angle <= anglelist[anglenum+1][1]:
            cv.line(blackline, pt1, pt2, (255,255,255),2)
    anglenum +=2
    black = cv.bitwise_and(blackcont,blackline)
```

Nakon zadovoljavajuće obrade slike, slijedi potražnja željenih kontura. Korištenjem *cv.findContours* naredbe moguće je pronaći sve konture koje se nalaze na toj slici. Zatim ih je potrebno filtrirati tako da ostanu samo one konture koje je potrebno pronaći. U ovom se slučaju traži kontura elipse vanjskog ruba pikado mete. Nakon što se izuzme željena kontura, poziva se funkcija *cv.fitEllipse* koja koristi metodu najmanjih kvadrata kako bi sve 2D točke pronađene konture smjestila u elipsu koja ih najbolje opisuje. U konačnici se poziva *cv.ellipse* funkcija koja iscrtava pronađenu elipsu na prethodno spomenutu sliku *blackcont* koja se kasnije koristi za pronalazak ulaznih točaka izračuna matrice homografije.

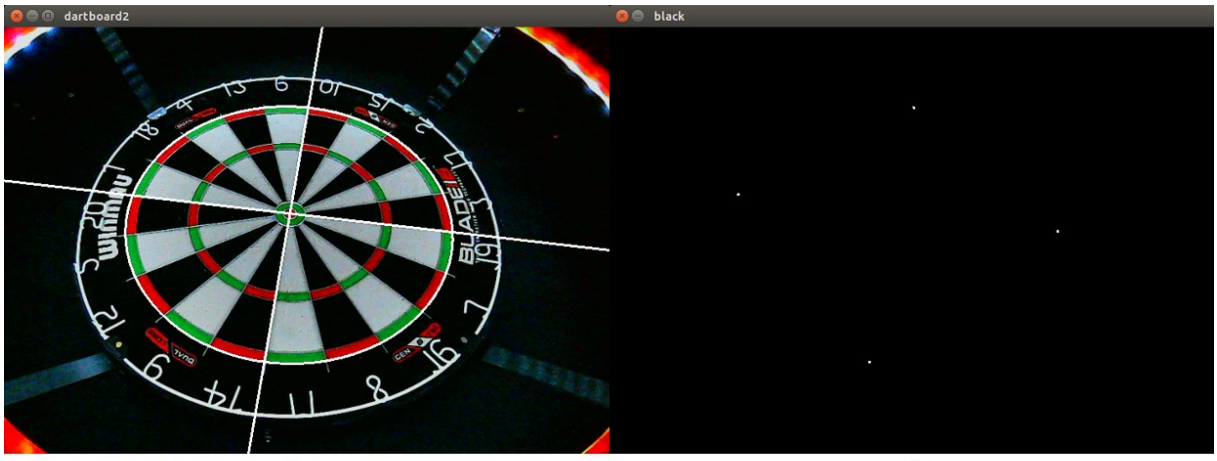
Nadalje, slijedi obrada slike koja se priprema za pronalaženje linija pikado mete. Ponovno se koriste metode koje su prethodno viđene, a sljedeća važna funkcija koja još nije spomenuta je *cv.Canny*. Ta funkcija omogućuje traženje i isticanje rubova slike. Prva brojčana vrijednost predstavlja *thresh* vrijednost za rubove. Dakle, ako neki rub ima veću vrijednost od te vrijednosti, on se označava kao siguran rub i iskazuje se na slici. Druga vrijednost predstavlja *thresh* vrijednost koja omogućuje uklanjanje slabih, krivo prepoznatih rubova koji su vezani na prethodno nađene sigurne rubove. Svi pikseli intenziteta između prethodnih vrijednosti se uklanjaju i ne uzimaju u obzir kao rubovi. U konačnici se koristi dodatan parametar *apertureSize* koji određuje veličinu matrice za metodu *Sobel* koja se koristi pri pronalaženju rubova. Veća veličina uzrokuje bolje uklanjanje šumova uz mogućnost zanemarivanja nekih dijelova rubova slike.



Slika 3.9. Primijenjen *canny* operator na slici pikado mete

Nakon pripreme slike, slijedi korištenje operacije *cv.HoughLines* koja se koristi za pronalazak linija pikado mete. Kao što je vidljivo na prethodnoj slici, cilj je pronaći rubove pikado mete te ih prikazati kao linije koje se kasnije koriste u operaciji pronalaska ulaznih točaka za izračun matrice homografije. Prethodno spomenuta metoda koristi prikaz linija u polarnim koordinatama koje izražavaju poziciju točke udaljenošću od ishodišta i kutem kojeg taj pravac zatvara s osi x. Važno je razumjeti kako ta metoda funkcionira. Nakon generirane *canny* slike pikado mete poziva se *cv.HoughLines* metoda koja prvo uzima svaki piksel ruba s te slike. Potrebno je imati na umu da je svaki pronađeni rub u slici zapravo niz piksela istog intenziteta. Nakon toga se računa udaljenost od ishodišta do odabranog piksela.

Slijedi generiranje niza linija kroz odabrani piksel tako da se iscrtavaju linije od 0 do 180 stupnjeva. U konačnici preostaje izračunati koliko piksela istog intenziteta presijeca pojedina generirana linija. Ona linija koja sadrži najviše glasova predstavlja kandidat za glavnu liniju. Isto tako se vrši izračun za sve ostale piksele ruba u slici. Nakon cjelokupnog izračuna, taži se ona kombinacija udaljenosti i kuta linija koja se najviše ponavlja. Ta linija predstavlja konačnu odabranu liniju s pomoću prethodno spomenute metode. Nakon pronalaska svih linija, slijedi ekstrakcija udaljenosti i kuta svake linije. Zatim se primjenom trigonometrijskih funkcija vrši izračun koordinata u klasičnom prikazu (*x,y*) one točke koja se nalazi na liniji, a koja je najbliža ishodištu. Konačno se izračunavaju krajevi linije te se spremaju u varijable *pt1* i *pt2*. Odabrana vrijednost 1000 osigurava da su krajevi linije izvan slike, odnosno da se sama pronađena linija proteže kroz cijelu sliku. Kako bi se dodatno filtrirale pronađene linije, vrši se izračun kuta pod kojim je svaka linija u odnosu na horizontalnu os te se vrijednost izražava u stupnjevima. Za potrebe izračuna matrice homografije, vrlo je važno pripaziti kako se vrši ekstrakcija linija. Potrebno je izabrati dvije linije koje su međusobno pod 90 stupnjeva. Na taj će se način osigurati pronalazak točaka na brojevima 20, 3, 6 i 11 koje predstavljaju četverokut zbog kasnijeg procesa homografskog izračuna koji će biti objašnjen kasnije. Izdvojene se linije iscrtavaju na slici *blackline*, a zatim se vrši operacija *cv.bitwise_and* čiji su ulazni parametri slike izdvojene elipse, odnosno linija. Ona predstavlja logičku konjunkciju koja daje istinu jedino kada su pikseli iste pozicije na obje slike intenziteta većeg od 0. Na taj se način pronalaze presjecišta elipse i linija.



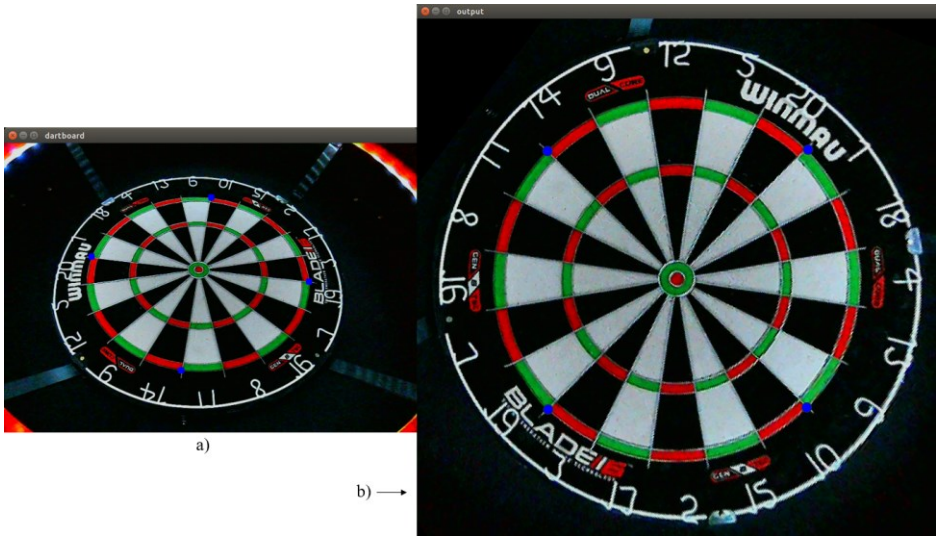
a) b)

Slika 3.10. a) iscrtana elipsa i linije na slici pikado mete, b) presjecišta linija i elipse

```
    black = cv.cvtColor(black, cv.COLOR_BGR2GRAY)
    contours, _ = cv.findContours(black,cv.RETR_EXTERNAL,cv.CHAIN_APPROX_NONE)
    avgpoints = []
    for c in contours:
        M = cv.moments(c)
        cx = int(M["m10"]/M["m00"])
        cy = int(M["m01"]/M["m00"])
        avgpoint = (cx,cy)
        cv.circle(dartboard2,avgpoint,1,(255,0,0),10)
        avgpoints.append(avgpoint)

    order = [(0,2,3,1),(2,3,1,0),(0,2,3,1)]

    avgpoints=[avgpoints[order[camera][0]],avgpoints[order[camera][1]],avgpoints[order[came
ra][2]],avgpoints[order[camera][3]]]
    avgpoints = np.float32(avgpoints) # type: ignore

    points2 = np.float32([[240,240], [715, 240], [715, 715], [240, 715]]) # type: ignore
    matrix, status = cv.findHomography(avgpoints,points2, cv.RANSAC, 1.0)
    pickle.dump(matrix, open(
"/home/fsb/Desktop/diplomski_opencv/tkinter/matrixwarp{}.pkl".format(camera), "wb"))
    output = cv.warpPerspective(correctedimage,matrix,dsize = (950,950),
flags=cv.INTER_NEAREST)
    if camera == 2:
        output = cv.flip(output,1)
    cv.imshow('output',output)
    cv.imwrite("/home/fsb/Desktop/diplomski_opencv/center_align/dartboardhomo{}.png".format
(camera), output)
    cv.waitKey(0)
    cv.destroyAllWindows()
```

Kako bi se mogle izdvojiti pozicije presjecišta, ponovno se koristi funkcija *findContours*. Pošto su presjecišta većih dimenzija od veličine jednog piksela, potrebno je pronaći centroid svakog presjecišta kako bi odabrali središnju točku istih. Za te potrebe je potrebno koristiti funkciju *cv.moments*. Ona omogućuje izračun *momenta* svake konture, odnosno težišta konture. *M10* predstavlja distribuciju piksela konture u smjeru osi *x*, dok *M01* predstavlja distribuciju piksela konture u smjeru osi *y*. Centar konture određuje se dijeljenjem pojedinih *momenata* s površinom konture označenom s *m00*. Koordinate centra konture spremaju se u varijablu *avgpoint*. Svaka pronađena točka sprema se u listu *avgpoints* određenim redoslijedom koji osigurava pravilno izuzimanje pojedine točke iz te liste za izračun homografije. Vrlo je važno osigurati podudarnost točaka s izvorne slike i odredišne slike za homografiju jer će jedino takav redoslijed osigurati ispravnost procesa.

U listu *points2* spremaju se ručno odabrane točke koje predstavljaju poziciju izvorišnih točaka u destinacijskoj slici. Vidljivo je kako su u listi točke koje su međusobno jednako udaljene. To je iznimno važno iz razloga da destinacijska slika ne bude rastegnuta ili stisnuta, odnosno da bude vjerni prikaz ispravljene perspektive slike. Napokon se dolazi do pozivanja funkcije *cv.findHomography*. Ulazni podaci tu pozicije točaka s izvorne slike, pozicije točaka s destinacijske slike te metoda kojom se računa matrica homografije. U ovom slučaju se koristi metoda *cv.RANSAC*. Generalno ta metoda omogućuje odabir 4 nasumično odabrana para točaka s pomoću kojih se računa matrica homografije. Nakon toga se vrši izračun destinacijskih točaka za sve preostale točke s izvorne slike s pomoću prethodno izračunate matrice homografije. Vrši se izračun broja točaka koje zadovoljavaju uvjet bliskosti. Dakle, odvija se usporedba između pozicije točke izračunate s pomoću homografije i stvarne pozicije te iste točke u destinacijskoj slici. Ako je razlika u poziciji manja od greške koju korisnik sam postavlja (u ovom slučaju 1.0), tada je to validna točka koja ulazi u zbroj izračuna točaka koje zadovoljavaju uvjet. U konačnici se taj proces izvršava par puta te se bira ona matrica koja ima najveći broj zadovoljavajućih točaka (eng. *inlier*-a). Zatim se poziva već spomenuta metoda za spremanje izračunate matrice homografije te se vrši pozivanje metode *cv.warpPerspective* koja služi za primjenu matrice homografije na izvorišnu sliku kako bi se izvršilo ispravljanje perspektive slike. Dodatna *if* petlja služi za korekciju ispravljene perspektive treće kamere. Naime, zbog FOV-a treće kamere (odozgo) ispravljena perspektiva daje obrnut rezultat koji je potrebno popraviti korištenjem funkcije *cv.flip*.



Slika 3.11. a) odabrane točke s izvorne slike za izračun homografije, b) rezultat primijene matrice homografije na izvornu sliku

4. DETEKCIJA POLJA METE

Sljedeći je korak stvaranje algoritma koji određuje broj postignutih bodova bačene strelice. Ideja je pronaći vrh strelice u klasičnom FOV-u kamere te ga izdvojiti kao točku, a zatim korištenjem prethodno izračunate matrice homografije pronaći poziciju te točke u slici ispravljene perspektive. Nakon toga se u polarnom koordinatnom sustavu određuje udaljenost te točke od centra pikado mete i kut u odnosu na određeni pravac. Na temelju tih informacija određuje se broj postignutih bodova. Prvo je potrebno postaviti referentne pozicije u odnosu na koje će se računati bodovi.

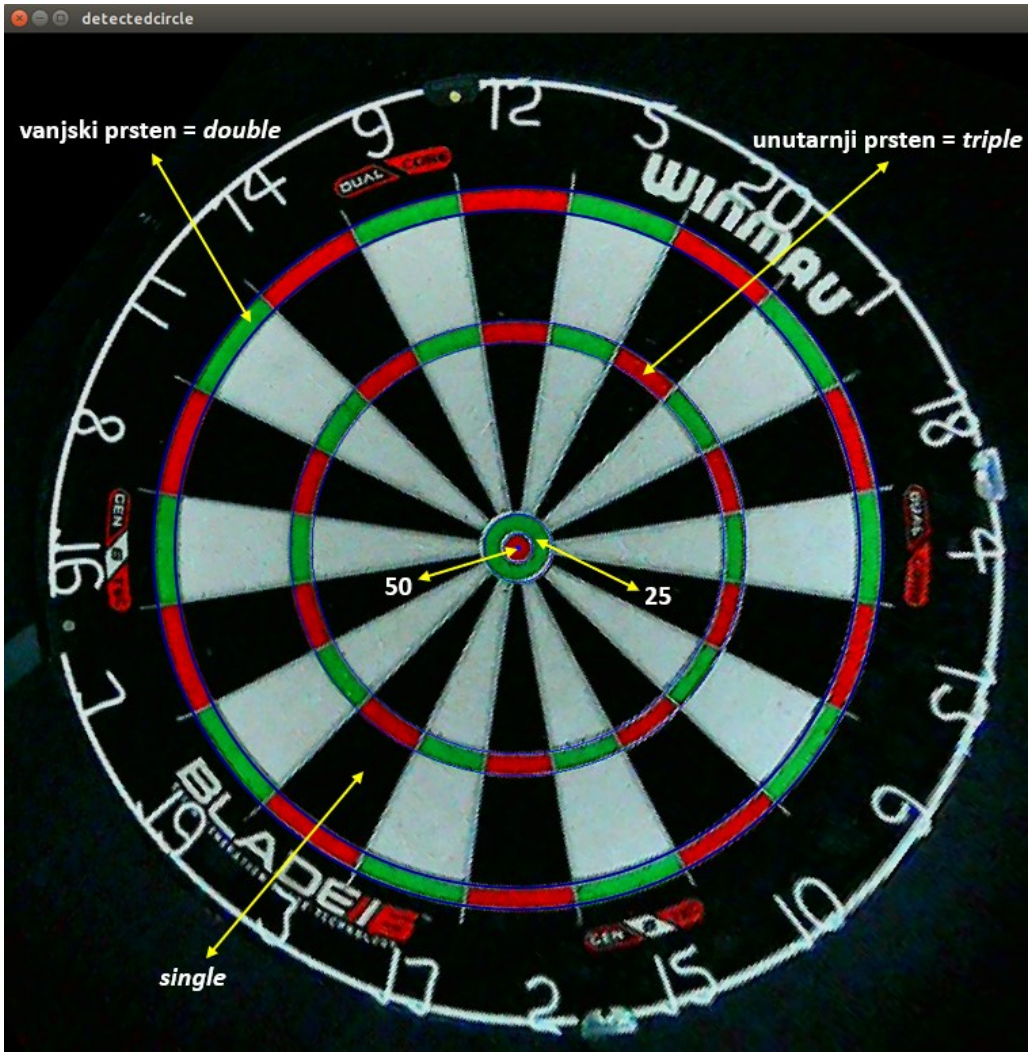
```
    blurparam = [(9,9,0.92),(9,9,0.92),(9,9,1.05)]
    houghparam = [(1,30,100,50,10,40),(1,30,100,50,10,40),(1,30,100,50,10,40)]
    circleslist = []

    darthomo=cv.imread("/home/fsb/Desktop/diplomski_opencv/center_align/dartboardhomo{}.png
".format(camera))

    gray_darthomo = cv.cvtColor(darthomo, cv.COLOR_BGR2GRAY)
    blur = cv.GaussianBlur(gray_darthomo, (blurparam[camera][0],blurparam[camera][1]),
blurparam[camera][2])
    circles = cv.HoughCircles(blur,cv.HOUGH_GRADIENT, dp=houghparam[camera][0],
minDist=houghparam[camera][1], param1=houghparam[camera][2], param2=houghparam[camera][3],
minRadius=houghparam[camera][4], maxRadius=houghparam[camera][5])

    if circles is not None:
        circles = np.round(circles[0,:]).astype("int")
        circleslist.append(circles)

        for (x,y,r) in circles:
            cv.circle(darthomo, (x,y), 1, (255,0,0), 2)
            cv.circle(darthomo, (x,y), 14, (255,0,0), 1)
            cv.circle(darthomo, (x,y), 32, (255,0,0), 1)
            cv.circle(darthomo, (x,y), 191, (255,0,0), 1)
            cv.circle(darthomo, (x,y), 211, (255,0,0), 1)
            cv.circle(darthomo, (x,y), 315, (255,0,0), 1)
            cv.circle(darthomo, (x,y), 335, (255,0,0), 1)
            print(x,y)
    cv.imshow("detectedcircle", darthomo)
    cv.waitKey(0)
    cv.destroyAllWindows()
```

Prvo su inicijalizirane liste koje su potrebne za rad ovog dijela koda. U listu *blurparam* spremljeni su parametri za kasnije korištenje funkcije *cv.GaussianBlur* u svrhu uklanjanja šuma u slici.

U listu *houghparam* spremljeni su parametri potrebni za korištenje funkcije *cv.HoughCircles* u svrhu pronalaska centra mete. *Circleslist* predstavlja listu koja omogućuje spremanje pronađenih krugova u slici. U varijablu *darthomo* se učitava prethodno dobiveni rezultat ispravljene perspektive kamere. Nakon toga se vrši pretvorba u crno-bijeli *colorspace* te se primjenjuje prethodno spomenuta funkcija *cv.GaussianBlur*. Rezultat primjene je sličan onom koji se dobiva korištenjem funkcije *cv.filter2D.* Cilj je ukloniti šum u slici tako da se vrši zamućenje cjelokupne slike pri čemu ostaju vidljive one bitne karakteristike. Prvi parametar funkcije predstavlja veličinu matrice koja mora biti pozitivna i neparnih dimenzija. Drugi parametar je standardna devijacija u smjeru osi *x*. U suštini taj parametar daje mogućnost kontrole primjene zamućenja u smjeru osi *x* pri čemu veće vrijednosti tog parametra povećavaju efekt zamućenja. Nakon primjerene obrade slike, slijedi pozivanje funkcije *cv.HoughCircles* za pronalazak krugova u slici. Poziv funkcije zahtjeva više ulaznih argumenata, ovdje objašnjenih redoslijedom ekvivalentnom onom u kodu[18]:

- *image* - slika na kojoj se odvija operacija pronalaska krugova
- *method* – metoda koja se koristi za pronalazak krugova
- *dp* – faktor koji utječe na preciznost pri određivanju krugova
- *minDist* – minimalna udaljenost centara dvaju krugova
- *param1* – određuje minimalni gradijent ruba potreban za piksel kako bi se on smatrao dijelom tog ruba – veće vrijednosti tog parametra daju manje detektiranih rubova u slici
- *param2* – *threshold* vrijednost broja glasova potrebnih da bi se rub smatrao krugom – veće vrijednosti tog parametra daju manje detektiranih krugova u slici
- *minRadius* – minimalni radijus kruga
- *maxRadius* – maksimalni radijus kruga

Nakon pronalaska krugova, koordinate centara istih je potrebno spremiti kao *integer* vrijednosti kako bi se dalje mogli koristiti za druge funkcije. U konačnici se pronađeni centar koristi kao centar za sve krugove koji se iscrtavaju na slici. Važno je napomenuti kako je nađeni krug onaj koji omeđuje dio mete čiji pogođeni iznos nosi 25 bodova. Iscrtavaju se svi krugovi koji predstavljaju granice između određenih bodova. Kako bi koncept bio jasniji, valja proučiti iduću sliku.



Slika 4.1. Pronađen centar i bodovne granice pikado mete

Na slici je jasno vidljiv pronađen centar mete te iscrtane bodovne granice krugovima plave boje. Valja istaknuti kako se iz iscrtavanja bodovnih granica može zaključiti kako je izračunata matrica homografije dovoljno precizno određena pošto se iscrtane granice dobro preklapaju sa stvarnim granicama. To je poprilično bitno zbog mogućnosti korištenja izračuna udaljenosti točke vrha strelice od pronađenog centra pikado mete. Vanjski prsten predstavlja dvostruku vrijednost, unutarnji prsten predstavlja trostruku vrijednost, a *single* vrijednost označava jednostruku vrijednost polja u kojem se strelica nalazi.

```
centercir = [(476,476),(480,478),(472,476)]
refpoint = [(627,177),(627,183),(622,178)]
regions = [14,32,191,211,315,335] # regions for multiplier
refanglelist = [63.281,63.205,63.280]
```

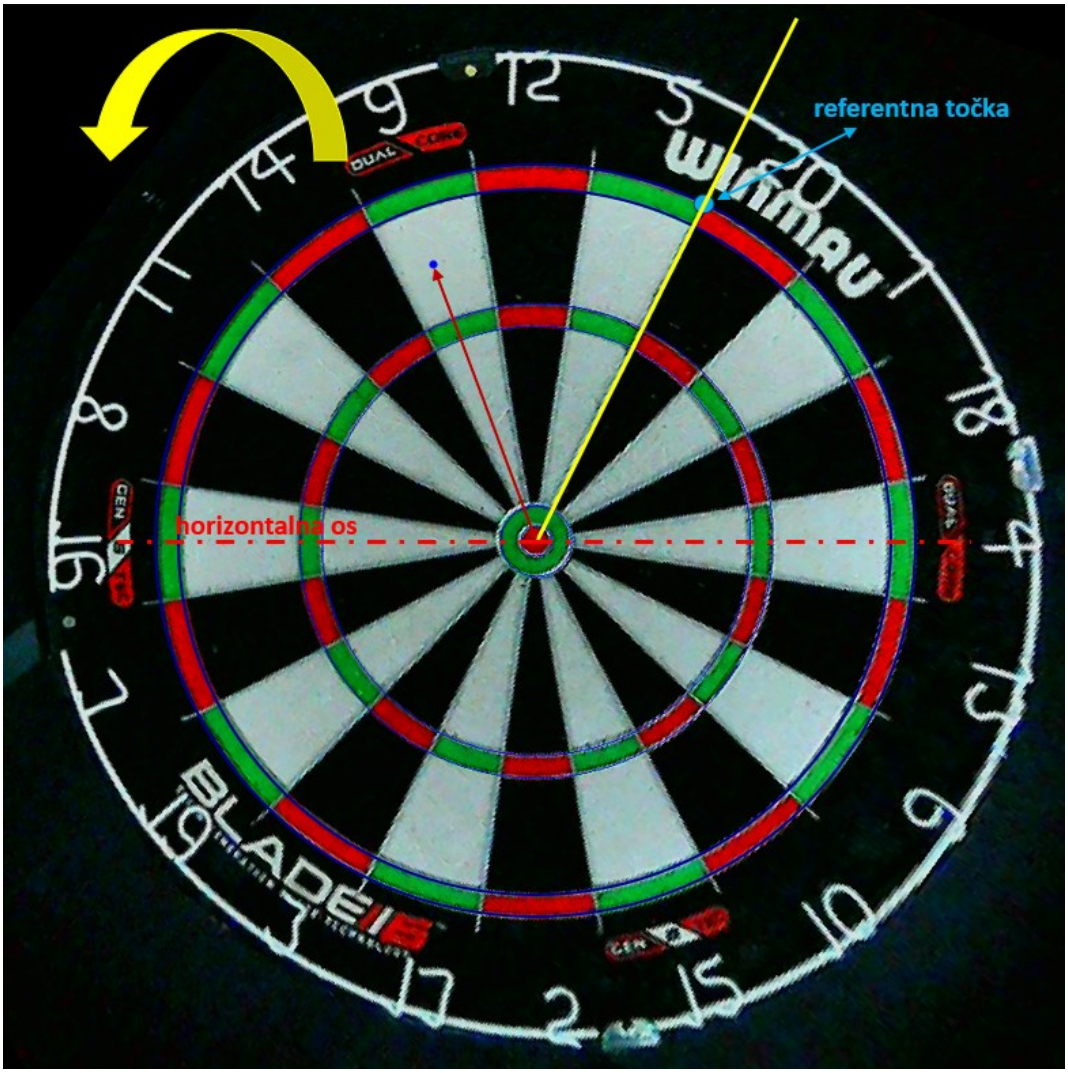
```
def getscore(point, nodart, flagnum):
    score = 0
    cv.circle(nodart, (point[0],point[1]), 1, (0,255,255),2)
    cv.circle(nodart, centercir[flagnum],1,(0,255,0),2)
    cv.circle(nodart, refpoint[flagnum], 1, (127,255,127),2)
    cv.line(nodart, centercir[flagnum], refpoint[flagnum], (0,255,255), 1)

    dist = round(((point[0]-centercir[flagnum][0]) ** 2 + ((point[1])-
centercir[flagnum][1]) ** 2) ** 0.5, 2)

    # refdir1 = (refpoint[flagnum][0] - centercir[flagnum][0])
    # refdir2 = (refpoint[flagnum][1] - centercir[flagnum][1])
    # refangle = ((math.atan2(refdir2,refdir1)) * 180) / np.pi # type: ignore
    # print(refangle)
    refangle = refanglelist[flagnum]

    dir1 = (point[0] - centercir[flagnum][0])
    dir2 = (centercir[flagnum][1] - point[1])
    angle = math.fmod(((((math.atan2(dir2,dir1)) * 180) / np.pi) + 360 - refangle), 360) #
type: ignore
    region = angle // 18.0
```

Prvo je potrebno definirati liste za ispravno odvijanje algoritma. Za svaku su kameru, u svakoj listi, zapisani parametri kako bi se čim točnije odredio broj postignutih bodova. U listi *centercir* nalaze se koordinate centra pikado mete za pojedine kamere. Lista *refpoint* sadrži koordinate točke u odnosu na koju se određuje pozicija vrha strelice za izvršavanje procesa bodovanja. Nadalje, lista *regions* sadrži udaljenosti od centra mete koje definiraju granice za proces bodovanja (*single*, *triple, double*). U konačnici, lista *refanglelist* sadrži referentni kut određen prema točkama iz liste *refpoint*, a koji se koristi za određivanje broja postignutih bodova. Nakon toga se definira funkcija koja se koristi za bodovanje. Ulazni argumenti funkcije su pronađena točka pozicije vrha strelice, ispravljena perspektiva slike kamere i varijabla *flagnum* koja označava broj kamere. Zatim se vrši inicijalizacija *score* varijable koja se koristi za pohranu broja postignutih bodova. Naredne 4 naredbe služe za iscrtavanje pojedinih točaka, odnosno linije na učitanoj slici. Prva naredba iscrtava poziciju registrirane točke vrha strelice na ispravljenoj perspektivi kamere. Druga naredba iscrtava centar pikado mete za pojedinu kameru, dok treća naredba iscrtava poziciju referentne točke prema kojoj se određuje pozicija točke. U konačnici se prethodne dvije točke povezuju pravcem kako bi se provjerila točnost određivanja centra i referentne točke.

Vrijedi napomenuti kako prethodne 4 naredbe nisu nužne za izvršavanje, već služe kao dodatna provjera korisniku kako bi se provjerila točnost određivanja pozicije vrha strelice. Varijabla *dist* se koristi za određivanje udaljenosti točke vrha strelice od centra pikado mete u ispravljenoj perspektivi slike kamere. Za izračun udaljenosti koristi se poznata formula izračuna euklidske udaljenosti koja predstavlja najkraću udaljenost između dviju točaka u sljedećoj formi:

$$udaljenost = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}. \quad (25)$$

Zakomentirani je dio koda namjerno ostavljen kako bi se mogao vidjeti način određivanja referentnog kuta potrebnog za proces bodovanja. Vrijednosti liste *refanglelist* određene su korištenjem formule za varijablu *refangle*. Biblioteka *math* koristi se za pozivanje funkcije kojom se računa tangens kuta između horizontalne osi slike i vektora koji povezuje centar mete i referentnu točku s obzirom na koju se određuje polje u kojem se nalazi vrh strelice. Nakon pronalaska kuta koji referentna točka zatvara s horizontalnom osi, potrebno je računati koji kut zatvara točka vrha strelice s horizontalnom osi. To se odvija prvotnim pronalaskom vektora smjera koji povezuje centar s točkom vrha strelice, a zatim slijedi izračun kuta koji ona zatvara s horizontalnom osi pozivanjem funcije *math.atan2*. Vrlo je važno naglasiti kako je potrebno koristiti tu funkciju da bi se izbjegla potencijalna višestruka rješenja, ovisno o kvadrantu u kojem se nalazi točka vrha strelice. Dobiveni se rezultat pretvara u stupnjeve te se nadodaje 360 stupnjeva kako bi se osiguralo rješenje veće od 0. Također se oduzima prethodno izračunata referentna vrijednost *refangle* od konačnog rješenja kako bi se osiguralo mjerenje kuta od referentne točke. Korištenje *math.fmod* funkcije omogućava da se dobiveno rješenje ograniči u rasponu od 0 – 360 stupnjeva. Napokon se za određivanje pogođene regije vrši pronalazak varijable *region* koja kao rezultat daje broj od 0 do 19. Svaki broj predstavlja jednu regiju na pikado meti između kojih je kut 18 stupnjeva.



Slika 4.2. Princip određivanja pozicije vrha strelice i adekvatno bodovanje

Na slici iznad je vidljiva horizontalna os koja se koristi za izračun više različitih podataka. U gornjem desnom kutu vidljiva je referentna točka koja je povezana s centrom preko žutog pravca. Kut između crvene horizontalne osi i žutog pravca predstavlja varijablu *refangle*. Dakle, žuti je pravac početna pozicija od koje se mjeri kut prema vidljivoj žutoj strelici – obrnuto od smjera kazaljke na satu. Primjerice, na slici je označena tamnoplava točka u polju 9. Ta točka predstavlja poziciju vrha strelice. Tamnocrvena kota označava udaljenost koja se računa prema izrazu (25). Osim udaljenosti, prisutan je i izračun kuta koji je onaj između žutog pravca i tamnocrvene kote. Taj se kut koristi za funkciju *math.fmod* koja kao rezultat daje polje u kojem se točka nalazi.

```
    if region == 0:
        score = 5
    elif region == 1:
        score = 12
    elif region == 2:
        score = 9
    elif region == 3:
        score = 14
    elif region == 4:
        score = 11
    elif region == 5:
        score = 8
    elif region == 6:
        score = 16
    elif region == 7:
        score = 7
    elif region == 8:
        score = 19
    elif region == 9:
        score = 3
    elif region == 10:
        score = 17
    elif region == 11:
        score = 2
    elif region == 12:
        score = 15
    elif region == 13:
        score = 10
    elif region == 14:
        score = 6
    elif region == 15:
        score = 13
    elif region == 16:
        score = 4
    elif region == 17:
        score = 18
    elif region == 18:
        score = 1
    elif region == 19:
        score = 20
    else:
        print("Error calculating score.")

    for distcount in range(0,6):
        if dist <= regions[distcount]:
            if distcount == 0:
```

```
                score = 50
            elif distcount == 1:
                score = 25
            elif distcount == 2 or distcount == 4:
                score *= 1
            elif distcount == 3:
                score *= 3
            elif distcount == 5:
                score *= 2
            break

    if dist > regions[5]:
        score = 0

    cv.imshow("nodart", nodart)
    cv.waitKey(0)
    cv.destroyAllWindows()

    return score
```

Zadnji dio koda predstavlja određivanje bodovanja prema prikupljenim informacijama regije i udaljenosti. Prvo se određuje polje u kojem se nalazi vrh strelice. Primjerice, potrebno je odrediti polje u kojem se nalazi točka vrha strelice sa slike 4.2. Izvršavanjem koda, varijabla *angle* poprima vrijednost od 46 stupnjeva. Kad se ta vrijednost podijeli s 18 (širina pojedinog polja mete), dobiva se rezultat 2.55. Vrijedi naglasiti kako se ovdje koristi cjelobrojno dijeljenje korištenjem operacije „//“, čime se kao rezultat dobiva broj bez ostatka. Dakle, u ovom slučaju je rezultat 2. S tim se rezultatom ulazi u *if-elif* dio koda gdje je svakom cjelobrojnom broju pridruženo odgovarajuće polje. Vidljivo je da se, prema dobivenom rezultatu 2, varijabli *score* pripisuje vrijednost 9. Nakon određivanja polja u kojem se vrh nalazi, slijedi određivanje udaljenosti od centra mete kako bi se odlučilo o tome je li vrh u *single*, *double* ili *triple* području. Ulazi se u *for* petlju gdje se provjerava je li prethodno izračunata varijabla *dist* manja od definiranih udaljenosti liste *regions*. U trenutku kad se ispuni ta vrijednost, provjerava se koje je vrijednosti varijabla *distcount*. U ovom je slučaju ta varijabla vrijednosti 4, što znači da je udaljenost vrha od mete između 211 i 315. Prema tome, točka se nalazi u *single* polju zbog čega se varijabla *score* množi s 1. U slučaju da je udaljenost veća od svih onih naznačenih u listi *regions*, znači da se vrh strelice nalazi izvan bodovnih granica mete. U konačnici se poziva funkcija *cv.imshow* koja omogućuje prikaz slike ispravljene perspektive kamere s iscrtanim točkama koje ukazuju na poziciju vrha, referentne točke i sl.

5. IDENTIFIKACIJA VRHA STRELICE

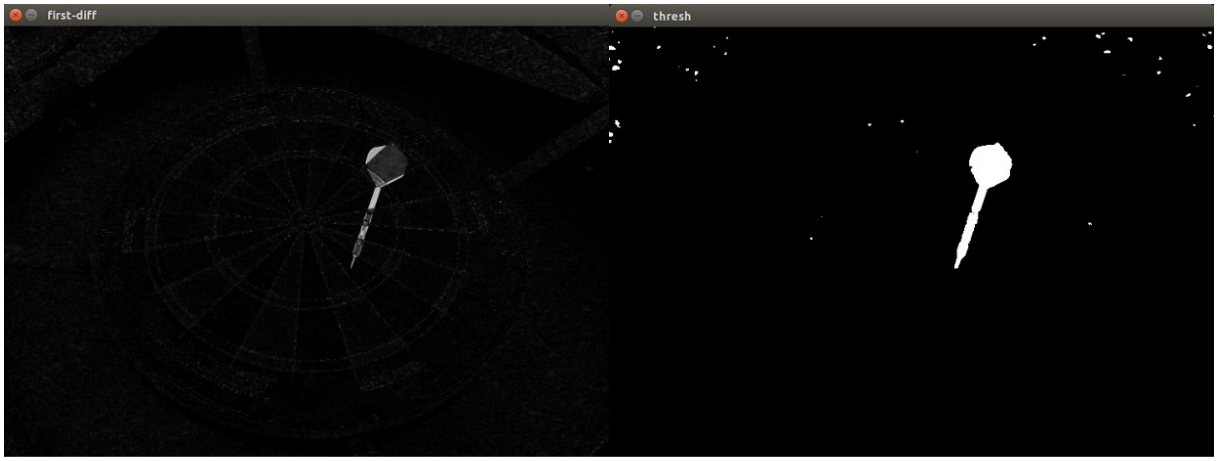
Posljednji je korak pronalazak vrha strelice. Kako bi cijeli algoritam imao smisla, potrebno je čim preciznije odrediti položaj vrha strelice u slici kamere pošto se upravo s obzirom na taj vrh odrađuje prethodno objašnjeno poglavlje vezano za bodovanje. Proces pronalaska vrha strelice je zamišljen kao sklop 2 različita postupka. Prvi je korak ekstrakcija strelice u slici kamere. Taj se postupak temelji na razlici između slika (prije i nakon upada strelice). Pozivanjem specifične funkcije biblioteke *OpenCV* moguće je izdvojiti razliku između dviju slika koju je dalje potrebno obraditi raznim funkcijama obrade slika. Nakon prikladne ekstrakcije i obrade, slijedi pronalazak vrha strelice korištenjem skupa 2D točaka pronađenih pozivanjem prethodno spomenute funkcije *cv.findContours*. Vrši se pronalazak pravca koji opisuje generalni smjer rasprostranjenih točaka, kao i pronalazak centra pronađene konture. Zatim se korištenjem tih informacija odvija postupak pronalaska smjera u kojem treba tražiti vrh te se u konačnici, nakon pronalaska smjera, vrši pronalazak samog vrha strelice.

5.1. SEGMENTACIJA KONTURE

```
def extractdart(nodart,dart, cammatrix):
    intrmatrix = cammatrix[0]; distcoefs = cammatrix[1]

    ############## UNDISTORTION ####################################################
    h,w = nodart.shape[:2]
    newintrmatrix, roi = cv.getOptimalNewCameraMatrix(intrmatrix, distcoefs, (w,h), 1,
(w,h))

    # Undistort
    nodart = cv.undistort(nodart, intrmatrix, distcoefs, None, newintrmatrix)
    dart = cv.undistort(dart, intrmatrix, distcoefs, None, newintrmatrix)

    # crop the image
    x, y, w, h = roi
    nodart = nodart[y:y+h, x:x+w]
    dart = dart[y:y+h, x:x+w]
    img = dart.copy()

    diff = cv.absdiff(dart,nodart)
    diff = cv.cvtColor(diff, cv.COLOR_BGR2GRAY)
    kernel = np.ones((5,5), np.float32) / 25 # type:ignore
    diff = cv.filter2D(diff,-1,kernel)
    _, thresh = cv.threshold(diff, 20, 255, cv.THRESH_BINARY)
    kernel = np.ones((3,3), np.uint8) # type: ignore
```

```
    diff = cv.dilate(thresh, kernel, iterations = 1)
    kernel = cv.getStructuringElement(cv.MORPH_RECT, (9,9))
    diff = cv.morphologyEx(diff, cv.MORPH_CLOSE, kernel, iterations=1)
```

Prvo je potrebno pozvati funkciju *extractdart* u kojoj se odvija pronalazak cjelokupne strelice u slici kamere. Iz definicije funkcije vidljivo je kako sadrži tri ulazna argumenta – *nodart* koji predstavlja sliku kamere u kojoj strelica nije prisutna, *dart* koji predstavlja sliku kamere u kojoj je prisutna strelica te *cammatrix* koji sadrži informacije o intrinzičnoj matrici i distorzijskim koeficijentima pojedine kamere. Nakon pozivanja funkcije slijedi već prije viđen proces obrade distorziranih slika. U obje je slike potrebno ukloniti distorzije kako bi se moglo raditi s preciznim informacijama sadržanih u pojedinim slikama. Nakon uspješnog uklanjanja distorzije te preostalih crnih piksela slike uslijed tog procesa, slijedi pozivanje funkcije *cv.absdiff* koja omogućuje izračun apsolutne razlike dviju slika. U ovom slučaju, rezultat takve operacije daje izdvojenu strelicu od ostatka okoline. Nakon uspješnog izdvajanja strelice, slijedi obrada slike u vidu naglašavanja bitnih karakteristika koje će se kasnije koristiti za pozivanje drugih funkcija. Ponovno se kreira *kernel* matrica koja se koristi u pozivu funkcije *cv.filter2D* koja je prethodno objašnjena. Slijedi pozivanje funkcije *cv.threshold* kako bi se stvorila binarna slika koja naglašava ono što korisnika zanima.

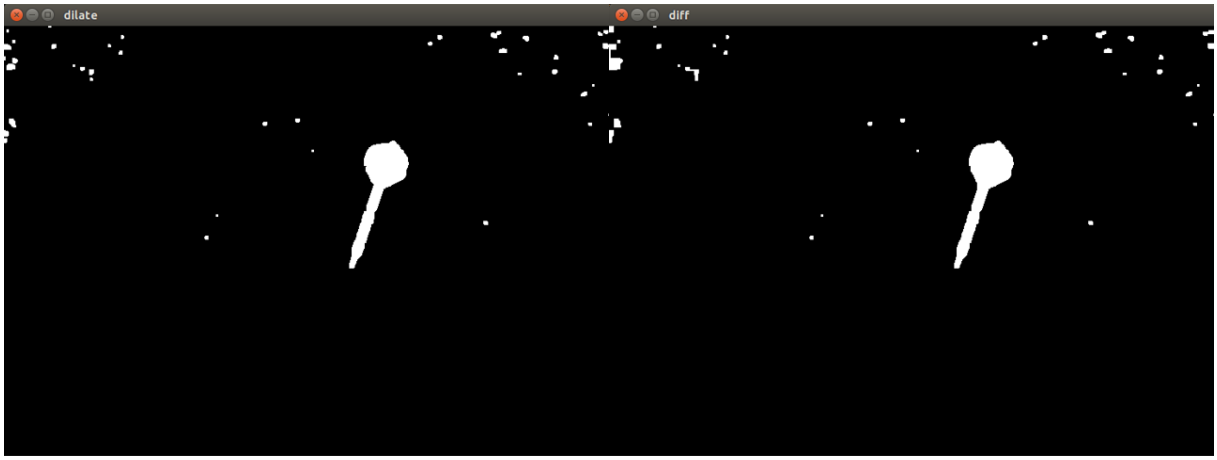


a) b)

Slika 5.1. a) rezultat apsolutne razlike dviju slika, b) rezultat primjene *threshold* funkcije

Na slici 5.1.b) je važno uočiti izdvojenu konturu strelice, uz naglasak na nužno kvalitetno izdvajanje cjelokupne neprekidne konture. Također je vidljiv niz šumova koji će u nastavku biti uklonjeni. Radi različitih scenarija pada osvjetljenja na strelicu, odbljesaka i sl., potrebno je pozvati funkciju *cv.dilate* koja proširuje postojeće izdvojene piksele koji se nalaze u *threshold*-anoj slici.

Tako se vrši osiguranje neprekidnosti konture koje je potrebno za kasnije određivanje usmjerenja strelice. Za potrebe pozivanja te funkcije ponovno je potrebno generirati matricu specifičnih dimenzija koje određuju količinu proširenja piksela. U konačnici se vrši pozivanje funkcije *cv.morphologyEx* koja ima sličnu funkciju kao i prethodno spomenuta funkcija *cv.dilate*. Međutim, ova funkcija se sastoji od dvije različite operacije. Prvo se primjenjuje prethodno spomenuta dilatacija, a zatim erozija koja predstavlja suprotnost dilataciji. Svrha korištenja ove funkcije je uklanjanje nepotpunosti unutrašnjosti bijelih piksela u slici, odnosno ispuna cjelokupne konture.



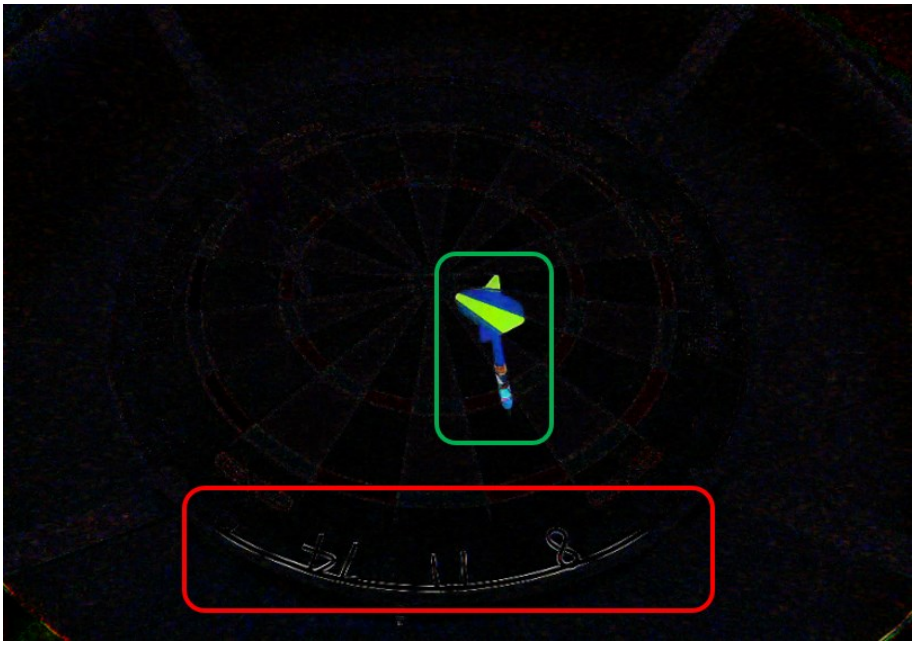
a) b)

Slika 5.2. a) rezultat korištenja funkcije dilatacije slike, b) rezultat korištenja funkcije zatvaranja (*morphologyEx*)

```
cont, _ = cv.findContours(diff, cv.RETR_EXTERNAL, cv.CHAIN_APPROX_NONE)
contlist = []; bigcont = []
for c in cont:
    if cv.contourArea(c) > 200:
        check = np.zeros_like(diff)
        cv.drawContours(check, [c], -1, (255), cv.FILLED)
        meanint = np.mean(meandiff[check == 255])
        if meanint > 40:
            bigcont = max(cont,key=cv.contourArea)
            contlist.append(bigcont)

rect = cv.minAreaRect(bigcont)
box = cv.boxPoints(rect)
transbox = box - np.mean(box,axis=0)
scaledbox = transbox * 1.25
retransbox = scaledbox + np.mean(box,axis=0)
box = np.int0(retransbox) #type:ignore
boxmask = np.zeros_like(diff)
```

```
    cv.drawContours(boxmask, [box], 0, (255,255,255), cv.FILLED)
    cv.drawContours(img, [box], 0, (255,0,0), 2)
    boxmask = cv.bitwise_and(thresh,thresh,mask=boxmask)
    boxmask = cv.cvtColor(boxmask, cv.COLOR_GRAY2BGR)
    return img,nodart,dart,boxmask,contlist, thresh
```

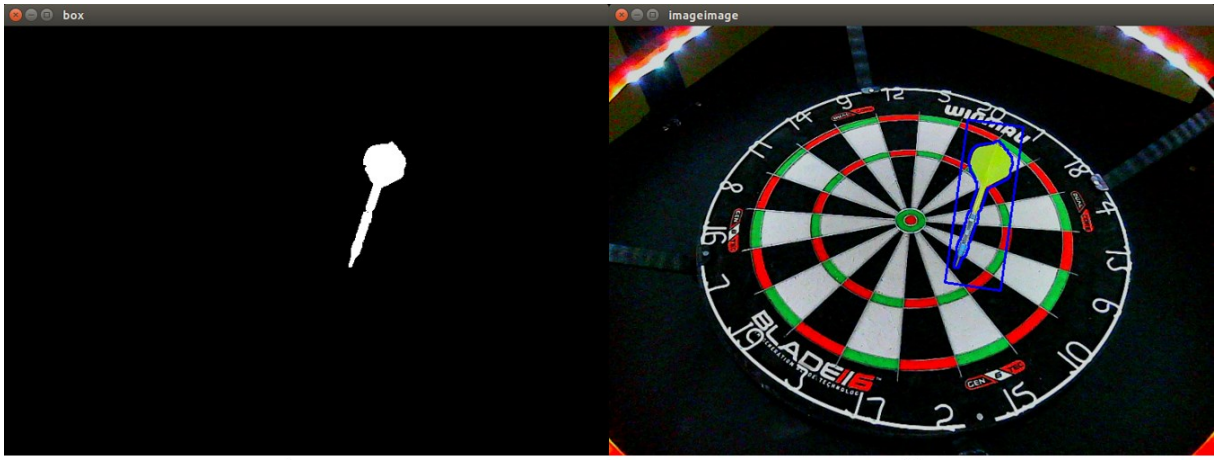
Slijedi nastavak obrade slike strelice. Prvo se pronalaze sve konture koje su prisutne na slici *diff*, a zatim se vrši filtriranje kontura temeljem njihovih karakteristika. Razlog potrebe za filtriranjem je vidljiv na slici u nastavku.



Slika 5.3. Filtriranje kontura

Na slici je jasno vidljiva željena kontura strelice označena zelenom bojom. Međutim, pri radu kamere se ponekad znaju događati anomalije koje uzrokuju detektiranja nepostojećih promjena koje mogu negativno utjecati na rad algoritma. Ako je kontura označena crvenim pravokutnikom površinom veća od konture strelice, tada će se ta kontura tretirati kao ona koja se koristi u daljnjoj analizi. Kako bi se to spriječilo, vrše se dva različita filtriranja. Prvo se filtriranje temelji na površini kontura, a zatim se u drugom filtriranju vrši izračun srednje vrijednosti intenziteta piksela. Pošto se anomalija generalno pojavljuje na tamnim područjima, moguće je koristiti pretpostavku da će kontura strelice generalno imati veću srednju vrijednost intenziteta piksela. Preostali se dio koda odnosi na odabir regije interesa. U ovom se dijelu koda rješava problem preostalog šuma slike koji je nužno ukloniti kako ne bi unosio krive i nepotrebne informacije u proces određivanja vrha strelice.

Poziv funkcije *cv.minAreaRect* omogućuje pronalazak pravokutnika minimalne površine koji obuhvaća cjelokupnu konturu, odnosno skup 2D točaka koji je opisuje. *Rect* sadrži informacije o centru, veličini te orijentaciji definiranog pravokutnika. Poziv funkcije *cv.boxPoints* omogućuje pronalazak 4 vrha prethodno definiranog pravokutnika. Iduća se naredba u kodu koristi za pronalazak centra pravokutnika. Traži se medijan točaka kutova duž svake osi što rezultira pronalaskom centra. Nakon toga se vrši prikladno skaliranje koje osigurava da je cjelokupna strelica obuhvaćena. U varijablu *retransbox* se spremaju nove koordinate točaka kutova skaliranog pravokutnika, a zatim se te koordinate pretvaraju u tip podatka *integer* kako bi se pravokutnik mogao iscrtati na slici. Ovaj je postupak potrebno primijeniti zbog mogućnosti neobuhvaćanja cjelokupne konture strelice. Naime, zbog raznih je smetnji moguće da vrh strelice ne bude povezan s preostalom konturom strelice, što znači da vrh neće biti obuhvaćen u procesu pronalaska konture. To je neprihvatljiva situacija jer se u vrhu strelice nalaze ključne informacije koje će osigurati ispravno bodovanje. Iz tog se razloga vrši odabir regije interesa na takav način da se ona dovoljno povećava skaliranjem kako bi se obuhvatili pikseli vrha koji potencijalno nisu povezani s ostatkom strelice. Kasnije se i ti pikseli uzimaju u obzir kao potencijalni vrhovi. Dakle, ovaj je pristup više mjera predostrožnosti u smislu uklanjanja opasnosti gubljenja bitnih informacija slike. Nakon toga se vrši iscrtavanje skaliranog pravokutnika pozivanjem funkcije *cv.drawContours*. U konačnici je potrebno stvoriti masku slike s pomoću prethodno skaliranog pravokutnika. Maska slike se obuhvaća pozivanjem funkcije *cv.bitwise_and* koja omogućava generiranje takve slike koja kao rezultat sadrži samo onaj dio slike koji je obuhvaćen odabranom maskom. U ovom se slučaju koristi maska skaliranog pravokutnika za odabir regije interesa u slici *thresh*. Osim prethodno spomenutih razloga korištenja maske, također je važno naglasiti da se na ovaj način uklanja preostali šum slike koji se ne nalazi unutar regije interesa. U konačnici se određeni podaci ove funkcije vraćaju u glavni algoritam kako bi se mogli koristiti u daljnjem postupku pronalaska točke vrha.



a) b)

Slika 5.4. a) rezultat korištenja maske za odabir regije interesa, b) prikaz iscrtanih kontura strelice u slici kamere

5.2. LOKALIZACIJA VRHA

Generalni princip rada ovog dijela algoritma temelji se na prikupljanju intenziteta piksela u oba smjera pronađene konture strelice. Zatim se vrši usporedba pojedinih intenziteta piksela te se, ovisno o rezultatu, reagira na prikladan način. U slučaju zadovoljavanja uvjeta razlike intenziteta dvaju piksela, obje se točke spremaju u pripadajuće liste. Nakon prolaska kroz cijelu konturu, uzimaju se medijan vrijednosti obje liste i vrši se usporedba prikupljenih vrijednosti. Na temelju tih informacija se odlučuje o smjeru protezanja konture. Strelice koje se koriste u ovom radu imaju namjerne specifičnosti glede odabira boja. Naime, jedan dio strelice je žute boje i predstavlja svjetliju stranu cjelokupne strelice, dok je s druge strane tamniji dio strelice koji ujedno sadrži i vrh koji je potrebno pronaći.

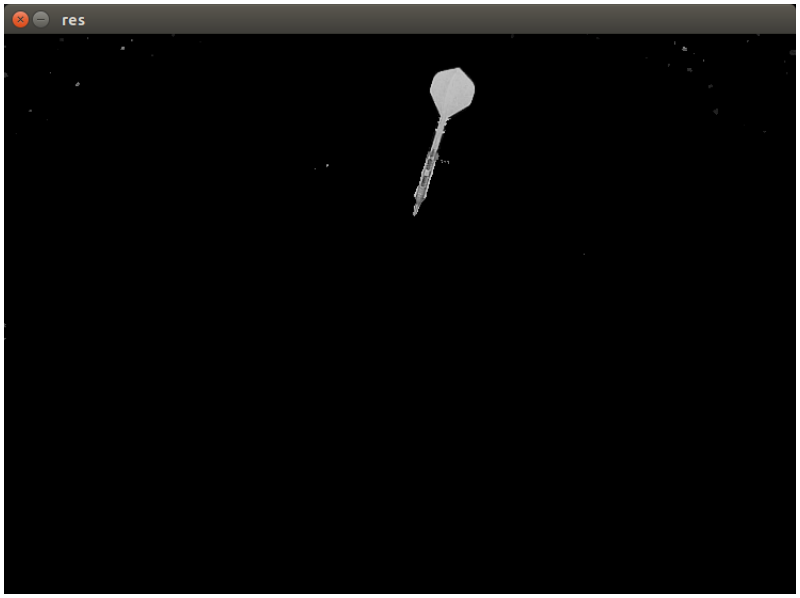
Slika 5.5. Izgled korištenih strelica pri izradi rada

Osnovna je ideja na temelju razlika u bojama i intenzitetima piksela duž strelice pronaći smjer u kojem je poželjno tražiti vrh strelice. Nakon pronalaska smjera, potrebno je pronaći vrh strelice u tom dijelu konture. U ovom se slučaju koristi pristup potrage za onim pikselom koji se nalazi na toj strani konture, a koji je najdalji od centra konture.

```
scale2list = np.linspace(0.7,1.3,65)
def findpoint(img,nodart,dart,boxmask,contlist, cammatrix, flagnum, thresh):
    scale = 0
    intdiff = 30
    intlist1 = []; intlist2 = []
    contthresh, _ = cv.findContours(thresh, cv.RETR_EXTERNAL,cv.CHAIN_APPROX_NONE)
    cv.drawContours(img,contthresh,-1, (255,0,0), 1)
    for i,c in enumerate(contlist):
        datapts = np.empty((len(c),2), dtype=np.float64) #type: ignore
        for k in range(datapts.shape[0]):
            datapts[k,0] = c[k,0,0]
            datapts[k,1] = c[k,0,1]
        mean, eigenvectors, eigenvalues = cv.PCACompute2(datapts, np.empty((0)))
        cntr = (int(mean[0,0]), int(mean[0,1]))
        cv.circle(img,cntr,1,(255,0,255),2)

        contmask = np.zeros_like(dart)
        cv.drawContours(contmask,contthresh,-1, (255,255,255),-1)

        res = cv.bitwise_and(dart,contmask)
        res = cv.cvtColor(res,cv.COLOR_BGR2GRAY)
```

Prije ulaska u strukturu funkcije *findpoint*, definirana je lista naziva *scale2list* koja je potrebna za pronalazak vrha strelice, a čiji će značaj biti objašnjen kasnije. Vidljivo je da su svi ulazni argumenti ove funkcije jednaki izlaznim argumentima prethodno objašnjene funkcije. Prije svega, definiraju se varijable i liste koje se koriste u radu ovog dijela koda. Prvo je definirana varijabla *scale* koja označava faktor koji određuje veličinu generiranog vektora s pomoću informacija funkcije *cv.PCACompute2*. Zatim je definirana varijabla *intdiff* koja predstavlja *thresh* vrijednost koju je potrebno zadovoljiti kako bi se izvršio određeni dio koda. U suštini razlika intenziteta dvaju piksela mora biti veća od ove vrijednosti kako bi se te točke uzele u obzir u daljnjem izračunu. Slijedi definiranje lista u koje se spremaju prethodno spomenute točke. Nakon pripreme varijabli i lista, slijedi pronalazak konture strelice na *thresh* slici. Vrlo je važno naglasiti kako se ovdje koristi neobrađena slika konture (prije dilatacije i sl.) kako bi informacije o vrhu strelice bile preciznije pronađene. Kontura se zatim iscrtava na sliku *img*. Naredni dio koda odnosi se na prethodno spomenutu funkciju *cv.PCACompute2*.

Ta funkcija predstavlja matematičku metodu analize glavnih komponenata. Glavni razlog korištenja te funkcije je mogućnost redukcije dimenzionalnosti seta podataka. U ovom slučaju je cilj smanjiti dimenzionalnost seta 2D točaka koje sačinjavaju pronađenu konturu strelice. Kako bi se to dogodilo, potrebno je te točke aproksimirati takvim pravcem koji najbolje opisuje razdiobu tih točaka. Tako je 2D set točaka sveden na 1D pravac. Osim prethodno objašnjenje mogućnosti, ova funkcija daje mogućnost pronalaska smjera u kojem podaci najviše variraju. Tako je moguće prikupiti svojstvene vektore koji predstavljaju varijancu podataka u dva različita smjera. Ti se vektori nazivaju principijelnim komponentama seta podataka.[19] U ovom su radu te komponente iskorištene u svrhu prikupljanja piksela duž prethodno generiranog pravca za proces određivanja smjera tamnijeg dijela strelice. Ulaskom u *for* petlju generira se matrica veličine 2 stupca s onolikim brojem redaka koliko je i točaka konture. Nakon toga se *x* i *y* točke konture spremaju u prethodno stvorenu matricu kako bi se generirao takav zapis koji je pogodan kao ulazni argument funkcije *cv.PCACompute2*. Nakon pripreme podataka, slijedi poziv glavne funkcije koja kao rezultat daje srednju vrijednost analiziranog seta podataka te principijelne komponente smjerova varijacije podataka. U varijablu *cntr* sprema se centar seta točaka, odnosno konture strelice. Varijabla *contmask* predstavlja crnu sliku dimenzija slike u kojoj se nalazi strelica, a u koju se iscrtava pronađena kontura strelice. Nakon toga se vrši poziv *cv.bitwise_and* funkcije koja omogućuje izuzimanje regije interesa na kojoj će se vršiti potraga za smjerom strelice. Odabrana regija je vidljiva na sljedećoj slici.



Slika 5.6. Odabrano područje strelice za daljnju analizu

```
        while True:
            p1 = (cntr[0] - scale*eigenvectors[0,0]*eigenvalues[0,0], cntr[1] -
scale*eigenvectors[0,1]*eigenvalues[0,0])
            p2 = (cntr[0] + scale*eigenvectors[0,0]*eigenvalues[0,0], cntr[1] +
scale*eigenvectors[0,1]*eigenvalues[0,0])
            x1 = int(p1[0]); y1 = int(p1[1]); x2 = int(p2[0]); y2 = int(p2[1])
            pt1 = (x1,y1); pt2 = (x2,y2)
            int1 = int(res[y1,x1]); int2 = int(res[y2,x2])
            ptcont1 = cv.pointPolygonTest(contlist[0], pt1, False)
            ptcont2 = cv.pointPolygonTest(contlist[0], pt2, False)

            if ptcont1 == -1 and ptcont2 == -1:
                intlist1 = sorted(intlist1); intlist2 = sorted(intlist2)
                mid1 = intlist1[len(intlist1) // 2]; mid2 = intlist2[len(intlist2) // 2]
                if mid1 > mid2:
                    pointlist = []
                    for scale2 in scale2list:
                        scale = 0
                        groundval = [255,255,255]
                        prevval = (0,0,0); currpos = 0; prevpos = 0
                        valcount = 0
                        while True:
                            p2 = (cntr[0] +
scale*scale2*eigenvectors[0,0]*eigenvalues[0,0], cntr[1] +
scale*eigenvectors[0,1]*eigenvalues[0,0])
                            x2 = int(p2[0]); y2 = int(p2[1]) # p2
                            val = boxmask[y2,x2]
                            if val[0] != groundval[0]:
                                if val[0] != prevval[0]:
                                    currpos = prevpos
                                valcount += 1
                                if valcount == 50:
                                    pointlist.append(currpos)
                                    break
                            prevval = boxmask[y2,x2]
                            prevpos = (x2,y2)
                            scale += 1e-4

                    prevdist = 0
                    for point in pointlist:
                        dist = ((point[0]-cntr[0]) ** 2 + ((point[1])-cntr[1]) ** 2) ** 0.5
                        if dist > prevdist:
                            prevdist = dist
                            finalpoint = point
                    break
```

Prethodni je dio koda cjelokupni prikaz procesa pronalaska tamnijeg dijela strelice, a zatim i točke vrha strelice. Nakon što se izvrši priprema slike na kojoj će se tražiti vrh strelice, ulazi se u *while* petlju koja staje u trenutku pronalaska točke vrha strelice. Varijable *p1* i *p2* omogućuju izračun točaka koje se nalaze na prethodno generiranom pravcu, s time da je početna pozicija centar konture. Dakle, s povećanjem faktora *scale* povećava se i udaljenost točke od centra. U ovom je slučaju cilj postepeno uzimati cijeli niz točaka konture počevši od njenog centra, sve dok se ne dođe do kraja konture na njena oba kraja. Tako su obuhvaćene sve točke konture za koje se vrši izračun intenziteta koji određuje svjetlinu tog dijela strelice. Nakon obuhvaćanja pojedine točke, vrši se izračun intenziteta piksela u tim koordinatama na slici *res.* Zatim se poziva funkcija *cv.pointPolygonTest* koja provjerava da li se određena točka nalazi na nekoj konturi. U ovom se slučaju provjerava jesu li točke *pt1* i *pt2* na pronađenoj konturi strelice. Ovo predstavlja jednu od bitnijih mjera predostrožnosti kako se ne bi uzimale točke koje uopće nisu povezane s konturom strelice. Funkcija *cv.pointPolygonTest* kao rezultat daje -1 ako se točka ne nalazi na konturi, odnosno 0 ako se točka nalazi na njenom rubu te 1 ako se točka nalazi unutar konture. U trenutku kad se u oba smjera dosegne kraj konture, odnosno kad na oba kraja uzete točke više ne budu na konturi, ulazi se u *if* petlju gdje se prvo sortiraju svi spremljeni rezultati. Nužno je sortirati obje liste kako bi intenziteti piksela bili redom poredani i pripremljeni za izračun medijana za daljnju usporedbu. Medijan se uzima kao srednji broj svake sortirane liste. Nakon definiranja medijana svake liste, slijedi njihova usporedba. U slučaju da je *mid1* veći od *mid2*, to znači da je područje intenziteta piksela iz liste *intlist2* generalno slabijeg intenziteta (tamnije) od intenziteta piksela liste *intlist1*. Nakon toga je pažljivo potrebno provjeriti na koji su način računati pikseli za ispunu liste *intlist2* i zapamtiti tu formulu kako bi se kasnije tražio vrh strelice. Ovime je odabran smjer u kojem se vrši potraga za točkom vrha strelice. Slijedi ulazak u novu *for* petlju koja se koristi pri traženju vrha strelice. Generira se nova lista u koju se spremaju krajnje točke konture strelice. Koristi se dodatni faktor *scale2* koji omogućuje promjenu kuta pri traženju vrha strelice. Prije ulaska u *while* petlju, potrebno je definirati sve informacije potrebne za traženje vrha. Lista *groundval* predstavlja tzv. *ground truth* informaciju. Ta lista zapravo predstavlja bijeli piksel u slici. Ona se koristi kao alat s kojim se uspoređuje intenzitet piksela na binarnoj slici. *Prevval* predstavlja prošlu vrijednost intenziteta piksela točke u slici. Varijabla *currpos* predstavlja trenutnu točku, dok *valcount* predstavlja svojevrstan brojač koji određuje kraj potrage u određenom smjeru. Definiranje svih varijabli i lista određuje ulazak u *while* petlju koja je odgovorna za potragu.

Princip rada *while* petlje je sljedeći. Svaka vrijednost faktora *scale2* određuje određen smjer u kojem se algoritam kreće. Sa svakom promjenom faktora *scale2* mijenja se i pravac po kojem se traži vrh strelice. Dakle, tehnički se generira „lepeza" pravaca koji se protežu od centra konture prema tamnijem dijelu strelice u različitim smjerovima. Za svaki takav pravac je potrebno pronaći točku koja se nalazi na njemu, a koja je posljednja u dodiru s konturom strelice. Upravo se taj dio procesa odvija u *while* petlji. Ponovno se varijacijom faktora *scale* uzimaju točke koje se nalaze na tom pravcu, počevši od samog centra u smjeru protezanja pravca. Intenzitet svake točke tog pravca u slici *boxmask* se uspoređuje s *ground truth* vrijednošću. Pošto je *boxmask* binarna slika, postoje dvije vrijednosti koje se pojavljuju – 0 i 255. Vrijednost 255 označava dio slike u kojem se nalazi kontura strelice, dok 0 predstavlja prostor koji ne sadrži bitne informacije u ovom procesu. U trenutku kad se intenzitet trenutnog piksela razlikuje od vrijednosti 255, znači da ta točka ne pripada pronađenoj konturi strelice. Takva bi točka mogla predstavljati kraj konture u smjeru tog pravca. Ulazi se u *if* petlju gdje se odmah uspoređuje intenzitet trenutnog piksela s *ground truth* intenzitetom. Ako je intenzitet drugačiji, ponovno se vrši provjera trenutnog intenziteta piksela s prošlim intenzitetom. Na taj se način osigurava spremanje zadnjeg bijelog piksela. Vrši se inkrementiranje varijable *valcount* za 1. U trenutku kad varijabla *valcount* dostigne broj 50, staje se s procesom pronalaska posljednje točke na pojedinom pravcu u prethodno odabranom smjeru konture. Ovaj je dio koda tako napisan u slučaju da se je vrh strelice odvojio od ostatka konture. Moguće je da postoji bijeli piksel vrha koji nosi ključnu informaciju, a koji nije spojen s ostatkom konture strelice. Zato je postavljen broj 50 koji osigurava da je uzeto dovoljno točaka na tom pravcu koje su se udaljile od kraja konture čime je izbjegnuta problematika s potencijalnom izgubljenom informacijom preciznije pozicije točke vrha strelice. Nakon što se taj uvjet ispuni, zadnja točka konture na tom pravcu se dodaje u listu *pointlist*. Nakon prolaska kroz sve elemente liste *scale2list*, lista *pointlist* sadrži sve krajnje točke konture u različitim smjerovima. Zatim se odvija iteracija kroz sve spremljene točke te se računa udaljenost istih od centra konture. Ona točka koja se nalazi najdalje od centra konture je vrh strelice te točka koju se traži. U sljedećem je prikazu moguće vidjeti iscrtane točke koje prikazuju način određivanja smjera tamnije strane strelice te pronađenu točku vrha strelice.

Slika 5.7. Princip pronalaska tamnije strane konture i vrha strelice

Plavi pravokutnik predstavlja prethodno spomenuti skalirani pravokutnik koji određuje regiju interesa u kojoj se traži vrh strelice. Rozom je bojom označen centar konture, dok su gusto poredanim crvenim i zelenim točkicama označeni sve točke konture koje su spremljene u liste *intlist1* i *intlist2*. U konačnici je vidljiva crvena točkica na samom vrhu strelice koja predstavlja pronađen vrh od strane algoritma.

```
                else:
                    pointlist = []
                    for scale2 in scale2list:
                        scale = 0
                        groundval = [255,255,255]
                        prevval = (0,0,0); currpos = 0; prevpos = 0
                        valcount = 0
                        while True:
                            p1 = (cntr[0] -
scale*scale2*eigenvectors[0,0]*eigenvalues[0,0], cntr[1] -
scale*eigenvectors[0,1]*eigenvalues[0,0])
                            x1 = int(p1[0]); y1 = int(p1[1])
                            val = boxmask[y1,x1]
                            if val[0] != groundval[0]:
                                if val[0] != prevval[0]:
                                    currpos = prevpos
                                valcount += 1
```

```
                            if valcount == 50:
                                pointlist.append(currpos)
                                break
                        prevval = boxmask[y1,x1]
                        prevpos = (x1,y1)
                        scale += 1e-4

                prevdist = 0
                for point in pointlist:
                    dist = ((point[0]-cntr[0]) ** 2 + ((point[1])-cntr[1]) ** 2) ** 0.5
                    if dist > prevdist:
                        prevdist = dist
                        finalpoint = point
                break
```

Prethodni kod predstavlja istu logiku koja je već objašnjena. Razlika je samo u odabranom smjeru gdje je potrebno paziti na definiranje formule za generiranje točaka čiji će se intezitet potraživati kako bi se odredio tamniji smjer strelice.

```
            elif abs(int1-int2) > intdiff and 50 < int1 < 220 and 50 < int2 < 220:
                if not ptcont1 == -1 and not ptcont2 == -1:
                    intlist1.append(int1)
                    intlist2.append(int2)
                    cv.circle(img, pt1, 1, (0,0,255),1)
                    cv.circle(img, pt2, 1, (0,255,0),1)
            scale += 1e-4
    nodart = cv.warpPerspective(nodart,cammatrix,dsize = (950,950), flags=cv.INTER_NEAREST)
    point = np.array([[finalpoint[0], finalpoint[1]]], dtype="float32") # type:ignore
    point = np.array([point])
    point2 = cv.perspectiveTransform(point, cammatrix)
    point2 = tuple((point2[0][0]))
    point = (int(point2[0]), int(point2[1]))

    if flagnum == 2:
        point = (950-point[0]-1,point[1])
        nodart = cv.flip(nodart,1)
    cv.circle(img, finalpoint, 1, (0,0,255),4)
    # cv.imshow("img", img)
    # cv.waitKey(0)
    # cv.destroyAllWindows()
    return point, nodart
```

Elif dio koda vezan je na onaj dio *if* koda koji se izvršava u slučaju kada se obje uzete točke ne nalaze na konturi. Dakle, ako taj uvjet nije ispunjen, ulazi se u ovaj dio koda ako su svi uvjeti zadovoljeni.

Prvi je uvjet da razlika između intenziteta piksela mora biti veća od vrijednosti varijable *intdiff* kako bi se izbjegle anomalije uzrokovane osvjetljenjem, odbljeskom i sl. Drugi je uvjet da intenzitet piksela ne bude manji od 50, odnosno veći od 220. Te su vrijednosti empirijski određene. Naime, pri pronalasku konture strelice često te konture ne budu idealno u skladu sa stvarnom konturom, već budu malo proširenije. To prošireno područje također ulazi u algoritam izračuna. Međutim, potrebno ga je maksimalno ukloniti kako se ti intenziteti piksela ne bi uzimali u obzir. Pošto je velik dio pikado mete crni ili bijeli, na ovaj način se isključuju oni scenariji u kojima su ti dijelovi mete obuhvaćeni u pronađenoj konturi strelice. Nakon ispunjenja prethodna dva uvjeta, *if* petljom se provjerava pripadnost točke konturi strelice. Ako je taj uvjet ispunjen, točke se pridružuju pripadajućim listama čiji se podaci kasnije obrađuju. Naredbom *cv.circle* se iscrtavaju sve one točkice vidljive na slici 5.6. Pronalazak točke vrha strelice iziskuje proces primjene matrice homografije na tu točku kako bi se pronašle koordinate te točke u ispravljenoj perspektivi slike kamere. To se čini pozivom funkcije *cv.perspectiveTransform* koja omogućuje pronalazak koordinata točke u destinacijskoj slici uz poznavanje koordinata točke u izvornoj slici te pripadajuće matrice homografije. Varijabla *point* označava koordinate točke u odredišnoj slici. Posljednja *if* petlja je važna zbog prethodno spomenutog problema s trećom kamerom i njenom obrnutom pogledu. Ovako se osigurava da sva tri rezultata ispravljanja perspektiva kamera budu ekvivalentna. U konačnici je moguće iscrtati pronađen vrh strelice u ispravljenoj perspektivi slike kamere. Time je završen proces pronalaska vrha strelice.

6. INTEGRACIJA PROGRAMSKOG RJEŠENJA

Za *real-time* međudjelovanje prethodnih algoritama potrebno je napisati algoritam koji objedinjuje sve napisane algoritme. Zbog toga su stvorena dodatna dva programska koda.

```
import Tkinter as tk #type: ignore
import cv2 as cv
import subprocess, pickle
from tip_recognize import extractdart, findpoint, getscore
from frames import Scenechange
from multiprocessing import Process, Manager
from threading import Thread

def updatescore(entry1, entry2, entry3, entry_total):
    try:
        total_score = int(entry1.get()) + int(entry2.get()) + int(entry3.get())
        entry_total.delete(0, tk.END)
        entry_total.insert(0, str(total_score))
    except ValueError:
        pass

def process_camera(cam, cammatrix, warpmatrix, flagnum, scorelist):
    runcam = Scenechange(cam)

    while True:
        try:
            nodart, dart = runcam.detect_scene_change()
            img, nodart, dart, boxmask, contlist, thresh = extractdart(nodart, dart,
cammatrix)
            point, nodart = findpoint(img, nodart, dart, boxmask, contlist,
warpmatrix,flagnum, thresh)
            score = getscore(point, nodart, flagnum)
        except IndexError:
            print("Dart contour is incomplete - score set to zero.")
            score = 0
            cv.destroyAllWindows()
        except TypeError:
            print("Can't find the impact point - score set to zero.")
            score = 0
            cv.destroyAllWindows()
        except:
            print("Something else is wrong.")
            score = 0
        scorelist.append(score)
```

Definirana funkcija *updatescore* koristi se kao *event* u slučaju potrebe za ispravljanjem rezultata. Ulazni argumenti su tri dobivena rezultata pri čemu je svaki rezultat (postignuti bodovi jedne strelice) dobiven glasovanjem triju kamera. Posljednji argument predstavlja zbroj triju rezultata. Dakle, ako rezultat bude krivo izračunat, korisnik može ručno promijeniti taj rezultat i pritiskom na tipku *Enter* ažurirati ukupno postignut rezultat. Iduća je funkcija ključna u radu cijelog sustava. Ulazni argumenti su svi oni parametri koji su specifični za pojedinu kameru. Ideja je koristiti *multiprocessing* biblioteku koja omogućuje pozivanje iste funkcije s različitim argumentima. Sve se funkcije zatim zasebno paralelno izvršavaju čime se omogućuje korištenje sve tri kamere odjednom. Pozivom ove funkcije, svaka kamera ulazi u *while* petlju koja se konstantno izvršava do prekida programa od strane korisnika. U prvoj se liniji tog dijela koda čeka na detekciju promjene u kadru pojedine kamere. U trenutku kad je promjena detektirana, odnosno kad je strelica upala u metu, kreće se s pozivanjem svakog prethodno objašnjenog dijela koda. U slučaju pojave grešaka, varijabli *score* se pridodaje vrijednost 0 uz ispis poruke koja upozorava na pogrešku pri radu algoritma. Varijabla *score* se zatim dodaje u listu *scorelist* koja sadrži podatke o dodijeljenim bodovima svake kamere za jednu bačenu strelicu.

```
def update_gui(entry1, entry2, entry3, entry_total, scorelist):
    entrylist = [entry1, entry2, entry3]
    numen = 0
    total_score = 0
    while True:
        if len(scorelist) == 3:
            print(scorelist)
            score = max(set(scorelist), key=scorelist.count)
            print(score)
            scorelist[:] = []
            entrylist[numen].delete(0, tk.END)
            entrylist[numen].insert(0, str(score))

            total_score += score

            numen += 1

            if numen == 3:
                entry_total.delete(0, tk.END)
                entry_total.insert(0, str(total_score))
                total_score = 0
                numen = 0
```

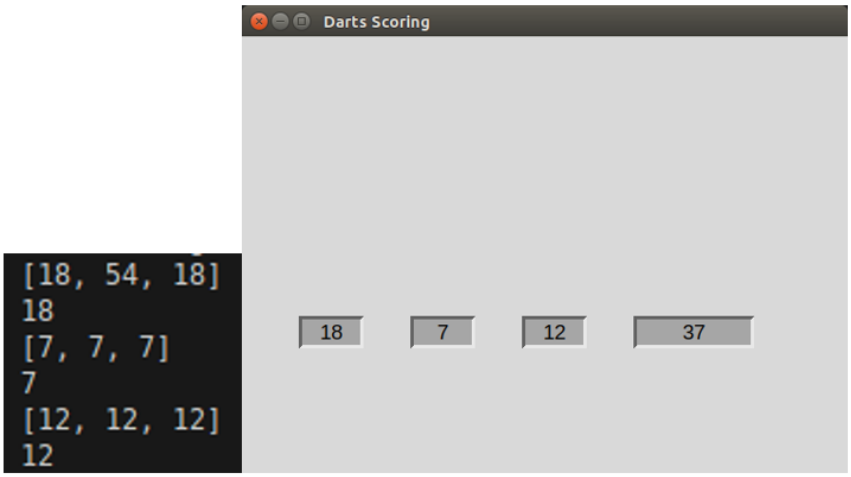
Funkcija *update_gui* je odgovorna za automatski upis rezultata dobivenog glasovanjem triju kamera. Dakle, u trenutku kad sve tri kamere daju svoj glas, duljina liste *scorelist* je 3. Tada je potrebno uzeti onaj broj koji se najviše puta ponavlja u toj listi. U konačnici je potrebno odabrani broj upisati u pojedino mjesto unutar *tkinter* sučelja. Kada varijabla *numen* bude vrijednosti 3, znači da su sve tri strelice bačene te da je potrebno upisati ukupan rezultat i resetirati varijable za bacanje novih triju strelica.

```
def main():
    manager = Manager()
    scorelist = manager.list()
    width = 800
    height = 600
    framerate = "10/1"

    subprocess.call(["v4l2-ctl", "-d","/dev/video0", "--set-ctrl=exposure_auto=1"])
    subprocess.call(["v4l2-ctl", "-d","/dev/video0", "--set-ctrl=exposure_absolute=31"])
    subprocess.call(["v4l2-ctl", "-d", "/dev/video0", "--set-ctrl=gamma=112"])
    subprocess.call(["v4l2-ctl", "-d", "/dev/video0", "--set-ctrl=sharpness=4"])

    subprocess.call(["v4l2-ctl", "-d","/dev/video1", "--set-ctrl=exposure_auto=1"])
    subprocess.call(["v4l2-ctl", "-d","/dev/video1", "--set-ctrl=exposure_absolute=29"])
    subprocess.call(["v4l2-ctl", "-d", "/dev/video1", "--set-ctrl=gamma=112"])
    subprocess.call(["v4l2-ctl", "-d", "/dev/video1", "--set-ctrl=sharpness=4"])

    subprocess.call(["v4l2-ctl", "-d","/dev/video2", "--set-ctrl=exposure_auto=1"])
    subprocess.call(["v4l2-ctl", "-d","/dev/video2", "--set-ctrl=exposure_absolute=30"])
    subprocess.call(["v4l2-ctl", "-d", "/dev/video2", "--set-ctrl=gamma=112"])
    subprocess.call(["v4l2-ctl", "-d", "/dev/video2", "--set-ctrl=sharpness=4"])

    cam1 = (
        "v4l2src device=/dev/video0 ! "
        "image/jpeg, width={}, height={}, framerate={} ! "
        "jpegdec ! videoconvert ! videobalance brightness=0.01 contrast=0.89
saturation=1.34 ! appsink drop=1"
        ).format(width, height, framerate)

    cam2 = (
        "v4l2src device=/dev/video1 ! "
        "image/jpeg, width={}, height={}, framerate={} ! "
        "jpegdec ! videoconvert ! videobalance brightness=-0.01 contrast=0.87
saturation=1.35 ! appsink drop=1"
        ).format(width, height, framerate)
```

```
    cam3 = (
        "v4l2src device=/dev/video2 ! "
        "image/jpeg, width={}, height={}, framerate={} ! "
        "jpegdec ! videoconvert ! videobalance brightness=0.01 contrast=0.87
saturation=1.35 ! appsink drop=1"
        ).format(width, height, framerate)

    camlist = [cam1, cam2, cam3]
    processes = []; cammatrices = []; warpmatrices = []; flag = []

    for num in range(0,3):
        file = open
("/home/fsb/Desktop/diplomski_opencv/intrinsic_calibration/calibration_cam{}.pkl".format(nu
m), "rb")
        cammatrix = pickle.load(file)
        file.close()
        cammatrices.append(cammatrix)

        file =
open("/home/fsb/Desktop/diplomski_opencv/tkinter/matrixwarp{}.pkl".format(num), "rb")
        warp = pickle.load(file)
        file.close()
        warpmatrices.append(warp)

        flag.append(num)

    for cam, matrix, warpmatrix, flagnum in zip (camlist, cammatrices, warpmatrices, flag):
        p = Process(target=process_camera, args=(cam, matrix, warpmatrix, flagnum,
scorelist,))
        p.daemon = True
        p.start()
        processes.append(p)

    root = tk.Tk()
    root.title("Darts Scoring")

    entry1 = tk.Entry(root, width=5, bg="gray65", bd=3, justify="center", font=("Arial",
14))
    entry1.place(x=50, y=250)
    entry2 = tk.Entry(root, width=5, bg="gray65", bd=3, justify="center", font=("Arial",
14))
    entry2.place(x=150, y=250)
```

```
    entry3 = tk.Entry(root, width=5, bg="gray65", bd=3, justify="center", font=("Arial",
14))
    entry3.place(x=250, y=250)

    entry_total = tk.Entry(root, width=10, bg="gray65", bd=3, justify="center",
font=("Arial", 14))
    entry_total.place(x=350, y=250)

    entry1.bind('<Return>', lambda event: updatescore(entry1, entry2, entry3, entry_total))
    entry2.bind('<Return>', lambda event: updatescore(entry1, entry2, entry3, entry_total))
    entry3.bind('<Return>', lambda event: updatescore(entry1, entry2, entry3, entry_total))

    gui_thread = Thread(target=update_gui, args=(entry1, entry2, entry3, entry_total,
scorelist))
    gui_thread.daemon = True
    gui_thread.start()

    root.mainloop()

if __name__ == "__main__":
    main()
```

U *main* funkciji je definiran glavni dio koda u kojem se prvo definiraju postavke svake kamere kako bi se osigurali optimalni uvjeti za specifičnost zadatka. Nakon toga se definiraju liste u koje se spremaju parametri za svaku kameru. U listu *cammatrices* se sprema kalibracijska matrica koja sadrži podatke o intrinzičnoj matrici te o distorzijskim koeficijenata svake kamere. U listu *warpmatrices* se spremaju matrice homografije svake kamere. U listu *flag* se sprema broj kamere. Nakon toga se ulazi u *for* petlju koja postepeno pokreće svaku kameru odgovarajućim redoslijedom. Ispunom svih lista može se pristupiti pokretanju triju paralelnih procesa koji izvršavaju proces bodovanja rezultata. Naredbom *Process* moguće je pokrenuti zaseban proces (izvršavanje funkcije *process_camera*) uz argumente koji su specifični za pojedinu kameru. U konačnici se stvara maleno *tkinter* sučelje koje služi kao vizualna reprezentacija postupka.



Slika 6.1. a) pojedinačno glasovanje kamera za definiranje ukupnog rezultata, b) prikaz u *tkinter* sučelju

```
import cv2 as cv
import keyboard as key # type: ignore

class Scenechange:
    def __init__(self, camera):
        self.cap = cv.VideoCapture(camera, cv.CAP_GSTREAMER)
        self.prev_frame = None; self.refframe = None; self.frame = None
        self.flag = True; self.flag2 = True; self.newread = False; self.thrown = 0

    def detect_scene_change(self):
        if self.newread:
            self.prev_frame = None; self.refframe = None; self.frame = None
            self.flag = True; self.flag2 = True

        while True:
            if self.thrown == 3:
                key.wait("space")
                print("\nNext player.")
                for _ in range(10): # buffer remove pictures
                    ret, self.frame = self.cap.read()
                self.thrown = 0
            ret, self.frame = self.cap.read()

            if not ret:
                print("Error: Unable to capture frame.")
                break

            if self.prev_frame is not None:
                diff = cv.absdiff(self.frame, self.prev_frame)
```

```
                gray_diff = cv.cvtColor(diff, cv.COLOR_BGR2GRAY)
                _, thresh = cv.threshold(gray_diff, 21, 255, cv.THRESH_BINARY)
                contours, _ = cv.findContours(thresh, cv.RETR_EXTERNAL,
cv.CHAIN_APPROX_SIMPLE)
                largest = max(contours, key=cv.contourArea)
                if 100 < cv.contourArea(largest):
                    if self.flag:
                        self.refframe = self.prev_frame
                        self.flag = False
                        self.flag2 = False
                elif self.flag2 == False:
                    self.newread = True
                    for _ in range(10):
                        ret, self.frame = self.cap.read()
                    self.thrown  += 1
                    return self.refframe, self.frame

            self.prev_frame = self.frame

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

Drugi programski kod je odgovaran za detekciju promjena u kadru kamera. Pokretanjem funkcije *process_camera* odvija se nasljeđivanje klase *Scenechange* koja automatski pokreće rad kamere. Definiraju se varijable i *flag*-ovi koji omogućuju upravljanje procesom detekcije promjene scene. Iduća definirana funkcija sadrži logiku detekcije promjene u kadru kamere. Pri prvom se ulasku u funkciju provjera stanje *flag* varijable *self.newread*. U prvom je pokretanju kamere ta varijabla jednaka vrijednosti *False*, što znači da se ne ulazi u tu petlju. Pri idućoj bačenoj strelici je ta varijabla jednaka *True* te je stoga potrebno resetirati sve varijable i *flag*-ove. U tom se slučaju ulazi u tu petlju kako bi se taj proces uspješno odradio. Sljedeća je bitna stavka *while* petlja. Prva *if* petlja se izvršava u trenutku kad su bačene sve tri strelice. Program staje s izvršavanjem dok korisnik ne izvadi sve tri strelice kako ne bi ometao rad algoritma, a zatim pritiskom tipke *space* pokreće daljnji rad. *For* petlja u tom dijelu programa je ključna kako bi se riješio problem nakupljanja slika koje su u *buffer* memoriji kamere. Na taj se način čisti ta memorija i omogućuje sigurnost pri očitanju novih promjena u kadru slike. Resetira se varijabla brojanja bačenih strelica te se očitava novi kadar kamere. U trenutku kad *self.prev_frame* varijabla poprimi vrijednost, ulazi se u glavni dio koda koji omogućuje detekciju promjene. U tu se petlju konstantno ulazi, a promjenom se smatra ona situacija u kojoj je pronađena kontura u kadru veća od površine veličine 100.

Na taj su način eliminirane malene promjene uslijed nesavršenosti izrade kamera, sitnih pomaka u kadru i slično. Prva *if* petlja omogućava spremanje referentnog kadra. U trenutku kad se pojavi promjena u kadru, ona se bilježi cijelo vrijeme dok se strelica ne upikne u metu. Zbog toga je nužno osigurati takav referentan kadar koji ne sadrži bačenu strelicu (nadolazeću promjenu) u svom kadru. Upravo zato se kao *self.refframe* uzima prvi prethodni kadar koji još ne sadrži strelicu. Tako je osiguran dobar rad funkcije *cv.absdiff* za izdvajanje konture strelice. Nakon detektirane promjene konstantno se ulazi u prethodno objašnjenu *if* petlju sve dok površina konture ne bude manja od 100. U tom trenutku se ulazi u *elif* dio koda koji označava stacioniranje strelice u meti. Radi sigurnosti pozicioniranja strelice u meti, ponovno se uzimaju dodatna 2-3 očitanja kadra kako bi konačan kadar sadržavao mirnu strelicu upiknutu u metu. U konačnici se ta dva kadra vraćaju kao rezultat izvršavanja koji se dalje koristi u izdvajanju konture strelice.

7. EVALUACIJA ALGORITMA

Kako bi se provjerila točnost i rad algoritma, proveden je postupak evaluacije. Provedeno je 7 različitih testiranja s pojedinačnim uzorkom od 100 bačenih strelica. Ključan uvjet u provođenju testiranja bio je bacanje strelica u razna polja pikado mete. Tako su se ispitala prethodno spomenuta *single*, *double* i *triple* područja. Dobiveni rezultati prikazani su u tablici ispod.

Tablica 7.1. Rezultati provedbe testiranja

TESTIRANJE	**TOČNOST** [%]
1.	97
2.	97
3.	98
4.	99
5.	96
6.	99
7.	96

Može se zaključiti kako su dobiveni rezultati zadovoljavajući. Prvotni je cilj bio postizanje točnosti od minimalno 90 %, što je uspješno ostvareno. Ako se sva testiranja uzmu u obzir, u prosjeku je postignuta točnost 97,43 %, odnosno ukupno je uspješno detektirano 682 strelice. Dodatnom je analizom utvrđeno kako su neuspješno detektirane strelice generalno posljedica obrade slike pri izdvajanju konture strelice. Glavni je problem izdvajanje konture strelice koja sadrži dobro izdvojen vrh koji se kasnije koristi za lokalizaciju vrha. Od veće bi pomoći bilo korištenje jačeg i kvalitetnijeg osvjetljenja. Osim toga, ponekad se pojavljuje problem izdvajanja krive konture kao prepoznate strelice. Za te bi potrebe bilo korisno osmisliti dodatne uvjete na temelju kojih bi se vršila segmentacija konture. Još jedan nedostatak ovog rada je problem sa strelicama u jako uskom području. Ako se prva strelica baci u polje *triple* 20, a zatim druga upadne u to isto polje uz pomicanje prethodne strelice, može doći do greške u radu algoritma. Razlog tomu je što se postojeći algoritam temelji na razlikama između slika. Stoga bi u prethodnom scenariju pomak već bodovane strelice rezultirao detektiranim pomakom koji najčešće negativno utječe na segmentaciju strelice koju je potrebno analizirati.

8. ZAKLJUČAK

U ovom se radu koriste 3 kamere međusobno jednako razmještene na 120 stupnjeva oko pikado mete kako bi sve pozicije bile dobro pokrivene. Isprintan je držač za LED traku kako bi osvjetljenje bilo stabilno što je vrlo važan faktor u izradi vizijskih sustava. Što se tiče softverskog dijela, isti se sastoji od više koraka. Prvo je izvršena intrinzična kalibracija kamera kako bi se ispravile distorzije leće. Nadalje je provedena ekstrakcija strelice te pronalaženje vrha korištenjem raznih metoda obrade slike. U konačnici je proveden proces homografije pronađenog vrha i određivanje pozicije istog kako bi se vršilo pravilno bodovanje postignutog rezultata. Nakon uspješno osiguranog pojedinačnog rada kamere, uspješno je osmišljena integracija triju kamera za paralelno izvođenje prethodno spomenutih algoritama. U izradi softverskog dijela koristilo se više različitih biblioteka, od kojih su najvažnije:

- *OpenCV* - biblioteka koja sadrži brojne funkcije u svrhu obrade i manipulacije slika/videa
- *gstreamer* – biblioteka važna za pokretanje kamera i osiguravanje primijene specifičnih parametara za svaku kameru (ekspozicija, kontrast, zasićenost i sl.)

U konačnici je prikazana evaluacija dobivenih rezultata u vidu ispitivanja točnosti sustava. Istaknuti su glavni nedostaci na kojima bi se moglo poraditi radi postizanja boljih rezultata. Ovaj bi se rad dodatno mogao unaprijediti razvitkom kvalitetnijeg *GUI* sučelja koje bi omogućilo igranje igre. Također bi bilo korisno osmisliti algoritam koji bi omogućio automatsku kalibraciju mete pritiskom na tipku. U tom se procesu prvenstveno misli na pronalazak centra mete i referentne točke kako bi definirane granice polja bile precizno određene i usklađene sa stvarnima. U konačnici bi se postojeći algoritam mogao dodatno unaprijediti solucijom za problematiku pomaka strelica koje su već bodovane, a koje su promijenile položaj uslijed bačene strelice.

Napomena: u literaturi je moguće pronaći poveznicu na *github* repozitorij rada.

LITERATURA

[1] White Shark Owl GWC-004, pristup: rujan 2023.
https://www.robertdyas.co.uk/white-shark-owl-gwc-004-1080p-webcam-black

[2] NVIDIA Jetson Nano Developer Kit, pristup: rujan 2023.
https://www.nvidia.com/en-us/autonomous-machines/embedded-systems/jetson-nano-developer-kit/

[3] 5 reasons why the One80 Gladiator 3 is the best dartboard, pristup: rujan 2023. https://www.eaglesports.com.au/blogs/news/2018/Jul/21/gladiator3

[4] Dartboard LED ring, pristup: rujan 2023.
https://www.thingiverse.com/thing:4112111

[5] Jetson Nano Case – Connector's edition, pristup: rujan 2023.
https://www.thingiverse.com/thing:3603594

[6] Camera positioning, Autodarts Docs, pristup: rujan 2023.
https://docs.autodarts.io/getting-started/camera-positioning/

[7] Shree Nayar: First Principles of Computer Vision - Types of Image Sensors, School of Engineering and Applied Sciences, Columbia University, New York, 2021., pristup: travanj 2024.
https://fpcv.cs.columbia.edu/Monographs

[8] Understanding the digital image sensor, pristup: travanj 2024.
https://thinklucid.com/tech-briefs/understanding-digital-image-sensors/

[9] Shree Nayar: First Principles of Computer Vision – Linear Camera Model, School of Engineering and Applied Sciences, Columbia University, New York, 2021., pristup: travanj 2024.
https://fpcv.cs.columbia.edu/Monographs

[10] What is Camera Calibration?, pristup: travanj 2024.
https://www.mathworks.com/help/vision/ug/camera-calibration.html

[11] Camera Modeling: Exploring Distortion and Distortion Models, Part I, pristup: travanj 2024. https://www.tangramvision.com/blog/camera-modeling-exploring-distortion-and-distortion-models-part-i#tangential-de-centering-distortions

[12] Calibration Patterns Explained, pristup: travanj 2024. https://calib.io/blogs/knowledge-base/calibration-patterns-explained

[13] Basic concepts of the homography explained with code, pristup: travanj 2024. https://docs.opencv.org/4.x/d9/dab/tutorial_homography.html

[14] CMU School of Computer Science: Image homographies, Pittsburgh University, Pennsylvania, 2021., pristup: travanj 2024. http://16385.courses.cs.cmu.edu/spring2021content/lectures/08_homographies/08_homographies_slides.pdf

[15] Shree Nayar: First Principles of Computer Vision – 2x2 Image Transformations – Image Stitching, School of Engineering and Applied Sciences, Columbia University, New York, 2021., pristup: travanj 2024. https://fpcv.cs.columbia.edu/Monographs

[16] Shree Nayar: First Principles of Computer Vision – 3x3 Image Transformations – Image Stitching, School of Engineering and Applied Sciences, Columbia University, New York, 2021., pristup: travanj 2024. https://fpcv.cs.columbia.edu/Monographs

[17] Shree Nayar: First Principles of Computer Vision – Computing Homography – Image Stitching, School of Engineering and Applied Sciences, Columbia University, New York, 2021., pristup: travanj 2024. https://fpcv.cs.columbia.edu/Monographs

[18] Hough Circle Transform, pristup: travanj 2024. https://docs.opencv.org/4.x/da/d53/tutorial_py_houghcircles.html

[19] Introduction to Principal Component Analysis (PCA), pristup: travanj 2024. https://docs.opencv.org/4.x/d1/dee/tutorial_introduction_to_pca.html

[20] GITHUB repozitorij – poveznica na *YouTube* video i algoritmi https://github.com/jcvetic/diplomski